Lua 热修复基本用法

lua 调用介绍

lua 里的方法调用是使用 `:` 来调用的，这个符号其实就是一个语法糖，比如

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

调用 OC 方法

调用类方法

```
local color = UIColor:greenColor()
```

调用实例方法

```
local view = UIView:alloc():init()
view:setNeedsLayout()
```

参数传递

当 lua 调用 OC 方法的时候，就不能再使用 `:` 号表示参数了，那么针对带参的方法调用时，我们就统一使用下划线 `_` 来表示参数。

```
local view = UIView:alloc():init()
view:setBackgroundColor_(UIColor:greenColor())

local indexPath = NSIndexPath:indexPathForRow_inSection_(0, 1)
```

属性访问

属性的读取/修改是通过 setter/getter 方法的调用

```
view:setBackgroundColor_(redColor)
local bgColor = view:backgroundColor()
```

定义类

可以通过 wpfix_class 重写一个存在的类

规则如下：

`参数1`： 类型信息，table 类型

- `key1`：要修改的类的类名，字符串类型

`参数2`： 用于声明要替换或者添加的实例方法，函数类型

`参数3`： 用于声明要替换或者添加的类方法，函数类型

```
wpfix_class({"ViewController"},
function(_ENV)
    -- 要添加或覆盖的实例方法
end,
function(_ENV)
    -- 要添加或覆盖的类方法
end)
```

覆盖实例方法

1、在 wpfix_class 里定义已经存在的方法可以覆盖 OC 方法的实现。

```
// OC
@implementation ViewController
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath
{
}
@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
    function tableView_didSelectRowAtIndexPath_(self,tableView, indexPath)
    end
end)
```

2、使用双下划线 __ 代表原 OC 方法名里的下划线 _

```
// OC
@implementation ViewController
- (NSArray *)_dataSource {
}
@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
```

```
    function __dataSource(self)
    end
end)
```

3、在 `self:origin(_ENV)` 即可调用未覆盖前的 OC 原方法

```
// OC
@implementation ViewController
- (void)viewDidLoad {
}
@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
    function viewDidLoad()
        self:origin(_ENV):viewDidLoad()
    end
end)
```

4、在 `self:super(_ENV)` 即可调用父类 OC 方法

```
// OC
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
}
@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
```

```
    function viewDidLoad()
        self:super(_ENV):viewDidLoad()
    end
end)
```

覆盖类方法

1、wpfix_class 第三个参数函数里定义已经存在的方法可以覆盖 OC 类方法的实现。

```
// OC
@implementation ViewController
+ (void)helloWorld {
}
@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
end
function(_ENV)
    function helloWorld()
    end
end)
```

2、 `class` 关键字可以调用 OC 类方法

```
// OC
@implementation ViewController
+ (void)helloWorld {
}
+ (void)printHello {
}
@end
```

```lua
-- lua
wpfix_class({"ViewController"},
function(_ENV)
end
function(_ENV)
    function helloWorld()
        class:printHello()
    end
end)
```

添加新方法

添加新方法，必须要知道新方法的方法签名，这里就需要使用定义协议的方式来声明一个方法签名

可以通过 wpfix_protocol 定义一个新的协议

用法如下：

`参数1`： 协议名，字符串类型

`参数2`： 实例方法签名，table 类型

`参数3`： 类方法签名，table 类型

类型列表的顺序规则

- 第1位: 返回值类型
- 第2位: arg0
- 第3位: arg1

```lua
-- 定义协议，用于给新增的方法添加方法签名
wpfix_protocol("WPFixNewClassProtocol", {
    refreshView = "NSString*,void",
},{
    refreshData_ = "NSDictionary*,NSDictionary*"
})
```

可以通过 wpfix_class 定义一个新类或者重写一个存在的类，并且指定要实现的协议

规则如下：

`参数1`： 类型信息，table 类型

- `key1`：要 新增类/重写类 的类名，字符串类型
- `key2`：要 新增类/重写类 的父类类名，字符串类型，其实当是重写类时，不用传入父类类名
- `protocols`：要实现的协议列表，table 数组类型

`参数2`： 用于声明要替换或者添加的实例方法，函数类型

`参数3`： 用于声明要替换或者添加的类方法，函数类型

```lua
-- lua
wpfix_class({"WPFixNewClass", "NSObject", protocols={"WPFixNewClassProtocol"}},
function(_ENV)
    function refreshView(self)
        return "有事"
    end
end,
function(_ENV)
    function refreshData_(self,data)
        data.thingName = "有事"
        return data
    end
end)
```

访问私有成员变量

```objc
// OC
@interface ViewController () {
    NSInteger _aInteger;
}
@end
```

```lua
-- lua
wpfix_class({"ViewController"},
function(_ENV)
```

```
    function viewDidLoad()
        self:super(_ENV):viewDidLoad()

        self:setIvar_withInteger_("_aInteger", 666)
        local a = self:getIvarInteger_("_aInteger")
    end
end)
```

特殊类型

结构体

1、支持 CGRect / CGPoint / CGSize / UIEdgeInsets 类型

```
// OC
UIView *view = [[UIView alloc] initWithFrame:CGRectMake(20, 20, 100, 100)];
[view setCenter:CGPointMake(10,10)];
[view sizeThatFits:CGSizeMake(100, 100)];
```

```
-- lua
local view = UIView:alloc():initWithFrame_(CGRect({ x = 20, y = 20,width =
100, height = 100}))
view:setCenter_(CGPoint({x = 10, y = 10}))
view:sizeThatFits_(CGSize({width = 100, height = 100}))
```

2、自定义结构体支持

自定义结构体，需要先注册类型和 keys，这样才能通过 key 去访问数据，可以通过 wpfix_struct 来注册一个结构体

```
wpfix_struct({name = "XPoint", types = "float,float", keys = "x,y"})
```

用法如下：

`参数1`： 注册信息，table 类型

- `name`：要注册的结构体名称
- `types`：要注册的结构体类型签名列表
- `keys`：要注册的结构体参数名称列表，keys 的顺序要跟 types 保持一致

类型列表的顺序规则

- 第1位: arg0
- 第2位: arg1
- 第3位: arg2

```
// OC
typedef struct XPoint {
    float x;
    float y;
} XPoint;
```

```
-- lua
wpfix_struct({name = "XPoint", types = "float,float", keys = "x,y"})
local point = XPoint({x = 10.0, y = 10.0})
point.x = 20.0
local x = point.x
local y = point.y
```

block

1、OC 传入 block 给 lua

lua 调用原生 block 时，直接根据参数个数调用即可

```
// OC
@implementation ViewController
```

```
3
4 - (int)blockOneArg:(int(^)(int i))block {
5       return block(11);
6 }
7
8 @end
```

```
1 -- lua
2 wpfix_class({"ViewController"},
3 function(_ENV)
4     function blockOneArg_(self,block)
5         -- 调用原生 block
6         reurn block(12)
7     end
8 end)
```

2、lua 返回 block 给 OC

lua 返回 block 给 OC，需要先定义 block 的签名。可以通过 wpfix_block 定义一个 block

```
1 wpfix_block(function(a) return "aa" end, "NSString *,NSString *")
```

用法如下：

`参数1`： lua 函数，函数类型

`参数2`： block签名列表，字符串类型

类型列表的顺序规则

- 第1位: 返回值类型
- 第2位: arg0
- 第3位: arg1

```
1 // OC
2 @implementation ViewController
3
```

```
- (NSString *(^)(NSString *))blockReturnStringWithString {
    return ^(NSString *arg1){
        return @"hh";
    };
}

@end
```

```
-- lua
wpfix_class({"ViewController"},
function(_ENV)
    function blockReturnStringWithString(self)
        return wpfix_block(function(string) return "哈哈" end, "NSString
*,NSString *")
    end
end)
```

c 函数调用

```
// OC
@implementation WPFixCFunctionTest

- (void)dispatchAfter {}

@end
```

```
-- lua
-- 调用 c 函数前，必须要先打开 c 函数绑定
wpfix.setConfig({openBindOCFunction=true})
wpfix_class({"WPFixCFunctionTest"},
function(_ENV)
    function dispatchAfter(self)
        print("非延后执行")
```

```
        -- 延后1s在朱队列执行一个定义的 block
        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, 1 * NSEC_PER_SEC),
                       dispatch_get_main_queue(),
                       wpfix_block(function()
                                   print("延后执行")
                               end, "void,void"))
    end
end)
```

支持的 c 函数列表

功能名	名称	类型
GCD	dispatch_async	函数
GCD	dispatch_sync	函数
GCD	dispatch_after	函数
GCD	dispatch_apply	函数
GCD	dispatch_once	函数
GCD	dispatch_time	函数
GCD	dispatch_get_global_queue	函数
GCD	dispatch_get_main_queue	函数
GCD	DISPATCH_QUEUE_PRIORITY_HIGH	常量
GCD	DISPATCH_QUEUE_PRIORITY_DEFAULT	常量
GCD	DISPATCH_QUEUE_PRIORITY_LOW	常量
GCD	DISPATCH_QUEUE_PRIORITY_BACKGROUND	常量
GCD	DISPATCH_TIME_NOW	常量
GCD	DISPATCH_TIME_FOREVER	常量
GCD	NSEC_PER_SEC	常量
Runtime	class_getName	函数

内置函数/常量说明

名称	类型	作用
wpfix.version	常量	获取 WPFix 的版本号
wpfix.isNull()	函数	用于判断实例或者类是否存在
wpfix.root()	函数	传入逗号分割的若干字符串参数，返回一个 绝对路径
wpfix.print()	函数	日志打印，把日志打印都输出到原生
wpfix.exit()	函数	退出 app，相当于杀掉进程
wpfix.appVersion	常量	获取 app 的版本号
NSDocumentDirectory	常量	获取 document 目录
NSLibraryDirectory	常量	获取 library 目录
NSCacheDirectory	常量	获取 cache 目录
wpfix.os.systemVersion	常量	获取系统版本号
wpfix.os.geOS()	函数	判断系统版本号是否大于等于目标版本
wpfix.device.screenWidth	常量	获取屏幕宽度
wpfix.device.screenHeight	常量	获取屏幕高度
wpfix.device.screenScale	常量	获取屏幕 scale