链滴

Cocos2dx 中 userdata 关联 Peer 表的实现

作者：matengli110

原文链接：https://ld246.com/article/1642669243586

来源网站：链滴

起因

在log里面看到一条报错，大概情况如下，在socket回调里面报了一条attempt to call a nil value(大)。然后类结构大概是这样的：

```
local MessageLayer = class("MessageLayer", CC.Layer)

function MessageLayer:showDisappearEffect()
-- do something
end

local a = class("layer", MessageLayer)

local scene = cc.Director:getInstance():getRunningScene()

local instance = a:create()
instance:addTo(scene)
```

然后这个instance被移除的情况下，它上面的luafunction调不到了。但是这个instance本身还是存在。所以我们关注的问题是：**在lua里，对于一个userData的实例，当C++对象析构了之后，为什么它面的lua方法会失效**

尝试解决

打印了一下userdata的原表，大概这样写：

```
function ThemeScene:onEnter( )

    for k,v in pairs(getmetatable(self)) do
        print(k,v)
    end
end
```

打印出的结果和我想的不太一样。

```
[LUA-print] getDefaultCamera    function: 0x133752a0
[LUA-print] __index    function: 0x13371448
[LUA-print] __newindex    function: 0x13371480
[LUA-print] __gc    function: 0x1336a220
[LUA-print] __eq    function: 0x1336bad8
[LUA-print] __lt    function: 0x1336bbb0
[LUA-print] __le    function: 0x1336bbe8
[LUA-print] getNavMesh    function: 0x13498df0
[LUA-print] __call    function: 0x13374888
[LUA-print] __add    function: 0x1336baa0
[LUA-print] __sub    function: 0x13371410
[LUA-print] __mul    function: 0x1336bb40
[LUA-print] __div    function: 0x1336bb78
[LUA-print] setNavMesh    function: 0x13498d88
[LUA-print] tolua_ubox    table: 0x1336c028
[LUA-print] setNavMeshDebugCamera    function: 0x13498d20
[LUA-print] setPhysics3DDebugCamera    function: 0x13498cb0
[LUA-print] getPhysics3DWorld    function: 0x13498c40
```

```
[LUA-print] createWithPhysics    function: 0x133753b0
[LUA-print] create    function: 0x13375340
[LUA-print] .classname    cc.Scene
[LUA-print] createWithSize    function: 0x13375308
[LUA-print] onProjectionChanged    function: 0x13374a00
[LUA-print] initWithPhysics    function: 0x1336e0c0
[LUA-print] setCameraOrderDirty    function: 0x13374e90
[LUA-print] render    function: 0x13374ef0
[LUA-print] stepPhysicsAndNavigation    function: 0x13374990
[LUA-print] new    function: 0x1336df70
[LUA-print] getPhysicsWorld    function: 0x13374a68
[LUA-print] initWithSize    function: 0x13375230
```

明显可以看出来，基本上所有的东西都是c++里的方法，也就是userdata里的方法，那么我们自己定的一些lua方法去哪里了呢?

前置知识

其实方向一开始就错了。本身userdata上面其实不应该直接绑定任何lua的东西的，正确的做法应该套一层，然后在套娃的那个luatable里来做事，这个套娃一般称之为**peer**

c++_obj的metatable的__index指向一个c函数，当访问c++_obj中的一个域的时候，会调用这个c函，这个c函数会去查找各个关联表，来取得我们要访问的域，这其中就包括对peer表的查询。

具体解决

另外，在functions.lua里其实也可以找到证据

先看下class方法的完整代码

```
function class(classname, ...)
    local cls = {__cname = classname}

    local supers = {...}
    for _, super in ipairs(supers) do
        local superType = type(super)
        assert(superType == "nil" or superType == "table" or superType == "function",
            string.format("class() - create class \"%s\" with invalid super class type \"%s\"",
                classname, superType))

        if superType == "function" then
            assert(cls.__create == nil,
                string.format("class() - create class \"%s\" with more than one creating function",
                    classname));
            -- if super is function, set it to __create
            cls.__create = super
        elseif superType == "table" then
            if super[".isclass"] then
                -- super is native class
                assert(cls.__create == nil,
                    string.format("class() - create class \"%s\" with more than one creating function o
 native class",
                        classname));
                cls.__create = function() return super:create() end
```

```
            else
                -- super is pure lua class
                cls.__supers = cls.__supers or {}
                cls.__supers[#cls.__supers + 1] = super
                if not cls.super then
                    -- set first super pure lua class as class.super
                    cls.super = super
                end
            end
        else
            error(string.format("class() - create class \"%s\" with invalid super type",
                        classname), 0)
        end
    end

    cls.__index = cls
    if not cls.__supers or #cls.__supers == 1 then
        setmetatable(cls, {__index = cls.super})
    else
        setmetatable(cls, {__index = function(_, key)
            local supers = cls.__supers
            for i = 1, #supers do
                local super = supers[i]
                if super[key] then return super[key] end
            end
        end})
    end

    if not cls.ctor then
        -- add default constructor
        cls.ctor = function() end
    end
    cls.new = function(...)
        local instance
        if cls.__create then
            instance = cls.__create(...)
        else
            instance = {}
        end
        setmetatableindex(instance, cls)
        instance.class = cls
        instance:ctor(...)
        return instance
    end
    cls.create = function(_, ...)
        return cls.new(...)
    end

    return cls
end
```

关键关注 setmetatableindex 这里。

他的定义是这样的：

```
local setmetatableindex_
setmetatableindex_ = function(t, index)
    if type(t) == "userdata" then
        local peer = tolua.getpeer(t)
        if not peer then
            peer = {}
            tolua.setpeer(t, peer)
        end
        setmetatableindex_(peer, index)
    else
        local mt = getmetatable(t)
        if not mt then mt = {} end
        if not mt.__index then
            mt.__index = index
            setmetatable(t, mt)
        elseif mt.__index ~= index then
            setmetatableindex_(mt, index)
        end
    end
end
setmetatableindex = setmetatableindex_
```

其实看到这里就很明显了，所有对self所做的操作其实都是放在这个peer表中的。

首先简单些一个lua类，继承自cc.Layer，并且自己在lua里面定义一些方法

```
local testLayer = class("layerClass",cc.Layer)

function testLayer:testFunc()
    print("testFunc")
end
```

然后我们将他实例化：

```
...
local layer = testLayer:create()
```

这个过程发生的事情，可以对照class方法里的东西

```
if not cls.ctor then
        -- add default constructor
        cls.ctor = function() end
    end
    cls.new = function(...)
        local instance
        if cls.__create then
            instance = cls.__create(...)
        else
            instance = {}
        end
        setmetatableindex(instance, cls)
        instance.class = cls
        instance:ctor(...)
        return instance
    end
```

```
    cls.create = function(_, ...)
        return cls.new(...)
    end
```

重点在于这个new方法里，如果他有__create方法，那么就用__create来创建实例。在此例中，便是ccLayer的create方法。它返回了一个layer的userData实例。

接下来的关键，是这个setmetatableindex，这是一个针对userdata重新定义的一个方法。看下它的现

```
setmetatableindex_ = function(t, index)
    if type(t) == "userdata" then
        local peer = tolua.getpeer(t)
        if not peer then
            peer = {}
            tolua.setpeer(t, peer)
        end
        setmetatableindex_(peer, index)
    else
        local mt = getmetatable(t)
        if not mt then mt = {} end
        if not mt.__index then
            mt.__index = index
            setmetatable(t, mt)
        elseif mt.__index ~= index then
            setmetatableindex_(mt, index)
        end
    end
end
setmetatableindex = setmetatableindex_
```

也很好懂，首先如果是要对一个userdata设置metatable,不可以直接设置，而是去创建一个这个userata的peer表。这个peer表可以看作是一个纯粹的luatable，之后所有的操作都是在这个peer表上进的。对userdata做索引，不但可以索引到它自己导出的C++方法，也可以索引到这个peer表。那么对于我们上面例子所定义的testLayer:testFunc它其实就是关联到了peer表上。

那么可以有一个猜想，当C++对象析构的时候，是不是这个peer表被清掉了导致找不到lua方法呢?

简单写个例子佐证一下。

```
local node = cc.Node:create()
    node:addTo(self)

    node._val = 1000

    print("<<<<<<<<<<<<<<<")
    print("node._peer",tolua.getpeer(node))
    print("node._val:",node._val)
    print("<<<<<<<<<<<<<<<")

    self:delayCall(0.01,function()
        node:removeFromParent()

        print("============")
        print("node._peer",tolua.getpeer(node))
```

```
        print("node._val:",node._val)
        print("============")
    end)
```

之后的输出验证了这点：

```
[LUA-print] <<<<<<<<<<<<<
[LUA-print] node._peer    table: 0x1fcb8e50
[LUA-print] node._val:    1000
[LUA-print] <<<<<<<<<<<<<

[LUA-print] ============
[LUA-print] node._peer    nil
[LUA-print] node._val:    nil
[LUA-print] ============
```

C++内部实现

清理的过程

那么当一个CCRef removeFromParent之后,为什么peer表会被清除呢。

```
Ref::~Ref()
{

    ScriptEngineProtocol* pEngine = ScriptEngineManager::getInstance()->getScriptEngine();
    if (pEngine != nullptr && _luaID)
    {
        // if the object is referenced by Lua engine, remove it
        pEngine->removeScriptObjectByObject(this);
    }
...
```

luaEngine里的代码

```
void LuaEngine::removeScriptObjectByObject(Ref* pObj)
{
    _stack->removeScriptObjectByObject(pObj);
    ScriptHandlerMgr::getInstance()->removeObjectAllHandlers(pObj);
}
```

关于scripthandler的东西我们不关心，我们关心的其实是这一句

```
    _stack->removeScriptObjectByObject(pObj);
```

顺着继续看下去

```
void LuaStack::removeScriptObjectByObject(Ref* pObj)
{
    toluafix_remove_ccobject_by_refid(_state, pObj->_luaID);
}
```

再继续

```
TOLUA_API int toluafix_remove_ccobject_by_refid(lua_State* L, int refid)
{
    void* ptr = NULL;
    const char* type = NULL;
    void** ud = NULL;
    if (refid == 0) return -1;

    // get ptr from tolua_refid_ptr_mapping
    lua_pushstring(L, TOLUA_REFID_PTR_MAPPING);
    lua_rawget(L, LUA_REGISTRYINDEX);                       /* stack: refid_ptr */
    lua_pushinteger(L, refid);                              /* stack: refid_ptr refid */
    lua_rawget(L, -2);                                      /* stack: refid_ptr ptr */
    ptr = lua_touserdata(L, -1);
    lua_pop(L, 1);                                          /* stack: refid_ptr */
    if (ptr == NULL)
    {
        lua_pop(L, 1);
        // Lua stack has closed, C++ object not in Lua.
        // printf("[LUA ERROR] remove CCObject with NULL ptr, refid: %d\n", refid);
        return -2;
    }

    // remove ptr from tolua_refid_ptr_mapping
    lua_pushinteger(L, refid);                              /* stack: refid_ptr refid */
    lua_pushnil(L);                                         /* stack: refid_ptr refid nil */
    lua_rawset(L, -3);                  /* delete refid_ptr[refid], stack: refid_ptr */
    lua_pop(L, 1);                                          /* stack: - */


    // get type from tolua_refid_type_mapping
    lua_pushstring(L, TOLUA_REFID_TYPE_MAPPING);
    lua_rawget(L, LUA_REGISTRYINDEX);                       /* stack: refid_type */
    lua_pushinteger(L, refid);                              /* stack: refid_type refid */
    lua_rawget(L, -2);                                      /* stack: refid_type type */
    if (lua_isnil(L, -1))
    {
        lua_pop(L, 2);
        printf("[LUA ERROR] remove CCObject with NULL type, refid: %d, ptr: %p\n", refid, ptr);
        return -1;
    }

    type = lua_tostring(L, -1);
    lua_pop(L, 1);                                          /* stack: refid_type */

    // remove type from tolua_refid_type_mapping
    lua_pushinteger(L, refid);                              /* stack: refid_type refid */
    lua_pushnil(L);                                         /* stack: refid_type refid nil */
    lua_rawset(L, -3);                  /* delete refid_type[refid], stack: refid_type */
    lua_pop(L, 1);                                          /* stack: - */

    // get ubox
    luaL_getmetatable(L, type);                             /* stack: mt */
    lua_pushstring(L, "tolua_ubox");                        /* stack: mt key */
    lua_rawget(L, -2);                                      /* stack: mt ubox */
```

```
    if (lua_isnil(L, -1))
    {
        // use global ubox
        lua_pop(L, 1);                                  /* stack: mt */
        lua_pushstring(L, "tolua_ubox");                        /* stack: mt key */
        lua_rawget(L, LUA_REGISTRYINDEX);                       /* stack: mt ubox */
    };


    // cleanup root
    tolua_remove_value_from_root(L, ptr);

    lua_pushlightuserdata(L, ptr);                          /* stack: mt ubox ptr */
    lua_rawget(L,-2);                                   /* stack: mt ubox ud */
    if (lua_isnil(L, -1))
    {
        // Lua object has released (GC), C++ object not in ubox.
        //printf("[LUA ERROR] remove CCObject with NULL ubox, refid: %d, ptr: %x, type: %s\n",
efid, (int)ptr, type);
        lua_pop(L, 3);
        return -3;
    }

    // cleanup peertable
    lua_pushvalue(L, LUA_REGISTRYINDEX);
    lua_setfenv(L, -2);

    ud = (void**)lua_touserdata(L, -1);
    lua_pop(L, 1);                                  /* stack: mt ubox */
    if (ud == NULL)
    {
        printf("[LUA ERROR] remove CCObject with NULL userdata, refid: %d, ptr: %p, type: %s\n"
 refid, ptr, type);
        lua_pop(L, 2);
        return -1;
    }

    // clean userdata
    *ud = NULL;

    lua_pushlightuserdata(L, ptr);                          /* stack: mt ubox ptr */
    lua_pushnil(L);                                 /* stack: mt ubox ptr nil */
    lua_rawset(L, -3);                          /* ubox[ptr] = nil, stack: mt ubox */

    lua_pop(L, 2);
    //printf("[LUA] remove CCObject, refid: %d, ptr: %x, type: %s\n", refid, (int)ptr, type);
    return 0;
}
```

这里清除了好几个东西，我们一个个来看。对lua栈操作不熟悉的可以参考这个参考文档

如果需要看5.1的参考，参考这个文档

首先明确的是，这个refid是CCRef里的luaid。

先看下第一部分

```
// get ptr from tolua_refid_ptr_mapping
    lua_pushstring(L, TOLUA_REFID_PTR_MAPPING);
    lua_rawget(L, LUA_REGISTRYINDEX);                           /* stack: refid_ptr */
    lua_pushinteger(L, refid);                                  /* stack: refid_ptr refid */
    lua_rawget(L, -2);                                          /* stack: refid_ptr ptr */
    ptr = lua_touserdata(L, -1);
    lua_pop(L, 1);                                              /* stack: refid_ptr */
    if (ptr == NULL)
    {
        lua_pop(L, 1);
        // Lua stack has closed, C++ object not in Lua.
        // printf("[LUA ERROR] remove CCObject with NULL ptr, refid: %d\n", refid);
        return -2;
    }
```

tolua在lua5.1的binding里面有一个重要的概念就是这个LUA_REGISTRYINDEX为index的全局注册。我们姑且把这个表称之为为Reg，暂时把他理解为一个luatable。

通过阅读可以知道，他其实是拿出 Reg[TOLUA_REFID_PTR_MAPPING][refid]存到ptr这个变量里。

接下来，它清除了Reg[TOLUA_REFID_PTR_MAPPING][refid]，也就是相当于调用了Reg[TOLUA_REFID_PTR_MAPPING][refid] = nil

```
// remove ptr from tolua_refid_ptr_mapping
    lua_pushinteger(L, refid);                                  /* stack: refid_ptr refid */
    lua_pushnil(L);                                             /* stack: refid_ptr refid nil */
    lua_rawset(L, -3);                  /* delete refid_ptr[refid], stack: refid_ptr */
    lua_pop(L, 1);                                              /* stack: - */
```

接下来的代码同理，只是换了一个表清.这次取得是Reg[``TOLUA_REFID_TYPE_MAPPING][refid`]，面存着一个string.

```
// get type from tolua_refid_type_mapping
    lua_pushstring(L, TOLUA_REFID_TYPE_MAPPING);
    lua_rawget(L, LUA_REGISTRYINDEX);                           /* stack: refid_type */
    lua_pushinteger(L, refid);                                  /* stack: refid_type refid */
    lua_rawget(L, -2);                                          /* stack: refid_type type */
    if (lua_isnil(L, -1))
    {
        lua_pop(L, 2);
        printf("[LUA ERROR] remove CCObject with NULL type, refid: %d, ptr: %p\n", refid, ptr);
        return -1;
    }

    type = lua_tostring(L, -1);
    lua_pop(L, 1);                                              /* stack: refid_type */

    // remove type from tolua_refid_type_mapping
    lua_pushinteger(L, refid);                                  /* stack: refid_type refid */
    lua_pushnil(L);                                             /* stack: refid_type refid nil */
    lua_rawset(L, -3);                  /* delete refid_type[refid], stack: refid_type */
    lua_pop(L, 1);                                              /* stack: - */
```

接下来是拿出ubox表。

这个ubox表有两种存的方法，有可能存在Reg[type]["tolua_ubox"]里，也有可能直接存在Reg[“tola_ubox”]。如果前者不存在，那么就拿后者。

```
// get ubox
    luaL_getmetatable(L, type);                         /* stack: mt */
    lua_pushstring(L, "tolua_ubox");                    /* stack: mt key */
    lua_rawget(L, -2);                                  /* stack: mt ubox */
    if (lua_isnil(L, -1))
    {
        // use global ubox
        lua_pop(L, 1);                                  /* stack: mt */
        lua_pushstring(L, "tolua_ubox");                /* stack: mt key */
        lua_rawget(L, LUA_REGISTRYINDEX);               /* stack: mt ubox */
    };
```

之后，拿着这个ubox表，清除tolua_value_from_root这个表，也就是tolua_remove_value_from_root方法里写的。具体如下

```
tolua_remove_value_from_root(L, ptr);
```

具体定义

其实只干了一件事，那就是执行了 Reg[TOLUA_VALUE_ROOT][prt]=nil，也就是把value_root表里ptr对应的项置空了。

```
TOLUA_API void tolua_remove_value_from_root (lua_State* L, void* ptr)
{
    lua_pushstring(L, TOLUA_VALUE_ROOT);
    lua_rawget(L, LUA_REGISTRYINDEX);                   /* stack: root */
    lua_pushlightuserdata(L, ptr);                      /* stack: root ptr */

    lua_pushnil(L);                                     /* stack: root ptr nil */
    lua_rawset(L, -3);                                  /* root[ptr] = nil, stack: root */
    lua_pop(L, 1);
}
```

接下来，就要在ubox表里取出来我们ptr所对应的值

```
lua_pushlightuserdata(L, ptr);                          /* stack: mt ubox ptr */
    lua_rawget(L,-2);                                   /* stack: mt ubox ubox[ptr] */
    if (lua_isnil(L, -1))
    {
        // Lua object has released (GC), C++ object not in ubox.
        //printf("[LUA ERROR] remove CCObject with NULL ubox, refid: %d, ptr: %x, type: %s\n",
efid, (int)ptr, type);
        lua_pop(L, 3);
        return -3;
    }
```

接下来就是我们的重点，peer表。此时我们的栈顶是 reg ubox ubox[ptr]

这里先把lua的全局注册表push进站，然后把他作为ubox[ptr]的新的lua_env

```
// cleanup peertable
```

```
    lua_pushvalue(L, LUA_REGISTRYINDEX);
    lua_setfenv(L, -2);
```

然后在注册表的env之下，把顶层指向userdata的指针,也就是ubox[ptr]取出来，把它所指向的userda a置空。

```
ud = (void**)lua_touserdata(L, -1);
    lua_pop(L, 1);                                  /* stack: mt ubox */
    if (ud == NULL)
    {
        printf("[LUA ERROR] remove CCObject with NULL userdata, refid: %d, ptr: %p, type: %s\n"
 refid, ptr, type);
        lua_pop(L, 2);
        return -1;
    }

    // clean userdata
    *ud = NULL;
```

最后，再把ubox里以ptr为key的值置空。ubox[ptr]=nil

```
lua_pushlightuserdata(L, ptr);                      /* stack: mt ubox ptr */
    lua_pushnil(L);                                 /* stack: mt ubox ptr nil */
    lua_rawset(L, -3);                      /* ubox[ptr] = nil, stack: mt ubox */
```

至此，整个清理过程完成。

创建绑定的过程

经过上面代码的阅读我们明白了lua部分是如何清理的，对需要清理哪些表有了一个大概的了解。但我们还需要知道一件事情：那就是我们的userdata是怎么调到peer表的。

那么相应的，我们需要看一下在绑定过程中，userdata到底需要绑定哪些东西。

从开始注册绑定开始

其实观察下lua的注册过程，会发现这个东西：

```
TOLUA_API void tolua_usertype (lua_State* L, const char* type)
{
    char ctype[128] = "const ";
    strncat(ctype,type,120);

    /* create both metatables */
    if (tolua_newmetatable(L,ctype) && tolua_newmetatable(L,type))
        mapsuper(L,type,ctype);             /* 'type' is also a 'const type' */
}
```

然后重点是tolua_newmetatable

```
static int tolua_newmetatable (lua_State* L, const char* name)
{
    int r = luaL_newmetatable(L,name);
```

```
#ifdef LUA_VERSION_NUM /* only lua 5.1 */
    if (r) {
        lua_pushvalue(L, -1);
        lua_pushstring(L, name);
        lua_settable(L, LUA_REGISTRYINDEX); /* reg[mt] = type_name */
    };
#endif

    if (r)
        tolua_classevents(L); /* set meta events */

    // metatable[".classname"] = name
    lua_pushliteral(L, ".classname");   // stack: metatable ".classname"
    lua_pushstring(L, name);            // stack: metatable ".classname" name
    lua_rawset(L, -3);                  // stack: metatable

    lua_pop(L,1);
    return r;
}
```

其中的tolua_classevents

```
TOLUA_API void tolua_classevents (lua_State* L)
{
    lua_pushstring(L,"__index");
    lua_pushcfunction(L,class_index_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__newindex");
    lua_pushcfunction(L,class_newindex_event);
    lua_rawset(L,-3);

    lua_pushstring(L,"__add");
    lua_pushcfunction(L,class_add_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__sub");
    lua_pushcfunction(L,class_sub_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__mul");
    lua_pushcfunction(L,class_mul_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__div");
    lua_pushcfunction(L,class_div_event);
    lua_rawset(L,-3);

    lua_pushstring(L,"__lt");
    lua_pushcfunction(L,class_lt_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__le");
    lua_pushcfunction(L,class_le_event);
    lua_rawset(L,-3);
    lua_pushstring(L,"__eq");
    lua_pushcfunction(L,class_eq_event);
    lua_rawset(L,-3);
```

```
    lua_pushstring(L,"__call");
    lua_pushcfunction(L,class_call_event);
    lua_rawset(L,-3);

    lua_pushstring(L,"__gc");
    lua_pushstring(L, "tolua_gc_event");
    lua_rawget(L, LUA_REGISTRYINDEX);
    /*lua_pushcfunction(L,class_gc_event);*/
    lua_rawset(L,-3);
}
```

是不是这里的内容就很熟悉了，正如我们正常操作原表来做继承的过程。只不过这里很多东西都会存注册表里，比纯粹lua里面的继承要复杂很多。

那当然我们要看下熟悉的__index方法注册的静态方法。

```
static int class_index_event (lua_State* L)
{
    int t = lua_type(L,1);
    if (t == LUA_TUSERDATA)
    {
        /* Access alternative table */
#ifdef LUA_VERSION_NUM /* new macro on version 5.1 */
        lua_getfenv(L,1);
        if (!lua_rawequal(L, -1, TOLUA_NOPEER)) {
            lua_pushvalue(L, 2); /* key */
            lua_gettable(L, -2); /* on lua 5.1, we trade the "tolua_peers" lookup for a gettable call *

            if (!lua_isnil(L, -1))
                return 1;
        };
#else
        lua_pushstring(L,"tolua_peers");
        lua_rawget(L,LUA_REGISTRYINDEX);        /* stack: obj key ubox */
        lua_pushvalue(L,1);
        lua_rawget(L,-2);                       /* stack: obj key ubox ubox[u] */
        if (lua_istable(L,-1))
        {
            lua_pushvalue(L,2);  /* key */
            lua_rawget(L,-2);                   /* stack: obj key ubox ubox[u] value */
            if (!lua_isnil(L,-1))
                return 1;
        }
#endif
        lua_settop(L,2);                        /* stack: obj key */
        /* Try metatables */
        lua_pushvalue(L,1);                     /* stack: obj key obj */
        while (lua_getmetatable(L,-1))
        {   /* stack: obj key obj mt */
            lua_remove(L,-2);                   /* stack: obj key mt */
            if (lua_isnumber(L,2))              /* check if key is a numeric value */
            {
                /* try operator[] */
                lua_pushstring(L,".geti");
                lua_rawget(L,-2);               /* stack: obj key mt func */
```

```
            if (lua_isfunction(L,-1))
            {
                lua_pushvalue(L,1);
                lua_pushvalue(L,2);
                lua_call(L,2,1);
                return 1;
            }
        }
        else
        {
            lua_pushvalue(L,2);                 /* stack: obj key mt key */
            lua_rawget(L,-2);                   /* stack: obj key mt value */
            if (!lua_isnil(L,-1))
                return 1;
            else
                lua_pop(L,1);
            /* try C/C++ variable */
            lua_pushstring(L,".get");
            lua_rawget(L,-2);                   /* stack: obj key mt tget */
            if (lua_istable(L,-1))
            {
                lua_pushvalue(L,2);
                lua_rawget(L,-2);                   /* stack: obj key mt value */
                if (lua_iscfunction(L,-1))
                {
                    lua_pushvalue(L,1);
                    lua_pushvalue(L,2);
                    lua_call(L,2,1);
                    return 1;
                }
                else if (lua_istable(L,-1))
                {
                    /* deal with array: create table to be returned and cache it in ubox */
                    void* u = *((void**)lua_touserdata(L,1));
                    lua_newtable(L);                /* stack: obj key mt value table */
                    lua_pushstring(L,".self");
                    lua_pushlightuserdata(L,u);
                    lua_rawset(L,-3);               /* store usertype in ".self" */
                    lua_insert(L,-2);               /* stack: obj key mt table value */
                    lua_setmetatable(L,-2);         /* set stored value as metatable */
                    lua_pushvalue(L,-1);            /* stack: obj key met table table */
                    lua_pushvalue(L,2);             /* stack: obj key mt table table key */
                    lua_insert(L,-2);               /*  stack: obj key mt table key table */
                    storeatubox(L,1);               /* stack: obj key mt table */
                    return 1;
                }
            }
        }
        lua_settop(L,3);
    }
    lua_pushnil(L);
    return 1;
}
else if (t== LUA_TTABLE)
```

```
    {
        lua_pushvalue(L,1);
        class_table_get_index(L);
        return 1;
    }
    lua_pushnil(L);
    return 1;
}
```

不要被长段代码吓到，其实我们关注的只有这些代码

```
if (t == LUA_TUSERDATA)
    {
        /* Access alternative table */
#ifdef LUA_VERSION_NUM /* new macro on version 5.1 */
        lua_getfenv(L,1);
        if (!lua_rawequal(L, -1, TOLUA_NOPEER)) {
            lua_pushvalue(L, 2); /* key */
            lua_gettable(L, -2); /* on lua 5.1, we trade the "tolua_peers" lookup for a gettable call *

            if (!lua_isnil(L, -1))
                return 1;
        };
```

其实这里就很明显了，就是拿下这个userdata对应的env表。然后如果没有设置过peer表，此时的en就是TOLUA_NOPEER，TOLUA_NOPEER其实就是LUA_REGISTRYINDEX

```
#define TOLUA_NOPEER    LUA_REGISTRYINDEX /* for lua 5.1 */
```

然后在这个env表中索引。注意,lua_gettable是会触发元方法的，比如__index。这也就是为什么我们在前面的lua代码中看到我们的peer上不会直接设置属性，而是给他设置原表通过__index触发。

同样的,__newindex也是同样的套路，这里就不展开说明了，读者可以自行考究。

tolua中的peer函数

最后，为了证明我们的想法，我们看下tolua中的getpeer和setpeer，来佐证我们的想法。

```
#ifdef LUA_VERSION_NUM /* lua 5.1 */
        tolua_function(L, "setpeer", tolua_bnd_setpeer);
        tolua_function(L, "getpeer", tolua_bnd_getpeer);
#endif
```

setpeer的实现

```
static int tolua_bnd_setpeer(lua_State* L) {

    /* stack: userdata, table */
    if (!lua_isuserdata(L, -2)) {
        lua_pushstring(L, "Invalid argument #1 to setpeer: userdata expected.");
        lua_error(L);
    };

    if (lua_isnil(L, -1)) {
```

```
        lua_pop(L, 1);
        lua_pushvalue(L, TOLUA_NOPEER);
    };
    lua_setfenv(L, -2);

    return 0;
};
```

getpeer的实现

```
static int tolua_bnd_getpeer(lua_State* L) {

    /* stack: userdata */
    lua_getfenv(L, -1);
    if (lua_rawequal(L, -1, TOLUA_NOPEER)) {
        lua_pop(L, 1);
        lua_pushnil(L);
    };
    return 1;
};
```

实现与我们前面的观察基本一致，也是利用了fenv来造一个peer_table。当然这只是lua5.1中的实现，ua后续的版本已经不用这套实现了，但是基本思路也大同小异，读者可以自行考究。

结

没想到一个简单的问题引起了这么多看代码的过程。纸上得来终觉浅，绝知此事要躬行。如果只是浅辄止地看了下大概的实现，可能这次看代码经历也就没什么意义了。希望以后遇到了一个看起来简单者直觉型的问题能够有时间，也有心思去细细看一遍整个代码的流程。带着问题多看源码永远都是一好处理方法。希望今后遇到问题能多看看源码，有时候也没有想象中那么复杂。