Estructuras de datos

- Tablas
- Arrays
- Manipulación de tablas

Tablas

Lua implementa un tipo de base con el que construir diferentes tipos de estructuras de datos. Se llaman tablas y ofrecen las características necesarias para la creación de estructuras de datos complejas que pueden ser manejadas de una manera eficiente.

En este capítulo vamos a ver qué es una tabla en Lua y diferentes tipos de estructuras de datos que pueden ser implementadas usando las tablas, para disponer de estructuras específicas para casos particulares, como arrays o diccionarios.

Tablas

Las tablas, *tables* por su denominación en inglés, en Lua son colecciones de pares clave/valor, cuyos valores pueden ser de cualquier tipo excepto `nil`. Constituyen la única estructura de datos disponible en Lua y sirve de base para la creación de otras estructuras de datos. Las tablas se indexan por la clave, así, se puede recuperar cualquier valor de la tabla si conocemos la clave. Dado que las claves y valores de las tablas en Lua pueden ser de cualquier tipo, una clave o un valor pueden ser a su vez de tipo tabla. Las tablas en Lua son dinámicas: no tienen un tamaño definido y pueden crecer según las necesidades.

A partir de una tabla podemos construir cualquier tipo de estructura de datos más compleja o específica, como por ejemplo arrays, arrays multidimensionales, matrices, listas, colas, etc.

Para facilitar la manipulación de las tablas, Lua dispone de una librería estándar que define una serie de funciones para operar con las tablas.

Sintaxis de las tablas

Las tablas se definen por medio de los constructores de tablas, que se representan por los caracteres de llaves `{` y `}`. Una tabla vacía se define como a continuación.

```
> tabla = {}
> print(tabla)
table: 00000000007d4230
```

En el ejemplo anterior el valor retornado por la función print es el tipo y el identificador del objeto. Para poder recuperar un valor contenido en la tabla debemos hacer referencia a un elemento por su clave.

Uso de las tablas

La tabla es un tipo complejo y flexible que permite la gestión de estructuras complejas de datos. A continuación tratamos las diferentes posibilidades de uso.

Definición y asignación

Una vez definida una tabla, podremos acceder a cada uno de sus elementos a través de la clave. Para insertar valores en una tabla, damos valor a la clave y le asignamos un valor.

```
> tabla = {}
> tabla["clave"] = "Lua"      -- Insertamos con la clave 'clave'  el valor 'Lua', ambos cadenas
> tabla[1] = 1000             -- Insertamos con la clave '1'  el valor '1000', ambos enteros
> = tabla["clave"]            -- Recuperamos el valor de la clave 'clave'
Lua
> = tabla[1]                  -- Recuperamos el valor de la clave '1'
1000
```

Como ves en el ejemplo anterior, las claves pueden ser de cualquier tipo, pudiéndose incluso mezclar. Si bien, por lo general, en muchos tipos de estructuras de datos las claves suelen ser del mismo tipo. Del mismo modo, los tipos de la clave y el valor pueden ser diferentes también. En todos los casos la clave nunca puede ser `nil` ni `NaN` (not a number).

```
> tabla = {}
> clave = {}
> funcion = function () end
> tabla[clave] = 250            -- Asignamos 250 a la clave 'clave'  que es una tabla
> tabla[funcion] = 500          -- Asignamos 500 a la clave 'funcion'  que es una funcion
> = tabla[clave]                -- Recuperamos el valor de la clave 'clave'
250
> = tabla[funcion]              -- Recuperamos el valor de la clave 'funcion'
500
> tabla[nil] = 125              -- Asignar un valor a una clave 'nil'  produce un error
stdin:1:  table index is nil
stack traceback:
        stdin:1:  in main chunk
        [C]:  in ?
> tabla[0/0] = 450              -- Asignar un valor a una clave 'NaN'  también produce error
stdin:1:  table index is NaN
stack traceback:
        stdin:1:  in main chunk
        [C]:  in ?
```

Cuando se dice que una clave puede ser de cualquier tipo, significa también que una clave puede ser el resultado de una expresión válida cualquiera.

```
> tabla = {}
> tabla[1/0] = 750                                    -- Clave basada en una operación aritmética
> tabla[0/1] = 1000                                   -- Otro ejemplo de operación aritmética
> = tabla[1/0]
750
> = tabla[0/1]
1000
> tabla[10^3] = 1250                                  -- Operación aritmética: exponenciación
> = tabla[10^3]
1250
> tabla[string.gsub("aa", "a", "b", 1)] = 1500     -- Clave basada en el retorno de la llamada
a una función
> = tabla[string.gsub("aa", "a", "b", 1)]
1500
```

Acceso a claves no definidas

En el caso de que recuperemos el valor de una clave que no existe, no se produce ningún error y devuelve `nil`.

```
> tabla = {}
> = tabla[100]
nil
```

Supresión de pares

Cualquier clave que hayamos declarado puede ser eliminada asignándole el valor `nil`.

```
> tabla = {}
> tabla["clave"] = 10
> = tabla["clave"]
10
> tabla["clave"] = nil
> = tabla["clave"]
nil
```

Acceso simplificado a claves tipo string

Las tablas también ofrecen un modo de acceso sencillo para las claves de tipo `string`, reduciendo el código necesario para acceder a una clave. Los únicos requisitos para este tipo de claves es respetar que sean de tipo `string` y que tengan la siguiente sintaxis:

> *Una cadena que contenga letras, números y guión bajo (_ underscore). No puede comenzar por un número.*

```
> tabla = {}
> tabla.clave = 1000
> = tabla.clave
1000
> tabla.1 = 500
stdin:1:  syntax error near '.1'
> tabla._1 = 500
> = tabla._1
500
```

Definición y asignación avanzada

Hasta ahora hemos visto una forma para la definición de una tabla: primero declaramos la tabla y luego le asignamos los valores. Sin embargo Lua ofrece la posibilidad de declarar e inicializar una tabla en una sola expresión. Veamos un ejemplo de definición y asignación en una sola sentencia.

```
tabla = {["clave"] = "valor", [123] = 1000}
= tabla.clave
valor
= tabla["clave"]
valor
= tabla[123]
1000
```

En el ejemplo anterior, puedes observar que en la primera línea, a la variable `tabla`, se le asigna una tabla en la cual los pares clave valor le son definidos en la misma línea.

Arrays

En el capítulo anterior hemos visto en detalle las tablas, que son la estructura de datos básica de Lua. Las tablas son

extremadamente flexibles y permiten crear tipos específicos de estructuras de datos. En este nuevo capítulo vamos a ver con más detalle un tipo específico de tablas: los arrays.

Los arrays son estructuras de datos indexadas, formadas por una lista de pares de elementos, cada elemento tiene una clave numérica y un valor que puede ser de cualquier tipo, incluyendo otros arrays. Por lo tanto un array en Lua no es más que una tabla en la que las claves son numéricas.

Como las tablas, los arrays no tienen una longitud fija y pueden albergar tantos datos como los recursos disponibles lo permitan.

Arrays unidimensionales

> A diferencia de otros lenguajes, en Lua, los índices empiezan en 1 y no en 0. Por ello diremos que los arrays en Lua son de índice 1.

Los arrays unidimensionales representan una lista de un solo nivel. Se pueden crear en Lua definiendo una tabla simple:

```
ar = {"elemento 1", "elemento 2", "elemento 3"}


for i = 1, 3 do
  print(i, "-", ar[i])
end
```

En el ejemplo anterior, hemos definido una tabla con tres elementos de tipo cadena. Automáticamente, como no hemos definido una clave para cada uno de ellos, Lua le asigna una clave numérica incremental. Así *elemento 1* tendrá el índice 1 y *elemento 3* tendrá índice 3. Cuando ejecutes el código anterior tenemos la siguiente salida:

```
1       -       elemento 1
2       -       elemento 2
3       -       elemento 3
```

Si un índice de un array no existe, cuando intentamos accederlo, Lua devuelve `nil`:

```
> ar = {"elemento 1", "elemento 2", "elemento 3"}
> print (ar[4])
nil
```

Un array también puede ser definido como una tabla vacía e inicializarlo usando un bucle, por ejemplo:

```
ar = {}

for i = 1, 10 do
  ar[i] = i
end

for i = 1, 10 do
  print(ar[i])
end
```

El ejemplo anterior produce la siguiente salida:

```
1
2
3
4
5
6
7
8
9
10
>
```

Array multidimensional

Los arrays multidimensionales son aquellos que tienen otros arrays anidados. Un ejemplo de array multidimensional se muestra a continuación:

```
> md = {{"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}}
> print(md[1][2])
```

```
elemento 1.2
```

En la primera línea del ejemplo definimos un array de dos elementos cuyos valores son dos arrays de dos elementos cada uno. Para acceder al los elementos de cada array hay que poner grupos de corchetes según los niveles que queremos acceder. En nuestro ejemplo `md[1][2]`, el primer grupo de corchetes por la izquierda `[1]` representa al array padre, que contiene dos arrays; el segundo grupo `[2]`, representa a un elemento dentro del array hijo. Por ello `md[1][2]`, representa al elemento *elemento 1.2*, que corresponde con el segundo elemento del primer array.

Los arrays multidimensionales se pueden recorrer anidando los bucles también. Primeramente tendremos un bucle que recorerá el array padre y luego otro bucle anidado que recorrerá el array hijo. Observa el siguiente ejemplo:

```
 md = {{"elemento 1.1", "elemento 1.2"}, {"elemento 2.1", "elemento 2.2"}}


for i = 1, 2 do
  for j = 1, 2 do
    print(i, j, md[i][j])
  end
end
```

Al ejecutar el código anterior se produce la siguiente salida:

```
1       1       elemento 1.1
1       2       elemento 1.2
2       1       elemento 2.1
2       2       elemento 2.2
```

Hasta ahora se ha presentado el caso de un array bidimensional, pero, no hay limitación en cuanto a los niveles de un array multidimensional.

Longitud de un array

Hasta ahora hemos mostrado en todos los ejemplos el valor constante de un array para recorrerlos con un bucle. En la mayoría de aplicaciones prácticas donde existan arrays de longitud variable, este método no es muy práctico.

El operador `#` de Lua permite recuperar la longitud de un array. Ahora, podemos reescribir uno de los ejemplos anteriores para que sea más dinámico y no tan rígido:

```
ar = {}
```

```
for i = 1, 10 do
  ar[i] = i
end

ar[#ar + 1] = #ar + 1

for i = 1, #ar do
  print(ar[i])
end
```

En el ejemplo anterior, inicializamos un array con 10 elementos del 1 al 10. En la siguiente línea usamos el operador `#` para introducir en la posición `# + 1` el valor `# + 1`, que corresponde con 11. De este modo podemos alterar de forma dinámica nuestro array, sin importar si tiene 5, 20 o 1000 elementos. Ahora cuando ejecutamos el código anterior obtenemos el siguiente resultado:

```
1
2
3
4
5
6
7
8
9
10
11
```

Asignación de valores en un array

Una vez inicializados los arrays, sus valores pueden ser modificados en todo momento. Observa los siguientes ejemplos:

```
ar = {"elemento 1", "elemento 2", {"elemento 3.1", {"elemento 3.2.1", "elemento 3.2.2"},
"elemento 3.3"}}

ar[2] = "modificado 2"
ar[3][2][1] = "modificado 3.2.1"
ar[3][3] = "modificado 3.3"

print(ar[2])
```

```
print(ar[3][2][1])
print(ar[3][3])
```

Al ejecutar el código anterior obtenemos:

```
modificado 2
modificado 3. 2. 1
modificado 3. 3
```

Manipulación de tablas

Lua implementa funciones que permiten la manipulación de las tablas facilitando la realización de operaciones con ellas. Todas ellas están definidas en el módulo `table` de la líbrería estándar de Lua. Vamos a ver a continuación más en detalle estas funciones:

- table.concat()
- table.insert()
- table.remove()
- table.move()
- table.sort()
- table.unpack()
- table.pack()

Función table.concat

La función `concat` permite la concatenación de elementos de una tabla. Todos los elementos de la tabla deben ser cadenas o números. Veamos la signatura del método:

```
table.concat(lista, [, sep [, i [, j]]])
```

Detalle de los argumentos:

> **lista**. Lista que se quiere concatenar, todos sus elementos deben ser cadenas o números.
> **sep**. Separador, opcional. Es el caracter que se usará como separador. Si no se especifica se usa por defecto la cadena vacía.
> **i**. Indice de inicio, opcional. Permite definir el índice del primer elemento que se concatenará. Por defecto es 1.
> **j**. Indice de fin, opcional. Permite definir el índice del último elemento que se concatenará. Por defecto es `#lista`.

Veamos ahora un ejemplo del uso de la función `concat`:

```
paises = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}


-- Concatenación de la lista con los parámetros por defecto
print ("Concatenacion con parametros por defecto: ", table.concat(paises))
```

```
-- Concatenación con un caracter separador
print ("Concatenacion con un separador: ", table.concat(paises, "; "))

-- Concatenación de una sublista
print ("Concatenacion de una sublista: ", table.concat(paises, "; ", 2, 4))
```

Al ejecutar él código anterior, obtenemos el siguiente resultado:

```
Concatenacion con parametros por defecto:       SuizaEspanaColombiaItaliaFrancia
Concatenacion con un separador:         Suiza; Espana; Colombia; Italia; Francia
Concatenacion de una sublista:  Espana; Colombia; Italia
```

En la primera línea, los elementos de la lista se han concatenado sin separación, ya que por defecto, se usa una cadena vacía. En la segunda, se ha definido un separador. En la tercera se han definido, además del separador, el elemento de inicio y fin para la concatenación.

Función table.insert

Permite insertar un elemento en la tabla en la posición especificada, desplazando los demás elementos si es necesario. La signatura de este método es como sigue:

```
table.insert(lista, [ pos,] valor)
```

Los argumentos son:

lista. Lista a la cual queremos insertarle un elemento.
pos. Posición, opcional. Define la posición donde queremos insertar el elemento dentro de la lista. Por defecto los elementos se insertal al final de la lista.
valor. Valor a insertar en la lista.

Veamos a continuación un ejemplo de uso de la función `insert`:

```
paises = {"Suiza", "Espana", "Colombia"}

-- Inserción de un elemento al final de la lista
table.insert (paises, "Cuba")
print ("Se agrego un elemento al final de la lista:", table.concat (paises, ", "))

-- Inserción de un elemento en el índice 3 de la lista
table.insert (paises, 3, "Venezuela")
print ("Se agrego un elemento en el indice 3 de la lista:", table.concat (paises, ", "))
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
Se agrego un elemento al final de la lista:     Suiza, Espana, Colombia, Cuba
Se agrego un elemento en el indice 3 de la lista:       Suiza, Espana, Venezuela, Colombia,
Cuba
```

En la primera línea, insertamos el elemento "Cuba" al final de la lista. En la segunda línea, insertamos el elemento "Venezuela" en la posición 3, como podemos ver, elemento que estaba en la posición 3, "Colombia", ha sido desplazado a la posición 4 y a su vez el elemento en la posición 4 ha sido desplazado a la posición 5.

Función table.remove

La función `table.remove` permite eliminar un elemento de la lista. Veamos la signatura del método:

```
table.remove (lista [, pos])
```

El detalle de los argumentos es:

> **lista**. Lista de la cual eliminaremos el elemento.
> **pos**. Posición, opcional. Posición que ocupa el elemento que queremos eliminar. Si no se especifica, se eliminará el último elemento de la lista.
>
> **Retorno**: esta función retorna el elemento eliminado.

Veamos un ejemplo del uso de la función `table.remove`:

```
paises = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}

-- Eliminamos el cuarto elemento de la lista
pais = table.remove (paises, 4)
print ("El pais eliminado de la lista es:", pais)
print ("Contenido de la lista:", table.concat (paises, ", "))

-- Vaciamos la lista
contenido = table.remove(paises)
print ("Paises eliminados de la lista:", contenido)
print ("Contenido de la lista:", table.concat(paises, ", "))
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
El pais eliminado de la lista es:       Italia
Contenido de la lista:  Suiza, Espana, Colombia, Francia
Paises eliminados de la lista:   Francia
Contenido de la lista:  Suiza, Espana, Colombia
```

En la primera línea se ha eliminado "Italia" cuya posición es 4 dentro de la lista. En la segunda línea vemos los elementos restantes en la lista después de eliminar uno. En la tercera línea, no hemos especificado ningún elemento, por lo que se eliminará el último elemento de la lista, que es, "Francia".

Función table.move

La función table.move nos permite copiar elementos de una tabla a otra. Podemos copiar los elementos en una nueva tabla o bien copiarlos en la misma tabla. Esta función está disponible a partir de Lua 5.3. Esta es la signatura del método:

```
table.move (lista1, desde, hasta, insercion, [, lista2])
```

Veamos el detalle de los argumentos:

lista1. Lista desde donde copiaremos los elementos.
desde. Índice desde el cual comenzarán a copiarse elementos de la lista1.
hasta. Índice hasta el cual se copiarán elementos de la lista1.
insercion. Indice de la lista2 donde se insertarán los elementos copiados.
lista2. Opcional, lista donde se insertarán los elementos copiados. Si no se especifica una lista, el destino será lista1.

Retorno: devuelve lista2.

Vamos a ver un ejemplo de la función `table.move`:

```
paises = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}
otros = {"Alemania"}

table.move (paises, 2, 3, 2, otros)

print ("Paises en tabla otros:", table.concat(otros, ", "))
print ("Paises en tabla paises:", table.concat(paises, ", "))
```

El resultado que obtendremos es el siguiente:

```
Paises en tabla otros:  Alemania, Espana, Colombia
Paises en tabla paises: Suiza, Espana, Colombia, Italia, Francia
```

Como puedes observar hemos copiado los elementos "Espana" y "Colombia" dentro de la nueva lista a continuación del elemento "Alemania", cuyo resultado es una lista con tres elementos. Por otro lado, vemos que la lista original conserva todos sus elementos.

Función table.sort

La función `table.sort`, permite ordenar los elementos de una lista. La signatura de la función es como sigue:

```
table.sort (lista [, comparador])
```

Veamos el detalle de los argumentos:

> **lista**. Lista que queremos ordenar.
> **comparador**. Opcional. Función que acepta dos elementos de la lista y que devuelve `true` si el primer elemento debe ir delante del segundo elemento. Si no se especifica, se usará el operador `<` de Lua.

Veamos un ejemplo:

```
paises = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}

-- Ordenar los países por orden alfabético
table.sort (paises)
print ("Paises ordenados por orden alfabetico:", table.concat(paises, ", "))

-- Ordenar los países por orden alfabetico inverso
table.sort (paises, function (el1, el2) return el1 > el2 end)
print ("Paises ordenados por orden alfabetico inverso:", table.concat(paises, ", "))

function reordena (elementoA, elementoB)
    if elementoA < elementoB then
        return true
    else
        return false
    end
end
```

```
-- Ordenar las cifras usando una función externa
cifras = {23, 56, 11, 9, 32, 60, 3}

table.sort (cifras, reordena)
print ("Cifras ordenadas por funcion:", table.concat(cifras, ", "))

function inversa (elementoA, elementoB)
    if elementoA > elementoB then
        return true
    else
        return false
    end
end

-- Ordenar las cifras usando una función externa inversa
table.sort (cifras, inversa)
print ("Cifras ordenadas por funcion inversa:", table.concat(cifras, ", "))
```

Al ejecutar el ejemplo anterior, obtendremos el siguiente resultado:

```
Paises ordenados por orden alfabetico:  Colombia, Espana, Francia, Italia, Suiza
Paises ordenados por orden alfabetico inverso:  Suiza, Italia, Francia, Espana, Colombia
Cifras ordenadas por funcion:   3, 9, 11, 23, 32, 56, 60
Cifras ordenadas por funcion inversa:   60, 56, 32, 23, 11, 9, 3
```

Función table.unpack

La función `table.unpack()` devuelve los elementos de una lista. Su signatura es:

```
table.unpack (lista [,i [,j]])
```

Veamos el detalle de los argumentos:

> **lista**. Lista de la cual se extraerán los elementos.
> **i**. Opcional. Índice a partir del cual comenzar a extraer elementos. Su valor por defecto es 1.
> **j**. Opcional. Índice del último elemento a extraer. Su valor por defecto es el último elemento de la lista (`#lista`)
>
> **Retorna**: los elementos extraidos.

Veamos a continuación un ejemplo de la función `table.unpack`:

```
paises = {"Suiza", "Espana", "Colombia", "Italia", "Francia"}

espana, colombia = table.unpack (paises, 2, 3)

print ("Paises extraidos:", espana, colombia)
```

Cuyo resultado es:

```
Paises extraidos:       Espana  Colombia
```

Funcion table.pack

La función `table.pack` crea una nueva tabla a partir de un número arbitrario de elementos pasados como argumentos. Su signatura corresponde con:

```
table.pack (elm1 [, elm2 [, elm_n]])
```

> **elm1, elm2 ... elm_n**. Elementos que serán insertados en la nueva tabla.
>
> **Retorna**: una nueva tabla con los elementos pasados.

Veamos un ejemplo de `table.pack`:

```
espana = "Espana"
colombia = "Colombia"
peru = "Peru"
argentina = "Argentina"
mexico = "Mexico"

paises = table.pack (espana, colombia, peru, argentina, mexico)

print ("Lista de paises:", table.concat(paises, ", "))
```

Si ejecutamos el código anterior obtendremos el siguiente resultado:

```
Lista de paises:        Espana, Colombia, Peru, Argentina, Mexico
```