LabLua
programming language research

Programming in Lua – Extending Lua

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Calling C functions

- There is a mountain of C libraries to do a lot of things: statistics, computer vision, database access, graphical user interfaces…
- The same API that lets us embed Lua in an application also lets us connect Lua scripts to C libraries
- To expose a library to Lua, we have to give it an *adapter* composed of a series of C functions that:
  - Marshall data from Lua
  - Call the C functions that do the heavy lifting of the library
  - Marshall back the results

The stack from Lua

- When Lua calls a function, the function gets its own *private* stack; as the stack in a fresh Lua state, this stack has space for 20 slots

- Unlike the stack in a fresh Lua state, this stack will be populated with the *arguments* for the function: index 1 is the first argument, index 2 is the second, and so on

lua_gettop() is # of arguments

- When a C function returns, it tells how many values should be popped from the stack as the return values of the function

return 2; → return two values

  - If the function returns 3, the top of the stack is the third returned value, the second value from the top is the second and the third value from the top is the first returned value

Writing a C function

- Any C function that we want to call from Lua must have this prototype:

```
typedef int (*lua_CFunction)(lua_State *L);
```

- The function receives the Lua state, and manipulates its stack, and returns how many return values there are in the top of the stack:

```
static int idiv(lua_State *L) {
    int n1 = lua_tointeger(L, 1); /* first argument */
    int n2 = lua_tointeger(L, 2); /* second argument */
    int q = n1 / n2; int r = n1 % n2;
    lua_pushinteger(L, q);        /* first return value */
    lua_pushinteger(L, r);        /* second return value */
    return 2;                     /* return two values */
}
```

- We can test the function by adding two lines to our simple REPL, just after opening the standard libraries:

```
lua_pushcfunction(L, idiv);
lua_setglobal(L, "idiv");
```

```
> print(idiv(11, 3))
3       2
```

Defensive programming

- The `idiv` function is unsafe: if we pass 0 as the second argument it crashes the REPL! We can raise a Lua error when that happens, with the `luaL_error` function:

```
if(n2 == 0) return luaL_error(L, "division by zero");
```

- This function never returns, but the C compiler does not know that, so we return after calling it

- As it is, the function accepts any Lua values as arguments, not just numbers, because `lua_tonumber` will convert anything else to 0

- We can be stricter if we use `luaL_checkinteger` instead of `lua_tointeger`:

```
> print(idiv({}, 5))
[string "print(idiv({}, 5))..."]:1: bad argument #1 to 'idiv' (number expected,
got table)
```

```
int n1 = luaL_checkinteger(L, 1);
int n2 = luaL_checkinteger(L, 2);
```

C modules

- A Lua module is a script that returns the module's table; a *C module* is a *dynamic library* that exports a function that does the same thing

- C modules have a different search path, stored in `package.cpath` (and coming from `LUA_CPATH_5_2` or `LUA_CPATH` environment variables, if present):

```
> print(package.cpath)
/usr/local/lib/lua/5.2/?.so;/usr/local/lib/lua/5.2/loadall.so;./?.so
```

- Lua will search for C modules if it has not found a Lua module; once Lua finds the module's .so file, it will try to run a function `luaopen_`*`modulename`*

- The module name is the argument passed to require, replacing dots with underscores, so `require "lib.mod"` will load `lib/mod.so` and run `luaopen_lib_mod`

luaL_newlib

- The luaL_newlib function helps in packaging a C module; it will take an array of name/function pairs and create and populate a table with these functions, leaving the table in the top of the stack:

```
static const struct luaL_Reg mylib[] = {
    {"idiv", idiv},
    {NULL, NULL}
};

int luaopen_mylib(lua_State *L) {
    luaL_newlib(L, mylib);
    return 1;
}
```

```
> lib = require "mylib"
> print(lib.idiv(11,3))
3       2
```

- luaL_newlib is just a convenience, the following would create and return the module in the same way:

```
int luaopen_mylib(lua_State *L) {
    lua_newtable(L);
    lua_pushcfunction(L, idiv);
    lua_setfield(L, -2, "idiv");
    return 1;
}
```

Array manipulation

- The C API has two functions for manipulating arrays:

```
void lua_rawgeti(lua_State *L, int index, int key);
void lua_rawseti(lua_State *L, int index, int key);
```

- They are similar to `lua_getfield` and `lua_setfield`, but the `raw` prefix means that they ignore metatables

- `lua_rawlen` takes the Lua state and the stack index of a table (or a string) and returns its length (#), again ignoring metatables; `lua_len` uses a metatable if present, but pushes the result instead of returning it

- In general, ignoring metatables on array manipulation is not a problem, as arrays are a “low-level” data structure, and we want their operations to be quick

map in C

- We can use the array functions to write a version of `map` in C:

```
static int map(lua_State *L) {
    int i, n;
    luaL_checktype(L, 1, LUA_TFUNCTION); /* f */
    luaL_checktype(L, 2, LUA_TTABLE);    /* t */
    n = lua_rawlen(L, 2);                /* #t */
    lua_newtable(L);                     /* new_t */
    for(i = 1; i <= n; i++) {
        lua_pushvalue(L, 1);             /* push f */
        lua_rawgeti(L, 2, i);            /* push t[i] */
        lua_call(L, 1, 1);               /* f(t[i]) */
        lua_rawseti(L, -2, i);           /* pop new_t[i] */
    }
    return 1;
}
```

- `lua_call` is like `lua_pcall`, but propagates errors instead of catching them, so it has no need for a fourth argument (the optional error handler in `lua_pcall`)

String manipulation

- `lua_pushlstring` makes a copy of the string, so we can use C “tricks” with the pointer and the size to push a substring of a string:

```
/* pushes the substring of s from i to j (inclusive) */
lua_pushlstring(L, s + i, j - i + 1);
```

- `lua_concat(L, n)` pops `n` strings from the stack and concatenates them (triggering `__concat` metamethods if necessary)

- `lua_pushfstring(L, fmt, …)` is like sprintf, but Lua allocates the buffer for the resulting string, and returns it after pushing the resulting string; Lua owns this buffer, so do not change it, and make a copy if you want it to survive the corresponding string

String buffers

- For more complicated efficient string creation in C functions, Lua provides an API for *string buffers*:

```
/* initializes a buffer, stack allocate it */
void luaL_buffinit  (lua_State *L, luaL_Buffer *B);
/* pop a string and add it to the buffer */
void luaL_addvalue  (luaL_Buffer *B);
/* add C strings to the buffer */
void luaL_addlstring(luaL_Buffer *B, const char *s, size_t l);
void luaL_addstring (luaL_Buffer *B, const char *s);
/* add a character to the buffer */
void luaL_addchar   (luaL_Buffer *B, char c);
/* create a Lua string with the buffer contents and push it */
void luaL_pushresult(luaL_Buffer *B);
```

- The functions in the buffer API keep temporary state in the Lua stack, so the effect of operations you do on the stack must be neutral between calls to buffer functions

The registry

- The registry is a table shared by all C modules, kept in a special `LUA_REGISTRYINDEX` stack *pseudo-index* (do not worry, there is no risk of collision with regular stack indexes)

- It is a regular table, so you can query and update it with the table functions: `lua_getfield`, `lua_setfield`, `lua_gettable`, `lua_settable`

- As this table is shared, you have to pick key names carefully, so it will not collide with other C modules; a good practice is to prefix the key name with the fully qualified module name (the same one you append to `luaopen_`)

- Numeric keys are reserved for *references*, we will see them shortly

References

- We can only take atomic values out of the stack, but we can “take” other values out of the stack if we use *references*:

```
int ref = luaL_ref(L, LUA_REGISTRYINDEX);
```

- luaL_ref pops a value and returns a *reference* to it

- A reference is just an index in the registry, so we can get the push the value back (in a different function, or even a different module) with `lua_rawgeti(L, LUA_REGISTRYINDEX, ref)`

- When we are done with the reference, we can free it with `luaL_unref(L, LUA_REGISTRYINDEX, ref)`

- You can get a reference to any value, even `nil`

C closures

- There is just one registry for all the C modules in the system; if you need to store state local to a single C function, or a single C module, it is best if you use *C closures*

- If you use lua_pushcclosure to push a C function, you can pass a third argument: how many values lua_pushcclosure should pop and put in the closure

- Inside the function, `lua_upvalueindex(i)` returns a pseudo-index for the i-th element in the closure

- A non-existing closure element has a pseudo-index with type LUA_TNONE

Sharing upvalues

- Each C closure has its own set of upvalues, but we can “share” an upvalue among several closures if we close over a table

- We can use this technique to have a “private registry” among all functions in a C module:

```
int luaopen_mylib(lua_State *L) {
    lua_newtable(L); /* module */
    lua_newtable(L); /* shared table */
    /* populate the module with functions, using the 1 upvalue for each */
    luaL_setfuncs(L, mylib, 1);
    /* the module is in the top now */
    return 1;
}
```

Quiz

- A library is trying to share a counter between inc and dec functions that decrement the counter using an upvalue, with the initialization function:

```
int luaopen_counter(lua_State *L) {
    lua_newtable(L);
    lua_pushinteger(L, 0);
    lua_setfuncs(L, mylib, 1);
    return 1;
}
```

- What is going to happen?

EACH FUNCTION HAS A DIFFERENT COUNTER!