[**浅谈Hybrid技术的设计与实现**](http://www.cnblogs.com/yexiaochai/p/4921635.html)

**前言**

[浅谈Hybrid技术的设计与实现](http://www.cnblogs.com/yexiaochai/p/4921635.html)

[浅谈Hybrid技术的设计与实现第二弹](http://www.cnblogs.com/yexiaochai/p/5524783.html)

[浅谈Hybrid技术的设计与实现第三弹——落地篇](http://www.cnblogs.com/yexiaochai/p/5813248.html)

随着移动浪潮的兴起，各种APP层出不穷，极速的业务扩展提升了团队对开发效率的要求，这个时候使用IOS&Andriod开发一个APP似乎成本有点过高了，而H5的低成本、高效率、跨平台等特性马上被利用起来形成了一种新的开发模式：Hybrid APP。

作为一种混合开发的模式，Hybrid APP底层依赖于Native提供的容器（UIWebview），上层使用Html&Css&JS做业务开发，底层透明化、上层多多样化，这种场景非常有利于前端介入，非常适合业务快速迭代，于是Hybrid火啦。

本来我觉得这种开发模式既然大家都知道了，那么Hybrid就没有什么探讨的价值了，但令我诧异的是依旧有很多人对Hybrid这种模式感到陌生，这种情况在二线城市很常见，所以我这里尝试从另一个方面向各位介绍Hybrid，期望对各位正确的技术选型有所帮助。

**Hybrid发家史**

最初携程的应用全部是Native的，H5站点只占其流量很小的一部分，当时Native有200人红红火火，而H5开仅有5人左右在打酱油，后面无线团队来了一个执行力十分强的服务器端出身的leader，他为了了解前端开发，居然亲手使用jQuery Mobile开发了第一版程序，虽然很快方案便被推翻，但是H5团队开始发力，在短时间内已经赶上了Native的业务进度：



突然有一天andriod同事跑过来告诉我们andriod中有一个方法最大树限制，可能一些页面需要我们内嵌H5的页面，于是Native与H5框架团队牵头做了第一个Hybrid项目，携程第一次出现了一套代码兼容三端的情况。这个开发效率杠杠的，团队尝到了甜头，于是乎后续的频道基本都开始了Hybrid开发，到我离开时，整个机制已经十分成熟了，而前端也有几百人了。

**场景重现**

狼厂有三大大流量APP，手机百度、百度地图、糯米APP，最近接入糯米的时候，发现他们也在做Hybrid平台化相关的推广，将静态资源打包至Native中，Native提供js调用原生应用的能力，从产品化和工程化来说做的很不错，但是有两个瑕疵：

① 资源全部打包至Naive中APP尺寸会增大，就算以增量机制也避免不了APP的膨胀，因为现在接入的频道较少一个频道500K没有感觉，一旦平台化后主APP尺寸会急剧增大

② 糯米前端框架团队封装了Native端的能力，但是没有提供配套的前端框架，这个解决方案是不完整的。很多业务已经有H5站点了，为了接入还得单独开发一套程序；而就算是新业务接入，又会面临嵌入资源必须是静态资源的限制，做出来的项目没有SEO，如果关注SEO的话还是需要再开发，从工程角度来说是有问题的。

但从产品可接入度与产品化来说，糯米Hybrid化的大方向是很乐观的，也确实取得了一些成绩，在短时间就有很多频道接入了，随着推广进行，明年可能会形成一个大型的Hybrid平台。但是因为我也经历过推广框架，当听到他们忽悠我说性能会提高70%，与Native体验基本一致时，不知为何我居然笑了......

**总结**

如果读了上面几个故事你依旧不知道为何要使用Hybrid技术的话，我这里再做一个总结吧：

Hybrid开发效率高、跨平台、低层本

Hybrid从业务开发上讲，没有版本问题，有BUG能及时修复

Hybrid是有缺点的，Hybrid体验就肯定比不上Native，所以使用有其场景，但是对于**需要快速试错、快速占领市场的团队来说，Hybrid一定是不二的选择，团队生存下来后还是需要做体验更好的原生APP**。

好了，上面扯了那么多没用的东西，今天的目的其实是为大家介绍Hybrid的一些设计知识，如果你认真阅读此文，可能在以下方面对你有所帮助：

① Hybrid中Native与前端各自的工作是什么

② Hybrid的交互接口如何设计

③ Hybrid的Header如何设计

④ Hybrid的如何设计目录结构以及增量机制如何实现

⑤ 资源缓存策略，白屏问题......

文中是我个人的一些开发经验，希望对各位有用，也希望各位**多多支持讨论**，指出文中**不足**以及提出您的一些**建议**。

然后文中Andriod相关代码由我的同事明月提供，这里特别感谢明月同学对我的支持，这里扫描二维码可以下载APP进行测试：

**Andriod APP二维码：**



**代码地址：**

<https://github.com/yexiaochai/hybrid>

**Native与前端分工**

在做Hybrid架构设计之前需要分清Native与前端的界限，首先Native提供的是一宿主环境，要合理的利用Native提供的能力，要实现通用的Hybrid平台架构，站在前端视角，我认为需要考虑以下核心设计问题。

**交互设计**

Hybrid架构设计第一个要考虑的问题是如何设计与前端的交互，如果这块设计的不好会对后续开发、前端框架维护造成深远的影响，并且这种影响往往是不可逆的，所以这里需要前端与Native好好配合，提供通用的接口，比如：

① NativeUI组件，header组件、消息类组件

② 通讯录、系统、设备信息读取接口

③ H5与Native的互相跳转，比如H5如何跳到一个Native页面，H5如何新开Webview做动画跳到另一个H5页面

**资源访问机制**

Native首先需要考虑如何访问H5资源，做到既能以file的方式访问Native内部资源，又能使用url的方式访问线上资源；需要提供前端资源增量替换机制，以摆脱APP迭代发版问题，避免用户升级APP。这里就会涉及到静态资源在APP中的存放策略，更新策略的设计，复杂的话还会涉及到服务器端的支持。

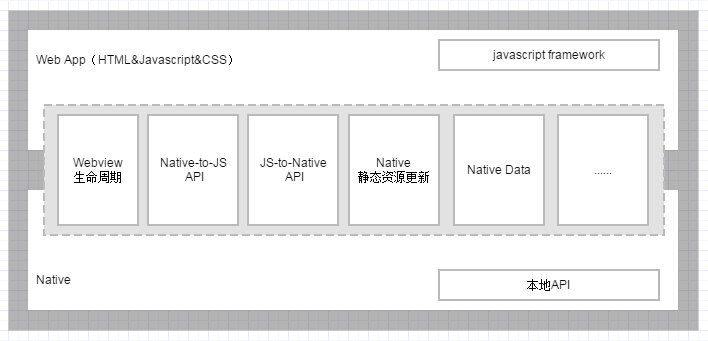
**账号信息设计**

账号系统是重要并且无法避免的，Native需要设计良好安全的身份验证机制，保证这块对业务开发者足够透明，打通账户信息。

**Hybrid开发调试**

功能设计完并不是结束，Native与前端需要商量出一套可开发调试的模型，不然很多业务开发的工作将难以继续，这个很多文章已经接受过了，本文不赘述。

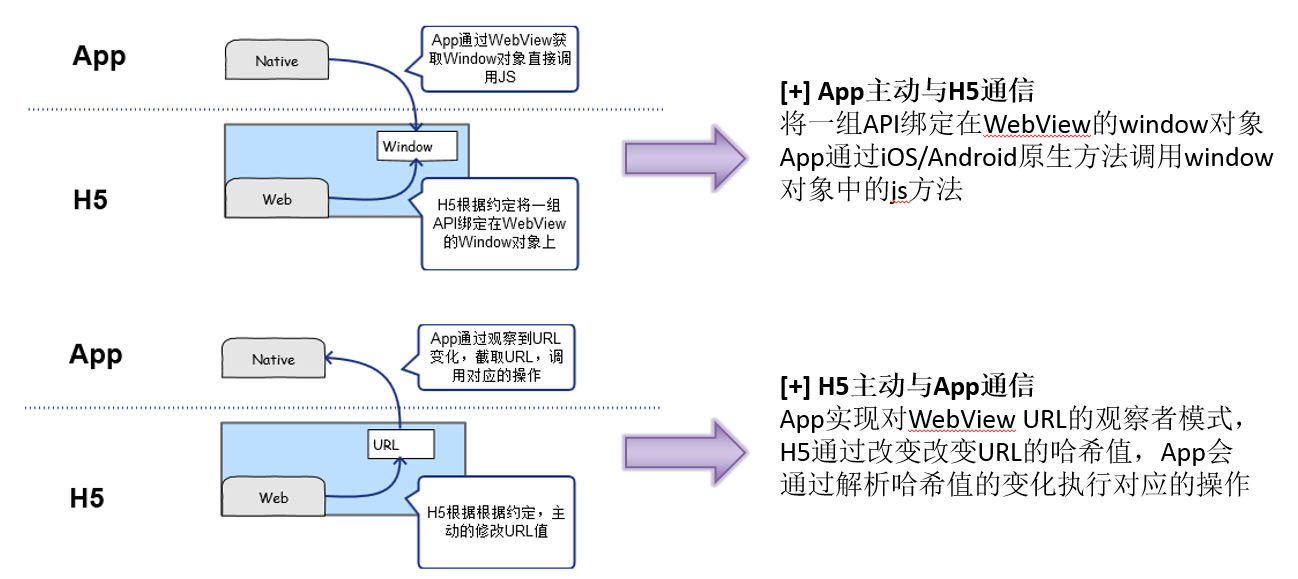
至于Native还会关注的一些通讯设计、并发设计、异常处理、日志监控以及**安全模块**因为不是我涉及的领域便不予关注了（事实上是想关注不得其门），而前端要做的事情就是封装Native提供的各种能力，整体架构是这样的：



真实业务开发时，Native除了会关注登录模块之外还会封装支付等重要模块，这里视业务而定。

**Hybrid交互设计**

Hybrid的交互无非是Native调用前端页面的JS方法，或者前端页面通过JS调用Native提供的接口，两者交互的桥梁皆Webview：



app自身可以自定义url schema，并且把自定义的url注册在调度中心， 例如

* ctrip://wireless 打开携程App
* weixin:// 打开微信

我们JS与Native通信一般就是创建这类URL被Native捕获处理，后续也出现了其它前端调用Native的方式，但可以做底层封装使其透明化，所以重点以及是如何进行前端与Native的交互设计。

**JS to Native**

Native在每个版本会提供一些API，前端会有一个对应的框架团队对其进行封装，释放业务接口。比如糯米对外的接口是这样的：

[复制代码](javascript:void(0);)

1 BNJS.http.get()；//向业务服务器拿请求据【1.0】 1.3版本接口有扩展

2 BNJS.http.post()；//向业务服务器提交数据【1.0】

3 BNJS.http.sign()；//计算签名【1.0】

4 BNJS.http.getNA()；//向NA服务器拿请求据【1.0】 1.3版本接口有扩展

5 BNJS.http.postNA()；//向NA服务器提交数据【1.0】

6 BNJS.http.getCatgData()；//从Native本地获取筛选数据【1.1】

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

1 BNJSReady(function(){

2 BNJS.http.post({

3 url : 'http://cp01-testing-tuan02.cp01.baidu.com:8087/naserver/user/feedback',

4 params : {

5 msg : '测试post',

6 contact : '18721687903'

7 },

8 onSuccess : function(res){

9 alert('发送post请求成功！');

10 },

11 onFail : function(res){

12 alert('发送post请求失败！');

13 }

14 });

15 });

[复制代码](javascript:void(0);)

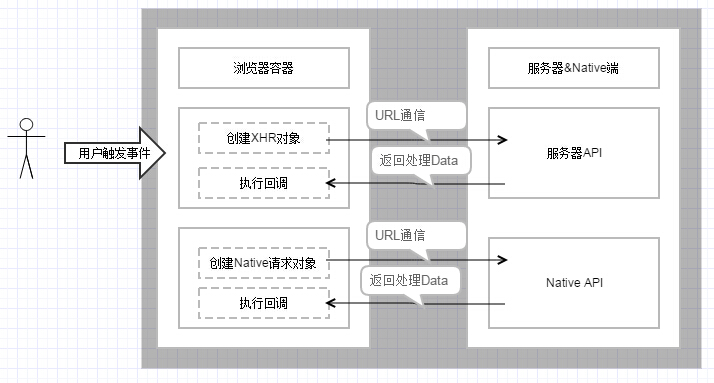
前端框架定义了一个全局变量BNJS作为Native与前端交互的对象，只要引入了糯米提供的这个JS库，并且在糯米封装的Webview容器中，前端便获得了调用Native的能力，我揣测糯米这种设计是因为这样便于第三方团队的接入使用，手机百度有一款轻应用框架也走的这种路线：

clouda.mbaas.account //释放了clouda全局变量

这样做有一个前提是，**Native本身已经十分稳定了**，很少新增功能了，否则在直连情况下就会面临一个尴尬，因为web站点永远保持最新的，就会在一些低版本容器中调用了没有提供的Native能力而报错。

**API式交互**

手白、糯米底层如何做我们无从得知，但我们发现调用Native API接口的方式和我们使用AJAX调用服务器端提供的接口是及其相似的：



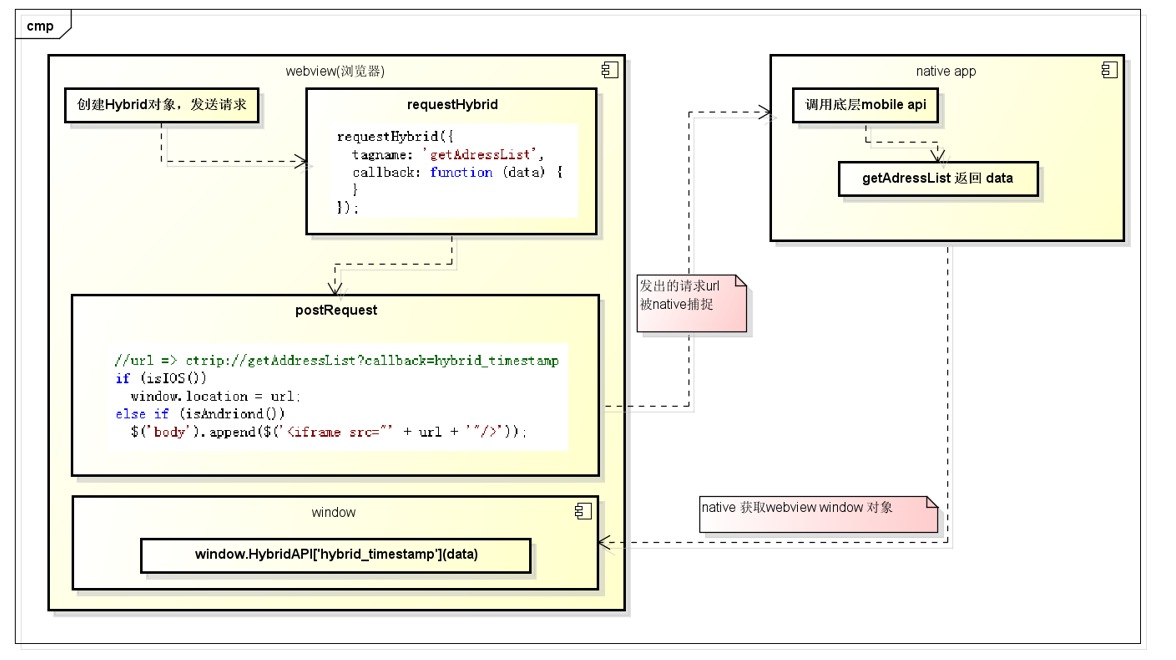
这里类似的微薄开放平台的接口是这样定义的：

|  |  |  |
| --- | --- | --- |
| **粉丝服务（**[**新手接入指南**](http://open.weibo.com/wiki/Messages_api_start)**）** | | |
| 读取接口 | [接收消息](http://open.weibo.com/wiki/%E6%8E%A5%E6%94%B6%E6%B6%88%E6%81%AF) | 接收用户私信、关注、取消关注、@等消息接口 |
| 写入接口 | [发送消息](http://open.weibo.com/wiki/%E5%8F%91%E9%80%81%E6%B6%88%E6%81%AF) | 向用户回复私信消息接口 |
| [生成带参数的二维码](http://open.weibo.com/wiki/%E7%94%9F%E6%88%90%E5%B8%A6%E5%8F%82%E6%95%B0%E7%9A%84%E4%BA%8C%E7%BB%B4%E7%A0%81) | 生成带参数的二维码接口 |

我们要做的就是通过一种方式创建ajax请求即可：

https://api.weibo.com/2/statuses/public\_timeline.json

所以我在实际设计Hybrid交互模型时，是以接口为单位进行设计的，比如获取通讯录的总体交互是：



**格式约定**

交互的第一步是设计数据格式，这里分为请求数据格式与响应数据格式，参考ajax的请求模型大概是：

$.ajax(options) ⇒ XMLHttpRequest

type (默认值："GET") HTTP的请求方法(“GET”, “POST”, or other)。

url (默认值：当前url) 请求的url地址。

data (默认值：none) 请求中包含的数据，对于GET请求来说，这是包含查询字符串的url地址，如果是包含的是object的话，$.param会将其转化成string。

所以我这边与Native约定的请求模型是：

[复制代码](javascript:void(0);)

requestHybrid({

//创建一个新的webview对话框窗口

tagname: 'hybridapi',

//请求参数，会被Native使用

param: {},

//Native处理成功后回调前端的方法

callback: function (data) {

}

});

[复制代码](javascript:void(0);)

这个方法执行会形成一个URL，比如：

hybridschema://hybridapi?callback=hybrid\_1446276509894&param=%7B%22data1%22%3A1%2C%22data2%22%3A2%7D

这里提一点，APP安装后会在手机上注册一个schema，比如淘宝是taobao://，Native会有一个进程监控Webview发出的所有schema://请求，然后分发到“控制器”hybridapi处理程序，Native控制器处理时会需要param提供的参数（encode过），处理结束后将携带数据获取Webview window对象中的callback（hybrid\_1446276509894）调用之

数据返回的格式约定是：

{

data: {},

errno: 0,

msg: "success"

}

真实的数据在data对象中，如果errno不为0的话，便需要提示msg，这里举个例子如果错误码1代表该接口需要升级app才能使用的话：

{

data: {},

errno: 1,

msg: "APP版本过低，请升级APP版本"

}

**代码实现**

这里给一个简单的代码实现，真实代码在APP中会有所变化：

[复制代码](javascript:void(0);)

1 window.Hybrid = window.Hybrid || {};

2 var bridgePostMsg = function (url) {

3 if ($.os.ios) {

4 window.location = url;

5 } else {

6 var ifr = $('<iframe style="display: none;" src="' + url + '"/>');

7 $('body').append(ifr);

8 setTimeout(function () {

9 ifr.remove();

10 }, 1000)

11 }

12 };

13 var \_getHybridUrl = function (params) {

14 var k, paramStr = '', url = 'scheme://';

15 url += params.tagname + '?t=' + new Date().getTime(); //时间戳，防止url不起效

16 if (params.callback) {

17 url += '&callback=' + params.callback;

18 delete params.callback;

19 }

20 if (params.param) {

21 paramStr = typeof params.param == 'object' ? JSON.stringify(params.param) : params.param;

22 url += '&param=' + encodeURIComponent(paramStr);

23 }

24 return url;

25 };

26 var requestHybrid = function (params) {

27 //生成唯一执行函数，执行后销毁

28 var tt = (new Date().getTime());

29 var t = 'hybrid\_' + tt;

30 var tmpFn;

31

32 //处理有回调的情况

33 if (params.callback) {

34 tmpFn = params.callback;

35 params.callback = t;

36 window.Hybrid[t] = function (data) {

37 tmpFn(data);

38 delete window.Hybrid[t];

39 }

40 }

41 bridgePostMsg(\_getHybridUrl(params));

42 };

43 //获取版本信息，约定APP的navigator.userAgent版本包含版本信息：scheme/xx.xx.xx

44 var getHybridInfo = function () {

45 var platform\_version = {};

46 var na = navigator.userAgent;

47 var info = na.match(/scheme\/\d\.\d\.\d/);

48

49 if (info && info[0]) {

50 info = info[0].split('/');

51 if (info && info.length == 2) {

52 platform\_version.platform = info[0];

53 platform\_version.version = info[1];

54 }

55 }

56 return platform\_version;

57 };

[复制代码](javascript:void(0);)

因为Native对于H5来是底层，框架&底层一般来说是不会关注业务实现的，所以真实业务中Native调用H5场景较少，这里不予关注了。

**常用交互API**

良好的交互设计是成功的第一步，在真实业务开发中有一些API一定会用到。

**跳转**

跳转是Hybrid必用API之一，对前端来说有以下跳转：

① 页面内跳转，与Hybrid无关

② H5跳转Native界面

③ H5新开Webview跳转H5页面，一般为做页面动画切换

如果要使用动画，按业务来说有向前与向后两种，forward&back，所以约定如下，首先是H5跳Native某一个页面

[复制代码](javascript:void(0);)

1 //H5跳Native页面

2 //=>baidubus://forward?t=1446297487682&param=%7B%22topage%22%3A%22home%22%2C%22type%22%3A%22h2n%22%2C%22data2%22%3A2%7D

3 requestHybrid({

4 tagname: 'forward',

5 param: {

6 //要去到的页面

7 topage: 'home',

8 //跳转方式，H5跳Native

9 type: 'native',

10 //其它参数

11 data2: 2

12 }

13 });

[复制代码](javascript:void(0);)

比如携程H5页面要去到酒店Native某一个页面可以这样：

[复制代码](javascript:void(0);)

1 //=>schema://forward?t=1446297653344&param=%7B%22topage%22%3A%22hotel%2Fdetail%20%20%22%2C%22type%22%3A%22h2n%22%2C%22id%22%3A20151031%7D

2 requestHybrid({

3 tagname: 'forward',

4 param: {

5 //要去到的页面

6 topage: 'hotel/detail',

7 //跳转方式，H5跳Native

8 type: 'native',

9 //其它参数

10 id: 20151031

11 }

12 });

[复制代码](javascript:void(0);)

比如H5新开Webview的方式跳转H5页面便可以这样：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'forward',

3 param: {

4 //要去到的页面，首先找到hotel频道，然后定位到detail模块

5 topage: 'hotel/detail ',

6 //跳转方式，H5新开Webview跳转，最后装载H5页面

7 type: 'webview',

8 //其它参数

9 id: 20151031

10 }

11 });

[复制代码](javascript:void(0);)

back与forward一致，我们甚至会有animattype参数决定切换页面时的动画效果，真实使用时可能会封装全局方法略去tagname的细节，这时就和糯米对外释放的接口差不多了。

**Header 组件的设计**

最初我其实是抵制使用Native提供的UI组件的，尤其是Header，因为平台化后，Native每次改动都很慎重并且响应很慢，但是出于两点核心因素考虑，我基本放弃了抵抗：

① 其它主流容器都是这么做的，比如微信、手机百度、携程

② 没有header一旦网络出错出现白屏，APP将陷入假死状态，这是不可接受的，而一般的解决方案都太业务了

PS：Native吊起Native时，如果300ms没有响应需要出loading组件，避免白屏

因为H5站点本来就有Header组件，站在前端框架层来说，需要确保业务的代码是一致的，所有的差异需要在框架层做到透明化，简单来说Header的设计需要遵循：

① H5 header组件与Native提供的header组件使用调用层接口一致

② 前端框架层根据环境判断选择应该使用H5的header组件抑或Native的header组件

一般来说header组件需要完成以下功能：

① header左侧与右侧可配置，显示为文字或者图标（这里要求header实现主流图标，并且也可由业务控制图标），并需要控制其点击回调

② header的title可设置为单标题或者主标题、子标题类型，并且可配置lefticon与righticon（icon居中）

③ 满足一些特殊配置，比如标签类header

所以，站在前端业务方来说，header的使用方式为（其中tagname是不允许重复的）：

[复制代码](javascript:void(0);)

1 //Native以及前端框架会对特殊tagname的标识做默认回调，如果未注册callback，或者点击回调callback无返回则执行默认方法

2 // back前端默认执行History.back，如果不可后退则回到指定URL，Native如果检测到不可后退则返回Naive大首页

3 // home前端默认返回指定URL，Native默认返回大首页

4 this.header.set({

5 left: [

6 {

7 //如果出现value字段，则默认不使用icon

8 tagname: 'back',

9 value: '回退',

10 //如果设置了lefticon或者righticon，则显示icon

11 //native会提供常用图标icon映射，如果找不到，便会去当前业务频道专用目录获取图标

12 lefticon: 'back',

13 callback: function () { }

14 }

15 ],

16 right: [

17 {

18 //默认icon为tagname，这里为icon

19 tagname: 'search',

20 callback: function () { }

21 },

22 //自定义图标

23 {

24 tagname: 'me',

25 //会去hotel频道存储静态header图标资源目录搜寻该图标，没有便使用默认图标

26 icon: 'hotel/me.png',

27 callback: function () { }

28 }

29 ],

30 title: 'title',

31 //显示主标题，子标题的场景

32 title: ['title', 'subtitle'],

33

34 //定制化title

35 title: {

36 value: 'title',

37 //标题右边图标

38 righticon: 'down', //也可以设置lefticon

39 //标题类型，默认为空，设置的话需要特殊处理

40 //type: 'tabs',

41 //点击标题时的回调，默认为空

42 callback: function () { }

43 }

44 });

[复制代码](javascript:void(0);)

因为Header左边一般来说只有一个按钮，所以其对象可以使用这种形式：

[复制代码](javascript:void(0);)

1 this.header.set({

2 back: function () { },

3 title: ''

4 });

5 //语法糖=>

6 this.header.set({

7 left: [{

8 tagname: 'back',

9 callback: function(){}

10 }],

11 title: '',

12 });

[复制代码](javascript:void(0);)

为完成Native端的实现，这里会新增两个接口，向Native注册事件，以及注销事件：

[复制代码](javascript:void(0);)

1 var registerHybridCallback = function (ns, name, callback) {

2 if(!window.Hybrid[ns]) window.Hybrid[ns] = {};

3 window.Hybrid[ns][name] = callback;

4 };

5

6 var unRegisterHybridCallback = function (ns) {

7 if(!window.Hybrid[ns]) return;

8 delete window.Hybrid[ns];

9 };

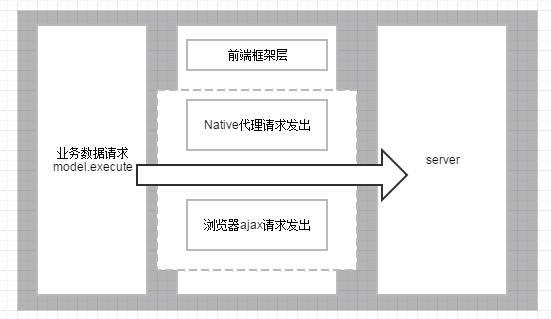
[复制代码](javascript:void(0);)

Native Header组件的实现：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif Native Header组件的封装

**请求类**

虽然get类请求可以用jsonp的方式绕过跨域问题，但是post请求却是真正的拦路虎，为了安全性服务器设置cors会仅仅针对几个域名，Hybrid内嵌静态资源是通过file的方式读取，这种场景使用cors就不好使了，所以每个请求需要经过Native做一层代理发出去。



这个使用场景与Header组件一致，前端框架层必须做到对业务透明化，业务事实上不必关心这个请求是由浏览器发出还是由Native发出：

1 HybridGet = function (url, param, callback) {

2 };

3 HybridPost = function (url, param, callback) {

4 };

真实的业务场景，会将之封装到数据请求模块，在底层做适配，在H5站点下使用ajax请求，在Native内嵌时使用代理发出，与Native的约定为：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'ajax',

3 param: {

4 url: 'hotel/detail',

5 param: {},

6 //默认为get

7 type: 'post'

8 },

9 //响应后的回调

10 callback: function (data) { }

11 });

[复制代码](javascript:void(0);)

**常用NativeUI组件**

最后，Native会提供几个常用的Native级别的UI，比如loading加载层，比如toast消息框：

[复制代码](javascript:void(0);)

1 var HybridUI = {};

2 HybridUI.showLoading();

3 //=>

4 requestHybrid({

5 tagname: 'showLoading'

6 });

7

8 HybridUI.showToast({

9 title: '111',

10 //几秒后自动关闭提示框，-1需要点击才会关闭

11 hidesec: 3,

12 //弹出层关闭时的回调

13 callback: function () { }

14 });

15 //=>

16 requestHybrid({

17 tagname: 'showToast',

18 param: {

19 title: '111',

20 hidesec: 3,

21 callback: function () { }

22 }

23 });

[复制代码](javascript:void(0);)

Native UI与前端UI不容易打通，所以在真实业务开发过程中，一般只会使用几个关键的Native UI。

**账号系统的设计**

根据上面的设计，我们约定在Hybrid中请求有两种发出方式：

① 如果是webview访问线上站点的话，直接使用传统ajax发出

② 如果是file的形式读取Native本地资源的话，请求由Native代理发出

因为静态html资源没有鉴权的问题，真正的权限验证需要请求服务器api响应通过错误码才能获得，这是动态语言与静态语言做入口页面的一个很大的区别。

以网页的方式访问，账号登录与否由是否带有秘钥cookie决定（这时并不能保证秘钥的有效性），因为Native不关注业务实现，而每次载入都有可能是登录成功跳回来的结果，所以每次载入后都需要关注秘钥cookie变化，以做到登录态数据一致性。

以file的方式访问内嵌资源的话，因为API请求控制方为Native，所以鉴权的工作完全由Native完成，接口访问如果没有登录便弹出Native级别登录框引导登录即可，每次访问webview将账号信息种入到webview中，这里有个矛盾点是Native种入webview的时机，因为有可能是网页注销的情况，所以这里的逻辑是：

① webview载入结束

② Native检测webview是否包含账号cookie信息

③ 如果不包含则种入cookie，如果包含则检测与Native账号信息是否相同，不同则替换自身

④ 如果检测到跳到了注销账户的页面，则需要清理自身账号信息

如果登录不统一会就会出现上述复杂的逻辑，所以真实情况下我们会对登录接口收口。

**简单化账号接口**

平台层面觉得上述操作过于复杂，便强制要求在Hybrid容器中只能使用Native接口进行登录和登出，前端框架在底层做适配，保证上层业务的透明，这样情况会简单很多：

① 使用Native代理做请求接口，如果没有登录直接Native层唤起登录框

② 直连方式使用ajax请求接口，如果没有登录则在底层唤起登录框（需要前端框架支持）

简单的登录登出接口实现：

[复制代码](javascript:void(0);)

1 /\*

2 无论成功与否皆会关闭登录框

3 参数包括：

4 success 登录成功的回调

5 error 登录失败的回调

6 url 如果没有设置success，或者success执行后没有返回true，则默认跳往此url

7 \*/

8 HybridUI.Login = function (opts) {

9 };

10 //=>

11 requestHybrid({

12 tagname: 'login',

13 param: {

14 success: function () { },

15 error: function () { },

16 url: '...'

17 }

18 });

19 //与登录接口一致，参数一致

20 HybridUI.logout = function () {

21 };

[复制代码](javascript:void(0);)

**账号信息获取**

在实际的业务开发中，判断用户是否登录、获取用户基本信息的需求比比皆是，所以这里必须保证Hybrid开发模式与H5开发模式保持统一，否则需要在业务代码中做很多无谓的判断，我们在前端框架会封装一个User模块，主要接口包括：

1 var User = {};

2 User.isLogin = function () { };

3 User.getInfo = function () { };

这个代码的底层实现分为前端实现，Native实现，首先是前端的做法是：

当前端页面载入后，会做一次异步请求，请求用户相关数据，如果是登录状态便能获取数据存于localstorage中，这里**一定不能存取敏感信息**

前端使用localstorage的话需要考虑极端情况下使用内存变量的方式替换localstorage的实现，否则会出现不可使用的情况，而后续的访问皆是使用localstorage中的数据做判断依据，以下情况需要清理localstorage的账号数据：

① 系统登出

② 访问接口提示需要登录

③ 调用登录接口

这种模式多用于单页应用，非单页应用一般会在每次刷新页面先清空账号信息再异步拉取，但是如果当前页面马上就需要判断用户登录数据的话，便不可靠了;处于Hybrid容器中时，因为Native本身就保存了用户信息，封装的接口直接由Native获取即可，这块比较靠谱。

**Hybrid的资源**

**目录结构**

Hybrid技术既然是将静态资源存于Native，那么就需要目录设计，经过之前的经验，目录结构一般以2层目录划分：



如果我们有两个频道酒店与机票，那么目录结构是这样的：

[复制代码](javascript:void(0);)

1 webapp //根目录

2 ├─flight

3 ├─hotel //酒店频道

4 │ │ index.html //业务入口html资源，如果不是单页应用会有多个入口

5 │ │ main.js //业务所有js资源打包

6 │ │

7 │ └─static //静态样式资源

8 │ ├─css

9 │ ├─hybrid //存储业务定制化类Native Header图标

10 │ └─images

11 ├─libs

12 │ libs.js //框架所有js资源打包

13 │

14 └─static

15 ├─css

16 └─images

[复制代码](javascript:void(0);)

最初设计的forward跳转中的topage参数规则是：频道/具体页面=>channel/page，其余资源会由index.html这个入口文件带出。

**增量机制**

真实的增量机制需要服务器端的配合，我这里只能简单描述，Native端会有维护一个版本映射表：

[复制代码](javascript:void(0);)

{

flight: 1.0.0,

hotel: 1.0.0,

libs: 1.0.0,

static: 1.0.0

}

[复制代码](javascript:void(0);)

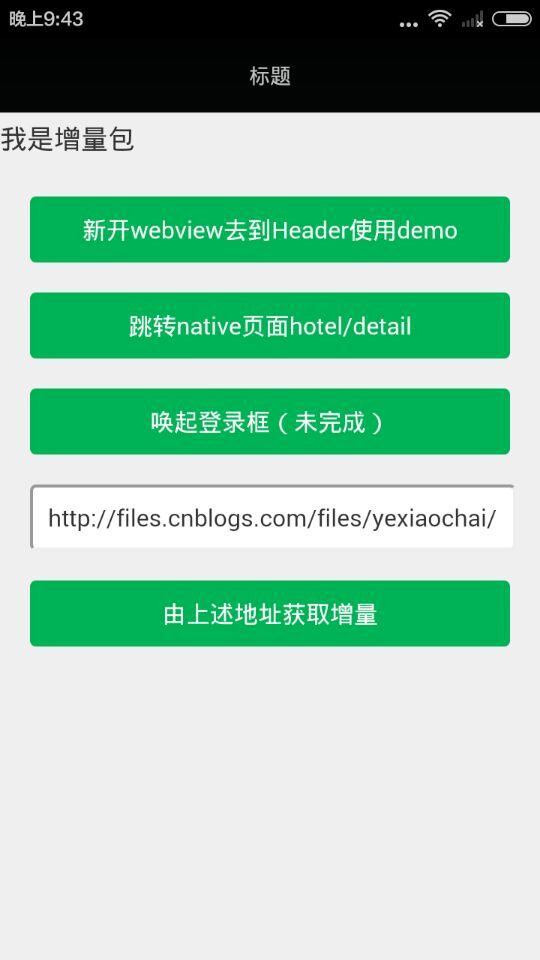
这个映射表是每次大版本APP发布时由服务器端生成的，如果酒店频道需要在线做增量发布的话，会打包一个与线上一致的文件目录，走发布平台发布，会在数据库中形成一条记录：

|  |  |  |
| --- | --- | --- |
| channel | ver | md5 |
| flight | 1.0.0 | 1245355335 |
| hotel | 1.0.1 | 455ettdggd |

当APP启动时，APP会读取版本信息，这里发现hotel的本地版本号比线上的小，便会下载md5对应的zip文件，然后解压之并且替换整个hotel文件，本次增量结束，因为所有的版本文件不会重复，APP回滚时可用回到任意想去的版本，也可以对任意版本做BUG修复。

**结语**

github上代码会持续更新，现在界面反正不太好看，大家多多包涵吧，这里是一些效果图：



Hybrid方案是快速迭代项目，快速占领市场的神器，希望此文能对准备接触Hybrid技术的朋友提供一些帮助，并且再次感谢明月同学的配合。

**微博求粉**

最后，我的微博粉丝及其少，如果您觉得这篇博客对您哪怕有一丝丝的帮助，微博求粉博客求赞！！！

[**浅谈Hybrid技术的设计与实现第二弹**](http://www.cnblogs.com/yexiaochai/p/5524783.html)

**前言**

[浅谈Hybrid技术的设计与实现](http://www.cnblogs.com/yexiaochai/p/4921635.html)

[浅谈Hybrid技术的设计与实现第二弹](http://www.cnblogs.com/yexiaochai/p/5524783.html)

[浅谈Hybrid技术的设计与实现第三弹——落地篇](http://www.cnblogs.com/yexiaochai/p/5813248.html)

接上文：[浅谈Hybrid技术的设计与实现](http://www.cnblogs.com/yexiaochai/p/4921635.html)（阅读本文前，建议阅读这个先）

上文说了很多关于Hybrid的概要设计，可以算得上大而全，有说明有demo有代码，对于想接触Hybrid的朋友来说应该有一定帮助，但是对于进阶的朋友可能就不太满足了，他们会想了解其中的每一个细节，甚至是一些Native的实现，小钗这里继续抛砖引玉，希望接下来的内容对各位有一定帮助。

进入今天的内容之前我们首先谈谈两个相关技术Ionic与React Native。

Ionic是一个基于Cordova的移动开发框架，他的一大优势是有一套配套的前端框架，因为是基于angularJS的，又提供了大量可用UI，可谓大而全，对开发效率很有帮助；与我们所说的Hybrid不同的是，我们的Native容器是由公司或者个人定制开发，Ionic的Native壳是第三方公司做的平台性产品，优劣不论，但是楼主是绝不会用太多第三方开源的东西的，举个例子来说，你的项目中如果第三方UI组件越多，那么性能和风险相对会越多，因为你不了解他。另一方面angular对于H5来说，尺寸着实过大，最后以逼格来说，Ionic还是多适用于外包了。

与Ionic不同的是React Native，根据他写出来的View完全是Native的View，那么这个逼格和体验就高了，小钗在这次Hybrid项目结束后，应该会着力在这方面做研究，这里没实际经验就不多说了。

文中是我个人的一些开发经验，希望对各位有用，也希望各位**多多支持讨论**，指出文中**不足**以及提出您的一些**建议**。

**设计类博客（还有最后一篇便完结）**

<http://www.cnblogs.com/yexiaochai/p/4921635.html>  
<http://www.cnblogs.com/yexiaochai/p/5524783.html>

**ios博客（持续更新）**

<http://www.cnblogs.com/nildog/p/5536081.html#3440931>

文中IOS代码由我现在的同事[Nil（http://www.cnblogs.com/nildog/p/5536081.html）](http://www.cnblogs.com/nildog/p/5536081.html)提供，感谢Nil对项目的支持。

之前Android代码由明月提供，后续明月也会持续支援我们Android的实现，感谢明月。

**代码地址：**[**https://github.com/yexiaochai/Hybrid**](https://github.com/yexiaochai/Hybrid)

因为IOS不能扫码下载了，大家自己下载下来用模拟器看吧，下面开始今天的内容。

**H5与Native通信**

**Url Schema**

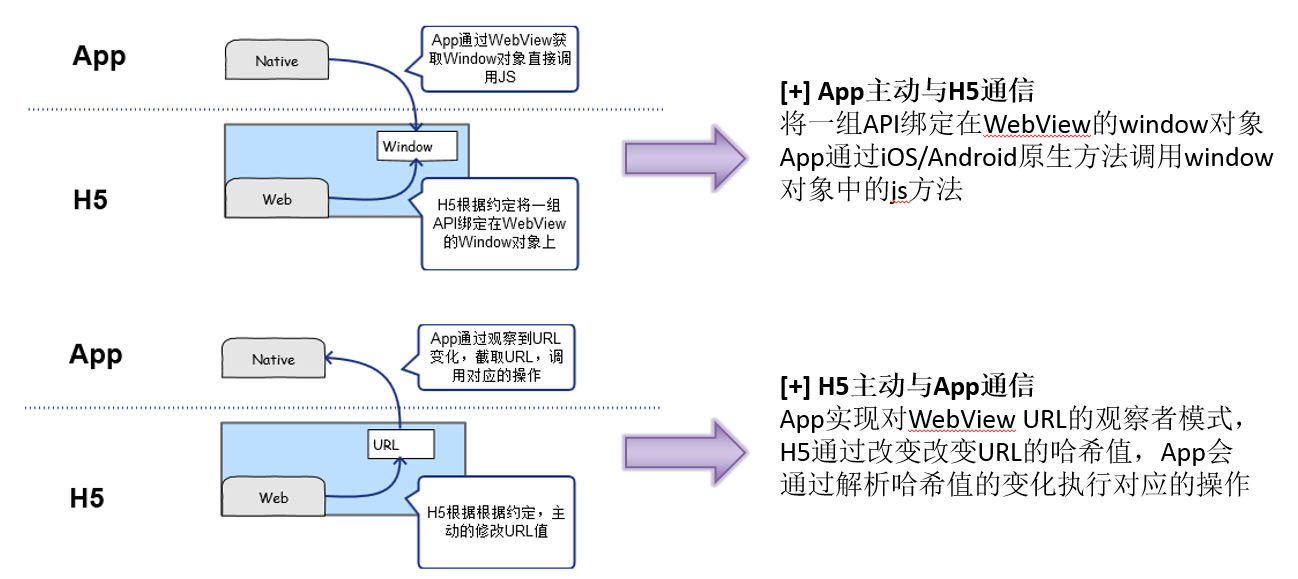
根据之前的知识，H5与Native交互的桥梁为Webview，而“联系”的方式是以url schema的方式做的，在用户安装app后，app可以自定义url schema，并且把自定义的url注册在调度中心， 例如

* ctrip://wireless 打开携程App
* weixin:// 打开微信

事实上Native能捕捉webview发出的一切请求，所以就算这里不是这种协议，Native也能捕捉，这个协议的意义在于可以在浏览器中直接打开APP，相关文献为：

[**又到周末了，我们一起来研究【浏览器如何检测是否安装app】吧**](http://www.cnblogs.com/yexiaochai/p/3439179.html)

这里盗用一张之前的交互模型图，确实懒得画新的了：



我们在H5获取Native方法时一般是会构造一个这样的请求，使用iframe发出（设置location会有多次请求覆盖的问题）：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 //创建一个新的webview对话框窗口

3 tagname: 'hybridapi',

4 //请求参数，会被Native使用

5 param: {},

6 //Native处理成功后回调前端的方法

7 callback: function (data) {

8 }

9 });

10 //=====>

11 hybridschema://hybridapi?callback=hybrid\_1446276509894&param=%7B%22data1%22%3A1%2C%22data2%22%3A2%7D

[复制代码](javascript:void(0);)

多数情况下这种方式没有问题，但是我们在后续的开发中，为了统一鉴权，将所有的请求全部代理到了Native发出，比如这样：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'post',

3 param: {

4 url: 'http://api.kuai.baidu.com/city/getstartcitys',

5 param1: 'param1',

6 param2: 'param2'

7 },

8 callback: function(data) {

9 }

10 });

[复制代码](javascript:void(0);)

请注意，这里可是POST请求，这里首先考虑的是长度限制，毕竟这个是由iframe的src设置的，虽然各个浏览器不一样，但必定会收到长度限制（2k），针对这个问题我咨询了糯米以及携程的Hybrid底层团队，得到了比较零星的回答：

① 移动端一般来说不会有这么长的请求（这个在理）

② 我们不支持IOS6了，现在用的JavaScriptCore

上面的答复不太满意，于是我尝试在页面上放一个全局变量（或者文本框）以解决参数过大的问题，而当我尝试解决的时候，产品告诉我：我们早不支持IOS6了！

如果只用支持chrome用户，那么坚决不支持IE的！抱着这一想法，小钗也就放弃IOS6了

如果不支持IOS6，那么事情似乎变得好办多了。

**JavaScriptCore**

在ios7后，Apple新增了一个JavaScriptCore让Native可以与H5更好的交互（Android早就有了），我们这里主要讨论js如何与Native通信，这里举一个简单的例子：

PS：楼主对ios不熟，这段代码引至https://hjgitbook.gitbooks.io/ios/content/04-technical-research/04-javascriptcore-note.html

① 首先定义一个js方法，这里注意其中调用了一个没有声明的方法：

function printHello() {

**//未声明方法**

print("Hello, World!");

}

然后，上述未声明方法事实上是Native注入给window对象的：

[复制代码](javascript:void(0);)

1 NSString \*scriptPath = [[NSBundle mainBundle] pathForResource:@"hello" ofType:@"js"];

2 NSString \*scriptString = [NSString stringWithContentsOfFile:scriptPath encoding:NSUTF8StringEncoding error:nil];

3

4 JSContext \*context = [[JSContext alloc] init];

5 [context evaluateScript:scriptString];

6

**7 self.context[@"print"] = ^(NSString \*text) {**

**8 NSLog(@"%@", text");**

**9 };**

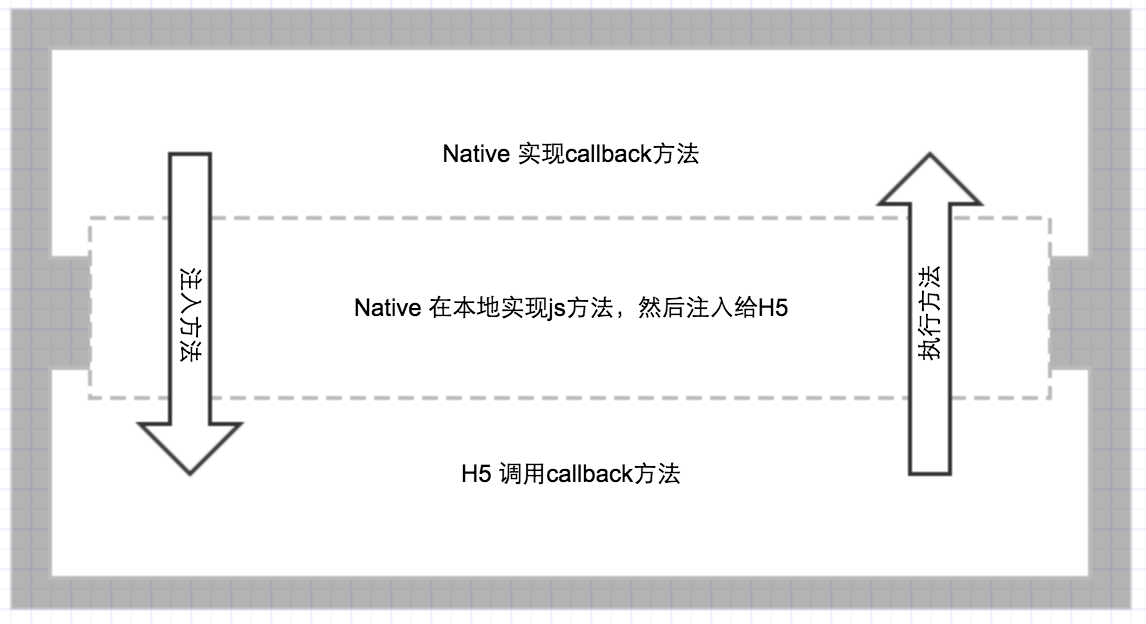
10

11 JSValue \*function = self.context[@"printHello"];

12 [function callWithArguments:@[]];

[复制代码](javascript:void(0);)

这个样子，JavaScript就可以调用Native的方法了，这里Native需要注意方法注入的时机，一般是一旦载入页面便需要载入变量，这里的交互模型是：



于是，我们这里只需要将原来底层的通信一块改下即可（Android本身就支持类似的实现，这里暂时不予关注）：

[复制代码](javascript:void(0);)

1 //使用jsCore与native通信

2 window.requestNative && requestNative(JSON.stringify(params));

3 return;

4 //兼容ios6

5 var ifr = $('<iframe style="display: none;" src="' + url + '"/>');

6 $('body').append(ifr);

7 setTimeout(function () {

8 ifr.remove();

9 ifr = null;

10 }, 1000)

[复制代码](javascript:void(0);)

**优劣**

URL Schema与JavaScriptCore的优劣不好说，还是看具体使用场景吧，不考虑参数问题的话，真正有使用经验的朋友可能会发现url schema方案可能更加适用，因为：

URL Schema方案是监控Webview请求，所以比较通用；而JavaScriptCore的注入是在Webview加载时候注入。  
如果页面刷新，这个注入也就丢了需要做特殊处理（糯米接入一个常见BUG就是糯米容器提供的方法不执行）

使用JavaScriptCore的话，页面刷新会导致Hybrid项目瘫痪的问题，我们IOS同事首先调整了注入方法的时间点，放到了webViewDidFinishLoad中，因为webViewDidFinishLoad的注入在页面js声明之后，所以如果一来就有Hybrid的交互便不会执行，比如：

1 //如果一开始便设置的话，将因为Native没有注入而不执行

2 Hybrid.ui.header.set({

3 title: '设置右边按钮',

4 });

所以我与Native约定在webViewDidFinishLoad后执行一个我定义的方法，我们将页面初始化逻辑放到这个事件里面，比如：

1 Hybrid.ready = function() {

2 hybridInit();

3 }

对比这个方法与之前jQuery的dom ready有点类似，我们可能会担心这样会影响页面的渲染速度，这里特别做了一个测试，这段代码对真实逻辑执行确实有一定影响，首次启动在30-90ms之间，第二次没什么影响了，这里也形成了一个一个优化点，只将那种页面一加载结束就需要执行的逻辑放入其中，影响主页面的逻辑可优先执行，如果觉得麻烦便直接将页面的载入放到这个方法中即可。

**选择建议**

根据我们的使用过程，发现JavaScriptCore还是不好用，因为对Native的不熟悉，在js方法注入的时间点一块我们踩了一些坑，我们想在webViewDidFinishLoad中注入方法，但是发现有一定几率是页面js已经执行完了才注入，导致Hybrid交互失效。

而且我们对Native一块声明js方法的生命周期与垃圾回收一块也不熟悉，总担心埋下什么坑，加之之前30-90ms的延迟，我们最终是实现了两套方案：

一般情况下仍旧使用URL Schema，如果有不满足的场景，我们会使用JavaScriptCore，因为底层架构搭建不能耗费太多时间，所以对JavaScriptCore的研究便暂时到此，后续有时间需要对他做深入研究。

**Hybrid版本**

APP会有版本号概念，每个版本会加一些新的特性或者会改一些BUG，一般的版本号是1.0.0，如果改了BUG打了补丁是1.0.1，有新特性就是1.1.0，如果有大改变的话就2.0.0咯，我们在实际业务代码中可能会有用到版本号的地方，所以Native需要将当前版本号告诉我们，一般是采用Native篡改navigator.userAgent写入特殊标志实现，我们这里是写入了这种标识：

xxx hybrid\_1.0.0 xxx

然后我们会在全局释放一个工具类方法获取当前版本号：

[复制代码](javascript:void(0);)

1 var getHybridInfo = function () {

2 var platform\_version = {};

3 var na = navigator.userAgent;

4 na = na.toLowerCase();

5 var info = na.match(/hybrid\_\d\.\d\.\d/);

6 if (info && info[0]) {

7 info = info[0].split('\_');

8 if (info && info.length == 2) {

9 platform\_version.platform = info[0];

10 platform\_version.version = info[1];

11 }

12 }

13 return platform\_version;

14 };

[复制代码](javascript:void(0);)

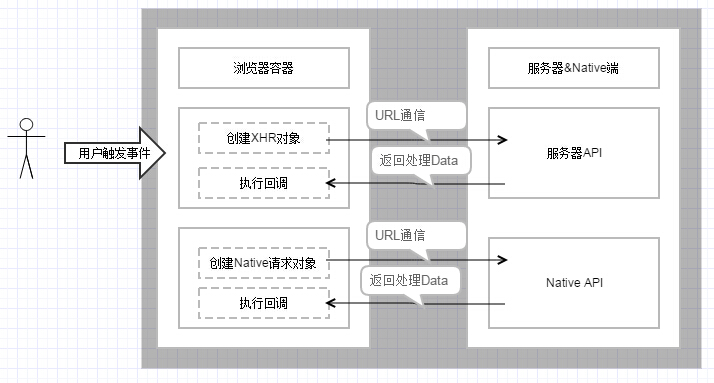
于是，我们在业务开发中便能知道当前是不是处于Native容器中，和获取版本号。

根据之前的共识，我们的代码只要运行在Native容器中就应该表现的像Hybrid，在浏览器中就应该表现的像H5

上面这句话可能很多朋友觉得有点奇怪，这里的界限在于有些方法H5提供了Native也提供了，究竟该用哪个的问题，比如获取当前位置信息，如果在Native容器中自然走Native获取，如果在浏览器中那就走H5接口。

**交互格式约定**

做一件事情重中之重的就是基础约定，我们这里做Hybrid架构首先就要做好交互格式约定，这种格式约定的要灵活一点，这个将会在后续扩展中提现他的优势，我们这里依旧采用类似Ajax的交互规则：



**请求格式**

[复制代码](javascript:void(0);)

1 requestHybrid({

2 //创建一个新的webview对话框窗口

3 tagname: 'hybridapi',

4 //请求参数，会被Native使用

5 param: {},

6 //Native处理成功后回调前端的方法

7 callback: function (data) {

8 }

9 });

[复制代码](javascript:void(0);)

tagname是标志这次请求的唯一标识，在接口比较多的情况有可能会有命名空间，比如：tagname: 'ns/api'。

**回调格式**

回调的方式都是Native调用H5的js方法，前端需要告诉Native去哪个对象拿回调方法，另外前端需要与Native约定返回时所带的参数，我们是这样设计的：

{

data: {},

errno: 0,

msg: "success"

}

其中每个错误码需要详细的约定，比如：

{

data: {},

errno: 1,

msg: "APP版本过低，请升级APP版本"

}

但是真实业务调用的时候却不需要特别去处理响应数据，因为前端应该有统一的地方处理，到具体业务回调时应该只需要使用data数据即可。

**调用方式的困惑**

一般来说，H5与Native通信都只会使用一个方法，我们之前是H5创建url schema，后面有有了新的方案，是Native释放一个requestNative给H5调用，这里就产生了一个之前没有的问题：

之前Native是没有能力将具体API方法注入给H5，所以我们使用唯一的方法传递tagname给Native，Native底层会使用类似反射的方式执行他的逻辑，这个tagname可以理解为方法名，而现在Native是有能力为前端注入所有需要的方法了，比如：



意思是之前要根据url schema然后native捕捉请求后，获取tagname字符串，再映射到具体NativeAPI，而现在Native已经有能力将这些Native API建立一个映射函数，注入给H5，所以H5可以直接调用这些方法了，实际的例子是：

[复制代码](javascript:void(0);)

1 //所有请求交互收口到一个方法,方法内部再做具体处理

2 requestHybrid({

3 tagname: 'getAdress',

4 param: {

5 param: 'param'

6 },

7 callback: function(data){}

8 });

9

10 //每个请求交互独立调用Native注入接口

11 hybrid.getAdress({

12 param: {

13 param: 'param'

14 },

15 callback: function(data){}

16 });

[复制代码](javascript:void(0);)

这里可以各位需要产生一个思考，方案一与方案二到底选哪个？这个时候就要多考虑框架的扩展性了，一旦有机会“收口”的都要考虑 “收口”，我们对某一类方法应该有统一的收口的地方，以便处理我们一些公共的逻辑，比如：

① 前端要对每个接口调用的次数打点

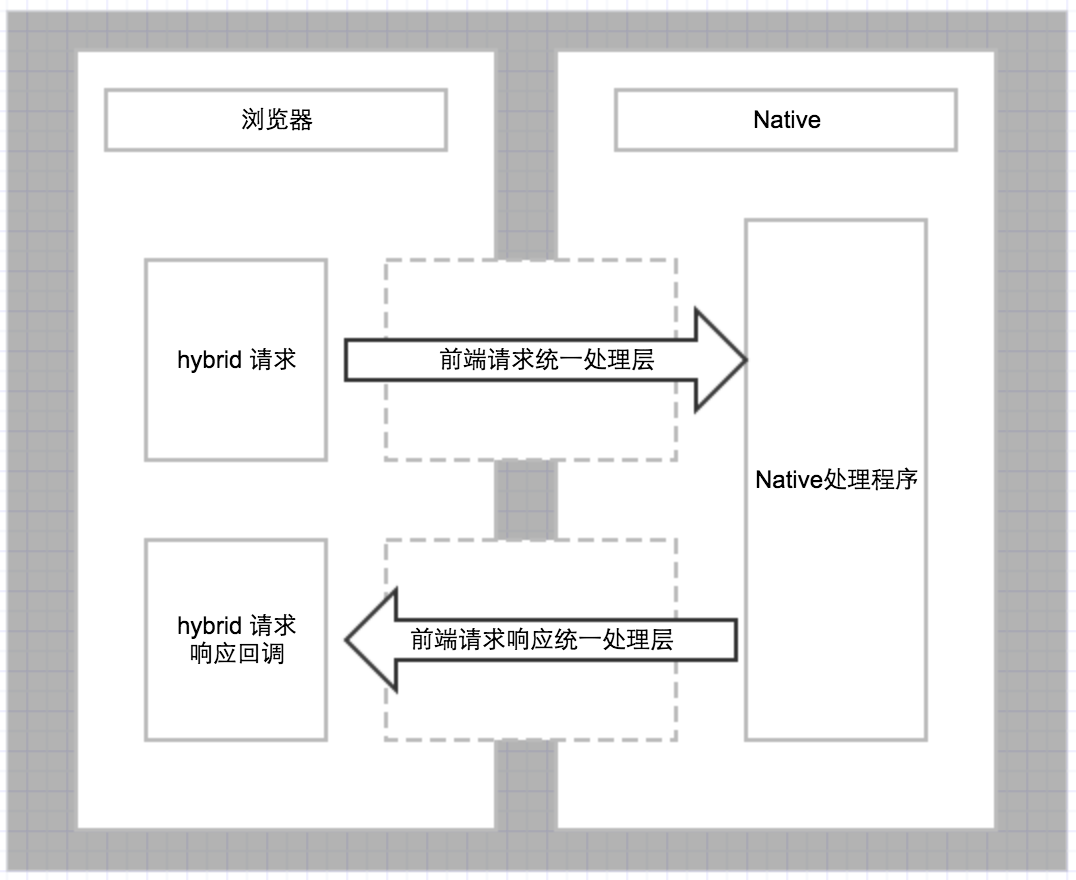
② 前端要对参数做统一处理

③ 我们突然要在底层改变与APP桥接的方式，不能走JavaScriptCore了（我们就实际遇到了这个问题）

④ 前端要为Native返回的错误码做统一封装

......

一个比较好的交互事实上是这样的，请求的时候要通过一个地方的处理，回调的时候也需要通过一个地方的处理，而这个地方便是我们能统一把关与控制的地方了，正如我们对ajax的收口，对settimeout的收口是一个道理：



**跳转**

无论什么系统，一个最重要的功能就是跳转，跳转设计的好坏很大程度决定你的框架好不好，好的跳转设计可以省下业务很多功夫，对迭代扩展也很有帮助，对于Hybrid来说，跳转可能会有：

① Native跳H5

② H5跳Native

③ H5跳H5（这里还要分内嵌的场景）

④ H5新开Webview打开H5

......

一般来说，Native跳H5事实上是用不着我们关注的，但是有一类Native跳转我们还不得不关注。

**入口类跳转**

所谓入口类跳转有以下特点：

① 一个入口往往会跳到一个独立的频道

② 每个独立的入口的实现首页关注不了

③ 频道可能是Native的，也可能是Hybrid的

几个常见的情况是：

比如糯米的美食或者携程的酒店，美食是Hybrid的，酒店是Native的，而跳转实现是做到Native上的，需要考虑到他的灵活性，也就是我一次点击想去哪就去哪，这个自然需要数据库的支持。

事实上在这类“入口类”跳转模块，每个模块点击往哪里跳转可能server端会给他一个类似这样的数据结构：

[复制代码](javascript:void(0);)

1 //跳Native

2 {

3 topage: 'hotel/index',

4 type: 'native'

5 }

6 //跳转H5

7 {

8 topage: 'https://mdianying.baidu.com',

9 type: 'h5'

10 }

[复制代码](javascript:void(0);)

当然，上述只是可能的数据结构，根据之前我们的实现，更有可能是这个样子，直接只是一个URL：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'forward',

3 param: {

4 topage: 'train/index.html',

5 type: 'h5'

6 }

7 });

8 //=>

9 hybrid://forward?param=%7B%22topage%22%3A%22hotel%2Findex.html%22%2C%22type%22%3A%22h5%22%7D

[复制代码](javascript:void(0);)

以这个做法来说，无论是怎么跳转，仍然可以统一将实现封装到forward的实现中。

如果你使用的是JavaScriptCore，URL Schema依旧要保留以处理这类跳转或者外部浏览器打开APP的需求，有时候当一种方案坑的时候才能体现另一种的可贵。

**动画约定**

Native体验好，其中一个原因就是有过场动画，我们这里约定了四种基本的动画效果：

[复制代码](javascript:void(0);)

//默认动画push 左进

requestHybrid({

tagname: 'forward',

param: {

topage: 'index2',

type: 'native'

}

});

//右出

requestHybrid({

tagname: 'forward',

param: {

topage: 'index2',

type: 'native',

animate: 'pop'

}

});

//从下往上动画，这种关闭的时候会自动从上向下滑出

requestHybrid({

tagname: 'forward',

param: {

topage: 'index2',

type: 'native',

animate: 'present'

}

});

[复制代码](javascript:void(0);)

如果没有动画animate参数便设置为none即可。

**back**

因为要保证H5与Native的特性一致，Native的页面路径事实上也是与浏览器一致的，所以我们只需要保证Native中的back与浏览器中一样，意思是什么都不做......

requestHybrid({

tagname: 'back'

});

这里back在webview中会检查history的记录，如果大于1则后退，否则会退回上一步操作。我们可以看出，back的功能是很单一的，往往不能满足我们的需求，所以常常使用forward+pop动画当做back使用，而这一做法将引起令人头疼的history错乱问题!!!

**forward**

forward是非常重要的一个API，在Hybrid中情况又比较复杂，所以这块要花点力气多思考，设计的好不好直接影响业务开发的接受情感。

我之前做框架时会禁止业务开发使用label标签与a标签，这个举动也受到了一些质疑（往往是语义化）

其实并不是label标签和a标签不好，而是解决移动端300ms延迟可能会对label标签做特殊处理，容易引起一些莫名其妙的BUG；

而a标签更是破坏单页应用路由的最佳选手，很多同事为a标签添加点击事件（没有设置href）又没有阻止默认事件而导致意想不到的BUG

像这种时候，你与其给人一个个解释要如何做特殊处理，倒不如直接禁止他们使用来得快......

**H5跳Native**

H5跳Native比较简单，只需要与Native同事约定topage的页面即可

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'forward',

3 param: {

4 topage: 'index2',

5 type: 'native'

6 }

7 });

[复制代码](javascript:void(0);)

如果要带参数的话，便直接写到topage后面的参数上：

topage: 'index2?a=1&b=2',

这个写法显然是有点怪的，因为我们之前跳转是这样写的：

this.forward('index2', {

a: 1, b: 2

});

为了保证业务代码一致，我们只需要在前端底层封装forward方法即可，这个将生成这种url，根据我们url schema的约定，这个链接就会进入到Native对应页面：

hybrid://forward?param=%7B%22topage%22%3A%22index2%22%2C%22type%22%3A%22native%22%7D

**H5新开Webview跳H5**

本来H5跳H5走的是浏览器内部体系，但为增强体验会新开一个Webview做动画，尽可能的模拟Native交互，这个代码是这样的：

[复制代码](javascript:void(0);)

requestHybrid({

tagname: 'forward',

param: {

//flight/detail.html?id=111

//hotel/index.html

//http:www.baidu.com

topage: 'flight/index.html',

type: 'h5'

}

});

[复制代码](javascript:void(0);)

如果一个团队前端成体系的话，一般每个频道的代码是有规则的，一般是频道名/页面名，事实上每个topage都应该是一个完整的http的链接（如果前端传过去不是完整的，就需要Native补齐），这个也会封装到前端底层形成一个语法糖：

[复制代码](javascript:void(0);)

1 this.forward('flight/detail', {id: 1})

2 //==>

3 requestHybrid({

4 tagname: 'forward',

5 param: {

6 topage: 'http://domain.com/webapp/flight/detail.html?id=1',

7 type: 'h5'

8 }

9 });

[复制代码](javascript:void(0);)

这个是针对线上的场景，而如果读取的是内嵌资源的话就不是这么回事了，比如之前的代码：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'forward',

3 param: {

4 topage: 'flight/detail.html?id=1',

5 type: 'h5'

6 }

7 });

[复制代码](javascript:void(0);)

这个是告诉Native去静态资源flight目录找寻detail.html然后载入，这里又涉及到一个问题是：业务到底该怎么写代码？因为很多时候我们是一套H5代码浏览器与Native两边运行，而我如果在H5想从首页到列表页直接这样写就行了：

this.forward('list', {id: 1})

业务是绝不希望这个代码要因为接入Hybrid而改变，就算业务开发接受，也会因为跳转选择而导致业务混乱引发各种问题，所以前端框架层要解决这个问题，保证业务最小程度的犯错几率，上面之所以不传完整http的链接给Native，是因为会有静态资源内嵌Native的场景，请看下面的例子：

[复制代码](javascript:void(0);)

requestHybrid({

tagname: 'forward',

param: {

//Native首先检查本地有没有这个文件，如果有就直接读取本地，没有就走http

topage: 'flight/index.html',

type: 'native'

}

});

[复制代码](javascript:void(0);)

这里为解决快速渲染提出了第一个约定：

跳转时，需要Native去解析URL，判断是否有本地文件，如果有则加载本地文件

举个例子来说：

http://domain.com/webapp/flight/index.html

//解析url得出关键词

//===>

flight/index.html

检查本地是否有该文件，有便直接返回

有这个规则的话，就可以最大程度上保证业务代码的一致性，而读取本地文件也能大大提高性能，缓存这块我们后面再说。

**history错乱**

前面说了History错乱的原因一般来说是因为使用了forward模拟back回退，这种业务场景经常发生：

① 订单填写页需要去支付页面，由于一些特殊业务需求，需要经过一个中间页做处理，然后再进入真正的支付页，这个时候支付页点击后退事实上是执行的forward的操作（因为点击回退就到中间页了）

② 发布产品的时候会有发布1->发布2->发布预览->完成->产品详情页，这个时候点击产品详情页的后退，我们也不会希望他回到发布预览页，而是首页

③ 有可能用户直接由浏览器直接打开APP进入产品详情页，这个时候点击后退是没有任何记录的，当然也是回到首页了。

以上按照业务逻辑的知识的话是正确的，但是以第三种情况来说，如果回到首页后再次点击后退，而首页的后退又刚好是back操作，那么会回到产品详情页（事实上用户是想退出该频道），而更加不妙的是用户再次点击产品详情的回退又回到了首页，形成了一个死循环！！！

history错乱，暂时没有很好的处理办法，我们要做的是一旦发现可能会发生history错乱的频道就都不要使用back了，比如上面首页back就写死回到app大首页

当然，有些页面也不是无规律的乱跳的，所以我们新开一个页面的时候需要让新开页面知道之前是哪个页面，如果单页应用倒是可以写在实例对象上，但是一刷新就丢了，所以比较靠谱的做法也许是带在url上，这个在新开webview的场景下是不可避免的，比如：

[复制代码](javascript:void(0);)

//从a页面进入b页面

this.forward('b');

//b页面的实例

this.refer == 'a' //true

//因为页面刷新会丢失这个管理，所以我们将这个关联写在url上

//b的url webapp/project/b.html?refer=a

[复制代码](javascript:void(0);)

**Header组件**

H5开发中对Header部分的操作是不可避免的，于是我们抽象出了UIHeader组件处理这种操作，事实上在Hybrid中的Header也应该是一个通用组件，前端做的仅仅是根据约定的格式去调用这个组件即可，但是因为要保证H5与Native调用的一致性，所以要规范化业务代码的使用，一般的使用方法为：

[复制代码](javascript:void(0);)

1 //Native以及前端框架会对特殊tagname的标识做默认回调，如果未注册callback，或者点击回调callback无返回则执行默认方法

2 //back前端默认执行History.back，如果不可后退则回到指定URL，Native如果检测到不可后退则返回Naive大首页

3 //home前端默认返回指定URL，Native默认返回大首页

4 this.header.set({

5 left: [

6 {

7 //如果出现value字段，则默认不使用icon

8 tagname: 'back',

9 value: '回退',

10 //如果设置了lefticon或者righticon，则显示icon

11 //native会提供常用图标icon映射，如果找不到，便会去当前业务频道专用目录获取图标

12 lefticon: 'back',

13 callback: function () { }

14 }

15 ],

16 right: [

17 {

18 //默认icon为tagname，这里为icon

19 tagname: 'search',

20 callback: function () { }

21 },

22 //自定义图标

23 {

24 tagname: 'me',

25 //会去hotel频道存储静态header图标资源目录搜寻该图标，没有便使用默认图标

26 icon: 'hotel/me.png',

27 callback: function () { }

28 }

29 ],

30 title: 'title',

31 //显示主标题，子标题的场景

32 title: ['title', 'subtitle'],

33

34 //定制化title

35 title: {

36 value: 'title',

37 //标题右边图标

38 righticon: 'down', //也可以设置lefticon

39 //标题类型，默认为空，设置的话需要特殊处理

40 //type: 'tabs',

41 //点击标题时的回调，默认为空

42 callback: function () { }

43 }

44 });

[复制代码](javascript:void(0);)

因为一般来说左边只有一个返回相关的按钮，所以会提供一个语法糖（在底层依旧会还原为上面的形式）：

[复制代码](javascript:void(0);)

1 this.header.set({

2 left: [{

3 tagname: 'back',

4 callback: function(){}

5 }],

6 title: '',

7 });

8 //语法糖=>

9 this.header.set({

10 back: function () { },

11 title: ''

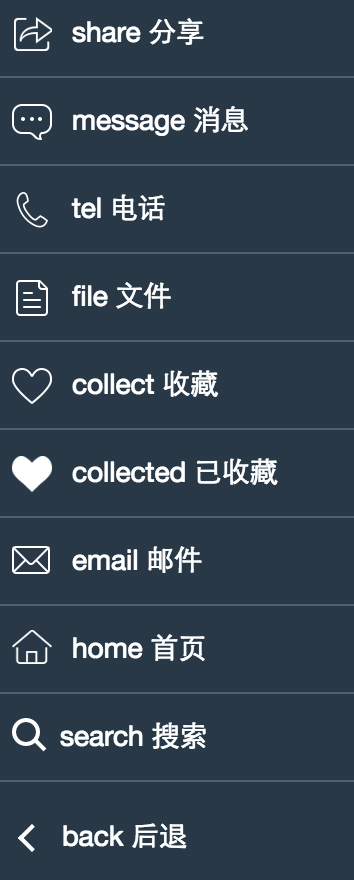
12 });

[复制代码](javascript:void(0);)

**图标**

header组件上会有很多的图标，而根据之前的约定，tagname与图标是一一对应的，这里就要给出一些基本的映射关系了：

因为H5与native是以tagname作为标识，所以一定不能重复



这些皆需要Native同事实现，如果是新出的图标的话，可以读取线上http的图标，比如这样：

[复制代码](javascript:void(0);)

1 Hybrid.ui.header.set({

2 back: function () {

3 requestHybrid({

4 tagname: 'back',

5 param: {

6 topage: 'index',

7 type: 'native'

8 }

9 });

10 },

11 title: '读取线上资源',

12 right: [

13 {

14 tagname: 'search',

15 icon: 'http://images2015.cnblogs.com/blog/294743/201511/294743-20151102143118414-1197511976.png',

16 callback: function () {

17 alert('读取线上资源')

18 }

19 }

20 ]

21 });

[复制代码](javascript:void(0);)

但如果是常用的图标还要去线上取的话，对性能不太好，而这里也引出了一个比较大的话题，静态资源缓存问题，这个我们在后面点描述。

**防止假死**

其实之前我提出过拒绝使用NativeUI的想法，当时最是抵制的就是Header组件，因为如果使用Native的Header的话：

① 我们的loading将header盖不住

② 每次前端header有什么特殊需求都实现不了，必须等待Native支持（比如Header隐藏之类的）

为了抵制我还提出了一些方案，但是以后面实际项目来说，事实上是很难摆脱Header组件：

① 断网情况下白屏问题

② js报错假死问题

正如所说，我们会使用Native的功能一个很大的原因是为了防止js出错而导致app假死，而经过我们之前的设计，连back按钮的回调也是我们定义的，如果js报错的话，这个back的回调可能没注册上去也可能回调报错了，为了处理这个问题，我们这里需要一个约定：

对header的tagname为back的按钮做了特殊化，类似可能做特殊化的tagname是home、tel

① 如果back按钮没有设置回调则执行webview（浏览器）的history.back

② 如果history为1的话，默认执行退回上一页

③ 如果点击back的时候具有回调则执行回调（JavaScript回调，必须返回true）

④ 如果js回调返回true则Native流程结束，如果300ms没有返回或者返回不为true则跳转到大首页（这个根据业务而定，也可能回到上一页）

这样的话，就算js报错，或者回调报错，也可以保证APP不会陷入假死的情况。

请注意，这样只能避免用户进了某一个页面出不去的情况，并不是说页面没BUG！！！

如果这里发生了阻塞主流程的BUG，页面应该要有自动预警与在线更改机制，避免用户&订单流失

这里一旦具有回调但是依旧执行了Native回调的场景就一定是页面有问题，这个时候就应该打点上报日志，日志收集后马上短信轰炸业务开发人员，这个日志也是有一定要求的：我们希望错误日志定位到哪一个页面甚至哪一个方法出了问题，如果有具体操作路径就更好了，后面的比较难，第一条一定要做到。当错误定位到后，我们便需要快速解决问题，上线代码，这里涉及Hybrid在线更新一块的逻辑，我们后面再说。

**数据请求**

事实上对H5来说，请求走Ajax是没有问题的，跨域等问题都有很多解决方案，真正让我们想用Native代理发出请求的是账号信息统一（后面又有Native走tcp的场景），请思考以下场景：

Native往往是可以持久化登录信息的，所以很多主流的Hybrid框架如果是直连（webview直接访问一个url）的话会直接将cookie信息注入给webview，这个时候业务就直接获取了登录态了，但总有业务可能会产生登出操作，然后换个账号登录进来，这个时候webview与Native的账号就不统一了，没有处理方案的话，这个时候用户就会懵逼了，觉得整个APP不可信！

有一种方案是可以绕过这个问题的，就是对登录登出“收口”（我们又提到收口一词了哦），限制业务开发登出必须使用APP系统提供的登录登出，因为一般大公司有统一的passport做鉴权，比如手机百度，就算你在webview中重新登录了，因为使用的是APP提供的登录登出，而其他频道应用与你皆是使用的passport鉴权，所以可以用这个方案，但是这个方案对于多数小公司可能是不可行的。

第一是很多小公司没那个意识去打造类似passport这种东西，这个也不是前端能推动的事情，就算你实现了整个passport机制，还得保证整个公司其它团队使用你的系统，如果有一个团队不买账就懵逼了。

没有统一的账号系统往往有历史包袱的因素，技术债需要及时还清

所以现在很多团队的现状是一个项目都会有自己的登陆注册（这个事实上很傻逼），频道之间登录态共享都没有做到，所以对登陆登出做收口便不适用了，但是因为Native是新业务，不存在历史包袱，APP一般又是战略性产品，前端做不到的事情，如果和APP挂钩，往往可以在某些方面完成类似的事情，就我们现在来说APP就有自己的一套鉴权机制，虽然不清楚他内部实现（后面需详细了解），但是每个业务接口对APP都是很友好的，所以请求直接走Native代理发出会是一个非常好的选择。

[复制代码](javascript:void(0);)

1 requestHybrid({

2 //post

3 tagname: 'get',

4 param: {

5 url: 'http://api/demain.com',

6 param: {a: 1, b: 2}

7 },

8 callback: function (data) {

9 }

10 });

[复制代码](javascript:void(0);)

解决了以上问题，事实上只需要Native端新释放一个接口即可，当然这里又会回到之前一个问题，post的参数问题，这个时候可能就需要配置为JavaScriptCore方式通信，或者将请求参数放在一个全局方法中等待Native调用获取。

业务开发中需要禁止出现登出操作，所有的登出都要走APP唯一页面的唯一登出按钮；如果APP本身未登陆，那么可以要求用户进入页面前先登陆，也可以在访问到具体需要登陆的接口时弹出登陆框让用户登陆了才能进行后续操作。

因为请求由native发出不会有跨域问题，考虑到安全性，这里会有一个域名白名单，只有白名单的请求才能发出去

**NativeUI**

像我们前面说的Header组件与登陆框，事实上都算得上Native组件，只不过header是单纯的UI组件，登陆框算得上业务组件的，H5会用到NativeUI的场景不多，但是loading这个东西因为要降低页面白屏时间会经常用到。

一般来说在webview加载html时会有一段时间白屏时间，这个时候便需要Native的loading出场，在页面加载完成后需要在前端框架层将Native loading关闭。

[复制代码](javascript:void(0);)

1 var HybridUI = {};

2 HybridUI.showLoading();

3 //=>

4 requestHybrid({

5 tagname: 'showLoading'

6 });

7

8 HybridUI.showToast({

9 title: '111',

10 //几秒后自动关闭提示框，-1需要点击才会关闭

11 hidesec: 3,

12 //弹出层关闭时的回调

13 callback: function () { }

14 });

15 //=>

16 requestHybrid({

17 tagname: 'showToast',

18 param: {

19 title: '111',

20 hidesec: 3,

21 callback: function () { }

22 }

23 });

[复制代码](javascript:void(0);)

这一套UI组件皆需要与前端框架中的组件使用做到一致性，这种业务类组件不多说，这里说一个可能会遇到的问题：

**NativeUI通信问题**

不可避免的，我们会遇到NativeUI组件与H5通信的问题，举个简单的例子，我们为了交互效果，新开了一Native的弹出层组件，大概这个样子：



大家这里不要把它当做单独的View，将它看做一个H5的弹出层，只不过这个弹出层是Native实现的，整个调用方式也许是这样的：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'showCitilist',

3 param: {

4 data: [

5 {name: '北京'}, {name: '上海'}

6 //......

7 ]

8 },

9 callback: function(item) {

10 alert(item.name)

11 }

12 });

[复制代码](javascript:void(0);)

这里Native弹出了一个弹出层，装载的是Native的UI，点击某一个城市，执行的是H5的回调，表面逻辑上这个Native的UI应该是基于Webview的，事实上这个NativeUI可能是一个单例，其实这个实现还比较简单，因为他的点击交互比较单一，Native可以很容易的将数据获得再回调H5的方法，这里与Header上的点击事件处理一致，比较复杂的是Native新开了一个弹出层而他是一个Webview，装载我们自己的H5代码，这个便复杂了。

**Webview通信**

请考虑以下业务场景，这次依旧是使用Native弹出层，但是这里的弹出层是一个Webview组件，里面的内容需要我们自定义，调用可能是这样的：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'showpagelayer',

3 param: {

4 html: '<input id="test" type="text" ><input type="button" id="btn" >',

5 events: {

6 'click #btn': function() {

7 var v = $('#test').val();

8 //调用父元素方法

9 //parentCallback(v);

10 //关闭当前弹出层

11 //Hybrid.ui.hidepagelayer()

12 }

13 },

14 }

15 });

[复制代码](javascript:void(0);)

这个代码之所以可以这样写，是因为我们对这个页面展示的Dom结构与事件有控制力，但是如果这个页面如果压根不是我写的，而且上面那种代码的应用场景基本为0，我们真实的使用场景往往是直接载入一个页面，比如这个例子：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'showpagelayer',

3 param: {

4 src: 'http://domain.com/webapp/common/city.html',

5 }

6 });

[复制代码](javascript:void(0);)

如果是以url载入一个页面的话，我们对页面的控制力就没有了，除非有一个规则让我们可以对页面的某些方法进行重写，比如依赖一个框架：

一个好的Hybrid平台除了基础实现外，还需要一配套使用前端框架，框架需要最大限度的保证业务代码一致，提升业务的开发效率

我们这里为了方便大家理解做简单实现即可。首先，我们约定，这类可以用弹出层打开的页面一定是具备某些“公共”特性的页面，比如：

① 城市列表页

② 常用联系人选择页

③ XX类型选择页

切记，这类页面一定是公共业务，不会包含过于业务化的东西，否则是不适用的，那种页面还是以url传参处理吧。

然后，我们对这类页面的处理也仅限于回调的处理，不会影响到他们本身的渲染，比如是这样的页面：

[复制代码](javascript:void(0);)

1 <input type="text" id="test" >

2 <input type="button" value="父页面通信" id="btn">

3 <script src="http://sandbox.runjs.cn/uploads/rs/279/2h5lvbt5/zepto.js" type="text/javascript"></script>

4 <script type="text/javascript">

5 $('#btn').click(function (){

6 var val = $('#test').val();

7 clickAction(val)

8 });

9 //override

10 function clickAction (val) {

11 alert(val)

12 };

13 </script>

[复制代码](javascript:void(0);)

而我们真实的调用是这样的：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'showpageview',

3 param: {

4 src: 'http://sandbox.runjs.cn/show/imbacaz7',

5 callbacks: {

6 //请注意,这里的key值

7 clickAction: function(val) {

8 //parentCallback(val);

9 //关闭当前webview，我们约定这类webview是单例

10 //Hybrid.ui.hidepageview()

11 }

12 }

13 }

14 });

[复制代码](javascript:void(0);)

webview载入结束后，我们会使用我们自己定义的方法将原来页面的方法重写掉，比如使用JavaScriptCore重写掉。当然，真实的使用场景不会这么简单，具体的业务逻辑就看依赖框架（blade）的实现吧。

PS：这里的实现过于复杂，不太实用，各位暂时还是保持url跳转通信吧，这里待研究

**静态资源读取&更新**

前面我们设置header时，用到了在线静态资源，那里直接是使用的http的资源，我们在实际业务中因为知道自己的图标在什么位置所以代码可能是这样的：

[复制代码](javascript:void(0);)

1 {

2 tagname: 'search',

3 //如果当前是机票频道,这个会转化为 http://domain.com/webapp/flight/static/hybrid/icon-search.png

4 icon: './static/hybrid/icon-search.png',

5 callback: function () {

6 alert('读取线上资源')

7 }

8 }

[复制代码](javascript:void(0);)

根据之前的规划，Native中如果存在静态资源，也是按频道划分的：

[复制代码](javascript:void(0);)

webapp //根目录

├─flight

├─hotel //酒店频道

│ │ index.html //业务入口html资源，如果不是单页应用会有多个入口

│ │ main.js //业务所有js资源打包

│ │

│ └─static //静态样式资源

│ ├─css

│ ├─hybrid //存储业务定制化类Native Header图标

│ └─images

├─libs

│ libs.js //框架所有js资源打包

│

└─static //框架静态资源样式文件

├─css

└─images

[复制代码](javascript:void(0);)

**如何读取缓存**

我们开始考虑webview读取Native静态资源时候想了几套方案，比如：

icon: 'hotel/icon.png'

这种形式就是业务开发知道Native的hotel有icon.png的静态资源，便直接Native读取了，但是后来我觉得这种方案不太好，谁知道哪次更新Native中就没有这个包了呢？那个时候岂不是代码就直接报错了，所以最后我们决定我们所有的静态资源一定要过http，因为：

很多业务最初开发的时候都是直接使用浏览器开发或者Native直连url开发，这种时候就能保证所有的静态资源的地址不会错

在正式上线后，我们可能有一部分公共资源内嵌，这个时候便需要一定机制让Native返回本地文件：

Native会拦截所有的Webview请求，如果发现某个资源请求本地也存在便直接返回

所以这里的症结点是Native如何过滤请求，首先，Native只拦截某些域名的请求，因为我们本地资源都一定会有一个规则，拿到请求后，我们会匹配这个规则，比如说，我们会将这个类型的请求映射到本地：

http://domain.com/webapp/flight/static/hybrid/icon-search.png

//===>>

file ===> flight/static/hybrid/icon-search.png

Native会直接去flight目录寻找是否有这个文件，如果有就直接返回了，但是我们这里会有一个忧虑点：

这种拦截所有请求的方法再检查文件是否存在是否会很耗时

因为我并不能肯定，于是让Native同事做了一个实验，检查100个文件本地是否存在，耗时都在10ms以内。

关于读取Native缓存，我们也可以使用前端构建工具直接以频道为单位生成一个清单，然后Native就只对清单内的请求做处理，但是这里会多一步操作，出错的几率可能增大，考虑的全部拦截的耗损不是很大，于是我们采用了全部拦截的方案，这里简单说下Native的实现方案，具体各位在代码中去看吧：

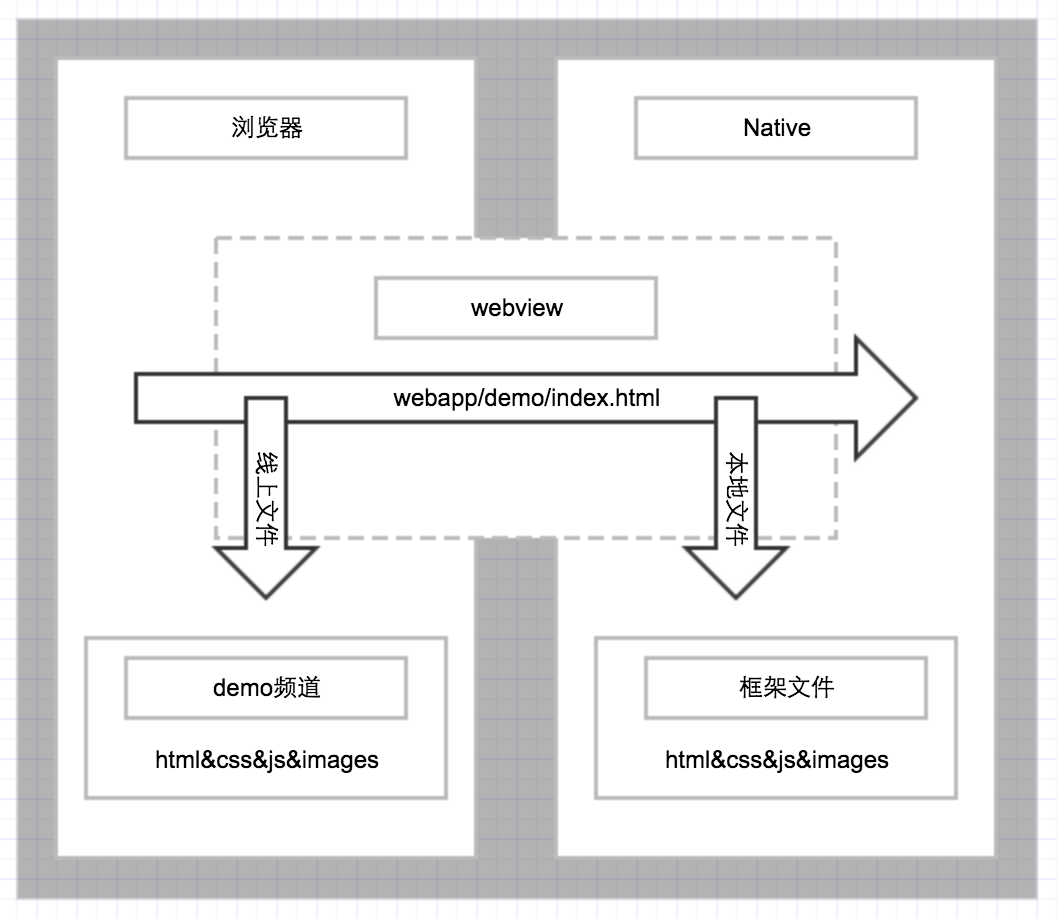
**实现方案**

这里IOS与Android实现大同小异，这里直接放出代码各位自己去研究吧：

PS：这里是测试时候的代码，最后实现请看git里面的

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif IOS

其实这里优点已经非常明显，业务写代码的时候压根不需要考虑文件是否需要本地读取，Native也可以有一个开关轻易的配置哪些频道需要读取本地文件：



以我们的demo为例，关于业务频道demo的所有静态资源全部走线上，有效减少APP包大小，公共文件或者框架文件在APP中全部走本地，因为核心框架一般比较大，这里可以提升70%以上的载入速度。

**增量更新**

有缓存策略就会有更新策略，事实上这里的更新策略涉及到在线发版等功能，这个工作是非常重的，如果说之前的工作前端与Native就能完成的话，那么这个工作会有server端的同事参加，还有可能形成一个功能庞大的发布平台。

最简单的模拟，就是每次Native大版本发布都会有一个版本映射表：

[复制代码](javascript:void(0);)

{//业务频道版本号

flight: 1.0.0,

hotel: 1.0.0,

libs: 1.0.0,

static: 1.0.0

}

[复制代码](javascript:void(0);)

其中每个，如果某一天我们发现了机票频道一个BUG，发布了一个增量包，那么机票的版本就会增加：

//bug修复

flight: 1.0.1

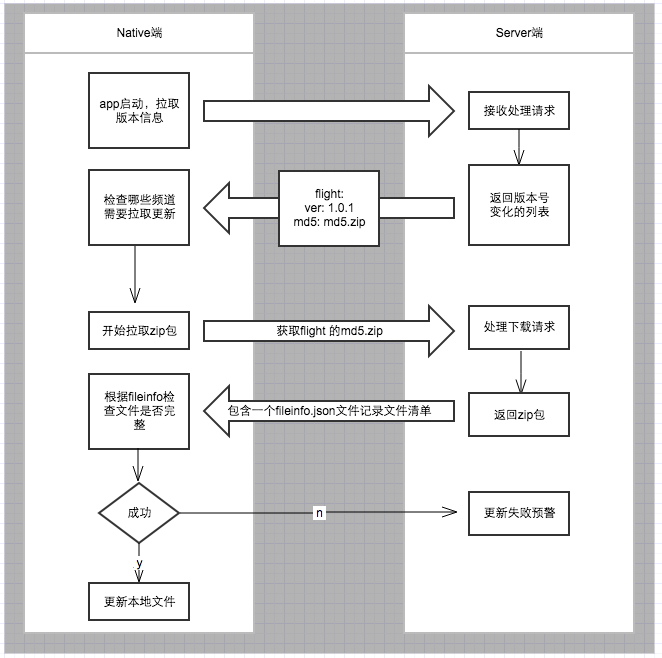
//功能发布

flight: 1.1.0

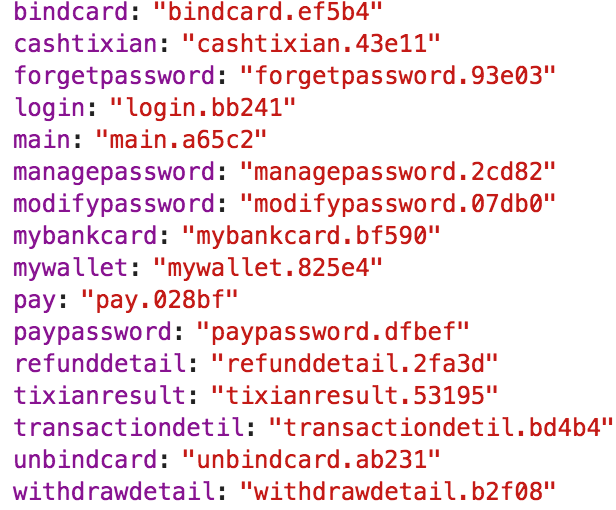
对于这个版本，后台数据可可能会有这么一个映射：

|  |  |  |
| --- | --- | --- |
| channel | ver | md5 |
| flight | 1.0.0 | 1245355335 |
| hotel | 1.0.1 | 455ettdggd |

每一个md5值对应着一个实际存在的增量包，在CDN服务器上，每次APP启动，就会检查server端的版本号是不是一致，如果不一致就需要重新拉取zip包，然后更新本地版本号：



这个是比较简单的场景，以一个频道为单位的更新，没有做到粒度更细，安全性方面一般情况我们也不必关心有人会篡改你的zip包（比如开发商），在你app流量不大的情况，没人有那么蛋疼，但是我们要考虑开发人员发布的zip包在某个环节出了问题的情况，一般来说，我们的打包程序会根据每个文件形成一个md5清单，比如这个样子的：

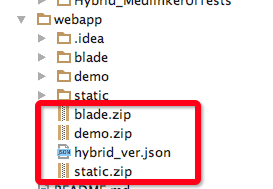


Native拿到后会去检查这个清单所有的文件是否完整，如果不完整就有问题，需要打日志预警放弃这次更新。

**简单实现**

我们这里由于暂时没有Server端的参与，不能做发布系统，所以暂时是将版本信息放到了项目根目录做简单实现，这里还包含三个频道的zip包：

PS：真实场景更复杂更严谨



{

"blade": "1.0.0",

"static": "1.0.0",

"demo": "1.0.0"

}

我们现在把更新做到了这个页面：



这里的流程是：

1 点击检查更新首先检查Native里面有没有hybrid\_ver.json这个文件，没有就去http://yexiaochai.github.io/Hybrid/webapp/hybrid\_ver.json*下载，完了拿到json串把对应文件全部下载下来解压：*

{

"blade": "1.0.0",

"static": "1.0.0",

"demo": "1.0.0"

}

对应规则是：

http://yexiaochai.github.io/Hybrid/webapp/blade.zip

http://yexiaochai.github.io/Hybrid/webapp/static.zip

http://yexiaochai.github.io/Hybrid/webapp/demo.zip

PS：这里真实情况下其实对应的是md5的压缩包，我们这里不去纠结。

如果第二次你点击，这个时候本地有hybrid\_ver.json文件了，你再去远端获取这个文件，对比三个频道的版本号，这个时候是一样的，所以不会拉取。

如果我们改动了demo文件中的某个文件，比如加了一个alert什么的，这个时候就重新形成一个zip包，*然后你把demo的版本号加大，比如这样：*

{

"blade": "1.0.0",

"static": "1.0.0",

"demo": "1.0.1"

}

*他就该拉取demo的增量包，再次进入系统的时候便能看到更新了。*

**结语**

与上次文章对比，我们这次在一些Hybrid设计的细节点把握的更多，希望此文能对准备接触Hybrid技术的朋友提供一些帮助，关于Hybrid的系列还会有最后一篇实战类文章介绍，有兴趣的朋友持续关注吧，这里是一些效果图：



[**浅谈Hybrid技术的设计与实现第三弹——落地篇**](http://www.cnblogs.com/yexiaochai/p/5813248.html)

**前言**

接上文：（阅读本文前，建议阅读前两篇文章先）

[浅谈Hybrid技术的设计与实现](http://www.cnblogs.com/yexiaochai/p/4921635.html)

[浅谈Hybrid技术的设计与实现第二弹](http://www.cnblogs.com/yexiaochai/p/5524783.html)

根据之前的介绍，大家对前端与Native的交互应该有一些简单的认识了，很多朋友就会觉得这个交互很简单嘛，其实并不难嘛，事实上单从Native与前端的交互来说就那点东西，真心没有太多可说的，但要真正做一个完整的Hybrid项目却不容易，要考虑的东西就比较多了，单从这个交互协议就有：

① URL Schema

② JavaScriptCore

两种，到底选择哪种方式，每种方式有什么优势，都是我们需要深度挖掘的，而除此之外，一个Hybrid项目还应该具有以下特性：

① 扩展性好——依靠好的约定

② 开发效率高——依赖公共业务

③ 交互体验好——需要解决各种兼容问题

我们在实际工作中如何落地一个Hybrid项目，如何推动一个项目的进行，这是本次我们要讨论的，也希望对各位有用。

文中是我个人的一些开发经验，希望对各位有用，也希望各位**多多支持讨论**，指出文中**不足**以及提出您的一些**建议**。

**设计类博客**

<http://www.cnblogs.com/yexiaochai/p/4921635.html>  
<http://www.cnblogs.com/yexiaochai/p/5524783.html>

**iOS博客**

<http://www.cnblogs.com/nildog/p/5536081.html#3440931>

**Android博客**

<https://home.cnblogs.com/u/vanezkw>

**代码地址：**[**https://github.com/yexiaochai/Hybrid**](https://github.com/yexiaochai/Hybrid)

因为IOS不能扫码下载了，大家自己下载下来用模拟器看吧，下面开始今天的内容。

**总体概述在第一章，有兴趣大家去看**

**细节设计在第二章，有兴趣大家去看**

**本章主要为打补丁**

**边界问题**

在我们使用Hybrid技术前要注意一个边界问题，什么项目适合Hybrid什么项目不适合，这个要搞清楚，适合Hybrid的项目为：

**① 有60%以上的业务为H5**

**② 对更新（开发效率）有一定要求的APP**

不适合使用Hybrid技术的项目有以下特点：

① 只有20%不到的业务使用H5做

② 交互效果要求较高（动画多）

任何技术都有适用的场景，千万不要妄想推翻已有APP的业务用H5去替代，最后会证明那是自讨苦吃，当然如果仅仅想在APP里面嵌入新的实验性业务，这个是没问题的。

**交互约定**

根据之前的学习，我们知道与Native交互有两种交互：

① URL Schema

② JavaScriptCore

而两种方式在使用上各有利弊，首先来说URL Schema是比较稳定而成熟的，如果使用上文中提到的“ajax”交互方式，会比较灵活；而从设计的角度来说JavaScriptCore似乎更加合理，但是我们在实际使用中却发现，注入的时机得不到保障。

iOS同事在实体JavaScriptCore注入时，我们的原意是在webview载入前就注入所有的Native能力，而实际情况是页面js已经执行完了才被注入，这里会导致Hybrid交互失效，如果你看到某个Hybrid平台，突然header显示不正确了，就可能是这个问题导致，所以JavaScriptCore就被我们弃用了。

JavaScriptCore可能导致的问题：

① 注入时机不唯一（也许是BUG）

② 刷新页面的时候，JavaScriptCore的注入在不同机型表现不一致，有些就根本不注入了，所以全部hybrid交互失效

如果非要使用JavaScriptCore，为了解决这一问题，我们做了一个兼容，用URL Schema的方式，在页面逻辑载入之初执行一个命令，将native的一些方式重新载入，比如：

1 \_.requestHybrid({

2 tagname: 'injection'

3 });

这个能解决一些问题，但是有些初始化就马上要用到的方法可能就无力了，比如：

① 想要获取native给予的地理信息

② 想要获取native给予的用户信息（直接以变量的方式获取）

作为生产来讲，我们还是求稳，所以最终选择了URL Schema。

明白了基本的边界问题，选取了底层的交互方式，就可以开始进行初步的Hybrid设计了，但是这离一个可用于生产，可离落地的Hybrid方案还比较远。

**账号体系**

一般来说，一个公司的账号体系健壮灵活程度会很大程度反映出这个研发团队的整体实力：

① 统一的鉴权认证

② 短信服务图形验证码的处理

③ 子系统的权限设计、公共的用户信息导出

④ 第三方接入方案

⑤ 接入文档输出

⑥ ......

这个技术方案，有没有是一回事（说明没思维），有几套是一回事（说明比较乱，技术不统一），对外的一套做到了什么程度又是一回事，当然这个不是我们讨论的重点，而账号体系也是Hybrid设计中不可或缺的一环。

账号体系涉及了接口权限控制、资源访问控制，现在有一种方案是，前端代码不做接口鉴权，账号一块的工作全部放到native端。

**native代理请求**

在H5想要做某一块老的App业务，这个APP80%以上的业务都是Native做的，这类APP在接口方面就没有考虑过H5的感受，会要求很多信息如：

① 设备号

② 地理信息

③ 网络情况

④ 系统版本

有很多H5拿不到或者不容易拿到的公共信息，因为H5做的往往是一些比较小的业务，像什么个人主页之类的不重要的业务，Server端可能不愿意提供额外的接口适配，而使用额外的接口还有可能打破他们统一的某些规则；加之native对接口有自己的一套公共处理逻辑，所以便出了Native代理H5发请求的方案，公共参数会由Native自动带上。

[复制代码](javascript:void(0);)

1 //暂时只关注hybrid调试,后续得关注三端匹配

2 \_.requestHybrid({

3 tagname: 'apppost',

4 param: {

5 url: this.url,

6 param: params

7 },

8

9 callback: function (data) {

10 scope.baseDataValidate(data, onComplete, onError);

11 }

12 });

[复制代码](javascript:void(0);)

这种方案有一些好处，接口统一，前端也不需要关注接口权限验证，但是这个会带给前端噩梦！

前端相对于native一个很大的优点，就是调试灵活，这种代理请求的方式，会限制请求只能在APP容器中生效，对前端调试造成了很大的痛苦

从真实的生产效果来说，也是很影响效率的，容易导致后续前端再也不愿意做那个APP的业务了，所以使用要慎重......

**注入cookie**

前端比较通用的权限标志还是用cookie做的，所以Hybrid比较成熟的方案仍旧是注入cookie，这里的一个前提就是native&H5有一套统一的账号体系（统一的权限校验系统）。

因为H5使用的webview可以有独立的登录态，如果不加限制太过混乱难以维护，比如：

我们在qq浏览器中打开携程的网站，携程站内第三方登录可以唤起qq，然后登录成功；完了qq浏览器本来也有一个登录态，发现却没有登录，点击一键登录的时候再次唤起了qq登录。

当然，qq作为一个浏览器容器，不应该关注业务的登录，他这样做是没问题的，但是我们自己的一个H5子应用如果登录了的话，便希望将这个登录态同步到native，这里如果native去监控cookie的变化就太复杂了，通用的方案是：

Hybrid APP中，所有的登录走Native提供的登录框

每次打开webview native便将当前登录信息写入cookie中，由此前端就具有登录态了，登录框的唤起在接口处统一处理：

[复制代码](javascript:void(0);)

1 /\*

2 无论成功与否皆会关闭登录框

3 参数包括：

4 success 登录成功的回调

5 error 登录失败的回调

6 url 如果没有设置success，或者success执行后没有返回true，则默认跳往此url

7 \*/

8 HybridUI.Login = function (opts) {

9 };

10 //=>

11 requestHybrid({

12 tagname: 'login',

13 param: {

14 success: function () { },

15 error: function () { },

16 url: '...'

17 }

18 });

19 //与登录接口一致，参数一致

20 HybridUI.logout = function () {

21 };

[复制代码](javascript:void(0);)

**账号切换&注销**

账户注销本没有什么注意点，但是因为H5 push了一个个webview页面，这个重新登录后这些页面怎么处理是个问题。

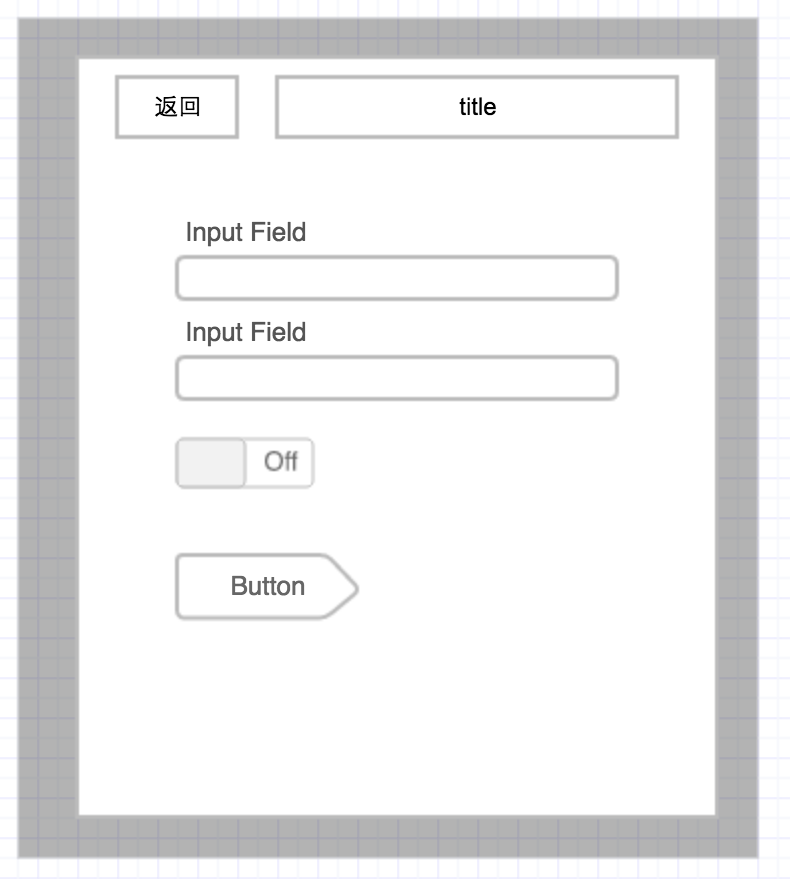
我们这边设计的是一旦重新登录或者注销账户，所有的webview都会被pop掉，然后再新开一个页面，就不会存在一些页面展示怪异的问题了。

**公共业务的设计-体系化**

在Hybrid架构中（其实就算在传统的业务中也是），会存在很多公共业务，这部分公共业务很多是H5做的（比如注册、地址维护、反馈等，登录是native化了的公共业务），我们一个Hybrid架构要真正的效率高，就得把各种公共业务做好了，不然单是H5做业务，效率未必会真的比Native高多少。

底层框架完善并且统一后，便可以以规范的力量限制各业务开发，在统一的框架下开发出来的公共业务会大大的提升整体工作效率，这里以注册为例，一个公共页面一般来说得设计成这个样子：

公共业务代码，应该可以让人在URL参数上对页面进行一定定制化，这里URL参数一般要独特一些，一面被覆盖，这个设计适用于native页面



URL中会包含以下参数：

① \_hashead 是否有head，默认true

② \_hasback 是否包含回退按钮，默认true

③ \_backtxt 回退按钮的文案，默认没有，这个时候显示为回退图标

④ \_title 标题

⑤ \_btntxt 按钮的文案

⑥ \_backurl 回退按钮点击时候的跳转，默认为空则执行history.back

⑦ \_successurl 点击按钮回调成功时候的跳转，必须

只要公共页面设计为这个样子，就能满足多数业务了，在底层做一些适配，可以很轻易的一套代码同时用于native与H5，这里再举个例子：

如果我们要点击成功后去到一个native页面，如果按照我们之前的设计，我们每个Native页面皆已经URL化了的话，我们完全可以以这种方向跳转：

[复制代码](javascript:void(0);)

1 requestHybrid({

2 tagname: 'forward',

3 param: {

4 topage: 'nativeUrl',

5 type: 'native'

6 }

7 });

[复制代码](javascript:void(0);)

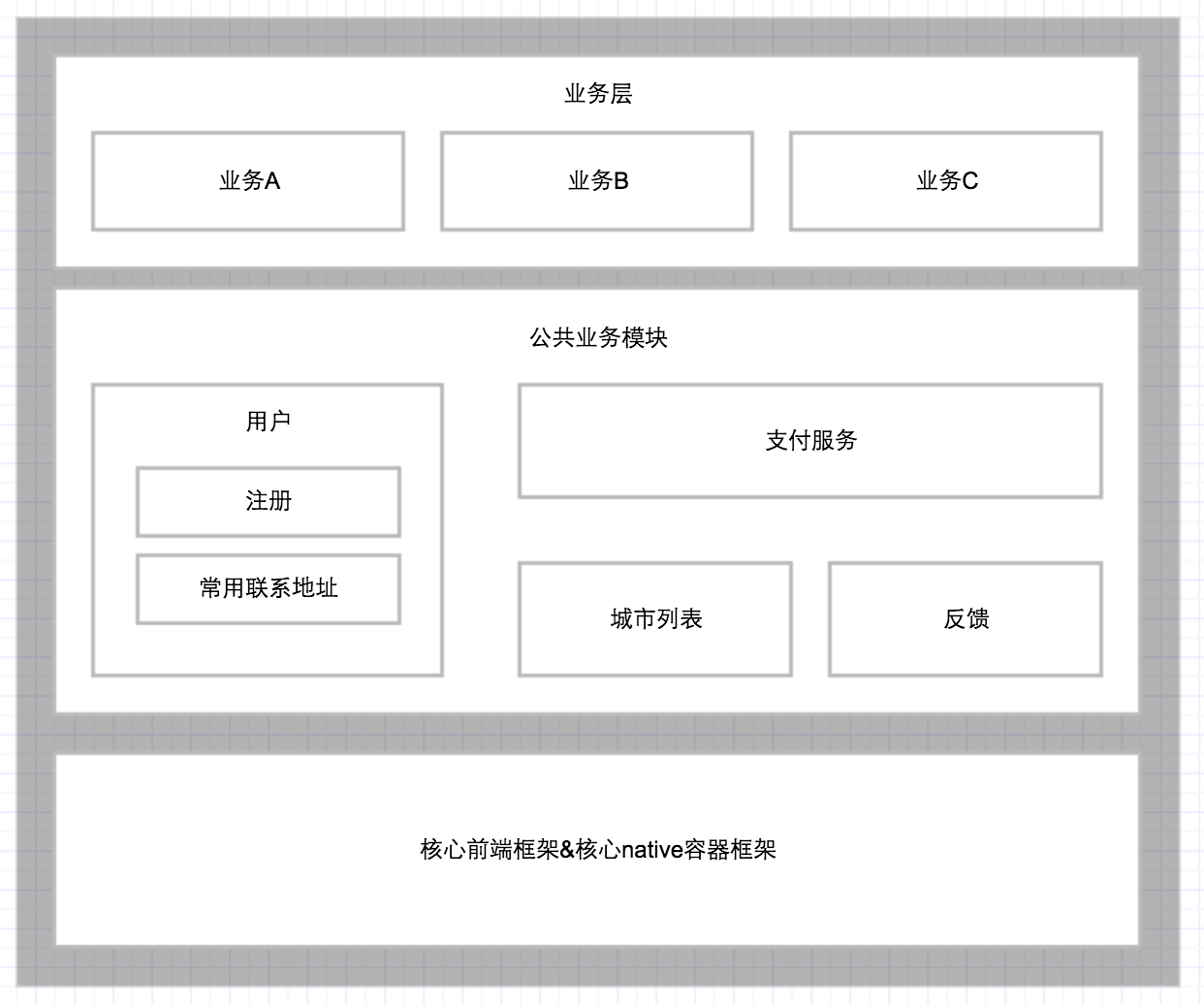
这个命令会生成一个这样的url的链接：

\_successurl == hybrid://forward?param=%7B%22topage%22%3A%22nativeUrl%22%2C%22type%22%3A%22native%22%7D

完了，在点击回调时要执行一个H5的URL跳转：

window.location = \_successurl

而根据我们之前的hybrid规范约定，这种请求会被native拦截，于是就跳到了我们想要的native页面，整个这一套东西就是我们所谓的体系化：



**离线更新**

根据之前的约定，Native中如果存在静态资源，也是按频道划分的：

[复制代码](javascript:void(0);)

webapp //根目录

├─flight

├─hotel //酒店频道

│ │ index.html //业务入口html资源，如果不是单页应用会有多个入口

│ │ main.js //业务所有js资源打包

│ │

│ └─static //静态样式资源

│ ├─css

│ ├─hybrid //存储业务定制化类Native Header图标

│ └─images

├─libs

│ libs.js //框架所有js资源打包

│

└─static //框架静态资源样式文件

├─css

└─images

[复制代码](javascript:void(0);)

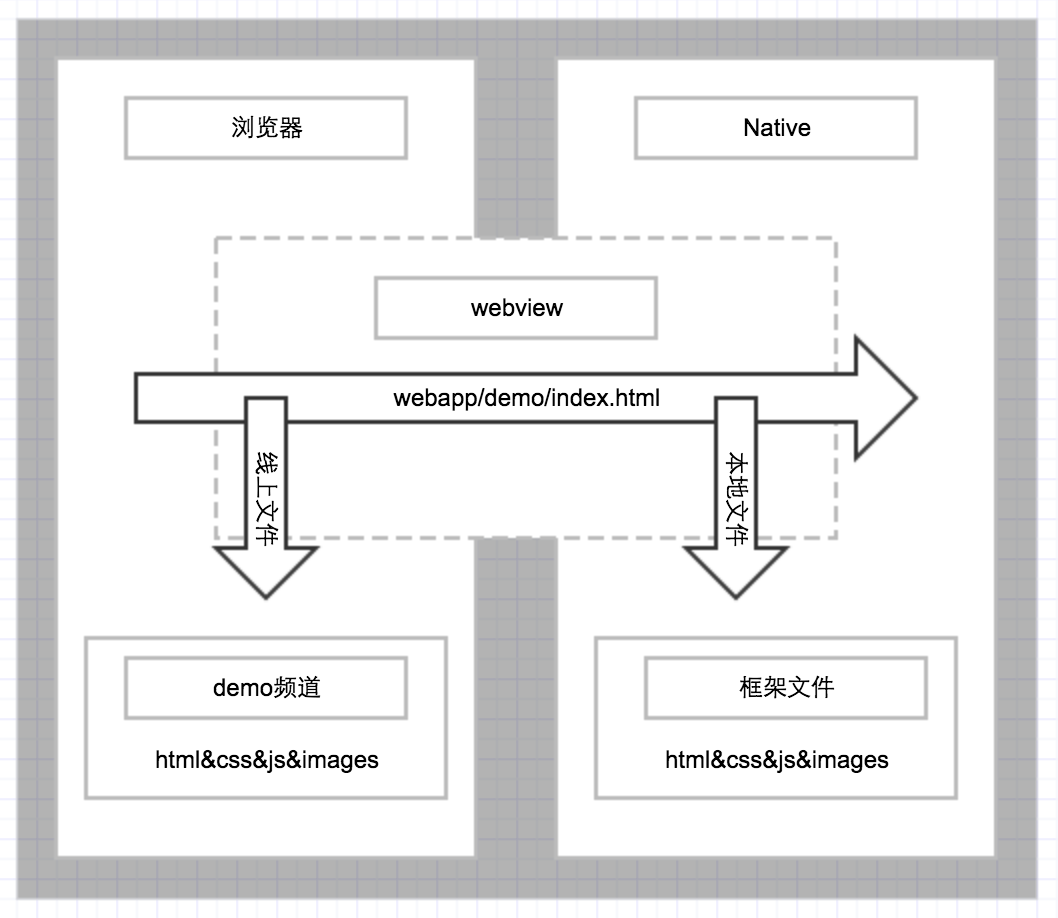
我们这里制定一个规则，native会过滤某一个规则的请求，检查本地是否有该文件，如果本地有那么就直接读取本地，比如说，我们会将这个类型的请求映射到本地：

http://domain.com/webapp/flight/static/hybrid/icon-search.png

//===>>

file ===> flight/static/hybrid/icon-search.png

这样在浏览器中便继续读取线上文件，在native中，如果有本地资源，便读取本地资源：



但是我们在真实使用场景中却遇到了一些麻烦。

**增量的粒度**

其实，我们最开始做增量设计的时候就考虑了很多问题，但是真实业务的时候往往因为时间的压迫，做出来的东西就会很简陋，这个只能慢慢迭代，而我们所有的缓存都会考虑两个问题：

① 如何存储&读取缓存

② 如何更新缓存

浏览器的缓存读取更新是比较单纯的：

浏览器只需要自己能读到最新的缓存即可

而APP的话，会存在最新发布的APP希望读到离线包，而老APP不希望读到增量包的情况（老的APP下载下来增量包压根不支持），更加复杂的情况是想对某个版本做定向修复，那么就需要定向发增量包了，这让情况变得复杂，而复杂即错误，我们往往可以以简单的约定，解决复杂的场景。

思考以下场景：

我们的APP要发一个新的版本了，我们把最初一版的静态资源给打了进去，完了审核中的时候，我们老版本APP突然有一个临时需求要上线，我知道这听起来很有一些扯淡，但这种扯淡的事情却真实的发生了，这个时候我们如果打了增量包的话，那么最新的APP在审核期间也会拉到这次代码，但也许这不是我们所期望的，于是有了以下与native的约定：

Native请求增量更新的时候带上版本号，并且强迫约定iOS与Android的大版本号一致，比如iOS为2.1.0Android这个版本修复BUG可以是2.1.1但不能是2.2.0

然后在服务器端配置一个较为复杂的版本映射表：

[复制代码](javascript:void(0);)

## 附录一

// 每个app所需的项目配置

const APP\_CONFIG = [

'surgery' => [ // 包名

'channel' => 'd2d', // 主项目频道名

'dependencies' => ['blade', 'static', 'user'], // 依赖的频道

'version' => [ // 各个版本对应的增量包范围，取范围内版本号最大的增量包

'2.0.x' => ['gte' => '1.0.0', 'lt' => '1.1.0'],

'2.2.x' => ['gte' => '1.1.0', 'lt' => '1.2.0']

],

'version\_i' => [ // ios需特殊配置的某版本

],

'version\_a' => [ // Android需特殊配置的某版本

]

]

];

[复制代码](javascript:void(0);)

这里解决了APP版本的读取限制，完了我们便需要关心增量的到达率与更新率，我们也会担心我们的APP读到错误的文件。

**更新率**

我们有时候想要的是一旦增量包发布，用户拿着手机就马上能看到最新的内容了，而这样需要app调用增量包的频率增高，所以我们是设置每30分钟检查一次更新。

**正确读取**

这里可能有点杞人忧天，因为Native程序不是自己手把手开发的，总是担心APP在正在拉取增量包时，或者正在解压时，读取了静态文件，这样会不会读取错误呢，后面想了想，便继续采用了之前的md5打包的方式，将落地的html中需要的文件打包为md5引用，如果落地页下载下来后，读不到本地文件就自己会去拉取线上资源咯。

**调试**

一个Hybrid项目，要最大限度的符合前端的开发习惯，并且要提供可调试方案

我们之前说过直接将所有请求用native发出有一个最大的问题就是调试不方便，而正确的hybrid的开发应该是有70%以上的时间，纯业务开发者不需要关心native联调，当所有业务开发结束后再内嵌简单调一下即可。

因为调试时候需要读取测试环境资源，需要server端qa接口有个全局开关，关闭所有的增量读取

关于代理调试的方法已经很多人介绍过了，我这里不再多说，说一些native中的调试方案吧，其实很多人都知道。

**iOS**

首先，你需要拥有一台Mac机，然后打开safari；在偏好设置中将开发模式打开：



然后打开模拟器，即可开始调试咯：



**Android**

Android需要能FQ的chrome，然后输入chrome://inspect/#devices即可，前提是native同事为你打开调试模式，当然Android也可以使用模拟器啦，但是Android的真机表现过于不一样，还是建议使用真机测试。

**一些坑点**

**不要命就用swift**

苹果官方出了swift，于是我们iOS团队好事者尝试了感觉不错，便迅速在团队内部推广了起来，而我们OC本身的体量本来就有10多万行代码量，我们都知道一个道理：

重构一时爽，项目火葬场

而重构过程中肯定又会遇到一些历史问题，或者一些第三方库，代码总会有一点尿不尽一点冗余，而不知道swift是官方有问题还是怎么回事，每次稍微多一些改动就需要**编译一个多小时**！！！！你没看错，是要编译一个多小时。

一次，我的小伙伴在打游戏，被我揪着说了两句，他说他在编译，我尼玛很不屑的骂了他，后面开始调iOS时，编译了2小时！！！从那以后看见他打游戏我一点脾气都没有了！！！

这种编译的感觉，就像吃坏了肚子，在厕所蹲了半天却什么也没拉出来一样！！！所以，不要命就全部换成swift吧。

如果有一定历史包袱的业务，或者新业务，最好不要全面使用新技术，不成熟的技术，如果有什么不可逆的坑，那么会连一点退路都没有了。

**iOS静态资源缓存**

Android有一个全局开关，控制静态资源部读取缓存，但是iOS中研究了好久，都没有找到这个开关，而他读取缓存又特别厉害，所以所有的请求资源在有增量包的情况下，最好加上时间戳或者md5

**Android webview兼容**

Android webview的表现不佳，闪屏等问题比较多，遇到的几个问题有：

① 使用hybrid命令（比如跳转），如果点击快了的话，Android因为响应慢要开两个新页面，需要对连续点击做冻结

② 4.4以下低版本不能捕获js回调，意思是Android拿不到js的返回值，一些特殊的功能就做不了，比如back容错

③ ......

**一些小特性**

为了让H5的表现更加像native我们会约定一些小的特性，这种特性不适合通用架构，但是有了会更有亮点。

**回退更新**

我们在hybrid中的跳转，事实上每次都是新开一个webview，当A->B的时候，事实上A只是被隐藏了，当B点击返回的时候，便直接将A展示了出来，而A不会做任何更新，对前端来说是无感知的。

事实上，这个是一种优化，为了解决这种问题我们做了一个下拉刷新的特性：

[复制代码](javascript:void(0);)

1 \_.requestHybrid({

2 tagname: 'headerrefresh',

3 param: {

4 //下拉时候展示的文案

5 title: '123'

6 },

7 //下拉后执行的回调,强暴点就全部刷新

8 callback: function(data) {

9 location.reload();

10 }

11 });

[复制代码](javascript:void(0);)

但，这个总没有自动刷新来的舒服，于是我们在页面第一次加载的时候约定了这些事件：

[复制代码](javascript:void(0);)

1 // 注册页面加载事件

2 \_.requestHybrid({

3 tagname: 'onwebviewshow',

4 callback: function () {

5

6 }

7 });

8 // 注册页面影藏事件

9 \_.requestHybrid({

10 tagname: 'onwebviewhide',

11 callback: function () {

12 scope.loopFlag = false;

13 clearTimeout(scope.t);

14 }

15 });

[复制代码](javascript:void(0);)

在webview展示的时候触发，和在webview隐藏的时候触发，这样用户便可以做自动数据刷新了，但是局部刷新要做到什么程度就要看开发的时间安排了，技术好时间多自然体验好。

**header-搜索**

根据我们之前的约定，header是比较中规中矩的，但是由于产品和视觉强迫，我们实现了一个不一样的header，最开始虽然不太乐意，做完了后感觉还行......



这块工作量主要是native的，我们只需要约定即可：

[复制代码](javascript:void(0);)

1 this.header.set({

2 view: this,

3 //左边按钮

4 left: [],

5 //右边按钮

6 right: [{

7 tagname: 'cancel',

8 value: '取消',

9 callback: function () {

10 this.back();

11 }

12 }],

13 //searchbox定制

14 title: {

15 //特殊tagname

16 tagname: 'searchbox',

17 //标题,该数据为默认文本框文字

18 title: '取消',

19 //没有文字时候的占位提示

20 placeholder: '搜索医院、科室、医生和病症',

21 //是否默认进入页面获取焦点

22 focus: true,

23

24 //文本框相关具有的回调事件

25 //data为一个json串

26 //editingdidbegin 为点击或者文本框获取焦点时候触发的事件

27 //editingdidend 为文本框失去焦点触发的事件

28 //editingchanged 为文本框数据改变时候触发的事件

29 type: '',

30 data: '' //真实数据

31 },

32 callback: function(data) {

33 var \_data = JSON.parse(data);

34 if (\_data.type == 'editingdidend' && this.keyword != $.trim(\_data.data)) {

35 this.keyword = $.trim(\_data.data);

36 this.reloadList();

37 }

38

39 }

40 });

[复制代码](javascript:void(0);)

**结语**

希望此文能对准备接触Hybrid技术的朋友提供一些帮助，关于Hybrid的系列这里是最后一篇实战类文章介绍，这里是demo期间的一些效果图，后续git库的代码会再做整理：







**落地项目**

真实落地的业务为医联通，有兴趣的朋友试试：





**推动感悟**

从项目调研到项目落地再到最近一些的优化，已经花了三个月时间了，要做好一件事是不容易的，而且我们这个还涉及到持续优化，和配套业务比如：

① passport

② 钱包业务

③ 反馈业务

.....

等同步制作，很多工作的意义，或者作用，是非技术同事看不到的，但是如果我们不坚持做下去，迫于业务压力或者自我松懈放纵，那么就什么也没有了，我们要推动一件事情，不可能一站出来就说，嘿，小样，我们这个不错，你拿去用吧，这样人家会猜疑你的，我们一定是要先做一定demo让人有一定初步印象，再强制或者偷偷再某一个生产业务试用，一方面将技术依赖弄进去，一方面要告诉其他同事，看看嘛，也没有引起多大问题嘛，呵呵。

做事难，推动难，难在坚持，难在携手共进，这里面是需要信念的，在此尤其感谢团队3个伙伴的无私付出（杨杨、文文、文文）。

后续，我们在持续推动hybrid建设的同时，会尝试React Native，找寻更好的更适合自己的解决方案。