2020 美团技术年货

CODE A BETTER LIFE

【前端篇】

美团 美团

目录

前端

移动端 UI 一致性解决方案

作者：韩洋　彦平　李肖　瀚阳　赵炎

1. 背景

1.1 行业现状与问题

很多技术同学都知道，移动端往往比较侧重业务开发，这会导致人员规模不断扩大，项目复杂度也会持续增长。而为了满足业务的快速上线，很难去落实统一的设计规范，在开发过程中由于 UI 缺乏标准导致的问题不断凸显，具体体现在以下 4 个层面：

- **设计层面**：由于 UI 缺乏标准化设计规范，在不同 App 及不同开发语言平台上设计风格不统一，用户体验不一致；设计资源与代码均缺乏统一管理手段，无法实现积累沉淀，无法适应新业务的开发需求。
- **开发层面**：组件代码实现碎片化，存在多次开发的情况，质量难以保证；各端代码 API 不统一，维护拓展成本较高，变更主题、适配 Dark Mode 等需求难以实现。
- **测试层面**：重复走查，频繁回归，每次发版均需验证组件质量。
- **产品层面**：版本迭代效率低，版本需求吞吐量低，不具备业务的快速拓展能力。

1.2 外卖移动端 UI 一致性情况

近来年，美团外卖业务开始由发展期走入成熟期，这更要求对细分场景的快速迭代。目前，外卖平台承载了餐饮、商超、闪购、跑腿、药品等多个业务品类，用户入口则

覆盖了美团 App 外卖频道、外卖 App、大众点评外卖频道等多个独立应用。由于前期侧重需求的快速上线，设计层面缺乏标准化的规范约束，UI 设计风格不统一，也存在多次开发的情况，目前的维护成本较高，在开发过程中逐渐暴露出一些问题，主要体现在以下三个层面。

指标一：移动端 UI 问题统计

在 Ones（美团内部研发需求管理工具）中，单个版本的 UI 适配问题占比超过总 Bug 数的 11.82%，亟待优化；交互适配问题在绝大多数版本中均有出现，一定程度上反映了其发生的普遍性。

发现版本	V7.15	V7.16	V7.17	V7.18	V7.19	...	平均
UI适配问题占比（%）	15.00%	8.00%	17.83%	18.24%	10.92%	...	11.82%

指标二：需求承接率数据统计

用户侧 UI 需求吞吐率达 18.3%，目前用户侧 UI 需求吞吐率较低，亟待解决。

版本	Q1季度	Q2季度	Q3季度	总计
交互视觉需求	5/26	8/52	4/15	17/93=0.183

指标三：需求入版情况统计

目前各版本 UI 同学都会提出一定数量的视觉优化需求，但实际入版量仅为三分之一左右，未上线的原因均为 RD 开发时间不足。

需求名称	是否入版	当前状态
首页视觉优化	❎	未上线：RD开发时间不足
点菜页视觉优化	❎	未上线：RD开发时间不足
提单页视觉优化	❎	未上线：RD开发时间不足
订单状态页视觉优化	✅	已上线：V7.2提出，V7.10上线
商家列表视觉优化	✅	已上线

从长远角度来看，随着固有业务渗透率的不断饱和，未来一段时间内，美团外卖还有开拓新业务、进入新市场的需求，如国际化 App、闪购 App 等，需要移动端能够高效地组建新业务 App。在此背景下，移动端具备快速调整适应的 UI 展现能力是重中之重。为了达到上述目标，需要 PM/UI/RD 共同维护一套设计规范，在产品上统一风格，在源头上做到统一设计，并在代码中统一进行实现。

1.3 UI 一致性项目

基于上述开发工作中的切实痛点，以及未来可预见的移动端能力需求，迫切需要一套统一的 UI 设计规范，以此沉淀设计风格，建立统一的 UI 设计标准。

UI 一致性项目自 2019 年 5 月份被提出，是外卖 UI 设计团队与研发团队的共建项目，该项目是为了改善用户端体验一致性，提升多技术方案间组件的通用性和复用率，降低整体视觉改版的研发成本。通过抽离成熟的业务场景，建立可提供高质量、可扩展、可统一配置的基于 Android/iOS/MRN 的组件代码库，使之具备支持多业务高层次的代码复用能力，进而提高 UI 业务中台能力，使项目保持高度一致性。

为了帮助团队提升产研效率，外卖技术成立了袋鼠 UI 共建项目组，将门户建设、工具链建设以及组件建设统一管理统一规划，并将工具链的品牌确定为“积木”，此前我们已经写过两篇文章《积木 Sketch Plugin：设计同学的贴心搭档》、《积木 Sketch 插件进阶开发指南》介绍过积木相关的内容，本文主要介绍 UI 一致性。

UI 一致性是绝大部分研发团队面临的共性问题，大家对落地设计规范，提高 UI 中台能力，提升产研效率具有强烈的诉求。通过 UI 一致性的建设，不仅可以在品牌上实现体验升级，更可以全面提高产研效率，为业务的快速迭代提供有力支持和有效保障。统一的品牌符号、品牌特征，有助于加深产品在用户心目中的印象。统一的用户界面和交互形式，能帮助用户加深对产品的熟悉感和信任感。而一个好的设计语言可以在体验上为产品加分，也能够更好的创造一致性体验。

2. UI 一致性整体方案

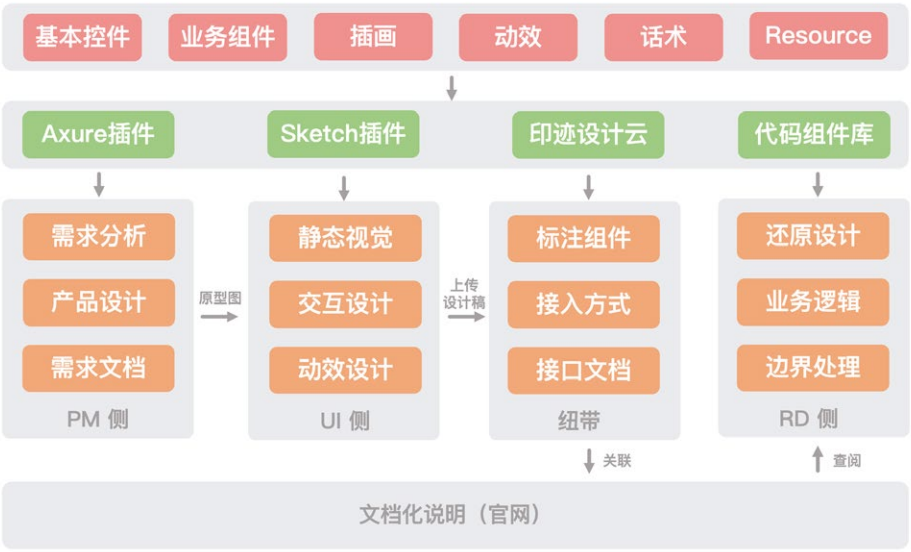
为了帮助更多的业务部门定制符合一致性原则的专属设计风格，外卖技术部在实践中不断总结经验，开发了一套通用的 UI 一致性解决方案。该方案通过 UI 一致性工具链落地项目建设，并打造一整套的闭环 UI 开发流程，目前已经取得了一定的成果，以下系具体方案的介绍。

2.1 方案全景

外卖 UI 一致性套件由积木工具链、代码组件库、定制化设计云协作平台以及文档化说明（官网）四部分组成。

1. **积木工具链**：通过建立包含相同设计元素的统一物料市场，PM 通过 Axure 插件拾取物料市场中的组件产出原型稿；UI/UE 通过 Sketch 插件落地物料市场中的设计规范，产出符合要求的设计稿。未来，希望通过高保真原型输出，可以给中后台项目、非依赖体验项目提供更好的服务体验，赋予产品同学直接向技术侧输出原型稿的能力。
2. **代码组件库（Android、iOS、MRN）**：设计稿中的组件与 RD 代码仓库中组件一一对应。
3. **文档化说明**：官网详细描述了代码组件库的集成方式、组件的使用方法，降低开发上手难度，只需要理解接口和职责即可进行业务开发。
4. **定制化设计云协作平台**：与美团内部的印迹团队（云协作平台）合作开发，在 RD 的设计稿中标注了哪些是代码组件库中已有的元素，避免重复开发，同时

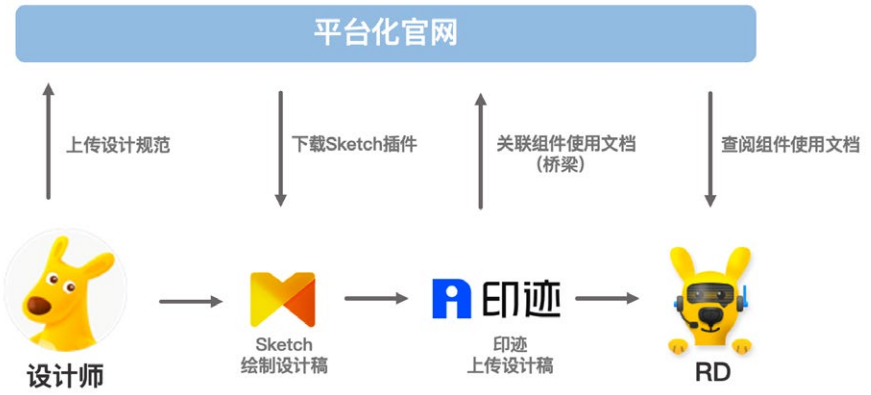
关联了官网中该组件的使用说明，是代码组件库与官网的纽带。



外卖 UI 一致性解决方案

2.2 接入指南

1. 设计师逐步将设计语言沉淀为设计规范（包括组件、颜色、字体、图片等）上传至官网供整个设计团队查阅，同时将其量化并内置于积木 Sketch 插件中；开发同学则将其代码化，针对 Android/iOS/MRN 三端进行组件库开发。
2. 设计师使用积木 Sketch 插件绘制设计稿，可以保证设计元素均从既定的设计标准中获取，产出符合业务设计规范的设计稿，而代码组件库中也有对应的实现。
3. 绘制完成的设计稿上传至印迹云协作平台，交付开发同学进行设计稿还原。
4. 开发同学拿到设计稿后，就可以知道本次需求哪些组件已内置于代码组件库中，并可以点击设计稿中的链接，直接查看组件的使用说明。



UI 一致性协作流程闭环

2.3 方案落地

虽然 UI 一致性在落地上会增加开发同学不少的工作量，推进一致性建设也是一个艰难的工作，由于成本较高，且无法量化评估收益，很多团队最终未达到预期效果，但一旦有效运作起来后，团队将获得丰厚的回报。UI 一致性的建设需要设计者对现有状态有足够的认识，对业务有充分理解，以及优秀的设计能力，同时还要不断地进行实践和优化。为了保证一致性项目的成功落地，避免“半途而废”，我们制定了一系列的推进措施：

1. 项目小组不能脱离日常需求开发工作。这样可以保证设计师所沉淀的设计元素始终来自于最新的业务场景，同时项目产出可以快速应用到最新的版本中得以验证。
2. 优先选择受视觉因素影响较大、投入产出比高的模块场景进行改造，化繁为简，确定最小验证闭环（MVP，Minimum Viable Product），在实践中不断优化，进而跑通整个流程。
3. 项目推进由 UI 同学按版本提出需求，移动端排期并落地实施，由 UI 统一验收。
4. 建立阶段性目标，并完成最近三期工作的具体规划，定期复盘完成情况，保证项目的持续推进。

2.4 一致性成果

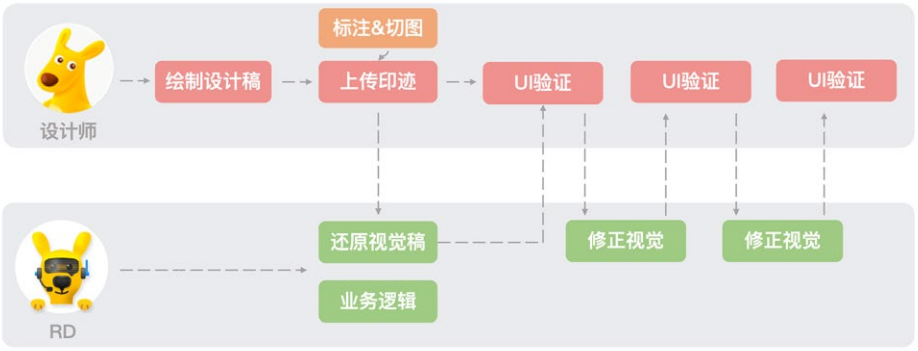
经过一段时间的 UI 一致性建设，在资源一致性方面，外卖 App 团队已经完成了近百个 Iconfont 的替换工作，有效减小了安装包的体积。在组件代码库建设方面，完成组件替换三十多处，中等业务需求平均节约 3pd 人力；在工具链方面，根据 UI/UE 提供的数据，对于强依赖设计资源的需求，在使用积木 Sketch 插件后，提效能够达到 30% 以上，对于 UI 资源依赖不强的流程需求，平均提效可以达到 50% 以上。

3. 设计体系建设

细化来看，UI 一致性整体方案主要分为两个部分，一个是设计体系建设，另一个则是工具链建设。设计体系建设是基础，主要是设计师沉淀设计风格，建立统一的 UI 设计标准的工作，而工具链建设则是支撑，是开发人员通过开发一系列的工具将开发过程闭环，实现设计体系落地。

3.1 外卖 DPL

DPL（Design Pattern Library）是一份面向 UED 设计人员的文档化说明，描述了设计模式库的规范以及应用场景等，外卖 DPL 主要包括组件搭建规范以及资源一致性两部分。DPL 的背面是技术实现，一般体现在 Android/iOS/RN 代码框架中，比如阿里的 FusionDesign 库、腾讯的 QMUI 库等，这些封装好的代码组件面向程序开发人员。在未建立 DPL 模型之前，开发同学拿到设计稿进行视觉还原后，需要修改多次，才能最终通过设计师的验证，极大影响了开发效率，还降低了需求吞吐率。



未建立外卖 DPL 模型之前开发流程

而通过 DPL 实现设计 – 开发流程的闭环，UI 同学由于设计规范的标准化，可使出稿效率、走查效率显著提升，重复组件甚至无需走查；对于 RD 同学来说，组件库中的组件在配置正确的情况下，由于已经经过了历史版本的检验，适配问题出现较少，无需重复进行视觉的修正；对于设计团队来说，优秀的设计体系具有包容性且充满生命力，好的设计模式库能够帮助实现规范化，从而减轻界面开发的工作量，提高一致性；而对于设计师来说，建立 DPL 有助于减少误用、滥用以及无效的创新。

3.2 组件搭建

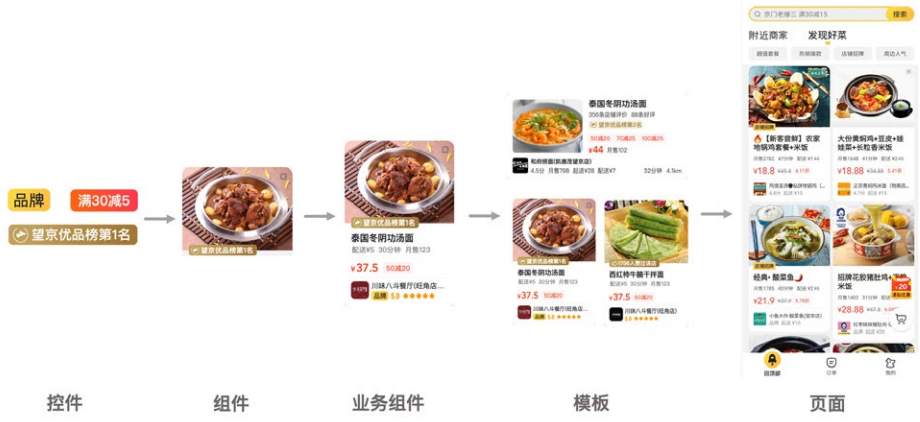
在长期的版本迭代中，随着功能的不断增加以及 UI 的持续改版，新旧样式混杂，维护极为困难。设计师通过将页面走查结果归纳梳理，制定设计规范，从而选取复用性高的组件进行组件库搭建。通过搭建组件库可以进行规范控制，避免控件的随意组合，减少页面之间的差异；组件库中组件满足业务特色，同时可以应对不断变化的环境，具有云端动态调整能力，可以在规范更新时进行统一调整。

在不影响需求实现以及设计效果的前提下，只有在方案设计中尽可能使用组件，提升组件设计稿中的覆盖度，才可能真正通过组件库来提效。而除了在新的需求中使用组件，还需要将已有页面内容尽量替换成组件，才能避免页面升级时的重复修改问题，真正提高产研效率。在进行组件库建设时要注意以下几点。

选择合适粒度

组件的粒度选择曾是困扰我们很久的一个问题，虽然有构建设计系统的核心理论——原子设计理论为指导，即按照“原子、分子、组织、模板、页面”五个层面进行页面设计。这一理论对于从零开发新应用没有任何问题，但进行一致性改造的 App，往往已经暴露出很多设计问题，已经存在数百个成熟的线上页面，改造存在非常大的困难，必须根据具体业务选择合适粒度。在进行组件制作前，项目同学对外卖的近百个页面进行了梳理，对使用到的组件进行了分类，并根据组件的使用频率进行排序，制定了逐步替换计划。从而避免了组件库做的很全、花费了很多的人力，但实际很多组件都用不上，或者开发的组件过少，覆盖场景不足等问题。

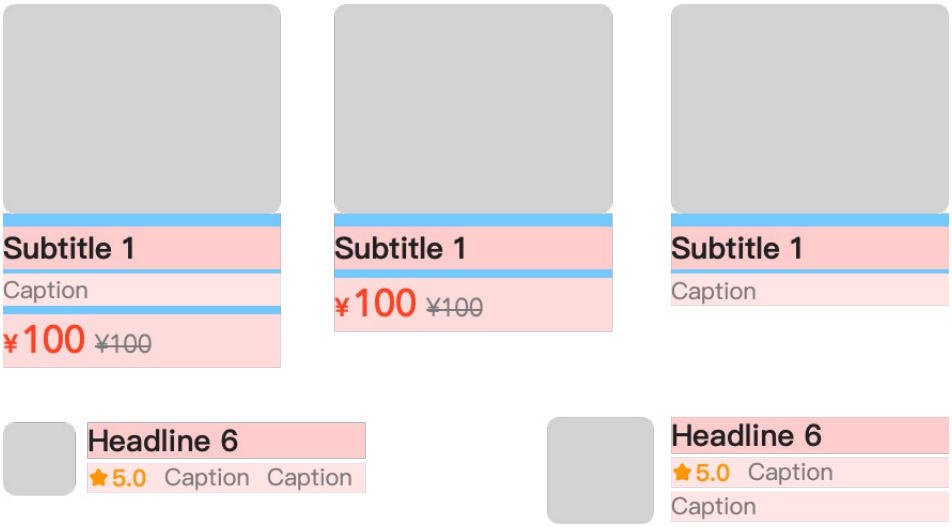
我们将走查结果与设计师反复交流，发现复用性较高的组件大体可以分为两类：第一类“基础控件”，也就是类似于标签、按钮、开关等具备基础功能的元素，对应原子理论中的原子；第二类“业务组件”，类似于商品卡片等，是由“基础控件”组成（比如商品卡片由“标签控件”与“图片控件”组成），同时“业务组件”还能相互组合，成为更高阶的“复杂组件”，类似于原子理论中的分子。“业务组件”的组合又是千变万化的，不同样式的业务组件可以组成类似“商家列表”、“菜品列表”等“模板”，而“模板”与“基本控件”组合在一起，就成为了“页面”。



外卖 DPL 模型建立

具备拓展性

组件必须具备一定的可配置属性才能提升适用场景。可配置属性体现在三个方面：组件支持局部元素展示隐藏，例如商品卡片的标题、说明、价格可根据接口数据控制展示逻辑；组件支持多种样式，例如商品卡片的左图右文排列、上图下文排列；组件支持业务方配置主题，如调整高亮色、调整对齐方式等。



组件应具有拓展性

支持统一管理

组件管理功能对外卖 UI 一致性起着至关重要的作用，这主要体现在两方面：首先是设计风格沉淀，目前袋鼠 UI 已经形成了自己的独特风格，外卖设计团队根据设计规范，对符合 UI 一致性外卖业务场景的组件不断进行抽象及建设，沉淀出越来越多的通用业务组件，这些组件需要及时扩充到 Library 中，供团队成员使用；另外一个作用则是保持团队使用的均为最新组件。由于各种原因，组件的设计元素（色彩、字体、圆角等属性）可能会发生变更，需要及时提醒团队成员更新组件，从而保持所有页面的一致性。

3.3 资源一致性

UI 设计语言与自身业务关联性很强，美团很多业务包括外卖、酒旅、团购等都有一套自己的设计系统。“通用”意味着无法满足具有业务特色的需求，不同业务的组件、色彩系统、动效、字体样式等千差万别，其中任意一环的缺失都会导致一致性被破坏。

设计语言并不是一个抽象的概念，大家提到美团就想起美团黄，想到袋鼠，想到菜品卡片列表，想到骑着摩托车穿着印有“美团外卖”衣服的骑手，通过设计语言可以传达品牌主张和设计理念。目前，袋鼠 UI 已经形成了一套属于自己的独特风格，对于一致性元素处理有了一套自己的标准，对于产品的设计者而言，必须将这种风格化延续，才能使我们整个项目具备高度的一致性，才能保持“袋鼠特色”，保证吸引力。

3.3.1 图片

建立插画库

插图作为一种视觉语言，是品牌识别度的关键核心元素，与单纯的文案信息不同，图形化在直观描述固有信息的同时，也在塑造情感背景，使用户更具沉浸感和共情性。插画在提升产品用户体验的同时完成商业目标，在表达效果及生产效率上有独特的优势，在追求效率的互联网产品中被大量地运用。

由于之前产品中的插图未经系统整合，而插画师的个人风格明显，不同的设计师在图形化的工作协同中，风格很难复现，而单纯由一名设计师去完成整体业务的插画建设工作也存在一定风险。不同设计师之前画过的元素无法互通，造成很多元素重复设计、风格不统一，缺乏系统性地创作和整理，无法最大化地提升生产效率，并且影响产品的品质感。所以插图体系在保持品牌一致性、提升工作效率以及规避风险上尤为重要。

插画规范示例

使用 Iconfont

Iconfont 可译为图标字体，顾名思义就是用字体文件取代图片文件来展示图标、特殊字体等元素的一种方法。简单来说，Iconfont 就是把多个图标文件打包为 ttf 字体文件，注册到系统中，App 可以像使用字体一样使用图标。其原理可以简单理解为通过 ttf 字体文件维护一个 Unicode 码与图形的映射关系。当使用 Iconfont 为项目助力的时候，配置多个图标不再需要去下载数个 PNG 文件，仅需要维护一套 ttf 字体文件即可。Iconfont 不仅具有矢量性、可自由变化大小的特点，而且支持任意改变颜色。从项目角度来看，由于无需针对不同手机分辨率内置多张图片，可以一定程度减小包体积，而且方便 UI 同学对图标进行统一管理，为无用 icon 和相似 icon 检测做基础。

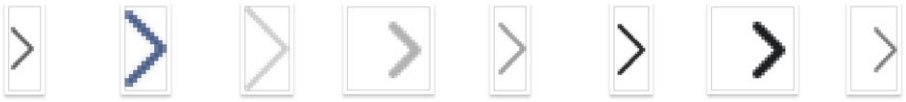
使用 iconfont 替换项目中的图片

归档图片文件

当 App 发展到一定阶段，必然面临着包体积会越来越大，无用图片与相似图片也会越来越多的问题。同时，由于开拓新业务而不断涌现的新场景，又不可避免地新增大量的图片。总结来看，图片文件在一致性项目中需要解决两个问题，即存量图片的处

理以及新增图片的管理。

对于存量图片，必须判断其合理性，项目中存在大量相似图片，这些图片可能仅是 padding 不同，或者颜色尺寸存在微小差异，可以通过脚本扫描相似图片，根据图片的特征 Hash 判断图片的相似度，相似度高的图片根据 UI 建议，保留一张即可。那如何防止新增图片“重蹈覆辙”呢？通过建立图片管理后台，将图片按场景分类，标准图片需从组件代码库中选取，新增图片执行 PR 策略，需相关负责人审核，可有效防止相似图片的堆积问题。

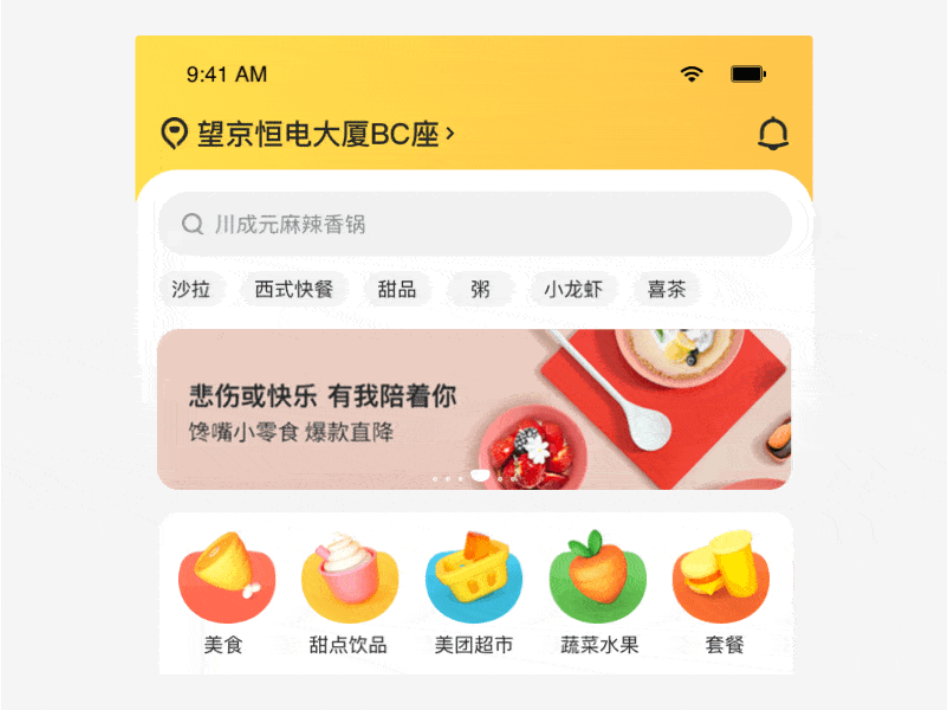


一致性项目实施前项目中的相似图片

3.3.2 动效

动效是指那些那些能够为产品赋予生机的动态界面元素及视觉效果，这些交互效果通常与特定的响应行为相关，甚至包括那些与交互行为没有直接关联的临时状态。精细而恰当的动画效果可以传达状态，增强用户对于直接操纵的感知，通过视觉化的方式向用户呈现操作结果。

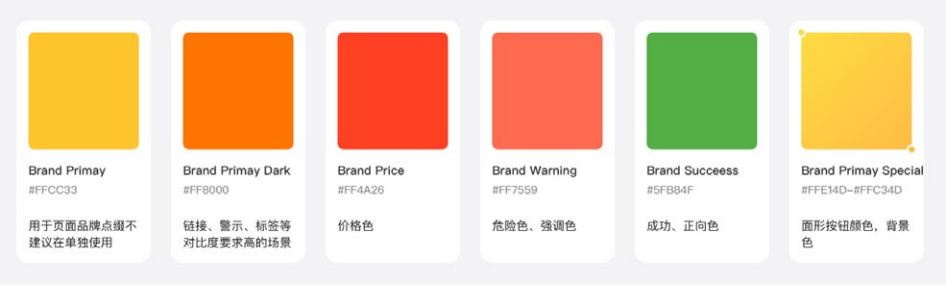
随着外卖业务的不断增加，动效的使用比重在不断增加，重要性日渐凸显，也是增强用户体验与竞品拉开差距的重要因素，因此，统一规范的使用动效尤为重要。通过动效库建设，UI 层面可以承载品牌、传递情感，加深用户对 App 的印象，让用户放松、愉悦；RD 层面同一组件可在多场景直接复用，还降低了研发成本。



动效

3.3.3 颜色

颜色可以起到传递品牌信息、区分信息的所属关系、标明控件的选中状态以及对内容的信息进行分层级展示等功能。重要的信息需要在页面中被突出展示。系统级色彩体系主要定义了外卖的主要颜色、文字颜色、辅助颜色以及标准渐变色，颜色在一定时期内不再支持新增。通过将标准色板内置于积木 Sketch 插件中，限制 UI 绘制设计稿时的使用范围，而 RD 同学仅可通过代码组件库中选取颜色，保证色值的准确性，也便于进行主题定制。



定义颜色使用场景

3.3.4 字体

字体是体系化界面设计中最基本的构成之一。用户通过文本来理解内容和完成工作，科学的字体系统将大大提升用户的阅读体验及工作效率。设计师在字体设计过程中需要关注非常多的方面，比如字体 family、字距、行高、段落等等。如何让文字看起来更自然，是设计师团队一直探寻的答案，UI 同学根据文字的层级关系，规定了 Headline、Subtitle、Body、Button 以及 Caption 的文字使用规范，根据设计稿中文字的位置，就可确定文字的具体样式。

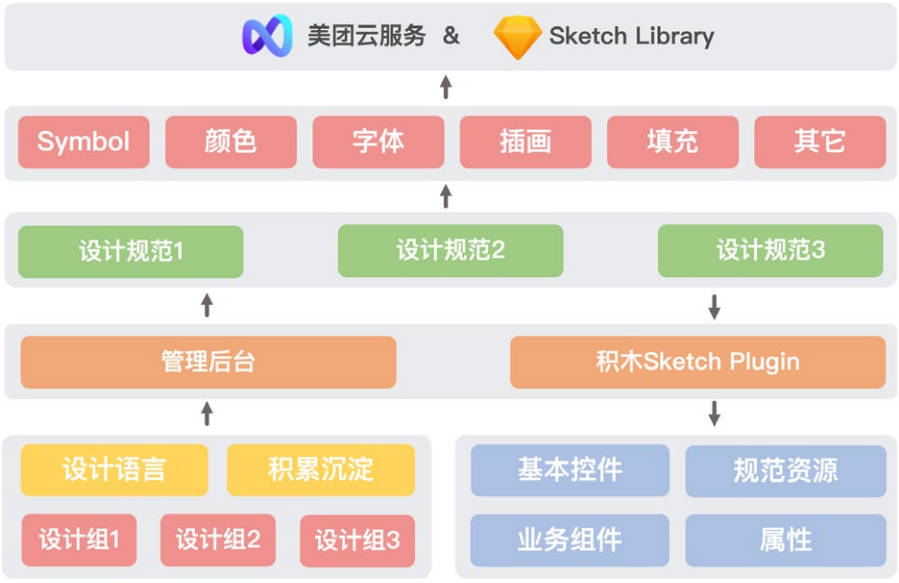
定义字体使用规范

4. 工具链建设

要想保持 UI 一致性，就不能打破规则。外卖 UI 一致性套件由积木工具链、代码组件库、定制化设计云协作平台以及官网四部分组成，通过将这四部分连接起来，形成一个闭环，把整个工作流限制在标准操作以内。

4.1 积木 Sketch 插件

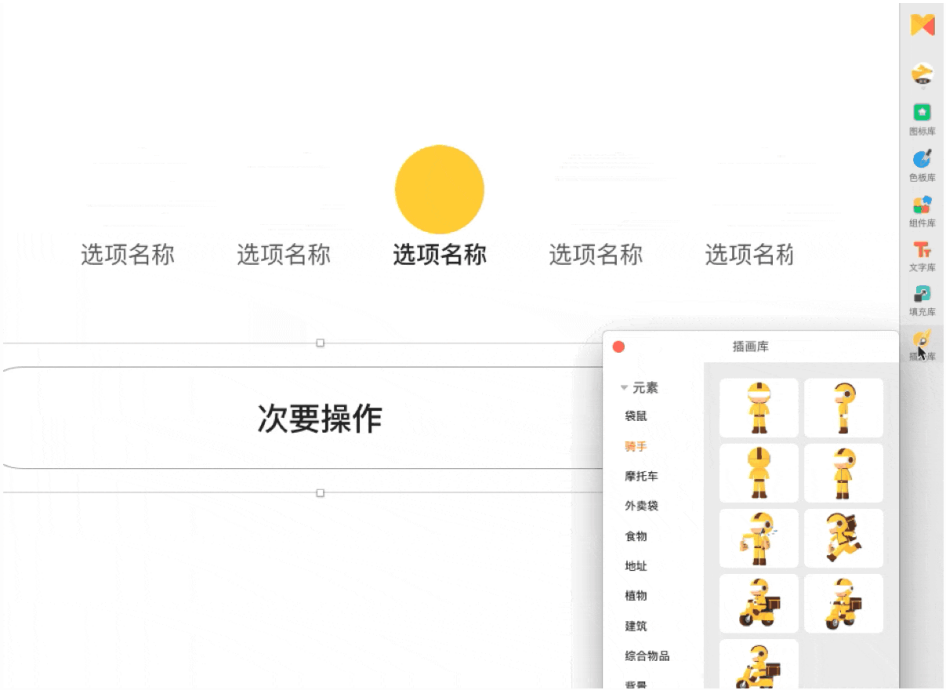
在之前的文章中，我们已经对积木插件进行了详细介绍，这里只作简要概述，介绍其在一致性项目中发挥的作用。从设计阶段颜色的选择、字体的规范、控件的样式，到 RD 开发阶段代码的统一管理、API 的制定、多端的实现方式都必须遵守一套规则，通过积木 Sketch 插件落地设计规范，可以保证设计元素均从既定设计标准中获取，产出符合业务设计语言的设计稿，而各平台 UI 代码库中也有对应实现，从而使积木插件成为 UI 一致性的抓手。



积木 Sketch Plugin 平台化示意

4.1.1 插件功能

积木 Sketch 插件经过一段时间的建设，目前已具备 Iconfont、标准色板、组件库、数据填充、文字模板等功能。通过 Iconfont 可以从公司图标库中拉取设计团队上传的 SVG 图标，并直接应用于设计稿；标准色板可以限定设计师的颜色使用范围，确保设计稿中的颜色均符合设计规范；组件库中包含从外卖业务抽离的基本控件与通用组件，具有可复用和标准化的特点，并与不同开发语言组件库中的代码一一对应；数据填充库可以使设计师采用真实数据进行填充，使设计稿更贴近线上环境；文字模板中内置了字体样式的使用规范，根据设计稿中文字的位置，点击文字图层即可直接应用。



积木 Sketch Plugin 功能演示

4.1.2 物料市场

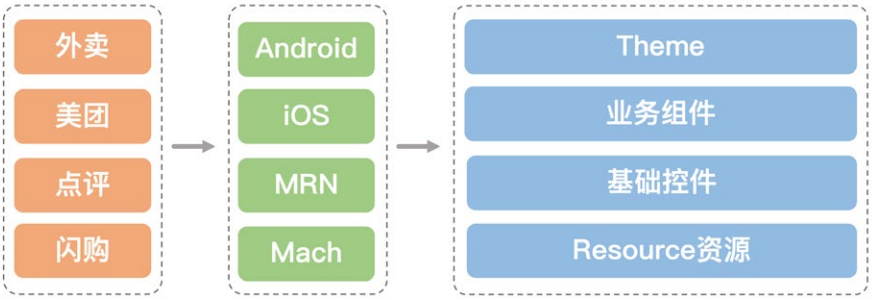
通过 Sketch 管理后台，设计师可以将配色规范、文字规范、话术、Iconfont、组件

库上传至云端并与整个设计团队中成员共享，并可实现设计资产的版本管理。通过将 Sketch Library 存储在后台物料市场，设计师可以与团队成员共享组件（Symbol），Library 可以实现“一处更改，处处生效”，即使是关联了远程组件库历史的设计稿检测到更新时，也会收到 Sketch 通知，确保工作中使用的是最新组件。

积木 Sketch Plugin 物料管理后台

4.2 代码模型建设

为了满足中小企业的需求，越来越多的开源组件库诞生，但开源代表着“通用”，无法满足业务特色的需求，于是很多企业也开始做起了自己的组件库。通过建立代码组件库，能帮助开发同学快速搭建 App 页面，减少设计与开发沟通成本，统一体验规范等。



代码组件库模型

4.2.1 代码库功能

提高项目可维护性

由于组件库中的组件职责单一，降低了系统的耦合度，开发人员可以很容易地了解该组件提供的能力。组件可以自由替换、组合为高阶组件，在进行组件更新时仅需修改一处。每个项目成员维护一定数量的组件，让团队中每个人都能发挥所长，可以最大化团队的开发效率。

实现文档化

组件接口有统一的规范，降低新人的上手难度，新成员只需要理解接口和职责即可开发组件代码，由于代码的影响范围仅限于组件内部，对项目的风险控制也非常有帮助。通过对组件统一管理，实现代码的积累沉淀与有效复用，全面提升了新业务的需求开发效率。

便于单元测试

由于组件职责单一而清晰，对外仅暴露接口，概念上可以把组件当成一个函数，输入对应着输出，这让自动化测试变得更加简单。

实现无障碍等定制化功能

无障碍功能可以改善残障人士的用户体验，组件库中的组件资源高内聚，完全由自身控制加载，不与全局或其他组件产生影响。组件的加载、渲染路径清晰可控，对于组件功能定制，实现类似于无障碍等功能较为方便。

4.2.2 方案设计

统一配置文件

前文也提到，外卖业务入口覆盖外卖独立 App、美团外卖频道以及大众点评外卖频道等，外卖组件需要在不同的移动端上适配宿主 App 的 UI 风格及交互体验，这就需要组件库支持主题配置功能。由于主题涉及 Android/iOS/MRN 多端，需要一套通用

的主题配置文件。经过了各端开发同学与设计师的多轮讨论，最终确定了包含主题颜色、文字外观、组件风格等内容的主题描述文件格式。配置文件通过下发，就可以实现全局替换主题的功能。

```
{
  // 主题颜色
  "rooBrandColors": {
    "rooBrandPrimary": "#FFCC33"
  },
  // 文本外观
  "rooTextAppearance": {
    "rooTextAppearanceHeadline1": {
      "fontFamily": "sans-serif-medium",    // 字体
      "textStyle": "normal",                // 风格(normal/bold/italic)
      "textSize": 44,                       // 字号
    }
  },
  // 组件风格
  "rooStyle":{
      "rooButtonStyle": {
        "textAppearance": "?attr/rooTextAppearanceButton",
        "backgroundColor": "?attr/rooBrandPrimary",
        "cornerRadius": 0,
      }
  }
```

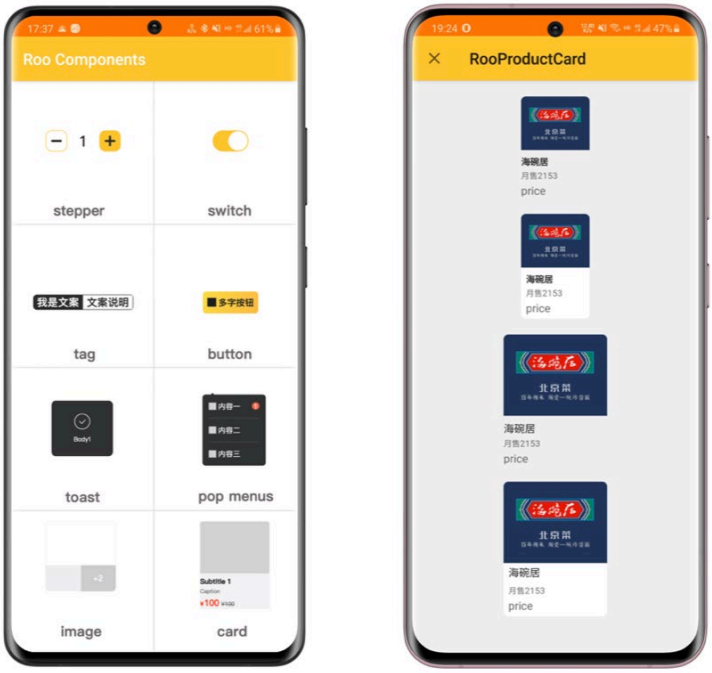
搭建全平台组件库

目前，市面上耳熟能详的组件库包括阿里的 Antd Desigin、Fusion Design 以及美团的 Roo Design 等，基本都是服务于 Web 开发的组件库，通过这些组件库可以快速搭建一些中后台系统。为什么没有知名的 Native 开源组件库呢？因为每个应用的主题、风格以及交互体验都是不同的，而这些不同恰恰是传达品牌主张和设计理念的灵魂，因此必须结合业务，针对 Android/iOS/MRN 三端进行组件库开发。通过搭建全平台代码组件库，可以保证同一个 UI 组件在各端表现一致，进行 UI 升级时降低改错或遗漏的风险，除此之外，还能降低测试压力，提高需求的吞吐率。

4.2.3 示例应用

我们针对 Android/iOS/MRN 三端代码开发了示例工程，通过示例工程，不仅可以帮

助 UI 同学完成组件验收，还能帮助开发同学快速查阅可以应用的所有组件，了解其使用方式以及进行代码调试。



组件库 demo 示例

4.3 官网门户建设

官网相当于项目的门面，一个好的门面才能吸引更多的用户，才能更好地对项目进行推广。官网作为设计师与开发同学沟通的媒介，需要两者共同维护。通过官网可以帮助团队内设计师沉淀设计风格，建立统一的 UI 设计规范，帮助 RD 同学进行组件文档管理与查阅。

4.3.1 官网功能

当前的官网主要由四部分组成，分别是设计语言、组件库、插画库以及资源下载，分别服务于 UI 和 RD 同学。

外卖平台化官网导航栏

设计语言

UI 一致性项目中采取了“原子理论”的构成原理，即从最小的元素开始定义，进而将这些元素按照规则进行组装，拼接成组件，最后通过这些组件拼接成最终的页面，这是一个由点到面的过程。设计语言章节主要服务于 UI/UE 同学，该章节通过视觉、设计模式、动效等三个子章节使得读者能够快速了解项目的设计规范，从而快速上手。

组件库

组件库是设计模式中各种元素的具体实现，在这个页面描述了组件的使用方式。

插画库

插画库中则介绍了插画的使用场景，插画的绘制规范以及插画案例展示。

资源下载

提供积木工具链产品下载功能。

外卖平台化官网

4.3.2 方案设计

由于官网以纯粹的图文展示为主，且迭代频率较快，因而选择了当下较为普遍的文档 – 网站生成系统，即只需按照一定规范将书写的 Markdown 文档放至相应目录，前端自动解析后生成导航，并且支持多语种、图片、文件、视频等素材。这种方式极大地缩短了官网的开发周期，即便是没有前端经验的同学也能快速上手。

为了方便 UI 同学对官网文档的修改，我们基于文档网站生成系统搭建了在线编辑平台。通过该平台，相关人员可以直接做到在线编辑、发布，节约了 UI 同学与 RD 同学的沟通成本以及发布成本。填充期间，使用者可以实时预览编辑的内容，修改后只需点击保存即可立即更新到网站中。

官网支持平台化功能，不同业务方可以创建属于自己的文档站点，一个好的文档站点对于设计组的方案推广、外部接入都大有裨益。

外卖平台化官网录入后台

4.4 工具链的闭环

当我们信心满满的把 UI 一致性解决方案推广到日常开发中时，除了听到可以提升效率的褒奖外，开发同学对项目的吐槽声也常常传入我们的耳边，“怎么才能知道设计稿中的哪个组件已经开发完成了?”，“查询这个组件的使用方法好麻烦，每次都要去

官网检索”，“规范颜色、图标的名称是什么？怎么才能引用到?”我们无法限制别人的选择，所以只能让这套体系变得更好用，如果不去优化整个流程，将其串联起来形成闭环，后面整个项目可能都会慢慢崩塌，所以我们对项目进行了重新复盘，经过大家集思广益，终于找到了能将工具链体系打通的解决方案：设计稿作为衔接 RD 与 UI 的纽带，可以把官网与代码仓库打通。

我们与美团内部的印迹团队合作开发，然后定制了一个设计云协作平台，在给 RD 的设计稿中标注了哪些是代码组件库中已有的元素，避免了重复开发，同时关联了官网上该组件的使用说明，RD 同学在开发过程中遇到组件使用问题无需检索，点击即可跳转至使用文档。后期我们还将颜色、iconfont 以及插画库中图片也进行了关联，真正做到了一致性元素的全覆盖。

与印迹合作支持组件展示的云协作平台

加入我们

UI 一致性项目原本只是外卖技术团队提升 UI/RD 协作效率的一次尝试，现在已经作为全面提升产研效率的媒介，承载了越来越多的功能。围绕设计日常工作，提供高效的设计元素获取方式，让工作变得更轻松，是我们的核心目标。如何推动设计规范落地，并且输出到各个业务系统灵活使用，是我们持续追寻的答案。探寻研发和设计更

为高效的协作模式，是我们一直努力的方向。

如你所见，通过 UI 一致性建设可以帮助设计团队提升设计效率、沉淀设计语言以及减少走查负担；让 RD 同学面对新项目时可以专注于业务需求，而无需把时间耗费在组件的编写上；减少 QA 工作量，保证控件质量无需频繁的回归测试；帮助 PM 提高版本迭代效率及版本需求吞吐量，提供业务的快速拓展能力。当然，我们除了希望制作一流的产品，也希望可以让大家在繁忙的工作中得以喘息。我们会继续以设计语言为依托，以工具链建设为抓手持续进行 UI 一致性建设，不断提高移动端 UI 业务中台能力。

如果你也想参与我们的 UI 一致性项目建设，欢迎加入我们！

致谢

- 感谢晓飞、彦平、杜瑶、冰冰、云鹏对项目的大力支持。
- 感谢到家优秀设计师淼林、昱翰、冉冉、璟琦、雪美、田园。
- 感谢用户平台组参与 UI 一致性建设的开发同学王鹏、腾飞、赵炎、瀚阳等，感谢美团印迹开发团队的支持。
- 感谢所有参与人的努力与坚持，为外卖 DPL 建立贡献力量！

参考文献

爱奇艺产品工作流优化：搭建组件库做高 ROI
阿里重磅开源中后台 UI 解决方案 Fusion
Ant Design 背后的故事
Google Material Design
前端组件化开发方案及其在 React Native 中的运用

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，欢迎加入外卖 App 大家庭。感兴趣的同学可投递简历至：tech@meituan.com（邮件主题请注明：外卖大前端）。

美团外卖 Flutter 动态化实践

作者：尚先

一、前言

Flutter 跨端技术一经推出便在业内赢得了不错的口碑，它在“多端一致”和“渲染性能”上的优势让其他跨端方案很难比拟。虽然 Flutter 的成长曲线和未来前景看起来都很好，但不可否认的是，目前 Flutter 仍处在发展阶段，很多大型互联网企业都无法毫无顾虑地让全线 App 接入，而其中最主要的顾虑是包大小与动态化。

动态化代表着更短的需求上线路径，代表着大大压缩了原始包的大小，从而获得更高的用户下载意向，也代表着更健全的线上质量维护体系。当明白这些意义后，我们也就不难理解，在 Flutter 的应用与适配趋近完善时，动态化自然就成为了一个无法避开的话题。RN 和 Weex 等成熟技术甚至让大家认为动态化是跨端技术的标配。

美团外卖 MTFlutter 团队从 2019 年 9 月开始对动态化进行研究，目前已在多个业务模块上线，内部项目代号“Flap”。

二、Flap 的特点与优势

Flap 研发的初心是为了提供一个完整解决方案，而不是一个过渡方案。项目组思考了当下最痛的点并逐一列出，然后再根据目标来做具体选型。在前期，只有需求考虑得越周全，后续的架构和研发才会越明确。在研发过程中，团队应该坚守底线，坚守初心，不断攻克困难，完成昔日定下的目标。

2.1 核心目标

- 通用性，保持 Flutter 多平台支持的能力且方案无平台差异。
- 低成本，动态化对齐 Flutter 生态和常规开发习惯，且可低成本转化现有的 Flutter 页面。

- 适用性，避免包过大、不稳定等不利于应用的缺陷。
- 高性能，保留 Flutter 渲染性能极佳的特点。

2.2 动态化选型

a. 产物替换

选型中首先考虑到的是下发产物替换，官方在也曾经推出了 Code Push 方案，甚至可以支持 Diff 差量下载，但是在 2019 年 4 月被叫停，这里引用一下官方的发言 Flutter/issues/14330：

> To comply with our understanding of store policies on Android and iOS, any solution would be limited to JIT code on Android and interpreted code on iOS. We are not confident that the performance characteristics of such a solution on iOS would reach the quality that we demand of our product. (In other words, “it would be too slow”.)
>
> There are some serious security concerns. Since these patches would essentially allow arbitrary code execution, they would be extremely attractive malware vectors. We could mitigate this by requiring that patches be signed using the same key as the original package, but this is error prone and any mistake would have serious consequences. This is, fundamentally, the same problem that has plagued platforms that allow execution of code from third-party sources. This problem could be mitigated by integrating with a platform update mechanism, but this defeats the purpose of an out-of-band patching mechanism.

简而言之，就是官方对动态化后的性能没有自信，并且对安全性有所顾虑。之前，官方提供方案的局限性也十分明显。比如对 Native-Flutter 混合 App 支持不友好，并且无法进行灰度等业务定制操作，所以不能满足通用性和高性能的核心目标。

b. AOT 搭载 JIT

Flutter 在 Release 模式下构建的是 AOT 编译产物，iOS 是 AOT Assembly，Android 默认 AOTBlob。同时 Flutter 也支持 JIT Release 模式，可以动态加载 Kernel snapshot 或 App-JIT snapshot。如果在 AOT 上支持 JIT，就可以实现动态化能力。但问题在于，AOT 依赖的 Dart VM 和 JIT 并不一样，AOT 需要一个编译后的“Dart VM”（更准确地说是 Precompiled Runtime），JIT 依赖的是 Dart VM（一个虚拟机，提供语言执行环境）；并且 JIT Release 并不支持 iOS 设备，构建的应用也不能在 AppStore 上发布。

实现此方案需要抽离一份 DartVM 独立编译，再以动态库的形式引入项目。通过初步测试，发现会增大包体积 20MB+，这超过了 MTFlutter 之前做 Flutter 包体积优化的总和。进一步让 Flutter 包体积成为推广与接入业务方的巨大阻碍，不满足我们对适用性的要求。

c. 动态生产 DSL

Native 侧本身具备 JS 动态执行环境，利用这个执行环境动态生成包含页面和逻辑事件绑定 DSL，进而解析为 Flutter 页面或组件，也可以实现动态化诉求。技术思路接近 RN，但与其不同的是利用 Flutter 渲染引擎和框架。这种先将代码执行起来再获取 DSL 的手段，我们简称为动态生产 DSL。

此方案可以很好地支持逻辑动态化，但弊端也比较明显。首先要对齐 Flutter 框架，JS 侧的开发量很大且开发体验受损。另外，对 JS 的依赖偏重，构建的 JS 框架本身解释执行有一定开销，对于页面逻辑与事件在运行中需要频繁地进行 Flutter 与 JS 的跨平台通信，同样也会产生一定开销。这不能满足 MTFlutter 团队对高性能的诉求。更严重的是，此方案对开发同学的开发习惯并不友好，将 Dart 改为 JS，现有的 Flutter 开发工具无法直接使用，这与低成本诉求背道而驰。

d. 静态生产 DSL

前面说“将代码执行起来再获取 DSL 的手段，我们简称为动态生产 DSL”，那么代码不执行直接转换 DSL，就称为静态生产 DSL 方案。

静态生产的特点是抹平了平台差异，因为 input 是 Dart source 与平台无关，直接将 Dart source 内的完整信息通过一层转换器转换到 DSL，然后通过 Native 和 Dart 的静态映射和基础的逻辑支持环境，使得其可以在纯 Dart 的环境下渲染与交互。

在具体实现上，可以利用 Dart-lang 官方提供的 Analyzer 分析库（该工具在 Dartfmt、Dart Doc、Dart Analyzer Server 中都有使用）构建 DSL。该库提供了一组 API 能对 Dart source 进行分析，按照文件粒度生成 AST 对象。AST 对象用整齐的数据结构包含了 Dart 文件的所有信息，利用这些信息可以便捷地生成所需的 DSL。**所有的这个分析 + 转换的过程全部在线下进行**。接下来，DSL-JSON 以 Zip 的形式下发，Flutter 的 AOT 侧以此为数据源，完成整个 Flutter 项目的渲染与交互。

这种方案，一来可以保持 Flutter/Dart 的开发体验，也没有平台差异，逻辑动态化依赖静态映射和基础逻辑支持，而非 JScore，有效地避免了性能上的开销。综上考虑，静态生产 DSL 最终成为 MTFlutter 团队选型的方案。

2.3 项目架构

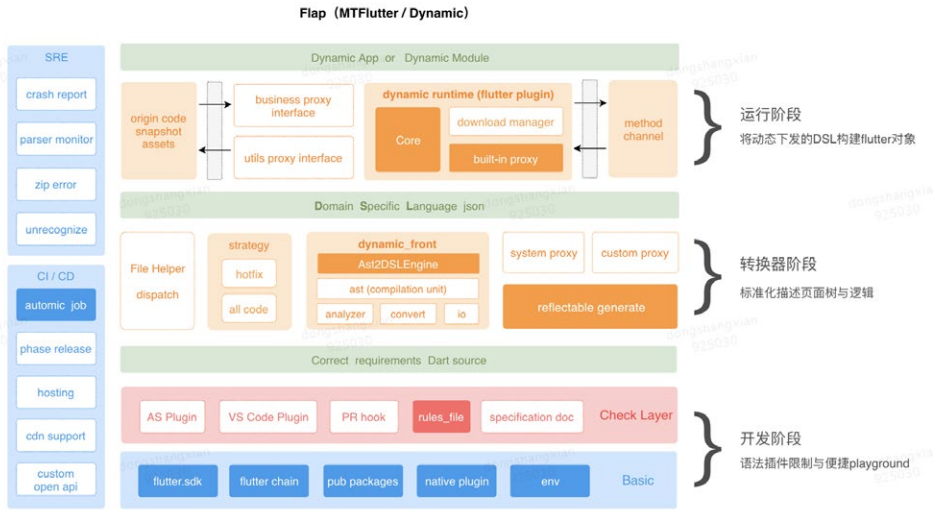


图 1　Flap 整体架构

如图 1 所示，三处浅绿色部分为一个阶段的阶段产物，起到承上启下的作用。以绿色部分为界，整体架构自然而然的就被划分成了三个区域：

- 下层第一部分是对开发阶段的赋能，产物是正确且规范（也满足 Flap 规范）的 Dart 源码。
- 第二部分是 DSL 的转换器，产物是 JSON 格式的 DSL，用于标准化的描述页面层级与逻辑。
- 上层的第三部分是运行时环境，准备了所有需要的符号构建 Dart 对象与逻辑，产物是动态化 App 或动态化的模块。

三、Flap 的原理与挑战

图 1 中的核心模块是转换器部分和运行时部分，接下来会介绍下这两个部分的原理与部分实现。

3.1 转换器原理

AST & DSL

AST 意为抽象语法树（Abstract Syntax Tree）。Dart 的 AST 和其他语言的 AST 基本概念类似。'package:front_end/src/scanner/token.dart' 中定义了所有的 Token，AST 也是通过词法分析、语法分析、解层级嵌套得到。ASTNode 对象作为存储编译单元中重要信息的基本数据结构，派生类基本分为 Declaration、Expression、Literal、Statement。

DSL 意为领域特定语言（Domain-specific Language）。表示专门针对特定问题领域的编程语言或者规范语言。相对自然语言，编程语言是不灵活的，它的语法和语义设计常取决于它的执行环境和特定目的。过去人们总是发明新的编程语言，近年来新出现的语言越来越相近，因此 DSL 也变得流行起来。

那 Flap 的 DSL 具体是什么？对于开发者而言，那这个 DSL 就是 Dart Code。而对于机器或 App 而言，那这个 DSL 就是 JSON。

前面的技术选型中提到：

> 利用 Dart-lang 官方提供了 Analyzer 分析库，官方的 Analyzer 的能力可以拿来直接用，该库提供了一组 API 能对 Dart source 进行分析，按照文件粒度生成 AST 对象，该数据结构包含了 input 的 Dart 文件的所有信息。

我们的 DSL 的基本原理就是对 AST 内数据的一个描述，并附带一些其他操作。

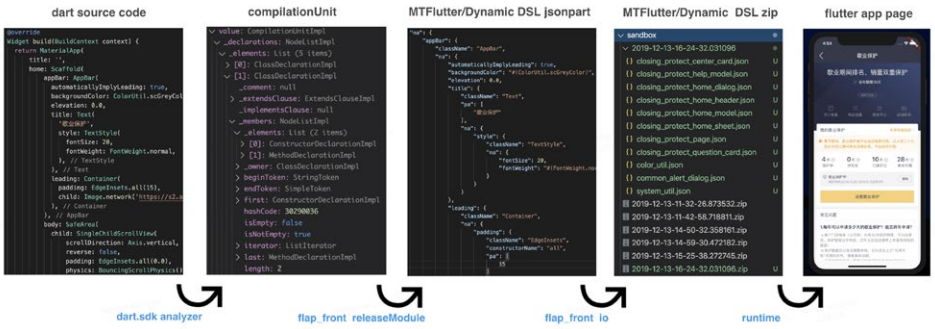


图 2　DSL-JSON 的转换步骤

因为用 Analyzer 的 API 跑出的 AST 也叫 CompilationUnit，实际上是一个编译单元，里面还存有很多编译相关的属性例如 lineInfo、beginToken 等。但使用 DSL 的方式不依赖编译，所以很多不需要的属性会被裁剪或忽略。

在转换器入口会对大类（identifier、statementImpl、literal、methodInvocation 等等）进行分发，每一个大类的数据结构使用一种中间结构 Dart model 来传输，然后对于大类中细分的类型（IfStatement、AssignmentStatement、DoStatement、SwitchStatement 等等），配有足够细粒度的转换接口，以 AST 结构作为输入，以 Map 节点作为输出。最终定义并提炼了 10 种标准的 Map 结构（class、method、variable、stmt 等等）来承载所有类型。

举个例子

一个简单的 Widget 节点经过转换后得到这样的 DSL-JSON，可以看到 DSL 的可读性还是 OK 的（默认下发时产物是一个压缩成单行并加密的二进制文件，这里是解密后 Format 换行后展示的）。我们在转换中会区分普通的字符串、变量名引用、系统枚举等类型，加以不同的符号表示。

图 3　常规 Widget 组件的源码与 DSL 示例

关于逻辑

举一个简单的四则运算的例子，可以看出在对于“乘法应当先计算”这个规则上，我们的 DSL 能够自动遵循，其中的奥秘是 Analyzer 帮我们做了这种运算优先级的判断，归根结底还是一种描述 AST 的工作，我们自己不会去根据静态代码做分析过程。

图 4　简单逻辑的代码与 DSL 示例

关于语法糖

语法糖往往画风清奇，结构与众不同，但是在 AST 中还是很诚实的，该什么结构就是什么结构。所以语法糖应该在转换器侧进行展开为常规结构再转 DSL，而不是对特殊格式设置特殊的 DSL 传到运行时再去解析。

```
AddressItem(this.header);
>
AddressItem(header){
  this.header = header;
}

AddressItem.defaultItem(this.header,
{Key key, this.card}) : super(key: key);
>
AddressItem.defaultItem(header,{key,card})
: super(key: key){
  this.header = header;
  this.card = card;
}

@override
_ShopPageState createState() => _ShopPageState();
>
@override
_ShopPageState createState() {
  return _ShopPageState();
}
```

图 5　部分语法糖的展开情况

这里只举了一些简单的例子，只是 DSL 体系中的一个片段，实际在项目落地时有很多较为复杂的逻辑，类似于循环套循环内进行集合操作或是异步回调内加多重三目逻辑等等。这里因为篇幅原因和涉及到业务代码相关就不展开详细的介绍了，其中的原理是一样的，都是描述 AST 的过程中增加一些特殊处理，最终会将转换产物的 Map 节点根据原有 AST 的层级结构组装起来，再通过 JSONEncode 转为 JSON。

图 6　DSL 内部结构层级

转换器侧能够完整的描述一个 Dart 文件的所有信息，如图 6 所示。值得一提的是，不同的节点还可能出现任意结构，method 里的 Argument 里可能是一个全局变量，条件表达式的右边又可能是一个方法。对于这种相同的结构即使出现在不同的位置也应当使用一套处理逻辑来转换，因此转换器是以迭代为主加小范围递归的设计思路。

将细粒度转换接口按照具体类别分在不同文件中（statement_factory、class_factory、function_factory）等待解析生产总线的调用。实际操作中各个类之间是近似于网状的调用，因此所有调用应当都是 Static 的，并且内部隔离，不引用不修改外部变量，做到无副作用。

DSL 转换器是一个命令行程序，因此可以无缝的部署到自动化的机器上。新代码合入主干后，接下来的 Bundle 生成与分发逻辑都可以使用各种图形化界面的发布系统来操作。

3.2 运行时原理

Prepare & Running

运行时相关的操作是在 App 内发生的，包括初始化，拉取 DSL，解析与使用。 简言之可以分为 Prepare 和 Running 两个阶段。Prepare 是准备各种运行时所需的符号，包括系统类符号与自定义符号，属性符号与方法符号（这里所说的符号实际就是 Dart 内的对象）。Prepare 阶段完成才能进行后续的 Running 相关操作，具体是页面的构建，事件的绑定，交互与逻辑的正常运转。

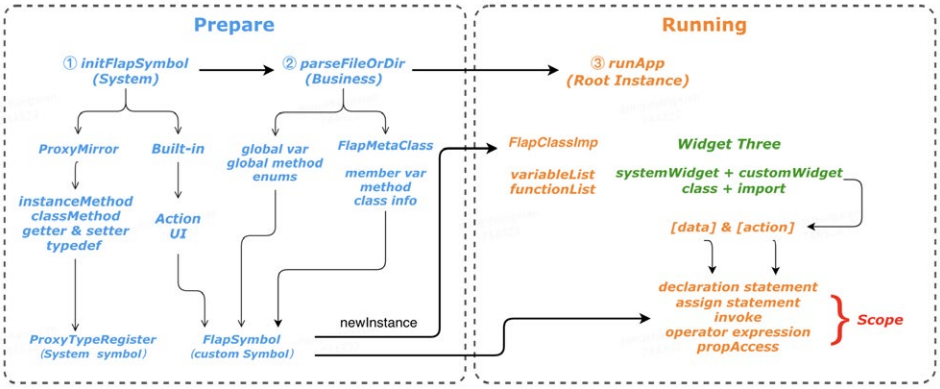


图 7　运行时原理的两大阶段

万能方法 Function.apply()

Flutter 期望线上产品是编译后的“完全体现”，同时为了避免生成过大的包，并不支持 Dart:Mirror。“Flutter apps are pre-compiled for production, and binary size is always a concern with mobile apps, we disabled dart:mirrors.”那么，在这种前提下，如何将外部符号转内部符号？ Function() 对象提供了这样一个万能方法。

```
// function.dart
external static apply(Function function, List positionalArguments,
      [Map<Symbol, dynamic> namedArguments]);
```

第一个参数是 Function 类型，后两个参数是该函数所需的参数（位置参数与命名参数，这两者在 DSL 中都可以取到），因此只要能获取到某个 Function，那就能在任何时候调用它。

此 Function 若为 Constructor Function 那返回值则为构造出的对象类型。

Proxy-Mirror

DSL 后只能得到字符串的标识，因此需要建立一个 String 与 Function 的映射关系，考虑到类名方法名，数据结构应该是 {String:{String:Function}}，通过 className 和 functionName 两个 String Key 即可取得一一对应的 Function()，下面给出一个

系统类的类方法（构造方法）的代码片段：

```
{
  'EdgeInsets':
  {
    'fromLTRB': (left, top, right, bottom) => EdgeInsets.fromLTRB(left,
top, right, bottom),
    // ...other function
  },
  // ...other class
};
```

然后对于系统类的实例方法、getter、setter 则需要在外部多传一个 instance 参数，instance 是外部通过该类的构造方法的 func 创建后传入。

```
// instance method
"inflateSize": (instance, size) => instance.inflateSize(size),
// getter
"horizontal": (instance) => instance.horizontal,
// setter
"last": (List instance, dynamic value) => instance.last = value,
```

Custom Class’s meta

对于自定义类，我们需要构建一个模拟的元类系统，存放所有符号信息，在解析时将所有的 JSON 节点转成可处理的对象。所有的属性声明都会构建成 FlapVariable 类型，所有的方法声明都会构建成 FlapFunction 类型。

如图 8 所示，父类和元类也是有相应的指针，父类的成员变量也会填充到子类，并且通过 mixin 的方式将类相关属性注入到派生类类型，例如 FlapState，FlapState 继承自 state，这样既可以让系统类的生命周期方法留个调用链的开口，也可以使用注入的运行时类属性。

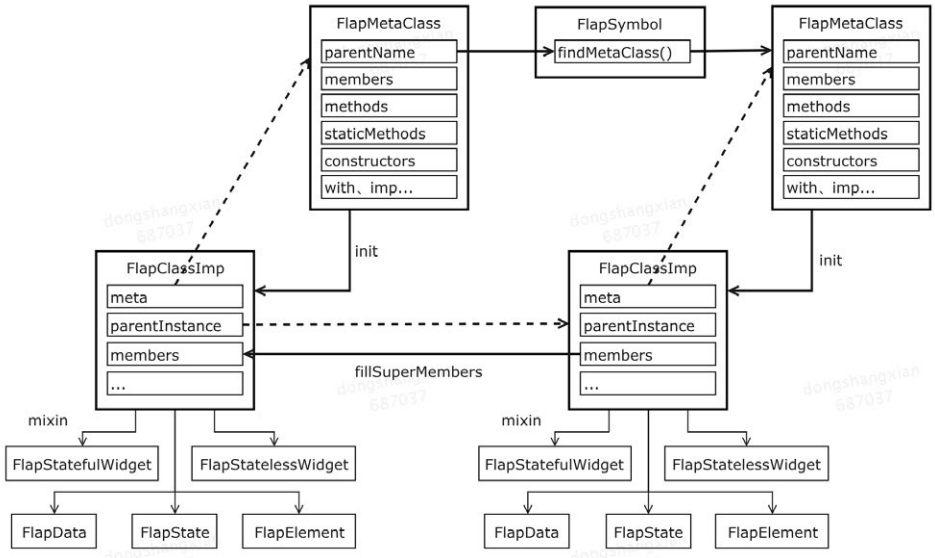


图 8　运行时模拟的元类系统

Evaluate

如下面代码的例子，一个 if 语句的 JSON 节点下发后，经过 parser 之后会得到一个 IfStatement 对象，这类对象都有一个特点就是包含几个属性，和一个运行时入口方法 evaluate（Scope scope）。这个方法在抽象类 Evaluative 类中，所有语句和表达式的类都会继承于此，自动获得 evaluate 方法，其中属性部分是在解析过程中解析成 Dart 对象后通过构造方法的参数传入的。

```
class IfStatement extends Statement {
  dynamic condition = undefined;
  Body thenBody;
  Body elseBody;
  IfStatement(this.condition, this.thenBody, [this.elseBody]);
  // 简化版代码
  ProcessResult evaluate(Scope scope) {
    bool conditionValue = condition.evaluate(scope)
    if (conditionValue){
      return thenBody(Scope);
    }else{
      return elseBody(Scope);
    }
  }
}
```

属性中的条件对象与语句对象在解析的过程中并不会被触发，真正的触发是方法被调用时从运行时的入口方法 evaluate 进入，此时才会通过作用域 Scope 判定条件是 true or false，然后调用到其他需要 evaluate 的 Dart 对象，如下图 9 所示：

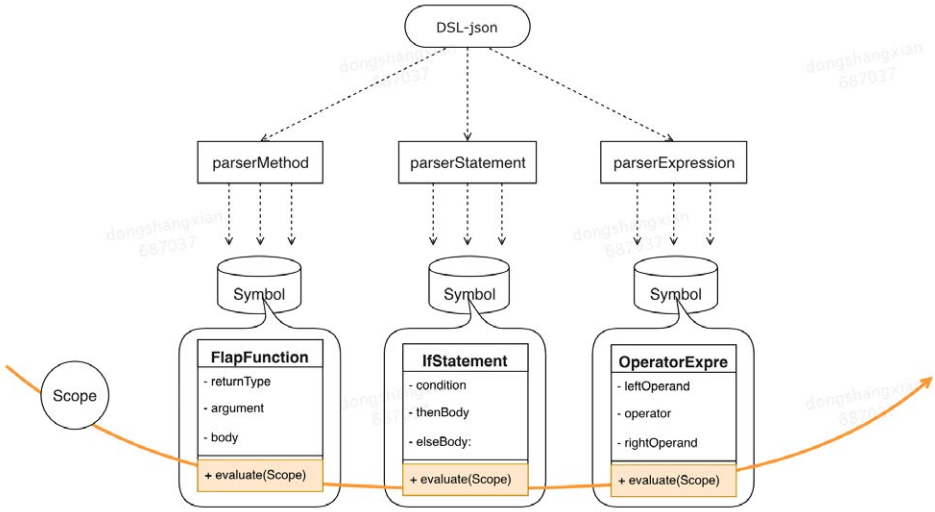


图 9　运行时 evaluate 触发链路

经过表达式的堆叠，实现了语句，经过语句的堆叠实现了 body，再补充上形参和返回值，则就构成了我们运行时中的自定义方法 FlapFunction。这里要用到一下仿真函数的概念，FlapFunction 要实现 call 方法，这样在外部调用时就真的和 Function 画风一致了。

动态化页面运行时，Flap 会维持一套作用域体系。Scope 的结构相当于双向链表，每一个 Scope 有 outer 和 inner 两个指针。全局作用域的 outer 为 null，inner 为类作用域；类作用域的 inner 为局部作用域；局部作用域的 inner 可能为 null 也可能又是一个局部作用域；随便哪一个作用域顺着 outer 一直往上找，肯定能找到全局作用域。

Scope

Scope 在逻辑的执行中实际就是充当了 Context 上下文的作用，因为每个方法或

表达式被 evalute 时需要一个 Scope 入参，这个 Scope 是从外部传入的，并且这一行语句对象执行后 Scope 还会作为入参传给下一行语句。比如第一行语句声明了一个“code”的变量，第二行语句对这个“code”进行修改，则需要先通过引用从 Scope 中取出这个“code”的值，不但可以从 Scope 中取出声明的属性，也可以取出声明过的方法，方法内也是可以调用方法的。这也就解释了为什么我们可以处理自定义方法中的逻辑。

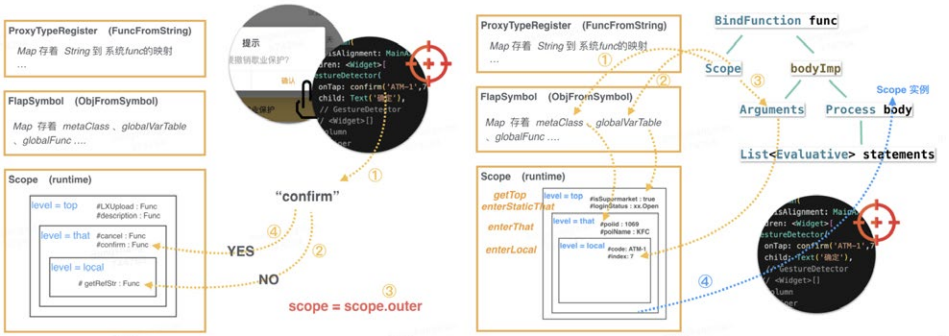


图 10　Scope 的寻找与构建

图 10 描述了 Scope 在实际运用中的两种场景。左半部分是点击按钮触发 onTap 回调，需要找到 confirm 方法，此时会先从局部作用域的方法列表里找，没找到，则会 outer 一层去类作用域里寻找，此时找到了该方法的实现。

右半部分展示了执行该方法的 body 时是需要传入的 Scope 是如何构建的。先从符号大本营中获取全局变量、全局属性构成全局作用域，再从此类的元类中取出属性和方法构成类作用域，再构建局部作用域，当然参数也是会放到局部作用域里的，以此构建了完整的 Scope 传入 body 的 evaluate 方法支撑后面的逻辑执行。

3.3 遇到的挑战

工作量大，需要长期有耐心

首先解释下，这里的工作量大并不是指系统方法映射等这种体力活的工作量大，这些我们都是有自动生成且按需生成的（生态部分会提到）。我们所说的工作量大，主要

是指涵盖转换器、运行时的研发以及生态相关建设等，我们要尽可能的满足所有的 Dart 语法才能让业务代码能够低成本的转换，并且有众多的脚本与工具支撑。

项目复杂，需要设计合理的架构以支撑扩展

在项目的分模块开发中，各个模块（parser、intermediate、runtime 等等）严格遵守单一职责原则与最小知道原则，最大化的杜绝了模块间耦合，模块与模块的通信由一些标准的数据结构进行（map 或继承自 ASTNode 的结构）。 这就使得任何一个模块出现重大重构时不会影响到其他模块，其中底层核心的几个类的单侧覆盖率接近 100%，有专人负责优化。并且在项目中随处可以抽象类、接口类、mixin 类等，这也就使得随着支持的能力越来越复杂时，项目的可读性不会成反比，代码不会变“恶心”，而是以整齐的方式扩张，文件多而不乱。

疑难杂症较多，对问题保持足够的信心

有时候会遇到一些诸如静态方法调用构造方法时作用域被覆盖、循环语句嵌套时内侧 continue 之后外侧语句也会跟着停、某方法参数的 Function 取完引用之后 Function 也跟着执行了等等的 Bug，解 Bug 是开发中必不可少的一部分，有时候加个 if else 用 easy way 可以很快解决，但我们不会那么做，探索优雅 Right Way 的乐趣是研发过程中的一个重要组成部分。

相比于草草了事之后，每晚睡前都会面临这段代码“灵魂”拷问，我们更愿意多花时间思考把代码写的像 Mac pro 主机的包装那样“丝滑”。这样的工作氛围培养了每位同学的信心，只要是必现问题，基本都能优雅地解决。

四、生态支撑

虽然 Flap 的设计理念使得其在开发效率与执行效率上有一定的亮点，但这还不足以让其在业务中快速推广。因此我们建设了一套完整的 Flap 生态体系，涵盖了开发、发布、测试、运维各阶段。

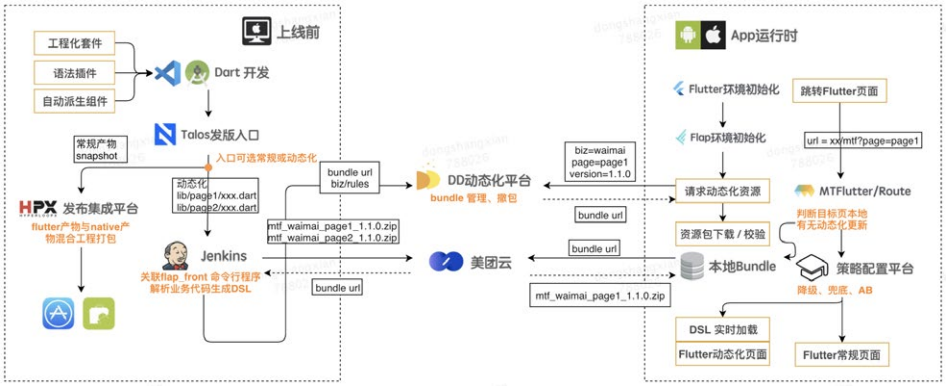


图 11　Flap 在美团内网生态

如图 11 所示，Flap 生态的特点可以用 稳、快、准、狠 四个字来表达。

4.1 稳

稳，意为可靠的质量管理体系。在① IDE 开发中②提测阶段③线上监控④降级容灾，我们都有对应的策略。其中②和③的基本是和 Native 类似的 PR 检查、QA、日志、上报之类的这里就不做赘述了，下面主要提一下①和④。

IDE 语法检测插件

这个功能的意义是尽早地将不支持的语法以编译错误的方式暴露出来，以便同学在开发期就能发现及时修改。 设想一下当你代码写完了，Code Review 也逃过了同学的眼睛，PR 的 Dart 检测也过了，开开心心下班了，突然一个电话打来说发 Bundle 的时候错了，有的语法 Flap 不支持，需要返工去改，此时你的内心一定会“万马奔腾”。

所以，我们将这种暂不支持的语法提前暴露，并推荐使用什么方式代替，可以有效的减少返工，得到一份满足 Dart 规范和 Flap 规范的代码。同样的 Lint 检测规则后续也配置到了 PR 阶段，如果真出现插件规则更新不及时场景，也会被拦在 PR 阶段。

图 12　IDE 语法检测插件

不过，目前 Flap 不支持的语法已经很少了，目前基本就是 await、as 和超过 2 个 with 等场景，其中 await 和多个 with 的理论上也能支持，但会让项目有较大的重构和多处的分别对待，不利于后期的维护，考虑到 await 完全可以使用 future.then 代替，所以这个语法就禁了。对于 mixin 的特性，在 Dart 侧本身就是排列组合的关系。超过 2 个 with 会产生多个派生类，动态化的实现类似，所以为了不让简单问题复杂化，我们也禁用了 2 个以上 with 的写法，还有一些写法上的限制，例如 import 不使用全路径也会报错。

目前开发中 Flap 动态化已经与 AOT 共用一份业务代码了，为了不让 Flap 的规则影响到项目中还未覆盖到动态化的页面，让其满屏报错，我们使用 @Flap 注解作为是否开启当前页面的 Flap 规范检测的开关。这也很好理解，当这个页面内没有 @Flap 时，肯定是个 AOT 模块则还是默认的 Dart 检测规则，一旦加上了 @Flap(‘pageID’)，说明此页面会被动态发版，所以会自动开启 Flap 检测规则。

降级容灾

Flap 接入了美团内部统一的动态化发布平台 DD，并利用 DD 平台的能力实现了 App 版本、平台类型、UUID、Flutter SDK 版本等细粒度的下发规则管控。业务方可以根据实际情况选择不同的策略灰度发布方案，如果发生了严重异常，Flap 也支持撤包操作。

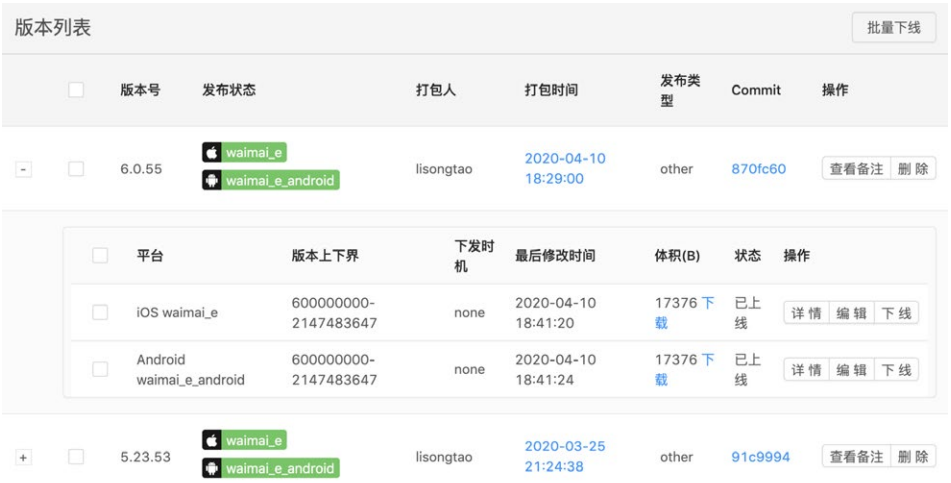


图 13　Bundle 发布系统的各项边界控制

某一个页面加了标记支持了动态化之后，也会继续进行 AOT 编译过渡 2 个版本，前置页面点击跳转是跳 AOT 页还是跳 Flap 页完全由 URL 里的参数控制，这个 URL 不是完全由云端下发的，是代码中先写上默认的 URL，若需要在配置平台修改后，下发的配置信息会让这个 URL 在路由侧完成替换。即使配置平台挂了，顶多丧失 URL 的替换能力而不是无法前往落地页。

图 14　URL 动态替换与条件配置

对于 Flap 还有个更犀利的功能，在过渡期间（Flap 已经上线且 AOT 代码还没删时），一旦 Flap 出现 Dart 异常，当用户退出页面再进入时会自行进入该 pageID 下的 Flutter AOT 页面，最大化降低对用户的干扰。

4.2 快

快，意为快速发版，快速更新。Flap 动态化改造使应用具备了分钟级动态发版的能力，为了更全面地释放这个能力，客户端业务迭代的流程也做了相应的调整。

当业务包发版上线，到了应用运行阶段，Flap 主要面对的问题变成敏捷与质量的平衡，即：如何保证动态代码能够尽快生效，同时又要保证加载性能和稳定性。

对于此问题，Flap 的解法是二级缓存与实时更新相结合，线上环境使用内存 + 磁盘二级缓存，进入页面之后再预拉取更新包，平衡加载性能与更新实时性。而线下环境则强制加载远程包，实现测试代码的快速交付。

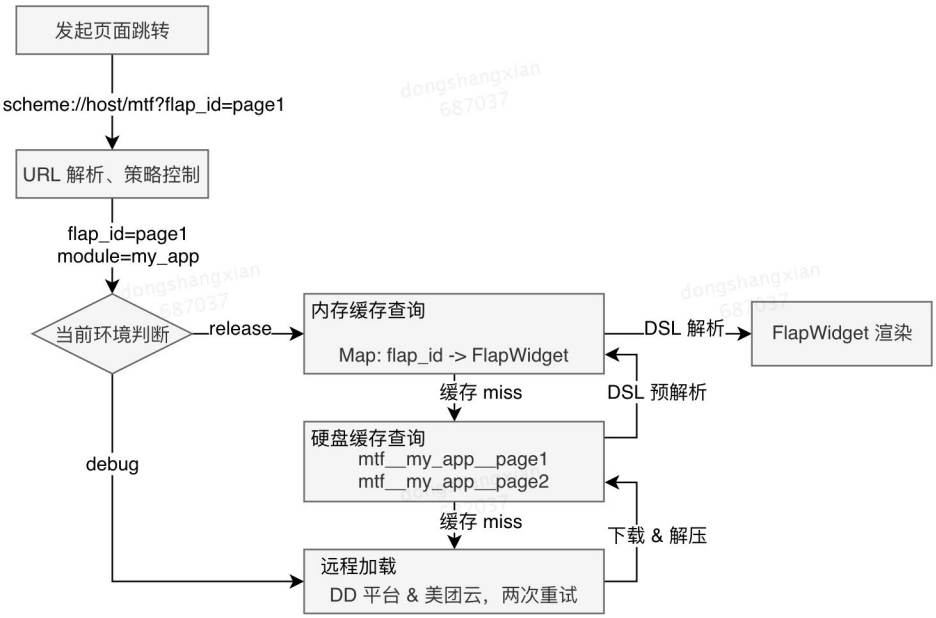


图 15　Flap 二级缓存策略

得益于这种机制，Flap 在线上可以实现接近 Web 的触达效率：应用会在启动时和具体业务入口处发起更新请求，每当业务有动态发布，新版本页面即可在用户下一次打开时触达至用户。在加载性能方面，二级缓存加持下的页面加载时间仅为数十毫秒，而远程加载的时间也只有 1 秒左右。

4.3 准

细粒度动态化

准，指哪打哪，可以页面级动态化，也可以局部 Widget 级别的细粒度动态化。事实上在 Flutter 的世界中，“页面”本身也是一个 Widget，业务方在实际开发中，只需要增加一行注解，即可实现对应 Widget 或页面的动态化。

```
@Flap('close_protect')
class CloseProtectWidget extends StatelessWidget {
  // ...Widget 的 UI 和逻辑实现
}
```

Flap 打包发版时，解析引擎会从注解标记的 Widget 入手，递归解析所有依赖的文件，转化成对应的 DSL 并打包。App 线上运行时，每个动态化的页面或组件都会按照注解的 FlapId，通过 FlapWidgetContainer 还原成对应的 UI。

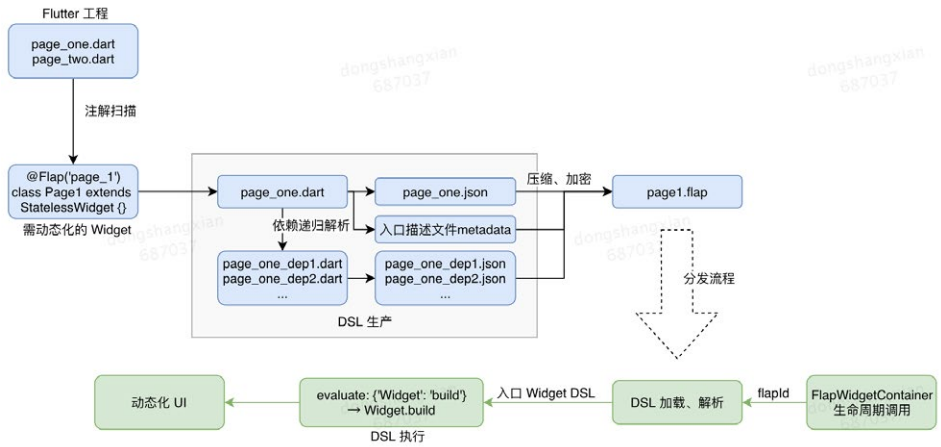


图 16　注解的扫描与 widget 构建

实际调用时，只需传入注解中标记的 FlapId，即可实现动态化区域或页面的加载和渲染。

```
// 局部 Widget 级别的动态化，通过 FlapWidgetContainer 加载
Column(
  children: <Widget>[
    MyAOTWidget(),  // 原生 Flutter AOT Widget
    FlapWidgetContainer(widgetId: 'kangaroo_card'), // Flap widget
  ],
);

// 页面级别的动态化，通过 MTFlutterRoute 路由跳转：
RouteUtils.open('scheme://host/mtf?mtf_page=flap&flap_id=close_
protect');
```

精准的 Debug 能力

在 Debug 阶段加上一个注解 @Flap（‘pageId’），就会自动尝试转 DSL。如果该页面非常独立，且语法没有太花哨，则直接就能看到转换完成的字样。这个就说明该页面用到的语法既支持 Dart 又支持 Flap，不需要做任何修改。如果出现错误，则会在终端下精准打印出错误的位置。在此功能支持之前，基本都是“一崩就崩”到系统类的某某方法，开发同学只能通过自己的经验去堆栈中往上找。目前的精准 Debug 能力实现了转换器、运行时 parser、运行时 evaluate 三个阶段的全面覆盖。

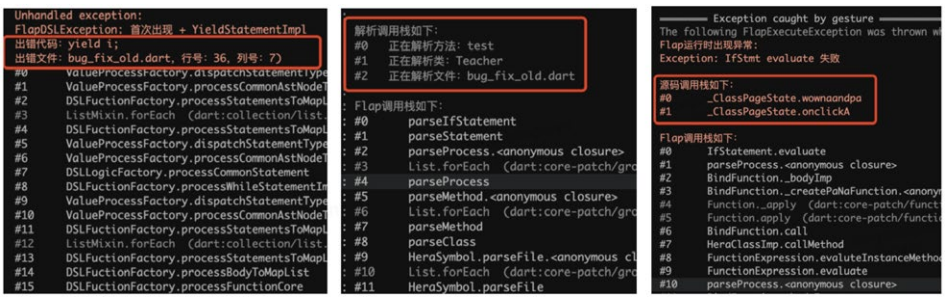


图 17　三个阶段的 Debug 定位

在转换器阶段的报错位置信息可直接在 Exception 中获得 AST 对象的 lineinfo 进而获取到列号行号信息。在 parser 与 evaluate 阶段的错误定位是根据对核心方法的

trycatch 与设置通用 Exception 类型逐层上抛实现的。因为 DSL-JSON 会被压缩且可以 format，行号列号并无意义，所以在运行时阶段的报错全是精确到某 class 中的某 method。

4.4 狠

狠，各种自动生成，实际转换步骤操作方式简单粗暴。Flap 在整个迭代流程环节都提供了便捷的自动化工具支撑。

imports 自动加载

基于 Flap 转换一个旧的 Flutter AOT 页面到 Flap 页面的操作是简单粗暴的，加上注解，一行终端指令就可以一把“梭”。但一个业务页面为了设计上的合理往往会分成多个文件，如果有 10 个文件是不是要重复 10 遍这样的工作？答案是否定的。Flap 无论是在 DSL 转换器侧，还是在运行时加载 DSL，都会做到 imports 的递归加载。

IDE 语言检测插件有一条限制是：import 必须使用 package 全路径，不能只 import 一个类名。因为多文件需要导入的位置都是根据全路径截取出的相对路径来计算的。

Proxy-mirror 按需生成

前面介绍过 Proxy-Mirror 是外部符号转内部符号的桥梁，那么具体 Dart 文件中哪些用到的类或方法需要内置 Proxy，而哪些类不需要呢？这个划分的边界就是，在转换的代码内能否看到此类或方法的声明。系统方法的声明肯定不在业务文件里，所以需要 Proxy。业务 Model 的声明在“我的业务”文件中有，所以不需要 Proxy。代码中使用到了官方 Pub 或是其他业务线的 Pub，例如美团金融的 Pub 里的方法，声明不在“我的业务”文件里，所以需要 Proxy。

在 Flutter AOT 迁移动态化初期，经常需要手动干预的问题是：项目中遇到 Proxy-Mirror 缺失会打断转换器，需要手动补充后继续进行转换。

对于这种问题后期研发 Proxy 自动生成按需生成的工具，主要原理是在预转换阶

段，先扫描代码的 AST Tree，压平层级获取所有的项目结构中 identifer 节点包裹的 Value，进行一系列判定规则，然后基于 reflectable 功能实现 Proxy 的自动生成。

发布链路“一条龙”服务

经过不断的提炼与简化，目前开发者大可以将注意力集中在开发阶段，一旦代码合入主干，接下来就会有完整的 Flap 工程化发布和托管系统协助开发者完成后续的打包、发布、运维流程。前面介绍过的所有细节工作，都会由这些工具自动化完成，实现便捷发布。Flap 也在路由层面对接了集团内通用的运维工具，开发者无须任何额外操作即可实现加载时间、FPS、异常率等基础指标的监控。对于指标波动、异常升高等情况，也会自动注册报警项并关联至当前的打包人。

五、业务实践经验

业务落地只是我们的目标之一，更重要的是在业务实践过程中，发现框架问题，完善各类语法特性支持，提高在复杂的混合场景下的兼容性，反哺促进框架的完善。不断打磨的同时完善工作流，思考与沉淀最佳实践，逐渐总结出合理的调试方案、操作步骤与协作方式，不断提升开发效率与体验。完善动态化基建及工具链建设，完成动态化流程的自动化与工程化，进一步降低转换与开发成本。

5.1 应用场景

对于 Flap 在业务中的实践，主要有两种应用场景。

场景 1. 原有 Flutter 页面，需要转换成动态化页面

设想一下，理想状态下一个好的动态化框架应该是怎样的？动态化框架将原有 Flutter 改写成支持动态化的页面？那加一个 @Flap 注解就好了。然后就可以提交代码，自动走工具链那一套。

目前，虽然没有达到理想状态，但我们也在无限接近中，当然还是要简单地本地调试

一下。基本都需要改个 URL 路由和 Mock 环境之类的步骤，我们已经提供了模板的调试工程，支持一键对比 AOT 与动态化运行之后的差异，如图 18 所示。基本就是加上注解，IDE 插件会报错哪些语法不支持，需要换一种写法，然后跑一下就可以，然后就提交代码。

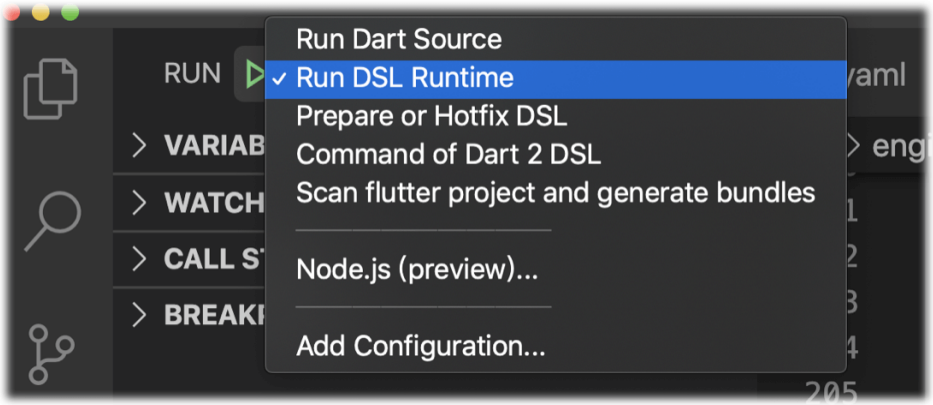


图 18　研发过程支持不同的运行模式

场景 2. 直接使用 Flap 技术栈开发新页面

重新开发场景很明显比第一种要简单，因为没有历史包袱。设想一下，好的动态化框架应该怎么做？就是和 Flutter 的 AOT 开发使用一套相同的 IDE 环境，相同的开发模式，就是 IDE 会多报几项语法错误罢了，开发时就能直接被提示到换一种写法就行。写完后加上注解，然后再提交代码。

5.2 实践经验

目前，我们团队已经把 Flutter 动态化能力在一些业务场景落地，当然业界也会有相似的或者不同的动态化方案。无论方案本身怎样，在落地时的步骤基本都大同小异，我们也总结了一些经验。

绕过问题并加以记录

初期任何框架的能力都不是完美的，都会存在问题。业务方同学遇到 Proxy 类缺失

之类等比较简单的问题可以直接解决，运行时环境的深层问题、某些语法在复杂叠加场景下出现异常等等，一般会先尝试用其他的语法绕过，记录文档，然后同步到 Flap 团队同学进行解决。

定时补充 IDE Plugin Rules

对明确不支持的语法、关键字等添加到 IDE Plugin Rules 中，并提供了相关语法的替代方案，Rules 也会定时补充和删减。

提前周知各方资源

包括确认好 Android 的上线节奏，QA 的测试节奏，以及周知 PM 动态化的覆盖占比。

关键权限收紧管理

相比于整理权限、灰度、降级、容灾等线上的 SOP 和 FAQ，让大家都学着操作，直接指定 2~3 位超级管理员看上去更靠谱，线上环境由“老司机”把控更好。

5.3 落地结果

业务应用涵盖 App 一级页在内的多个页面，场景既有页面动态化，也有局部动态化，经受住了一级、二级页面的流量验证。

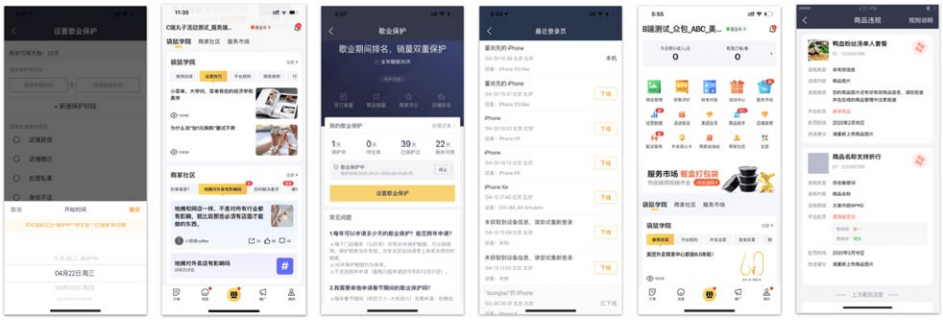


图 19　部分动态化落地页面

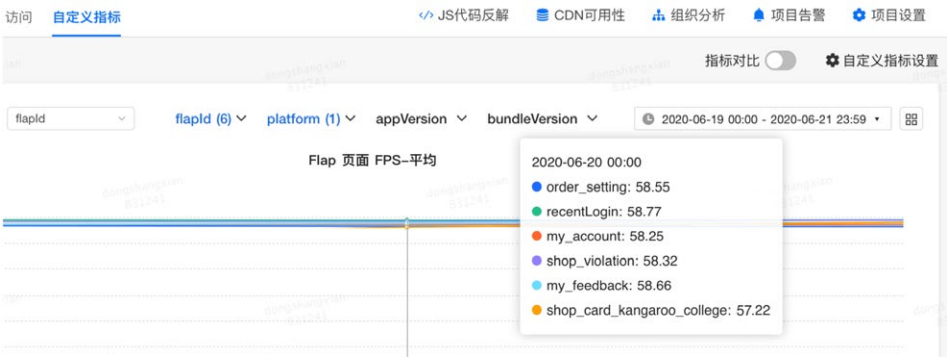


图 20　部分动态化页面 FPS 数据

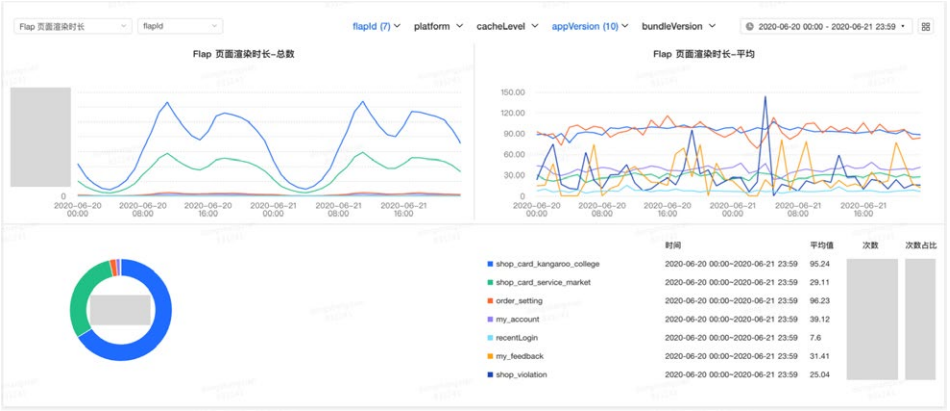


图 21　部分动态化页面渲染时长

图 21 涉及到 PV 的地方打了马赛克，Flap 团队对包括 FPS、加载时间、Bundle 下载时长、渲染时长等 11 项指标进行了统计，可以看到 FPS 平均是在 58 以上，渲染时长根据页面复杂度的不同在 7~96ms 之间。

总的来说，各项指标表现均接近于 Flutter 原生性能。并且图中的数据都还有可提升的空间，目前的平均值也受到了局部较差数值的影响，后续会根据不同的 TP 分位使用分层的优化方案。

六、总结与展望

我们通过静态生产 DSL+Runtime 解释运行的思路，实现了动态下发与解释的逻辑页面一体化的 Flutter 动态化方案，建设了一套 Flap 生态体系，涵盖了开发、发布、测试、运维各阶段。目前 Flap 已在美团多个业务场景落地，大大缩短了需求的发版路径，增强了线上问题修复能力。Flap 的出现让 Flutter 动态化和包大小这两个短板得到了一定程度的弥补，促进了 Flutter 生态的发展。此外，多个技术团队对 Flap 表示出了极大的兴趣，Flap 在更多场景的接入和共建也正在进行中。

未来我们还会进一步完善复杂语法支持能力和生态建设，降低开发和转换 Flap 的成本，提升开发体验，争取覆盖更多业务场景，积极探索与业务方共建。然后基于大前端融合，探索打通其他技术栈，基于 Flap DSL 抹平终端差异的可能。

参考文献

[1] Gilad Bracha. The Dart Programming Language [M]. Addison-Wesley Professional, 2015
[2] Code Push/Hot Update
[3] Analyzer 0.39.10
[4] Extension API
[5] Flutter 核心技术与实战
[6] App Store Review Guidelines

作者简介

尚先，2015 年加入美团，到家研发平台前端技术专家。
杨超，2016 年加入美团，到家研发平台前端资深工程师。
松涛，2018 年加入美团，到家研发平台前端资深工程师。

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，欢迎加入外卖 App 大家庭。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：美团外卖技术团队）

美团开源 Logan Web：前端日志在 Web 端的实现

作者：孙懿

1. 前言

Logan 是美团点评推出的大前端日志系统，支持多端环境运行，可为客户端、Web、小程序等用户端环境提供前端日志的存储、收集、上报及分析能力，能够帮助开发人员快速定位并解决端上问题，便于及时响应用户反馈与排除异常。

2018 年 10 月，Logan 在社区开源了 Android 与 iOS 端的 SDK，实现了在客户端进行日志存储及上报代码的功能，引起用户端相关开发者的广泛关注。详细可参见博客文章：《Logan：美团点评的开源移动端基础日志库》。

2019 年 12 月 12 日，Logan 开源了在 Web 环境运行的 SDK、日志分析平台以及服务端代码，为开发者们提供了 Logan 大前端日志系统的一整套实现方案，进一步解决了多端环境中日志的存储与采集问题。

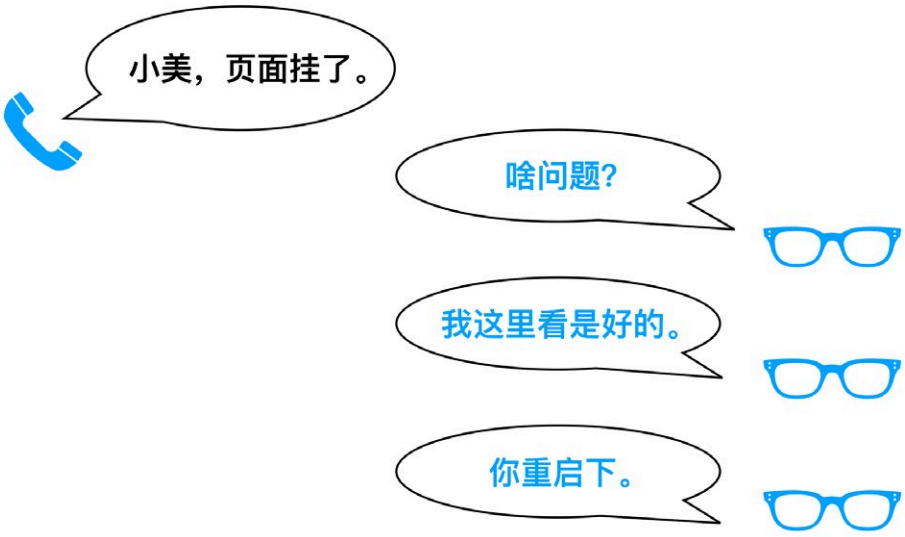
本文将围绕 Logan 在 Web 端的应用背景、技术实现、美团点评的实践、开源整体进展以及未来规划这几个方面展开介绍，以方便读者对 Logan 大前端日志系统有更加深入的了解。

Logan 项目地址：https://github.com/Meituan-Dianping/Logan。

2. 背景

2.1 为何需要 Logan？

在 Logan 诞生前，用户端开发者在面对用户反馈页面功能异常时，最常说的灵魂三答是：



这三条回答分别对应着开发者在解决端上问题时的心路历程：

- **“啥问题”**：通过与用户沟通，获取异常发生前后过程的详细描述，尝试在开发者本地模拟，以期复现问题。
- **“我这里看是好的”**：问题没有复现，应该是用户端兼容性的 bug。
- **“你重启下”**：想不出修复的办法，只能“死马当活马医”，碰碰运气。

对用户端的开发者来说，本地无法复现的问题好比“断了线的风筝”，让人无计可施。如果有办法获取到事发时完整的日志流以及用户环境的上下文信息，就能够帮助开发者快速了解并还原问题的发生现场，可以更有效地定位排查问题，让风筝最终被拉回到开发者手里。正是因为这样的迫切需要，Logan 大前端日志系统才应运而生。

2.2 Logan 日志系统的策略与核心

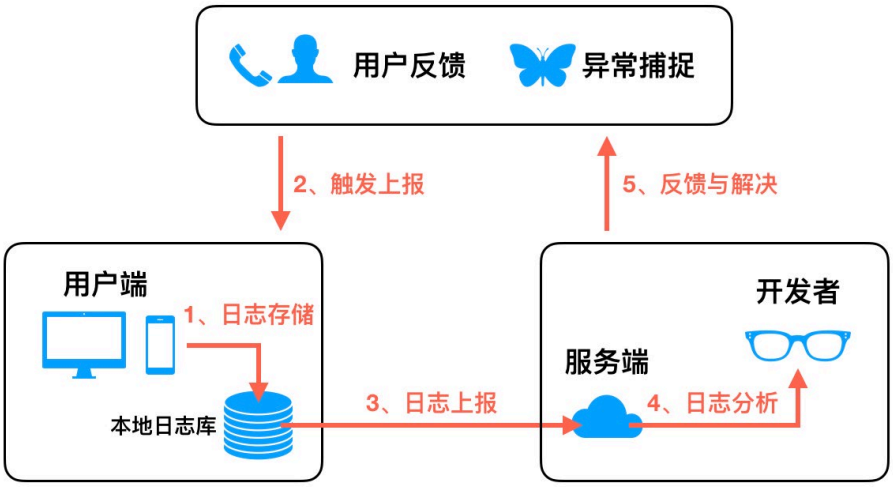
2.2.1 Logan 在各端实行的通用策略

虽然用户端上完整的日志流及上下文环境信息更有助于开发者定位到问题，但对每个

用户端的日志流都进行实时上报的话，也会出现如下问题：

- **巨大开销**：耗费用户流量，占用企业带宽与服务器存储资源。
- **大海捞针**：在海量日志中可能只有极少部分的日志能够帮助复现问题。

因此 Logan 在各端上实行的通用策略，是本地日志存储结合触发上报的模式，如流程图所示：



a. 平时在用户端脚本执行过程中产生的日志会落地到本地的存储容器中。

b. 当遇到用户反馈或者端上异常被捕获时，Logan 以特定机制触发本地日志的上报。

c. 本地日志流将在用户端上传，由服务端收集并解析，最后上传至云端存储。

d. 由 Logan 统一的日志分析平台向开发者提供日志数据的可视化展示。

e. 开发者利用 Logan 日志排查定位并解决问题后，向用户反馈或者排除异常。

2.2.2 Logan 的三大核心

上文所阐述的 Logan 通用策略中的工作流程也决定了 Logan 日志系统拥有三大核心：

- **用户端 SDK（客户端版、Web 版及小程序版）**：负责存储与上报端上日志。
- **服务器端**：负责接收、解析、整合与分析日志。
- **日志分析平台**：提供日志的查询与数据可视化展示。

2.3 Logan 在 Web 端面临的问题

在 Web 环境中若要实现端上日志存储及上报需要解决三大难点：

- **如何存储？** 需要解决 Web 本地大体积日志流的存取。
- **如何保障日志安全？** 在本地已存储的日志需要有数据安全保障。
- **如何上报？** 需要有效的机制触发日志的上报。

2.4 Logan Web 做了什么？

Logan Web 是 Logan 在 Web 端的存储及上报实现方案，利用现有的前端技术加以优化与整合，有效地解决了 Web 端面临的三大难点。我们将存储与上报的实现封装在 Web 端的 SDK 内，开发者只需在页面脚本中引入该 SDK，便可直接使用 Logan 在 Web 端上的日志安全存储与上报能力。

下面将重点围绕存储方案、数据安全及上报机制这三点，具体阐述 Logan Web 目前的技术实现。

3. 技术实现

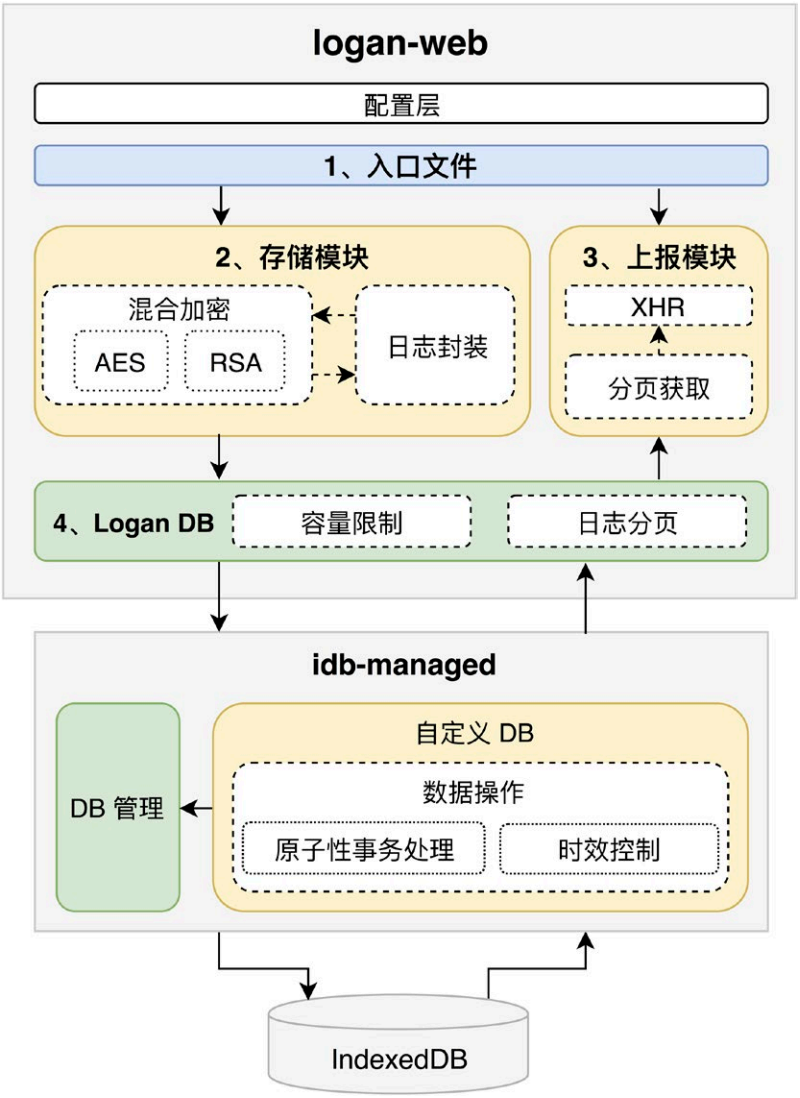
Logan Web 在底层利用了现有前端技术来实现大体积日志的安全存储：

- **存储方面**：利用浏览器的 IndexedDB 作为本地日志的大容量存储容器。
- **日志安全方面**：使用混合加密模式确保本地已存的隐私日志数据不会被破解。

Logan 为 Web 开发者提供了 logan-web 这个 SDK 包，其内部主要分为存储与上报两大核心模块，底层依赖了 IndexedDB 存储与加密组件。开发者可在页面脚本中引入并加载该 SDK 来调用日志存储或上报接口。

3.1 Logan Web 整体技术架构

以下是 Logan Web 的整体架构示意图：



a. logan-web 提供了一个入口文件，它将在日志存储方法或者日志上报方法被触发时，异步地获取存储或上报模块。

b. 存储模块中会优先处理日志内容的加密及包装，再执行后续的分页存储流程。

c. 上报模块会分页读取指定天的日志数据，并行上报至接收日志的服务端，进行后续的日志解析、解密、整合及分析。

d. 这两大核心模块都会使用 Logan DB 模块封装的日志存取逻辑，该模块还会控制本地日志数据的存储容量以及日志分页。

对 IndexedDB API 的调用被封装在独立于 logan-web 的 idb-managed 包中，该包主要解决在使用 IndexedDB 进行本地存储时遇到的技术挑战。

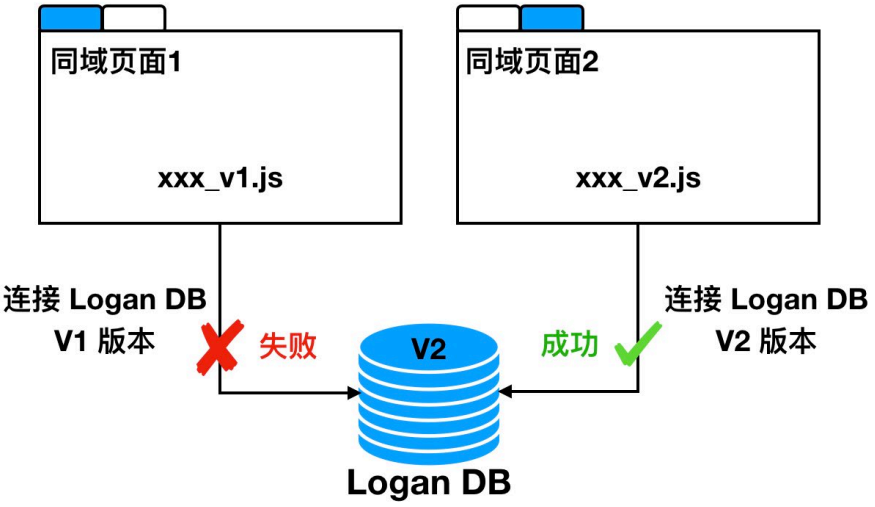
3.2 本地存储方案：idb-managed

3.2.1 原生 IndexedDB API 的局限

IndexedDB 支持大容量存储，并且其读取操作是异步化的，非常适合作为 Logan Web 的本地存储容器。但 IndexedDB API 在使用上会遇到如下几个问题：

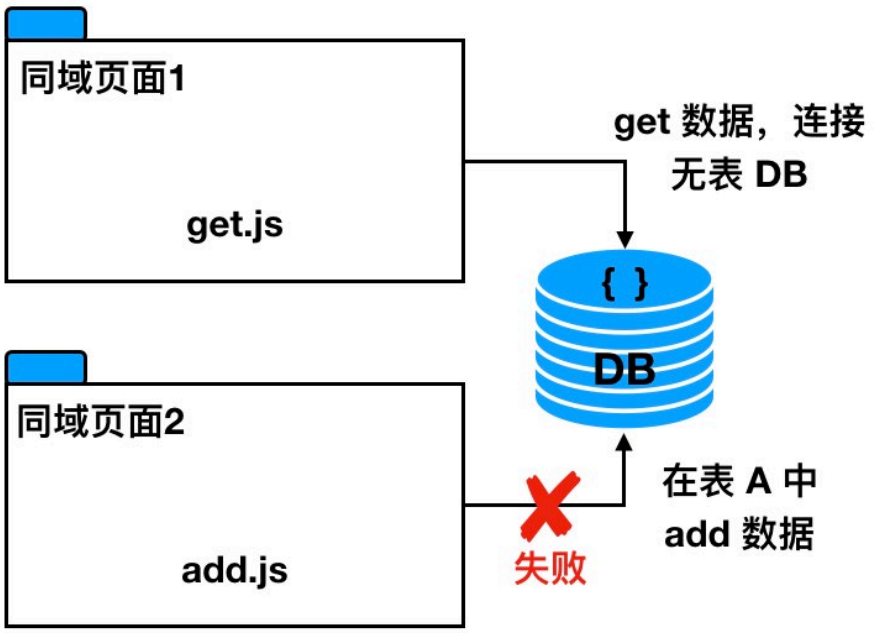
（1）DB 版本升级问题

本地 DB 依靠版本的升级来更新库表结构，当本地该 DB 的版本升级后，尝试连接低版本 DB 的操作将失败。



（2）获取 DB 数据前需要设置 DB 版本及库表信息

获取 DB 数据前需要首先连接 DB，如果连接时没有设置恰当的库表信息，下一次存储时依然以同版本建立连接，IndexedDB 则不再更新该 DB 的库表结构，最终会因为存储数据不匹配表结构而导致存储失败。



（3）IndexedDB 不提供数据的时效设置与过期数据清理

IndexedDB 默认数据是持久化落地的，尽管它的可用容量远大于 LocalStorage，但在像 Logan Web 这样需要随时间不断更新本地数据的使用场景下，就需要一套随时间迭代的数据更新机制来“除旧存新”。

（4）原生 IndexedDB API 不提供多表间的原子性增删操作

原生的 IndexedDB API 只提供了单条数据的添加，以及单表内的数据批量删除操作，并不直接提供 API 对多表的数据进行添加或者删除的原子性操作（要么全部生

效，要么全不生效）。Logan DB 的库表结构涉及到两张表数据间的联动，需要多表间原子性增删来保证日志数据在两张表间保持一致。

3.2.2 idb-managed 的解决手段

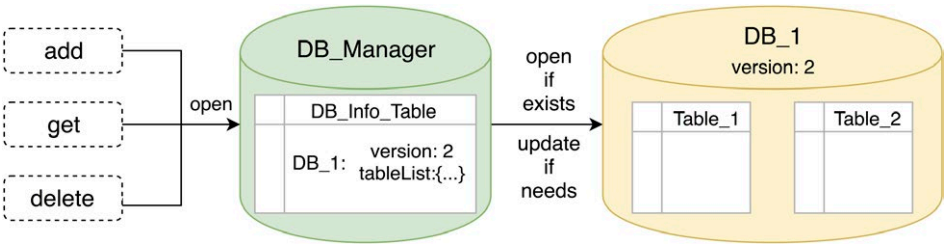
为了统一解决以上在使用 IndexedDB 时面临的困扰，我们额外封装了 idb-managed 包，该包随着 Logan Web 一起开源。

idb-managed 中分别针对：

- 问题 1 与 2：提出并实现了 DB Manager 机制来解决 DB 版本升级与数据获取的困境。
- 问题 3：封装了对存储数据的时效设置、过期处理逻辑。
- 问题 4：利用事务回滚方法实现在多表内的原子性增删操作。

（1）DB Manager 机制

idb-managed 中内建了一个 DB Manager 来管理当前所有本域下的 DB 版本与库表结构信息，其实 DB Manager 自身就是一个版本号固化的 DB，它本身不存在升级问题。建立链接示意图如下：



当对一个新 DB 建立连接时，会将脚本中设置的 DB 版本及库表信息注册到 DB Manager 中并把数据存储下来。下一次如果有该 DB 的低版本尝试连接时，DB Manager 会用当前已注册的库表信息连接并打开 DB，由此避免了 DB 升级而导致连接低版本失败的问题。

同时，因为有 DB Manager 来统一管理本地的 DB 信息，所以从 DB 获取数据可以无需知晓 DB 库表结构，只需要 DB 名和表名即可。当本地库表不存在时，DB Manager 会阻止对该 DB 的连接，直接返回空数据，避免了错误的库表结构污染本地 DB。

（2）数据时效控制

idb-managed 会为每张表建立一个到期时间索引，开发者可对单条数据、单个表或者单个库设置一个持久化时间限制，在数据存储时 idb-managed 会根据这些限制及优先级顺序（单条 > 单表 > 单库）计算该条数据的到期时间，并与数据一起保存下来，过期的数据会在该表下一次添加数据时先行删除。

（3）多表间原子性增删操作

IndexedDB 中数据的增删操作全部建立在事务（IDBTransaction）基础之上，idb-managed 封装了多表批量数据的增删操作接口，并将这些操作包裹在一次事务内。如果在一次事务中发生异常，idb-managed 将执行本次事务的回滚，从而保证这批操作具有原子性。

3.3 日志加密方案：混合加密

混合加密方式吸取了对称加密与非对称加密各自的优势：

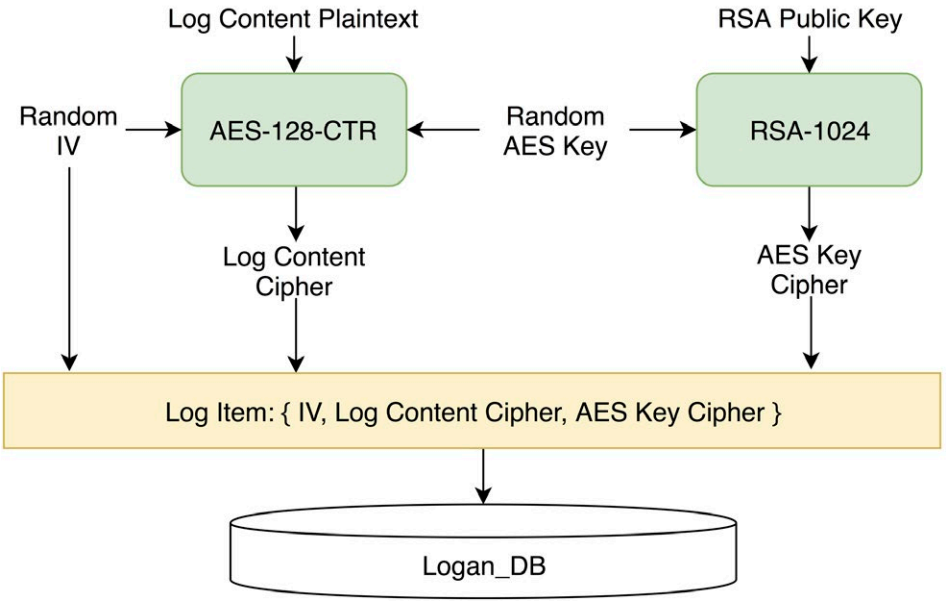
- 对称加密：保证对长内容加密的效率。
- 非对称加密：保证对称密钥的安全性。

Logan Web 选择了 AES-128-CTR 结合 RSA-1024 的混合加密模式。在存储每条具有私密性的日志前都会经历以下加密流程：

a. 准备对称密钥与初始向量：随机产生 AES 对称密钥 AES Key 及初始向量 IV。
b. 对称加密：使用 AES Key 及 IV 对日志明文进行 AES-CTR 对称加密，得到日志密文。
c. 非对称加密 AES Key：使用 RSA 公钥对 AES Key 进行非对称加密，得到 AES Key 密文。该 RSA 公钥与服务器端的私钥是成套的，只有该私钥可以

解开该 AES Key 密文，从而解开日志密文。

d. 包装数据：将以上初始向量、日志密文与 AES Key 密文包装成一条日志对象，随后存储落地。



3.4 上报的触发机制

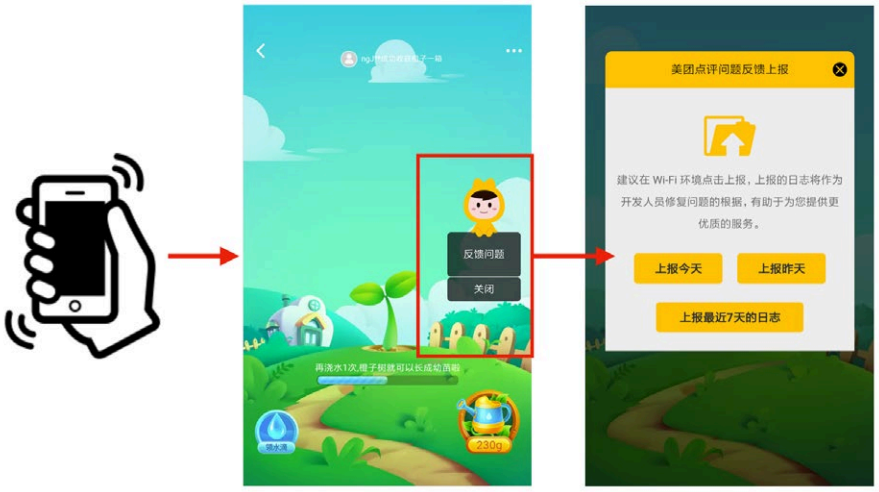
用户端的日志上报触发机制一般分为两大类：

- **用户主动触发**。优点是上报的日志能够对应到用户反馈的个案；缺点是存在交互上的用户教育成本，同时依赖用户反馈的异常处理流程，过于滞后。
- **代码层面触发**。优点是用户无需感知上报流程；缺点是可能存在大量“无帮助”的上报日志，需要对触发条件做好频率控制。

Logan Web 在两类方式上提供了配置与接口，供 SDK 使用方自行选择与扩展，例如：

（1）用户主动触发

- **PC 端**：Logan Web 在美团点评内实践时，为业务方提供了 DOM 元素配置项，用于 Logan Web 绑定触发上报的钩子函数。
- **H5 手机端**：Logan Web 可扩展内置设备摇动检测，利用浏览器的 DeviceMotionEvent 事件监听，当用户“摇一摇”时，引导上报流程的美团卡通形象会出现在页面侧栏，如下图所示。
- **通用**：Logan Web 在美团点评内扩展提供了接口供业务方显示或者隐藏引导上报流程的侧栏。



（2）代码层面触发

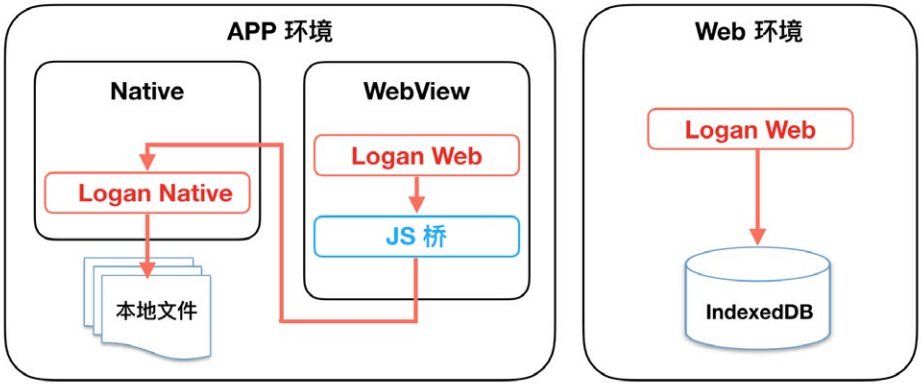
考虑到用户主动触发方式存在的弊端，Logan Web 提供了上报接口供使用者在代码层面调用。另外 Logan Web 也正在迭代实现内部的代码触发上报逻辑，提供在 Web 端内异常发生等特定场景下主动上报日志的能力。

4. Logan Web 在美团点评内的实践

Logan Web 已在美团内部上线，并持续迭代了一年多的时间，覆盖公司很多主要的

业务线。截止 2019 年 11 月，公司内使用 Logan Web 的页面 PV 量已达日均 1.5 亿余次，Web 上报日志量达日均 500 余次，约 97% 上报的 Web 日志在 Logan 日志分析平台上被公司研发同学搜索查阅。利用 Logan 日志系统，公司各研发团队已能够尽早获得完整的用户端日志流及环境信息，帮助业务及时排查问题并响应用户反馈。

另外，在美团，Logan Web 除了提供上文介绍的技术实现之外，还利用美团内部的通用 JS 桥组件实现了与 Logan 客户端日志的打通，如下图流程所示：



这意味着在美团现有的 App 环境（如美团 App、点评 App 等）中运行的 H5 页面如果使用了 Logan Web，所记录的日志会利用 JS 桥传给 Logan 客户端，与客户端日志一起落地在 App 本地文件中。因此在美团的 App 环境内上报的日志流中可查看上下文连续的 Web 端日志与客户端日志，日志分析平台展示的某篇日志详情示意图如下：

5. Logan 开源进展与未来规划

在 Logan Web 开源前，我们在美团 Logan 开源技术交流群中进行了开发者需求调研，依据收集到的建议，Logan Web 这次开源版本将支持 TypeScript，同时提供了本地日志的加密策略选择。开源的代码及使用文档可在 Meituan-Dianping/Logan/WebSDK 仓库下查阅，开发者也可以在项目中直接通过 npm 包引入的方式引入 logan-web。同时 Logan Web 底层依赖的 idb-managed 也已在 GitHub 与 npm 仓库开源。

随着本次 Logan Web 同时开源的还有 Logan 服务端与 Logan 日志分析平台的实现，读者可一并在 Logan 代码仓库 下找到相应的代码和使用文档。客户端 SDK 的实现博客可点击参考：《美团点评移动端基础日志库——Logan》。

模块	已开源	规划中
iOS 端 SDK	✓（2018年10月）	
Android 端 SDK	✓（2018年10月）	
Web 端 SDK	✓（2019年12月）	
服务端	✓（2019年12月）	
日志分析平台	✓（2019年12月）	
动态化框架适配		✓（2020年Q1）
小程序 SDK		✓（2020年Q3）

目前 Logan 已开源了客户端 SDK、Web 端 SDK、服务端及日志分析平台，已经为社区开发者们提供了初步完善的整套 Logan 日志系统实现。在未来我们将继续优化扩展 Logan 能力，帮助开发者们在各端环境中，都能更快更早更方便地定位问题及排除异常。

6. 联系我们

本次开源只是 Logan 贡献社区的一小步，我们希望在未来 Logan 能够为社区提供更完整可靠的大前端日志服务，我们诚挚地欢迎开发者向我们提出宝贵的建议，与我们共建社区。您可以挑选以下方式向我们反馈建议和问题：

- 在 github.com/Meituan-Dianping/Logan 提交 PR 或者 Issue。
- 微信添加 MTDPtech03，该美团助手可邀请您加入美团 Logan 开源技术交流微信群，或者回复您的建议。
- 邮件发送至 edp.bfe.opensource@meituan.com。

7. 作者简介

孙懿，美团点评基础技术部前端技术中心资深工程师。

8. 招聘信息

美团点评基础技术部前端技术中心负责美团云平台运维领域的前端基础研发工作，包含前端监控、日志系统、长连通信、运维工具等基础建设。欢迎各路英雄扫码投递简历，我们满心期待您的加入。感兴趣的同学可将简历投递至：tech@meituan.com（邮件标题注明：基础技术部前端技术中心）。

外卖客户端容器化架构的演进

作者：郭赛　同同　徐宏

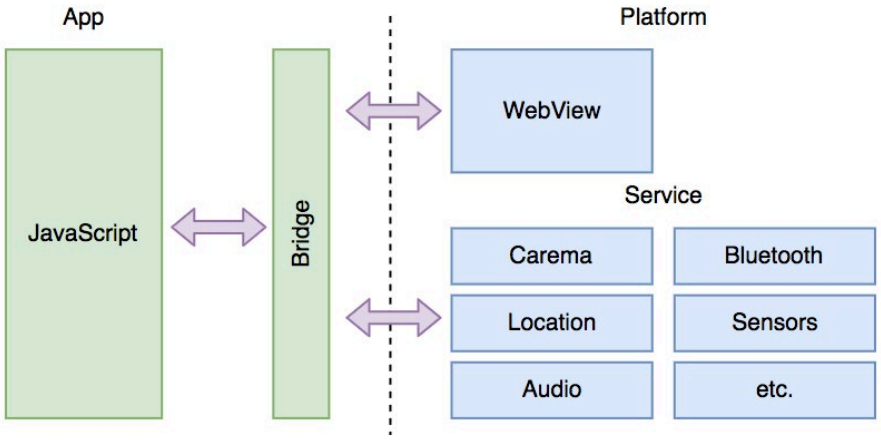
1. 背景

1.1 移动端跨平台技术的介绍

移动端的跨平台技术不是一个新话题，早在几年前，WebView 容器、React Native、Weex、Flutter、小程序等移动端跨平台框架就风起云涌。为什么跨平台这么有吸引力呢？我们设想一下如果可以做到一次开发，多端复用，那么对于公司来说，就可以降低用人成本。对于开发来说，只需要学习一个框架，就可以在 Android 和 iOS 双平台上开发。节约下来的成本，可以投入到产品快速验证、快速上线。这对所有人来说都有着极大的吸引力。本节先针对部分移动端跨平台技术进行一些简要的介绍，以便读者能够更好地理解后面的内容。

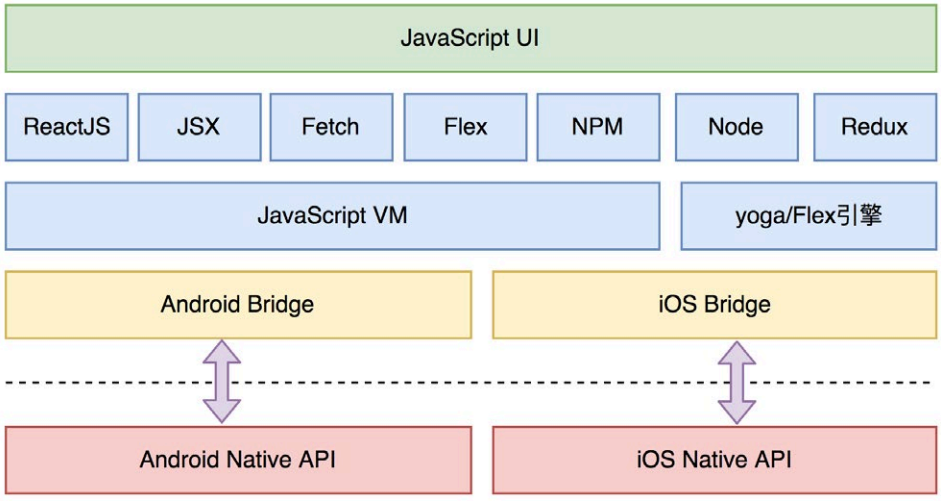
1.1.1 WebView 容器

WebView 容器的工作原理是基于 Web 技术来实现界面和功能，通过将原生的接口封装、暴露给 JavaScript 调用，JavaScript 编写的页面可以运行在系统自带的 WebView 中。这样做的优势是，对于前端开发者比较友好，可以很快地实现页面跨端，同时保留调用原生的能力，通过搭建桥接层和原生能力打通。但这种设计，跨端的能力受限于桥接层，当调用之前没有的原生能力时，就需要增加桥。另外，浏览器内核的渲染独立于系统组件，无法保证原生体验，渲染的效果会差不少。



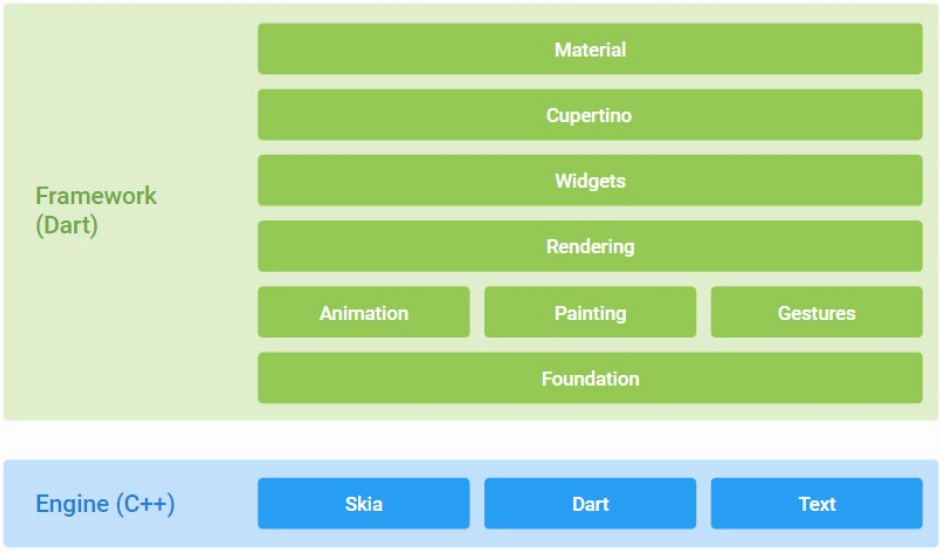
1.1.2 React Native

2015 年，Facebook 推出了 React Native，一经推出就备受关注。它的思路是最大化地复用前端的生态和 Native 的生态，和 WebView 容器的最大区别在于 View 的渲染体系。React Native 抛弃了低效的浏览器内核渲染，转而使用自己的 DSL 生成中间格式，然后映射到对应的平台，渲染成平台的组件。相对 WebView 容器，体验会有一定的提升。不过，渲染时需要 JavaScript 和原生之间通信，在有些场景可能会导致卡顿。另外就是，渲染还是在 Native 层，要求开发人员对 Native 有一定的熟悉度。



1.1.3 Flutter

2018 年 Google 推出 Flutter，通过 Dart 语言构建一套跨平台的开发组件，所有组件基于 Skia 引擎自绘，在性能上可以和 Native 平台的 View 相媲美。Flutter 站在前人的肩膀上，参考了 React 的状态管理、Web 的自绘制 UI、React Native 的 HotReload 等特点，同时考虑了与 Native 通信的 Channel 机制、自渲染、完备的开发工具链。Flutter 与上述 Recat Native、WebView 容器本质上都是不同的，它没有使用 WebView、JavaScript 解释器或者系统平台自带的原生控件，而是有一套自己专属的 Widget，底层渲染使用自身的高性能 C/C++ 引擎自绘。但大部分移动端发展到今天，都已经形成了自己的架构，在现有基础上加上 Flutter，会形成原有架构和 Flutter 双平台共存的问题。目前，对新的 App 来说，是最被看好的跨端方案。



1.2 美团外卖业务介绍

作为中国领先的生活服务电子商务平台，美团致力于用科技连接消费者和商家，提供服务以满足人们日常“吃”的需求，并进一步扩展至多种生活和旅游服务。而作为公司最为重要的业务之一，美团外卖从 2013 年创建以来，已经从单一的品类扩

展到附近美食、水果、蔬菜、超市、鲜花、蛋糕等多品类，从早午晚餐，发展到下午茶、宵夜，中餐、西餐、家常菜、小吃、快餐、海鲜、火锅、川菜、蛋糕、烤肉、水果、饮料、甜点等多种类餐饮。美团外卖可以说是当前电商领域，最为复杂的业务之一。

业务的复杂，给系统架构也带来了不小的挑战。美团外卖业务之所以说是当前电商领域最为复杂的业务，主要源于以下几点特征：

- 美团外卖业务承载在三个 App 上，美团外卖 App、美团 App 外卖频道、点评 App 外卖频道。
- 美团外卖作为美团公司重要的用户入口，还承担着流量平台的作用，提供平台能力支撑频道业务的发展，如闪购、跑腿、金融等。
- 美团外卖作为已经成熟的业务，需提供可复用的平台能力，支撑新业务的发展，例如菜大全 App 的发展。
- 美团外卖作为一个超级业务方，业务内又运营多个方向业务，如流量业务、交易业务、商家业务、商品业务、营销业务、广告业务等。

综上所述，可以发现美团外卖不仅仅自身业务比较复杂，而且对外的角色也很复杂。在美团内部，外卖不仅仅是美团平台的一个频道业务，而且自己本身也是一个平台业务，同时美团外卖还承担着新业务发展的平台角色。这意味着想要支持好美团外卖业务的发展是一件非常有挑战的事情。

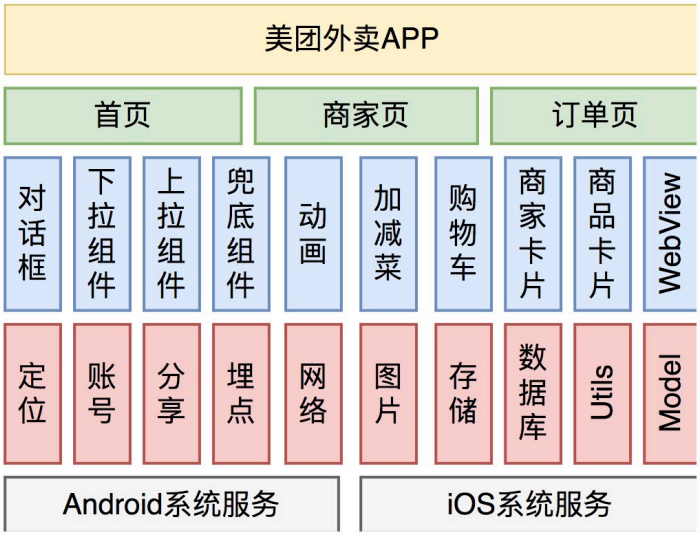
1.3 美团外卖移动端历史架构概述

好的架构源于不停地衍变而非设计。美团外卖的架构，历史上也是经历了很多次迭代。由于外卖业务形态不断地发生变化，原有的设计也需要不断地跟随业务形态进行演进。在不断探索和实践过程中，我们经历了若干个大的架构变迁。从考虑如何高效地复用代码支持外卖 App，逐渐地衍变成如何去解决多端代码复用问题，再从多端的代码复用到支持其他频道业务的平台架构上。在平台化架构建设完成后，我们又开始尝试利用动态化技术去支持业务快速上线的诉求。如今，我们面临着多端复用、平台

能力、平台支撑、单页面多业务团队、业务动态诉求强等多个业务场景问题。下文我们针对美团外卖移动端架构的变迁史，做一些简单的概述，以便读者阅读本文时能有更好的延续性。

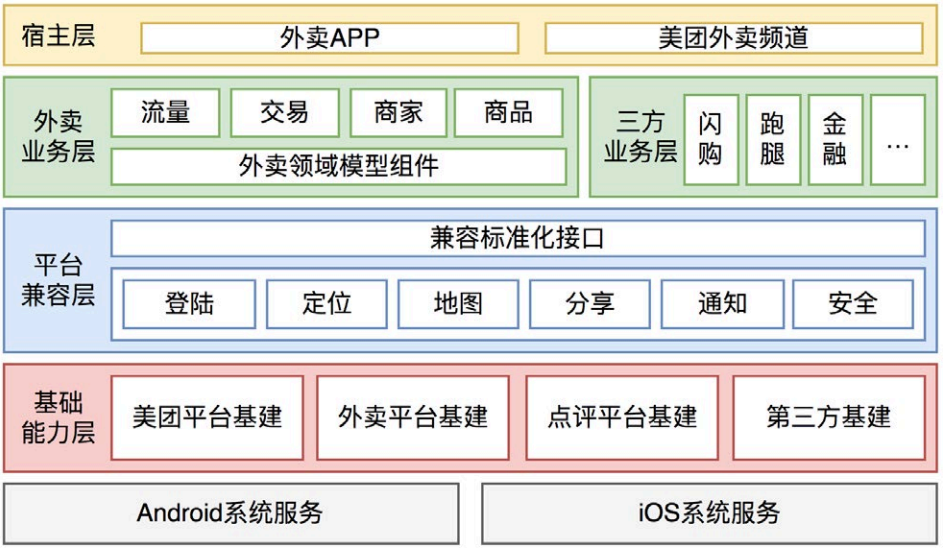
1.3.1 组件化架构

早期阶段，美团外卖作为公司的一个孵化业务，在 2013 年底完成了美团外卖 App 的 1.0 版本。随着外卖业务的验证成功和跑通，订单量也快速增长，在 2014 年底突破了日订单量 100 万。随后在 2015 年 2 月，外卖以 Native 的形式接入美团 App，成为美团 App 的一个业务频道。在接入过程中，我们从美团外卖 App 拷贝了大量的代码到美团 App 的外卖频道，两个 App 上的外卖业务代码也分别由两个独立的团队维护。早期外卖业务变化快，App 迭代频繁，写代码的方式也比较粗放，同时美团 App 也处在一个平台化转变的时期，代码的稳定性和质量都在变化和提升当中。这些因素导致了外卖代码内各子系统之间耦合严重，边界模糊，“你中有我，我中有你”的情况随处可见。这对代码质量、功能扩展以及开发效率都造成很大的影响。此时，我们架构重构的目的，就是希望将各个子系统划分为相对独立的组件，建设组件可以直接复用，架构如下图所示：



1.3.2 平台化架构

如上文所述，大家可以知道美团外卖和美团外卖频道是由不同的团队在维护发展。2015 年，公司考虑到业务发展的一致性，将美团外卖频道团队正式归于美团外卖。从组织架构上来说，美团外卖和美团外卖频道，逐渐融合成一个团队，但是两端的差异性，导致我们不得不仍然阶段性地维持原有的两班人马，各自去维护独立外卖 App 和美团外卖频道。如何解决这个问题？两端代码复用看起来是唯一的途径。另外，随着业务的快速发展，外卖 App 所承载的业务模块越来越多，产品功能越来越复杂，团队规模也越来越大，如闪购、跑腿等业务想以独立的 Native 包的形态接入外卖 App，还有外卖的异地研发团队的建立，都带来了挑战。这使得我们在 2017 年开始了第二次架构重构——平台化架构，目标是希望能够支持多端复用和支持不同团队的业务发展。通过抽象出平台能力层、业务解耦、建立壳容器，最终实现了平台化架构，架构如下图所示：



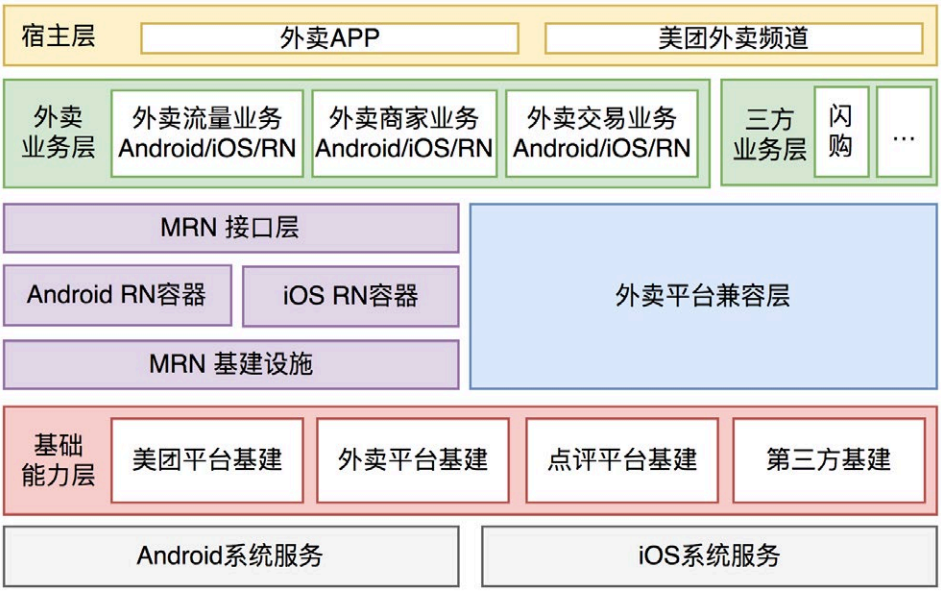
1.3.3 RN 混合架构

在平台化架构之后，美团外卖功能持续增加，美团外卖客户端安装包的体积也在持续增加。回顾 2017 年和 2018 年，每年几乎都增长 100%。如果没有一个有效的手

段，安装包将变得越发臃肿。另外，由于原生应用需要依托于应用市场进行更新，每次产品的更新，必须依赖用户的主动更新，使得版本的迭代周期很长。业务上的这些痛点，不断地督促我们去反思到底有没有一种框架可以解决这些问题。

在 2015 年的时候，Facebook 发布了非常具有颠覆性的 React Native 框架，简称 RN。从名字上看，就可以清楚的明白，这是混合式开发模式，RN 使用 Native 来渲染，JS 来编码，从而实现了跨平台开发、快速编译、快速发布、高效渲染和布局。RN 作为一种跨平台的移动应用开发框架，它的特性非常符合我们的诉求。美团也积极的探索 RN 技术。在 RN 的基础上，美团在脚手架、组件库、预加载、分包构建、发布运维等方面进行了全面的定制及优化，大幅提升 RN 的开发及发布运维效率，形成了 MRN（Meituan React Native）技术体系。

从 2018 年开始，美团外卖客户端团队开始尝试使用 MRN 框架来解决业务上的问题。使用 RN 的另一方面的好处是，能逐渐的抹平 Android 和 iOS 开发技术栈带来的问题，使用一套代码，两个平台上线，理论上人效可以提升一倍，支持的业务需求也可以提升一倍，架构如下图所示：



2. 美团外卖容器化架构全景图

2.1 什么是容器化架构

上文说到，外卖业务已经发展到多 App 复用、单页面多业务团队开发的业务阶段。要满足这样的业务场景下，寻求一个可持续发展的业务架构是件不容易的事情。经过我们之前架构演进，我们获得了宝贵的经验：在平台化架构的时候，我们将 App 和业务进行解耦，将 App 做成壳容器，业务形成独立的业务库，集成到壳容器里面，从而屏蔽了多 App 的问题，提高了业务的复用度。在 RN 混合式架构里面，我们引入了 RN 容器，通过这个容器，使得业务屏蔽了 Android 和 iOS 的平台差异。借助这些成功的经验，我们进一步思考，如果我们尝试进一步的细分外卖的业务场景，将不同场景下的基础能力建设成壳容器，业务集成到容器内，是否可以更好的支撑我们多 App 复用、单页面多业务团队的当前现状呢?

容器化架构的愿景是：

- 希望将前端呈现业务的环境抽象出来，将能力进行标准化，形成统一的容器，通过容器去屏蔽平台和端的差异。容器提供上层标准统一的能力接口，使得业务开发人员专注于容器内的业务逻辑的实现，最大复用已有的能力，而不用关注现在的环境是 Android 还是 iOS，现在的端是美团 App 还是大众点评 App。
- 容器和远端达成呈现协议，使得端上的内容具备随时可变化的能力。容器化架构的实现是存在一定前提的，如果业务的发展本身处在一个探索阶段，还有较多可变的因素，是无法形成稳定的能力层的，这时候建设容器化架构反而使得架构偏向复杂。但对于外卖业务场景来说，经过多年的沉淀固定，外卖业务逐渐形成了一套稳定的业务形态，已经进入到场景细分和快速迭代业务模块的阶段。在这样的阶段下，容器化架构才有可实施的前提。

2.2 容器化架构的优势

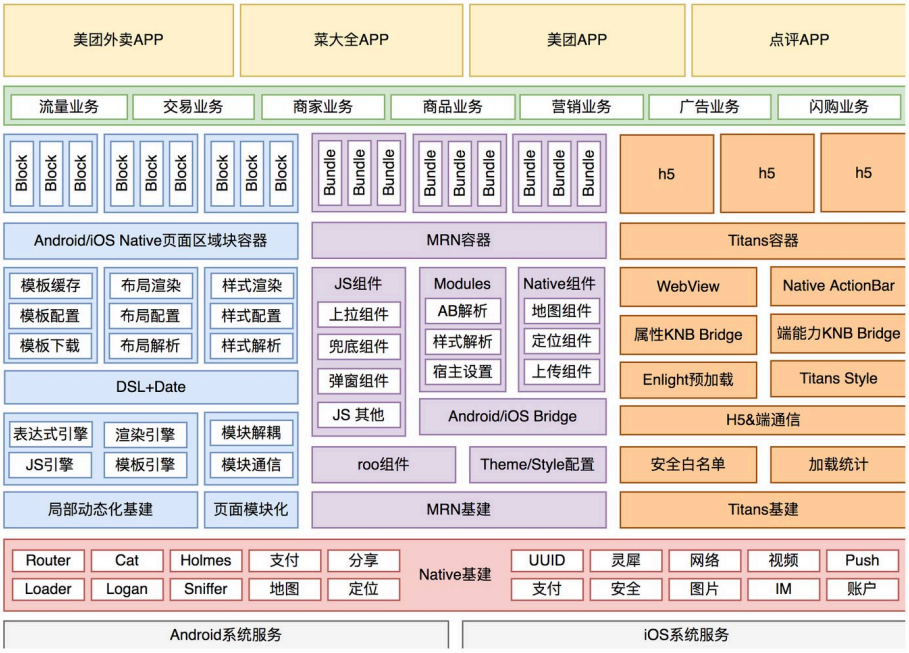
当我们把承载外卖业务的环境进行了抽象和标准化后，就可以获得以下若干点好处。首先动态化属性提升，我们可以把原有必须在客户端上写的业务放到了远端，业务的动态性得到很大的提升，具备随时上线业务的可能。对于开发过程而言，编译部署的

速度也得到了极大提升。如果涉及到客户端的代码改动，那客户端的编译打包，即使是增量的编译，也至少是秒级的编译速度。而容器化后，我们只打包必要的业务，把业务动态下发到容器呈现，客户端代码本身不会有变化，这样就可以从秒级的编译减少到毫秒级的编译。同样，业务动态下发，对减少客户端的包大小也有很大的帮助。

然后，容器位于应用之内，我们向应用中引入相同的容器 SDK，容器屏蔽了应用之间的差异，对于 Android 和 iOS 平台，在设计上，通过容器这一层去尽可能屏蔽平台之间的差异，使业务开发人员只需要认识容器，不需要花费大量的精力去关注应用和平台之间的差异，从而使得开发效率得到了极大的提升。

其次，容器化后，容器对承载的内容是有接口协议要求的，承载的内容只有满足容器定义的协议才能得到容器带来的好处，这促使业务得到了更细粒度的细分，业务开发时候，对模块化的意识得到了保障。另外，容器这一层提供的接口在 Android 和 iOS 上是标准化的，业务的开发也因为依赖的标准化，而趋向标准化，双端的业务一致性得到了提升。这些潜在的架构好处，对未来的业务维护和扩展都打下了比较好的地基。

2.3 外卖容器化架构全景图



整个外卖容器化架构可以按照从下到上，从左到右的视角进行解读：

最底层是系统服务，因为我们采用了 H5 和 RN 这样跨端的技术栈，使得 iOS 系统和 Android 系统成为了最底层。

系统服务之上是集团基于 Native 建设的基建，全公司通用，覆盖了研发工程中方方面面的基础服务。

在基建之上是我们定义的容器层。我们尝试用单一技术栈解决所有问题。但经过我们的探索，觉得不太可能实现。好的架构要匹配业务形态，业务的诉求决定了我们不能选择唯一的技术栈去解决所有问题，细分外卖的业务场景可得到以下 3 个方向的页面分类：

- 高 PV 主流程页面，例如首页、金刚页、商家页等，这类页面的 PV 远高于其他页面。这类页面的特点是，交互强、曝光度高、多团队参与建设和维护。针对这类页面，为了给用户提供极致的体验，是无法采用 H5 或 RN 实现的，即使性能上能够满足，但是这些页面是多团队共同参与建设，如果用 H5 或 RN 实现，那么单一团队改动，都会造成全页面受到无法预料的影响，其他未改动的业务方肯定是无法接受的。所以这类页面，我们采用的是局部动态化 + 页面模块化的方案，我们针对页面进行容器化改造，将页面变成容器，容器承载模块。每个模块归不同的业务方进行维护，从模块的维度进行解耦，每个模块都可以动态配置和下发。
- 低 PV 辅助页面，例如帮助、足迹、收藏等，这类页面的 PV 低，但胜在数量多，都用原生实现成本比较高，如果都用 H5 来实现，不少页面和原生的关联还是比较近，不是非常适合。针对这类页面，我们采用集团提供的 MRN 基建去承载，MRN 作为跨端的技术栈，我们已经在之前的 RN 混合架构的时候，建设了较为丰富的组件，针对自定义的 MRN 容器做了比较丰富的建设。同时，MRN 具备动态下发的能力，满足业务的诉求。
- 向外投放的运营活动页面，例如圣诞节活动，时效性比较强。这类页面由 H5 技术栈去完成，一方面可以满足时效性，另一方面 H5 的动态下发能力也是最强的，这样的特性能够充分的满足业务的诉求。我们使用集团提供的 Titans 基建，通过建设业务自定义的 Titans 容器，支撑业务的发展。

再往上，就是垂直的业务，外卖目前有流量业务、交易业务、商家业务、商品业务、广告业务、营销业务、闪购业务等。业务都是垂直向下依赖，直接可见容器，可见基建，能够很好地获取到各种已经建设的能力去完成业务的需求。

最上面是承载的 App 端，目前有四端，包括外卖、点评、美团、闪购等等。

右侧是测试发布和线上监控，相对于常规的移动端 App 架构而言，容器化架构的测试发布和监控是更为精细化的。不仅仅要关注端本身的可用性，还需要关注容器、容器承载的模块、模块展示的模板，模板里面的样式这些的可用性。

2.4 容器化的挑战

容器化架构相对常规的移动端架构而言，它从管理移动端的代码转变成管理移动端的容器建设代码和业务远端开发代码，多出了容器和业务远端下发。这不仅仅是对技术上的挑战，对长期做客户端开发同学，也需要一个思维转变的跳转。

一致性的挑战：容器需要在多个宿主应用之中运行，宿主应用的环境一致性直接影响了容器的一致性。我们的策略是两手准备，一方面利用外卖业务的优势推动宿主应用的环境对齐；另一方面将容器建设成 SDK，通过 SDK 将长期保持容器的一致性，也通过 SDK 内部的设计屏蔽应用之间的差异；对于 Android 和 iOS 平台，我们通过分层的设计，尽可能屏蔽平台的差异。综上所述，一致性的挑战在于（1）容器运行的宿主应用的环境一致性；（2）不同应用不同版本容器的一致性；（3）Android 和 iOS 平台容器的对业务的一致性。

动态发布的挑战：长期以来，客户端同学的开发概念里面只有 App 版本的概念，而当我们逐渐把业务代码做成远端下发时，将会新增一个线上动态发版的概念。当我们在发布业务的时候，相对以往的工作，多出需要去考虑这个业务的版本，可以运行的容器对应的 App 上下界版本。另外，发版的周期也会新增业务的发版周期，不仅仅是 App 的发版周期。这两者在一起将会产生新的火花，业务的版本和 App 的版本如何适配的问题，业务动态发版的周期和 App 的发版周期如何适配的问题。外卖这边的解决方式是建设主版本迭代 + 周迭代的模型。

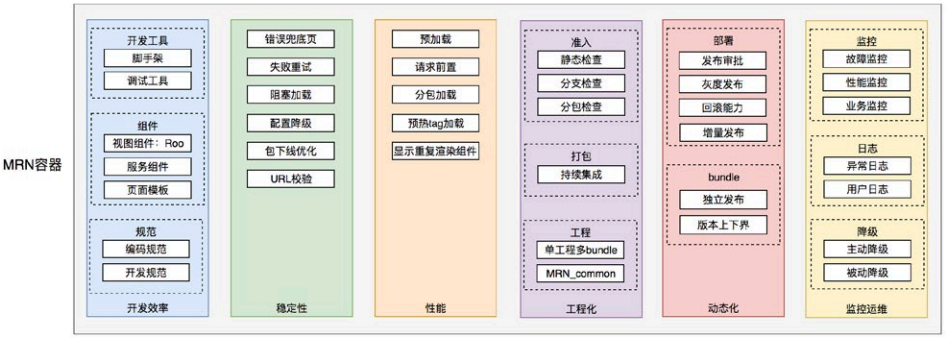
监控运维的挑战：以往的移动端架构，我们更加关注的是端本身的可用性，然而当我们演进到容器化架构的时候，仅仅关注端的可用性已经远远不能确定业务是可用的了。我们需要从端的可用性延伸出下载链路、加载链路，使用链路上的可用性，针对每个重要的环境，都做好监控运维。

3. 外卖跨端容器建设

3.1 MRN 容器

3.1.1 MRN 容器简介

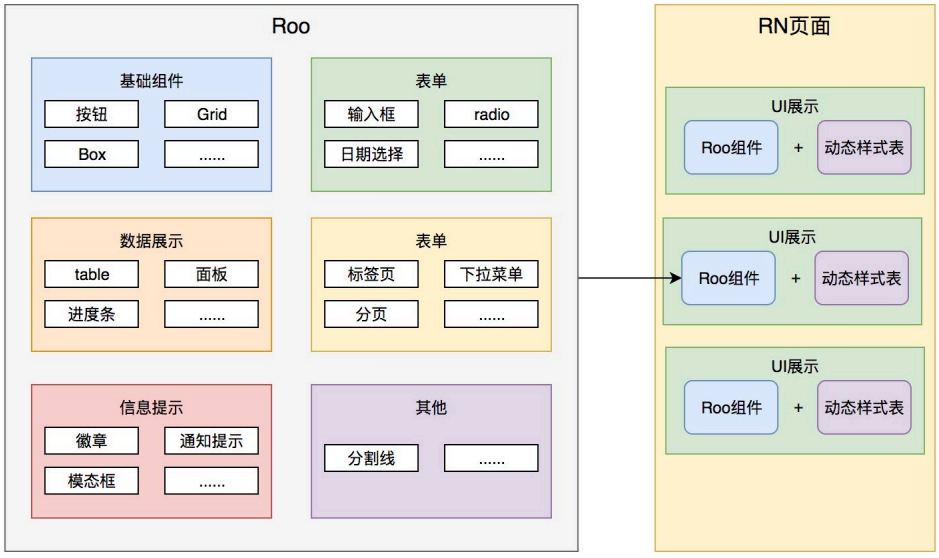
React Native 框架本身只是一个运行时环境中的渲染引擎，可以将同一套 JS 代码分别在 Android 和 iOS 系统上最终以 Native 的方式渲染页面，从而为 App 提供了基础的跨端能力。但从工程化的角度来看，如果想在 App 中大规模地应用 RN 技术，除了 RN 框架本身外，还需要在开发、构建、测试、部署、运维等诸多方面的配合。MRN（Meituan React Native）是美团基于 React Native 框架改造并完善而成的一套动态化方案，在 RN 的基础上提供了容器化能力、动态化能力、多端复用能力和工程化保障。MRN 在开发效率、稳定性、性能体验、动态化和监控运维等多方面进行了能力升级和扩展，满足了美团 RN 开发工程化的需要。目前，MRN 已接入美团 40 多个 App，核心框架及生态工具有超过 100 位内部代码贡献者，总 PV 超过 4 亿。



3.1.2 Roo 组件库

下面再介绍一下外卖建设的两个 UI 相关的技术项目，Roo 组件库和组件样式动态配置。

- **Roo 组件库**：外卖在 RN 及 MRN 框架提供的 UI 组件基础之上，又扩展了适用于外卖业务的标准化 UI 组件库。UI 组件库一方面统一了我们的设计规范和开发规范，提高了 UI 一致性；另一方面，组件封装也提升了 RN 页面的开发效率和质量。
- **组件样式动态配置**：有了标准的 Roo 组件，我们进一步给标准组件的动态添加了样式动态配置能力。在使用组件时，很多样式是支持动态下发的，例如字体、圆角、背景色等，方便我们进行 UI 的适配和改版。

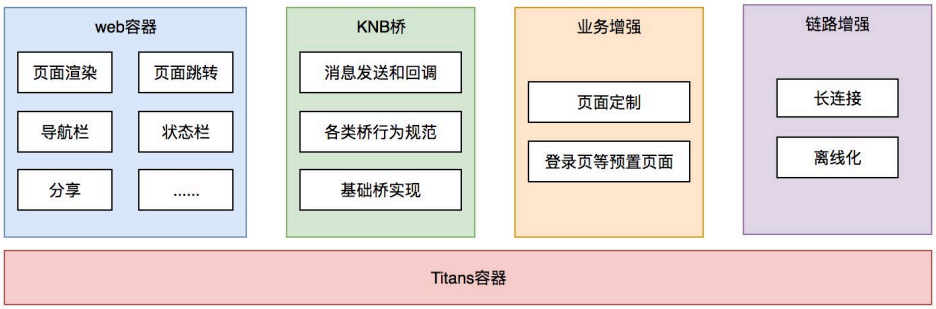


外卖在 2018 年底开始试验 MRN 容器在外卖业务上的应用，并在 2019 年上半年进行了大面积的页面落地。目前，外卖已有近 60 个 RN 页面上线，占外卖页面比例超 80%，其中包括 Tab 页面“我的”、提单选择红包页、订单评价页等高 PV 页面。MRN 容器的接入，给外卖 App 的容器化、动态化、人效提升、包大小瘦身等方面都做出了不小的贡献。

3.2 Titans 容器

3.2.1 Titans 容器简介

Titans 容器是美团系 App 统一的 Web 容器组件，基于苹果提供的 WebView 组件，将 WebView 容器化，统一了 WebView 的 UI 展示和交互方式，规范了桥协议的使用范式，同时预置了诸多基础能力和业务能力。Titans 容器大大提高了 Web 页面的开发效率和用户体验上的一致性。



- **Web 容器**：Titans 容器提供了统一的 UI 展示和自定义样式，例如导航栏样式、页面 Loading 状态、进度条样式等；还有统一的交互方式，例如页面跳转、Scheme 协议的解析等。
- **KNB 统一桥服务**：Web 容器虽然在动态化和 Android、iOS 双端复用上很好地弥补了 Native 的不足，但在很多地方体验上又难以达到 Native 的标准。因此，KNB 桥应运而生，KNB 定义了 Native 和 JS 通信的标准方式，方便开发时进行桥协议扩展，同时 KNB 也内置众多的 Native 基础能力，极大地提高了 Titans 容器的用户体验和开发效率。
- **Web 业务增强能力**：Titans 容器中预置了丰富的基础业务页面，例如登录页面、分享弹窗等。
- **提供链路增强**：提供了长连接、离线化等方式来提高网络请求的速度和成功率。
- **WebView 预加载**：在 App 启动之后，用户点击网页入口之前，提前加载好 HTML 主文档和基础库，这样当用户点击页面入口时，App 直接使用已准备

好的 WebView，仅需加载少量的业务代码。从而达到白屏时间短、加载页面迅速的效果。

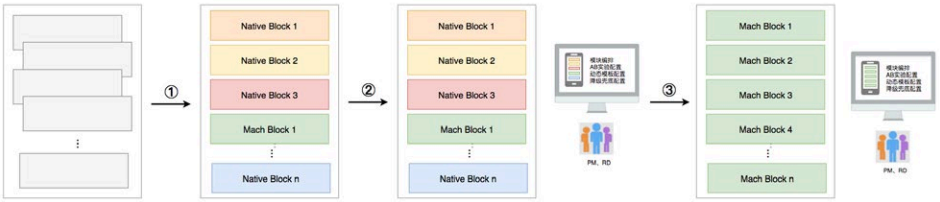
Titans 容器在外卖业务中的使用场景非常丰富，其中最重要的使用场景是各种运营页和活动页，例如点击首页顶部 Banner 的广告落地页、为你优选、限时秒杀等活动运营页等；还有客服页、帮助反馈页、商家入驻页、美团公益页等功能性页面；作为一级入口页面的美团会员页面，也是一个基于 Enlight 的 Titans 容器。

4. 外卖页面容器建设

外卖容器化建设，首先需要要区分的是核心页面和非核心页面。外卖业务中对核心页面的定义是页面 DAU> 美团 DAU 的 5% 或者是下单关键路径。为什么要先按照是否为核心页面进行拆分呢？重点就在于改造的成本。核心页面的业务复杂度决定了它不容易做全页面的动态化，它比较适合做局部的动态化方案。核心页面的复杂度在于业务本身复杂，最重要的是核心页面往往会有多个垂直业务团队共同的开发维护，大家各自有重点关注的模块，做全页面的动态化，无法做到有效的物理隔离。

而对于非核心页面，业务功能和交互相对简单，组织关系也较为确定，更适合做标准的 MRN 和 Titans 容器化。所以我们的策略是核心页面做到支撑页面模块级别的业务动态和复用，非核心页面可以做到页面级别的动态化和复用。页面容器化的核心含义就是把一个页面划分为若干个模块，每个模块成为一个业务容器，每个容器的填充既可以用 Native 的方式实现，也可以用 Mach 实现（Mach 是外卖自研的页面局部动态化技术），可以支持 iOS/Android/ 小程序三端跨平台运行。页面本身则化身为容器的管理者，负责子容器的编排和布局，并支持其动态化。

4.1 页面容器化设计思路



页面容器化设计中主要分为三个阶段，模块有序化、模块编排化、渐进式业务落地。

- **模块有序化**：将耦合的外卖业务代码按模块维度进行拆分，建立标准化的模块间组合和交互方案，降低模块内改动对其他模块的影响。这个阶段我们同时完成了 Native 原生模块和局部动态化模块的标准化改造。
- **模块编排化**：页面容器化的一个特点是页面具备编排模块的能力，在这个阶段我们在客户端增加了对业务模块结构编排能力的支持，同时我们跟后端的同学共建了配置平台，通过配置化的方式打通了 AB 实验平台、统一数据服务等多个平台。在标准化的数据协议的支撑下，页面支持了 AB 实验动态上线，大大降低了客户端在 AB 实验方面的开发成本，做到了客户端零成本，后端低成本，高效地支撑了外卖首页六周年的大改版。
- **渐进式业务落地**：页面容器化后最明显的收益是支持业务需求的动态上线，只有页面容器中的业务模块具备动态能力才能实现这个目标，这会涉及 Native 模块迁移为 Mach 模块的过程。在这个方面，我们的思路是跟随业务需求渐进式落地。在模块编排阶段，我们设计了 Native 模块向 Mach 模块迁移的能力，同时设计了覆盖模板维度和 API 接口维度的三重降级方案，来保障模块动态化迁移的稳定性。

4.2 业务构建模块标准化

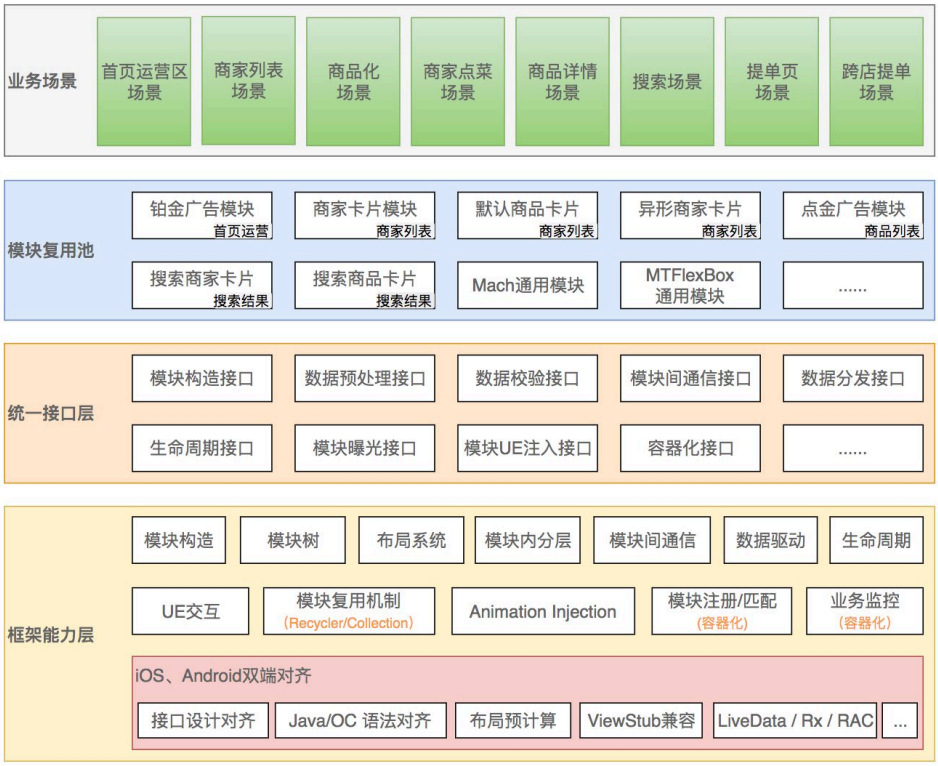
从 App 页面开发的角度看，一个完整的页面可以按照不同的功能及不同业务属性划分出多个不同的模块。

业务构建泛指由多个业务模块组合拼装为一个业务页面的过程，涉及页面本身

（UIViewController/Activity）以及各个业务模块的构造过程，前后端业务数据以及页面和业务模块之间的数据交互过程，业务模块内部的数据处理以及视图刷新流程。

模块标准化指的将业务构建涉及到的多个过程通过规范化的方式确定下来，形成唯一的标准。模块标准化一方面能够在解决业务共性问题的基础上提供业务难点专项解决方案，另一方面能够在框架基础上形成能力约束，减少重复建设、低质量建设的问题。

业务构建模块标准化中我们抽象了四层，下面将分别进行解读。



- 最底层是框架能力层，是外卖业务团队自研的符合外卖业务特点的双端模块化框架。框架解决了不同页面场景下的共性问题，对典型的业务痛点也进行了支持。它是一种页面框架设计在 iOS、Android 双端对齐的实现方案，这种双端

对齐的能力为页面容器化设计的双端一致性提供了保障。

- 统一接口层是对框架能力层的标准化抽象，它可以保证任一模块调用的能力在各个业务场景下的实现都是一致的，有了这一层抽象任一模块都可以直接在各个场景下复用。
- 在往上就是 App 全局的模块复用层，标准化后的模块可以通过唯一标示向模块复用池注册模块，这种中心化的注册方式可以让业务模块在跨业务库的场景下可以灵活地复用。
- 最上层就是外卖的核心业务场景层，每一个场景都对应了一个标准化的页面容器，页面容器通过实现容器化接口来完成页面容器的构建。

通过业务构建模块标准化的建设，业务模块已经是标准化的了，可以在跨页面间自由组合，这为页面容器化打下了基础。

在页面容器化中最基础的能力有以下几点：页面中业务模块可编排能力，动态上线前端 AB 实验的能力，增量上线动态模块的能力。实现这些能力最重要的就是进行前后端数据协议标准化建设。客户端根据数据协议中的模块唯一标识匹配并构造业务模块，在完成模块数据的填充后会根据数据协议中的模块布局信息完成模块的布局。针对 Mach 动态模块，我们创建了基于模板 ID 的模块匹配和数据填充流程，可以支持 Mach 动态模板的增量上线。在数据协议中针对前端 AB 实验我们预留了 AB 实验和通参字段，在数据填充阶段通过容器化接口传入动态模块中，用于支持 AB 实验的动态上线。

在容器化上线的过程中属于接口的大版本升级，为了保证容器的高可用性，客户端从模块级别和 API 级别实现了两套降级容灾方案。

模块级别的降级方案主要针对 Mach 动态模块，与 Native 模块不同，Mach 动态模块需要预先下载动态模板才能正常地完成模块的载入和渲染。为了保证动态模块的加载成功率，我们一方面在接口上线前利用 Eva（美团内部系统）对 Mach 模板的下载进行预热。另一方面，我们设计了动态模块的主动降级方案，针对动态模块的动态上

线使用 Native 模块进行兜底降级，对于跟版动态模块使用 App 内置模板的方案进行兜底降级。

API 级别的容灾方案主要为了保障客户端在新接口不稳定的情况下可以自行降级到旧接口。针对这个问题，我们对线上老接口设计了数据结构映射方案，在客户端通过配置化的方式可以把老接口的数据结构映射为新接口的数据结构。这样在上层业务无感知的情况下，可以做到容灾方案的上下线。

4.3 小结

通过页面容器化，使得页面只需要关心页面级的构造方式，而无需关心某一模块内部如何实现动态化的。把页面与页面的模块分离，也符合目前外卖客户端的组织结构，有利于业务组间的协作。同时，页面容器化使得外卖核心页面具备了符合外卖业务场景下的动态能力，渐进式把 Native 静态模块过渡到具备动态能力的模块，从模块的维度使整个页面具备了动态能力。这种渐进式的迁移方案把容器迁移跟业务模块的迁移分隔开，大大降低了页面容器化改造的风险。

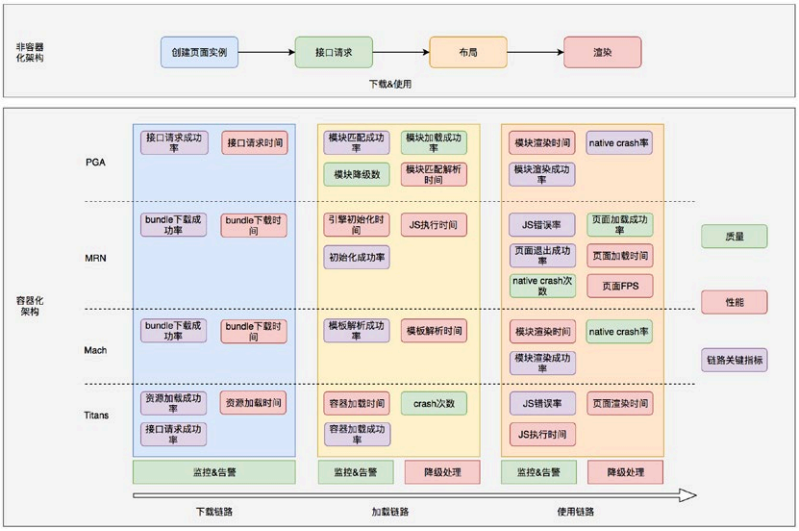
5. 外卖容器化架构的衡量指标

5.1 容器化架构衡量指标的特点

质量和性能指标是衡量我们 App 开发质量和用户体验的重要依据，是我们一直都非常关注的重点数据。在非容器化时代，我们大多数的指标都和 App 的使用环节紧密相关，因为在非容器化时代，逻辑链路相对简单，例如我们打开一个新页面时，我们首先创建页面实例，然后发起网络请求，同时页面会经历一系列生命周期方法，最后渲染。这时我们可能会关注网络请求的成功率和请求时间，页面的渲染时间，和过成功是否发生 Crash，这个过程相对更短暂，指标更少，所以监控起来也更容易。

外卖的容器化大大提升了外卖业务的复用能力、动态能力、模块化和开发效率，但同时也带来了更长的逻辑链路，链路从时间维度上划分是：下载链路、加载链路、使用链路。例如我们在使用 MRN 容器的时候，会涉及到 bundle 的启动下载或预热下载，

bundle 解压缩，MRN 容器引擎初始化，bundle 加载，JS 的加载、执行，页面渲染等步骤，其中的每个步骤都可能存在性能问题和质量风险。因此，我们需要升级我们的衡量指标系统来应对容器化带来的新的挑战。



5.2 链路指标

- **下载链路**：在下载链路中下载容器所需的各种资源，在 MRN 和 Mach 中主要是 bundle 的下载任务，只有 bundle 下载成功，才能进行后面的各项操作。所以 bundle 下载的成功率是下载链路中最重要的指标，同时 bundle 下载的时机也很重要。外卖业务中有各种 bundel 上百个，如果在启动时拉取所有 bundle，对冷启动时间会造成极大的影响。我们采用了 bundle 预热的方法，发布 bundle 是给 bundle 打上相应的 Tag，在适当的时机去下载，避免集中下载。
- **加载链路**：在加载链路中重要工作是对下载链路中下载的资源的解压和解析。例如在用 PGA 加载页面时，会进行模块的解析、模块匹配、模块降级、数据模型生成等步骤。在 MRN 中会进行 bundle 解压、引擎初始化、bundle 加载等步骤。加载链路往往是比较消耗计算资源的步骤，对页面打开和加载时间影

响较大，所以我们会比较关注加载链路的性能指标。

- **使用链路**：使用链路和非容器化的使用阶段基本相同，会主要关注页面的加载时间、Crash 率、页面页面 FPS、页面卡顿等指标。除此之外，还会关注和容器本身特性相关的一些指标，例如在 MRN 容器中，我们还会关注 JS 错误率、JS 渲染时间、白屏率等指标。

5.3 关键指标

因为容器化的使用形成了一个串行的链路，所以如果某个关键节点失败，会导致容器功能不可使用，关键指标的任务就是从上述众多的指标当中筛选出这些关键节点。例如在下载链路中 bundle 下载的成功率和 API 的成功率，加载链路中 bundle 加载的成功率和模块匹配的成功率，下载或加载失败都无法再进行链路中的后续步骤，针对上面的成功率指标，我们会添加分钟级别的实时监控告警，做到及时发现，快速响应和紧急修复。

在使用链路中模块渲染的成功率、Native Crash 率、JS 错误率也属于关键指标，这些任务的失败也会导致容器的不可用，针对这些指标我们也会采用实时监控措施，并且添加降级手段，例如回滚 bundle 版本，或者把 MRN 和 Mach 容器降级为 Native 容器。

6. 外卖容器化架构的监控运维

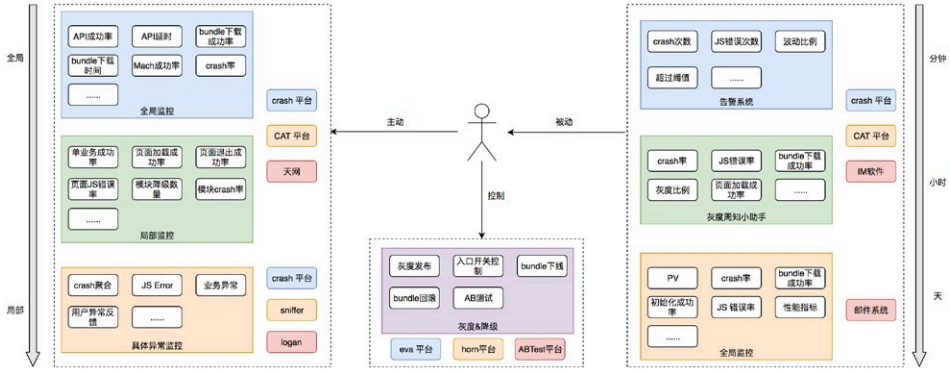
上面讲到了容器化架构的各项衡量指标，那么把这些指标具体落到实处的工作就是线上的运维监控工作。工欲善其事，必先利其器，对于监控运维工作，一定要有合适的监控工具辅助配合才能事半功倍，公司内有很多优秀的监控统计工具可供使用，这里的难点就是如何根据监控的需要判断选择合适的工具。还有就是合理的划分监控维度和数据指标的优先级，例如对于能够影响到链路稳定性的关键指标，我们需要做到分钟级的监控，一旦出现问题就能及时收到告警，对于非关键指标，则通过生成日报的方式，方便开发者的统计和分析。

工具的使用上主要分为大盘工具、具体异常工具、灰度降级工具、告警工具等（以下是美团内部使用的工具）。

- **大盘工具**：主要使用 CAT、天网、Crash 平台等工具。这些工具收集、统计大盘数据，然后生成可视化的图标和曲线。方便开发者了解大盘的整体情况和变化趋势。
- **具体异常工具**：使用 Sniffer、Logan 等工具。这些工具可以用来获取发生异常时的上下文和设备信息，回捞用户行为日志，方便定位排查具体问题。
- **灰度降级工具**：使用 ABTest 平台、Horn 等工具。用于下发配置，以进行灰度控制或开关控制。
- **告警工具**：告警小助手。将告警通过 IM 软件及时发送到个人或群组，做到及时发现及时处理。
- **业务覆盖维度监控**可以分为全局监控和局部（单业务）监控。
- **全局监控**：监控各项容器化质量指标、性能指标，生成每日报表，方便跟踪监控容器化的整体质量。
- **局部（单业务）监控**：实时监控每个页面、每个容器的线上数据，做到有问题及时发现，及时定位，及时处理。
- **时间维度监控**：可以按天、小时、分钟的时间维度。天级别的监控主要是一些非关键路径指标，例如一些性能指标，页面加载时间、页面 FPS、JS 渲染时间等，我们可以按天维度的生成数据报表，已邮件的数据发送日报。当 App 灰度上线时，我们会开始小时级别的监控，每过半小时通过 IM 软件向广播一些关键指标，方便开发者跟踪线上数据的稳定性。分钟级别的监控则是针对关键指标，观察分钟维度上的变化，如果关键指标超过阈值，或者波动过大，就会及时产生告警。

下面我们以一个开发者的视角去看一下外卖容器化架构的监控运维系统。从获取信息的方式上可以分为主动查询和被动推送，开发者可以通过监控工具监控全局和局部数据的变化趋势，也可以分析具体异常 Case；也可以从 IM 工具，邮件等收到相关的推送数据，以便及时响应。在控制运维上，开发者可以通过 Eva、Horn 等美团内部的灰度系统进行灰度发布，当灰度期发现问题的时候，可以及时地通过停止灰度，版

本回滚，关闭入口的方式进行降级容灾处理。



7. 外卖容器化架构的发布能力

7.1 容器化架构发布体系

容器化使外卖业务具备了强大的动态化能力，但动态化能力又和需要相应的发布能力来支持，发布能力是我们业务开发质量和效率的重要保障，也是我们容器化建设工作过程中的重点环节，这一节主要介绍一下外卖容器化的发布能力。

从发布能力类型的角度看主要可以分为三种类型：(1) 容器内容的发布，包括发布整个页面或者发布页面中的局部模块；(2) 配置下发，通过 API 或其他配置平台，下发布局协议、AB 测试、样式配置、功能配置、模板配置、容器配置等，大大提高了业务的灵活度和线上验证能力；(3) 灰度、降级下发，通过 UUID，用户画像等信息做到灰度发布，降级回滚等控制能力。

从发布资源的的角度看主要分为两种：一种是普通的资源，例如发布一个 Web 页面，或者通过发布新版 API 来控制页面局部容器的展示与否和展示的位置，同时我们也可以进行一些 AB Test 操作；另一种是 bundle 资源，主要是针对 MRN 容器和 Mach 容器，每个 MRN 容器和 Mach 容器的资源都会先被打包成一个 bundle，然后通过发布系统下发到终端，然后终端解析 bundle 中的代码和资源，最终渲染页面。

从发布阶段的角度看，可以分为测试阶段、上线阶段、灰度阶段和全量阶段，其中上线阶段是最终的环节，我们增加了很多校验和保护手段来尽量保证上线操作的正确性。

7.2 跟版本发布流程

虽然我们具随时备动态发布能力，但正常的版本迭代还是会存在中，所以外卖这边的节奏是周动态迭代 + 双周版本迭代，这保证了我们的开发工作有个一清晰的周期。在动态发布阶段中最关键的阶段操作上线阶段。以 MRN 为例，目前外卖业务中应有 70 多个 bundle，再算上测试环境的 bundle 就有接近 150 个 bundle，只是管理这些 bundle 就是一个复杂的工作，况且在进行上线操作时还是涉及发布的目标 App、App 版本的上下界、MRN 版本的上下界等，一不小心就会造成操作失误，所以进行上线操作时需要非常谨慎。

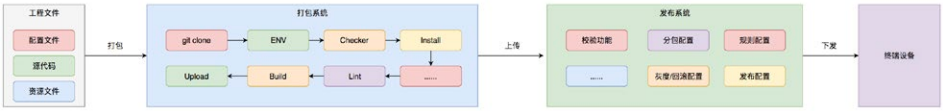
我们针对操作上线阶段进行了事务流水线，通过流水线建立保护措施，一个 bundle 的上线要经历一个流水线的若干操作。首先，操作人根据上线 SOP 手册进行若干检查操作，同时编写标准格式的发布说明，然后周知相关核心人员后在操作系统上发起上线申请，Leader 和 QA 收到申请后会进行检查并审批，审批通过后还要避开 App 使用的高峰期或节假日上线，上线后通过灰度发布观察各项数据指标，指标正常后全量发布。

7.3 bundle 资源发布

bundle 是我们最常发布的资源类型，这里再结合发布工具讲解一下 bundle 的发布过程。MRN 和 Mach 都是以 bundle 的形式下发到设备终端的，我们在发布 bundle 的时候主要会用到两个工具，打包工具 Talos 和发布工具 Eva（美团内部工具）。一

个 bundle 的工程文件主要由三个部分组成：配置文件、源代码和资源文件，其中配置文件用于指导 Talos 对工程文件进行打包，多个 bundle 可以共享一份配置文件。当我们准备发布一个 bundle 时，先找到该 bundle 在 Talos 的发布模板，选择发布环境（测试或线上），然后进行一键打包，然后 Talos 会进行一系列流水线操作，包括 Clone 代码、配置环境、进行 Lint 检查、构建和上传等。Talos 打包完毕后将 bundle 上传到 Eva 系统，然后 Eva 负责 bundle 的分包、上线、下线、灰度等操作，最终下发到终端设备上。

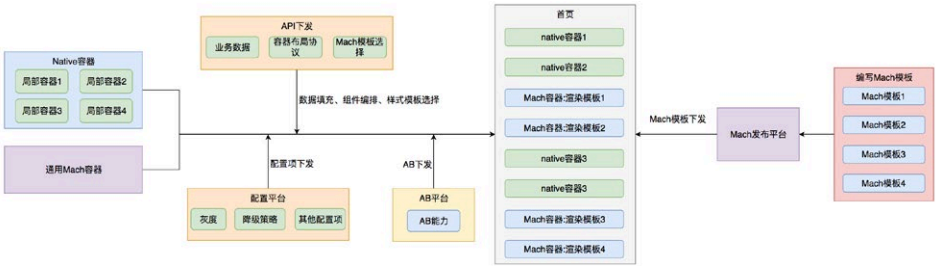
未来，我们还将引入美团住宿的 MRN-DevOps 来进一步的屏蔽当前多系统的问题，降低整个周期管理的成本，特别是发布前的人工检查成本，逐渐实现 RD 在一个平台上操作从研发到发布运维的所有实现。尽可能地减少人工成本，提升自动化。



7.4 多种发布能力综合使用

上面介绍的是以 bundle 资源形式的发布流程，过程较为清晰简单。下面再结合外卖首页，介绍一下局部容器化的发布方式。外卖首页是典型的流式列表，在局部容器化的架构下，首页就是由一个个矩形容器以 ListView 方式布局的，容器分两种，Native 容器和 Mach 容器，Mach 容器是一个通用容器，我们可以编写不同的样式模板，下发到终端后交由通用 Mach 容器来渲染，以此达到只使用通用容器展示不同 UI 样式的目的，这里涉及了 Mach 的发布系统。

首页各子容器相当于一块块积木，它们的位置排布、展示与否、模板的选择等最终交由 API 控制，API 具备了控制首页布局，样式展示的能力，而不再是单纯的数据源。同时，首页也涉及了 AB 能力、灰度降级策略等其实配置项下发系统。可以看到外卖首页的容器化是由多种发布能力配合支撑的，是外卖发布能力体系的“集大成者”。



8. 总结

好的架构是要随着业务的发展，不断演变去适应业务的发展。美团外卖从一个很小规模，每日单量只有几千的业务，逐渐地走到今天，每日单量峰值超过 4000 万，组织架构也从一个十几个人的团队，逐渐发展到现在多角色、多垂直业务方向，上千人共同协作的团队。移动端上的架构，为了适应业务的发展要求，也经历了组件化、平台化、RN 混合化，再到现在向容器化的变迁。

容器化架构相对于传统的移动端架构而言，充分地利用了现在的跨端技术，将动态化的能力最大化的赋予业务。通过动态化，带来业务迭代周期缩短、编译的加速、开发效率的提升等好处。同时，也解决了我们面临着的多端复用、平台能力、平台支撑、单页面多业务团队、业务动态诉求强等业务问题。

当然，容器化架构带来好处的同时，对线上的可用性、容器的可用性、支撑业务的线上发布上提出了更加严格的要求。我们通过监控下载、加载、使用链路上的可用性，来保障线上动态业务的可用性。针对容器，我们利用成熟的测试基建，建设容器的自动化测试来保障容器的可用性。针对发布，我们建设迭代流程，配合发布流水线，将线上的发布变得更为可控。

截止到目前为止，外卖业务经过了几十个动态化业务上线窗口，累积共发版百次以上。未来半年，我们还将进一步从业务需求入手，将业务需求细分归类，让产品侧逐渐建立容器和动态化需求的概念，能够从源头上，逐渐的将业务进行划分，最终使得每个业务需求，都可以归类抽象成可以动态下发的业务和容器能力建设，从而进一步的完善容器化架构的能力和支持更多的的业务场景。

9. 参考资料

React Native 在美团外卖客户端的实践
美团外卖 iOS 多端复用的推动、支撑与思考
美团外卖前端容器化演进实践
UC 浏览器客户端容器化架构演进
你知道支付宝容器化架构是怎么搭建的吗?
讲一讲移动端跨平台技术的演进之路
盘点主流移动端跨平台 UI 技术：实现原理、技术优劣、横向对比等

10. 作者简介

郭赛，同同，徐宏，均为美团外卖 iOS 工程师。

11. 招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，欢迎有兴趣的同学投递简历到 wangxiaofei03@meituan.com。

Flutter 包大小治理上的探索与实践

作者：艳东　宗文　会超

一、背景

Flutter 作为一种全新的响应式、跨平台、高性能的移动开发框架，在性能、稳定性和多端体验一致上都有着较好的表现，自开源以来，已经受到越来越多开发者的喜爱。随着 Flutter 框架的不断发展和完善，业内越来越多的团队开始尝试并落地 Flutter 技术。不过在实践过程中我们发现，Flutter 的接入会给现有的应用带来比较明显的包体积增加。不论是在 Android 还是在 iOS 平台上，仅仅是接入一个 Flutter Demo 页面，包体积至少要增加 5M，这对于那些包大小敏感的应用来说其实是很难接受的。

对于包大小问题，Flutter 官方也在持续跟进优化：

- Flutter V1.2 开始支持 Android App Bundles，支持 Dynamic Module 下发。
- Flutter V1.12 优化了 2.6% Android 平台 Hello World App 大小（3.8M -> 3.7M）。
- Flutter V1.17 通过优化 Dart PC Offset 存储以减少 StackMap 大小等多个手段，再次优化了产物大小，实现 18.5% 的缩减。
- Flutter V1.20 通过 Icon font tree shaking 移除未用到的 icon fonts，进一步优化了应用大小。

除了 Flutter SDK 内部或 Dart 实现的优化，我们是否还有进一步优化的空间呢？答案是肯定的。为了帮助业务方更好的接入和落地 Flutter 技术，MTFlutter 团队对 Flutter 的包大小问题进行了调研和实践，设计并实现了一套基于动态下发的包大小优化方案，瘦身效果也非常可观。这里分享给大家，希望对大家能有所帮助或者启发。

二、Flutter 包大小问题分析

在 Flutter 官方的优化文档中，提到了减少应用尺寸的方法：在 V1.16.2 及以上使用—split-debug-info 选项（可以分离出 debug info）；移除无用资源，减少从库中带入的资源，控制适配的屏幕尺寸，压缩图片文件。这些措施比较直接并容易理解，但为了探索进一步瘦身空间并让大家更好的理解技术方案，我们先从了解 Flutter 的产物构成开始，然后再一步步分析有哪些可行的方案。

2.1 Flutter 产物介绍

我们首先以官方的 Demo 为例，介绍一下 Flutter 的产物构成及各部分占比。不同 Flutter 版本以及打包模式下，产物有所不同，本文均以 Flutter 1.9 Release 模式下的产物为准。

2.1.1 iOS 侧 Flutter 产物

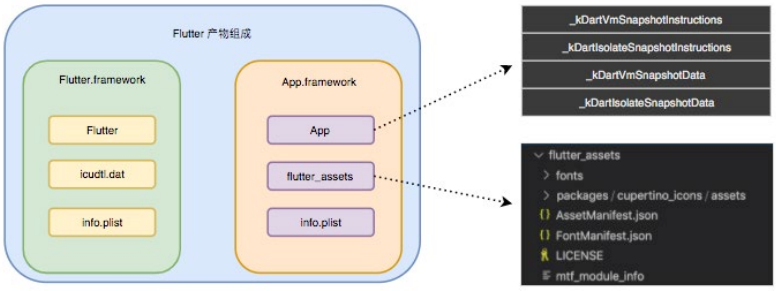


图 1　Flutter iOS 产物组成示意图

iOS 侧的 Flutter 产物主要由四部分组成（info.plist 比较小，对包体积的影响可忽略，这里不作为重点介绍），表格 1 中列出了各部分的详细信息。

表 1　Flutter 产物组成

产物	大小	占比	介绍
Flutter	7.2MB	50.6%	即为Flutter Engine，C++代码编译而成。
icudtl.dat	833KB	5.7%	国际化支持相关数据文件，大小固定为 883KB
App	5.2MB	36.7%	dart业务代码AOT编译的产物，其主要包括： _kDartIsolateSnapshotData、 _kDartVmSnapshotData、 _kDartIsolateSnapshotInstructions、 _kDartVmSnapshotInstructions四部分。
Flutter_assets	1MB	7%	包括图片、字体、LICENSE等静态资源

2.1.2 Android 侧 Flutter 产物

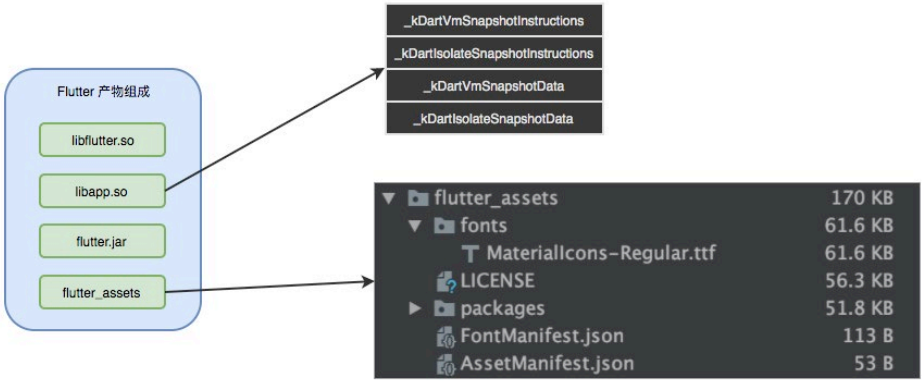


图 2　Flutter Android 产物组成示意图

Android 侧的 Flutter 产物总共 5.16MB，由四部分组成，表格 2 中列出了各部分的详细信息。

表 2　Flutter Android 产物组成

产物	大小	占比	介绍
libflutter.so	3.2MB	62%	Flutter引擎的C++代码编译产物
libapp.so	1.5MB	29%	Flutter业务与框架的Dart代码编译产物，内部由四部分组成
flutter.jar	310.5KB	5.9%	Flutter引擎的Java代码编译产物
flutter_assets	170KB	3.1%	包括图片、字体、LICENSE等静态资源

2.1.3 各部分产物的变化趋势

无论是 Android 还是 iOS，Flutter 的产物大体可以分为三部分：

1. Flutter 引擎，该部分大小固定不变，但初始占比较高。
2. Flutter 业务与框架，该部分大小随着 Flutter 业务代码的增多而逐渐增加。它是这样的一个曲线：初始增长速度极快，随着代码增多，增长速度逐渐减缓，最终趋近线性增长。原因是 Flutter 有一个 Tree Shaking 机制，从 Main 方法开始，逐级引用，最终没有被引用的代码，比如类和函数都会被裁剪掉。一开始引入 Flutter 之后随便写一个业务，就会大量用到 Flutter/Dart SDK 代码，这样初期 Flutter 包体积极速增加，但是过了一个临界点，用户包体积的增加就基本取决于 Flutter 业务代码增量，不会增长得太快。
3. Flutter 资源，该部分初始占比较小，后期增长主要取决于用到的本地图片资源的多少，增长趋势与资源多少成正比。

下图 3 展示了 Flutter 各资源变化的趋势：

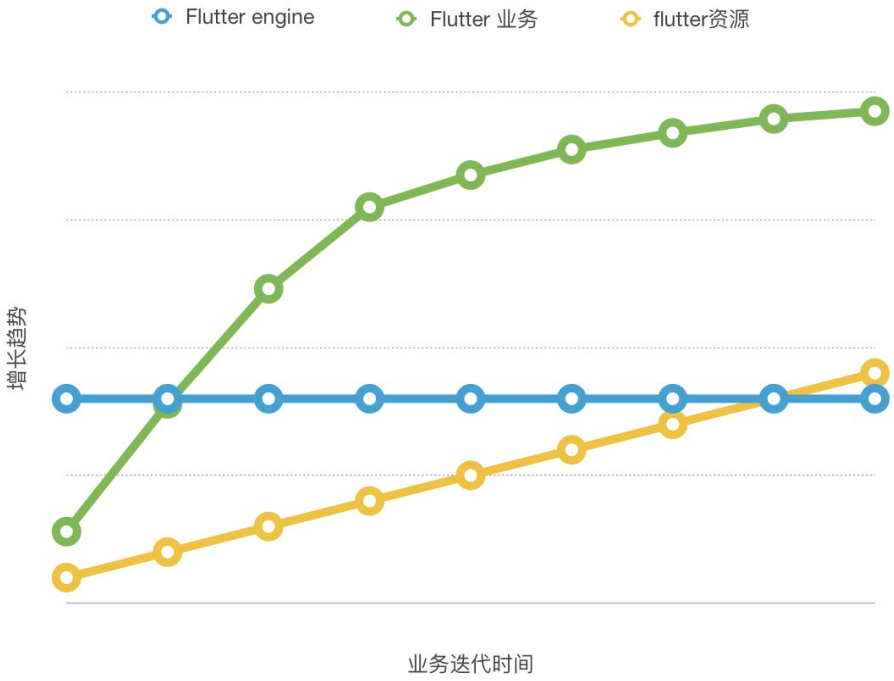


图 3　Flutter 各资源大小变化的趋势图

2.2 不同优化思路分析

上面我们对 Flutter 产物进行了分析，接下来看一下官方提供的优化思路如何应用于 Flutter 产物，以及对应的困难与收益如何。

1. 删减法

Flutter 引擎中包括了 Dart、skia、boringssl、icu、libpng 等多个模块，其中 Dart 和 skia 是必须的，其他模块如果用不到倒是可以考虑裁掉，能够带来几百 k 的瘦身收益。业务方可以根据业务诉求自定义裁剪。

Flutter 业务产物，因为 Flutter 的 Tree Shaking 机制，该部分产物从代码的角度已经是精简过的，要想继续精简只能从业务的角度去分析。

Flutter 资源中占比较多的一般是图片，对于图片可以根据业务场景，适当降低图片分辨率，或者考虑替换为网络图片。

2. 压缩法

因为无论是 Android 还是 iOS，安装包本身已经是压缩包了，对 Flutter 产物再次压缩的收益很低，所以该方法并不适用。

3. 动态下发

对于静态资源，理论上是 Android 和 iOS 都可以做到动态下发。而对于代码逻辑部分的编译产物，在 Android 平台支持可执行产物的动态加载，iOS 平台则不允许执行动态下发的机器指令。

经过上面的分析可以发现，除了删减、压缩，对所有业务适用、可行且收益明显的进一步优化空间重点在于动态下发了。能够动态下发的部分越多，包大小的收益越大。因此我们决定从动态下发入手来设计一套 Flutter 包大小优化方案。

三、基于动态下发的 Flutter 包大小优化方案

我们在 Android 和 iOS 上实现的包大小优化方案有所不同，区别在于 Android 侧可以做到 so 和 Flutter 资源的全部动态下发，而 iOS 侧由于系统限制无法动态下发可执行产物，所以需要对产物的组成和其加载逻辑进行分析，将其中非必须和动态链接库一起加载的部分进行动态下发、运行时加载。

当将产物动态下发后，还需要对引擎的初始化流程做修改，这样才能保证产物的正常加载。由于两端技术栈的不同，在很多具体实现上都采用了不同的方式，下面就分别来介绍下两端的方案。

3.1 iOS 侧方案

在 iOS 平台上，由于系统的限制无法实现在运行时加载并运行可执行文件，而在上文产物介绍中可以看到，占比较高的 App 及 Flutter 这两个均是可执行文件，理论上是不能进行动态下发的，实际上对于 Flutter 可执行文件我们能做的确实不多，但对于 App 这个可执行文件，其内部组成的四个模块并不是在链接时都必须存在的，可以考虑部分移出，进而来实现包体积的缩减。

因此，在该部分我们首先介绍 Flutter 产物的生成和加载的流程，通过对流程细节的分析来挖掘出产物可以被拆分出动态下发的部分，然后基于实现原理来设计实现工程化的方案。

3.1.1 实现原理简析

为了实现 App 的拆分，我们需要了解下 App.framework 是怎样生成以及各部分资源时如何加载的。如下图 4 所示，Dart 代码会使用 gen_snapshot 工具来编译成 .S 文件，然后通过 xcrun 工具来进行汇编和链接最终生成 App.framework。其中 gen_snapshot 是 Dart 编译器，采用了 Tree Shaking 等技术，用于生成汇编形式的机器代码。

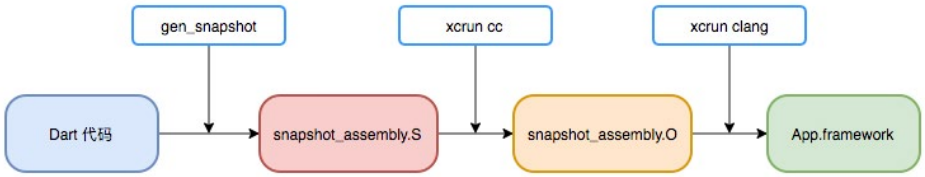


图 4　App.framework 生成流程示意图

产物加载流程：

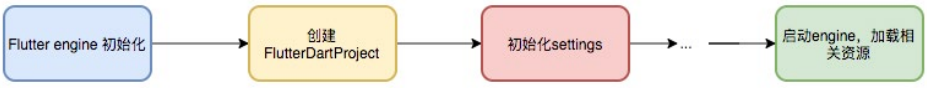


图 5　Flutter 产物加载流程图

如上图 5 所示，Flutter engine 在初始化时会从根据 FlutterDartProject 的 settings 中配置资源路径来加载可执行文件（App）、flutter_assets 等资源，具体 settings 的相关配置如下：

```
// settings
{
...
  // snapshot 文件地址或内存地址
```

```
  std::string vm_snapshot_data_path;
  MappingCallback vm_snapshot_data;
  std::string vm_snapshot_instr_path;
  MappingCallback vm_snapshot_instr;

  std::string isolate_snapshot_data_path;
  MappingCallback isolate_snapshot_data;
  std::string isolate_snapshot_instr_path;
  MappingCallback isolate_snapshot_instr;

  // library 模式下的 lib 文件路径
  std::string application_library_path;
  // icudlt.dat 文件路径
  std::string icu_data_path;
  // flutter_assets 资源文件夹路径
  std::string assets_path;
  //
...
}
```

以加载 vm_snapshot_data 为例，它的加载逻辑如下：

```
std::unique_ptr<DartSnapshotBuffer> ResolveVMData(const Settings&
settings) {
  // 从 settings.vm_snapshot_data 中取
  if (settings.vm_snapshot_data) {
    ...
  }

  // 从 settings.vm_snapshot_data_path 中取
  if (settings.vm_snapshot_data_path.size() > 0) {
    ...
  }
  // 从 settings.application_library_path 中取
  if (settings.application_library_path.size() > 0) {
    ...
  }

  auto loaded_process = fml::NativeLibrary::CreateForCurrentProcess();
  // 根据 kVMDataSymbol 从 native library 中加载
  return DartSnapshotBuffer::CreateWithSymbolInLibrary(
      loaded_process, DartSnapshot::kVMDataSymbol);
}
```

对于 iOS 来说，它默认会根据 kVMDataSymbol 来从 App 中加载对应资源，而其实 settings 是给提供了通过 path 的方式来加载资源和 snapshot 入口，那么对于

flutter_assets、icudtl.dat 这些静态资源，我们完全可以将其移出托管到服务端，然后动态下发。

而由于 iOS 系统的限制，整个 App 可执行文件则不可以动态下发，但在第二部分的介绍中我们了解到，其实 App 是由 kDartVmSnapshotData、kDartVmSnapshotInstructions、kDartIsolateSnapshotData、kDartIsolateSnapshotInstructions 等四个部分组成的，其中 kDartIsolateSnapshotInstructions、kDartVmSnapshotInstructions 为指令段，不可通过动态下发的方式来加载，而 kDartIsolateSnapshotData、kDartVmSnapshotData 为数据段，它们在加载时不存在限制。

到这里，其实我们就可以得到 iOS 侧 Flutter 包大小的优化方案：将 flutter_assets、icudtl.dat 等静态资源及 kDartVmSnapshotData、kDartIsolateSnapshotData 两部分在编译时拆分出去，通过动态下发的方式来实现包大小的缩减。但此方案有个问题，kDartVmSnapshotData、kDartIsolateSnapshotData 是在编译时就写入到 App 中了，如何实现自动化地把此部分拆分出去是一个待解决的问题。为了解决此问题，我们需要先了解 kDartVmSnapshotData、kDartIsolateSnapshotData 的写入时机。接下来，我们通过下图 6 来简单地介绍一下该过程：

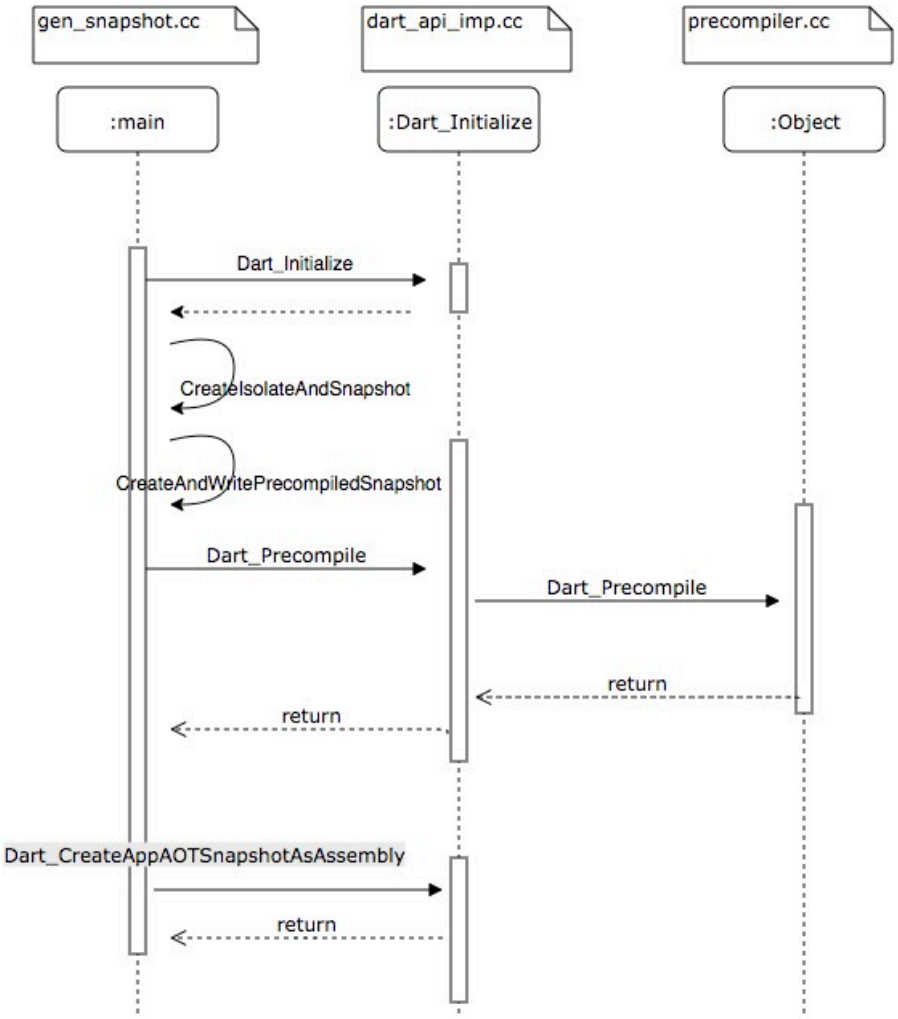


图 6　Flutter Data 段写入时序图

代码通过 gen_snapshot 工具来进行编译，它的入口在 gen_snapshot.cc 文件，通过初始化、预编译等过程，最终调用 Dart_CreateAppAOTSnapshotAsAssembly 方法来写入 snapshot。因此，我们可以通过修改此流程，在写入 snapshot 时只将 instructions 写入，而将 data 重定向输入到文件，即可实现 kDartVmSnapshot-Data、kDartIsolateSnapshotData 与 App 的分离。此部分流程示意图如下图 7 所示：

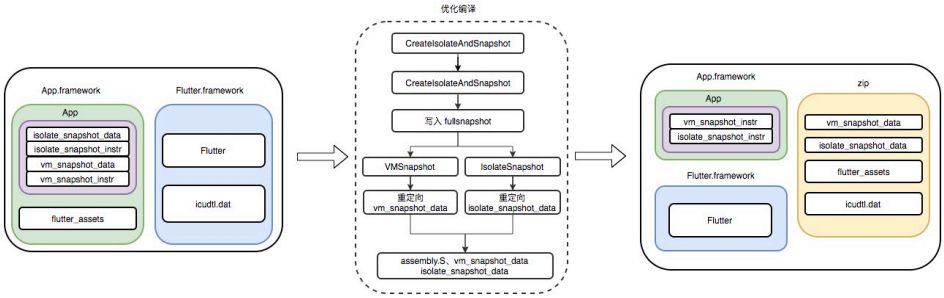


图 7　Flutter 产物拆分流程示意图

3.1.2 工程化方案

在完成了 App 数据段与代码段分离的工作后，我们就可以将数据段及资源文件通过动态下发、运行时加载的方式来实现包体积的缩减。由此思路衍生的 iOS 侧整体方案的架构如下图 8 所示；其中定制编译产物阶段主要负责定制 Flutter engine 及 Flutter SDK，以便完成产物的“瘦身”工作；发布集成阶段则为产物的发布和工程集成提供了一套标准化、自动化的解决方案；而运行阶段的使命是保证“瘦身”的资源在 engine 启动的时候能被安全稳定地加载。

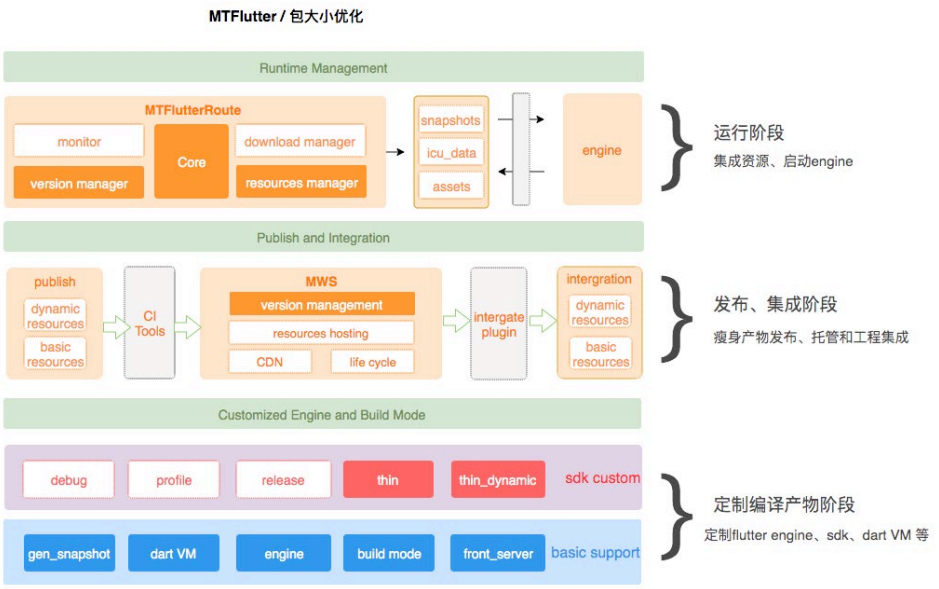


图 8　架构设计

注：图例中 MTFlutterRoute 为 Flutter 路由容器，MWS 指的是美团云。

3.1.2.1 定制编译产物阶段

虽然我们不能把 App.framework 及 Flutter.framework 通过动态下发的方式完全拆分出去，但可以剥离出部分非安装时必须的产物资源，通过动态下发的方式来达到 Flutter 包体积缩减的目的，因此在该阶段主要工作包括三部分。

1. 新增编译 command

在将 Flutter 包瘦身工程化时，我们必须保证现有的流程的编译规则不会被影响，需要考虑以下两点：

- 增加编译“瘦身”的 Flutter 产物构建模式，该模式应能编译出 AOT 模式下的瘦身产物。
- 不对常规的编译模式（debug、profile、release）引入影响。

对于 iOS 平台来说，AOT 模式 Flutter 产物编译的关键工作流程图如下图 9 所示。runCommand 会将编译所需参数及环境变量封装传递给编译后端（gen_snapshot 负责此部分工作），进而完成产物的编译工作：

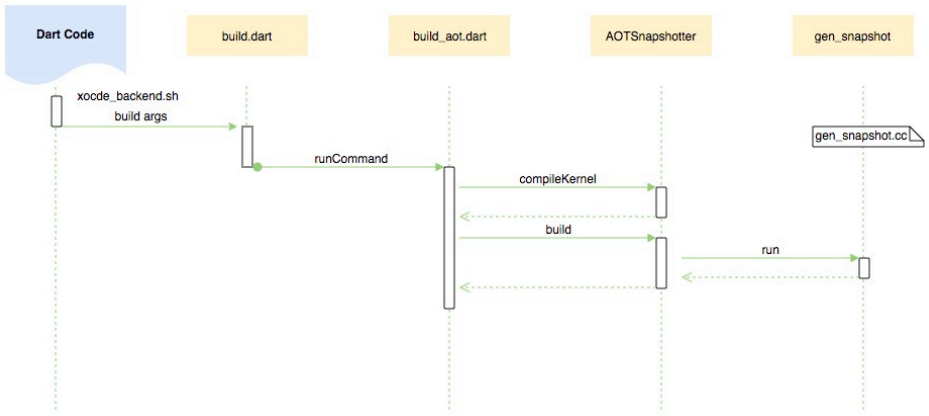


图 9　AOT 模式 Flutter 产物编译的关键工作流程图

为了实现“瘦身”的工作流，工具链在图 9 的流程中新增了 buildwithoutdata 的编译 command，该命令针对通过传递相应参数（without-data=true）给到编译后端（gen_snapshot），为后续编译出剥离 data 段提供支撑：

```
if [[ $# == 0 ]]; then
  # Backwards-compatibility: if no args are provided, build.
  BuildApp
else
  case $1 in
    "build")
      BuildApp ;;
    "buildWithoutData")
      BuildAppWithoutData ;;
    "thin")
      ThinAppFrameworks ;;
    "embed")
      EmbedFlutterFrameworks ;;
  esac
fi
```

```
..addFlag('without-data',
        negatable: false,
        defaultsTo: false,
        hide: true,
  )
```

2. 编译后端定制

该部分主要对 gen_snapshot 工具进行定制，当 gen_snapshot 工具在接收到 Dart 层传来的“瘦身”命令时，会解析参数并执行我们定制的方法 Dart_CreateAppAOTSnapshotAsAssembly，该部分主要做了两件事：

1. 定制产物编译过程，生成剥离 data 段的编译产物。
2. 重定向 data 段到文件中，以便后续进行使用。

具体到处理的细节，首先我们需要在 gen_sanpshot 的入口处理传参，并指定重定向 data 文件的地址：

```
  CreateAndWritePrecompiledSnapshot() {
    ...
```

```
    if (snapshot_kind == kAppAOTAssembly) { // 常规 release 模式下产物的编译流
程
      ...
    } else if (snapshot_kind == kAppAOTAssemblyDropData) {
      ...
      result = Dart_
CreateAppAOTSnapshotAsAssembly(StreamingWriteCallback,
                                                    file,
                                                    &vm_snapshot_data_
buffer,
                                                    &vm_snapshot_data_
size,
                                                    &isolate_snapshot_
data_buffer,
                                                    &isolate_snapshot_
data_size,
                                                    true); // 定制产物编译过
程，生成剥离 data 段的编译产物 snapshot_assembly.S
      ...
    } else if (...) {
      ...
    }
    ...
  }
```

在接受到编译“瘦身”模式的命令后，将会调用定制的 FullSnapshotWriter 类来实现 Snapshot_assembly.S 的生成，该类会将原有编译过程中 vm_snapshot_data、isolate_snapshot_data 的写入过程改写成缓存到 buff 中，以便后续写入到独立的文件中：

```
// drop_data=true, 表示后瘦身模式的编译过程
// vm_snapshot_data_buffer、isolate_snapshot_data_buffer 用于保存 vm_
snapshot_data、isolate_snapshot_data 以便后续写入文件
Dart_CreateAppAOTSnapshotAsAssembly(Dart_StreamingWriteCallback callback,
                                    void* callback_data,
                                    bool drop_data,
                                    uint8_t** vm_snapshot_data_buffer,
                                    uint8_t** isolate_snapshot_data_
buffer) {
  ...
  FullSnapshotWriter writer(Snapshot::kFullAOT, &vm_snapshot_data_
buffer,
                            &isolate_snapshot_data_buffer,
ApiReallocate,
                            &image_writer, &image_writer);
```

```
  if (drop_data) {
    writer.WriteFullSnapshotWithoutData(); // 分离出数据段
  } else {
    writer.WriteFullSnapshot();
  }
  ...
}
```

当 data 段被缓存到 buffer 中后，便可以使用 gen_snapshot 提供的文件写入的方法 WriteFile 来实现数据段以文件形式从编译产物中分离：

```
static void WriteFile(const char* filename, const uint8_t* buffer,
const intptr_t size);
// 写 data 到指定文件中
{
  ...
      WriteFile(vm_snapshot_data_filename, vm_snapshot_data_buffer, vm_
snapshot_data_size); // 写入 vm_snapshot_data
      WriteFile(isolate_snapshot_data_filename, isolate_snapshot_data_
buffer, isolate_snapshot_data_size); // 写入 isolate_snapshot_data
  ...
}
```

3. engine 定制

编译参数修改

iOS 侧使用 -Oz 参数可以获得包体积缩减的收益（大约为 700KB 左右的收益），但会有相应的性能损耗，因此该部分作为一个可选项提供给业务方，工具链提供相应版本的 Flutter engine 的定制。

资源加载方式定制

对于 engine 的定制，主要围绕如何“手动”引入拆分出的资源来展开，好在 engine 提供了 settings 接口让我们可以实现自定义引入文件的 path，因此我们需要做的就是对 Flutter engine 初始化的过程进行相应改造：

```
/**
 * custom icudtl.dat path
```

```
 */
@property(nonatomic, copy) NSString* icuDataPath;

/**
 * custom flutter_assets path
 */
@property(nonatomic, copy) NSString* assetPath;

/**
 * custom isolate_snapshot_data path
 */
@property(nonatomic, copy) NSString* isolateSnapshotDataPath;

/**
 *custom vm_snapshot_data path
 */
@property(nonatomic, copy) NSString* vmSnapshotDataPath;
```

在运行时“手动”配置上述路径，并结合上述参数初始化 FlutterDartProject，从而达到 engine 启动时从配置路径加载相应资源的目的。

engine 编译自动化

在完成 engine 的定制和改造后，还需要手动编译一下 engine 源码，生成各平台、架构、模式下的产物，并将其集成到 Flutter SDK 中，为了让引擎定制的流程标准化、自动化，MTFlutter 工具链提供了一套 engine 自动化编译发布的工具。如流程图 10 所示，在完成 engine 代码的自定义修改之后，工具链会根据 engine 的 patch code 编译出各平台、架构及不同模式下的 engine 产物，然后自动上传到美团云上，在开发和打包时只需要通简单的命令，即可安装和使用定制后的 Flutter engine：

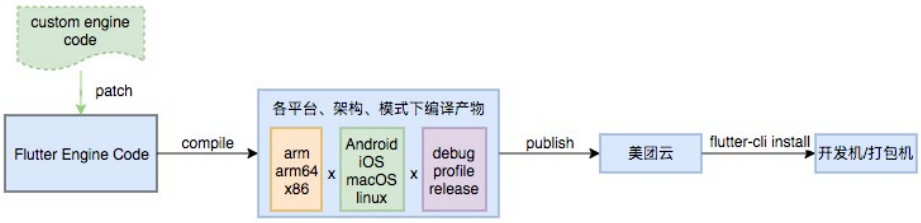


图 10　Flutter engine 自动化编译发布流程

3.1.2.2 发布集成阶段

当完成 Dart 代码编译产物的定制后，我们下一步要做的就是改造 MTFlutter 工具链现有的产物发布流程，支持打出“瘦身”模式的产物，并将瘦身模式下的产物进行合理的组织、封装、托管以方便产物的集成。从工具链的视角来看，该部分的流程示如下图 11 所示：

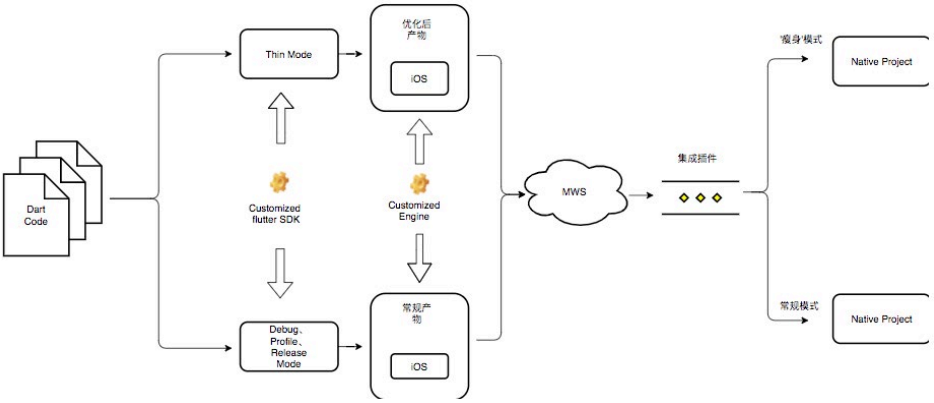


图 11　Flutter 产物发布集成流程示意图

自动化发布与版本管理

MTFlutter 工具链将“瘦身”集成到产物发布的流水线中，新增一种 thin 模式下的产物，在 iOS 侧该产物包括 release 模式下瘦身后的 App.framework、Flutter.framework 以及拆分出的数据、资源等文件。当开发者提交了代码并使用 Talos（美团内部前端持续交付平台）触发 Flutter 打包时，CI 工具会自动打出瘦身的产物包及需要运行时下载的资源包、生成产物相关信息的校验文件并自动上传到美团云上。对于产物资源的版本管理，我们则复用了美团云提供资源管理的能力。在美团云上，产物资源以文件目录的形式来实现各版本资源的相互隔离，同时对“瘦身”资源单独开一个 bucket 进行单独管理，在集成产物时，集成插件只需根据当前产物 module 的名称及版本号便可获取对应的产物。

自动化集成

针对瘦身模式 MTFlutter 工具链对集成插件也进行了相应的改造，如下图 12 所示。我们对 Flutter 集成插件进行了修改，在原有的产物集成模式的基础上新增一种 thin 模式，该模式在表现形式与原有的 debug、release、profile 类似，区别在于：为了方便开发人员调试，该模式会依据当前工程的 buildconfigration 来做相应的处理，即在 debug 模式下集成原有的 debug 产物，而在 release 模式下才集成“瘦身”产物包。

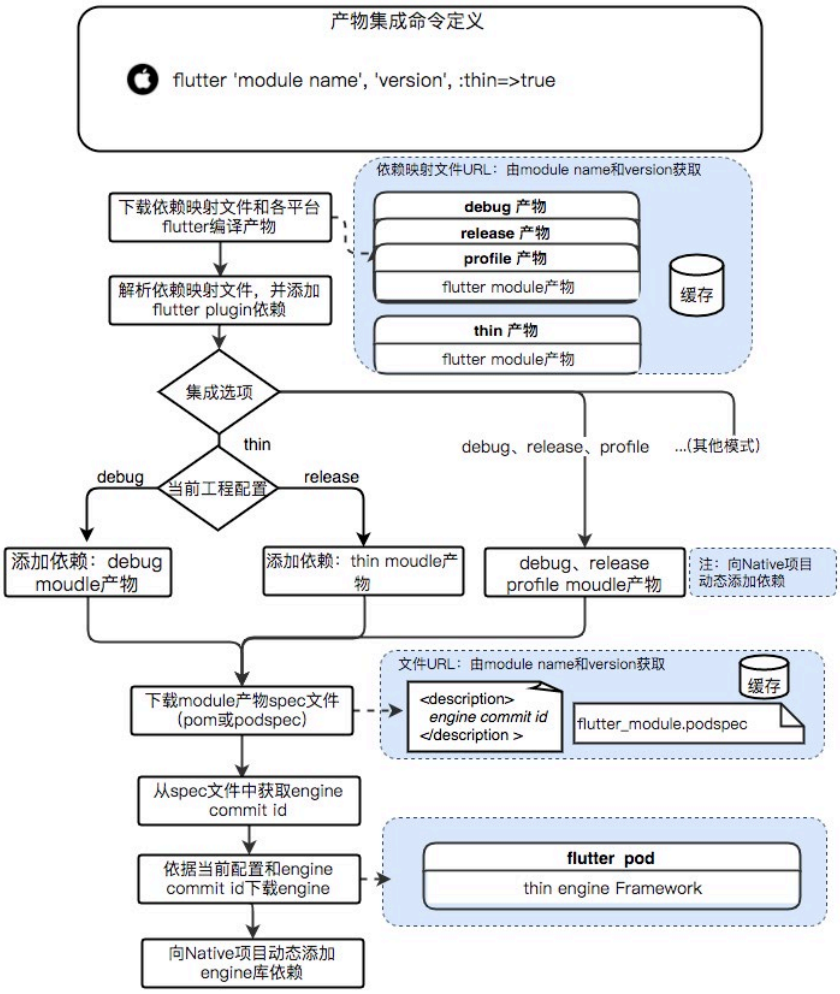


图 12　Flutter iOS 端集成插件修改

3.1.2.3 运行阶段

运行阶段所处理的核心问题包括资源下载、缓存、解压、加载及异常监控等。一个典型的瘦身模式下的 engine 启动的过程如图 13 所示。

该过程包括：

- **资源下载**：读取工程配置文件，得到当前 Flutter module 的版本，并查询和下载远程资源。
- **资源解压和校验**：对下载资源进行完整性校验，校验完成则进行解压和本地缓存。
- **启动 engine**：在 engine 启动时加载下载的资源。
- **监控和异常处理**：对整个流程可能出现的异常情况进行处理，相关数据情况进行监控上报。

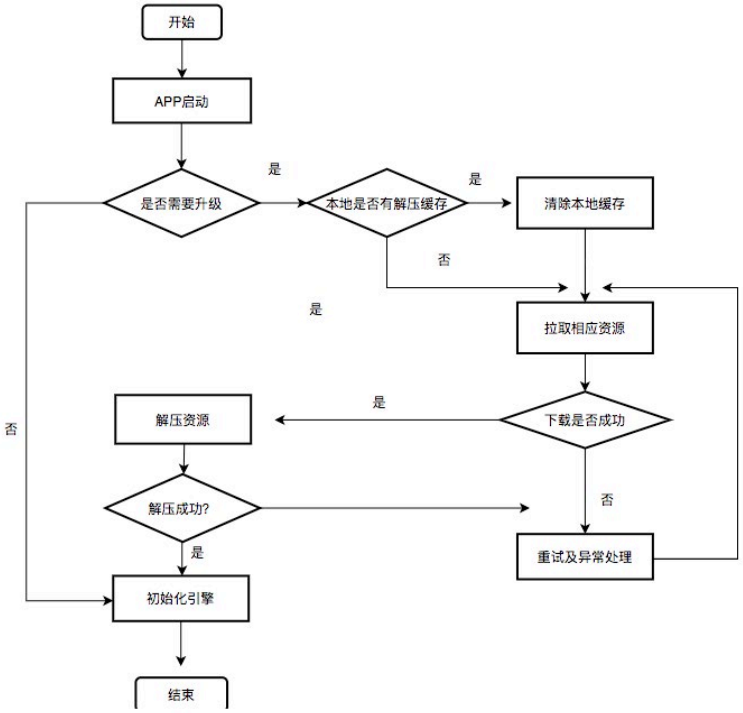


图 13　iOS 侧瘦身模式下 engine 启动流程图

为了方便业务方的使用、减少其接入成本，MTFlutter 将该部分工作集成至 MTFlutterRoute 中，业务方仅需引入 MTFlutterRoute 即可将“瘦身”功能接入到项目中。

3.2 Android 侧方案

3.2.1 整体架构

在 Android 侧，我们做到了除 Java 代码外的所有 Flutter 产物都动态下发。完整的优化方案概括来说就是：动态下发 + 自定义引擎初始化 + 自定义资源加载。方案整体分为打包阶段和运行阶段，打包阶段会将 Flutter 产物移除并生成瘦身的 APK，运行阶段则完成产物下载、自定义引擎初始化及资源加载。其中产物的上传和下载由 DynLoader 完成，这是由美团平台迭代工程组提供的一套 so 与 assets 的动态下发框架，它包括编译时和运行时两部分的操作：

1. 工程配置：配置需要上传的 so 和 assets 文件。
2. App 打包时，会将配置 1 中的文件压缩上传到动态发布系统，并从 APK 中移除。
3. App 每次启动时，向动态发布系统发起请求，请求需要下载的压缩包，然后下载到本地并解压，如果本地已经存在了，则不进行下载。

我们在 DynLoader 的基础上，通过对 Flutter 引擎初始化及资源加载流程进行定制，设计了整体的 Flutter 包大小优化方案：

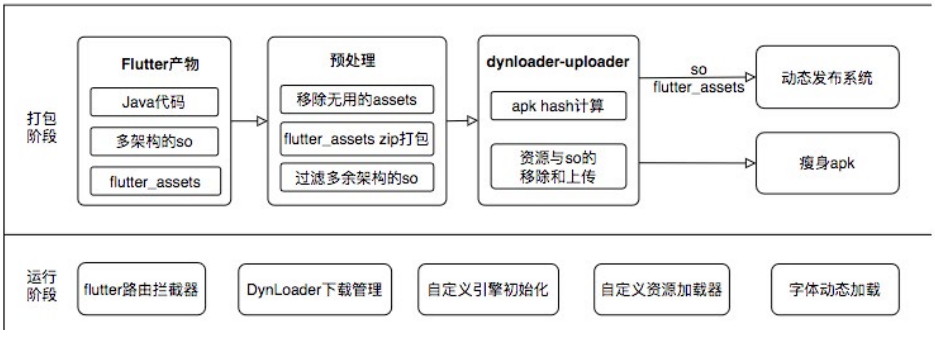


图 14　Android 侧 Flutter 包大小优化方案整体架构

打包阶段：我们在原有的 APK 打包流程中，加入一些自定义的 gradle plugin 来对 Flutter 产物进行处理。在预处理流程，我们将一些无用的资源文件移除，然后将 flutter_assets 中的文件打包为 bundle.zip。然后通过 DynLoader 提供的上传插件将 libflutter.so、libapp.so 和 flutter_assets/bundle.zip 从 APK 中移除，并上传到动态发布系统托管。其中对于多架构的 so，我们通过在 build.gradle 中增加 abiFilters 进行过滤，只保留单架构的 so。最终打包出来的 APK 即为瘦身后的 APK。

不经处理的话，瘦身后的 APK 一进到 Flutter 页面肯定会报错，因为此时 so 和 flutter_assets 可能都还没下载下来，即使已经下载下来，其位置也发生了改变，再使用原来的加载方式肯定会找不到。所以我们在运行阶段需要做一些特殊处理：

1. Flutter 路由拦截

首先要使用 Flutter 路由拦截器，在进到 Flutter 页面之前，要确保 so 和 flutter_assets 都已经下载完成，如果没有下载完，则显示 loading 弹窗，然后调用 DynLoader 的方法去异步下载。当下载完成后，再执行原来的跳转逻辑。

2. 自定义引擎初始化

第一次进到 Flutter 页面，需要先初始化 Flutter 引擎，其中主要是将 libflutter.so 和 libapp.so 的路径改为动态下发的路径。另外还需要将 flutter_assets/bundle.zip 进行解压。

3. 自定义资源加载

当引擎初始化完成后，开始执行 Dart 代码的逻辑。此时肯定会遇到资源加载，比如字体或者图片。原有的资源加载器是通过 method channel 调用 AssetManager 的方法，从 APK 中的 assets 中进行加载，我们需要改成从动态下发的路径中加载。

下面我们详细介绍下某些部分的具体实现。

3.2.2 自定义引擎初始化

原有的 Flutter 引擎初始化由 FlutterMain 类的两个方法完成，分别为 startIni-

tialization 和 ensureInitializationComplete，一般在 Application 初始化时调用 startInitialization（懒加载模式会延迟到启动 Flutter 页面时再调用），然后在 Flutter 页面启动时调用 ensureInitializationComplete 确保初始化的完成。

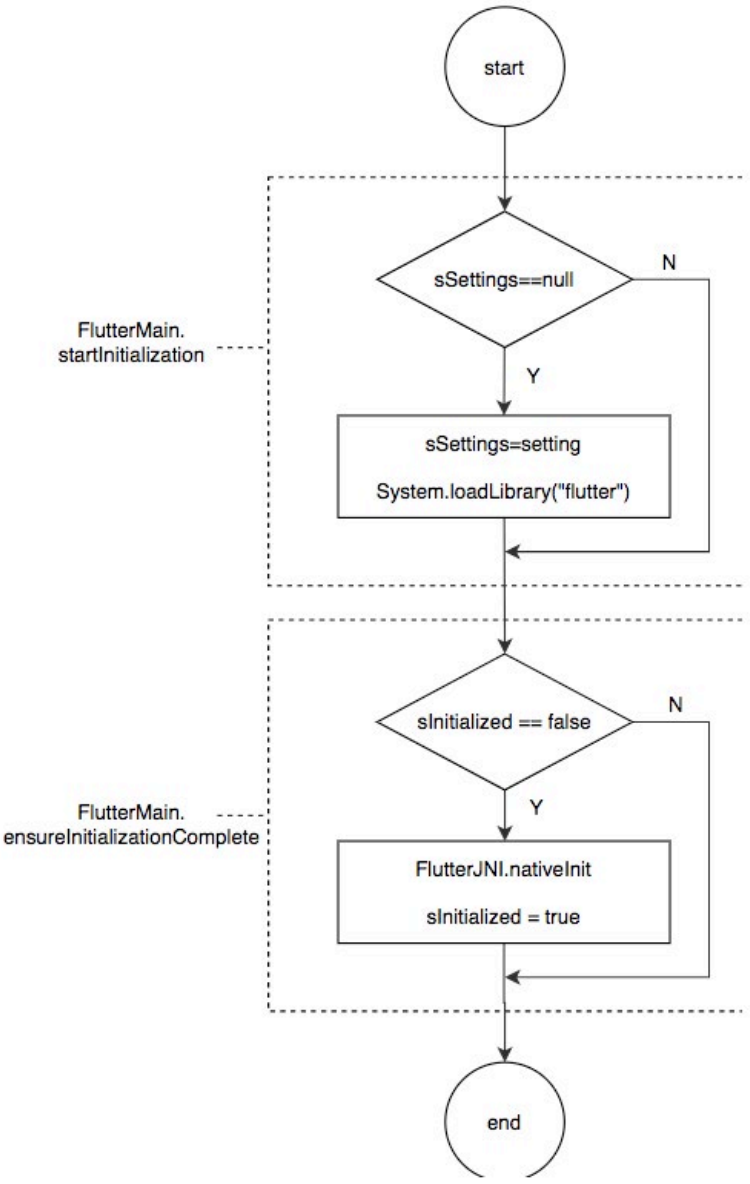


图 15　Android 侧 Flutter 引擎初始化流程图

在 startInitialization 方法中，会加载 libflutter.so，在 ensureInitializationComplete 中会构建 shellArgs 参数，然后将 shellArgs 传给 FlutterJNI.nativeInit 方法，由 jni 侧完成引擎的初始化。其中 shellArgs 中有个参数 AOT_SHARED_LIBRARY_NAME 可以用来指定 libapp.so 的路径。

自定义引擎初始化，主要要修改两个地方，一个是 System.loadLibrary(“flutter”)，一个是 shellArgs 中 libapp.so 的路径。有两种办法可以做到：

1. 直接修改 FlutterMain 的源码，这种方式简单直接，但是需要修改引擎并重新打包，业务方也需要使用定制的引擎才可以。
2. 继承 FlutterMain 类，重写 startInitialization 和 ensureInitializationComplete 的逻辑，让业务方使用我们的自定义类来初始化引擎。当自定义类完成引擎的初始化后，通过反射的方式修改 sSettings 和 sInitialized，从而使得原有的初始化逻辑不再执行。

本文使用第二种方式，需要在 FlutterActivity 的 onCreate 方法中首先调用自定义的引擎初始化方法，然后再调用 super 的 onCreate 方法。

3.2.3 自定义资源加载

Flutter 中的资源加载由一组类完成，根据数据源的不同分为了网络资源加载和本地资源加载，其类图如下：

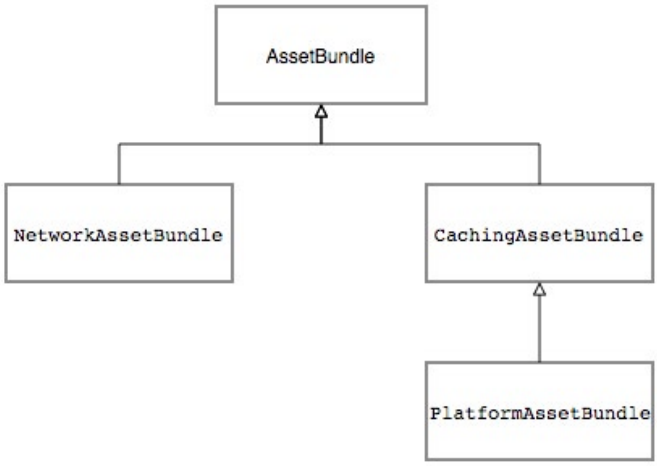


图 16　Flutter 资源加载相关类图

AssetBundle 为资源加载的抽象类，网络资源由 NetworkAssetBundle 加载，打包到 Apk 中的资源由 PlatformAssetBundle 加载。

PlatformAssetBundle 通过 channel 调用，最终由 AssetManager 去完成资源的加载并返回给 Dart 层。

我们无法修改 PlatformAssetBundle 原有的资源加载逻辑，但是我们可以自定义一个资源加载器对其进行替换：在 widget 树的顶层通过 DefaultAssetBundle 注入。

自定义的资源加载器 DynamicPlatformAssetBundle，通过 channel 调用，最终从动态下发的 flutter_assets 中加载资源。

3.2.4 字体动态加载

字体属于一种特殊的资源，其有两种加载方式：

1. **静态加载**：在 pubspec.yaml 文件中声明的字体及为静态加载，当引擎初始化的时候，会自动从 AssetManager 中加载静态注册的字体资源。
2. **动态加载**：Flutter 提供了 FontLoader 类来完成字体的动态加载。

当资源动态下发后，assets 中已经没有字体文件了，所以静态加载会失败，我们需要改为动态加载。

3.2.5 运行时代码组织结构

整个方案的运行时部分涉及多个功能模块，包括产物下载、引擎初始化、资源加载和字体加载，既有 Native 侧的逻辑，也有 Dart 侧的逻辑。如何将这些模块合理的加以整合呢？平台团队的同学给了很好的答案，并将其实现为一个 Flutter Plugin：flutter_dynamic（美团内部库）。其整体分为 Dart 侧和 Android 侧两部分，Dart 侧提供字体和资源加载方法，方法内部通过 method channel 调到 Android 侧，在 Android 侧基于 DynLoader 提供的接口实现产物下载和资源加载的逻辑。

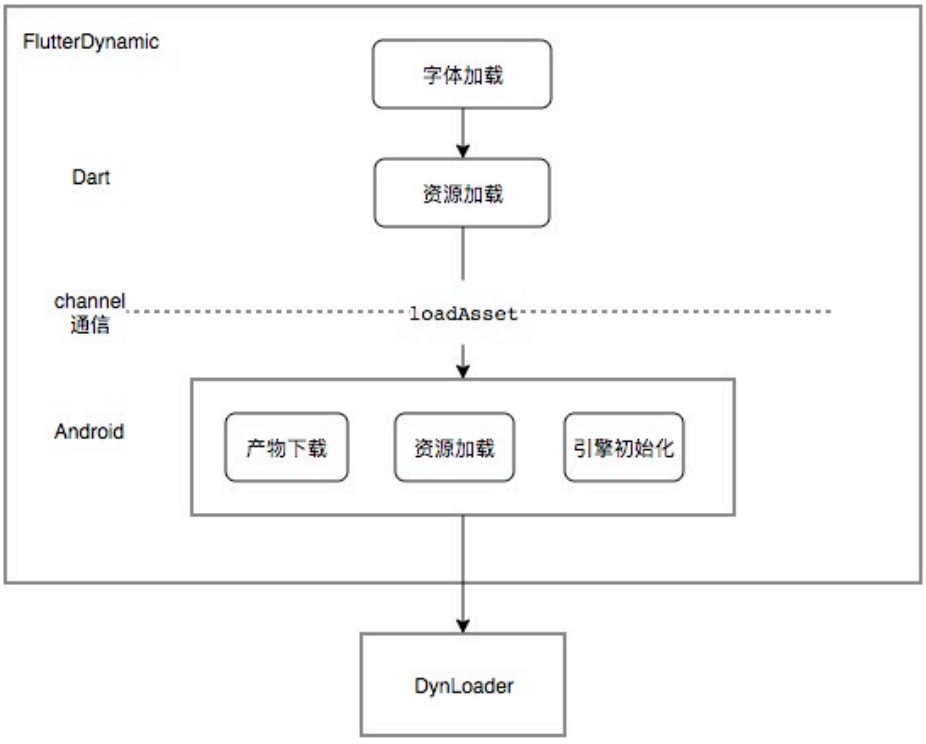


图 17　FlutterDynamic 结构图

四、方案的接入与使用

为了让大家了解上述方案使用层面的设计，我们在此把美团内部的使用方式介绍给大家，其中会涉及到一些内部工具细节我们暂不展开，重点解释设计和使用体验部分。由于 Android 和 iOS 的实现方案有所区别，故在接入方式相应的也会有些差异，下面针对不同平台分开来介绍：

4.1 iOS

在上文方案的设计中，我们介绍到包瘦身功能已经集成进入美团内部 MTFlutter 工具链中，因此当业务方在使用了 MTFlutter 后只需简单的几步配置便可实现包瘦身功能的接入。iOS 的接入使用上总体分为三步：

1. 引入 Flutter 集成插件（cocoapods-flutter-plugin 美团内部 Cocoapods 插件，进一步封装 Flutter 模块引入，使之更加清晰便捷）：

```
gem 'cocoapods-flutter-plugin', '~> 1.2.0'
```

2. 接入 MTFlutterRoute 混合业务容器（美团内部 pod 库，封装了 Flutter 初始化及全局路由等能力），实现基于“瘦身”产物的初始化：

Flutter 业务工程中引入 mt_flutter_route：

```
dependencies:
  mt_flutter_route: ^2.4.0
```

3. 在 iOS Native 工程中引入 MTFlutterRoute pod：

```
binary_pod 'MTFlutterRoute', '2.4.1.8'
```

经过上面的配置后，正常 Flutter 业务发版时就会自动产生“瘦身”后的产物，此时只需在工程中配置瘦身模式即可完成接入：

```
flutter 'your_flutter_project', 'x.x.x', :thin => true
```

4.2 Android

4.2.1 Flutter 侧修改

在 Flutter 工程 pubspec.yaml 中添加 flutter_dynamic（美团内部 Flutter Plugin，负责 Dart 侧的字体、资源加载）依赖。

在 main.dart 中添加字体动态加载逻辑，并替换默认资源加载器。

```
void main() async {
   // 动态加载字体
  await dynFontInit();
  // 自定义资源加载器
  runApp(DefaultAssetBundle(
    bundle: dynRootBundle,
    child: MyApp(),
  ));
}
```

4.2.2 Native 侧修改

1. 打包脚本修改

在 App 模块的 build.gradle 中通过 apply 特定 plugin 完成产物的删减、压缩以及上传。

2. 在 Application 的 onCreate 方法中初始化 FlutterDynamic。

3. 添加 Flutter 页面跳转拦截。

在跳转到 Flutter 页面之前，需要使用 FlutterDynamic 提供的接口来确保产物已经下载完成，在下载成功的回调中来执行真正的跳转逻辑。

```
class FlutterRouteUtil {
    public static void startFlutterActivity(final Context context,
Intent intent) {
        FlutterDynamic.getInstance().ensureLoaded(context, new
LoadCallback() {
            @Override
            public void onSuccess() {
              // 在下载成功的回调中执行跳转逻辑
                context.startActivity(intent);
```

```
            }
        });
    }
}
```

备注：如果 App 有使用类似 WMRoute 之类的路由组件的话，可以自定义一个 UriHandler 来统一处理所有的 Flutter 页面跳转，同样在 ensureLoaded 方法回调中执行真正的跳转逻辑。

4. 添加引擎初始化逻辑

我们需要重写 FlutterActivity 的 onCreate 方法，在 super.onCreate 之前先执行自定义的引擎初始化逻辑。

```
public class MainFlutterActivity extends FlutterActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState)
      // 确保自定义引擎初始化完成
        FlutterDynamic.getInstance().ensureFlutterInit(this);
        super.onCreate(savedInstanceState);
    }
}
```

五、总结展望

目前，动态下发的方案已在美团内部 App 上线使用，Android 包瘦身效果到达 95%，iOS 包瘦身效果达到 30%+。动态下发的方案虽然能显著减少 Flutter 的包体积，但其收益是通过运行时下载的方式置换回来的。当 Flutter 业务的不断迭代增长时，Flutter 产物包也会随之不断变大，最终导致需下载的产物变大，也会对下载成功率带来压力。后续，我们还会探索 Flutter 的分包逻辑，通过将不同的业务模块拆分来降低单个产物包的大小，来进一步保障包瘦身功能的可用性。

六、作者简介

艳东，2018 年加入美团，到家平台前端工程师。
宗文，2019 年加入美团，到家平台前端高级工程师。
会超，2014 年加入美团，到家平台前端技术专家。

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家。欢迎感兴趣的同学投递简历至：tech@meituan.com（邮件标题请注明：美团外卖技术团队）。

美团外卖持续交付的前世今生

作者：晓飞　王鹏　江伟

0. 前言

美团外卖自 2013 年创建以来，业务一直在高速发展，目前日订单量已突破 3000 万单，已成为美团点评最重要的业务之一。美团外卖所承载的业务，从早期单一的美食业务发展成为了外卖平台业务。目前除餐饮业务外，闪购、跑腿、闪付、营销、广告等产品形态的业务也陆续在外卖平台上线。参与到美团外卖平台的业务团队，也从早期的单一的外卖团队发展成为多业务团队。每个业务团队虽然都有不同的业务形态，但是几乎都有相同的诉求：需求能不能尽快地上线?

然而，Native 应用的发布依赖于应用市场的更新，周期非常长，非常不利于产品的快速迭代、快速试错。同时，作为平台方，我们需要考虑到各个业务团队的诉求，统筹考虑如何建立怎么样的模型、配套什么样的技术手段，才能实现最佳的状态，满足各业务更短周期、高质量的交付业务的诉求。本文会首先回顾美团外卖从早期的月交付，逐渐演变成双周交付，再从双周交付演变成双周版本交付配合周动态交付的过程。然后从外卖的历史实践中，浅谈一个好的持续交付需要综合考虑哪些关键因素，希望对大家有所帮助或启发。

1. 交付模型

一个需求从产生到交付再到用户的手上，要经历需求调研、需求分析、程序设计、代码开发、测试、部署上线等多个环节。在整个过程中，由于涉及到不同角色的人员，而不同角色人员的认知存在差异，不同的程序语言存在差异，不同的开发方式也存在差异。在整个交付需求的过程中，还面临着需求可能会被变更、交付周期可能会被变更等各种情况。这些情况使得需求的交付变得非常复杂、不可预期。而软件开发的首要任务就是持续、尽早地交付有价值的软件。怎么解决这一问题，是软件工程一直在

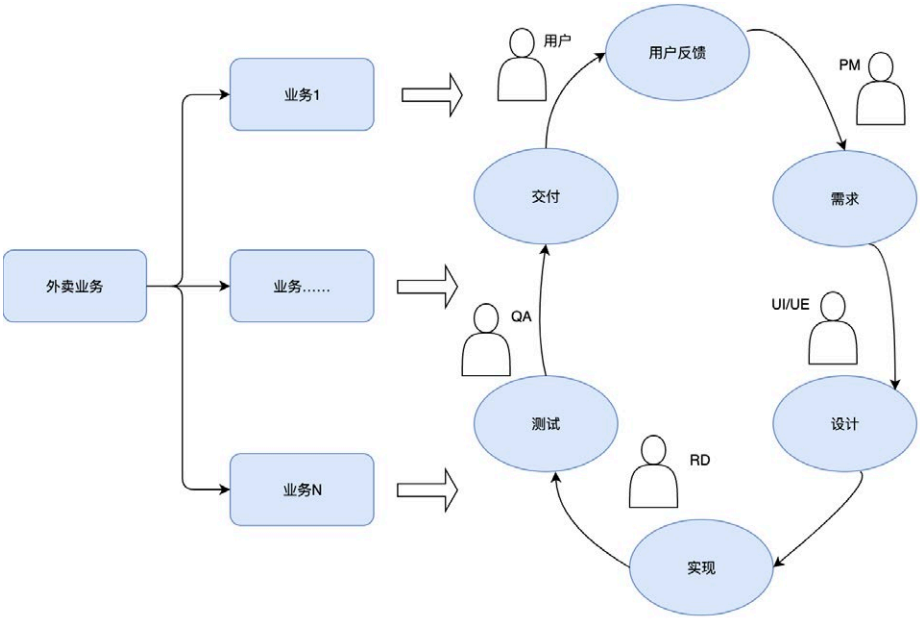
研究的话题。早在 20 世纪 50 年代，软件领域就在积极地探索设计什么样的模型可以解决这些问题。常见的包括瀑布流模型、迭代模型、螺旋模型、敏捷模型等等。由于篇幅原因，本文不再做详细的介绍。

2. 什么是持续交付

关注持续交付，不同的企业、不同的团队站在不同的角度存在不同的定义。《持续交付 2.0：业务引领的 DevOps 精要》一书认为，站在企业的角度，将持续交付定位为一个产品价值的开发框架，是一个工具集，其中包含了一系列的原则和众多实践，帮助提升企业的内部运转速度和交付效率。

从知乎话题“如何理解持续集成、持续交付、持续部署”，我们可以看到大部分研发团队，会从软件研发的角度进行定义，他们将软件的开发步骤拆解为持续集成、持续交付、持续部署，其中持续集成指开发人员从编码到构建的过程；持续交付指将已经集成构建完成的代码，交付给测试团队进行测试的过程；持续部署指将测试通过的软件交付给用户的过程。而产品团队会站在产品的角度来看，他们认为持续交付，是从需求的 PRD 文档提出来，到用户能够感受到这个需求的周期。

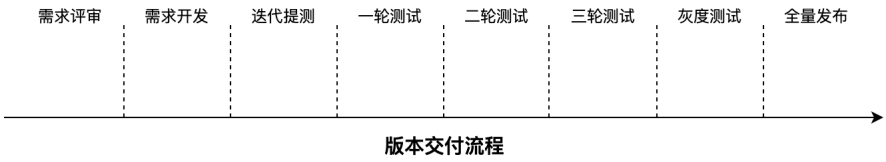
从美团外卖业务的角度，我们可以将持续交付定位为“外卖用户”和“外卖团队”之间的紧密反馈流。而外卖团队涉及到 PM、UI/UE、RD 和 QA 角色。如下图所示，产品同学从市场或用户的需求和反馈中收集到需求，转换为需求 PRD 文档，交由设计交互同学设计成期望用户所见的界面和行为，然后交给研发团队进行调研实现。研发团队实现后，再交给测试团队进行测试。等测试团队完成测试后，提交到应用市场，最终交付到用户手上，这个过程是本文所考虑的交付。“持续”指的是，外卖团队将外卖的业务拆分成不同维度的子业务，每个子业务持续通过这个迭代流程不断地优化各个子业务达到最优，从而使整个外卖业务达到最优。



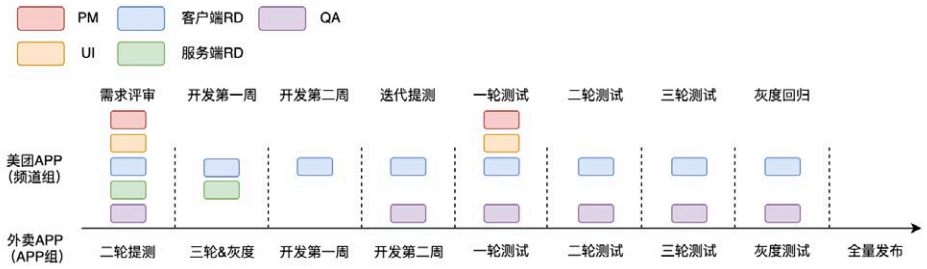
3. 外卖持续交付模型的变迁史

3.1 外卖的早期交付模型

早期外卖的交付模式为“串行交付”，一个版本完成后，开启下一个版本。整个交付过程包括需求评审、需求开发、迭代提测、一轮提测、二轮提测、三轮提测、灰度提测和全量发布 8 个环节：



由于美团的外卖业务是一个双平台业务，需要同时开发外卖 App 和美团 App，在人力安排上，我们会分成两个组，App 组和频道组并行开发。App 组开发当前的版本业务，频道组同步上个版本的业务到当前的美团平台版本。整个版本周期耗时六周半左右，遇到节假日会顺延，全年能发版 11 ~ 12 个，版本交付周期如下图所示：



双端APP的交付周期

交付模型中关键点

1. 需求评审分为主打需求、非主打需求和统计需求。
   - 主打需求：客户端开发 10d 之前进行评审，评审 8d 后 UI 提供初稿，评审 14d 后 UI 定稿。
   - 非主打需求：两个三方评审窗口，窗口 1 为窗口 2 的前 5d，窗口 2 为上一版本二轮测试地一天，UI 定稿需要在客户端开发第一周内完成。
2. 统计需求：需求收集截止到立项评审前一天，三方评审为非主打需求评审后第二天。服务器在需求评审后，客户端开发第一周前，提供接口文档。
3. 客户端开发周期为两周，客户端开发第 5 个自然日 API 提测，客户端开发第 5 天发迭代提测包，第二周结束发提测包。
4. QA 验收提测包，一轮 3d、二轮 3d、三轮 1d。
5. 灰度 3d，需避开节假日前发灰度。
6. 全量发布电子市场。

单月发版优缺点

优点：

1. 每个版本可根据当版的情况控制版本的周期，可同时支持多个大需求的并行开发。
2. 版本较为固定，各角色的分工明确，开发人员在开发期较为专注，开发效率高。
3. 测试周期长，App 能够得到充分的测试。

缺点：

1. 版本迭代周期较长，从需求评审开始到需求灰度，需要 6.5 周，无法满足日益增长的业务诉求，对于一些尝试性项目，无法满足 PM 急迫上线验证效果的需求。
2. 发版频率低，每年只有 11~12 个版本，对标其他互联网公司，业务产出相对较低。
3. 每个角色的利用率不高，如 RD 在开发期的时候，QA 处在待测试状态。

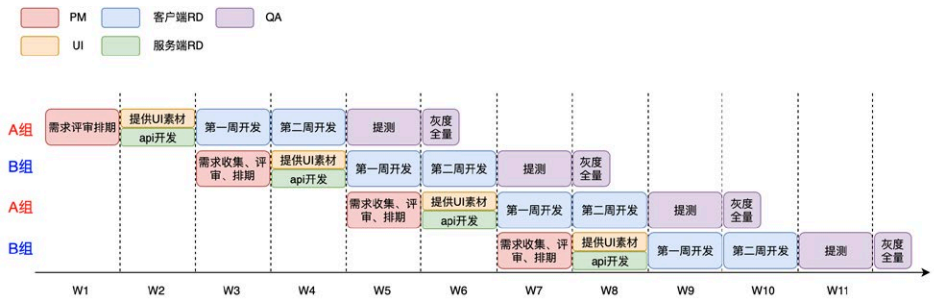
3.2 双周迭代的交付模型实践

为了满足日益增长的业务诉求，我们提出了双周发版的交付模式，从而实现业务的快速迭代。贯彻的总体原则为：评审、开发、测试完全并行，以两周为固定周期，以需求维度持续交付。

3.2.1 双周迭代模型

在切换双周迭代模式之前，需要落地一些准备工作，例如统一版本排期的时间表、确定项目中各个角色的交付时间，让 PM、UI、客户端 RD、服务端 RD 和 QA，各角色都能够清晰的知道自己接入的时间点，并在规定的时间点交付内容。由于各角色的任务和分工得到了明确，减少了协作中的沟通成本，各角色也可以如齿轮一样，持续产出交付给下游团队，整个交付周期效率得到了有效的提升，一个版本的周期缩短为 5.5 周，发版次数上升到 22~24 版本。

版本各个角色分工详情示意图如下所示：



双周模型中关键点：

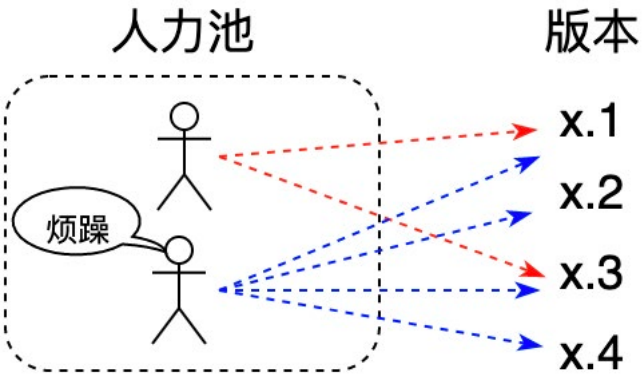
- W1 需求评审：周三 PM 组织三方评审业务需求 + 埋点需求 +UI 需求。
- W2 排期：周三 API RD 出排期，周四客户端 RD 出排期，UI 产出视觉初稿。客户端 RD 排期需细化至需求可提测时间点，便于 QA、PM 和 UI 提前协调测试、验收时间。
- W3 接口评审 + 服务端先行开发：接口文档，UI 资源周四前到位。
- W4 客户端开发，周四 API 提测（最晚）。
- W5 客户端开发 + 冒烟测试：RD 开发完后直接冒烟测试，以需求维度交付给 QA，不需要等到最后一天集中冒烟测试；需求交付后，PM 与 UI 可介入验收环节。
- W6 客户端测试 + 周二服务端上线：客户端一轮测试结束前，PM 与 UI 应完成需求验收。
- W7 灰度 + 周二全量理想状态一个版本的周期为五周半。
- W3 时，PM 会继续组织三方评审；W4 时，API 继续开发下版本；W5 时，客户端 RD 会继续下一个版本的开发。

3.2.2 分组交付

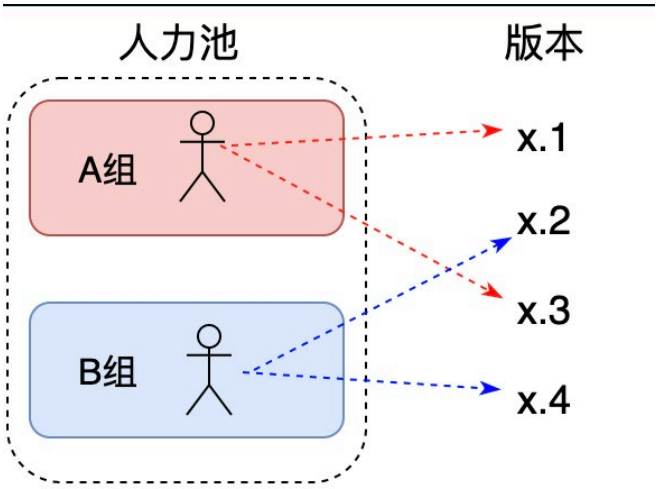
双周迭代模式的变化也推动我们在人力分配方面的改进，人力安排演变经历了三个过程：

在双周迭代的过程中，客户端的交付是连贯的，理想状态是每个 RD 可以隔版进行需求开发。但是大家都知道，写完需求是不可能没有 Bug 的，需求的估时也不可能都是 5 的倍数，这样就会造成很多 RD 既要开发当前版本，又要解决上个版本的 Bug。这间接引入了 2 个问题：

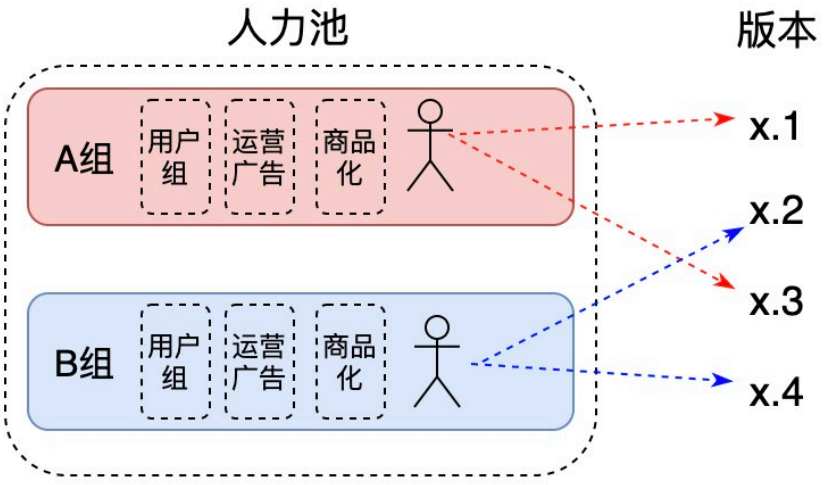
1. 个人成长方面：单个 RD 可能会在同一个时间段内，出现又要评审需求、又要开发需求、又要改 Bug，时间长了会堆积疲惫感。另外，需求的估时不是 5 的倍数，每个版本单个 RD 是否能满人力工作是不确定的，不利于资源的分配。
2. 人力分配合理性：版本的需求是存在不确定性的，而 App 的需求熟悉度是存在和 RD 一一对应的，当熟悉度最高的 RD 在上个版本的时候，当前版本的版本主 R 就会很头疼，是“能者多劳”还是“合理分配”，给人力分配的合理性又带来了新的挑战。



后来我们经过内部研究，提出了 AB 分组的方案，保证 AB 组是隔版本进行开发，当组内需求拆分有冲突时，需要在内部协调消化，避免 RD 出现每个版本都会参与开发的情况发生。



AB 分组施行以后每个 RD 都是各版本开发的，不会出现同一个时间段内，要处理上个版本的 Bug 和当版需求的情况。但业务的熟悉度和 RD 的强对应关系还是没有得到解决，所以我们就进一步的思考，提出采用业务维度 AB 组。如下图所示，我们先将同学进行业务划分，在以业务组进行 AB 的划分。当然这个业务组人员的数量需要考虑业务的规划情况，当某一个业务需求量增长上来的时候，业务组的大小也需要同步进行增长。



3.2.3 双周迭代的优缺点

双周迭代的优点

- 提高了业务迭代效率，从而提升了业务的产出，以一月能够完成二个版本的交付速度，持续地交付版本，达到一年交付 20-24 个版本的交付目标。
- 在人力整合利用上更加合理化，各角色都能够持续的产出，版本的交付周期变为 5.5 周。

双周迭代的不足

相比之前的当月发版，虽然双周迭代带来一定收益，但还是存在一些客观的问题：

- 从评审到交付，发布周期还是太长，产品快速上线的诉求依然不能得到很好的满足。
- 敏捷性受到一定影响，对于特定需求和小需求不是很友好。

3.3 双周结合周交付的模型实践

外卖在历经了双周迭代后，一定程度的缩短了版本交付周期，提高了业务迭代效率，优化了人力安排，但是仍然存在一些问题。对于一些简单的需求来说，像外卖中“我的页面”模块的点击热区扩大（“我的页面”已经是 RN 页面），实际上开发只需要 2 天，却仍然要跟随版本迭代的节奏，5.5 周后才能上线，需求交付周期仍需进一步缩短。另外，外卖自研动态化框架 Mach 在大量业务的应用落地，多种开发流程及交付需要，使得双周交付越来越难满足外卖业务版本迭代及交付的节奏。

3.3.1 动态化能力的建设

为了满足业务需求的快速上线及外卖面临的包体积的压力，外卖引入了动态化来解决此类痛点问题。外卖动态化能力建设的思路主要是：核心流程页面区域动态化，区域动态化方案采用外卖自研的 Mach 动态化框架，核心流程采用区域动态化，保证页面的稳定性及用户体验，非核心流程页面 MRN 化（Meituan React Native），通过 MRN 来替换所有的低 PV 页面，下线 Native 页面。具体技术细节可参考《React Native 在美团外卖客户端的实践》一文。

目前外卖低 PV 的 MRN 页面数量已达：56/67，已经覆盖了外卖所有的低 PV 页面，除核心流量页面均为 MRN 页面，从页面数量维度，MRN 页面占比已达 83.58%。

Mach 区域动态化覆盖了首页核心流量区、二级金刚页、订单页、点菜页及搜索页，目前上线的模板数已达 43 个。

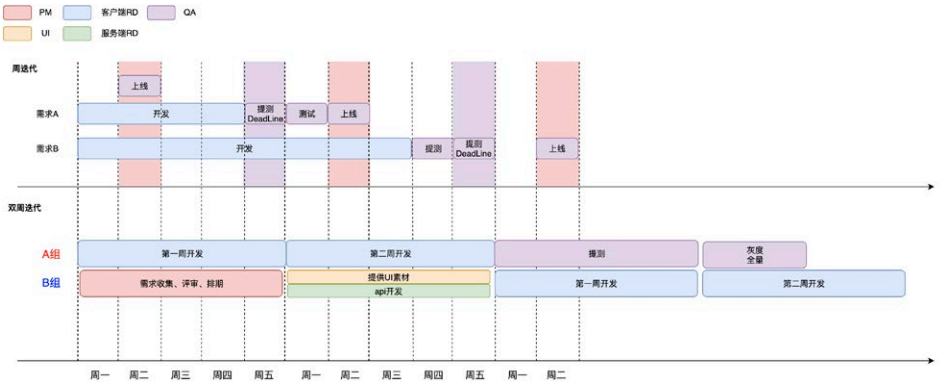
3.3.2 双周迭代 & 周迭代

在完成了外卖动态化能力的建设后，外卖业务逐渐演进出了两类需求：

1. 主版本需求，主版本需求主要是 Native 方式实现，结合少部分动态化需求。
2. 敏捷开发需求，主要通过动态化和敏捷模型的加入实现高效的版本迭代，动态化主要指上节所介绍的动态化能力的建设，敏捷模型采用前文所说的敏捷模型。

面对上述两类需求，现有双周迭代及分组交付流程，对于部分特定需求及小需求不太友好，无法做到需求最快速上线，如上述“我的页面”点击热区扩大。

因此，我们在双周迭代的基础上，演进出了现在的双周迭代结合周迭代的交付模型，交付模型图如下：



双周迭代结合周迭代的交付模型是在原有双周迭代的基础上，保持主版本需求双周迭代不变，增加每周的动态上线窗口，以每周二为动态上线窗口时间，周五为提测 DeadLine，根据动态化需求的开发及提测时间来动态搭车上线。

目前，外卖的周迭代需求分为两类：

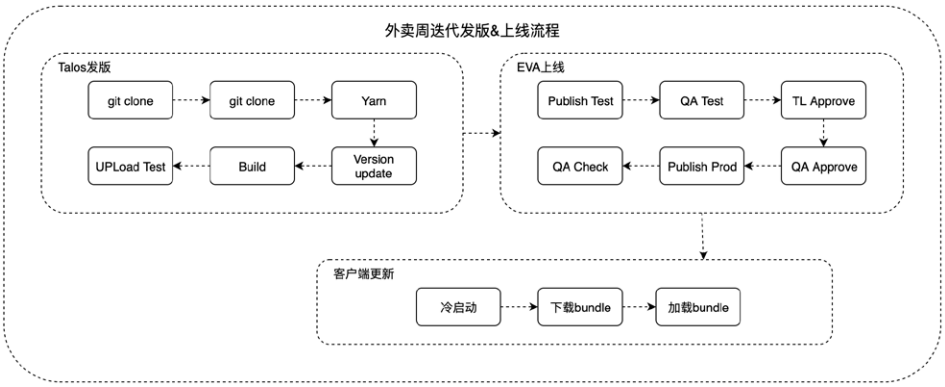
1. 纯动态化需求，根据需求开发及测试排期选择上线窗口，搭最近一次的上线窗口。
2. 动态化结合 Native 需求，跟随主版本交付流程，在提测周发版上线，主要面向敏捷迭代内容。

纯动态化需求由于不依赖 Native 发版，无需依赖 Native 的全回归，开发完成可随时提测，提测后功能测试完成即可上线，可以做到需求的最快速上线。敏捷迭代内容，不依赖主版本的评审，采用自主评审，自主决策跟版的策略。

而多种动态化方案的开发模式，需要统一的动态化发版及上线能力的支撑。外卖周迭代发版及上线具体可以分为两步：

1. 发版。将动态化模板进行打包，得到 Bundle 并上传，通过 Talos 平台（美团内部自研）进行打包，Talos 是基于前后端分离思想设计的一套前端 DevOps 解决方案。
2. 上线。将通过 Talos 打包得到的 Bundle 上线发布，发布通过 EVA 客户端动态分发平台进行线上发布。

具体流程如下图所示：



截止目前，外卖周迭代发版经过了 15 个上线窗口，共计发版 50+ 次。

双周迭代结合周迭代的交付方式进一步缩短了版本的交付周期，从需求收集评审到需求上线对于敏捷需求可以缩短到 3.5 周。对于纯动态化需求可缩短到 1 周，最终做到了需求评审做到了需求的快速上线、快速验证、快速修复，敏捷迭代。

3.4 自动化版本管理系统

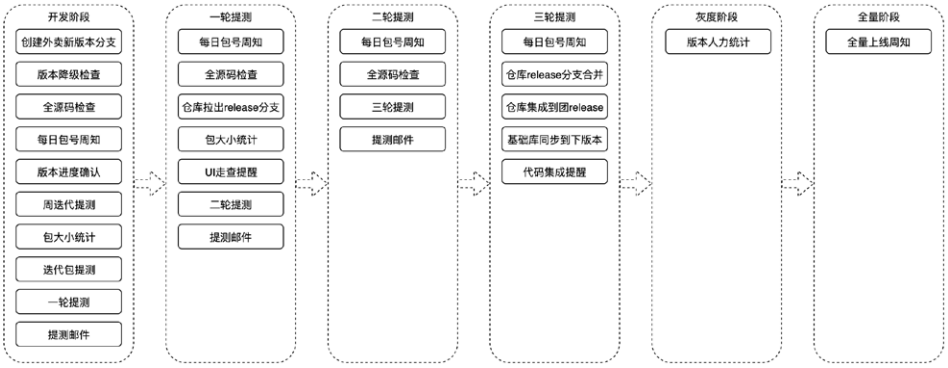
在施行了双周迭代结合周迭代后，大大缩短了需求的上线周期，但是同时也提升了版本流程管理的复杂度。怎么样保证版本流程的正常运作呢？外卖 C 端涉及开发人员分为 A、B 组及敏捷开发人力；同时涉及美团频道及外卖 App 双平台，需要考虑双平台的版本迭代周期；涉及阶段多，各阶段都需要人工操作，执行事件多，双端执行事件总计 40+；涉及双版本并行开发，仓库代码管理复杂。

原有版本迭代流程，版本各阶段的所有事件都由相应负责人执行。

1. 分支创建、版本降级检查、每日 QA 包号周知，仓库集成及基础库同步、打包，提测邮件等都由迭代工程小组人员负责执行。
2. 仓库拉出 Release 分支，提审灰度后分支合并、集成、发版由外卖各仓库负责人负责执行。
3. 周迭代、迭代、一轮、二轮、三轮提测阶段冒烟周知，则通过排班的形式，在冒烟提测当天，由人工提醒并监督。

在版本迭代流程上会耗费了大量的人力。如何解放这部分人力，让整个版本迭代流程自动化，做到无人值守，减少人工操作的失误，是此前外卖前端技术团队面临的非常头疼的问题。

在我们规范了双周迭代 & 周迭代的流程后，版本各阶段的时间能够基本固定。为了能够做到自动化，我们把版本各阶段需要人工操作的事件整理归类，把事件确定到当前版本的固定一天固定时间点，以一个版本为例，我们整理归类如下：



确定好所有事件的执行时间后，我们便尝试把所有需要人工操作的事件自动化。并通过在确定的时间去触发，来做到整个版本流程的自动化。

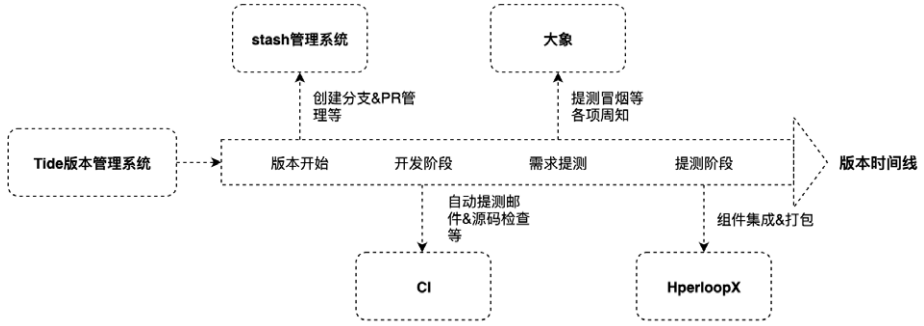
早期的版本管理，我们依托于 Gulf（美团内部日期管理系统）、QA-Assistant 系统（美团内部事件通知系统）、CI 及大象（美团内部通讯工具）消息，通过在 Gulf 系统提前配置好各版本各事件的时间，并通过 QA-Assistant 配置人员名单及消息，在固定时间节点，Gulf 去触发 QA-Assistant，通过上述方式，做到了各类提醒的自动化，并且在 CI 上部署上自动化 Job，由人工触发。

但是通过这种方式，存在一些弊端：

1. 版本各节点事件都需人工单独一个个配置，且 Gulf 和 QA-Assistant 系统的触发一直需要关注。
2. 仍然需要大量人工操作，只能做到流程的半自动化，人力还是无法做到解放。
3. 版本流程无法做到可视化。

为了解决上述问题，我们和 Tide 版本管理系统（美团内部组件）共建，实现了现在的外卖 C 端自动化流程管理。以 Gulfstream 的版本排期为基准，通过 Tide 版本管理系统进行事件触发，通过脚本将 CI、Stash 仓库管理系统、HyperloopX 发版打包系统、大象整合，做到代码 / 分支管理自动化、打包集成自动化、周知提醒自动化、版本流程可视化。

在版本开始前配置好当前版本，后续则会在版本的每个关键节点自动执行需要执行的事件。



Tide 版本管理可视化界面如下，按照版本各阶段及时间线顺序排列：

各类事件提醒及执行通知如下：

各类提醒确认结果即时展示：

3.5 CI 建设

在 CI 建设方面，我们工作主要集中在以下 5 个阶段：准备阶段、PR 检测、开发阶段、提测阶段和发版阶段。

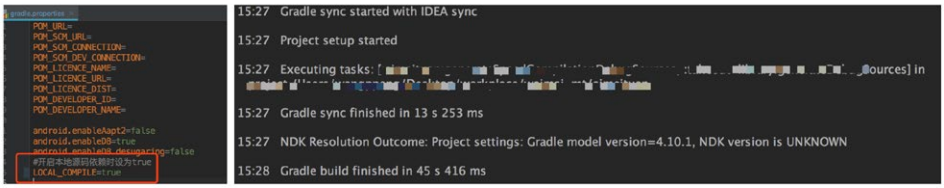
下面介绍三个 CI 工作中，外卖场景下的特定检查：

- Aimeituan 工程独立编译自动配置：同事反馈配置团 App 源码依赖的配置项较繁琐，切换开发分支时需要反复进行配置，为了提供 RD 的开发体验，通过脚本自动修改 WM 提测分支的源码依赖配置项，只需修改 Aimeituan 的 gradle.properties 一个属性实现源码依赖切换，实现一键切换的功能，提高版本开发效率，优化项包括：
  - 业务方依赖只保留团首页和外卖业务。
  - 只保留开发调试必要的插件：ServiceLoader 和 WmRouter，提高编译速度。
  - 只保留调试 Flavor：提高编译速度。
- 业务库检测是否有 PR 未合入：版本快速迭代对代码管理也带来了挑战，发生了几次因为 RD 未及时合入代码，影响 QA 提测进度的问题，因此在打提测包之前增加了一个检测环节，如果检测到有 PR 未合入提测分支，将在大象群统一周知发起人，提高版本的交付质量。

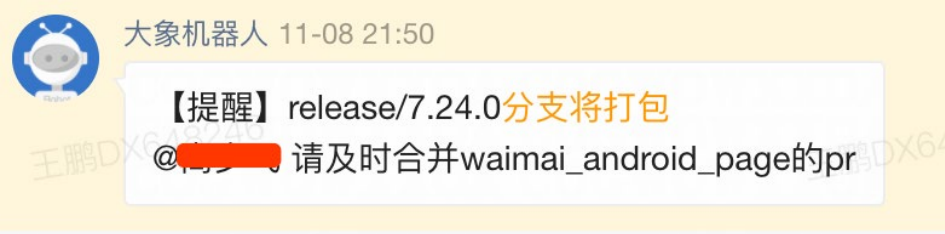
- 定时检测壳工程是否有更新，自动触发打包，保证 QA 第二天能回归前一天所有的修改，避免测试返工的问题，提高版本测试有效性。

总结

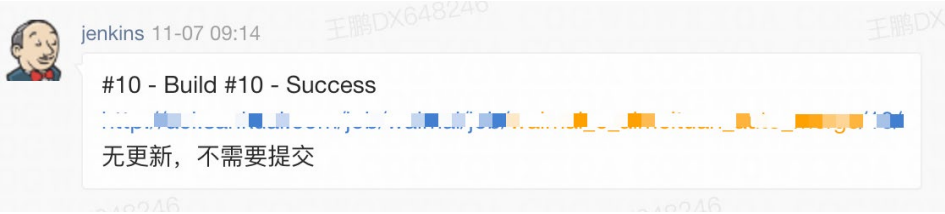
- Aimeituan 独立编译配置从 5 ~ 10 分钟（同事反馈从修改配置到同步完成的耗时）降低到 45s 左右，优化了 80% ~ 90%。



- 在经过了几个版本的实践证明，自动化检测 PR 合入是有必要的，而且能够有效降低开发人为失误导致漏提交代码发生的概率。



- 每天早上同步 Aimeituan 的 Release 分支，及时周知负责人同步结果，减少人工 Check 的重复工作。

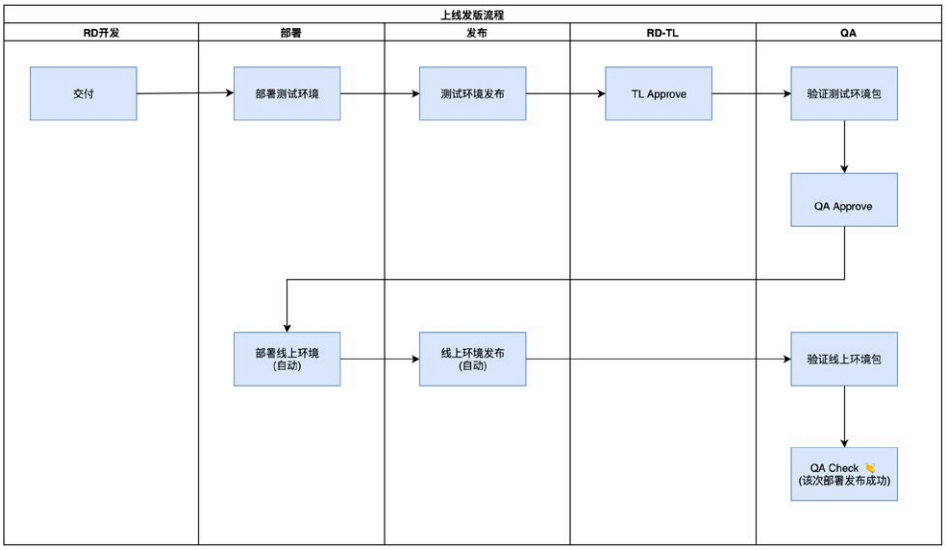


3.6 部署发布

部署和发布，对于移动 App 来说，已经到了最后的环节。我们认为的部署行为主要指将一个还处在测试的版本提交到测试环境，然后让 QA 进行测试，发布行为主要指将测试完成的稳定版本提交到线上环境交付给用户。当然对于部署和发布，每个团队都可以思考自己的定义。例如我们进一步的细化部署和发布的行为，可以划分为部署测试环境、部署线上环境、部署回滚、发布灰度、发布灰度监控、发布全量、发布回滚、发布监控等等。下面主要介绍外卖业务在部署发布中，建设的业务事务流和无人灰度监控。

3.6.1 发布事务流

发布在整个交付过程中是最重要的一环，而这个环节涉及的角色主要是 RD 和 QA。我们需要明确的定义这个环节 RD 和 QA 的事务，避免角色间沟通不畅，跨角色的任务不清的问题。外卖通过持续的优化流程，明确流程的操作行为，来监督流程的正常运行，形成当前的事务流，如下图所示：



这里需要重点讲解图中的关键点：

- 事务流中，RD 负责交付代码到测试环境，QA 同学负责测试环境的和线上环境的验证。RD 和 QA 角色按照时间顺序，各自前面工作和后面工作，事项归属清晰明确。
- 流程中设置 TL Approve 和 QA Approve，同时，RD 作为代码的完成者，只具备测试环境的部署权限，最终线上的权限由 QA 掌控，做到了上线的双角色 Check，避免了 RD 即是“裁判员”又是“运动员”的情况。
- 当完成测试时候，部署线上环境和线上环境发布，都做到自动执行，通过自动化的手段节约重复的工作。

3.6.2 无人值守灰度

一个功能上线，往往要经历灰度到全量的过程。由于灰度已经是到了交付到用户手上的最终环境，灰度前的检查和放量都非常重要，需要做到万无一失。灰度的次数往往是全量次数的 2~3 倍，这使得灰度的成本在整个交付流程中也变得非常高。理想情况下，我们希望发布前的流程都做到自动化，灰度的过程，也无需人工参与，自动的放量、停灰、全量，出现异常自动执行灰度 SOP、监控项也在灰度的时候自动开启，出现异常自动同步信息给相关方。但在实际过程中，由于平台的差异，部分环节还是很难做到，例如 iOS 的发布。下图是外卖业务无人值守灰度的流程：

其中几个关键点：

- 起始点来源于我们录入的 GulfStream 日期，由于双周迭代后，版本的迭代周期是可预期的，所以能够以季度维度录入 GulfStream 中，由此处触发整个链条的启动。
- 我们利用 Tide 版本管理系统，将版本的日期自动读入，在需要发布的节点前，自动检查，做好自动化检查，部分无法自动化的，由 Tide 系统提醒 RD 和 QA 去做检查。

- 定义好灰度放量策略，在发布时按照设定策略，自动发布放量。
- 外卖业务非常复杂，监控的内容也很复杂，对这个环节不是很熟悉的同学，是很难胜任监控运维工作的，自动化更是无法执行，因此外卖的策略是将监控报警标准化，制定 SLA，分级别定义监控、报警、预案、执行人，当版本发布后，触发监控 Job，以分钟级去 CAT（已经开源）、Sniffer、Crash 等等平台去拉取当前灰度版本的数据和定义的指标比较。如果在合理定义范围内，继续监控。如果出现异常，大象定向通知相关处理人。这样使得灰度的自动化监控工作的执行过程是可操作的。

4. 外卖持续交付的总结

美团外卖业务，应该建立怎么样的模型、配套什么的技术手段，才可以在更短的周期内，高质量地交付业务，满足各团队的发展呢？我们从早期的单月交付，到双周交付，再到双周交付结合周交付，我们在不断地尝试和演进外卖的持续交付模型。随着不断的深入了解，我们发现这是一个很大的话题。到目前为止，美团外卖团队在这条路上只能算得上是刚刚起步。不过，在实践的过程中，我们也逐渐地摸索到一些经验，在此分享一下我们的思考。

4.1 遵循原则

首先我们认为持续交付需要遵循以下的原则：

持续自动化

在我们整个交付的过程中，有大量的工作是会重复进行的，如果我们不能将这些重复的操作逐渐变成自动化，我们就需要频繁、反复地去做一些看不到收益，但是又不得不去进行的事情。随着交付的内容变得越来越多，交付的速度要越来越快，团队的疲劳感会越来越重，最终疲于奔命。在实际的工作中，由于发布的形态流程的不标准、生成环境和数据的不一致，我们无法一次性做到所有环节自动化，但这不影响我们先站在全局去思考，去持续地拆解子问题，将子问题逐渐地变成自动化。随着整个交付过程的自动化程度不断提升，整个流程会变得简单容易，这样开发人员可以更加聚焦

在完成功能本身的工作上。

前置解决

大部分人心理上对于比较麻烦、比较痛苦的事项，都倾向不做或者拖到最后去做。由于存在这样的心理，如果在交付流程中遇到问题，我们的方案很容易掉到一个陷阱中，倾向保持现状或延后再处理。但是，对于一个多人协作交付的业务来说，这是非常不可取的，越往后处理，对整个交付的影响越大，处理效率越低下。举个例子来说，一个 PR 可能导致一个历史功能不可用，如果我们在这个同学提这个 PR 的时候就拦截到，那么只会影响这个同学，也不需要回溯问题；如果延到代码集成环节，那么就需要想办法去找到是谁，然后因为什么原因提这段代码，本次集成就可能直接导致失败，影响其他正在集成的代码。如果拖到测试环境，那么就会涉及至少 RD 和 QA 等两个角色去追寻这个问题。所以在持续交付的过程中，我们尽可能尽早地将问题在刚出现时就解决掉，保障整个交付流水线的通畅。

版本化

在我们持续交付的过程中，要将每个环节尽可能的版本化。不仅仅从代码方面版本化，针对数据、配置、环境都可以进一步的版本化。版本化的好处是不言而喻的，首先在实现版本化的过程中，尽可能地将各个环节解耦开，让环节和环节之间没有强依赖。当某一个环节出现异常的时候，我们可以很快地降级到无问题的版本，从而保障整个流水线的持续运转。版本化的另一个好处，就是可回溯，当出现异常的时候，我们可以很快地对比历史版本和当前版本的差异，缩小范围，快速的定位问题。

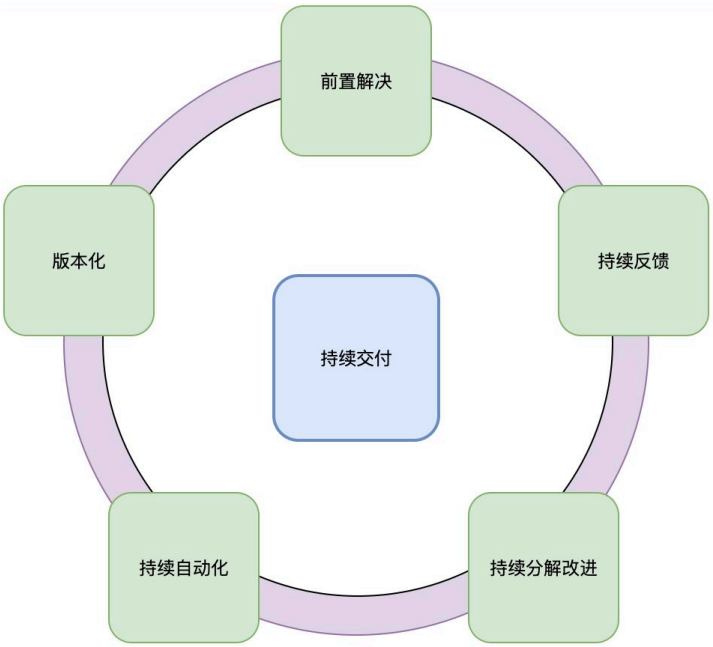
持续反馈

在整个交付的过程中，每个环节都可能出错。当出错的时候，需要快速地反馈到相关的负责人。该负责人也需要尽快地将反馈的问题处理，给予一个稳定的版本。如果没有反馈这个环节，导致了整个交付流水线被阻塞的人，可能完全不知道自己造成了阻塞。另外，在一个功能的交付过程中，可能会有多次提交代码、多次集成、多次测试，如果每次是什么样的结果，都无法得到反馈，那么整个交付的风险会逐渐累积。

只有将问题更早地反馈、更早地处理，才能有效地降低整个功能交付的风险。

持续分解改进

不管我们当前将整个交付的过程做到了多么高效，未来随着业务的发展，新成员的加入和离开，交付的过程都在逐渐发生衍变。正如《谈谈持续集成，持续交付，持续部署之间的区别》一文提到的“持续部署是理想的工作流”一样，想一劳永逸解决所有问题是一种理想的工作状态。随着时间的推移，交付的过程会出现新的问题，交付的效率又会开始逐渐下降。如果要长久地保持交付效率能够有持续的改善，我们需要不断地对抽象的交付过程进行分析，层层剥离，将整个交付过程转变成一系列小而可以解决的问题，然后持续地解决这些小问题。



4.2 关键步骤

针对上述原则，我们可以进一步地来指导外卖持续交付中的关键过程。通过前文所讲，我们认为持续交付就是从产品同学收集到需求，提交到研发团队，最终形成稳定

的产品交给用户手上的过程，针对这个过程主要细化包括：

提需求

持续交付启动的第一环对应的就是 PM 提需求。俗话说“万事开头难”，“提需求”看似是一件容易的事情，其实背后是一个非常复杂的问题，涉及到策略选择、动态规划、排队论等一系列的理论。往往在大部分的团队中，“提需求”被认为是最简单的事情，这使得整个交付流程后续的工作，无法高效开展。

我们细分析下，需求根据大小可以分成大需求、小需求；根据优先级可以分成高、中、低需求；根据开发的人员不同，又可以分成前端需求、后端需求、策略需求等等。需求提出来后，怎么快速地判断出如何划分这个需求，需求的优先级是什么，需求的上线时间怎么样，谁来做这个需求，这个需求会涉及到谁。在涉及到这个需求和其他需求的比较时，才能回答刚才提到的问题，这又涉及到需求的定义，需求产出比的基线问题。

在外卖的交付模型中，对于“提需求”，主要遵循的是组织管理原则，即以组织结构为单位，当前组织下的 PM 给予当前组织下的 RD 提出需求，需求的优先级也以当前组织下的事项进行综合考虑。这样的好处，不言而喻，比较敏捷。劣势是，当遇到跨组织需求，工作成本会急剧增加。同时，我们通过交付模型，将提需求的时间点可预见，在每次提需求的环节，严格要求每一组织下的需求给予明确的优先级。每次提出的需求数量、需求评审时长标准化。并且配套标准的需求模板，让 PM 在提出需求时，自我完成需求边界检查，需求涉及方的前置沟通。通过这样的组织管理手段、模板管理手段、流程标准化手段，来尽可能地保证提需求的质量。

提代码

当单个 RD 同学认为已经基本完成了功能开发后，就会进入到提代码的环节，也可以叫做代码集成环节。这个环节是个人行为和整体交付产生对接的环节。在这个环节我们要确保个人的行为不影响整体的流水线的进行。

如何保障？我们采用的手段主要是本地检查 +CI 检查 + 集成测试。当代码检查通过后，代码的集成是自动完成，无需个人干预。不过，这里也存在一个矛盾点，如果本地检查 +CI 检查项 + 集成测试项越来越多，检查的时间将会非常长，这将会使得每次提交代码变得异常痛苦。而如何解决这个问题？最好的手段，我们认为是通过持续优化代码的架构，来让每次 RD 的代码修改尽可能范围小。这样对于有特定范围的修改，本次 CI 检查就存在可简化的可能性。

部署测试

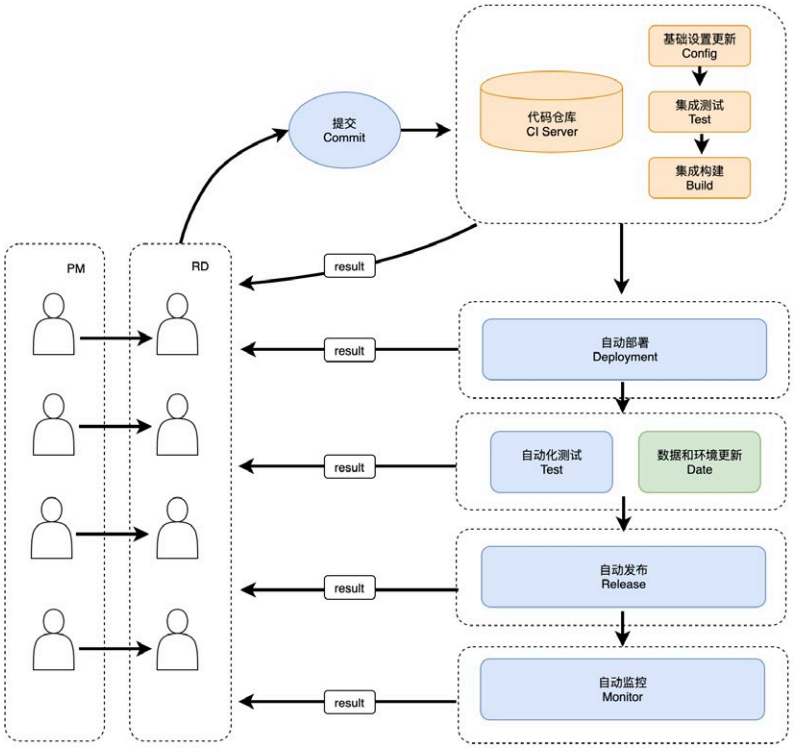
当完成代码集成后，就需要进入到测试环节了。那首先我们需要将代码部署到测试环境，因为外卖主要提供 App 供测试包，所以我们的部署，主要就是打出 App 包来，周知 QA 测试。目前外卖的测试，分位一轮新功能测试、二轮全回归测试、三轮主流程回归测试。新功能因为新引入，可能存在前置条件，而在本次测试的过程中，存在需要编写自动化用例和人工测试的用例。但对于老功能，我们希望未来能够尽可能地采用自动化为主的方式。

而在实践的过程中，我们发现自动化测试的一个很重要的前置条件是，数据和环境的可配置性、稳定性。试想下，数据不稳定，一会返回是空，一会返回不为空，是没办法自动化测试校验的。对于这个问题，目前我们的思路是通过建立各环节下根据标准协议定义的 Mock 数据，将复杂的全链条，拆分成不同节点，在通过配置去控制数据的环境。通过保障每个环节下的质量，来确保整体的质量。

线上监控

保障外卖业务正常线上运行，会需要监控多项的指标。对于如何做好线上监控，我们的思路是设立好各项基线的标准基线，通过脚本获取线上的版本，当检测到新版本，就会自动的创建当前版本的监控。当监控发现线上指标的值和设定的指标基线出现降低时，发生报警行为。而自动化监控最需要考虑的是误报率的问题。版本的发布正常通常又会分为 2 个阶段，灰度阶段和全量阶段。全量阶段量大，比较稳定，误报率较低；而灰度阶段，量小，不稳定，容易出现误报。而解决这个问题，最好的办法是，

本次灰度对比上一次灰度的指标，而非对比上一次全量的指标，维度一样，可以有效地减少误报率。



5. 展望

当前，美团外卖业务仍然处于快速增长阶段，建立怎么样的流程、配套什么的技术手段，才能满足在更短的周期内，高质量地交付业务，进而满足各个业务的发展呢？毫无疑问，持续交付是解决这个问题的重要一环。但针对持续交付这个主题，美团外卖也算是刚刚起步。

未来我们的建设主要集中在 2 个方向，精细化运营持续集成和建设自动化测试。如上文所述，随着业务越来越复杂，涉及的角色越来越多，代码集成的管控需更加严格，而严格的代码集成管控将增加团队成员每次提代码的痛苦。如何做到差异化的检查和准入，将是未来持续集成的建设方向。

我们可以针对不同类型的 PR，不同时间节点下的 PR，实施不同的差异化定制规则检查，在合入质量和检查周期上寻找到一个合适的平衡点。另外对于测试环节，目前美团外卖还是主要集中在人工保障环节，大量的测试 Case。我们希望未来能够尽可能地采用自动化为主的方式进行测试，而让 QA 可以集中精力测试新功能或非常边界异常的场景验证。

6. 参考文献

如何理解持续集成、持续交付、持续部署
谈谈持续集成、持续交付、持续部署的区别
持续交付 2.0：业务引领的 DevOps 精要

作者简介

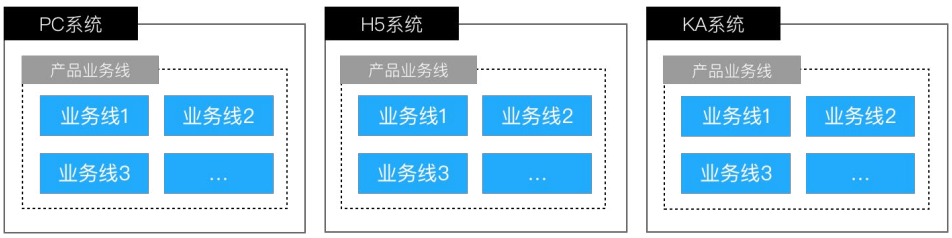
晓飞，2015 年加入美团，美团外卖团队技术专家。
王鹏，2017 年加入美团，美团外卖团队高级工程师。
江伟，2018 年加入美团，美团外卖团队高级工程师。

微前端在美团外卖的实践

作者：张啸　魏潇　天尧

背景

微前端是一种利用微件拆分来达到工程拆分治理的方案，可以解决工程膨胀、开发维护困难等问题。随着前端业务场景越来越复杂，微前端这个概念最近被提起得越来越多，业界也有很多团队开始探索实践并在业务中进行了落地。可以看到，很多团队也遇到了各种各样的问题，但各自也都有着不同的处理方案。诚然，任何技术的实现都要依托业务场景才会变得有意义，所以在阐述美团外卖广告团队的微前端实践之前，我们先来简单介绍一下外卖商家广告端的业务形态。目前，我们开发和维护的系统主要包括三端：



- PC 系统：单门店投放系统 PC 端
- H5 系统：单门店投放系统 H5 端
- KA 系统：多门店投放系统 PC 端

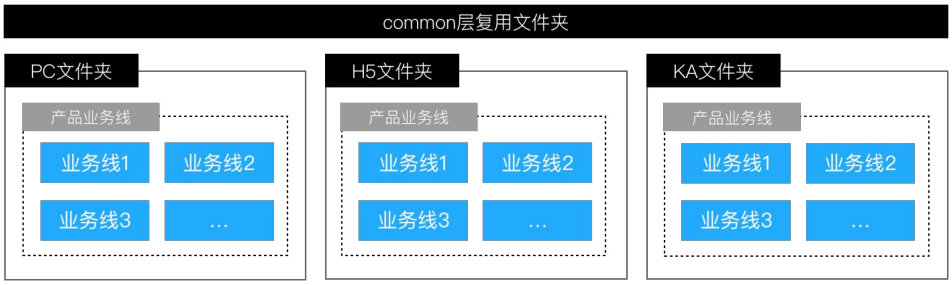
如上图所示，原始解决方案的三端由各自独立开发和维护，各自包含所有的业务线，而我们的业务开发情况是：

- PC 端和 H5 端相同业务线的**基本业务逻辑一致**，UI 差异大。
- PC 端和 KA 端相同业务线的**部分业务逻辑一致**，UI 差异小。

在这种特殊的业务场景下，就会出现一个有关开发效率的抉择问题。即我们希望能复用的部分只开发一次，而不是三次。那么接下来，就有两个问题摆在我们面前：

- 如何进行**物理层面的复用**（不同端的代码在不同地址的 Git 仓库）。
- 如何进行**逻辑层面的复用**（不同端的相同逻辑如何使用一份代码进行抽象）。

我们这里重点看一下物理层面的复用，即：如何在物理空间上使得各自独立的三端系统（不同仓库）引入我们的复用层？我们尝试了 NPM 包、Git subtree 等类“共享文件”的方式后发现，最有效率的复用方式是把三个系统放在一个仓库里，去消除物理空间上的隔离，而不是去连接不同的物理空间。当然，我们三端系统的技术栈是一致的，所以就进行了如下图的改造：

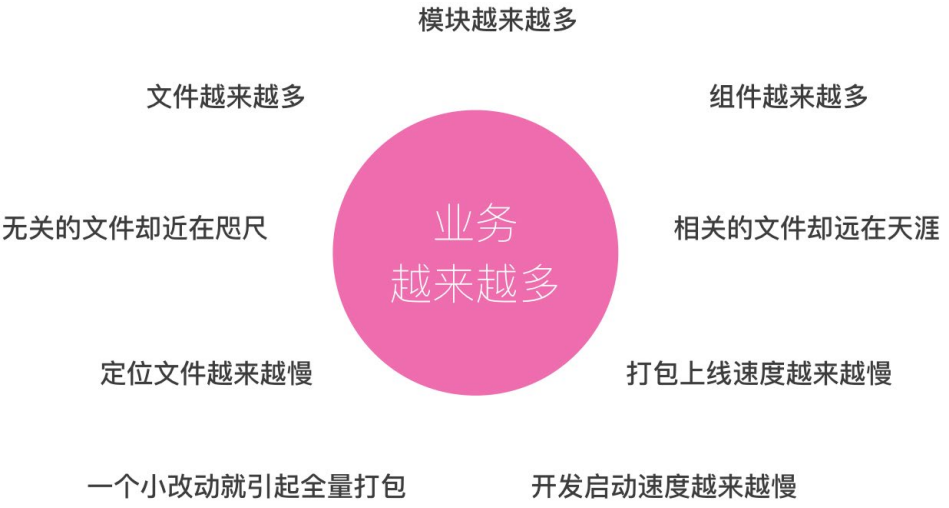


可以看到，当我们把三端系统放在一个仓库中时，通过 common 文件夹提供了物理层面可复用的土壤，不再需要“共享文件”式地进行频繁地拉取操作，直接引用复用即可。不过，在带来物理层面复用效率提升的同时，也加速了整个工程出现了爆炸式发展的问题，随着产品线从最初的几个发展到现在的几十个之多，工程管理成本也在迅速增长。具体来说，包括如下四个方面：

- 新业务线产品急速增加，同时为了保证三端系统复用效率的最大化，把文件放入同一仓库管理，导致文件数量增长极快，管理及协同开发难度也在不断加大。
- 文件越来越多，文件结构越不受控制，业务开发寻址变得越来越困难。
- 文件越来越多，开发、构建、部署速度变得越来越慢，开发体验在持续下降。

- 不同业务线间没有物理隔离，出现了跨业务线互相引用混乱，例如 A 业务线出现了 B 业务线名字的组件。

如下图所示，具体地说明了原有架构存在的问题。为了要解决这些问题，我们意识到需要拆分这些应用，即进行工程优化的常规手段进行“分治”。那么要怎么拆呢？自然而然地我们就想到了微前端的概念。也从这个概念出发，我们参考业界优秀方案，同时也深度结合了广告端实际业务的开发情况，对现有工程进行了微前端的实践与落地。



需求分析

结合现有工程的状况，我们进行了深度的分析。不过，在进行微前端方案确定前，我们先确定了需求点及期望收益，如下表所示：

需求点	收益与要求
拆分解耦	(1) 按业务领域拆分成不同的仓库进行维护，不同业务线的开发者更加独立，不同业务线之间互不影响。(2) 物理层面拆分，加速寻址，新增功能修改 Bug 更加迅速。(3) 逻辑层面拆分，杜绝引用混乱，不会出现 A 业务线引用 B 业务线组件的情况。
加速体验	(1) 开发环境急速启动，提高开发体验。(2) 业务线按需打包，急速部署上线。

需求点	收益与要求
侵入性低	微前端方案改动原有代码的侵入性降到最小，无需大规模改造，减少甚至消除回归测试的成本。
学习成本低	开发人员无需感知拆分的存在，保持单页应用的开发体验，不需要学习额外的规则。
统一技术栈	为了统一共建与技术沉淀，团队内工程已经统一到了 React 技术栈，禁止使用不同的技术栈进行开发。

方案选择

经过以上的需求分析，我们调研了业界及公司周边的微前端方案，并总结了以下几种方案以及它们各自主要的特点：

- **NPM 式**：子工程以 NPM 包的形式发布源码；打包构建发布还是由基座工程管理，打包时集成。
- **iframe 式**：子工程可以使用不同技术栈；子工程之间完全独立，无任何依赖；基座工程和子工程需要建立通信机制；无单页应用体验；路由地址管理困难。
- **通用中心路由基座式**：子工程可以使用不同技术栈；子工程之间完全独立，无任何依赖；统一由基座工程进行管理，按照 DOM 节点的注册、挂载、卸载来完成。
- **特定中心路由基座式**：子业务线之间使用相同技术栈；基座工程和子工程可以单独开发单独部署；子工程有能力复用基座工程的公共基建。

通过对各个方案特点进行分析，我们将重点关注项进行了对比，如下表所示：

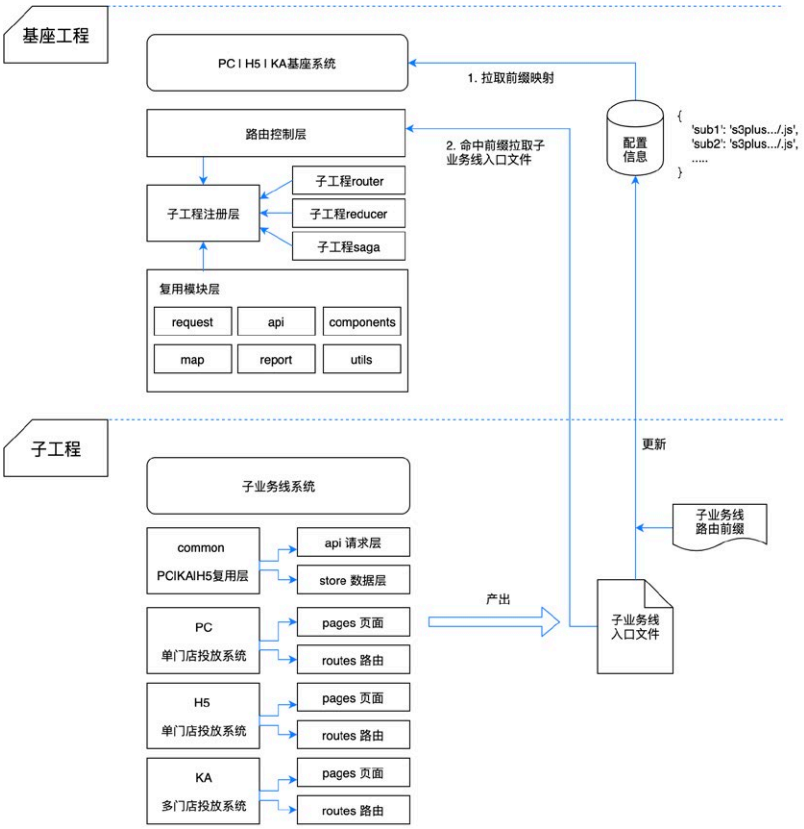
方案	技术栈是否能统一	单独打包	单独部署	打包部署速度	单页应用体验	子工程切换速度	工程间通信难度	现有工程侵入性	学习成本
NPM 式	是（不强制）	否	否	慢	是	快	正常	高	高
iframe 式	是（不强制）	是	是	正常	否	慢	高	高	低
通用中心路由基座式	是（不强制）	是	是	正常	是	慢	高	高	高
特定中心路由基座式	是（强制）	是	是	快	是	快	正常	低	低

经过上面的调研对比之后，我们确定采用了特定中心路由基座式的开发方案，并命名为：**基于 React 的中心路由基座式微前端**。这种方案的优点包括以下几个方面：

- 保证技术栈统一在 React。
- 子工程之间开发互相独立，互不影响。
- 子工程可单独打包、单独部署上线。
- 子工程有能力复用基座工程的公共基建。
- 保持单页应用的体验，子工程之间切换不刷新。
- 改造成本低，对现有工程侵入度较低，业务线迁移成本也较低。
- 开发子工程和原有开发模式基本没有不同，开发人员学习成本较低。

微前端实践概览

通过对方案的分析及技术方向上的梳理，我们确定了微前端的整体方案，如下图所示：



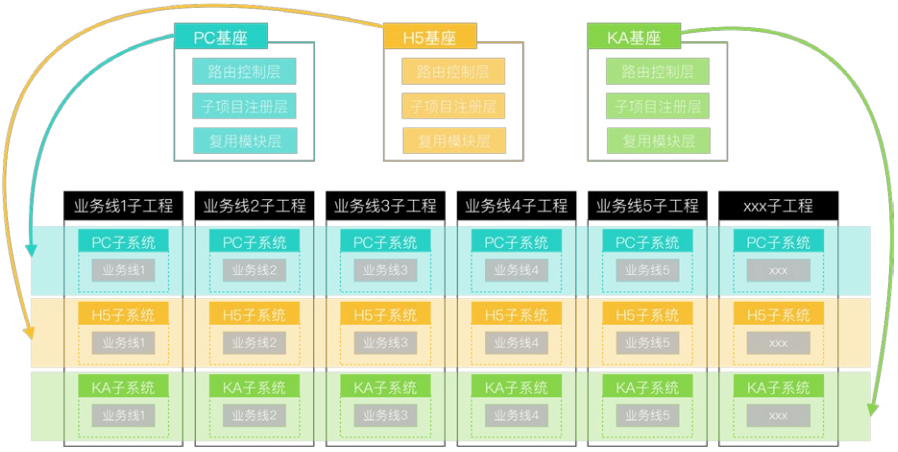
可以看到，整个方案非常简单明确，即按照业务线进行了**路由级别的拆分**。整个系统可分为两个部分：

- 基座工程：用于管理子工程的路由切换、注册子工程的路由和全局 Store 层、提供全局库和复用层。
- 子工程：用于开发子业务线业务代码，一个子工程对应一个子业务线，并且包含三端代码和复用层代码。

基座工程和子工程联系起来的桥梁则是**子工程的入口文件地址和路由地址的映射信息**。这些映射信息可以让基座工程准确地发现子工程资源的路径从而进行加载。

微前端架构下的业务变化

经过微前端实践的改造，我们的业务在结构上发生了如下的变化：



如上图所示，我们进行了微前端式的业务线拆分：

- 原有的 PC 系统、H5 系统、KA 系统分别改造成了 PC 基座系统、H5 基座系统和 KA 基座系统。
- 原有的子业务线被拆分成了单独的子仓库，成为了业务线子工程（上图中 6 个黑框竖列）。

- 业务线子工程分别包含 PC 端、H5 端、KA 端以及该业务线复用层的代码（上图中 3 个纯色背景横列）。

新的拆分使得子工程能够按照业务线进行划分，独立维护。在解决复用层的同时保证了子工程大小可控，即子工程只有单个业务线的代码。而单个业务线的复杂度并不高，也降低了工程维护的复杂度。

采用微前端拆分的方案，使得我们的业务不仅**在纵向上保有了复用的能力**，更重要的是**拥有了横向扩展的能力**，无论产品业务线如何膨胀，我们都可以更轻松地应对。那么为了实现以上的能力，我们做了哪些工作呢？下文我们会详细进行说明。

基于 React 技术栈的中心路由基座式微前端

微前端拆分的方案，我们命名为：基于 React 技术栈的中心路由基座式微前端。在具体实现上，我们会分为动态化方案、路由配置信息设计、子工程接口设计、复用方案设计和流程方案设计等几个模块来逐一进行说明。

动态化方案

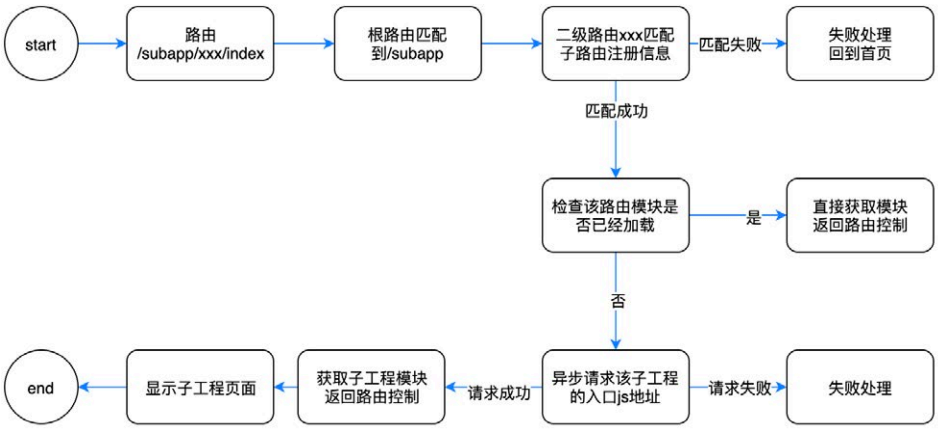
首先，我们需要**路由的管理方案**，使得子工程之间有能力互通切换。其次，我们需要**Store 层的方案**，让子工程有能力使用全局 Store。并且，我们还需要 **CSS 的加载方案**，来加载子工程的样式布局。下面来详细说明这三个方案。

动态路由

动态路由方案是想要进行路由级别的拆分，首先我们要确定用什么来管理路由？很多实现方案倾向于使用特制路由来管理模块。例如开源框架 Single-Spa，实现了自己的一套路由监听来切换子工程，并且需要子工程实现特定的注册、挂载、卸载等接口来完成子工程和基座工程的动态对接，还需要特定的模块管理系统，例如 systemjs 来辅助完成这一过程。毋庸置疑，这对我们原有工程的改造成本很大，还需要添加额外库，进而造成包体积大小上的开销。并且子工程的开发者需要熟悉这些特定的接口，学习成本也比较高。显然，这对于我们的业务场景和需求来说很不划算。

那么，我们选择什么来做路由管理呢？最终我们使用了 React-Router，这样能够保持我们原来的技术栈不变，同时对于工程的侵入也是最低，几乎可以忽略不计。此外，React-Router 能完全可以满足我们的需求，而且自动会帮助我们管理页面的加载与卸载，而不是每次切换路由都重新初始化整个子应用，所以在加载速度体验上也是最优的，跟单页应用体验一致。

实现上也很简单，如下图：



上面这个流程图，展示了我们在基座工程中切换到子工程路由时，加载子工程并进行展示的过程。这里的重点步骤是加载子工程入口文件，并动态注册子工程路由的过程。由于我们使用的是 React-Router，显然要使用其提供的动态能力来完成。这一过程也非常轻量，由于 React-Router 从版本 4 开始有了“破坏级”的升级，于是我们就调研了两种方式进行动态加载路由（目前我们使用的是 React-Router 版本 5），如下表所示：

React-Router 版本	动态加载方式
3	利用 Route 的 getChildRoutes 的 API 异步加载路由。
4 及以上	版本 4 及以上，React-Router 在实现思路上有了非常大的变动，即不再以提前注册路由的集中式路由为设计理念，转变成路由即组件的思路。对于动态加载路由来说，就是动态加载组件，使得我们的动态加载更加容易实现，无须依赖任何 API，只需写一个异步组件即可。

React-Router 版本 3 中，实现的基本代码思路如下：

```
// react-router V3 用于接收子工程的路由
export default () => (
    <Route
        path="/subapp"
        getChildRoutes={(location: any, cb: any) => {
            const { pathname } = location.location;
            // 取路径中标识子工程前缀的部分，例如 '/subapp/xxx/index' 其中
xxx 即路由唯一前缀
            const id = pathname.split('/')[2];
            const subappModule = (subAppMapInfo as any)[id];
            if (subappModule) {
                if (subappRoutes[id]) {
                    // 如果已经加载过该子工程的模块，则不再加载，直接取缓存的
routes
                    cb(null, [subappRoutes[id]]);
                    return;
                }
                // 如果能匹配上前缀则加载相应子工程模块
                currentPrefix = id;
                loadAsyncSubapp(subappModule.js)
                    .then(() => {
                        // 加载子工程完成
                        cb(null, [subappRoutes[id]]);
                    })
                    .catch(() => {
                        // 如果加载失败
                        console.log('loading failed');
                    });
            } else {
                // 可以重定向到首页去
                goBackToIndex();
            }
        }}
    />
);
```

而在 React-Router 版本 4 中，实现的基本代码思路如下：

```
export const AyncComponent: React.FC<{ hotReload?: number; } &
RouteComponentProps> = ({ location, hotReload }) => {
    // 子工程资源是否加载完成
    const [ayncLoaded, setAyncLoaded] = useState(false);
    // 子工程 url 配置信息是否加载完成
    const [subAppMapInfoLoaded, setSubAppMapInfoLoaded] =
useState(false);
```

```
    const [ayncComponent, setAyncComponent] = useState(null);
    const { pathname } = location;
    // 取路径中标识子工程前缀的部分，例如 '/subapp/xxx/index' 其中 xxx 即路由
唯一前缀
    const id = pathname.split('/')[2];
    useEffect(() => {
        // 如果没有子工程配置信息，则请求
        if (!subAppMapInfoLoaded) {
            fetchSubappUrlPath(id).then((data) => {
                subAppMapInfo = data;
                setSubAppMapInfoLoaded(true);
            }).catch((url: any) => {
                // 失败处理
                goBackToIndex();
            });
            return;
        }
        const subappModule = (subAppMapInfo as any)[id];
        if (subappModule) {
            if (subappRoutes[id]) {
                // 如果已经加载过该子工程的模块，则不再加载，直接取缓存的 routes
                setAyncLoaded(true);
                setAyncComponent(subappRoutes[id]);
                return;
            }
            // 如果能匹配上前缀则加载相应子工程模块
            // 如果请求成功，则触发 JSONP 钩子 window.wmadSubapp
            currentPrefix = id;
            setAyncLoaded(false);
            const jsUrl = subappModule.js;
            loadAsyncSubapp(jsUrl)
                .then(() => {
                    // 加载子工程完成
                    setAyncComponent(subappRoutes[id]);
                    setAyncLoaded(true);
                })
                .catch((urlList) => {
                    // 如果加载失败
                    setAyncLoaded(false);
                    console.log('loading failed...');
                });
        } else {
            // 可以重定向到首页去
            goBackToIndex();
        }
    }, [id, subAppMapInfoLoaded, hotReload]);
    return ayncLoaded ? ayncComponent : null;
};
```

可以看到，这种方式实现起来非常简单，不需要额外依赖，同时满足了我们“拆分”的诉求。

动态 Store

对于 Store 层，我们原工程使用的是 Redux，子工程通过路由动态注册进来天然就可以访问到全局 Store，所以对于 Store 的访问能够自动支持。那么，如果子工程想要注册自己的全局 Store 该怎么办呢？而且我们还用了 redux-saga 来作为异步处理方案。redux-saga 如何动态注册呢？还是利用它们各自的 API 就可以达到我们的目的？从下图中可以看到，支持动态 Store 也是花费很小的改造成本就可以完成。

动态 CSS

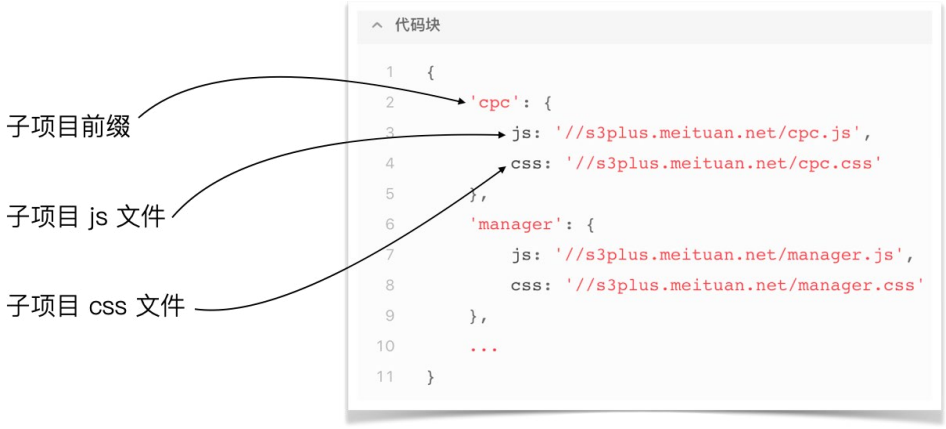
同样的对应子工程的样式布局，我们也需要通过某种途径加载到基座工程中来。这个很自然地用异步加载 CSS 文件通过 style 标签注入来完成，不过这里需要注意两个问题：

一个问题是，加载子工程的 JS 入口文件和 CSS 文件可以同时发起请求，但是需要保证 CSS 文件加载完成后再进行 JS 入口文件的路由注册。因为如果路由先注册了页面就会显示出来，如果这时 CSS 文件还没有加载完毕，就会出现页面样式闪动的问题。我们通过先加载 CSS 再加载 JS 的策略来避免这个问题的发生。

另一个问题是，怎么保证子工程的 CSS 不会和其他子工程冲突。我们利用 PostCSS 插件在编译子工程时，按照分配给子工程的唯一业务线标识，为每一组 CSS 规则生成了命名空间来解决这个问题。而子业务线开发者是没有感知的，可以没有“心智负担”地书写子工程的样式。

路由配置信息方案

在动态加载方案确定之后，基座工程怎么才能知道子工程的资源路径，进而加载对应的 JS 和 CSS 资源呢？我们需要一组映射信息。如下图所示，业务线唯一标识为 Key，相应的静态资源地址为 Value。这样的话，当基座工程切换到子工程时就可以拉取这个配置信息，在路由切换时准确地找到对应的子工程，进而进行后续的资源加载过程。这里可能会遇到的一个问题，即如果 JS 和 CSS 过大，是否能进行拆分？



根据我们业务的实际情况，目前静态资源的大小是可控的，无需注册多个，单一入口

地址完全能够满足我们的业务需求，并且由于我们的改造完全基于现有技术栈。如果业务很复杂，完全可以在子工程中通过 webpack 的动态 import 进行路由懒加载，也就是说，子工程完全可以按照路由再次切分成 chunks 来减少 JS 的包体积。至于 CSS 本身就很小，长期也不会有进行切分的需要。

子工程接口方案

子工程需要暴露它要注册给基座工程的对象，来进行基座工程加载子工程的过程。在子工程入口文件中定义 registerApp 来传递注册的对象，主要代码如下：

```
import reducers from 'common/store/labor/reducer';
import sagas from 'common/store/labor/saga';
import routes from './routes/index';
function registerApp(dep: any = {}): any {
    return {
        routes, // 子工程路由组件
        reducers, // 子工程 Redux 的 reducer
        sagas, // 子工程的 Redux 副作用处理 saga
    };
}
export default registerApp
```

我们这里暴露了子工程的三个对象：这里最重要的就是 routes 路由组件，就是在写 React-Router（版本 4 及以上）的路由。子工程开发者只需要配置 routes 对象即可，没有任何学习成本，其代码如下：

```
/**
 * 子工程路由注册说明
 * 如注册的路由如下：
 * path: 'index'
 * 路由前缀会被追加上，路由前缀规则见变量 urlPrefix
 * 在主工程的访问路劲为：/subapp/${ 工程注册名称 }/index
 */
const urlPrefix = `/subapp/${microConfig.name}/`;
const routes = [
    {
        path: 'index',
        component: IndexPage,
    },
];
```

```
const AppRoutes = () => (
    <Switch>
        {
            routes.map(item => (
                <Route
                    key={item.path}
                    exact
                    path={`${urlPrefix}${item.path}`}
                    component={item.component}
                />
            ))
        }
        <Redirect to="/" />
    </Switch>
);
export default AppRoutes;
```

除了上方的 routes 对象，还剩下两个接口对象是：reducers 和 sagas，用于动态注册全局 Store 相关的数据和副作用处理。这两个接口我们在子工程中暂时没有开放，因为按照业务线拆分过后，由于业务线间独立性很强，全局 Store 的意义就不大了。我们希望子工程可以自行处理自己的 Store，即每个业务线维护自己的 Store，这里就不再展开进行说明了。

复用方案

基座工程除了路由管理之外，还作为共享层共享全局的基建，例如框架基本库、业务组件等。这样做的目的是，子业务线间如果有相同的依赖，切换的时候就不会出现重复加载的问题。例如下面的代码，我们把 React 相关库都以全局的方式导出，而子工程加载的时候就会以 external 的形式加载这些库，这样子工程的开发者不需要额外的第三方模块加载器，直接引用即可，和平时开发 React 应用一致，没有任何学习成本。而和各个业务都相关的公用组件等，我们会放到 wmadMicro 的全局命名空间下进行管理。主要代码如下：

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import * as ReactRouterDOM from 'react-router-dom';
import * as Axios from 'axios';
```

```
import * as History from 'history';
import * as ReactRedux from 'react-redux';
import * as Immutable from 'immutable';
import * as ReduxSagaEffects from 'redux-saga/effects';
import Echarts from 'echarts';
import ReactSlick from 'react-slick';

function registerGlobal(root: any, deps: any) {
    Object.keys(deps).forEach((key) => {
        root[key] = deps[key];
    });
}
registerGlobal(window, {
    // 在这里注册暴露给子工程的全局变量
    React,
    ReactDOM,
    ReactRouterDOM,
    Axios,
    History,
    ReactRedux,
    Immutable,
    ReduxSagaEffects,
    Echarts,
    ReactSlick,
});
export default registerGlobal;
```

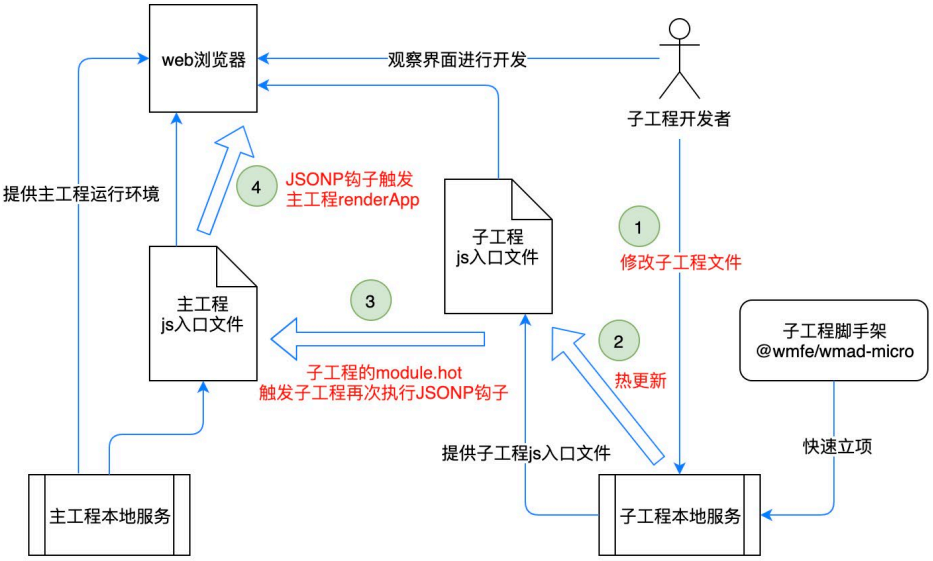
流程方案

在确定了程序拆分运行的整体衔接之后，我们还要确定开发方案、部署方案以及回滚方案。我们如何开始开发一个子工程？以及我们如何部署我们的子工程？

开发流程

有两种开发方案可以满足独立开发的目的：第一种是提供一个基座工程的 Dev 环境，子工程在本地启动后在 Dev 环境进行开发，这种开发方式要求有一套基座工程的更新机制，例如基座工程更新后要同步部署到 Dev 环境。第二种是子工程开发者拉取基座工程到本地并启动本地开发环境，然后拉取子工程到本地，再启动子工程本地开发环境进行开发，这种开发方式是目前我们使用的方式。如下图所示，我们提供了子工程脚手架来快速创建子工程，开发者无需做任何配置和额外学习成本，就可以像开

发 React 应用一样进行开发。



热更新

在开发过程中，我们希望我们的开发体验和开发单页应用的体验一致，也要支持热更新。由于我们的拆分，实际上有两个服务，即基座和子工程，所以我们以上图的方式完成了热更新的支持：在子工程的 module.hot 中通过再次触发基座工程中的 JSONP 钩子来通知基座工程，来再次触发 renderApp 达到子工程更新代码则页面热刷新的目的。主要代码如下：

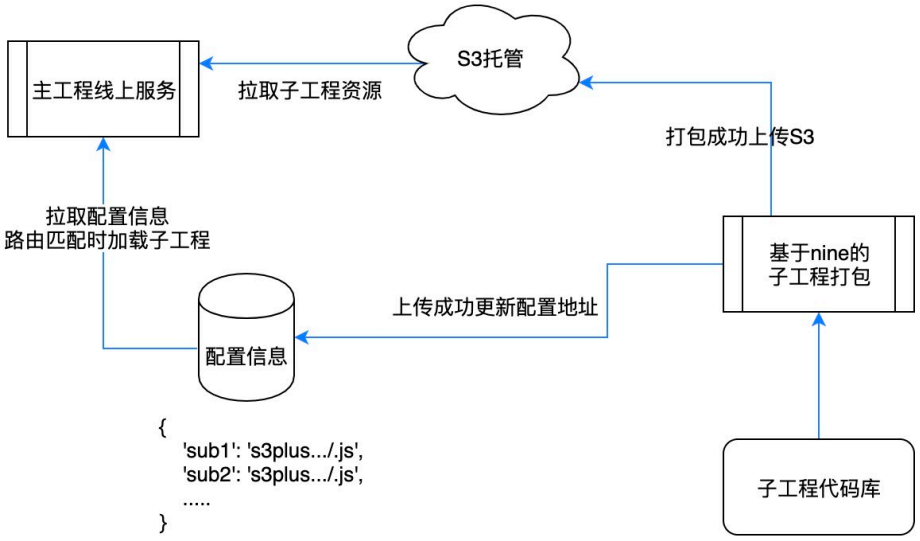
```
// 在子工程入口文件
import routes from './routes/index';
function registerApp(dep: any = {}): any {
    return {
        routes,
    };
}
if ((module as any).hot) {
    (module as any).hot.accept('./routes/index', (): any => {
        window.wmadSubapp(registerApp, true); // 支持子工程热加载的信息传递
    });
}
export default registerApp
```

Mock 数据

子工程目前 Mock 数据的方式有三种：一是在基座本地 Mock，这种 Mock 方式天然支持，因为基座工程基于外卖工程化 Nine 脚手架进行开发，本身支持本地 Mock。二是支持子工程本地 Mock。三是使用公共 Mock 服务 YAPI。目前子工程开发的 Mock 功能结合第一种方式和第三种方式进行。

部署方案

最后是部署方案，我们达成了独立部署上线的目的，即子工程发布不需要基座工程的参与。之前所有子业务线都在一个工程中，打包速度随着业务线的膨胀越来越慢，而如下的方案使得子工程的开发和部署完全独立，单个业务线的打包速度会非常快，从之前的分钟级别降到了秒级别。如下图所示，可以看到，子工程部署只需要把子工程打包，并在上传 CDN 之后，把配置信息更新即可，因为配置信息中有子工程新的资源地址，这样就达到了发布上线的目的。

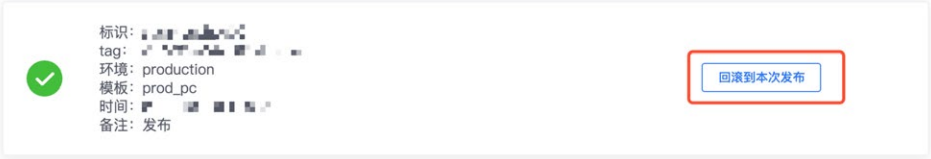


整个部署过程我们是托管到 Talos（美团内部自研的部署工具）上的，配置信息我们是托管到 Portm（美团内部自研的文件存储）上的（通过我们开发的 Talos 的插件

UpdatePubInfo-To-Portm 来更新我们的配置信息)。在静态资源上传到 CDN 之后，就可以更新配置信息，供主工程调用，也就完成了子工程上线的过程。利用美团现有服务，我们很迅速地完成了子工程单独部署上线的整个流程。

回滚方案

在部署方案中，我们通过 Talos 进行部署，它本身就带有回滚功能。得益于子工程的发布和普通工程的发布并没什么本质不同，都是将静态资源放置到 CDN 上，通过静态资源的的 contenthash 值来区分不同版本，所以回滚的时候，Talos 取到上个版本(或者某个前版本)的静态资源，再通过 Portm 更新我们的配置信息即可完成。整个过程和普通工程没有区别，发版人员只需简单地点下回滚按钮即可。



监控方案

改变了原有的开发模式后，我们还对几个关键节点进行了监控报警的埋点。利用美团 CAT(已经在 GitHub 上开源)和天网(美团内部的监控系统)，我们分别在子工程的配置信息、静态资源加载等节点上进行了埋点上报，统计子工程加载成功率，及时发现可能出现的子工程切换问题。具体情况如下图所示：

上方左图是按照端维度进行统计的示例，上方右图是 PC 端按照产品线统计加载成功数的示例。默认都是统计当天的数据，显示‘–’的表明当前没有数据。对资源加载的监控目前有三种类型：JSON、JS 和 CSS，资源加载失败的统计也包含这三种类型。天网的监控按照分钟级进行，每分钟内如果有加载失败就会发出报警，偶尔的报警可能是用户网络的问题，如果出现大批量的报警就要引起重视了。

总结

以上就是微前端在外卖商家广告端的实践过程。总的来说，我们完成了以下的目标：

- 按照领域（业务线）拆分工程，工程的可维护性得到提高，相关领域进行了内聚，无关领域进行了解耦。
- 子工程提供了 PC、H5、KA 三端的物理复用土壤，消除了工程膨胀问题，工程大小也变得可控。
- 子工程打包速度从分钟级降为秒级，提高了开发体验，加快了上线的速度。
- 子工程开发支持热更新，开发体验不降级。
- 子工程能够单独开发、单独部署、单独上线，业务线间互不影响。
- 整体工程改造成本低，插拔式开发，无侵入式代码，在正常业务开发的同时短期内就可以完成上线。
- 开发者学习成本低，完整地保留了单页应用开发的开发体验，开发者可快速上手。
- 目前在美团广告端，以微前端模式上线的子业务线已经有很多个。另外还有多个正在开发的微前端子工程，剩余在主工程中的子业务线后续也可以无痛迁移出来成为子工程。我们内部也在此过程中搜集了不少意见反馈，未来继续在实

践中进行思考和完善。在此过程中，我们深知还有很多做得不够完善甚至存在问题的地方，欢迎大家跟我们进行交流，帮我们提出宝贵意见或者给予指导。当然也欢迎大家加入我们团队，一起共建。

作者简介

张啸、魏潇、天尧，均为美团外卖前端团队研发工程师。

招聘信息

美团外卖广告前端团队诚招高级前端开发、前端开发专家。我们为商家提供变现服务平台，为用户提供优质广告体验，是外卖商业变现中的重要环节。欢迎各位小伙伴的加入，共同打造极致广告产品。感兴趣的同学可投递简历至：tech@meituan.com（邮件标题注明：美团外卖广告前端团队）

积木 Sketch 插件进阶开发指南

作者：韩洋　思琪　李肖　彦平

> The fewer sources of truth we have for a design system, the more efficient we are.——Jon Gold
>
> 设计系统的真理来源越少，效率就越高。——Jon Gold，知名全栈设计师

背景

1. 积木工具链体系

前段时间我们在美团技术团队公众号上发表了《积木 Sketch Plugin：设计同学的贴心搭档》一文，不曾想到，这个仅在美团外卖 C 端使用的插件受到了更多的关注，美团多个业务团队纷纷向我们抛出“橄榄枝”，表示想要接入以及并表达了愿意共同开发的意向，其它互联网同行也纷纷询问相关的技术，一时让我们有些“受宠若惊”。回想起写第一篇文章的时候，我们的内心还是有些不安的。作为 UI 同学的一个设计工具，有些 RD 甚至没有听说过 Sketch 这个名字，我们很认真地修改过上一篇文章的每一句措辞，争取让内容更丰富有趣，当时还很担心不会被读者接受。

积木 Sketch 插件的“意外走红”，确实有些出乎我们的意料，但正是如此，才让我们知道 UI 一致性是绝大部分开发团队面临的共性问题，大家对落地设计规范，提高 UI 中台能力，提升产研效率都有着强烈的诉求。为了帮助更多团队提升产研效率，我们成立了袋鼠 UI 共建项目组，将门户建设、工具链建设以及组件建设统一管理统一规划，并将工具链的品牌确定为“积木”，而积木 Sketch 插件便是其中重要的一环。

积木品牌 Logo

我们通过建立包含相同设计元素的统一物料市场，PM 通过 Axure 插件拾取物料市场中的组件产出原型稿；UI/UE 通过 Sketch 插件落地物料市场中的设计规范，产出符合要求的设计稿；而物料市场中的组件又与 RD 代码仓库中的组件一一对应，从而形成了一个闭环。未来，我们希望通过高保真原型输出，可以给中后台项目、非依赖体验项目提供更好的服务体验，赋予产品同学直接向技术侧输出原型稿的能力。

袋鼠 UI 工具链体系

2. 积木插件平台化

伴随着“积木”品牌的确立，越来越多的团队希望可以接入积木 Sketch 插件，其中部分团队也在和我们探讨技术合作的可能性。UI 设计语言与自身业务关联性很强，不同业务的色彩系统、图形、栅格系统、投影系统、图文关系千差万别，其中任意一环的缺失都会导致一致性被破坏，业务方都希望通过积木插件实现设计规范的落地。为

了帮助更多团队的 UI 同学提升设计效率，节约 RD 同学页面调整的时间，同时也让 App 界面具有一致性，从而更好地传达品牌主张和设计理念，我们决定对积木插件进行平台化改造。平台化是指积木插件可以接入各个业务团队的整套设计规范，通过平台化改造，可以使积木插件提供的设计元素与业务强关联，满足不同业务团队的设计需求。

积木 Skecth Plugin 平台化示意

积木插件原本只是外卖提升 UI/RD 协作效率的一次尝试，最初的目标仅是 UI 一致性，但是现在已经作为全面提升产研效率的媒介，承载了越来越多的功能。围绕设计日常工作，提供高效的设计元素获取方式，让工作变得更轻松，是积木的核心使命。如何推动设计规范落地，并且输出到各个业务系统灵活使用，是我们持续追寻的答案。而探寻研发和设计更为高效的协作模式也是我们一直努力的方向。

通过一段时间的平台化建设，目前美团已经有 7 个设计团队接入了积木插件，覆盖了美团到家事业部大部分设计同学，未来我们会持续推进积木插件的平台化建设，不断完善功能，期望能将积木插件打造成业界一流的品牌。

3. Sketch 插件开发进阶

第一篇文章可能是为数不多的入门教程，而本篇可能是你能找到的唯一一篇进阶开发文章。进阶开发主要涉及如何切换业务方数据，即选择所属业务方后，对应的组件、颜色等设计素材切换为当前业务方在物料市场中上传的元素；将承载组件库的

Library 文件转化为插件可以识别的格式，并在插件上展示，以供设计师在绘制设计稿时选择使用；一些优化运行效率，提升用户体验的方法。

Sketch 插件代码由于和业务强相关，且实现方式较为复杂，可能存在部分敏感信息，所以基本没有成熟的插件开源。在进行一些复杂功能开发时，我们也常常“丈二和尚摸不到头脑”，“要不这个功能算了吧”的想法也不止一次出现，可是每当开会看到旁边的设计同学在使用“积木”插件认真作图时，又一次次坚定了我们的信念，要不加班再试试吧，没准就能实现了呢？一次“委曲求全”，后面可能导致整个项目慢慢崩塌，所以我们一直以将积木插件打造成为业界领先的插件为信念。如果说看过了第一篇文章你已经知道了如何开发一款插件，那么通过本篇文章的学习你就可以真正实现一款可以与业务强关联且功能可定制的成熟工具，与其说是介绍如何开发一个进阶版的 Sketch 插件，不如说是分享给大家完成一个商业化项目的经验。

支持多业务切换

为了当面对“我们可以接入积木插件吗”这种灵魂拷问时不再手足无措，平台化进程迅速启动。平台化的核心其实就是当发生业务线变更时，物料市场中的素材整体同步切换，因此我们需要进行如下几个操作：首先建立全局变量，存储当前用户所述业务方信息及鉴权信息；用户选择功能模块后，根据用户所述业务方，拉取对应素材；处理 Library 等素材并渲染页面展示；根据素材信息变更画板中的相关 Layer。这部分主要介绍如何依靠持久化存储来实现业务切换的功能，就像在第一篇启蒙文档中说的那样，这里不会贴大段的代码，只会帮你梳理最核心的流程，相信你亲自实践一次之后，以后的困难都可以轻松解决。

1. 定义通用变量

功能模块展示的素材与当前选择的业务相关，因此需要在每个功能模块的 Redux 初始化状态中增加一些全局状态变量。比如所有模块都需要使用 businessType 来确定当前选择的业务，使用 theme 进行主题切换，使用 commonRequestParams 获取用户鉴权信息等。

```
export const ACTION_TYPE = 'roo/colorData/index';
const intialState = {
  id: 'color',
```

```
  title: '颜色库',
  theme: 'black',
  businessType: 'waimai-c',
  commonRequestParams: {
    userInfo: '',
  },
};
export default reducerCreator(intialState, ACTION_TYPE);
```

2. 实现数据交换

第一步：WebView 侧获取用户选择，将所选的业务方数据通过 window.postMessage 方法传递至插件侧。

```
window.postMessage('onBusinessSelected', item);
```

第二步：Plugin 侧通过 webViewContents.on( ) 方法接收从 WebView 侧传递过来的数据。Sketch 官方通过 Settings API 提供了一些类的方法来处理用户的参数设置，这些设置在 Sketch 关闭后依然会保存，除了存储一段 JSON 数据外，Layer、Document 甚至是 Session variable 都是支持的。

```
 webViewContents.on('onBusinessSelected', item => {
    Settings.setSettingForKey('onBusinessSelected', JSON.
stringify(item));
  });

// 除此之外，插件侧也可以通过 localStorage 向 WebView 注入数据
browserWindow.webContents.executeJavaScript(
      `localStorage.setItem("${key}",'${data}')`
);
```

第三步：当用户通过工具栏选择某一功能模块（例如“插画库”）时，会回调 NSButton 的点击事件监听，此时除了需要要让 WebView 展示（Show）以及获取焦点（Focus）外，还需要将第二步存储的业务方信息传入，并以此加载当前业务方的物料数据。

```
// 用户打开的功能模块
const button = NSButton.alloc().initWithFrame(rect)
  button.setCOSJSTargetFunction(() => {
```

```
    initWebviewData(browserWindow);
    browserWindow.focus();
    browserWindow.show();
  });

// 注入全局初始化信息
function initWebviewData(browserWindow) {
  const businessItem = Settings.settingForKey('onBusinessSelected');
  browserWindow.webContents.
executeJavaScript(`initBusinessData(${businessItem})`);
}
```

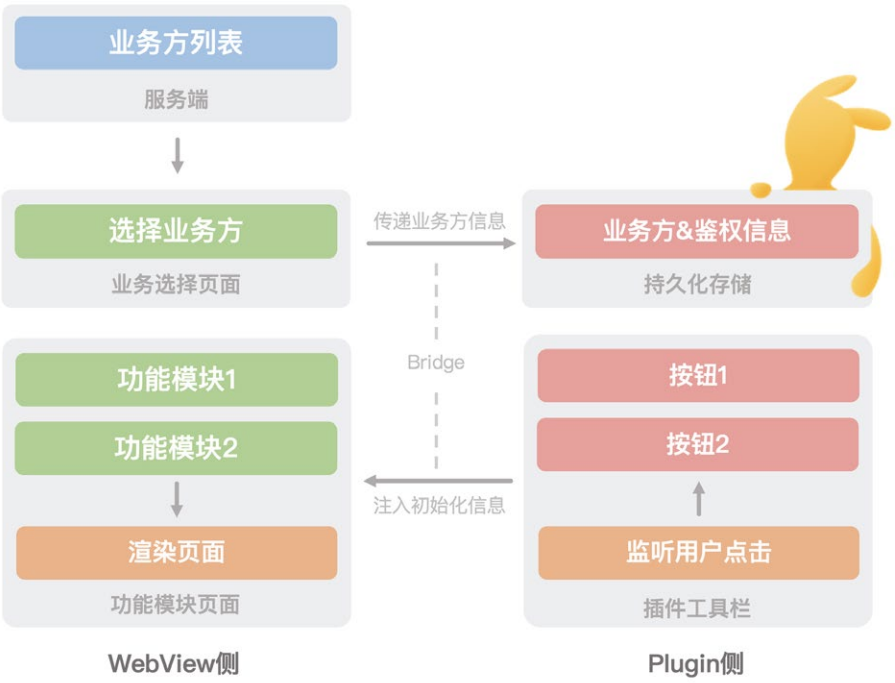
WebView 侧功能模块收到初始化信息，开始进行页面渲染前的数据准备。Object.keys() 方法会返回一个由给定对象的自身可枚举属性组成的数组，遍历这个数组即可拿到所有被注入的初始化数据，之后通过 redux 的 store.dispatch 方法更新 state 即可。至此实现业务切换功能的流程就全部结束了，是不是觉得非常简单，忍不住想亲自动手试一下呢?

```
ReactDOM.render(<Provider store={store}><App /></Provider>,
  document.getElementById('root')
);

window.initBusinessData = data => {
  const businessItem = {};
  Object.keys(data).forEach(key => {
    businessItem[key] = { $set: initialData[key] };
  });
  store.dispatch(update(businessItem));
};

const update = payload => ({
  type: ACTION_TYPE,
  payload,
});
```

3. 小结

有小伙伴会问，为什么 WebView 与 Plugin 侧需要数据传递呢，它们不都属于插件的一部分么? 根本原因是我们的界面是通过 WebView 展示的，但是对 Layer 的各种操作是通过 Sketch 的 API 实现的，WebView 只是一个网页，本身与 Sketch 并

无关系，因此必须使用 bridge 在两者之间进行数据传递。别担心，这里再带你把整个流程梳理一遍：①在插件启动后会从服务端拉取业务方列表；②用户在 WebView 中选择自己所属的业务方；③将业务方数据通过 bridge 传递至 Plugin 侧，并通过 Sketch 的 Settings API 进行持久化存储，这样就可以保证每次启动 Sketch 的时候无需再次选择所属业务方；④用户点击插件工具栏的按钮选择所需功能（例如色板库、组件库等），从持久化数据中读取当前所属业务方，并通知 WebView 侧拉取当前业务方数据。至此，整个流程结束。



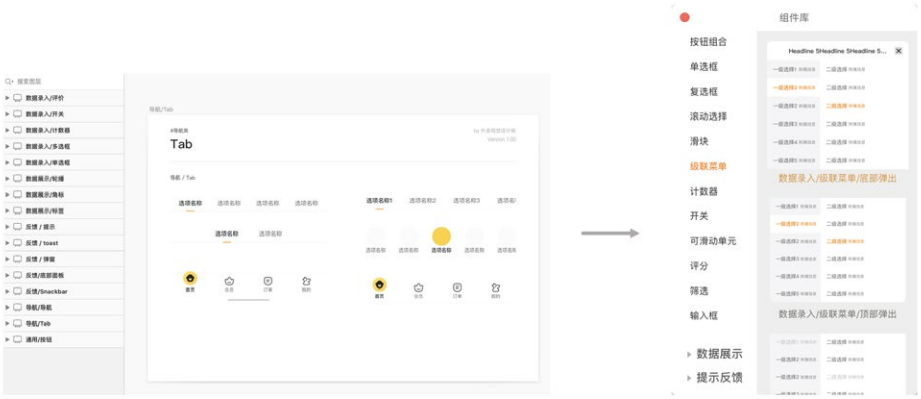
Library 库文件自动化处理

这部分将介绍如何将 Library 库文件转化为插件可以识别的 JSON 格式，并在插件上展示。

如果要问 Sketch 插件最重要的功能是什么，组件库绝对是无可争议的 C 位。在长期

的版本迭代中，随着功能的不断增加以及 UI 的持续改版，新旧样式混杂，维护极为困难。设计师通过将页面走查结果归纳梳理，制定设计规范，从而选取复用性高的组件进行组件库搭建。通过搭建组件库可以进行规范控制，避免控件的随意组合，减少页面差异；组件库中组件满足业务特色，同时具有云端动态调整能力，可以在规范更新时进行统一调整。

目前，我们将组件集成进 Sketch 供 UI 使用大致分为两个流派：一个是基于 Sketch 官方的 Library 库文件，设计师通过将业务中复用性高的 Symbol 组件归纳整理生成库文件（后缀 .sketch），并上传至云端，插件拉取库文件转化为 JSON 并在操作面板展示供选取使用；另一个则是采用类似 Airbnb 开源的 React-Sketchapp 这样的框架，它可以让你使用 React 代码来制作和管理视觉稿及相关设计资源，官方把它称作“用代码来绘画”，这种方案的实施难度较大，因为本质上设计是感性和理性的结合，设计师使用 Sketch 是画，而非带有逻辑和层级关系的写，他们对于页面的树形结构很难理解，上手成本较高，而且代码维护成本相对较大。我们不去评价哪种方案的好坏，只是第一种方案可以更好地满足我们的核心诉求。



Library文件　　插件中展示

Sketch 组件库处理效果示意

1. 订阅远程组件库

Library 库文件实际上是一个包含 components 的文档，components 包括了 Symbols、Text Styles 以及 Layer Styles 三类，将 Library 存储在云端就可以在不同文档甚至不同团队间共享这些 components。由于组件库实时指向最新，因此当其维护者更新库中的 components 时，使用了这些 components 的文档将会收到通知，这可以保证设计稿永远指向最新的设计规范。

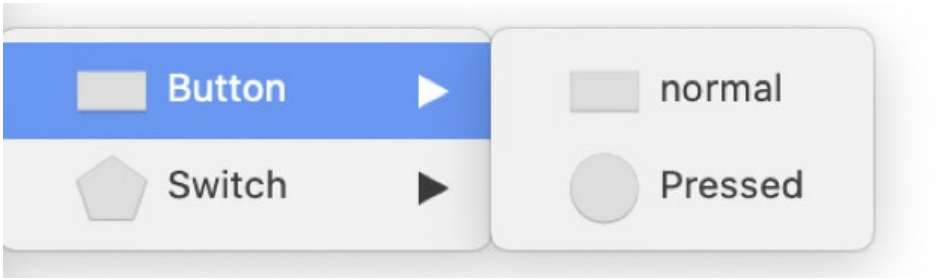
订阅云端组件库的方式很简单，首先创建一个云端组件库，具体可以参照上一篇文章，如果需要服务多个设计部门，则需要创建多个库，每个库有唯一的 RSS 地址；在插件中获取到这些 RSS 地址后，可以通过 Library.getRemoteLibraryWithRSS 方法对其进行订阅。

```
// 启动插件时添加远程组件库
export const addRemoteLibrary = context => {
  fetch(LibraryListURL, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
  })
    .then(res => res.json())
    .then(response => response.data)
    .then(json => {
      const { remoteLibraryList } = json;
      _.forEach(remoteLibraryList, fileName => {
        Library.getRemoteLibraryWithRSS(fileName, (err, library) => {
        });
      });
      return list;
    });
};
```

2. Library 库文件转换 JSON 数据

将 Sketch 的 Library 文件转换为 JSON 的过程，实际上就是转换为 WebView 可以识别格式的过程。因为积木插件是将组件按照一定分组展示在面板中供设计师选取，因此需要根据组件分类组织其结构。Sketch 原生支持采用“/”符号对其进行分组：

Group-name/Symbol-name，比如命名为 Button/Normal 和 Button/Pressed 的两个 Symbols 会成为 Button Group 的一部分。



Symbol 分组结构

实际中可以根据业务需要采用三级以上分组命名的方式，通过 split 方法将 Symbol 名称通过 “/” 符号拆分为数组，第一级名称、第二级名称等各级名称作为 JSON 结构的不同层级即可，具体操作可以参照如下示例代码：

```
const document = library.getDocument();
   const symbols = [];
   _.forEach(document.pages, page => {
     _.forEach(page.layers, l => {
       if (l.type && l.type === 'SymbolMaster') {
         symbols.push(l);
       }
     });
   });

// 对 symbol 进行分组处理，并生成 json 数据
for (let i = 0; i < symbols.length; i++) {
     const name = symbols[i].name;
     const subNames = name.split('/');
     // 去掉所有空格
     const groupName = subNames[0].replace(/\s/g, '');
     const typeName = subNames[1].replace(/\s/g, '');
     const symbolName = subNames.join('/').replace(/\s/g, '');
     result[groupName] = result[groupName] || {};
     result[groupName][typeName] = result[groupName][typeName] || [];
     result[groupName][typeName].push({
       symbolID:symbolID,
       name: symbolName,
     });
}
```

经过以上操作后，一个简化版的 JSON 文件如下方所示：

```
{
    "美团外卖 C 端组件库": {
        "icon": [{
            "symbolID": "E35D2CE8-4276-45A1-972D-E14A06B0CD23",
            "name": "28/ 问号"
        },{
            "symbolID": "E57D2CE8-4276-45A1-962D-E14A06B0CD61",
            "name": "27/ 花朵"
        }]
    }
}
```

3. Symbol 缩略图处理

WebView 默认是不支持直接显示 Symbol 供用户拖拽使用的，解决该问题的方案有两种：（1）通过 dump 分析 Sketch 的头文件发现，可以采用 MSSymbolPreviewGenerator 的 imageForSymbolAncestry 方法导出缩略图，该方法支持图片大小、颜色空间等多种属性设置，优势是较为灵活，可以根据需要进行任意配置，不过要承担后期 API 变更的风险；（2）直接采用 sketchDOM 提供的 export 方法，将 Symbol 组件导出为缩略图，之后在 WebView 中显示缩略图，当拖拽缩略图至画板时，再将其替换为 Library 中对应的 Symbol 即可。

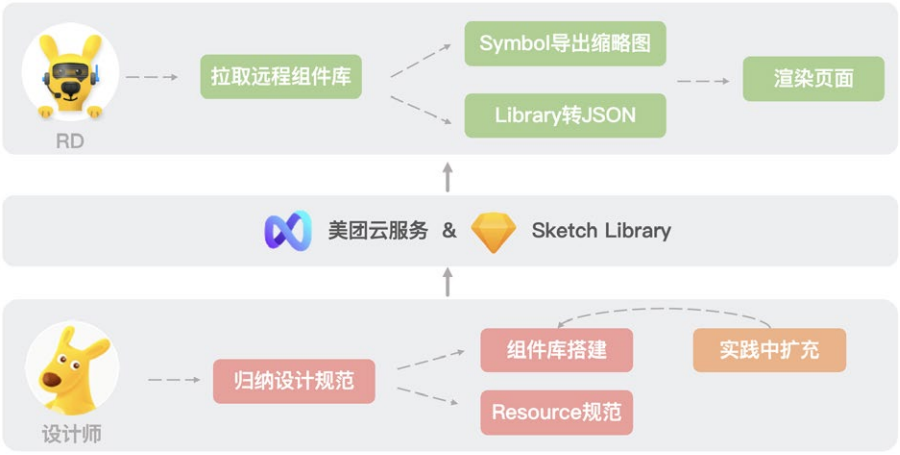
```
import sketchDOM from 'sketch/dom';

sketchDOM.export(symbolMaster, {
     overwriting: true,
     'use-id-for-name': true,
     output: path,
     scales: '1',
     formats: 'png',
     compression: 1,
});
```

4. 小结

以上就是实现平台化的一个基本流程了，不知道此时你有没有听得“云里雾里”。在这里，再把核心点带大家复习一下。本节主要讲了两件事情：第一，插件如何才能支

持多个业务方，即在插件的业务方列表中选择相关业务方，就可以切换对应的设计资源；第二，如何处理 Library 文件，将其转换为 JSON 供 WebView 展示使用。具体流程如下：

1. 不同设计组的 UI 同学制作完成包含各种 components 的 Library 后，通过后台上传至云端。
2. RD 同学根据当前使用者所属的设计团队拉取对应的包括 Library 在内的设计素材，颜色、图片，iconFont 等设计素材可以直接展示，可是 Library 文件不支持在 WebView 中直接显示，需要进行处理。
3. 根据和 UI 同学约定组件的命名规则，通过使用“/”分割，将第一级名称、第二级名称等各级名称作为 JSON 结构的不同层级，再通过 sketchDOM 提供的 export 方法将 Symbol 转换为 png 格式的缩略图即可在插件中显示。
4. 将选中的缩略图拖拽至 Sketch 的画板时，再将缩略图替换为 Library 中的真实 Symbol 即可。



Library 库文件处理小结

操作体验优化

完成了上述步骤后，就可以完成一款支持多业务方的插件了。但是随着积木 Sketch 插件接入的业务方越来越多，除了听到可以显著提升效率的褒奖外，对插件的吐槽声也常常传入我们的耳边。市面上成熟的插件也有很多，我们无法限制别人的选择，所以只能让积木变得更好用，本节主要介绍插件的优化方法。为了更好地倾听大家意见，积木插件通过各种措施了解用户的真实想法。首先积木插件接入美团内部的 TT（Trouble Tracker）系统，相比公司很多专业系统，TT 不带任何专业流程和定制化，只做纯流转，是一套适用于公司内部的、通用的问题发起、响应和追踪系统，用户反馈的问题自动创建工单并与对应 RD 关联，Bug 可以最快速修复；插件内部增加反馈渠道，用户反馈及时发送给相关 PM，作为下次功能排期的权重指标；插件内部增加多维度埋点统计，从设计渗透到高频使用两个方面了解 UI 同学的核心诉求。以下介绍了根据反馈整理的部分高优先级问题的解决方案。

1. 操作界面优化

很多 RD 在开发过程中，对界面美化往往嗤之以鼻，“这个功能能用就可以了”常常被挂在嘴边。难道 UI 的需求真的是中看不中用？一个产品设计师说过，最早的产品仅依靠功能就可以在竞品中脱颖而出，能不能用就成为了一个产品是否合格的标准。后来在越来越成熟的互联网环境中，易用性成了一个新的且更重要的标准，这时同类产品间的功能已经非常接近，无法通过不断堆叠功能产生明显差异。而当同类产品的易用性也趋于相近时，如何解决产品竞争力的问题就再一次摆在面前，这时就要赋予产品情感，好的产品关注功能，优秀的产品关注情感，可以让用户在使用中感受到温暖。

WebView 优化

当我们经过了仔细的功能验证，兴致勃勃的把第一版积木插件给用户体验时，本来满心欢喜准备迎接夸奖，没想到却得到了很多交互体验不好的反馈，“仅仅实现功能就及格了”这个理论在一像素都不肯放过的设计师眼中肯定行不通。原生 WebView 给

用户的体验往往不够优秀，其实只要一些很简单的设置就可以解决，但是这并不代表它不重要。

WebView 视图优化点举例

禁止触摸板拖拽造成页面整体“橡皮筋”效果，禁止用户选择（user-select），溢出隐藏等操作，使 WebView 具有“类原生”效果，都会提升用户的实际使用体验。

```
html,
body,
#root {
  height: 100%;
  overflow: hidden;
  user-select: none;
  background: transparent;
  -webkit-user-select: none;
}
```

除此之外在正式环境中（NODE_ENV 为 production），我们并不希望当前界面响应右键菜单，需要通过给 document 添加 EventListener 监听将相关事件处理方法屏蔽。

```
document.addEventListener('contextmenu', e => {
  if (process.env.NODE_ENV === 'production') {
    e.preventDefault();
  }
});
```

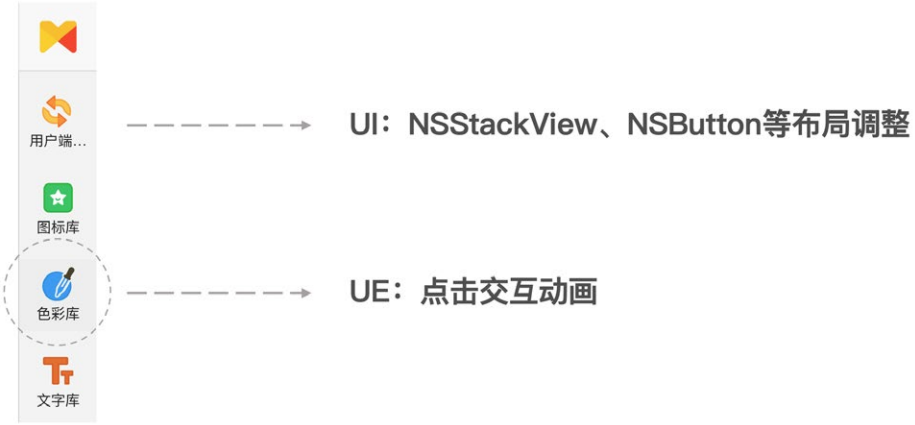
工具栏优化

Sketch 对于设计师的意义，就像代码编辑器对于程序员一样，工作中几乎无时无刻也离不开。在积木 Sketch 插件走出美团外卖，被越来越多的设计团队采用后，为了让它更加赏心悦目，UI 同学决定对工具条进行一次全新的视觉升级。原生界面开发指的是通过 macOS 的 AppKit 进行用户界面开发，在插件开发中一些需要嵌入 Sketch 面板的 UI 模块就需要进行原生界面开发，比如吸附式工具条就属于通过 macOS 原生 API 开发的界面。

原生开发既可以使用 Objective-C 语言，也可以使用 CocoaScript 通过写 JavaScript 的方式进行开发。CocoaScript 通过 Mocha 实现 JS 到 Objective-C 的映射，可以让我们通过 JS 调用 Sketch 内部 API 以及 macOS 的 Framework。在通过 CocoaScript 原生开发前需要了解一些基础知识：

1. 在使用相关框架前需要通过 framework() 方法进行引入，而 Foundation 以及 CoreGraphics 是默认内置的，无需再单独操作。
2. 一些 Objective-C 的 selectors 选择器需要指针参数，由于 JavaScript 不支持通过引用传递对象，因此 CocoaScript 提供了 MOPointer 作为变量引用的代理对象。

UI 调整一般分为三个部分：布局调整、动效调整、图片替换。下面的章节会进行逐一介绍。



新版积木工具栏效果图

布局调整

这里 UI 的需求是 NSButton 的宽度填充满整个 NSStackView，高度自定义。由于此功能看起来过于简单，当时认为估时 0.5 天绰绰有余，可是没想到搭进去了 1 个工作日加上 2 天周末的时间，因为无论如何设置 NSStackView 中子 View 尺寸都无法生效。

在顶住了周围人“UI 问题不影响功能使用，以后有时间再优化吧”的“舆论压力”后，终于在官方文档里面发现了线索：“NSStackView A stack view employs Auto Layout (the system’s constraint-based layout feature) to arrange and align an array of views according to your specification. To use a stack view effectively, you need to understand the basics of Auto Layout constraints as described in Auto Layout Guide.”简而言之，NSStackView 使用 constraints 的方式进行自动布局（可以类比 Android 中的 ConstraintLayout），在进行尺寸修改时，是需要添加锚点的，因此需要通过 Anchor 的方式进行尺寸修改。

```
// 创建工具条
const toolbar = NSStackView.alloc().initWithFrame(NSMakeRect(0, 0, 45,
400));
toolbar.setSpacing(7);
// 创建 NSButton
```

```
const button = NSButton.alloc().initWithFrame(rect)
// 设置 NSButton 宽高
button
    .widthAnchor()
    .constraintEqualToConstant(rect.size.width)
    .setActive(1);
button
    .heightAnchor()
    .constraintEqualToConstant(rect.size.height)
    .setActive(1);
button.setBordered(false);
// 设置回调点击事件
button.setCOSJSTargetFunction(onClickListener);
button.setAction('onClickListener:');
// 添加 NSButton 至 NSStackView 中
toolbar.addView_inGravity(button, inGravityType);
```

动效调整

NSButton 内置的点击效果大约 15 种，可以通过 NSBezelStyle 进行设置。积木插件工具栏并没有采用点击后 icon 反色的通用处理方式，而是点击后将背景色置为浅灰。如果想要自定义一些点击效果，只需在 NSButton 点击事件的回调中设置即可。

```
onClickListener:sender => {
  const threadDictionary = NSThread.mainThread().threadDictionary();
  const currentButton = threadDictionary[identifier];
   if (currentButton.state() === NSOnState) {
      currentButton.setBackgroundColor(NSColor.colorWithHex('#E3E3E3'));
   } else {
      currentButton.setBackgroundColor(NSColor.windowBackgroundColor());
   }
}
```

图片加载

Sketch 插件既支持加载本地图片，也支持加载网络图片。加载本地图片时，可以通过 context.plugin 的方法获取一个 MSPluginBundle 对象，即当前插件 bundle 文件，它的 url() 方法会返回当前插件的路径信息，进而帮助我们找到存储在插件中的本地文件；而加载网络图片则更加简单，通过 NSURL.URLWithString( ) 可以获得一个使用图片网址初始化得到的 NSURL 对象，这里要格外注意的是，对于网络图片请使用 https 域名。

```
// 本地图片加载
const localImageUrl =
       context.plugin.url()
      .URLByAppendingPathComponent('Contents')
      .URLByAppendingPathComponent('Resources')
      .URLByAppendingPathComponent(`${imageurl}.png`);

// 网络图片加载
const remoteImageUrl = NSURL.URLWithString(imageUrl);

// 根据 ImageUrl 获取 NSImage 对象
const nsImage = NSImage.alloc().initWithContentsOfURL(imageURL);
nsImage.setSize(size);
nsImage.setScalesWhenResized(true);
```

2. 执行效率优化

只有在设计稿中尽可能多地使用组件进行设计，并且将已有页面中的内容通过设计师的走查梳理逐渐替换成组件，才能真正通过建设组件库来进行提效。随着设计团队逐步将设计语言沉淀为设计规范，并将其量化内置于积木插件中，组件的数量越来越多，积木插件组件库作为 UI 同学使用最频繁的功能，需要格外关注其运行效率。

前置组件库加载

将组件库的加载逻辑前置，在打开文档时对远程组件库进行订阅操作。Sketch 所提供的了 Action API 可以使插件对应用程序中的事件做出反应，监听回调只需在插件的 manifest.json 文件中添加一个 handler 即可，添加了对于“OpenDocument”的监听，也就是告诉插件在新文档被打开时要去执行 addRemoteLibrary 这个 function。

```
{
      "script": "./libraryProcessor.js",
      "identifier": "libraryProcessor",
      "handlers": {
        "actions": {
          "OpenDocument": "addRemoteLibrary"
        }
      }
}
```

增加缓存逻辑

组件库的处理需要将 Library 文件转换为带有层级信息的 JSON 文件，并且需要将 Symbol 导出为缩略图显示。由于这个步骤较为耗时，因此可以将经过处理的 Library 信息缓存起来，并通过持久化存储记录已缓存的 Library 版本。若已缓存的版本与最新版本一致，且缩略图与 JSON 文件均完整，则可以直接使用缓存信息，极大的提高 Library 的加载速度。以下非完整代码，仅作示例：

```
verifyLibraryCache(businessType, libraryVersion) {
    const temp = Settings.settingForKey('libraryJsonCache');
    const libraryJsonCache = temp ? JSON.parse(temp) : null;

    // 1. 验证缓存版本信息
    if (libraryJsonCache.version.businessType !== libraryVersion) {
      return null;
    }

    // 2. 验证缩略图完整性
    const home = getAssertURL(this.mContext, 'libraryImage');
    const path = join(home, businessType);
    if (!fs.existsSync(path) || !fs.readdirSync(path)) {
      return null;
    }

    // 3. 验证业务库 Json 文件完整性
    if (libraryJsonCache[businessType]) {
      console.info(`当前 ${businessType} 命中缓存`);
      return libraryJsonCache;
    } else {
      return null;
    }
  }
}
```

3. 自定义 Inspector 属性面板

与 Objective-C 工程混合开发

随着各个设计组的组件库建设不断完善，抽离的组件数量不断增多，不少 UI 同学反馈 Sketch 原生组件样式修改面板操作不够便捷，无法约束选择范围，希望可以提供一种更有效的组件 overrides 修改方式，并且当修改“图片”、“图标”、“文字”等图

层时，可以和积木插件的这些功能模块进行联动选择。实现自定义 Inspector 面板功能既可以使操作更便捷，又可以对修改项进行约束。

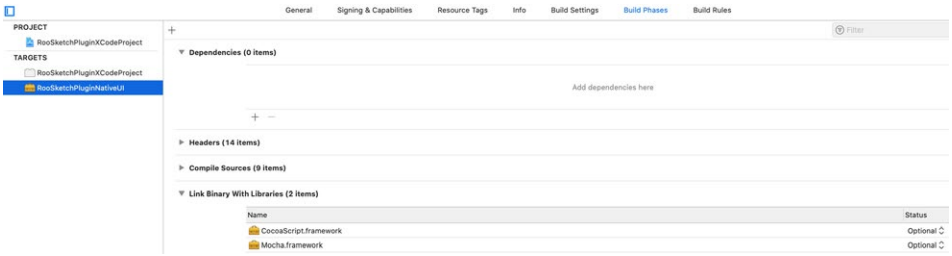
自定义属性面板功能的基本思想，是将组件从组件库拖至 Sketch 画板中时，组件的可修改属性可以显示在 Sketch 本身的属性面板上。我们引入了 Objective-C 原生开发以实现对 Sketch 界面的修改，为什么要使用原生开发？虽然官方提供了 JS API 并承诺持续维护，但这项工作一直处于 Doing 状态，而且官方文档更新缓慢，没有明确的时间节点，因此对于自定义 Native Inspector Panel 这种需要 Hook API 的功能，使用原生开发较为便捷，而且对于 iOS 开发者也更加友好，无需再学习前端界面开发知识。

Sketch Inspector 面板操作区优化

Xcode 工程配置

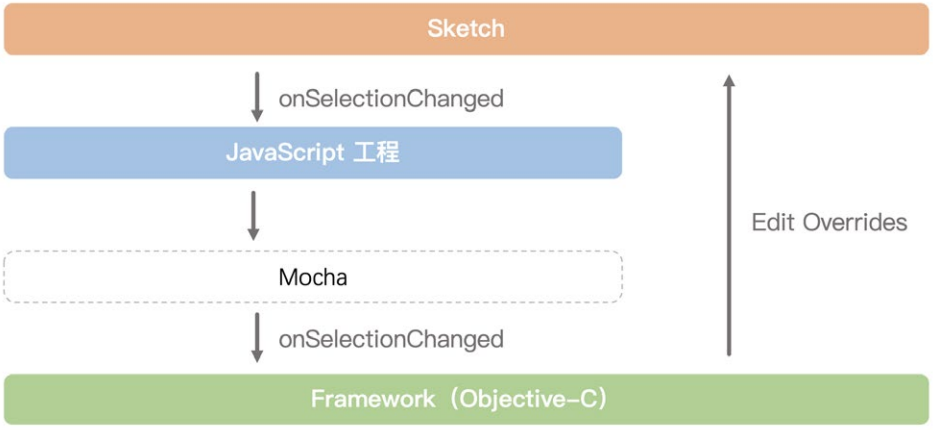
通过 Xcode 工程构建自定义属性面板，最终生成一个可以供 JS 侧调用的 Framework。可以参考上一篇文章介绍的方法创建 Xcode 工程，该工程在每次构建后会自动生成测试 Sketch 插件并放入对应的文件夹中。需要注意的一点是，这里生成的插件只是为了方便开发和调试，后面会介绍如何将 XCode 工程构建的 Framework 集

成至 JS 主工程中。



Xcode 工程配置示意

积木插件的主体功能使用 JS 代码实现，但是自定义属性选择面板使用 Objective-C 代码实现。为了实现积木插件的 JS 侧功能模块与 OC 侧模块之间的通信和桥接，这里借助了 Mocha 框架来实现相关的功能，Mocha 框架也被 Sketch 官方所使用，将原生侧的方法封装为官方 API 后暴露给 JS 侧。



Sketch 与插件 Framework 通信原理

组件选中时，Sketch 软件会回调 onSelectionChanged 方法给 JS 侧，JS 侧借助 Mocha 框架可以实现对 OC 侧的调用，同时将参数以 OC 对象的方式传递。JS 侧传递给 OC 侧的 Context 内容很丰富，包含了选中的组件、相关图层还有 Sketch 软件本身的信息。虽然 Sketch 没有提供 API，但是 Objective-C 语言本身具备 KVO

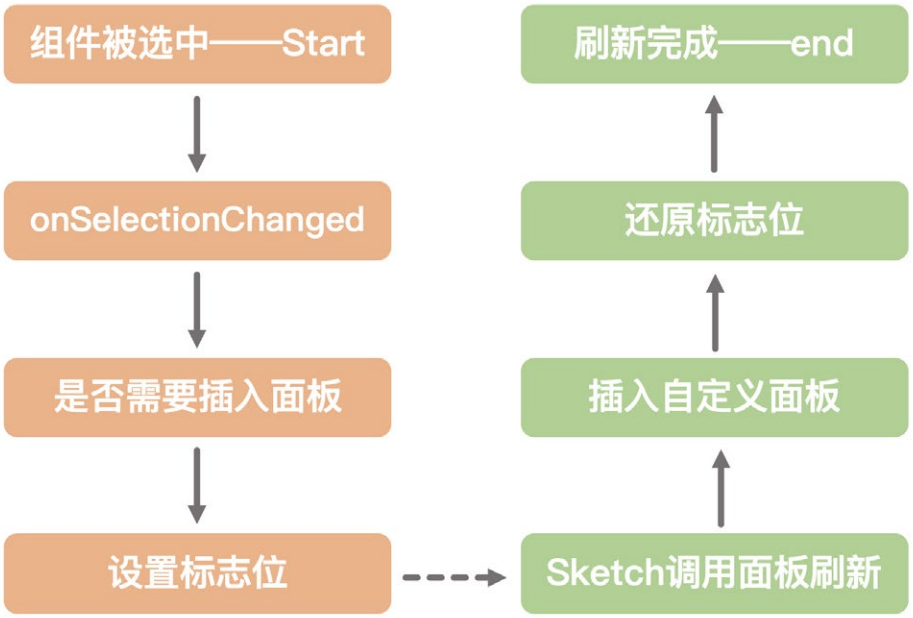
监听对象属性的能力，我们通过读取对应的属性值，就可以获取需要的对象数据。

```
+ (instancetype)onSelectionChanged:(id)context {

    [self setSharedCommand:[context valueForKeyPath:@"command"]];

    NSString *key = [NSString stringWithFormat:@"%@-
RooSketchPluginNativeUI", [document description]];
    __block RooSketchPluginNativeUI *instance = [[Mocha sharedRuntime]
valueForKey:key];

    NSArray *selection = [context valueForKeyPath:@"actionContext.
document.selectedLayers"];
    [instance onSelectionChange:selection];
    return instance;
}
```

Sketch 官方没有将属性面板的修改能力暴露给插件侧，通过查询 Sketch 头文件发现通过 reloadWithViewControllers: 方法可以实现属性面板刷新，但是在实际开发过程中发现在某些版本的 Sketch 上会出现面板闪动的问题，这里借助 Objective-C 的 Method Swizzle 特性，直接修改 reloadWithViewControllers: 的运行时行为解决。

```
[NSClassFromString(@"MSInspectorStackView") swizzleMethod:@
selector(reloadWithViewControllers:)
withMethod:@selector(roo_reloadWithViewControllers:)
error:nil];
```

Swizzle 方法会修改原始方法的行为，实际操作中只有在满足特定条件的情况下才应触发 Swizzle 后的方法。



Swizzle 方法触发条件

组件属性修改与替换原理

通过自定义面板可以修改组件的可覆盖项（即 override），目前可以应用可覆盖项的 affectedLayer 有 Text/Image/Symbol Instance 三种。设计师与开发者在此前对图层的格式进行了约定，保证我们可以按照统一的方式读取并替换图层的属性值。

替换文本

基于 class-dump，我们可以找出 Sketch 中声明的所有类的属性和方法，文本处理的策略是，找到图层中的所有 MSAvailableOverride 对象，这些对象即表示可用的覆盖项，对文本信息的修改实际上是通过修改 MSAvailableOverride 对象的 overridePoint 来实现的。

```
id overridePoint = [availableOverride valueForKeyPath:@"overridePoint"];
[symbolInstance setValue:text forOverridePoint:overridePoint];
```

更改样式

样式设置的策略，是找到当前选中组件对应的 Library 中相关样式的组件。由于所有的组件都遵循统一的命名格式，因此只要根据组件命名就能筛选出符合要求的组件。

```
// 命名方式：一级分类 / 二级分类 / 组件名称，基于图层获取对应 library
id library = [self getLibraryBySymbol:layer];
// 读取组件名称
NSString *layerName = [symbol valueForKeyPath:@"name"];
// 配置符合当前业务的 Predicate
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name
BEGINSWITH [cd] %@", prefix];
// 筛选符合 Predicate 的所有组件
NSArray *filterResult = [allSybmols
filteredArrayUsingPredicate:predicate];
```

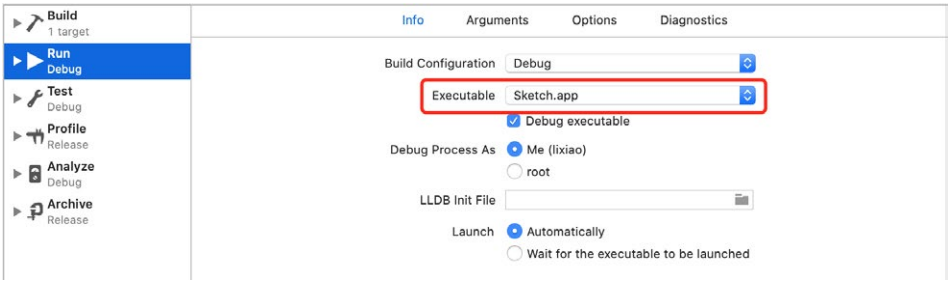
当使用者选中某一个样式后，插件会将设计稿上的组件替换为选中的组件，这里需要使用 MSSymbolInstance 中的 changeInstanceToSymbol 方法来实现。需要注意的是，changeInstanceToSymbol 仅仅替换了图层中的组件，但是并没有修改图层上组件的属性，对于位置和大小等信息需要单独进行处理。

```
// 在更新图层上的组件之前，我们需要把组件导入到当前 document 对象中
id foreignSymbol = [libraryController
importShareableObjectReference:sharedObject intoDocument:documentData];

// 更新图层上的组件
[symbolInstance changeInstanceToSymbol:localSymbol];
```

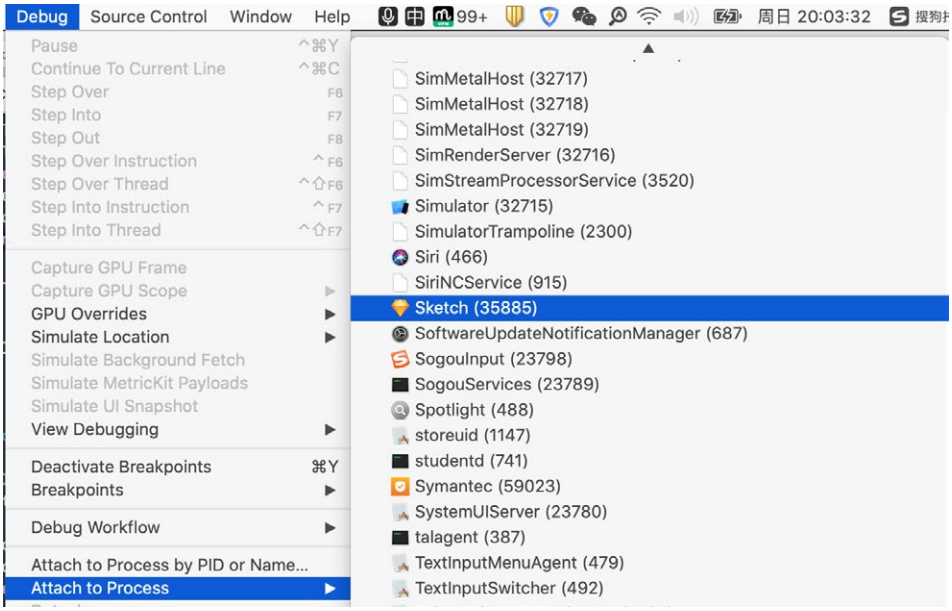
调试技巧

OC 侧开发的最大问题，在于没有官方 API 的支持。因此调试器就显得非常重要，单步调试可以让我们非常方便地深入到 Sketch 内部了解 Document 内部的数据结构。调试环境需要配置，但足够简单，并且对于开发效率的提升是指数级的。

1. 对构建 Scheme 的配置。



2.Attach 到 Sketch 软件上，这样就可以实现断点调试。



与当前 JS 工程混合编译

1. 通过 skpm 中内置的 @skpm/xcodeproj-loader 编译 XCode 工程，并将产物 framework 拷贝至插件文件夹。

```
const framework = require('../../RooSketchPluginXCodeProject/
RooSketchPluginXCodeProject.xcworkspace/contents.xcworkspacedata');
```

2. 通过 Mocha 提供的 loadFrameworkWithName_inDirectory 方法，设置 Framework 的名称及路径即可进行加载。

```
function() {
    var mocha = Mocha.sharedRuntime();
    var frameworkName = 'RooSketchPluginXCodeProject';
    var directory = frameworkPath;

    if (mocha.valueForKey(frameworkName)) {
      console.info('JSloadFramework: `' + frameworkName + '` has
loaded.');
      return true;
    } else if (mocha.loadFrameworkWithName_inDirectory(frameworkName,
directory)) {
      console.info('JSloadFramework: `' + frameworkName + '`
success!');
      mocha.setValue_forKey_(true, frameworkName);
      return true;
    } else {
      console.error('JSloadFramework load failed');
      return false;
    }
  }
```

3. 调用 framework 中的方法。

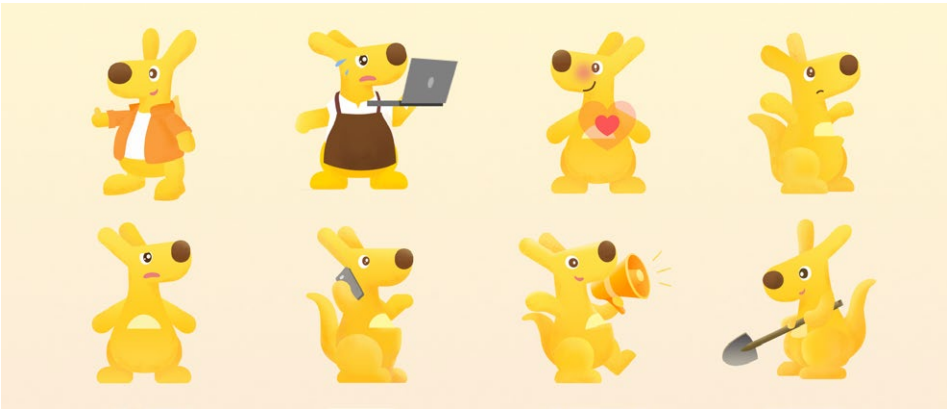
```
// 找到已经被加载的 framework
 const frameworkClass = NSClassFromString('RooSketchPluginNativeUI');
// 调用暴露的方法
frameworkClass.onSelectionChanged(context);
```

一起拼积木

目前，积木插件已经在美团到家事业部遍地开花，我们希望未来积木品牌产品可以在更大范围内得到应用，帮助更多团队落地设计规范，提升产研效率，也欢迎更多团队接入积木工具链。“不忘初心，方得始终”，就像第一篇启蒙文章中说的那样，我们除了希望制作一流的产品，也希望积木插件可以让大家在繁忙的工作中得以喘息。我们会继续以设计语言为依托，以积木工具链为抓手，不断完善优化，拓展插件的使用场景，让设计与开发变得更轻松。

总有人在问，积木插件现在好用吗？我想说，还不够好用。但是每次评审需求时看到旁边的设计师在认真地使用我们的插件作图，看到积木插件爱好者为我们制作表情包帮助我们推广，我们深知唯有交付最棒的产品，才能不辜负大家的期待。

平台化二期的需求刚刚确定完毕，人力分配排期结束，我们又想了一大波令你拍手称赞的功能，马上就要踏上新的征程。夜深了，看着窗外人家的灯，一个个熄灭，夜空也变得越来越明亮。我们的目标，是星辰大海。



使用积木插件插画库制作的表情包 Design by 雪美

致谢

感谢外卖技术部晓飞、彦平、瑶哥、云鹏、冰冰对项目的大力支持。 感谢到家事业部优秀的设计师冉冉、昱翰、淼林、雪美、田园、璟琦。 感谢闪购技术团队章琦、CRM 团队的怡婷、CI 王鹏协助技术开发。

参考文献

百度 Sketch 插件开发总结
爱奇艺产品工作流优化：搭建组件库做高 ROI
阿里重磅开源中后台 UI 解决方案 Fusion
Painting with Code
Sketch Developers Discussion

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，欢迎加入外卖 App 大家庭。感兴趣的同学可投递简历至：tech@meituan.com（邮件主题请注明：美团外卖前端）。

积木 Sketch Plugin：设计同学的贴心搭档

作者：韩洋　昱翰　云鹏　沛东

| A consistent experience is a better experience.——Mark Eberman |

一致的体验是更好的体验。——Mark Eberman《摘自设计师的 16 句名言》

背景

1.UI 一致性项目

积木（Tangram）Sketch 插件源于美团外卖 UI 的一致性项目，该项目自 2019 年 5 月份被提出，是 UI 设计团队与研发团队共建的项目，目的是改善用户端体验的一致性，提升多技术方案间组件的通用性和复用率，整体降低视觉改版的研发成本。

一直以来，外卖业务都处于高速发展阶段，人员规模在不断扩大，项目复杂度在持续增加。目前平台承载了美团餐饮、商超、闪购、跑腿、药品等多个业务品类，用户入口也覆盖了美团 App 外卖频道、外卖 App、大众点评等多个独立应用。因为客户端一直比较侧重业务开发，为了满足业务快速上线的需求，UI 组件并没有统一的实现，而是分散到各个业务场景中，在开发过程中因 UI 缺乏同一的标准而导致以下问题不断凸显：

UI/UE 层面

① UI 缺乏标准化的设计规范，在不同 App 及不同语言平台上设计风格不统一，用户体验不一致。

② 设计资源与代码均缺乏统一的管理手段，无法实现积累沉淀，无法适应新业务的开发需求。

RD 层面

① 组件代码实现碎片化，存在多次开发的情况，质量难以得到保证。

② 各端代码 API 不统一，维护拓展成本较高，变更主题、适配 Dark Mode 等需求难以实现。

QA 层面

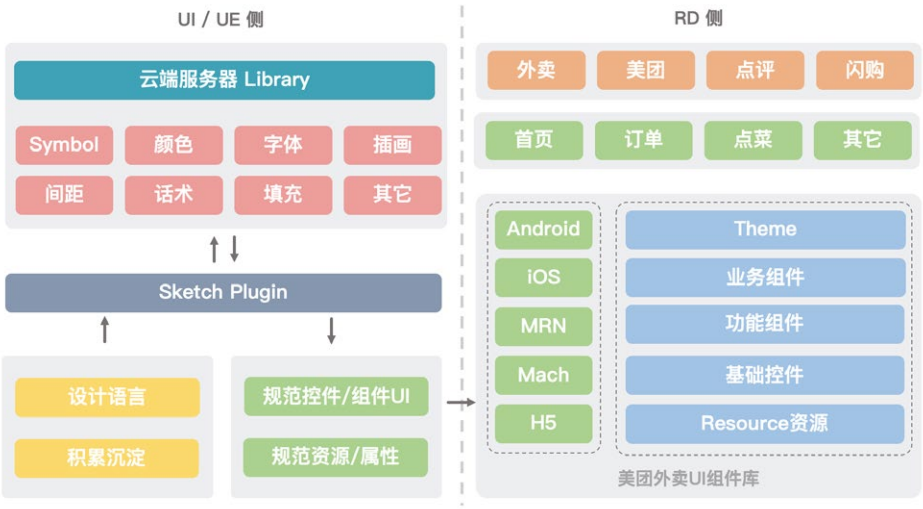
重复走查，频繁回归，每次发版均需验证组件质量。

PM 层面

版本迭代效率低，版本需求吞吐量低，不能满足业务的快速拓展能力。

基于上述开发工作中的切实痛点，以及未来可预见的对客户端能力的开发需求，我们迫切需要一套统一的 UI 设计规范，以此沉淀出设计风格，建立统一的 UI 设计标准，从而抽离成熟的业务场景，提供高质量、可扩展、可统一配置的同时能基于 Android/iOS/MRN/Mach 组件开发的代码库，且具备支持多业务高层次的代码复用能力，提高 UI 业务的中台能力，使项目具有高度一致性。

我们通过积木 Sketch 插件来落地设计规范，可以保证设计元素均从既定设计标准中获取，产出符合业务设计语言的设计稿，而各平台 UI 组件库中也有对应实现，从而使积木插件成为 UI 一致性的抓手，最终可以减少开发成本，提升交付质量，服务好我们美团的多个业务团队。



外卖 UI 一致性项目

2. Sketch & Sketch Plugin

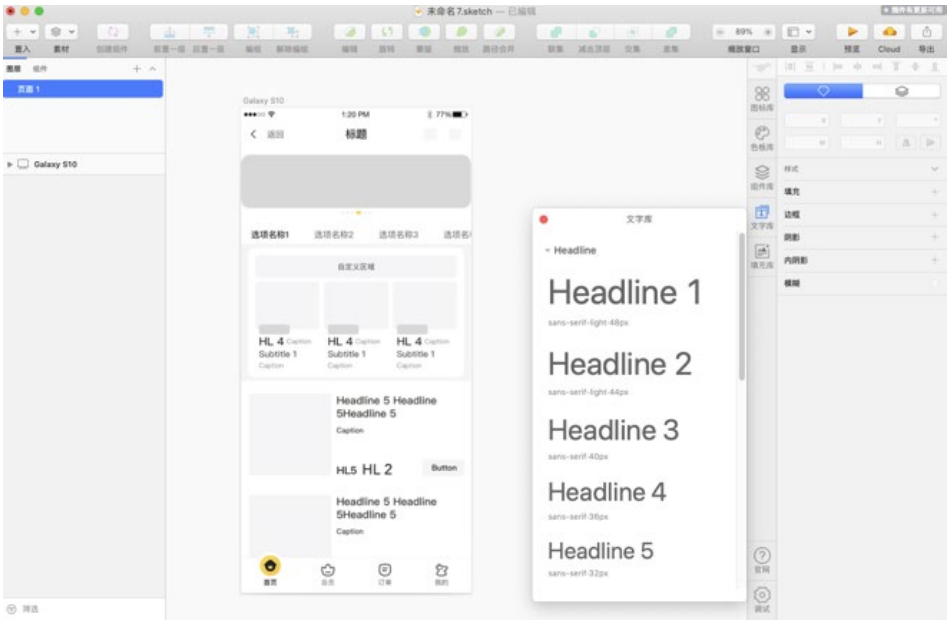
要想保持 UI 一致性，就不能打破规则。从设计阶段颜色的选择、字体的规范、控件的样式到 RD 开发阶段代码的统一管理、API 的制定、多端的实现方式，都必须遵守一套规则，而 Sketch Plugin 建设则是让规范落地执行的解决方案。

在讨论其重要性之前，我们首先简单介绍一下 Sketch：Sketch 是一个设计工具包，由总部位于荷兰海牙的 BohemianCoding 团队开发，该团队成员目前不足百人，来自全球多个国家，通过互联网远程协作开发，属于典型的高效开发团队。

Sketch 容易理解且上手简单；可与团队中的每个人创建、更新和共享所有 Symbol 组件，实现设计资源的共享和版本管理，从此告别“final-final-final-1”；其版本迭代速度非常快，且能不断添加新功能，满足用户的需求，更符合互联网时代；Sketch 可以使用真实数据进行设计。目前，我们设计团队已经全面使用 Sketch 进行设计。

设计语言包括 Iconfont、色板、文字规范、话术、插画、动画、组件等。其实它并不是一个抽象的概念，比如大家提到“美团”就会想起“美团黄”，想到可爱的“袋鼠”，想到那些骑着摩托车、穿着印有“美团外卖”亮黄色衣服的骑手小哥。通过设

计语言，我们可以更好地传达品牌主张和设计理念。UI 团队逐步将设计语言沉淀为设计规范，并将其量化内置于积木 Sketch Plugin 中，使产出的设计稿和 RD 代码库中的组件一一对应，从而形成一个完整的闭环，进而可加速整个业务的交付流程。



使用 Sketch Plugin 可以快速设计出标准页面

3. 积木 Sketch 插件项目

其实，市面上已存在类似插件，为什么我们还要自己动手开发呢？因为 UI 设计语言与自身业务关联性很强，不同业务的色彩系统、图形、栅格系统、投影系统、图文关系千差万别，其中任意一环的缺失都会导致一致性被破坏。现有插件所提供的通用设计元素无法满足外卖设计团队的需求，开发一款可以与业务强关联且功能可定制的插件，显得尤为重要。

此外，统一的品牌符号、品牌特征，也有助于加深产品在用户心中的印象，统一的颜色和交互形式能帮助用户加深对产品的熟悉感和信任感，一个好的设计语言本身可以在体验上为产品加分，也能够更好创造一致性的体验。

积木 Sketch 插件经过一段时间的建设，目前已具备 Iconfont、标准色板、组件库、数据填充、文字模板等功能。

我们通过 Iconfont 可以从美团的图标库中拉取设计团队上传的 SVG 图标，并直接应用于设计稿；标准色板可以限定设计师的颜色使用范围，确保设计稿中的颜色均符合设计规范；组件库中包含从外卖业务中抽离的基本控件与通用组件，具有可复用和标准化的特点，并与不同语言平台组件库中的代码一一对应，使用组件库中的组件进行设计，可以提升 UI 的设计效率、开发效率以及走查效率；数据填充库可以实现图片填充和文本填充，图片包含了商品及商家素材，文字则包含了菜品、商铺名等信息，通过数据填充可以使设计师采用真实数据进行填充，让设计稿更为直观，也更贴近线上环境；文字模板中内置了 Head、SubTitle、Body、Caption 的使用规范，根据设计稿中文字的位置，点击文字图层即可直接应用字体、行高、字距等属性。

此外，我们还根据设计同学的使用反馈，不断增添新功能。同时也在拓展插件的使用场景，增加业务线切换功能，使积木插件可以为更多的团队服务，并期待它能成为更多设计师的“贴心搭档”。

积木 Sketch Plugin 已支持功能

4. 为什么要写这篇文章?

相信你读完上面的内容，肯定迫不及待的想了解一下 Sketch 插件，以此迅速提升自己团队开发效率了吧?

其实在开始之前，我们可先了解一些不利的条件。第一点，由于 Sketch 更新速度极快，但是官方文档却十分简单且陈旧，因此很多知名的 Sketch Plugin 因每次 API 的变更过大纷纷放弃维护；第二点，由于开发技术栈混乱，成熟项目一般还未开源，而开源的项目基本上没有什么参考价值，绝大多数都是“update 3 years ago”；最后一点，macOS 开发资料更是少的可怜。

我们阅读了大量的文档却没有理清头绪，仿佛很多 Wiki 讲到关键地方，比如某个非常期待的功能是怎么实现的时候，作者竟然一笔带过，让人摸不到头脑。知乎上一篇 Sketch Plugin 的科普文，很多网友会评论“求教学视频，我可以花钱买的”。经过一步步踩坑，我们就总结了一些开发经验，为了避免大家“重复踩坑”，晚上可以早点下班陪陪家人，我们决定写一篇文章记录下开发的过程。虽然比起那些已经更新多版的成熟项目，但还有不少的差距，至少可以让大家不再那么迷茫。

当然，即使你觉得自己是个“跟 Sketch 八竿子打不着”的开发同学，我们也觉得这篇文章同样也值得阅读，因为你会通过本文接触到前端、移动端、桌面端、服务端的各种开发知识。我们都知道，越来越多的公司开始喜欢招全栈工程师，像 Facebook 基本上只招全栈工程师。你心里是不是在想：“是不是在搞笑啊？不过一个插件而已?”先别轻易下结论。

准备好了吗？盘它！

准备放手 Coding 之前

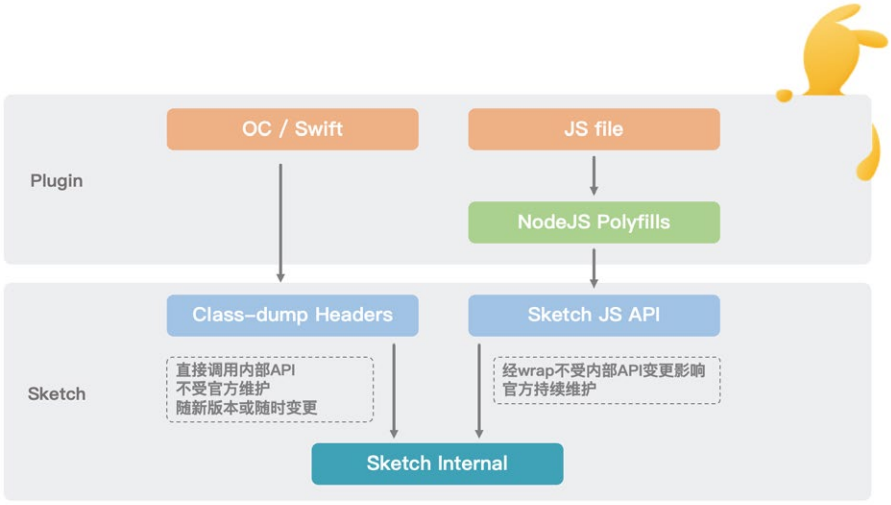
好，先别着急敲击键盘。毕竟我们连使用哪种语言去开发都没决定，这曾经也是困恼我们许久的一个问题。目前 Sketch Plugin 开发大概有两种方式：

① 使用 JavaScript + CocoaScript 的混合开发模式，Sketch 团队官方维护了一套 JS API，并在开发者官网写了一句非常振奋人心的话：“ Take advantage of ES6, access macOS frameworks and use the Sketch APIs without learning Objective-C or Swift.”

理想很美满，但现实很骨感。这个 API 目前还不算完善，很多功能无法实现，因此我们需要搭配 CocoaScript 访问更丰富的内部 API。

② 直接采用 Objective-C 或 Swift，并搭配 macOS 的 UI 框架 AppKit 进行开发，简单粗暴，并且可以利用 OC 运行时直接调用 Sketch 内部 API。但这里要特别提醒一下，你要承担的风险是：随着 Sketch 的不断更新，内部 API 的命名和使用方式可能会发生较大变化，很多知名插件都因此放弃更新。

本文采用了“混合开发模式”进行讲解，希望能够给你一些小启发。



Sketch 开发原理

1. Sketch Plugin 开发流派

开发流派	官方推荐	原生开发
开发语言	Javascript + CocoaScript	Objective-C或Swift
相关技术	任意前端框架React、VUE等，部分Objective-C和AppKit 知识辅助开发	通过 dump出的 Sketch-Headers，使用Objective-C和AppKit 开发
运行机制	CocoaScript 提供了Cocoa frameworks的Bridge，并通过Mocha 实现JS到Objective-C的 Bridge	Sketch本身采用OC开发，通过OC运行时调用内部API实现对图层的操作
工程开发	VSCode + Webpack等任意前端打包框架	XCode + Cocoapods或Carthage等

2. 环境配置

Skpm（Sketch Plugin Manager）是 Sketch 提供的用于 Plugin 创建、Build 以及发布的官方工具。Skpm 采用 Webpack 作为打包工具，当然如果你对前端知识足够熟悉，也可以采用 Rollup 或者 roadhog。但是，为了防止遇到各种各样的报错，这里并不建议你这么做。

Skpm 提供了一系列帮助快速入门的模板，最有用的莫过于 skpm/with-webview，它可以帮助我们创建一个基于 WebView 展示的 Demo 示例，而且 Skpm 会在构建完成后，自动创建一个 Symbolic Link 将插件添加到 Sketch 的安装目录，使 Plugin 立即可用。

```
// 基于 webpack 的 Sketch 官方打包工具 skpm
npm install -g skpm
// 创建示例工程
skpm create my-plugin --template=skpm/with-webview
//Install the dependencies
npm install
// 构建插件
npm run build
```

3. 项目结构

Plugin Bundle 按照上面的步骤操作完成后，我们会得到如下插件目录，它以标准化的分层结构存储了源码文件以及构建生成的 Sketch 插件安装包。这里没有使用官方文档中最简单的 Demo，而是使用目前开发中最为常用的 With-Webview 模板进行分析，以免出现学完“1+1”后遇到的全是“微积分”问题，并且大部分插件均是在此基础上进行拓展。

目录中的参数，相信你在看完注释后马上就能明白。可是如果此前没有前端开发经验，可能不了解在经过 Webpack 打包后，脚本文件的文件名会发生变更，比如 resources 中的 webview.js 经过打包后会储存在插件的 Resources 文件夹中，而文件名则变更为 resources_webview.js，因此在进行代码编写时，如果需要在 html 中引用此文件，也要使用打包后的文件名，即：。这里有个小技巧，如果你不知道脚本文件打包后的文件名及路径，建议先使用 Webpack 进行编译，然后查看其在打包后的 Plugin 中的位置和名称，然后再进行引用。

```
├── assets // 资源文件夹，如需更改需在 package.json 中的 skpm.assets 中设置
├── my-plugin.sketchplugin    //skpm 构建过程生成的插件包
│   └── Contents
│       ├── Resources
│       │   └── _webpack_resources
│       │   └── resources_webview.js
│       │   └── resources_webview.js.map
│       └── Sketch
│           ├── manifest.json
│           ├── __my-command.js
│           └── __my-command.js.map
├── package.json
├── webpack.skpm.config.js
├── resources // 资源文件
│   ├── style.css
│   ├── webview.html
│   └── webview.js
└── src // 需要被 webpack 打包的脚本文件以及 manifest 清单文件
    ├── manifest.json
    └── my-command.js
```

Manifest

你没有看错！ plugin 中也有 manifest.json，它与其它平台比如 Android 开发中的清单文件意义相同。清单文件记录了作者信息、描述、图标以及获取更新的途径等等。想想看，每天熬夜加班写代码，总得有个地方把你的名字记录下来吧。但 manifest 最重要的作用其实是告诉 Sketch 如何运行插件，以及如何将插件集成进 Sketch 的菜单栏中。

commands 使用一个数组，记录了插件所提供的所有命令。比如下面的例子，当用户从菜单栏点击“显示工具栏”这个条目时，就会执行 script.js 中的 function showPlugin() 。menu 则提供了插件在 Sketch 菜单栏中的布局信息，Sketch 会在插件被加载时初始化菜单。

```
{
  "commands": [
   {
      "name": " 显示工具栏 ",
      "identifier": "roo-sketch-plugin.toolbar",
      "script": "./script.js",
      "handlers": {
        "run": "showPlugin"
      }
    }
  ],
  "menu": {
    "title": " 口外卖积木 SketchPlugin 工具栏 ",
    "items": ["roo-sketch-plugin.toolbar"]
  }
}
```

package.json

简单来说，只要你的项目中用到了 NPM，根目录下就会自动生成 package.json 文件。Node.js 项目遵循模块化的架构，package.json 定义了这个项目所需要的各种模块以及配置信息。使用 npm install 命令会根据这个配置文件，自动下载所需的模块，也就是配置项目所需的运行和开发环境。

非常值得称赞的是，Plugin 开发中对于网络请求、I/O 操作以及其它功能，可以使用与 Node.js 兼容的 polyfill，其中许多常用 modules 已经预装到了 Sketch 中，比如 console、fetch、process、querystring、stream、util 等。

这里你只需要知道以下几点：

- 需要参与 Webpack 打包的脚本文件必须在 resources 目录下声明，否则不会参与编译（重点！考试要考！）。
- assets 目录需要配置在 skpm.assets 下。
- 常用的命令可以定义在 scripts 中方便直接调用。
- dependencies 字段指定了项目运行所依赖的模块，devDependencies 指定项目开发所需要的模块。

```
{
  "name": "roo-sketch-plugin",
  "author": "hanyang",
  "description": "外卖积木 Sketch plugin，UI 同学好喜欢~",
  "version": "0.1.0",
  "skpm": {
    "manifest": "src/manifest.json",
    "main": "roo-sketch-plugin.sketchplugin",
    "assets": ["assets/**/*"]
  },
  "resources": [
    "src/webview/template/webview.js"
  ],
  "scripts": {
    "build": "rm -rf roo-sketch-plugin.sketchplugin && NODE_
ENV=development skpm-build",
  },
  "dependencies": {},
  "devDependencies": {}
}
```

4. API Reference

Javascript API

由于使用了与 Safari 相同的 JS 引擎，Plugin 脚本可以获得完整 ES6 支持。官方的

JavaScript API 由 Sketch 团队维护，并允许访问和修改 Sketch 文档，通过 API 可以向 Sketch 用户提供数据并提供一些基本的用户界面集成。

```
// 访问、修改和创建文档从 color 到 layer 再到 symbol 等方方面面
var sketchDom = require('sketch/dom')
// 对于异步操作，JavaScript API 提供了 fibers 延长 contex 的 lifeTime
var async = require('sketch/async')
// 直接在 Sketch 中提供图像或文本数据，DataSupplier 直接与 Sketch 用户界面集成。
var DataSupplier = require('sketch/data-supplier')
// 无需重新 build 的情况下显示通知以及获取用户输入
var UI = require('sketch/ui')
// 保存图层或文档的自定义数据，并存储插件的用户设置。
var Settings = require('sketch/settings')
```

CocoaScript Syntax

CocoaScript 通过赋予了 JavaScript 调用 Sketch 内部 API 以及 macOS Cocoa frameworks 的能力，这意味着除了标准的 JavaScript 库外，还可以使用许多很棒的类与函数。CocoaScript 建立在苹果的 JavaScriptCore 之上，而 JavaScriptCore 是为 Safari 提供支持的 JavaScript 引擎。

因此，当你使用 CocoaScript 编写代码的时候，你就是在写 JavaScript。CocoaScript 中的 Mocha 实现 JS 到 Objective-C 的 Bridge，虽然 Mocha 包含在 CocoaScript 中，但文档仍保留在原始 Github 中。因此，你在 CocoaScript 的 Readme 中看不到任何语法教程。这里一个诀窍是，如果你想了解 Mocha 将原生的 Sketch Objects 通过 bridge，从 Objective-C 传递到 JavaScript 层的属性、类或者实例方法的信息，可以将其通过 console 打印出来：

```
let mocha = context.document.class().mocha()
console.log(mocha.properties())
//OC
[executeOperation:withObject:error:]
//CocoaScript
executeOperation_withObject_error()
```

通过 CocoaScript 提供的 Bridge 使用 JavaScript 调用 Objective-C 的基本语法如下：

- Objective-C 的方括号语法“[ ]”转换为 JavaScript 中的点“ . ”语法。
- Objective-C 的属性导出到 JavaScript 时 Getter 为 object.name() 而 Setter 为 object.name = ‘Sketch’。
- Objective-C 的 selectors 被暴露为 JavaScript 的代理方法。
- “ : ”冒号被转换为下划线“ _”, 最后一个下划线是可选的。
- 调用带有一个下划线的方法需要加倍为两个下划线 : sketch_method 变为 sketch__method。
- selector 的每个 component 被连接成不带有分隔符的单个字符串。

5. Actions

行为定义

Action 指的是由于用户交互而在应用程序中发生的事件，比如“打开文档”、“关闭文档”、“保存”等。Sketch 所提供的了 Action API 可以使插件对应用程序中的事件做出反应，有点类似 Android 开发中的的 BroadCast 或者 Job Scheduler。官方文档列举了数百个可供监听的 Action，但最常用到的只有下面几个：

Action	触发条件	应用举例
Open/CloseDocument	Document 被打开/关闭	处理文档Layer
Startup	插件安装与启动、Sketch 启动	读取用户配置、下载数据以及插件自启动
Shutdown	插件被禁用或卸载、Sketch 关闭	清理插件数据、存储在Sessions间需数据
SelectionChanged	当文档中用户选择的Layers发生改变时触发	通过Action Context拿到Document、oldSelection以及newSelection

监听回调

我们只需在插件的 manifest.json 文件中添加一个 handler 即可。比如下面的例子添加了对于“OpenDocument”的监听，也就是告诉插件在新文档被打开时要去执行 onOpenDocument 这个 function。

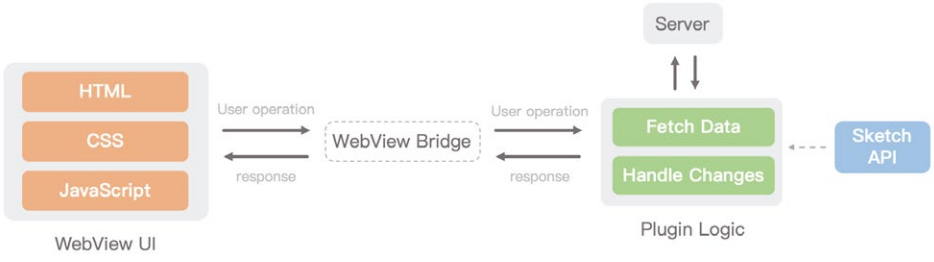
```
{
      "script": "action.js",
      "identifier": "my-action-listener-identifier",
      "handlers": {
        "actions": {
          "OpenDocument": "onOpenDocument"
        }
      }
}
```

当一个 Action 被触发时，会回调 JS 中的监听方法，与此同时 Sketch 可以向目标函数发送 Action Context，其中包含动作本身的一些信息。在下面例子中，每次打开文档时都会弹出一个 Toast。

```
function onOpenDocument(context) {
    context.actionContext.document.showMessage('Document Opened')
}
```

6. Bridge 双向通信

在常规的插件开发中，UI 层一般采用 Webview 实现，因此你可以使用各种前端开发框架，比如 React 或者 Vue 等；而插件的逻辑层（负责调用 Skecth API）显然不在 WebView 中，因此需要通过 Bridge 进行通信。逻辑层将从服务器获取到的数据传递给 UI 层展示，而 UI 层则将用户的操作反馈传递给逻辑层，使其调用 Sketch API 更新 Layers。



Sketch 通信原理

插件发送消息到 WebView

```
//On the plugin:
browserWindow.webContents
  .executeJavaScript('someGlobalFunctionDefinedInTheWebview("hello")')
  .then(res => {
    // do something with the result
  })

//On the WebView:
window.someGlobalFunctionDefinedInTheWebview = function(arg) {
  console.log(arg)
}
```

WebView 发送消息给插件

```
//On the webview:
window.postMessage('nativeLog', 'Called from the webview')
//On the plugin:
var sketch = require('sketch')
browserWindow.webContents.on('nativeLog', function(s) {
  sketch.UI.message(s)
})
```

经过了以上步骤，我们就得到了一个基础插件，它以 WebView 作为内容载体，并具有双向通信功能。打开插件时，Webview 会将页面加载完成的事件传递给逻辑层，逻辑层调用 Sketch API 弹出 Toast；点击 Get a random number 可以从逻辑层获取一个随机数。

skpm/with-webview 运行效果

快来正式加入开发队伍

相信阅读完上面的部分，制作一个简单的插件对于你来说，已经有点“游刃有余”了。但这个时候，疑惑也随之而来，为什么 Demo 和我们常用插件的 UI 差别如此之大?

没错，官方文档只教给我们最基础的插件开发流程，一个成熟的商业项目绝不仅仅是以上这些。一个功能完善的插件应该包括以下三部分：工具栏、WebView 容器以及业务数据。下面，我们会一步步为你展示如何开发一个商业化插件 UI，同时也会演示美团外卖“填充功能”的实现（注：篇幅原因文档中仅保留关键代码。）

常规 Sketch 插件结构

1. 创建吸附工具栏

所谓吸附式工具栏，就是展示在 Skecth 右侧 Inspector Panel 旁边的工具栏，它以吸附的方式与 Sketch 操作界面融为一体，这也是绝大多数插件的视觉呈现方式。工具栏中展示了当前插件可以提供的大部分功能，方便我们在操作 Document 时快速选取使用。

开发工具栏主要使用 NSStackView、NSButton、NSImage 以及 NSFont 这几个

类，如果没有开发过 macOS 应用的同学可能对这些类有些陌生，可以类比 iOS 开发中以 UI 作为前缀的控件类，NS 前缀主要是 AppKit 以及 Foundation 的相关类，MS 前缀则是 Skecth 的相关类，CA、CF 前缀为核心动画库和核心基础类。

下面的代码记录了创建工具栏的关键步骤，更为详细的操作可以参考一些 Github 仓库，比如 sketch-plugin-boilerplate 等。

```
const contentView = context.document.documentWindow().contentView();
const stageView = contentView.subviews().objectAtIndex(0);

//1. 创建 toolbar
const toolbar = NSStackView.alloc().initWithFrame(NSMakeRect(0, 0, 27,
420));
toolbar.setBackgroundColor(NSColor.windowBackgroundColor());
toolbar.orientation = 1;

//2. 创建 Button
const button =  NSButton.alloc().initWithFrame(rect)
const Image = NSImage.alloc().initWithContentsOfURL(imageURL)
button.setImage(image)
button.setTitle(" 数据填充 ")
button.setFont(NSFont.fontWithName_size('Arial',11))

//3. 将 Button 加入 toolbar
toolbar.addView_inGravity(button, gravityType);

//4. 将 toolbar 加入 SketchWindow
const views = stageView.subviews()
const finalViews = []
for (let i = 0; i < views.count(); i++) {
 finalViews.push(view)
 if(view[i].identifier() === 'view_canvas'){
   finalViews.push(toolbar)
}
 stageView.subviews = finalViews
 stageView.adjustSubviews()
```

2. 创建 WebView 容器

除了通过 CocoaScript 创建原生 NSPanel 外，这里推荐使用官方的 sketch-module-web-view 快速创建 WebView 容器，它提供了丰富的 API 对窗口的展示样式和行为进行定制，包括 Frameless Window、Drag 等，同时还封装了

WebView 与插件层的通信的 Bridge，使你可以轻松在”frontend”（the WebView）和”backend”（the plugin running in Sketch）之间发送消息。

```
//(1) 方法一：原生方式加入 webview
const panel = NSPanel.alloc().init();
panel.setFrame_display(NSMakeRect(0, 0, panelWidth, panelHeight), true);
const wkwebviewConfig = WKWebViewConfiguration.alloc().init()
const webView = WKWebView.alloc().initWithFrame_configuration(
  CGRectMake(0, 0, panelWidth, panelWidth),
  wkwebviewConfig
)
panel.contentView().addSubview(webView);
webview.loadFileURL_allowingReadAccessToURL(
  NSURL.URLWithString(url),
  NSURL.URLWithString('file:///')
)
//(2) 方法二：使用官方的 BrowserWindow
import BrowserWindow from "sketch-module-web-view";
const browserWindow = new BrowserWindow(options);
const webViewContents = browserWindow.webContents;

 webViewContents
    .executeJavaScript(`someGlobalFunctionDefinedInTheWebview(${JSON.
stringify(someObject)})`)
    .then(res => {
      // do something with the result
    })
 browserWindow.loadURL(require('./webview.html'))
```

3. 创建内容页面

历尽千辛万苦，我们终于拿到了 WebView，这下就可以发挥你“天马行空”的想象力了。不管是 React 还是 Vue，亦或只是一些简单的静态页面对于你而言应该都不在话下。在完成界面开发后，只需通过 Window 向插件发送指令即可。下面的例子演示了积木插件的“数据填充”功能。

UI 侧

```
import React from 'react';
import ReactDOM from 'react-dom';

// 使用 react 搭建用户页面
ReactDOM.render(<Provider store={store}><App /></Provider>, document.
getElementById('root'));
```

```
// 传递用户点击填充类目给插件层，这里以填充文字为例
export const PostMessage = (name, fillData) => {
  try {
    window.postMessage("fill-text-layer", fillData);
  } catch (e) {
    console.error(name, " 出现异常！！！ " + fillData);
  }
};
```

插件侧

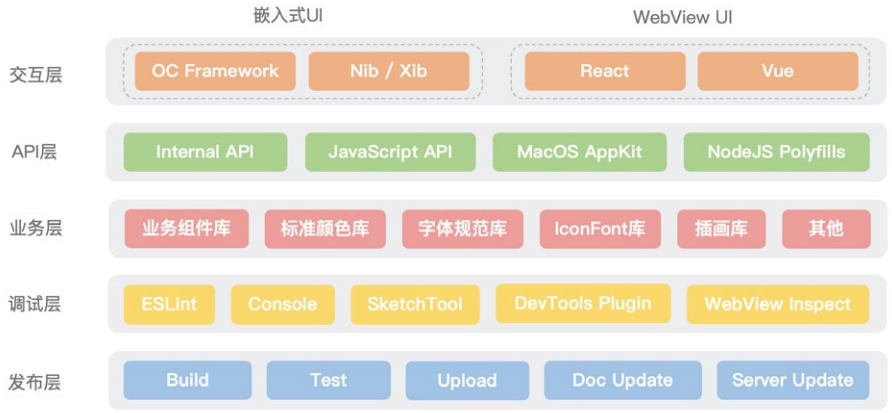
```
  browserWindow.webContents.on('fill-text-layer', function(s) {
   // 找到当前页面 document
  const document = context.document;
   // 获取用户选择的 layers
     const selection = Document.fromNative(document).selectedLayers;
        layers.forEach(item => {
          // 判断 layer 类型是否为文字
          if (item.type === 'Text') {
            // 更新 textlayer
            item.text = value;
          }
   });
})
```

4. 还想加点出彩的功能

如果你还不满足于此，说明你真的是个很爱学习，也很有潜力的开发同学。一个完善的插件需要包括交互层、API 层、业务层、调试层以及发布层，每层各司其职，它们都在默默干好自己的工作。

前面的步骤，通过构件菜单栏、创建 Webiew 完成了交互层的开发；通过 Webview 的 Bridge 传递用户操作到插件侧代码，之后调用 Sketch API 对图层进行操作，这是 API 层的工作；而根据自身需求并依托交互层与 API 层的实现去编写业务代码，则是业务层的工作；至此，你应该就拥有了一个可运行的插件了。

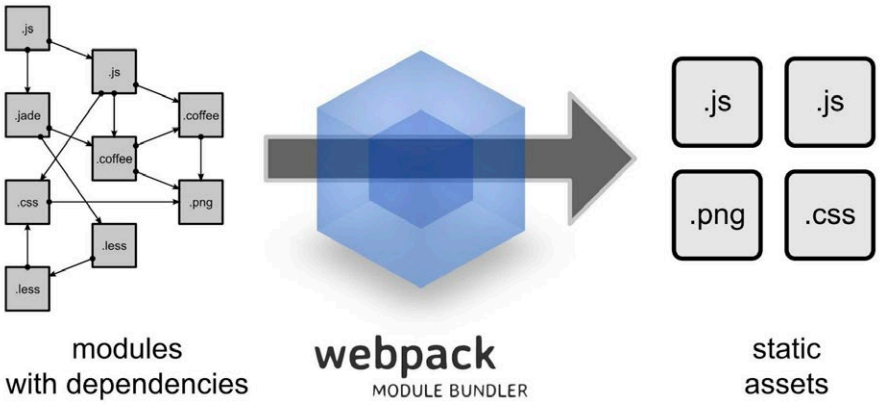
但除此之外，在代码编写过程中还需要 Lint 组件辅助开发，发现问题需要使用各类 Dev 工具进行调试，通过 QA 验证后，需要 Cli 工具打包并发布插件更新。这一小节，我们将简单介绍一些基本的调试层和发布层知识。



积木 Sketch Plugin 结构

Webpack 配置

Skpm 默认采用 Webpack 作为打包工具。Webpack 是一个现代 JavaScript 应用程序的静态模块打包器（Module Bundler）。当 Webpack 处理应用程序时，它会递归地构建一个依赖关系图（Dependency Graph），其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 Bundle，需要在 webpack.config.js 进行配置，类似于 Android 中的 Gradle，同样支持各种插件。



Webpack 处理流程示意

由于插件的开发者未必是前端同学，可能之前并没有接触过 Webpack，因此我们在这里介绍它的一些常用配置，让你有更多的时间关注业务代码。第一次接触 Webpack 是在去年一次公司内部的技术培训上（美团技术学院提供了很多技术培训课程，加入我们就可以尽情地在知识的海洋中遨游了），美团 MRN 项目的打包方案就是 Webpack。

在前端圈有各种各样的打包工具，比如 Webpack、Rollup、Gulp、Grunt 等等。RN 打包用的是 Facebok 实现的一套叫做 Metro 的工具，而美团 MRN 打包工具的选型是 Webpack，因为 Webpack 具有强大的插件机制和丰富的社区生态，可以完成复杂的流水线打包工作，Webpack 在 Plugin 开发中同样发挥了非常重要的作用。Webpack 有五个核心概念：

核心概念	释义	代码示例
入口文件	Entry Point指示Webpack 应该使用哪个模块作为构建其内部依赖图的开始。	module.exports = { entry: './path/to/my/entry/file.js' };
出口文件	output属性告诉Webpack 在哪里输出它所创建的bundles，以及如何命名这些文件。	output: { path: path.resolve(__dirname, 'dist'), filename: 'webpack.bundle.js' };
Loader	处理非 JavaScript 文件。	image-loader：图片格式压缩转换、重命名
插件	插件用于执行范围更广的任务，包括从打包优化和压缩，一直到重新定义环境中的变量。	CopyWebpackPlugin： 将单个文件或整个目录复制到构建目录
模式	Development或Production	module exports = { mode: 'production' };

在插件开发中需要处理 html、css、sass、jpg、style 等各种文件，只有在 Webpack 中配置相应的 Loader 后，这些文件才能被处理。而且我们很可能遇到某些文件需要使用特定的插件，而其它文件又无需处理的情况。下面的示例中列举了添加插件、对文件单独处理以及参数配置这三个常用的基本操作。

```
module.exports = function (config, entry) {
  // 常用功能 1：增加插件
    config.module.rules.push({
    test: /\.(svg)([\?]?.*)$/,
```

```
      use: [
        {
          loader: "file-loader",
          options: {
            outputPath: url => path.join(WEBPACK_DIRECTORY, url),
            publicPath: url => {return url;}}}
      ]
    });}

// 常用功能 2：对文件单独处理
if (entry.script === "src/script.js") {
    config.plugins.push(
      new htmlWebpackPlugin({ })
    );
}

// 常用功能 3：定制 js 处理
  config.module.rules.push({
    test: /\.jsx?$/,
    use: [
      { loader: "babel-loader",
        options: {
          presets: [
            "@babel/preset-react",
            "@babel/preset-env"
          ],
          plugins: [
            // 引入 antd 组件库
            ["import",{libraryName: "antd",libraryDirectory:
"es",style: "css"}]
      ]}}]
  });
```

ESLint 配置

JavaScript 是一门非常灵活的语言，很多错误往往运行时才爆出，通过配置前端代码检查方案，在编写代码过程中可直接得到错误反馈，也可以进行代码风格检查，不仅提升了开发效率，同时对不良代码编写习惯也能起到纠正作用。在 ESLint 中需要配置基础语法规则、React 规则、JSX 规则等，由于 Sketch 插件的 CocoaScript 语法较为特殊，需要配置全局变量以此忽略 AppKit 中无法识别的类。

虽然，我们曾在部门组会中被多次“安利”ESLint 的强大作用（这里给大家推荐一篇技术文章：ESLint 在中大型团队的应用实践），但如果不是做前端或者 RN 开发的

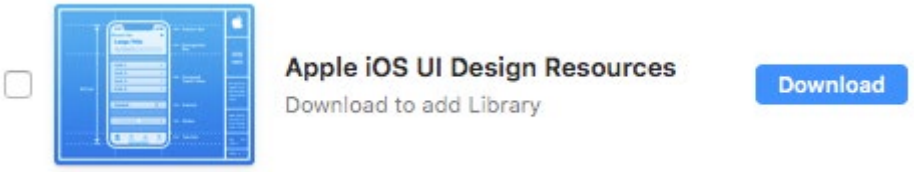
同学，可能对于 ESLint 的复杂配置并不熟悉。可以直接使用 Skpm 提供的 ESlint Config，里面配置了包含 Sketch 和 macOS 的头文件的全局变量，而代码格式化则推荐使用 Prettier。

```
npm install --save-dev eslint-config-sketch
// 或者直接使用带 prettier 以 eslint 的 skpm template 工程
$ skpm create my-plugin --template=skpm/with-prettier
```

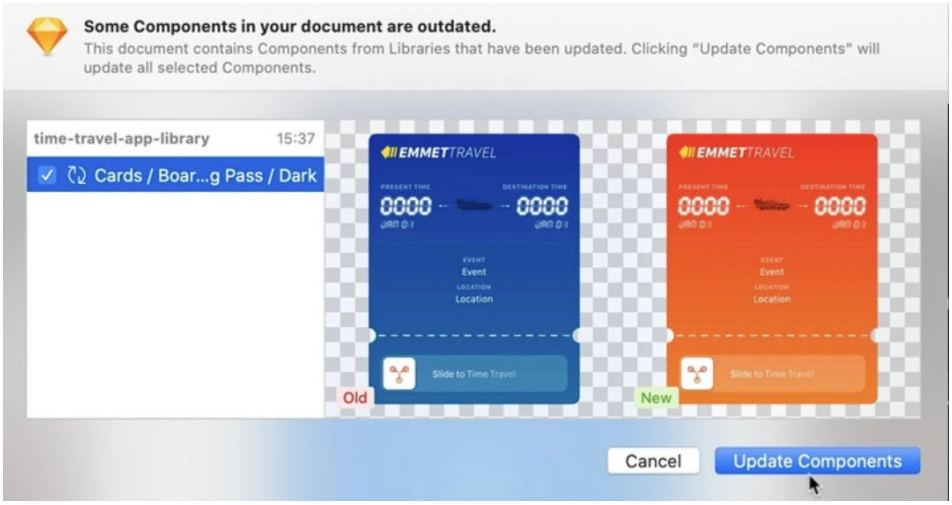
内容服务端化

Sketch 推出的库（Library）功能对于维护设计系统或风格指南，起到非常重要的作用，可以给团队带来高效工作体验，甚至改变设计团队工作方式和流程。我们通过组件库可以在整个设计团队中共享组件（Symbol），Library 可以实现“一处更改，处处生效”，即使是关联了远程组件库历史的设计稿检测到更新时，也会收到 Sketch 通知，确保工作中使用的是最新组件。

库功能对美团外卖 UI 一致性起着至关重要的作用，这主要体现在两方面：首先是实现设计风格沉淀，目前袋鼠 UI 已经形成了自己的独特风格，外卖设计团队根据设计规范，对符合 UI 一致性外卖业务场景的组件不断进行抽象及建设，沉淀出越来越多的通用业务组件，这些组件需要及时扩充到 Library 中，供团队成员使用；另外一个作用，则是保持团队使用的均为最新组件，由于各种原因，组件的设计元素（色彩、字体、圆角等属性）可能会发生变更，需要及时提醒团队成员更新组件，保持所有页面的一致性。



Sketch 内置的 iOS 远程组件库



Library 中的 Symbol 提示更新

库组件自动更新，其实就是“库列表”－“库 ID”－“外部组件原始 ID”这三者的关联。Sketch 内部是靠 UUID 进行对象识别的，通过库组件的库 ID，从库面板的列表中，按照添加的时间从新到旧依次检索所有未被禁用的、链接完好的库，直到匹配到库的 ID ，然后查找该库文件内是否有与库组件 SymbolID 匹配的组件，如果包含且内容有差异就提醒更新，更新的过程实际上是内容替换。

我们通过以下步骤使用 RSS 技术共享 Library 供整个 UI 设计团队使用：

- 将 Library Document 托管到公司内网服务器上。
- 创建一个 XML 文件记录版本信息和更新地址。
- 最后使用 Meyerweb URL 编码器之类的工具（或直接 encodeURIComponent）对 XML feed URL 进行编码并将其添加到以下内容：sketch://add-library?url=https://***.xml。
- 将此 URI 在浏览器中打开即可。

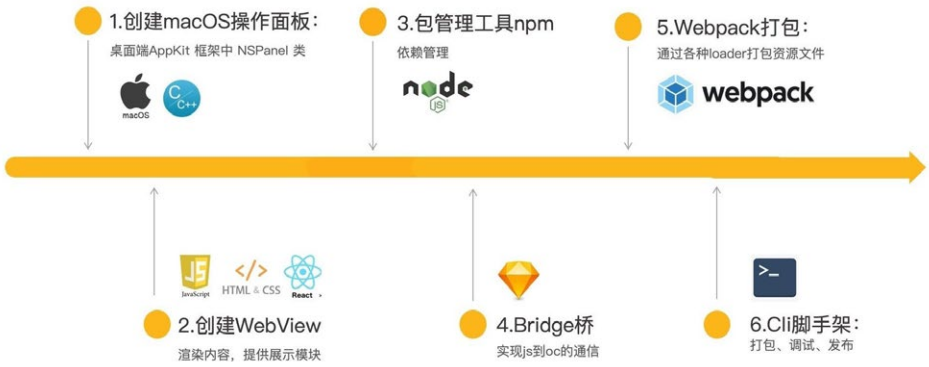
```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:content="http://purl.org/rss/1.0/modules/content/"
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:sparkle="http://
www.andymatuschak.org/xml-namespaces/sparkle">
```

```
  <channel>
    <title>My Sketch Library</title>
    <description>My Sketch Library</description>
    <image>
      <url></url>
    </image>
    <item>
      <title>My Sketch Library</title>
      <pubDate>Wed, 23 Jun 2019 11:19:04 +0000</pubDate>
      <enclosure url="mysketchlibrary.sketch" type="application/
octet-stream" sparkle:version="1"/>
    </item>
  </channel>
</rss>
```

5. 开发流程小结

前面一口气讲述了很多内容，可能你一时无法消化，这里对插件的开发流程作个简要的总结：

- 首先利用 JavaScript 或 CocoaScript 开发操作面板。
- 使用 NPM 安装所需依赖。
- 通过 Bridge 传递用户操作到插件逻辑侧，通过调用 Skecth API 对文档进行处理。
- 使用 Webpack 进行打包。
- 通过测试后发布插件更新。



Sketch Plugin 开发流程

别人可能没告诉你的事儿

这部分主要记录了积木 Sketch Plugin 开发过程中的踩坑经历，但是这里，我们没有贴大段的代码，没有直接告诉你答案，而是把分析问题的过程记录下来。“授人以鱼不如授人以渔”，相信只要你了解了这些分析技巧，即使之后遇到更多的问题，也可以轻（jia）松（ban）解决。

1. 与 Xcode 工程混合编译

首先，我们要明确一个问题，为什么要使用 XCode 工程?

虽然官方提供了 JS API 并承诺持续维护，但这项工作一直处于 Doing 状态，而且官方文档更新缓慢，没有明确的时间节点。因此，对于某些功能，比如我们想建一个具有 Native Inspector Panel 的插件，就不得不使用 XCode 进行开发。使用 Xcode 开发对于 iOS 开发者也更加友好，无需再学习前端界面开发知识。

这里推荐 Invison 的开发成员 James Tang 分享的博客文章《Sketch Plugin Xcode Template》，里面详细描述了构建插件 XCode 工程的步骤，这也成为很多插件开发者遵循的范本。当然随着 Sketch 的不断升级，某些 API 已经不受支持，但作者讲述的开发流程和思路依然没有改变，具有很高的学习价值。

```
JavaScript
// 利用 Mocha 加载 framework
var mocha = Mocha.sharedRuntime();
[mocha loadFrameworkWithName:frameworkName inDirectory:pluginRootPath]
```

除此之外，Skpm 中已经内置了 @skpm/xcodeproj-loader，也可在 JS 中直接加载 Framework。

```
JavaScript
// 加载 framework
const framework = require('../xcode-project-name/project-name.
xcodeproj/project.pbxproj');
const nativeClass = framework.getClass('NativeClassName');
// 获取 nib 文件
const ui = framework.getNib('NativeNibFile');
// 也可以直接加载 xib 文件
```

```
const NibUI = require('../xcode-project-name/view-name.xib')
var nib = NibUI()
let dialog = NSAlert.alloc().init()
dialog.setAccessoryView(nib.getRoot())
dialog.runModal()
```

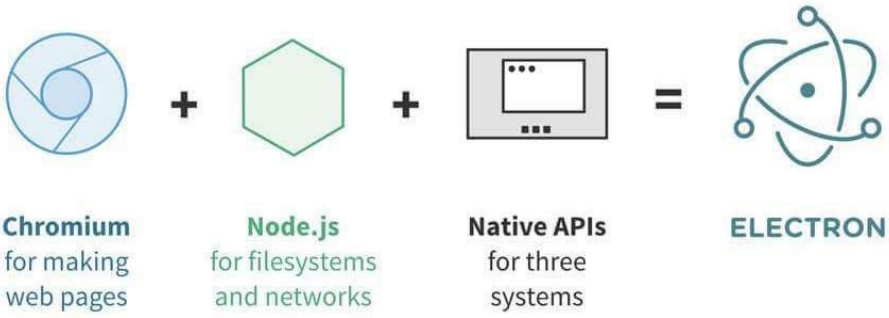
当然你也可以直接使用 Github 上一些知名的开源项目，有些会直接提供 Framework 供你使用，比如更改原生的 toolbar：

2. 了解 Electron

为什么在讲述 Sketch Plugin 的时候，忽然会提到 Electron？这里有一个小故事，某天上班打开大象（美团内部沟通软件）。

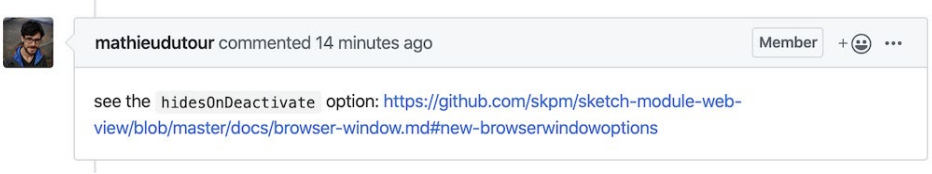
MacOS 版大象截图

看到一条公众号推送，是公司成立了 Electron 技术俱乐部（美团技术团队内部自发成立了很多技术俱乐部），经过了解发现 Electron 基于 Chromium 和 Node.js，可以使用 HTML、CSS 和 JavaScript 构建桌面应用程序，Electron 负责其中比较复杂的部分，而开发者只需关心应用的核心需求即可。大象的 Mac 端就大量使用了 Electron 技术，用 Web 框架去开发桌面应用，可以直接复用 Web 现有的开发成果并获得出色的运行效率。



我们就进行了简单的学习，在之后的一段时间并没有再去关注这项技术，直到某天在插件开发的过程中忽然遇到一个问题：在插件 WebView 显示的情况下，在桌面空白处点击使 Sketch 软件失去焦点，整个 App 就会被隐藏。试了几个流行的插件，发现大部分均有此问题，这给设计师的工作造成了诸多不便。试想，我只是去打开 Finder 找一个文件，你为什么要把我的软件最小化？在 Github 上留言后，很快得到了项目开发者 Mathieu Dutour 的官方回复，原来只需要设置一个 hidesOnDeactivate 属性即可。

等等！这不是 Electron 中的属性么？仔细查看 Readme 才发现作者写道“The API is mimicking the BrowserWindow API of Electron.”这下可方便多了！你想自定义窗口的表现，只需按照 Electron 的 API 设置即可，想想看其实 Electron 的工作方式是不是和 Sketch Plugin 如出一辙？



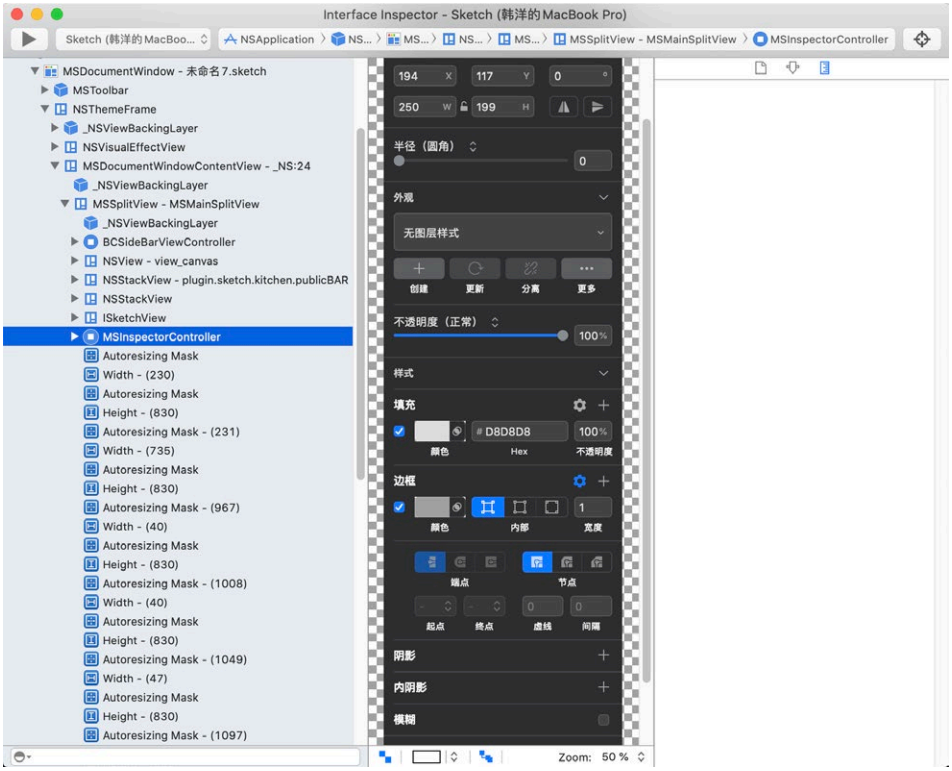
3. 更新原生属性面板

为了更好地提升积木 Sketch Plugin 的使用体验，UI 同学通过建立公共 Wiki 记录我们设计团队在插件使用过程中的反馈建议，其中有一条很奇怪：“通过插件面板更新 Layer 属性后，右侧面板不刷新。”和上一个问题一样，经测试其它插件大部分也有

此问题，但是如何去更新右侧属性面板呢？翻阅了 Sketch 的 API 文档还是“丈二和尚，摸不着头脑”。这个时候想起了 macOS 开发的一个神器 Interface Inspector，它可以在运行时分析正在运行的 Mac 应用程序的界面结构和属性，非常强大。

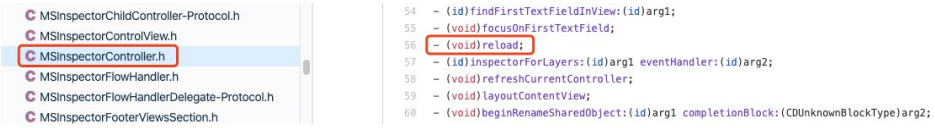
开心的下载下来后，发现这个软件上次的更新时间是 6 年前，忽然有了一种不祥的预感。果然 Attach 任何 App 时都会提示无法 Attach，在 macOS Catalina 版本已经无法运行。可是这怎么能难倒“万能”的程序员呢？我们查看系统报错，发现是 mach_inject_bundle_stub 错误，查阅发现 mach_inject_bundle_stub 是 Github 上的一个开源库，所以自己下载源码重新编译个 Bundle 包就可以了。

Attach 成功后，就可以对 Sketch 的面板进行属性分析了，是不是忽然感觉打开了新世界的大门？经过查阅发现右侧面板在 MSInspectorController 中。如下图所示：



Interface Inspector 对 Sketch 进行运行时分析

下一步需要用 Class-Dump 工具来提取 Sketch 的头文件，查看可以对 inspector 面板进行操作的所有方法：



通过 class-dump 得到的头文件

不出所料，我们发现了 reload()，猜测调用这个方法可以刷新面板，测试一下发现问题被修复了。如果你使用 Sketch 的 JavaScript API 的话，名称不一定能完全对应，但是基本差不多，稍加分析也可以找到。这里只是教大家一个思路，这样即使遇到其它问题，按照上面的步骤试试看，没准就可以解决。

```
JavaScript
// reload the inspector to see the changes
var sketch = require('sketch')
var document = sketch.getSelectedDocument()
document.sketchObject.inspectorController().reload()
```

未来等你加入

如你所见，积木 Sketch Plugin 可以帮助设计团队提升设计效率、沉淀设计语言以及减少走查负担；让 RD 同学面对新项目时，可以专注于业务需求而无需把时间耗费在组件的编写上；减少 QA 工作量，保证控件质量无需频繁回归测试；帮助 PM 提高版本迭代效率及版本需求吞吐量，提供业务的快速拓展能力。

当然，我们除了希望制作一流的产品，也希望积木插件可以让你在繁忙的工作中得以喘息。我们会继续以设计语言为依托，以 Skecth Plugin 为抓手持续进行 UI 一致性建设，提高客户端 UI 业务中台能力。

可能对于一个前端工程师来说，对 React、Webpack 等配置可以信手拈来；对于一个 iOS 工程师来说，XCode 调试、Objective-C 语法是开发前的基础；对于一个桌面工程师来说，对 Electron、Hook 分析已司空见惯。可 Sketch Plugin 开发就是这

么有趣，虽然只是一个小小的插件，但它会让你接触各个端的技术，提升技术视野，但同样会让你在开发过程中遇到很多困难，曾经困扰了我好几天的一个 Webpack 问题，部门同事帮我们联系了一个开发经验丰富的前端妹子去咨询，对方一行代码竟然就解决了。做你害怕做的事，然后你会发现，不过如此。

目前，积木插件开发还处于较为初级的阶段，包括 Mach（外卖自研动态化框架）实时预览、模板代码自动生成、自建插画库等功能已经在路上。除此之外，我们还规划了很多激动人心的功能，需要制作更多精美的前端页面，需要更完善的后台管理。

这里加个广告吧！不管你是 FE、Android、iOS、后端，只要你对 Bug 毫不手软，精益求精，都欢迎你加入我们外卖技术团队，跟我们一起完善 Sketch 插件生态，让积木插件可以为更多业务场景提供服务，为用户提供卓越的体验。让我们一起用“积木”拼出万千世界！

嗯，就先写到这里吧！ UI 团队同学说我们的实现和设计稿竟然差了一个像素，我们要回去改 Bug 了。

致谢

特别感谢优秀的设计师昱翰、沛东、淼林、雪美，他们在插件开发过程中给予的帮助。
特别感谢技术团队的云鹏、晓飞在技术上给予的指导。
“前人栽树，后人乘凉。”我们向优秀开源项目开发者致敬。

参考文献

Sketch Plugin 开发官方文档
深入理解 Sketch 库
凹凸实验室高大师 Sketch 插件开发实践
Sketch Plugin Xcode Template
Beginning Sketch Plugins Development in Xcode
携程机票 Sketch 插件开发实践

招聘信息

美团外卖长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家，欢迎加入外卖 App 大家庭。欢迎感兴趣的同学发送简历至：tech@meituan.com（邮件标题注明：美团外卖技术团队）

Native 地图与 Web 融合技术的应用与实践

作者：加鹏　张斌　杨睿　邱博　海峰

1. 背景

美团打车业务很早就在美团 App 与点评 App 中提供了服务入口，并在技术上采用了 H5 与 Native 的混合开发技术。随着业务上线，有用户反馈我们的地图性能有一些问题，原因是我们打车地图使用的是 Web 版的地图（通过腾讯地图 JavaScript API），业内同类产品使用的是 Native 版的地图 SDK，Native 地图相比 Web 地图具有天然的性能优势，所以美团打车地图从首屏地图加载到后续的地图操作体验都有一定差距。

问题和挑战

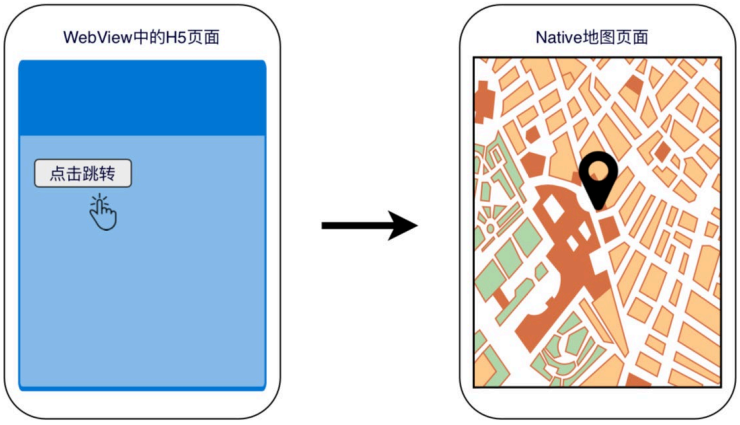
为了改善打车业务的地图体验，我们想到的方案是在展示地图的部分使用 Native 地图，而非地图部分使用 H5 页面来显示，这样既能追平与竞品的地图性能差距，又能充分发挥 H5 的开发效率。这种方案乍一看似乎是传统的 Hybrid 开发，没什么难度与新奇。比如地图使用预先内置到 App 中的地图 SDK 实现，H5 与 Native 的交互使用业界成熟的 JSBridge 技术。但从打车业务角度来看，因为打车业务有很多功能入口需要漂浮在地图之上，如起终点卡片、用户中心入口等，这种漂浮功能在技术上并不容易实现，而且还要保证用户触摸动作在漂浮元素与地图上发生时，分别派发给各自的事件系统，Hybrid 技术在这方面没有经验可以借鉴。

带着这些挑战，我们进行一系列的尝试与试验，最终将问题解决并封装出我们打车业务的地图调用框架，我们称之为 Native 地图与 Web 融合框架（下文简称融合框架）。在这个过程中，我们总结出了一些经验，希望能给从事相关研究的同学带来一些帮助。

2. 调研

基于混合技术开发体系，我们研究了市面上大部分 H5 页面与 Native 地图的应用场景，主要分为如下两类：

- H5 页面与 Native 地图分别是 2 个独立的页面：H5 业务逻辑用到地图时候，通过交互技术打开一个新地图页面，在新页面内，Native 地图按照传入参数调用对应地图组件，完成业务功能的展示。



H5 页面与 Native 地图位于同一页面内：两者将屏幕分割为两部分，如下图所示：Native 地图位于上半部分，WebView H5 页面位于下半部分。

经过分析后，我们发现这两种形式都无法满足打车业务场景的需求，因为目前市面上主流的打车业务场景由 4 部分构成，如下图所示：

- **起终点选择面板**：占据页面下半部分，可以上下滑动露出更多内容。
- **地图部分**：页面上半部分，显示起终点、线路等地图要素信息。
- **更多菜单**：左上角图标，点击后跳转到 H5 功能菜单页面。
- **广告入口**：右上角图标，点击后跳转到 H5 运营页面。

上文第一类，H5 页面与 Native 地图分别位于两个独立页面中，只能满足部分地图场景的需求，无法布局为上图 H5 与地图同框显示的效果。

上文第二类，实现这样的布局需要多个 WebView 才能实现，存在如下缺点：

- 下方 WebView 与上方 Native 地图是平级的组件，各占屏幕的一半，相互间不存在压盖关系，实现起终点面板上下滑动效果困难。
- 左上角、右上角的更多菜单，广告入口位置需要新增 2 个 WebView 组件才能实现覆盖在地图之上，WebView 组件再加载对应 H5 页面实现上述布局，整个步骤比较繁琐。

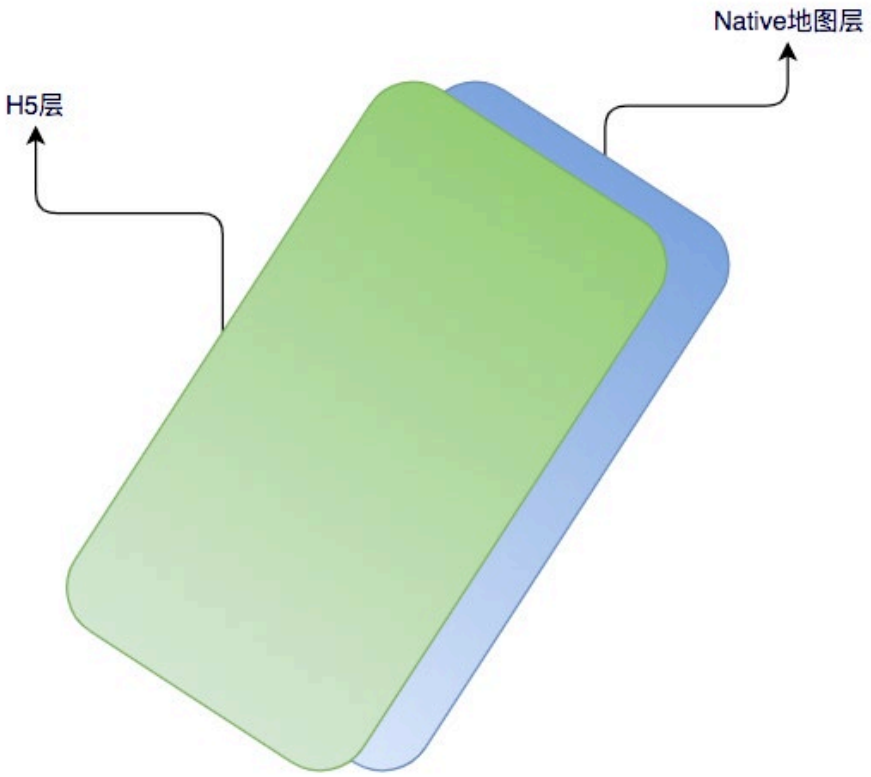
多个 WebView 组件构成的页面布局，由于内存空间不共享，它们之间信息的同步比

较困难，太多的 WebView 组件对系统性能也是一种浪费。

调研结论是：市面上现存技术都无法满足打车场景的需求。

全新方案的提出

基于打车场景的特殊性，我们做了一个大胆的假设：把页面分为 2 层，下层是 Native 地图层，布满屏幕；上层是 WebView 层，完全覆盖到 Native 地图层之上，如下图所示：



我们期望的效果是：

- 点击 H5 元素时，点击事件会派发给 H5 WebView 容器处理。
- 点击地图区域时，点击事件会派发给 Native 地图组件处理。

- H5 与 Native 地图间的信息交互，可采用成熟的 JSBridge 技术实现。

具体实现思路有如下几点，参照下图：

- Native 地图位于下层，WebView 置于 Native 地图之上，WebView 背景透明，透过 WebView 可以看到下边的地图。红框区域是上层 WebView 打开的 H5 页面元素。
- 增加一个手势消息分发层，该层会智能判断手势事件落在 H5 元素还是地图元素中。举例：点击红框区域，消息会传递到 WebView 层的 H5 逻辑处理，点击红框之外的区域，消息会传递到 Native 地图层处理（地图移动、缩放等操作）。
- H5 与 Native 地图交互使用 JSBridge 完成。比如在地图中添加一个 Marker，H5 层业务逻辑发出添加 Marker 的消息，H5 层通过 JSBridge 技术将消息发送到 Native 地图层，Native 地图收到消息后在地图中添加 Marker 元素。

为了验证想法是否正确，我们首先通过 Android 平台开发出 Demo，验证这种分层智能传递消息的做法是可行的，该方案最大优点是兼顾了 H5 的开发效率与 Native 地图的高性能特性，非常符合美团业务地图场景的需求。为了让想法落地时更规范、更系统，我们进行了如下的框架设计。

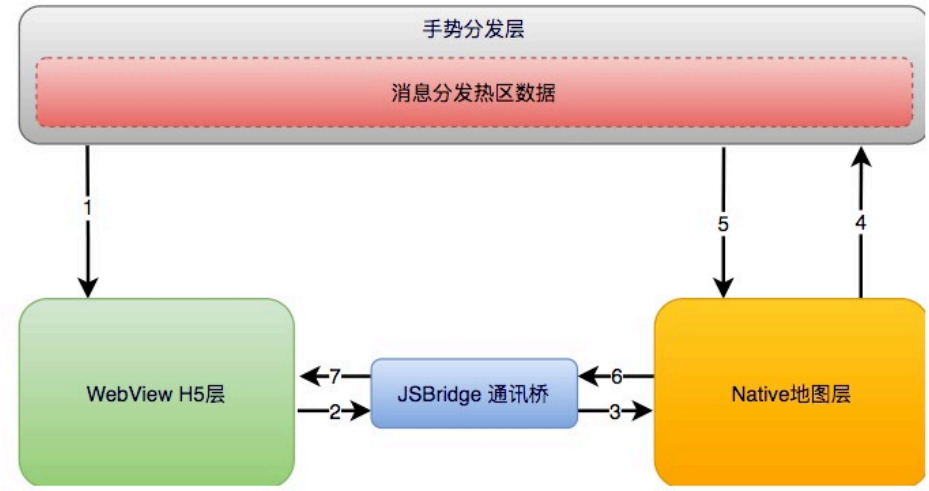
3. 框架设计

3.1 热区数据介绍

先介绍一个“热区数据”的概念，下图（3.2 节）在手势分发层存在着消息分发热区数据部分，下文简称热区数据。热区数据是针对上层 WebView 的一个概念，只对 WebView 层有效，对下层 Native 地图层无效。如果用户点击屏幕事件想让 H5 来捕获处理，可以在屏幕区域内设置一个逻辑上的矩形区域，如：[0, 0, 50, 50]（上图左上角区域），这个数据被称为**热区数据**。

我们通过编写代码逻辑，控制手势消息分发的策略，如果手势消息发生在热区数据矩形范围内，我们把消息发送给 WebView 处理，否则发送给 Native 地图处理。如上图所示，可以在同一屏幕内设定多个热区，[0, 0, 50, 50]、[430, 0, 50, 50]、[0, 200, 480, 200]，热区的格式可以自己定义，我们这里采用的基于 WebView 组件左上角为原点的像素坐标格式：[left, top, width, height]。

3.2 框架图介绍



手势消息分发给 WebView 层流程

主要为上图 1 - >2 - >3 - >4 过程，如下：

- 用户触摸动作首先被手势分发层捕获，手势分发层判断用户点击到热区数据范围内，将消息分发到 WebView H5 层处理。
- WebView H5 层收到消息，对消息进行处理（比如：在地图中添加一个终点 Marker），通过通讯桥将消息传递到 Native 地图层。
- Native 地图层收到消息，并执行添加 Marker 操作，完成后返回成功信息。上述总体流程为：手势分发层 - >1 - >2 - >3 - >6 - >7。

手势消息分发给 Native 地图层流程

主要为上图 5->6->7 过程，如下：

- 手势分发层捕获到消息，发现用户手势与当前热区数据矩形没有交集，于是将获取的消息分发到 Native 地图层。
- 如果消息是拖动操作，则 Native 地图自动识别拖动地图消息，实现移动地图的效果，涉及流程为：手势分发层 ->5。
- 如果消息是点击操作，比如我们想实现点击地图中的 Marker，将消息传递给 H5 处理的功能。实现步骤为我们事先在添加 Marker 时增加一个点击事件（Native 地图层实现），Marker 被点击时 Native 地图层会派发此事件，事件消息会通过 JSBridge 技术从 Native 地图层传到 H5 层，最后 H5 层获取到点击消息。整个操作流程为：手势分发层 ->5->6->7。

热区数据的动态更新策略

因为打车业务底部的面板高度是可伸缩的，所以底部的热区数据并不是静止不动的，需要考虑热区数据也要随着 DOM 元素的拉伸做同步调整。可以通过在 WebView H5 层监控 DOM 的变化，DOM 元素发生变化时，获取变化后的 DOM 元素位置、大小，格式化为热区数据，并更新到消息分发热区数据部分。因为拉伸动作是一个连续的动画效果，为了高效，我们只在动画结束的那一刻更新热区数据，中间过渡期不做处理。此整体流程为：2->3->4。

4. 点评 App 中的落地实践

4.1 手势分发层关键代码

这部分功能需要 Native 端同学实现，包括 iOS 与 Android。两端分别在启动 App 时设置三层内容，最上层是手势触摸事件接收层，中间是 WebView 层（背景设置透明），最下层是 Native 地图层（如腾讯地图 SDK）。用数组记录当前热区数据，当手势分发层有事件发生时，通过 Touch 事件获取手指位置信息，遍历热区数组判断手

指位置是否与热区的矩形相交，如相交则将消息分发给 WebView 层，否则分发给 Native 层。下边是 Android 与 iOS 消息分发关键代码：

Android 分发层关键代码

```
@Override
public boolean dispatchTouchEvent(MotionEvent event){
  if(event.getAction() == MotionEvent.ACTION_DOWN) {
    // 分发层接收到手势触摸消息，通过 dispatchService 类判断手势是否落在热区内，
从而确定消息分发的对象
    this.touchHandler = dispatchService.inRegion(event) ? TouchHandler.
WebView : TouchHandler.MapView;
  }
  // 分发给 Native 地图层
  if(this.touchHandler == TouchHandler.MapView) {
    return this.mapView.dispatchTouchEvent(event);
  }
  // 分发给 WebView H5 层
  return super.dispatchTouchEvent(event);
}
```

iOS 分发层关键代码

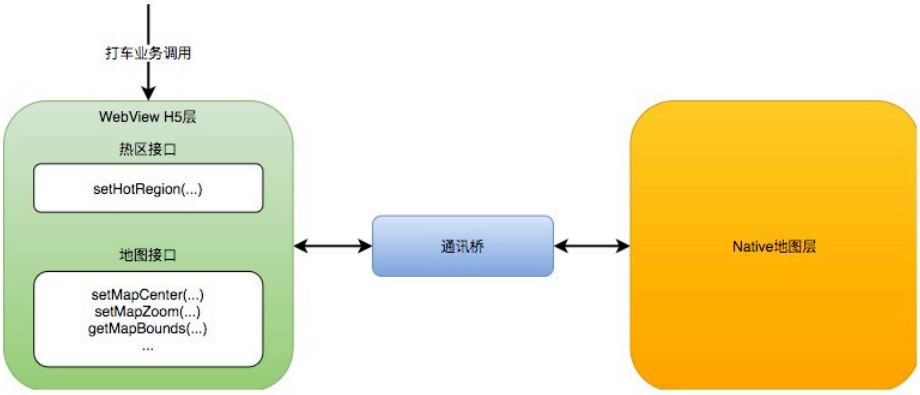
```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
   UIView *hitTestView = nil;
   // 分发层接收到手势触摸消息，通过 pointInHotspot 判断手势是否落在热区内，从而确定
消息分发的对象
   if ([self pointInHotspot:point]) {
     // 分发给 WebView H5 层
     hitTestView = [self.WebView hitTest:point withEvent:event];
   }else{
     // 分发给 Native 地图层
     hitTestView = [self.mapView hitTest:point withEvent:event];
   }
   return hitTestView;
}
```

4.2 WebView H5 层

该层有 2 个功能：

- 提供设置热区的 JS 接口 setHotRegion，业务可通过此接口设置屏幕中的热区。

- 封装一层 JS 形式的地图接口，为上层业务提供地图服务，该层借助 JS-Bridge 通讯桥实现 H5 与 Native 层的异步通讯。

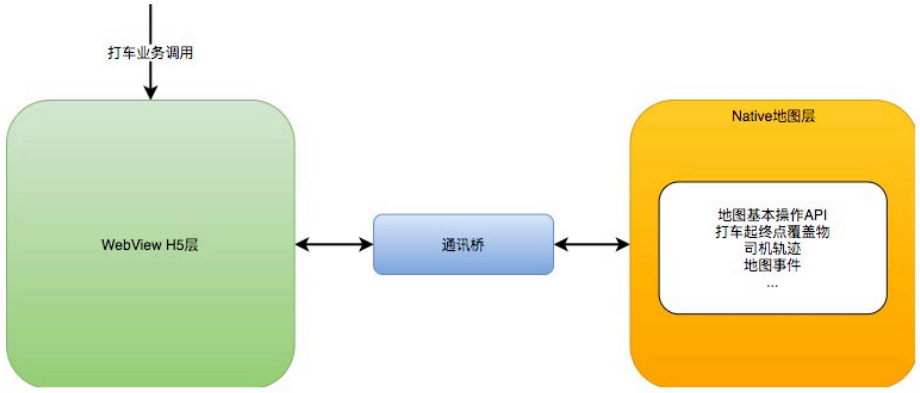


4.3 通讯桥简介

通讯桥即 JSBridge 技术，主要实现 H5 与 Native 的信息交互，这方面的技术都已比较成熟，业界有非常多的 JSBridge 实现，原理也都类似，常见的有：原生对象注入到 H5 层、URL 拦截技术，Native 调用 JS 常用的内置函数 stringByEvaluatingJavaScriptFromString 等。美团内部有比较成熟的 KNB 框架，所以项目中直接使用了 KNB 框架。

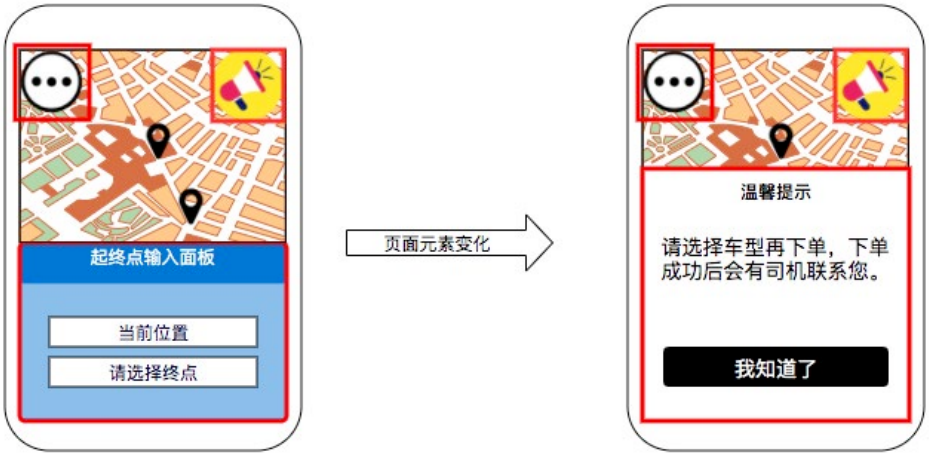
4.4 Native 地图层

该层在地图 SDK（如腾讯地图 SDK）基础上进行了封装，提供一些打车业务友好的接口，如地图基本操作、打车起终点 Marker 添加、接送驾司机小车动画、地图事件、各种 Marker 的信息弹窗等。



4.5 Dom 元素热区数据的自动维护技术

打车业务前端的技术栈是：Vue + VueX + Vue-Router 构建的单页系统。如下图所示，页面中存在很多 H5 元素需要添加热区，逐个元素编写代码添加的话会很繁琐，而且页面元素的位置、大小变化时还需要同步更新热区数据，这里我们使用了 Vue 中的 directive（指令）来解决了此问题。



以上左右 2 图是用户操作时页面展示的不同状态，很明显右图底部卡片变高了，卡片变化同时需要同步更新对应的热区数据，directive 技术可以很方便解决此问题，原理如下：

- 在添加元素时，Vue 指令的 bind 钩子函数被触发，此时计算出弹窗元素的大小和位置：使用 getBoundingClientRect 函数可以获取到元素的 left、top、width、height 等信息，将新的热区数据通过 setHotRegion 函数更新到手势分发层。
- 在移除元素时，unbind 钩子函数被触发，此时将热区数据移除，这样便实现了热区的自动添加删除功能了。
- 使用指令技术很简捷，编写好指令的逻辑后注册到全局，在需要动态更新热区的元素上设置个 v-hotRegion 标签就可以了。

4.6 调试工具及测试

调试工具使用模拟器、真机都可以，开发期间我们使用的模拟器开发，测试期间 QA 使用真机验证。调试过程中主要验证 2 部分功能，分别是热区的验证与地图接口验证。

热区验证

主要验证主页面设置的热区是否正确，包括是否可以点击、底部卡片是否能正常拖拉、业务功能是否正常等。因为热区数据是一串数字，形如：[0, 0, 50, 50]，无法直观判断出该数据是否有误，于是我们开发了一个可视化工具，将设置热区的元素都用红色矩形高亮显示，如下图所示，这样就能快速诊断出热区数据是否有异常。工具是使用 Canvas 画布实现的，画布大小与屏幕大小完全重合，借助画布就可以将矩形热区数据在屏幕中实时绘制出来。

地图接口验证

主要是编写单元测试完成的，本项目封装了 50 多个地图接口，每个接口都编写单测用例，观察入参、出参、控制台输出结果，地图展示效果是否正确等。测试主要在 iOS 模拟器中完成，这样方便在控制台打印一些调试信息进行诊断。

5. 上线效果

该框架在大众点评 App 中上线后地图体验明显提升，主要有体现在以下几个方面：

地图的操作体验，如地图移动、缩放明显好于 H5 地图，用户利用 Native 地图选择起终点、下单叫车、接送驾小车动画效果更加流畅。

首屏地图数据指标也有显著提升，如下表所示：

统计项	H5地图模式（单位：ms）	Native地图模式（单位：ms）	降低比率	备注
responseStart	2286	1400	-38%	从输入URL到首字节开始
domInteractive	2638	1680	-29%	domReady
loadEventEnd	2674	1706	-36%	从输入URL，首页资源加载完毕

- 目前线上运行稳定，上线 2 月期间，Crash 数量为个位数，Crash 率远低于 0.1‰。
- 框架上线后，大众点评 App 中业务迭代可以按照 H5 节奏上线，实现随时发版的开发效率。

Native 地图层代码接口稳定、功能丰富，基本满足地图场景的业务需求。只需首次跟版发布，后续只需要迭代 H5 的业务逻辑即可。

6. 本文小结

本文将 WebView 与 Native 地图组件叠加到一起，实现了用户手势事件智能分发的机制，解决了 WebView 与 Native 地图在同一页面内布局困难的问题。这种融合机制为打车业务提升迭代效率同时保障地图体验提供了一种有效的途径，日常业务功能上线采用 H5 技术迭代，Native 地图作为不常更新的基础能力首次发版时安装到用户手机上，实现业务需求随时发版的能力，不再受各大应用商店的限制，用户操作地图的体验也更加流畅。该融合框架适合以下业务场景：

业务中使用了地图功能，并对地图的加载、操作体验等有较高要求的业务。

业务属于 Hybrid 业务，并且 H5 页面与地图在同一页面内布局的功能。

如果你的业务是基于多个 WebView 与 Native 地图构建的系统，非常建议你了解下此文章。

7. 作者简介

美团打车技术部终端研发中心，加鹏、张斌、杨睿、邱博、海峰等。

8. 招聘信息

美团打车技术部终端研发中心诚招高级前端开发工程师、前端开发专家、高级 iOS 工程师、高级 Android 工程师。我们为美团点评用户提供优质的打车服务，是本地生活吃喝玩乐行的重要环节。欢迎各位小伙伴的加入，共同打造极致出行产品。感兴趣的同学可投递简历至：tech@meituan.com（邮件标题注明：美团打车技术部终端研发中心）


CODE A BETTER LIFE
一行代码 亿万生活
美团
技术团队
长按二维码关注我们