跨平台解决方案的技术分析

程序员成长指北 2025年01月04日 21:38 北京

以下文章来源于ELab团队 ， 作者ELab.yangjialong

ELab团队
分享技术新见解

程序员成长指北
专注 Node.js 技术栈分享，从 前端 到 Node.js 再到 后端数据库，祝您成为优秀的高...
104篇原创内容

公众号

点击上方 程序员成长指北，关注公众号
回复1，加入高级Node交流群

背景

近 20 年是中国互联网蓬勃发展的时代，以 2010 年为界限，前 10 年是 PC 互联网时代，PC 互联网时代培养了国民上网冲浪的用户习惯，为后 10 多年的以智能手机为终端的移动互联网时代带来丰厚的人口红利，而在移动互联网时代，以智能手机为依托的软硬件也就成为各大互联网公司争夺流量的焦点战场。

移动互联网发展早期，或是依托母公司的强大技术实力支撑，或是抓住时代发展机遇，或是努力创新求变，一批又一批现象级的移动应用成为广大用户的生活工作等方面难以分割的一部分，类似微信之于社交通讯、支付宝之于支付理财、头条之于资讯创作、美团之于生活服务等等不一而足。移动互联网发展至今，早已是一片红海市场，早期发展起来的 APP 要么成为国民级别的超级 APP，要么在垂直领域深耕难以撼动。

那么，行业龙头型 APP 如何持续拓展服务边界，快速响应市场需求变化以保持竞争优势，后进的 APP 如何通过产品、商业模式创新，迅速切入市场，提高研发的灵活机动性同时不降低产品的用户体验。

针对当前移动互联网的发展现状，跨平台开发的概念和解决方案应运而生。跨平台开发的诞生使命就是围绕着研发效能和用户体验两个主题去打造的，但是就如同一个符合特定场景和高效算法在时间和空间上的 trade-off，跨平台解决方案的不同实现在研发效能和用户体验上同样面临权衡取舍。

本文旨在介绍不同跨平台解决方案的技术架构和特点，分析各个解决方案的优势和不足之处，以便对业界当前的跨平台技术方案建立起整体的认知和对团队的技术选型提供一定的参考作用。注意的是，这里的跨平台特指的是针对 iOS 和安卓进行的跨平台开发。

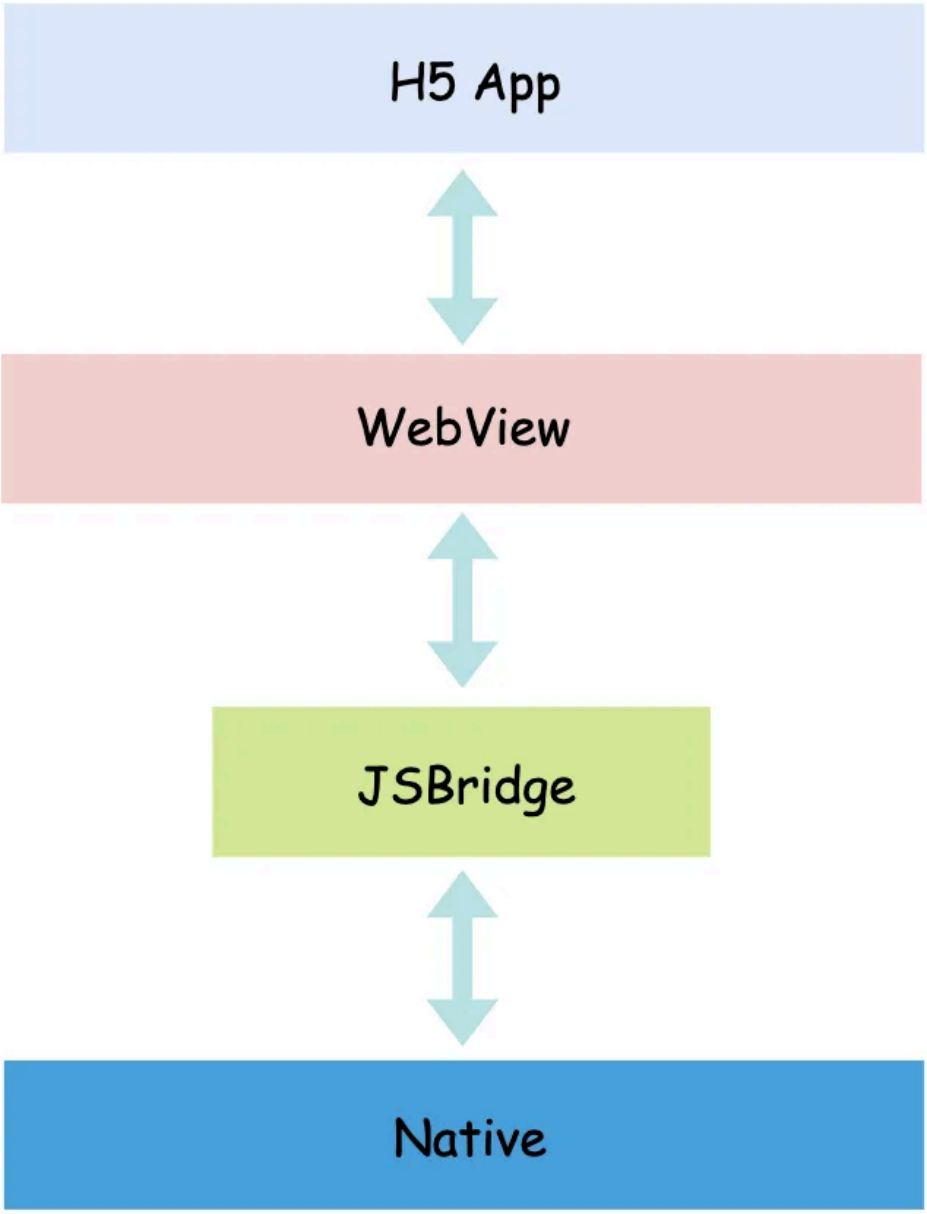
跨平台解决方案

根据采用的渲染技术不同，跨平台解决方案可分为以下三类：

- Web 渲染方案
- 原生渲染方案
- 自建渲染引擎渲染方案

Web 渲染方案

Web 渲染方案主要是使用原生 WebView 控件渲染 HTML 页面，并在原生应用中定义可供 H5 页面访问原生部分能力的接口 JSBridge，从而实现 H5 和 Native 双向通信，也使得 H5 的能力向端侧进一步扩展。



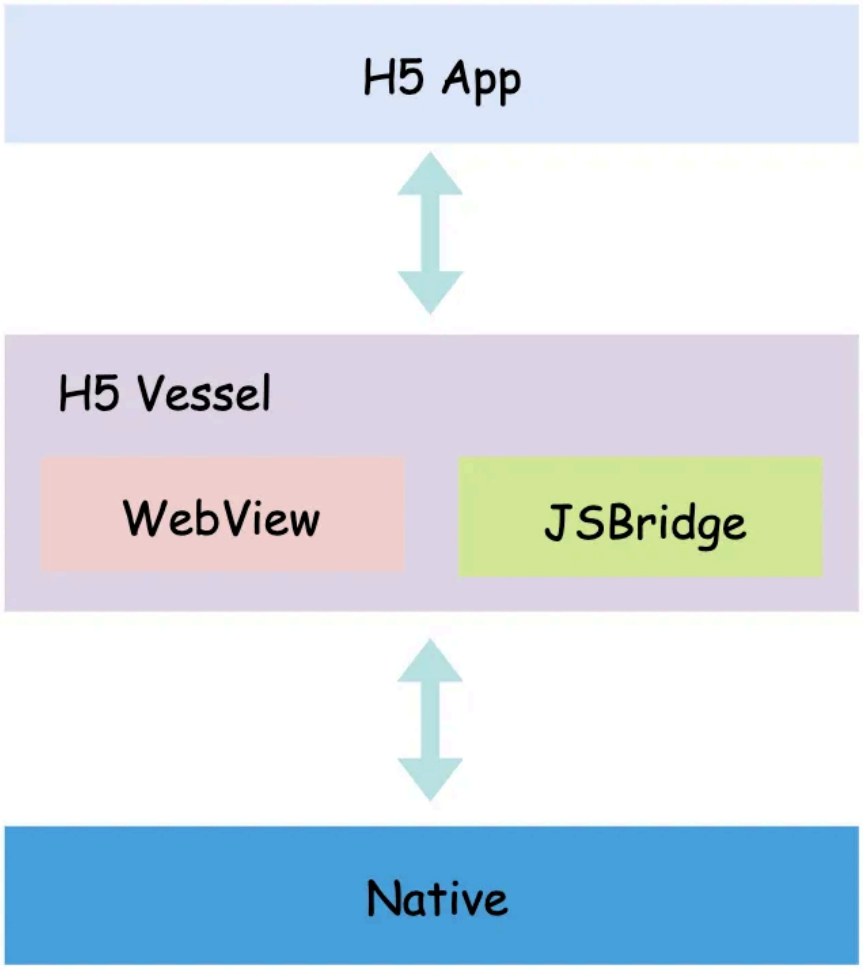
Web 渲染方案本质上是依托原生应用的内嵌浏览器控件 WebView 去渲染 H5 页面，因此 h5 App 的渲染流水线和 Web 页面渲染相一致，能力也局限在 WebView 这一沙箱。下图描述从 WebView 初始化到 H5 页面最终渲染的全过程。

从上图上看，Web 渲染方案的性能瓶颈和 Web 页面开发中遇到的类似，即首屏渲染优化问题，同时多出了一个 WebView 初始化的特有问题。

对于 WebView 初始化所带来的性能开销，不少公司针对自身的 APP 进行内核的定制化改造，诸如腾讯的 X5 内核以及阿里 UC 技术团队的 UCWebView 等。

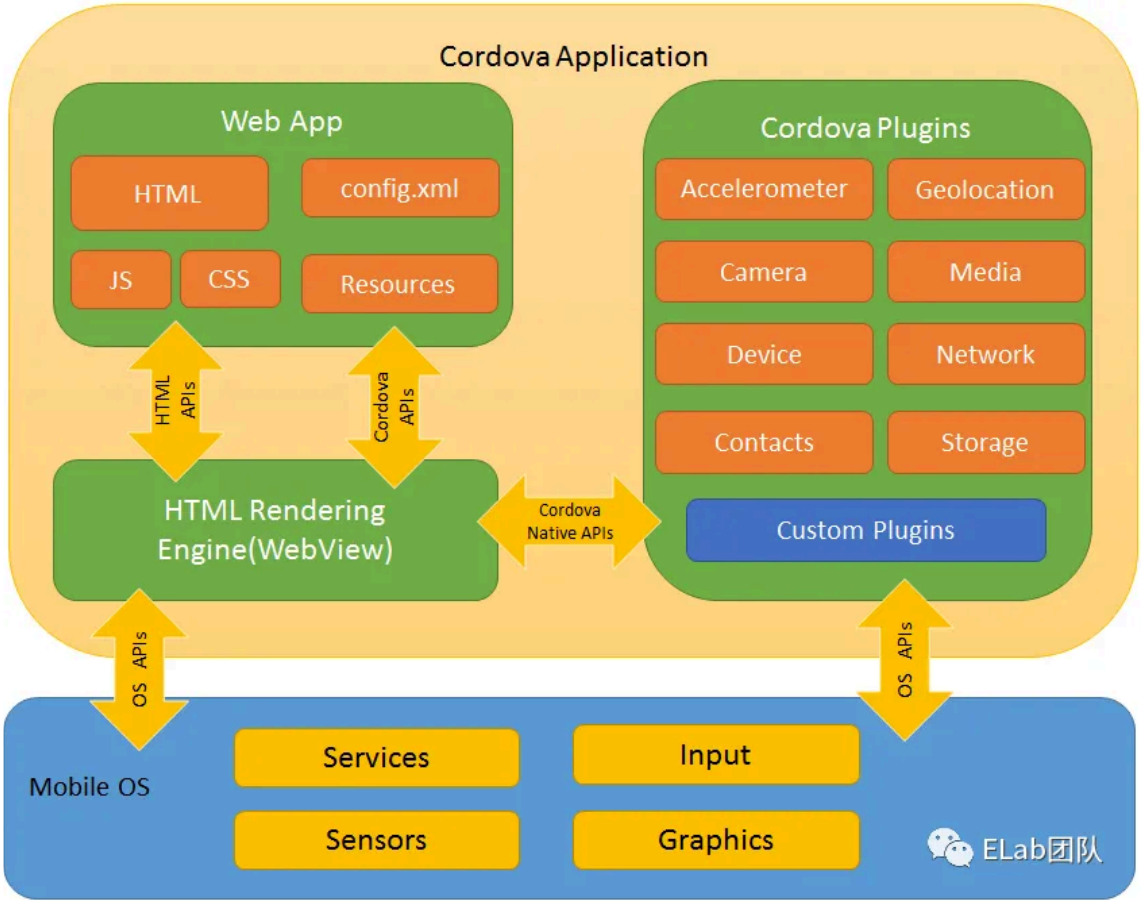
针对资源加载所带来的白屏问题，业界又提出了离线包的优化方案。所谓离线包机制，大体思路就是将原有从线上加载 H5 应用，提前下发到本地，通过 FileIO 或是内存等方式直接进行页面渲染，达到接近原生的用户体验。

上面所描述的是最为原始的 Web 渲染方案，在这基础上业内又提出 h5 容器的技术解决方案，h5 容器提供丰富的内置 JSAPI，增强版的 WebView 控件以及插件机制等能力，对原始版本的方案做了进一步功能高内聚和模块低耦合。



ELab团队

下面以 Cordova 为例，概述一下 H5 容器的大致架构，Cordova 是 Apache 一个开源的移动开发框架，这一框架的核心实现原理就是基于 Web 渲染技术。



图片来源：Cordova 官网

Cordova 应用程序由几部分组成：

- Web App

应用程序代码的实现地方，采用的是 Web 技术，应用运行在原生控件 WebView 中

- HTML Rendering Engine

应用的渲染引擎，即 WebView，该渲染引擎是页面和 Native 实现双向通信的桥梁

- Cordova 插件

提供了 Cordova 和原生组件相互通信的接口并绑定到了标准的设备API上。这使你能够通过 JavaScript 调用原生代码，这些核心插件包括的应用程序访问设备功能，比如：电源，相机，联系人等。

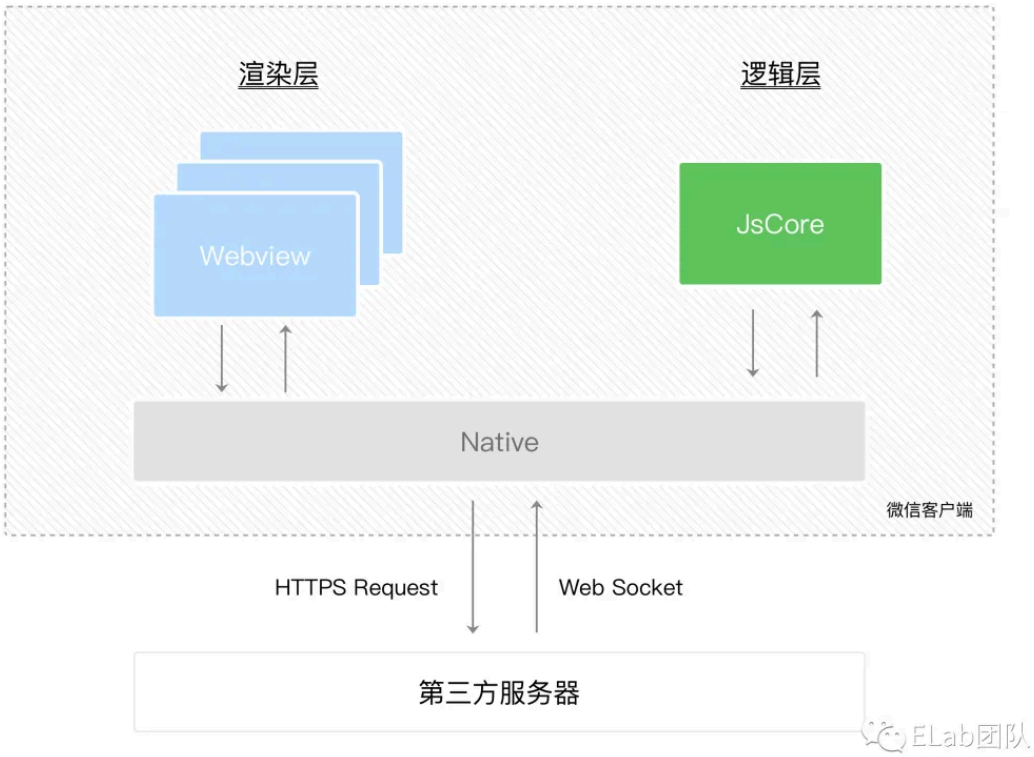
- Mobile OS

原生系统层，提供系统能力

小程序

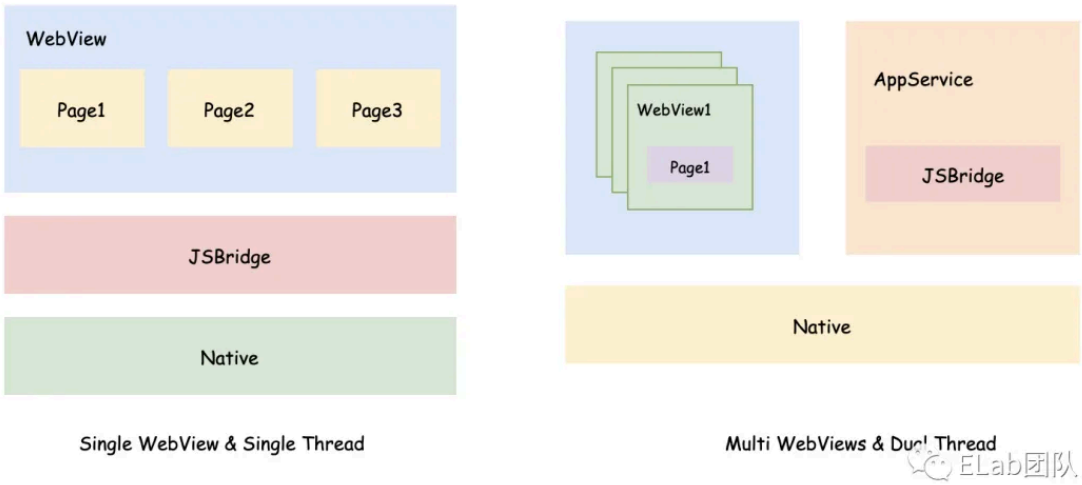
小程序是微信在 2017 年提出一项创新性的轻应用，不需要下载安装即可使用。记得 Brendan Eich 曾对 JavaScript 的评价它的原创之处并不优秀，它的优秀之处并非原创。我想微信小程序也是这样，因为小程序采用的技术手段仍脱离不了 Web 渲染方案，即采用 WebView 作为渲染引擎、JSBridge 的封装和离线包机制等，但是其最大创新之处在于将渲染层和逻辑层进行了分

离，提供一个干净纯粹的 JavaScript 运行时，多 WebView 的架构使得用户体验进一步逼近原生体验。



图片来源：微信小程序官网

具体来看，小程序的渲染层和逻辑层分别由两个线程管理，渲染层采用 WebView 进行页面渲染（iOS 使用 UIWebView/WKWebView，Android 使用 WebView），小程序的多页面也由多 WebView 接管。逻辑层从 WebView 分离，使用 JavaScript 引擎（iOS 使用 JavaScriptCore，Android 使用 V8）单独开启一个 Worker 线程去执行 JavaScript 代码。逻辑层和渲染层之间的通信经由 Native 层中转，网络 IO 也通过 Native 层进行转发。



和之前的 Web 渲染技术相比较来看，小程序采用多 WebView + 双线程模型的架构。由多 WebView 构成的视图层为页面性能赋予更加接近原生的用户体验，单个 WebView 承载更加轻量的页面渲染任务，JavaScript 脚本单独抽离在 Worker 线程限制了开发者直接操作页面的能力，进一步约束在微信小程序的规范下，这也是小程序无法直接操作 DOM 的缘由。

这里多提一点的是，小程序的组件分为原生组件和非原生组件，对于原生组件而言，这就脱离的 Web 渲染方案的范畴，属于原生渲染方案的一部分，所以从这点上看，小程序也可以算得上是 Web 渲染和原生渲染的融合解决方案。综上来看，Web 渲染跨平台方案经历了三个阶段性的发展，从原始时期的 h5 + JSBridge + WebView，到 h5 容器的抽象提升，再到目前如火如荼的小程序。不难看出，Web 渲染方案的有如下特点：

- 开发效率高

采用 Web 技术，技术门槛相对较低，技术人员积累丰厚，社区资源丰富，对前端友好，一次开发，多端运行

- 动态化好

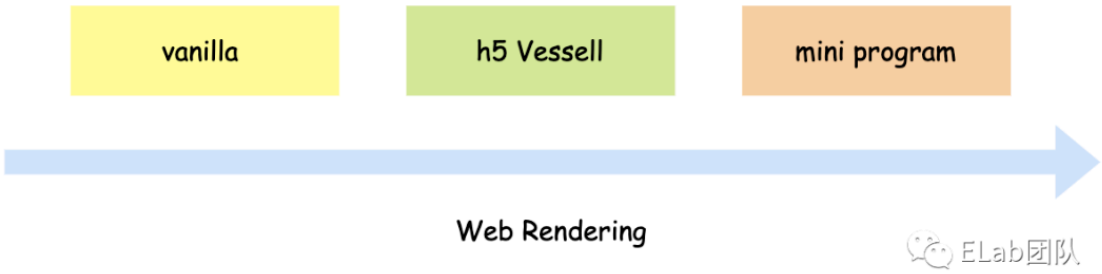
Web 技术的天然动态特性支持，无需发版

- 表现一致性佳

Web 页面除了个别元素和属性的差异、多屏适配外，其双端表现相对一致

- 性能较差

页面采用 WebView 渲染，页面加载耗时长，功能受限于沙箱，能力有限，难以承接复杂交互或是需要高性能的任务，整体用户体验差



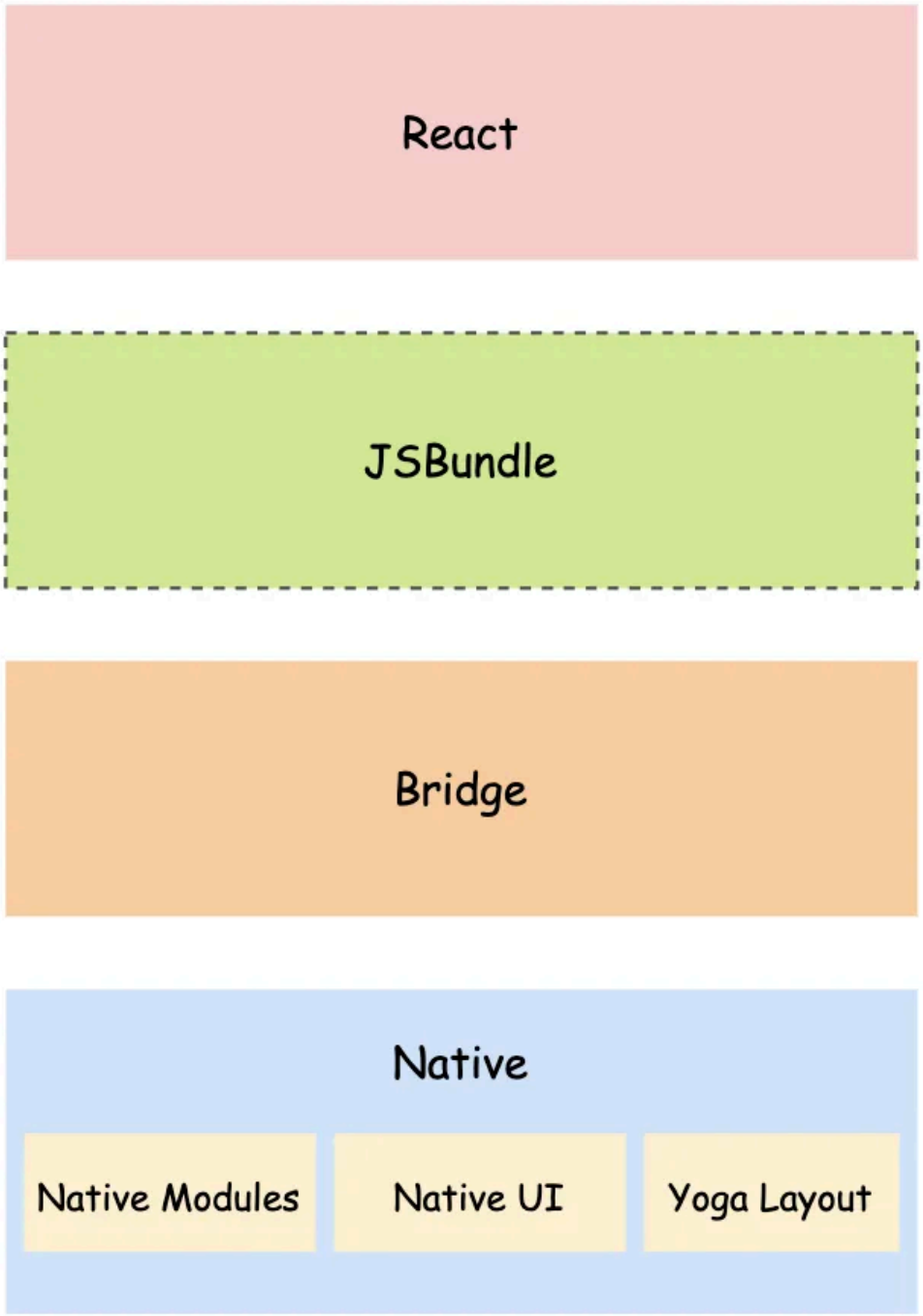
原生渲染方案

Web 渲染方案的致命弱点在于无法出色地完成高性能和体验的目标，但是其良好的社区生态、跨平台一致性和高研发效率都是其无法忽视的优势，那么如何做到二者的平衡，答案就是原生渲染方案。

原生渲染方案的基本思路是在 UI 层采用前端框架，然后通过 JavaScript 引擎解析 JS 代码，JS 代码通过 Bridge 层调用原生组件和能力，代表的框架是 React Native 和 Weex。

从原生渲染方案的实现思路不难看出，顶层采用类 Web 框架用于降低开发成本和统一技术栈，UI 的渲染通过 JSBridge 由原生控件直接接管，从而获得性能和体验的提升。

下面以 React Native 为例，具体展开讲解一下原生渲染方案，React Native 的整体架构图如下：



ELab团队

- React 层

最顶层是 React 层，利用 React 框架进行 UI 的数据描述，开发者使用 Class Component 或 Functional Component 进行页面开发，框架内部将会把页面描述转化为 ReactElement 这一代

表的虚拟 DOM 的数据结构，用于运行时的 Diff 对比和消息收发等

- [JS Bundle 中间产物]

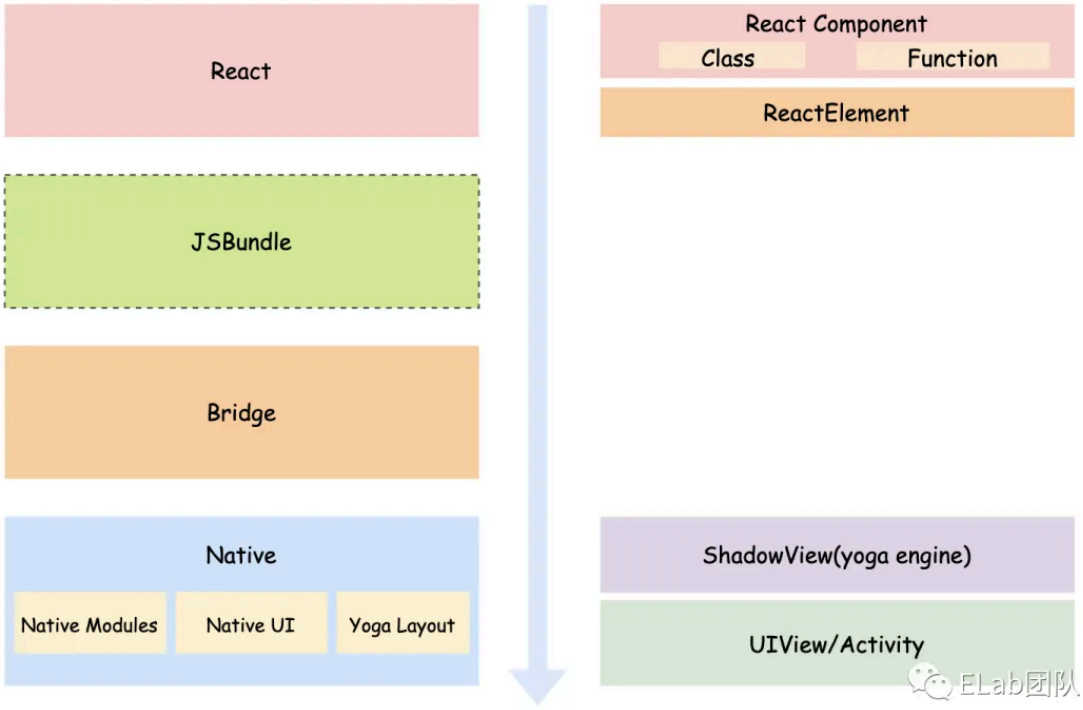
React Native 通过 metro 打包功能直接将整个 RN 应用打包为一个 JSBundle，通过 Bridge 层在 RN 应用初始化时加载整个 JS 包进来

- Bridge 层

Bridge 是连接 React 和 Native 的中间层，React 层的 UI 需要通过 Bridge 层的 UIManager 接口实现原生控件的创建和更新，通过 NativeModules 接口实现原生能力的调用

- Native 层

在 Native 层中，Native Modules 实现了与上层交互的原生能力接口，Native UI 实现终端实际的控件展示，Yoga 跨平台布局引擎实现了基于 Flexbox 布局系统的 JS 和 Native 的镜像映射关系。值得注意的是，整个 RN 架构中，存在以下 UI 视图数据结构：



下面从线程模型角度，分析一下 RN 的运行机制：

- UI 线程

应用的主线程，用于处理原生控件的绘制

- JS 线程

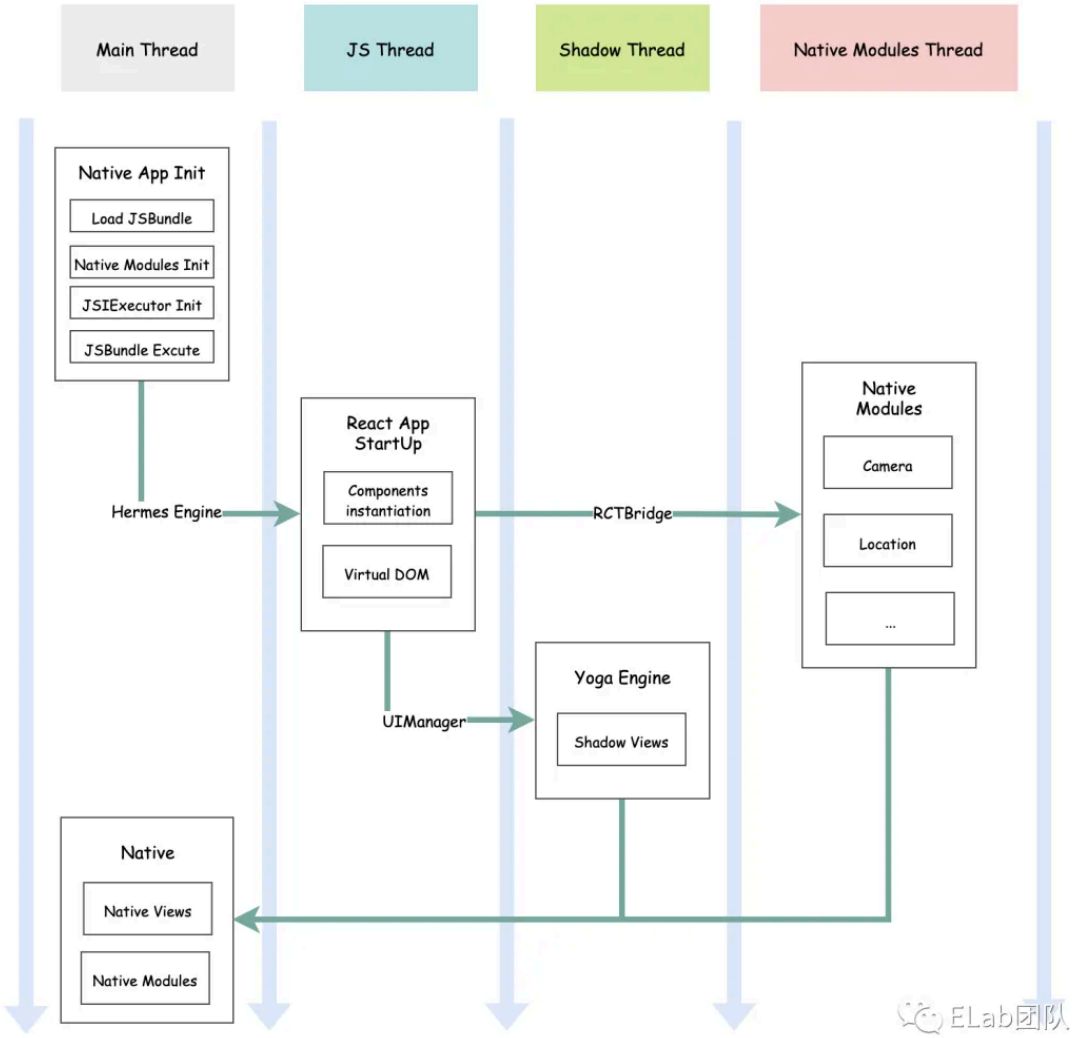
React 构成的 JS 代码运行在此线程

- Shadow 线程

主要用于构建 JS 与原生控件的布局镜像数据

- Native Modules 线程

提供原生能力，这里采用的是多线程模型，iOS 端通过 GCD 实现，Android 端通过 AsyncTask 实现



RN 应用在 UI 线程进行初始化，初始化的内容包括加载 JSBundle、初始化 Native Modules 等原生能力模块、创建 JSC/Hermes JavaScript 引擎，执行 JS 代码。

创建的 JS 引擎独立在一个 JS 线程，解释执行 React 代码，并将生成的布局或逻辑信息序列化后经由 Bridge 发送给 Native。

React 代码中视图层的渲染通过 UIManager 调 createView/updateView 等方法，基于 Yoga 布局引擎创建对应的 shadowView；逻辑层中涉及原生能力调用的部分通过 RCTBridge 对象转发到相应的原生接口。Native 接收到 Bridge 层的消息，进行视图的更新或是功能处理。

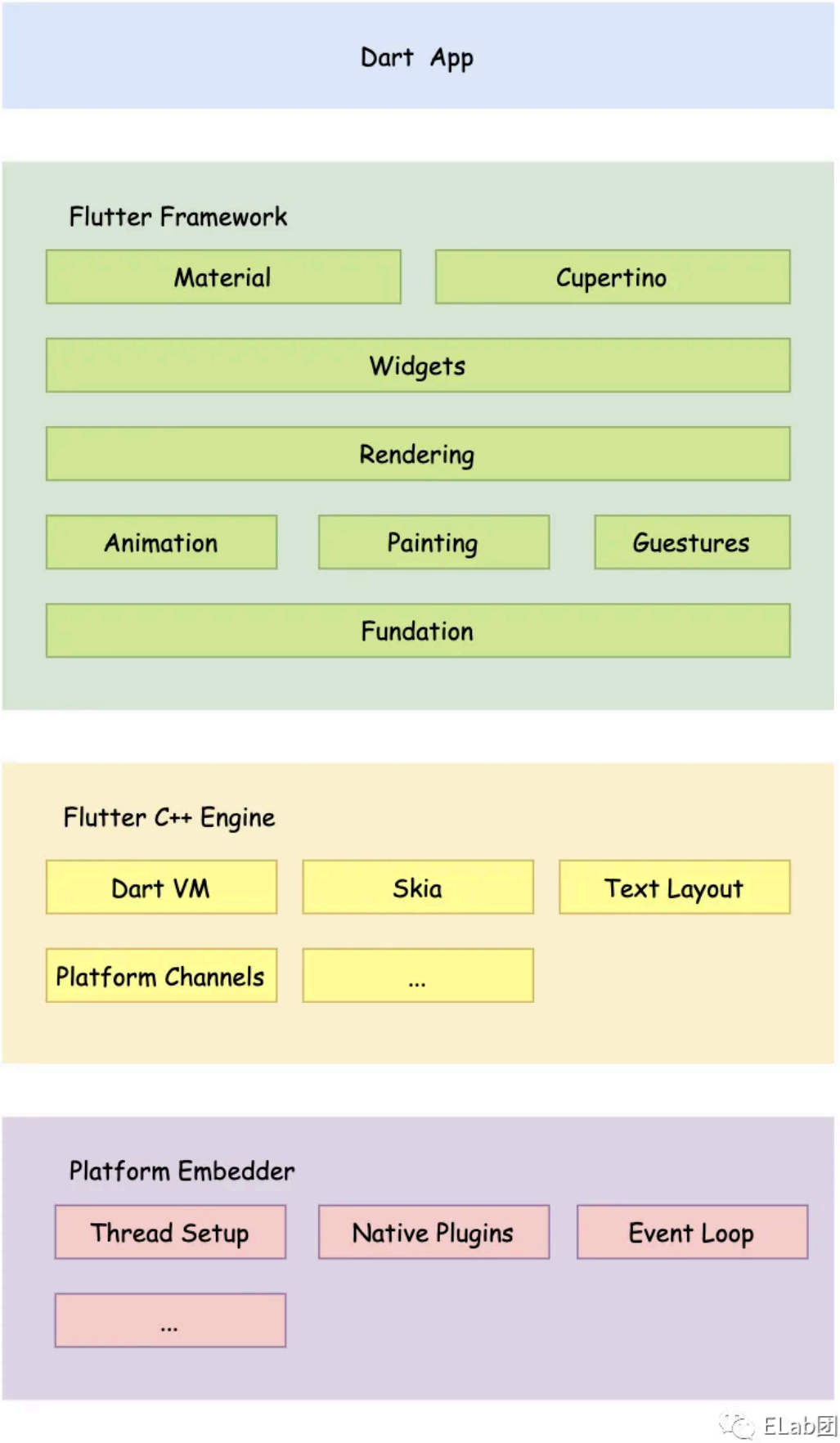
原生渲染方案通过直接接管渲染层的方案，弥补了 Web 渲染方法在性能和体验上的不足，同时在顶层采用类 Web 的语法集，将开发技术边界延展至 Web 领域，同时可以很好的复用当前前端主流 UI 框架 React/Vue 的繁荣生态系统。

虽然原生渲染方案有上述的优势，但是有一个致命的弱点就是 Native 层和 JS 层的通信所带来的性能瓶颈。一方面页面的更新和事件的响应经由 Native 触达 JS 层，再由 JS 层返回给 Native 层需要来回的时间成本，另一方面数据的交互需要频繁进行序列化和反序列化的转换。因此，

在一些 UI 线程和 JS 线程存在持续频繁交互的场景（动画、滚动）等，RN 表现就不尽如人意。

自建渲染引擎渲染方案

自建渲染引擎渲染方案，是有别于 Web 渲染采用 WebView 容器进行渲染 UI、原生渲染通过 Bridge 方式转化为原生控件渲染 UI 等方案，另辟蹊径通过自建渲染引擎方式，直接从底层渲染上实现 UI 的绘制，而 Flutter 就是跨平台、自渲染的代表。Flutter 的架构设计如下所示：



ELab团队

整体来看，Flutter 应用可以分为四层：

- Dart App 层

最顶层是 Dart App 层，以 Widget 为基本视图描述单元，构建起 UI 体系

- Flutter Framework 层

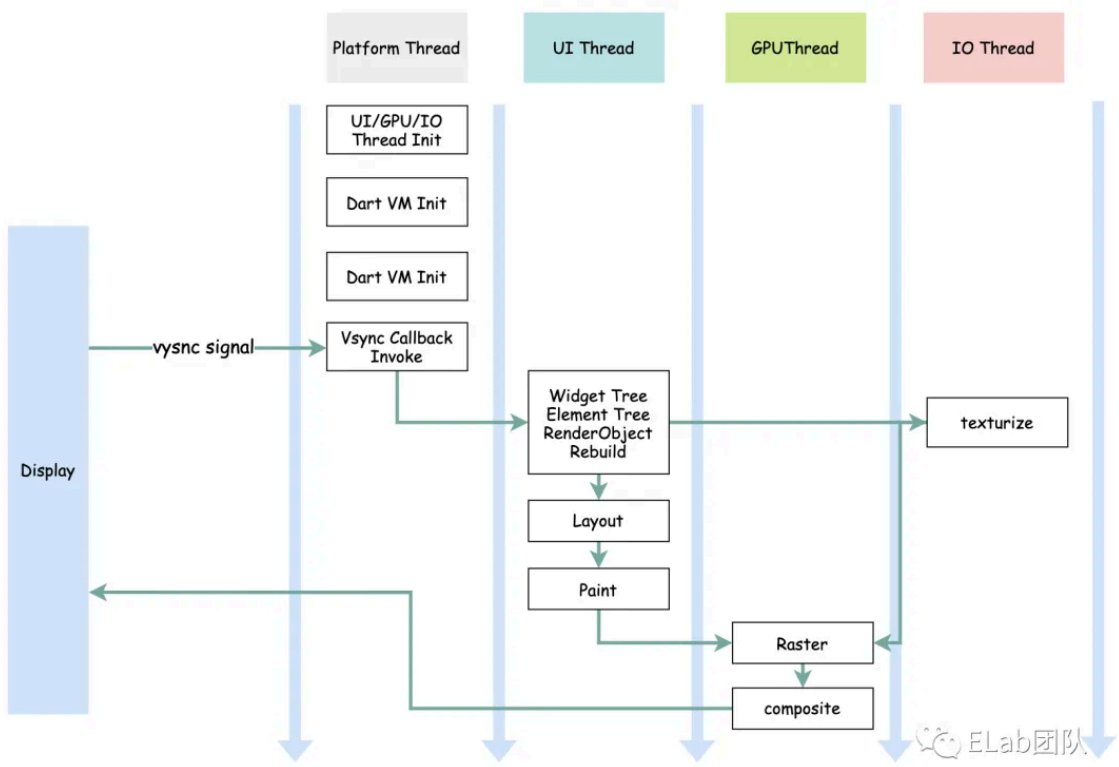
内置基础的 Flutter 组件，并根据不同平台的视觉风格体系，封装 Material 和 Cupertino 两套 UI 库供上层使用

- Flutter Engine 层

Flutter 框架的核心所在，包括 Dart 虚拟机、Skia 跨平台渲染引擎、文字排版、平台通道等，通过 Engine 层，建立起 Dart App 层和原生平台之间联系，从而实现二者的双向通信

- Embedder 层

平台嵌入层为 Flutter App 提供宿主环境、线程创建以及基于插件机制的原生能力扩展等 Flutter 在打包的时候，将 Dart 业务代码和 Flutter Engine 代码基于 iOS/Android 不同平台分别进行打包。Native 在启动时会通过调用 C++ 的各自实现（Java 通过 JNI，OC 天然支持）初始化 Flutter Engine 层提供的接口，创建 UI/GPU/IO 三个线程和实例化 Dart VM。Dart 业务代码在 Release 模式下采用 AOT 的方式进行编译，并运行在 Dart VM 中。下面从线程模型机制，分析一下 Flutter App 的运行机制：



- Platform 线程

Flutter 的主线程，由 Native 创建。负责平台 vsync 信号的回调注册，即当接收到从显示设备的 vsync 信号后，Platform 线程驱动 UI 线程的执行

- UI 线程

负责响应 vsync 信号，执行 Dart 层代码，驱动渲染管线的运行，将 Widget Tree 生成 Layer Tree 并提交给 GPU 线程做进一步处理

- GPU 线程

GPU 线程将 Layer Tree 转化为具体的绘制指令，并调用 skia 跨平台渲染引擎进行光栅化上屏

- IO 线程

主要负责请求图片资源并完成解码，然后将解码的图片生成纹理并传递给 GPU 线程

显示器在一帧 vblank 后，会向 GPU 发送 vsync 信号，Native 的 Plaform 线程接收到 vsync 信号后，执行绘制帧回调方法，即驱动 UI 线程进行 UI 绘制。

UI 线程中，Native 通过调用 C++ 的各自实现，将绘制指令通过 window 对象发送给 Dart 层，Dart 层会重构代表 UI 的数据树（Widget Tree，Element Tree 和 RenderObject Tree）并生成布局信息。根据布局信息生成一系列绘制指令的 Layer Tree，并通过 window 对象传递给 GPU 线程。

- 这里多提一句，Dart 层通过三棵树去描述 UI 的视图结构。
  - Widget Tree 是直接面向开发者的 UI 元素的配置信息，Widget 是 Immutable 的，如果 Widget 的状态发生更新，会发生重建。实际业务场景中，Widget 会频繁触发重建。
  - Element Tree 是 Widget Tree 和 RenderObject Tree 的桥梁，当Widget 发生变化后，会将其 Element 标记为 Dirty Element，在下一次 vsync 信号到来时进行渲染。当 Widget 挂载到 Widget Tree 时，会调用 widget.createElement 方法，创建其对应的 Element，Flutter 再讲这个 Element 挂载到 Element Tree 并持有有创建它的 Widget 的引用
  - RenderObject Tree 是真正执行组件布局渲染的工作，通过 RenderObjectToWidgetAdapter 这个 RenderObjectWidget 建立起 Widget 、Element 和 RenderObject 三者之间的联系

GPU 线程从 UI 线程获取 Layer Tree 构成的绘制指令，通过 Skia 这一跨平台渲染引擎进行光栅化，绘制成帧数据，将帧数据放在帧缓冲区，然后等待显示器上屏。综上来看，以 Flutter 为代表的的自建渲染引擎方案的优势在于：

- UI 控件是直接采用 Skia 这一跨平台渲染引擎进行绘制

顶层使用 Dart 的语法进行 UI 的配置信息描述，并通过 Diff 算法优化渲染流程，生成 Layer Tree 后，再调用 C++ 的代码将布局信息发送给 Flutter Engine，Flutter Engine 直接通过 Skia 将 UI 控件绘制上屏。这里与原生渲染方案最大的不同点在于，Native 应用仅作为宿主环境，UI 控件不需要转化为原生控件，直接采用渲染引擎进行绘制，从而保证了双端的一致性和良好的性能与体验。

- Dart 在 Release下采用 AOT 的 编译模式

Dart 代码在 Release 采用 AOT 的编译模式转化为二进制代码，从而在 Dart 运行时环境中执行效率更高，性能也更为卓越。对比 React Native 来说，由于打包的是 JSBundle，所以在运行时仍是基于 JavaScript 运行时进行解释执行 JS 代码，因而产生较大的性能瓶颈。

- UI 层与原生层的数据交换性能更高

跨平台技术发展现状与展望

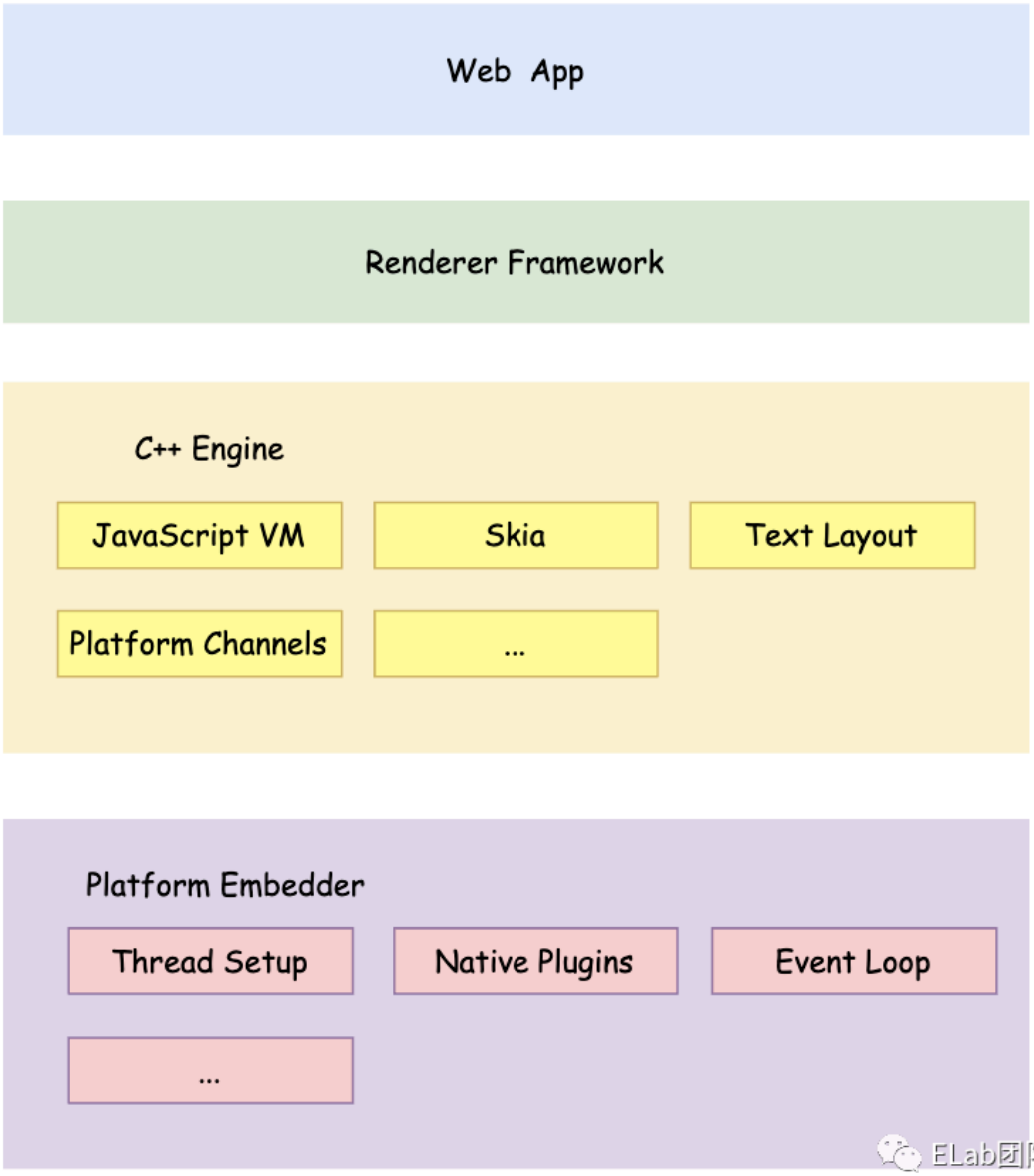
通过上文的讲述，我们对不同跨平台的技术实现方案有了基本了解，落实到实际业务研发层面看，这几种方案目前都是有各自的用武之地。

对于中小型公司而言，内部技术实力不足以支撑多端研发，Web 渲染方案是一种现实的解决措施。对于大公司来说，在 Web 渲染方案上，更是可以通过小程序框架的搭建，从而基于自家 APP 打造周边轻应用的生态闭环，同时在性能和体验方面更进一步。

原生渲染和自建渲染引擎渲染方案对于在性能和体验方面有着更高要求的产品来说，是一个合适的选择，当然自建渲染引擎的性能上限更高更为出色。

通过分析不同的跨平台解决方案，单纯性能和体验上考虑，自建渲染引擎是当前的一个较优解，虽然目前 Flutter 的动态化能力还不算出色，但是其架构思路或许能够启发我们，去设计一套权衡不同维度的新的框架出来，以下是笔者的一种设想：

最顶层是 Web App，采用前端 DSL 开发，Renderer Framework 是将前端的 UI 信息通过 JS 绑定的 C++ 层的接口经由 JS VM 传递给引擎层，引擎层再调用 Skia 进行 UI 的绘制。这样一来，就能实现跨平台研发、多端表现一致、动态化、性能和体验高效的目标。



ELab团队

参考资料

[1] H5 容器简介: *https://tech.antfin.com/docs/2/59192*

[2] 离线包介绍: *https://tech.antfin.com/docs/2/59594*

[3] Hybrid App 离线包方案实践: *https://juejin.cn/post/6844904031773523976*

[4] Cordova 架构: *https://cordova.axuer.com/docs/zh-cn/6.x/guide/overview/*

[5] 小程序架构: *https://developers.weixin.qq.com/miniprogram/dev/framework/quickstart/#小程序简介*

[6] 微信小程序技术原理分析: *https://zhaomenghuan.js.org/blog/wechat-miniprogram-principle-analysis.html*

[7] 小程序同层渲染原理剖析: *https://developers.weixin.qq.com/community/develop/article/doc/000c4e433707c072c1793e56f5c813*

[8] React Native 架构一览: *http://www.ayqy.net/blog/react-native-architecture-overview/*

[9] 「ReactNative 原理」启动流程: *https://juejin.cn/post/6844904184500715527*

[10] React Native 新架构分析: *https://juejin.cn/post/6893032764124168206#heading-9*

[11] [译] Flutter 的编译模式: *https://zhuanlan.zhihu.com/p/61903658*

[12] Flutter 跨平台演进及架构开篇: *http://gityuan.com/flutter/*

[13] 超详解析 Flutter 渲染引擎|业务想创新，不了解底层原理怎么行？: *https://developer.aliyun.com/article/759901?spm=a2c6h.12883283.1362933.27.7bd3201cRSybz5#*

程序员成长指北

专注 Node.js 技术栈分享，从 前端 到 Node.js 再到 后端数据库，祝您成为优秀的高级 N...

104篇原创内容

公众号

Node 社群

我组建了一个氛围特别好的 Node.js 社群，里面有很多 Node.js小伙伴，如果你对Node.js学习感兴趣的话（后续有计划也可以），我们可以一起进行Node.js相关的交流、学习、共建。下方加 考拉 好友回复「Node」即可。

“分享、点赞、在看”支持一波👍