C++反射：全方位解读Lura库的前世今生！

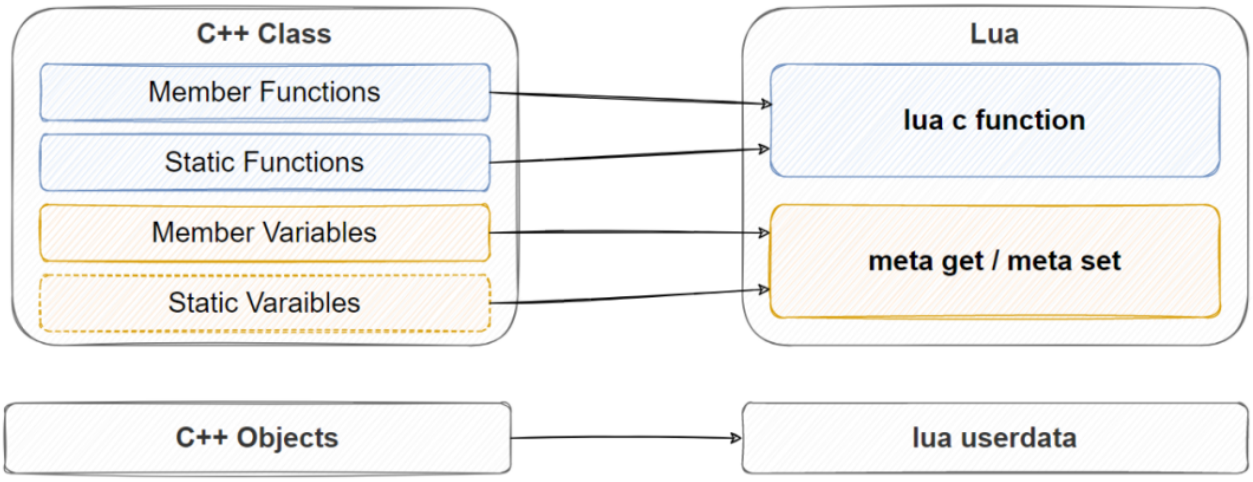
原创 沈芳 云加社区 2022-03-21 18:07

导语 | 本文我们将以lura库为例，介绍如何以C++反射作为基础设施，以更简洁的方式来实现一版lua的bridge，主要围绕lura库的前世今生来逐步展开。(本文一些知识需要适当了解lua c api和lua的meta table相关知识，了解相关知识阅读效果更佳。)

在上篇《***C++反射：深入探究function实现机制！***》中我们对反射中的Function实现做了相关的介绍，本篇将深入lura这部分进行阐述。

一、lua bridge核心功能概述

Lua的bridge层实现比较核心的功能是导出C++类到Lua中使用，基本都要完成如下图所示的几项功能:



（一）函数的处理

不管是“member functions”或者是“static functions”的处理，在bridge库上的实现都比较统一，整个过程与我们前面提到的函数的类型擦除基本是一致的。c++函数向lua注册的核心目的只有一个：将需要在lua中调用的c++函数，转换为统一类型的lua c function。剩下的调用过程就比较简单了，正确填入参数，我们即可以像一

个标准的lua函数那样使用这些c++函数了。从上面提到的4个库的实现方式上来看，除tolua++外的luabind，luatinker，luabridge，它们的实现都会依赖C++模板来完成函数向lua c function的转换，细节可参考第三篇《***C++反射：深入探究function实现机制！***》中关于lua c function注册的部分，区别在于这几者都向下兼容了c++98，相关的模板使用部分看起来会晦涩很多，因为varadic template在c++98尚未支持，我们会发现大量的从0个参数到N个参数展开的模板代码，导致他们函数类型统一部分的实现非常复杂，但实际上只是做了函数类型统一这一件事件。tolua++的方式相对简单直接，通过自动生成大量的中间代码来将c++函数转换为lua c function，生成的代码量虽然比较多，但对比用模板的方式，因为不需要考虑模板的包装，小的功能扩展会简单直白不少。

（二）属性的处理

这部分的实现基本都严重依赖下一节中c++对象到userdata的包装，核心功能是依托于自定义的meta get/set方法，完成对userdata中对应C++对象某成员变量的获取。

（三）c++对象->userdata

这部分更多的是作为一个对象容器载体，然后通过meta table来方便lua访问这个载体，主要提供函数获取，属性获取等功能，基本上每个bridge库都会有自己的实现，以及在C++中实现特定的meta get，meta set函数，细节比较多，lura相关的功能实现基本都被UserObject代替，userdata再对UserObject进行包装即可，很多复杂度都转移到c++反射实现本身了，所以这里不详细展开相关的细节了。

二、Lura的前世-从过往说起

（一）过往使用的lua bridge实现

lua的bridge层实现特别多，就我自己用过的也不少，下面仅列出个人有在实际项目中使用过的:

- luabind: 依赖boost的一个lua实现，当时应该是各种特性提供的最多的lua bridge。

- luatinker: 对比luabind简化了支持的特性，同时也完全不依赖boost库。
- tolua++: Cocos2dx使用的lua bridge，不依赖模板等特性，利用生成器生成bridge代码。
- luabridge: 某项目框架之前用到的一个lua bridge实现，有挺多优点，配合基于libclang实现的导出器，能够很好的完成bridge的工作。

注：lua的bridge实现还有不少，像以zero overhead abstraction为卖点的sol2这些我们就不展开了，感觉参考意义大于实际项目使用的意义。另外还有一些c的FFI实现，比如luajit自带的FFI，不是本文关注的重点，这里也不详细赘述了。

（二）实践总结

上面列举的这些库，**优点**还是挺明显的:

- 功能特性齐全，基本都覆盖了上面说到的bridge核心功能。
- 配合自带或者项目自己维护的导出器，日常维护使用便利。
- 除luabind外核心代码都比较简洁，调整难度不高。
- C++与Lua的边界明确，便于添加Debug和Profiler等功能。
- 基于这些库实现一些复杂特性成本可空，可以比较好的适配特定需求。

如在Lua中override c++ class的virtual function等功能

上面介绍了Luabridge在使用过程中我们体验比较好的那部分，但它其实依然有很多跨语言库的通病:

- 对像唯一性
- 类型丢失
- 对象的生命周期管理

这些都会存在一些坑点，外部使用者比较容易出现一些特定情况下出现诡异Bug没法排查的问题。它只是一个Bridge，能够很好的帮你完成Bridge相关的功能，但并不能帮我们梳理出清晰的跨语言机制，肯定没法依托机制本身来提供足够多的基础特性来保证跨语言的行为安全有效，在我们有了基本的c++反射机制后，可以尝试结合反射来组织更好的跨语言实现。

（三）加入c++反射后的lua bridge实现

当有了c++反射机制，尝试整合原有的lua bridge实现，我们会发现，缺乏基础机制支持的bridge实现相关的问题会暴露的特别的明显，主要是以下几点:

- **类同功能的实现因为语言特性的依赖导致的代差问题**

像函数类型擦除相关的代码实现，每个库都是自己独立实现的，依赖的语言特性越少，复杂度就越高，这部分也是比较难迭代维护的部分。除了上面说到的，一些函数类型擦除的工作，随着语言特性的不断更新升级(如noexcept关键字的引入)，使用新的特性的函数可能不兼容原有的类型擦除实现，这种调整兼容相对麻烦，也容易出错，随着业务侧使用复杂度的提高，可能还会引入一些新的问题。

- **C++反射与LuaBridge的冲突**

如UserObject与Value，对比上面介绍的bridge中用userdata包装c++对象来说，对象类型无法统一，导致Meta相关的接口设计非常受限，或者需要在边界处不停处理数据类型的转换，带来不必要的性能损耗。相关的函数调用方式差异非常大，无法有效的无性能损耗的情况下进行统一。对象的生命周期问题也从原来Luabridge没有很好解决的状况下拖到了更麻烦的情况。

- **导出工具的分裂**

类似的事情，重复进行，而且特性之间无法共享，需要各个地方单独实现。

（四）从Ponder重新思考

ponder应该是出于演示自己爆表特性的目的，撸了个比较简洁的expose反射信息到lua的实现，以及一个QT的实现，就特性完整度和实际性能来说，个人感觉炫技的性

质重过实用，比如像Lua的Enum导出，你还能在Lua侧修改导出的Enum的值。&emsp不过它其实很好的探索了c++的反射，怎么作为底层来支撑跨语言bridge的问题，至少基于它现有的实现，我们通过阅读理解相关代码，可以有效的评估如果结合c++反射，我们的lua bridge层应该如何来设计才是比较符合项目需要的。

（五）重新梳理的实现思路-lura

Ponder本身的Lua导出并不完备，使用的便利性和周边设施，相比成熟的Lua中间层有差距。所以我们最后的实现思路就变成了：

- 更多的吸收成熟Lua中间层的外围设施和封装思路。
- 利用反射库的基础设施实现函数类型擦除。
- 利用反射库的UserObject去封装userdata。

这样对于lua bridge的实现来说，一方面内核已经切换到c++反射，另外业务侧保留了常规bridge实现接口和相关特性，一开始也有了一个比较高的业务使用成熟度。下文中我们直接以实现已经比较完整的lura来展开相关的内容。

三、lura的今生

我们先以Vector3为例，从业务侧逐渐深入底层来看一下当前lura的整个设计。

反射信息注册：

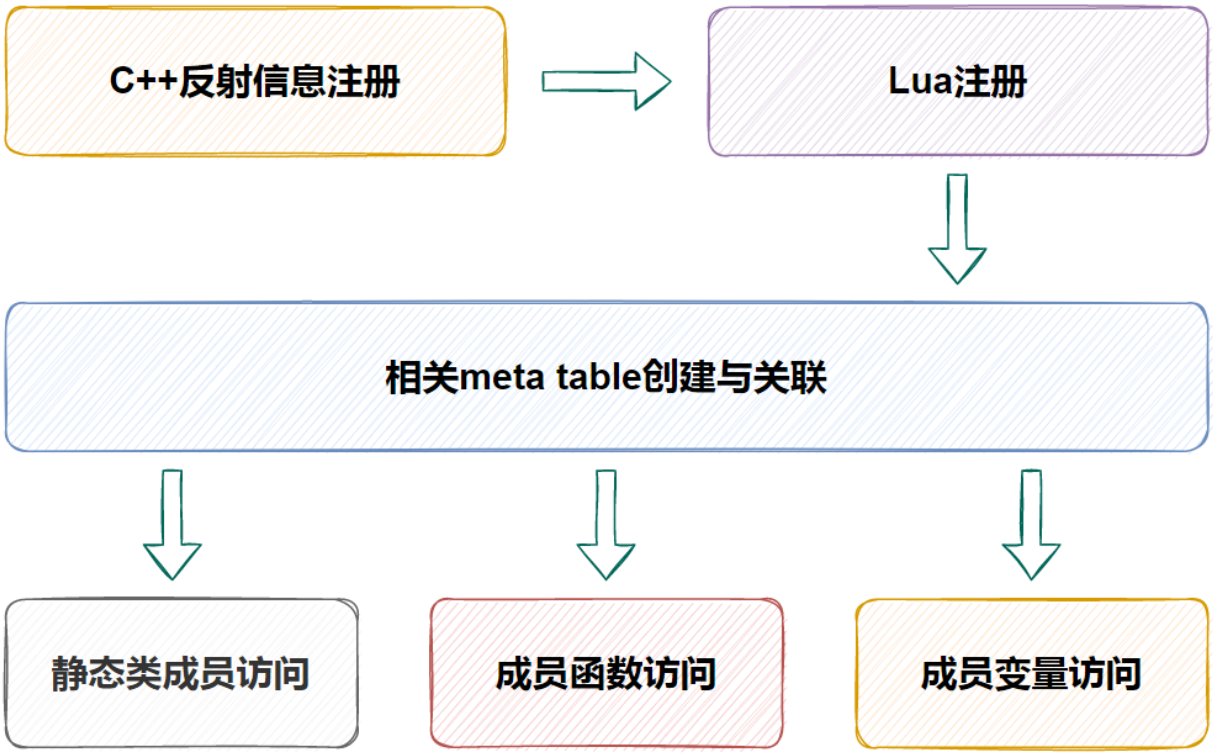
```
__register_type<rstudio::math::Vector3>("rstudio::math::Vector3")

        //member fields export here
        .property("x", &rstudio::math::Vector3::x)
        .property("y", &rstudio::math::Vector3::y)
        .property("z", &rstudio::math::Vector3::z)
        .static_property("ZERO", [](){ return rstudio::math::Vector3::ZE
        //member properties export here
```

```
        //constructor export here
        .constructor<double, double, double>()
        .constructor<const rstudio::math::Vector3&>()
        .constructor<double>()
        .constructor<>()
        //methods export here
        .overload("__assign"
            ,[](rstudio::math::Vector3* self, double fScalar){return sel
            ,[](rstudio::math::Vector3* self, const rstudio::math::Vecto
        )
        .function("Length", &rstudio::math::Vector3::Length)
    ;
```

Lua注册:

```
lura::get_global_namespace(L).begin_namespace("math3d")
    .begin_class<rstudio::math::Vector3>("Vector3")
    .end_class().end_namespace();
```

对比luabridge的实现，Lua注册部分namespace和class部分保留了，概念也基本对齐，但我们的具体property和function注册已经是由反射部分负责了，Lua部分不再重复相关的工作。



下文我们会结合部分代码对这部分进行讲述。

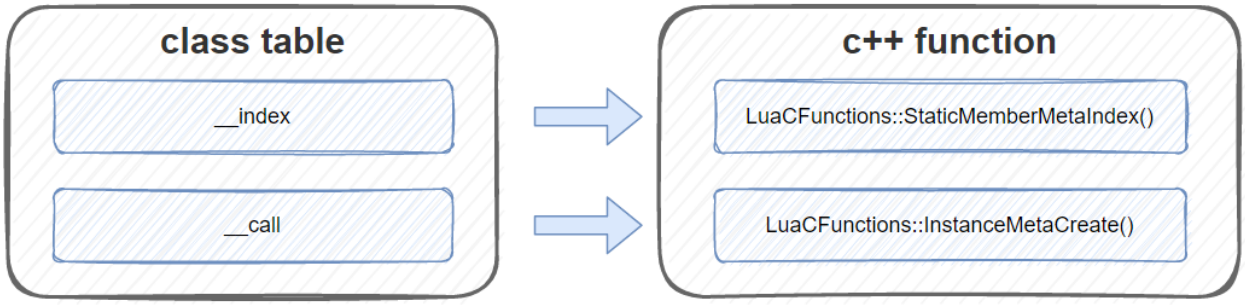
（一）lura核心机制简述

lura整体的代码因为各种实际项目需求，还是比较多的，不过核心机制相关的代码比较简单，上图中像:

- 静态类成员访问
- 成员函数访问
- 成员变量访问

这几个功能都是由两个特定构造的meta table来完成的，所以我们了解了这两个meta table的创建，以及相关meta method对应的c++实现，基本就掌握了lura的bridge实现机制。

- **提供静态类成员访问功能的meta table**



class table提供了两个元方法:

- __index
- __call

__index用于完成对类的静态函数和静态变量进行访问，我们可以从上图中看到该功能实现被关联到了c++函数StaticMemberMetaIndex()函数上

__call用于直接在lua中构造一个对应的c++对象，我们将class table作为自己的meta table设置后，也会将这个meta table暴露到_G上，如上例中的Vector3，在lua中，我们通过如下代码，即可完成对Vector3的class table的__call的访问，最后在lua中创建对应的c++对象，具体构造的细节会在下面章节中进行说明。

```
local vec =  math3d.Vecto3()
```

接下来我们看一下具体的创建代码，lua c api相关的代码，整体实现配合注释比较好理解，这里不逐行展开了。

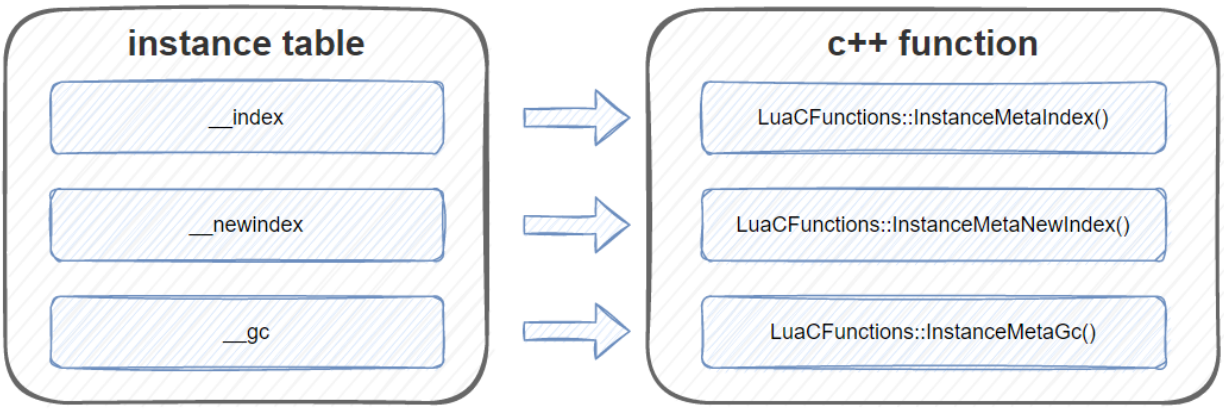
```
void LuaCFunctions::CreateClassContentTable(lua_State* L, const rstudio:
                                            const std::string_view
                                            const void* classMetaTa
                                            const void* instanceMet
  lua_createtable(L, 0, 20);  // Stack: class table (cl)
  lua_pushvalue(L, -1);       // Stack: cl, cl
  lua_setmetatable(L, -2);    // Stack: cl        ->cl.__mt = cl

  lua_pushlstring(L, name.data(), name.length());
  lua_rawsetp(L, -2, GetTypeKey());  // Stack: cl        -> cl[typeKey]
```

```
11
12    lua_pushlstring(L, cppName.data(), name.length());
13    lua_rawsetp(L, -2, GetCppTypeNameKey());  // Stack: cl        -> cl[cp
14
15    lua_pushliteral(L, "__index");                                 // Stack:
16    lua_pushlightuserdata(L, (void*)&cls);                         // Stack:
17    lua_pushvalue(L, -3);                                          // Stack:
18    lua_pushcclosure(L, LuaCFunctions::StaticMemberMetaIndex, 2);  // Stac
19    lua_rawset(L, -3);                                             // Stack:
20
21    // Help name
22    lua_pushliteral(L, "__tostring");                          // Stack: cl,
23    lua_pushlightuserdata(L, (void*)&cls);                     // Stack: cl,
24    lua_pushcclosure(L, ClassContentTableToString, 1);  // Stack: cl, "__t
25    lua_rawset(L, -3);                                         // Stack: cl
26
27    // Bind class table to registry table
28    lua_pushvalue(L, -1);  // Stack: cl, cl
29    lua_rawsetp(L, LUA_REGISTRYINDEX,
30                classMetaTableKey);  // Stack: cl        -> _R[classMetaTa
31
32    // Namespace bind static class do not need "__call" method
33    if (needMetaCallMethod) {
34      lua_pushliteral(L, "__call");                            // Stack: cl
35      lua_pushlightuserdata(L, (void*)&cls);                   // Stack: cl
36      lua_pushcclosure(L, LuaCFunctions::InstanceMetaCreate, 1);  // Stack
37      lua_rawset(L, -3);                                       // Stack: cl
38
39      int clTableIndex = lua_absindex(L, -1);
40      CreateInstanceMetaTable(L, cls, clTableIndex);  // Stack: cl, instar
41      lua_rawsetp(L, LUA_REGISTRYINDEX,
42                  instanceMetaTableKey);  // Stack: cl        -> _R[instar
43
44      // Register key to MetaClass
45      cls.SetUserdata<rstudio::reflection::ClassUserdata::eLura>(instanceM
46    }
47  }
```

小技巧：需要注意的一点是我们注册元方法的时候会利用lua的up value机制将一些额外的参数带入对应的c++函数中，这样在调用发生时，就能够很简单的通过up value取到注册时附加上去的值了，如上面代码中的MetaClass指针，class table本身，我们都通过这种方式带入了对应的c++函数调用中，这个是lua中间层比较常用的实现技巧，大家可以自行活学活用。

小建议：大家写lua/c++交互代码的时候，可以如示例中一样，在注释中给出每行api调用后stack发生的变化，这样能够清晰的知道当前栈的情况，整个交互代码的实现会变得更简单，也有利于其他小伙伴阅读理解相关的代码。

- **提供实例成员访问的meta table**



如上图所示，对于每个C++对象，挂接的meta table就包含了：

- __index
- __newindex
- __gc

等元方法，这样在我们对类对象进行函数调用或者成员变量访问的时候，都能够正确的触发相关的逻辑，再利用前面介绍到的up value，每次调用都能很好的访问到相关的对象。

```
1  void LuaCFunctions::CreateInstanceMetaTable(lua_State* L, const MetaClas
2    lua_createtable(L, 0, 3);  // Stack: mt                    -> +1 mt
3
4    lua_pushliteral(L, "__index");              // Stack: mt, "__index"
```

```
  lua_pushlightuserdata(L, (void*)&cls);     // Stack: mt, "__index", cl
  lua_pushvalue(L, clTableIndex);            // Stack: mt, "__index", cl
  lua_pushcclosure(L, InstanceMetaIndex, 2);  // Stack: mt, "__index", f
  lua_rawset(L, -3);                         // Stack: mt
                     // l_inst_index

  lua_pushliteral(L, "__newindex");             // Stack: mt, "__newinde
  lua_pushlightuserdata(L, (void*)&cls);        // Stack: mt, "__newinde
  lua_pushvalue(L, clTableIndex);               // Stack: mt, "__newinde
  lua_pushcclosure(L, InstanceMetaNewIndex, 2);  // Stack: mt, "__newinc
  lua_rawset(L, -3);                            // Stack: mt
                     // l_inst_newindex   -2

  lua_pushliteral(L, "__gc");        // Stack: mt, "__gc"              -
  lua_pushcfunction(L, InstanceMetaGc);  // Stack: mt, func[0u]
  lua_rawset(L, -3);                 // Stack: mt                   -> mt
                     // l_finalize  -2

  // Help name
  lua_pushliteral(L, "__tostring");             // Stack: mt, "__tostrir
  lua_pushlightuserdata(L, (void*)&cls);        // Stack: mt, "__tostrir
  lua_pushcclosure(L, InstanceMetaToString, 1);  // Stack: mt, "__tostri
  lua_rawset(L, -3);                            // Stack: mt

  InstanceRegisterCustomMetaMethod(L, cls, "__add");
  InstanceRegisterCustomMetaMethod(L, cls, "__sub");
  InstanceRegisterCustomMetaMethod(L, cls, "__mul");
  InstanceRegisterCustomMetaMethod(L, cls, "__div");
}
```

这里给出 __index->InstanceMetaIndex()的实现，方便大家参考:

```
// obj[key]
int LuaCFunctions::InstanceMetaIndex(lua_State* L) {
  lua_pushvalue(L, lua_upvalueindex(1));
  const auto* cls = (const ::framework::reflection::MetaClass*)lua_touse

```

```
  void* ud = lua_touserdata(L, 1);  // userobj - (obj, key) -> obj[key]
  //if obj is nothing, just return nil
  ::framework::reflection::UserObject* uobj = (::framework::reflection::
  if(uobj == nullptr || *uobj == ::framework::reflection::UserObject::nc
  {
    return 0;
  }

  if (FRAMEWORK_UNLIKELY(lua_isinteger(L, 2))) {
    // try to handle as ArrayObject first
    return InstanceMetaIndexForArrayObject(L, cls, ud);
  }

  const char* skey = lua_tostring(L, 2);
  const ::framework::reflection::IdRef key(skey ? skey : "");

  // check if getting property value
  const framework::reflection::Property* pp = nullptr;
  if (cls->TryGetProperty(key, pp)) {
    ::framework::reflection::UserObject* uobj = (::framework::reflectior
    return LuraHelper::PushValue(L, pp->Get(*uobj));
  }

  // check if calling function object
  const ::framework::reflection::Function* fp = nullptr;
  if (cls->TryGetFunction(key, fp)) {
    return PushReflectionFunction(L, fp);
  }

  // for pure lua function support here
  lua_pushvalue(L, lua_upvalueindex(2));       // cl
  lua_rawgetp(L, -1, GetClassCFunctionKey());  // cl, cft
  if (lua_istable(L, -1)) {
    // cfunction table not null
    lua_pushvalue(L, 2);  // cl, cft, key
    lua_rawget(L, -2);    // cl, cft, func
    return 1;
  }

  return 0;
```

```
46 }
```

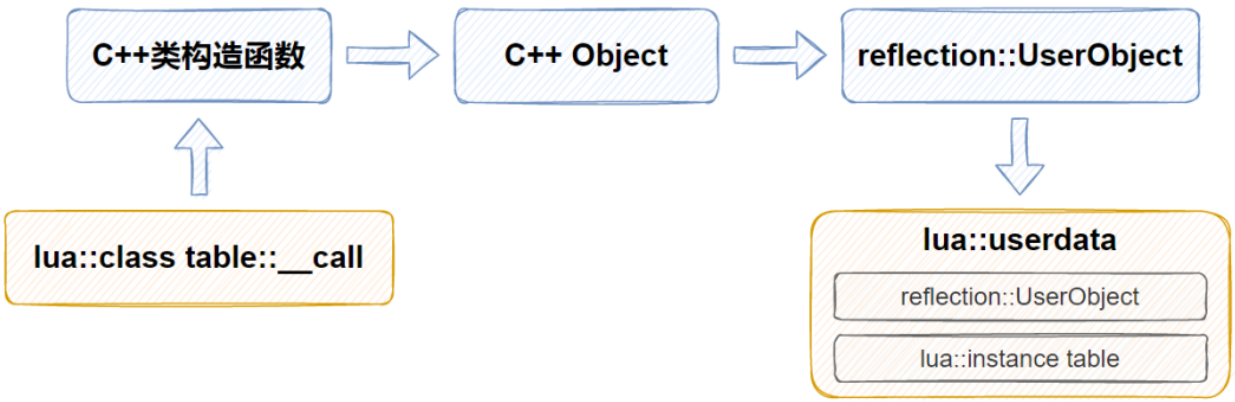
其他几个元方法的实现依托反射也比较简单，__index是其中最复杂的一个，不重复贴出了。

- **构造函数-串接两个meta table的桥梁**

两个meta table各自完成了自己的功能，但其实我们会发现，以上面出现过的示例代码为例:

```
local vec =  math3d.Vecto3()
print("vec:", vec.x, vec.y, vec.z)
print("len:", vec:Length())
```

vec是什么，怎么支撑上面的.x，.y，.z成员获取和Length()函数的调用的呢？答案就在上面提到过的LuaCFunctions::InstanceMetaCreate()函数上，我们结合相关的代码和图来了解一下实现原理:




```
int LuaCFunctions::InstanceMetaCreate(lua_State* L) {
  // get Class* from class object
  lua_pushvalue(L, lua_upvalueindex(1));
  if (!lua_isuserdata(L, -1)) {
    lua_pop(L, 1);
    luaL_error(L, "Can not find upvalue for create object!");
    return 0;
```

```
  }

  const MetaClass* cls = (const MetaClass*)lua_touserdata(L, -1);
  lua_pop(L, 1);

  framework::reflection::Args args;
  constexpr int c_argOffset = 2;  // 1st arg is userdata object
  const int nargs = lua_gettop(L) - (c_argOffset - 1);
  for (int i = c_argOffset; i < c_argOffset + nargs; ++i) {
    // there may be multiple constructors so don't check types
    args += LuraHelper::GetValue(L, i);
  }

  // Search an arguments match among the list of available constructors
  framework::reflection::UserObject obj;
  for (size_t nb = cls->GetConstructorCount(), i = 0; i < nb; ++i) {
    const auto& constructor = *(cls->GetConstructor(i));
    if (constructor.CheckLuaSignatureMatchs(L, 1, nargs)) {
      // Match found: use the constructor to create the new instance
      obj = constructor.CreateFromLua(L, nullptr, 1);
    }
  }

  // framework::reflection::runtime::ObjectFactory fact(*cls);
  // framework::reflection::UserObject obj(fact.construct(args));
  if (obj == framework::reflection::UserObject::nothing) {
    lua_pop(L, 1);  // pop new user data
    luaL_error(L, "Matching constructor not found");
    return 0;
  }

  void* ud = lua_newuserdata(L, sizeof(UserObject));  // Stack: ud
  new (ud) UserObject(obj);

  const void* insMetaKey = cls->GetUserdata<ClassUserdata::eLura>();
  lua_rawgetp(L, LUA_REGISTRYINDEX, insMetaKey);  // Stack: ud, ins_meta
  lua_setmetatable(L, -2);                        // Stack: ud

  return 1;
}
```

这段代码主要的作用就是找出正确的构造函数，然后构建对应的UserObject，再进一步的构建lua的userdata，借助反射的类型擦除，这部分已经变得很简洁了，我们构建userdata后再将前面介绍过的instance meta table跟这个userdata进行关联，整体的机制就串接起来了。相关的成员变量获取和成员函数的调用，都依托于反射本身的实现，这里不一一展开了。

- **拾遗-与luabridge实现的简单对比**

这个地方的meta table对比LuaBridge库的实现做了大量的简化，LuaBridge的实现之前有其他研究者整整用了4张纸做了相关的分析。

luabridge的实现很精细的对c++类的各种成员进行了分类处理，不一定能带来多少性能提升，但肉眼可见的导致整个处理机制复杂，难于维护，相关机制定制异常的麻烦。所以lura的实现选用了更简洁和易于维护的方式。当然，这也是得益于c++反射底层打底，bridge的重点落在了了机制串接和lua/c++交互上，对应代码的理解复杂度直线下降，定制性和可维护性也高了非常多。对象创建的时候已经正确关联meta table了，运行时不需要额外的操作，我们就能依赖关联的meta table和meta method来完成在lua中调用c++函数和访问c++成员变量的功能了。

- **C++调用Lua函数**

这块基本就是依赖原生的lua c api: lua_pcall()，方式都比较既定，与本篇的主题内容关联不大，先略过了。

- **Lua协程处理**

lura的协程处理主要完成了两件事:

- 协程池的管理，这部分不管是skynet还是公司内开源的hive，都有良好的实现，可以直接参考。
- 与框架的c++协程保持比较一致的使用方式。

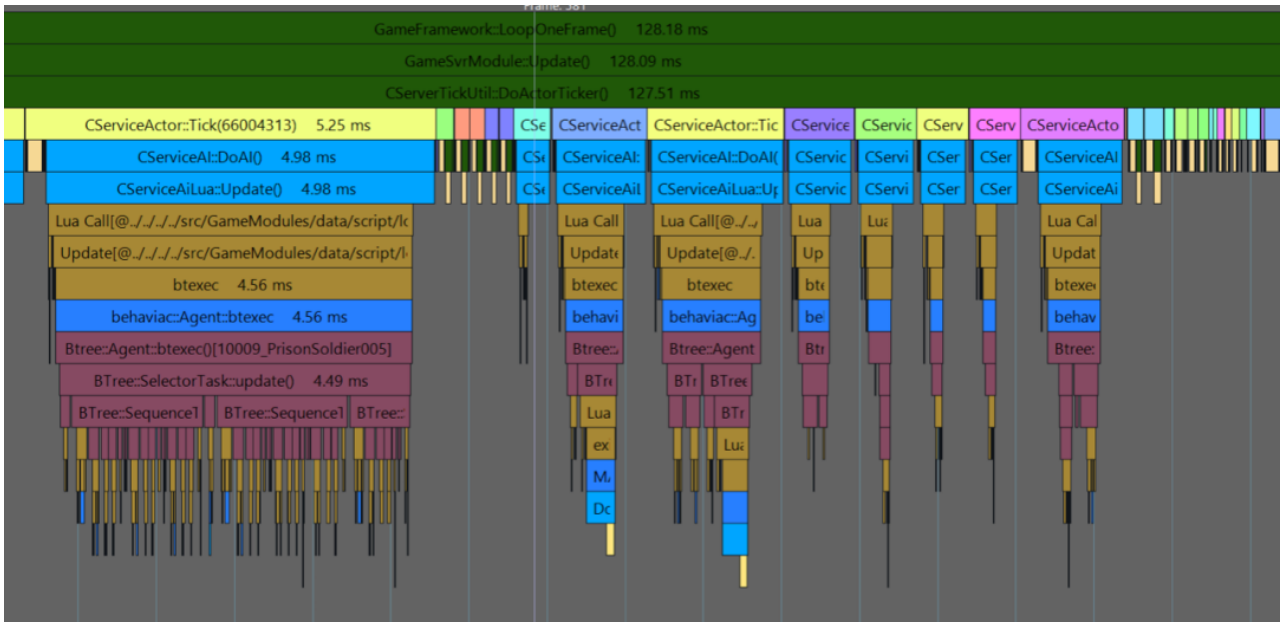
这部分跟本篇的内容关联不大，不详细赘述了。

- **使用时应该遵循的一些基本约束**

主要是注意生命周期的问题，更多依赖值类型，而不是Ref类型来跨边界传递C++对象，先保证生命周期正确，再处理其他问题。集中C++调用Lua的代码，有统一的地方发起相关的使用，也方便问题的定位以及代码本身的迭代。动态语言，依托报错告警机制，建立快速修复的闭环，有问题第一时间解决，反而是符合其动态特性的方式。

- **关于Profiler**

lura本体是直接选择了商用的FramePro，集成了它的SDK。因为跨语言边界处理的代码都非常集中，所以接入其他第三方的profiler也相对容易，这里直接上最后的效果图了:



FramePro本身提供了多种模式，比如在Server端，我们一般是将Profiler数据记录成文件，或者直接使用Remote直连的方式抓取Profiler数据，相关数据也支持直接记录成文件再次打开，使用还是比较方便的。

四、其它

（一）关于其他脚本的bridge实现

对于其它动态类型的脚本语言的Bridge来说，lua的bridge实现具有代表性，差异主要在于:

- 每个脚本提供的c api可能都存在一些差异。
- 语言专有的特性(如Lua专有的非对称协程)

抛开这些差异性的部分，相关的C++函数->脚本函数的类型擦除包装，属性的处理等均具有大量的共性，大家可以自行参考对比，基于反射来实现一个跨多种脚本的Bridge，成本肯定是比裸封装要小非常多的。

总结

通过全篇的分析，我们不难发现，依托于C++反射，实现一个lua的bridge变得非常简单，相当多复杂的模板代码都下沉到了反射本身，代码的层次化变得很好，一般来说，c++/lua交互代码在这种情况下都变得很纯粹，也更容易维护和迭代了。一般lua的bridge也会配合相关的导出器，这也是后续文章会展开的内容。

参考资料:

1.github ponder库

2.luabridge库

作者简介

沈芳

腾讯后台开发工程师

IEG研发效能部开发人员，毕业于华中科技大学。目前负责CrossEngine Server的开发工作，对GamePlay技术比较感兴趣。

推荐阅读

C++反射：深入探究function实现机制！

C++反射：全面解读property的实现机制！

C++反射：深入浅出剖析ponder库实现机制！

从Golang调度器的作者视角探究其设计之道！

面向 js 程序员的 rust 教程分享给大家！

互联网大厂面试

Google C++ 编程风格指南（八）：格式

CPP开发前沿