# JS Bridge 通信原理

## 前言

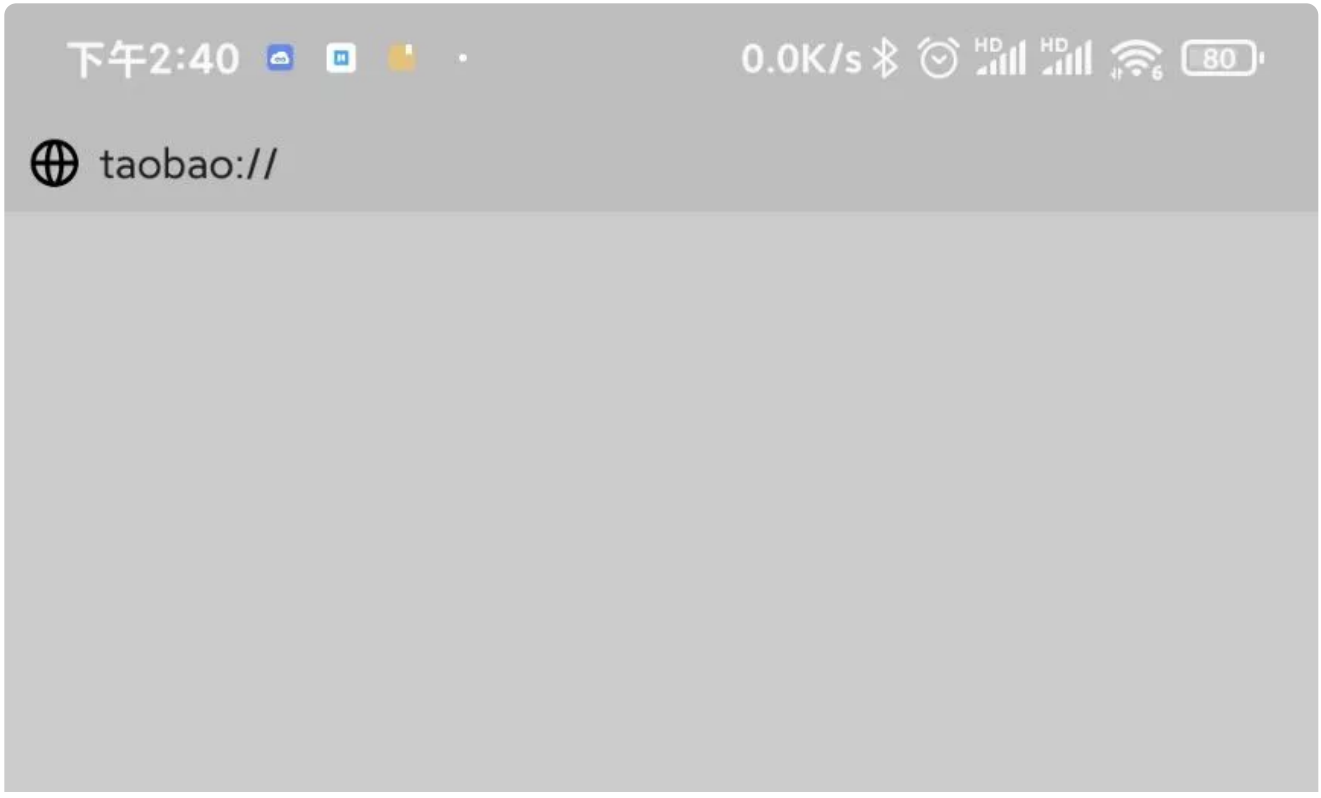
上一篇介绍了移动端开发的相关技术，这一篇主要是从 Hybrid 开发的 JS Bridge 通信讲起。

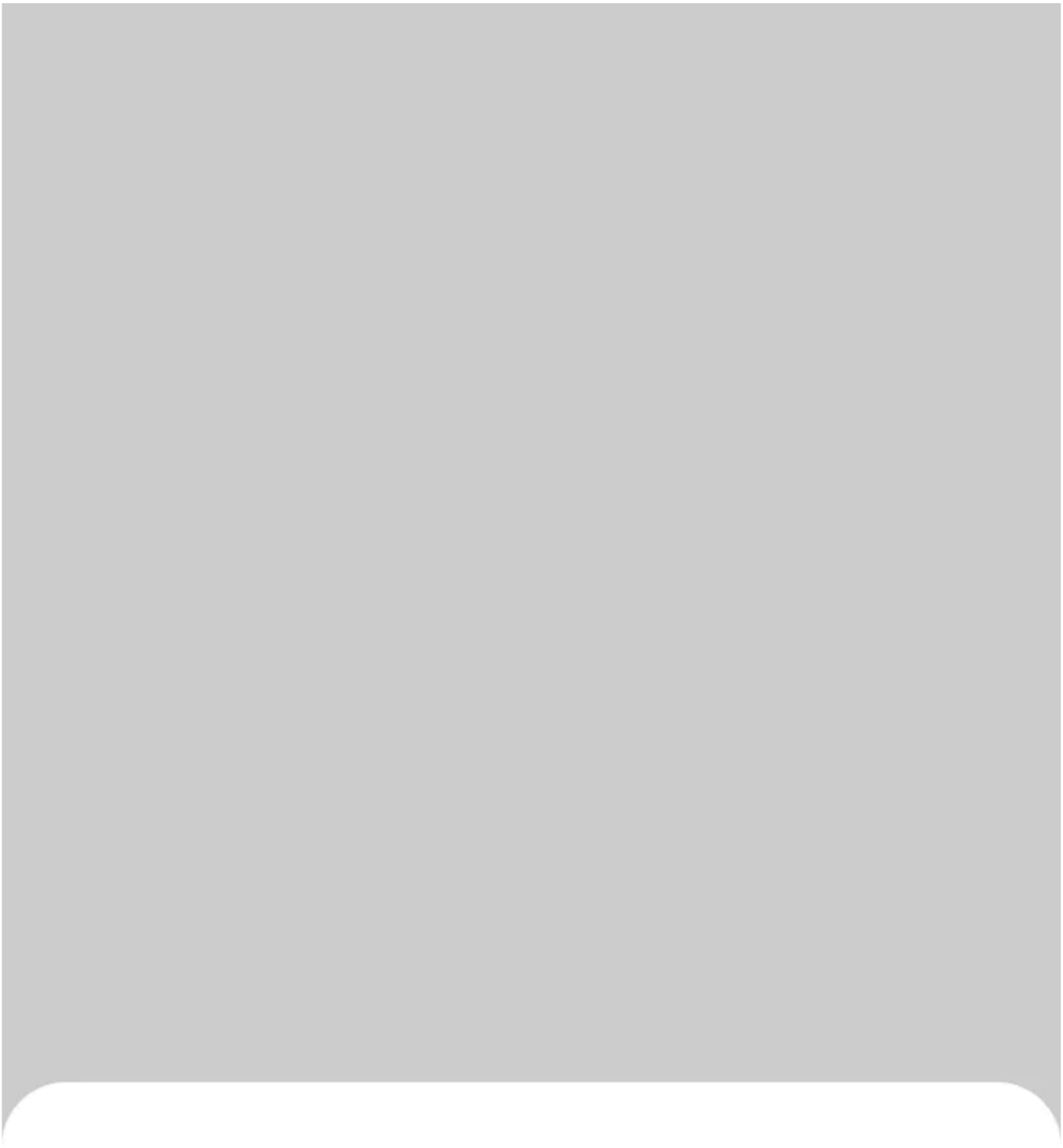
顾名思义，JS Bridge 的意思就是桥，也就是连接 JS 和 Native 的桥梁，它也是 Hybrid App 里面的核心。一般分为 JS 调用 Native 和 Native 主动调用 JS 两种形式。

## URL Scheme

URL Scheme 是一种特殊的 URL，一般用于在 Web 端唤醒 App，甚至跳转到 App 的某个页面，比如在某个手机网站上付款的时候，可以直接拉起支付宝支付页面。

这里有个小例子，你可以在浏览器里面直接输入 `weixin://`，系统就会提示你是否要打开微信。输入 `mqq://` 就会帮你唤起手机 QQ。





Firefox Lite 正在尝试开启 手机淘宝，是否允许?

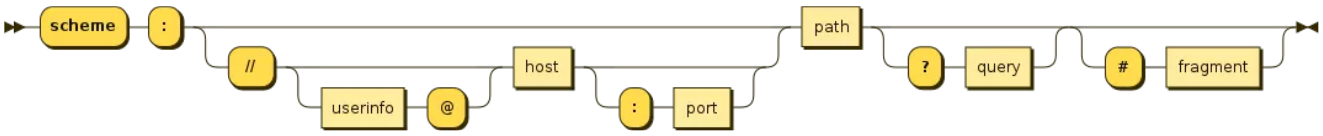
拒绝

允许

这里有个常用 App URL Scheme 汇总：URL Schemes 收集整理

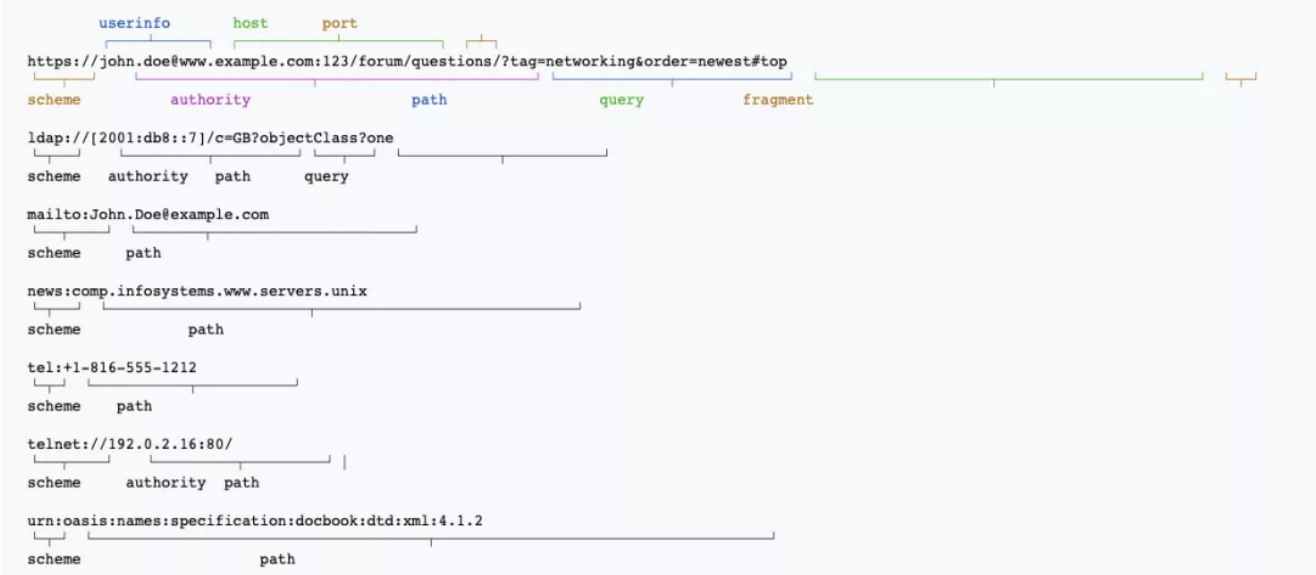
在手机里面打开这个页面后点击这里，就会提示你是否要打开微信。

我们常说的 Deeplink 一般也是基于 URL Scheme 来实现的。一个 URI 的组成结构如下：




```
URI = scheme:[//authority]path[?query][#fragment]
// scheme      = http
// authority   = www.baidu.com
// path        = /link
// query       = url=xxxxx
authority = [userinfo@]host[:port]
```

除了 http/https 这两个常见的协议，还可以自定义协议。借用维基百科的一张图：



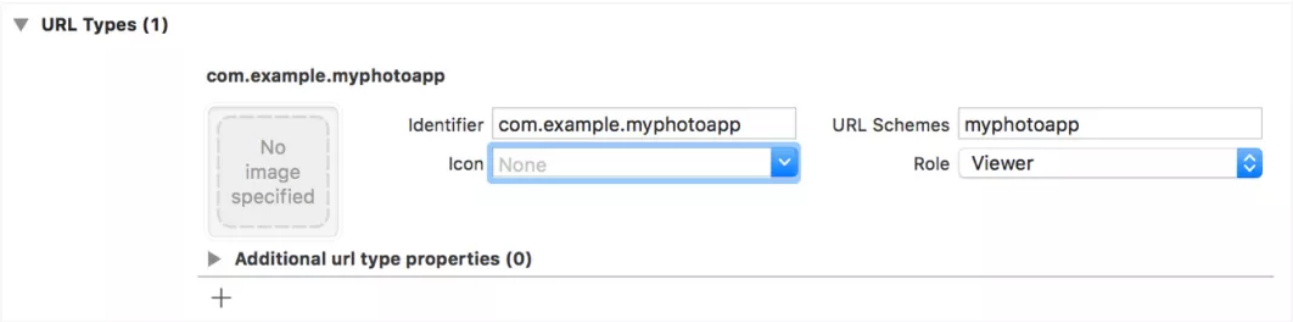
通常情况下，App 安装后会在手机系统上注册一个 Scheme，比如 `weixin://` 这种，所以我们在手机浏览器里面访问这个 scheme 地址，系统就会唤起我们的 App。

一般在 Android 里面需要到 AndroidManifest.xml 文件中去注册 Scheme：

```
<activity
    android:name=".login.dispatch.DispatchActivity"
    android:launchMode="singleTask"
    android:theme="@style/AppDispatchTheme">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="taobao" />
        <data android:host="xxx" />
        <data android:path="/goods" />
    </intent-filter>
</activity>
```

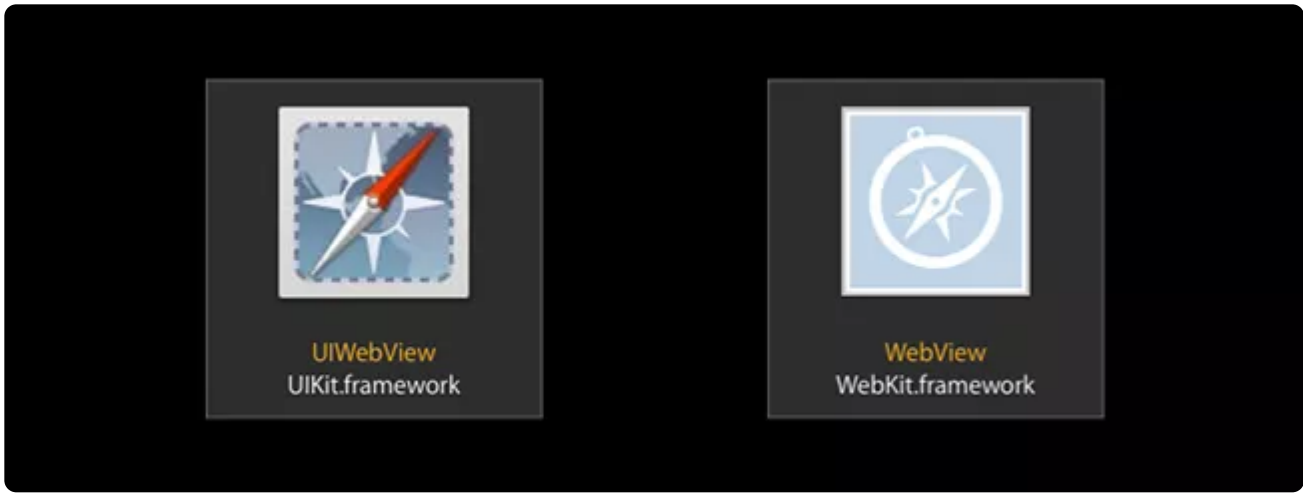
在 iOS 中需要在 Xcode 里面注册，有一些已经是系统使用的不应该使用，比如 Maps、YouTube、Music。

具体可以参考苹果开发者官网文档：Defining a Custom URL Scheme for Your App




## JS 调用 Native

在 iOS 里面又需要区分 UIWebView 和 WKWebView 两种 WebView：




WKWebView 是 iOS8 之后出现的，目的是取代笨重的 UIWebView，它占用内存更少，大概是 UIWebView 的 1/3，支持更好的 HTML5 特性，性能更加强大。

但也有一些缺点，比如不支持缓存，需要自己注入 Cookie，发送 POST 请求的时候带不了参数，拦截 POST 请求的时候无法解析参数等等。

**「题外话（小小吐槽一下）」**

我们这边有很多场景是需要客户端和银行交互的，为了从银行 H5 回调到我们的客户端，常常是会让银行跳到我们的 H5 中间页，客户端去拦截并解析银行带来的参数。

以前有些页面是 Python 服务端渲染做的，支持 POST 请求打开页面。后来 iOS 客户端升级 WKWebView，导致无法获取银行 POST 带过来的参数。

为了配合客户端发版上线，曾经连夜修改了我们 App 里面所有相关的绑卡和支付 H5 页面，还好没有搞出重大线上事故。

JS 调用 Native 通信大致有三种方法：

1. 拦截 Scheme
2. 弹窗拦截
3. 注入 JS 上下文

这三种方式总体上各有利弊，下面会一一介绍。

## 拦截 Scheme

仔细思考一下，如果是 JS 和 Java 之间传递数据，我们该怎么做呢？

对于前端开发来说，调 Ajax 请求接口是最常见的需求了。不管对方是 Java 还是 Python，我们都可以通过 http/https 接口来获取数据。实际上这个流程和 JSONP 更加类似。

已知客户端是可以拦截请求的，那么可不可以在这个上面做文章呢？

如果我们请求一个不存在的地址，上面带了一些参数，通过参数告诉客户端我们需要调用的功能呢？

比如我要调用扫码功能：

```
axios.get('http://xxxx?func=scan&callback_id=yyyy')
```

客户端可以拦截这个请求，去解析参数上面的 `func` 来判断当前需要调起哪个功能。客户端调起扫码功能之后，会获取 WebView 上面的 callbacks 对象，根据 callback_id 回调它。

所以基于上面的例子，我们可以把域名和路径当做通信标识，参数里面的 func 当做指令，callback_id 当做回调函数，其他参数当做数据传递。对于不满足条件的 http 请求不应该拦截。

当然了，现在主流的方式是前面我们看到的自定义 Scheme 协议，以这个为通信标识，域名和路径当做指令。

这种方式的好处就是 iOS6 以前只支持这种方式，兼容性比较好。

## JS 端

我们有很多种方法可以发起请求，目前使用最广泛的是 iframe 跳转：

1. 使用 a 标签跳转

```
<a href="taobao://">点击我打开淘宝</a>
```

2. 重定向

```
location.href = "taobao://"
```

3. iframe 跳转

```
const iframe = document.createElement("iframe");
iframe.src = "taobao://"
iframe.style.display = "none"
document.body.appendChild(iframe)
```

## Android 端

在 Android 侧可以用 `shouldOverrideUrlLoading` 来拦截 url 请求。

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if (url.startsWith("taobao")) {
        // 拿到调用路径后解析调用的指令和参数，根据这些去调用 Native 方法
        return true;
    }
}
```

## iOS 端

在 iOS 侧需要区分 UIWebView 和 WKWebView 两种方式。在 UIWebView 中：

```
- (BOOL)shouldStartLoadWithRequest:(NSURLRequest *)request
                    navigationType:(BPWebViewNavigationType)navigationType
{
```

```
    if (xxx) {
        // 拿到调用路径后解析调用的指令和参数，根据这些去调用 Native 方法
        return NO;
    }

    return [super shouldStartLoadWithRequest:request navigationType:navigat
}
```

在 WKWebView 中：

```
- (void)webView:(WKWebView *)webView decidePolicyForNavigationAction:(nonnu
{
    if(xxx) {
        // 拿到调用路径后解析调用的指令和参数，根据这些去调用 Native 方法
        BLOCK_EXEC(decisionHandler, WKNavigationActionPolicyCancel);
    } else {
        BLOCK_EXEC(decisionHandler, WKNavigationActionPolicyAllow);
    }

    [self.webView.URLLoader webView:webView decidedPolicy:policy forNavigat
}
```

目前不建议只使用拦截 URL Scheme 解析参数的形式，主要存在几个问题。

1. 连续续调用 `location.href` 会出现消息丢失，因为 WebView 限制了连续跳转，会过滤掉后续的请求。
2. URL 会有长度限制，一旦过长就会出现信息丢失 因此，类似 WebViewJavaScriptBridge 这类库，就结合了注入 API 的形式一起使用，这也是我们这边目前使用的方式，后面会介绍一下。

## 弹窗拦截

### Android 实现

这种方式是利用弹窗会触发 WebView 相应事件来拦截的。

一般是在 setWebChromeClient 里面的 onJsAlert 、 onJsConfirm 、 onJsPrompt 方法拦截并解析他们传来的消息。

```
// 拦截 Prompt
@Override
public boolean onJsPrompt(WebView view, String url, String message, String
        if (xxx) {
         // 解析 message 的值，调用对应方法
        }
        return super.onJsPrompt(view, url, message, defaultValue, result);
    }
// 拦截 Confirm
@Override
public boolean onJsConfirm(WebView view, String url, String message, JsResu
        return super.onJsConfirm(view, url, message, result);
    }
// 拦截 Alert
@Override
public boolean onJsAlert(WebView view, String url, String message, JsResult
        return super.onJsAlert(view, url, message, result);
    }
```

## iOS 实现

我们以 WKWebView 为例：

```
+ (void)webViewRunJavaScriptTextInputPanelWithPrompt:(NSString *)prompt
    defaultText:(NSString *)defaultText
    completionHandler:(void (^)(NSString * _Nullable))completionHandler
{
    /** Triggered by JS:
    var person = prompt("Please enter your name", "Harry Potter");
    if (person == null || person == "") {
        txt = "User cancelled the prompt.";
    } else {
        txt = "Hello " + person + "! How are you today?";
    }
    */
```

```
    */
    if (xxx) {
        BLOCK_EXEC(completionHandler, text);
    } else {
        BLOCK_EXEC(completionHandler, nil);
    }
}
```

这种方式的缺点就是在 iOS 上面 UIWebView 不支持，虽然 WKWebView 支持，但它又有更好的 `scriptMessageHandler`，比较尴尬。

### 注入上下文

前面我们有讲过在 iOS 中内置了 JavaScriptCore 这个框架，可以实现执行 JS 以及注入 Native 对象等功能。

这种方式不依赖拦截，主要是通过 WebView 向 JS 的上下文注入对象和方法，可以让 JS 直接调用原生。

PS：iOS 中的 Block 是 OC 对于闭包的实现，它本质上是个对象，定义 JS 里面的函数。

## iOS UIWebView

iOS 侧代码：

```
// 获取 JS 上下文
JSContext *context = [webview valueForKeyPath:@"documentView.webView.mainFr
// 注入 Block
context[@"callHandler"] = ^(JSValue * data) {
    // 处理调用方法和参数
    // 调用 Native 功能
    // 回调 JS Callback
}
```

JS 代码：

```
window.callHandler(JSON.stringify({
    type: "scan",
    data: "",
    callback: function(data) {
    }
}));
```

这种方式的牛逼之处在于，JS 调用是同步的，可以立马拿到返回值。

我们也不再需要像拦截方式一样，每次传值都要把对象做 `JSON.stringify`，可以直接传 JSON 过去，也支持直接传一个函数过去。

## iOS WKWebView

WKWebView 里面通过 `addScriptMessageHandler` 来注入对象到 JS 上下文，可以在 WebView 销毁的时候调用 `removeScriptMessageHandler` 来销毁这个对象。

前端调用注入的原生方法之后，可以通过 `didReceiveScriptMessage` 来接收前端传过来的参数。

```
WKWebView *wkWebView = [[WKWebView alloc] init];
WKWebViewConfiguration *configuration = wkWebView.configuration;
WKUserContentController *userCC = configuration.userContentController;

// 注入对象
[userCC addScriptMessageHandler:self name:@"nativeObj"];
// 清除对象
[userCC removeScriptMessageHandler:self name:@"nativeObj"];

// 客户端处理前端调用
- (void)userContentController:(WKUserContentController *)userContentControl
{
    // 获取前端传来的参数
    NSDictionary *msgBody = message.body;
    // 如果是 nativeObj 就进行相应处理
```

```
    if (![message.name isEqualToString:@"nativeObj"]) {
        //
        return;
    }
}
```

使用 addScriptMessageHandler 注入的对象实际上只有一个 postMessage 方法，无法调用更多自定义方法。前端的调用方式如下：

```
window.webkit.messageHandlers.nativeObj.postMessage(data);
```

需要注意的是，这种方式要求 iOS8 及以上，而且返回不是同步的。和 UIWebView 一样的是，也支持直接传 JSON 对象，不需要 stringify。

## Android addJavascriptInterface

安卓4.2之前注入 JS 一般是使用 addJavascriptInterface ，和前面的 addScriptMessageHandler 有一些类似，但又没有它的限制。

```
public void addJavascriptInterface() {
        mWebView.addJavascriptInterface(new DatePickerJSBridge(), "DatePick
    }
private class PickerJSBridge {
    public void _pick(...) {
    }
}
```

在 JS 里面调用：

```
window.DatePickerBridge._pick(...)
```

但这种方案有一定风险，可以参考这篇文章：WebView中接口隐患与手机挂马利用

在 Android4.2 之后提供了 `@JavascriptInterface` 注解，暴露给 JS 的方法必须要带上这个。

所以前面的 `_pick` 方法需要带上这个注解。

```
private class PickerJSBridge {
    @JavascriptInterface
    public void _pick(...) {
    }
}
```

## Native 调用 JS

Native 调用 JS 一般就是直接 JS 代码字符串，有些类似我们调用 JS 中的 `eval` 去执行一串代码。一般有 `loadUrl` 、 `evaluateJavascript` 等几种方法，这里逐一介绍。

但是不管哪种方式，客户端都只能拿到挂载到 `window` 对象上面的属性和方法。

### Android

在 Android 里面需要区分版本，在安卓4.4之前的版本支持 loadUrl，使用方式类似我们在 a 标签的 `href` 里面写 JS 脚本一样，都是 `javascript:xxx` 的形式。

这种方式无法直接获取返回值。

```
webView.loadUrl("javascript:foo()")
```

在安卓4.4以上的版本一般使用 `evaluateJavascript` 这个 API 来调用。这里需要判断一下版本。

```
if (Build.VERSION.SDK_INT > 19) //see what wrapper we have
{
    webView.evaluateJavascript("javascript:foo()", null);
} else {
```

```
} else {
    webView.loadUrl("javascript:foo()");
}
```

### UIWebView

在 iOS 的 UIWebView 里面使用 `stringByEvaluatingJavaScriptFromString` 来调用 JS 代码。这种方式是同步的，会阻塞线程。

```
results = [self.webView stringByEvaluatingJavaScriptFromString:"foo()"];
```

### WKWebView

WKWebView 可以使用 `evaluateJavaScript` 方法来调用 JS 代码。

```
[self.webView evaluateJavaScript:@"document.body.offsetHeight;" completionH
    // 获取返回值 response
    }];
```

## JS Bridge 设计

前面讲完了 JS 和 Native 互调的所有方法，这里来介绍一下我们这边 JS Bridge 的设计吧。

我们这边的 JS Bridge 通信是基于 WebViewJavascriptBridge 这个库来实现的。

主要是结合 Scheme 协议+上下文注入来做。考虑到 Android 和 iOS 不一样的通信方式，这里进行了封装，保证提供给外部的 API 一致。

具体功能的调用我们封装成了 npm 包，下面的是几个基础 API：

1. callHandler(name, params, callback)：这个是调用 Native 功能的方法，传模块名、参数、回调函数给 Native。

2. hasHandler(name)：这个是检查客户端是否支持某个功能的调用。
3. registerHandler(name)：这个是提前注册一个函数，等待 Native 回调，比如 `pageDidBack` 这种场景。

那么这几个 API 又是如何实现的呢？这里 Android 和 iOS 封装不一致，应当分开来说。

## Android Bridge

前面我们有说过安卓可以通过 `@JavascriptInterface` 注解来将对象和方法暴露给 JS。

所以这里的几个方法都是通过注解暴露给 JS 来调用的，在 JS 层面做了一些兼容处理。

### hasHandler

首先最简单的是这个 `hasHandler`，就是在客户端里面维护一张表（其实我们是写死的），里面有支持的 Bridge 模块信息，只需要用 `switch...case` 判断一下就行了。

```
@JavascriptInterface
public boolean hasHandler(String cmd) {
        switch (cmd) {
            case xxx:
            case yyy:
            case zzz:
                return true;
        }
        return false;
    }
```

### callHandler

然后我们来看 `callHandler` 这个方法，它是提供 JS 调用 Native 功能的方法。在调用这个方法之前，我们一般需要先判断一下 Native 是否支持这个功能。

```
function callHandler(name, params, callback) {
    if (!window.WebViewJavascriptBridge.hasHandler(name)) {
    }
}
```

如果 Native 没有支持这个 Bridge，我们就需要对回调进行兼容性处理。这个兼容性处理包括两个方面，一个是功能方面，一个是 callback 的默认回参。

比如我们调用 Native 的弹窗功能，如果客户端没支持这个 Bridge，或者我们是在浏览器里面打开的这个页面，此时应该降级到使用 Web 的 alert 弹窗。

对于 callback，我们可以默认给传个 0，表示当前不支持这个功能。

假设这个 alert 的 bridge 接收两个参数，分别是 title 和 content，那么此时就应该使用浏览器自带的 alert 展示出来。

```
function fallback(params, callback) {
    let content = `${params.title}\n{params.content}`
    window.alert(content);
    callback && callback(0)
}
```

这个 fallback 函数我们希望能够更加通用，每个调用方法都应该有自己的 fallback 函数，所以前面的 callHandler 应该设计成这样：

```
function callHandler(name, params, fallback) {
    return function(...rest, callback) {
        const paramsList = {};
        for (let i = 0; i < params.length; i++) {
            paramsList[params] = rest[i];
        }
        if (!callback) {
            callback = function(result) {};
        }
        if (fallback && !window.WebViewJavascriptBridge.hasHandler(name)))
```

```
            fallback(paramsList, callback);
        } else {
            window.WebViewJavascriptBridge.callHandler(name, params, callba
        }
    }
}
```

我们可以基于这个函数封装一些功能方法，比如前面的 alert：

```
function fallback(params, callback) {
    let content = `${params.title}\n{params.content}`
    window.alert(content);
    callback && callback(0)
}

function alert(
  title,
  content,
  cb: any
) {
  return callHandler(
    'alert',
    ['title', 'content'],
    fallback
  )(title, content, cb);
}
alert(`this is title`, `hahaha`, function() {
    console.log('success')
})
```

具体效果类似下面这种，这是从 Google 上随便找的一张图（侵删）：

那么客户端又如何实现回调 callback 函数的呢？前面说过，客户端想调用 JS 方法，只能调用挂载到 window 对象上面的。

因此，这里使用了一种很巧妙的方法，实际上 callback 函数依然是 JS 执行的。

在调用 Native 之前，我们可以先将 callback 函数和一个 uniqueId 映射起来，然后存在 JS 本地。我们只需要将 callbackId 传给 Native 就行了。

```
function callHandler(name, data, callback) {
    const id = `cb_${uniqueId++}_${new Date().getTime()}`;
    callbacks[id] = callback;
    window.bridge.send(name, JSON.stringify(data), callbackId)
}
```

在客户端这里，当 send 方法接收到参数之后，会执行相应功能，然后使用 webView.loadUrl 主动调用前端的一个接收函数。

```
@JavascriptInterface
public void send(final String cmd, String data, final String callbackId) {
    // 获取数据，根据 cmd 来调用对应功能
    // 调用结束后，回调前端 callback
    String js = String.format("javascript: window.bridge.onReceive(\'%1$s\'
    webView.loadUrl(js);
}
```

所以 JS 需要事前定义好这个 onReceive 方法，它接收一个 callbackId 和一个 result。

```
window.bridge.onReceive = function(callbackId, result) {
    let params = {};
    try {
        params = JSON.parse(result)
    } catch (err) {
        //
    }
    if (callbackId) {
        const callback = callbacks[callbackId];
        callback(params)
        delete callbacks[callbackId];
    }
}
```

大致流程如下：

## registerHandler

注册的流程比较简单，也是我们把 callback 函数事先存到一个 messageHandler 对象里面，不过这次的 key 不再是一个随机的 id，而是 name 。

```
function registerHandler(handlerName, callback) {
    if (!messageHandlers[handlerName]) {
      messageHandlers[handlerName] = [handler];
    } else {
      // 支持注册多个 handler
      messageHandlers[handlerName].push(handler);
    }
}
// 检查是否有这个注册可以直接检查 messageHandlers 里面是否有
function hasRegisteredHandler(handlerName) {
    let has = false;
    try {
      has = !!messageHandlers[handlerName];
    } catch (exception) {}
      return has;
    }
```

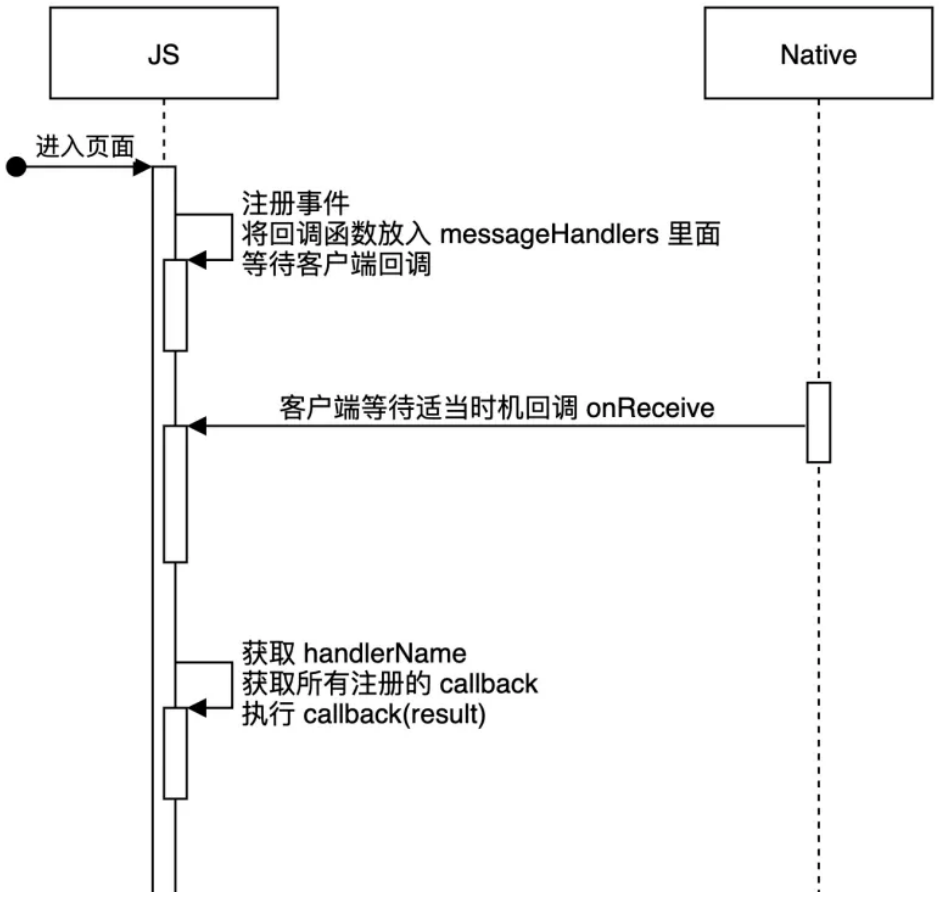
这里不像 callHandler 需要主动调用 window.bridge.send 去通知客户端，只需要等客户端到了相应的时机来调用 window.bridge.onReceive 就行了。

所以这里还需要改造一下 onReceive 方法。由于不再会有 callbackId 了，所以客户端可以传个空值，然后将 handlerName 放到 result 里面。

```
window.bridge.onReceive = function(callbackId, result) {
    let params = {};
    try {
        params = JSON.parse(result)
    } catch (err) {
        //
    }
    if (callbackId) {
```

```
        const callback = callbacks[callbackId];
        callback(params)
        delete callbacks[callbackId];
    } else if (params.handlerName)(
        // 可能注册了多个
        const handlers =  messageHandlers[params.handlerName];
        for (let i = 0; i < handlers.length; i++) {
            try {
                delete params.handlerName;
                handlers[i](params);
            } catch (exception) {
            }
        }
    )
}
```

这种情况下的流程如下，可以发现完全不需要 JS 调用 Native：



## iOS Bridge

讲完了 Android，我们再来讲讲 iOS，原本 iOS 可以和 Android 设计一致，可是由于种种原因导致有不少差异。

iOS 和 Android 中最显著的差异就在于这个 `window.bridge.send` 方法的实现，Android 里面是直接调用 Native 的方法，iOS 中是通过 URL Scheme 的形式调用。

协议依然是 WebViewJavaScriptBridge 里面的协议，URL Scheme 本身不会传递数据，只是告诉 Native 有新的调用。

然后 Native 会去调用 JS 的方法，获取队列里面所有需要执行的方法。

所以我们需要事先创建好一个 iframe，插入到 DOM 里面，方便后续使用。

```
const CUSTOM_PROTOCOL_SCHEME = 'wvjbscheme';
const QUEUE_HAS_MESSAGE = '__WVJB_QUEUE_MESSAGE__';
function _createQueueReadyIframe(doc) {
    messagingIframe = doc.createElement('iframe');
    messagingIframe.style.display = 'none';
    messagingIframe.src = CUSTOM_PROTOCOL_SCHEME + '://' + QUEUE_HAS_MESSAG
    doc.documentElement.appendChild(messagingIframe);
  }
```

### callHandler

每次调用的时候只需要复用这个 iframe 就行了。这里是处理 callback 并通知 Native 的代码：

```
function callHandler(handlerName, data, responseCallback) {
    _doSend({ handlerName: handlerName, data: data }, responseCallback);
  }
function _doSend(
    message,
    callback
  ) {
    if (responseCallback) {
      const callbackId = `cb_${uniqueId++}_${new Date().getTime()}`;
      callbacks[callbackId] = callback;
      message['callbackId'] = callbackId;
    }
    sendMessageQueue.push(message);
    messagingIframe.src = CUSTOM_PROTOCOL_SCHEME + '://' + QUEUE_HAS_MESSAG
  }
```

通知 Native 之后，它怎么拿到我们的 handlerName 和 data 呢？我们可以实现一个 fetchQueue 的方法。

```
  function _fetchQueue() {
    const messageQueueString = JSON.stringify(sendMessageQueue);
    sendMessageQueue = [];
    return messageQueueString;
  }
```

然后将其挂载到 window.WebViewJavascriptBridge 对象上面。

```
  window.WebViewJavascriptBridge = {
    _fetchQueue: _fetchQueue
  };
```

这样 iOS 就可以使用 evaluateJavaScript 轻松拿到这个 messageQueue 。

```
- (void)flushMessageQueue
```

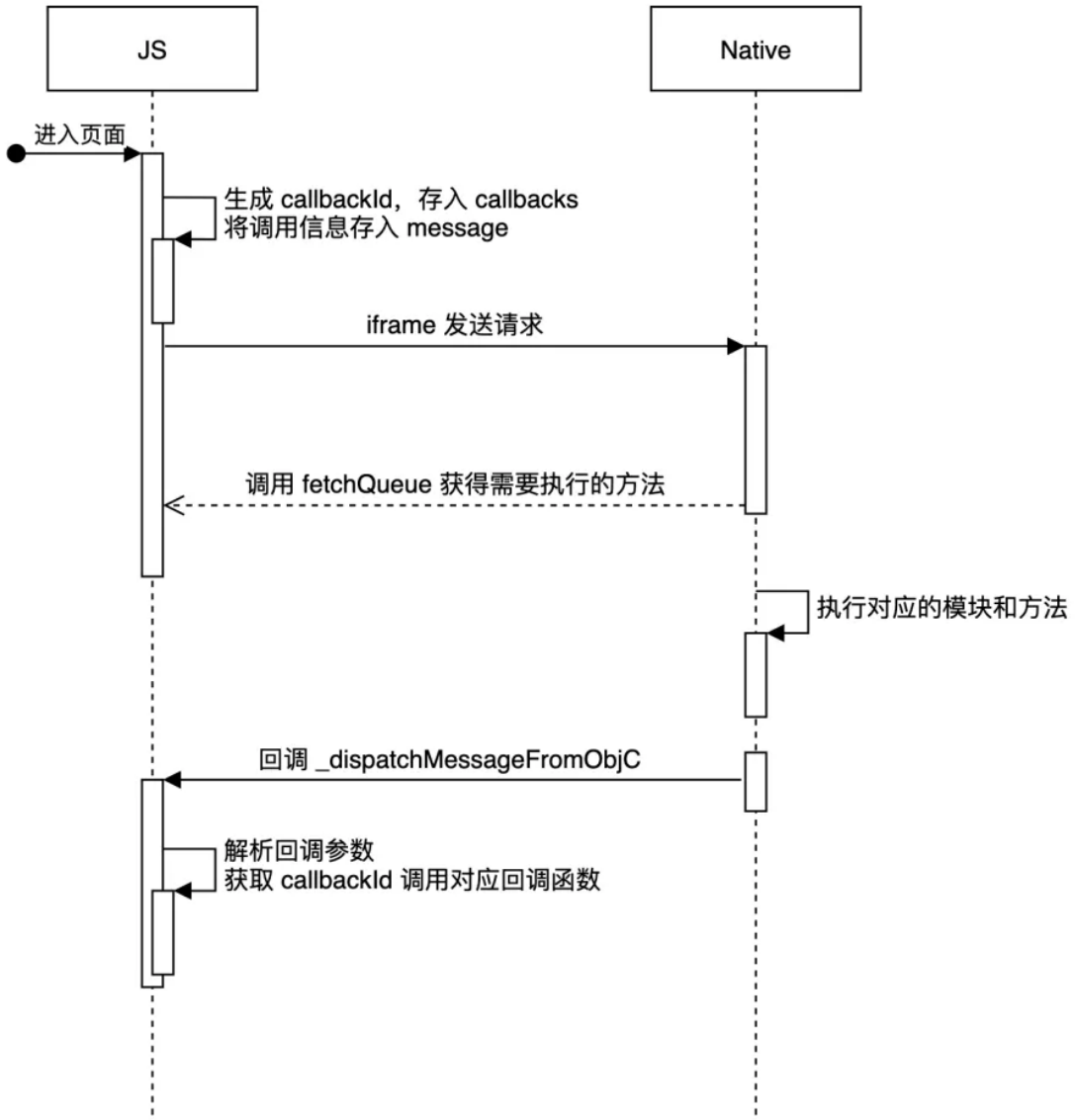
```
{
    [_webView evaluateJavaScript:@"WebViewJavascriptBridge._fetchQueue();"
        [self _flushMessageQueue:result];
    }];
}
- (void)_flushMessageQueue:(id)messageQueueObj
{
    // 解析 messageQueueString
    // 根据传入的 handlerName 执行对应操作
}
```

那么 iOS 又是如何回调 JS 的 callback 函数呢？这个其实和 Android 的 onReceive 是同样的原理。

这里可以实现一个 _handleMessageFromObjC 方法，同样挂载到 window.WebViewJavascriptBridge 对象上面，等待 iOS 回调。

```
function _dispatchMessageFromObjC(messageJSON) {
    const message = JSON.parse(messageJSON);
    if (message.responseId) {
        var responseCallback = callbacks[message.responseId];
        if (!responseCallback) {
          return;
        }
        responseCallback(message.responseData);
        delete callbacks[message.responseId];
     }
}
```

流程如下图：




## registerHandler

registerHandler 和 Android 原理是一模一样的，都是提前注册一个事件，等待 iOS 调用，具体就不多讲了，这里直接放代码：

```
// 注册
function registerHandler(handlerName, handler) {
    if (typeof messageHandlers[handlerName] === 'undefined') {
      messageHandlers[handlerName] = [handler];
    } else {
      messageHandlers[handlerName].push(handler);
```

```
    }
  }
// 回调
function _dispatchMessageFromObjC(messageJSON) {
    const message = JSON.parse(messageJSON);
    if (message.responseId) {
        var responseCallback = callbacks[message.responseId];
        if (!responseCallback) {
          return;
        }
        responseCallback(message.responseData);
        delete callbacks[message.responseId];
     } else if (message.handlerName){
        handlers = messageHandlers[message.handlerName];
        for (let i = 0; i < handlers.length; i++) {
          try {
            handlers[i](message.data, responseCallback);
          } catch (exception) {
          }
        }
     }
}
```

## 总结

这些就是 Hybrid 里面 JS 和 Native 交互的大致原理，忽略了不少细节，比如初始化 WebViewJavascriptBridge 对象等等，感兴趣的也可以参考一下这个库：JsBridge