**1.mac卸载node**

sudo npm uninstall npm -g

 sudo rm -rf /usr/local/lib/node /usr/local/lib/node\_modules /var/db/receipts/org.nodejs.\* sudo rm -rf /usr/local/include/node /Users/$USER/.npm

 sudo rm /usr/local/bin/node

 sudo rm /usr/local/share/man/man1/node.1

 sudo rm /usr/local/lib/dtrace/node.d

**2.**[**超详细的Node中require第三方包规则**](https://juejin.im/post/5cd6b1fde51d453ab5096ab8)

var printMsg= require('dtq\_test'); 当require一个包名（不是路径名）时，node会去node\_modules下找包名对应的文件夹名下的package,json文件对应的入口文件；

如果是路径名var printMsg= require('./dtq\_test');则找相应的文件。

**3.CDN**

参考：[CDN加速原理](https://www.jianshu.com/p/1dae6e1680ff)

**4.iview Input组件限制输入两位数字**

<Input v-model="refundAmountQuery" @on-keyup="formateRefundAmountQuery" @on-blur="getOrderList">

formateRefundAmountQuery(e) {

this.refundAmountQuery = this.refundAmountQuery.match(/^\d{0,4}/g)[0] || null;

},

**5.iview datepicker js改变v-model值**

<DatePicker

v-model="date"

type="datetimerange"

placeholder="不限"

@on-change="formateDatetime"

@on-ok="selectChange"

style="width: 280px"

></DatePicker>

不要this.date[0] = new Date(), 改变date数组里的值不会导致输入框的值变化; 而要this.date = [new Date(), this.date[1]];

**6.hybrid**

摘要 [#](https://doc.bytedance.net/docs/326/495/4571/#%E6%91%98%E8%A6%81)

移动一体化 是充分融合客户端和前端优秀移动端能力的一种开发模式，达到 兼顾用户体验、降低开发成本、提升迭代速度，也是敏捷开发的一种实践。业内各个公司都在不停尝试，从先驱者PhoneGap到Apache的Cordova，再到手淘、京东等。移动一体化 中适用性最广的就是Hybrid，究竟什么是Hybrid以及Hybrid的关键技术点，本文将为大家一一解读。同时，本文直面现实，针对于IES的Hybrid业务存在的问题进行剖析，为后续IES 多产品、复杂业务、大规模 的Hybrid应用场景，设计并实现 Falcon解决方案。

前言 [#](https://doc.bytedance.net/docs/326/495/4571/#%E5%89%8D%E8%A8%80)

**Hybrid简介**[#](https://doc.bytedance.net/docs/326/495/4571/#hybrid%E7%AE%80%E4%BB%8B)

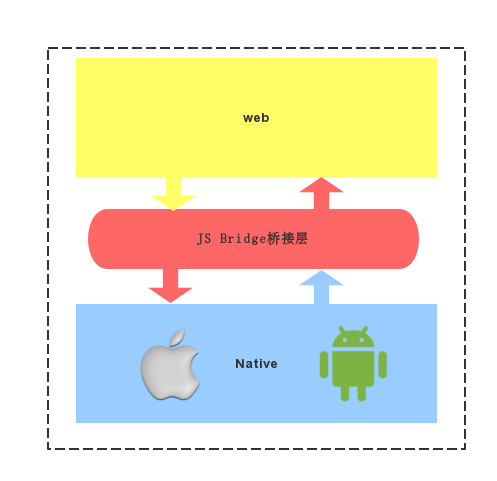
Hybrid App（混合模式移动应用）是指介于web-app、native-app这两者之间的app，兼具“Native App良好用户交互体验的优势”和“Web App跨平台开发的优势”。

概括起来，Hybrid是一种移动端半Native半web的开发模式。底层功能API均由 [原生容器] 通过某种方式提供，然后业务逻辑由H5页面完成，最后原生容器加载H5，完成整个业务流程。

原生容器 即为webview，在Native App开发中有一个webview组件用于加载HTML文件，Android中是webview，iOS7以下有UIWebview，7以上有了WKWebview。

**Hybrid基本架构**[#](https://doc.bytedance.net/docs/326/495/4571/#hybrid%E5%9F%BA%E6%9C%AC%E6%9E%B6%E6%9E%84)

如简介所述，Hybrid顶层是web，底层是Native，中间的桥接层或者胶水层，即是Webview容器。web和Native的通信机制JSBridge主要也是依托于Webview容器，最核心的是容器中是JavaScript引擎。



关于JavaScript起源，JavaScript引擎和JavaScript桥接技术，不了解的可参考《从前端到全端：JavaScript逆袭之路》，链接见参考文档。

**Hybrid核心技术 - JSBridge**[#](https://doc.bytedance.net/docs/326/495/4571/#hybrid%E6%A0%B8%E5%BF%83%E6%8A%80%E6%9C%AF-jsbridge)

客户端内的H5页面是运行在客户端内的Webview中而非系统浏览器，而端内的Webview本质上是客户端的一个组件（iOS中称之为view，Android称之为activity），组件必然归属于客户端的一个线程（iOS可能是主线程，Android是子线程）。客户端的主线程是可以和Webview组件进行数据交互的。

所以，Bridge的实现一般来说有三种解决方案：

**信号式（Signal）**[#](https://doc.bytedance.net/docs/326/495/4571/#%E4%BF%A1%E5%8F%B7%E5%BC%8F%EF%BC%88signal%EF%BC%89)

信号式，顾名思义是通过某种信号来做信息交换。端内的H5页面所有的请求，包括网页请求、资源请求、ajax等，都是需要经过Webview的，而Webview又与客户端主线程可以通信，那么可以在Webview上加入自定义的协议schema，当Webview捕获到约定的协议时通知客户端主线程执行某个操作。即可以完成H5 → 客户端的调用。比如微信的schema是 weixin:// ，支付宝的schema是 alipay:// 。关于schema的知识，后续再展开。

优点： 兼容性好。

缺点： 调用时延会比较高 200ms~400ms。

**注入式（Inject）**[#](https://doc.bytedance.net/docs/326/495/4571/#%E6%B3%A8%E5%85%A5%E5%BC%8F%EF%BC%88inject%EF%BC%89)

注入式，简单说就是通过JS引擎往window全局挂载方法，供H5调用。比如，平时我们使用的很多的alert，setTimeout这些JS方法，其实并非JavaScript本身提供，而是BOM。以此类推，我们一样是可以将一些自定义的Native方法，通过JS引擎已BOM的形式注入到全局供H5调用。

优点： 调用速度非常快，参照alert。

缺点： 低版本iOS系统不支持，安卓 4.1以下有安全漏洞，另外这种方式在调用的时候会阻塞Webview线程，有引起线程死锁的风险，所以要求编写注入方法的开发者编写时需要更严谨。

**请求式（Request）**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%AF%B7%E6%B1%82%E5%BC%8F%EF%BC%88request%EF%BC%89)

请求式，这种方式基本上很少见。在客户端启动一个小型的服务器，H5需要调用客户端的时候，通过往 127.0.0.1:port/path/ 发送网络请求来实现。

优点： 似乎没有。。。

缺点： 耗电，单工

JSBridge有多种方式可以实现，实现思路也不尽相同，以上是简单介绍一下我了解的几种，更多的实现方式还在探索中。多种实现方式最后本质是在兼容性和性能之间进行取舍，且并没有最优的方案。只有根据自己需要面对的业务形态，选择最适合的方案。

**Hybrid App类型划分**[#](https://doc.bytedance.net/docs/326/495/4571/#hybrid-app%E7%B1%BB%E5%9E%8B%E5%88%92%E5%88%86)

**多view/activity混合**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%A4%9Aview-activity%E6%B7%B7%E5%90%88)

一页NA，一页H5，纯H5，无端能力。

**单个view/activity混合**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%8D%95%E4%B8%AAview-activity%E6%B7%B7%E5%90%88)

NA与H5在一页页面内层叠，体验好，开发成本高，比较适合NA开发人员。应用场景，外卖or打车App内嵌入web地图，再叠加客户端控件。

**Web主体型**[#](https://doc.bytedance.net/docs/326/495/4571/#web%E4%B8%BB%E4%BD%93%E5%9E%8B)

PhoneGap，AppCan，Html+，Cordova，Ionic等，NA尽最大可能将所有端能力封装给H5使用，所有界面都是H5。

**多主体共存型**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%A4%9A%E4%B8%BB%E4%BD%93%E5%85%B1%E5%AD%98%E5%9E%8B)

NA开发和H5开发共存，需要一定技术前提，实现容器。



纵观Hybrid开发过程或者说移动开发过程，其实就是在H5和客户端直接找到一个合适的平衡点，选择什么位置作为平衡点主要是取决于业务形态。一般来说大部分有技术实力的公司都会选择多主体共存的开发模式，比如：美团，手淘，微信，京东。

**Native vs Web优缺点对比**[#](https://doc.bytedance.net/docs/326/495/4571/#native-vs-web%E4%BC%98%E7%BC%BA%E7%82%B9%E5%AF%B9%E6%AF%94)

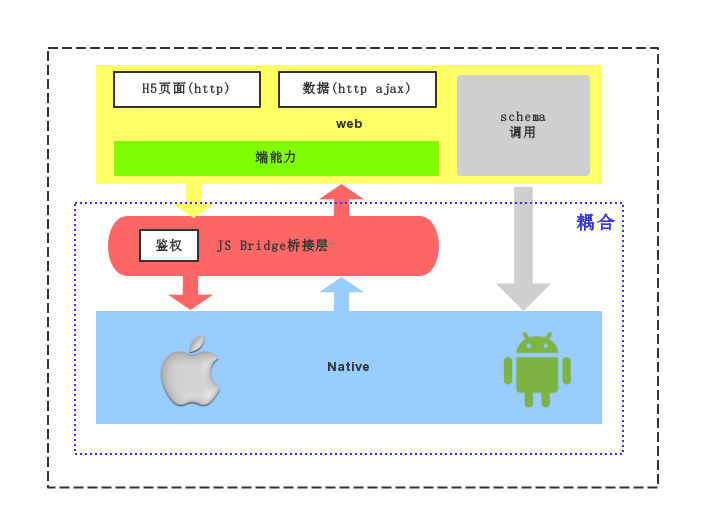
|  |  |  |
| --- | --- | --- |
|  | Native | Web |
| 优点 | \* 可做更多的底层优化（内存优化，预加载等）  \* 可做网络层优化（动态选路，http-dns等）  \* 可做重交互（视频拍摄，视频剪辑，视频合成，直播推拉流）  \* 可使用GPU实现更流畅的转场动画，各类动效 | \* 处理UI布局更快更方便  \* 一次编写跨平台使用，节省开发成本  \* 迭代速度更快，随时可以上线，上线即时生效  \* 不影响包大小 |
| 缺点 | \* 需求改动需要发版  \* AB test需要一定开发成本&发版  \* 随着产品形态完善，包大小会逐渐增加  \* 无法跨平台 | \* 页面载入需要一定时间，会有白屏时间，影响体验  \* 不适合做重交互的业务场景  \* 无法做底层的网络优化  \* Webview有一定的内存开销，特别是UIWebview有内存泄漏问题 |

通过对比，我们可以发现Native和Web相互的优缺点正好是有一定的互补作用，所以如果我们的Hybrid方案做的足够强大，我们是完全有可能最大限度的发挥Native和Web的优势，在保证体验的前提下，加速产品迭代。

IES Hybrid 现状 [#](https://doc.bytedance.net/docs/326/495/4571/#ies-hybrid-%E7%8E%B0%E7%8A%B6)

了解了Hybrid技术的基本架构和实现，下面再简单介绍下IES的Hybrid现状。

**IES Hybrid App架构**[#](https://doc.bytedance.net/docs/326/495/4571/#ies-hybrid-app%E6%9E%B6%E6%9E%84)



**Web方面存在的问题：**[#](https://doc.bytedance.net/docs/326/495/4571/#web%E6%96%B9%E9%9D%A2%E5%AD%98%E5%9C%A8%E7%9A%84%E9%97%AE%E9%A2%98%EF%BC%9A)

Web方面存在的问题可以拆解成3+1，4个部分，最后一个是黑盒&安全性问题：

**H5页面**

页面部分目前主要还都是走常规的http/https请求。需要在网络请求上耗费一定的时间（网络时延，DNS解析，加载静态资源等），无法解决一些网络劫持问题。

**数据请求**

数据请求部分也都还是常规的http/https ajax异步请求。只能走常规的ajax，无法解决网络劫持问题，无法做动态选路和接口加密。

**端能力**

我们统一的把通过JSBridge提供的跟客户端双向调用的能力称之为端能力。

**schema调用**

这个是一个黑盒的存在，本质上他也属于端能力的一个范畴，但是没走正规途径。正规的端内里调用都需要走JSBridge，在桥接层会先做一次鉴权，如果没有权限则拒绝调用，有权限才会调用客户端执行对应操作并回调，参考微信JSSDK。而schema是可以理解为客户端内的一个路由控制中心，用来统一控制端内页面的跳转。比如 A页面要去B页面，客户端应该是触发一个schema，如Router.go(“myschema://gotoB”)这样来进行跳转。而当我们在网页里打开 “myschema://gotoB” 时，也会被客户端捕获进而触发客户端路由跳转。这种方式是相当于调用到了客户端的底层路由，而绕过了【鉴权环节】，客户端底层路由是不应该直接暴露出来的，暴露出来的话，相当于任何知道schema的网页，都可以随意调用客户端能力，存在安全问题。

**Native方面存在的问题：**[#](https://doc.bytedance.net/docs/326/495/4571/#native%E6%96%B9%E9%9D%A2%E5%AD%98%E5%9C%A8%E7%9A%84%E9%97%AE%E9%A2%98%EF%BC%9A)

**鉴权**

目前鉴权这块鉴权api是固定的一个地址，多个App混在一起，无法区分App进行鉴权和授权，不利于后续打造整体开放平台。

**JSBridge和Native业务层耦合**

JSBridge桥接技术，桥接层应该是一个通用技术，IES多个客户端共享一套通用基础技术，而不应该各个业务线各自实现一套，重复造轮子。要复用的前提是基础技术要够纯粹，不能夹杂业务代码，比如之前的App里在桥接层，处理JS调用的时候是直接import客户端的一个类或者方法来处理，有时候还会在桥接层做一些业务逻辑判断（判断是否登录，取当前用户userid，打点上报等等），耦合比较严重，无法横向复用。

**其他问题：**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%85%B6%E4%BB%96%E9%97%AE%E9%A2%98%EF%BC%9A)

经过这一年多，IES发展迅速，从最开始的只有内涵段子一款App，到现在的内涵段子，火山小视频，火山直播SDK，抖音短视频，Flipagram，Hypstar（火山国际化版本），TikTok（抖音国际化版本）多个独立App，每个App需要最少开发iOS和Android两端，还需要考虑一些国际化方面的方案，如果每个App都单独做一套Hybrid方案，开发需要耗费大量的客户端开发资源，做重复造轮子工作。前端也没有办法一套H5页面代码运行在多个客户端，经常需要写很多的if...else...做兼容。各个客户端的文档也不尽相同，有的可能还不完善。再加上一些历史原因，一些比较老的业务线和比较老版本的App，JSBridge的实现和API定义也都比较随意，没有严格遵守的设计和实现规范，有时候都是因为在仓促着急的情况下，为了满足需求，快速怼出来，后续版本迭代&人员更替，一些接口经常没人维护也无法维护，经常是原有接口不管，单独重新开一个接口或者接口加一个type，客户端实现加个if...else...做判断来兼容。这个情况实在是不够优雅，对整体客户端的稳定性和前端开发的效率都不够友好。

综上，我们可以得出的结论是，我们IES目前的Hybrid现状还不够强力，或者说只是做到了提供端能力，还不能称之为方案，与业内大公司的差距还有很大，我们把目前的现在称之为Hybrid 1.0。

Falcon: Hybrid解决方案 [#](https://doc.bytedance.net/docs/326/495/4571/#falcon-hybrid%E8%A7%A3%E5%86%B3%E6%96%B9%E6%A1%88)

基于对业内移动端混合应用解决方案的调研，综合我们目前现状，我设计出了一套Hybrid 2.0方案，称之为 “Falcon”（猎隼）。

Falcon是一套极速&稳定的Hybrid 2.0解决方案，做方案设计的第一要素是脚踏实地出发，从现状出发，而不是架起空中楼阁，首先还是要承认我们目前确实是做的不够好，当然目前也有一些可取的地方，就是桥接层桥接协议的设计和实现非常不错，我们可以保留已有的好的设计，分析不足的地方，整体统一重新设计并一步一步实现。

每个领域都会有自己的一个领域边界，领域边界内的区域，即是运行环境或者容器。对于Java工程师来说，JVM就是Java的容器，而对于web前端工程师来说，我们的大部分工作都是跟浏览器打交道，浏览器就是我们的容器。所以我们设计Falcon方案，出发点，就是希望从容器的角度出发，为前端开发提供更强有力的能力，屏蔽底层运行环境差异，加速迭代速度，为客户端提供一套整套的解决方案，SDK，和接口协议规范，减少客户端开发重复工作，提升客户端代码可维护性。

Falcon的实战价值是：

从火山小视频客户端出发，试点重塑客户端Hybrid方案

整合IES全线产品Hybrid方案，统一设计，统一接口规范，统一维护文档和开发调试Demo页

为公司其他业务线Hybrid开发提供参考样本和技术输出，加速迭代

关于取名的插叙

Falcon，猎隼，隼科鸟类和鹰科鸟类同属于传统隼形目，然而隼与鹰雕等鹰科鸟类差异较大，在基于全基因组的分类系统中甚至抛弃其他猛禽，被单独分出去。

鹰喜欢在盘旋中发现目标，翅膀较宽，适合长时间空中停留，有力量和洞察力的象征。

而隼总是闪电般快速飞行，翅膀尖长，适合急速冲刺，有迅猛强悍的象征。

鹰体型大小因种类不同而有很大悬殊，而隼则通常都跟鸽子差不多大，当然即使个子再小的隼，捕食的时候依然是十足猛禽范儿。

所以，

Falcon，第一方面，是取隼鸟本身，迅猛精悍的含义。

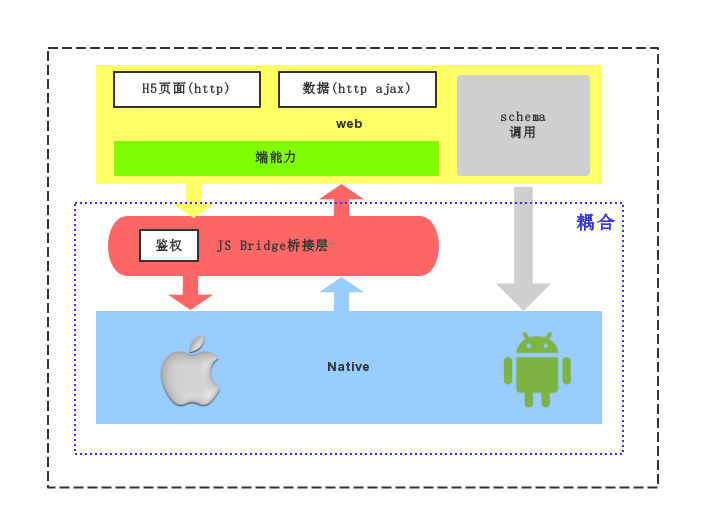
千年隼号（英文：Millennium Falcon） 是《星球大战》系列作品中一艘宇宙飞船，它的主要驾驶员是船长汉·索洛和他的副驾驶员楚巴卡。在电影中，千年隼是一艘经过汉·索洛大幅改装的YT-1300型货船，其特性能够很好地服务于汉的私人事务。千年隼是星战电影及星战其他形式文艺作品中出场次数最频繁的飞船之一。除了在经典三部曲中具有举足轻重的地位外，它还在前传第三部中客串出场，而在星战为数众多的小说及游戏等作品中也处处可见它的行踪，堪称经久不衰的不死鸟。英国广播公司在一篇新闻稿中评论道，按理说千年隼号可以算是这个宇宙中最著名的宇宙飞船了。

Falcon，第二方面，是取千年隼号飞船，经久不衰的含义。

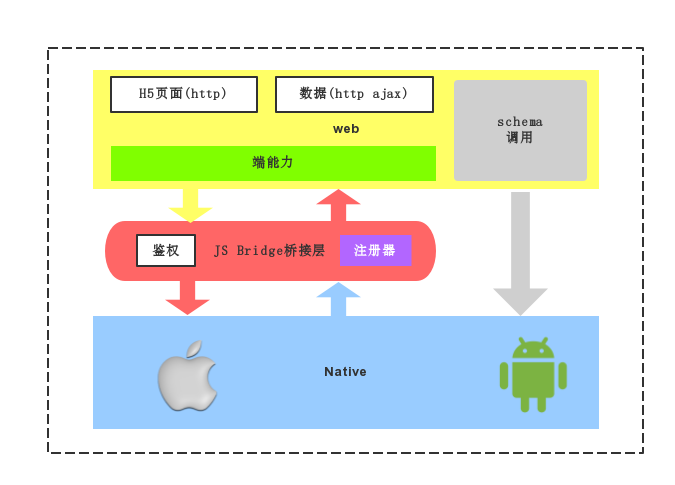
Falcon是一套极速&稳定的Hybrid 2.0解决方案，我们主要做了以下几个事情：

**架构解耦**[#](https://doc.bytedance.net/docs/326/495/4571/#%E6%9E%B6%E6%9E%84%E8%A7%A3%E8%80%A6)

前面我们提到了一个耦合的问题，就是之前JSBridge的处理都是在桥接层，直接引入的业务代码实现，相当于是引用式实现。单个App这么实现问题不多，但是遇到我们这种多产品线的情况就不太合适了，引用式的代码耦合太严重，桥接层无法横向复用，其他App无法直接接入。



我们做的第一件事，是把引用式JSBridge实现改成注册式的实现。简单说，就是我们希望桥接层做做纯粹的桥接的功能，联通H5和Native而不耦合业务逻辑，因此，我们在桥接层和Webview封装在一起，封装成IESWebview，并设计了一个注册机制，做一个注册器，Native端只需要接入IESWebview，并在初始化的时候调用其注册接口，将Native可以支持JSBridge方法注入进去即可。注册表里，其实就是一个Key-Value的Map，key是JSBridge的方法名，value就是客户端对应该JSBridge方法的实现。当H5调用JSBridge时，桥接层即是到注册表中去查找对应的处理方法，如果有则调用，没有即是返回调用失败。



*至于注册机制的实现：*

*Android端：使用类的继承来实现，定义一个BaseJSBridgeHandler，接入方只需要继承这个基类，实现一个自己的CustomJSBridgeHandler将自己需要支持的JSBridge方法写进去，初始化IESWebview的时候，实例化一个CustomJSBridgeHandler，传递给IESWebview即可。*

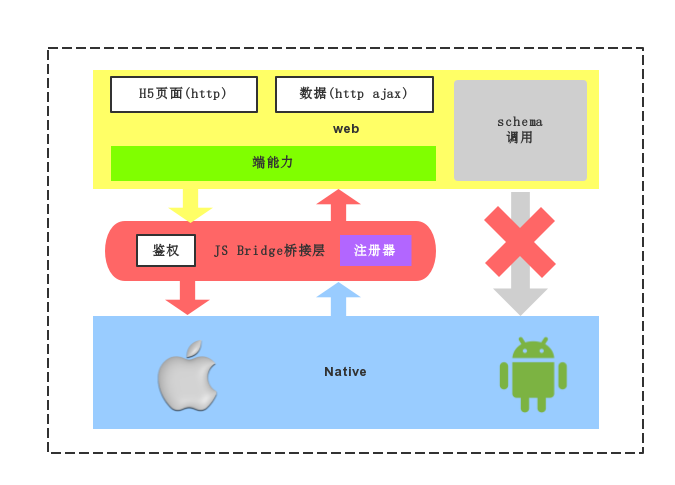
*iOS端：通过继承IESJSBridge，实现自定义的方法来实现，对UIWebView/WKWebView也做了处理，均会自动集成，无缝切换，业务方可直接使用。*

**路由整合**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%B7%AF%E7%94%B1%E6%95%B4%E5%90%88)

前面我们提到了客户端schema，schema是客户端的一个路由控制中心，客户端一些重要页面跳转均是基于schema来跳转。而我们的H5页面经常也有需要打开一些客户端的页面且是重要页面（比如：进用户profile页、进直播间、进视频播放页，进详情页等）。之前很多时候，都是代码里直接去唤起某个客户端的view/activity，我们觉得这样实现虽然问题不大，但是不够优雅，因此，所有进入/打开页面的操作，我们都收敛到一个openHotsoon方法中，再以各种type做区分，客户端处理openHotsoon方法是实现，就都是统一的把JSBridge传递的参数，转化成客户端的schema，然后走客户端的统一跳转逻辑。借此，整合H5和客户端路由。

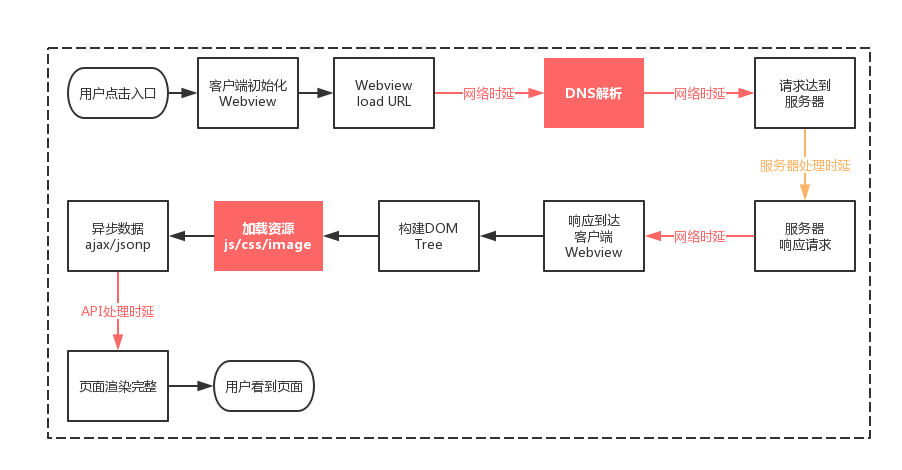
**权限收敛**[#](https://doc.bytedance.net/docs/326/495/4571/#%E6%9D%83%E9%99%90%E6%94%B6%E6%95%9B)

前面也提到了一个客户端黑盒般的存在，schema直接调用到客户端路由，存在一定的安全性问题，方案中，我们移除这种直接调用的方式，所有的H5→Native的调用都需要走JSBridge，都需要经过鉴权，鉴权服务器地址也是可以在介入IESWebview的时候进行配置，可以各个业务线各自把控。



**资源离线**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%B5%84%E6%BA%90%E7%A6%BB%E7%BA%BF)

解决了前面的问题，我们接着就是要解决H5的速度/性能问题，尽可能快的达到极速&稳定加载。我们先看一下传统流程上，从客户端点击一个cell或者banner到一个网页展现在用户手机上需要经过哪些过程。

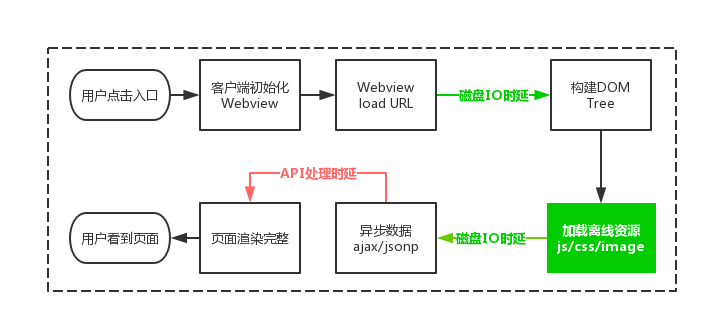


从上图我们可以看出，在这个过程中，很大一部分的时间开销是花在网络开销上（红色箭头），特别的是红色的两个关键节点（DNS解析，加载资源js/css/image）。

DNS解析需要一层一层往上查找直到顶级域名服务器，过程中需要经过比较多的运营商节点，经常也会遇到很多的劫持问题，影响请求达到率和引起用户反馈。

关于静态资源加载，在网页上，实际上HTML文档和API数据请求所占的大小和带宽都非常小，带宽消耗的大户是各类静态资源（js/css/image）等。

所以做优化我们第一步是先从大头的入手，解决资源加载的问题。这一块业内通用方案都是离线化，即是把资源文件提前下发到客户端，当需要加载资源文件的时候，从客户端本地读取，这样一来所有的资源传输网络时延，基本就都变成了客户端本地磁盘读取的IO时间了，基本上可以是 百毫秒 级别，甚至 十毫秒。进一步的，html文档也是一种资源，我们可以把html文档也离线化，这样一来除了异步数据之外，所以的时延就都是本地磁盘读写时间，我们的加载过程就变成这样：



为了印证我们这个修改方案的有效性，我们特意拿一个线上页面比较复杂的页面做了一个demo实验，选用的是火山小视频 - 直播金主排行榜页面。我们在线上有这个页面的性能监控，可以得到线上这个页面的DOM ready时延和load时延，我们将它改成了离线版，然后在客户端上加载离线版页面，打点计算这两个时间，得到离线版的时延测试数据。



我们对比一下在线版和离线版的时延数据，验证下方案：

|  |  |  |
| --- | --- | --- |
|  | DOM ready | load |
| 线上版 | 806ms | 955ms |
| 离线版 | 56ms | 561ms |

（线上版，我们取平均时延最低的一个时间段，计算平均值，实际线上全部平均要更高；离线版，此处只是单机多次测试数据平均，实际不同平台不同机型可能会有一些差异，时延可能会再高一些，此处离线测试数据，我们可以看做实验室数据）

我们先看load时延，差不多可以有一个最少2倍的提升。所以方案是值得继续深入去做的。

**域名拦截**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%9F%9F%E5%90%8D%E6%8B%A6%E6%88%AA)

**关于URL的思考**

到上面这一步，我们测试离线包的时候使用的还是file协议打开本地文件，虽然这种做法没什么毛病，但是我们觉得这种做法有些不够极致：

URL是file:// 开头的不好看，不优雅；

客户端内很多URL都是服务端下发的，下发file:// 协议的，服务端必须要知道客户端的绝对路径；

iOS平台和Android平台有一定的差异性，文件存储位置不同，无法共用一个相同的绝对地址，如果只维护最后的文件名，iOS打开 file://AAA/target.html，Android打开 file://BBB/target.html，又不太好后续维护和扩展；

使用file协议的话，如果离线包没有下载成功或者离线资源缺失/格式错误，客户端内打开不存在/有问题的文件资源就会出问题，也没有降级服务方案，这意味着最少需要在客户端内内置一个保底的离线包；

安全性，使用file:// 协议的是一个绝对路径，没有域名一说，对于JSBridge的话就没有办法做鉴权，也就意味着桥接层，需要对file:// 协议的地址单独开一个白名单特权，不利于权限收敛。

所以我们决定做 域名拦截。

简单说，就是我们在客户端里打开离线网页地址是正常的http/https链接，如：

https://hotsoon.snssdk.com/falcon/live\_app/page/inapp/rank\_rich\_anchor.html

然后走的逻辑流程跟file:// 差不多。

跟客户端同学经过调研之后，我们知道不管是iOS还是Android，在Webview上都是可以拿到请求URL的。所以我们决定在Webview上做一个域名拦截，同时将Webview在客户端上做一层封装成为IESWebview。IESWebview是一个SDK，主要做的是离线化域名拦截和注入基于注册机制实现JSBridge。

**IESWebview SDK需要的功能和设计**

我们要实现的就是拦截 https://hotsoon.snssdk.com/falcon/live\_app/page/inapp/rank\_rich\_anchor 这样的请求，将其转化成一个本地资源读取。如果我们将这个URL转化成读取本地的 Android/data/com.ss.android.ugc.live/cache/index.html 文件的话，那意味着如果我们维护做一个 URL-本地Res 的映射表，过于复杂，实现成本也会比较高。结合前面提到的，iOS和Android存储路径不一致的问题，我们想，文件路径不一致其实只是到存储根目录不一致而已，之后的基本是一致的。举个例子，刚刚的那个文件，web URL、iOS存储绝对路径、Android存储绝对路径分别是：

|  |  |
| --- | --- |
| 平台 | 路径 |
| web URL | https://hotsoon.snssdk.com/falcon/live\_app/page/inapp/rank\_rich\_anchor |
| iOS | Library/Cache/live/index.html |
| Android | Android/data/com.ss.android.ugc.live/cache/index.html |

路径越复杂，路径差异越大，文件查找复杂度越高&效率越低。

所以我们必须要想办法进行 路径整合，尽可能统一这三种路径，降低文件查找复杂程度。

在web环境中，URL本来就是表示一种资源定位的路径，于是我们先从web下手，再到客户端各个击破，把路径都改成这样：

|  |  |
| --- | --- |
| 平台 | 路径 |
| web URL | https://hotsoon.snssdk.com/falcon/ live\_app/page/inapp/rank\_rich\_anchor.html |
| iOS | Library/Cache/live/ live\_app/page/inapp/rank\_rich\_anchor.html |
| Android | Android/data/com.ss.android.ugc.live/cache/ live\_app/page/inapp/rank\_rich\_anchor.html |

我们可以看做三端的后半部分的path都是相同的，只是前面的起始根目录不同而已，web上来说，这个叫做baseline，web端的我们可以在自己随时配置URL，客户端的只需要在接入SDK的时候，各自设置一个各自的baseline即可完全屏蔽底层差异，文件查找规则也就变得非常方便，也不需要存在之前提到的 URL-本地Res 映射表了。

*把 web的URL设计成以 .html结尾的另一个好处是在于的降级服务的实现和稳定性，后面会说到。*

基于上面提到的，我们SDK接入时就需要有一些必须的配置，也可以叫输入。

接入时必须的要输入

prefix {Array}，一组拦截前缀，prefix必须包含域名和一个特殊标识的path，使用一个特殊标识的path是为了避免错误拦截在线页面，此处我们设置了 “hotsoon.snssdk.com/falcon/” 包含域名“hotsoon.snssdk.com”和特殊标识path “falcon”；

baseline {String}，客户端离线包存储根目录的路径。

域名拦截和文件查找过程

IESWebview拦截所有http/https的网络请求，拿到网络请求完整URL，“https://hotsoon.snssdk.com/falcon/live\_app/page/inapp/rank\_rich\_anchor.html”；

忽略http/https协议头，得到domain/path；为“hotsoon.snssdk.com/falcon/live\_app/page/inapp/rank\_rich\_anchor.html”；

将我们得到的domain/path和接入是配置的prefix挨个进行正则匹配或者字符匹配，因为我们可能会有多个产品线H5共通的需求，多个产品线域名又是不一样的，所以设计的时候设计了可以指定多个拦截prefix来解决这类业务需求，这样火山的离线包可以发布到段子、抖音等；

如果没命中匹配，则走正常的网络请求过程；

如果命中匹配，则取出prefix之后的path，我们称之为realpath。

*此处，我们设置的prefix是“hotsoon.snssdk.com/falcon/”，domain/path 匹配命中后取到的realpath即是“live\_app/page/inapp/rank\_rich\_anchor.html”*

对于命中匹配的，我们到离线包存储目录下进行资源查找，基于前面我们对路径的整合，使得我们的文件查找过程也就变得异常简单

(1) 将SDK中的baseline和刚刚得到的realpath进行拼接得到资源在客户端的绝对路径 abspath （baseline/realpath）；

(2) 接着只需要要在客户端调用文件系统提供的函数，判断这个绝对路径的文件是否存在即可 isExist(abspath)

如果路径对应文件存在，则直接返回该资源作为请求的response

如果路径对应文件不存在，则走正常的网络请求过程

以上举例的，对于html的离线化实现，对于js/css/image/... 其他任何资源文件的离线均适用，只要满足拦截prefix即可。

至此我们就完成了 资源离线 最关键的一步。

**离线存储**[#](https://doc.bytedance.net/docs/326/495/4571/#%E7%A6%BB%E7%BA%BF%E5%AD%98%E5%82%A8)

**资源路径**

鉴于我们前面做了路径整合，来实现的高效域名拦截&文件查找，下一个问题，就是我们要如何把离线文件按照 “live\_app/page/inapp/rank\_rich\_anchor.html”这样的路径存放在baseline离线根目录下。

我们的解决思路是，我们云端给客户端下发一个zip包，客户端下载成功后直接解压到baseline目录下，zip包解压后即是“live\_app/page/inapp/rank\_rich\_anchor.html”这样的路径。依旧逻辑和实现 简单。

**定制构建**

接着就是打包构建的问题，传统的前端工程，不管使用什么构建工具，我们一般到最后都是产出两个包：

output
|- template *//所有html模板，推送到后端服务器*
|- project\_name *//区分多个前端工程模板*
|- page
|- search
|- index.html
|- resource *// 所有静态资源文件，推送到CDN*
|- project\_name *//区分多个前端工程静态资源*
|- component
|- node\_modules
|- page
|- search
|- index\_f2fa68d.js
|- pkg
|- third\_31fb036.js
|- **static**
|- script
|- jssdk\_9528f52.js
|- style
|- base\_f48d0ff.css
|- image
|- ...

一个是存放html文件的template，最终要推送到自己对应的后端服务器，一个是resource目录将所有静态资源打包在一起，用来推送到CDN。

对于我们的离线化工程，跟传统前端工程有些不一样：

无须后端服务器，因为所有的html文件都是直接读的本地磁盘文件，而不是走网络请求，所有没有经过后端服务器，没有后端服务器也就意味着失去了服务端渲染的能力，不可使用 extend、include等高级后端模板语法，只能使用最基础的html语法，复用能力有限；

无须CDN，因为所有的静态资源理论上也都是可以直接读的本地磁盘文件，所以静态资源无须推送到CDN，同时也就无须配置CDN域名前缀了，传统的我们都是给静态资源配置['s3.bytecdn.cn', 's3a.bytecdn.cn']的随机静态资源CDN域名，此处我们直接走离线化即可，直接将静态资源前缀配置成 “hotsoon.snssdk.com/falcon/” ，触发域名拦截，走本地磁盘读取即可。

现在我们已经非常明确，我们要做的所有资源的离线化，基于路径整合的考虑，我们不能像传统前端工程一样进行打包（两个包），而是需要把html/js/css/image等都打成一个包 live\_app，统一下发给客户端即可。因此我们需要对构建进行定制化。

IES使用 mya集成构建&开发环境 （基于fis3）我们定制了一个构建脚本，参考配置如下：

**const** PROJECT\_NAME = 'live\_inapp';
fis.require('falcon')(fis, {
*/\* 项目名 \*/*
PROJECT\_NAME: PROJECT\_NAME,
*/\* 自定义占位符替换
\* > 线上包替换变量
\* eg. {'{{ GA\_ID }}': '123'}，
\* 则页面中的 '{{ GA\_ID }}' 会被替换为 '123'
\*/*
ONLINE\_REPLACER: {
'{{ GA\_ID }}': '123',
},
*/\*
\* > 离线包替换变量
\*/*
OFFLINE\_REPLACER: {
'{{ GA\_ID }}': '456'
}
});
*// 将工程中所有内容都构建输出到 template目录下*
fis.set('build.templatePublishPath', `template/${PROJECT\_NAME}`);
fis.set('build.staticPublishPath', `template/${PROJECT\_NAME}/res`);
*// 将所有静态资源的前缀替换成 //hotsoon.snssdk.com/falcon/*
fis.set('build.staticPath', `/falcon/${PROJECT\_NAME}/res`);
fis.get('setDomain')([
'//hotsoon.snssdk.com'
]);
*// 关闭MD5*
fis.match('\*', {
useHash: false
});
*// 利用hook注入环境变量*
fis.match('\*\*.{js,html:js,vue}', {
postprocessor: **function**(content, file, settings) {
**return** content.replace(/MYA\_ENV/g, **typeof** process.env.MYA\_ENV === "undefined" ? '\'PRODUCTION\'' : `\'${process.env.MYA\_ENV}\'`)
}
});

通过定制构建，我们最终输出脚本目录结构为：

output
|- template
|- project\_name *//区分多个前端工程*
|- page
|- search
|- index.html
|- res
|- component
|- node\_modules
|- page
|- search
|- index.js
|- pkg
|- third.js
|- **static**
|- script
|- jssdk.js
|- style
|- base.css
|- image
|- ...

最终我们只需要将project\_name目录打包成一个zip包下发给客户端即可。

**资源单例**

在定制构建中，我们有个配置是{ useHash: false }关闭文件md5，这么配置原因是，我们希望一个文件只在客户端存在一个版本（最新版），不希望保留历史版本的文件，以减少离线包对客户端存储空间的占用，所以我们选择关闭文件md5。

当然，熟悉前端开发的同学可能会问，关闭了文件md5的话，如何解决浏览器缓存问题？

经典的case就是，我们在浏览器里访问了离线包v1的网页 a.html，a.html 引用了 aa.js，访问过之后 aa.js 就已经被浏览器cache住。这时候我们给客户端下发新的离线包v2，我们修改了 aa.js ，再次访问页面 a.html 这时候读到的 aa.js 还是浏览器cache里的，导致页面不准确。

解决方案有三种：

打包静态资源的时候，在构建工具里定制，在静态资源后面加query打时间戳，“aa.js?timestamp”，来强制浏览器刷新缓存；

强制浏览器不缓存，通过网页meta设置cache策略。

我们采用的是第2种解决方案 - 强制不缓存

<meta http-equiv="cache-control" content="no-store">

最开始设置的是no-cache，后来觉得还是设置为no-store更保险。

关于 cache-control(指定请求和响应遵循的缓存机制)

最主要的用法是指导浏览器如何缓存某个响应以及缓存多长时间。Google Developers中给出的答案。

stickPicture.png

cache-control的content值和含义是：

1. no-cache: 先发送请求，与服务器确认该资源是否被更改，如果未被更改，则使用缓存。

2. no-store: 不允许缓存，每次都要去服务器上，下载完整的响应。（安全措施）

3. public : 缓存所有响应，但并非必须。因为max-age也可以做到相同效果

4. private : 只为单个用户缓存，因此不允许任何中继进行缓存。（比如说CDN就不允许缓存private的响应）

5. maxage : 表示当前请求开始，该响应在多久内能被缓存和重用，而不去服务器重新请求。例如：max-age=60表示响应可以再缓存和重用 60 秒。

**合并请求**

虽然我们使用了离线化的技术，但是浏览器依旧是单一时间内对同一个域名有最大并发请求量的限制。虽然我们读取资源很快，不过我们还是希望做到更极致，性能更好。

基于现有设计，我们想出了几个方案：

SDK可以配置多个拦截域名，多配置几个域名，随机分配给静态资源做前缀；

尽可能合并请求量。

最后我们选择了第2个方案，合并请求，原因是：

一般来说，在一个文件系统里读文件时，文件查找和读文件是开销比较大的，而读100k和读200k的文件其实差别不是很多，所以，读一个大文件比读N个散列的小文件要划算的多，可以节省很多次域名拦截、文件查找和读文件的开销；

是和我们后面要做在线化降级服务的部署相关，多域名会增加部署的复杂性。

所以，文件读取方面性能增强的大体思路是：单域名，大文件

实现方面，

我们使用fis3的资源嵌入功能（inline），把布局图片转成base64内嵌到css/html中，把一些基础库的js/css内嵌到html模板中；

借助于 mya，我们在构建过程中对每个页面的依赖进行分析，将每个页面需要依赖的多个node\_modules进行合并，合并成一个/pkg/third.js 进一步减少因为引入第三方库带来的请求开销。

我们前面提到了我们失去了后端高级模板的能力，使得我们无法复用一些html的问题，在做资源内嵌时，我详细阅读了fis3的文档得到了启发，我们可以借助fis3的资源嵌入功能一定弥补我们的后端模板能力。后端模板里比较常用的语法之一是 include ，写过PHP、Python web、Java web的同学应该都很熟悉。通常我们会把公用的html代码片段抽取出来，在需要使用的页面include进来。同理，在离线工程开发过程中，我一样会把公用html片段抽取出来，放到一个目录里（注意：该目录不需要发布，构建时需要略过。），然后在需要用到的地方使用fis3的嵌入，借此达到代码复用，提供类似于include的功能。这也算是编码过程中的意外收获 :-)

以下是资源嵌入示例代码：

**<!DOCTYPE html>**
<html lang="en">
<head>
<meta charset="UTF-8">
<title>示例代码</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=0, minimum-scale=1.0, maximum-scale=1.0, minimal-ui">
<meta name="format-detection" content="telephone=no">
<meta name="keywords" content="火山小视频">
<meta name="description" content="火山小视频(Huoshan.com)">
<link rel="shortcut icon" href="/static/image/logo/favicon.ico" type="image/x-icon">
<meta http-equiv="X-UA-Compatible" content="IE=Edge;chrome=1">
*<!-- 强制不缓存 -->*
<meta http-equiv="cache-control" content="no-store">
*<!-- DNS 预解析 -->*
<link rel="dns-prefetch" href="//p.pstatp.com/">
*<!-- 样式表嵌入 -->*
<link rel="stylesheet" href="/static/style/layout.scss?\_\_inline">*<!--ignore-->*
*<!-- 脚本嵌入 -->*
<script src="/static/script/lib/adapter.js?\_\_inline"></script>*<!--ignore-->*
<script src="/static/script/lib/ies\_jssdk.min.js?\_\_inline"></script>*<!--ignore-->*
</head>
<body>
*<!-- html片段嵌入 -->*
<link rel="import" href="/page/include/demo.html?\_\_inline">
<div id="demo"></div>
<script src="/static/script/lib/mod.js?\_\_inline"></script>*<!--ignore-->*
<script type="text/javascript">require('index.js');</script>
</body>
</html>

**目录规范**

除了对离线化工程的定制构建和特殊处理，我们还有考虑到另一个问题，就是分频道的话要如何处理。电商、OTA中经常会遇到这种问题，当整个App做的比较大的时候，就会分很多的垂直频道、也叫垂类，比如：OTA里的机票、酒店、火车票，电商里的3C、商超、生鲜、到家等。一个频道就是一个tab，一般有一个单独的团队负责开发和维护，App里有多个频道，就意味着需要支持多个团队的项目离线化，显然，如果都在一个包live\_app里写和下发不合适，甚至会相互影响。于是我们在离线包分发的时候支持了多频道离线包的分发。并约定打包时的目录规范如下

output
|- template
|- project\_name *//区分多个前端工程 -> 频道*
|- page
|- search
|- index.html
|- res
|- component
|- node\_modules
|- page
|- search
|- index.js
|- pkg
|- third.js
|- **static**
|- script
|- jssdk.js
|- style
|- base.css
|- image
|- ...

一个频道对应一个仓库，一个仓库对应一个唯一的project\_name，以project\_name目录来做频道相互隔离。并且约定频道之间不能有相互的依赖，以降低跨项目依赖打包带来的维护复杂度。

最后频道A使用project\_a包，下发 https://hotsoon.snssdk.com/falcon/project\_a/page/search/index.html 来访问。

频道B使用project\_b包，下发https://hotsoon.snssdk.com/falcon/project\_a/page/search/index.html 来访问。

**在线降级服务**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%9C%A8%E7%BA%BF%E9%99%8D%E7%BA%A7%E6%9C%8D%E5%8A%A1)

前面我们在离线化中提到一个降级服务的概念，在域名拦截个文件查找时，如果找不到本地离线的文件则走线上请求。

会走到这种降低服务流程的，一般只有几种case：

离线包还没下载解压完，用户就开始访问离线网页URL；

离线包损坏，导致解压失败；

本地资源被其他程序篡改、删除；

本地资源由于某些原因损坏（SD卡损坏等）。

所以我们需要部署一套在线的降级服务措施，保证即使离线化流程失败了，用户也能正常访问离线网页，保证最起码的功能可用性。也是为了做这一步，我们才使用正常的http/https协议而不是file协议。

我们先回顾下离线流程：

用户访问 https://hotsoon.snssdk.com/falcon/live\_app/page/search/index.html ；

域名拦截，得到realpath为live\_app/page/search/index.html；

baseline与realpath拼接，文件查找；

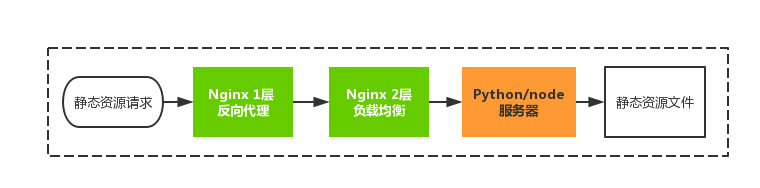
命中返回本地文件。

抽象一下，这个过程的本质就是一个 Native静态资源服务器。

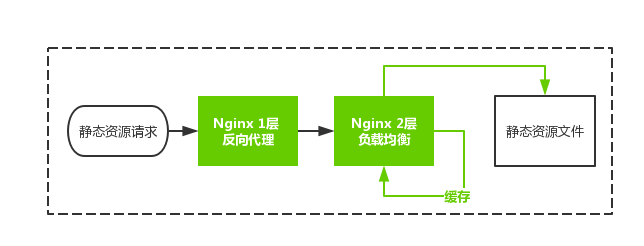
可以把 https://hotsoon.snssdk.com/falcon/ 看成是一个web的baseline，所以我们要做的在线降级服务，跟本地离线化的区别其实就是baseline。问题也就变得很简单了，我们只需要部署一个 在线静态资源服务器 就可以啦。

一说到静态资源服务器的搭建，有的同学可能会想简单搭一个python的Django/Flask或者node的Koa/Express，静态资源直接丢webroot目录下不就行了嘛。但其实我想说，这并不是最佳实践。

我们先来看看这种思路的整个资源请求过程：



除了两层Nginx之外，我们还需要启动一个服务器，这个服务器只做静态资源读取，这么做其实有点浪费，还徒增运维成本。前面标绿色的两层Nginx其实是非常稳定且运维成本很低的。加之Nginx除了做反向代理和负载均衡之外，其实是可以做静态资源服务器的，所以，最佳实践是通过Nginx来做。第二层Nginx直接读取静态资源/可缓存即可，把整个资源请求过程改成这样：



*最佳实践的思路是没问题的，不过最后我们由于部署方面的技术原因没有采用。*

nginx静态资源服务器配置示例：

server {
listen 443;
server\_name hotsoon.snssdk.com;
access\_log logs/host.access.log main;
# static files server(html|css|js|image..)
location ~ ^/falcon/ {
root /opt/tiger/falcon/;
access\_log off;
expires 3d;
}
# TODO: redirect server error pages to the static page 5xx.html/404.html;
}

**开发体验**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%BC%80%E5%8F%91%E4%BD%93%E9%AA%8C)

传统的C端开发，是通过网络完成整个页面访问过程，以前都是2G/3G网络，网速比较慢，用户也有各种参差不齐的移动设备，需要考虑设备的浏览器兼容性，也需要考虑低端移动设备的内存占用。所以我们一般开发是使用原生JavaScript/Zepto + underscore前端模板引擎来做。

从业内看，前端开发，组件化是一个大趋势。这几年各类优秀的MVVM框架层出不穷。所以我们觉得我们C端开发也应该进行技术升级。

C端开发技术升级，是综合考虑了一下几点：

时至今日，4G/4G+已经非常普及，5G也快出来，基于这样的网速下，传输多几十k内容差别并不大，更何况我们的资源都是离线化的。

我们Falcon是在自己的App客户端内使用，我们的客户端iOS最低支持到iOS8，Android基本是从Android 4.x开始支持，再往下的基本可以不考虑。就App支持的这个最低系统版本来说，很多的JS新的高级特性都已经支持了。

所以，我们使用的新技术栈是：npm + vue + ES6

对依赖都进行模块化封装，使用npm做依赖管理；

使用vue做MVVM双向数据绑定，借助vue单文件组件做组件化；

引入data Bus和vuex做数据流管理；

使用更严谨的ES6语法编写代码。

在本地开发时，我们借助 mya 集成开发环境，在本地启动一个node server，随时修改代码，随时发布到node server中，提供请求转发、mock数据、live reload、vue source map等一系列功能。

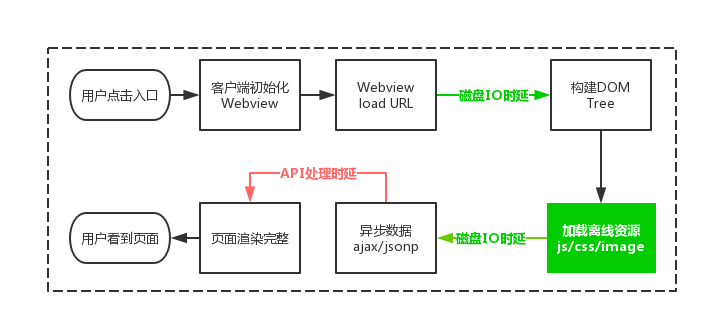
另外我们还在 mya 中提供离线工程的初始化脚手架工具，使用一下命令，填写几项必要的工程设置即可开始开发。

mya init falcon

*关于mya更多的功能请参阅mya相关的介绍文章*

本地开发完成之后，只需要把代码提交入库，发布到内测版测试，测试通过开始下发给客户端即可。

**请求代理**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%AF%B7%E6%B1%82%E4%BB%A3%E7%90%86)



资源离线加载是省去了资源网络时延，接着在这个图上，还有一个API处理时延会影响页面展现的时间。

API处理时延主要是分两段时延：

请求到达服务器的时延；

服务器处理请求的时延。

通过跟服务端同学交流得知，我们大部分的接口都已经做过优化和缓存，基本都是毫秒级响应。因此API处理时延主要还是花在网络传输上。再加上一些网络问题，DNS劫持、运营商劫持、网络节点/链路不稳定等问题，不仅仅是影响时延，甚至会影响请求到达率，造成一定的白屏。

通过跟客户端同学交流得知，端上一般是在底层网络库上做优化，来尽可能解决时延和劫持等网络问题。

客户端网络层有几个比较优秀的技术：

动态选路技术，客户端会通过测速，来选择API的最佳线路，动态切换；

多域名cookie共享，主要是为了支持动态选路；

HttpDNS，业内比较优秀的网络技术；

接口签名加密，对anti-spam；

HttpDNS 是一种业内比较优秀的DNS域名解析方案，原理也是比较简单，主要分两步：

客户端直接访问HttpDNS接口，获取业务在域名配置管理系统上配置的访问延迟最优的IP；

客户端向获取到的IP后就向直接往此IP发送业务协议请求。以Http请求为例，通过在header中指定host字段，向HttpDNS返回的IP发送标准的Http请求即可。

关于HttpDNS的详细介绍见参考文档

*IES HttpDNS目前还在开发中，有规划，暂未投入使用*

鉴于此，我有了一个想法，我们应该把客户端上这些优秀的网络应用和web共享，解决web上网络问题的痛点，共同在移动端打造高效、安全的网络环境。

所以，网络技术共享的基本思路就是 请求代理。

要实现请求代理，有两种方案：

主动代理，跟离线化的域名拦截类似，客户端在Webview层拦截所有网络请求，先触发离线化匹配流程，匹配中的走离线化，没匹配中的，走正常网络访问，将走正常网络访问的请求，转给客户端底层网络库，由客户端底层网络库做请求代理。这种方式优点是简单、对web接入方也是透明；

被动代理，通过JSBridge封装一个方法提供给web使用，web在需要使用的时候方去调用。

主动代理从理论上来说应该是比较好的方案，但是通过研究我们发现，这种方案也不是没有缺陷的：

虽然是主动代理，对web透明，但是依然会受浏览器跨域策略的限制；

在Android 4.x比较低的版本里，客户端只能捕获到请求的URL，其他的内容比如：query参数、POST body、header等拿不到。

考虑到上述的问题和需要对用户机型的兼容，我们最终实现的方案是被动代理。在JSBridge上封装了一个fetch方法来着请求被动代理。在需要调用重要接口的时候，我们会调用fetch，把URL、参数等传给fetch，由客户端来代理改请求。因为是调用的JSBridge所以也就不存在跨域问题了，同时客户端会自动帮我们做cookie跨域共享、接口签名加密。

对于fetch方法是设计，我又在想另一个问题，有时候，一个H5页面既能在客户端内打开，也能被分享到客户端外，在客户端内我们可以使用JSBridge来做请求，而在客户端外，JSBridge是使用不了的，需要做一个兼容，使得一套代码端内端外都可用。思路是对发送请求的前端网络库进行一层封装，如果是在端内就使用JSBridge，如果是在端外就使用web正常的XHR请求。鉴于现代化前端一般都是使用axios，所以，对fetch方法的参数设计，我全部参照axios进行定义，方便做端内端外的无缝切换。

{
"url": "xxx",
"method": "GET/POST",
"params": {
"ID": 6666
},
"data": {
"firstName": "Arthas"
}
}

**资源分发**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%B5%84%E6%BA%90%E5%88%86%E5%8F%91)

说完了前面的离线化和端能力，现在我们再说说离线化的一个关键支撑技术点 —— “离线包的分发和管理”。

对于分发的解决方案，我们设计了 Gecko：统一分发系统。

Gecko主要有以下几个主要特点：

**业务无关**[#](https://doc.bytedance.net/docs/326/495/4571/#%E4%B8%9A%E5%8A%A1%E6%97%A0%E5%85%B3)

分发，逻辑上即是，客户端在App启动或者其他自定义时机时，调用检查更新的API询问是否需要更新，如果不需要更新则不做任何事情，如果需要更新，API会返回给客户端一个更新包的CDN地址，客户端用这个地址去下载一个zip压缩包并解压到对应目录。至于解压完之后怎么用这个包，什么时候用这个包，均是业务方决定。资源分发逻辑与业务无关的。

**内容无关**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%86%85%E5%AE%B9%E6%97%A0%E5%85%B3)

zip包的本质是一个二进制文件，至于解压完的内容是什么并不重要。所以只要是可以打包成二进制zip的文件，都是可以被分发的。分发与内容无关。

**全量&增量**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%85%A8%E9%87%8F-%E5%A2%9E%E9%87%8F)

全量下载和增量下载是文件下载的老生常谈，全量是最简单暴力的实现。常见的增量下载策略一般是 基于文件粒度的增量，实现起来比较简单，原理就是一个工程文件夹下，那个文件有修改就下发那个有修改的文件的新版本。

而在Gecko中，我们觉得基于文件粒度的增量还不够极致，还可以做的更细粒度，我们采用的是 基于内容粒度的增量，简单说就是一个文件中我们如果只修改了一行，我们就只下发修改的这一行而不是这整个有修改的文件。跟Git的diff和merge类似。

由于我们下发的包都是二进制的zip，所以在实现增量的时候，我们使用了 bsdiff 算法 来做基于二进制的差量计算和patch。

关于bsdiff算法和实现见参考文档。

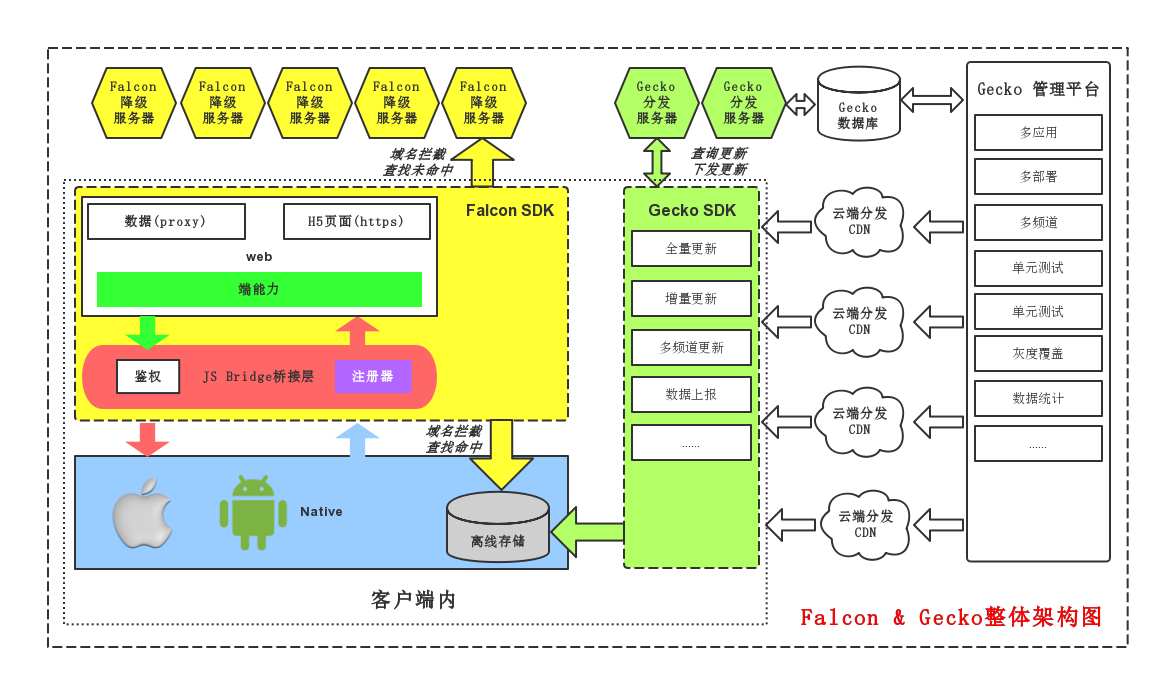
**灰度分发**[#](https://doc.bytedance.net/docs/326/495/4571/#%E7%81%B0%E5%BA%A6%E5%88%86%E5%8F%91)

**覆盖率统计**[#](https://doc.bytedance.net/docs/326/495/4571/#%E8%A6%86%E7%9B%96%E7%8E%87%E7%BB%9F%E8%AE%A1)

灰度发布和覆盖率统计也是Gecko提供的对于资源分发的功能。

篇幅有限，更多关于Gecko的设计、架构和实现细节详见[整体介绍-架构设计](https://doc.bytedance.net/docs/326/495/4553/)一文。

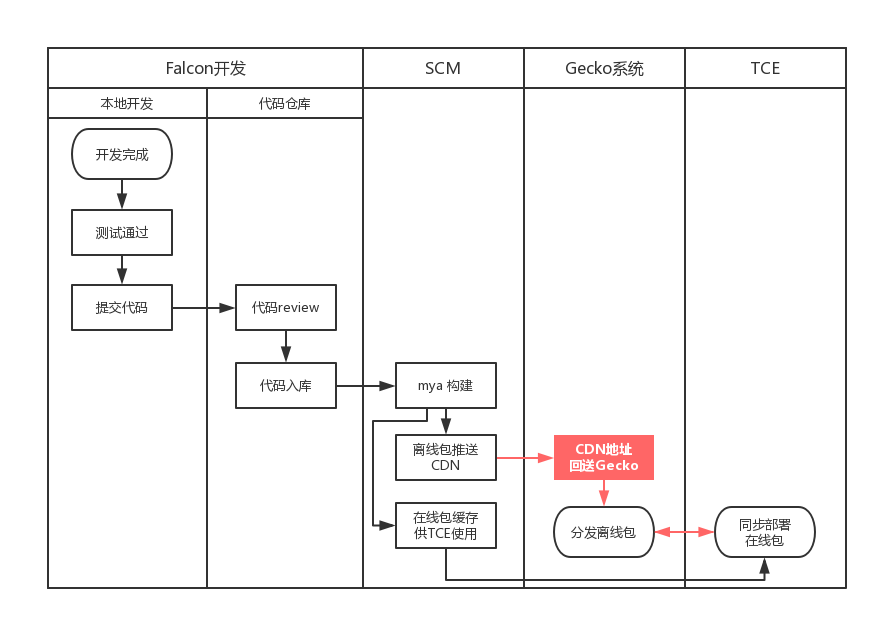
Falcon与Gecko [#](https://doc.bytedance.net/docs/326/495/4571/#falcon%E4%B8%8Egecko)



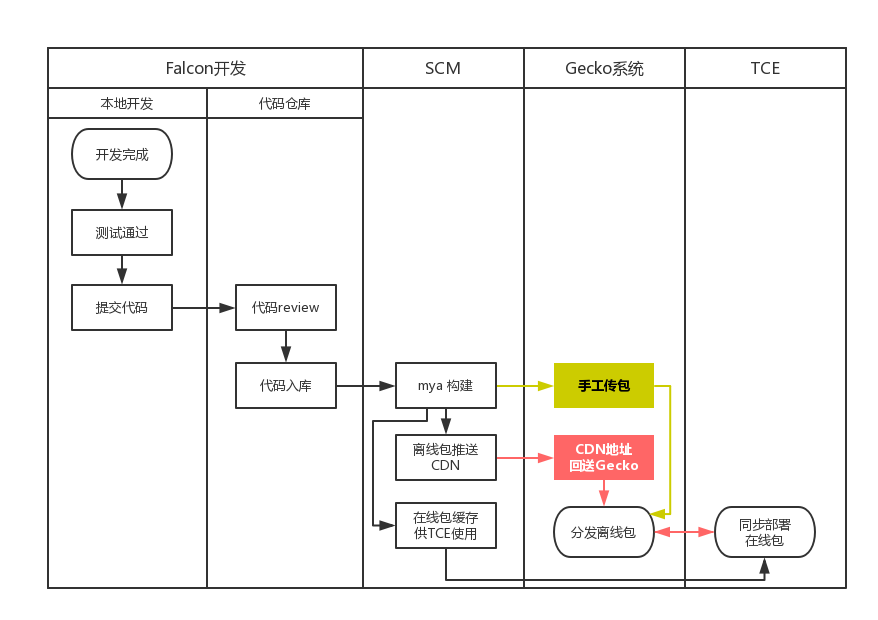
持续集成 [#](https://doc.bytedance.net/docs/326/495/4571/#%E6%8C%81%E7%BB%AD%E9%9B%86%E6%88%90)

开发完了Falcon方案和Gecko统一分发系统，最后我们要做的一件事情就是持续集成，打通开发、构建、部署（在线服务和离线包）、发布多个环节。目标是使用者开发起来尽可能简单，提交代码到发布上线环节尽可能的少。

以下是我们规划的持续集成是离线效果图



除了红色的部分之外，其他的我们都已经完成，红色部分主要是 打通SCM 还有 TCE同步上线 的问题。这块我们还在努力中。鉴于CDN地址回送Gecko这块现在还没法打通，我们先做了一个一期的临时解决方案。



临时在SCM构建完成后，下载SCM构建出来的包，手工到Gecko平台传包，由Gecko调用接口上传离线包到CDN并记录CDN地址下发给客户端。

这个只是我们折中做的一个临时方案，未来我们会在持续集成上继续努力。

工程设计 [#](https://doc.bytedance.net/docs/326/495/4571/#%E5%B7%A5%E7%A8%8B%E8%AE%BE%E8%AE%A1)

**关于设计的几点看法**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%85%B3%E4%BA%8E%E8%AE%BE%E8%AE%A1%E7%9A%84%E5%87%A0%E7%82%B9%E7%9C%8B%E6%B3%95)

**“懒”是第一生产力**[#](https://doc.bytedance.net/docs/326/495/4571/#%E2%80%9C%E6%87%92%E2%80%9D%E6%98%AF%E7%AC%AC%E4%B8%80%E7%94%9F%E4%BA%A7%E5%8A%9B)

一直以来人们都说“懒人推动社会进步”，从农业时代到工业时代，这个说法一直被印证。到了移动互联网时代也依旧是成立的。工程师的“懒”，不想重复写代码，不想重复copy代码，才会催生出各种抽象、各种封装、造出各种框架、各种工具，推动技术升级。

**实战是检验设计的唯一标准**[#](https://doc.bytedance.net/docs/326/495/4571/#%E5%AE%9E%E6%88%98%E6%98%AF%E6%A3%80%E9%AA%8C%E8%AE%BE%E8%AE%A1%E7%9A%84%E5%94%AF%E4%B8%80%E6%A0%87%E5%87%86)

作为一个方案的设计者，在设计一个方案的时候，出发点一定是要有实用性和易用性，设计一定是为了提高生产力，切忌为了设计而设计。

如何检验设计呢？

很简单，拿到一线业务中实战一下就知道了。Falcon实现出来之后，我第一件事就是自己把火山小视频客户端内原来在线版的H5功能页面全部改成离线书写，看看自己编写、调试、联调、上线整个过程爽不爽。同时也邀请组内同学尝试书写几个离线页面，体验一下，倾听用户反馈，采纳合理意见。

**Falcon的设计理念**[#](https://doc.bytedance.net/docs/326/495/4571/#falcon%E7%9A%84%E8%AE%BE%E8%AE%A1%E7%90%86%E5%BF%B5)

从个人角度来说，我特别讨厌记忆各种繁琐冗长的配置，我想要的是 简单可依赖，易懂好上手，横向能复用。

因此，Falcon的设计理念就是四个字：大道至简。

Falcon的整体设计和细节设计，我均尽最大努力遵循 “约定优于配置” 的软件设计范式。e.g: 路径整合，文件查找，离线存储等

*约定优于配置（convention over configuration）* *，也称作按约定编程，是一种软件设计范式，旨在减少软件开发人员需做决定的数量，获得简单的好处，而又不失灵活性。*

**7.httpDns**

参考：[HttpDns 原理是什么](http://www.linkedkeeper.com/171.html)

**8.动态生成正则表达式**

参考：[js如何生成动态正则表达式？](https://blog.csdn.net/u011031396/article/details/44810821)

**9.Vue created vs mounted**

参考：[Vue生命周期中mounted和created的区别](https://blog.csdn.net/xdnloveme/article/details/78035065)

created:在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。

mounted:在模板渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作。

**10.**[**点击其他区域,input输入框不失去焦点**](https://blog.csdn.net/qq_24581629/article/details/75175250)

**11.**[**vue 如何获取input中光标位置，并且点击按钮在当前光标后追加内容**](https://segmentfault.com/q/1010000017101606)

**12.**[**js字符串替换 - 无replaceAll替换所有字符串的解决方案**](https://blog.csdn.net/chwshuang/article/details/52583496)

var result = str.replace(/-/g,'');
console.log(result);

**13.**[**JavaScript正则表达式分组模式：捕获性分组与非捕获性分组及前瞻后顾(断言)**](https://blog.csdn.net/lihefei_coder/article/details/53022253)