6

Écrivez des modules complets en C, embarquez du Lua

Dans ce chapitre, nous allons interfacer une librairie tierce écrite en C pour qu'elle puisse être appelée depuis Lua (écrire un *binding*).

1. zlib

La librairie de compression *zlib* est utilisée dans un nombre extrêmement élevé d'applications et d'appareils électroniques. Elle est notamment à la base des formats de compression *ZIP* ainsi que *GZIP*. Il est fort probable que vous l'ayez déjà utilisée dans un développement et serez alors à l'aise avec son API.

Nous n'allons pas faire un *binding* exhaustif de toutes les fonctions de la librairie, nous nous concentrerons sur les plus utilisées : les algorithmes de contrôle d'intégrité `crc32` et `adler32`, les fonctions très simples d'emploi `compress2` et `uncompress`, ainsi que l'interface plus bas niveau utilisant `deflateInit`, `deflate`, `deflateEnd`, `inflateInit`, `inflate` et `inflateEnd`. Il manque à cette liste toute la gestion du format *GZIP* avec les fonctions préfixées par `gz`, ainsi que de nombreuses fonctions rarement utilisées visant à contrôler de manière plus fine le processus de compression. Vous pourrez ajouter certaines de ces fonctionnalités à titre d'exercice.

2. Les sommes de contrôle

Dans le code de *zlib* est inclus deux algorithmes de hachage pour contrôler l'intégrité des données. Le premier est le contrôle de redondance cyclique (CRC). Calculé sur 32 bits, c'est un classique dans le domaine du contrôle d'intégrité. Le deuxième algorithme s'appelle *Adler*, du nom de l'inventeur, et a été développé spécifiquement pour *zlib* en tant qu'alternative plus rapide au CRC. Disposer en Lua d'un algorithme de hachage peut s'avérer fort utile dans bien des situations.

La signature des fonctions de hachage est la suivante :

```
uLong adler32(uLong adler, const Bytef *buf, uInt len);
uLong crc32(uLong crc, const Bytef *buf, uInt len);
```

Ces deux fonctions prennent donc en argument une éventuelle valeur de hachage précédente à mettre à jour, un pointeur sur des octets de données et leur longueur. Quelle est alors l'interface la plus appropriée dans le langage Lua ? C'est une chaîne de caractères Lua qui représente le mieux un bloc d'octets, et le résultat est évidemment un nombre. Il serait aussi possible d'imaginer que les octets soient donnés sous forme d'une table de nombres ou une *userdata*, voire de supporter ces trois options. Cela montre que le *binding* n'est jamais un processus univoque, il faut opérer des choix.

En Lua, les chaînes connaissent leur longueur, il n'est donc pas nécessaire d'ajouter un argument de taille. Et comme le premier argument de `crc32` ou de `adler32` est en général 0, nous allons le mettre optionnel et en deuxième position. Toutes les fonctions de la librairie seront mises dans une table `zlib`, ce qui signifie que la signature correspondante en Lua sera :

```
adler`number = zlib.adler(data`string [, adler`number])
crc`number = zlib.crc(data`string [, crc`number])
```

Note *› Dans le pseudo-code ci-dessus, le caractère ` et le mot qui suit en italique représentent le type de données attendu, ils ne font aucunement partie de la syntaxe Lua.*

Voici donc le code C qui permet d'exporter les fonctions de hachage :

```
#include "zlib.h"
#include "lua.h"
#include "lauxlib.h"

static int cksum_common(lua_State* L, int defval,
    uLong (*cksum)(uLong, const Bytef*, uInt)) {
  size_t srclen;
  const Bytef* src=(const Bytef*)luaL_checklstring(L,1,&srclen);
  uLong val = luaL_optunsigned(L, 2, defval);
  val = cksum(val, src, srclen);
  lua_pushunsigned(L, val);
  return 1;
}

static int zlib_adler(lua_State* L) {
  return cksum_common(L, 1, adler32);
}
```

```
static int zlib_crc(lua_State* L) {
  return cksum_common(L, 0, crc32);
}

static const luaL_Reg zlib_fcts[] = {
  { "crc",        zlib_crc },
  { "adler",      zlib_adler },
  { NULL, NULL }
};

int luaopen_zlib(lua_State* L) {
  luaL_newlib(L, zlib_fcts);
  lua_pushstring(L, ZLIB_VERSION);
  lua_setfield(L, -2, "_VERSION");
  return 1;
}
```

Les deux en-têtes `lua.h` et `lauxlib.h` sont ceux généralement nécessaires pour un *binding* vers Lua ; l'inclusion de `zlib.h` est quant à elle bien évidemment requise pour déclarer les fonctions de *zlib*.

Une fonction C appelable depuis Lua doit obligatoirement avoir le type `lua_CFunction`, c'est-à-dire avoir la signature :

```
int fonction_C_pour_Lua(lua_State* L);
```

C'est le cas des fonctions `zlib_adler` et `zlib_crc`, qui pourront être appelées depuis Lua comme `zlib.adler` et `zlib.crc` respectivement, grâce à la table de correspondance `zlib_fcts`. Étant donné que les deux fonctions ont une interface identique, nous avons regroupé leur code de liaison dans une seule fonction `cksum_common`, qui prend en argument, outre l'état Lua, la valeur pour l'initialisation et un pointeur vers une fonction de calcul.

La première tâche de `cksum_common` est de récupérer depuis la pile Lua les arguments passés à la fonction. La fonction `luaL_checklstring` permet d'obtenir un pointeur sur la chaîne de caractères à hacher ainsi que sa longueur. Si le premier argument passé n'est pas une chaîne, `luaL_checklstring` lance une erreur. Au contraire du deuxième argument qui peut être un nombre ou `nil` ; dans ce dernier cas `luaL_optunsigned` retournera 0. Le traitement proprement dit se fait lors de l'appel à la fonction pointée par `cksum`. Pour retourner la valeur calculée à Lua, il faut le placer en haut de la pile avec `lua_pushunsigned` et signaler avec `return 1` que la fonction a produit 1 résultat.

La fonction `luaopen_zlib`, qui a aussi la signature `lua_CFunction`, sert à initialiser le module `zlib`. La fonction auxiliaire `luaL_newlib` va créer une nouvelle table sur la pile,

et y ajouter toutes les fonctions de la table `zlib_fcts` jusqu'à tomber sur la ligne de terminaison obligatoire `{ NULL, NULL }`. Il est d'usage courant dans un module en Lua d'ajouter un champ `_VERSION` qui contient le numéro de version du module, sous forme de chaîne de caractères. Lors de l'édition de ces lignes, `zlib._VERSION` avait la valeur `"1.2.6"`.

Note *› Dans cet exemple, seule la fonction `luaopen_zlib` n'est pas déclarée `static` car elle doit être appelée depuis l'extérieur. Toutes les autres sont référencées directement ou non depuis la table d'exportation. Ce n'est pas un cas isolé : il en est de même pour tous les modules de la librairie standard par exemple.*

3. Initialisation du module

Et comment doit-on appeler `luaopen_zlib` pour initialiser le module ? Il existe au moins ces trois possibilités :

- Éditer le fichier `linit.c` des sources Lua, ou en faire une copie modifiée. Ce fichier définit la fonction `luaL_openlibs` que l'on appelle généralement après la création d'un état Lua avec `luaL_newstate`. Dans ce fichier se trouve un tableau avec la liste des fonctions `luaopen_*` et le nom de la variable globale associée. Si vous avez la possibilité de recompiler Lua dans votre application, il suffit alors d'ajouter une ligne à ce tableau.
- Ajouter dans votre application un enregistrement manuel du module avec la fonction `luaL_requiref`, généralement juste après l'appel à `luaL_openlibs` :

```
lua_State *L = luaL_newstate();
luaL_openlibs(L);
luaL_requiref(L, "zlib", luaopen_zlib, 1);
```

- Compiler le module séparément dans une librairie partagée, avec une extension `.dll` sous Windows, `.so` sous Linux ou encore `.dylib` sous MacOS X. En plaçant la librairie dans le chemin déterminé par `package.cpath`, Lua pourra charger dynamiquement le module lorsqu'il rencontrera la syntaxe `zlib=require "zlib"`. C'est de loin la solution la plus compliquée, car générer une telle librairie dépend du système d'exploitation et du compilateur, mais elle a l'avantage de permettre à la librairie d'être distribuée plus facilement. Pour réaliser cela, il vaut mieux recopier et modifier le `Makefile` d'un autre module Lua, par exemple celui de *LuaFileSystem*.