21

Le module LFS et ses utilisations

Comme on l'a vu au chapitre précédent, le module io a été volontairement restreint pour avoir une librairie à portabilité maximale, se limitant pour l'essentiel (à part `io.popen`) aux facilités offertes par le C ANSI. Cela permet de l'utiliser sur la plupart des systèmes où Lua est présent, des classiques Windows, Macintosh, Linux aux systèmes embarqués et téléphones portables.

Mais cela interdit certaines opérations de base, très courantes, comme lister les fichiers du système ou avoir leurs attributs (comme "lecture seule" ou "exécutable"). Ces opérations sont, en effet, très spécifiques au système sous-jacent, avec des séparateurs de chemins différents (certains systèmes n'ayant même pas de hiérarchie de fichiers), des attributs très différents, et ainsi de suite.

Heureusement, des librairies supplémentaires offrent ces facilités, au moins pour la trilogie de systèmes ci-dessus, ayant un certain nombre de points communs. Si nécessaire, les utilisateurs avancés peuvent adapter ces librairies à source libre pour les spécificités de leur système.

Nous parlons ici de LFS (LuaFileSystem), une librairie légère et simple, faisant partie du projet Kepler. Ce dernier semble malheureusement en suspens, à ce jour, LFS n'a pas été mis à jour pour Lua 5.2, mais il est assez facile de la modifier et de la compiler pour Lua 5.2.

78. Lister les fichiers d'un répertoire

LFS offre un itérateur sur les fichiers d'un chemin donné, ce qui permet d'utiliser une boucle `for` pour traiter chaque fichier.

```
local lfs = require"lfs"

for file in lfs.dir("C:/") do
  print(file)
end
```

Note *› LFS est disponible sous forme de librairie binaire partagée : `.dll` sous Windows, `.so` sous Linux, `.dylib` pour Macintosh. La ligne `require"lfs"`, que nous ne répéterons pas pour chaque exemple, permet de charger la librairie dans l'interpréteur Lua et d'utiliser ses fonctions. L'affectation à une variable est redondant en Lua 5.1, nécessaire en 5.2.*

Le paramètre de `lfs.dir()` est un chemin (dépendant donc du système) pointant sur le répertoire que nous voulons lister. Ce chemin peut être relatif (à l'emplacement du script ou du chemin courant, voir plus loin) ou absolu. Un simple point `"."` réfère au répertoire courant, `".."` remonte d'un niveau dans la hiérarchie de fichiers, ces deux symboles fonctionnent sur les trois systèmes.

79. Lister les fichiers et les sous-répertoires d'un répertoire

Le code ci-dessus liste aussi bien les fichiers que les répertoires. Si on sait distinguer les deux types, on peut alors lister le contenu des répertoires, puis des répertoires qu'on y trouve, etc.

`lfs.attributes(filePath [, attributeName])` permet de récupérer la liste des attributs d'un fichier sous forme de table, ou si `attributeName` est précisé, il ne récupère que la valeur de cet attribut. Les attributs retournés sont :

- `mode` : une chaîne représentant le mode de protection associé. Les valeurs peuvent être : `file`, `directory`, `link`, `socket`, `named pipe`, `char device`, `block device` ou `other` (dépendant du système, évidemment).
- `access` : la date du dernier accès au fichier, sous forme de *timestamp*, c'est-à-dire le nombre de secondes écoulées depuis une date de référence pour le système sous-jacent. On peut utiliser `os.date()` pour mettre cette date dans un format plus familier.
- `modification` : la date (cf. `access`) de dernière modification du fichier.
- `change` : la date (cf. `access`) du dernier changement de statut du fichier, ce qui correspond généralement à la date de création du fichier.
- `size` : la taille du fichier, en octets.
- `ino` : la numéro d'i-node. N'a de sens que sous Unix, vaut 0 sous Windows.
- `dev` : sous Unix, représente l'unité physique sur laquelle l'i-node réside. Sous Windows, représente le numéro du disque (ou de la partition) contenant le fichier.
- `rdev` sous Unix, représente le type d'unité, pour les i-nodes spéciaux. Sous Windows, a la même valeur que `dev`.

- `nlink` : le nombre de "hard links" (liens physiques) sur le fichier. Généralement 1 sous Windows.
- `blocks` : le nombre de blocs alloués pour ce fichier (Unix ; `nil` sous Windows).
- `blksize` : la taille optimale du bloc pour E/S système (Unix ; `nil` sous Windows).
- `uid` : l'identifiant de l'utilisateur possédant le fichier sous Unix, toujours 0 sous Windows.
- `gid` : l'identifiant du groupe de l'utilisateur possédant le fichier sous Unix, toujours 0 sous Windows.

On peut donc utiliser `lfs.attributes` avec le paramètre `mode` pour distinguer les fichiers des répertoires. Avec une méthode récursive, il est facile d'itérer sur toute la hiérarchie de fichiers :

```
function listDirectory(dirToList)
  local dirListing = {}
  for file in lfs.dir(dirToList) do
    -- Saute les répertoires spéciaux courant et parent
    if file ~= ".." and file ~= "." then
      local filePath = dirToList .. file
      local fileAttr = lfs.attributes(filePath, "mode")
      if fileAttr == "directory" then
        -- Appel récursif
        dirListing[file] = listDirectory(filePath .. '/')
      else
        -- Crée une entrée avec le fichier comme clé et la
        -- taille du fichier comme valeur
        dirListing[file] = lfs.attributes(filePath, "size")
      end
    end
  end
  return dirListing
end
-- Le chemin doit se terminer avec une barre oblique
local list = listDirectory("H:/Temp/")
```

À la fin, nous obtenons un tableau avec les noms comme clés, et pour valeurs la taille du fichier si la clé représente un fichier, ou un sous-tableau avec les fichiers du répertoire pour les clés de répertoire.

80. Manipuler des répertoires

LFS offre des fonctions pour créer, supprimer un répertoire, changer de répertoire ou savoir où l'on est.

- `lfs.currentdir()` retourne le répertoire courant, qui est généralement, par défaut, l'emplacement du script exécuté.
- `lfs.chdir(path)` change le répertoire courant vers le chemin donné. Ce dernier peut être relatif ou absolu et peut être, sous Windows, sur un disque (ou partition) différent. Retourne `true` en cas de succès, ou `nil` et un message d'erreur si le chemin n'existe pas ou n'est pas accessible.
- `lfs.mkdir(path)` crée un nouveau répertoire dans le chemin donné. Ce dernier doit exister, ce qui veut dire que pour créer une hiérarchie, il faut créer chaque niveau un par un. Retourne `true` en cas de succès, ou `nil` et un message d'erreur si le chemin ne peut pas être créé.
- `lfs.rmdir(path)` supprime le répertoire indiqué. Ce dernier doit être vide. Pour mémoire, `os.remove()` peut être utilisé pour supprimer un fichier. Retourne `true` en cas de succès, ou sinon `nil` et un message d'erreur.

81. Manipuler des fichiers

LFS offre quelques fonctions pour manipuler des fichiers, en plus de celles trouvées dans la librairie os de Lua.

- `lfs.touch(filePath [, accessTime [, modifiedTime])` change, par défaut, la date de modification du fichier à la date courante du système. Cela change aussi la date du dernier accès. On peut spécifier la date d'accès, qui sera utilisée pour la date de modification, ou les deux dates. Les dates sont exprimées en secondes (cf. plus haut). Le fichier doit exister.
- `lfs.setmode(file, mode)`, où `file` est un descripteur de fichier provenant de `io.open`, change le mode de lecture du fichier. Le mode peut être `"binary"` ou `"text"`. N'est disponible que sous Windows, il faut tester si la fonction est disponible avant de s'en servir.
- `lfs.symlinkattributes(filePath [, attributeName])` est identique à `lfs.attributes()` mais sur un lien symbolique, s'applique au lien et non au fichier lié. N'est disponible que sous Unix, il faut tester si la fonction est disponible avant de s'en servir.

82. Verrouiller répertoires et fichiers

On peut vouloir verrouiller un répertoire ou un fichier, pour interdire l'accès par un autre programme ou un autre fil d'exécution (*thread*).

- `lfs.lock_dir(path)` crée un fichier de verrouillage (nommé `lockfile.lfs`) dans le chemin donné, si le fichier n'existe pas encore, et retourne le verrou. En cas d'erreur, par exemple si le fichier existe déjà, retourne `nil` et un message d'erreur, qui est `"File exists"` quand le fichier existe. Le verrou est enlevé en appelant `lock:free()` où `lock` est le verrou retourné.
- `lfs.lock(fileHandle, mode [, start[, length]])` verrouille tout ou partie d'un fichier déjà ouvert. Le mode peut être `"r"` (pour un verrou en lecture ou un verrou partagé) ou `"w"` (pour un verrou en écriture ou exclusif). Les autres paramètres spécifient éventuellement la partie du fichier à verrouiller (portée en octets).
- `lfs.unlock(fileHandle [, start[, length]])` déverrouille le fichier ou une partie.