

Lua и C

Говорят, что когда Билл Гейтс и Пол Аллен писали первый интерпретатор Бейсика для компьютера Альтаир, они не пользовались генераторами лексических и синтаксических анализаторов и формальными грамматиками, потому что ничего о них не знали. В этом факте можно усмотреть как проявление гениальности основателей Микрософт, так и отсутствие у них надлежащей квалификации. Как бы там ни было, современные программисты так же могут наделять свои программы возможностями интерпретации языков, ничего не зная об инструментах создания интерпретаторов. Умные люди упростили решение этой задачи до такой степени, что теперь результатами их труда могут пользоваться все желающие. Остается надеяться, что с годами избалованные программисты не станут глупее.

В этой, заключительной части речь пойдет о том, что делает Lua таким привлекательным языком программирования. Все синтаксические красоты Lua вместе взятые вряд ли привлекли бы к нему внимание, если бы не главная функция Lua – возможность встраивать интерпретатор этого языка программирования в программы, написанные на C и других компилируемых языках. О том, что такая возможность дает разработчикам, было рассказано в первой части этой серии. С точки зрения программного интерфейса объединение Lua и C позволяет решить две задачи (как правило, эти задачи одновременно и решаются): управление программой Lua из программы, написанной на C и вызов из программ Lua функций, написанных на C (в этом случае можно сказать, что программа Lua управляет C-кодом).



Рисунок 1. Игра UnknownHorizon — пример программы, использующей интерпретатор Lua.

Ядро интерпретатора Lua находится в библиотеке `liblua`. Программа-интерпретатор, которой мы пользовались до сих пор (файл `lua5.1`), представляет собой лишь тоненькую обертку вокруг этой библиотеки. Для того чтобы наделить программу на C возможностью использовать язык Lua, нужно связать ее с библиотекой `liblua` и задействовать экспортируемый библиотекой API. Прежде чем выполнять примеры из этой статьи, убедитесь, что файл `liblua.so` в вашей системе является ссылкой на библиотеку `liblua.so.5.1`. С API довольно заметно изменился при переходе от версии Lua 5.0 к версии 5.1, а в наших примерах мы будем использовать API последней версии.

Два наиболее важных понятия в Lua C API это «состояние Lua» и стек. Функции и макросы Lua API должны получать разного рода информацию об интерпретаторе Lua. Вся необходимая информация хранится в структуре данных, именуемой состоянием Lua (Lua state). Адрес этой структуры является обязательным параметром для всех функций и макросов Lua C API (и всегда передается в качестве первого параметра, так что дальше мы не будем каждый раз упоминать о нем). Для тех, кто регулярно имеет дело с различными программными интерфейсами в стиле C, структура, описывающая состояние Lua, не представляет собой ничего необычного. Во многих программных интерфейсах есть такие структуры, содержащие информацию о состоянии программируемого элемента (окна, кодека, драйвера устройства и тому подобного). Указатель на структуру, описывающую состояние Lua, можно рассматривать как хэндл экземпляра интерпретатора Lua (только нужно помнить, что экземпляра, как такового, не существует). Если состояние Lua выглядит абстрактным понятием, то стек Lua еще более абстрактен. Как мы знаем, переменные Lua очень сильно отличаются от переменных C, поэтому нет никакой возможности установить прямое соответствие между переменными в программе-хозяине и переменными в программе Lua. Обмен данными между Lua и C выполняется через стек Lua, который является частью состояния Lua. Стек Lua ведет себя как классический стек для абстрактных типов данных. В него можно помещать элементы, которые затем могут быть извлечены в порядке, обратном порядку их помещения. Практически все функции C Lua API выполняют операции над элементами стека и помещают результат операции в стек. Поскольку переменные Lua полиморфны, ячейки стека могут содержать данные любого типа, поддерживаемого Lua. Прежде чем использовать данные из стека в программе C, необходимо убедиться, что эти данные имеют требуемый тип. С помощью стека между программой на C и кодом Lua передаются не только простые переменные, но и такие вещи как ссылки на функции и таблицы, описывающие загруженные модули. Вот почему стек Lua является важнейшей концепцией всего API. Все это не так сложно как кажется. Рассмотрим пример программы на C, загружающий скрипт Lua (вы найдете этот текст на диске, в файле `runlua.c`):

```
#include <lua.h>
#include <luauxlib.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int result;
    lua_State * L;
    L = luaL_newstate();
    luaL_openlibs(L);
    result = luaL_loadfile(L, "script.lua");
    if (result) {
        fprintf(stderr, "Ошибка загрузки: %s\n", lua_tostring(L, -1));
        exit(1);
    }
    lua_pushnumber(L, 2);
    lua_setglobal(L, "var");
    result = lua_pcall(L, 0, LUA_MULTRET, 0);
    if (result) {
```

```

    fprintf(stderr, "Ошибка выполнения: %s\n", lua_tostring(L, -
1));
    exit(1);
}
printf("Успешное завершение\n");
lua_close(L);
return EXIT_SUCCESS;
}

```

Эта программа загружает файл `script.lua`, создает глобальную переменную `Lua var` со значением 2 и выполняет Lua-программу. Рассмотрим этот код подробнее. Элементы Lua C API описаны в заголовочных файлах `lua.h` и `luaXlib.h`. Переменная `L` – это указатель на структуру, описывающую состояние Lua. Саму структуру мы создаем с помощью функции `luaL_newstate()`. Функция `luaL_openlibs()` открывает стандартные библиотеки Lua для заданного состояния Lua. Если не вызывать эту функцию, загруженная программа Lua не сможет обращаться к элементам стандартных библиотек Lua, к таким, например, как функция `print()`. Функция `luaL_loadfile()` загружает текст программы Lua из файла. В случае успеха функция возвращает 0, в противном случае – ненулевое значение, свидетельствующее об ошибке.

Библиотека Lua возвращает подробные текстовые описания всех обнаруженных ошибок, причем использует для этого стандартный механизм передачи данных между Lua и программой-хозяином, то есть, стек. В случае возникновения ошибки мы читаем ее описание с помощью функции `lua_tostring()`. Обратите внимание на второй аргумент функции. Как и в большинстве функций Lua API, имеющих дело с данными Lua, в качестве источника данных для `lua_tostring()` мы указываем ячейку стека. Ячейки обозначаются индексами, причем верхушка (последний добавленный элемент) стека имеет индекс 1, следующая ячейка – индекс 2 и так далее. Таким образом, вызов функции

```
lua_tostring(L, -1)
```

преобразует в строку C значение, содержащееся в верхней ячейке стека. Аналогично, конструкция `lua_tostring(L, 1)`

считывает значение из последней ячейки стека (мы считаем первую ячейку вершиной стека, а последнюю – дном, в то время как документация по Lua противоположную систему обозначений). В нашем примере будет считано описание ошибки, которое помещается на верхушку стека. Важно отметить, что функция `lua_tostring()` (так же как функция `lua_tonumber()` и ей подобные) не удаляет значения из стека. При работе с функциями Lua API всегда важно помнить, как именно эти функции влияют на стек. Библиотеки Lua не следят за переполнением стека, - вы сами должны заботиться о том, чтобы в стеке не было ничего лишнего. Для удаления заданного значения из стека предназначена функция `lua_pop()`. Мы не вызываем эту функцию после вызова `lua_tostring()` только потому, что программа в этом случае все равно завершается и очищать стек специально нет необходимости. Если текст программы загружен успешно, функция `luaL_loadfile()` создает выполняемый интерпретатором Lua фрагмент и помещает его на вершину стека. Далее мы создаем глобальную переменную `var`. Функция `lua_pushnumber()` помещает в стек числовое значение. Макрос `lua_setglobal()` создает глобальную переменную и присваивает ей значение из ячейки, находящейся на вершине стека. Если мы посмотрим определение макроса `lua_setglobal()`, то увидим, что этот макрос использует функцию `lua_setfield()`. Первым аргументом этой функция является указатель на структуру, описывающую состояние Lua, вторым аргументом – индекс таблицы, в которую добавляется переменная, а третьим – строка C с именем переменной. Тут необходимо напомнить, что все переменные Lua являются элементами каких-либо таблиц. Глобальные переменные являются элементами таблицы `_G` (см. предыдущую статью). Именно с этой таблицей и работает макрос `lua_setglobal()`. Разумеется, что с помощью функции `lua_setfield()` (и макроса `lua_setglobal()`) можно не только создавать новые переменные, но и модифицировать значения уже существующих. С помощью этих же функций можно уничтожать переменные Lua. Делается это так же, как и внутри Lua-кода, то есть путем присвоения переменной значения `nil`. Для записи в стек значения `nil` предназначена специальная функция `lua_pushnil()`. В заключение

описания функции `lua_setfield()` отметим, что эта функция удаляет значение с вершины стека, так что вызов `lua_pop()` после нее не нужен.

Функция `lua_pcall()` выполняет самую волшебную часть нашей программы – запускает сценарий Lua на исполнение. Второй аргумент функции `lua_pcall()` – число аргументов, переданных выполняемому фрагменту (в нашем примере – 0). Третий аргумент `lua_pcall()` – число значений, которые фрагмент возвращает (мы используем константу `LUA_MULTRET`, которая указывает, что число возвращаемых значений может быть переменным). Помимо функции `lua_pcall()` API предоставляет нам функцию `lua_call()`. Те, кто читал предыдущую статью, наверняка уже догадались, в чем разница между этими функциями. Функция `lua_pcall()` выполняет загруженный код в «защищенном режиме», то есть, подавляет все возникшие во время выполнения кода Lua ошибки, а не передает их на более высокий уровень. Откуда функция `lua_pcall()` знает, какой фрагмент кода Lua она должна выполнить? В процессе вызова `lua_pcall()` из стека извлекаются аргументы, переданные фрагменту кода Lua (если они есть). Функция `lua_pcall()` ожидает, что после всех аргументов в стеке расположена ссылка на исполняемый фрагмент. Эта ссылка тоже извлекается из стека. Все значения, возвращенные выполненным фрагментом Lua, помещаются в стек. Таким образом, после завершения `lua_pcall()`, стек содержит только возвращенные значения. Последний аргумент `lua_pcall()` – индекс функции Lua, используемой для обработки ошибок. Если это значение равно 0, то в случае возникновения ошибки на вершине стека возвращается ее текстовое описание (которое мы выводим так же, как и в случае функции `luaL_loadfile()`). Теперь мы можем понять, почему интерпретатор Lua способен выполнять несколько фрагментов программы, содержащихся в разных файлах, в едином контексте. Все, что для этого нужно – вызов нескольких функций `lua_(p)call()` с одной и той же переменной, описывающей состояние интерпретатора Lua.

Функция `lua_close()` уничтожает структуру, описывающую состояние Lua.

Файл `script.lua`, который мы загружаем на выполнение в программе-примере, может содержать любой корректный фрагмент программы Lua. Нашу среду выполнения Lua отличает то, что загруженному фрагменту доступна глобальная переменная `var`. Это можно проверить с помощью простейшей конструкции:

```
print("переменная var", var)
```

Сохраним эту строку в файле `script.lua`.

Для компиляции нашей программы на C скомандуем:

```
gcc runlua.c -o runlua -llua
```

Теперь можно запустить программу `runlua` и посмотреть, как она работает. Отредактируйте текст `script.lua` и посмотрите, как наша программа реагирует на различные ошибки в коде Lua.

В предыдущей статье мы использовали возможности Lua для создания программы-калькулятора на языке Lua. Мы можем использовать Lua для добавления функций калькулятора в программу, написанную на C++ (файл `calc.cpp`):

```
#include <iostream>
#include <string>
extern "C" {
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
}
```

```
using namespace std;
```

```
size_t l;
```

```
const char * reader (lua_State *L, void *data, size_t *size) {
    char * result;
    * size = l;
    if (!l)
        result = 0;
```

```

    else
        result = (char *) data;
    l = 0;
    return result;
}

int main()
{
    int result;
    lua_State * L;
    L = luaL_newstate();
    luaL_openlibs(L);
    while (true) {
        cout << "Введите строку или нажмите Ctrl-C" << endl;
        string s;
        cin >> s;
        s = "do return " + s + " end \0";
        l = s.length() + 1;
        result = lua_load(L, reader, &s[0], "calc");
        if (result) {
            cout << lua_tostring(L, -1) << endl;
        }
        else
        {
            result = lua_pcall(L, 0, 1, 0);
            if (result != LUA_ERRRUN)
                cout << lua_tonumber(L, -1) << endl;
            else
                cout << lua_tostring(L, -1) << endl;
        }
        lua_pop(L, 1);
    }
    lua_close(L);
}

```

Программа-калькулятор считывает строку со стандартного потока ввода и пытается вычислить содержащееся в ней математическое выражение. Наш микрокалькулятор принимает выражения вида $2*(3+4)$, $\text{math.sin}(3.14)$ и им подобные. Работа выполняется в бесконечном цикле, из которого можно выйти с помощью Ctrl-C.

Обратите внимание, что в программе, написанной на C++, мы должны явным образом указывать, что функции Lua API экспортируются в формате C. После того как мы считали строку (переменная *s*) мы должны скомпилировать ее в функцию Lua. В калькуляторе, написанном на Lua, для этого можно было использовать функции, предназначенные специально для компиляции исходного кода из строк Lua. В Lua C API такие функции тоже были (например, функция `lua_dostring()`), но теперь они признаны устаревшими. Вместо них следует использовать функцию `lua_load()`, которая может загружать исходный текст программы Lua из любого источника с помощью специальной вспомогательной функции. В нашем примере это функция `reader()`. Функция `reader()` используется Lua C API как функция обратного вызова. При каждом вызове функция должна вернуть указатель на новый фрагмент исходного текста программы Lua. Длина очередного фрагмента возвращается в параметре `size`. Параметр `data` представляет собой указатель на данные, определенные программистом. Функция `reader()` сигнализирует о том, что она прочитала весь исходный текст, возвращая значение `NULL`. Наша функция `reader()` при первом просто преобразует введенную пользователем программы строку C++ в строку `char *` и возвращает ее длину. При втором вызове функция возвращает `NULL`.

Вернемся к функции `lua_load()`. Второй параметр этой функции – адрес вспомогательной функции для чтения исходного текста. Третий аргумент – это адрес данных для вспомогательной функции. Далее следует строка с именем загружаемого фрагмента (можно передать пустую строку). Обработка ошибок, возникших во время выполнения функции `lua_load()` осуществляется так же, как и в случае `luaL_loadfile()`. Так же, как и функция `luaL_loadfile()`, функция `lua_load()` создает выполняемый интерпретатором Lua фрагмент и помещает его на вершину стека. После завершения `lua_pcall()` стек содержит значение, вычисленное нашим калькулятором. Это значение, которое может иметь тип `nil` в случае ошибки, мы считываем с верхушки стека с помощью вызова `lua_tonumber()`. Если вершина стека содержит нечисловое значение, вызов `lua_tonumber()` возвращает 0. После этого нам остается только удалить значение из стека с помощью `lua_pop()`.

```

andrei@linux-3nc6:~/Articles/lua-4
andrei@linux-3nc6:~/Articles/lua-4>
andrei@linux-3nc6:~/Articles/lua-4> ./calc
Введите строку или Ctrl-C
math.sin(1.57)^2+math.cos(1.57)^2
1
Введите строку или Ctrl-C
1/2^2+1/3^2+1/4^2+1/5^2
0.463611
Введите строку или Ctrl-C
(1+100)^2/2
5100.5
Введите строку или Ctrl-C
(1+100)^2
10201
Введите строку или Ctrl-C
1*2*3*4*5*6*7*8*9*10
3.6288e+06
Введите строку или Ctrl-C
math.tan(math.pi/2)
1.63318e+16
Введите строку или Ctrl-C
^C
andrei@linux-3nc6:~/Articles/lua-4>

```

Рисунок 2: Наш калькулятор может вычислить все, даже тангенс от $\pi/2$!

Посмотрим теперь, как вызвать из Lua функцию, написанную на C. Разумеется, в Lua нельзя импортировать любую функцию из C-библиотеки, уж слишком разные это языки. Тем не менее, написать C-функцию в «экспортном варианте» оказывается на удивление просто. Рассмотрим сначала исходный текст библиотеки `testlib` (файл `testlib.c`), написанной на C:

```

#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
#include <stdlib.h>

static int _system (lua_State *L)
{
    int result = system(lua_tostring(L, -1));
    lua_pushinteger(L, result);
    return 1;
}

int luaopen_testlib(lua_State *L)
{
    static const luaL_Reg Map [] = {"system", _system, {NULL, NULL}}
;

```

```

    luaL_register(L, "testlib", Map);
    return 1;
}

```

Эта библиотека предоставляет Lua одну функцию – system(). Функция testlib.system() (под этим именем она будет доступна в Lua) делает то же, что и ее тезка из стандартной библиотеки C (разумеется в стандартной библиотеке Lua тоже есть подобная функция). Во избежание неоднозначностей, в нашей C-библиотеке экспортируемой функции присвоено имя _system. Для того чтобы понять, как работает экспорт C-функций, необходимо различать передачу аргументов и возврат значений в контексте C и те же действия в контексте Lua. Единственным аргументом экспортируемой функции в контексте C должен быть указатель на структуру lua_State, а возвращаемым значением (в контексте C) – число значений, возвращаемых в контексте Lua. Да, вы правильно поняли, в контексте Lua функция экспортируемая из C, может получать произвольное количество аргументов различных типов и возвращать произвольное число значений. Все аргументы и значения в контексте Lua передаются, разумеется, через стек. Мы считываем единственный аргумент функции testlib.system() с вершины стека, превращаем его в строку, вызываем C-функцию system() и помещаем результат выполнения функции в стек Lua. Поскольку в контексте Lua наша функция возвращает одно значение, в контексте C она возвращает значение 1.

Для того чтобы подготовить библиотеку к работе с Lua, нам надо написать еще одну функцию – luaopen_libname(), где libname соответствует имени библиотеки. Эта функция будет вызвана интерпретатором Lua при загрузке нашей библиотеки. Именно она позволяет использовать C-библиотеку как стандартный пакет Lua.

Массив Map состоит из пар «имя функции в контексте Lua – указатель на C функцию». Эти данные интерпретатор Lua будет использовать для вызова функции, написанной на C. Между прочим, поскольку имя функции в библиотеке C не имеет значения для интерпретатора, функции, в принципе, можно экспортировать и из кода C++, не указывая формат функции (это, правда, не касается функции luaopen_*, которую интерпретатор ищет по имени C). Функция luaL_register() регистрирует новую библиотеку и помещает ссылку на соответствующий объект на вершину стека. Вторым аргументом функции luaL_register() должно быть имя библиотеки в контексте Lua, которое может и не совпадать с именем разделяемого модуля.

Для компиляции библиотеки командуем

```
gcc testlib.c -shared -o testlib.so
```

Теперь можно перейти к Lua-коду:

```

require "testlib"
print("Вызов функции C")
res = testlib.system("/usr/bin/mc")
if res ~= 0 then
print("Код завершения программы", res)
else
print("Нормальный выход")
end

```

Наш модуль testlib подключается к программе, написанной на Lua, так же, как и библиотеки Lua, рассмотренные в предыдущей статье. Обратите внимание, что хотя в библиотеке testlib и используются функции Lua API, при компиляции мы не связываем нашу библиотеку с библиотекой liblua. В этом нет необходимости, так как библиотека testlib будет загружена интерпретатором в то же адресное пространство, что и библиотека liblua. Между прочим, экспортировать функции C в Lua можно не только из разделяемых библиотек, но и из исполнимого файла программы, загружающей код Lua на выполнение. В этом случае нам пригодится другая функция Lua API – lua_register().

Каким образом функция C может узнать, сколько аргументов ей передано? В этом поможет функция lua_gettop(), которая возвращает число элементов в стеке. В момент вызова функции C это число равно числу аргументов, переданных функции. Строго говоря, lua_gettop() возвращает

индекс последней ячейки стека, которая в терминологии Lua именуется вершиной стека (в описании стека я, как и авторы многих руководств по Lua, придерживался других обозначений - вершина – первый элемент стека). Этой функцией можно пользоваться и для того, чтобы узнать, не переполнен ли стек. Глубина стека Lua контролируется константой MAX_STACK. Значение этой константы невелико, обычно не превышает 32, но для любого разумного использования этого должно быть достаточно. В конце концов, глубина стека математического спороцессора в процессорах Intel всего 8 ячеек.

Встраивание скриптовых языков программирования в пользовательские программы вполне соответствует идеологии Unix, согласно которой все, что может быть настроено пользователем под свои нужды, не должно быть зашито в исходном коде программы, а должно быть вынесено в отдельные системы конфигурации. Язык Lua – простое и мощное средство, позволяющее наделять вашу программу такой системой.