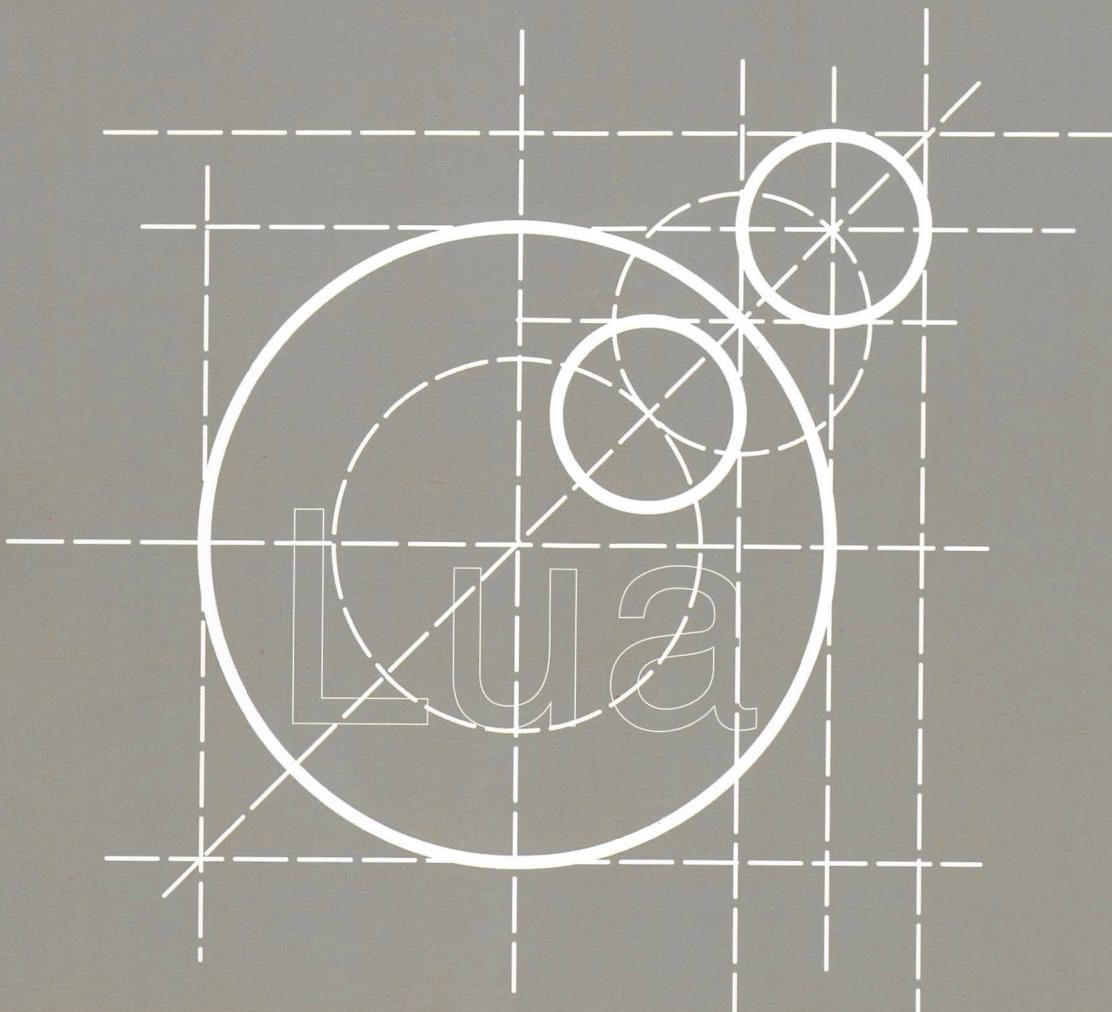


Lua程序设计

Programming in Lua (fourth edition)

(第4版)

[巴西] Roberto Ierusalimschy 著
梅隆魁 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



作者简介

Roberto Lerusalischi 是 Lua 语言的首席架构师，从 1993 年开始从事Lua 语言的研究和开发工作。目前是巴西里约热内卢天主教大学 (PUC-Rio, Pontifical Catholic University of Rio de Janeiro) 计算机科学专业的客座教授，主要从事编程语言的设计和实现工作。

Roberto 在里约热内卢天主教大学获得计算机科学专业的硕士和博士学位，并曾为访问学者在清华大学 (University of Waterloo) ICSI (International Computer Science Institute) 工作。目前在 Illinois at Urbana-Champaign (UIUC) 和斯坦福大学 (Stanford University) 工作和学习。作为里约热内卢天主教大学的数据挖掘小组负责人，Roberto 指导了 Lua 社区有重要影响力的数名弟子。他的同时也是 ACM 协会的独立撰稿人和 IEP 编辑。

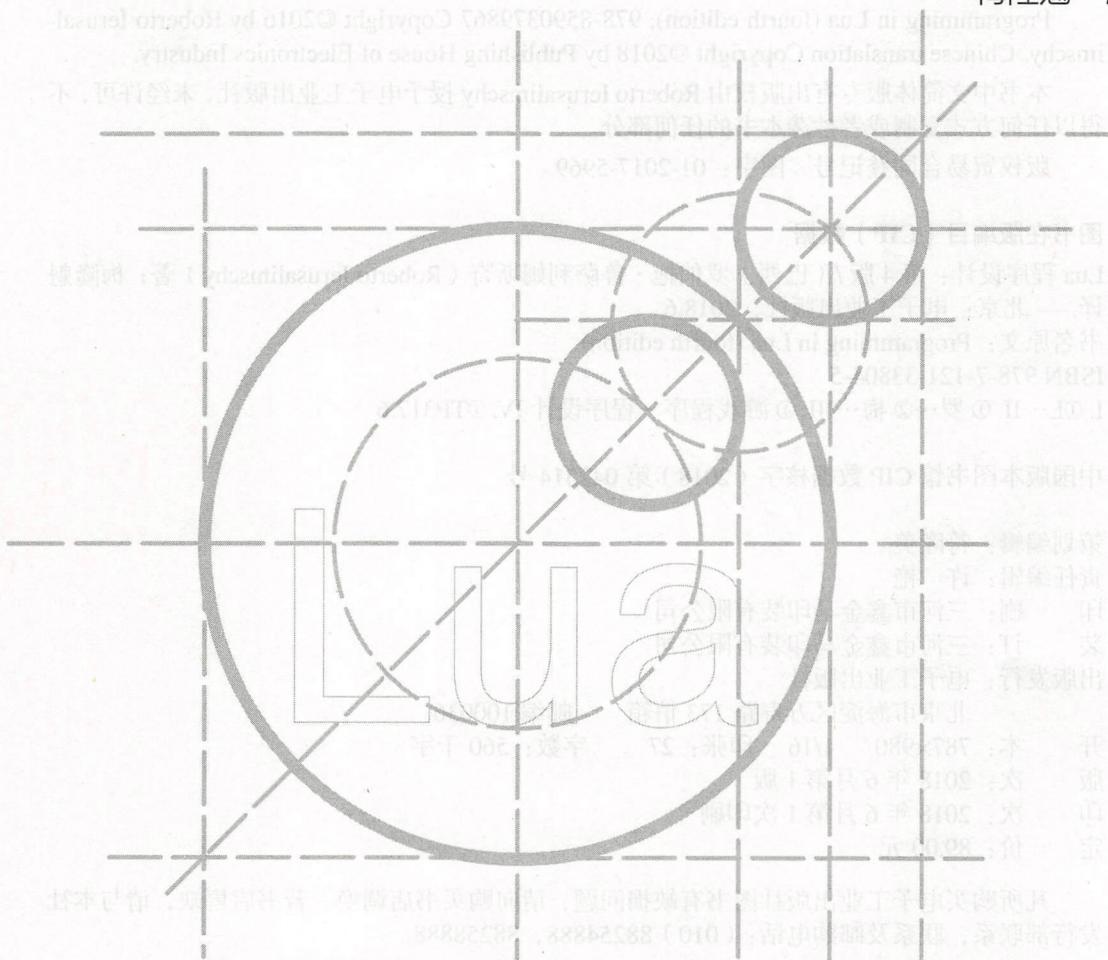


内容简介

Lua程序设计

Programming in Lua (fourth edition)

(第4版)

[巴西] Roberto Ierusalimschy 著
梅隆魁 译

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING





内 容 简 介

本书由 Lua 语言作者亲自撰写，针对 Lua 语言本身由浅入深地从各个方面进行了完整和细致的讲解。作为第 4 版，本书主要针对的是 Lua 5.3，这是本书撰写时 Lua 语言的最新版本。作者从语言使用者的角度出发，讲解了语言基础、编程实操、高级特性及 C 语言 API 等四个方面的内容，既有 Lua 语言基本数据类型、输入输出、控制结构等基础知识，也有对模块、闭包、元表、协程、延续、反射、环境、垃圾回收、函数式编程、面向对象编程、C 语言 API 等高级特性的系统讲解，还有对 Lua 5.3 中引入的整型、位运算、瞬表、延续等新功能的细致说明。

所有与 Lua 语言打交道的人均能从本书受益，包括游戏、嵌入式、物联网、软件安全、逆向工程、移动互联网、C 语言核心系统开发等诸多领域中对 Lua 语言有一般使用需要的从业人员，以及需要从编译原理或语言设计哲学和实现角度深入学习 Lua 语言脚本引擎的高级开发者或研究人员。

Programming in Lua (fourth edition), 978-8590379867 Copyright ©2016 by Roberto Ierusalimschy. Chinese translation Copyright ©2018 by Publishing House of Electronics Industry.

本书中文简体版专有出版权由 Roberto Ierusalimschy 授予电子工业出版社，未经许可，不得以任何方式复制或者抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2017-5969

图书在版编目 (CIP) 数据

Lua 程序设计：第 4 版 / (巴西) 罗伯拖·鲁萨利姆斯奇 (Roberto Ierusalimschy) 著；梅隆魁译。—北京：电子工业出版社，2018.6

书名原文：Programming in Lua (fourth edition)

ISBN 978-7-121-33804-5

I. ①L…II. ①罗…②梅…III. ①游戏程序－程序设计 IV. ①TP317.6

中国版本图书馆 CIP 数据核字 (2018) 第 042514 号

策划编辑：符隆美

责任编辑：许 艳

印 刷：三河市鑫金马印装有限公司

装 订：三河市鑫金马印装有限公司

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：27 字数：560 千字

版 次：2018 年 6 月第 1 版

印 次：2018 年 6 月第 1 次印刷

定 价：89.00 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, 88258888。

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。

本书咨询联系方式：(010) 51260888-819 faq@phei.com.cn。



推荐序一

Lua 这种类似于“胶水”的语言在游戏行业被广泛应用。我已经在游戏行业摸爬滚打了很多年，对游戏行业变化之快深有体会：游戏策划时常更改设计是行业特点，工程师必须把“不要写死，要能热更”这句话刻在心里。因此在做开发时，大家喜欢把逻辑放在 Lua 这种嵌入型语言中，一方面是因为 Lua 性能好，另一方面主流引擎都支持通过推送 Lua 脚本来实现热更新，这样在修改 Bug 或者更新内容时，用户就不需要重新下载整个游戏安装包。并且 Lua 上手难度不高，所以很多初入游戏行业的程序员往往先学 Lua。但是，Lua 的中文学习资料很有限，社区上的知识比较零碎，为数不多的英文书翻译本也质量平平或其中所讲的版本已经过时，增加了初学者系统学习 Lua 的难度。所以，现在我们手中的这本用心打磨的译本，无疑是初学者的福音。

翻译一本有用的但不蹭热点的书，就像我们开发一款源自内心的的喜爱却不跟风的游戏，也许不会成为爆款，但是总会对得起自己，也总能收获一群用户的喜爱。这本《Lua 程序设计（第 4 版）》，体现了译者的“傻劲”——不追捧热点、专注自己想做的事情。这种“傻劲”是这个时代稀缺的。然而读者也好，游戏用户也好，往往就喜欢这种“傻人”和他们的“傻劲”，我真心希望这样的“傻人”“傻劲”能多一些。

译者还计划要做一个 Lua 的社区，欢迎大家关注，也欢迎推荐给身边的朋友，独乐乐不如众乐乐。最后，开卷有益，祝大家学习愉快。

焦洋

盖娅互娱 CTO



推荐序二

这几年来，由于阅读 Lua 虚拟机实现源码的缘故，我深入了解了 Lua 的很多内部实现原理。Lua 作为一门诞生已经超过 20 年的语言，在设计上是非常克制的，以 Lua 5.1.4 版本来说，这个版本是 Lua 发展了十几年之后稳定使用了很长时间的版本，其解释器加上周边的库函数等不过就是一万多行的代码。

在设计上，Lua 语言从一开始就把简单、高效、可移植、可嵌入、可扩展等作为自己的目标。打一个可能不是太恰当的比方，Lua 语言专注于做一个配角，作为胶水语言来辅助像 C、C++ 这样的主角来更好地完成工作，当其他语言在前面攻城拔寨时，Lua 语言在后方实现自己辅助的作用。现在大部分主流编程语言都在走大而全的路线，在号称学会某一门语言就能成为所谓的“全栈工程师”的年代，Lua 语言始终恪守本分地做好“胶水语言”的本职工作，不得不说是一个异类的存在。

“上善若水，水善利万物而不争”，这大概是我能想到的最适合用于来描述 Lua 语言设计哲学的句子。

然而，我发现想找到一本关于 Lua 语言本身设计相关的书籍却很难。打开任何一个电商网站，以关键字“Lua”来进行搜索，能找到的相关书籍大多是如何基于 Lua 做应用开发，如游戏、OpenResty 等。在 2008 年，国内曾引进并翻译了《Lua 程序设计（第 2 版）》。然而，这一本书已经绝版不再印刷，而且 Lua 在这些年里也发生了不少的变化，从当时的 5.1 版本到了现在的 5.3 版本，也在更多领域有了广泛的应用。此时，引进并且翻译最新版本的《Lua 程序设计（第 4 版）》就显得很有必要了。

 推荐那些常年要与 Lua 打交道的应用开发者都读一下这本由 Lua 创作者亲自编写的《Lua 程序设计（第 4 版）》，系统了解一下这门精致的语言，这不但对于深入理解并且使用好 Lua 有帮助，同时其设计哲学和思想也能在某种程度上开阔我们的视野。

Codedump

《Lua 设计与实现》作者





该书主要由本书作者本人独立撰写而成，感谢对我的支持和鼓励。书中涉及到的许多观点和结论都是本人的原创，但其中可能包含一些错误或遗漏，还请各位读者批评指正。书中的一些示例代码和实验数据可能会有误，敬请谅解。

译者序

本书由国内知名网络安全专家陈昊鹏编著，由机械工业出版社出版。陈昊鹏在网络安全领域有着丰富的经验，曾多次在国内外网络安全比赛中获奖，是网络安全领域的权威人士之一。本书内容丰富，深入浅出，非常适合网络安全爱好者阅读。

2016年2月，时年27岁的我因春节期间暴饮暴食导致急性胰腺炎入院治疗两个余月。当真正别无选择地终日躺在病床上时，就似乎不可避免地开始面对和尝试回答那个亘古不变的问题：“假设有一天我死了，究竟能够留下什么？”

Lua语言从1993年诞生至今已20余年，是开源嵌入式脚本语言领域中一门独树一帜的语言，在包括嵌入式、物联网、游戏、游戏外挂、软件安全、逆向工程乃至机器学习等领域中均具有不可替代的重要地位和极为广泛的应用。截至2017年7月，Lua语言在IEEE Spectrum编程语言排行榜中名列第21位（<http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017>），在TIOBE排行榜中名列第27位（The TIOBE Programming Community index, <https://www.tiobe.com/tiobe-index>）。近年来，除了游戏领域典型的应用外，包括Redis、Nginx/OpenResty、NMAP、WOW、OpenWRT、PhotoShop等大量的著名产品也均使用Lua作为其嵌入式脚本引擎，以供开发者进行功能扩展和二次开发等。伴随着移动互联网、DevOps等的迅猛发展，Lua语言在包括热更新、不停机部署等的实现方面也提供了一种现实的解决方案（例如银行等金融应用中某些采用C语言编写的性能密集型核心交易逻辑）。在Lua语言多年的发展过程中，也有大量的第三方机构对Lua语言进行了多方面的改进和增强，诸如LuaJIT等的发展也十分迅速。

除了语言本身的使用外，从语言的实现、原理、设计哲学等角度看，Lua语言也是学习编译原理、虚拟机、脚本引擎等的重要参考和现实依据，可以成为相关领域教科书式的典范，在国外也一直是部分高校计算机专业开展相关课程时的重要学习对象之一。在游戏领域，深入学习Lua语言后进行消化、吸收、优化、重构、增强甚至基于Lua语言的思想重新开发一种脚本语言或一个脚本引擎的例子屡见不鲜；在软件安全领域，基于虚拟机的思想设计出的各类混淆、VM保护产品更是有无数的先例；在各种灰色产业中，Lua语言也同样扮演了更加鲜为人知的重要角色。

我在研究生期间学习嵌入式和游戏逆向领域的过程中涉猎了Lua语言，之后由于在工作中需要编写Nginx和Redis的Lua脚本（Redis的Lua脚本在原子性、执行效率等方面具有显著优势）才开始对Lua语言进行深入的学习。然而，在学习Lua语言的过程中，我发现国





内对 Lua 语言的应用仍主要集中在传统的游戏领域，市面上有限的几本在售书籍也主要针对 Lua 语言在游戏开发中的使用，并没有一本书从语言本身的维度进行系统性介绍。目前网络上的各类中文资料、教程、手册也大多是碎片化的，而且面向的还主要是 Lua 语言的陈旧版本。对于有一定专业素养的从业人员而言，通常可以通过文档或速成式的教程在数天或数周内基本掌握一门语言，对于非计算机专业的开发人员或一般使用者则会难些。但我认为即便只考虑专业从业者，也需要一本权威、系统且工具性的书籍对 Lua 语言进行全面的介绍，以帮助实现低成本地快速学习和快速上手。此外，从事游戏逆向等软件安全领域的人士也有快速建立对 Lua 语言认知甚至进行深入学习的必要。

2006 年左右，本书的英文第 2 版出版后，国内出版了其中文译本，但至今已经超过 10 年，且本书的英文第 3 版也已经对全书的内容进行了重大重构，最新的 Lua 5.3 也发生了较大的变化，因此之前的中文第 2 版和网络上流传的影印版 PDF 均已经不能满足读者现有的需求。在这样的情况下，加上机缘的巧合，我于 2016 年 11 月开始与电子工业出版社博文视点的符隆美编辑一起联系了远在巴西的 Lua 语言的作者，并最终从国内诸多出版商和译者中杀出重围，艰难地争取到了作者的翻译版授权。

目前 Lua 语言在国内的发展不像 Python 语言、R 语言等为人熟知，也不似 Go 语言等站在风口浪尖，但 Lua 语言在国外却一直保持着持续性的演进，在过去 20 年间表现出了极为强大的生命力（随便举一个嵌入式领域 OpenWrt 路由器操作系统的例子，目前在各大主流路由器品牌或 KOS/小博无线等商业 WIFI 服务商中均扮演着不可替代的重要作用）。我相信，尽管略显小众，译文中也难免有值得商榷之处，这样一本针对 Lua 语言最新版本的权威、系统性的中文译本都应该能够为游戏、嵌入式、物联网、逆向工程、软件安全、移动互联网、C 语言核心系统开发工程师等诸多领域的学生、爱好者和从业人员提供些许帮助——而这也是我作为一名计算机行业从业人员的愿望。

在开始本书的翻译工作前，我自诩具有尚可的文字感知和表达能力，在多年的学习和工作中也阅读过计算机行业多个不同领域的大量中英文文档，力图以“信、达、雅”的原则要求自己，从一名计算机行业一线从业者的角度，在尽可能正确地理解了原著英文意思后，用尽可能专业的语言进行表述，避免出现读者“感觉还不如直接去看英本原版”的情况。但是，2017 年 4 月 20 日我拿到本书的部分原稿并开始着手翻译后，我发现“信、达、雅”三者间做好平衡着实不是一件易事。受精力和能力所限，我也并未在实际生产代码中使用过原著中讲解的所有机制，所以译文中也一定会有诸多不妥、失误甚至错误，如果读者有任何意见或建议可以直接通过我的邮箱（mlkui@163.com）或 QQ 读者交流群（QQ 群号：662640785）联系我，我会虚心接受一切批评和指正。





最后，我要感谢对本书的出版做出了直接和间接贡献的人。

感谢我的父母、妻子及亲人们多年来给予的无限关心、支持和陪伴，你们是我今天幸福生活的缔造者和组成者，也是我奋斗的根本动力和首要原因。

感谢我不便在此一一列举的领导和同事们，感谢他们一直以来在工作和生活中给予的无限支持、认可、包容和指点，尽管他们中的有一些已经离开。

感谢中学、本科及研究生的朋友、同学、老师和团队，感谢他们多年以来给予的陪伴、认可和信任，也祝愿我们在未来有机会携手共创辉煌。

感谢电子工业出版社及其计算机图书分社博文视点，感谢博文视点符隆美编辑的认可和信任，感谢他们在本书引进并最终出版发行全过程中的卓越眼光和艰辛努力。

这是我真正署名的第一本技术书籍，我会把本书微薄的版税收入全部用于 Lua 语言中文官方网站 <http://www.lua-lang.org.cn> 的日常服务器及带宽开支，并捐献给其他投入国内 Lua 语言推广的相关组织和活动。我想，我终于可以给这个世界留下点什么了。

真诚希望我的劳动能够帮助更多有需要的人，帮助他们创造更多的价值！

梅隆魁

2017 年 7 月于北京





前言

1993 年，当我和 Waldemar、Luiz 开发 Lua 语言时，我们并没有想象到它会像今天这样被如此广泛地使用。当年，Lua 语言只是为了两个特定项目而开发的实验室项目；如今，Lua 语言被大量应用于需要一门简明、可扩展、可移植且高效的脚本语言的领域中，例如嵌入式系统、移动设备、物联网，当然还有游戏。

Lua 语言从一开始就被设计为能与 C/C++ 及其他常用语言开发的软件集成在一起使用的语言，这种设计带来了非常多的好处。一方面，Lua 语言不需要在性能、与三方软件交互等 C 语言已经非常完善的地方重复“造轮子”，可以直接依赖 C 语言实现上述特性，因而 Lua 语言非常精简；另一方面，通过引入安全的运行时环境、自动内存管理、良好的字符串处理能力和可变长的多种数据类型，Lua 语言弥补了 C 语言在非面向硬件的高级抽象能力、动态数据结构、鲁棒性、调试能力等方面不足。

Lua 语言强大的原因之一就在于它的标准库，这不是偶然，毕竟扩展性本身就是 Lua 语言的主要能力之一。Lua 语言中的许多特性为扩展性的实现提供了支持：动态类型使得一定程度的多态成为了可能，自动内存管理简化了接口的实现（无须关心内存的分配/释放及处理溢出），作为第一类值的函数支持高度的泛化，从而使得函数更加通用。

Lua 语言除了是一门可扩展的语言外，还是一门胶水语言 (*glue language*)。Lua 语言支持组件化的软件开发方式，通过整合已有的高级组件构建新的应用。这些组件通常是通过 C/C++ 等编译型强类型语言编写的，Lua 语言充当了整合和连接这些组件的角色。通常，组件（或对象）是对程序开发过程中相对稳定逻辑的具体底层（如小部件和数据结构）的抽象，这些逻辑占用了程序运行时的大部分 CPU 时间，而产品生命周期中可能经常发生变化的逻辑则可以使用 Lua 语言来实现。当然，除了整合组件外，Lua 语言也可以用来适配和改造组件，甚至创建全新的组件。

诚然，Lua 语言并非这个世界上唯一的脚本语言，还有许多其他的脚本语言提供了类似的能力。尽管如此，Lua 语言的很多特性使它成为解决许多问题的首选，这些特性如下。

可扩展： Lua 语言具有卓越的可扩展性。Lua 的可扩展性好到很多人认为 Lua 超越了编程语言的范畴，其甚至可以成为一种用于构建领域专用语言 (Domain-Specific Language)，



DSL) 的工具包。Lua 从一开始就被设计为可扩展的，既支持使用 Lua 语言代码来扩展，也支持使用外部的 C 语言代码来扩展。在这一点上有一个很好的例证：Lua 语言的大部分基础功能都是通过外部库实现的。我们可以很容易地将 Lua 与 C/C++、Java、C# 和 Python 等结合在一起使用。

简明： Lua 语言是一门精简的语言。尽管它本身具有的概念并不多，但每个概念都很强大。这样的特性使得 Lua 语言的学习成本很低，也有助于减小其本身的大小（其包含所有标准库的 Linux 64 位版本仅有 220 KB）。

高效： Lua 语言的实现极为高效。独立的性能测试说明 Lua 语言是脚本语言中最快的语言之一。

可移植： Lua 语言可以运行在我们听说过的几乎所有平台之上，包括所有的类 UNIX 操作系统（Linux、FreeBSD 等）、Windows、Android、iOS、OS X、IBM 大型机、游戏终端（PlayStation、Xbox、Wii 等）、微处理器（如 Arduino）等。针对所有这些平台的源码本质上是一样的，Lua 语言遵循 ANSI (ISO) C 标准，并未使用条件编译来对不同平台进行代码的适配。因此，当需要适配新平台时，只要使用对应平台下的 ISO C 编译器重新编译 Lua 语言的源码就可以了。

预期读者

除了本书的最后一部分（其中讨论了 Lua 语言的 C 语言 API）外，阅读本书并不需要对 Lua 语言或其他任何一种编程语言有预先了解。不过，阅读本书的确需要了解一些基本的编程概念，尤其是变量与赋值、控制结构、函数与参数、流与文件及数据结构等。

Lua 语言有三类典型用户：在应用程序中嵌入式地使用 Lua 语言的用户、单独使用 Lua 语言的用户，以及和 C 语言一起使用 Lua 语言的用户。

诸如 Adobe Lightroom、Nmap 和魔兽世界等在内的许多应用程序中嵌入式地使用了 Lua 语言。这些应用使用 Lua 语言的 C 语言 API 去注册新函数、创建新类型和改变部分运算符的行为，以最终达到将 Lua 语言用于特定领域的目的。一般情况下，这些应用的用户根本感受不到 Lua 语言其实是一门被用于特定领域的独立编程语言。例如，Lightroom 插件的很多开发者压根儿不知道他们使用的是 Lua 语言，Nmap 的用户也倾向于将 Lua 语言视为 Nmap 脚本引擎所使用的语言，魔兽世界的很多玩家也认为 Lua 语言是这个游戏所独有的。尽管应用场景各异，Lua 语言的核心是相同的，本书中将要讲的编程技巧也都是适用的。

除了用于文本处理和用后即弃的小程序外，作为一门独立的编程语言，Lua 语言也同样适用于大中型项目。对于这些应用而言，Lua 语言的主要能力源于标准库。例如，标准库提供了模式匹配和其他字符串处理函数。随着 Lua 语言不断改进对标准库的支持，第三方库的数量在不断增加。LuaRocks 是一个 Lua 语言模块的部署和管理系统，该系统在 2015 年管理了 1000 多个涵盖各个领域的模块。

最后，还有一部分程序员会在编写程序时将 Lua 语言当作 C 语言的一个标准库来使用。他们通常更多地用 C 语言（相对于 Lua 语言）来进行编码，但是只有较好地理解了 Lua 语言才能写出简单易用且便于二者集成的接口。

全书结构

本书的这一版本增加了针对很多领域的新内容和示例，包括沙盒、协程以及日期和时间处理。此外，还增加了 Lua 5.3 的相关内容，包括整型值、位运算及无符号整型值等。

更具体地说，这一版对全书结构进行了重大的重构。在本版中，笔者尝试围绕编程中的常见主题来组织内容，而不是围绕编程语言（例如分章节介绍每个标准库）去组织内容。新的组织方式来自于 Lua 语言教学的实际经验，它能帮助读者从简单的主题开始循序渐进地学习。特别地，笔者认为这一版的组织方式让本书成为了 Lua 语言相关课程的一份更好的教学资源。

和前几版一样，本书共由 4 个部分组成，每个部分包括 9 章左右的内容，各有侧重。

第1部分涵盖了 Lua 语言的基础知识（因此这一部分被命名为语言基础），主要围绕数值、字符串、表和函数等几种主要数据类型，也对基本输入/输出模型和 Lua 语言的整体语法进行了介绍。

第2部分为编程实操，涵盖了在其他类似编程语言中也经常涉及的高级主题，如闭包、模式匹配、时间和日期处理、数据结构、模块和错误处理等。

第3部分为语言特性。顾名思义，这一部分介绍了 Lua 语言与其他语言相比的不同之处，如元表及其使用、环境、弱引用表、协程和反射等高级特性。

最后，和以前的版本一样，本书的第4部分介绍了 Lua 语言和 C 语言之间的 API，以便于使用 C 语言的开发者能够发挥出 Lua 语言的全部能力。由于在这一部分中将使用 C 语言而非 Lua 语言进行编程，所以这一部分和本书的其他部分大相径庭。一些读者可能对 Lua 语言的 C 语言 API 毫无兴趣，而其他一些读者可能觉得这一部分是本书中最有意义的部分。

在本书的所有部分中，我们都专注于不同的语言结构，并且使用了大量的示例和练习来演示如何将这些语言结构应用于实际需求中。在一些章节之间，我们也设置了几个“插曲”，每个“插曲”都提供了一个简短但完整的 Lua 语言程序，以帮助读者建立对 Lua 语言的更多整体认识。

其他资源

官方文档是所有真正希望学习一门语言的人所必须具备的资料。本书无意取代 Lua 语言官方文档。相反，本书是对官方文档的补充。一方面，官方文档只描述了 Lua 语言，其中既没有代码实例，也没有语言结构的基本原理。另一方面，官方文档覆盖了 Lua 语言的所有内容，本书则跳过了 Lua 语言中的一些极少使用的边边角角。此外，官方文档是有关 Lua 语言最权威的文档，本书中任何与官方文档不同的地方都应该以官方文档为准。我们可以在 Lua 语言的官方网站 <http://www.lua.org> 上找到官方文档和其他的更多内容。

此外，在 Lua 语言的用户社区 <http://lua-users.org> 中也有不少有用的信息。与其他资源相比，用户社区提供了教程、第三方库列表、文档以及 Lua 语言官方邮件列表的存档等资料。

本书内容基于 Lua 5.3 版本，不过书中的大部分内容对于老版本和可能的后续版本同样适用。Lua 5.3 和 Lua 5.x 老版本之间的区别都已经被清晰地描述了出来；如果读者使用的是本书出版后更新的版本，那么也可以在官方文档中找到相应版本之间的具体差异。

排版约定

在本书中，我们使用双引号表示字符串常量（如 "literal strings"），使用单引号表示单个字符（如 'a'）。用作模式的字符串也会被单引号引起，例如 '[%w_]*'。此外，代码段（chunks of code）和标识符（identifier）使用等宽字体，强调的内容则使用粗体。大段代码使用如下格式给出：

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

记号--> 表示一条语句的输出或表达式求值的结果：

```
print(10)      --> 10
```

```
13 + 3      --> 16
```

由于 Lua 语言中两个连续的连字符（--）表示单行注释，因此在程序中使用记号--> 不会有任何问题。

本书前几章中有一些代码需要在交互模式下输入，对于这种情况下的每一行代码，我们使用 Lua 语言的提示符（">"）进行了标注：

```
> 3 + 5      --> 8
> math.sin(2.3)  --> 0.74570521217672
```

在 Lua 5.2 及更早版本中，如果需要打印表达式求值的结果，必须在每个表达式前加上一个等号：

```
> = 3 + 5      --> 8
> a = 25
> = a          --> 25
```

为了向下兼容，Lua 5.3 也允许这种语法结构。

最后，本书使用符号<--> 表示两者完全等价：

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

运行示例

运行本书中的示例需要使用 Lua 语言解释器。尽管理想情况下应该使用 Lua 5.3 版本，但本书中的大部分示例无须修改也能在旧版本中运行。

可以从 Lua 语言的官网 (<http://www.lua.org>) 上下载解释器的源码。如果读者知道如何使用 C 语言编译器在自己的机器上编译 C 代码，那么建议尝试从源码编译并安装 Lua 语言（这非常简单）。*Lua Binaries* 网站（搜索 luabinaries）为大多数主流平台提供了已经编译好的 Lua 语言解释器。如果读者使用的是 Linux 或者其他的类 UNIX 操作系统，那么通常在软件库中已经提供了 Lua 语言执行环境，很多个发行版中已经提供了 Lua 语言相关的包。

Lua 语言有几种集成开发环境（IDE），在搜索引擎中搜一下就可以找到（尽管如此，笔者还是推荐 Windows 下的命令行接口或者其他操作系统下的文本编辑器，尤其是对于初学者而言）。

致谢

从本书第 1 版发行到现在已经十余年了。在这十余年间，很多朋友和机构都给予过很多帮助。

像过去一样，Luiz Henrique de Figueiredo、Waldemar Celes 和 Lua coauthors 给予了很多帮助。Reuben Thomas、Robert Day、André Carregal、Asko Kauppi、Brett Kapilik、Diego Nehab、Edwin Moragas、Fernando Jefferson、Gavin Wraith、John D. Ramsdell 和 Norman Ramsey 提出了无数的建议，也为丰富本书第 4 版的内容提供了很多启发。Luiza Novaes 为本书的封面设计提供了关键性的支持。

感谢 Lightning Source 公司在本书印刷和发行过程中表现出的可靠和高效。如果没有他们的帮助，自己出版本书几乎是不可能的。

感谢 Marcelo Gattass 领导的 Tecgraf，他们从 1993 年 Lua 项目诞生到 2005 年期间一直为 Lua 语言提供资助，并且仍在以很多方式持续地推动 Lua 语言的发展。

我还要感谢里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro, PUC-Rio）和巴西国家研究理事会（Brazilian National Research Council, CNPq），感谢他们对我工作的一贯支持。如果没有 PUC-Rio 为我提供的环境，那么 Lua 语言项目的开发根本不可能进行。

最后，我必须向 Noemi Rodriguez 表达我最诚挚的感谢（包括技术方面和非技术方面），感谢她点亮了我的生活。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/33804>



目录

第1部分 语言基础	1
1 Lua 语言入门	2
1.1 程序段	3
1.2 一些词法规范	5
1.3 全局变量	7
1.4 类型和值	7
1.4.1 nil	8
1.4.2 Boolean	8
1.5 独立解释器	10
1.6 练习	12
2 小插曲：八皇后问题	13
2.1 练习	15
3 数值	17
3.1 数值常量	17
3.2 算术运算	19
3.3 关系运算	21
3.4 数学库	21
3.4.1 随机数发生器	22
3.4.2 取整函数	22
3.5 表示范围	24

3.6 惯例	25
3.7 运算符优先级	26
3.8 兼容性	27
3.9 练习	28
4 字符串	30
4.1 字符串常量	31
4.2 长字符串/多行字符串	33
4.3 强制类型转换	34
4.4 字符串标准库	36
4.5 Unicode 编码	39
4.6 练习	41
5 表	43
5.1 表索引	44
5.2 表构造器	46
5.3 数组、列表和序列	48
5.4 遍历表	50
5.5 安全访问	52
5.6 表标准库	53
5.7 练习	54
6 函数	56
6.1 多返回值	58
6.2 可变长参数函数	61
6.3 函数 table.unpack	64
6.4 正确的尾调用	65
6.5 练习	66

7 输入输出	68
7.1 简单 I/O 模型	68
7.2 完整 I/O 模型	72
7.3 其他文件操作	74
7.4 其他系统调用	75
7.4.1 运行系统命令	75
7.5 练习	77
8 补充知识	78
8.1 局部变量和代码块	78
8.2 控制结构	80
8.2.1 if then else	81
8.2.2 while	81
8.2.3 repeat	82
8.2.4 数值型 for	82
8.2.5 泛型 for	83
8.3 break 、 return 和 goto	84
8.4 练习	88
第2部分 编程实操	91
9 闭包	92
9.1 函数是第一类值	93
9.2 非全局函数	95
9.3 词法定界	97
9.4 小试函数式编程	101
9.5 练习	104

10 模式匹配	106
10.1 模式匹配的相关函数	106
10.1.1 函数 string.find	106
10.1.2 函数 string.match	107
10.1.3 函数 string.gsub	108
10.1.4 函数 string.gmatch	108
10.2 模式	109
10.3 捕获	113
10.4 替换	115
10.4.1 URL 编码	117
10.4.2 制表符展开	119
10.5 诀窍	120
10.6 练习	124
11 小插曲：出现频率最高的单词	125
11.1 练习	127
12 日期和时间	129
12.1 函数 os.time	130
12.2 函数 os.date	131
12.3 日期和时间处理	133
12.4 练习	135
13 位和字节	136
13.1 位运算	136
13.2 无符号整型数	137
13.3 打包和解包二进制数据	140
13.4 二进制文件	143
13.5 练习	145

14 数据结构	146
14.1 数组	147
14.2 矩阵及多维数组	147
14.3 链表	150
14.4 队列及双端队列	151
14.5 反向表	152
14.6 集合与包	153
14.7 字符串缓冲区	155
14.8 图形	157
14.9 练习	159
15 数据文件和序列化	160
15.1 数据文件	161
15.2 序列化	163
15.2.1 保存不带循环的表	166
15.2.2 保存带有循环的表	168
15.3 练习	170
16 编译、执行和错误	172
16.1 编译	172
16.2 预编译的代码	176
16.3 错误	178
16.4 错误处理和异常	180
16.5 错误信息和栈回溯	181
16.6 练习	183
17 模块和包	185
17.1 函数 require	186
17.1.1 模块重命名	188

17.1.2 搜索路径	189
17.1.3 搜索器	191
17.2 Lua 语言中编写模块的基本方法	191
17.3 子模块和包	194
17.4 练习	195
第 3 部分 语言特性	197
18 迭代器和泛型 for	198
18.1 迭代器和闭包	198
18.2 泛型 for 的语法	201
18.3 无状态迭代器	202
18.4 按顺序遍历表	204
18.5 迭代器的真实含义	206
18.6 练习	207
19 小插曲：马尔可夫链算法	209
19.1 练习	212
20 元表和元方法	213
20.1 算术运算相关的元方法	214
20.2 关系运算相关的元方法	217
20.3 库定义相关的元方法	218
20.4 表相关的元方法	220
20.4.1 __index 元方法	220
20.4.2 __newindex 元方法	221
20.4.3 具有默认值的表	222
20.4.4 跟踪对表的访问	223
20.4.5 只读的表	225
20.5 练习	226

21 面向对象 (Object-Oriented) 编程	227
21.1 类 (Class)	229
21.2 继承 (Inheritance)	231
21.3 多重继承 (Multiple Inheritance)	233
21.4 私有性 (Privacy)	236
21.5 单方法对象 (Single-method Object)	238
21.6 对偶表示 (Dual Representation)	239
21.7 练习	241
22 环境 (Environment)	242
22.1 具有动态名称的全局变量	243
22.2 全局变量的声明	244
22.3 非全局环境	247
22.4 使用 _ENV	249
22.5 环境和模块	252
22.6 _ENV 和 load	253
22.7 练习	255
23 垃圾收集	256
23.1 弱引用表	256
23.2 记忆函数 (Memorize Function)	258
23.3 对象属性 (Object Attribute)	260
23.4 回顾具有默认值的表	261
23.5 瞬表 (Ephemeron Table)	262
23.6 析构器 (Finalizer)	263
23.7 垃圾收集器	267
23.8 控制垃圾收集的步长 (Pace)	268
23.9 练习	269

24 协程 (Coroutine)	271
24.1 协程基础	271
24.2 哪个协程占据主循环	274
24.3 将协程用作迭代器	277
24.4 事件驱动式编程	280
24.5 练习	285
25 反射 (Reflection)	286
25.1 自省机制 (Introspective Facility)	287
25.1.1 访问局部变量	289
25.1.2 访问非局部变量	290
25.1.3 访问其他协程	292
25.2 钩子 (Hook)	293
25.3 调优 (Profile)	294
25.4 沙盒 (Sandbox)	297
25.5 练习	301
26 小插曲：使用协程实现多线程	302
26.1 练习	307
第 4 部分 C 语言 API	309
27 C 语言 API 总览	310
27.1 第一个示例	311
27.2 栈	314
27.2.1 压入元素	315
27.2.2 查询元素	316
27.2.3 其他栈操作	319
27.3 使用 C API 进行错误处理	322

目录

27.3.1 处理应用代码中的错误	322
27.3.2 处理库代码中的错误	323
27.4 内存分配	324
27.5 练习	326
28 扩展应用	327
28.1 基础知识	327
28.2 操作表	329
28.2.1 一些简便方法	333
28.3 调用 Lua 函数	335
28.4 一个通用的调用函数	336
28.5 练习	340
29 在 Lua 中调用 C 语言	341
29.1 C 函数	341
29.2 延续 (Continuation)	344
29.3 C 模块	347
29.4 练习	349
30 编写 C 函数的技巧	351
30.1 数组操作	351
30.2 字符串操作	353
30.3 在 C 函数中保存状态	357
30.3.1 注册表	357
30.3.2 上值	360
30.3.3 共享的上值 (Shared upvalue)	363
30.4 练习	364

Lua 程序设计 (第 4 版)

31 C 语言中的用户自定义类型	365
31.1 用户数据 (Userdata)	366
31.2 元表 (Metatable)	369
31.3 面向对象访问	372
31.4 数组访问	374
31.5 轻量级用户数据	376
31.6 练习	377
32 管理资源	378
32.1 目录迭代器	378
32.2 XML 解析器	382
32.3 练习	392
33 线程和状态	394
33.1 多线程	394
33.2 Lua 状态	399
33.3 练习	408

語音識別

第1部分

TLK首字母

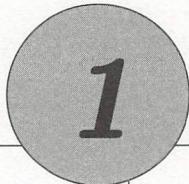


中國出生率連續下降，這和出生率的下降一樣，是中國人口增長速度減緩的主要原因之一。這和中國政府的計劃生育政策有關，該政策自1979年開始實施，旨在控制中國的人口增長。

這項政策已經取得了顯著的成效，中國的出生率從1979年的2.8‰下降到了2018年的1.1‰。這項政策的實施，使得中國的人口增長速度減緩，並在一定程度上緩解了中國的資源壓力。

然而，這項政策也引起了許多爭議。一些人認為，這項政策違反了人權，並導致了中國社會的不平等。另一些人則認為，這項政策是必要的，因為中國的資源有限，人口過多會導致資源匱乏。

總體來說，中國的出生率連續下降是一個複雜的問題，需要考慮多方面的因素。這項政策的實施，既有利也有弊，需要在尊重人權的前提下，尋找一個合理的平衡點。



1

Lua 语言入门

遵照惯例，我们的第一个 Lua 程序是通过标准输出打印字符串 "Hello World"：

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

如果读者使用的是 Lua 语言独立解释器（stand-alone interpreter），要运行这第一个程序的话，直接调用解释器（通常被命名为 `lua` 或者 `lua5.3`）运行包含程序代码的文本文件就可以了。例如，如果把上述代码保存为名为 `hello.lua` 的文件，那么可以通过以下命令运行：

```
% lua hello.lua
```

再来看一个稍微复杂点的例子，以下代码定义了一个计算阶乘的函数，该函数先让用户输入一个数，然后打印出这个数的阶乘结果：

```
-- 定义一个计算阶乘的函数
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact(n - 1)
    end
end

print("enter a number:")
```

```
a = io.read("*n")           -- 读取一个数字
print(fact(a))
```

1.1 程序段

我们将 Lua 语言执行的每一段代码（例如，一个文件或交互模式下的一行）称为一个程序段（*Chunk*），即一组命令或表达式组成的序列。

程序段既可以简单到只由一句表达式构成（例如输出“Hello World”的示例），也可以由多句表达式和函数定义（实际是赋值表达式，后面会详细介绍）组成（例如计算阶乘的示例）。程序段在大小上并没有限制，事实上，由于 Lua 语言也可以被用作数据定义语言，所以几 MB 的程序段也很常见。Lua 语言的解释器可以支持非常大的程序段。

除了将源码保存成文件外，我们也可以直接在交互式模式（interactive mode）下运行独立解释器（stand-alone interpreter）。当不带参数地调用 `lua` 时，可以看到如下的输出：

```
% lua
Lua 5.3 Copyright (C) 1994-2016 Lua.org, PUC-Rio
>
```

此后，输入的每一条命令（例如：`print "Hello World"`）都会在按下回车键后立即执行。我们可以通过输入 EOF 控制字符（End-Of-File、POSIX 环境下使用 `ctrl-D`，Windows 环境下使用 `ctrl-Z`），或调用操作系统库的 `exit` 函数（执行 `os.exit()`）退出交互模式。

从 Lua 5.3 版本开始，可以直接在交互模式下输入表达式，Lua 语言会输出表达式的值，例如：

```
% lua
Lua 5.3 Copyright (C) 1994-2016 Lua.org, PUC-Rio
> math.pi / 4      --> 0.78539816339745
> a = 15
> a^2            --> 225
> a + 2          --> 17
```

与之相比，在 Lua 5.3 之前的老版本中，需要在表达式前加上一个等号：

```
% lua5.2
Lua 5.2.3 Copyright (C) 1994-2013 Lua.org, PUC-Rio
```

Lua 程序设计（第 4 版）

```
> a = 15
> = a^2           --> 225
```

为了向下兼容，Lua 5.3 也支持这种语法结构。

要以代码段的方式运行代码（不在交互模式下），那么必须把表达式包在函数 `print` 的调用中：

```
print(math.pi / 4)
a = 15
print(a^2)
print(a + 2)
```

在交互模式下，Lua 语言解释器一般会把我们输入的每一行当作完整的程序块或表达式来解释执行。但是，如果 Lua 语言解释器发现我们输入的某一行不完整，那么它会等待直到程序块或表达式被输入完整后再进行解释执行。这样，我们也可以直接在交互模式下输入一个像阶乘函数示例那样的由很多行组成的多行定义。不过，对于这种较长的函数定义而言，将其保存成文件然后再调用独立解释器来执行通常更方便。

我们可以使用 `-i` 参数让 Lua 语言解释器在执行完指定的程序段后进入交互模式：

```
% lua -i prog
```

上述的命令行会在执行完文件 `prog` 中的程序段后进入交互模式，这对于调试和手工测试很有用。在本章的最后，我们会学习有关独立解释器的更多参数。

另一种运行程序段的方式是调用函数 `dofile`，该函数会立即执行一个文件。例如，假设我们有一个如下所示的文件 `lib1.lua`：

```
function norm (x, y)
    return math.sqrt(x^2 + y^2)
end

function twice (x)
    return 2.0 * x
end
```

然后，在交互模式下运行：

```
> dofile("lib1.lua")      -- 加载文件①
```

^① 即加载库，但联系上下文，这里应该是“加载文件”。

```
> n = norm(3.4, 1.0)
> twice(n)           --> 7.0880180586677
```

函数 `dofile` 在开发阶段也非常有用。我们可以同时打开两个窗口，一个窗口中使用文件编辑器编辑的代码（例如文件 `prog.lua`），另一个窗口中使用交互模式运行 Lua 语言解释器。当修改完代码并保存后，只要在 Lua 语言交互模式的提示符下执行 `dofile("prog.lua")` 就可以加载新代码，然后就可以观察新代码的函数调用和执行结果了。

1.2 一些词法规范

Lua 语言中的标识符（或名称）是由任意字母^①、数字和下画线组成的字符串（注意，不能以数字开头），例如：

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

“下画线 + 大写字母”（例如 `_VERSION`）组成的标识符通常被 Lua 语言用作特殊用途，应避免将其用作其他用途。我通常会将“下画线 + 小写字母”用作哑变量（Dummy variable）。

以下是 Lua 语言的保留字（reserved word），它们不能被用作标识符：

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

Lua 语言是对大小写敏感的，因而虽然 `and` 是保留字，但是 `And` 和 `AND` 就是两个不同的标识符。

Lua 语言中使用两个连续的连字符（--）表示单行注释的开始（从--之后直到此行结束都是注释），使用两个连续的连字符加两对连续左方括号表示长注释或多行注释的开始（直到两个连续的右括号为止，中间都是注释），例如：^②

^①译者注：在 Lua 语言的早期版本中，“字母”的概念与操作系统的区域（Locale）设置有关，因此可能导致同一个程序在更换区域设置后不能正确运行的情况。所以，在新版 Lua 语言中标识符中的“字母”仅允许使用 A-Z 和 a-z。

^②长注释可能比这更复杂，更多内容参见4.2节。

Lua 程序设计（第 4 版）

```
--[[ 多行
长注释
]]
```

在注释一段代码时，一个常见的技巧是将这些代码放入--[[和--]] 之间，例如：

```
--[[[
print(10)      -- 无动作（被注释掉了）
--]]]
```

当我们需要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，第一行的--[[表示一段多行注释的开始，直到遇到两个连续的右括号这段多行注释才会结束，因而尽管最后一行有两个连续的连字符，但由于这两个连字符在最后两个右方括号之前，所以仍然被注释掉了。在第二个示例中，由于第一行的--[[实际是单行注释，因此最后一行实际上也是一条独立的单行注释（最后的两个连续右方括号没有与之匹配的--[[），print 并没有被注释掉。

在 Lua 语言中，连续语句之间的分隔符并不是必需的，如果有需要的话可以使用分号来进行分隔。在 Lua 语言中，表达式之间的换行也不起任何作用。例如，以下 4 个程序段都是合法且等价的：

```
a = 1      +
b = a * 2

a = 1;
b = a * 2;

a = 1; b = a * 2

a = 1  b = a * 2      -- 可读性不佳，但是却是正确的
```

我个人的习惯只有在同一行中书写多条语句的情况下（这种情况一般也不会出现），才会使用分号做分隔符。



1.3 全局变量

在 Lua 语言中，全局变量（Global Variable）无须声明即可使用，使用未经初始化的全局变量也不会导致错误。当使用未经初始化的全局变量时，得到的结果是 nil：

```
> b          --> nil
> b = 10
> b          --> 10
```

当把 nil 赋值给全局变量时，Lua 会回收该全局变量（就像该全局变量从来没有出现过一样），例如：

```
> b = nil
> b          --> nil
```

Lua 语言不区分未初始化变量和被赋值为 nil 的变量。在上述赋值语句执行后，Lua 语言会最终回收该变量占用的内存。

1.4 类型和值

Lua 语言是一种动态类型语言（Dynamically-typed language），在这种语言中没有类型定义（type definition），每个值都带有其自身的类型信息。

Lua 语言中有 8 种基本类型：nil（空）、boolean（布尔）、number（数值）、string（字符串）、userdata（用户数据）、function（函数）、thread（线程）和 table（表）。使用函数 type 可获取一个值对应的类型名称：

```
> type(nil)      --> nil
> type(true)     --> boolean
> type(10.4 * 3) --> number
> type("Hello world") --> string
> type(io.stdin)   --> userdata
> type(print)      --> function
> type(type)       --> thread
> type({})         --> table
> type(type(X))    --> string
```



Lua 程序设计（第 4 版）

不管 X 是什么，最后一行返回的永远是"string"。这是因为函数 type 的返回值永远是一个字符串。

userdata 类型允许把任意的 C 语言数据保存在 Lua 语言变量中。在 Lua 语言中，用户数据类型除了赋值和相等性测试外，没有其他预定义的操作。用户数据被用来表示由应用或 C 语言编写的库所创建的新类型。例如，标准 I/O 库使用用户数据来表示打开的文件。我们会在后面涉及 C API 时再讨论更多的相关内容。

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：

```
> type(a)          --> nil    ('a' 尚未初始化)
> a = 10
> type(a)          --> number
> a = "a string!!"
> type(a)          --> string
> a = nil
> type(a)          --> nil
```

一般情况下，将一个变量用作不同类型时会导致代码的可读性不佳；但是，在某些情况下谨慎地使用这个特性可能会带来一定程度的便利。例如，当代码发生异常时可以返回一个 nil 以区别于其他正常情况下的返回值。

本章接下来将学习简单类型 nil 和 Boolean，在后续的章节中我们会依次对 number（第3章）、string（第4章）、table（第5章）和 function（第6章）进行详细学习。我们会在第24章中学习 thread 类型。

1.4.1 nil

nil 是一种只有一个 nil 值的类型，它的主要作用就是与其他所有值进行区分。Lua 语言使用 nil 来表示无效值（non-value，即没有有用的值）的情况。像我们之前所学习到的，一个全局变量在第一次被赋值前的默认值就是 nil，而将 nil 赋值给全局变量则相当于将其删除。

1.4.2 Boolean

Boolean 类型具有两个值，*true* 和 *false*，它们分别代表了传统布尔值。不过，在 Lua 语言中，Boolean 值并非是用于条件测试的唯一方式，任何值都可以表示条件。在 Lua 语言中，



条件测试（例如控制结构中的分支语句）将除 Boolean 值 `false` 和 `nil` 外的所有其他值视为真。特别的是，在条件检测中 Lua 语言把零和空字符串也都视为真。

在本书中，“`false`”代表的是所有为假的值，包括 Boolean 类型的 `false` 或 `nil`；而“`false`”特指 Boolean 类型的值。“`true`”和“`true`”亦然。

Lua 语言支持常见的逻辑运算符：`and`、`or` 和 `not`。和条件测试一样，所有的逻辑运算将 Boolean 类型的 `false` 和 `nil` 当作假，而把其他值当作真。逻辑运算符 `and` 的运算结果为：如果它的第一个操作数为“`false`”，则返回第一个操作数，否则返回第二个操作数。逻辑运算符 `or` 的运算结果为：如果它的第一个操作数不为“`false`”，则返回第一个操作数，否则返回第二个操作数。例如：

```
> 4 and 5      --> 5
> nil and 13   --> nil
> false and 13  --> false
> 0 or 5       --> 0
> false or "hi"  --> "hi"
> nil or false  --> false
```

`and` 和 `or` 都遵循短路求值（Short-circuit evaluation）原则，即只在必要时才对第二个操作数进行求值。例如，根据短路求值的原则，表达式 (`i~=0 and a/i>b`) 不会发生运行时异常（当 `i` 等于 0 时，`a/i` 不会执行）。

在 Lua 语言中，形如 `x=x or v` 的惯用写法非常有用，它等价于：

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

即，当 `x` 未被初始化时，将其默认值设为 `v`（假设 `x` 不是 Boolean 类型的 `false`）。

另一种有用的表达式形如 `((a and b) or c)` 或 `(a and b or c)`（由于 `and` 的运算符优先级高于 `or`，所以这两种表达形式等价，后面会详细介绍），当 `b` 不为 `false` 时，它们还等价于 C 语言的三目运算符 `a?b:c`。例如，我们可以使用表达式 `(x>y) and x or y` 选出数值 `x` 和 `y` 中较大的一个。当 `x>y` 时，`and` 的第一个操作数为 `true`，与第二个操作数 (`x`) 进行 `and` 运算后结果为 `x`，最终与 `or` 运算后返回第一个操作数 `x`。当 `x>y` 不成立时，`and` 表达式的值为 `false`，最终 `or` 运算后的结果是第二个操作数 `y`。

`not` 运算符永远返回 Boolean 类型的值：

```
> not nil      --> true
> not false    --> true
```



```
> not 0      --> false
> not not 1  --> true
> not not nil --> false
```

1.5 独立解释器

独立解释器（Stand-alone interpreter，由于源文件名为 `lua.c`，所以也被称为 `lua.c`；又由于可执行文件为 `lua`，所以也被称为 `lua`）是一个可以直接使用 Lua 语言的小程序。这一节介绍它的几个主要参数。

如果源代码文件第一行以井号 (#) 开头，那么解释器在加载该文件时会忽略这一行。这个特征主要是为了方便在 POSIX 系统中将 Lua 作为一种脚本解释器来使用。假设独立解释器位于 `/usr/local/bin` 下，当使用下列脚本：

```
#!/usr/local/bin/lua
```

或

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

时，不需要显式地调用 Lua 语言解释器也可以直接运行 Lua 脚本。

`lua` 命令的完整参数形如：

```
lua [options] [script [args]]
```

其中，所有的参数都是可选的。如前所述，当不使用任何参数调用 `lua` 时，就会直接进入交互模式。

`-e` 参数允许我们直接在命令行中输入代码，例如：

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

请注意，在 POSIX 系统下需要使用双引号，以防止 Shell 错误地解析括号。

`-l` 参数用于加载库。正如之前提到的那样，`-i` 参数用于在运行完其他命令行参数后进入交互模式。因此，下面的命令会首先加载 `lib` 库，然后执行 `x=10` 的赋值语句，并最终进入交互式模式：

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



如果在交互模式下输入表达式，那么解释器会输出表达式求值后的结果：

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

请记住，这个特性只在 Lua 5.3 及之后的版本中才有效。在之前的版本中，必须在表达式前加上一个等号。如果不想输出结果，那么可以在行末加上一个分号：

```
> io.flush()          --> true
> io.flush();
```

分号使得最后一行在语法上变成了无效的表达式，但可以被当作有效的命令执行。

解释器在处理参数前，会查找名为 LUA_INIT_5_3 的环境变量，如果找不到，就会再查找名为 LUA_INIT 的环境变量。如果这两个环境变量中的任意一个存在，并且其内容为 @ filename，那么解释器就会运行相应的文件；如果 LUA_INIT_5_3（或者 LUA_INIT）存在但是不以 @ 开头，那么解释器就会认为其包含 Lua 代码，并会对其进行解释执行。由于可以通过上面的方法完整地配置 Lua，因而 LUA_INIT 使得我们可以灵活地配置独立解释器。例如，我们可以预先加载程序包（Package）、修改路径、定义自定义函数、对函数进行重命名或删除函数，等等。

我们可以通过预先定义的全局变量 arg 来获取解释器传入的参数。例如，当执行如下命令时：

```
% lua script a b c
```

编译器在运行代码前会创建一个名为 arg 的表，其中存储了所有的命令行参数。索引 0 中保存的内容为脚本名，索引 1 中保存的内容为第一个参数（本例中的 "a"），依此类推；而在脚本之前的所有选项则位于负数索引上，例如：

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

解释器按照如下的方式获取参数：

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



一般情况下，脚本只会用到索引为正数的参数（本例中的 `arg[1]` 和 `arg[2]`）。

Lua 语言也支持可变长参数，可以通过可变长参数表达式来获取。在脚本文件中，表达式... (3 个点) 表示传递给脚本的所有参数。我们将在 6.2 节中学习可变长参数的使用。

1.6 练习

练习 1.1：运行阶乘的示例并观察，如果输入负数，程序会出现什么问题？试着修改代码来解决问题。

练习 1.2：分别使用 `-l` 参数和 `dofile` 运行 `twice` 示例，并感受你喜欢哪种方式。

练习 1.3：你是否能举出其他使用“`--`”作为注释的语言？

练习 1.4：以下字符串中哪些是有效的标识符？

`--` `_end` `End` `end` `until?` `nil` `NULL` `one-step`

练习 1.5：表达式 `type(nil) == nil` 的值是什么？你可以直接在 Lua 中运行来得到答案，但是你能够解释原因吗？

练习 1.6：除了使用函数 `type` 外，如何检查一个值是否为 Boolean 类型？

练习 1.7：考虑如下的表达式：

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

其中的括号是否是必需的？你是否推荐在这个表达式中使用括号？

练习 1.8：请编写一个可以打印出脚本自身名称的程序（事先不知道脚本自身的名称）。



2

小插曲：八皇后问题

本章作为小插曲将讲解如何用 Lua 语言编写的简单但完整的程序来解决八皇后问题 (*eight-queen puzzle*, 其目标是把 8 个皇后合理地摆放在棋盘上, 让每个皇后之间都不能相互攻击)。

本书中给出的代码并不只适用于 Lua 语言, 只要稍加改动, 就能将代码转化成其他几种语言。之所以要在本章安排这个小插曲, 是为了在不深究细节的情况下, 先直观地呈现 Lua 语言的特点 (尤其是其大致语法结构)。我们会在后面的章节中学习所有缺失的细节。

要解决八皇后问题, 首先必须认识到每一行中只能有一个皇后。因此, 可以用一个由 8 个数字组成的简单数组 (一个数字对应一行, 代表皇后在这一行的哪一列) 来表示可能的解决方案。例如, 数组 {3, 7, 2, 1, 8, 6, 5, 4} 表示皇后在棋盘中的位置分别是 (1, 3)、(2, 7)、(3, 2)、(4, 1)、(5, 8)、(6, 4)、(7, 5) 和 (8, 4)。当然, 这个示例并不是一个正确的解, 例如 (3, 2) 中的皇后就可以攻击 (4, 1) 中的皇后。此外, 我们还必须认识到正确的解必须是整数 1 到 8 组成的排列 (Permutation), 这样才能保证每一列中也只有一个皇后。

完整的程序参见示例 2.1。

示例 2.1 求解八皇后问题的程序

```
N = 8      -- 棋盘大小

-- 检查(n, c)是否不会被攻击
function isplaceok (a, n, c)
```



```

for i = 1, n - 1 do    -- 对于每一个已经被放置的皇后
    if (a[i] == c) or           -- 同一列?
        (a[i] - i == c - n) or   -- 同一对角线?
        (a[i] + i == c + n) then -- 同一对角线?
        return false            -- 位置会被攻击
    end
end
return true      -- 不会被攻击; 位置有效
end

-- 打印棋盘
function printsolution (a)
    for i = 1, N do      -- 对每一行
        for j = 1, N do    -- 和每一列
            -- 输出 "X" 或 "-", 外加一个空格
            io.write(a[i] == j and "X" or "-", " ")
        end
        io.write("\n")
    end
    io.write("\n")
end

-- 把从 'n' 到 'N' 的所有皇后放在棋盘 'a' 上
function addqueen (a, n)
    if n > N then    -- 是否所有的皇后都被放置好了?
        printsolution(a)
    else    -- 尝试着放置第 n 个皇后
        for c = 1, N do
            if isplaceok(a, n, c) then
                a[n] = c    -- 把第 n 个皇后放在列 'c'
                addqueen(a, n + 1)
            end
        end
    end
end

```



```
-- 运行程序
addqueen({}, 1)
```

第一个函数是 `isplaceok`，该函数用来检查如果在棋盘上指定位置放置皇后，是否会受到之前被放置的皇后的攻击。更确切地说，该函数用来检查将第 n 个皇后放在第 c 列上时，是否会与之前已经被放置在数组 a 中的 $n-1$ 个皇后发生冲突。请注意，由于我们使用的表示方法保证了两个皇后不会位于同一行中，所以函数 `isplaceok` 只需检查新的位置上是否有皇后在同一列或对角线上即可。

接下来，我们使用函数 `printsolution` 打印出棋盘。该函数只是简单地遍历整个棋盘，在有皇后的位置输出 `X`，而在其他位置输出`-`，没有使用花哨的图形（注意 `and-or` 的用法）。每个摆放结果形如：

```
X - - - - -
- - - X - -
- - - - - X
- - - - X -
- - X - - -
- - - - - X -
- X - - - -
- - - X - - -
```

最后一个函数 `addqueen` 是这段程序的核心，该函数尝试着将所有大于等于 n 的皇后摆放在棋盘上，使用回溯法来搜索正确的解。首先，该函数检查当前解是否已经完成了所有皇后的摆放，如果已经完成则打印出当前解对应的摆放结果；如果还没有完成，则为第 n 个皇后遍历所有的列，将皇后放置在不会受到攻击的每一列上，并递归地寻找下一个皇后的可能摆放位置。

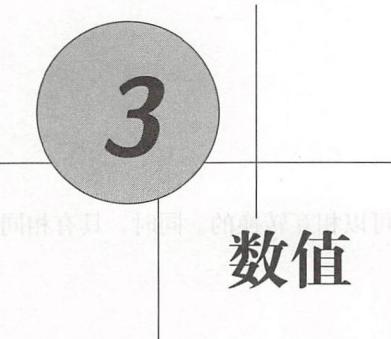
最后，代码在一个空白的解上^①调用 `addqueen` 开始进行求解。

2.1 练习

练习 2.1：修改八皇后问题的程序，使其在输出第一个解后即停止运行。

^①译者注：参数 {}。

练习 2.2：解决八皇后问题的另一种方式是，先生成 1~8 之间的所有排列，然后依次遍历这些排列，检查每一个排列是否是八皇后问题的有效解。请使用这种方法修改程序，并对比新程序与旧程序之间的性能差异（提示，比较调用 `isplaceok` 函数的次数）。



3 数值

在 Lua 5.2 及之前的版本中，所有的数值都以双精度浮点格式表示。从 Lua 5.3 版本开始，Lua 语言为数值格式提供了两种选择：被称为 *integer* 的 64 位整型和被称为 *float* 的双精度浮点类型（注意，在本书中“*float*”不代表单精度类型）。对于资源受限的平台，我们可以将 Lua 5.3 编译为精简 Lua (*Small Lua*) 模式，在该模式中使用 32 位整型和单精度浮点类型。^①

整型的引入是 Lua 5.3 的一个重要标志，也是与之前版本相比的主要区别。不过尽管如此，由于双精度浮点型能够表示最大为 2^{53} 的整型值，所以不会造成太大的不兼容性。我们接下来学习的大多数内容对于 Lua 5.2 及更早版本也同样适用。在本章末尾，我们会讨论兼容性方面的更多细节。

3.1 数值常量

我们可以使用科学计数法（一个可选的十进制部分外加一个可选的十进制指数部分）书写数值常量，例如：

> 4	--> 4
> 0.4	--> 0.4
> 4.57e-3	--> 0.00457

^①除了使用了 LUA_32BITS 宏定义以外，精简 Lua 和标准 Lua 的源码是一样的。除了数值表示占用的字节大小不一样外，精简 Lua 和标准 Lua 完全一致。

```
> 0.3e12          --> 300000000000.0
> 5E+20           --> 5e+20
```

具有十进制小数或者指数的数值会被当作浮点型值，否则会被当作整型值。

整型值和浮点型值的类型都是 "number"：

```
> type(3)      --> number
> type(3.5)    --> number
> type(3.0)    --> number
```

由于整型值和浮点型值的类型都是 "number"，所以它们是可以相互转换的。同时，具有相同算术值的整型值和浮点型值在 Lua 语言中是相等的：

```
> 1 == 1.0       --> true
> -3 == -3.0    --> true
> 0.2e3 == 200   --> true
```

在少数情况下，当需要区分整型值和浮点型值时，可以使用函数 `math.type`：

```
> math.type(3)    --> integer
> math.type(3.0)  --> float
```

在 Lua 5.3 中：

```
> 3              --> 3
> 3.0            --> 3.0
> 1000           --> 1000
> 1e3             --> 1000.0
```

Lua 语言像其他语言一样也支持以 `0x` 开头的十六进制常量。与其他很多编程语言不同，Lua 语言还支持十六进制的浮点数，这种十六进制浮点数由小数部分和以 `p` 或 `P` 开头的指数部分组成。^①例如：

```
> 0xff           --> 255
> 0xA3            --> 419
> 0x0.2           --> 0.125
> 0x1p-1           --> 0.5
> 0xa.bp2          --> 42.75
```

^①是在 Lua 5.2 中被引入的。

可以使用`%a`参数，通过函数`string.format`对这种格式进行格式化输出：

```
> string.format("%a", 419)           --> 0x1.a3p+8
> string.format("%a", 0.1)           --> 0x1.999999999999999ap-4
```

虽然这种格式很难阅读，但是这种格式可以保留所有浮点数的精度，并且比十进制的转换速度更快。

3.2 算术运算

除了加、减、乘、除、取负数（单目减法，即把减号当作一元运算符使用）等常见的算术运算外，Lua语言还支持取整除法（`floor`除法）、取模和指数运算。

对于Lua 5.3中引入的整型而言，主要的建议就是“开发人员要么选择忽略整型和浮点型二者之间的不同，要么就完整地控制每一个数值的表示。”^①因此，所有的算术操作符不论操作整型值还是浮点型值，结果都应该是一样的。

两个整型值进行相加、相减、相乘、相除和取负操作的结果仍然是整型值。对于这些算术运算而言，操作数是用整型还是用浮点型表示的整数都没有区别（除非发生溢出，参见3.5节）：

```
> 13 + 15           --> 28
> 13.0 + 15.0      --> 28.0
```

如果两个操作数都是整型值，那么结果也是整型值；否则，结果就是浮点型值。当操作数一个是整型值一个是浮点型值时，Lua语言会在进行算术运算前先将整型值转换为浮点型值：

```
> 13.0 + 25         --> 38.0
> -(3 * 6.0)        --> -18.0
```

由于两个整数相除的结果并不一定是整数（数学领域称为不能整除），因此除法不遵循上述规则。为了避免两个整型值相除和两个浮点型值相除导致不一样的结果，除法运算操作的永远是浮点数且产生浮点型值的结果：

```
> 3.0 / 2.0          --> 1.5
> 3 / 2              --> 1.5
```

^①参考 *Lua 5.3 Reference Manual*。

Lua 5.3 针对整数除法引入了一个称为 *floor* 除法的新算术运算符`//`。顾名思义，*floor* 除法会对得到的商向负无穷取整，从而保证结果是一个整数。这样，*floor* 除法就可以与其他算术运算一样遵循同样的规则：如果操作数都是整型值，那么结果就是整型值，否则就是浮点型值（其值是一个整数）。

```
> 3 // 2      --> 1
> 3.0 // 2    --> 1.0
> 6 // 2      --> 3
> 6.0 // 2.0  --> 3.0
> -9 // 2     --> -5
> 1.5 // 0.5  --> 3.0
```

以下公式是取模运算的定义：

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

如果操作数是整数，那么取模运算的结果也是整数。因此，取模运算也遵从与算术运算相同的规律，即如果两个操作数均是整型值，则结果为整型，否则为浮点型。

对于整型操作数而言，取模运算的含义没什么特别的，其结果的符号永远与第二个操作数的符号保持一致。特别地，对于任意指定的正常量 K ，即使 x 是负数，表达式 $x \% K$ 的结果也永远在 $[0, K - 1]$ 之间。例如，对于任意整型值 i ，表达式 $i \% 2$ 的结果均是 0 或 1。

对于实数类型的操作数而言，取模运算有一些不同。例如， $x - x \% 0.01$ 恰好是 x 保留两位小数的结果， $x - x \% 0.001$ 恰好是 x 保留三位小数的结果：

```
> x = math.pi
> x - x%0.01      --> 3.14
> x - x%0.001    --> 3.141
```

再比如，我们可以使用取模运算检查某辆车在拐过了指定的角度后是否能够原路返回。假设使用度作为角度的单位，那么我们可以使用如下的公式：

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

该函数对负的角度而言也同样适用：

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

假设使用弧度作为角度的单位，那么我们只需要简单地修改常量的定义即可：

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

表达式 `angle%(2*math.pi)` 实现了将任意范围的角度归一化到 $[0, 2\pi]$ 之间。

Lua 语言同样支持幂运算，使用符号 `^` 表示。像除法一样，幂运算的操作数也永远是浮点类型（整型值在幂运算时不能整除，例如， 2^{-2} 的结果不是整型值）。我们可以使用 `x^0.5` 来计算 `x` 的平方根，使用 `x^(1/3)` 来计算 `x` 的立方根。

3.3 关系运算

Lua 语言提供了下列关系运算：

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

这些关系运算的结果都是 Boolean 类型。

`==` 用于相等性测试，`~` 用于不等性测试。这两个运算符可以应用于任意两个值，当这两个值的类型不同时，Lua 语言认为它们是不相等的；否则，会根据它们的类型再对两者进行比较。

比较数值时应永远忽略数值的子类型，数值究竟是以整型还是浮点型类型表示并无区别，只与算术值有关（尽管如此，比较具有相同子类型的数值时效率更高）。

3.4 数学库

Lua 语言提供了标准数学库 `math`。标准数学库由一组标准的数学函数组成，包括三角函数（`sin`、`cos`、`tan`、`asin` 等）、指数函数、取整函数、最大和最小函数 `max` 和 `min`、用于生成伪随机数的伪随机数函数（`random`）以及常量 `pi` 和 `huge`（最大可表示数值，在大多数平台上代表 `inf`）。

所有的三角函数都以弧度为单位，并通过函数 `deg` 和 `rad` 进行角度和弧度的转换。

3.4.1 随机数发生器

函数 `math.random` 用于生成伪随机数，共有三种调用方式。当不带参数调用时，该函数将返回一个在 $[0, 1)$ 范围内均匀分布的伪随机实数。当使用带有一个整型值 n 的参数调用时，该函数将返回一个在 $[1, n]$ 范围内的伪随机整数。例如，我们可以通过调用 `random(6)` 来模拟掷骰子的结果。当使用带有两个整型值 l 和 u 的参数调用时，该函数返回在 $[l, u]$ 范围内的伪随机整数。

函数 `randomseed` 用于设置伪随机数发生器的种子，该函数的唯一参数就是数值类型的种子。在一个程序启动时，系统固定使用 1 为种子初始化伪随机数发生器。如果不设置其他的种子，那么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，然而，对于一个游戏来说却会导致相同的场景重复不断地出现。为了解决这个问题，通常调用 `math.randomseed(os.time())` 来使用当前系统时间作为种子初始化随机数发生器（后续 12.1 节中会对 `os.time` 进行介绍）。

3.4.2 取整函数

数学库提供了三个取整函数：`floor`、`ceil` 和 `modf`。其中，`floor` 向负无穷取整，`ceil` 向正无穷取整，`modf` 向零取整。当取整结果能够用整型表示时，返回结果为整型值，否则返回浮点型值（当然，表示的是整数值）。除了返回取整后的值以外，函数 `modf` 还会返回小数部分作为第二个结果。^①

```
> math.floor(3.3)          --> 3
> math.floor(-3.3)         --> -4
> math.ceil(3.3)           --> 4
> math.ceil(-3.3)          --> -3
> math.modf(3.3)           --> 3    0.3
> math.modf(-3.3)          --> -3   -0.3
> math.floor(2^70)          --> 1.1805916207174e+21
```

如果参数本身就是一个整型值，那么它将被原样返回。

如果想将数值 x 向最近的整数（nearest integer）取整，可以对 $x+0.5$ 调用 `floor` 函数。不过，当参数是一个很大的整数时，简单的加法可能会导致错误。例如，考虑如下的代码：

^①详见 6.1 节，Lua 语言支持一个函数返回多个值。

```
x = 2^52 + 1
print(string.format("%d %d", x, math.floor(x + 0.5)))
--> 4503599627370497 4503599627370498
```

$2^{52} + 1.5$ 的浮点值表示是不精确的，因此内部会以我们不可控制的方式取整。为了避免这个问题，我们可以单独地处理整数值：

```
function round (x)
    local f = math.floor(x)
    if x == f then return f
    else return math.floor(x + 0.5)
    end
end
```

上例中的函数总是会向上取整半个整数（例如 2.5 会被取整为 3）。如果想进行无偏取整（unbiased rounding），即向距离最近的偶数取整半个整数，上述公式在 $x+0.5$ 是奇数的情况下会产生不正确的结果：

```
> math.floor(3.5 + 0.5)      --> 4   (ok)
> math.floor(2.5 + 0.5)      --> 3   (wrong)
```

这时，还是可以利用取整操作来解决上述公式中存在的问题：表达式 $(x \% 2.0 == 0.5)$ 只有在 $x+0.5$ 为奇数时（也就是我们的公式会出错的情况）为真。基于这些情况，定义一个无偏取整函数就很简单了：

```
function round (x)
    local f = math.floor(x)
    if (x == f) or (x % 2.0 == 0.5) then
        return f
    else
        return math.floor(x + 0.5)
    end
end

print(round(2.5))      --> 2
print(round(3.5))      --> 4
print(round(-2.5))     --> -2
print(round(-1.5))     --> -2
```

3.5 表示范围

大多数编程语言使用某些固定长度的比特位来表达数值。因此，数值的表示在范围和精度上都是有限制的。

标准 Lua 使用 64 个比特位来存储整型值，其最大值为 $2^{63} - 1$ ，约等于 10^{19} ；精简 Lua 使用 32 个比特位存储整型值，其最大值约为 20 亿。数学库中的常量定义了整型值的最大值 (`math.maxinteger`) 和最小值 (`math.mininteger`)。

64 位整型值中的最大值是一个很大的数值：全球财富总和（按美分计算）的数千倍和全球人口总数的数十亿倍。尽管这个数值很大，但是仍然有可能发生溢出。当我们在整型操作时出现比 `mininteger` 更小或者比 `maxinteger` 更大的数值时，结果就会回环（*wrap around*）。

在数学领域，回环的意思是结果只能在 `mininteger` 和 `maxinteger` 之间，也就是对 2^{64} 取模的算术结果。在计算机领域，回环的意思是丢弃最高进位（the last carry bit）。假设最高进位存在，其将是第 65 个比特位，代表 2^{64} 。因此，忽略第 65 个比特位不会改变值对 2^{64} 取模的结果。在 Lua 语言中，这种行为对所有涉及整型值的算术运算都是一致且可预测的：

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

最大可以表示的整数是 `0x7ff...fff`，即除最高位（符号位，零为非负数值）外其余比特位均为 1。当我们对 `0x7ff...fff` 加 1 时，其结果变为 `0x800...000`，即最小可表示的整数。最小整数比最大整数的表示幅度大 1：

```
> math.maxinteger      --> 9223372036854775807
> 0xfffffffffffffff   --> 9223372036854775807
> math.mininteger      --> -9223372036854775808
> 0x8000000000000000  --> -9223372036854775808
```

对于浮点数而言，标准 Lua 使用双精度。标准 Lua 使用 64 个比特位表示所有数值，其中 11 位为指数。双精度浮点数可以表示具有大致 16 个有效十进制位的数，范围从 -10^{308} 到 10^{308} 。精简 Lua 使用 32 个比特位表示的单精度浮点数，大致具有 7 个有效十进制位，范围从 -10^{38} 到 10^{38} 。

双精度浮点数对于大多数实际应用而言是足够大的，但是我们必须了解精度的限制。如果我们使用十位表示一个数，那么 $1/7$ 会被取整到 0.142857142。如果我们使用十位计算 $1/7$

$7 * 7$, 结果会是 0.999999994 而不是 1。此外, 用十进制表示的有限小数在用二进制表示时可能是无限小数。例如, $12.7 - 20 + 7.3$ 即便是用双精度表示也不是 0, 这是由于 12.7 和 7.3 的二进制表示不是有限小数 (参见练习 3.5)。

由于整型值和浮点型值的表示范围不同, 因此当超过它们的表示范围时, 整型值和浮点型值的算术运算会产生不同的结果:

```
> math.maxinteger + 2          --> -9223372036854775807
> math.maxinteger + 2.0        --> 9.2233720368548e+18
```

在上例中, 两个结果从数学的角度看都是错误的, 而且它们错误的方式不同。第一行对最大可表示整数进行了整型求和, 结果发生了回环。第二行对最大可表示整数进行了浮点型求和, 结果被取整成了一个近似值, 这可以通过如下的比较运算证明:

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

尽管每一种表示方法都有其优势, 但是只有浮点型才能表示小数。浮点型的值可以表示很大的范围, 但是浮点型能够表示的整数范围被精确地限制在 $[-2^{53}, 2^{53}]$ 之间 (不过这个范围已经很大了)。在这个范围内, 我们基本可以忽略整型和浮点型的区别; 超出这个范围后, 我们则应该谨慎地思考所使用的表示方式。

3.6 惯例

我们可以简单地通过增加 0.0 的方法将整型值强制转换为浮点型值, 一个整型值总是可以被转换成浮点型值:

```
> -3 + 0.0                  --> -3.0
> 0xfffffffffffff + 0.0      --> 9.2233720368548e+18
```

小于 2^{53} (即 9007199254740992) 的所有整型值的表示与双精度浮点型值的表示一样, 对于绝对值超过了这个值的整型值而言, 在将其强制转换为浮点型值时可能导致精度损失:

```
> 9007199254740991 + 0.0 == 9007199254740991  --> true
> 9007199254740992 + 0.0 == 9007199254740992  --> true
> 9007199254740993 + 0.0 == 9007199254740993  --> false
```

在最后一行中, $2^{53} + 1$ 的结果被取整为 2^{53} , 打破了等式, 表达式结果为 false。

通过与零进行按位或运算，可以把浮点型值强制转换为整型值：^①

```
> 2^53           --> 9.007199254741e+15      (浮点型值)
> 2^53 | 0     --> 9007199254740992      (整型值)
```

在将浮点型值强制转换为整型值时，Lua 语言会检查数值是否与整型值表示完全一致，即没有小数部分且其值在整型值的表示范围内，如果不满足条件则会抛出异常：

```
> 3.2 | 0       -- 小数部分
stdin:1: number has no integer representation
> 2^64 | 0       -- 超出范围
stdin:1: number has no integer representation
> math.random(1, 3.5)
stdin:1: bad argument #2 to 'random' (数值没有用整型表示)
```

对小数进行取整必须显式地调用取整函数。

另一种把数值强制转换为整型值的方式是使用函数 `math.tointeger`，该函数会在输入参数无法转换为整型值时返回 `nil`：

```
> math.tointeger(-258.0)    --> -258
> math.tointeger(2^30)      --> 1073741824
> math.tointeger(5.01)      --> nil        (不是整数值)
> math.tointeger(2^64)      --> nil        (超出范围)
```

这个函数在需要检查一个数字能否被转换成整型值时尤为有用。例如，以下函数在可能时会将输入参数转换为整型值，否则保持原来的值不变：

```
function cond2int (x)
    return math.tointeger(x) or x
end
```

3.7 运算符优先级

Lua 语言中的运算符优先级如下（优先级从高到低）：

^① 位操作在 Lua 5.3 中引入，我们会在 13.1 节中对其进行讨论。

\wedge	(按位或)
-	(一元运算符 (-, #, ~, not))
*	(乘)
/	(除)
//	(整除)
%	(取余数)
+	(加)
-	(减)
..	(连接)
<<	(按位移位)
>>	(按位移位)
&	(按位与)
~	(按位异或)
	(按位或)
< > <= >= ~= ==	

and
or

在二元运算符中，除了幂运算和连接操作符是右结合的外，其他运算符都是左结合的。因此，以下各个表达式的左右两边等价：

$a+i < b/2+1$	\leftrightarrow	$(a+i) < ((b/2)+1)$
$5+x^2*8$	\leftrightarrow	$5+((x^2)*8)$
$a < y \text{ and } y \leq z$	\leftrightarrow	$(a < y) \text{ and } (y \leq z)$
$-x^2$	\leftrightarrow	$-(x^2)$
x^y^z	\leftrightarrow	$x^{(y^z)}$

当不能确定某些表达式的运算符优先级时，应该显式地用括号来指定所希望的运算次序。这比查看参考手册方便，也不至于让别人在阅读你的代码时产生同样的疑问。

3.8 兼容性

诚然，Lua 5.3 中引入的整型值导致其相对于此前的 Lua 版本出现了一定的不兼容，但如前所述，程序员基本上可以忽略整型值和浮点型值之间的不同。当忽略这些不同时，也就忽略掉了 Lua 5.3 和 Lua 5.2（该版本中所有的数值都是浮点型）之间的不同（至于数值，Lua 5.0 及 Lua 5.1 与 Lua 5.2 完全一致）。

Lua 5.3 和 Lua 5.2 之间的最大不同就是整数的表示范围。Lua 5.2 支持的最大整数为 2^{53} ，而 Lua 5.3 支持的最大整数为 2^{63} 。在当作计数值使用时，它们之间的区别通常不会导致问题；然而，当把整型值当作通用的比特位使用时（例如，把 3 个 20-bit 的整型值放在一起使用），它们之间的区别则可能很重要。

虽然 Lua 5.2 不支持整型，但是在几个场景下仍然会涉及整型的问题。例如，C 语言实现的库函数通常使用整型参数，但 Lua 5.2 却并没有约定这些情况下浮点型值和整型值之间的转换方法：官方文档里只是说“数值会以某种不确定的方式被截断”。这个问题非常现实，根据具体的不同平台，Lua 5.2 可能将 -3.2 转换成 -3，也可能转换为 -4。与 Lua 5.2 不同的是，Lua 5.3 明确了这种类型转换的规则，即只有数值恰好可以表示为整数时才可以进行转换。

由于 Lua 5.2 中的数值类型只有一种，所以没有提供函数 `math.type`。由于 Lua 5.2 中不存在整型的概念，所以也没有常量 `math.maxinteger` 及 `math.mininteger`。虽然可以实现，但 Lua 5.2 中也没有 `floor` 除法（毕竟，Lua 5.2 中的取模运算基本上和 `floor` 除法是等价的）。

可能让人感到震惊的是，与整型引入相关的问题的根源在于，Lua 语言将数值转换为字符串的方式。Lua 5.2 将所有的整数值格式化为整型（不带小数点），而 Lua 5.3 则将所有的浮点数格式化为浮点型（带有十进制小数点或指数）。因此，Lua 5.2 会将 `3.0` 格式化为 "3" 输出，而 Lua 5.3 则会将其格式化为 "3.0" 输出。虽然 Lua 语言从未说明过格式化数值的方式，但是很多程序员默认的是早期版本的格式化输出行为。在将数值转换为字符串时，我们可以通过显式地指明格式的方式来避免这种问题。然而，这个问题实际上提示我们，语言设计思想中可能存在更深层的瑕疵，即无理由地将整数转换为浮点型值来可能并非好事（实际上，这也正是 Lua 5.3 中引入新格式化规则的主要动机。将整数值使用浮点型表示通常会使得程序可读性不佳，而新的格式化规则避免了这些问题）。

3.9 练习

练习 3.1：以下哪些是有效的数值常量？它们的值分别是多少？

.0e12 .e12 0.0e 0x12 0xABFG 0xA FFFF 0xFFFFFFFF
0x 0x1P10 0.1e1 0x0.1p1

练习 3.2：解释下列表达式之所以得出相应结果的原因。（注意：整型算术运算总是会回环。）

```
> math.maxinteger * 2          --> -2
> math.mininteger * 2          --> 0
> math.maxinteger * math.maxinteger  --> 1
> math.mininteger * math.mininteger  --> 0
```

练习 3.3：下列代码的输出结果是什么？

```
for i = -10, 10 do
    print(i, i % 3)
end
```

练习 3.4: 表达式 $2^3 \cdot 4$ 的值是什么? 表达式 $2^{-3} \cdot 4$ 呢?

练习 3.5：当分母是 10 的整数次幂时，数值 12.7 与表达式 $127/10$ 相等。能否认为当分母是 2 的整数次幂时，这是一种通用规律？对于数值 5.5 情况又会怎样呢？

练习 3.6：请编写一个通过高、母线与轴线的夹角来计算正圆锥体体积的函数。

练习 3.7：利用函数 `math.random` 编写一个生成遵循正态分布（高斯分布）的伪随机数发生器。

4

字符串

字符串用于表示文本。Lua 语言中的字符串既可以表示单个字符，也可以表示一整本书籍^①。在 Lua 语言中，操作 100K 或者 1M 个字母组成的字符串的程序也很常见。

Lua 语言中的字符串是一串字节组成的序列，Lua 核心并不关心这些字节究竟以何种方式编码文本。在 Lua 语言中，字符使用 8 个比特位来存储（eight-bit clean^②）。Lua 语言中的字符串可以存储包括空字符在内的所有数值代码，这意味着我们可以在字符串中存储任意的二进制数据。同样，我们可以使用任意一种编码方法（UTF-8、UTF-16 等）来存储 Unicode 字符串；不过，像我们接下来很快要讨论的那样，最好在一切可能的情况下优先使用 UTF-8 编码。Lua 的字符串标准库默认处理 8 个比特位（1Byte）的字符，但是也同样可以非常优雅地处理 UTF-8 字符串。此外，从 Lua 5.3 开始还提供了一个帮助使用 UTF-8 编码的函数库。

Lua 语言中的字符串是不可变值（immutable value）。我们不能像在 C 语言中那样直接改变某个字符串中的某个字符，但是我们可以通过创建一个新字符串的方式来达到修改的目的，例如：

```
a = "one string"
b = string.gsub(a, "one", "another") -- 改变字符串中的某些部分
print(a)      --> one string
print(b)      --> another string
```

^①译者注：实际是指可以存储比单个字符多得多的文本内容。

^②译者注：通常与之对比的是 7-bit ASCII。

像 Lua 语言中的其他对象（表、函数等）一样，Lua 语言中的字符串也是自动内存管理的对象之一。这意味着 Lua 语言会负责字符串的分配和释放，开发人员无须关注。

可以使用长度操作符 (*length operator*) (#) 获取字符串的长度：

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

该操作符返回字符串占用的字节数，在某些编码中，这个值可能与字符串中字符的个数不同。

我们可以使用连接操作符 .. (两个点) 来进行字符串连接。如果操作数中存在数值，那么 Lua 语言会先把数值转换成字符串：

```
> "Hello" .. "World"      --> Hello World
> "result is" .. 3        --> result is 3
```

在某些语言中，字符串连接使用的是加号，但实际上 3+5 和 3..5 是不一样的。

应该注意，在 Lua 语言中，字符串是不可变量。字符串连接总是创建一个新字符串，而不会改变原来作为操作数的字符串：

```
> a = "Hello"
> a .. " World"          --> Hello World
> a                      --> Hello
```

4.1 字符串常量

我们可以使用一对双引号或单引号来声明字符串常量 (literal string)：

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

使用双引号和单引号声明字符串是等价的。它们两者唯一的区别在于，使用双引号声明的字符串中出现单引号时，单引号可以不用转义；使用单引号声明的字符串中出现双引号时，双引号可以不用转义。

从代码风格上看，大多数程序员会选择使用相同的方式来声明“同一类”字符串，至于

“同一类”究竟具体指什么则是依赖于具体实现的。^①比如，由于 XML 文本中一般都会有双引号，所以一个操作 XML 的库可能就会使用单引号来声明 XML 片段^②。

Lua 语言中的字符串支持下列 C 语言风格的转义字符：

\a	响铃 (bell)
\b	退格 (back space)
\f	换页 (form feed)
\n	换行 (newline)
\r	回车 (carriage return)
\t	水平制表符 (horizontal tab)
\v	垂直制表符 (vertical tab)
\\\	反斜杠 (backslash)
\"	双引号 (double quote)
\'	单引号 (single quote)

下述示例展示了转义字符的使用方法：

```
> 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: '\'
```

在字符串中，还可以通过转义序列 `\ddd` 和 `\xhh` 来声明字符。其中，`ddd` 是由最多 3 个十进制数字组成的序列，`hh` 是由两个且必须是两个十六进制数字组成的序列。举一个稍微有点刻意的例子，在一个使用 ASCII 编码的系统中，“AL0\n123\””和“\x41L0\10\04923”实际上是一样的^③：0x41（十进制的 65）在 ASCII 编码中对应 A，10 对应换行符，49 对应数字 1

^①译者注：即大多数情况下要么使用单引号声明字符串，要么就使用双引号来声明字符串，不会一会儿使用单引号一会儿使用双引号。

^②译者注：XML 结构中一般包括大量的双引号，如果使用双引号来声明代表 XML 文本的字符串，那么 XML 文本中原有的双引号都得进行转义。

^③译者注：字面值一样，但不一定在相同内存位置。

(在这个例子中,由于转义序列之后紧跟了其他的数字,所以49必须写成\049,即用0来补足三位数字;否则,Lua语言会将其错误地解析为\492)。我们还可以把上述字符串写成'\x41\x4c\x4f\x0a\x31\x32\x33\x22',即使用十六进制来表示字符串中的每一个字符。

从Lua 5.3开始,也可以使用转义序列\u{h...h}来声明UTF-8字符,花括号中可以支持任意有效的十六进制:

```
> "\u{3b1} \u{3b2} \u{3b3}"      --> α β γ
```

上例中假定终端使用的是UTF-8编码。

4.2 长字符串/多行字符串

像长注释/多行注释一样,可以使用一对双方括号来声明长字符串/多行字符串常量。被方括号括起来的内容可以包括很多行,并且内容中的转义序列不会被转义。此外,如果多行字符串中的第一个字符是换行符,那么这个换行符会被忽略。多行字符串在声明包含大段代码的字符串时非常方便,例如:

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

有时字符串中可能有类似a=b[c[i]]这样的内容(注意其中的]],或者,字符串中可能有被注释掉的代码。为了应对这些情况,可以在两个左方括号之间加上任意数量的等号,如[===[。这样,字符串常量只有在遇到了包含相同数量等号的两个右方括号时才会结束(就前例而言,即]===])。Lua语言的语法扫描器会忽略所含等号数量不相同的方括号。通过选择恰当数量的等号,就可以在无须修改原字符串的情况下声明任意的字符串常量了。

对注释而言，这种机制也同样有效。例如，我们可以使用`--[=]`来进行长注释，从而降低了对内部已经包含注释的代码进行注释的难度。

当代码中需要使用常量文本时，使用长字符串是一种理想的选择。但是，对于非文本的常量我们不应该滥用长字符串。虽然 Lua 语言中的字符串常量可以包含任意字节，但是滥用这个特性并不明智（例如，可能导致某些文本编辑器出现异常）。同时，像`"\r\n"`一样的 EOF 序列在被读取的时候可能会被归一化成`"\n"`。作为替代方案，最好就是把这些可能引起歧义的二进制数据用十进制数值或十六进制的数值转义序列进行表示，例如`"\x13\x01\xA1\xBB"`。不过，由于这种转义表示形成的字符串往往很长，所以对于长字符串来说仍可能是个问题。针对这种情况，从 Lua 5.2 开始引入了转义序列`\z`，该转义符会跳过其后的所有空白字符，直到遇到第一个非空白字符。下例中演示了该转义符的使用方法：

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

第一行最后的`\z`会跳过其后的 EOF 和第二行的制表符，因此在最终得到的字符串中，`\x08`实际上是紧跟着`\x07`的。

4.3 强制类型转换

Lua 语言在运行时提供了数值与字符串之间的自动转换（conversion）。针对字符串的所有算术操作会尝试将字符串转换为数值。Lua 语言不仅仅在算术操作时进行这种强制类型转换（coercion），还会在任何需要数值的情况下进行，例如函数 `math.sin` 的参数。

相反，当 Lua 语言发现在需要字符串的地方出现了数值时，它就会把数值转换为字符串：

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

当在数值后紧接着使用字符串连接时，必须使用空格将它们分开，否则 Lua 语言会把第一个点当成小数点。

很多人认为自动强制类型转换算不上是 Lua 语言中的一项好设计。作为原则之一，建议最好不要完全寄希望于自动强制类型转换。虽然在某些场景下这种机制很便利，但同时也给语言和使用这种机制的程序带来了复杂性。

作为这种“二类状态（second-class status）”的表现之一，Lua 5.3 没有实现强制类型转换与整型的集成，而是采用了另一种更简单和快速的实现方式：算术运算的规则就是只有在

两个操作数都是整型值时结果才是整型。因此，由于字符串不是整型值，所以任何有字符串参与的算术运算都会被当作浮点运算处理：

```
> "10" + 1      --> 11.0
```

如果需要显式地将一个字符串转换成数值，那么可以使用函数 `tonumber`。当这个字符串的内容不能表示为有效数字时该函数返回 `nil`；否则，该函数就按照 Lua 语法规则返回对应的整型值或浮点类型值：

```
> tonumber("-3")      --> -3
> tonumber("10e4")    --> 100000.0
> tonumber("10e")     --> nil (not a valid number)
> tonumber("0x1.3p-4") --> 0.07421875
```

默认情况下，函数 `tonumber` 使用的是十进制，但是也可以指明使用二进制到三十六进制之间的任意进制：

```
> tonumber("100101", 2)      --> 37
> tonumber("fff", 16)        --> 4095
> tonumber("-ZZ", 36)        --> -1295
> tonumber("987", 8)         --> nil
```

在最后一行中，对于指定的进制而言，传入的字符串是一个无效值，因此函数 `tonumber` 返回 `nil`。

调用函数 `tostring` 可以将数值转换成字符串：

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

上述的这种转换总是有效，但我们需要记住，使用这种转换时并不能控制输出字符串的格式（例如，结果中十进制数字的个数）。我们会在下一节中看到，可以通过函数 `string.format` 来全面地控制输出字符串的格式。

与算术操作不同，比较操作符不会对操作数进行强制类型转换。请注意，“`0`”和`0`是不同的。此外，`2 < 15` 明显为真，但“`2 < "15"` 却为假（字母顺序）。为了避免出现不一致的结果，当比较操作符中混用了字符串和数值（比如 `2 < "15"`）时，Lua 语言会抛出异常。

4.4 字符串标准库

Lua 语言解释器本身处理字符串的能力是十分有限的。一个程序能够创建字符串、连接字符串、比较字符串和获取字符串的长度，但是，它并不能提取字符串的子串或检视字符串的内容。Lua 语言处理字符串的完整能力来自其字符串标准库。

正如此前提到的，字符串标准库默认处理的是 8 bit (1 byte) 字符。这对于某些编码方式（例如 ASCII 或 ISO-8859-1）适用，但对所有的 Unicode 编码来说都不适用。不过尽管如此，我们接下来会看到，字符串标准库中的某些功能对 UTF-8 编码来说还是非常有用的。

字符串标准库中的一些函数非常简单：函数 `string.len(s)` 返回字符串 `s` 的长度，等价于 `#s`。函数 `string.rep(s, n)` 返回将字符串 `s` 重复 `n` 次的结果。可以通过调用 `string.rep("a", 2^20)` 创建一个 1MB 大小的字符串（例如用于测试）。函数 `string.reverse` 用于字符串翻转。函数 `string.lower(s)` 返回一份 `s` 的副本，其中所有的大写字母都被转换成小写字母，而其他字符则保持不变。函数 `string.upper` 与之相反，该函数会将小写字母转换成大写字母。

```
> string.rep("abc", 3)           --> abcabcabc
> string.reverse("A Long Line!") --> !enil gnol A
> string.lower("A Long Line!")  --> a long line!
> string.upper("A Long Line!")  --> A LONG LINE!
```

作为一种典型的应用，我们可以使用如下代码在忽略大小写差异的原则下比较两个字符串：

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

函数 `string.sub(s, i, j)` 从字符串 `s` 中提取第 `i` 个到第 `j` 个字符（包括第 `i` 个和第 `j` 个字符，字符串的第一个字符索引为 1）。该函数也支持负数索引，负数索引从字符串的结尾开始计数：索引 `-1` 代表字符串的最后一个字符，索引 `-2` 代表倒数第二个字符，依此类推。这样，对字符串 `s` 调用函数 `string.sub(s, 1, j)` 得到的是字符串 `s` 中长度为 `j` 的前缀，调用 `string.sub(s, j, -1)` 得到的是字符串 `s` 中从第 `j` 个字符开始的后缀，调用 `string.sub(s, 2, -2)` 返回的是去掉字符串 `s` 中第一个和最后一个字符后的结果：

```
> s = "[in brackets]"
> string.sub(s, 2, -2)      --> in brackets
> string.sub(s, 1, 1)        --> [
> string.sub(s, -1, -1)     --> ]
```

请注意，Lua 语言中的字符串是不可变的。和 Lua 语言中的所有其他函数一样，函数 `string.sub` 不会改变原有字符串的值，它只会返回一个新字符串。一种常见的误解是以为 `string.sub(s, 2, -2)` 返回的是修改后的 `s`^①。如果需要修改原字符串，那么必须把新的值赋值给它：

```
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。函数 `string.char` 接收零个或多个整数作为参数，然后将每个整数转换成对应的字符，最后返回由这些字符连接而成的字符串。函数 `string.byte(s, i)` 返回字符串 `s` 中第 `i` 个字符的内部数值表示，该函数的第二个参数是可选的。调用 `string.byte(s)` 返回字符串 `s` 中第一个字符（如果字符串只由一个字符组成，那么就返回这个字符）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
```

在最后一行中，使用了负数索引来访问字符串的最后一个字符。

调用 `string.byte(s, i, j)` 返回索引 `i` 到 `j` 之间（包括 `i` 和 `j`）的所有字符的数值表示：

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

一种常见的写法是 `{string.byte(s, 1, -1)}`，该表达式会创建一个由字符串 `s` 中的所有字符代码组成的表（由于 Lua 语言限制了栈大小，所以也限制了一个函数的返回值的最大个数，默认最大为一百万个。因此，这个技巧不能用于大小超过 1MB 的字符串）。

函数 `string.format` 是用于进行字符串格式化和将数值输出为字符串的强大工具，该函数会返回第一个参数（也就是所谓的格式化字符串 (*format string*)）的副本，其中的每一个指示符 (*directive*) 都会被替换为使用对应格式进行格式化后的对应参数。格式化字符串中的指示符与 C 语言中函数 `printf` 的规则类似，一个指示符由一个百分号和一个代表格式化方式的字母组成：`d` 代表一个十进制整数、`x` 代表一个十六进制整数、`f` 代表一个浮点数、`s` 代表字符串，等等。

^①译者注：实际上字符串 `s` 不会被修改。

Lua 程序设计（第 4 版）

```
> string.format("x = %d y = %d", 10, 20) --> x = 10 y = 20
> string.format("x = %x", 200) --> x = c8
> string.format("x = 0x%X", 200) --> x = 0xC8
> string.format("x = %f", 200) --> x = 200.000000
> tag, title = "h1", "a title"
> string.format("<%s>%s</%s>", tag, title, tag)
--> <h1>a title</h1>
```

在百分号和字母之间可以包含用于控制格式细节的其他选项。例如，可以指定一个浮点数中小数点的位数：

```
print(string.format("pi = %.4f", math.pi)) --> pi = 3.1416
d = 5; m = 11; y = 1990
print(string.format("%02d/%02d/%04d", d, m, y)) --> 05/11/1990
```

在上例中，`%.4f` 表示小数点后保留 4 位小数；`%02d` 表示一个十进制数至少由两个数字组成，不足两个数字的用 0 补齐，而`%2d` 则表示用空格来补齐。关于这些指示符的完整描述可以参阅 C 语言 `printf` 函数的相关文档，因为 Lua 语言是通过调用 C 语言标准库来完成实际工作的。

可以使用冒号操作符像调用字符串的一个方法那样调用字符串标准库中的所有函数。例如，`string.sub(s, i, j)` 可以重写为 `s:sub(i, j)`，`string.upper(s)` 可以重写为 `s:upper()`（我们会在第 21 章中学习冒号操作符的细节）。

字符串标准库还包括了几个基于模式匹配的函数。函数 `string.find` 用于在指定的字符串中进行模式搜索：

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

如果该函数在指定的字符串中找到了匹配的模式，则返回模式的开始和结束位置，否则返回 `nil`。函数 `string.gsub` (Global SUBstitution) 则把所有匹配的模式用另一个字符串替换：

```
> string.gsub("hello world", "l", ".") --> he..o wor.d 3
> string.gsub("hello world", "ll", "..") --> he..o world 1
> string.gsub("hello world", "a", ".") --> hello world 0
```

该函数还会在第二个返回值中返回发生替换的次数。

我们会在第 10 章中继续学习上面提到的所有函数和关于模式匹配的所有知识。

4.5 Unicode 编码

从 Lua 5.3 开始，Lua 语言引入了一个用于操作 UTF-8 编码的 Unicode 字符串的标准库。当然，在引入这个标准库之前，Lua 语言也提供了对 UTF-8 字符串的合理支持。

UTF-8 是 Web 环境中用于 Unicode 的主要编码之一。由于 UTF-8 编码与 ASCII 编码部分兼容，所以 UTF-8 对于 Lua 语言来说也是一种理想的编码方式。这种兼容性保证了用于 ASCII 字符串的一些字符串操作技巧无须修改就可以用于 UTF-8 字符串。

UTF-8 使用变长的多个字节来编码一个 Unicode 字符。例如，UTF-8 编码使用一个字节的 65 来代表 A，使用两个字节的 215-144 代表希伯来语（Hebrew）字符 Aleph（其在 Unicode 中的编码是 1488）。UTF-8 使用一个字节表示所有 ASCII 范围内的字符（小于 128）。对于其他字符，则使用字节序列表示，其中第一个字节的范围是 [194, 244]，而后续的字节范围是 [128, 191]。更准确地说，对于两个字节组成的序列来说，第一个字节的范围是 [194, 223]；对于三个字节组成的序列来说，第一个字节的范围是 [224, 239]；对于四个字节组成的序列来说，第一个字节的范围是 [240, 244]，这些范围相互之间均不重叠。这种特点保证了任意字符对应的字节序列不会在其他字符对应的字节序列中出现。特别地，一个小于 128 的字节永远不会出现在多字节序列中，它只会代表与之对应的 ASCII 字符。

Lua 语言中的一些机制对 UTF-8 字符串来说同样“有效”。由于 Lua 语言使用 8 个字节来编码字符，所以可以像操作其他字符串一样读写和存储 UTF-8 字符串。字符串常量也可以包含 UTF-8 数据（当然，读者可能需要使用支持 UTF-8 编码的编辑器来处理使用 UTF-8 编码的源文件）。字符串连接对 UTF-8 字符串同样适用。对字符串的比较（小于、小于等于，等等）会按照 Unicode 编码中的字符代码顺序进行^①。

Lua 语言的操作系统库和输入输出库是与对应系统之间的主要接口，所以它们是否支持 UTF-8 取决于对应的操作系统。例如，在 Linux 操作系统下文件名使用 UTF-8 编码，而在 Windows 操作系统下文件名使用 UTF-16 编码。因此，如果要在 Windows 操作系统中处理 Unicode 文件名，那么要么使用额外的库，要么就要修改 Lua 语言的标准库。

让我们看一下字符串标准库中的函数是如何处理 UTF-8 字符串的。函数 `reverse`、`upper`、`lower`、`byte` 和 `char` 不适用于 UTF-8 字符串，这是因为它们针对的都是一字节字符。函数 `string.format` 和 `string.rep` 适用于 UTF-8 字符串（格式选项 '`%c`' 除外，该格式选项针对一字节字符）。函数 `string.len` 和 `string.sub` 可以用于 UTF-8 字符串，其中的索引以字节

^①译者注：即代码点，后面会详细介绍。

为单位而不是以字符为单位。通常，这些函数就够用了。

现在让我们学习一下新的 utf8 标准库。函数 `utf8.len` 返回指定字符串中 UTF-8 字符（代码点）的个数^①。此外，该函数还会验证字符串，如果该函数发现字符串中包含无效的字节序列，则返回 `false` 外加第一个无效字节的位置：

```
> utf8.len("résumé")           --> 6
> utf8.len("ação")            --> 4
> utf8.len("Månen")           --> 5
> utf8.len("ab\x93")          --> nil    3
```

当然，需要使用支持 UTF-8 的终端来运行上述示例。

函数 `utf8.char` 和 `utf8.codepoint` 在 UTF-8 环境下等价于 `string.char` 和 `string.byte`：

```
> utf8.char(114, 233, 115, 117, 109, 233)   --> résumé
> utf8.codepoint("résumé", 6, 7)                --> 109    233
```

请注意最后一行的索引。`utf8` 库中大多数函数使用字节为索引。例如，调用 `string.codepoint(s, i, j)` 时 `i` 和 `j` 都会被当作字符串 `s` 中的字节位置。如果想使用字符位置作为索引，那么可以通过函数 `utf8.offset` 把字符位置转换为字节位置：

```
> s = "Nähdäään"
> utf8.codepoint(s, utf8.offset(s, 5))      --> 228
> utf8.char(228)                           --> ä
```

在这个示例中，我们使用函数 `utf8.offset` 来获取字符串中第 5 个字符的字节索引，然后将这个值作为参数调用函数 `codepoint`。

像在字符串标准库中一样，函数 `utf8.offset` 使用的索引可以是负值，代表从字符串末尾开始计数：

```
> s = "ÃøÆÊÐ"
> string.sub(s, utf8.offset(s, -2))     --> ÆÐ
```

`utf8` 标准库中的最后一个函数是 `utf8.codes`，该函数用于遍历 UTF-8 字符串中的每一个字符：

^①译者注：正如前文所述，一个诸如 Unicode 等的超大字符集中的字符可能需要用两个或两个以上的字节表示，一个完整的 Unicode 字符就叫做代码点，不能直接使用字节位置或字节长度来对 Unicode 字符进行操作。

```

for i, c in utf8.codes("Ação") do
    print(i, c)
end
--> 1    65
--> 2    231
--> 4    227
--> 6    111

```

上述的代码结构会遍历指定字符串中的所有字符，将每个字符对应的字节索引和编码赋给两个局部变量。在上例中，循环体会打印出这两个变量的值（我们会在第18章中进一步学习迭代器）。

不幸的是，除了上述的内容外，Lua 语言没有再提供其他机制。Unicode 具有如此多稀奇古怪的特性，以至于想从特定的语言中抽象出其中的任意一个概念基本上都是不太可能的。由于 Unicode 编码的字符和字素（grapheme）之间没有一对一的关系，所以甚至连字符的概念都是模糊的。例如，常见的字素 é 既可以使用单个代码点 "\u{E9}" 表示，也可以使用两个代码点表示 ("e\u{301}"，即 e 后面跟一个区分标记)。其他诸如字母之类的基本概念在不同的语系中也有差异。由于这些复杂性的存在，如果想支持完整的 Unicode 就需要巨大的表，而这又与 Lua 语言精简的大小相矛盾。因此，对于这些特殊需求来说，最好的选择就是使用外部库。

4.6 练习

练习 4.1：请问如何在 Lua 程序中以字符串的方式使用如下的 XML 片段：

```

<![CDATA[
Hello world
]]>

```

请给出至少两种实现方式。

练习 4.2：假设你需要以字符串常量的形式定义一组包含歧义的转义字符序列，你会使用哪种方式？请注意考虑诸如可读性、每行最大长度及字符串最大长度等问题。

练习 4.3：请编写一个函数，使之实现在某个字符串的指定位置插入另一个字符串：

```

> insert("hello world", 1, "start: ")      --> start: hello world
> insert("hello world", 7, "small ")       --> hello small world

```

Lua 程序设计（第 4 版）

练习 4.4：使用 UTF-8 字符串重写下例：

```
> insert("ação", 5, "!")
--> ação!
```

注意，这里的起始位置和长度都是针对代码点（CodePoint）而言的。

练习 4.5：请编写一个函数，该函数用于移除指定字符串中的一部分，移除的部分使用起始位置和长度指定：

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

练习 4.6：使用 UTF-8 字符串重写下例：

```
> remove("ação", 2, 2) --> ao
```

注意，起始位置和长度都是以代码点来表示的。

练习 4.7：请编写一个函数判断指定的字符串是否为回文字符串（palindrome）：

```
> ispali("step on no pets") --> true
> ispali("banana") --> false
```

练习 4.8：重写之前的练习，使得它们忽略空格和标点符号。

练习 4.9：使用 UTF-8 字符串重写之前的练习。

5

表

表 (Table) 是 Lua 语言中最主要 (事实上也是唯一的) 和强大的数据结构。使用表, Lua 语言可以以一种简单、统一且高效的方式表示数组、集合、记录和其他很多数据结构。Lua 语言也使用表来表示包 (package) 和其他对象。当调用函数 `math.sin` 时, 我们可能认为是“调用了 `math` 库中函数 `sin`”; 而对于 Lua 语言来说, 其实际含义是“以字符串“`sin`”为键检索表 `math`”。

Lua 语言中的表本质上是一种辅助数组 (associative array), 这种数组不仅可以使用数值作为索引, 也可以使用字符串或其他任意类型的值作为索引 (nil 除外)。

Lua 语言中的表要么是值要么是变量, 它们都是对象 (*object*)。如果读者对 Java 或 Scheme 中的数组比较熟悉, 那么应该很容易理解上述概念。可以认为, 表是一种动态分配的对象, 程序只能操作指向表的引用 (或指针)。除此以外, Lua 语言不会进行隐藏的拷贝 (hidden copies) 或创建新的表^①。

我们使用构造器表达式 (*constructor expression*) 创建表, 其最简单的形式是 {}:

```
> a = {}           -- 创建一个表然后用表的引用赋值
> k = "x"
> a[k] = 10      -- 新元素, 键是"x", 值是10
> a[20] = "great" -- 新元素, 键是20, 值是"great"
> a["x"]          --> 10
```

^①译者注: 此处所谓的隐藏的拷贝是指深拷贝, 即拷贝的是对象的引用而非整个对象本身。



Lua 程序设计（第 4 版）

```
> k = 20
> a[k]                      --> "great"
> a["x"] = a["x"] + 1        -- 增加元素"x"的值
> a["x"]                      --> 11
```

表永远是匿名的，表本身和保存表的变量之间没有固定的关系：

```
> a = {}
> a["x"] = 10
> b = a                      -- 'b'和'a'引用同一张表
> b["x"]                      --> 10
> b["x"] = 20
> a["x"]                      --> 20
> a = nil                     -- 只有'b'仍然指向表
> b = nil                     -- 没有指向表的引用了
```

对于一个表而言，当程序中不再有指向它的引用时，垃圾收集器会最终删除这个表并重用其占用的内存。

5.1 表索引

同一个表中存储的值可以具有不同的类型索引^①，并可以按需增长以容纳新的元素：

```
> a = {}          -- 空的表
> -- 创建1000个新元素
> for i = 1, 1000 do a[i] = i*2 end
> a[9]             --> 18
> a["x"] = 10
> a["x"]             --> 10
> a["y"]             --> nil
```

请注意上述代码的最后一行：如同全局变量一样，未经初始化的表元素为 nil，将 nil 赋值给表元素可以将其删除。这并非巧合，因为 Lua 语言实际上就是使用表来存储全局变量的（详见第22章）。

^①译者注：即不同数据类型的键。



当把表当作结构体使用时，可以把索引当作成员名称使用（`a.name` 等价于 `a["name"]`）。因此，可以使用这种更加易读的方式改写前述示例的最后几行：

```
> a = {}          -- 空白表
> a.x = 10       -- 等价于 a["x"] = 10
> a.x           --> 10      -- 等价于 a["x"]
> a.y           --> nil     -- 等价于 a["y"]
```

对 Lua 语言而言，这两种形式是等价且可以自由混用的；不过，对于阅读程序的人而言，这两种形式可能代表了不同的意图。形如 `a.name` 的点分形式清晰地说明了表是被当作结构体使用的，此时表实际上是由固定的、预先定义的键组成的集合；而形如 `a["name"]` 的字符串索引形式则说明了表可以使用任意字符串作为键，并且出于某种原因我们操作的是指定的键。

初学者常常会混淆 `a.x` 和 `a[x]`。实际上，`a.x` 代表的是 `a["x"]`，即由字符串 "x" 索引的表；而 `a[x]` 则是指由变量 `x` 对应的值索引的表，例如：

```
> a = {}
> x = "y"
> a[x] = 10      -- 把 10 放在字段 "y" 中
> a[x]           --> 10      -- 字段 "y" 的值
> a.x           --> nil     -- 字段 "x" 的值 (未定义)
> a.y           --> 10      -- 字段 "y" 的值
```

由于可以使用任意类型索引表，所以在索引表时会遇到相等性比较方面的微妙问题。虽然确实都能用数字 0 和字符串 "0" 对同一个表进行索引，但这两个索引的值及其所对应的元素是不同的。同样，字符串 "+1"、"01" 和 "1" 指向的也是不同的元素。当不能确定表索引的真实数据类型时，可以使用显式的类型转换：

```
> i = 10; j = "10"; k = "+10"
> a = {}
> a[i] = "number key"
> a[j] = "string key"
> a[k] = "another string key"
> a[i]           --> 数值类型的键
> a[j]           --> 字符串类型的键
> a[k]           --> 另一个字符串类型的键
> a[tonumber(j)] --> 数值类型的键
> a[tonumber(k)] --> 数值类型的键
```



如果不注意这一点，就会很容易在程序中引入诡异的 Bug。

整型和浮点型类型的表索引则不存在上述问题。由于 2 和 2.0 的值相等，所以当它们被当作表索引使用时指向的是同一个表元素：

```
> a = {}
> a[2.0] = 10
> a[2.1] = 20
> a[2]           --> 10
> a[2.1]         --> 20
```

更准确地说，当被用作表索引时，任何能够被转换为整型的浮点数都会被转换成整型数。例如，当执行表达式 `a[2.0]=10` 时，键 2.0 会被转换为 2。相反，不能被转换为整型数的浮点数则不会发生上述的类型转换。

5.2 表构造器

表构造器（Table Constructor）是用来创建和初始化表的表达式，也是 Lua 语言中独有的也是最有用、最灵活的机制之一。

正如我们此前已经提到的，最简单的构造器是空构造器 {}。表构造器也可以被用来初始化列表，例如，下例中使用字符串 "Sunday" 初始化了 `days[1]`（构造器第一个元素的索引是 1 而不是 0）、使用字符串 "Monday" 初始化了 `days[2]`，依此类推：

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

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

Lua 语言还提供了一种初始化记录式（record-like）表的特殊语法：

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

上述代码等价于：

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

不过，在第一种写法中，由于能够提前判断表的大小，所以运行速度更快。

无论使用哪种方式创建表，都可以随时增加或删除表元素：



```
w = {x = 0, y = 0, label = "console"}
x = {math.sin(0), math.sin(1), math.sin(2)}
w[1] = "another field"    -- 把键1增加到表'w'中
x.f = w                  -- 把键"f"增加到表'x'中
print(w["x"])            --> 0
print(w[1])               --> another field
print(x.f[1])             --> another field
w.x = nil                -- 删除字段"x"
```

不过，正如此前所提到的，使用合适的构造器来创建表会更加高效和易读。

在同一个构造器中，可以混用记录式（record-style）和列表式（list-style）写法：

```
polyline = {color="blue",
            thickness=2,
            npoints=4,
            {x=0, y=0},   -- polyline[1]
            {x=-10, y=0}, -- polyline[2]
            {x=-10, y=1}, -- polyline[3]
            {x=0, y=1}   -- polyline[4]
        }
```

上述的示例也同时展示了如何创建嵌套表（和构造器）以表达更加复杂的数据结构。每一个元素 `polyline[i]` 都是代表一条记录的表：

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

不过，这两种构造器都有各自的局限。例如，使用这两种构造器时，不能使用负数索引初始化表元素^①，也不能使用不符合规范的标识符作为索引。对于这类需求，可以使用另一种更加通用的构造器，即通过方括号括起来的表达式显式地指定每一个索引：

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

i = 20; s = "-"

a = {[i+0] = s, [i+1] = s..s, [i+2] = s..s..s}
```

^①译者注：意思是索引必须以 1 作为开始，不能是负数或其他值。



```

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

```

这种构造器虽然冗长，但却非常灵活，不管是记录式构造器还是列表式构造器均是其特殊形式。例如，下面的几种表达式就相互等价：

```

{x = 0, y = 0}      <->  {[ "x" ] = 0, [ "y" ] = 0}
{"r", "g", "b"}     <->  {[1] = "r", [2] = "g", [3] = "b"}

```

在最后一个元素后总是可以紧跟一个逗号。虽然总是有效，但是否加最后一个逗号是可选的：

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

这种灵活性使得开发人员在编写表构造器时不需要对最后一个元素进行特殊处理。

最后，表构造器中的逗号也可以使用分号代替，这主要是为了兼容 Lua 语言的旧版本，目前基本不会被用到。

5.3 数组、列表和序列

如果想表示常见的数组（array）或列表（list），那么只需要使用整型作为索引的表即可。同时，也不需要预先声明表的大小，只需要直接初始化我们需要的元素即可：

```
-- 读取10行，然后保存在一个表中
a = {}
for i = 1, 10 do
    a[i] = io.read()
end
```

鉴于能够使用任意值对表进行索引，我们也可以使用任意数字作为第一个元素的索引。不过，在 Lua 语言中，数组索引按照惯例是从 1 开始的（不像 C 语言从 0 开始），Lua 语言中的其他很多机制也遵循这个惯例。

当操作列表时，往往必须事先获取列表的长度。列表的长度可以存放在常量中，也可以存放在其他变量或数据结构中。通常，我们把列表的长度保存在表中某个非数值类型的字段中（由于历史原因，这个键通常是“n”）。当然，列表的长度经常也是隐式的。请注意，由于



未初始化的元素均为 nil，所以可以利用 nil 值来标记列表的结束。例如，当向一个列表中写入了 10 行数据后，由于该列表的数值类型的索引为 1, 2, ..., 10，所以可以很容易地知道列表的长度就是 10。这种技巧只有在列表中不存在空洞（*hole*）时（即所有元素均不为 nil）才有效，此时我们把这种所有元素都不为 nil 的数组称为序列（*sequence*）。^①

Lua 语言提供了获取序列长度的操作符 #。正如我们之前所看到的，对于字符串而言，该操作符返回字符串的字节数；对于表而言，该操作符返回表对应序列的长度。例如，可以使用如下的代码输出上例中读入的内容：

```
-- 输出行，从1到#a
for i = 1, #a do
    print(a[i])
end
```

长度操作符也为操作序列提供了几种有用的写法：

```
print(a[#a])          -- 输出序列'a'的最后一个值
a[#a] = nil           -- 移除最后一个值
a[#a + 1] = v         -- 把'v'加到序列的最后
```

对于中间存在空洞（nil 值）的列表而言，序列长度操作符是不可靠的，它只能用于序列（所有元素均不为 nil 的列表）。更准确地说，序列（*sequence*）是由指定的 n 个正数数值类型的键所组成集合 {1, ..., n} 形成的表（请注意值为 nil 的键实际不在表中）。特别地，不包含数值类型键的表就是长度为零的序列。

将长度操作符用于存在空洞的列表的行为是 Lua 语言中最具争议的内容之一。在过去几年中，很多人建议在操作存在空洞的列表时直接抛出异常，也有人建议扩展长度操作符的语言。然而，这些建议都是说起来容易做起来难。其根源在于列表实际上是一个表，而对于表来说，“长度”的概念在一定程度上是不容易理解的。例如，考虑如下的代码：

```
a = {}
a[1] = 1
a[2] = nil      -- 什么也没做，因为a[2]已经是nil了
a[3] = 1
a[4] = 1
```

^①译者注：此处原文的逻辑有问题，作者实际想表达的意思是，像 C 语言使用空字符\0 作为字符串结束一样，Lua 语言中可以使用 nil 来隐式地代表列表的结束，而非直接使用 1, 2, ..., 10 的索引值来判断列表的长度。



我们可以很容易确定这是一个长度为 4、在索引 2 的位置上存在空洞的列表。不过，对于下面这个类似的示例是否也如此呢？

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

是否应该认为 a 是一个具有 10000 个元素、9998 个空洞的列表？如果代码进行了如下的操作：

```
a[10000] = nil
```

那么该列表的长度会变成多少？由于代码删除了最后一个元素，该列表的长度是不是变成了 9999？或者由于代码只是将最后一个元素变成了 nil，该列表的长度仍然是 10000？又或者该列表的长度缩成了 1？^①

另一种常见的建议是让 # 操作符返回表中全部元素的数量。虽然这种语义听起来清晰且定义明确，但并非特别有用和符合直觉。请考虑一下我们在此讨论过的所有例子，然后思考一下对这些例子而言，为什么让 # 操作符返回表中全部元素的数量并非特别有用。

更复杂的是列表以 nil 结尾的情况。请问如下的列表的长度是多少：

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

请注意，对于 Lua 语言而言，一个为 nil 的字段和一个不存在的元素没有区别。因此，上述列表与 {10, 20, 30} 是等价的——其长度是 3，而不是 5。

可以将以 nil 结尾的列表当作一种非常特殊的情况。不过，很多列表是通过逐个添加各个元素创建出来的。任何按照这种方式构造出来的带有空洞的列表，其最后一定存在为 nil 的值。

尽管讨论了这么多，程序中的大多数列表其实都是序列（例如不能为 nil 的文件行）。正因如此，在多数情况下使用长度操作符是安全的。在确实需要处理存在空洞的列表时，应该将列表的长度显式地保存起来。

5.4 遍历表

我们可以使用 pairs 迭代器遍历表中的键值对：

^①译者注：在 Lua5.3 中此时表达式 #a 的结果是 1。



```
t = {10, print, x = 12, k = "hi"}
for k, v in pairs(t) do
    print(k, v)
end
--> 1    10
--> k    hi
--> 2    function: 0x420610
--> x    12
```

受限于表在 Lua 语言中的底层实现机制，遍历过程中元素的出现顺序可能是随机的，相同的程序在每次运行时也可能产生不同的顺序。唯一可以确定的是，在遍历的过程中每个元素会且只会出现一次。

对于列表而言，可以使用 `ipairs` 迭代器：

```
t = {10, print, 12, "hi"}
for k, v in ipairs(t) do
    print(k, v)
end
--> 1    10
--> 2    function: 0x420610
--> 3    12
--> 4    hi
```

此时，Lua 会确保遍历是按照顺序进行的。

另一种遍历序列的方法是使用数值型 for 循环：

```
t = {10, print, 12, "hi"}
for k = 1, #t do
    print(k, t[k])
end
--> 1    10
--> 2    function: 0x420610
--> 3    12
--> 4    hi
```



5.5 安全访问

考虑如下的情景：我们想确认在指定的库中是否存在某个函数。如果我们确定这个库确实存在，那么可以直接使用 `if lib.foo then ...`；否则，就得使用形如 `if lib and lib.foo then ...` 的表达式。

当表的嵌套深度变得比较深时，这种写法就会很容易出错，例如：

```
zip = company and company.director and
      company.director.address and
      company.director.address.zipcode
```

这种写法不仅冗长而且低效，该写法在一次成功的访问中对表进行了 6 次访问而非 3 次访问。

对于这种情景，诸如 C# 的一些编程语言提供了一种安全访问操作符（*safe navigation operator*）。在 C# 中，这种安全访问操作符被记为“`?.`”。例如，对于表达式 `a ?.b`，当 `a` 为 `nil` 时，其结果是 `nil` 而不会产生异常。使用这种操作符，可以将上例改写为：

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

如果上述的成员访问过程中出现 `nil`，安全访问操作符会正确地处理 `nil`^① 并最终返回 `nil`。

Lua 语言并没有提供安全访问操作符，并且认为也不应该提供这种操作符。一方面，Lua 语言在设计上力求简单；另一方面，这种操作符也是非常有争议的，很多人就无理由地认为该操作符容易导致无意的编程错误。不过，我们可以使用其他语句在 Lua 语言中模拟安全访问操作符。

对于表达式 `a or {}`，当 `a` 为 `nil` 时其结果是一个空表。因此，对于表达式 `(a or {}).b`，当 `a` 为 `nil` 时其结果同样是 `nil`。这样，我们就可以将之前的例子重写为：

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

再进一步，我们还可以写得更短和更高效：

```
E = {}      -- 可以在其他类似表达式中复用
...
zip = (((company or E).director or E).address or E).zipcode
```

^①译者注：原文中的用词为 `propagate nil`（传播 `nil`）。



确实，上述的语法比安全访问操作符更加复杂。不过尽管如此，表中的每一个字段名都只被使用了一次，从而保证了尽可能少地对表进行访问（本例中对表仅有 3 次访问）；同时，还避免了向语言中引入新的操作符。就我个人看来，这已经是一种足够好的替代方案了。

5.6 表标准库

表标准库提供了操作列表和序列的一些常用函数。^①

函数 `table.insert` 向序列的指定位置插入一个元素，其他元素依次后移。例如，对于列表 `t={10, 20, 30}`，在调用 `table.insert(t, 1, 15)` 后它会变成 `{15, 10, 20, 30}`，另一种特殊但常见的情况是调用 `insert` 时不指定位置，此时该函数会在序列的最后插入指定的元素，而不会移动任何元素。例如，下述代码从标准输入中按行读入内容并将其保存到一个序列中：

```
t = {}
for line in io.lines() do
    table.insert(t, line)
end
print(#t)          --> (读取的行数)
```

函数 `table.remove` 删除并返回序列指定位置的元素，然后将其后的元素向前移动填充删除元素后造成的空洞。如果在调用该函数时不指定位置，该函数会删除序列的最后一个元素。

借助这两个函数，可以很容易地实现栈（Stack）、队列（Queue）和双端队列（Double queue）。以栈的实现为例，我们可以使用 `t={}` 来表示栈，Push 操作可以使用 `table.insert(t, x)` 实现，Pop 操作可以使用 `table.remove(t)` 实现，调用 `table.insert(t, 1, x)` 可以实现向栈的顶部进行插入，调用 `table.remove(t, 1)` 可以从栈的顶部移除^②。由于后两个函数涉及表中其他元素的移动，所以其运行效率并不是特别高。当然，由于 `table` 标准库中的这些函数是使用 C 语言实现的，所以移动元素所涉及循环的性能开销也并不是很昂贵。因而，对于几百个元素组成的小数组来说这种实现已经足矣。

^①可以认为表标准库是“列表库（The List Library）”或“序列库（The Sequence Library）”。之所以保留这两个概念，是为了兼容老版本。

^②译者注：原文中的表达不准确，上述 4 个函数实际是针对栈的实现来说的，对队列和双端队列来说还需稍做调整。



Lua 5.3 对于移动表中的元素引入了一个更通用的函数 `table.move(a, f, e, t)`，调用该函数可以将表 `a` 中从索引 `f` 到 `e` 的元素（包含索引 `f` 和索引 `e` 对应的元素本身）移动到位置 `t` 上。例如，如下代码可以在列表 `a` 的开头插入一个元素：

```
table.move(a, 1, #a, 2)
a[1] = newElement
```

如下的代码可以删除第一个元素：

```
table.move(a, 2, #a, 1)
a[#a] = nil
```

应该注意，在计算机领域，移动 (*move*) 实际上是将一个值从一个地方拷贝 (*copy*) 到另一个地方。因此，像上面的例子一样，我们必须在移动后显式地把最后一个元素删除。

函数 `table.move` 还支持使用一个表作为可选的参数。当带有可选的表作为参数时，该函数将第一个表中的元素移动到第二个表中。例如，`table.move(a, 1, #a, 1, {})` 返回列表 `a` 的一个克隆 (*clone*)（通过将列表 `a` 中的所有元素拷贝到新列表中），`table.move(a, 1, #a, #b + 1, b)` 将列表 `a` 中的所有元素复制到列表 `b` 的末尾^①。

5.7 练习

练习 5.1：下列代码的输出是什么？为什么？

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

练习 5.2：考虑如下代码：

```
a = {};
a.a = a
```

`a.a.a.a` 的值是什么？其中的每个 `a` 都一样吗？

如果将如下代码追加到上述的代码中：

```
a.a.a.a = 3
```

^①译者注：在计算机领域中，移动的概念实际是依赖于具体实现的，原文有两层含义，一方面想说明在 Lua 语言中不带第二个表参数的 `table.move` 对被移动的元素不默认进行删除（与之对应的是被移出的元素默认赋为空值），另一方面想说明带第二个参数的 `table.move` 也不会对第一个表进行改动，也就是原文中所谓的拷贝。

现在 $a.a.a.a$ 的值变成了什么？

练习 5.3: 假设要创建一个以转义序列为值、以转义序列对应字符串为键的表(参见4.1节),请问应该如何编写构造器?

练习 5.4: 在 Lua 语言中, 我们可以使用由系数组成的列表 $\{a_0, a_1, \dots, a_n\}$ 来表达多项式 $a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x^1 + a_0$ 。

请编写一个函数，该函数以多项式（使用表表示）和值 x 为参数，返回结果为对应多项式的值。

练习 5.5：改写上述函数，使之最多使用 n 个加法和 n 个乘法（且没有指数）。

练习 5.6：请编写一个函数，该函数用于测试指定的表是否为有效的序列。

练习 5.7: 请编写一个函数, 该函数将指定列表的所有元素插入到另一个列表的指定位置。

练习 5.8：表标准库中提供了函数 `table.concat`，该函数将指定表的字符串元素连接在一起：

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

请实现该函数，并比较在大数据量（具有上百万个元素的表，可利用 **for** 循环生成）情况下与标准库之间的性能差异。

6

函数

在 Lua 语言中，函数（Function）是对语句和表达式进行抽象的主要方式。函数既可以用 于完成某种特定任务（有时在其他语言中也称为过程（procedure）或子例程（subroutine）），也可以只是进行一些计算然后返回计算结果。在前一种情况下，我们将一句函数调用视为一条语句；而在后一种情况下，我们则将函数调用视为表达式：

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

无论哪种情况，函数调用时都需要使用一对圆括号把参数列表括起来。即使被调用的函数不需要参数，也需要一对空括号（）。对于这个规则，唯一的例外就是，当函数只有一个参数且该参数是字符串常量或表构造器时，括号是可选的：

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

Lua 语言也为面向对象风格的调用（object-oriented call）提供了一种特殊的语法，即冒号操作符。形如 `o:foo(x)` 的表达式意为调用对象 `o` 的 `foo` 方法。在第21章中，我们会继续学习这种调用方式及面向对象编程。

一个 Lua 程序既可以调用 Lua 语言编写的函数，也可以调用 C 语言（或者宿主程序使用的其他任意语言）编写的函数。一般来说，我们选择使用 C 语言编写的函数来实现对性能要求更高，或不容易直接通过 Lua 语言进行操作的操作系统机制等。例如，Lua 语言标准库中所有的函数就都是使用 C 语言编写的。不过，无论一个函数是用 Lua 语言编写的还是用 C 语言编写的，在调用它们时都没有任何区别。

正如我们已经在其他示例中所看到的，Lua 语言中的函数定义的常见语法格式形如：

```
-- 对序列'a'中的元素求和
function add (a)
    local sum = 0
    for i = 1, #a do
        sum = sum + a[i]
    end
    return sum
end
```

在这种语法中，一个函数定义具有一个函数名 (*name*, 本例中的 *add*)、一个参数 (*parameter*) 组成的列表和由一组语句组成的函数体 (*body*)。参数的行为与局部变量的行为完全一致，相当于一个用函数调用时传入的值进行初始化的局部变量。

调用函数时使用的参数个数可以与定义函数时使用的参数个数不一致。Lua 语言会通过抛弃多余参数和将不足的参数设为 nil 的方式来调整参数的个数。例如，考虑如下的函数：

```
function f (a, b) print(a, b) end
```

其行为如下：

f()	-->	nil	nil
f(3)	-->	3	nil
f(3, 4)	-->	3	4
f(3, 4, 5)	-->	3	4 (5被丢弃)

虽然这种行为可能导致编程错误（在单元测试中容易发现），但同样又是有用的，尤其是对于默认参数（default argument）的情况。例如，考虑如下递增全局计数器的函数：

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

该函数以 1 作为默认实参，当调用无参数的 `incCount()` 时，将 `globalCounter` 加 1。在调用 `incCount()` 时，Lua 语言首先把参数 `n` 初始化为 `nil`，接下来的 `or` 表达式又返回了其第二个操作数，最终把 `n` 赋成了默认值 1。

6.1 多返回值

Lua 语言中一种与众不同但又非常有用的特性是允许一个函数返回多个结果（Multiple Results）。Lua 语言中几个预定义函数就会返回多个值。我们已经接触过函数 `string.find`，该函数用于在字符串中定位模式（pattern）。当找到了对应的模式时，该函数会返回两个索引值：所匹配模式在字符串中起始字符和结尾字符的索引。使用多重赋值（multiple assignment）可以同时获取到这两个结果：

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

请记住，字符串的第一个字符的索引值为 1。

Lua 语言编写的函数同样可以返回多个结果，只需在 `return` 关键字后列出所有要返回的值即可。例如，一个用于查找序列中最大元素的函数可以同时返回最大值及该元素的位置：

```
function maximum (a)
    local mi = 1          -- 最大值的索引
    local m = a[mi]        -- 最大值
    for i = 1, #a do
        if a[i] > m then
            mi = i; m = a[i]
        end
    end
    return m, mi          -- 返回最大值及其索引
end

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

Lua 语言根据函数的被调用情况调整返回值的数量。当函数被作为一条单独语句调用时，其所有返回值都会被丢弃；当函数被作为表达式（例如，加法的操作数）调用时，将只保留函数的第一个返回值。只有当函数调用是一系列表达式中的最后一个表达式（或是唯一一个

表达式)时,其所有的返回值才能被获取到。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、表构造器和**return**语句。为了分别展示这几种情况,接下来举几个例子:

```
function foo0 () end          -- 不返回结果
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,如果一个函数调用是一系列表达式中的最后(或者是唯一)一个表达式,则该函数调用将产生尽可能多的返回值以匹配待赋值变量:

```
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"
```

在多重赋值中,如果一个函数没有返回值或者返回值个数不够多,那么Lua语言会用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
```

请注意,只有当函数调用是一系列表达式中的最后(或者是唯一)一个表达式时才能返回多值结果,否则只能返回一个结果:

```
x,y = foo2(), 20      -- x="a", y=20    ('b'被丢弃)
x,y = foo0(), 20, 30  -- x=nil, y=20    (30被丢弃)
```

当一个函数调用是另一个函数调用的最后一个(或者是唯一)实参时,第一个函数的所有返回值都会被作为实参传给第二个函数。我们已经见到过很多这样的代码结构,例如函数**print**。由于函数**print**能够接收可变数量的参数,所以**print(g())**会打印出g返回的所有结果。

<code>print(foo0())</code>	<code>--></code>	(没有结果)
<code>print(foo1())</code>	<code>--></code>	a
<code>print(foo2())</code>	<code>--></code>	a b
<code>print(foo2(), 1)</code>	<code>--></code>	a 1
<code>print(foo2() .. "x")</code>	<code>--></code>	ax (后详)



当在表达式中调用 `foo2` 时，Lua 语言会把其返回值的个数调整为 1。因此，在上例的最后一行，只有第一个返回值 "`a`" 参与了字符串连接操作。

当我们调用 `f(g())` 时，如果 `f` 的参数是固定的，那么 Lua 语言会把 `g` 返回值的个数调整成与 `f` 的参数个数一致。这并非巧合，实际上这正是多重赋值的逻辑。

表构造器会完整地接收函数调用的所有返回值，而不会调整返回值的个数：

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

不过，这种行为只有当函数调用是表达式列表中的最后一个时才有效，在其他位置上的函数调用总是只返回一个结果：

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

最后，形如 `return f()` 的语句会返回 `f` 返回的所有结果：

```
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))      -- (无结果)
print(foo(3))      -- (无结果)
```

将函数调用用一对圆括号括起来可以强制其只返回一个结果：

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

应该意识到，`return` 语句后面的内容是不需要加括号的，如果加了括号会导致程序出现额外的行为。因此，无论 `f` 究竟返回几个值，形如 `return (f(x))` 的语句只返回一个值。有时这可能是我们所希望出现的情况，但有时又可能不是。



6.2 可变长参数函数

Lua 语言中的函数可以是可变长参数函数 (*variadic*)，即可以支持数量可变的参数。例如，我们已经使用一个、两个或更多个参数调用过函数 `print`。虽然函数 `print` 是在 C 语言中定义的，但也可以在 Lua 语言中定义可变长参数函数。

下面是一个简单的示例，该函数返回所有参数的总和：

```
function add (...)  
    local s = 0  
    for _, v in ipairs{...} do  
        s = s + v  
    end  
    return s  
end  
  
print(add(3, 4, 10, 25, 12)) --> 54
```

参数列表中的三个点 (...) 表示该函数的参数是可变长的。当这个函数被调用时，Lua 内部会把它的所有参数收集起来，我们把这些被收集起来的参数称为函数的额外参数 (*extra argument*)。当函数要访问这些参数时仍需用到三个点，但不同的是此时这三个点是作为一个表达式来使用的。在上例中，表达式 {...} 的结果是一个由所有可变长参数组成的列表，该函数会遍历该列表来累加其中的元素。

我们将三个点组成的表达式称为可变长参数表达式 (*vararg expression*)，其行为类似于一个具有多个返回值的函数，返回的是当前函数的所有可变长参数。例如，`print(...)` 会打印出该函数的所有参数。又如，如下的代码创建了两个局部变量，其值为前两个可选的参数（如果参数不存在则为 nil）：

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

实际上，可以通过变长参数来模拟 Lua 语言中普通的参数传递机制，例如：

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

可以写成：

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



喜欢 Perl 参数传递机制的人可能会更喜欢第二种形式。

形如下例的函数只是将调用它时所传入的所有参数简单地返回：

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

该函数是一个多值恒等式函数（multi-value identity function）。下列函数的行为则类似于直接调用函数 `foo`，唯一不同之处是在调用函数 `foo` 之前会先打印出传递给函数 `foo` 的所有参数：

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

当跟踪对某个特定的函数调用时，这个技巧很有用。

接下来再让我们看另外一个很有用的示例。Lua 语言提供了专门用于格式化输出的函数 `string.format` 和输出文本的函数 `io.write`。我们会很自然地想到把这两个函数合并为一个具有可变长参数的函数：

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

注意，在三个点前有一个固定的参数 `fmt`。具有可变长参数的函数也可以具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 语言会先将前面的参数赋给固定参数，然后将剩余的参数（如果有）作为可变长参数。

要遍历可变长参数，函数可以使用表达式 `{...}` 将可变长参数放在一个表中，就像 `add` 示例中所做的那样。不过，在某些罕见的情况下，如果可变长参数中包含无效的 `nil`，那么`{...}` 获得的表可能不再是一个有效的序列。此时，就没有办法在表中判断原始参数究竟是不是以 `nil` 结尾的。对于这种情况，Lua 语言提供了函数 `table.pack`。^① 该函数像表达式`{...}`一样保存所有的参数，然后将其放在一个表中返回，但是这个表还有一个保存了参数个数的额外字段“`n`”。例如，下面的函数使用了函数 `table.pack` 来检测参数中是否有 `nil`：

```
function nonils (...)  
    local arg = table.pack(...)  
    for i = 1, arg.n do
```

^① 该函数在 Lua 5.2 中被引入。



```
if arg[i] == nil then return false end
end
return true
end

print(nonils(2,3,nil)) --> false
print(nonils(2,3)) --> true
print(nonils()) --> true
print(nonils(nil)) --> false
```

另一种遍历函数的可变长参数的方法是使用函数 `select`。函数 `select` 总是具有一个固定的参数 `selector`, 以及数量可变的参数。如果 `selector` 是数值 `n`, 那么函数 `select` 则返回第 `n` 个参数后的所有参数; 否则, `selector` 应该是字符串 "`#`", 以便函数 `select` 返回额外参数的总数。

```
print(select(1, "a", "b", "c")) --> a    b    c
print(select(2, "a", "b", "c")) --> b    c
print(select(3, "a", "b", "c")) --> c
print(select("#", "a", "b", "c")) --> 3
```

通常, 我们在需要把返回值个数调整为 1 的地方使用函数 `select`, 因此可以把 `select(n, ...)` 认为是返回第 `n` 个额外参数的表达式。

来看一个使用函数 `select` 的典型示例, 下面是使用该函数的 `add` 函数:

```
function add (...)

local s = 0
for i = 1, select("#", ...) do
    s = s + select(i, ...)
end
return s
end
```

对于参数较少的情况, 第二个版本的 `add` 更快, 因为该版本避免了每次调用时创建一个新表。不过, 对于参数较多的情况, 多次带有很多参数调用函数 `select` 会超过创建表的开销, 因此第一个版本会更好 (特别地, 由于迭代的次数和每次迭代时传入参数的个数会随着参数的个数增长, 因此第二个版本的时间开销是二次代价 (quadratic cost) 的)。



6.3 函数 table.unpack

多重返回值还涉及一个特殊的函数 `table.unpack`。该函数的参数是一个数组，返回值为数组内的所有元素：

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

顾名思义，函数 `table.unpack` 与函数 `table.pack` 的功能相反。`pack` 把参数列表转换成 Lua 语言中一个真实的列表（一个表），而 `unpack` 则把 Lua 语言中的真实的列表（一个表）转换成一组返回值，进而可以作为另一个函数的参数被使用。

`unpack` 函数的重要用途之一体现在泛型调用（generic call）机制中。泛型调用机制允许我们动态地调用具有任意参数的任意函数。例如，在 ISO C 中，我们无法编写泛型调用的代码，只能声明可变长参数的函数（使用 `stdarg.h`）或使用函数指针来调用不同的函数。但是，我们仍然不能调用具有可变数量参数的函数，因为 C 语言中的每一个函数调用的实参数是固定的，并且每个实参的类型也是固定的。而在 Lua 语言中，却可以做到这一点。如果我们想通过数组 `a` 传入可变的参数来调用函数 `f`，那么可以写成：

```
f(table.unpack(a))
```

`unpack` 会返回 `a` 中所有的元素，而这些元素又被用作 `f` 的参数。例如，考虑如下的代码：

```
print(string.find("hello", "ll"))
```

可以使用如下的代码动态地构造一个等价的调用：

```
f = string.find  
a = {"hello", "ll"}  
  
print(f(table.unpack(a)))
```

通常，函数 `table.unpack` 使用长度操作符获取返回值的个数，因而该函数只能用于序列。不过，如果有需要，也可以显式地限制返回元素的范围：

```
print(table.unpack({"Sun", "Mon", "Tue", "Wed"}, 2, 3))  
--> Mon Tue
```

虽然预定义的函数 `unpack` 是用 C 语言编写的，但是也可以利用递归在 Lua 语言中实现：



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

在第一次调用该函数时，只传入一个参数，此时 *i* 为 1，*n* 为序列长度；然后，函数返回 *t*[1] 及 *unpack(t, 2, n)* 返回的所有结果，而 *unpack(t, 2, n)* 又会返回 *t*[2] 及 *unpack(t, 3, n)* 返回的所有结果，依此类推，直到处理完 *n* 个元素为止。

6.4 正确的尾调用

Lua 语言中有关函数的另一个有趣的特性是，Lua 语言是支持尾调用消除（tail-call elimination）的。这意味着 Lua 语言可以正确地（properly）尾递归（tail recursive），虽然尾调用消除的概念并没有直接涉及递归，参见练习 6.6。

尾调用（tail call）是被当作函数调用使用的跳转^①。当一个函数的最后一个动作是调用另一个函数而没有再进行其他工作时，就形成了尾调用。例如，下列代码中对函数 *g* 的调用就是尾调用：

```
function f (x) x = x + 1; return g(x) end
```

当函数 *f* 调用完函数 *g* 之后，*f* 不再需要进行其他的工作。这样，当被调用的函数执行结束后，程序就不再需要返回最初的调用者。因此，在尾调用之后，程序也就不需要在调用栈中保存有关调用函数的任何信息。当 *g* 返回时，程序的执行路径会直接返回到调用 *f* 的位置。在一些语言的实现中，例如 Lua 语言解释器，就利用了这个特点，使得在进行尾调用时不使用任何额外的栈空间。我们就将这种实现称为尾调用消除（tail-call elimination）。

由于尾调用不会使用栈空间，所以一个程序中能够嵌套的尾调用的数量是无限的。例如，下列函数支持任意的数字作为参数：

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

^①译者注：原文为 A tail call is a goto dressed as a call。



```
end
```

该函数永远不会发生栈溢出。

关于尾调用消除的一个重点就是如何判断一个调用是尾调用。很多函数调用之所以不是尾调用，是由于这些函数在调用之后还进行了其他工作。例如，下例中调用 `g` 就不是尾调用：

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

这个示例的问题在于，当调用完 `g` 后，`f` 在返回前还不得不丢弃 `g` 返回的所有结果。类似地，以下的所有调用也都不符合尾调用的定义：

```
return g(x) + 1      -- 必须进行加法  
return x or g(x)    -- 必须把返回值限制为1个  
return (g(x))       -- 必须把返回值限制为1个
```

在 Lua 语言中，只有形如 `return func(args)` 的调用才是尾调用。不过，由于 Lua 语言会在调用前对 `func` 及其参数求值，所以 `func` 及其参数都可以是复杂的表达式。例如，下面的例子就是尾调用：

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

6.5 练习

练习 6.1：请编写一个函数，该函数的参数为一个数组，打印出该数组的所有元素。

练习 6.2：请编写一个函数，该函数的参数为可变数量的一组值，返回值为除第一个元素之外的其他所有值。

练习 6.3：请编写一个函数，该函数的参数为可变数量的一组值，返回值为除最后一个元素之外的其他所有值。

练习 6.4：请编写一个函数，该函数用于打乱（shuffle）一个指定的数组。请保证所有的排列都是等概率的。

练习 6.5：请编写一个函数，其参数为一个数组，返回值为数组中元素的所有组合。提示：可以使用组合的递推公式 $C(n, m) = C(n - 1, m - 1) + C(n - 1, m)$ 。要计算从 n 个元素中选出 m 个组成的组合 $C(n, m)$ ，可以先将第一个元素加到结果集中，然后计算所有的其他元素的 $C(n - 1, m - 1)$ ；然后，从结果集中删掉第一个元素，再计算其他所有剩余元素的 $C(n - 1, m)$ 。当 n 小于 m 时，组合不存在；当 m 为 0 时，只有一种组合（一个元素也没有）。



练习 6.6: 有时, 具有正确尾调用 (proper-tail call) 的语句被称为正确的尾递归 (*properly tail recursive*), 争论在于这种正确性只与递归调用有关 (如果没有递归调用, 那么一个程序的最大调用深度是静态固定的)。

请证明上述争论的观点在像 Lua 语言一样的动态语言中不成立：不使用递归，编写一个能够实现支持无限调用链（unbounded call chain）的程序（提示：参考6.1节）。



7

输入输出

由于 Lua 语言强调可移植性和嵌入性，所以 Lua 语言本身并没有提供太多与外部交互的机制。在真实的 Lua 程序中，从图形、数据库到网络的访问等大多数 I/O 操作，要么由宿主程序实现，要么通过不包括在发行版中的外部库实现。单就 Lua 语言而言，只提供了 ISO C 语言标准支持的功能，即基本的文件操作等。在这一章中，我们将会学习标准库如何支持这些功能。

7.1 简单 I/O 模型

对于文件操作来说，I/O 库提供了两种不同的模型。简单模型虚拟了一个当前输入流（*current input stream*）和一个当前输出流（*current output stream*），其 I/O 操作是通过这些流实现的。I/O 库把当前输入流初始化为进程的标准输入（C 语言中的 `stdin`），将当前输出流初始化为进程的标准输出（C 语言中的 `stdout`）。因此，当执行类似于 `io.read()` 这样的语句时，就可以从标准输入中读取一行。

函数 `io.input` 和函数 `io.output` 可以用于改变当前的输入输出流。调用 `io.input(filename)` 会以只读模式打开指定文件，并将文件设置为当前输入流。之后，所有的输入都来自该文件，除非再次调用 `io.input`。对于输出而言，函数 `io.output` 的逻辑与之类似。如果出现错误，这两个函数都会抛出异常。如果想直接处理这些异常，则必须使用完整 I/O 模型。

由于函数 `write` 比函数 `read` 简单，我们首先来看函数 `write`。函数 `io.write` 可以读取任意数量的字符串（或者数字）并将其写入当前输出流。由于调用该函数时可以使用多个参



数，因此应该避免使用 `io.write(a..b..c)`，应该调用 `io.write(a, b, c)`，后者可以用更少的资源达到同样的效果，并且可以避免更多的连接动作。

作为原则，应该只在“用后即弃”的代码或调试代码中使用函数 `print`；当需要完全控制输出时，应该使用函数 `io.write`。与函数 `print` 不同，函数 `io.write` 不会在最终的输出结果中添加诸如制表符或换行符这样的额外内容。此外，函数 `io.write` 允许对输出进行重定向，而函数 `print` 只能使用标准输出。最后，函数 `print` 可以自动为其参数调用 `tostring`，这一点对于调试而言非常便利，但这也容易导致一些诡异的 Bug。

函数 `io.write` 在将数值转换为字符串时遵循一般的转换规则；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
```

函数 `io.read` 可以从当前输入流中读取字符串，其参数决定了要读取的数据：^①

"a"	读取整个文件
"l"	读取下一行（丢弃换行符）
"L"	读取下一行（保留换行符）
"n"	读取一个数值
<i>num</i>	以字符串读取 <i>num</i> 个字符

调用 `io.read("a")` 可从当前位置开始读取当前输入文件的全部内容。如果当前位置处于文件的末尾或文件为空，那么该函数返回一个空字符串。

因为 Lua 语言可以高效地处理长字符串，所以在 Lua 语言中编写过滤器（filter）的一种简单技巧就是将整个文件读取到一个字符串中，然后对字符串进行处理，最后输出结果为：

```
t = io.read("a")                      -- 读取整个文件
t = string.gsub(t, "bad", "good")      -- 进行处理
io.write(t)                            -- 输出结果
```

^① 在 Lua 5.2 及更早版本中，所有字符串选项之前要有一个星号。出于兼容性考虑，Lua 5.3 也可以支持星号。

举一个更加具体的例子，以下是一段将某个文件的内容使用 MIME 可打印字符引用编码 (*quoted-printable*) 进行编码的代码。这种编码方式将所有非 ASCII 字符编码为 =xx，其中 xx 是这个字符的十六进制。为保证编码的一致性，等号也会被编码：

```
t = io.read("all")
t = string.gsub(t, "([\128-\255])", function (c)
    return string.format("=%02X", string.byte(c))
end)
io.write(t)
```

函数 `string.gsub` 会匹配所有的等号及非 ASCII 字符（从 128 到 255），并调用指定的函数完成替换（在第 10 章中会讨论有关模式匹配的细节）。

调用 `io.read("l")` 会返回当前输入流的下一行，不包括换行符在内；调用 `io.read("L")` 与之类似，但会保留换行符（如果文件中存在）。当到达文件末尾时，由于已经没有内容可以返回，该函数会返回 `nil`。选项 "`l`" 是函数 `read` 的默认参数。我通常只在逐行处理数据的算法中使用该参数，其他情况则更倾向于使用选项 "`a`" 一次性地读取整个文件，或者像后续介绍的按块（`block`）读取。

作为面向行的（line-oriented）输入的一个简单例子，以下的程序会在将当前输入复制到当前输出中的同时对每行进行编号：

```
for count = 1, math.huge do
    local line = io.read("L")
    if line == nil then break end
    io.write(string.format("%6d  ", count), line)
end
```

不过，如果要逐行迭代一个文件，那么使用 `io.lines` 迭代器会更简单：

```
local count = 0
for line in io.lines() do
    count = count + 1
    io.write(string.format("%6d  ", count), line, "\n")
end
```

另一个面向行的输入的例子参见示例 7.1，其中给出了一个对文件中的行进行排序的完整程序。

示例 7.1 对文件进行排序的程序

```
local lines = {}
```

-- 将所有行读取到表'lines'中

```
for line in io.lines() do
```

```
    lines[#lines + 1] = line
```

```
end
```

-- 排序

```
table.sort(lines)
```

-- 输出所有的行

```
for _, l in ipairs(lines) do
```

```
    io.write(l, "\n")
```

```
end
```

调用 `io.read("n")` 会从当前输入流中读取一个数值，这也是函数 `read` 返回值为数值（整型或者浮点型，与 Lua 语法扫描器的规则一致）而非字符串的唯一情况。如果在跳过了空格后，函数 `io.read` 仍然不能从当前位置读取到数值（由于错误的格式问题或到了文件末尾），则返回 `nil`。

除了上述这些基本的读取模式外，在调用函数 `read` 时还可以用一个数字 `n` 作为其参数：在这种情况下，函数 `read` 会从输入流中读取 `n` 个字符。如果无法读取到任何字符（处于文件末尾）则返回 `nil`；否则，则返回一个由流中最多 `n` 个字符组成的字符串。作为这种读取模式的示例，以下的代码展示了将文件从 `stdin` 复制到 `stdout` 的高效方法^①：

```
while true do
```

```
    local block = io.read(2^13)
```

-- 块大小是8KB

```
    if not block then break end
```

```
    io.write(block)
```

```
end
```

`io.read(0)` 是一个特例，它常用于测试是否到达了文件末尾。如果仍然有数据可供读取，它会返回一个空字符串；否则，则返回 `nil`。

^①译者注：实际就是上文提到的按块读取的方式。

调用函数 `read` 时可以指定多个选项，函数会根据每个参数返回相应的结果。假设有一个每行由 3 个数字组成的文件：

```
6.0      -3.23     15e12
4.3      234       1000001
...
```

如果想打印每一行的最大值，那么可以通过调用函数 `read` 来一次性地同时读取每行中的 3 个数字：

```
while true do
    local n1, n2, n3 = io.read("n", "n", "n")
    if not n1 then break end
    print(math.max(n1, n2, n3))
end
```

7.2 完整 I/O 模型

简单 I/O 模型对简单的需求而言还算适用，但对于诸如同时读写多个文件等更高级的文件操作来说就不够了。对于这些文件操作，我们需要用到完整 I/O 模型。

可以使用函数 `io.open` 来打开一个文件，该函数仿造了 C 语言中的函数 `fopen`。这个函数有两个参数，一个参数是待打开文件的文件名，另一个参数是一个模式（*mode*）字符串。模式字符串包括表示只读的 `r`、表示只写的 `w`（也可以用来删除文件中原有的内容）、表示追加的 `a`，以及另外一个可选的表示打开二进制文件的 `b`。函数 `io.open` 返回对应文件的流。当发生错误时，该函数会在返回 `nil` 的同时返回一条错误信息及一个系统相关的错误码：

```
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
```

检查错误的一种典型方法是使用函数 `assert`：

```
local f = assert(io.open(filename, mode))
```

如果函数 `io.open` 执行失败，错误信息会作为函数 `assert` 的第二个参数被传入，之后函数 `assert` 会将错误信息展示出来。

在打开文件后，可以使用方法 `read` 和 `write` 从流中读取和向流中写入。它们与函数 `read` 和 `write` 类似，但需要使用冒号运算符将它们当作流对象的方法来调用。例如，可以使用如下的代码打开一个文件并读取其中所有内容：

```
local f = assert(io.open(filename, "r"))
local t = f:read("a")
f:close()
```

关于冒号运算符的细节将会在第21章中讨论。

I/O 库提供了三个预定义的 C 语言流的句柄：`io.stdin`、`io.stdout` 和 `io.stderr`。例如，可以使用如下的代码将信息直接写到标准错误流中：

```
io.stderr:write(message)
```

函数 `io.input` 和 `io.output` 允许混用完整 I/O 模型和简单 I/O 模型。调用无参数的 `io.input()` 可以获得当前输入流，调用 `io.input(handle)` 可以设置当前输入流（类似的调用同样适用于函数 `io.output`）。例如，如果想要临时改变当前输入流，可以像这样：

```
local temp = io.input()          -- 保存当前输入流
io.input("newinput")            -- 打开一个新的当前输入流
对新的输入流进行某些操作
io.input():close()              -- 关闭当前流
io.input(temp)                  -- 恢复此前的当前输入流
```

注意，`io.read(args)` 实际上是 `io.input():read(args)` 的简写，即函数 `read` 是用在当前输入流上的。同样，`io.write(args)` 是 `io.output():write(args)` 的简写。

除了函数 `io.read` 外，还可以用函数 `io.lines` 从流中读取内容。正如之前的示例中展示的那样，函数 `io.lines` 返回一个可以从流中不断读取内容的迭代器。给函数 `io.lines` 提供一个文件名，它就会以只读方式打开对该文件的输入流，并在到达文件末尾后关闭该输入流。若调用时不带参数，函数 `io.lines` 就从当前输入流读取。我们也可以把函数 `lines` 当作句柄的一个方法。此外，从 Lua 5.2 开始，函数 `io.lines` 可以接收和函数 `io.read` 一样的参数。例如，下面的代码会以在 8KB 为块迭代，将当前输入流中的内容复制到当前输出流中：

```

for block in io.input():lines(2^13) do
    io.write(block)
end

```

7.3 其他文件操作

函数 `io.tmpfile` 返回一个操作临时文件的句柄，该句柄是以读/写模式打开的。当程序运行结束后，该临时文件会被自动移除（删除）。

函数 `flush` 将所有缓冲数据写入文件。与函数 `write` 一样，我们也可以把它当作 `io.flush()` 使用，以刷新当前输出流；或者把它当作方法 `f:flush()` 使用，以刷新流 `f`。

函数 `setvbuf` 用于设置流的缓冲模式。该函数的第一个参数是一个字符串：“`no`”表示无缓冲，“`full`”表示在缓冲区满时或者显式地刷新文件时才写入数据，“`line`”表示输出一直被缓冲直到遇到换行符或从一些特定文件（例如终端设备）中读取到了数据。对于后两个选项，函数 `setvbuf` 支持可选的第二个参数，用于指定缓冲区大小。

在大多数系统中，标准错误流（`io.stderr`）是不被缓冲的，而标准输出流（`io.stdout`）按行缓冲。因此，当向标准输出中写入了不完整的行（例如进度条）时，可能需要刷新这个输出流才能看到输出结果。

函数 `seek` 用来获取和设置文件的当前位置，常常使用 `f:seek(whence, offset)` 的形式来调用，其中参数 `whence` 是一个指定如何使用偏移的字符串。当参数 `whence` 取值为“`set`”时，表示相对于文件开头的偏移；取值为“`cur`”时，表示相对于文件当前位置的偏移；取值为“`end`”时，表示相对于文件尾部的偏移。不管 `whence` 的取值是什么，该函数都会以字节为单位，返回当前新位置在流中相对于文件开头的偏移。

`whence` 的默认值是“`cur`”，`offset` 的默认值是 0。因此，调用函数 `file:seek()` 会返回当前的位置且不改变当前位置；调用函数 `file:seek("set")` 会将位置重置到文件开头并返回 0；调用函数 `file:seek("end")` 会将当前位置重置到文件结尾并返回文件的大小。下面的函数演示了如何在不修改当前位置的情况下获取文件大小：

```

function fsize (file)
    local current = file:seek()           -- 保存当前位置
    local size = file:seek("end")         -- 获取文件大小
    file:seek("set", current)            -- 恢复当前位置
    return size
end

```

此外，函数 `os.rename` 用于文件重命名，函数 `os.remove` 用于移除（删除）文件。需要注意的是，由于这两个函数处理的是真实文件而非流，所以它们位于 `os` 库而非 `io` 库中。

上述所有的函数在遇到错误时，均会返回 `nil` 外加一条错误信息和一个错误码。

7.4 其他系统调用

函数 `os.exit` 用于终止程序的执行。该函数的第一个参数是可选的，表示该程序的返回状态，其值可以为一个数值（0 表示执行成功）或者一个布尔值（`true` 表示执行成功）；该函数的第二个参数也是可选的，当值为 `true` 时会关闭 Lua 状态^① 并调用所有析构器释放所占用的所有内存（这种终止方式通常是非必要的，因为大多数操作系统会在进程退出时释放其占用的所有资源）。

函数 `os.getenv` 用于获取某个环境变量，该函数的输入参数是环境变量的名称，返回值为保存了该环境变量对应值的字符串：

```
print(os.getenv("HOME"))    --> /home/lua
```

对于未定义的环境变量，该函数返回 `nil`。

7.4.1 运行系统命令

函数 `os.execute` 用于运行系统命令，它等价于 C 语言中的函数 `system`。该函数的参数为表示待执行命令的字符串，返回值为命令运行结束后的状态。其中，第一个返回值是一个布尔类型，当为 `true` 时表示程序成功运行完成；第二个返回值是一个字符串，当为 "exit" 时表示程序正常运行结束，当为 "signal" 时表示因信号而中断；第三个返回值是返回状态（若该程序正常终结）或者终结该程序的信号代码。例如，在 POSIX 和 Windows 中都可以使用如下的函数创建新目录：

```
function createDir (dirname)
    os.execute("mkdir " .. dirname)
end
```

^①译者注：请参见最后一部分的相关内容。

另一个非常有用的函数是 `io.popen`。^①同函数 `os.execute` 一样，该函数运行一条系统命令，但该函数还可以重定向命令的输入/输出，从而使得程序可以向命令中写入或从命令的输出中读取。例如，下列代码使用当前目录中的所有内容构建了一个表：

```
-- 对于POSIX系统而言，使用'ls'而非'dir'
local f = io.popen("dir /B", "r")
local dir = {}
for entry in f:lines() do
    dir[#dir + 1] = entry
end
```

其中，函数 `io.popen` 的第二个参数 "`r`" 表示从命令的执行结果中读取。由于该函数的默认行为就是这样，所以在上例中这个参数实际是可选的。

下面的示例用于发送一封邮件：

```
local subject = "some news"
local address = "someone@somewhere.org"

local cmd = string.format("mail -s '%s' '%s'", subject, address)
local f = io.popen(cmd, "w")
f:write([[

Nothing important to say.

-- me
]])
f:close()
```

注意，该脚本只能在安装了相应工具包的 POSIX 系统中运行^②。上例中函数 `io.popen` 的第二个参数是 "`w`"，表示向该命令中写入。

正如我们在上面的两个例子中看到的一样，函数 `os.execute` 和 `io.popen` 都是功能非常强大的函数，但它们也同样是非常依赖于操作系统的。

如果要使用操作系统的其他扩展功能，最好的选择是使用第三方库，比如用于基本目录操作和文件属性操作的 `LuaFileSystem`，或者提供了 POSIX.1 标准支持的 `luaposix` 库。

^①由于部分依赖的机制不是 ISO C 标准的一部分，因此该函数并非在所有的 Lua 版本中都能使用。不过，尽管标准 C 中没有该函数，但由于其在主流操作系统中存在的普遍性，所以 Lua 语言标准库还是提供了该函数。

^②译者注：即必须支持 `mail` 命令。

7.5 练习

练习 7.1：请编写一个程序，该程序读取一个文本文件然后将每行的内容按照字母表顺序排序后重写该文件。如果在调用时不带参数，则从标准输入读取并向标准输出写入；如果在调用时传入一个文件名作为参数，则从该文件中读取并向标准输出写入；如果在调用时传入两个文件名作为参数，则从第一个文件读取并将结果写入到第二个文件中。

练习 7.2：请改写上面的程序，使得当指定的输出文件已经存在时，要求用户进行确认。

练习 7.3：对比使用下列几种不同的方式把标准输入流复制到标准输出流中的 Lua 程序的性能表现：

- 按字节
- 按行
- 按块（每个块大小为 8KB）
- 一次性读取整个文件

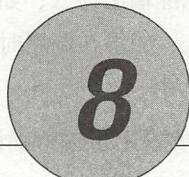
对于最后一种情况，输入文件最大支持多大？

练习 7.4：请编写一个程序，该程序输出一个文本文件的最后一行。当文件较大且可以使用 seek 时，请尝试避免读取整个文件。

练习 7.5：请将上面的程序修改得更加通用，使其可以输出一个文本文件的最后 n 行。同时，当文件较大且可以使用 seek 时，请尝试避免读取整个文件。

练习 7.6：使用函数 `os.execute` 和 `io.popen`，分别编写用于创建目录、删除目录和输出目录内容的函数。

练习 7.7：你能否使用函数 `os.execute` 来改变 Lua 脚本的当前目录？为什么？

8

补充知识

在之前的示例中，尽管我们已经使用过 Lua 语言中大部分的语法结构，但仍然容易忽略一些细节。本章作为全书第1部分的最后一章，将会补充这些被忽略的部分，介绍更多的相关细节。

8.1 局部变量和代码块

Lua 语言中的变量在默认情况下是全局变量，所有的局部变量在使用前必须声明。与全局变量不同，局部变量的生效范围仅限于声明它的代码块。一个代码块（*block*）是一个控制结构的主体，或是一个函数的主体，或是一个代码段（即变量被声明时所在的文件或字符串）：

```
x = 10
local i = 1           -- 对于代码段来说是局部的
```

```
while i <= x do
    local x = i * 2 -- 对于循环体来说是局部的
    print(x)         --> 2, 4, 6, 8, ...
    i = i + 1
end
```

```

if i > 20 then
    local x          -- 对于"then"来说是局部的
    x = 20
    print(x + 2)   -- (如果测试成功会输出22)
else
    print(x)        --> 10 (全局的)
end

print(x)          --> 10 (全局的)

```

请注意，上述示例在交互模式中不能正常运行。因为在交互模式中，每一行代码就是一个代码段（除非不是一条完整的命令）。一旦输入示例的第二行 (`local i = 1`)，Lua 语言解释器就会直接运行它并在下一行开始一个新的代码段。这样，**局部 (local)** 的声明就超出了原来的作用范围。解决这个问题的一种方式是显式地声明整个代码块，即将它放入一对 **do-end** 中。一旦输入了 **do**，命令就只会在遇到匹配的 **end** 时才结束，这样 Lua 语言解释器就不会单独执行每一行的命令。

当需要更好地控制某些局部变量的生效范围时，**do** 程序块也同样有用：

```

local x1, x2
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'的范围在此结束
print(x1, x2)           -- 'x1'和'x2'仍在范围内

```

尽可能地使用局部变量是一种良好的编程风格。首先，局部变量可以避免由于不必要的命名而造成全局变量的混乱；其次，局部变量还能避免同一程序中不同代码部分中的命名冲突；再次，访问局部变量比访问全局变量更快；最后，局部变量会随着其作用域的结束而消失，从而使得垃圾收集器能够将其释放。

鉴于局部变量优于全局变量，有些人就认为 Lua 语言应该把变量默认视为局部的。然而，把变量默认视为局部的也有一系列的问题（例如非局部变量的访问问题）。一个更好的解决办法并不是把变量默认视为局部变量，而是在使用变量前必须先声明。Lua 语言的发行版中有一个用于全局变量检查的模块 `strict.lua`，如果试图在一个函数中对不存在的全局

变量赋值或者使用不存在的全局变量，将会抛出异常。这在开发 Lua 语言代码时是一个良好的习惯。

局部变量的声明可以包含初始值，其赋值规则与常见的多重赋值一样：多余的值被丢弃，多余的变量被赋值为 nil。如果一个声明中没有赋初值，则变量会被初始化为 nil：

```
local a, b = 1, 10
if a < b then
    print(a)    --> 1
    local a    -- '= nil' 是隐式的
    print(a)    --> nil
end          -- 结束 'then' 开始的代码块
print(a, b)  --> 1  10
```

Lua 语言中有一种常见的用法：

```
local foo = foo
```

这段代码声明了一个局部变量 `foo`，然后用全局变量 `foo` 对其赋初值（局部变量 `foo` 只有在声明之后才能被访问）。这个用法在需要提高对 `foo` 的访问速度时很有用。当其他函数改变了全局变量 `foo` 的值，而代码段又需要保留 `foo` 的原始值时，这个用法也很有用，尤其是在进行运行时动态替换（monkey patching，猴子补丁）时。即使其他代码把 `print` 动态替换成了其他函数，在 `local print = print` 语句之前的所有代码使用的还都是原先的 `print` 函数。

有些人认为在代码块的中间位置声明变量是一个不好的习惯，实际上恰恰相反：我们很少会在不赋初值的情况下声明变量，在需要时才声明变量可以避免漏掉初始化这个变量。此外，通过缩小变量的作用域还有助于提高代码的可读性。

8.2 控制结构

Lua 语言提供了一组精简且常用的控制结构（control structure），包括用于条件执行的 `if` 以及用于循环的 `while`、`repeat` 和 `for`。所有的控制结构语法上都有一个显式的终结符：`end` 用于终结 `if`、`for` 及 `while` 结构，`until` 用于终结 `repeat` 结构。

控制结构的条件表达式（condition expression）的结果可以是任何值。请记住，Lua 语言将所有不是 `false` 和 `nil` 的值当作真（特别地，Lua 语言将 0 和空字符串也当作真）。

8.2.1 if then else

if 语句先测试其条件，并根据条件是否满足执行相应的 *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
```

如果要编写嵌套的 **if** 语句，可以使用 **elseif**。它类似于在 **else** 后面紧跟一个 **if**，但可以避免重复使用 **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
```

由于 Lua 语言不支持 **switch** 语句，所以这种一连串的 **else-if** 语句比较常见。

8.2.2 while

顾名思义，当条件为真时 **while** 循环会重复执行其循环体。Lua 语言先测试 **while** 语句的条件，若条件为假则循环结束；否则，Lua 会执行循环体并不断地重复这个过程。

```

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

```

8.2.3 repeat

顾名思义，**repeat-until** 语句会重复执行其循环体直到条件为真时结束。由于条件测试在循环体之后执行，所以循环体至少会执行一次。

```

-- 输出第一个非空的行
local line
repeat
    line = io.read()
until line ~= ""
print(line)

```

和大多数其他编程语言不同，在 Lua 语言中，循环体内声明的局部变量的作用域包括测试条件：

```

-- 使用Newton-Raphson法计算'x'的平方根
local sqr = x / 2
repeat
    sqr = (sqr + x/sqr) / 2
    local error = math.abs(sqr^2 - x)
until error < x/10000      -- 局部变量'error'此时仍然可见

```

8.2.4 数值型 for

for 语句有两种形式：数值型 (*numerical*) **for** 和泛型 (*generic*) **for**。

数值型 **for** 的语法如下：

```

for var = exp1, exp2, exp3 do
    something
end

```

在这种循环中，`var` 的值从 `exp1` 变化到 `exp2` 之前的每次循环会执行 `something`，并在每次循环结束后将步长（`step`）`exp3` 增加到 `var` 上。第三个表达式 `exp3` 是可选的，若不存在，Lua 语言会默认步长值为 1。如果不想给循环设置上限，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，还需要了解一些细节。首先，在循环开始前，三个表达式都会运行一次；其次，控制变量是被 `for` 语句自动声明的局部变量，且其作用范围仅限于循环体内。一种典型的错误是认为控制变量在循环结束后仍然存在：

```
for i = 1, 10 do print(i) end
max = i      -- 可能会出错！此处的'i'是全局的
```

如果需要在循环结束后使用控制变量的值（通常在中断循环时），则必须将控制变量的值保存到另一个变量中：

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

最后，不要改变控制变量的值，随意改变控制变量的值可能产生不可预知的结果。如果要在循环正常结束前停止 `for` 循环，那么可以参考上面的例子，使用 `break` 语句。

8.2.5 泛型 for

泛型 `for` 遍历迭代函数返回的所有值，例如我们已经在很多示例中看到过的 `pairs`、`ipairs` 和 `io.lines` 等。虽然泛型 `for` 看似简单，但它的功能非常强大。使用恰当的迭代器可以在保证代码可读性的情况下遍历几乎所有的数据结构。

当然，我们也可以自己编写迭代器。尽管泛型 **for** 的使用很简单，但编写迭代函数却有不少细节需要注意。我们会在后续的第18章中继续讨论该问题。

与数值型 **for** 不同，泛型 **for** 可以使用多个变量，这些变量在每次循环时都会更新。当第一个变量变为 `nil` 时，循环终止。像数值型 **for** 一样，控制变量是循环体中的局部变量，我们也不应该在循环中改变其值。

8.3 break、return 和 goto

break 和 **return** 语句用于从当前的循环结构中跳出，**goto** 语句则允许跳转到函数中的几乎任何地方。

我们可以使用 **break** 语句结束循环，该语句会中断包含它的内层循环（例如 **for**、**repeat** 或者 **while**）；该语句不能在循环外使用。**break** 中断后，程序会紧接着被中断的循环继续执行。

return 语句用于返回函数的执行结果或简单地结束函数的运行。所有函数的最后都只有一个隐含的 **return**，因此我们不需要在每一个没有返还值的函数最后书写 **return** 语句。

按照语法，**return** 只能是代码块中的最后一句：换句话说，它只能是代码块的最后一句，或者是 **end**、**else** 和 **until** 之前的最后一句。例如，在下面的例子中，**return** 是 **then** 代码块的最后一句：

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

通常，这些地方正是使用 **return** 的典型位置，**return** 之后的语句不会被执行。不过，有时在代码块中间使用 **return** 也是很有用的。例如，在调试时我们可能不想让某个函数执行。在这种情况下，可以显式地使用一个包含 **return** 的 **do**：

```
function foo ()
    return --<< SYNTAX ERROR
    -- 'return' 是下一个代码块的最后一句
    do return end -- OK
    other statements
end
```

goto 语句用于将当前程序跳转到相应的标签处继续执行。**goto** 语句一直以来备受争议，至今仍有很多人认为它们不利于程序开发并且应该在编程语言中禁止。不过尽管如此，仍有很多语言出于很多原因保留了 **goto** 语句。**goto** 语句有很强大的功能，只要足够细心，我们就能够利用它来提高代码质量。

在 Lua 语言中，**goto** 语句的语法非常传统，即保留字 **goto** 后面紧跟着标签名，标签名可以是任意有效的标识符。标签的语法稍微有点复杂：标签名称前后各紧跟两个冒号，形如`::name::`。这个复杂的语法是有意而为的，主要是为了在程序中醒目地突出这些标签。

在使用 **goto** 跳转时，Lua 语言设置了一些限制条件。首先，标签遵循常见的可见性规则，因此不能直接跳转到一个代码块中的标签（因为代码块中的标签对外不可见）。其次，**goto** 不能跳转到函数外（注意第一条规则已经排除了跳转进一个函数的可能性）。最后，**goto** 不能跳转到局部变量的作用域。

关于 **goto** 语句典型且正确的使用方式，请参考其他一些编程语言中存在但 Lua 语言中不存在的代码结构，例如 **continue**、多级 **break**、多级 **continue**、**redo** 和局部错误处理等。**continue** 语句仅仅相当于一个跳转到位于循环体最后位置处标签的 **goto** 语句，而 **redo** 语句则相当于跳转到代码块开始位置的 **goto** 语句：

```
while some_condition do
    ::redo::
    if some_other_condition then goto continue
    else if yet_another_condition then goto redo
    end
    some code
    ::continue::
end
```

Lua 语言规范中一个很有用的细节是，局部变量的作用域终止于声明变量的代码块中的最后一个有效 (*non-void*) 语句处，标签被认为是无效 (*void*) 语句。下列代码展示了这个实用的细节：

```
while some_condition do
    if some_other_condition then goto continue end
    local var = something
    some code
    ::continue::
end
```

读者可能认为，这个 `goto` 语句跳转到了变量 `var` 的作用域内。但实际上这个 `continue` 标签出现在该代码块的最后一个有效语句后，因此 `goto` 并未跳转进入变量 `var` 的作用域内。

`goto` 语句在编写状态机时也很有用。示例 8.1 给出了一个用于检验输入是否包含偶数个 0 的程序。

示例 8.1 一个使用 `goto` 语句的状态机的示例

```

::s1:: do
    local c = io.read(1)
    if c == '0' then goto s2
    elseif c == nil then print('ok'); return
    else goto s1
    end
end

::s2:: do
    local c = io.read(1)
    if c == '0' then goto s1
    elseif c == nil then print('not ok'); return
    else goto s2
    end
end

goto s1

```

虽然可以使用更好的方式来编写这段代码，但上例中的方法有助于将一个有限自动机（finite automaton）自动地转化为 Lua 语言代码（请考虑动态代码生成（dynamic code generation））。

再举一个简单的迷宫游戏的例子。迷宫中有几个房间，每个房间的东南西北方向各有一扇门。玩家每次可以输入移动的方向，如果在这个方向上有一扇门，则玩家可以进入相应的房间，否则程序输出一个警告，玩家的最终目的是从第一个房间走到最后一个房间。

这个游戏是一个典型的状态机，当前玩家所在房间就是一个状态。为实现这个迷宫游戏，我们可以为每个房间对应的逻辑编写一段代码，然后用 `goto` 语句表示从一个房间移动到另一个房间。示例 8.2 展示了如何编写一个由 4 个房间组成的小迷宫。

示例 8.2 一个迷宫游戏

```

        goto room1 -- 起始房间
::room1:: do
    local move = io.read()
    if move == "south" then goto room3
    elseif move == "east" then goto room2
    else
        print("invalid move")
        goto room1 -- 待在同一个房间
    end
end

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

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

::room4:: do
    print("Congratulations, you won!")

```

```
end
```

对于这个简单的游戏，读者可能会发现，使用数据驱动编程（使用表来描述房间和移动）是一种更好的设计方法。不过，如果游戏中的每间房都各自不同，那么就非常适合使用这种状态机的实现方法。

8.4 练习

练习 8.1：大多数 C 语法风格的编程语言都不支持 **elseif** 结构，为什么 Lua 语言比这些语言更需要这种结构？

练习 8.2：描述 Lua 语言中实现无条件循环的 4 种不同方法，你更喜欢哪一种？

练习 8.3：很多人认为，由于 **repeat--until** 很少使用，因此在像 Lua 语言这样的简单的编程语言中最好不要出现，你怎么看？

练习 8.4：正如在 6.4 节中我们所见到的，尾部调用伪装成了 goto 语句。请用这种方法重写 8.2.5 节的迷宫游戏。每个房间此时应该是一个新函数，而每个 goto 语句都变成了一个尾部调用。

练习 8.5：请解释一下为什么 Lua 语言会限制 goto 语句不能跳出一个函数。（提示：你要如何实现这个功能？）

练习 8.6：假设 goto 语句可以跳转出函数，请说明示例 8.3 中的程序将会如何执行。

示例 8.3 一种诡异且不正确的 goto 语句的使用

```
function getlabel ()
    return function () goto L1 end
::L1::
    return 0
end

function f (n)
    if n == 0 then return getlabel()
    else
        local res = f(n - 1)
        print(n)
```

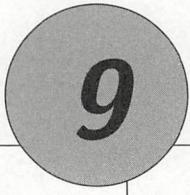
```
    return res
  end
end

x = f(10)
x()
```

请试着解释为什么标签要使用与局部变量相同的作用范围规则。

第2部分

编程实操



9

闭包

在 Lua 语言中，函数是严格遵循词法定界（lexical scoping）的第一类值（first-class value）。

“第一类值”意味着 Lua 语言中的函数与其他常见类型的值（例如数值和字符串）具有同等权限：一个程序可以将某个函数保存到变量中（全局变量和局部变量均可）或表中，也可以将某个函数作为参数传递给其他函数，还可以将某个函数作为其他函数的返回值返回。

“词法定界”意味着 Lua 语言中的函数可以访问包含其自身的外部函数中的变量（也意味着 Lua 语言完全支持 Lambda 演算）。^①

上述两个特性联合起来为 Lua 语言带来了极大的灵活性。例如，一个程序可以通过重新定义函数来增加新功能，也可以通过擦除函数来为不受信任的代码（例如通过网络接收到的代码）创建一个安全的运行时环境^②。更重要的是，上述两个特性允许我们在 Lua 语言中使

^①译者注：此处原文大致为“Lexical scoping means that functions can access variables of their enclosing functions”，实际上是指 Lua 语言中的一个函数 *A* 可以嵌套在另一个函数 *B* 中，内部的函数 *A* 可以访问外部函数 *B* 中声明的变量。原著中对此概念在此处一带而过，并未做过多解释，而是在本章“词法定界”一节中进行了说明；但实际上，即便如此，较原著中的简单解释，词法定界是具有更加明确含义的术语。建议读者阅读完“词法定界”一节后结合此处的注解一并理解。为了便于读者理解，译者认为此处非常有必要针对定界（scope）的概念进行详细解释。定界是计算机科学中的专有名词，指变量与变量所对应实体之间绑定关系的有效范围，在部分情况下也常与可见性（visibility）混用。词法定界也被称为静态定界（static scoping），常常与动态定界（dynamic scoping）比较，其中前者被大多数现代编程语言采用，后者常见于 Bash 等 Shell 语言。使用静态定界时，一个变量的可见性范围仅严格地与组成程序的静态具体词法上下文有关，而与运行时的具体堆栈调用无关；使用动态定界时，一个变量的可见性范围在编译时无法确定，依赖于运行时的实际堆栈调用情况。更加具体的例子等建议读者仔细阅读 Wiki 中有关定界的深入解释，链接为：[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))。

^②译者注：通常通过网络等方式动态加载的代码只应该具有访问其自身代码和数据的能力，而不应该具有访问除其自身代码和数据外其他固有代码和数据的能力，否则就可能出现越权或各种溢出类风险，因此可以通过在使用完成后将这些动态加载的代码擦除的方式消除由于动态加载了非受信代码而导致的安全风险。

用很多函数式语言（functional-language）的强大编程技巧。即使对函数式编程毫无兴趣，也不妨学习一下如何探索这些技巧，因为这些技巧可以使程序变得更加小巧和简单。

9.1 函数是第一类值

如前所述，Lua 语言中的函数是第一类值。以下的示例演示了第一类值的含义：

```
a = {p = print}      -- 'a.p'指向'print'函数
a.p("Hello World") --> Hello World
print = math.sin     -- 'print'现在指向sine函数
a.p(print(1))       --> 0.8414709848079
math.sin = a.p        -- 'sin'现在指向print函数
math.sin(10, 20)     --> 10      20
```

如果函数也是值的话，那么是否有创建函数的表达式呢？答案是肯定的。事实上，Lua 语言中常见的函数定义方式如下：

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

就是所谓的语法糖（*syntactic sugar*）的例子^①，它只是下面这种写法的一种美化形式：

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

赋值语句右边的表达式（`function (x) body end`）就是函数构造器，与表构造器 {} 相似。因此，函数定义实际上就是创建类型为“function”的值并把它赋值给一个变量的语句。

请注意，在 Lua 语言中，所有的函数都是匿名的（*anonymous*）。像其他所有的值一样，函数并没有名字。当讨论函数名时，比如 `print`，实际上指的是保存该函数的变量。虽然我们通常会把函数赋值给全局变量，从而看似给函数起了一个名字，但在很多场景下仍然会保留函数的匿名性^②。下面来看几个例子。

表标准库提供了函数 `table.sort`，该函数以一个表为参数并对其中的元素排序。这种函数必须支持各种各样的排序方式：升序或降序、按数值顺序或按字母顺序、按表中的键等。函数 `sort` 并没有试图穷尽所有的排序方式，而是提供了一个可选的参数，也就是所谓的排

^①译者注：语法糖也称糖衣语法，由英国计算机科学家 Peter J. Landin 发明，他最先发现了 Lambda 演算，由此而创立了函数式编程。糖衣语法意指那些没有给语言添加新功能但对程序员来说更“甜蜜”的语法，这种语法能使程序员更方便地使用语言开发程序，同时增强程序代码的可读性和避免出错。

^②译者注：即使用匿名函数。

序函数 (*order function*)，排序函数接收两个参数并根据第一个元素是否应排在第二个元素之前返回不同的值。例如，假设有一个如下所示的表：

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

如果想针对 name 字段、按字母顺序逆序对这个表排序，只需使用如下语句：

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

可见，匿名函数在这条语句中显示出了很好的便利性。

像函数 `sort` 这样以另一个函数为参数的函数，我们称之为高阶函数 (*higher-order function*)。高阶函数是一种强大的编程机制，而利用匿名函数作为参数正是其灵活性的主要来源。不过尽管如此，请记住高阶函数也并没有什么特殊的，它们只是 Lua 语言将函数作为第一类值处理所带来结果的直接体现。

为了进一步演示高阶函数的用法，让我们再来实现一个常见的高阶函数，即导数 (*derivative*)。按照通常的定义，函数 f 的导数为 $f'(x) = (f(x + d) - f(x))/d$ ，其中 d 趋向于无穷小^①。根据这个定义，可以用如下方式近似地计算导数：

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

对于指定的函数 f ，调用 `derivative(f)` 将返回（近似地）其导数，也就是另一个函数：

```
c = derivative(math.sin)
> print(math.cos(5.2), c(5.2))
--> 0.46851667130038 0.46856084325086
print(math.cos(10), c(10))
```

^①译者注：在数学领域中导数的定义方法有很多，上述定义是常见的一种近似形式。

```
--> -0.83907152907645 -0.83904432662041
```

9.2 非全局函数

由于函数是一种“第一类值”，因此一个显而易见的结果就是：函数不仅可以被存储在全局变量中，还可以被存储在表字段和局部变量中。

我们已经在前面的章节中见到过几个将函数存储在表字段中的示例，大部分 Lua 语言的库就采用了这种机制（例如 `io.read` 和 `math.sin`）。如果要在 Lua 语言中创建这种函数，只需将到目前为止我们所学到的知识结合起来：

```
Lib = {}
Lib.foo = function (x,y) return x + y end
Lib.goo = function (x,y) return x - y end
print(Lib.foo(2, 3), Lib.goo(2, 3)) --> 5 -1
```

当然，也可以使用表构造器：

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

除此以外，Lua 语言还提供了另一种特殊的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
```

正如我们将在第21章中看到的，在表字段中存储函数是 Lua 语言中实现面向对象编程的关键要素。

当把一个函数存储到局部变量时，就得到了一个局部函数（*local function*），即一个被限定在指定作用域中使用的函数。局部函数对于包（package）而言尤其有用：由于 Lua 语言将每个程序段（chunk）作为一个函数处理，所以在一段程序中声明的函数就是局部函数，这些局部函数只在该程序段中可见。词法定界保证了程序段中的其他函数可以使用这些局部函数。

对于这种局部函数的使用, Lua 语言提供了一种语法糖:

```
local function f (params)
    body
end
```

在定义局部递归函数 (recursive local function) 时, 由于原来的方法不适用, 所以有一点是极易出错的。考虑如下的代码:

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

当 Lua 语言编译函数体中的 `fact(n-1)` 调用时, 局部的 `fact` 尚未定义。因此, 这个表达式会尝试调用全局的 `fact` 而非局部的 `fact`。我们可以通过先定义局部变量再定义函数的方式来解决这个问题:

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

这样, 函数内的 `fact` 指向的是局部变量。尽管在定义函数时, 这个局部变量的值尚未确定, 但到了执行函数时, `fact` 肯定已经有了正确的赋值。

当 Lua 语言展开局部函数的语法糖时, 使用的并不是之前的基本函数定义。相反, 形如

```
local function foo (params) body end
```

的定义会被展开成

```
local foo; foo = function (params) body end
```

因此, 使用这种语法来定义递归函数不会有问题。

当然, 这个技巧对于间接递归函数 (indirect recursive function) 是无效的。在间接递归的情况下, 必须使用与明确的前向声明 (explicit forward declaration) 等价的形式:

```

local f      -- "前向" 声明

local function g ()
    some code  f() some code
end
function f ()
    some code  g() some code
end

```

请注意，不能在最后一个函数定义前加上 `local`。否则，Lua 语言会创建一个全新的局部变量 `f`，从而使得先前声明的 `f`（函数 `g` 中使用的那个）变为未定义状态。

9.3 词法定界

当编写一个被其他函数 `B` 包含的函数 `A` 时，被包含的函数 `A` 可以访问包含其的函数 `B` 的所有局部变量，我们将这种特性称为词法定界（*lexical scoping*）^①。虽然这种可见性规则听上去很明确，但实际上并非如此。词法定界外加嵌套的第一类值函数可以为编程语言提供强大的功能，但很多编程语言并不支持将这两者组合使用。

先看一个简单的例子。假设有一个表，其中包含了学生的姓名和对应的成绩，如果我们想基于分数对学生姓名排序，分数高者在前，那么可以使用如下的代码完成上述需求：

```

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

```

现在，假设我们想创建一个函数来完成这个需求：

```

function sortbygrade (names, grades)
    table.sort(names, function (n1, n2)
        return grades[n1] > grades[n2]      -- 比较分数
    end)

```

^①译者注：请注意回顾本章开始时译者注中对词法定界概念的解释。

```
end)
end
```

在后一个示例中，有趣的一点就在于传给函数 `sort` 的匿名函数可以访问 `grades`，而 `grades` 是包含匿名函数的外层函数 `sortbygrade` 的形参。在该匿名函数中，`grades` 既不是全局变量也不是局部变量，而是我们所说的非局部变量 (*non-local variable*) (由于历史原因，在 Lua 语言中非局部变量也被称为上值)。

这一点之所以如此有趣是因为，函数作为第一类值，能够逃逸 (*escape*) 出它们变量的原始定界范围。考虑如下的代码：

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

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

在上述代码中，匿名函数访问了一个非局部变量 (`count`) 并将其当作计数器。然而，由于创建变量的函数 (`newCounter`) 已经返回，因此当我们调用匿名函数时，变量 `count` 似乎已经超出了作用范围。但其实不然，由于闭包 (*closure*) 概念的存在，Lua 语言能够正确地应对这种情况。简单地说，一个闭包就是一个函数外加能够使该函数正确访问非局部变量所需的其他机制。如果我们再次调用 `newCounter`，那么一个新的局部变量 `count` 和一个新的闭包会被创建出来，这个新的闭包针对的是这个新变量：

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

因此，`c1` 和 `c2` 是不同的闭包。它们建立在相同的函数之上，但是各自拥有局部变量 `count` 的独立实例。

从技术上讲，Lua 语言中只有闭包而没有函数。函数本身只是闭包的一种原型。不过尽管如此，只要不会引起混淆，我们就仍将使用术语“函数”来指代闭包。

闭包在许多场合中均是一种有价值的工具。正如我们之前已经见到过的，闭包在作为诸如 `sort` 这样的高阶函数的参数时就非常有用。同样，闭包对于那些创建了其他函数的函数也很有用，例如我们之前的 `newCounter` 示例及求导数的示例；这种机制使得 Lua 程序能够综合运用函数式编程世界中多种精妙的编程技巧。另外，闭包对于回调（*callback*）函数来说也很有用。对于回调函数而言，一个典型的例子就是在传统 GUI 工具箱中创建按钮。每个按钮通常都对应一个回调函数，当用户按下按钮时，完成不同的处理动作的回调函数就会被调用。

例如，假设有一个具有 10 个类似按钮的数字计算器（每个按钮代表一个十进制数字），我们就可以使用如下的函数来创建这些按钮：

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

在上述示例中，假设 `Button` 是一个创建新按钮的工具箱函数，`label` 是按钮的标签，`action` 是当按钮按下时被调用的回调函数。回调可能发生在函数 `digitButton` 早已执行完后，那时变量 `digit` 已经超出了作用范围，但闭包仍可以访问它。

闭包在另一种很不一样的场景下也非常有用。由于函数可以被保存在普通变量中，因此在 Lua 语言中可以轻松地重新定义函数，甚至是预定义函数。这种机制也正是 Lua 语言灵活的原因之一。通常，当重新定义一个函数的时候，我们需要在新的实现中调用原来的那个函数。例如，假设要重新定义函数 `sin` 以使其参数以角度为单位而不是以弧度为单位。那么这个新函数就可以先对参数进行转换，然后再调用原来的 `sin` 函数进行真正的计算。代码可能形如：

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

另一种更清晰一点的完成重新定义的写法是：

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

上述代码使用了 **do** 代码段来限制局部变量 `oldSin` 的作用范围；根据可见性规则，局部变量 `oldSin` 只在这部分代码段中有效。因此，只有新版本的函数 `sin` 才能访问原来的 `sin` 函数，其他部分的代码则访问不了。

我们可以使用同样的技巧来创建安全的运行时环境（secure environment），即所谓的沙盒（sandbox）。当执行一些诸如从远程服务器上下载到的未受信任代码（untrusted code）时，安全的运行时环境非常重要。例如，我们可以通过使用闭包重定义函数 `io.open` 来限制一个程序能够访问的文件：

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

上述示例的巧妙之处在于，在经过重新定义后，一个程序就只能通过新的受限版本来调用原来未受限版本的 `io.open` 函数。示例代码将原来不安全的版本保存为闭包的一个私有变量，该变量无法从外部访问。通过这一技巧，就可以在保证简洁性和灵活性的前提下在 Lua 语言本身上构建 Lua 沙盒。相对于提供一套大而全（one-size-fits-all）的解决方案，Lua 语言提供

的是一套“元机制（meta-mechanism）”，借助这种机制可以根据特定的安全需求来裁剪具体的运行时环境（真实的沙盒除了保护外部文件外还有更多的功能，我们会在 25.4 节中再次讨论这个话题）。

9.4 小试函数式编程

再举一个函数式编程（functional programming）的具体示例。在本节中我们要开发一个用来表示几何区域的简单系统。^①我们的目标就是开发一个用来表示几何区域的系统，其中区域即为点的集合。我们希望能够利用该系统表示各种各样的图形，同时可以通过多种方式（旋转、变换、并集等）组合和修改这些图形。

为了实现这样的一个系统，首先需要找到表示这些图形的合理数据结构。我们可以尝试着使用面向对象的方案，利用继承来抽象某些图形；或者，也可以直接利用特征函数（characteristic or indicator function）来进行更高层次的抽象（集合 A 的特征函数 f_A 是指当且仅当 x 属于 A 时 $f_A(x)$ 成立）。鉴于一个几何区域就是点的集合，因此可以通过特征函数来表示一个区域，即可以提供一个点（作为参数）并根据点是否属于指定区域而返回真或假的函数来表示一个区域。

举例来说，下面的函数表示一个以点 $(1.0, 3.0)$ 为圆心、半径 4.5 的圆盘（一个圆形区域）：

```
function disk1 (x, y)
    return (x - 1.0)^2 + (y - 3.0)^2 <= 4.5^2
end
```

利用高阶函数和词法定界，可以很容易地定义一个根据指定的圆心和半径创建圆盘的工厂：

```
function disk (cx, cy, r)
    return function (x, y)
        return (x - cx)^2 + (y - cy)^2 <= r^2
    end
end
```

形如 $\text{disk}(1.0, 3.0, 4.5)$ 的调用会创建一个与 disk1 等价的圆盘。

^① 本示例源于由 Paul Hudak 和 Mark P. Jones 撰写的研究报告 *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*。

下面的函数创建了一个指定边界的轴对称矩形：

```
function rect (left, right, bottom, up)
    return function (x, y)
        return left <= x and x <= right and
            bottom <= y and y <= up
    end
end
```

按照类似的方式，可以定义函数以创建诸如三角形或非轴对称矩形等其他基本图形。每一种图形都具有完全独立的实现，所需的仅仅是一个正确的特征函数。

接下来让我们考虑一下如何改变和组合区域。我们可以很容易地创建任何区域的补集：

```
function complement (r)
    return function (x, y)
        return not r(x, y)
    end
end
```

并集、交集和差集也很简单，参见示例 9.1。

示例 9.1 区域的并集、交集和差集

```
function union (r1, r2)
    return function (x, y)
        return r1(x, y) or r2(x, y)
    end
end

function intersection (r1, r2)
    return function (x, y)
        return r1(x, y) and r2(x, y)
    end
end

function difference (r1, r2)
    return function (x, y)
        return r1(x, y) and not r2(x, y)
    end
end
```

```
    end
end
```

以下函数按照指定的增量平移指定的区域：

```
function translate (r, dx, dy)
    return function (x, y)
        return r(x - dx, y - dy)
    end
end
```

为了使一个区域可视化，我们可以遍历每个像素进行视口（viewport）测试；位于区域内的像素被绘制为黑色，而位于区域外的像素被绘制为白色。为了用简单的方式演示这个过程，我们接下来写一个函数来生成一个 PBM（可移植位图，*portable bitmap*）格式的文件来绘制指定的区域。

PBM 文件的结构很简单（这种结构也同样极为高效，但是这里强调的是简单性）。PBM 文件的文本形式以字符串 "P1" 开头，接下来的一行是图片的宽和高（以像素为单位），然后是对应每一个像素、由 1 和 0 组成的数字序列（黑为 1，白为 0，数字和数字之间由可选的空格分开），最后是 EOF。示例 9.2 中的函数 `plot` 创建了指定区域的 PBM 文件，并将虚拟绘图区域 $(-1, 1], [-1, 1]$ 映射到视口区域 $[1, M], [1, N]$ 中。

示例 9.2 在 PBM 文件中绘制区域

```
function plot (r, M, N)
    io.write("P1\n", M, " ", N, "\n")      -- 文件头
    for i = 1, N do                      -- 对每一行
        local y = (N - i^2)/N
        for j = 1, M do                  -- 对每一列
            local x = (j^2 - M)/M
            io.write(r(x, y) and "1" or "0")
        end
        io.write("\n")
    end
end
```

为了让示例更加完整，以下的代码绘制了一个南半球（southern hemisphere）所能看到的娥眉月（waxing crescent moon）：

```
c1 = disk(0, 0, 1)
plot(difference(c1, translate(c1, 0.3, 0)), 500, 500)
```

9.5 练习

练习 9.1：请编写一个函数 `integral`，该函数以一个函数 `f` 为参数并返回其积分的近似值。

练习 9.2：请问如下的代码段将输出怎样的结果：

```
function F (x)
    return {
        set = function (y) x = y end,
        get = function () return x end
    }
end

o1 = F(10)
o2 = F(20)
print(o1.get(), o2.get())
o2.set(100)
o1.set(300)
print(o1.get(), o2.get())
```

练习 9.3：练习 5.4 要求我们编写一个以多项式（使用表表示）和值 `x` 为参数、返回结果为对应多项式值的函数。请编写该函数的柯里化（*curried*）^①版本，该版本的函数应该以一个多项式为参数并返回另一个函数（当这个函数的入参是值 `x` 时返回对应多项式的值）。考虑如下的示例：

```
f = newpoly({3, 0, 1})
print(f(0))    --> 3
print(f(5))    --> 28
print(f(10))   --> 103
```

^①译者注：在计算机科学中，柯里化（Currying）又被译为卡瑞化或加里化，是指将通过一个函数对多个参数（或单个由多个参数组成的结构）求值的过程变换为对一个只接收一个参数的函数序列进行求值的技巧。柯里化在实践和理论上均非常有用，更多详情可参考如下链接：<https://en.wikipedia.org/wiki/Currying>。

练习 9.4：使用几何区域系统的例子，绘制一个北半球（northern hemisphere）所能看到的娥眉月（waxing crescent moon）。

练习 9.5：在几何区域系统的例子中，增加一个函数来实现将指定的区域旋转指定的角度。



酒四友对

五时九月九日，为酒友倾玉。POSIX 用剪齐酒酒肴所用，同不首阳本脚转其酒其。
一：面向小大千奇因那要生的酒肴有以泡水。（quitting wine）酒叫先烈齐酒来去赤青烟
随小大益酒新种管探酒食出益，酒升益 000+1 长歌要酒海尖头占势倾玉。POSIX 用脚真个
大菊的酒肴。酒食只 000 但不育只脚分的实酒酒酒酒酒酒。不玄出叶。大丕半一
强轴间，大要常非然脚酒酒酒酒酒酒。酒故音酒酒酒酒。酒故音酒酒酒酒。酒故音酒酒酒酒。
虽农神美殿太已正又明同不 ZOSIX 酒冠已造一宵具

姚函关卧的酒四友对 1.01

buit 烧酒长罪了进呼登日日寒。姚函个 4 酒（wining），大努干基丁进酒酒酒酒串串
盈长呼其挂酒。《晋书》John M. Gould) datapmp 耐 d5fom 墓歌长歌两个重余其，dug 吻
苦睡酒姚函个山

buit. guring 酒函 1.1.01

个一量脚友题面单商景。左藉如宝酒酒酒中串桥守制目的歌诗道于甲 buit. guring 酒函
。“offed”串于察卦中串桥不被日召会“offed”为群。酸闻。良本辨单个玄酒酒酒酒只空”。同单
置音朱东峰持索曲置负部升方熟医酒四；两个两回致会。百互歌个一仰非 buit. guring 酒函
。fli 回祖歌。酒叫叫音底酒音底效果报。比素唯
以

10

模式匹配

与其他几种脚本语言不同，Lua 语言既没有使用 POSIX 正则表达式，也没有使用 Perl 正则表达式来进行模式匹配（pattern matching）。之所以这样做的主要原因在于大小问题：一个典型的 POSIX 正则表达式实现需要超过 4000 行代码，这比所有 Lua 语言标准库总大小的一半还大。相比之下，Lua 语言模式匹配的实现代码只有不到 600 行。尽管 Lua 语言的模式匹配做不到完整 POSIX 实现的所有功能，但是 Lua 语言的模式匹配仍然非常强大，同时还具有一些与标准 POSIX 不同但又可与之媲美的功能。

10.1 模式匹配的相关函数

字符串标准库提供了基于模式（pattern）的 4 个函数。我们已经初步了解过函数 `find` 和 `gsub`，其余两个函数分别是 `match` 和 `gmatch`（*Global Match* 的缩写）。现在让我们学习这几个函数的细节。

10.1.1 函数 `string.find`

函数 `string.find` 用于在指定的目标字符串中搜索指定的模式。最简单的模式就是一个单词，它只会匹配到这个单词本身。例如，模式'`hello`'会在目标字符串中搜索子串"`hello`"。函数 `string.find` 找到一个模式后，会返回两个值：匹配到模式开始位置的索引和结束位置的索引。如果没有找到任何匹配，则返回 `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
```

匹配成功后，可以以函数 `find` 返回的结果为参数调用函数 `string.sub` 来获取目标字符串中匹配相应模式的子串。对于简单的模式来说，这一般就是模式本身。

函数 `string.find` 具有两个可选参数。第 3 个参数是一个索引，用于说明从目标字符串的哪个位置开始搜索。第 4 个参数是一个布尔值，用于说明是否进行简单搜索（plain search）。字如其名，所谓简单搜索就是忽略模式而在目标字符串中进行单纯的“查找子字符串”的动作：

```
> string.find("a [word]", "[")
stdin:1: malformed pattern (missing ']')
> string.find("a [word]", "[", 1, true) --> 3 3
```

由于'['在模式中具有特殊含义，因此第 1 个函数调用会报错。在第 2 个函数调用中，函数只是把'['当作简单字符串。请注意，如果没有第 3 个参数，是不能传入第 4 个可选参数的。

10.1.2 函数 `string.match`

由于函数 `string.match` 也用于在一个字符串中搜索模式，因此它与函数 `string.find` 非常相似。不过，函数 `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
```

对于诸如'hello'这样固定的模式，使用这个函数并没有什么意义。然而，当模式是变量时，这个函数的强大之处就显现出来了，例如：

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
print(d) --> 17/7/1990
```

后续，我们会讨论模式'%'的含义及函数 string.match 的更高级用法。

10.1.3 函数 string.gsub

函数 string.gsub 有 3 个必选参数：目标字符串、模式和替换字符串（replacement string），其基本用法是将目标字符串中所有出现模式的地方换成替换字符串：

```
s = string.gsub("Lua is cute", "cute", "great")
print(s)      --> Lua is great
s = string.gsub("all lii", "l", "x")
print(s)      --> axx xii
s = string.gsub("Lua is great", "Sol", "Sun")
print(s)      --> Lua is great
```

此外，该函数还有一个可选的第 4 个参数，用于限制替换的次数：

```
s = string.gsub("all lii", "l", "x", 1)
print(s)      --> axl lii
s = string.gsub("all lii", "l", "x", 2)
print(s)      --> axx lii
```

除了替换字符串以外，string.gsub 的第 3 个参数也可以是一个函数或一个表，这个函数或表会被调用（或检索）以产生替换字符串；我们会在 10.4 节中学习这个功能。

函数 string.gsub 还会返回第 2 个结果，即发生替换的次数。

10.1.4 函数 string.gmatch

函数 string.gmatch 返回一个函数，通过返回的函数可以遍历一个字符串中所有出现的指定模式。例如，以下示例可以找出指定字符串 s 中出现的所有单词：

```
s = "some string"
words = {}
for w in string.gmatch(s, "%a+") do
    words[#words + 1] = w
end
```

后续我们马上会学习到，模式'`%a+`'会匹配一个或多个字母组成的序列（也就是单词）。因此，`for`循环会遍历所有目标字符串中的单词，然后把它们保存到列表`words`中。

10.2 模式

大多数模式匹配库都使用反斜杠（backslash）作为转义符。然而，这种方式可能会导致一些不良的后果。对于Lua语言的解析器而言，模式仅仅是普通的字符串。模式与其他的字符串一样遵循相同的规则，并不会被特殊对待；只有模式匹配相关的函数才会把它们当作模式进行解析。由于反斜杠是Lua语言中的转义符，所以我们应该避免将它传递给任何函数。模式本身就难以阅读，到处把"`\`"换成"`\\`"就更加火上浇油了。

我们可以使用双括号把模式括起来构成的长字符串来解决这个问题（某些语言在实践中推荐这种办法）。然而，长字符串的写法对于通常比较短的模式而言又往往显得冗长。此外，我们还会失去在模式内进行转义的能力（某些模式匹配工具通过再次实现常见的字符串转义来绕过这种限制）。

Lua语言的解决方案更加简单：Lua语言中的模式使用百分号（percent sign）作为转义符（C语言中的一些函数采用的也是同样的方式，如函数`printf`和函数`strftime`）。总体上，所有被转义的字母都具有某些特殊含义（例如'`%a`'匹配所有字母），而所有被转义的非字母则代表其本身（例如'`%.1`'匹配一个点）。

我们首先来学习字符分类（character class）的模式。所谓字符分类，就是模式中能够与一个特定集合中的任意字符相匹配的一项。例如，分类`%d`匹配的是任意数字。因此，可以使用模式'`%d%d/%d%d/%d%d%d%d`'来匹配`dd/mm/yyyy`格式的日期：

```
s = "Deadline is 30/05/1999, firm"
date = "%d%d/%d%d/%d%d%d%d"
print(string.match(s, date)) --> 30/05/1999
```

下表列出了所有预置的字符分类及其对应的含义：

.	任意字符
<code>%a</code>	字母
<code>%c</code>	控制字符
<code>%d</code>	数字
<code>%g</code>	除空格外的可打印字符

续表

%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
%x	十六进制数字

这些类的大写形式表示类的补集。例如，'%A' 代表任意非字母的字符：

```
print((string.gsub("hello, up-down!", "%A", "."))
--> hello..up.down.
```

在输出函数 `gsub` 的返回结果时，我们使用了额外的括号来丢弃第二个结果，也就是替换发生的次数^①。

当在模式中使用时，还有一些被称为魔法字符（*magic character*）的字符具有特殊含义。Lua 语言的模式所使用的魔法字符包括：

```
( ) . % + - * ? [ ] ^ $
```

正如我们之前已经看到的，百分号同样可以用于这些魔法字符的转义。因此，'%?' 匹配一个问号，'%%' 匹配一个百分号。我们不仅可以用百分号对魔法字符进行转义，还可以将其用于其他所有字母和数字外的字符。当不确定是否需要转义时，为了保险起见就可以使用转义符。

可以使用字符集（*char-set*）来创建自定义的字符分类，只需要在方括号内将单个字符和字符分类组合起来即可。例如，字符集 '[%w_]' 匹配所有以下划线结尾的字母和数字，'[01]' 匹配二进制数字，'[[%%]]' 匹配方括号。如果想要统计一段文本中元音的数量，可以使用如下的代码：

```
_ , nvow = string.gsub(text, "[AEIOUaeiou]", "")
```

还可以在字符集中包含一段字符范围，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将它们连接在一起。由于大多数常用的字符范围都被预先定义了，所以这个功能很少

^①译者注：如果只用一个括号则还会输出替换发生的次数，也就是 4。



被使用。例如，'%d' 相当于'[0-9]'，'%x' 相当于'[0-9a-fA-F]'。不过，如果需要查找一个八进制的数字，那么使用'[0-7]'就比显式地枚举'[01234567]'强多了。

在字符集前加一个补字符^就可以得到这个字符集对应的补集：模式'^[0-7]'代表所有八进制数字以外的字符，模式'^\n'则代表除换行符以外的其他字符。尽管如此，我们还是要记得对于简单的分类来说可以使用大写形式来获得对应的补集：'%S'显然要比'^%s'更简单。

还可以通过描述模式中重复和可选部分的修饰符（modifier，在其他语言中也被译为限定符）来让模式更加有用。Lua 语言中的模式提供了 4 种修饰符：

+	重复一次或多次
*	重复零次或多次
-	重复零次或多次（最小匹配）
?	可选（出现零次或一次）

修饰符 + 匹配原始字符分类中的一个或多个字符，它总是获取与模式相匹配的最长序列。例如，模式'%a+'代表一个或多个字母（即一个单词）：

```
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 * 类似于修饰符 +，但是它还接受对应字符分类出现零次的情况。该修饰符一个典型的用法就是在模式的部分之间匹配可选的空格。例如，为了匹配像()或()这样的空括号对，就可以使用模式'%(%s*)'，其中的'%s*'匹配零个或多个空格（括号在模式中有特殊含义，所以必须进行转义）。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头，并紧跟零个或多个由下画线、字母或数字组成的序列。

修饰符-和修饰符 * 类似，也是用于匹配原始字符分类的零次或多次出现。不过，跟修饰符 * 总是匹配能匹配的最长序列不同，修饰符-只会匹配最短序列。虽然有时它们两者并没有什么区别，但大多数情况下这两者会导致截然不同的结果。例如，当试图用模式'[_%a][_%w]-'查找标识符时，由于'[_%w]-'总是匹配空序列，所以我们只会找到第一个字母。又如，假设我们想要删掉某 C 语言程序中的所有注释，通常会首先尝试使用'/*.*/*/'



（即“/*”和“*/”之间的任意序列，使用恰当的转义符对 * 进行转义）。然而，由于‘.*’会尽可能长地匹配^①，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, "/%*.*/%", ""))
--> int x;
```

相反，模式‘.-’则只会匹配到找到的第一个“*/”，这样就能得到期望的结果：

```
test = "int x; /* x */ int y; /* y */"
print(string.gsub(test, "/%*.-%*/%", ""))
--> int x; int y;
```

最后一个修饰符? 可用于匹配一个可选的字符。例如，假设我们想在一段文本中寻找一个整数，而这个整数可能包括一个可选的符号，那么就可以使用模式'[+-]?%d+' 来完成这个需求，该模式可以匹配像"-12"、"23" 和 "+1009" 这样的数字。其中，字符分类'[+-]' 匹配加号或减号，而其后的问号则代表这个符号是可选的。

与其他系统不同的是，Lua 语言中的修饰符只能作用于一个字符模式，而无法作用于一组分类。例如，我们不能写出匹配一个可选的单词的模式（除非这个单词只由一个字母组成）。通常，可以使用一些将在本章最后介绍的高级技巧来绕开这个限制。

以补字符 ^ 开头的模式表示从目标字符串的开头开始匹配。类似地，以 \$ 结尾的模式表示匹配到目标字符串的结尾。我们可以同时使用这两个标记来限制匹配查找和锚定（anchor）模式。例如，如下的代码可以用来检查字符串 s 是否以数字开头：

```
if string.find(s, "^%d") then ...
```

如下的代码用来检查字符串是否为一个没有多余前缀字符和后缀字符的整数：

```
if string.find(s, "^[-]?%d+$") then ...
```

^ 和 \$ 字符只有位于模式的开头和结尾时才具有特殊含义；否则，它们仅仅就是与其自身相匹配的普通字符。

模式'%b' 匹配成对的字符串，它的写法是'%bxy'，其中 x 和 y 是任意两个不同的字符，x 作为起始字符而 y 作为结束字符。例如，模式'%b()' 匹配以左括号开始并以对应右括号结束的子串：

^①译者注：在标准正则表达式中即为贪婪匹配。



```
s = "a (enclosed (in) parentheses) line"
print(string.gsub(s, "%b()", ""))
--> a line
```

通常，我们使用'%'、'%b[]'、'%b{}'或'%b<>'等作为模式，但实际上可以用任意不同的字符作为分隔符。

最后，模式'%f[char-set]'代表前置模式(*frontier pattern*)。该模式只有在后一个字符位于char-set内而前一个字符不在时匹配一个空字符串^①：

```
s = "the anthem is the theme"
print(string.gsub(s, "%f[%w]the%f[%W]", "one"))
--> one anthem is one theme
```

模式'%f[%w]'匹配位于一个非字母或数字的字符和一个字母或数字的字符之间的前置，而模式'%f[%W]'则匹配一个字母或数字的字符和一个非字母或数字的字符之间的前置。因此，指定的模式只会匹配完整的字符串"the"^②。请注意，即使字符集只有一个分类，也必须把它用括号括起来。

前置模式把目标字符串中第一个字符前和最后一个字符后的位置当成空字符(ASCII编码的\0)。在前例中，第一个"the"在不属于集合'[%w]'的空字符和属于集合'[%w]'的t之间匹配了一个前置。

10.3 捕获

捕获(*capture*)机制允许根据一个模式从目标字符串中抽出与该模式匹配的内容来用于后续用途，可以通过把模式中需要捕获的部分放到一对圆括号内来指定捕获。

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；换句话说，该函数会将字符串切分成多个被捕获的部分：

```
pair = "name = Anna"
key, value = string.match(pair, "(%a+)%s*=%s*(%a+)")
print(key, value)
--> name Anna
```

^①译者注：原文中前置模式的解释不够充分，实际上在<http://lua-users.org/wiki/FrontierPattern>中对此有额外说明，即The frontier pattern %f followed by a set detects the transition from “not in set” to “in set”。

^②译者注：不会匹配到上例中anthem中的the或theme中的the。



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白序列。因此，上例中的这个模式表示一个字母序列、紧跟着空白序列、一个等号、空白序列以及另一个字母序列。模式中的两个字母序列被分别放在圆括号中，因此在匹配时就能捕获到它们。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
```

在这个示例中，使用了 3 个捕获，每个捕获对应一个数字序列。

在模式中，形如'`%n`' 的分类（其中 `n` 是一个数字），表示匹配第 `n` 个捕获的副本。举一个典型的例子，假设想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。那么可能会尝试使用模式'`"'".-["']`'，它表示一个引号后面跟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"这样的字符串时会有问题。要解决这个问题，可以捕获第一个引号然后用它来指明第二个引号：

```
s = [[then he said: "it's all right!"]]
q, quotedPart = string.match(s, "(['']).-[%1]")
print(quotedPart) --> it's all right
print(q) --> "
```

第 1 个捕获是引号本身，第 2 个捕获是引号中的内容（与'`.-`' 匹配的子串）。

下例是一个类似的示例，用于匹配 Lua 语言中的长字符串的模式：

```
%[=(*)%[(.-)%]%)%1%
```

它所匹配的内容依次是：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 1 个捕获是等号序列（在本例中只有一个），第 2 个捕获是字符串内容^①。

^①译者注：`%n` 表示匹配第 `n` 个捕获的副本，在本例中的第一个捕获就是若干个等号组成的序列，因此`%1` 相当于匹配这若干个等号。



被捕获对象的第 3 个用途是在函数 `gsub` 的替代字符串中。像模式一样，替代字符串同样可以包括像 "%n" 一样的字符分类，当发生替换时会被替换为相应的捕获。特别地，"%0" 意味着整个匹配，并且替换字符串中的百分号必须被转义为 "%%。下面这个示例会重复字符串中的每个字母，并且在每个被重复的字母之间插入一个减号：

```
print((string.gsub("hello Lua!", "%a", "%0-%0")))
--> h-he-el-l-l-lo-o L-Lu-ua-a!
```

下例交换了相邻的字符：

```
print((string.gsub("hello Lua", "(.)(.)", "%2%1")))
--> ehlL ouLa
```

以下是一个更有用的示例，让我们编写一个原始的格式转换器，该格式转换器能读取 `LATEX` 风格的命令，并将它们转换成 `XML` 风格：

```
\command{some text} --> <command>some text</command>
```

如果不允许嵌套的命令，那么以下调用函数 `string.gsub` 的代码即可完成这项工作：

```
s = [[the \quote{task} is to \em{change} that.]]
s = string.gsub(s, "\\\(%a+){(.)}", "<%1>%2</%1>")
print(s)
--> the <quote>task</quote> is to <em>change</em> that.
```

(在下一节中，我们将会学习如何处理嵌套的命令。)

另一个有用的示例是剔除字符串两端空格：

```
function trim (s)
  s = string.gsub(s, "^\s*(.-)\s*$", "%1")
  return s
end
```

请注意模式中修饰符的合理运用。两个定位标记 (^ 和 \$) 保证了我们可以获取到整个字符串。由于中间的'..-' 只会匹配尽可能少的内容，所以两个 '%s*' 可以匹配到首尾两端的空格。

10.4 替换

正如我们此前已经看到的，函数 `string.gsub` 的第 3 个参数不仅可以是字符串，还可以是一个函数或表。当第 3 个参数是一个函数时，函数 `string.gsub` 会在每次找到匹配时调用



该函数，参数是捕获到的内容而返回值则被作为替换字符串。当第 3 个参数是一个表时，函数 `string.gsub` 会把第一个捕获到的内容作为键，然后将表中对该键的值作为替换字符串。如果函数的返回值为 `nil` 或表中不包含这个键或表中键的对应值为 `nil`，那么函数 `gsub` 不改变这个匹配。

先举一个例子，下述函数用于变量展开（variable expansion），它会把字符串中所有出现的 `$varname` 替换为全局变量 `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?
```

（`_G` 是预先定义的包括所有全局变量的表，我们会在第 22 章中讨论相关细节。）对于每个与 `'$(%w+)'` 匹配的地方（`$` 符号后紧跟一个名字），函数 `gsub` 都会在全局表 `_G` 中查找捕获到的名字，并用找到的结果替换字符串中相匹配的部分；如果表中没有对应的键，则不进行替换：

```
print(expand("$othername is $status, isn't it?"))
--> $othername is great, isn't it?
```

如果不确定是否指定变量具有字符串值，那么可以对它们的值调用函数 `tostring`。在这种情况下，可以用一个函数来返回要替换的值：

```
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
```

在函数 `expand` 中，对于所有匹配 `'$(%w+)'` 的地方，函数 `gsub` 都会调用给定的函数，传入捕获到的名字作为参数，并使用返回字符串替换匹配到的内容。



最后一个例子，让我们再回到上一节中提到的格式转换器。我们仍然是想将 `LATEX` 风格的命令 (`\example{text}`) 转换成 XML 风格的 (`<example>text</example>`)，但这次允许嵌套的命令。以下的函数用递归的方式完成了这个需求：

```
function toxml (s)
    s = string.gsub(s, "\\(%a+)(%b{})", function (tag, body)
        body = string.sub(body, 2, -2) -- 移除括号
        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>
```

10.4.1 URL 编码

我们的下一个示例中将用到 *URL* 编码，也就是 HTTP 所使用的在 URL 中传递参数的编码方式。这种编码方式会将特殊字符（例如 `=`、`&` 和 `+`）编码为 "`%xx`" 的形式，其中 `xx` 是对应字符的十六进制值。此外，URL 编码还会将空格转换为加号。例如，字符串 "`a+b =c`" 的 URL 编码为 "`a%2Bb+%3D+c`"。最后，URL 编码会将每对参数名及其值用等号连接起来，然后将每对 `name=value` 用 `&` 连接起来。例如，值

```
name = "al"; query = "a+b = c"; q="yes or no"
```

对应的 URL 编码为 "`name=al&query=a%2Bb+%3D+c&q=yes+or+no`"。

现在，假设要将这个 URL 解码并将其中的键值对保存到一个表内，以相应的键作为索引，那么可以使用以下的函数完成基本的解码：

```
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
```



```

end
print(unescape("a%2Bb+%3D+c")) --> a+b = c

```

第一个 `gsub` 函数将字符串中的所有加号替换为空格，第二个 `gsub` 函数则匹配所有以百分号开头的两位十六进制数，并对每处匹配调用一个匿名函数。这个匿名函数会将十六进制数转换成一个数字（以 16 为进制，使用函数 `tonumber`）并返回其对应的字符（使用函数 `string.char`）。

可以使用函数 `gmatch` 来对键值对 `name=value` 进行解码。由于键名和值都不能包含 & 或 =，所以可以使用模式'[^&=]++' 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

```

调用函数 `gmatch` 会匹配所有格式为 `name=value` 的键值对。对于每组键值对，迭代器会返回对应的捕获（在匹配的字符串中被括号括起来了），捕获到的内容也就是 `name` 和 `value` 的值。循环体内只是简单地对两个字符串调用函数 `unescape`，然后将结果保存到表 `cgi` 中。

对应的编码函数也很容易编写。先写一个 `escape` 函数，用它将所有的特殊字符编码为百分号紧跟对应的十六进制形式（函数 `format` 的参数 "%02X" 用于格式化输出一个两位的十六进制数，若不足两位则以 0 补齐），然后把空格替换成加号：

```

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表，然后构造出最终的字符串：

```

function encode (t)
    local b = {}
    for k,v in pairs(t) do
        b[#b + 1] = (escape(k) .. "=" .. escape(v))
    end
    -- 将'b'中所有的元素连接在一起，使用"&"分隔
    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

```

10.4.2 制表符展开

在 Lua 语言中，像'()'这样的空白捕获（empty capture）具有特殊含义。该模式并不代表捕获空内容（这样的话毫无意义），而是捕获模式在目标字符串中的位置（该位置是数值）：

```
print(string.match("hello", "()ll()")) --> 3 5
```

（请注意，由于第 2 个空捕获的位置是在匹配之后，所以这个示例的结果与调用函数 `string.find` 得到的结果并不一样。）

另一个关于位置捕获（position capture）的良好示例是在字符串中进行制表符展开：

```

function expandTabs (s, tab)
    tab = tab or 8          -- 制表符的"大小"（默认是8）
    local corr = 0           -- 修正量
    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

```

函数 `gsub` 会匹配字符串中所有的制表符并捕获它们的位置。对于每个制表符，匿名函数会根据其所在位置计算出需要多少个空格才能恰好凑够一列（整数个 `tab`）：该函数先将

位置减去 1 以从 0 开始计数，然后加上 corr 凑整之前的制表符（每一个被展开的制表符都会影响后续制表符的位置）。之后，该函数更新下一个制表符的修正量：为正在被去掉的制表符减 1，再加上要增加的空格数 sp。最后，这个函数返回由替代制表符的合适数量的空格组成的字符串。

为了完整起见，让我们再看一下如何实现逆向操作，即将空格转换为制表符。第一种方法是通过空捕获来对位置进行操作，但还有一种更简单的方法：即在字符串中每隔 8 个字符插入一个标记，然后将前面有空格的标记替换为制表符。

```
function unexpandTabs (s, tab)
    tab = tab or 8
    s = expandTabs(s, tab)
    local pat = string.rep(".", tab)          -- 辅助模式
    s = string.gsub(s, pat, "%0\1")          -- 在每8个字符后添加一个标记\1
    s = string.gsub(s, " +\1", "\t")          -- 将所有以此标记结尾的空格序列
                                                -- 都替换为制表符\t
    s = string.gsub(s, "\1", "")              -- 将剩下的标记\1删除
    return s
end
```

这个函数首先对字符串进行了制表符展开以移除其中所有的制表符，然后构造出一个用于匹配所有 8 个字符序列的辅助模式，再利用这个模式在每 8 个字符后添加一个标记（控制字符\1）。接着，它将所有以此标记结尾的空格序列都替换为制表符。最后，将剩下的标记删除（即那些没有位于空格后的标记）。

10.5 诀窍

模式匹配是进行字符串处理的强大工具之一。虽然通过多次调用函数 `string.gsub` 就可以完成许多复杂的操作，但是还是应该谨慎地使用该函数。

模式匹配替代不了传统的解析器。对于那些用后即弃的程序来说，我们确实可以在源代码中做一些有用的操作，但却很难构建出高质量的产品。例如，考虑一下之前曾经用来匹配 C 语言程序中注释的模式'/*.*-%*/'。如果 C 代码中有一个字符串常量含有"/*"，那么就会得到错误的结果：

```
test = [[char s[] = "a /* here"; /* a tricky string */]]
```