### **Lua**

小巧高效的脚本语言 Lua，号称是“最快的脚本语言”。

你可能对 Lua 不太了解，但你一定听说过《魔兽世界》《愤怒的小鸟》吧，它们就在内部大量使用了 Lua 来编写逻辑。在游戏开发领域，Lua 可以说是一种通用的工作语言。

Lua 与其他语言最大的不同点在于它的设计目标：不追求“大而全”，而是“小而美”。Lua 自身只有很小的语言核心，能做的事情很少。但正是因为它小，才能够很容易地嵌入到其他语言里，为“宿主”添加脚本编程的能力，让“宿主”更容易扩展和定制。

标准的 Lua（PUC-Rio Lua）使用解释器运行，速度虽然很快，但和 C/C++ 比起来还是有差距的。所以，你还可以选择另一个兼容的项目：LuaJIT（https://luajit.org/）。它使用了 JIT（Just in time）技术，能够把 Lua 代码即时编译成机器码，速度几乎可以媲美原生 C/C++ 代码。

不过，LuaJIT 也有一个问题，它是一个个人项目，更新比较慢，最新的 2.1.0-beta3 已经是三年前的事情了。所以，我推荐你改用它的一个非官方分支：OpenResty-LuaJIT（https://github.com/openresty/luajit2）。它由 OpenResty 负责维护，非常活跃，修复了很多小错误。

git clone git@github.com:openresty/luajit2.git

make && make install

和 Python 一样，Lua 也有 C 接口用来编写扩展模块，但因为它比较小众，所以 C++ 项目不是很多。现在我用的是 LuaBridge，虽然它没有用到太多的 C++11 新特性，但也足够好。

LuaBridge 是一个纯头文件的库，只要下载下来，把头文件拷贝到包含路径，就能够直接用：

git clone git@github.com:vinniefalco/LuaBridge.git

我们先来看看在 Lua 里怎么调 C++ 的功能。

和前面说的 pybind11 类似，LuaBridge 也定义了很多的类和方法，可以把 C++ 函数、类注册到 Lua 里，让 Lua 调用。

但我不建议你用这种方式，因为我们现在有 LuaJIT。它内置了一个 ffi 库（Foreign Function Interface），能够在 Lua 脚本里直接声明接口函数、直接调用，不需要任何的注册动作，更加简单方便。而且这种做法还越过了 Lua 传统的栈操作，速度也更快。

使用 ffi 唯一要注意的是，它只能识别纯 C 接口，不认识 C++，所以，写 Lua 扩展模块的时候，内部可以用 C++，但对外的接口必须转换成纯 C 函数。

下面我写了一个简单的 add() 函数，还有一个全局变量，注意里面必须要用 extern "C"声明：

extern "C" { // 使用纯C语言的对外接口

int num = 10;

int my\_add(int a, int b);

}

int my\_add(int a, int b) // 一个简单的函数，供Lua调用

{

return a + b;

}

然后就可以用 g++ 把它编译成动态库，不像 pybind11，它没有什么特别的选项：

g++ lua\_shared.cpp -std=c++11 -shared -fPIC -o liblua\_shared.so

在 Lua 脚本里，你首先要用 ffi.cdef 声明要调用的接口，再用 ffi.load 加载动态库，这样就会把动态库所有的接口都引进 Lua，然后就能随便使用了：

local ffi = require "ffi" -- 加载ffi库

local ffi\_load = ffi.load -- 函数别名

local ffi\_cdef = ffi.cdef

ffi\_cdef[[ // 声明C接口

int num;

int my\_add(int a, int b);

]]

local shared = ffi\_load("./liblua\_shared.so") -- 加载动态库

print(shared.num) -- 调用C接口

local x = shared.my\_add(1, 2) -- 调用C接口

在 ffi 的帮助下，让 Lua 调用 C 接口几乎是零工作量，但这并不能完全发挥出 Lua 的优势。

因为和 Python 不一样，Lua 很少独立运行，大多数情况下都要嵌入在宿主语言里，被宿主调用，然后再“回调”底层接口，利用它的“胶水语言”特性去粘合业务逻辑。

要在 C++ 里嵌入 Lua，首先要调用函数 luaL\_newstate()，创建出一个 Lua 虚拟机，所有的 Lua 功能都要在它上面执行。

因为 Lua 是用 C 语言写的，Lua 虚拟机用完之后必须要用函数 lua\_close() 关闭，所以最好用 RAII 技术写一个类来自动管理。可惜的是，LuaBridge 没有对此封装，所以只能自己动手了。这里我用了智能指针 shared\_ptr，在一个 lambda 表达式里创建虚拟机，顺便再打开 Lua 基本库：

auto make\_luavm = []() // lambda表达式创建虚拟机

{

std::shared\_ptr<lua\_State> vm( // 智能指针

luaL\_newstate(), lua\_close // 创建虚拟机对象，设置删除函数

);

luaL\_openlibs(vm.get()); // 打开Lua基本库

return vm;

};

#define L vm.get() // 获取原始指针，宏定义方便使用

在 LuaBridge 里，一切 Lua 数据都被封装成了 LuaRef 类，完全屏蔽了 Lua 底层那难以理解的栈操作。它可以隐式或者显式地转换成对应的数字、字符串等基本类型，如果是表，就可以用“[]”访问成员，如果是函数，也可以直接传参调用，非常直观易懂。

使用 LuaBridge 访问 Lua 数据时，还要注意一点，它只能用函数 getGlobal() 看到全局变量，所以，如果想在 C++ 里调用 Lua 功能，就一定不能加“local”修饰。

给你看一小段代码，它先创建了一个 Lua 虚拟机，然后获取了 Lua 内置的 package 模块，输出里面的默认搜索路径 path 和 cpath：

auto vm = make\_luavm(); // 创建Lua虚拟机

auto package = getGlobal(L, "package"); // 获取内置的package模块

string path = package["path"]; // 默认的lua脚本搜索路径

string cpath = package["cpath"]; // 默认的动态库搜索路径

你还可以调用 luaL\_dostring() 和 luaL\_dofile() 这两个函数，直接执行 Lua 代码片段或者外部的脚本文件。注意，luaL\_dofile() 每次调用都会从磁盘载入文件，所以效率较低。如果是频繁调用，最好把代码读进内存，存成一个字符串，再用 luaL\_dostring() 运行：

luaL\_dostring(L, "print('hello lua')"); // 执行Lua代码片段

luaL\_dofile(L, "./embedded.lua"); // 执行外部的脚本文件

在 C++ 里嵌入 Lua，还有另外一种方式：提前在脚本里写好一些函数，加载后在 C++ 里逐个调用，这种方式比执行整个脚本更灵活。

具体的做法也很简单，先用 luaL\_dostring() 或者 luaL\_dofile() 加载脚本，然后调用 getGlobal() 从全局表里获得封装的 LuaRef 对象，就可以像普通函数一样执行了。由于 Lua 是动态语言，变量不需要显式声明类型，所以写起来就像是 C++ 的泛型函数，但却更简单：

string chunk = R"( -- Lua代码片段

function say(s) -- Lua函数1

print(s)

end

function add(a, b) -- Lua函数2

return a + b

end

)";

luaL\_dostring(L, chunk.c\_str()); // 执行Lua代码片段

auto f1 = getGlobal(L, "say"); // 获得Lua函数

f1("say something"); // 执行Lua函数

auto f2 = getGlobal(L, "add"); // 获得Lua函数

auto v = f2(10, 20); // 执行Lua函数

只要掌握了上面的这些基本用法，并合理地划分出 C++ 与 Lua 的职责边界，就可以搭建出“LuaJIT + LuaBridge + C++”的高性能应用，运行效率与开发效率兼得。比如说用 C++ 写底层的框架、引擎，暴露出各种调用接口作为“业务零件”，再用灵活的 Lua 脚本去组合这些“零件”，写上层的业务逻辑。