8

Tables

Les tables Lua sont des tableaux associatifs offrant la possibilité d'implémenter la plupart des structures de données utiles au programmeur. Les tables peuvent donc recevoir tout type d'indice (sauf `nil` et `NaN`), mais aussi tout type de valeur, y compris des fonctions. On peut donc les utiliser pour implémenter des tableaux numériques, des enregistrements mais aussi des objets, des listes, des piles ou encore des queues. En outre, Lua offre des options syntaxiques intéressantes, permettant une écriture appropriée à chaque cas d'utilisation.

En interne, Lua distingue deux types de tables :

- Les tableaux numériques, appelés aussi *séquences*. Ces tables sont indexées numériquement et séquentiellement à partir de 1 et bénéficient de certains traitements et fonctions accélérés pour les parcourir et les traiter.
- Les autres tables, indexées par n'importe quelle valeur. On y regroupe les *hash-tables*, les dictionnaires, les structures ou enregistrements, les objets, les tables de fonctions, etc. Ces tables bénéficient elles aussi de fonctions de parcours et de traitements spécifiques.

Note *› Pour être complet, il faudrait ajouter un troisième type de tables : les métatables, qui permettent de modifier les opérations de base.*

57. Construire une table et y accéder

En Lua, les accolades permettent de construire ou d'initialiser une table : `{ }`.

```
local data_table = {} -- Construit une table vide
```

Une fois une table construite, on peut y placer des valeurs et y accéder grâce aux crochets qui encapsulent l'indice `[ ]`.

```
-- Construire la table
local t = {}
-- Stocker des valeurs
t[1] = 10
t[2] = 100
t[3] = "THIS IS A STRING"
t["NAME"] = "YOUR NAME HERE"
-- Récupérer ces valeurs
print(t[1]) --> 10
```

58. Initialiser une table

On peut initialiser directement une table lors de sa construction :

```
-- Avec des indices numériques automatiques à partir de 1
local t = { "DUPONT", "JEAN", 32 }
print(t[1], t[2], t[3]) --> DUPONT  JEAN  32

-- Avec des indices numériques définis
local t = { [40] = "DUPONT", [64] = "JEAN", [96] = 32 }
print(t[40], t[64], t[96]) --> DUPONT  JEAN  32

-- Comme un enregistrement
local t = { ["LASTNAME"] = "DUPONT", ["FIRSTNAME"] = "JEAN",
  ["AGE"] = 32 }
print(t["LASTNAME"], t["FIRSTNAME"], t["AGE"])
```

La syntaxe exacte d'initialisation est `[field] = value`. Comme vous l'avez compris, si `[field]` est omis, un indice numérique partant de 1 est affecté. De plus, pour les enregistrements, la simplification suivante est aussi supportée :

```
-- Directement : FIELD=VALUE est interprété comme ["FIELD"]=VALUE
local t = { LASTNAME = "DUPONT", FIRSTNAME = "JEAN", AGE = 32 }
```

L'utilisation des crochets est importante pour certains indices, tels que les mots clés, des appels à des fonctions ou bien pour différencier des indices numériques d'indices textuels :

```
local t = { ["64"] = "CELL 64 - TEXT", [64] = "CELL 64 - NUM" }
-- t[64] N'EST PAS t["64"]

local function prefix_idx(idx) return "PREFIX_" .. idx end
local t = { [prefix_idx(100)] = "DATA CELL" }
print(t["PREFIX_100"])
```

59. Créer et utiliser un enregistrement

Une table de type enregistrement se construit usuellement avec des indices de type *string* :

```
local t = { LASTNAME = "DUPONT", FIRSTNAME = "JEAN", AGE = 32 }
-- Est équivalent à
local t = {}
t["LASTNAME"] = "DUPONT"
t["FIRSTNAME"] = "JEAN"
t["AGE"] = 32
```

Lua offre une option syntaxique permettant de simuler une structure de données avec des champs nommés :

```
t["LASTNAME"] = "DUPONT"
-- Strictement équivalent à
t.LASTNAME = "DUPONT"
```

Cette syntaxe est naturellement beaucoup plus agréable à lire et à utiliser.

60. Comment obtenir la taille d'une table ?

Sur des tables indexées numériquement et séquentiellement à partir de 1 (c'est-à-dire les *séquences*), l'opérateur `#` retourne sa taille.

```
local tabvalue = { 0, 1, 2, 3, 4, 5, 6 }
print(#tabvalue) --> 7
```

Sur tous les autres types de table, `#` retourne une valeur qui va dépendre de la première valeur `nil` trouvée :

```
local tabvalue = { [8]=4, [9]=5, [10]=6 }
print(#tabvalue) --> 0
local tabvalue = { STRING_IDX = 20, ["40"] = 100 }
print(#tabvalue) --> 0
```

L'opérateur retourne `0` dans les deux cas, car il ne s'agit pas de tables séquentiellement indexées à partir de 1. Il faut donc utiliser cet opérateur avec rigueur et uniquement sur des séquences (ou sur des chaînes de caractères, bien sûr).

Pour obtenir la taille d'une table autre qu'une séquence, il est donc préférable de parcourir la table à l'aide – en général – d'une boucle `for`. À partir de Lua 5.2, il est possible de programmer cet opérateur en recourant à une *métatable* et son champ `__len`, comme expliqué au chapitre *Les métatables*.

61. Pourquoi une indexation numérique à partir de 1 ?

Cette question revient souvent, car elle peut perturber les développeurs ayant déjà pratiqué d'autres langages. L'indexation à partir de 1 vient de l'histoire de Lua, qui était initialement plutôt destiné à des ingénieurs avec un bagage mathématique, mais peu habitués à la programmation. En cela, Lua est semblable au FORTRAN. L'indice de position 0 est totalement valide, mais les fonctions utilisant des séquences ne tiendront pas compte de cet indice ; il en va de même pour l'opérateur de taille `#`.

62. Table de fonctions

On peut aussi utiliser les tables pour créer des tables d'indirections en fonction d'un indice d'entrée.

```
-- Table de sauts
local hashjump = {
  FORWARD  = function (data)
    -- gère l'avance...
  end,
  BACKWARD = function (data)
    -- gère le recul...
  end
}
-- Récupération de la commande (on suppose ici une fonction
-- externe renvoyant la commande et une donnée)
local cmd, data = get_external_command()
-- On vérifie la validité de la commande et de la donnée associée
if hashjump[cmd] and data ~= nil then
  hashjump[cmd](data)
else
  print("Unsupported command!")
end
```

Note *› Le code ci-dessus est aussi une manière de simuler un `switch/case` C/C++.*

63. Instancier des objets avec les tables

Comme les tables Lua acceptent tout type de données comme indices ou valeurs, on peut y référencer des fonctions et des données en même temps. Ainsi, on transforme les tables Lua en objets, capables de contenir méthodes et données.

```
-- Construction d'un objet room avec deux champs et une méthode
local room1 = {
  width  = 10, height = 5,
  get_surface = function (obj)
    return obj.width * obj.height
  end
}
-- Appel de la méthode
print(room1.get_surface(room1)) --> 50
-- Simplification syntaxique apportée par Lua
print(room1:get_surface()) --> 50
```

L'appel *via* l'opérateur `:` permet de s'affranchir d'indiquer à nouveau la table en premier paramètre et ainsi de renseigner automatiquement la variable locale `obj`. Cette simplification syntaxique est très agréable à utiliser. Il est aussi possible d'utiliser l'opérateur `:` dans la déclaration de la méthode elle-même :

```
local room1 = { width  = 10, height = 5 }
function room1:get_surface() return self.width * self.height end
-- Appel de la méthode
print(room1.get_surface(room1)) --> 50
-- Simplification syntaxique apportée par Lua
print(room1:get_surface()) --> 50
```

La syntaxe `function class:method(param)` est une simplication syntaxique de `function class.method(self, param)`. Une variable locale nommée `self` est automatiquement ajoutée comme premier argument à la fonction. C'est un peu l'équivalent de `this` en C++, C#, Java ou JavaScript.

Note *› Le nom `self` n'est pas un mot clé en Lua, mais un peu comme `_ENV`, il est utilisé par le compilateur comme une variable spéciale. Il n'est dès lors pas recommandé d'utiliser ce nom de variable dans un autre contexte.*

Naturellement, il s'agit ici d'une approche objet très simplifiée : on ne bénéficie pas de la notion de classe, d'héritage et des principes de données et méthodes privées. En revanche, c'est possible en recourant aux métatables. On appelle métaméthode un champ d'une métatable contenant une valeur de type fonction.

64. Parcourir une table avec un *for* générique

Pour parcourir une table, on utilise généralement une boucle `for` générique (de préférence aux boucles numériques). Lua fournit deux fonctions d'itération prédéfinies : `pairs` et `ipairs`. La fonction interne `ipairs` est destinée à être utilisée avec des séquences, tandis que `pairs` permet de parcourir toutes les tables.

```
-- Parcours d'une séquence avec ipairs
local datatab = { 78, 124, 12, 90, 12.6, 54.89 }
for k, v in ipairs(datatab)
  -- k contient l'indice, ici de 1 à 6, et v contient la valeur
  print(k, v)
end

-- Parcours d'une séquence avec pairs
-- Possible, mais moins performant que l'utilisation de ipairs
local datatab = { 78, 124, 12, 90, 12.6, 54.89 }
for k, v in pairs(datatab)
  -- k contient l'indice, ici de 1 à 6, et v contient la valeur
  print(k, v)
end

-- Parcours d'une table hétérogène de type objet
local room1 = { width  = 10, height = 5 }
function room1:get_surface() return self.width * self.height end
for k, v in pairs(room1)
  print(k, v)
end
```

Le dernier exemple renvoie typiquement :

```
get_surface function: 0x03fab718
width 10
height  5
```

Comme pour d'autres opérations, on peut modifier le fonctionnement des itérateurs `pairs/ipairs` avec l'aide des métaméthodes `__pairs/__ipairs`.

65. Copier une table

Copier une table nécessite l'utilisation d'une boucle afin de recopier l'intégralité des champs présents. La fonction ci-dessous effectue aussi une copie des tables sous-jacentes et des métatables.

```
-- Fonction de copie
function copy_table(intab)
  local outtab = {}
  -- Boucle de copie des champs et récursion éventuelle (tables)
  for k, v in pairs(intab) do
    outtab[k] = type(v) == "table" and copy_table(v) or v
  end
  -- Recopie éventuelle de la métatable présente
  if getmetatable(intab) then
    setmetatable(outtab, copy_table(getmetatable(intab)))
  end
  return outtab
end

tab = { 34, 55, 23, 34 ,12, {12, 32, 16}, 34, 23 }
out = copy_table(tab)
print(tab, out) --> table: 0x636bc0  table: 0x635c40
print(table.unpack(out))
--> 34  55  23  34  12  table: 0x636a80  34  23
```

66. Que sont les *weak tables* ?

Les *weak tables* sont des tables dont les clés, les valeurs ou les deux ne sont pas comptées comme référence par le ramasse-miettes de gestion de la mémoire. Cela est fréquemment utilisé pour l'écriture de cache, où il est important que la table ne grossisse pas indéfiniment. On peut rendre une table *faible* en ajoutant un champ `__mode` à sa métatable, qui peut contenir `"k"`, `"v"` ou `"kv"`, suivant qu'on veut que les clés, les valeurs ou les deux ne soient pas prises en compte par le ramasse-miettes.

```
local t = setmetatable({}, {__mode="kv"})
-- Comme t est weak, la quantité de mémoire n'augmente pas
-- durant l'exécution (essayez de changer __mode).
for i=1,math.huge do
  t[i] = {}
  collectgarbage('collect')
  print(i, collectgarbage('count'))
end
```

67. Utiliser des tables pour faire des matrices

Comme en C/C++, il est tout à fait possible d'utiliser les tables pour créer des matrices à *n* dimensions. Lua offre d'ailleurs plusieurs méthodes pour les instancier :

```
-- Soit une matrice de 10 par 10
-- Avec des tableaux imbriqués (c'est-à-dire une ligne = une table)
local matrix = {}
for i = 1, 10 do
  matrix[i] = {}
  for j = 1, 10 do
    matrix[i][j] = 0
  end
end

-- Avec une multiplication d'indices
local matrix = {}
for i = 1, 10 do
  for j = 1, 10 do
    matrix[i * 10 + j] = 0
  end
end

-- Avec des indices de type string (comme une hashtable)
local matrix = {}
for i = 1, 10 do
  for j = 1, 10 do
    matrix[i .. ";" .. j] = 0 -- Ou matrix[i .. "," .. j], etc.
  end
end
```

68. Le module *table*

Outre l'accès direct aux tables, la librairie interne de Lua intègre un module `table` qui permet d'effectuer des actions complémentaires ou de simplifier l'écriture en résumant sous forme de fonctions ce qui nécessiterait plusieurs lignes de code à réaliser. Le module travaille sur des tables de type séquence, c'est-à-dire indexées numériquement de 1 à *n*. Il offre les méthodes suivantes :

- `table.concat` : concaténation des éléments d'une séquence dans une chaîne ;
- `table.insert` : insertion d'une valeur ;
- `table.remove` : suppression d'une valeur ;
- `table.pack` : construction d'une table à partir d'une liste ;

- `table.unpack` : retourne une liste de valeurs issues d'une séquence ;
- `table.sort` : tri d'une séquence.

69. La fonction *table.concat*

`table.concat(table_in [, separator [, istart [, iend]]])` renvoie une chaîne contenant la concaténation des éléments de `table_in` entre les indices `istart` et `iend`, chaque élément étant séparé par `separator`. Par défaut, `separator` est la chaîne vide `""`, `istart` vaut 1 et `iend` est égal à la taille de `table_in`.

```
-- Une table de type séquence
local tab = { 8, 12, 7, "U", 90 }
-- Génération d'un ligne de type CSV (Comma Separated Values)
print(table.concat(tab, ";")) --> 8;12;7;U;90
```

70. Insérer une valeur dans une séquence

`table.insert(table_in[, index], value)` insère l'élément `value` à la position `index`, en décalant si nécessaire les indices suivants. Si `index` est omis, alors `value` est insérée à la fin de la table.

```
-- Une table de type séquence
local tab = { 8, 12, 7, "U", 90 }
-- Insertion d'une valeur en position 3
table.insert(tab, 3, "NEW VALUE")
print(#tab, tab[#tab], tab[3]) --> 6  90  NEW VALUE
```

Attention *› Il y a un piège avec la syntaxe courante* `table.insert(t, fct() )`*. Si la fonction* `fct` *retourne plus d'une valeur, la première sera prise comme l'indice de position, ce qui n'est généralement pas désiré. Pour éviter cela, il faut ajouter une paire de parenthèses :* `table.insert(t, (fct()) )`*.*

71. Supprimer une valeur dans une séquence

`table.remove(table_in, [index])` supprime la cellule `index` en effectuant si nécessaire un décalage des indices restants. Par défaut, `index` vaut `#table`, ce qui signifie que l'appel `table.remove(table_in)` supprime le dernier élément de la table. La valeur supprimée est retournée.

```
-- Une table de type séquence
local tab = { 8, 12, 7, "U", 90 }
-- Suppression de l'élément en position 3 (valeur 7) et
```

```
-- décalage de l'élément 4 en position 3 suite à sa suppression
local deleted_value = table.remove(tab, 3)
print(deleted_value, #tab, tab[3]) --> 7   4   U
```

72. Créer un tableau depuis une liste de valeurs

La fonction `table.pack(...)` est nouvelle dans 5.2 et retourne une table contenant les paramètres passés indexés numériquement de 1 à *n*. La table contient de plus le champ `n` qui contient le nombre de paramètres reçus.

```
-- Mise en table de paramètres
local function print_to_tab(...)
  return table.pack(...)
end

-- Appel de la fonction
local mytab = print_to_tab("UN MESSAGE",
  "DE L'ESPACE", "42")
print(mytab[1], mytab.n) --> UN MESSAGE  3
```

73. Extraire une liste de valeurs d'une séquence

`table.unpack(table_in [, istart [, iend]])` retourne sous forme de liste de valeurs le contenu de `table_in` entre les bornes d'index `istart` et `iend`. Par défaut `istart` vaut 1 et `iend` vaut `#table_in`.

```
-- Une table de type séquence
local mytab = { 8, 12, 7, "U", 90 }

-- Liste des valeurs à partir de l'index 2
print(table.unpack(mytab, 2)) --> 12  7  U  90
```

Note *› Dans Lua 5.1, la même fonction ne faisait pas partie du module* `table` *et s'appelait* `unpack`.

74. Trier une séquence

`table.sort(table_in [, compfunc])` trie l'ensemble de `table_in`, selon la fonction de comparaison `compfunc(val1, val2)`. La fonction `compfunc` retourne `true` si `val1` se trouve avant `val2` dans le tableau trié, `false` sinon. Si `compfunc` n'est pas spécifiée, alors c'est l'opérateur d'infériorité qui est utilisé. Le tri est effectué *in situ*, c'est-à-dire que `table_in` est modifiée directement.

```lua
-- Une table de type séquence
local mytab = { 8, 12, 7, 90 }

-- Tri croissant
table.sort(mytab)
print(table.unpack(mytab)) --> 7  8  12  90

-- Tri décroissant
table.sort(mytab, function (v1,v2) return v1 > v2 end)
print(table.unpack(mytab)) --> 90  12  8  7
```