## 项目背景

纯原生的应用拥有最丝滑的体验，但是App迭代会受限于应用市场的审核周期和审核规则，尤其是最近特别时期，各大市场审核越发严格，发版工作严重受阻，安卓可以用应用内更新以及热更新的方式来弥补，但是iOS上却无能为力。而web应用相较于纯原生，拥有即发即生效的优势，迭代不受外界阻碍，但是 体验一直是一个无法根治的问题。为了将两者的优势结合，劣势互补，我们需要一种混合方案，来让 web应用拥有接近纯原生的丝滑体验。

经过不懈的努力，最终完成了一整套App混编方案，让H5模块秒开加载，并且与纯原生部分，Flutter部分有机结合。

## 来龙去脉

原App是一个基于 flutter webView的web应用，业务分布在两个部分，flutter 和 h5。 由于 flutter webView 处于一个不稳定的状态，并且兼容性较差，与纯原生webView 对比起来，体验和兼容性更差，某些问题根本无解。所以为了让H5应用的体验得到优化，我们采用原生webView作为替代。而为了复用原有的flutter模块，我们引入了 混合栈框架flutterBoost，并且搭建了H5-native-flutter三层代码之间的通信架构。经过长时间的迭代，我们将 核心代码独立出来形成SDK，使得其他app（flutter搭建，或者原生搭建）接入之后可以直接享受到web应用的丝滑运行，打破局限，极限接近原生体验。

## 核心特性

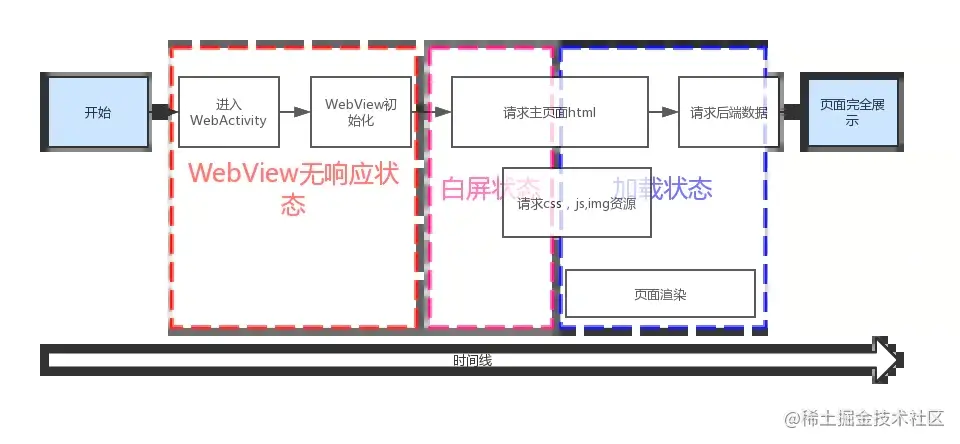
我们探索了一套混编开发方案，它既保证了纯原生的丝滑体验，又能让 web业务模块无延迟更新迭代。它是以 纯原生App 作为web容器，web应用模块 作为 **动态加载插件**的方式来运作。用前者来保证体验，用后者来确保迭代。

主要解决3个关键问题：

1. 提升web页面的打开速度
2. 支持web应用的即时更新
3. 让web应用使用原生能力

### 提升web页面的打开速度

webView加载一个url时，常规流程如下：



移动端 webView 加载H5页面的过程大概分为以下3个阶段：

* webView 无响应状态
* webView 白屏状态
* webView 加载状态

第一个阶段，webView 尚未创建，而webView初始化的过程其实是加载浏览器内核，尤其是在打开web首页时，以及打开web子页面时感受较为明显，前者加载内核会肉眼可见的慢。但是解决方式也很简单，在打开页面之前，**提前初始化webView**，让浏览器内核提前加载，可以一定程度上提升首次打开的速度。

第二个阶段，此时webView 开始请求网络资源和网络接口，此时网络速度直接决定了页面打开的速度。要想让资源加载加快，我们采用了 **资源离线化**的处理方式，让静态资源（html，css，js以及其他资源 ）打包放置在 App内部，并重写**webView的资源拦截函数**（android/iOS均有）让本来指向**网络资源**的返回值 直接指向**本地资源**。同时为了保证web应用的即时更新，本地离线资源要与远端的 远程资源实现**协同机制**，让App端的web离线资源始终与远端保持一致。这样就同时解决了加载速度和web即时发布的问题。而，网络接口方面，纯原生的http请求比 webView 自身执行的网络请求有明显的速度优势，我们在原生端提供 **http原生接口**，让H5直接使用原生层去**代为执行网络接口**，同时返回结果给H5，能提升一定加载效率。

第三个阶段，web加载状态。此时要想在视觉上让用户感觉加载很快，就要求H5的渲染顺序上做出优化，开始加载时先展示出**页面框架以及loading**动画，有多个接口时，让接口返回结果**单独与组件对应渲染**，而不是所有接口都完毕之后再渲染。

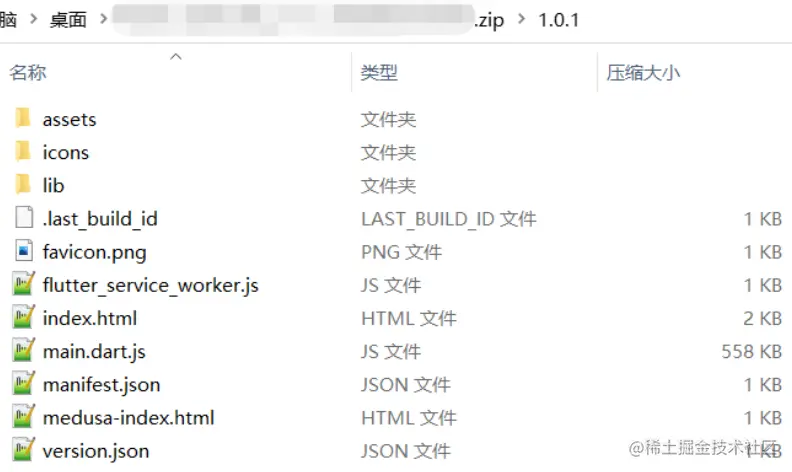
### 支持web应用的即时更新

在本方案中，App本身是一个webView容器，容器上运行的是web应用，这两者的关系类似“插件化”的概念，webView 容器作为宿主，web应用作为插件。而插件必须依托一个托管平台。宿主在特定时机与 托管平台进行通信更新插件，再去运行插件代码。

解决即时更新，需要 App端和 托管平台合作完成。

#### 托管平台

web应用，打包成zip包，任何你想要离线的静态资源都可以通过打包的形式放到zip包内，我们称为“离线包”，每一个离线包都代表一个web应用。



##### 支持离线包的上传

托管平台就类似一个离线包的“池子”，在复杂的业务场景中，多个业务方上传各自的离线包进入“池子”，并附带上每个离线包的配置信息，就是上图中的**manifest.json**文件。

##### 支持离线包与 宿主App的关联绑定

**manifest.json** 是 每个web应用的身份配置信息，内容展示如下：



上图中规定了一个web应用的一些关键元素。

**entryUrl** ： web应用入口地址，在webView启动应用时将会去加载这个url。

**version**： 应用版本号，提供web应用的版本管理。

**updateStrategy**： 更新策略。一个web应用可以支持多个安卓或iOS应用，上面支持配置原生应用的包名，版本号（支持 **^2.1.0**这种写法的模糊匹配）。

当App去通过网络请求拉取离线包时，需要传入 平台标识（Android/iOS），AppId（App包名），AppVersion（App版本号），接口从 离线包的“池子” 中根据 所有离线包的 updateStrategy 进行检索，找出当前App能够使用的最新离线包列表。

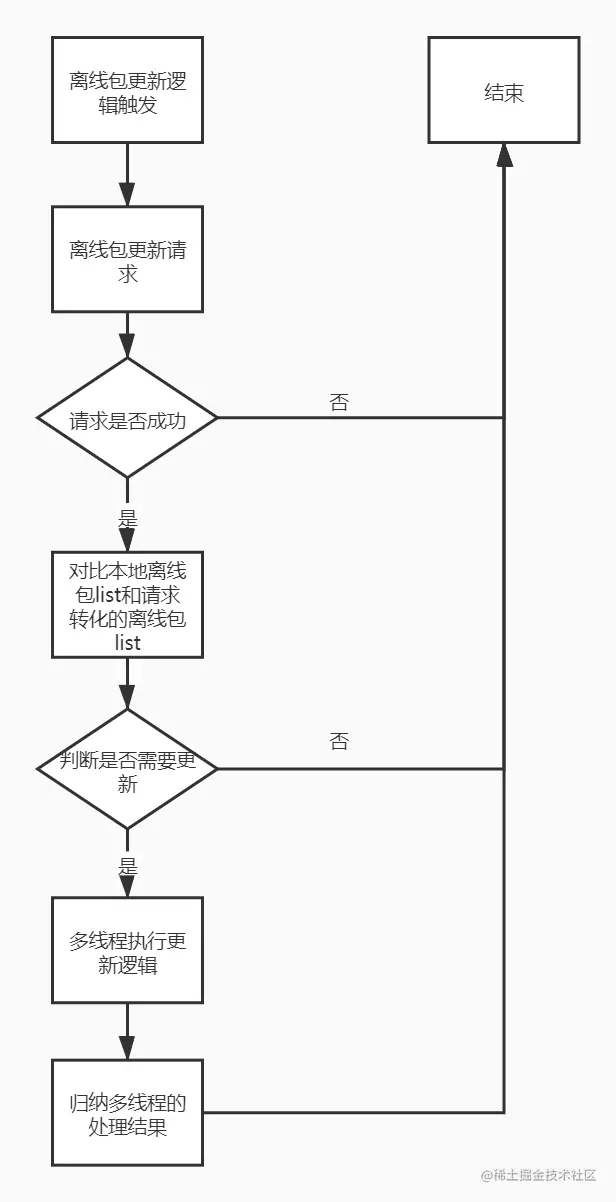
#### App端

##### 确保离线包始终最新

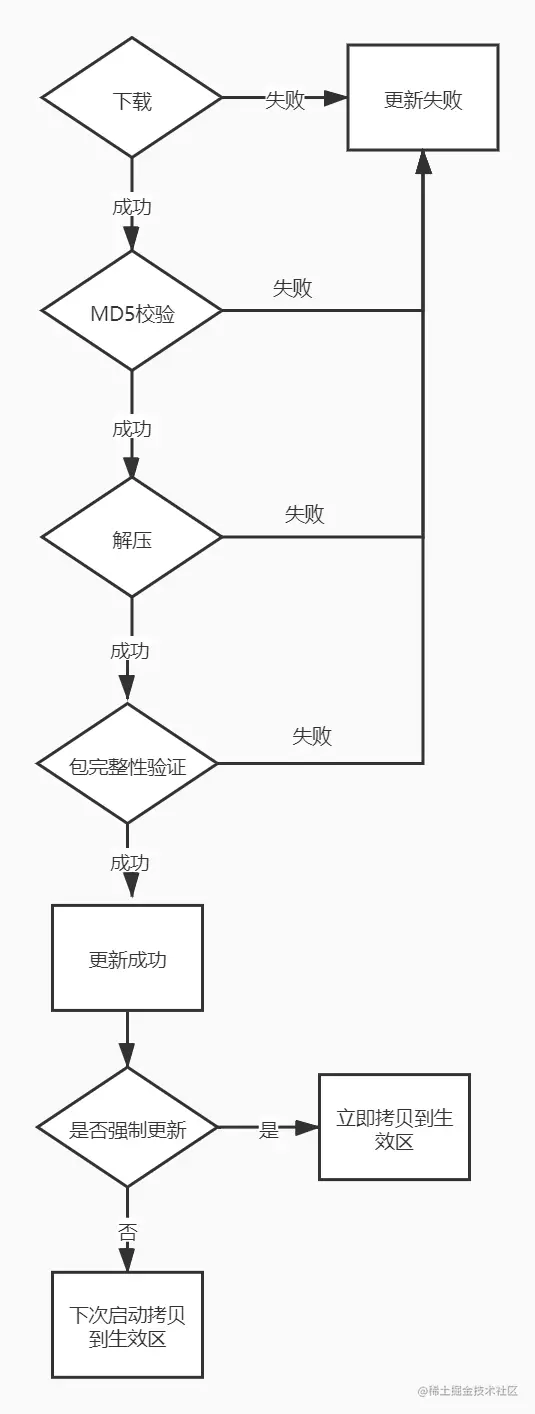
App端在特定的时机进行离线包更新，目前采用两个时机，一是App启动时，一是每次App回到前台时，并且为了避免过度请求，更新接口有30S的间隔时间。为了确保离线包的数据安全，我们将离线包放置在app的沙盘内部，并且**不提供内置离线包**，这是为了防止应用包被破解而泄露业务代码。同时为了确保首次加载的体验，当发现本地离线包为空白时，采用阻塞下载的形式确保离线包完整，再进入主页。

在 App 沙盒内，建立两个目录，一个temp目录，一个 active目录，分别表示离线包**暂存区**，以及 **生效区**。能够被匹配到的只有生效区的离线包，暂存区只做暂时存储，在合适的时机会被移动到 生效区。（类似 git 的管理方式）

每次接口返回的离线包列表，我们会按照下面的流程进行更新替换：



上图中有一个多线程执行，每个线程的逻辑如下：



环节解释：



通过以上流程，保证在宿主app上打开web应用时，都能使用业务方发布的最新代码。

##### 支持web静态资源匹配

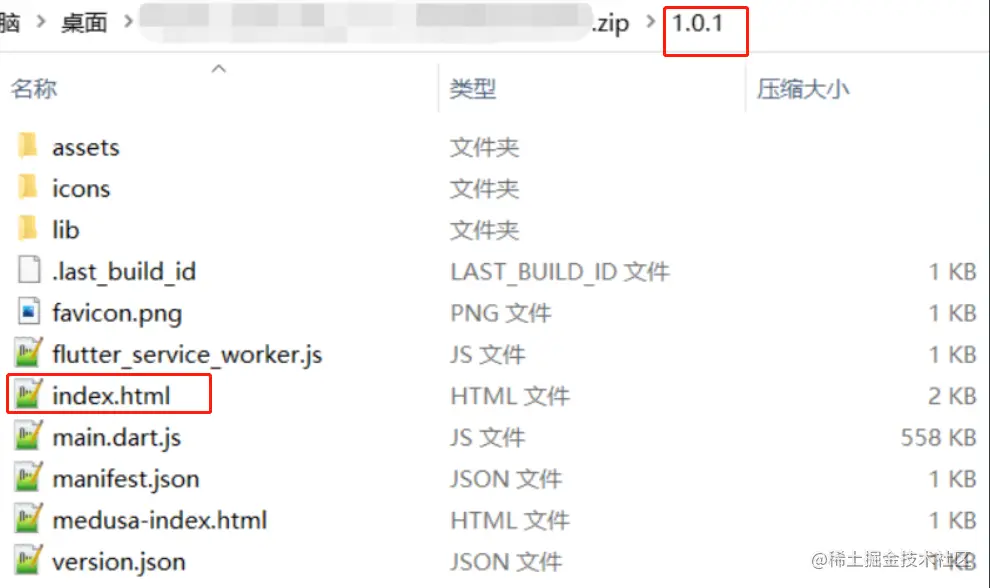
上文提到，要加快web页面的打开速度，其中一个重要环节是 **静态资源离线化**。

资源的匹配，本质上是**网络**资源路径和**本地**资源路径的**映射**关系。基本的匹配规则如下：



##### [www.baidu.com/demo/1.0.1/…](https://link.juejin.cn/?target=http://www.baidu.com/demo/1.0.1/index.html" \o "http://www.baidu.com/demo/1.0.1/index.html" \t "/Users/zpw/Documents\\x/_blank) 是一个完整的网络资源路径，

##### 它指向的就是离线包 demo 内的 /1.0.1/index.html文件。



但是其中有个**特例**，我们去访问一个页面入口html的时候，写法往往是形如 [www.baidu.com/](https://link.juejin.cn?target=http://www.baidu.com/" \o "http://www.baidu.com/" \t "/Users/zpw/Documents\\x/_blank) ，会省略掉资源的路径。此时就只能建立特殊规则，当这个请求：[www.baidu.com](https://link.juejin.cn?target=http://www.baidu.com" \o "http://www.baidu.com" \t "/Users/zpw/Documents\\x/_blank) 到来的时候，自动去离线包内查找 一个固定文件名的html，也就相当于内置了一套映射关系 [www.baidu.com](https://link.juejin.cn?target=http://www.baidu.com" \o "http://www.baidu.com" \t "/Users/zpw/Documents\\x/_blank) => **index.html**. 由此来补全规则漏洞。

### 让web应用使用原生能力

纯Web应用与纯原生应用之间的差距，除了启动速度之外，就是运行中的体验，差距明显。为了抹平这一差距，我们提供丰富的原生能力让H5去调用，使得app使用体验最大程度接近纯原生应用。

WebView的多媒体功能，比如视频播放，web前端的体验是完全和原生无法比拟的。像类似这种功能，使用 我们 App 的提供的原生功能去实现，要比纯前端播放视频的开发要简单，兼容新更强，并且前端的工作量更低，甚至可以轻易做到前端无法做到的效果，比如 京东 App 上能看到的 **可拖动视频悬浮窗**，而web开发处理视频层级覆盖很麻烦。

类似的功能还有，**音乐播放，视频剪辑，图片截取**等。诸如此类纯web不好做的功能，都能通过原生能力来弥补，然后由 web和原生之间的通信来进行数据传递。

原生能力主要分为以下几个方面：

app/页面生命周期监测

如：app进入前后台，页面进入前后台

页面路由跳转

如：用原生的方式跳转新页面

页面样式定制

如：设置导航栏，状态栏样式

原生系统功能

如：相机，相册，指纹（人脸）识别，获取当前位置，经纬度

原生组件扩展

如：播放视频，地图选点，预览网络文件等。

必须说明的是，要扩展原生能力，或者修复原生能力的bug，我们都需要对Web容器进行版本更新，所以接入方有少量的维护成本。

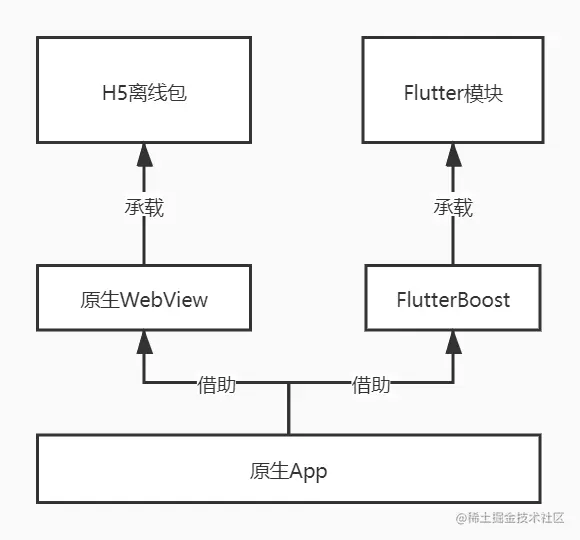
## 其他特性

此方案的核心目的是优化web应用的体验，除了以上3个核心特性之外，还有一些额外特性，给与更多拓展可能。

### flutter-Native混合栈管理

引入混合栈框架[flutterBoost](https://link.juejin.cn?target=https://github.com/alibaba/flutter_boost" \o "https://github.com/alibaba/flutter_boost" \t "/Users/zpw/Documents\\x/_blank)，原本是为了让Android/iOS共用一套原生业务代码，避免两端分别开发同步的麻烦。而在此本SDK中则意义发生改变，在某些极端场景下，比如 某些功能，用H5很难完美实现，但是有现成的flutter解决方案时，我们可以利用flutter代码来承载这部分业务，并且用flutterBoost进行混合栈的管理以及数据传递。

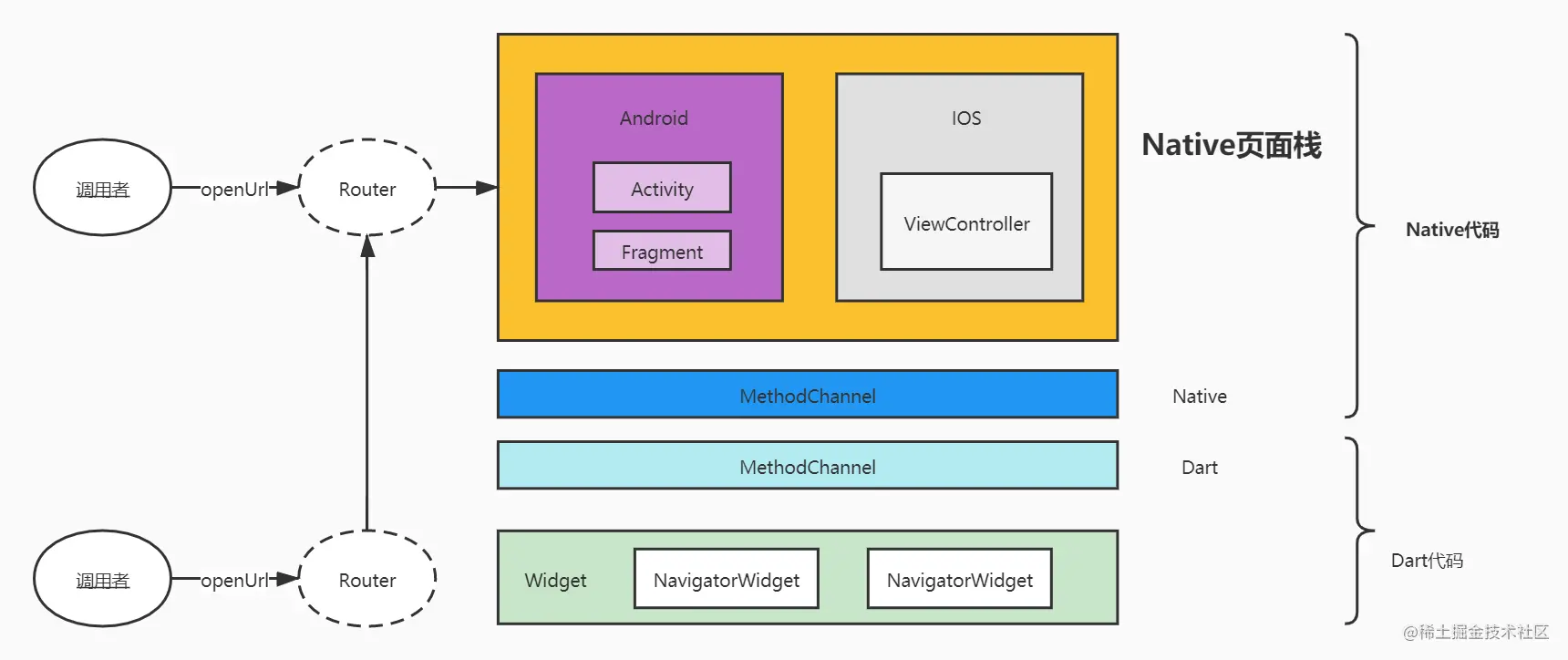
如果完整来看待一个完整的SDK项目，它的结构如下：



flutterBoost的核心作用：

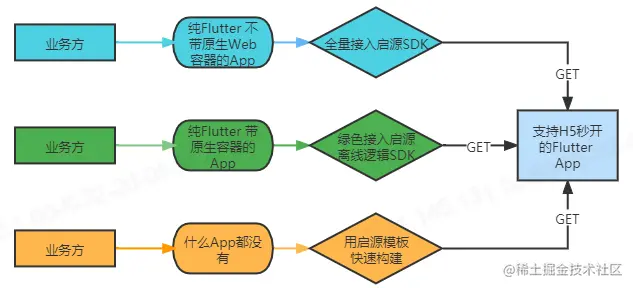
1. 统一用 Native 页面栈管理所有的页面，**native** 和 **dart** 代码都可以打开 **native**页面 或者 **flutter widget**
2. 建立 MethodChannel 通道，让 flutter 和 native 通信，用于打开页面时传递参数等

它的架构图如下：



多种接入方式

我们提供三种接入方式，由各业务方根据需求自主选择：



针对一个纯flutter的app，可采用全量接入的方式，通过flutterBoost打开一个秒开H5的web容器，使得H5上的业务功能拥有最佳体验。并且原来的部分flutter业务，也可以逐步转化成H5，用纯原生web容器去承载。

踩坑提醒：如果原flutter app本身就有flutter webView承载的落地页，那么强烈**不建议**接入本SDK，因为flutter webView与flutterBoost之间存在不可调和的bug，官方没有完全解决。

如果是一个原生app，或者是一个flutter项目，但是用到了原生容器展示H5，可以以最小的成本，仅接入本SDK的离线包**协同逻辑**SDK，这样也可以得到一个支持H5秒开的app

实用性最强的接入方式可能就是第三种，**快速构建**，利用本SDK的快速构建模板，直接构建出一个flutter app，接入方只需要在托管平台更新自己的H5离线包，即可完成Web应用的发布更新。

## 实践中遇到的一些坑

关于vConsole，这个东西有时候显示不出完整的调试信息，有时候遇到一些譬如跨域的问题，有可能被业务方纠结甩锅说是webView容器的问题，每次都让我们给出证据，事实证明容器不可能有这种限制。为了完整显示出页面调试信息，首先webView的配置里面必须 setWebContentsDebuggingEnabled为true，然后用 chrome://inspect 的方式进行调试。

离线包的指标具象化。我们开发的是一个方案的SDK，比如说，某个应用接入了我们的SDK，他使用了哪些离线包，版本号等指标上报到bugly其他埋点平台，在 新的离线包发布之后，可以具体感知到离线包已生效的有多少，如果发现发布了新离线包，但是更新率在活跃用户中很低，那就必然是更新机制某部分出现问题。

最好加上 网络监控机制，优化无网/微网体验，无网-有网的变化要自动刷新页面，有网-无网要通知用户，播放视频时，如果当前是wifi就直接播放，是4G就通知用户是否要在流量下播放。

白屏监测，利用像素点的监测，在web页面加载完成之后，如果判定一定程度都是白点，比如98%，就判定是白屏，实施刷新策略。

## 未来展望

目前我们实现了web应用贴近原生的体验，但是仍有提升空间。

极致体验 虽然已经使用部分原生功能优化了前端页面的体验，并且后续会扩展更多原生功能。但是这都是独立于webView之外的做法。针对webView本身的优化，在原理上是可以干涉页面的渲染过程，使用同层渲染的方式，既不影响当前页面的渲染层级，又能用native组件替换H5元素。

容灾能力 整个方案的核心，就是离线包的托管，同步，匹配，展示。那么，无论是托管在服务器的离线包，还是保存在 App 沙盒内的离线包由于工作失误或者外来入侵导致离线包全部丢失，或者部分丢失。我们都需要做出预案防止此类问题发生，即将出现问题的告警，以及万一出现问题之后的弥补。 以及WebView本身的bug，或者加载页面时发生意外问题，都需要容灾机制去避免恶化。

数据安全 虽然前端代码会通过混淆的方式打包到zip中，然后存储在托管平台上，但是混淆毕竟不是加密，还是存在敏感数据被劫持的风险，在前端展示和管理后台同步离线包的过程中，还是需要对信息安全进行防范。

快速构建生态完善

现在虽然可以通过快速构建的方式直接得到 H5秒开的web app，但是仍然需要我们手动操作，而且缺少app的自动更新机制。下一步可以继续完善快速构建app的生态，让容器可以自动迭代，比如当某个业务方要需要用flutter插件的方式引入某些功能时，就需要对容器进行发版。我们的目标是让接入方脱手，只需要更新离线包就行，其他问题，全部交给本方案。

一个方案从诞生到完善，能做的还有很多，万里长征，始于足下。