Hybrid 混合开发

1、参考方案（美图秀秀）

1. 改造端（Z

2. 目标：

JSBridge:

基于 webview 机制
完成 native端（java、objective-c）和 js端 的 调度者、双向 通讯

3. 通知、调用 原理：

1. native 直接 API 注入：
即 native 直接获取 JS 环境上下文，在上面 挂载对象 或者 方法，供 JS 可以直接调用；
但是 android 和 ios 分别 有不同的 挂载方式

2. Webview 的 几个不常用方法 的 拦截处理：
prompt(提示)、console、alert

这几个方法的 冲突 概率低

实现原理：

--> 其实 就是 依赖于：
webview 发出的 网络请求、客户端 都能监听 和 捕获；
--> 特殊处理 我们这种 自定义的协议头 即可

2. 协议的定制

我们需要制定一套**URL Scheme**规则，通常我们的请求会带有对应的协议开头。例如常见的 https://xxx.com 或者 file://1.jpg，代表着不同的含义。我们这里可以将协议类型的请求定制为：

```
xxcommand://xxxx?param1=1&param2=2
```

这里有几个需要注意点的是：

(1) xxcommand:// 只是一种规则，可以根据业务进行制定，使其具有含义，例如我们定义 xxcommand:// 为公司所有App共通用，为通用工具协议：

```
xxcommand://getProxy?h=1
```

而定义 xxapp:// 为每个App单独的业务协议。

```
xxapp://openCameraTh=2
```

不同的协议头代表着不同的含义，这样便能清楚知道每个协议的适用范围。

(2) 这项不要使用 location.href 发送，因为其自身机制有个问题是即时并发多次请求会被合并成为一次，导致协议被吞噬，而并发协议其实是非常常见的功能。我们会使用**创建 iframe 发送请求**的方式。

(3) 通常考虑到安全性，需要在客户端中设置域名白名单或者限制，避免公司内部业务协议被第三方直接调用。

定制协议：

3.协议的拦截

客户端可以通过 API 对 WebView 发出的请求进行拦截：

- IOS 上：shouldStartLoadWithRequest
- Android：shouldOverrideUrlLoading

当解析到请求 URL 为制定的协议时，**即不发起对应的页面请求，而是解析参数，并进行相关功能或者方法的调用，完成协议功能的映射。**

3、协议 拦截处理：

android 上的 WebViewClient.shouldOverrideUrlLoading(WebView view, String url)

--> 这里，发现 url 是 自定义的协议头 时，就 可以 转为 解析参数，native 接口 调用了 。

4. 通知、调用 原理：

4.协议回调

由于协议的本质其实是发送请求，这属于一个异步的过程，因此我们便需要处理对应的回调机制。这里我们采用的方式是：

- 1. 发送协议时，通过协议的唯一标识注册自定义事件，并响应监听绑定到对应的事件上。
- 1. 客户端完成对应的功能后，调用 Bridge 的dispatch API，直接传送 data 触发该协议的自定义事件。

```
// 业务调用API：JS
Bridge.getNetwork(data => {});

// Bridge内部功能：
//创建唯一标识 handler
// 注册自定义事件：
// 拼接并发送协议；
const handler = 1;
window.addEventListener(`getNetwork_${handler}`, callback, false);
Bridge.send(`xxcommand://getNetwork?handler=${handler}`);

// Native层获取网络状态后通过bridge再次传回
// 将网络状态通过事件直接触发自定义事件并传递数据
event.data = network;
window.dispatchEvent(event);
```

通过事件的机制，会让开发更符合我们前端的习惯，例如当你需要监听客户端的通知时，同样只需要在通过 `addEventListener` 进行监听即可。

Tips: 这很有一点需要注意的是，应该避免事件的多次重复绑定，因此当唯一标识重置时，需要 `removeEventListener` 对应的事件。

5. 通知、调用 原理：

3. Webview 自定义 URL scheme 协议 跳转拦截：
即 拦截 webview 的 信息传递、特殊处理

5.参数传递方式

由于 WebView 对 URL 会有长度的限制，因此常规的通过 search参数 进行传递的方式便具有一个问题，既 **当需要传递的参数过长时，可能会导致被截断**。例如传递base64或者传递大量数据时。

因此我们需要制定新的参数传递规则，我们使用的是通数调用的方式。这里的原理主要是基于：

Native 可以直接调用 JS 方法并直接获取函数的返回值。

我们需要对每条协议标记一个唯一标识，并把参数存入参数池中，到时客户端再通过该唯一标识从参数池中获取对应的参数即可。 **大长的参数，不能直接传递，就让 被调用方（native）自己 根据 标识 去读**

5、（过长的）参数传递：

2. JS 调用 native

1. 改造端（Z

图片 概览：

native 调用 JS

(二) Native 通知 Javascript

由于 Native 可以操作 H5 的前主，因此拥有更大的权限，上面也提到了 **Native 可以通过 WebView API 直接执行 Js 代码**。这样的权限也就让这个方向的通讯变得十分的便捷。

- IOS: stringByEvaluatingJavaScriptFromString

```
// Swift
webview.stringByEvaluatingJavaScriptFromString("alert('NativeCall')")
```

- Android: loadUrl (4.4-)

```
// 调用js中的JSBridge.trigger方法
// 这方法的意思是无法获取函数返回值。
webview.loadUrl("javascript:JSBridge.trigger('NativeCall')")
```

Tips: 当系统低于 4.4时，evaluateJavascript 是无法使用的，因此单纯的使用 loadUrl 无法获取 JS 返回值。这时候 **需要使用间接的 prompt 的方法进行事件，让 H5端 调过 prompt 进行数据的发送，客户端进行二次 拦截 获取数据。 js 调用 prompt () 方法 来 回传 数据的数据 --> native 端 特殊处理**

- Android: evaluateJavascript (4.4+)

```
// 4.4以后使用这个方法可以直接获取函数返回值：
mWebView.evaluateJavascript("javascript:JSBridge.trigger('NativeCall')",      new ValueCallb
    @Override
    public void onReceiveValue(String value) {
        //这里是 js 函数的结果
    }
});
```

基于上面的原理，我们已经明白 JSBridge 最基础的原理，并且能实现 Native <=> H5 的双向通讯机制了。

通过 webview 调用 同一个 JS 方法：
--> javascript：JSBridge.trigger('某 js 方法')

1.
4.4 之前：
webview.loadUrl("上述 语句")

这方法是 单向调用，没有返回值！
所以 返回值 需要 特殊 获取，比如 拦截处理 prompt 方法

2.
4.4 之后：
webview.evaluateJavascript("上述语句", new valueCallback(){
onReceiveValue(String value){
// 返回值
}}

4. JS Bridge 脚本的接入：封装 sdk 引入

(三) JSBridge 的接入

接下来，我们来撸下代码上需要的资源。实现这款方案，从上图可以看出，其实可以分为两个部分：

- JS部分(bridge): 在JS环境中注入 bridge 的实现代码，包含了协议的拼装/发送/参数池/回调池等一些基础功能。
- Native部分(SDK): 在客户端中 bridge 的实现插件代码，实现了URL拦截与解析/环境信息的注入/通用功能映射等功能。

我们这款的做法是，将这两部分一起封装成一个 **Native SDK**，由客户端统一引入。客户端在初始化一个 WebView 打开页面时，如果页面地址在白名单中，会**直接在 HTML 的头部注入内容的 bridge.js**。这样的做法有以下好处：

- 双方的代码统一维护，避免出现版本沟通的烦恼。有更新时，只要告诉客户端更新SDK即可，不会出现版本兼容的问题；
- App的接入十分方便，只需要给又拍摄入最新版本的SDK，即可自动运行整套Hybrid方案，便于在多个App中快速的落地；
- H5端无感知，这样有利于将 bridge 开放给第三方页面使用。

这部有一点需要注意的是，**协议的调用，一定是需要确保执行自bridge.js 成功注入后。**由于客户端的注入行为属于一个整加的异步行为，从H5方面来上报端使用的协议时，**可能会出现**协议还没来得及注入的情况。基于上面的问题H5端加 计时器，当指令被可调 `window.addEventListener('bridgeReady', e => {})` 进行监听时，

**H5 不用担心 的 bridge解析 加载完毕 问题，
需要 native端，通过 js 的 window 机制 发送一个 事件 告知**

即 往 当前的 H5页面，注入 bridge 运行脚本，方便 当前页面 的调用 --> 调求 各 native 接口

5. APP页 H5 网页的读取

1. 在线 h5
只一个 url，直接 去加载即可

2. 内置 H5 网页

(四) App中 H5 的接入方式

将 H5 接入 App 中通常有两种方式：

>> **在线H5**，这是最常见的一种方式，我们只需要将一个代码页部署到服务器上，只要把对应的 URL地址 给到客户端，同 WebView 打开读URL，即可嵌入。这方式的好处在于：

- 我空性强，有非常强记的开发/调试/更新/上线能力；
- 因更容易复用重上，可以不仅单纯满足客户端的功能，
- 嵌入成本最低，完全的跨端原生机。

但相对的，这种方式也有对应的缺点：

- 完全的网络依赖，在离线的情况下无法打开页面；
- 首屏加载速度依赖于网络，网络较慢时，首屏加载功能慢；

通常，这种方式**不适用在一些比较重要的页面上**，例如一些带支付、登录功能、使用功能等页面。这些页面的特点**功能性不强，不太需要跨端的能力，但不需要高性能体验**，比一些第三方页面嵌入上，也会使用这种方式，例如我们的页面调用插件 SDK，**登录、注册等 使用频繁度较 强 不强，，
切换 手机网络 ！**

>> **内置H5**，这是一种**未来的趋势与方式，我们需要把内核进行打包**，下发到客户端，并由客户端直接解压到本地缓存中，通常情况 以同一份H5嵌入 相对较重要的模块上，，H5大点面。

- 由于网络未响应，首屏加载速度快，用户体验更为稳定顺畅；
- 可以不依赖网络，离线运行；

但同时，它的劣势也十分明显：

- 开发流程/更新机制复杂ios，需要客户端，甚至服务端的共同协作；
- 空间的可控性 App 有限制；

这两种嵌入方式可有自己的优缺点，你谁能扬不同场景进行选择。

2. 实战端：

github 地址：

这款h5的使用：
1. 统一化 页面变化，迭代快的页面 写成 h5
2. 一些核心通用功能，用原生页面来写 即可

3. h5页面，是 缓存在本地的 --> 即 不依赖于 实时网络，加载速度快！

而且 H5 页面，都加载了 jsbridge代码，通过 native 扩展了一些能力，比如 热更新

h5页面的动态，去服务器 检查 自己这个app的这个页面，有没有对应的那h5包（app名、模块名、版本号等）
有的话，调用 native 支持接口 去 下载、解压、下次直接 使用新的 h5页面 即 实现了 热更新

--> 即，会 可以 快速上线，且 不依赖于 实时网络，页面 加载速度 也比较快

--> 缺点 可能就是：
一开始的 app包 稍大（有那加 h5），且 在多native端 ui表现 不一致，webview 源码性能差，不如 native

2、 支付宝 方案

1、h5开发 + uc 自定义 内核（优秀！）

2、基于上面，还封装了 小程序 DSL语言，实现了 生态。，

mPaaS H5 容器

mobile platform as a service
移动 开发平台服务，，

支付宝的小程序 生态平台，，

https://github.com/wsycomeon/
XMind