Lua 和 C 的交互说明（函数）

Lua 与 C 交互(1)

lua_newtable

void lua_newtable (lua_State *L); 创建一个空 table ，并将之压入堆栈。 它等价于 lua_createtable(L, 0, 0) 。

lua_gettop

int lua_gettop (lua_State *L);返回栈顶元素的索引。 因为索引是从 1 开始编号的， 所以这个结果等于堆栈上的元素个数（因此返回 0 表示堆栈为空）。

luaL_newmetatable

int luaL_newmetatable (lua_State *L, const char *tname);

如果注册表中已经有 Key 为 tname 的数据则返回0. 否则创建一个新表作为 userdata 的 metatable，并在注册表中注册它然后返回1. 不过两种情况都会把注册表中 tname 相关的值压入堆栈。

luaL_checkudata

void *luaL_checkudata (lua_State *L, int narg, const char *tname);Checks whether the function argument narg is a userdata of the type tname (see luaL_newmetatable).

lua_pushstring

void lua_pushstring (lua_State *L, const char *s);把指针 s 指向的以零结尾的字符串压栈。 Lua 对这个字符串做一次内存拷贝（或是复用一个拷贝）， 因此 s 处的内存在函数返回后，可以释放掉或是重用于其它用途。 字符串中不能包含有零字符；第一个碰到的零字符会认为是字符串的结束。

lua_pushlstring

void lua_pushlstring (lua_State *L, const char *s, size_t len);把指针 s 指向的长度为 len 的字符串压栈。 Lua 对这个字符串做一次内存拷贝（或是复用一个拷贝）， 因此 s 处的内存在函数返回后，可以释放掉或是重用于其它用途。 字符串内可以保存有零字符。

lua_pushvalue

void lua_pushvalue (lua_State *L, int index);把堆栈上给定有效处索引处的元素作一个拷贝压栈。

lua_settable

void lua_settable (lua_State *L, int index);作一个等价于 t[k] = v 的操作， 这里 t 是一个给定有效索引 index 处的值， v 指栈顶的值， 而 k 是栈顶之下的那个值。
这个函数会把键和值都从堆栈中弹出。 和在 Lua 中一样，这个函数可能触发 "newindex" 事件的元方法 （参见 §2.8）。

lua_pushcfunction

void lua_pushcfunction (lua_State *L, lua_CFunction f);将一个 C 函数压入堆栈。这个函数接收一个 C 函数指针，并将一个类型为 function 的 Lua 值 压入堆栈。当这个栈定的值被调用时，将触发对应的 C 函数。
注册到 Lua 中的任何函数都必须遵循正确的协议来接收参数和返回值 （参见 lua_CFunction）。
lua_pushcfunction 是作为一个宏定义出现的：

```
#define lua_pushcfunction(L,f) lua_pushcclosure(L,f,0)
```

lua_setmetatable

int lua_setmetatable (lua_State *L, int index);把一个 table 弹出堆栈，并将其设为给定索引处的值的 metatable 。

lua_pushcclosure

void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);把一个新的 C closure 压入堆栈。
当创建了一个 C 函数后，你可以给它关联一些值，这样就是在创建一个 C closure （参见§3.4）； 接下来无论函数何时被调用，这些值都可以被这个函数访问到。 为了将一些值关联到一个 C 函数上， 首先这些值需要先被压入堆栈（如果有多个值，第一个先压）。 接下来调用 lua_pushcclosure 来创建出 closure 并把这个 C 函数压到堆栈上。 参数 n 告之函数有多少个值需要关联到函数上。 lua_pushcclosure 也会把这些值从栈上弹出。

lua_newuserdata

void *lua_newuserdata (lua_State *L, size_t size);这个函数分配分配一块指定大小的内存块，把内存块地址作为一个完整的 userdata 压入堆栈，并返回这个地址。
userdata 代表 Lua 中的 C 值。 完整的 userdata 代表一块内存。 它是一个对象（就像 table 那样的对象）： 你必须创建它，它有着自己的元表，而且它在被回收时，可以被监测到。 一个完整的 userdata 只和它自己相等（在等于的原生作用下）。
当 Lua 通过 gc 元方法回收一个完整的 userdata 时， Lua 调用这个元方法并把 userdata 标记为已终止。 等到这个 userdata 再次被收集的时候，Lua 会释放掉相关的内存。

lua_touserdata

void *lua_touserdata (lua_State *L, int index);
如果给定索引处的值是一个完整的 userdata ，函数返回内存块的地址。 如果值是一个 light userdata ，那么就返回它表示的指针。 否则，返回 NULL 。

Lua 调用 C++类要点：
1. 为此类建立一个全局表，表名为类名 tbClass；

```
lua_newtable(L);
int methods = lua_gettop(L);

lua_pushstring(L, T::className);

lua_pushvalue(L, methods);
lua_settable(L, LUA_GLOBALSINDEX);
```

2．注册一个 key 为 T::className 的 metatable，并制定其中的一些成员，用于之后生成的 userdata。

```
// 这个表用于 userdata(T 的对象)的 metatable

        luaL_newmetatable(L, T::className);
        int metatable = lua_gettop(L);

            // metatable["__index"] = tbClass

        lua_pushliteral(L, "__index");

        lua_pushvalue(L, methods);
        lua_settable(L, metatable);

// metatable["__tostring"] = tostring_T

        lua_pushliteral(L, "__tostring");

        lua_pushcfunction(L, tostring_T);

        lua_settable(L, metatable);


    // metatable["__gc"] = gc_T

        lua_pushliteral(L, "__gc");

        lua_pushcfunction(L, gc_T);
        lua_settable(L, metatable);
```

3． 为此表指定成员，每个成员的 key 为类的成员函数名，Value 为一个带有闭包的统一函数。比如 tbClass[FunName] = thunk，之后可以根据闭包得到具体是调用到哪个函数。闭包中有函数名和相应函数的组合结构(以 lightuserdata 的形式赋给闭包)。这些类成员函数参数都必须包括 lua_State，因为它需要的参数都会在 lua 堆栈中。

```
// 为 tbClass 填充成员函数

        for (RegType *l = T::methods; l->name; l++)

        {

            /* edited by Snaily: shouldn't it be const RegType *l ... ? */

            lua_pushstring(L, l->name);


// 把(函数名,函数地址)pair 以 lightuserdata 的形式作为 C closure 的 upvalue 入栈

            lua_pushlightuserdata(L, (void*)l);


// 把一个新的 C closure 压入堆栈。为 upvalue 的个数，并指定回调函数统一为 thunk    lua_pushcclosure(L, thunk, 1);



// tbClass[FunName] = Function
```

```
lua_settable(L, methods);
        }
```

4. 创建 C 对象给脚本使用 b = Account.new(Account, 30); new 是 tbClass 下的一个函数(另外指定的，不会掉到 thunk，这一句会调用到 C 的一个函数，里面会生成一个 C 对象，然后创建一个 userdata 用于关联到这个新生成的 C 对象。最后为这个 userdata 绑定上我们上面注册为 T::classname 的 metatable。因为定制了 metatable 的__index 成员，所以当 userdata 找不到的成员会去调用__index，因为之前我们把__index 绑定到 tbClass，所以也会调用到 tbClass 的相应成员。

```
// 创建一个新的 T 对象，并创建一个基于 userdataType 的 userdata，其中保护了指向 T 对象的指针
static int new_T(lua_State *L)
{
    lua_remove(L, 1);   // use classname:new(), instead of classname.new()
    T *obj = new T(L); // call constructor for T objects
    userdataType *ud =
        static_cast<userdataType*>(lua_newuserdata(L, sizeof(userdataType)));
    ud->pT = obj; // store pointer to object in userdata
    luaL_getmetatable(L, T::className); // lookup metatable in Lua registry
    lua_setmetatable(L, -2);
    return 1; // userdata containing pointer to T object
}
```

5. 当脚本中指定函数被调用的时候，比如 b:deposit(50.30)的时候，b 是 userdata，它的 metatable 的__index 和 tbClass 绑定(见 4)，所以会调用到 tbClass 的相应成员，就是之前关联的 thunk：这个时候 L 的堆栈里面有这个函数的两个参数，一个是 b 本身，一个是 50.30。b 是 userdata，可以根据它取出对象的指针。见第 4 步。另外函数被调用的时候，它相关的 upvalue 也可以取得到，见步骤 3。有了对象指针和相应的函数，调用也不为难了，记住参数 50.30 是保存在堆栈中传给类的成员函数来取得。

```
// 所有成员函数都会调用到这里，然后根据 upvalue 来执行具体的成员函数
static int thunk(lua_State *L)
{
    // stack has userdata, followed by method args
    T *obj = check(L, 1); // the object pointer from the table at index 0.
    lua_remove(L, 1); // remove self so member function args start at index 1
    // get member function from upvalue
    RegType *l = static_cast<RegType*>(lua_touserdata(L, lua_upvalueindex(1)));
    return (obj->*(l->mfunc))(L); // call member function
}
```

```
// 根据指定位置 narg 获得对象指针,这个 userdata 是在 new_T 的时候创建的
static T *check(lua_State *L, int narg)
{
    void *pUserData = luaL_checkudata(L, narg, T::className);
    userdataType *ud = static_cast<userdataType*>(pUserData);    // 这个是函数的 upvalue
    if(!ud)
        luaL_typerror(L, narg, T::className);
    return ud->pT;
}
```