Kurs języka Lua

07 – Wstęp do C API

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Podstawach C API

Uwaga

Za tydzień nie ma wykładu ani pracowni
(deadline na kolejną listę jest 24-25.04)

Podstawy C API

Lua: *embeddable* i *extensible*

Lua jako język osadzalny

- Lua nigdy nie jest *standalone* – nawet interpreter to po prostu osadzenie Lua w `lua.c`
- W tym scenariuszu C ma kontrolę, a Lua jest biblioteką.
- Możemy tworzyć wirtualne maszyny Lua, zapisywać i odczytywać ich zmienne globalne, wywoływać funkcje Lua, fragmenty kodu, itd.

Lua jako język rozszerzalny

- Mamy również interakcję w drugą stronę
- Lua przejmuje kontrolę a C jest biblioteką (Lua wciąż pozostaje biblioteką interpretera `lua`).
- Możemy wtedy rejestrować funkcje C w Lua i wywoływać je w Lua

Nagłówki

- *lua.h* (funkcje `lua_`)
  – Lua API
- *luaxlib.h* (funkcje `luaL_`)
  – Funkcje pomocnicze zbudowane na bazie Lua API
- *lualib.h* (funkcje `luaL_`)
  – Funkcje otwierające biblioteki standardowe

Lua w C++

- Lua nie wykrywa czy jest kompilowana jako C++
- Lua poprawnie kompiluje się jako C++, ale wymaga `extern`
- zazwyczaj załącza się po prostu `lua.hpp`:

```
// lua.hpp
extern "C" {
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
}
```

Przerwa na nadrobienie braków

- `pcall (f [, arg1, ...])` uruchamia funkcję `f` w trybie chronionym, nie propagując powstałych w niej błędów
- zwraca `true` i wynik działania funkcji, lub `false` i komunikat błędu

```
function f (x) return x+5 end
f(nil) --> tests.lua:386: attempt to perform ...
print (pcall (f, nil))
--> false  tests.lua:386: attempt to perform ...
print (pcall (f, 7))
--> true  12
```

- `load` (kiedyś `loadstring`) ładuje zadany napisem chunk kodu.

```
f = load('i = i + 1')
i=0
f(); print (i) --> 1
f(); print (i) --> 2
```

Prosty interpreter Lua

example01.cpp

- tworzymy nowy stan Lua, ładujemy biblioteki standardowe
- wczytujemy z konsoli kolejne fragmenty (chunks) kodu Lua
- stdout jest automatycznie przekierowywany na konsolę C
- w przypadku błędu wypisujemy jego komunikat

Obsługa błędów

- bywa problematyczna...
- funkcje API mogą zawsze zawieść (ze względu na dynamiczne zarządzanie pamięcią)
- funkcje `luaL_newstate`, `lua_load`, `lua_pcall`, `lua_close` są bezpieczne
- `lua_error` i `luaL_error` pozwalają generować błędy w Lua

Prosta funkcja do obsługi błędów

```
void error (lua_State *L, const char *fmt, ...)
{
  va_list argp;
  va_start(argp, fmt);
  vfprintf(stderr, fmt, argp);
  va_end(argp);
  lua_close(L);
  exit(EXIT_FAILURE);
} // error(L, "'%s' should be a number\n", var);
```

Lua stack

Problemy na styku Lua i C

- ręczne/automatyczne zarządzanie pamięcią
- dynamiczne/statyczne typowanie
- zwracanie i przekazywanie wielu wartości

Stos

- Wykorzystywany do komunikacji (przekazywania wartości) pomiędzy C a Lua
- Przechowuje dane Lua
- Funkcje C API pobierają ze stosu dane których potrzebują i odkładają tam swoje rezultaty
- Z punktu widzenia Lua stos jest kolejką LIFO
- Z poziomu C możemy nim manipulować praktycznie dowolnie
- Nie jest taki straszny jak się wydaje ;-)

Lua stack

Stos

- Każdy stan Lua ma swój stos
- Naszą odpowiedzialnością jest dbanie o odpowiednią ilość miejsca
- Domyślna wielkość stosu to 20, możemy ją modyfikować:

```
sucess = lua_checkstack(L, 50); // 0 = failure
```

- Stos możemy indeksować od dołu: 1, 2, 3, ...
- lub od góry: -1, -2, -3, ...

Przykład

-1	'hello'	3
-2	4	2
-3	nil	1
	$\perp$	

Wstawianie na stos

```
void lua_pushnil (lua_State *L);

// 0 to fałsz, wszystko inne true:
void lua_pushboolean (lua_State *L, int bool);

// typedef double lua_Number;
void lua_pushnumber (lua_State *L, lua_Number n);
// typedef long long lua_Integer;
void lua_pushinteger (lua_State *L, lua_Integer);

// kopiuje napis - pamięć można zwolnić
const char * lua_pushlstring (lua_State *L,
  const char *s, size_t len);
// tylko dla 0-terminated (pushlstring z strlen)
const char * lua_pushstring (lua_State *L,
  const char *s);
```

Sprawdzanie typu

Funkcje zwracają 1 jeśli wartość w danym miejscu stosu jest **konwertowalna** do typu zapytania; w przeciwnym wypadku 0.

```
int lua_isnil (lua_State *L, int index);
int lua_isboolean (lua_State *L, int index);
// number or number as string (!)
int lua_isnumber (lua_State *L, int index);
// string or number (!)
int lua_isstring (lua_State *L, int index);
int lua_istable (lua_State *L, int index);
int lua_isfunction (lua_State *L, int index);
```

Zwraca kod typu (stała zdefiniowana w `lua.h`)

```
int lua_type (lua_State *L, int index);
LUA_TNIL (0), LUA_TBOOLEAN (1), LUA_TNUMBER (3),
LUA_TSTRING (4), LUA_TTABLE (5), LUA_TFUNCTION (6),
LUA_TLIGHTUSERDATA (2), ...
```

Pobieranie ze stosu

```
// zwraca 0 dla false i nil, wpp.\ 1
int lua_toboolean (lua_State *L, int index);

// dla nieprawidłowych wartości zwracają 0:
lua_Number lua_tonumber(lua_State *L,int index);
lua_Integer lua_tointeger(lua_State *L,int index);

// Dla nieprawidłowych wartości zwracają NULL:
lua_State *lua_tothread(lua_State *L,int index);
const char *lua_tolstring(lua_State *L,int index,
                          size_t *len);
```

- Powyższe funkcje sczytują dane ze stosu ale ich nie ściągają
- Wczytywanie napisów zwraca wskaźnik na wewnętrzną kopię napisu: wskaźnik jest ważny tak długo jak wartość ta jest na stosie!

Pobieranie ze stosu

```
lua_Number lua_tonumberx (lua_State *L,
                          int index, int *isnum);
lua_Integer lua_tointegerx (lua_State *L,
                            int index, int *isnum);
```

- Jeśli `isnum` jest różne od `NULL`, zawiera informację boolowską o tym czy operacja konwersji się powiodła

```
size_t len;
const char *s = lua_tolstring(L, -1, &len);
assert(s[len] == '\0');
assert(strlen(s) <= len);
```

- Napis zwracany przez `tolstring` zawsze kończy się zerem, ale może mieć też zero na końcu
- Prawdziwa długość jest trzymana w `len`

```
#def lua_tostring(L,i) lua_tolstring(L, i, NULL)
```

Drukowanie stanu stosu

example02.cpp → `void stackDump(lua_State *L)`

Manipulacja stosem

- Zwraca liczbę elementów na stosie:

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

- Ustawia wysokość stosu
  (usuwa nadmiarowe wartości, niedomiar dopełnia `nil`ami):

```
void lua_settop (lua_State *L, int index);
lua_settop(L,0); // czyści stos
```

- Pobiera `n` elementów z wierzchu stosu:

```
#define lua_pop(L,n) lua_settop(L,-(n) - 1)
```

- Kładzie na stos kopię elementu o zadanym indeksie:

```
void lua_pushvalue (lua_State *L, int index);
```

Wszystkie indeksy mogą być zarówno pozytywne jak i negatywne.

Manipulacja stosem

- Obraca elementy na stosie od zadanego indeksu o `n` pozycji (5.3):

```
int lua_rotate (lua_State *L, int index, int n);
```

- Usuwa element na zadanej pozycji odpowiednio przesuwając resztę:

```
#define lua_remove(L,idx) \
  (lua_rotate(L, (idx), -1), lua_pop(L, 1))
```

- Przesuwa wierzchni element na zadaną pozycję (przesuwając resztę):

```
#define lua_insert(L,idx) lua_rotate(L, (idx), 1)
```

- Sciąga wartość i wstawia na zadaną pozycję (bez przesunięć):

```
void lua_replace (lua_State *L, int index);
```

- kopiuje wartość z jednego indeksu na drugi (Lua 5.2)
  (oryginał nie jest modyfikowany):

```
void lua_copy(lua_State *L,int fromidx,int toidx);
```

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);




lua_close(L);
```

Manipulacja stosem

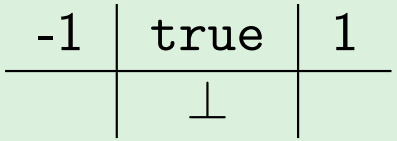
example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);




lua_close(L);
```



Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
   luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);




lua_close(L);
```

-1	10	2
-2	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");




lua_close(L);
```

-1	nil	3
-2	10	2
-3	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);




lua_close(L);
```

-1	"hello"	4
-2	nil	3
-3	10	2
-4	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);



lua_close(L);
```

-1	true	5
-2	"hello"	4
-3	nil	3
-4	10	2
-5	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);



lua_close(L);
```

-1	"hello"	4
-2	true	3
-3	10	2
-4	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);
lua_rotate(L, 3, 1);


lua_close(L);
```

-1	nil	6
-2	nil	5
-3	"hello"	4
-4	true	3
-5	10	2
-6	true	1
	$\perp$	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);
lua_rotate(L, 3, 1);
lua_remove(L, -3);

lua_close(L);
```

-1	nil	6
-2	"hello"	5
-3	true	4
-4	nil	3
-5	10	2
-6	true	1
	$\perp$	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);
lua_rotate(L, 3, 1);
lua_remove(L, -3);
lua_settop(L, -5);
lua_close(L);
```

-1	nil	5
-2	"hello"	4
-3	nil	3
-4	10	2
-5	true	1
	⊥	

Manipulacja stosem

example02.cpp

Stos

```
lua_State *L =
  luaL_newstate();
lua_pushboolean(L, 1);
lua_pushnumber(L, 10);
lua_pushnil(L);
lua_pushstring(L, "hello");
lua_pushvalue(L, -4);
lua_replace(L, 3);
lua_settop(L, 6);
lua_rotate(L, 3, 1);
lua_remove(L, -3);
lua_settop(L, -5);
lua_close(L);
```

-1	true	1
	⊥	

Bezpieczne wywołanie funkcji

- Wywołując funkcje C w Lua jesteśmy narażeni na błędy C
- W przypadku błędu API Lua wywołuje `lua_atpanic` i zamyka aplikację.
- Aby nadać błędom konteks, funkcję należy wywoływać poprzez Lua, w zabezpieczonym środowisku:

```
static int foo (lua_State *L)
{
  <chroniony kod>
  return 0;
}
int secure_foo (lua_State *L)
{
  // wstawiamy f jako funkcję Lua
  lua_pushcfunction(L, foo);
  // i bezpiecznie wywołujemy
  return (lua_pcall(L, 0, 0, 0) == 0);
}
```

Alokacja pamięci

- Lua alokuje pamięć korzystając z jednej funkcji alokującej
- Funkcję tę mamy możliwośc podać tworząc nowy stan Lua

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

- Parametrami funkcja alokującej `f` (poza userdata), są:
  - adres (re) alokowanego bloku
  - oryginalny rozmiar bloku (debug info jeśli `ptr==NULL`)
  - żądany rozmiar bloku
- zwraca `NULL` lub nowy adres

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

- (wartość `NULL` w `size` traktowana jest jak 0)

Alokacja pamięci

Standardowa funkcja alokująca (luaxlib.c)

```
static void *l_alloc (void *ud, void *ptr,
                      size_t osize, size_t nsize)
{
  (void)ud;  (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

- zakłada, że `free(NULL)` nic nie robi
- oraz `realloc(NULL, size)` działa jak `malloc(size)`

Rozszerzanie aplikacji

Lua jako język konfiguracyjny

```
-- define window size
width  = 200
height = 300
```

- W Lua możemy wygodnie pisać pliki konfiguracyjne,
- z komentarzami,
- złożonymi typami danych
- i funkcjami opisującymi część logiki

```
local env = require 'UserEnvironment'
if env.get("DISPLAY") == ":0.0" then
  width = 300; height = 300
else
  width = 200; height = 200
end
```

Lua jako język konfiguracyjny

Wczytywane *quick and dirty* (ładnie: example03.cpp+example03.lua)

```
void load_cfg (lua_State *L, const char *fname,
               int *w, int *h)
{
  if (luaL_loadfile(L, fname)
      || lua_pcall(L, 0, 0, 0))
        error(L, "cannot load config: %s",
                 lua_tostring(L, -1));
  lua_getglobal(L, "width"); // wstawiamy na stos
  lua_getglobal(L, "height");// wstawiamy na stos
  if (!lua_isnumber(L, -2))
    error(L, "'width' should be a number\n");
  if (!lua_isnumber(L, -1))
    error(L, "'height' should be a number\n");
  *w = lua_tointeger(L, -2);
  *h = lua_tointeger(L, -1);
}
```

Tworzenie zmiennych globalnych

- Z poziomu C możemy tworzyć zmienne globalne Lua

```
void lua_setglobal(lua_State *L,const char *name);
```

- Pobieramy (pop) wartość ze szczytu stosu i ustawiamy jako zmienną globalną name

```
lua_pushstring(L, "0.1a");   // lua_gettop(L) -> 1
lua_setglobal(L, "version");// lua_gettop(L) -> 0
```

example04.cpp + example03.lua

Dziękuję za uwagę

Za **2 tygodnie**:
dziedziczenie,
rozszerzanie aplikacji c.d.,
uruchamianie C z Lua,
...?