JSPatch技术原理分享

交付实施组（施小辉）

JSPatch技术内容

- JSPatch的技术支撑（iOS的Runtime机制）
- JSPatch的基础技术介绍
- JSPatch的JS与OC之间通信机制
- JSPatch的extension运用机制
- JSPatch的load机制
- 如何发布JS的hotfix的版本

Runtime——class

super_class:父类

name:类名

version:类的版本信息，默认为0

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class super_class                                       OBJC2_UNAVAILABLE;
    const char *name                                        OBJC2_UNAVAILABLE;
    long version                                            OBJC2_UNAVAILABLE;
    long info                                               OBJC2_UNAVAILABLE;
    long instance_size                                      OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                            OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists                   OBJC2_UNAVAILABLE;
    struct objc_cache *cache                                OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols                    OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
```

info:类信息，供运行期使用的一些标示位

instance_size:类的实例变量大小

methodLists:方法定义的列表

cache:方法缓存

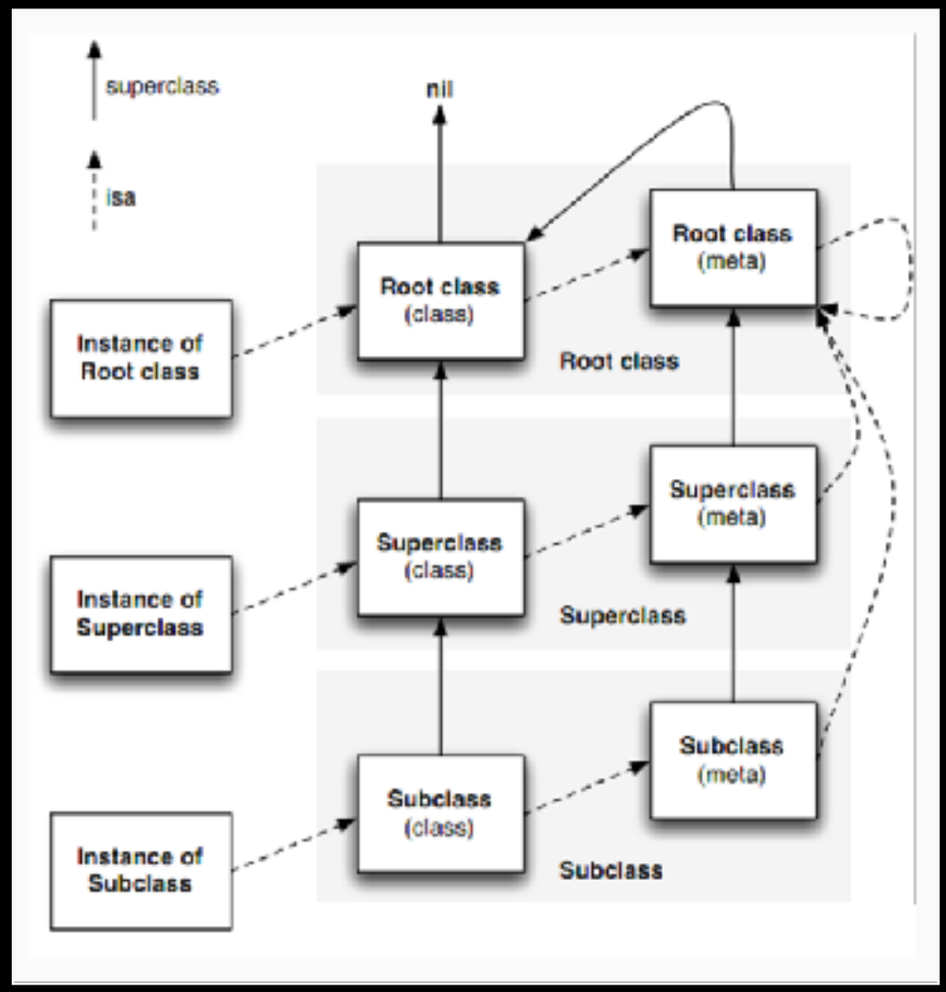
ivars:类的成员变量链表

protocols:协议链表

Runtime——类结构链表

isa：实例对象->类->（不经过父元类）直接到根元类（NSObject的元类），根元类的isa指向自己

superclass：类->父类->…->根类NSObject，元类->父元类->…->根元类->根类，NSObject的superclass指向nil



实例对象的isa指针指向类，类的isa指针指向其元类（metaClass），对象就是一个含isa指针的结构体。类存储实例对象的方法列表，元类存储类的方法列表，元类也是类对象。

Runtime——类的本质

class_相关	
isa	元类
superClass	父类
get	类名，父类；实例变量，成员变量；属性；实例方法，类方法，方法实现
copy	成员变量列表；属性列表；方法列表；协议列表；
add	成员变量；属性；方法；协议；
replace	属性；方法
respond	响应方法判断（内省）
isMetaclass	元类判断（内省）
conform	遵循协议判断（内省）

get:类名，父类，元类；实例变量，成员变量；属性；实例方法，类方法，方法实现；

```
// 获取类的类名
const char * class_getName (Class cls);
// 获取类的父类
Class class_getSuperclass (Class cls);
// 获取实例大小
size_t class_getInstanceSize (Class cls);
// 获取类中指定名称实例成员变量的信息
Ivar class_getInstanceVariable (Class cls, const char *name);
// 获取类成员变量的信息
Ivar class_getClassVariable (Class cls, const char *name);
// 获取指定的属性
objc_property_t class_getProperty (Class cls, const char *name);
// 获取实例方法
Method class_getInstanceMethod (Class cls, SEL name);
// 获取类方法
Method class_getClassMethod (Class cls, SEL name);
// 获取方法的具体实现
IMP class_getMethodImplementation (Class cls, SEL name);
IMP class_getMethodImplementation_stret (Class cls, SEL name);
```

copy: 成员变量列表；属性列表；方法列表；协议列表；

```
// 获取整个成员变量列表
Ivar * class_copyIvarList (Class cls, unsigned int *outCount);
// 获取属性列表
objc_property_t * class_copyPropertyList (Class cls, unsigned int *outCount);
// 获取所有方法的列表
Method * class_copyMethodList (Class cls, unsigned int *outCount);
// 获取类实现的协议列表
Protocol * class_copyProtocolList (Class cls, unsigned int *outCount);
```

add: 成员变量；属性；方法；协议；(添加成员变量只能在运行时创建的类，且不能为元类)

```
// 添加成员变量
BOOL class_addIvar (Class cls, const char *name, size_t size, uint8_t alignment, const char *types);
// 添加属性
BOOL class_addProperty (Class cls, const char *name,const objc_property_attribute_t *attributes, unsigned int
attributeCount);
// 添加方法
BOOL class_addMethod (Class cls, SEL name, IMP imp,const char *types);
// 添加协议
BOOL class_addProtocol (Class cls, Protocol *protocol);
```

replace：属性；方法；

```
// 替换类的属性
void class_replaceProperty (Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount);
// 替代方法的实现
IMP class_replaceMethod (Class cls, SEL name, IMP imp, const char *types);
```

respond:响应方法判断（内省）

```
// 类实例是否响应指定的selector
BOOL class_respondsToSelector (Class cls, SEL sel);
```

isMetaClass:元类判断（内省）

```
// 判断给定的Class是否是一个元类
BOOL class_isMetaClass (Class cls);
```

遵循协议判断（内省）

```
// 返回类是否实现指定的协议
BOOL class_conformsToProtocol (Class cls, Protocol *protocol);
```

Runtime——objc_

objc_相关	
get	实例变量；成员变量；类名；类；元类；关联对象；
copy	对象；类；类列表；协议列表；
set	实例变量；成员变量；类；类列表；协议；关联对象；
dispose	对象；
add	成员变量；属性；方法；协议；

get: 实例变量；成员变量；类名；类；元类；关联对象

```
// 获取对象实例变量
Ivar object_getInstanceVariable (id obj,const char *name, void **outValue);
// 获取对象中实例变量的值
id object_getIvar (id obj, Ivar ivar);
// 获取对象的类名
const char * object_getClassName (id obj);
// 获取对象的类
Class object_getClass (id obj );
Class objc_getClass (const char *name);
// 返回指定类的元类
Class objc_getMetaClass (const char *name);
//获取关联对象
id objc_getAssociatedObject(self, &myKey);
```

copy:对象；类；类列表；协议列表；

```
// 获取指定对象的一份拷贝
id object_copy (id obj, size_t size);
// 创建并返回一个指向所有已注册类的指针列表
Class * objc_copyClassList (unsigned int *outCount);
```

set: 实例变量；类；类列表；协议；关联对象；

```
// 设置类实例的实例变量的值
Ivar object_setInstanceVariable (id obj, const char *name, void *value);
// 设置对象中实例变量的值
void object_setIvar (id obj, Ivar ivar, id value);
//设置关联对象
void objc_setAssociatedObject(self, &myKey, anObject,
OBJC_ASSOCIATION_RETAIN);
```

dispose: 对象；

```
// 释放指定对象占用的内存
id object_dispose ( id obj );
```

动态创建/销毁类：

```
// 创建一个新类和元类
Class objc_allocateClassPair (Class superclass, const char *name, size_t
extraBytes);
// 销毁一个类及其相关联的类
void objc_disposeClassPair (Class cls);
// 在应用中注册由objc_allocateClassPair创建的类
void objc_registerClassPair (Class cls);
```

动态创建/销毁对象：

```
// 创建类实例
id class_createInstance (Class cls, size_t extraBytes);
// 在指定位置创建类实例
id objc_constructInstance (Class cls, void *bytes);
// 销毁类实例
void * objc_destructInstance (id obj );
```

Runtime——实例变量属性

/*数据类型 -- Ivar*/

```
typedef struct objc_ivar *Ivar;
struct objc_ivar {
    char *ivar_name OBJC2_UNAVAILABLE;  // 变量名
    char *ivar_type OBJC2_UNAVAILABLE;  // 变量类型
    int ivar_offset OBJC2_UNAVAILABLE;  // 基地址偏移字节
#ifdef __LP64__
    int space OBJC2_UNAVAILABLE;
#endif
}
```

```
typedef struct {
    const char *name; // 特性名
    const char *value; // 特性值
} objc_property_attribute_t;
```

/*objc_property_attribute_t（属性的特性有：返回值、是否为atomic、getter/setter名字、是否为dynamic、背后使用的ivar名字、是否为弱引用等）*/

Runtime—实例变量属性

ivar_get:

```
// 获取成员变量名
const char * ivar_getName(Ivar v);
// 获取成员变量类型编码
const char * ivar_getTypeEncoding(Ivar v);
// 获取成员变量的偏移量
ptrdiff_t ivar_getOffset (Ivar v);
```

property_:

```
// 获取属性名
const char * property_getName (objc_property_t property);
// 获取属性特性描述字符串
const char * property_getAttributes (objc_property_t property);
// 获取属性中指定的特性
char * property_copyAttributeValue (objc_property_t property,const char *attributeName);
// 获取属性的特性列表
objc_property_attribute_t * property_copyAttributeList (objc_property_t property, unsigned int *outCount);
```

Runtime——方法消息

SEL：

```
/*
SEL又叫选择器，是表示一个方法的selector的指针,映射方法的名字。Objective-C在编译时，会依据每一个方法的名字、参数序列，生成一个唯一的整型标识(Int类型的地址)，这个标识就是SEL。
SEL的作用是作为IMP的KEY，存储在NSSet中，便于hash快速查询方法。SEL不能相同，对应方法可以不同。所以在Objective-C同一个类(及类的继承体系)中，不能存在2个同名的方法，就算参数类型不同。多个方法可以有同一个SEL。
 不同的类可以有相同的方法名。不同类的实例对象执行相同的selector时，会在各自的方法列表中去根据selector去寻找自己对应的IMP。
 */
```

相关概念：类型编码（Type Encoding）

```
/*
编译器将每个方法的返回值和参数类型编码为一个字符串，并将其与方法的selector关联在一起。可以使用@encode编译器指令来获取它。
中没有公开具体的objc_selector结构体成员。但通过log可知SEL本质是一个字符串。
 */


 typedef struct objc_selector *SEL;
```

Runtime——方法消息

IMP:

```
/*
IMP是指向实现函数的指针，通过SEL取得IMP后，我们就获得了最终要找的实现函数的入口。
 */

 typedefine id (*IMP)(id, SEL, ...)
```

Method:

```
/*
这个结构体相当于在SEL和IMP之间作了一个绑定。这样有了SEL，我们便可以找到对应的IMP，从而调用方法的实现代码。（在运行时才将SEL和IMP绑定, 动态配置方法）
 */ typedef struct objc_method *Method;
    struct objc_method {
        SEL method_name                 OBJC2_UNAVAILABLE;  // 方法名
        char *method_types                  OBJC2_UNAVAILABLE; // 参数类型
        IMP method_imp                      OBJC2_UNAVAILABLE;  // 方法实现
    }
/*
objc_method_list 就是用来存储当前类的方法链表，objc_method存储了类的某个方法的信息。
 */
    struct objc_method_list {
        struct objc_method_list *obsolete  OBJC2_UNAVAILABLE;
        int method_count                                        OBJC2_UNAVAILABLE;
    #ifdef __LP64__
        int space                                                   OBJC2_UNAVAILABLE;
    #endif
        /* variable length structure */
        struct objc_method method_list[1]   OBJC2_UNAVAILABLE;
    }
```

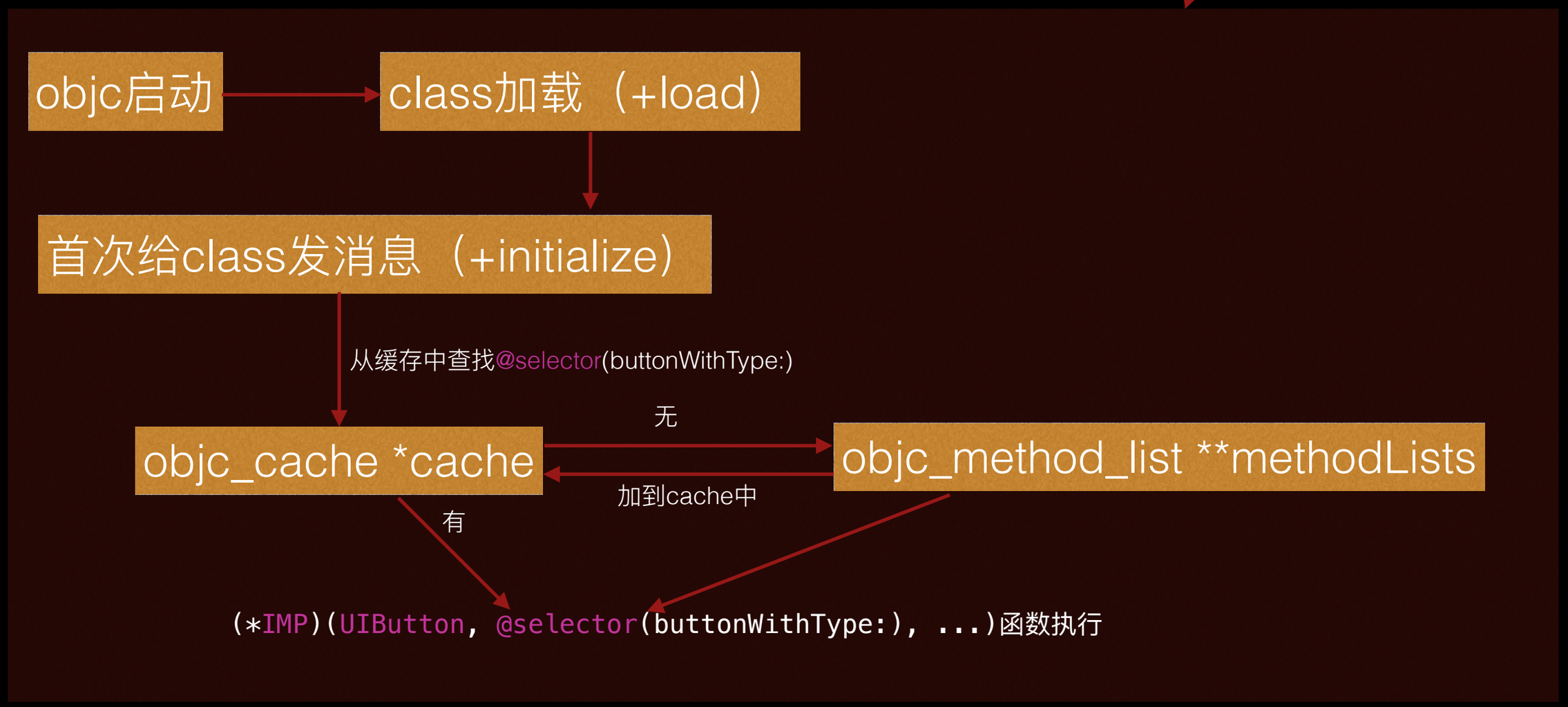
Runtime——方法消息

方法调用：

```
[UIButton buttonWithType:UIButtonTypeCustom]
```

```
/*
IMP是指向实现函数的指针，通过SEL取得IMP后，我们就获得了最终要找的实现函数的入口。
 */

 typedefine id (*IMP)(id, SEL, ...)
```

类方法调用过程



Runtime——方法消息

method_操作函数：

```
/*
 invoke: 方法实现的返回值
 */
        // 调用指定方法的实现
        id method_invoke (id receiver, Method m, ...);
        // 调用返回一个数据结构的方法的实现
        void method_invoke_stret (id receiver, Method m, ...);

 /*
get: 方法名；方法实现；参数与返回值相关
*/
        // 获取方法名
        SEL method_getName (Method m);
        // 返回方法的实现
        IMP method_getImplementation (Method m);
        // 获取描述方法参数和返回值类型的字符串
        const char * method_getTypeEncoding (Method m);
        // 返回方法的参数的个数
        unsigned int method_getNumberOfArguments (Method m);
        // 通过引用返回方法指定位置参数的类型字符串
        void method_getArgumentType (Method m, unsigned int index, char *dst, size_t dst_len);
 /*
  copy: 返回值类型，参数类型
  */
        // 获取方法的返回值类型的字符串
        char * method_copyReturnType (Method m);
        // 获取方法的指定位置参数的类型字符串
        char * method_copyArgumentType (Method m, unsigned int index);
        // 通过引用返回方法的返回值类型字符串
        void method_getReturnType (Method m, char *dst, size_t dst_len);
```

Runtime——方法消息

method_操作函数:

```
/*
 set: 方法实现
 */

            // 设置方法的实现
            IMP method_setImplementation ( Method m, IMP imp );


 /*
 exchange: 交换方法实现
 */


            // 交换两个方法的实现
            void method_exchangeImplementations ( Method m1, Method m2 );


 /*
  description : 方法描述
  */
            // 获取方法的返回值类型的字符串
            char * method_copyReturnType (Method m);
            // 获取方法的指定位置参数的类型字符串
            char * method_copyArgumentType (Method m, unsigned int index);
            // 通过引用返回方法的返回值类型字符串
            void method_getReturnType (Method m, char *dst, size_t dst_len);
```

Runtime——方法消息

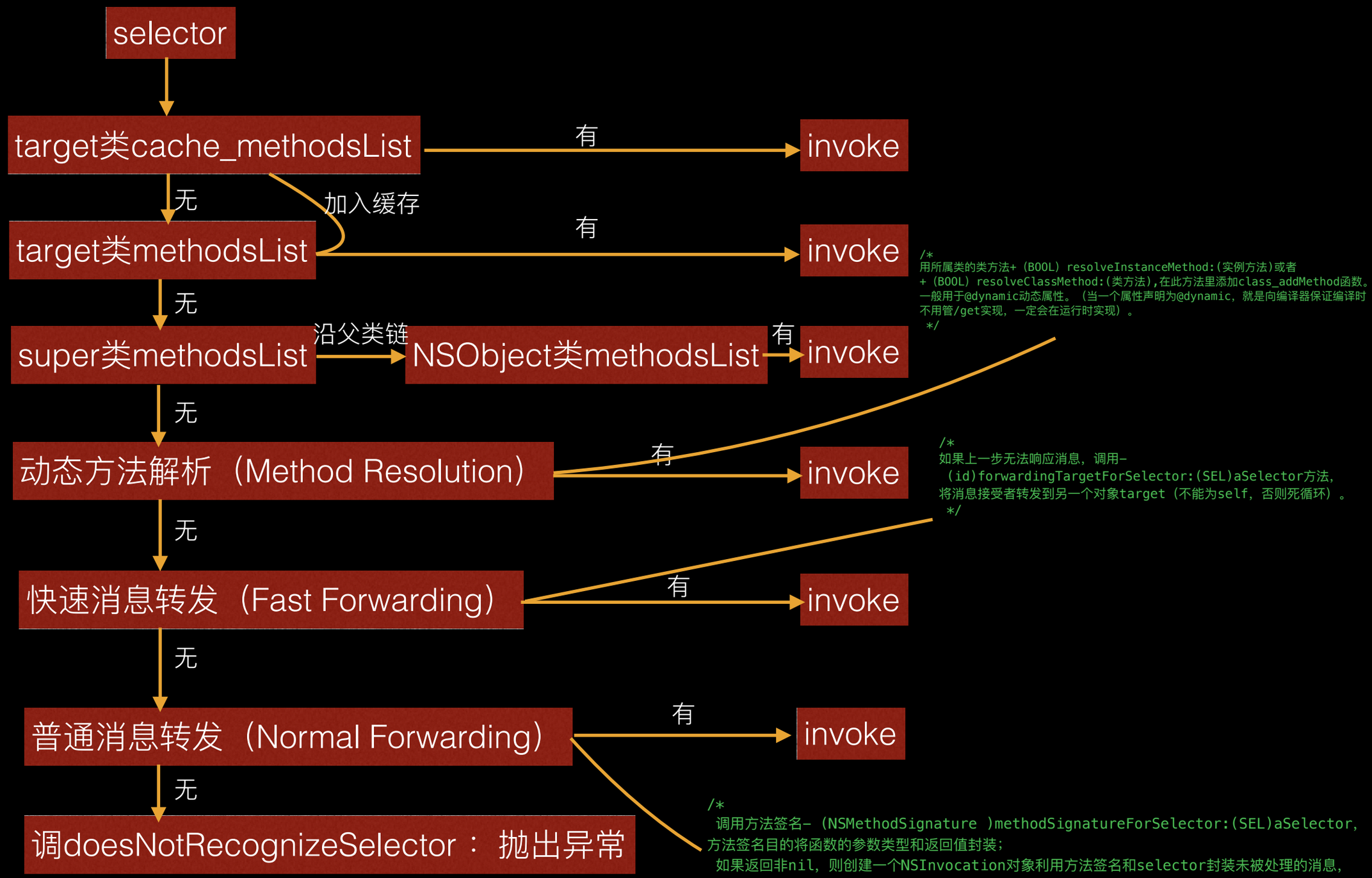
sel_操作函数：

```
// 返回给定选择器指定的方法的名称
const char * sel_getName ( SEL sel );
// 在Objective-C Runtime系统中注册一个方法，将方法名映射到一个选择器，并返回这个选择器
SEL sel_registerName ( const char *str );
// 在Objective-C Runtime系统中注册一个方法
SEL sel_getUid ( const char *str );
// 比较两个选择器
BOOL sel_isEqual ( SEL lhs, SEL rhs );
```

方法调用入口：

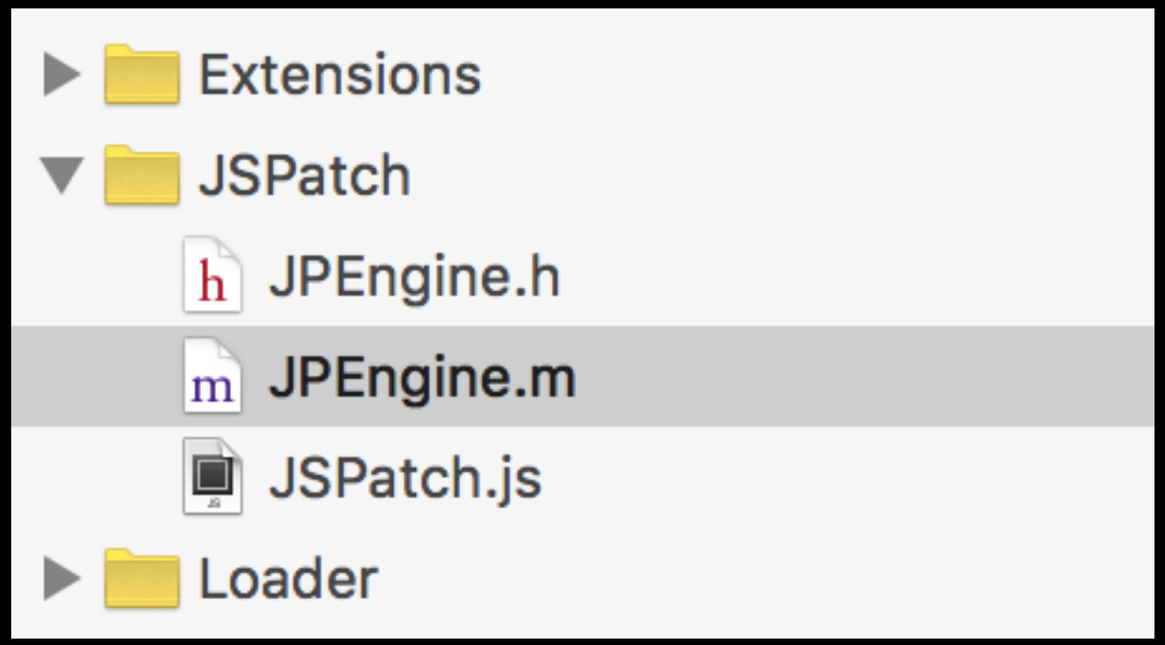
```
/*
 *  (1)向对象发送消息，实际上是调用objc_msgSend函数，obj_msgSend的实际动作就是：找到这个函数指针，然后调用它。
 *
 *  (2)self和_cmd是隐藏参数，在编译期被插入实现代码。
 *
 *  (3)self：指向消息的接受者target的对象类型，作为一个占位参数，消息传递成功后self将指向消息的receiver。
 *  (4)_cmd：指向方法实现的SEL类型。
 *
 *  (5)当向一般对象发送消息时，调用objc_msgSend；当向super发送消息时，调用的是objc_msgSendSuper；  如果返回值是一个结构体，则会调用objc_msgSend_stret或
objc_msgSendSuper_stret
 */
   id objc_msgSend(receiver self, selector _cmd, arg1, arg2, ...)


/*****************************************************objc_msgSend执行流程*****************************************************/
/*
    (1)检查target是否为nil。如果为nil，直接cleanup，然后return。(这就是我们可以向nil发送消息的原因。)如果方法返回值是一个对象，那么发送给nil的消息将返回nil；如果方法返回值为指针类型，其指针大小为小于或者等于sizeof(void*)，float，double，long double 或者long long的整型标量，发送给nil的消息将返回0；如果方法返回值为结构体,发送给nil的消息将返回0。结构体中各个字段的值将都是0；如果方法的返回值不是上述提到的几种情况，那么发送给nil的消息的返回值将是未定义的。
    (2)如果target非nil，在target的Class中根据Selector去找IMP。（因为同一个方法可能在不同的类中有不同的实现，所以我们需要依赖于接收者的类来找到的确切的实现）。
 */
/*****************************************************************************************************************************/
```

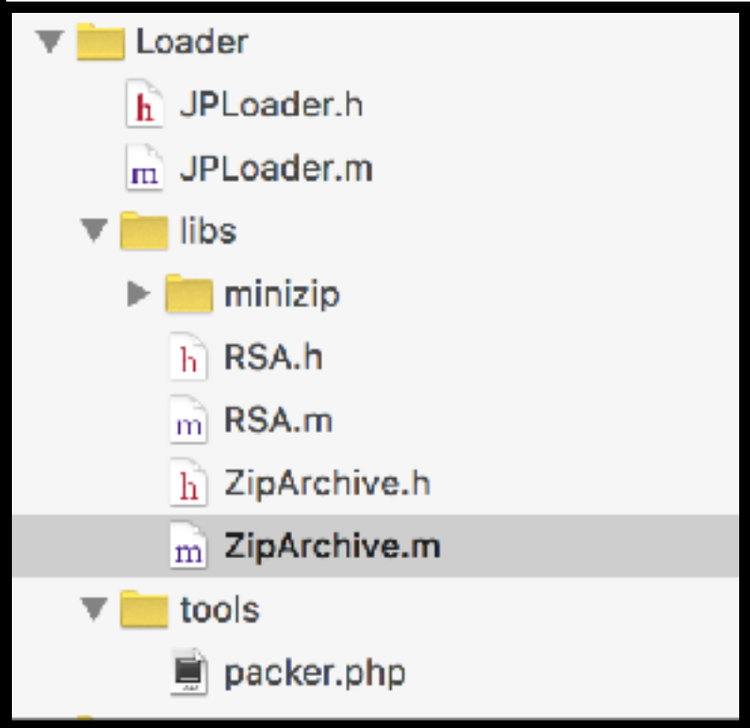
Runtime——方法消息

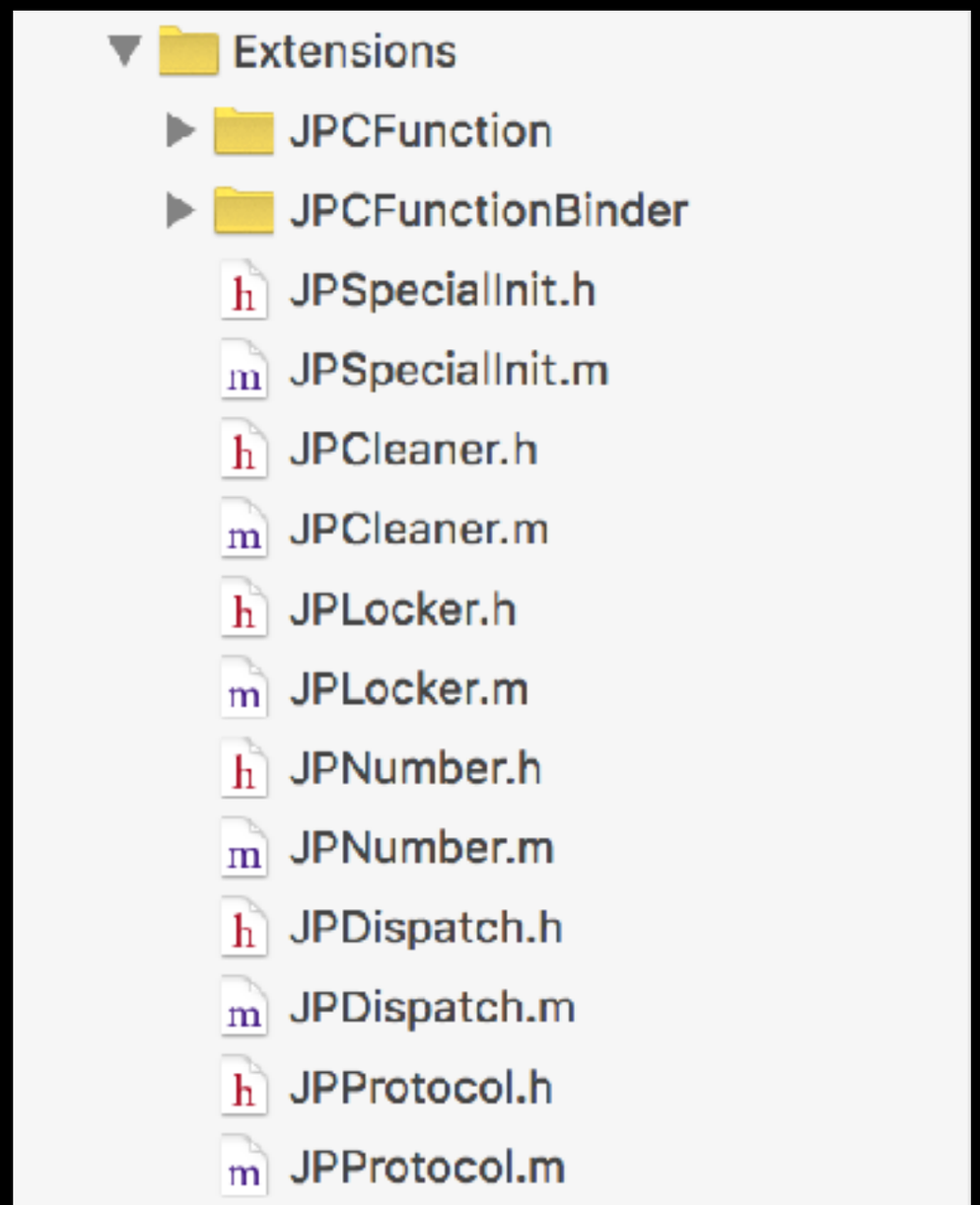


JSPatch基础技术介绍

JSPatch代码组织结构







JSPatch基础技术介绍

怎样根据OC代码写JS代码

F项目中的OC代码

```objc
//将rootviewcontroller设置为tabbarcontroller
- (void)openRootViewController {
    [self willOpenRootViewController];

    //强制保留tabbar模块
    [APP_ModuleManager requireModuleWithId:kPAFFTabbarModuleId];
    NSError *tabbarError = nil;
    _rootViewController = [APP_ModuleManager
tabbarViewControllerWithErro:&tabbarError];
    AppDelegate *delegete = [[UIApplication sharedApplication] delegate];
    delegete.window.rootViewController = _rootViewController;

    self.navigationController.delegate = self;
}
```

JSConventor

JS代码

```javascript
require("PAFFModuleManager, PAFFModuleManager, UIAp");

defineClass("PAFFNavigator", {
            openRootViewController: function() {
            self.willOpenRootViewController();
            PAFFModuleManager.shareInstance().requireModuleWithId("PAFFRoot");
            _rootViewController =
PAFFModuleManager.shareInstance().tabbarViewControllerWithErro(null);
            var delegete = UIApplication.sharedApplication().delegate();
            delegete.window().setRootViewController(_rootViewController);
            self.navigationController().setDelegate(self);
            }
            }, {});
```

```objc
AppDelegate *delegete = [[UIApplication sharedApplication] delegate];
```

JSPatch如何映射OC方法

```javascript
var delegete = UIApplication.sharedApplication().delegate();
```

JSPatch基础技术介绍

JS代码调用OC代码原理

JS调OC的原理：JS传递字符串给OC，OC通过 Runtime 接口调用和替换OC方法

```
AppDelegate *delegete = [[UIApplication sharedApplication] delegate];
```

_OC_callC

```
context[@"_OC_callC"] = ^id(NSString *className, NSString *selectorName, JSValue *arguments) {
        return callSelector(className, selectorName, arguments, nil, NO);
    };
```

```
context[@"_OC_callI"] = ^id(JSValue *obj, NSString *selectorName, JSValue *arguments, BOOL isSuper) {
        return callSelector(nil, selectorName, arguments, obj, isSuper);
    };
```

_OC_callI

JSPatch.js中的_methodFunc(var string)方法

JSPatch.js中的__c(var string)方法

UIApplication.__C(“sharedApplication”).__C(“delegate”)

formateJS

```
{
   __clsName:”UIApplication”
}
```

调用require(‘UIApplication’)后就可以直接去调用相应的类方法，require 做的事很简单，就是在JS全局作用域上创建一个同名变量，变量指向一个对象，对象属性__isCls表明这是一个 Class，__clsName保存类名，在调用方法时会用到这两个属性。

JSPatch如何映射OC方法

```
var delegete = UIApplication.sharedApplication().delegate();
```

JSPatch基础技术介绍

JS与OC之间类型映射关系

```
 <pre>
@textblock
   Objective-C type   |   JavaScript 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)
@/textblock
</pre>

(1) Instances of NSBlock with supported arguments types will be presented to
JavaScript as a callable Function object. For more information on supported
argument types see JSExport.h. If a JavaScript Function originating from an
Objective-C block is converted back to an Objective-C object the block will
be returned. All other JavaScript functions will be converted in the same
manner as a JavaScript object of type Object.

(2) For Objective-C instances that do not derive from the set of types listed
above, a wrapper object to provide a retaining handle to the Objective-C
instance from JavaScript. For more information on these wrapper objects, see
JSExport.h. When a JavaScript wrapper object is converted back to Objective-C
the Objective-C instance being retained by the wrapper is returned.

(3) For Objective-C Class objects a constructor object containing exported
class methods will be returned. See JSExport.h for more information on
constructor objects.
```

JSPatch中JS与OC通信机制

JSContext中定义JS调用的OC方法

```
//定义添加实现类的实例方法和类方法
    context[@"_OC_defineClass"] = ^(NSString *classDeclaration, JSValue *instanceMethods, JSValue *classMethods) {
        return defineClass(classDeclaration, instanceMethods, classMethods);
    };


//定义添加协议方法
   context[@"_OC_defineProtocol"] = ^(NSString *protocolDeclaration, JSValue *instProtocol, JSValue *clsProtocol) {
        return defineProtocol(protocolDeclaration, instProtocol,clsProtocol);
    };


//调用OC类的实例方法
    context[@"_OC_callI"] = ^id(JSValue *obj, NSString *selectorName, JSValue *arguments, BOOL isSuper) {
        return callSelector(nil, selectorName, arguments, obj, isSuper);
    };



//调用类的类方法
   context[@"_OC_callC"] = ^id(NSString *className, NSString *selectorName, JSValue *arguments) {
        return callSelector(className, selectorName, arguments, nil, NO);
   };
```

JSPatch中JS与OC通信机制

JSContext中定义JS调用的OC方法

```
//将JS对象转换成OC对象
    context[@"_OC_formatJSToOC"] = ^id(JSValue *obj) {
        return formatJSToOC(obj);
    };


//将OC对象转换成JS对象
   context[@"_OC_formatOCToJS"] = ^id(JSValue *obj) {
        return formatOCToJS([obj toObject]);
    };


 //定义属性get方法
    context[@"_OC_getCustomProps"] = ^id(JSValue *obj) {
        id realObj = formatJSToOC(obj);
        return objc_getAssociatedObject(realObj, kPropAssociatedObjectKey);
    };



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

JSPatch中JS与OC通信机制

JSContext中定义JS调用的OC方法

```
//JS弱引用OC对象
    context[@"__weak"] = ^id(JSValue *jsval) {
        id obj = formatJSToOC(jsval);
        return [[JSContext currentContext][@"_formatOCToJS"] callWithArguments:@[formatOCToJS([JPBoxing
boxWeakObj:obj])]];
    };

//JS强引用OC对象
   context[@"__strong"] = ^id(JSValue *jsval) {
        id obj = formatJSToOC(jsval);
        return [[JSContext currentContext][@"_formatOCToJS"] callWithArguments:@[formatOCToJS(obj)]];
    };


 //得到一个类的父类名称
    context[@"_OC_superClsName"] = ^(NSString *clsName) {
        Class cls = NSClassFromString(clsName);
        return NSStringFromClass([cls superclass]);
    };
//将一个JS文件包含到JSPatch引擎中去
   context[@"include"] = ^(NSString *filePath) {
       NSString *absolutePath = [_scriptRootDir stringByAppendingPathComponent:filePath];
       if (!_runnedScript) {
           _runnedScript = [[NSMutableSet alloc] init];
       }
       if (absolutePath && ![_runnedScript containsObject:absolutePath]) {
           [JPEngine _evaluateScriptWithPath:absolutePath];
           [_runnedScript addObject:absolutePath];
       }
   };
```

JSPatch中JS与OC通信机制

OC类和对象在JS中的存在形式

```
Class UIViewController
```

```
{
  __clsName: “UIViewController”,
  __c: function(methodName){…},

  super: function(){…},
  performSelectorInOC: function(){…},
  performSelector: function(){…},
  …
}
```

```
UIViewController* vc
```

```
{
  __obj: vc,
  __clsName: “UIViewController”,
  __c: function(methodName){…},

  super: function(){…},
  performSelectorInOC: function(){…},
  performSelector: function(){…},
  …

}
```

JSPatch中extension运用机制

JPExtension的类说明

```
//JPExtension的入口方法，当添加JPExtension到JSContext中时会自动执行该方法
+ (void)main:(JSContext *)context;

//将JS中的指针转换成OC中的对象指针
+ (void *)formatPointerJSToOC:(JSValue *)val;

//将CoreFoundation中的指针转换成JS中的指针，通过JPBoxing
+ (id)formatRetainedCFTypeOCToJS:(CFTypeRef)CF_CONSUMED type;

//将OC中的指针类型转换成JS中的指针类型
+ (id)formatPointerOCToJS:(void *)pointer;

//将JS对象转换成OC对象
+ (id)formatJSToOC:(JSValue *)val;

//将OC对象转换成JS对象
+ (id)formatOCToJS:(id)obj;

//计算结构体的大小
+ (int)sizeOfStructTypes:(NSString *)structTypes;
```

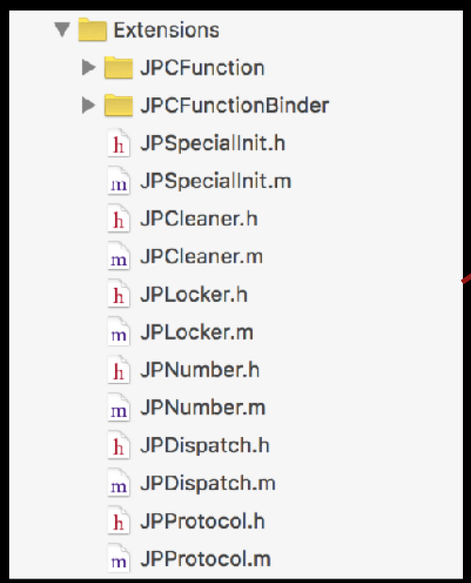
JSPatch中extension运用机制

JPExtension的类说明

```
//从字典中读取结构体并存储在起始地址为structData的内存块中
//structDefine的结构为
    /* @{
        @"name": @"CGAffineTransform",   //struct name
        @"types": @"ffffff",  //struct types
        @"keys": @[@"a", @"b", @"c", @"d", @"tx", @"ty"]  //struct keys in JS
     }*/

+ (void)getStructDataWidthDict:(void *)structData dict:(NSDictionary *)dict structDefine:(NSDictionary
*)structDefine;
//从起始地址为structData的内存块中读取结构体数据到字典中并返回
+ (NSDictionary *)getDictOfStruct:(void *)structData structDefine:(NSDictionary *)structDefine;

/*!
 @method
 @description Return the registered struct definition in JSPatch,
 the key of dictionary is the struct name.
 */
//当前在JSContext中注册的结构体
+ (NSMutableDictionary *)registeredStruct;
//当前在JS中重载过的类方法
+ (NSDictionary *)overideMethods;
//当前在JSContext中记载的js文件集
+ (NSMutableSet *)includedScriptPaths;
```

JSPatch中extension运用机制

如何使用JPExtension




```
[JPEngine addExtensions:@[@"JPNumber", @"JPProtocol"]];
```

```
+ (void)addExtensions:(NSArray *)extensions
{
    if (![JSContext class]) {
        return;
    }
    if (!_context) _exceptionBlock(@"please call [JPEngine startEngine]");
    for (NSString *className in extensions) {
        Class extCls = NSClassFromString(className);
        [extCls main:_context];
    }
}
```

```
@implementation JPNumber

+ (void)main:(JSContext *)context {…}
```

```
@implementation JPProtocol
+ (void)main:(JSContext *)context{…}
```

自定义extension =

```
#import "JPEngine.h"

@interface JPMyTestExtension : JPExtension

@end
```

+

```
#import "JPMyTestExtension.h"

@implementation JPMyTestExtension

+ (void)main:(JSContext *)context {
    //实现自己在JS中需要使用的功能
}

@end
```

如何发布JSPatch中hotfix版本

如何在F项目中本地调试JSPatch

(1) 在工程目录下创建加载js的入口文件main.js
(2) 在工程目录下创建需要修改的类文件对应的js文件（如testClass1.js，testClass2.js）
(3) 在main.js中把需要的js文件包含进去，包含用的代码如下include('testClass1.js'); include('testClass2.js');
(4) 在PAFFUnitUpdater.m文件中- (void)setUpHotAndRNWithLaunchOptions:(NSDictionary *)launchOptions函数体内加入如下一行代码:

```
- (void)setUpHotAndRNWithLaunchOptions:(NSDictionary *)launchOptions
{
    NSString *updateApiHost;
    NSString *updateRSAkey;

    if(APP_Context.isDebug && !_prdEnv){
        updateApiHost = kUpdateSTGHost;
        updateRSAkey = kUpdateHostSTGRSAKey;
    }else{
        updateApiHost = kUpdatePRDHost;
        updateRSAkey = kUpdateHostPRDRSAKey;
    }

     NSURL *jsCodeLocation = [NSURL URLWithString:APP_Config.rnBundlePath];
    [_updater setUpHotAndRCTUpdaterWithUpdateApiHost:updateApiHost
                                        updateAESKey:APP_Config.updateAESKey
                                        updateRSAkey:updateRSAkey
                                defaultJsCodeLocation:jsCodeLocation
                                              options:launchOptions];

    NSString *coreBundle = [[NSBundle mainBundle] pathForResource:@"core" ofType:@"jsbundle"];
    NSString *paffBunle = [[NSBundle mainBundle] pathForResource:@"paff" ofType:@"jsbundle"];
    HFInterface_RCTUpdater *rctUpdater = [_updater rctUpdater];
    if ([coreBundle length] > 0 && [paffBunle length] > 0) {
        NSArray *bundlePathArray = @[coreBundle, paffBunle];
        [rctUpdater bundleFile:bundlePathArray];
    }

    APP_Context.rctUpdater = rctUpdater;
    //JSPatch 补丁本地调试
    [[_updater hotUpdater] evaluateScriptWithPath:[[NSBundle mainBundle] pathForResource:@"main" ofType:@"js"]
        isLocalDebugMode:YES];
}
```