Programacion Lua por Venkman

Introducción a Lua

Escrito por Venkman bajo una licencia de Creative Commons

Lua es un lenguaje de scripting diseñado con el principal objetivo de ser ligero y extensible. El pequeño tamaño de su *runtime* y su API C sencilla han hecho que sea elegido en muchas ocasiones para ser embebido en aplicaciones como nmap o Monotone, o en multitud de juegos (WOW, FarCry...). De hecho, viene utilizándose en videojuegos desde hace unos diez años.

Lua apareció por primera vez en 1998 y tiene su origen en la Universidad Pontifica Católica de Rio de Janeiro [PUC-Rio].

El lenguaje

Lua es un lenguaje sencillo. Una de sus principales influencias declaradas es Lisp y su idea de una única estructura, la lista. En el caso de Lua, esta estructura clave fundamental de su arquitectura es la *tabla*. Como lenguaje más parecido podría nombrarse Javascript. Ambos son lenguajes de script, con soporte para OO basada en prototipos.

Lua es rápido para su naturaleza interpretada; suele quedar en buen lugar en las comparativas.

La sintaxis recuerda a una mezcla de Pascal con otros lenguajes. Un clásico:

```
-- Una función:
function diHola(persona)
      -- Un bucle:
      for i=1,3 do
            print("Hola " .. persona .. "!")
      end
end
print("¿Cómo te llamas?")
nombre = io.read()
-- Llamada a una función:
diHola(nombre)
```

Lua soporta un estilo procedural imperativo, pero también un estilo funcional. De hecho, en Lua las funciones son entidades de primera categoría. La función del ejemplo anterior, en realidad se interpreta como:

```
diHola = function(persona)
      print("Hola " .. persona .. "!")
```

```
end
```

Si tenéis cierta experiencia con Javascript quizá esa forma de escribirlo os sea familiar.

Lua, como decía antes, también soporta orientación a objetos, y lo hace basándose en la estructura de la *tabla*. Una tabla como t:

```
t = {}
```

puede contener funciones, como:

```
function t.diHola(persona)
      print("Hola " .. persona .. "!")
end
t.diHola("Peter")
```

O, si utilizamos **:** en lugar de **.** Lua nos pasará como primer parámetro *self*, una referencia a la tabla (u objeto):

```
t = {}
t.nombre = "Peter"
function t:diHola()
      print("Hola " .. self.nombre .. "!")
end
t:diHola()
-- Que es lo mismo que t.diHola(t) o que t['diHola'](t)
```

Con esto tenemos todo lo necesario para encapsular objetos. Para otros aspectos de la OO, Lua se se basa en la idea de *metatabla*, una forma de asociar a una tabla T otra T' en la que se irán a buscar los métodos que no se encuentren en T. Con esto tenemos un mecanismo de herencia. Entrar en detalle del manejo de las metatablas, OO, metaprogramación u otros aspectos más avanzados quedan (quizá) para posibles próximos artículos.

Referencias y Herramientas

La referencia fundamental para aprender Lua es, sin duda, el libro Programming in Lua, del cual se puede encontrar online la primera edición. Es un buen libro, bien escrito y con explicaciones detalladas pero asequibles a cualquiera. Como complemento, el Manual de Referencia que también se puede adquirir impreso o consultar online.

La otra referencia habitual es el Wiki de Usuarios de Lua donde podemos encontrar tutoriales y código, su lista de correo lua-l o el canal de IRC en Freenode.

Existen, además, unos cuantos libros y tutoriales más específicamente orientados a desarrollar modificaciones de determinados juegos, o a la programación de juegos en general.

El soporte por parte de los editores e IDEs es suficientemente amplio. Personalmente recomiendo SciTE un editor muy sencillo con la cualidad de que utiliza Lua para programar macros. Existen también una buena cantidad de librerías y enlaces que se encuentran listadas en el Wiki.

LuaForge es un buen lugar para encontrar proyectos de y para este lenguaje. El Kepler Project es otra interesante referencia que desarrolla y mantiene una variada colección de módulos principal pero no exclusivamente orientados al desarrollo web.

Por último, para comenzar con Lua en Windows se puede encontrar un cómodo instalador que incluye binarios de Lua, un buen número de librerías fundamentales y el editor SciTE, además de unos cuántos ejemplos y documentación.

Fuente: http://es.debugmodeon.com/articulo/introduccion-a-lua

[Lua] Detalles de sintaxis

Palabras reservadas

Lua cuenta con una lista relativamente pequeña de palabras reservadas. El lenguaje tan sólo define 21 de ellas.

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

Los comentarios de una única línea comienzan por dos guiones (--) y los bloques de comentario están delimitados por --[[y]]

Variables y Tipos

Lua es un lenguaje dinámico. No es necesario declarar el tipo de las variables, puesto que estas no tienen tipo. Los *valores* que pueden almacenar pueden ser de uno de los 8 tipos básicos que existen en Lua: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread* y *table*.

Se puede acceder a una variable antes de haberle asignado ningún valor. No se considera un error y su valor en ese caso será *nil*. nil es el único valor del tipo nil y su función es esta, ser distinto a cualquier otro valor.

Lua es de esos lenguajes que consideran como *false* los valores *false* y *nil*, y cualquier otra cosa como *true*. Detalle: *0* y *""* se evalúan como true en una condición. Lo único posiblemente diferente de otros lenguajes en cuanto a las cadenas de texto es la posibilidad de delimitarlas con [[y]] permitiendo así que contengan saltos de línea y otros caracteres literales.

Como en Javascript, por ejemplo, Lua realiza conversiones automáticas entre cadenas y números. Pero a diferencia de Javascript, si comparamos 10 y "10" no nos devuelve true.

La tabla es el tipo único de Lua para manejar todas las colecciones y estructuras de datos. table implementa un array asociativo y todo en Lua se organiza en torno a las tablas. De hecho, cuando llamamos a io.read(), una función de la librería o módulo io, lo que realmente estamos haciendo es acceder al elemento read de la tabla io. Las tablas son objetos dinámicos que se construyen siempre a partir de una expresión de tabla, como {} para una tabla vacía.

Como decíamos en el primer artículo de la serie, las funciones son entidades de primera categoría en Lua. Su tipo es function.

El tipo userdata sirve para tipos de datos que vienen de C (i.e. del sistema externo donde esté insertado Lua), y thread es el tipo que implementan las *corutinas*, que Lua soporta para ofrecer programación multihilo.

Construcciones básicas y bloques

Lua soporta -como se puede ver más arriba, en la lista de palabras reservadas- las construcciones típicas de bucles, asignaciones, condicionales, etc. La sintaxis puede recordar ligeramente a algún lenguaje tipo Pascal, sobre todo por los bloques delimitados por la palabra *end*.

Los bloques no sólo delimitan los trozos de código sino que también establecen el alcance de las variables declaradas como locales.

```
while i < 10 do
      local x = i +1
      -- ...
end
```

La variable x, declarada como local, sólo tiene alcance dentro del bloque while-do-end. Una variable local puede utilizar el mismo nombre que una (más) global y entonces enmascarará su referencia. Dentro del bloque x siempre hará referencia a la variable local.

Un detalle interesante de Lua es que soporta asignación múltiple directa, sin necesidad de utilizar listas u otros mecanismos. También las funciones pueden, y muy habitualmente así lo hacen las de las librerías estándar, devolver varios valores. Más aún, la asignación múltiple se considera una operación atómica, de modo que es perfectamente posible intercambiar dos valores en una única instrucción sin efectos secundarios:

```
a = 1
b = 2
a,b = b,a
print("a: " .. a .. ", b: " .. b)
-- imprimirá: a: 2, b: 1
```

El intérprete automáticamente descarta los valores sobrantes o rellena con *nil*s hasta completar la cantidad necesaria de valores sin que suponga error.

```
a,b,c = 1,2    -- c quedará sin asignar
a,b = 1,2,3,4  -- 3 y 4 se descartan
```

Funciones

Como decía, las funciones en Lua pueden devolver varios valores. Es simple cuestión de separarlos por comas en la instrucción return. e.g.:

```
function numeros()
      return 1,2,3,4
end
```

Similarmente, una función puede aceptar un número variable de argumentos declarando su firma con ...:

```
function imprimir(...)
      print(...)
end
```

Para quienes hayan visto algo de Perl, ... es similar a lo que allí sería @_

Las funciones, como hemos visto al hablar de los tipos, son valores de primera categoría. Esto quiere decir que igual que con otros valores, podemos asignarlos, pasarlas como parámetro, declararlas como local, etc.

Como otros detalles interesantes, Lua soporta closures y hace correctamente eliminación de llamadas finales (soporta recursividad de cola... *proper tail recursion*). También soporta, apoyado por las closures, un modo sencillo de definir iteradores, que se basa en devolver una función.

Fuente: http://es.debugmodeon.com/articulo/lua-detalles-de-sintaxis

[Lua] Tablas y metatablas (parte I)

En Lua las tablas son la base de cualquier estructura de datos. Y no se quedan sólo ahí, sino que proporcionan todos los mecanismos para la organización y arquitectura del código.

Tablas

Como vimos en la introducción de esta serie sobre Lua, *table* es uno de los 7 tipos existentes en Lua. Su creación se hace siempre a través de una *expresión constructora* de tabla. En su versión

mínima, la tabla vacía, es así:

```
tabla = {}
```

table implementa lo que normalmente conocemos como array asociativo con algunas peculiaridades. Si no especificamos claves, como en un array sencillo como:

```
tabla = {1,2,3,4,5,6}
```

Se aisgnan claves numéricas automáticamente (es decir, como un array tradicional). El detalle es que Lua es uno de esos lenguajes en que los índices de los arrays empiezan en 1 ;) Podemos hacer que empiecen por 0, pero la costumbre y convención es que empiecen por 1.

Con un array podemos hacer todas las operaciones típicas que se nos pueden ocurrir.

```
print(tabla[3])
tabla[2] = 7
tabla["hola"] = 16      -- Es lo mismo que tabla.hola = 16
print(tabla[44])        -- No da un error, simplemente devuelve nil
```

Dado que los elementos de una tabla pueden ser a su vez tablas, no necesitamos nada más para implementar cualquier otra estructura típica como listas enlazadas, colas, conjuntos, árboles, etc. Se sale del ámbito de este artículo explicar esto con detalle, pero se puede encontrar una muy buena explicación en el capítulo 11 de PIL.

Otro detalle en la importancia de las tablas en Lua, es que, dado que los elementos también pueden ser funciones, la tabla se convierte en la estructura básica de modularización y encapsulación del código. Las librerías no son más que tablas con funciones dentro. Cuando llamamos a...

```
fichero = io.open(FILE, 'r')
```

realmente no estamos haciendo otra cosa más que acceder al elemento open de la tabla io:

```
fichero = io["open"](FILE, 'r')
```

Objetos

En esto mismo se basa también la forma de Lua para proporcionar orientación a objetos. Los objetos son tablas. Tablas que contendrán datos y métodos. Con algunas pequeñas peculiaridades ofrecidas por Lua podemos disponer de todo lo necesario para estructurar nuestra arquitectura en objetos.

Veamos algunos de esos detalles partiendo de una tabla como

```
Caja = {abierta=false, contenido = nil}
```

Podemos, en primer lugar añadirle métodos para abrir y/o cerrar una caja:

```
function Caja.abrir()  { Caja.abierta = true  }
function Caja.cerrar() { Caja.abierta = false }
```

Lógicamente el uso de Caja.abierta directamente en el método no es una buena práctica, de modo que tendríamos que pasarnos el objeto receptor como parámetro. habitualmente se llama *self* a este parámetro:

```
function Caja.abrir(self)  { self.abierta = true  }
function Caja.cerrar(self) { self.abierta = false }
-- Llamaríamos con:
Caja.abrir(Caja)
```

Lo cual, claramente, es un rollo. Así que Lua nos ofrece una forma de especificar que queremos

recibir y enviar este parámetro self de forma automática. Esto lo hacemos con el operador **:** en lugar de **.**:

```
function Caja:abrir()  { self.abierta = true  }
function Caja:cerrar() { self.abierta = false }
```

Una llamada sería:

```
Caja:abrir()
```

En realidad, el operador : es poco más que decoración por encima de la convención de enviar el objeto receptor como primer parámetro, y hacerlo de una forma u otra es equivalente (e incluso compatible).

La filosofía de Lua de no complicar las cosas tiene que decir en lo referente al encapsulamiento que "si no quieres acceder algo de dentro de un objeto, no accedas"; Para quien necesite realmente establecer unos límites de privacidad en sus objetos, existe la posibilidad de utilizar variables locales y construir *closures* sobre ellas. Como es sabido arrays asociativos y closures, es todo lo que necesitas para establecer un sistema de orientación a objetos :)

Como veremos en la segunda parte de este artículo, las metatablas nos van a ofrecer todo lo que necesitamos para crear ese sistema de objetos, soportando, si lo necesitamos, clases, encapsulamiento y herencia (incluso múltiple si queremos). Más interesante que eso será ver el uso en sí de las metatablas.

Fuentes: http://es.debugmodeon.com/articulo/lua-tablas-y-metatablas-parte-i

[Lua] Tablas y metatablas (parte II)

Las tablas, como hemos dicho ya repetidas veces, son la estructura fundamental de Lua. Su comportamiento sin embargo no está definido para todas las operaciones que sí tienen comportamiento definido con otros tipos de datos. No podemos sumar dos tablas. {1,2,3} + {4,5} no es una operación definida en el propio lenguaje. Si lo intentamos Lua se queja porque intentamos realizar una operación aritmética sobre un valor de tabla.

Pero lo que Lua sí nos proporciona es la manera de definir nosotros mismos las operaciones sobre nuestras tablas (objetos). Ese mecanismo son las metatablas y metemétodos.

La asignación de una metatabla es simple:

```
t = {}
mt = {}
setmetatable(t,mt)
-- También podemos obtener la metatabla de una tabla con:
print(getmetatable(t))
```

Lo interesante viene cuando definimos metamétodos en la metatabla. Sea mt nuestra metatabla base para este ejemplo:

```
mt = { __add = function(a,b)
                  local s = {}
                  for k,v in ipairs(a) do s[#s + 1] = v end
                  for k,v in ipairs(b) do s[#s + 1] = v end
                  return s
              end
      }

Ahora, si asignamos mt como metatabla:

t1 = {1,2,3}
```

```
t2 = {4,5}
setmetatable(t1,mt)
```

Podemos ahora hacer:

```
t1 + t2
```

sin que dé error. De hecho, funciona como esperamos:

```
print(table.concat(t1+t2, ", "))
-- 1, 2, 3, 4, 5
```

Lo que ocurre es que Lua comprueba que tratamos de ejecutar la operación sobre la tabla y que esta no tiene definida la operación. Así que Lua comprueba si la tabla tiene asignada una metatabla y si esa metatabla tiene definido un metamétodo *__add*. En caso afirmativo, lo ejecuta pasándole como parámetros los operandos.

Igual que \add, existe una razonable lista de metamétodos como \mul, \sub, \div, \unm (negación), \mod (modulo) y \pow (potencia) para las operaciones aritméticas. También son definibles metamétodos como *__concat* para el operador de concatenación o \tostring para la conversión a cadena. También son soportados metamétodos para operadores relacoinales como \eq (igual), \lt (menor que) y \le (menor o igual que).

Como mecanismo de protección, podemos establecer el atributo \metatable de una tabla a algún valor distinto de nil. Si hacemos esto, impedimos que se pueda cambiar la metatabla a esa tabla.

index, newindex y __call

Además de los ya comentados existen 3 metamétodos cuyo uso da una flexibilidad mucho más amplia al lenguaje y nos permite cambiar de manera muy profunda el comportamiento de nuestras tablas.

__index es llamado cuando tratamos de leer un elemento de la tabla que no está definido. En anteriores artículos vimos que si tratábamos de hacer esto, únicamente obteníamos nil como respuesta. La realidad es que, antes de devolver ese nil, Lua comprueba si hay una metatabla asignada y si esta tiene definido \index. Si no, nos devuelve nil como ya sabemos, pero si lo tiene, entonces delega la responsabilidad llamando a este metamétodo.

__newindex es el metamétodo complementario: Es llamado (si está definido) cuando tratamos de asignar un valor a un elemento nuevo. \index y \newindex proporcionan potencia suficiente por sí mismos para establecer formas de herencia en programación orientada a objetos. También son útiles para implementar otro tipo de construcciones como, por ejemplo, tablas de sólo lectura o tablas con valores por defecto.

Por su parte, la utilidad de *__call* es algo diferente. Si dfinido, será el método que se ejecute cuando tratemos de llamar a una tabla como si fuera una función.

Como ejemplo incluyo a continuación varias implementaciones del clásico cálculo de números de Fibonacci. La implementación más directa (y menos conveniente) podría ser:

```
function fib(n) return n<2 and n or fib(n-1)+fib(n-2) end
print(os.clock())
for n=30,35 do print(fib(n)) end
print(os.clock())
```

Pero como es conocido tiene la pega de necesitar recalcular cada vez todos los números anteriores para llegar al que nos interesa. La solución es, como ya sabréis, ir guardando esos valores. En una primera aproximación podríamos guardarlos en una tabla global:

```
function fastfib(n)
```

```
      fibs={1,1} -- variable global
      for i=3,n do
            fibs[i]=fibs[i-1]+fibs[i-2]
      end
      return fibs[n]
end
print(os.clock())
for n=50,100 do print(fastfib(n)) end
print(os.clock())
```

Esto es mucho más rápido (mucho), pero no es muy elegante tener por ahí la tabla fibs definida globalmente. Así que podemos intentar algo así:

```
fibotabla = {[1] = 0, [2] = 1}
setmetatable(fibotabla, {
      __index = function (t, i)
            t[i] = t[i-1]+t[i-2]
            return t[i]
      end
})
print(os.clock())
for n=50,100 do print(fibotabla[n]) end
print(os.clock())
```

Lo que estamos haciendo es declarar una metatabla anónima para fibotabla. En ella definimos una función \index. Si el elemento ya está presente en la tabla, no llamará a \index sino que lo obtendrá directamente. Si no existe aún, el método asignado a \index calcula el número elemento.

La pega (aunque no es muy grande) es que ahora fibotabla es una tabla, no una función, de modo que accedemos con [] en lugar de (). Si esto nos supone un problema, podemos usar \call en lugar de \index:

```
fibocall = {[1] = 1, [2] = 1}
setmetatable(fibocall, {
      __call = function(t, ...)
            local args = {...}
            local n = args[1]
            if not t[n] then
                  for i = 3, n do
                        t[i] = t[i-2] + t[i-1]
                  end
            end
            return t[n]
      end
})
print(os.clock())
for n=50,100 do print(fibocall(n)) end
print(os.clock())
```

Fin de la serie

Hasta aquí la serie de artículos de introducción a las principales características de Lua. Espero que haya sido interesante para vosotros como lo ha sido para mi y que si alguien ha tenido alguna experiencia con Lua en algún entorno interesante, se anime a escribir comentando esas experiencias.

Fuente: http://es.debugmodeon.com/articulo/lua-tablas-y-metatablas-parte-ii