Kurs języka Lua

09 – moduły C (i obiektowość, c.d.)

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Wywoływaniu C z Lua
2. Różnych sposobach na obiektowość

Wywoływanie C z Lua

Podstawowe zasady współpracy

- W Lua chcemy trzymać kod który dobrze byłoby modyfikować bez konieczności rekompilacji projektu
- oraz taki który w Lua będzie się pisało wygodniej (wysokopoziomowy)
- Z dobrodziejstw C chcemy korzystać jeśli kod ma być szybki
- lub chcemy użyć już napisanego kodu, np. z gotowych bibliotek

- Z poziomu Lua nie możemy wywołać dowolnej funkcji C – funkcja ta musi zostać prawidłowo opakowana w C API
- Co więcej, funkcja ta musi zostać odpowiednio zarejestrowana, tak aby Lua potrafiła ją wywołać
- Podobnie jak ma to miejsce w relacji Lua z C, czeka nas:
  - przetaczanie danych typu Lua,
  - wywołanie rzeczywistego działania funkcji,
  - przetaczanie wyników z powrotem.

Wywoływanie funkcji

- Każda funkcja C wywołana z poziomu Lua otrzymuje swój prywatny stos (wielkości 20)
- Stos ten zawiera argumenty funkcji: pierwszy argument na pozycji 1, drugi na pozycji 2, itd.
- Funkcja zwraca wynik podając ile wartości należy zabrać z góry stosu i traktować jako jej rezultaty.
- Dla funkcji zwracającej n rezultatów, n-ta zwrócona wartość to szczyt stosu, $n-1$ to pierwsza komórka pod szczytem, itd.
- Po zakończeniu działania wywołanej funkcji, stos tego wywołania jest czyszczony.

Definiowanie funkcji C

```c
static int l_sin (lua_State *L)
{
  double d = lua_tonumber(L, 1);
  lua_pushnumber(L, sin(d));
  return 1;
}
```

```c
static int l_idiv(lua_State *L)
{
  int n1 = lua_tointeger(L, 1);
  int n2 = lua_tointeger(L, 2);
  int q = n1 / n2; int r = n1 % n2;
  lua_pushinteger(L, q);
  lua_pushinteger(L, r);
  return 2;
}
```

Rejestrowanie funkcji C

- Każda funkcja rejestrowana w C musi mieć ten sam prototyp

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

- Jak zarejestrować funkcję?
- Na razie quick-and-dirty:

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

```
luaL_loadstring(L, "print (mysin(3.14))")
  || lua_pcall(L, 0, 0, 0); // 0.0015926529164868
luaL_loadstring(L, "print (idiv(11,3))")
  || lua_pcall(L, 0, 0, 0); // 3   2
```

Obsługa błędów

```
luaL_loadstring(L, "print (mysin('a'))")...
// -->
```

Obsługa błędów

```
luaL_loadstring(L, "print (mysin('a'))")...
// -->  0.0
```

Sprawdzanie typu argumentu

```
// double d = lua_tonumber(L, 1);
double d = luaL_checknumber(L, 1);
```

- Rodzina funkcji `luaL_check*TYPE*` zwraca wartość zadanego typu ze stosu,
- chyba że się nie da,
- wtedy generuje błąd Lua zawierający przydatne informacje
- i przerywa działanie funkcji

```
luaL_loadstring(L, "print (mysin('a'))")...
// --> bad argument #1 to 'mysin2'
// --> (number expected, got string)
```

Obsługa błędów

```
luaL_loadstring(L, "print (idiv(11,0))")...
// -->
```

Obsługa błędów

```
luaL_loadstring(L, "print (idiv(11,0))")...
// -->  Floating point exception
```

Obsługa innych błędów

```
int n2 = luaL_checkinteger(L, 2);
if(n2==0) return luaL_error(L,"division by zero");
```

- Z poziomu C możemy wygenerować własny błąd Lua korzystając z:

```
int luaL_error(lua_State *L,const char *fmt, ...);
```

- (automatycznie uzupełniany o nazwę pliku/numer linii)
- Funkcja posiada adnotację, że *never returns*
- Zazwyczaj stosuje się `return luaL_error(args)`

```
luaL_loadstring(L, "print (idiv(11,0))")..
// --> division by zero
```

Z C do Lua

example01.cpp

Przykład: listowanie zawartości folderu

`example02.cpp`

- udostępniamy Lua funkcje `dir`, która pobiera ścieżkę i zwraca sekwencję wszystkich wpisów na tej ścieżce
- Korzystamy przy tym z POSIXowych `opendir`, `readdir`, `closedir`
- Funkcja `l_dir` może (`in the unlikely event of ...`) doprowadzić do wycieków pamięci.
  Gdzie?

Przykład: listowanie zawartości folderu

`example02.cpp`

- udostępniamy Lua funkcje `dir`, która pobiera ścieżkę i zwraca sekwencję wszystkich wpisów na tej ścieżce
- Korzystamy przy tym z POSIXowych `opendir`, `readdir`, `closedir`
- Funkcja `l_dir` może (`in the unlikely event of ...`) doprowadzić do wycieków pamięci.
  Gdzie?
  - `lua_newtable`, `lua_pushstring`, `lua_settable` mogą zawieść z powodu braku pamięci.
  - Jeśli tak się stanie, działanie funkcji zostanie przerwane bez wywołania `closedir`.

Rejestrowanie modułu C

- Zwykły moduł Lua to zazwyczaj tablica z polami/metodami
- Moduł C jest biblioteką dynamiczną która z naszego punktu widzenia zachowuje się tak samo
- W tym celu musi on eksportować funkcję rejestrującą, która odpowiednio poukłada elementy modułu i opakuje w tablicę będącą „główną tablicą" modułu.
- Po rejestracji, Lua wywołuje funkcję bezpośrednio przez wywołanie pod zadanym adresem, a więc niezależnie od położenia modułu, nazwy funkcji itd.
- Zazwyczaj moduł C ma jedną publiczną metodę która odpowiada za otworzenie biblioteki, natomiast wszystkie pozostałe składowe są prywatne i statyczne

Rejestrowanie modułu C

- Po pierwsze tworzymy tablicę definiującą zawartość naszego modułu.
- Wpisy do tablicy są typu `luaL_Reg`, a więc zawierają nazwę pod jaką funkcja ma być widziana oraz wskaźnik na tę funkcję.
- Tablicę zamykamy specjalnym wpisem z `NULL`ami

```
static const struct luaL_Reg mylib [] =
 {
  {"mysin", l_sin2},
  {"idiv", l_idiv2},
  {NULL, NULL}  // sentinel
};
```

Rejestrowanie modułu C

- Następnie możemy skorzytać z makra `luaL_newlib`, które pobiera zadane tablicą składowe modułu i rejestruje je w nowo utworzonej tablicy.
- Tablica ta leży na stosie, więc nasza funkcja zwracając 1, przekazuje ją jako wynik

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

- (może wymagać `extern "C"`)

```
local e3 = require"???"
print (e3.mysin(3.14)) --> 0.0015926529164868
print (e3.idiv(11,3))  --> 3   2
```

Rejestrowanie modułu C

- Następnie możemy skorzytać z makra `luaL_newlib`, które pobiera zadane tablicą składowe modułu i rejestruje je w nowo utworzonej tablicy.
- Tablica ta leży na stosie, więc nasza funkcja zwracając 1, przekazuje ją jako wynik

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

- (może wymagać `extern "C"`)

```
local e3 = require"exampleThree"
print (e3.mysin(3.14)) --> 0.0015926529164868
print (e3.idiv(11,3))  --> 3   2
```

Rejestrowanie modułu C

- Po kompilacji modułu do biblioteki dynamicznej (`.dll` lub `.so`) i umieszczeniu jej w odpowiednim miejscu (C path), możemy załadować ją do Lua.
- Polega to na odnalezieniu funkcji ładującej w bibliotece, zarejestrowaniu jako funkcji C, i uruchomieniu które zwraca moduł
- (wymagany prototyp jest taki jak przy każdej funkcji rejestrowanej)

```
lua5.3: error loading module 'example3'
from file './example3.so':
./example3.so: undefined symbol: luaopen_example3
```

- Aby uruchomić funkcję `luaopen_exampleThree`, linker musi wiedzieć jak się ona nazywa.
- Dlatego nazywać się musi `luaopen_*NAZWA_MODUŁU*`,
- gdzie nazwa modułu to nazwa skompilowanego pliku biblioteki

Rejestrowanie modułu C

example03.cpp + example03.lua

- Definiując funkcję `luaopen_*`, wcale nie musimy korzystać z funkcji rejestrującej bibliotekę.
- Choć jest ona wygodna i pomocna, możemy (i potrafimy) to samo zrobić ręcznie:

```
int luaopen_exampleThree(lua_State *L)
{
  lua_newtable(L);
  lua_pushcfunction(L, l_sin2);
  lua_setfield(L, -2, "mysin");
  lua_pushcfunction(L, l_idiv2);
  lua_setfield(L, -2, "idiv");
  return 1;
}
```

Obiektowość, cd.

Prywatność

- Standardowa implementacja obiektów w Lua nie daje możliwości definiowania prywatnych pól i metod
- Zazwyczaj po prostu nie tworzy się niczego co nie powinno być widoczne z zewnątrz,
- lub zaznacza się „prywatne” nazwy poprzez dodanie na końcu podkreślnika

- Lua jest językiem na tyle elastycznym, że oczywiście możemy taki mechanizm zasymulować
- Ale raczej nie jest on często wykorzystywany w praktyce

- Idea polega na trzymaniu obiektu w dwóch tablicach
- Jednej zawierającej jego niedostępny z zewnatrz stan
- Drugiej zawierającej jego dostępne operacje
- Do ukrycia prywatnego stanu korzystamy z domknięć

Prywatność

```
function newAccount (initialBalance)
  local self = {balance = initialBalance}
  local withdraw = function (v)
          self.balance = self.balance - v
        end
  local deposit = function (v)
          self.balance = self.balance + v
        end
  local getBalance =
        function () return self.balance end
  return {
           withdraw = withdraw,
           deposit = deposit,
           getBalance = getBalance
         }
end
```

Prywatność

- Metody obiektu mają dostęp do stanu za pośrednictwem domknięcia, które to domknięcie zapewnia również pełą prywatność stanu
- Ponieważ metody nie korzystają z dodatkowego parametru `self` powinniśmy je wywoływać z kropką

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance()) --> 60
```

- Możemy także definiować prywatne metody
- Wystarczy, że nie dołączymy ich do tablicy z interfejsem

- Tak naprawdę stosujemy ten sam mechanizm co w modułach, tylko w odniesieniu do klas

Prywatność

```
function newAccount (initialBalance)
  local self = {
    balance = initialBalance,
    LIM = 10000.00,
  }
  local extra = function ()
    if self.balance > self.LIM then
      return self.balance*0.10
    else
      return 0
    end
  end
  local getBalance = function ()
    return self.balance + extra()
  end
  <reszta jak wcześniej>
```

Model jednofunkcyjny

- Ze specjalnym przypadkiem mamy do czynienia jeśli obiekt udostępnia tylko jedną metodę
- Możemy wtedy pominąć tworzenie interfejsu w formie tablicy i zwrócić samą tą metodę jako reprezentację obiektu
- Z takim rozwiązaniem zetknęliśmy się już w postaci iteratorów

- Takie rozwiązanie można zastosować jeśli potrzebujemy zaimplementować metodę typu *dispatch*
- Tzn. taką która wykonuje różne akcje w zależności od wartości jednego z argumentów

```
function newObject (value)
  return function (action, v)
    if action == "get" then return value
    elseif action == "set" then value = v
    else error("invalid action")
    end
  end
```

Model jednofunkcyjny

```
d = newObject(0)
print(d("get")) --> 0
d("set", 10)
print(d("get")) --> 10
```

- To niekonwencjonalne rozwiązanie jest tak naprawdę całkiem efektywne
- Składnia wyrażenia jest jedynie kilka znaków dłuższa od standardowego wywołania metody z dwukropkiem
- Każdy obiekt wykorzystuje jedno domknięcie, co jest zazwyczaj mniej kosztowne niż używanie tablicy
- Tracimy możliwość dziedziczenia, ale mamy zapewniona prywatność – do obiektu można się dostać wyłącznie za pośrednictwem jego metody

Dualna reprezentacja

- Dualna reprezentacja odwraca logikę przypisania klucza do tablicy
- I zamiast tego przypisujemy tablicę do klucza

```
table[key] = value

key = {}
...
key[table] = value
```

```
function Account:withdraw (v)
  balance[self] = balance[self] - v
end
```

- Zyskujemy prywatność – tablica `balance` jest trzymana lokalnie w module `Account`
- Tracimy garbage collecting – obiekt jest kluczem w tablicy i nie zostanie odśmiecony dopóki ktoś explicite tego wskaźnika nie usunie

Dualna reprezentacja

```
local balance = {}
Account = {}
function Account:withdraw (v)
  balance[self] = balance[self] - v
end
function Account:deposit (v)
  balance[self] = balance[self] + v
end
function Account:balance(v)
  return balance[self]
end
function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  balance[o] = 0
  return o
```

Dualna reprezentacja

```
a = Account:new{}
a:deposit(100.00)
print (a:balance())
```

- Zapewniamy prywatność danych obiektu i bezpieczny interfejs dostępu do nich
- Dziedziczenie działa bez zmian
- Koszt obliczeniowy jest porównywalny z podejściem standardowym
- Problem z garbage collectorem da się obejść

Dziękuję za uwagę

Za **tydzień**:
techniki pisania funkcji w C,
userdata,
...?