现代 Hybrid 开发与原理解析

开始之前, 咱们先来罗列一下当前市面上, 移动端的各种开发方式

1. Native App

   纯原生的 app 开发模式, 有 android 和 iOS 两大系统。

   分别使用 Java/Swift、C++ 、Objective-C 等语言。

   优点: 拥有最好的性能和最好的体验
   缺点: 开发和发布成本非常高, 两端需要不同的技术人员来维护, 而原生开发人员又非常稀缺。

2. WebApp

   移动端运行在浏览器上的网站，我们都称之为 H5 应用, 一般指我们平时开发的 SPA 或者 MPA。

   使用的语言当然就是我们大家最熟悉的 javascript. 又分为 vue/react/angular 等等框架

   优点： 1. 开发和发布是最方便的 2. 可以随时发布随时更新 3. 可以跨平台, 调试非常方便 4. 不存在多版本问题, 维护成本非常低
   缺点： 1. 性能和体验一般 2. 受限于浏览器, 能做的事情不多, 而且需要兼容各个浏览器的各种奇葩情况。 3. 入口强依赖浏览器, 只能以 url 的形式存在

3. React Navite App / Weex App

   两者都是为跨平台而生的 App 开发框架, 分别支持 React 和 Vue 语言的开发。

4. Flutter

   全新的跨平台语言, 使用 dart 语言开发。

   说完这几种开发模式, 就轮到了咱们今天的主角：Hybrid 模式

Hybrid 基本介绍

Hybrid 的开发模式, 翻译过来就是指"混合开发".

那么这个混合开发, 到底是将什么和什么给混合起来了呢?

大家可能已经猜到了，就是将 h5 + native 混合了起来。

那么直接在 app 的 webview 里面嵌入一个 h5 页面, 这就是 Hybrid 了吗？？

当然不是！！！！ hybrid 最大的特点就是 h5 和 native 可以双向交互。

通过微信JSSDK介绍Hybrid

如果通过上述概念, 大家对 Hybrid 还是没有一个比较清晰的概念。

那咱们就来看下现代 Hybrid 开发覆盖用户最多的方案吧, 大家可能在猜哪个方案能称之为覆盖用户最多呢?

那就是微信的 JSSDK https://developers.weixin.qq.com/doc/offiaccount/OA_Web_Apps/JS-SDK.html#1

可以看到微信JSSDK中封装了各种微信的功能供h5使用，比如分享、支付、位置等等。h5开发者只需要关注sdk中提供的方法即可, 其他的都由sdk和微信app进行通信来完成功能。

可以大概看一下微信SDK的源码 https://res.wx.qq.com/open/js/jweixin-1.4.0.js.

- [ ] 详述微信JSSDK的源码和使用 https://github.com/a951055/weixin-js-bridge-code

比如我们最常用的微信分享

```
wx.config({
    appId: 'xxxxxx',
    debug: false
});

wx.ready(() => {
    wx.onMenuShareAppMessage({
        title: '哈哈哈哈',
        desc: '哈哈哈哈哈哈哈哈哈',
        link: 'https://www.baidu.com',
        imgUrl: 'https://xxxxxx.png'
    })
```

```
})
```

咱们大概封装一下wechat基础类, 方便以后使用。

现代 Hybrid 开发与原理解析

接下来咱们开始详细介绍一下Hybrid开发的架构，最后会尝试实现一个js端的bridge.

Hybrid开发架构

先来看一张宏观的架构图....

所以最核心的就是Navite和H5的双向通讯, 而通讯是完全依赖于native提供的webview容器，那native提供的这个webview容器有什么特点能支撑起h5和native的通讯呢？具体的通讯流程到底是什么样子呢?

首先说明有两种方式：

- URL Schema， 客户端通过拦截webview请求来完成通讯
- native向webview中的js执行环境，注入API, 以此来完成通讯

一、URL Schema, 客户端拦截webview请求

1. 原理

   在webview中发出的网络请求，都会被客户端监听和捕获到。

   这是我们本节课所有实现的基石。

2. 定义自己的私有协议

   上面说过, 所有网络请求都会被监听到, 网络请求最常见的就是http协议, 比如https://a.b.com/fetchInfo, 这是一个很常见的请求。

   Webview内的H5页面肯定有很多类似的http请求, 我们为了区别于业务请求, 需要定制一套h5和native进行交互的私有协议, 我们通常称呼为URL Schema。

   比如我们现在定义协议头为 qiuku://,

那么随后我们要在webview请求中都带上这个私有协议开头, 比如有一个请求是setLeftButton, 实际发出的请求会是qiuku://setLeftButton?params1=xxx&params2=xxx.

这里大家记住, 这个协议的名称是我们自定义的, 只要h5和native协商好即可。
但是如果公司旗下有多个app, 对于通用的业务一般会定义一个通用的协议头, 比如common://；对于每个app自己比较独立的业务, 基本每个app都会自己定义一套协议, 比如appa://, appb://, appc://.

3. 请求的发送

对于webview请求的发送, 我们一般使用iframe的方式。也可以使用location.href的方式, 但是这种方式不适用并行发请求的场景。

```
const doc = window.document;
const body = window.document.body;
const iframe = doc.createElement('iframe');

iframe.style.display = 'none';
iframe.src = 'qiuku://setLeftButton?param1=12313123';

body.appendChild(iframe);
setTimeout(() => {
    body.removeChild(iframe);
}, 200)
```

而且考虑到安全性, 客户端中一般会设置域名白名单, 比如客户端设置了qiuku.com为白名单, 那么只有qiuku.com域下发出的请求, 才会被客户端处理。

这样可以避免自己app内部的业务逻辑, 被第三方页面直接调用。

4. 客户端拦截协议请求

iOS和Android对webview请求的拦截方法不太相同。

- iOS: shouldStartLoadWithRequest
- Android: shouldOverrideUrlLoading

当客户端解析到请求的URL协议是约定要的qiuku://时, 便会解析参数, 并根据h5传入的方法名比如setLeftButton, 来进行相关操作（设置返回按钮的处理逻辑）。

5. 请求处理完成后的回调

因为咱们webview的请求本质上还是异步请求的过程, 当请求完成后, 我们需要有一个callback触发, 无论是通知h5执行结果，还是返回一些数据， 都离不开callback的执行。

我们可以使用Js自带的事件机制，window.addEventListener和window.dispatchEvent这两个API。

还是这个例子, 比如咱们现在要调用setLeftButton方法, 方法要传入一个callback来得知是否执行成功了。

```
webview.setLeftButton({ params1: 111 }, (err) => {
    if (err) {
        console.error('执行失败');
        return;
    }
    console.log('执行成功');
    // 业务逻辑
})
```

JsBridge中具体的步骤应该是这样的：

- 在H5调用setLeftButton方法时, 通过 webview_api名称+参数 作为唯一标识,注册自定义事件

```
const handlerId = Symbol();
const eventName = `setLeftButton_${handlerId}`;
const event = new Event(eventName);
window.addEventListener(eventName, (res) => {
    if (res.data.errcode) {
        console.error('执行失败');
        return;
    }
    console.log('执行成功');
```

```
    // 业务逻辑
});

JsBridge.send(`qiuku://setLeftButton?handlerId=${eventName}&params1=111`);
```

- 客户端在接收到请求, 完成自己的对应处理后, 需要调用JsBridge中的dispatch, 携带回调的数据触发自定义事件。

```
event.data = { errcode: 0 };
window.dispatchEvent(event);
```

注入API

上述方式有个比较大的确定, 就是参数如果太长会被截断。以前用这种方式主要是为了兼容iOS6， 现在几乎已经不需要考虑这么低的版本了。

所以现在主流的实现是native向js的执行环境中注入API.

具体怎么操作呢, 咱们分步骤来看：

1. 向native传递信息

   由于native已经向window变量注入了各种api, 所以咱们可以直接调用他们。

   比如现在window.QiukuWebview = { setLeftButton: (params) => {}} 就是native注入的对象api。

   我们可以直接这样调用, 就可以传参数给native了

```
window.QiukuWebview['setLeftButton'](params)
```

   但是为了安全性, 或者为了不要乱码等问题, 我们一般会对参数进行编码, 比如转换为base64格式。

2. 准备接收native的回调

咱们同样可以在window上声明接收回调的api

```
window['setLeftButton_Callback_1'] = (errcode, response) => {
    console.log(errcode);
}
```

同样为了安全性和参数传递的准确性, native也会将回调的数据进行base64编码, 咱们需要在回调函数里进行解析。

3. Native调用回调函数

Native怎么知道哪个是这次的回调函数呢? 他们确实不知道, 所以我们需要在调用的时候就告诉native。

```
window.QiukuWebview['setLeftButton'](params)
```

这个Params中, 我们会加入一个属性叫做trigger, 它的值是回调函数的名称, 比如

```
const callbackName = 'setLeftButton_Callback_1';
window.QiukuWebview['setLeftButton']({
    trigger: callbackName,
    ...otherParams
});

window[callbackName] = (errcode, response) => {
    console.log(errcode);
}
```

同时为了保证callbackName的唯一性, 我们一般会加入各种Date.now() + id, 使其保证唯一。

大功告成！！ 注入API的方式就是这么简单, 接下来我们尝试实现一个相对完整的webview类。

h5在app内的运行方式

1. app的webview直接加载一个h5链接

   这个很简单, 就是大家理解的那个样子。非常的灵活和方便。

   但是缺点就是, 没有太好的体验, 除了能用一些native的能力, 其他和在浏览器里打开h5链接没什么区别。

   因为说到底还是加载网络上的资源, 网络一旦不好, 页面加载会非常慢。

2. app内置h5资源

   优点大家应该也猜到了:

- 首屏加载速度很快, 用户体验接近原生
- 可以不依赖网络, 离线运行

同时, 它也有缺点:

- 会增大app的体积
- 需要多方合作完成方案

这个方案比较核心的点是, app内资源应该能随着h5的代码更新而随之更新， 那么具体应该怎么做呢?

这里咱们看一张比较经典的图。。。。

上图的意思是:

开发阶段H5代码可以通过手机设置HTTP代理方式直接访问开发机。
完成开发之后，将H5代码推送到管理平台进行构建、打包，然后管理平台再通过事先设计好的长连接通道将H5新版本信息推送给客户端，客户端收到更新指令后开始下载新包、对包进行完整性校验、merge回本地对应的包，更新结束。

其中，管理平台推送给客户端的信息主要包括项目名（包名）、版本号、更新策略（增量or全量）、包CDN地址、MD5等。

开发中的常见问题

1. iOS webview中滑动不流畅

   一般给滚动容器加一个css属性

```
-webkit-overflow-scrolling: touch; // 惯性滚动，当手指从屏幕移开，滚动条会保持一段时间的滚动
```

2. 滚动穿透

   这是一个非常经典的问题，主要出现在弹窗出现时。

   2.1 弹窗内无滚动，背景页面有滚动

   这种一般直接在弹窗容器元素上加一个监听事件就可以了

```
document.body.addEventListener('touchmove', function(e) {
    // 阻止默认事件
    e.preventDefault();
});
```

   在vue中就更简单了，直接在元素上加一个@touchmove.prevent即可

   2.2 弹窗内有滚动，背景页面有滚动

   这种情况就不能简单的组织touchmove事件了，一般这种情况我都是写一个指令，

```
const inserted = () => {
    const scrollTop = document.body.scrollTop ||
document.documentElement.scrollTop;
    document.body.style.cssText += 'position:fixed;width:100%;top:-' +
scrollTop + 'px;';
};
```

```
const unbind = () => {
    const body = document.body || document.documentElement;
    body.style.position = '';
    const top = body.style.top;
    document.body.scrollTop = document.documentElement.scrollTop = -
parseInt(top!, 10);
    body.style.top = '';
};

// 适用于弹窗滚动，弹窗需v-if条件渲染
export const vScroll = {
    inserted,
    unbind
};

Vue.directive('scroll', vScroll);

div(v-scroll)
```

3. 刘海屏的安全区域留白

设置 viewport-fit 为 cover

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-
scalable=yes, viewport-fit=cover">
```

对应元素使用safe area inset 变量

```
.bottom {
    padding-bottom: 0;
    padding-bottom: constant(safe-area-inset-bottom);
    padding-bottom: env(safe-area-inset-bottom);
}

margin-bottom: 16.6rem;
```

```
margin-bottom: calc(constant(safe-area-inset-bottom) + 16.6rem);
margin-bottom: calc(env(safe-area-inset-bottom) + 16.6rem);
```