坑1：NSString、NSMutableString与LuaString的自动转换

从OC中转到Lua中的String变量在Lua中不能直接调用NSString的方法，要先将LuaString转换为NSString对象。

示例：获取NSString的长度

local var = NSString:stringwithCString(“testString”)

local length = toobjc(var):length()

说明：toobjc是wax提供的工具，可以将Lua中的变量转换为相应的OC对象。

坑2：NSArray、NSDictionary与LuaTable的自动转换

同坑1。

如果在Lua中有local test1 = {}; test[1] = 0; test[2] = 0; 那么test1被传到oc时，会自动转换成NSArray。

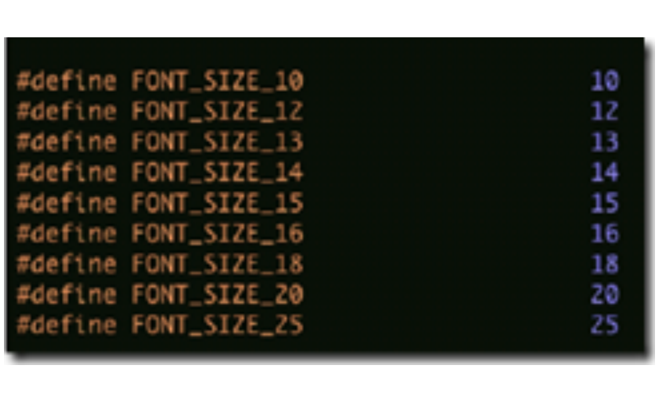
如果在Lua中有local test2 = {}; test[“x”] = 0; test[“y”] = 0;那么test2被传到oc时，会自动转换成NSDictionary。

坑3：在OC中定义的宏无法在Lua中直接使用

如果我们想用那些宏，就得在Lua中重新声明。

如：

我们在OC中定义了字体大小的宏如图1



```
#define FONT_SIZE_10          10
#define FONT_SIZE_12          12
#define FONT_SIZE_13          13
#define FONT_SIZE_14          14
#define FONT_SIZE_15          15
#define FONT_SIZE_16          16
#define FONT_SIZE_18          18
#define FONT_SIZE_20          20
#define FONT_SIZE_25          25
```

图1

如图2，则是在Lua中定义的相同的“宏”（26～34行）

```
1   require "HEEnum"
2   require "HEStruct"
3
4   PLIST_BACKGROUND_BLINK     = "201010.plist"
5   UINOTIFACTIONSYSTEM_PLIST  = "UINotifactionSystem.plist"
6   H_NC_MEDIATOR              = "NCMediator"
7   NEWS_BG                    = "news_bg.png"
8
9   PNG_BACKGROUND_BLINK       = "201010.png"
10
11  FONT_NAME_GEORGIA_BOLDITALIC  = "Georgia-BoldItalic"
12  FONT_COLOR_GEORGIA            = _ccColor3B(156, 72, 43)
13
14  TOUCH_PRIORITY_BUTTON      = kCCMenuTouchPriority
15  DIALOG_PRIORITY_OTHER      = 0 --DIALOG_PRIORITY_OUT_TANK_BASE
16
17  TEXT_TOTAL_UI_TABLE_NAME   = "TextTotalUI"
18  BTN_VISITFRIEND            = "btn_visitfriend"
19  SOCIAL_FB_NETWORK_ERROR    = "social_fb_network_error"
20  NC_FRIEND_LOADING          = "nc_friend_loading"
21  NC_TITLE                   = "nc_title"
22
23  kButtonClickSound          = "button_click.aac"
24  kBackClickSound            = "close_panel.aac"
25
26  FONT_SIZE_10  = 10
27  FONT_SIZE_12  = 12
28  FONT_SIZE_13  = 13
29  FONT_SIZE_14  = 14
30  FONT_SIZE_15  = 15
31  FONT_SIZE_16  = 16
32  FONT_SIZE_18  = 18
33  FONT_SIZE_20  = 20
34  FONT_SIZE_25  = 25
```

图2

坑4：在OC中定义的enum（枚举）无法在Lua中直接使用，我们也得必须在Lua中重新定义。

如果你发现在使用系统的枚举时，能直接使用，告诉你，那是因为Wax已经为我们重新定义部分的OC枚举类型，可以去APP_ROOT/wax/lib/stdlib/enums.lua中查证。

另外，要说明的是stdlib中的所有lua文件都要被硬编码（转换成二进制后，新生成wax_stdlib.h文件）后才能被使用，而编码工具是APP_ROOT/wax/lib/build-scripts/compile-stdlib.sh，笔者尝试过使用，但一直失败，似乎失败的结果会很严重，所以笔者建议不要随便使用。

笔者采用的方法是自己新件一个lua文件，当然是资源的一部分，用的时候require(“…”)就好。

如：

在OC中定义一个枚举如图3：

图3

在Lua中则定义如图4中的(9~18行)：

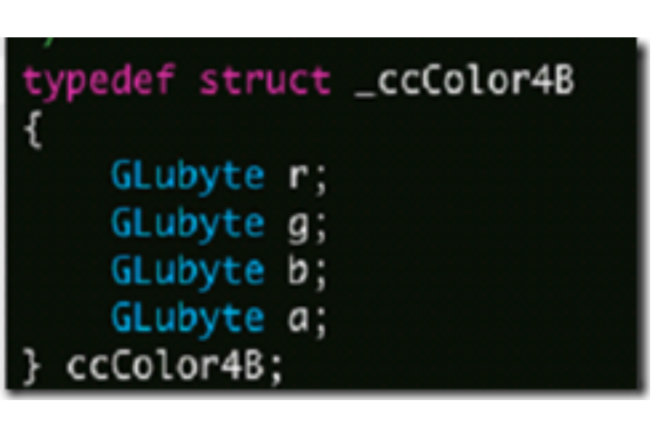
图4

坑5：在OC中的结构体也不能直接被Lua调用，也是要重新定义的。

像CGSize，CGPoint，CGRect… 这些OC自带的结构体，Wax已经帮我们生成好了，详见APP_ROOT/wax/lib/stdlib/structs.lua

如：

在OC中定义了如图5的结构体：



```
typedef struct _ccColor4B
{
    GLubyte r;
    GLubyte g;
    GLubyte b;
    GLubyte a;
} ccColor4B;
```

图5

在Lua中只要调用下Wax提供的一个方法就能轻易定义，如图6（第2行）：

```
1   wax.struct.create("_ccColor3B", "CCC", "r", "g", "b")
2   wax.struct.create("_ccColor4B", "CCCC", "r", "g", "b", "a")
3
```

图6

说明：wax.struct.create表示要定义一个结构体，第一个参数表示结构体名称，第二个参数表示结构体中的成员个数与相应的类型，后面的参数个数要与成员个数相同，表示各成员变量名。

第二个参数的类型说明字符意义：f表示float，d表示double，I(大写i)表示int，C表示char，B表示BOOL。

有了这个，像要创建一个结构体变量，如_ccColor4B(1,1,1,1)即可，不要用ccc4这样的在OC里用宏定义的方法。

坑6：在坑5中有个更深的坑，就是第二个参数的类型C与类型B的意义不同的，看例子理解吧。(笔者是较早的时候认为B表示Byte，才有这坑6)

如：

如果定义wax.struct.create(“test”,”CC”,”a”,”b”)，

再定义local var = test(48, 54)，此时将var.a与var.b输出分别为’0’和’3’，这是ASCII码转成字符，值是正常的。

但如果定义wax.struct.create(“test”,”BB”,”a”,”b”),

再定义local var = test(48, 54)，此时将var.a与var.b输出分别为’4’和’5’，这就有问题了。原因就是这个时候，Lua将每个参数中的第一个字符取出来当作字符赋值结结构体变量，剩下的字符全被丢弃了。

故，要谨慎！！

坑7：在Lua中使用waxClass时，要注意此时生成的类不能和OC代码中的类名有重复，否则在Lua中会生成不成功。

坑8：OC中使用了va_list参数的方法不能被Lua识别。

如：

在OC中，NSMutableArray中的arrayWithObjects:(id)firstObj, …

在Lua中只能先创建tabel后，再一个一个地加入表中。这里要注意的是，可以直接把tabel当成NSArray或NSDictionary传进去！为什么？见坑2。

坑9：在OC中已经实现了的extern，inline等非类中的方法不能在Lua中直接调用。

如果你想问为什么类的方法为什么能直接调用，那是因为Wax会动态解释Lua中对象的方法，再采用反射机制为我们转义成正确的方法名并调用！！！那这时怎么解决调用非类方法的方法？手动注册这样的方法实际就是普通的C函数，所以我们要做的事情就是C与Lua的交互，就是要对C函数进行封装。

在看网上的配置教程中，启动Lua脚本的示例如：wax_start(“AppDelegate.lua”, luaopern_wax_http, luaopen_wax_json, nil);这行代码，第一个参数表示要运行的lua脚本，后面的参数表示要使用的扩展库，扩展库的路径：APP_ROOT/wax/lib/extensions/，在这里放了一些扩展库。

那解决坑9的问题，笔者是能扩展库的形式解决问题的。在extensions/下，新建了一个”HETool”的文件夹，在这里新建了文件”wax_HEUtills.h”和”wax_HEUtills.m”。如图7所示：

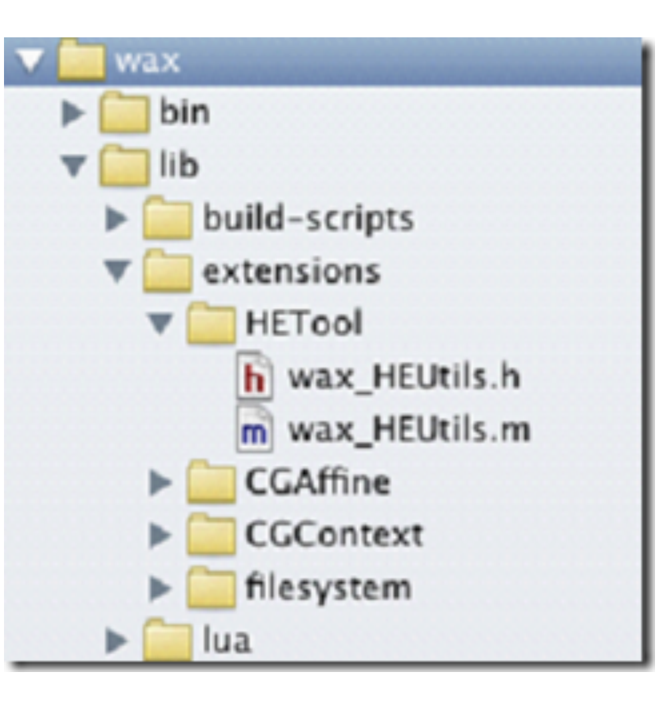


图7

在头文件中只声明一个函数即可，这个函数用来启动注册方法的，如图8：

```
//
//  Created by Guorui.Chen on 20/03/13.
//  Copyright 2013年 Fortuna. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "lua.h"

int luaopen_wax_HETool(lua_State *L);
```

图8

然后在.m文件中引入如下的头文件和宏定义，其余要引入哪些头文件，视功能而定，其中TABLE_NAME表示把方法注册到Lua表中的表名，这个很重要，因为调用方法时，要带这个前缀。如图9：

```
#import "wax_HEUtils.h"

#import "wax_instance.h"
#import "wax_helpers.h"

#import "lua.h"
#import "lauxlib.h"

#define TABLE_NAME "wax.HETool"
```

图9

如为了让Lua正常使用OC中的getLocalizedString(NSString *p1, NSString *p2)这个方法，这里在wax_HEUtils.m中定义了wax_getLocalizedString(…)的方法，如图10：

```
static int wax_getLocalizedString(lua_State *L) {
    const char* tbl = luaL_checkstring(L, 1);
    const char* key = luaL_checkstring(L, 2);

    NSString *ret = getLocalizedString(
            [NSString stringWithUTF8String:tbl],
            [NSString stringWithUTF8String:key]);

    lua_pushstring(L, [ret UTF8String]);
    return 1;
}
```

图10

这里的参数lua_State表示从Lua传到OC来的Lua环境指针。图10中的第1行和第2行是从lua中取出参数（要取其它类型的参数，调用luaL_checkxxxx对号入座就行），这里要注意的是，Lua的下标起始值是1，不是0哦！！！将这两个参数作为getLocalizedString方法的参数，并调用之，就能获取本地化的字符串了，然后调用lua_pushxxxxxx（类型也是对号入座）即可到达在Lua中return 一个值的效果。

这里注意的是，Lua的return值不仅仅只能有一个哦，可以有多个，所以当你lua_pushxxxxxx调用多次里，在Lua中能相应的return多个值。

函数结尾要return 1， 否则在注册时会认为注册失败而不写入Lua的函数表中。

接下来，声明一个Lua函数注册列表，如图11：

```
static const struct luaL_Reg functions[] = {
    {"wax_getLocalizedString", wax_getLocalizedString},

    {NULL, NULL}
};
```

图11

最后一定要以{NULL, NULL}结束。在{“wax_getLocalizedString”, wax_getLocalizedString}中，前者表示在Lua的函数表中的是后者的映射名称，意思就是要在Lua中调用OC中wax_getLocalizedString这个方法时，要写前对应的前者名称，Lua即根据名称调用相应的方法。如果前者名称发为”getLocalizedString”的话，在图14中第98行处则相应的改成wax.HETool.getLocalizedString(…)了。

然后实现luaopen_wax_HETool方法即可，如图12所示：

```
int luaopen_wax_HETool(lua_State *L) {
    BEGIN_STACK_MODIFY(L);

    luaL_register(L, TABLE_NAME, functions);

    END_STACK_MODIFY(L, 0)

    return 1;
}
```

图12

到此，完成了OC的全局方法在Lua中调用准备工作。只要我们在需要的地方开启该扩展库即可，（别忘记了要引入HETool.h哦）如图13所示：

```
wax_start("RunNCScene.lua", luaopen_wax_HETool, nil);
```

图13

那么我们就能使用getLocalizedString方法了，如图14第98行所示：

图14

说明：第98行中，wax.HETool.xxxxxxxxx表示要调用wax.HETool中的函数，要与图9中的TABLE_NAME保持一致，xxxxxxxx表示要调用的方法。

坑10：OC中的方法名带有了下划线。

由于OC的方法中会给所有的参数给出参数名，如

-(void) perform:(SEL)sel withTarget:(id)target;

Wax针对这样设定给出的解决方案是将所有的冒号转换成下划线，如

xx:perform_withTarget(sel, target);

再通过逆转换后根据SelectorName来反射到原始的方法，以达到让Lua正常调用OC方法的效果，如

“perform:withTarget:”

那么现在遇到的问题的，如果方法中原本就带有下划线时，如

-(void) setA_B:(int)a withB:(int)b;

经过转义后，在Lua中呈现形式如

xx:setA_B_withB(a, b);

再逆转换回OC方法时，就会变成下面的SelectorName，这就与原OC方法不同了，导致无法正常调用。

"setA:B:withB:”

坑11：调试过程中由于多线程而出现的问题。

尝试在HappyFish中进行Lua调试的过程中，出现了一个问题：

IDE驱动开启NCSCene界面时，总会有图片显示不全的问题，但使用模拟器自己启动，则正常

原因：调试过程中是使用多线程来监听网络，一旦IDE对模拟器进行操作，这些操作是在当前线程（也就是子线程）中执行，而cocos2d是基于主线程的引擎，如果在子线程中加载了资源，那在主线程使用将会出现异常

解决：将IDE对模拟器的操作进行封装，采用[NSObject performSelectorOnMainThread….]（在RpcHandlerWapper类中封装）的方法将操作转至主线程，保证执行Lua脚本的线程是主线程。

坑12：调试过程中由于多线程而出现的问题。

在一个Lua文件中定义的类（如LuaFishEngineBase）下定义了一个局部变量（如local isRushing_），该变量的声明相当于private，所以在Lua中的该类的子类也不能直接访问，若要访问，必须加上setters/getters。另外，类的变量声明没有protected和public特性，只有相当于private和public static特性。

坑13：在Lua中实现协议后，运行时报Runtime couldn’t find protocl。

这是实际是Wax库的一个Bug。大部分情况下，在OC声明的协议能为wax所识别，但依然有少部分情况是例外的，当遇到这种例外情况时（如IDeerEngine）,我们要在ProtocolLoader.h中手动加入协议的加载，具体方式参照该文件即可。

在图6中第2行为什么定义结构体的名称为_ccColor4B，而不是ccColor4B呢？如果你要在Lua中获得OC对象的该结构体的变量时，Lua会获取到_ccColor4B这个结构体名称，而不是ccColor4B，进而在注册的结构体列表中寻找相同的结构体名称，以完成从OC到Lua的转换。