Lua 基础学习

要学习 tolua++的使用，首先也得先学会 lua 的基本语法和 lua 与 c/c++之间的接口调用。本次总结意在把 lua 的一些基本的用法总结归纳一下，好让大家可以在短时间内对 lua 有一个简单的了解，达到可以看懂 lua 代码的目的。

注：

注释：单行注释:--

多行注释：--[[--]]

lua 大小写敏感

1. 基本类型

1.1 nil

它就相当于 c++里边的 NULL，但又与 NULL 不同，lua 中的全局变量变量如果没有赋初值，则它的值就默认等于 nil，如果想要删除一个全局变量变量，则可以给它赋值 nil。

1.2 booleans

它的取值跟 c++一样，有 true 和 false 两种。但是在 lua 中有一个特别的地方，在判断语句的时候，除了 false 和 nil 为假，其它的值都为真，这点 c++程序员应该格外注意，0 和空串也为真。

1.3 numbers

在 lua 中 number 代表实数，他强大的可以代表任何你能想象到的数字（玩笑开大啦），在应用中去探索吧。例：整数、浮点、指数似乎还有复数等。

1.4 strings

lua 中的 string 就类似 c++中的 const char[]，不过人家是通过 lua 来自动分配内存和释放内存的。另外注意那个 const，它的值是不允许修改的；

它支持转义字符，类似 c++的转义字符；

lua 会自动在 string 和 numbers 之间自动进行类型转换，当一个字符串使用算术操作符时，string 就会被转成数字，如果 string 中的内容并非 number 的时候，可会报错哦；

在 lua 中字符串的连接符是..，如果要连接的是数字，则要在..和数字间留一个空格；

另外 lua 还提供 tonumber()和 tostring()函数，支持数字和字符串间的转换，不过在字符串转换数字的时候，如果字符串不是正确的数字 tonumber()会返回 nil。

1.5 functions

在 lua 中，函数也可以作为变量保存，可以作为函数的参数和返回值，了解到这个，你就可以在你的 lua 代码中发挥自己的想象去实现一些强大的功能啦。

1.6 userdata

顾名思义，就是用户自己定义的类型，其实就是在 lua 和 c++交互过程中，c++定义的 类型。

2. 表达式

2.1 算数运算符

跟 c++一样，加减乘除啥的。

2.2 关系运算符

跟 c++类似，就是!=换为~=。

2.3 逻辑运算符

这个可比较特别哦，要多加注意，lua 的逻辑运算符有 not、and、or；

lua 中只有 false 和 nil 为假，其它都为真；

not a： a 为 false，值为 true，a 为 true，值为 false；

a and b： 如果 a 为 false，则取 a 值，否则取 b 值；

a or b： 如果 a 为 true，则取 a 值，否则取 b 值；

c++中的三目运算符 a?b:c 在 lua 中的实现：(a and b) or c。

2.4 连接运算符

.. -- 这个前边介绍过，就是连接两个字符串

2.5 优先级

类似 c++，按老套路用就行

2.6 表的构造

这是 lua 的重头戏，它的构造函数是{}。

table = {} -- 创建一个空表

表里边可以存放任意类型的成员，就把它当做一个 c++的类即可，以下聚几个例子

test = {“a”,”b”,”c”,”d”,”e”,”f”,”g”}

print (test[4]) -- d // 这里注意，lua 的下标操作是从 1 开始计算的

```
test = {x=1,y=2,z=3}
print (test.x,test.y,test.z)       --  1  2  3
test[1] = "a"
test1 = {"b","c"}
test1.a = test
print (test[1],test.x,test.y,test.z)  --  a  1  2  3
print (test1.a[1],test1.a.x,test1[1])  --  a  1  b
```

更多细节请参照 lua 官网

3. 基本语法

3.1 赋值

lua 可以对多个变量同时赋值，各变量和值用逗号分隔，赋值语句右边的值会依次赋给左边的变量；

当变量个数比值多时，多出的变量会赋值 nil；

当变量个数少于值时，多余的值会忽略；

例：

```
x,y = y,x      -- 交换 x 和 y 的值
a,b,c = 0,1    -- a=0,b=1,c=nil
a,b = 3,2,1    -- a=3,b=2
```

3.2 局部变量

lua 使用 local 创建局部变量，它的作用域就跟 c++类似，不过它没有{}，取而代之的是 do..end，如果想要设置一个局部变量，则可以将它定义在 do..end 当中；

lua 中使用局部变量有两个好处：

- 避免命名冲突
- 访问局部变量的速度比全局变量快

例：

```
do
local a = 2
b = 4
print (a,b)   -- 2       4
end
print (a,b)          -- nil      4
```

3.3 控制语句

3.3.1 if 语句

```
if conditions then
then-part
end;

if conditions then
then-part
else
else-part
end;
```

if conditions then

then-part

elseif conditions then

elseif-part

.. --->多个 elseif

else

else-part

end;

3.3.2 while 语句

while condition do

statements;

end;

3.3.3 repeat-until

repeat

statements;

until conditions;

这个可能比较眼生，其实类似 c++的 do..while 语句，即运行 statements 直到 conditions 为真。

3.3.4 for 循环

3.3.4.1 数值 for 循环

for var=exp1,exp2,exp3 do

loop-part

end

for 将用 exp3 作为 step 从 exp1（初始值）到 exp2（终止值），执行 loop-part。其中 exp3 可以省略，默认 step=1.

3.3.4.2 范型 for 循环

遍历表

for k in pairs(t) do print(k) end -- 遍历表 t 中的所有元素

for i,v in ipairs(a) do print(v) end

--遍历表 a，i 为其索引，v 为其值。

3.3.5 break 和 return

这个语法与 c++类似，break 是用来退出控制语句，return 是用来退出函数的。

注：在 lua 中，break 和 return 必须在一个块得结束部分出现，即它的后边出现的必须是 end、else、until，如果真的想要在某个语句位置使用它们，可以显式的使用 do..end 来包含它们实现想要的功能。例：do return end;

4. 函数

4.1 函数的定义

```
function name (param)
   return value
end;
```

4.2 多返回值

lua 是允许返回多个值的，如果函数有多个返回值，则对变量的赋值按照 3.1 赋值里边规定的方式赋值。

例：

```
function foo0 end
function foo1 return 'a' end
function foo2 return 'a' 'b' end

x,y = foo2()          -- x='a', y='b'
x = foo2()              -- x='a', 返回值'b'废弃
x,y,z = 10,foo2()        -- x=10, y='a', z='b'、
x,y = foo0()          -- x=nil, y=nil
x,y = foo1()          -- x='a', y=nil
x,y,z = foo2()         -- x='a', y='b', z=nil
```

4.3 可变参数

lua 函数可以接受可变数目的参数，和 C 语言类似在函数参数列表中使用三点（...）表示函数有可变的参数。lua 将函数的参数放在一个叫 arg 的表中，除了参数以外，arg 表中还有一个域 n 表示参数的个数

例：

```
printResult = ""

function test(...)
   print ("haveing " .. arg.n .. " param")
   for i,v in ipairs(arg) do
      printResult = printResult .. tostring(v)
   end;
   print (printResult)
```

```
end

test('a','b','c','d','e')
```

结果：

```
haveing 5 param
abcde
```

4.4 命名参数

lua 的调用过程中，参数是依次把实参传递给相应的形参的。但如果有时候我们很难记清参数的前后顺序，这个时候就可以使用命名参数。顾名思义，就是为每个参数起一个名字，调用的时候，只需要为这个名字的参数赋值即可。

例：

```
function rename (arg)
   return os.rename(arg.old, arg.new)
end

rename(old="temp.lua", new="temp1.lua") –调用的时候为命名参数赋值
```

4.5 闭包

首先明确一个概念，词法定界。当一个函数内部嵌套另一个函数定义时，内部的函数体可以访问外部的函数的局部变量，这种特征我们称作**词法定界**

所谓“闭包”，官方解释指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

是不是有点糊涂呢？举个例子：

```
function newCounter()
   local i = 0
   return function()     -- 无名函数
      i = i + 1
      return i
   end
end

c1 = newCounter()
print(c1())  --> 1
print(c1())  --> 2
c2 = newCounter()
print(c2())  --> 1
```

```
print(c1())  --> 3
print(c2())  --> 2
```

这段代码有两个特点，1. “匿名函数”嵌套在函数 newCounter 内，2. 函数 newCounter 把“匿名函数”返回了。

在执行完 c1 = newCounter()后，其实 c1 指向的是“匿名函数”，当我们执行 c1 的时候，就可以操作函数 newCounter 的局部变量 i，这个 i 我们称之为外部的局部变量（external local variable）或者 upvalue。我们再次声明一个变量 c2= newCounter()后，c1 和 c2 其实是建立在同一个函数上的，但是他们的局部变量 i 确是两个不同的实例，即我们创建了两个闭包。就是说：当函数 newCounter 的内部函数（即上例的“匿名函数”）被函数 newCounter 外的变量 c1 引用的时候，就创建了一个闭包。

闭包的应用场景：

- 保护函数内的变量安全。以最开始的例子为例，函数 a 中 i 只有函数 b 才能访问，而无法通过其他途径访问到，因此保护了 i 的安全性。
- 在内存中维持一个变量。依然如前例，由于闭包，函数 a 中 i 的一直存在于内存中，因此每次执行 c()，都会给 i 自加 1。

4.6 非全局函数

lua 中函数可以作为全局变量也可以作为局部变量

1. 表和函数放在一起

```
Lib = {}
Lib.foo = function (x,y) return x + y end
Lib.goo = function (x,y) return x - y end
```

2. 使用表构造函数

```
Lib = {
    foo = function (x,y) return x + y end,
    goo = function (x,y) return x - y end
}
```

3. lua 提供的另一种语法方式

```
Lib = {}
function Lib.foo (x,y)
    return x + y
end
function Lib.goo (x,y)
    return x - y
end
```

当我们将函数保存在一个局部变量内时，我们得到一个局部函数，也就是说局部函数像局部变量一样在一定范围内有效。定义局部函数的两种方式：

```
local f = function (...)
   ...
end

local function f (...)
   ...
end
```

看如下代码：

```
local fact = function (n)
   if n == 0 then
      return 1
   else
      return n*fact(n-1)   -- 此处 lua 不能识别 fact，因为是局部函数，lua 不能识别
   end
end
```

修改为先声明，如下：

```
local fact

fact = function (n)
   if n == 0 then
      return 1
   else
      return n*fact(n-1)
   end
end
```

C++调用 lua

本节我将一步一步带领大家完成 c++调用 lua 函数并接受 lua 的返回值，通过分析调用的方式来封装一个类，最终封装完成的类并不是最优的，但应该能够满足一般的项目中对 lua 调用的功能。不足之处欢迎大家给予指正。

1 基本概念

1.1 栈

c++调用 lua 是通过一个抽象的栈来实现数据的交换的。C++调用 lua 时，首先需要把 lua 函数需要的参数压入这个抽象的栈中，如果 c++想要从 lua 中获取数据，则 lua 需要先把数据压入栈中，然后 c++从

栈中取得需要的数据。Lua 是以严格的 LIFO 规则来操作栈的，即后进先出原则，而 c++则可以操作栈上的任何一个元素。

2 常用函数

- void lua_pushnil (lua_State *L);

往栈中压入空值

- void lua_pushboolean (lua_State *L, int bool);

往栈中压入布尔型值

- void lua_pushnumber (lua_State *L, double n);

往栈中压入 double 型数值

- void lua_pushlstring (lua_State *L, const char *s, size_t length);

往栈中压入字符串，但是该字符串中可以包含'\0'，字符串的长度为 length

- void lua_pushstring (lua_State *L, const char *s);

往栈中压入 c 风格的字符串，以'\0'结尾

- int lua_is... (lua_State *L, int index);

用来检查栈上的一个元素是否指定的类型

- int lua_type (lua_State *L, int index);

返回栈中元素的类型

在 lua.h 中定义了各个元素对应的常量：LUA_TNIL、LUA_TBOOLEAN、LUA_TNUMBER、LUA_TSTRING、LUA_TTABLE、LUA_TFUNCTION、LUA_TUSERDATA 以及 LUA_TTHREAD

- int lua_toboolean (lua_State *L, int index);

将元素值转换为布尔型

- double lua_tonumber (lua_State *L, int index);

将元素值转换为数值型

- const char * lua_tostring (lua_State *L, int index);

该函数返回一个指向 lua 栈内元素的指针，该指针是不允许修改的

切记这里的指针是指向栈上元素的，如果 lua 清空了栈内元素，则该指针就无效了。

- int lua_gettop (lua_State *L);

返回堆栈中的元素个数，它也是栈顶元素的索引

- void lua_settop (lua_State *L, int index);

设置栈顶为一个指定的值

如果原栈顶大于新栈顶，则顶部的值被丢弃，如果原栈顶小于新栈顶，则将多出的元素赋 nil。lua_settop(L,0)清空堆栈。

你也可以用负数索引作为调用 lua_settop 的参数；那将会设置栈顶到指定的索引。利用这种技巧，API 提供了下面这个宏，它从堆栈中弹出 n 个元素：

#define lua_pop(L,n) lua_settop(L, -(n)-1)

Ø void lua_pushvalue (lua_State *L, int index);

将指定元素的一个备份拷贝到栈顶

Ø void lua_replace (lua_State *L, int index);

移除指定元素，其上的其它元素依次下移

Ø void lua_insert (lua_State *L, int index);

将栈顶的元素移动到指定的索引处，其它元素依次上移

Ø void lua_replace (lua_State *L, int index);

将栈顶的值替换指定索引的元素，其它元素不变

3 一个简单的例子

该例子最终是通过 c++代码调用 lua 脚本实现加法的操作并把返回值打印到控制台上。以下是代码：

test.cpp

```
#include <iostream>
#include "lua.hpp"
using namespace std;

int main()
{
    int iRet = 0;
    double iValue = 0;
    lua_State * L = luaL_newstate();// 创建 lua 状态
    if (NULL == L)
    {
        cout << "luaL_newstate()没有足够的内存分配\n" << endl;
        return 0;
    }
    iRet = luaL_dofile(L, "test.lua");// 加载并运行 lua 脚本
    if (0 != iRet)
    {
        cout << "luaL_dofile() failed\n" << endl;
        return 0;
```

```
    }
    lua_getglobal(L, "add"); // 把函数名压入堆栈

    lua_pushnumber(L, 2);
    lua_pushnumber(L, 3);
    iRet = lua_pcall(L, 2, 1, 0); // 调用函数
    if (0 != iRet)
    {
        printf("lua_pcall failed:%s\n",lua_tostring(L,-1));
        return 0;
    }

    if (lua_isnumber(L, -1) == 1)
    {
        iValue = lua_tonumber(L, -1);
    }
    cout << iValue << endl;

    lua_close(L);
    return 0;
}
```

test.lua

```
function add (x,y)
    return x+y
end
```

编译：g++ -o test test.cpp -ltolua++

运行结果：5

以下是对该例子中用到的函数的解释：

- lua_State *luaL_newstate (void);

该函数用于创建一个新的 lua 状态，当内存分配错误的时候返回 NULL

- int luaL_loadfile (lua_State *L, const char *filename);

该函数加载 lua 文件，但不运行。

- int lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);

该函数用于调用 lua 的函数，nargs 是入参个数，nresults 是出参个数，如果 errfunc 为 0，表示把出错的信息放入堆栈中，如果不为 0，则该值代表一个错误处理函数在 lua 堆栈中的索引。函数成功时返回 0，错误时返回如下三个值：

LUA_ERRRUN：运行时错误；

LUA_ERRMEM：内存分配错误。对于这样的错误，Lua 不调用错误处理函数；

LUA_ERRERR：运行错误处理函数的错误。

- int luaL_dofile (lua_State *L, const char *filename);

加载并运行指定的文件，正确的时候返回 0，错误的时候返回 1。该函数在 lua 中被定义为一个宏：

(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))

- void lua_getglobal (lua_State *L, const char *name);

将函数名压入到堆栈中，它被定义为如下宏：

#define lua_getglobal(L,s) lua_getfield(L, LUA_GLOBALSINDEX, s)

- void lua_close (lua_State *L);

该函数销毁 lua 状态的所有对象，如果是守护进程或 web 服务器，则需要在使用完后尽快释放，以避免过大。

如果想直接验证 lua 脚本可以按如下步骤操作：

/home/tolua/test#lua

Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio

> dofile("test.lua")

> print (add(1,4))

5

4　封装

让我们仔细研究下这个简单的例子，是不是发现了很多共同点呢？我们完全可以利用 c++的特性对其进行封装，这样我们再调用 lua 的时候，就可以直接调用我们封装好的类啦。

CToLua.h

```
#ifndef    CTOLUA_H_
#define  CTOLUA_H_

#include "lua.hpp"
#include <iostream>
using namespace std;

class CToLua
{
```

```
public:
    CToLua();
    ~CToLua();
    bool loadLuaFile(const char* pFileName); //加载指定的 Lua 文件
    double callFileFn(const char* pFunctionName, const char* format, ...);
//执行指定 Lua 文件中的函数
    lua_State* getState();
private:
    int parseParameter(const char* format, va_list& arg_ptr);
    lua_State* m_pState;
};

#endif // CTOLUA_H_
```

```
#include "CToLua.h"

extern "C"
{
#include "tolua++.h"
}

CToLua::CToLua()
{
    m_pState = luaL_newstate();
    if (NULL == m_pState)
    {
        printf("luaL_newstate()没有足够的内存分配\n");
    }
    else
    {
        luaopen_base(m_pState);
    }
}

CToLua::~CToLua()
{
    if (NULL != m_pState)
    {
        lua_close(m_pState);
        m_pState = NULL;
    }
}
```

```
bool CToLua::loadLuaFile(const char* pFileName)
{
    int iRet = 0;
    if (NULL == m_pState)
    {
        printf("[CToLua::LoadLuaFile]m_pState is NULL.\n");
        return false;
    }

    iRet = luaL_dofile(m_pState, pFileName);
    if (iRet != 0)
    {
        printf("[CToLua::LoadLuaFile]luaL_dofile(%s) is error(%d)(%s).\n",
            pFileName, iRet, lua_tostring(m_pState, -1));
        lua_pop(m_pState, 1); // 及时清理堆栈
        return false;
    }

    return true;
}

double CToLua::callFileFn(const char* pFunctionName, const char* format,
 ...)
{
    int iRet = 0;
    double iValue = 0;

    int iTop = lua_gettop(m_pState);
    lua_pop(m_pState, iTop); // 清栈

    if(NULL == m_pState)
    {
        printf("[CLuaFn::CallFileFn]m_pState is NULL.\n");
        return 0;
    }

    lua_getglobal(m_pState, pFunctionName);

    va_list arg_ptr;
    va_start(arg_ptr, format);
    iRet = parseParameter(format, arg_ptr);
    va_end(arg_ptr);
```

```
    iRet = lua_pcall(m_pState, iRet, 1, 0);
    if (iRet != 0)
    {
        printf("[CLuaFn::CallFileFn]call function(%s) error(%d).\n", pFunc
tionName, iRet);
        return 0;
    }

    if (lua_isnumber(m_pState, -1) == 1)
    {
        iValue = lua_tonumber(m_pState, -1);
    }

    return iValue;
}

int CToLua::parseParameter(const char* format, va_list& arg_ptr)
{
    int iRet = 0;
    char* pFormat = (char*) format;
    while (*pFormat != '\0')
    {
        if ('%' == *pFormat)
        {
            ++pFormat;
            switch (*pFormat)
            {
            case 'f':
                lua_pushnumber(m_pState, va_arg( arg_ptr, double));
                break;
            case 'd':
            case 'i':
                lua_pushnumber(m_pState, va_arg( arg_ptr, int));
                break;
            case 's':
                lua_pushstring(m_pState, va_arg( arg_ptr, char*));
                break;
            case 'z':
                lua_pushlightuserdata(m_pState, va_arg( arg_ptr, void*));
                break;
            default:
                break;
            }
            ++iRet;
```

```
        }
        ++pFormat;
    }
    return iRet;
}

lua_State* CToLua::getState()
{
    return m_pState;
}
```

main.cpp

```
#include "CToLua.h"

int main()
{
    CToLua tolua;
    tolua.loadLuaFile("C:\\c++\\lua\\test.lua");
    double iValue = tolua.callFileFn("add", "%d%f", 20, 3.69); // 23.69
    cout << iValue << endl;
    iValue = tolua.callFileFn("sub", "%f%i", 23.69,20); // 3.69
    cout << iValue << endl;
    return 0;
}
```

test.lua

```
function add (x,y)
        return x+y
end

function sub (x,y)
        return x-y
end
```

callFileFn 方法使用类似 printf，%d 或%i 代表整数，%f 代表浮点数，%s 代表字符串，%z 代表自定义类型。

由于 lua 可以支持多返回值，所以原先的设计思想是返回一个 vector，里边包含一个结构体，有两个成员，一个是 string type，一个是 void* pValue。但当我实现了后，发现虽然支持了多个返回值了，但是使用

起来却不是那么方便。所以索性就不实现这种返回多个返回值的接口了。函数的返回值固定，只返回 double 类型的值，如果想要多个返回值，可以传自定义参数来实现。

上一节我们实现了一个 c++的封装类，通过该类我们就可以调用 lua 中的函数。可是这还满足不了我们的需求，我们还想通过 lua 来调用我们 c++的方法。通过研究/tolua++-1.0.93/src/tests 下的例子，结合 c++的特性，我总结了一个 tolua 的例子。不能说相当完美，但是基本的功能已经能够满足项目的需求了，而且通过这个例子，也可以使各位对 tolua 的语法以及用法有一个初步的了解。

本例只是一个简单的 lua 与 c++互调的示例，如果想要更进一步的学习 tolua，可以参考 /tolua++-1.0.93/src/tests 下的例子，那些例子都是相当的经典。

另外鉴于 tolua 的强大，文章中可能有一些描述不清楚的地方，望大家能够给予指出，我再给予完善。如有不足之处还望大家给予指正，如有疑问可在评论中指出，我会尽快给予解决。

1 代码

CToLua.h(参照上一节)

CToLua.cpp(参照上一节)

CArray.h

```
#ifndef CARRAY_H_
#define CARRAY_H_

struct Point
{
    float x;
    float y;
};

extern int a[10];
extern Point p[10];
#endif
```

CArray.cpp

```
#include "CArray.h"

int a[10] = {1,2,3,4,5,6,7,8,9,10};
Point p[10] = {{0,1},{1,2},{2,3},{3,4},{4,5},{5,6},{6,7},{7,8},{8,9},{9,1
0}};
```

CBase.h

```cpp
#ifndef CBASE_H_
#define CBASE_H_

#include <iostream>
#include <string>
using namespace std;

class CBase
{
public:
    CBase();
    virtual ~CBase();
    void dispalyName();
    virtual void Print();
protected:
    string m_sName;
};

extern CBase* toBase(void* p);

#endif
```

CBase.cpp

```cpp
#include "CBase.h"

CBase::CBase()
{
    m_sName = "CBase";
}

CBase::~CBase()
{
}

void CBase::dispalyName()
{
    cout << m_sName <<endl;
}

void CBase::Print()
{
    cout << "I'm CBase" << endl;
```

```
}

CBase* toBase(void* p)
{
    return (CBase*)p;
}
```

CChildA.h

```
#ifndef CCHILDA_H_
#define CCHILDA_H_

#include "CBase.h"

class CChildA : public CBase
{
public:
    CChildA();
    ~CChildA();
    void Print();
};
#endif
```

CChildA.cpp

```
#include "CChildA.h"

CChildA::CChildA()
{
    m_sName = "CChildA";
}

CChildA::~CChildA()
{
}

void CChildA::Print()
{
    cout << "I'm CChildA" << endl;
}
```

CChildB.h

```
#ifndef CCHILDB_H_
#define CCHILDB_H_

#include "CBase.h"

class CChildB : public CBase
{
public:
    CChildB();
    ~CChildB();
    void Print();
};
#endif
```

CChildB.cpp

```
#include "CChildB.h"

CChildB::CChildB()
{
    m_sName = "CChildB";
}

CChildB::~CChildB()
{
}

void CChildB::Print()
{
    cout << "I'm CChildB" << endl;
}
```

CLuaComm.h

```
#ifndef CLUACOMM_H_
#define CLUACOMM_H_
```

```
int  tolua_array_open (lua_State*);
int  tolua_base_open (lua_State*);
int  tolua_childA_open (lua_State*);
int  tolua_childB_open (lua_State*);

#endif
```

mian.cpp

```
#include "CToLua.h"
#include "CLuaComm.h"
#include "CBase.h"
#include "CChildA.h"
#include "CChildB.h"

int main()
{
    CToLua tolua;
    tolua_array_open(tolua.getState());
    tolua.loadLuaFile("/home/tolua/test/array.lua");

    tolua_base_open(tolua.getState());
    tolua_childA_open(tolua.getState());
    tolua_childB_open(tolua.getState());
    tolua.loadLuaFile("/home/tolua/test/test.lua");
    double iValue = tolua.callFileFn("add", "%d%f", 20, 3.69); // 23.69
    cout << iValue << endl;
    iValue = tolua.callFileFn("sub", "%f%i", 23.69,20); // 3.69
    cout << iValue << endl;

    CBase base;
    CChildA childA;
    CChildB childB;
    tolua.callFileFn("test1", "%z%z%z", &base, &childA, &childB);
    tolua.callFileFn("test2","");
    return 0;
}
```

CArray.pkg

```
$#include "CArray.h"

struct Point
{
    float x;
    float y;
};

extern int a[10];
extern const Point p[10];
```

CBase.pkg

```
$#include "CBase.h"

class CBase
{
public:
    CBase();
    virtual ~CBase();
    void dispalyName();
    virtual void Print();
};

extern CBase* toBase(void* p);
```

CChildA.pkg

```
$#include "CChildA.h"

class CChildA : public CBase
{
public:
    CChildA();
    ~CChildA();
    void Print();
};
```

CChildB.pkg

```
$#include "CChildB.h"

class CChildB : public CBase
{
public:
    CChildB();
    ~CChildB();
    void Print();
};
```

array.lua

```
for i=0,9 do
       print (a[i])
end

for i=0,9 do
       print (p[i].x .. p[i].y)
end
```

test.lua

```
function add (x,y)
       return x+y
end

function sub (x,y)
       return x-y
end

function test1 (base,childA,childB)
       b = toBase(base)
       cA = toBase(childA)
       cB = toBase(childB)
       b:dispalyName()
       cA:dispalyName()
       cB:dispalyName()
       b:Print()
       cA:Print()
       cB:Print()
       return 1
end
```

```
function test2 ()
	b = CBase:new()
	cA = CChildA:new()
	cB = CChildB:new()
	b:dispalyName()
	cA:dispalyName()
	cB:dispalyName()
	b:Print()
	cA:Print()
	cB:Print()
	b:delete()
	cA:delete()
	cB:delete()
end
```

2 命令

tolua++ -n array -o LArray.cpp CArray.pkg

tolua++ -n base -o LBase.cpp CBase.pkg

tolua++ -n childA -o LChildA.cpp CChildA.pkg

tolua++ -n childB -o LChildB.cpp CChildB.pkg

这些命令的作用在《tolua++安装》中已经做了解释，不明白的可以去那里先学习一下。

3 编译&运行

因为我用的 eclipse 进行的开发，eclipse 自动为我生成了 makefile，所以此处就不费事去写 makefile 了，如果不喜欢使用 eclipse 的，可以使用如下这个偷懒的编译方式：

g++ -o test *.cpp -ltolua++

运行结果：

1

2

3

4

5

6

7

8

9

10

01

12

23

34

45

56

67

78

89

910

23.69

3.69

// 以下为 test1 的结果

CBase

CChildA

CChildB

I'm CBase

I'm CChildA

I'm CChildB

// 以下为 test2 的结果

CBase

CChildA

CChildB

I'm CBase

I'm CChildA

I'm CChildB

4 解说

这个例子包含了全局函数、数组、继承、多态。不仅支持 c++传自定义的对象到 lua 中，也支持 lua 调用 c++对象的方法。已经基本包含了我们所想要实现的功能。另外 tolua 还支持枚举、命名空间、变量等，这些可以参考/tolua++-1.0.93/src/tests 下的例子，都很易理解。

CArray.pkg 文件。这里边的$#include "CArray.h"中的$代表其后的内容将原方不动的插入到 tolua 生成的 cpp 文件中。

CBase.pkg 文件。因为我们通过传参把 CBase*传入到 parseParameter 方法，又利用 va_arg 将指针转化为 void*，再通过 lua_pushlightuserdata 将指针传入到 lua 脚本中。在 lua 脚本中，我们并不能直接使用这个 void*的指针去操作，而需要将该指针再转化为 CBase*，所以我们添加了 extern CBase* toBase(void* p);方法，用来将 void*转化为 CBase*。这样我们在 lua 脚本中只需要先调用 toBase 方法将 void*转化为 CBase*然后赋值给一个变量，就可以通过这个变量操作该 CBase*实际指向的对象的方法了，这里就体现出多态的特性啦。

如果我们在 pkg 文件中，也对类得构造和析构做了定义，那么在 lua 文件中，构造函数将映射为 new()，而析构函数将映射为 delete()。C++中 new 的对象将由 c++去释放，而在 lua 中生成的对象将由 lua 释放。

在 main 函数中，我们在调用 loadLuaFile 方法前，需要先调用 tolua 生成的 cpp 中的 int tolua_*_open(lua_State*)方法。我们将所有 open 方法的声明都添加到 CLuaComm.h 中。

tolua.callFileFn("test1", "%z%z%z", &base, &childA, &childB);方法是将我们生成的类传入到 lua 脚本中执行。

tolua.callFileFn("test2","");是直接调用 lua 脚本中的 test2 方法，该方法没有参数。