Módulos

En este artículo vamos a presentar qué son los módulos en Lua. Posteriormente veremos como definir diferentes tipos de módulos en las versiones modernas de Lua. Por último y como referencia, hablaremos de la definición de módulos en Lua para versiones más antiguas, en las que los módulos se definían de otra forma.

¿Qué son los módulos en Lua?

Los módulos, en Lua, son la forma que ofrece el lenguaje para encapsular código de forma que pueda ser reutilizado en cualquier programa. Un módulo en Lua es lo que llamamos librería en otros lenguajes. En cualquier caso la idea que reside en un módulo es la de exportar una serie de funcionalidades que puedan ser usadas en otros programas.

Lua no impone ninguna política estricta de cómo deben ser definidos los módulos, con lo que pueden ser definidos según el estilo de cada programador.

Definiendo módulos en Lua

Como comentamos anteriormente, en Lua no hay definida una política estricta de cómo debe ser definido un módulo, dejándolo al estilo del programador. Desde un punto de vista práctico vamos a ver tres casos distintos de definición de un módulo según unos usos específicos. Estos son:

- Módulo que define una interfaz a través de una tabla.
- Módulo definido como un objeto.
- Módulo definido como un singleton.

Veamos a continuación estos tres casos de uso en más detalle.

Definir un módulo con una interfaz a través de una tabla

Este caso es el más común y recomendable para aquellos módulos que están orientados a ser distribuidos como una librería. En él, definimos una tabla que contiene una referencia a todas las funciones y variables del módulo que queremos que sean públicas quedando el resto de variables y

funciones ocultas fuera del módulo. Esto se consigue declarando las funciones y variables como locales. Una vez definida la tabla, esta es devuelta. Veamos un ejemplo para entender mejor la estructura de un módulo.

Primeramente vamos a definir el módulo en si, para ello creamos un archivo llamado `modulo1.lua`. El archivo lo podemos llamar como queramos, lógicamente, pero para el tutorial usaremos ese nombre. **El nombre del archivo define el nombre del módulo**.

```
-- Definición de modulo1

-- Declaración de la tabla que contendrá las funciones del módulo
M = {}

-- Definición de funciones

-- Calcula el cuadrado de un número
local function cuadrado(x)
   return x * x
end

-- Calcula el cubo de un número
local function cubo(x)
   return x * x * x
end

-- Determina si un número es par
local function esPar(x)
   return x % 2 == 0
end

-- Se asignan las funciones cuadrado, cubo y esPar a la tabla M
-- Esta asignación constituye la interfaz del módulo.
M.cuadrado = cuadrado
M.cubo = cubo
M.esPar = esPar

-- Retorno de la tabla M
return M
```

En el módulo hemos definido tres funciones matemáticas sencillas de ámbito local. También se ha declarado una tabla, compuesta de tres elementos a los cuales se les han asignado una referencia

a cada una de las funciones. Las claves de estos tres elementos coinciden con el nombre de las funciones a las que hacen referencia.

Ahora necesitamos un segundo archivo en el que definiremos el código que se encargará de consumir el módulo. Este archivo lo crearemos en el mismo directorio donde reside el archivo del módulo.

```
-- Referencia al módulo que queremos cargar
local modulo1 = require 'modulo1'

print("Cuadrado de 5:", modulo1.cuadrado(5))
print("Cubo de 5:", modulo1.cubo(5))
print("5 es par?", modulo1.esPar(5))
```

Observa que hemos incluido una referencia al módulo `modulo1` usando la función `require`. Como puedes ver, la referencia al módulo se hace por su nombre de archivo descartando la extensión `.lua`. El resultado de ejecutar el archivo es:

```
Cuadrado de 5:  25
Cubo de 5:      125
5 es par?       false
```

Definir un módulo como un objeto

Un módulo que es definido como un objeto permite crear varias instancias del objeto, creando objetos que tienen espacios de memoria diferentes. Esto evita que la manipulación de un objeto afecte a otro. Este paradigma es el mismo que en la programación orientada a objetos. Veamos a continuación un ejemplo para entender mejor este concepto.

Partimos del ejemplo anterior y llamaremos al archivo `modulo2.lua`. En esta versión vamos a refactorizar la definición del módulo y le añadiremos una nueva función contador.

```
-- Declaración del módulo
local mod = function()
    -- Declaración de la tabla que contendrá las funciones del módulo
    local M = {}
    local contador = 0

    -- Definición de funciones

    -- Calcula el cuadrado de un número
```

```
    local function cuadrado(x)
        return x * x
    end

    -- Calcula el cubo de un número
    local function cubo(x)
        return x * x * x
    end

    -- Determina si un número es par
    local function esPar(x)
        return x % 2 == 0
    end

    -- Incrementa un contador
    local function incrementarContador()
        contador = contador + 1
        return contador
    end

    -- Se asignan las funciones cuadrado, cubo y esPar a la tabla M
    -- Esta asignación constituye la interfaz del módulo.
    M.cuadrado = cuadrado
    M.cubo = cubo
    M.esPar = esPar
    M.incrementarContador = incrementarContador

    -- Retorno de la tabla M
    return M
end

return mod
```

En este ejemplo, hemos definido el módulo como una función anónima asignada a la variable `mod`. Dentro de la función anónima declaramos el módulo, al igual que en el archivo `modulo1.lua` y le añadimos una nueva función. Esta función nos permitirá verificar el aislamiento entre instancias. En el módulo hemos definido una variable `contador` que está disponible en el ámbito del módulo, por lo que su supervivencia está ligada a la del módulo mismo. Cada llamada a la función `incrementarContador()` incrementará `contador` en una unidad.

Desde un punto de vista técnico, en la definición de este módulo, estamos haciendo uso de los *closures* de Lua, definiendo una serie de funciones dentro de una función envolvente.

El funcionamiento del módulo es sencillo: cada vez que llamamos al módulo, consumimos la función anónima, la cual nos devuelve una referencia distinta de la tabla interna `M`. Esto nos da la garantía de que las instancias están aisladas entre si. Definamos ahora el código de prueba.

```
local modulo2 = require 'modulo2'

local inst1 = modulo2()
local inst2 = modulo2()
local inst3 = modulo2()

inst1.incrementarContador()

inst2.incrementarContador()
inst2.incrementarContador()

inst3.incrementarContador()
inst3.incrementarContador()
inst3.incrementarContador()

print("Cuadrado de 5:", inst1.cuadrado(5))
print("Cubo de 5:", inst1.cubo(5))
print("5 es par?", inst1.esPar(5))
print("Contador:", inst1.incrementarContador())
print()
print("Cuadrado de 3:", inst2.cuadrado(3))
print("Cubo de 3:", inst2.cubo(3))
print("3 es par?", inst2.esPar(3))
print("Contador:", inst2.incrementarContador())
print()
print("Cuadrado de 2:", inst3.cuadrado(2))
print("Cubo de 2:", inst3.cubo(2))
print("2 es par?", inst3.esPar(2))
print("Contador:", inst3.incrementarContador())
```

Al igual que en el ejemplo anterior, hacemos referencia al módulo, en este caso `modulo2`. Creamos tres instancias del módulo, consumiento la función anónima y asignando el valor devuelto a variables distintas. Para probar el aislamiento hacemos varias llamadas a la funcion `incrementarContador()` en número variable para cada instancia. Finalmente imprimimos el resultado con diversos valores para cada instancia. El resultado que obtenemos es el siguiente:

```
Cuadrado de 5:   25
Cubo de 5:       125
5 es par?        false
Contador:        2

Cuadrado de 3:   9
Cubo de 3:       27
3 es par?        false
Contador:        3

Cuadrado de 2:   4
Cubo de 2:       8
2 es par?        true
Contador:        4
```

Observa que para `inst1` hemos llamado dos veces `incrementarContador()`, tres veces para el caso de `inst2` y de cuatro veces para `inst3`. Demostrando el aislamiento de las instancias.

Definición de un módulo como un singleton

En este caso también se define el módulo como un objeto, pero esta vez no se busca el aislamiento, sino más bien todo lo contrario. El singleton es un modelo de programación que busca tener una sola instancia de un objeto disponible para toda la aplicación. Esta técnica se usa, por ejemplo, para almacenar valores o datos que deben ser comunes para toda la aplicación. Un caso típico son los parámetros de la aplicación.

Vamos a ver un ejemplo, en el cual vamos a definir un módulo tipo singleton que contiene los parámetros de conexión a una base de datos. Partiremos de una tabla de parámetros por defecto y, posteriormente los modificaremos.

```
local instancia
local parametrosPorDefecto = {}

-- Definición de los parámetros por defecto
parametrosPorDefecto["host"] = "localhost"
parametrosPorDefecto["port"] = 5432
parametrosPorDefecto["dbname"] = "postgres"
parametrosPorDefecto["user"] = "postgres"
```

```
local mod = function()
    if not instancia then
        -- Declaración de las funciones
        local function obtenerHost ()
            return parametrosPorDefecto["host"]
        end

        local function obtenerPort ()
            return parametrosPorDefecto["port"]
        end

        local function obtenerDbname ()
            return parametrosPorDefecto["dbname"]
        end

        local function obtenerUser ()
            return parametrosPorDefecto["user"]
        end

        local function establecerHost (host)
            parametrosPorDefecto["host"] = host
        end

        local function establecerPort (port)
            parametrosPorDefecto["port"] = port
        end

        local function establecerDbname (dbname)
            parametrosPorDefecto["dbname"] = dbname
        end

        local function establecerUser (user)
            parametrosPorDefecto["user"] = user
        end

        -- Declaración de la tabla que contendrá las funciones del módulo
        local M = {}

        M.obtenerHost = obtenerHost
        M.obtenerPort = obtenerPort
```

```
        M.obtenerDbname = obtenerDbname
        M.obtenerUser = obtenerUser
        M.establecerHost = establecerHost
        M.establecerPort = establecerPort
        M.establecerDbname = establecerDbname
        M.establecerUser = establecerUser

        instancia = M
    end

    return instancia
end

return mod
```

Ahora definamos el código para consumir el módulo.

```
local modulo3 = require 'modulo3'

local inst1 = modulo3()
local inst2 = modulo3()
local inst3 = modulo3()

inst1.establecerHost("10.123.67.90")
inst1.establecerPort(5433)

inst2.establecerHost("67.98.234.1")
inst2.establecerUser("my_user")

print("Host de inst3:", inst3.obtenerHost())
print("Port de inst3:", inst3.obtenerPort())
print("Dbname de inst3:", inst3.obtenerDbname())
print("User de inst3:", inst3.obtenerUser())
```

Ahora ejecutamos el código anterior y obtenemos el siguiente resultado:

```
Host de inst3:   67.98.234.1
Port de inst3:   5433
Dbname de inst3:         postgres
User de inst3:   my_user
```

Como era de esperar, el valor final de los parámetros es aquel de la última modificación. El valor de `host` se ha modificado dos veces, su valor final es el de la última modificación (`67.98.234.1`). Para el valor de `dbname` tenemos `postgres` que es el valor por defecto, ya que ninguna llamada a instancia modificó este valor. Observa que hemos modificado el objeto desde tres instancias distintas, pero internamente hemos trabajado con la misma tabla. Con esto se demuestra el funcionamiento de un singleton.

La función `require`

La función `require` se usa para cargar y ejecutar módulos o librerías en Lua. Esta función se encarga de buscar el módulo y posteriormente verificar si ya ha sido ejecutado, para evitar asi, ejecutarlo dos veces.

La función `require` acepta un argumento, que es la ruta del módulo, que es en esencia, el nombre del módulo que queremos cargar.

Recordemos que el nombre de un módulo es el nombre del archivo donde está definido el módulo sin la extensión `.lua`.

Cuando requerimos el modulo, por ejemplo, `modulo3` Lua buscará el módulo en varios lugares distintos. Si no lo encuentra nos mostrará los lugares donde lo buscó:

```
no field package.preload['module3']
no file 'C:\Utilities\Lua\5.4.2\lua\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\lua\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\..\share\lua\5.4\module3.lua'
no file 'C:\Utilities\Lua\5.4.2\..\share\lua\5.4\module3\init.lua'
no file '.\module3.lua'
no file '.\module3\init.lua'
no file 'C:\Utilities\Lua\5.4.2\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\..\lib\lua\5.4\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\loadall.dll'
no file '.\module3.dll'
no file 'C:\Utilities\Lua\5.4.2\module354.dll'
no file '.\module354.dll'
```

Esos lugares se corresponden con unos patrones definidos en Lua para buscar los módulos. En este ejemplo de salida, `C:\Utilities\Lua\5.4.2` es donde se encuentra el ejecutable de Lua, que se corresponde con la variable `PATH`.

En todo caso, podemos definir la ruta completa al modulo, pero no funcionaría correctamente al portar el código a otras plataformas. Una solución, sería definir los módulos en una carpeta llamada ´modulos´ y requerir los módulos incluyendo la carpeta:

```
require "modulos/modulo3"
```

Definición de módulos en versiones antiguas

En las **versiones 5.0 y 5.1** de Lua, los módulos se definían de un modo diferente. A modo de referencia, veamos como era:

```
-- Definición de modulo1
module("modulo1", package.seeall)

-- Definición de funciones

-- Calcula el cuadrado de un número
function cuadrado(x)
    return x * x
end

-- Calcula el cubo de un número
function cubo(x)
    return x * x * x
end

-- Determina si un número es par
function esPar(x)
    return x % 2 == 0
end
```

En estas versiones, los modulos se usaban del siguiente modo:

```
require("modulo1")
modulo1.cuadrado(5)
modulo1.cubo(3)
```

```
modulo1.esPar(10)
```

Las versiones modernas no soportan este modelo, por lo que si ejecutas el código obtendrás un error.

Revisión #1
Creado 9 abril 2024 14:45:12 por Guillermo
Actualizado 9 abril 2024 14:46:25 por Guillermo