

```
producer = coroutine.create(producer)
```

在这种设计中，程序通过调用消费者启动。当消费者需要新值时就唤醒生产者，生产者向消费者返回新值后挂起，直到消费者再次将其唤醒。因此，我们将这种设计称为消费者驱动 (*consumer-driven*) 式的设计。另一种方式则是使用生产者驱动 (*producer-driven*) 式的设计，其中消费者是协程。虽然上述两种设计思路看上去是相反的，但实际上它们的整体思想相同。

我们可以使用过滤器来扩展上述设计^①。过滤器位于生产者和消费者之间，用于完成一些对数据进行某种变换的任务。过滤器 (*filter*) 既是一个消费者又是一个生产者，它通过唤醒一个生产者来获得新值，然后又将变换后的值传递给消费者。例如，我们可以在前面代码中添加一个过滤器以实现在每行的起始处插入行号。参见示例 24.1。

示例 24.1 使用过滤器的生产者和消费者

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

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

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

function filter (prod)
```

^①译者注：参考 Pipe-And-Filter 管道-过滤器模式。

```

function producer()
    for line = 1, math.huge do
        local x = receive(prod) -- 接收新值
        x = string.format("%5d %s", line, x)
        send(x) -- 发送给消费者
    end
end

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

consumer(filter(producer()))

```

代码的最后一行只是简单地创建出所需的各个组件，将这些组件连接在一起，然后启动消费者。

如果读者在阅读了上例后想起了 POSIX 操作系统下的管道 (pipe)，那么这并非偶然。毕竟，协程是一种非抢占式 (non-preemptive) 多线程。使用管道时，每项任务运行在各自独立的进程中；而使用协程时，每项任务运行在各自独立的协程中。管道在写入者 (生产者) 和读取者 (消费者) 之间提供一个缓冲区，因此它们的相对运行速度可以存在一定差异。由于进程间切换的开销很高，所以这一点在使用管道的场景下非常重要。在使用协程时，任务切换的开销则小得多 (基本与函数调用相同)，因此生产者和消费者可以手拉手以相同的速度运行。

24.3 将协程用作迭代器

我们可以将循环迭代器视为生产者-消费者模式的一种特例：一个迭代器会生产由循环体消费的内容。因此，用协程来实现迭代器看上去就很合适。的确，协程为实现这类任务提供了一种强大的工具。同时，协程最关键的特性是能够颠倒调用者与被调用者之间的关系。有了这种特性，我们在编写迭代器时就无须担心如何保存连续调用之间的状态了。

为了说明这类用途，让我们来编写一个遍历指定数组所有排列的迭代器。要直接编写这种迭代器并不容易，但如果要编写一个递归函数来产生所有的排列则不是很难。思路很简单，只要依次将每个数组元素放到最后一个位置，然后递归地生成其余元素的所有排列即可。代码参见示例 24.2。

示例 24.2 一个生成排列的函数

```
function permgen (a, n)
    n = n or #a           -- 'n' 的默认大小是'a'
    if n <= 1 then         -- 只有一种组合
        printResult(a)
    else
        for i = 1, n do
            -- 把第i个元素当做最后一个
            a[n], a[i] = a[i], a[n]

            -- 生成其余元素的所有排列
            permgen(a, n - 1)

            -- 恢复第i个元素
            a[n], a[i] = a[i], a[n]
        end
    end
end
```

还需要定义一个合适的函数 `printResult` 来输出结果，并使用恰当的参数调用 `permgen`:

```
function printResult (a)
    for i = 1, #a do io.write(a[i], " ") end
    io.write("\n")
end

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

如果不行，就从头再来。如果重新开始，你会选择不同的策略？重新开始的话，你会选择--> 2 1 3 4 还是选择--> 1 2 3 4 呢？为什么呢？

当有了生成器后，将其转换为迭代器就很容易了。首先，我们把 `printResult` 改为 `yield`:

```
function permgen (a, n)
  n = n or #a
  if n <= 1 then
    coroutine.yield(a)
  else
    同前
```

然后，我们定义一个将生成器放入协程运行并创建迭代函数的工厂。迭代器只是简单地唤醒协程，让其产生下一个排列：

```
function permutations (a)
    local co = coroutine.create(function () permgen(a) end)
    return function () -- 迭代函数
        local code, res = coroutine.resume(co)
        return res
    end
end
```

有了上面的这些，在 **for** 循环中遍历一个数组的所有排列就非常简单了：

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

函数 `permutations` 使用了 Lua 语言中一种常见的模式，就是将唤醒对应协程的调用包装在一个函数中。由于这种模式比较常见，所以 Lua 语言专门提供了一个特殊的函数 `coroutine.`

`wrap` 来完成这个功能。与函数 `create` 类似，函数 `wrap` 也用来创建一个新的协程。但不同的是，函数 `wrap` 返回的不是协程本身而是一个函数，当这个函数被调用时会唤醒协程。与原始的函数 `resume` 不同，该函数的第一个返回值不是错误代码，当遇到错误时该函数会抛出异常。我们可以使用函数 `wrap` 改写 `permutations`：

```
function permutations (a)
    return coroutine.wrap(function () permgen(a) end)
end
```

通常，函数 `coroutine.wrap` 比函数 `coroutine.create` 更易于使用。它为我们提供了对于操作协程而言所需的功能，即一个唤醒协程的函数。不过，该函数缺乏灵活性，我们无法检查通过函数 `wrap` 所创建的协程的状态，也无法检查运行时的异常。

24.4 事件驱动式编程

虽然第一眼看上去不是特别明显，但实际上传统的事件驱动编程（event-driven programming）伴随的典型问题就衍生自 who-is-the-boss 问题。

在典型的事件驱动平台下，一个外部的实体向我们程序中所谓的事件循环（*event loop*）或运行循环（*run loop*）生成事件。这里，我们的代码很明显不是主循环。我们的程序变成了事件循环的附属品，使得我们的程序成为了一组无须任何显式关联的、相互独立的事件处理程序的集合。

再举一个更加具体的例子，假设有一个与 libuv 类似的异步 I/O 库，该库中有四个与我们的示例有关的函数：

```
lib.runloop();
lib.readline(stream, callback);
lib.writeline(stream, line, callback);
lib.stop();
```

第一个函数运行事件循环，在其中处理所有发生的事件并调用对应的回调函数。一个典型的事件驱动程序初始化某些机制然后调用这个函数，这个函数就变成了应用的主循环。第二个函数指示库从指定的流中读取一行，并在读取完成后带着读取的结果调用指定的回调函数。第三个函数与第二个函数类似，只是该函数写入一行。最后一个函数打破事件循环，通常用于结束程序。

示例 24.3 展示了上述库的一种实现。

示例 24.3 异步 I/O 库的简单实现

```

local cmdQueue = {} -- 挂起操作的队列
local lib = {}

function lib.readline (stream, callback)
    local nextCmd = function ()
        callback(stream:read())
    end
    table.insert(cmdQueue, nextCmd)
end

function lib.writeline (stream, line, callback)
    local nextCmd = function ()
        callback(stream:write(line))
    end
    table.insert(cmdQueue, nextCmd)
end

function lib.stop ()
    table.insert(cmdQueue, "stop")
end

function lib.runloop ()
    while true do
        local nextCmd = table.remove(cmdQueue, 1)
        if nextCmd == "stop" then
            break
        else
            nextCmd() -- 进行下一个操作
        end
    end
end

```

```
return lib
```

上述代码是一种简单而丑陋的实现。该程序的“事件队列（event queue）”实际上是一个由挂起操作组成的列表，当这些操作被异步调用时会产生事件。尽管很丑陋，但该程序还是完成了之前我们提到的功能，也使得我们无须使用真实的异步库就可以测试接下来的例子。

现在，让我们编写一个使用这个库的简单程序，这个程序把输入流中的所有行读取到一个表中，然后再逆序将其写到输出流中。如果使用同步 I/O，那么代码可能如下：

```
local t = {}
local inp = io.input() -- 输入流
local out = io.output() -- 输出流

for line in inp:lines() do
    t[#t + 1] = line
end

for i = #t, 1, -1 do
    out:write(t[i], "\n")
end
```

现在，让我们再使用异步 I/O 库按照事件驱动的方式重写这个程序，参见示例 24.4。

示例 24.4 使用事件驱动方式逆序一个文件

```
local lib = require "async-lib"

local t = {}
local inp = io.input()
local out = io.output()
local i

-- 写入行的事件处理函数
local function putline ()
    i = i - 1
    if i == 0 then -- 没有行了?
        lib.stop() -- 结束主循环
    else -- 写一行然后准备下一行
        out:write(t[i], "\n")
        lib.read(inp) -- 读取下一行
    end
end

lib.event("read", putline)
lib.read(inp)
```

```

        lib.writeline(out, t[i] .. "\n", putline)
    end
end

-- 读取行的事件处理函数
local function getline (line)
    if line then
        t[#t + 1] = line
        lib.readline(inp, getline)
    else
        i = #t + 1
        putline()
    end
end

lib.readline(inp, getline)
lib.runloop() -- 运行主循环

```

作为一种典型的事件驱动场景，由于主循环位于库中，因此所有的循环都消失了，这些循环被以事件区分的递归调用所取代。尽管我们可以通过使用闭包以后续传递风格 (Continuation-Passing Style, CPS) 进行改进，但仍然不能编写我们自己的循环。如果要这么做，那么必须通过递归来重写。

协程可以让我们使用事件循环来简化循环的代码，其核心思想是使用协程运行主要代码，即在每次调用库时将回调函数设置为唤醒协程的函数然后让出执行权。示例 24.5 使用这种思想实现了一个在异步 I/O 库上运行传统同步代码的示例：

示例 24.5 使用异步库运行同步代码

```

local lib = require "async-lib"

function run (code)
    local co = coroutine.wrap(function ()
        code()
        lib.stop() -- 结束时停止事件循环
    end)
    co() -- 启动协程

```

```

lib.runloop()      -- 启动事件循环
end

function putline (stream, line)
    local co = coroutine.running()      -- 调用协程
    local callback = (function () coroutine.resume(co) end)
    lib.writeline(stream, line, callback)
    coroutine.yield()
end

function getline (stream, line)
    local co = coroutine.running()      -- 调用协程
    local callback = (function (l) coroutine.resume(co, l) end)
    lib.readline(stream, callback)
    local line = coroutine.yield()
    return line
end

```

顾名思义，run 函数运行通过参数传入的同步代码。该函数首先创建一个协程来运行指定的代码，并在完成后停止事件循环。然后，该函数唤醒协程（协程会在第一次 I/O 操作时挂起），进入时间循环。

函数 getline 和 putline 模拟了同步 I/O。正如之前强调的，这两个函数都调用了恰当的异步函数，这些异步函数被当作唤醒调用协程的回调函数传入（请注意函数 coroutine.running 的用法，该函数用来访问调用协程）。之后，异步函数挂起，然后将控制权返回给事件循环。一旦异步操作完成，事件循环就会调用回调函数来唤醒触发异步函数的协程。

使用这个库，我们就可以在异步库上运行同步代码了。如下示例再次实现了逆序行的例子：

```

run(function ()
    local t = {}
    local inp = io.input()
    local out = io.output()

    while true do
        local line = getline(inp)

```

```

        if not line then break end
        t[#t + 1] = line
    end

    for i = #t, 1, -1 do
        putline(out, t[i] .. "\n")
    end
end)

```

除了使用了 `get`/`putline` 来进行 I/O 操作和运行在 `run` 以内，上述代码与之前的同步示例等价。在同步代码结构的外表之下，程序其实是以事件驱动模式运行的。同时，该程序与以更典型的事件驱动风格编写的程序的其他部分也完全兼容。

24.5 练习

练习 24.1： 使用生产者驱动 (*producer-driven*) 式设计重写 24.2 节中生产者-消费者的示例，其中消费者是协程，而生产者是主线程。

练习 24.2： 练习 6.5 要求编写一个函数来输出指定数组元素的所有组合。请使用协程把该函数修改为组合的生成器，该生成器的用法如下：

```

for c in combinations({"a", "b", "c"}, 2) do
    printResult(c)
end

```

练习 24.3： 在示例 24.5 中，函数 `getline` 和 `putline` 每一次被调用都会产生一个新的闭包。请使用记忆技术来避免这种资源浪费。

练习 24.4： 请为基于协程的库（示例 24.5）编写一个行迭代器，以便于使用 `for` 循环来读取一个文件。

练习 24.5： 你能否使用基于协程的库（示例 24.5）来同时运行多个线程？要做哪些修改呢？

练习 24.6： 请在 Lua 语言中实现一个 `transfer` 函数。如果读者认为唤醒-挂起 (*resume-yield*) 与调用-返回 (`call-return`) 类似，那么 `transfer` 就类似于 `goto`：它挂起运行中的协程，然后唤醒其他被当作参数给出的协程（提示：使用某种调度机制来控制协程。之后，`transfer` 会把执行权让给调度器以通知下一个协程运行，而调度器则唤醒下一个协程）。



25

反射 (Reflection)

反射是程序用来检查和修改其自身某些部分的能力。像 Lua 语言这样的动态语言支持几种反射机制：环境允许运行时观察全局变量；诸如 `type` 和 `pairs` 这样的函数允许运行时检查和遍历未知数据结构；诸如 `load` 和 `require` 这样的函数允许程序在自身中追加代码或更新代码。不过，还有很多方面仍然是缺失的：程序不能检查局部变量，开发人员不能跟踪代码的执行，函数也不知道是被谁调用的，等等。调试库（debug library）填补了上述的缺失。

调试库由两类函数组成：自省函数（*introspective function*）和钩子（*hook*）。自省函数允许我们检查一个正在运行中的程序的几个方面，例如活动函数的栈、当前正在执行的代码行、局部变量的名称和值。钩子则允许我们跟踪一个程序的执行。

虽然名字里带有“调试”的字眼，但调试库提供的并不是 Lua 语言的调试器（`debugger`）。不过，调试库提供了编写我们自己的调试器所需的不同层次的所有底层机制。

调试库与其他库不同，必须被慎重地使用。首先，调试库中的某些功能的性能不高。其次，调试库会打破语言的一些固有规则，例如不能从一个局部变量的词法定界范围外访问这个局部变量。虽然调试库作为标准库直接可用，但笔者建议在使用调试库的代码段中显式地加载调试库。



25.1 自省机制 (Introspective Facility)

调试库中主要的自省函数是 `getinfo`, 该函数的第一个参数可以是一个函数或一个栈层次。当为某个函数 `foo` 调用 `debug.getinfo(foo)` 时, 该函数会返回一个包含与该函数有关的一些数据的表。这个表可能具有以下字段。

source: 该字段用于说明函数定义的位置。如果函数定义在一个字符串中 (通过调用 `load`), 那么 `source` 就是这个字符串; 如果函数定义在一个文件中, 那么 `source` 就是使用 @ 作为前缀的文件名。

short_src: 该字段是 `source` 的精简版本 (最多 60 个字符), 对于错误信息十分有用。

linedefined: 该字段是该函数定义在源代码中第一行的行号。

lastlinedefined: 该字段是该函数定义在源代码中最后一行的行号。

what: 该字段用于说明函数的类型。如果 `foo` 是一个普通的 Lua 函数, 则为 "Lua"; 如果是一个 C 函数, 则为 "C"; 如果是一个 Lua 语言代码段的主要部分, 则为 "main"。

name: 该字段是该函数的一个适当的名称, 例如保存该函数的全局变量的名称。

namewhat: 该字段用于说明上一个字段^①的含义, 可能是 "global"、"local"、"method"、"field" 或 "" (空字符串)。空字符串表示 Lua 语言找不到该函数的名称。

nups: 该字段是该函数的上值的个数。

nparams: 该字段是该函数的参数个数。

isvararg: 该字段表明该函数是否为可变长参数函数 (一个布尔值)。

activelines: 该字段是一个包含该函数所有活跃行的集合。活跃行 (*active line*) 是指除空行和只包含注释的行外的其他行 (该字段的典型用法是用于设置断点。大多数调试器不允许在活跃行外设置断点, 因为非活跃行是不可达的)。

func: 该字段是该函数本身。

当 `foo` 是一个 C 函数时, Lua 语言没有多少关于该函数的信息。对于这种函数, 只有字段 `what`、`name`、`namewhat`、`nups` 和 `func` 是有意义的。

^①译者注: 即 `name` 字段。



当使用一个数字 n 作为参数调用函数 `debug.getinfo(n)` 时，可以得到有关相应栈层次上活跃函数的数据。栈层次 (*stack level*) 是一个数字，代表某个时刻上活跃的特定函数。调用 `getinfo` 的函数 A 的层次是 1，而调用 A 的函数的层次是 2，以此类推（层次 0 是 C 函数 `getinfo` 自己）。如果 n 大于栈中活跃函数的数量，那么函数 `debug.getinfo` 返回 `nil`。当通过带有栈层次的 `debug.getinfo` 查询一个活跃函数时，返回的表中还有两个额外字段：`currentline`，表示当前该函数正在执行的代码所在的行；`istailcall`（一个布尔值），如果为真则表示函数是被尾调用所调起（在这种情况下，函数的真实调用者不再位于栈中）。

字段 `name` 有些特殊。请注意，由于函数在 Lua 语言中是第一类值，因此函数既可以没有名称也可以有多个名称。Lua 语言会通过检查调用该函数的代码来看函数是如何被调用的，进而尝试找到该函数的名称。这种方法只有在以一个数字为参数调用 `getinfo` 时才会起作用，即我们只能获取关于某一具体调用的信息。

函数 `getinfo` 的效率不高。Lua 语言以一种不影响程序执行的形式来保存调试信息，至于获取这些调试信息的效率则是次要的。为了实现更好的性能，函数 `getinfo` 有一个可选的第二参数，该参数用于指定希望获取哪些信息。通过这个参数，函数 `getinfo` 就不会浪费时间去收集用户不需要的数据。这个参数是一个字符串，其中每个字母代表选择一组字段，如下表所示：

<code>n</code>	选择 <code>name</code> 和 <code>namewhat</code>
<code>f</code>	选择 <code>func</code>
<code>S</code>	选择 <code>source</code> 、 <code>short_src</code> 、 <code>what</code> 、 <code>linedefined</code> 和 <code>lastlinedefined</code>
<code>l</code>	选择 <code>currentline</code>
<code>L</code>	选择 <code>activelines</code>
<code>u</code>	选择 <code>nup</code> 、 <code>nparams</code> 和 <code>isvararg</code>

下面这个函数演示了函数 `debug.getinfo` 的用法，它打印出了活跃栈的栈回溯：

```
function traceback ()
    for level = 1, math.huge do
        local info = debug.getinfo(level, "Sl")
        if not info then break end
        if info.what == "C" then -- 是否是C函数?
            print(string.format("%d\tC function", level))
        else -- Lua函数
            print(string.format("%d\t[%s]:%d", level,
```



```

        info.short_src, info.currentline))
end
end
end

```

要改进这个函数并不难，只需要让函数 `getinfo` 返回更多数据即可。事实上，调试库也提供了这样一个改进版本，即函数 `traceback`。与我们的版本不同的是，函数 `debug.traceback` 不会打印结果，而是返回一个（可能会很长的）包含栈回溯的字符串：

```

> print(debug.traceback())
stack traceback:
      [C]: in ?
stdin:1: in main chunk
[C]: in ?

```

25.1.1 访问局部变量

我们可以通过函数 `debug.getlocal` 来检查任意活跃函数的局部变量。该函数有两个参数，一个是要查询函数的栈层次，另一个是变量的索引。该函数返回两个值，变量名和变量的当前值。如果变量索引大于活跃变量的数量，那么函数 `getlocal` 返回 `nil`。如果栈层次无效，则会抛出异常（我们可以使用函数 `debug.getinfo` 来检查栈层次是否有效）。

Lua 语言按局部变量在函数中的出现顺序对它们进行编号，但编号只限于在函数当前作用域中活跃的变量。例如，考虑如下的代码：

```

function foo (a, b)
    local x
    do local c = a - b end
    local a = 1
    while true do
        local name, value = debug.getlocal(1, a)
        if not name then break end
        print(name, value)
        a = a + 1
    end
end

```

调用 `foo(10, 20)` 会输出：



```
a      10
b      20
x      nil
a      4
```

索引为 1 的变量是 `a`（第一个参数），索引为 2 的变量 `b`，索引为 3 的变量是 `x`，索引为 4 的变量是内层的 `a`。在 `getlocal` 被调用的时候，`c` 已经离开了作用域，而 `name` 和 `value` 还未出现于作用域内（请注意，局部变量只在初始化后才可见）。

从 Lua 5.2 开始，值为负的索引获取可变长参数函数的额外参数，索引 -1 指向第一个额外参数。此时，变量的名称永远是 "`(*vararg)`"。

我们还可以通过函数 `debug.setlocal` 改变局部变量的值，该函数的前两个参数与 `getlocal` 相同，分别是栈层次和变量索引，而第三个参数是该局部变量的新值。该函数的返回值是变量名，如果变量索引超出了范围则返回 `nil`。

25.1.2 访问非局部变量

调试库还提供了函数 `getupvalue`，该函数允许我们访问一个被 Lua 函数所使用的非局部变量。与局部变量不同，被一个函数所引用的非局部变量即使在引用它的函数已经不活跃的情况下也会一直存在（毕竟这就是闭包的实质）。因此，函数 `getupvalue` 的第一个参数不是栈层次，而是一个函数（更确切地说，是一个闭包）。函数 `getupvalue` 的第二个参数是变量索引，Lua 语言按照函数引用非局部变量的顺序对它们编号，但由于一个函数不能用同一名称访问两个非局部变量，所以这个顺序是无关紧要的。

我们还可以通过函数 `debug.setupvalue` 更新非局部变量的值。就像读者可能预想的一样，该函数有三个参数：一个闭包、一个变量索引和一个新值。与函数 `setlocal` 一样，该函数返回变量名，如果变量索引超出范围则返回 `nil`。

示例 25.1 演示了如何通过变量名访问一个函数中变量的值。

示例 25.1 获取变量的值

```
function getvarvalue (name, level, isenv)
    local value
    local found = false

    level = (level or 1) + 1
```



```

-- 尝试局部变量
for i = 1, math.huge do
    local n, v = debug.getlocal(level, i)
    if not n then break end
    if n == name then
        value = v
        found = true
    end
end
if found then return "local", value end

-- 尝试非局部变量
local func = debug.getinfo(level, "f").func
for i = 1, math.huge do
    local n, v = debug.getupvalue(func, i)
    if not n then break end
    if n == name then return "upvalue", v end
end

if isenv then return "noenv" end      -- 避免循环

-- 没找到; 从环境中获取值
local _, env = getvarvalue("_ENV", level, true)
if env then
    return "global", env[name]
else          -- 没有有效的_ENV
    return "noenv"
end

```

用法如下：

```

> local a = 4; print(getvarvalue("a"))  --> local    4
> a = "xx"; print(getvarvalue("a"))     --> global    xx

```



参数 `level` 指明在哪个栈层次中寻找函数，1（默认值）意味着直接的调用者^①。代码中多加的 1 将层次纠正为包括 `getvarvalue` 自己。笔者稍后会解释参数 `isenv`。

该函数首先查找局部变量。如果有多个局部变量的名称与给定的名称相同，则获取具有最大索引的那个局部变量。因此，函数必须执行完整个循环。如果找不到指定名称的局部变量，那么就查找非局部变量。为了遍历非局部变量，该函数使用 `debug.getinfo` 函数获取调用闭包，然后遍历非局部变量。最后，如果还是找不到指定名字的非局部变量，就检索全局变量：该函数递归地调用自己来访问合适的 `_ENV` 变量并在相应环境中查找指定的名字。

参数 `isenv` 避免了一个诡异的问题。该参数用于说明我们是否处于一个从 `_ENV` 变量中查询全局名称的递归调用中。一个不使用全局变量的函数可能没有上值 `_ENV`。在这种情况下，如果我们试图把 `_ENV` 当作全局变量来查询，那么由于我们需要 `_ENV` 来得到其自身的值，所以可能会陷入无限递归循环。因此，当 `isenv` 为真且函数 `getvarvalue` 找不到局部变量或上值时，`getvarvalue` 就不应该再尝试全局变量。

25.1.3 访问其他协程

调试库中的所有自省函数都能够接受一个可选的协程作为第一个参数，这样就可以从外部来检查这个协程。例如，考虑如下的示例：

```
co = coroutine.create(function ()
    local x = 10
    coroutine.yield()
    error("some error")
end)

coroutine.resume(co)
print(debug.traceback(co))
```

对函数 `traceback` 的调用作用在协程 `co` 上，结果如下：

```
stack traceback:
[C]: in function 'yield'
temp:3: in function <temp:1>
```

^①译者注：`debug.getlocal` 和 `debug.getinfo` 栈层次为 0 时表示其自己，栈层次为 1 时表示调用它们的函数。



由于协程和主程序运行在不同的栈上，所以回溯没有跟踪到对函数 `resume` 的调用。

当协程引发错误时并不会进行栈展开，这就意味着可以在错误发生后检查错误。继续上面的示例，如果再次唤醒协程，它会提示引起了错误：

```
print(coroutine.resume(co))      --> false  temp:4: some error
```

现在，如果输出栈回溯，会得到这样的结果：

```
stack traceback:
```

```
[C]: in function 'error'  
temp:4: in function <temp:1>
```

即使在错误发生后，也可以检查协程中的局部变量：

```
print(debug.getlocal(co, 1, 1))      --> x = 10
```

25.2 钩子 (Hook)

调试库中的钩子机制允许用户注册一个钩子函数，这个钩子函数会在程序运行中某个特定事件发生时被调用。有四种事件能够触发一个钩子：

- 每当调用一个函数时产生的 `call` 事件；
- 每当函数返回时产生的 `return` 事件；
- 每当开始执行一行新代码时产生的 `line` 事件；
- 执行完指定数量的指令后产生的 `count` 事件。(这里的指令指的是内部操作码，在16.2节中对其有简单的描述。)

Lua 语言用一个描述导致钩子函数被调用的事件的字符串为参数来调用钩子函数，包括"call"（或"tail call"）、"return"、"line" 或 "count"。对于 `line` 事件来说，还有第二个参数，即新行号。我们可以在钩子函数内部调用函数 `debug.getinfo` 来获取更多的信息。

要注册一个钩子，需要用两个或三个参数来调用函数 `debug.sethook`：第一个参数是钩子函数，第二个参数是描述要监控事件的掩码字符串，第三个参数是一个用于描述以何种频率获取 `count` 事件的可选数字。如果要监控 `call`、`return` 和 `line` 事件，那么需要把这几个事件的首字母（c、r 或 l）放入掩码字符串。如果要监控 `count` 事件，则只需要在第三个参数中指定一个计数器。如果要关闭钩子，只需不带任何参数地调用函数 `sethook` 即可。



作为一个简单的示例，以下代码安装了一个简单的跟踪器（primitive tracer），它会输出解释器执行的每一行代码：

```
debug.sethook(print, "l")
```

这句调用只是简单地把函数 `print` 安装为一个钩子函数，并告诉 Lua 语言在 `line` 事件发生时调用它。一个更精巧的跟踪器可以使用函数 `getinfo` 获取当前文件名并添加到输出中：

```
function trace (event, line)
```

```
    local s = debug.getinfo(2).short_src
```

```
    print(s .. ":" .. line)
```

```
end
```

```
debug.sethook(trace, "l")
```

与钩子一起被使用的一个很有用的函数是 `debug.debug`。这个简单的函数可以提供一个能够执行任意 Lua 语言命令的提示符，其等价于如下的代码：

```
function debug1 ()
```

```
    while true do
```

```
        io.write("debug> ")
```

```
        local line = io.read()
```

```
        if line == "cont" then break end
```

```
        assert(load(line))()
```

```
    end
```

```
end
```

当用户输入“命令” `cont` 时，函数返回。这种标准的实现十分简单，并且在全局环境中运行命令，位于正在被调试代码的定界范围之外。练习 25.4 中讨论了一种更好的实现。

25.3 调优（Profile）

除了调试，反射的另外一个常见用法是用于调优，即程序使用资源的行为分析。对于时间相关的调优，最好使用 C 接口，因为每次钩子调用函数开销太大从而可能导致测试结果无效。不过，对于计数性质的调优，Lua 代码就可以做得很好。在本节中，我们将开发一个原始的性能调优工具（profiler）来列出程序执行的每个函数的调用次数。



性能调优工具的主要数据结构是两个表，其中一个表将函数和它们的调用计数关联起来，另一个表关联函数和函数名。这两个表的索引都是函数自身：

```
local Counters = {}
local Names = {}
```

我们可以在性能分析完成后再获取函数的名称，但是如果能在一个函数 F 处于活动状态时获取其名称可能会得到更好的结果。这是因为，在函数 F 处于活动状态时，Lua 语言可以通过分析正在调用函数 F 的代码来找出函数 F 的名称。

现在，我们定义一个钩子函数，该钩子函数的任务是获取当前正在被调用的函数，并递增相应的计数器，再收集函数名。代码参见示例 25.2。

示例 25.2 用于计算调用次数的钩子

```
local function hook ()
    local f = debug.getinfo(2, "f").func
    local count = Counters[f]
    if count == nil then      -- 'f'第一次被调用?
        Counters[f] = 1
        Names[f] = debug.getinfo(2, "Sn")
    else          -- 只需递增计数器即可
        Counters[f] = count + 1
    end
end
```

接下来，运行带有钩子的程序。假设我们要分析的程序位于一个文件中，且用户通过参数把该文件名传递给性能分析器，如下：

```
% lua profiler main-prog
```

这样，性能分析器就可以从 `arg[1]` 中得到文件名、设置钩子并运行文件：

```
local f = assert(loadfile(arg[1]))
debug.sethook(hook, "c")  -- 设置call事件的钩子
f()                      -- 运行主程序
debug.sethook()           -- 关闭钩子
```

最后一步是显示结果。示例 25.3 中的函数 `getname` 为每个函数生成一个函数名。



示例 25.3 获取一个函数的函数名

```

function getname (func)
    local n = Names[func]
    if n.what == "C" then
        return n.name
    end
    local lc = string.format("[%s]:%d", n.short_src, n.linedefined)
    if n.what ~= "main" and n.namewhat ~= "" then
        return string.format("%s (%s)", lc, n.name)
    else
        return lc
    end
end

```

由于 Lua 语言中的函数名并不是特别确定，所以我们给每个函数再加上位置信息，以 *file:line* 这样的形式给出。如果一个函数没有名称，那么就只使用它的位置。如果函数是 C 函数，那么就只使用它的名称（因为没有位置）。在上述函数定义后，我们输出每个函数及其计数器的值：

```

for func, count in pairs(Counters) do
    print(getname(func), count)
end

```

如果把这个性能调优工具用于第 19 章中开发的马尔可夫链算法示例的话，会得到大致如下的结果：

```

[markov.lua]:4 884723
write 10000
[markov.lua]:0 1
read   31103
sub    884722
...

```

这个结果意味着第 4 行的匿名函数（在 `allwords` 中定义的迭代函数）被调用了 884723 次，函数 `write` (`io.write`) 被调用了 10000 次，等等。

对于这个性能分析器，还有几个地方可以改进。例如，可以对输出进行排序、打印更易读的函数名和美化输出格式等。不过，这个原始的性能分析器本身已经是可用的了。

25.4 沙盒 (Sandbox)

在22.6节中，我们已经看到过，利用函数 `load` 在受限的环境中运行 Lua 代码是非常简单的。由于 Lua 语言通过库函数完成所有与外部世界的通信，因此一旦移除了这些函数也就排除了一个脚本能够影响外部环境的可能。不过尽管如此，我们仍然可能会被消耗大量 CPU 时间或内存的脚本进行拒绝服务（DoS）攻击。反射，以调试钩子的形式，提供了一种避免这种攻击的有趣方式。

首先，我们使用 `count` 事件钩子来限制一段代码能够执行的指令数。示例 25.4 展示了一个在沙盒中运行指定文件的程序。

示例 25.4 一个使用钩子的简单沙盒

```
local debug = require "debug"

-- 最大能够执行的"steps"
local steplimit = 1000

local count = 0      -- 计数器

local function step ()
    count = count + 1
    if count > steplimit then
        error("script uses too much CPU")
    end
end

-- 加载
local f = assert(loadfile(arg[1], "t", {}))

debug.sethook(step, "", 100)  -- 设置钩子
```

```
f() -- 运行文件
```

这个程序加载了指定的文件，设置了钩子，然后运行文件。该程序把钩子设置为监听 count 事件，使得 Lua 语言每执行 100 条指令就调用一次钩子函数。钩子（函数 step）只是递增一个计数器，然后检查其是否超过了某个固定的限制。这样做之后还会有问题么？

当然有问题。我们还必须限制所加载的代码段的大小：一段很长的代码只要被加载就可能耗尽内存。另一个问题是，程序可以通过少量指令消耗大量的内存。例如：

```
local s = "123456789012345"
for i = 1, 36 do s = s .. s end
```

上述的几行代码用不到 150 行的指令就试图创建出一个 1T 字节的字符串。显然，单纯限制指令数量和程序大小是不够的。

一种改进是检查和限制函数 step 使用的内存，参见示例 25.5。

示例 25.5 控制内存使用

```
-- 最大能够使用的内存（单位KB）
local memlimit = 1000

-- 最大能够执行的"steps"
local steplimit = 1000

local function checkmem ()
    if collectgarbage("count") > memlimit then
        error("script uses too much memory")
    end
end

local count = 0
local function step ()
    checkmem()
    count = count + 1
    if count > steplimit then
        error("script uses too much CPU")
    end
end
```

```
end
```

同前

由于通过少量指令就可以消耗很多内存，所以我们应该设置一个很低的限制或以很小的步进来调用钩子函数。更具体地说，一个程序用 40 行以内的指令就能把一个字符串的大小增加上千倍。因此，我们要么以比 40 条指令更高的频率调用钩子，要么把内存限制设为我们能够承受的最大值的一千分之一。笔者可能两种方式都会采用。

一个微妙的问题是字符串标准库。我们可以对字符串调用该库中的所有函数。因此，即使环境中没有这些函数，我们也可以调用它们；字符串常量把它们“走私”到了我们的沙盒中。字符串标准库中没有函数能够影响外部世界，但是它们绕过了我们的指令计数器（一个对 C 函数的调用相当于 Lua 语言中的一条指令）。字符串标准库中的有些函数对于 DoS 攻击而言可能会非常危险。例如，调用 ("x"):rep(2^30) 在一步之内就吞噬了 1GB 的内存。又如，在笔者的新机器上，Lua 5.2 耗费了 13 分钟才运行完下述代码：

```
s = "0123456789012345678901234567890123456789"
s:find("%.%.%.%.%.%.%.%.%x")
```

一种限制对字符串标准库访问的有趣方式是使用 call 钩子。每当有函数被调用时，我们就检查函数调用是不是合法的。示例 25.6 实现了这种思路。

示例 25.6 使用钩子阻止对未授权函数的访问

```
local debug = require "debug"
-- 最大能够执行的"steps"
local steplimit = 1000
-- 计数器
local count = 0
-- 设置授权的函数
local validfunc = {
    [string.upper] = true,
    [string.lower] = true,
    ...
}
```

```

local function hook (event)
    if event == "call" then
        local info = debug.getinfo(2, "fn")
        if not validfunc[info.func] then
            error("calling bad function: " .. (info.name or "?"))
        end
    end
    count = count + 1
    if count > steplimit then
        error("script uses too much CPU")
    end
end

-- 加载代码段
local f = assert(loadfile(arg[1], "t", {}))

debug.sethook(hook, "", 100)      -- 设置钩子

f()      -- 运行代码段

```

在上述代码中，表 `validfunc` 表示一个包含程序所能够调用的函数的集合。函数 `hook` 使用调试库来访问正在被调用的函数，然后检查函数是否在集合 `validfunc` 中。

对于任何一种沙盒的实现而言，很重要的一点是沙盒内允许使用哪些函数。用于数据描述的沙盒可以限制所有或大部分函数；其他的沙盒则需要更加宽容，也许应该对某些函数提供它们自己带限制的实现（例如，被限制只能处理小代码段的 `load`、只能访问固定目录的文件操作或只能对小对象使用的模式匹配）。

我们绝不考虑移除哪些函数，而是应该思考增加哪些函数。对于每一个要增加的函数，必须仔细考虑函数可能的弱点，这些弱点可能隐藏得很深。根据经验，所有数学标准库中的函数都是安全的。字符串库中的大部分也是安全的，只要小心涉及资源消耗的那些函数即可。调试库和模块库则不靠谱，它们中的几乎全部函数都是危险的。函数 `setmetatable` 和 `getmetatable` 同样很微妙：首先，它们可以访问别人访问不了的值；其次，它们允许创建带有析构器的表，在析构器中可以安装各种各样的“时间炸弹（time bomb）”（当表被垃圾回收时，代码可能在沙盒外被执行）。

25.5 练习

练习 25.1：改进 `getvarvalue`（示例 25.1），使之能处理不同的协程（与调试库中的函数 `debug` 类似）。

练习 25.2：请编写一个与函数 getvarvalue（示例 25.1）类似的 setvarvalue。

练习 25.3：请编写函数 `getvarvalue`（示例 25.1）的另一个版本，该函数返回一个包括调用函数可见的所有变量的表（返回的表中不应该包含环境中的变量，而应该从原来的环境中继承这些变量）。

练习 25.4：请编写一个函数 `debug.debug` 的改进版，该函数在调用 `debug.debug` 函数的词法定界中运行指定的命令（提示：在一个空环境中运行命令，并使用 `_index` 元方法让函数 `getvarvalue` 进行所有的变量访问）。

练习 25.5：改进上例，使之也能处理更新操作。

练习 25.6：实现 25.3 节中开发的基本性能调优工具中的一些建议的改进。

练习 25.7：请编写一个用于断点的库，这个库应该包括至少两个函数：

```
setbreakpoint(function, line)    --> 返回处理句柄  
removebreakpoint(handle)
```

我们通过一个函数和对应函数中的一行来指定断点（breakpoint）。当程序命中断点时，这个库应该调用函数 `debug.debug`（提示：对于基本的实现，使用一个检查是否位于断点中的 `line` 事件钩子即可；要改进性能，可以使用一个 `call` 事件钩子来跟踪执行并只在程序运行到目标函数中时再启动 `line` 事件钩子）。

练习 25.8：示例 25.6 中沙盒的问题之一在于沙盒中的代码不能调用其自身的函数。请问如何纠正这个问题？

26

小插曲：使用协程实现多线程

在本章这个小插曲中，我们将学习如何利用协程实现多线程。

正如我们此前所看到的，协程能够实现一种协作式多线程（collaborative multithreading）。每个协程都等价于一个线程。一对 `yield-resume` 可以将执行权在不同线程之间切换。不过，与普通的多线程的不同，协程是非抢占的。当一个协程正在运行时，是无法从外部停止它的。只有当协程显式地要求时（通过调用函数 `yield`）它才会挂起执行。对于有些应用而言，这并没有问题，而对于另外一些应用则不行。当不存在抢占时，编程简单得多。由于在程序中所有的线程间同步都是显式的，所以我们无须为线程同步问题抓狂，只需要确保一个协程只在它的临界区（critical region）之外调用 `yield` 即可。

不过，对于非抢占式多线程来说，只要有一个线程调用了阻塞操作，整个程序在该操作完成前都会阻塞。对于很多应用程序来说，这种行为是无法接受的，而这也正是导致许多程序员不把协程看作传统多线程的一种实现的原因。接下来，我们会用一个有趣（且显而易见）的方法来解决这个问题。

让我们假设一个典型的多线程场景：我们希望通过 HTTP 下载多个远程文件。为了下载多个远程文件，我们必须先知道如何下载一个远程文件。在本例中，我们将使用 `LuaSocket` 标准库。要下载一个文件，必须先打开一个到对应站点的连接，然后发送下载文件的请求，接收文件（按块），最后关闭连接。在 `Lua` 语言中，可以按以下步骤来完成这项任务。首先，加载 `LuaSocket` 库：

```
local socket = require "socket"
```

然后，定义主机和要下载的文件。在本例中，我们从 Lua 语言官网下载 Lua 5.3 的手册：

```
host = "www.lua.org"
file = "/manual/5.3/manual.html"
```

接下来，打开一个 TCP 连接，连接到该站点的 80 端口（HTTP 协议的默认端口）：

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

这步操作返回一个连接对象，可以用它来发送下载文件的请求：

```
local request = string.format(
    "GET %s HTTP/1.0\r\nhost: %s\r\n\r\n",
    file, host)
c:send(request)
```

接下来，以 1KB 为一块读取文件，并将每块写入到标准输出中：

```
repeat
    local s, status, partial = c:receive(2^10)
    io.write(s or partial)
until status == "closed"
```

函数 `receive` 要么返回它读取到的字符串，要么在发生错误时返回 `nil` 外加错误码 (`status`) 及出错前读取到的内容 (`partial`)。当主机关闭连接时，把输入流中剩余的内容打印出来，然后退出接收循环。

下载完文件后，关闭连接：

```
c:close()
```

既然我们知道了如何下载一个文件，那么再回到下载多个文件的问题上。最简单的做法是逐个地下载文件。不过，这种串行的做法太慢了，它只能在下载完一个文件后再下载一个文件。当读取一个远程文件时，程序把大部分的时间耗费在了等待数据到达上。更确切地说，程序将时间耗费在了对 `receive` 的阻塞调用上。因此，如果一个程序能够同时并行下载所有文件的话，就会快很多。当一个连接没有可用数据时，程序便可以从其他连接读取数据。很明显，协程为构造这种并发下载的代码结构提供了一种简便的方式。我们可以为每个下载任务创建一个新线程，当一个线程无可用数据时，它就可以将控制权传递给一个简单的调度器 (`dispatcher`)，这个调度器再去调用其他的线程。

在用协程重写程序前，我们先把之前下载的代码重写成一个函数。如示例 26.1 所示。

示例 26.1 下载 Web 页面的函数

```

function download (host, file)
    local c = assert(socket.connect(host, 80))
    local count = 0      -- 计算读取的字节数
    local request = string.format(
        "GET %s HTTP/1.0\r\nhost: %s\r\n\r\n", file, host)
    c:send(request)
    while true do
        local s, status = receive(c)
        count = count + #s
        if status == "closed" then break end
    end
    c:close()
    print(file, count)
end

```

由于我们对远程文件的内容并不感兴趣，所以不需要将文件内容写入到标准输出中，只要计算并输出文件大小即可。（多个线程同时读取多个文件时，输出的结果也是乱的。）

在新版代码中，我们使用一个辅助函数 `receive` 从连接接收数据。在串行的下载方式中，`receive` 的代码如下：

```

function receive (connection)
    local s, status, partial = connection:receive(2^10)
    return s or partial, status
end

```

在并行的实现中，这个函数在接收数据时不能阻塞。因此，在没有足够的可用数据时，该函数会挂起，如下：

```

function receive (connection)
    connection:settimeout(0)      -- 不阻塞
    local s, status, partial = connection:receive(2^10)
    if status == "timeout" then
        coroutine.yield(connection)
    end
    return s or partial, status
end

```

```
end
```

调用 `settimeout(0)` 使得后续所有对连接进行的操作不会阻塞。如果返回状态为“timeout”（超时），就表示该操作在返回时还未完成。此时，线程就会挂起。传递给 `yield` 的非假参数通知调度器线程仍在执行任务中。请注意，即使在超时的情况下，连接也会返回超时前已读取到的内容，也就是变量 `partial` 中的内容。

示例 26.2 展示了调度器及一些辅助代码。

示例 26.2 调度器

```
tasks = {}      -- 所有活跃任务的列表

function get (host, file)
    -- 为任务创建协程
    local co = coroutine.wrap(function ()
        download(host, file)
    end)
    -- 将其插入列表
    table.insert(tasks, co)
end

function dispatch ()
    local i = 1
    while true do
        if tasks[i] == nil then    -- 没有其他的任务了?
            if tasks[1] == nil then -- 列表为空?
                break    -- 从循环中退出
            end
            i = 1           -- 否则继续循环
        end
        local res = tasks[i]()
        if not res then    -- 任务结束?
            table.remove(tasks, i)
        else
            i = i + 1    -- 处理下一个任务
        end
    end
end
```



```
    end  
end
```

表 tasks 为调度器保存着所有正在运行中的线程的列表。函数 get 保证每个下载任务运行在一个独立的线程中。调度器本身主要就是一个循环，它遍历所有的线程，逐个唤醒它们。调度器还必须在线程完成任务后，将该线程从列表中删除。在所有线程都完成运行后，调度器停止循环。

最后，主程序创建所有需要的线程并调起调度器。例如，如果要从 Lua 官网上下载几个发行包，主程序可能如下：

```
get("www.lua.org", "/ftp/lua-5.3.2.tar.gz")  
get("www.lua.org", "/ftp/lua-5.3.1.tar.gz")  
get("www.lua.org", "/ftp/lua-5.3.0.tar.gz")  
get("www.lua.org", "/ftp/lua-5.2.4.tar.gz")  
get("www.lua.org", "/ftp/lua-5.2.3.tar.gz")  
  
dispatch() -- 主循环
```

在笔者的机器上，串行实现花了 15 秒下载到这些个文件，而协程实现比串行实现快了三倍多。

尽管速度提高了，但最后一种实现还有很大的优化空间。当至少有一个线程有数据可读取时不会有大问题；然而，如果所有的线程都没有数据可读，调度程序就会陷入忙等待（busy wait），不断地从一个线程切换到另一个线程来检查是否有数据可读。这样，会导致协程版的实现比串行版实现耗费多达 3 倍的 CPU 时间。

为了避免这样的情况，可以使用 LuaSocket 中的函数 select，该函数允许程序阻塞直到一组套接字的状态发生改变^①。要实现这种改动，只需要修改调度器即可，参见示例 26.3。

示例 26.3 使用 select 的调度器

```
function dispatch ()  
    local i = 1  
    local timedout = {}  
    while true do  
        if tasks[i] == nil then -- 没有其他的任务了?  
            break
```

^①译者注：此即非阻塞 I/O 的一种。



```
if tasks[1] == nil then    -- 列表为空?  
    break      -- 从循环中跳出  
end  
i = 1                      -- 否则继续循环  
timedout = {}  
end  
local res = tasks[i]()    -- 运行一个任务  
if not res then    -- 任务结束?  
    table.remove(tasks, i)  
else                  -- 超时  
    i = i + 1  
    timedout[#timedout + 1] = res  
    if #timedout == #tasks then    -- 所有任务都阻塞了?  
        socket.select(timedout)    -- 等待  
    end  
end  
end  
end
```

在循环中，新的调度器将所有超时的连接收集到表 `timedout` 中。请记住，函数 `receive` 将这种超时的连接传递给 `yield`，然后由 `resume` 返回。如果所有的连接均超时，那么调度器调用 `select` 等待这些连接的状态就会发生改变。这个最终的实现与上一个使用协程的实现一样快。另外，由于它不会有忙等待，所以与串行实现耗费的 CPU 资源一样多。

26.1 练习

练习 26.1：实现并运行本章中展示的代码。



第4部分 分享 C 语言 API

本章将介绍如何使用 C 语言 API。首先，我们将讨论如何使用 C 语言 API 来操作文件和目录。然后，我们将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来处理文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

在本章中，我们将探讨如何使用 C 语言 API 来操作文件和目录。我们将学习如何使用 C 语言 API 来读取文件的内容，以及如何使用 C 语言 API 来写入文件的内容。我们还将学习如何使用 C 语言 API 来处理字符串。

第4部分 分享 C 语言 API



第4部分 分享 C 语言 API



27

C 语言 API 总览

Lua 是一种嵌入式语言 (*embedded language*)，这就意味着 Lua 并不是一个独立运行的应用，而是一个库，它可以链接到其他应用程序，将 Lua 的功能融入这些应用。

读者可能会有疑问：如果 Lua 不是一个独立的应用，那么在本书中为什么一直独立地使用它呢？答案是 Lua 解释器，即可执行的 `lua`。这个可执行文件是一个小应用，大概有 600 行代码，它是用 Lua 标准库实现的独立解释器（stand-alone interpreter）。这个解释器负责与用户的交互，将用户的文件和字符串传递给 Lua 标准库，由标准库完成主要的工作（例如，真正地运行 Lua 代码）。

因为能被当作库来扩展某个应用程序，所以 Lua 是一种嵌入式语言 (*embeddable language*)。同时，使用了 Lua 语言的程序也可以在 Lua 环境中注册新的函数，比如用 C 语言（或其他语言）实现的函数，从而增加一些无法直接用 Lua 语言编写的功能，因此 Lua 也是一种可扩展的语言 (*extensible language*)。

上述两种对 Lua 语言的定位（嵌入式语言和可扩展语言）分别对应 C 语言和 Lua 语言之间的两种交互形式。在第一种形式中，C 语言拥有控制权，而 Lua 语言被用作库，这种交互形式中的 C 代码被称为应用代码（*application code*）。在第二种形式中，Lua 语言拥有控制权，而 C 语言被用作库，此时的 C 代码被称为库代码（*library code*）。应用代码和库代码都使用相同的 API 与 Lua 语言通信，这些 API 被称为 C API。

C API 是一个函数、常量和类型组成的集合^①，有了它，C 语言代码就能与 Lua 语言交

^① 本书中，术语“函数”实际上是指“函数或者宏”。C API 以宏的方式实现了各种功能。



互。C API 包括读写 Lua 全局变量的函数、调用 Lua 函数的函数、运行 Lua 代码段的函数，以及注册 C 函数（以便于其后可被 Lua 代码调用）的函数等。通过调用 C API，C 代码几乎可以做 Lua 代码能够做的所有事情。

C API 遵循 C 语言的操作模式 (*modus operandi*)，与 Lua 的操作模式有很大区别。在使用 C 语言编程时，我们必须注意类型检查、错误恢复、内存分配错误和其他一些复杂的概念。C API 中的大多数函数都不会检查其参数的正确性，我们必须在调用函数前确保参数的合法性^①一旦出错，程序会直接崩溃而不会收到规范的错误信息。此外，C API 强调的是灵活性和简洁性，某些情况下会以牺牲易用性为代价，即便是常见的需求，也可能需要调用好几个 API。这么做虽然有些烦琐，但我们却可以完全控制所有细节。

正如本章标题所示，本章的目的是概述在 C 语言中使用 Lua 时需要注意的事项。不要试图现在就理解所有的细节，后面我们还会进一步学习。但是记住，在 Lua 语言参考手册 (reference manual) 中总是能够找到关于某个特定函数的更多细节。此外，在 Lua 的发行版中也可以找到若干使用 C API 的实例。Lua 独立解释器 (`lua.c`) 给出了几个应用代码的实例，而 Lua 标准库 (`lmathlib.c`、`lstrlib.c` 等) 则给出了几个库代码的实例。

从现在开始，我们就要变成 C 语言程序员了。

27.1 第一个示例

首先来学习一个简单的应用程序的例子：一个独立解释器。示例 27.1 就是一个简单的 Lua 独立解释器。

示例 27.1 一个简单的独立解释器

```
#include <stdio.h>
#include <string.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main (void) {
    char buff[256];
```

^①在编译 Lua 时，可以使用宏定义 `LUA_USE_APICHECK` 来启用某些检查。这个选项在调试 C 代码的时候特别有用。不过尽管如此，C 语言中的某些错误也是无法检测到的，例如无效的指针。



```
int error;
lua_State *L = luaL_newstate(); /* 打开Lua */
luaL_openlibs(L); /* 打开标准库 */

while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadstring(L, buff) || lua_pcall(L, 0, 0, 0);
    if (error) {
        fprintf(stderr, "%s\n", lua_tostring(L, -1));
        lua_pop(L, 1); /* 从栈中弹出错误信息 */
    }
}
lua_close(L);
return 0;
}
```

头文件 `lua.h` 声明了 Lua 提供的基础函数，其中包括创建新 Lua 环境的函数、调用 Lua 函数的函数、读写环境中的全局变量的函数，以及注册供 Lua 语言调用的新函数的函数，等等。`lua.h` 中声明的所有内容都有一个前缀 `lua_`（例如 `lua_pcall`）。

头文件 `lauxlib.h` 声明了辅助库（*auxiliary library*, *auxlib*）所提供的函数，其中所有的声明均以 `luaL_` 开头（例如，`luaL_loadstring`）。辅助库使用 `lua.h` 提供的基础 API 来提供更高层次的抽象，特别是对标准库用到的相关机制进行抽象。基础 API 追求经济性和正交性（orthogonality），而辅助库则追求对常见任务的实用性。当然，要在程序中创建其他所需的抽象也是非常简单的。请记住，辅助库不能访问 Lua 的内部元素，而只能通过 `lua.h` 中声明的官方基础 API 完成所有工作。辅助库能实现什么，你的程序就能实现什么。

Lua 标准库没有定义任何 C 语言全局变量，它将其所有的状态都保存在动态的结构体 `lua_State` 中，Lua 中的所有函数都接收一个指向该结构的指针作为参数。这种设计使得 Lua 是可重入的，并且可以直接用于编写多线程代码。

顾名思义，函数 `luaL_newstate` 用于创建一个新的 Lua 状态。当它创建一个新状态时，新环境中没有包含预定义的函数，甚至连 `print` 也没有。为了保持 Lua 语言的精炼，所有的标准库都被组织成不同的包，这样我们在不需要使用某些包时可以忽略它们。头文件 `lualib.h` 中声明了用于打开这些库的函数。函数 `luaL_openlibs` 用于打开所有的标准库。

当创建好一个状态并在其中加载标准库以后，就可以处理用户的输入了。程序会首先调



用函数 `luaL_loadstring` 来编译用户输入的每一行内容。如果没有错误，则返回零，并向栈中压入编译后得到的函数（27.2节我们会学习这个神奇的栈）。然后，程序调用函数 `lua_pcall` 从栈中弹出编译后的函数，并以保护模式（protected mode）运行。与函数 `luaL_loadstring` 类似，如果没有错误发生，函数 `lua_pcall` 返回零；当发生错误时，这两个函数都会向栈中压入一条错误信息。随后我们可以通过函数 `lua_tostring` 获取错误信息，并在打印出错误信息后使用函数 `lua_pop` 将其从栈中删除。

在 C 语言中，真实的错误处理可能会相当复杂，并且如何处理错误取决于应用的性质。Lua 核不会直接向任何输出流写入数据，它只会通过返回错误信息来提示错误。每个应用可以用其所需的最恰当的方式来处理这些错误信息。为了简化讨论，假设以下示例使用如下简单的错误处理函数，即打印一条错误信息，关闭 Lua 状态并结束整个应用：

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

void error(lua_State *L, const char *fmt, ...) {
    va_list argp;
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    lua_close(L);
    exit(EXIT_FAILURE);
}
```

后面我们会讨论更多在应用代码中进行错误处理的内容。

由于 Lua 既可以作为 C 代码来编译，也可以作为 C++ 代码来编译，因此 `lua.h` 中并没有包含以下这种在 C 标准库中的常见的写法：

```
#ifdef __cplusplus
extern "C" {
#endif
...
#endif // __cplusplus
```



如果将 Lua 作为 C 代码编译出来后又要在 C++ 中使用，那么可以引入 `lua.hpp` 来替代 `lua.h`，定义如下：

```
extern "C" {
#include "lua.h"
}
```

27.2 栈

Lua 和 C 之间通信的主要组件是无处不在的虚拟栈（*stack*），几乎所有的 API 调用都是在操作这个栈中的值，Lua 与 C 之间所有的数据交换都是通过这个栈完成的。此外，还可以利用栈保存中间结果。

当我们想在 Lua 和 C 之间交换数据时，会面对两个问题：第一个问题是动态类型和静态类型体系之间不匹配；第二个问题是自动内存管理和手动内存管理之间不匹配。

在 Lua 中，如果我们写 `t[k] = v`，`k` 和 `v` 都可以是几种不同类型；由于元表的存在，甚至 `t` 也可以有不同的类型。然而，如果要在 C 语言中提供这种操作，任意给定的 `settable` 函数都必须有一个固定的类型。为了实现这样的操作，我们就需要好几十个不同的函数（为三个不同类型参数的每一种组合都要写一个函数）。

可以通过在 C 语言中声明某种联合体类型来解决这个问题，假设这种类型叫 `lua_Value`，它能够表示 Lua 语言中所有的值。然后，可以把 `settable` 声明为：

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

这种方法有两个缺点。首先，我们很难将如此复杂的类型映射到其他语言中；而在设计 Lua 时，我们又要求 Lua 语言不仅能方便地与 C/C++ 交互，而且还能与 Java、Fortran、C# 等其他语言方便地交互。其次，Lua 语言会做垃圾收集：由于 Lua 语言引擎并不知道 Lua 中的一个表可能会被保存在一个 C 语言变量中，因此它可能会（错误地）认为这个表是垃圾并将其回收。

因此，Lua API 中没有定义任何类似于 `lua_Value` 的类型，而是使用栈在 Lua 和 C 之间交换数据。栈中的每个元素都能保存 Lua 中任意类型的值。当我们想要从 Lua 中获取一个值（例如一个全局变量的值）时，只需调用 Lua，Lua 就会将指定的值压入栈中。当想要将一个值传给 Lua 时，首先要将这个值压入栈，然后调用 Lua 将其从栈中弹出即可。尽管我们仍然需要一个不同的函数将每种 C 语言类型的值压入栈，还需要另一个不同的函数从栈中弹出

每种 C 语言类型的值，但是避免了过多的组合（combinatorial explosion）。另外，由于这个栈是 Lua 状态的一部分，因此垃圾收集器知道 C 语言正在使用哪些值。

几乎 C API 中的所有函数都会用到栈。正如第一个示例，函数 `luaL_loadstring` 将其结果留在栈中（不管是编译好的代码段还是一条错误消息）；函数 `lua_pcall` 从栈中取出要调用的函数，并且也会将错误消息留在栈中。

Lua 严格地按照 LIFO（Last In First Out，后进先出）的规则来操作栈。在调用 Lua 时，只有栈顶部的部分会发生改变；而 C 语言代码则有更大的自由度。更具体地说，C 语言可以检视栈中的任何一个元素，甚至可以在栈的任意位置插入或删除元素。

27.2.1 压入元素

针对每一种能用 C 语言直接表示的 Lua 数据类型，C API 中都有一个对应的压栈函数：常量 `nil` 使用 `lua_pushnil`；布尔值（在 C 语言中是整型）使用 `lua_pushboolean`；双精度浮点数使用 `lua_pushnumber`^①；整型使用 `lua_pushinteger`；任意字符串（一个指向 `char` 的指针，外加一个长度）使用 `lua_pushlstring`；以`\0`终止的字符串使用 `lua_pushstring`。

```
void lua_pushnil      (lua_State *L);
void lua_pushboolean  (lua_State *L, int bool);
void lua_pushnumber   (lua_State *L, lua_Number n);
void lua_pushinteger  (lua_State *L, lua_Integer n);
void lua_pushlstring  (lua_State *L, const char *s, size_t len);
void lua_pushstring   (lua_State *L, const char *s);
```

当然，也有向栈中压入 C 函数和用户数据的函数，我们后面会讨论它们。

类型 `lua_Number` 相当于 Lua 语言中的浮点数类型，默认为 `double`，但可以在编译时配置 Lua，让 `lua_Number` 为 `float` 甚至 `long double`。类型 `lua_Integer` 相当于 Lua 语言中的整型，通常被定义为 `long long`，即有符号 64 位整型。同样，要把 Lua 语言中的 `lua_Integer` 配置为使用 `int` 或 `long` 也很容易。如果使用 `float-int` 组合，也就是 32 位浮点类型和整型，即我们所说的精简 Lua（Small Lua），对于资源受限的机器和硬件而言，相当高效。^②

Lua 语言中的字符串不是以`\0`结尾的，它们可以包含任意的二进制数据。因此，将字符串压栈的基本函数 `lua_pushlstring` 需要一个明确的长度作为参数。对于以`\0`结尾的字符

^①由于历史的原因，C API 中的术语“number”指的是双精度浮点类型。

^②对于这些配置，请参见头文件 `luaconf.h`。

串，也可以使用函数 `lua_pushstring`，该函数通过 `strlen` 来计算字符串的长度。Lua 语言不会保留指向外部字符串（或指向除静态的 C 语言函数外的任何外部对象）的指针。对于不得不保留的字符串，Lua 要么生成一个内部副本，要么复用已有的字符串。因此，一旦上述函数返回，即使立刻释放或修改缓冲区也不会出现问题。

无论何时向栈内压入一个元素，我们都应该确保栈中有足够的空间。请注意，现在你是一个 C 语言程序员，Lua 语言也不会宠着你。当 Lua 启动时，以及 Lua 调用 C 语言时，栈中至少会有 20 个空闲的位置（slot）（头文件 `lua.h` 中将这个常量定义为 `LUA_MINSTACK`）。对于大多数情况，这个空间完全够用，所以我们一般无须考虑栈空间的问题。不过，有些任务可能会需要更多的栈空间，特别是循环向栈中压入元素时。在这些情况下，就需要调用函数 `lua_checkstack` 来检查栈中是否有足够的空间：

```
int lua_checkstack (lua_State *L, int sz);
```

这里，`sz` 是我们所需的额外栈位置的数量。如果可能，函数 `lua_checkstack` 会增加栈的大小，以容纳所需的额外空间；否则，该函数返回零。

辅助库也提供了一个高层函数来检查栈空间：

```
void luaL_checkstack (lua_State *L, int sz, const char *msg);
```

该函数类似于函数 `lua_checkstack`，但是如果栈空间不能满足请求，该函数会使用指定的错误信息抛出异常，而不是返回错误码。

27.2.2 查询元素

C API 使用索引（`index`）来引用栈中的元素。第一个被压入栈的元素索引为 1，第二个被压入的元素索引为 2，依此类推。我们还可以以栈顶为参照，使用负数索引来访问栈中的元素。此时，-1 表示栈顶元素（即最后被压入栈的元素），-2 表示在它之前被压入栈的元素，依此类推。例如，调用 `lua_tostring(L, -1)` 会将栈顶的值作为字符串返回。正如你接下来要看到的，有些情况下从栈底对栈进行索引更加自然（即使用正数索引），而有些情况下则使用负数索引更好。

要检查栈中的一个元素是否为特定的类型，C API 提供了一系列名为 `lua_is*` 的函数，其中 * 可以是任意一种 Lua 数据类型。这些函数包括 `lua_isnil`、`lua_isnumber`、`lua_isstring` 和 `lua_istable` 等。所有这些函数都有同样的原型：

```
int lua_is* (lua_State *L, int index);
```

实际上，函数 `lua_isnumber` 不会检查某个值是否为特定类型，而是检查该值是否能被转换为此特定类型。函数 `lua_isstring` 与之类似，特别之处在于，它接受数字。

还有一个函数 `lua_type`，用于返回栈中元素的类型，每一种类型都由一个对应的常量表示，包括 `LUA_TNIL`、`LUA_TBOOLEAN`、`LUA_TNUMBER`、`LUA_TSTRING` 等。该函数一般与 `switch` 语句连用。当需要检查字符串和数值是否存在潜在的强制类型转换时，该函数也同样有用。

函数 `lua_to*` 用于从栈中获取一个值：

```
int          lua_toboolean (lua_State *L, int index);
const char  *lua_tolstring (lua_State *L, int index,
                           size_t *len);
lua_State   *lua_tothread (lua_State *L, int index);
lua_Number   lua_tonumber (lua_State *L, int index);
lua_Integer  lua_tointeger (lua_State *L, int index);
```

即使指定的元素的类型不正确，调用这些函数也不会有问题。函数 `lua_toboolean` 适用于所有类型，它可以按照如下的规则将任意 Lua 值转换为 C 的布尔值：`nil` 和 `false` 转换为 0，所有其他的 Lua 值转换为 1。对于类型不正确的值，函数 `lua_tolstring` 和 `lua_tothread` 返回 `NULL`。不过，数值相关的函数都无法提示数值的类型错误，因此只能简单地返回 0。以前我们需要调用函数 `lua_isnumber` 来检查类型，但 Lua 5.2 引入了如下的新函数：

```
lua_Number  lua_tonumberx (lua_State *L, int idx, int *isnum);
lua_Integer  lua_tointegerx (lua_State *L, int idx, int *isnum);
```

出口参数 `isnum` 返回了一个布尔值，来表示 Lua 值是否被强制转换为期望的类型。

函数 `lua_tolstring` 返回一个指向该字符串内部副本的指针，并将字符串的长度存入到参数 `len` 指定的位置。我们无法修改这个内部副本（`const` 表明了这一点）。Lua 语言保证，只要对应的字符串还在栈中，那么这个指针就是有效的。当 Lua 调用的一个 C 函数返回时，Lua 就会清空栈。因此，作为规则，永远不要把指向 Lua 字符串的指针存放到获取该指针的函数之外。

函数 `lua_tolstring` 返回的所有字符串在其末尾都会有一个额外的`\0`，不过这些字符串中也可能会有`\0`，因此可以通过第三个参数 `len` 获取字符串的真实长度。特别的，假设栈顶的值是一个字符串，那么如下推断永远成立：

```
size_t len;
const char *s = lua_tolstring(L, -1, &len); /* 任意Lua字符串 */
assert(s[len] == '\0');
```

```
assert(strlen(s) <= len);
```

如果不需 要长度信息，可以在调用函数 `lua_tolstring` 时将第三个参数设为 `NULL`。不过，使用宏 `lua_tostring` 会更好，因为这个宏就是用 `NULL` 作为第三个参数来调用函数 `lua_tolstring` 的。

为了演示这些函数的用法，示例 27.2 提供了一个有用的辅助函数，它输出整个栈的内容。

示例 27.2 对栈进行 Dump

```
static void stackDump (lua_State *L) {
    int i;
    int top = lua_gettop(L); /* 栈的深度 */
    for (i = 1; i <= top; i++) { /* 循环 */
        int t = lua_type(L, i);
        switch (t) {
            case LUA_TSTRING: { /* 字符串类型 */
                printf("%s", lua_tostring(L, i));
                break;
            }
            case LUA_TBOOLEAN: { /* 布尔类型 */
                printf(lua_toboolean(L, i) ? "true" : "false");
                break;
            }
            case LUA_TNUMBER: { /* 数值类型 */
                printf("%g", lua_tonumber(L, i));
                break;
            }
            default: { /* 其他类型 */
                printf("%s", lua_typename(L, t));
                break;
            }
        }
        printf(" "); /* 输出分隔符 */
    }
    printf("\n"); /* 换行符 */
}
```

这个函数从栈底向栈顶遍历，并根据每个元素的类型打印其值。它打印字符串时会用单引号将其括起来，对数值类型的值则使用格式 "%g" 输出，对于其他 C 语言中不存在等价类型的值（表、函数等）则只打印出它们的类型（函数 `lua_typename` 可以将类型编码转换为类型名称）。

在 Lua 5.3 中，由于整型总是可以被强制转换为浮点型，因此仍然可以用函数 `lua_tonumber` 和 "%g" 的格式打印所有的数值。但是，我们倾向于将整数打印为整型，以避免损失精度。此时，我们可以用新函数 `lua_isinteger` 来区分整型和浮点型：

```
case LUA_TNUMBER: { /* 数值 */
    if (lua_isinteger(L, i)) /* 整型? */
        printf("%lld", lua_tointeger(L, i));
    else /* 浮点型 */
        printf("%g", lua_tonumber(L, i));
    break;
}
```

27.2.3 其他栈操作

除了上述在 C 语言和栈之间交换数据的函数外，C API 还提供了下列用于通用栈操作的函数：

```
int  lua_gettop  (lua_State *L);
void lua_settop (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_rotate   (lua_State *L, int index, int n);
void lua_remove   (lua_State *L, int index);
void lua_insert   (lua_State *L, int index);
void lua_replace  (lua_State *L, int index);
void lua_copy     (lua_State *L, int fromidx, int toidx);
```

函数 `lua_gettop` 返回栈中元素的个数，也即栈顶元素的索引。函数 `lua_settop` 将栈顶设置为一个指定的值，即修改栈中的元素数量。如果之前的栈顶比新设置的更高，那么高出来的这些元素就会被丢弃；反之，该函数会向栈中压入 `nil` 来补足大小。特别的，函数 `lua_settop(L, 0)` 用于清空栈。在调用函数 `lua_settop` 时也可以使用负数索引；基于这个功能，C API 提供了下面的宏，用于从栈中弹出 `n` 个元素：

```
#define lua_pop(L,n)  lua_settop(L, -(n) - 1)
```

函数 `lua_pushvalue` 用于将指定索引上的元素的副本压入栈。

函数 `lua_rotate` 是 Lua 5.3 中新引入的。顾名思义，该函数将指定索引的元素向栈顶转动 n 个位置。若 n 为正数，表示将元素向栈顶方向转动，而 n 为负数则表示向相反的方向转动。这是一个非常有用的函数，另外两个 C API 操作实际上是基于使用该函数的宏定义的。其中一个是 `lua_remove`，用于删除指定索引的元素，并将该位置之上的所有元素下移以填补空缺，其定义如下：

```
#define lua_remove(L,idx) \
(lua_rotate(L, (idx), -1), lua_pop(L, 1))
```

也就是说，该函数会将栈转动一格，把想要的那个元素移动到栈顶，然后弹出该元素。另一个宏是 `lua_insert`，用于将栈顶元素移动到指定位置，并上移指定位置之上的所有元素以开辟出一个元素的空间：

```
#define lua_insert(L,idx)      lua_rotate(L, (idx), 1)
```

函数 `lua_replace` 弹出一个值，并将栈顶设置为指定索引上的值，而不移动任何元素。最后，函数 `lua_copy` 将一个索引上的值复制到另一个索引上，并且原值不受影响^①。请注意，以下的操作不会对空栈产生影响：

```
lua_settop(L, -1); /* 将栈顶设为当前的值 */
lua_insert(L, -1); /* 将栈顶的元素移动到栈顶 */
lua_copy(L, x, x); /* 把一个元素复制到它当前的位置 */
lua_rotate(L, x, 0); /* 旋转零个位置 */
```

示例 27.3 中的程序使用 `stackDump`（在示例 27.2 中定义）演示了这些栈操作。

示例 27.3 栈操作示例

```
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"

static void stackDump (lua_State *L) {
```

^① 函数 `lua_copy` 是在 Lua 5.2 中引入的。

参见示例 27.2

}

```

int main (void) {
    lua_State *L = luaL_newstate();
    lua_pushboolean(L, 1);
    lua_pushnumber(L, 10);
    lua_pushnil(L);
    lua_pushstring(L, "hello");
    stackDump(L);
    /* 将输出:  true  10  nil  'hello' */

    lua_pushvalue(L, -4); stackDump(L);
    /* 将输出:  true  10  nil  'hello'  true */

    lua_replace(L, 3); stackDump(L);
    /* 将输出:  true  10  true  'hello' */

    lua_settop(L, 6); stackDump(L);
    /* 将输出:  true  10  true  'hello'  nil  nil */

    lua_rotate(L, 3, 1); stackDump(L);
    /* 将输出:  true  10  nil  true  'hello'  nil */

    lua_remove(L, -3); stackDump(L);
    /* 将输出:  true  10  nil  'hello'  nil */

    lua_settop(L, -5); stackDump(L);
    /* 将输出:  true */

    lua_close(L);
    return 0;
}

```

27.3 使用 C API 进行错误处理

Lua 中所有的结构都是动态的：它们会按需扩展，并且在可能时最后重新收缩（shrink）。这意味着在 Lua 中内存分配失败可能无处不在，几乎所有的操作最终都可能会面临内存分配失败。此外，许多操作可能会抛出异常^①。例如，访问一个全局变量可能会触发 `_index` 元方法，而该元方法又可能会抛出异常。最后，分配内存的操作会触发垃圾收集器，而垃圾收集器又可能会调用同样可能抛出异常的析构器。简而言之，Lua API 中的绝大部分函数都可能抛出异常。

Lua 语言使用异常来提示错误，而没有在 API 的每个操作中使用错误码。与 C++ 或 Java 不同，C 语言没有提供异常处理机制。为了解决这个问题，Lua 使用了 C 语言中的 `setjmp` 机制，`setjmp` 营造了一个类似异常处理的机制。因此，大多数 API 函数都可以抛出异常（即调用函数 `longjmp`）而不是直接返回。

在编写库代码时（被 Lua 语言调用的 C 函数），由于 Lua 会捕获所有异常，因此，对我们来说使用 `longjmp` 并不用进行额外的操作。不过，在编写应用程序代码（调用 Lua 的 C 代码）时，则必须提供一种捕获异常的方式。

27.3.1 处理应用代码中的错误

如果应用调用了 Lua API 中的函数，就可能发生错误。正如我们前面的讨论，Lua 语言通常通过长跳转来提示错误。但是，如果没有相应的 `setjmp`，解释器就无法进行长跳转。此时，API 中的任何错误都会导致 Lua 调用紧急函数（panic function），当这个函数返回后，应用就会退出。我们可以通过函数 `lua_atpanic` 来设置自己的紧急函数，但作用不大。

要正确地处理应用代码中的错误，就必须通过 Lua 语言调用我们自己的代码，这样 Lua 语言才能设置合适的上下文来捕获异常，即在 `setjmp` 的上下文中运行代码。类似于通过函数 `pcall` 在保护模式中运行 Lua 代码，我们也可以用函数 `lua_pcall` 运行 C 代码。更具体地说，可以把 C 代码封装到一个函数 F 中，然后使用 `lua_pcall` 调用这个函数 F。通过这种方式，我们的 C 代码会在保护模式下运行。即便发生内存分配失败，函数 `lua_pcall` 也会返回一个对应的错误码，使解释器能够保持一致的状态（consistent state），如下所示：

^①译者注：在编程语言中，异常方面通常有“raise error（引发错误）”和“throw exception（抛出异常）”两种说法，经常混用。本文原文的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，故在本章之前的所有译文采用的均是“抛出异常”。由于本章讲的就是 Lua 语言的错误处理机制，因此本章中使用“引发错误”的译法。

```

static int foo (lua_State *L) {
    code to run in protected mode (要以保护模式运行的代码)
    return 0;
}

int secure_foo (lua_State *L) {
    lua_pushcfunction(L, foo); /* 将'foo'作为Lua函数压栈 */
    return (lua_pcall(L, 0, 0, 0) == 0);
}

```

在上述示例中，无论发生什么，调用 `secure_foo` 时都会返回一个布尔值，来表示 `foo` 执行是否成功。特别的，请注意，栈中已经预先分配了空间，而且函数 `lua_pushcfunction` 不会分配内存，这样才不会引发错误。（函数 `foo` 的原型是函数 `lua_pushcfunction` 所要求的，后者用于在 Lua 中创建一个代表 C 函数的 Lua 函数。我们会在 29.1 节讨论 C 函数有关的细节。）

27.3.2 处理库代码中的错误

Lua 是一种安全 (*safe*) 的语言。这意味着不管用 Lua 写的是什么，也不管写出来的内容多么不正确，我们总是能用它自身的机制来理解程序的行为。此外，程序中的错误 (*error*) 也是通过 Lua 语言的机制来检测和解释的。与之相比，许多 C 语言代码中的错误只能从底层硬件的角度来解释（例如，把异常位置作为指令地址给出）。

只要往 Lua 中加入新的 C 函数，这种安全性就可能被打破。例如，一个等价于 BASIC 命令 `poke` 的函数（该函数用于将任意的字节存储到任意的内存地址中）就可能导致各种各样的内存崩溃。因此，我们必须确保新加入的内容对 Lua 语言来说是安全的，并提供妥善的错误处理。

正如之前所讨论的，C 语言程序必须通过 `lua_pcall` 设置错误处理。不过，在为 Lua 编写库函数时，通常无须处理错误。库函数抛出的错误要么被 Lua 中的 `pcall` 捕获，要么被应用代码中的 `lua_pcall` 捕获。因此，当 C 语言库中的函数检测到错误时，只需简单地调用 `lua_error` 即可（或调用 `luaL_error` 更好，它会格式化错误信息，然后调用 `lua_error`）。函数 `lua_error` 会收拾 Lua 系统中的残局，然后跳转回保护模式调用处，并传递错误信息。

27.4 内存分配

Lua 语言核心对内存分配不进行任何假设, 它既不会调用 `malloc` 也不会调用 `realloc` 来分配内存。相反, Lua 语言核心只会通过一个分配函数 (*allocation function*) 来分配和释放内存, 当用户创建 Lua 状态时必须提供该函数。

`luaL_newstate` 是一个用默认分配函数来创建 Lua 状态的辅助函数。该默认分配函数使用了来自 C 语言标准函数库的标准函数 `malloc-realloc-free`, 对于大多数应用程序来说, 这几个函数 (或应该是) 够用了。但是, 要完全控制 Lua 的内存分配也很容易, 使用原始的 `lua_newstate` 来创建我们自己的 Lua 状态即可:

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

该函数有两个参数: 一个是分配函数, 另一个是用户数据。用这种方式创建的 Lua 状态会通过调用 `f` 完成所有的内存分配和释放, 甚至结构 `lua_State` 也是由 `f` 分配的。

分配函数必须满足 `lua_Alloc` 的类型声明:

```
typedef void * (*lua_Alloc) (void *ud,
                           void *ptr,
                           size_t osiz,
                           size_t nsiz);
```

第一个参数始终为 `lua_newstate` 所提供的用户数据; 第二个参数是正要被 (重) 分配或者释放的块的地址; 第三个参数是原始块的大小; 最后一个参数是请求的块大小。如果 `ptr` 不是 `NULL`, Lua 会保证其之前被分配的大小就是 `osize` (如果 `ptr` 是 `NULL`, 那么这个块之前的大小肯定是零, 所以 Lua 使用 `osize` 来存放某些调试信息)。

Lua 语言使用 `NULL` 表示大小为零的块。当 `nsiz` 为零时, 分配函数必须释放 `ptr` 指向的块并返回 `NULL`, 对应于所要求的大小 (为零) 的块。当 `ptr` 是 `NULL` 时, 该函数必须分配并返回一个指定大小的块; 如果无法分配指定的块, 则必须返回 `NULL`。如果 `ptr` 是 `NULL` 并且 `nsiz` 为零, 则两条规则都适用: 最终结果是分配函数什么都不做, 返回 `NULL`。

最后, 当 `ptr` 不是 `NULL` 并且 `nsiz` 不为零时, 分配函数应该像 `realloc` 一样重新分配块并返回新地址 (可能与原地址一致, 也可能不一致)。同样, 当出现错误时分配函数必须返回 `NULL`。Lua 假定分配函数在块的新尺寸小于或等于旧尺寸时不会失败 (Lua 在垃圾收集期间会压缩某些结构的大小, 并且无法从垃圾收集时的错误中恢复)。

`luaL_newstate` 使用的标准分配函数定义如下 (从文件 `lauxlib.c` 中直接抽取):

```

void *l_alloc (void *ud, void *ptr, size_t osize, size_t nsize) {
    (void)ud; (void)osize; /* 未使用 */
    if (nsize == 0) {
        free(ptr);
        return NULL;
    }
    else
        return realloc(ptr, nsize);
}

```

该函数假设 `free(NULL)` 什么也不做，并且 `realloc(NULL, size)` 等价于 `malloc(size)`。ISO C 标准会托管^①这两种行为。

我们可以通过调用 `lua_getallocf` 恢复（recover）Lua 状态的内存分配器：

```
lua_Alloc lua_getallocf (lua_State *L, void **ud);
```

如果 `ud` 不是 `NULL`，那么该函数会把 `*ud` 设置为该分配器的用户数据。我们可以通过调用 `lua_setallocf` 来更改 Lua 状态的内存分配器：

```
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
```

请记住，所有新的分配函数都有责任释放由前一个分配函数分配的块。通常情况下，新的分配函数是在旧分配函数的基础上做了包装，来追踪分配（trace allocation）或同步访问堆（heap）的。

Lua 在内部不会为了重用而缓存空闲内存。它假定分配函数会完成这种缓存工作；而优秀的分配函数确实也会这么做。Lua 不会试图压缩内存碎片。研究表明，内存碎片更多是由糟糕的分配策略导致的，而非程序的行为造成的；而优秀的分配函数不会造成太多内存碎片。

对于已有的优秀分配函数，想要做到比它更好是很难的，但有时候也不妨一试。例如，Lua 会告诉你已经释放或者重新分配的块的原有大小。因此，一个特定的分配函数不需要保存有关块大小的信息，以此减少每个块的内存开销。

还有一种可以改善的内存分配的场景，是在多线程系统中。这种系统通常需要对内存分配函数进行线程同步，因为这些函数使用的是全局资源（堆）。不过，对 Lua 状态的访问也必须是同步的——或者更好的情况是，限制只有一个线程能够访问 Lua 状态，正如在第33章

^①译者注：此处的托管是指 `malloc` 的实现是基于特定平台的，ISO C 标准只规定 `malloc` 函数应该“做什么”，而不对“如何做”进行任何假设和限定。

中实现的 lproc 一样。因此，如果每个 Lua 状态都从私有的内存池中分配内存，那么分配函数就可以避免线程同步导致的额外开销。

27.5 练习

练习 27.1：编译并运行简单的独立运行的解释器（示例 27.1）。

练习 27.2：假设栈是空的，执行下列代码后，栈中会是什么内容？

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_rotate(L, 1, -1);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```

练习 27.3：使用函数 stackDump（见示例 27.2）检查上一道题的答案。

练习 27.4：请编写一个库，该库允许一个脚本限制其 Lua 状态能够使用的总内存大小。该库可能仅提供一个函数 setlimit，用来设置限制值。

这个库应该设置它自己的内存分配函数，此函数在调用原始的分配函数之前，应该检查在使用的内存总量，并且在请求的内存超出限制时返回 NULL。

（提示：这个库可以使用分配函数的用户数据来保存状态，例如字节数、当前内存限制等；请记住，在调用原始分配函数时应该使用原始的用户数据。）

28

扩展应用

Lua 的重要用途之一就是用作配置 (*configuration*) 语言。本章将介绍如何使用 Lua 语言来配置一个程序，从一个简单的示例开始，然后对其逐步扩展来完成更复杂的任务。

28.1 基础知识

让我们想象一个简单的需要配置的场景：假设我们的 C 程序有一个窗口，并希望用户能够指定窗口的初始大小。显然，对于这种简单的任务，有许多比使用 Lua 语言更简单的方法，例如使用环境变量或使用基于键值对的配置文件。但即便是使用一个简单的文本文件，我们也需要对其进行解析。因此，我们决定使用一个 Lua 配置文件（也即一个普通的文本文件，只不过它是一个 Lua 程序）。下面所示的是这种文件最简单的形式，它可以包含如下内容：

```
-- 定义窗口大小
width = 200
height = 300
```

现在，我们必须使用 Lua API 来指挥 Lua 语言解析该文件，并获取全局变量 `width` 和 `height` 的值。示例 28.1 中的函数 `load` 完成了此项工作。

示例 28.1 从配置文件中获取用户信息

```
int getglobint (lua_State *L, const char *var) {
```

```

int isnum, result;
lua_getglobal(L, var);
result = (int)lua_tointegerx(L, -1, &isnum);
if (!isnum)
    error(L, "'%s' should be a number\n", var);
lua_pop(L, 1); /* 从栈中移除结果 */
return result;
}

void load (lua_State *L, const char *fname, int *w, int *h) {
if ( luaL_loadfile(L, fname) || luaL_pcall(L, 0, 0, 0))
    error(L, "cannot run config. file: %s", lua_tostring(L, -1));
*w = getglobint(L, "width");
*h = getglobint(L, "height");
}

```

假设我们已经按照第27章学习的内容创建了一个 Lua 状态。它调用函数 `luaL_loadfile` 从文件 `fname` 中加载代码段，然后调用函数 `lua_pcall` 运行编译后的代码段。如果发生错误（例如配置文件中有语法错误），那么这两个函数会把错误信息压入栈，并返回一个非零的错误码。此时，程序可以用索引-1 来调用函数 `lua_tostring` 从栈顶获取错误信息（我们在27.1节已定义了函数 `error`）。

当运行完代码段后，C 程序还需要获取全局变量的值。因此，该程序调用了两次辅助函数 `getglobint`（也在示例 28.1 中）。`getglobint` 首先调用函数 `lua_getglobal` 将相应全局变量的值压入栈，`lua_getglobal` 只有一个参数（除了无所不在的 `lua_State`），就是变量名。然后，`getglobint` 调用函数 `lua_tointegerx` 将这个值转换为整型以保证其类型正确。

那么，用 Lua 语言来完成这类任务是否值得呢？正如笔者之前所说，对于这类简单的任务，用一个仅仅包含两个数字的简单文件会比用 Lua 语言更方便。尽管如此，使用 Lua 还是会有一些好处。首先，Lua 为我们处理了所有的语法细节，甚至配置文件都可以有注释！其次，用户还可以使用 Lua 来实现一些更复杂的配置。例如，脚本可以提示用户输入某些信息，或者查询环境变量来选择合适的窗口大小：

```

-- 配置文件

if getenv("DISPLAY") == ":0.0" then
    width = 300; height = 300
else
    width = 1024; height = 768
end

```

```
width = 200; height = 200
end
```

即使是在这样一个简单的配置场景中，要满足用户的需求也非易事；不过，只要脚本定义了这两个变量，我们的 C 程序无须修改就能运行。

最后一个使用 Lua 的理由是，使用它以后，向程序中添加新的配置机制会很方便。这种便利性可以让人形成一种态度，这种态度让程序变得更加灵活。

28.2 操作表

让我们一起来践行这种态度。现在，我们要为每个窗口配置一种背景色。假设最终的颜色格式是由三个数字分量组成的 RGB 颜色。通常，在 C 语言中，这些数字是在区间 $[0,255]$ 中的整型数；而在 Lua 语言中，我们会使用更自然的区间 $[0,1]$ ^①。

一种直接的方法是要求用户用不同的全局变量设置每个分量：

```
-- 配置文件
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

这种方法有两个缺点：第一，太烦琐（在真实的程序中可能需要数十种不同的颜色，用于设置窗口背景、窗口前景、菜单背景等）；第二，无法预定义常用颜色，如果能预定义常用颜色，用户只需要写 `background =WHITE` 之类的语句就好。为了避免这些缺点，我们将用一张表来表示颜色：

```
background = {red = 0.30, green = 0.10, blue = 0}
```

使用表可以让脚本变得更加结构化。现在，用户（或者应用程序）就可以很容易地在配置文件中预定义后面要用的颜色了：

```
BLUE = {red = 0, green = 0, blue = 1.0}
```

^①译者注：在国内的参考书中 RGB 分量通常还是 0~255 范围内，包括 Windows 操作系统中采用的也是 0~255 的范围，只是作者自己觉得 0~1 的范围更“自然”罢了。

other color definitions (其他颜色定义)

```
background = BLUE
```

若要在 C 语言中获取这些值，可以使用如下的代码：

```
lua_getglobal(L, "background");
if (!lua_istable(L, -1))
    error(L, "'background' is not a table");

red = getcolorfield(L, "red");
green = getcolorfield(L, "green");
blue = getcolorfield(L, "blue");
/* 译者注：本章中颜色示例的最终目的就是在C语言中获得这三个变量red、
green和blue的值，牢记 */
```

上述代码先获取全局变量 `background` 的值，并确认它是一张表；然后使用 `getcolorfield` 获取每个颜色的分量。

当然，函数 `getcolorfield` 不是 Lua API 的一部分，必须先定义它。此外，我们还面临多态的问题：`getcolorfield` 函数可能有许多版本，它们有不同类型的键、不同类型的值和错误处理等。Lua API 只提供了一个函数 `lua_gettable` 来处理所有的类型，该函数以这个表在栈中的位置为参数，从栈中弹出键再压入相应的值。示例 28.2 中定义了私有的 `getcolorfield`，这个函数假设表位于栈顶。

示例 28.2 `getcolorfield` 的详细实现

```
#define MAX_COLOR      255

/* 假设表位于栈顶 */
int getcolorfield (lua_State *L, const char *key) {
    int result, isnum;
    lua_pushstring(L, key); /* 压入键 */
    lua_gettable(L, -2); /* 获得background[key] */
    result = (int)(lua_tonumberx(L, -1, &isnum) * MAX_COLOR);
    if (!isnum)
        error(L, "invalid component '%s' in color", key);
    lua_pop(L, 1); /* 移除数值 */
```

```
        return result;  
    }  
}
```

使用 `lua_pushstring` 压入键以后，表就位于索引-2 上。在 `getcolorfield` 返回前，它会从栈中弹出检索到的值以达到栈平衡。

我们继续拓展这个示例，为用户引入颜色的名字。用户除了可以使用颜色表，还可以使用更多常用颜色的预定义名字。要实现这个功能，在 C 程序中就要有一张颜色表：

```
struct ColorTable {
    char *name;
    unsigned char red, green, blue;
} colortable[] = {{"WHITE", MAX_COLOR, MAX_COLOR, MAX_COLOR},
                 {"RED", MAX_COLOR, 0, 0},
                 {"GREEN", 0, MAX_COLOR, 0},
                 {"BLUE", 0, 0, MAX_COLOR},
                 other colors (其他颜色)
                 {NULL, 0, 0, 0} /* 哨兵 */
};
```

我们的实现会使用这些颜色名来创建全局变量，然后用颜色表来初始化这些全局变量。最终的结果相当于用户在其脚本中写了如下的内容：

WHITE = {red = 1.0, green = 1.0, blue = 1.0}
RED = {red = 1.0, green = 0, blue = 0}
other colors (其他颜色)

为了设置表的字段，我们定义了一个辅助函数 `setcolorfield`，该函数会将索引和字段名压入栈，然后调用函数 `lua_settable`：

```
/* 假设表位于栈顶 */
void setcolorfield (lua_State *L, const char *index, int value) {
    lua_pushstring(L, index); /* 键 */
    lua_pushnumber(L, (double)value / MAX_COLOR); /* 值 */
    lua_settable(L, -3);
}
```

与其他 API 函数一样，函数 `lua_settable` 需要处理很多不同的数据类型，因此它会从栈中获取所有的操作数，将表索引当作参数并弹出键和值。函数 `setcolorfield` 假设在调用前表位于栈顶（索引为-1）；压入了键和值以后，表位于索引为-3 的位置上。

下一个函数是 `setcolor`，用于定义单个颜色，它会创建一张表，设置相应的字段，并将这个表赋给相应的全局变量：

```
void setcolor (lua_State *L, struct ColorTable *ct) {
    lua_newtable(L);           /* 创建表（译者注：这其实是一个宏，详情见后文） */
    setcolorfield(L, "red", ct->red);
    setcolorfield(L, "green", ct->green);
    setcolorfield(L, "blue", ct->blue);
    lua_setglobal(L, ct->name); /* 'name' = table */
}
```

函数 `lua_newtable` 创建一个空表，并将其压入栈；其后三次调用 `setcolorfield` 设置表的各个字段；最后，函数 `lua_setglobal` 弹出表，并将其设置为具有指定名称全局变量的值。

有了上述的函数，下面的这个循环就会为配置脚本注册所有的颜色：

```
int i = 0;
while (colortable[i].name != NULL)
    setcolor(L, &colortable[i++]);
```

请注意，在运行脚本前应用程序必须先执行这个循环^①。

示例 28.3 演示了另一种实现颜色命名的方法。

示例 28.3 用字符串或表表示颜色

```
lua_getglobal(L, "background");
/* 译者注：获取全局变量background值，结果位于栈顶 */
if (lua_isstring(L, -1)) { /* 值是一个字符串？ */
    const char *name = lua_tostring(L, -1); /* 获得字符串 */
    int i; /* 搜索颜色表 */
    for (i = 0; colortable[i].name != NULL; i++) {
        if (strcmp(colorname, colortable[i].name) == 0)
            break;
    }
}
```

^①译者注：即将 C 语言中定义的颜色注册到 Lua 中。

```

    }

    if (colortable[i].name == NULL) /* 没有发现字符串? */
        error(L, "invalid color name (%s)", colortable[i].name);
    else { /* 使用colortable[i] */
        red = colortable[i].red;
        green = colortable[i].green;
        blue = colortable[i].blue;
    }
} else if (lua_isstable(L, -1)) {
    red = getcolorfield(L, "red");
    green = getcolorfield(L, "green");
    blue = getcolorfield(L, "blue");
} else
    error(L, "invalid value for 'background'");
}

```

除了全局变量，用户还可以使用字符串来表示颜色名，例如通过 `background = "BLUE"` 来进行设置。因此，`background` 既可以是表又可以是字符串。在这种设计下，在运行用户脚本前应用无须做任何事情；不过，应用在获取颜色时需要做更多的工作。当应用获取变量 `background` 的值时，必须测试该值是否为字符串，然后在颜色表中查找这个字符串。

哪一个是最好的方法呢？在 C 语言程序中，用字符串来表示选项并不是一个好做法，因为编译器无法检测到拼写错误。不过，在 Lua 语言中，对于拼错了的颜色，该配置“程序”的作者可能会发现其错误信息。程序员和用户之间的区别没有那么明确，因此编译错误和运行时错误之间的区别也不明确。

使用字符串时，`background` 的值可能会有拼写错误；因此，应用程序可以把这个错误的拼写添加到错误信息中。应用程序还可以在比较字符串时忽略大小写，这样用户就可以使用"`white`"、"`WHITE`" 甚至"`White`"。此外，如果用户的脚本很小且颜色很多，那么用户只需要几种颜色却注册上百种颜色（创建上百张表和全局变量）的做法会很低效。使用字符串则可以避免这种开销。

28.2.1 一些简便方法

尽管 Lua 语言的 C API 追求简洁性，但 Lua 也没有做得过于激进。因此，C API 为一些常用的操作提供了一些简便方法。接下来就让我们一起来看几种简便方法。

由于通过字符串类型的键来检索表是很常见的操作，因此 Lua 语言针对这种情况提供了一个特定版本的 `lua_gettable` 函数：`lua_getfield`。使用这个函数，可以将 `getcolorfield` 中的如下两行代码：

```
lua_pushstring(L, key);
lua_gettable(L, -2); /* 获取background[key] */
```

重写为：

```
lua_getfield(L, -1, key); /* 获取background[key] */
```

因为没有把这个字符串^①压栈，所以调用 `lua_getfield` 时，表的索引仍然是-1。

由于经常要检查 `lua_gettable` 返回的值的类型，因此，在 Lua 5.3 中，该函数（以及与 `lua_getfield` 类似的函数）会返回结果的类型。所以，我们可以简化 `getcolorfield` 中后续的访问和检查：

```
if (lua_getfield(L, -1, key) != LUA_TNUMBER)
    error(L, "invalid component in background color");
```

正如你可能期望的那样，Lua 语言还为字符串类型的键提供了一个名为 `lua_setfield` 的特殊版本的 `lua_settable`。使用该函数，可以重写之前对 `setcolorfield` 的定义：

```
void setcolorfield (lua_State *L, const char *index, int value) {
    lua_pushnumber(L, (double)value / MAX_COLOR);
    lua_setfield(L, -2, index);
}
```

作为一个小优化，我们还可以在函数 `setcolor` 中替代对函数 `lua_newtable` 的使用。Lua 提供了另一个函数 `lua_createtable`，它可以创建表并为元素预分配空间。Lua 将这些函数声明为：

```
void lua_createtable (lua_State *L, int narr, int nrec);

#define lua_newtable(L)      lua_createtable(L, 0, 0)
```

参数 `narr` 是表中连续元素（即具有连续整数索引的元素）的期望个数，而 `nrec` 是其他元素的期望数量。在 `setcolor` 中，我们会用 `lua_createtable(L, 0, 3)` 提示该表中会有三个元素（在编写表构造器时，Lua 代码也会做类似的优化）。

^①译者注：即键名，变量 `key`。

28.3 调用 Lua 函数

Lua 语言的一大优势在于允许在一个配置文件中定义应用所调用的函数。例如，我们可以用 C 语言编写一个应用来绘制某个函数的图形，并用 Lua 定义要绘制的函数。

调用 Lua 函数的 API 规范很简单：首先，将待调用的函数压栈；然后，压入函数的参数；接着用 `lua_pcall` 进行实际的调用；最后，从栈中取出结果。

举一个例子，假设配置文件中有如下的函数：

```
function f (x, y)
    return (x^2 * math.sin(y)) / (1 - x)
end
```

我们想在 C 语言中对指定的 x 和 y 计算表达式 $z=f(x,y)$ 的值。假设我们已经打开了 Lua 库并运行了该配置文件，示例 28.4 中的函数 `f` 计算了表达式 $z=f(x,y)$ 的值。

示例 28.4 从 C 语言中调用 Lua 函数

```
/* 调用Lua语言中定义的函数'f' */
double f (lua_State *L, double x, double y) {
    int isnum;
    double z;

    /* 函数和参数压栈 */
    lua_getglobal(L, "f"); /* 要调用的函数 */
    lua_pushnumber(L, x); /* 压入第一个参数 */
    lua_pushnumber(L, y); /* 压入第二个参数 */

    /* 进行调用（两个参数，一个结果） */
    if (lua_pcall(L, 2, 1, 0) != LUA_OK)
        error(L, "error running function 'f': %s",
              lua_tostring(L, -1));
    /* 获取结果 */
    z = lua_tonumberx(L, -1, &isnum);
    if (!isnum)
        error(L, "function 'f' should return a number");
```

```

lua_pop(L, 1); /* 弹出返回值 */
return z;
}

```

在调用函数 `lua_pcall` 时，第二个参数表示传递的参数数量，第三个参数是期望的结果数量，第四个参数代表错误处理函数（稍后讨论）。就像 Lua 语言的赋值一样，函数 `lua_pcall` 会根据所要求的数量来调整返回值的个数，即压入 `nil` 或丢弃多余的结果。在压入结果前，`lua_pcall` 会把函数和其参数从栈中移除。当一个函数返回多个结果时，那么第一个结果最先被压入。例如，如果函数返回三个结果，那么第一个结果的索引是-3，最后一个结果的索引是-1。

如果函数 `lua_pcall` 在运行过程中出现错误，它会返回一个错误码，并在栈中压入一条错误信息（但是仍会弹出函数及其参数）。不过，如果有错误处理函数，在压入错误信息前，`lua_pcall` 会先调用错误处理函数。我们可以通过 `lua_pcall` 的最后一个参数指定这个错误处理函数，零表示没有错误处理函数，即最终的错误信息就是原来的消息；若传入非零参数，那么参数应该是该错误处理函数在栈中的索引。在这种情况下，错误处理函数应该被压入栈且位于待调用函数之下。

对于普通的错误，`lua_pcall` 会返回错误代码 `LUA_ERRRUN`。但有两种特殊的错误会生成不同的错误码，因为它们不会运行错误处理函数。第一种错误是内存分配失败，对于这类错误，`lua_pcall` 会返回 `LUA_ERRMEM`。第二种错误是消息处理函数本身出错，此时再次调用错误处理函数基本上没用，因此 `lua_pcall` 会立即返回错误码 `LUA_ERRERR`。自 Lua 5.2 后，Lua 语言还区分了第三种错误，即当一个析构器引发错误时，`lua_pcall` 会返回错误码 `LUA_ERRGCMM` (*error in a GC metamethod*)，表示错误并非与调用自身直接相关。

28.4 一个通用的调用函数

下例是一个更高级的示例，我们将编写一个调用 Lua 函数的包装程序，其中用到了 C 语言的 `stdarg` 机制。这个包装函数名为 `call_va`，它接受一个待调用的全局函数的名字、一个描述参数类型和结果类型的字符串、参数列表，以及存放结果的一组指向变量的指针。函数 `call_va` 会处理有关 API 的所有细节。用这个函数，可以将示例 28.4 中的例子简化为：

```
call_va(L, "f", "dd>d", x, y, &z);
```

其中，字符串"dd>d"表示“两个双精度浮点型的参数和一个双精度浮点型的结果”。在这种表示方法中，字母 d 表示双精度浮点型，字母 i 表示整型，字母 s 表示字符串，> 用于分隔参数和结果。如果该函数没有结果，那么 > 可以没有。

示例 28.5 演示了 call_va 的具体实现。

示例 28.5 一个通用的调用函数

```
#include <stdarg.h>

void call_va (lua_State *L, const char *func,
              const char *sig, ...) {
    va_list vl;
    int narg, nres; /* 参数和结果的个数 */

    va_start(vl, sig);
    lua_getglobal(L, func); /* 函数压栈 */

    push and count arguments (压入参数并计数, 参见示例 28.6)

    nres = strlen(sig); /* 期望的结果数 */

    if (lua_pcall(L, narg, nres, 0) != 0) /* 进行调用 */
        error(L, "error calling '%s': %s", func,
              lua_tostring(L, -1));

    retrieve results (获取结果, 参见示例 28.7)

    va_end(vl);
}
```

尽管该函数具有通用性，但它与第一个示例的执行步骤相同：压入函数、压入参数（见示例 28.6）、完成调用，并获取结果（见示例 28.7）。

示例 28.6 为通用调用函数压入参数

```
for (narg = 0; *sig; narg++) { /* 对于每一个参数循环 */
```

```

/* 检查栈空间 */
 luaL_checkstack(L, 1, "too many arguments");

switch (*sig++) {

    case 'd': /* double类型的参数 */
        lua_pushnumber(L, va_arg(vl, double));
        break;

    case 'i': /* int类型的参数 */
        lua_pushinteger(L, va_arg(vl, int));
        break;

    case 's': /* string类型的参数 */
        lua_pushstring(L, va_arg(vl, char *));
        break;

    case '>': /* 参数部分结束 */
        goto endargs; /* 从循环中跳出 */

    default:
        error(L, "invalid option (%c)", *(sig - 1));
}

endargs:

```

示例 28.7 为通用调用函数检索结果

```

nres = -nres; /* 第一个结果的栈索引 */
while (*sig) { /* 对于每一个结果循环 */
    switch (*sig++) {

        case 'd': { /* double类型的结果 */
            int isnum;

```

```

        double n = lua_tonumberx(L, nres, &isnum);
        if (!isnum)
            error(L, "wrong result type");
        *va_arg(vl, double *) = n;
        break;
    }

    case 'i': { /* int类型的结果 */
        int isnum;
        int n = lua_tointegerx(L, nres, &isnum);
        if (!isnum)
            error(L, "wrong result type");
        *va_arg(vl, int *) = n;
        break;
    }

    case 's': { /*类型的結果 */
        const char *s = lua_tostring(L, nres);
        if (s == NULL)
            error(L, "wrong result type");
        *va_arg(vl, const char **) = s;
        break;
    }

    default:
        error(L, "invalid option (%c)", *(sig - 1));
    }
    nres++;
}

```

以上大部分代码都很直观，不过有些细节需要说明一下。首先，通用调用函数无须检查 func 是否是一个函数，因为 `lua_pcall` 会抛出这类异常。其次，由于通用调用函数会压入任意数量的参数，因此必须确保栈中有足够的空间。第三，由于被调用的函数可能会返回字符串，因此 `call_va` 不能将结果弹出栈。调用者必须在使用完字符串结果（或将字符串复制到恰当的缓冲区）后弹出这些字符串。

28.5 练习

练习 28.1：请编写一个 C 程序，该程序读取一个定义了函数 `f` 的 Lua 文件（函数以一个数值参数对一个数值结构的形式给出），并绘制出该函数（无须你做任何特别的事情，程序会像 16.1 节中的例子一样用 ASCII 星号绘出结果）。

练习 28.2：修改函数 `call_va`（见示例 28.5）来处理布尔类型的值。

练习 28.3：假设有一个函数需要监控一些气象站。此函数在内部使用四个字节的字符串来表示每个气象站，并且有一个配置文件将每个字符串映射到相应气象站的实际 URL 上。一个 Lua 配置文件可以以多种方式进行这种映射：

- 一组全局变量，每个变量对应一个气象站。
- 一个表，将字符串映射到 URL 上。
- 一个函数，将字符串映射到 URL 上。

讨论每种方法的优劣，请考虑诸如气象站的总数、URL 的规则（例如，从字符串到 URL 是否存在某种规则）以及用户的类型等因素。

否會將大類神中間相連，若有，有一枚金錢藏在暗處。那吉市指揮升任職大王恩
王人和父親兩祖國的正統，為其一子承父業赴去了 Hejia_eui 殿內，讓那个一派否極轉
運的風雲浦也難耐的開始生出。不就，同這兩尊神像中升職的腰帶插開，這令御醫驚奇
連同身上骨子裡，果真中醫學系認為有破壞作用的，好出錢求吉卦道不。av_Hejia 殿內申
甲首不當露出城郭（即中醫稱之爲

29

在 Lua 中调用 C 语言

我们说 Lua 可以调用 C 语言函数，但这并不意味着 Lua 可以调用所有的 C 函数^①。正如我们在第28章所看到的，当 C 语言调用 Lua 函数时，该函数必须遵循一个简单的规则来传递参数和获取结果。同样，当 Lua 调用 C 函数时，这个 C 函数也必须遵循某种规则来获取参数和返回结果。此外，当 Lua 调用 C 函数时，我们必须注册该函数，即必须以一种恰当的方式为 Lua 提供该 C 函数的地址。

Lua 调用 C 函数时，也使用了一个与 C 语言调用 Lua 函数时相同类型的栈，C 函数从栈中获取参数，并将结果压入栈中。

此处的重点在于，这个栈不是一个全局结构；每个函数都有其私有的局部栈（private local stack）。当 Lua 调用一个 C 函数时，第一个参数总是位于这个局部栈中索引为 1 的位置。即使一个 C 函数调用了 Lua 代码，而且 Lua 代码又再次调用了同一个（或其他）的 C 函数，这些调用每一次都只会看到本次调用自己的私有栈，其中索引为 1 的位置上就是第一个参数。

29.1 C 函数

先举一个例子，让我们实现一个简化版本的正弦函数，该函数返回某个给定数的正弦值：

^①有很多包使 Lua 能够调用任意的 C 语言函数，但是这些包要么不具有 Lua 的可移植性，要么不安全。

```
static int l_sin (lua_State *L) {
    double d = lua_tonumber(L, 1); /* 获取参数 */
    lua_pushnumber(L, sin(d)); /* 压入返回值 */
    return 1; /* 返回值的个数 */
}
```

所有在 Lua 中注册的函数都必须使用一个相同的原型，该原型就是定义在 `lua.h` 中的 `lua_CFunction`：

```
typedef int (*lua_CFunction) (lua_State *L);
```

从 C 语言的角度看，这个函数只有一个指向 Lua 状态类型的指针作为参数，返回值为一个整型数，代表压入栈中的返回值的个数。因此，该函数在压入结果前无须清空栈。在该函数返回后，Lua 会自动保存返回值并清空整个栈。

在 Lua 中，调用这个函数前，还必须通过 `lua_pushcfunction` 注册该函数。函数 `lua_pushcfunction` 会获取一个指向 C 函数的指针，然后在 Lua 中创建一个 "function" 类型，代表待注册的函数。一旦完成注册，C 函数就可以像其他 Lua 函数一样行事了。

一种快速测试函数 `l_sin` 的方法是，将其代码放到简单解释器中（见示例 27.1），并将下列代码添加到 `lual_openlibs` 调用的后面：

```
lua_pushcfunction(L, l_sin);
lua_setglobal(L, "mysin");
```

上述代码的第一行压入一个函数类型的值，第二行将这个值赋给全局变量 `mysin`。完成这些修改后，我们就可以在 Lua 脚本中使用新函数 `mysin` 了。在接下来的一节中，我们会讨论如何用更好的方式把新的 C 函数与 Lua 链接在一起。现在，我们先来探索如何编写更好的 C 函数。

要编写一个更专业的正弦函数，必须检查其参数的类型，而辅助库可以帮助我们完成这个任务。函数 `luaL_checknumber` 可以检查指定的参数是否为一个数字：如果出现错误，该函数会抛出一个告知性的错误信息；否则，返回这个数字。只需对上面这个正弦函数稍作修改：

```
static int l_sin (lua_State *L) {
    double d = luaL_checknumber(L, 1);
    lua_pushnumber(L, sin(d));
    return 1; /* 返回值的个数 */
}
```

在做了上述修改后，如果调用 `mysin('a')` 就会出现如下的错误：

```
bad argument #1 to 'mysin' (number expected, got string)
```

函数 `luaL_checknumber` 会自动用参数的编号 (#1)、函数名 ("mysin")、期望的参数类型 (`number`) 及实际的参数类型 (`string`) 来填写错误信息。

下面是一个更复杂的示例，编写一个函数返回指定目录下的内容。由于 ISO C 中没有具备这种功能的函数，因此 Lua 没有在标准库中提供这样的函数。这里，我们假设使用一个 POSIX 兼容的操作系统。这个函数（在 Lua 语言中我们称之为 `dir`，在 C 语言中称之为 `l_dir`）以一个目录路径字符串作为参数，返回一个列表，列出该目录下的内容。例如，调用 `dir("/home/lua")` 会得到形如 `{".", "..", "src", "bin", "lib"}` 的表。该函数的完整代码参见示例 29.1。

示例 29.1 一个读取目录的函数

```
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include "lua.h"
#include "lauxlib.h"

/* 译者注：l_dir是在Lua中被调用的，以下代码中所有以lua_开头的函数都是在向Lua返回值 */
static int l_dir (lua_State *L) {
    DIR *dir;
    struct dirent *entry;
    int i;
    const char *path = luaL_checkstring(L, 1);

    /* 打开目录 */
    dir = opendir(path);
    if (dir == NULL) { /* 打开目录失败？ */
        lua_pushnil(L); /* 返回nil... */
        lua_pushstring(L, strerror(errno)); /* 和错误信息 */
        return 2; /* number of results */
    }
}
```

```

/* 创建结果表 */
lua_newtable(L);
i = 1;
while ((entry = readdir(dir)) != NULL) { /* 对于目录中的每一个元素 */
    lua_pushinteger(L, i++); /* 压入键 */
    lua_pushstring(L, entry->d_name); /* 压入值 */
    lua_settable(L, -3); /* table[i] = 元素名 */
}
closedir(dir);
return 1; /* 表本身就在栈顶 */
}

```

该函数先使用与 `luaL_checknumber` 类似的函数 `luaL_checkstring` 检查目录路径是否为字符串，然后使用函数 `opendir` 打开目录。如果无法打开目录，该函数会返回 `nil` 以及一条用函数 `strerror` 获取的错误信息。在打开目录后，该函数会创建一张新表，然后用目录中的元素填充这张新表（每次调用 `readdir` 都会返回下一个元素）。最后，该函数关闭目录并返回 1，在 C 语言中即表示该函数将其栈顶的值返回给了 Lua（请注意，函数 `lua_settable` 会从栈中弹出键和值。因此，循环结束后，栈顶的元素就是最终结果的表）。

在某些情况中， `l_dir` 的这种实现可能会造成内存泄漏。该函数调用的三个 Lua 函数（ `lua_newtable` 、 `lua_pushstring` 和 `lua_settable` ）均可能由于内存不足而失败。这三个函数中的任意一个执行失败都会引发错误，并中断函数 `l_dir` 的执行，进而也就无法调用 `closedir` 了。在第 32 章中，我们会看到能够避免此类错误的另一种实现。

29.2 延续（Continuation）^①

通过 `lua_pcall` 和 `lua_call` ，一个被 Lua 调用的 C 函数也可以回调 Lua 函数。标准库中有一些函数就是这么做的： `table.sort` 调用了排序函数， `string.gsub` 调用了替换函数，

^①译者注：本章的原文中有一些找不到对应中文名词的英文术语，涉及非对称式协程、编译原理、call/cc、CPS 等不少理论性内容，原著者直接假设了读者具有相关的背景，因而也并未对所有细节进行解释。在原文中对于部分术语的使用也与传统教科书和文献中的用法不同，如有不明之处，烦请读者查阅相关资料。以 Continuation 为例，它实际上是函数调用方式的一种，与 C 语言等使用栈帧（stackframe）记录函数调用的上下文的方式不同，continuation 使用的是 continuation record 而非栈帧；而在本书中，原著者使用 Continuation 表达了更多的含义。

`pcall` 和 `xpcall` 以保护模式来调用函数。如果你还记得 Lua 代码本身就是被 C 代码（宿主程序）调用的，那么你应该知道调用顺序类似于：C（宿主）调用 Lua（脚本），Lua（脚本）又调用了 C（库），C（库）又调用了 Lua（回调）。

通常，Lua 语言可以处理这种调用顺序；毕竟，与 C 语言的集成是 Lua 的一大特点。但是，有一种情况下，这种相互调用会有问题，那就是协程（coroutine）。

Lua 语言中的每个协程都有自己的栈，其中保存了该协程所挂起调用的信息。具体地说，就是该栈中存储了每一个调用的返回地址、参数及局部变量。对于 Lua 函数的调用，解释器只需要这个栈即可，我们将其称为软栈（soft stack）。然而，对于 C 函数的调用，解释器必须使用 C 语言栈。毕竟，C 函数的返回地址和局部变量都位于 C 语言栈中。

对于解释器来说，拥有多个软栈并不难；然而，ISO C 的运行时环境却只能拥有一个内部栈。因此，Lua 中的协程不能挂起 C 函数的执行：如果一个 C 函数位于从 `resume` 到对应 `yield` 的调用路径中，那么 Lua 无法保存 C 函数的状态以便于在下次 `resume` 时恢复状态。请考虑如下的示例（使用的是 Lua 5.1）：

```
co = coroutine.wrap(function ()
    print(pcall(coroutine.yield))
end)
co()
--> false      attempt to yield across metamethod/C-call boundary
```

函数 `pcall` 是一个 C 语言函数；因此，Lua 5.1 不能将其挂起，因为 ISO C 无法挂起一个 C 函数并在之后恢复其运行。

在 Lua 5.2 及后续版本中，用延续（continuation）改善了对这个问题的处理。Lua 5.2 使用长跳转（long jump）实现了 `yield`，并使用相同的方式实现了错误处理。长跳转简单地丢弃了 C 语言栈中关于 C 函数的所有信息，因而无法 `resume` 这些函数。但是，一个 C 函数 `foo` 可以指定一个延续函数（continuation function）`foo_k`，该函数也是一个 C 函数，在要恢复 `foo` 的执行时它就会被调用。也就是说，当解释器发现它应该恢复函数 `foo` 的执行时，如果长跳转已经丢弃了 C 语言栈中有关 `foo` 的信息，则调用 `foo_k` 来替代。

为了说得更具体些，我们将 `pcall` 的实现作为示例。在 Lua 5.1 中，该函数的代码如下：

```
static int luaB_pcall (lua_State *L) {
    int status;
    luaL_checkany(L, 1); /* 至少一个参数 */
    status = lua_pcall(L, luaL_gettop(L) - 1, LUA_MULTRET, 0);
```

```

    lua_pushboolean(L, (status == LUA_OK)); /* 状态 */
    lua_insert(L, 1); /* 状态是第一个结果 */
    return lua_gettop(L); /* 返回状态和所有结果 */
}

```

如果程序正在通过 `lua_pcall` 被调用的函数 `yield`, 那么后面就不可能恢复 `luaB_pcall` 的执行。因此, 如果我们在保护模式的调用下试图 `yield` 时, 解释器就会抛出异常。Lua 5.3 使用基本类似于示例 29.2 中的方式实现了 `pcall`。^①

示例 29.2 使用延续实现 `pcall`

```

static int finishpcall (lua_State *L, int status, intptr_t ctx) {
    (void)ctx; /* 未使用的参数 */
    status = (status != LUA_OK && status != LUA_YIELD);
    lua_pushboolean(L, (status == 0)); /* 状态 */
    lua_insert(L, 1); /* 状态是第一个结果 */
    return lua_gettop(L); /* 返回状态和所有结果 */
}

static int luaB_pcall (lua_State *L) {
    int status;
    luaL_checkany(L, 1);
    status = lua_pcallk(L, lua_gettop(L) - 1, LUA_MULTRET, 0,
                        0, finishpcall);
    return finishpcall(L, status, 0);
}

```

与 Lua 5.1 中的版本相比, 上述实现有三个重要的不同点: 首先, 新版本用 `lua_pcallk` 替换了 `lua_pcall`; 其次, 新版本在调用完 `lua_pcallk` 后把完成的状态传给了新的辅助函数 `finishpcall`; 第三, `lua_pcallk` 返回的状态除了 `LUA_OK` 或者一个错误外, 还可以是 `LUA_YIELD`。

如果没有发生 `yield`, 那么 `lua_pcallk` 的行为与 `lua_pcall` 的行为完全一样。但是, 如果发生 `yield`, 情况则大不相同。如果一个被原来 `lua_pcall` 调用的函数想要 `yield`, 那么 Lua 5.3 会像 Lua 5.1 版本一样引发错误。但当被新的 `lua_pcallk` 调用的函数 `yield` 时, 则不会

^① 在 Lua 5.2 中, 延续的相关 API 稍有不同。具体细节烦请参阅参考手册。

出现发生错误：Lua 会做一个长跳转并且丢弃 C 语言栈中有关 `luaB_pcall` 的元素，但是会在协程软栈（soft stack）中保存传递给函数 `lua_pcallk` 的延续函数（*continuation function*）的引用（在我们的示例中即 `finishpcall`）。后来，当解释器发现应该返回到 `luaB_pcall` 时（而这是不可能的），它就会调用延续函数。

当发生错误时，延续函数 `finishpcall` 也可能被调用。与原来的 `luaB_pcall` 不同，`finishpcall` 不能获取 `lua_pcallk` 所返回的值。因此，`finishpcall` 通过额外的参数 `status` 获取这个结果。当没有错误时，`status` 是 `LUA_YIELD` 而不是 `LUA_OK`，因此延续函数可以检查它是如何被调用的。当发生错误时，`status` 还是原来的错误码。

除了调用的状态，延续函数还接收一个上下文（*context*）。`lua_pcallk` 的第 5 个参数是一个任意的整型数，这个参数被当作延续函数的最后一个参数来传递（这个参数的类型为 `intptr_t`，该类型也允许将指针当作上下文传递）。这个值允许原来的函数直接向延续函数传递某些任意的信息（我们的示例没有使用这种机制）。

Lua 5.3 的延续体系是一种为了支持 `yield` 而设计的精巧机制，但它也不是万能的。某些 C 函数可能会需要给它们的延续传递相当多的上下文。例如，`table.sort` 将 C 语言栈用于递归，而 `string.gsub` 则必须跟踪捕获（`capture`），还要跟踪和一个用于存放部分结果的缓冲区。虽然这些函数能以“yieldable”的方式重写，但与增加的复杂性和性能损失相比，这样做似乎并不值得。

29.3 C 模块

Lua 模块就是一个代码段，其中定义了一些 Lua 函数并将其存储在恰当的地方（通常是在表中的元素）。为 Lua 编写的 C 语言模块可以模仿这种行为。除了 C 函数的定义外，C 模块还必须定义一个特殊的函数，这个特殊的函数相当于 Lua 库中的主代码段，用于注册模块中所有的 C 函数，并将它们存储在恰当的地方（通常也是表中的元素）。与 Lua 的主代码段一样，这个函数还应该初始化模块中所有需要初始化的其他东西。

Lua 通过注册过程感知到 C 函数。一旦一个 C 函数用 Lua 表示和存储，Lua 就会通过对其实地址（就是我们注册函数时提供给 Lua 的信息）的直接引用来调用它。换句话说，一旦一个 C 函数完成注册，Lua 调用它时就不再依赖于其函数名、包的位置以及可见性规则。通常，一个 C 模块中只有一个用于打开库的公共（外部）函数^①；其他所有的函数都是私有的，在

^①译者注：即前文中提到的打开函数，在本章中也与初始化函数混用。



C 语言中被声明为 `static`。

当我们使用 C 函数来扩展 Lua 程序时，将代码设计为一个 C 模块是个不错的想法。因为即使我们现在只想注册一个函数，但迟早（通常比想象中早）总会需要其他的函数。通常，辅助库为这项工作提供了一个辅助函数。宏 `luaL_newlib` 接收一个由 C 函数及其对应函数名组成的数组，并将这些函数注册到一个新表中。举个例子，假设我们要用之前定义的函数 `l_dir` 创建一个库。首先，必须定义这个库函数：

```
static int l_dir (lua_State *L) {
    同前
}
```

然后，声明一个数组，这个数组包含了模块中所有的函数及其名称。数组元素的类型为 `luaL_Reg`，该类型是由两个字段组成的结构体，这两个字段分别是函数名（字符串）和函数指针。

```
static const struct luaL_Reg mylib [] = {
    {"dir", l_dir},
    {NULL, NULL} /* 哨兵 */
};
```

在上例中，只声明了一个函数 (`l_dir`)。数组的最后一个元素永远是 `{NULL, NULL}`，并以此标识数组的结尾。最后，我们使用函数 `luaL_newlib` 声明一个主函数^①：

```
int luaopen_mylib (lua_State *L) {
    luaL_newlib(L, mylib);
    return 1;
}
```

对函数 `luaL_newlib` 的调用会新创建一个表，并使用由数组 `mylib` 指定的“函数名-函数指针”填充这个新创建的表。当 `luaL_newlib` 返回时，它把这个新创建的表留在了栈中，在表中它打开了这个库。然后，函数 `luaopen_mylib` 返回 1，表示将这个表返回给 Lua。

编写完这个库以后，我们还必须将其链接到解释器。如果 Lua 解释器支持动态链接的话，那么最简便的方法是使用动态链接机制 (dynamic linking facility)。在这种情况下，必须用代码 (Windows 系统下为 `mylib.dll`, Linux 类系统下为 `mylib.so`) 创建一个动态链接库，并将这个库放到 C 语言路径中的某个地方。在完成了这些步骤后，就可以使用 `require` 在 Lua 中直接加载这个模块了：

^①译者注：即打开函数。



```
local mylib = require "mylib"
```

上述的语句会将动态库 `mylib` 链接到 Lua，查找函数 `luaopen_mylib`，将其注册为一个 C 语言函数，然后调用它以打开模块（这也就解释了为什么 `luaopen_mylib` 必须使用跟其他 C 语言函数一样的原型）。

动态链接器必须知道函数 `luaopen_mylib` 的名字才能找到它。它总是寻找名为“`luaopen_+ 模块名`”这样的函数。因此，如果我们的模块名为 `mylib`，那么该函数应该命名为 `luaopen_mylib`（我们已经在第17章中讨论过有关该函数名的细节）。

如果解释器不支持动态链接，就必须连同新库一起重新编译 Lua 语言。除了重新编译，还需要以某种方式告诉独立解释器，它应该在打开一个新状态时打开这个库。一个简单做法是把 `luaopen_mylib` 添加到由 `luaL_openlibs` 打开的标准库列表中，这个列表位于文件 `linit.c` 中。

29.4 练习

练习 29.1：请使用 C 语言编写一个可变长参数函数 `summation`，来计算数值类型参数的和：

```
print(summation())          --> 0
print(summation(2.3, 5.4))   --> 7.7
print(summation(2.3, 5.4, -34)) --> -26.3
print(summation(2.3, 5.4, {}))
--> stdin:1: bad argument #3 to 'summation'
      (number expected, got table)
```

练习 29.2：请实现一个与标准库中的 `table.pack` 等价的函数。

练习 29.3：请编写一个函数，该函数接收任意个参数，然后逆序将其返回。

```
print(reverse(1, "hello", 20)) --> 20 hello 1
```

练习 29.4：请编写一个函数 `foreach`，该函数的参数为一张表和一个函数，然后对表中的每个键值对调用传入的函数。

```
foreach({x = 10, y = 20}, print)
--> x 10
--> y 20
```



(提示：在 Lua 语言手册中查一下函数 `lua_next`。)

练习 29.5：请重写练习 29.4 中的函数 `foreach`，让它所调用的函数支持 `yield`。

练习 29.6：用前面所有练习中的函数创建一个 C 语言模块。



30

编写 C 函数的技巧

官方的 C API 和辅助库都提供了一些机制来帮助用户编写 C 函数。本章将介绍这些机制，包括数组操作、字符串操作，以及如何在 C 语言中保存 Lua 语言的值。

30.1 数组操作

Lua 中的“数组”就是以特殊方式使用的表。像 `lua_settable` 和 `lua_gettable` 这种用来操作表的通用函数，也可用于操作数组。不过，C API 为使用整数索引的表的访问和更新提供了专门的函数：

```
void lua_geti (lua_State *L, int index, int key);
void lua_seti (lua_State *L, int index, int key);
```

Lua 5.3 之前的版本只提供了这些函数的原始版本，即 `lua_rawgeti` 和 `lua_rawseti`。这两个函数类似于 `lua_geti` 和 `lua_seti`，但进行的是原始访问（即不调用元方法）。当区别并不明显时（例如，表没有元方法），那么原始版本可能会稍微快一点。

`lua_geti` 和 `lua_seti` 的描述有一点令人困惑，因为其用了两个索引：`index` 表示表在栈中的位置，`key` 表示元素在表中的位置。当 `t` 为正数时，那么调用 `lua_geti(L, t, key)` 等价于如下的代码（否则，则必须对栈中的新元素进行补偿）：

```
lua_pushnumber(L, key);
lua_gettable(L, t);
```



调用 `lua_seti(L, t, key)` (`t` 仍然为正数值) 等价于：

```
lua_pushnumber(L, key);
lua_insert(L, -2); /* 把'key'放在之前的值下面 */
lua_settable(L, t);
```

作为使用这些函数的具体示例，示例 30.1 实现了函数 `map`，该函数对数组中的所有元素调用一个指定的函数，然后用此函数返回的结果替换掉对应的数组元素。

示例 30.1 C 语言中的函数 map

```
int l_map(lua_State *L) {
    int i, n;

    /* 第一个参数必须是一张表 (t) */
    luaL_checktype(L, 1, LUA_TTABLE);

    /* 第二个参数必须是一个函数 (f) */
    luaL_checktype(L, 2, LUA_TFUNCTION);

    n = luaL_len(L, 1); /* 获取表的大小 */

    for (i = 1; i <= n; i++) {
        lua_pushvalue(L, 2); /* 压入f */
        lua_geti(L, 1, i); /* 压入t[i] */
        lua_call(L, 1, 1); /* 调用f(t[i]) */
        lua_seti(L, 1, i); /* t[i] = result */
    }

    return 0; /* 没有返回值 */
}
```

这个示例还引入了三个新函数：`luaL_checktype`、`luaL_len` 和 `lua_call`。

函数 `luaL_checktype` (来自 `lauxlib.h`) 确保指定的参数具有指定的类型，否则它会引发一个错误。

原始的 `lua_len` (在上例中并未使用) 类似于长度运算符。由于元方法的存在，该运算符能够返回任意类型的对象，而不仅仅是数字；因此，`lua_len` 会在栈中返回其结果。函数



`luaL_len`（在上例中使用了，来自辅助库）会将长度作为整型数返回，如果无法进行强制类型转换则会引发错误。

函数 `lua_call` 做的是不受保护的调用，该函数类似于 `lua_pcall`，但在发生错误时 `lua_call` 会传播错误而不是返回错误码。在一个应用中编写主函数时，不应使用 `lua_call`，因为我们需要捕获所有的错误。不过，编写一个函数时，一般情况下使用 `lua_call` 是个不错的主意；如果发生错误，就留给关心错误的人去处理吧。

30.2 字符串操作

当 C 函数接收到一个 Lua 字符串为参数时，必须遵守两条规则：在使用字符串期间不能从栈中将其弹出，而且不应该修改字符串。

当 C 函数需要创建一个返回给 Lua 的字符串时，要求则更高。此时，是 C 语言代码负责缓冲区的分配/释放、缓冲区溢出，以及其他对 C 语言来说比较困难的任务。因此，Lua API 提供了一些函数来帮助完成这些任务。

标准 API 为两种最常用的字符串操作提供了支持，即子串提取和字符串连接。要提取子串，那么基本的操作 `lua_pushlstring` 可以获取字符串长度作为额外的参数。因此，如果要把字符串 `s` 从 `i` 到 `j`（包含）的子串传递给 Lua，就必须：

```
lua_pushlstring(L, s + i, j - i + 1);
```

举个例子，假设需要编写一个函数，该函数根据指定的分隔符（单个字符）来分割字符串，并返回一张包含子串的表。例如，调用 `split("hi:ho:there", ":")` 应该返回表 `{"hi", "ho", "there"}`。示例 30.2 演示了该函数的一种简单实现。

示例 30.2 分割字符串

```
static int l_split(lua_State *L) {
    const char *s = luaL_checkstring(L, 1);      /* 目标字符串 */
    const char *sep = luaL_checkstring(L, 2);    /* 分隔符 */
    const char *e;
    int i = 1;

    lua_newtable(L);   /* 结果表 */

    /* 依次处理每个分隔符 */
    while ((e = strchr(s + i, *sep)) != NULL) {
        /* ... */
    }
}
```



```

        while ((e = strchr(s, *sep)) != NULL) {
            lua_pushlstring(L, s, e - s); /* 压入子串 */
            lua_rawseti(L, -2, i++); /* 向表中插入 */
            s = e + 1; /* 跳过分隔符 */
        }

        /* 插入最后一个子串 */
        lua_pushstring(L, s);
        lua_rawseti(L, -2, i);

        return 1; /* 将结果表返回 */
    }
}

```

该函数无须缓冲区，并能处理任意长度的字符串，Lua 语言会负责处理所有的内存分配（由于我们创建表时知道其没有元表，因此可以用原始操作对其进行处理）。

要连接字符串，Lua 提供了一个名为 `lua_concat` 的特殊函数，该函数类似于 Lua 中的连接操作符 `(..)`，它会将数字转换为字符串，并在必要时调用元方法。此外，该函数还能一次连接两个以上的字符串。调用 `lua_concat(L, n)` 会连接（并弹出）栈最顶端的 `n` 个值，并将结果压入栈。

另一个有帮助的函数是 `lua_pushfstring`：

```
const char *lua_pushfstring (lua_State *L, const char *fmt, ...);
```

该函数在某种程度上类似于 C 函数 `sprintf`，它们都会根据格式字符串和额外的参数来创建字符串。然而，与 `sprintf` 不同，使用 `lua_pushfstring` 时不需要提供缓冲区。不管字符串有多大，Lua 都会动态地为我们创建。`lua_pushfstring` 会将结果字符串压入栈中并返回一个指向它的指针，该函数能够接受如下所示的指示符。

<code>%s</code>	插入一个以\0结尾的字符串
<code>%d</code>	插入一个 int
<code>%f</code>	插入一个 Lua 语言的浮点数
<code>%p</code>	插入一个浮点数
<code>%I</code>	插入一个 Lua 语言的整型数
<code>%c</code>	插入一个以 int 表示的单字节字符
<code>%U</code>	插入一个以 int 表示的 UTF-8 字节序列
<code>%%</code>	插入一个百分号



该函数不能使用诸如宽度或者精度之类的修饰符。^①

当只需连接几个字符串时，`lua_concat` 和 `lua_pushfstring` 都很有用。不过，如果需要连接很多字符串（或字符），那么像 14.7 节中那样逐个连接就会非常低效。此时，我们可以使用由辅助库提供的缓冲机制（*buffer facility*）。

缓冲机制的简单用法只包含两个函数：一个用于在组装字符串时提供任意大小的缓冲区；另一个用于将缓冲区中的内容转换为一个 Lua 字符串。^②示例 30.3 用源文件 `lstrlib.c` 中 `string.upper` 的实现演示了这些函数。

示例 30.3 函数 `string.upper`

```
static int str_upper (lua_State *L) {
    size_t l;
    size_t i;
    luaL_Buffer b;
    const char *s = luaL_checklstring(L, 1, &l);
    char *p = luaL_buffinit(L, &b, l);
    for (i = 0; i < l; i++)
        p[i] = toupper(uchar(s[i]));
    luaL_pushresultsize(&b, l);
    return 1;
}
```

使用辅助库中缓冲区的第一步是声明一个 `luaL_Buffer` 类型的变量。第二步是调用 `luaL_buffinit` 获取一个指向指定大小缓冲区的指针，之后就可以自由地使用该缓冲区来创建字符串了。最后需要调用 `luaL_pushresultsize` 将缓冲区中的内容转换为一个新的 Lua 字符串，并将该字符串压栈。其中，第二步调用时就确定了字符串的最终长度。通常情况下，像我们的示例一样，字符串的最终大小与缓冲区大小相等，但也可能更小。假如我们并不知道返回字符串的准确长度，但知道其最大不超过多少，那么可以保守地为其分配一个较大的空间。

请注意，`luaL_pushresultsize` 并未获取 Lua 状态作为其第一个参数。在初始化之后，缓冲区保存了对 Lua 状态的引用，因此在调用其他操作缓冲区的函数时无须再传递该状态。

^①指示符 p 是在 Lua 5.2 中引入的。指示符 I 和 U 是在 Lua 5.3 中引入的。

^②这两个函数在 Lua 5.2 中引入。



如果不知道返回结果大小的上限值，我们还可以通过逐步增加内容的方式来使用辅助库的缓冲区。辅助库提供了一些用于向缓冲区中增加内容的函数：`luaL_addvalue` 用于在栈顶增加一个 Lua 字符串，`luaL_addlstring` 用于增加一个长度明确的字符串，`luaL_addstring` 用于增加一个以\0 结尾的字符串，`luaL_addchar` 用于增加单个字符。这些函数的原型如下：

```
void luaL_buffinit (lua_State *L, luaL_Buffer *B);
void luaL_addvalue (luaL_Buffer *B);
void luaL_addlstring (luaL_Buffer *B, const char *s, size_t l);
void luaL_addstring (luaL_Buffer *B, const char *s);
void luaL_addchar (luaL_Buffer *B, char c);
void luaL_pushresult (luaL_Buffer *B);
```

示例 30.4 通过函数 `table.concat` 的一个简化的实现演示了这些函数的使用。

示例 30.4 函数 `table.concat` 一个简化的实现

```
static int tconcat (lua_State *L) {
    luaL_Buffer b;
    int i, n;
    luaL_checktype(L, 1, LUA_TTABLE);
    n = luaL_len(L, 1);
    luaL_buffinit(L, &b);
    for (i = 1; i <= n; i++) {
        lua_geti(L, 1, i); /* 从表中获取字符串 */
        luaL_addvalue(b); /* 将其放入缓冲区 */
    }
    luaL_pushresult(&b);
    return 1;
}
```

在该函数中，首先调用 `luaL_buffinit` 来初始化缓冲区。然后，向缓冲区中逐个增加元素，本例中用的是 `luaL_addvalue`。最后，`luaL_pushresult` 刷新缓冲区并在栈顶留下最终的结果字符串。

在使用辅助库的缓冲区时，我们必须注意一个细节。初始化一个缓冲区后，Lua 栈中可能还会保留某些内部数据。因此，我们不能假设在使用缓冲区之前栈顶仍然停留在最初的位置。此外，尽管使用缓冲区时我们可以将该栈用于其他用途，但在访问栈之前，对栈的压入



和弹出次数必须平衡。唯一的例外是 `luaL_addvalue`，该函数会假设要添加到缓冲区的字符串是位于栈顶的。

30.3 在 C 函数中保存状态

通常情况下，C 函数需要保存一些非局部数据，即生存时间超出 C 函数执行时间的数据。在 C 语言中，我们通常使用全局变量（`extern`）或静态变量来满足这种需求。然而，当我们为 Lua 编写库函数时^①，这并不是一个好办法。首先，我们无法在一个 C 语言变量中保存普通的 Lua 值。其次，使用这类变量的库无法用于多个 Lua 状态。

更好的办法是从 Lua 语言中寻求帮助。Lua 函数有两个地方可用于存储非局部数据，即全局变量和非局部变量，而 C API 也提供了两个类似的地方来存储非局部数据，即注册表（`registry`）和上值（`upvalue`）。

30.3.1 注册表

注册表（`registry`）是一张只能被 C 代码访问的全局表。^②通常情况下，我们使用注册表来存储多个模块间共享的数据。

注册表总是位于伪索引（*pseudo-index*）`LUA_REGISTRYINDEX` 中。伪索引就像是一个栈中的索引，但它所关联的值不在栈中。Lua API 中大多数接受索引作为参数的函数也能将伪索引作为参数，像 `lua_remove` 和 `lua_insert` 这种操作栈本身的函数除外。例如，要获取注册表中键为“Key”的值，可以使用如下的调用：

```
lua_getfield(L, LUA_REGISTRYINDEX, "Key");
```

注册表是一个普通的 Lua 表，因此可以使用除 `nil` 外的任意 Lua 值来检索它。不过，由于所有的 C 语言模块共享的是同一个注册表，为了避免冲突，我们必须谨慎地选择作为键的值。当允许其他独立的库访问我们的数据时，字符串类型的键尤为有用，因为这些库只需知道键的名字就可以了。对于这些键，选择名字时没有一种可以绝对避免冲突的方法；不过，诸如避免使用常见的名字，以及用库名或类似的东西作为键名的前缀，仍然是好的做法（用 `lua` 或者 `lualib` 作为前缀不是明智的选择）。

^①译者注：用 C 语言编写的库函数。

^②实际上，我们可以通过 Lua 中的调试函数 `debug.getregistry` 来访问注册表，但除了调试外真的不应该使用这个函数。

在注册表中不能使用数值类型的键，因为 Lua 语言将其用作引用系统 (*reference system*) 的保留字。引用系统由辅助库中的一对函数组成，有了这两个函数，我们在表中存储值时不必担心如何创建唯一的键。函数 `luaL_ref` 用于创建新的引用：

```
int ref = luaL_ref(L, LUA_REGISTRYINDEX);
```

上述调用会从栈中弹出一个值，然后分配一个新的整型的键，使用这个键将从栈中弹出的值保存到注册表中，最后返回该整型键，而这个键就被称为引用 (*reference*)。

顾名思义，我们主要是在需要在一个 C 语言结构体中保存一个指向 Lua 值的引用时使用引用。正如我们之前所看到的，不应该将指向 Lua 字符串的指针保存在获取该指针的函数之外。此外，Lua 语言甚至没有提供指向其他对象（例如表或者函数）的指针。因此，我们无法通过指针来引用 Lua 对象。当需要这种指针时，我们可以创建一个引用并将其保存在 C 语言中。

要将与引用 `ref` 关联的值压入栈中，只要这样写就行：

```
. lua_rawgeti(L, LUA_REGISTRYINDEX, ref);
```

最后，要释放值和引用，我们可以调用 `luaL_unref`：

```
luaL_unref(L, LUA_REGISTRYINDEX, ref);
```

在这句调用后，再次调用 `luaL_ref` 会再次返回相同的引用。

引用系统将 `nil` 视为一种特殊情况。无论何时为一个 `nil` 值调用 `luaL_ref` 都不会创建新的引用，而是会返回一个常量引用 `LUA_REFNIL`。如下的调用没什么用处：

```
luaL_unref(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

而如下的代码则会像我们期望地一样向栈中压入一个 `nil`：

```
lua_rawgeti(L, LUA_REGISTRYINDEX, LUA_REFNIL);
```

引用系统还定义了一个常量 `LUA_NOREF`，这是一个不同于其他合法引用的整数，它可以用于表示无效的引用。

当创建 Lua 状态时，注册表中有两个预定义的引用：

`LUA_RIDX_MAINTHREAD`

指向 Lua 状态本身，也就是其主线程。

`LUA_RIDX_GLOBALS`

指向全局变量。

另一种在注册表中创建唯一键的方法是，使用代码中静态变量的地址，C 语言的链接编辑器（link editor）会确保键在所有已加载的库中的唯一性^①。要使用这种方法，需要用到函数 `lua_pushlightuserdata`，该函数会在栈中压入一个表示 C 语言指针的值。下面的代码演示了如何使用这种方法在注册表中保存和获取字符串：

```
/* 具有唯一地址的变量 */
static char Key = 'k';

/* 保存字符串 */
lua_pushlightuserdata(L, (void *)&Key); /* 压入地址 */
lua_pushstring(L, myStr); /* 压入值 */
lua_settable(L, LUA_REGISTRYINDEX); /* registry[&Key] = myStr */

/* 获取字符串 */
lua_pushlightuserdata(L, (void *)&Key); /* 压入地址 */
lua_gettable(L, LUA_REGISTRYINDEX); /* 获正值 */
myStr = lua_tostring(L, -1); /* 转换为字符串 */
```

在31.5节中，我们将会讨论更多关于轻量级用户数据（light userdata）的细节。

为了简化将变量地址用作唯一键的用法，Lua 5.2 中引入了两个新函数：`lua_rawgetp` 和 `lua_rawsetp`。这两个函数类似于 `lua_rawgeti` 和 `lua_rawseti`，但它们使用 C 语言指针（转换为轻量级用户数据）作为键。使用这两个函数，可以将上面的代码重写为：

```
static char Key = 'k';

/* 保存字符串 */
lua_pushstring(L, myStr);
lua_rawsetp(L, LUA_REGISTRYINDEX, (void *)&Key);

/* 获取字符串 */
lua_rawgetp(L, LUA_REGISTRYINDEX, (void *)&Key);
myStr = lua_tostring(L, -1);
```

这两个函数都使用了原始访问。由于注册表没有元表，因此原始访问与普通访问相同，而且效率还会稍微高一些。

^①译者注：实际上就是代码重定位，可以参考《程序员的自我修养》一书。

30.3.2 上值

注册表提供了全局变量，而上值（*upvalue*）则实现了一种类似于 C 语言静态变量（只在特定的函数中可见）的机制。每一次在 Lua 中创建新的 C 函数时，都可以将任意数量的上值与这个函数相关联，而每个上值都可以保存一个 Lua 值。后面在调用该函数时，可以通过伪索引来自由地访问这些上值。

我们将这种 C 函数与其上值的关联称为闭包（*closure*）。C 语言闭包类似于 Lua 语言闭包。特别的，可以用相同的函数代码来创建不同的闭包，每个闭包可以拥有不同的上值。

接下来看一个简单的示例，让我们用 C 语言创建一个函数 newCounter（我们在第 9 章中用 Lua 语言定义过一个类似的函数）。该函数是一个工厂函数，每次调用时都会返回一个新的计数函数，如下所示：

```
c1 = newCounter()
print(c1(), c1(), c1())    --> 1    2    3
c2 = newCounter()
print(c2(), c2(), c1())    --> 1    2    4
```

尽管所有的计数器都使用相同的 C 语言代码，但它们各自都保留了独立的计数器。工厂函数的代码形如：

```
static int counter (lua_State *L); /* 前向声明 */

int newCounter (lua_State *L) {
    lua_pushinteger(L, 0);
    lua_pushcclosure(L, &counter, 1);
    return 1;
}
```

这里的关键函数是 `lua_pushcclosure`，该函数会创建一个新的闭包。`lua_pushcclosure` 的第二个参数是一个基础函数（示例中为 `counter`），第三个参数是上值的数量（示例中为 1）。在创建一个新的闭包前，我们必须将上值的初始值压栈。在此示例中，我们压入了零作为唯一一个上值的初始值。正如我们预想的那样，`lua_pushcclosure` 会将一个新的闭包留在栈中，并将其作为 `newCounter` 的返回值。

现在，来看一下 `counter` 的定义：

```
static int counter (lua_State *L) {
```

```

int val = lua_tointeger(L, lua_upvalueindex(1));
lua_pushinteger(L, ++val); /* 新值 */
lua_copy(L, -1, lua_upvalueindex(1)); /* 更新上值 */
return 1; /* 返回新值 */
}

```

这里的关键是宏 `lua_upvalueindex`，它可以生成上值的伪索引。特别的，表达式 `lua_upvalueindex(1)` 给出了正在运行的函数的第一个上值的伪索引，该伪索引同其他的栈索引一样，唯一区别的是它不存在于栈中。因此，调用 `lua_tointeger` 会以整型返回第一个（也是唯一一个）上值的当前值。然后，函数 `counter` 将新值 `++val` 压栈，并将其复制一份作为新上值的值，再将其返回。

接下来是一个更高级的示例，我们将使用上值来实现元组（tuple）。元组是一种具有匿名字段的常量结构，我们可以用一个数值索引来获取某个特定的字段，或者一次性地获取所有字段。在我们的实现中，将元组表示为函数，元组的值存储在函数的上值中。当使用数值参数来调用该函数时，函数会返回特定的字段。当不使用参数来调用该函数时，则返回所有字段。以下代码演示了元组的使用：

```

x = tuple.new(10, "hi", {}, 3)
print(x(1))    --> 10
print(x(2))    --> hi
print(x())      --> 10  hi  table: 0x8087878  3

```

在 C 语言中，我们会用同一个函数 `t_tuple` 来表示所有的元组，代码参见示例 30.5。

示例 30.5 元组的实现

```

#include "lauxlib.h"

int t_tuple(lua_State *L) {
    lua_Integer op = luaL_optinteger(L, 1, 0);
    if (op == 0) { /* 没有参数 */
        int i;
        /* 将每一个有效的上值压栈 */
        for (i = 1; !lua_isnone(L, lua_upvalueindex(i)); i++)
            lua_pushvalue(L, lua_upvalueindex(i));
        return i - 1; /* 值的个数 */
    }
}

```

```

    }

    else { /* 获取字段'op' */
        luaL_argcheck(L, 0 < op && op <= 256, 1,
                      "index out of range");
        if (lua_isnone(L, lua_upvalueindex(op)))
            return 0; /* 字段不存在 */
        lua_pushvalue(L, lua_upvalueindex(op));
        return 1;
    }
}

int t_new (lua_State *L) {
    int top = lua_gettop(L);
    luaL_argcheck(L, top < 256, top, "too many fields");
    lua_pushcclosure(L, t_tuple, top);
    return 1;
}

static const struct luaL_Reg tuplelib [] = {
    {"new", t_new},
    {NULL, NULL}
};

int luaopen_tuple (lua_State *L) {
    luaL_newlib(L, tuplelib);
    return 1;
}

```

由于调用元组时既可以使用数字作为参数也可以不用数字作为参数，因此 `t_tuple` 使用 `luaL_optinteger` 来获取可选参数。该函数类似于 `luaL_checkinteger`，但当参数不存在时不会报错，而是返回指定的默认值（本例中为零）。

C 语言函数中最多可以有 255 个上值，而 `lua_upvalueindex` 的最大索引值是 256。因此，我们使用 `luaL_argcheck` 来确保这些范围的有效性。

当访问一个不存在的上值时，结果是一个类型为 `LUA_TNONE` 的伪值（pseudo-value）（当访问的索引超出了当前栈顶时，也会得到一个类型为 `LUA_TNONE` 的伪值）。函数 `t_tuple` 使

用 `lua_isnone` 测试指定的上值是否存在。不过，我们永远不应该使用负数或者超过 256（C 语言函数上值的最多个数加 1）的索引值来调用 `lua_upvalueindex`，因此必须对用户提供的索引进行检查。函数 `luaL_argcheck` 可用于检查给定的条件，如果条件不符合，则会引发错误并返回一条友好的错误信息：

```
> t = tuple.new(2, 4, 5)
> t(300)
--> stdin:1: bad argument #1 to 't' (index out of range)
```

`luaL_argcheck` 的第三个参数表示错误信息的参数编号（上例中为 1），第四个参数表示对消息的补充（"index out of range"，表示索引超出范围）。

创建元组的函数 `t_new`（参见示例 30.5）很简单，由于其参数已经在栈中，因此该函数先检查字段的数量是否符合闭包中上值个数的限制，然后将所有上值作为参数调用 `lua_pushclosure` 来创建一个 `t_tuple` 的闭包。最后，数组 `tuplelib` 和函数 `luaopen_tuple`（参见示例 30.5）是创建 `tuple` 库的标准代码，该库只有一个函数 `new`。

30.3.3 共享的上值（Shared upvalue）

我们经常需要在同一个库的所有函数之间共享某些值或变量，虽然可以用注册表来完成这个任务，但也可以使用上值。

与 Lua 语言的闭包不同，C 语言的闭包不能共享上值，每个闭包都有其独立的上值。但是，我们可以设置不同函数的上值指向一张共同的表，这张表就成为一个共同的环境，函数在其中能够共享数据。

Lua 语言提供了一个函数，该函数可以简化同一个库中所有函数间共享上值的任务。我们已经使用 `luaL_newlib` 打开了 C 语言库。Lua 将这个函数实现为如下的宏：

```
#define luaL_newlib(L,lib) \
    (luaL_newlibtable(L,lib), luaL_setfuncs(L,lib,0))
```

宏 `luaL_newlibtable` 只是为库创建了一张新表（该表预先分配的大小等同于指定库中函数的数量）。然后，函数 `luaL_setfuncs` 将列表 `lib` 中的函数添加到位于栈顶的新表中。

我们在这里感兴趣的是 `luaL_setfuncs` 的第三个参数，这个参数给出了库中的新函数共享的上值个数。当调用 `lua_pushclosure` 时，这些上值的初始值应该位于栈顶。因此，如果要创建一个库，这个库中的所有函数共享一张表作为它们唯一的上值，则可以使用如下的代码：

```

/* 创建库的表 ('lib'是函数的列表) */
luaL_newlibtable(L, lib);
/* 创建共享上值 */
lua_newtable(L);
/* 将表 'lib' 中的函数加入到新库中，将之前的表共享为上值 */
luaL_setfuncs(L, lib, 1);

```

最后一个函数调用从栈中删除了这张共享表，只留下了新库。

30.4 练习

练习 30.1：用 C 语言实现一个过滤函数（filter function），该函数接收一个列表和一个判定条件，然后返回指定列表中满足该判定条件的所有元素组成的新列表。

```
t = filter({1, 3, 20, -4, 5}, function (x) return x < 5 end)
-- t = {1, 3, -4}
```

判定条件就是一个函数，该函数测试一些条件并返回一个布尔值。

练习 30.2：修改函数 `l_split`（见示例 30.2），使其可以处理包含\0 的字符串（可以用 `memchr` 替代 `strchr`）。

练习 30.3：用 C 语言重新实现函数 `transliterate`（练习 10.3）。

练习 30.4：通过修改 `transliterate` 实现一个库，让翻译表不是作为参数给出，而是直接由库给出。这个库应该提供如下的函数：

```

lib.settrans (table)    -- 设置翻译表
lib.gettrans ()          -- 获得翻译表
lib.transliterate(s)    -- 根据当前的表翻译's'
```

使用注册表来保存翻译表。

练习 30.5：使用上值保存翻译表并重新实现练习 30.4。

练习 30.6：你认为把翻译表作为库的状态的一部分而并非作为 `transliterate` 的一个参数是否是一种好的设计？

31

C 语言中的用户自定义类型

在上一章中，我们介绍了如何通过 C 语言编写新函数来扩展 Lua。本章将介绍如何用 C 语言编写新的类型来扩展 Lua。我们将从一个简单的例子入手，然后在本章中用元表和其他机制来扩展它。

这个示例实现了一种很简单的类型，即布尔数组。选用这个示例的主要动机在于它不涉及复杂的算法，便于我们专注于 API 的问题。不过尽管如此，这个示例本身还是很有用的。当然，我们可以在 Lua 中用表来实现布尔数组。但是，在 C 语言实现中，可以将每个布尔值存储在一个比特中，所使用的内存量不到使用表方法的 3%。

这个实现需要以下定义：

```
#include <limits.h>

#define BITS_PER_WORD (CHAR_BIT * sizeof(unsigned int))
#define I_WORD(i)      ((unsigned int)(i) / BITS_PER_WORD)
#define I_BIT(i)       (1 << ((unsigned int)(i) % BITS_PER_WORD))
```

`BITS_PER_WORD` 表示一个无符号整型数的位数，宏 `I_WORD` 用于根据指定的索引来计算存放相应比特位的字，`I_BIT` 用于计算访问这个字中相应比特位要用的掩码。

我们可以使用以下的结构体来表示布尔数组：

```
typedef struct BitArray {
    int size;
```

```
    unsigned int values[1]; /* 可变部分 */
} BitArray;
```

由于 C 89 标准不允许分配长度为零的数组，所以我们声明数组 `values` 的大小为 1，仅有一个占位符；等分配数组时，我们再设置数组的实际大小。下面这个表达式可以计算出拥有 `n` 个元素的数组大小：

```
sizeof(BitArray) + I_WORD(n - 1) * sizeof(unsigned int)
```

此处 `n` 减去 1 是因为原结构体中已经包含了一个元素的空间。

31.1 用户数据 (Userdata)

在第一个版本中，我们使用显式的调用来设置和获取值，如下所示：

```
a = array.new(1000)
for i = 1, 1000 do
    array.set(a, i, i % 2 == 0)      -- a[i] = (i % 2 == 0)
end
print(array.get(a, 10))           --> true
print(array.get(a, 11))           --> false
print(array.size(a))             --> 1000
```

后续我们将介绍如何同时支持像 `a:get(i)` 这样的面向对象风格和像 `a[i]` 这样的常见语法。在所有版本中，下列函数是一样的，参见示例 31.1。

示例 31.1 操作布尔数组

```
static int newarray (lua_State *L) {
    int i;
    size_t nbytes;
    BitArray *a;

    int n = (int) luaL_checkinteger(L, 1); /* 比特位的个数 */
    luaL_argcheck(L, n >= 1, 1, "invalid size");
    nbytes = sizeof(BitArray) + I_WORD(n - 1)*sizeof(unsigned int);
    a = (BitArray *)lua_newuserdata(L, nbytes);
```

```

    a->size = n;
    for (i = 0; i <= I_WORD(n - 1); i++)
        a->values[i] = 0; /* 初始化数组 */

    return 1; /* 新的用户数据已经位于栈中 */
}

static int setarray (lua_State *L) {
    BitArray *a = (BitArray *)lua_touserdata(L, 1);
    int index = (int) luaL_checkinteger(L, 2) - 1;

    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    luaL_argcheck(L, 0 <= index && index < a->size, 2,
                  "index out of range");
    luaL_checkany(L, 3);

    if (lua_toboolean(L, 3))
        a->values[I_WORD(index)] |= I_BIT(index); /* 置位 */
    else
        a->values[I_WORD(index)] &= ~I_BIT(index); /* 复位 */
    return 0;
}

static int getarray (lua_State *L) {
    BitArray *a = (BitArray *)lua_touserdata(L, 1);
    int index = (int) luaL_checkinteger(L, 2) - 1;

    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    luaL_argcheck(L, 0 <= index && index < a->size, 2,
                  "index out of range");

    lua_pushboolean(L, a->values[I_WORD(index)] & I_BIT(index));
    return 1;
}

```

下面让我们来一点一点地分析。

我们首先关心的是如何在 Lua 中表示一个 C 语言结构体。Lua 语言专门为这类任务提供了一个名为用户数据 (*userdata*) 的基本类型。用户数据为 Lua 语言提供了可以用来存储任何数据的原始内存区域，没有预定义的操作。

函数 `lua_newuserdata` 分配一块指定大小的内存，然后将相应的用户数据压栈，并返回该块内存的地址：

```
void *lua_newuserdata (lua_State *L, size_t size);
```

如果因为一些原因需要用其他方法来分配内存，可以很容易地创建一个指针大小的用户数据并在其中存储一个指向真实内存块的指针。我们将在第32章中看到使用这种技巧的例子。

示例 31.1 中的第一个函数 `newarray` 使用 `lua_newuserdata` 创建新的数组。`newarray` 的代码很简单，它检查了其唯一的参数（数组的大小，单位是比特），以字节为单位计算出数组的大小，创建了一个适当大小的用户数据，初始化用户数据的各个字段并将其返回给 Lua。

第二个函数是 `setarray`，它有三个参数：数组、索引和新的值。`setarray` 假定数组索引像 Lua 语言中的那样是从 1 开始的。因为 Lua 可以将任意值当作布尔类型，所以我们用 `luaL_checkany` 检查第三个参数，不过 `luaL_checkany` 只能确保该参数有一个值（可以是任意值）。如果用不符合条件的参数调用了 `setarray`，将会收到一条解释错误的信息，例如：

```
array.set(0, 11, 0)
--> stdin:1: bad argument #1 to 'set' ('array' expected)
array.set(a, 1)
--> stdin:1: bad argument #3 to 'set' (value expected)
```

示例 31.1 中的最后一个函数是 `getarray`，该函数类似于 `setarray`，用于获取元素。

我们还需要定义一个获取数组大小的函数和一些初始化库的额外代码，参见示例 31.2。

示例 31.2 布尔数组库的额外代码

```
static int getsize (lua_State *L) {
    BitArray *a = (BitArray *)lua_touserdata(L, 1);
    luaL_argcheck(L, a != NULL, 1, "'array' expected");
    lua_pushinteger(L, a->size);
    return 1;
}
```

```

static const struct luaL_Reg arraylib [] = {
    {"new", newarray}, {"set", setarray}, {"get", getarray},
    {"size", getsize}, {"NULL", NULL}
};

int luaopen_array (lua_State *L) {
    luaL_newlib(L, arraylib);
    return 1;
}

```

我们再一次使用了辅助库中的 `luaL_newlib`, 该函数创建了一张表, 并且用数组 `arraylib` 指定的“函数名-函数指针”填充了这张表。

31.2 元表 (Metatable)

我们当前的实现有一个重大的漏洞。假设用户写了一条像 `array.set(io.stdin, 1, false)` 这样的语句, 那么 `io.stdin` 的值会是一个带有指向文件流 (FILE *) 的指针的用户数据, `array.set` 会开心地认为它是一个合法的参数; 其后果可能就是内存崩溃 (或者幸运的话, 程序提示出现一个超出索引范围的错误)。这种行为对于任何一个 Lua 库而言都是不可接受的。无论你如何使用库, 都不应该破坏 C 语言的数据, 也不应该让 Lua 语言崩溃。

要区别不同类型的用户数据, 一种常用的方法是为每种类型创建唯一的元表。每次创建用户数据时, 用相应的元表进行标记; 每当获取用户数据时, 检查其是否有正确的元表。由于 Lua 代码不能改变用户数据的元表, 因此不能绕过这些检查。

我们还需要有个地方来存储这个新的元表, 然后才能用它来创建新的用户数据和检查指定的用户数据是否具有正确的类型。我们之前已经看到过, 存储元表有两种方法, 即存储在注册表中或者库函数的上值中。在 Lua 语言中, 惯例是将所有新的 C 语言类型注册到注册表中, 用类型名 (*type name*) 作为索引, 以元表作为值。由于注册表中还有其他索引, 所以必须谨慎地选择类型名以避免冲突。在我们的示例中将使用 "`Luabook.array`" 作为这个新类型的名称。

通常, 辅助库会提供一些函数来帮助实现这些内容。我们将使用的新的辅助函数包括:

```

int luaL_newmetatable (lua_State *L, const char *tname);
void luaL_getmetatable (lua_State *L, const char *tname);
void *luaL_checkudata (lua_State *L, int index,
                       const char *tname);

```

函数 `luaL_newmetatable` 会创建一张新表（被用作元表），然后将其压入栈顶，并将该表与注册表中的指定名称关联起来。函数 `luaL_getmetatable` 从注册表中获取与 `tname` 关联的元表。最后，`luaL_checkudata` 会检查栈中指定位置上的对象是否是与指定名称的元表匹配的用户数据。如果该对象不是用户数据，或者该用户数据没有正确的元表，`luaL_checkudata` 就会引发错误；否则，`luaL_checkudata` 就返回这个用户数据的地址。

现在让我们开始修改前面的代码。第一步是修改打开库的函数，让该函数为数组创建元表：

```

int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_newlib(L, arraylib);
    return 1;
}

```

下一步是修改 `newarray` 使其能为其新建的所有数组设置这个元表：

```

static int newarray (lua_State *L) {
    luaL_getmetatable(L, "LuaBook.array");
    lua_setmetatable(L, -2);

    return 1; /* 新的用户数据已经位于栈中 */
}

```

函数 `lua_setmetatable` 会从栈中弹出一个表，并将其设置为指定索引上对象的元表。在本例中，这个对象就是新建的用户数据。

最后，`setarray`、`getarray` 和 `getsize` 必须检查其第一个参数是否是一个有效的数组。为了简化这项任务，我们定义如下的宏：

```
#define checkarray(L) \
```

```
(BitArray *)luaL_checkudata(L, 1, "LuaBook.array")
```

有了这个宏，`getsize` 的定义就很简单了：

```
static int getsize(lua_State *L) {
    BitArray *a = checkarray(L);
    lua_pushinteger(L, a->size);
    return 1;
}
```

由于 `setarray` 和 `getarray` 还共享了用来读取和检查它们的第二个参数（索引）的代码，所以我们将其通用部分提取出来组成了一个新的辅助函数（`getparams`）。

示例 31.3 setarray/getarray 的新版本

```
static unsigned int *getparams(lua_State *L,
                               unsigned int *mask) {
    BitArray *a = checkarray(L);
    int index = (int)luaL_checkinteger(L, 2) - 1;
    luaL_argcheck(L, 0 <= index && index < a->size, 2,
                  "index out of range");
    *mask = I_BIT(index); /* 访问指定比特位的掩码 */
    return &a->values[I_WORD(index)]; /* 字所在的地址 */
}

static int setarray(lua_State *L) {
    unsigned int mask;
    unsigned int *entry = getparams(L, &mask);
    luaL_checkany(L, 3);
    if (lua_toboolean(L, 3))
        *entry |= mask;
    else
        *entry &= ~mask;
    return 0;
}
```

```

}

static int getarray (lua_State *L) {
    unsigned int mask;
    unsigned int *entry = getparams(L, &mask);
    lua_pushboolean(L, *entry & mask);
    return 1;
}

```

在这个新版本中，`setarray` 和 `getarray` 都很简单，参见示例 31.3。现在，如果调用它们时使用了无效的用户数据，我们将会收到一条相应的错误信息：

```

a = array.get(io.stdin, 10)
--> bad argument #1 to 'get' (LuaBook.array expected, got FILE*)

```

31.3 面向对象访问

下一步是将这种新类型转换成一个对象，以便用普通的面向对象语法来操作其实例。例如：

```

a = array.new(1000)
print(a:size())      --> 1000
a:set(10, true)
print(a:get(10))    --> true

```

请注意，`a:size()` 等价于 `a.size(a)`。因此，我们必须让表达式 `a.size` 返回函数 `getsize`。此处的关键机制在于元方法 `__index`。对于表而言，Lua 会在找不到指定键时调用这个元方法；而对于用户数据而言，由于用户数据根本没有键，所以 Lua 在每次访问时都会调用该元方法。

假设我们运行了以下代码：

```

do
    local metaarray = getmetatable(array.new(1))
    metaarray.__index = metaarray
    metaarray.set = array.set
    metaarray.get = array.get

```

```

metaarray.size = array.size
end

```

在第一行中，我们创建了一个数组用于获取分配给 `metaarray` 的元表（我们无法在 Lua 中设置用户数据的元表，但是可以获取用户数据的元表）。然后，将 `metaarray.__index` 设置为 `metaarray`。当对 `a.size` 求值时，因为对象 `a` 是一个用户数据，所以 Lua 在对象 `a` 中无法找到键"size"。因此，Lua 会尝试通过 `a` 的元表的 `__index` 字段来获取这个值，而这个字段正好就是 `metaarray`。由于 `metaarray.size` 就是 `array.size`，所以 `a.size(a)` 就是我们想要的 `array.size(a)`。

当然，用 C 语言也可以达到相同的效果，甚至还可以做得更好：既然数组有自己的操作的对象，那么在表 `array` 中也就无须包含这些操作了。我们的库只需导出一个用于创建新数组的函数 `new` 就行了，所有的其他操作都变成了对象的方法。C 语言代码同样可以直接注册这些方法。

操作 `getsize`、`getarray` 和 `setarray` 无须作任何改变，唯一需要改变的是注册它们的方式。换而言之，我们必须修改打开库的函数。首先，我们需要两个独立的函数列表，一个用于常规的函数，另一个用于方法。

```

static const struct luaL_Reg arraylib_f [] = {
    {"new", newarray},
    {NULL, NULL}
};

```

```

static const struct luaL_Reg arraylib_m [] = {
    {"set", setarray},
    {"get", getarray},
    {"size", getsize},
    {NULL, NULL}
};

```

新的打开函数 `luaopen_array` 必须创建元表，并把它赋给自己的 `__index` 字段，然后在元表中注册所有方法，创建和填充表 `array`：

```

int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array"); /* 创建元表 */
    lua_pushvalue(L, -1); /* 复制元表 */
    lua_setfield(L, -2, "__index"); /* mt.__index = mt */
}

```

```

    luaL_setfuncs(L, arraylib_m, 0); /* 注册元方法 */
    luaL_newlib(L, arraylib_f); /* 创建库 */
    return 1;
}

```

这里，我们再次使用了 `luaL_setfuncs` 将列表 `arraylib_m` 中的函数复制到栈顶的元表中。然后，调用 `luaL_newlib` 创建一张新表，并在该表中注册来自列表 `arraylib_f` 的函数。

最后，向新类型中新增一个 `__tostring` 元方法，这样 `print(a)` 就可以打印出“array”以及用括号括起来的数组的大小了。该函数如下：

```

int array2string (lua_State *L) {
    BitArray *a = checkarray(L);
    lua_pushfstring(L, "array(%d)", a->size);
    return 1;
}

```

调用 `lua_pushfstring` 格式化字符串，并将其保留在栈顶。我们还需要将 `array2string` 添加到列表 `arraylib_m` 中，以此将该函数加入到数组对象的元表中：

```

static const struct luaL_Reg arraylib_m [] = {
    {"__tostring", array2string},
    other methods (其他方法)
};

```

31.4 数组访问

另一种更好的面向对象的表示方法是，使用普通的数组符号来访问数组。只需简单地使用 `a[i]` 就可以替代 `a:get(i)`。对于上面的示例，由于函数 `setarray` 和 `getarray` 本身就是按照传递给相应元方法的参数的顺序来接收参数的，所以很容易做到这一点。一种快速的解决方案就是直接在 Lua 中定义这些元方法：

```

local metaarray = getmetatable(array.new(1))
metaarray.__index = array.get
metaarray.__newindex = array.set
metaarray.__len = array.size

```



必须在数组原来的实现中运行这段代码，无须修改面向对象的访问。这样，就可以使用标准语法了：

```
a = array.new(1000)
a[10] = true           -- 'setarray'
print(a[10])          -- 'getarray' --> true
print(#a)              -- 'getsize' --> 1000
```

如果还要更加完美，可以在 C 语言代码中注册这些元方法。为此，需要再次修改初始化函数，参见示例 31.4。

示例 31.4 新的初始化比特数组库的代码

```
static const struct luaL_Reg arraylib_f [] = {
    {"new", newarray},
    {NULL, NULL}
};

static const struct luaL_Reg arraylib_m [] = {
    {"__newindex", setarray},
    {"__index", getarray},
    {"__len", getsize},
    {"__tostring", array2string},
    {NULL, NULL}
};

int luaopen_array (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.array");
    luaL_setfuncs(L, arraylib_m, 0);
    luaL_newlib(L, arraylib_f);
    return 1;
}
```

在这个新版本中，仍然只有一个公有函数 new，所有的其他函数都只是特定操作的元方法。



31.5 轻量级用户数据

到现在为止，我们使用的用户数据称为完全用户数据 (*full userdata*)。Lua 语言还提供了另一种用户数据，称为轻量级用户数据 (*light userdata*)。

轻量级用户数据是一个代表 C 语言指针的值，即它是一个 `void *` 值。因为轻量级用户数据是一个值而不是一个对象，所以无须创建它（就好比我们也不需要创建数值）。要将一个轻量级用户数据放入栈中，可以调用 `lua_pushlightuserdata`:

```
void lua_pushlightuserdata (lua_State *L, void *p);
```

尽管名字差不多，但实际上轻量级用户数据和完全用户数据之间区别很大。轻量级用户数据不是缓冲区，而只是一个指针，它们也没有元表。与数值一样，轻量级用户数据不受垃圾收集器的管理。

有时，人们会将轻量级用户数据当作完全用户数据的一种廉价的替代物来使用，但这种用法并不普遍。首先，轻量级用户数据没有元表，因此没有办法得知其类型。其次，不要被“完全”二字所迷惑，实际上完全用户数据的开销也并不大。对于给定的内存大小，完全用户数据与 `malloc` 相比只增加了一点开销。

轻量级用户数据的真正用途是相等性判断。由于完全用户数据是一个对象，因此它只和自身相等；然而，一个轻量级用户数据表示的是一个 C 语言指针的值。因此，它与所有表示相同指针的轻量级用户数据相等。因此，我们可以使用轻量级用户数据在 Lua 语言中查找 C 语言对象。

我们已经见到过轻量级用户数据的一种典型用法，即在注册表中被用作键（见 30.3.1 节）。在这种情况下，轻量级用户数据的相等性是至关重要的。每次使用 `lua_pushlightuserdata` 压入相同的地址时，我们都会得到相同的 Lua 值，也就是注册表中相同的元素。

Lua 语言中另一种典型的场景是把 Lua 语言对象当作对应的 C 语言对象的代理。例如，输入/输出库使用 Lua 中的用户数据来表示 C 语言的流。当操作是从 Lua 语言到 C 语言时，从 Lua 对象到 C 对象的映射很简单。还是以输入/输出库为例，每个 Lua 语言流会保存指向其相应 C 语言流的指针。不过，当操作是从 C 语言到 Lua 语言时，这种映射就可能比较棘手。例如，假设在输入/输出系统中有某些回调函数（例如，那些告诉我们还有多少数据需要被读取的函数），回调函数接收它要操作的 C 语言流，那么如何从中得到其相应的 Lua 对象呢？由于 C 语言流是由 C 语言标准库定义的而不是我们定义的，因此无法在 C 语言流中存储任何东西。



轻量级用户数据为这种映射提供了一种好的解决方案。我们可以保存一张表，其中键是带有流地址的轻量级用户数据，值是 Lua 中表示流的完全用户数据。在回调函数中，一旦有了流地址，就可以将其作为轻量级用户数据，把它当作这张表的索引来获取对应的 Lua 对象（这张表很可能得是弱引用的；否则，这些完全用户数据可能永远不会被作为垃圾回收）。

31.6 练习

练习 31.1：修改 `setarray` 的实现，让它只能接受布尔值。

练习 31.2：我们可以将一个布尔数组看作是一个整型的集合（在数组中值为 `true` 的索引）。向布尔数组的实现中增加计算两个数组间并集和交集的函数，这两个函数接收两个布尔数组并返回一个新数组且不修改其参数。

练习 31.3：在上一个练习的基础上扩展，让我们可以用加法来获取两个数组的并集，用乘法来获取两个数组的交集

练习 31.4：修改元方法 `__tostring` 的实现，让它可以用一种恰当的方式显示数组的所有内容。请使用字符串缓冲机制（见 30.2 节）创建结果字符串。

练习 31.5：基于布尔数组的例子，为整数数组实现一个小型的 C 语言库。

整数数组类的实现由三个部分组成：一个头文件 `intarray.h`，一个源文件 `intarray.c` 和一个测试文件 `intarray-test.c`。头文件声明了三个全局变量：`intarray *array`、`int array_size` 和 `int array_capacity`。源文件实现了三个主要操作：`intarray *intarray_new(int capacity)`、`void intarray_free(intarray *array)` 和 `int intarray_get(intarray *array, int index)`。测试文件展示了如何使用这些操作。

如果希望了解更多关于这个库的信息，请参阅 [http://www.lua.org/pil/3.6.1.html#3.6.1](#)。

如果希望了解更多关于 `intarray` 库的实现细节，请参阅 [http://www.lua.org/pil/3.6.1.html#3.6.2](#)。

如果希望了解更多关于 `intarray` 库的测试，请参阅 [http://www.lua.org/pil/3.6.3.html#3.6.3](#)。

intarray *intarray_new(int capacity)

void intarray_free(intarray *array)

int intarray_get(intarray *array, int index)



32

管理资源

在上一章的布尔数组实现中，我们无须担心管理资源（managing resource）的事情。那些数组只需要内存，每个表示数组的用户数据都有各自的内存，而这些内存是由 Lua 来管理的。当一个数组成为垃圾时（即程序无法访问），Lua 最终会将其回收并释放其占用的内存。

然而，事情并非总是这么简单。有时，除了内存之外，对象还需要使用其他资源，例如文件描述符、窗口句柄及其他类似的东西（这些资源通常也是内存，但由系统的其他部分管理）。在这种情况下，当一个对象被当成垃圾收集后，其他资源也需要被释放。

正如我们在 23.6 节中所看到的，Lua 以 `_gc` 元方法的形式提供了析构器。为了完整地演示在 C 语言中对该元方法和 API 的使用，本章中我们会开发两个使用外部功能的示例。第一个示例是遍历目录的函数的另一种实现方式，第二个（更重要）示例与 *Expat* 有关，它是一个开源的 XML 解析器。

32.1 目录迭代器

在 29.1 节中，我们实现了函数 `dir`，该函数会遍历目录并返回一张包含指定目录下所有内容的表。本章中对 `dir` 新的实现会返回一个迭代器，每次调用这个迭代器时它都会返回一个新元素。通过这种实现，我们就能使用如下的循环来遍历目录：

```
for fname in dir.open(".") do
    print(fname)
end
```



要在 C 语言中遍历一个目录，我们需要用到 DIR 结构体。DIR 的实例由 `opendir` 创建，且必须通过调用 `closedir` 显式地释放^①。在之前的实现中，我们将 DIR 的实例当作局部变量，并在获取最后一个文件名后释放了它。而在新实现中，由于必须通过多次调用来查询该值，因此不能把 DIR 的实例保存到局部变量中。此外，不能在获取最后一个文件名后再释放 DIR 的实例，因为如果程序从循环中跳出，那么迭代器永远不会获取最后一个文件名。因此，为了确保 DIR 的实例能被正确释放，需要把该实例的地址存入一个用户数据中，并且用这个用户数据的元方法 `_gc` 来释放该结构体。

尽管用户数据在我们的实现中处于核心地位，但这个表示目录的用户数据并不一定需要对 Lua 可见。函数 `dir.open` 会返回一个 Lua 可见的迭代函数，而目录可以作为迭代函数的一个上值。这样，迭代函数能直接访问这个结构体，而 Lua 代码则不能（也没有必要）。

总之，我们需要三个 C 语言函数。首先，我们需要函数 `dir.open`，该函数是一个工厂函数，Lua 调用该函数来创建迭代器；它必须打开一个 DIR 结构体，并将这个结构体作为上值创建一个迭代函数的闭包。其次，我们需要迭代函数。最后，我们需要 `_gc` 元方法，该元方法用于释放 DIR 结构体。通常情况下，我们还需要一个额外的函数进行一些初始化工作，例如为目录创建和初始化元表。

先来看函数 `dir.open`，参见示例 32.1。

示例 32.1 工厂函数 `dir.open`

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

#include "lua.h"
#include "lauxlib.h"

/* 迭代函数的前向声明 */
static int dir_iter (lua_State *L);

static int l_dir (lua_State *L) {
    const char *path = luaL_checkstring(L, 1);
```

^①译者注：在 C 语言中，实际上不存在实例的概念，作者在此要表达的意思是“一个 DIR 类型的变量”，请注意合理地理解后文中作者的表述。当然，对于实际的 C 代码来说，获取到的实际是一个指向 DIR 类型变量的指针。



```
/* 创建一个保存DIR结构体的用户数据 */
/* 译者注：请注意这里的用户数据保存的是一个‘指向DIR类型结构体的指针’ */
DIR **d = (DIR **)lua_newuserdata(L, sizeof(DIR *));

/* 预先初始化 */
*d = NULL;

/* 设置元表 */
 luaL_getmetatable(L, "LuaBook.dir");
lua_setmetatable(L, -2);

/* 尝试打开指定目录 */
/* 译者注：opendir返回的是一个指向DIR类型结构体的指针
   虽然作者在下文中一直没有明确地指出指针的概念，但实际上所有对该结构体的操作都是通过这个指针进行的。
   请读者注意理解 */
*d = opendir(path);
if (*d == NULL) /* 打开目录失败？ */
    luaL_error(L, "cannot open %s: %s", path, strerror(errno));

/* 创建并返回迭代函数；该函数唯一的上值，即代表目录的用户数据本身就位于栈顶 */
lua_pushcclosure(L, dir_iter, 1);
return 1;
}
```

在这个函数中要注意的是，必须在打开目录前先创建用户数据。如果先打开目录再调用 `lua_newuserdata`，那么会引发内存错误，该函数会丢失并泄漏 DIR 结构体^①。如果顺序正确，DIR 结构体一旦被创建就会立即与用户数据相关联；无论此后发生什么，元方法 `_gc` 最终都会将其释放。

另一个需要注意的点是用户数据的一致性。一旦设置了元表，元方法 `_gc` 就一定会被调用。因此，在设置元表前，我们需要使用 `NUL` 预先初始化用户数据，以确保用户数据具有定义明确的值。

^①译者注：函数 `opendir` 会在内部使用 `malloc` 分配 DIR 结构体并返回指向该结构体的指针，如果不能保存这个指针，那么后续也没有办法释放 `malloc` 分配的内存，从而造成内存泄漏。



下一个函数是 `dir_iter` (在示例 32.2 中), 也就是迭代器本身。

示例 32.2 dir 库中的其他函数

```
static int dir_iter (lua_State *L) {
    DIR *d = *(DIR **)lua_touserdata(L, lua_upvalueindex(1));
    struct dirent *entry = readdir(d);
    if (entry != NULL) {
        lua_pushstring(L, entry->d_name);
        return 1;
    }
    else return 0; /* 遍历完成 */
}

static int dir_gc (lua_State *L) {
    DIR *d = *(DIR **)lua_touserdata(L, 1);
    if (d) closedir(d);
    return 0;
}

static const struct luaL_Reg dirlib [] = {
    {"open", l_dir},
    {NULL, NULL}
};

int luaopen_dir (lua_State *L) {
    luaL_newmetatable(L, "LuaBook.dir");
    /* 设置__gc字段 */
    lua_pushcfunction(L, dir_gc);
    lua_setfield(L, -2, "__gc");

    /* 创建库 */
    luaL_newlib(L, dirlib);
    return 1;
}
```



上述代码很简单，它从上值中获取 DIR 结构体的地址，然后调用 `readdir` 读取下一个元素。

函数 `dir_gc`（也在示例 32.2 中）就是元方法 `__gc`，该元方法用于关闭目录。正如之前提到的，该元方法必须做好防御措施：如果初始化时出现错误，那么目录可能会是 `NULL`。

示例 32.2 中的最后一个函数 `luaopen_dir` 用于打开 `dir`，它是只有一个函数的库。

整个示例中还有一点需要注意。`dir_gc` 似乎应该检查其参数是否为一个目录以及目录是否已经被关闭；否则，恶意用户可能会用其他类型的用户数据（例如，一个文件）来调用 `dir_gc` 或者关闭一个目录两次，这样会造成灾难性后果。然而，Lua 程序是无法访问这个函数的：该函数被保存在目录的元表中^①，而用户数据又被保存为迭代函数的上值，因此 Lua 代码无法访问这些目录。

32.2 XML 解析器

接下来，我们介绍一种使用 Lua 语言编写的 Expat 绑定（binding）的简单实现，称为 `lxp`^②。Expat 是一个用 C 语言编写的开源 XML1.0 解析器，实现了 SAX，即 *Simple API for XML*。SAX 是一套基于事件的 API，这就意味着一个 SAX 解析器在读取 XML 文档时会边读取边通过回调函数向应用上报读取到的内容。例如，如果让 Expat 解析形如 "`<tag cap="5">hi</tag>`" 的字符串，那么 Expat 会生成三个事件：当读取到子串 "`<tag cap="5">`" 时，生成开始元素（*start-element*）事件；当读取到 "`hi`" 时，生成文本（*text*）事件，也称为字符数据（*character data*）事件；当读取到 "`</tag>`" 时，生成结束元素（*end-element*）事件。每个事件都会调用应用中相应的回调处理器（*callback handler*）。

在此我们不会介绍整个 Expat 库，只关注于那些用于演示与 Lua 交互的新技术部分。虽然 Expat 可以处理很多种不同的事件，但我们只考虑前面示例中所提到的三个事件（开始元素、结束元素和文本事件）。^③

本例中用到的 Expat API 很少。首先，我们需要用于创建和销毁 Expat 解析器的函数：

```
XML_Parser XML_ParserCreate (const char *encoding);
void XML_ParserFree (XML_Parser p);
```

^①译者注：即 DIR 所对应用数据的元表中。

^②译者注：此处的 binding 类似于 SL4J 与 Log4J、Logback 的关系，是接口与实现分离的一种模式。

^③LuaExpat 包提供了非常完整的 Expat 接口。

参数 `encoding` 是可选的，本例中将使用 `NULL`。

当解析器创建完成后，必须注册回调处理器：

```
void XML_SetElementHandler(XML_Parser p,
                           XML_StartElementHandler start,
                           XML_EndElementHandler end);
void XML_SetCharacterDataHandler(XML_Parser p,
                                 XML_CharacterDataHandler hndl);
```

第一个函数为开始元素和结束元素事件注册了处理函数，第二个函数为文本（XML 术语中的字符数据，*character data*）事件注册了处理函数。

所有回调处理函数的第一个参数都是用户数据，开始元素事件的处理函数还能接收标签名（tag name）及其属性（attribute）：

```
typedef void (*XML_StartElementHandler)(void *userData,
                                         const char *name,
                                         const char **atts);
```

属性是一个以 `NULL` 结尾的字符串数组，其中每对连续的字符串保存一个属性的名称和值。结束元素事件函数除了用户数据外还有一个额外的参数，即标签名：

```
typedef void (*XML_EndElementHandler)(void *userData,
                                       const char *name);
```

最后，文本事件处理函数只接收文本作为额外参数，该文本字符串不是以 `NULL` 结尾的，它有一个显式的长度：

```
typedef void (*XML_CharacterDataHandler)(void *userData,
                                         const char *s,
                                         int len);
```

为了将文本输入 Expat，可以使用如下的函数：

```
int XML_Parse (XML_Parser p, const char *s, int len, int isLast);
```

Expat 通过连续调用函数 `XML_Parse` 一段一段地接收要解析的文档。`XML_Parse` 的最后一个参数，布尔类型的 `isLast`，告知 Expat 该片段是否是文档的最后一个片段。如果检测到解析

错误，`XML_Parse` 返回零（`Expat` 还提供了用于获取错误信息的函数，但为了简单起见，此处忽略了错误信息）。

`Expat` 中要用到的最后一个函数允许我们设置传递给事件处理函数的用户数据：

```
void XML_SetUserData (XML_Parser p, void *userData);
```

现在，让我们看一下如何在 `Lua` 中使用这个库。第一种方法是一种直接的方法，即简单地把所有函数导出给 `Lua`。另一个更好的方法是让这些函数适配 `Lua`。例如，因为 `Lua` 语言不是强类型的，所以不需要为每一种回调函数设置不同的函数。我们可以做得更好，甚至免去所有注册回调函数的函数。我们要做的只是在创建解析器时提供一个包含所有事件处理函数的回调函数表，其中每一个键值对是与相应事件对应的键和事件处理函数。例如，如果需要打印出一个文档的布局（`layout`），可以使用如下的回调函数表：

```
local count = 0

callbacks = {
    StartElement = function (parser, tagname)
        io.write("+ ", string.rep(" ", count), tagname, "\n")
        count = count + 1
    end,
    EndElement = function (parser, tagname)
        count = count - 1
        io.write("- ", string.rep(" ", count), tagname, "\n")
    end,
}
```

输入内容"`<to> <yes/> </to>`" 时，这些事件处理函数会打印出如下内容：

```
+ to
+ yes
- yes
- to
```

有了这个 API，我们就不再需要那些操作回调函数的函数了，可以直接在回调函数表中操作它们。因此，整个 API 只需用到三个函数：一个用于创建解析器，一个用于解析文本，一个用于关闭解析器。实际上，我们可以将后两个函数实现为解析器对象的方法。该 API 的典型用法形如：

```

local lxp = require "lxp"

p = lxp.new(callbacks)          -- 创建新的解析器

for l in io.lines() do         -- 迭代输入文本
    assert(p:parse(l))        -- 解析一行
    assert(p:parse("\n"))      -- 增加换行符
end

assert(p:parse())              -- 解析文档
p:close()                      -- 关闭解析器

```

现在，让我们来看看如何实现它。首先要决定如何在 Lua 语言中表示一个解析器。我们会很自然地想到使用用户数据来包含 C 语言结构体，但是需要在用户数据中放些什么东西呢？我们至少需要实际的 Expat 解析器和回调函数表。由于这些解析器对象都是 Expat 回调函数接收的，并且回调函数需要调用 Lua 语言，因此还需要保存 Lua 状态。我们可以直接在 C 语言结构体中保存 Expat 解析器和 Lua 状态（它们都是 C 语言值）；而对于作为 Lua 语言值的回调函数表，一个选择是在注册表中为其创建引用并保存该引用（我们将在练习 32.2 中讨论这个做法），另一个选择是使用用户值 (*user value*)。每个用户数据都可以有一个与其直接关联的唯一的 Lua 语言值，这个值就被叫作用户值^①。要是使用这种方式的话，解析器对象的定义形如：

```

#include <stdlib.h>
#include "expat.h"
#include "lua.h"
#include "lauxlib.h"

typedef struct lxp_userdata {
    XML_Parser parser;           /* 关联的Expat解析器 */
    lua_State *L;                /* 外部为注册函数 */
} lxp_userdata;

```

下一步是创建解析器对象的函数 `lxp_make_parser`，参见示例 32.3。

^① 在 Lua 5.2 中，用户值必须是表。

示例 32.3 创建 XML 解析器对象的函数

```

/* 回调函数的前向声明 */
static void f_StartElement (void *ud,
                            const char *name,
                            const char **atts);
static void f_CharData (void *ud, const char *s, int len);
static void f_EndElement (void *ud, const char *name);

static int lxp_make_parser (lua_State *L) {
    XML_Parser p;

    /* (1) 创建解析器对象 */
    lxp_userdata *xpu = (lxp_userdata *)lua_newuserdata(L,
                                                          sizeof(lxp_userdata));
    xpu->parser = NULL;

    /* 预先初始化以防止错误发生 */
    xpu->parser = XML_ParserCreate(NULL);
    if (!p)
        luaL_error(L, "XML_ParserCreate failed");

    /* (2) 创建Expat解析器 */
    p = xpu->parser;
    luaL_checktype(L, 1, LUA_TTABLE);
    lua_pushvalue(L, 1); /* 回调函数表入栈 */
    lua_setuservalue(L, -2); /* 将回调函数表设为用户值 */

    /* (3) 检查并保存回调函数表 */
    XML_SetUserData(p, xpu);
}

```

```

XML_SetElementHandler(p, f_StartElement, f_EndElement);
XML_SetCharacterDataHandler(p, f_CharData);
return 1;
}

```

该函数有四个主要步骤。

- 第一步遵循常见的模式：先创建用户数据，然后使用一致性的值预先初始化用户数据，最后设置用户数据的元表（其中的预先初始化确保如果在初始化过程中发生了错误，析构器能够以一致性的状态处理用户数据）。
- 第二步中，该函数创建了一个 Expat 解析器，将其存储到用户数据中，并检查了错误。
- 第三步保证该函数的第一个参数是一个表（回调函数表），并将其作为用户值赋给了新的用户数据。
- 最后一步初始化 Expat 解析器，将用户数据设为传递给回调函数的对象，并设置了回调函数。请注意，这些回调函数对于所有的解析器来说都是相同的；毕竟，用户无法在 C 语言中动态地创建新函数。不同点在于，这些固定的 C 语言函数会通过回调函数表来决定每次应该调用哪些 Lua 函数。

接下来是解析函数 `lxp_parse`（参见示例 32.4），该函数用于解析 XML 数据片段。

示例 32.4 解析 XML 片段的函数

```

static int lxp_parse (lua_State *L) {
    int status;
    size_t len;
    const char *s;
    lxp_userdata *xpu;

    /* 获取并检查第一个参数（应该是一个解析器） */
    xpu = (lxp_userdata *) luaL_checkudata(L, 1, "Expat");

    /* 检查解析器是否已经被关闭了 */
    luaL_argcheck(L, xpu->parser != NULL, 1, "parser is closed");

    /* 获取第二个参数（一个字符串） */
    s = luaL_optlstring(L, 2, NULL, &len);

```

```

/* 将回调函数表放在栈索引为3的位置 */
lua_settop(L, 2);
lua_getuservalue(L, 1);

xpu->L = L; /* 设置Lua状态 */

/* 调用Expat解析字符串 */
status = XML_Parse(xpu->parser, s, (int)len, s == NULL);

/* 返回错误码 */
lua_pushboolean(L, status);
return 1;
}

```

该函数有两个参数，即解析器对象（方法本身）和一个可选的 XML 数据。如果调用该函数时未传入 XML 数据，那么它会通知 Expat 文档已结束。

当 `lxp_parse` 调用 `XML_Parse` 时，后一个函数会为指定文件片段中找到的每个相关元素调用处理函数。这些处理函数需要访问回调函数表，因此 `lxp_parse` 会将这个表放到栈索引为 3（正好在参数后）的位置。在调用 `XML_Parse` 时还有一个细节：请注意，该函数的最后一个参数会告诉 Expat 文本的指定片段是否为最后一个片段。当不带参数调用 `parse` 时，`s` 是 `NULL`，这样最后一个参数就为真。

现在我们把注意力放到处理回调的 `f_CharData`、`f_StartElement` 和 `f_EndElement` 函数上。这三个函数的代码结构类似，它们都会检查回调函数表是否为指定的事件定义了 Lua 处理函数，如果是，则准备好参数并调用这个处理函数。

首先来看示例 32.5 中的处理函数 `f_CharData`。

示例 32.5 字符数据事件的处理函数

```

static void f_CharData (void *ud, const char *s, int len) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;

    /* 从回调函数表中获取处理函数 */
    lua_getfield(L, 3, "CharacterData");
}

```

```

if (lua_isnil(L, -1)) { /* 没有处理函数? */
    lua_pop(L, 1);
    return;
}

lua_pushvalue(L, 1); /* 解析器压栈 ('self') */
lua_pushlstring(L, s, len); /* 压入字符串数据 */
lua_call(L, 2, 0); /* 调用处理函数 */
}

```

该函数的代码很简单。由于创建解析器时调用了 XML_SetUserData，所以处理函数的第一个参数是 lxp_userdata 结构体。在获取 Lua 状态后，处理函数就可以访问由 lxp_parse 设置的位于栈索引 3 位置的回调函数表，以及位于栈索引 1 位置的解析器。然后，该函数就可以用解析器和字符串（一个字符串）作为参数调用 Lua 中对应的处理函数了（如果存在的话）。

处理函数 f_EndElement 与 f_CharData 十分相似，参见示例 32.6。

示例 32.6 结束元素事件的处理函数

```

static void f_EndElement (void *ud, const char *name) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;

    lua_getfield(L, 3, "EndElement");
    if (lua_isnil(L, -1)) { /* 没有处理函数? */
        lua_pop(L, 1);
        return;
    }

    lua_pushvalue(L, 1); /* 解析器压栈 ('self') */
    lua_pushstring(L, name); /* 压入标签名 */
    lua_call(L, 2, 0); /* 调用处理函数 */
}

```

该函数也以解析器和标签名（也是一个字符串，但是以 null 结尾）作为参数调用相应的 Lua 处理函数。

示例 32.7 演示了最后一个处理函数 `f_StartElement`。

示例 32.7 开始元素事件的处理函数

```
static void f_StartElement (void *ud,
                           const char *name,
                           const char **atts) {
    lxp_userdata *xpu = (lxp_userdata *)ud;
    lua_State *L = xpu->L;

    lua_getfield(L, 3, "StartElement");
    if (lua_isnil(L, -1)) { /* 没有处理函数? */
        lua_pop(L, 1);
        return;
    }

    lua_pushvalue(L, 1); /* 解析器压栈 ('self') */
    lua_pushstring(L, name); /* 压入标签名 */

    /* 创建并填充属性表 */
    lua_newtable(L);
    for (; *atts; atts += 2) {
        lua_pushstring(L, *(atts + 1));
        lua_setfield(L, -2, *atts); /* table[*atts] = *(atts+1) */
    }

    lua_call(L, 3, 0); /* 调用处理函数 */
}
```

该函数以解析器、标签名和一个属性列表为参数，调用了 Lua 处理函数。处理函数 `f_StartElement` 比其他的处理函数稍微复杂一点，因为它需要将属性的标签列表转换为 Lua 语言。`f_StartElement` 使用了一种非常自然的转换方法，即创建了一张包含属性名和属性值的表。例如，类似这样的开始标签

```
<to method="post" priority="high">
```

会产生如下的属性表：

```
{method = "post", priority = "high"}
```

解析器的最后一个方法是 `close`, 参见示例 32.8。

示例 32.8 关闭 XML 解析器的方法

```
static int lxp_close (lua_State *L) {
    lxp_userdata *xpu =
        (lxp_userdata *)luaL_checkudata(L, 1, "Expat");

    /* 释放Expat解析器（如果有） */
    if (xpu->parser)
        XML_ParserFree(xpu->parser);
    xpu->parser = NULL; /* 避免重复关闭 */
    return 0;
}
```

当关闭解析器时, 必须释放其资源, 也就是 `Expat` 结构体。请注意, 由于在创建解析器时可能会发生错误, 解析器可能没有这些资源。此外还需注意, 如何像关闭解析器一样, 在一致的状态中保存解析器, 这样当我们试图再次关闭解析器或者垃圾收集器结束解析器时才不会产生问题。实际上, 我们可以将这个函数当作终结器来使用。这样便可以确保, 即使程序员没有关闭解析器, 每个解析器最终也会释放其资源。

示例 32.9 是最后一步, 它演示了打开库的 `luaopen_lxp`。`luaopen_lxp` 将前面所有的部分组织到了一起。

示例 32.9 lxp 库的初始化代码

```
static const struct luaL_Reg lxp_meths[] = {
    {"parse", lxp_parse},
    {"close", lxp_close},
    {"__gc", lxp_close},
    {NULL, NULL}
};

static const struct luaL_Reg lxp_funcs[] = {
    {"new", lxp_make_parser},
    {NULL, NULL}
};
```

```

};

int luaopen_lxp (lua_State *L) {
    /* 创建元表 */
    luaL_newmetatable(L, "Expat");

    /* metatable.__index = metatable */
    lua_pushvalue(L, -1);
    lua_setfield(L, -2, "__index");

    /* 注册方法 */
    luaL_setfuncs(L, lxp_meths, 0);

    /* 注册 (只有 lxp.new ) */
    luaL_newlib(L, lxp_funcs);

    return 1;
}

```

此处使用的代码结构与31.3节中面向对象的布尔数组的示例相同，我们创建一个元表，将元表的 `__index` 字段指向自身，并将所有的方法放入其中。因此，需要一个具备解析器方法的列表 (`lxp_meths`)，还需要一个包含库函数的列表 (`lxp_funcs`)，像常见的面向对象的库一样，这个列表中只有一个创建新解析器的函数。

32.3 练习

练习 32.1：修改示例 32.2 中的函数 `dir_iter`，使其在结束遍历时关闭 DIR 结构体。这样修改后，由于程序知道不再需要 DIR，所以无须等待垃圾收集器来释放资源。

（当关闭目录时，应该把保存在用户数据中的地址设为 `NULL`，以通知析构器该目录已经关闭。此外，`dir_iter` 在使用目录前也必须检查目录是否已经关闭。）

练习 32.2：在 `lxp` 的例子中，我们使用用户值将回调函数表和表示解析器的用户数据关联在一起。由于 C 语言回调函数接收到的是 `lxp_userdata` 结构体，而该结构体并不能提供对表的直接访问，因此这种实现会有一点小问题。我们可以通过在解析每个片段时将回调函数表保存在栈中固定索引的位置来解决这个问题。

另一种设计是通过引用来关联回调函数表和用户数据（见30.3.1节）：创建一个指向回调函数表的引用，并将这个引用（一个整数）保存在 `lxp_userdata` 结构体中。请实现这个方法，不要忘记在关闭解析器时释放该引用。

33

线程和状态

Lua 语言不支持真正的多线程，即不支持共享内存的抢占式线程。原因有两个，其一是 ISO C 没有提供这样的功能，因此也没有可移植的方法能在 Lua 中实现这种机制；其二，也是更重要的原因，在于我们认为在 Lua 中引入多线程不是一个好主意。

多线程一般用于底层编程。像信号量（semaphore）和监视器（monitor）这样的同步机制一般都是操作系统上下文（以及老练的程序员）提供的，而非应用程序提供。要查找和纠正多线程相关的 Bug 是很困难的，其中有些 Bug 还会导致安全隐患。此外，程序中的一些需要同步的临界区（例如内存分配函数）还可能由于同步而导致性能问题。

多线程的这些问题源于线程抢占（preemption）和共享内存，因此如果使用非抢先式的线程或者不使用共享内存就可以避免这些问题。Lua 语言同时支持这两种方案。Lua 语言的线程（也就是所谓的协程）是协作式的，因此可以避免因不可预知的线程切换而带来的问题。另一方面，Lua 状态之间不共享内存，因此也为 Lua 语言中实现并行化提供了良好基础。本章将会介绍这两种方式。

33.1 多线程

在 Lua 语言中，协程的本质就是线程（*thread*）。我们可以认为协程是带有良好编程接口的线程，也可以认为线程是带有底层 API 的协程。

从 C API 的角度来看，把线程当作一个栈会比较有用；而从实现的角度来看，栈实际上就是线程。每个栈都保存着一个线程中挂起的函数调用的信息，外加每个函数调用的参数和局部变量。换句话说，一个栈包括了一个线程得以继续运行所需的所有信息。因此，多个线程就意味着多个独立的栈。

Lua 语言中 C API 的大多数函数操作的是特定的栈，Lua 是如何知道应该使用哪个栈的呢？当调用 `lua_pushnumber` 时，是怎么指定将数字压入何处的呢？秘密在于 `lua_State` 类型，即这些函数的第一个参数，它不仅表示一个 Lua 状态，还表示带有该状态的一个线程（许多人认为这个类型应该叫作 `lua_Thread`，也许他们是对的）。

当创建一个 Lua 状态时，Lua 就会自动用这个状态创建一个主线程，并返回代表该线程的 `lua_State`。这个主线程永远不会被垃圾回收，它只会在调用 `lua_close` 关闭状态时随着状态一起释放。与线程无关的程序会在这个主线程中运行所有的代码。

调用 `lua_newthread` 可以在一个状态中创建其他的线程：

```
lua_State *lua_newthread (lua_State *L);
```

该函数会将新线程作为一个 "thread" 类型的值压入栈中，并返回一个表示该新线程的 `lua_State` 类型的指针。例如，考虑如下的语句：

```
L1 = lua_newthread(L);
```

执行上述代码后，我们就有了两个线程 `L1` 和 `L`，它们都在内部引用了相同的 Lua 状态。每个线程都有其自己的栈。新线程 `L1` 从空栈开始运行，而老线程 `L` 在其栈顶会引用这个新线程：

```
printf("%d\n", lua_gettop(L1));           --> 0
printf("%s\n", luaL_typename(L, -1));      --> thread
```

除主线程以外，线程和其他的 Lua 对象一样都是垃圾回收的对象。当新建一个线程时，新创建的线程会被压入栈中，这样就保证了新线程不会被垃圾收集。永远不要使用未被正确锚定在 Lua 状态中的线程（主线程是内部锚定的，因此无须担心这一点）。所有对 Lua API 的调用都有可能回收未锚定的线程，即使是正在使用这个线程的函数调用。例如，考虑如下的代码：

```
lua_State *L1 = lua_newthread (L);
lua_pop(L, 1);           /* L1现在是垃圾 */
lua_pushstring(L1, "hello");
```

调用 `lua_pushstring` 可能会触发垃圾收集器并回收 `L1`, 从而导致应用崩溃, 尽管 `L1` 正在被使用。要避免这种情况, 应该在诸如一个已锚定线程的栈、注册表或 Lua 变量中保留一个对使用中线程的引用。

一旦拥有一个新线程, 我们就可以像使用主线程一样来使用它了。我们可以将元素压入栈中, 或者从栈中弹出元素, 还可以用它来调用函数等等。例如, 如下代码在新线程中调用了 `f(5)`, 然后将结果传递到老线程中:

```
lua_getglobal(L1, "f"); /* 假设 'f' 是一个全局函数 */
lua_pushinteger(L1, 5);
lua_call(L1, 1, 1);
lua_xmove(L1, L, 1);
```

函数 `lua_xmove` 可以在同一个 Lua 状态的两个栈之间移动 Lua 值。一个形如 `lua_xmove(F, T, n)` 的调用会从栈 `F` 中弹出 `n` 个元素, 并将它们压入栈 `T` 中。

不过, 对于这类用法, 我们不需要用新线程, 用主线程就足够了。使用多线程的主要目的是实现协程, 从而可以挂起某些协程的执行, 并在之后恢复执行。因此, 我们需要用到函数 `lua_resume`:

```
int lua_resume (lua_State *L, lua_State *from, int narg);
```

要启动一个协程, 我们可以像使用 `lua_pcall` 一样使用 `lua_resume`: 将待调用函数 (协程体) 压入栈, 然后压入协程的参数, 并以参数的数量作为参数 `narg` 调用 `lua_resume` (参数 `from` 是正在执行调用的线程, 或为 `NULL`)。这个行为与 `lua_pcall` 类似, 但有三个不同点。首先, `lua_resume` 中没有表示期望结果数量的参数, 它总是返回被调用函数的所有结果。其次, 它没有表示错误处理函数的参数, 发生错误时不会进行栈展开, 这样我们就可以在错误发生后检查栈的情况。最后, 如果正在运行的函数被挂起, `lua_resume` 就会返回代码 `LUA_YIELD`, 并将线程置于一个可以后续再恢复执行的状态中。

当 `lua_resume` 返回 `LUA_YIELD` 时, 线程栈中的可见部分只包含传递给 `yield` 的值。调用 `lua_gettop` 会返回这些值的个数。如果要将这些值转移到另一个线程, 可以使用 `lua_xmove`。

要恢复一个挂起的线程, 可以再次调用 `lua_resume`。在这种调用中, Lua 假设栈中所有的值都会被调用的 `yield` 返回。例如, 如果在一个 `lua_resume` 返回后到再次调用 `lua_resume` 时不改变线程的栈, 那么 `yield` 会原样返回它产生的值。

通常, 我们会把一个 Lua 函数作为协程体启动协程。这个 Lua 函数可以调用其他 Lua 函数, 并且其中任意一个函数都可以挂起, 从而结束对 `lua_resume` 的调用。例如, 假设有如下定义:

```

function foo (x) coroutine.yield(10, x) end
function foo1 (x) foo(x + 1); return 3 end

```

现在运行以下 C 语言代码：

```

lua_State *L1 = lua_newthread(L);
lua_getglobal(L1, "foo1");
lua_pushinteger(L1, 20);
lua_resume(L1, L, 1);

```

调用 `lua_resume` 会返回 `LUA_YIELD`, 表示线程已交出了控制权。此时, `L1` 的栈便有了为 `yield` 指定的值：

```

printf("%d\n", lua_gettop(L1));      --> 2
printf("%lld\n", lua_tointeger(L1, 1));    --> 10
printf("%lld\n", lua_tointeger(L1, 2));    --> 21

```

当恢复此线程时, 它会从挂起的地方 (即调用 `yield` 的地方) 继续执行。此时, `foo` 会返回到 `foo1`, `foo1` 继而又返回到 `lua_resume`:

```

lua_resume(L1, L, 0);
printf("%d\n", lua_gettop(L1));      --> 1
printf("%lld\n", lua_tointeger(L1, 1));    --> 3

```

第二次调用 `lua_resume` 时会返回 `LUA_OK`, 表示一个正常的返回。

一个协程也可以调用 C 语言函数, 而 C 语言函数又可以反过来调用其他 Lua 函数。我们已经讨论过如何使用延续 (continuation) 来让这些 Lua 函数交出控制权 (参见 29.2 节)。C 语言函数也可以交出控制权。在这种情况下, 它必须提供一个在线程恢复时被调用的延续函数 (continuation function)。要交出控制权, C 语言函数必须调用如下的函数：

```

int lua_yieldk (lua_State *L, int nresults, int ctx,
                lua_CFunction k);

```

在返回语句中我们应该始终使用这个函数, 例如:

```

static int myCfunction (lua_State *L) {
    ...
    return lua_yieldk(L, nresults, ctx, k);
}

```

这个调用会立即挂起正在运行的协程。参数 `nresults` 是将要返回给对应的 `lua_resume` 的栈中值的个数；参数 `ctx` 是传递给延续的上下文信息；参数 `k` 是延续函数。当协程恢复运行时，控制权会直接交给延续函数 `k`；当协程交出控制权后，`myCfunction` 就不会再有其他任何动作，它必须将所有后续的工作委托给延续函数处理。

让我们来看一个典型的例子。假设要编写一个读取数据的函数，如果无数据可读则交出控制权。我们可能会用 C 语言写出一个这样的函数：^①

```
int readK (lua_State *L, int status, lua_KContext ctx) {
    (void)status; (void)ctx; /* 未使用的参数 */
    if (something_to_read()) {
        lua_pushstring(L, read_some_data());
        return 1;
    }
    else
        return lua_yieldk(L, 0, 0, &readK);
}

int prim_read (lua_State *L) {
    return readK(L, 0, 0);
}
```

在这个示例中，`prim_read` 无须做任何初始化，因此它可以直接调用延续函数（`readK`）。如果有数据可读，`readK` 会读取并返回数据；否则，它会交出控制权。当线程恢复时，`prim_read` 会再次调用延续函数，该延续函数会再次尝试读取数据。

如果 C 语言函数在交出控制权之后什么都不做，那么它可以不带延续函数调用 `lua_yieldk` 或者使用宏 `lua_yield`：

```
return lua_yield(L, nres);
```

在这一句调用之后，当线程恢复时，控制权会返回到名为 `myCfunction` 的函数中。

^①正如笔者之前提到过的，在 Lua 5.3 之前，延续的 API 有一点不同。特别是，延续函数只有一个参数，即 Lua 状态。

33.2 Lua 状态

每次调用 `luaL_newstate`（或 `lua_newstate`）都会创建一个新的 Lua 状态。不同的 Lua 状态之间是完全独立的，它们根本不共享数据。也就是说，无论在一个 Lua 状态中发生了什么，都不会影响其他 Lua 状态。这也意味着 Lua 状态之间不能直接通信，因而必须借助一些 C 语言代码的帮助。例如，给定两个状态 L1 和 L2，如下命令会将 L1 栈顶的字符串压入 L2 的栈中：

```
lua_pushstring(L2, lua_tostring(L1, -1));
```

由于所有数据必须由 C 语言进行传递，因此 Lua 状态之间只能交换能够使用 C 语言表示的类型，例如字符串和数值。其他诸如表之类的类型必须序列化后才能传递。

在支持多线程的系统中，一种有趣的设计是为每个线程创建一个独立的 Lua 状态。这种设计使得线程类似于 POSIX 进程，它实现了非共享内存的并发（concurrency）。在本节中，我们会根据这种方法开发一个多线程的原型实现。在这个实现中，将会使用 POSIX 线程（`pthread`）。因为这些代码只使用了一些基础功能，所以将它们移植到其他线程系统中并不难。

我们要开发的系统很简单，其主要目的是演示在一个多线程环境中使用多个 Lua 状态。在这个系统开始运行之后，我们可以为它添加几个高级功能。我们把这个库称为 `lproc`，它只提供 4 个函数：

`lproc.start(chunk)`

启动一个新进程来运行指定的代码段（一个字符串）。这个库将 Lua 进程（`process`）实现为一个 C 语言线程（`thread`）外加与其相关联的 Lua 状态。

`lproc.send(channel, val1, val2, ...)`

将所有指定值（应为字符串）发送给指定的、由名称（也是一个字符串）标识的通道（`channel`）。后面有一个练习，该练习要求对上述函数进行修改，使其支持发送其他类型的数据。

`lproc.receive(channel)`

接收发送给指定通道的值。

`lproc.exit()`

结束一个进程。只有主进程需要这个函数。如果主程序不调用 `lproc.exit` 就直接结束，那么整个程序会终止，而不会等待其他进程结束。

这个库通过字符串标识不同的通道，并通过字符串来匹配发送者和接收者。一个发送操作可以发送任意数量的字符串，这些字符串由对应的接收操作返回。所有的通信都是同步的，向通道发送消息的进程会一直阻塞，直到有进程从该通道接收信息，而从通道接收信息的进程会一直阻塞，直至有进程向其发送消息。

`lproc` 的实现像其接口一样简单，它使用了两个循环双向链表（circular double-linked list），一个用于等待发送消息的进程，另一个用于等待接收消息的进程。`lproc` 使用一个互斥量（mutex）来控制对这两个链表的访问。每个进程有一个关联的条件变量（condition variable）。当进程要向通道发送一条消息时，它会遍历接收链表以查找一个在该通道上等待的进程。如果找到了这样的进程，它会将该进程从等待链表中删除，并将消息的值从自身转移到找到的进程中，然后通知其他进程；否则，它就将自己插入发送链表，然后等待其条件变量发生变化。接收消息的操作也与此基本类似。

在这种实现中，主要的元素之一就是表示进程的结构体：

```
#include <pthread.h>
#include "lua.h"
#include "lauxlib.h"

typedef struct Proc {
    lua_State *L;
    pthread_t thread;
    pthread_cond_t cond;
    const char *channel;
    struct Proc *previous, *next;
} Proc;
```

前两个字段表示进程使用的 Lua 状态和运行该进程的 C 线程。第三个字段 `cond` 是条件变量，线程会在等待匹配的发送/接收时用它来使自己进入阻塞状态。第四个字段保存了进程正在等待的通道（如果有的话）。最后两个字段 `previous` 和 `next` 将进程的结构体组成等待链表。

下面的代码声明了两个等待链表及关联的互斥量：

```
static Proc *waitsend = NULL;
static Proc *waitreceive = NULL;

static pthread_mutex_t kernel_access = PTHREAD_MUTEX_INITIALIZER;
```

每个进程都需要一个 Proc 结构体，并且进程脚本调用 send 或 receive 时就需要访问这个结构体。这些函数接收的唯一参数就是进程的 Lua 状态；因此，每个进程都应将其 Proc 结构体保存在其 Lua 状态中。在我们的实现中，每个状态都将其对应的 Proc 结构体作为完整的用户数据存储在注册表中，关联的键为 "_SELF"。辅助函数 getself 可以从指定的状态中获取相关联的 Proc 结构体：

```
static Proc *getself (lua_State *L) {
    Proc *p;
    lua_getfield(L, LUA_REGISTRYINDEX, "_SELF");
    p = (Proc *)lua_touserdata(L, -1);
    lua_pop(L, 1);
    return p;
}
```

下一个函数，movevalues，将值从发送进程移动到接收进程：

```
static void movevalues (lua_State *send, lua_State *rec) {
    int n = lua_gettop(send);
    int i;
    luaL_checkstack(rec, n, "too many results");
    for (i = 2; i <= n; i++) /* 将值传给接收进程 */
        lua_pushstring(rec, lua_tostring(send, i));
}
```

这个函数将发送进程的栈中所有的值（除了第一个，它是通道）移动到接收进程的栈中。请注意，在压入任意数量的元素时，需要检查栈空间。

示例 33.1 定义了函数 searchmatch，该函数会遍历列表以寻找等待指定通道的进程。

示例 33.1 用于寻找等待通道的进程的函数

```
static Proc *searchmatch (const char *channel, Proc **list) {
    Proc *node;
    /* 遍历列表 */
    for (node = *list; node != NULL; node = node->next) {
        if (strcmp(channel, node->channel) == 0) { /* 匹配? */
            /* 将结点从列表移除 */
            if (*list == node) /* 结点是否为第一个元素? */

```

Lua 程序设计（第 4 版）

```

        *list = (node->next == node) ? NULL : node->next;
        node->previous->next = node->next;
        node->next->previous = node->previous;
        return node;
    }
}

return NULL; /* 没有找到匹配 */
}

```

如果找到一个进程，那么该函数会将这个进程从列表中移除并返回该进程；否则，该函数会返回 NULL。

当找不到匹配的进程时，会调用最后的辅助函数，参见示例 33.2。

示例 33.2 用于在等待列表中新增一个进程的函数

```

static void waitonlist (lua_State *L, const char *channel,
                      Proc **list) {
    Proc *p = getself(L);

    /* 将其自身放到链表的末尾 */
    if (*list == NULL) { /* 链表为空? */
        *list = p;
        p->previous = p->next = p;
    } else {
        p->previous = (*list)->previous;
        p->next = *list;
        p->previous->next = p->next->previous = p;
    }

    p->channel = channel; /* 等待的通道 */

    do { /* 等待其条件变量 */
        pthread_cond_wait(&p->cond, &kernel_access);
    } while (p->channel);
}

```

在这种情况下，进程会将自己链接到相应等待链表的末尾，然后进入等待状态，直到另一个进程与之匹配并将其唤醒（`pthread_cond_wait` 附近的循环会处理 POSIX 线程允许的虚假唤醒，spurious wakeup）。当一个进程唤醒另一个进程时，它会将另一个进程的 `channel` 字段设置为 `NULL`。因此，如果 `p->channel` 不是 `NULL`，那就表示尚未出现与进程 `p` 匹配的进程，所以需要继续等待。

有了这些辅助函数，我们就可以编写 `send` 和 `receive` 了（参见示例 33.3）。

示例 33.3 用于发送和接收消息的函数

```
static int ll_send (lua_State *L) {
    Proc *p;
    const char *channel = luaL_checkstring(L, 1);

    pthread_mutex_lock(&kernel_access);

    p = searchmatch(channel, &waitreceive);
    if (p) { /* 找到匹配的接收线程？ */
        movevalues(L, p->L); /* 将值传递给接收线程 */
        p->channel = NULL; /* 标记接收线程无须再等待 */
        pthread_cond_signal(&p->cond); /* 唤醒接收线程 */
    }
    else
        waitonlist(L, channel, &waitsend);

    pthread_mutex_unlock(&kernel_access);
    return 0;
}

static int ll_receive (lua_State *L) {
    Proc *p;
    const char *channel = luaL_checkstring(L, 1);
    lua_settop(L, 1);

    pthread_mutex_lock(&kernel_access);
```

Lua 程序设计（第 4 版）

```

    p = searchmatch(channel, &waitsend);

    if (p) { /* 找到匹配的发送线程? */
        movevalues(p->L, L); /* 从发送线程获取值 */
        p->channel = NULL; /* 标记发送线程无须再等待 */
        pthread_cond_signal(&p->cond); /* 唤醒发送线程 */
    }
    else
        waitonlist(L, channel, &waitreceive);

    pthread_mutex_unlock(&kernel_access);

    /* 返回除通道外的栈中的值 */
    return lua_gettop(L) - 1;
}

```

函数 `ll_send` 先获取通道，然后锁住互斥量并搜索匹配的接收进程。如果找到了，就把待发送的值传递给这个接收进程，然后将接收进程标记为就绪状态并唤醒接收进程。否则，发送进程就将自己放入等待链表。当操作完成后，`ll_send` 解锁互斥量且不向 Lua 返回任何值。函数 `ll_receive` 与之类似，但它会返回所有接收到的值。

现在，让我们看一下如何创建新进程。新进程需要一个新的 POSIX 线程，而 POSIX 线程的运行需要一个线程体。我们会在后面的内容中定义这个线程体。在此，先看一下它的原型，这是 `pthread` 所要求的：

```
static void *ll_thread (void *arg);
```

要创建并运行一个新进程，我们开发的系统必须创建一个新的 Lua 状态，启动一个新线程，编译指定的代码段，调用该代码段，最后释放其资源。原线程会完成前三个任务，而新线程则负责其余任务（为了简化错误处理，我们的系统只在成功编译了指定的代码段后才启动新的线程）。

函数 `ll_start` 可以创建一个新的进程（见示例 33.4）。

示例 33.4 用于创建进程的函数

```

static int ll_start (lua_State *L) {
    pthread_t thread;

```

```

const char *chunk = luaL_checkstring(L, 1);
lua_State *L1 = luaL_newstate();

if (L1 == NULL)
    luaL_error(L, "unable to create new state");

if (luaL_loadstring(L1, chunk) != 0)
    luaL_error(L, "error in thread body: %s",
               lua_tostring(L1, -1));

if (pthread_create(&thread, NULL, ll_thread, L1) != 0)
    luaL_error(L, "unable to create new thread");

pthread_detach(thread);
return 0;
}

```

该函数创建了一个新的 Lua 状态 L1，并在其中编译了指定的代码段。如果有错误发生，该函数会把错误传递给原来的状态 L。然后，该函数使用 ll_thread 作为线程体创建一个新线程（使用 pthread_create 创建），同时将新状态 L1 作为参数传递给这个线程体。最后，该函数调用 pthread_detach 通知系统我们不需要该线程的任何运行结果。

每个新线程的线程体都是函数 ll_thread（见示例 33.5），它接收相应的 Lua 状态（由 ll_start 创建），这个 Lua 状态的栈中只含有预编译的主代码段。

示例 33.5 新线程的线程体

```

int luaopen_lproc (lua_State *L); /* $-size-arg-size */

static void *ll_thread (void *arg) {
    lua_State *L = (lua_State *)arg;
    Proc *self; /* 进程自身的控制块 */

    openlibs(L); /* 打开标准库 */
    luaL_requiref(L, "lproc", luaopen_lproc, 1);
    lua_pop(L, 1); /* 移除之前调用的结果 */
}

```

```

    self = (Proc *)lua_newuserdata(L, sizeof(Proc));
    lua_setfield(L, LUA_REGISTRYINDEX, "_SELF");
    self->L = L;
    self->thread = pthread_self();
    self->channel = NULL;
    pthread_cond_init(&self->cond, NULL);

    if (lua_pcall(L, 0, 0, 0) != 0) /* 调用主代码段 */
        fprintf(stderr, "thread error: %s", lua_tostring(L, -1));

    pthread_cond_destroy(&getself(L)->cond);
    lua_close(L);
    return NULL;
}

```

首先，该函数打开 Lua 标准库和库 lproc；之后，它创建并初始化其自身的控制块^①；然后，调用主代码段；最后，销毁其条件变量并关闭 Lua 状态。

请注意使用 luaL_requiref 打开库 lproc 的用法。^②这个函数在某种意义上等价于 require，但它用指定函数（示例 33.5 中的 luaopen_lproc）来打开库而没有搜索打开函数（loader）。在调用这个打开函数后，luaL_requiref 会在表 package.loaded 中注册结果，这样以后再调用 require 加载这个库时就无须再次打开库了。当 luaL_requiref 的最后一个参数为真时，该函数还会在相应的全局变量（示例 33.5 中为 lproc）中注册这个库。

示例 33.6 演示了这个模块中的最后一个函数。

示例 33.6 模块 lproc 的其他函数

```

static int ll_exit (lua_State *L) {
    pthread_exit(NULL);
    return 0;
}

static const struct luaL_Reg ll_funcs[] = {
    {"start", ll_start},
}

```

^①译者注：在操作系统领域经常将封装了进程的结构体称为控制块。

^②这个函数是在 Lua 5.2 中引入的。

```

    {"send", ll_send},
    {"receive", ll_receive},
    {"exit", ll_exit},
    {NULL, NULL}
};

int luaopen_lproc (lua_State *L) {
    luaL_newlib(L, ll_funcs); /* open library */
    return 1;
}

```

这两个函数都很简单。函数 `ll_exit` 应该只能在主进程结束时由主进程调用，以避免整个程序立即结束。函数 `luaopen_lproc` 是用于打开这个模块的标准函数。

正如笔者之前说过的，在 Lua 语言中这种进程的实现方式非常简单。我们可以对它进行各种改进，这里简单介绍几种。

第一种显而易见的改进是改变对匹配通道的线性查找，更好的选择是用哈希表来寻找通道，并为每个通道设置一个独立的等待列表。

另一种改进涉及创建进程的效率。创建一个新的 Lua 状态是一个轻量级操作，但打开所有的标准库可不是轻量级的，并且大部分进程可能并不需要用到所有的标准库。我们可以通过对库进行预注册来避免打开无用的库，这一点已经在 17.1 节中讨论过。相对于为每个标准库调用 `luaL_requiref`，使用这种方法时我们只需将库的打开函数放入表 `package.preload` 中即可。当且仅当进程调用 `require "lib"` 时，`require` 才会调用这个与库相关的函数来打开库。示例 33.7 中的函数 `registerlib` 会完成这样的注册。

示例 33.7 注册按需打开的库

```

static void registerlib (lua_State *L, const char *name,
                        lua_CFunction f) {
    lua_getglobal(L, "package");
    lua_getfield(L, -1, "preload"); /* 获取'package.preload' */
    lua_pushcfunction(L, f);
    lua_setfield(L, -2, name); /* package.preload[name] = f */
    lua_pop(L, 2); /* 弹出'package'和'preload' */
}

```

```

static void openlibs (lua_State *L) {
    luaL_requiref(L, "_G", luaopen_base, 1);
    luaL_requiref(L, "package", luaopen_package, 1);
    lua_pop(L, 2); /* 移除之前调用的结果 */
    registerlib(L, "coroutine", luaopen_coroutine);
    registerlib(L, "table", luaopen_table);
    registerlib(L, "io", luaopen_io);
    registerlib(L, "os", luaopen_os);
    registerlib(L, "string", luaopen_string);
    registerlib(L, "math", luaopen_math);
    registerlib(L, "utf8", luaopen_utf8);
    registerlib(L, "debug", luaopen_debug);
}

```

一般情况都需要打开基础库。另外，我们还需要 package 库；如果没有 package 库，就无法通过 require 来打开其他库。所有其他的库都是可选的。因此，除了调用 luaL_openlibs 之外，可以在打开新状态时调用我们自己的函数 openlibs（在示例 33.7 中也有展示）。当进程需要用到其中任意一个库时，只需显式地调用 require，require 就会调用相应的 luaopen_* 函数。

另一个改进涉及通信原语（communication primitive）。例如，为 lproc.send 和 lproc.receive 设置一个等待匹配的时间阈值会非常有用。特别的，当等待时间阈值为零时，这两个函数会成为非阻塞的。在 POSIX 线程中，可以用 pthread_cond_timedwait 实现这个功能。

33.3 练习

练习 33.1：正如我们所见，如果函数调用 lua_yield（没有延续的版本），当线程唤醒时，控制权会返回给调用它的函数。请问调用函数会接收到什么样的值作为这次调用的返回结果？

练习 33.2：修改库 lproc，使得这个库能够发送和接收其他诸如布尔值和数值的类型时无须将其转换成字符串（提示：只需要修改函数 movevalues 即可）。

练习 33.3：修改库 lproc，使得这个库能够发送和接收表（提示：可以通过遍历原表在接收状态中创建一个副本）。

练习 33.4：在库 lproc 中实现无阻塞的 send 操作。

梅隆魁，2013 年硕士毕业于北京邮电大学计算机科学与技术专业嵌入式系统与网络通信方向。毕业后就职于中国民生银行总行信息科技部，主要从事 J2EE 企业级及分布式嵌入式系统的应用和架构设计开发及项目管理工作，业余对嵌入式软硬件、移动应用开发及 Android 移动安全也有所涉猎，是一名“会画画”的工程师。

译者简介

如果您对翻译经典外版图书感兴趣，欢迎和我们联系。
联系邮箱：fujm@pheu.com.cn

译者招募

