

第四章 脚本模块

4.1 脚本介绍

一般学了前 3 章后，基本可以写出自己的服务端了，单纯用 c++，适合代码量不大的工程（早期服务端大多都是硬编码，程序逻辑游戏 AI 都是写死的，修改了还要重新编译）。一旦代码量太多，或者项目代码编写人员多，人员编程水平参差不齐，c++对程序员要求比较苛刻，不熟练的程序员经常会写出 bug 代码或内存泄漏等代码，对项目进程是非常不利的，为了项目能更快的编写出高质量的代码，脚本应运而生。

比如写服务端，有的用 python 就可以胜任，由于 py 的库丰富，很容易编写出高质量可靠的程序，而有的混合 c++和脚本，c++处理内核搭建框架，脚本负责处理逻辑事务。这样只需要几个高级程序员负责框架接口，一些程序员负责编写脚本即可。可以大大降低项目成本。一举多得。现在很多公司都在使用 lua 编写程序，上手快，技术人员比较便宜。

本章将使用 lua 脚本和 c++混合编程，讲前几章的纯 c++服务端改造成 lua 脚本

4.2 LUA

Lua 是一个小巧的脚本语言。其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。Lua 由标准 C 编写而成，几乎在所有操作系统和平台上都可以编译，运行。Lua 并没有提供强大的库，这是由它的定位决定的。所以 Lua 不适合作为开发独立应用程序的语言

Lua 脚本可以很容易的被 C/C++ 代码调用，也可以反过来调用 C/C++的函数，这使得 Lua 在应用程序中可以被广泛应用。不仅仅作为扩展脚本，也可以作为普通的配置文件，代替 XML,ini 等文件格式，并且更容易理解和维护。Lua 由标准 C 编写而成，代码简洁优美，几乎在所有操作系统和平台上都可以编译，运行。一个完整的 Lua 解释器不过 200k，在目前所有脚本引擎中，Lua 的速度是最快的。这一切都决定了 Lua 是作为嵌入式脚本的最佳选择。（摘自网络）

如果要全面讲解 Lua 的话可以写一本书，限于篇幅，本章着重讲解实际使用中必须用到的知识，其他的相关知识读者可以通过其他方式学习。

下面开始学习 lua 的相关操作。

4.3 lua 嵌入 c/c++

官网 <http://www.lua.org/>

下载 <http://www.lua.org/download.html>

当前版本是 5.3.1

我使用 5.1.4

解压后如图

名称	大小	压缩后大小	类型	修改时间	CRC32
..			文件夹		
doc			文件夹	2015/4/24 2:25	
etc			文件夹	2015/4/24 2:29	
src			文件夹	2015/4/24 2:30	
test			文件夹	2015/4/24 2:25	
COPYRIGHT	1,528	829	文件	2008/1/18 1:59	8A23B6FF
HISTORY	7,907	3,199	文件	2007/3/27 0:05	4D8A1F57
INSTALL	3,868	1,672	文件	2006/10/27 0:26	6D46E7...
Makefile	3,695	1,502	文件	2008/8/12 8:40	1C2ADF...
README	1,378	743	文件	2007/3/29 8:19	7590D7E7

src 目录里的就是 lua 的源码，所以很方便的嵌入到程序里。

新建一个 c++控制台工程，把除 lua.c 和 luac.c 了以外，src 里的代码文件都复制进新工程，如果顺利，编译一下即可成功。说明我们 lua 代码没有编译错误。

下一节开始介绍 lua 和 c++混合编程。

4.4 lua c/c++之 hello world

首先来个最简单的例子，lua helloworld，c++工程运行 lua 脚本，lua 脚本就一句代码 `print("hello world!lua!")`，功能就是打印输出字符串。

打开上一节的 lua 工程，

添加如下代码

```
void hello_lua()
{
    //Create new lua state
```

```

lua_State * L = lua_open();
luaL_openlibs( L );

char* filename= "hello.lua";

// Load script file
if (luaL_dofile(L, filename) )
    printf( "运行lua文件[%s]失败: ",filename);

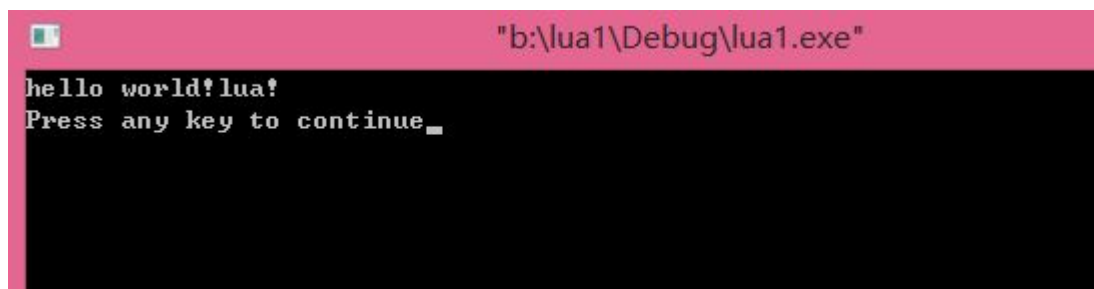
lua_close(L);          // Close the lua state
}

```

然后在 main 函数里调用 hello_lua();

项目目录下建立一个 hello.lua 文件，写入 lua 代码：
 print("hello world!lua!")

运行 lua 工程，结果如图



运行 hello.lua 成功打印出了一行字符串。

Lua的初始化很简单, lua_State * L = lua_open(); 打开一个lua状态句柄, 所有lua的函数都需要这个lua状态句柄, 可以认为打开了一个lua环境。

luaL_openlibs(L); 加载lua的内部库。 数学库、table库、字符串库、I/O库等。

luaL_dofile 加载并运行 lua 脚本。

lua_close(L); 关闭 lua 环境

4.5 lua 基本语法

Lua 没有 "main" 程序的概念，它只能嵌入到宿主程序中工作，宿主程序可以通过调用函数执行一小段 Lua 代码，可以读写 Lua 变量，可以注入 C 函数让 Lua 代码调用。

注释

```
-- 单行注释
```

流程控制

```
ret=true
if ret ==true then
    print( " true " )
else
    print ( " not true" )
end
```

Lua 不用分号，也不用花括号，而是使用 if,then、do 之类的语句。类似 asp ， python 语法。

变量

直接 var = value 即可，无需声明变量类型。默认全局变量。你可以一次赋值多个变量：var1, var2, var3 = val1, val2, val3

nil 表示“无”，包括未定义的变量。

只有 nil 和 false 为假，其他值均为真。

运算符

Lua 具有大多数标准的运算符,下面只列出常用的

```
..
and、or、not
<    >    <=    >=    ~=    ==
```

+ -
* / %

不等号为~=

^指数运算,而不是 xor

#获取长度

..连接字符串

循环

while

```
i = 1
```

```
while i <= 5 do
```

```
    i = i + 1
```

```
    print(i)
```

```
end
```

break 循环跳出

没有 continue 关键字

函数

```
function hello()
```

```
    print("hello")
```

```
end
```

```
function add(a,b)
```

```
    local z = a+b
```

```
    return z
```

```
end
```

```
function add2(a,b)
```

```
    return a+b
```

```
end
```

```
function hello()
```

```
    print("hello")
```

end

hello() --调用函数

print(add(1,2)) --调用 add 进行加法运算

函数的定义没有先后顺序的依赖问题。

局部变量需要使用 local 关键字声明。

函数可以返回多个值。

不直接支持参数的默认值。

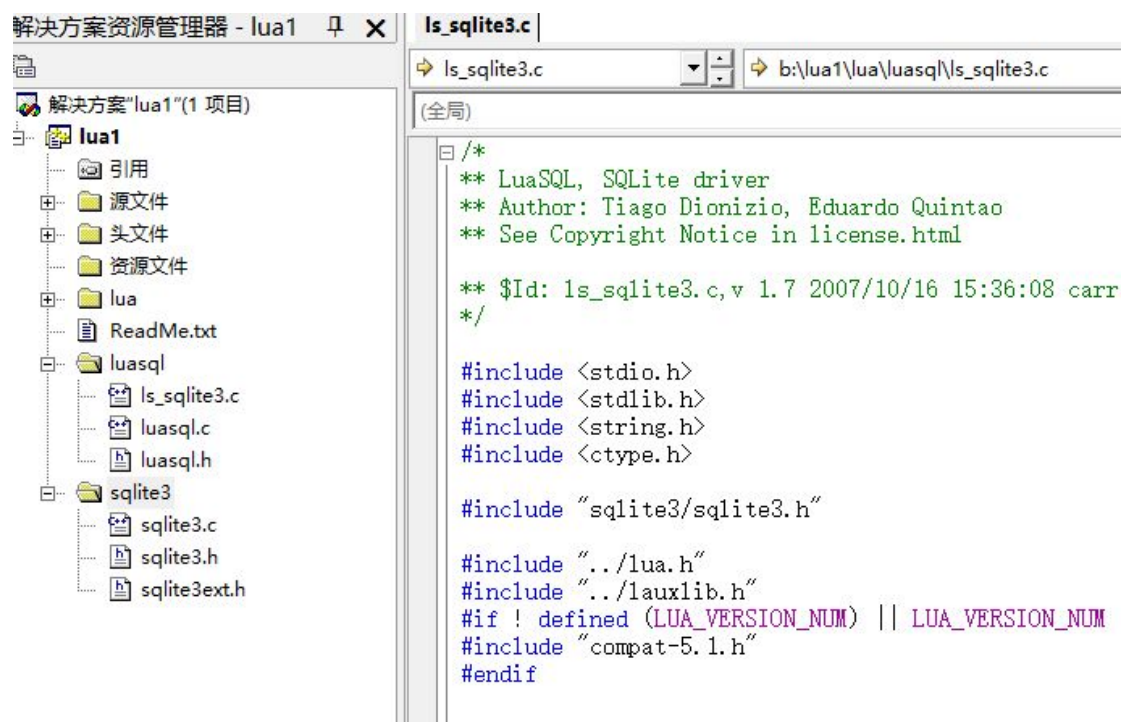
4.6 LUA 之数据库扩展

Lua 官方没有提供数据库模块，所以要么自己写一个数据库模块，要么把现有的 c++数据库接口导出提供给 lua 调用，我不想写一个，网络上已经有人写了，老外写的 Luasql. 提供 odbc,mysql,sqlite,postgres 多种 db 访问接口，很方便，我们把它整合进 lua 就有了数据库访问的功能了。

下面来看步骤吧。

下载 luasql-2.1.1.zip，luasql 有多种使用方法。一种是编译成 sql 插件，lua 脚本加载 sql 插件，还有一种是把 luasql 源码加入 lua 的系统库，lua 脚本运行时，即拥有数据库的访问能力。我采用编译进系统库的方法。

解压出来，它的 src 文件夹就是 luasql 源码。把源码复制到我们的 lua 工程项目里，由于我只需要访问 sqlite3 数据库的功能，所以只选择 ls_sqlite3.c luasql.c luasql.h 3 个文件。由于 luasql 的 sqlite3 引用了 sqlite3 数据库功能，所以还要导入 sqlite3 源码，即 sqlite3.c sqlite3.h sqlite3ext.h，添加完成后，如图所示



有的目录改变，所以代码头文件包含路径要对应修改一下,否则找不到头文件

Lualib.h 添加 如下代码

```
#define LUA_SQLLITE3LIBNAME "sqlite3"  
LUALIB_API int (luaopen_luaql_sqlite3) (lua_State *L);
```

Linit.c lualibs追加lib定义 添加 如下代码

```
{LUA_SQLLITE3LIBNAME, luaopen_luaql_sqlite3},
```

这两步操作就把 luaql_sqlite3 定义为了名叫 sqlite3 的 lua 内部库了

现在可以编译一下，看代码是否无措。

运行生成。如图

输出
生成
ls_sqlite3.c b:\lua1\lua\luaql\ls_sqlite3.c(425) : warning C4244: “函数” : 从 “sqlite3_int64” 转换到 “lua_Number”，可能丢失数据正在链接... LINK : LNK6004: 没有找到 Debug/lua1.exe 或上一个增量链接没有生成它：正在执行完全链接 生成日志保存在“file://b:\lua1\Debug\BuildLog.htm”中 lua1 - 0 错误, 1 警告

Ok,下面开始写 lua 脚本，来测试 sqlite3 操作， 看代码是否工作。

```
function fetch(c)  
    return c:fetch ({}, "a")  
end  
function next(c,row)  
    return c:fetch (row, "a")  
end  
  
env = luaql.sqlite3()  
db = env:connect("test.db")  
  
db:execute [[ drop TABLE people ]]  
  
db:execute [[CREATE TABLE people(name text, sex text)]]  
  
db:execute [[INSERT INTO people VALUES('张三','男')]]  
db:execute [[INSERT INTO people VALUES('李四','女')]]
```

```

rs = db:execute [[SELECT * FROM people]]
row = fetch( rs )
while row do
    print (string.format("Name: %s, sex: %s", row.name, row.sex))
    row = next ( rs,row )
end

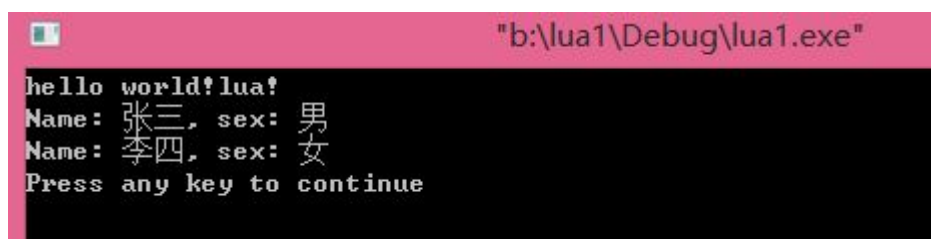
rs:close()

db:close()
env:close()

```

代码添加到上个工程的 hello.lua 文件

运行工程，结果如图



The screenshot shows a Lua debug console window titled "b:\lua1\Debug\lua1.exe". The output text is as follows:

```

hello world! lua!
Name: 张三, sex: 男
Name: 李四, sex: 女
Press any key to continue

```

数据库 sqlite3 操作无误。我们 lua 有了数据库的访问能力了。

代码其实很简单。

`env = luasql.sqlite3()` 这句是创建一个 sqlite3 的 env 环境对象。

`db = env:connect("test.db")` env 环境对象打开数据库 test.db, 得到一个 db 对象

`db:execute (" drop TABLE people ")` db 对象 execute 方法。执行 sql 语句。对于不返回记录集的 sql 语句（如 CREATE, DELETE），返回操作影响的记录个数。对于返回记录集的 sql 语句（如 SELECT），返回一个 cursor 对象（记录集）执行错误，则返回 nil，空。

读者可以同理添加自己需要的数据库接口类型，比如 mysql, odbc, postgresql, oracle，不再赘述。

4.7 LUA C 服务端框架

Lua 默认不提供网络模块，所以要么使用别人的 lua 扩展库 socket，要么自己将

c++的网络实现类导出接口给 lua 使用，采用 c++作为外壳程序，使 lua 拥有网络程序的能力。

本节用 lua 和 c 实现一个简易框架。Lua 有 StartUp Update Shutdown 三个函数,c 通过分别调用来实现 lua 的服务端流程。

下面是 lua c 服务端框架主要代码

```
extern "C" {
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
}

lua_State * L;
bool g_bTerminate=false;//退出标志

void WRITE_LOG(std::string s)
{
    std::cout<< s <<std::endl;
}

//调用lua函数
bool call_lua(const char* fun)
{
    lua_getglobal( L, fun );
    if( lua_pcall(L,0,0,0) )
    {
        std::string errStr =fun;
        errStr+="  err:";
        errStr+=lua_tostring( L, -1 );
        WRITE_LOG( errStr );
        lua_pop( L, 1 );
        return false;
    }
    return true;
}

int G_Exit(lua_State * L)
{
    g_bTerminate=true;
```

```

        return 0;
    }

void init_Lua()
{
    //初始化lua环境
    L = luaL_newstate();
    luaL_openlibs( L );

    //注册一个函数给lua,提供lua退出函数
    lua_pushcfunction( L, G_Exit );
    lua_setglobal( L, "G_Exit" );

    std::string ResultStr;

    //加载并启动
    if(luaL_dofile(L,"sim.lua"))
    //if( luaL_loadfile(L,"sim.lua") || lua_pcall(L,0,0,0) )
    {
        std::string errStr = "加载失败:";
        errStr+=lua_tostring( L, -1 );
        //WRITE_LOG( errStr );
        lua_pop( L, 1 );
    }

    //调用lua 的StartUp
    if(! call_lua( "StartUp" ) )
    {
        //("启动失败");
    }
}

void Update_Lua(    )
{
    //调用lua 的Update
    if(! call_lua( "Update" ) )
    {
        // "Update err: ";
    }
}

```

```
}
```

```
void Shutdown_Lua(    )  
{  
    //调用lua 的Shutdown  
    if(! call_lua( "Shutdown" ) )  
    {  
        // "Shutdown err:";  
  
    }  
  
    //关闭lua环境  
    lua_close(L);  
}
```

```
//-----  
// Main  
//-----
```

```
int _tmain(int argc, _TCHAR* argv[])  
{  
    //初始化lua  
    init_Lua();  
  
    while (!g_bTerminate)  
    {  
        //不断更新lua  
        Update_Lua();  
  
        Sleep(1000);  
    }  
  
    //关闭lua  
    Shutdown_Lua();  
  
    return 0;  
}
```

sim.lua 代码如下

```
print( "hello world!lua!" )
```

```
function StartUp( )  
    print("StartUp")
```

```
    env = luasql.sqlite3()  
    db = env:connect("test.db")
```

```
end
```

```
TickCount=1
```

```
function Update( )  
    print("Update "..TickCount )
```

```
    db:execute [[ drop    TABLE people ]]
```

```
    db:execute [[CREATE TABLE people(name text, sex text)]]
```

```
    db:execute [[INSERT INTO people VALUES('张三','男')]]
```

```
    db:execute [[INSERT INTO people VALUES('李四','女')]]
```

```
    rs =  db:execute [[SELECT * FROM people]]
```

```
    row = fetch( rs )
```

```
    while row do
```

```
        print (string.format("Name: %s, sex: %s", row.name, row.sex))
```

```
        row =  next ( rs,row )
```

```
    end
```

```
    rs:close()
```

```
    TickCount=TickCount+1
```

```
    if TickCount>5 then
```

```
        G_Exit()
```

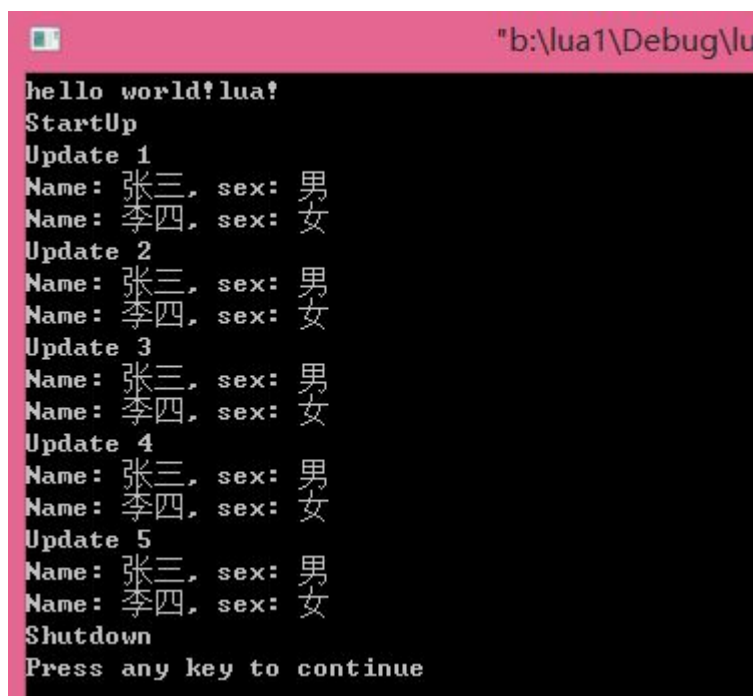
```
    end
```

```
end
```

```
function Shutdown()  
    print("Shutdown")
```

```
    db:close()  
    env:close()  
end
```

```
function fetch(c)  
    return c:fetch ({}, "a")  
end  
function next(c,row)  
    return c:fetch (row, "a")  
end
```



```
hello world! lua!  
StartUp  
Update 1  
Name: 张三, sex: 男  
Name: 李四, sex: 女  
Update 2  
Name: 张三, sex: 男  
Name: 李四, sex: 女  
Update 3  
Name: 张三, sex: 男  
Name: 李四, sex: 女  
Update 4  
Name: 张三, sex: 男  
Name: 李四, sex: 女  
Update 5  
Name: 张三, sex: 男  
Name: 李四, sex: 女  
Shutdown  
Press any key to continue
```

如图， 首先调用 StartUp 来实现 lua 脚本的内部相关数据初始化， 然后 Update 里不断更新 lua， 程序的主要更新逻辑就写在这里， 最后连续 update 5 次后， 调用了 G_Exit， 我们导出给 lua 的函数， 调用 G_Exit 就会退出， 使程序的主循环退出， 最终调用 Shutdown_Lua， 停止 lua 脚本。

把这个封装成 c++ 的风格也很简单。现在框架有了， 但是网络模块还没有添加，

4.8 LUA 框架之网络模块

上一节的框架都有了，但是还没有网络功能。我打算使用我们的 c++ 网络引擎整合进 lua。

网络引擎采用全局函数的形式导出给 lua 使用。

```
//初始化lua环境
L = lua_open();
luaL_openlibs( L );

//注册一个函数给lua,提供lua退出函数
lua_pushcfunction( L, G_Exit );
lua_setglobal( L, "G_Exit" );

//网络api
lua_pushcfunction( L, G_NetInit );
lua_setglobal( L, "G_NetInit" );

lua_pushcfunction( L, G_NetSend );
lua_setglobal( L, "G_NetSend" );

lua_pushcfunction( L, G_NetShutdown );
lua_setglobal( L, "G_NetShutdown" );

lua_pushcfunction( L, G_NetUpdate );
lua_setglobal( L, "G_NetUpdate" );

lua_pushcfunction( L, G_NetConnNums );
lua_setglobal( L, "G_NetConnNums" );
```

其实核心的只要 3 个函数搞定。G_NetInit 对应调用网络引擎 init 函数, G_NetSend 对应网络引擎的 Send 函数, G_NetUpdate 对应网络更新, G_NetShutdown 关闭网络引擎。

关于网络引擎的数据回调，则是先调用 c 的 callback，callback 再调用 lua 的全局函数 function OnRecvData(idx,data,len) 参数分别对应 callback 参数。连接断开则会调用 lua 的全局函数 function OnClose(idx) 参数是客户索引。

下面给出 G_NetInit 的实现

c 编写函数提供给 lua 使用，需要遵守 lua 的规范，c 函数必须类似

```
typedef int (*lua_Cfunction)(lua_State* L)
```

返回值 int 表示函数调用完后返回多少个参数个数，函数只有一个参数 lua_State*

L 一切数据都要通过这个 lua 状态机获得。

lua_gettop(L);可以获得lua调用该函数时传递的参数个数
lua_tostring(L , 1); 表示获得第一个传递的string参数，所以格式还要对应不能选错
lua_tointeger(L , 2); 表示获得第二个传递的int参数
CreateNetServer 按参数创建一个网络引擎。
lua_pushlightuserdata(L,net); 把创建的网络引擎指针 push 作为返回值。

return 1; 因为我们只返回一个值，所以函数返回值为1

```
int G_NetInit( lua_State * L )
{
    INetServer* net;//网络引擎

    try
    {

        int paras = lua_gettop( L );
        if( paras < 2 )
        {
            std::ostringstream os;
            os << "G_NetInit: parameter missing. " << 2 << " " << paras;
            WRITE_LOG( os.str() );
        }
        else
        {
            std::string ip=lua_tostring( L , 1 );
            int port = lua_tointeger( L , 2 );

            net= CreateNetServer(
                (char*)ip.c_str() ,    //"0.0.0.0" ,
                port,
                RecvDataCall,2000); //创建网络引擎

        }

    }
    catch( ... )
    {
        WRITE_LOG( "G_NetInit: unknown exception" );
    }

    lua_pushlightuserdata(L,net);
```

```

        return 1;
    }

int G_NetUpdate( lua_State * L )
{
    try
    {
        INetServer* net= (INetServer*)lua_topointer( L , 1 );
        if (net)
        {
            net->Update();
        }
    }
    catch( ... )
    {
        WRITE_LOG( "G_NetUpdate: unknown exception" );
    }

    return 0;
}

```

G_NetUpdate很简单，只需要获得一个参数，即网络引擎指针。然后net->Update();就完成了G_NetUpdate的任务。不需要返回值给lua,所以return 0; 即可

下面是 sim.lua 代码

```
print( "hello world!lua!" )
```

```

function StartUp( )
    print("StartUp")

    --连接 sqlite3 数据库
    env = luasql.sqlite3()
    db = env:connect("test.db")

    --启动网络
    net= G_NetInit("0.0.0.0",123)
    if net ~=nil then
        print( "G_NetInit  ok" )
    else

```



```

        print( "G_NetInit  err" )
    end

end

TickCount=1
function Update( )
    print("Update "..TickCount )

    G_NetUpdate(net)

    print("当前连接数: "..G_NetConnNums(net) )

    db:execute [[ drop   TABLE people ]]

    db:execute [[CREATE TABLE people(name text, sex text)]]

    db:execute [[INSERT INTO people VALUES('张三','男')]]
    db:execute [[INSERT INTO people VALUES('李四', '女')]]


    rs =  db:execute [[SELECT * FROM people]]
    row = fetch( rs )
    while row do
        print (string.format("Name: %s, sex: %s", row.name, row.sex))
        row =  next ( rs,row )
    end

    rs:close()


    --TickCount=TickCount+1
    --if TickCount>5 then
    --  G_Exit()
    --end
end

function Shutdown( )
    print("Shutdown")

    G_NetShutdown(net)

    db:close()

```

```

        env:close()
end

function fetch(c)
    return c:fetch ({}, "a")
end
function next(c,row)
    return c:fetch (row, "a")
end

--数据接收
function OnRecvData(idx,data,len)

    print("-----OnRecvData-----")
    print("-----idx["..idx.."]" )
    print("-----len["..len.."]--data[" .. data .. "]------")

    --echo
    G_NetSend(net,idx,len,data )

end

-- 连接断开
function OnClose(idx)
    print( "OnClose idx:"..idx );
end

```

下面来测试一下 网络接口是否工作正常，先运行本 lua 服务端,再运行 第一章
 \1.12\netclient\debug\netclient.exe 我的测试结果如下

```
F:\code\doc\source\第一章\1.12\netclient\Debug\netclient.exe
createNetClient ok! 连接ok
ecv size:128 a:2 time:1484
ecv size:128 a:3 time:2500
ecv size:128 a:4 time:3516
ecv size:128 a:5 time:4516
ecv size:128 a:6 time:5515
ecv size:128 a:7 time:6516
ecv size:128 a:8 time:7531
ecv size:128 a:9 time:8532
ecv size:128 a:10 time:9547
ecv size:128 a:11 time:10562
ecv size:128 a:12 time:11563
ecv size:128 a:13 time:12578
ecv size:128 a:14 time:13578
ecv size:128 a:15 time:14584

Name: 张三, sex: 男
Name: 李四, sex: 女
Update 1
-----OnRecvData-----
-----idx[456]
-----len[128]--data[0]
当前连接数: 1
Name: 张三, sex: 男
Name: 李四, sex: 女
Update 1
-----OnRecvData-----
-----idx[456]
-----len[128]--data[0]
```

Lua 服务端 OnRecvData 处理数据里只简单的 G_NetSend 原文返回给客户端，即 echo。客户端也收到了服务端发来的 echo 信息，并打印了出来。

网络模块基本完工了，但是还缺少一个数据解包打包功能，lua 并不知道 OnRecvData 的 data 究竟是什么数据，该怎么读取，

4.9 数据解包打包

我编写了一个 lua 扩展库，lua buffer，buffer 提供一些方法，实现对数据的解包打包。这样 lua 里 OnRecvData 的 data 数据指针就可以解析出具体数值了。

项目添加 luabuffer.h luabuffer.c

```
#include "luabuffer.h"
```

初始化lua环境后 添加一句luaopen_buffer(L); 即可打开我们的buffer扩展库。

对于 结构体

```
struct MSG_Login
{
    int cmd; //命令
    char name[20]; //用户名
    char pwd[20]; //密码
};
```

对应 解包

```
b = buffer.new()
b:init( data,len) 传入data数据指针和data数据长度
cmd=b:read_int() 命令
name=b:read_string(20) 用户名，参数是字符数组的大小
```

```
pwd=b:read_string(20) 密码
```

对应 打包

```
ret = buffer.new()
ret:write_int(1)
ret:write_string("user1",20)  参数2是字符数组的大小
ret:write_string("123456",20)
```

网络发送buffer 数据指针和数据长度

```
NetSend(    ret:size(),ret:getptr() )
```

我暂时只实现了 int 和 char 数组的读写操作，可以满足一般需求了，读者可以自行添加其他的数据类型读写，比如 float double char short 等等。

4.10 Lua 服务端消息处理

继续完善我们的 lua 服务端，有了解包功能后，接下来开始编写 PacketHandler，消息处理模块。

新建一个PacketHandler.lua

写入如下代码

```
--对应 msgdef.h 的消息 id
C2S_LOGIN=0  --登陆
S2C_LOGIN_Ret=1 --登陆 ret
```

--消息处理

```
function ParsePacket( idx,data,len)
    print("ParsePacket")

    local b = buffer.new()
    b:init( data,len)
    cmd=b:read_int()

    print("cmd:" .. cmd )
```

```

--登录
if cmd==C2S_LOGIN then
    On_C2S_LOGIN(idx,b)
    return
end

error("无法识别的 cmd:",cmd)

end

function On_C2S_LOGIN( idx,b)

    print("---On_C2S_LOGIN---")

    --int cmd; //命令  C2S_LOGIN
    --char name[20]; //用户名
    --char pwd[20]; //密码

    --int cmd; //命令  S2C_LOGIN_Ret
    --int errcode; //1=登陆成功 , 0=登陆失败
    --char pwd[20]; //密码

    --打印 数据
    local name=b:read_string(20) --参数是字符数组的大小
    local pwd=b:read_string(20)
    print("name:" .. name ,#name )
    print("pwd:" .. pwd ,#pwd )

    --简单登陆检查
    if name=="abc" and pwd=="123456" then

        print("登陆成功" )

        --登陆成功
        local ret = buffer.new()
        ret:write_int(S2C_LOGIN_Ret) --cmd
        ret:write_int(1) --errcode
        ret:write_string ("pwdpwd",20)
        --发送给客户端
        G_NetSend(net,idx,ret:size(),ret:getptr())
    end
end

```

```

        print("---On_C2S_LOGIN end---")
        return
    end
    print("登陆失败" )

    --登陆失败
    local ret = buffer.new()
    ret:write_int(S2C_LOGIN_Ret)    --cmd
    ret:write_int(0) --errcode
    ret:write_string ("pwdpwd",20)
    --发送给客户端
    G_NetSend(net,idx,ret:size(),ret:getptr())

    print("---On_C2S_LOGIN end---")

end

```

sim.lua 的 OnRecvData 使用 ParsePacket(idx,data,len)一句即可。

运行本节的 lua 项目，然后运行 source\第二章\2.2\netclient\Debug\netclient.exe

运行结果如下

```

C:\Users\...
CreateNetClient ok! 连接ok
PacketHandler::ParsePacket msg 1
登陆成功!

f:\code\doc\source\第四章\5.0\lua1\Debug
Name: 李四, sex: 女
Update 1
-----OnRecvData-----
-----idx[468]
-----len[441]---data[
ParsePacket
cmd:0
---On_C2S_LOGIN---
name:abc 3
pwd:123456 6
登陆成功
---On_C2S_LOGIN end---

```

可以看见客户端收到一包数据，打印了一句 登陆成功。

这样我们成功把纯 c++服务端改造成了 lua/c++混合服务端（Lua 服务端）。

4.11 总结

本章我们学会了编写 Lua 服务端，使得编码难度降低到脚本级别，任何新手也可以写出有质量的运行稳定的代码。但是 Lua 依然有许多新知识需要学习，本章的框架也依然有改进的空间，比如协议目前是 buffer 打包解包，可以考虑目前流行的 Google Protocol Buffer，或者使用 json xml。

2016.1.3 作者：司马威
作者博客 <http://blog.csdn.net/smwhotjay>

编程交流 q 群: 316641007