# LUA封装注册类成员函数

这其实是一个伪命题，用过lua的同学都知道，在c++中注册lua回调函数是有固定格式的，就像这样

int luacallback(lua\_state\* L);

如何把这一系列函数注册到lua中呢，这类函数具有的特点

1. 只有部分固定格式 例如 int func1(const funcContext& a,…);
2. 是类的成员函数
3. 附加参数各种和类型不固定，例如下边的函数

class A

{

Public:

Int func(const funcContext& context);

Int func2(const funcContext& context,int a,float b);

};

## 注册成员函数的意义

非要提意义，个人认为，这样可以更好的实现系统的封装，一般lua的封装库都会把lua回调函数放倒一个统一模块内，可以共享数据，比如物理回调模块，渲染回调模块，游戏逻辑等，成员函数在一个整个的类内可以共享数据，更好的封装和访问，一般跟lua的面向对象用法结合紧密，这个下一篇再扯了

最好的意义是，当一个开发人员写回调写成这样

Int func2(const funcContext& context,int a,float b)

{

Float x = (float)a + b;

// funcContext数据结构封装了lua环境，所以做参数获取和返回值压栈very easy

context.pushReturn(x);

}

不必关系从lua环境里取参数和获得参数，像写一个c++普通函数一样，只是多了一点点格式限制，是不是很幸福，肯定比这样写幸福

Int func2(lua\_state\* L)

{

float a = lua\_tonumber(L,1);

float b= lua\_tonumner(L,2);

Float x = a + b;

Lua\_pushnumber(x);

}

各种lua的CAPI，这还是简化版本，为了程序的健壮性，还必须考虑参数类型的校验，错误的处理，栈的清除等一系列操作，而且呢，每个函数都这么写，大量重复代码会不会让处女座的同事们很纠结？

## 实施方案

说了那么多，还是说说该怎么做吧，这里要用到一个lua的特性，叫closure，中文叫闭包，为了防止越说越乱，简单说，就是函数+数据=闭包，lua中注册一个函数，其实就是注册一个闭包的特例，那就是附加的userdata为空，当把一个字符串跟一个闭包对应的时候，当注册的函数触发，不光有该函数触发，还可以在该函数内获得一起注册的数据，也就是可以用一个标准的lua回调做为外壳，真正的回调函数在userdata内，如果这个数据里包含this指针，函数指针，就可以这样调用了

This->\*func();

貌似closure只是解决了注册成员函数的问题，那参数个数和类型如何解决，这就需要借助强大的模板的力量了，这块逻辑有点小复杂，总结下来分三步

1. 回调函数的抽象，也就是参数个数和类型作为模版参数
2. 参数个数和类型的展开，这个通过模版重载实现
3. 由于要注册到lua中，回调回来，模版属性已经没有了，所以需要记忆下，需要用某种办法记录模版信息，这里用的是静态模版函数实例化，通过调用静态函数来把参数展开，展开就是说，我们把一个函数指针当作一个类型抽象，怎么特化成一个具有特定参数个数和类型的具体函数指针
4. 很遗憾的说，这块代码菜鸟作者只能写出一层模版的，也就是只对参数个数和类型进行抽象，利用模版重载实现函数参数展开，像这样对模版进行二次抽象的神作，只能膜拜了，这种写法免去了n-1个注册函数和n-1个dispatcher类的定义，只能说，真牛，不多说，代码奉上，这是本人亲自敲出来加上注释的版本

主要的逻辑代码

Templatefunc.h

#ifndef \_TEMPLATE\_FUNCTION\_H\_

#define \_TEMPLATE\_FUNCTION\_H\_

extern "C"

{

#include "../lua/Lua/lua.h"

};

class FunctionContext

{

public:

FunctionContext(lua\_State\* pState)

{

m\_pState = pState;

}

~FunctionContext(){}

//这里需要特化多种模版，做测试就写一种就ok了

template<class T>

bool GetParam(int index,T& value){ return true;}

template<>

bool GetParam(int index,double& value)

{

//这里可以加入一些校验逻辑，防止程序crash

value = lua\_tonumber(m\_pState,index);

return true;

}

private:

lua\_State\* m\_pState;

};

//重载模版函数，利用参数类型匹配,只做了2个参数的，要有使用价值，怎么也得做到3个…

template<typename caller>

static int CallFunction(caller\* p,int (caller::\*pFunc)(FunctionContext\*),FunctionContext\* pContext)

{

return (p->\*pFunc)(pContext);

}

template<typename caller,typename Param1>

static int CallFunction(caller\* p,int (caller::\*pFunc)(FunctionContext\*,Param1),FunctionContext\* pContext)

{

Param1 p1;

pContext->GetParam(1,p1);

return (p->\*pFunc)(pContext,p1);

}

template<typename caller,typename Param1,typename Param2>

static int CallFunction(caller\* p,int (caller::\*pFunc)(FunctionContext\*,Param1,Param2),FunctionContext\* pContext)

{

Param1 p1;

Param2 p2;

pContext->GetParam(1,p1);

pContext->GetParam(2,p2);

return (p->\*pFunc)(pContext,p1,p2);

}

//这个类的作用就是，当把this指针和function指针放入lua的userdata后，必须保存function的模版类型信息，比如，有几个参数，参数类型

//那么就可以利用这个dispatcher通过他的实例化得到一个静态函数，这个静态函数也放入到userdata中

//简化版本，userdata中有dispatcher的静态函数，真正c++回调函数的指针，对象的this指针，当然也可以加入其他一些内容，比如函数签名，跟类型相关的逻辑，防止重名等

typedef int (\*dispatcherFunc)(FunctionContext\* pContext,const char\* pUserData);

template<typename caller,typename Func>

class dispatcher

{

public:

static int CallFunc(FunctionContext\* pContext,const char\* pUserData)

{

//这个地方实现了参数展开的功能，通过注册的时候的模版参数实例化，创建了一个针对该参数个数和类型的回调函数的调用函数，在利用模版重载，调用正确的call function

return CallFunction((caller\*)pUserData,\*(Func\*)(pUserData+sizeof(caller\*)),pContext);

}

};

int luaShellFuc(lua\_State\* L)

{

FunctionContext context(L);

//从栈中取出userdata

unsigned char \*pBuffer = (unsigned char\*)lua\_touserdata(L,lua\_upvalueindex(1));

//从userdata中取出dispatcher函数

dispatcherFunc\* pfunc = (dispatcherFunc\*)pBuffer;

(\*pfunc)(&context,(const char\*)pBuffer+ sizeof(dispatcherFunc\*));

return 1;

}

template <typename caller,typename func>

void RegisterFunc(lua\_State\* pState,const char\* funcName,caller\*p,const func& function)

{

unsigned nTotalSize = sizeof(dispatcherFunc\*) + sizeof(caller\*) + sizeof(func);

unsigned char \*pBuffer = (unsigned char\*)lua\_newuserdata(pState, nTotalSize );

int offset = 0;

//实例化一个针对特定参数个数和类型的静态函数，实现了模版参数的保存

dispatcherFunc disfunc = dispatcher<caller,func>::CallFunc;

//这里一定要取地址，否则会把函数的一部分代码copy过去，而不是入口地址

/\*参考这个例子好理解，int\* p = new int;

memcpy(a,p,sizeof(int));

memcpy(a,&p,sizeof(int\*));

一个是把int的值copy进去，一个是把指向这个int的地址copy过去

拷贝p的值，就要对p取地址，拷贝p指向的值，就用p当地址，我们这边要得到函数地址，显然是拷贝disfunc的值，而不是他指向的函数

\*/

memcpy(pBuffer,&disfunc,sizeof(dispatcherFunc));

offset += sizeof(dispatcherFunc);

//此处同理，拷贝指针的值，不是对象，奇怪的是，把p前边的&去掉依然是正确的，亲测。。。求达人解答

memcpy(pBuffer+offset,&p,sizeof(caller\*));

offset += sizeof(caller\*);

memcpy(pBuffer+offset,&function,sizeof(function));

lua\_pushcclosure( pState,luaShellFuc,1 );

//这里可以根据自己需要放到特定表里，做简化

lua\_setglobal(pState,funcName);

}

#endif

测试代码

Luacallbacktest.cpp

#include "stdafx.h"

#include <iostream>

#include "templateFunc.h"

#include "string.h"

extern "C"

{

#include "../Lua/Lua/lauxlib.h"

};

class TestClass

{

public:

int func0(FunctionContext\* pContext)

{

std::cout<<"0";

return 1;

}

int func1(FunctionContext\* pContext,double testA)

{

std::cout<<"1";

return 1;

}

int func2(FunctionContext\* pContext,double param1,double param2)

{

std::cout<<"2";

return 2;

}

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

lua\_State\* pState = lua\_open();

// luaL\_openlibs(pState);

TestClass test;

RegisterFunc(pState,"func0",&test,&TestClass::func0);

RegisterFunc(pState,"func1",&test,&TestClass::func1);

RegisterFunc(pState,"func2",&test,&TestClass::func2);

luaL\_dostring(pState, "func1(2.0) func2(1.0,2.0)");

return 0;

}