

如果我们要在lua环境中使用上面3个带有继承关系的c++类, 要先注册这三个类到lua环境中, 然后在lua环境需要的时候, 推送对象到lua环境中, 在lua环境调用对象方法时, 会回调我们注册时指定的c++静态方法.

注册的时候分别把三个类叫做LUA\_BASE, LUA\_DERIVED1, LUA\_DERIVED2.

一个基本原则是, 不管我们实际上push到lua环境中的对象指向什么, push的时候声称它是什么, lua就会把它当成什么来对待, 包括它可以调用的方法和回调方法中栈底对象的类名都是当初push的时候声明的那个.

# 注册

注册基类时指定父类为空

XLLRT\_RegisterClass(hEnv, LUA\_BASE, s\_LuaBaseMemberFunctions,NULL,0)  
注册子类时指定父类为LUA\_BASE

XLLRT\_RegisterClass(hEnv, LUA\_DERIVED1, s\_LuaDerived1MemberFunctions,LUA\_BASE,0)

XLLRT\_RegisterClass(hEnv, LUA\_DERIVED2, s\_LuaDerived2MemberFunctions,LUA\_BASE,0)

注册子类时的方法列表中不包含基类已经注册过的方法, (不知道是Lua干的还是迅雷干的), 它们的包含方法的metatable会被自动合并的.

lua环境中的”继承”没有什么多态的概念, 我们注册时候指定的方法就是lua会调用的方法.

如果注册LUA\_BASE的方法列表是这样:

XLLRTGlobalAPI LuaBase::s\_LuaBaseMemberFunctions[] =

{

{"Click", LuaBase::Click},

{"\_\_gc", LuaBase::DeleteSelf},

{NULL,NULL}

};

注册LUA\_DERIVED1的方法列表是这样:

XLLRTGlobalAPI LuaDerived1::s\_LuaDerived1MemberFunctions[] =

{

{"Click1", LuaDerived1::Click1},

{NULL,NULL}

};

那么在lua环境中, LUA\_DERIVED1的方法列表就是这样:

{"Click", LuaBase::Click},

{"\_\_gc", LuaBase::DeleteSelf},

{"Click1", LuaDerived1::Click1},

{NULL,NULL}

如果一个声称是”LUA\_DERIVED1”的对象被push到lua环境中, lua就认为这个对象有上面的这些方法(被“”起来的是lua调用的时候用的方法名)可以用, 例如叫做”Click”的方法在lua中被调用, 与之对应的c++静态方法LuaBase::Click就会被调用, 即使我们push的时候传的是一个LuaDerived1的指针.

所以多态是在LuaBase::Click的时候通过Base->Click()来实现的, 和Lua没有关系.

# Lua向c++求对象

Lua环境向C++要求一个对象时, 对返回的对象类型不做检查, c++把对象push到lua环境中的时候声称是什么, lua环境就认为它是什么

Base \*obj = new Base()

Base \*obj1 = new Derived1()

Base \*obj2 = new Derived2()

XLLRT\_PushXLObject(luaState, LUA\_BASE, obj);

XLLRT\_PushXLObject(luaState, LUA\_DERIVED1, obj1); // 需要obj是子类指针还是父类指针?

XLLRT\_PushXLObject(luaState, LUA\_DERIVED2, obj2);

这里不管obj指向什么, lua环境都会按照pushXLObject的类名对应的方法列表来解释它. 即push时声称obj是个LUA\_BASE, 则lua环境认为这个对象有LUA\_BASE被注册时带有的方法列表这些方法可以使用.

\*如果我们把Derived1 \*obj1声称是LUA\_DERIVED2 push到lua环境中去, lua也会接受, 并且认为obj1有LUA\_DERIVED2注册的那些方法可以调用.

这样三个对象被push到lua环境中. Lua可以对obj调用LUA\_BASE注册时带有的方法”Click”, 对obj1调用LUA\_BASE+LUA\_DERIVED1的方法, 对obj2调用LUA\_BASE+LUA\_DERIVED2的方法. 在Lua方法对应的c++静态方法被lua回调时, 当前的LuaState栈底是之前push到lua环境中的对象(obj, obj1, obj2), 往上依次是参数1, 参数2...

# 使用对象

Lua环境得到了c++返回的对象LUA\_BASE obj, LUA\_DERIVED1 obj1, LUA\_DERIVED2 obj2, 可以调用它们的被注册的方法.

LuaBase::Click(lua\_State \*L)被调用时, 我们可以得到当前被调用对象的注册类名:

const char \*className = XLLRT\_GetXLObjectClass(L, 1);

虽然LuaBase::Click()是通过LUA\_BASE注册到lua环境, 但是此处的className返回的是push的时候具体的类名, 有可能是LUA\_DERIVED1, LUA\_DERIVED2

我们还可以得假设当前栈底对象的类名

long result = XLLRT\_CheckXLObject(luaState, 1, LUA\_BASE, &lpObjBase);

对于obj, obj1, obj2上述调用都是成功的, 因为lua看来这三个对象都满足LUA\_BASE所注册的方法, lpObjBase指向栈底对象.

long result = XLLRT\_CheckXLObject(luaState, 1, LUA\_DERIVED1, &lpObjDerived1);

如果当前被调用的是obj1, 上述调用成功; 如果当前被调用的是obj2, 上述调用崩坏, 因为obj2是作为”LUA\_DERIVED2”被push到lua环境中的, 没有LUA\_DERIVED1注册的方法

为了安全的类型检查, 我们可以调用

long result1 = XLLRT\_GetXLObject(luaState, 1, LUA\_DERIVED1, &lpObjDerived1);

如果当前被调用的是obj1, 同样调用成功; 如果是obj2, 则result返回非0, 且lpObjDerived1被赋值null. 这样保证了我们不会在一个作为LUA\_DERIVED2 push到lua环境中的对象上调用LUA\_DERIVED1注册的方法.

在上面两个新接口出现之前, 我们用的是:

Derived1 \*\*ppDataObject = reinterpret\_cast<Derived1\*\*>(lua\_touserdata(luaState, 1));

这个调用不会检查栈底对象的类型

LuaDerived1::Click1()和LuaDerived2::Click2()只有可能被obj1和obj2分别调用. 因为它们只被一个类名注册过.

\*\*回到刚才push的时候, 如果我们在push的时候故意声称一个不符合的类名, 即

Base \*obj2 = new Derived1();

XLLRT\_PushXLObject(luaState, LUA\_DERIVED2, obj2);

在push的时候是没有任何问题的, lua不管c++开辟内存的时候做了什么, 它就认为这个对象有LUA\_DERIVED2的方法可以用.

但是在调用方法的时候, 如果我们在lua中调用:

obj2:Click2()

对应的c++方法被调用时我们检查栈底元素不会发现任何异常:

LuaDerived2::Click2(lua\_State\* L)

{

// 该调用是成功的

long result2 = XLLRT\_CheckXLObject(luaState, 1, LUA\_DERIVED2, &lpObjDerived2);

Derived2 \*pDataObject = (Derived2\*)lpObjDerived2;

//这个调用是成功还是失败, 取决于Derived2::Click2()的实现有没有访问类的非静态成员数据, 如果有的话, 会崩溃

pDataObject->Click2();

}