<https://www.baidu.com/link?url=qpSw7THWsAVWb_PejXNZd_6sHf9yjPXMj-DbPeeijfzmV6LfkMH9BvXxPm_XOMQCuwOQvGOs1uHffJnUJoB3Ia&wd=&eqid=ee4206490002f670000000025ad6c09e>

**Registry**

**lua提供了一个独立的被称为registry的表，c代码可以自由使用，但lua代码不能访问**

registry就是一张普通的lua表，由于所有的c库共享相同的registry表（实际上行它存在于global\_state中），你必须注意使用什么样的值作为key，否则会导致命名冲突。

**References**

应该记住，永远不要使用数字作为registry的key，因为这种类型的key是保留给reference系统使用。Reference系统是由辅助库中的一对函数组成，这对函数用来不需要担心名称冲突的将值保存到registry中去（实际上，这些函数可以用于任何一个表，但他们典型的被用于registry）

lua\_getglobal(L, "show1");

lua\_getglobal(L, "show2");

lua\_getglobal(L, "hello");

int ref1 = luaL\_ref(L, LUA\_REGISTRYINDEX);

int ref2 = luaL\_ref(L, LUA\_REGISTRYINDEX);

int ref3 = luaL\_ref(L, LUA\_REGISTRYINDEX);

cout << "ref1 = " << ref1 << ", ref2 = " << ref2 << ", ref3 = " << ref3 << endl;

luaL\_unref(L, LUA\_REGISTRYINDEX, ref2);

luaL\_unref(L, LUA\_REGISTRYINDEX, ref3);

int ref4 = luaL\_ref(L, LUA\_REGISTRYINDEX);

int ref5 = luaL\_ref(L, LUA\_REGISTRYINDEX);

cout << "ref4 = " << ref4 << ", ref5 = " << ref5 << endl;



源码分析：

LUALIB\_API int luaL\_ref (lua\_State \*L, int t) {

int ref;

t = abs\_index(L, t);

if (lua\_isnil(L, -1)) {

lua\_pop(L, 1); /\* remove from stack \*/

return LUA\_REFNIL; /\* `nil' has a unique fixed reference \*/

}

lua\_rawgeti(L, t, FREELIST\_REF); /\* get first free element \*/ **// FREELIST\_REF是一个为0的宏，实际上它是空闲位置链表的头结点**

ref = (int)lua\_tointeger(L, -1); /\* ref = t[FREELIST\_REF] \*/ **// 通过ref是否等于0判断当前时候有空闲位置**

lua\_pop(L, 1); /\* remove it from stack \*/

if (ref != 0) { /\* any free element? \*/ **// 取头空闲位置，然后删除这个空闲位置，将FREELIST\_REF指向下一个空闲位置**

lua\_rawgeti(L, t, ref); /\* remove it from list \*/

lua\_rawseti(L, t, FREELIST\_REF); /\* (t[FREELIST\_REF] = t[ref]) \*/

}

else { /\* no free elements \*/ **// 如果没有空闲位置，说明table中的数组一定是连续的，可以直接获取长度+1**

ref = (int)lua\_objlen(L, t);

ref++; /\* create new reference \*/

}

lua\_rawseti(L, t, ref);

return ref;

}

**// 解引用实际上是在FREELIST\_REF空闲位置前插空闲位置ref**

LUALIB\_API void luaL\_unref (lua\_State \*L, int t, int ref) {

if (ref >= 0) {

t = abs\_index(L, t);

lua\_rawgeti(L, t, FREELIST\_REF);

lua\_rawseti(L, t, ref); /\* t[ref] = t[FREELIST\_REF] \*/

lua\_pushinteger(L, ref);

lua\_rawseti(L, t, FREELIST\_REF); /\* t[FREELIST\_REF] = ref \*/

}

}

分析上面的例子：

在ref1、ref2、ref3执行完后：

Resigiter[1] = hello(function)

Resigiter[2] = show2(function)

Resigiter[3] = show1(function)

在第一次unref后：

Resigiter[2] = resigiter[0] = nil

Resigiter[0] = 2

在第二次unref后：

Regisiter[3] = resigiter[0] = 2

Resigiter[0] = 3

接下来第一次ref：

由于resigiter[0]=3，因此得到ref=3，然后将resigiter[0]=2

接下来第二次ref

由于resigiter[0]=2，一次得到ref=2，然后将resigiter[0]=nil

**Upvalues**

Registry实现了全局的值，upvalue机制实现了与C static变量等价的东西，这种变量只能在特定的函数内可见。每当你在lua中创建一个新的C函数，你可以将这个函数与任意多个upvalues联系起来，每个upvalue可以持有一个单独的lua值。我们称这种一个C函数和它的upvalues的组合为闭包。记住：在lua代码中，一个闭包是一个从外部函数访问局部变量的函数。一个C闭包与一个lua闭包相近。关于闭包的一个有趣的事实是，你可以使用相同的函数代码创建不同的闭包，带有不同的upvalues

static int counter(lua\_State \*L)

{

double val = lua\_tonumber(L, lua\_upvalueindex(1));

lua\_pushnumber(L, ++val);

lua\_pushvalue(L, -1);

lua\_replace(L, lua\_upvalueindex(1));

return 1;

}

static int newCounter(lua\_State \*L)

{

lua\_pushnumber(L, 0);

lua\_pushcclosure(L, &counter, 1);

return 1;

}

local t1 = mylib.newCounter()

t1()

t1()

local d1 = t1()

print("d1 = ", d1)

local t2 = mylib.newCounter()

t2()

local d2 = t2()

print("d2 = ", d2)

得到结果为：d1=3 d2=2

与lua闭包不同的是，c闭包不能共享upvalues，每一个闭包都有自己独立的变量集。然而我们可以设置通函数的upvalues指向同一个表，这样这个表就变成了一个所有函数共享数据的地方