15

Modules and Packages

Usually, Lua does not set policies. Instead, Lua provides mechanisms that are powerful enough for groups of developers to implement the policies that best suit them. However, this approach does not work well for modules. One of the main goals of a module system is to allow different groups to share code. The lack of a common policy impedes this sharing.

Starting in version 5.1, Lua defines a set of policies for modules and packages (a package being a collection of modules). These policies do not demand any extra facility from the language; programmers can implement them using what we have seen so far: tables, functions, metatables, and environments. However, two important functions ease the adoption of these policies: `require`, for using modules, and `module`, for building modules. Programmers are free to re-implement these functions with different policies. Of course, alternative implementations may lead to programs that cannot use foreign modules and modules that cannot be used by foreign programs.

From the user point of view, a *module* is a library that can be loaded through `require` and that defines one single global name containing a table. Everything that the module exports, such as functions and constants, it defines inside this table, which works as a namespace. A well-behaved module also arranges for `require` to return this table.

An obvious benefit of using tables to implement modules is that we can manipulate modules like any other table and use the whole power of Lua to create extra facilities. In most languages, modules are not first-class values (that is, they cannot be stored in variables, passed as arguments to functions, etc.), so those languages need special mechanisms for each extra facility they want to offer for modules. In Lua, you get extra facilities for free.

For instance, there are several ways for a user to call a function from a module. The simplest is this:

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

If she prefers a shorter name for the module, she can set a local name for it:

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

She can also provide alternative names for individual functions:

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

The nice thing about these facilities is that they involve no explicit support from the language. They use what the language already offers.

15.1 The `require` Function

Lua offers a high-level function to load modules, called `require`. This function tries to keep to a minimum its assumptions about what a module is. For `require`, a module is just any chunk of code that defines some values (such as functions or tables containing functions).

To load a module, we simply call `require"`*modname*`"`. Typically, this call returns a table comprising the module functions, and it also defines a global variable containing this table. However, these actions are done by the module, not by `require`, so some modules may choose to return other values or to have different side effects.

It is a good programming practice always to require the modules you need, even if you know that they would be already loaded. You may exclude the standard libraries from this rule, because they are pre-loaded in Lua. Nevertheless, some people prefer to use an explicit `require` even for them:

```
local m = require "io"
m.write("hello world\n")
```

Listing 15.1 details the behavior of `require`. Its first step is to check in table `package.loaded` whether the module is already loaded. If so, `require` returns its corresponding value. Therefore, once a module is loaded, other calls to `require` simply return the same value, without loading the module again.

If the module is not loaded yet, `require` tries to find a *loader* for this module. (This step is illustrated by the abstract function `findloader` in Listing 15.1.) Its first attempt is to query the given library name in table `package.preload`. If it finds a function there, it uses this function as the module loader. This `preload` table provides a generic method to handle some non-conventional situations (e.g., C libraries statically linked to Lua). Usually, this table does not have an entry for the module, so `require` will search first for a Lua file and then for a C library to load the module from.

If `require` finds a Lua file for the given module, it loads it with `loadfile`; otherwise, if it finds a C library, it loads it with `loadlib`. Remember that both

Listing 15.1. The require function:

```
function require (name)
  if not package.loaded[name] then    -- module not loaded yet?
    local loader = findloader(name)
    if loader == nil then
      error("unable to load module " .. name)
    end
    package.loaded[name] = true         -- mark module as loaded
    local res = loader(name)            -- initialize module
    if res ~= nil then
      package.loaded[name] = res
    end
  end
  return package.loaded[name]
end
```

loadfile and loadlib only load some code, without running it. To run the code, require calls it with a single argument, the module name. If the loader returns any value, require returns this value and stores it in table package.loaded to return the same value in future calls for this same library. If the loader returns no value, require returns whatever value is in table package.loaded. As we will see later in this chapter, a module can put the value to be returned by require directly into package.loaded.

An important detail of that previous code is that, before calling the loader, require marks the module as already loaded, assigning **true** to the respective field in package.loaded. Therefore, if the module requires another module and that in turn recursively requires the original module, this last call to require returns immediately, avoiding an infinite loop.

To force require into loading the same library twice, we simply erase the library entry from package.loaded. For instance, after a successful require"foo", package.loaded["foo"] will not be **nil**. The following code will load the library again:

```
package.loaded["foo"] = nil
require "foo"
```

When searching for a file, require uses a path that is a little different from typical paths. The path used by most programs is a list of directories wherein to search for a given file. However, ANSI C (the abstract platform where Lua runs) does not have the concept of directories. Therefore, the path used by require is a list of *patterns*, each of them specifying an alternative way to transform a module name (the argument to require) into a file name. More specifically, each component in the path is a file name containing optional question marks. For each component, require replaces the module name for each '?' and checks whether there is a file with the resulting name; if not, it goes to the next

component. The components in a path are separated by semicolons (a character seldom used for file names in most operating systems). For instance, if the path is

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

then the call `require "sql"` will try to open the following files:

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

The `require` function assumes only the semicolon (as the component separator) and the question mark; everything else, such as directory separators or file extensions, is defined by the path itself.

The path that `require` uses to search for Lua files is always the current value of the variable `package.path`. When Lua starts, it initializes this variable with the value of the environment variable `LUA_PATH` or with a compiled-defined default path, if this environment variable is not defined. When using `LUA_PATH`, Lua substitutes the default path for any substring "`;;`". For instance, if you set `LUA_PATH` to "`mydir/?.lua;;`", the final path will be the component "`mydir/?.lua`" followed by the default path.

If `require` cannot find a Lua file compatible with the module name, it looks for a C library. For this search, it gets the path from variable `package.cpath` (instead of `package.path`). This variable gets its initial value from the environment variable `LUA_CPATH` (instead of `LUA_PATH`). A typical value for this variable in Unix is like this:

```
./?.so;/usr/local/lib/lua/5.1/?.so
```

Note that the file extension is defined by the path (e.g., the previous example uses `.so` for all templates). In Windows, a typical path is more like this one:

```
.\?.dll;C:\Program Files\Lua501\dll\?.dll
```

Once it finds a C library, `require` loads it with `package.loadlib`, which we discussed in Section 8.2. Unlike Lua chunks, C libraries do not define one single main function. Instead, they can export several C functions. Well-behaved C libraries should export one function called `luaopen_`*modname*, which is the function that `require` tries to call after linking the library. In Section 26.2 we will discuss how to write C libraries.

Usually, we use modules with their original names, but sometimes we must rename a module to avoid name clashes. A typical situation is when we need to load different versions of the same module, for instance for testing. For a Lua module, either it does not have its name fixed internally (as we will see later) or we can easily edit it to change its name. But we cannot edit a binary module to correct the name of its `luaopen_*` function. To allow for such renamings, `require` uses a small trick: if the module name contains a hyphen, `require`

strips from the name its prefix up to the hyphen when creating the `luaopen_*` function name. For instance, if a module is named `a-b`, `require` expects its open function to be named `luaopen_b`, instead of `luaopen_a-b` (which would not be a valid C name anyway). So, if we need to use two modules named `mod`, we can rename one of them to `v1-mod` (or `-mod`, or anything like that). When we call `m1=require"v1-mod"`, `require` will find both the renamed file `v1-mod` and, inside this file, the function with the original name `luaopen_mod`.

15.2 The Basic Approach for Writing Modules

The simplest way to create a module in Lua is really simple: we create a table, put all functions we want to export inside it, and return this table. Listing 15.2 illustrates this approach. Note how we define `inv` as a private name simply by declaring it local to the chunk.

The use of tables for modules does not provide exactly the same functionality as provided by real modules. First, we must explicitly put the module name in every function definition. Second, a function that calls another function inside the same module must qualify the name of the called function. We can ameliorate these problems using a fixed local name for the module (`M`, for instance), and then assigning this local to the final name of the module. Following this guideline, we would write our previous module like this:

```
local M = {}
complex = M           -- module name

M.i = {r=0, i=1}
function M.new (r, i) return {r=r, i=i} end

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

   <as before>
```

Whenever a function calls another function inside the same module (or whenever it calls itself recursively), it still needs to prefix the name. At least, the connection between the two functions does not depend on the module name anymore. Moreover, there is only one place in the whole module where we write the module name. Actually, we can avoid writing the module name altogether, because `require` passes it as an argument to the module:

```
local modname = ...
local M = {}
_G[modname] = M

M.i = {r=0, i=1}
  <as before>
```

Listing 15.2. A simple module:

```
complex = {}

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

-- defines a constant 'i'
complex.i = complex.new(0, 1)

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

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

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

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

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

return complex
```

With this change, all we have to do to rename a module is to rename the file that defines it.

Another small improvement relates to the closing return statement. It would be nice if we could concentrate all module-related setup tasks at the beginning of the module. One way of eliminating the need for the return statement is to assign the module table directly into `package.loaded`:

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
  <as before>
```

With this assignment, we do not need to return `M` at the end of the module: remember that, if a module does not return a value, `require` returns the current

value of `package.loaded[modname]`.

15.3 Using Environments

A major drawback of that basic method for creating modules is that it calls for special attention from the programmer. She must qualify names when accessing other public entities inside the same module. She has to change the calls whenever she changes the status of a function from private to public (or from public to private). Moreover, it is all too easy to forget a **local** in a private declaration.

Function environments offer an interesting technique for creating modules that solves all these problems. Once the module main chunk has an exclusive environment, not only all its functions share this table, but also all its global variables go to this table. Therefore, we can declare all public functions as global variables and they will go to a separate table automatically. All the module has to do is to assign this table to the module name and also to `package.loaded`. The next code fragment illustrates this technique:

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
setfenv(1, M)
```

Now, when we declare function `add`, it goes to `complex.add`:

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

Moreover, we can call other functions from the same module without any prefix. For instance, `add` gets `new` from its environment, that is, it gets `complex.new`.

This method offers a good support for modules, with little extra work for the programmer. It needs no prefixes at all. There is no difference between calling an exported and a private function. If the programmer forgets a **local**, he does not pollute the global namespace; instead, a private function simply becomes public.

What is missing, of course, is access to other modules. Once we make the empty table `M` our environment, we lose access to all previous global variables. There are several ways to recover this access, each with its pros and cons.

The simplest solution is inheritance, as we saw earlier:

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
setmetatable(M, {__index = _G})
setfenv(1, M)
```

(You must call `setmetatable` before calling `setfenv`; can you tell why?) With this construction, the module has direct access to any global identifier, paying a small overhead for each access. A funny consequence of this solution is that, conceptually, your module now contains all global variables. For instance, someone using your module may call the standard sine function by writing `complex.math.sin(x)`. (Perl's package system has this peculiarity, too.)

Another quick method of accessing other modules is to declare a local that holds the old environment:

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
local _G = _G
setfenv(1, M)
```

Now you must prefix any global-variable name with `_G.`, but the access is a little faster, because there is no metamethod involved.

A more disciplined approach is to declare as locals only the functions you need, or at most the modules you need:

```
-- module setup
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M

-- Import Section:
-- declare everything this module needs from outside
local sqrt = math.sqrt
local io = io

-- no more external access after this point
setfenv(1, M)
```

This technique demands more work, but it documents your module dependencies better. It also results in code that runs faster than code with the previous schemes.

15.4 The `module` Function

Probably you noticed the repetitions of code in our previous examples. All of them started with this same pattern:

```
local modname = ...
local M = {}
_G[modname] = M
package.loaded[modname] = M
  <setup for external access>
setfenv(1, M)
```

Lua 5.1 provides a new function, called `module`, that packs this functionality. Instead of this previous setup code, we can start a module simply like this:

```
module(...)
```

This call creates a new table, assigns it to the appropriate global variable and to the `loaded` table, and then sets the table as the environment of the main chunk.

By default, `module` does not provide external access: before calling it, you must declare appropriate local variables with the external functions or modules you want to access. You can also use inheritance for external access adding the option `package.seeall` to the call to `module`. This option does the equivalent of the following code:

```
setmetatable(M, {__index = _G})
```

Therefore, simply adding the statement

```
module(..., package.seeall)
```

in the beginning of a file turns it into a module; you can write everything else like regular Lua code. You need to qualify neither module names nor external names. You do not need to write the module name (actually, you do not even need to know the module name). You do not need to worry about returning the module table. All you have to do is to add that single statement.

The `module` function provides some extra facilities. Most modules do not need these facilities, but some distributions need some special treatment (e.g., to create a module that contains both C functions and Lua functions). Before creating the module table, `module` checks whether `package.loaded` already contains a table for this module, or whether a variable with the given name already exists. If it finds a table in one of these places, `module` reuses this table for the module; this means we can use `module` for reopening a module already created. If the module does not exist yet, then `module` creates the module table. After that, it populates the table with some predefined variables: `_M` contains the module table itself (it is an equivalent of `_G`); `_NAME` contains the module name (the first argument passed to `module`); and `_PACKAGE` contains the package name (the name without the last component; see next section).

15.5 Submodules and Packages

Lua allows module names to be hierarchical, using a dot to separate name levels. For instance, a module named `mod.sub` is a *submodule* of `mod`. Accordingly, you may assume that module `mod.sub` will define all its values inside a table `mod.sub`, that is, inside a table stored with key `sub` in table `mod`. A *package* is a complete tree of modules; it is the unit of distribution in Lua.

When you require a module called `mod.sub`, `require` queries first the table `package.loaded` and then the table `package.preload` using the original module name "`mod.sub`" as the key; the dot has no significance whatsoever in this search.

However, when searching for a file that defines that submodule, `require` translates the dot into another character, usually the system's directory separator (e.g., '/' for Unix or '\' for Windows). After the translation, `require` searches for the resulting name like any other name. For instance, assuming the path

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

and '/' as the directory separator, the call `require"a.b"` will try to open the following files:

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

This behavior allows all modules of a package to live in a single directory. For instance, if a package has modules `p`, `p.a`, and `p.b`, their respective files can be named `p/init.lua`, `p/a.lua`, and `p/b.lua`, with the directory `p` within some appropriate directory.

The directory separator used by Lua is configured at compile time and can be any string (remember, Lua knows nothing about directories). For instance, systems without hierarchical directories can use a '_' as the "directory" separator, so that `require"a.b"` will search for a file `a_b.lua`.

C-function names cannot contain dots, so a C library for submodule `a.b` cannot export a function `luaopen_a.b`. Here `require` translates the dot into another character, an underscore. So, a C library named `a.b` should name its initialization function `luaopen_a_b`. We can use the hyphen trick here too, with some subtle results. For instance, if we have a C library `a` and we want to make it a submodule of `mod`, we can rename the file to `mod/-a`. When we write `require"mod.-a"`, `require` correctly finds the new file `mod/-a` as well as the function `luaopen_a` inside it.

As an extra facility, `require` has one more option for loading C submodules. When it cannot find either a Lua file or a C file for a submodule, it again searches the C path, but this time looking for the package name. For example, if the program requires a submodule `a.b.c`, and `require` cannot find a file when looking for `a/b/c`, this last search will look for `a`. If it finds a C library with this name, then `require` looks into this library for an appropriate open function, `luaopen_a_b_c` in this example. This facility allows a distribution to put several submodules together into a single C library, each with its own open function.

The `module` function also offers explicit support for submodules. When we create a submodule, with a call like `module("a.b.c")`, `module` puts the environment table into variable `a.b.c`, that is, into a field `c` of a table in field `b` of a table `a`. If any of these intermediate tables do not exist, `module` creates them. Otherwise, it reuses them.

From the Lua point of view, submodules in the same package have no explicit relationship other than that their environment tables may be nested. Requiring a module `a` does not automatically load any of its submodules; similarly, requiring `a.b` does not automatically load `a`. Of course, the package implementer is

free to create these links if she wants. For instance, a particular module `a` may start by explicitly requiring one or all of its submodules.