

```
print((string.gsub(test, "/%*.-%*/", "<COMMENT>")))
--> char s[] = "a <COMMENT>
```

由于含有注释标记的字符串十分少见，因此对于我们自用的程序而言，这个模式可能能够满足需求；但是，我们不应该将这个带有缺陷的程序发布出去。

通常，在 Lua 程序中使用模式匹配时的效率是足够高的：笔者的新机器可以在不到 0.2 秒的时间内计算出一个 4.4MB 大小（具有 85 万个单词）的文本中所有单词的数量。^①但仍需要注意，应该永远使用尽可能精确的模式，不精确的模式会比精确的模式慢很多。一个极端的例子是模式'(.-%\$)'，它用于获取字符串中第一个 \$ 符号前的所有内容。如果目标字符串中有 \$ 符号，那么这个模式工作很正常；但是，如果字符串中没有 \$ 符号，那么模式匹配算法就会首先从字符串起始位置开始匹配，直至为了搜索 \$ 符号而遍历整个字符串。当到达字符串结尾时，这次从字符串起始位置开始的模式匹配就失败了。之后，模式匹配算法又从字符串的第二个位置开始第二次搜索，结果仍然是无法匹配这个模式。这个匹配过程会在字符串的每个位置上进行一次，从而导致 $O(n^2)$ 的时间复杂度。在笔者的新机器上，搜索 20 万个字符需要耗费超过 4 分钟的时间。要解决这个问题，我们只需使用'^(.-%\$)' 将模式锚定在字符串的开始位置即可。这样，如果不能从起始位置开始找到匹配，搜索就会停止。有了 ^ 的锚定以后，该模式匹配就只需要不到 0.01 秒的时间了。

此外，还要留心空模式，也就是那些匹配空字符串的模式。例如，如果试图使用模式'%a*' 来匹配名字，那么就会发现到处都是名字：

```
i, j = string.find(";%*#$hello13", "%a*")
print(i,j) --> 1 0
```

在这个示例中，函数 `string.find` 在字符串的开始位置正确地找到一个空的字母序列。

在模式的结束处使用修饰符 - 是没有意义的，因为这样只会匹配到空字符串。该修饰符总是需要在其后跟上其他的东西来限制扩展的范围。同样，含有'.*' 的模式也非常容易出错，这主要是因为这种模式可能会匹配到超出我们预期范围的内容。

有时，用 Lua 语言来构造一个模式也很有用。我们已经在将空格转换为制表符的程序中使用过这个技巧。接下来再看另外一个示例，考虑如何找出一个文本中较长的行（比如超过 70 个字符的行）。较长的行就是一个具有 70 个或更多字符的序列，其中每个字符都不为换行符，因而可以用字符分类'[\n]' 来匹配除换行符以外的其他单个字符。这样，就能够通

^① “笔者的新机器”是内存为 8GB，CPU 主频为 3.6GHz 的 Intel Core i7-4790。本书中所有的性能测试数据都是从这台机器上得来的。

过把这个匹配单个字符的模式重复 70 次来匹配较长的行。除了手写以外，还可以使用函数 `string.rep` 来创建这个模式：

```
pattern = string.rep("[^\n]", 70) .. "+"
```

再举一个例子，假设要进行大小写无关的查找。一种方法就是将模式中的所有字母 `x` 用'`[xX]`'替换，即同时包含原字母大小写形式的字符分类。我们可以使用如下函数来自动地完成这种转换：

```
function nocase (s)
    s = string.gsub(s, "%a", function (c)
        return "[" .. string.lower(c) .. string.upper(c) .. "]"
    end)
    return s
end

print(nocase("Hi there!")) --> [hH][iI] [tT][hH][eE][rR][eE]!
```

有时，我们可能需要将所有出现的 `s1` 替换为 `s2`，而不管其中是否包含魔法字符。如果字符串 `s1` 和 `s2` 是常量，那么可以在编写字符串时对魔法字符进行合理的转义；但如果字符串是一个变量，那么就需要用另一个 `gsub` 函数来进行转义：

```
s1 = string.gsub(s1, "(%W)", "%%%1")
s2 = string.gsub(s2, "%%", "%%%%")
```

在进行字符串搜索时，我们对所有字母和数字外的字符进行了转义（即大写的 `W`）。而在替换字符串中，我们只对百分号进行了转义。

模式匹配的另一个有用的技巧就是，在进行实际工作前先对目标字符串进行预处理。假设想把一个字符串中所有被双引号（"）引起的内容改为大写，但又允许内容中包含转义的引号（\"）：

```
follows a typical string: "This is \"great\"!".
```

处理这种情况的方法之一就是先对文本进行预处理，将所有可能导致歧义的内容编码成别的内容。例如，可以将"`\"`" 编码为"`\1`"。不过，如果原文中本身就含有"`\1`"，那么就会遇到问题。另一种可以避免这个问题的简单做法是将所有"`\x`" 编码为"`\ddd`"，其中 `ddd` 为字符 `x` 的十六进制表示形式：

```

function code (s)
    return (string.gsub(s, "\\\(.)", function (x)
        return string.format("\%03d", string.byte(x))
    end))
end

```

这样，由于原字符串中所有的"\ddd"都进行了编码，所以编码后字符串中的"\ddd"序列一定都是编码造成的。这样，解码也就很简单了：

```

function decode (s)
    return (string.gsub(s, "\\\(%d%d%d)", function (d)
        return "\\" .. string.char(tonumber(d))
    end))
end

```

现在我们就可以完成把一个字符串中被双引号（"）引起的内容改为大写的需求。由于编码后的字符串中不包含任何转义的引号（""），所以就可以直接使用'".-'来查找位于一对引号中的内容：

```

s = [[ follows a typical string: "This is \"great\"!". ]]
s = code(s)
s = string.gsub(s, '".-', string.upper)
s = decode(s)
print(s) --> follows a typical string: "THIS IS \"GREAT\"!".

```

或者写成：

```
print(decode(string.gsub(code(s), '".-', string.upper)))
```

是否能够将模式匹配函数用于 UTF-8 字符串取决于模式本身。由于 UTF-8 的主要特性之一就是任意字符的编码不会出现在别的字符的编码中，因此文本类的模式一般可以正常工作。字符分类（character class）和字符集（character set）只对 ASCII 字符有效。例如，可以对 UTF-8 字符串使用模式'%s'，但它只能匹配 ASCII 空格，而不能匹配诸如 HTML 空格（即 &nbs;, NBSP, Non-Break Space, U+00A0）或蒙古文元音分隔符（mongolian vowel separator, U+180E）等其他的 Unicode 空格。

恰当的模式能够为处理 Unicode 带来额外的能力。一个优秀的例子是预定义模式 utf8.charpattern，该模式只精确地匹配一个 UTF-8 字符。utf8 标准库中就是按照下面的方法定义这个模式的：

```
utf8.charpattern = [\0-\x7F\xC2-\xF4][\x80-\xBF]*
```

该模式的第 1 部分匹配 ASCII 字符（范围 [0, 0x7F]）或多字节序列的起始字节（范围 [0xC2, 0xF4]），第 2 部分则匹配零个或多个后续的字节（范围 [0x80, 0xBF]）。

10.6 练习

练习 10.1：请编写一个函数 `split`，该函数接收两个参数，第 1 个参数是字符串，第 2 个参数是分隔符模式，函数的返回值是分隔符分割后的原始字符串中每一部分的序列：

```
t = split("a whole new world", " ")
-- t = {"a", "whole", "new", "world"}
```

你编写的函数是如何处理空字符串的呢？特别是，一个空字符串究竟是空序列（an empty sequence），还是一个具有空字符串的序列（a sequence with one empty string）呢？

练习 10.2：模式'`%D`' 和'`[^%d]`' 是等价的，那么模式'`[^%d%u]`' 和'`[%D%D]`' 呢？

练习 10.3：请编写一个函数 `transliterate`，该函数接收两个参数，第 1 个参数是字符串，第 2 个参数是一个表。函数 `transliterate` 根据第 2 个参数中的表使用一个字符替换字符串中的字符。如果表中将 `a` 映射为 `b`，那么该函数则将所有 `a` 替换为 `b`。如果表中将 `a` 映射为 `false`，那么该函数则把结果中的所有 `a` 移除。

练习 10.4：在 10.3 节的最后，我们定义了一个 `trim` 函数。由于该函数使用了回溯，所以对于某些字符串来说该函数的时间复杂度是 $O(n^2)$ 。例如，在笔者的新机器上，针对一个 100KB 大小字符串的匹配可能会耗费 52 秒。

- 构造一个可能会导致函数 `trim` 耗费 $O(n^2)$ 时间复杂度的字符串。
- 重写这个函数使得其时间复杂度为 $O(n)$ 。

练习 10.5：请使用转义序列`\x` 编写一个函数，将一个二进制字符串格式化为 Lua 语言中的字符串常量：

```
print(escape("\0\1hello\200"))
--> \x00\x01\x68\x65\x6C\x6C\x6F\xC8
```

作为优化，请同时使用转义序列`\z` 打破较长的行。

练习 10.6：请为 UTF-8 字符重写函数 `transliterate`。

练习 10.7：请编写一个函数，该函数用于逆转一个 UTF-8 字符串。

11

小插曲：出现频率最高的单词

在本章中，我们要开发一个读取并输出一段文本中出现频率最高的单词的程序。像之前的小插曲一样，本章的程序也十分简单，但是也使用了诸如迭代器和匿名函数这样的高级特性。

该程序的主要数据结构是一个记录文本中出现的每一个单词及其出现次数之间关系的表。使用这个数据结构，该程序可以完成 3 个主要任务。

- 读取文本并计算每一个单词的出现次数。
- 按照出现次数的降序对单词列表进行排序。
- 输出有序列表中的前 n 个元素。

要读取文本，可以遍历每一行，然后遍历每一行的每一个单词。对于我们读取的每一个单词，增加对应计数器的值：

```
local counter = {}

for line in io.lines() do
    for word in string.gmatch(line, "%w+") do
        counter[word] = (counter[word] or 0) + 1
    end
end
```

这里，我们使用模式 '`%w+`' 来描述“单词”，也就是一个或多个字母或数字。

下一步就是对单词列表进行排序。不过，就像一些有心的读者可能已经注意到的那样，我们并没有可以用来排序的单词列表。尽管如此，使用表 counter 中作为键的单词来创建一个列表还是很简单的：

```
local words = {}      -- 文本中所有单词的列表

for w in pairs(counter) do
    words[#words + 1] = w
end
```

一旦有了单词列表，就可以使用函数 `table.sort` 对其进行排序：

```
table.sort(words, function (w1, w2)
    return counter[w1] > counter[w2] or
           counter[w1] == counter[w2] and w1 < w2
end)
```

请记住，排序函数必须在 `w1` 位于 `w2` 之前时返回真。计数值越大的单词排得越前，具有相同计数值的单词则按照字母顺序排序。

示例 11.1 中展示了完整的代码。

示例 11.1 统计单词出现频率的程序

```
local counter = {}

for line in io.lines() do
    for word in string.gmatch(line, "%w+") do
        counter[word] = (counter[word] or 0) + 1
    end
end

local words = {}      -- 文本中所有单词的列表

for w in pairs(counter) do
    words[#words + 1] = w
end

table.sort(words, function (w1, w2)
```

```

        return counter[w1] > counter[w2] or
               counter[w1] == counter[w2] and w1 < w2
    end)

-- 要输出的字数
local n = math.min tonumber(arg[1]) or math.huge, #words

for i = 1, n do
    io.write(words[i], "\t", counter[words[i]], "\n")
end

```

最后一个循环输出了结果，也就是前 n 个单词及它们对应的计数值。这个程序假定第 1 个参数是要输出单词的个数；默认情况下，如果没有参数，它会输出所有的单词。

作为示例，我们给出了上述程序针对本书内容^①的运行结果：

```

$ lua wordcount.lua 10 < book.of
the    5996
a      3942
to     2560
is     1907
of     1898
in     1674
we     1496
function 1478
and    1424
x      1266

```

11.1 练习

练习 11.1：当我们对一段文本执行统计单词出现频率的程序时，结果常常是一些诸如冠词和介词之类的没有太多意义的短词汇。请改写该程序，使它忽略长度小于 4 个字母的单词。

^①译者注：英文原版书。

练习 11.2: 重复上面的练习,除了按照长度标准忽略单词外,该程序还能从一个文本文件中读取要忽略的单词列表。

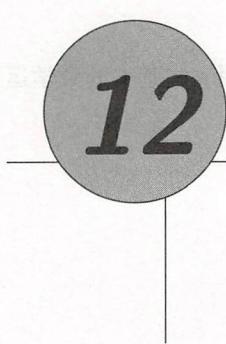
```

-- 读取文件并忽略其中的指定单词
function ignoreWords(filename, ignore)
    local file = io.open(filename, "r")
    local words = {}
    for line in file:lines() do
        for word in line:gmatch("%w+") do
            if not ignore[word] then
                words[#words + 1] = word
            end
        end
    end
    file:close()
    return words
end

```

习题 11.1

读取名为 `ignore.txt` 的文件,该文件列出要忽略的单词。将本文档一词计算出总词数,忽略列表中的所有单词。



日期和时间

Lua 语言的标准库提供了两个用于操作日期和时间的函数，这两个函数在 C 语言标准库中也存在，提供的是同样的功能。虽然这两个函数看上去很简单，但依旧可以基于这些简单的功能完成很多复杂的工作。

Lua 语言针对日期和时间使用两种表示方式。第 1 种表示方式是一个数字，这个数字通常是一个整型数。尽管并非是 ISO C 所必需的，但在大多数系统中这个数字是自一个被称为纪元 (*epoch*) 的固定日期后至今的秒数。特别地，在 POSIX 和 Windows 系统中这个固定日期均是 Jan 01, 1970, 0:00 UTC。

Lua 语言针对日期和时间提供的第 2 种表示方式是一个表。日期表 (*date table*) 具有以下几个重要的字段：year、month、day、hour、min、sec、wday、yday 和 isdst，除 isdst 以外的所有字段均为整型数。前 6 个字段的含义非常明显，而 wday 字段表示本周中的第几天（第 1 天为星期天）；yday 字段表示当年中的第几天（第 1 天是 1 月 1 日）；isdst 字段表示布尔类型，如果使用夏时令则为真。例如，Sep 16, 1998, 23:48:10（星期三）对应的表是：

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
hour = 23, min = 48, sec = 10, isdst = false}
```

日期表中不包括时区，程序需要负责结合相应的时区对其进行正确解析。

12.1 函数 os.time

不带任何参数调用函数 `os.time`，会以数字形式返回当前的日期和时间：

```
> os.time()          --> 1439653520
```

对应的时间是 Aug 15, 2015, 12:45:20。^①在一个 POSIX 系统中，可以使用一些基本的数学运算分离这个数值：

```
local date = 1439653520
local day2year = 365.242           -- 1年的天数
local sec2hour = 60 * 60          -- 1小时的秒数
local sec2day = sec2hour * 24      -- 1天的秒数
local sec2year = sec2day * day2year -- 1年的秒数

-- 年
print(date // sec2year + 1970)    --> 2015.0

-- 小时 (UTC格式)
print(date % sec2day // sec2hour)   --> 15

-- 分钟
print(date % sec2hour // 60)        --> 45

-- 秒
print(date % 60)                  --> 20
```

如果以一个日期表作为参数调用函数 `os.time`，那么该函数会返回该表中所描述日期和时间对应的数字。`year`、`month` 和 `day` 字段是必需的，`hour`、`min` 和 `sec` 字段如果没有提供的话则默认为 12:00:00，其余字段（包括 `wday` 和 `yday`）则会被忽略。

```
> os.time({year=2015, month=8, day=15, hour=12, min=45, sec=20})
--> 1439653520
> os.time({year=1970, month=1, day=1, hour=0})      --> 10800
> os.time({year=1970, month=1, day=1, hour=0, sec=1})
```

^①除非特别注明，本书中的日期来自一个运行在巴西里约热内卢的 POSIX 系统。

```
--> 10800
> os.time({year=1970, month=1, day=1}) --> 54000
```

请注意，10800 是 3 个小时的秒数，54000 则是 10800 加上 12 个小时的秒数。

12.2 函数 os.date

函数 `os.date` 在一定程度上是函数 `os.time` 的反函数（尽管这个函数的名字写的是 `date`），它可以将一个表示日期和时间的数字转换为某些高级的表示形式，要么是日期要么是字符串。该函数的第一个参数是描述期望表示形式的格式化字符串 (*format string*)，第二个参数是数字形式的日期和时间（如果不提供，则默认为当前日期和时间）。

要生成一个日期表，可以使用格式化字符串 "`*t`"。例如，调用函数 `os.date("*t", 906000490)` 会返回下列表：

```
{year = 1998, month = 9, day = 16, yday = 259, wday = 4,
hour = 23, min = 48, sec = 10, isdst = false}
```

大致上，对于任何有效的时间 `t`，`os.time(os.date("*t", t)) == t` 均成立。

除了 `isdst`，结果中的其余字段均为整型数且范围分别是：

year	一整年
month	1~12
day	1~31
hour	0~23
min	0~59
sec	0~60
wday	1~7
yday	1~366

（秒的最大范围是 60，允许闰秒（leap second）的存在。）

对于其他格式化字符串，函数 `os.date` 会将日期格式化为一个字符串，该字符串是根据指定的时间和日期信息对特定的指示符进行了替换的结果。所有的指示符都以百分号开头紧跟一个字母，例如：

```
print(os.date("a %A in %B"))           --> a Tuesday in May
print(os.date("%d/%m/%Y", 906000490)) --> 16/09/1998
```

所有的表现形式取决于当前的区域设置。例如，当前区域被设为巴西-葡萄牙语时，%A 会是“terça-feira”，而%B 则会是“maio”。

表 12.1 列出了主要的指示符，这些指示符使用的时间为 1998 年 9 月 16 日（星期三）23:48:10。

表 12.1 函数 os.date 的指示符

%a	星期几的简写 (例如, Wed)
%A	星期几的全名 (例如, Wednesday)
%b	月份的简写 (例如, Sep)
%B	月份的全名 (例如, September)
%c	日期和时间 (例如, 09/16/98 23:48:10)
%d	一个月中的第几天 (16) [01-31]
%H	24 小时制中的小时数 (23) [00-23]
%I	12 小时制中的小时数 (11) [01-12]
%j	一年中的第几天 (259) [001-365]
%m	月份 (09) [01-12]
%M	分钟 (48) [00-59]
%p	"am" 或 "pm" (pm)
%S	秒数 (10) [00-60]
%w	星期 (3) [0-6 = Sunday-Saturday]
%W	一年中的第几周 (37) [00-53]
%x	日期 (例如, 09/16/98)
%X	时间 (例如, 23:48:10)
%y	两位数的年份 (98) [00-99]
%Y	完整的年份 (1998)
%z	时区 (例如, -0300)
%%	百分号

对于数值，表 12.1 中也给出了它们的有效范围。以下是一些演示如何创建 ISO 8601 格式日期和时间的示例：

```
t = 906000490
-- ISO 8601格式的日期
print(os.date("%Y-%m-%d", t))          --> 1998-09-16
-- ISO 8601格式的日期和时间
print(os.date("%Y-%m-%dT%H:%M:%S", t)) --> 1998-09-16T23:48:10
-- ISO 8601格式的序数日期①
print(os.date("%Y-%j", t))              --> 1998-259
```

如果格式化字符串以感叹号开头，那么函数 `os.date` 会以 UTC 格式对其进行解析：

```
-- 纪元
print(os.date("!%c", 0))      --> Thu Jan 1 00:00:00 1970
```

如果不带任何参数调用函数 `os.date`，那么该函数会使用格式 `%c`，即以一种合理的格式表示日期和时间信息。请注意，`%x`、`%X` 和 `%c` 会根据不同的区域和系统而发生变化。如果需要诸如 `dd/mm/yyyy` 这样的固定表示形式，那么就必须显式地使用诸如 `"%d/%m/%Y"` 这样的格式化字符串。

12.3 日期和时间处理

当函数 `os.date` 创建日期表时，该表的所有字段均在有效的范围内。当我们给函数 `os.time` 传入一个日期表时，其中的字段并不需要归一化。这个特性对于日期和时间处理非常重要。

举一个简单的例子，假设想知道从当前向后数 40 天的日期，那么可以使用如下的代码进行计算：

```
t = os.date("*t")      -- 获取当前时间
print(os.date("%Y/%m/%d", os.time(t))) --> 2015/08/18
t.day = t.day + 40
print(os.date("%Y/%m/%d", os.time(t))) --> 2015/09/27
```

如果我们把数字表示的时间转换成日期表，那么就能得到日期和时间的归一化形式：

^①译者注：序数日期 [Ordinal Date] 指年外加 1~365 范围内的天数。

```
t = os.date("*t")
print(t.day, t.month)          --> 26    2
t.day = t.day - 40
print(t.day, t.month)          --> -14   2
t = os.date("*t", os.time(t))
print(t.day, t.month)          --> 17    1
```

在这个例子中, Feb -14 被归一化为 Jan 17, 也就是 Feb 26 的前 40 天。

在大多数系统中, 也可以对数字形式的时间增加或减少 3456000 (40 天对应的秒数)。不过, 由于标准 C 并不要求数值表示的时间是从纪元开始的, 因此标准 C 并不保证这种操作的正确性。此外, 如果我们想增加的是月份而非天数, 由于不同的月份具有不同的天数, 那么直接操作秒数就会有问题。而以归一化的方式处理则没有这些问题:

```
t = os.date("*t")      -- 获取当前时间
print(os.date("%Y/%m/%d", os.time(t)))  --> 2015/08/18
t.month = t.month + 6  -- 从当天开始往后6个月
print(os.date("%Y/%m/%d", os.time(t)))  --> 2016/02/18
```

在操作日期时, 我们必须要小心。虽然归一化是以显而易见的方式进行的, 但是也可能有一些不明显的后果。例如, 如果计算 March 31 之后的一个月, 将会得到 April 31, 而实际上应该被归一化成 May 1 (April 30 之后的一天)。尽管这听上去很自然, 但实际上如果从结果 (May 1) 中减去一个月, 得到的却是 April 1 而不是原来的 March 31。请注意, 这种不一致是日历机制导致的结果, 与 Lua 语言无关。

函数 `os.difftime` 用来计算两个时间之间的差值, 该函数以秒为单位返回两个指定数字形式表示的时间的差值。对于大多数系统而言, 这个差值就是一个时间相对于另一个时间的减法结果。但是, 与减法不同, 函数 `os.difftime` 的行为在任何系统中都是确定的。以下的示例计算了 Lua 5.2 和 Lua 5.3 发布时间之间间隔的天数:

```
local t5_3 = os.time({year=2015, month=1, day=12})
local t5_2 = os.time({year=2011, month=12, day=16})
local d = os.difftime(t5_3, t5_2)
print(d // (24 * 3600))      --> 1123.0
```

使用函数 `difftime` 可以获取指定日期相对任意时刻的秒数:

```
> myepoch = os.time{year = 2000, month = 1, day = 1, hour = 0}
> now = os.time{year = 2015, month = 11, day = 20}
> os.difftime(now, myepoch)    --> 501336000.0
```

通过归一化，可以很容易地将用秒表示的时间转换为合法的数字形式表示的时间，即我们可以创建一个带有开始时刻的日期表并将日期表中的秒数设置为想要转换的数字。例如：

```
> T = {year = 2000, month = 1, day = 1, hour = 0}
> T.sec = 501336000
> os.date("%d/%m/%Y", os.time(T)) --> 20/11/2015
```

我们还可以使用函数 `os.difftime` 来计算一段代码的执行时间。不过，对于这个需求，更好的方式是使用函数 `os.clock`，该函数会返回程序消耗的 CPU 时间（单位是秒）。函数 `os.clock` 在性能测试（benchmark）中的典型用法形如：

```
local x = os.clock()
local s = 0
for i = 1, 100000 do s = s + i end
print(string.format("elapsed time: %.2f\n", os.clock() - x))
```

与函数 `os.time` 不同，函数 `os.clock` 通常具有比秒更高的精度，因此其返回值为一个浮点数。具体的精度与平台相关，在 POSIX 系统中通常是 1 毫秒。

12.4 练习

练习 12.1：请编写一个函数，该函数返回指定日期和时间后恰好一个月的日期和时间（假设日期和时间使用数字形式表示）。

练习 12.2：请编写一个函数，该函数返回指定日期是星期几（用整数表示，1 表示星期天）。

练习 12.3：请编写一个函数，该函数的参数为一个日期和时间（使用数值表示），返回当天中已经经过的秒数。

练习 12.4：请编写一个函数，该函数的参数为年，返回该年中第一个星期五是第几天。

练习 12.5：请编写一个函数，该函数用于计算两个指定日期之间相差的天数。

练习 12.6：请编写一个函数，该函数用于计算两个指定日期之间相差的月份。

练习 12.7：向指定日期增加一个月再增加一天得到的结果，是否与先增加一天再增加一个月得到的结果相同？

练习 12.8：请编写一个函数，该函数用于输出操作系统的时区。

13

位和字节

Lua 语言处理二进制数据的方式与处理文本的方式类似。Lua 语言中的字符串可以包含任意字节，并且几乎所有能够处理字符串的库函数也能处理任意字节。我们甚至可以对二进制数据进行模式匹配。以此为基础，Lua 5.3 中引入了用于操作二进制数据的额外机制：除了整型数外，该版本还引入了位操作及用于打包/解包二进制数据的函数。在本章中，我们会学习上述内容，以及 Lua 语言用于处理二进制数据的其他工具。

13.1 位运算

Lua 语言从 5.3 版本开始提供了针对数值类型的一组标准位运算符（bitwise operator）。与算术运算符不同的是，位运算符只能用于整型数。位运算符包括 &（按位与）、|（按位或）、~（按位异或）、>>（逻辑右移）、<<（逻辑左移）和一元运算符 ~（按位取反）。（请注意，在其他一些语言中，异或运算符为 ^，而在 Lua 语言中 ^ 代表幂运算。）

```
> string.format("%x", 0xff & 0xabcd)      --> cd
> string.format("%x", 0xff | 0xabcd)      --> abff
> string.format("%x", 0xaaaa ~ -1)        --> ffffffff5555
> string.format("%x", ~0)                  --> ffffffff0000
```

（本章中的几个例子会使用函数 `string.format` 来输出十六进制形式的结果。）

所有的位运算都针对构成一个整型数的所有位。在标准 Lua 中，也就是 64 位。这对于使用 32 位整型数的算法可能会成为问题（例如，SHA-2 密码散列算法）。不过，要操作 32 位整型数也不难。除了右移操作外，只要忽略高 32 位，那么所有针对 64 位整型数的操作与针对 32 位整型数的操作都一样。这对于加法、减法和乘法都有效。因此，在操作 32 位整型数时，只需要在进行右移前抹去高 32 位即可（对于这类计算很少会做除法）。

两个移位操作都会用 0 填充空出的位，这种行为通常被称为逻辑移位（logical shift）。Lua 语言没有提供算术右移（arithmetic right shift），即使用符号位填充空出的位。我们可以通过向下取整除法（floor 除法），除以合适的 2 的整数次幂来实现算术右移（例如， $x//16$ 与算术右移 4 位等价）。

移位数是负数表示向相反的方向移位，即 $a>>n$ 与 $a<<-n$ 等价：

```
> string.format("%x", 0xff << 12)      --> ff000
> string.format("%x", 0xff >> -12)     --> ff000
```

如果移位数等于或大于整型表示的位数（标准 Lua 为 64 位，精简 Lua 为 32 位），由于所有的位都被从结果中移出了，所以结果是 0：

```
> string.format("%x", -1 << 80)      --> 0
```

13.2 无符号整型数

整型表示中使用一个比特来存储符号位。因此，64 位整型数最大可以表示 $2^{63} - 1$ 而不是 $2^{64} - 1$ 。通常，这点区别是无关紧要的，因为 $2^{63} - 1$ 已经相当大了。不过，由于我们可能需要处理使用无符号整型表示的外部数据或实现一些需要 64 位整型数的算法，因而有时也不能浪费这个符号位。此外，在精简 Lua 中，这种区别可能会很重要。例如，如果用一个 32 位有符号整型数表示文件中的位置，那么能够操作的最大文件大小就是 2GB；而一个无符号整型数能操作的最大文件大小则是有符号整型数的 2 倍，即 4GB。

Lua 语言不显式支持无符号整型数。不过尽管如此，只要稍加注意，在 Lua 语言中处理无符号整型数并不难，我们后续就会看到。

虽然看上去不太友好，但可以直接写出比 $2^{63} - 1$ 大的常量：

```
> x = 13835058055282163712      -- 3 << 62
> x                                --> -4611686018427387904
```

这里的问题并不在于常量本身，而在于 Lua 语言输出常量的方式：默认情况下，打印数值时是将其作为有符号整型数进行处理的。我们可以使用选项%u 或%x 在函数 `string.format` 中指定以无符号整型数进行输出：

```
> string.format("%u", x)           --> 13835058055282163712
> string.format("0x%X", x)        --> 0xC000000000000000
```

根据有符号整型数的表示方式（2 的补码），加法、减法和乘法操作对于有符号整型数和无符号整型数是一样的：

```
> string.format("%u", x)           --> 13835058055282163712
> string.format("%u", x + 1)        --> 13835058055282163713
> string.format("%u", x - 1)        --> 13835058055282163711
```

（对于这么大的数，即便 `x` 乘以 2 也会溢出，所以示例中没有演示乘法。）

关系运算对于有符号整型数和无符号整型数是不一样的，当比较具有不同符号位的整型数时就会出现问题。对于有符号整型数而言，符号位被置位的整数更小，因为它代表的是负数：

```
> 0xfffffffffffffff < 0x8000000000000000    --> false
```

如果把两个整型数都当作无符号的，那么结果显然是不正确的。因此，我们需要使用一种不同的操作来比较无符号整型数。Lua 5.3 提供了函数 `math.ult` (*unsigned less than*) 来完成这个需求：

```
> math.ult(0xfffffffffffffff, 0x8000000000000000)  --> true
```

另一种方法是在进行有符号比较前先用掩码掩去两个操作数的符号位：

```
> mask = 0x8000000000000000
> (0xfffffffffffffff ~ mask) < (0x8000000000000000 ~ mask)
--> true
```

无符号除法和有符号除法也不一样，示例 13.1 给出了一种无符号除法的算法。

示例 13.1 无符号除法

```
function udiv (n, d)
  if d < 0 then
    if math.ult(n, d) then return 0
```

```

    else return 1
  end
end
local q = ((n >> 1) // d) << 1
local r = n - q * d
if not math.ult(r, d) then q = q + 1 end
return q
end

```

第一个比较 ($d < 0$) 等价于比较 d 是否大于 2^{63} 。如果大于，那么商只能是 1（如果 n 等于或大于 d ）或 0。否则，我们使被除数除以 2，然后除以除数，再把结果乘以 2。右移 1 位等价于除以 2 的无符号除法，其结果是一个非负有符号整型数。后续的左移则纠正了商，还原了之前的除法。

总体上说， $\text{floor}(\text{floor}(n/2)/d)*2$ （算法进行的计算）与 $\text{floor}(((n/2)/d)*2)$ （正确的结果）并不等价。不过，要证明它们之间最多相差 1 并不困难。因此，算法计算了余数（变量 r ），然后判断余数是否比除数大，如果余数比除数大则纠正商（加 1）即可。

无符号整型数和浮点型数之间的转换需要进行一些调整。要把一个无符号整型数转换为浮点型数，可以先将其转换成有符号整型数，然后通过取模运算纠正结果：

```

> u = 11529215046068469760          -- 一个示例
> f = (u + 0.0) % 2^64
> string.format("%.0f", f)           --> 11529215046068469760

```

由于标准转换把 u 当作有符号整型数，因此表达式 $u+0.0$ 的值是 -6917529027641081856，而之后的取模操作会把这个值限制在有符号整型数的表示范围内（在实际的代码中，由于涉及浮点型数的取模运算肯定会进行类型转换，所以并不需要进行这次加法运算）。

要把一个浮点型数转换为无符号整型数，可以使用如下的代码：

```

> f = 0xA000000000000000.0          -- 一个示例
> u = math.tointeger(((f + 2^63) % 2^64) - 2^63)
> string.format("%x", u)             --> a000000000000000

```

加法把一个大于 2^{63} 的数转换为一个大于 2^{64} 的数，取模运算把这个数限制到 $[0, 2^{63})$ 范围内，然后通过减法把结果变成一个“负”值（即最高位置位的值）。对于小于 2^{63} 的值，加法结果小于 2^{64} ，所以取模运算没有任何效果，之后的减法则把它恢复到了之前的值。

13.3 打包和解包二进制数据

Lua 5.3 还引入了一个在二进制数和基本类型值（数值和字符串类型）之间进行转换的函数。函数 `string.pack` 会把值“打包（pack）”为二进制字符串，而函数 `string.unpack` 则从字符串中提取这些值。

函数 `string.pack` 和函数 `string.unpack` 的第 1 个参数是格式化字符串，用于描述如何打包数据。格式化字符串中的每个字母都描述了如何打包/解包一个值，例如：

```
> s = string.pack("iii", 3, -27, 450)
> #s                                --> 12
> string.unpack("iii", s)           --> 3   -27   450   13
```

调用函数 `string.pack` 将创建一个字符串，其中为 3 个整型数的二进制代码（根据“`iii`”），每一个“`i`”编码对与之对应的参数进行了编码，而字符串的长度则是一个整型数本身大小的 3 倍（在笔者的机器上是 3×4 字节）。调用函数 `string.unpack` 对给定字符串中的 3 个整型数进行了解码（还是根据“`iii`”）并返回解码后的结果。

为了便于迭代，函数 `string.unpack` 还会返回最后一个读取的元素在字符串中的位置（这解释了上例中的 13）。相应地，该函数还有一个可选的第 3 个参数，这个参数用于指定开始读取的位置。例如，下例输出了一个指定字符串中所有被打包的字符串：

```
s = "hello\0Lua\0world\0"
local i = 1
while i <= #s do
    local res
    res, i = string.unpack("z", s, i)
    print(res)
end
--> hello
--> Lua
--> world
```

正如我们马上要看到的，选项 `z` 意味着一个以`\0`结尾的字符串。因此，调用函数 `unpack` 会从 `s` 中提取位于 `i` 的字符串，并返回该字符串外加循环迭代的下一个位置。

对于编码一个整型数而言有几种选项，每一种对应了一种整型大小：`b`（`char`）、`h`（`short`）、`i`（`int`）、`l`（`long`）和 `j`（代表 Lua 语言中整型数的大小）。要是使用固定的、与机器无关的

大小，可以在选项 i 后加上一个 1~16 的数。例如，i7 会产生 7 字节的整型数。所有的大小都会被检查是否存在溢出的情况：

```
> x = string.pack("i7", 1 << 54)
> string.unpack("i7", x)           --> 18014398509481984 8
> x = string.pack("i7", -(1 << 54))
> string.unpack("i7", x)           --> -18014398509481984 8
> x = string.pack("i7", 1 << 55)
stdin:1: bad argument #2 to 'pack' (integer overflow)
```

我们可以打包和解包比 Lua 语言原生整型数更大的整型数，但是在解包的时候它们的实际值必须能够被 Lua 语言的整型数容纳：

```
> x = string.pack("i12", 2^61)
> string.unpack("i12", x)         --> 2305843009213693952 13
> x = "aaaaaaaaaaaa"            -- 模拟一个大的12字节的数值
> string.unpack("i12", x)
stdin:1: 12-byte integer does not fit into Lua Integer
```

每一个针对整型数的选项都有一个对应的大写版本，对应相应大小的无符号整型数：

```
> s = "\xFF"
> string.unpack("b", s)          --> -1    2
> string.unpack("B", s)          --> 255   2
```

同时，无符号整型数对于 size_t 而言还有一个额外的选项 T (size_t 类型在 ISO C 中是一个足够容纳任意对象大小的无符号整型数)。

我们可以用 3 种表示形式打包字符串：\0 结尾的字符串、定长字符串和使用显式长度的字符串。 \0 结尾的字符串使用选项 z；定长字符串使用选项 cn，其中 n 是被打包字符串的字节数。显式长度的字符串在存储时会在字符串前加上该字符串的长度。在这种情况下，选项的格式形如 sn，其中 n 是用于保存字符串长度的无符号整型数的大小。例如，选项 s1 表示把字符串长度保存在一个字节中：

```
s = string.pack("s1", "hello")
for i = 1, #s do print((string.unpack("B", s, i))) end
--> 5                  (length)
--> 104                ('h')
--> 101                ('e')
```

```
--> 108 ('l')
--> 108 ('l')
--> 111 ('o')
```

如果用于保存长度的字节容纳不了字符串长度，那么 Lua 语言会抛出异常。我们也可以单纯使用选项 `s`，在这种情况下，字符串长度会被以足够容纳任何字符串长度的 `size_t` 类型保存（在 64 位机器中，`size_t` 通常是 8 字节的无符号整型数，对于较短的字符串来说可能会浪费空间）。

对于浮点型数，有 3 种选项：`f` 用于单精度浮点数、`d` 用于双精度浮点数、`n` 用于 Lua 语言浮点数。

格式字符串也有用来控制大小端模式和二进制数据对齐的选项。在默认情况下，格式使用的是机器原生的大小端模式。选项 `>` 把所有后续的编码转换改为大端模式或网络字节序 (*network byte order*)：

```
s = string.pack(">i4", 1000000)
for i = 1, #s do print((string.unpack("B", s, i))) end
--> 0
--> 15
--> 66
--> 64
```

选项 `<` 则改为小端模式：

```
s = string.pack("<i2 i2", 500, 24)
for i = 1, #s do print((string.unpack("B", s, i))) end
--> 244
--> 1
--> 24
--> 0
```

最后，选项 `=` 改回机器默认的原生大小端模式。

对于对齐而言，选项 `!n` 强制数据对齐到以 `n` 为倍数的索引上。更准确地说，如果数据比 `n` 小，那么对齐到其自身大小上；否则，对齐到 `n` 上。例如，假设格式化字符串为 `!4`，那么 1 字节整型数会被写入以 1 为倍数的索引位置上（也就是任意索引位置上），2 字节的整型数会被写入以 2 为倍数的索引位置上，而 4 字节或更大的整型数则会被写入以 4 为倍数的索引位置上，而选项 `!`（不带数字）则把对齐设为机器默认的对齐方式。

函数 `string.pack` 通过在结果字符串到达合适索引值前增加 0 的方式实现对齐，函数 `string.unpack` 在读取字符串时会简单地跳过这些补位（padding）。对齐只对 2 的整数次幂有效，如果把对齐设为 4 但试图操作 3 字节的整型数，那么 Lua 语言会抛出异常。

所有的格式化字符串默认带有前缀“`=!1`”，即表示使用默认的大小端模式且不对齐（因为每个索引都是 1 的倍数）。我们可以在程序执行过程中的任意时点改变大小端模式和对齐方式。

如果需要，可以手工添加补位。选项 `x` 代表 1 字节的补位，函数 `string.pack` 会在结果字符串中增加一个 0 字节，而函数 `string.unpack` 则从目标字符串中跳过 1 字节。

13.4 二进制文件

函数 `io.input` 和 `io.output` 总是以文本方式（*text mode*）打开文件。在 POSIX 操作系统中，二进制文件和文本文件是没有差别的。然而，在其他一些像 Windows 之类的操作系统中，必须用特殊方式来打开二进制文件，即在 `io.open` 的模式字符串中使用字母 `b`。

通常，在读取二进制数据时，要么使用模式“`a`”来读取整个文件，要么使用模式 `n` 来读取 `n` 字节（在二进制文件中，“行”是没有意义的）。下面是一个简单的示例，它会把 Windows 格式的文本文件转换为 POSIX 格式，即把 `\r\n` 转换为 `\n`：

```
local inp = assert(io.open(arg[1], "rb"))
local out = assert(io.open(arg[2], "wb"))

local data = inp:read("a")
data = string.gsub(data, "\r\n", "\n")
out:write(data)

assert(out:close())
```

由于标准 I/O 流（`stdin/stdout`）是以文本模式打开的，所以上例不能使用标准 I/O 流。相反，该程序假设输入和输出文件的名称是由程序的参数指定的。可以使用如下的命令调用该程序：

```
> lua prog.lua file.dos file.unix
```

再举一个例子，以下的程序输出了一个二进制文件中的所有字符串：

```

local f = assert(io.open(arg[1], "rb"))
local data = f:read("a")
local validchars = "[%g%s]"
local pattern = "(" .. string.rep(validchars, 6) .. "+)\0"
for w in string.gmatch(data, pattern) do
    print(w)
end

```

这个程序假定字符串是一个以\0结尾的、包含6个或6个以上有效字符的序列，其中有效字符是指能与模式validchars匹配的任意字符。在这个示例中，这个模式由可打印字符组成。我们使用函数string.rep和字符串连接创建用于捕获以\0结尾的、包含6个或6个以上有效字符validchars的模式，这个模式中的括号用于捕获不带\0的字符串。

最后一个示例用于以十六进制内容输出二进制文件的Dump，示例13.2展示了在POSIX操作系统下将这个程序用于其自身时的结果：

示例 13.2 对 dump 程序执行 Dump 操作

```

6C 6F 63 61 6C 20 66 20 3D 20 61 73 73 65 72 74 local f = assert
28 69 6F 2E 6F 70 65 6E 28 61 72 67 5B 31 5D 2C (io.open(arg[1],
20 22 72 62 22 29 29 0A 6C 6F 63 61 6C 20 62 6C "rb")).local bl
6F 63 6B 73 69 7A 65 20 3D 20 31 36 0A 66 6F 72 ocksize = 16.for
20 62 79 74 65 73 20 69 6E 20 66 3A 6C 69 6E 65 bytes in f:line
...
25 63 22 2C 20 22 2E 22 29 0A 20 20 69 6F 2E 77 %c", "."). io.w
72 69 74 65 28 22 20 22 2C 20 62 79 74 65 73 2C rite(" ", bytes,
20 22 5C 6E 22 29 0A 65 6E 64 0A 0A "\n").end..

```

完整的程序如下：

```

local f = assert(io.open(arg[1], "rb"))
local blocksize = 16
for bytes in f:lines(blocksize) do
    for i = 1, #bytes do
        local b = string.unpack("B", bytes, i)
        io.write(string.format("%02X ", b))
    end
    io.write(string.rep("   ", blocksize - #bytes))
end

```

```

bytes = string.gsub(bytes, "%c", ".")
io.write(" ", bytes, "\n")
end

```

同样，程序的第一个参数是输入文件名，结果则是被输出到标准输出中的普通文本。这个程序以 16 字节为一个块读取文件，对于每个块先输出每个字节的十六进制表示，然后将控制字符替换为点，最后把整个块作为文本输出。函数 `string.rep` 用于填充最后一行中的空白（因为最后一行往往不到 16 字节）以保持对齐。

13.5 练习

练习 13.1：请编写一个函数，该函数用于进行无符号整型数的取模运算。

练习 13.2：请实现计算 Lua 语言中整型数所占用位数的不同方法。

练习 13.3：如何判断一个指定整数是不是 2 的整数次幂？

练习 13.4：请编写一个函数，该函数用于计算指定整数的汉明权重（一个数的汉明权重 (*Hamming weight*) 是其二进制表示中 1 的个数）。

练习 13.5：请编写一个函数，该函数用于判断指定整数的二进制表示是否为回文数。

练习 13.6：请在 Lua 语言中实现一个比特数组 (*bit array*)，该数组应支持如下的操作。

- `newBitArray(n)` (创建一个具有 n 个比特的数组)。
- `setBit(a, n, v)` (将布尔值 v 赋值给数组 a 的第 n 位)。
- `testBit(a, n)` (将第 n 位的值作为布尔值返回)。

练习 13.7：假设有一个以一系列记录组成的二进制文件，其中的每一个记录的格式为：

```

struct Record {
    int x;
    char[3] code;
    float value;
};

```

请编写一个程序，该程序读取这个文件，然后输出 `value` 字段的总和。

14

数据结构

Lua 语言中的表并不是一种数据结构，它们是其他数据结构的基础。我们可以用 Lua 语言中的表来实现其他语言提供的数据结构，如数组、记录、列表^①、队列、集合等。而且，用 Lua 语言中的表实现这些数据结构还很高效。

在像 C 和 Pascal 这样更加传统的语言中，通常使用数组和列表（列表 = 记录 + 指针）来实现大多数数据结构。虽然在 Lua 语言中也可以使用表来实现数组和列表（有时我们确实也这么做），但表实际上比数组和列表强大得多。使用表时，很多算法可以被简化。例如，由于表本身就支持任意数据类型的直接访问，因此我们很少在 Lua 语言中编写搜索算法。

学习如何高效地使用表需要花费一点时间。这里，我们先来学习如何通过表来实现一些典型的数据结构并给出一些使用这些数据结构的例子。首先，我们学习数组和列表，这并不是因为需要它们作为其他结构的基础，而是因为大多数程序员已经对数组和列表比较熟悉了。（我们已经在本书第5章学习过这方面的基础，但为了完整起见本章将更详细地进行讨论。）之后，我们再继续学习更加高级的例子，比如集合、包和图。

^①译者注：如果是从数据结构的维度出发此处译为线性表可能更贴切，但实际原文中的用词全都是 list，所以译文也都翻译成了列表，请读者自己体会。

14.1 数组

在 Lua 语言中，简单地使用整数来索引表即可实现数组。因此，数组的大小不用非得是固定的，而是可以按需增长的。通常，在初始化数组时就间接地定义了数组的大小。例如，在执行了以下的代码后，任何访问范围 1~1000 之外的元素都会返回 nil 而不是 0：

```
local a = {}      -- 新数组
for i = 1, 1000 do
    a[i] = 0
end
```

长度运算符 (#) 正是基于此来计算数组大小的：

```
print(#a)          --> 1000
```

可以使用 0、1 或其他任何值来作为数组的起始索引：

```
-- 创建一个索引范围为 -5~5 的数组
a = {}
for i = -5, 5 do
    a[i] = 0
end
```

不过，在 Lua 语言中一般以 1 作为数组的起始索引，Lua 语言的标准库和长度运算符都遵循这个惯例。如果数组的索引不从 1 开始，那就不能使用这些机制。

可以通过表构造器在一句表达式中同时创建和初始化数组：

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

这种表构造器根据需求要多大就能多大。在 Lua 语言中，利用数据描述文件 (data-description file) 创建包含几百万个元素组成的构造器很常见。

14.2 矩阵及多维数组

在 Lua 语言中，有两种方式来表示矩阵。第一种方式是使用一个不规则数组 (*jagged array*)，即数组的数组，也就是一个所有元素均是另一个表的表。例如，可以使用如下的代码来创建一个全 0 元素的 NxM 维矩阵：

```

local mt = {}          -- 创建矩阵
for i = 1, N do
    local row = {}      -- 创建新的一行
    mt[i] = row
    for j = 1, M do
        row[j] = 0
    end
end

```

由于表在 Lua 语言中是一种对象，因此在创建矩阵时必须显式地创建每一行。一方面，这比在 C 语言中直接声明一个多维数组更加具体；另一方面，这也给我们提供了更多的灵活性。例如，只需将前例中的内层循环改为 `for j=1,i do ...end` 就可以创建一个三角形矩阵。使用这套代码，三角形矩阵较原来的矩阵可以节约一半的内存。

在 Lua 中表示矩阵的第二种方式是将两个索引合并为一个。典型情况下，我们通过将第一个索引乘以一个合适的常量再加上第二个索引来实现这种效果。在这种方式下，我们可以使用以下的代码来创建一个全 0 元素的 $N \times M$ 维矩阵：

```

local mt = {}          -- 创建矩阵
for i = 1, N do
    local aux = (i - 1) * M
    for j = 1, M do
        mt[aux + j] = 0
    end
end

```

应用程序中经常会用到稀疏矩阵 (*sparse matrix*)，这种矩阵中的大多数元素是 0 或 nil。例如，我们可以使用邻接矩阵 (*adjacency matrix*) 来表示图。当矩阵 (m, n) 处元素的值为 x 时，表示图中的节点 m 和 n 是相连的，连接的权重为 x ；若上述的两个节点不相连，那么矩阵的 (m, n) 处元素的值为 nil。如果要表示一个具有 1 万个节点的图（其中每个节点有 5 个邻居），那么需要一个能包含 1 亿个元素的矩阵（10000 列 \times 10000 行的方阵），但是其中大约只有 5 万个元素不为 nil（每行有 5 列不为 nil，对应每个节点有 5 个邻居）。许多有关数据结构的书籍都会深入地讨论如何实现这种稀疏矩阵而不必浪费 800MB 内存空间，但在 Lua 语言中却很少需要用到那些技巧。这是因为，我们使用表实现数组而表本来就是稀疏的。在第一种实现中（表的表），需要 1 万个表，每个表包含 5 个元素，总共 5 万个元素。在第二种实现中，只需要一个表，其中包含 5 万个元素。无论哪种实现，都只有非 nil 的元素才占用

空间。

由于在有效元素之间存在空洞 (nil 值)，因此不能对稀疏矩阵使用长度运算符。这没什么大不了的，即使我们能够使用长度运算符，最好也不要那么做。对于大多数针对稀疏矩阵的操作来说，遍历空元素是非常低效的。相反，可以使用 pairs 来只遍历非 nil 的元素。例如，考虑如何进行由不规则数组表示的稀疏矩阵的矩阵乘法。

假设矩阵 $a[M, K]$ 乘以矩阵 $b[K, N]$ 的结果为矩阵 $c[M, N]$ ，常见的矩阵相乘算法形如：

```

for i = 1, M do
    for j = 1, N do
        c[i][j] = 0
        for k = 1, K do
            c[i][j] = c[i][j] + a[i][k] * b[k][j]
        end
    end
end

```

外层的两个循环遍历了整个结果矩阵，然后使用内层循环计算每一个元素的值。

对于使用不规则矩阵实现的稀疏矩阵，内层循环会有问题。由于内层循环遍历的是一列 b 而不是一行，因此不能在此处使用 pairs：这个循环必须遍历每一行来检查对应的行是否在对应列中有元素。除了遍历了少量非 0 元素以外，这个循环还遍历了所有的 0 元素。(由于不知道元素的空间位置，所以在其他场景下遍历一列也可能有问题。)

以下的算法与之前的示例非常类似，但是该算法调换了两个内层循环的顺序。通过这个简单的调整，该算法避免了遍历列：

```

-- 假设 'c' 的元素都是0
for i = 1, M do
    for k = 1, K do
        for j = 1, N do
            c[i][j] = c[i][j] + a[i][k] * b[k][j]
        end
    end
end

```

这样，中间的一层循环遍历行 $a[i]$ ，而内层循环遍历行 $b[k]$ 。这两个遍历都可以使用 pairs 来实现仅遍历非 0 元素。由于一个空的稀疏矩阵本身就是使用 0 填充的，所以对结果矩阵 c 的初始化没有任何问题。

示例 14.1 展示了上述算法的完整实现，其中使用了 `pairs` 来处理稀疏的元素。这种实现只访问非 `nil` 元素，同时结果也是稀疏矩阵。此外，下面的代码还删去了结果中偶然为 0 的元素。

示例 14.1 稀疏矩阵相乘

```
function mult (a, b)
    local c = {}           -- 结果矩阵
    for i = 1, #a do
        local resultline = {}           -- 即 'c[i]'
        for k, va in pairs(a[i]) do      -- 'va' 即 a[i][k]
            for j, vb in pairs(b[k]) do      -- 'vb' 即 b[k][j]
                local res = (resultline[j] or 0) + va * vb
                resultline[j] = (res ~= 0) and res or nil
            end
        end
        c[i] = resultline
    end
    return c
end
```

14.3 链表

由于表是动态对象，所以在 Lua 语言中可以很容易地实现链表（linked list）。我们可以把每个节点用一个表来表示（也只能用表表示），链接则为一个包含指向其他表的引用的简单表字段。例如，让我们实现一个单链表（singly-linked list），其中每个节点具有两个字段 `value` 和 `next`。最简单的变量就是根节点：

```
list = nil
```

要在表头插入一个值为 `v` 的元素，可以使用如下的代码：

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

可以通过如下的方式遍历链表：

```
local l = list
```

```

while l do
    visit l.value
    l = l.next
end

```

诸如双向链表（doubly-linked list）或环形表（circular list）等其他类型的链表也很容易实现。不过，由于通常无须链表即可用更简单的方式来表示数据，所以在 Lua 语言中很少需要用到这些数据结构。例如，我们可以通过一个无界数组（unbounded array）来表示栈。

14.4 队列及双端队列

在 Lua 语言中实现队列（queue）的一种简单方法是使用 `table` 标准库中的函数 `insert` 和 `remove`。正如我们在 5.6 节中所看到的，这两个函数可以在一个数组的任意位置插入或删除元素，同时根据所做的操作移动其他元素。不过，这种移动对于较大的结构来说开销很大。一种更高效的实现是使用两个索引，一个指向第一个元素，另一个指向最后一个元素。使用这种实现方式，我们就可以像在示例 14.2 中所展示的那样以 $O(1)$ 时间复杂度同时在首尾两端插入或删除元素了。

示例 14.2 一个双端队列

```

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

function pushFirst (list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end

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

```

```

function popFirst (list)
    local first = list.first
    if first > list.last then error("list is empty") end
    local value = list[first]
    list[first] = nil      -- 使得元素能够被垃圾回收
    list.first = first + 1
    return value
end

function popLast (list)
    local last = list.last
    if list.first > last then error("list is empty") end
    local value = list[last]
    list[last] = nil      -- 使得元素能够被垃圾回收
    list.last = last - 1
    return value
end

```

如果希望严格地遵循队列的规范使用这个结构，那么就只能调用 `pushLast` 和 `popFirst` 函数，`first` 和 `last` 都会不断地增长。不过，由于我们在 Lua 语言中使用表来表示数组，所以我们既可以在 1~20 的范围内对数组进行索引，也可以在 16777201~16777220 的范围内索引数组。对于一个 64 位整型数而言，以每秒 1000 万次的速度进行插入也需要运行 3 万年才会发生溢出的问题。

14.5 反向表

正如此前提到的，我们很少在 Lua 语言中进行搜索操作。相反，我们使用被称为索引表（index table）或反向表（reverse table）的数据结构。

假设有一个存放了一周每一天名称的表：

```

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

```

如果想要将一周每一天的名称转换为其在一周里的位置，那么可以通过搜索这个表来寻找指定的名称。不过，一种更高效的方式是构造一个反向表，假定为 revDays，该表中的索引为一周每一天的名称而值为其在一周里的位置。这个表形如：

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

然后，只需要直接在反向表中根据名称进行索引就可以了：

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

当然，这个反向表不用手工声明，可以从原始的表中自动地构造出反向表：

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

上例中的循环会对每个元素 days 进行赋值，变量 k 获取到的是键 (1, 2, ...) 而变量 v 获取到的是值 ("Sunday", "Monday", ...)。

14.6 集合与包

假设我们想列出一个程序源代码中的所有标识符，同时过滤掉其中的保留字。一些 C 程序员可能倾向于使用字符串数组来表示保留字集合，然后搜索这个数组来决定某个单词是否属于该集合。为了提高搜索的速度，他们还可能会使用二叉树来表示该集合。

在 Lua 语言中，还可以用一种高效且简单的方式来表示这类集合，即将集合元素作为索引放入表中。那么，对于指定的元素无须再搜索表，只需用该元素检索表并检查结果是否为 nil 即可。以上述需求为例，代码形如：

```
reserved = {
    ["while"] = true,     ["if"] = true,
    ["else"] = true,      ["do"] = true,
}
```

```

for w in string.gmatch(s, "[%a_][%w_]*") do
    if not reserved[w] then
        do something with 'w' -- 'w'不是一个保留字
    end
end

```

(在定义 `reserved` 时, 由于 `while` 是 Lua 语言的保留字, 所以不能直接写成 `while=true`, 而应该写为 `["while"] = true`。)

我们可以借助一个辅助函数来构造集合, 使得初始化过程更清晰:

```

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

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

```

我们还可以使用另一个集合来保存标识符:

```

local ids = {}
for w in string.gmatch(s, "[%a_][%w_]*") do
    if not reserved[w] then
        ids[w] = true
    end
end

-- 输出每一个标识符
for w in pairs(ids) do print(w) end

```

包 (bag), 也被称为多重集合 (*multiset*), 与普通集合的不同之处在于其中的元素可以出现多次。在 Lua 语言中, 包的简单表示类似于此前集合的表示, 只不过其中的每一个键都有一个对应的计数器。^①如果要插入一个元素, 可以递增其计数器:

```
function insert (bag, element)
```

^① 我们已经在第11章的计算出现频率最高的单词的程序中使用过这种表示方法。

```

bag[element] = (bag[element] or 0) + 1
end

```

如果要删除一个元素，可以递减其计数器：

```

function remove(bag, element)
    local count = bag[element]
    bag[element] = (count and count > 1) and count - 1 or nil
end

```

只有当计数器存在且大于 0 时我们才会保留计数器。

14.7 字符串缓冲区

假设我们正在开发一段处理字符串的程序，比如逐行地读取一个文件。典型的代码可能形如：

```

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

```

虽然这段 Lua 语言代码看似能够正常工作，但实际上在处理大文件时却可能导致巨大的性能开销。例如，在笔者的新机器上用这段代码读取一个 4.5MB 大小的文件需要超过 30 秒的时间。

这是为什么呢？为了搞清楚到底发生了什么，让我们想象一下读取循环中发生了什么。假设每行有 20 字节，当我们读取了大概 2500 行后，buff 就会变成一个 50KB 大小的字符串。在 Lua 语言中进行字符串连接 `buff..line..\n` 时，会创建一个 50020 字节的新字符串，然后从 `buff` 中复制 50000 字节中到这个新字符串中。这样，对于后续的每一行，Lua 语言都需要移动大概 50KB 且还在不断增长的内存。因此，该算法的时间复杂度是二次方的。在读取了 100 行（仅 2KB）以后，Lua 语言就已经移动了至少 5MB 内存。当 Lua 语言完成了 350KB 的读取后，它已经至少移动了 50GB 的数据。（这个问题不是 Lua 语言特有的：在其他语言中，只要字符串是不可变值（immutable value），就会出现类似的问题，其中最有名的例子就是 Java。）

在继续学习之前，我们必须说明，上述场景中的情况并不常见。对于较小的字符串，上述循环并没什么问题。当读取整个文件时，Lua 语言提供了带有参数的函数 `io.read("a")` 来一次性地读取整个文件。不过，有时候我们必须面对这个问题。Java 提供了 `StringBuffer` 类来解决这个问题；而在 Lua 语言中，我们可以把一个表当作字符串缓冲区，其关键是使用函数 `table.concat`，这个函数会将指定列表中的所有字符串连接起来并返回连接后的结果。使用函数 `concat` 可以这样重写上述循环：

```
local t = {}
for line in io.lines() do
    t[#t + 1] = line .. "\n"
end
local s = table.concat(t)
```

之前的代码读取同样的文件需要超过半分钟，而上述实现则只需要不到 0.05 秒。（不过尽管如此，读取整个文件最好还是使用带有参数“a”的 `io.read` 函数。）

我们还可以做得更好。函数 `concat` 还有第 2 个可选参数，用于指定插在字符串间的分隔符。有了这个分隔符，我们就不必在每行后插入换行符了。

```
local t = {}
for line in io.lines() do
    t[#t + 1] = line
end
s = table.concat(t, "\n") .. "\n"
```

虽然函数 `concat` 能够在字符串之间插入分隔符，但我们还需要增加最后一个换行符。最后一次字符串连接创建了结果字符串的一个副本，这个副本可能已经相当长了。虽然没有直接的选项能够让函数 `concat` 插入这个额外的分隔符，但我们可以想办法绕过，只需在字符串 `t` 后面添加一个空字符串就行了：

```
t[#t + 1] = ""
s = table.concat(t, "\n")
```

现在，正如我们所期望的那样，函数 `concat` 会在结果字符串的最后添加一个换行符。

14.8 图形

像其他现代编程语言一样，Lua语言也允许开发人员使用多种实现表示图，每种实现都有其所适用的特定算法。这里，我们接下来将介绍一种简单的面向对象的实现方式，在这种实现中使用对象来表示节点（实际上是表）、将边（arc）表示为节点之间的引用。

我们使用一个由两个字段组成的表来表示每个节点，即 name（节点的名称）和 adj（与此节点邻接的节点的集合）。由于我们会从一个文本文件中加载图对应的数据，所以需要能够根据节点的名称来寻找指定节点的方法。因此，我们使用了一个额外的表来建立节点和节点名称之间的映射。函数 name2node 可以根据指定节点的名称返回对应的节点：

```
local function name2node (graph, name)
    local node = graph[name]
    if not node then
        -- 节点不存在，创建一个新节点
        node = {name = name, adj = {}}
        graph[name] = node
    end
    return node
end
```

示例 14.3 展示了构造图的函数。

示例 14.3 从文件中加载图

```
function readgraph ()
    local graph = {}
    for line in io.lines() do
        -- 把一行分割为两个名字
        local namefrom, nameto = string.match(line, "(%S+)%s+(%S+)")
        -- 找到对应的节点
        local from = name2node(graph, namefrom)
        local to = name2node(graph, nameto)
        -- 把'to'增加到邻接集合'from'中
        from.adj[to] = true
    end
    return graph
end
```

```
end
```

该函数逐行地读取一个文件，文件的每一行中有两个节点的名称，表示从第 1 个节点到第 2 个节点有一条边。对于每一行，调用函数 `string.match` 将一行中的两个节点的名称分开，然后根据名称找到对应的节点（如果需要的话则创建节点），最后将这些节点连接在一起。

示例 14.4 展示了一个使用这种图的算法。

示例 14.4 寻找两个节点之间的路径

```
function findpath (curr, to, path, visited)
    path = path or {}
    visited = visited or {}
    if visited[curr] then      -- 是否节点已被访问?
        return nil            -- 不存在路径
    end
    visited[curr] = true       -- 标记节点为已被访问
    path[#path + 1] = curr     -- 增加到路径中
    if curr == to then        -- 是否是最后一个节点?
        return path
    end
    -- 尝试所有的邻接节点
    for node in pairs(curr.adj) do
        local p = findpath(node, to, path, visited)
        if p then return p end
    end
    table.remove(path)         -- 从路径中删除节点
end
```

函数 `findpath` 使用深度优先遍历搜索两个节点之间的路径。该函数的第一个参数是当前节点，第二个参数是目标节点，第三个参数用于保存从起点到当前节点的路径，最后一个参数为所有已被访问节点的集合（用于避免回路）。请读者注意分析该算法是如何不通过节点名称而直接对节点进行操作的。例如，`visited` 是一个节点的集合，而不是节点名称的集合。类似地，`path` 也是一个节点的列表。

为了测试上述代码，我们编写一个打印一条路径的函数，再编写一些代码让上述所有代码跑起来：

```

function printpath (path)
    for i = 1, #path do
        print(path[i].name)
    end
end

g = readgraph()
a = name2node(g, "a")
b = name2node(g, "b")
p = findpath(a, b)
if p then printpath(p) end

```

14.9 练习

练习 14.1: 请编写一个函数, 该函数用于两个稀疏矩阵相加。

练习 14.2: 改写示例 14.2 中队列的实现, 使得当队列为空时两个索引都返回 0。

练习 14.3: 修改图所用的数据结构, 使得图可以保存每条边的标签。该数据结构应该使用包括两个字段的对象来表示每一条边, 即边的标签和边指向的节点。与邻接集合不同, 每一个节点保存的是从当前节点出发的边的集合。

修改函数 `readgraph`, 使得该函数从输入文件中按行读取, 每行由两个节点的名称外加边的标签组成 (假设标签是一个数字)。

练习 14.4: 假设图使用上一个练习的表示方式, 其中边的标签代表两个终端节点之间的距离。请编写一个函数, 使用 Dijkstra 算法寻找两个指定节点之间的最短路径。

15

数据文件和序列化

在处理数据文件时，写数据通常比读数据简单得多。当向一个文件中写时，我们拥有绝对的控制权；但是，当从一个文件中读时，我们并不知道会读到什么东西。一个健壮的程序除了能够处理一个合法文件中所包含的所有类型的数据外，还应该能够优雅地处理错误的文件。因此，编写一个健壮的处理输入的程序总是比较困难的。在本章中，我们会学习如何使用 Lua 语言、通过简单地将数据以恰当的格式写入到文件中来从程序中剔除不必要的读取数据的代码。更确切地说，我们将学习如何像 Lua 程序在运行中写入数据那样，在运行时重建数据。

Lua 语言自 1993 年发布以来，其主要用途之一就是描述数据（data description）。在那个年代，主要的文本数据描述语言之一是 SGML。对于很多人来说（包括我们在内），SGML 既臃肿又复杂。在 1998 年，有些人将其简化成了 XML，但以我们的眼光看仍然臃肿又复杂。有些人跟我们的观点一致，进而在 2001 年开发了 JSON。JSON 基于 JavaScript，类似于一种精简过的 Lua 语言数据文件。一方面，JSON 的一大优势在于它是国际标准，包括 Lua 语言在内的多种语言都具有操作 JSON 文件的标准库。另一方面，Lua 语言数据文件的读取更加容易和灵活。

使用一门全功能的编程语言来描述数据确实非常灵活，但也会带来两个问题。问题之一在于安全性，这是因为“数据”文件能够肆意地在我们的程序中运行。我们可以通过在沙盒中运行程序来解决这个问题，详见 25.4 节。

另一个问题是性能问题。Lua 语言不仅运行得快，编译也很快。例如，在笔者的新机器上，Lua 5.3 可以在 4 秒以内，占用 240MB 内存，完成 1000 万条赋值语句的读取、编译和运行。作为对比，Perl 5.18 需要 21 秒、占用 6GB 内存，Python 2.7 和 Python 3.4 直接崩溃，Node.js 0.10.25 在运行 8 秒后抛出“内存溢出（out of memory）”异常，Rhino 1.7 在运行 6 分钟后也抛出了“内存溢出”异常。

15.1 数据文件

对于文件格式来说，表构造器提供了一种有趣的替代方式。只需在写入数据时做一点额外的工作，就能使得读取数据变得容易。这种技巧就是将数据文件写成 Lua 代码，当这些代码运行时，程序也就把数据重建了。使用表构造器时，这些代码段看上去会非常像是一个普通的数据文件。

下面通过一个示例来进一步展示处理数据文件的方式。如果数据文件使用的是诸如 CSV（comma-separated value，逗号分隔值）或 XML 等预先定义好的格式，那么我们能够选择的方法不多。不过，如果处理的是出于自身需求而创建的数据文件，那么就可以将 Lua 语言的构造器用于格式定义。此时，我们把每条数据记录表示为一个 Lua 构造器。这样，原来类似

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

的数据文件就可以改为：

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

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

请注意，`Entry{code}` 与 `Entry({code})` 是相同的，后者以表作为唯一的参数来调用函数 `Entry`。因此，上面这段数据也是一个 Lua 程序。当需要读取该文件时，我们只需要定义一

个合法的 Entry，然后运行这个程序即可。例如，以下的代码用于计算某个数据文件中数据条目的个数^①：

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

下面的程序获取某个数据文件中所有作者的姓名，然后打印出这些姓名：

```
local authors = {}      -- 保存作者姓名的集合
function Entry (b) authors[b[1]] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

请注意，上述的代码段中使用了事件驱动(event-driven)的方式：函数 Entry 作为一个回调函数会在函数 `dofile` 处理数据文件中的每个条目时被调用。

当文件的大小并不是太大时，可以使用键值对的表示方法：^②

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

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

^①译者注：原文中并未对 `dofile` 函数进行解释，读者可以查阅 Lua 语言的手册或参考本书中后续章节的相关内容来了解 `dofile` 函数的使用方法。实际上，下例中 `dofile("data")` 的 `data` 是数据文件的相对路径，数据文件中存放的内容是上述的两个实体，这样，当 `dofile` 执行数据文件时会对每一个实体调用一次 `Entry` 函数。

^②如果这种格式让读者想起 `BIBTEX`，这并非巧合。`BIBTEX` 正是 Lua 语言构造器语法的灵感来源之一。

这种格式是所谓的自描述数据 (*self-describing data*) 格式，其中数据的每个字段都具有一个对应其含义的简略描述。自描述数据比 CSV 或其他压缩格式的可读性更好（至少看上去如此）；同时，当需要修改时，自描述数据也易于手工编辑；此外，自描述数据还允许我们在不改变数据文件的情况下对基本数据格式进行细微的修改。例如，当我们想要增加一个新字段时，只需对读取数据文件的程序稍加修改，使其在新字段不存在时使用默认值。

使用键值对格式时，获取作者姓名的程序将变为：

```
local authors = {}      -- 保存作者姓名的集合
function Entry (b) authors[b.author] = true end
dofile("data")
for name in pairs(authors) do print(name) end
```

此时，字段的次序就无关紧要了。即使有些记录没有作者字段，我们也只需要修改 `Entry` 函数：

```
function Entry (b)
    authors[b.author or "unknown"] = true
end
```

15.2 序列化

我们常常需要将某些数据序列化/串行化，即将数据转换为字节流或字符流，以便将其存储到文件中或者通过网络传输。我们也可以将序列化后的数据表示为 Lua 代码，当这些代码运行时，被序列化的数据就可以在读取程序中得到重建^①。

通常，如果想要恢复一个全局变量的值，那么可能会使用形如 `varname=exp` 这样的代码。其中，`exp` 是用于创建这个值的 Lua 代码，而 `varname` 是一个简单的标识符。接下来，让我们学习如何编写创建值的代码。例如，对于一个数值类型而言，可以简单地使用如下代码：

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

^①译者注：原文混用了序列化/反序列化、写入 Write/读取 Read、保存 Save/恢复 Restore/重建等词汇，为了便于读者理解，译者尽可能统一使用序列化和反序列化这一对词，请读者注意体会。

```
    end
end
```

不过，用十进制格式保存浮点数可能损失精度。此时，可以利用十六进制格式来避免这个问题，使用格式 "%a" 可以保留被读取浮点型数的原始精度。此外，由于从 Lua 5.3 开始就对浮点类型和整数类型进行了区分，因此通过使用正确的子类型就能够恢复它们的值：

```
local fmt = {integer = "%d", float = "%a"}

function serialize(o)
    if type(o) == "number" then
        io.write(string.format(fmt[math.type(o)], o))
    else other cases
    end
end
```

对于字符串类型的值，最简单的序列化方式形如：

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

不过，如果字符串包含特殊字符（比如引号或换行符），那么结果就会是错误的。

也许有人会告诉读者通过修改引号来解决这个问题：

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

这里，要当心代码注入（code injection）！如果某个恶意用户设法使读者的程序保存了形如 "[[]].os.execute('rm *')..[]]" 这样的内容（例如，恶意用户可以将其住址保存为该字符串），那么最终被保存下来的代码将变成：

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

一旦这样的“数据”被加载，就会导致意想不到的后果。

我们可以使用一种安全的方法来括住一个字符串，那就是使用函数 `string.format` 的 "%q" 选项，该选项被设计为以一种能够让 Lua 语言安全地反序列化字符串的方式来序列化字符串，它使用双引号括住字符串并正确地转义其中的双引号和换行符等其他字符。

```
a = 'a "problematic" \\string'
print(string.format("%q", a)) --> "a \"problematic\" \\string"
```

通过使用这个特性，函数 `serialize` 将变为：

```

function serialize (o)
    if type(o) == "number" then
        io.write(string.format(fmt[math.type(o)], o))
    elseif type(o) == "string" then
        io.write(string.format("%q", o))
    else other cases
        end
    end
end

```

Lua 5.3.3 对格式选项 "%q" 进行了扩展，使其也可以用于数值、nil 和 Boolean 类型，进而使它们能够正确地被序列化和反序列化。（特别地，这个格式选项以十六进制格式处理浮点类型以保留完整的精度。）因此，从 Lua 5.3.3 开始，我们还能够再对函数 `serialize` 进行进一步的简化和扩展：

```

function serialize (o)
    local t = type(o)
    if t == "number" or t == "string" or t == "boolean" or
       t == "nil" then
        io.write(string.format("%q", o))
    else other cases
        end
    end
end

```

另一种保存字符串的方式是使用主要用于长字符串的 [= [...] =]。不过，这种方式主要是为不用改变字符串常量的手写代码提供的。在自动生成的代码中，像函数 `string.format` 那样使用 "%q" 选项来转义有问题的字符更加简单。

尽管如此，如果要在自动生成的代码中使用 [= [...] =]，那么还必须注意几个细节。首先，我们必须选择恰当数量的等号，这个恰当的数量应比原字符串中出现的最长等号序列的长度大 1。由于在字符串中出现长等号序列很常见（例如代码中的注释），因此我们应该把注意力集中在以方括号开头的等号序列上。其次，Lua 语言总是会忽略长字符串开头的换行符，要解决这个问题可以通过一种简单方式，即总是在字符串开头多增加一个换行符（这个换行符会被忽略）。

示例 15.1 中的函数 `quote` 考虑了上述的注意事项。



示例 15.1 引用任意字符串常量

```

function quote (s)
    -- 寻找最长等号序列的长度
    local n = -1
    for w in string.gmatch(s, "]=*") do
        n = math.max(n, #w - 1) -- -1用于移除']'
    end

    -- 生成一个具有'n'+1个等号的字符串
    local eq = string.rep("=", n + 1)

    -- 创建被引起来的字符串
    return string.format(" [%s[\n%s]%s] ", eq, s, eq)
end

```

该函数可以接收任意一个字符串，并返回按长字符串对其进行格式化后的结果。函数 `gmatch` 创建一个遍历字符串 `s` 中所有匹配模式 '`]=*`' 之处的迭代器（即右方括号后跟零个或多个等号）。在每个匹配的地方，循环会用当前所遇到的最大等号数量更新变量 `n`。循环结束后，使用函数 `string.rep` 重复等号 `n+1` 次，也就是生成一个比原字符串中出现的最长等号序列的长度大 1 的等号序列。最后，使用函数 `string.format` 将 `s` 放入一对具有正确数量等号的方括号中，并在字符串 `s` 的开头插入一个换行符。

15.2.1 保存不带循环的表

接下来，更难一点的需求是保存表。保存表有几种方法，选用哪种方法取决于对具体表结构的假设，但没有一种算法适用于所有的情况。对于简单的表来说，不仅可以使用更简单的算法，而且输出也会更简洁和清晰。

第一种尝试参见示例 15.2。

示例 15.2 不使用循环序列化表

```

function serialize (o)
    local t = type(o)
    if t == "number" or t == "string" or t == "boolean" or

```



```

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

```

尽管这个函数很简单，但它却可以合理地满足需求。只要表结构是一棵树（即没有共享的子表和环），那么该函数甚至能处理嵌套的表（即表中还有其他的表）。（在输出中以缩进形式输出嵌套表看上去会更具美感，请参见练习 15.1。）

上例中的函数假设了表中的所有键都是合法的标识符，如果一个表的键是数字或者不是合法的 Lua 标识符，那么就会有问题。解决该问题的一种简单方式是像下列代码一样处理每个键：

```
io.write(string.format(" [%s] = ", serialize(k)))
```

经过这样的修改后，我们提高了该函数的健壮性，但却牺牲了结果文件的美观性。考虑如下的调用：

```
serialize{a=12, b='Lua', key='another "one"'}  
{ a = 12,  
  b = "Lua",  
  key = "another \"one\"",  
}
```

第 1 版的函数 `serialize` 会输出：

```
{  
  a = 12,  
  b = "Lua",  
  key = "another \"one\"",  
}
```

与之对比，第 2 版的函数 `serialize` 则会输出：



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

通过测试每个键是否需要方括号，可以在健壮性和美观性之间得到平衡。同样，我们将此实现留做练习。

15.2.2 保存带有循环的表

由于表构造器不能创建带循环的或共享子表的表，所以如果要处理表示通用拓扑结构（例如带循环或共享子表）的表，就需要采用不同的方法。我们需要引入名称来表示循环。因此，下面的函数把值外加其名称一起作为参数。另外，还必须使用一个额外的表来存储已保存表的名称，以便在发现循环时对其进行复用。这个额外的表使用此前已被保存的表作为键，以表的名称作为值。

示例 15.3 中为相应的代码。

示例 15.3 保存带有循环的表

```
function basicSerialize (o)
    -- 假设'o'是一个数字或字符串
    return string.format("%q", o)
end

function save (name, value, saved)
    saved = saved or {}                      -- 初始值
    io.write(name, " = ")
    if type(value) == "number" or type(value) == "string" then
        io.write(basicSerialize(value), "\n")
    elseif type(value) == "table" then
        if saved[value] then                  -- 值是否已被保存?
            io.write(saved[value], "\n")      -- 使用之前的名称
        else
            saved[value] = name           -- 保存名称供后续使用
        end
    end
end
```



```

io.write("{}\n")           -- 创建新表
for k,v in pairs(value) do
    k = basicSerialize(k)
    local fname = string.format("%s[%s]", name, k)
    save(fname, v, saved)
end
else
    error("cannot save a " .. type(value))
end
end

```

我们假设要序列化的表只使用字符串或数值作为键。函数 `basicSerialize` 用于对这些基本类型进行序列化并返回序列化后的结果，另一个函数 `save` 则完成具体的工作，其参数 `saved` 就是之前所说的用于存储已保存表的表。例如，假设要创建一个如下所示的表：

```

a = {x=1, y=2; {3,4,5}}
a[2] = a      -- 循环
a.z = a[1]    -- 共享子表
调用 save("a", a) 会将其保存为:

```

```

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

```

```

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

```

取决于表的遍历情况，这些赋值语句的实际执行顺序可能会有所不同。不过尽管如此，上述算法能够保证任何新定义节点中所用到的节点都是已经被定义过的。

如果想保存具有共享部分的几个表，那么可以在调用函数 `save` 时使用相同的表 `saved` 函数。例如，假设有如下两个表：



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

如果以独立的方式保存这些表，那么结果中不会有共同的部分。不过，如果调用 `save` 函数时使用同一个表 `saved`，那么结果就会共享共同的部分：

```
local t = {}
save("a", a, t)
save("b", b, t)

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

在 Lua 语言中，还有其他一些比较常见的方法。例如，我们可以在保存一个值时不指定全局名称而是通过一段代码来创建一个局部值并将其返回，也可以在可能的时候使用列表的语法（参见本章的练习），等等。Lua 语言给我们提供了构建这些机制的工具。

15.3 练习

练习 15.1: 修改示例 15.2 中的代码，使其带缩进地输出嵌套表（提示：给函数 `serialize` 增加一个额外的参数来处理缩进字符串）。

练习 15.2: 修改前面练习中的代码，使其像 15.2.1 节中推荐的那样使用形如 `["key"] = value` 的语法。

练习 15.3: 修改前面练习中的代码，使其只在必要时（即当键为字符串而不是合法标识符时）才使用形如 `["key"] = value` 的语法。

练习 15.4: 修改前面练习中的代码，使其在可能时使用列表的构造器语法。例如，应将表 `{14, 15, 19}` 序列化为 `{14, 15, 19}` 而不是 `{[1] = 14, [2] = 15, [3] = 19}`（提示：只要键不是 `nil` 就从 1, 2, … 开始保存对应键的值。请注意，在遍历其余表的时候不要再次保存它们）。



练习 15.5：在保存具有循环的表时，避免使用构造器的方法过于激进了。对于简单的情况，是能够使用表构造器以一种更加优雅的方式来保存表的，并且也能够在后续使用赋值语句来修复共享表和循环。请使用这种方式重新实现函数 `save`（示例 15.3），其中要运用前面练习中的所有优点（缩进、记录式语法及列表式语法）。



16

编译、执行和错误

虽然我们把 Lua 语言称为解释型语言 (interpreted language)，但 Lua 语言总是在运行代码前先预编译 (precompile) 源码为中间代码 (这没什么大不了的，很多解释型语言也这样做)。编译 (compilation) 阶段的存在听上去超出了解释型语言的范畴，但解释型语言的区分并不在于源码是否被编译，而在于是否有能力 (且轻易地) 执行动态生成的代码。可以认为，正是由于诸如 `dofile` 这样函数的存在，才使得 Lua 语言能够被称为解释型语言。

在本章中，我们会详细学习 Lua 语言运行代码的过程、编译究竟是什么意思和做了什么、Lua 语言是如何运行编译后代码的以及在编译过程中如何处理错误。

16.1 编译

此前，我们已经介绍过函数 `dofile`，它是运行 Lua 代码段的主要方式之一。实际上，函数 `dofile` 是一个辅助函数，函数 `loadfile` 才完成了真正的核心工作。与函数 `dofile` 类似，函数 `loadfile` 也是从文件中加载 Lua 代码段，但它不会运行代码，而只是编译代码，然后将编译后的代码段作为一个函数返回。此外，与函数 `dofile` 不同，函数 `loadfile` 只返回错误码而不抛出异常。可以认为，函数 `dofile` 就是：

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



请注意，如果函数 `loadfile` 执行失败，那么函数 `assert` 会引发一个错误^①。

对于简单的需求而言，由于函数 `dofile` 在一次调用中就做完了所有工作，所以该函数非常易用。不过，函数 `loadfile` 更灵活。在发生错误的情况下，函数 `loadfile` 会返回 `nil` 及错误信息，以允许我们按自定义的方式来处理错误。此外，如果需要多次运行同一个文件，那么只需调用一次 `loadfile` 函数后再多次调用它的返回结果即可。由于只编译一次文件，因此这种方式的开销要比多次调用函数 `dofile` 小得多（编译在某种程度上相比其他操作开销更大）。

函数 `load` 与函数 `loadfile` 类似，不同之处在于该函数从一个字符串或函数中读取代码段，而不是从文件中读取。^②例如，考虑如下的代码：

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

在这句代码执行后，变量 `f` 就会变成一个被调用时执行 `i=i+1` 的函数：

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

尽管函数 `load` 的功能很强大，但还是应该谨慎地使用。相对于其他可选的函数而言，该函数的开销较大并且可能会引起诡异的问题。请先确定当下已经找不到更简单的解决方式后再使用该函数。

如果要编写一个用后即弃的 `dostring` 函数（例如加载并运行一段代码），那么我们可以直接调用函数 `load` 的返回值：

```
load(s)()
```

不过，如果代码中有语法错误，函数 `load` 就会返回 `nil` 和形如“试图调用一个 `nil` 值 (*attempt to call a nil value*)”的错误信息。为了更清楚地展示错误信息，最好使用函数 `assert`：

```
assert(load(s))()
```

通常，用函数 `load` 来加载字符串常量是没有意义的。例如，如下的两行代码基本等价：

^①译者注：在编程语言中，异常方面通常有“引发错误（raise a error）”和“抛出异常（throw a exception）”两种说法，经常混用。本书原作者倾向于使用前者，但译者认为抛出异常的表达方式更符合中国国情，故本章前的所有译文采用的均是“抛出异常”的译法。而由于本章内容就是针对 Lua 语言的错误处理机制，因此本章中使用“引发错误”的译法。

^②在 Lua 5.1 中，函数 `loadstring` 用于完成 `load` 所完成的从字符串中加载代码的功能。



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

但是，由于第 2 行代码会与其外层的函数一起被编译，所以其执行速度要快得多。与之对比，第一段代码在调用函数 `load` 时会进行一次独立的编译。

由于函数 `load` 在编译时不涉及词法定界，所以上述示例的两段代码可能并不完全等价。为了清晰地展示它们之间的区别，让我们稍微修改一下上面的例子：

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

函数 `g` 像我们所预期地那样操作局部变量 `i`，但函数 `f` 操作的却是全局变量 `i`，这是由于函数 `load` 总是在全局环境中编译代码段。

函数 `load` 最典型的用法是执行外部代码（即那些来自程序本身之外的代码段）或动态生成的代码。例如，我们可能想运行用户定义的函数，由用户输入函数的代码后调用函数 `load` 对其求值。请注意，函数 `load` 期望的输入是一段程序，也就是一系列的语句。如果需要对表达式求值，那么可以在表达式前添加 `return`，这样才能构成一条返回指定表达式值的语句。例如：

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

由于函数 `load` 所返回的函数就是一个普通函数，因此可以反复对其进行调用：

```
print "enter function to be plotted (with variable 'x'):"
local line = io.read()
local f = assert(load("return " .. line))
for i = 1, 20 do
    x = i    -- 全局的'x' (当前代码段内可见)
    print(string.rep("*", f()))
```



```
end
```

我们也可以使用读取函数 (*reader function*) 作为函数 `load` 的第 1 个参数。读取函数可以分几次返回一段程序，函数 `load` 会不断地调用读取函数直到读取函数返回 `nil`（表示程序段结束）。作为示例，以下的调用与函数 `loadfile` 等价：

```
f = load(io.lines(filename, "*L"))
```

正如我们在第7章中所看到的，调用 `io.lines(filename, "*L")` 返回一个函数，这个函数每次被调用时就从指定文件返回一行。因此，函数 `load` 会一行一行地从文件中读出一段程序。以下的版本与之相似但效率稍高：

```
f = load(io.lines(filename, 1024))
```

这里，函数 `io.lines` 返回的迭代器会以 1024 字节为块读取源文件。

`Lua` 语言将所有独立的代码段当作匿名可变长参数函数的函数体。例如，`load("a = 1")` 的返回值与以下表达式等价：

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

像其他任何函数一样，代码段中可以声明局部变量：

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

使用这个特性，可以在不使用全局变量 `x` 的情况下重写之前运行用户定义函数的示例：

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

在上述代码中，在代码段开头增加了"`local x = ...`" 来将 `x` 声明为局部变量。之后使用参数 `i` 调用函数 `f`，参数 `i` 就是可变长参数表达式的值 (...)。

函数 `load` 和函数 `loadfile` 从来不引发错误。当有错误发生时，它们会返回 `nil` 及错误信息：



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

此外，这些函数没有任何副作用，它们既不改变或创建变量，也不向文件写入等。这些函数只是将程序段编译为一种中间形式，然后将结果作为匿名函数返回。一种常见的误解是认为加载一段程序也就是定义了函数，但实际上在 Lua 语言中函数定义是在运行时而不是在编译时发生的一种赋值操作。例如，假设有一个文件 `foo.lua`：

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

当执行

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

时，编译 `foo` 的命令并没有定义 `foo`，只有运行代码才会定义它：

```
f = loadfile("foo.lua")
print(foo)    --> nil
f()          -- 运行代码
foo("ok")    --> ok
```

这种行为可能看上去有些奇怪，但如果不用语法糖对其进行重写则看上去会清晰很多：

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

如果线上产品级别的程序需要执行外部代码，那么应该处理加载程序段时报告的所有错误。此外，为了避免不愉快的副作用发生，可能还应该在一个受保护的环境中执行这些代码。我们会在第 22 章中讨论相关的细节。

16.2 预编译的代码

正如笔者在本章开头所提到的，Lua 语言会在运行源代码之前先对其进行预编译。Lua 语言也允许我们以预编译的形式分发代码。

生成预编译文件（也被称为二进制文件，*binary chunk*）的最简单的方式是，使用标准发行版中附带的 luac 程序。例如，下列命令会创建文件 prog.lua 的预编译版本 prog.lc：

```
$ luac -o prog.lc prog.lua
```

Lua 解析器会像执行普通 Lua 代码一样执行这个新文件，完成与原来代码完全一致的动作：

```
$ lua prog.lc
```

几乎在 Lua 语言中所有能够使用源码的地方都可以使用预编译代码。特别地，函数 `loadfile` 和函数 `load` 都可以接受预编译代码。

我们可以直接在 Lua 语言中实现一个最简单的 luac：

```
p = loadfile(arg[1])
f = io.open(arg[2], "wb")
f:write(string.dump(p))
f:close()
```

这里的关键函数是 `string.dump`，该函数的入参是一个 Lua 函数，返回值是传入函数对应的字符串形式的预编译代码（已被正确地格式化，可由 Lua 语言直接加载）。

luac 程序提供了一些有意思的选择。特别地，选项-l 会列出编译器为指定代码段生成的操作码（opcode）。例如，示例 16.1 展示了函数 luac 针对如下只有一行内容的文件在带有-l 选项时的输出：

```
a = x + y - z
```

示例 16.1 luac -l 的输出示例

```
main <stdin:0,0> (7 instructions, 28 bytes at 0x988cb30)
0+ params, 2 slots, 0 upvalues, 0 locals, 4 constants, 0 functions
 1 [1] GETGLOBAL    0 -2      ; x
 2 [1] GETGLOBAL    1 -3      ; y
 3 [1] ADD         0 0 1
 4 [1] GETGLOBAL    1 -4      ; z
 5 [1] SUB         0 0 1
 6 [1] GETGLOBAL    0 -1      ; a
 7 [1] RETURN       0 1
```

（我们不会在本书中讨论 Lua 语言的内部细节；如果读者对这些操作码的更多细节感兴趣，可以在网上搜索“lua opcode”来获得相关资料。）

预编译形式的代码不一定比源代码更小，但是却加载得更快。预编译形式的代码的另一个好处是，可以避免由于意外而修改源码。然而，与源代码不同，蓄意损坏或构造的二进制代码可能会让 Lua 解析器崩溃或甚至执行用户提供的机器码。当运行一般的代码时通常无须担心，但应该避免运行以预编译形式给出的非受信代码。这种需求，函数 `load` 正好有一个选项可以适用。

除了必需的第 1 个参数外，函数 `load` 还有 3 个可选参数。第 2 个参数是程序段的名称，只在错误信息中被用到。第 4 个参数是环境，我们会在第 22 章中对其进行讨论。第 3 个参数正是我们这里所关心的，它控制了允许加载的代码段的类型。如果该参数存在，则只能是如下的字符串：字符串“`t`”允许加载文本（普通）类型的代码段，字符串“`b`”只允许加载二进制（预编译）类型的代码段，字符串“`bt`”允许同时加载上述两种类型的代码段（默认情况）。

16.3 错误

人人皆难免犯错误^①。因此，我们必须尽可能地处理错误。由于 Lua 语言是一种经常被嵌入在应用程序中的扩展语言，所以当错误发生时并不能简单地崩溃或退出。相反，只要错误发生，Lua 语言就必须提供处理错误的方式。

Lua 语言会在遇到非预期的情况时引发错误。例如，当试图将两个非数值类型的值相加，对不是函数的值进行调用，对不是表类型的值进行索引等（我们会在后续学习中使用元表（*metatable*）来改变上述行为）。我们也可以显式地通过调用函数 `error` 并传入一个错误信息作为参数来引发一个错误。通常，这个函数就是在代码中提示出错的合理方式：

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

由于“针对某些情况调用函数 `error`”这样的代码结构太常见了，所以 Lua 语言提供了一个内建的函数 `assert` 来完成这类工作：

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

^①译者注：原文为 *Errare humanum est*，拉丁语。

函数 `assert` 检查其第 1 个参数是否为真，如果该参数为真则返回该参数；如果该参数为假则引发一个错误。该函数的第 2 个参数是一个可选的错误信息。不过，要注意函数 `assert` 只是一个普通函数，所以 Lua 语言会总是在调用该函数前先对参数进行求值。如果编写形如

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

的代码，那么即使 `n` 是一个数值类型，Lua 语言也总是会进行字符串连接。在这种情况下使用显式的测试可能更加明智。

当一个函数发现某种意外的情况发生时（即异常 *exception*），在进行异常处理（*exception handling*）时可以采取两种基本方式：一种是返回错误代码（通常是 `nil` 或者 `false`），另一种是通过调用函数 `error` 引发一个错误。如何在这两种方式之间进行选择并没有固定的规则，但笔者通常遵循如下的指导原则：容易避免的异常应该引发错误，否则应该返回错误码。

以函数 `math.sin` 为例，当调用时参数传入了一个表该如何反应呢？如果要检查错误，那么就不得不编写如下的代码：

```
local res = math.sin(x)
if not res then      -- 错误?
    error-handling code
```

当然，也可以在调用函数前轻松地检查出这种异常：

```
if not tonumber(x) then      -- x 是否为数字?
    error-handling code
```

通常，我们既不会检查参数也不会检查函数 `sin` 的返回值；如果 `sin` 的参数不是一个数值，那么就意味着我们的程序可能出现了问题。此时，处理异常最简单也是最实用的做法就是停止运行，然后输出一条错误信息。

另一方面，让我们再考虑一下用于打开文件的函数 `io.open`。如果要打开的文件不存在，那么该函数应该有怎么样的行为呢？在这种情况下，没有什么简单的方法可以在调用函数前检测到这种异常。在很多系统中，判断一个文件是否存在的唯一方法就是试着去打开这个文件。因此，如果由于外部原因（比如“文件不存在（file does not exist）”或“权限不足（permission denied）”）导致函数 `io.open` 无法打开一个文件，那么它应返回 `false` 及一条错误信息。通过这种方式，我们就有机会采取恰当的方式来处理异常情况，例如要求用户提供另一个文件名：

```

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

```

如果不想处理这些情况，但又想安全地运行程序，那么只需使用 `assert`:

```

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

```

这是 Lua 语言中一种典型的技巧：如果函数 `io.open` 执行失败，`assert` 就引发一个错误。请读者注意，错误信息（函数 `io.open` 的第 2 个返回值）是如何变成 `assert` 的第 2 个参数的。

16.4 错误处理和异常

对于大多数应用而言，我们无须在 Lua 代码中做任何错误处理，应用程序本身会负责处理这类问题。所有 Lua 语言的行为都是由应用程序的一次调用而触发的，这类调用通常是要求 Lua 语言执行一段代码。如果执行中发生了错误，那么调用会返回一个错误代码，以便应用程序采取适当的行为来处理错误。当独立解释器中发生错误时，主循环会打印错误信息，然后继续显示提示符，并等待执行指定的命令。

不过，如果要在 Lua 代码中处理错误，那么就应该使用函数 `pcall` (*protected call*) 来封装代码。

假设要执行一段 Lua 代码并捕获 (try-catch) 执行中发生的所有错误，那么首先需要将这段代码封装到一个函数中，这个函数通常是一个匿名函数。之后，通过 `pcall` 来调用这个函数：

```

local ok, msg = pcall(function ()
    some code
    if unexpected_condition then error() end
    some code
    print(a[i]) -- 潜在错误: 'a' 可能不是一个表
end)

```

```

    some code
end)

if ok then -- 执行被保护的代码时没有错误发生
    regular code
else -- 执行被保护的代码时有错误发生：进行恰当的处理
    error-handling code
end

```

函数 `pcall` 会以一种保护模式 (*protected mode*) 来调用它的第 1 个参数，以便捕获该函数执行中的错误。无论是否有错误发生，函数 `pcall` 都不会引发错误。如果没有错误发生，那么 `pcall` 返回 `true` 及被调用函数（作为 `pcall` 的第 1 个参数传入）的所有返回值；否则，则返回 `false` 及错误信息。

使用“错误信息”的命名方式可能会让人误解错误信息必须是一个字符串，因此称之为错误对象 (*error object*) 可能更好，这主要是因为函数 `pcall` 能够返回传递给 `error` 的任意 Lua 语言类型的值。

```

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

```

这些机制为我们提供了在 Lua 语言中进行异常处理的全部。我们可以通过 `error` 来抛出异常 (throw an exception)，然后用函数 `pcall` 来捕获 (catch) 异常，而错误信息则用来标识错误的类型。

16.5 错误信息和栈回溯

虽然能够使用任何类型的值作为错误对象，但错误对象通常是一个描述出错内容的字符串。当遇到内部错误（比如尝试对一个非表类型的值进行索引操作）出现时，Lua 语言负责产生错误对象（这种情况下的错误对象永远是字符串；而在其他情况下，错误对象就是传递给函数 `error` 的值。）如果错误对象是一个字符串，那么 Lua 语言还会尝试把一些有关错误发生位置的信息附上：

```

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

```

位置信息中给出了出错代码段的名称（上例中的 `stdin`）和行号（上例中的 1）。

函数 `error` 还有第 2 个可选参数 `level`，用于指出向函数调用层次中的哪层函数报告错误，以说明谁应该为错误负责。例如，假设编写一个用来检查其自身是否被正确调用了的函数：

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

如果调用时被传递了错误的参数：

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

由于是函数 `foo` 调用的 `error`，所以 Lua 语言会认为是函数 `foo` 发生了错误。然而，真正的肇事者其实是函数 `foo` 的调用者。为了纠正这个问题，我们需要告诉 `error` 函数错误实际发生在函数调用层次的第 2 层中（第 1 层是 `foo` 函数自己）：

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

通常，除了发生错误的位置以外，我们还希望在错误发生时得到更多的调试信息。至少，我们希望得到具有发生错误时完整函数调用栈的栈回溯（traceback）。当函数 `pcall` 返回错误信息时，部分的调用栈已经被破坏了（从 `pcall` 到出错之处的部分）。因此，如果希望得到一个有意义的栈回溯，那么就必须在函数 `pcall` 返回前先将调用栈构造好。为了完成这个需求，Lua 语言提供了函数 `xpcall`。该函数与函数 `pcall` 类似，但它的第 2 个参数是一个消息处理函数（*message handler function*）。当发生错误时，Lua 会在调用栈展开（`stack unwind`）前调用这个消息处理函数，以便消息处理函数能够使用调试库来获取有关错误的更多信息。两个常用的消息处理函数是 `debug.debug` 和 `debug.traceback`，前者为用户提供一个 Lua 提示符来让用户检查错误发生的原因；后者则使用调用栈来构造详细的错误信息，Lua 语言的独立解释器就是使用这个函数来构造错误信息的。

16.6 练习

练习 16.1: 通常，在加载代码段时增加一些前缀很有用。（我们在本章前面部分已经见过相应的例子，在那个例子中，我们在待加载的表达式前增加了一个 `return` 语句。）请编写一个函数 `loadwithprefix`，该函数类似于函数 `load`，不过会将第 1 个参数（一个字符串）增加到待加载的代码段之前。

像原始的 `load` 函数一样，函数 `loadwithprefix` 应该既可以接收字符串形式的代码段，也可以通过函数进行读取。即使待加载的代码段是字符串形式的，函数 `loadwithprefix` 也不应该进行实际的字符串连接操作。相反，它应该调用函数 `load` 并传入一个恰当的读取函数来实现功能，这个读取函数首先返回要增加的代码，然后返回原始的代码段。

练习 16.2: 请编写一个函数 `multiload`，该函数接收一组字符串或函数来生成函数 `loadwithprefix`，如下例所示：

```
f = multiload("local x = 10;",  
              io.lines("temp", "*L"),  
              "print(x)")
```

在上例中，函数 `multiload` 应该加载一段等价于字符串"local..."、`temp` 文件的内容和字符串"print(x)" 连接在一起后的代码。与上一练习中的函数 `loadwithprefix` 类似，函数 `multiload` 也不应该进行任何实际的字符串连接操作。

练习 16.3: 示例 16.2 中的函数 `stringrep` 使用二进制乘法算法（binary multiplication algorithm）来完成将指定字符串 `s` 的 `n` 个副本连接在一起的需求：

示例 16.2 字符串复制

```
function stringrep (s, n)  
    local r = ""  
    if n > 0 then  
        while n > 1 do  
            if n % 2 ~= 0 then r = r .. s end  
            s = s .. s  
            n = math.floor(n / 2)  
        end  
        r = r .. s  
    end
```

```
    return r
end
```

对于任意固定的 n , 我们可以通过将循环展开为一系列的 $r=r..s$ 和 $s=s..s$ 语句来创建一个特殊版本的 `stringrep`。例如, 在 $n=5$ 时可以展开为如下的函数:

```
function stringrep_5 (s)
    local r = ""
    r = r .. s
    s = s .. s
    s = s .. s
    r = r .. s
    return r
end
```

请编写一个函数, 该函数对于指定的 n 返回特定版本的函数 `stringrep_n`。在实现方面, 不能使用闭包, 而是应该构造出包含合理指令序列 ($r=r..s$ 和 $s=s..s$ 的组合) 的 Lua 代码, 然后再使用函数 `load` 生成最终的函数。请比较通用版本的 `stringrep` 函数 (或者使用该函数的闭包) 与我们自己实现的版本之间的性能差异。

练习 16.4: 你能否想到一个使 `pcall(pcall, f)` 的第 1 个返回值为 `false` 的 `f`? 为什么这样的 `f` 会有存在的意义呢?

17 模块和包

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

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

从用户观点来看，一个模块（module）就是一些代码（要么是 Lua 语言编写的，要么是 C 语言编写的），这些代码可以通过函数 `require` 加载，然后创建和返回一个表。这个表就像是某种命名空间，其中定义的内容是模块中导出的东西，比如函数和常量。

例如，所有的标准库都是模块。我们可以按照如下的方法使用数学库：

```
local m = require "math"
print(m.sin(3.14))           --> 0.0015926529164868
```

独立解释器会使用跟如下代码等价的方式提前加载所有标准库：

```
math = require "math"
string = require "string"
...
```

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

使用表来实现模块的显著优点之一是，让我们可以像操作普通表那样操作模块，并且能利用 Lua 语言的所有功能实现额外的功能。在大多数语言中，模块不是第一类值（即它们不能被保存在变量中，也不能被当作参数传递给函数等），所以那些语言需要为模块实现一套专门的机制。而在 Lua 语言中，我们则可以轻易地实现这些功能。

例如，用户调用模块中的函数就有几种方法。其中常见的方法是：

```
local mod = require "mod"
mod.foo()
```

用户可以为模块设置一个局部名称：

```
local m = require "mod"
m.foo()
```

也可以为个别函数提供不同的名称：

```
local m = require "mod"
local f = m.foo
f()
```

还可以只引入特定的函数：

```
local f = require "mod".foo          -- (require("mod")).foo
f()
```

上述这些方法的好处是无须语言的特别支持，它们使用的都是语言已经提供的功能。

17.1 函数 require

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

```
local m = require('math')
```

```
local modname = 'math'
local m = require(modname)
```

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

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

如果模块尚未加载，那么函数 `require` 则搜索具有指定模块名的 Lua 文件（搜索路径由变量 `package.path` 指定，我们会在后续对其进行讨论）。如果函数 `require` 找到了相应的文件，那么就用函数 `loadfile` 将其进行加载，结果是一个我们称之为加载器（*loader*）的函数（加载器就是一个被调用时加载模块的函数^①）。

如果函数 `require` 找不到指定模块名的 Lua 文件，那么它就搜索相应名称的 C 标准库。^②（在这种情况下，搜索路径由变量 `package.cpath` 指定。）如果找到了一个 C 标准库，则使用底层函数 `package.loadlib` 进行加载，这个底层函数会查找名为 `luaopen_modname` 的函数。在这种情况下，加载函数就是 `loadlib` 的执行结果，也就是一个被表示为 Lua 函数的 C 语言函数 `luaopen_modname`。

不管模块是在 Lua 文件还是 C 标准库中找到的，函数 `require` 此时都具有了用于加载它的加载函数。为了最终加载模块，函数 `require` 带着两个参数调用加载函数：模块名和加载函数所在文件的名称（大多数模块会忽略这两个参数）。如果加载函数有返回值，那么函数 `require` 会返回这个值，然后将其保存在表 `package.loaded` 中，以便于将来在加载同一模块时返回相同的值。如果加载函数没有返回值且表中的 `package.loaded[@rep{modname}]` 为空，函数 `require` 就假设模块的返回值是 `true`。如果没有这种补偿，那么后续调用函数 `require` 时将会重复加载模块。

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

```
package.loaded.modname = nil
```

^①译者注：在原书中，作者混用了 loader function、open function 及 initial function，因此后面加载器也与打开函数和初始化函数混用了，但译者倾向于将 loader function 翻译为加载函数。

^②在29.3节中，我们会学习如何编写 C 标准库。

下一次再加载这个模块时，函数 `require` 就会重新加载模块。

对于函数 `require` 来说，一个常见的抱怨是它不能给待加载的模块传递参数。例如，数学模块可以对角度和弧度的选择增加一个选项：

```
-- 错误的代码
local math = require("math", "degree")
```

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

```
local mod = require "mod"
mod.init(0, 0)
```

如果加载函数返回的是模块本身，那么还可以写成：

```
local mod = require "mod".init(0, 0)
```

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

17.1.1 模块重命名

通常，我们通过模块本来的名称来使用它们，但有时，我们也需要将一个模块改名以避免命名冲突。一种典型的情况就是，出于测试的目的而需要加载同一模块的不同版本。对于一个 Lua 语言模块来说，其内部的名称并不要求是固定的，因此通常修改 `.lua` 文件的文件名就够了。不过，我们却无法修改 C 标准库的二进制目标代码中 `luaopen_*` 函数的名称。为了进行这种重命名，函数 `require` 运用了一个连字符的技巧：如果一个模块名中包含连字符，那么函数 `require` 就会用连字符之前的内容来创建 `luaopen_*` 函数的名称。例如，如果一个模块的名称为 `mod-v3.4`，那么函数 `require` 会认为该模块的加载函数应该是 `luaopen_mod` 而不是 `luaopen_mod-v3.4`（这也不是有效的 C 语言函数名）。因此，如果需要使用两个名称均为 `mod` 的模块（或相同模块的两个不同版本），那么可以对其中的一个进行重命名，如 `mod-v1`。当调用 `m1=require "mod-v1"` 时，函数 `require` 会找到改名后的文件 `mod-v1` 并将其中原名为 `luaopen_mod` 的函数作为加载函数。

17.1.2 搜索路径

在搜索一个 Lua 文件时，函数 `require` 使用的路径与典型的路径略有不同。典型的路径是很多目录组成的列表，并在其中搜索指定的文件。不过，ISO C（Lua 语言依赖的抽象平台）并没有目录的概念。所以，函数 `require` 使用的路径是一组模板（*template*），其中的每项都指定了将模块名（函数 `require` 的参数）转换为文件名的方式。更准确地说，这种路径中的每一个模板都是一个包含可选问号的文件名。对于每个模板，函数 `require` 会用模块名来替换每一个问号，然后检查结果是否存在对应的文件；如果不存在，则尝试下一个模板。路径中的模板以在大多数操作系统中很少被用于文件名的分号隔开。例如，考虑如下路径：

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

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

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

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

函数 `require` 用于搜索 Lua 文件的路径是变量 `package.path` 的当前值。当 `package` 模块被初始化后，它就把变量 `package.path` 设置成环境变量 `LUA_PATH_5_3` 的值。如果这个环境变量没有被定义，那么 Lua 语言则尝试另一个环境变量 `LUA_PATH`。如果这两个环境变量都没有被定义，那么 Lua 语言则使用一个编译时定义的默认路径。^①在使用一个环境变量的值时，Lua 语言会将其中所有的";;" 替换成默认路径。例如，如果将 `LUA_PATH_5_3` 设为"`mydir/?..lua;;`"，那么最终路径就会是模板"`mydir/?..lua`" 后跟默认路径。

搜索 C 标准库的路径的逻辑与此相同，只不过 C 标准库的路径来自变量 `package.cpath` 而不是 `package.path`。类似地，这个变量的初始值也来自环境变量 `LUA_CPATH_5_3` 或 `LUA_CPATH`。在 POSIX 系统中这个路径的典型值形如：

```
./?.so;/usr/local/lib/lua/5.2/?.so
```

请注意定义文件扩展名的路径。在上例中，所有模板使用的都是`.so`，而在 Windows 操作系统中此典型路径通常形如：

^①从 Lua 5.2 开始，独立解释器可以通过命令行参数-E 来阻止使用这些环境变量而强制使用默认值。

Lua 程序设计（第 4 版）

```
.\\?.dll;C:\\Program Files\\Lua502\\dll\\?.dll
```

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

```
> path = ".\\?.dll;C:\\Program Files\\Lua502\\dll\\?.dll"
> print(package.searchpath("X", path))
nil
no file '.\\X.dll'
no file 'C:\\Program Files\\Lua502\\dll\\X.dll'
```

作为一个有趣的练习，我们在示例 17.1 中实现了一个与函数 `package.searchpath` 类似的函数。

示例 17.1 实验版的 `package.searchpath`

```
function search (modname, path)
    modname = string.gsub(modname, "%.", "/")
    local msg = {}
    for c in string.gmatch(path, "[^;]+") do
        local fname = string.gsub(c, "?", modname)
        local f = io.open(fname)
        if f then
            f:close()
            return fname
        else
            msg[#msg + 1] = string.format("\n\nno file '%s'", fname)
        end
    end
    return nil, table.concat(msg) -- 没找到
end
```

上述函数首先替换目录分隔符，在本例中即把所有的点换成斜杠（我们会在后续看到模块名中的点具有特殊含义）。之后，该函数遍历路径中的所有组成部分，也就是每一个不含分号的最长匹配。对于每一个组成部分，该函数使用模块名来替换问号得到最终的文件名，然后检查相应的文件是否存在。如果存在，该函数关闭这个文件，然后返回文件的名称；否

则，该函数保存失败的文件名用于可能的错误提示（请注意字符串缓冲区在避免创建无用的长字符串时的作用）。如果一个文件都找不到，该函数则返回 nil 及最终的错误信息。

17.1.3 搜索器

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

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

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

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

默认的 `package.searchers` 中的第 4 个函数只与子模块有关，我们会在 17.3 节对其进行介绍。

17.2 Lua 语言中编写模块的基本方法

在 Lua 语言中创建模块的最简单方法是，创建一个表并将所有需要导出的函数放入其中，最后返回这个表。示例 17.2 演示了这种方法。

示例 17.2 一个用于复数的简单模块

```

local M = {}          -- 模块

-- 创建一个新的复数
local function new (r, i)
    return {r=r, i=i}
end

M.new = new           -- 把'new'加到模块中

-- constant 'i'
M.i = new(0, 1)

function M.add (c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end

function M.sub (c1, c2)
    return new(c1.r - c2.r, c1.i - c2.i)
end

function M.mul (c1, c2)
    return new(c1.r*c2.r - c1.i*c2.i, c1.r*c2.i + c1.i*c2.r)
end

local function inv (c)
    local n = c.r^2 + c.i^2
    return new(c.r/n, -c.i/n)
end

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

```

```

function M.tostring (c)
    return string.format("(%.g,%.g)", c.r, c.i)
end

return M

```

请注意我们是如何通过简单地把 `new` 和 `inv` 声明为局部变量而使它们成为代码段的私有函数（private function）的。

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

```

local M = {}
package.loaded[...] = M

```

跟之前一样，但没有返回语句

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

另一种编写模块的方法是把所有的函数定义为局部变量，然后在最后构造返回的表，参见示例 17.3。

示例 17.3 使用导出表的模块

```

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

-- 定义常量'i'
local i = complex.new(0, 1)

return {
    new      = new,
    i        = i,
    add     = add,
}

```

跟之前的其他的函数

```

    sub      = sub,
    mul      = mul,
    div      = div,
    tostring = tostring,
}

```

这种方式的优点在于，无须在每一个标识符前增加前缀 `M.` 或类似的东西。通过显式的导出表，我们能够以与在模块中相同的方式定义和使用导出和内部函数。这种方式的缺点在于，导出表位于模块最后而不是最前面（把前面的话当作简略文档的话更有用），而且由于必须把每个名字都写两遍，所以导出表有点冗余（这一缺点其实可能会变成优点，因为这允许函数在模块内和模块外具有不同的名称，不过程序很少会用到）。

不管怎样，无论怎样定义模块，用户都能用标准的方法使用模块：

```

local cpx = require "complex"
print(cpx.tostring(cpx.add(cpx.new(3,4), cpx.i)))
--> (3,5)

```

后续，我们会看到如何使用诸如元表和环境之类的高级 Lua 语言功能来编写模块。不过，除了发现由于失误而定义的全局变量时有一个技巧外，笔者在编写模块时都是用基本功能。

17.3 子模块和包

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

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

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

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

调用 `require "a.b"` 会尝试打开以下文件：

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

这种行为使得一个包中的所有模块能够放到一个目录中。例如，一个具有模块 `p`、`p.a` 和 `p.b` 的包对应的文件可以分别是 `p/init.lua`、`p/a.lua` 和 `p/b.lua`，目录 `p` 又位于其他合适的目录中。

`Lua` 语言使用的目录分隔符是编译时配置的，可以是任意的字符串（请记住，`Lua` 并不知道目录的存在）。例如，没有目录层次的系统可以使用下画线作为“目录分隔符”，因此调用 `require "a.b"` 会搜索文件 `a_b.lua`。

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

作为一种额外的机制，函数 `require` 在加载 `C` 语言编写的子模块时还有另外一个搜索器。当该函数找不到子模块对应的 `Lua` 文件或 `C` 文件时，它会再次搜索 `C` 文件所在的路径，不过这次将搜索包的名称。例如，如果一个程序要加载子模块 `a.b.c`，搜索器会搜索文件 `a`。如果找到了 `C` 标准库 `a`，那么函数 `require` 就会在该库中搜索对应的加载函数 `luaopen_a_b_c`。这种机制允许一个发行包将几个子模块组织为一个 `C` 标准库，每个子模块有各自的加载函数。

从 `Lua` 语言的视角看，同一个包中的子模块没有显式的关联。加载一个模块并不会自动加载它的任何子模块。同样，加载子模块也不会自动地加载其父模块。当然，只要包的实现者愿意，也可以创造这种关联。例如，一个特定的模块可能一开始就显式地加载它的一个或全部子模块。

17.4 练习

练习 17.1：将双端队列的实现（示例 14.2）重写为恰当的模块。

练习 17.2：将几何区域系统的实现（9.4 节）重写为恰当的模块。

练习 17.3：如果库搜索路径中包含固定的路径组成（即没有包含问号的组成部分）会发生什么？这一行为有什么用？

练习 17.4：编写一个同时搜索 `Lua` 文件和 `C` 标准库的搜索器。例如，搜索器使用的路径可能形如：

```
./?.lua;./?.so;/usr/lib/lua5.2/?.so;/usr/share/lua5.2/?.lua
```

（提示：使用函数 `package.searchpath` 寻找正确的文件，然后试着依次使用函数 `loadfile` 和函数 `package.loadlib` 加载该文件。）

如果想要在当前目录下运行一个脚本，但该目录没有包含该脚本的依赖库文件，那么将导致运行失败。

不正确的到正确的文件名映射关系，可能会导致程序无法正常运行。举个例子，假设在当前目录下有一个名为 `script.lua` 的脚本，其内容如下：

```
--script.lua
local file = assert(io.open("dofile.lua"))
dofile(file)
file:close()
```

如果将 `dofile.lua` 放在与 `script.lua` 同一目录下，那么运行 `script.lua` 时会输出以下结果：

```
script.lua:1: attempt to open 'dofile.lua' (a nil value or a closed file)
stack traceback:
        [C]: at 0x0000000000000000
```

从错误信息中可以看出，Lua 脚本尝试打开一个名为 `dofile.lua` 的文件，但该文件不存在。为了避免这种情况发生，可以在 `script.lua` 中使用 `package.path` 表示 `dofile.lua` 的位置，从而使得 Lua 脚本能够正确地加载外部模块。修改后的 `script.lua` 如下所示：

```
--script.lua
local file = assert(io.open(package.path.."/dofile.lua"))
dofile(file)
file:close()
```

通过将 `package.path` 表示为 `"/dofile.lua"`，Lua 脚本将优先在当前目录下查找 `dofile.lua` 文件，从而避免了因找不到文件而引起的错误。

需要注意的是，`package.path` 表示的路径是相对于脚本文件的，因此在修改 `script.lua` 时，需要将其放在与 `dofile.lua` 相同的目录下。

通过以上方法，可以有效地解决 Lua 脚本无法找到外部模块的问题。希望对大家有所帮助！

语言特性

第3部分



18

迭代器和泛型 for

我们已经在本书中的几个需求中使用过泛型 **for**，比如读取一个文件的所有行或遍历一个对象所有匹配的模式。然而，我们仍然不知道如何创建迭代器。在本章中，我们将学习这一部分内容，先从简单的迭代器入手，再学习如何利用泛型 **for** 的所有功能来编写各种各样的迭代器。

18.1 迭代器和闭包

迭代器 (*iterator*) 是一种可以让我们遍历一个集合中所有元素的代码结构。在 Lua 语言中，通常使用函数表示迭代器：每一次调用函数时，函数会返回集合中的“下一个”元素。一个典型的例子是 `io.read`，每次调用该函数时它都会返回标准输入中的下一行，在没有可以读取的行时返回 `nil`。

所有的迭代器都需要在连续的调用之间保存一些状态，这样才能知道当前迭代所处的位置及如何从当前位置步进到下一位置。对于函数 `io.read` 而言，C 语言会将状态保存在流的结构体中。对于我们自己的迭代器而言，闭包则为保存状态提供了一种良好的机制。请注意，一个闭包就是一个可以访问其自身的环境中一个或多个局部变量的函数。这些变量将连续调用过程中的值并将其保存在闭包中，从而使得闭包能够记住迭代所处的位置。当然，要创建一个新的闭包，我们还必须创建非局部变量。因此，一个闭包结构通常涉及两个函数：闭包本身和一个用于创建该闭包及其封装变量的工厂 (*factory*)。



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

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

在这个例子中，`values` 就是工厂。每当调用这个工厂时，它就会创建一个新的闭包（即迭代器本身）。这个闭包将它的状态保存在其外部的变量 `t` 和 `i` 中，这两个变量也是由 `values` 创建的。每次调用这个迭代器时，它就从列表 `t` 中返回下一个值。在遍历完最后一个元素后，迭代器返回 `nil`，表示迭代结束。

我们可以在一个 `while` 循环中使用这个迭代器：

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

不过，使用泛型 `for` 更简单。毕竟，泛型 `for` 正是为了这种迭代而设计的：

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

泛型 `for` 为一次迭代循环做了所有的记录工作：它在内部保存了迭代函数，因此不需要变量 `iter`；它在每次做新的迭代时都会再次调用迭代器，并在迭代器返回 `nil` 时结束循环（在下一节中，我们将会看到泛型 `for` 还完成了更多的工作）。

下面是一个更高级的示例，示例 18.1 展示了一个迭代器，它可以遍历来自标准输入的所有单词。

示例 18.1 遍历来自标准输入的所有单词的迭代器

```
function allwords ()
```



```
local line = io.read() -- 当前行
local pos = 1           -- 当前行的当前位置
return function ()      -- 迭代函数
    while line do        -- 当还有行时循环
        local w, e = string.match(line, "(%w+)", pos)
        if w then          -- 发现一个单词?
            pos = e         -- 下一个位置位于该单词后
            return w         -- 返回该单词
        else
            line = io.read() -- 没找到单词; 尝试下一行
            pos = 1           -- 从第一个位置重新开始
        end
    end
    return nil            -- 没有行了: 迭代结束
end
end
```

为了完成这样的遍历，我们需要保存两个值：当前行的内容（变量 `line`）及当前行的当前位置（变量 `pos`）。有了这些数据，我们就可以不断产生下一个单词。这个迭代函数的主要部分是调用函数 `string.match`，以当前位置作为起始在当前行中搜索一个单词。函数 `string.match` 使用模式`'%w+'`来匹配一个“单词”，也就是匹配一个或多个字母/数字字符。如果函数 `string.match` 找到了一个单词，它就捕获并返回这个单词及该单词之后的第一个字符的位置（一个空匹配），迭代函数则更新当前位置并返回该单词；否则，迭代函数读取新的一行，然后重复上述搜索过程。在所有的行都被读取完后，迭代函数返回 `nil` 以表示迭代结束。

尽管迭代器本身有点复杂，但 `allwords` 的使用还是很简明易懂的：

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

对于迭代器而言，一种常见的情况就是，编写迭代器可能不太容易，但使用迭代器却十分简单。这也不是一个大问题，因为使用 Lua 语言编程的最终用户一般不会去定义迭代器，而只会使用那些宿主应用已经提供的迭代器。



18.2 泛型 for 的语法

上述那些迭代器都有一个缺点，即需要为每个新的循环创建一个新的闭包。对于大多数情况而言，这或许不会有什么问题。例如，在之前的 allwords 迭代器中，创建一个闭包的开销相对于读取整个文件的开销而言几乎可以忽略不计。但是，在另外一些情况下，这样的开销可能会很可观。在这些情况中，我们可以通过使用泛型 **for** 自己保存迭代状态。在本节中，我们会详细说明泛型 **for** 提供的用来保存状态的机制。

泛型 **for** 在循环过程中在其内部保存了迭代函数。实际上，泛型 **for** 保存了三个值：一个迭代函数、一个不可变状态（*invariant state*）和一个控制变量（*control variable*）。下面让我们进行进一步学习。

泛型 **for** 的语法如下：

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

其中，*var-list* 是由一个或多个变量名组成的列表，以逗号分隔；*exp-list* 是一个或多个表达式组成的列表，同样以逗号分隔。通常，表达式列表只有一个元素，即一句对迭代器工厂的调用。例如，在如下代码中，变量列表是 *k, v*，表达式列表只有一个元素 *pairs(t)*：

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

我们把变量列表的第一个（或唯一的）变量称为控制变量（*control variable*），其值在循环过程中永远不会是 *nil*，因为当其值为 *nil* 时循环就结束了。

for 做的第一件事情是对 **in** 后面的表达式求值。这些表达式应该返回三个值供 **for** 保存：迭代函数、不可变状态和控制变量的初始值。类似于多重赋值，只有最后一个（或唯一的）表达式能够产生不止一个值；表达式列表的结果只会保留三个，多余的值会被丢弃，不足三个则以 *nil* 补齐。例如，在使用简单迭器时，工厂只会返回迭代函数，因此不可变状态和控制变量都是 *nil*。

在上述的初始化步骤完成后，**for** 使用不可变状态和控制变量为参数来调用迭代函数。从 **for** 代码结构的立足点来看，不可变状态根本没有意义。**for** 只是把从初始化步骤得到的状态值传递给所有迭代函数。然后，**for** 将迭代函数的返回值赋给变量列表中声明的变量。如果第一个返回值（赋给控制变量的值）为 *nil*，那么循环终止；否则，**for** 执行它的循环体并再次调用迭代函数，再不断地重复这个过程。



更确切地说，形如

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

这样的代码结构与下列代码等价：

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

因此，假设迭代函数为 f ，不可变状态为 s ，控制变量的初始值为 a_0 ，那么在循环中控制变量的值依次为 $a_1 = f(s, a_0), a_2 = f(s, a_1)$ ，依此类推，直至 a_i 为 nil。如果 **for** 还有其他变量，那么这些变量只是简单地在每次调用 f 后得到额外的返回值。

18.3 无状态迭代器

顾名思义，无状态迭代器（stateless iterator）就是一种自身不保存任何状态的迭代器。因此，可以在多个循环中使用同一个无状态迭代器，从而避免创建新闭包的开销。

正如刚刚所看到的，**for** 循环会以不可变状态和控制变量为参数调用迭代函数。一个无状态迭代器只根据这两个值来为迭代生成下一个元素。这类迭代器的一个典型例子就是 **ipairs**，它可以迭代一个序列中的所有元素：

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

迭代的状态由正在被遍历的表（一个不可变状态，它不会在循环中改变）及当前的索引值（控制变量）组成。**ipairs**（工厂）和迭代器都非常简单，我们可以在 Lua 语言中将其编写出来：



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

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

当调用 **for** 循环中的 `ipairs(t)` 时, `ipairs(t)` 会返回三个值, 即迭代函数 `iter`、不可变状态表 `t` 和控制变量的初始值 0。然后, Lua 语言调用 `iter(t, 0)`, 得到 1, `t[1]` (除非 `t[1]` 已经变成了 `nil`)。在第二次迭代中, Lua 语言调用 `iter(t, 1)`, 得到 2, `t[2]`, 依此类推, 直至得到第一个为 `nil` 的元素。

函数 `pairs` 与函数 `ipairs` 类似, 也用于遍历一个表中的所有元素。不同的是, 函数 `pairs` 的迭代函数是 Lua 语言中的一个基本函数 `next`:

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

在调用 `next(t, k)` 时, `k` 是表 `t` 的一个键, 该函数会以随机次序返回表中的下一个键及 `k` 对应的值 (作为第二个返回值)。调用 `next(t, nil)` 时, 返回表中的第一个键值对。当所有元素被遍历完时, 函数 `next` 返回 `nil`。

我们可以不调用 `pairs` 而直接使用 `next`:

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

请注意, **for** 循环会把表达式列表的结果调整为三个值, 因此上例中得到的是 `next`、`t` 和 `nil`, 这也正与 `pairs(t)` 的返回值完全一致。

关于无状态迭代器的另一个有趣的示例是遍历链表的迭代器 (链表在 Lua 语言中并不常见, 但有时也需要用到)。我们的第一反应可能是只把当前节点当作控制变量, 以便于迭



代函数能够返回下一个节点：

```
local function getnext (node)
    return node.next
end

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

但是，这种实现会跳过第一个节点。所以，我们需要使用如下的代码：

```
local function getnext (list, node)
    if not node then
        return list
    else
        return node.next
    end
end

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

这里的技巧是，除了将当前节点作为控制变量，还要将头节点作为不可变状态 (`traverse` 返回的第二个值)。第一次调用迭代函数 `getnext` 时，`node` 为 `nil`，因此函数返回 `list` 作为第一个节点。在后续的调用中，`node` 不再是 `nil`，所以迭代函数会像我们所期望的那样返回 `node.next`。

18.4 按顺序遍历表

一个常见的困惑发生在开发人员想要对表中的元素进行排序时。由于一个表中的元素没有顺序，所以如果想对这些元素排序，就不得不先把键值对拷贝到一个数组中，然后再对数组进行排序。

我们在第11章“小插曲：出现频率最高的单词”项目中已经看到过这个技巧的例子。这

里，让我们再举一个例子。假设我们要读取一个源文件，然后构造一个表来保存每个函数的名称及其声明所在的行数，形式如下：

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

现在，我们想按照字母顺序输出这些函数名。如果使用 pairs 遍历表，那么函数名会按照随机的顺序出现。由于这些函数名是表的键，所以我们无法直接对其进行排序。不过，如果我们把它们放到数组中，那么就可以对它们进行排序了。首先，我们必须创建一个包含函数名的数组，然后对其进行排序，再最终输出结果。

```
a = {}
for n in pairs(lines) do a[#a + 1] = n end
table.sort(a)
for _, n in ipairs(a) do print(n) end
```

有些人可能会困惑。毕竟，对于 Lua 语言来说，数组也没有顺序（毕竟它们是表）。但是我们知道如何数数！因此，当我们使用有序的索引访问数组时，就实现了有序。这正是应该总是使用 ipairs 而不是 pairs 来遍历数组的原因。第一个函数通过有序的键 1、2 等来实现有序，然而后者使用的则是天然的随机顺序（虽然大多数情况下顺序随机也无碍，但有时可能并非我们想要的）。

现在，我们已经准备好写一个按照键的顺序来遍历表的迭代器了：

```
function pairsByKeys (t, f)
    local a = {}
    for n in pairs(t) do      -- 创建一个包含所有键的表
        a[#a + 1] = n
    end
    table.sort(a, f)          -- 对列表排序
    local i = 0                -- 迭代变量
    return function ()         -- 迭代函数
        i = i + 1
        return a[i], t[a[i]]   -- 返回键和值
    end
end
```

```
end
```

工厂函数 `pairsByKeys` 首先把键放到一个数组中，然后对数组进行排序，最后返回迭代函数。在每一步中，迭代器都会按照数组 `a` 中的顺序返回原始表中的下一个键值对。可选的参数 `f` 允许指定一种其他的排序方式。

使用这个函数，可以很容易地解决开始时提出的按顺序遍历表的问题：

```
for name, line in pairsByKeys(lines) do
    print(name, line)
end
```

像通常的情况一样，所有的复杂性都被隐藏到了迭代器中。

18.5 迭代器的真实含义

“迭代器”这个名称多少有点误导性，这是因为迭代器并没有进行实际的迭代：真正的迭代是 `for` 循环完成的，迭代器只不过为每次的迭代提供连续的值。或许，称其为“生成器（generator）”更好，表示为迭代生成（*generate*）元素；不过，“迭代器”这个名字已在诸如 Java 等其他语言中被广泛使用了。

然而，还有一种创建迭代器的方式可以让迭代器进行实际的迭代操作。当使用这种迭代器时，就不再需要编写循环了。相反，只需要调用这个迭代器，并传入一个描述了在每次迭代时迭代器需要做什么的参数即可。更确切地说，迭代器接收一个函数作为参数，这个函数在循环的内部被调用，这种迭代器就被称为真正的迭代器（true iterator）。

举一个更具体的例子，让我们使用这种风格再次重写 `allwords` 迭代器：

```
function allwords (f)
    for line in io.lines() do
        for word in string.gmatch(line, "%w+") do
            f(word)      -- 调用函数
        end
    end
end
```

使用这个迭代器时，我们必须传入一个函数作为循环体。如果我们只想输出每个单词，那么简单地使用函数 `print` 即可：

```
器示例 allwords(print)
```

通常，我们可以使用一个匿名函数作为循环体。例如，以下的代码用于计算单词“hello”在输入文件中出现的次数：

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

同样的需求，如果采用之前的迭代器风格，差异也不是特别大：

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

真正的迭代器在老版本的 Lua 语言中曾非常流行，那时还没有 **for** 语句。真正的迭代器与生成器风格（generator-style）的迭代器相比怎么样呢？这两种风格都有大致相同的开销，即每次迭代都有一次函数调用。一方面，编写真正的迭代器比较容易（不过，我们可以使用24.3节中的方法使用协程来弥补）。另一方面，生成器风格的迭代器则更灵活。首先，生成器风格的迭代器允许两个或更多个并行的迭代（例如，考虑逐个单词比较两个文件的迭代器）。其次，生成器风格的迭代器允许在循环体中使用 **break** 和 **return** 语句。使用真正的迭代器，**return** 语句从匿名函数中返回而并非从进行迭代的函数中返回。基于这些原因，笔者一般更喜欢生成器风格的迭代器。

18.6 练习

练习 18.1：请编写一个迭代器 **fromto**，使得如下循环与数值型 **for** 等价：

```
for i in fromto(n, m) do
    body
end
```

你能否以无状态迭代器实现？

练习 18.2：向上一个练习中的迭代器增加一个步进的参数。你能否也用无状态迭代器实现？

练习 18.3：请编写一个迭代器 `uniquewords`，该迭代器返回指定文件中没有重复的所有单词（提示：基于示例 18.1 中 `allwords` 的代码，使用一个表来存储已经处理过的所有单词）。

练习 18.4：请编写一个迭代器，该迭代器可以返回指定字符串的所有非空子串。

练习 18.5：请编写一个真正的迭代器，该迭代器遍历指定集合的所有子集（该迭代器可以使用同一个表来保存所有的结果，只需要在每次迭代时改变表的内容即可，不需要为每个子集创建一个新表）。

19

小插曲：马尔可夫链算法

下一个完整的程序是一个马尔可夫链（Markov chain）算法的实现，该算法由 Kernighan 和 Pike 在他们的书 *The Practice of Programming*（Addison-Wesley 出版社 1999 年出版）中进行了描述。

马尔可夫链算法根据哪个单词能出现在基础文本中由 n 个前序单词组成的序列之后，来生成伪随机（pseudo-random）文本。对于本例中的实现，我们假定 n 为 2。

程序的第一部分读取原始文本并创建一个表，该表的键为每两个单词组成的前缀，值为紧跟这个前缀的单词所组成的列表。当这个表构建好后，程序就利用它来生成随机文本，随机文本中每个单词出现在它之前两个单词后的概率与其出现在基础文本中相同两个前序单词后的概率相同。最终，我们会得到一串相对比较随机的文本。例如，以本书的英文原版作为基础文本，那么该程序的输出形如 “*Constructors can also traverse a table constructor, then the parentheses in the following line does the whole file in a field n to store the contents of each function, but to show its only argument. If you want to find the maximum element in an array can return both the maximum value and continues showing the prompt and running the code. The following words are reserved and cannot be used to convert between degrees and radians.*”

要将由两个单词组成的前缀作为表的键，需要使用空格来连接两个单词：

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

我们使用字符串 NOWORD（换行符）初始化前缀单词及标记文本的结尾。例如，对于文本 "the more we try the more we do" 而言，构造出的表如下：

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

程序将表保存在变量 statetab 中。如果要向表中的某个前缀所对应的列表中插入一个新单词，可以使用如下的函数：

```
function insert (prefix, value)  
    local list = statetab[prefix]  
    if list == nil then  
        statetab[prefix] = {value}  
    else  
        list[#list + 1] = value  
    end  
end
```

该函数首先检查某前缀是否已经有了对应的列表，如果没有，则以新值来创建一个新列表；否则，就将新值添加到现有列表的末尾。

为了构造表 statetab，我们使用两个变量 w1 和 w2 来记录最后读取的两个单词。我们使用 18.1 节中的 allwords 迭代器读取单词，只不过修改了其中“单词”的定义以便将可选的诸如逗号和句号等标点符号包括在内（参见示例 19.1）。对于新读取的每一个单词，把它添加到与 w1-w2 相关联的列表中，然后更新 w1 和 w2。

在构造完表后，程序便开始生成具有 MAXGEN 个单词的文本。首先，程序重新初始化变量 w1 和 w2。然后，对于每个前缀，程序从其对应的单词列表中随机地选出一个单词，输出这个单词，并更新 w1 和 w2。示例 19.1 和示例 19.2 给出了完整的程序。

示例 19.1 马尔可夫链程序的辅助定义

```
function allwords ()
```

```

local line = io.read()      -- 当前行
local pos = 1                -- 当前行的当前位置
return function ()          -- 迭代函数
    while line do            -- 当还有行时循环
        local w, e = string.match(line, "(%w+[,;.:]?)(()", pos)
        if w then
            pos = e
            return w
        else
            line = io.read()      -- 没找到单词；尝试下一行
            pos = 1                -- 从第一个位置重新开始
        end
    end
    return nil                -- 没有行了：迭代结束
end

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

local statetab = {}

function insert (prefix, value)
    local list = statetab[prefix]
    if list == nil then
        statetab[prefix] = {value}
    else
        list[#list + 1] = value
    end
end

```

示例 19.2 马尔可夫链程序

```
local MAXGEN = 200
```

```

local NOWORD = "\n"

-- 创建表
local w1, w2 = NOWORD, NOWORD
for nextword in allwords() do
    insert(prefix(w1, w2), nextword)
    w1 = w2; w2 = nextword;
end
insert(prefix(w1, w2), NOWORD)

-- 生成本文
w1 = NOWORD; w2 = NOWORD      -- 重新初始化
for i = 1, MAXGEN do
    local list = statetab[prefix(w1, w2)]
    -- 从列表中随机选出一个元素
    local r = math.random(#list)
    local nextword = list[r]
    if nextword == NOWORD then return end
    io.write(nextword, " ")
    w1 = w2; w2 = nextword
end

```

19.1 练习

练习 19.1：使马尔可夫链算法更加通用，以支持任意长度的前缀单词序列。

20

元表和元方法

通常，Lua 语言中的每种类型的值都有一套可预见的操作集合。例如，我们可以将数字相加，可以连接字符串，还可以在表中插入键值对等。但是，我们无法将两个表相加，无法对函数作比较，也无法调用一个字符串，除非使用元表。

元表可以修改一个值在面对一个未知操作时的行为。例如，假设 `a` 和 `b` 都是表，那么可以通过元表定义 Lua 语言如何计算表达式 `a + b`。当 Lua 语言试图将两个表相加时，它会先检查两者之一是否有元表 (*metatable*) 且该元表中是否有 `_add` 字段。如果 Lua 语言找到了该字段，就调用该字段对应的值，即所谓的元方法 (*metamethod*) (是一个函数)，在本例中就是用于计算表的和的函数。

可以认为，元表是面向对象领域中的受限制类。像类一样，元表定义的是实例的行为。不过，由于元表只能给出预先定义的操作集合的行为，所以元表比类更受限；同时，元表也不支持继承。不过尽管如此，我们还是会在第21章中看到如何基于元表构建一个相对完整的类系统。

Lua 语言中的每一个值都可以有元表。每一个表和用户数据类型都具有各自独立的元表，而其他类型的值则共享对应类型所属的同一个元表。Lua 语言在创建新表时不带元表：

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

可以使用函数 `setmetatable` 来设置或修改任意表的元表：

```
t1 = {}
setmetatable(t, t1)
print(getmetatable(t) == t1) --> true
```

在 Lua 语言中，我们只能为表设置元表；如果要为其他类型的值设置元表，则必须通过 C 代码或调试库完成（该限制存在的主要原因是防止过度使用对某种类型的所有值生效的元表。Lua 语言老版本中的经验表明，这样的全局设置经常导致不可重用的代码）。字符串标准库为所有的字符串都设置了同一个元表，而其他类型在默认情况下都没有元表：

```
print(getmetatable("hi"))          --> table: 0x80772e0
print(getmetatable("xuxu"))        --> table: 0x80772e0
print(getmetatable(10))            --> nil
print(getmetatable(print))         --> nil
```

一个表可以成为任意值的元表；一组相关的表也可以共享一个描述了它们共同行为的通用元表；一个表还可以成为它自己的元表，用于描述其自身特有的行为。总之，任何配置都是合法的。

20.1 算术运算相关的元方法

在本节中，我们将介绍一个解释元表基础的示例。假设有一个用表来表示集合的模块，该模块还有一些用来计算集合并集和交集等的函数，可以参见示例 20.1。

示例 20.1 一个用于集合的简单模块

```
local Set = {}

-- 使用指定的列表创建一个新的集合
function Set.new (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
```

```

for k in pairs(a) do res[k] = true end
for k in pairs(b) do res[k] = true end
return res
end

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

-- 将集合表示为字符串
function Set.tostring (set)
local l = {}      -- 保存集合中所有元素的列表
for e in pairs(set) do
  l[#l + 1] = tostring(e)
end
return "{" .. table.concat(l, ", ") .. "}"
end

return Set

```

现在，假设想使用加法操作符来计算两个集合的并集，那么可以让所有表示集合的表共享一个元表。这个元表中定义了这些表应该如何执行加法操作。首先，我们创建一个普通的表，这个表被用作集合的元表：

```
local mt = {}      -- 集合的元表
```

然后，修改用于创建集合的函数 `Set.new`。在新版本中只多了一行，即将 `mt` 设置为函数 `Set.new` 所创建的表的元表：

```

function Set.new (l)    -- 第二个版本
local set = {}
setmetatable(set, mt)
for _, v in ipairs(l) do set[v] = true end
return set

```

```

    return set
end

```

在此之后，所有由 Set.new 创建的集合都具有了一个相同的元表：

```

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1))          --> table: 0x00672B60
print(getmetatable(s2))          --> table: 0x00672B60

```

最后，向元表中加入元方法 (*metamethod*) `__add`，也就是用于描述如何完成加法的字段：

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

此后，只要 Lua 语言试图将两个集合相加，它就会调用函数 Set.union，并将两个操作数作为参数传入。

通过元方法，我们就可以使用加法运算符来计算集合的并集了：

```

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

```

类似地，还可以使用乘法运算符来计算集合的交集：

```

mt.__mul = Set.intersection

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

```

每种算术运算符都有一个对应的元方法。除了加法和乘法外，还有减法 (`__sub`)、除法 (`__div`)、floor 除法 (`__idiv`)、负数 (`__unm`)、取模 (`__mod`) 和幂运算 (`__pow`)。类似地，位操作也有元方法：按位与 (`__band`)、按位或 (`__bor`)、按位异或 (`__bxor`)、按位取反 (`__bnot`)、向左移位 (`__shl`) 和向右移位 (`__shr`)。我们还可以使用字段 `__concat` 来定义连接运算符的行为。

当我们把两个集合相加时，使用哪个元表是确定的。然而，当一个表达式中混合了两种具有不同元表的值时，例如：

```

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

```

Lua 语言会按照如下步骤来查找元方法：如果第一个值有元表且元表中存在所需的元方法，那么 Lua 语言就使用这个元方法，与第二个值无关；如果第二个值有元表且元表中存在所需

的元方法，Lua 语言就使用这个元方法；否则，Lua 语言就抛出异常。因此，上例会调用 `Set.union`，而表达式 `10+s` 和 `"hello"+s` 同理（由于数值和字符串都没有元方法 `__add`）。

Lua 语言不关心这些混合类型，但我们在实现中需要关心混合类型。如果我们执行了 `s = s + 8`，那么在 `Set.union` 内部就会发生错误：

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

如果想要得到更明确的错误信息，则必须在试图进行操作前显式地检查操作数的类型，例如：

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

```

请注意，函数 `error` 的第二个参数（上例中的 2）说明了出错的原因位于调用该函数的代码中^①。

20.2 关系运算相关的元方法

元表还允许我们指定关系运算符的含义，其中的元方法包括等于 (`__eq`)、小于 (`__lt`) 和小于等于 (`__le`)。其他三个关系运算符没有单独的元方法，Lua 语言会将 `a ~= b` 转换为 `not (a == b)`，`a > b` 转换为 `b < a`，`a >= b` 转换为 `b <= a`。

在 Lua 语言的老版本中，Lua 语言会通过将 `a <= b` 转换为 `not (b < a)` 来把所有的关系运算符转化为一个关系运算符。不过，这种转化在遇到部分有序 (*partial order*) 时就会不正确。所谓部分有序是指，并非所有类型的元素都能够被正确地排序。例如，由于 *Not a Number* (*NaN*) 的存在，大多数计算机中的浮点数就不是完全可以排序的。根据 IEEE 754 标准，*NaN* 代表未定义的值，例如 *0/0* 的结果就是 *NaN*。标准规定任何涉及 *NaN* 的比较都应返回假，这就意味着 *NaN* `<= x` 永远为假，`x < NaN` 也为假。因此，在这种情况下，`a <= b` 到 `not (b < a)` 的转化也就不合法了。

在集合的示例中，我们也面临类似的问题。`<=` 显而易见且有用的含义是集合包含：`a <= b` 通常意味着 `a` 是 `b` 的一个子集。然而，根据部分有序的定义，`a <= b` 和 `b < a` 可能同时为假。因此，我们就必须实现 `__le`（小于等于，子集关系）和 `__lt`（小于，真子集关系）：

^①译者注：即错误的级别，参见第二部分最后一章的相关内容。

```

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

mt.__lt = function (a, b)    -- 真子集
    return a <= b and not (b <= a)
end

```

最后，我们还可以通过集合包含来定义集合相等：

```

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

```

有了这些定义后，我们就可以比较集合了：

```

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

```

相等比较有一些限制。如果两个对象的类型不同，那么相等比较操作不会调用任何元方法而直接返回 **false**。因此，不管元方法如何，集合永远不等于数字。

20.3 库定义相关的元方法

到目前为止，我们见过的所有元方法针对的都是核心 Lua 语言。Lua 语言虚拟机（virtual machine）会检测一个操作中涉及的值是否有存在对应元方法的元表。不过，由于元表是一个普通的表，所以任何人都可以使用它们。因此，程序库在元表中定义和使用它们自己的字段也是一种常见的实践。

函数 `tostring` 就是一个典型的例子。正如我们此前所看到的，函数 `tostring` 能将表表示为一种简单的文本格式：

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

函数 `print` 总是调用 `tostring` 来进行格式化输出。不过，当对值进行格式化时，函数 `tostring` 会首先检查值是否有一个元方法 `__tostring`。如果有，函数 `tostring` 就调用这个元方法来完成工作，将对象作为参数传给该函数，然后把元方法的返回值作为函数 `tostring` 的返回值。

在之前集合的示例中，我们已经定义了一个将集合表示为字符串的函数。因此，只需要在元表中设置 `__tostring` 字段：

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

之后，当以一个集合作为参数调用函数 `print` 时，`print` 就会调用函数 `tostring`，`tostring` 又会调用 `Set.tostring`：

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

函数 `setmetatable` 和 `getmetatable` 也用到了元方法，用于保护元表。假设想要保护我们的集合，就要使用户既不能看到也不能修改集合的元表。如果在元表中设置 `__metatable` 字段，那么 `getmetatable` 会返回这个字段的值，而 `setmetatable` 则会引发一个错误：

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

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

从 Lua 5.2 开始，函数 `pairs` 也有了对应的元方法，因此我们可以修改表被遍历的方式和为非表的对象增加遍历行为。当一个对象拥有 `__pairs` 元方法时，`pairs` 会调用这个元方法来完成遍历。

20.4 表相关的元方法

算术运算符、位运算符和关系运算符的元方法都定义了各种错误情况的行为，但它们都没有改变语言的正常行为。Lua 语言还提供了一种改变表在两种正常情况下的行为的方式，即访问和修改表中不存在的字段。

20.4.1 __index 元方法

正如我们此前所看到的，当访问一个表中不存在的字段时会得到 nil。这是正确的，但不是完整的真相。实际上，这些访问会引发解释器查找一个名为 `__index` 的元方法。如果没有这个元方法，那么像一般情况下一样，结果就是 nil；否则，则由这个元方法来提供最终结果。

下面介绍一个关于继承的原型示例。假设我们要创建几个表来描述窗口，每个表中必须描述窗口的一些参数，例如位置、大小及主题颜色等。所有的这些参数都有默认值，因此我们希望在创建窗口对象时只需要给出那些不同于默认值的参数即可。第一种方法是使用一个构造器来填充不存在的字段，第二种方法是让新窗口从一个原型窗口继承所有不存在的字段。首先，我们声明一个原型：

```
-- 创建具有默认值的原型
prototype = {x = 0, y = 0, width = 100, height = 100}
```

然后，声明一个构造函数，让构造函数创建共享同一个元表的新窗口：

```
local mt = {}      -- 创建一个元表
-- 声明构造函数
function new(o)
    setmetatable(o, mt)
    return o
end
```

现在，我们来定义元方法 `__index`：

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

在这段代码后，创建一个新窗口，并查询一个创建时没有指定的字段：

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

Lua 语言会发现 w 中没有对应的字段 "width"，但却有一个带有 `__index` 元方法的元表。因此，Lua 语言会以 w (表) 和 "width" (不存在的键) 为参数来调用这个元方法。元方法随后会用这个键来检索原型并返回结果。

在 Lua 语言中，使用元方法 `__index` 来实现继承是很普遍的方法。虽然被叫作方法，但元方法 `__index` 不一定必须是一个函数，它还可以是一个表。当元方法是一个函数时，Lua 语言会以表和不存在的键为参数调用该函数，正如我们刚刚所看到的。当元方法是一个表时，Lua 语言就访问这个表。因此，在我们此前的示例中，可以把 `__index` 简单地声明为如下样式：

```
mt.__index = prototype
```

这样，当 Lua 语言查找元表的 `__index` 字段时，会发现字段的值是表 `prototype`。因此，Lua 语言就会在这个表中继续查找，即等价地执行 `prototype["width"]`，并得到预期的结果。

将一个表用作 `__index` 元方法为实现单继承提供了一种简单快捷的方法。虽然将函数用作元方法开销更昂贵，但函数却更加灵活：我们可以通过函数来实现多继承、缓存及其他一些变体。我们将会在第21章中学习面向对象编程时讨论这些形式的继承。

如果我们希望在访问一个表时不调用 `__index` 元方法，那么可以使用函数 `rawget`。调用 `rawget(t, i)` 会对表 t 进行原始 (raw) 的访问，即在不考虑元表的情况下对表进行简单的访问。进行一次原始访问并不会加快代码的执行（一次函数调用的开销就会抹杀用户所做的这些努力），但是，我们后续会看到，有时确实会用到原始访问。

20.4.2 `__newindex` 元方法

元方法 `__newindex` 与 `__index` 类似，不同之处在于前者用于表的更新而后者用于表的查询。当对一个表中不存在的索引赋值时，解释器就会查找 `__newindex` 元方法：如果这个元方法存在，那么解释器就调用它而不执行赋值。像元方法 `__index` 一样，如果这个元方法是一个表，解释器就在此表中执行赋值，而不是在原始的表中进行赋值。此外，还有一个原始函数允许我们绕过元方法：调用 `rawset(t, k, v)` 来等价于 `t[k]=v`，但不涉及任何元方法。

组合使用元方法 `__index` 和 `__newindex` 可以实现 Lua 语言中的一些强大的结构，例如只读的表、具有默认值的表和面向对象编程中的继承。在本章中，我们会介绍其中的一些应用，面向对象编程会在后续单独的章节中进行介绍。

20.4.3 具有默认值的表

一个普通表中所有字段的默认值都是 `nil`。通过元表，可以很容易地修改这个默认值：

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

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

在调用 `setDefault` 后，任何对表 `tab` 中不存在字段的访问都将调用它的 `__index` 元方法，而这个元方法会返回零（这个元方法中的值是 `d`）。

函数 `setDefault` 为所有需要默认值的表创建了一个新的闭包和一个新的元表。如果我们有很多需要默认值的表，那么开销会比较大。然而，由于具有默认值 `d` 的元表是与元方法关联在一起的，所以我们不能把同一个元表用于具有不同默认值的表。为了能够使所有的表都使用同一个元表，可以使用一个额外的字段将每个表的默认值存放到表自身中。如果不担心命名冲突的话，我们可以使用形如“`__`”这样的键作为额外的字段：

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

请注意，这里我们只在 `setDefault` 外创建了一次元表 `mt` 及对应的元方法。

如果担心命名冲突，要确保这个特殊键的唯一性也很容易，只需要创建一个新的排除表，然后将它作为键即可：

```

local key = {}      -- 唯一的键
local mt = {__index = function (t) return t[key] end}
function setDefault (t, d)
    t[key] = d
    setmetatable(t, mt)
end

```

还有一种方法可以将每个表与其默认值关联起来，称为对偶表示 (*dual representation*)，即使用一个独立的表，该表的键为各种表，值为这些表的默认值。不过，为了正确地实现这种做法，我们还需要一种特殊的表，称为弱引用表 (*weak table*)。在这里，我们暂时不会使用弱引用表，而在第23章中再讨论这个话题。

另一种为具有相同默认值的表复用同一个元表的方式是记忆 (*memorize*) 元表。不过，这也需要用到弱引用表，我们会在第23章中继续学习。

20.4.4 跟踪对表的访问

假设我们要跟踪对某个表的所有访问。由于 `__index` 和 `__newindex` 元方法都是在表中的索引不存在时才有用，因此，捕获对一个表所有访问的唯一方式是保持表是空的。如果要监控对一个表的所有访问，那么需要为真正的表创建一个代理 (*proxy*)。这个代理是一个空的表，具有用于跟踪所有访问并将访问重定向到原来的表的合理元方法。示例 20.2 使用这种思想进行了实现。

示例 20.2 跟踪对表的访问

```

function track (t)
    local proxy = {}          -- 't'的代理表
    -- 为代理创建元表
    local mt = {
        __index = function (_, k)
            print("*access to element " .. tostring(k))
            return t[k]    -- 访问原来的表
        end,
        __newindex = function (_, k, v)

```



```

print("*update of element " .. tostring(k) ..
      " to " .. tostring(v))
t[k] = v -- 更新原来的表
end,

__pairs = function ()
    return function (_, k) -- 迭代函数
        local nextkey, nextvalue = next(t, k)
        if nextkey == nil then -- 避免最后一个值
            print("*traversing element " .. tostring(nextkey))
        end
        return nextkey, nextvalue
    end
end,
end,

__len = function () return #t end
}

setmetatable(proxy, mt)
return proxy
end

```

以下展示了上述代码的用法：

```

> t = {}          -- 任意一个表
> t = track(t)
> t[2] = "hello"
--> *update of element 2 to hello
> print(t[2])
--> *access to element 2
--> hello

```

元方法 `__index` 和 `__newindex` 按照我们设计的规则跟踪每一个访问并将其重定向到原来的表中。元方法 `__pairs` 使得我们能够像遍历原来的表一样遍历代理，从而跟踪所有的访问。最后，元方法 `__len` 通过代理实现了长度操作符：



```
t = track({10, 20})  
print(#t)           --> 2  
for k, v in pairs(t) do print(k, v) end  
--> *traversing element 1  
--> 1  10  
--> *traversing element 2  
--> 2  20
```

如果想要同时监控几个表，并不需要为每个表创建不同的元表。相反，只要以某种形式将每个代理与其原始表映射起来，并且让所有的代理共享一个公共的元表即可。这个问题与上节所讨论的把表与其默认值关联起来的问题类似，因此可以采用相同的解决方式。例如，可以把原来的表保存在代理表的一个特殊的字段中，或者使用一个对偶表示建立代理与相应表的映射。

20.4.5 只读的表

使用代理的概念可以很容易地实现只读的表，需要做的只是跟踪对表的更新操作并抛出异常即可。对于元方法 `__index`，由于我们不需要跟踪查询，所以可以直接使用原来的表来代替函数。这样做比把所有的查询重定向到原来的表上更简单也更有效率。不过，这种做法要求为每个只读代理创建一个新的元表，其中 `__index` 元方法指向原来的表：

```
function readOnly (t)
    local proxy = {}
    local mt = { -- 创建元表
        __index = t,
        __newindex = function (t, k, v)
            error("attempt to update a read-only table", 2)
        end
    }
    setmetatable(proxy, mt)
    return proxy
end
```

作为示例，我们可以创建一个表示星期的只读表：

```
days = readOnly["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```



```

    "Thursday", "Friday", "Saturday"} --> Sunday
print(days[1]) --> Sunday
days[2] = "Noday"
--> stdin:1: attempt to update a read-only table

```

20.5 练习

练习 20.1：请定义一个元方法 `__sub`，该元方法用于计算两个集合的差集（集合 $a - b$ 是位于集合 a 但不位于集合 b 中的元素）。

练习 20.2：请定义一个元方法 `__len`，该元方法用于实现使用 `#s` 计算集合 s 中的元素个数。

练习 20.3：实现只读表的另一种方式是将一个函数用作 `__index` 元方法。这种方式使得访问的开销更大，但是创建只读表的开销更小（因为所有的只读表能够共享同一个元表）。请使用这种方式重写函数 `readOnly`。

练习 20.4：代理表可以表示除表外的其他类型的对象。请编写一个函数 `fileAsArray`，该函数以一个文件名为参数，返回值为对应文件的代理，当执行 `t = fileAsArray("myFile")` 后，访问 `t[i]` 返回指定文件的第 i 个字节，而对 `t[i]` 的赋值更新第 i 个字节。

练习 20.5：扩展之前的示例，使得我们能够使用 `pairs(t)` 遍历一个文件中的所有字节，并使用 `#t` 来获得文件的大小。



21

面向对象 (Object-Oriented) 编程

从很多意义上讲，Lua 语言中的一张表就是一个对象。首先，表与对象一样，可以拥有状态。其次，表与对象一样，拥有一个与其值无关的标识 (*self*^①)；特别地，两个具有相同值的对象（表）是两个不同的对象，而一个对象可以具有多个不同的值；最后，表与对象一样，具有与创建者和被创建位置无关的生命周期。

对象有其自己的操作。表也可以有自己的操作，例如：

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

上面的代码创建了一个新函数，并将该函数存入 `Account` 对象的 `withdraw` 字段。然后，我们就可以进行如下的调用：

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

这种函数差不多就是所谓的方法 (*method*) 了。不过，在函数中使用全局名称 `Account` 是一个非常糟糕的编程习惯。首先，这个函数只能针对特定对象工作。其次，即使针对特定

^①译者注：类似于 `this` 指针。



的对象，这个函数也只有在对象保存在特定的全局变量中时才能工作。如果我们改变了对象的名称，`withdraw` 就不能工作了：

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

这种行为违反对象拥有独立生命周期的原则。

另一种更加有原则的方法是对操作的接受者 (*receiver*) 进行操作。因此，我们的方法需要一个额外的参数来表示该接受者，这个参数通常被称为 *self* 或 *this*：

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

此时，当我们调用该方法时，必须指定要操作的对象：

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

通过使用参数 *self*，可以对多个对象调用相同的方法：

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

使用参数 *self* 是所有面向对象语言的核心点。大多数面向对象语言都向程序员隐藏了这个机制，从而使得程序员不必显式地声明这个参数（虽然程序员仍然可以在方法内使用 *self* 或者 *this*）。Lua 语言同样可以使用冒号操作符 (*colon operator*) 隐藏该参数。使用冒号操作符，我们可以将上例重写为 `a2:withdraw(260.00)`：

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

冒号的作用是在一个方法调用中增加一个额外的实参，或在方法的定义中增加一个额外的隐藏形参。冒号只是一种语法机制，虽然很便利，但没有引入任何新的东西。我们可以使用点分语法来定义一个函数，然后用冒号语法调用它，反之亦然，只要能够正确地处理好额外的参数即可：



```

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

function Account:deposit (v)
    self.balance = self.balance + v
end

Account.deposit(Account, 200.00)
Account:withdraw(100.00)

```

21.1 类 (Class)

截至目前，我们的对象具有了标识、状态和对状态进行的操作，但还缺乏类体系、继承和私有性。让我们先来解决第一个问题，即应该如何创建多个具有类似行为的对象。更具体地说，我们应该如何创建多个银行账户呢？

大多数面向对象语言提供了类的概念，类在对象的创建中扮演了模子（mold）的作用。在这些语言中，每个对象都是某个特定类的实例（instance）。Lua 语言中没有类的概念；虽然元表的概念在某种程度上与类的概念相似，但是把元表当作类使用在后续会比较麻烦。相反，我们可以参考基于原型的语言（prototype-based language）中的一些做法来在 Lua 语言中模拟类，例如 Self 语言（JavaScript 采用的也是这种方式）。在这些语言中，对象不属于类。相反，每个对象可以有一个原型（prototype）。原型也是一种普通的对象，当对象（类的实例）遇到一个未知操作时会首先在原型中查找。要在这种语言中表示一个类，我们只需要创建一个专门被用作其他对象（类的实例）的原型对象即可。类和原型都是一种组织多个对象间共享行为的方式。

在 Lua 语言中，我们可以使用 20.4.1 节中所述的继承的思想来实现原型。更准确地说，如果有两个对象 A 和 B，要让 B 成为 A 的一个原型，只需要：

```
setmetatable(A, {__index = B})
```

在此之后，A 就会在 B 中查找所有它没有的操作。如果把 B 看作对象 A 的类，则只不过是术语上的一个变化。



让我们回到之前银行账号的示例。为了创建其他与 Account 行为类似的账号，我们可以使用 `__index` 元方法让这些新对象从 Account 中继承这些操作。

```
local mt = {__index = Account}

function Account.new (o)
    o = o or {}      -- 如果用户没有提供则创建一个新的表
    setmetatable(o, mt)
    return o
end
```

在这段代码执行后，当我们创建一个新账户并调用新账户的一个方法时会发生什么呢？

```
a = Account.new{balance = 0}
a:deposit(100.00)
```

当我们创建一个新账户 `a` 时，`a` 会将 `mt` 作为其元表。当调用 `a:deposit(100.00)` 时，实际上调用的是 `a.deposit(a, 100.00)`，冒号只不过是一个语法糖。不过，Lua 语言无法在表 `a` 中找到字段"deposit"，所以它会在元表的 `__index` 中搜索。此时的情况大致如下：

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

`a` 的元表是 `mt`，而 `mt.__index` 是 `Account`。因此，上述表达式等价于：

```
Account.deposit(a, 100.00)
```

即，Lua 语言调用了原来的 `deposit` 函数，传入了 `a` 作为 `self` 参数。因此，新账户 `a` 从 `Account` 继承了函数 `deposit`。同样，它还从 `Account` 继承了所有的字段。

对于这种模式，我们可以进行两个小改进。第一种改进是，不创建扮演元表角色的新表而是把表 `Account` 直接用作元表。第二种改进是，对 `new` 方法也使用冒号语法。加入了这两个改动后，方法 `new` 会变成：

```
function Account:new (o)
    o = o or {}
    self.__index = self
    setmetatable(o, self)
    return o
end
```



现在，当我们调用 `Account:new()` 时，隐藏的参数 `self` 得到的实参是 `Account`，`Account.__index` 等于 `Account`，并且 `Account` 被用作新对象的元表。可能看上去第二种修改（冒号语法）并没有得到大大的好处，但实际上当我们在下一节中引入类继承的时候，使用 `self` 的优点就会很明显了。

继承不仅可以作用于方法，还可以作用于其他在新账户中没有的字段。因此，一个类不仅可以提供方法，还可以为实例中的字段提供常量和默认值。请注意，在第一版 `Account` 的定义中，有一个 `balance` 字段的值是 0。因此，如果在创建新账户时没有提供初始的余额，那么余额就会继承这个默认值：

```
b = Account:new()
print(b.balance)    --> 0
```

当在 `b` 上调用 `deposit` 方法时，由于 `self` 就是 `b`，所以等价于：

```
b.balance = b.balance + v
```

表达式 `b.balance` 求值后等于零，且该方法给 `b.balance` 赋了初始的金额。由于此时 `b` 有了它自己的 `balance` 字段，因此后续对 `b.balance` 的访问就不会再涉及元方法了。

21.2 继承 (Inheritance)

由于类也是对象，因此它们也可以从其他类获得方法。这种行为使得继承（即常见的面向对象的定义）可以很容易地在 Lua 语言中实现。

假设有一个类似于 `Account` 的基类，参见示例 21.1。

示例 21.1 `Account` 类

```
Account = {balance = 0}

function Account:new(o)
  o = o or {}
  self.__index = self
  setmetatable(o, self)
  return o
end
```



```

function Account:deposit (v)
    self.balance = self.balance + v
end

function Account:withdraw (v)
    if v > self.balance then error"insufficient funds" end
    self.balance = self.balance - v
end

```

若想从这个类派生一个子类 `SpecialAccount` 以允许客户透支，那么可以先创建一个从基类继承了所有操作的空类：

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

直到现在，`SpecialAccount` 还只是 `Account` 的一个实例。下面让我们来见证奇迹：

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` 就像继承其他方法一样从 `Account` 继承了 `new`。不过，现在执行 `new` 时，它的 `self` 参数指向的是 `SpecialAccount`。因此，`s` 的元表会是 `SpecialAccount`，其中字段 `_index` 的值也是 `SpecialAccount`。因此，`s` 继承自 `SpecialAccount`，而 `SpecialAccount` 又继承自 `Account`。之后，当执行 `s:deposit(100.00)` 时，Lua 语言在 `s` 中找不到 `deposit` 字段，就会查找 `SpecialAccount`，仍找不到 `deposit` 字段，就查找 `Account` 并最终会在 `Account` 中找到 `deposit` 的最初实现。

`SpecialAccount` 之所以特殊是因为我们可以重新定义从基类继承的任意方法，只需要编写一个新方法即可：

```

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

function SpecialAccount:getLimit ()
    return self.limit or 0
end

```



现在，当调用 `s:withdraw(200.00)` 时，因为 Lua 语言会在 `SpecialAccount` 中先找到新的 `withdraw` 方法，所以不会再从 `Account` 中查找。由于 `s.limit` 为 `1000.00`（我们创建 `s` 时设置了这个值），所以程序会执行取款并使 `s` 变成负的余额。

Lua 语言中的对象有一个有趣的特性，就是无须为了指定一种新行为而创建一个新类。如果只有单个对象需要某种特殊的行为，那么我们可以直接在该对象中实现这个行为。例如，假设账户 `s` 表示一个特殊的客户，这个客户的透支额度总是其余额的 10%，那么可以只修改这个账户：

```
function s:getLimit ()
    return self.balance * 0.10
end
```

在这段代码后，调用 `s:withdraw(200.00)` 还是会执行 `SpecialAccount` 的 `withdraw` 方法，但当 `withdraw` 调用 `self:getLimit` 时，调用的是上述的定义。

21.3 多重继承 (Multiple Inheritance)

由于 Lua 语言中的对象不是基本类型，因此在 Lua 语言中进行面向对象编程时有几种方式。上面所见到的是一种使用 `__index` 元方法的做法，也可能是在简易、性能和灵活性方面最均衡的做法。不过尽管如此，还有一些其他的实现对某些特殊的情况可能更加合适。在此，我们会看到允许在 Lua 语言中实现多重继承的另一种实现。

这种实现的关键在于把一个函数用作 `__index` 元方法。请注意，当一个表的元表中的 `__index` 字段为一个函数时，当 Lua 不能在原来的表中找到一个键时就会调用这个函数。基于这一点，就可以让 `__index` 元方法在其他期望的任意数量的父类中查找缺失的键。

多重继承意味着一个类可以具有多个超类。因此，我们不应该使用一个（超）类中的方法来创建子类，而是应该定义一个独立的函数 `createClass` 来创建子类。函数 `createClass` 的参数为新类的所有超类，参见示例 21.2。该函数创建一个表来表示新类，然后设置新类元表中的元方法 `__index`，由元方法实现多重继承。虽然是多重继承，但每个实例仍属于单个类，并在其中查找所有的方法。因此，类和超类之间的关系不同于类和实例之间的关系。尤其是，一个类不能同时成为其实例和子类的元表。在示例 21.2 中，我们将类保存为其实例的元表，并创建了另一个表作为类的元表。



示例 21.2 一种多重继承的实现

```
-- 在表'plist'的列表中查找'k'
local function search (k, plist)
    for i = 1, #plist do
        local v = plist[i][k]      -- 尝试第'i'个超类
        if v then return v end
    end
end

function createClass ...
    local c = {}                  -- 新类
    local parents = {...}         -- 父类列表

    -- 在父类列表中查找类缺失的方法
    setmetatable(c, {__index = function (t, k)
        return search(k, parents)
    end})

    -- 将'c'作为其实例的元表
    c.__index = c

    -- 为新类定义一个新的构造函数
    function c:new (o)
        o = o or {}
        setmetatable(o, c)
        return o
    end

    return c                      -- 返回新类
end
```

让我们用一个简单的示例来演示 `createClass` 的用法。假设前面提到的类 `Account` 和另一个只有两个方法 `setname` 和 `getname` 的类 `Named`:

```
Named = {}
```



21 面向对象 (Object-Oriented) 编程

```

function Named:getname ()
    return self.name
end

function Named:setname (n)
    self.name = n
end

```

要创建一个同时继承 Account 和 Named 的新类 NamedAccount，只需要调用 createClass：

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

可以像平时一样创建和使用实例：

```

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

```

现在，让我们来学习 Lua 语言是如何对表达式 account:getname() 求值的；更确切地说，让我们来学习 account["getname"] 的求值过程。首先，Lua 语言在 account 中找不到字段"getname"；因此，它就查找 account 的元表中的 __index 字段，在我们的示例中该字段为 NamedAccount。由于在 NamedAccount 中也不存在字段"getname"，所以再从 NamedAccount 的元表中查找 __index 字段。由于这个字段是一个函数，因此 Lua 语言就调用了这个函数（即 search）。该函数先在 Account 中查找"getname"；未找到后，继而在 Named 中查找并最终在 Named 中找到了一个非 nil 的值，也就是最终的搜索结果。

当然，由于这种搜索具有一定的复杂性，因此多重继承的性能不如单继承。一种改进性能的简单做法是将被继承的方法复制到子类中，通过这种技术，类的 __index 元方法会变成：

```

setmetatable(c, {__index = function (t, k) search() --> see [1]
    local v = search(k, parents)
    t[k] = v           -- 保存下来用于下次访问
    return v
end})

```

使用了这种技巧后，在第一次访问过被继承的方法后，再访问被继承的方法就会像访问局部方法一样快了。这种技巧的缺点在于当系统开始运行后修改方法的定义就比较困难了，这是因为这些修改不会沿着继承层次向下传播。

21.4 私有性 (Privacy)

许多人认为，私有性（也被称为信息隐藏，*information hiding*）是一门面向对象语言不可或缺的一部分：每个对象的状态都应该由它自己控制。在一些诸如 C++ 和 Java 的面向对象语言中，我们可以控制一个字段（也被称为实例变量，*instance variable*）或一个方法是否在对象之外可见。另一种非常流行的面向对象语言 Smalltalk，则规定所有的变量都是私有的，而所有的方法都是公有的。第一种面向对象语言 Simula，则不提供任何形式的私有性保护。

此前，我们所学习的 Lua 语言中标准的对象实现方式没有提供私有性机制。一方面，这是使用普通结构（表）来表示对象所带来的后果；另一方面，这也是 Lua 语言为了避免冗余和人为限制所采取的方法。如果读者不想访问一个对象内的内容，那就不要去访问就是了。一种常见的做法是把所有私有名称的最后加上一个下画线，这样就能立刻区分出全局名称了。

不过，尽管如此，Lua 语言的另外一项设计目标是灵活性，它为程序员提供能够模拟许多不同机制的元机制（meta-mechanism）。虽然在 Lua 语言中，对象的基本设计没有提供私有性机制，但可以用其他方式来实现具有访问控制能力的对象。尽管程序员一般不会用到这种实现，但是了解这种实现还是有好处的，因为这种实现既探索了 Lua 语言中某些有趣方面，又可以成为其他更具体问题的良好解决方案。

这种做法的基本思想是通过两个表来表示一个对象：一个表用来保存对象的状态，另一个表用于保存对象的操作（或接口）。我们通过第二个表来访问对象本身，即通过组成其接口的操作来访问。为了避免未授权的访问，表示对象状态的表不保存在其他表的字段中，而只保存在方法的闭包中。例如，如果要用这种设计来表示银行账户，那么可以通过下面的工厂函数创建新的对象：

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}

    local withdraw = function (v)
        self.balance = self.balance - v
    end

    local deposit = function (v)
        self.balance = self.balance + v
    end
```

```

local getBalance = function () return self.balance end

return {
    withdraw = withdraw,
    deposit = deposit,
    getBalance = getBalance
}
end

```

首先，这个函数创建了一个用于保存对象内部状态的表，并将其存储在局部变量 `self` 中。然后，这个函数创建了对象的方法。最后，这个函数会创建并返回一个外部对象，该对象将方法名与真正的方法实现映射起来。这里的关键在于，这些方法不需要额外的 `self` 参数，而是直接访问 `self` 变量。由于没有了额外的参数，我们也就无须使用冒号语法来操作这些对象，而是可以像普通函数那样来调用这些方法：

```

acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance()) --> 60

```

这种设计给予了存储在表 `self` 中所有内容完全的私有性。当 `newAccount` 返回后，就无法直接访问这个表了，我们只能通过在 `newAccount` 中创建的函数来访问它。虽然我们的示例只把一个实例变量放到了私有表中，但还可以将一个对象中的所有私有部分都存入这个表。我们也可以定义私有方法，它们类似于公有方法但不放入接口中。例如，我们的账户可以给余额大于某个值的用户额外 10% 的信用额度，但是又不想让用户访问到这些计算细节，就可以将这个功能按以下方法实现：

```

function newAccount (initialBalance)
    local self = {
        balance = initialBalance,
        LIM = 10000.00,
    }

    local extra = function ()
        if self.balance > self.LIM then
            return self.balance*0.10
        end
    end

```

```

    else
        return 0
    end
end

local getBalance = function ()
    return self.balance + extra()
end

```

同前

与前一个示例一样，任何用户都无法直接访问 extra 函数。

21.5 单方法对象 (Single-method Object)

上述面向对象编程实现的一个特例是对象只有一个方法的情况。在这种情况下，可以不用创建接口表，只要将这个单独的方法以对象的表示形式返回即可。如果读者觉得这听上去有点奇怪，那么应该回忆一下诸如 io.lines 或 string.gmatch 这样的迭代器。一个在内部保存了状态的迭代器就是一个单方法对象。

单方法对象的另一种有趣情况是，这个方法其实是一个根据不同的参数完成不同任务的分发方法 (dispatch method)。这种对象的一种原型实现如下：

```

function newObject (value)
    return function (action, v)
        if action == "get" then return value
        elseif action == "set" then value = v
        else error("invalid action")
        end
    end
end

```

其使用方法很简单：

```

d = newObject(0)
print(d("get"))    --> 0
d("set", 10)

```

```
print(d("get"))
```

这种非传统的对象实现方式是很高效的。虽然 `d("set", 10)` 这样的语法有些奇怪，但也不过只是比传统的 `d:set(10)` 多出了两个字符而已。每个对象使用一个闭包，要比使用一个表的开销更低。虽然使用这种方式不能实现继承，但我们却可以拥有完全的私有性：访问单方法对象中某个成员只能通过该对象所具有的唯一方法进行。

Tcl/Tk 对它的窗口部件使用了类似的做法。在 Tk 中，一个窗口部件的名称就是一个函数（一个窗口命令，*widget command*），这个函数可以根据它的第一个参数完成所有针对该部件的操作。

21.6 对偶表示 (Dual Representation)

实现私有性的另一种有趣方式是使用对偶表示 (*dual representation*)。让我们先看一下什么是对偶表示。

通常，我们使用键来把属性关联到表，例如：

```
table[key] = value
```

不过，我们也可以使用对偶表示：把表当作键，同时又把对象本身当作这个表的键：

```
key = []
...
key[table] = value
```

这里的关键在于：我们不仅可以通过数值和字符串来索引一个表，还可以通过任何值来索引一个表，尤其是可以使用其他的表来索引一个表。

例如，在我们银行账户的实现中，可以把所有账户的余额放在表 `balance` 中，而不是把余额放在每个账户里。我们的 `withdraw` 方法会变成：

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

这样做好处在于私有性。即使一个函数可以访问一个账户，但是除非它能够同时访问表 `balance`，否则也不能访问余额。如果表 `balance` 是一个在模块 `Account` 内部保存的局部变量，那么只有模块内部的函数才能访问它。因此，只有这些函数才能操作账户余额。

在我们继续学习前，必须讨论一下这种实现的一个大的缺陷。一旦我们把账户作为表 balance 中的键，那么这个账户对于垃圾收集器而言就永远也不会变成垃圾，这个账户会留在表中直到某些代码将其从表中显式地移除。这对于银行账户而言可能不是问题（除非销户，否则一个账户通常需要一直有效），但对于其他场景来说则可能是一个较大的缺陷。我们会在 23.3 节中学习如何解决这个问题，但现在我们先忽略它。

示例 21.3 展示了如何使用对偶表示来实现账户。

示例 21.3 使用对偶表示实现账户

```
local balance = {}

Account = {}

function Account:withdraw (v)
    balance[self] = balance[self] - v
end

function Account:deposit (v)
    balance[self] = balance[self] + v
end

function Account:balance ()
    return balance[self]
end

function Account:new (o)
    o = o or {}          -- 如果用户没有提供则创建表
    setmetatable(o, self)
    self.__index = self
    balance[o] = 0        -- 初始余额
    return o
end
```

我们可以像使用其他类一样使用这个类：

```
a = Account:new{}
```

```
a:deposit(100.00)
print(a:balance())
```

不过，我们不能恶意修改账户余额。这种实现通过让表 `balance` 为模块所私有，保证了它的安全性。

对偶表示无须修改即可实现继承。这种实现方式与标准实现方式在内存和时间开销方面基本相同。新对象需要一个新表，而且在每一个被使用的私有表中需要一个新的元素。访问 `balance[self]` 会比访问 `self.balance` 稍慢，这是因为后者使用了局部变量而前者使用了外部变量。通常，这种区别是可以忽略的。正如我们后面会看到的，这种实现对于垃圾收集器来说也需要一些额外的工作。

21.7 练习

练习 21.1: 实现一个类 `Stack`，该类具有方法 `push`、`pop`、`top` 和 `isempty`。

练习 21.2: 实现类 `Stack` 的子类 `StackQueue`。除了继承的方法外，还给这个子类增加一个方法 `insertbottom`，该方法在栈的底部插入一个元素（这个方法使得我们可以把这个类的实例用作队列）。

练习 21.3: 使用对偶表示重新实现类 `Stack`。

练习 21.4: 对偶表示的一种变形是使用代理表示对象（20.4.4节）。每一个对象由一个空的代理表表示，一个内部的表把代理映射到保存对象状态的表。这个内部表不能从外部访问，但是方法可以使用内部表来把 `self` 变量转换为要操作的真正的表。请使用这种方式实现银行账户的示例，然后讨论这种方式的优点和缺点。

22

环境（Environment）

全局变量在大多数编程语言中是让人爱恨交织又不可或缺的。一方面，使用全局变量会明显地使无关的代码部分纠缠在一起，容易导致代码复杂。另一方面，谨慎地使用全局变量又能更好地表达程序中真正的全局概念；此外，虽然全局常量看似无害，但像 Lua 语言这样的动态语言是无法区分常量和变量的。像 Lua 这样的嵌入式语言更复杂：虽然全局变量是在整个程序中均可见的变量，但由于 Lua 语言是由宿主应用调用代码段（chunk）的，因此“程序”的概念不明确。

Lua 语言通过不使用全局变量的方法来解决这个难题，但又不遗余力地在 Lua 语言中对全局变量进行模拟。在第一种近似的模拟中，我们可以认为 Lua 语言把所有的全局变量保存在一个称为全局环境（*global environment*）的普通表中。在本章的后续内容中，我们可以看到 Lua 语言可以用几种环境来保存“全局”变量，但现在还是来关注第一种近似的模拟。

由于不需要再为全局变量创造一种新的数据结构，因此使用一个表来保存全局变量的一个优点是简化了 Lua 语言的内部实现。另一个优点是，可以像操作其他表一样操作这个表。为了便于实现这种操作方式，Lua 语言将全局环境自身^①保存在全局变量 _G 中（因此，_G 与 _G 等价）。例如，如下代码输出了全局环境中所有全局变量的名称：

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

^①译者注：因为全局环境就是一个表。

22.1 具有动态名称的全局变量

通常，赋值操作对于访问和设置全局变量已经足够了。然而，有时我们也需要某些形式的元编程 (meta-programming)。例如，我们需要操作一个全局变量，而这个全局变量的名称却存储在另一个变量中或者经由运行时计算得到。为了获取这个变量的值，许多程序员会写出下面的代码：

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

例如，如果 `varname` 是 `x`，那么字符串连接的结果就是`"return x"`，当执行时就能得到期望的结果。然而，在这段代码中涉及一个新代码段的创建和编译，在一定程度上开销昂贵。我们可以使用下面的代码来实现相同的效果，但效率却比之前的高出一个数量级：

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

由于全局环境是一个普通的表，因此可以简单地使用对应的键（变量名）直接进行索引。

类似地，我们可以通过编写 `_G[varname]=value` 给一个名称为动态计算出的全局变量赋值。不过，请注意，有些程序员对于这种机制的使用可能有些过度而写出诸如 `_G["a"]=_G["b"]` 这样的代码，而这仅仅是 `a=b` 的一种复杂写法。

上述问题的一般化形式是，允许字段使用诸如`"io.read"`或`"a.b.c.d"`这样的动态名称。如果直接使用 `_G["io.read"]`，显然是不能从表 `io` 中得到字段 `read` 的。但我们可以编写一个函数 `getfield` 让 `getfield("io.read")` 返回想要的结果。这个函数主要是一个循环，从 `_G` 开始逐个字段地进行求值：

```
function getfield (f)
    local v = _G      -- 从全局表开始
    for w in string.gmatch(f, "[%a_][%w_]*") do
        v = v[w]
    end
    return v
end
```

我们使用函数 `gmatch` 来遍历 `f` 中的所有标识符。

与之对应的设置字段的函数稍显复杂。像 `a.b.c.d=v` 这样的赋值等价于以下的代码：

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

也就是说，我们必须一直取到最后一个名称，然后再单独处理最后的这个名称。示例 22.1 中的函数 `setfield` 完成了这个需求，并且同时创建了路径中不存在路径对应的中间表。

示例 22.1 函数 `setfield`

```
function setfield (f, v)
    local t = _G           -- 从全局表开始
    for w, d in string.gmatch(f, "([%a_][%w_]*)(%.?)") do
        if d == "." then   -- 不是最后一个名字?
            t[w] = t[w] or {} -- 如果不存在则创建表
            t = t[w]          -- 获取表
        else                -- 最后一个名字
            t[w] = v          -- 进行赋值
        end
    end
end
```

上例中使用的模式将捕获字段名称保存在变量 `w` 中，并将其后可选的点保存在变量 `d` 中。如果字段名后没有点，那么该字段就是最后一个名称。

下面的代码通过上例中的函数创建了全局表 `t` 和 `t.x`，并将 10 赋给了 `t.x.y`：

```
setfield("t.x.y", 10)
print(t.x.y)           --> 10
print(getfield("t.x.y")) --> 10
```

22.2 全局变量的声明

Lua 语言中的全局变量不需要声明就可以使用。虽然这种行为对于小程序来说较为方便，但在大型程序中一个简单的手误^①就有可能造成难以发现的 Bug。不过，如果我们乐意的话，也可以改变这种行为。由于 Lua 语言将全局变量存放在一个普通的表中，所以可以通过元表来发现访问不存在全局变量的情况。

一种方法是简单地检测所有对全局表中不存在键的访问：

^①译者注：指打字打错。

```

setmetatable(_G, {
    __newindex = function (_, n)
        error("attempt to write to undeclared variable " .. n, 2)
    end,
    __index = function (_, n)
        error("attempt to read undeclared variable " .. n, 2)
    end,
})

```

这段代码执行后，所有试图对不存在全局变量的访问都将引发一个错误：

```

> print(a)
stdin:1: attempt to read undeclared variable a

```

但是，我们应该如何声明一个新的变量呢？方法之一是使用函数 `rawset`，它可以绕过元方法：

```

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

```

其中，`or` 和 `false` 保证新变量一定会得到一个不为 nil 的值。

另外一种更简单的方法是把对新全局变量的赋值限制在仅能在函数内进行，而代码段外层的代码则被允许自由赋值。

要检查赋值是否在主代码段中必须用到调试库。调用函数 `debug.getinfo(2, "S")` 将返回一个表，其中的字段 `what` 表示调用元方法的函数是主代码段还是普通的 Lua 函数还是 C 函数（我们会在 25.1 节中学习函数 `debug.getinfo` 的细节）。使用该函数，可以将 `__newindex` 元方法重写：

```

__newindex = function (t, n, v)
    local w = debug.getinfo(2, "S").what
    if w ~= "main" and w ~= "C" then
        error("attempt to write to undeclared variable " .. n, 2)
    end
    rawset(t, n, v)
end

```

这个新版本还可以接受来自 C 代码的赋值，因为一般 C 代码都知道自己究竟在做什么。

如果要测试一个变量是否存在，并不能简单地将它与 nil 比较。因为如果它为 nil，那么访问就会引发一个错误。这时，应该使用 rawget 来绕过元方法：

```
if rawget(_G, var) == nil then
    -- 'var' 未被声明
    ...
end
```

正如前面所提到的，我们不允许值为 nil 的全局变量，因为值为 nil 的全局变量都会被自动地认为是未声明的。但是，要允许值为 nil 的全局变量也不难，只需要引入一个辅助表来保存已声明变量的名称即可。一旦调用了元方法，元方法就会检查该表，看变量是否是未声明过的。最终的代码可能与示例 22.2 中的代码类似。

示例 22.2 检查全局变量的声明

```
local declaredNames = {}

setmetatable(_G, {
    __newindex = function (t, n, v)
        if not declaredNames[n] then
            local w = debug.getinfo(2, "S").what
            if w ~= "main" and w ~= "C" then
                error("attempt to write to undeclared variable "..n, 2)
            end
            declaredNames[n] = true
        end
        rawset(t, n, v) -- 进行真正的赋值
    end,
    __index = function (_, n)
        if not declaredNames[n] then
            error("attempt to read undeclared variable "..n, 2)
        else
            return nil
        end
    end
})
```

```
    end,
}
```

现在，即使像 `x = nil` 这样的赋值也能够声明全局变量了。

上述两种方法所导致的开销都基本可以忽略不计。在第一种方法中，在普通操作期间元方法不会被调用。在第二种方法中，元方法只有当程序访问一个值为 `nil` 的变量时才会被调用。

Lua 语言发行版本中包含一个 `strict.lua` 模块，它使用示例 22.2 中的基础代码实现了对全局变量的检查。在编写 Lua 语言代码时使用它是一个良好的习惯。

22.3 非全局环境

在 Lua 语言中，全局变量并不一定非得是真正全局的。正如笔者此前所提到的，Lua 语言甚至根本没有全局变量。由于我们在本书中不断地使用全局变量，所以一开始听上去这可能很诡异。正如笔者所说，Lua 语言竭尽全力地让程序员有全局变量存在的幻觉。现在，让我们看看 Lua 语言是如何构建这种幻觉的。^①

首先，让我们忘掉全局变量而从自由名称的概念开始讨论。一个自由名称 (*free name*) 是指没有关联到显式声明上的名称，即它不出现在对应局部变量的范围内。例如，在下面的代码段中，`x` 和 `y` 是自由名称，而 `z` 则不是：

```
local z = 10
x = y + z
```

接下来就到了关键的部分：Lua 语言编译器将代码段中的所有自由名称 `x` 转换为 `_ENV.x`。因此，此前的代码段完全等价于：

```
local z = 10
_ENV.x = _ENV.y + z
```

但是这里新出现的 `_ENV` 变量又究竟是什么呢？

我们刚才说过，Lua 语言中没有全局变量。因此，`_ENV` 不可能是全局变量。在这里，编译器实际上又进行了一次巧妙的工作。笔者已经提到过，Lua 语言把所有的代码段都当作匿名函数。所以，Lua 语言编译器实际上将原来的代码段编译为如下形式：

^①这种机制是 Lua 5.1 和 Lua 5.2 之间改变最大的部分。接下来的讨论基本都不适用于 Lua 5.1。

```

local _ENV = some value (某些值)
return function (...)
    local z = 10
    _ENV.x = _ENV.y + z
end

```

也就是说，Lua 语言是在一个名为 `_ENV` 的预定义上值（一个外部的局部变量，*upvalue*）存在的情况下编译所有的代码段的。因此，所有的变量要么是绑定到了一个名称的局部变量，要么是 `_ENV` 中的一个字段，而 `_ENV` 本身是一个局部变量（一个上值）。

`_ENV` 的初始值可以是任意的表（实际上也不用一定是表，我们会在后续讨论）。任何一个这样的表都被称为一个环境。为了维持全局变量存在的幻觉，Lua 语言在内部维护了一个表来用作全局环境（*global environment*）。通常，当加载一个代码段时，函数 `load` 会使用预定义的上值来初始化全局环境。因此，原始的代码段等价于：

```

local _ENV = the global environment (全局环境)
return function (...)
    local z = 10
    _ENV.x = _ENV.y + z
end

```

上述赋值的结果是，全局环境中的字段 `x` 得到全局环境中字段 `y` 加 10 的结果。

乍一看，这可能像是操作全局变量的一种相当拐弯抹角的方式。笔者也不会去争辩说这是最简单的方式，但是，这种方式比那些更简单的实现方法具有更多的灵活性。

在继续学习前，让我们总结一下 Lua 语言中处理全局变量的方式：

- 编译器在编译所有代码段前，在外层创建局部变量 `_ENV`；
- 编译器将所有自由名称 `var` 变换为 `_ENV.var`；
- 函数 `load`（或函数 `loadfile`）使用全局环境初始化代码段的第一个上值，即 Lua 语言内部维护的一个普通的表。

实际上，这也并不是太复杂。

有些人由于试图从这些规则中引申出额外的“魔法”而感到困惑；其实，这些规则并没有额外的含义。尤其是，前两条规则完全是由编译器进行的。除了是由编译器预先定义的，`_ENV` 只是一个单纯的普通变量。抛开编译器，名称 `_ENV` 对于 Lua 语言来说根本没有特殊含

义。^①类似地，从 `x` 到 `_ENV.x` 的转换是纯粹的语法转换，没有隐藏的含义。尤其是，在转换后，按照标准的可见性规则，`_ENV` 引用的是其所在位置所有可见的 `_ENV` 变量。

22.4 使用 `_ENV`

在本节中，我们会看到一些探索由 `_ENV` 带来的灵活性的手段。请记住，本节中的大部分示例必须以单段代码的方式运行。如果在交互模式下一行一行地输入代码，那么每一行代码都会变成一段独立的代码，因此每一行都会有一个不同的 `_ENV` 变量。为了把代码当作一个代码段运行，要么把代码保存在一个文件中运行，要么使用 `do-end` 将代码段包围起来。

由于 `_ENV` 只是一个普通的变量，因此可以对其赋值或像访问其他变量一样访问它。赋值语句 `_ENV = nil` 会使得后续代码不能直接访问全局变量。这可以用来控制代码使用哪种变量：

```
local print, sin = print, math.sin
_ENV = nil
print(13)           --> 13
print(sin(13))     --> 0.42016703682664
print(math.cos(13)) -- error!
```

任何对自由名称（“全局变量”）的赋值都会引发类似的错误。

我们可以显式地使用 `_ENV` 来绕过局部声明：

```
a = 13          -- 全局的
local a = 12
print(a)        --> 12 (局部的)
print(_ENV.a)   --> 13 (全局的)
```

用 `_G` 也可以：

```
a = 13          -- 全局的
local a = 12
print(a)        --> 12 (局部的)
print(_G.a)     --> 13 (全局的)
```

^①说实话，Lua 语言将 `_ENV` 用于错误信息，以便于能够像报告涉及 `global x` 的一个错误一样报告涉及变量 `_ENV.x` 的错误。

通常，_G 和 _ENV 指向的是同一个表。但是，尽管如此，它们是很不一样的实体。_ENV 是一个局部变量，所有对“全局变量”的访问实际上访问的都是 _ENV。_G 则是一个在任何情况下都没有任何特殊状态的全局变量。按照定义，_ENV 永远指向的是当前的环境；而假设在可见且无人改变过其值的前提下，_G 通常指向的是全局环境。

_ENV 的主要用途是用来改变代码段使用的环境。一旦改变了环境，所有的全局访问就都将使用新表：

```
-- 将当前的环境改为一个新的空表
_ENV = {}

a = 1           -- 在_ENV中创建字段
print(a)
--> stdin:4: attempt to call global 'print' (a nil value)
```

如果新环境是空的，就会丢失所有的全局变量，包括函数 print。因此，应该首先把一些有用的值放入新环境，比如全局环境：

```
a = 15          -- 创建一个全局变量
_ENV = {g = _G} -- 改变当前环境
a = 1           -- 在_ENV中创建一个字段
g.print(_ENV.a, g.a) --> 1    15
```

这时，当访问“全局”的 g（位于 _ENV 而不是全局环境中）时，我们使用的是全局环境，在其中能够找到函数 print。

我们可以使用 _G 代替 g，从而重写前面的例子：

```
a = 15          -- 创建一个全局变量
_ENV = {_G = _G} -- 改变当前环境
a = 1           -- 在_ENV中创建一个字段
_G.print(_ENV.a, _G.a) --> 1    15
```

_G 只有在 Lua 语言创建初始化的全局表并让字段 _G 指向它自己的时候，才会出现特殊状态。Lua 语言并不关心该变量的当前值。不过尽管如此，就像我们在上面重写的示例中所看到的那样，将指向全局环境的变量命名为同一个名字 (_G) 是一个惯例。

另一种把旧环境装入新环境的方式是使用继承：

```
a = 1
local newgt = {}          -- 创建新环境
```

```

setmetatable(newgt, {__index = _G})
_ENV = newgt          -- 设置新环境
print(a)              --> 1

```

在这段代码中，新环境从全局环境中继承了函数 `print` 和 `a`。不过，任何赋值都会发生在新表中。虽然我们仍然能通过 `_G` 来修改全局环境中的变量，但如果误改了全局环境中的变量也不会有什么影响。

```

-- 接此前的代码
a = 10
print(a, _G.a)        --> 10    1
_G.a = 20
print(_G.a)           --> 20

```

作为一个普通的变量，`_ENV` 遵循通常的定界规则。特别地，在一段代码中定义的函数可以按照访问其他外部变量一样的规则访问 `_ENV`：

```

_ENV = {_G = _G}
local function foo ()
  _G.print(a)      -- 编译为 '_ENV._G.print(_ENV.a)'
end
a = 10
foo()               --> 10
_ENV = {_G = _G, a = 20}
foo()               --> 20

```

如果定义一个名为 `_ENV` 的局部变量，那么对自由名称的引用将会绑定到这个新变量上：

```

a = 2
do
  local _ENV = {print = print, a = 14}
  print(a)      --> 14
end
print(a)      --> 2  (回到原始的_ENV中)

```

因此，可以很容易地使用私有环境定义一个函数：

```

function factory (_ENV)
  return function () return a end
end

```

```

end

f1 = factory{a = 6}
f2 = factory{a = 7}
print(f1())      --> 6
print(f2())      --> 7

```

`factory` 函数创建了一个简单的闭包，这个闭包返回了其中“全局”的 `a`。每当闭包被创建时，闭包可见的变量 `_ENV` 就成了外部 `factory` 函数的参数 `_ENV`。因此，每个闭包都会使用自己的外部变量（作为上值）来访问其自由名称。

使用普遍的定界规则，我们可以有几种方式操作环境。例如，可以让多个函数共享一个公共环境，或者让一个函数改变它与其他函数共享的环境。

22.5 环境和模块

在 17.2 节中，当我们讨论如何编写模块时，笔者提到过模块的缺点之一在于很容易污染全局空间，例如在私有声明中忘记 `local` 关键字。环境为解决这个问题提供了一种有趣的方式。一旦模块的主程序块有一个独占的环境，则不仅该模块所有的函数共享了这个环境，该模块的全局变量也进入到了这个环境中。我们可以将所有的公有函数声明为全局变量，这样它们就会自动地进入分开的环境中。模块所要做的就是将这个环境赋值给变量 `_ENV`。之后，当我们声明函数 `add` 时，它会变成 `M.add`：

```

local M = {}
_ENV = M
function add (c1, c2)
    return new(c1.r + c2.r, c1.i + c2.i)
end

```

此外，我们在调用同一模块中的其他函数时不需要任何前缀。在此前的代码中，`add` 会从其环境中得到 `new`，也就是 `M.new`。

这种方法为模块提供了一种良好的支持，只需要程序员多做一点额外的工作。使用这种方法，完全不需要前缀，并且调用一个导出的函数与调用一个私有函数没有什么区别。即使程序员忘记了 `local` 关键字，也不会污染全局命名空间。相反，他只是让一个私有函数变成了公有函数而已。

不过尽管如此，笔者目前还是倾向于使用原始的基本方法。也许原始的基本方法需要更多的工作，但代码会更加清晰。为了避免错误地创建全局变量，笔者使用把 nil 赋给 _ENV 的方式。在把 _ENV 设为 nil 后，任何对全局变量的赋值都会抛出异常。这种方式的另一个好处是无须修改代码也可以在老版本的 Lua 语言中运行（在 Lua 5.1 中，给 _ENV 赋值虽然不能阻止出错，但也并不会造成问题）。

为了访问其他模块，我们可以使用在之前章节中讨论过的方法。例如，可以声明一个保存全局环境的局部变量：

```
local M = {}
local _G = _G
_ENV = nil
```

然后在全局名称前加上 _G 和模块名 M 即可。

另一种更规范的访问其他模块的做法是只把需要的函数或模块声明为局部变量：

```
-- 模块初始化
local M = {}

-- 导入部分：
-- 声明该模块需要的外部函数或模块等
local sqrt = math.sqrt
local io = io

-- 从此以后不能再进行外部访问
_ENV = nil
```

这种方式需要做更多工作，但是它能清晰地列出模块的依赖。

22.6 _ENV 和 load

正如笔者此前提到的，函数 `load` 通常把被加载代码段的上值 `_ENV` 初始化为全局环境。不过，函数 `load` 还有一个可选的第四个参数来让我们为 `_ENV` 指定一个不同的初始值。（函数 `loadfile` 也有一个类似的参数。）

例如，假设我们有一个典型的配置文件，该配置文件定义了程序要使用的几个常量和函数，如下：

```
-- 文件'config.lua'
width = 200
height = 300
...
```

可以使用如下的代码加载该文件：

```
env = {}
loadfile("config.lua", "t", env)()
```

配置文件中的所有代码会运行在空的环境 `env` 中，类似于某种沙盒。特别地，所有的定义都会进入这个环境中。即使出错，配置文件也无法影响任何别的东西，甚至是恶意的代码也不能对其他东西造成任何破坏。除了通过消耗 CPU 时间和内存来制造拒绝服务（Denial of Service，DoS）攻击，恶意代码也做不了什么其他的事。

有时，我们可能想重复运行一段代码数次，每一次使用一个不同的环境。在这种情况下，函数 `load` 可选的参数就没用了。此时，我们有另外两种选择。

第一种选择是使用调试库中的函数 `debug.setupvalue`。顾名思义，函数 `setupvalue` 允许改变任何指定函数的上值，例如：

```
f = load("b = 10; return a")
env = {a = 20}
debug.setupvalue(f, 1, env)
print(f())           --> 20
print(env.b)         --> 10
```

`setupvalue` 的第一个参数是指定的函数，第二个参数是上值的索引，第三个参数是新的上值。对于这种用法，第二个参数永远是 1：当函数表示的是一段代码时，Lua 语言可以保证它只有一个上值且上值就是 `_ENV`。

这种方式的一个小缺点在于依赖调试库。调试库打破了有关程序的一些常见假设。例如，`debug.setupvalue` 打破了 Lua 语言的可见性规则，而可见性规则可以保证我们不能从词法定界的范围外访问局部变量。

另一种在几个不同环境中运行代码段的方式是每次加载代码段时稍微对其进行一下修改。假设我们在要加载的代码段前加入一行：

```
_ENV = ...;
```

请注意，由于 Lua 语言把所有的代码段都当作可变长参数函数进行编译，因此，多出的这一行代码会把传给代码段的第一个参数赋给 `_ENV`，从而把参数设为环境。以下的代码使用练习 16.1 中实现的函数 `loadwithprefix` 演示了这种做法：

```
prefix = "_ENV = ...;"  
f = loadwithprefix(prefix, io.lines(filename, "*L"))  
...  
env1 = {}  
f(env1)  
env2 = {}  
f(env2)
```

22.7 练习

练习 22.1：本章开始时定义的函数 `getfield`，由于可以接收像 `math?sin` 或 `string!!!gsub` 这样的“字段”而不够严谨。请将其进行重写，使得该函数只能支持点作为名称分隔符。

练习 22.2：请详细解释下列程序做了什么，以及输出的结果是什么。

```
local foo  
do  
    local _ENV = _ENV  
    function foo () print(X) end  
end  
X = 13  
_ENV = nil  
foo()  
X = 0
```

练习 22.3：请详细解释下列程序做了什么，以及输出的结果是什么。

```
local print = print  
function foo (_ENV, a)  
    print(a + b)  
end  
  
foo({b = 14}, 12)  
foo({b = 10}, 1)
```

23

垃圾收集

Lua 语言使用自动内存管理。程序可以创建对象（表、闭包等），但却没有函数来删除对象。Lua 语言通过垃圾收集（*garbage collection*）自动地删除成为垃圾的对象，从而将程序员从内存管理的绝大部分负担中解放出来。更重要的是，将程序员从与内存管理相关的大多数 Bug 中解放出来，例如无效指针（dangling pointer）和内存泄漏（memory leak）等问题。

在一个理想的环境中，垃圾收集器对程序员来说是不可见的，就像一个好的清洁工不会和其他工人打交道一样。不过，有时即使是最智能的垃圾收集器也会需要我们的辅助。在某些关键的性能阶段，我们可能需要将其停止，或者让其只在特定的时间运行。另外，一个垃圾收集器只能收集它确定是垃圾的内容，而不能猜测我们把什么当作垃圾。没有垃圾收集器能够做到让我们完全不用操心资源管理的问题，比如驻留内存（hoarding memory）和外部资源。

弱引用表（weak table）、析构器（finalizer）和函数 `collectgarbage` 是在 Lua 语言中用来辅助垃圾收集器的主要机制。弱引用表允许收集 Lua 语言中还可以被程序访问的对象；析构器允许收集不在垃圾收集器直接控制下的外部对象；函数 `collectgarbage` 则允许我们控制垃圾收集器的步长。在本章中，我们会学习这几种机制。

23.1 弱引用表

正如此前所说的，垃圾收集器不能猜测我们认为哪些是垃圾。一个典型的例子就是栈，栈通常由一个数组和一个指向栈顶的索引实现。我们知道，数组的有效部分总是向顶部扩展

的，但 Lua 语言却不知道。如果弹出一个元素时只是简单地递减顶部索引，那么这个仍然留在数组中的对象对于 Lua 语言来说并不是垃圾。同理，即使是程序不会再用到的、存储在全局变量中的对象，对于 Lua 语言来说也不是垃圾。在这两种情况下，都需要我们（的代码）将这些对象所在的位置赋为 nil，以便这些位置不会锁定可释放的对象。

不过，简单地清除引用可能还不够。在有些情况下，还需要程序和垃圾收集器之间的协作。一个典型的例子是，当我们要保存某些类型（例如，文件）的活跃对象的列表时。这个需求看上去很简单，我们只需要把每个新对象插入数组即可；但是，一旦一个对象成为了数组的一部分，它就再也无法被回收了！虽然已经没有其他任何地方在引用它，但数组依然在引用它。除非我们告诉 Lua 语言数组对该对象的引用不应该阻碍对此对象的回收，否则 Lua 语言本身是无从知晓的。

弱引用表就是这样一种用来告知 Lua 语言一个引用不应阻止对一个对象回收的机制。所谓弱引用（*weak reference*）是一种不在垃圾收集器考虑范围内的对象引用。如果对一个对象的所有引用都是弱引用，那么垃圾收集器将会回收这个对象并删除这些弱引用。Lua 用语言通过弱引用表实现弱引用，弱引用表就是元素均为弱引用的表，这意味着如果一个对象只被一个弱引用表持有，那么 Lua 语言最终会回收这个对象。

表由键值对组成，其两者都可以容纳任意类型的对象。在正常情况下，垃圾收集器不会回收一个在可访问的表中作为键或值的对象。也就是说，键和值都是强（*strong*）引用，它们会阻止对其所指向对象的回收。在一个弱引用表中，键和值都可以是弱引用的。这就意味着有三种类型的弱引用表，即具有弱引用键的表、具有弱引用值的表及同时具有弱引用键和值的表。不论是哪种类型的弱引用表，只要有一个键或值被回收了，那么对应的整个键值对都会被从表中删除。

一个表是否为弱引用表是由其元表中的 `__mode` 字段所决定的。当这个字段存在时，其值应为一个字符串：如果这个字符串是“k”，那么这个表的键是弱引用的；如果这个字符串是“v”，那么这个表的值是弱引用的；如果这个字符串是“kv”，那么这个表的键和值都是弱引用的。下面的示例虽然有些刻意，但演示了弱引用表的基本行为：

```
a = {}
mt = {__mode = "k"}
setmetatable(a, mt)      -- 现在'a'的键是弱引用的了
key = {}                  -- 创建第一个键
a[key] = 1
key = {}                  -- 创建第二个键
a[key] = 2
```

```

collectgarbage()          -- 强制进行垃圾回收
for k, v in pairs(a) do print(v) end
--> 2

```

在本例中，第二句赋值 `key={}` 覆盖了指向第一个键的索引。调用 `collectgarbage` 强制垃圾收集器进行一次完整的垃圾收集。由于已经没有指向第一个键的其他引用，因此 Lua 语言会回收这个键并从表中删除对应的元素。然而，由于第二个键仍然被变量 `key` 所引用，因此 Lua 不会回收它。

请注意，只有对象可以从弱引用表中被移除，而像数字和布尔这样的“值”是不可回收的。例如，如果我们在表 `a`（之前的示例）中插入一个数值类型的键，那么垃圾收集器永远不会回收它。当然，如果在一个值为弱引用的弱引用表中，一个数值类型键相关联的值被回收了，那么整个元素都会从这个弱引用表中被删除。

字符串在这里表现了一些细微的差别，虽然从实现的角度看字符串是可回收的，但字符串又与其他的可回收对象不同。其他的对象，例如表和闭包，都是被显式创建的。例如，当 Lua 语言对表达式 `{}` 求值时会创建一个新表。然而，当对表达式 `"a".."b"` 求值时，Lua 语言会创建一个新字符串么？如果当前系统中已有了一个字符串 `"ab"` 会怎么样？Lua 语言会创建一个新的字符串么？编译器会在运行程序前先创建这个字符串吗？其实，这些都无关紧要，因为它们都是实现上的细节。从程序员的角度看，字符串是值而不是对象。所以，字符串就像数值和布尔值一样，对于一个字符串类型的键来说，除非它对应的值被回收，否则是不会从弱引用表中被移除的。

23.2 记忆函数（Memorize Function）

空间换时间是一种常见的编程技巧。我们可以通过记忆（*memorize*）函数的执行结果，在后续使用相同参数再次调用该函数时直接返回之前记忆^①的结果，来加快函数的运行速度。^②

假设有一个通用的服务器，该服务器接收的请求是以字符串形式表示的 Lua 语言代码。每当服务器接收到一个请求时，它就对字符串运行 `load` 函数，然后再调用编译后的函数。不过，函数 `load` 的开销很昂贵，而且发送给服务器的某些命令的出现频率可能很高。这样，与其每次收到一条诸如 `"closeconnection()"` 这样的常见命令就重复地调用函数 `load`，还不如

^①译者注：这里的记忆实际就是保存下来。

^②虽然单词“*memorize*”精确地表达了我们的想法，但是程序员社区还创造了一个新词，*memoize*，来描述这种技术。但是笔者坚持使用前者。

让服务器用一个辅助表记忆所有函数 `load` 的执行结果。在调用函数 `load` 前，服务器先在表中检查指定的字符串是否已经被处理过。如果没有，就（且只在这种情况下）调用函数 `load` 并将返回值保存到表中。我们可以将这种行为封装为一个新的函数：

```
local results = {}

function mem_loadstring (s)
    local res = results[s]
    if res == nil then
        -- 已有结果么?
        res = assert(load(s))
        -- 计算新结果
        results[s] = res
        -- 保存结果以便后续重用
    end
    return res
end
```

这种模式节省的开销非常可观。但是，它也可能导致不易察觉的资源浪费。虽然有些命令会重复出现，但也有很多命令可能就出现一次。渐渐地，表 `results` 会堆积上服务器收到的所有命令及编译结果；在运行了一段足够长的时间后，这种行为会耗尽服务器的内存。

弱引用表为解决这个问题提供了一种简单的方案，如果表 `results` 具有弱引用的值，那么每个垃圾收集周期都会删除所有那个时刻未使用的编译结果（基本上就是全部）：

```
local results = {}

setmetatable(results, {__mode = "v"}) -- 让值成为弱引用的

function mem_loadstring (s)
    同前
end
```

实际上，因为索引永远是字符串，所以如果愿意的话，我们可以让这个表变成完全弱引用的：

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

最终达到的效果是完全一样的。

记忆技术（memorization technique）还可以用来确保某类对象的唯一性。例如，假设一个系统用具有三个相同取值范围的字段 `red`、`green` 和 `blue` 的表来表示颜色，一个简单的颜色工厂函数每被调用一次就生成一个新颜色：

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

使用记忆技术，我们就可以为相同的颜色复用相同的表。要为每一种颜色创建一个唯一的键，只需要使用分隔符把颜色的索引连接起来即可：

```
local results = {}

setmetatable(results, {__mode = "v"}) -- 让值成为弱引用的

function createRGB (r, g, b)
    local key = string.format("%d-%d-%d", r, g, b)
    local color = results[key]
    if color == nil then
        color = {red = r, green = g, blue = b}
        results[key] = color
    end
    return color
end
```

这种实现的一个有趣结果是，由于两种同时存在的颜色必定是由同一个表来表示，所以用户可以使用基本的相等运算符比较两种颜色。因为随着时间的迁移垃圾收集器会清理表 `results`，所以一种指定的颜色在不同的时间内可能由不同的表来表示。不过，只要一种颜色正在被使用，它就不会从 `results` 中被移除。因此，一种颜色与一种新颜色相比已经存在了多长时间，这种颜色对应的表也存在了对应长度的时间，也可以被新颜色复用。

23.3 对象属性 (Object Attribute)

弱引用表的另外一种重要应用是将属性与对象关联起来。在各种各样的情况下，我们都需要把某些属性绑定到某个对象，例如函数名、表的默认值及数组的大小等。

当对象是一个表时，可以通过适当的唯一键把属性存储在这个表自身中（正如之前所看到的，创建唯一键的一种简单和防止出错的方法是创建一个新表并把它当作键使用）。不过，如果对象不是一个表，那么它就不能保存它自己的属性。另外，即使是表，有时我们也不想把属性保存在原始的对象中。例如，当想保持属性的私有性时，或不想让属性干扰表的遍历时，就需要用其他办法来关联对象与属性。

当然，外部表为对象和属性的映射提供了一种理想的方法，也就是我们在21.6节中看到的对偶表示，其中将对象用作键、将对象的属性用作值。由于 Lua 语言允许使用任意类型的对象作为键，因此一个外部表可以保存任意类型对象的属性。此外，存储在外部表中的属性不会干扰其他对象，并且可以像表本身一样是私有的。

不过，这个看似完美的方案有一个重大缺陷：一旦我们把一个对象当作表中的一个键，那么就是引用了它。Lua 语言无法回收一个正在被用作键的对象。例如，如果使用一个普通的表来映射函数和函数名，那么这些函数就永远无法被回收。正如读者可能猜到的一样，可以使用弱引用表来解决这个缺陷。不过，这次我们需要的是弱引用的键。使用弱引用键时，如果没有其他的引用，则不会阻止键被回收。另一方面，这个表不能有弱引用的值，否则，活跃对象的属性也可能被回收。

23.4 回顾具有默认值的表

20.4.3 节中讨论了如何实现具有非 nil 默认值的表。我们已经见到过一种特殊的技术，也注明了还有两种技术需要弱引用表的支持待后续讨论。现在，到了回顾这个主题的时候。正如我们马上要看到的，这两种用于默认值的技术其实是刚刚学习过的对偶表示和记忆这两种通用技术的特例。

在第一种解决方案中，我们使用了一个弱引用表来映射每一个表和它的默认值：

```
local defaults = {}
setmetatable(defaults, {__mode = "k"})
local mt = {__index = function (t) return defaults[t] end}
function setDefault (t, d)
    defaults[t] = d
    setmetatable(t, mt)
end
```

这是对偶表示的一种典型应用，其中使用了 `defaults[t]` 来表示 `t.default`。如果表 `defaults` 没有弱引用的键，那么所有具有默认值的表就会永远存在下去。

在第二种解决方案中，我们对不同的默认值使用了不同的元表，在遇到重复的默认值时会复用相同的元表。这是记忆技术的一种典型应用：

```
local metas = {}
setmetatable(metas, {__mode = "v"})
function setDefault (t, d)
    local mt = metas[d]
    if mt == nil then
        mt = {__index = function () return d end}
        setmetatable(mt, {__mode = "v"})
        metas[d] = mt
    end
    setmetatable(t, mt)
```

```

    metas[d] = mt      -- 记忆
end
setmetatable(t, mt)
end

```

在这种情况下，我们使用弱引用的值使得不再被使用的元表能够被回收。

这两种实现哪种更好取决于具体的情况。这两种实现具有类似的复杂度和性能表现，第一种实现需要为每个具有默认值的表（表 `defaults` 中的一个元素）分配几个字节的内存，而第二种实现则需要为每个不同的默认值分配若干内存（一个新表、一个新闭包和表 `metas` 中的一个元素）。因此，如果应用中有上千个具有少量不同默认值的表，那么第二种实现明显更好。不过，如果只有少量共享默认值的表，那么就应该选择第一种实现。

23.5 瞬表 (Ephemeron Table)

一种棘手的情况是，一个具有弱引用键的表中的值又引用了对应的键。

这种情况比看上去的更加常见。一个典型的示例是常量函数工厂 (constant-function factory)。这种工厂的参数是一个对象，返回值是一个被调用时返回传入对象的函数：

```

function factory (o)
    return (function () return o end)
end

```

这种工厂是实现记忆的一种很好的手段，可以避免在闭包已经存在时又创建新的闭包。示例 23.1 展示了这种改进。

示例 23.1 使用记忆技术的常量函数工厂

```

do
    local mem = {}      -- 记忆表
    setmetatable(mem, {__mode = "k"})
    function factory (o)
        local res = mem[o]
        if not res then
            res = (function () return o end)
            mem[o] = res
        end
        return res
    end
end

```

```

    end
    return res
end
end

```

不过，这里另有玄机。请注意，表 `mem` 中与一个对象关联的值（常量函数）回指了它自己的键（对象本身）。虽然表中的键是弱引用的，但是表中的值却不是弱引用的。从一个弱引用表的标准理解看，记忆表中没有任何东西会被移除。由于值不是弱引用的，所以对于每一个函数来说都存在一个强引用。每一个函数都指向其对应的对象，因而对于每一个键来说都存在一个强引用。因此，即使有弱引用的键，这些对象也不会被回收。

不过，这种严格的理解不是特别有用。大多数人希望一个表中的值只能通过对应的键来访问。我们可以认为之前的情况是某种环，其中闭包引用了指向闭包（通过记忆表）的对象。

Lua 语言通过瞬表^①的概念来解决上述问题。^②在 Lua 语言中，一个具有弱引用键和强引用值的表是一个瞬表。在一个瞬表中，一个键的可访问性控制着对应值的可访问性。更确切地说，考虑瞬表中的一个元素 (k, v) ，指向的 v 的引用只有当存在某些指向 k 的其他外部引用存在时才是强引用，否则，即使 v （直接或间接地）引用了 k ，垃圾收集器最终会收集 k 并把元素从表中移除。

23.6 析构器 (Finalizer)

虽然垃圾收集器的目标是回收对象，但是它也可以帮助程序员来释放外部资源。出于这种目的，几种编程语言提供了析构器。析构器是一个与对象关联的函数，当该对象即将被回收时该函数会被调用。

Lua 语言通过元方法 `__gc` 实现析构器，如下例所示：

```

o = {x = "hi"}
setmetatable(o, {__gc = function (o) print(o.x) end})
o = nil

```

^①译者注：ephemeron 是蜉蝣的意思，表示生命极短暂之物，译者查阅了很多中文资料但并未找到任何有关 ephemeron 的已有中文翻译，故按照理自己的理解翻译为了瞬表。在 <http://www.inf.puc-rio.br/~roberto/docs/ry08-06.pdf> 中，有一篇名为《Eliminating Cycles in Weak Tables》中有对于该问题的更详细说明，读者可以参阅和意会。总之，瞬表主要就是为了解决“Cyclic references between keys and values in weak tables prevent the elements inside a cycle from being collected, even if they are no longer reachable from outside”的问题。

^②瞬表是在 Lua 5.2 中引入的，Lua 5.1 依然存在我们描述的问题。

```
collectgarbage() --> hi
```

在本例中，我们首先创建一个带有 `__gc` 元方法元表的表。然后，抹去与这个表的唯一联系（全局变量），再强制进行一次完整的垃圾回收。在垃圾回收期间，Lua 语言发现表已经不再是可访问的了，因此调用表的析构器，也就是元方法 `__gc`。

Lua 语言中，析构器的一个微妙之处在于“将一个对象标记为需要析构”的概念。通过给对象设置一个具有非空 `__gc` 元方法的元表，就可以把这个对象标记为需要进行析构处理。如果不标记对象，那么对象就不会被析构。我们编写的大多数代码会正常运行，但会发生某些奇怪的行为，比如：

```
o = {x = "hi"}
mt = {}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage() --> (prints nothing)
```

这里，我们确实给对象 `o` 设置了元表，但是这个元表没有 `__gc` 元方法，因此对象没有被标记为需要进行析构处理。即使我们后续给元表增加了元方法 `__gc`，Lua 语言也发现不了这种赋值的特殊之处，因此不会把对象标记为需要进行析构处理。

正如我们所提到的，这很少会有问题。在设置元表后，很少会改变元方法。如果真的需要在后续设置元方法，那么可以给字段 `__gc` 先赋一个任意值作为占位符：

```
o = {x = "hi"}
mt = {__gc = true}
setmetatable(o, mt)
mt.__gc = function (o) print(o.x) end
o = nil
collectgarbage() --> hi
```

现在，由于元表有了 `__gc` 字段，因此对象会被正确地标记为需要析构处理。如果后续再设置元方法也不会有问题，只要元方法是一个正确的函数，Lua 语言就能够调用它。

当垃圾收集器在同一个周期中析构多个对象时，它会按照对象被标记为需要析构处理的顺序逆序调用这些对象的析构器。请考虑如下的示例，该示例创建了一个由带有析构器的对象所组成的链表：

```

mt = {__gc = function (o) print(o[1]) end}
list = nil
for i = 1, 3 do
    list = setmetatable({i, link = list}, mt)
end
list = nil
collectgarbage()
--> 3
--> 2
--> 1

```

第一个被析构的对象是 3，也就是最后一个被标记的对象。

一种常见的误解是认为正在被回收的对象之间的关联会影响对象析构的顺序。例如，有些人可能认为上例中的对象 2 必须在对象 1 之前被析构，因为存在从 2 到 1 的关联。但是，关联会形成环。所以，关联并不会影响析构器执行的顺序。

有关析构器的另一个微妙之处是复苏 (*resurrection*)。当一个析构器被调用时，它的参数是正在被析构的对象。因此，这个对象会至少在析构期间重新变成活跃的。笔者把这称为临时复苏 (*transient resurrection*)。在析构器执行期间，我们无法阻止析构器把该对象存储在全局变量中，使得该对象在析构器返回后仍然可访问，笔者把这称为永久复苏 (*permanent resurrection*)。

复苏必须是可传递的。考虑如下的代码：

```

A = {x = "this is A"}
B = {f = A}
setmetatable(B, {__gc = function (o) print(o.f.x) end})
A, B = nil
collectgarbage() --> this is A

```

B 的析构器访问了 A，因此 A 在 B 析构前不能被回收，Lua 语言在运行析构器之前必须同时复苏 B 和 A。

由于复苏的存在，Lua 语言会在两个阶段中回收具有析构器的对象。当垃圾收集器首次发现某个具有析构器的对象不可达时，垃圾收集器就把这个对象复苏并将其放入等待被析构的队列中。一旦析构器开始执行，Lua 语言就将该对象标记为已被析构。当下一次垃圾收集器又发现这个对象不可达时，它就将这个对象删除。如果想保证我们程序中的所有垃圾都被

真正地释放了的话，那么必须调用 `collectgarbage` 两次，第二次调用才会删除第一次调用中被析构的对象。

由于 Lua 语言在被析构对象上设置的标记，每一个对象的析构器都会精确地运行一次。如果一个对象直到程序运行结束还没有被回收，那么 Lua 语言就会在整个 Lua 虚拟机关闭后调用它的析构器。这种特性在 Lua 语言中实现了某种形式的 `atexit` 函数，即在程序终结前立即运行的函数。我们所要做的就是创建一个带有析构器的表，然后把它锚定在某处，例如锚定到全局表中：

```
local t = {__gc = function ()
    -- 'atexit' 的代码位于此处
    print("finishing Lua program")
end}
setmetatable(t, t)
_G["*AA*"] = t
```

另外一个有趣的技巧会允许程序在每次完成垃圾回收后调用指定的函数。由于析构器只运行一次，所以这种技巧是让每个析构器创建一个用来运行下一个析构器的新对象，参见示例 23.2。

示例 23.2 在每次 GC 后运行一个函数

```
do
    local mt = {__gc = function (o)
        -- 要做的工作
        print("new cycle")
        -- 为下一次垃圾收集创建新对象
        setmetatable({}, getmetatable(o))
    end}
    -- 创建第一个对象
    setmetatable({}, mt)
end

collectgarbage()    --> 一次垃圾收集
collectgarbage()    --> 一次垃圾收集
collectgarbage()    --> 一次垃圾收集
```

具有析构器的对象和弱引用表之间的交互也有些微妙。在每个垃圾收集周期内，垃圾收集器会在调用析构器前清理弱引用表中的值，在调用析构器之后再清理键。这种行为的原理在于我们经常使用带有弱引用键的表来保存对象的属性（参见23.3节），因此，析构器可能需要访问那些属性。不过，我们也会使用具有弱引用值的表来重用活跃的对象，在这种情况下，正在被析构的对象就不再有用了。

23.7 垃圾收集器

一直到Lua 5.0，Lua语言使用的都是一个简单的标记-清除（mark-and-sweep）式垃圾收集器（Garbage Collector, GC）。这种收集器又被称为“stop-the-world（全局暂停）”式的收集器，意味着Lua语言会时不时地停止主程序的运行来执行一次完整的垃圾收集周期（garbage-collection cycle）。每一个垃圾收集周期由四个阶段组成：标记（mark）、清理（cleaning）、清除（sweep）和析构（finalization）。

标记阶段把根结点集合（root set）标记为活跃，根结点集合由Lua语言可以直接访问的对象组成。在Lua语言中，这个集合只包括C注册表（在30.3.1节中我们会看到，主线程和全局环境都是在这个注册表中预定义的元素）。

保存在一个活跃对象中的对象是程序可达的，因此也会被标记为活跃（当然，在弱引用表中的元素不遵循这个规则）。当所有可达对象都被标记为活跃后，标记阶段完成。

在开始清除阶段前，Lua语言先执行清理阶段，在这个阶段中处理析构器和弱引用表。首先，Lua语言遍历所有被标记为需要进行析构、但又没有被标记为活跃状态的对象。这些没有被标记为活跃状态的对象会被标记为活跃（复苏，resurrected），并被放在一个单独的列表中，这个列表会在析构阶段用到。然后，Lua语言遍历弱引用表并从中移除键或值未被标记的元素。

清除阶段遍历所有对象（为了实现这种遍历，Lua语言把所有创建的对象放在一个链表中）。如果一个对象没有被标记为活跃，Lua语言就将其回收。否则，Lua语言清理标记，然后准备进行下一个清理周期。

最后，在析构阶段，Lua语言调用清理阶段被分离出的对象的析构器。

使用真正的垃圾收集器意味着Lua语言能够处理对象引用之间的环。在使用环形数据结构时，我们不需要花费额外的精力，它们会像其他数据一样被回收。

Lua 5.1 使用了增量式垃圾收集器（incremental collector）。这种垃圾收集器像老版的垃圾收集器一样执行相同的步骤，但是不需要在垃圾收集期间停止主程序的运行。相反，它与

解释器一起交替运行。每当解释器分配了一定数量的内存时，垃圾收集器也执行一小步（这意味着，在垃圾收集器工作期间，解释器可能会改变一个对象的可达性。为了保证垃圾收集器的正确性，垃圾收集器中的有些操作具有发现危险改动和纠正所涉及对象标记的内存屏障 [barrier]）。

Lua 5.2 引入了紧急垃圾收集（*emergency collection*）。当内存分配失败时，Lua 语言会强制进行一次完整的垃圾收集，然后再次尝试分配。这些紧急情况可以发生在 Lua 语言进行内存分配的任意时刻，包括 Lua 语言处于不一致的代码执行状态时，因此，这些收集动作不能运行析构器。

23.8 控制垃圾收集的步长（Pace）

通过函数 `collectgarbage` 可以对垃圾收集器进行一些额外的控制，该函数实际上是几个函数的集合体：第一个参数是一个可选的字符串，用来说明进行何种操作；有些选项使用一个整型作为第二个参数，称为 `data`。

第一个参数的选项包括如下七个。

"stop": 停止垃圾收集器，直到使用选项"restart" 再次调用 `collectgarbage`。

"restart": 重启垃圾收集器。

"collect": 执行一次完整的垃圾收集，回收和析构所有不可达的对象。这是默认的选项。

"step": 执行某些垃圾收集工作，第二个参数 `data` 指明工作量，即在分配了 `data` 个字节后垃圾收集器应该做什么。

"count": 以 KB 为单位返回当前已用内存数，该结果是一个浮点数，乘以 1024 得到的就是精确的字节数。该值包括了尚未被回收的死对象。

"setpause": 设置收集器的 `pause` 参数（间歇率）。参数 `data` 以百分比为单位给出要设定的新值：当 `data` 为 100 时，参数被设为 1 (100%)。

"setstepmul": 设置收集器的 `stepmul` 参数（步进倍率，`step multiplier`）。参数 `data` 给出新值，也是以百分比为单位。

两个参数 `pause` 和 `stepmul` 控制着垃圾收集器的角色。任何垃圾收集器都是使用 CPU 时间换内存空间。在极端情况下，垃圾收集器可能根本不会运行。但是，不耗费 CPU 时间

是以巨大的内存消耗为代价的。在另外一种极端的情况下，收集器可能每进行一次赋值就得运行一次完整的垃圾收集。程序能够使用尽可能少的内存，但是是以巨大的 CPU 消耗为代价的。`pause` 和 `stepmul` 的默认值正是试图在这两个极端之间找到的对大多数应用来说足够好的平衡点。不过，在某些情况下，还是值得试着对它们进行优化。

参数 `pause` 用于控制垃圾收集器在一次收集完成后等待多久再开始新的一次收集。当值为零时表示 Lua 语言在上一次垃圾回收结束后立即开始一次新的收集。当值为 200% 时表示在重启垃圾收集器前等待内存使用翻番。如果想使消耗更多的 CPU 时间换取更低的内存消耗，那么可以把这个值设得小一点。通常，我们应该把这个值设在 0 到 200% 之间。

参数 `stepmul` 控制对于每分配 1KB 内存，垃圾收集器应该进行多少工作。这个值越高，垃圾收集器使用的增量越小。一个像 100000000% 一样巨大的值会让收集器表现得像一个非增量的垃圾收集器。默认值是 200%。低于 100% 的值会让收集器运行得很慢，以至于可能一次收集也完不成^①。

函数 `collectgarbage` 的另外一些参数用来在垃圾收集器运行时控制它的行为。同样，对于大多数程序员来说，默认值已经足够好了，但是对于一些特殊的应用，用手工控制可能更好，游戏就经常需要这种类型的控制。例如，如果我们不想让垃圾收集在某些阶段运行，那么可以通过调用函数 `collectgarbage("stop")` 停止垃圾收集器，然后再调用 `collectgarbage("restart")` 重新启动垃圾收集器。在一些具有周期性休眠阶段的程序中，可以让垃圾收集器停止，然后在程序休眠期间调用 `collectgarbage("step", n)`。要设置在每一个休眠期间进行多少工作，要么为 `n` 实验性地选择一个恰当的值，要么把 `n` 设成零（意为最小的步长），然后在一个循环中调用函数 `collectgarbage` 直到休眠结束。

23.9 练习

练习 23.1：请编写一个实验证明为什么 Lua 语言需要实现瞬表（记得调用函数 `collectgarbage` 来强制进行一次垃圾收集）。如果可能的话，分别在 Lua 5.1 和 Lua 5.2/5.3 中运行你的代码来看看有什么不同。

练习 23.2：考虑 23.6 节的第一个例子，该示例创建了一个带有析构器的表，该析构器在执行时只是输出一条消息。如果程序没有进行过垃圾收集就退出会发生什么？如果程序调用了函数 `os.exit` 呢？如果程序由于出错而退出呢？

^①译者注：该参数与增量式垃圾收集算法有关，作者在原文中也言之不详，有兴趣的读者可以参考其他资料，在此译者也不再展开。

练习 23.3：假设要实现一个记忆表，该记忆表所针对函数的参数和返回值都是字符串。由于弱引用表不把字符串当作可回收对象，因此将这个记忆表标记为弱引用并不能使得其中的键值对能够被垃圾收集。在这种情况下，你该如何实现记忆呢？

练习 23.4：解释示例 23.3 中程序的输出。

示例 23.3 析构器和内存

```

local count = 0

local mt = {__gc = function () count = count - 1 end}
local a = {}

for i = 1, 10000 do
    count = count + 1
    a[i] = setmetatable({}, mt)
end

collectgarbage()
print(collectgarbage("count") * 1024, count)
a = nil
collectgarbage()
print(collectgarbage("count") * 1024, count)
collectgarbage()
print(collectgarbage("count") * 1024, count)

```

练习 23.5：对于这个练习，你需要至少一个使用很多内存的 Lua 脚本。如果你没有这样的脚本，那就写一个（一个创建表的循环就可以）。

- 使用不同的 pause 和 stepmul 运行脚本。它们的值是如何影响脚本的性能和内存使用的？如果把 pause 设成零会发什么？如果把 pause 设成 1000 会发什么？如果把 stepmul 设成零会发什么？如果把 stepmul 设成 1000000 会发什么？
- 调整你的脚本，使其能够完整地控制垃圾收集器。脚本应该让垃圾收集器停止运行，然后时不时地完成垃圾收集的工作。你能够使用这种方式提高脚本的性能么？

24

协程 (Coroutine)

我们并不经常需要用到协程，但是当需要的时候，协程会起到一种不可比拟的作用。协程可以颠倒调用者和被调用者的关系，而且这种灵活性解决了软件架构中被笔者称为“谁是老大 (who-is-the-boss)”或者“谁拥有主循环 (who-has-the-main-loop)”的问题。这正是对诸如事件驱动编程、通过构造器构建迭代器和协作式多线程等几个看上去并不相关的问题的泛化，而协程以简单和高效的方式解决了这些问题。

从多线程 (multithreading) 的角度看，协程 (coroutine) 与线程 (thread) 类似：协程是一系列的可执行语句，拥有自己的栈、局部变量和指令指针，同时协程又与其他协程共享了全局变量和其他几乎一切资源。线程与协程的主要区别在于，一个多线程程序可以并行运行多个线程，而协程却需要彼此协作地运行，即在任意指定的时刻只能有一个协程运行，且只有当正在运行的协程显式地要求被挂起 (suspend) 时其执行才会暂停。

在本章中，我们会学习 Lua 语言中的协程是如何运行的，同时也将学习如何使用协程来解决一系列的问题。

24.1 协程基础

Lua 语言中协程相关的所有函数都被放在表 `coroutine` 中。函数 `create` 用于创建新协程，该函数只有一个参数，即协程要执行的代码的函数 (协程体 (`body`))。函数 `create` 返回一个“`thread`”类型的值，即新协程。通常，函数 `create` 的参数是一个匿名函数，例如：

```
co = coroutine.create(function () print("hi") end)
print(type(co))      --> thread
```

一个协程有以下四种状态，即挂起（suspended）、运行（running）、正常（normal）和死亡（dead）。我们可以通过函数 `coroutine.status` 来检查协程的状态：

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

当一个协程被创建时，它处于挂起状态，即协程不会在被创建时自动运行。函数 `coroutine.resume` 用于启动或再次启动一个协程的执行，并将其状态由挂起改为运行：

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

如果在交互模式下运行上述代码，最好在最后一行加上一个分号来阻止输出函数 `resume` 的返回值。在上例中，协程体只是简单地打印了"hi" 后便终止了，然后协程就变成了死亡状态：

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

到目前为止，协程看上去也就是一种复杂的调用函数的方式。协程的真正强大之处在于函数 `yield`，该函数可以让一个运行中的协程挂起自己，然后在后续恢复运行。例如下面这个简单的示例：

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

其中，协程进行了一个循环，在循环中输出数字并在每次打印后挂起。当唤醒协程后，它就会开始执行直到遇到第一个 `yield`：

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

此时，如果我们查看协程状态，会发现协程处于挂起状态，因此可以再次恢复运行：

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

从协程的角度看，在挂起期间发生的活动都发生在协程调用 `yield` 期间。当我们唤醒协程时，函数 `yield` 才会最终返回，然后协程会继续执行直到遇到下一个 `yield` 或执行结束：

```

coroutine.resume(co)      --> co 1 2
coroutine.resume(co)      --> co 1 3
...
coroutine.resume(co)      --> co 10
coroutine.resume(co)      -- 不输出任何数据

```

在最后一次调用 `resume` 时，协程体执行完毕并返回，不输出任何数据。如果我们试图再次唤醒它，函数 `resume` 将返回 `false` 及一条错误信息：

```

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

```

请注意，像函数 `pcall` 一样，函数 `resume` 也运行在保护模式中。因此，如果协程在执行中出错，Lua 语言不会显示错误信息，而是将错误信息返回给函数 `resume`。

当协程 A 唤醒协程 B 时，协程 A 既不是挂起状态（因为不能唤醒协程 A），也不是运行状态（因为正在运行的协程是 B）。所以，协程 A 此时的状态就被称为正常状态。

Lua 语言中一个非常有用的机制是通过一对 `resume-yield` 来交换数据。第一个 `resume` 函数（没有对应等待它的 `yield`）会把所有的额外参数传递给协程的主函数：

```

co = coroutine.create(function (a, b, c)
    print("co", a, b, c + 2)
end)
coroutine.resume(co, 1, 2, 3)    --> co 1 2 5

```

在函数 `coroutine.resume` 的返回值中，第一个返回值为 `true` 时表示没有错误，之后的返回值对应函数 `yield` 的参数：

```

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

```

与之对应的是，函数 `coroutine.yield` 的返回值是对应的 `resume` 的参数：

```

co = coroutine.create (function (x)
    print("co1", x)
    print("co2", coroutine.yield())
end)

```

```
coroutine.resume(co, "hi")      --> co1  hi
coroutine.resume(co, 4, 5)       --> co2  4  5
```

最后，当一个协程运行结束时，主函数所返回的值都将变成对应函数 `resume` 的返回值：

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

我们很少在同一个协程中用到所有这些机制，但每种机制都有各自的用处。

虽然协程的概念很容易理解，但涉及的细节其实很多。因此，对于那些已经对协程有一定了解的读者来说，有必要在进行进一步学习前先理清一些细节。Lua 语言提供的是所谓的非对称协程 (*asymmetric coroutine*)，也就是说需要两个函数来控制协程的执行，一个用于挂起协程的执行，另一个用于恢复协程的执行。而其他一些语言提供的是对称协程 (*symmetric coroutine*)，只提供一个函数用于在一个协程和另一个协程之间切换控制权。

一些人将非对称协程称为 *semi-coroutines*。然而，其他人则用相同的术语半协程 (*semi-coroutine*) 表示协程的一种受限制版实现。在这种实现中，一个协程只能在它没有调用其他函数时才可以挂起，即在调用栈中没有挂起的调用时。换句话说，只有这种半协程的主函数才能让出执行权（Python 中的 `generator` 正是这种半协程的一个例子）。

与对称协程和非对称协程之间的区别不同，协程与 `generator`（例如 Python 中的）之间的区别很大。`generator` 比较简单，不足以实现某些最令人关心的代码结构，而这些代码结构可以使用完整功能的协程实现。Lua 语言提供了完整的、非对称的协程。对于那些更喜欢对称协程的用户而言，可以基于非对称协程实现对称协程（参见练习 24.6）。

24.2 哪个协程占据主循环

有关协程的最经典示例之一就是生产者-消费者问题。在生产者-消费者问题中涉及两个函数，一个函数不断地产生值（比如，从一个文件中读取），另一个函数不断地消费这些值（比如，将值写入另一个文件中）。这两个函数可能形式如下：

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

```

    send(x)          -- 发给消费者
end
end

function consumer ()
while true do
    local x = receive()      -- 接收来自生产者的值
    io.write(x, "\n")        -- 消费
end
end

```

为了简化这个示例，生产者和消费者都是无限循环的；不过，可以很容易地将其修改为没有数据需要处理时退出循环。这里的问题在于如何将 `send` 与 `receive` 匹配起来，也就是“谁占据主循环（who-has-the-main-loop）”问题的典型实例。其中，生产者和消费者都处于活跃状态，它们各自具有自己的主循环，并且都将对方视为一个可调用的服务（callable service）。对于这个特定的示例，可以很容易地修改其中一个函数的结构，展开它的循环使其成为一个被动的代理。不过，在其他的真实场景下，这样的代码结构改动可能会很不容易。

由于成对的 `resume-yield` 可以颠倒调用者与被调用者之间的关系，因此协程提供了一种无须修改生产者和消费者的代码结构就能匹配它们执行顺序的理想工具。当一个协程调用函数 `yield` 时，它不是进入了一个新函数，而是返回一个挂起的调用（调用的是函数 `resume`）。同样地，对函数 `resume` 的调用也不会启动一个新函数，而是返回一个对函数 `yield` 的调用。这种特性正好可以用于匹配 `send` 和 `receive`，使得双方都认为自己是主动方而对方是被动方（这也是笔者称之为 who-is-the-boss 问题的原因）。因此，`receive` 唤醒生产者的执行使其能生成一个新值，然后 `send` 则让出执行权，将生成的值传递给消费者：

```

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

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

```

当然，生产者现在必须运行在一个协程里：