Lua热修复实现原理

Lua 热修复实现原理

Lua 语言特性简单介绍

Lua 是一种轻量小巧的脚本语言，用标准 C 语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。游戏领域里经常可以看到 Lua 的身影，比如：魔兽世界里的 Lua 脚本插件、Unity 里的 XLua 热修复框架。

官网是 https://www.lua.org/

这里选用的是 Lua 5.4.3 版本，Lua 5.4 与 Lua 5.1 相比做了内存优化，并且拥有更小的内存占用，更快的执行效率（提升40%），更有效的垃圾回收机制。

数据类型

数据类型	描述
nil	表示一个空值（在条件表达式中相当于false）
boolean	包含两个值：false 和 true
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
function	由 C 或 Lua 编写的函数
userdata	表示任意存储在变量中的 C 数据结构
thread	表示执行的独立线路，用于执行协同程序
table	Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表

表

Lua 里的表，类似于 JS 里的 json 格式，但是不仅仅只是表示 json。

一个 Lua 的表结构:

```
-- 局部变量 tab1 表示的是一个 key-value 结构的表，功能跟 json 一样
local tab1 = { key1 = "val1", key2 = "val2"}
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end

-- 局部变量 tbl 表示的是一个只有 value 的数组。
-- 不同于其他语言的数组把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。
local tbl = {"apple", "pear", "orange", "grape"}
for index, val in pairs(tbl) do
   print(index .. " - " .. val)
end
```

函数

跟 JS 一样，Lua 的函数也是一等公民，Lua 里的函数可以有多个返回值

```
--[[ 函数返回两个值的最大值 --]]
function max(num1, num2)

   if (num1 > num2) then
      result = num1;
   else
      result = num2;
   end

   return result;
end
-- 调用函数
print("两值比较最大值为 ",max(10,4))
```

元表

Lua 里的元表，可以看作是为表提供了一组改变行为的函数，比如想要把两个表进行相加，需要添加 `__add` 的元事件。

```
local tab1 = { key1 = "val1", key2 = "val2"}
local tab2 = { key3 = "val3", key4 = "val4"}

local metatable = {
    __add = function(mytable, newtable)
      for k,v in pairs(newtable) do
        mytable[k] = v
      end

      return mytable
    end
  }

-- 两表相加操作
setmetatable(tab1, metatable)
local mytable = tab1 + tab2
for k,v in pairs(mytable) do
  print(k .. " - " .. v)
end
```

而这里主要是要介绍 3 个元事件：

- __index: 索引 table[key]。 当表 table 中不存在 key 这个键时，这个事件被触发。 这个事件的元方法其实可以是一个函数也可以是一张表。 如果它是一个函数，则以 table 和 key 作为参数调用它。如果它是一张表，最终的结果就是以 key 取索引这张表的结果。
- __newindex: 索引赋值 table[key] = value 。 和索引事件类似，它发生在表 table 中不存在 key 这个键的时候。 同索引过程那样， 这个事件的元方法即可以是函数，也可以是一张表。 如果是一个函数， 则以 table、 key、以及 value 为参数传入。 如果是一张表， Lua 对这张表做索引赋值操作。
- __call: 函数调用操作 func(args)。 当 Lua 尝试调用一个非函数的值的时候会触发这个事件 （即 func 不是一个函数）。

下面代码演示下，这个 3 个事件的具体作用：

```
local tab = {}
local metatable = {
  __index = function(table, key)
```

```
    -- 正在索引取值 key3
    print("正在索引取值 "..key)
    return "val3"
  end,
  __newindex = function(table, key, value)
    -- 正在索引赋值 key4 - val4
    print("正在索引赋值 " .. key .. " - " .. value)
  end,
  __call = function(table, args)
    -- 函数调用操作 1 - 1
    for k, v in pairs(args) do
      print("函数调用操作 " .. k .. " - " .. v)
    end
  end
}
setmetatable(tab, metatable)
local val3 = tab["key3"]
tab["key4"] = "val4"
tab({"1"}) -- 类似于函数调用
```

这 3 个元事件在与 iOS 交互中，分别扮演不同的角色。

`__index` 事件可以充当查询原生方法的作用 `__newindex` 事件可以充当 Hook 原生方法的作用 `__call` 事件可以方便调用原生定义的 block

全局环境

在 Lua 中，有一个全局的 `_G` 表，里面存放的是系统函数，比如 `pairs`，`print` 等函数。

然而在 Lua 5.2 开始，引入了一个全局环境的概念叫做 `_ENV`，默认的全局环境是 `_G` 表。

在代码里使用 `print` 函数时，为什么可以找到这个函数，就是因为全局环境关系是这样的 `_ENV=_G`，首先 Lua 会查找当前的全局环境 `_ENV` 对应的表，从对应的表里查找 `print` 函数，因为这里对应的表是 `_G` 表，所以是可以找到并调用的。

当然我们也可以修改全局环境，如下代码就是修改了全局环境。

```
-- 这段代码是会打印失败的，因为虽然修改了全局环境，但是全局环境是一张空表，什么都没有，那么就会找不到 print 函数而打印失败
local test_env = {}
_ENV = test_env
print("打印失败")

```

```
-- 这里指所以成功，因为全局环境虽然是一张空表，但这种空表有一个元表，而元表又添加了 __index 索引取值事件，当在空表里找不到 print 时，就会从
-- 这里指所以成功，因为全局环境虽然是一张空表，但这种空表有一个元表，而元表又添加了 __index 索引取值事件，当在空表里找不到 print 时，就会从  对应的 _G 表里查找。
local test_env = {}
setmetatable(test_env, {__index = _G})
_ENV = test_env
print("打印成功")
```

之所以要介绍全局环境，因为不管是给全局变量赋值还是取值都会从当前的全局环境里查找。这点对于定义函数来说也同样适用。

这里在修改了全局环境后，定义了一个函数，这个函数将会定义在当前的全局环境对应的 `test_env` 表里。

```
local test_env = {}
setmetatable(test_env, {__index = _G})
_ENV = test_env
function a()
  print("print a")
end
a()
_ENV.a() -- 等价于 a()
test_env.a() -- 等价于 a()
```

有了这个特性后，就能非常方便的定义一个全局 Lua 函数去 Hook 原生的方法了。

冒号语法糖

在 `OC` 里，我们都知道调用一个方法，最后都会走到 `objc_msgSend`，这个函数的第一个入参就是调用对象本身。

```
[a hello:@"who"];
objc_msgSend(a, hello, @"who");
```

那在 Lua 里冒号语法糖，有点类似于 `OC` 里方法调用。

```lua
local table = {
  who = "who",
  hello = function(table, key)
    print("冒号语法糖 "..table[key] .. " - "..key)
  end
}
table.hello(table, "who")
table:hello("who") -- 等价于 table.hello(table, "who")
```

在实际 Hook 的 Lua 函数里去调用原生方法的时候，我们是需要使用冒号语法糖来获取当前是谁在调用这个方法的，这样才能做到让 Lua 调用原生。比如如下 Lua 代码就可以通过冒号语法糖优雅的调用原生方法。

```lua
UIColor:greenColor()
等价于
UIColor.greenColor(UIColor)
```

Lua 与 C 通信

Lua 是一个小巧的脚本语言，且非常容易和 C/C++ 交互，与大多数脚本语言一样，Lua 也需要开启虚拟机去解释执行 Lua 脚本。Lua 想要调用 C 能力，必须先在 C 成加载自定义 Lua 库，然后就可以在 Lua 脚本里就可以调用该库的功能。

声明一个自定义 Lua 库

创建一个自定义库。这个库里提供了一个 C 函数，用于接收两个 double，返回这两个 double 的积和和。

```lua
/// 提供给 lua 调用的 l_f c 函数
/// 输入两个double，返回两个double
static int l_f(lua_State *L)
{
    double arg1 = luaL_checknumber(L, 1);// 获取栈中索引1位置的值，如果获取失败内部会把错误信息压入栈顶
```

```
    double arg2 = luaL_checknumber(L, 2);// 获取栈中索引2位置的值，如果获取失败内部会把错误信息压入栈顶
    double r1 = arg1 * arg2;
    double r2 = arg1 + arg2;

    lua_pushnumber(L, r1);// 把结果压入栈顶
    lua_pushnumber(L, r2);// 把结果压入栈顶

    return 2;// 返回结果的数量
}

static const struct luaL_Reg mylib[] = {
    {"c_f", l_f},
    {NULL, NULL}
};

LUAMOD_API int luaopen_mylib(lua_State *L)
{
    luaL_newlib(L, mylib);// 1、创建空table并压入栈顶 2、把所有函数都设置到栈顶的table里
    return 1;
}
```

声明好后，需要加载该自定义 Lua 库。

```
int main() {
    // 创建一个 Lua 虚拟机
    lua_State *L = luaL_newstate();
    // 加载 Lua 标准库
    luaL_openlibs(L);
    // 加载 Lua 自定义模块
    luaL_requiref(L, "mylib.util", luaopen_mylib, 0);
    // 加载 lua 脚本文件
    loadLuaFile(L, "myfunction.lua");
}
```

加载后就可以在 Lua 脚本里使用了。

```
-- Lua 函数调用 C 函数
local util = require("mylib.util")
```

```
local r1,r2 = util.c_f(2.0, 5.0);
print("调用 C 函数 util.c_f", r1, r2)
```

栈

Lua 使用一个 虚拟栈 来和 C 互传值。 栈上的的每个元素都是一个 Lua 值 （nil，数字，字符串，等等）。

无论何时 Lua 调用 C，被调用的函数都得到一个新的栈， 这个栈独立于 C 函数本身的栈，也独立于之前的 Lua 栈。 它里面包含了 Lua 传递给 C 函数的所有参数， 而 C 函数则把要返回的结果放入这个栈以返回给调用者。

栈结构

第一个入参是在栈顶，后面的参数依次压栈，函数调用结果会放到栈顶。栈有两种索引方式，可以从下往上索引，也可以从上往下索引。

栈索引	栈顶	值描述	栈索引
4	7	结果2	-1
3	10	结果1	-2
2	5	参数2	-3
1	2	参数1	-4
从下往上	栈底		从上往下

C 调用 Lua 函数

C 调用 Lua 函数时，也需要用到栈结构。

比如有如下的 Lua 函数，一个入参，一个返回值：

```
function hello(who)
    print("hello "..who)
    return 1
end
```

那么 C 在调用 Lua 函数时，需要用到 `lua_pcall` 函数。

> L：是虚拟机实例
>
> argumentCount：参数个数
>
> returnCount：结果个数

```
// 压入 hello lua 函数
lua_getglobal(L, "hello");
// 压入参数
lua_pushstring(L, "siri");
// 调用 hello
lua_pcall(L, argumentCount, returnCount, 0);
```

那么在调用这个 Lua 函数时，栈结构如下：

栈索引	栈顶	值描述	栈索引
3	1	结果1	-1
2	siri	参数1	-2
1	function	hello 函数	-3
从下往上	栈底		从上往下

userdata

userdata 类型允许将 C 中的数据保存在 Lua 变量中。 用户数据类型的值是一个内存块， 可以用于 C 层构建自定义数据结构，并让 Lua 变量可以访问这些自定义数据。

常用的用法是在 C 层里定义一个结构体，然后使用 userdata 包装下这个结构，就可以返回给 Lua 层里使用了。

在 iOS 里，对于每个类和每个实例，都需要早 C 层构建一个与之对应的 userdata。

比如下面这个 Lua 调用，实际上是需要先在 C 层构建一个 userdata 类型，这个 userdata 里需要保存类名。另外还需要在 C 层给这个构建的 userdata 设置一个元表，因为当调用 `greenColor` 方法时，需要在 C 层查找要调用的方法名。

```
UIColor:greenColor()
```

在 Lua 里还存在另一种 userdata，叫做 lightuserdata（轻量userdata），他的用途只适合返回一个指针地址给到 Lua 里。

怎么构建一个 userdata

需要现在 C 层定义一个用户数据结构体。

```
typedef struct Userdata {
    __unsafe_unretained id instance;// 如果是类用户数据，代表的是 class；如果是实例用户数据，代表的是 实例
    bool isClass;
} Userdata;
```

然后需要创建用户数据实例，最后返回 userdata 给到 Lua 层。Lua 层拿到 userdata 就可以做方法调用，变量取值和赋值。

```
// 创建一个 userdata 并压栈
size_t nbytes = sizeof(Userdata);
Userdata *userData = (Userdata *)lua_newuserdata(L, nbytes);
userData->instance = klass;
userData->isClass = true;

// 给 class userdata 设置 元表，这个元表是预先定义好的
luaL_getmetatable(L, "CLASS_USER_DATA_META_TABLE");
lua_setmetatable(L, -2);
```

Runtime 基础原理

Lua 能做到通过 Lua 调用和改写 OC 方法最根本的原因是 Objective-C 是动态语言，OC 上所有方法的调用/类的生成都通过 Objective-C Runtime 在运行时进行，我们可以通过类名/方法名反射得到相应的类和方法：

```
Class class = NSClassFromString("UIViewController");
id viewController = [[class alloc] init];
SEL selector = NSSelectorFromString("viewDidLoad");
[viewController performSelector:selector];
```

也可以替换某个类的方法为新的实现：

```
static void newViewDidLoad(id slf, SEL sel) {}
class_replaceMethod(class, selector, newViewDidLoad, @"");
```

还可以新注册一个类，为类添加方法：

```
Class cls = objc_allocateClassPair(superCls, "JPObject", 0);
class_addMethod(cls, selector, implement, typedesc);
objc_registerClassPair(cls);
```

理论上你可以在运行时通过类名/方法名调用到任何 OC 方法，替换任何类的实现以及新增任意类。

Lua 与 iOS 通信

Hook 原生方法

先看一个简单的例子：

```
wpfix_class({"SubObject"}, function(_ENV)
    function hello_(self,who)
        print("【LUA】hello "..who..."!")
    end
end)
```

`wpfix_class` 源码实现：

```lua
local wpfix_classN = require("wpfix.class")
-- 定义一个 类，用于替换原生 类，或者添加一个 新类
function wpfix_class(options, instance_function, class_function)
    -- 类名，字符串
    local class_name = options[1]
    -- 父类名，字符串
    local super_class_name = options[2]

    -- 基于要 hook 的类名，创建 class user data
    local class_userdata = wpfix_classN(class_name, super_class_name, options.protocols)

    -- class 作为 key，这样在函数里就可以使用 class 关键字了，这里 class 不是变量，只是单纯的 key
    -- realClsName 是真实类名
    local scope = {
        class = class_userdata,
        __properties = {
            realClsName = class_name
        }
    }

    -- 设置 scope 元表
    setmetatable(scope, {
        -- 当有新的 key 存在时，比如 lua 文件中新定义的函数，都会触发 __newindex
        __newindex = function(scope, key, value)
            -- print("【LUA】", class_name, "==== __newindex", scope, key, value)
            class_userdata[key] = value
        end,

        -- 当获取 key 不存在 scope 时，就会触发 __index
        __index = function(scope, key)
            -- print("【LUA】", class_name, "==== __index", scope, key)
            -- 当检索的是 其他原生类 时，比如 UIColor，那么就会先去创建 UIColor 的 class user data
            -- 当检索的是 当前类原生 静态方法时，就去 class_userdata 的元方法里检索 key 对应的静态方法
            -- 如果以上都不是，那就需要从 全局 _G 表中找，比如 要找 print lua 函数
            return wpfix_classN.findUserData(key) or class_userdata[key] or _G[key]
        end,
    })

    -- 安装需要替换或者新增的实例方法
    if (instance_function) then
```

```
        instance_function(scope)
    end

    -- 安装需要替换或者新增的类方法
    if (class_function) then
        class_function(class_scope)
    end
end
```

`wpfix_class` 函数解析：

该函数的目的有两个。

- 创建一个类名是 SubObject 的 ClassUserData，同时还要创建该 ClassUserData 的元表和关联表。ClassUserData 元表里有 `__index` 和 `__newindex` 事件。
  a. 原生方法的获取，实际上需要一个约定的，因为在 Lua 里不能用冒号作为函数名称的一部分，那就需要另一个符号来表示参数了，这里使用的是下划线 `_` 来表示参数，所以 `hello_` 对应的是原生方法 `hello:`
  b. ClassUserData 元表 `__newindex` 事件: 该事件用于把原生方法的实现替换成 `_objc_msgForward`，并且把 消息转发第三步方法 `forwardInvocation:` 的实现替换成自定的实现 `static void __WPFIX_ARE_BEING_CALLED__(__unsafe_unretained NSObject *self, SEL selector, NSInvocation *invocation)`，最后需要在关联表里记录下 Hook 后的 Lua 函数
  c. ClassUserData 元表 `__index` 事件: 该事件用于查找原生类的类方法
  d. ClassUserData 关联表就是一个普通的表，用于存放被 Hook 的原生方法对应的 Lua 函数实现，该表的 <Key：Value> 分别是原生方法名和 Lua 函数体。
- 修改 _ENV 全局环境。
  a. 最后两个参数都是函数， `instance_function` 用于定义 Lua 实例函数去 Hook 原生实例方法，`class_function` 用于定义 Lua 类函数去 Hook 原生类方法。
  b. 当调用 `instance_function` 函数的时候，实际上已经修改了函数体内的 _ENV 全局环境了。`instance_function(scope)` 类似于 `instance_function(_ENV=scope)`，所以当在里面定义 Lua 函数时，就会触发 scope 表的元表 `__newindex` 事件，而 `__newindex` 事件对应的函数里，又会调用 `class_userdata[key] = value`，而这里又会触发 `class_userdata` 元表的 `__newindex` 事件所对应的原生 C 函数实现。

综上所述，我们可以知道 `hello` Lua 函数最后是存储在了名字为 SubObject 的 ClassUserData 对应的关联表里。

原生方法实际调用时

还是上面那个例子，它对应的原生代码实现如下：

```
@interface SubObject : BaseObject
- (void)hello:(NSString *)who;
@end
@implementation SubObject

- (void)hello:(NSString *)who {
    NSLog(@"hello %@", who);
}
@end
```

那么在实际调用时，会发生什么呢?

```
SubObject *obj = [SubObject new];
[obj hello:@"siri"];
```

由于 `hello` 方法已经被替换成了 `_objc_msgForward` 函数，那么在调用的时候，就会走到 `forwardInvocation:` 的实现里，而由于 `forwardInvocation:` 本身也被 Hook 了，那么实际上是会走到 `static void __WPFIX_ARE_BEING_CALLED__(__unsafe_unretained NSObject *self, SEL selector, NSInvocation *invocation)` 这个函数的实现里了。

```
/// 实例方法调用时，self 是 实例。类方法调用时，self 是 类
static void __WPFIX_ARE_BEING_CALLED__(__unsafe_unretained NSObject *self, SEL selector, NSInvocation *invocation)
{
    lua_State *L = [WPFix currentLuaState];
    if (pushLuaFunction(L, self, invocation.selector)) {// 从类继承链里找 lua 函数，如果能找到 lua 函数，就调用 lua 函数
        int nresults = [WPFixHelper callLuaFunction:L assignSlf:self selector:invocation.selector invocation:invocation];
        if (nresults > 0) {
            NSMethodSignature *signature = [self methodSignatureForSelector:invocation.selector];
            void *pReturnValue = [WPFixConverter toOCObject:L typeDescription:[signature methodReturnType] index:-1];
```

```
            if (pReturnValue != NULL) {
                [invocation setReturnValue:pReturnValue];
                free(pReturnValue);
            }
        }
    } else {// 当 lua 函数里，调用没有 hook 的父类方法时，就会找不到 lua 函数，此时就需要调用原始转发方法
        // 调用原始消息转发方法
        SEL origForwardSelector = NSSelectorFromString(WPFIX_ORIGIN_FORWARD_INVOCATION_SELECTOR_NAME);
        ((void(*)(id, SEL, id))objc_msgSend)(self, origForwardSelector, invocation);
    }
}
```

这个 C 函数作用，就是拦截了原生方法的调用，然后通过原生方法的名字，去 ClassUserData 对应的关联表里找到对应的 Lua 函数实现，再根据原生方法的签名获取参数个数和参数类型，然后把 Lua 函数压栈，把 self 压栈，和实际参数压栈就可以通过 `pcall` Lua 自带的 C 标准库函数来完成 Lua 函数的调用。

self 是怎么来的

因为已经拦截了原生方法的调用了，那其实是有原生对象的实例的，`static void __WPFIX_ARE_BEING_CALLED__(__unsafe_unretained NSObject *self, SEL selector, NSInvocation *invocation)` C 函数里第一个参数就是原生对象的实例，但是 Lua 里是不能识别这个对象的，那么就需要一种方式来让 Lua 去识别，这里采用的其实还是 userdata 类型，一个可以自定义数据结构且能被 Lua 访问。

```
// 创建 实例 userdata
size_t nbytes = sizeof(Userdata);
Userdata *instance = (Userdata *)lua_newuserdata(L, nbytes);
instance->instance = self;// 这个 self 就是原生对象的实例
instance->isClass = false;
```

有了 userdata，那么就是在调用 Lua 函数的时候，先压栈 Lua 函数，在压栈这个 userdata，最后压栈实际的参数，就可以让 Lua 函数获取到 self 了。所以 Lua 函数里的第一个参数就是 self，它其实就是一个 userdata。

```
wpfix_class({"SubObject"}, function(_ENV)
    function hello_(self,who)
        -- slef 就是一个 userdata
        print("【LUA】hello "..who.."!")
    end
end)
```

super 实现

OC 里调用父类的方法其实就是把当前对象当成 self 传入父类方法，那么我们只要模拟它这个过程就行了。

```
wpfix_class({"SubObject"}, function(_ENV)

    function resizeA(self)
        -- self 就是一个 实例 UserData，上面也讲过
        print("【LUA】SubObject1 self ", self)
        self:super(_ENV):resizeA()
        print("【LUA】SubObject2 ")
    end
end)
```

上面的 Lua 代码里 `self:super(_ENV)` 这句话会触发实例 UserData 的 `static int LUserData__index(lua_State *L)` C 函数。

之前也讲过 UserData 也是可以绑定元表的，只不过这里的元表是原生 C 代码实现的一个元表，元表里有一个 `__index` 事件，这个事件绑定了 `LUserData__index` 函数，所以自然就可以调用到这里来。

```
static int LUserData__index(lua_State *L)
{
  // 第一个参数
  Userdata *instanceUserData = lua_touserdata(L, 1);// 转换栈顶数据到一个 实例 userdata 指针
  // 第二个参数
  const char* func = lua_tostring(L, 2);

  // 检测到 super 关键字
```

```
  if ([selector isEqualToString:WPFIX_SUPER_KEYWORD]) {// 如果是父类调用，就设置临时设置 super 为 true，待实际调用完成时，需要还原成 false
        instanceUserData->isCallSuper = true;
        // 压入一个 super 闭包，目的是获取 _ENV
        lua_pushcclosure(L, super_keyword_closure, 1);
        return 1;
   }
   ...
}

static int super_keyword_closure(lua_State *L)
{
     // 复制 实例 user data 并压栈，其实就是返回自己
     lua_pushvalue(L, 1);
     return 1;
}
```

当发现是 super 关键字时，这里就先记一个标记，表示下次调用的是父类方法。然后返回一个闭包，闭包其实就是一个函数的意思，只不过这个函数捕获了一个变量，这里捕获的变量就是第二个入参。而该 C 函数的实现就只是返回了实例 UserData 本身。

最后函数返回到了 Lua 里，而在 Lua 里使用 `()` 作为结尾时，其实就是在调用函数，所以这里调用的是原生的 `super_keyword_closure` C 函数，并返回实例 UserData。

最后通过 UserData 本身去调用其他目标方法。

在实际调用的时候，就会先判断实例 UserData 是否有打上 super 标记，有的话，就获取父类的方法实现，并为当前类添加一个新方法指向父类的方法实现。那么在调用这个类新方法相当于是在调用 super 方法了。

```
static int invoke_closure(lua_State *L) {
  if (instance->isCallSuper) {// 如果是调用父类方法，就需要为当前类添加一个父类方法实现
        NSString *superSelectorName = [NSString stringWithFormat:@"%@%@", WPFIX_SUPER_PREFIX, selectorName];
        SEL superSelector = NSSelectorFromString(superSelectorName);
        Class klass = object_getClass(instance->instance);
        Class superClass = class_getSuperclass(klass);

        // 获取父类的方法实现
        Method superMethod = class_getInstanceMethod(superClass, NSSelectorFromString(selectorName));
        IMP superMethodImp = method_getImplementation(superMethod);
        char *typeDescription = (char *)method_getTypeEncoding(superMethod);
```

```
        // 如果是调用父类方法，就为当前类添加一个父类的实现
        class_addMethod(klass, superSelector, superMethodImp, typeDescription);
    }
}
```

新增方法

当我们 Hook 一个存在原生方法时，实际调用的是原生方法对应的一个 Lua 函数体，光有函数体还不行，我们还需要知道入参有几个，入参的类型是什么，返回的类型又是什么，这些问题其实是由原生方法的方法签名来告诉我们的。

获取原生方法签名的方式也简单。

```
// 获取方法签名
NSMethodSignature *signature = [object methodSignatureForSelector:selector];
// 获取入参个数，这里的入参个数前两个是 self 和 _cmd
int nargs = (int)[signature numberOfArguments];
// 获取入参类型
for (NSUInteger i = 2; i < [signature numberOfArguments]; i++) { // start at 2 because to skip the automatic self and _cmd arugments
    const char *typeDescription = [signature getArgumentTypeAtIndex:i];
}
// 获取返回个数
int nresults = [signature methodReturnLength];
// 获取返回类型
const char *returnType = [signature methodReturnType];
```

那当我们给一个类添加一个不存在方法时，那谁又来告诉我们方法签名呢?

所以这里就需要我们自己来定义方法签名了，我们可以通过使用 `wpfix_protocol` 来定义一个新的协议。比如这里是定义了一个协议名为 `SubObjectProtocol` 的新协议，协议里只有一个方法 `convertString:`,方法签名是 `NSString*,NSString*`。在方法签名里，前面的字符表示返回类型，后面的字符都表示入参类型，所以这里定义是一个返回值是字符串，入参也是字符串的 `convertString:` 方法。

```
wpfix_protocol("SubObjectProtocol", {
    convertString_ = "NSString*,NSString*",
})
```

而内部实现其实是通过下面三个 runtime API 注册了一个新的协议

> objc_allocateProtocol: 创建一个协议 protocol_addMethodDescription: 给协议添加方法签名 objc_registerProtocol: 注册新协议

注册好了新协议，那类怎么知道呢?

这是就需要在 Lua Hook Class 的时候指定出要实现的协议名称了。 而内部的实现是通过 runtime 方法 `class_addProtocol(klass, protocol);` 来实现的。这样在调用新增方法的时候，就可以跟存在的原生方法一样去获取方法签名了。

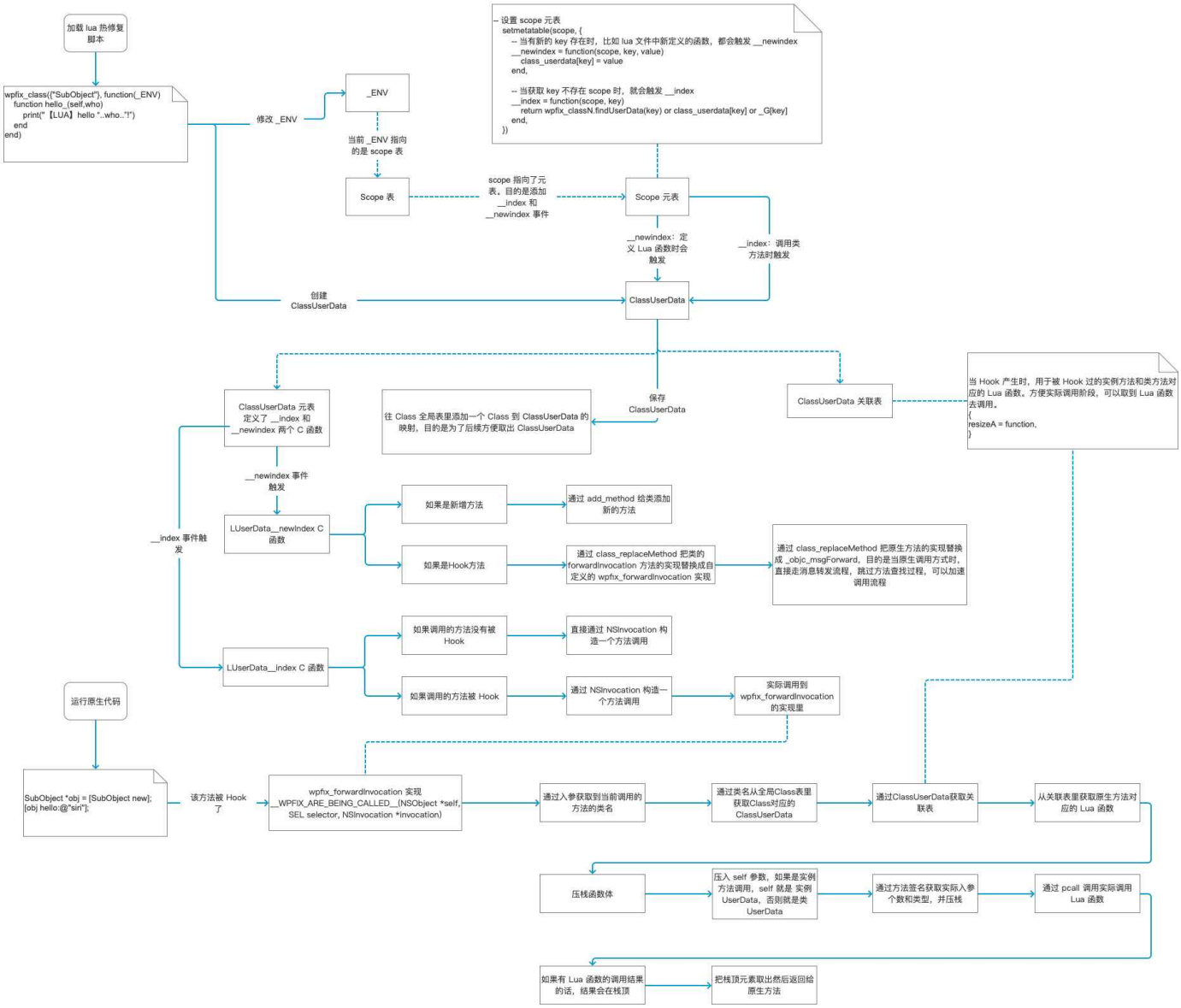
```
wpfix_class({"SubObject"}, protocols={"SubObjectProtocol"}, function(_ENV)

    function hello_(self,who)
        print("【LUA】hello "..who.."!")

        local converted = self:convertString_("anything")
        print("【LUA】converted ", converted)
    end

    function convertString_(self,str)
        return "converted "..str
    end
end)
```

交互流程图



结构体支持

结构体就是内存中是的一段连续的数据。

```
1 /* Points. */
2 struct
3 CGPoint {
4     CGFloat x;
5     CGFloat y;
6 };
7 typedef struct CGPoint CGPoint;
8
9 /* Sizes. */
```

```
10
11 struct CGSize {
12     CGFloat width;
13     CGFloat height;
14 };
15 typedef struct CGSize CGSize;
16
17 struct CGRect {
18     CGPoint origin;
19     CGSize size;
20 };
21 typedef struct CGRect CGRect;
```

比如有下面这样的一个 CGRect 结构体，他在内存的结构是连续的4个 double 类型。

```
1 CGRect xrect;
2 xrect.origin.x = 3.0;
3 xrect.origin.y = 4.0;
4 xrect.size.width = 5.0;
5 xrect.size.height = 6.0;
```

x	y	width	height
8bytes	8bytes	8bytes	8bytes
3	4	5	6

在 Lua 中使用结构体

要想使用结构体，那就需要知道结构都有哪些类型组成。另外还需要给每个类型定义一个别名，来方便存和取。下面的代码就是在 Lua 中注册了一个结构体描述。

```
1 wpfix_struct({name = "CGRect", types = "CGFloat,CGFloat,CGFloat,CGFloat", keys
  = "x,y,width,height"})
```

注册的目的有两个： 1、把结构体名字，类型和别名缓存起来，放在一个结构体描述字典里。 2、生成一个跟结构体名字同名的全局 Lua 函数。

结构体名字	类型	别名
CGRect	CGFloat,CGFloat,CGFloat,CGFloat	x,y,width,height

所以在 Lua 中创建结构体可以通过这个全局 Lua 函数来构建一个结构出来。

```
wpfix_struct({name = "CGRect", types = "CGFloat,CGFloat,CGFloat,CGFloat", keys = "x,y,width,height"})
-- CGRect 就是一个 Lua 函数
CGRect({x=3.0, y=4.0, width=5.0, height=6.0})
```

调用后就会走到原生的 C 函数实现里。大概的步骤如下:

1. 先获取结构体名字，然后从结构体描述的缓存字典里，获取类型和别名。
2. 根据类型创建一块连续内存，遍历类型字符串往结构体里填充每个类型的数据。
3. 最后构建一个结构体 UserData，返回给 Lua。

最后原生代码就可以通过 `StructUserdata->data` 来访问 Lua 里创建的结构体数据。如果是原生代码创建的结构体，那也是需要把结构封装成结构体 UserData 的，这样就可以打通结构体在 Lua 和原生间的使用了。

```
static int create_userdata_closure(lua_State *L) {
    /// 获取结构体名字
    const char *name = lua_tostring(L, lua_upvalueindex(1));

    /// 从结构体描述字典里取出结构体的描述
    NSDictionary *structDefine = registeredStructs()[[NSString stringWithUTF8String:name]];

    /// 获取结构体类型和别名
    NSString *realTypeDescription = structDefine[@"types"];
    NSString *keys = structDefine[@"keys"];

    /// 根据类型创建一块连续内存
    const char *typeDescription = realTypeDescription.UTF8String;
    size_t size = [WPFixConverter sizeOfStructTypes:typeDescription];
    void *structData = malloc(size);

    /// 遍历类型去偏移结构体指针，然后把数据填充到偏移后内存里
    for (int i = 0; i < typeDescription.count; i ++) {
```

```
        size_t size = sizeof(typeDescription[i]);  //types[i] 是 float double int 等类型
        void *val = malloc(size);
        memcpy(val, structData + position, size);
        position += size;
    }

    /// 根据结构体指针创建一个 struct user data
    create_userdata(L, name, typeDescription, structData);
    free(structData);
    return 1;
}

/// 自定义用户数据，结构体会用到
typedef struct StructUserdata {
    void *data;// 实际数据，里面存储的数据根据 typeDescription 来决定
    size_t size;// 数据总大小
    char *name;// lua定义的结构体名字，比如 "CGSize"
    char *types;// lua定义的结构体签名，比如 "dd"
} StructUserdata;
```

Block 支持

Block 底层就是一个结构体，里面有个 invoke 函数指针，指向就是 block 函数体。在实际调用的时候，就是使用的这个 block 函数体来完成调用的。

```
struct Block {
    void *isa; // initialized to &_NSConcreteStackBlock or &_NSConcreteGlobalBlock
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct BlockDescriptor *descriptor;
    // imported variables
    void *wrapper;
};

struct BlockDescriptor {
    struct {
        unsigned long int reserved;
        unsigned long int size;
    };
```

```
    struct {
        // requires BLOCK_HAS_COPY_DISPOSE
        void (*copy)(void *dst, const void *src);
        void (*dispose)(const void *);
    };
    struct {
        // requires BLOCK_HAS_SIGNATURE
        const char *signature;
    };
};
```

这个结构体里，除了函数指针，还有一个重要的字段就是 `signature`，通过它我们可以获取这个 block 的签名。block 的签名跟方法签名类似，都是返回类型在前，参数类型在后面。

```
// 签名：i@?i，其中 @? 代表的是 block 本身
int (^block)(int) = ^(int){

};

// 这里注意下，block 的签名与方法的签名是有一些差异的。
// 签名：i@:i，其中 @ 代表的是 对象 本身，: 代表选择器
- (int)convertedWithNum:(int)num;
```

通过上面我们了解到，block 最主要的是函数指针和签名，那在 Lua 里，我们可以通过下面的代码来定义一个 block。

```
- (int(^)(int))convertedWithNumBlock
{
    return nil;
}
```

```
wpfix_class({"BlockDefineTest"},
function(_ENV)
    function convertedWithNumBlock(self)
        return wpfix_block(function(i)
                               return 1
```

```
                        end, "int,int")
    end
end)
```

`wpfix_block` 函数有两个入参，第一个是要执行的 Lua 函数，第二个是 block 的签名，这里的签名是 `int,int`，表示的是返回类型是 `int`，且只有一个入参类型是 `int`。

而在 C 函数的实现里主要是做了如下几件事情：

1. 获取类型签名，创建一个 block 对应的实例 UserData。
2. 把 Lua 层定义的要执行的 Lua 函数保存到新创建的实例 UserData 里的关联表里。
3. 最后返回一个 block 函数指针转换成一个轻量的 UserData，最后返回这个轻量的 UserData 给到 Lua。

```
static int LF_define_block(lua_State *L)
{
    NSString *typeEncoding = @"void,void";
    // 获取类型签名
    const char* type_encoding = lua_tostring(L, 2);
    if (type_encoding) {
        typeEncoding = [NSString stringWithUTF8String:type_encoding];
    }

    // 创建 实例 userdata
    ...

    // 获取 实例 user data 的关联表，并压栈
    lua_getuservalue(L, -1);
    // 压入key
    lua_pushstring(L, "f");
    // 把函数压栈
    lua_pushvalue(L, funcIndex);
    // 把函数保存到关联表里，相当于 associated_table["f"] = lua 函数
    lua_rawset(L, -3);
    // pop 关联表
    lua_pop(L, 1);

    void *ptr = [block blockPtr];
    // 把 block 指针压栈
    lua_pushlightuserdata(L, ptr);
    return 1;
}
```

block 指针怎么来的

首先要介绍 FFI, FFI 全名 Foreign Function Interface ，中文名 外部函数接口。简单的一句话：FFI 是 C 函数的运行时，可以动态替换和创建一个 C 函数。详细的信息大家感兴趣的话可以搜索下相关资料。

创建一个 block 指针，有下面几个步骤：

1. 通过参数个数，返回值类型，参数类型，去创建一个函数调用模板。
2. 把函数调用模板与 blockIMP 函数指针相关联。
3. 构建一个原生的 block 对象。而上面的 blockIMP 就是 block 的函数调用指针。当 block 实际执行的时候，是会使用 blockIMP 来处理调用逻辑。

```
- (void *)blockPtr {
    NSString *typeEncoding = self.typeEncoding;
    NSMethodSignature *signature = [NSMethodSignature signatureWithObjCTypes:typeEncoding.UTF8String];
    if (typeEncoding.length <= 0) {
        return nil;
    }
    // 第一个参数是自身block的参数
    unsigned int argCount = (unsigned int)signature.numberOfArguments;
    void *imp = NULL;
    _cifPtr = malloc(sizeof(ffi_cif));  //不可以free
    _closure = ffi_closure_alloc(sizeof(ffi_closure), (void **)&imp);
    ffi_type *returnType = (ffi_type *)typeEncodingToffiType(signature.methodReturnType);
    _args = malloc(sizeof(ffi_type *) * argCount);
    _args[0] = &ffi_type_pointer;

    /// 生成 ffi 的参数类型
    for (int i = 1; i < argCount; i++) {
        _args[i] = (ffi_type *)typeEncodingToffiType([signature getArgumentTypeAtIndex:i]);
    }

    /// 需要知道参数个数，返回值类型，参数类型，去创建函数调用模板
    if (ffi_prep_cif(_cifPtr, FFI_DEFAULT_ABI, argCount, returnType, _args) == FFI_OK) {
        /// 把函数调用模板与 blockIMP 函数指针相关联
        if (ffi_prep_closure_loc(_closure, _cifPtr, blockIMP, (__bridge void *)self, imp) != FFI_OK) {
```

```
            NSAssert(NO, @"block 生成失败");
        }
    }

    struct KitBlockDescriptor descriptor
        = { 0, sizeof(struct KitBlock), (void (*)(void *dst, const void
*src))copy_helper, (void (*)(const void *src))dispose_helper, nil };

    _descriptor = malloc(sizeof(struct KitBlockDescriptor));
    memcpy(_descriptor, &descriptor, sizeof(struct KitBlockDescriptor));

    struct KitBlock newBlock
        = { &_NSConcreteStackBlock, (Kit_BLOCK_HAS_COPY_DISPOSE |
Kit_BLOCK_HAS_SIGNATURE), 0, imp, _descriptor, (__bridge void *)self };

    _blockPtr = Block_copy(&newBlock);
    CFRelease(&descriptor);
    CFRelease(&newBlock);
    return _blockPtr;
}
```

当 Lua 层把 block 对应的轻量 UserData 返回给原生时，原生可以通过该轻量 UserData 获取到 block 对象，在 block 对象实际调用时，就会触发 blockIMP C 函数的调用。

blockIMP 内部实现跟 Hook 原生方法的实现类似，首先要获取实例 UserData，然后通过实例 UserData 对应的关联表里去获取保存的 Lua 函数，最后调用 Lua 函数。

```
static void blockIMP(ffi_cif *cif, void *ret, void **args, void *userdata) {
    // 伪代码
    // 获取 实例 userdata
    // 获取 实例 userdata 的关联表
    // 从关联表中获取要执行的 Lua 函数
    // Lua 函数压栈
    // 参数压栈
    // Lua 函数调用
}
```

Lua 里调用原生的 block

而如果是原生把 block 传入给 Lua 时，此时 block 会被包装成实例 UserData 传入到 Lua 里。

比如：

```
/// lua 脚本 hook 这个方法
- (void)blockOneArg:(int(^)(int i))block {
     self.index = block(11);
}
```

```
-- hook 带有 oc block 参数的实例方法
function blockOneArg_(self,block)
    -- 调用原生 block
    self:setIndex_(block(12))
end
```

我们之前讲过元表有一个 `__call` 事件，它可以让对象像调用函数一样使用，所以这里只要给实例 UserData 设置一个元表，并在元表里添加 `__call` 事件就可以让实例 UserData 使用 `()` 来调用原生的 block。下面的 C 函数就是该事件对应的实现。

函数里主要有如下几个步骤： 1.获取实例 UserData，并判断包装的对象是不是一个 block 对象。 2.如果是 block，就要解析出 block 的签名信息。这里实际上是通过 BlockDescriptor 结构体里的 `signature` 获取到的。 3.创建 NSInvocation 对象，设置参数并完成调用。

```
/// 用于 lua 实际调用原生 block
static int LUserData__call(lua_State *L)
{
    InstanceUserdata *instance = lua_touserdata(L, 1);
    id object = instance->instance;
    if ([Helper isBlock:object]) {
        return [Helper callBlock:L];
    }
    return 0;
}
/// 通过 lua 调用 oc block
+ (int)callBlock:(lua_State *)L {
    InstanceUserdata* instance = lua_touserdata(L, 1);
    id block = instance->instance;
    /// 获取 block 签名，比如 i12@?0i8
```

```
16      BlockDescription *blockDescription = [[BlockDescription alloc]
    initWithBlock:block];
17      NSMethodSignature *signature = blockDescription.blockSignature;
18
19      int nresults = [signature methodReturnLength] ? 1 : 0;
20
21      NSInvocation *invocation = [NSInvocation
    invocationWithMethodSignature:signature];
22      [invocation setTarget:block];
23
24      for (unsigned long i = [signature numberOfArguments] - 1; i >= 1; i--) {
25          const char *typeDescription = [signature getArgumentTypeAtIndex:i];
26          void *value = [WPFixConverter toOCObject:L
    typeDescription:typeDescription index:-1];
27          [invocation setArgument:value atIndex:i];
28          if (value != NULL) {
29              free(value);
30          }
31      }
32
33      /// 调用 block 实现
34      [invocation invoke];
35
36      if (nresults > 0) {
37          const char *typeDescription = [signature methodReturnType];
38          NSUInteger size = 0;
39          NSGetSizeAndAlignment(typeDescription, &size, NULL);
40          void *buffer = malloc(size);
41          [invocation getReturnValue:buffer];
42          [WPFixConverter toLuaObjectWithBuffer:L
    typeDescription:typeDescription buffer:buffer];
43          free(buffer);
44      }
45
46      return nresults;
47  }
```

C 函数支持

有时候我们是需要在 Lua 中调用 GCD 函数，而又不可能每个都实现，所以主要还是实现常用的 GCD 函数就可以了。

那么怎么支持在 Lua 中支持 C 函数的，这里很自然的就会想到在 C 层要创建全局的 Lua 函数。比如想要在 Lua 中调用 `dispatch_after` 函数来做延后时间的操作。

```
dispatch_after()
```

这样的 GCD C 函数也有好几个，一个一个在 C 层手动创建是不可能的，这辈子都不可能。。。

此时我们就需要借助一个名叫 `tolua++` 的工具来帮助我们做这个事情了。

`tolua++` 是一个可以把包含 C 函数描述的 `pkg` 文件 转换成可以在 Lua 中使用的 C 函数源码的工具。

```
// dispatch_lua.pkg
$#include <dispatch/dispatch.h>

#define DISPATCH_TIME_NOW 0
#define NSEC_PER_SEC        1000000000

void* dispatch_get_main_queue ( void );
void dispatch_after ( unsigned long long when, void* queue, void* block );
unsigned long long dispatch_time ( unsigned long long when, long long delta );
```

比如我们有一个上面的 `dispatch_lua.pkg` C 函数描述文件。

然后通过下面的命令行工具，就可以导出一个 `dispatch_lua.m` 文件。

tolua++ -o dispatch_lua.m dispatch_lua.pkg

```
// dispatch_lua.m
TOLUA_API int tolua_dispatch_lua_open (lua_State* tolua_S)
{
 tolua_open(tolua_S);
 tolua_reg_types(tolua_S);
 tolua_module(tolua_S,NULL,0);
 tolua_beginmodule(tolua_S,NULL);
 // 创建全局 Lua 常量
  tolua_constant(tolua_S,"DISPATCH_TIME_NOW",DISPATCH_TIME_NOW);
  tolua_constant(tolua_S,"DISPATCH_TIME_FOREVER",DISPATCH_TIME_FOREVER);
  tolua_constant(tolua_S,"NSEC_PER_SEC",NSEC_PER_SEC);
  // 创建全局 Lua 函数
```

```
13
   tolua_function(tolua_S,"dispatch_get_main_queue",tolua_dispatch_lua_dispatch_ge
   t_main_queue00);
14   tolua_function(tolua_S,"dispatch_after",tolua_dispatch_lua_dispatch_after00);
15   tolua_function(tolua_S,"dispatch_time",tolua_dispatch_lua_dispatch_time00);
16  tolua_endmodule(tolua_S);
17  return 1;
18 }
```

导出后的 `m` 文件做的事情也很容易理解。就是创建全局 Lua 常量和全局 Lua 函数。

那一个完整的 GCD 调用可以是这样的。

```
1 dispatch_after(dispatch_time(DISPATCH_TIME_NOW, 1 * NSEC_PER_SEC),
  dispatch_get_main_queue(), wpfix_block(function()
2     wpfix.print("【LUA】hello siri")
3 end, "void,void"))
```

其他文章

Lua 热修复脚本调试 Lua 热修复基本用法 Lua 热修复类型签名列表

参考

wax

spa

JSPatch

TTPatch