前言

2018 年距离第一代 iOS 系统发布（2007 年）已经过去 11 年，这 11 年中移动端日益成熟，Web 端的时代逐步转移到了移动端。各种跨平台技术层出不穷，很多公司采取了 Hybrid 方案，这就涉及到 JS 和端（Android 和 iOS）进行交互，相信你和我一样对这方面的知识感到非常吃力，往往因为一个小小的交互而加班熬夜。市面上的文章有很多，但都大同小异、不全面。为了解决这些问题，让你掌握 JS 与端交互所用到的知识点，知识小集 打算从基础出发，介绍 JS 与 iOS 交互时用到的技术点，比如 JSCore、JS 基础、JSCore 的实际使用场景（深度剖析 JSPatch 的实现）。

耗时将近 3 个月的时间，JS-Native 交互专题终于与大家见面了，其中花费了作者大量的业余时间。有很多读者朋友建议做成收费的，为了能够让更多的读者受益，我们选择免费送给读者朋友们。如果你认为这个专题不错，帮忙把它分享给你的好友。如果你有好的建议可以通过公众号 **知识小集** 给我们留言。

本主题有基础部分和进阶部分，如果你已经掌握了 JS，可以跳过 JS 部分。进阶部分主要是 JSPatch 的深度剖析。

下文提到的 JSCore 是 JavaScriptCore 的简写。

目录

第一章 认识 JS

1.1 JS 基础知识

我知道，非常多的同学，尤其是编程经验在2年以内的，他们的意识里学习一门编程语言非常难。其实不然，因为我当年和你们一样遇到同样的问题，当时学习 JS 的时候，总在想我 OC 还没学好，哪有精力学习 JS。最后得到我老大的一句点拨，通过其他语言也许你可以掌握在 OC 中掌握不了的知识。从那以后，我下定决心，两周后就可以用 JS 写 Node 了。如果，你已经下定决心，我们一起约定，一起坚持读完这个专题。

打仗前需要武器，学习一门新的语言也需要武器，我推荐使用微软出的 visual studio code ，下文会简称 vscode。

代码写完后需要执行，最简单的方式是通过浏览器来执行，这种方式需要把 JS 代码放到 HTML 中，下文中的例子会使用这种方式执行 JS 代码。当然，可以直接使用 Node 来执行 JS 代码，我更喜欢这种方式。

新建一个 `index.html` 和 `index.js` 文件， `index.js` 文件保存要执行的 JS 代码，在浏览器中打开 `index.html` 即可执行 `index.js` 文件中的 JS 代码。

```
<html>
    <head>
        <script src="index.js"></script>
    </head>
</html>
```

别放弃，你已经掌握了如何执行 JS 代码，我们一起学习 JS 基本语法。

数据类型

数据类型这部内容对我们后续介绍 JSPatch 时，JS 和 OC 间数据类型非常有用。JS 使用 `var` 声明一个变量，不需要指定变量的类型。

- Number

JS 不区分整数和浮点数，统一用 Number表示，而在 OC 中有 NSNumber，是不是很像；

```
var age = 30;
var price = 18.4;
```

- 布尔值

使用 true 和 false 表示；

```
var is_animating = false;
```

- 数组

数组用`[]`表示，元素之间用,分隔；

```
var names = ["知识小集", "iOS-tips"];
```

- 字符串

字符串是以单引号'或双引号"括起来的任意文本;

```
var name = "知识小集";
var name = '知识小集';
```

- Date

用来表示日期和时间

```
var now = new Date();
```

- 对象

JS 的对象是一组由键-值组成的无序集合；

```
var tip = {
    name : "知识小集",
    members : ["南峰子","Vong","Lefe_x","高老师很忙","故胤道
长","halohily","KANGZUBIN","陈满iOS"],
    avg_age : 25
};
tip.name; // 获取 person 的 name 属性
```

- Map

> 它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现 --- ECMAScript 6 入门

```
// 使用一个键值对的数组创建 map
var member_map = new Map([['Lefe_x', 25], ['Vong', 24]]);
console.log(member_map.get('Lefe_x')); // 25
```

- Set

Set 与 OC 中的`NSSet`类似，表示集合。保存一组 key 的集合，由于 key 不能重复，所以，在 Set 中，没有重复的 key。

```
var member_set = new Set(['Lefex', 'Vong']);
console.log(member_set.has('Lefex'));
```

函数

学完 JS 中的数据类型后，接下来主要讲一下 JS 中的函数与 OC 中有哪些不一样的地方。下面的代码是三种不同的函数定义方法。

```
// 定义一个正常的函数，函数名为 abs
function abs(x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
};

// 匿名函数
var abs2 = function (x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
};

// 匿名函数，自动调用
(function (x) {
    console.log('Hello');
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
})();
```

调用函数时参数可以传递任意个，不像 OC 中如果函数定义了 3 个参数，那么调用者必须传 3 个参数。每个函数中有一个关键字 `arguments`，通过它可以获取到参数的值，这里需要强调一点它不是数组，通常可以通过 `Array.prototype.slice.call(arguments)` 转换成数组。

```
abs(-20, 30, 40);
console.log(arguments);

// { '0': -20, '1': 30, '2': 40 }
```

如果把函数定义到某个对象中，那么这个函数就成了这个对象的方法，比如：

```
var person = {
    name : "知识小集",
    members : ["南峰子","Vong"],
    first_member: function(){
        return this.members[0]
    }
};
console.log(person.first_member());
```

apply、call 和 bind 方法

每个 JS 函数都会有方法，apply、call 和 bind，但它们的作用不同。call 和 apply 都是为了改变某个函数运行时的上下文（context）而存在的，换句话说，就是为了改变函数体内部 this 的指向。而 bind 也可以改变函数体 this 的指向，只不过它会返回一个绑定函数。

区别：

- call：

a. 用来改变函数体中 this 的指向，并立即调用;
b. 第一个参数为 this 要指向的对象，调用方式为 `fun.call(obj, parm1, parm2, parm3)`

- apply：

a. 用来改变函数体中 this 的指向，并立即调用
b. 第一个参数为 this 要指向的对象，调用方式为 `fun.call(obj, [parm1, parm2, parm3])`

- bind：

a. 用来改变函数体中 this 的指向，返回绑定后的函数，供调用者调用
b. 第一个参数为 this 要指向的对象，调用方式为
`var bindfun = fun.call(obj, parm1, parm2, parm3)`

举例：

```
var person = {
    name : "知识小集",
    members : ["南峰子","Vong"],
    member_at_index: function(index){
        return this.members[index]
    }
};

console.log(person.member_at_index(0));

var memberAtIndexFunc = person.member_at_index;

// 会报错
// console.log(memberAtIndexFunc(0));

/**
```

```
/Users/apple/Desktop/jsnative.js:6
        return this.members[index]
                            ^

TypeError: Cannot read property '0' of undefined
*/

var person2 = {
    members : ["南峰子", "Lefe_x", "Vong", "高老师很忙"],
}

// 改变 this 的指向为 person2
console.log('call:')
console.log(memberAtIndexFunc.call(person2, 1));

// 改变 this 的指向为 person2
console.log('apply:')
console.log(memberAtIndexFunc.apply(person2, [1]));

// 改变 this 的指向为 person2
console.log('bind:')
console.log((memberAtIndexFunc.bind(person2, 1))());
```

更详细的介绍

关于 `call, apply , bind` 在 JSPatch 中有非常多的应用，掌握后有助于后面的章节学习。

JSON 与字符串转换

```
// 抓换为 JSON
var js = JSON.stringify(person);
// JSON 转换为 Object
var per = JSON.parse(js);
```

类

在 JS 中，每个类都有原型 prototype。用 new Member()创建的对象会从原型上获得一个 `constructor` 属性，它指向函数 Member 本身。每个 JS 对象都是一个属性的集合，类的实现是基于其原型原型继承机制的。

```
function Member(name) {
    this.name = name;
    this.hello = function(){
        return "Hello I am " + this.name;
    }
};

var lefe = new Member('Lefe_x');
var vong = new Member('Vong');
```

```
// 对象 lefe 和 vong 都含有方法 hello，其实只需要一个 hello 方法即可，所有可以直接在
 Member 的原型上添加一个 hello 方法。

Member.prototype.hello = function(){
    return "Hello I am " + this.name;
}
```

Member 的原型链是这样的：

```
lefe ----> Member.prototype ----> Object.prototype ----> null
```

JS 对象是动态的，可以新增属性也可以删除属性。除了字符串、数字、bool、null和undefined之外，JS中的其它值都是对象。

属性特性（property attribute）：

- 可写（ writable）： 表明是否可以设置该属性的值；
- 可枚举（enumerable）： 表明是否可以通过for/in循环返回该属性；
- 可配置（configurable）： 表明是否可以删除或修改改属性。

对象的创建

- 通过对象直接量创建

```
var person = {
    name: "知识小集"
}
```

- 通过 new 创建

通过 new 创建对象，它后面跟随一个函数调用，这个函数称为构造函数（constructor）

```
var a = new Array();
```

原型

所有通过对象直接量创建的对象都具有同一个原型对象，并且可以通过 `Object.prototype` 获得对原型对象的引用。通过 new 和构造函数创建的对象的原型就是构造函数的 prototype 属性的值。比如通过 `new Object()` 创建的对象原型是 `Object.prototype`，通过 `new Array()` 创建的对象，原型是 `Array.prototype` 。

属性

`hasOwnProperty` 检查某个对象是否有某个属性，只会检查当前对象，不会检查原型对象。

```
if (person.hasOwnProperty('name')){
    console.log("has name");
}
```

使用 `in` 来检查某个对象是否有某个属性。如果自有属性或继承属性都有这个属性，则返回 ture。

```
if ('name' in person) {
    console.log("has name");
}
```

某个属性是否可以枚举。

```
if (person.propertyIsEnumerable('name')) {
    console.log("name is enumerable");
}
```

掌握了 JS 基本的语法后，我们变开始后序的章节。结下来主要掌握 JS 与 OC 之间的交互。

第二章 JSCore

2.1 JSCore 总览

JSCore 是专门用来解释和执行 JS 代码，是苹果提供给开发者可以执行 JS 代码的库，可以直接使用 OC 代码执行一段 JS 代码。这样就给 OC 与 JS 相互执行提供了可能。大名鼎鼎的 JSPatch 正是使用了 JSCore 做到动态执行 JS 代码，它内部主要的类有：

- JSContext

JSContext 表示 JS 执行上下文，当 JS 在执行的过程中，都可以通过 JSContext 来获取执行时的数据，比如某个对象的值。

- JSVirtualMachine

JS运行的虚拟机，有独立的堆空间和垃圾回收机制，它主要为 JS 执行提供资源保障。

- JSValue

JSValue 用来 JS 和 OC 中数据转换，它用来表示 JS 中的数据。我们需要明确一点，JSValue 可以是一个 JS 函数。

- JSExport

主要用来把 OC 中的属性和方法导出到 JS 环境中，方便在 JS 调用 OC。

2.2 JS 与 OC 间的类型转换

`JSValue` 表示 JS 中的数据类型，使用它可以做到 `JS` 的类型和 `Objective-C` 之间的类型转换。具体类型转换如下：

```
Objective-C type  |   JS type
------------------+---------------------
     nil          |     undefined
    NSNull        |        null
   NSString       |       string
   NSNumber       |   number, boolean
 NSDictionary     |   Object object
   NSArray        |    Array object
    NSDate        |     Date object
   NSBlock (1)    |   Function object (1)
      id (2)      |   Wrapper object (2)
    Class (3)     | Constructor object (3)
```

在 JS 中用 `JSValue` 表示不同的数据类型，可以通过不同的 OC 类型来创建 `JSValue`

通过基本的数据类型创建 `JSValue`：

```
JSValue *intValue = [JSValue valueWithInt32:10 inContext:self.context];
JSValue *boolValue = [JSValue valueWithBool:YES inContext:self.context];
```

创建一个 `JSValue` 对象，并添加 `name` 和 `age` 属性：

```
JSValue *person = [JSValue valueWithNewObjectInContext:self.context];
[person setObject:@"Lefe_x" forKeyedSubscript:@"name"];
[person setObject:@25 forKeyedSubscript:@"age"];

// 获取 name 的值
// name: Lefe_x
NSLog(@"name: %@", person[@"name"]);
// name: Lefe_x
NSLog(@"name: %@", [person objectForKeyedSubscript:@"name"]);
```

创建结构体 `JSValue`，目前支持的结构体有 CGPoint、NSRange、CGRect 和 CGSize:

```
JSValue *rectValue = [JSValue valueWithRect:CGRectMake(0, 0, 100, 100)
inContext:self.context];
[rectValue toRect];
```

通过 `JSValue` 调用某个函数：

```
[self.context evaluateScript:@"function add(a, b) {return a + b;}"];
// 调用 JS 函数
JSValue *addValue = [self.context[@"add"] callWithArguments:@[@2, @3]];
```

2.3 JS 与 OC 通信

JS 与 OC 通信，目前主要的方式有两种：

- 通过 JSCore 中的 block
- 通过 JSCore 中的 JSExport

本文主要通过这而种方式展开来讲：

通过 JSCore 中的 block

block 这种方式比较简单直接，大名鼎鼎的 `JSPatch` 就是通过这种方式实现 JS 调用 Native 的。我们看一个 JSPatch 中的例子：

在 JS 执行环境中添加一个 `_OC_catch` 的 block，那么在 JS 代码中就可以直接调用 `_OC_catch` 这个函数，当在 JS 中调用 `_OC_catch` 这个函数后，我们刚才注册的 block 就会被执行。也就是通过 JS 成功的调用了 OC 代码。

```
context[@"_OC_catch"] = ^(JSValue *msg, JSValue *stack) {
    _exceptionBlock([NSString stringWithFormat:@"js exception, \nmsg: %@,
\nstack: \n %@", [msg toObject], [stack toObject]]);
};
```

而在 `JSPatch.js` 的这个文件中，是这样被执行的：

```
try {
   // 省略具体执行的代码
} catch(e) {
    _OC_catch(e.message, e.stack)
}
```

上面这个例子主要的作用是在 JS 执行过程中抛出异常到 OC，OC 对异常进行处理。当然如果想直接在 OC 中通过 context 也可以执行 JS 代码，下面这个例子中，定义一个 `multiply` 函数，返回两个整数相乘。

```
- (void)excuteWithBlock
{
    self.context[@"multiply"] = ^(NSInteger a, NSInteger b){
        return a * b;
    };

    // 执行结果
    JSValue *result = [self.context evaluateScript:@"multiply(2,3)"
 withSourceURL:[NSURL URLWithString:@"multiply.js"]];
    // 执行后结果为 6
    NSLog(@"multiply result: %@", [result toString]);
}
```

通过 JSCore 中的 JSExport

`JSExport` 可以导出 `Objective-C` 的属性、实例方法、类方法和初始化方法到 JS 环境，这样就可以通过 JS 代码直接调用 `Objective-C`。通过 JSExport 不仅可以导出自定义类的方法、属性，也可以导出已有类的方法、属性。在导出过程中，类的方法名会被转换成 JS 类型命名，第二个参数的第一个字母会被大写，比如：

```
- (void)addX:(int)x andY:(int)y;
```

会被转换为：

```
addXAndY(x, y);
```

导出自定义类的方法、属性

在 OC 中定义一个知识小集对象 `TeachSet`，使用 `JSExport` 导出。

```
@class Member;
@class TeachSet;

@protocol JSTeachSetExportProtocol<JSExport>
@property (nonatomic, copy) NSString *name;
+ (TeachSet *)teachSet;
- (void)initWithName:(NSString *)name
             members:(NSArray<Member *> *)members;
- (NSArray<Member *> *)currentMembers;
JSExportAs(add, -(BOOL)addMember:(Member *)member);
@end


NS_ASSUME_NONNULL_BEGIN


@interface TeachSet : NSObject<JSTeachSetExportProtocol>

@property (nonatomic, copy) NSString *name;

+ (TeachSet *)teachSet;
- (instancetype)initWithName:(NSString *)name
                     members:(NSArray<Member *> *)members;

- (BOOL)addMember:(Member *)member;
- (NSArray<Member *> *)currentMembers;
+ (BOOL)maxMemberCount;

@end
```

```
// 导出 TeachSet 对象
TeachSet *teachSet = [TeachSet teachSet];
self.context[@"_OC_teachSet"] = teachSet;

// 导出 TeachSet 类
self.context[@"_OC_TeachSet"] = [TeachSet class];

// 导出 Member 类，并创建一个 Member 对象添加到 TeachSet 对象中
// 通过构造函数的方式创建 Member 对象
// addMember 被重命名为 add
self.context[@"_OC_Member"] = [Member class];
[self.context evaluateScript:@"var member = new _OC_Member('Lefe_x',
25);_OC_teachSet.add(member);"];

// 通过类方法创建 Member 对象添加到 TeachSet 对象中
[self.context evaluateScript:@"var member =
_OC_Member.member();member.name='Lefe_x_1';member.age=26;_OC_teachSet.add(
member);"];

// 没导出会报错，_OC_teachSet.maxMemberCount is not a function
[self.context evaluateScript:@"_OC_teachSet.maxMemberCount()"];

// 获取最终 TeachSet 中的成员数
JSValue *membersValue = [self.context
evaluateScript:@"_OC_teachSet.currentMembers()"];

/**
 membersValue: (
   "name: Lefe_x, age: 25",
   "name: Lefe_x_1, age: 26"
 )*/
NSLog(@"membersValue: %@", [membersValue toArray]);
```

导出已有类的方法、属性

把已有类的方法，属性导出到 JS 环境，允许 JS 调用。下面这个例子是导出 `UILabel` 的 `text` 属性到 JS 中，这样可以直接在 JS 中就可以设置 `UILabel` 的 `text`。我们需要做的事就是创建一个遵循 `<JSExport>` 协议的 `UILabelExportProtocol` 协议。

```
@protocol UILabelExportProtocol<JSExport>
@property (nullable, nonatomic, copy) NSString *text;
@end
```

```
// 通过 runtime 给 UILabel 添加协议 UILabelExportProtocol
class_addProtocol([UILabel class], @protocol(UILabelExportProtocol));

UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(40, 100, 200,
44)];
label.text = @"知识小集";
label.textColor = [UIColor blackColor];
[self.view addSubview:label];

// 把 label 导出到 JS 环境，在 JS 执行环境中 label 的名字为 _OC_label，通过这个名
字来调用 label 的 text 属性
self.context[@"_OC_label"] = label;

// 通过 JS 修改 label 的 text 属性，执行完成后 label 的 text 被修改为 '关注知识小
集公众号'
[self.context evaluateScript:@"_OC_label.text='关注知识小集公众号'"];
```

2.4 OC 与 JS 通信

目前来说 OC 与 JS 通信，主要有 UIWebView 、WKWebView 和 JSCore 这三种方式。而 UIWebView 的方式其实可以看作是 JSCore 的方式。

第一种方式：通过 JSCore 来调用

在说明 JSCore 这种方式的时候，需要先了解一下 JSCore。它是苹果 Safari 浏览器的 JS 引擎。

1.**创建 Context**

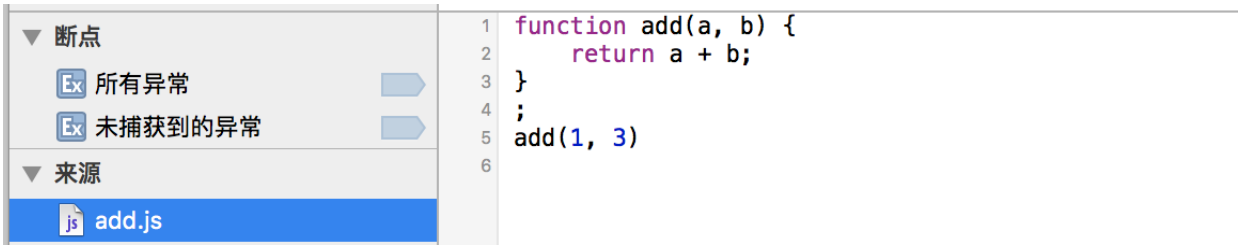
```
_context = [[JSContext alloc] init];
// 设置 context 的名字后，调试的时候特别重要
_context.name = @"lefex.context";
```

2.**执行 JS 代码**

使用 `evaluateScript: withSourceURL:` 执行 JS 代码，其中 `add.js` 是为了调试 JS 代码，不会影响 JS 代码的执行结果。

```
// 要执行的 JS 代码，定义一个 add 函数并执行
NSString *addjs = @"function add(a, b) {return a + b;};add(1,3)";
// sumValue 为执行后的结果
JSValue *sumValue = [self.context evaluateScript:addjs withSourceURL:
[NSURL URLWithString:@"add.js"]];
NSLog(@"sum: %@", @([sumValue toInt32])); // 4
```

通过 safari 调试 JSContext，可以看到 `add.js` 文件，它是刚才运行 js 代码时设置的名字。



可以通过 `callWithArguments:` 来调用 JS 中的某个方法。

```
NSString *addjs = @"function add(a, b) {return a + b;}";
[self.context evaluateScript:addjs withSourceURL:[NSURL
URLWithString:@"add.js"]];
JSValue *resultValue = [self.context[@"add"] callWithArguments:@[@2, @4]];
NSLog(@"Result: %@", @([resultValue toInt32])); // 6
```

执行一段 JS 代码，使用 Safari 调试可以看到全局变量 `name`，`log_name`，`age` 和 `sum_age`。有了这些全局变量，就可以通过 context 获取 `name`，`log_name`，`age` 和 `sum_age` 的值。

```
NSString *js = @"var name = \"Lefe_x\";var log_name = function(aname){var
res = 'Hello ' + aname;console.log(res);};log_name(name);var age = 24;var
sum_age = age + 1;";
[self.context evaluateScript:js withSourceURL:[NSURL
URLWithString:@"lefe.js"]];

NSLog(@"%@", [self.context[@"name"] toString]); // Lefe_x
NSLog(@"%@", [self.context[@"sum_age"] toString]); // 25
```

给 JSContext 全局对象（Gloable object）添加一个属性，名为 `ObjectC_add` 值为 `ObjectC`

```
[self.context setObject:@"ObjectC" forKeyedSubscript:@"ObjectC_add"];
```

3.**监听异常**

如果执行 JS 代码时，如果有异常会执行这个 block。

```
[self.context setExceptionHandler:^(JSContext *context, JSValue
*exception) {

}];
```

第二种方式：通过 来调用：

这种方式说白了就是使用 `JSCore`，通过 `UIWebView` 来获取 `JSContext`，这样直接通过获取到 context 来执行 JS 代码。

```
JSContext *context = [_webView
valueForKeyPath:@"documentView.webView.mainFrame.JSContext"];
```

第二种方式：通过 WKWebView 来调用：

WKWebView 没有提供获取 JSContext 的方法，但是它提供了执行 JS 的方法 `evaluateJS:`，通过这个方法来执行 JS 代码。

```
[self.webView evaluateJS:@"function add(a, b) {return a + b;};add(1,3)"
completionHandler:^(id _Nullable msg, NSError * _Nullable error) {
   NSLog(@"evaluateJS add: %@, error: %@", msg, error);
}];
```

第三章 调试

3.1 MAC 搭建本地 Web 服务

想让应用访问网页，那么必须启动一个服务 WebServer，Mac 电脑自带了一个 web 服务器 `Apache`

```
查看服务器的信息：
➜  WebKit httpd -v
Server version: Apache/2.4.28 (Unix)
Server built:   Oct  9 2017 19:54:20
```

启动服务器：

```
➜  ~ sudo apachectl start
```

启动成功后，在浏览器中输入 `http://127.0.0.1/`，即可访问到默认的网页。在目录 `/Library/WebServer/Documents` 会放人网页中的信息。把 hybird 这个网页信息放到这个目录下既可以访问。

```
http://127.0.0.1/hybird/home
```

3.2 调试 WebView

目前调试 WebView 主要有三种方式：

通过 Safari 和手机配合

当运行 APP 的时候，iOS 端加载 WebView（WKWebView 或 UIWebView ）时可以通过 Mac 自带的 Safari 来调试所显示的页面，其实调试 JSPatch 的时候也是这么用的。

我们来模拟加载 Web 页时的场景，首先需要开启本地的 WebServer，mac 自带 Apache 服务器，我们只需启动这个服务器，即可加载一个网页。

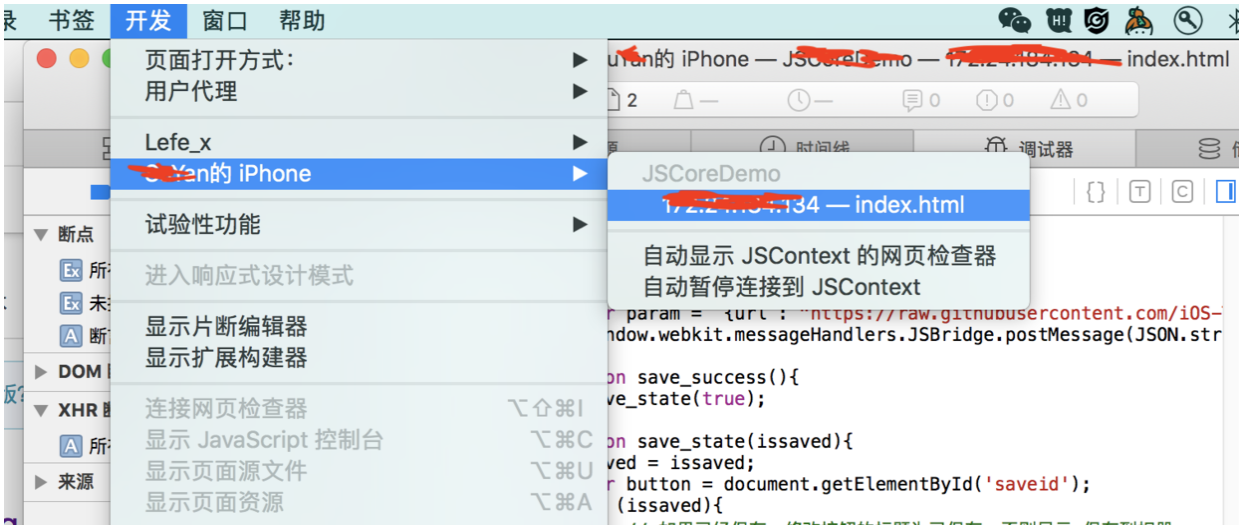
```
// 开启 Apache
sudo apachectl start
```

Apache 开启后，站点的目录在 `/Library/WebServer/Documents` 下，我们把写好的网页放到这个目录下，然后直接可以根据 URL 访问对应的页面，比如在浏览器中输入：`http://电脑ip地址/web/index.html` 即可访问 `index.html` 这个页面。

使用 WKWebView 加载 `index.html` 这个页面，即可调试这个页面，调试前需要做以下两件事：

- 手机端开启Web 检查器：设置 -> 通用 -> Safari -> 高级 -> Web 检查器
- Mac端显示开发菜单：Safari 浏览器默认没有显示“开发”菜单，需要通过：Safari 浏览器 -> 偏好设置 -> 高级 -> 勾选在菜单中显示“开发”设置。

设置完后，当启动 APP ，加载 WKWebView 后即可看到 `index.html` 这个页面。这时即可通过断点进行调试，当然可以查看当前的 HTML 代码，JS 代码，网络情况等。具体如下图所示：



把日志信息转移到 NA 端输出

在 debug 环境植入一个 webview console，可以在黑盒下不连电脑不连 safari 调 DOM 和 JS，另外在开发期间 JS 中的异常信息比如： console.log、 alert，window.onerror 改为 bridge 的方式，通过 NA 端来输出异常信息。

把 WebView 用来调试的 log、alert、error 显示到 NA ，在调试时会方便不少。做 WebView 与端交互的时候，主要用 `window.webkit.messageHandlers.xxx.postMessage(params);` 来给端发消息，也就是说 WebView 想给端发消息的时候直接调用这个方法即可，端会通过 `WKScriptMessageHandler` 的代理方法来接收消息，而此时端根据和 WebView 约定的规则进行通信即可。

```
- (void)userContentController:(WKUserContentController
*)userContentController didReceiveScriptMessage:(WKScriptMessage *)message
```

而添加调试信息，无非就是给 WebView 添加了 log、alert、error 这些消息的 bridge，这样当 WebView 给端发送消息后，端根据和 WebView 约定的规则解析 log、alert、error 为端对应的事件，比如 log 直接调用端的 `NSLog`，alert 调用端的 `UIAlertController`。

集成控制台

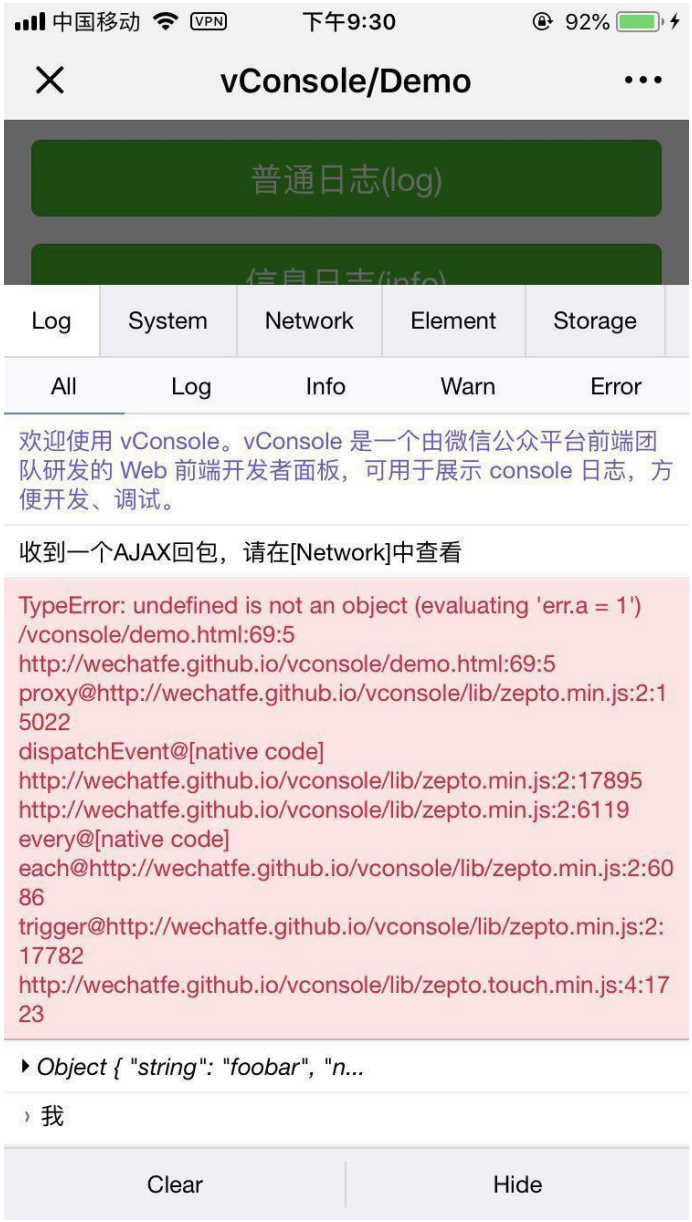
黑盒下调试 WebView，无需连接电脑和 safari 即可调试 DOM，这个可以参考小程序的 vConsole 或者 eruda 。可以直接在 WebView 中接入，或者在端中接入。这里以在端中接入 eruda 为例，这里踩到几个坑：

1.有些页面显示不出来，估计是故意屏蔽掉的，味精特意使用 JSBox 试了下其它页面，发现百度等都不可以显示调试按钮，而掘金是可以的；

2.使用本地的页面也显示不出来，这是 webview 跨域安全方面的考虑，file 协议下会禁止 js css html 以部分 file，部分网络的方式加载。

下面这段代码直接在 webview 加载完成后执行即可。

```
NSString *js = @"(function() {var script =
document.createElement('script');script.type = 'text/JS';script.src
= 'https://xteko.blob.core.windows.net/neo/eruda-
loader.js';document.body.appendChild(script);})();";
[self.webView evaluateJS:js completionHandler: nil];
```



3.3 了解 WKWebView

在 PC 届比较出名的浏览器有：

- Chrome：Google 的基于 Webkit 内核浏览器，内置了非常强悍的JS引擎——V8；
- Mozilla 自己研制的 Gecko 内核和 JS引擎OdinMonkey；
- Apple 的 Mac系统自带的基于Webkit内核的浏览器；

目前在移动端主要使用的是 Apple 的 Safari 和 Google 的 Chrome。在浏览器中可以有一个全局对象 `window` 表示当前显示的窗体。

```
获取浏览器窗口的整个宽高：

console.log(window.innerWidth);
console.log(window.innerHeight);
```

navigator 可以获取到浏览器的信息：

```
console.log('userAgent = ' + navigator.userAgent);
```

screen 对象获取屏幕的信息:

```
screen.width
screen.height
```

location 获取当前页面的URL信息:

```
console.log(location.href);
```

一个 HTML 渲染到屏幕上时，其实是一颗树。而 DOM （ Document Object Model ）就表示这棵树的结构，document 是 DOM 的根节点，可以根据 document 提供的一些方法，动态的来操作 DOM。

WKWebView 是开源的，详细信息可以看 源码

第四章 实战

4.1 自己动手实现一个 Hybrid WebView

如今，端与 Web 页的交互越来越频繁，很多页面都交给 Web 页面来实现，而有些情况下 Web 需要与端进行交互。面对这种需求，各种第三方库源源不断出现，而 WebViewJSBridge 无疑是 star 最多的一个。其实目前在 iOS 开发当中，大多数都切换到了 WKWebView，且对 Web 的交互越来越重，所以不妨自己实现一个 Hybrid WebView 来满足自己的业务需求。一个 Hybrid WebView 最基本的应该满足双方可以自由通信。

- WebView 上的事件可以传递到端上；
- WebView 可以从端上获取数据；
- 端可以监听到 WebView 上发生的事件。

本文旨在说明一个 Hybrid WebView 需要的技术手段，所以打算从一个具体的需求出发，一步一步搭建一个 Hybrid WebView。大多数的文章只会讲解端上如何实现，而本文会结合前端一块讲讲两端是如何实现的。

需求说明

Web 页面上有一张图和一个保存按钮，当点击保存按钮时会提示用户是否需要保存图片到相册。如果保存成功，按钮的标题将变为已保存，否则标题为保存到相册。如果已保存，下次进入 Web 页时显示已保存。

分析上面的需求，可以拆分为:

- 页面加载后，需要获取图片是否已经保存过，如果已保存，按钮的标题为“已保存”，否则为“保存到相册”；
- 点击按钮需要提示用户“是否需要保存图片到相册”，点击“保存”执行保存操作。点击取消将什么也不做；

- 保存成功，按钮上的标题需要变为“已保存”。

分析完上面具体需求后，转换为技术需要考虑的问题：

- 页面加载后，Web 页可以从端上获取到图片是否已经保存的状态；
- 点击保存按钮，需要在端上提示用户，用户点击保存需要把图片保存到相册，这时需要获取到当前显示的图片，也就是说需要把 Web 页面中的数据传递到端；
- 保存成功后需要修改 Web 页面按钮的标题。

先做一个 Web 页面

整体页面是如上图所示。我们逐步剖析是如何实现的。

在前面的章节中（这些章节后续会发出来），已经介绍了在 Web 页面中执行 JS 。可以把一段 JS 代码嵌入到 HTML 中，这时在 HTML 中可以直接调用 JS 代码，而 JS 可以通过 DOM 动态来操作 HTML 中的标签，这样既可以达到动态修改 Web 页。

Web与端通信的JS代码，这段代码是嵌入在 HTML 中的。

```
<script>
    // 标记保存的状态
    var saved = false;
    // 保存事件
    function saveaction(){
        if (saved) {
            return;
        }
        alert("确定要保存该图片吗？");
        // 发送消息给客户端 JS 中发送消息给 OC
        var param =  {url : "https://raw.githubusercontent.com/iOS-Tips/iOS-tech-set/master/images/qrcode.jpg"};

 window.webkit.messageHandlers.JSBridge.postMessage(JSON.stringify(param));
    };
    // 保存成功后端会调用这个方法通知Web页保存成功
    function save_success(){
        change_state(true);
    };
    // 修改是否已保存的状态，修改按钮标题
    function change_state(issaved){
        saved = issaved;
        var button = document.getElementById('saveid');
        if (issaved){
            // 如果已经保存，修改按钮的标题为已保存，否则显示 保存到相册
            button.innerText = "已保存";
        } else {
            button.innerText = "保存到相册";
        }
    }
</script>
```

保存到相册 按钮，监听点击事件，当点击按钮后会调用 `saveaction` 函数。

```
<div id="saveid" class="save_button" onclick="saveaction()">保存到相册</div>
```

而 `saveaction` 函数首先会发一个 `alert("确定要保存该图片吗？")` 到端，端会执行 `WKUIDelegate` 代理方法，我们在这个方法需要弹窗端内的提示框：

```
- (void)webView:(WKWebView *)webView runJSAlertPanelWithMessage:(NSString
*)message initiatedByFrame:(WKFrameInfo *)frame completionHandler:(void
(^)(void))completionHandler
{
    UIAlertController *alert = [UIAlertController
alertControllerWithTitle:@"温馨提示" message:message
preferredStyle:UIAlertControllerStyleAlert];
    [alert addAction:[UIAlertAction actionWithTitle:@"保存"
style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action)
{
        self.isOKAction = YES;
        completionHandler();
    }]];
    [alert addAction:[UIAlertAction actionWithTitle:@"取消"
style:UIAlertActionStyleDefault handler:^(UIAlertAction * _Nonnull action)
{
        self.isOKAction = NO;
        completionHandler();
    }]];
    [self presentViewController:alert animated:YES completion:nil];
}
```

当用户点击保存按钮后，会保存图片到相册。所以客户端需要拿到图片的地址，这是需要给端发送图片的地址。如果想给端发送一条消息，直接在 Web 页通过 JS 执行，其中 xxxx 是端与Web之间约定的名字。

```
window.webkit.messageHandlers.xxxx.postMessage(JSON.stringify(param))
```

而我们此时定义的名字是 `JSBridge`，当用户点击保存后，需要根据Web传递过来的 URL 保存图片。

```
var param =  {url : "https://raw.githubusercontent.com/iOS-Tips/iOS-tech-
set/master/images/qrcode.jpg"};
window.webkit.messageHandlers.JSBridge.postMessage(JSON.stringify(param));
```

当端接收到 Web 发过来的消息后，会调用 `WKScriptMessageHandler` 的代理方法，在这个方法中我们来下载图片并保存到相册：

```
- (void)userContentController:(WKUserContentController
*)userContentController didReceiveScriptMessage:(WKScriptMessage *)message
{
    if ([message.body isKindOfClass:[NSString class]]) {
        if ([message.name isEqualToString:kScriptMsgName] &&
self.isOKAction) {
            // 保存图片
            NSDictionary *msgInfo = [NSJSONSerialization
JSONObjectWithData:[message.body dataUsingEncoding:NSUTF8StringEncoding]
options:NSJSONReadingAllowFragments error:nil];
            UIImage *image = [[UIImage alloc] initWithData:[NSData
dataWithContentsOfURL:[NSURL URLWithString:msgInfo[@"url"]]]];
            if (image) {
                UIImageWriteToSavedPhotosAlbum(image, self,
@selector(imageSavedToPhotosAlbum:didFinishSavingWithError:contextInfo:),
nil);
            }
        }
    }
}
```

当把图片保存到相册后，需要刷新 Web 页面上的按钮的标题，这时需要执行 Web 页中已经定义好的 `change_state` 方法：

```
- (void)updateSaveState:(BOOL)isSave
{
    NSString *script = isSave ? @"change_state(true);" :
@"change_state(false);";
    [self.webView evaluateJS:script completionHandler:^(id _Nullable msg,
NSError * _Nullable error) {}];
}
```

至此，我们还剩下最后一件事没有完成，当加载出 WebView 后，需要根据本地是否已经保存了图片更新按钮的标题，直接调用 `updateSaveState` 函数即可。

总结

本文主要介绍一个 Hybrid WebView 如何实现，它仅仅是从一个具体的需求出发，而如果做一个通用 Hybrid WebView 框架需要两端设计一种通信规则。具体细节可以参考味精的两篇关于 Hybrid 的实践 (从零收拾一个hybrid框架)。本文的 demo 会在这个专题完成后一块放出。

4.2 Hybrid 实战：如何完整下载一个 wap 页面

> 本文的读者需要有一定的 Hybrid 基础，相关的概念已经有很多优秀的文章进行过讲解，这里不再赘述。本文的重点在于如何基于 Hybrid 框架，在移动端（不限于具体平台）完成一个 wap 页面的完整下载。这个页面会包含**丰富的媒体资源，包括图片、音频、视频**等。下载完成之后，我们在**离线**时依然能够流畅阅读该页面的完整内容。

场景分析

在移动端使用 wap 页面展示内容有非常多的应用场景，比如常见的新闻资讯阅读、简书或掘金的文章展示。客户端提供 webview 容器，配套一系列前端与客户端交互的能力，前端使用模板批量生成不同的内容后下发，发挥着前端极高灵活性的同时又可以享受到客户端原生的许多能力。

前面说的场景，用户很少有缓存的需求。不过对于某些产品中用户可能会反复查看的内容，提供页面的下载以供离线查看就很有必要了。wap 页面不仅可以展示文本，还经常出现图片，有的页面中还会包含音频甚至视频，该采用什么方式完成下载，需要根据不同的场景具体分析。

方案选择

一般而言，提供了下载功能的 wap 页面，它将展示的内容形式必然是有限制的。如果任何一个页面丢过来都要求缓存，我们很难保证离线情况下的展示效果。

利用 NSURLProtocol 实现缓存

对于简单的页面，比如它只包含文本和图片，那么不依赖 Hybrid 也可以实现。NSURLProtocol 就是一个可选方案，并且它的覆盖范围可以更广。使用 NSURLProtocol 对网络请求进行过滤，如果是特定的 html、css、js 等请求和图片请求，就在本地缓存数据中进行查找，命中缓存则使用本地数据包装成网络请求的响应数据进行回传，否则才执行对应的网络请求。这样即使在网络可用的情况下，也会优先使用本地缓存数据，不仅可以完成页面的离线展示，还有效地加快了网络可用时页面的加载速度。自然想到，这也是很多 wap 页面加速方案会采用的方式。

基于 Hybrid 的 wap 页面

而对于包含了音频、视频等内容的 wap 页面，出于用户体验等目的，客户端往往会将这些资源的展示进行接管。比如页面中包含音乐时，用户进行点击会发现是客户端的播放器在进行播放，这便是 Hybrid 应用的例子。它的一个常见实现方案是：客户端为前端提供了播放音频的接口，对于 wap 页面中的所有音频，放弃前端的原生写法，采用客户端提供的接口。而客户端中这个接口的实现，自然是调起客户端音频播放器，播放对应的音频，这个过程中又会涉及到是否成功的回调、客户端播放器和 wap 页中播放状态的同步等诸多问题，不过这不是本文关注的重点。**本文讨论的需要完整下载的 wap 页面的场景，就是基于这种方案。**

具体实现

页面的展示方案

通过上面的例子可以看到，基于 Hybrid 方案展示的 wap 页面，不仅需要客户端提供面向前端调用的各种能力，也需要前端在编写代码时进行相应的调用。两端相互配合，完成高度定制化的 wap 页面。对于上面的例子，客户端提供出来图片、音频、视频的接口，这样 wap 页面中对应内容的交互，便都交由客户端的原生实现。而客户端接口的实现，自然就是**首先在本地缓存数据中进行查找，若该数据已下载，则使用本地数据，否则进行对应的网络获取，以此兼容有网和离线的情况。**

交由客户端接管的好处，除了上述离线缓存的应用，还有一点是能够让 webview 中的内容也可以享受到客户端原生的处理方式。比如图片接口的客户端实现，就可以交由项目中使用的网络图片异步加载工具（SDWebImage 或 YYWebImage 等），这样 webview 中的图片加载和原生场景下的图片加载可以共享缓存数据，原生场景加载过的图片，在 webview 中展示时便无需重复网络获取，反过来也如此。基于局部性原理，这样可以在一定程度上达到图片加载加速的效果（减少了重复网络请求）。

进一步优化

事实上，在高度定制的 wap 页面场景下，我们对于 webview 中可能出现的页面类型会进行严格控制。可以通过内容的控制，避免 wap 页中出现外部页面的跳转，也可以通过 webview 的对应代理方法，禁掉我们不希望出现的跳转类型，或者同时使用，双重保护来确保当前 webview 容器中只会出现我们定制过的内容。既然 wap 页的类型是有限的，自然想到，同类型页面大都由前端采用模板生成，页面所使用的 html、css、js 的资源很可能是同一份，或者是有限的几份，把它们直接随客户端打包在本地也就变得可行。加载对应的 url 时，直接 load 本地的资源。

对于 webview 中的网络请求，其实也可以交由客户端接管，比如在你所采用的 Hybrid 框架中，为前端注册一个发起网络请求的接口。wap 页中的所有网络请求，都通过这个接口来发送。这样客户端可以做的事情就非常多了，举个例子，NSURLProtocol 无法拦截 WKWebview 发起的网络请求，采用 Hybrid 方式交由客户端来发送，便可以实现对应的拦截。

基于上面的方案，我们的 wap 页的完整展示流程是这样：**客户端在 webview 中加载某个 url，判断符合规则，load 本地的模板 html，该页面的内部实现是通过客户端提供的网络请求接口，发起获取具体页面内容的网络请求，获得填充的数据从而完成展示。**

下载时如何确定具体需要下载的资源

客户端如何确定某个具体 wap 页中会包含的资源，我们没有找到很高效的办法。所幸这种场景下的页面多是内部生产、高度定制的，这也就为后端提供资源统计提供了可能（页面数据本来就是后端生产的~），所以我们采用了后端提供对应接口支持的方案，根据 wap 页的 id，可以获取该页面包含的具体资源。执行用户的下载行为时，使用对应页面 id 调用资源统计接口，获得该页面包含的资源。这其中包括 JSON 格式的页面填充数据，也包括媒体文件的 url。对于 JSON 数据，可以采用简单的缓存框架进行缓存（YYCache 等），图片采用常见的图片缓存框架，音频、视频等可以处理成常规的文件下载任务。

结语

- 本文并没有关注具体的 Hybrid 框架如何实现，重点在于提供一个基于 Hybrid 可以实现的具体业务场景。
- 文中所述的方案，是公司内部场景下剥离了业务的技术实现阐述。会有不尽合理的地方，欢迎讨论。

4.3 深入 JSPatch 原理

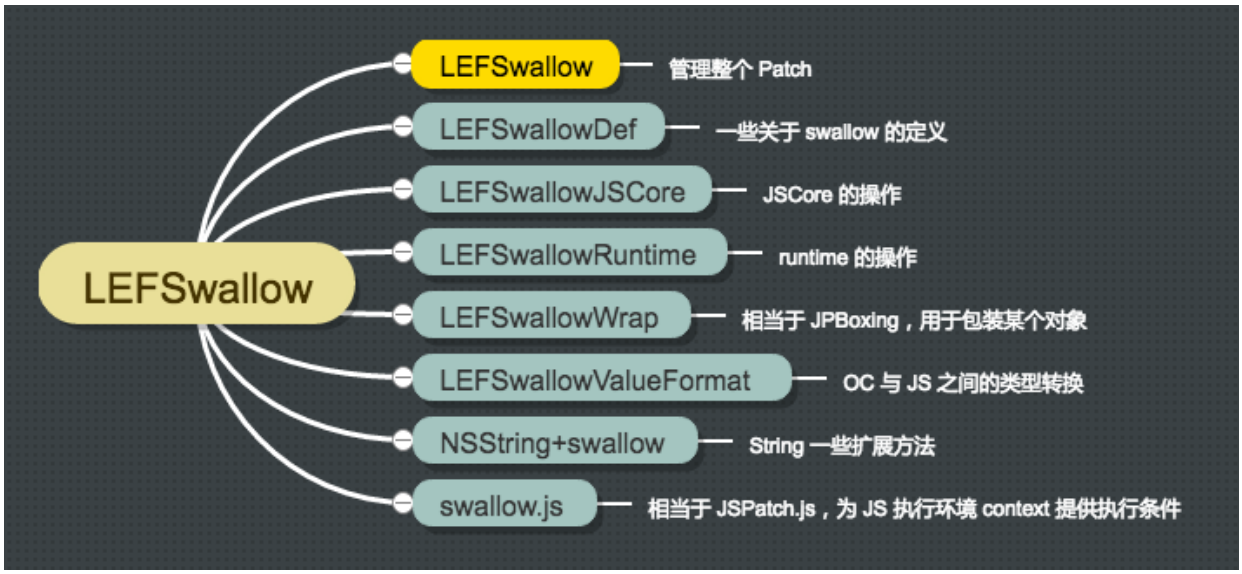
4.3.1 苹果已经禁止 JSPatch 上线，为什么我还在研究它?

大前端目前非常火，各种跨平台方案横扫移动端圈内，很多 iOS 程序员开始转入大前端的潮流。虽然开发者往往把 JSPatch 用来做热修复，而我认为他有更好玩的地方，比如使用 JS 开发原生界面。目前 JSBox 非常好地利用了 Native 与 JS 交互这一特性，相信你和我一样都非常好奇其中所使用到的技术，好奇终归为好奇，毕竟没有开源。那么不如从 JSPatch 出发，探一探用 JS 如何写原生界面。下面

几个点是我学习 JSPatch 主要目的：

- 好奇，JSPatch 仅仅用了不到 2000 行代码就可以使用 JS 开发原生界面；
- 学习 JSCore 框架，对这个框架总是一知半解，希望通过 JSPatch 来掌握它；
- 学习 JS，JSPatch.js 这个文件中有很多 JS 高级用法；
- 学习 runtime 的实际使用场景，通过 runtime 的真实场景来掌握 runtime；
- 世面上缺乏关于 JSPatch 原理全面介绍的文章，几乎所有的文章对某些知识点点到为止，根本没有理解具体的实现细节；

本文也是对前几章的实践，如果你对 JS，JSCore 都不了解，建议阅读前面的章节，有助于理解本文的内容。JSPatch 虽然只有 3 个文件（不算扩展），其实它把很多内容都融入到了一起，而且使用了大量的宏定义，读起来十分困难。最终 Lefe_x 写了 JSPatch 一个乞丐版，目的是为了掌握 JSPatch，有大量的注释，它也许会是你入门 JSPatch 的一个突破口。项目主要分为了 runtime 部分，JSCore 部分和管理部分，旨在能让你更容易地掌握 JSPatch 的细节。



4.3.2 概述

> JSPatch bridge Objective-C and JS using the Objective-C runtime. You can call any Objective-C class and method in JS by just including a small engine. JSPatch is generally used to hotfix iOS App.

JSPatch 使用 Objective-C 的 runtime 让 JS 调用任意的 Objective-C 的类和方法。如果想要动态调用 Objective-C 的任意类、任意方法，恰巧 Objective-C 的 runtime 可以做到这一点。那么 JSPatch 的核心就是把 JS 中的方法调用，转换成 Objective-C 的方法调用，并且 JS 和 Objective-C 之间需要进行数据传递。

总之，你可以想成 JSPatch 就是把某些字符串转换成 Objective-C 可以识别的类和方法，而字符串没有逻辑运行能力，也没有执行能力，这就是为什么要用 JS 的原因，当然主要是因为在 iOS 中集成的 JSCore 这个库，这样就让 iOS 拥有了执行 JS 的能力。

> JS是一种解释型的脚本语言，C、C++ 等语言先编译后执行，而 JS 是在程序的运行过程中逐行进行解释。- 百度百科

JS 不需要编译即可解释执行，这样就可以在程序执行的过程中动态地执行某些 JS 代码。而 JSPatch 在程序启动后使用 JSCore 执行了 JSPatch.js 这个文件中的脚本，这个文件主要为 JS 执行环境（JSContext）提前创建一个全局属性、方法等，这样在后续补丁执行的过程中才能找到对应的方法。

说了这么多，估计你已经晕了，没关系，我本来就没打算让你听懂。下面我们开始梳理 JSPatch 的核心逻辑。掌握了 JSPatch 的实现，你就可以很好地掌握 JS 与 Objective-C 之间的交互，而且对你理解 runtime 非常有帮助。

4.3.3 先理解 JSPatch.js

我们试想一下，为什么 JSPatch 能执行下发的脚本?

JSPatch 要想执行下发的脚本，必须能够在 JSCore 中找到对应的方法，如果方法不存在，将直接报错。而 JSPatch.js 的作用就是提前在 JSContext 注入后续执行补丁用到的方法。在 JPEngine 的 `startEngine` 的方法中可以看到在执行脚本前需要加载 JSPatch.js：

```
 [_context evaluateScript:jsCore withSourceURL:[NSURL
URLWithString:@"JSPatch.js"]];
```

除此之外，还在 `_context` 注入了其它方法，而下面这些方法就是 JS 与 OC 之间执行的桥梁，在 JS 中可以执行下面已经定义好的方法。

- `_OC_defineClass`：在执行环境 context 中定义要替换或新增的方法、属性；
- `_OC_defineProtocol`：定义协议；
- `_OC_callI`：调用实例方法；
- `_OC_callC`：调用类方法；
- `_OC_formatJSToOC`：JS 转换成 OC；
- `_OC_formatOCToJS`：OC 转换成 JS；
- `_OC_getCustomProps`：获取自定义的属性；
- `_OC_setCustomProps`：增加自定义的属性；
- `__weak`：`__weak` 的实现；
- `__strong`：`__strong` 的实现；
- `_OC_superClsName`：获取父类；
- `autoConvertOCType`：自动转换成 OC 类型；
- `convertOCNumberToString`：自动把 Number 类型转换成字符串；
- `include`：引入其它 JS 文件；
- `resourcePath`：资源文件路径；
- `_OC_log`：把 console.log 打印信息转换到 NSLog 打印；
- `_OC_catch`：把 JS 中的 try catch 转换到 OC 中；
- `dispatch_sync_main`：GCD 的支持。

我们需要看看 JSPatch.js 中究竟都做了那些事：

自执行函数

JSPatch.js 文件中的代码是自执行的，它不需要主动调用，也就是一旦使用 JSCore 加载完这个文件，就会执行。

```
;(function() {
   // JS 代码
})()
```

给 Object 添加属性

`_customMethods` 这个对象中定义了函数 `__c`、`super`、`performSelectorInOC`、`performSelector`，这些函数被添加到 Object 的原型链（prototype）上，这样每个 JS 对象都会有这 4 个方法。

```
Object.defineProperty(Object.prototype, method, {value:
_customMethods[method],configurable: false, enumerable: false});
```

- `__c`

脚本的执行过程中，每个方法的调用都会经过它进行转发。JSPatch 想要调用任意类的方法，前提是需要在 JS 执行环境中存在这些类的方法，不然调用就会出错，就如同在 OC 中调用一个不存在的方法将直接 crash，JS 中也类似。所以作者巧妙的利用了 `__c` 作为消息的转发函数。假如执行一段脚本。

脚本执行前：

```
defineClass('ViewController', {
    changeName_age: function(name, age) {
        self.setTitle("Lefe_x");
    }
})
```

脚本格式转换后，把 `self.setTitle("Lefe_x");` 这个方法转换后变成了 `self.__c("setTitle")("Lefe_x");`，这样当调用 setTitle 这个方法时，其实是调用的 `__c` 这个方法，最后在转发到 OC 中。

```
;(function(){try{
defineClass('ViewController', {
    changeName_age: function(name, age) {
        self.__c("setTitle")("Lefe_x");
    }
})

}catch(e){_OC_catch(e.message, e.stack)}})();
```

defineClass

defineClass 的定义如下：

```
defineClass(classDeclaration, [properties,] instanceMethods, classMethods)

@param classDeclaration: 字符串，类名/父类名和Protocol
@param properties: 新增property，字符串数组，可省略
@param instanceMethods: 要添加或覆盖的实例方法
@param classMethods: 要添加或覆盖的类方法
```

想要添加或重写某个方法就需要使用 defineClass 来申明，通过 JS 来告诉 OC 的 runtime，那些方法需要修改，需要添加那些属性等。

require

require 其实是在全局生成一个 JS 对象，这样当调用某个对象的方法时便可以找到这个对象，以免找不到对象而报错。

```
global[clsName] = {
   __clsName: clsName
}
```

console.log

主要目的是把在 JS 中使用 console.log 打印的日志信息使用 OC 中的 NSLog 打印，方便调试。

defineProtocol

协议的定义，主要给 OC 中添加协议。它主要的目的是给某个对象增加方法时，需要知道方法参数的类型时，通过协议中的方法来获取 method type。

方法说明

- `var _propertiesGetFun = function(name){}`

属性的 Get 方法，当新增属性后，会调用对象的 Get 方法和 Set 方法；

- `var _propertiesSetFun = function(name) {}`

属性的 Set 方法，当新增属性后，会调用对象的 Get 方法和 Set 方法；

- `var _formatDefineMethods = function(methods, newMethods, realClsName) {}`

负责消息转换，它会把补丁中的方法转换成 [参数个数，自定义函数] 的形式传给 OC 层，OC 利用它来做 runtime 的工作和消息回调；

- `var _setupJSMethod = function(className, methods, isInst, realClsName){}`

在 JS 执行环境中添加实例方法和类方法；

- `var _formatOCToJS = function(obj) {}`

负责 OC 与 JS 类型的转换；

- `var _methodFunc = function(instance, clsName, methodName, args, isSuper, isPerformSelector) {}`

主要负责把 JS 方法转换成 OC 方法并回调到 OC，它最终调用了 OC 的 `_OC_callC` 和 `_OC_callI`；

- `global.require = function(clsName) {}`

在全局生成一个 JS 对象，目的是与 OC 进行交互，相当于创建一个对象；

- `global.defineProtocol = function(declaration, instProtocol, clsProtocol)`

`{}`

定义一个协议，主要用来获取方法的 method type，这样可以添加不同类型参数的方法，否则只能添加参数和返回值都为 id 类型的方法；

- `global.defineJSClass = function(declaration, instMethods, clsMethods) {}`

如果不需要与 OC 进行交互，可以用它来定义 JS 类、方法；

- `global.defineClass = function (declaration, properties, instanceMethods, classMethods) {}`

定义要修改或添加的方法、属性；

- `__c: function(methodName) {}`

JS 消息转发函数；

- `super: function() {}`

super 的实现，可以使用 OC 中的 super 关键字。

总结

JSPatch.js 中有很多关于 JS 闭包的使用，如果对 JS 不太熟悉，看起来源码比较吃力，建议补一下 JS 的知识，当然本专题前面的章节有提到这里用到的 JS 知识。看到这里，你其实只要明白了 JSPatch.js 的作用即可，这里总结一下：

它主要在 JSCore 中的 context 中提前注入了一些方法，为后续的补丁执行提供执行条件，比如 `defineClass` 和 `__c` 方法是在 JSPatch.js 中提前定义的方法，这样在下发的补丁才会找到找到这两个方法，保证调用不出错。

4.3.4 方法是如何被调用的呢?

我们从一个例子出发，例子中需要替换在 ViewController 中定义个的空方法 `clickAction:` 为当点击按钮时把按钮上的标题显示到 ViewController 的标题上。脚本下发前如下：

```
defineClass('ViewController', {
    clickAction: function(sender) {
        self.setTitle(sender.currentTitle());
    }
})
```

脚本被格式化后为：

```
;(function(){try{
defineClass('ViewController', {
    clickAction: function(sender) {
        self.__c("setTitle")(sender.__c("currentTitle")());
    }
})

}catch(e){_OC_catch(e.message, e.stack)}})();
```

当按钮被点击后，`clickAction:` 将被执行，由于它先前已经存在，所以在 OC 的 runtime 会把它替换为补丁代码中的实现。这里需要注意脚本被格式后的形式，方法的调用都变成了对 `__c` 的方法调用，而它恰好在 JSPatch.js 中已添加到了 Object 对象上，这样所有的对象都有 `__c` 方法，保证调用不出错。

defineClass 声明替换方法

在这个例子中，defineClass 有两个参数:

- 类的声明： `ViewController` 字符串
- 需要替换的实例方法:

```
{
    clickAction: function(sender) {
        self.__c("setTitle")(sender.__c("currentTitle")());
    }
}
```

defineClass 会把 clickAction：方法转换成 `[参数个数，自定义函数]` 的形式传递给 OC 中的 defineClass，这样在 OC 的 runtime 中即可知道参数的个数和原函数。为何要转换成这种形式，需要说明一下：

参数个数的存在的意义主要有：

- a. 在 runtime 中如果想给某个类中增加方法需要用到

```
class_addMethod(Class _Nullable cls, SEL _Nonnull name, IMP _Nonnull imp,
const char * _Nullable types)
```

除了 types 这个参数外，其它的参数都可以获取到，而 types 这个这个值称为 Method type，如果方法存在可以直接通过 `method_getTypeEncoding` 来获取，否则只能根据参数的类型自己拼接，而 JSPatch 新增的方法必须是 id 类型（使用协议除外），这样知道参数个数就可以拼接出方法的 Method type。

- b. 修复方法名，clickAction 方法名在 OC 中应该是 `clickAction:`，所以需要根据参数个数来添加一个 `:`

```
if ([selectorName componentsSeparatedByString:@":"].count - 1 <
numberOfArg){
  selectorName = [selectorName stringByAppendingString:@":"];
}
```

自定义函数存在的意义主要有：

- a: 对于替换的方法 clickAction，当按钮点击后会触发这个方法，而这个方法的实现是通过补丁下发实现的，所以调用这个方法的实现需要调用补丁中的实现。JSValue 可以是个函数，通过 `callWithArguments:` 方法来调用，这就达到了我们的目的，通过下发一个 JS 函数，当按钮点击后，获取到方法调用的参数，直接调用 JS 函数即可调用的补丁中的实现。
- b: self 的实现，我们都清楚 JSPatch 支持 self，原因就在于它在全局对象 global 中保存了最后一次调用的类。

其实理解起来也不难，你可以把 JS 函数想象成 OC 中的 block，当按钮被点击后直接执行一个 block。

下面这段代码就是传递到 OC 时的形式：

```
{
    clickAction =     (
        1,
        function () {
            try {
            var args =
_formatOCToJS(Array.prototype.slice.call(arguments));
            var lastslf = global.self;
            global.self = args[0];
            if (global.self) {
                global.self.__realClsName = realClsName;
            }
            args.splice(0,1);
            var ret = originMethod.apply(originMethod, args);
            global.self = lastslf;
            return ret;
        } catch (e) {
            _OC_catch(e.message, e.stack);
        }
       }
  )
}
```

回到 OC 中的 defineClass 方法

经过 JS 把需要替换的方法（clickAction：）转换为`[参数个数，自定义函数]`形式后，需要利用 OC 的 runtime 对 `clickAction:` 方法进行替换。

当按钮被点击后 `clickAction:` 会被执行，这个时候我们需要获取 `clickAction:` 的参数址，传给 jsFunc（上面提到的自定义函数）。

我们一起看下 jsFunc 的执行过程：

```
function () {
     try {
            var args =
_formatOCToJS(Array.prototype.slice.call(arguments));
            var lastslf = global.self;

            global.self = args[0];
            if (global.self) {
                global.self.__realClsName = realClsName;
            }
            args.splice(0,1);
            var ret = originMethod.apply(originMethod, args);
            global.self = lastslf;
            return ret;
       } catch (e) {
            _OC_catch(e.message, e.stack);
      }
}
```

- a. 通过 `_formatOCToJS` 把参数转换成 JS 类型（因为在执行 JS 代码，必须使用 JS 的数据类型）；
- b. arguments 这个是参数不是数组，利用 `Array.prototype.slice.call(arguments)` 转换成数组，arguments[0] 为 self；
- c. 修改 global.self，目的是调用类的实例方法时可以使用 self 关键字；

> JSPatch支持直接在defineClass里的实例方法里直接使用 self 关键字，跟OC一样 self 是指当前对象，这个 self 关键字是怎样实现的呢？实际上这个self是个全局变量，在 defineClass 里对实例方法进行了包装，在调用实例方法之前，会把全局变量 self 设为当前对象，调用完后设回空，就可以在执行实例方法的过程中使用 self 变量了。

d. JS 函数调用 `var ret = originMethod.apply(originMethod, args);` 。

⚠️：这里用到的 JS 知识在 JS 部分有讲解。

当调用 `originMethod.apply(originMethod, args);` 它会调用下面的这个方法。

```
clickAction: function(sender) {
   self.__c("setTitle")(sender.__c("currentTitle")());
}
```

self 指向当前实例对象，调用 `__c` 方法，sender 也调用 `__c` 方法，从源码中可以看出，它返回一个函数。当执行 `sender.__c("currentTitle")()` 方法时，其实最终执行的是 `_methodFunc` 方法。

```
__c: function(methodName) {
      return function() {
         var args = Array.prototype.slice.call(arguments);
         return _methodFunc(slf.__obj, slf.__clsName, methodName, args,
slf.__isSuper);
      }
}
```

`_methodFunc` 最终会调用到 `_OC_callI` 或者 `_OC_callC`，这样就达到了 JS 调用 OC 代码。

```
var _methodFunc = function(instance, clsName, methodName, args, isSuper,
isPerformSelector) {
      var selectorName = methodName;
      // 省略转换成 OC 方法的代码
      var ret = instance ? _OC_callI(instance, selectorName, args, isSuper)
: _OC_callC(clsName, selectorName, args);
      return _formatOCToJS(ret);
}
```

总结

以上就是一个脚本的完整执行过程，我们可以总结为：

- 补丁下发
- 补丁格式替换为 `__c`
- 对需要添加或修改的方法进行处理，传递给 OC，使用 runtime 处理
- 调用下发的 JS 函数
- 调用补丁代码的实现
- 调用 `__c` 函数
- 调用 `_OC_callI` 或者 `_OC_callC`

4.3.5 JSPatch 中的 runtime

上面的例子中，我们并没有对 JSPatch 中的 runtime 进行展开，主要围绕 JSPatch.js 中的方法进行说明，下面我们谈一谈 JSPatch 中的 runtime 是如何工作的。

试想一下，runtime 是如何进行方法添加或替换的?

JSPatch 中的关键点就是把补丁中要新增或替换的方法与 OC 中的方法对应起来，在 OC 方法执行中，如果执行的方法是补丁中的方法，那么就要执行补丁中的实现，在执行补丁中的方法时需要把参数的值传递给它。

JSPatch 中使用 defineClass 与 OC 中的 runtime 交互，当某个补丁下发时，需要告诉 runtime 新增的属性，要添加或替换的类方法、实例方法，类名，父类名，协议，这样 runtime 即可根据这些信息进行修改。

从下面的 `class_addMethod` 和 `class_replaceMethod` 方法可知，要想添加或替换某个方法，需要知道它的 methodType，而对于已有方法可以通过 `method_getTypeEncoding` 获取到，反之，JSPatch 会通过添加协议来获取，如果未定义协议，只能添加参数和返回值都为 id 类型的方法。

```
class_addMethod(Class _Nullable cls, SEL _Nonnull name, IMP _Nonnull imp,
                const char * _Nullable types)
class_replaceMethod(Class _Nullable cls, SEL _Nonnull name, IMP _Nonnull
imp,
                    const char * _Nullable types)
```

到这里我们需要想一下，当某个方法被调用时如何获取它的参数值?

我们仍然使用这段代码进行说明：

```
defineClass('ViewController', {
    clickAction: function(sender) {
        self.setTitle(sender.currentTitle());
    }
})
```

由于 `clickAction:` 方法已经存在于 `ViewController` 这个类中，它的实现将被替换为补丁中 `clickAction` 的实现。那么当按钮点击后，需要调用补丁中的实现，而补丁的实现需要获取它的参数值 `sender`。咋办?

为了达到这个目的 JSPatch 做了一系列的方法替换。

> 当调用一个 NSObject 对象不存在的方法时，并不会马上抛出异常，而是会经过多层转发，层层调用对象的 -resolveInstanceMethod:, -forwardingTargetForSelector:, -methodSignatureForSelector:, -forwardInvocation: 等方法，其中最后 -forwardInvocation: 是会有一个 NSInvocation 对象，这个 NSInvocation 对象保存了这个方法调用的所有信息，包括 Selector 名，参数和返回值类型，最重要的是有所有参数值，可以从这个 NSInvocation 对象里拿到调用的所有参数值 - JSPatch

有了 `-forwardInvocation:` 这个突破口，目前需要做的就是把要替换的方法指向 `_objc_msgForward` 或者 `_objc_msgForward_stret`， 这样当调用 `clickAction:` 方法时就调用了 `-forwardInvocation:` 方法。

每个 NSObject 都有 `-forwardInvocation:` 方法，而我们想在 `-forwardInvocation:` 中做的事就是获取 `clickAction:` 的参数后，调用补丁中的实现。目前 JSPatch 把 `-forwardInvocation:` 这个方法的实现替换成了 `JPForwardInvocation`。这样消息转发后会执行 `JPForwardInvocation`。

到目前为止：

- a. `clickAction:` 被替换成了 `_objc_msgForward`
- b. `-forwardInvocation:` 被替换成了 `JPForwardInvocation`

那么问题来了，如果我想执行未被替换前的 `clickAction:` 和 `-forwardInvocation:` 方法咋么办，作者巧妙的给原先的实现替换为方法 `ORIGclickAction:` 和 `-ORIGforwardInvocation:`。

> 我们把 ViewController 的 -forwardInvocation: 方法的实现给替换掉了，如果程序里真有用到这个方法对消息进行转发，原来的逻辑怎么办？首先我们在替换 -forwardInvocation: 方法前会新建一个方法 -ORIGforwardInvocation:，保存原来的实现IMP，在新的 -forwardInvocation: 实现里做了个判断，如果转发的方法是我们想改写的，就走我们的逻辑，若不是，就调 -ORIGforwardInvocation: 走原来的流程。

```
JSValue *jsFunc = getJSFunctionInObjectHierachy(slf, JPSelectorName);
if (!jsFunc) {
    JPExecuteORIGForwardInvocation(slf, selector, invocation);
    return;
}
```

总结

这部分的核心内容就是要达到执行补丁中方法实现的目的，并可以当方法被调用时可以把参数的值传给 JS 中的实现。JSPatch 的做法就是把方法的调用转向 JPForwardInvocation，从而获取方法调用时的参数值。

4.3.6 从官方的例子理解

官方例子的作用是替换 JPViewController 中的 handleBtn：方法，并创建一个新的视图控制器 JPTableViewController，点击按钮后 push 出 JPTableViewController。我们将以这个例子来逐步展开。下面这段代码，我们称为补丁 `main.js`，文中所提到的 `main.js` 指的都是它。

```
// defineClass 函数在 JSPatch.js 文件中已经定义，在执行补丁执行前 JSPatch.js 文件已被执行
defineClass('JPViewController', {
  handleBtn: function(sender) {
    var tableViewCtrl = JPTableViewController.alloc().init()
    self.navigationController().pushViewController_animated(tableViewCtrl, YES)
  }
})

defineClass('JPTableViewController : UITableViewController <UIAlertViewDelegate>', ['data'], {
  dataSource: function() {
    var data = self.data();
    if (data) return data;
    var data = [];
    for (var i = 0; i < 1; i ++) {
      data.push("cell from js " + i);
    }
    self.setData(data)
    return data;
  },
  tableView_numberOfRowsInSection: function(tableView, section) {
    return self.dataSource().length;
  },
```

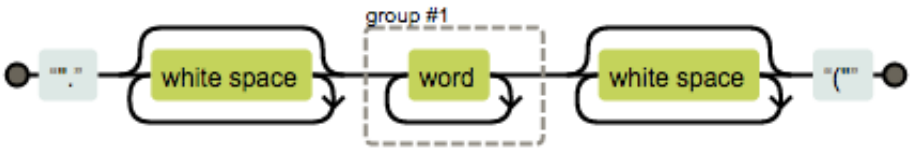
```
    tableView_cellForRowAtIndexPath: function(tableView, indexPath) {
      var cell = tableView.dequeueReusableCellWithIdentifier("cell")
      if (!cell) {
        cell =
  require('UITableViewCell').alloc().initWithStyle_reuseIdentifier(0,
  "cell")
      }
      cell.textLabel().setText(self.dataSource()[indexPath.row()])
      return cell
    },
    tableView_didSelectRowAtIndexPath: function(tableView, indexPath) {
       var alertView =
  require('UIAlertView').alloc().initWithTitle_message_delegate_cancelButton
  Title_otherButtonTitles("Alert",self.dataSource()[indexPath.row()], self,
  "OK",  null);
       alertView.show()
    },
    alertView_willDismissWithButtonIndex: function(alertView, idx) {
      console.log('click btn ' + alertView.buttonTitleAtIndex(idx).toJS())
    }
  })
```

JSPatch 为了保证调用每个方法不出错，在调用每个方法时都会加一个 `__c` 方法，这样执行 OC 方法时其实执行的是 `__c` 方法。而转换过程中使用的正则表达式为：

```
"(?<!\\\\)\\.\\s*(\\w+)\\s*\\("
```

这样看起来非常不舒服，我们把转译 `\` 去掉，变成了：

```
"(?<!\\)\.\s*(\w+)\s*\("
```



`?<!\\` 的作用就是忽略掉注释。

举例：

```
JPTableViewController.alloc().init()
```

首先找出满足正则表达式条件的字符串为：`.alloc(` 和 `.init(`，然后执行替换 `__c("$1")(` 操作，其中 `$1` 表示第一个分组，它们的值为：`alloc` 和 `init`。

最终的替换结果为：

```
JPTableViewController.__c("alloc")().__c("init")()
```

一句话总结：正则表达式的作用是找到类似 `.xxxx(` 的字符串（不包含被注释的代码），把它替换为 `__c("xxxx")(`。

我们看看整体的替换结果：

```
;(function(){try{
defineClass('JPViewController', {
  handleBtn: function(sender) {
    var tableViewCtrl = JPTableViewController.__c("alloc")().__c("init")()
    self.__c("navigationController")().__c("pushViewController_animated")
(tableViewCtrl, YES)
  }
})

defineClass('JPTableViewController : UITableViewController
<UIAlertViewDelegate>', ['data'], {
  dataSource: function() {
    var data = self.__c("data")();
    if (data) return data;
    var data = [];
    for (var i = 0; i < 1; i ++) {
      data.__c("push")("cell from js " + i);
    }
    self.__c("setData")(data)
    return data;
  },
  tableView_numberOfRowsInSection: function(tableView, section) {
    return self.__c("dataSource")().length;
  },
  tableView_cellForRowAtIndexPath: function(tableView, indexPath) {
    var cell = tableView.__c("dequeueReusableCellWithIdentifier")("cell")
    if (!cell) {
      cell = require('UITableViewCell').__c("alloc")
().__c("initWithStyle_reuseIdentifier")(0, "cell")
    }
    cell.__c("textLabel")().__c("setText")(self.__c("dataSource")()
[indexPath.__c("row")()])
    return cell
  },
  tableView_didSelectRowAtIndexPath: function(tableView, indexPath) {
     var alertView = require('UIAlertView').__c("alloc")
().__c("initWithTitle_message_delegate_cancelButtonTitle_otherButtonTitles
")("Alert",self.__c("dataSource")()[indexPath.__c("row")()], self, "OK",
 null);
     alertView.__c("show")()
  },
```

```
  alertView_willDismissWithButtonIndex: function(alertView, idx) {
    console.__c("log")('click btn ' + alertView.__c("buttonTitleAtIndex")
(idx).__c("toJS")())
  }
})

}catch(e){_OC_catch(e.message, e.stack)}})();
```

替换完后调用执行脚本的方法：

```
[_context evaluateScript:formatedScript withSourceURL:resourceURL]
```

我们看到补丁中主要调用了 `defineClass` 这个方法，而 `defineClass` 会把方法进行转换执行 OC 中的 `defineClass` 方法，OC 会使用 runtime 对这些方法进行处理。由于 JPViewController 中的 `handleBtn:` 这个方法已经存在，执行替换操作。而 JPTableViewController 这个类不存在，需要添加 JPTableViewController 这个类，由于它实现了协议 `UIAlertViewDelegate` 需要添加该协议。

JS 转换后交给 OC 时的格式：

```
▸[0] = @"alertView_willDismissWithButtonIndex" : @"2 elements"
▸[1] = @"data" : @"2 elements"
▸[2] = @"tableView_cellForRowAtIndexPath" : @"2 elements"
▸[3] = @"setData" : @"2 elements"
▸[4] = @"tableView_numberOfRowsInSection" : @"2 elements"
▸[5] = @"tableView_didSelectRowAtIndexPath" : @"2 elements"
▸[6] = @"dataSource" : @"2 elements"
```

OC 中的 `defineClass` 最终都会交给下面这个方法统一处理：

```
overrideMethod(Class cls, NSString *selectorName, JSValue *function, BOOL
isClassMethod, const char *typeDescription)
```

这个方法主要作用就是改变方法的 IMP，在 JSPatch 的 runtime 部分有详细说明。

补丁中关键点是 `defineClass` 这个函数，我们一起看看它在 JSPatch.js 中的定义：

```
defineClass(classDeclaration, [properties,] instanceMethods, classMethods)

@param classDeclaration: 字符串，类名/父类名和Protocol
@param properties: 新增property，字符串数组，可省略
@param instanceMethods: 要添加或覆盖的实例方法
@param classMethods: 要添加或覆盖的类方法
```

这个方法的主要作用是告诉 JSPatch 我想要对那些方法做出修改，增加那些属性。JSPatch 想要执行这段脚本，前提需要在 JSContext 中定义 defineClass 这个方法。

JPEngine 中的 startEngine 方法的作用就是在 JSContext 上下文中注册后续执行脚本需要的方法，而这些方法在 `JSPatch.js` 这个文件中已经定义了，而这个文件的主要作用是为后序下发的脚本做铺垫，保证下发的脚本可以正常执行。

在 OC 这层，在 JS 执行环境中添加了 `_OC_defineClass`，这样在 JS 中可以直接调用 `_OC_defineClass` 这个函数，它最终会调用 `defineClass` 这个函数，关于这个函数在上面我们有对它进行分析：

```
context[@"_OC_defineClass"] = ^(NSString *classDeclaration, JSValue
*instanceMethods, JSValue *classMethods) {
    return defineClass(classDeclaration, instanceMethods, classMethods);
};
```

在 JS 这层，我们看看关于 defineClass 的定义（在 JSPatch.js 文件中），这段代码比较长，中间删除一部分细节，读者可自行在 JSPatch.js 中找到对应的定义：

```
global.defineClass = function(declaration, properties, instMethods,
clsMethods) {
    // 这里调用到了 OC 这层中的 _OC_defineClass
    var ret = _OC_defineClass(declaration, newInstMethods, newClsMethods)

    return require(className)
}
```

我们可以把 `defineClass` 的作用理解成：为 JS 执行环境与 OC 的 runtime 打通关系，以便后续代码执行时可以顺利执行补丁中的方法。

上面的补丁调用了两次 defineClass 函数，两次调用传递的参数个数不一样，这点不像 Objective-C 定义了几个参数就必须传几个参数，在 defineClass 方法内部会根据具体的参数类型做判断。

第一个 `defineClass` 主要做了：

- 声明了类 JPViewController
- 定义了要添加或覆盖的实例方法 handleBtn

第二个 `defineClass` 主要做了：

- `JPTableViewController : UITableViewController <UIAlertViewDelegate>` 申明了类名，父类以及协议；
- 新增了属性 data；
- 定义了要添加或覆盖的实例方法 `dataSource, tableView_numberOfRowsInSection, tableView_cellForRowAtIndexPath, tableView_didSelectRowAtIndexPath , alertView_willDismissWithButtonIndex`

声明

在 defineClass 中必须声明要给那个类添加方法，它的继承关系是什么，实现的协议什么。

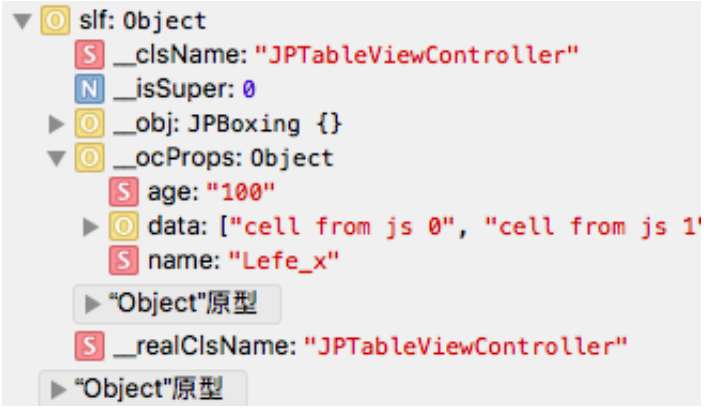
属性

JSPatch 可以给某个类新增属性，在 JS 这层，主要在实例方法中新增了 get 和 set 方法。在 Objective-C 这层，主要通过关联属性给某个类添加属性。从 `_propertiesGetFun` 和 `_propertiesSetFun` 方法中可以看出它们调用了 `_OC_getCustomProps` 和 `_OC_setCustomProps` 方法。

```
instMethods[name] = _propertiesGetFun(name);
instMethods[nameOfSet] = _propertiesSetFun(name);

var _propertiesGetFun = function(name){
return function(){
  // this 指调用者，谁调用这个方法，this 就指向谁
  var slf = this;
  if (!slf.__ocProps) {
    var props = _OC_getCustomProps(slf.__obj)
    if (!props) {
      props = {}
      _OC_setCustomProps(slf.__obj, props)
    }
    slf.__ocProps = props;
  }
  return slf.__ocProps[name];
};
}

var _propertiesSetFun = function(name){
return function(jval){
  var slf = this;
  if (!slf.__ocProps) {
    var props = _OC_getCustomProps(slf.__obj)
    if (!props) {
      props = {}
      _OC_setCustomProps(slf.__obj, props)
    }
    slf.__ocProps = props;
  }
  slf.__ocProps[name] = jval;
};
}
```

如果添加多个属性后，可以看到所有的属性将被添加到 `_ocProps` 中。

```
defineClass('JPTableViewController', ['data', 'name', 'age'], {...})
```



我们知道在 Objective-C 中可以通过 `objc_setAssociatedObject` 给一个类添加属性，通过 `objc_getAssociatedObject` 来获取属性，我们看看 Objective-C 层的具体实现。

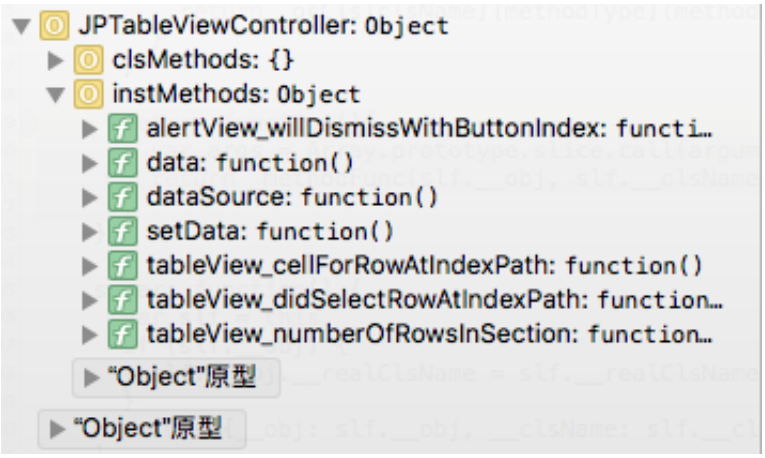
```
context[@"_OC_getCustomProps"] = ^id(JSValue *obj) {
   id realObj = formatJSToOC(obj);
   // JS 对象
   return objc_getAssociatedObject(realObj, kPropAssociatedObjectKey);
};

context[@"_OC_setCustomProps"] = ^(JSValue *obj, JSValue *val) {
   id realObj = formatJSToOC(obj);
   objc_setAssociatedObject(realObj, kPropAssociatedObjectKey, val,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
};
```

通过 `objc_getAssociatedObject` 获取到的属性为一个 JSValue 对象。

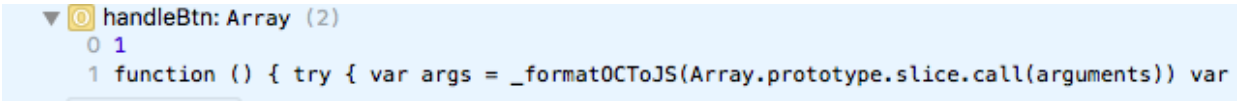
```
{
    age = 100;
    data =     (
        "cell from js 0"
    );
    name = "Lefe_x";
}
```

实例、类方法

我们需要明确一点，JSPatch 把补丁中的实例方法和类方法保存到了 `_OCCls` 这个变量中，它的结构如下：



实例和类方法会通过 `_formatDefineMethods` 这个方法转换，然后传给 Objective-C 层，通过 runtime 进行处理。比如：它会把函数 `handleBtn:` 转换为下面这种形式：



数组里保存了参数的个数和函数新的实现。我们看看具体的转换代码：

```
var _formatDefineMethods = function(methods, newMethods, realClsName) {
for (var methodName in methods) {
  if (!(methods[methodName] instanceof Function)) return;
  (function(){
    var originMethod = methods[methodName]
    newMethods[methodName] = [originMethod.length, function() {
      try {
        var args = _formatOCToJS(Array.prototype.slice.call(arguments))
        var lastSelf = global.self
        global.self = args[0]
        if (global.self) global.self.__realClsName = realClsName
        args.splice(0,1)
        var ret = originMethod.apply(originMethod, args)
        global.self = lastSelf
        return ret
      } catch(e) {
        _OC_catch(e.message, e.stack)
      }
    }]
  })()
 }
}
```

把转换后的函数通过 `_OC_defineClass(declaration, newInstMethods, newClsMethods)` 传递给 Objective-C 层。

到这里我们已经把 defineClass 要做的事通通讲完了，我们做个总结：

- 下发补丁，补丁中主要是使用了 defineClass 函数
- JSPatch 会把下发的补丁使用正则进行替换，把方法的执行替换成了 `__c(#1)` 的形式，目的是

为了下发的补丁代码在后续可以顺利执行，因为在 JS 中已经为 Object 定义了 `__c` 方法；

- defineClass 会对脚本的声明、实例方法、类方法、属性，进行处理交给 OC；
- OC 中的 defineClass 会利用 runtime 进行处理；

整个 defineClass 其实就做了这些事情。除了定义还有一个关键的部分，就是执行阶段。

脚本执行过程

我们一块看看下面这段脚本的调用过程：

```
handleBtn: function(sender) {
    var tableViewCtrl = JPTableViewController.__c("alloc")().__c("init")()
    self.__c("navigationController")().__c("pushViewController_animated")
(tableViewCtrl, YES)
}
```

`JSPatch.log:` 开头的为 JSPatch.js 中消息转发 `__c` 的调用。

```
1.点击事件触发 handleBtn: ，走 OC 的消息转发
JPForwardInvocation: <JPViewController: 0x7f8fbb5106e0> = handleBtn:

2. alloc 的调用，先调用 JS 的消息转发 `__c`，然后调用 OC 中的 alloc 方法，返回值为一
个 JS 对象

JSPatch.log: __c ->JPTableViewController -> method: alloc
 [JPTableViewController alloc]
return value: {
    "__clsName" = JPTableViewController;
    "__obj" = "<JPTableViewController: 0x7f8fbb500d50>";
}

3. init 的调用
JSPatch.log: __c ->JPTableViewController -> method: init
[<JPTableViewController: 0x7f8fbb500d50> init]
return value: {
    "__clsName" = JPTableViewController;
    "__obj" = "<JPTableViewController: 0x7f8fbb500d50>";
}

4. navigationController 的调用
JSPatch.log: __c ->JPViewController -> method: navigationController
[JPBoxing - JPViewController navigationController]
return value: {
    "__clsName" = UINavigationController;
    "__obj" = "<UINavigationController: 0x7f8fbc80f400>";
}

5. pushViewController: 调用
```

```
JSPatch.log: __c ->UINavigationController -> method:
pushViewController_animated
[<UINavigationController: 0x7f8fbc80f400> pushViewController:animated:]
```

我们看看第二个 `defineClass` 中补丁方法的调用过程，调用下面这段代码的整体过程如下：

```
dataSource: function() {
    var data = self.__c("data")();
    if (data) return data;
    var data = [];
    for (var i = 0; i < 1; i ++) {
      data.__c("push")("cell from js " + i);
    }
    self.__c("setData")(data)
    return data;
},

tableView_numberOfRowsInSection: function(tableView, section) {
    return self.__c("dataSource")().length;
},
```

```
1. OC 中的 JPForwardInvocation 被调用
JPForwardInvocation: <JPTableViewController: 0x7f8fbb500d50> =
tableView:numberOfRowsInSection:

2. dataSource 方法被调用，涉及到属性的添加和获取
JSPatch.log: __c ->JPTableViewController -> method: dataSource

3. data 属性的获取，它会与 OC 层进行交互，因为 OC 层也保存了属性
JSPatch.log: __c ->JPTableViewController -> method: data
_OC_getCustomProps JPBoxing - JPTableViewController -
<JPTableViewController: 0x7f8fbb500d50> - (null)
_OC_setCustomProps JPBoxing - JPTableViewController -
<JPTableViewController: 0x7f8fbb500d50> - {
}

4. push 方法调用
JSPatch.log: __c ->undefined -> method: push

5. setData 方法被调用
JSPatch.log: __c ->JPTableViewController -> method: setData
```

`tableView_cellForRowAtIndexPath` 方法调用过程：

```
tableView_cellForRowAtIndexPath: function(tableView, indexPath) {
    var cell = tableView.dequeueReusableCellWithIdentifier("cell")
    if (!cell) {
      cell =
require('UITableViewCell').alloc().initWithStyle_reuseIdentifier(0,
"cell")
    }
    cell.textLabel().setText(self.dataSource()[indexPath.row()])
    return cell
},
```

```
1. JPForwardInvocation 方法被调用
JPForwardInvocation: <JPTableViewController: 0x7fe9c150d360> =
tableView:cellForRowAtIndexPath:

2. JS 层的 dequeueReusableCellWithIdentifier 方法被调用
JSPatch.log: __c ->UITableView -> method:
dequeueReusableCellWithIdentifier

3. 回到 OC 层，调用 dequeueReusableCellWithIdentifier
[<UITableView: 0x7fe9c207fa00 dequeueReusableCellWithIdentifier:]
return value: {
    "__isNil" = 1;
}

4. js 层调用 UITableViewCell 的创建
JSPatch.log: __c ->UITableViewCell -> method: alloc

5. 回到 OC 层调用 alloc 方法
[UITableViewCell alloc]
return value: {
    "__clsName" = UITableViewCell;
    "__obj" = "<UITableViewCell: 0x7fe9c1904000>";
}

6. js 层调用 initWithStyle_reuseIdentifier 方法
JSPatch.log: __c ->UITableViewCell -> method:
initWithStyle_reuseIdentifier

7. 回到 OC 层调用 initWithStyle:reuseIdentifier: 方法
[<UITableViewCell: 0x7fe9c1904000 initWithStyle:reuseIdentifier:]
return value: {
    "__clsName" = UITableViewCell;
    "__obj" = "<UITableViewCell: 0x7fe9c1904000>";
}

8. 调用方法 textLabel
JSPatch.log: __c ->UITableViewCell -> method: textLabel
```

```
[<UITableViewCell: 0x7fe9c1904000 textLabel]
return value: {
    "__clsName" = UITableViewLabel;
    "__obj" = "<UITableViewLabel: 0x7fe9c1438450>";
}

9. 调用 setText，它的值需要从 dataSource 中获取
JSPatch.log: __c ->UITableViewLabel -> method: setText
JSPatch.log: __c ->JPTableViewController -> method: dataSource
JSPatch.log: __c ->JPTableViewController -> method: data
_OC_getCustomProps JPBoxing - JPTableViewController -
<JPTableViewController: 0x7fe9c150d360> - {
    data =     (
        "cell from js 0"
    );
}
JSPatch.log: __c ->NSIndexPath -> method: row
[<NSIndexPath: 0xc000000000000016> {length = 2, path = 0 - 0} row]
return value: 0

10. 回到 OC 层调用 setText: 方法
[<UITableViewLabel: 0x7fe9c1438450;> setText:]
```

总结

通过对上面的方法调用，我们可以得知，JS 与 OC 之间主要通过 `JPForwardInvocation` 和 `__c` 相互调用，这两个方法相当于扮演一个桥梁的角色，它们会根据不同的方法在 JS 和 OC 中彼此调用。这里需要注意返回值是一个 JS 对象，它是链式调用的前提条件。

4.2.7 总结

JSPatch 中有太多的内容要学，本章还没有讲到比如不定参数的调用，扩展等。不过如果掌握本章提到的内容后，掌握其它内容便非常容易。现在对 JSPatch 做个总结。

JSPatch 主要分两个部分：

第一个部分：为补丁执行做准备

这部分主要通过 JSPatch.js 为 JSContext 提供 JS 与 OC 交互的 bridge，这样 JS 和 OC 之间即可进行通信。当补丁下发后，JSPatch 会把补丁进行格式化，补丁中含有方法 defineClass ，这个方法会告诉 JSPatch 哪些类的哪些方法需要修改替换，然后利用 OC 的 runtime 对类进行处理。

第二部分：补丁执行

补丁中某个方法被执行时，它会通过 OC 的消息转发到 `JPForwardInvocation` 方法中，通过 `JPForwardInvocation` 中的 NSInvocation 获取方法调用时的参数值，根据方法名找到补丁中的实现，调用 JS 方法， JS 方法通过 `__c` 函数处理后调用不同的 OC 方法。

关于我们

> 知识小集是一个团队公众号，主要定位在移动开发领域，分享移动开发技术，包括 iOS、Android、小程序、移动前端、React Native、weex 等。每天都会有 **原创** 文章分享，我们的文章都会在公众号首发。

目前我们维护了几个微信群（iOS，flutter，微信小程序，前端），群里气氛还不错，如果想加入可以关注公众号后台留言即可获取加群方式。

我们经常会整理一篇 iOS 相关的小知识发到微博话题 `#ios知识小集` ，如果你想查看我们每天发的内容可以前往我们的微信小程序或者 GitHub

本文由知识小集所有，转载注明出处。

写在最后

本专题在书写过程中难免会有未考虑到的地方，如果你对某些地方有疑惑，可以联系作者 Lefe_x 和 halohily，或者通过 `知识小集` 公众号留言。**相关 demo 在 GitHub 上可以找到。**

致谢

本专题吸取了其它优秀作者的文章，特别感谢 bang 关于 JSPatch 原理的文章，味精 关于 Hybrid 方案的文章，对我帮助很大。

- JSPatch 实现原理详解
- 从零收拾一个hybrid框架（二）-- WebView容器基础功能设计思路
- 从零收拾一个hybrid框架（一）-- 从选择JS通信方案开始
- 廖雪峰
- iOS 中的 JS
- javascriptcore
- javascriptcore全面解析
- Webkit
- JSPatch