www.Linuxidc.com

Second Edition

Programming in Lua

Lua 程序设计（第2版）

[巴西]Roberto Ierusalimschy 著
周惟迪 译
飞思科技产品研发中心 监制

Lua

2nd EDITION

Lua领域最权威的书籍之一

- 全面展示Lua 5.1的新特性
- 详细阐释Lua程序设计的高级话题
- 注重实战，使你迅速掌握Lua程序的精髓
- 内容精要，将Lua程序设计思想发挥到极致

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Lua 程序设计(第2版)

Programming in Lua Second Edition

本书是《Lua程序设计》（第1版）的更新和扩展，在第1版的基础上进行了较大的改进，并且加入了很多新的知识点。

- 介绍了Lua语言所具有的功能，并使用大量示例来演示如何将它们运用到实际的任务中
- 深入地介绍了Lua中唯一的数据结构——table，还讨论了数据结构、持久化、包和面向对象编程
- 展示了Lua的标准库，对那些想将Lua作为一门独立语言来使用的开发者特别有用，每一章介绍一个库，包括数学库、table库、字符串库、I/O库、操作系统库、调试库
- 介绍Lua与C语言之间的API，这是为那些想用C语言来访问Lua功能的人准备的

读者对象：

本书可以作为广大Lua爱好者的自学用书，也可以作为大学相关专业的教学参考书。

飞思在线：http://www.fecit.com.cn
飞思科技产品研发中心总策划

责任编辑：王树伟

责任美编：张　跃

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

内容简介

Title of the original edition: Programming in Lua:Second Edition

Copyright ©2006

Chinese translation copyright ©the 2008.

This Chinese edition was published by arrangement with Roberto Ierusalimschy, Rio de Janeiro.

本书中文版由 Roberto Ierusalimschy 授权电子工业出版社出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2006-5852

图书在版编目（CIP）数据

Lua 程序设计：第 2 版 /（巴西）莱鲁萨利姆斯奇（Ierusalimschy,R.）著；周惟迪译.

北京：电子工业出版社，2008.5

书名原文:Programming in Lua:Second Edition

ISBN 978-7-121-06187-5

I. L… II.①莱…②周… III.程序语言－程序设计 IV.TP312

中国版本图书馆 CIP 数据核字（2008）第 032375 号

责任编辑：王树伟
文字编辑：权　舆
印　　刷：北京智力达印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：18.75　字数：450 千字
印　　次：2008 年 5 月第 1 次印刷
印　　数：4 000 册　　定价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

前 言

1993 年 Waldemar、Luiz 和我刚开始开发 Lua，那时我们还很难想象 Lua 会发展成今天这个样子。刚开始时，Lua 只是作为一种内部语言服务于两个特定项目，而如今 Lua 被广泛地应用于各种领域，只要在需要一种简单、可扩展、可移植及高效的脚本语言的领域，都可以看到 Lua 的身影，例如嵌入式系统、移动设备、Web 服务器，当然还有游戏。

从一开始我们就将 Lua 设计成能与 C 语言或其他常用语言编写的软件相互集成。这种可集成的特点带来了许多好处。Lua 还是一门简单小巧的语言，之所以这么设计，一部分原因是因为它不准备去做那些 C 语言已经做得很好的方面。例如，C 语言高超的性能、对底层操作的能力及与第三方软件间的接口。Lua 依靠 C 语言来完成这些任务。而 Lua 提供的特性则是 C 语言所不太擅长的。例如，相对于硬件的高层抽象、动态结构、无冗余（No Redundancy）、简易的测试和调试。为此，Lua 还实现了一个安全的运行环境、一套自动内存管理机制、优秀的字符串处理能力和动态大小数据的处理功能。

Lua 的大部分功能来源于它的标准库，这也符合 Lua 的设计原则。因为 Lua 的主要特性就是它的可扩展性。语言中的许多特性都体现出了这点。动态类型为多态提供了支持。自动内存管理简化了接口，从而无须决定谁分配内存、谁释放内存及如何处理溢出等。高阶（Higher-order）函数和匿名函数允许实现更高层的参数化，能使函数变得更加通用。

Lua 除了是一门可扩展的语言外，还是一门“胶水语言”。Lua 支持一种基于组件的软件开发方法，这种方法可以通过黏合现有的高层组件来创建新的应用程序。这些组件可以是已编译好的，也可以是静态类型语言（如 C 或 C++）编写的。Lua 则可以成为组织和连接各种组件的胶水。通常，组件（或对象）表示了一个更具体、更底层的概念（例如窗口部件、数据结构），在程序开发过程中很少会修改它们，并且它们会占用最终程序的大部分 CPU 时间。Lua 则可以给出一个应用程序最终的样子，而这部分内容可能会在一个产品的开发周期中被反复修改。与其他胶水技术不同的是，Lua 是一门功能齐全的语言。从而使 Lua 不仅可用于黏合组件，还可用于适配组件或改造组件，甚至借此创建出一个全新的组件。

显然 Lua 不是唯一的脚本语言，还有许多其他语言都可以或多或少地解决同样的问题。但是 Lua 提供了一组特性，使它变得与众不同，成为解决许多问题的首选语言。

可扩展性：Lua 的可扩展性非常卓越，以至于许多人以为 Lua 不是一种编程语言，而是一种用于构建特定领域语言的工具包。Lua 从一开始就被设计为可扩展的，既可用 Lua 代码来扩展，又可以用外部的 C 代码来扩展。作为此概念的一个例证，Lua 的大部分基础功能就是通过外部库实现的。要将 Lua 与 C/C++相配接是非常容易的。此外，Lua 还可以集成到一些其他语言中，例如 Fortran、Java、Smalltalk、Ada、C#，甚至还可以集成到其他脚本语言中，如 Perl 和 Ruby。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

简易性：Lua 是一种简单、小巧的语言。它具有的概念不多，但每个概念都很有用。这样的简易性使 Lua 非常易于学习，同时还有利于减小 Lua 自身的大小。一个完整的 Lua 发布版本[1]可以很轻松地存放在一张软盘中。

高效：Lua 具有一个非常高效的实现。独立的评测结果显示 Lua 是脚本（解释型的）语言领域中最快的语言之一。

可移植性：当我们提及 Lua 的可移植性时，不仅是指 Lua 可以同时运行在 Windows 和 UNIX 平台上，而是指 Lua 可以运行在任何平台上，包括：PlayStation、XBox、Mac OS-9、OS X、BeOS、QUALCOMM Brew、MS-DOS、IBM 主机、RISC 操作系统、Symbian 操作系统、PalmOS、ARM 处理器、Rabbit 处理器、类 UNIX 或类 Windows 的系统。针对所有这些平台的源代码实质上都是同一套代码。Lua 并未使用条件编译来对不同平台进特殊处理，我们只依赖于 ANSI（ISO）C 标准来编写 Lua 的实现代码。这样就无须为了适应某个新平台而修改它。如果你有一个 ANSI C 编译器，那么只需编译 Lua 就可以了。

用户与读者

Lua 的用户一般可分为三大类：使用嵌入在某个应用程序中的 Lua 的用户、使用 Lua 解释器程序的用户、同时使用 Lua 和 C 的用户。

许多人在使用嵌入在某个应用程序中的 Lua，例如 CGILua[2]或某款游戏。这些程序都用 Lua 的 C API 来注册某些新函数、创建新类型，以及改变语言中某些操作的行为，并根据它们特定的需求来配置 Lua。通常这种程序的用户可能根本不知道这样一个事实，即 Lua 是一种独立的编程语言，只是被应用到了某个特定领域中。例如，CGILua 的用户往往认为 Lua 是一种专门为 Web 而设计的语言，而某些游戏的玩家也常认为 Lua 是一种游戏自身特有的语言。

作为一种独立的编程语言，Lua 也是非常有用的。它不仅可用于文本处理或编写一次性的小程序，还可用于大中规模的项目。对于后者，Lua 的主要功能都来自于它的扩展库。Lua 的标准库就提供了模式匹配和一些其他的字符串处理函数[3]。另一方面，Lua 的扩展库数量也在不断地增加，当前存在着一大批可使用的外部库。例如，Kepler 项目[4]就是一个用于 Web 开发的 Lua 库，它提供了页面生成、数据库访问，以及 LDAP、XML 和 SOAP 方面的包（Package）。LuaForge 站点[5]罗列了许多 Lua 的外部库。

最后一类 Lua 用户是那些 C 程序员，他们在编写应用程序时，将 Lua 作为一个 C 程序库

① 源代码、手册及某些平台的二进制文件。
② 一种用于构造动态 Web 页面的程序。
③ 这样便使 Lua 语言具有了字符串处理和文本处理的能力。
④ http://www.keplerproject.org
⑤ http://www.luaforge.net

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

来使用。这类人更多地是在写 C 代码，而不是 Lua 代码。尽管如此，他们仍需要较好地掌握 Lua，从而创建出简单易用且便于和语言相集成的接口。

这本书为这三类人都提供了充足的内容。第一部分介绍了语言本身，展示了语言所具有的所有能力。我们会集中地介绍语言中的各种功能，并使用大量示例来演示如何将它们运用到实际的任务中。其中，还会有一些章节会涉及到基础概念，例如控制结构。而另外一些章节会介绍一些高级主题，例如迭代器和协同程序。

第二部分深入地介绍了 table，它是 Lua 中唯一的一种数据结构。其他章节还讨论了数据结构、持久化、包和面向对象编程。这一部分将揭示语言真正的功能。

第三部分展示了 Lua 的标准库。这部分对那些想将 Lua 作为一门独立语言来使用的人特别有用。这部分中每一章介绍一个库，包括：数学库、table 库、字符串库、I/O 库、操作系统库、调试库。

本书的最后一部分将介绍 Lua 与 C 之间的 API，这是为那些想用 C 来访问 Lua 功能的人准备的。相对于本书的其他章，这部分内容显得非常与众不同。因为，我们将讨论 C 的编程，而非 Lua 编程。所以，那时我们需要换一种角度来思考问题。对于某些读者来说，C API 的讨论或许并非兴趣所在，但对于另一些读者来说，这可能是他们在本书中最关心的部分。

关于本书的第 2 版

本书是第 1 版《Lua 程序设计[①]》的更新和扩展。虽然两个版本的结构几乎一致，但新版本中具有一些全新的素材。

首先，将整本书更新到了 Lua 5.1。特别是那些与模块和包相关的章节，其中的大部分内容都是重写的。另外，还重写了一些示例，用来演示如何利用 Lua 5.1 提供的新特征。不过也清楚地注明了某些 Lua 5.0 中没有的特征，因此本书也可用于 Lua 5.0。

其次，多了一些新的示例。这些示例包括：图形表示、tab 扩展与压缩、一个元组（tuple）的实现等。

最后，本书还具有两章全新的内容。一章介绍了如何在 C 中使用多个状态和多个线程，并包含了一个实用的示例，演示如何为 Lua 实现多进程的能力。另一章介绍了内存管理，并演示了如何与内存分配及垃圾收集器打交道。

在《Lua 程序设计》的第 1 版出版后，一些出版社曾表示有意出版本书的第 2 版。但最终我们还是决定像第 1 版那样，由自己来出版这第 2 版。尽管这么做可能会影响收入效益，但这么做的优势也是很明显的，因为我们可以完全地控制本书的内容，可以自由地选择何时发布另一个版本，可以确保存有充足的货源，还可以保留以其他形式来发表本书的所有权利。

① “Programming in Lua”，也可简称为“PiL”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

其他资源

参考手册是所有希望学习一门语言的人所必需的东西，本书无意取代《Lua 参考手册》的地位。相反，它们之间是相互补充的。手册只描述了 Lua，其中既没有代码实例，也没有解释语言中构件的原理。但从另一方面看，参考手册描述了整个语言，而本书则没有触及语言中某些很少用到的“阴暗角落”。此外，手册是 Lua 的权威文档。如果本书与手册有矛盾之处，请相信手册。若要获取手册，或要获取更多有关 Lua 的信息，请访问 Lua 的官方网站：http://www.lua.org。

还可以在 Lua 的用户网站找到一些有用的信息，这个网站是由用户团体维护的，网址是：http://lua-users.org。这个网站提供了一套教程、一张第三方程序包和文档的列表及 Lua 官方邮件列表的档案。另外，还可以看一下本书的网页：http://www.inf.puc-rio.br/~roberto/pil2/。

在这里可以查到最新的勘误表，书中某些示例的代码和一些额外的材料。

这本书讲述的是 Lua 5.1，但书中绝大多数内容也适用于 Lua 5.0。本书会清楚地标明 Lua 5.1 和 Lua 5.0 之间的区别。如果正在使用一个较新的版本，请查阅相应的手册，以了解不同版本间的细微区别。如果还在用 5.0 之前的版本，那么应该升级一下了。

版式约定

本书将字面常量用一对双引号（""）括住，将单个字母（如'a'）用一对单引号括住。用于模式（Pattern）的字符串也用双引号括住，如"[%w_]*"。本书对一些程序块代码和标识符使用 Courier New 字体。更大的程序块则按以下方式给出：

```
-- "Hello World" 程序
print("Hello World")        --> Hello World
```

标记-->表示一条语句的输出结果，有时也表示一个表达式的结果：

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

由于两个连字符（--）在 Lua 中表示开始一条注释，因此在程序中包含这样的标记不会有什么问题。最后，本书使用标记<-->来表示某些等价的东西：

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

这表示对于 Lua 来说，无论是用 this 还是 that，结果都一样。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

致谢

本书若没有一些朋友和组织的帮助是不可能完成的。感谢 Lua 的共同开发者 Luiz Henrique de Figueiredo 和 Waldemar Celes 一直以来提供的各方面帮助。

感谢 Gavin Wraith、André Carregal、Asko Kauppi、Brett Kapilik、John D. Ramsdell 和 Edwin Moragas 对本书草稿的审读及提出的宝贵建议。

Lightning Source 公司在本书的印刷和发行中表现得非常可靠和高效，没有他们的话，由自己出版本书几乎是不可能的。

Dimaquina 的 Antonio Pedro 在耐心地听取了我对封面设计的意见后，最终提供了一个正确的封面设计。

Norman Ramsey 热心地提供了一些对于本书出版有益的建议。

我还要感谢 PUC-Rio 和 CNPq，感谢它们对我工作的一贯支持。

最后，我必须向 Noemi Rodriguez 表达我最深切的感谢，感谢你点亮了我的生活。

著　者

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

译 序

关于 Lua

Lua 为大多数国人所熟悉可能是源于几款知名的游戏。虽然作为一位游戏领域的开发者，可我对 Lua 的了解却始于一次无目的的闲暇自学（有时候学习本身，特别是对未知事物的探索过程，也是一种充满乐趣的体验）。当时，我了解并对比了 Lua 和 Python 这两种脚本语言，并且最终选择深入学习 Lua。因为我更喜爱 Lua 的简洁，即它的简单与整洁。

说 Lua 简单，是因为从语言的角度看，Lua 只提供了一些最基本的构件。通过这些构件，Lua 的用户可以创造出更复杂的功能。例如，Lua 并非严格意义上的面向对象语言，这是因为在它的语言层面上并没有直接提供诸如 class 这样的关键字，也没有显式的继承语法和 virtual 函数，但 Lua 却提供了一种创建这些面向对象要素的能力。在另一方面，Lua 中没有结构（例如 C 中的 struct）、没有 name space，甚至没有原生的数组。可是 Lua 却提供了创造这些元素的构件。而且更有趣的是，这些“万能的”构件归根到底就是一种常用的数据结构——关联数组，在 Lua 中称之为 table。

说 Lua 整洁，是因为无论其语言本身，还是其面向 C 语言的 API，都非常规整。Lua 的语法趋向于传统的结构化语言（如 Pascal），简单易懂。而它的 C API 则是基于一个特别的“虚拟栈”而工作的，这个栈用于在宿主语言与 Lua 之间交换数据。栈的设立大大简化了 C API 的设计与使用，并且能将 C 程序员（或其他语言的程序员）与 Lua 本身的实现有效地隔离开。

然而，我们还是需要辩证地去评判一样事物。不可否认，Lua 的简单和整洁也是有代价的。其直接后果可能就是我们很难用 Lua 去开发中型或大型的程序。但如果稍加思考的话，可以发现，要求一门脚本语言去开发一套中大规模的程序似乎显得有些“物非所用”。脚本的作用无非是完成一些相对简单、易变的任务，或作为宿主语言的延伸和扩展，或作为一种“胶水”粘合各种应用。在这些方面，简洁的 Lua 确实提供了良好的支持，“可扩展性”在 Lua 设计之初就被列为基本的设计目标之一。你可以很方便地为 Lua 编写扩展库，而不是只局限于语言所附带的标准程序库。

关于本书

我最初是在 Lua 官方网站上阅读了《Programming in Lua（第 1 版）》，而后一年中又断断续续地将第 1 版看了两遍有余。当获悉这本书的第 2 版问世后，便迫不及待地从 amazon.com 购买了它。当再次阅读时，深感本书对于学习 Lua 所具有的价值。因此我将它推荐给了电子工业出版社，并最终在与出版社的合作下引进了本书的中文版权。

作为关于Lua的最权威著作之一，本书由Lua的主要设计者和实现者Roberto Ierusalimschy

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

撰写。无论是从内容覆盖面的广度，还是对概念叙述的深度来看，本书都是学习和使用 Lua 的首选资料。

致谢

本书的翻译工作经历了春夏秋冬 4 个季节。此刻当我校对完本书的所有章节后，回忆起的是翻译过程中每个阶段的情景片段。本书最终得以成功出版离不开下面这些人的帮助与贡献。

在本书引进之初，电子工业出版社的何郑燕（Ina）起了主要的推动和促进作用，并且是她负责协调本书后续的翻译工作。另外，感谢她在过去的一年中对几次影响到译者翻译进度的事件所给予的包容和理解。

在翻译过程中，好友徐翎完成了本书第 3 部分（共 6 章）的初译稿，并且他和他的妻子龙春晖对本书的第一部分（共 10 章）进行了审校，并给出了不少专业的批注。

在临近交稿时，我的妻子张骥从一个非专业的角度对全书进行了逐字的阅读，并检查出许多错别字和语法问题。作为一个非专业读者，她对这本犹如天书般的读物所表现出的耐心，让我自愧不如。

在交稿后，王树伟编辑负责本书的后续工作，他在排版和审校上付出了很多的时间和精力。

最后，还要感谢张骥，是她让我体验到了非常美好的生活。

周惟迪

www.zhouweidi.name

联系方式

咨询电话：（010）88254160　88254161-67

电子邮件：support@fecit.com.cn

服务网址：http://www.fecit.com.cn　　http://www.fecit.net

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

目 录

第 1 部分

第 1 章 开始 …… 3
1.1 程序块（chunk） …… 3
1.2 词法规范 …… 5
1.3 全局变量 …… 6
1.4 解释器程序（the dtand-slone interpreter） …… 7
第 2 章 类型与值 …… 9
2.1 nil（空） …… 10
2.2 boolean（布尔） …… 10
2.3 number（数字） …… 10
2.4 string（字符串） …… 11
2.5 table（表） …… 14
2.6 function（函数） …… 17
2.7 userdata（自定义类型）和 thread（线程） …… 18
第 3 章 表达式 …… 19
3.1 算术操作符 …… 19
3.2 关系操作符 …… 20
3.3 逻辑操作符 …… 20
3.4 字符串连接 …… 22
3.5 优先级 …… 22
3.6 table 构造式（table constructor） …… 23
第 4 章 语句 …… 27
4.1 赋值 …… 27
4.2 局部变量与块（block） …… 28
4.3 控制结构 …… 30
4.3.1 if then else …… 30
4.3.2 while …… 31
4.3.3 repeat …… 31
4.3.4 数字型 for（numeric for） …… 31
4.3.5 泛型 for（generic for） …… 32

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.4 break 与 return ····· 34
第 5 章 函数 ····· 35
5.1 多重返回值（multiple results） ····· 36
5.2 变长参数（variable number of arguments） ····· 39
5.3 具名实参（named arguments） ····· 42
第 6 章 深入函数 ····· 45
6.1 closure（闭合函数） ····· 47
6.2 非全局的函数（non-global function） ····· 50
6.3 正确的尾调用（proper tail call） ····· 52
第 7 章 迭代器与泛型 for ····· 55
7.1 迭代器与 closure ····· 55
7.2 泛型 for 的语义 ····· 57
7.3 无状态的迭代器 ····· 58
7.4 具有复杂状态的迭代器 ····· 60
7.5 真正的迭代器 ····· 61
第 8 章 编译、执行与错误 ····· 63
8.1 编译 ····· 63
8.2 C 代码 ····· 66
8.3 错误（error） ····· 67
8.4 错误处理与异常 ····· 69
8.5 错误消息与追溯（traceback） ····· 70
第 9 章 协同程序（coroutine） ····· 73
9.1 协同程序基础 ····· 73
9.2 管道（pipe）与过滤器（filter） ····· 76
9.3 以协同程序实现迭代器 ····· 78
9.4 非抢先式的（non-preemptive）多线程 ····· 81
第 10 章 完整的示例 ····· 87
10.1 数据描述 ····· 87
10.2 马尔可夫链（markov chain）算法 ····· 90

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 2 部分

第 11 章　数据结构 …… 95
11.1　数组 …… 95
11.2　矩阵与多维数组 …… 96
11.3　链表 …… 97
11.4　队列与双向队列 …… 98
11.5　集合与无序组（bag） …… 99
11.6　字符串缓冲 …… 100
11.7　图 …… 102
第 12 章　数据文件与持久性 …… 105
12.1　数据文件 …… 105
12.2　串行化（Serialization） …… 107
12.2.1　保存无环的 table …… 109
12.2.2　保存有环的 table …… 110
第 13 章　元表（metatable）与元方法（meatmethod） …… 113
13.1　算术类的元方法 …… 114
13.2　关系类的元方法 …… 116
13.3　库定义的元方法 …… 117
13.4　table 访问的元方法 …… 118
13.4.1　__index 元方法 …… 118
13.4.2　__newindex 元方法 …… 120
13.4.3　具有默认值的 table …… 120
13.4.4　跟踪 table 的访问 …… 121
13.4.5　只读的 table …… 123
第 14 章　环境 …… 125
14.1　具有动态名字的全局变量 …… 125
14.2　全局变量声明 …… 127
14.3　非全局的环境 …… 129
第 15 章　模块与包 …… 133
15.1　require 函数 …… 134
15.2　编写模块的基本方法 …… 136

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

15.3 使用环境 …… 138
15.4 module 函数 …… 140
15.5 子模块与包 …… 141
第 16 章 面向对象编程 …… 143
16.1 类 …… 144
16.2 继承 …… 146
16.3 多重继承 …… 148
16.4 私密性 …… 150
16.5 单一方法（single-method）做法 …… 152
第 17 章 弱引用 table …… 153
17.1 备忘录（memoize）函数 …… 154
17.2 对象属性 …… 156
17.3 回顾 table 的默认值 …… 157

第 3 部分

第 18 章 数学库 …… 161
第 19 章 table 库 …… 163
19.1 插入和删除 …… 163
19.2 排序 …… 163
19.3 连接 …… 165
第 20 章 字符串库 …… 167
20.1 基础字符串函数 …… 167
20.2 模式匹配（pattern-matching）函数 …… 169
20.2.1 string.find 函数 …… 169
20.2.2 string.match 函数 …… 170
20.2.3 string.gsub 函数 …… 170
20.2.4 string.gmatch 函数 …… 171
20.3 模式 …… 172
20.4 捕获（capture） …… 175
20.5 替换 …… 177
20.5.1 URL 编码 …… 178

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

20.5.2 tab 扩展 …… 180
20.6 技巧 …… 181
第 21 章 I/O 库 …… 185
21.1 简单 I/O 模型 …… 185
21.2 完整 I/O 模型 …… 188
21.2.1 性能小诀窍 …… 189
21.2.2 二进制文件 …… 190
21.2.3 其他文件操作 …… 192
第 22 章 操作系统库 …… 193
22.1 日期和时间 …… 193
22.2 其他系统调用 …… 195
第 23 章 调试库 …… 197
23.1 自省机制 …… 197
23.1.1 访问局部变量 …… 199
23.1.2 访问非局部的变量（non-local varible） …… 200
23.1.3 访问其他协同程序 …… 201
23.2 钩子 …… 202
23.3 性能剖析（profile） …… 202

第 4 部分

第 24 章 C API 概述 …… 207
24.1 第一个示例 …… 208
24.2 栈 …… 210
24.2.1 压入元素 …… 211
24.2.2 查询元素 …… 212
24.2.3 其他栈操作 …… 214
24.3 C API 中的错误处理 …… 215
24.3.1 应用程序代码中的错误处理 …… 216
24.3.2 库代码中的错误处理 …… 216
第 25 章 扩展应用程序 …… 219
25.1 基础 …… 219

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

25.2 table 操作 …… 220
25.3 调用 Lua 函数 …… 224
25.4 一个通用的调用函数 …… 226
第 26 章 从 Lua 调用 C …… 229
26.1 C 函数 …… 229
26.2 C 模块 …… 231
第 27 章 编写 C 函数的技术 …… 233
27.1 数组操作 …… 233
27.2 字符串操作 …… 234
27.3 在 C 函数中保存状态 …… 237
27.3.1 注册表（registry） …… 237
27.3.2 C 函数的环境 …… 239
27.3.3 upvalue …… 239
第 28 章 用户自定义类型 …… 243
28.1 userdata …… 243
28.2 元表 …… 246
28.3 面向对象的访问 …… 248
28.4 数组访问 …… 250
28.5 轻量级 userdata（light userdata） …… 251
第 29 章 管理资源 …… 253
29.1 目录迭代器 …… 253
29.2 XML 分析器 …… 256
第 30 章 线程和状态 …… 265
30.1 多个线程 …… 265
30.2 Lua 状态 …… 269
第 31 章 内存管理 …… 277
31.1 分配函数 …… 277
31.2 垃圾收集器 …… 279
31.2.1 原子操作 …… 280
31.2.2 垃圾收集器的 API …… 280

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第1部分

Second Edition

Programming in Lua

Lua

- 第 1 章　开始
- 第 2 章　类型与值
- 第 3 章　表达式
- 第 4 章　语句
- 第 5 章　函数
- 第 6 章　深入函数
- 第 7 章　迭代器与泛型 for
- 第 8 章　编译、执行与错误
- 第 9 章　协同程序（coroutine）
- 第 10 章　完整的示例

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 1 章　开　　始

第一个 Lua 程序将遵循传统做法，打印一句“Hello World”：

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

如果已经拥有了一个独立的 Lua 解释器程序，要运行上面的 Lua 程序，只需将这段代码保存到一个文本文件，然后以这个文件名作为参数来调用解释器[1]即可。例如，已将上述代码保存为文件 hello.lua，那么可以使用以下命令来运行它：

```
% lua hello.lua
```

再来看一个稍微复杂点的例子。下面这段程序定义了一个计算阶乘的函数。它首先要求用户输入一个数字，然后打印出这个数的阶乘结果：

```
-- 定义一个阶乘函数
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")        -- 读取一个数字
print(fact(a))
```

如果用户使用的是某种内嵌入应用程序中的 Lua，例如 CGILua 或 IUPLua，那么可能需要参考一下这些程序的使用手册（或就近咨询某些熟悉它们的专家），以了解如何运行上述程序。无论何种形式，所用的 Lua 都是同一种编程语言。不过，推荐使用独立的 Lua 解释器来学习和运行书中的示例程序。

1.1　程序块（chunk）

Lua 执行的每段代码，例如一个源代码文件或在交互模式中输入的一行代码，都称为一个

① 通常是一个叫“Lua”的程序。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

“程序块”。一个程序块也就是一连串的语句或命令。

几条连续的 Lua 语句之间并不需要分隔符，但如果愿意，也可以使用分号来分隔语句。例如，当两条或多条语句并列出现在同一行时使用分号分隔它们。在 Lua 的语法中，代码中的换行不起任何作用。例如，以下 4 个程序块都是合法的，并且完全等价：

```
a = 1
b = a*2

a = 1;
b = a*2;

a = 1; b = a*2

a = 1  b = a*2    -- 有点难看，但却合法
```

一个程序块可以简单到只包含一条语句，就像“Hello World”示例那样，但也可以像阶乘示例那样，由多条不同的语句及函数定义[1]构成。程序块可以是任意大小的。另外，Lua 通常还被作为一种数据描述语言来使用，几兆字节的程序块也是很常见的。Lua 解释器对于大型程序块的处理不会有任何问题。

除了将程序代码保存到一个文件中再执行外，还可以在交互模式中运行解释器。当不使用任何参数而直接运行解释器时，就会看到这样的提示符：

```
Lua 5.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
>
```

在这种模式中，输入的每条命令（例如 print "Hello World"）都将立即被执行。要退出交互模式和解释器，只需输入一个 end-of-file 控制字符[2]，或者调用操作系统库的 exit 函数，输入“os.exit()”。

在交互模式中，解释器通常会将所输入的每行内容作为一个完整的程序块来解释。如果它检测到某行的内容不足以构成一个完整的程序块，那么它会等待输入更多内容，直至所有这些内容可以构成一个完整的程序块。有了这种机制，就可以在交互模式下直接输入像阶乘函数那样的多行定义了。不过，更常见的做法是先将多行内容放入一个文件中，然后再调用解释器执行这个文件。

若使用命令行参数-i 来启动 Lua 解释器，那么解释器就会在运行完指定程序块后进入交互模式。例如在命令行中输入：

```
% lua -i prog
```

① 接下来会看到，所谓“函数定义”其实是一种赋值语句。
② 在 UNIX 中为【Ctrl+D】组合键，在 DOS/Windows 中为【Ctrl+Z】组合键。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这样会先运行文件 prog 中的程序块，然后再进入交互模式。参数-i 对于调试和手工测试尤为有用。在本章最后，还会介绍解释器的一些其他参数。

另一种运行程序块的方式是使用函数 dofile，这个函数会立即执行一个文件。例如，假设有一个叫 lib1.lua 的文件，内容如下：

```
function norm (x, y)
    return (x^2 + y^2)^0.5
end

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

那么，在交互模式中输入：

```
> dofile("lib1.lua")    -- 加载程序库
> n = norm(3.4, 1.0)
> print(twice(n))       --> 7.0880180586677
```

如果要测试一段代码，dofile 函数会比较有用。可以同时打开两个窗口，一个文本编辑器用于编辑代码文件（例如 prog.lua），另一个命令行窗口运行解释器的交互模式。当编辑完代码并保存后，可以在解释器的交互模式窗口中执行“dofile("prog.lua")”来加载新的代码。这样便可以测试新代码，调用其中的函数并打印运行结果了。

1.2 词法规范

Lua 中的标识符可以是由任意字母、数字和下画线构成的字符串，但不能以数字开头。下面是一些例子：

```
i     j     i10     _ij
aSomewhatLongName    _INPUT
```

应该避免使用以一个下画线开头并跟着一个或多个大写字母（例如“_VERSION”）的标识符，Lua 将这类标识符保留用作特殊用途。通常保留标识符“_”（一个下画线）作为“哑变量（Dummy Variable）”使用。

在 Lua 中，“什么是字母”的概念依赖于区域设置（Locale）。如果设置了一个正确的区域，那么便可以使用例如“índice”或“ação”作为变量名了。但要注意，这样的变量名可能会使程序无法在那些不支持该区域的系统上运行。

以下是 Lua 的保留字，不能将它们用作标识符：

```
and     break     do     else     elseif
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
end      false    for      function  if
in       local    nil      not       or
repeat   return   then     true      until
while
```

Lua 是有大小写之分的。“**and**”是一个保留字，但“And”和“AND”却是两个不同的标识符。

可以在任何地方以两个连字符（--）开始一个“行注释”，该注释一直延伸到一行的结尾。Lua 也提供了“块注释”，以“--[[”开始，直至“]]”[1]。当注释一段代码时，一个常见的技巧是将这些代码放入“--[[”和“--]]”中间，例如如下代码：

```
--[[
print(10)        -- 不起作用（因为这是注释）
--]]
```

当重新启用这段代码时，只需在第一行行首添加一个连字符即可：

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

在第一个示例中，最后一行的“--”仍在一个块注释中。在第二个示例中，序列“---[[”只是开始了一个普通的行注释，而非想象中的块注释结尾。它的第一行和最后一行是两个彼此独立的行注释，因此 print 位于注释之外。

1.3 全局变量

全局变量（Global Variables）不需要声明。只需将一个值赋予一个全局变量就可以创建了。在 Lua 中，访问一个未初始化的变量不会引发错误，访问结果是一个特殊的值 **nil**。例如：

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

通常没有必要删除一个全局变量。如果一个变量只有较短的生存周期，那么就应该使用局部变量。但是，如果一定要删除某个全局变量的话，只需将其赋值为 **nil**：

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

在这句赋值之后，Lua 就会好像从未使用过这个变量一样。换句话说，如果存在一个全局

① 实际上，“块注释”可以复杂得多，详见 2.4 节。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

变量，那么它必定具有一个非 nil 的值。

1.4 解释器程序（the dtand-slone interpreter）

解释器[1]是一个小型的程序，可以通过它来直接使用 Lua。这一节将介绍它的几个主要选项参数。

如果代码文件的第一行以一个井号（#）开头，那么在加载该文件时，解释器将忽略这一行。这项特征主要是为了方便在 UNIX 系统中将 Lua 作为一种脚本解释器来使用。如果用下面这行开始脚本代码的编写：

```
#!/usr/local/bin/lua②
```

或

```
#!/usr/bin/env lua
```

那么便可以直接调用脚本文件，而无须显式地调用 Lua 解释器了。

解释器程序的用法如下：

```
lua [选项参数] [脚本[参数]]
```

所有这些参数都是可选的。就像刚才所说的，当不使用任何参数来启动解释器时，就会直接进入交互模式。

选项参数“-e”可以直接在命令行中输入代码，例如：

```
% lua -e "print(math.sin(12))"③   --> -0.53657291800043
```

选项参数“-l”用于加载库文件。而“-i”则如先前所说的，表示在运行完其他命令行参数后进入交互模式。最后，来看一个调用实例：

```
% lua -i -l a -e "x = 10"
```

这样会先加载库文件 a，然后执行赋值语句“x = 10”，最后显示一个交互模式的命令提示符。

只要定义了一个名为“_PROMPT”的全局变量，解释器就会用它的值作为交互模式的命令提示符。所以，可以使用下面的调用来改变命令提示符：

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

① 根据其源代码文件，也可以称其为“lua.c”；或者，根据其可执行文件名，简单地称之为“lua”。
② 假设解释器程序位于"/usr/local/bin"。
③ 在 UNIX 中需要使用双引号来防止 shell 误解括号。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

假设“%”是 shell[1]提示符。在上例中，外面的双引号用于阻止 shell 误解内层的单引号，这些内容应由 Lua 来解释处理。更明确地说，Lua 会接收并执行以下内容：

```
_PROMPT=' lua> '
```

这句语句将字符串" lua> "赋值给全局变量_PROMPT。

在交互模式中，如要打印任何表达式的值，可以用等号开头，并跟随一个表达式。例如：

```
> = math.sin(3)          --> 0.14112000805987
> a = 30
> = a                    --> 30
```

这个特征有助于将 Lua 作为一个计算器来使用。

在解释器执行其参数前，会先查找一个名为 LUA_INIT 的环境变量，如果找到了这个变量，并且其内容为“@文件名”，那么解释器会先执行这个文件。如果 LUA_INIT 没有以“@”开头，那么解释器就假设变量内容为 Lua 代码，并运行此代码。由于 LUA_INIT 可以很灵活地配置解释器，并且可以完全控制如何配置它。例如，可以预先加载一个程序包（Package）、修改命令提示符和路径、定义函数、对函数进行改名或删除等。

在脚本代码中，可以通过全局变量 arg 来检索脚本的启动参数。例如：

```
% lua 脚本 a b c
```

解释器在运行脚本前，会用所有的命令行参数创建一个名为“arg”的 table。脚本名称位于索引 0 上，它的第一个参数（示例中的“a”）位于索引 1，以此类推。而在“脚本”之前的所有选项参数则位于负数索引上。举例如下：

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

解释器将所有参数组织排列为：

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

通常脚本只会使用正数索引（示例中的 arg[1]和 arg[2]）。

在 Lua 5.1 中，一个脚本还可以通过“变长参数语法（vararg syntax）”来检索其参数。在脚本主体中，表达式“...”（3 个点）表示所有传递给脚本的参数。将在 5.2 节中讨论变长参数语法。

① 译注：操作系统的命令解释器。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第2章 类型与值

Lua 是一种动态类型的语言。在语言中没有类型定义的语法，每个值都“携带”了它自身的类型信息。

在 Lua 中有 8 种基础类型：nil（空）、boolean（布尔）、number（数字）、string（字符串）、userdata（自定义类型）、function（函数）、thread（线程）和 table（表）。函数 type 可根据一个值返回其类型名称。

```
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
```

最后一行将永远返回“string”，而无关乎 X 这个值的内容。这是因为 type 函数总是返回一个字符串。

变量没有预定义的类型，任何变量都可以包含任何类型的值：

```
print(type(a))        --> nil （a 尚未初始化）
a = 10
print(type(a))        --> number
a = "a string!!"
print(type(a))        --> string
a = print             -- 是的，这是合法的！
a(type(a))            --> function
```

请注意最后两行。在 Lua 中，函数是作为“第一类值（first-class value）”来看待的，可以像操作其他值一样来操作一个函数值（更多关于这方面的内容，在第 6 章中有详细讨论）。

将一个变量用于不同类型，通常会导致混乱的代码，但有时明智地使用这种特性会带来便利。例如，在异常情况下，可以返回一个 **nil** 以区别于其他正常的返回值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.1 nil（空）

nil 是一种类型，它只有一个值 **nil**，它的主要功能是用于区别其他任何值。就像之前所说的，一个全局变量在第一次赋值前的默认值就是 **nil**，将 **nil** 赋予一个全局变量等同于删除它。Lua 将 **nil** 用于表示一种“无效值（non-value）”的情况，即没有任何有效值的情况。

2.2 boolean（布尔）

boolean 类型有两个可选值：**false** 和 **true**，这与传统的布尔值一样。然而 boolean 却不是一个条件值的唯一表示方式。在 Lua 中任何值都可以表示一个条件[①]。Lua 将值 **false** 和 **nil** 视为“假”，而将除此之外的其他值视为“真”。请注意，还有一点不同于某些脚本语言，Lua 在条件测试中，将数字零和空字符串也都视为“真”。

2.3 number（数字）

number 类型用于表示实数[②]。Lua 没有整数类型，因为没有必要。一直以来都存在着一个关于浮点数算术错误的误解。有人会担心即使对浮点数进行一个简单的递增运算都有可能导致错误的结果。而事实上，只要使用双精度来表示一个整数，就不会出现“四舍五入（Rounding）”的错误[③]。因此，Lua 中的数字可以表示任何 32 位整数，而不会产生四舍五入的错误。此外，当今大多数 CPU 的浮点数运算速度和整数运算一样快，而有的 CPU 的浮点数运算可能还更快一点。

不过，通过重新编译 Lua 也可以非常方便地使用其他类型来表示数字，例如使用长整数 long 或单精度浮点数 float。这对于某些没有浮点数硬件支持的平台来说尤为有用。具体做法详见发行版中的 luaconf.h 文件。

书写一个数字常量时，可以使用普通的写法，也可以使用科学计数法，以下是一些合法的数字常量：

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

① 例如流程控制语句中的条件表达式。
② 双精度浮点数。
③ 除非这个数字大于 1014。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.4　string（字符串）

Lua 中的字符串通常表示“一个字符序列”。Lua 完全采用 8 位编码，Lua 字符串中的字符可以具有任何数值编码，包括数值 0。也就是说，可以将任意二进制数据存储到一个字符串中。

Lua 的字符串是不可变的值（immutable values）。不能像在 C 语言中那样直接修改字符串的某个字符，而是应该根据修改要求来创建一个新的字符串。如下所示：

```
a = "one string"
b = string.gsub(a, "one", "another")  -- 修改字符串的一部分
print(a)      --> one string
print(b)      --> another string
```

Lua 的字符串和其他 Lua 对象（例如 table 或函数等）一样，都是自动内存管理机制所管理的对象。这表示无须担心字符串的分配和释放，Lua 你处理这些事情。一个字符串可以小到只包含一个字母，也可以大到包含整本书。Lua 能够高效地处理长字符串。在 Lua 程序中操作 100K 或 1M 的字符串是很常见的。

字面字符串（literal string）需要以一对匹配的单引号或双引号来界定：

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

根据编程风格，应该坚持在程序中使用相同类型的引号（单引号或双引号）。除非字符串本身包含引号，那么可以使用另一种引号来界定字面字符串。或者使用反斜杠对引号进行转义。Lua 字符串中可以包含类似于 C 语言中的转义序列，如表 2-1 所示。

表 2-1　Lua 字符串中的转义序列

\a	响铃
\b	退格（back space）
\f	提供表格（form feed）
\n	换行
\r	回车
\t	水平 tab
\v	垂直 tab
\\	反斜杠
\"	双引号
\'	单引号

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

以下示例说明了它们的用法：

```
> 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: '\'
```

还可以通过数值来指定字符串中的字符，数值以转义序列“\<ddd>”给出，其中<ddd>是一个至多 3 个十进制数字组成的序列。例如，字符串"alo\n123\""与字符串'\97lo \10\04923"'是相同的。因为在使用 ASCII 的系统中，'a'的 ASCII 编码是 97，“换行（new line）”的编码是 10，'1'的 ASCII 编码是 49[①]。

另外，还可以用一对匹配的双方括号来界定一个字母字符串，就像写“块注释”那样。以这种形式书写的字符串可以延伸多行，Lua 不会解释其中的转义序列。此外，如果字符串的第一个字符是一个换行字符（new line），那么 Lua 会忽略它。这种写法对于书写那种含有程序代码的字符串尤为有用。如下例：

```
page = [[
<html>
<head>
<title>An HTML Page</title>
</head>
<body>
<a href="http://www.lua.org">Lua</a>
</body>
</html>
]]

write(page)
```

有时字符串中可能需要包含这样的内容：a=b[c[i]][②]。或者，可能需要包含已经被注释掉的代码。为了应付这种情况，需要在两个左方括号间加上任意数量的等号，就像[===[[③]。经过这样修改后，字面字符串只有在遇到了一个内嵌有相同数量等号的双右方括号时才会结束（就前例而言，即]===]）。如果一组左右方括号中的等号数量不等，那么 Lua 会忽略它。通过选择适当数量的等号，就可以在不加转义的情况下，直接嵌入任意的字符串内容了。

① 在此例中，由于紧随 49 之后的也是一个数字，所以必须将 49 写成 3 个数字，即“\049”。不然 Lua 会读取到数值 492。
② 注意其中的“]]”。
③ 这是 Lua 5.1 的新功能。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这套机制同样适用于注释。例如，以“--[=[”开始的一个块注释将延伸至“]=]”结束。如此便简化了注释那些“已经包含了注释块”的代码。

Lua 提供了运行时的数字与字符串的自动转换。在一个字符串上应用算术操作时，Lua 会尝试将这个字符串转换成一个数字：

```
print("10" + 1)          --> 11
print("10 + 1")          --> 10 + 1
print("-5.3e-10"*"2")     --> -1.06e-09
print("hello" + 1)        -- 错误（不能转换"hello"）
```

Lua 不仅在算术操作中会施以这种强制转换，还会在其他任何需要数字的地方这么做。

相反，在 Lua 期望一个字符串但却得到一个数字时，它也会将数字转换成字符串：

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

在 Lua 中，“..”是字符串连接操作符。当直接在一个数字后面输入它的时候，必须要用一个空格来分隔它们。不然，Lua 会将第一个点理解为一个小数点。

如今，仍不能确定在 Lua 的设计中，这些自动的强制转换是否算一项好的设计，建议最好不要依赖它们。虽然在某些地方这些转换显得很便利，但它们也给语言和使用它们的程序带来了复杂性。毕竟，字符串和数字是两样不同的东西。比较运算 10=="10"总为 false，因为 10 是一个数字，而"10"是一个字符串。如果需要显式地将一个字符串转换成数字，可以使用函数 tonumber。当这个字符串的内容不能表示一个正确的数字时，tonumber 将返回 **nil**。

```
line = io.read()      -- 读取一行
n = tonumber(line)    -- 尝试将它转换为一个数字
if n == nil then
  error(line .. " is not a valid number")
else
  print(n*2)
end
```

若要将一个数字转换成字符串，可以调用函数 tostring，或者将该数字与一个空字符串相连接：

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

这样的转换永远是合法的。

在 Lua 5.1 中，可以在字符串前放置操作符“#”[①]来获得该字符串的长度：

```
a = "hello"
print(#a)              --> 5
print(#"good\0bye")    --> 8
```

① 称为“长度操作符（length operator）”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.5 table（表）

table 类型实现了“关联数组（associative array）”。“关联数组”是一种具有特殊索引方式的数组。不仅可以通过整数来索引它，还可以使用字符串或其他类型的值（除了 **nil**）来索引它。此外，table 没有固定的大小，可以动态地添加任意数量的元素到一个 table 中。table 是 Lua 中主要的（事实上也是仅有的）数据结构机制，具有强大的功能。基于 table，可以以一种简单、统一和高效的方式来表示普通数组、符号表（symbol table）、集合、记录、队列和其他数据结构。Lua 也是通过 table 来表示模块（module）、包（package）和对象（object）的。当输入 io.read 的时候，其含义是“io 模块中的 read 函数”。对于 Lua 而言，这表示“使用字符串"read"作为 key（键）来索引 table io”。

在 Lua 中，table 既不是“值”也不是“变量”，而是“对象”。如果了解 Java 或 Scheme 中的数组，就会清楚明白其中的意思了。可以将一个 table 想象成一种动态分配的对象，程序仅持有一个对它们的引用（或指针），Lua 不会暗中产生 table 的副本或创建新的 table。此外，在 Lua 中也不需要声明一个 table。事实上也没有办法可以声明 table。table 的创建是通过“构造表达式（constructor expression）”完成的，最简单的构造表达式就是{}。

```
a = {}                  -- 创建一个table，并将它的引用存储到a
k = "x"
a[k] = 10               -- 新条目，key="x"，value=10
a[20] = "great"         -- 新条目，key=20，value="great"
print(a["x"])           --> 10
k = 20
print(a[k])             --> "great"
a["x"] = a["x"] + 1         -- 递增条目"x"
print(a["x"])           --> 11
```

table 永远是“匿名的（anonymous）”，一个持有 table 的变量与 table 自身之间没有固定的关联性。

```
a = {}
a["x"] = 10
b = a              -- b与a引用了同一个table
print(b["x"])       --> 10
b["x"] = 20
print(a["x"])       --> 20
a = nil            -- 现在只有b还在引用table
b = nil            -- 再也没有对table的引用了
```

当一个程序再也没有对一个 table 的引用时，Lua 的垃圾收集器（garbage collector）最终

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

会删除该 table，并复用它的内存。

所有 table 都可以用不同类型的索引来访问 value（值），当需要容纳新条目（entry）时，table 会自动增长。

```
a = {}     -- 空的table
-- 创建1000个新条目
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
```

请注意上例中的最后一行，这与全局变量一样，当 table 的某个元素没有初始化时，它的内容就为 **nil**。另外还可以像全局变量一样，将 **nil** 赋予 table 的某个元素来删除该元素。这种相似性是有原因的，因为 Lua 正是将全局变量存储在一个普通 table 中。在第 14 章中会详细讨论这个话题。

为了表示一条记录，可以将字段名作为索引。Lua 对于诸如 a["name"]的写法提供了一种更简便的“语法糖（syntactic sugar）”，可以直接输入 a.name。因此，上例中的最后几行，可以更简单地写为：

```
a.x = 10          -- 等同于a["x"] = 10
print(a.x)        -- 等同于print(a["x"])
print(a.y)        -- 等同于print(a["y"])
```

对于 Lua 来说，这两种形式是等价的，可供自由选择使用。然而这两种形式对于一个读者来说，可能就暗示了不同的意图。点的写法可能更明确地暗示了读者，将 table 作为一条记录来使用，每条记录都有一组固定的、预定义的 key。而字符串的写法可能暗示了该 table 会以任何字符串作为 key，而现在出于某些原因，需要访问某个特定的 key。

初学者常会将 a.x 和 a[x]搞错。前者表示 a["x"]，表示以字符串"x"来索引 table。而后者是以变量 x 的值来索引 table。下例说明了这种区别：

```
a = {}
x = "y"
a[x] = 10                  -- 将10放入字段"y"
print(a[x])    --> 10      -- 字段"y"的值
print(a.x)     --> nil     -- 字段"x"（未定义的）的值
print(a.y)     --> 10      -- 字段"y"的值
```

若要表示一个传统的数组或线性表，只需以整数作为 key 来使用 table 即可。这里不需要（也没有必要）声明一个大小值，直接初始化元素就可以了：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
-- 读取10行内容，并存储到一个table中
a = {}
for i=1,10 do
a[i] = io.read()
end
```

虽然可以用任何值作为一个 table 的索引，也可以用任何数字作为数组索引的起始值。但就 Lua 的习惯而言，数组通常以 1 作为索引的起始值[①]。并且还有不少机制依赖于这个惯例。

在 Lua 5.1 中，长度操作符“#”用于返回一个数组或线性表的最后一个索引值（或为其大小）[②]。例如，可以像下面这样打印上例中读取到的每行内容：

```
-- 打印所有的行
for i=1, #a do
  print(a[i])
end
```

以下是几种长度操作符在 Lua 中的习惯写法：

```
print(a[#a])          -- 打印列表a的最后一个值
a[#a] = nil           -- 删除最后一个值
a[#a+1] = v           -- 将v添加到列表末尾
```

下例演示了另一种方法来读取一个文件的前 10 行：

```
a = {}
for i=1,10 do
a[#a+1] = io.read()
end
```

由于数组实际上是一个 table，所以关于其大小的概念可能会有些模糊。例如，以下这个数组的大小算是多少呢？

```
a = {}
a[10000] = 1
```

请记住对于所有未初始化的元素的索引结果都是 **nil**。Lua 将 nil 作为界定数组结尾的标志。当一个数组有“空隙（Hole）”时，即中间含有 nil 时，长度操作符会认为这些 nil 元素就是结尾标记。可以肯定读者不会想要这种不可预期性。因此应该避免对那些含有“空隙”的数组使用长度操作符。大多数数组不会包含“空隙”（例如，前例文件中每个行不可能为 **nil**），因此大多数时候使用长度操作符是安全的。如果真的需要处理那些含有“空隙”的数组，可以使用函数 table.maxn[③]，它将返回一个 table 的最大正索引数：

① 这不同于 C 语言以 0 作为起始索引值的习惯。
② Lua 5.0 不支持长度操作符。可以使用函数 table.getn 来获得类似的结果。
③ 这是 Lua 5.1 的新函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
a = {}
a[10000] = 1
print(table.maxn(a))          --> 10000
```

由于可以用任何类型的值来索引 table，因此可能会遇到一些看似相同，但却实际不同的索引方式。例如，可以用数字 0 和字符串"0"来索引一个 table，这两个索引值是不同的（根据相等性测试），因此也就表示了 table 中两个不同的条目。与此类似的还有字符串"+1"、"01"和"1"，这些都表示了不同的条目。当对索引的实际类型不是很确定时，可以明确地使用一个显式转换：

```
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
```

如果对这点不加以注意，可能会给程序带来一些难以解决的 bug。

2.6　function（函数）

在 Lua 中，函数是作为“第一类值”来看待的。这表示函数可以存储在变量中，可以通过参数传递给其他函数，还可以作为其他函数的返回值。这种特性使语言具有了极大的灵活性。为了给一个函数添加新的功能，程序可以重新定义该函数。而在运行一些不受信任的代码时[①]，可以先删除某些函数，从而创建一个安全的运行环境。此外，Lua 对“函数式编程（functional programming）”也提供了良好的支持。例如，允许在某些词法域（lexical scoping）中编写嵌套的函数，第 6 章对此会进行详细讨论。最后在 Lua 的面向对象机制中，“第一类的”函数也扮演了重要的角色，详见第 16 章。

Lua 既可以调用以自身 Lua 语言编写的函数，又可以调用以 C 语言编写的函数。Lua 所有的标准库都是用 C 语言写的，标准库中包括对字符串的操作、table 的操作、I/O、操作系统的功能调用、数学函数和调试函数。同样，应用程序也可以用 C 语言来定义其他函数。

将在第 5 章详细讨论 Lua 的函数，在第 26 章讨论如何用 C 语言编写 Lua 函数。

① 例如通过网络接收到的代码。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.7 userdata（自定义类型）和 thread（线程）

由于 userdata 类型可以将任意的 C 语言数据存储到 Lua 变量中。在 Lua 中，这种类型没有太多的预定义操作，只能进行赋值和相等性测试。userdata 用于表示一种由应用程序或 C 语言库所创建的新类型，例如标准的 I/O 库就用 userdata 来表示文件。稍后将在 CAPI 中详细讨论这种类型。

第 9 章会解释 thread 类型，其中将讨论到“协同程序（coroutine）”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第3章 表 达 式

表达式用于表示值。Lua 的表达式中可以包含数字常量、字面字符串、变量、一元和二元操作符及函数调用。另外有别于传统的是，表达式中还可以包括函数定义和 table 构造式。

3.1 算术操作符

Lua 支持常规的算术操作符有：二元的“+”（加法）、“-”（减法）、“*”（乘法）、“/”（除法）、“^”（指数）、“%”（取模）[①]，一元的“-”（负号）。所有这些操作符都可用于实数。例如，x^0.5 将计算 x 的平方根，x^(-1/3)将计算 x 立方根的倒数。

取模操作符是根据以下规则定义的：

```
a % b == a - floor(a/b)*b
```

对于整数来说，以上算式通常都是有意义的，计算结果的符号永远与第二个参数相同。而对于实数，则可能有其他用途。例如，x%1 的结果就是 x 的小数部分，而 x-x%1 的结果就是其整数部分。类似地，x-x%0.01 则是 x 精确到小数点后两位的结果。

```
x = math.pi
print(x - x%0.01)      --> 3.14
```

另外还有一个使用取模操作符的例子。假设，检查一辆车是否在旋转了一定角度后开始往回开。那么可以这么做，假设旋转角是以角度给出的。

```
local tolerance = 10
function isturnback (angle)
  angle = angle % 360
  return (math.abs(angle - 180) < tolerance)
end
```

以上代码对于负的角度值也可以正常工作。

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

如果想以弧度代替角度，则只需修改代码中的常量即可。

① 取模操作是 Lua 5.1 中新增的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local tolerance = 0.17
function isturnback (angle)
  angle = angle % (2*math.pi)
  return (math.abs(angle - math.pi) < tolerance)
end
```

其中表达式 angle % (2*math.pi)就是将任意角度规范化为区间[0, 2π]。

3.2 关系操作符

Lua 提供了以下关系操作符：

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

所有这些操作符的运算结果都是 **true** 或 **false**。

操作符==用于相等性测试，操作符～=用于不等性测试。这两个操作符可以应用于任意两个值。如果两个值具有不同的类型，Lua 就认为它们是不相等的。否则，Lua 会根据它们的类型来比较两者。特别需要说明的是，**nil** 只与其自身相等。

对于 table、userdata 和函数，Lua 是作引用比较的。也就是说，只有当它们引用同一个对象时，才认为它们相等。例如，以下代码：

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

其结果是 a==c，但 a～=b。

只能对两个数字或两个字符串作大小性比较。Lua 是按照字母次序（alphabetical order）比较字符串的，具体的字母次序取决于对 Lua 的区域设置。例如，若将区域设置为“European Latin-1”，那么会得到"acai"<"açaí"<"acorde"。而数字和字符串之外的其他类型只能进行相等性或不等性比较。

当对两个不同类型的值作比较时，要格外小心。请记住，"0"与 0 是不同的。此外 2<15 显然是 true，但"2"<"15"却是 false[①]。为了避免类型不一致的比较，Lua 会在遇到字符串和数字的大小比较时引发一个错误，例如 2<"15"就会导致这种错误。

3.3 逻辑操作符

逻辑操作符有 **and**、**or** 和 **not**。与条件控制语句一样，所有的逻辑操作符将 **false** 和 **nil**

① 因为是按照字母次序来比较的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

视为假，而将其他的任何东西视为真。对于操作符 **and** 来说，如果它的第一个操作数为假，就返回第一个操作数；不然返回第二个操作数。对于操作符 **or** 来说，如果它的第一个操作数为真，就返回第一个操作数；不然返回第二个操作数。

```
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
```

and 和 **or** 都使用“短路求值（short-cut evaluation）”，也就是说，它们只会在需要时才去评估第二个操作数。短路求值可以确保像(type(v)=="table" and v.tag=="h1")这样的表达式不会导致运行时错误[①]。

有一种常用的 Lua 习惯写法“x=x or v”，它等价于：

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

它可用在没有设置 x 的时候（即对 x 的求值结果为假时），将其设为一个默认值 v。

另外，还有一种习惯写法是“(a and b) or c”[②]，这类似于 C 语言中的表达式 a ? b : c，但前提是 b 不为假[③]。例如，为了选出数字 x 和 y 中的较大者，可以使用以下语句：

```
max = (x > y) and x or y
```

若 x>y，则 and 的第一个操作数为真。那么 and 运算的结果就是其第二个操作数 x，而 x 是一个永远为真的表达式（因为它是一个数字）。然后 or 运算的结果就是其第一个操作数 x。当 x>y 为假的时候，and 表达式为假，因此 or 的结果是其第二个操作数 y。

操作符 not 永远只返回 true 或 false：

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

① 当 v 不是一个 table 时，Lua 不会对 v.tag 进行求值评估。

② 由于 and 的优先级高于 or，所以还可以更简单地写为 a and b or c。

③ 译注：如果还不是很明白此处作者所指的意思，那么可以看看以下注解。
这个的前提条件是很必要的。要明白它，必须先了解 Lua 的 and 和 or 的运算规则，即本节开头所述的内容。其次，还需要明白 C 语言中的三元表达式 a ? b : c 的意思，即“a 为真，结果为 b；a 为假，结果为 c”。
Lua 中的 and-or 写法若要完全等价于 C 语言中的?:表达式的语义，则必须满足此处的前提。试想，在(a and b) or c 中，b 为假，会产生怎么样的结果呢？若 a 为真时，a and b 的结果为 b。接下去计算 b or c，若 b 为假，那么根据 Lua 中 or 的运算规则，将返回 c。这显然与 C 语言的“a 为真，结果为 b；a 为假，结果为 c”的语义不同。
译者建议，在这种无法确信 b 为真的情况下，最安全的做法是使用正常的 if-else 语句（详见第 4 章）。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

3.4 字符串连接

要在 Lua 中连接两个字符串，可以使用操作符“..”（两个点）。如果其任意一个操作数是数字的话，Lua 会将这个数字转换成一个字符串：

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

请记住，Lua 中的字符串是不可变的值（immutable value）。连接操作符只会创建一个新字符串，而不会对其原操作数进行任何修改：

```
a = "Hello"
print(a .. " World")   --> Hello World
print(a)               --> Hello
```

3.5 优 先 级

Lua 操作符的优先级如表 3-1 所示（从高到低）。

表 3-1 Lua 操作符的优先级

^
not # - （一元）
* / %
+ -
..
< > <= >= ~= ==
and
or

在二元操作符中，除了指数操作符“^”和连接操作符“..”是“右结合”的，所有其他操作符都是“左结合（left associative）”的。因此，下例中左边的表达式等价于右边的表达式：

```
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)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

若不确定某些操作符的优先级时，就应显式地用括号来指定所期望的运算次序。这比查参考手册方便得多，而且当再次阅读代码时，也不会产生什么疑问。

3.6 table 构造式（table constructor）

构造式是用于创建和初始化 table 的表达式。这是 Lua 特有的一种表达式，并且也是 Lua 中最有用、最通用的机制之一。

最简单的构造式就是一个空构造式{}，用于创建一个空 table。构造式还可以用于初始化数组[1]。例如，以下语句：

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

会将 days[1]初始化为字符串"Sunday"[2]、days[2]初始化为"Monday"，以此类推。

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

Lua 还提供了一种特殊的语法用于初始化记录风格的 table：

```
a = {x=10, y=20}
```

以上这行代码等价于这些语句：

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

无论使用哪种方式来创建 table，都可以在 table 创建之后添加或删除其中的某些字段：

```
w = {x=0, y=0, label="console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "another field"     -- 添加key 1到table w
x.f = w                    -- 添加key "f"到table x
print(w["x"])                  --> 0
print(w[1])                    --> another field
print(x.f[1])                  --> another field
w.x = nil                      -- 删除字段"x"
```

那就是所有 table 都可以构建成一样的，而构造式只是在 table 的初始化时刻发挥作用。

每当 Lua 评估一个构造式时，都会先创建一个新 table，然后初始化它。这样，就能用 table 写出以下的链表代码：

```
list = nil
```

① 即“序列（sequence）”或“列表（list）”。
② 注意，构造式的第一个元素的索引为 1，而非 0。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

这段代码从标准输入中读取每行的内容，然后将每行按相反的次序存储到一个链表中。链表的每个结点都是一个 table，table 中含有两个字段：value（每行的内容）和 next（指向下一个结点的引用）。以下代码遍历了该链表，并打印了其内容[①]：

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

上例虽具有一定的教学意义，但在真实的 Lua 程序中很少会用到链表。列表数据一般是通过数组来实现的，第 11 章会讨论这个问题。

将记录风格的初始化与列表风格的初始化混合在一个构造式中使用：

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

上例演示了如何通过嵌套的构造式来表示复杂的数据结构。每个 polyline[i]元素都是一个 table，表示一条记录：

```
print(polyline[2].x)    --> -10
print(polyline[4].y)    --> 1
```

这两种风格的构造式各有其限制。例如，不能使用负数的索引，也不能用运算符作为记录的字段名。为了满足这些要求，Lua 还提供了一种更通用的格式。这种格式允许在方括号之间，显式地用一个表达式来初始化索引值：

```
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])         --> ---
```

① 由于是像栈（stack）一样来实现这个链表的，所以行的打印次序与读入次序相反。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这种语法看似烦琐了一点，但却非常灵活。无论是列表风格的初始化，还是记录风格的初始化，其实都是这种通用语法特例。构造式{x=0, y=0}等价于{["x"]=0, ["y"]=0}，构造式{"r", "g", "b"}等价于{[1]="r", [2]="g", [3]="b"}。

对于某些情况如果真的需要以 0 作为一个数组的起始索引的话，通过这种语法也可以轻松做到：

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

现在第一个值"Sunday"的索引就为 0 了。这个索引 0 并不影响其他元素，"Monday"照常索引为 1，因为它是构造式中列表风格中的第一个值，后续其他值的索引依次递增。但无论是否使用这种初始化语法，都不推荐在 Lua 中以 0 作为数组的起始索引。大多数内建函数都假设数组起始于索引 1，若遇到以索引 0 开始的数组，它们就无法进行正确地处理了。

你可以在最后一个元素后面写一个逗号，这个特性是可选的，也是合法的：

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

这种灵活性对于那些生成 Lua table 的程序来说很有用，它们无须将最后一个元素作为特例来处理。

最后，在一个构造式中还可以用分号代替逗号。通常会将分号用于分隔构造式中不同的成分，例如将列表部分与记录部分明显地区分开：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 4 章　语　　句

Lua 支持的常规语句基本上与 C 语言或 Pascal 语言中所支持的那些语句差不多。这些语句包括赋值、控制结构和过程调用。另外 Lua 还支持一些不太常见的语句，例如多重赋值（multiple assignment）和局部变量声明。

4.1　赋　　值

赋值（assignment）的基本含义是修改一个变量或一个 table 中字段的值：

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

Lua 允许“多重赋值”，也就是一下子将多个值赋予多个变量。每个值或每个变量之间以逗号分隔。例如：

```
a, b = 10, 2*x
```

赋值后，变量 a 为 10，b 为 2*x。

在多重赋值中，Lua 先对等号右边的所有元素求值，然后才执行赋值。这样便可以用一句多重赋值来交互两个变量了，如下所示：

```
x, y = y, x                    -- 交换x与y
a[i], a[j] = a[j], a[i]        -- 交换a[i]与a[j]
```

Lua 总是会将等号右边值的个数调整到与左边变量的个数相一致。规则是：若值的个数少于变量的个数，那么多余的变量会被赋为 nil；若值的个数更多的话，那么多余的值会被“静悄悄地”丢弃掉：

```
a, b, c = 0, 1
print(a, b, c)             --> 0   1   nil
a, b = a+1, b+1, b+2       -- 其中b+2会被忽略
print(a, b)                --> 1   2
a, b, c = 0
print(a, b, c)             --> 0   nil   nil
```

上例的最后那句赋值展示了一种比较常见的错误。若要初始化一组变量，应为每个变量

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

提供一个值：

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

实际上前面两个例子都是为了说明多重赋值的用法而刻意制造出来的。一般很少会在一行中为几个没有关联的变量使用多重赋值。多重赋值并不会比相等价的多条单一变量赋值语句更快。但有时的确会需要多重赋值，例如之前所说的交换两个变量。还有一种比较常见的应用是用于收集函数的多个返回值。在第5章中会讨论到一个函数允许返回多个值，在这种情况下，一次函数返回的结果就需要有多个变量来接收。例如，a, b = f()，函数f将返回两个值，a接收了第一个，b接收了第二个。

4.2 局部变量与块（block）

相对于全局变量，Lua还提供了局部变量。通过local语句来创建局部变量：

```
j = 10                -- 全局变量
local i = 1           -- 局部变量
```

与全局变量不同的是，局部变量的作用域仅限于声明它们的那个块。一个块（block）是一个控制结构的执行体、或者是一个函数的执行体再或者是一个程序块（chunk）：

```
x = 10
local i = 1           -- 程序块中的局部变量

while i <= x do
  local x = i*2       -- while循环体中的局部变量
  print(x)            --> 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x             -- then中的局部变量
  x = 20
  print(x + 2)        -- 如果测试成功会打印22
else
  print(x)            --> 10   （全局变量）
end

print(x)              --> 10   （全局变量）
```

请注意，如果是在交互模式中输入这段代码的话，该示例可能不会如预期的那样工作。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

因为在交互模式中每行输入内容自身就形成了一个程序块[1]。一旦输入了本例的第二行（local i = 1），Lua 就会马上运行这句话，并为下一行的运行开启一个新的程序块。到那时 **local** 声明就已经超出其作用域。为了解决这个问题，可以显式地界定一个块，只需将这些内容放入一对关键字 **do-end** 中即可。每当输入了 **do** 时，Lua 就不会单独地执行后面每行的内容，而是直至遇到一个相应的 end 时，才会执行整个块的内容。

如果需要更严格地控制某些局部变量的作用域时，这些 **do** 块也会有所帮助：

```
do
  local a2 = 2*a
  local d = (b^2 - 4*a*c)^(1/2)
  x1 = (-b + d)/a2
  x2 = (-b - d)/a2
end          -- a2 和 d 的作用域至此结束
print(x1, x2)
```

“尽可能地使用局部变量”是一种良好的编程风格。局部变量可以避免将一些无用的名称引入全局环境（global environment），避免搞乱了全局环境。此外，访问局部变量比访问全局变量更快。最后，一个局部变量通常会随着其作用域的结束而消失，这样便使垃圾收集器可以释放其值。

Lua 将局部变量的声明当做语句来处理。因此可以在任何允许书写语句的地方书写局部变量的声明。所声明的局部变量的作用域从声明语句开始，直至所在块的结尾。声明语句中还可以包含初始化赋值，其规则与普通的赋值语句完全一样：额外的值会被丢弃；额外的变量会被赋予 **nil**。如果一条声明语句没有初始化赋值，那么它声明的所有变量都会初始化为 **nil**：

```
local a, b = 1, 10
if a < b then
  print(a)    --> 1
  local a     -- 具有隐式的“= nil”
  print(a)    --> nil
end           -- then 块至此结束
print(a, b)   --> 1   10
```

在 Lua 中，有一种习惯写法是：

```
local foo = foo
```

这句代码创建了一个局部变量 foo，并将用全局变量 foo 的值初始化它[2]。如果后续其他函数改变了全局 foo 的值，那么可以在这里先将它的值保存起来。这种方式还可以加速在当前作用域中对 foo 的访问。

① 有一种特殊情况除外，那就是输入的一行内容不足以构成一条完整的命令。
② 这个局部的 foo 只有在其声明后才会变为可见（visible）。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于许多语言都强制程序员在一个块（或一个过程）起始处声明所有的局部变量，所以某些人就认为在一个块的中间使用声明语句是一种不好的习惯。但事实恰恰相反，在需要时才声明变量，可以使这个变量在初始化时刻就拥有一个有意义的初值[1]。此外，缩短变量的作用域有助于提高代码的可读性。

4.3 控制结构

Lua 提供了一组传统的、小巧的控制结构，包括用于条件执行的 **if**，用于迭代的 **while**、**repeat** 和 **for**。所有的控制结构都有一个显式的终止符：**if**、**for** 和 **while** 以 **end** 作为结尾，**repeat** 以 **until** 作为结尾。

控制结构中的条件表达式可以是任何值，Lua 将所有不是 **false** 和 **nil** 的值视为“真”[2]。

4.3.1 if then else

if 语句先测试其条件，然后根据测试结果执行 then 部分或 else 部分。else 部分是可选的。

```
if a < 0 then a = 0 end

if a < b then return a else return b end

if line > MAXLINES then
    showpage()
    line = 0
end
```

若要编写嵌套的 **if**，可以使用 **elseif**。它类似于在 **else** 后面紧跟一个 **if**，它还可以避免在这样的嵌套中出现多个 **end**：

```
if op == "+" then
  r = a + b
elseif op == "-" then
  r = a - b
elseif op == "*" then
  r = a*b
elseif op == "/" then
  r = a/b
else
  error("invalid operation")
end
```

① 这样就不会忘记去初始化它。
② 特别注意，Lua 将 0 和空字符串也视为真。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于 Lua 不支持 switch 语句，所以这种一连串的 if-else if 代码是很常见的。

4.3.2 while

与其他语言中的 while 循环一样，Lua 先测试 **while** 的条件。如果条件为假，那么循环结束；不然，Lua 执行循环体，并重复这一过程。

```
local i = 1
while a[i] do
  print(a[i])
  i = i + 1
end
```

4.3.3 repeat

正如这个名字所暗示的那样，一条 **repeat-until** 语句重复执行其循环体直到条件为真时结束。测试是在循环体之后做的，因此循环体至少会执行一次。

```
-- 打印输入的第一行不为空的内容
repeat
  line = io.read()
until line ~= ""
print(line)
```

与其他大多数语言不同的是，在 Lua 中，一个声明在循环体中的局部变量的作用域包括了条件测试[①]：

```
local sqr = x/2
repeat
  sqr = (sqr + x/sqr)/2
   local error = math.abs(sqr^2 - x)
until error < x/10000          --在此仍可以访问 error
```

4.3.4 数字型 for（numeric for）

for 语句有两种形式：数字型 **for**（numeric for）和泛型 **for**（generic for）。

数字型 **for** 的语法如下：

```
for var=exp1,exp2,exp3 do
  <执行体>
end
```

① 这是 Lua 5.1 的新功能。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

var 从 exp1 变化到 exp2，每次变化都以 exp3 作为步长（step）递增 var，并执行一次“执行体”。第三个表达式 exp3 是可选的，若不指定的话，Lua 会将步长默认为 1。以下是这种循环的一个典型示例：

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

如果不想给循环设置上限的话，可以使用常量 math.huge：

```
for i=1,math.huge do
  if (0.3*i^3 - 20*i^2 - 500 >= 0) then
    print(i)
    break
  end
end
```

为了更好地使用 **for** 循环，还需要了解一些小细节。首先，for 的 3 个表达式是在循环开始前一次性求值的。例如，上例中的 f(x)只会执行一次。其次，控制变量会被自动地声明为 **for** 语句的局部变量，并且仅在循环体内可见。因此，控制变量在循环结束后就不存在了：

```
for i=1,10 do print(i) end
max = i      -- 可能是错误的！这里访问的是一个全局的 i
```

如果需要在循环结束后访问控制变量的值（通常是在 break 循环时），必须将该值保存到另一个变量中：

```
-- 在一个列表中查找一个值
local found = nil
for i=1,#a do
if a[i] < 0 then
    found = i      -- 包含 i 的值
    break
  end
end
print(found)
```

最后一点，不要在循环过程中修改控制变量的值，否则会导致不可预知的效果。如果想在 **for** 循环正常结束前终止循环，可以像上例中那样使用 **break** 语句。

4.3.5 泛型 for（generic for）

泛型 **for** 循环通过一个迭代器（iterator）函数来遍历所有值：

```
-- 打印数组 a 的所有值
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

Lua 的基础库提供了 ipairs，这是一个用于遍历数组的迭代器函数。在每次循环中，i 会被赋予一个索引值，同时 v 被赋予一个对应于该索引的数组元素值。下面是另一个类似的示例，演示了如何遍历一个 table 中所有的 key：

```
-- 打印table t 中所有的key
for k in pairs(t) do print(k) end
```

从外观上看泛型 **for** 比较简单，但其实它是非常强大的。通过不同的迭代器，几乎可以遍历所有的东西，而且写出的代码极具可读性。标准库提供了几种迭代器，包括用于迭代文件中每行的（io.lines）、迭代 table 元素的（pairs）、迭代数组元素的（ipairs）、迭代字符串中单词的（string.gmatch）等。当然，读者还可以编写自己的迭代器。虽然泛型 **for** 的使用非常简单，但编写迭代器函数却有不少细节需要注意。将在第 7 章中讨论这个主题。

泛型 for 循环与数字型 for 循环有两个相同点：① 循环变量是循环体的局部变量；② 决不应该对循环变量作任何赋值。

对于泛型 **for** 的使用，再来看一个更具体的示例。假设有这样一个 table，它的内容是一周中每天的名称：

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

现在要将一个名称转换成它在一周中的位置。为此，需要根据给定的名称来搜索这个 table。然而在Lua中，通常更有效的方法是创建一个“逆向table”。例如这个逆向table叫revDays，它以一周中每天的名称作为索引，位置数字作为值：

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

接下来，要找出一个名称所对应的序号，只需用名字来索引这个 reverse table 即可：

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

当然，不必手动声明这个逆向 table，而是通过原来的 table 自动地构造出这个逆向 table：

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

这个循环会为每个元素进行赋值，其中变量 k 为 key（1、2、…），变量 v 为 value（"Sunday"、

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

"Monday"、…）。

4.4 break 与 return

break 和 **return** 语句用于跳出当前的块。

break 语句用于结束一个循环，它只会跳出包含它的那个内部循环（**for**、**repeat** 或 **while**），而不会改变外层的循环。在执行了 break 后，程序会在那个被跳出的循环之后继续执行。

return 语句用于从一个函数中返回结果，或者用于简单地结束一个函数的执行。任何函数的结尾处都有一句隐式的 return。所以如果有一个函数，它没有值需要返回，那么就无须在其结尾处添加 return 语句。

由于语法构造的原因，**break** 或 **return** 只能是一个块的最后一条语句。换句话说，它们应是程序块的最后一条语句，或者是 **end**、**else**[①]或 **until** 前的一条语句。例如，下例中的 **break** 就是 **then** 块的最后一条语句。

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

因为那些位于 return 或 break 之后的语句将无法执行到，所以通常只能在上述几个位置使用这些语句。然而有时可能希望在一个块的中间插入一句 **return** 或 **break**。例如，准备调试一个函数，但又不想执行该函数的内容。在这种情况中，可以使用一个显式的 **do** 块来包住一条 return 语句：

```
function foo ()
return                  --<< 语法错误
  -- 在下一个块中 return 就是最后一条语句
  do return end         -- OK
  <其他语句>
end
```

① 译注：“else 前的一条语句”，其具体所指请看下例：

```
if ... then
  ...
  return 或 break   -- 这里就是 else 之前
else
  ...
  return 或 break   -- 这里是 end 之前
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第5章 函 数

在 Lua 中，函数是一种对语句和表达式进行抽象的主要机制。函数既可以完成某项特定的任务[①]，也可以只做一些计算并返回结果。在第一种情况中，一句函数调用被视为一条语句；而在第二种情况中，则将其视为一句表达式：

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

无论哪种用法都需要将所有参数放到一对圆括号中。即使调用函数时没有参数，也必须写出一对空括号。对于此规则只有一种特殊的例外情况：一个函数若只有一个参数，并且此参数是一个字面字符串或 table 构造式，那么圆括号便是可有可无的。见下例：

```
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 为面向对象式的调用也提供了一种特殊的语法——冒号操作符。表达式 o.foo(o, x)的另一种写法是 o:foo(x)，冒号操作符使调用 o.foo 时将 o 隐含地作为函数的第一个参数。关于这种调用方式及面向对象编程的问题，将在第 16 章详细讨论。

一个 Lua 程序既可以使用以 Lua 编写的函数，又可以调用以 C 语言[②]编写的函数。例如，所有 Lua 标准程序库中的函数都是用 C 语言写的。但这些细节对于 Lua 程序员来说是透明的。无论一个函数是用 Lua 编写的还是用 C 语言编写的，在调用它时没有任何区别。

关于一个函数定义的常规语法，可以回顾一下之前的一个例子：

```
function add (a)
local sum = 0
  for i,v in ipairs(a) do
sum = sum + v
end
  return sum
end
```

① 有些语言也称之为“过程（procedure）”或“子程序（subroutine）”。
② 或宿主程序使用的其他语言。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在这种语法中，一个函数定义具有一个名称（本例中的 add）、一系列的参数（参数表）和一个函数体（即一系列的语句）。

“形式参数（parameter）[①]”的使用方式与局部变量非常相似，它们是由调用函数时的“实际参数（argument）[②]”初始化的。调用函数时提供的实参数量可以与形参数量不同。Lua 会自动调整实参的数量，以匹配参数表的要求。这项调整与多重赋值（multiple assignment）很相似，即“若实参多余形参，则舍弃多余的实参；若实参不足，则多余的形参初始化为 nil”。举例来说，假设一个函数如下：

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

在以下几种调用中，实参与形参的对应关系为：

```
调用          形参
f(3)          a=3, b=nil
f(3, 4)       a=3, b=4
f(3, 4, 5)    a=3, b=4   (5 被丢弃了)
```

虽然这种调整行为会导致一些编程错误[③]，但它也是很有用的，尤其是对于默认实参的应用。举例来说，考虑以下这个函数，它用于递增一个全局的计数器：

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

该函数以 1 作为默认实参，当调用 incCount()[④]时，count 增加 1。这是因为当调用 incCount() 时，Lua 先将 n 初始化为 **nil**，而接下来的 or 又返回了其第二个操作数。最终 Lua 将默认值 1 赋予了 n。

5.1 多重返回值（multiple results）

Lua 具有一项非常与众不同的特征，允许函数返回多个结果。Lua 的几个预定义函数就是返回多个值的。例如，用于在字符串中定位一个模式（pattern）的函数 string.find。该函数若在字符串中找到了指定的模式，将返回匹配的起始字符和结尾字符的索引。在此就需要使用多重赋值语句来接收函数的返回值。

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

① 译注：以下简称为“形参”，即函数定义时参数表中的参数。
② 译注：以下简称为“实参”，即调用函数时传入的参数。
③ 主要发生在运行时。
④ 即不带实参。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

以 Lua 编写的函数同样可以返回多个结果，只需在 **return** 关键字后列出所有的返回值即可。例如，需要写一个函数用于查找数组中的最大元素，并返回该元素的位置：

```
function maximum (a)
local mi = 1          -- 最大值的索引
local m = a[mi]       -- 最大值
  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 会调整一个函数的返回值数量以适应不同的调用情况。若将函数调用作为一条单独语句时，Lua 会丢弃函数的所有返回值。若将函数作为表达式的一部分来调用时，Lua 只保留函数的第一个返回值。只有当一个函数调用是一系列表达式中的最后一个元素(或仅有一个元素)时，才能获得它的所有返回值。这里所谓的“一系列表达式”在 Lua 中表现为 4 种情况：多重赋值、函数调用时传入的实参列表、table 的构造式和 **return** 语句。接下来将分别列举这几种情况，首先，假设有以下这些函数定义：

```
function foo0 () end                    -- 无返回值
function foo1 () return "a" end         -- 返回1个结果
function foo2 () return "a","b" end     -- 返回2个结果
```

在多重赋值中，若一个函数调用是最后的（或仅有的）一个表达式，那么 Lua 会保留其尽可能多的返回值，用于匹配赋值变量：

```
x,y = foo2()              -- x="a", y="b"
x = foo2()                -- x="a", "b"被丢弃
x,y,z = 10,foo2()         -- x=10, y="a", z="b"
```

如果一个函数没有返回值或者没有返回足够多的返回值，那么 Lua 会用 **nil** 来补充缺失的值：

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

如果一个函数调用不是一系列表达式的最后一个元素，那么将只产生一个值：

```
x,y = foo2(), 20          -- x="a", y=20
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
x,y = foo0(), 20, 30        -- x=nil, y=20, 30 被丢弃
```

当一个函数调用作为另一个函数调用的最后一个（或仅有的）实参时，第一个函数的所有返回值都将作为实参传入第二个函数。这样的例子已经见到很多了，如 print：

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

当 foo2 出现在一个表达式中时，Lua 会将其返回值数量调整为 1。因此在上例最后一行中，只有"a"参与了字符串连接操作。

print 函数可以接受不同数量的实参。例如在 f(g())的调用中，f 具有固定数量的实参，如之前所看到的，Lua 会将 g 的返回值个数调整为 f 的参数个数。

table 构造式可以完整地接收一个函数调用的所有结果，即不会有任何数量方面的调整：

```
t = {foo0()}        -- t = {}   （一个空的table）
t = {foo1()}        -- t = {"a"}
t = {foo2()}        -- t = {"a", "b"}
```

不过，这种行为只有当一个函数调用作为最后一个元素时才会发生，而在其他位置上的函数调用总是只产生一个结果值：

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

最后一种情况是 return 语句，诸如 return f()这样的语句将返回 f 的所有返回值：

```
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))     --  （无返回值）
print(foo(3))     --  （无返回值）
```

也可以将一个函数调用放入一对圆括号中，从而迫使它只返回一个结果：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

请注意 return 语句后面的内容是不需要圆括号的，在该位置上书写圆括号会导致不同的行为。例如 return (f(x))，将只返回一个值，而无关乎 f 返回了几个值。

关于多重返回值还要介绍一个特殊函数——unpack。它接受一个数组作为参数，并从下标 1 开始返回该数组的所有元素：

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

unpack 的一项重要用途体现在“泛型调用（generic call）”机制中。泛型调用机制可以动态地以任何实参来调用任何函数。举例来说，在 ANSI C 中是没有办法编写泛型调用的代码。最多是声明一个能接收变长参数的函数（通过 stdarg.h），或者使用一个函数指针来调用不同的函数。并且在 C 语言中，无法在同一次函数调用中传入动态数量的参数。也就是说，在每次调用函数时必须传入固定数量的参数，并且每个参数都具有确定的类型。然而在 Lua 中就可以做到这点。如果想调用任意函数 f，而所有的参数都在数组 a 中，那么可以这么写：

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

unpack 将返回 a 中所有的值，这些值将作为 f 的实参。举例来说，假设执行：

```
f = string.find
a = {"hello", "ll"}
```

f(unpack(a))将返回 3 和 4，这与直接调用 string.find("hello", "ll")所返回的结果一摸一样。

虽然这个预定义函数 unpack 是用 C 语言编写的，但是仍可以在 Lua 中通过递归实现一样效果：

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

第一次调用时只传入一个实参，此时 i 为 1。然后，函数返回 t[1]与 unpack(t, 2)调用的结果，后者又将返回 t[2]与 unpack(t, 3)的结果，依此类推。直至遇到第一个 nil 元素停止递归。

5.2 变长参数（variable number of arguments）

Lua 中的函数还可以接受不同数量的实参。例如，在调用 print 时可以传入一个、二个或多个实参。虽然 print 是用 C 语言编写的，但也可以用 Lua 编写出这种能接受不同数量实参的函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

下面是一个简单的例子，这个函数返回了所有参数的总和：

```
function add (...)
local s = 0
  for i, v in ipairs{...} do
    s = s + v
  end
return s
end

print(add(3, 4, 10, 25, 12))   --> 54
```

参数表中的 3 个点（...）表示该函数可接受不同数量的实参。当这个函数被调用时，它的所有参数都会被收集到一起。这部分收集起来的实参称为这个函数的“变长参数（variable arguments[①]）”。一个函数要访问它的变长参数时，仍需用到 3 个点（...）。但不同的是，此时这 3 个点是作为一个表达式来使用的。在上例中，表达式{...}表示一个由所有变长参数构成的数组。而函数 add 遍历了该数组，并累加了每个元素。

表达式“...”的行为类似于一个具有多重返回值的函数，它返回的是当前函数的所有变长参数。例如：

```
local a, b = ...
```

上例用第一个和第二个变长参数来初始化这两个局部变量[②]。实际上，还可以通过变长参数来模拟 Lua 中普通的参数传递机制，例如：

```
function foo (a, b, c)
```

可以转换为：

```
function foo (...)
local a, b, c = ...
end
```

对于那些喜爱 Perl 参数传递机制的人来说，可能会更倾向于这第二种形式。

若有这样一个函数：

```
function id (...) return ... end
```

它只是简单地返回调用它时所传入的所有实参，这是一个“多值恒定式（multi-value identity）”函数。下这个函数的行为非常类似于直接调用函数 foo，但在调用 foo 前先调用了 print 打印出其所有的实参：

① 简称 varargs。
② 如果没有一个或二个变长参数，则相应地返回 nil。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
function foo1 (...)
print("calling foo:", ...)
  return foo(...)
end
```

这种技巧对于跟踪某个特定的函数调用很有帮助。

再来看另外一个示例。Lua提供了专门用于格式化文本(string.format)和输出文本(io.write)的函数。很自然地会想到将这两个函数合二为一：

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

注意在3个点前有一个固定参数fmt。具有变长参数的函数同样也可以拥有任意数量的固定参数，但固定参数必须放在变长参数之前。Lua会将前面的实参赋予固定参数，而将余下的实参（如果有的话）视为变长参数。下面列出了一些函数调用及对应的参数值：

```
调用                    参数
fwrite()                fmt = nil, 没有变长参数
fwrite("a")             fmt = "a", 没有变长参数
fwrite("%d%d", 4, 5)    fmt = "%d%d", 变长参数= 4和5
```

通常一个函数在遍历其变长参数时只需使用表达式{...}，这就像访问一个table一样，访问所有的变长参数。然而在某些特殊的情况下，变长参数中可能会包含一些故意传入的nil，那么此时就需要用函数select来访问变长参数了。调用select时，必须传入一个固定实参selector（选择开关）和一系列变长参数。如果selector为数字*n*，那么select返回它的第*n*个可变实参；否则，selector只能为字符串"#"，这样select会返回变长参数的总数。下面的循环演示了如何使用select来遍历一个函数的所有变长参数：

```
for i=1, select('#', ...) do
local arg = select(i, ...)    -- 得到第i个参数
  <循环体>
end
```

特别需要指出的是，select("#", ...)会返回所有变长参数的总数，其中包括nil。

Lua 5.0对于变长参数则有另外一套机制。声明函数的语法是一样的，也是将3个点作为最后一个参数。但Lua 5.0没有提供“...”表达式。而是通过一个隐含的局部table变量“arg”来接受所有的变长参数。这个table还有一个名为“n”的字段，用来记录变长参数的总数。可以用以下代码来模拟老版本中的行为：

```
function foo (a, b, ...)
local arg = {...};  arg.n = select("#", ...)
  <函数体>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
end
```

这套旧机制的缺点在于，每当程序调用了一个具有变长参数的函数时，都会创建一个新的 table。而在新机制中，只有在需要时才会去创建这个用于变长参数访问的 table。

5.3 具名实参（named arguments）

Lua 中的参数传递机制是具有“位置性”的，也就是说在调用一个函数时，实参是通过它在参数表中的位置与形参匹配起来的。第一个实参的值与第一个形参相匹配，依此类推。但有时通过名称来指定实参也是很有用的。为了说明这种有用性，来思考一个函数 os.rename[1]，这个函数用于文件改名。通常会忘记第一个参数是表示新文件名还是旧文件名。因此，会希望这个函数能接受两个具有名称的实参，例如：

```
-- 无效的演示代码
rename(old="temp.lua", new="temp1.lua")
```

Lua 并不直接支持这种语法，但可以通过一种细微的改变来获得相同的效果。主要是将所有实参组织到一个 table 中，并将这个 table 作为唯一的实参传给函数。另外，还需要用到一种 Lua 中特殊的函数调用语法，就是当实参只有一个 table 构造式时，函数调用中的圆括号是可有可无的：

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

另一方面，将 rename 改为只接受一个参数，并从这个参数中获取实际的参数：

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

若一个函数拥有大量的参数，而其中大部分参数是可选的话，这种参数传递风格会特别有用。例如在一个 GUI 库中，一个用于创建新窗口的函数可能会具有许多的参数，而其中大部分都是可选的，那么最好使用具名实参：

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

Window 函数可以根据要求检查一些必填参数，或者为某些参数添加默认值。假设“_Window”才是真正用于创建新窗口的函数，它要求所有参数以正确的次序传入，那么

① 来自于标准库中的 OS 库。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Window 函数可以这么写：

```
function Window (options)
-- 检查必要的参数
  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

-- 其他参数都是可选的
  _Window(options.title,
          options.x or 0,     -- 默认值
          options.y or 0,     -- 默认值
          options.width, options.height,
          options.background or "white",   -- 默认值
          options.border      -- 默认值为 false (nil)
         )
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第6章 深入函数

在 Lua 中，函数是一种“第一类值（First-Class Value）”，它们具有特定的词法域（Lexical Scoping）。

“第一类值”是什么意思呢？这表示在 Lua 中函数与其他传统类型的值（例如数字和字符串）具有相同的权利。函数可以存储到变量中（无论全局变量还是局部变量）或 table 中，可以作为实参传递给其他函数，还可以作为其他函数的返回值。

“词法域”是什么意思呢？这是指一个函数可以嵌套在另一个函数中，内部的函数可以访问外部函数中的变量[①]。接下来就会看到，这项听似平凡的特性将给语言带来极大的能力。因为它允许在 Lua 中应用各种函数式语言（functional-language）中的强大编程技术。即使对函数式编程毫无了解，那也不妨花点时间来探索一下这些技术，因为它们可以使程序变得更加小巧简单。

在 Lua 中有一个容易混淆的概念是，函数与所有其他值一样都是匿名的，即它们都没有名称。当讨论一个函数名时（例如 print），实际上是在讨论一个持有某函数的变量。这与其他变量持有各种值一个道理，可以以多种方式来操作这些变量。下面这个示例足以说明这点：

```
a = {p = print}
a.p("Hello World")     --> Hello World
print = math.sin       -- 'print'现在引用了正弦函数
a.p(print(1))          --> 0.841470
sin = a.p              -- 'sin'现在引用了print 函数
sin(10, 20)            --> 10     20
```

关于这项特性，接下还会看到更多有用的应用。

如果说函数是“值”的话，那是否可以说函数就是由一些表达式创建的呢？是的，事实上在 Lua 中最常见的是函数编写方式，诸如：

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

只是一种所谓的“语法糖”而已。也就是说，这只是以下代码的一种简化书写形式：

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

因此，一个函数定义实际就是一条语句（更准确地说是一条赋值语句），这条语句创建了

① 这也意味着 Lua 完全可以包含“λ演算（Lambda Calculus）”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一种类型为“函数”的值，并将这个值赋予一个变量。可以将表达式“function (x) <body> end”视为一种函数的构造式，就像 table 的构造式{}一样。将这种函数构造式的结果称为一个“匿名函数”。虽然一般情况下，会将函数赋予全局变量，即给予其一个名称。但在某些特殊情况中，仍会需要用到匿名函数。下面来看几个例子。

table 库提供了一个函数 table.sort，它接受一个 table 并对其中的元素排序。像这种函数就必须支持各种各样可能的排序准则，例如升序还是降序、按数字顺序还是按字符顺序或者按 table 中 key 的顺序等。sort 函数并没有提供所有这些排序准则，而是提供了一个可选的参数，所谓“次序函数（order function）”。这个函数接受两个元素，并返回在有序情况下第一个元素是否应排在第二个元素之前。举例来说，假设有一个 table 内容如下：

```
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"},
}
```

如果想以 name 字段、按反向的字符顺序来对这个 table 排序的话，只需这么写：

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

可见匿名函数在这条语句中就显示出了极好的便捷性。

像 sort 这样的函数，接受另一个函数作为实参的，称其是一个“高阶函数（higher-order function）”。高阶函数是一种强大的编程机制，应用匿名函数来创建高阶函数所需的实参则可以带来更大的灵活性。但请记住，高阶函数并没有什么特权。Lua 强调将函数视为“第一类值”，所以高阶函数只是一种基于该观点的应用体现而已。

为了进一步演示高阶函数的应用，将再写一个关于导数的高阶函数。在一个非形式化的定义中，一个函数 f 在点 x 的导数就是(f(x+d)-f(x))/d，其中 d 趋向于无限小。可以用如下方式来近似地计算这个函数 f 的导数：

```
function derivative (f, delta)
  delta = delta or 1e-4
  return function (x)
           return (f(x + delta) - f(x))/delta
         end
end
```

对于特定的函数 f 调用 derivative(f)将（近似地）返回其导数，例如：

```
c = derivative(math.sin)
print(math.cos(10), c(10))
  -->  -0.83907152907645   -0.83904432662041
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于函数在 Lua 中是一种“第一类值”，所以不仅可以将其存储在全局变量中，还可以存储在局部变量甚至 table 的字段中。接下来还将看到，将函数存储在 table 的字段中可以支持许多 Lua 的高级应用，例如模块（module）和面向对象编程。

6.1 closure（闭合函数）

若将一个函数写在另一个函数之内，那么这个位于内部的函数便可以访问外部函数中的局部变量，这项特征称之为“词法域”。虽然这种可见性准则听上去很容易理解，但在实际应用中可能还会存在诸多问题。然而在编程语言中，词法域与“第一类的”函数是两项极其有用的概念，但支持它们的语言却不多。

先来看一个简单的例子。假设有一个学生姓名的列表和一个对应于每个姓名的年级列表，需要根据每个学生的年级来对他们的姓名进行排序（由高到低）。可以这么做：

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

现在假设要单独创建一个函数来做这项工作：

```
function sortbygrade (names, grades)
table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2]      -- 比较年级
  end)
end
```

在上例中有一点很有趣，传递给 sort 的匿名函数可以访问参数 grades，而 grades 是外部函数 sortbygrade 的局部变量[①]。在这个匿名函数内部，grades 既不是全局变量也不是局部变量，将其称为一个“非局部的变量（non-local variable）”[②]。

为什么在 Lua 中允许这种访问呢？原因在于函数是“第一类值”。考虑以下代码：

```
function newCounter ()
local i = 0
  return function ()      -- 匿名函数
          i = i + 1
         return i
        end
```

① 译注：参数（形参）也是一种局部变量。
② 由于一些历史原因，在 Lua 中“非局部的变量”也称为 upvalue。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
end

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

在这段代码中，匿名函数访问了一个“非局部的变量”i，该变量用于保持一个计数器。初看上去，由于创建变量 i 的函数（newCounter）已经返回，所以之后每次调用匿名函数时，i 都应是已超出了作用范围的。但其实不然，Lua 会以 closure 的概念来正确地处理这种情况。简单地讲，一个 closure 就是一个函数加上该函数所需访问的所有“非局部的变量”。如果再次调用 newCounter，那么它会创建一个新的局部变量 i，从而也将得到一个新的 closure：

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

因此 c1 和 c2 是同一个函数所创建的两个不同的 closure，它们各自拥有局部变量 i 的独立实例。

从技术上讲，Lua 中只有 closure，而不存在“函数”。因为，函数本身就是一种特殊的 clousure[①]。不过只要不会引起混淆，仍将采用术语“函数”来指代 closure。

在许多场合中 closure 都是一种很有价值的工具。就像之前所看到的，它们可作为 sort 这类高阶函数的参数。closure 对于那些创建其他函数的函数也很有价值，例如前例中的 newCounter。这种机制使 Lua 程序可以混合那些在函数式编程世界中久经考验的编程技术。另外，closure 对于回调函数也很有用。这里有一个典型的例子，假设有一个传统的 GUI 工具包可以创建按钮，每个按钮都有一个回调函数，每当用户按下按钮时 GUI 工具包都会调用这些回调函数。再假设，基于此要做一个十进制计算器，其中需要 10 个数字按钮。会发现这些按钮之间的区别其实并不大，仅需在按下不同按钮时做一些稍微不同的操作就可以了。那么可以使用以下函数来创建这些按钮：

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

① 译注：原文为“Technically speaking, what is a value in Lua is the closure, not the function. The function itself is just a prototype for closures.”。这段叙述可能会让初学者感到困惑，下面来简单解释一下。
closure 是指一个函数及一系列这个函数会访问到“非局部的变量（或 upvalue）”。因此若一个 closure 没有那些会访问的“非局部的变量”，那它就是一个传统概念中的“函数”。所以作者在这里想表示的应该是先前一直所见闻的各种“函数”其实都是 closure 的一种特殊情况。
另外在 Lua 的 C API 中，所有关于“Lua 中的函数”的核心 API 都是以 closure（而非 function）来命名的，也可视为是这一观点的延续。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
        }
end
```

在本例中假设 Button 是工具包中一个用于创建新按钮的函数，label 是按钮的标签，action 是回调 closure，每当按钮按下时就会调用它。回调一般发生在 digitButton 函数执行完后，那时局部变量 digit 已经超出了作用范围，但 closure 仍可以访问到这个变量。

closure 在另一种情况中也非常有用。例如在 Lua 中函数是存储在普通变量中的，因此可以轻易地重新定义某些函数，甚至是重新定义那些预定义的函数。这也正是 Lua 相当灵活的原因之一。通常当重新定义一个函数的时候，需要在新的实现中调用原来的那个函数。举例来说，假设要重新定义函数 sin，使其参数能使用角度来代替原先的弧度。那么这个新函数就必须得转换它的实参，并调用原来的 sin 函数完成真正的计算。这段代码可能是这样的：

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

还有一种更彻底的做法是这样的：

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

将老版本的 sin 保存到了一个私有变量中，现在只有通过新版本的 sin 才能访问到它了。可以使用同样的技术来创建一个安全的运行环境，即所谓的“沙盒（sandbox）”。当执行一些未受信任的代码时就需要一个安全的运行环境，例如在服务器中执行那些从 Internet 上接收到的代码。举例来说，如果要限制一个程序访问文件的话，只需使用 closure 来重定义函数 io.open 就可以了。

```
do
local oldOpen = io.open
  local access_OK = function (filename, mode)
    <检查访问权限>
  end
  io.open = function (filename, mode)
    if access_OK(filename, mode) then
      return oldOpen(filename, mode)
    else
      return nil, "access denied"
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    end
  end
end
```

这个示例的精彩之处在于，经过重新定义后，一个程序就只能通过新的受限版本来调用原来那个未受限的 open 函数了。示例将原来不安全的版本保存到 closure 的一个私有变量中，从而使得外部再也无法直接访问到原来的版本了。通过这种技术，可以在 Lua 的语言层面上就构建出一个安全的运行环境，且不失简易性和灵活性。相对于提供一套大而全的解决方案，Lua 提供的则是一套“元机制（meta-mechanism）”，因此可以根据特定的安全需要来创建一个安全的运行环境。

6.2 非全局的函数（non-global function）

由于函数是一种“第一类值”，因此一个显而易见的推论就是，函数不仅可以存储在全局变量中，还可以存储在 table 的字段中和局部变量中。

前面讲到了几个将函数存储在 table 字段中的示例，大部分 Lua 库也采用了这种机制（例如 io.read、math.sin）。若要在 Lua 中创建这种函数，只需将常规的函数语法与 table 语法结合起来使用即可：

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

当然，还可以使用构造式：

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

除了这些之外，Lua 还提供了另一种语法来定义这类函数：

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

只要将一个函数存储到一个局部变量中，即得到了一个“局部函数（local function）”，也就是说该函数只能在某个特定的作用域中使用。对于“程序包（package）”而言，这种函数定义是非常有用的。因为 Lua 是将每个程序块（chunk）作为一个函数来处理的，所以在一个程序块中声明的函数就是局部函数，这些局部函数只在该程序块中可见。词法域确保了程序包中的其他函数可以使用这些局部函数：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local f = function (<参数>)
   <函数体>
end

local g = function (<参数>)
   <一些代码>
f()              -- 'f'在这里是可见的
   <一些代码>
end
```

对于这种局部函数的定义，Lua 还支持一种特殊的“语法糖”：

```
local function f (<参数>)
   <函数体>
end
```

在定义递归的局部函数时，还有一个特别之处需要注意。像下面这种采用了基本函数定义语法的代码多数是错误的：

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

当 Lua 编译到函数体中调用 fact(n-1)的地方时，由于局部的 fact 尚未定义完毕，因此这句表达式其实是调用了一个全局的 fact，而非此函数自身。为了解决这个问题，可以先定义一个局部变量，然后再定义函数本身：

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

现在函数中的 fact 调用就表示了局部变量。即使在函数定义时，这个局部变量的值尚未完成定义，但之后在函数执行时，fact 则肯定已经拥有了正确的值。

当 Lua 展开局部函数定义的“语法糖”时，并不是使用基本函数定义语法。而是对于局部函数定义：

```
local function foo (<参数>)  <函数体>  end
```

Lua 将其展开为：

```
local foo
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
foo = function (<参数>)  <函数体>  end
```

因此，使用这种语法来定义递归函数不会产生错误：

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

当然，这个技巧对于间接递归的函数而言是无效的。在间接递归的情况中，必须使用一个明确的前向声明（Forward Declaration）：

```
local f, g    -- 前向声明

function g ()
  <一些代码>  f()  <一些代码>
end

function f ()
  <一些代码>  g()  <一些代码>
end
```

注意，别把第二个函数定义写为“local function f”。如果那样的话，Lua 会创建一个全新的局部变量 f，而将原来声明的 f（函数 g 中所引用的那个）置于未定义的状态。

6.3 正确的尾调用（proper tail call）

Lua 中的函数还有一个有趣的特征，那就是 Lua 支持“尾调用消除（tail-call elimination）”。[①]

所谓“尾调用（tail call）”就是一种类似于 goto 的函数调用。当一个函数调用是另一个函数的最后一个动作时，该调用才算是一条“尾调用”。举例来说，以下代码中对 g 的调用就是一条“尾调用”：

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

也就是说，当 f 调用完 g 之后就再无其他事情可做了。因此在这种情况中，程序就不需要返回那个“尾调用”所在的函数了。所以在“尾调用”之后，程序也不需要保存任何关于该函数的栈（stack）信息了。当 g 返回时，执行控制权可以直接返回到调用 f 的那个点上。有一些语言实现（例如 Lua 解释器）可以得益于这个特点，使得在进行“尾调用”时不耗费任何栈空间。将这种实现称为支持“尾调用消除”。

① 这也表示 Lua 能正确地处理函数结尾时的递归调用。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于“尾调用”不会耗费栈空间，所以一个程序可以拥有无数嵌套的“尾调用”。举例来说，在调用以下函数时，传入任何数字作为参数都不会造成栈溢出：

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

有一点需要注意的是，当想要受益于“尾调用消除”时，务必要确定当前的调用是一条“尾调用”。判断的准则就是“一个函数在调用完另一个函数之后，是否就无其他事情需要做了”。有一些看似是“尾调用”的代码，其实都违背了这条准则。举例来说，在下面的代码中，对 g 的调用就不是一条“尾调用”：

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

这个示例的问题在于，当调用完 g 后，f 并不能立即返回，它还需要丢弃 g 返回的临时结果。类似地，以下所有调用也都不符合上述准则：

```
return g(x) + 1          -- 必须做一次加法
return x or g(x)         -- 必须调整为一个返回值
return (g(x))            -- 必须调整为一个返回值
```

在 Lua 中，只有“return <func>(<args>)”这样的调用形式才算是一条“尾调用”。Lua 会在调用前对<func>及其参数求值，所以它们可以是任意复杂的表达式。举例来说，下面的调用就是一条“尾调用”：

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

在之前提到了，一条“尾调用”就好比是一条 goto 语句。因此，在 Lua 中“尾调用”的一大应用就是编写“状态机（state machine）”。这种程序通常以一个函数来表示一个的状态，改变状态就是 goto（或调用）到另一个特定的函数。举一个简单的迷宫游戏的例子来说明这个问题。例如，一个迷宫有几间房间，每间房间中最多有东南西北 4 扇门。用户在每一步移动中都需要输入一个移动的方向。如果在某个方向上有门，那么用户可以进入相应的房间；不然，程序就打印一条警告。游戏目标就是让用户从最初的房间走到最终的房间。

这个游戏就是一种典型的状态机，其中当前房间就是一个状态。可以将迷宫中的每间房间实现为一个函数，并使用“尾调用”来实现从一间房间移动到另一间房间。在以下代码中，实现一个具有 4 间房间的迷宫：

```
function room1 ()
  local move = io.read()
  if move == "south" then return room3()
  elseif move == "east" then return room2()
  else
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    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("congratulations!")
end
```

通过调用初始房间来开始这个游戏：

```
room1()
```

若没有“尾调用消除”的话，每次用户的移动都会创建一个新的栈层（stack level），移动若干步之后就有可能会导致栈溢出。而“尾调用消除”则对用户移动的次数没有任何限制。这是因为每次移动实际上都只是完成一条 goto 语句到另一个函数，而非传统的函数调用。

对于这个简单的游戏而言，或许会觉得将程序设计为数据驱动的会更好一点，其中将房间和移动记录在一些 table 中。不过，如果游戏中的每间房间都有各自特殊的情况的话，采用这种状态机的设计则更为合适。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 7 章　迭代器与泛型 for

本章将介绍如何编写适用于泛型 for 的迭代器（Iterator）。先从简单的迭代器入手，然后将学习如何利用泛型 for 的各种能力来编写更简单、更有效率的迭代器。

7.1　迭代器与 closure

所谓“迭代器”就是一种可以遍历（iterate over）一种集合中所有元素的机制。在 Lua 中，通常将迭代器表示为函数。每调用一次函数，即返回集合中的“下一个”元素。

每个迭代器都需要在每次成功调用之间保持一些状态，这样才能知道它所在的位置及如何步进到下一个位置。closure 对于这类任务提供了极佳的支持，一个 closure 就是一种可以访问其外部嵌套环境中的局部变量的函数。对于 closure 而言，这些变量就可用于在成功调用之间保持状态值，从而使 closure 可以记住它在一次遍历中所在的位置。当然，为了创建一个新的 closure，还必须创建它的这些“非局部的变量（non-local variable）”。因此一个 closure 结构通常涉及到两个函数：closure 本身和一个用于创建该 closure 的工厂（factory）函数。

作为示例，来为列表编写一个简单的迭代器。与 ipairs 不同的是该迭代器并不是返回每个元素的索引，而是返回元素的值：

```
function values (t)
  local i = 0
  return function ()  i = i + 1; return t[i]  end
end
```

在本例中，values 就是一个工厂。每当调用这个工厂时，它就创建一个新的 closure（即迭代器本身）。这个 closure 将它的状态保存在其外部变量 t 和 i 中。每当调用这个迭代器时，它就从列表 t 中返回下一个值。直到最后一个元素返回后，迭代器就会返回 **nil**，以此表示迭代的结束。

可以在一个 **while** 循环中使用这个迭代器：

```
t = {10, 20, 30}
iter = values(t)                -- 创建迭代器
while true do
  local element = iter()        -- 调用迭代器
  if element == nil then break end
```

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  print(element)
end
```

然而使用泛型 **for** 则更为简单。接下来会发现，它正是为这种迭代而设计的：

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

泛型 **for** 为一次迭代循环做了所有的薄记工作。它在内部保存了迭代器函数，因此不再需要 iter 变量。它在每次新迭代时调用迭代器，并在迭代器返回 **nil** 时结束循环。在下一节中将会看到泛型 **for** 能够做更多的事情。

下面是一个更高级的示例，展现了一个可以遍历当前输入文件中所有单词的迭代器——allwords。为了完成这样的遍历，需要保持两个值：当前行的内容（变量 line）及在该行中所处的位置（变量 pos）。有了这些信息，就可以不断产生下一个单词了。这个迭代器函数的主要部分就是 string.find 的调用。此调用会在当前行中，以当前位置作为起始来搜索一个单词。使用模式（pattern）'%w+'来描述一个“单词”，它用于匹配一个或多个的文字/数字字符（alphanumeric characters）。如果 string.find 找到了一个单词，迭代器就会将当前位置更新为该单词之后的第一个字符，并返回该单词[①]。否则，它就读取新的一行并反复这个搜索过程。若没有剩余的行，则返回 **nil**，以此表示迭代的结束。

尽管迭代器本身具有复杂性，但 allwords 的使用还是很简明易懂的：

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

对于迭代器而言，一种常见的情况就是：编写迭代器本身或许不太容易，但使用它们却是很容易的。这也不会成为一个大问题，因为通常使用 Lua 编程的最终用户不会去定义迭代器，而只是使用那些程序提供的迭代器。

```
function allwords ()
  local line = io.read()  -- 当前行
  local pos = 1           -- 一行中的当前位置
  return function ()       -- 迭代器函数
    while line do          -- 若为有效的行内容就进入循环
      local s, e = string.find(line, "%w+", pos)
      if s then            -- 是否找到一个单词
        pos = e + 1        -- 该单词的下一个位置
        return string.sub(line, s, e)     -- 返回该单词
      else
```

① string.sub 用于提取 line 的给定位置上的子字符串，在 20.2 节中对此有详细叙述。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
      line = io.read()  -- 没有找到单词，尝试下一行
      pos = 1           -- 在第一个位置上重新开始
    end
  end
  return nil            -- 没有其余行了，遍历结束
end
end
```

7.2　泛型 for 的语义

上述的那些迭代器都有一个缺点，就是需要为每个新的循环都创建一个新的 closure。对于大多数情况而言，这或许不会有什么问题。例如在之前的 allwords 迭代器中，创建一个 closure 的代价相对于读取整个文件的代价而言，几乎可以忽略不计。但是，在另外一些情况下，这样的开销就不太容易接受了。因此，在这类情况中，希望能通过泛型 **for** 的自身来保存迭代器状态。在本节中会详细说明泛型 **for** 的这种保存状态的机制。

泛型 **for** 在循环过程内部保存了迭代器函数。实际上它保存着 3 个值：一个迭代器函数、一个恒定状态（invariant state）和一个控制变量（control variable）。接下来将对此进行详细说明。

泛型 **for** 的语法如下：

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

其中，<var-list>是一个或多个变量名的列表，以逗号分隔；*<exp-list>*是一个或多个表达式的列表，同样以逗号分隔。通常表达式列表只有一个元素，即一句对迭代器工厂的调用。例如，以下代码：

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

其中变量列表是“k, v”，表达式列表只有一个元素 pairs(t)。一般来说变量列表中也只有一个变量，例如下面这个循环：

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

变量列表的第一元素称为“控制变量”。在循环过程中该值决不会为 **nil**，因为当它为 **nil** 时循环就结束了。

for 做的第一件事情是对 **in** 后面的表达式求值。这些表达式应该返回 3 个值供 **for** 保存：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

迭代器函数、恒定状态和控制变量的初值。这里有点类似于多重赋值，即只有最后一个表达式才会产生多个结果，并且只会保留前 3 个值，多余的值会被丢弃；而不足的话，将以 **nil** 补足[①]。

在初始化步骤之后，**for** 会以恒定状态和控制变量来调用迭代器函数[②]。然后 **for** 将迭代器函数的返回值赋予变量列表中的变量。如果第一个返回值[③]为 **nil**，那么循环终止。否则，**for** 执行它的循环体，随后再次调用迭代器函数，并重复这个过程。

更明确地说，以下语句：

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

等价于以下代码：

```
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
```

因此，假设迭代器函数为 f，恒定状态为 s，控制变量的初值为 a_0。那么在循环过程中控制变量的值依次为 $a_1 = f(s, a_0)$、$a_2 = f(s, a_1)$，*依此*类推，直至 a_i 为 **nil** 结束循环。如果 **for** 还有其他变量，那么它们也会在每次调用 *f* 后获得额外的值。

7.3 无状态的迭代器

所谓“无状态的迭代器”，正如其名所暗示的那样，就是一种自身不保存任何状态的迭代器。因此，我们可以在多个循环中使用同一个无状态的迭代器，避免创建新的 closure 开销。

在每次迭代中，**for** 循环都会用恒定状态和控制变量来调用迭代器函数。一个无状态的迭代器可以根据这两个值来为下次迭代生成下一个元素。这类迭代器的一个典型例子就是 ipairs，它可以用来迭代一个数组的所有元素：

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

① 当使用较为简单的迭代器时，工厂只返回一个迭代器函数，因此恒定状态和控制变量就为 nil 了。
② 从 for 构建的角度来看，恒定状态的内容与 for 本身是完全无关的。for 只是保存了初始化中返回的值，并在调用迭代器函数时传入该值。
③ 即赋予控制变量的那个值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在这里，迭代的状态就是需要遍历的 table（一个恒定状态，它不会在循环中改变）及当前的索引值（控制变量）。ipairs（工厂）和迭代器都非常简单，在 Lua 中就可以编写出来：

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

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

当 Lua 调用 **for** 循环中的 ipairs(a)时，它会获得 3 个值：迭代器函数 iter、恒定状态 a 和控制变量的初值 0。然后 Lua 调用 iter(a, 0)，得到 1, a[1][①]。在第二次迭代中，继续调用 iter(a,1)，得到 2, a[2]，依此类推，直至得到第一个 nil 元素为止。

函数 pairs 与 ipairs 类似，也是用于遍历一个 table 中的所有元素。不同的是，它的迭代器函数是 Lua 中的一个基本函数 next。

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

在调用 next(t, k)时，k 是 table t 的一个 key。此调用会以 table 中的任意次序返回一组值：此 table 的下一个 key，及这个 key 所对应的值。而调用 next(t, nil)时，返回 table 的第一组值。若没有下一组值时，next 返回 **nil**。

有些用户喜欢不通过 pairs 调用而直接使用 next：

```
for k, v in next, t do
  <loop body>
end
```

记住，Lua 会自动将 **for** 循环中表达式列表的结果调整为 3 个值。因此上例中得到了 next、t 和 **nil**，这也正与调用 pairs(t)的结果完全一致。

关于无状态迭代器的另一个有趣例子是一种可以遍历链表的迭代器[②]。

```
local function getnext (list, node)
  if not node then
    return list
```

① 假设 a[1]不为 nil 的话。
② 之前也提到过，在 Lua 中很少用到链表，不过有时还是会需要用它们来解决一些特殊问题的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  else
      return node.next
    end
  end

  function traverse (list)
    return getnext, list, nil
  end
```

这里使用了一个技巧就是将链表的头结点作为恒定状态（traverse 返回的第二个值），而将当前结点作为控制变量。第一次调用迭代器函数 getnext 时，node 为 **nil**，因此函数返回 list 作为第一个结点。在后续调用中 node 不再为 **nil** 了，所以迭代器如期望的那样返回 node.next。对于此迭代器的使用则非常简单：

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

for node in traverse(list) do
  print(node.val)
end
```

7.4 具有复杂状态的迭代器

通常，迭代器需要保存许多状态，可是泛型 for 却只提供的一个恒定状态和一个控制变量用于状态的保存。一个最简单的解决方法就是使用 closure。或者还可以将迭代器所需的所有状态打包为一个 table，保存在恒定状态中。一个迭代器通过这个 table 就可以保存任意多的数据了。此外，它还能在循环过程中改变这些数据。虽然，在循环过程中恒定状态总是同一个 table（故称之“恒定”），但这个 table 的内容却可以发生改变。由于这种迭代器可以在恒定状态中保存所有数据，所以它们通常可以忽略泛型 **for** 提供的第二个参数（控制变量）。

作为该技术的一个示例，将重写 allwords 迭代器，这个迭代器可以遍历当前输入文件中的所有单词。这次将它的状态保存到一个 table 中，这个 table 具有两个字段：line 和 pos。

迭代的起始函数比较简单，它只需返回迭代器函数和初始状态：

```
local iterator   -- 在后面定义

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

iterator 函数才开始真正的工作：

```
function iterator (state)
  while state.line do          -- 若为有效的行内容就进入循环
    -- 搜索下一个单词
    local s, e = string.find(state.line, "%w+", state.pos)
    if s then                  -- 找到了一个单词?
      -- 更新下一个位置（到这个单词之后）
      state.pos = e + 1
      return string.sub(state.line, s, e)
    else                       -- 没有找到单词
      state.line = io.read() -- 尝试下一行…
      state.pos = 1            -- 从第一个位置开始
    end
  end
  return nil                   -- 没有更多行了，结束循环
end
```

尽可能地尝试编写无状态的迭代器，那些迭代器将所有状态保存在 **for** 变量中，不需要在开始一个循环时创建任何新的对象。如果迭代器无法套用这个模型，那么就应该尝试使用 closure。closure 显得更加优雅一点，通常一个基于 closure 实现的迭代器会比一个使用 table 的迭代器更为高效。这是因为，首先创建一个 closure 就比创建一个 table 更廉价，其次访问“非局部的变量”也比访问 table 字段更快。以后会看到另一种使用协同程序（coroutine）编写迭代器的方式，这种方式是功能最强的，但稍微有一点开销。

7.5　真正的迭代器

“迭代器”这个名称多少有点误导的成分。因为迭代器并没有做实际的迭代，真正做迭代的是 **for** 循环。而迭代器只是为每次迭代提供一些成功后的返回值。或许，更准确地应称其为“生成器（generator）”。不过“迭代器”这个名称已在其他语言中被广泛使用，例如 Java。

还有一种创建迭代器的方式就是，在迭代器中做实际的迭代操作。当使用这种迭代器时，就不需要写一个循环了。相反，需要一个描述了在每次迭代时需要做什么的参数，并以此参数来调用迭代器。更确切地说，迭代器接受一个函数作为参数，并在其内部的循环中调用这个函数。

在此列举一个更具体的例子，就使用这种风格来再次重写 allwords 迭代器：

```
function allwords (f)
for line in io.lines() do
   for word in string.gmatch(line, "%w+") do
     f(word)    -- call the function
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    end
  end
end
```

使用这个迭代器时，需要传入一个描述循环体的函数。例如，只想打印每个单词，那么可以使用 print：

```
allwords(print)
```

通常，还可以使用一个匿名函数作为循环体。例如，以下代码计算了单词“hello”在输入文件中出现的次数：

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

同样的任务若采用之前的迭代器风格也不是很困难的：

```
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”在老版本的 Lua 中曾非常流行，那时语言还没有 **for** 语句。那它们比之生成器风格的迭代器又如何呢？这两种风格都有大致相同的开销，即每次迭代都有一次函数调用。编写真正的迭代器相对比较容易[①]。不过，生成器风格的迭代器则更具灵活性。这种灵活性体现在两方面，首先它允许两个或多个并行的迭代过程[②]，其次它允许在迭代体中使用 **break** 和 **return** 语句。对于真正的迭代器来说，**return** 语句只能从匿名函数中返回，而并不能从做迭代的函数中返回。综上所述，我一般更喜爱生成器。

① 使用协同程序来编写迭代器则更为简易。
② 例如，考虑要逐个单词地比较两个文件，就需要同时遍历两个文件。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 8 章　编译、执行与错误

尽管将 Lua 称为是一种解释型的语言，但 Lua 确实允许在运行源代码前，先将源代码预编译为一种中间形式[1]。听上去“编译”似乎不应在一种解释型语言的范畴之列。其实，区别解释型语言的主要特征并不在于是否能编译它们，而是在于编译器是否是语言运行时库的一部分，即是否有能力（并且轻易地）执行动态生成的代码。可以说正是因为存在了诸如 dofile 这样的函数，才可以将 Lua 称为是一种解释型的语言。

8.1 编　　译

前面已经提到了 dofile 函数，它是一种内置的操作，用于运行 Lua 代码块。但实际上 dofile 是一个辅助函数，loadfile 才做了真正核心的工作。类似于 dofile，loadfile 会从一个文件加载 Lua 代码块，但它不会运行代码，只是编译代码，然后将编译结果作为一个函数返回。此外，与 dofile 不同的还有 loadfile 不会引发错误，它只是返回错误值并不处理错误。一般 dofile 可以这样来定义：

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

注意，如果 loadfile 失败，那么其中 assert 就会引发一个错误。

对于简单任务而言，dofile 非常便捷，它在一次调用中做完了整件事情。然而 loadfile 更灵活，在发生错误的情况中，loadfile 会返回 **nil** 及错误消息，这便可以按自定义的方式来处理错误。此外，如果需要多次运行一个文件，那么只需在调用一次 loadfile 后，多次调用它的返回结果就可以了。相对于多次调用 dofile 来说，由于只编译一次文件，开销就小得多了。

函数 loadstring 与 loadfile 类似，不同之处在于它是从一个字符串中读取代码，而非从文件读取。例如，以下代码：

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

f 就变成了一个函数，每次调用时就执行“i = i + 1”：

① 这也不是什么重大突破，许多解释型的语言都可以这么做。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

loadstring 的功能是非常强大的，但应该谨慎使用。因为相对于其他功能而言，它也是一个开销较大的函数，并且可能会导致难以理解的代码。在决定使用它之前，请先确定一时间已找不到更简单的方法可以解决某个问题。

如果想塑造一个便捷但略为粗糙的 dostring（完成加载并运行代码），那么直接调用 loadstring 的返回值即可：

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

不过，如果代码中有语法错误，loadstring 就会返回 **nil**，那么最终的错误消息可能就会是“attempt to call a nil value（试图调用一个 nil 值）”。为了更清楚地显示错误消息，可以使用 assert：

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

一般将 loadstring 用于字面字符串是没有意义的。例如：

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

基本上就等价于：

```
f = function () i = i + 1 end
```

但第二段代码快得多，因为它只在编译对应程序块时被编译了一次。而第一段代码却在每次调用 loadstring 时都被重新编译。

由于 loadstring 在编译时不涉及词法域（lexical scoping），所以上述两段代码并不等价。为了看清楚这种区别，稍微修改一下上面的例子：

```
i = 32
local i = 0
f = loadstring("i = i + 1; print(i)")
g = function () i = i + 1; print(i) end
f()             --> 33
g()             --> 1
```

函数 g 如预期地操作了局部的 i，但是 f 操作的却是全局的 i，这是因为 loadstring 总是在全局环境中编译它的字符串。

loadstring 最典型的用处是执行外部代码，也就是那些位于程序之外的代码。例如，若让用户来参与一个函数的定义，那么这时就需要让用户输入函数代码，然后调用 loadstring 来对其求值。值得注意的是，loadstring 的期望输入是一个程序块，也就是一系列的语句。如果需要对一个表达式求值，则必须在其之前添加 **return**，这样才能构成一条语句，返回表达式的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

值。如下例：

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

由于 loadstring 返回的函数就是一个正规的函数，因此可以多次调用它：[①]

```
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
```

若对 loadfile 和 loadstring 作进一步研究，会发现在 Lua 中其实有一个真正的原始函数 load。loadfile 和 loadstring 分别从文件和字符串中读取程序块，load 则接收一个“读取器函数（reader function）”，并在内部调用它来获取程序块。读取器函数可以分几次返回一个程序块，load 会反复调用它，直到它返回 **nil**（表示程序块结束）为止。一般很少使用 load，只有当程序块不在文件中[②]，或者程序块过大而无法放入内存时[③]，才会用到它。

Lua 将所有独立的程序块视为一个匿名函数的函数体，并且该匿名函数还具有可变长实参（variable number of arguments）。例如，loadstring(“a = 1”)返回的结果等价于以下表达式：

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

与所有其他函数一样，程序块中可以声明局部变量：

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

通过这个特性，就可以重写那个用户参与的示例，在其中避免使用全局变量 x：

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

在程序块的开头添加了"local x = ..."，用来声明了一个局部变量 x。然后在调用 f 时传入实

① 函数 string.rep 根据指定次数复制一个字符串。
② 例如，动态创建或者需要从特定源头读取的代码。
③ 如果可以放入内存，那么可以使用 loadstring。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

参 i，这样 i 就变成了“变长实参（varargs）”表达式（...）的值了。

load 函数不会引发错误。在错误情况中，load 会返回 **nil** 及一条错误消息：

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

此外，这些函数也不会带来任何副作用。它们只是将程序块编译为一种中间表示，然后将结果作为一个匿名函数来返回。常见的误解是认为加载了一个程序块，也就是定义了其中的函数。其实在 Lua 中，函数定义是一种赋值操作。也就是说，它们是在运行时才完成的操作。例如，假设有一个文件 foo.lua 如下：

```
function foo (x)
  print(x)
end
```

执行以下代码：

```
f = loadfile("foo.lua")
```

在此之后，函数 foo 就完成编译了，但是还没有定义它。为了定义它，必须执行以下程序块：

```
print(foo)    --> nil
f()           -- 定义'foo'
foo("ok")     --> ok
```

若需要在一个商业品质的程序中执行外部代码，那么还应该处理加载程序块时报告的任何错误。此外，如果代码是不可信任的话，可能还需要在一个保护环境中执行这些代码，以防止各种问题产生。

8.2 C 代 码

与 Lua 代码不同的是，C 代码需要在使用前先链接入一个应用程序。在大多数主流系统中，达成这种链接最简单的方法是动态链接机制。不过，动态链接却不是 ANSI C 标准的一部分，也就是说不存在任何可移植的方案来实现它。

Lua 通常不会包含任何无法通过 ANSI C 来实现的机制。不过，动态链接却有些不同。将其视为所有其他机制的母机制，只要拥有它，就可以动态地加载任何其他不在 Lua 中的机制了。因此，对于这项特殊情况 Lua 打破了其对于可移植性的准则，为几种平台实现了一套动态链接机制。标准实现支持的平台有 Windows、Mac OS X、Linux、FreeBSD、Solaris 及其他一些 UNIX 实现。除此之外，要在其他平台上实现这种机制应该也不会很难，请参阅相

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

关文档。[1]

Lua 提供的所有关于动态链接的功能都聚集在一个函数中，即 package.loadlib。该函数有两个字符串参数：动态库的完整路径和一个函数名称。所以，典型的调用代码就像这样：

```
local path = "/usr/local/lib/lua/5.1/socket.so"
local f = package.loadlib(path, "luaopen_socket")
```

loadlib 函数加载指定的库，并将其链接入 Lua。不过，它并没有调用库中的任何函数。相反，它将一个 C 函数作为 Lua 函数返回。如果在加载库或查找初始化函数时发生任何错误，loadlib 返回 **nil** 及一条错误消息。

loadlib 是一个非常底层的函数。必须提供库的完整路径及正确的函数名称[2]。通常使用 require 来加载 C 程序库，这个函数会搜索指定的库，然后用 loadlib 来加载库，并返回初始化函数。这个初始化函数应将库中提供的函数注册到 Lua 中，就好像一段 Lua 代码定义了其他的函数一样。将在 15.1 节中详细讨论 require，在 26.2 节则会深入 C 程序库。

8.3　错误（error）

“犯错误是人的天性（Errare humanum est）”，因此，必须以最佳的方式来处理错误。由于 Lua 是一种扩展（extension）语言，通常嵌入在应用程序中，因此在发生错误时它不能简单地崩溃或退出。相反，只要发生了一个错误，Lua 就应该结束当前程序块并返回应用程序。

Lua 所遇到的任何未预期条件都会引发一个错误。例如，当试图将两个非数字的值相加、对一个不是函数的值施以调用操作并索引一个不是 table 的值，等。[3]也可以显式地引发一个错误，通过调用 error 函数并传入一个错误消息的参数。通常这个函数就是一种比较恰当的处理错误的方式：

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

由于像“if not <condition> then error end”这样的组合是非常通用的代码，所以 Lua 提供了一个内建（built-in）函数 assert 来完成此类工作：

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

① 若要检测某一平台是否支持动态链接机制，只需在 Lua 提示符中运行 print(package.loadlib("a", "b"))，然后观察执行结果。如果报告不存在指定文件，那么就说明该平台具有动态链接机制。反之会有错误消息提示不支持或未安装该机制。
② 有些编译器会在库函数名前添加一些起始下画线，那么也必须包括这些下画线。
③ 当然，这些操作行为都可以通过修改“元表（metatable）”得以合法化，后文详述。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

assert 函数检查其第一个参数是否为 true。若为 true，则简单地返回该参数；否则（为 **false** 或 **nil**）就引发一个错误。它的第二个参数是一个可选的信息字符串。注意，assert 是一个正规的函数，所以 Lua 同样会在调用该函数前对其参数求值。因此，如果代码类似于：

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

即使 n 是一个数字，Lua 也总是会进行字符串连接。或许在这里更聪明的做法是使用一句显式的测试代码。

当一个函数遭遇了一种未预期的状况（即“异常”），它可以采取两种基本的行为：返回错误代码（通常是 **nil**）或引发一个错误（调用 error）。在这两种选择之间并没有固定的法则，但通常的指导原则是：易于避免的异常应引发一个错误，否则应返回错误代码。

以 sin 函数为例，当调用时传入一个 table，它应如何反应呢？假设，它返回一个错误代码。若要检查错误就不得不这么写：

```
local res = math.sin(x)
if not res then     -- 错误吗?
   <错误处理代码>
```

然而，也可以在调用 sin 前轻易地检测到这种异常情况：

```
if not tonumber(x) then     -- x 不是一个数字吗?
   <错误处理代码>
```

通常既不会检查参数也不会检查 sin 的返回值，如果 sin 的参数不是一个数字，那么或许就表示程序出了问题。此时，处理异常最简单也是最实用的做法就是停止计算，然后给出一条错误消息。

再考虑一下函数 io.open，它用于打开一个文件。如果一个文件不存在，那么它应该具有怎么样的行为呢？在这种情况中，没有什么简单的方法可以在调用函数前检测到这种异常情况。在大多数系统中，要获知一个文件存在与否的唯一方法就是去打开它。因此，如果由于外部原因[1]而使 io.open 无法打开一个文件时，它应返回 **nil**，并附加一条错误消息。通过这种方式，就有机会采取适当的做法来处理异常情况，例如要求用户提供另一个文件名。

```
local file, msg
repeat
  print "enter a file name:"
  local name = io.read()
  if not name then return end   -- 无输入
  file, msg = io.open(name, "r")
  if not file then print(msg) end
```

① 例如，文件不存在或拒绝访问。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
until file
```

如果不想处理这种情况，但仍想安全地运行程序，只需使用 assert 来检测操作即可。

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

这是一种典型的 Lua 技巧，如果 io.open 失败了，assert 就引发一个错误。

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

注意，这里的错误消息是如何成为 assert 的第二个参数的，它是 io.open 的第二个返回值。

8.4　错误处理与异常

对于大多数应用程序而言，无须在 Lua 代码中作任何错误处理，应用程序本身会负责这类问题。所有的 Lua 活动都是由应用程序的一次调用而开始的，这类调用通常是要求 Lua 执行一个程序块。如果执行中发生了错误，此调用会返回一个错误代码，这样应用程序就能采取适当的行动来处理错误。在解释器程序中发生错误时，主循环会打印错误消息，然后继续显示提示符，并等待执行后续命令。

如果需要在 Lua 中处理错误，则必须使用函数 pcall[1]来包装需要执行的代码。

假设在执行一段 Lua 代码时，捕获所有执行中引发的错误，那么第一步就是将这段代码封装到一个函数中，将其称为 foo：

```
function foo ()
  <一些代码>
  if 未预期的条件 then error() end
  <一些代码>
  print(a[i])    -- 潜在的错误：a 可能不是一个 table
  <一些代码>
end
```

然后使用 pcall 来调用 foo：

```
if pcall(foo) then
-- 在执行 foo 时没有发生错误
  <常规代码>
else
-- foo 引发了一个错误，采取适当的行动
  <错误处理代码>
end
```

① protected call，受保护的调用。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

当然，在调用 pcall 时可以传入一个匿名函数：

```
if pcall(function ()
    <受保护的代码>
end) then
    <常规代码>
else
    <错误处理代码>
end
```

pcall 函数会以一种“保护模式（protected mode）”来调用它的第一个参数，因此 pcall 可以捕获函数执行中的任何错误。如果没有发生错误，pcall 会返回 **true** 及函数调用的返回值；否则，返回 **false** 及错误消息。

人们可能会潜意识认为“错误消息”就是一个字符串。但事实上任何类型的 Lua 值都可以作为错误消息传递给 error 函数，并且这些值也会成为 pcall 的返回值。

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

有了这些机制便可以在 Lua 中完成所有的异常处理了。可以使用 error 来抛出一个异常或使用 pcall 来捕获异常，而错误消息则可以标识出错误的类型或内容。

8.5 错误消息与追溯（traceback）

虽然可以使用任何类型的值作为错误消息，但是错误消息通常是一个描述了出错内容的字符串。当遇到一个内部错误时[①]，Lua 就会产生错误消息。而其他时候，错误消息就是传递给 error 函数的值。只要错误消息是一个字符串，Lua 就会附加一些关于错误发生位置的信息。

```
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
```

位置信息中包含了文件名（上例中的 stdin）及行号（上例中的 1）。

error 函数还有第二个附加参数 level（层），用于指出应由调用层级中的哪个（层）函数来报告当前的错误，也就是说明了谁应该为错误负责。举例来说，假设在一个函数中，一开始便

① 例如试图索引一个非 table 的值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

检查传入的参数是否正确：

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected")
  end
    <常规代码>
end
```

然后，其他人在调用函数时传入了错误的参数：

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

由于 foo 是调用了 error，所以 Lua 会认为是函数发生了错误。但实际上却是 foo 的调用者造成的错误。为了纠正这个问题，就要告知 error 函数错误是发生在调用层级的第二层中（第一层是读函数）：

```
function foo (str)
  if type(str) ~= "string" then
    error("string expected", 2)
  end
    <常规代码>
end
```

通常在错误发生时，希望得到更多的调试信息，而不是只有发生错误的位置。至少，能够追溯到发生错误时的函数调用情况，显示一个完整的函数调用栈。当 pcall 返回其错误消息时，它已经销毁了调用栈的部分内容[①]。因此，如果希望得到一个有意义的调用栈，那么就必须在 pcall 返回前获取该信息。为了达成这一要求，Lua 提供了函数 xpcall。该函数除了接受一个需要被调用的函数之外，还接受第二个参数——一个错误处理函数。当发生错误时，Lua 会在调用栈展开（unwind）前调用错误处理函数。于是就可以在这个函数中使用 debug 库来获取关于错误的额外信息了。debug 库提供了两个通用的错误处理函数，一个是 debug.debug，它会提供一个 Lua 提示符，让用户来检查错误的原因；另一个是 debug.traceback，它会根据调用栈来构建一个扩展的错误消息[②]。解释器程序也使用 debug.traceback 来构建其错误消息。另外，也可以在任何时候调用这个函数来获取当前执行的调用栈：

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

① 也就是从 pcall 到错误发生点的这部分调用。
② 当我们以后讨论 debug 库时，你会看到更多关于这些函数的介绍。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 9 章　协同程序（coroutine）

协同程序与线程（thread）[1]差不多，也就是一条执行序列，拥有自己独立的栈、局部变量和指令指针，同时又与其他协同程序共享全局变量和其他大部分东西。从概念上讲线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作地运行。就是说，一个具有多个协同程序的程序在任意时刻只能运行一个协同程序，并且正在运行的协同程序只会在其显式地要求挂起（suspend）时，它的执行才会暂停。

协同程序是一个强大的解决方案，同样地它的几种主要用法也比较复杂。在第一次阅读本章时，不用为无法理解某些示例而感到苦恼。可以先继续阅读后续章节，然后再回过头来阅读本章。但请一定要回来看看，这会是很值得付出的时间。

9.1　协同程序基础

Lua 将所有关于协同程序的函数放置在一个名为“coroutine”的 table 中。函数 create 用于创建新的协同程序，它只有一个参数，就是一个函数。该函数的代码就是协同程序所需执行的内容。create 会返回一个 thread 类型的值，用以表示新的协同程序。通常 create 的参数是一个匿名函数，例如：

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

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

一个协同程序可以处于 4 种不同的状态：挂起（suspended）、运行（runnning）、死亡（dead）和正常（normal）。当创建一个协同程序时，它处于挂起状态。也就是说，协同程序不会在创建它时自动执行其内容。可以通过函数 status 来检查协同程序的状态：

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

函数 coroutine.resume 用于启动或再次启动一个协同程序的执行，并将其状态由挂起改为运行：

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

① “多线程”中所指的线程。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在本例中，协同程序的内容只是简单地打印了“hi”后便终止了，然后它就处于死亡状态，也就再也无法返回了：

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

到目前为止，协同程序看上去还只是像一种复杂的函数调用方法。其实协同程序的真正强大之处在于函数 yield 的使用上，该函数可以让一个运行中的协同程序挂起，而之后可以再恢复它的运行。请看下面这个简单的示例：

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

现在当唤醒这个协同程序时，它就会开始执行，直到第一个 yield：

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

如果此时检查其状态，会发现协同程序处于挂起状态，因此可以再次恢复其运行：

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

从协同程序的角度看，所有在它挂起时发生的活动都发生在 yield 调用中。当恢复协同程序的执行时，对于 yield 的调用才最终返回。然后协同程序继续它的执行，直到下一个 yield 调用或执行结束：

```
coroutine.resume(co)   --> co  2
coroutine.resume(co)   --> co  3
...
coroutine.resume(co)   --> co  10
coroutine.resume(co)   -- 什么都不打印
```

在最后一次调用 resume 时，协同程序的内容已经执行完毕，并已经返回。因此，这时协同程序处于死亡状态。如果试图再次恢复它的执行，resume 将返回 false 及一条错误消息：

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

请注意，resume 是在保护模式中运行的。因此，如果在一个协同程序的执行中发生任何错误，Lua 是不会显示错误消息的，而是将执行权返回给 resume 调用。

当一个协同程序 A 唤醒另一个协同程序 B 时，协同程序 A 就处于一个特殊状态，既不是挂起状态（无法继续 A 的执行），也不是运行状态（是 B 在运行）。所以将这时的状态称为“正

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

常”状态。

Lua 的协同程序还具有一项有用的机制，就是可以通过一对 resume-yield 来交换数据。在第一次调用 resume 时，并没有对应的 yield 在等待它，因此所有传递给 resume 的额外参数都将视为协同程序主函数的参数：

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

在 resume 调用返回的内容中，第一个值为 **true** 则表示没有错误，而后面所有的值都是对应 yield 传入的参数：

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

与此对应的是，yield 返回的额外值就是对应 resume 传入的参数：

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

最后，当一个协同程序结束时，它的主函数所返回的值都将作为对应 resume 的返回值：

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

很少在同一个协同程序中使用所有这些功能，但每种功能各有其用途。

对于那些已经接触过协同程序的读者来说，有必要进一步讨论前先阐述几个概念问题。首先，Lua 提供的是一种 “非对称的协同程序（asymmetric coroutine）”。也就是说，Lua 提供了两个函数来控制协同程序的执行，一个用于挂起执行，另一个用于恢复执行。而一些其他的语言则提供了“对称的协同程序（symmetric coroutine）”，其中只有一个函数用于转让协同程序之间的执行权。

有人将“非对称的协同程序”称为“semi-coroutine[①]”。还有一些人则将相同的术语“semi-coroutine”用于表示某种使用受限制的协同程序实现。在这种实现中，一个协同程序只

① 半支持的、支持有限的协同程序，意指其不是对称的，不是真正的协同程序。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

能在它没有调用其他函数时[1]，才可以挂起执行。换句话说，只有协同程序的主函数才能调用类似于 yield 这样的函数。Python 中的“generator”正是这种 semi-coroutine 所表示的一个实例。

与协同程序之间的对称性区别相比，协同程序与 generator（Python 所提供的）之间的区别则很大。generator 比较简单，但无法实现某些结构，不过完整的协同程序却可以写出这些结构。Lua 则提供了完整的、非对称的协同程序。对于那些更喜欢对称的协同程序的用户而言，则可以基于 Lua 所提供的非对称功能实现出对称的协同程序。这是一个非常简单的任务[2]。

9.2 管道（pipe）与过滤器（filter）

一个关于协同程序的经典示例就是“生产者—消费者”的问题。这其中涉及到两个函数，一个函数不断地产生值（比如从一个文件中读取值），另一个则不断地消费这些值（比如将这些值写到另一个文件）。通常，这两个函数大致是这样的：[3]

```
function producer ()
  while true do
    local x = io.read()          -- 产生新的值
    send(x)                      -- 发送给消费者
  end
end

function consumer ()
  while true do
    local x = receive()          -- 从生产者接收值
    io.write(x, "\n")            -- 消费新的值
  end
end
```

这里有一个问题是如何将 send 与 receive 匹配起来。这是一个典型的“谁具有主循环（who-has-the-main-loop）”的问题。由于生产者和消费者都处于活动状态，它们各自具有一个主循环，并且都将对方视为一个可调用的服务。对于这个特定的示例，可以很容易地修改其中一个函数的结构，展开它的循环，使其成为一个被动调用的函数。不过这样的结构改动可能会使某些真实的应用情况变得复杂。

协同程序被称为是一种匹配生产者和消费者的理想工具，一对 resume-yield 完全一改典型的调用者与被调用者之间的关系。当一个协同程序调用 yield 时，它不是进入了一个新的函数，而是从一个悬而未决的 resume 调用中返回。同样地，对于 resume 的调用也不会启动一个新函

① 也就是说，它的调用栈中没有未完成的调用。
② 基本上只需在每次改变执行权时连续地调用一次 yield 和一次 resume 即可。
③ 在这段代码中，生产者和消费者会一直运行下去。但若要使它们能在没有更多数据处理时停止运行也很容易修改。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数，而是从一次 yield 调用中返回。这项特性正可用于匹配 send 和 receive，这两者都认为自己是主动方，对方是被动方。receive 唤醒生产者的执行，促使其能产出一个新值。而 send 则产出一个新值返还给消费者：

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

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

因此，生产者现在一定是一个协同程序：

```
producer = coroutine.create(
  function ()
    while true do
      local x = io.read()    -- 产生新值
      send(x)
    end
  end)
```

在这种设计中，程序通过调用消费者来启动。当消费者需要一个新值时，它唤醒生产者。生产者返回一个新值后停止运行，并等待消费者的再次唤醒。将这种设计称为“消费者驱动（consumer-driven）”。

还可以扩展上述设计，实现“过滤器（filter）”。过滤器是一种位于生产者和消费者之间的处理功能，可用于对数据的一些变换。过滤器既是一个消费者又是一个生产者，它唤醒一个生产者促使其产生新值，然后又将变换后的值传递给消费者。例如可以在前面代码中添加一个过滤器，在每行起始处插入一个行号。代码如下：

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

function send (x)
  coroutine.yield(x)
end
function producer ()
  return coroutine.create(function ()
    while true do
      local x = io.read()              -- 产生新值
      send(x)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    end
  end)
end

function filter (prod)
  return coroutine.create(function ()
    for line = 1, math.huge do
      local x = receive(prod)          -- 获取新值
      x = string.format("%5d %s", line, x)
      send(x)       -- 将新值发送给消费者
    end
  end)
end

function consumer (prod)
  while true do
    local x = receive(prod)            -- 获取新值
    io.write(x, "\n")                  -- 消费新值
  end
end
```

接下来创建运行代码就非常简单了，只需将这些函数串联起来，然后启动消费者：

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

或者，更简单地写为：

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

如果接触过 UNIX 的 pipe（管道），那么本例的内容就不会很陌生。毕竟，协同程序也是一种（非抢先的）多线程。在 pipe 中每项任务都在各自独立的进程中运行，而在协同程序中每项任务都在各自独立的协同程序中运行。pipe 在 writer（消费者）与 reader（生产者）之间提供一个缓冲器，因此它们的运行速度允许存在一定差异。值得注意的是，在 pipe 中进程间的切换代价很高。而在协同程序中，切换代价则小得多[①]，因此 writer 和 reader 可以彼此协作地运行。

9.3 以协同程序实现迭代器

将循环迭代器视为“生产者—消费者”模式的一种特例，一个迭代器会产出一些内容，而循环体则会消费这些内容。因此，这样看来协同程序似乎也适用于实现迭代器。的确，协同

① 大致相当于一次函数调用的代价。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序为实现这类任务提供了一项有用的工具。那就是先前提到的，协同程序可以一改传统的调用者与被调用者之间关系。有了这种特性，在编写迭代器时，就无须顾及如何在每次成功的迭代调用之间保存状态信息了。

为了说明这类应用，下面来写一个迭代器，使其可以遍历某个数组的所有排列组合形式。若直接编写这种迭代器可能不太容易，但若编写一个递归函数来产生所有的排列组合则不会很困难。想法很简单，只要将每个数组元素都依次放到最后一个位置，然后递归地生成其余元素的排列。代码如下：

```
function permgen (a, n)
  n = n or #a            -- 默认 n 为 a 的大小
  if n <= 1 then         -- 还需改变吗?
    printResult(a)
  else
    for i=1,n do
      -- 将第 i 个元素放到数组末尾
      a[n], a[i] = a[i], a[n]
      -- 生成其余元素的排列
      permgen(a, n - 1)
      -- 恢复第 i 个元素
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

然后，还需要定义其中调用到的打印函数 printResult，并以适当的参数来调用 permgen：

```
function printResult (a)
  for i = 1, #a do
    io.write(a[i], " ")
  end
  io.write("\n")
end

permgen ({1,2,3,4})
  --> 2 3 4 1
  --> 3 2 4 1
  --> 3 4 2 1
    ...
  --> 2 1 3 4
  --> 1 2 3 4
```

当生成函数完成后，将其转换为一个迭代器就非常容易了。首先，将 printResult 改为 yield：

```
function permgen (a, n)
  n = n or #a
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  if n <= 1 then
    coroutine.yield(a)
  else
    <as before>
```

然后，定义一个工厂函数，用于将生成函数放到一个协同程序中运行，并创建迭代器函数。迭代器只是简单地唤醒协同程序，让其产生下一种排列：

```
function permutations (a)
  local co = coroutine.create(function () permgen(a) end)
  return function ()   -- 迭代器
    local code, res = coroutine.resume(co)
    return res
  end
end
```

有了上面的函数，在 for 语句中遍历一个数组的所有排列就非常简单了：

```
for p in permutations{"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
```

permutations 函数使用了一种在 Lua 中比较常见的模式，就是将一条唤醒协同程序的调用包装在一个函数中。由于这种模式比较常见，所以 Lua 专门提供了一个函数 coroutine.wrap 来完成这个功能。类似于 create，wrap 创建了一个新的协同程序。但不同的是，wrap 并不是返回协同程序本身，而是返回一个函数。每当调用这个函数，即可唤醒一次协同程序。但这个函数与 resume 的不同之处在于，它不会返回错误代码。当遇到错误时，它会引发错误。若使用 wrap，可以这么写 permutations：

```
function permutations (a)
  return coroutine.wrap(function () permgen(a) end)
end
```

通常，coroutine.wrap 比 coroutine.create 更易于使用。它提供了一个对于协同程序编程实际所需的功能，即一个可以唤醒协同程序的函数。但也缺乏灵活性。无法检查 wrap 所创建的协同程序的状态，此外，也无法检测出运行时的错误。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

9.4 非抢先式的（non-preemptive）多线程

协同程序提供了一种协作式的多线程。每个协同程序都等于是一个线程。一对 yield-resume 可以将执行权在不同线程之间切换。然而，协同程序与常规的多线程的不同之处在于，协同程序是非抢先式的。就是说，当一个协同程序运行时，是无法从外部停止它的。只有当协同程序显式地要求挂起时（调用 yield），它才会停止。对于有些应用而言，这没有问题，而对于另外一些应用则可能无法接受这种情况。当不存在抢先时，编程会简单许多。无须为同步的 bug 而抓狂，在程序中所有线程间的同步都是显式的，只需确保一个协同程序在它的临界区域之外调用 yield 即可。

对于非抢先式的多线程来说，只要有一个线程调用了一个阻塞的（blocking）操作，整个程序在该操作完成前，都会停止下来。对于大多数应用程序来说，这种行为是无法接受的。这也导致了许多程序员放弃协同程序，转而使用传统的多线程。接下来会用一个有趣的方法来解决这个问题。

先假设一个典型的多线程使用情况：希望通过 HTTP 下载几个远程的文件。当然，若要下载几个远程文件，就必须先知道如何下载一个远程文件。在本例中，将使用 Diego Nehab 开发的 LuaSocket。为了下载一个文件，必须先打开一个到该站点的连接，然后发送下载文件的请求，并接收文件（数据块），最后关闭连接。在 Lua 中可以按以下步骤来完成这项任务。首先，加载 LuaSocket 库。

```
require "socket"
```

然后，定义主机和下载的文件。本例，将从 World Wide Web Consortium（环球网协会）下载《HTML 3.2 参考规范》。

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

接下来，打开一个 TCP 连接，连接到该站点的 80 端口[1]。

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

这步操作将返回一个连接对象，可以用它来发送文件请求。

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

下一步，按 1K 的字节块来接收文件，并将每块写到标准输出：

① HTTP 连接的标准端口。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
while true do
  local s, status, partial = c:receive(2^10)
  io.write(s or partial)
  if status == "closed" then break end
end
```

在正常情况下 receive 函数会返回一个字符串。若发生错误，则会返回 **nil**，并且附加错误代码（status）及出错前读取到的内容（partial）。当主机关闭连接时，就将其余接收到的内容打印出来，然后退出接收循环。

下载完文件后，关闭连接。

```
c:close()
```

现在已经掌握了如何下载一个文件，那么再回到下载几个文件的问题上。最烦琐的做法是逐个地下载文件。因为，这种顺序的做法太慢了，它只能在下载完一个文件后才开始读取该文件。当接收一个远程文件时，程序将大部分的时间花费在等待数据接收上。更明确地说，是将时间用在了对 receive 阻塞调用上。因此，如果一个程序可以同时下载所有文件的话，那么它的运行速度就可以快很多了。当一个连接没有可用数据时，程序便可以从其他连接处读取数据。很明显协同程序提供了一种简便的方式来构建这种并发下载的结构。可以为每个下载任务创建一个新的线程，只要一个线程无可用数据，它就可以将控制权转让给一个简单的调度程序，而这个调度程序则会去调用其他的下载线程。

在以协同程序来重写程序前，先将前面的下载代码重新写为一个函数。代码如下：

```
function download (host, file)
  local c = assert(socket.connect(host, 80))
  local count = 0    -- 记录接收到的字节数
  c:send("GET " .. file .. " HTTP/1.0\r\n\r\n")
  while true do
    local s, status, partial = receive(c)
    count = count + #(s or partial)
    if status == "closed" then break end
  end
  c:close()
  print(file, count)
end
```

由于对远程文件的内容并不感兴趣，所以不需要将文件内容写到标准输出中[1]，只需计算并打印出文件大小即可。在上述代码中，还使用了一个辅助函数 receive 来从连接接收数据。在顺序下载的方法中，receive 的代码可以是这样的：

```
function receive (connection)
```

① 另一方面，几个线程同时接收几个文件并输出的话，输出的内容则会混乱不堪。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

而在并发的实现中，这个函数在接收数据时绝对不能阻塞。因此，它需要在没有足够的可用数据时挂起执行。新代码如下：

```
function receive (connection)
  connection:settimeout(0)      -- 使recevie调用不会阻塞
  local s, status, partial = connection:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connection)
  end
  return s or partial, status
end
```

对 settimeout(0)的调用可使以后所有对此连接进行的操作不会阻塞。若一个操作返回的 status 为“timeout（超时）”，就表示该操作在返回时还未完成。此时，线程就会挂起执行。而以非假的参数来调用 yield，可以告诉调度程序线程仍在执行任务中。注意，即使在超时的情况下，连接也是会返回已经读取到的内容，即记录在 partial 变量中的值。

以下这段代码展示了调度程序及一些辅助代码。table threads 为调度程序保存着所有正在运行中的线程。函数 get 确保每个下载任务都在一个独立的线程中执行。调度程序本身主要就是一个循环，它遍历所有的线程，逐个唤醒它们的执行。并且当线程完成任务时，将该线程从列表中删除。在所有线程都完成运行后，停止循环。

```
threads = {}    -- 用于记录所有正在运行的线程

function get (host, file)
  -- 创建协同程序
  local co = coroutine.create(function ()
    download(host, file)
  end)
  -- 将其插入记录表中
  table.insert(threads, co)
end

function dispatch ()
  local i = 1
  while true do
    if threads[i] == nil then                    -- 还有线程吗?
      if threads[1] == nil then break end        -- 列表是否为空?
      i = 1                                      -- 重新开始循环
    end
    local status, res = coroutine.resume(threads[i])
    if not res then                              -- 线程是否已经完成了任务?
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
      table.remove(threads, i)
    else
      i = i + 1
    end
  end
end
```

最后，主程序需要创建所有的线程，并调用调度程序。例如，若要下载 W3C 站点上的 4 个文件，主程序如下：

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

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

dispatch()   -- 主循环
```

通过协同程序，计算机只需要 6 秒便可下载完成这 4 个文件。但若使用顺序下载的话，则需要多耗两倍的时间（15 秒左右）。

除了速度有所提高外，上面这个实现还不够完美。只要有一个线程在读取数据，就不会有问题。但若所有线程都没有数据可读，调度程序就会执行一个“忙碌等待（Busy Wait）”，不断地从一个线程切换到另一个线程，仅仅是为了检测是否还有数据可读。这样便导致了这个协同程序的实现会比顺序下载多耗费将近 30 倍的 CPU 时间。

为了避免这样的情况，可以使用 LuaSocket 中的 select 函数。这个函数可以用于等待一组 socket 的状态改变，在等待时程序陷入阻塞（block）状态。若要在当前实现中应用这个函数，只需要修改调度程序即可，新版本如下：

```
function dispatch ()
  local i = 1
  local connections = {}
  while true do
    if threads[i] == nil then          -- 还有线程吗?
      if threads[1] == nil then break end
      i = 1                            -- 重新开始循环
      connections = {}
    end
    local status, res = coroutine.resume(threads[i])
    if not res then                    -- 线程是否已经完成了任务?
      table.remove(threads, i)
    else                               -- 超时
      i = i + 1
      connections[#connections + 1] = res
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    if #connections == #threads then          -- 所有线程都阻塞了吗?
      socket.select(connections)
    end
  end
 end
end
```

新的调度程序将所有超时的连接收集到一个名为 connections 的 table 中。记住，receive 会将超时的连接通过 yield 传递，也就是 resume 会返回它们。如果所有的连接都超时了，调度程序就调用 select 来等待这些连接的状态发生变化。这个最终版本的实现与上一个使用协同程序的实现一样快，另外由于它不会有“忙碌等待”，所以只比顺序下载耗费 CPU 资源略多而已。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 10 章　完整的示例

作为对于语言部分介绍的结尾，本章将演示两个完整的程序示例，以此来展示 Lua 不同的应用能力。第一个示例演示了将 Lua 作为一种数据描述语言的用法：第二个示例则实现了“马尔可夫链（Markov chain）”算法，关于这个算法在 Kernighan 与 Pike 所著的《The Practice of Programming（Addison-Wesley，1999）》有所描述。

10.1　数 据 描 述

Lua 的官方网站上有一个数据库，记录了世界范围内一些使用 Lua 的项目。在数据库中，将每个条目表示为一个构造式，这看上去就像是一种具有自然格式的文档，如下所示。更有趣的是，使用这种条目序列的文件居然还是一个有效的 Lua 程序，其中进行了一系列对函数 entry 的调用，而参数正是一个个 table。

```
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.]]
}
```

编写一个能将这些数据以 HTML 的方式展现出来的程序，其结果就是这个网页：http://www.lua.org/uses.html。由于有很多的项目，所以最终页面先使用列表来显示所有项目的标题，然后再显示每个项目的详细信息。以下就是这个程序典型的输出内容：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
<html>
<head><title>Projects using Lua</title></head>
<body bgcolor="#FFFFFF">
Here are brief descriptions of some projects around the
world that use <a href="home.html">Lua</a>.
<br>
<ul>
<li><a href="#1">Tecgraf</a>
<li> <other entries>
</ul>

<h3>
<a name="1" href="http://www.tecgraf.puc-rio.br/">Tecgraf</a>
<br>
<small><em>Computer Graphics Technology Group,
        PUC-Rio</em></small>
</h3>

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

<a name="2"></a><hr>

<other entries>

</body></html>
```

要读取这些数据，程序只需简单地给出一个关于 entry 的合适定义，然后将数据文件作为一个程序来运行（通过 dofile）。注意，必须对所有的条目遍历两次，第一次为获得标题列表，第二次为获得项目描述。这种方式通常是将所有的条目收集到一个数组中。另外，还有一种方法，即将数据文件运行两次，每次使用不同的 entry 定义。那么在下面的程序中将采用这种方式。

首先，定义一个辅助函数用于输出格式化后的文本（这个函数在 5.2 节中已出现过）：

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

函数 writeheader 只是简单地写出一个始终不变的页面首部：

```
function writeheader()
  io.write([[
    <html>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    <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
```

entry 的第一个定义将每个项目的标题以一个列表项目输出。参数 o 就将是每个描述项目的 table：

```
function entry1 (o)
  count = count + 1
  local title = o.title or '(no title)'
  fwrite('<li><a href="#%d">%s</a>\n', count, title)
end
```

如果 o.title 为 nil（table 中没有这个字段），函数就使用一个固定的字符串“(no title)”。第二个定义写出所有关于项目的详细数据。有些复杂，因为所有字段都是可选的。[①]

```
function entry2 (o)
  count = count + 1
  fwrite('<hr>\n<h3>\n')

  local href = o.url and string.format(' href="%s"', o.url) or ''
  local title = o.title or o.org or 'org'
  fwrite('<a name="%d"%s>%s</a>\n', count, href, title)

if o.title and o.org then
   fwrite('<br>\n<small><em>%s</em></small>', o.org)
  end
  fwrite('\n</h3>\n')

if o.description then
   fwrite('%s<p>\n',
         string.gsub(o.description, '\n\n+', '<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
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

最后一个函数用来结束一个页面：

```
function writetail ()
  fwrite('</body></html>\n')
end
```

主程序如下所示。先开始一个页面，并加载数据文件。然后使用 entry 的第一个定义（entry1）来运行数据文件，创建了标题列表；再以 entry 的第二个定义来运行数据文件。最后关闭这个页面。

```
local inputfile = 'db.lua'
writeheader()

count = 0
f = loadfile(inputfile)              -- 加载数据文件

entry = entry1                       -- 定义'entry'
fwrite('<ul>\n')
f()                                  -- 运行数据文件
fwrite('</ul>\n')

count = 0
entry = entry2                       -- 重定义'entry'
f()                                  -- 再次运行数据文件

writetail()
```

10.2 马尔可夫链（markov chain）算法

第二个示例是一个马尔可夫链算法的实现。该算法根据原始文本中 n 个单词的序列来确定后面的单词，从而生成随机的文本。在本例中，将 n 定为 2。

程序先读取原始文本，并创建一个 table。table 的创建过程为：以每两个单词为一个前缀，在 table 中创建一个列表，该列表记录了原始文本中所有位于该前缀之后的单词。当构建好这个 table 后，程序就利用它来生成随机文本。结果中的每个单词都来自于它在原始文本中的前缀，而具有相同前缀的单词出现在结果中的概率也与原始文本一样。最终会得到一串比较随机的文本。例如，以本书原版作为原始文本输入程序，那么输出的结果中就会有这么一段："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."

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

将两个单词以空格相连，编码成一个前缀：

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

使用字符串 NOWORD（"\n"）来初始化一个前缀单词，并且标记文本的结尾。例如，对于原始文本：

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

由此构造出的 table 的内容如下：

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

程序将这个 table 保存在变量 statetab 中。若要向此 table 中的某个前缀列表插入一个新单词，可以使用以下函数：

```
function insert (index, value)
  local list = statetab[index]
  if list == nil then
    statetab[index] = {value}
else
    list[#list + 1] = value
  end
end
```

先检查某前缀是否已拥有了一个列表。如果没有，便以新值（参数 value）来创建一个新列表。否则，就将新值添加到现有列表的末尾。

为了创建 table statetab，用两个变量 w1 和 w2 来记录最后读取的两个单词。每读取一个新单词，就将它添加到与 w1-w2 相关联的列表中，然后更新 w1 和 w2。

当创建完 table 后，程序便开始生成具有 MAXGEN 单词的文本。首先，重新初始化变量 w1 和 w2。然后，对于每个前缀，程序从其对应的单词列表中随机地选出下一个单词，并打印。最后更新 w1 和 w2。

下面是该程序的辅助函数定义：

```
function allwords ()
  local line = io.read()                    -- 当前行
  local pos = 1                             -- 行中的当前位置
  return function ()                        -- 迭代器函数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    while line do                                   -- 只要还有行就一直循环
      local s, e = string.find(line, "%w+", pos)
      if s then                                     -- 找到了一个单词吗?
        pos = e + 1                                 -- 更新下一个位置
        return string.sub(line, s, e)               -- 返回该单词
      else
        line = io.read()                            -- 没有找到单词，尝试下一行
        pos = 1                                     -- 从行首位置重新开始
      end
    end
    return nil                                      -- 所有行都遍历完毕
end
end

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

local statetab = {}

function insert (index, value)
  local list = statetab[index]
  if list == nil then
   statetab[index] = {value}
  else
    list[#list + 1] = value
  end
end
```

下面是主程序：

```
local N  = 2
local MAXGEN = 10000
local NOWORD = "\n"

-- 构建table
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)

-- 生成文本
w1 = NOWORD; w2 = NOWORD      -- 重新初始化
for i=1, MAXGEN do
   local list = statetab[prefix(w1, w2)]
   -- 从列表中选择一个随机项
   local r = math.random(#list)
   local nextword = list[r]
   if nextword == NOWORD then return end
   io.write(nextword, " ")
   w1 = w2; w2 = nextword
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第2部分

Second Edition

Programming in Lua

Lua

- 第 11 章　数据结构
- 第 12 章　数据文件与持久性
- 第 13 章　元表（metatable）与元方法（meatmethod）
- 第 14 章　环境
- 第 15 章　模块与包
- 第 16 章　面向对象编程
- 第 17 章　弱引用 table

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 11 章　数 据 结 构

Lua 中的 table 不是一种简单的数据结构，它可以作为其他数据结构的基础。其他语言提供的数据结构，如数组、记录、线性表、队列、集合等，在 Lua 中都可以通过 table 来表示。此外，用 Lua 的 table 来实现这些结构的效率高。

在 C 和 Pascal 这样的传统语言中，尽管可以使用 Lua 的 table 来实现数组和列表，但通常以数组和列表（记录+指针）来实现大多数的数据结构。因为 table 本身就比数组和列表的功能强大得多。由此许多算法都可以忽略一些细节问题，从而简化它们的实现。例如，在 Lua 中很少编写搜索算法，这是因为 table 本身就提供了直接访问任意类型的功能。

高效地使用 table 是问题的关键。接下来将演示如何通过 table 来实现一些传统的数据结构，并且还将给出一些使用这些结构的例子。首先从数组和列表开始，不是因为需要它们来作为其他结构的基础，而是因为大多数程序员都比较熟悉它们了。在前面关于语言的章节中也曾提到过这方面的内容，不过为了完整性起见，本章将更详细地进行讨论。

11.1　数　　组

使用整数来索引 table 即可在 Lua 中实现数组。因此，数组没有一个固定的大小，可以根据需要增长。通常，当初始化一个数组时，也就间接地定义了它的大小。例如，在执行了以下代码后，任何对字段范围 1~1000 之外的访问都会返回一个 **nil**，而不是 0：

```
a = {}    -- 新建一个数组
for i=1, 1000 do
  a[i] = 0
end
```

长度操作符（#）依赖于这个事实来计算数组的大小：

```
print(#a)       --> 1000
```

可以使用 0、1 或其他任意值来作为数组的起始索引：

```
-- 使用索引值-5～5 来创建一个数组
a = {}
for i=-5, 5 do
  a[i] = 0
end
```

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

然而，在 Lua 中的习惯一般是以 1 作为数组的起始索引。Lua 库和长度操作符都遵循这个约定。如果你的数组不是从 1 开始的，那就无法使用这些功能了。

通过 table 的构造式，可以在一句表达式中创建并初始化数组：

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

这种构造式可以根据要求变得更长[1]。

11.2 矩阵与多维数组

在 Lua 中，有两种方式来表示矩阵。第一种是使用一个“数组的数组”，也就是说，一个 table 中的每个元素是另一个 table。例如，使用以下代码来创建 N×M 的零矩阵：

```
mt = {}              -- 创建矩阵
for i=1,N do
mt[i] = {}           --  创建一个新行
  for j=1,M do
    mt[i][j] = 0
  end
end
```

由于在 Lua 中 table 是一种对象，因此在创建矩阵时，必须显式地创建每一行。从一方面看，这的确比在 C 和 Pascal 中直接声明一个多维数组烦琐；但从另一方面看，它也给予了更多的灵活性。例如，创建一个三角形矩阵，只需将前例中的循环 for j=1,M do … end 改为 for j=1,i do … end。这种修改同时可以使三角形矩阵只使用原先一半的内存。

在 Lua 中表示矩阵的第二种方式是将两个索引合并为一个索引。如果两个索引是整数，可以将第一个索引乘以一个适当的常量，并加上第二个索引。以下代码就使用这种方法来创建 N×M 的零矩阵：

```
mt = {}          -- 创建矩阵
for i=1,N do
   for j=1,M do
    mt[(i-1)*M + j] = 0
  end
end
```

如果索引是字符串，那么可以把索引拼接起来，中间使用一个字符来分隔。例如，使用字符串 s 和 t 来索引一个矩阵，可以通过代码 m[s..":"..t]。其中，s 和 t 都不能包含冒号，否则像("a:","b") 或("a",":b")这样的索引会使最终索引变成”a::b”。如果无法保证这点的话，可以使

① 当然，最好不要超过几百万个元素。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

用例如'\0'这样的控制字符来分隔两个索引。

通常应用程序会用到一种特殊的矩阵，称为“稀疏矩阵”，这种矩阵中的大多数元素为 0 或 nil。例如，可以通过稀疏矩阵来表示一个图（graph）。当矩阵的 m,n 位置上有一个值 x，即表示图中的结点 m 和 n 是相连的，其权重（cost）为 x；若图中这些结点不相连的话，则矩阵 m,n 位置上的值为 nil。若要表示一个具有 1 万个结点的图，其中每个结点大约会与其他 5 个结点相连，那么就需要一个能包含 1 亿个元素的矩阵[①]，但是其中大约只有 5 万个元素不为 nil[②]。许多数据结构的书籍都会讨论到这个大小问题，如何才能实现这种稀疏矩阵而不浪费 400MB 内存。当在 Lua 中编程时，则无须用到这些技术。因为，数组是以 table 来表示的，它们本身就是稀疏的。在第一种表示（table 的 table）中，需要 1 万个 table，每个 table 包含 5 个元素，总共 5 万个条目。在第二种表示中，需要一个 table，其中包含 5 万个条目。无论哪种表示方式，都只需要为非 nil 的元素付出空间。

虽然对稀疏矩阵使用长度操作符不是一种语法错误。但也不能进行该操作，因为在有效条目之间存在“空洞（nil 值）”。在大多数对稀疏矩阵的操作中，由于存在许多空条目，遍历矩阵是非常低效的。所以，一般使用 pairs 且只遍历那些非 nil 的元素。例如，要将矩阵的一行与一个常量相乘，可以使用以下代码：

```
function mult (a, rowindex, k)
  local row = a[rowindex]
  for i, v in pairs(row) do
    row[i] = v * k
  end
end
```

注意，table 中的 key 是无序的，所以使用 pairs 的迭代并不保证会按递增次序来访问元素。对于一些任务而言（像上面这个例子），这没有问题；但对于另一些任务而言，或许就需要采用另外的方法了，比如链表。

11.3 链　　表

由于 table 是动态的实体，所以在 Lua 中实现链表是很方便的。每个结点以一个 table 来表示，一个“链接”只是结点 table 中的一个字段，该字段包含了对其他 table 的引用。例如，要实现一个基础的列表，其中每个结点具有两个字段：next 和 value，先创建一个用作列表头结点的变量：

```
list = nil
```

① 一个“1 万×1 万”的正方形矩阵。
② 每行（表示一个结点）具有 5 个非 nil 的列，对应于 5 个邻接结点。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在表头插入一个元素，元素值为 v：

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

遍历此列表：

```
local l = list
while l do
  <访问 l.value>
  l = l.next
end
```

至于其他类型的列表，例如双向链表或环形表，都可以使用相同的方法实现。然而，在 Lua 中很少需要这类结构，因为通常存在着一些更简单的方式来表示数据。例如，可以通过一个（几乎无限大的）数组来表示一个栈。

11.4 队列与双向队列

在 Lua 中实现队列的一种简单方法是使用 table 库的函数 insert 和 remove。这两个函数可以在一个数组的任意位置插入或删除元素，并且根据操作要求移动后续元素。不过对于较大的结构，移动的开销是很大的。一种更高效的实现是使用两个索引，分别用于首尾的两个元素：

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

为了避免污染全局名称空间，将在一个 table 内部定义所有的队列操作，这个 table 且称为 List[1]。这样，将上例重新写为：

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

现在就可以在常量时间内完成在两端插入或删除元素了。

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

① 在后面还会为此创建一个模块（Module）。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

function List.popfirst (list)
  local first = list.first
  if first > list.last then error("list is empty") end
  local value = list[first]
  list[first] = nil        -- 为了允许垃圾收集
  list.first = first + 1
  return value
end

function List.poplast (list)
  local last = list.last
  if list.first > last then error("list is empty") end
  local value = list[last]
  list[last] = nil        -- 为了允许垃圾收集
  list.last = last - 1
  return value
end
```

如果希望该结构能严格地遵循队列的操作规范，那么只调用 pushlast 和 popfirst 就可以了，这样 first 和 last 都会不断地增长。然而，在 Lua 中使用 table 来表示数组，既可以在 1~20 的范围内索引，也可以在 16777216~16777236 的范围内索引。因为 Lua 使用双精度来表示数字，程序可以以每秒 1 百万次的速度进行插入操作，如此运行 200 年都不会发生溢出问题。

11.5　集合与无序组（bag）

假设列一份程序代码中的所有标识符，并且过滤掉其中所有的保留字。一些 C 程序员会倾向于使用字符串的数组来表示保留字集合，然后搜索这个数组来查看一个单词是否属于该集合。为了提高搜索的速度，他们可能还会使用二叉树来表示该集合。

在 Lua 中有一种高效且简单的方式来表示这类集合，就是将集合元素作为索引放入一个 table 中。那么对于任意值都无须搜索 table，只需用该值来索引 table，并查看结果是否为 **nil**。在当前假设的示例中，可如下：[①]

```
reserved = {
  ["while"] = true,     ["end"] = true,
```

① 由于 reserved 中的某些单词是 Lua 的保留字，所以不能直接将其用作标识符。例如，不能写为 while = true。但可以写为["while"] = true。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    ["function"] = true,  ["local"] = true,
    }

for w in allwords() do
if not reserved[w] then
    <对'w'作任意处理>   -- 'w'不是保留字
  end
end
```

若要使初始化过程变得更清晰，可以借助一个辅助函数来创建集合：

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

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

包，有时也称为“多重集合（Multiset）”，与普通的集合的不同之处在于其每个元素可以出现多次。在 Lua 中包的表示类似于上面的集合表示，只不过包需要将一个计数器与 table 的 key 关联。若要插入一个元素，则需要递增其计数器：

```
function insert (bag, element)
  bag[element] = (bag[element] or 0) + 1
end
```

若要删除一个元素，则需要递减其计数器：

```
function remove (bag, element)
  local count = bag[element]
  bag[element] = (count and count > 1) and count - 1 or nil
end
```

只有当计数器已存在或大于 0 时，才保留它。

11.6 字符串缓冲

假设正在编写一段关于字符串的代码，例如正在逐行地读取一个文件。典型的读取代码是这样的：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这段代码看似可以正常工作，但如果面对较大的文件时，它却会导致极大的性能开销。例如，用这段代码来读取一个 350KB 大小的文件就需要将近 1 分钟的时间。

为什么会这样呢？为了搞清楚执行期间到底发生了什么，来设想一下当前正处于读取循环的中间。假设每行有 20 个字节，已读了 2500 行，那么 buff 现在就是一个 50KB 大小的字符串。当 Lua 作字符串连接 buff .. line .. "\n"时，就创建了一个长 50020 字节的新字符串，并从 buff 中复制了 50000 字节到这个新字符串。这样，对于后面的每一行，Lua 都需要移动 50KB 甚至更多的内存。在读取了 100 行（仅 2KB）以后，Lua 就已经移动了至少 5MB 的内存。此外，这个算法还具有二次复杂度。最后，当 Lua 完成了 350KB 的读取后，它已至少移动了 50GB 的数据。

这个问题不是 Lua 所特有的，在其他语言中，只要字符串是不可变的（immutable）值，也会有类似的问题。Java 就是最有名的例证。

在继续讲解前，需要注明一下这种情况并不是一种常见的问题。对于较小的字符串，上述循环可以很好地工作。当需要读取整个文件时，Lua 提供了 io.read("*all")选项，这样便可以一次性读取整个文件。不过，Java 也提供了 StringBuffer 来解决这个问题。在 Lua 中，我们可以将一个 table 作为字符串缓冲。其关键是使用函数 table.concat，它会将给定列表中的所有字符串连接起来，并返回连接的结果。使用 concat 来重写上述循环：

```
local t = {}
for line in io.lines() do
  t[#t + 1] = line .. "\n"
end
local s = table.concat(t)
```

先前代码读取同样的文件需要 1 分钟，而这个实现只需花小于 0.5 秒的时间。[①]

concast 函数还有第二个可选的参数，可以指定一个插在字符串间的分隔符。有了这个分隔符参数，就不必在每行后插入一个"\n"了。

```
local t = {}
for line in io.lines() do
   t[#t + 1] = line
end
s = table.concat(t, "\n") .. "\n"
```

虽然 concat 函数会在字符串之间插入分隔符，但还需要在结尾处添加一个换行。为此上例在最后一次连接时复制了整个结果字符串，而这时的字符串也已经相当长了。没有直接的选项让 concat 插入这个额外的分隔符，不过可以“欺骗”concat，只需在 t 后面添加一个空字符串：

```
t[#t + 1] = ""
s = table.concat(t, "\n")
```

① 若要读取整个文件，最好还是使用 io.read 和"*all"选项。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

concat 在空字符串前插入了这个额外的换行符，位于结果字符串的末尾。

从内部来看，concast 和 io.read("*all")都使用了一个相同的算法来连接许多小的字符串。标准库中的其他几个函数也使用这个算法来创建较大的字符串。下面来看一下它是如何工作的。

一开始循环采用了一种线性的方法来连接字符串，把较小的字符串逐个地连接起来，然后每次都将连接结果存入一个累加器。而新的算法可以避免这么做，它采用了一种二分的方法（binary approach）。它只在某些情况下将几个较小的字符串连接起来，然后再将结果字符串与更大的字符串进行连接。其算法的核心是一个栈，已创建的大字符串位于栈的底部，而较小的字符串则通过栈顶进入。对栈中元素的处理类似于著名的“汉诺塔（Tower of Hanoi）”。栈中的任意字符串都比下面的字符串短。如果压入的新字符串比下面已存在的字符串长，就将两者连接。然后，再将连接后的新字符串与更下面的字符串比较，如果是新建字符串更长的话，则再次连接它们。这样的连接一直向下延续应用，直到遇到一个更大的字符串或者到达了栈底为止。

```
function addString (stack, s)
stack[#stack + 1] = s      -- 将's'压入栈
  for i = #stack-1, 1, -1 do
    if #stack[i] > #stack[i+1] then
      break
    end
    stack[i] = stack[i] .. stack[i + 1]
    stack[i + 1] = nil
  end
end
```

为了获取栈缓冲中的最终内容，只需连接其中所有的字符串就可以了。

11.7 图

就像其他编程语言一样，Lua 允许程序员写出多种图的实现，每种实现都有其所适用的算法。接下来将介绍一种简单的面向对象的实现，其中结点表示为对象[1]及边（arc）表示为结点间的引用。

每个结点表示为一个 table，这个 table 有两个字段：name（结点的名称）和 adj（与此结点邻接的结点集合）。由于从一个文本文件中读取图数据，所以需要一种通过一个结点名来找到该结点的方法。因此，使用了一个额外的 table 来将名称对应到结点。函数 name2node 可以根据给定的名称返回对应的结点：

```
local function name2node (graph, name)
if not graph[name] then
```

① 当然，实际上就是 table。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  -- 结点不存在，创建一个新的
  graph[name] = {name = name, adj = {}}
  end
  return graph[name]
end
```

以下这个函数用于构造一个图。它逐行地读取一个文件，文件中的每行都有两个结点名称，表示了在两个结点之间有一条边，边的方向从第一个结点到第二个结点。函数对于每行都使用 string.match 来切分一行中的两个名称，然后根据名称查找结点（如果需要还会创建结点），最后连接结点。

```
function readgraph ()
  local graph = {}
  for line in io.lines() do
    -- 切分行中的两个名称
    local namefrom, nameto = string.match(line, "(%S+)%s+(%S+)")
    -- 查找相应的结点
    local from = name2node(graph, namefrom)
    local to = name2node(graph, nameto)
    -- 将'to'添加到'from'的邻接集合
    from.adj[to] = true
  end
  return graph
end
```

以下这个函数展示了一个使用图的算法。函数 findpath 采用深度优先的遍历，在两个结点间搜索一条路径。它的第一个参数是当前结点，第二个参数是其目标结点，第三个参数用于保存从起点到当前结点的路径，最后一个参数是所有已访问过结点的集合（用于避免回路）。注意，该算法直接对结点进行操作，而不是它们的名称。例如，visited 是一个结点集合，而不是结点名称的集合。path 也一样是一个结点的列表。

```
function findpath (curr, to, path, visited)
  path = path or {}
  visited = visited or {}
  if visited[curr] then          -- 结点是否已访问过?
    return nil                   -- 这里没有路径
  end
  visited[curr] = true           -- 将结点标记为已访问过
  path[#path + 1] = curr         -- 将其加到路径中
  if curr == to then             -- 最后的结点吗?
    return path
  end
  -- 尝试所有的邻接节点
  for node in pairs(curr.adj) do
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
      local p = findpath(node, to, path, visited)
      if p then return p end
    end
    path[#path] = nil          -- 从路径中删除结点
  end
```

为了测试上述代码，首先编写一个函数来打印一条路径，然后再编写一些代码来使其运行。

```
function printpath (path)
  for i=1, #path do
    print(path[i].name)
  end
end

g = readgraph()
a = name2node(g, "a")
b = name2node(g, "b")
p = findpath(a, b)
if p then printpath(p) end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 12 章　数据文件与持久性

当涉及到数据文件的处理时，人们往往会认为写数据比读数据简单得多。当写一个文件时，对写的内容拥有完全的控制权。但是当读一个文件时，却无从得知会读到什么内容。一个强健的程序除了需要处理一个合法文件中所包含的所有类型的数据，还应能很好地处理坏损的文件。因此，编写一个强健的输入程序总是比较困难的。

在本章中，我们将看到如何使用 Lua 来避免程序中所有有关数据读取的代码，只需将数据按一种适当的格式书写就可以了。

12.1　数 据 文 件

就像 10.1 节介绍的，可以借由 table 构造式来定义一种文件格式。只需在写数据时做一点额外的工作，读取数据就会变得相当容易。这项技术也就是将数据作为 Lua 代码来输出，当运行这些代码时，程序也就读取了数据。而 table 的构造式可以使这些输出代码看上去更像是一个普通的数据文件。

下面通过一个示例来更清楚地理解这种做法。如果数据文件是一种预定义的格式，例如 CSV（Comma-Separated Values，逗号分隔值）或 XML，那么可以选择的做法很少。不过，如果是为了应用而创建数据文件的话，那么就可以使用 Lua 的构造式作为格式。在这种格式中，每条数据记录表示为一个 Lua 构造式。这样，原来以这种形式书写的数据文件：

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

现在可以改为：

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

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

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

记住，Entry{<code>}与 Entry({<code>})是完全等价的，都是以一个 table 作为参数来调用函数 Entry。因此，上面这段数据也是一个 Lua 程序。为了读取该文件，我们只需定义一个合适的 Entry，然后运行此程序就可以了。例如，以下程序计算了数据文件中条目的数量：

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

下一个程序则可用于收集数据文件中所有作者的姓名，然后打印出这些姓名（不需要与文件中的次序相同）：

```
local authors = {}      -- 作者姓名的集合
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

可以看到这些代码片段都采用了事件驱动的做法。Entry 函数作为一个回调函数，在 dofile 时为数据文件中的每个条目所调用。

若文件不是非常大，可以使用名值对来表示每个字段：[①]

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

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

这种格式就是“自描述的数据（self-describing data）”格式，其中每项数据都伴随一个表示其含义的简短描述。自描述的数据比 CSV 或其他紧缩格式更具可读性。当需要修改时，也易于手工编辑，可以在基本格式中作出一个细小的改动，而不需要同时改变数据文件。例如，如果要新增一个字段，只需修改读取程序中的一小块就可以了，内容就是当该字段不存在时提供一个默认值。

使用名值对格式后，那个收集作者姓名的程序改为：

① 如果这种格式让你想到了 BibTeX，但其实是不太一样的。不过 BibTeX 却是 Lua 中构造式语法的灵感来源之一。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local authors = {}      -- 作者姓名的集合
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

现在字段的次序就不重要了，即使有些条目没有作者字段，也只需要修改 Entry 函数：

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

Lua 不仅运行速度快，而且编译速度也快。例如，上面这个用于列出作者的程序在处理 2MB 数据时，只需不到 1 秒钟的时间。这不是偶然的结果，自从 Lua 创建之初就把数据描述作为 Lua 的主要应用之一来考虑的，开发人员为能较快地编译大型程序投入了很多的努力。

12.2　串行化（Serialization）

通常需要串行化一些数据，也就是将数据转换为一个字节流或字符流。然后就可以将其存储到一个文件中，或者通过网络连接发送出去了。串行化后的数据可以用 Lua 代码来表示，这样当运行这些代码时，存储的数据就可以在读取程序中得到重构了。

如果想要恢复一个全局变量的值，那么串行化的结果或许可以是“varname = <exp>”，其中<exp>是一段用于创建该值的 Lua 代码，而 varname 只是一个简单的标识符。接下来，看一下如何编写创建一个值的代码。例如，对于一个数字值，方法如下：

```
function serialize (o)
  if type(o) == "number" then
    io.write(o)
  else <其他情况>
    end
end
```

对于一个字符串值，方法如下：

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

然而，如果字符串中包含特殊字符（例如引号、换行），那么最终代码就不是一段有效的 Lua 程序了。

也可以使用另一种字符串字面表示方法，如下所示：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

注意，如果有用户故意使其字符串为"]]..os.execute('rm *')..[["[①]，那么最终保存下来的结果将变成：

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

若加载这个“数据”将会出现不可做量的后果。

可以使用一种简单且安全的方法来括住一个字符串，那就是以“%q”来使用 string.format 函数。这样它就会用双引号来括住字符串，并且正确地转移其中的双引号和换行符等其他特殊字符。

```
a = 'a "problematic" \\string'
print(string.format("%q", a))    --> "a \"problematic\" \\string"
```

通过使用这个特性，serialize 函数可以改为：

```
function serialize (o)
  if type(o) == "number" then
   io.write(o)
 elseif type(o) == "string" then
   io.write(string.format("%q", o))
 else <其他情况>
 end
end
```

Lua 5.1 还提供了另一种可以以一种安全的方法来括住任意字符串的方法。这是一种新的标记方式[=[...]=]，用于长字符串。然而，这种新方式主要是为手写的代码提供方便的，通过它就不需要改变任何字符串的内容了。在自动生成的代码中，要转移那些问题字符，还是使用 string.format 与“%q”选项更为方便。

如果仍然在自动生成的代码中使用长字符串标记的话，那么就需要注意两个细节问题。首先，必须使用正确数量的等号。这个正确的数量应比字符串中出现的最长的等号序列还大 1。由于，在字符串中出现长序列的等号是很有可能的，并且其他序列也不会产生一个错误的字符串结尾的标记，所以要注意等号序列。第二个细节是，Lua 总是会忽略所有长字符串开头的换行符。一种避免这个问题的简单方法就是，在字符串起始处添加一个换行符。

以下这个 quote 函数就是根据上面提到的两个注意点编写的处理函数。

```
function quote (s)
  -- 查找最长的等号序列
  local n = -1
  for w in string.gmatch(s, "]=*") do
    n = math.max(n, #w - 1)
  end
```

① 例如，将这个字符串用在了用户自己的地址上。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
-- 产生'n'+1 个等号
local eq = string.rep("=", n + 1)

-- 生成长字符串的字面表示
  return string.format(" [%s[\n%s]%s] ", eq, s, eq)
end
```

它可以接收任意字符串，并返回其格式化为长字符串的结果。对 string.gmatch 的调用会创建一个迭代器，通过该迭代器就可以遍历字符串 s 中所有出现模式']=*'[①]的地方。在每处出现等号的地方，循环就会更新 n，使其保持为当前所遇到的最大等号数量。在循环结束后使用 string.rep 将等号重复 n+1 遍，也就是生成一个等号序列字符串，其长度比现有字符串中的最长等号序列还多 1。最后，string.format 将 s 嵌入一对具有正确数量等号的方括号对中，并在方括号外添加一些额外的空格，以及在 s 开头插入一个换行符。

12.2.1　保存无环的 table

下一个任务是保存 table。保存 table 有几种方法，选用哪种方法取决于对 table 的结构作出了哪些限制性的假设。没有一种算法适用于所有的情况。简单的 table 不仅需要更简单的算法，而且需要更完美地输出结果。

第一个算法如下：

```
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
```

① 这个模式表示“一个右方括号后面跟着零个或多个等号”。将在第 20 章中讨论模式匹配的详细内容。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

尽管这个函数很简单，但却可以完成基本的保存工作。只要 table 的结构是一个树结构[①]，它甚至还能处理嵌套的 table（table 中的 table）。可以作为一个练习，尝试在输出格式中缩进那些嵌套的 table[②]。

上例函数假设了一个 table 中的所有 key 都是合法的标识符。但如果一个 table 的 key 为数字或者为非法的 Lua 标识符[③]，那么就会出现问题。一个简单的解决方法是将这行：

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

改为：

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

这样，便增强了这个函数的强健性，但却损失了结果文件的美观性。对于调用：

```
serialize{a=12, b='Lua', key='another "one"'}
```

第一个版本的 serialize 会输出：

```
{
  a = 12,
  b = "Lua",
  key = "another \"one\"",
}
```

而第二个版本则输出：

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

可以测试每种需要方括号的情况，从而改善结果的美观性。

12.2.2 保存有环的 table

若要处理具有任意拓扑结构（带环的 table 或共享子 table）的 table，就需要采用另外一种方法了，table 构造式是无法表示这类 table 的。所以为了表示“环”，则需要引入名称，接下来这个保存函数要求将待保存的值及其名称一起作为参数传入。此外，还必须持有一份所有已保持过的 table 的名称记录，以此来检测环并复用其中的 table。使用一个额外的 table 用作此

① 也就是，不共享“子 table”，并且没有“环”。

② 提示：为 serialize 添加一个额外的参数，用于表示缩进字符串。

③ 译注：指一个字符串类型的 key。回忆一下，在 Lua 中 table 的构造式{ xxx = ... }等价于{ ["xxx"] = ... }，在第二种形式中 xxx 可以是任意内容的字符串，但在第一种形式中，xxx 就必须是一个合法的 Lua 标识符。例如，你可以写出{ ["if"] = ... }，但{ if = ... }就非法了。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

项记录，这个 table 以其他 table 作为 key，并以其他 table 的名称作为 value。代码如下：

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

function save (name, value, saved)
  saved = saved or {}                    -- 初始值
  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                 -- 该value是否已保存过?
      io.write(saved[value], "\n")       -- 使用先前的名字
    else
      saved[value] = name                -- 为下次使用保持名字
      io.write("{}\n")                   -- 创建一个新的table
      for k,v in pairs(value) do         -- 保存其字段
        k = basicSerialize(k)
        local fname = string.format("%s[%s]", name, k)
        save(fname, v, saved)
      end
    end
  else
    error("cannot save a " .. type(value))
  end
end
```

假设准备保存的 table 的 key 只为字符串或数字。函数 basicSerialize 用于串行化这些基本类型，返回串行化的结果。而另一个函数 save 则完成真正的工作。saved 参数是一个 table，用于记录已保存过的 table。假设有一个 table 如下所示：

```
a = {x=1, y=2; {3,4,5}}
a[2] = a    --     环
a.z = a[1]  --     共享子table
```

然后，调用 save("a", a)将它保存为：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

这些赋值语句的实际顺序可能会有所不同，这取决于一个 table 的遍历顺序。不过，该算法可以保证在一句新的定义中所用到的变量都已经定义过了。

如果想以共享的方式来保存几个 table 中的共同部分，只需在调用 save 时使用相同的 saved 参数。例如，假设有两个 table：

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

如果以独立的方式保存它们，那么结果中不会有共同部分：

```
save("a", a)
save("b", b)

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

然而，当使用同一个 saved table 来调用 save 时，串行化结果就会共享共同部分：

```
local t = {}
save("a", a, t)
save("b", b, t)

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

在 Lua 中，还有一些其他比较常见的方法。有的在保存一个值时无须给出一个全局名称（而是通过一段代码来构造一个局部值，并返回这个值），有的则可以处理函数（通过构造一个辅助 table，来将函数与它的名称关联起来）等。Lua 赋予了构建这些机制的能力。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 13 章　元表（metatable）与元方法（meatmethod）

通常，Lua 中的每个值都有一套预定义的操作集合。例如，可以将数字相加，可以连接字符串，还可以在 table 中插入一对 key-value 等。但是我们无法将两个 table 相加，无法对函数作比较，也无法调用一个字符串。

可以通过元表来修改一个值的行为，使其在面对一个非预定义的操作时执行一个指定的操作。例如，假设 a 和 b 都是 table，通过元表可以定义如何计算表达式 a+b。当 Lua 试图将两个 table 相加时，它会先检查两者之一是否有元表，然后检查该元表中是否有一个叫__add 的字段。如果 Lua 找到了该字段，就调用该字段对应的值。这个值也就是所谓的“元方法”，它应该是一个函数，在本例中，这个函数用于计算 table 的和。

Lua 中的每个值都有一个元表。table 和 userdata 可以有各自独立的元表，而其他类型的值则共享其类型所属的单一元表。Lua 在创建新的 table 时不会创建元表：

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

可以使用 setmetatable 来设置或修改任何 table 的元表：

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

任何 table 都可以作为任何值的元表，而一组相关的 table 也可以共享一个通用的元表，此元表描述了它们共同的行为。一个 table 甚至可以作为它自己的元表，用于描述其特有的行为。总之，任何搭配形式都是合法的。

在 Lua 代码中，只能设置 table 的元表。若要设置其他类型的值的元表，则必须通过 C 代码来完成[1]。在第 20 章中，将会看到标准的字符串程序库为所有的字符串都设置了一个元表，而其他类型在默认情况中都没有元表。

```
print(getmetatable("hi"))          --> table: 0x80772e0
print(getmetatable(10))            --> nil
```

① 设置该限制的目的是为了防止过度地使用某些特定于类型的（type-wide）元表。些设置通常会导致不可复用的代码。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

13.1 算术类的元方法

在本节中，会引入一个简单的示例，以说明如何使用元表。假设用 table 来表示集合，并且有一些函数用来计算集合的并集和交集等。为了保持名称空间的整齐，则将这些函数存入一个名为 Set 的 table 中。

```
Set = {}

-- 根据参数列表中的值创建一个新的集合
function Set.new (l)
  local set = {}
  for _, v in ipairs(l) do set[v] = true end
  return set
end

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

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

为了帮助检查此示例，还定义了一个用于打印集合的函数：

```
function Set.tostring (set)
  local l = {}     -- 用于存放集合中所有元素的列表
  for e in pairs(set) do
    l[#l + 1] = e
  end
  return "{" .. table.concat(l, ", ") .. "}"
end

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

假设使用加号（+）来计算两个集合的并集，那么就需要让所有用于表示集合的 table 共享一个元表，并且在该元表中定义如何执行一个加法操作。第一步是创建一个常规的 table，准备用作集合的元表：

```
local mt = {}    -- 集合的元表
```

下一步是修改 Set.new 函数。这个函数是用于创建集合的，在新版本中只加了一行，即将 mt 设置为当前所创建 table 的元表：

```
function Set.new (l)   -- 第 2 版
  local set = {}
  setmetatable(set, mt)
  for _, v in ipairs(l) do set[v] = true end
  return set
end
```

在此之后，所有由 Set.new 创建的集合都具有一个相同的元表：

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

最后，将元方法加入元表中。在本例中，这个元方法就是用于描述如何完成加法的__add 字段。

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

此后只要 Lua 试图将两个集合相加，它就会调用 Set.union 函数，并将两个操作数作为参数传入。可以使用加号来求集合的并集：

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

类似地，还可以使用乘号来求集合的交集：

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

在元表中，每种算术操作符都有对应的字段名。除了上述的__add 和__mul 外，还有__sub（减法）、__div（除法）、__unm（相反数）、__mod（取模）和__pow（乘幂）。此外，还可以定义__concat 字段，用于描述连接操作符的行为。

当两个集合相加时，可以使用任意一个集合的元表。然而，当一个表达式中混合了具有不同元表的值时，例如：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

Lua 会按照如下步骤来查找元表：如果第一个值有元表，并且元表中有__add 字段，那么 Lua 就以这个字段为元方法，而与第二个值无关；反之，如果第二个值有元表并含有__add 字段，Lua 就以此字段为元方法；如果两个值都没有元方法，Lua 就引发一个错误。因此，上例会调用 Set.union，而表达式 10 + s 和"hello" + s 也是一样的。

Lua 可以包含这些混合类型，但实现需要注意如果执行了 s = s + 8，那么在 Set.union 内部就会发生错误：

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

如果想要得到更清楚的错误消息，则必须在实际操作前显式地检查操作数的类型：

```
function Set.union (a, b)
  if getmetatable(a) ~= mt or getmetatable(b) ~= mt then
    error("attempt to 'add' a set with a non-set value", 2)
  end
  <与前例相同的内容>
```

注意，error 的第二个参数（上例中的 2）用于指示哪个函数调用造成了该错误消息。

13.2 关系类的元方法

元表还可以指定关系操作符的含义，元方法为__eq（等于）、__lt（小于）和__le（小于等于）。而其他 3 个关系操作符则没有单独的元方法，Lua 会将 a ~= b 转化为 not (a == b)，将 a > b 转化为 b < a，将 a >= b 转化为 b <= a。

在 Lua 4.0 之前，所有的顺序操作符都被转化为一种操作符（小于），例如，a <= b 转化为 not (b < a)。不过，这种转化遇到“部分有序（partial order）”就会发生错误。所谓“部分有序”是指，对于一种类型而言，并不是所有的值都能排序的。例如，大多数计算机中的浮点数就不是完全可以排序的。因为存在着一种叫“Not a Number（NaN）”的值。IEEE 754 是一份当前所有浮点数硬件都采用的事实标准，其中将 NaN 视为一种未定义的值，例如 0/0 的结果就是 NaN。标准规定了任何涉及 NaN 的比较都应返回 false（假）。这意味着 NaN <= x 永远为假，但是 x < NaN 也为假。因此，前面提到的将 a <= b 转化为 not (b < a)就不合法了。

在上面的集合示例中，也存在着类似的问题。在集合操作中<=通常表示集合间的包含关系：a <= b 通常意味着 a 是 b 的一个子集。根据这样的表示，仍有可能得到 a <= b 和 b < a 同时为假的情况。因此需要分别为__le（小于等于）和__lt（小于）提供实现：

```
mt.__le = function (a, b)    -- 集合包含
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
  for k in pairs(a) do
    if not b[k] then return false end
  end
  return true
end

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

最后，还可以定义集合的相等性判断：

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

有了这些定义后，就可以比较集合了：

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

与算术类的元方法不同的是，关系类的元方法不能应用于混合的类型。对于混合类型而言，关系类元方法的行为就模拟这些操作符在 Lua 中普通的行为。如果试图将一个字符串与一个数字作顺序性比较，Lua 会引发一个错误。同样，如果试图比较两个具有不同元方法的对象，Lua 也会引发一个错误。

等于比较永远不会引发错误。但是如果两个对象拥有不同的元方法，那么等于操作不会调用任何一个元方法，而是直接返回 false。这种行为模拟了 Lua 的普通行为。在 Lua 的普通行为中，字符串总是不等于数字的，与它们的值无关。另外，只有当两个比较对象共享一个元方法时，Lua 才调用这个等于比较的元方法。

13.3　库定义的元方法

各种程序库在元表中定义它们自己的字段是很普遍的方法。到目前为止介绍的所有元方法都只针对于 Lua 的核心，也就是一个虚拟机（virtual machine）。它会检测一个操作中的值是否有元表，这些元表中是否定义了关于此操作的元方法。从另一方面说，由于元表也是一种常规的 table，所以任何人、任何函数都可以使用它们。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

函数 tostring 就是一个典型的实例。在前面已介绍过 tostring 了，它能将各种类型的值表示为一种简单的文本格式：

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

函数 print 总是调用 tostring 来格式化其输出。当格式化任意值时，tostring 会检查该值是否有一个__tostring 的元方法。如果有这个元方法，tostring 就用该值作为参数来调用这个元方法。接下来由这个元方法完成实现的工作，它返回的结果也就是 tostring 的结果。

在集合的示例中，已定义了一个将集合表示为字符串的函数。接下来要做的就是设置元表中的__tostring 字段：

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

此后只要调用 print 来打印集合，print 就会调用 tostring 函数，进而调用到 Set.tostring：

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

函数 setmetatable 和 getmetatable 也会用到元表中的一个字段，用于保护元表。假设想要保护集合的元表，使用户既不能看也不能修改集合的元表。那么就需要用到字段__metatable。当设置了该字段时，getmetatable 就会返回这个字段的值，而 setmetatable 则会引发一个错误：

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

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

13.4 table 访问的元方法

算术类和关系类运算符的元方法都为各种错误情况定义了行为，它们不会改变语言的常规行为。但是 Lua 还提供了一种可以改变 table 行为的方法。有两种可以改变的 table 行为：查询 table 及修改 table 中不存在的字段。

13.4.1 __index 元方法

当访问一个 table 中不存在的字段时，得到的结果为 **nil**。这是对的，但并非完全正确。实际上，这些访问会促使解释器去查找一个叫__index 的元方法。如果没有这个元方法，那么访问结果如前述的为 **nil**。否则，就由这个元方法来提供最终结果。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

下面将介绍一个有关继承的典型示例。假设要创建一些描述窗口的 table，每个 table 中必须描述一些窗口参数，例如位置、大小及主题颜色等。所有这些参数都有默认值，因此希望在创建窗口对象时可以仅指定那些不同于默认值的参数。第一种方法是使用一个构造式，在其中填写那些不存在的字段。第二种方法是让新窗口从一个原型窗口处继承所有不存在的字段。首先，声明一个原型和一个构造函数，构造函数创建新的窗口，并使它们共享同一个元表：

```
Window = {}        -- 创建一个名字空间
-- 使用默认值来创建一个原型
Window.prototype = {x=0, y=0, width=100, height=100}
Window.mt = {}     -- 创建元表
-- 声明构造函数
function Window.new (o)
  setmetatable(o, Window.mt)
  return o
end
```

现在，来定义__index 元方法：

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

在这段代码之后，创建一个新窗口，并查询一个它没有的字段：

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

若 Lua 检测到 w 中没有某字段，但在其元表中却有一个__index 字段，那么 Lua 就会以 w（table）和"width"（不存在的 key）来调用这个__index 元方法。随后元方法用这个 key 来索引原型 table，并返回结果。

在 Lua 中，将__index 元方法用于继承是很普遍的方法，因此 Lua 还提供了一种更便捷的方式来实现此功能。__index 元方法不必一定是一个函数，它还可以是一个 table。当它是一个函数时，Lua 以 table 和不存在的 key 作为参数来调用该函数，这就如同上述内容。而当它是一个 table 时，Lua 就以相同的方式来重新访问这个 table。因此，前例中__index 的声明可以简单地写为：

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

现在，当 Lua 查找到元表的__index 字段时，发现__index 字段的值是一个 table，那么 Lua 就会在 Window.prototype 中继续查找。也就是说，Lua 会在这个 table 中重复这个访问过程，类似于执行这样的代码：

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

然后由这次访问给出想要的结果。

将一个 table 作为__index 元方法是一种快捷的、实现单一继承的方式。虽然将函数作为__index 来实现相同功能的开销较大，但函数更加灵活。可以通过函数来实现多重继承、缓存及其他一些功能。将在第 16 章中详细讨论这些继承形式。

如果不想在访问一个 table 时涉及到它的__index 元方法，可以使用函数 rawget。调用 rawget(t, i)就是对 table t 进行了一个“原始的（raw）”访问，也就是一次不考虑元表的简单访问。一次原始访问并不会加速代码执行[①]，但有时会用到它。

13.4.2 __newindex 元方法

__newindex 元方法与__index 类似，不同之处在于前者用于 table 的更新，而后者用于 table 的查询。当对一个 table 中不存在的索引赋值时，解释器就会查找__newindex 元方法。如果有这个元方法，解释器就调用它，而不是执行赋值。如果这个元方法是一个 table，解释器就在此 table 中执行赋值，而不是对原来的 table。此外，还有一个原始函数允许绕过元方法：调用 rawset(t, k, v)就可以不涉及任何元方法而直接设置 table t 中与 key k 相关联的 value v。

组合使用__index 和__newindex 元方法就可以实现出 Lua 中的一些强大功能，例如，只读的 table、具有默认值的 table 和面向对象编程中的继承。在本章后续内容中，会介绍其中的一些应用。关于面向对象编程则有单独章节进行介绍。

13.4.3 具有默认值的 table

常规 table 中的任何字段默认都是 nil。通过元表就可以很容易地修改这个默认值：

```
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
```

在调用 setDefault 后，任何对 tab 中存在字段的访问都将调用它的__index 元方法，而这个元方法会返回 0（这个元方法中 d 的值）。

setDefault 函数为所有需要默认值的 table 创建了一个新的元表。如果准备创建很多需要默认值的 table，这种方法的开销或许就比较大了。由于在元表中默认值 d 是与元方法关联在一

① 一次函数调用的开销就会抹杀用户所做的这些努力。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

起的，所以 setDefault 无法为所有 table 都使用同一个元表。若要让具有不同默认值的 table 都使用同一个元表，那么就需要将每个元表的默认值存放到 table 本身中。可以使用一个额外的字段来保持默认值。如果不担心名字冲突的话，可以使用“___”这样的 key 作为这个额外的字段：

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

如果担心名称冲突，那么要确保这个特殊 key 的唯一性也很容易。只需创建一个新的 table，并用它作为 key 即可：

```
local key = {}    -- 唯一的key
local mt = {__index = function (t) return t[key] end}
function setDefault (t, d)
  t[key] = d
  setmetatable(t, mt)
end
```

还有一种方法可以将 table 与其默认值关联起来：使用一个独立的 table，它的 key 为各种 table，value 就是各种 table 的默认值。不过，为了正确地实现这种做法，我们还需要一种特殊性质的 table，就是“弱引用 table（Weak Table）”。在这里我们就不使用它了，而是在第 17 章再讨论这个话题。

还有一种备忘录（memoize）元表的方法，它能使具有相同默认值的 table 复用同一个元表。不过，这项技术也需要用到弱引用 table，将在第 17 章中详细讨论。

13.4.4 跟踪 table 的访问

__index 和__newindex 都是在 table 中没有所需访问的 index 时才发挥作用的。因此，只有将一个 table 保持为空，才有可能捕捉到所有对它的访问。为了监视一个 table 的所有访问，就应该为真正的 table 创建一个代理。这个代理就是一个空的 table，其中__index 和__newindex 元方法可用于跟踪所有的访问，并将访问重定向到原来的 table 上。假设，我们想跟踪 table t 的访问。那么可以这么做：

```
t = {}   -- 原来的table（在其他地方创建的）

-- 保持对原table的一个私有访问
local _t = t
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
-- 创建代理
t = {}

-- 创建元表
local mt = {
   __index = function (t, k)
   print("*access to element " .. tostring(k))
   return _t[k]   -- 访问原来的table
  end,

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

这段代码跟踪了所有对 t 的访问：

```
> t[2] = "hello"
*update of element 2 to hello
> print(t[2])
*access to element 2
hello
```

但上例中的方法存在一个问题，就是无法遍历原来的 table。函数 pairs 只能操作代理 table，而无法访问原来的 table。

如果想要同时监视几个 table，无须为每个 table 创建不同的元表。相反，只要以某种形式将每个代理与其原 table 关联起来，并且所有代理都共享一个公共的元表。这个问题与上节所讨论的将 table 与其默认值相关联的问题类似。例如将原来的 table 保存在代理 table 的一个特殊的字段中。代码如下：

```
local index = {}              -- 创建私有索引

local mt = {                  -- 创建元表
   __index = function (t, k)
   print("*access to element " .. tostring(k))
   return t[index][k]   -- 访问原来的table
  end,

__newindex = function (t, k, v)
   print("*update of element " .. tostring(k) ..
                        " to " .. tostring(v))
t[index][k] = v   -- 更新原来的table
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  end
}

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

现在，若要监视 table t，唯一要做的就是执行：t = track(t)。

13.4.5 只读的 table

通过代理的概念，可以很容易地实现出只读的 table。只需跟踪所有对 table 的更新操作，并引发一个错误就可以了。由于无须跟踪查询访问，所以对于__index 元方法可以直接使用原 table 来代替函数。这也更简单，并且在重定向所有查询到原 table 时效率也更高。不过，这种做法要求为每个只读代理创建一个新的元表，其中__index 指向原来的 table。

```
function readOnly (t)
  local proxy = {}
  local mt = {       -- 创建元表
    __index = t,
    __newindex = function (t, k, v)
      error("attempt to update a read-only table", 2)
    end
  }
  setmetatable(proxy, mt)
  return proxy
end
```

下面是一个使用的示例，创建了一个表示星期的只读 table：

```
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
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 14 章　环　　境

Lua 将其所有的全局变量保存在一个常规的 table 中，这个 table 称为“环境（environment）”[①]。这种组织结构的优点在于，其一，不需要再为全局变量创造一种新的数据结构，因此简化了 Lua 的内部实现。另一个优点是，可以像其他 table 一样操作这个 table。为了便于实施这种操作，Lua 将环境 table 自身保存在一个全局变量_G 中[②]。例如，以下代码打印了当前环境中所有全局变量的名称：

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

在本章中，将看到几种关于环境操作的实用技术。

14.1　具有动态名字的全局变量

对于访问和设置全局变量，通常赋值操作就可以了。不过，有时也会用到一些元编程（meta-programming）的形式。例如，当操作一个全局变量时，而它的名称却存储在另一个变量中，或者需要通过运行时的计算才能得到。为了获取这个变量的值，许多程序员都试图写出这样的代码：

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

如果 varname 是 x，那么连接操作的结果就是字符串"return x"。这段代码就执行了这个字符串，并得到了 x 的值。然而，在这段代码中包含了一个新程序块的创建和编译。因此可以使用以下代码来完成相同的效果，但效率却比上例高出一个数量级：

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

正因为环境是一个常规的 table，才可以使用一个 key（变量名）去直接索引它。类似地，还可以动态地计算出一个名称，然后将一个值赋予具有该名称的全局变量：

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

不过注意，有些程序员对于该技能的运用就有些过度了，他们写出的_G["a"] = _G["var1"],

① 更为准确地说，Lua 是将其“全局的”变量保存在几个环境中，但是先暂时忽略这种多样性。
② _G._G 等于_G。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

其实就是简单的一句 a = var1。

上面问题的一般化形式是，允许使用动态的字段名，如"io.read"或"a.b.c.d"。如果直接写 _G["io.read"]则不会从 table io 中得到字段 read。但可以写一个函数 getfield 来实现这个效果，即通过调用 getfield("io.read")返回所要求的结果。这个函数是一个循环，从_G 开始逐个字段地深入求值：

```
function getfield (f)
  local v = _G    -- 从全局变量的table开始
  for w in string.gmatch(f, "[%w_]+") do
    v = v[w]
  end
  return v
end
```

依靠 string 库中的 gmatch 来遍历 f 中所有的单词[①]。

与之对应的设置字段的函数则稍显复杂。像 a.b.c.d = v 这样的赋值等价于以下代码：

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

也就是说，必须一直检索到最后一个名称，然后分别进行操作。下面这个函数 setfield 就完成了这项任务，并且创建路径中间那些不存在的 table。

```
function setfield (f, v)
  local t = _G                -- 从全局变量的table开始
  for w, d in string.gmatch(f, "([%w_]+)(%.?)") do
    if d == "." then          -- 是最后一个字段吗?
      t[w] = t[w] or {}       -- 如果不存在就创建table
      t = t[w]                -- 获取该table
    else                      -- 最后的字段
      t[w] = v                -- 完成赋值
    end
  end
end
```

上例中用到了一种字符串模式（pattern），通过这种模式就可以将字段名捕获到变量 w 中，并将一个可选的句号捕获到 d 中。通过调用上面这个函数：

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

便创建了两个 table：全局 t 和 t.x，并将 10 赋予 t.x.y：

```
print(t.x.y)     --> 10
print(getfield("t.x.y"))   --> 10
```

① “单词”就是一个或多个字母与下画线的序列。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

14.2 全局变量声明

Lua 中的全局变量不需要声明就可以使用。对于小型程序来说较为方便，但在大型程序中，一处简单的笔误就有可能造成难以发现的错误。不过这种性能可以改变。由于 Lua 将全局变量存放在一个普通的 table 中，则可以通过元表来改变其访问全局变量时的行为。

一种方法是简单地检测所有对全局 table 中不存在 key 的访问：

```
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,
})
```

执行过这段代码后，所有对全局 table 中不存在 key 的访问都将引发一个错误：

```
> print(a)
stdin:1: attempt to read undeclared variable a
```

但是该如何声明一个新的变量呢？其一是使用 rawset，它可以绕过元表：[1]

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

另外一种更简单的方法就是只允许在主程序块中对全局变量进行赋值，那么当声明以下变量时：

```
a = 1
```

就只需检查此赋值是否在主程序块中。这可以使用 debug 库，调用 debug.getinfo(2, "S") 将返回一个 table，其中的字段 what 表示了调用元方法的函数是主程序块还是普通的 Lua 函数，又或是 C 函数[2]。可以通过该函数将--newindex 元方法重写为：

```
--newindex = function (t, n, v)
  local w = debug.getinfo(2, "S").what
```

① 其中 or 与 false 的用法可以确保新的全局变量不会为 nil。
② 将在第 23 章中详细介绍 debug.getinfo。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    if w ~= "main" and w ~= "C" then
      error("attempt to write to undeclared variable " .. n, 2)
    end
    rawset(t, n, v)
  end
```

这个新版本还可以接受来自 C 代码的赋值，因为一般 C 代码都知道自己是做什么的。

为了测试一个变量是否存在，就不能简单地将它与 **nil** 比较。因为如果它为 **nil**，访问就会抛出一个错误。这时同样可以使用 rawget 来绕过元方法：

```
if rawget(_G, var) == nil then
  -- 'var'没有声明
  ...
end
```

正如前面提到的，不允许全局变量具有 nil 值，因为具有 nil 的全局变量都会被自动地认为是未声明的。但要纠正这个问题并不难，只需引入一个辅助 table 用于保存已声明变量的名称。一旦调用了元方法，元方法就检查该 table，以确定变量是否已声明过，代码如下所示：

```
local declaredNames = {}

setmetatable(_G, {
  --newindex = function (t, n, v)
   if not declaredNames[n] then
     local w = debug.getinfo(2, "S").what
     if w ~= "main" and w ~= "C" then
       error("attempt to write to undeclared variable "..n, 2)
   end
     declaredNames[n] = true
   end
   rawset(t, n, v)   -- 完成实际的设置
  end,

--index = function (_, n)
   if not declaredNames[n] then
     error("attempt to read undeclared variable "..n, 2)
   else
     return nil
   end
  end,
})
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

此时，即使是 x = nil 这样的赋值也可以起到声明全局变量的作用。

上述两种方法所导致的开销基本可以忽略不计。在第一种方法中，完全没有涉及到元方法的调用。第二种方法虽然会使程序调用到元方法，但只有当程序访问一个为 **nil** 的变量时才会发生。

有些 Lua 发行版本中包含一个叫 strict.lua 的模块，它实现了对全局变量的检查。究其本质就是使用了上述的技术。推荐在编写 Lua 代码时使用它，可以养成良好的习惯。

14.3 非全局的环境

关于“环境”的一大问题在于它是全局的，任何对它的修改都会影响程序的所有部分。例如，若安装一个元表用于控制全局变量的访问，那么整个程序都必须遵循这个规范。当使用某个库时，没有先声明就使用了全局变量，那么这个程序就无法运行了。

Lua 5 对这个问题进行了改进，它允许每个函数拥有一个自己的环境来查找全局变量。第一次听到这项机制可能会感觉不理解，毕竟一个用于记录全局变量的 table 本身也应该是全局的。在 15.3 节中看到几种基于该机制的技术，它们能使全局变量访问任何地方。

可以通过函数 setfenv[①]来改变一个函数的环境。该函数的参数是一个函数和一个新的环境 table。第一个参数除了可以指定为函数本身，还可以指定为一个数字，以表示当前函数调用栈中的层数。数字 1 表示当前函数，数字 2 表示调用当前函数的函数[②]，依此类推。

第一次天真地试用 setfenv 可能会带来糟糕的结果。如下代码：[③]

```
a = 1   -- 创建一个全局变量
-- 将当前环境改为一个新的空table
setfenv(1, {})
print(a)
```

会导致：

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

一旦改变了环境，所有的全局访问就都会使用新的 table。如果新 table 是空的，那么就会丢失所有的全局变量，包括_G。所以应该先将一些有用的值录入其中，例如原来的环境：

```
a = 1                    -- 创建一个全局变量
setfenv(1, {g = _G})     -- 改变当前的环境
```

① set function environment，设置函数环境。
② 由此可以很方便地写出一些辅助函数，在其中修改它们调用者的环境。
③ 必须将这段代码作为一个程序块运行。如果在交互模式中逐行地输入运行，那么每行都将成为一个函数，setfenv 调用也将只影响它所在的行。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
g.print(a)                  --> nil
g.print(g.a)                --> 1
```

此时访问“全局的”g 就会得到原来的环境，这个环境中包含了字段 print。

可以使用名字_G 来代替 g，从而重写前例：

```
setfenv(1, {_G = _G})
_G.print(a)                 --> nil
_G.print(_G.a)              --> 1
```

对于 Lua 来说，_G 只是一个普通的名字。当 Lua 创建最初的全局 table 时，只是将这个 table 赋予了全局变量_G，Lua 不会在意这个变量_G 的当前值。setfenv 不会在新环境中设置这个变量。但如果希望在新环境中引用最初的全局 table，一般使用_G 这个名称即可，如上例。

另一种组装新环境的方法是使用继承：

```
a = 1
local newgt = {}            -- 创建新环境
setmetatable(newgt, {--index = _G})
setfenv(1, newgt)           -- 设置它
print(a)                    --> 1
```

在这段代码中，新环境从原环境中继承了 print 和 a。然而，任何赋值都发生在新的 table 中。若误改了一个全局变量也没什么，仍然能通过_G 来修改原来的全局变量：

```
-- 继续前面的代码
a = 10
print(a)                    --> 10
print(_G.a)                 --> 1
_G.a = 20
print(_G.a)                 --> 20
```

每个函数及某些 closure 都有一个继承的环境。下面这段代码就演示了这种机制：

```
function factory ()
  return function ()
         return a           -- "全局的" a
       end
end

a = 3

f1 = factory()
f2 = factory()
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(f1())        --> 3
print(f2())        --> 3

setfenv(f1, {a = 10})
print(f1())        --> 10
print(f2())        --> 3
```

factory 函数创建了一个简单的 closure，这个 closure 返回了它的全局 a 的值。每次调用 factory 都会创建一个新的 closure 和一个属于该 closure 的环境。每个新创建的函数都继承了创建它的函数的环境。因此，上例中的 closure 都共享一个全局环境。在这个环境中 a 为 3，当调用 setfenv(f1, {a = 10})时，就改变了 f1 的环境，在新环境中 a 为 10。这期间 f2 的环境并未受到影响。

由于函数继承了创建其函数的环境。所以一个程序块若改变了它自己的环境，那么后续由它创建的函数都将共享这个新环境。这项机制对于创建名称空间是很有用的，会在第 15 章中看到这方面的介绍。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 15 章　模 块 与 包

通常，Lua 不会设置规则（policy）。相反，Lua 会提供许多强有力的机制来使开发者有能力实现出最适合的规则。然而，这种方法对于模块就不可行了。模块系统的一个主要目标是允许以不同的形式来共享代码。但若没有一项公共的规则就无法实现这样的共享。

Lua 从 5.1 版开始，为模块和包（package）[①]定义了一系列的规则。这些规则不需要语言引入额外的技能，程序员可以使用他们早已熟知的 table、函数、元表和环境来实现这些规则。然而，有两个重要的函数可以很容易通过这些规则，它们是 require（用于使用模块）和 module（用于创建模块）。程序员完全可以使用不同的规则来重新实现这两个函数。但是，新的实现可能会使程序无法使用外部模块，或者编写的模块无法被外部程序所使用。

从用户观点来看，一个模块就是一个程序库，可以通过 require 来加载。然后便得到了一个全局变量，表示一个 table。这个 table 就像是一个名称空间，其内容就是模块中导出的所有东西，例如函数和常量。一个规范的模块还应使 require 返回这个 table。

使用 table 来实现模块的优点在于，可以像操作普通 table 那样来操作模块，并且能利用 Lua 现有的功能来实现各种额外功能。在大多数语言中，模块不是“第一类值（first-class value）”，所以那些语言需要为模块实现一套专门的机制。在 Lua 中，可以轻易地实现所有这些功能。

例如，一个用户要调用一个模块中的函数。其中最简单的方法是：

```
require "mod"
mod.foo()
```

如果希望使用较短的模块名称，则可以为模块设置一个局部名称：

```
local m = require "mod"
m.foo()
```

还可以为个别函数提供不同的名称：

```
require "mod"
local f = mod.foo
f()
```

上述这些方法，都不需要来自于语言的显式支持，只需使用语言现有的内容。

① 一个包就是一系列的模块。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

15.1 require 函数

Lua 提供了一个名为 require 的高层函数用来加载模块，但这个函数只假设了关于模块的基本概念。对于 require 而言，一个模块就是一段定义了一些值[1]的代码。

要加载一个模块，只需简单地调用 require "<模块名>"。该调用会返回一个由模块函数组成的 table，并且还会定义一个包含该 table 的全局变量。然而，这些行为都是由模块完成的，而非 require。所以，有些模块会选择返回其他值，或者具有其他的效果。

即使知道某些用到的模块可能已加载了，但只要用到 require 就是一个良好的编程习惯。可以将标准库排除在此规则之外，因为 Lua 总是会预先加载它们。不过，有些用户还是喜欢为标准库中的模块使用显式的 require：

```
local m = require "io"
m.write("hello world\n")
```

以下代码详细说明了 require 的行为：

```
function require (name)
  if not package.loaded[name] then        -- 模块是否已加载?
    local loader = findloader(name)
    if loader == nil then
      error("unable to load module " .. name)
    end
    package.loaded[name] = true           -- 将模块标记为已加载
    local res = loader(name)              -- 初始化模块
    if res ~= nil then
      package.loaded[name] = res
    end
  end
  return package.loaded[name]
end
```

首先，它在 table package.loaded 中检查模块是否已加载。如果是的话，require 就返回相应的值。因此，只要一个模块已加载过，后续的 require 调用都将返回同一个值，不会再次加载它。

如果模块尚未加载，require 就试着为该模块找一个加载器（loader）[2]，会先在 table package.preload 中查询传入的模块名。如果在其中找到了一个函数，就以该函数作为模块的加

① 这些值可能是函数，或者包含函数的 table。
② 这一步在演示代码中以抽象函数 findloader 来表示。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

载器。通过这个 preload table，就有了一种通用的方法来处理各种不同的情况[1]。通常这个 table 中不会找到有关指定模块的条目，那么 require 就会尝试从 Lua 文件或 C 程序库中加载模块。

如果 require 为指定模块找到了一个 Lua 文件，它就通过 loadfile 来加载该文件。而如果找到的是一个 C 程序库，就通过 loadlib 来加载。注意，loadfile 和 loadlib 都只是加载了代码，并没有运行它们。为了运行代码，require 会以模块名作为参数来调用这些代码。如果加载器有返回值，require 就将这个返回值存储到 table package.loaded 中，以此作为将来对同一模块调用的返回值。如果加载器没有返回值，require 就会返回 table package.loaded 中的值。在本章后面会看到，一个模块还可以将返回给 require 的值直接放入 package.loaded 中。

上述代码中还有一个重要的细节，就是在调用加载器前，require 先将 **true** 赋予了 package.loaded 中的对应字段，以此将模块标记为已加载。这是因为如果一个模块要求加载另一个模块，而后者又要递归地加载前者。那么后者的 require 调用就会马上返回，从而避免了无限循环。

若要强制使 require 对同一个库加载两次的话，可以简单地删除 package.loaded 中的模块条目。例如，在成功地 require "foo"后，package.loaded["foo"]就不为 **nil** 了。下面代码就可以再次加载该模块：

```
package.loaded["foo"] = nil
require "foo"
```

在搜索一个文件时，require 所使用的路径与传统的路径有所不同。大部分程序所使用的路径就是一连串目录，指定了某个文件的具体位置。然而，ANSI C[2]却没有任何关于目录的概念。所以，require 采用的路径是一连串的模式（pattern），其中每项都是一种将模块名转换为文件名的方式。进一步说，这种路径中的每项都是一个文件名，每项中还可以包含一个可选的问号。require 会用模块名来替换每个“?”，然后根据替换的结果来检查是否存在这样一个文件。如果不存在，就会尝试下一项。路径中的每项以分号[3]隔开。例如，假设路径为：

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

那么，调用 require "sql"就会试着打开以下文件：

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

require 函数只处理了分号（作为各项之间的分隔符）和问号。其他例如目录分隔符或文

① 例如，静态链接到 Lua 的 C 程序库。
② Lua 运行的抽象平台。
③ 在大多数操作系统中，这个字符很少作为文件名的一部分。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

件扩展名，都由路径自己定义。

require 用于搜索 Lua 文件的路径存放在变量 package.path 中。当 Lua 启动后，便以环境变量 LUA_PATH 的值来初始化这个变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。在使用 LUA_PATH 时，Lua 会将其中所有的子串";;"替换成默认路径。例如，假设 LUA_PATH 为"mydir/?.lua;;"，那么最终路径就是"mydir/?.lua"，并紧随默认路径。

如果 require 无法找到与模块名相符的 Lua 文件，它就会找 C 程序库。这类搜索会从变量 package.cpath（相对于 package.path）获取路径。而这个变量则是通过环境变量 LUA_CPATH（相对于 LUA_PATH）来初始化。在 UNIX 中，它的值一般是这样的：

```
./?.so;/usr/local/lib/lua/5.1/?.so
```

注意，文件的扩展名是由路径定义的（例如，上例中使用的.so）。而在 Windows 中，此路径通常可以是这样的：

```
.\?.dll;C:\Program Files\Lua501\dll\?.dll
```

当找到一个 C 程序库后，require 就会通过 package.loadlib 来加载它，loadlib 在 8.2 节中已讨论过。C 程序库与 Lua 程序块是不同的，它没有定义一个单一的主函数，而是导出了几个 C 函数。具有良好行为的 C 程序库应该导出一个名为“luaopen_<模块名>”的函数。require 会在链接完程序库后，尝试调用这个函数。将在 26.2 节中讨论如何编写 C 程序库。

一般通过模块的名称来使用它们。但有时必须将一个模块改名，以避免名称冲突。一种典型的情况是，在测试中需要加载同一模块的不同版本。对于一个 Lua 模块来说，其内部名称不是固定的，可以轻易地编辑它以改变其名称。但是却无法编辑一个二进制数据模块中 luaopen_*函数的名称。为了允许这种重命名，require 用到了一个小技巧：如果一个模块名中包含了连字符，require 就会用连字符后的内容来创建 luaopen_*函数名。例如，若一个模块名为 a-b，require 就认为它的 open 函数名为 luaopen_b，而不是 luaopen_a-b[①]。因此，如果要使用的两个模块名都为 mod，那么可以将其中一个重命名为 v1-mod（或者-mod，或其他类似形式）。当调用 m1 = require "v1-mod"时，require 会找到改名后的文件 v1-mod，并将其中的函数 luaopen_mod 作为 open 函数。

15.2 编写模块的基本方法

在 Lua 中创建一个模块最简单的方法是：创建一个 table，并将所有需要导出的函数放入其中，最后返回这个 table。以下代码演示这种方法。注意，将 inv 声明为程序块的局部变量，就是将其定义成一个私有的名称。

① 也不是一个合法的 C 名称。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
complex = {}

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

-- 定义一个常量'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

local function inv (c)
  local n = c.r^2 + c.i^2
  return complex.new(c.r/n, -c.i/n)
end

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

return complex
```

上例中使用 table 编写模块时，没有提供与真正模块完全一致的功能性，首先，必须显式地将模块名放到每个函数定义中。其次，一个函数在调用同一模块中的另一个函数时，必须限定被调用函数的名称。可以使用一个固定的局部名称（例如 M）来定义和调用模块内的函数，然后将这个局部名称赋予模块的最终名称。通过这种方法，可以将上例改写为：

```
local M = {}
complex = M          -- 模块名

M.i = {r=0, i=1}
function M.new (r, i) return {r=r, i=i} end

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
<如前>
```

只要一个函数调用了同一模块中另一个函数（或者递归地调用自己），就仍需要一个前缀名称。但至少两个函数之间的连接不再需要依赖于模块名，并且也只需在整个模块中的一处写出模块名。实际上，可以完全避免写模块名，因为 require 会将模块名作为参数传给模块：

```
local modname = ...
local M = {}
_G[modname] = M

M.i = {r=0, i=1}
<如前>
```

经过这样的修改，若需要重命名一个模块，只需重命名并定义它的文件就可以了。

另一项小改进与结尾的 return 语句有关。若能将所有与模块相关的设置任务集中在模块开头，会更好。消除 return 语句的一种方法是，将模块 table 直接赋予 package.loaded：

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
  <如前>
```

通过这样的赋值，就不需要在模块结尾返回 M 了。注意，如果一个模块无返回值的话，require 就会返回 package.loaded[modname]的当前值。

15.3 使用环境

创建模块的基本方法的缺点在于，它要求程序员投入一些额外的关注。当访问同一模块中的其他公共实体时，必须限定其名称。并且，只要一个函数的状态从私有改为公有（或从公有改为私有），就必须修改调用。另外，在私有声明中也很容易忘记关键字 **local**。

“函数环境”是一种有趣的技术，它能够解决所有上述创建模块时遇到的问题。基本想法就是让模块的主程序块有一个独占的环境。这样不仅它的所有函数都可共享这个 table，而且它的所有全局变量也都记录在这个 table 中。还可以将所有公有函数声明为全局变量，这样它们就都自动地记录在一个独立的 table 中了。模块所要做的就是将这个 table 赋予模块名和 package.loaded。以下代码片段演示了这种技术：

```
local modname = ...
local M = {}
_G[modname] = M
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
package.loaded[modname] = M
setfenv(1, M)
```

此时，当声明函数 add 时，它就成为了 complex.add：

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

此外，在调用同一模块的其他函数时，也不再需要前缀。例如，add 会从其环境中得到 new，也就是 complex.new。

这种方法为模块提供了一种良好的支持，并且只引入了一点额外的工作。此时完全不需要前缀，并且调用一个导出的函数与调用一个私有函数没有什么区别。如果程序员忘记了写 **local** 关键字，那么也不会污染全局名称空间。只会将一个私有函数变成了公有而已。

还缺少什么？是的，那就是访问其他模块。当创建了一个空 table M 作为环境后，就无法访问前一个环境中全局变量了。以下提出几种重获访问的方法，每种方法各有其优缺点。

最简单的方法是继承，就像之前看到的那样：

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
setmetatable(M, {__index = _G})
setfenv(1, M)
```

必须先调用 setmetatable 再调用 setfenv，因为通过这种方法，模块就能直接访问任何全局标识了，每次访问只需付出很小的开销。这种方法导致了一个后果，即从概念上说，此时的模块中包含了所有的全局变量。例如，某人可以通过你的模块来调用标准的正弦函数：complex.math.sin(x)。[①]

还有一种更快捷的方法来访问其他模块，即声明一个局部变量，用以保存对旧环境的访问：

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
local _G = _G
setfenv(1, M)
```

此时必须在所有全局变量的名称前加“_G .”。由于没有涉及到元方法，这种访问会比前面的方法略快。

① Perl 的 package 系统也有这种特性。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一种更正规的方法是将那些需要用到的函数或模块声明为局部变量：

```
-- 模块设置
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M

-- 导入段:
-- 声明这个模块从外界所需的所有东西
local sqrt = math.sqrt
local io = io

-- 在这句之后就不再需要外部访问了
setfenv(1, M)
```

这种技术要求做更多的工作，但是它能清晰地说明模块的依赖性。同时，较之前面的两种方法，它的运行速度也更快。

15.4 module 函数

读者或许注意到了，前面几个示例中的代码形式。它们都以相同的模式开始：

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
  <setup for external access>
setfenv(1, M)
```

Lua 5.1 提供了一个新函数 module，它囊括了以上这些功能。在开始编写一个模块时，可以直接用以下代码来取代前面的设置代码：

```
module(...)
```

这句调用会创建一个新的 table，并将其赋予适当的全局变量和 loaded table，最后还会将这个 table 设为主程序块的环境。

默认情况下，module 不提供外部访问。必须在调用它前，为需要访问的外部函数或模块声明适当的局部变量。也可以通过继承来实现外部访问，只需在调用 module 时加一个选项 package.seeall。这个选项等价于以下代码：

```
setmetatable(M, {--index = _G})
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

因而只需这么做：

```
module(..., package.seeall)
```

在一个模块文件的开头有了这句调用后，后续所有的代码都可以像普通的 Lua 代码那样编写了。不需要限定模块名和外部名字，同样也不需要返回模块 table。要做的只是加上这么一句调用。

module 函数还提供了一些额外的功能。虽然大部分模块不需要这些功能，但有些发行模块可能需要一些特殊处理（例如，一个模块中同时包含 C 函数和 Lua 函数）。module 在创建模块 table 之前，会先检查 package.loaded 是否已包含了这个模块，或者是否已存在与模块同名的变量。如果 module 由此找到了这个 table，它就会复用该 table 作为模块。也就是说，可以用 module 来打开一个已创建的模块。如果没有找到模块 table，module 就会创建一个模块 table。然后在这个 table 中设置一些预定义的变量，包括：_M，包含了模块 table 自身（类似于_G）；_NAME，包含了模块名（传给 module 的第一个参数）；_PACKAGE，包含了包（package）的名称（详见下节）。

15.5 子模块与包

Lua 支持具有层级性的模块名，可以用一个点来分隔名称中的层级。假设，一个模块名为 mod.sub，那么它就是 mod 的一个子模块。因此，可以认为模块 mod.sub 会将其所有值都定义在 table mod.sub 中，也就是一个存储在 table mod 中且 key 为 sub 的 table。一个“包（Package）”就是一个完整的模块树，它是 Lua 中发行的单位。

当 require 一个模块 mod.sub 时，require 会用原始的模块名“mod.sub”作为 key 来查询 table package.loaded 和 package.preload，其中，模块名中的点在搜索中没有任何含义。

然而，当搜索一个定义子模块的文件时，require 会将点转换为另一个字符，通常就是系统的目录分隔符[①]。转换之后 require 就像搜索其他名称一样来搜索这个名称。例如，假设路径为：

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

并且目录分隔符为“/”，那么调用 require "a.b"就会尝试打开以下文件：

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

通过这样的加载策略，就可以将一个包中的所有模块组织到一个目录中。例如，一个包

① UNIX 上为“/”，Windows 上为“\”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

中有模块 p、p.a 和 p.b，那么它们对应的文件名就分别为 p/init.lua，p/a.lua 和 p/b.lua，它们都是目录 p 下的文件。

Lua 使用的目录分隔符是编译时配置的，可以是任意的字符串[1]。例如，在没有目录层级的系统中，就可以使用“_”作为“目录分隔符”。那么 require "a.b"就会搜索到文件 a_b.lua。

C 函数名中不能包含点，因此一个用 C 编写的子模块 a.b 无法导出函数 luaopen_a.b。所以，require 会将点转换为下画线。例如，一个名为 a.b 的 C 程序库就应将其初始化函数命名为 luaopen_a_b。在此又可以巧用连字符，来实现一些特殊的效果。例如，有一个 C 程序库 a，现在想将它作为 mod 的一个子模块，那么就可以将文件名改为 mod/-a。当执行 require "mod.-a" 时，require 就会找到改名后的文件 mod/-a 及其中的函数 luaopen_a。

作为一项扩展功能，require 在加载 C 子模块时还有一些选项。当 require 加载子模块时，无法找到对应的 Lua 文件或 C 程序库。它就会再次搜索 C 路径，不过这次将以包的名称来查找。例如，一个程序 require 子模块 a.b.c，当无法找到文件 a/b/c 时，再次搜索就会找到文件 a。如果找到了 C 程序库 a，require 就查看该程序库中是否有 open 函数 luaopen_a_b_c。这项功能使得一个发行包可以将几个子模块组织到一个单一 C 程序库中，并且具有各自的 open 函数。

module 函数也为子模块提供了显式的支持。当我们创建一个子模块时，调用 module ("a.b.c")，module 就会将环境 table 放入变量 a.b.c，也就是“table a 中的 table b 中的 table c”。如果这些中间的 table 不存在，module 就会创建它们。否则，就复用它们。

从 Lua 的观点看，同一个包中的子模块除了它们的环境 table 是嵌套的之外，它们之间并没有显式的关联性。require 模块 a 并不会自动地加载它的任何子模块。同样，require 子模块 a.b 也并不会自动地加载 a。当然只要愿意，包的实现者完全可以实现这种关联。例如，模块 a 的一个子模块在加载时会显式地加载 a。

① 注意，Lua 不知道“目录”的概念。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 16 章　面向对象编程

Lua 中的 table 就是一种对象，这句话可以从 3 个方面来证实。首先，table 与对象一样可以拥有状态。其次，table 也与对象一样拥有一个独立于其值的标识（一个 self）。例如，两个具有相同值的对象（table）是两个不同的对象。最后，table 与对象一样具有独立于创建者和创建地的生命周期。

对象有其自己的操作。同样 table 也有这些操作：

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

上面的代码创建了一个新函数，并将该函数存入 Account 对象的 withdraw 字段中。则可进行如下调用：

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

这种函数就是所谓的“方法（Method）”。不过，在函数中使用全局名称 Account 是一个不好的编程习惯。因为这个函数只能针对特定对象工作，并且，这个特定对象还必须存储在特定的全局变量中。如果改变了对象的名称，withdraw 就再也不能工作了：

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

这种行为违反了前面提到的对象特性，即对象拥有独立的生命周期。

有一种灵活的方法，即指定一项操作所作用的“接受者”。因此需要一个额外的参数来表示该接受者。这个参数通常称为 self 或 this：

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

此时当调用该方法时，必须指定其作用的对象：

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

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

通过对 self 参数的使用，还可以针对多个对象使用同样的方法：

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

使用 self 参数是所有面向对象语言的一个核心。大多数面向对象语言都能对程序员隐藏部分 self 参数，从而使得程序员不必显式地声明这个参数[①]。Lua 只需使用冒号，则能隐藏该参数。即可将上例重写为：

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

调用时可写为：

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

冒号的作用是在一个方法定义中添加一个额外的隐藏参数，以及在一个方法调用中添加一个额外的实参。冒号只是一种语法便利，并没有引入任何新的东西。例如，用点语法来定义一个函数，并用冒号语法调用它。反之，只要能正确地处理那个额外参数即可：

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

function Account:deposit (v)
  self.balance = self.balance + v
end

Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

现在的对象已有一个标识、一个状态和状态之上的操作。不过还缺乏一个类（class）系统、继承和私密性（privacy）。首先解决第一个问题，如何创建多个具有类似行为的对象？更准确地说，如何创建多个 account 账户对象？

16.1 类

一个类就像是一个创建对象的模具。有些面向对象语言提供了类的概念，在这些语言中

① 程序员仍可以在其方法内部使用 self 或 this 变量。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

每个对象都是某个特定类的实例。Lua 则没有类的概念，每个对象只能自定义行为和形态。不过，要在 Lua 中模拟类也并不困难，可以参照一些基于原型的语言，例如 Self 和 NewtonScript。在这些语言中，对象是没有“类型”的（objects have no classes）。而是每个对象都有一个原型（prototype）。原型也是一种常规的对象，当其他对象（类的实例）遇到一个未知操作时，原型会先查找它。在这种语言中要表示一个类，只需创建一个专用作其他对象（类的实例）的原型。类和原型都是一种组织对象间共享行为的方式。

在 Lua 中实现原型很简单，使用 13.4.1 节中所述的继承即可。更准确地说，如果有两个对象 a 和 b，要让 b 作为 a 的一个原型，只需输入如下语句：

```
setmetatable(a, {--index = b})
```

在此之后，a 就会在 b 中查找所有它没有的操作。若将 b 称为是对象 a 的类，只不过是术语上的一个变化。

回到先前银行账号的示例。为了创建更多与 Account 行为类似的账号，可以让这些新对象从 Account 行为中继承这些操作。具体做法就是使用--index 元方法。可以应用一项小优化，则无须创建一个额外的 table 作为账户对象的元表。而是使用 Account table 自身作为元表：

```
function Account:new (o)
  o = o or {}   -- 如果用户没有提供table，则创建一个
  setmetatable(o, self)
  self.__index = self
  return o
end
```

当调用 Account:new 时，self 就等于 Account。因此可以直接使用 Account 来代替 self。不过，当引入类继承时，使用 self 则会更为准确。在这段代码之后，创建一个新账户或调用一个方法时会发生什么呢？

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

当创建新账户时，a 会将 Account（Account::new 调用中的 self）作为其元表。而当调用 a:deposit(100.00)时，就是调用了 a.deposit(a, 100.00)。因此冒号只不过是一个“语法糖”。当 Lua 无法在 table a 中找到条目“deposit”时，它会进一步搜索元表的--index 条目。最终的调用情况为：

```
getmetatable(a).--index.deposit(a, 100.00)
```

a 的元表是 Account，Account.--index 也是 Account[1]。因此，上面这个表达式可以简化为：

① 因为在方法 new 中做了 self.--index = self。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
Account.deposit(a, 100.00)
```

结果为 Lua 调用了原来的 deposit 函数，但传入 a 作为 self 参数。因此新账户 a 从 Account 继承了 deposit 函数。同样，它还能从 Account 继承所有的字段。

继承不仅可以作用于方法，还可以作用于所有其他在新账户中没有的字段。因此，一个类不仅可以提供方法，还可以为实例中的字段提供默认值。回忆一下，在第一个 Account 定义中，有一个 balance 字段为 0。如果在创建新账户时没有提供 balance 的初值，那么它就会继承这个默认值：

```
b = Account:new()
print(b.balance)    --> 0
```

在 b 上调用 deposit 方法时，self 就是 b，就相当于执行了：

```
b.balance = b.balance + v
```

在第一次调用 deposit 时，对表达式 b.balance 的求值结果为 0，然后一个初值被赋予了 b.balance。后续对 b.balance 的访问就不会再涉及到--index 元方法了，因为此时 b 已有自己的 balance 字段。

16.2 继　　承

由于类也是对象，它们也可以从其他类获得方法。这种行为就是一种继承，可以很容易地在 Lua 中。

假设有一个基类 Account：

```
Account = {balance = 0}

function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.--index = self
  return o
end

  function Account:deposit (v)
  self.balance = self.balance + v
end

function Account:withdraw (v)
  if v > self.balance then error"insufficient funds" end
  self.balance = self.balance - v
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

若想从这个类派生出一个子类 SpecialAccount，以使客户能够透支。则先需要创建一个空的类，从基类继承所有的操作：

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

直到现在，SpecialAccount 还只是 Account 的一个实例。如下所示：

```
s = SpecialAccount:new{limit=1000.00}
```

SpecialAccount 从 Account 继承了 new，就像继承其他方法一样。不过这次 new 在执行时，它的 self 参数表示为 SpecialAccount。因此，s 的元表为 SpecialAccount，SpecialAccount 中字段--index 的值也是 SpecialAccount。s 继承自 SpecialAccount，而 SpecialAccount 又继承自 Account。当执行：

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

Lua 在 s 中找不到 deposit 字段时，就会查找 SpecialAccount。如果仍找不到 deposit 字段，就查找 Account。最终会在那里找到 deposit 的原始实现。

SpecialAccount 之所以特殊是因为可以重定义那些从基类继承的方法。编写一个方法的新实现只需：

```
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，当调用 s:withdraw(200.00)时，Lua 就不会在 Account 中查找了。因为 Lua 会在 SpecialAccount 中先找到 withdraw 方法。由于 s.limit 为 1000.00，程序会执行取款，并使 s 变成一个负的余额。

Lua 中的对象有一个特殊现象，就是无须为指定一种新行为而创建一个新类。如果只有一个对象需要某种特殊的行为，那么可以直接在该对象中实现这个行为。例如，账户 s 表示一个特殊的客户，这个客户的透支额度总是其余额的 10%。那么可以只修改这个对象：

```
function s:getLimit ()
  return self.balance * 0.10
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在这段代码后，调用 s:withdraw(200.00)还是会执行 SpecialAccount 的 withdraw。但 withdraw 所调用的 self:getLimit 则是上面这个定义。

16.3 多重继承

由于 Lua 中的对象不是原生的（Primitive），因此在 Lua 中进行面向对象编程时有几种方法。上面介绍了一种使用--index 元方法的做法，这是集简易、性能和灵活性于一体的做法。另外还有一些其他的做法，可能更适用于某些特殊的情况。在此将介绍另一种做法，可以在 Lua 中实现多重继承。

这种做法的关键在于用一个函数作为--index 元字段。例如，若在一个 table 的元表中，--index 字段为一个函数。那么只要 Lua 在原来的 table 中找不到一个 key，就会调用这个函数。基于这点，就可以让--index 函数在其他地方查找缺失的 key。

多重继承意味着一个类可以具有多个基类。因此无法使用一个类中的方法来创建子类，而是需要定义一个特殊的函数来创建。下面的 createClass 就是这样的函数，它会创建一个 table 表示新类，其中一个参数表示新类的所有基类。创建时它会设置元表中的--index 元方法，而多重继承正是在这个--index 元方法中完成的。虽然是多重继承，但每个对象实例仍属于单个类，并且都在这个类中查找所有的方法。因此，类和基类之间的关系不同于类和实例之间的关系。尤其是一个类不能同时作为其实例和子类的元表。在以下代码中，将类作为其实例的元表，并创建了另一个 table 作为类的元表。

```
-- 在table 'plist'中查找'k'
 local function search (k, plist)
  for i=1, #plist do
   local v = plist[i][k]     -- 尝试第 i 个基类
   if v then return v end
  end
end

function createClass (...)
 local c = {}        -- 新类
 local parents = {...}

-- 类在其父类列表中的搜索方法
 setmetatable(c, {--index = function (t, k)
  return search(k, parents)
 end})

-- 将'c'作为其实例的元表
 c.--index = c
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
-- 为这个新类定义一个新的构造函数（construction）
  function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
  end

return c            -- 返回新类
end
```

接下来是一个使用 createClass 的例子。假设有两个类，一个是前面提到的 Account 类；另一个是 Named 类，它有两个方法 setname 和 getname：

```
Named = {}
function Named:getname ()
  return self.name
end

function Named:setname (n)
  self.name = n
end
```

要创建一个新类 NamedAccount，同时从 Account 和 Named 派生，那么只需调用 createClass：

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

如下要创建并使用实例：

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

现在，来研究最后代码是如何工作的。首先，Lua 在 account 中无法找到字段“getname”。因此，就查找 account 元表中的--index 字段，该字段为 NamedAccount。由于在 NamedAccount 也无法提供字段“getname”。因此，Lua 查找 NamedAccount 元表中的--index 字段。由于这个字段也是一个函数，Lua 就调用了它。该函数则先在 Account 中查找“getname”。未找到后，继而查找 Named。最终在 Named 中找到了一个非 nil 的值，即为搜索的最终结果。

由于这项搜索具有一定的复杂性，则多重继承的性能不如单一继承。有一种改进性能的简单做法是将继承的方法复制到子类中。通过这种技术，类的--index 元方法如下所示：

```
setmetatable(c, {--index = function (t, k)
  local v = search(k, parents)
  t[k] = v        -- 保存下来，以备下次访问
  return v
end})
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

用了这种技术后，访问继承的方法就能像访问局部方法一样快了。但缺点是当系统运行后就较难修改方法的定义，因为这些修改不会沿着继承体系向下传播。

16.4 私 密 性

许多人认为私密性应成为面向对象语言不可或缺的一部分，每个对象的状态都应该是由它自己掌握。在一些面向对象语言中，例如 C++和 Java，能控制对象中的字段或方法是否在对象之外可见。而对于其他语言，例如 Smalltalk，规定所有的变量都是私有的，但所有的方法却都是公有的。第一个面向对象语言 Simula 则不提供任何形式的私密性保护。

Lua 在设计对象时，没有提供私密性机制，这具体章节已看到了。一方面这是因为使用了普通的结构（table）来表示对象，另一方面也反映了 Lua 某些基本的设计决定。Lua 并不打算构建需要许多程序员长期投入的大型程序。相反，Lua 定位于开发中小型程序，这些程序通常是一个更大系统的一部分。而参与编程的程序员一般只有一名或几名，甚至还可以是非程序员。因此，Lua 尽量避免过多冗余和人为限制。如果不想访问一个对象中的内容，则无须进行操作。

Lua 的另外一项设计目标是灵活性。Lua 提供给程序员各种元机制，以使他们能模拟许多不同的机制。虽然在 Lua 对象的基础设计中没有提供私密性机制。但可以用其他方法来实现对象，从而获得对象的访问控制。这种实现不常用，只做基本的了解，它既探索了 Lua 中的某些知识又可以成为其他问题的解决方案。

这种做法的基本思想是，通过两个 table 来表示一个对象。一个 table 用来保存对象的状态；另一个用于对象的操作，或称为“接口”。对象本身是通过第二个 table 来访问的，即通过其接口的方法来访问。为了避免未授权的访问，表示状态的 table 不保存在其他 table 中，而只是保存在方法的 closure 中。例如，若使用这种设计来表示一个银行账户，可以调用下面这个工厂函数来创建新的账户对象：

```
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 {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    withdraw = withdraw,
    deposit = deposit,
    getBalance = getBalance
  }
end
```

这个函数先创建了一个 table，用于保存对象的内部状态，并将其存储在局部变量 self 中。然后再创建对象的方法。最后，函数创建并返回一个供外部使用的对象，其中将方法名与真正的方法实现匹配起来。区别关键在于，这些方法不需要额外的 self 参数，因为它们可以直接访问 self 变量。由于没有了额外的参数，也就无须使用冒号语法来操作对象。则可以像普通函数那样来调用这些方法：

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())     --> 60
```

这种设计给予存储在 self table 中所有东西完全的私密性。当 newAccount 返回后，就无法直接访问这个 table 了。只能通过 newAccount 中创建的函数来访问它。上例只将一个成员变量放到了私有 table 中，其实可以将一个对象中所有的私有部分都存入这个 table。另外还可以定义私有的方法，它们类似于公有方法，但不放入接口中。例如，该账户可以给那些余额大于某个值的用户额外 10%的信用额度，但是又不想让用户访问到这些计算细节。那么可以将这个功能按以下方法实现：

```
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 + extra()
   end

  <如前>
```

与前一个示例一样，任何用户都无法直接访问 extra 函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

16.5 单一方法（single-method）做法

上述面向对象编程的做法有一种特殊情况，就是当一个对象只有一个方法时，可以不用创建接口 table，但要将这个单独的方法作为对象表示来返回。如果无法理解，请参阅 7.1 节中的内容。7.1 节介绍了如何构造一个迭代器函数，那个函数将状态保存为 closure。一个具有状态的迭代器是一个单一方法对象。

单一方法对象还有一种情况，若这个方法是一个调度（dispatch）方法，它根据某个参数来完成不同的操作。则可以这样来实现一个对象：

```
function newObject (value)
  return function (action, v)
    if action == "get" then return value
    elseif action == "set" then value = v
    else error("invalid action")
    end
  end
end
```

如下所示：

```
d = newObject(0)
print(d("get"))    --> 0
d("set", 10)
print(d("get"))    --> 10
```

这种非传统的对象实现方式是很高效的。语句 d("set", 10)虽然有些奇特，但只比传统的 d:set(10)多出两个字符。每个对象都用一个 closure，这比都用一个 table 更高效。虽然无法实现继承，却拥有了完全的私密性控制。访问一个对象状态只有一个方式，那就是通过它的单一方法。

Tcl/Tk 对它的窗口部件使用了类似的做法。在 Tk 中，一个窗口部件的名称就是一个函数名，通过这个函数就可以完成所有针对该部件的操作。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 17 章　弱引用 table

Lua 采用了自动内存管理。一个程序只需创建对象，而无须删除对象。通过使用垃圾收集机制，Lua 会自动地删除那些已成为垃圾的对象。这减轻了程序员在内存管理方面的负担，更重要的是将程序员从许多内存相关的 bug（例如无效指针、内存泄漏）中解放出来。

Lua 的垃圾收集器与一些其他的收集器有所不同，它没有环形引用的问题。当用到环形数据结构时，无须作出任何特殊的处理，它们也可以像其他数据一样被正常回收。不过，有时即使是再聪明的收集器也需要帮助。垃圾收集器无法解决所有内存管理的问题。

垃圾收集器只能回收那些它认为是垃圾的东西，它不会回收那些用户认为是垃圾的东西。一个典型的例子就是栈，栈通常由一个数组和一个表示顶部的索引来实现。这个数组的有效部分总是向顶部扩展的，但 Lua 却不知道。如果弹出一个元素时只是简单地递减顶部索引，那么这个仍留在数组中的对象对于 Lua 来说就不是垃圾。同理，对于那些存储在全局变量中的对象，即使程序不会再用到它们，但对于 Lua 来说它们也不是垃圾。在这两种情况中，都需要由用户来将这些对象变量赋值为 nil，这样才能使它们得以释放。

不过，简单地清除引用可能还不够。有些情况需要程序和收集器之间进行更多的协作。例如，如果要将一些对象放在一个数组中，这看似很简单，好像只需把每个对象插入数组即可。但是，当一个对象处于数组中时，它就无法被回收。这是因为即使当前没有其他地方在使用它，但数组仍引用着它。除非用户告诉 Lua 这项引用不应该阻碍此对象的回收，否则 Lua 是无从得知这个事实的。

弱引用 table（weak table）就是这样一种机制，用户能用它来告诉 Lua 一个引用不应该阻碍一个对象的回收。所谓“弱引用（weak reference）”就是一种会被垃圾收集器忽视的对象引用。如果一个对象的所有引用都是弱引用，那么 Lua 就可以回收这个对象了，并且还可以以某种形式来删除这些弱引用本身。Lua 用“弱引用 table”来实现“弱引用”，一个弱引用 table 就是一个具有弱引用条目的 table。如果一个对象只被一个弱引用 table 所持有，那么最终 Lua 是会回收这个对象的。

table 中有 key 和 value，这两者都可以包含任意类型的对象。通常，垃圾收集器不会回收一个可访问 table 中作为 key 或 value 的对象。也就是说，这些 key 和 value 都是强引用（strong reference），它们会阻止对其所引用对象的回收。在一个弱引用 table 中，key 和 value 是可以回收的。有 3 种弱引用 table：具有弱引用 key 的 table、具有弱引用 value 的 table、同时具有两种弱引用 table。不论是哪种类型的弱引用 table，只要有一个 key 或 value 被回收了，那么它们所在的整个条目都会从 table 中删除。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一个 table 的弱引用类型是通过其元表中的--mode 字段来决定的。这个字段的值应为一个字符串，如果这个字符串中包含字母'k'，那么这个 table 的 key 是弱引用的；如果这个字符串中包含字母'v'，那么这个 table 的 value 是弱引用的。下面这个示例虽然是人为制造的，但演示了弱引用 table 的一些基本行为：

```
a = {}
b = {__mode = "k"}
setmetatable(a, b)      -- 现在'a'的key就是弱引用
key = {}                -- 创建第一个key
a[key] = 1
key = {}                -- 创建第二个key
a[key] = 2
collectgarbage()        -- 强制进行一次垃圾收集
for k, v in pairs(a) do print(v) end
--> 2
```

在本例中，第二句赋值 key = {}会覆盖第一个 key。当收集器运行时，由于没有其他地方在引用第一个 key，因此第一个 key 就被回收了，并且 table 中的相应条目也被删除了。至于第二个 key，变量 key 仍引用着它，因此它没有被回收。

注意，Lua 只会回收弱引用 table 中的对象。而像数字和布尔这样的“值”是不可回收的。例如，对于一个插入 table 的数字 key，收集器是永远不会删除它的。当然，如果一个数字 key 所对应的 value 被回收了，那么整个条目都会从这个弱引用 table 中删除。

字符串在此则显得有些特殊。虽然从实现的角度看，字符串是可回收的。但在有些方面，字符串却与其他可回收的对象不同。其他对象，例如 table 和函数都是显式创建的。又如，当 Lua 对表达式{}求值时，它就会创建一个新的 table。同样地，求值 function() ... end 时就会创建一个新函数[①]。然而，当 Lua 对"a".."b"求值时，它会创建一个新字符串吗？如果当前系统中已有了一个字符串"ab"，它会复用吗？还是创建一个新的字符串？编译器会在运行程序前先创建这个字符串吗？这些都无关紧要，它们都是实现的细节。从程序员的角度看，字符串就是值，而非对象。因此，字符串就像数字和布尔一样，不会从弱引用 table 中删除[②]。

17.1 备忘录（memoize）函数

一项通用的编程技术是“用空间换时间”。例如有一种做法就可以提高一些函数的运行速度，记录下函数计算的结果，然后当使用同样的参数再次调用该函数时，便可以复用之前的结果了。

① 实际上是一个 closure。
② 除非它关联的 value 被回收了。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

假设有一个普通的服务器，在它收到的请求中包含 Lua 代码。每当服务器收到一个请求，它就要对代码字符串调用 loadstring，然后再调用编译好的函数。不过，loadstring 是一个昂贵的函数，而有些发给服务器的命令具有很高的频率，例如"closeconnection()"。与其每次收到一条常见命令就调用 loadstring，还不如让服务器用一个辅助的 table 记录下所有调用 loadstring 的结果。因此，在每次调用 loadstring 前，服务器先检查 table 中是否已记录了代码字符串编译后的结果。如果没有，才调用 loadstring，并将结果存储到 table 中。可以将这个行为写成一个新函数：

```
local results = {}
function mem_loadstring (s)
  local res = results[s]
  if res == nil then                    -- 是否已记录过?
    res = assert(loadstring(s))         -- 计算新结果
    results[s] = res                    -- 存下以备之后复用
  end
  return res
end
```

这项优化节省的时间非常可观。但，它也可能导致不易察觉的开销。虽然有些命令会重复出现，但还有许多命令只发生一次。例如，table results 会逐渐地累积服务器收到的所有命令及其编译结果。经过一定的时间后，这种累积会耗费服务器的内存。弱引用的 table 正好可以解决这个问题，如果 results table 具有弱引用的 value，那么每次垃圾收集都会删除所有在执行时未使用的编译结果。

```
local results = {}
setmetatable(results, {--mode = "v"})  -- 使value成为弱引用
function mem_loadstring (s)
  <如前>
```

实际上，由于 key 总是字符串，则可以使这个 table 变成完全弱引用。若这么做：

```
setmetatable(results, {--mode = "kv"})
```

则最终效果完全一样。

“备忘录”技术还可以用于确保某类对象的唯一性。假设一个系统用 table 来表示颜色，其中 3 个字段 red、green 和 blue 都具有相同的取值范围。最简单的颜色生成函数是：

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

通过备忘录技术，可以复用具有相同颜色的 table。备忘录 table 的 key 可以根据颜色分量来生成，本例中是将颜色分量以分隔符连接起来：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local results = {}
setmetatable(results, {--mode = "v"})  -- 使value成为弱引用
function createRGB (r, g, b)
  local key = r .. "-" .. g .. "-" .. b
  local color = results[key]
  if color == nil then
    color = {red = r, green = g, blue = b}
    results[key] = color
  end
return color
end
```

这种实现可以使用户通过原始的相等性操作符比较两种颜色。若两种同时存在的颜色相等，那么它们必定是由同一个 table 表示的。不过，相同的颜色也可能在不同时间由不同的 table 表示，这是因为期间执行过垃圾收集，清除了 results table。只要一种颜色正在使用，就不会被清除出 results。因此，只要一个颜色未被清除，它就可与新颜色进行比较，它的表示也可作为后续调用来复用。

17.2 对象属性

关于弱引用 table，还有一项重要的应用是将属性与对象关联起来。有很多情况需要把有些属性绑定到某个对象，例如函数与其名称、table 的默认值及数组的大小等。

当对象是一个 table 时，可以通过适当的 key 将属性存储在这个 table 中。正如先前所看到的，创建唯一性 key 的最简单办法是创建一个新对象（通常是一个 table）。不过，若对象不是一个 table，它就无法保存属性了。另外，即使是 table，有时也不想将属性存储在原 table 中。例如，想保持属性的私有性，或者不想让属性扰乱 table 的遍历就需要用其他办法来关联属性与对象了。显然，使用外部 table 来关联它们是一种理想的做法[1]。可以将对象作为 key，对象的属性作为 value。这个外部 table 可以保存任意对象的属性，Lua 也允许将任何对象作为 table 的 key。另外，存储在外部对象中的属性不会干扰其他对象，只要 table 本身是私有的，这些属性也会是私有的。

然而，这个看似完美的做法却有一个重大缺陷。当用户将一个对象作为外部 table 的 key 时，就是引用了它。Lua 是无法回收一个作为 table key 的对象。如果用这个外部 table 来关联函数和函数名，那么这些函数就永远无法回收。用户可以使用弱引用 table 来解决这个问题。而本例需要的是弱引用 key。当一个弱引用 key 没有其他引用时，Lua 就可以回收它。注意，这个 table 不能使用弱引用 value，否则“存留的”对象的属性就有可能被回收。

① 可见，有时将 table 称为“关联数组”也正是由于这个原因。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

17.3 回顾 table 的默认值

13.4.3 节讨论了如何实现具有非 nil 默认值的 table。在 13.1.3 节中注明了还有两种技术需要弱引用 table 的支持。这里将介绍两种用于默认值的技术，它们其实是上述备忘录和对象属性的特殊应用。

第一种做法是使用一个弱引用 table，通过它将每个 table 与其默认值关联起来：

```
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 没有弱引用 key，它就会使所有具有默认值的 table 持久存在下去。

第二种做法是对每种不同的默认值使用不同的元表。不过，只要有重复的默认值，就复用同样的元表。这是备忘录的典型应用：

```
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     -- 备忘录
  end
  setmetatable(t, mt)
end
```

这里用到了弱引用 value，这样当 metas 中的元表在不使用时就可以被回收了。

这两种默认值的实现，哪种更好呢？一般而言，它们具有类似的复杂度和性能表现。第一种做法需要为每个 table 的默认值（defaults 中的一个条目）使用内存。而第二种做法则需要为每种不同的默认值使用一组内存（一个新 table、一个新 closure 和 metas 中的一个条目）。因此，如果程序中有上千个 table 和一些默认值，第二种做法则是首选。但如果只有很少的 table 共享几个公用的默认值，那么就应该选择第一种做法。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第3部分

Second Edition

Programming in Lua

- 第 18 章　数学库
- 第 19 章　table 库
- 第 20 章　字符串库
- 第 21 章　I/O 库
- 第 22 章　操作系统库
- 第 23 章　调试库

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第18章 数 学 库

从本章开始将介绍标准程序库，这些章节中并不会给出每个函数的完整说明，而只说明标准库中提供了哪些功能。为了说明得更清楚，讲解过程中会回避某些微妙的选项和行为。用户可以在 Lua 参考手册中学习更多的知识。

math（数学）库由一组标准的数学函数构成，包括三角函数（sin、cos、tan、asin、acos 等）、指数和对数函数（exp、log、log10）、取整函数（floor、ceil）、max 和 min、生成伪随机数的函数（random、randomseed），以及变量 pi 和 huge。其中 huge 为 Lua 可以表示的最大数字[①]。

所有的三角函数都使用弧度单位，可以用函数 deg 和 rad 来转换角度和弧度。如果使用角度单位，可以像这样重新定义三角函数：

```
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
...
```

函数 math.random 用于生成伪随机数，可以用 3 种方式来调用它。如果在调用它时不提供任何参数，它将返回一个在区间[0,1)内均匀分布的伪随机实数。如果提供了一个整数 *n* 作为参数，它将返回一个在区间[1,*n*]内的伪随机整数。例如，random(6)就可以用来模拟一次掷骰子的结果。最后一种方式是在调用它时提供两个整数参数 *m* 和 *n*，这样会得到一个在区间[m,n]内的伪随机整数。

函数 randomseed 用于设置伪随机数生成器的种子数，它的唯一参数就是这个种子数。通常在一个程序启动时，用一个固定的种子数来调用它，以此初始化伪随机数生成器。这样每次程序运行时，都会生成相同的伪随机数序列。从调试的角度看，这是一个不错的特性。对于一个游戏来说，则每次都会得到相同的情景。对此通常的解决方法是使用当前时间作为种子数：

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

函数 os.time 返回一个表示当前时间的数字，一般这个数字表示从某个时间点开始至今的秒数。

---

① 在某些平台上，huge 可能是一个特殊的 inf 值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

math.random 函数使用了 C 标准库中的 rand 函数，在某些 C 标准库的实现中，这个函数所产生的数字并不具备统计意义上的均匀分布特性。Lua 的某些独立发行的版本中包含了更好的伪随机数生成器。[①]

① 为了避免版权问题，Lua 的标准发行版中没有包含这类生成器。标准版中只包含了 Lua 作者编写的代码。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 19 章　table 库

table 库是由一些辅助函数构成的，这些函数将 table 作为数组来操作。其中，有用于在列表中插入和删除元素的函数，有对数组元素进行排序的函数，还有连接一个数组中所有字符串的函数。

19.1　插入和删除

函数 table.insert 用于将一个元素插入到一个数组的指定位置，它会移动后续元素以空出空间。例如，如果 t 是数组{10, 20, 30}，当调用 table.insert(t, 1, 15)后，t 会变为{15, 10, 20, 30}。但有一种特殊情况，如果在调用 insert 时没有指定位置参数，则会将元素添加到数组末尾[①]。下面这段代码逐行地读取了程序的输入，并将所有的行保存在一个数组中：[②]

```
t = {}
for line in io.lines() do
  table.insert(t, line)
end
print(#t)          --> (读入的行数)
```

函数 table.remove 会删除（并返回）数组指定位置上的元素，并将该位置之后的所有元素前移，以填补空隙。如果在调用这个函数时不指定位置参数，它就会删除数组的最后一个元素。

有了这两个函数，就可以很容易地实现栈、队列和双向队列。可以用 t={}来初始化这种结构，table.insert(t, x)等价于压入操作，table.remove(t)等价于弹出操作。table.insert(t, 1, x)会在结构的另一端（也就是起始处）插入一个新元素，而 table.remove(t, 1)会从这一端删除元素。后两个操作不是很高效，因为它们必须移动元素。不过，由于 table 库中的函数都是用 C 语言实现的，所以这些循环的花销并不高，对于较小的数组[③]来说使用这种实现较好。

19.2　排　　序

另一个有用的数组函数是 table.sort，我们在此之前已经用到过它。它可以对一个数组进行排序，还可以指定一个可选的次序函数。这个次序函数有两个参数，如果希望第一个参数在排

① 因此也无须移动任何元素。
② 在 Lua 5.0 中，示例中的添加做法是很常见的。而在 Lua 5.1 中，建议使用 t[#t+1] = line 来给一个列表添加元素。
③ 最多几百个元素的数组。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

序结果中位于第二个参数值前，就应当返回 true。如果没有提供这个函数，sort 就使用默认的小于操作[1]。

一个常见的错误是试图对一个 table 的索引进行排序。在 table 中，索引是一个无序的集合。如果对它们进行排序，则必须将它们复制到一个数组中，然后对这个数组进行排序。下面演示一个示例，假设要读取一个源文件，并构建一个 table 记录每个函数并定义它的行号：

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

现在要求按字母次序打印这些函数名。如果使用 pairs 来遍历这个 table，就会发现遍历所得的名称是无序的。由于这些名称是 table 的 key，因此不能对它们进行直接排序。但是，可以将它们放到一个数组中，然后对这个数组进行排序。正确的做法是先用这些名称创建一个数组，然后对数组排序，最后打印结果：

```
a = {}
for n in pairs(lines) do a[#a + 1] = n end
table.sort(a)
for i,n in ipairs(a) do print(n) end
```

注意，对于 Lua 来说，数组也是无序的，它们本质上是 table。然而由于用户知道如何计算索引，因此在访问数组时，只要使用有序的索引，就可以顺序地访问数组。这就是为什么必须用 ipairs 而不是 pairs 来遍历数组的原因。前者使 key 为 1、2、……的顺序，而后者采用 table 的原始顺序。

另外还有一个更高级的方法，就是写一个迭代器，使它根据 table key 的次序来进行遍历。同时，还有一个可选参数 f，用于指定某种特殊次序。以下函数先将 key 排序到一个数组中，然后迭代这个数组，且每步都返回原 table 中的 key 和 value。

```
function pairsByKeys (t, f)
  local a = {}
  for n in pairs(t) do a[#a + 1] = n end
  table.sort(a, f)
  local i = 0            -- 迭代器变量
  return function ()     -- 迭代器函数
    i = i + 1
    return a[i], t[a[i]]
  end
end
```

① 对应于操作符“<”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

通过这个函数就可以很容易地按字母次序来打印那些函数名了：

```
for name, line in pairsByKeys(lines) do
  print(name, line)
end
```

19.3 连 接

在 11.6 节中已经看到过 table.concat。它接受一个字符串数组，并返回这些字符串连接后的结果。它有一个可选参数，用于指定插到字符串之间的分隔符。这个函数另外还接受两个可选参数，用于指定第一个和最后一个要连接的字符串索引。

下面这个函数是 table.concat 的一个扩展，它能处理嵌套的字符串数组：

```
function rconcat (l)
  if type(l) ~= "table" then return l end
  local res = {}
  for i=1, #l do
    res[i] = rconcat(l[i])
  end
  return table.concat(res)
end
```

对于数组中的每个元素，rconcat 都递归地调用自己，以此来连接所有可能嵌套的字符串数组。最后，它调用 table.concat 来连接这些结果部分。

```
print(rconcat{{"a", {" nice"}}, " and", {{" long"}, {" list"}}})
  --> a nice and long list
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 20 章　字 符 串 库

原始的 Lua 解释器操作字符串的能力是很有限的。一个程序只能创建字符串常量、连接字符串及获取字符串的长度。它无法提取子串或者检索字符串的内容。在 Lua 中真正的字符串操作能力来源于字符串库。

字符串库中的所有函数都导出在模块string中。在Lua 5.1中，它还将这些函数导出作为string类型的方法[①]。这样，假设要将一个字符串转换到大写形式，可以写 string.upper(s)，也可以写 s:upper()。但是为了避免与 Lua 5.0 不兼容，在本书的大多数示例中将采用基于模块的写法。

20.1　基础字符串函数

字符串库中有一些函数非常简单。函数string.len(s)可返回字符串 s 的长度。函数string.rep(s, n)（或 s:rep(n)）可返回字符串 s 重复 *n* 次的结果。例如，可以用 string.rep("a", 2^20)来创建一个 1MB 的字符串。函数 string.lower(s)可返回一份 s 的副本，其中所有的大写字母都被转换成小写形式，而其他字符则保持不变。string.upper 与之相反，它将小写转换成大写。大小写转换函数有一个典型用途，假设要对一个字符串数组进行排序，并且不区分大小写，可以这样写：

```
table.sort(a, function (a, b)
  return string.lower(a) < string.lower(b)
end)
```

函数 string.upper 和 string.lower 都遵循当前的区域设置（locale）。因此，如果在 European Latin-1 区域工作，那么表达式 string.upper("ação")的结果就是"AÇÃO"。

函数 string.sub(s, i, j)可以从字符串 s 中提取第 i 个到第 j 个字符。在 Lua 中，字符串的第一个字符的索引是 1。还可以用负数索引，这样会从字符串的尾部开始计数，索引-1 代表字符串的最后一个字符，-2 代表倒数第二个字符，依此类推。这样，调用函数 string.sub(s, 1, j)或 s:sub(1, j)，就可以得到字符串 s 中长度为 j 的前缀。调用 string.sub(s, j, -1)或 s:sub(j)[②]，可以得到字符串中从第 j 个字符开始的一个后缀。调用 string.sub(s, 2, -2)可以返回去掉字符串 s 的第一个和最后一个字符后的复制。

```
s = "[in brackets]"
```

① 通过该类型的元表实现的。
② 第二个参数默认为-1。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


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

记住，Lua 中的字符串是不可变的。和 Lua 中的所有其他函数一样，string.sub 不会改变字符串的值，它只会返回一个新字符串。一种常见的错误是这么写：

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

这里假定 s 的值就这样被改变了。如果要改变一个变量的值，就必须赋予它一个新的值：

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

函数 string.char 和 string.byte 用于转换字符及其内部数值表示。string.char 函数接受零个或多个整数，并将每个整数转换成对应的字符，然后返回一个由这些字符连接而成的字符串。string.byte(s, i)返回字符串 s 中第 i 个字符的内部数值表示，它的第二个参数是可选的，调用 string.byte(s)可返回字符串 s 中第一个字符的内部数值表示。在下例中，假定字符是用 ASCII 表示的：

```
print(string.char(97))                    --> a
i = 99; print(string.char(i, i+1, i+2))   -->cde
print(string.byte("abc"))                 --> 97
print(string.byte("abc", 2))              --> 98
print(string.byte("abc", -1))             --> 99
```

最后一行用了一个负数索引来访问字符串的最后一个字符。

在 Lua 5.1 中，string.byte 还可以接受可选的第三个参数。调用 string.byte(s, i, j)可以返回索引 i 到 j 之间（包括 i 和 j）的所有字符的内部表示值。

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

j 的默认值是 i，因此在调用该函数时若不指定这个参数，那么就只返回第 i 个字符的值，这就与 Lua 5.0 一样了。还有一种习惯写法是{s:byte(1, -1)}，这种写法会创建一个 table，其中包含了 s 中所有字符的编码。有了这个 table，就可以调用 string.char(unpack(t))来重建原字符串。但是，由于 Lua 限制了一个函数的返回值数量，因此这项技术无法应用于较长的字符串[①]。

函数 string.format 是用于格式化字符串的利器，经常用在输出上。它会根据第一个参数的描述，返回后续其他参数的格式化版本，这第一个参数也称为“格式化字符串”。编写格式化字符串的规则，与标准 C 语言中 printf 等函数的规则基本相同：它由常规文本和指示（directive）组成，这些指示控制了每个参数应放到格式化结果的什么位置，及如何放入它们。一个指示由字符'%'加上一个字母组成，这些字母指定了如何格式化参数，例如'd'用于十进制数、'x'用于十六进制数、'o'用于八进制数、'f'用于浮点数和's'用于字符串等。在字符'%'和字母之间可以再指定一些其他选项，用于控制格式的细节。例如，指定一个浮点数中有几个十进制数字。

① 不用于大于 2K 字节的字符串。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(string.format("pi = %.4f", math.pi))          --> pi = 3.1416
d = 5; m = 11; y = 1990
print(string.format("%02d/%02d/%04d", d, m, y))     --> 05/11/1990
tag, title = "h1", "a title"
print(string.format("<%s>%s</%s>", tag, title, tag))
  --> <h1>a title</h1>
```

在第一次打印中，%.4f 表示一个浮点数的小数点后有 4 个数字。在第二次打印中，%02d 表示一个十进制数至少有两个数字，如不足两个数字，则用 0 补足；而指示%2d 则表示用空格来补足。关于这些指示的完整描述可以参阅 Lua 参考手册。或者，查阅 C 语言手册，因为 Lua 是通过调用 C 标准库来完成实际的工作。

20.2　模式匹配（pattern-matching）函数

字符串库中最强大的函数是 find、match、gsub（global substitution，全局替换）和 gmatch（global match，全局匹配），它们都是基于“模式（pattern）”的。

不同于其他脚本语言，Lua 既没有使用 POSIX（regexp），也没有使用 Perl 正则表达式来进行模式匹配。其原因主要是考虑到 Lua 的大小。一个典型的 POSIX 正则表达式实现需要超过 4000 行代码，这相当于所有 Lua 标准库加在一起的大小。而相比之下，Lua 采用的模式匹配实现的代码只有 500 行不到。当然，Lua 的模式匹配所能达到的功能不及完整的 POSIX 实现。但是，Lua 的模式匹配仍是一个强大的工具，并且它还具有一些特性，能在进行某些匹配时，比标准 POSIX 实现更为方便。

20.2.1　string.find 函数

string.find 函数用于在一个给定的目标字符串中搜索一个模式。最简单的模式就是一个单词，它只会匹配与自己完全相同的拷贝。例如，模式"hello"会搜索目标字符串中的子串"hello"。当 find 找到一个模式后，它会返回两个值：匹配到的起始索引和结尾索引。如果没有找到任何匹配，它就返回 nil。

```
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
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

如果匹配成功，就可以用 string.find 的返回值来调用 string.sub，以此提取出目标字符串中匹配于该模式的那部分子串[1]。

string.find 函数还具有一个可选的第三个参数，它是一个索引，告诉函数应从目标字符串的哪个位置开始搜索。当处理所有与给定模式相匹配的部分时，这个参数就很有用。可以重复搜索新的匹配，且每次搜索都从上一次找到的位置开始。下面这个示例用字符串中所有换行符的位置创建了一个 table：

```
local t = {}                          -- 存储索引的table
local i = 0
while true do
  i = string.find(s, "\n", i+1)       -- 查找下一个换行符
  if i == nil then break end
  t[#t + 1] = i
end
```

接下来将介绍一种更简单的方法来编写这个循环，其中用到了 string.gmatch 迭代器。

20.2.2 string.match 函数

从某种意义上说，函数 string.match 与 string.find 非常相似，它也是用于在一个字符串中搜索一种模式。不同之处在于，string.match 返回的是目标字符串中与模式相匹配的那部分子串，而非该模式所在的位置。

```
print(string.match("hello world", "hello"))   --> hello
```

对于固定的模式，例如"hello"，使用这个函数就没有什么意义了。但当使用变量模式（Variable Pattern）时，它的特性就显现出来了，如下示例：

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
print(d)  --> 17/7/1990
```

在后面将会讨论到模式"%d+/%d+/%d+"的含义及 string.match 的高级用法。

20.2.3 string.gsub 函数

string.gsub 有 3 个参数：目标字符串、模式和替换字符串。它的基本用法是将目标字符串中所有出现模式的地方替换为替换字符串（最后一个参数）：

```
s = string.gsub("Lua is cute", "cute", "great")
```

① 对于像单词这样的简单模式，取出的就是模式自身。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(s)            --> Lua is great
s = string.gsub("all lii", "l", "x")
print(s)            --> axx xii
s = string.gsub("Lua is great", "Sol", "Sun")
print(s)            --> Lua is great
```

另外还有可选的第四个参数，可以限制替换的次数：

```
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 还有另一个结果，即实际替换的次数。例如，以下代码就是一种统计字符串中空格数量的简单方法：

```
count = select(2, string.gsub(str, " ", " "))
```

20.2.4　string.gmatch 函数

string.gmatch 会返回一个函数，通过这个函数可以遍历到一个字符串中所有出现指定模式的地方。例如，以下示例找出了给定字符串 s 中所有的单词：

```
words = {}
for w in string.gmatch(s, "%a+") do
  words[#words + 1] = w
end
```

其中模式"%a+"表示匹配一个或多个字母字符的序列（也就是单词）。在本例中，for 循环会遍历目标字符串中所有的单词，并且将它们储存到列表 words 中。

通过 gmatch 和 gsub 可以模拟出 Lua 中的 require 在寻找模块时所用的搜索策略，如下：

```
function search (modname, path)
  modname = string.gsub(modname, "%.", "/")
  for c in string.gmatch(path, "[^;]+") do
    local fname = string.gsub(c, "?", modname)
    local f = io.open(fname)
    if f then
      f:close()
      return fname
    end
  end
  return nil    -- 未找到
end
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

首先，用点符号来替换所有的目录分隔符，示例假设目录分隔符为'/'。接下去我们就会看到，在模式中“点”具有特殊的含义，因此若要表示一个点必须写为"%."。其次，函数遍历路径中的所有组成部分，这里所谓的组成部分是指那些不包含分号的最长子串。对于每个组成部分，都用模块名来替换其中的问号，以此获得最终的文件名。最后，检查该文件是否存在。如果存在，则关闭该文件，并返回它的名称。

20.3 模　　式

可以用字符分类（character class）创建更多有用的模式。字符分类就是模式中的一项，可以与一个特定集合中的任意字符相匹配。例如，分类%d 可匹配任意数字。如下例可以用模式"%d%d/%d%d/%d%d%d%d"搜索符合"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分类：

.	所有字符
%a	字母
%c	控制字符
%d	数字
%l	小写字母
%p	标点符号
%s	空白字符
%u	大写字母
%w	字母和数字字符
%x	十六进制数字
%z	内部表示为 0 的字符

这些分类的大写形式表示它们的补集，例如，"%A"表示所有非字母字符：

```
print(string.gsub("hello, up-down!", "%A", "."))
  --> hello..up.down. 4
```

注意，打印出的“4”不是结果字符串的一部分，而是 gsub 的第二个结果，即替换的次数。在后续打印 gsub 结果的示例里，将忽略这个计数结果。

在模式中还有一些字符被称为“魔法字符”，它们具有特殊的含义。这些魔法字符有：

```
( ) . % + - * ? [ ] ^ $
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

字符'%'作为这些魔法字符的转义符。因此"%."表示匹配一个点，"%%"表示匹配字符'%'本身。不仅可以将'%'用于魔法字符，还可以用于其他所有非字母和数字的字符。当不确定某个字符是否需要转移时，应该前置一个转义符。

对于 Lua 来说，模式就是普通的字符串，并像其他字符串一样遵循相同的规则。只有模式函数才会解释它们，此时才会将'%'当作转义符来处理。在模式中放入一个引号的方法，与在普通字符串中放入引号的方法相同。也就是说，需要用 Lua 的转义符'\'来对引号进行转义。

在一对方括号内将不同的字符分类或单个字符组合起来，即可创建出属于用户自己的字符分类，这种新的字符分类叫做“字符集（char-set）”。例如，字符集"[%w_]"表示同时匹配字母、数字和下画线；字符集"[01] "表示匹配二进制数字；字符集"[%[%]]"表示匹配方括号本身。若要统计一段文本中元音的数量，可以这么写：

```
nvow = select(2, string.gsub(text, "[AEIOUaeiou]", ""))
```

在字符集中包含一段字符范围的做法是写出字符范围的第一个字符和最后一个字符，并用横线连接它们。这个方法用的很少，因为大多数常用的字符范围都已预定义好了，例如"[0-9]"即为"%d"，"0-9a-fA-F]"则为"%x"。不过，如果需要查找一个八进制数字，那么可以写"[0-7]"，而不是"[01234567]"。在一个字符集前加一个'^'，就可以得到这个字符集的补集，像上例模式"[^0-7]"表示所有非八进制数字的字符，而模式"[^\n]"则表示除了换行符以外的其他字符。对于简单的分类，使用其大写形式也可以得到其补集，"%S"显然要比"[^%s]"简单。

字符分类使用与 Lua 相同的区域设置。因此，"[a-z]"可能并不等同于"%l"。在某些区域设置中，后者可能会包括像"ç"和"ã"这样的字母。一般情况下，选用后者，因为它更简单、更具移植性、也更高效。

还可以通过修饰符来描述模式中的重复部分和可选部分。Lua 的模式提供 4 种修饰符：

+	重复 1 次或多次
*	重复 0 次或多次
-	也是重复 0 次或多次
?	可选部分（出现 0 或 1 次）

"+"修饰符可匹配属于字符分类的一个或多个字符。它总是获取与模式相匹配的最长序列，例如，模式"%a+"表示一个或多个字母，即单词：

```
print(string.gsub("one, and two; and three", "%a+", "word"))
  --> word, word word; word word
```

模式"%d+"匹配一个或多个数字（一个整数）：

```
print(string.match("the number 1298 is even", "%d+"))   --> 1298
```

修饰符"*"类似于"+"，但它还接受出现 0 次的情况。一种典型的用途是匹配一个模式不同

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

部分之间的空格。例如，匹配像"()"或"( )"这样的一对空圆括号，可以使用模式"%(%s*%)"。其中"%s*"可匹配0个或多个空格[1]。另一个示例是用模式"[_%a][_%w]*"匹配Lua程序中的标识符，标识符是一个由字母或下画线开始，并伴随0个或多个下画线、字母或数字的序列。

修饰符'-'和'*'一样，也是用于匹配属于字符分类的0个和多个字符的。不过，它会匹配最短的子串。虽然有时候"*"和"-"没什么差别，但通常它们所表现出来的结果却是截然不同的。例如，当试图用模式"[_%a][_%w]-"来查找一个标识符时，只会找到第一个字母，因为"[_%w]-"总是匹配空串。又如，假设要查找一个C程序中的注释，通常会首先尝试"/%*.*%*/"[2]，然而由于".*"会尽可能地扩展，因此程序中的第一个"/*"只会与最后一个"*/"相匹配：

```
test = "int x; /* x */  int y; /* y */"
print(string.gsub(test, "/%*.*%*/", "<COMMENT>"))
  --> int x; <COMMENT>
```

若使用".-"模式，则会以尽可能少的扩展来找到第一个"*/"。这样就能得到想要的结果了：

```
test = "int x; /* x */  int y; /* y */"
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
  --> int x; <COMMENT>  int y; <COMMENT>
```

最后一个修饰符"?"可用于匹配一个可选的字符。例如，要在一段文本中寻找一个整数，而这个整数可以包括一个可选的正负号。那么使用模式"[+-]?%d+"就可以完成这项任务，它可以匹配像"-12"、"23"、"+1009"这样的数字，而"[+-]"是一个匹配'+'和'-'号的字符分类，后面的'?'说明这个符号是可选的。

与其他系统不同的是，Lua中的修饰符只能应用于一个字符分类，无法对一组分类进行修饰。例如，无法写出匹配一个可选单词的模式[3]。在本章的最后会介绍一些高级技术，可以使用它们来绕开这条限制。

如果模式以一个'^'起始，那么它只会匹配目标字符串的开头部分。类似地，如果模式以'$'结尾，那么它只会匹配目标字符串的结尾部分。这些标记可用于限定一个模式的查找。例如，下面这行测试：

```
if string.find(s, "^%d") then ...
```

可检查字符串s是否以一个数字开头，而以下测试：

```
if string.find(s, "^[+-]?%d+$") then ...
```

则可检查这个字符串是否表示一个整数，并且没有多余的前导字符和结尾字符。

在模式中，还可以使用"%b"，用于匹配成对的字符。它的写法是"%b<x><y>"，其中<x>

① 在模式中，圆括号也有特定的含义，所以必须前置转义符'%'。
② 此处用了适当的转义，表示以"/*"开头，伴随任何字符序列，并以"*/"结尾。
③ 除非该单词只有一个字母。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

和<y>是两个不同的字符，<x>作为一个起始字符，<y>作为一个结束字符。例如，模式"%b()"可匹配以'('开始，并以')'结束的子串：

```
s = "a (enclosed (in) parentheses) line"
print(string.gsub(s, "%b()", ""))    --> a  line
```

这种模式的典型用法包括"%b()"、"%b[]"、"%b{}"和"%b<>"，但实际上可以用任何字符作为分隔符。

20.4 捕获（capture）

捕获功能可根据一个模式从目标字符串中抽出匹配于该模式的内容。在指定捕获时，应将模式中需要捕获的部分写到一对圆括号内。

对于具有捕获的模式，函数 string.match 会将所有捕获到的值作为单独的结果返回。即它会将目标字符串切成多个捕获到的部分[①]：

```
pair = "name = Anna"
key, value = string.match(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)  --> name  Anna
```

模式"%a+"表示一个非空的字母序列，模式"%s*"表示一个可能为空的空格序列。因此上例中的这个模式表示一个字母序列，伴随着一个空格序列，一个'='，一些空格，以及另一个字母序列。模式中表示两个字母序列的部分都放在一对圆括号中，因此如果发现匹配，就能捕获到它们。下面是一个类似的示例：

```
date = "Today is 17/7/1990"
d, m, y = string.match(date, "(%d+)/(%d+)/(%d+)")
print(d, m, y)  --> 17  7  1990
```

还可以对模式本身使用捕获。在一个模式中，可以有"%d"这样的项，其中 *d* 是一个只有一位的数字，该项表示只匹配与第 *d* 个捕获相同的内容。有一个典型的实例可以说明它的作用，假设要在一个字符串中寻找一个由单引号或双引号括起来的子串。那么可以用这样的模式"["']-["']"，它表示一个引号后面是任意内容及另外一个引号。但是，这种模式在处理像"it's all right"这样的字符串时就出现问题。要解决这个问题，可以捕获第一个引号，然后用它来指定第二个引号：

```
s = [[then he said: "it's all right"!]]
q, quotedPart = string.match(s, "([\"'])(.-)%1")
print(quotedPart)   --> it's all right
```

① 在 Lua 5.0 中，这个任务是由 string.find 完成的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(q)            --> "
```

第一个捕获是引号字符本身，第二个捕获是引号中的内容，即与".-"相匹配的子串。

又如，匹配 Lua 中的长字符串：

```
%[(=*)%[(.-)%]%1%]
```

它匹配的内容依次是：一个左方括号、零个或多个等号、另一个左方括号、任意内容（即字符串的内容）、一个右方括号、相同数量的等号及另一个右方括号：

```
p = "%[(=*)%[(.-)%]%1%]"
s = "a = [=[[[ something ]] ]==] ]=]; print(a)"
print(string.match(s, p))    --> =      [[ something ]] ]==]
```

第一个捕获是等号序列，本例的等号序列中只有一个等号。第二个捕获是字符串的内容。

对于捕获到的值，还可用于 gsub 函数的字符串替换。和模式一样，用于替换的字符串中也可以包含"*%d*"这样的项。当进行替换时，这些项就对应于捕获到的内容。"%0"表示整个匹配，并且替换字符串中的"%"必须被转义为"%%"。下面这个示例会重复字符串中的每个字符，并且在每个副本之间插入一个减号：

```
print(string.gsub("hello Lua!", "%a", "%0-%0"))
  --> h-he-el-ll-lo-o L-Lu-ua-a!
```

下例交换了所有相邻的字符：

```
print(string.gsub("hello Lua", "(.)(.)", "%2%1"))  --> ehll ouLa
```

以下是一个更有用的示例，写一个简单的格式转换器，它能读取用 LaTeX 风格书写的命令字符串，例如：

```
\command{some text}
```

并将它转换成 XML 风格的格式：

```
<command>some text</command>
```

如果不考虑嵌套的命令[①]，那么下面这行代码即可完成这项工作：

```
s = string.gsub(s, "\\(%a+){(.-)}", "<%1>%2</%1>")
```

例如，如果 s 是一个字符串，其内容为：

```
the \quote{task} is to \em{change} that.
```

① 在下一节中将会看到如何处理嵌套的命令。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

调用 gsub 后，s 会变成：

```
the <quote>task</quote> is to <em>change</em> that.
```

最后，还有一个剔除字符串两端空格的示例：

```
function trim (s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end
```

注意，模式中某些部分的作用，两个定位标记（'^'和'$'）表示在操作整个字符串；而".-"会试图匹配尽可能少的内容，所以首尾两处的"%s*"便可匹配到两端所有的空格。其次，gsub 会返回两个值，并用一对额外的括号来丢弃多余的结果，即丢弃“匹配的总数”。

20.5 替　换

string.gsub 函数的第三个参数不仅是一个字符串，还可以是一个函数或 table。当用一个函数来调用时，string.gsub 会在每次找到匹配时调用该函数，调用时的参数就是捕获到的内容，而该函数的返回值则作为要替换的字符串。当用一个 table 来调用时，string.gsub 会用每次捕获到的内容作为 key，在 table 中进行查找，并将对应的 value 作为要替换的字符串。如果 table 中不包含这个 key，那么 string.gsub 不改变这个匹配。

例如，以下函数将完成一次变量展开。它对字符串中所有格式为$varname 的部分，替换为对应全局变量 varname 的值：

```
function expand (s)
  return (string.gsub(s, "$(%w+)", _G))
end

name = "Lua"; status = "great"
print(expand("$name is $status, isn't it?"))
  --> Lua is great, isn't it?
```

对每处与"$(%w+)"相匹配的地方[1]，string.gsub 都会在 table _G 中查找捕获到的名称，并用找到的名称替换字符串中的匹配部分。如果 table 中没有这个 key，则不进行替换。

```
print(expand("$othername is $status, isn't it?"))
  --> $othername is great, isn't it?
```

如果不确定所有的变量都有一个对应的字符串值，则可以对它们的值应用 tostring。在这种情况下，可以用一个函数来提供要替换的值：

① 即一个美元符号跟随着一个名称。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
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
```

现在，对于所有匹配"$(%w+)"的地方，string.gsub 都会调用给定的函数，并传入捕获到的名称。如果函数返回 nil，则不作替换。在本例中不会出现这种情况，因为 tostring 不会返回 nil。

最后一个示例则继续回到上一节中提及的格式转换器。本例仍然是将 LaTeX 风格的命令（\example{text}）转换成 XML 风格（<example>text</example>），但是允许嵌套的命令。以下函数用递归的方式完成了这项任务：

```
function toxml (s)
  s = string.gsub(s, "\\(%a+)(%b{})", function (tag, body)
        body = string.sub(body, 2, -2)  -- 删除花括号
        body = toxml(body)              -- 处理嵌套的命令
        return string.format("<%s>%s</%s>", tag, body, tag)
      end)
  return s
end

print(toxml("\\title{The \\bold{big} example}"))
  --> <title>The <bold>big</bold> example</title>
```

20.5.1 URL 编码

下一个示例是关于 URL 编码，这是 HTTP 所使用的一种编码方式，用于在一个 URL 中传送各种参数。这种编码方式会将特殊字符（如'='、'&'、'+'）编码为"%<xx>"的形式，其中<xx>是字符的十六进制表示。此外，它还会将空格转换为"+"。例如，它会将字符串"a+b = c"编码为"a%2Bb+%3D+c"。最后，它会将每对参数名及其值用"="连接起来，并将每对结果 name=value 用"&"连接起来。例如，对于值：

```
name = "al";  query = "a+b = c"; q="yes or no"
```

会被编码为：

```
"name=al&query=a%2Bb+%3D+c&q=yes+or+no"
```

现在，假设要对这个 URL 进行解码。要求对编码中的每个值，以其名称作为 key，保存到一个 table 内。以下函数完成一次基本的解码：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
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
```

第一条语句将字符串中的所有的"+"改为空格，第二句 gsub 则匹配所有以"%"为前缀的两位十六位数字，并对每处匹配调用一个匿名函数。这个函数会将十六进制数转换成一个数字[①]，然后调用 string.char 返回相应的字符。如下所示：

```
print(unescape("a%2Bb+%3D+c"))  --> a+b = c
```

用 gmatch 来对 name=value 进行解码。由于名称和值都不能包含"&"和"="，所以可以用模式"[^&=]+"来匹配它们：

```
cgi = {}
function decode (s)
  for name, value in string.gmatch(s, "([^&=]+)=([^&=]+)") do
    name = unescape(name)
    value = unescape(value)
    cgi[name] = value
  end
end
```

调用 gmatch 会匹配所有格式为 name=value 的部分。对于每组参数，迭代器都会将捕获到的内容作为变量 name 和 value 的值。循环体内只是对两个字符串调用 unescape，然后将结果保存到 table cgi 中。

另一方面，编码函数也很容易编写。首先，写一个 escape 函数，它会将所有的特殊字符编码为"%"并伴随该字符的十六进制码[②]。另外，它还会将空格转换为"+"。

```
function escape (s)
  s = string.gsub(s, "[&=+%%%c]", function (c)
        return string.format("%%%02X", string.byte(c))
      end)
  s = string.gsub(s, " ", "+")
  return s
end
```

encode 函数会遍历整个待编码的 table，构建出最终的结果字符串：

① 以 16 为底的 tonumber。
② 格式化字符串为"%02X"，这样会创建出一个两位的十六进制数；若不足两位，则以 0 补充。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
function encode (t)
  local b = {}
  for k,v in pairs(t) do
    b[#b + 1] = (escape(k) .. "=" .. escape(v))
  end
  return table.concat(b, "&")
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
```

20.5.2 tab 扩展

在 Lua 中，像"()"这样的空白捕获具有特殊的含义。这个模式不是代表捕获空内容[①]，而是捕获它在目标字符串中的位置，返回的是一个数字：

```
print(string.match("hello", "()ll()"))  --> 3  5
```

注意，这个示例的结果与调用 string.find 得到的结果并不一样，因为第二个空捕获的位置是在匹配之后的。

这里有一个关于空捕获应用的示例，在一个字符串中扩展 tab（制表符）：

```
function expandTabs (s, tab)
  tab = tab or 8        -- 制表符的“大小”(默认为8)
  local corr = 0
  s = string.gsub(s, "()\t", function (p)
        local sp = tab - (p - 1 + corr)%tab
        corr = corr - 1 + sp
        return string.rep(" ", sp)
      end)
  return s
end
```

gsub 会匹配字符串中所有的 tab，捕获它们的位置。内嵌函数会根据每个 tab 的位置，计算出还需多少空格才能达到整数倍 tab 的列。它先对位置减一，使其从 0 开始计数，然后加上 corr 以补偿前面的 tab[②]。并且，它还会更新这个补偿值，用于下一个 tab 的修正，减一以去掉当前 tab，再加上要添加的空格数 sp。最后这个函数返回正确数量的空格。

为了完整起见，再看一下如何实现逆向的操作，即将空格转换为 tab。第一种方法可以考虑通过空捕获来对位置进行操作。还有一种更简单的方法即可以在字符串中每 8 个字符后插入一个标记。无论该标记是否位于空格前，都用 tab 替换它。

① 这实在是一个没什么用的操作。
② 每个 tab 的扩展都会影响到后续 tab 的位置。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
function unexpandTabs (s, tab)
  tab = tab or 8
  s = expandTabs(s)
  local pat = string.rep(".", tab)
  s = string.gsub(s, pat, "%0\1")
  s = string.gsub(s, " +\1", "\t")
  s = string.gsub(s, "\1", "")
  return s
end
```

这个函数首先对字符串中所有的 tab 进行了扩展。然后计算出一个辅助模式，用于匹配所有的 tab 字符序列，并通过这个模式在每个 tab 字符后添加一个标记（控制字符"\1"）。之后，它将所有以此标记结尾的空格序列都替换为 tab。最后，删除剩下的标记，即那些没有位于空格后的标记。

20.6 技　　巧

模式匹配是一种操作字符串的强大工具。通过很少的 string.gsub 调用就可以完成许多复杂的操作，但是应该谨慎使用。

模式匹配不能代替一个传统的分析器。对于某些要求并不严格的程序来说，可以在源代码中做一些有用的匹配操作，但很难构建出高质量的产品。例如，之前曾用一个模式来匹配 C 代码中的注释。如果 C 代码中有一个字符串常量含有"/*"，就会得到一个错误的结果：

```
test = [[char s[] = "a /* here";  /* a tricky string */]]
print(string.gsub(test, "/%*.-%*/", "<COMMENT>"))
  --> char s[] = "a <COMMENT>
```

含有注释标记的字符串并不常见，所以对于自用的程序而言，这个模式足以达到要求。但是，不应将这个有问题的程序发布出去。

通常，在 Lua 程序中使用模式匹配时非常高效。一台奔腾 333MHz 计算机可以在不到十分之一秒的时间内，匹配一个具有 20 万个字符的文本中所有的单词[①]，但需要注意的是，应该提供尽可能精确的模式，宽泛的模式会比精确的模式慢许多。一个较极端的示例就是模式"(.-)%$"，它可以获取一个美元符号前所有的字符。如果目标字符串中有一个美元符号，那么一切都正常。但如果字符串中不存在美元符号，这个算法就会试着从字符串起始位置开始匹配此模式，为了搜索美元符号，它会遍历整个字符串。当到达字符串结尾时，这次模式匹配就会失败，但此失败仅意味着从字符串起始位置开始的匹配失败了。然后，这个算法还会从字符串的第二个位置开始第二次搜索，其结果仍是无法匹配这个模式。这样的匹配过程依此类推，从

① 大约有 3 万个单词。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

而表现为二次方的时间复杂度。但在一台奔腾 333MHz 的计算机上，搜索 20 万个字符需要执行超过 3 小时的时间。要解决这个问题，只需将模式限定为仅匹配字符串的起始部分，也就是将模式指定为"^(.-)%$"。这样便告诉了算法，如果不能在起始位置上找到匹配，就停止搜索。有了这种位置限定后，再运行该模式就只需要不到十分之一秒的时间了。

此外还要小心“空模式”的使用，也就是那些会匹配空字符串的模式。例如，如果准备用模式"%a*"来匹配名称，那么会发现到处都是名称：

```
i, j = string.find(";$%  **#$hello13", "%a*")
print(i,j)   --> 1  0
```

在这个示例中，string.find 调用会成功地在字符串起始处找到一个空的字母序列。

在模式的开始或结束处使用修饰符"-"是没有意义的，因此这样总会匹配到空字符串。此修饰符总是需要有些东西在它周围以限制它的扩展范围。同样，使用含有".*"的模式也需要注意，因为这种指示可能会扩展到预期的范围之外。

有时用 Lua 自身来构造一个模式也是很有用的。在前面的空格转换为 tab 的程序中，已用到了这个技巧。接下来再看另外一个示例，如何找出一个文本中较长的行，这里的“较长”是指超过 70 个字符的行。其实，一个长行就是一个具有 70 个或更多个字符的序列，其中每个字符都不为换行符。可以用字符分类"[^\n]"来匹配除换行符以外的其他单个字符。因此，可以将这个匹配单个字符的模式重复 70 次来匹配一个长行。在这里，可以用 string.rep 代替手写来创建这个模式：

```
pattern = string.rep("[^\n]", 70) .. "[^\n]*"
```

另外还有一个示例，假设要进行一次与大小写无关的查找。一种方法是将模式中的任何字母 x 用分类"[xX]"来替换，也就是一种同时包含原字母大小写形式的分类。可以用一个函数来自动地做这种转换：

```
function nocase (s)
  s = string.gsub(s, "%a", function (c)
        return "[" .. string.lower(c) .. string.upper(c) .. "]"
      end)
  return s
end

print(nocase("Hi there!"))   --> [hH][iI] [tT][hH][eE][rR][eE]!
```

有时要将所有出现 s1 的地方替换为 s2，而不管其中是否包含魔法字符。如果字符串 s1 和 s2 是字面常量，那么可以在编写字符串时对魔法字符使用适当的转义。但如果字符串是一个变量值，那么就需要用另一个 gsub 来做转义了：

```
s1 = string.gsub(s1, "(%W)", "%%%1")
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
s2 = string.gsub(s2, "%%", "%%%%")
```

在搜索字符串时，要对所有非字母和非数字的字符进行转义[1]，而在替换字符串时，只对"%"进行转义。

关于模式匹配还有一项有用的技术，就是在进行实际工作前，对目标字符串进行预处理。假设，要将一个字符串中位于一对双引号内的部分改为大写，但又允许其中包含转移的引号（"\""）：

```
follows a typical string: "This is \"great\"!".
```

处理这种情况的做法是对文本进行预处理，将所有可能导致歧义的部分编码为另一种形式。例如，可以将"\""编码为"\1"。不过，若原文中已含有"\1"，就会出错。一种可以避免此类问题的做法是将所有"\x"序列编码为"\ddd"形式，其中 ddd 是字符 x 的十六进制表示形式：

```
function code (s)
  return (string.gsub(s, "\\(.)", function (x)
           return string.format("\\%03d", string.byte(x))
         end))
end
```

现在，编码后的字符串中的"\ddd"序列都来源于编码，因为原字符串中所有的"\ddd"都已被 Lua 翻译成对应字符了。这样解码就很简单了：

```
function decode (s)
  return (string.gsub(s, "\\(%d%d%d)", function (d)
           return "\\" .. string.char(d)
         end))
end
```

现在便完成了这项任务。由于编码后的字符串中不包含任何转义的引号（"\""），就可以直接用".-*"来查找位于一对引号中的内容：

```
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\"!".
```

或者，更紧凑的写法：

```
print(decode(string.gsub(code(s), '".-"', string.upper)))
```

① 即大写的"W"。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 21 章　I/O 库

I/O 库为文件操作提供了两种不同的模型，简单模型（simple model）和完整模型（complete model）。简单模型假设有一个当前输入文件和一个当前输出文件，它的 I/O 操作均作用于这些文件。完整模型则使用显式的文件句柄。它采用了面向对象的风格，并将所有的操作定义为文件句柄上的方法。

在本书前面的章节示例中涉及到的简单操作都使用简单模型并且十分方便。但是对于更多的高级文件操作，例如同时读取多个文件，它就无法做到了。对于这些高级操作，需要使用完整模型。

21.1　简单 I/O 模型

简单模型的所有操作都作用于两个当前文件。I/O 库将当前输入文件初始化为进程标准输入（stdin），将当前输出文件初始化为进程标准输出（stdout）。在执行 io.read()操作时，就会从标准输入中读取一行。

用函数 io.input 和 io.output 可以改变这两个当前文件。io.input(filename)调用会以只读模式打开指定的文件，并将其设为当前输入文件。之后除非再次调用 io.input，否则所有的输入都将来源于这个文件。在输出方面，io.output 也可以完成类似的工作。如果出现错误，这两个函数都会引发（raise）错误。如果想直接处理这些错误，则必须使用完整模型中的 io.open。

通常 write 比 read 简单些，首先看一下 write。函数 io.write 接受任意数量的字符串参数，并将它们写入当前输出文件。它也可以接受数字参数，数字参数会根据常规的转换规则转换为字符串。如果想要完全地控制这种转换，则应该使用函数 string.format：

```
> io.write("sin (3) = ", math.sin(3), "\n")
  --> sin (3) = 0.14112000805987
> io.write(string.format("sin (3) = %.4f\n", math.sin(3)))
  --> sin (3) = 0.1411
```

在实际操作中应当避免写出 io.write(a..b..c)这样的代码，而是应该调用 io.write(a, b, c)，它能达到与 io.write(a..b..c)相同的效果，并且可以避免连接操作，因此效率更高。

无论使用 print 还是 io.write 都有一个原则。即在随意编写（quick-and-dirty）的程序中，或者为调试目的而编写的代码中，提倡使用 print；而在其他需要完全控制输出的地方使用

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

write。

```
> print("hello", "Lua"); print("Hi")
  --> hello   Lua
  --> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
  --> helloLuaHi
```

write 与 print 有几点不同。首先，write 在输出时不会添加像制表符或回车这样的额外字符。其次，write 使用当前输出文件，而 print 总是使用标准输出。最后，print 会自动调用其参数的 tostring()方法，因此它还能显示 table、函数和 nil。

函数 io.read 从当前输入文件中读取字符串，它的参数决定了要读取的数据：

"*all"	读取整个文件
"*line"	读取下一行
"*number"	读取一个数字
<num>	读取一个不超过<num>个字符的字符串。

调用 io.read("*all")会读取当前输入文件的所有内容，以当前位置作为开始。如果当前位置处于文件的末尾，或者文件为空，那么该调用会返回一个空字符串。

由于 Lua 可以高效地处理长字符串，因此在 Lua 中一种简单的、编写过滤器的技术就可以将整个文件读到一个字符串中，然后处理这个字符串（通常使用 gsub），最后把这个字符串写到输出：

```
t = io.read("*all")          -- 读取整个文件
t = string.gsub(t, ...)      -- 做相关的处理
io.write(t)                  -- 写输出
```

下面是一个完整的示例，这段代码使用 MIME quoted-printable 编码方式对文件内容进行编码。在这种编码方式中，非 ASCII 字符被编码为=<xx>的形式，其中<xx>是这个字符的十六进制数字代码。此外，为了保持编码的一致性，字符“=”也需要被编码：

```
t = io.read("*all")
t = string.gsub(t, "([\128-\255=])", function (c)
      return string.format("=%02X", string.byte(c))
    end)
io.write(t)
```

gsub 中使用的模式可以捕获所有编码为 128~255 的字符及等号字符。

调用 io.read("*line")会返回当前文件的下一行，但不包括换行字符。当到达文件末尾时，该调用会返回 nil，以表示无后续行可返回。它也是 read 的默认模式。通常，我只在需要逐行

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

处理的算法中使用这种模式。另外，建议使用*all 一次性读取整个文件，或者像后面介绍的按块来读取。

作为使用该模式的一个简单示例，下面这个程序将当前输入中的内容复制到当前输出中，并对每行进行编号：

```
for count = 1, math.huge do
  local line = io.read()
  if line == nil then break end
  io.write(string.format("%6d  ", count), line, "\n")
end
```

如果只为了迭代文件中的所有行，那么 io.lines 迭代器更为合适。例如，下面这个程序可以对文件中的所有行进行排序：

```
local lines = {}
-- 读取table 'lines'中所有行
for line in io.lines() do  lines[#lines + 1] = line  end
-- 排序
table.sort(lines)
-- 输出所有行
for _, l in ipairs(lines) do  io.write(l, "\n")  end
```

调用 io.read("*number")会从当前输入文件中读取一个数字。此时，read 会返回一个数字，而不是字符串。当一个程序需要从文件中读取大量数字时，应当避免生成中间的字符串过渡形式，这样可以提高程序的性能。*number 选项会忽略数字前面所有的空格，并且能处理像-3、+5.2、1000 及-3.4e-23 这样的数字格式。如果无法在当前文件位置读到一个数字[①]，read 会返回 nil。

在调用 read 时可以指定多个选项，函数会根据每个选项参数返回相应的内容。假设，有一个文件，其中每行有 3 个数字：

```
6.0      -3.23     15e12
4.3      234       1000001
...
```

现在要打印出每一行中最大的数字。可以用一次 read 函数调用来读取每行的 3 个数字：

```
while true do
  local n1, n2, n3 = io.read("*number", "*number", "*number")
  if not n1 then break end
  print(math.max(n1, n2, n3))
end
```

① 例如，由于格式不正确或者到达文件末尾。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

对于这类问题，还可以采用“*all”读取整个文件，然后再用 gmatch 来提取其中内容：

```
local pat = "(%S+)%s+(%S+)%s+(%S+)%s+"
for n1, n2, n3 in string.gmatch(io.read("*all"), pat) do
  print(math.max(tonumber(n1), tonumber(n2), tonumber(n3)))
end
```

除了以上这些基本的读取模式，还可以用一个数字 n 作为 read 的参数。此时，read 会试着从输入文件中读取 n 个字符。如果读不到任何字符[①]，它会返回 nil。否则会返回一个最多 n 个字符的字符串。下面这个示例演示了这种读取模式，它是一种将数据从 stdin 复制到 stdout 的高效方法：

```
while true do
  local block = io.read(2^13)          -- 缓冲大小为 8K
  if not block then break end
  io.write(block)
end
```

io.read(0)是一个特殊情况，它用于检查是否到达了文件末尾。如果还有数据可以读取，它会返回一个空字符串，否则返回 nil。

21.2 完整 I/O 模型

若要作更多的 I/O 控制，可以使用完整模型。这个模型是基于文件句柄的，它等价于 C 语言中的流（FILE *），表示一个具有当前位置的打开文件。

要打开一个文件，可以使用 io.open 函数。它仿照了 C 语言中的 fopen 函数，并且具有两个参数一个是要打开的文件名，另一个是模式（Mode）字符串。模式字符串可以是：“r”表示读取、“w”表示写入（同时会删除文件原来的内容）及“a”表示追加，另外还有一个可选的“b”表示打开二进制文件。open 函数会返回表示文件的新句柄。若发生错误，则返回 nil，及一条错误消息和一个错误代码。

```
print(io.open("non-existent-file", "r"))
  --> nil    non-existent-file: No such file or directory   2

print(io.open("/etc/passwd", "w"))
  --> nil    /etc/passwd: Permission denied  13
```

错误代码的解释依赖于系统。

一个错误检查的典型做法是：

① 例如，到底了文件末尾。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local f = assert(io.open(filename, mode))
```

如果打开失败，错误消息就会成为 assert 的第二个参数，然后 assert 会显示这个信息。

当打开一个文件后，就可以用 read/write 方法读写文件了。这与 read/write 函数相似，但是需要用冒号语法，将它们作为文件句柄的方法来调用。例如，要打开一个文件，并读取其所有内容，可以这么做：

```
local f = assert(io.open(filename, "r"))
local t = f:read("*all")
f:close()
```

I/O 库提供了 3 个预定义 C 语言流的句柄：io.stdin、io.stdout 和 io.stderr。这样，就可以将信息直接写到错误流：

```
io.stderr:write(message)
```

用户可以混合使用完整模式和简单模式。通过不指定参数调用 io.input()，可以得到当前输入文件的句柄。而通过 io.input(handle)，可以设置当前输入文件的句柄[①]。例如，要临时改变当前输入文件，可以这么做：

```
local temp = io.input()      -- 保存当前文件
io.input("newinput")         -- 打开一个新的当前文件
<对新的输入文件做一些操作>
io.input():close()           -- 关闭当前文件
io.input(temp)               -- 恢复原来的输入文件
```

21.2.1 性能小诀窍

通常在 Lua 中，一次性读取整个文件比逐行地读取要快一些。但必须处理一些大文件（几十或几百兆字节）时，就无法一次性地读取所有的内容。如果希望以最高性能来处理这种大文件，那么最快的方法就是用足够大的块（例如，8KB 大小的块）来读取文件。为了避免在行中间断开，只需在读一个块时再加上一行：

```
local lines, rest = f:read(BUFSIZE, "*line")
```

变量 rest 包含了被块所断开的那行的剩余部分。这样就可以将块与行的剩余部分连接起来，从而得到了一个总是起止于行边界上的块。

下面这个示例运行此技术实现了 wc，wc 是一个用于统计文件中字符数、单词数和行数的程序：

```
local BUFSIZE = 2^13          -- 8K
```

① 类似的调用也适用于 io.output。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
local f = io.input(arg[1])   -- 打开输入文件
local cc, lc, wc = 0, 0, 0   -- 字符、行、单词的计数
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 + #lines
  -- 统计块中的单词数
  local _, t = string.gsub(lines, "%S+", "")
  wc = wc + t
  -- 统计块中的换行字符数量
  _,t = string.gsub(lines, "\n", "\n")
  lc = lc + t
end
print(lc, wc, cc)
```

21.2.2 二进制文件

简单模式中的函数 io.input 和 io.output 总是以文本方式打开文件（默认行为）。在 UNIX 中，二进制文件和文本文件是没有差别的。但在其他一些系统中，特别是在 Windows 中，必须用特殊的标识来打开二进制文件。在处理二进制文件时，io.open 的模式字符串中必须带有字母“b”。

在 Lua 中，二进制数据的处理与文本处理类似。Lua 中的字符串可能包含任意字节，库中几乎所有函数都能处理任意字节。只要模式字符串中不包含值为零的字节，甚至还可以对二进制数据作模式匹配。如果确实需要在模式字符串中包含值为零的字节，可以用转义字符%z 来表示。

通常在读取二进制数据时，使用*all 模式来读取整个文件，或者使用<num>模式来读取 n 个字节。下面是一个简单的示例程序，它会把 DOS 格式的文本文件转换为 UNIX 格式[①]。它并没有使用标准的 I/O 文件（stdin 和 stdout），因为这些文件都是以文本方式打开的。它假设输入文件和输出文件的名称分别由程序的参数指定：

```
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())
```

① 就是将“回车-换行（Carriage Return–newline）”序列转换为“换行（Newline）”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

可以用以下命令行来调用这个程序：

```
> lua prog.lua file.dos file.unix
```

下面是另外一个示例，它打印了在一个二进制文件中找到的所有字符串：

```
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.gmatch(data, pattern) do
  print(w)
end
```

这个程序假定字符串是一个以 0 结尾，并包含至少 6 个有效字符的序列。所谓“有效字符”是指被模式 validchars 所认可的任意字符。在这个示例中，这个模式包含了数字、字母、标点符号和空格字符。然后，通过 string.rep 和连接操作创建了一个新的模式，这个新模式可用于捕获 6 个或更多的 validchars。而其结尾的%z 用于匹配字符串末尾的零字节。

下面是最后一个示例，它打印了一个二进制文件的内容：

```
local f = assert(io.open(arg[1], "rb"))
local block = 16
while true do
  local bytes = f:read(block)
  if not bytes then break end
  for _, b in ipairs{string.byte(bytes, 1, -1)} do
    io.write(string.format("%02X ", b))
  end
  io.write(string.rep("   ", block - string.len(bytes)))
  io.write(" ", string.gsub(bytes, "%c", "."), "\n")
end
```

同样，程序的第一个参数是输入文件名，而结果则被输出到标准输出。这个程序以 16 字节作为一块读取文件。对于每个块，它先输出每个字节的十六进制表示。然后，将整个块作为文本输出，而块中的控制字符都会替换为点符号[1]。

以下是在 UNIX 系统上将这个程序应用于自身后的结果。

```
6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74  local f = assert
28 69 6F 2E 6F 70 65 6E 28 61 72 67 5B 31 5D 2C  (io.open(arg[1],
20 22 72 62 22 29 29 0A 6C 6F 63 61 6C 20 62 6C   "rb")).local bl
6F 63 6B 20 3D 20 31 36 0A 77 68 69 6C 65 20 74  ock = 16.while t
72 75 65 20 64 6F 0A 20 20 6C 6F 63 61 6C 20 62  rue do.  local b
...
```

① 注意其中的惯用写法{strings.byte(bytes, 1, -1)}，它会创建一个 table，包含了字符串 bytes 中的所有字节。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
6E 67 2E 67 73 75 62 28 62 79 74 65 73 2C 20 22  ng.gsub(bytes, "
25 63 22 2C 20 22 2E 22 29 2C 20 22 5C 6E 22 29  %c", "."), "\n")
0A 65 6E 64 0A                                   .end.
```

21.2.3 其他文件操作

函数 tmpfile 返回一个临时文件的句柄，这个句柄是以读/写方式打开。这个文件会在程序结束时自动删除。函数 flush 会将缓冲中的数据写入文件。它与 write 函数一样，将其作为一个函数调用时，io.flush()会刷新当前输出文件；而将其作为一个方法调用时，f:flush()会刷新某个特定的文件 f。

函数 seek 可以获取和设置一个文件的当前位置。它的一般形式是 f:seek(whence, offset)，其中 whence 参数是一个字符串，指定了如何解释 offset 参数。它的有效值包括："set"，offset 解释为相对于文件起始的偏移量；"cur"，offset 解释为相对于当前位置的偏移量；"end"，offset 解释为相对于文件末尾的偏移量。函数的返回值与 whence 无关，它总是返回文件的当前位置，即相对于文件起始处的偏移字节数。

whence 参数的默认值是"cur"，offset 的默认值是 0。因此，调用 file:seek()不会改变文件的当前位置，并会返回当前的文件位置。调用 file:seek("set")会将当前位置重置为文件的起始处（并返回 0）。调用 file:seek("end")会将当前位置设置到文件的末尾，并返回文件的大小。下面这个函数可以不改变文件的当前位置而获取文件的大小：

```
function fsize (file)
  local current = file:seek()        -- 获取当前位置
  local size = file:seek("end")      -- 获取文件大小
  file:seek("set", current)          -- 恢复位置
  return size
end
```

如果发生错误，所有这些函数都会返回 nil 和一条错误消息。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 22 章　操作系统库

操作系统库定义在 table os 中，其中包含了文件操作函数、获取当前日期和时间的函数，以及其他一些与操作系统相关的功能。为了保证 Lua 的可移植性，设计这个库时做了一些折中。由于 Lua 是使用 ANSI C 编写的，Lua 只使用了 ANSI 标准中定义的函数。而像目录操作和套接字这类操作系统功能并不是 ANSI 标准的一部分，因此操作系统库也就不包含它们了。不过，另外有一些独立的 Lua 库则提供了对这些操作系统功能的访问。例如，posix 库为 Lua 提供了 POSIX. 1 标准的功能，而 luasocket 库提供了网络支持。

对于文件操作而言，这个库只提供了两个函数，一个是用于文件改名的 os.rename 函数，另一个是用于删除文件的 os.remove 函数。

22.1　日期和时间

在 Lua 中，函数 time 和 date 提供了所有的日期和时间功能。

如果不带任何参数调用 time 函数，它会以数字形式返回当前的日期和时间[①]。如果用一个 table 作为参数调用它，它会返回一个数字，表示该 table 中所描述的日期和时间。这种 table 具有以下有效字段：

year	一个完整的年份
month	01 - 12
day	01 - 31
hour	00 - 23
min	00 - 59
sec	00 - 59
isdst	一个布尔值，true 表示夏令时

前三个字段是必须要有的，其他字段默认为中午（12:00:00）。在里约热内卢[②]运行的一个 UNIX 系统[③]上，可以写这样一个示例：

```
print(os.time{year=1970, month=1, day=1, hour=0})    --> 10800
```

① 在大多数系统中，这个数字表示从某一时刻至今的秒数。
② 比格林威治时间晚 3 个小时。
③ 开始日期为 1970 年 1 月 1 日 00:00:00 UTC。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(os.time{year=1970, month=1, day=1, hour=0, sec=1})
  --> 10801
print(os.time{year=1970, month=1, day=1})          --> 54000
```

其中，10800 是 3 小时的秒数，54000 则是 10800 加上 12 小时的秒数。

函数 date 是 time 的一个反函数，它可以将一个表示日期和时间的数字转换成某些高级的表现形式。其第一个参数是格式字符串，指定了期望的表示形式；第二个参数是日期和时间的数字，默认为当前日期和时间。

为了生成一个日期 table，可以使用格式字符串"*t"。例如，调用 os.date("*t", 906000490) 会返回下面这个 table：

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
 hour = 23, min = 48, sec = 10, isdst = false}
```

注意，除了用于 os.time 的那些字段之外，os.date 所创建的 table 中还包含了星期数（wday，1 表示星期天）和一年中的第几天（yday，1 是一月一日）。

而对于其他格式字符串，os.date 会将日期格式化为一个字符串，这个字符串是传入格式字符串的一个复制，但其中的某些特殊标记被替换成了时间和日期信息。所有的标记都以"%"开头，并伴随一个字母，例如：

```
print(os.date("today is %A, in %B"))    --> today is Tuesday, in May
print(os.date("%x", 906000490))         --> 09/16/1998
```

所有的表现形式取决于当前的区域设置。例如，当前区域设为“巴西—葡萄牙语”时，%B 会变成"setembro"，而%x 变为"16/09/98"。

下表列出了所有的标记及其含义，这些标记说明中使用的示范时间为 1998 年 9 月 16 日（星期三）23:48:10。对于数字值，表中也列出了它们的有效范围：

%a	一星期中天数的简写（例如：Wed）
%A	一星期中天数的全称（例如：Wednesday）
%b	月份的简写（例如：Sep）
%B	月份的全称（例如：September）
%c	日期和时间（例如：09/16/98 23:48:10）
%d	一个月中的第几天（16）[01~31]
%H	24 小时制中的小时数（23）[00~23]
%I	12 小时制中的小时数（11）[01~12]
%j	一年中的第几天（259）[001~366]
%M	分钟数（48）[00~59]
%m	月份数（09）[01~12]
%p	“上午（am）”或“下午（pm）”（pm）

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

%S	秒数（10）[00~59]
%w	一星期中的第几天（3）[0~6=星期天~星期六]
%x	日期（例如：09/16/98）
%X	时间（例如：23:48:10）
%y	两位数的年份（98）[00~99]
%Y	完整的年份（1998）
%%	字符'%'

如果不带任何参数调用 date 函数，它会使用格式%c，即以一种合理的格式表示完整的日期和时间信息。另外，%x、%X 和%c 会根据不同的区域和系统而发生变化。如果需要一种固定的表示形式，例如 mm/dd/yyyy，可以使用显式的格式字符串"%m/%d/%Y"。

函数 os.clock 会返回当前 CPU 时间的秒数，一般可用于计算一段代码的执行时间：

```
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 可中止当前程序的执行。函数 os.getenv 可获取一个环境变量的值，并接受一个变量名，返回对应的字符串值：

```
print(os.getenv("HOME"))    --> /home/lua
```

如果一个环境变量没有定义，则返回 nil。函数 os.execute 可运行一条系统命令，它等价于 C 语言中的 system 函数。它需要接收一个命令字符串，并返回一个错误代码。例如，在 UNIX 和 DOS-Windows 环境中，可以用以下函数创建新目录：

```
function createDir (dirname)
  os.execute("mkdir " .. dirname)
end
```

os.execute 函数有很多用处，但它与系统之间也有密切相关系。

函数 os.setlocale 设置当前 Lua 程序所使用的区域。区域定义了不同文化或不同语言间的差异之处。setlocale 函数有两个字符串参数区域名和分类名，分类参数指定了区域参数中哪组特征将起作用。区域中有 6 种分类："collate"控制字符串的字母顺序，"ctype"控制单个字符的类型[①]及其大小写间的转换，"monetary"不影响 Lua 程序，"numeric"控制如何格式化数字，"time"

① 例如，什么字符是字母。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

控制如何格式化日期和时间[1]，以及"all"控制上述所有功能。默认的分类是"all"，因此如果在调用 setlocale 时只指定区域名，那么它会设置所有的分类。setlocale 函数会返回区域名，如果失败则返回 nil[2]。

```
print(os.setlocale("ISO-8859-1", "collate"))  --> ISO-8859-1
```

另外要说一下分类"numeric"。在葡萄牙语或其他拉丁语言中，表示十进制小数时，是用逗号代替小数点的。区域设置会改变 Lua 打印和读取这些数字的方式，但是区域不会改变 Lua 分析数字的方法[3]。如果打算用 Lua 程序来创建另一段 Lua 代码，则可能会遇到这样的问题：

```
print(os.setlocale("pt_BR"))        --> pt_BR
s = "return (" .. 3.4 .. ")"
print(s)                            --> 返回(3,4)
print(loadstring(s))
  --> nil    [string "return (3,4)"]:1: ')' expected near ','
```

① 例如，会影响函数 os.date 的结果。
② 通常是因为系统不支持给定的区域。
③ 原因之一是因为像 print(3,4)这样的表达式在 Lua 中已经有了特定的含义。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 23 章　调　试　库

调试库并没有提供一个 Lua 的调试器，而是提供了一个编写调试器所必须具有的原语。考虑到性能因素，这些原语的标准接口是通过 C API 给出的。而 Lua 中的调试库则提供了一条在 Lua 代码中直接访问这些接口的途径。

调试库与其他库不同，必须慎重使用。因为，首先它的一些功能性能不高。其次，它会打破语言的一些固有原则，例如用户无法在一个函数之外访问这个函数内部创建的局部变量。通常，用户不会希望在产品的最终版本中打开这个库，或者删除它，那么可以使用 debug=nil 来移除这个库。

调试库由两类函数组成：自省函数（introspective function）和钩子（hook）。自省函数允许检查一个正在运行中程序的各个方面，例如它的活动函数栈、当前执行的行、局部变量的名称和值。钩子则允许跟踪一个程序的执行。

在调试库中有一个重要的概念是“栈层（stack level）”。一个栈层是一个数字，它表示某一时刻某个活动的函数，即一个已被调用但尚未返回的函数。调用调试库的函数是层 1，调用这个函数的函数是层 2，依此类推。

23.1　自 省 机 制

调试库中主要的自省函数是 debug.getinfo 函数。它的第一个参数可以是一个函数或一个栈层。当为某函数 foo 调用 debug.getinfo(foo)时，就会得到一个 table，其中包含了一些与该函数相关的信息。这个 table 中的字段有以下几种。

- source：函数定义的位置。如果函数是通过 loadstring 在一个字符串中定义的，那么 source 就是这个字符串。如果函数是在一个文件中定义的，那么 source 就是这个文件名加前缀‘@’。
- short_src：source 的短版本（最多 60 个字符），可用于错误信息中。
- linedefined：该函数定义在源代码中第一行的行号。
- lastlinedefined：该函数定义的源代码中最后一行的行号。
- what：函数的类型。如果 foo 是一个普通的 Lua 函数，则为"Lua"；如果是一个 C 函数，则为"C"；如果是一个 Lua 程序块（chunk）的主程序部分，则为"main"。
- name：该函数的一个适当的名称。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

- namewhat：上一个字段的含义。它可能是"global"、"local"、"method"、"field"或""（空字符串）。空字符串表示 Lua 没有找到该函数的名称。
- nups：该函数的 upvalue 的数量。
- activelines[①]：一个 table，包含了该函数的所有活动行的集合。所谓“活动行”就是含有代码的行，这是相对于空行或只包含注释的行而言的[②]。
- func：函数本身，见下文。

当 foo 是一个 C 函数时，Lua 没有关于它更多的信息。此时只有字段 what、name 和 namewhat 是有意义的。

当用一个数字<n>调用 debug.getinfo(n)，就可以得到相应栈层上函数的数据。例如，如果<n>是 1，就可以得到调用 debug.getinfo 的那个函数的数据[③]。如果<n>大于栈中函数的总数时，debug.getinfo 返回 nil。当用一个数字来调用 debug.getinfo 时，返回的 table 中还会包含一个额外字段 currentline，表示此时这个函数正在执行的那行。并且，func 就是处于该层上的那个函数本身。

字段 field 有些特殊。由于函数在 Lua 中是作为“第一类值（First-Class Value）”，它可以没有名称，也可以有多个名称。Lua 会检查调用该函数的代码，看它是如何被调用的，从而试着找出该函数的名称。这种方法只有当用一个数字调用 getinfo 时，才会起作用，即用户只能获取关于某一具体调用的信息。

getinfo 函数的效率不高。Lua 会以一种不影响程序正常执行的方式来保存调试信息。至于访问这些调试信息的效率则是次要的。为了得到更好的性能，getinfo 有第二个可选的参数，用于指定希望获取哪些信息。通过这个参数，函数就不会浪费时间去收集那些用户不需要的数据了。这个参数是一个字符串，其中每个字母代表一组字段，这些字母有：

'n'	选择 name 和 namewhat
'f'	选择 func
'S'	选择 source、short_src、what、linedefined 和 lastlinedefined
'l'	选择 currentline
'L'	选择 activelines
'u'	选择 nups

下面这个函数演示了 debug.getinfo 的使用。它打印出了一个简单的当前栈的追溯（traceback）：

```
function traceback ()
  for level = 1, math.huge do
```

① activelines 字段是 Lua 5.1 新引入的。
② 这些信息通常可用于设置断点。大多数调试器不允许在活动行之外设置断点，因为程序不会执行到那里。
③ 若<n>为 0，则得到 getinfo 自身的信息，它是一个 C 函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    local info = debug.getinfo(level, "Sl")
    if not info then break end
    if info.what == "C" then   -- 是一个C函数吗?
      print(level, "C function")
    else   -- a Lua function
      print(string.format("[%s]:%d", info.short_src,
                                     info.currentline))
    end
  end
end
```

要让这个函数打印出更多 getinfo 返回的数据也很容易。事实上，调试库也提供了这样一个改进版本，即函数 traceback。不同于 Lua 5.1 版本的是，debug.traceback 不会打印出结果，而是返回一个表示追溯结果的字符串（通常很长）。

23.1.1 访问局部变量

可以用 debug.getlocal 来检查任意活动函数的局部变量。这个函数有两个参数：一个是希望查询的函数栈层，另一个是变量的索引。它也会返回两个值变量的名字和它的当前值。如果变量索引大于活动变量的总数，getlocal 会返回 nil。如果栈层是无效的，getlocal 就会引发一个错误。可以使用 debug.getinfo 来检查栈层是否有效。

Lua 按局部变量在一个函数中的出现顺序为它们编号，但编号只限于在函数的当前作用域中活跃的变量。例如以下代码：

```
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)
```

会打印出：

```
a     10
b     20
x     nil
a     4
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

变量是 a（第一个参数）的索引是 1，b 是 2，x 是 3，另一个 a 是 4。在 getlocal 的调用点上，c 已经离开了作用域，而 name 和 value 还未出现于作用域内[①]。

还可以通过 debug.setlocal 改变局部变量的值。它的前两个参数与 getlocal 相同，是栈层和变量索引，第三个参数是该变量的新值。它会返回变量名，如果变量索引超出了范围，则返回 nil。

23.1.2 访问非局部的变量（non-local varible）

调试库还提供了函数 getupvalue，它使用户可以访问为一个 Lua 函数所使用的“非局部的变量”。与局部变量不同的是，被一个函数所引用的“非局部的变量”会一直存在着，即使这个引用它的函数已经执行完毕了[②]。因此，getupvalue 的第一个参数不是栈层，而是一个函数，更确切地说是一个 closure。第二个参数是变量索引。Lua 按照一个函数引用“非局部的变量”的顺序给它们编号。这个顺序是无关紧要的，因为一个函数不能用同一名称来访问两个“非局部的变量”。

还可以通过 debug.setupvalue 来修改“非局部的变量”。它有 3 个参数：一个 closure、一个变量索引和一个新值。与 setlocal 一样，它会返回变量的名称，如果变量索引超出范围，则返回 nil。

以下程序演示了如何通过变量名访问一个函数中任意变量的值。

```
function getvarvalue (name)
  local value, found

  -- 尝试局部变量
  for i = 1, math.huge do
    local n, v = debug.getlocal(2, i)
    if not n then break end
    if n == name then
      value = v
      found = true
    end
  end
 if found then return value end

  -- 尝试“非局部的变量”
  local func = debug.getinfo(2, "f").func
  for i = 1, math.huge do
    local n, v = debug.getupvalue(func, i)
    if not n then break end
    if n == name then return v end
```

① 注意，局部变量只有执行过它们的初始化代码之后才可见。
② 这也正是 closure 所蕴含的意思。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  end

  -- 还没有找到，访问环境
  return getfenv(func)[name]
end
```

如果有多个局部变量的名称与给定的名称相同，则尝试“局部变量”以获取具有最高索引的局部变量。这样就必须执行完整个循环。如果找不到具有该名称的局部变量，那么就尝试“非局部的变量”。首先，用 debug.getinfo 获取函数。然后，遍历它的“非局部的变量”。最后，如果还是找不到具有该名字的“非局部的变量”，就检索全局变量。注意，本例中使用数字 2 作为 debug.getlocal 和 debug.getinfo 调用的第一个参数，以此表示调用 getvarvalue 的那个函数。

23.1.3 访问其他协同程序

调试库中的所有自省函数都接受一个可选的协同程序参数作为第一个参数，这样就可以从外部来检查这个协同程序[①]。例如：

```
co = coroutine.create(function ()
  local x = 10
  coroutine.yield()
  error("some error")
end)

coroutine.resume(co)
print(debug.traceback(co))
```

对 traceback 的调用将作用于协同程序 co 上，会得到这样的结果：

```
stack traceback:
        [C]: in function 'yield'
        temp:3: in function <temp:1>
```

追溯没有进行到 resume 调用，因为协同程序和主程序运行在不同的栈上。

如果一个协同程序引发了一个错误，它并不会展开自身的栈。这样才可以在错误发生后检查它的信息。继续上面的示例，如果再次恢复协同程序的执行，它会提示一个错误：

```
print(coroutine.resume(co))     --> false  temp:4: some error
```

现在，如果打印它的追溯会得到以下的结果：

```
stack traceback:
```

① 该特性是 Lua 5.1 新引入的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
[C]: in function 'error'
temp:4: in function <temp:1>
```

即使是在发生错误后，也可以检查协同程序的局部变量：

```
print(debug.getlocal(co, 1, 1))    --> x     10
```

23.2 钩　　子

调试库中的钩子机制使用户可以注册一个钩子函数，这个函数会在程序运行中某个特定事件发生时被调用。有 4 种事件会触发一个钩子：每当 Lua 调用一个函数时产生的 call 事件；每当函数返回时产生的 return 事件；每当 Lua 开始执行一行新代码时产生的 line 事件；以及当执行完指定数量的指令后产生的 count 事件。Lua 会用一个字符串参数来调用钩子函数，这个字符串描述了导致调用钩子的事件"call"、"return"、"line"或"count"。对于 line 事件来说，Lua 还传入第二个参数，表示代码的行号。在钩子函数内部可以调用 debug.getinfo 获取更多的信息。

若要注册一个钩子，需要用两个或 3 个参数来调用 debug.sethook：第一个参数是钩子函数；第二个参数是一个字符串，描述了需要监控的事件；第三个参数是一个可选的数字，用于说明多久获得一次 count 事件。若要监控 call、return 和 line 事件，需要将它们的首字母（'c'、'r'、'l'）放入掩码字符串。若要监控 count 事件，则需要在第三个参数中指定一个计数器。若要关闭钩子，只需不带任何参数调用 sethook。

作为一个简单的示例，以下代码安装了一个简单的跟踪器，它会打印出解释器执行到的每一行：

```
debug.sethook(print, "l")
```

这句调用只是简单地将 print 安装为一个钩子函数，并告诉 Lua 在 line 事件时调用它。下面是一个具有更详细信息的跟踪器，它用 getinfo 获取了当前文件名，并将文件名添加到输出中：

```
function trace (event, line)
  local s = debug.getinfo(2).short_src
  print(s .. ":" .. line)
end

debug.sethook(trace, "l")
```

23.3 性能剖析（profile）

虽然这个库名为“调试库”，但也可以将它用于一些非调试的工作。一项常见的任务就是

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

性能剖析。如果是做计时性的剖析，最好使用 C 接口，因为每次 Lua 调用钩子的代价太高了，从而使得测试结果偏差较大。不过对于计数性的剖析，Lua 代码则可以做得很好。在这一节中，将开发一个原始的剖析器（Profiler），它会列出程序运行中每个函数的调用次数。

程序的主要数据结构是两个 table：一个将函数和它们的调用计数器关联起来，另一个关联函数和它们的名称。两个 table 的索引都是函数自身：

```
local Counters = {}
local Names = {}
```

可以在性能剖析完成后再检索函数的名称，但是如果能在一个函数处于活动状态时获取其名称，可能会得到更好的结果。这是因为那时 Lua 可以查看调用这个函数的代码来找出其名称。

现在定义一个钩子函数。它的任务是获取当前正在被调用的函数，并递增相应的计数器值，同时它还收集了函数名：

```
local function hook ()
  local f = debug.getinfo(2, "f").func
  if Counters[f] == nil then    -- 'f'是第一次被调用吗?
    Counters[f] = 1
    Names[f] = debug.getinfo(2, "Sn")
  else    -- 仅递增计数器值
    Counters[f] = Counters[f] + 1
  end
end
```

接下来用这个钩子运行程序。假设，被测试程序的主程序块位于一个文件中，并且用户向剖析器指定了该文件名：

```
% lua profiler main-prog
```

这样，剖析器就能在 arg[1]中得到文件名了。然后，打开钩子并运行文件：

```
local f = assert(loadfile(arg[1]))
debug.sethook(hook, "c")        -- 为后续调用打开钩子
f()                             -- 运行主程序
debug.sethook()                 -- 关闭钩子
```

最后一步是显示结果。下一个函数可以为每个被调用的函数产生一个名称。由于函数名在 Lua 中是不确定的，所以给每个函数再加上一个位置信息，以 file:line 这样的形式给出。如果一个函数没有名称，那么就只使用它的位置。如果一个函数是一个 C 函数，那么就只使用它的名称[①]：

① 因为它没有位置信息。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
function getname (func)
  local n = Names[func]
  if n.what == "C" then
    return n.name
  end
  local lc = string.format("[%s]:%s", n.short_src, n.linedefined)
  if n.namewhat ~= "" then
    return string.format("%s (%s)", lc, n.name)
  else
    return lc
  end
end
```

最后，打印出每个函数及其计数器：

```
for func, count in pairs(Counters) do
  print(getname(func), count)
end
```

如果将这个剖析器应用于10.2节中编写的“马尔可夫（Markov）”示例的话，会得到以下结果：

```
[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
```

这份结果显示第4行上的匿名函数[1]被调用了884723次，write（即io.write）函数被调用了10000次等。

对于这个剖析器，还有几个地方可以改进。例如，对输出进行排序，打印更好的函数名及美化输出格式。然而，这个基本的剖析器已经很有用了，可以作为更多高级工具的基础。

① 在allwords内定义的迭代器函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第4部分

Second Edition

Programming in Lua

- 第 24 章　C API 概述
- 第 25 章　扩展应用程序
- 第 26 章　从 Lua 调用 C
- 第 27 章　编写 C 函数的技术
- 第 28 章　用户自定义类型
- 第 29 章　管理资源
- 第 30 章　线程和状态
- 第 31 章　内存管理

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 24 章　C API 概述

Lua 是一种嵌入式语言。即 Lua 不是一个单独运行的程序，而是一个可以链接到其他程序的库。通过链接就可以将 Lua 的功能合并入这些程序。

如果 Lua 不是一个独立运行的程序，那么在本书中使用的 Lua 程序是怎么来的？这个问题的答案是 Lua 解释器，即可执行程序“lua”。这个解释器是一个简单的应用程序[①]，它依靠 Lua 库来实现主要功能。这个程序会处理与用户的交互，它会将用户的文件或字符串输入 Lua 库，由 Lua 库来完成主要的工作，例如真正地运行 Lua 代码等。

这种使用一个库来扩展应用程序的能力使得 Lua 成为一种“扩展语言（Extension Language）”。而与此同时，一个使用了 Lua 的程序可以在 Lua 环境中注册用 C 语言（或其他语言）实现的新函数，由此就可以向 Lua 添加某些无法直接用 Lua 编写的功能，这便使 Lua 成为一种“可扩展的语言（extensible language）”。

这两种使用 Lua 的方法对应于 C 语方和 Lua 之间的两种交互形式。在第一种形式中，C 语言拥有控制权，Lua 是一个库。这种形式中的 C 代码称为“应用程序代码”。在第二种形式中，Lua 拥有控制权，C 语言是一个库。因此 C 代码称为“库代码”。应用程序代码和库代码都使用同样的 API 来与 Lua 通信，这些 API 称为 C API。

C API 是一组能使 C 代码与 Lua 交互的函数[①]。其中包括读写 Lua 全局变量、调用 Lua 函数、运行一段 Lua 代码，以及注册 C 函数以供 Lua 代码调用等。

C API 遵循 C 语言的操作模式（modus operandi），这与 Lua 的操作模式大不相同。当在 C 语言中编程时，应该注意类型检测（和类型错误），错误恢复、内存分配错误和其他的源代码复杂性。API 中的大多数函数不会检测其参数的正确性，但必须保证传入参数的合法性。如果传入了错误的参数，并不会得到一个经过设计的错误消息，而是得到一个“segmentation fault[②]”或类似的错误。此外，API 的设计强调灵活性和简单性，因此有时它们的使用会略显复杂。普通的一个任务可能需要涉及几个 API 的调用。虽然有些麻烦，但却可以使用户完全控制所有的细节。

如本章标题所示，本章的目的是给出一个在 C 语言中使用 Lua 的概述。其中的某些细节，在后面的内容中会进行更详细地介绍。同时，在 Lua 的参考手册中还可以找到更多关于具体

---

① 少于 400 行代码。

① 更确切地说，这里所说的“函数”应表示“函数或宏”。有些 API 是通过宏实现的。

② 译者注：这是一种错误情况，发生在一个程序试图访问一个不允许访问的内存位置时，或以一种不允许的方式访问一个内存位置时（例如，写一块只读的区域，或者写一块属于操作系统内存）。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

函数的介绍。此外，在 Lua 的发行版本中也可以找到几个使用 API 的示例。Lua 的解释器程序（lua.c）就是“应用程序代码”的一个实例，而 Lua 标准库（lmathlib.c、lstrlib.c 等）则是“库代码”的实例。

从现在开始，就作为 C 程序员来工作。当出现“你”的时候，表示编写 C 的用户或用户编写的 C 代码。

Lua 和 C 语言通信的主要方法是一个无所不在的虚拟栈。几乎所有的 API 调用都会操作这个栈上的值。所有的数据交换，无论是 Lua 到 C 语言或 C 语言到 Lua 都通过这个栈来完成。此外，还可以用这个栈来保存一些中间结果。栈可以解决 Lua 和 C 语言之间存在的两大差异，第一种差异是 Lua 使用垃圾收集，而 C 语言要求显式地释放内存；第二种是 Lua 使用动态类型，而 C 语言使用静态类型。将在 24.2 节中详细讨论栈。

24.1 第一个示例

将通过一个简单的 Lua 解释器程序来开始 C API 的学习。以下代码就是一个最原始的解释器程序：

```
#include <stdio.h>
#include <string.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (void) {
  char buff[256];
  int error;
  lua_State *L = luaL_newstate();   /* 打开Lua */
  luaL_openlibs(L);                 /* 打开标准库 */

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);  /* 从栈中弹出错误消息 */
}
}

lua_close(L);
return 0;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

头文件 lua.h 定义了 Lua 提供的基础函数，包括创建 Lua 环境、调用 Lua 函数（如 lua_pcall）、读写 Lua 环境中全局变量，以及注册供 Lua 调用的新函数等。lua.h 中定义所有内容都有一个 lua_前缀。

头文件 lauxlib.h 定义了辅助库（auxiliary library，auxlib）提供的函数。它的所有定义都以 luaL_开头（如 luaL_loadbuffer）。辅助库是一个使用 lua.h 中 API 编写出的一个较高的抽象层。Lua 的所有标准库编写都用到了辅助库。基础 API 的设计保持原子性和正交性，而辅助库则侧重于解决具体的任务。当然，程序若要创建自己的抽象也是非常简单的。注意，辅助库并没有直接访问 Lua 的内部，它都是用官方的基础 API 来完成所有工作的。

Lua 库中没有定义任何全局变量。它将所有的状态都保存在动态结构 lua_State 中，所有的 C API 都要求传入一个指向该结构的指针。这种实现使得 Lua 可以重入（reenter），稍加修改即可用于多线程的代码中。

luaL_newstate 函数用于创建一个新环境（或状态）。当 luaL_newstate 创建一个新的环境时，新环境中没有包含预定义的函数，甚至没有 print。为了使 Lua 保持小巧，所有的标准库都被组织到了不同的包中。这样便可以忽略那些不需要的包。在头文件 lualib.h 中定义了打开这些库的函数，而辅助库函数 luaL_openlibs 则可以打开所有的标准库。

当创建好一个状态，并在其中加载了标准库后，就可以解释用户的输入了。程序调用 luaL_loadbuffer 来编译用户输入的每行内容。如果没有错误，此调用返回 0，并向栈中压入编译后的程序块[①]。然后，程序调用 lua_pcall，这个函数会将程序块从栈中弹出，并在保护模式中运行它。与 luaL_loadbuffer 类似，lua_pcall 返回 0 表示没有错误。若发生错误，那么这些函数就会向栈中压入一条错误消息。用 lua_tostring 可以获取这条消息，打印后可以用 lua_pop 把它从栈中删除。

注意，在发生错误时，这个程序只是简单地将错误消息打印到标准错误流。在实际应用中真正的错误处理则可能更加复杂，如何做取决于具体程序的要求。Lua 的核心是不会直接将任何内容写到任何输出流中的，当发生错误时，它只会返回错误代码或错误消息来通知调用者。每个程序都可以用恰当的方式来处理这些消息。为了简化讨论，假设现在只需一系列简单的错误处理就可以了，那么处理过程就是打印一条错误消息，然后关闭 Lua 状态，并结束整个程序：

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void error (lua_State *L, const char *fmt, ...) {
  va_list argp;
  va_start(argp, fmt);
```

① 在下一节中将讨论这个“神奇的”栈。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  lua_close(L);
  exit(EXIT_FAILURE);
}
```

在后面的内容中还会讨论更多的在应用程序代码中的错误处理方法。

Lua 可以同时作为 C 代码或 C++代码来编译。在某些 C 程序库中常会出现以下这种调节代码，而在 lua.h 中并没有包含它们：

```
#ifdef --cplusplus
extern "C" {
#endif
...
#ifdef --cplusplus
}
#endif
```

如果将 Lua 作为 C 代码来编译，并在 C++中使用它，那么可以包含 lua.hpp 来代替 lua.h。lua.hpp 定义为：

```
extern "C" {
#include "lua.h"
}
```

24.2 栈

在 Lua 和 C 语言之间交换数据时，需要面对两个问题。首先是动态类型和静态类型之间的区别，其次是自动内存管理和手动内存管理之间的区别。

在 Lua 中，当用户写 a[k] = v，k 和 v 可以是任意类型。甚至于 a 也有可能是其他类型，因为可以通过元表重载操作符。假设要在 C 语言中提供一个 API 函数 settable。由于 C 函数的参数是固定类型，所以必须为各种类型的参数编写一个 settable 函数。

可以在 C 语言中声明一些联合（union）类型来解决这个问题。假设这种类型叫 lua_Value，能够表示所有的 Lua 值。那么 settable 就可以声明为：

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

这种做法有两个缺点。首先，很难将这种复杂的类型映射到其他语言中。Lua 的设计目标不仅是为了便于 C/C++访问，还应该可以被 Java、Fortran、C#或其他语言访问。其次，Lua 采用垃圾收集机制。如果将一个 Lua table 保持在一个 C 变量中，Lua 引擎则无法搜索出。因此，它会认为这个 table 是垃圾文件，并回收它。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于上述原因，Lua API 中没有定义任何类似于 lua_Value 的类型，而是使用了一个抽象的栈，在 Lua 和 C 语言之间交换数据。栈中的每个元素都能保存任何类型的 Lua 值。要获取 Lua 中的一个值时（例如一个全局变量的值），只要调用一个 Lua API 函数，Lua 就会将指定的值压入栈中。要将一个值传给 Lua 时，需要先将这个值压入栈，然后调用 Lua API，Lua 就会获取该值并将其从栈中弹出。为了将 C 类型的值压入栈，或者从栈中获取不同类型的值，就需要为每种类型定义一个特定的函数。但这种定义的数量远远小于上例中提到的定义 settable 的数量。另外，由于这个栈是 Lua 管理的，垃圾收集器能确定 C 语言使用哪些值。

几乎所有的 API 函数都会用到这个栈。在第一个示例中已看到了，luaL_loadbuffer 将它的结果（编译好的程序块或错误消息）留在栈中；lua_pcall 会调用栈中的一个函数，若发生错误则将错误消息留在栈中。

Lua 严格地按 LIFO（Last in, First out，先出后进）规范来操作这个栈。当调用 Lua 时，Lua 只会改变栈的顶部。不过，C 代码则有更大的自由度，它可以检索栈中间的元素，甚至在栈的任意位置插入或删除元素。

24.2.1 压入元素

对于每种可以呈现在 Lua 中的 C 类型，API 都有一个对应的压入函数：常量 nil 可使用 lua_pushnil，双精度浮点数可使用 lua_pushnumber，整数可使用 lua_pushinteger，布尔（C 语言中的整数）可使用 lua_pushboolean，任意字符串（char*及长度）可使用 lua_pushlstring，以及零结尾的字符串可使用 lua_pushstring。

```
void lua_pushnil     (lua_State *L);
void lua_pushboolean (lua_State *L, int bool);
void lua_pushnumber  (lua_State *L, lua_Number n);
void lua_pushinteger (lua_State *L, lua_Integer n);
void lua_pushlstring (lua_State *L, const char *s, size_t len);
void lua_pushstring  (lua_State *L, const char *s);
```

另外，还有压入 C 函数和 userdata 值的函数，之后会再讨论。

类型 lua_Number 是 Lua 中的数字类型。默认为双精度浮点数，但有些发行版本为了适应某种硬件受限的环境，会将数字类型改为单精度浮点数或长整数。类型 lua_Integer 是一种整数类型，它足以存储大型字符串的长度，通常定义为 ptrdiff_t 类型。

Lua 中的字符串不是以零结尾的，它们可以包含任意二进制数据。因此，它们必须同时保存一个显式的长度。将字符串压入栈的基本函数是 lua_pushlstring，它要求传入一个显式的长度参数。对于零结尾的字符串，可以使用函数 lua_pushstring，这个函数通过 strlen 来计算字符串的长度。Lua 不会持有指向外部字符串的指针[①]。对于所有 Lua 持有的字符串，它都会生成

① 也不会持有指向任何其他外部对象的指针，但除了 C 函数，因为 C 函数总是静态的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一个内部副本，或者复用现有的内容。因此，即使在这些函数返回后立刻释放或修改这些字符串，也不会出现问题。

向栈中压入一个元素时，应该确保栈中具有足够的空间。当 Lua 启动时，或 Lua 调用 C 语言时，栈中至少会有 20 个空闲的槽[1]。这些空间对于普通的应用是足够了，所以一般我们无须顾及空间上的问题。然而，有些任务会需要更多的栈空间，例如，调用一个具有很多参数的函数。在这些情况中，就要调用 lua_checkstack 来检查栈中是否有足够的空间：

```
int lua_checkstack (lua_State *L, int sz);
```

24.2.2 查询元素

API 使用“索引”来引用栈中的元素。第一个压入栈中的元素索引为 1；第二个压入的元素索引为 2，依此类推直到栈顶。还可以以栈顶为参考物，使用负数的索引来访问栈中的元素。此时，-1 表示栈顶元素（最后压入的元素），-2 表示栈顶下面的元素，依此类推。例如，调用 lua_tostring(L, -1)会将栈顶的值作为一个字符串返回。其中有些情况适用于从栈底索引栈（即使用正数索引），而另外一些情况则便于使用负数索引。

为了检查一个元素是否为特定的类型，API 提供了一系列的函数 lua_is*，其中*可以是任意 Lua 类型。这些函数有 lua_isnumber、lua_isstring 和 lua_istable 等。所有这些函数都有同样的原型：

```
int lua_is* (lua_State *L, int index);
```

实际上，lua_isnumber 不会检查值是否为数字类型，而是检查值是否能转换为数字类型。lua_isstring 也具有同样的行为。因此，对于任意数字，lua_isstring 都返回真。

还有一个函数 lua_type，它会返回栈中元素的类型。每种类型都对应于一个常量，这些常量定义在头文件 lua.h 中，它们是 LUA_TNIL、LUA_TBOOLEAN、LUA_TNUMBER、LUA_TSTRING、LUA_TTABLE、LUA_TTHREAD、LUA_TUSERDATA 和 LUA_TFUNCTION。这个函数一般可用在一个 switch 语句中。另外，若要检查一个元素是否为真正的字符串或数字（无须转换的），也可以使用这个函数。

lua_to*函数用于从栈中获取一个值：

```
int        lua_toboolean (lua_State *L, int index);
lua_Number lua_tonumber  (lua_State *L, int index);
lua_Integer lua_tointeger (lua_State *L, int index);
const char *lua_tolstring (lua_State *L, int index, size_t *len);
size_t     lua_objlen    (lua_State *L, int index);
```

① 这个常量是由 lua.h 中的 LUA_MINSTACK 定义的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

如果指定的元素不具有正确的类型，调用这些函数也不会有问题。在这种情况下，lua_toboolean、lua_tonumber、lua_tointeger 和 lua_objlen 会返回 0，而其他函数会返回 NULL。返回 0 并不是很有用，但 ANSI C 也没有提供其他可以表示错误的值。至于其他 lua_to*函数，通常不先使用 lua_is*函数，只需在调用它们之后测试返回结果是否为 NULL 就可以了。

lua_tolstring 函数会返回一个指向内部字符串副本的指针，并将字符串的长度存入最后一个参数 len 中。这个内部副本不能修改，返回类型中的 const 也说明了这点。Lua 保证只要这个对应的字符串值还在栈中，那么这个指针就是有效的。当 Lua 调用的一个 C 函数返回时，Lua 就会清空它的栈。这就形成了一条规则，不要在 C 函数之外使用在 C 函数内获得的指向 Lua 字符串的指针。

所有 lua_tolstring 返回的字符串在其末尾都会有一个额外的零，不过这些字符串的中间也可能会有零。字符串长度通过第三个参数 len 返回，这才是真正的字符串长度。进一步说，假设栈顶的值是一个字符串，如下总是为真：

```
size_t l;
const char *s = lua_tolstring(L, -1, &l);   /* 任何 Lua 字符串 */
assert(s[l] == '\0');
assert(strlen(s) <= l);
```

如果不需要长度信息，可以将第三个参数设为 NULL 来调用 lua_tolstring。或者使用宏 lua_tostring，这个宏就是用 NULL 作为第三个参数来调用 lua_tolstring。

lua_objlen 函数可以返回一个对象的“长度”。对于字符串和 table，这个值是长度操作符‘#’的结果。这个函数还可用于获取一个“完全 userdata（full userdata）”的大小[①]。

为了演示这些函数的使用，以下代码实现了一个有用的辅助函数，它会打印整个栈的内容。这个函数会由下而上地遍历栈，并根据每个元素的类型打印其值，字符串放在一对单引号内打印，数字使用格式“%g”来打印，其他值（table、函数等）则只打印它们的类型。其中，lua_typename 可将一个类型编码转换为一个类型名。

```
static void stackDump (lua_State *L) {
  int i;
  int top = lua_gettop(L);
  for (i = 1; i <= top; i++) {  /* 遍历所有层 */
    int t = lua_type(L, i);
    switch (t) {
      case LUA_TSTRING: {  /* 字符串 */
        printf("'%s'", lua_tostring(L, i));
        break;
      }
      case LUA_TBOOLEAN: {  /* 布尔 */
```

① 将在 28.1 节中讨论 userdata。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
      printf(lua_toboolean(L, i) ? "true" : "false");
      break;
    }
    case LUA_TNUMBER: {  /* 数字 */
      printf("%g", lua_tonumber(L, i));
      break;
    }
    default: {  /* 其他值 */
      printf("%s", lua_typename(L, t));
      break;
    }
  }
  printf("  ");  /* 打印一个分隔符 */
 }
 printf("\n");  /* 列表结尾 */
}
```

24.2.3 其他栈操作

除了在 C 语言和栈之间交换数据的函数外，API 还提供了以下这些用于普通栈操作的函数：

```
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);
```

lua_gettop 函数返回栈中元素的个数，也可以说是栈顶元素的索引。lua_settop 将栈顶设置为一个指定的位置，即修改栈中元素的数量。如果之前的栈顶比新设置的更高，那么高出来的这些元素会被丢弃；反之，会向栈中压入 nil 来补足大小。有一个特例，调用 lua_settop(L,0) 能清空栈。也可以用负数索引来使用 lua_settop。另外，API 根据这个函数还提供了一个宏，用于从栈中弹出 *n* 个元素：

```
#define lua_pop(L,n)  lua_settop(L, -(n) - 1)
```

lua_pushvalue 函数会将指定索引上值的副本压入栈。lua_remove 删除指定索引上的元素，并将该位置之上的所有元素下移以填补空缺。lua_insert 会上移指定位置之上的所有元素以开辟一个槽的空间，然后将栈顶元素移到该位置。lua_replace 弹出栈顶的值，并将该值设置到指定索引上，但它不会移动任何东西。注意，以下操作不会对栈产生影响：

```
lua_settop(L, -1);  /* 将栈顶元素设置为它的当前值 */
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
lua_insert(L, -1);  /* 将栈顶元素移动栈顶 */
```

以下程序演示了这些栈操作，它还用到了上节中的 stackDump 函数：

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"

static void stackDump (lua_State *L) {
  <如上节中示例>
}

int main (void) {
  lua_State *L = luaL_newstate();

  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

C 语言不同于 C++和 Java，它没有提供异常处理机制。为了克服这个困难，Lua 使用 C 语言中的 setjmp 机制，这是一种类似于异常处理的机制。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Lua 中所有的结构都是动态的，它们会根据需要来增长，或者缩小。在 Lua 中有许多地方可能会发生内存分配错误。几乎所有的函数都要面对这种潜在的错误。那么在发生错误时，与其让 API 中的每个函数返回错误代码，不如使用异常来标记这些错误。因此，几乎所有的 API 函数都会抛出错误（即调用 longjmp），而不是返回错误。

当编写库代码时（被 Lua 调用的 C 函数），使用 longjmp 几乎和使用异常处理机制一样方便，Lua 会捕获所有可能的错误。而当编写应用程序代码时（调用 Lua 的 C 代码），必须提供一种捕获错误的方式。

24.3.1 应用程序代码中的错误处理

通常情况下，应用程序代码是以“无保护（unprotected）”模式运行的。由于它们不是由 Lua 调用的，Lua 无法设置适当的上下文来捕获错误[①]。因此，当 Lua 发现了例如“内存不足”这类错误时，它基本上不会进行太多的处理。此时 Lua 会调用一个“紧急”函数（Panic Function），当这个函数返回后，Lua 就会结束应用程序。用户可以通过函数 lua_atpanic 来设置自己的“紧急”函数。

不是所有的 API 函数都会抛出异常。函数 luaL_newstate、lua_load、lua_pcall 和 lua_close 都是安全的。如果发生内存分配错误，其他大多数函数都会抛出异常。例如，如果 luaL_loadfile 无法为文件名字符串分配到足够的内存，它就会失败。有些程序无须处理内存不足的情况，它们可以忽略这些异常。而对于有些程序，若 Lua 发生内存不足，则可以让其正常地调用“紧急”函数。

如果发生了内存分配错误，而又不想结束应用程序，那么有两种做法。第一种是设置一个“紧急”函数，让它不要把控制权返回给 Lua。例如，可以调用 longjmp 转到之前 setjmp 所设置的位置。第二种做法是让代码在“保护模式”下运行。

大多数应用程序（包括 Lua 解释器程序）都采用第二种做法，它们调用 lua_pcall 来运行 Lua 代码。因而这些 Lua 代码也都是运行在保护模式中的。如果发生了内存分配错误，lua_pcall 会返回一个错误代码，并将解释器封固在一致的状态。如果要保护那些与 Lua 交互的 C 代码，可以使用 lua_cpcall。这个函数类似于 lua_pcall，但它接受一个 C 函数作为参数，然后调用这个 C 函数。将一个函数压入栈中不会有内存分配失败的可能。

24.3.2 库代码中的错误处理

Lua 是一种安全的语言，无论写什么，写出来的内容是否正确，都能用 Lua 自身的术语来理解程序的行为。此外，错误也是通过 Lua 的术语来检测和解释的。可以用 C 语言来作一个

① 也就是无法调用 setjmp。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

对比，许多 C 程序的错误行为只能用底层硬件的术语来解释，而错误位置则是由“程序计数器”寄存器给出的。

当将新的 C 函数加入 Lua 时，就有可能打破这种安全性。例如，添加一个函数 poke，它能在任意内存地址上存储任意字节。这个函数就有可能引起各种内存破坏。因此必须确保新加入的函数对 Lua 是安全的，并提供良好的错误处理。

正如之前所说的，每个 C 程序都有其各自处理错误的方法。然而，当为 Lua 编写库函数时，却只有一种标准的错误处理方法。当一个 C 函数检测到一个错误时，它就应该调用 lua_error。lua_error 函数会清理 Lua 中所有需要清理的东西，然后跳转回发起执行的那个 lua_pcall，并附上一条错误消息。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 25 章　扩展应用程序

Lua 的一项重要用途就是作为一种配置语言（configuration language）。本章将介绍如何用 Lua 来配置一个程序。将从一个简单的示例开始，逐步地扩展它，使其完成更复杂的任务。

25.1 基　　础

第一个任务是一个简单的配置应用。假设 C 程序有一个窗口，并希望用户能指定窗口的初始大小。显然，对于这种简单的任务，有多种比 Lua 更简单的做法，例如使用环境变量或使用记录了名值对的文件。不过就算使用一个简单的文本文件，也需要进行分析。因此使用 Lua 来作为配置文件[①]。下面是这种文件最简单的形式，它可以包含如下内容：

```
-- 定义窗口大小
width = 200
height = 300
```

此时，必须用 Lua API 来指挥 Lua 分析这个文件，并获取全局变量 width 和 height 的值。下面这个 load 函数完成了此项工作：[②]

```
void load (lua_State *L, const char *fname, int *w, int *h) {
if (luaL_loadfile(L, fname) || lua_pcall(L, 0, 0, 0))
   error(L, "cannot run config. 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");
  *w = lua_tointeger(L, -2);
  *h = lua_tointeger(L, -1);
}
```

假设已经创建了一个 Lua 状态。这个函数调用 luaL_loadfile 从文件 fname 加载程序块，然后调用 lua_pcall 运行编译好的程序块。若发生错误（例如配置文件中的语法错误），这两个函

① 其实也就是一个普通的文件，但同时又是一个有效的 Lua 程序。
② 其中函数 error 的定义请参阅 24.1 节。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数都会把错误消息压入栈，并返回一个非零的错误代码。此时，程序就调用 lua_tostring 从栈顶获取该消息。

当运行完程序块后，程序需要获取全局变量的值。程序调用了 lua_getglobal 两次，这个函数除了第一个常规的 lua_State 参数外，还需要变量的名称。每次调用这个函数，它都会将相应的全局变量值压入栈中。因此 width 处于索引-2 上，height 位于索引-1 上（栈顶）。另外，由于栈事先是空的，也可以从栈底进行索引，第一个值使用索引 1，第二个值使用 2。不过，自上向下的索引可以使代码即使是在栈不为空时依然可以工作。接下来程序使用 lua_isnumber 来检查两个值是否为数字。最后调用 lua_tointeger 将这些值转换为整型，并赋予对应的参数变量。

那么是否值得用 Lua 来完成这类任务呢？其实对于这类简单的任务，用一个简单的文件来记录这两个数字就足够了，这比 Lua 更易于使用。但使用 Lua 却可以带来一些优势。首先，Lua 会处理所有的语法细节或语法错误，包括配置文件中的注释。其次，用户可以实现一些更复杂的配置逻辑。例如，脚本可以提示用户某些信息，或者查询一个环境变量来选择合适的大小：

```
-- 配置文件
if getenv("DISPLAY") == ":0.0" then
  width = 300; height = 300
else
  width = 200; height = 200
end
```

即使是在这样一个简单的配置示例中，用户也可能会有很多实现需求。不过无论是何种实现，只要脚本定义了这两个变量，C 程序则无须修改就能工作的。

最后一个使用 Lua 的理由是，它更易于将新的配置机制添加到程序中。这种简易性可以让人形成一种态度，从而使程序变得更加灵活。

25.2 table 操作

接下来要完成的任务是，配置一个窗口的背景颜色。假设，颜色的格式是由 3 个数字组成的，每个数字都是 RGB 的一个颜色分量。在 C 语言中，这些数字通常是在区间[0, 255]中的整型。但在 Lua 中，由于所有的数字都是实数，所以可以使用区间[0, 1]。

一种最其本的做法是要求用户将每个分量设置在不同的全局变量中：

```
-- 配置文件
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

但是，这种做法有两个缺点：第一，它太冗长了；第二，无法预定义常用颜色。如果能定义常用颜色，用户就可以简单地写出 background = WHITE。为了避免这些缺点，使用 table 来表示颜色：

```
background = {r=0.30, g=0.10, b=0}
```

使用 table 可以让脚本变得更加结构化。现在，用户就可以很容易地在配置文件中预定义常用颜色了：

```
BLUE = {r=0, g=0, b=1}
<其他颜色定义>

background = BLUE
```

若要在 C 语言中获取这些值，可以如下所示：

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
  error(L, "'background' is not a table");

red = getfield(L, "r");
green = getfield(L, "g");
blue = getfield(L, "b");
```

先获取全局变量 backgroud 的值，并确认其是一个 table。然后，使用 getfield 获取颜色中的各个分量。不过，这个函数不是 API 函数，因此必须定义它。然而由于在这里又遇到了多态的问题，所以需要更多版本的 getfield 函数，以针对不同的 key 类型、value 类型和错误处理等。Lua API 只提供了一个函数 lua_gettable，它能处理所有的类型。但它需要知道 table 在栈中的位置，然后才会从栈中弹出 key，并压入相应的 value。getfield 的定义如下：

```
#define MAX_COLOR      255

/* 假设table位于栈顶 */
int getfield (lua_State *L, const char *key) {
   int result;
  lua_pushstring(L, key);
  lua_gettable(L, -2);  /* 获取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);  /* 删除数字 */
  return result;
}
```

这个函数假设 table 位于栈顶。当用 lua_pushstring 压入 key 后，table 就位于索引-2 上。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在返回前，getfield 弹出从栈中检索到的值，并使栈保持为调用前的样子。

由于经常需要用字符串来索引 table，为此 Lua 5.1 提供了一个 lua_gettable 的特化版本 lua_getfield。通过这个函数，可以将如下两行：

```
lua_pushstring(L, key);
lua_gettable(L, -2);  /* 获取background[key] */
```

重写为：

```
lua_getfield(L, -1, key);
```

由于没有向栈中压入字符串，所以当调用 lua_getfiled 时，table 的索引仍为-1。

接下来继续扩展这个示例。现在就为用户预定义各种常用颜色。用户除了可以使用自己创建的颜色 table 外，就还可以使用预定义的常用颜色。下面在 C 程序中创建这些颜色 table：

```
struct ColorTable {
      char *name;
      unsigned char red, green, blue;
    } colortable[] = {
      {"WHITE",   MAX_COLOR, MAX_COLOR, MAX_COLOR},
      {"RED",     MAX_COLOR,   0,   0},
      {"GREEN",     0, MAX_COLOR,   0},
      {"BLUE",      0,   0, MAX_COLOR},
      <other colors>
      {NULL,        0, 0, 0}  /* 结尾 */
    };
```

接下来要根据这些颜色名来创建相应的全局变量，然后用颜色 table 来初始化这些变量。最终结果应等价于用户在其脚本中写入这些内容：

```
WHITE = {r=1, g=1, b=1}
RED   = {r=1, g=0, b=0}
<其他颜色>
```

定义一个辅助函数 setfield 来设置 table 字段。它会将字段名和字段值压入栈中，然后调用 lua_settable：

```
/* 假设table位于栈顶 */
void setfield (lua_State *L, const char *index, int value) {
lua_pushstring(L, index);
  lua_pushnumber(L, (double)value/MAX_COLOR);
  lua_settable(L, -3);
}
```

就像其他 API 函数一样，lua_settable 能处理各种类型，它会从栈中获取所需的操作数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

lua_settable 要求传入一个 table 索引参数，然后它会设置这个 table，并弹出 key 和 value。setfield 函数假设在调用前 table 就已在栈顶（索引为-1）。当压入 key 和 value 后，table 就位于索引-3。

Lua 5.1 同样为字符串 key 提供了一个 lua_settable 的特化版本，名为 lua_setfield。通过这个函数，可以将上述 setfield 的定义重写为：

```
void setfield (lua_State *L, const char *index, int value) {
  lua_pushnumber(L, (double)value/MAX_COLOR);
  lua_setfield(L, -2, index);
}
```

接下来的一个函数是 setcolor，它用于定义一个颜色。它会创建一个 table，并设置相应的字段，最后将这个 table 赋予相应的全局变量：

```
void setcolor (lua_State *L, struct ColorTable *ct) {
   lua_newtable(L);                /* 创建一个table */
   setfield(L, "r", ct->red);      /* table.r = ct->r */
   setfield(L, "g", ct->green);    /* table.g = ct->g */
   setfield(L, "b", ct->blue);     /* table.b = ct->b */
   lua_setglobal(L, ct->name);     /* 'name' = table */
}
```

setcolor 先调用 lua_newtable，这个函数会创建一个空的 table，并将其压入栈中。然后，setcolor 调用 setfield 来设置 table 的各个字段。最后，lua_setglobal 弹出 table，并根据名称将其赋予全局变量。

通过上述函数，下面这个循环便会为配置脚本注册所有的颜色：

```
int i = 0;
while (colortable[i].name != NULL)
  setcolor(L, &colortable[i++]);
```

记住，应用程序必须在运行脚本前，执行这个循环。

下面是另一种实现“具名（Named）”颜色的做法。

```
lua_getglobal(L, "background");
if (lua_isstring(L, -1)) {   /* 该值是一个字符串吗？ */
  const char *name = lua_tostring(L, -1);  /* 获取字符串 */
  int i;   /* 搜索颜色table */
  for (i = 0; colortable[i].name != NULL; i++) {
    if (strcmp(colorname, colortable[i].name) == 0)
      break;
  }
  if (colortable[i].name == NULL)  /* 没有找到字符串吗？ */
    error(L, "invalid color name (%s)", colorname);
  else {  /* 使用colortable[i] */
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
      red = colortable[i].red;
      green = colortable[i].green;
      blue = colortable[i].blue;
    }
  } else if (lua_istable(L, -1)) {
    red = getfield(L, "r");
    green = getfield(L, "g");
    blue = getfield(L, "b");
  } else
    error(L, "invalid value for 'background'");
```

这里没有用到全局变量，而是让用户用字符串来表示颜色名称。例如，background = "BLUE"。现在，background 既可以是 table 又可以是字符串。若以这种方式来实现，应用程序则无须在运行用户脚本前做任何事情。不过，它需要在获取颜色时做更多的事情。当它获取变量 background 的值时，必须测试该值是否为合法的字符串，这需要在颜色表中查找该字符串。

在 C 程序中，用字符串来表示选项并不是一个好方法，因为编译器无法检测到拼写错误。在 Lua 中，全局变量无须声明，因此若用户错误地拼写了一个颜色名，Lua 也不会报告错误。如果用户写了 WITE，而非 WHITE，background 变量会变成 nil[①]。而应用程序却只知道 background 是 nil，除此之外没有其他信息可以说明错误的原因。另一方面，使用字符串时，若 background 的值（字符串）拼写错误，则应用程序可以将这个信息附加到错误消息中，还可以用与大小写无关的方式来比较字符串，如用户可以写"white"、"WHITE"或"White"。此外，如果用户脚本很小，而颜色很多，那么就需要注册许多颜色，但只有其中一些会被用户用到。在这种情况下，使用字符串方式可以避免这种开销。

25.3 调用 Lua 函数

Lua 允许在一个配置文件中定义函数，并且还允许应用程序调用这些函数。例如，若用户写的一个应用程序可以用来绘制一些函数的图形，那么就可以用 Lua 来定义这些函数。

调用函数的 API 协议很简单。首先，将待调用的函数压入栈，并压入函数的参数。然后，使用 lua_pcall 进行实际的调用。最后，将调用结果从栈中弹出。

例如，假设配置文件中有这样一个函数：

```
function f (x, y)
  return (x^2 * math.sin(y))/(1 - x)
end
```

可以在 C 语中对它求值，对于给定的 x 和 y，有 z = f(x, y)。假设，已打开了 Lua 库，并

① WITE 的值，也就是一个未初始化变量的值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

运行了配置文件。那么可以用下面这个 C 函数来调用这个 Lua 函数：

```
/* 调用Lua中定义的函数'f' */
double f (double x, double y) {
  double z;

/* 压入函数和参数 */
  lua_getglobal(L, "f");  /* 待调用的函数 */
  lua_pushnumber(L, x);   /* 压入第一个参数 */
  lua_pushnumber(L, y);   /* 压入第二个参数 */

/* 完成调用（2 个参数，1 个结果） */
  if (lua_pcall(L, 2, 1, 0) != 0)
    error(L, "error running function 'f': %s",
            lua_tostring(L, -1));

/* 检索结果 */
  if (!lua_isnumber(L, -1))
    error(L, "function 'f' must return a number");
  z = lua_tonumber(L, -1);
  lua_pop(L, 1);  /* 弹出返回值 */
  return z;
}
```

在调用 lua_pcall 时，第二个参数是传给待调用函数的参数数量，第三个参数是期望的结果数量，第四个参数是一个错误处理函数的索引。就像 Lua 的赋值一样，lua_pcall 会根据要求的数量来调整实际结果的数量，即压入 nil 或丢弃多余的结果。在压入结果前，lua_pcall 会先删除栈中的函数及其参数。如果一个函数会返回多个结果，那么第一个结果最先压入。例如，函数返回了 3 个结果，第一个的索引就是-3，最后一个的索引是-1。

如果在 lua_pcall 的运行过程中有任何错误，lua_pcall 会返回一个非零值，并在栈中压入一条错误消息。不过即使如此，它仍会弹出函数及其参数。然而，在压入错误消息前，如果存在一个错误处理函数，lua_pcall 就会先调用它。通过 lua_pcall 的最后一个参数可以指定这个错误处理函数。零表示没有错误处理函数，那么最终的错误消息就是原来的消息。若传入非零参数，那么这个参数就应该是一个错误处理函数在栈中索引。因此，错误处理函数必须先压入栈中，也就是必须位于待调用函数及其参数的下面。

对于普通的错误，lua_pcall 会返回错误代码 LUA_ERRRUN。但有两种特殊的错误情况，不会运行错误处理函数。第一种是内存分配错误，对于这类错误，lua_pcall 总是返回 LUA_ERRMEM。第二类错误则发生在 Lua 运行错误处理函数时，在这种情况中，是没有必要再次调用错误处理函数的，因此 lua_pcall 会立即返回错误代码 LUA_ERRERR。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

25.4 一个通用的调用函数

本例作为一个更高级的示例，将编写一个调用 Lua 函数的辅助函数，其中用到了 C 语言的可变参数机制。这个辅助函数称为 call_va，它接受一个待调用函数的名字、一个描述参数类型和结果类型的字符串，以及所有的参数变量和所有存放结果的指针。call_va 会处理所有的 API 细节。通过这个函数，可以将前例写为：

```
call_va("f", "dd>d", x, y, &z);
```

其中字符串"dd>d"表示“两个双精度类型的参数和一个双精度类型的结果”。在这段描述字符串中，可以用字母'd'表示双精度浮点数、'i'表示整数、's'表示字符串，而'>'表示参数与结果的分隔符。如果函数没有结果，'>'便是可选的。

以下是函数 call_va 的实现，这个函数执行了与第一示例中相同的步骤：压入函数、压入参数、完成调用和获取结果。

```
#include <stdarg.h>

void call_va (const char *func, const char *sig, ...) {
  va_list vl;
  int narg, nres;  /* 参数和结果的数量 */

va_start(vl, sig);
  lua_getglobal(L, func);  /* 压入函数 */

<压入参数>

nres = strlen(sig);  /* 期望的结果数量 */

if (lua_pcall(L, narg, nres, 0) != 0)  /* 完成调用 */
  error(L, "error calling '%s': %s", func,
                                  lua_tostring(L, -1));

<检索结果>

  va_end(vl);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

下面是“压入参数”的代码：

```
for (narg = 0; *sig; narg++) {  /* 遍历所有参数 */

/* 检查栈中空间 */
luaL_checkstack(L, 1, "too many arguments");

switch (*sig++) {
   case 'd':  /* double 参数 */
      lua_pushnumber(L, va_arg(vl, double));
            break;
 case 'i':  /* int 参数 */
  lua_pushinteger(L, va_arg(vl, int));
     break;
case 's':  /* 字符串参数 */
  lua_pushstring(L, va_arg(vl, char *));
  break;
case '>':  /* 参数结束 */
  goto endargs;
default:
   error(L, "invalid option (%c)", *(sig - 1));
}

}
endargs:
```

下面是“检索结果”的代码：

```
nres = -nres;  /* 第一个结果的栈索引 */
while (*sig) {  /* 遍历所有结果 */
  switch (*sig++) {
   case 'd':  /* double 结果 */
     if (!lua_isnumber(L, nres))
      error(L, "wrong result type");
     *va_arg(vl, double *) = lua_tonumber(L, nres);
     break;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
case 'i':  /* int 结果 */
    if (!lua_isnumber(L, nres))
      error(L, "wrong result type");
    *va_arg(vl, int *) = lua_tointeger(L, nres);
    break;
  case 's':  /* string 结果 */
    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++;
}
```

以上大部分代码都很直观，不过有些地方需要说明一下。首先，无须检查 func 是否为一个函数，因为 lua_pcall 会发现这类错误。其次，由于它要压入任意数量的参数，因此必须确保栈中具有足够的空间。第三，由于函数可能会返回字符串，因此 call_va 不能将结果弹出栈。调用者必须在使用完字符串结果（或将字符串复制到其他缓冲）后弹出所有结果。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 26 章　从 Lua 调用 C

扩展 Lua 的一项基本含义就是，应用程序将新的 C 函数注册到 Lua 中。

Lua 能调用 C 函数，但并不意味着 Lua 可以调用任意 C 函数[①]。在上一章中，当 C 语言调用 Lua 函数时，它必须遵循一个简单的协议，以此来向 Lua 传递参数，并从 Lua 获取结果。同样，对于一个能被 Lua 调用的 C 函数，它也必须遵循一个获取参数和返回结果的协议。此外，还必须注册 C 函数，以便用某种适当的方式将函数地址告诉 Lua。

当 Lua 调用 C 函数时，也使用了一个与 C 语言调用 Lua 时相同的栈。C 函数从栈中获取函数参数，并将结果压入栈中。为了在栈中将函数结果与其他值区分开，C 函数还应返回其压入栈中的结果数量。

栈不是一个全局性的结构，这是一个重要概念。每个函数都有自己的局部私有栈。当 Lua 调用一个 C 函数时，第一个参数总是这个局部栈的索引 1。即使这个 C 函数调用了 Lua 代码，并且 Lua 代码又调用了相同的 C 函数，这些 C 函数调用只看到自己的私有栈，它们的第一个参数都是索引 1。

26.1　C　函　数

第一个示例实现了一个简化版本的正弦函数：

```
static int l_sin (lua_State *L) {
  double d = lua_tonumber(L, 1);  /* 获取参数 */
  lua_pushnumber(L, sin(d));  /* 压入结果 */
  return 1;  /* 结果的数量 */
}
```

所有注册到 Lua 中的函数都具有相同的原型，该原型就是定义在 lua.h 中的 lua_CFunction：

```
typedef int (*lua_CFunction) (lua_State *L);
```

从 C 语言的观点来看，这个 C 函数仅有一个参数，即 Lua 的状态。它返回一个整数，表示其压入栈中的返回值数量。因此，这个函数无须在压入结果前清空栈。在它返回后，Lua 会自动删除栈中结果之下的内容。

① 有一些扩展包可以让 Lua 调用任意 C 函数，不过这些扩展包都不可移植，并且不安全。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在 Lua 使用这个函数前，必须注册这个函数。可以用 lua_pushcfunction 来进行注册，这个函数要求传入一个指向 C 函数的指针，它会在 Lua 中创建一个“函数”类型的值，该值就表示这个 C 函数。当注册完后，这个 C 函数就具有了与其他 Lua 函数一样的行为。

为了方便测试 l_sin，可以将 l_sin 的代码直接放入文件 lua.c 中，并将下列内容添加到 luaL_openlibs 调用后面：

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

第一行压入一个函数类型的值，第二行将这个值赋予全局变量 mysin。修改完后，需要重新编译 Lua 的执行程序。然后便可以在 Lua 程序中使用这个新函数 mysin 了。[①]

一个更专业的正弦函数还应该检查参数的类型。辅助库中的 luaL_checknumber 可以检查某个参数是否为一个数字。如果不是，它会扔出一个错误消息；反之，就返回这个数字。对上面这个正弦函数所作的修改很小，如下所示：

```
static int l_sin (lua_State *L) {
  double d = luaL_checknumber(L, 1);
  lua_pushnumber(L, sin(d));
  return 1;  /* 结果的数量 */
}
```

修改后，若调用 mysin('a')，就会得到消息：

```
bad argument #1 to 'mysin' (number expected, got string)
```

luaL_checknumber 会自动填充消息中的参数序号（#1）、函数名（“Mysin”）、期望的参数类型（number）和实际的参数类型（string）。

下面是一个更复杂的示例，这个函数可以返回指定目录下所有的子目录。Lua 在其标准库中没有提供这样的函数，这是因为 ANSI C 中没有目录访问方面的函数。但可以假设拥有一套 POSIX 兼容的系统。这个函数从其字符串参数中获取目录路径，并返回一个数组，记录所有的子目录。例如，调用 dir("/home/lua")会返回 table {".", "..", "src", "bin", "lib"}。若发生错误，函数返回 nil 以及一个包含错误消息的字符串。以下是该函数的完整代码：

```
#include <dirent.h>
#include <errno.h>

static int l_dir (lua_State *L) {
  DIR *dir;
  struct dirent *entry;
  int i;
```

① 在下一节中，会介绍一种更好的方法来将 C 函数链接到 Lua。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  const char *path = luaL_checkstring(L, 1);

  /* 打开目录 */
  dir = opendir(path);
  if (dir == NULL) {  /* 打开目录错误? */
    lua_pushnil(L);  /* 返回nil */
    lua_pushstring(L, strerror(errno));  /* 以及错误消息 */
    return 2;  /* 结果数量 */
  }

/* 创建结果table */
  lua_newtable(L);
  i = 1;
  while ((entry = readdir(dir)) != NULL) {
    lua_pushnumber(L, i++);  /* 压入key */
    lua_pushstring(L, entry->d_name);  /* 压入value */
    lua_settable(L, -3);
  }

closedir(dir);
  return 1;  /* table已位于栈顶 */
}
```

注意，其中用到的 luaL_checkstring 类似于 luaL_checknumber，也来自于辅助库，可用于检查字符串。

在特殊情况中，以上这个 l_dir 实现可能会造成一些较小的内存泄漏。它所调用的 3 个 Lua 函数 lua_newtable、lua_pushstring 和 lua_settable 会由于内存不足而失败。如果这些函数失败了，它们就会引发（raise）一个错误，并中断 l_dir 的执行。因此也就不会调用 closedir 了。就像之前所说的，许多程序可以解决这个问题。如果发生内存不足，一个程序所能做的最佳处理就是结束运行。在第 29 章中，会介绍该目录函数的另一种实现，在那种实现中可以避免这个问题。

26.2　C　模　块

Lua 模块是一个程序块（chunk），其中定义了一些 Lua 函数，这些函数通常存储为 table 的条目。一个为 Lua 编写的 C 模块可以模仿这种行为。除了 C 函数的定义外，C 模块还必须定义一个特殊的函数，这个函数相当于一个 Lua 模块中的主程序块。它应该注册模块中所有的 C 函数，并将它们存储在一个适当的地方。并且，这个函数还应初始化模块中所有需要初始化的东西。

Lua 通过这个注册过程记录下 C 函数，然后使用这些函数地址直接调用它。也就是说，Lua 调用 C 函数时，并不依赖于函数名、包的位置或可见性规则，而只依赖于注册时传入的函数地址。通常，C 模块中只有一个公共（外部）函数，用于创建 C 模块。而其他所有函数都

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

是私有的，在 C 语言中声明为 static。

当用 C 函数扩展 Lua 时，最好将代码设计为一个 C 模块。因为，即使现在只注册一个函数，但说之后可能会需要更多的函数。辅助库为这项工作提供了一个函数 luaL_register，这个函数接收一些 C 函数及其名称，并将这些函数注册到一个与模块同名的 table 中。例如，假设创建一个模块，其中包含了上节定义的 l_dir 函数。首先，必须定义这个模块函数：

```
static int l_dir (lua_State *L) {
  <如前>
}
```

然后，声明一个数组，其中包含模块中所有的函数及名称。这个数组元素的类型为 luaL_Reg 结构，该结构有两个字段，一个字符串和一个函数指针：

```
static const struct luaL_Reg mylib [] = {
  {"dir", l_dir},
  {NULL, NULL}  /* 结尾 */
};
```

在示例中，只声明了一个函数 l_dir。数组的最后一个元素总是{NULL, NULL}，并以此标识结尾。最后，声明一个主函数，其中用到了 luaL_register：

```
int luaopen_mylib (lua_State *L) {
  luaL_register(L, "mylib", mylib);
  return 1;
}
```

luaL_register 根据给定的名称（"mylib"）创建（或复用）一个 table，并用数组 mylib 中的信息填充这个 table。在 luaL_Register 返回时，会将这个 table 留在栈中。最后，luaopen_mylib 函数返回 1，表示将这个 table 返回给 Lua。

当写完 C 模块后，必须将其链接到解释器。如果 Lua 解释器支持动态链接的话，那么最简便的方法是使用动态链接机制。在这种情况中，必须将 C 代码编译成动态链接库[①]，并将这个库放入 C 路径（LUA_CPATH）中。然后，便可以用 require 从 Lua 中加载这个模块：

```
require "mylib"
```

这句调用会将动态库 mylib 链接到 Lua，并会寻找 luaopen_mylib 函数，将其注册为一个 Lua 函数，然后调用它以打开模块。[②]

如果解释器不支持动态链接，那么就必须用新的模块来重新编译 Lua。此外，还需要以某种方式来告诉解释器，它应在打开一个新状态的同时打开这个模块。最简单的做法是，将 luaopen_mylib 加到 luaL_openlibs 会打开的标准库列表中，这个列表在文件 linit.c 中。

① Windows 上为 mylib.dll，其它一些系统上为 mylib.so。
② 这个行为也解释了为什么 luaopen_mylib 必须具有与其他 C 函数相同的原型。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 27 章　编写 C 函数的技术

官方 API 和辅助库都提供了一些机制来帮助编写 C 函数。本章将介绍这些机制，其中包括数组操作、字符串操作，以及如何在 C 语言中保存 Lua 值。

27.1 数组操作

在Lua中，“数组”只是table的一个别名，是指以一种特殊的方法来使用table。像lua_settable和lua_gettable这种操作table的函数，也可用于操作数组。然而，API为数组操作提供了专门的函数。这有两个原因。首先，出于性能考虑，通常会在算法（如排序）中用循环来访问数组。因此，若能提高访问操作的性能，就能提高整个算法的性能。其次是为了方便，像字符串key、整数key是很常用的。

API为数组操作提供了两个函数：

```
void lua_rawgeti (lua_State *L, int index, int key);
void lua_rawseti (lua_State *L, int index, int key);
```

lua_rawgeti和lua_rawseti的参数中涉及到两个索引，index表示table在栈中的位置，key表示元素在table中的位置。假设t为正数，那么调用lua_rawgeti(L, t, key)等价于：

```
lua_pushnumber(L, key);
lua_rawget(L, t);
```

调用lua_rawseti(L, t, key)等价于：

```
lua_pushnumber(L, key);
lua_insert(L, -2);  /* 将'key'放到前一个值的下面 */
lua_rawset(L, t);
```

注意，这两个函数都是原始（raw）操作，比涉及元表的table访问更快。通常，作为数组使用的table很少会用到元表。

接下来是一个具体应用示例，实现了一个变换（map）函数。它对一个数组中的所有元素应用了一个给定的函数，并以每次函数调用的结果来替换现有的数组元素。

```
int l_map (lua_State *L) {
  int i, n;
```

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
/* 第一个参数必须是一个table (t) */
  luaL_checktype(L, 1, LUA_TTABLE);

/* 第二个参数必须是一个函数 (f) */
  luaL_checktype(L, 2, LUA_TFUNCTION);

n = lua_objlen(L, 1);  /* 获取table的大小 */

for (i = 1; i <= n; i++) {
    lua_pushvalue(L, 2);   /* 压入f */
    lua_rawgeti(L, 1, i);  /* 压入t[i] */
    lua_call(L, 1, 1);     /* 调用f(t[i]) */
    lua_rawseti(L, 1, i);  /* t[i] = 结果 */
  }

return 0;  /* 无结果 */
}
```

这个示例还引入了两个新函数。luaL_checktype 函数确保给定参数具有特定的类型，否则它会引发一个错误。lua_call 函数完成一次无保护的调用，它类似于 lua_pcall，不过在发生错误时，它会传播错误，而非返回错误代码。在一个应用程序中编写主函数时，不应使用 lua_call，因为这样需要捕获所有的错误。而编写 C 函数时，通常可以用 lua_call。若有错误发生，只需将错误留下。

27.2 字符串操作

当一个 C 函数从 Lua 收到一个字符串参数时，必须遵守两条规则：不要在访问字符串时从栈中弹出它，不要修改字符串。

当一个 C 函数需要创建一个字符串返回给 Lua 时，C 代码还必须处理字符串缓冲的分配和释放、缓冲溢出等问题。Lua API 也提供了一些函数来帮助完成这些任务。

标准 API 为两种常用的字符串操作提供了支持：提取子串和字符串连接。lua_pushlstring 支持提取子串，它接受一个额外的字符串长度参数。把一个字符串 s 的子串（区间为[i, j]）传递给 Lua 时，只需这么做：

```
lua_pushlstring(L, s + i, j - i + 1);
```

接下来是一个示例。假设，需要编写一个字符串切割函数，它根据一个给定的分隔符来切割一个字符串，并返回一个 table，其中包含了所有切割后的子串。例如，调用 split("hi,ho,there", ",")会返回 table {"hi", "ho", "there"}。下面是这个函数的一个简化实现。它无须额外的缓冲，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

也不限制其所能处理的字符串大小。

```
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);  /* 结果 */

/* 遍历所有分隔符 */
  while ((e = strchr(s, *sep)) != NULL) {
    lua_pushlstring(L, s, e-s);  /* 压入子串 */
    lua_rawseti(L, -2, i++);
    s = e + 1;  /* 跳过分隔符 */
  }

/* 压入最后一个子串 */
  lua_pushstring(L, s);
  lua_rawseti(L, -2, i);

return 1;  /* 返回table */
}
```

为了连接字符串，Lua API 提供了一个叫 lua_concat 的函数。它类似于 Lua 中的“..”操作符。不过，它可以同时连接多个字符串，调用 lua_concat(L, n)连接（并弹出）栈顶的 n 个值，然后压入结果。此外，这个函数会将数字转换为字符串，并在需要时调用元方法。

另一个有用的函数是 lua_pushfstring：

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

这个函数有点类似于 C 函数 sprintf，它们都会根据一个格式字符串和一些额外的参数来创建一个新字符串。但与 sprintf 不同的是，无须提供这个新字符串的缓冲。Lua 会动态地创建一个足够大的缓冲来存放新字符串，确保不会有缓冲溢出等问题。这个函数会将结果字符串压入栈中，并返回一个指向它的指针。当前，这个函数接受的指示符只有：%%（表示字符%）、%s（表示字符串）、%d（表示整数）、%f（表示 Lua 中的数字，即双精度浮点数）及%c（接受一个整数，并将其格式化为一个字符）。除此之外，它不接受任何例如宽度或精度选项。

如果只是连接一些字符串的话，lua_concat 和 lua_pushfstring 就够用了。但如果要连接很多字符串（或字符）的话，像 11.6 节中那样逐个地连接字符串就比较低效了。在此，可以使用辅助库提供的缓冲机制，这套机制包含了两个层面的缓冲。第一层类似于 I/O 操作中的缓冲，就是在一个本地缓冲中收集较小的字符串，并在本地缓冲填满后将结果传递给 Lua（通过 lua_pushlstring）。第二层是使用 lua_concat 或其他的栈算法（如 11.6 节）来连接多次缓冲填满

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

后的结果。

为了更详细地描述辅助库的缓冲机制，下面来看一个简单的应用示例。以下代码展示了string.upper 函数的实现，它位于源代码文件 lstrlib.c 中：

```
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_addchar(&b, toupper((unsigned char)(s[i])));
  luaL_pushresult(&b);
  return 1;
}
```

使用缓冲机制的第一步是声明一个 luaL_Buffer 变量，并用 luaL_buffinit 来初始化它。在初始化后，这个变量中就会保留一份状态 L 的副本，由此再调用其他操作缓冲的函数时，就无须传递状态参数了。宏 luaL_addchar 会将一个字符放入缓冲。辅助库还提供了将字符串放入缓冲的函数，对于具有显式长度的字符串有 luaL_addlstring，而对于“0 结尾”的字符串有 luaL_addstring。最后，luaL_pushresult 会更新缓冲，并将最终的字符串留在栈顶。上述函数的原型为：

```
void luaL_buffinit   (lua_State *L, luaL_Buffer *B);
void luaL_addchar    (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);
```

通过这些函数，就无须再关心缓冲的分配、溢出等细节了。此外，这种连接算法也非常高效。用 str_upper 函数来处理大型的字符串（如大约 1MB）也不会有什么问题。

在使用辅助库的缓冲机制时，必须注意一个细节。当向缓冲中添加东西时，缓冲会将一些中间结果放到栈中。因此，不应假设栈顶还是和使用缓冲前一样。此外，虽然可以在使用缓冲的过程中将栈用于其他任务（甚至是创建另一个缓冲），但这些任务所调用的压入和弹出的次数必须相等。这项限制有些严格，例如将一个 Lua 返回的字符串放入缓冲。在这种情况中，既不能在字符串未加入缓冲前弹出它（因为无法在弹出字符串后继续使用它），又不能在弹出字符串前将它加入缓冲（否则栈的层数就会不正确）。

由于这是一种很常见的情况，辅助库提供了一个专门的函数来将栈顶的值加入缓冲：

```
void luaL_addvalue (luaL_Buffer *B);
```

如果栈顶的值不是字符串或数字的话，调用这个函数就会是一个错误。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

27.3 在 C 函数中保存状态

通常，C 函数需要保存一些非局部的数据，这些数据的生存时间会比 C 函数的执行更久。在 C 语言中，通常用全局变量或静态变量来达到这个目的。然而在为 Lua 编写库函数时，使用全局变量或静态变量却不是一个好方法。这是因为，首先，无法在一个 C 变量中保存普通的 Lua 对象。其次，若一个库使用了全局变量或静态变量，那它也就无法用于多个 Lua 状态了。

对于一个 Lua 函数来说，有 3 种地方可以存放非局部的数据，它们是全局变量、函数环境和非局部的变量（closure 中）。而 C API 也提供了 3 种地方来保存这类数据：注册表、环境和 upvalue。

注册表是一个全局的 table，它只能被 C 代码访问。通常，可以用它来保存那种需要在几个模块中共享的数据。如果需要保存一个模块的私有数据，那么应该使用环境。与 Lua 函数一样，每个 C 函数都有自己的环境 table[①]。通常，一个模块内的所有函数共享同一个环境 table，由此它们可以共享数据。最后，C 函数也可以拥有 upvalue，upvalue 是一种与特定函数相关联的 Lua 值。

27.3.1 注册表（registry）

注册表总是位于一个“伪索引（Pseudo-Index）”上，这个索引值由 LUA_REGISTRYINDEX 定义。伪索引就像是一个栈中的索引，但它所关联的值不在栈中。Lua API 中的大多数函数都能接受伪索引，但像 lua_remove 和 lua_insert 这种操作栈本身的函数却只能使用普通索引。例如，为了获取注册表中 key 为"Key"的值，可以这么做：

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

注册表是一个普通的 Lua table，可以用任何 Lua 值（除了 nil）来索引它。然而，由于所有的 C 模块共享同一个注册表，为了避免使用冲突，必须谨慎地选择 key 的值。如果允许其他库来访问数据，那么用字符串作为 key 会比较合适。因为这些库只需知道 key 的名字就可以了。对于 key 名字的选择，没有一种可以绝对避免冲突的方法。但有一些好的建议，例如，避免常用的名字，用库名或公司名作为 key 名字的前缀。尽量不使用像 lua 或 lualib 这样的前缀。避免冲突的另一种选择是使用“通用唯一标识符（universal unique identifier，uuid）[②]”，现在大多数系统中都有生成 uuid 的程序（如 Linux 中的 uuidgen）。

① 这是 Lua 5.1 的新功能。

② 一个 uuid 就是一个 128 位的数字，是由主机的 MAC 地址、一个时间戳和一个随机数组成生成的。这样可以使一个 uuid 不同于其他的 uuid。uuid 通常以十六进制字符串的形式来书写。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在注册表中不应使用数字类型的 key，因为这种 key 是被“引用系统”所保留的。这个系统是由辅助库中的一系列函数组成的，它可以在向一个 table 存储 value 时，忽略如何创建一个唯一的 key。以下调用：

```
int r = luaL_ref(L, LUA_REGISTRYINDEX);
```

会从栈中弹出一个值，然后用一个新分配的整数 key 来将这个值保存到注册表中，最后返回这个整数 key。这个 key 被称为“引用”。

当需要在一个 C 变量中保存一个指向 Lua 值的引用时，就要用到“引用系统”。因为，不要在一个检索 Lua 字符串的 C 函数之外继续持有指向 Lua 字符串的指针。此外，Lua 也没有提供任何指向 table 或函数的指针。因此，无法通过指针来使用 Lua 对象。需要这种指针功能时，可以创建一个引用，并可以将这个引用保存在 C 中。

为了将与引用 r 关联的值压入栈中，可以这么写：

```
lua_rawgeti(L, LUA_REGISTRYINDEX, r);
```

最后，释放该值和引用，可以这么写：

```
luaL_unref(L, LUA_REGISTRYINDEX, r);
```

在这句调用后，再调用 luaL_ref 会返回相同的引用。

引用系统将 nil 视为一种特殊情况。为一个 nil 值调用 luaL_ref 时，并不会创建新引用，而是返回一个常量引用 LUA_REFNIL。以下调用：

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

是没有什么效果的，然而

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

会压入一个 nil。

引用系统还定义了一个常量 LUA_NOREF，这是一个不同于其他合法引用的整数。它可用于表示无效的引用。就像 LUA_REFNIL 一样，用 LUA_NOREF 调用上述函数会返回 nil，释放 LUA_NOREF 也不会有什么效果。

另一种创建注册表 key 的方法是，使用代码中静态变量的地址，C 链接器可以确保这种 key 在所有库中的唯一性。为了实现这种方法，需要用到函数 lua_pushlightuserdata，这个函数会在 Lua 栈中压入一个表示 C 指针的值。下面代码演示了如何通过这种方法在注册表中保存一个字符串，并检索它：

```
/* 具有唯一地址的变量 */
static char Key = 'k';
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
/* 保存一个字符串 */
lua_pushlightuserdata(L, (void *)&Key);  /* 压入地址 */
lua_pushstring(L, myStr);  /* 压入值 */
lua_settable(L, LUA_REGISTRYINDEX);  /* registry[&Key] = myStr */

/* 检索一个字符串 */
lua_pushlightuserdata(L, (void *)&Key);  /* 压入地址 */
lua_gettable(L, LUA_REGISTRYINDEX);  /* 检索值 */
myStr = lua_tostring(L, -1);  /* 转换成字符串 */
```

在 28.5 节中会详细讨论轻量级 userdata（light userdata）。

27.3.2 C 函数的环境

从 Lua 5.1 开始，在 Lua 中注册的所有 C 函数都有自己的环境 table。一个函数可以像访问注册表那样，通过一个伪索引来访问它的环境 table。环境 table 的伪索引是 LUA_ENVIRONINDEX。

通常，使用这些环境的方法与在 Lua 模块中使用环境的方法差不多。就是先为模块创建一个新的 table，然后使模块中的所有函数都共享这个 table。在 C 语言中设置这种共享环境的做法与在 Lua 中设置环境的做法类似。就是修改主程序块的环境，然后所有新建的函数都会自动地套用这个新环境。在 C 语言中，设置环境的代码如下：

```
int luaopen_foo (lua_State *L) {
  lua_newtable(L);
  lua_replace(L, LUA_ENVIRONINDEX);
  luaL_register(L, <库名>, <函数列表>);
  ...
}
```

这个初始化函数 luaopen_foo 创建了一个新的 table，用于共享的环境。它使用 lua_replace 将这个 table 设为当前环境，然后调用 luaL_register 时，所有新建的函数都会继承当前环境。

尽可能使用环境来代替注册表，除非需要在不同模块间共享数据。进一步说，可以在环境 table 中使用引用系统，而由此创建的引用也只对本模块可见。

27.3.3 upvalue

注册表提供了全局变量的存储，环境提供了模块变量的存储，而 upvalue 机制则实现了一种类似于 C 语言中静态变量的机制，这种变量只在一个特定的函数中可见。每当在 Lua 中创建一个函数时，都可以将任意数量的 upvalue 与这个函数相关联。每个 upvalue 都可以保存一个 Lua 值。以后，在调用这个函数时，就可以通过伪索引来访问这些 upvalue 了。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

将这种 C 函数与 upvalue 的关联称为 closure。一个 C closure 类似于 Lua closure。closure 可以用同一个函数代码来创建多个 closure，每个 closure 可以拥有不同的 upvalue。

接下来是一个简单的示例，在 C 语言中创建一个 newCounter 函数[1]。这个函数是一个工厂函数，每次调用都返回一个新的账户函数。虽然所有的账户函数共享相同的 C 代码，但每个函数却可持有一个独立的账户。这个工厂函数如下：

```
static int counter (lua_State *L);  /* 前向声明 */

int newCounter (lua_State *L) {
  lua_pushinteger(L, 0);
  lua_pushcclosure(L, &counter, 1);
  return 1;
}
```

这里的关键函数是 lua_pushclosure，它会创建一个新的 closure。其中第二个参数是一个基础函数(示例中为 counter)，第三个参数是 upvalue 的数量(示例中为 1)。在创建一个新的 closure 前，必须将 upvalue 的初值压入栈中。在此示例中，压入了数字 0 作为 upvalue 的初值。最后，lua_pushcclosure 将一个新的 closure 留在了栈上，并以此作为 newCounter 的返回值。

现在来看一下 counter 的定义：

```
static int counter (lua_State *L) {
  int val = lua_tointeger(L, lua_upvalueindex(1));
  lua_pushinteger(L, ++val);  /* 新的值 */
  lua_pushvalue(L, -1);  /* 复制它 */
  lua_replace(L, lua_upvalueindex(1));  /* 更新upvalue */
  return 1;  /* 返回新的值 */
}
```

其中，关键函数 lua_upvalueindex（其实是一个宏）可以生成一个 upvalue 的伪索引。注意，这个伪索引可以像其他栈索引一样使用，唯一不同的是它不存在于栈中。表达式 lua_upvalueindex(1)表示此函数的第一个 upvalue 的索引。上例调用 lua_tointeger 来检索第一个（也是唯一的一个）upvalue 的整数值。然后压入新值++val，并复制了一份。接着使用这份副本替换了 upvalue 的值。最后，返回了另一个副本。

接下来是一个更高级的示例，通过 upvalue 来实现元组（tuple）。元组是一种具有匿名字段的常量记录。可以用一个数字索引来检索某个字段，或者一次性检索所有字段。在实现中，将元组表示为函数，元组的值存储在函数的 upvalue 中。当用一个数字参数来调用此函数时，它会返回指定的字段。当不用参数来调用此函数时，则返回所有的字段。以下代码演示了元组的使用：

① 在 6.1 节中，已在 Lua 中定义过同样的函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
x = tuple.new(10, "hi", {}, 3)
print(x(1))     --> 10
print(x(2))     --> hi
print(x())      --> 10  hi  table: 0x8087878  3
```

在 C 语言中，用同一个函数 t_tuple 来表示所有的元组，代码如下所示：

```
int t_tuple (lua_State *L) {
  int op = luaL_optint(L, 1, 0);
  if (op == 0) {  /* 无参数? */
    int i;
    /* 将所有合法upvalue压入栈中 */
    for (i = 1; !lua_isnone(L, lua_upvalueindex(i)); i++)
      lua_pushvalue(L, lua_upvalueindex(i));
    return i - 1;  /* 栈中值的数量 */
  }
  else {  /* 获取'op'字段 */
    luaL_argcheck(L, 0 < op, 1, "index out of range");
    if (lua_isnone(L, lua_upvalueindex(op)))
      return 0;  /* 无此字段 */
    lua_pushvalue(L, lua_upvalueindex(op));
    return 1;
  }
}

int t_new (lua_State *L) {
  lua_pushcclosure(L, t_tuple, lua_gettop(L));
  return 1;
}

static const struct luaL_Reg tuplelib [] = {
  {"new", t_new},
  {NULL, NULL}
};

int luaopen_tuple (lua_State *L) {
  luaL_register(L, "tuple", tuplelib);
  return 1;
}
```

由于可以使用或不使用参数来调用一个元组，所示 t_tuple 用 luaL_optint 来获取这个可选参数。luaL_optint 函数类似于 luaL_checkint，但它允许参数不存在。若不存在，则返回一个指定的默认值（本例中为 0）。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

索引一个不存在的 upvalue，结果是一个类型为 LUA_TNONE 的伪值（pseudo-value）。当访问的索引超出了当前栈的范围，也会得到这样一个类型为 LUA_TNONE 的伪值。因此，t_tuple 函数用 lua_isnone 来测试一个 upvalue 是否存在。另外，不应该用负数来调用 lua_upvalueindex。所以，必须对用户提供的索引进行检查。luaL_argcheck 函数就是用来检查这种情况的，如果发现索引是负数，则引发一个错误。

t_new 是创建元组的函数，而且它很简单。由于它的参数已经在栈中，所以只需将这些参数作为 upvalue，并调用 lua_pushcclosure 来创建一个基于 t_tuple 的 closure 即可。最后，数组 tuplelib 和函数 luaopen_tuple 是创建库的标准代码，这个 tuple 库中只有一个函数 new。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 28 章　用户自定义类型

上一章介绍了如何通过 C 语言编写新函数来扩展 Lua。本章将介绍如何用 C 语言编写新的类型来扩展 Lua。下面将从一个小示例入手，使用元表和其他机制来扩展它。

这个示例实现了一种很简单的类型——布尔数组。选用这个示例是因为它不涉及到复杂的算法，从而可以使读者专注于 API 的问题。不过，这个示例本身还是具有实用价值的。当然，可以在 Lua 中用 table 来实现布尔数组。但 C 语言实现可以将每个布尔值存储在一个 bit 中，从而将内存用量减少到不足 table 方法的 3%。

这个实现需要以下定义：

```
#include <limits.h>

#define BITS_PER_WORD (CHAR_BIT*sizeof(unsigned int))
#define I_WORD(i)     ((unsigned int)(i) / BITS_PER_WORD)
#define I_BIT(i)      (1 << ((unsigned int)(i) % BITS_PER_WORD))
```

BITS_PER_WORD 是一个无符号整型的 bit 数量。宏 I_WORD 根据给定的索引来计算对应的 bit 位所存放的 word（字），I_BIT 计算出一个掩码，用于访问这个 word 中的正确 bit。

可以使用以下结构来表示数组：

```
typedef struct NumArray {
  int size;
  unsigned int values[1];  /* 可变部分 */
} NumArray
```

这里。由于 C 89 标准不允许分配 0 长度的数组，所以声明了数组 values 需要有一个元素来作为占位符。接下来会在分配数组时定义实际的大小。下面这个表达式可以计算出具有 *n* 个元素的数组大小：

```
sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int)
```

注意，这里无须对 I_WORD 加 1，因为原来的结构中已经包含了一个元素的空间。

28.1　userdata

首先要面临的问题是如何在 Lua 中表示这个 NumArray 结构。Lua 为此提供了一种基本类

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

型 userdata。userdata 提供了一块原始的内存区域，可以用来存储任何东西。并且，在 Lua 中 userdata 没有任何预定义的操作。

函数 lua_newuserdata 会根据指定的大小分配一块内存，并将对应的 userdata 压入栈中，最后返回这个内存块的地址：

```
void *lua_newuserdata (lua_State *L, size_t size);
```

如果由于某些原因，需要通过其他机制来分配内存。那么可以创建只有一个指针大小的 userdata，然后将指向真正内存块的指针存入其中。在下一章中就有这样的例子。

以下函数就用 lua_newuserdata 创建了一个新的布尔数组：

```
static int newarray (lua_State *L) {
  int i, n;
  size_t nbytes;
  NumArray *a;

n = luaL_checkint(L, 1);
  luaL_argcheck(L, n >= 1, 1, "invalid size");
  nbytes = sizeof(NumArray) + I_WORD(n - 1)*sizeof(unsigned int);
  a = (NumArray *)lua_newuserdata(L, nbytes);

a->size = n;
  for (i=0; i <= I_WORD(n-1); i++)
    a->values[i] = 0;  /* 初始化数组 */

return 1;  /* 新的userdata 已在栈上 */
}
```

其中，宏 luaL_checkint 只是在调用 luaL_checkinteger 后进行了一个类型转换。只要在 Lua 中注册好 newarray，就可以通过语句 a = array.new(1000)来创建一个新数组。

可以通过这样的调用 array.set(array, index, value)，在数组中存储元素。后面的内容会介绍如何使用元表来实现更传统的语法 array[index] = value。无论哪种写法，底层函数都是相同的。下面将遵循 Lua 惯例，假设索引从 1 开始。

```
static int setarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2) - 1;
  luaL_checkany(L, 3);

luaL_argcheck(L, a != NULL, 1, "'array' expected");

luaL_argcheck(L, 0 <= index && index < a->size, 2,
              "index out of range");
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
if (lua_toboolean(L, 3))
   a->values[I_WORD(index)] |= I_BIT(index);  /* 设置bit */
 else
   a->values[I_WORD(index)] &= ~I_BIT(index);  /* 重置bit */
 return 0;
}
```

由于 Lua 中任何值都可以转换为布尔，所以这里对第 3 个参数使用 luaL_checkany，它只确保了在这个参数位置上有一个值。如果用错误的参数调用了 setarray，就会得到这样的错误消息：

```
array.set(0, 11, 0)
  --> stdin:1: bad argument #1 to 'set' ('array' expected)
array.set(a, 0)
  --> stdin:1: bad argument #3 to 'set' (value expected)
```

下一个函数用于检索元素：

```
static int getarray (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  int index = luaL_checkint(L, 2) - 1;

  luaL_argcheck(L, a != NULL, 1, "'array' expected");

  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                "index out of range");

  lua_pushboolean(L, a->values[I_WORD(index)] & I_BIT(index));
  return 1;
}
```

下面还定义了一个函数用于检索一个数组的大小：

```
static int getsize (lua_State *L) {
  NumArray *a = (NumArray *)lua_touserdata(L, 1);
  luaL_argcheck(L, a != NULL, 1, "'array' expected");
  lua_pushinteger(L, a->size);
  return 1;
}
```

最后，需要一些代码来初始化这个库：

```
static const struct luaL_Reg arraylib [] = {
  {"new", newarray},
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_register(L, "array", arraylib);
  return 1;
}
```

同样其中用到了辅助库的 luaL_register，它会根据给定的名称（本例中为“array”）创建一个 table，并用数组 arraylib 中指定的名称/函数对来填充它。

在打开库后，就可以在 Lua 中使用这个新类型了：

```
a = array.new(1000)
print(a)                  --> userdata: 0x8064d48
print(array.size(a))      --> 1000
for i=1,1000 do
  array.set(a, i, i%5 == 0)
end
print(array.get(a, 10))  --> true
```

28.2 元　　表

当前的实现有一个重大的安全漏洞，假定用户写了这样的语句 array.set(io.stdin, 1, false)。io.stdin 的值是一个 userdata，是一个文件流指针（FILE *）。由于这是一个 userdata，array.set 会认为它是一个合法的参数，结果就使内存遭到破坏（如果幸运的话，可能会得到一个索引超出范围的错误）。但对于有些 Lua 库来说，这种行为是不可接受的。问题的原因不在于如何使用一个 C 程序库，而在于程序库不应破坏 C 数据或在 Lua 中导致核心转储（Core Dump）。

一种辨别不同类型的 userdata 的方法是，为每种类型创建一个唯一的元表。每当创建了一个 userdata 后，就用相应的元表来标记它。而每当得到一个 userdata 后，就检查它是否拥有正确的元表。由于 Lua 代码不能改变 userdata 的元表，因此也就无法欺骗代码了。

另外还需要有个地方来存储这个新的元表，然后才能用它来创建新的 userdata，并检查给定的 userdata 是否具有正确的类型。在前面已提到过，有三个候选地可用于存储元表：注册表、环境或程序库中函数的 upvalue。在 Lua 中，通常习惯是将所有新的 C 类型注册到注册表中，以一个类型名作为 key，元表作为 value。由于注册表中还有其他的内容，所以必须小心地选择类型名，以避免与 key 冲突。在示例中，将使用"LuaBook.array"作为其新类型的名称。

通常，辅助库中提供了一些函数来帮助实现这些内容。可以使用的辅助库函数有：

```
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);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

luaL_newmetatable 函数会创建一个新的 table 用作元表，并将其压入栈顶，然后将这个 table 与注册表中的指定名称关联起来。luaL_getmetatable 函数可以在注册表中检索与 tname 关联的元表。luaL_checkudata 可以检查栈中指定位置上是否为一个 userdata，并且是否具有与给定名称相匹配的元表。如果该对象不是一个 userdata，或者它不具有正确的元表，就会引发一个错误；否则，它就返回这个 userdata 的地址。

现在可以修改前面的实现了。第一步是修改打开库的函数。新版本必须为数组创建一个元表：

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_register(L, "array", arraylib);
  return 1;
}
```

下一步是修改 newarray，使其能为所有新建的数组设置这个元表：

```
static int newarray (lua_State *L) {
  <如前>

  luaL_getmetatable(L, "LuaBook.array");
  lua_setmetatable(L, -2);

  return 1;  /* 新的userdata 已在栈中 */
}
```

lua_setmetatable 函数会从栈中弹出一个 table，并将其设为指定索引上对象的元表。在本例中，这个对象就是一个新建的 userdata。

最后，setarray、getarray 和 getsize 必须检查其第一个参数是否为一个合法的数组。为了简化这样的任务，定义了以下宏：

```
#define checkarray(L) \
        (NumArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

使用了这个宏 getsize 同样简单：

```
static int getsize (lua_State *L) {
  NumArray *a = checkarray(L);
  lua_pushinteger(L, a->size);
  return 1;
}
```

由于 setarray 和 getarray 使用了一段相同的代码来检查第二个索引参数，所以将这段公共部分单独组成以下函数：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
static unsigned int *getindex (lua_State *L,
                               unsigned int *mask)
{
  NumArray *a = checkarray(L);
  int index = luaL_checkint(L, 2) - 1;

  luaL_argcheck(L, 0 <= index && index < a->size, 2,
                "index out of range");

  /* 返回元素地址 */
  *mask = I_BIT(index);
  return &a->values[I_WORD(index)];
}
```

使用了 getindex 的 setarray 和 getarray 也同样简单明了：

```
static int setarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getindex(L, &mask);
  luaL_checkany(L, 3);
  if (lua_toboolean(L, 3))
    *entry |= mask;
else
   *entry &= ~mask;

return 0;
}

static int getarray (lua_State *L) {
  unsigned int mask;
  unsigned int *entry = getindex(L, &mask);
  lua_pushboolean(L, *entry & mask);
  return 1;
}
```

现在，如果试图这样调用 array.get(io.stdin, 10)，就会得到一个正确的错误消息：

```
error: bad argument #1 to 'get' ('array' expected)
```

28.3 面向对象的访问

下一步是将这种新类型变换成一个对象，然后就可以用普通的面向对象语法来操作它的实例了。例如：

```
a = array.new(1000)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
print(a:size())     --> 1000
a:set(10, true)
print(a:get(10))    --> true
```

注意，a:size()等价于 a.size(a)。因此，必须使表达式 a.size 返回前面定义的函数 getsize。实现这点的关键是使用--index 元方法。对于 table 而言，Lua 会在找不到指定 key 时调用这个元方法。对于 userdata，则会在每次访问时都调用它，因为 userdata 根本没有 key。

假设，运行了以下代码：

```
local metaarray = getmetatable(array.new(1))
metaarray.--index = metaarray
metaarray.set = array.set
metaarray.get = array.get
metaarray.size = array.size
```

第一行创建了一个数组，并将它的元表赋予了 metaarray。然后将 metaarray.--index 设为 metaarray。当对 a.size 求值时，由于 a 是一个 userdata，所以 Lua 无法在对象 a 中找到 key "size"。因此，Lua 会尝试通过 a 的元表的--index 字段来查找这个值，而这个字段也就是 metaarray 自身。由于 metaarray.size 为 array.size，因此 a.size(a)的结果就是 array.size(a)。

其实，在 C 中也可以达到相同的效果，甚至还可以做得更好。现在的数组是一种具有操作的对象，可以无须在 table array 中保存这些操作。程序库只要导出一个用于创建新数组的函数 new 就可以了，所有其他操作都可作为对象的方法。C 代码同样可以直接注册这些方法。

操作 getsize、getarray 和 setarray 无须作任何改变，唯一需要改变的是注册它们的方式。现在，需要修改打开程序库的函数。首先，需要设置两个独立的函数列表，一个用于常规的函数，另一个用于方法：

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
  {"set", setarray},
  {"get", getarray},
  {"size", getsize},
  {NULL, NULL}
};
```

新的打开函数 luaopen_array 必须创建元表，并将它赋予--index 字段，然后在元表中注册所有的方法，最后创建并填充 array table：

```
int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  /* 元表.--index =元表 */
  lua_pushvalue(L, -1);  /* 复制元表 */
  lua_setfield(L, -2, "--index");

  luaL_register(L, NULL, arraylib_m);

  luaL_register(L, "array", arraylib_f);
  return 1;
}
```

其中用到了 luaL_register 的另一个特性。在第一次调用中，以 NULL 作为库名，luaL_register 不会创建任何用于存储函数的 table，而是以栈顶的 table 作为存储函数的 table。在本例中，栈顶 table 就是元表本身，因此 luaL_register 会将所有的方法放入其中。第二次调用 luaL_register 则提供了一个库名，它就根据此名（array）创建了一个新 table，并将指定的函数注册在这个 table 中（也就是本例中唯一的 new 函数）。

最后，给这个数组类型添加一个--tostring 方法。使 print(a)可以打印出“array”以及数组的大小，就像“array(1000)”。这个函数如下：

```
int array2string (lua_State *L) {
  NumArray *a = checkarray(L);
  lua_pushfstring(L, "array(%d)", a->size);
  return 1;
}
```

lua_pushfstring 可以在栈顶创建并格式化一个字符串。另外，还需要将 array2string 加到列表 arraylib_m 中，从而将这个函数加入数组对象的元表。

```
static const struct luaL_Reg arraylib_m [] = {
  {"--tostring", array2string},
  <其他方法>
};
```

28.4 数组访问

另一种面向对象写法是使用常规的数组访问写法。相对于写 a:get(i)，可以简单地写为 a[i]。对于上面的示例，很容易可以做到这点。由于函数 setarray 和 getarray 所接受的参数次序暗合相应元方法的参数次序，因此在 Lua 代码中可以快速地将这些元方法定义为：

```
local metaarray = getmetatable(array.new(1))
metaarray.--index = array.get
metaarray.--newindex = array.set
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
metaarray.--len = array.size
```

必须在第一个数组实现上运行这段代码，而不能应用于那个为面向对象访问而修改的版本[①]。使用这些标准语法很简单：

```
a = array.new(1000)
a[10] = true          -- setarray
print(a[10])          -- getarray  --> true
print(#a)             -- getsize   --> 1000
```

如果还要更完美，可以在 C 代码中注册这些方法。为此，需要再次修改初始化函数：

```
static const struct luaL_Reg arraylib_f [] = {
  {"new", newarray},
  {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
  {"--newindex", setarray},
  {"--index", getarray},
  {"--len", getsize},
  {"--tostring", array2string},
  {NULL, NULL}
};

int luaopen_array (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.array");
  luaL_register(L, NULL, arraylib_m);
  luaL_register(L, "array", arraylib_f);
  return 1;
}
```

在这个新版本中，仍只有一个公共函数 new。所有其他函数都作为特定操作的元方法。

28.5 轻量级 userdata（light userdata）

到现在为止所使用的 userdata 都称为“完全 userdata（full userdata）”。Lua 还提供另一种“轻量级 userdata（light userdata）”。

轻量级 userdata 是一种表示 C 指针的值（即 void*）。由于它是一个值，所以不用创建它。要将一个轻量级 userdata 放入栈中，只需调用 lua_pushlightuserdata 即可：

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

① 译者注：因为在 Lua 代码中无法访问 userdata 的元表。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

尽管两种 userdata 在名称上差不多，但它们之间还是存在很大不同的。轻量级 userdata 不是缓冲，只是一个指针而已。它也没有元表，就像数字一样，轻量级 userdata 无须受垃圾收集器的管理。

有时会将轻量级 userdata 当作一种廉价的完全 userdata 来使用。但这种用法并没有太大意义。首先，使用轻量级 userdata 时用户必须自己管理内存，因为轻量级 userdata 不属于垃圾收集的范畴。其次，不要被“完全”二字所迷惑，完全 userdata 的开销并不比轻量级 userdata 大多少。它们只为分配内存增加了一些 malloc 的开销。

轻量级 userdata 的真正用途是相等性判断。一个完全 userdata 是一个对象，它只与自身相等。而一个轻量级 userdata 则表示了一个 C 指针的值。因此，它与所有表示同一个指针的轻量级 userdata 相等。可以将轻量级 userdata 用于查找 Lua 中的 C 对象。

以下是一种比较典型的情况，假设正在实现一种 Lua 与某个窗口系统的绑定。在这种绑定中，用完全 userdata 表示窗口。每个 userdata 可以包含整个窗口的数据结构，也可以只包含一个指向系统所创建窗口的指针。当在一个窗口中发生了一个事件时（例如单击鼠标），系统要调用对应于该窗口的回调函数。而窗口是通过其地址来识别的。为了调用 Lua 中实现的回调函数，必须先找到表示指定窗口的 userdata。若要寻找这个 userdata，可以用一个 table 来保存窗口的信息，它的 key 是表示窗口地址的轻量级 userdata，而 value 则是表示窗口本身的完全 userdata。当得到一个窗口地址时，就可以把它作为一个轻量级 userdata 压入栈中，并用这个 userdata 来索引 table[1]。从而得到那个表示窗口本身的完全 userdata，并由此调用回调函数。

① 这个 table 还应具有弱引用的 value，否则那些完全 userdata 就永远无法回收。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第29章 管理资源

在上一章中，实现了布尔数组时无须关心资源的管理。那些数组只需要内存。每个表示数组的 userdata 都具有各自的内存，而这些内存是由 Lua 来管理的。当一个数组成为垃圾时[①]，Lua 最终会收集它，并释放其占用的内存。

有时一个对象需要使用原始内存之外的其他资源，例如文件描述符、窗口句柄及其他类似的东西[②]。在这种情况中，当一个对象成为垃圾被收集后，这些类型的资源也都必须被释放。一些面向对象的语言提供了一种称为“终结函数（finalizer）”的特殊机制来达到这点。Lua 通过元方法--gc 来指定终结函数。这个元方法只能对 userdata 值有效。在回收一个 userdata 时，如果它的元表中有一个--gc 字段，Lua 就会调用这个字段的值（应该是一个函数），并以这个 userdata 自身作为参数传入。然后，这个函数便可以释放与此 userdata 相关联的资源了。

为了整体地演示这个元方法和 API 的使用，本章将开发两个使用外部功能的示例。第一个示例实现了另一种遍历目录的函数。第二个示例则更贴近实际，它使用开源代码 Expat，实现了一个 XML 解释器。

29.1 目录迭代器

在 26.1 节中，实现了一个 dir 函数，它返回一个 table，包含了指定目录下所有的文件。新实现会返回一个迭代器，每次调用这个迭代器，它便返回一个新条目。有了这种实现后，便可以用以下循环来遍历一个目录了：

```
for fname in dir(".") do print(fname) end
```

为了在 C 语言中遍历一个目录，需要一个 DIR 结构。函数 opendir 可以创建一个 DIR 的示例，而这个实例最终必须要调用 closedir 来显式地释放。上一个 dir 的实现将它的 DIR 示例作为一个局部变量，并在检索完最后一个文件名后释放了它。而在新实现中，无法将这个 DIR 实例保存到局部变量中，因为它必须要在几次调用中用到这个值。此外，它不能在检索完最后一个文件名后释放它。因为如果程序中断了循环，迭代器就永远不会检索到最后一个名称。因此，为了确保 DIR 实例能被正确释放，可以将它的地址存入一个 userdata，并使用这个 userdata

① 就是当程序无法再访问到它时。
② 通常这些资源也是一种内存，但由系统的其他部分所管理。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

的--gc 元方法来释放这个目录结构。

尽管 userdata 是这次实现中的主要角色，Lua 却无须直接访问到这个表示目录的 userdata。dir 函数会返回一个迭代器函数，这才是 Lua 直接使用的东西。目录 userdata 可以作为迭代器函数的一个 upvalue。这样迭代器函数就能直接访问到这个结构了，而 Lua 代码则不能（也没有必要）。

总的来说，需要 3 个 C 函数。首先是这个 dir 函数，它是一个工厂，Lua 调用它来创建迭代器。它还必须打开一个 DIR 结构，并将其作为迭代器函数的 upvalue。其次，需要这个迭代器函数。最后是--gc 元方法，用于关闭 DIR 结构。通常，还需要一个额外的函数，用于进行一些初始化工作，例如为目录创建一个元表并初始化元表。

先介绍 dir 函数的代码，如下：

```
#include <dirent.h>
#include <errno.h>

/* 迭代器函数的前向声明 */
static int dir_iter (lua_State *L);

static int l_dir (lua_State *L) {
  const char *path = luaL_checkstring(L, 1);

  /* 创建一个userdata，用来保存一个DIR的地址 */
  DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

  /* 设置它的元表 */
  luaL_getmetatable(L, "LuaBook.dir");
  lua_setmetatable(L, -2);

  /* 尝试打开指定的目录 */
  *d = opendir(path);
  if (*d == NULL)  /* 打开目录错误? */
    luaL_error(L, "cannot open %s: %s", path, strerror(errno));

  /* 创建并返回迭代器，它唯一的upvalue就是目录userdata，此时正位于栈顶 */
  lua_pushcclosure(L, dir_iter, 1);
return 1;
}
```

在这个函数中要注意，必须在打开目录前创建 userdata。如果先打开目录，再调用 lua_newuserdata 就会引发一个错误，这样会丢失 DIR 结构。通过正确的次序，DIR 结构一旦创建就立即与 userdata 关联了起来。然后不管发生什么，--gc 元方法最终都会释放这个结构。

下一个函数是 dir_iter，也就是迭代器本身。它的代码很容易理解。首先，它从 upvalue 获取 DIR 结构的地址，然后调用 readdir 读取下一个条目。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
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;  /* 没有返回值 */
}
```

下一个函数 dir_gc 是--gc 元方法，这个元方法会关闭一个目录。由于是在打开目录前创建 userdata 的，不论 opendir 的结果是什么，userdata 都会被回收。因此如果 opendir 失败了，就无须关闭什么了。

```
static int dir_gc (lua_State *L) {
  DIR *d = *(DIR **)lua_touserdata(L, 1);
  if (d) closedir(d);
  return 0;
}
```

下一个函数是 luaopen_dir，它会打开只有一个函数的库。

```
int luaopen_dir (lua_State *L) {
  luaL_newmetatable(L, "LuaBook.dir");

  /* 设置它的--gc 字段 */
  lua_pushstring(L, "--gc");
  lua_pushcfunction(L, dir_gc);
  lua_settable(L, -3);

  /* 注册'dir'函数 */
  lua_pushcfunction(L, l_dir);
  lua_setglobal(L, "dir");

return 0;
}
```

整个实例还有一点需要注意。表面看来，dir_gc 似乎应该检查其参数是否为一个目录。如果没有这样的检查，一个恶意的用户可能会用另一种 userdata（例如一个文件 userdata）来调用它，这样会造成未定义的后果。其实，一个 Lua 程序是无法访问到这个函数的。因为这个函数存在于目录 userdata 的元表中，Lua 代码是永远访问不到这些 userdata 及其元表的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

29.2 XML 分析器

接下来介绍一个 lxp 的简化实现，lxp 是一套封装了 Expat 1.2 的 Lua 库。Expat 是一个用 C 语言编写的、开源的 XML 1.0 分析器。它实现了 SAX（Simple API for XML 和 XML 的简单 API）。SAX 是一种基于事件的 API。一个 SAX 分析器在读取 XML 文档时，它会一边读取一边通过回调函数报告读取到的内容。例如，如果让 Expat 分析字符串"<tag cap="5">hi</tag>"，它会产生 3 个事件：当读取到子串"<tag cap="5">"时，产生“开始元素（start-element）”事件；当读取到"hi"时，产生“文本（Text）”事件，也称为“字符数据（character data）”事件；当读取到"</tag>"时，生成“结束元素（end-element）”事件。每个事件都会调用程序中对应的回调处理函数（callback handler）。

这里不会介绍整个 Expat 库，只关注于那些能演示与 Lua 交互的部分。虽然 Expat 可以处理一大堆的事件，不过这里只考虑其中的 3 个事件，即前例中所介绍的“开始元素”、“结束元素”和“文本”。[①]

本例会用到的 Expat API 很少。首先，是关于创建和销毁一个 Expat 分析器的函数：

```
XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

参数 encoding 是一个可选参数，本例将使用 NULL。

当创建完一个分析器后，必须注册其回调处理函数：

```
XML_SetElementHandler(XML_Parser p,
                      XML_StartElementHandler start,
                      XML_EndElementHandler end);

XML_SetCharacterDataHandler(XML_Parser p,
                            XML_CharacterDataHandler hndl);
```

第一个函数为了“开始元素”和“结束元素”注册处理函数。第二个函数为了“文本”注册处理函数。

所有的回调处理函数的第一个参数都是一个用户数据。“开始元素”处理函数还接受一个标记（Tag）名及其属性：

```
typedef void (*XML_StartElementHandler)(void *uData,
                                        const char *name,
                                         const char **atts);
```

① 来自于 Kepler 项目的 LuaExpat 包提供了一套完整的 Expat 访问接口。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

属性是一个以 0 结尾的字符串数组，其中每两个连续的字符串保存着一个属性的名称和值。“结束元素”处理函数除了用户数据这个参数外，只有一个额外的参数，即标记名：

```
typedef void (*XML_EndElementHandler)(void *uData,
                                      const char *name);
```

最后，“文本”处理函数的额外参数是文本的内容。这些内容可以不是 0 结尾的，因此这个函数还有一个显式的长度参数：

```
typedef void (*XML_CharacterDataHandler)(void *uData,
                                         const char *s,
                                         int len);
```

为了将数据输入 Expat，需要使用以下函数：

```
int XML_Parse (XML_Parser p, const char *s, int len, int isLast);
```

在 Expat 中，可以多次调用 XML_Parse 以分段的方式来分析文档。XML_Parse 的最后一个参数 isLast 用于告诉 Expat 一个片段是否是文档的最后内容。注意，每段内容无须以 0 结尾，而是需要显式地指出其长度。如果 XML_Parse 发现一个分析错误，它会返回 0。[①]

最后一个需要用到的 Expat 函数是设置用户数据的函数，通过它设置的用户数据将传给所有的处理函数：

```
void XML_SetUserData (XML_Parser p, void *uData);
```

如何在 Lua 中使用这个程序库，最简单的做法是将所有这些函数导出给 Lua。而更好的做法是编写一个适用于 Lua 的功能层。例如，由于 Lua 是弱类型的，就不必为每种回调函数的设置而编写不同的函数。甚至可以完全避免这些用于注册回调的函数。只需在创建一个分析器时，提供一个回调函数 table，其中包含所有的回调处理函数。例如，如果要打印一个文档的结构，可以使用下面这个回调函数 table：

```
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")
```

① Expat 还提供了一个用于检索错误信息的函数，不过为了简化起见，这里就忽略它了。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  end,
}
```

输入内容"<to> <yes/> </to>"，这些处理函数会打印：

```
+ to
+   yes
-   yes
- to
```

有了这个 API，就无须操作那些回调的函数了。我们可以直接在回调函数 table 中操作它们。由此，整个 API 只需用到 3 个函数：创建分析器的函数、分析一个数据片段的函数、关闭分析器的函数。事实上，可以将后两个函数实现为分析器对象的方法。这套 API 的典型应用代码如下：

```
p = lxp.new(callbacks)          -- 创建新的分析器

for l in io.lines() do          -- 遍历输入行
  assert(p:parse(l))            -- 分析行
  assert(p:parse("\n"))         -- 添加一个回车
end

assert(p:parse())               -- 完成文档
p:close()
```

现在讨论如何来实现。首先要决定如何在 Lua 中表示一个分析器。通常会想到使用 userdata，不过要在 userdata 中放什么东西呢？至少，需要持有实际的 Expat 分析器和回调函数 table。不过，无法在一个 userdata 中（或任何 C 函数中）保存一个 Lua table。在 Lua 5.0 中，可以使用一个 table 的引用。而在 Lua 5.1 中，可以将这个 table 设置为 userdata 的环境。另外，还必须在一个分析器对象中保存一个 Lua 状态。因为 Expat 回调函数需要调用 Lua 的 API 函数，而这些回调函数唯一能收到的参数就是这些分析器对象。这样，一个分析器对象的定义如下：

```
#include <stdlib.h>
#include "xmlparse.h"
#include "lua.h"
#include "lauxlib.h"

typedef struct lxp_userdata {
  lua_State *L;
  XML_Parser *parser;          /* 关联的 expat 分析器 */
} lxp_userdata;
```

下面介绍创建分析器对象的函数 lxp_make_parser。代码如下：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
/* 回调函数的前向声明 */
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts);
static void f_CharData (void *ud, const char *s, int len);
static void f_EndElement (void *ud, const char *name);

static int lxp_make_parser (lua_State *L) {
  XML_Parser p;
  lxp_userdata *xpu;

  /* (1) 创建一个分析器对象 */
  xpu = (lxp_userdata *)lua_newuserdata(L,
                                sizeof(lxp_userdata));

  /* 预先初始化它，以备错误情况 */
  xpu->parser = NULL;

  /* 设置它的元表 */
  luaL_getmetatable(L, "Expat");
  lua_setmetatable(L, -2);

  /* (2) 创建 Expat 分析器 */
  p = xpu->parser = XML_ParserCreate(NULL);
  if (!p)
    luaL_error(L, "XML_ParserCreate failed");

  /* (3) 检查并保存回调 table */
  luaL_checktype(L, 1, LUA_TTABLE);
  lua_pushvalue(L, 1);  /* 压入 table 的副本 */
  lua_setfenv(L, -2);   /* 将它设为 userdata 的环境 */

  /* (4) 配置 Expat 分析器 */
  XML_SetUserData(p, xpu);
  XML_SetElementHandler(p, f_StartElement, f_EndElement);
  XML_SetCharacterDataHandler(p, f_CharData);
  return 1;
}
```

这个函数有 4 个主要步骤。第一步是常见的初始化方式。首先创建一个 userdata，然后预初始化 userdata，最后设置它的元表。其中的预初始化不可或缺。如果在初始化过程中发生了任何错误，必须确保“终结函数（也就是--gc 元方法）”能正确地释放 userdata。第二步创建一个 Expat 分析器，将其存储到 userdata 中，并检查了错误情况。第三步确认函数的第一个参数是一个 table（回调函数 table），并将其设为新 userdata 的环境。最后一步初始化 Expat 分析器。它将 userdata 设为传递给回调函数的参数，并设置了所有的回调函数。注意，这些回调函数对

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

于所有分析器都是相同的。毕竟，用户无法在 C 语言中动态地创建新函数。这些固定的 C 函数会通过回调函数 table 来决定每次应调用哪些 Lua 函数。

接下来是 parse 方法，这个方法用于分析一段 XML 数据。用函数 lxp_parse 来表示这个方法，代码如下：

```
static int lxp_parse (lua_State *L) {
  int status;
  size_t len;
  const char *s;
  lxp_userdata *xpu;

  /* 获取并检查第一个参数（应为一个分析器） */
  xpu = (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* 获取第二个参数（一个字符串） */
  s = luaL_optlstring(L, 2, NULL, &len);

  /* 为回调处理函数准备环境： */
  /* 将回调函数table放入栈的索引3上 */
  lua_settop(L, 2);
  lua_getfenv(L, 1);
  xpu->L = L;  /* 设置Lua状态 */

  /* 调用Expat来分析字符串 */
  status = XML_Parse(xpu->parser, s, (int)len, s == NULL);

  /* 返回错误代码 */
  lua_pushboolean(L, status);
  return 1;
}
```

这个方法有两个参数：分析器对象和一个可选的 XML 数据片段。如果调用它时不传入任何数据，即通知 Expat 文档已结束。

当 lxp_parse 调用 XML_Parse 时，XML_Parse 会为每个找到的元素回调处理函数。因此，lxp_parse 先为这些处理函数的调用准备了一个环境。XML_Parse 还要注意一个细节，这个函数的最后一个参数用于告诉 Expat，传入的数据片段是否是最后一块。但当不用参数调用 parse 时，s 就为 NULL，从而使得最后这个参数为真。

然后详细讨论回调函数 f_StartElement、f_EndElement 和 f_CharData。这 3 个函数都有一个类似的结构，它们都检查了 table 是否为特定事件定义了 Lua 处理函数，如果定义了则准备好参数，并调用这个 Lua 处理函数。

先看一下 f_CharData，代码如下：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
static void f_CharData (void *ud, const char *s, int len) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  /* 获取处理函数 */
  lua_getfield(L, 3, "CharacterData");
  if (lua_isnil(L, -1)) {  /* 没有处理函数吗? */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* 压入分析器('self') */
  lua_pushlstring(L, s, len);  /* 压入字符数据 */
  lua_call(L, 2, 0);  /* 调用处理函数 */
}
```

它的代码很简单。这个回调函数（也包括其他的回调函数）接收一个 lxp_userdata 结构作为其第一个参数。这是在创建分析器时调用 XML_SetUserData 所设置的。当检索到 Lua 状态后，函数便可以访问到那个由 lxp_parse 设置的环境了。其中，回调函数 table 位于索引 3 上，分析器自身位于索引 1 上。然后，它用分析器和字符数据（作为一个字符串）作为参数，调用了 Lua 中对应的处理函数（如果有的话）。

f_EndElement 处理函数与 f_CharData 一样简单，代码如下：

```
static void f_EndElement (void *ud, const char *name) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

lua_getfield(L, 3, "EndElement");
  if (lua_isnil(L, -1)) {  /* 没有处理函数吗? */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* 压入分析器('self') */
  lua_pushstring(L, name);  /* 压入标记名 */
  lua_call(L, 2, 0);  /* 调用处理函数 */
}
```

它也以分析器和标记名（也是作为一个字符串，但现在以 0 结尾）作为参数，调用了相应的 Lua 处理函数。

最后一个处理函数是 f_StartElement，代码如下：

```
static void f_StartElement (void *ud,
                            const char *name,
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
                                const char **atts) {
  lxp_userdata *xpu = (lxp_userdata *)ud;
  lua_State *L = xpu->L;

  lua_getfield(L, 3, "StartElement");
  if (lua_isnil(L, -1)) {  /* 没有处理函数吗？ */
    lua_pop(L, 1);
    return;
  }

  lua_pushvalue(L, 1);  /* 压入分析器('self') */
  lua_pushstring(L, name);  /* 压入标记名 */

  /* 创建并填充属性table */
  lua_newtable(L);
  for (; *atts; atts += 2) {
    lua_pushstring(L, *(atts + 1));
    lua_setfield(L, -2, *atts);  /* table[*atts] = *(atts+1) */
  }

  lua_call(L, 3, 0);  /* 调用处理函数 */
}
```

它以分析器、标记名和一个属性列表作为参数，调用了 Lua 中的处理函数。它比上面两个函数要复杂一些，因为它需要将属性的标记列表翻译到 Lua 中。它使用了一般的翻译方法，创建了一个包含属性名和属性值的 table。例如，对于以下开始标记：

```
<to method="post" priority="high">
```

会产生这样一个属性 table：

```
{method = "post", priority = "high"}
```

分析器的最后一个方法是 close，代码如下：

```
static int lxp_close (lua_State *L) {
  lxp_userdata *xpu =
              (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

  /* 释放Expat分析器（如果有的话） */
  if (xpu->parser)
    XML_ParserFree(xpu->parser);
  xpu->parser = NULL;
  return 0;
}
```

当关闭一个分析器时，必须释放其资源，也就是 Expat 结构。然而，由于在创建分析器时

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

可能会发生错误，userdata 中可能没有这项资源。另一方面，在释放分析器后仍将其设为零，这样当再次关闭它时或当垃圾收集器终结它时，都不会发生问题。实际上，还可以将这个函数用作“终结函数”。这样便可以确保，即使程序员没有关闭一个分析器，最终它也能释放其资源。

最后是一个打开库的函数，它将前面所有的部分组织到了一起。代码如下：

```
static const struct luaL_Reg lxp_meths[] = {
  {"parse", lxp_parse},
  {"close", lxp_close},
  {"--gc", lxp_close},
  {NULL, NULL}
};

static const struct luaL_Reg lxp_funcs[] = {
  {"new", lxp_make_parser},
  {NULL, NULL}
};

int luaopen_lxp (lua_State *L) {
  /* 创建元表 */
  luaL_newmetatable(L, "Expat");

  /* metatable.--index = metatable */
  lua_pushvalue(L, -1);
  lua_setfield(L, -2, "--index");

  /* 注册方法 */
  luaL_register(L, NULL, lxp_meths);

  /* 注册函数（只有lxp.new） */
  luaL_register(L, "lxp", lxp_funcs);
  return 1;
}
```

在此使用的形式与 28.3 节中那个面向对象布尔数组示例相同。就是，创建一个元表，并将所有的方法放入其中，最后将元表的--index 字段指向其自身。为此，需要一个分析器方法列表（lxp_meths），还需要一个库函数列表（lxp_funcs）。与大多数面向对象的库一样，这里的库函数列表中也只有一个函数，就是创建分析器对象的函数。在这个打开函数中创建了一个元表，并使这个元表指向自身（通过--index），最后注册了方法与函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 30 章　线程和状态

Lua 不支持真正的多线程，也就是不支持那种共享内存的抢先式（preemptive）多线程。有两个原因导致 Lua 不支持这种多线程。首先，ANSI C 没有提供这样的功能，并且也没有可移植的方法能在 Lua 中实现这种机制。第二个理由则更为充分，在 Lua 中引入多线程并不是一个好的选择。

多线程一般用于底层的程序开发，像信号量（semaphore）和监视器（monitor）这样的同步机制一般都是在操作系统层面的开发中使用的，而非普通的应用程序。查找并修正多线程相关的错误是很困难的，其中有些错误还会导致安全隐患。此外，在一个程序的某些临界部分，由于需要同步几个线程，由此还可能导致性能损失。例如，内存分配器就是这样的例子。

多线程的这些问题源于两种事物的组合：抢先式的线程与共享的内存。如果使用非抢先式的线程或者不共享内存，那么就可以避免这些问题。Lua 同时提供了对这两种解决方案的支持。Lua 的线程（也就是所谓的“协同程序”）是协作式的（collaborative），因此可以避免由不可预知的线程切换所带来的问题。另一方面，Lua 的多个状态之间不共享内存，这样便为 Lua 中的并发操作提供了良好的基础。本章将会介绍这两种做法。

30.1　多 个 线 程

在 Lua 中，一个线程本质上就是一个协同程序。即可以认为协同程序就是一个线程外加一套良好的操作接口。或者，也可以将线程认为是一个具有底层 API 的协同程序。

从 C API 的角度看，将线程想像成一个栈可能更形象些。不过从实现的观点来看，一个线程的确就是一个栈。每个栈都保留着一个线程中所有未完成的函数调用信息，这些信息包括调用的函数、每个调用的参数和局部变量。换句话说，一个栈拥有一个线程得以继续运行的所有信息。因为，多个线程就意味着多个独立的栈。

当调用 Lua C API 中的大多数函数时，这些函数都作用于某个特定的栈。例如，lua_pushnumber 必须将数字压入一个指定的栈中，lua_pcall 则需要一个用于调用的栈。那 Lua 又如何知道该使用哪个栈呢？又是如何将一个数字压入不同的栈中呢？答案就在类型 lua_State 中，这些 C API 的第一个参数不仅表示了一个 Lua 状态，还表示了一个记录在该状态中的线程。

只要创建一个 Lua 状态，Lua 就会自动在这个状态中创建一个新线程，这个线程称为“主

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

线程”。主线程永远不会被回收。当使用 lua_close 关闭状态时，它会随着状态一起释放。

调用 lua_newthread 便可以在一个状态中创建其他的线程：

```
lua_State *lua_newthread (lua_State *L);
```

这个函数会返回一个 lua_State 指针，表示新建的线程。它还会将新线程作为一个类型为“thread”的值压入栈中。例如，在执行了以下语句后：

```
L1 = lua_newthread(L);
```

拥有了两个线程 L1 和 L，它们内部都引用了相同的 Lua 状态。每个线程都有其自己的栈。新线程 L1 以一个空栈开始运行，老线程 L 的栈顶就是这个新线程：

```
printf("%d\n", lua_gettop(L1));          --> 0
printf("%s\n", luaL_typename(L, -1));    --> thread
```

除了主线程之外，其他线程和其他 Lua 对象一样都是垃圾回收的对象。当新建一个线程时，线程会压入栈中，这样能确保新线程不会成为垃圾。不要使用未被正确系缚（anchored）的线程[①]。所有 Lua API 的调用都有可能回收未系缚的线程，即使是这个调用使用到了这个线程。例如，以下代码：

```
lua_State *L1 = lua_newthread (L);
lua_pop(L, 1);          /* 对于Lua 而言，L1 现在就是垃圾了 */
lua_pushstring(L1, "hello");
```

lua_pushstring 可能会触发垃圾收集器，并回收 L1（从而间接地导致后续代码在使用 L1 时崩溃）。尽管从 C 代码角度看，它仍然被引用，但 Lua 却是无从得知的。为了避免这种情况，可以持有一个对该线程的引用。例如，将线程始终保留在栈中，或者保留在注册表中。

当拥有一个新线程后，就可以像主线程那样来使用它。可以将元素压入栈中，或者从栈中弹出元素，还可以用它来调用函数等。例如，以下代码在新线程中调用了 f(5)，然后将结果移到了旧线程中：

```
lua_getglobal(L1, "f");   /* 假设有一个全局函数'f' */
lua_pushinteger(L1, 5);
lua_call(L1, 1, 1);
lua_xmove(L1, L, 1);
```

函数 lua_xmove 可以在两个栈之间移动 Lua 值。调用 lua_xmove(F, T, n)会从栈 F 中弹出 n

① 主线程是在内部自动系缚的，因此它不会被回收。
译者注：“未被正确系缚”指的是一个 Lua 对象既不在栈中，又不为其他任何 Lua 对象所引用的情况。在这种情况下，Lua 的垃圾收集器会认为这个对象已是垃圾，进而回收它。通常，在 C 代码中都是通过栈来获得 Lua 对象的值（例如 table 中某个条目的 key 或 value），获得的并非对象本身（或其引用）。因此即使相应的 Lua 对象被回收，也不会影响 C 代码中已获得的该对象的值。但是，thread 类型的对象却是一个例外，具体情况请继续阅读正文。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个元素，并将它们压入 T 中。

然而对于某些应用而言，使用一个主线程已经足够了。其实，使用多线程的主要目的是实现协同程序。从而可以挂起某些协同程序的执行，并在稍后恢复执行。为此，需要使用 lua_resume 函数：

```
int lua_resume (lua_State *L, int narg);
```

lua_resume 可以启动一个协同程序，它的用法就像 lua_call 一样。将待调用的函数压入栈中，并压入其参数，最后在调用 lua_resume 时传入参数的数量 narg。这个行为与 lua_pcall 类似，但有 3 点不同。首先，lua_resume 没有参数用于指出期望的结果数量，它总是返回被调用函数的所有结果。其次，它没有用于指定错误处理函数的参数，发生错误时不会展开（unwind）栈，这就可以在错误发生后检查栈中的情况。最后，如果正在运行的函数交出（yield）了控制权，lua_resume 就会返回一个特殊的代码 LUA_YIELD，并将线程置于一个可以被再次恢复执行的状态。

当 lua_resume 返回 LUA_YIELD 时，线程的栈中只能看到交出控制权时所传递的那些值。调用 lua_gettop 则会返回这些值的数量。若要将这些值移到另一个线程，可以使用 lua_xmove。

为了恢复一个挂起线程的执行，可以再次调用 lua_resume。在这种调用中，Lua 假设栈中所有的值都是由 yield 调用返回的。作为一个特例，如果在一个 lua_resume 返回后与再次调用 lua_resume 之间没有改变过线程栈中的内容，那么 yield 恰好返回它交出的值。

通常，以一个 Lua 函数作为一个协同程序来启动。这个 Lua 函数可以调用其他 Lua 函数，任意一个函数都可以交出控制权，从而使 lua_resume 调用返回。例如，假设有以下定义：

```
function foo (x)  coroutine.yield(10, x)  end

function foo1 (x)  foo(x + 1); return 3  end
```

现在运行这段 C 代码：

```
lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "foo1");
lua_pushinteger(L1, 20);
lua_resume(L1, 1);
```

调用 lua_resume 会返回 LUA_YIELD，表示线程已交出了控制权。此时 L1 的栈中便有了交出的值：

```
printf("%d\n", lua_gettop(L1));          --> 2
printf("%d\n", lua_tointeger(L1, 1));    --> 10
printf("%d\n", lua_tointeger(L1, 2));    --> 21
```

当再次恢复线程的执行时，它会从停止的地方（就是 yield 调用处）继续执行。也就是，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

从 foo 返回到 foo1，foo1 继而返回到 lua_resume。

```
lua_resume(L1, 0);
printf("%d\n", lua_gettop(L1));          --> 1
printf("%d\n", lua_tointeger(L1, 1));     --> 3
```

第二次调用 lua_resume 会返回 0，这表示一个正常的返回。

一个协同程序也可以调用 C 函数。因此在 C 语言中编程时就产生了一个很常的问题，能从一个 C 函数中交出控制权吗？

标准的 Lua 在交出控制权时无法穿越 C 函数的调用[①]。这就意味着一个 C 函数不能挂起它自己。一个 C 函数只有在返回时，才会交出控制权。因此 C 函数实际上是不会停止自身执行的。不过它的调用者可以是一个 Lua 函数。那么这个 C 函数调用 lua_yield，就可以挂起 Lua 调用者：

```
return lua_yield(L, nres);
```

其中 nres 是准备返回给相应 resume 的栈顶值的个数。当线程再次恢复执行时，Lua 调用者会收到传递给 resume 的值。

由于 C 函数不能交出控制权，所以会带来一些限制。特别是在一个 Lua 循环中调用 yield 时，当函数交出控制权并再次恢复执行时，循环会再次调用这个函数。例如，假设要编写一个函数来读取一些数据，并且在无数据可读时交出控制权。那么在 C 中可以这么写：

```
int prim_read (lua_State *L) {
  if (nothing_to_read())
    return lua_yield(L, 0);
  lua_pushstring(L, read_some_data());
  return 1;
}
```

如果这个函数有数据可读，那么它会读取并返回这些数据。否则，它会交出控制权。而当线程再次恢复执行时，并不是从 prim_read 中继续执行，而是从它的调用者开始执行的。

假设，调用者在一个循环中调用了 prim_read：

```
function read ()
  local line
  repeat
    line = prim_read()
  until line
  return line
end
```

① 一些 Lua 补丁可以做到这点。不过它们都使用了不可移植的代码，有些还使用了少量的汇编代码。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

当 prim_read 交出控制权时，这个线程就挂起了。而当它恢复执行时，它是从 prim_read 返回处继续执行，也就是对 line 的赋值。而这时赋予 line 的值实际上是传给 resume 的值。如果没有传给 resume 任何值，line 就为 nil。然后，循环会继续执行，并再次调用 prim_read。整个过程会反复进行，直至读到了一些数据，或者 resume 传入了一个非 nil 的值。

30.2 Lua 状态

每次调用 luaL_newstate（或者 lua_newstate，详见第 31 章）都会创建一个新的 Lua 状态。不同的 Lua 状态是各自完全独立的，它们之间不共享任何数据。也就是说，在一个 Lua 状态中发生的错误不会影响其他 Lua 状态。并且，Lua 状态之间不能直接沟通，必须写一些辅助代码来做到这点。例如，对于两个状态 L1 和 L2，以下代码会将 L1 栈顶的字符串压入 L2 中：

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

由于所有交换的数据必须经由 C 代码中转，所以只能在 Lua 状态间交换那些可以在 C 语言中表示的类型，例如字符串和数字。

在提供了多线程的系统中，有一种 Lua 应用方式，就是为每个线程创建一个独立的 Lua 状态。这种架构使得线程类似于 UNIX 中的进程，它们可以并发执行但却不共享内存。本节，将根据这种方法开发一个多线程的原型实现。在这个实现中，我将使用 POSIX 线程（pthreads）。你可以很容易地将这些代码移植到其他线程系统中，因为这些代码只用到了一些基础的线程功能。

要开发的系统很简单，主要目的就是演示在一个多线程环境中使用多个 Lua 状态。在理解并运行后，可以在它之上添加更多高级的功能。接下来将这个库称为 lproc。它只提供了 4 个函数：

- lproc.start(chunk)：启动一个新的 Lua 进程，并运行指定的程序块（一个字符串）。一个 Lua 进程实现为一个 C 线程和一个关联的 Lua 状态。
- lproc.send(channel, val1, val2, ...)：将所有给出的值（应为字符串）发送到指定的通道。通道由其名称来标识，这也是一个字符串。
- lproc.receive(channel)：接收发送给指定通道的值。
- lproc.exit()：结束一个进程。只有主进程需要这个函数。如果主进程不调用 lproc.exit 就结束了，那么整个程序会终止，并且不会等待其他进程的结束。

通道只是一些用于匹配发送者和接收者的简单字符串。一个发送操作可以发送任意多个字符串值，它们会由对应的接收操作返回。所有的通讯都是同步的，一个进程在将消息发送到指定通道时，会一直处于阻塞状态，直到另外有一个进程对这个通道进行了接收操作。类似地，当一个进程在接收一个通道时，也会处于阻塞状态，直到另一个进程向这个通道进行

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

了发送操作。

和系统的接口一样，实现本身也很简单。它用到了两个双向环形链表，一个用于进程等待发送消息，另一个用于进程等待接收消息。它用一个互斥量（Mutex）来控制对这些链表的访问。每个进程有一个关联的条件变量。当一个进程要向一个通道发送消息时，它会遍历接收链表，查看是否有一个正在等待接收的通道。如果找到了这样一个进程，它就将这个进程从等待链表中删除，并将消息的值从自身移到这个进程中，最后通知这个进程。如果没有这个进程，它就将自己插入发送链表，然后等待它的条件变量。对于接收操作，实现的操作也基本类似。

在这个实现中，一个主要的元素就是表示进程的结构：

```
struct Proc {
  lua_State *L;
  pthread_t thread;
  pthread_cond_t cond;
  const char *channel;
  struct Proc *previous, *next;
} Proc;
```

前两个字段表示进程使用的 Lua 状态和运行 Lua 进程的 C 线程。其他字段只在进程等待一个对应的接收/发送时使用。第三个字段 cond 是一个条件变量，线程用它来使自己进入阻塞状态。第四个字段存储了进程正在等待的通道，最后两个字段 previous 和 next 用于在等待链表中链接各个进程。

两个等待链表和关联的互斥量声明如下：

```
static Proc *waitsend = NULL;
static Proc *waitreceive = NULL;

static pthread_mutex_t kernel_access = PTHREAD_MUTEX_INITIALIZER;
```

每个进程都需要一个 Proc 结构，每当进程的脚本调用了 send 或 receive，进程就需要访问这个结构。而这些函数所需的唯一参数就是进程的 Lua 状态。因此，每个进程都应将其 Proc 结构存储在它的 Lua 状态中。例如，以"_SELF"为 key，将 Proc 结构作为 userdata 存储在注册表中。getself 函数可以检索与一个指定 Lua 状态所关联的 Proc 结构：

```
static Proc *getself (lua_State *L) {
  Proc *p;
  lua_getfield(L, LUA_REGISTRYINDEX, "_SELF");
  p = (Proc *)lua_touserdata(L, -1);
  lua_pop(L, 1);
  return p;
}
```

下一个函数 movevalues 可以将一些值从发送者进程复制到接收者进程：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
static void movevalues (lua_State *send, lua_State *rec) {
  int n = lua_gettop(send);
  int i;
  for (i = 2; i <= n; i++)  /* 将值复制到接收者 */
    lua_pushstring(rec, lua_tostring(send, i));
}
```

这个函数将发送者栈中所有的值（除了第一个，它是通道名）复制到接收者栈中。

下面代码定义了一个辅助函数 searchmatch：

```
static Proc *searchmatch (const char *channel, Proc **list) {
  Proc *node = *list;
  if (node == NULL) return NULL;  /* 空链表? */
  do {
    if (strcmp(channel, node->channel) == 0) {  /* 匹配? */
      /* remove node from the list */
      if (*list == node)  /* 这个节点是第一个元素吗? */
        *list = (node->next == node) ? NULL : node->next;
      node->previous->next = node->next;
      node->next->previous = node->previous;
      return node;
    }
    node = node->next;
  } while (node != *list);
  return NULL;  /* 无匹配 */
}
```

这个函数遍历一个等待链表，查找是否有一个进程正在等待指定的通道。如果找到了，函数就将此进程从链表中删除，并返回它；否则，函数返回 NULL。

最后一个辅助函数 waitonlist 定义如下：

```
static void waitonlist (lua_State *L, const char *channel,
                                      Proc **list) {
Proc *p = getself(L);

  /* 将自身链接到链表末尾 */
  if (*list == NULL) {  /* empty list? */
    *list = p;
    p->previous = p->next = p;
  }
  else {
    p->previous = (*list)->previous;
    p->next = *list;
    p->previous->next = p->next->previous = p;
  }
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  p->channel = channel;

  do {  /* 等待其条件变量 */
     pthread_cond_wait(&p->cond, &kernel_access);
   } while (p->channel);
}
```

当进程无法找到一个对应的发送/接收操作时就会调用这个函数。在这种情况下，进程会将自己链接到相应等待链表的末尾，然后进入等待状态，直到另一个进程调用了相应的操作，从而唤醒它。当一个进程唤醒另一个进程时，它会将另一个进程的 channel 字段设置为 NULL。因此，如果 p->channel 不是 NULL，就表示尚未出现与进程 p 对应的进程，所以它必须继续等待。[①]

有了这些辅助函数后，就可以编写 send 和 receive：

```
static int ll_send (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);

pthread_mutex_lock(&kernel_access);

p = searchmatch(channel, &waitreceive);

if (p) {  /* 找到了一个对应的接收者？ */
   movevalues(L, p->L);  /* 将值复制到接收者 */
   p->channel = NULL;  /* 将接收者标记为不用等待 */
   pthread_cond_signal(&p->cond);  /* 唤醒它 */
  }
  else
   waitonlist(L, channel, &waitsend);

  pthread_mutex_unlock(&kernel_access);
  return 0;
}

static int ll_receive (lua_State *L) {
  Proc *p;
  const char *channel = luaL_checkstring(L, 1);
  lua_settop(L, 1);

pthread_mutex_lock(&kernel_access);
```

① pthread_cond_wait 外面的循环用于保护 POSIX 线程所认可的一种“假醒（spurious wakeup）”。
译者注：详见 POSIX 线程的相关文档，其中有关于 pthread_cond_wait 返回后，它所等待的条件变量仍为“假”（这种情况即为“假醒”）的解释。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
p = searchmatch(channel, &waitsend);

if (p) {  /* 找到了一个对应的发送者? */
   movevalues(p->L, L);  /* 从发送者获取值 */
   p->channel = NULL;  /* 将发送者标记为不用等待 */
   pthread_cond_signal(&p->cond);  /* 唤醒它 */
 }
 else
   waitonlist(L, channel, &waitreceive);

pthread_mutex_unlock(&kernel_access);

/* 返回栈中除通道以外所有的值 */
 return lua_gettop(L) - 1;
}
```

send 函数首先检查通道，然后锁住互斥量，并搜索一个对应的接收者。如果找到了，就将它的值复制到这个接收者，并将接收者标记为“准备好了”，最后唤醒它。否则，就将自己放入等待链表。当 send 完成所有操作后，就对互斥量解锁，并直接返回 Lua。receive 函数与之类似，但它会返回所有接收到的值。

现在，来看一下如何创建新进程。一个新的 Lua 进程需要一个新的 C 线程，而一个 C 线程需要一个函数体。在后面内容中会定义这个函数体，在此先看一下它的原型，这也是 POSIX 线程所要求的：

```
static void *ll_thread (void *arg);
```

为了创建并运行一个新进程，系统必须创建一个新的 Lua 状态、启动一个新线程、编译指定的程序块、调用程序块，最后释放这些资源。原线程会做前 3 件事情，而新线程则做其余的事情。另外，为了简化错误处理，系统只在成功编译指定程序块后才启动新的线程。创建新进程的函数 ll_strat 如下：

```
static int ll_start (lua_State *L) {
  pthread_t thread;
  const char *chunk = luaL_checkstring(L, 1);
  lua_State *L1 = luaL_newstate();

if (L1 == NULL)
   luaL_error(L, "unable to create new state");

if (luaL_loadstring(L1, chunk) != 0)
   luaL_error(L, "error starting thread: %s",
                 lua_tostring(L1, -1));
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  if (pthread_create(&thread, NULL, ll_thread, L1) != 0)
    luaL_error(L, "unable to create new thread");

  pthread_detach(thread);
  return 0;
}
```

这个函数创建一个新 Lua 状态 L1，并编译给定的程序块。若发生错误，它就将错误报告到原 Lua 状态 L。然后，它用函数 ll_thread 创建了一个新线程（pthread_create），并以新状态 L1 为参数传入线程函数。最后调用 pthread_detach 用于告诉系统，不再需要知道这个线程的最终运行结果。

每个新线程都以函数 ll_thread 作为其执行体：

```
static void *ll_thread (void *arg) {
  lua_State *L = (lua_State *)arg;
  luaL_openlibs(L);  /* 打开标准库 */
  lua_cpcall(L, luaopen_lproc, NULL);  /* 打开 lproc 库 */
  if (lua_pcall(L, 0, 0, 0) != 0)  /* 调用主程序块 */
    fprintf(stderr, "thread error: %s", lua_tostring(L, -1));
  pthread_cond_destroy(&getself(L)->cond);
  lua_close(L);
  return NULL;
}
```

它从 ll_start 接收相应的 Lua 状态，这个状态的栈中只有已编译好的主程序块。新线程会打开 Lua 的标准库和 lproc 库，然后调用它的主程序块。最后，它释放其条件变量（由 luaopen_lproc 创建的），并关闭 Lua 状态。

模块中最后一个函数 exit 非常简单：

```
static int ll_exit (lua_State *L) {
  pthread_exit(NULL);
  return 0;
}
```

注意，只有在主进程需要结束时调用这个函数。调用它可以避免整个程序被立即结束。

最后一步是为 lproc 模块定义打开函数。打开函数 luaopen_lproc 如下：

```
static const struct luaL_reg ll_funcs[] = {
  {"start", ll_start},
  {"send", ll_send},
  {"receive", ll_receive},
  {"exit", ll_exit},
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
  {NULL, NULL}
};

int luaopen_lproc (lua_State *L) {
  /* 创建自己控制的块 */
  Proc *self = (Proc *)lua_newuserdata(L, sizeof(Proc));
  lua_setfield(L, LUA_REGISTRYINDEX, "_SELF");
  self->L = L;
  self->thread = pthread_self();
  self->channel = NULL;
  pthread_cond_init(&self->cond, NULL);
  luaL_register(L, "lproc", ll_funcs);  /* 打开库 */
  return 1;
}
```

它必须注册模块中所有的函数，另外还需要为当前运行的进程创建并初始化 Proc 结构。

Lua 进程实现是非常简单的，可以对它做许多改进，在这里将简单地介绍几种。

第一项显而易见的改进是将用于查找匹配通道的算法由线性搜索改为使用散列表，并为每个通道设置一个等待链表。

另一项改进涉及更高效地创建进程。创建一个新的 Lua 状态是一个很简单的操作。然而，打开所有的标准库则需花费十倍于创建新状态的时间。大部分进程可能并不需要用到所有的标准库，而只需用到一到两个库。可以对库进行预注册来避免打开一个无用库的代价，这点已在 15.1 节中讨论过了。相对于为每个标准库调用 luaopen_*函数，使用这种方法时只需将每个标准库的打开函数放入 package.preload 中即可。如果进程调用了 require "lib"，那么 require 就会调用与这个 lib 关联的函数，从而打开这个库。以下函数就完成了这样的注册：

```
static void registerlib (lua_State *L, const char *name,
                                        lua_CFunction f) {
  lua_getglobal(L, "package");
  lua_getfield(L, -1, "preload");  /* 获取'package.preload' */
  lua_pushcfunction(L, f);
  lua_setfield(L, -2, name);  /* package.preload[name] = f */
  lua_pop(L, 2);  /* 弹出table'package'和'preload' */
}
```

一般情况都需要打开 base 库。另外，还需要 package 库。如果没有 package 库，就无法通过 require 来打开其他库。其他所有的库都是可选的。因此，现在打开一个新状态时，使用下面这个 openlibs 函数来代替 luaL_openlibs：

```
static void openlibs (lua_State *L) {
  lua_cpcall(L, luaopen_base, NULL);  /* 打开基础库 */
  lua_cpcall(L, luaopen_package, NULL);  /* 打开package库 */
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
    registerlib(L, "io", luaopen_io);
    registerlib(L, "os", luaopen_os);
    registerlib(L, "table", luaopen_table);
    registerlib(L, "string", luaopen_string);
    registerlib(L, "math", luaopen_math);
    registerlib(L, "debug", luaopen_debug);
}
```

如果一个进程需要其中任意一个库，只需显式地 require 它，require 就会调用相应的 luaopen_*函数了。

另一个改进涉及进程间的通讯。例如，给 lproc.send 和 lproc.receive 提供一个等待时间的参数，用于限制它们等待一个匹配操作的时间。0 可以作为一个特例，用于表示以非阻塞的方式调用这些函数。在 POSIX 线程中，可以用 pthread_cond_timedwait 来实现这个功能。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第31章 内存管理

除了递归下降（recursive-descendent）分析器在C栈上分配的一些数组外，Lua会动态地分配其所有数据结构。所有这些结构都会根据需要增长，并最终缩小或释放。

Lua对其内存使用具有严格控制。当关闭一个Lua状态时，Lua会显式地释放它的所有内存。此外，所有Lua中的对象都是垃圾收集的目标，其中不仅包括table和字符串，还包括函数、线程和模块[①]。如果加载了一个很大的Lua模块，并在之后删除了所有对它的引用，Lua最终会回收这个模块使用的所有内存。

Lua的内存机制可以适用于大多数应用程序。不过，某些特殊的应用程序可能需要一些定制功能。例如，某些在内存受限环境中运行的程序，或者需要将垃圾收集运行时间减至最小的情况。Lua在两个层面提供了对这些定制的支持。在较低层面，可以设置Lua使用的分配函数。在较高层面，可以设置一些控制垃圾收集器的参数，或者直接控制垃圾收集器。在本章中，将介绍这些功能。

31.1 分配函数

Lua 5.1核心不会对内存分配方式作任何假设。它不会调用malloc或realloc来分配内存。而是通过一个“分配函数”来完成所有的内存分配和释放。当用户创建一个Lua状态时，必须提供这个函数。

先前使用的函数luaL_newstate是一个辅助函数，它会以一个默认的分配函数来创建Lua状态。默认的分配函数使用了C标准库中的malloc-realloc-free函数，对于普通的应用程序这已经足够了。然而，要获取对Lua内存分配的完全控制也非常容易，只需用原始的lua_newstate来创建状态就可以了：

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

这个函数接收两个参数分配函数和用户数据。以这种方式创建的状态会调用f来完成所有的内存分配和释放[②]。

分配函数的类型lua_Alloc定义如下：

① 其实它们也是table。
② 甚至lua_State结构本身也是由f分配的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

第一个参数总是在调用 lua_newstate 时提供的用户数据，第二个参数是准备重新分配或释放的内存块地址，第三个参数是内存块的原大小，最后一个参数是要求的内存块大小。

Lua 确保如果 ptr 不是 NULL，那么它就指向一个已分配并且大小为 osize 的内存。Lua 将 NULL 视为一个 0 字节大小的内存块，如果 ptr 为 NULL，osize 就为 0。Lua 不保证 oszie 与 nsize 不同，两者甚至都可能为 0。在这种情况中，分配函数可以简单地返回 ptr[①]。

Lua 希望分配函数将 NULL 视为 0 字节大小的内存块。当 nsize 为 0 时，分配函数必须释放 ptr 指向的内存块，并返回 NULL。这也符合于对一块 0 字节内存的请求。当 osize 为 0 时（此时 ptr 必为 NULL），函数必须分配并返回一个具有指定大小的内存块，如果它无法分配出这样的内存块，就必须返回 NULL[②]。最后，当 osize 和 nsize 都不为 0 时，分配函数应像 realloc 那样重新分配内存块，并返回新的地址（可能是与原来一样的地址）。注意，如果发生错误，必须返回 NULL。Lua 假设若 nsize 小于或等于 osize，分配函数绝不会失败[③]。

luaL_newstate 使用的标准分配函数定义如下：

```
void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
  }
```

它假设 free(NULL)什么也不做，而 realloc(NULL, size)等价于 malloc(size)。ANSI C 标准也保证了同样的行为。

可以调用 lua_getallocf 来获取一个 Lua 状态的内存分配函数：

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

如果 ud 不为 NULL，函数会将*ud 设为这个分配函数的用户数据。还可以修改一个 Lua 状态的内存分配函数，通过 lua_setallocf：

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

记住，任何新的分配函数都要能释放上一个分配函数所分配的内存块。通常，新函数都

① 当 osize 和 nsize 都为 0 时，ptr 就为 NULL。
② 如果 osize 和 nsize 都为 0，前面的描述也同样正确。其最终结果就是，分配函数什么都没有做，只返回了 NULL。
③ Lua 会在垃圾收集时缩小某些结构，而此时它是无法从这类错误中恢复的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

是对老函数的一层包装。例如，跟踪分配或同步堆的访问。

从内部来看，Lua 不会为重用而缓存内存块。它假设分配函数（或者一些较好的分配函数）会做这些。Lua 也不会刻意避免内存碎片。另外，研究表明，内存碎片一般是由于不当的分配策略引起的，而非程序行为所致。

要实现一个优秀的分配函数是比较困难的，不过有时也可以尝试一下。例如，在释放或重分配内存时，Lua 会传来每个内存块的原大小，这是从普通 free 函数中无法获得的大小。因此，一个特殊的分配函数就无须保存关于块大小的信息了，从而可以减少每个块的内存开销。

另一种可以优化内存分配的情况是在多线程的系统中。这种系统通常要求对它们的内存分配函数作同步处理，就像它们使用一个全局资源时所做的那样。然而，对 Lua 状态的访问也必须同步，使同一时间的访问限制在一个线程中，例如在第 30 章实现的 lproc。因此，如果每个 Lua 状态都能从一个私有的内存池分配内存的话，那么分配函数就可以避免这种额外的开销了。

31.2 垃圾收集器

Lua 从第一版本到 5.0 一直采用一种简单的“标记并清扫（mark-and-sweep）”垃圾收集器。这种收集器有时也称为“停止世界的（stop-the-world）”收集器。也就是说，Lua 有时会为了完成一个完整的垃圾收集周期而暂停与主程序的交互。每个周期都由 4 个阶段组成：标记（mark）、整理（cleaning）、清扫（sweep）和收尾（finalization）。

在标记阶段，Lua 先将“根集合（root set）”中的对象标记为“活跃（alive）”。根集合中的对象就是 Lua 可以直接访问的对象，它们是注册表中的对象和主线程对象。然后，Lua 将任何程序可以通过根集合对象访问到的对象也都标记为“活跃”。这样会使所有可到达的对象都标记为“活跃”了。

在开始清扫阶段前，Lua 先要进入整理阶段。这个阶段为“终结函数（finalizer）”和弱引用 table。首先，Lua 遍历所有的 userdata，找出所有未被标记且具有--gc 元方法的 userdata。然后，将这些 userdata 标记为“活跃”，并放入一个单独的列表中。这个列表在收尾阶段会用到。另一方面，Lua 还会遍历所有的弱引用 table，并根据弱引用设置删除其中未被标记的 key 和 value。

在清扫阶段中，Lua 遍历所有的对象[①]。如果当前遍历到的对象未被标记，就收集它。否则，Lua 就清除它的标记，从而为下一个收集周期做准备。

最后是收尾阶段，其中会根据整理阶段中生成的 userdata 列表来调用它们的终结函数。在最后才进行这些调用是为了简化错误处理。因为，一个有问题的终结函数可能会抛出一个错误，而垃圾收集器却不能在一个收集周期的其他阶段中停止下来，否则就有可能使 Lua 置于一个不一致的状态。不过，它若在最后这个阶段停止了，则不会有什么问题。因为，下一个周期还

① 为了做到这样的遍历，Lua 会将它创建的所有对象都保存在一个链表中。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

会继续调用那些留在列表中的 userdata 的终结函数。

Lua 5.1 开始改用了一种增量式的收集器。这种新的收集器做了与原收集器一样的步骤。但是它的运行不会暂停整个程序的响应。它以隔行扫描的方式与解释器一起工作。每当解释器分配了一些固定量的内存后，收集器就会运行一小步。即当收集器工作时，解释器仍可以改变对一个对象的引用关系。为了确保解释器能正常工作，解释器中的有些操作还会检测危险的修改，并纠正所涉及对象的标记。

31.2.1 原子操作

为了避免过于复杂，增量式收集器会以原子的方式来完成某些操作，即某些操作是无法打断的。换句话说，Lua 仍会在一个原子操作中“停止世界”。如果一个原子操作需要很长时间才能完成，它就可能会影响程序的计时。主要的原子操作是 table 的遍历和整理阶段。

原子的 table 遍历表示收集器在遍历一个 table 时是不会停止的。只有当一个程序中具有一个极大的 table 时，才会成为一个问题。如果遇到了这类问题，就应该将 table 分为一些较小的部分。通常的组织方式是将 table 按分类体系来分隔，将相关的条目组织到一个子 table 中。注意，table 中每个条目的大小不会影响 table 遍历，影响的因素是条目的数量。

原子的整理阶段意味着收集器会在一轮运行中收集所有需要清理的 userdata，并清理所有的弱引用 table。如果一个程序中具有特别多的 userdata，或者弱引用 table 中具有特别多的条目时[①]，才会成为一个问题。

这些问题不常出现在实际应用中，不过仍需要有更多这方面的知识来运用新的收集器。

31.2.2 垃圾收集器的 API

Lua 提供了一个 API 可以控制垃圾收集器的某些行为。在 C 语言中，使用 lua_gc：

```
int lua_gc (lua_State *L, int what, int data);
```

在 Lua 中，我们使用 collectgarbage 函数：

```
collectgarbage(what [, data])
```

这两者都提供了相同的功能。what 参数指定了要做的事情，它们有：

LUA_GCSTOP（"stop"），停止收集器，直到再次以"restart"、"collect"或"step"来调用 collectgarbage（或 lua_gc）。

LUA_GCRESTART（"restart"），重启收集器。

LUA_GCCOLLECT（"collect"），执行一轮完整的垃圾收集周期，收集并释放所有不可到

① 无论是一些很大的弱引用 table，还是无数较小的弱引用 table。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

达的对象。这是 collectgarbage 的默认选项。

LUA_GCSTEP（"step"），执行一些垃圾收集工作。工作的总量由第二个参数 data 以一种模糊的方式指定（较大的值表示更多的工作）。

LUA_GCCOUNT（"count"），返回 Lua 当前使用的内存数量，以千字节为单位。这个数字包含了已死亡但尚未回收的对象。

LUA_GCCOUNTB（无对应参数），返回 Lua 当前使用的内存数量的千字节余数。在 C 语言中，总内存用量可以用以下表达式来计算：[①]

```
lua_gc(L, LUA_GCCOUNT, 0)*1024 + lua_gc(L, LUA_GCCOUNTB, 0)
```

在 Lua 中，collectgarbage("count")的结果是一个浮点数，总的内存用量可以这样计算：

```
collectgarbage("count") * 1024
```

因此，collectgarbage 没有等价于 LUA_GCCOUNTB 的选项。

LUA_GCSETPAUSE（"setpause"），设置收集器的 pause 参数，其值由 data 参数指定，表示一个百分比。当 data 为 100 时，pause 参数设为 1（100%）。

LUA_GCSETSTEPMUL（"setstepmul"），设置收集器的 stepmul 参数，其值也是由 data 参数指定，表示一个百分比。

pause 和 stepmul 这两个参数可用于控制收集器的某些特性。它们仍处于试验阶段，目前尚未完全掌握它们会对一个程序的整体性能产生如何的影响。

pause 参数控制了收集器在完成一轮收集至启动下一轮收集间等待的时间。Lua 用一种算法来决定是否开始一轮新的收集。假设，一轮收集结束后，Lua 正在使用 m 千字节，那么它会等到使用 m*pause 千字节时再开始新一轮的收集。也就是说，将 pause 设置为 100%，Lua 会在一轮收集结束后马上开始新一轮收集。而若将 pause 设置为 200%，那么 Lua 会在使用到当前内存的两倍时再启动收集器，这也是默认的行为。如果想用更多的 CPU 时间来换取较小的内存用量，那么可以将 pause 设得小一点。通常，应该将这个值维持在 100%～300%之间。

stepmul 参数控制了收集器的工作速度，这个速度是一个相对于内存分配的速度。这个值越大收集器步进的速度越快。像 100000000%这样的巨值会使收集器变成一个非增量式的收集器。默认值是 200%。小于 100%的值会使收集器变得很慢，慢到无法完成一轮收集。

lua_gc 的其他选项可以更明确地控制收集器。游戏程序通常就需要这类控制。例如，若不想在某段时间内做任何垃圾收集的工作，就可以调用 collectgarbage("stop")来停止它，并在之后通过 collectgarbage("restart")来重新启动它。而在某些具有周期性空闲的系统中，则可以将收集器保持在停止状态，然后在空闲期间调用 collectgarbage("step", n)。为了设置每次空闲期间要做的工作量，可以通过试验找出一个恰当的 n 值，或者在一次循环中调用 collectgarbage 时，将 n 设为 0（表示很小的一步）。

① 假设最终结果未超出一个 int 的表示范围。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

[General Information]
书名=Lua程序设计 （第二版）
作者=[巴西]Roberto, Ierusalimschy著
页数=281
出版社=电子工业出版社
出版日期=2008.5
SS号=12030087
DX号=000006532966

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

封面
版权
前言
目录
第１章　开始
　１．１程序块（ｃｈｕｎｋ）
　１．２词法规范
　１．３全局变量
　１．４解释器程序（ｔｈｅｄｔａｎｄ－ｓｌｏｎｅｉｎｔｅｒｐｒｅｔｅｒ）
第２章　类型与值
　２．１ｎｉｌ（空）
　２．２ｂｏｏｌｅａｎ（布尔）
　２．３ｎｕｍｂｅｒ（数字）
　２．４ｓｔｒｉｎｇ（字符串）
　２．５ｔａｂｌｅ（表）
　２．６ｆｕｎｃｔｉｏｎ（函数）
　２．７ｕｓｅｒｄａｔａ（自定义类型）和ｔｈｒｅａｄ（线程）
第３章　表达式
　３．１算术操作符
　３．２关系操作符
　３．３逻辑操作符
　３．４字符串连接
　３．５优先级
　３．６ｔａｂｌｅ构造式（ｔａｂｌｅｃｏｎｓｔｒｕｃｔｏｒ）
第４章　语句
　４．１赋值
　４．２局部变量与块（ｂｌｏｃｋ）
　４．３控制结构
　　４．３．１ｉｆｔｈｅｎｅｌｓｅ
　　４．３．２ｗｈｉｌｅ
　　４．３．３ｒｅｐｅａｔ
　　４．３．４数字型ｆｏｒ（ｎｕｍｅｒｉｃｆｏｒ）
　　４．３．５泛型ｆｏｒ（ｇｅｎｅｒｉｃｆｏｒ）
　４．４ｂｒｅａｋ与ｒｅｔｕｒｎ
第５章　函数
　５．１多重返回值（ｍｕｌｔｉｐｌｅｒｅｓｕｌｔｓ）
　５．２变长参数（ｖａｒｉａｂｌｅｎｕｍｂｅｒｏｆａｒｇｕｍｅｎｔｓ）
　５．３具名实参（ｎａｍｅｄａｒｇｕｍｅｎｔｓ）
第６章　深入函数
　６．１ｃｌｏｓｕｒｅ（闭合函数）
　６．２非全局的函数（ｎｏｎ－ｇｌｏｂａｌｆｕｎｃｔｉｏｎ）
　６．３正确的尾调用（ｐｒｏｐｅｒｔａｉｌｃａｌｌ）
第７章　迭代器与泛型ｆｏｒ
　７．１迭代器与ｃｌｏｓｕｒｅ
　７．２泛型ｆｏｒ的语义
　７．３无状态的迭代器
　７．４具有复杂状态的迭代器
　７．５真正的迭代器
第８章　编译、执行与错误
　８．１编译
　８．２Ｃ代码
　８．３错误（ｅｒｒｏｒ）

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

8．4错误处理与异常
8．5错误消息与追溯（traceback）
第9章　协同程序（coroutine）
9．1协同程序基础
9．2管道（pipe）与过滤器（filter）
9．3以协同程序实现迭代器
9．4非抢先式的（non-preemptive）多线程
第10章　完整的示例
10．1数据描述
10．2马尔可夫链（markovchain）算法
第11章　数据结构
11．1数组
11．2矩阵与多维数组
11．3链表
11．4队列与双向队列
11．5集合与无序组（bag）
11．6字符串缓冲
11．7图
第12章　数据文件与持久性
12．1数据文件
12．2串行化（Serialization）
12．2．1保存无环的table
12．2．2保存有环的table
第13章　元表（metatable）与元方法（meatmethod）
13．1算术类的元方法
13．2关系类的元方法
13．3库定义的元方法
13．4table访问的元方法
13．4．1_index元方法
13．4．2__newindex元方法
13．4．3具有默认值的table
13．4．4跟踪table的访问
13．4．5只读的table
第14章　环境
14．1具有动态名字的全局变量
14．2全局变量声明
14．3非全局的环境
第15章　模块与包
15．1require函数
15．2编写模块的基本方法
15．3使用环境
15．4module函数
15．5子模块与包
第16章　面向对象编程
16．1类
16．2继承
16．3多重继承
16．4私密性
16．5单一方法（single-method）做法
第17章　弱引用table
17．1备忘录（memoize）函数
17．2对象属性

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

１７．３回顾ｔａｂｌｅ的默认值
第１８章　数学库
第１９章　ｔａｂｌｅ库
１９．１插入和删除
１９．２排序
１９．３连接
第２０章　字符串库
２０．１基础字符串函数
２０．２模式匹配（ｐａｔｔｅｒｎ-ｍａｔｃｈｉｎｇ）函数
２０．２．１ｓｔｒｉｎｇ．ｆｉｎｄ函数
２０．２．２ｓｔｒｉｎｇ.ｍａｔｃｈ函数
２０．２．３ｓｔｒｉｎｇ．ｇｓｕｂ函数
２０．２．４ｓｔｒｉｎｇ．ｇｍａｔｃｈ函数
２０．３模式
２０．４捕获（ｃａｐｔｕｒｅ）
２０．５替换
２０．５．１ＵＲＬ编码
２０．５．２ｔａｂ扩展
２０．６技巧
第２１章　Ｉ／Ｏ库
２１．１简单Ｉ／Ｏ模型
２１．２完整Ｉ／Ｏ模型
２１．２．１性能小诀窍
２１．２．２二进制文件
２１．２．３其他文件操作
第２２章　操作系统库
２２．１日期和时间
２２．２其他系统调用
第２３章　调试库
２３．１自省机制
２３．１．１访问局部变量
２３．１．２访问非局部的变量（ｎｏｎ-ｌｏｃａｌｖａｒｉｂｌｅ）
２３．１．３访问其他协同程序
２３．２钩子
２３．３性能剖析（ｐｒｏｆｉｌｅ）
第２４章　ＣＡＰＩ概述
２４．１第一个示例
２４．２栈
２４．２．１压入元素
２４．２．２查询元素
２４．２．３其他栈操作
２４．３ＣＡＰＩ中的错误处理
２４．３．１应用程序代码中的错误处理
２４．３．２库代码中的错误处理
第２５章　扩展应用程序
２５．１基础
２５．２ｔａｂｌｅ操作
２５．３调用Ｌｕａ函数
２５．４一个通用的调用函数
第２６章　从Ｌｕａ调用Ｃ
２６．１Ｃ函数
２６．２Ｃ模块

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第２７章　编写C函数的技术
２７．1数组操作
２７．2字符串操作
２７．3在C函数中保存状态
２７．３．1注册表（ｒｅｇｉｓｔｒｙ）
２７．３．２C函数的环境
２７．３．３ｕｐｖａｌｕｅ
第２８章　用户自定义类型
２８．1ｕｓｅｒｄａｔａ
２８．2元表
２８．3面向对象的访问
２８．4数组访问
２８．5轻量级ｕｓｅｒｄａｔａ（ｌｉｇｈｔｕｓｅｒｄａｔａ）
第２９章　管理资源
２９．1目录迭代器
２９．2ＸＭＬ分析器
第３０章　线程和状态
３０．1多个线程
３０．2Ｌｕａ状态
第３１章　内存管理
３１．1分配函数
３１．2垃圾收集器
３１．２．1原子操作
３１．２．2垃圾收集器的ＡＰＩ

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。