2019-2021美团技术年货

CODE A BETTER LIFE

【前端篇】



美团 美团

目录

前端 | 2021 1

让 Flutter 在鸿蒙系统上跑起来 1

FlutterWeb 在美团外卖的实践 14

FlutterWeb 性能优化探索与实践 36

设计稿（UI 视图）自动生成代码方案的探索 56

美团外卖终端容器无关化研发框架 74

一款可以让大型 iOS 工程编译速度提升 50% 的工具 96

从预编译的角度理解 Swift 与 Objective-C 及混编机制 108

美团民宿跨端复用框架设计与实践 165

美团跨端一体化富文本管理技术实践 186

前端 | 2020 202

移动端 UI 一致性解决方案 202

美团外卖 Flutter 动态化实践 227

美团开源 Logan Web：前端日志在 Web 端的实现 255

外卖客户端容器化架构的演进 270

Flutter 包大小治理上的探索与实践 297

美团外卖持续交付的前世今生 326

微前端在美团外卖的实践 352

积木 Sketch 插件进阶开发指南 372

积木 Sketch Plugin：设计同学的贴心搭档 400

Native 地图与 Web 融合技术的应用与实践 431

前端 | 2019 446
MTFlexbox 自动化埋点探索 447
Litho 在美团动态化方案 MTFlexbox 中的实践 459
开源 React Native 组件库 beeshell 2.0 发布 473
React Native 在美团外卖客户端的实践 506
Android 静态代码扫描效率优化与实践 530
Probe：Android 线上 OOM 问题定位组件 560
活动 Web 页面人机识别验证的探索与实践 580
React Native 工程中 TSLint 静态检查工具的探索之路 589
ESLint 在中大型团队的应用实践 610
美团 iOS 工程 zsource 命令背后的那些事儿 627
客户端单周发版下的多分支自动化管理与实践 635
美团外卖前端容器化演进实践 643
Bifrost 微前端框架及其在美团闪购中的实践 664
Litho 的使用及原理剖析 680
Android 兼容 Java 8 语法特性的原理分析 697
美团外卖商家端视频探索之旅 715

让 Flutter 在鸿蒙系统上跑起来

作者：杨超

前言

鸿蒙系统（HarmonyOS）是华为推出的一款面向未来、面向全场景的分布式操作系统。在传统单设备系统能力的基础上，鸿蒙提出了基于同一套系统能力、适配多种终端形态的分布式理念。自 2020 年 9 月 HarmonyOS 2.0 发布以来，华为加快了鸿蒙系统大规模落地的步伐，预计 2021 年底，鸿蒙系统会覆盖包括手机、平板、智能穿戴、智慧屏、车机在内数亿台终端设备。对移动应用而言，**新的系统理念、新的交互形式，也意味着新的机遇**。如果能够利用好鸿蒙的开发生态及其特性能力，可以让应用覆盖更多的交互场景和设备类型，从而带来新的增长点。

与面临的机遇相比，适配鸿蒙系统带来的挑战同样巨大。当前手机端，尽管鸿蒙系统仍然支持安卓 APK 安装及运行，但长期来看，华为势必会抛弃 AOSP，逐步发展出自己的生态，这意味着现有安卓应用在鸿蒙设备上将会逐渐变成“二等公民”。然而，如果在 iOS 及 Android 之外再重新开发和维护一套鸿蒙应用，在如今业界越来越注重开发迭代效率的环境下，所带来的开发成本也是难以估量的。因此，通过打造一套合适的跨端框架，以相对低的成本移植应用到鸿蒙平台，并利用好该系统的特性能力，就成为了一个非常重要的选项。

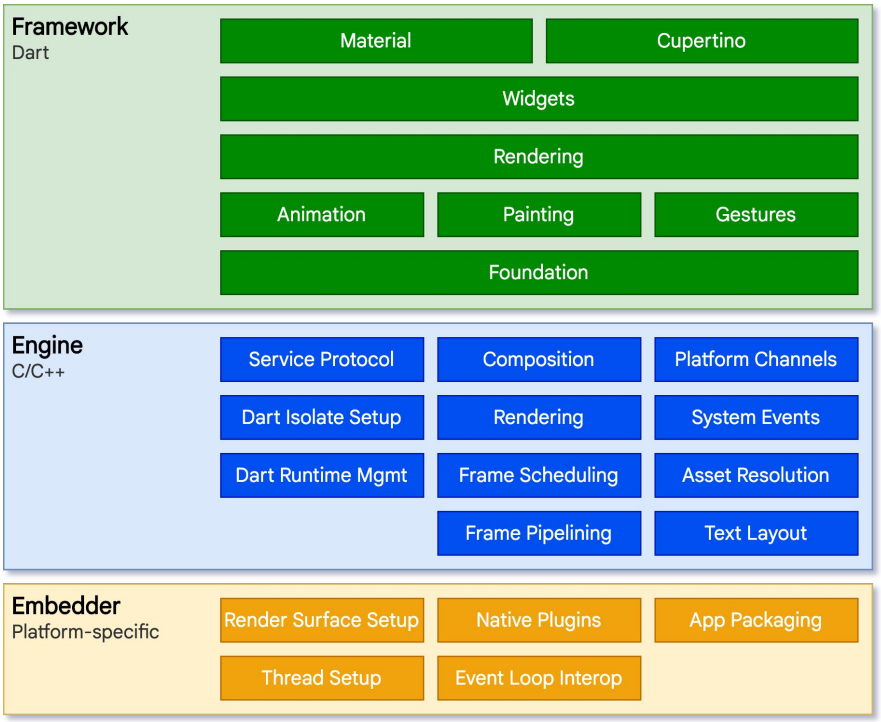
在现有的众多跨端框架当中，Flutter 以其自渲染能力带来的多端高度一致性，在新系统的适配上有着突出的优势。虽然 Flutter 官方并没有适配鸿蒙的计划，但经过一段时间的探索和实践，美团外卖 MTFlutter 团队成功实现了 Flutter 对于鸿蒙系统的

原生支持。

这里也要提前说明一下，因为鸿蒙系统目前还处于 Beta 版本，所以这套适配方案还没有在实际业务中上线，属于技术层面比较前期的探索。接下来本文会通过原理和部分实现细节的介绍，分享我们在移植和开发过程中的一些经验。希望能对大家有所启发或者帮助。

背景知识和基础概念介绍

在适配开始之前，我们要明确好先做哪些事情。先来回顾一下 Flutter 的三层结构：



在 Flutter 的架构设计中，最上层为**框架层**，使用 Dart 语言开发，面向 Flutter 业务的开发者；中间层为**引擎层**，使用 C/C++ 开发，实现了 Flutter 的渲染管线和 Dart 运行时等基础能力；最下层为**嵌入层**，负责与平台相关的能力实现。显然我们要做的是将嵌入层移植到鸿蒙上，确切地说，我们要**通过鸿蒙原生提供的平台能力，重新实**

现一遍 Flutter 嵌入层。

对于 Flutter 嵌入层的适配，Flutter 官方有一份不算详细的指南，实际操作起来成本很高。由于鸿蒙的业务开发语言仍然可用 Java，在很多基础概念上与 Android 也有相似之处（如下表所示），我们可以从 Android 的实现入手，完成对鸿蒙的移植。

HarmonyOS	Android	说明
Ability	Activity & Service & ContentProvider	FA（Feature Ability）– Activity PA（Particle Ability）– Service & ContentProvider
AbilitySlice	Fragment	应用的单个页面及其控制逻辑的总和
CommonEventSubscriber	BroadcastReceiver	鸿蒙公共事件服务（CES）的一部分，支持接收公共事件
AbilityPackage	Application	
Intent	Intent	对象之间传递信息的载体
Ability Form	Widget	类似Android的桌面小部件
NotificationHelper	NotificationManager	通知管理器
SystemPasteboard	ClipboardManager	系统剪贴板服务

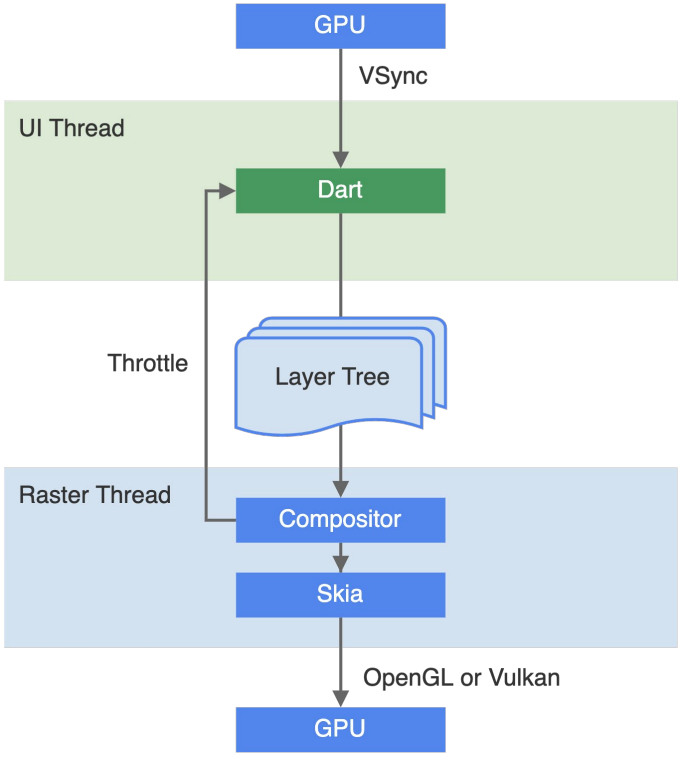
Flutter 在鸿蒙上的适配

如前文所述，要完成 Flutter 在新系统上的移植，我们需要完整实现 Flutter 嵌入层要求的所有子模块，而从能力支持角度，**渲染**、**交互**以及**其他必要的原生平台能力**是保证 Flutter 应用能够运行起来的最基本的要素，需要优先支持。接下来会依次进行介绍。

1. 渲染流程打通

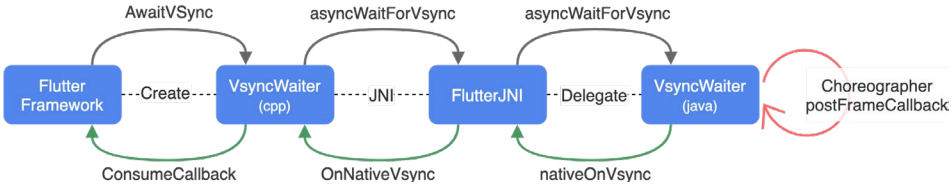
我们再来回顾一下 Flutter 的图像渲染流程。如图所示，设备发起**垂直同步**（VSync）信号之后，先经过 UI 线程的渲染管线（Animate/Build/Layout/Paint），再经过 Raster 线程的组合和栅格化，最终通过 OpenGL 或 Vulkan 将图像**上屏**。这个流程的大部分工作都由框架层和引擎层完成，对于鸿蒙的适配，我们主要关注的是与设备自身能力相关的问题，即：

(1) 如何监听设备的 VSync 信号并通知 Flutter 引擎? (2) OpenGL/Vulkan 用于上屏的窗口对象从何而来?



VSync 信号的监听及传递

在 Flutter 引擎的 Android 实现中，设备的 VSync 信号通过 Choreographer 触发，它产生及消费流程如下图所示：



Flutter VSync

Flutter 框架注册 VSync 回调之后，通过 C++ 侧的 VsyncWaiter 类等待 VSync

信号，后者通过 JNI 等一系列调用，最终 Java 侧的 VsyncWaiter 类调用 Android SDK 的 Choreographer.postFrameCallback 方法，再通过 JNI 一层层传回 Flutter 引擎消费掉此回调。Java 侧的 VsyncWaiter 核心代码如下：

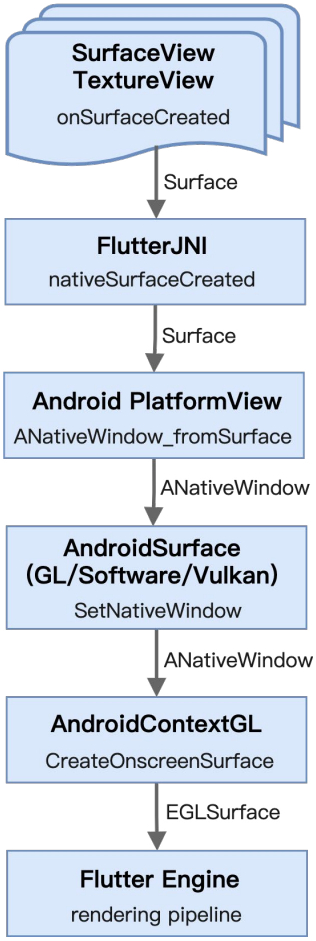
```
@Override
public void asyncWaitForVsync(long cookie) {
  Choreographer.getInstance()
      .postFrameCallback(
        new Choreographer.FrameCallback() {
          @Override
          public void doFrame(long frameTimeNanos) {
            float fps = windowManager.getDefaultDisplay().
getRefreshRate();
            long refreshPeriodNanos = (long) (1000000000.0 / fps);
            FlutterJNI.nativeOnVsync(
              frameTimeNanos, frameTimeNanos + refreshPeriodNanos,
cookie);
          }
        });
}
```

在整个流程中，除了来自 Android SDK 的 Choreographer 以外，大多数逻辑几乎都由 C++ 和 Java 的基础 SDK 实现，可以直接在鸿蒙上复用，问题是鸿蒙目前的 API 文档中尚没有开放类似 Choreographer 的能力。所以现阶段我们可以借用鸿蒙提供的类似 iOS Grand Central Dispatch 的线程 API，模拟出 VSync 的信号触发与回调：

```
@Override
public void asyncWaitForVsync(long cookie) {
  // 模拟每秒 60 帧的屏幕刷新间隔：向主线程发送一个异步任务，16ms 后调用
  applicationContext.getUITaskDispatcher().delayDispatch(() -> {
    float fps = 60; // 设备刷新帧率，HarmonyOS 未暴露获取帧率 API，先写死 60 帧
    long refreshPeriodNanos = (long) (1000000000.0 / fps);
    long frameTimeNanos = System.nanoTime();
    FlutterJNI.nativeOnVsync(frameTimeNanos, frameTimeNanos +
refreshPeriodNanos, cookie);
  }, 16);
};
```

渲染窗口的构建及传递

在这一部分，我们需要在鸿蒙系统上构建平台容器，为 Flutter 引擎的图形渲染提供用于上屏的窗口对象。同样，我们参考 Flutter for Android 的实现，看一下 Android 系统是怎么做的：



Flutter 在 Android 上支持 Vulkan 和 OpenGL 两种渲染引擎，篇幅原因我们只关注 OpenGL。抛开复杂的注册及调用细节，本质上整个流程主要做了三件事：

- 创建了一个**视图对象**，提供可用于直接绘制的 Surface，将它通过 JNI 传递给原生侧；

- 在原生侧获取 Surface 关联的**本地窗口对象**，并交给 Flutter 的平台容器；
- 将本地窗口对象转换为 OpenGL ES 可识别的**绘图表面（EGLSurface）**，用于 Flutter 引擎的渲染上屏。

接下来我们用鸿蒙提供的平台能力实现这三点。

a. 可用于直接绘制的视图对象

鸿蒙系统的 UI 框架提供了很多常用视图组件（Component），比如按钮、文字、图片、列表等，但我们需要抛开这些上层组件，获得直接绘制的能力。借助官方 媒体播放器开发指导 文档，可以发现鸿蒙提供了 SurfaceProvider 类，它管理的 Surface 对象可以用于视频解码后的展示。而 Flutter 渲染与视频上屏从原理上是类似的，因此我们可以借用 SurfaceProvider 实现 Surface 的管理和创建：

```
// 创建一个用于管理 Surface 的容器组件
SurfaceProvider surfaceProvider = new SurfaceProvider(context);
// 注册视图创建回调
surfaceProvider.getSurfaceOps().get().addCallback(surfaceCallback);

// ... 在 surfaceCallback 中
@Override
public void surfaceCreated(SurfaceOps surfaceOps) {
  Surface surface = surfaceOps.getSurface();
  // ...将 surface 通过 JNI 交给 Native 侧
  FlutterJNI.onSurfaceCreated(surface);
}
```

b. 与 Surface 关联的本地窗口对象

鸿蒙目前开放的 Native API 并不多，在官方文档中我们可以比较容易地找到 Native_layer API。根据文档的说明，Native API 中的 NativeLayer 对象刚好对应了 Java 侧的 Surface 类，借助 GetNativeLayer 方法，我们实现了两者之间的转化：

```
// platform_view_android_jni_impl.cc
static void SurfaceCreated(JNIEnv* env, jobject jcaller, jlong shell_
holder, jobject jsurface) {
  fml::jni::ScopedJavaLocalFrame scoped_local_reference_frame(env);
  // 通过鸿蒙 Native API 获取本地窗口对象 NativeLayer
  auto window = fml::MakeRefCounted<AndroidNativeWindow>(
```

```
      GetNativeLayer(env, jsurface));
  ANDROID_SHELL_HOLDER->GetPlatformView()->NotifyCreated(std::move(wind
ow));
}
```

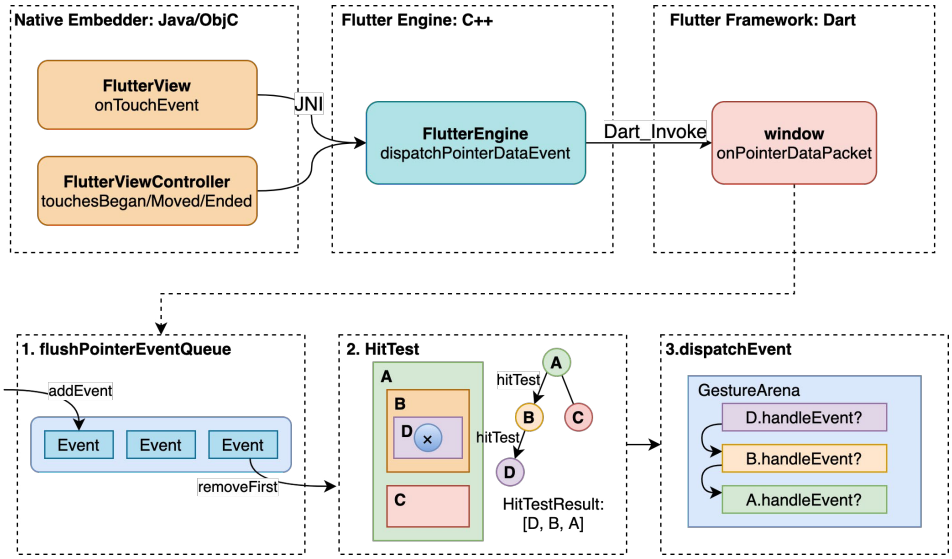
c. 与本地窗口对象关联的 EGLSurface

在 Android 的 AOSP 实现中，EGLSurface 可通过 EGL 库的 eglCreateWindowSurface 方法从本地窗口对象 **ANativeWindow** 创建而来。对于鸿蒙而言，虽然我们没有从公开文档找到类似的说明，但是 鸿蒙标准库 默认支持了 OpenGL ES，而且鸿蒙 SDK 中也附带了 EGL 相关的库及头文件，我们有理由相信在鸿蒙系统上，EGLSurface 也可以通过此方法从前一步生成的 **NativeLayer** 转化而来，在之后的验证中我们也确认了这一点：

```
// window->handle() 即为之前得到的 NativeLayer
EGLSurface surface = eglCreateWindowSurface(
      display, config_, reinterpret_cast<EGLNativeWindowType>(window-
>handle()),
      attribs);
//... 交给 Flutter 渲染管线
```

2. 交互能力实现

交互能力是支撑 Flutter 应用能够正常运行的另一个基本要求。在 Flutter 中，交互包含了各种触摸事件、鼠标事件、键盘录入事件的传递及消费。以触摸事件为例，Flutter 事件传递的整个流程如下图所示：



Flutter 事件分发

iOS/Android 的原生容器通过触摸事件的回调 API 接收到事件之后，会将其打包传递至引擎层，后者将事件传发给 Flutter 框架层，并完成事件的消费、分发和逻辑处理。同样，整个流程的大部分工作已经由 Flutter 统一，我们要做的仅仅是在原生容器上**监听**用户的输入，并**封装**成指定格式交给引擎层而已。

在鸿蒙系统上，我们可以借助平台提供的 多模输入 API，实现多种类型事件的监听：

```
flutterComponent.setTouchEventListener(touchEventListener); // 触摸及鼠标事件
flutterComponent.setKeyEventListener(keyEventListener); // 键盘录入事件
flutterComponent.setSpeechEventListener(speechEventListener); // 语音录入事件
```

对于事件的封装处理，可以复用 Android 已有逻辑，只需要关注鸿蒙与 Android 在事件处理上的对应关系即可，比如触摸事件的部分对应关系：

Android MotionEvent	HarmonyOS TouchEvent	说明
getActionMasked()	getAction()	当前触摸行为，点击/按下/移动等
getActionIndex()	getIndex()	按下的是第几个手指
getToolType()	getSourceDevice()	触摸设备类型，是手指还是鼠标
getX、getY	getPointerPosition	坐标偏移信息
ACTION_DOWN ACTION_UP ACTION_POINTER_DOWN ACTION_POINTER_UP	PRIMARY_POINT_DOWN PRIMARY_POINT_UP OTHER_POINT_DOWN OTHER_POINT_UP	触摸行为的具体类型

3. 其他必要的平台能力

为了保证 Flutter 应用能够正常运行，除了最基本的渲染和交互外，我们的嵌入层还要提供资源管理、事件循环、生命周期同步等平台能力。对于这些能力 Flutter 大多都在嵌入层的公共部分有抽象类声明，只需要使用鸿蒙 API 重新实现一遍即可。

比如资源管理，引擎提供了 AssetResolver 声明，我们可以使用鸿蒙 Rawfile API 来实现：

```
class HAPAssetMapping : public fml::Mapping {
 public:
  HAPAssetMapping(RawFile* asset) : asset_(asset) {}
  ~HAPAssetMapping() override { CloseRawFile(asset_); }

  size_t GetSize() const override { return GetRawFileSize(asset_); }

  const uint8_t* GetMapping() const override {
    return reinterpret_cast<const uint8_t*>(GetRawFileBuffer(asset_));
  }

 private:
  RawFile* const asset_;

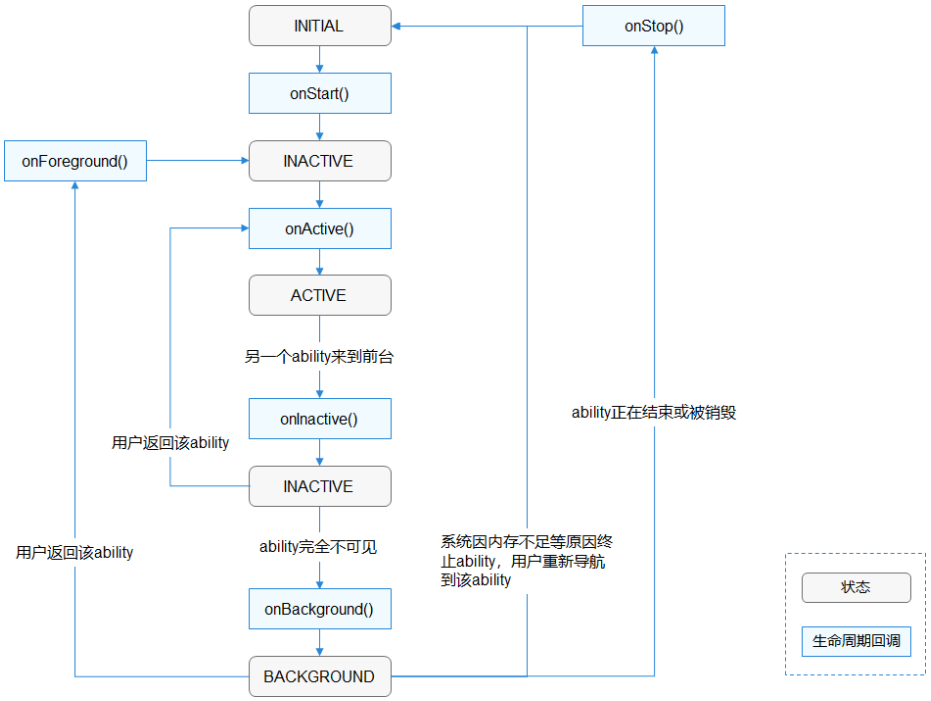
  FML_DISALLOW_COPY_AND_ASSIGN(HAPAssetMapping);
};
```

对于事件循环，引擎提供了 MessageLoopImpl 抽象类，我们可以使用鸿蒙 Native_EventHandler API 实现：

```
// runner_ 为鸿蒙 EventRunnerNativeImplement 的实例
void MessageLoopHarmony::Run() {
  FML_DCHECK(runner_ == GetEventRunnerNativeObjForThread());
  int result = ::EventRunnerRun(runner_);
  FML_DCHECK(result == 0);
}

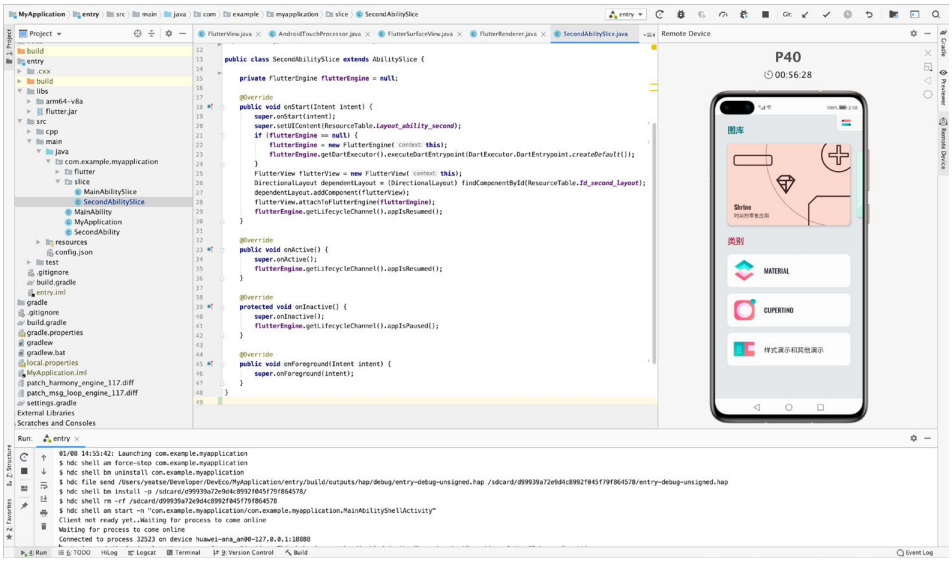
void MessageLoopHarmony::Terminate() {
  int result = ::EventRunnerStop(runner_);
  FML_DCHECK(result == 0);
}
```

对于生命周期的同步，鸿蒙的 Page Ability 提供了完整的生命周期回调（如下图所示），我们只需要在对应的时机将状态上报给引擎即可。



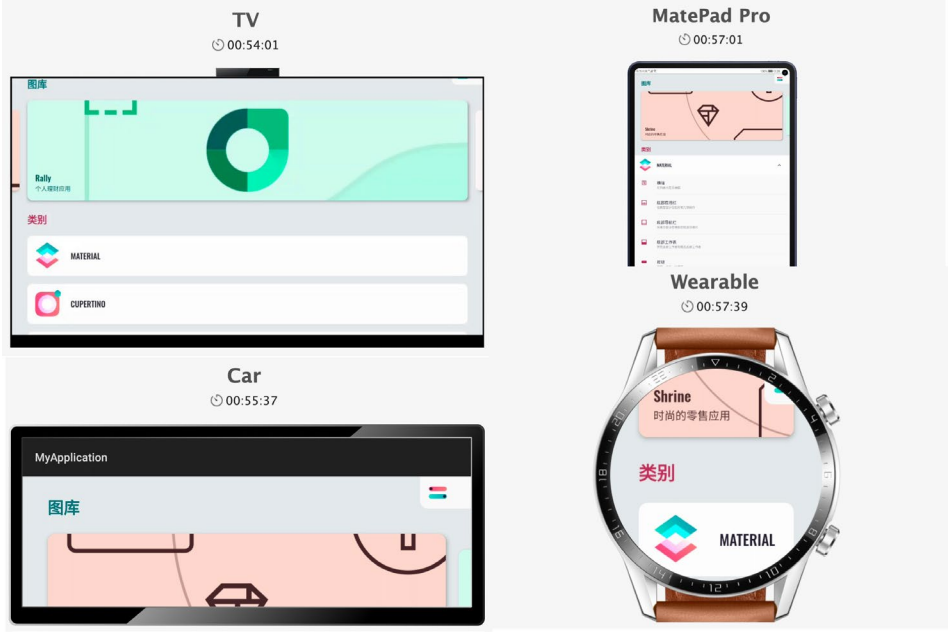
Page Ability Lifecycle

当以上这些能力都准备好之后，我们就可以成功把 Flutter 应用跑起来了。以下是通过 DevEco Studio 运行官方 flutter gallery 应用的截图，截图中 Flutter 引擎已经使用鸿蒙系统的平台能力进行了重写：



DevEco Running Flutte

借由鸿蒙的多设备支持能力，此应用甚至可在 TV、车机、手表、平板等设备上运行：



Flutter Multiple Devices

总结和展望

通过上述的构建和适配工作，我们以极小的开发成本实现了 Flutter 在鸿蒙系统上的移植，基于 Flutter 开发的上层业务几乎不做任何修改就可以在鸿蒙系统上原生运行，为迎接鸿蒙系统后续的大规模推广也提前做好了技术储备。

当然，故事到这里并没有结束。在最基本的运行和交互能力之上，我们更需要关注 Flutter 与鸿蒙自身生态的结合：如何优雅地适配鸿蒙的分布式技术？如何用 Flutter 实现设备之间的快速连接、资源共享？现有的众多 Flutter 插件如何应用到鸿蒙系统上？未来 MTFlutter 团队将在这些方面做更深入的探索，因为解决好这些问题，才是真正能让应用覆盖用户生活的全场景的关键。

参考文献

- https://developer.huawei.com/consumer/cn/events/hdc2020/
- https://developer.harmonyos.com/cn/documentation
- https://flutter.dev/docs/resources/architectural-overview
- https://github.com/flutter/flutter/wiki/Custom-Flutter-Engine-Embedders

作者简介

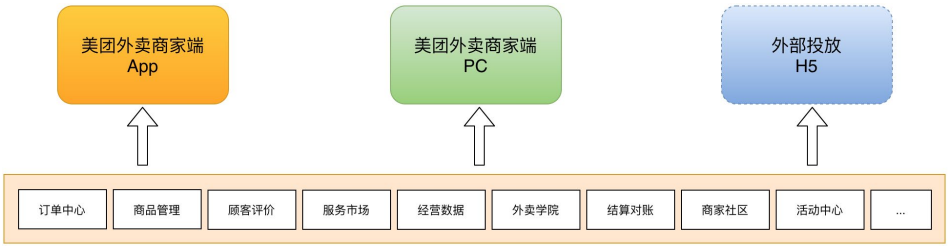
杨超，2016 年加入美团外卖技术团队，目前主要负责 MTFlutter 相关的基础建设工作。

FlutterWeb 在美团外卖的实践

作者：典胜　凌霄　海阔

一、背景

1.1 业务背景



美团外卖商家端业务形态

美团外卖商家端业务围绕数百万商家，在 PC 和 App 上分别提供了交易履约、运营、广告、营销等一系列功能，且经常有外投 H5 的场景（如外卖学院、商家社区、营销活动等）。在这种多形态的业务场景下，如何保障多端体验的一致性，以及如何提升多端迭代的效率，一直是商家端产研关注的重点。

1.1.1 保障多端体验一致性

由于端能力的不同，导致了业务在 App 和 Web 上存在较大的表现差异，例如：App 上自带动画转场，而在 Web 中的实现成本却较高，往往也就降级舍弃了这部分功能。此外，即使我们可利用公司内部的 Roo、MTDUI 等多端 UI 组件库来尽量抹平各端的 UI 差异，但由于组件库在各端的实现不尽相同，很难做到完美的一致性体验。

1.1.2 提升多端迭代效率

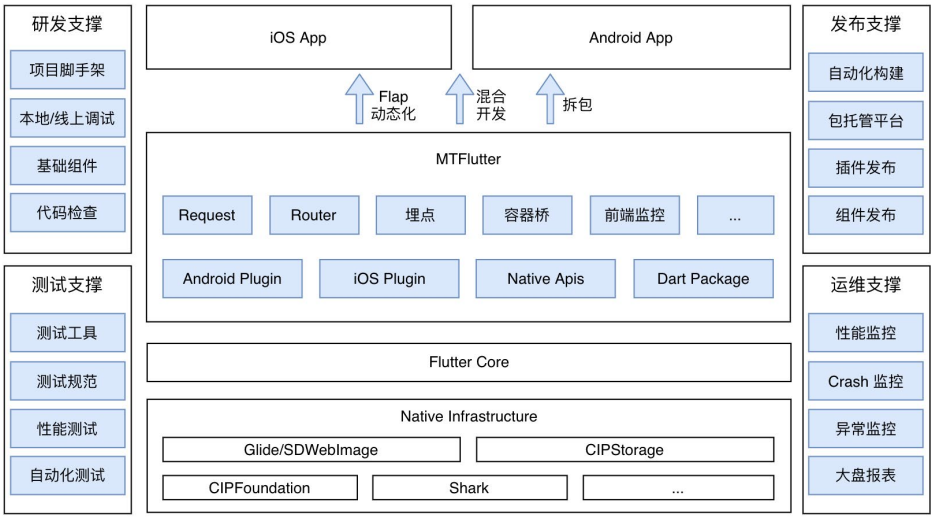
由于各端技术体系的不同，涉及多端的需求往往需要不同的开发、测试团队各自完成开发、联调、测试、上线等流程，占用资源巨大，在各团队不可并行支持的情况下，

甚至可能导致整个业务交付周期被拉长。虽然 React Native、Flutter 等跨平台方案解决了一部分复用的问题，但显然在商家端业务场景下是远远不够的，我们的目标是要达到全平台（Android、iOS、PC、H5）复用，最大化地提升多端的迭代效率。

1.2 技术背景

1.2.1 Flutter 在美团外卖商家端的储备

MTFlutter 是美团外卖搭建起的公司级 Flutter 研发生态，它的架构图如下图所示：

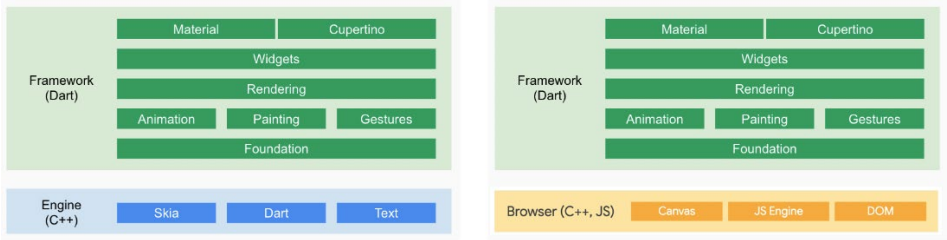


MTFlutter 架构图

如图所示，MTFlutter 已涵盖研发、调试、测试、发布、线上运维及工程管理整套闭环，同时落地了动态化解决方案，支撑了公司多个业务发展。在大前端融合的趋势下，美团外卖商家端持续在探索更优的多端复用方案，通过 MTFlutter 生态的建设，目前 Flutter 技术栈已覆盖商家端 App 中 90% 以上的业务，同时具备 Flutter 开发能力的同学也达到 90% 以上。因此，在有足够技术“储备”的前提下，我们能够基于 Flutter 做全平台（Android、iOS、PC、H5）复用的探索。

1.2.2 Flutter Web 的支持

2018 年 Google 首次公开 Flutter Web Beta 版，旨在进一步实现一份代码、多端运行的愿景。目前，Flutter Web 已被正式合入 Master，期间经过无数工程师的努力，Flutter Web 已能提供与 Flutter Natvie 较统一的交互行为和视觉体验。



Flutter Native VS Flutter Web

如上图可知，Flutter Web 与 Flutter Native 的整体架构相似，二者共用 Framework 层（绿色部分），提供了包括动画、手势、基础 Widget 类，以及大部分应用所需的 Material/Cupertino 主题 Widget 集合。区别在于：Flutter Web 重写了 dart:ui 层（黄色部分），利用 DOM、Canvas 对齐了 Flutter Native 的 UI 渲染能力，使得 Flutter 编写的 UI 能够在现代浏览器上正常展示。

此外，得益于 dart2js 这个早已成熟的工具，Dart 逻辑能够很容易的转换为 JavaScript，进而在 Web 中被正常运行。

二、面临的挑战

综上所述，我们选择基于 Flutter Web 探索跨端（App\PC\H5）解决方案，真正实现“Write Once & Run AnyWhere”。当然，面临挑战也是巨大的，主要体现在 Flutter 和 MTFlutter 现阶段对 Web 支持还不是很充足。

2.1 Flutter Web 现状

Google 官方目前对 Flutter Web 的工作主要还集中在 dart:ui（Web）的对齐，工程化和性能相关的事项做的还比较少，例如：

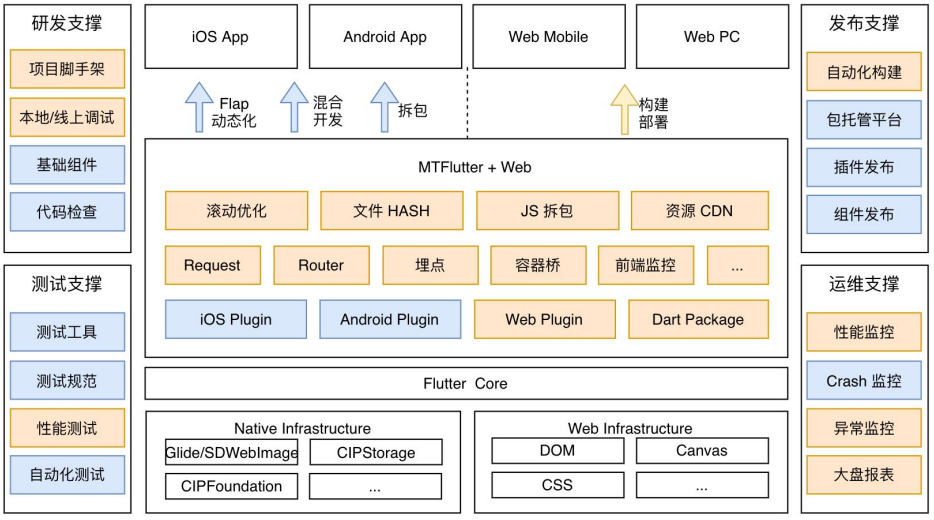
- Flutter Web 构建产物较简陋，只是简单的输出 main.dart.js（1.1M，未 Gzip）和 图片等静态资源，缺少 JS 拆包、文件 Hash、资源上传 CDN 等优化工作，极大影响了页面的加载性能。
- 由于 Flutter Web 自身实现了一套页面滚动机制，页面滚动过程中，会频繁计算位置信息，引起滚动区域内容被重新创建，最终导致页面滚动性能较差。

2.2 MTFlutter 现状

虽然 MTFlutter 做了诸多 Flutter Native 层面的定制与优化，但在 Flutter Web 上的建设才刚起步，具体表现在：

- MTFlutter 现有的基础依赖如：Request（请求封装）、Router（路由）、埋点、容器桥、前端监控，尚未支持在 Web 中的实现。
- MTFlutter 已实现了完整的 Flutter Module 的打包发布流程，但并不支持 Web 的构建与部署。

三、整体设计



MTFlutter 架构图

上图为 MTFlutter + Web 架构图，由图可知 Flutter Web 页面要满足投产要求，还有大量的工作（上图黄色部分所示），主要包括：

- 扩展基础依赖（如：Request、Router、埋点等）在 Web 侧的支持。
- 完善工程化建设，例如：静态资源优化、构建与部署自动化。
- 深入滚动性能与页面加载性能优化，使得 Flutter Web 能够满足基本的投产要求。

四、详细设计

4.1 基础依赖建设

企业级应用的基础开发依赖（如：请求库、路由库、埋点库等），要重新在 Flutter 中用 Dart 搭建一套，时间成本、兼容性、风险等都是不可控的。而 MTFlutter 是基于原有 Native 基础依赖开发的 Plugin，因此并不支持 Web 端。此章节将展开介绍如何丝滑无感地扩展 MTFlutter 基础依赖在 Web 端的实现。

4.1.1 Flutter Package 分平台编程

在 Flutter 中通过使用 Package 可以创建易于共享的模块化代码。官方强烈推荐使用 Package 形式管理各种工具方法。在官方定义中 Package 包含以下两种类别：

- **Dart Package**：用 Dart 编写的常规 Package，其中一些可能包含依赖于 Flutter 框架的特定功能，其使用范围仅限于 Flutter，例如 path。
- **Plugin Package**：用 Dart 编写 API 多个平台各自实现的特殊 Dart Package。Plugin Package 可以为 Android（使用 Kotlin 或 Java）、iOS（使用 Swift 或 Objective-C）、Web、macOS、Windows 或 Linux 或其任意组合编写插件包。

下面分别对这两种类型 Package 中如何分平台编程进行介绍。

(1) Dart Package

Dart Package 是纯 Dart 编写，因此大部分代码均可由 dart2js 直接编译出 Web 平台可运行的代码，但某些涉及 Native 能力的库（如 dart:io）是无法被转译的，因此

需要有对平台进行兼容的方法，下面介绍两种在 Dart Package 中分平台编程的方案。

代码级别分平台

针对代码级别的分平台，我们可以借助 Flutter SDK 提供的一个常量 kIsWeb。使用方法如下：

> 查看源码可知，kIsWeb 之所以能被用于判断 Web 平台，是利用了 JavaScript 不支持整型的特征，在 Web 环境下，Dart 的 double 和 int 由相同类型的对象支持，浮点数“0.0”等于整数“0”，对于在 AOT 或 VM 上运行的 Dart 代码却并非如此。

```
import 'package:flutter/foundation.dart';
if (kIsWeb) {
  print('Web 端')
} else {
  print('其他端');
}
```

文件级别分平台

针对文件级别分平台，我们利用条件导入导出，其中条件导出具体用法如下：

```
// tool.dart
export 'src/tool_native.dart' // 兜底导出，即没有命中条件时导出的文件
  if (dart.library.html) 'src/tool_web.dart'; // web 端导出的文件，该文件中可以使用 dart:html，也可以通过判断 dart.library.js 导出 Web 端文件。
```

```
// 引入 tool.dart
import 'package:tool/tool.dart';
void main() {
  print('import tool');
}
```

条件导入和条件导出类似，仅需将 export 改为 import 即可。在业务开发中这也是一种非常实用的分平台编程方法。

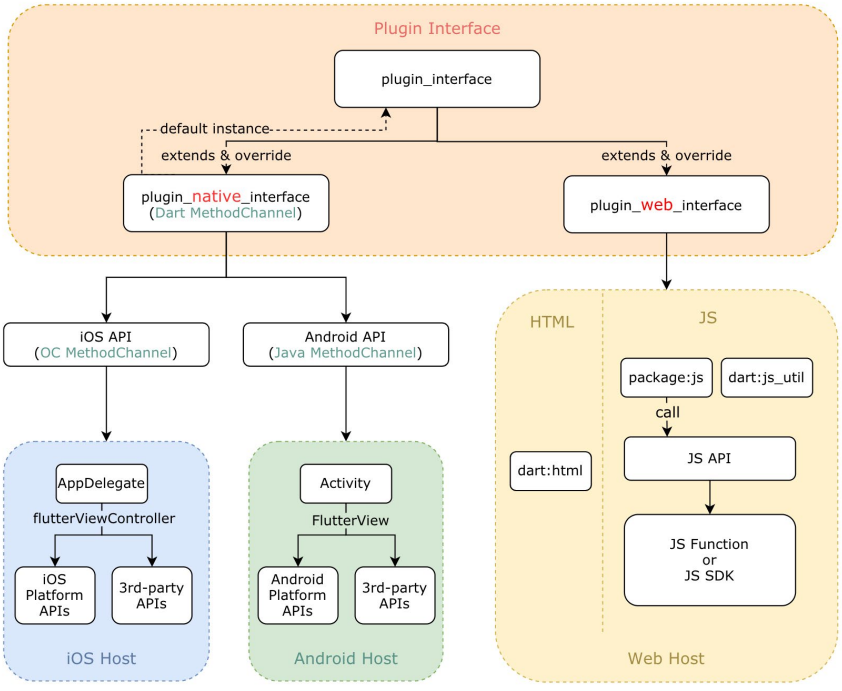
(2) Plugin Package

Plugin Package (下文简称为 Plugin) 在 Android 和 iOS 平台都是通过 Method-Channel 实现在 UI 层和 Platform 层传递消息从而达到特定平台支持的，官方文档中也全方位介绍了在 Android 和 iOS 平台的具体实现方法及例子，Web 平台的实现却介绍的较少。总结起来，Web 平台和 Native 平台实现方式的不同主要集中在下面两点。

首先，Web Plugin 推荐的方式不是以其平台特有的 JS 语言实现，而是通过 Dart Library 或 Package 实现，对于已有现成可用的 JS SDK 或需要大量使用 JS 实现功能的情况下，官方提供了 package:js 包调用 Javascript，从而实现与 Javascript 的交互。

其次，Web Plugin 不是通过注册 MethodChannel 传递消息的，Flutter 内部可直接调用通过官方指定形式 (Federated Plugin) 编写的 Flutter Web Plugin 类。

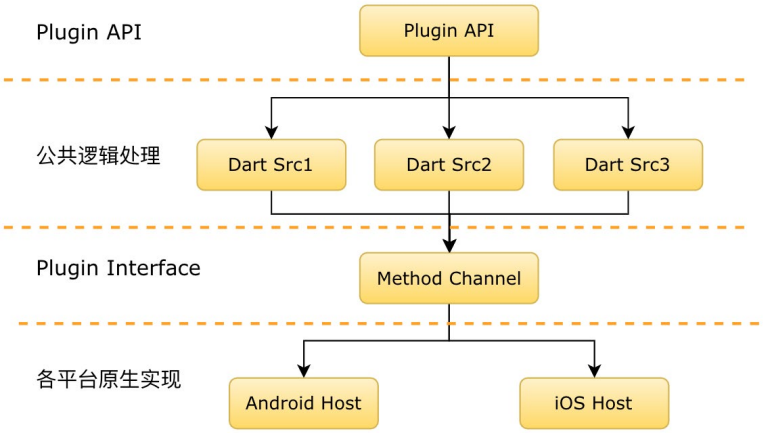
下图完整的展示了一个 Plugin 的整体架构：



Flutter Plugin 架构图

4.1.2 基础依赖建设

整体来讲，MTFlutter 基础依赖都是使用 Plugin 的形式开发维护的。为处理依赖中的公共逻辑，提高 Plugin 的可扩展性，MTFlutter Plugin 在 Flutter Plugin 架构（各平台原生实现层和 Plugin Interface 层）之上又增加了公共逻辑处理层，最终暴露给用户是 Plugin API 层提供的接口。MTFlutter Plugin 架构图如下：



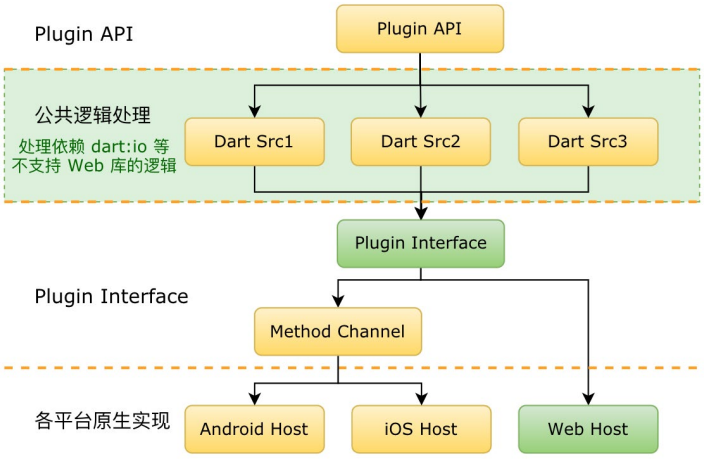
MTFlutter Plugin 架构图

在细节实现上，由于项目中各种依赖的类型之间存在着差异，因此在依赖处理上也略有不同，下面介绍拥有不同特点的依赖所对应解决方案。

（1）各平台实现能在 Web 侧对齐的场景，如埋点库

埋点库无论在 Native 端还是在 Web 端都是使用公司统一提供的 SDK，在 API 设计上具有天然的一致性，因此我们完全有能力在 Plugin Interface 层对齐所有接口，上层业务逻辑只需按需做些兼容处理即可。埋点库 Web 端扩展的整体设计思路如下：

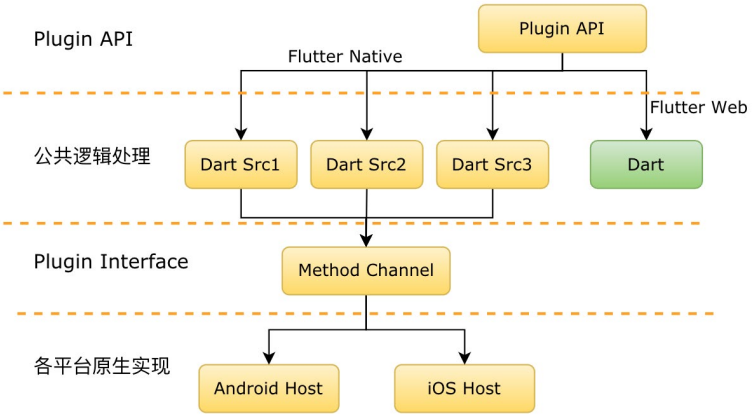
- 在业务项目的 web/index.html 文件中直接引入 Script 脚本并且进行初始化（注意：引入 Script 的位置，需要放在 main.dart.js 前面）。
- 借助 package:js 库调用埋点 JS SDK，对齐 Flutter 埋点库的 API，实现 Flutter Plugin 的 Web 端支持，详细架构图如下图所示：



埋点库架构图

（2）各平台实现在 Web 侧无法对齐的场景，如路由库

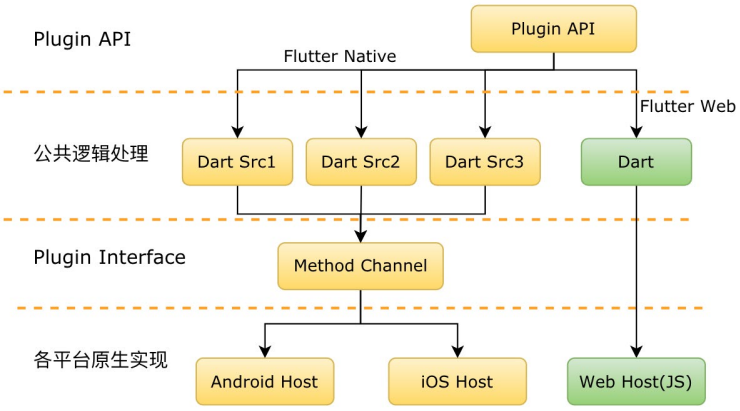
MTFlutter 路由库是 Native 底层维护的一套全新的路由体系，依靠原生支持提供了强大的定制化功能，而在 Web 端无法这些无法在各平台原生实现层达到 100% 支持。由于 MTFlutter Plugin 最终暴露的是 Plugin API，因此我们选择直接对齐 Plugin API 实现路由库在 Web 端的支持（借助 Flutter Navigator、dart:html 用纯 Dart 语言完成了扩展），详细架构如下图所示：



路由库架构图

（3）Web 端需要通过大量 JS 实现功能的依赖库，如请求库

由于在现有的 Web 请求中统一封装着大量的业务处理逻辑（如拦截器、异常上报等），如果用 Dart 重新实现一遍，成本还是较高的。想复用原有基于 Axios（ JS 请求库）封装的请求库就相当于让 Plugin 的 Web 平台实现使用 JS 语言。Dart 和 JS 交互是通过 package:js 进行接口调用，因此我们在公共逻辑处理层用 Dart 对齐了相应的 API，详细架构图如下图所示：



请求库架构图

4.2 性能优化

常规的 Web 项目中，为了保证页面有更好的加载和渲染性能，在静态资源文件的处理方面，我们需要做很多的工作，例如：资源文件 Hash 化、CDN 化、按需加载处理等，这些可以通过 Webpack、Rollup 等构建工具进行预处理。但在 Flutter Web 中，这些预处理的操作目前官方还不支持，原因是 Flutter 暴露给我们的命令只有一个 `flutter build web`，导致我们无法直接进行更细粒度的个性化定制。如果想要让 Flutter Web 达到企业级应用的标准，我们需要更深层次的探索 Flutter SDK 的运行原理。下面我们列出目前遇到的性能问题及其解决方案。

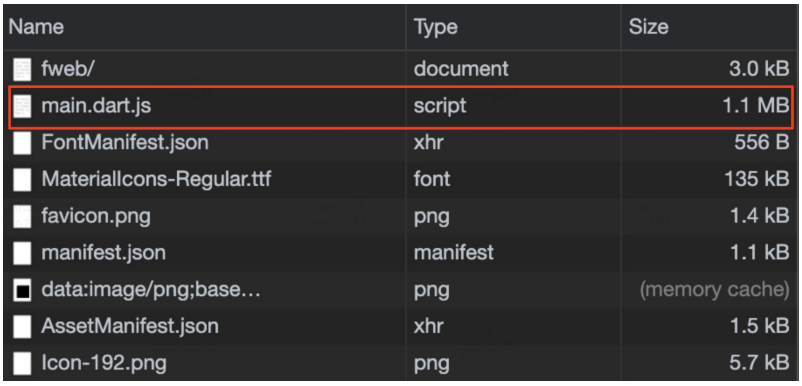
4.2.1 目前存在的性能问题

Google 官方对 Flutter Web 性能优化所做的事项还比较少，编译输出的页面存在较

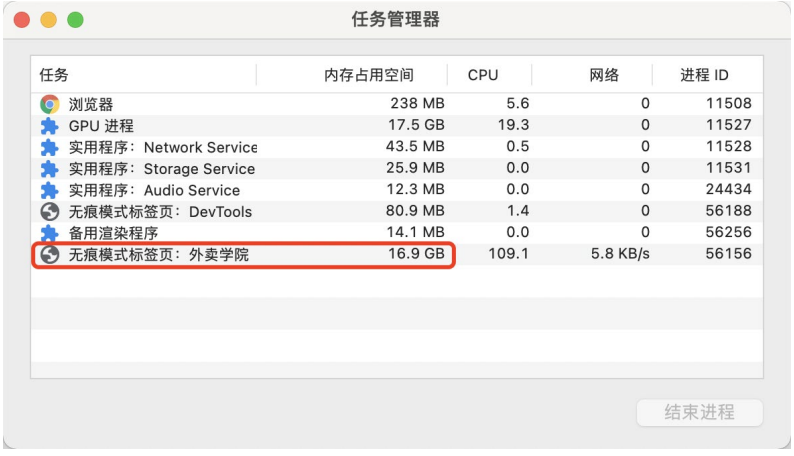
大的性能问题，主要体现在以下两方面：

- **首屏渲染时间长**。即使使用了 FutureBuilder 把业务代码拆分成 xxx.part.js 之后，main.dart.js 体积依然维持在 1.1M。单一文件加载、解析时间过长，且静态资源缺少 CDN 化的支持，势必会影响首屏的渲染时间。
- **滚动性能较差**。 Flutter Web 自身实现了一套页面滚动机制，在页面滚动过程中，会频繁的创建 Canvas，最终导致滚动性能问题，甚至引起页面 Crash。

通过下图对浏览器网络监控情况的展示，可以清晰的反映出以上问题：

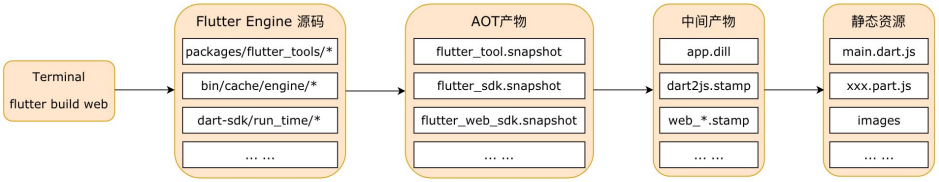


浏览器网络监控



页面滚动过程中，内存的占用情况

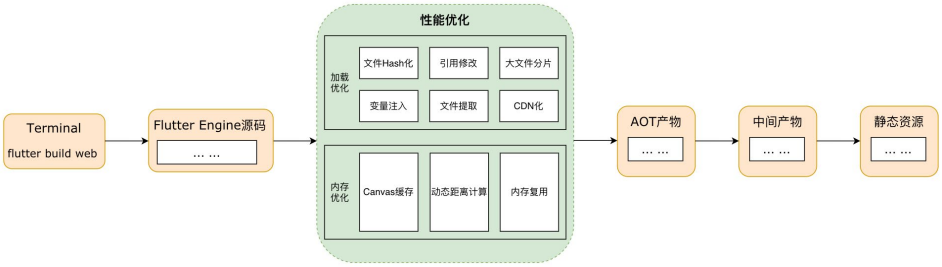
为了解决上述的性能问题，我们探索了 Flutter SDK 编译过程，总结出从 Flutter 业务代码到 Web 产物的整体流程，详细流程如下图所示：



编译流程

从流程中我们可以看到，Flutter 在 Web 端目前只支持 Dart->JS 的转换，以及 UI 层的对齐，在工程化和性能优化方面做的工作并不多。

因此，我们必须解决以上的性能问题，才能保证我们的业务可以正常的交付。通过对编译流程的仔细分析与梳理，我们在 AOT 产物生成之前对 Flutter SDK 进行定制，分别进行**加载性能优化**和**内存性能优化**，下面分别介绍这两部分的内容。



Flutter SDK 进行定制后的流程

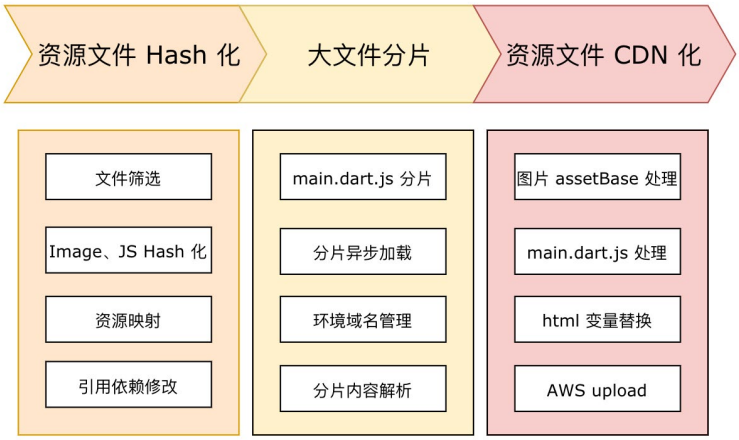
4.2.2 加载性能优化

运行 `flutter build web` 命令之后，我们得到的主要静态资源有：主文件 main.dart.js（1.1M），各页面的业务代码 xxx.part.js（使用 FutureBuilder 后）、图片文件。直接应用这些资源到项目中，会遇到以下问题：

- **功能无法及时更新**：浏览器对同名文件的缓存，可能导致程序代码不被及时更新或者出现执行错乱。

- **首屏渲染性能差**：main.dart.js 文件过大，单一文件加载、解析时间过长，势必会影响首屏的渲染时间。
- **无法使用 CDN**：Flutter 仅支持相对路径的加载方式，无法使用当前域名以外的 CDN 域名，导致无法享受 CDN 带来的优势。

为此，在加载部分我们对 Flutter SDK 增加了如下三方面的优化，以达到线上运行的标准，优化步骤如下图所示：



优化步骤

资源文件 Hash 化

除了 web/index.html 文件之外，我们要对所有的引用到文件进行 Hash 化。对 build_system/web.dart 的修改按以下步骤进行：

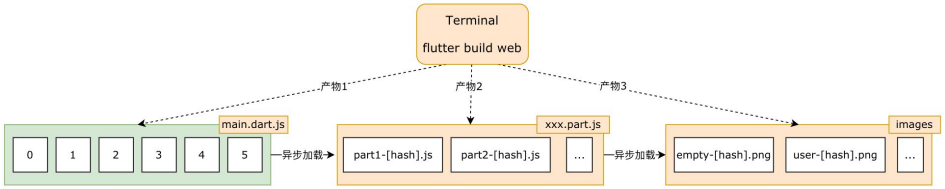
- 遍历产物目录，并建立 ResourceMap。
- 分别计算每个文件的 Hash 值。
- 为新文件命名为 name-[hash].xxx。
- 修改新文件名在对应文件中的引用关系。

大文件分片

Flutter Web 编译之后会生成 main.dart.js 这一主文件，体积为 1.1M（ Gzip 之后

约 400K ），这给页面的加载性能带来很大的影响。为此，我们对代码进行分片，借助浏览器对多文件并行加载的特性，可以有效提升页面的加载性能。

具体实施步骤是：将 `main.dart.js` 在 Dart 侧拆分成多份纯文本文件，前端通过 XHR 的方式并行加载并按顺序拼接成 Javascript 代码置于 <script> 标签中，从而实现分片文件的并行加载。



Hash 化以及分片之后，静态资源的引用关系

资源文件 CDN 化

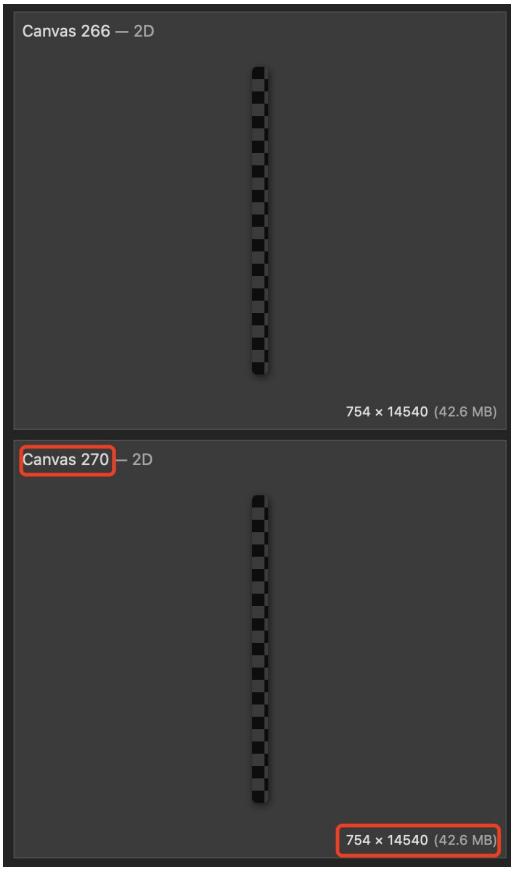
由于 Flutter Web 资源引用机制的不同，即使在资源文件 Hash 化的过程中，把文件的相对路径替换成带 CDN 域名的绝对路径，也无法实现 CDN 资源的加载。同时本地测试发现图片和 Javascript 资源的加载逻辑还不尽相同，为此针对各自的加载逻辑要分别进行优化。

- **图片处理**：经过对源码的大量阅读及梳理，我们发现图片请求的 URL 首先会读取 `meta` 标签中 `assetBase` 值进行 URL 路径拼接，根据拼接好的 URL 来获取资源。目前，在项目 `web/index.html` 模板文件中并没有 `meta` 标签，于是就会根据相对路径进行请求。解决方案是在编译过程中，根据请求环境增加 `meta` 标签并把 `content` 设置为 CDN 路径。
- **JavaScript 处理**：为了解决图片资源文件的加载问题，我们虽然增加了 `assetBase` 的 `meta` 标签，但发现 `xxx.part.js` 文件依然使用当前域名进行加载，可见 Javascript 资源的加载和图片资源加载的逻辑不尽相同。对 `main.dart.js` 源码分析，我们发现请求 `xxx.part.js` 的域名取决于包含 `main.dart.js` 内容的 `Script` 标签的 `src` 属性。通过对 `js_helper.`

dart 的动态编译，我们把读取 src 属性修改为读取 window.assetBase 这一全局变量（meta 标签中 assetBase 值加工后的变量）来实现 xxx.part.js 文件的 CDN 加载。

4.2.3 滚动性能优化

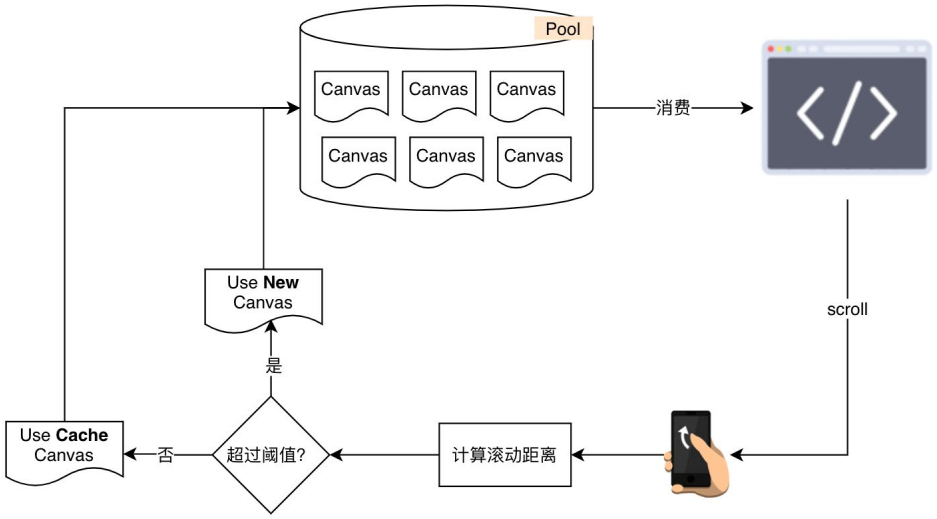
当页面出现可滚动区域时，每次页面滚动会创建大量的 Canvas。使用 Safari 的 Canvas 分析工具，我们发现问题的根本原因是页面滚动的过程中，Flutter 会频繁的创建滚动区域的 Canvas，每次创建的 Canvas 内存都在 10 ~ 70M 不等，滚动的内容越多，内存的占用就会越大，这样滚动几帧之后，内存的占用就会超过浏览器的阈值。



Safari 图形工具中展示 Canvas 的占用

Flutter 对 Canvas 的管理有一个 ReusablePool 的概念，在初始过程中会创建一定的数量的 Canvas，页面交互过程中没有变化的部分，会优先使用 pool 中已经缓存过的 Canvas 以便能够节省内存。由于 Flutter Web 自身实现了一套页面滚动机制，页面滚动过程中，会频繁计算位置信息，引起滚动区域内容被重新创建，这就是为什么每次滚动都会创建 Canvas 的原因。

我们设计的解决方案是：修改 FlutterSDK，在滚动的过程中定义一个阈值，当滚动的高度在阈值范围内，我们就会把当前的 Canvas 缓存起来。这样选择性的创建和销毁 Canvas 可以有效的缓解内存压力，从而提升页面滚动性能。



优化之后浏览器创建和销毁 Canvas 的过程

4.3 构建与部署

4.3.1 Docker 镜像定制

由于 MTFlutter Web 环境安装步骤较固定，且整个安装过程耗时较长（> 80s）。因此将其定制为 Docker 镜像并集成至 Talos，Flutter Web 编译阶段便能免去安装流程，有效提升构建效率。Docker 镜像定制和发布的详细流程见官方文档，本文不再赘述。其中用于定制 Flutter Web 镜像的 Dockerfile 文件如下：

```
FROM $BaseImage \# 继承基础镜像
RUN apt-get update
RUN apt-get install rubygems -y
RUN gem install flutter-cli
RUN flutter-cli install
ENV PATH="/$User/.flutter_sdk/bin:${PATH}"
ENV PUB\_HOSTED\_URL="https://xxx.com" \# 私有 pub 服务
ENV FLUTTER\_STORAGE\_BASE_URL="https://storage.flutter-io.cn"
RUN ~/.flutter_sdk/bin/flutter config --enable-web
```

4.3.2 持续交付与部署

为了实现持续交付与部署，我们建立起了 Flutter Web 在 Talos（美团内部前端持续交付解决方案）中的发布流水线：



Talos 发布流水线

可以看到，流水线中已经免去了 MTFlutter Web 环境的安装流程，现有流水线中重要节点介绍如下：

- Flutter-Web-Build 利用 Docker 内置的 MTFlutter 进行 Web 编译。
- Flutter-Web-Publish 负责将编译产物上传美团资源存储服务器。

五、成果展示

5.1 效果展示

我们在美团外卖商家学院（一个以文章、视频等形式帮助商家学习外卖运营知识、了解行业发展和平台策略的平台，它有很强的传播属性，具有外部投放的场景）率先落地了 Flutter Web，现以商家学院视频内容页为例，对比 Flutter Native 和 Flutter Web 的展现效果：



Flutter Native

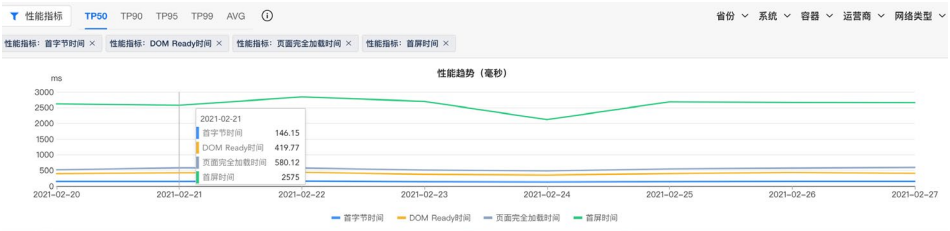


Flutter Web

可以看出，两者的交互、视觉体验是高度一致的，既保证了业务在 App 内接近 Native 的体验，又极大提高了 Web 与 Flutter Native 的体验一致性。

5.2 页面加载性能

如前文所述，我们实施了一系列针对 Flutter Web 的资源优化手段，使得页面加载性能有较大提升，其中页面完全加载时间大致由 **1300ms（TP50）**降到了 **580ms（TP50）**，更多的性能指标数据见下图：

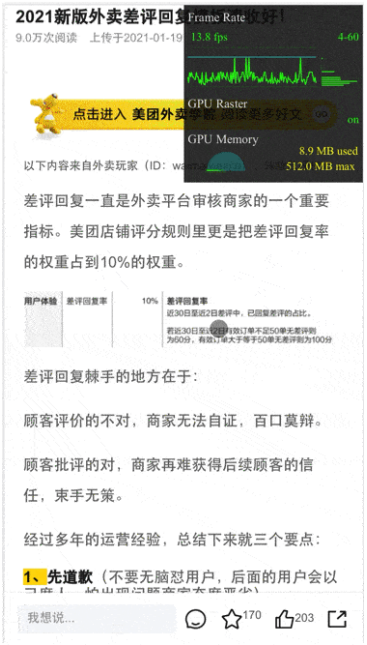


某 7 日性能趋势图

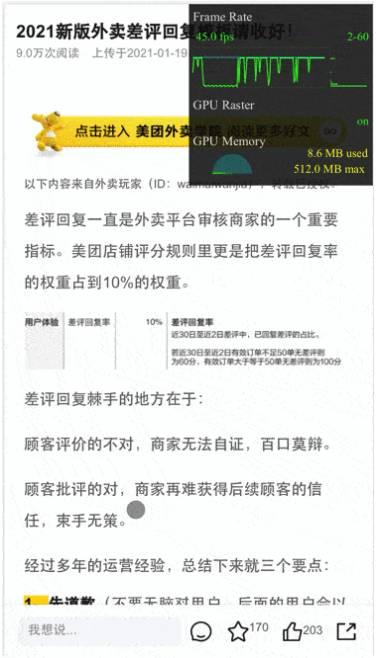
可以看到 Flutter Web 与现有 Web 项目性能指标数据差距已不大，可满足日常业务要求。但加载性能数据仍有较大的优化空间，我们会持续对其进行探索。

5.3 滚动性能

针对滚动优化，我们通过修改 Flutter SDK，使得 Canvas 在页面滚动时无需重复创建，而是被缓存起来。这样大大节省了内存的开销（优化后页面内存占用稳定为 100M 左右，与常规 Web 页面无异），同时在一定程度上提升了滚动性能。以商家学院文章内容页为例，对比优化前后滚动 FPS ：



优化前 FPS



优化后 FPS

可以看到，Flutter Web 页面滚动性能已得到较大提升，足以应对大部分业务场景。但由于 Flutter Web 页面滚动过程中会频繁进行位置信息的计算，在复杂的业务场景（如页面存在大量动画）仍然会暴露出一定的问题。因此对滚动性能的进一步优化也会是我们未来的工作重心。

5.4 业务迭代效率

基于团队对 Flutter Web 工程化能力的建设和 Flutter 良好的跨平台特性，Flutter Web 在美团外卖商家学院改版需求的落地，大大提升了迭代效率，估算人效提升 40% 以上，计算公式为：

$$E = (1 - (1 / Np)) - Ci$$

其中 E 代表人效提升，Ci 指的是兼容和适配所耗费的时间，Np 表示业务跨端数量，目前美团外卖商家学院在 Native 和 H5 两端完成了复用，后续在 PC 侧需求的对齐中，效率提升数值会被放大，预计人效提升达 60% 以上。同时我们将在更多的业务中进行推广与应用，提升整体业务的迭代效率。

六、总结与展望

综上所述，美团外卖商家端多元的业务形态和足够的技术“储备”，使得基于 Flutter 实现多端复用成为了可能。而 Flutter Web 在美团外卖商家学院业务中也取得了阶段性的成果，实现了 App、H5 侧的体验一致性，为后续推动更多业务线实现 App-Web 一体化打下了坚实的基础。

可以预见的是，基于 Flutter Web 实现的多端复用，势必会有效缩短项目交付周期。但由于我们对页面加载性能、滚动性能做的仍不够完美，不足以应对更加复杂的业务场景，因此我们依然还有许多工作：

- **页面滚动性能优化**：由于 Flutter 与 Web 的布局差异，使得 dart:ui（Web）也受 Flutter Native 的布局约束，如何打破这样的约束，是解决滚动性能问题的关键。

- **页面加载性能优化**：当前的页面加载性能仍有较大优化空间，需要对 Flutter 进行编译干预与优化（如按需分离 main.dart.js），减小资源包大小，有效提升页面加载性能。
- **Flutter Web 基建**：完善并优化开发、调试、编译、构建、部署链路，使得新老项目能快速接入 Flutter Web。
- **Flutter Web 在 PC 侧的复用**：与 UED 团队共同制订 PC 与 App 适配规范，同时基于 Dart2js 和 dart:ui（Web）的强大能力，实现逻辑的抽象，完成组件、模块的适配，达到提效最大化；
- **跟进 Flutter 官方动向**：Flutter 2.0 的发布，稳定了对 Web 的支持，同时默认采用 Canvaskit 编译模式，此模式下对页面滚动性能有较大提升。但由于 canvaskit.wasm 文件过于庞大（2.5M），降低了加载性能，因此目前仍不建议在 Web 侧直接使用 Canvaskit。不过官方承诺会在 2021 年对性能进行整体优化，还是值得期待的，我们也将保持跟进和沟通。

我们会持续基于 Flutter Web 做更多的探索和尝试。如果您对 Flutter Web 也感兴趣，欢迎大家在文末评论区留言或者给出建议，非常感谢。

版权声明

本文系美团技术团队出品，著作权归属美团，未经允许，请勿转载或者用于其他商业用途。如用作公益用途，请务必注明：内容转载自美团技术团队。任何商用行为，请发送邮件至 tech@meituan.com 申请授权。

FlutterWeb 性能优化探索与实践

作者：典胜　海阔　徐亮

一、背景

1.1 关于 FlutterWeb

时间回拨到 2018 年，Google 首次公开 FlutterWeb Beta 版，表露出要实现一份代码、多端运行的愿景。经过无数工程师两年多的努力，在今年年初（2021 年 3 月份），Flutter 2.0 正式对外发布，它将 FlutterWeb 功能并入了 Stable Channel，意味着 Google 更加坚定了多端复用的决心。



图 1　FlutterWeb 历史

当然 Google 的“野心”不是没有底气的，主要体现在它强大的跨端能力上，我们看一下 Flutter 的跨端能力在 Web 侧是如何体现的：

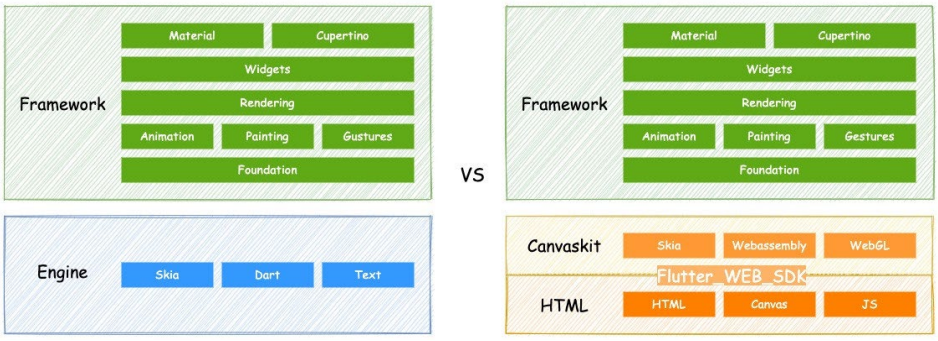


图 2　Flutter 跨端能力

上图分别是 FlutterNative 和 FlutterWeb 的架构图。通过对比可以看出，应用层

Framework 是公用的，意味着在 FlutterWeb 中我们也可以直接使用 Widgets、Gestures 等组件来实现逻辑跨端。而关于渲染跨端，FlutterWeb 提供了两种模式来对齐 Engine 层的渲染能力：Canvaskit Render 和 HTML Render，下方表格对两者的区别进行了对比：

模式	加载性能	滚动性能	兼容性
CanvasKit Render	**加载性能差**，初始包大小约为 2.5M（主要为 wasm 文件），优化空间小	**滚动性能较好**	**兼容性差**，使用 WebAssembly + WebGL 对齐渲染能力
HTML Render	**加载性能一般**，初始包大小约为 1.2M，优化空间大	**滚动性能一般**，在复杂动画、图表场景下稍显吃力	**兼容性优秀**，使用 HTML + Canvas 对齐渲染能力

图 3　模式对比

Canvaskit Render 模式：底层基于 Skia 的 WebAssembly 版本，而上层使用 WebGL 进行渲染，因此能较好地保证一致性和滚动性能，但糟糕的兼容性（WebAssembly 从 Chrome 57 版本才开始支持）是我们需要面对的问题。此外 Skia 的 WebAssembly 文件大小达到了 2.5M，且 Skia 自绘引擎需要字体库支持，这意味着需要依赖超大的中文字体文件，对页面加载性能影响较大，因此目前并不推荐在 Web 中直接使用 Canvaskit Render（官方也建议将 Canvaskit Render 模式用于桌面应用）。

HTML Render 模式：利用 HTML + Canvas 对齐了 Engine 层的渲染能力，因此兼容性表现优秀。另外，MTFlutterWeb 对滚动性能已有过探索和实践，目前能够应对大部分业务场景。而关于加载性能，此模式下的初始包为 1.2M，是 Canvaskit Render 模式产物体积的 1/2，且我们可对编译流程进行干预，控制输出产物，因此优化空间较大。

基于以上原因，美团外卖技术团队选择在 HTML Render 模式下对 FlutterWeb 页面的性能进行优化探索。

1.2 业务现状

美团外卖商家端以 App、PC 等多元化的形态为商家提供了订单管理、商品维护、顾客评价、外卖课堂等一系列服务，且 App、PC 双端业务功能基本对齐。此外，我们还在 PC 上特供了针对连锁商家的多店管理功能。同时，为满足平台运营诉求，部分业务具有外投 H5 场景，例如美团外卖商家课堂，它是一个以文章、视频等形式帮助商家学习外卖运营知识、了解行业发展和跟进经营策略的内容平台，具有较强的传播属性，因此我们提供了站外分享的能力。

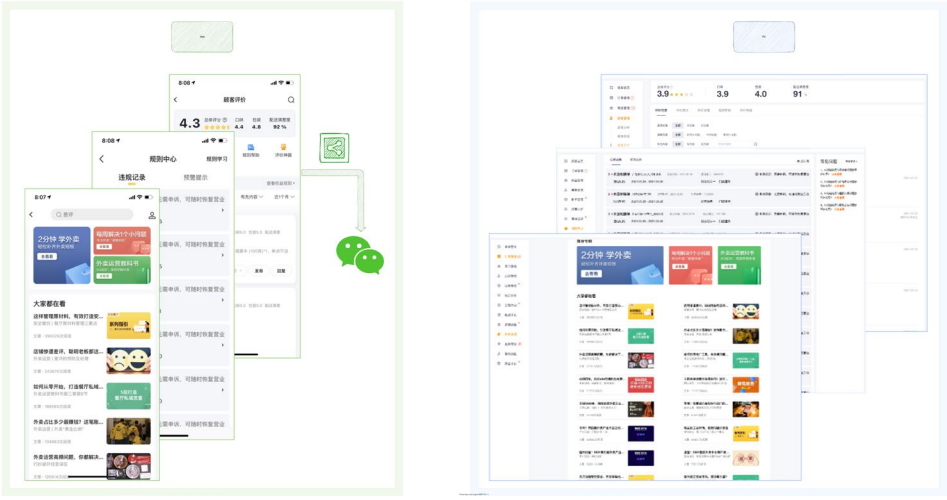


图 4　业务形态

为了实现多端（App、PC、H5）复用，提升研发效率，我们于 2021 年年初开始着手 MTFlutterWeb 研发体系的建设。目前，我们基于 MTFlutterWeb 完成提效的业务超过了 9 个，在 App 中，能够基于 FlutterNative 提供高性能的服务；在 PC 端和 Mobile 浏览器中，利用 FlutterWeb 做到了低成本适配，提升了产研的整体效率。

然而，加载性能问题是 MTFlutterWeb 应用推广的最大障碍。这里依然以美团外卖商家课堂业务为例，在项目之初页面完全加载时间 TP90 线达到了 6s 左右，距离我们的指标基线值（页面完全加载时间 TP90 线不高于 3s，基线值主要依据美团外卖商家端的业务场景、用户画像等来确定）有些差距，用户访问体验有很大的提升空间，

因此 FlutterWeb 页面加载性能优化，是我们亟需解决的问题。

二、挑战

不过，想要突破 FlutterWeb 页面加载的性能瓶颈，我们面临的挑战也是巨大的。这主要体现在 FlutterWeb 缺失静态资源的优化策略，以及复杂的架构设计和编译流程。下图展示了 Flutter 业务代码被转换成 Web 平台产物的流程，我们来具体进行分析：

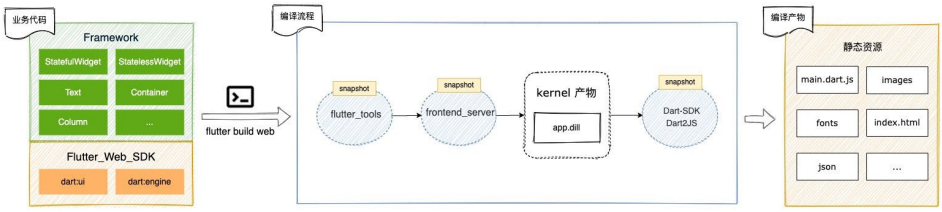


图 5　FlutterWeb 编译流程

- **Framework、Flutter_Web_SDK**（Flutter_Web_SDK 基于 HTML、Canvas，承载 HTML Render 模式的具体实现）等底层 SDK 是可被业务代码直接引入的，帮助我们快速开发出跨端应用；
- **flutter_tools** 是各平台（Android、iOS、Web）的编译入口，它接收 flutter build web 命令和参数并开始编译流程，同时等待处理结果回调，在回调中我们可对编译产物进行二次加工；
- **frontend_server** 负责将 Dart 转换为 AST，生成 kernel 中间产物 app.dill 文件（实际上各平台的编译过程都会生成这样的中间产物），并交由各平台 Compiler 进行转译；
- **Dart2JS Compiler** 是 Dart-SDK 中具体负责转译 JS 的模块，它将上述中间产物 app.dill 进行读取和解析，并注入 Math、List、Map 等 JS 工具方法，最终生产出 Web 平台所能执行的 JS 文件。
- **编译产物**主要为 main.dart.js、index.html、images 等静态资源，Flutter-Web 对这些静态资源缺少常规 Web 项目中的优化手段，例如：文件 Hash 化、文件分片、CDN 支持等。

可以看出，要完成对 FlutterWeb 编译产物的优化，就需要干预 FlutterWeb 的众多编译模块。而为了提升整体的编译效率，大部分模块都被提前编译成了 snapshot 文件（一种 Dart 的编译产物，可被 Dart VM 所运行，用于提升执行效率），例如：flutter_tools.snapshot、frontend_server.snapshot、dart2js.snapshot 等，这又加大了对 FlutterWeb 编译流程进行干预的难度。

三、整体设计

如前文所述，为了实现逻辑、渲染跨平台，Flutter 的架构设计及编译流程都具有一定的复杂性。但由于各平台（Android、iOS、Web）的具体实现是解耦的，因此我们的思路是定位各模块（Dart-SDK、Framework、Flutter_Web_SDK、flutter_tools）的 Web 平台实现并寻求优化，整体设计图如下所示：

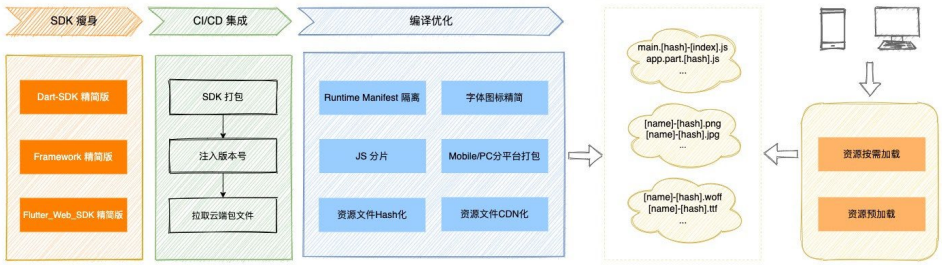


图 6　整体设计

- **SDK 瘦身**：我们分别对 FlutterWeb 所依赖的 Dart-SDK、Framework、Flutter_Web_SDK 进行了瘦身，并将这些精简版 SDK 集成合入 CI/CD（持续集成与部署）系统，为减小产物包体积奠定了基础；
- **编译优化**：此外，我们在 flutter_tools 中的编译流程做了干预，分别进行了 JS 文件分片、静态资源 Hash 化、资源文件上传 CDN 等优化，使得这些在常规 Web 应用中基础的性能优化手段得以在 FlutterWeb 中落地。同时加强了 FlutterWeb 特殊场景下的资源优化，如：字体图标精简、Runtime Manifest 隔离、Mobile/PC 分平台打包等；
- **加载优化**：在编译阶段进行静态资源优化后，我们在前端运行时，支持了资源

预加载与按需加载，通过设定合理的加载时机，从而减小初始代码体积，提升页面首屏的渲染速度。

下面，我们分别对各项优化进行详细的说明。

四、设计与实践

4.1 精简 SDK

4.1.1 包体积分析

工欲善其事，必先利其器，在开始做体积裁剪之前，我们需要一套类似于 web-pack-bundle-analyzer 的包体积分析工具，便于直观地比较各个模块的体积占比，为优化性能提供帮助。

Dart2JS 官方提供了 --dump-info 命令选项来分析 JS 产物，但其表现差强人意，它并不能很好地分析各个模块的体积占比。这里更推荐使用 source-map-explorer，它的原理是通过 sourcemap 文件进行反解，能清晰地反映出每个模块的占用大小，为 SDK 的精简提供了指引。下图展示了 FlutterWeb JS 产物的反解信息（截图仅包含 Framework 和 Flutter_Web_SDK）：

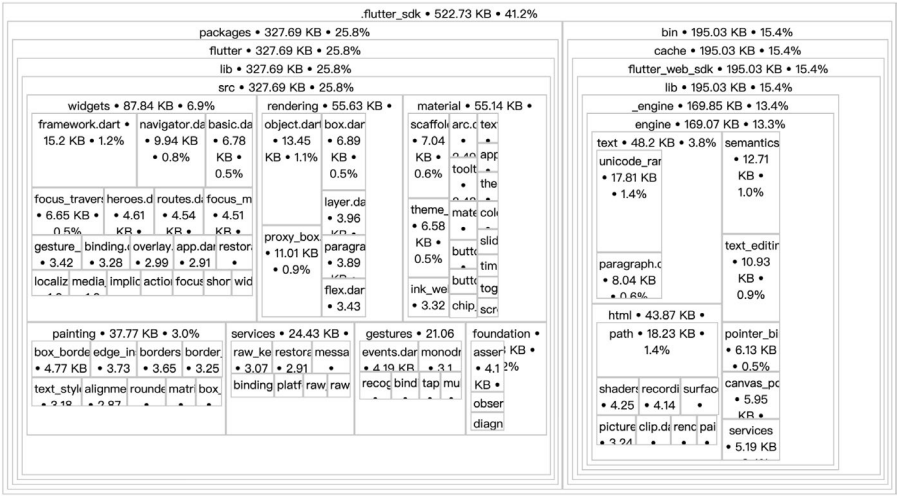


图 7　反解信息

4.1.2 SDK 裁剪

FlutterWeb 依赖的 SDK 主要包括 Dart-SDK、Framework 和 Flutter_Web_SDK，这些 SDK 对包体积的影响是巨大的，几乎贡献了初始化包的所有大小。虽然在 Release 模式下的编译流程中，Dart Compiler 会利用 Tree-Shaking 来剔除那些引入但未使用的 packages、classes、functions 等，很大程度上减少了包体积。但这些 SDK 中仍然存在一些能被进一步优化的代码。

以 Flutter Framework 为例，由于它是全平台公用的模块，因此不可避免地存在各平台的兼容逻辑（通常以 if-else、switch 等条件判断形式出现），而这部分代码是不能被 Tree-Shaking 剔除的，我们观察如下的代码：

```
// FileName: flutter/lib/src/rendering/editable.dart
void _handleKeyEvent(RawKeyEvent keyEvent) {
  if (kIsWeb) {
    // On web platform, we should ignore the key.
    return;
  }
  // Other codes ...
}
```

上述代码选自 Framework 中的 RenderEditable 类，当 kIsWeb 变量为真，表示当前应用运行在 Web 平台。受限于 Tree-Shaking 的机制原理，上述代码中，其它平台的兼容逻辑即注释 Other codes 的部分是无法被剔除的，但这部分代码，对 Web 平台来说却是 Dead Code（永远不可能被执行到的代码），是可以被进一步优化的。

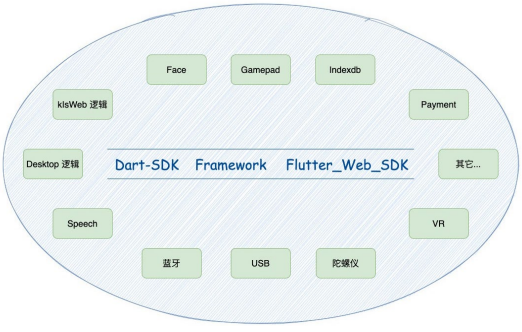


图 8 部分功能构成

上图展示了 SDK 的一部分功能构成，从图中可以看出，FlutterWeb 依赖的这些 SDK 中包含了一些使用频率较低的功能，例如：蓝牙、USB、WebRTC、陀螺仪等功能的支持。为此，我们提供了对这些长尾功能的定制能力（这些功能默认不开启，但业务可配置），将未被启用长尾的功能进行裁剪。

通过上述分析可得，我们的思路就是对 Dead Code 进行二次剔除，以及对这些长尾功能做裁剪。基于这样的思路，我们深入 Dart-SDK、Framework 和 Flutter_Web_SDK 各个击破，最终将 JS Bundle 产物体积由 1.2M 精简至 0.7M，为 FlutterWeb 页面性能优化打下了坚实的基础。

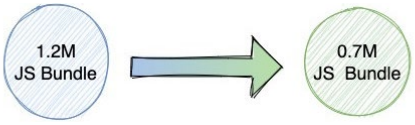


图 9　精简成果

4.1.3 SDK 集成 CI/CD

为了提升构建效率，我们将 FlutterWeb 依赖的环境定制为 Docker 镜像，集成入 CI/CD（持续集成与部署）系统。SDK 裁剪后，我们需要更新 Docker 镜像，整个过程耗时较长且不够灵活。因此，我们将 Dart-SDK、Framework、Flutter_Web_SDK 按版本打包传至云端，在编译开始前读取 CI/CD 环境变量：sdk_version（SDK 版本号），远程拉取相应版本的 SDK 包，并替换当前 Docker 环境中的对应模块，基于以此方案实现 SDK 的灵活发布，具体流程图如下图所示：

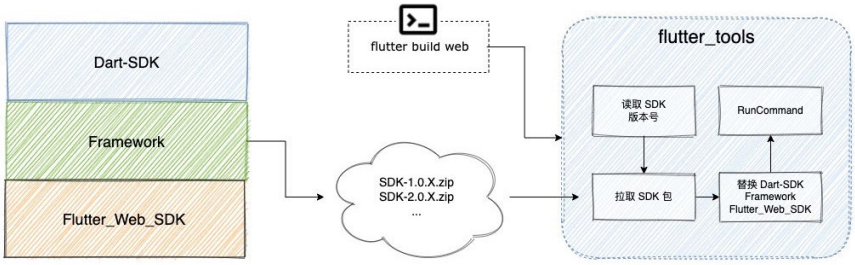


图 10　集成 CI/CD

4.2 JS 分片

FlutterWeb 编译之后默认会生成 main.dart.js 文件，它囊括了 SDK 代码以及业务逻辑，这样会引起以下问题：

- **功能无法及时更新**：为了实现浏览器的缓存优化，我们的项目开启了对静态资源的强缓存，若 main.dart.js 产物不支持 Hash 命名，可能导致程序代码不能被及时更新；
- **无法使用 CDN**：FlutterWeb 默认仅支持相对域名的资源加载方式，无法使用当前域名以外的 CDN 域名，导致无法享受 CDN 带来的优势；
- **首屏渲染性能不佳**：虽然我们进行了 SDK 瘦身，但 main.dart.js 文件依然维持在 0.7M 以上，单一文件加载、解析时间过长，势必会影响首屏的渲染时间。

针对文件 Hash 化和 CDN 加载的支持，我们在 flutter_tools 编译流程中对静态资源进行二次处理：遍历静态资源产物，增加文件 Hash（文件内容 MD5 值），并更新资源的引用；同时通过定制 Dart-SDK，修改了 main.dart.js、字体等静态资源的加载逻辑，使其支持 CDN 资源加载。

更详细的方案设计请参考《Flutter Web 在美团外卖的实践》一文。下面我们重点介绍 main.dart.js 分片相关的一些优化策略。

4.2.1 Lazy Loading

Flutter 官方提供 `deferred as` 关键字来实现 Widget 的懒加载，而 dart2js 在编译过程中可以将懒加载的 Widget 进行按需打包，这样的拆包机制叫做 Lazy Loading。借助 Lazy Loading，我们可以在路由表中使用 deferred 引入各个路由（页面），以此来达到业务代码拆离的目的，具体使用方法和效果如下所示：

```
// 使用方式
import 'pages/index/index.dart' deferred as IndexPageDefer;
{
  '/index': (context) => FutureBuilder(
    future: IndexPageDefer.loadLibrary(),
    builder: (context, snapshot) => IndexPageDefer.Demo(),
  )
  ... ...
}
```

Name	Status
main.dart.js	200
main.dart.js_3.part.js	200
main.dart.js_28.part.js	200
main.dart.js_10.part.js	200
main.dart.js_12.part.js	200
main.dart.js_66.part.js	200
main.dart.js_57.part.js	200
main.dart.js_59.part.js	200

图 11　效果演示

使用 Lazy Loading 后，业务页面的代码会被拆分到了多个 PartJS（对应图中 xxx.part.js 文件）中。这样看似解决了业务代码与 SDK 耦合的问题，但在实际操作过程中，我们发现每次业务代码的变动，仍然会导致编译后的 main.dart.js 随之发生变化（文件 Hash 值变化）。经过定位与跟踪，我们发现这个变化的部分是 PartJS 的加载逻辑和映射关系，我们称之为 Runtime Manifest。因此，需要设计一套方案对 Runtime Manifest 进行抽离，来保证业务代码的修改对 main.dart.js 的影响达到最低。

4.2.2 Runtime Manifest 抽离

通过对业务代码的抽离，此时 main.dart.js 文件的构成主要包含 SDK 和 Runtime Manifest：

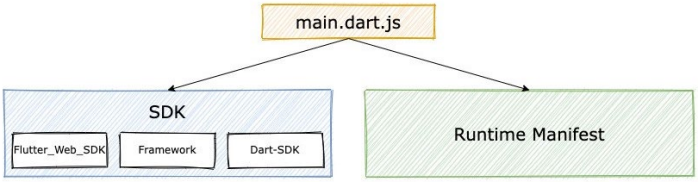


图 12　main.dart.js 构成

那如何能将 Runtime Manifest 进行抽离呢？对比常规 Web 项目，我们的处理方式是把 SDK、Utils、三方包等基础依赖，利用 Webpack、Rollup 等打包工具进行抽离并赋予一个稳定的 Hash 值。同时，将 Runtime Manifest（分片文件的加载

逻辑和映射关系）注入到 HTML 文件中，这样保证了业务代码的变动不会影响到公共包。借助常规 Web 项目的编译思路，我们深入分析了 FlutterWeb 中 Runtime Manifest 的生成逻辑和 PartJS 的加载逻辑，定制出如下的解决方案：

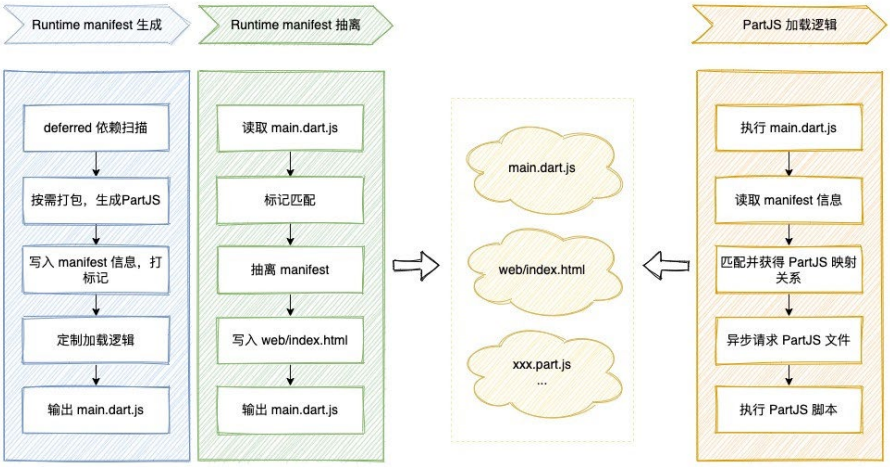


图 13　Runtime Manifest 抽离

在上图中，Runtime Manifest 的生成逻辑位于 Dart2JS Compiler 模块，在该生成逻辑中，我们对 Runtime Manifest 代码块进行了标记，之后在 flutter_tools 中将标记的 Runtime Manifest 代码块抽离并写入 HTML 文件中（以 JS 常量形式存在）。而在 PartJS 的加载流程中，我们将 manifest 信息的读取方式改为了 JS 常量的获取。按照这样的拆分方式，业务代码的变更只会改变 Runtime Manifest 信息，而不会影响到 main.dart.js 公共包。

4.2.3 main.dart.js 切片

经过以上引入 Lazy Loading、Runtime Manifest 抽离，main.dart.js 文件的体积稳定在 0.7M 左右，浏览器对大体积单文件的加载，会有很沉重的网络负担，所以我们设计了切片方案，充分地利用浏览器对多文件并行加载的特性，提升文件的加载效率。

具体实现方案为：将 main.dart.js 在 flutter_tools 编译过程拆分成多份纯文本文件，前端通过 XHR 的方式并行加载并按顺序拼接成 JavaScript 代码置于 < script > 标

签中，从而实现切片文件的并行加载。

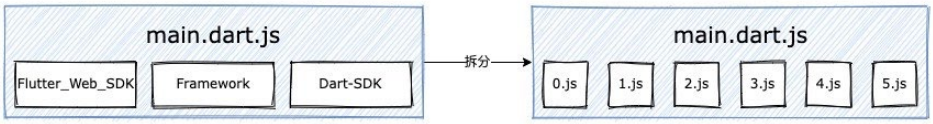


图 14　并行加载

4.3 预加载方案

如上一节所述，虽然我们做了很多工作来稳定 main.dart.js 的内容，但在 Flutter Tree-Shaking 的运行机制下，各个项目引用不同的 Framework Widget，就会导致每个项目生成的 main.dart.js 内容不一致。随着接入 FlutterWeb 的项目越来越多，每个业务的页面互访概率也越来越高，我们的期望是当访问 A 业务时，可以预先缓存 B 业务引用的 main.dart.js，这样当用户真正进入 B 业务时就可以节省加载资源的时间，下面为详细的技术方案。

4.3.1 技术方案

我们把整体的技术方案分为编译、监听、运行三个阶段。

- 编译阶段，在发布流水线上根据前期定制的匹配规则，筛选出符合条件的资源文件路径，生成云端 JSON 并上传；
- 监听阶段，在 DOMContentLoaded 之后，对网络资源、事件、DOM 变动进行监听，并对监听结果根据特定规则进行分析加权，得到一个首屏加载完成的状态标识；
- 运行阶段，在首屏加载完成之后对配置平台下发的云端 JSON 文件进行解析，对符合配置规则的资源进行 HTTP XHR 预加载，从而实现文件的预缓存功能。

下图为预缓存的整体方案设计：

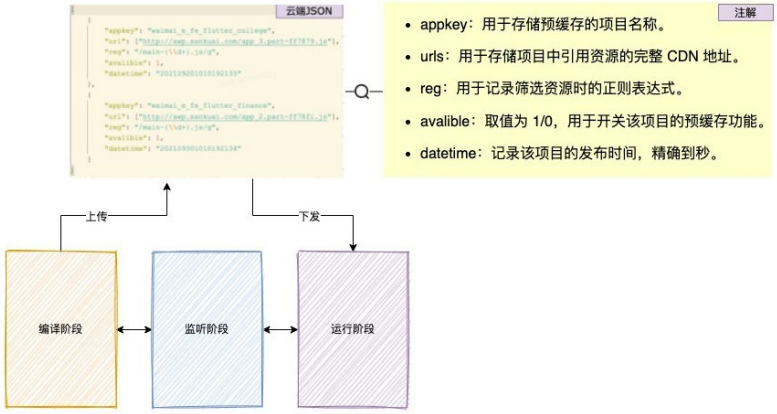


图 15　预缓存方案设计

编译阶段

编译阶段会扩展现有的发布流水线，在 flutter build 之后增加 prefetch build 作业，这样 build 之后就可以对产物目录进行遍历和筛选，得到我们所需资源进而生成云端 JSON，为运行阶段提供数据基础。下面的流程图为编译阶段的详细方案设计：

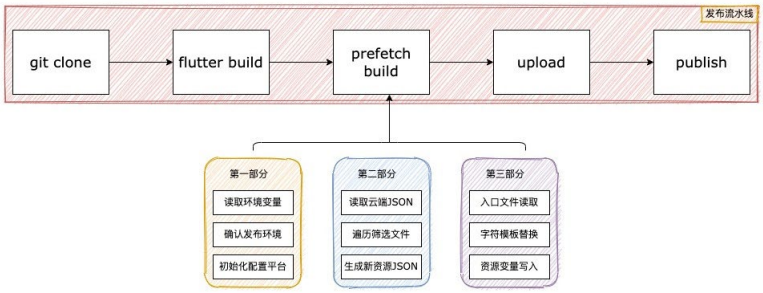


图 16　预缓存编译阶段

编译阶段分为三部分：

- 第一部分：根据不同的发布环境，初始化线上 / 线下的配置平台，为配置文件的读写做好准备；
- 第二部分：下载并解析配置平台下发的资源组 JSON，筛选出符合配置规则的资源路径，更新 JSON 文件并发布到配置平台；

- 第三部分：通过发布流水线提供的 API，把 PROJECT_ID、发布环境注入 HTML 文件中，为运行阶段提供全局变量以便读取。

通过对流水线编译期的整合，我们可以生成新的云端 JSON 并上传到云端，为运行阶段的下发提供数据基础。

监听阶段

我们知道，浏览器对文件请求的并发数量是有限制的，为了保证浏览器对当前页面的渲染处于高优先级，同时还能完成预缓存的功能，我们设计了一套对缓存文件的加载策略，在不影响当前页面加载的情况下，实现对缓存文件的加载操作。以下为详细的技术方案：

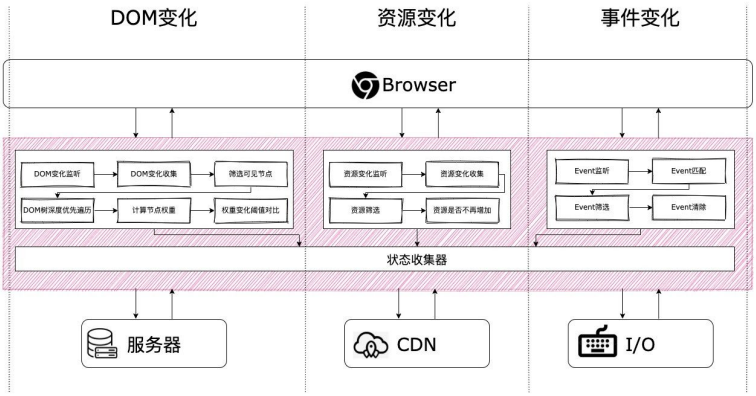


图 17　预缓存监听阶段

在页面 DOMContentLoaded 之后，我们会监听三部分的的变化。

- 第一部分是监听 DOM 的变化。这部分主要是在页面发生 Ajax 请求之后，随着 MV 模式的变动，DOM 也会随之发生变化。我们使用浏览器提供的 MutationObserver API 对 DOM 变化进行收集，并筛选有效节点进行深度优先遍历，计算每个 DOM 的递归权重值，低于阈值我们就认为首屏已加载完成。
- 第二部分是监听资源的变化。我们利用浏览提供的 PerformanceObserver API，筛选出 img/script 类型的资源，在 3 秒内收集的资源没有增加时，我们

认为首屏已加载完成。

- 第三部分是监听 Event 事件。当用户发生 click、wheel、touchmove 等交互行为时，我们就认为当前页面处于一个可交互的状态，即首屏加载已完成，这样会在后续进行资源的预缓存。

通过上述步骤，我们就可以得到一个首屏渲染完成的时机，之后就可以实现预缓存功能了。以下为预缓存功能的实现。

运行阶段

预缓存的整体流程为：下载编译阶段生成的云端 JSON，解析出需要进行预缓存资源的 CDN 路径，最后通过 HTTP XHR 进行缓存资源进行请求，利用浏览器本身的缓存策略，把其他业务的资源文件写入。当用户访问已命中缓存的页面时，资源已被提前加载，这样可以有效地减少首屏的加载时间。下图为运行阶段的详细方案设计：

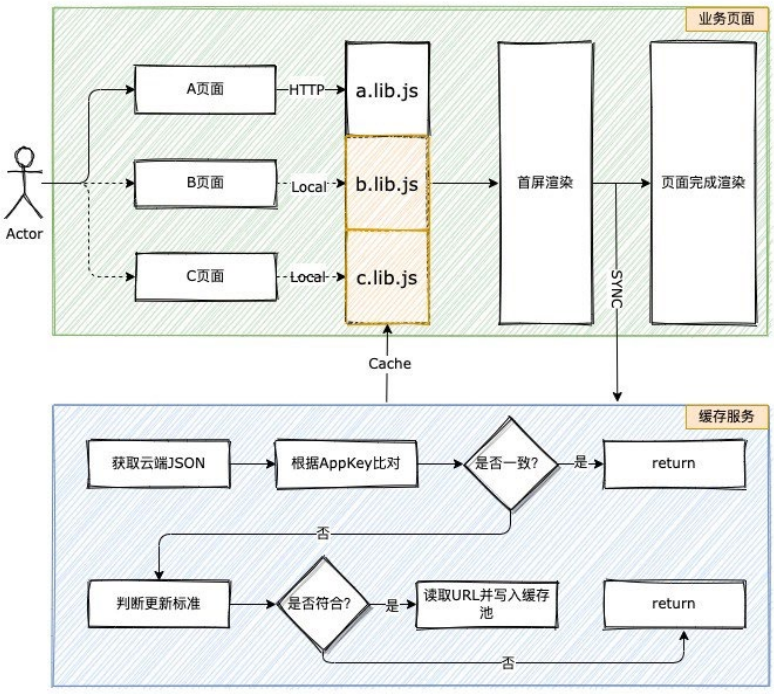


图 18　预缓存运行阶段

在监听阶段，我们可以获取到页面的首屏渲染完成的时机，会获取到云端 JSON，首先判断该项目的缓存是否为启用状态。当该项目可用时，会根据全局变量 PROJECT_ID 进行资源数组的匹配，再以 HTTP XHR 方式进行预访问，把缓存文件写入浏览器缓存池中。至此，资源预缓存已执行完毕。

4.3.2 效果展示与数据对比

当有页面间互访问命中预缓存时，浏览器会以 200（Disk Cache）的方式返回数据，这样就节省了大量资源加载的时间，下图为命中缓存后资源加载情况：

0.js	awp-assets.meituan.net	(disk cache)	7 ms
1.js	awp-assets.meituan.net	(disk cache)	11 ms
2.js	awp-assets.meituan.net	(disk cache)	17 ms
3.js	awp-assets.meituan.net	(disk cache)	16 ms
4.js	awp-assets.meituan.net	(disk cache)	16 ms
5.js	awp-assets.meituan.net	(disk cache)	15 ms

图 19　预缓存效果展示

目前，美团外卖商家端业务已有 10+ 个页面接入了预缓存功能，资源加载 90 线平均值由 400ms 下降到 350ms，降低了 12.5%；50 线平均值由 114ms 下降到 100ms，降低了 12%。随着项目接入接入越来越多，预缓存的效果也会越发的明显。

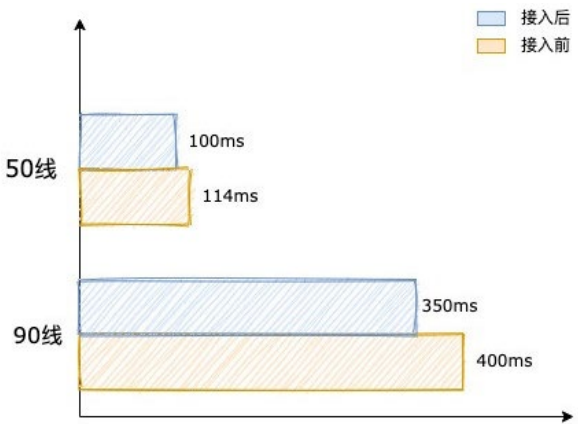


图 20　预缓存数据展示

4.4 分平台打包

如前文所述，美团外卖商家业务大部分都是双端对齐的。为了实现提效的最大化，我们对 FlutterWeb 的多平台适配能力进行加强，实现了 FlutterWeb 在 PC 侧的复用。

在 PC 适配过程中，我们不可避免地需要书写双端的兼容代码，如：为了实现在列表页面中对卡片组件的复用。为此我们开发了一个适配工具 ResponsiveSystem，分别传入 PC 和 App 的各端实现，内部会区分平台完成适配：

```
// ResponsiveSystem 使用举例
Container(
  child: ResponsiveSystem(
    app: AppWidget(),
    pc: PCWidget(),
  ),
)
```

上述代码能较方便的实现 PC 和 App 适配，但 AppWidget 或 PCWidget 在编译过程中都将无法被 Tree-Shaking 去除，因此会影响包体积大小。对此，我们将编译流程进行优化，设计分平台打包方案：

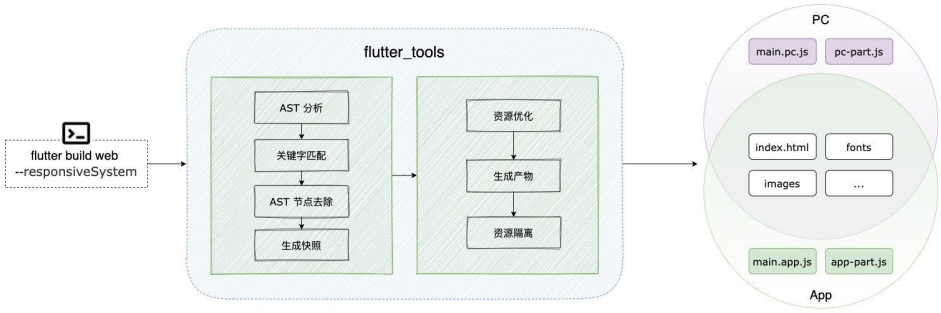


图 21　分平台打包

- 修改 flutter-cli，使其支持 –responsiveSystem 命令行参数；
- 我们在 flutter_tools 中的 AST 分析阶段增加了额外的处理：ResponsiveSystem 关键字的匹配，同时结合编译平台（PC 或 Mobile）来进行 AST 节点的改写；

- 去除无用 AST 节点后，生成各个平台的代码快照（每份快照仅包含单独平台代码）；
- 根据代码快照编译生成 PC 和 App 两套 JS 产物，并进行资源隔离。而对于 images、fonts 等公用资源，我们将其打入 common 目录。

通过这样的方式，我们去除了各自平台的无用代码，避免了 PC 适配过程中引起的包体积问题。依然以美团外卖商家课堂业务（6 个页面）为例，接入分平台打包后，单平台代码体积减小 100KB 左右。

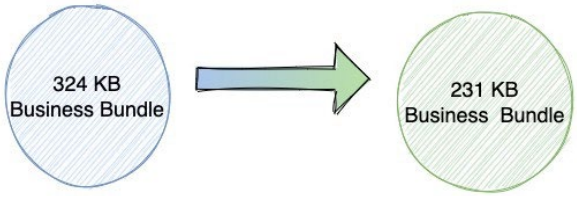


图 22　效果展示

4.5 图标字体精简

当访问 FlutterWeb 页面时，即使在业务代码中并未使用 Icon 图标，也会加载一个 920KB 的图标字体文件：MaterialIcons-Regular.woff。通过探究，我们发现是 Flutter Framework 中一些系统 UI 组件（如：CalendarDatePicker、PaginatedDataTable、PopupMenuButton 等）使用到了 Icon 图标导致，且 Flutter 为了便于开发者使用，提供了全量的 Icon 图标字体文件。

Flutter 官方提供的 `--tree-shake-icons` 命令选项是将业务使用到的 Icon 与 Flutter 内部维护的一个缩小版字体文件（大约 690KB）进行合并，能一定程度上减小字体文件大小。而我们需要的是只打包业务使用的 Icon，所以我们对官方 `tree-shake-icons` 进行了优化，设计了 Icon 的按需打包方案：

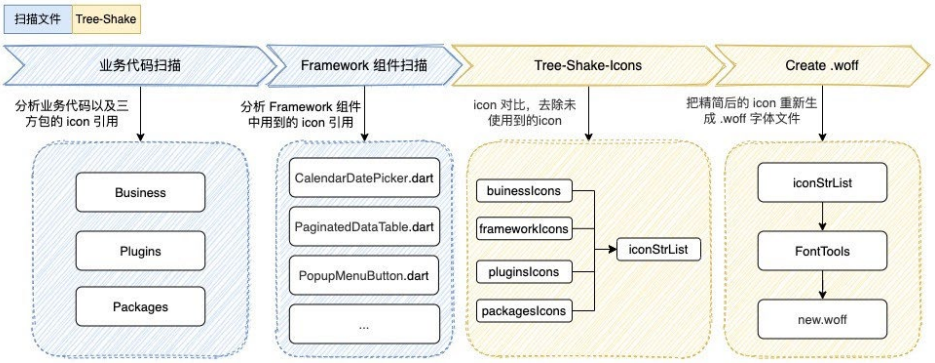


图 23　图标字体精简

- 扫描全部业务代码以及依赖的 Plugins、Packages、Flutter Framework，分析出所有用到的 Icon；
- 把扫描到的所有 Icon 与 material/icons.dart（该文件包含 Flutter Icon 的 unicode 编码集合）进行对比，得到精简后的图标编码列表：iconStrList；
- 使用 FontTools 工具把 iconStrList 生成字体文件 .woff，此时的字体文件仅包含真正使用到的 Icon。

通过以上的方案，我们解决了字体文件过大带来的包体积问题，以美团外卖课堂业务（业务代码中使用了 5 个 Icon）为例，字体文件从 920KB 精简为 11.6kB。

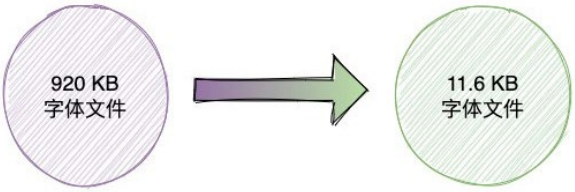


图 24　效果展示

五、总结与展望

综上所述，我们基于 HTML Render 模式对 FlutterWeb 性能优化进行了探索和实践，主要包括 SDK（Dart-SDK、Framework、Flutter_Web_SDK）的精简，静

态资源产物优化（例如：JS 分片、文件 Hash、字体图标文件精简、分平台打包等）和前端资源加载优化（预加载与按需请求）。**最终使得 JS 产物由 1.2M 减少至 0.7M（非业务代码），页面完全加载时间 TP90 线由 6s 降到了 3s**，这样的结果已能满足美团外卖商家端的大部分业务要求。而未来的规划将聚焦于以下 3 个方向：

- **降低 Web 端适配成本**：目前已有 9+ 个业务借助 MTFlutterWeb 实现多端复用，但在 Web 侧（尤其是 PC 侧）的适配效率依然有优化空间，目标是将适配成本降低到 10% 以下（目前大约是 20% ）；
- **构建 FlutterWeb 容灾体系**：Flutter 动态化包有一定的加载失败概率，而 FlutterWeb 作为兜底方案，能提升整体业务的加载成功率。此外 FlutterWeb 可以提供“免安装更新”的能力，降低 FlutterNative 老旧历史版本的维护成本；
- **性能优化的持续推进**：性能优化的阶段性成果为 MTFlutterWeb 的应用推广巩固了基础，但依然是有进一步优化空间的，例如：目前我们仅将业务代码和 Runtime Manifest 进行了拆离，而 Framework 及 三方包在一定程度上也影响到了浏览器缓存的命中率，将这部分代码进行抽离，可进一步提升页面加载性能。

美团外卖技术团队正在基于 FlutterWeb 做更多的探索和尝试。如果你对这方面的技术也比较感兴趣，可以在文末留言，跟我们一起讨论。也欢迎大家给提出一些建议，非常感谢。

设计稿（UI 视图）自动生成代码方案的探索

作者：田贝　少宽　腾飞

1. 背景

设计稿（UI 视图）转代码是前端工程师日常不断重复的工作，这部分工作复杂度较低但工作占比较高，所以提升设计稿转代码的效率一直是前端工程师追求的方向之一。此前，前端工程师尝试过将业务组件模块化构建成通用视图库，并通过拖拽、拼接等形式搭建业务模块，从而实现视图复用，降低设计稿转代码的研发成本。但随着业务的发展和个性化的驱动，通用视图库无法覆盖所有应用场景，本文提出了一种设计稿自动生成代码的方案。

目前，业内主流的代码生成方案有两种，一种是通过训练神经网络，从图片或草图直接生成代码，以微软 sketch2json 为代表；另一种是基于 Sketch 源文件，从中解析出图层信息转化成 DSL 并生成代码，以 imgCook 为代表。经过实践，我们发现第一种方案基于神经网络的代码生成算法虽然简单粗暴，但复杂层布局的准确率较低、可解释程度不高导致后续无法持续优化。方案二中 Sketch 源文件信息量丰富、算法自定义程度高、优化空间大。因此，我们调研了业界基于 Sketch 的代码自动生成方案（已对外公布或者开源），发现了一些不足并尝试解决，下面从算法准确率、代码可读性、研发流程覆盖度等方面做一下对比（**该对比结果仅考察业界方案对我们自己业务的适用性，实际结果可能存在差异**）：

算法准确率			
比较维度	组件识别	布局类型	结论
imgCook	支持进度条、标签等复杂组件识别	支持行/列/包含/悬浮/循环，不支持成组	准确率中
58 Picasso	仅识别基础组件	支持行/列/包含，不支持成组/悬浮/循环	准确率低
我们的方案	仅识别基础组件	支持行/列/包含/成组/悬浮/循环	准确率中

代码可读性		
比较维度	是否使用绝对布局	结论
imgCook	绝对布局使用较多，不符合预期	可读性一般
58 Picasso	绝对布局使用较多，不符合预期	可读性一般
我们的方案	相对布局，符合RD预期	可读性好

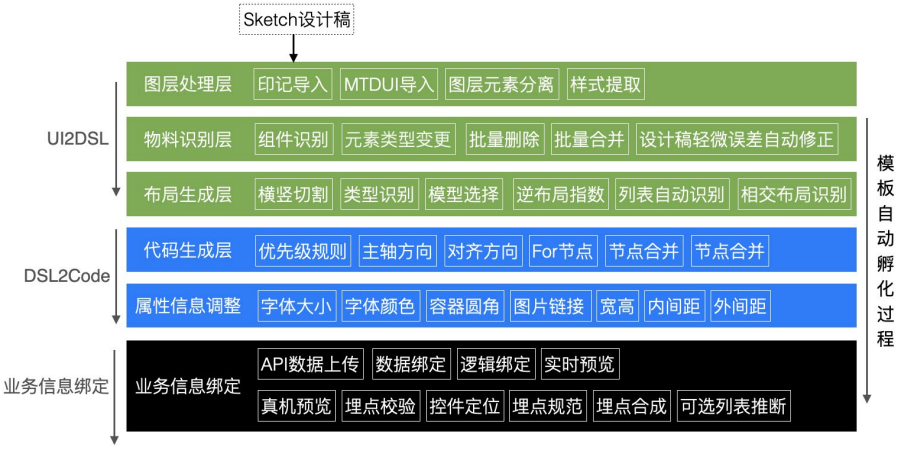
研发流程覆盖度				
比较维度	数据绑定	逻辑绑定	埋点绑定	结论
imgCook	智能化/可视化	智能化/可视化	可视化	覆盖度广
58 Picasso	不支持	不支持	不支持	覆盖度窄
我们的方案	可视化	可视化	可视化	覆盖度广

- **算法准确率方面**：淘宝 imgCook 支持基于 AI 的组件识别，不支持成组布局，准确率中等（从官网了解到可以识别循环布局，但不能识别出测试样本中的循环布局），58 Picasso 仅支持原始组件的识别，复杂组件生成错误较多，不支持成组 / 悬浮 / 循环布局，准确率较低。
- **代码可读性方面**：淘宝 imgCook 在生成布局时，测试样本中图层重叠区域使用到了基于根布局的绝对定位方式，不符合 RD 预期，可读性一般，而我们的方案使用相对定位方式，可读性较好。

- **研发流程覆盖度方面**：淘宝 imgCook 从 RD 视角构建了一个 IDE，支持在 IDE 中完成样式调整、逻辑绑定；而我们的方案从产研协作视角出发，支持数据、逻辑、埋点的可视化配置及上线。

2. 方案介绍

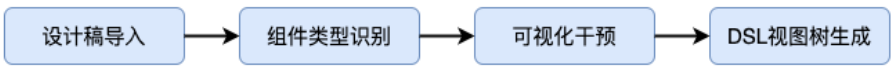
如图所示，配置平台主要分成三块包括：设计稿转视图树（UI2DSL）、视图树转代码（DSL2Code）、以及业务信息绑定，下面简单介绍一下每一块的作用。



- **设计稿转 DSL 视图树（UI2DSL）**：将设计稿转化成平台无关的 DSL 视图树。
- **视图树转代码（DSL2Code）**：将 DSL 视图树转化成基于 Flex 布局的 MTFlexBox 静态代码。
- **业务信息绑定**：提供可视化配置工具，支持 MTFlexBox 静态代码绑定后台数据、业务逻辑、以及曝光 / 点击等埋点逻辑。

2.1 设计稿转视图树（UI2DSL）

UI2DSL 主要经历以下四个步骤：



2.1.1 设计稿导入

在日常开发过程中，我们接触比较多的组件有按钮、标题、进度条、评分组件等，但是 Sketch 数据源中并没有这些组件只有图层信息，图层是设计师在设计 UI 视图时用到的视图控件。组件与图层的对应关系是一对多的关系，图层在 Sketch 数据源中的表现形式如下图中的 JSON 数据结构所示，描述了图层的坐标、大小等信息，后续布局生成就是基于对图层的切割来实现的。




```
[
    {
        "class_name":"MSTextLayer",
        "font_face":"PingFangSC-Medium",
        "font_size":13.44,
        "height":36.5,
        "index":8,
        "line_height":18.24,
        "name":" 恒都民生精选猪小排带骨 400g±25g",
        "object_id":"EF55F482-A690-4EC2-8A6E-6E7D2C6A9D91",
        "opacity":0.9000000357627869,
        "text":" 恒都民生精选猪小排带骨 400g±25g",
        "text_align":"left",
        "text_color":"#FF000000",
        "type":"text",
        "width":171.8,
        "x":164.2,
        "y":726.7
    },
    //......
]
```

2.1.2 组件识别

从上面的数据源可以看出，图层有图片、文字、矩形等基本类型，在组件识别这一步图层需要被转化成文字 / 图片 / 进度条 / 评分组件 / 价格组件 / 角标等日常开发使用的组件类型。但是目前我们的进展还停留在只能将图层识别为文字或者图片的阶段，后续我们将接入淘宝开源的 pipcook 框架，基于神经网络算法进行更加丰富的组件类型识别。

2.1.3 可视化干预

设计稿作为输入源是设计稿自动转代码的基础，这对设计稿的设计规范要求较高。但在实践中，我们发现设计师会利用 Sketch 中的基本图形（每个图形最终形成数据源中的一个图层）叠加来描述一个组件的视觉效果，因此设计稿中不可避免会出现冗余图层的问题，干扰 DSL 的生成。虽然我们也尝试了利用自动化的手段删除冗余图层，但对于算法不能识别的部分（例如：图片上有一个文本图层，但是实际情况中文本是显示在图片里的，这个时候无法从算法层面决定是否删除文本），仍然需要靠人工进行图层删除、合并等，否则无法正常生成 DSL。设计稿主要有以下几类问题。

图层未合并



上图是从设计稿解析出来的结果，可以发现在“美团优选”文字上方的图片中有很多红色的矩形框（每个矩形框是一个单独的图层），而算法预期的输入是一个图层，因此需要在算法处理前将多个图层合并成一个图层，右侧的三张图也有类似问题。我们与设计同学进行过沟通，设计同学表示愿意在产出设计稿之前将图层进行合并，但由于目前无法提供检测机制（图层合并是否有遗漏无法自动检测出来），也就无法彻底避免图层未合并的问题。

图层位置交叉



实践中发现当设计稿中不同字体 / 大小 / 颜色的文字排列在一起时，解析出来的图层信息往往会出现重叠的情况，由于 DSL 视图树算法依赖位置来确定不同组件的约束关系，因此位置的交叉会对算法准确度造成较大的影响。

复杂背景图层



上图中红色背景是由 2 个图层（2 个蓝色矩形框）拼接形成的，左图上的蓝色图层是纯色，右图上的蓝色图层是渐变色，在两个图层未合并的情况下，算法生成的代码将会出错。

上面提出的问题，通过约束设计师来达到设计稿的规范化，难度较大，所以我们提供了可视化干预工具。下面对上述问题做一个简单的总结：– 问题一：图层未合并问题肉眼很**容易识别**出来，利用工具将冗余图层进行**快速合并删除**即可。 – 问题二：图层交叉问题肉眼不易识别，因此我们提供了**检测工具**，基于检测工具可以对设计稿中的交叉问题**快速修复**。 – 问题三：复杂背景问题肉眼不易识别，暂时也没有有效的检测工具，用户可以采用**边干预边生成**的方式生成 DSL。

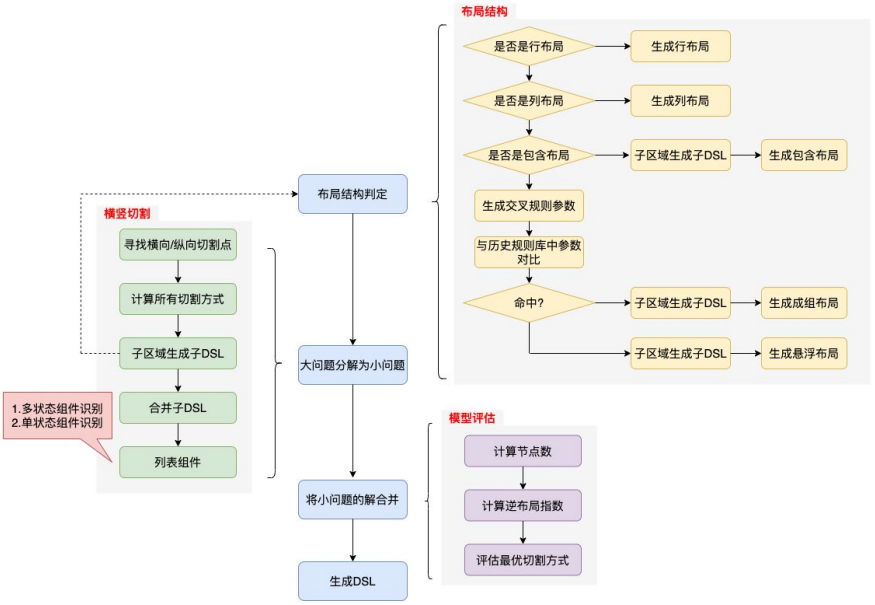


可视化干预是 UI2DSL 重要的一环，**经过可视化干预，将不标准的设计稿转化为标准的图层信息后再输入给算法，可以极大地提升算法的准确率**。这里我们和 imgCook 的处理方式有一个区别：imgCook 在引入了阈值处理等算法后（更智能，出错概率更大），可视化干预能力主要体现在事后，而我们在生成 DSL 之前允许用户对图层进行干预，在干预时用户面对的是直观的图层信息，可以有效降低工具的使用门槛（更稳定，效果更好）。

2.1.4 视图树生成

将扁平的数据源转化为树状结构的 DSL，这个过程如果是人脑来做会怎么思考呢？先确定布局的整体结构是行布局或者列布局，然后再确定局部区域应该是什么布局结构，最后组装起来形成视图树。这个过程与递归算法类似，因此我们采用了递归算法

作为算法的主框架，同时引入了“横竖切割 + 布局结构 + 模型评估”三大利器。



利器一：横竖切割

生成 DSL 时采用了整分的思路，即将大布局不断的切分成小布局，下面以动画的形式看一下简化过的 DSL 生成过程：



将设计稿一部分区域视为一个子区域，最开始的时候子区域和整个模板的面积一样大，基于图层的位置、大小信息，计算每个图层的上 / 下 / 左 / 右边缘坐标与其他图

层的相对关系，就可以寻找到切割点（如上图中红色箭头所指的位置）。接下来依据切割点，将子区域切割成更小的子区域，在切割的过程中如果切割点是横向的，则生成列布局；如果切割点是纵向的，则生成行布局。通过不断的切割子区域得到更小的子区域，直到所有的子区域只剩下图片或者文本等不可切割的图层，这样就可以生成完整的 DSL 视图树了。为了方便读者理解，图例中只演示了行布局、列布局的切分过程，实际情况还包含了其他布局类型，会要复杂许多。

这里还要注意一个问题，当有 3 个切割点时，我们选择了直接将子区域切割成 4 个子区域，实际上我们可以只选择 1 个切割点进行切割，也可以选择 2 个切割点进行切割，当有 N 个切割点时，实际上存在（N 的阶乘 +1）种切割方式，具体选择哪种切割方式，我们会在利器三中讨论。

利器二：布局结构

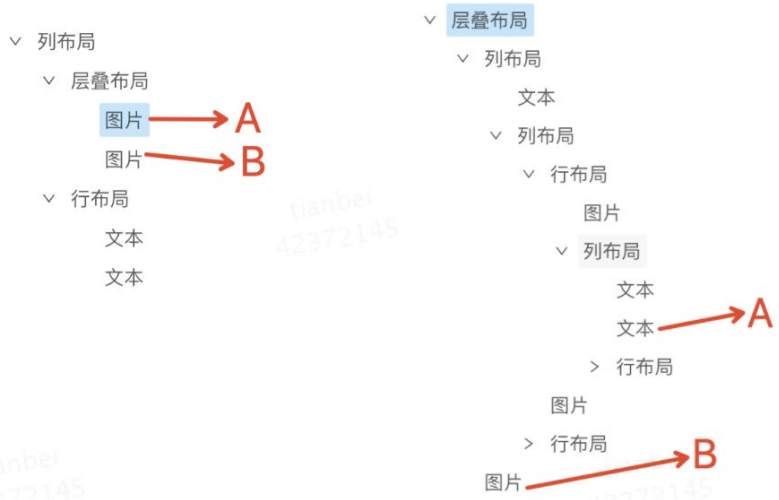
每个图层都是一个矩形，为了生成布局结构只能依赖矩形的上下左右坐标信息。因此，对布局结构进行分类时，我们根据矩形与矩形之间的位置关系（相交、相离和包含关系）做了以下分类。



注意：从生成 DSL 的结果来看，**包含布局**和**成组布局**的处理方式其实是一样的，都是使用类似于 FrameLayout 的层叠布局包含内部图层元素，但是我们仍然保持分类原则（矩形之间的位置关系）不变。

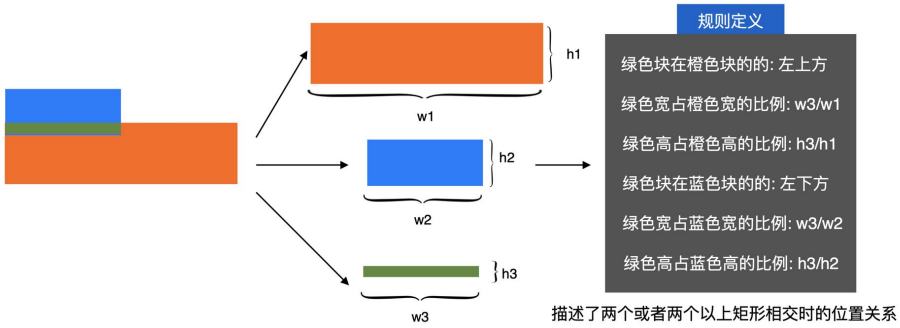
上图中，相离、包含比较好理解，为什么两个图层相交的时候，会有成组和悬浮两种类型的布局结构呢？我们看下上述成组布局、悬浮布局两个设计稿中分别标出了相交的元素 A、B，它们在位置上的相对关系是一样的，都是 A、B 两个图层对应的矩形框发生了交叉。但是我们希望理想态的 DSL 视图树却有所差异，如下图所示：

- 成组布局中：A、B 逻辑上是一个整体，交叉是必然的，最终 DSL 中 A、B 被层叠布局包含，层叠布局中没有其他元素。
- 悬浮布局中：A、B 逻辑上不是整体，只是碰巧交叉了，最终 DSL 中 A、B 分别在不同的层级中。



因此，对于图层相交时可能有两种类型的布局结构，分别是成组布局和悬浮布局。从上图可以看出使用成组布局还是悬浮布局是由图层内容决定的，那么就需要算法理解图层内容了，比如基于 AI 构建样本库，记住所有的角标样式（上面表格中 4 描述的），下次遇到角标相交时就生成成组布局。考虑到 AI 模型也是对规则的抽象，我们先搭建一套自定义识别规则。成组布局其位置信息是有规律可循的，例如：角标经常出现

在右上角，标签经常出现在左上角，头像经常横向或者纵向交叉等，因此我们针对图层之间的位置关系构建了交叉模型，如下图所示：



描述了两个或者两个以上矩形相交时的位置关系

上图的交叉模型可以记住历史模板中成组布局图层之间的位置关系，下次遇到相交布局时判断是否在历史规则库中即可完成识别，如果在就按成组布局处理否则按照悬浮布局处理。下图是通过历史模板构建的成组规则库。

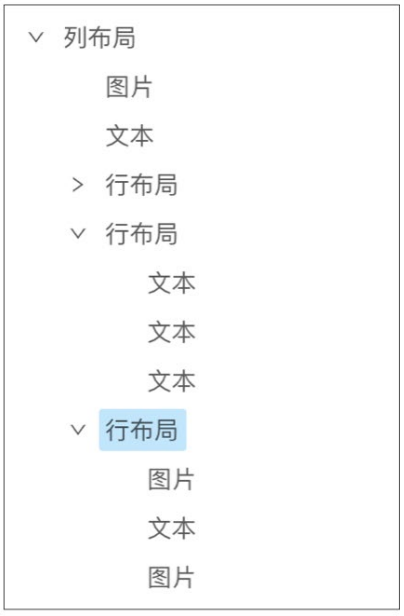


上面介绍了本方案中涉及的 5 种布局类型，目前来看这五种布局类型可以描述所有的模板布局，并且生成代码符合 RD 的预期。下面展示两个设计稿 DSL 实例：









利器三：模型评估

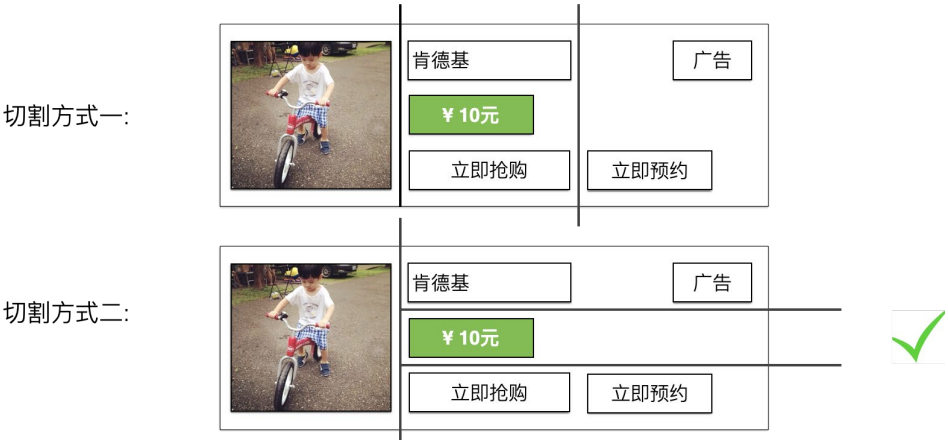
在介绍横竖切割时，可以看到当存在多个切割点时，对所有切割点同时进行了切割，但实际上算法在切割时复杂度会更高，当有三个切割点时，实际上有 5 种切割方式，每种切割方式都会生成一个 DSL。既然有 5 种切割方式，那么到底应该选择哪一种

DSL 呢？模型评估算法就是用来解决这个问题的。

目前模型评估算法有两个指标：布局节点数和逆布局指数。

- DSL 中布局节点数越少，切割方式越好。
- 逆布局指数用来评估 DSL 中的行列布局的合理程度，其中逆布局指数越大越不合理，反之，逆布局指数越小，切割方式越好。

以下图为例，看下视图不同切割方式下对应的模型评估方式：



如果模型评估算法只衡量布局节点数的话，那么会选择第一种切割方式生成的 DSL 作为最终的结果。但实际上，第二种切割方式更加合理。在切割方式一中，广告、立即预约处于一个列布局中，但是横向对齐方式（交叉轴）却不一样，“广告”是右对齐，“立即预约”是左对齐，逆布局指数表示交叉轴对齐方式不一致的节点数量，因此通过逆布局指数，我们可以规避掉不合理的切割方式。

2.1.5 列表布局

上一节介绍了基本的布局结构，虽然说这些布局结构已经可以描述所有的 UI 布局，但是与 RD 的编码习惯还是有一些差异。



对于上面的布局，RD 通常不会把相同的 item 写五遍，而是会将 item 放在一个类似于 ListView 的列表组件中，使代码看起来简洁易懂。因此在 DSL 生成阶段，除了识别基本的行 / 列 / 包含 / 成组 / 悬浮布局外，还需要进一步识别行 / 列布局中的元素是否形成列表布局。在试验过程中，我们发现列表布局分为两种：**单状态列表组件**和**多状态列表组件**。上图中每一个 item 的布局结构都是一样的，我们称为单状态列表组件，再来看一下多状态列表组件（如下图所示），每个 item 有多种状态（选中态和非选中态），并且不同状态的布局结构不一致。



对行 / 列布局中单状态列表组件的识别，只需要比较 item 子视图树的结构，子视图树结构一致则判断为单状态列表组件。对多状态列表组件的识别我们采取了自动识别 + 人工干预的方式，自动识别的方式比较粗暴，只要行列布局中子 item 的宽 / 高接近，并且子 item 不是基本组件（基本组件容易形成误判），就判定为多状态列表组件。具体算法是计算子 item 宽高的标准差，小于阈值就判定为多状态列表组件，否则不是。公式如下：

$$(\sqrt{(w_1-\bar{w})^2+(w_2-\bar{w})^2+\ldots+(w_n-\bar{w})^2} < \text{阈值}_w) \quad \wedge \quad (\sqrt{(h_1-\bar{h})^2+(h_2-\bar{h})^2+\ldots+(h_n-\bar{h})^2} < \text{阈值}_h)$$

那为什么还要人工干预呢？因为是否使用列表组件其实与产品逻辑相关，但是目前我们无法将产品文档中的逻辑识别出来，只能尽可能识别出所有的多状态列表组件，并允许用户对生成结果进行变更。比如上述送恋人的设计稿，产品可能约定每一个 item

都有选中态 / 非选中态两种状态，也可能是从业务角度考虑需要着重突出送恋人这个 item，这时每个 item 就只有一种确定的状态，这两种不同的产品逻辑在编写代码时有不同的最优技术方案。

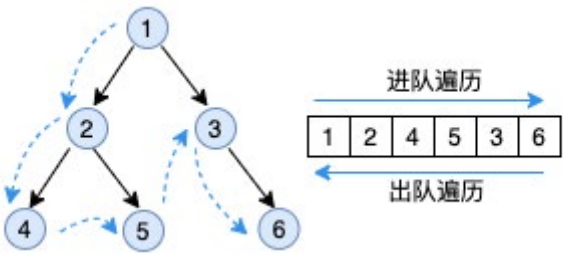
2.2 视图树转代码（DSL2Code）

DSL 视图树只是生成代码的中间产物，还需要对 DSL 进行代码还原，DSL2Code 主要包括两个步骤：属性推断、属性信息调整。

2.2.1 属性推断

属性推断包括两个部分：样式属性和结构属性。样式属性包括字体、背景色、圆角等可以直接通过数据源信息中获取得到的属性；结构属性包括大小、内外边距、主辅轴对齐等结构信息，这些信息无法从数据源中直接获取，所以结构信息的推断是这部分工作的重点。

结构信息推断算法同样使用递归算法作为主框架，通过一次递归对所有元素进行两次遍历来完成结构信息的推断。如下图所示，在对 DSL 所有节点进行递归遍历时，把所有元素依次加入队列中，递归完成后，再把所有节点依次移出队列，这样一进一出便对所有元素完成了两次遍历，我们把这两次遍历称为进队遍历和出队遍历。

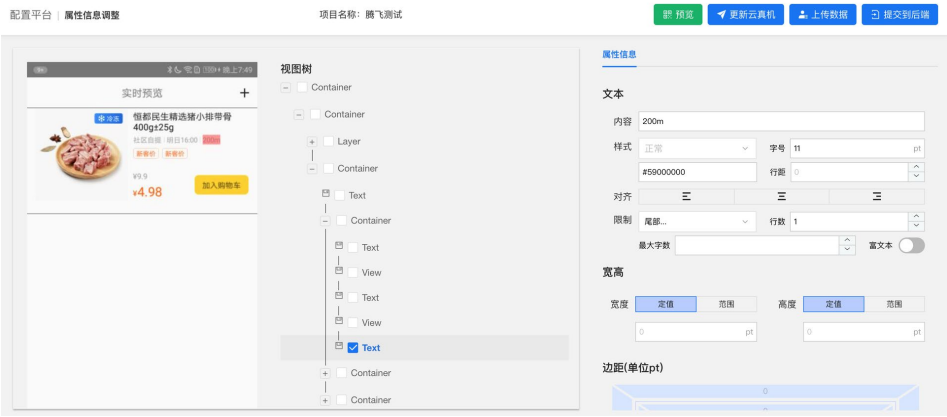


进队遍历时，推断算法根据数据源中信息记录每个节点的大小和位置信息，并根据位置关系计算每个子节点在父节点中期望的主辅轴对齐方式和内外边距。出队遍历时，父节点会根据子节点期望的对齐方式确定父节点最终的主辅轴对齐方式，并根据子节点的拉伸意图修正父节点的大小。拉伸意图即节点的大小不固定，根据显示内容不

同，在水平或垂直方向上可能会变大或变小，例如文本节点根据显示字数的多少长度会发生变化，字数过多时甚至还会换行。

2.2.2 属性信息调整

由于输入源是基于设计稿呈现的静态效果图，设计稿中每个元素缺失了真实的业务含义，同样的展示效果在不同的业务场景中会有不同的属性要求，对于这部分内容，我们无法从输入源中进行准确推断。为此，我们提供了可视化的属性信息调整功能来辅助代码生成，页面效果如下图所示，在这个页面可以对 DSL 中的所有节点属性进行查看和修改调整。



经过业务信息补充后，便可进行最后的自动代码转化，通过语法映射自动把 DSL 转化成 MTFlexbox 模板代码。

3 成果展示

下面是设计稿直接生成代码未经修改展示后的手机屏幕截图，可以看到取得了不错的还原效果：


今日必抢
爆款大牌
每天上新
¥4.9 3.6折
喜茶·多肉草莓芝士凉茶
500m
¥4.9 3.6折
喜茶·多肉草莓芝士凉茶
500m
¥4.9 3.6折
喜茶·多肉草莓芝士凉茶
500m
冷冻
恒都民生精选猪小排带骨
400g±25g
社区自提 | 明日16:00 | 200m
新客价 | 限2份
新品
¥9.9
¥4.98
加入购物车
限时秒杀 03:24:42
¥5.9
喜茶·满杯莫吉托
周一特卖 9.9封顶
¥6.5
¥6.5
周一特卖 9.9封顶
¥6.5
¥6.5
美团优选
超低价
限时秒
¥1.88 ¥4.5
限时秒
¥1.88 ¥4.5
限时秒
¥1.88 ¥4.5
优选 天天1元抢秒杀 · 次日达
水果蔬菜 零食蛋奶
一站购齐
水果蔬菜
¥1.88
水果蔬菜
¥1.88
水果蔬菜
¥1.88
水果蔬菜
¥1.88
爆款推荐
限量抢购
¥1.88
¥1.88
¥1.88
¥1.88

以上就是我们近期对代码自动生成的探索及实践，后续我们将引入机器学习及神经网络算法，对过程进行进一步优化。如果您有其他的看法或建议，也欢迎在文末进行评论，或者跟我们联系。

作者简介

田贝、少宽、腾飞等，美团平台终端业务研发团队研发工程师。

团队简介

美团平台终端业务研发团队的职责是保障平台业务高效、稳定迭代的同时，持续优化用户体验和研发效率。团队负责的业务主要包括美团首页等千万级 DAU 高频业务以及分享、账号、音 / 视频等基础业务，支撑和对接外卖、酒店等 30 多个业务方。团队通过动态化能力建设，加快业务上线速度，帮助产品（PM）快速验证业务选型，做出业务决策；架构 / 服务标准化体系建设，提升前后端以及平台与业务线的沟通、合作效率；业务监控和体验优化，有效保障核心业务服务成功率的同时，提升用户使用美团 App 过程中的稳定性和流畅性。团队开发技术栈包括 Android、iOS、React Native、Flexbox 等。

招聘信息

美团平台终端业务研发团队是一个活力四射、对技术充满激情的团队，现诚聘 Android、iOS、FE 工程师，欢迎有兴趣的同学投简历至 tech@meituan.com（备注：美团平台终端业务研发团队）。

美团外卖终端容器无关化研发框架

作者：正浩　宝石　彭震

| 终端容器无关化（Containerless）：与服务无关化（Serverless）的概念类似，即在保持顶层业务研发语言不变更的情况下，在下层可以兼容性地升级、替换终端容器的能力，让用户无需关心终端容器的运维，只要将精力聚焦到业务逻辑上的技术。

一、前言

React2X 是一款面向多终端、跨平台、容器无关化研发框架。在整个美团前端技术栈日益规范的趋势下，React 技术栈在我们技术体系环节中的地位变得越来越重要。在广告、营销这些推广属性的业务上，在各个终端（包括美团 App、美团外卖 App、大众点评 App，以及站外的微信小程序、百度小程序、头条 & 抖音小程序等其他终端）实现“一次开发，同步需求上线”的业务诉求也变得越来越多。在这样的背景下，我们定义了 React2X 应用的核心场景：

- 面对美团内部丰富多样的技术容器体系（Mach、MRN、Titans、MTFlutter、MMP 等），如何保证跨容器开发体验的**一致性**，以及建设跨容器应用开发的**生态能力**，是我们需要解决的问题。
- 公司内丰富的终端容器化技术蓬勃发展，而因业务升级带来的改造成本也比较大，亟待一款**高扩展性**设计的顶层框架作为技术抓手。
- 跨容器动态化能力覆盖，逐步成为各个业务方越来越重视的基础能力，可以大幅缩短需求交付的周期，提高上线发版的效率，并能有效地解决包体积大小的问题，提升业务的**敏捷性**。
- 多场景下的**同构诉求**，例如在各种推广页、模块化、游戏、轻量布局差异的 PC/App 同构场景下，可以节省多端研发的人力。

最终我们的**核心痛点**围绕在了**美团系 · 小程序**和**美团系 · App 矩阵上的**同一个需求的多次开发运维上，为了**解决研发人力瓶颈问题**，我们需要一款“**一次研发，多终端**

容器复用”的研发框架来**提升研发效率**。

调研整个前端领域，我们找到了一些业界的解决方案，像是美团最早的 mpvue、腾讯的 Wepy、滴滴的 Chameleon、京东的 Taro 等等。在经过比较与试用之后，我们最终基于投入产出比的价值判断，选择站在巨人的肩膀上研发定制一款满足美团技术、业务场景的研发框架——**React2X**（后面简称 **R2X**）。从 **R2X** 第一个版本发布到现在，已经接受了来自于公司各个业务两年多的考验。所以我们希望通过本文帮助大家对 **R2X** 有一个大致的了解。

二、目标与场景

2.1 核心目标

为了解决业务需求在多端容器需要重复开发的难题，通过代码复用实现开发提效，我们确定了以下的目标：

- **解决公司内部多终端容器开发痛点**：实现 Webview 容器、小程序容器、MRN 容器、Mach 容器、游戏容器、部分运营推广场景 PC 容器的代码同构复用，统一开发规范，抹平开发差异，并提供对其他容器的扩展能力。
- **建设跨容器动态化能力**：跨容器动态化能力的缺失，导致产品不能够通过快速迭代来验证需求的效果，这个问题严重限制了业务的发展。跨容器动态化能力可以解决美团外卖业务端上发版和包体的问题，帮助业务实现快速发版上线、线上问题热修复、以及容灾的能力。
- **建设容器无关的开发生态体系**：R2X 最终要解决的是容器差异性，进行统一的技术生态能力建设，为多终端容器开发场景提升生产和运维效率。

2.2 应用场景

R2X 开发框架主要期望能最终面向多终端应用的终端容器，用于场景化研发：

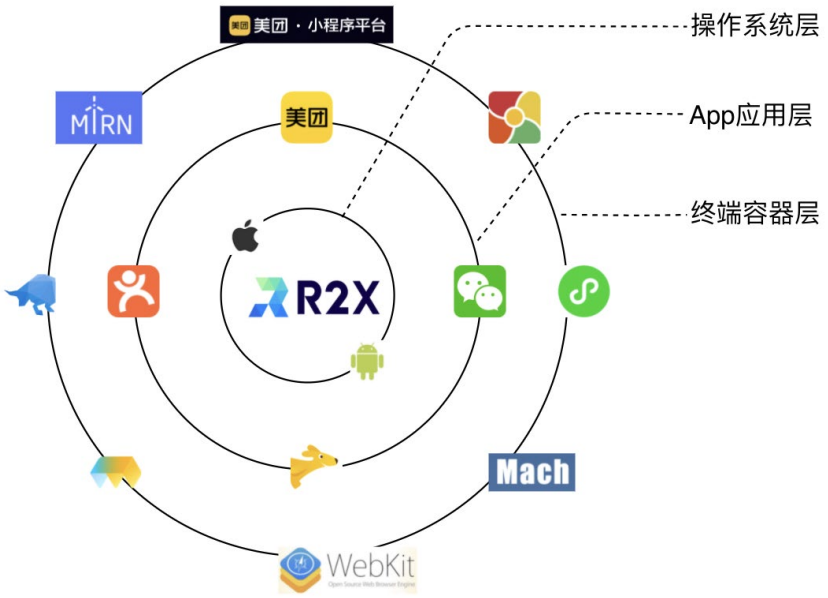


图 1

即：

- 业务项目基于 React 语法为技术框架基础。
- 业务方有在多终端 / 多容器（包括 MRN 容器、Webview 容器、MP 容器、Flutter 容器、Mach 容器、PC 浏览器容器）运行的需求。
- 业务方有特定的场景化诉求，包括推广页、模块化、小游戏、PC/App 同构等等。

三、挑战与优势

3.1 业界调研

针对上述核心目标和应用场景，我们对市面上的跨容器框架进行了调研。由于美团外卖的技术栈统一是 React 为主，所以我们的必备要求是：一款以 React 为 DSL 语言的复用框架，能快速融入美团的技术生态。

根据下表的对比，如果以 React 为 DSL 语言出发，当时就只有 Taro 一家能满足我们的业务诉求，但它的生态环境并不适合在美团体系内使用。基于多方面因素的考

虑，我们决定结合各大主流框架之所长，然后开发出一款属于美团外卖的跨容器复用框架。

对比项	mpvue	Taro 1.3	Chameleon	WePY	UniApp
DSL	Vue	类 React（Nerv）	类 Vue	Vue	Vue
是否支持 React Native	否	是，但支持效果不佳	Weex	否	否
兼容 API	无	有（API 支持程度不一）	自研多态协议	无	是
跨端组件库	无	有	有	无	无
美团生态	有	无	无	无	无
语法校验	无	ESLint	自研	无	有
TypeScript	有	有	无	有	有
定制化扩展	无	可自研 Plugin	无	无	有
编译扩展	无	无	无	无	有
调研结论	不匹配	部分满足	部分满足	不匹配	不匹配

注：前期调研时间截止到 2019 年 05 月，可能与当前数据存在一定的出入。

3.2 技术挑战

当我们决定要打造一款属于美团外卖的跨容器复用框架之后，在实现的过程中主要遇到了以下挑战：

① **各个容器之间差异性适配成本** – 语法语义：MRN/ 小程序 /Webview 在 DSL 上就有着完全不同的语法语义。 – 端能力：同一容器在不同端上表现也存在不少差异，比如外卖 App 中 MRN 容器和美团 App 中 MRN 容器分别有定制的 Native 模块以及各类桥协议等。

② **业务接入的使用成本** – 首次成本：作为一个新定义的框架如何让新业务方快速上手，如何从存量业务线进行迁移。 – 边际成本：如何融合美团的基建生态，让业务方快速复用平台能力。

③ **顶层架构的合理设计** – 高可维护性、高度扩展性：如何快速升级、替换、新增一款底层容器。

注：以上问题我们会在下文“技术全景”章节中给予解答。

3.3 项目优势

3.3.1 功能特点对比

目前，业界以小程序作为跨端目标平台的框架较多，但大多都是使用 Web 前端技术栈作为基础，但同时兼顾 React Native 的技术栈就比较少。下表中列出的是支持以 React、类 React 作为 DSL 的相关框架进行对比。

	R2X	Taro 1.3	Taro 3.0	Rax	Remax
原理	R2X 1.0 重编译时，R2X 2.0 重运行时	重编译时	重运行时	重运行时	重运行时
容器重点	以 MRN，小程序，WebView 为主，同时支持 MTFlutter、Mach、游戏、PC 浏览器	以小程序、Web 为主，React Native 支持不多	以小程序、Web 为主，React Native 交给 58 团队支持	小程序、Web、Flutter	小程序、Web
API	支持 KNB 桥 & 对多平台 API 进行了统一	对多平台 API 进行了统一	对多平台 API 进行了统一	多平台不统一	多平台不统一
跨端组件库	有	有 TaroUI，但不支持 React Native	有 TaroUI，但不支持 React Native	无	无
业务组件扩展	提供扩展方案	参考 TaroUI	参考 TaroUI	提供扩展方案	提供扩展方案
美团内部生态支持	已支持埋点监控等	无	无	无	无
模块化能力	支持	不支持	不支持	不支持	不支持
编译插件扩展	支持	不支持	支持	支持	支持 Webpack 配置

综上所述，从目前的业务形态来看，R2X 在容器的匹配程度以及美团生态支持程度上看，是现阶段的最佳方案。R2X 相较于业界其他框架来说，拥有更加完善的适用于美团外卖团队的本地化实现。

3.3.2 性能数据对比

基于业界跨平台框架和美团内部的跨平台框架，我们针对性能也进行了 Benchmark 测试，最终对比结果如下。

小程序性能对比

框架	创建	更新	添加	交换	删除
R2X-MP	947.6	586.8	1467.2	1355.2	82.2
Remax	2798.2	1872.6	5162.2	4818.2	86.4
Taro-MP	1653.4	976.4	2483.2	2256.6	65.2

结论：可以看到，在小程序 Benchmark 测试结果中，R2X-MP 是领先于 Remax 和 Taro-MP。

与 React Native 性能对比

框架	创建	更新	添加	交换	删除
R2X-MRN	309.875	83.75	384	191.875	82.125
MRN	297.625	105.25	400.125	231.625	65.875
Taro-RN	209.5	77.5	246.25	85.125	17.125

结论：在 React Native 的 Benchmark 测试结果中，R2X-MRN 和 MRN 基本持平，但都低于纯 React Native 的性能表现。

3.3.3 同构场景对比

除了支持了基本的 React Native、小程序和 Webview 容器同构场景之外，R2X 还实现了在 MTFlutter、Mach、小游戏（Webview 游戏、微信小游戏 & 小程序、美团小游戏）、PC 浏览器等容器上的同构能力扩展，相比于业内的其他跨容器开发框架的生态也更加丰富和健全。

	React Native	小程序	Webview	Flutter	模块级容器	小游戏容器	PC浏览器（PC/App同构）
R2X	支持	支持	支持	支持	支持	支持	支持
Taro	支持	支持	支持	不支持	不支持	不支持	不支持
Rax	支持	支持	支持	不支持	不支持	不支持	不支持
Remax	支持	支持	支持	不支持	不支持	不支持	不支持

四、技术全景

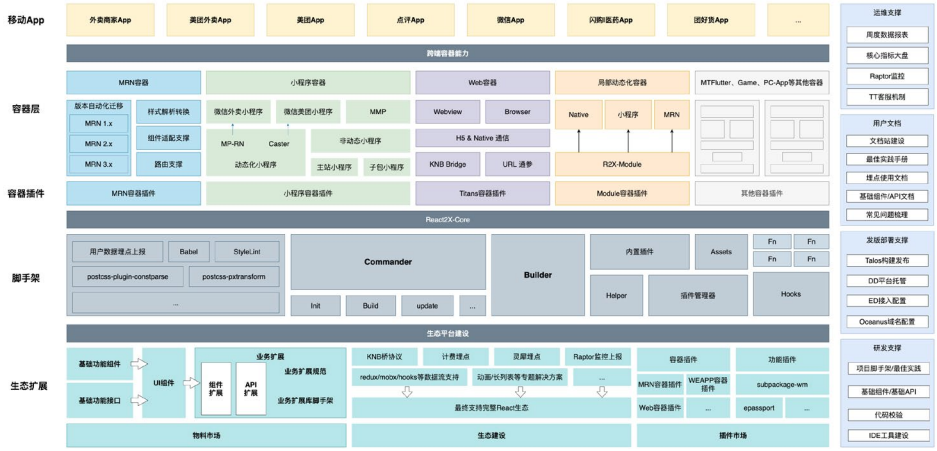


图 2

上图为 R2X 的架构全景图，整体架构可以按照从下到上，从左到右的视角进行解读：

- 最下层是 R2X 的生态环境建设，在 R2X 内部去实现公司生态的常用 SDK 以及业务中的各项专题能力；并通过搭建物料市场 / 插件市场，以业务共建的形式丰富 R2X 生态。
- 再上层是 R2XCore 的根基，通过解析 Command 命令来执行唤起构建器，并实现了类似 Webpack 的插件系统，以插件化的形式组织驱动整个核心构建流程，便于维护以及扩展。
- 再往上是跨端容器层，它是整个跨端能力的核心，通过实现了不同的容器插件来将 R2X 代码编译成各端可执行代码，并通过运行时能力对组件 /API 进行对齐。
- 最上层是承载的 App 端，目前有美团外卖、大众点评、美团等多款移动 App 终端。
- 最右边是 R2X 在研发、发布、运维以及用户文档上做的一些建设。

因为 R2X 覆盖了美团内部大部分的主流容器场景，所以技术体系较为复杂和庞大，大家可以根据自身的业务形态，选择性地去了解对应场景的同构方案。

4.1 底层基础框架

4.1.1 R2X-CLI 的设计

CLI 作为 R2X 项目驱动器，它不仅包含了命令行的启动，更重要的，它是各个编译构建的核心。

在早期，CLI 执行 build 命令时，我们通过 –type 来区分不同的构建容器，从而加载不同的编译逻辑。通过指定构建容器的形式来实现同一套代码能够构建出不同的容器产物。但经过长时间的业务迭代，我们发现了这种结构存在的问题：

- 整体流程冗长且复杂，长时间迭代会变得越来越难以维护。
- 每个容器的构建流程相互独立，且构建逻辑各不一致，有多处重复的逻辑处理。
- 编译流程缺少统一的关键节点，编译时无法进行业务方的定制扩展。

针对以上问题，我们考虑对 CLI 进行了一次全新的重构，引入插件化能力（关于插件化能力的具体实现会在下文详细描述）。CLI 整体结构变成如下图所示：

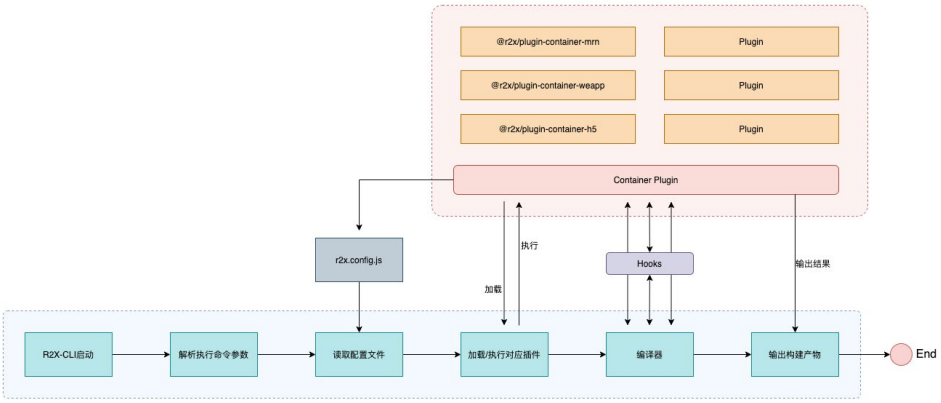


图 3

整个 CLI 模块只需要关心参数的解析以及插件的加载执行，不需要再实现各个容器的具体编译逻辑。通过 Hooks 的形式，将编译的各个时机暴露给插件，插件基于这些 Hook 进行编译能力的实现，最终输出产物给 CLI 模块。这种形式带来了以下几个好处：

- CLI 结构变得清晰，只需要维护配置解析、插件解析等功能。
- 扩展性增强，可通过插件化的形式新增或删减容器 / 编译能力，保证代码独立维护功能的单一性。
- 编译流程可梳理，无论什么容器的编译流程都基于编译器暴露的时机执行并串联，整体流程清晰明了。

4.1.2 组件及 API 的设计

R2X 的目的是希望通过一套代码能够在多端上运行，但是由于多端差异的存在，我们需要设计一套统一的标准规范来进行对齐。在运行时部分，主要分为组件 / 接口的对齐。

多端差异

在开始讲述实现之前，我们先来看看各端之间的差异到底在哪些地方。

组件（标签）差异

- Webview 标签采用的是 XML 写法，所提供的基础标签有 <a/> 等等。
- 小程序采用的是 WXML（WeiXin Markup Language）标签语言，也提供了一套完整的基础标签，但是和 Webview 有着较大的差异。
- React Native 则是采用的 JSX（JS-XML）语法，虽然和 XML 很接近，但是又有着很多的不同点，同时它也有自己的一套基础组件，和 Webview、小程序又截然不同。

API 差异

- **接口差异**：在不同端中都提供了相同或近似的功能，但是其实现方式以及调用参数可能存在着很大的差异，比如数据缓存 Storage，小程序中用 wx.setStorage/wx.setStorageSync，Webview 则 用 localStorage.setItem， 而 MRN 中 AsyncStorage.setItem，几乎每一个功能点都有着多多少少的差异。
- **容器差异**：各个端所提供的 API 都是各自的容器量身打造的，比如小程序的各业务接口类 API，完全是针对小程序所处的微信环境打造的，这类功能与其他

端并不相关。

- **能力差异**：各个端之间的差异我们可以通过定制的手段来适配，然而并不是所有的功能点在各个端上都能够实现。比如在 Webview 中就无法做到像小程序、React Native 中提供很多原生能力，像是文件保存读取等等，这一类差异性在适配过程中都属于不可抗拒、不可抹平的差异。

样式差异

小程序的 WXSS 和 Webview 的 CSS 在参数属性上其实是几乎一致的，但是在层级关系上有着很大的差别，小程序分为全局样式与局部样式，各个组件之间的样式也是不会相互影响（默认配置下）。而对比 React Native 采用的 StyleSheet，是用 Inline Style 的方式，不支持全局样式，不支持标签样式，并且属性有诸多限制，如只能使用 Flex 布局等等。

如何适配

从以上章节我们已经了解到了各个端之间有着非常大的差异点，那我们应该如何克服这些困难呢?

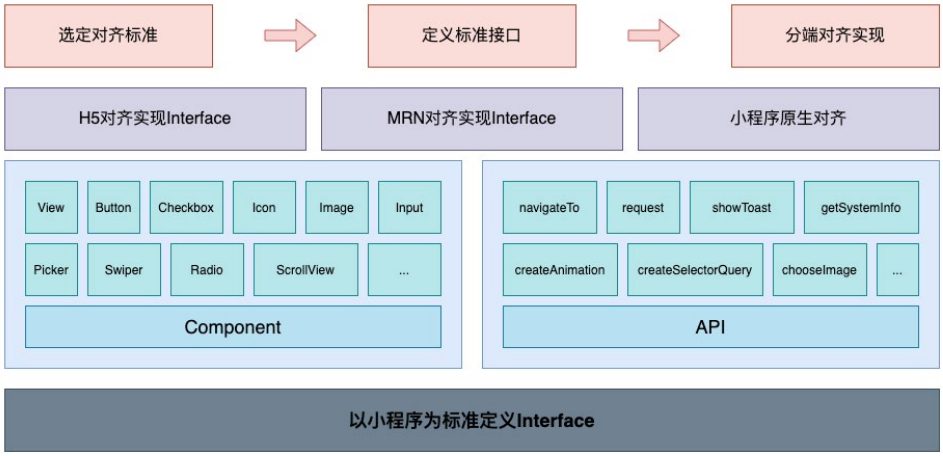


图 4

由于各端对组件和 API 的支持程度不同，我们选定了一端为基础标准，定义好各个组

件的属性以及接口参数，通过 TypeScript 的 Interface 进行实现。然后在各个端分别基于以上的接口进行功能对齐实现，对于端能力限制的功能进行了一定取舍，对高优功能进行了 SDK 底层实现适配。最终，我们基于已有的功能封装实现了一套完整的基础组件 `@r2x/components` 和基础 API`@r2x/r2x`。

4.1.3 开放式插件能力

随着 R2X 的在美团内部的应用越来越多，大家对于 R2X 模式的认可度也在不断提高，我们从业务方中经常听到以下这些问题："是否可以增加支持某某功能 / 容器"，"我们业务架构比较特殊，能否做出一些调整"。业务方对 R2X 会有更多功能 / 容器的诉求，也会有更多定制化的需求出现。

所以，我们决定实现一套完整的开放式插件能力，提供一种相对比较简单的方式，让大家能够自己来定制这些特殊需求。在最新的版本中，我们将 R2X 的编译时进行了重构，在新的编译时架构中引入了基于 Tapable 的插件系统。开发者可以通过编写插件的方式为 R2X 拓展更多功能，或者为自身业务线定制更多的个性化功能。

在插件类型分为两类：

- **容器插件**，用于封装 R2X 所支持的容器的核心编译能力。
- **功能插件**，基于已有的容器插件，在此基础上进行某种特定功能的自定义实现。

插件能力的整体架构如下：

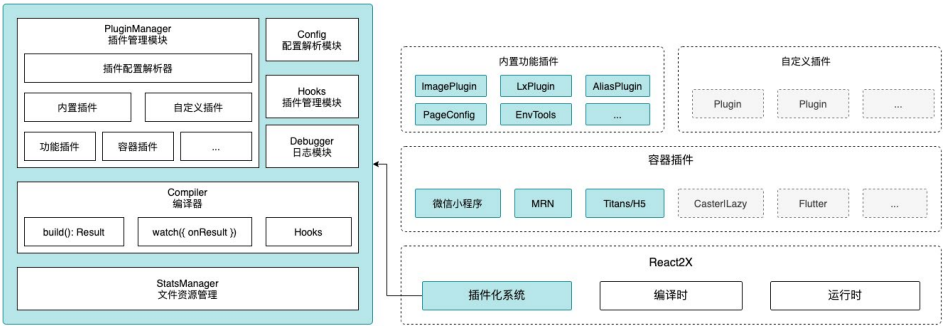


图 5

借助开发式插件能力，我们将之前编写了若干个平台容器插件，开发者安装后即可使用：

- 小程序容器插件：@r2x/plugin-container-wxapp。
- MRN 容器插件：@r2x/plugin-container-mrn。
- Titans 容器插件：@r2x/plugin-container-h5。

除了扩展新的容器平台，我们还可以通过继承现有的容器插件，来编写一些特殊的定制化功能插件。

1. 对代码进行预处理

基于开放式插件能力，我们可以像 Babel 插件一样，通过对 AST 语法的修改对代码源文件进行编译前后的修改。比如：修改文件引用路径、插入代码片段、处理本地图片等等。

2. 对文件产物进行修改

在编译产出生成时，我们可以对编译文件的内容、文件路径、文件结构进行修改。结合自身业务的定制化，CLI 可以将 R2X 项目和现有的原生项目进行结合改造。

除了以上功能，插件化能力为用户在编译时提供了极大的自由度。如果你想体验的话，欢迎加入美团外卖技术团队。

4.1.4 特性能力 - 多态能力

为什么需要多态能力?

多态能力是用于提供跨端时各端组件及 API 的统一解决方案。基于多态能力，开发者可以定制自己的跨端组件。而 R2X 具备了完善的跨端能力，能够覆盖多终端和容器，为什么还需要多态?

业务研发为了满足各自场景的要求需要一定的灵活性。同时，Webview/ 小程序 / React Native 容器存在端上的差异，需要开发者人为进行环境判断。逻辑一复杂、跨端数量一多，代码可读性变低、维护成本起飞，这不是我们的本意。

基于这样的背景，R2X 提供了扩展性良好的多态能力。

R2X 多态能力介绍

对于多态能力的支持，我们分为两类：

- 多态组件 /API，R2X 根据文件后缀区分编译目标端。

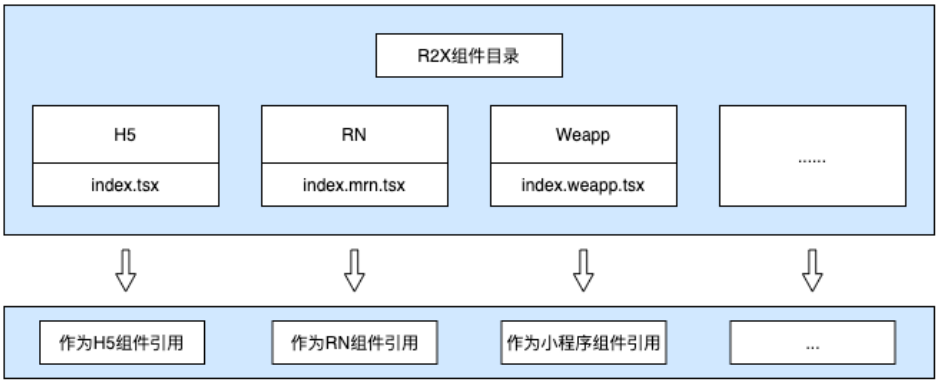


图 6

- 差异化代码，R2X 提供 getEnv 环境方法用于判断当前语句编译目标端类型。

```
<View style={{color: R2X.getEnv() === "WEAPP" ? "green" : "blue" }} />
```

通过差异化代码可轻松满足端差异诉求。

4.2 应用场景同构

4.2.1 页面级容器场景同构

页面级容器场景同构我们借鉴了业内 Taro1.x、Rax 等优秀跨端框架的做法，结合美团内部容器、基建特点做了很多本地化实现和定制，在 Webview、MRN、小程序三种容器上通过不同的编译时方案进行预处理，并且引入了 React 运行时，保证了对 React DSL 支持的完整程度和代码同构率，编译时转化流程方案和运行时结构如下图所示：

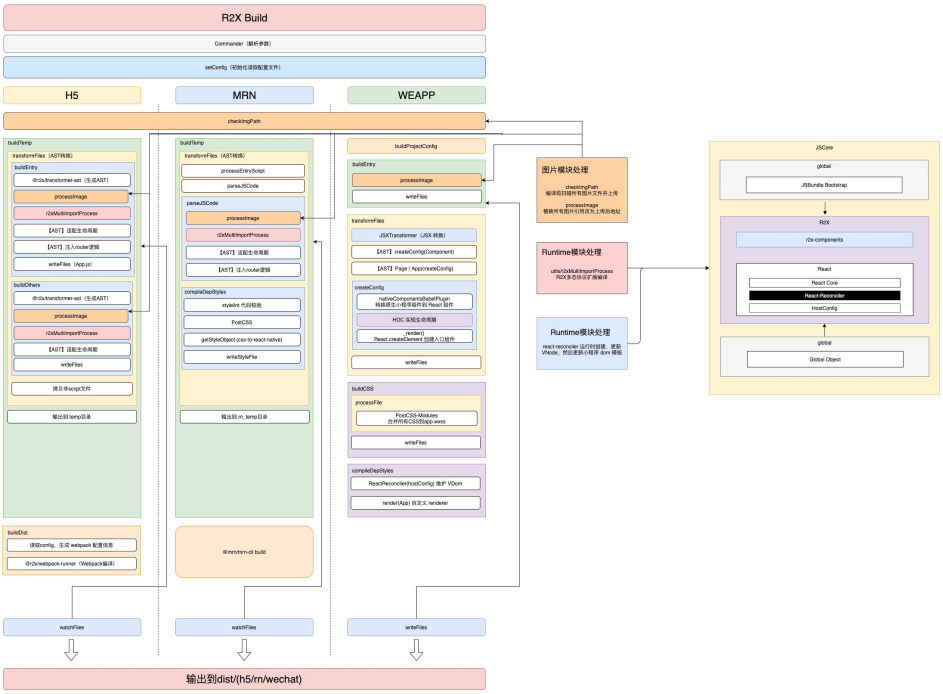


图 7

R2X2.0 相比于 R2X1.0，运行时方案能够解决编译时方案带来的语法限制问题：

- 不能在包含 JSX 元素的 map 循环中使用 if 表达式 ✅
- 不能使用 Array#map 之外的方法操作 JSX 数组 ✅
- 不能在 JSX 参数中使用匿名函数 ✅
- 暂不支持在 render() 之外的方法定义 JSX ✅
- 不允许在 JSX 参数 (props) 中传入 JSX 元素 ✅
- 不能在 JSX 参数中使用对象展开符 ✅

同时也支持大部分 React 第三方生态库，目前已支持使用原生 react-redux。彻底抹平各端的差异，无论是 MRN、Flutter、小程序、Webview 的自定义组件都可以直接当成 React 组件引入，小程序原生自定义组件也无需配置 usingComponents。

4.2.2 模块级容器场景同构

在模块级同构方案上，我们在 App 上依赖 Mach 容器；在小程序容器中，我们克服了 Mach 容器渲染机制的约束（运行时与虚拟 DOM 的使用限制），单独在小程序上设计了 Mach 容器渲染方案，实现了 R2X- 模块化（**R2X-Module**）在客户端和小程序上 99% 以上的代码同构率。

整体方案

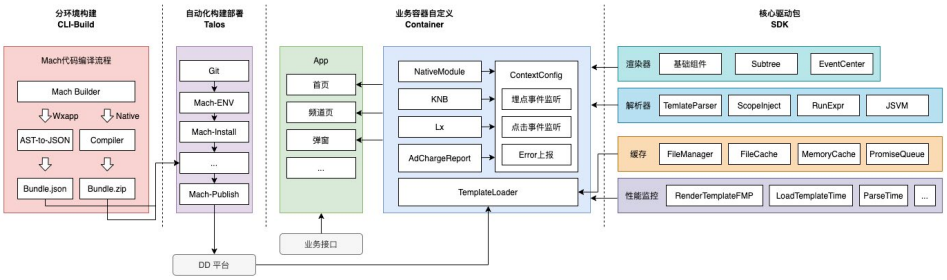


图 8

- **核心驱动包**，容器驱动的核心，针对渲染能力、解析能力、缓存能力、性能监控四个方面进行了实现，达到动态化驱动效果。
- **业务容器自定义**，基于 SDK 提供的驱动能力，针对不同展位特性进行了容器自定义功能扩展配置，让业务方可根据实际业务场景自行扩展。
- **分环境构建**，主要实现了将类 React 语法进行 AST 编译解析，根据构建平台分别编译成对应的 Bundle 产物。
- **自动化构建部署**，将构建能力接入 Talos（美团内部自研的构建部署工具），再结合低代码业务工具实现一键部署，将编译产物根据配置项上传至 DD（美团内部自研移动端动态下发平台）。

模板驱动方案

目前，R2X-Module 在客户端和小程序容器的同构率在 99.3% 以上，在性能方面首次渲染时长和模板渲染时长的 TP50 时间分别是 **185ms** 和 **144ms**，比较优秀但还存在优化空间。同时提供 R2X-Module SDK 供业务方选择。R2X-Module SDK

初始化以及模板加载渲染流程如下图所示：

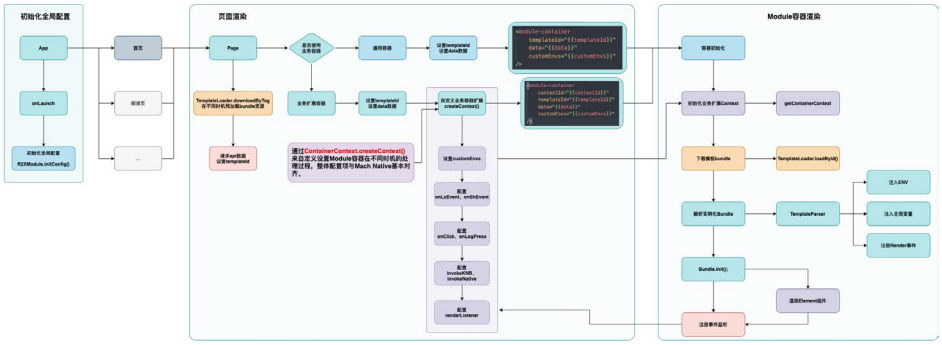


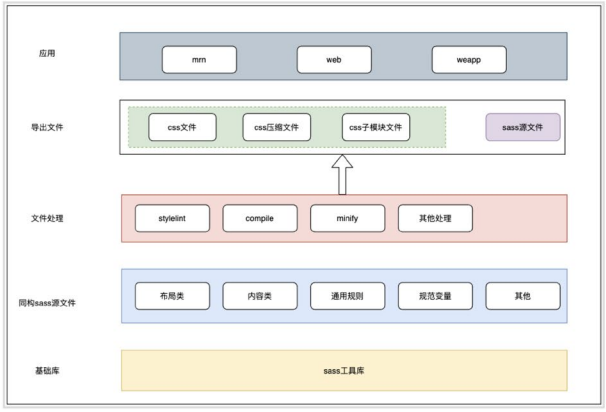
图 9

4.2.3 PC/App 适配同构

在移动互联网发展已经高度成熟的今天，移动端的 PV 流量占比绝大数，以外卖广告商家端为例，PC 端仅仅占有很少比例，其中 PC 流量占比在我们部分业务上已经不及 5%。因此在某些场景下实现 PC/App 的同构方案能够解放一部分人力，对提高开发效率来说是十分必要的。目前，外卖广告商家端的一些轻量布局差异的页面，已经完成了 PC/App 同构的方案设计和落地。

样式同构适配

同构样式库设计：



R2X样式转换插件：

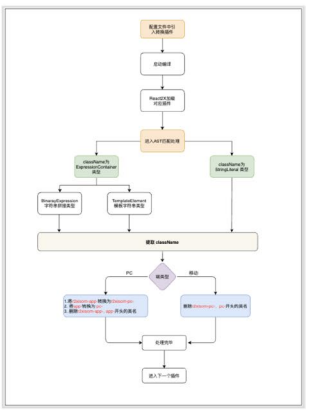


图 10

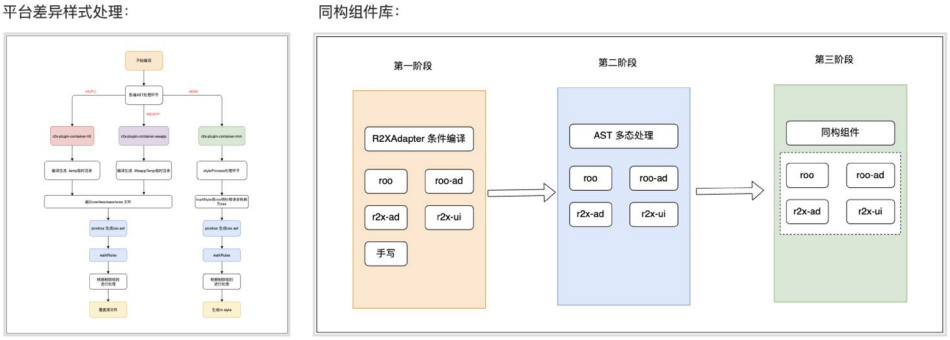


图 11

端能力扩展

R2X 的基础能力支持 Webview/MRN/ 小程序三端，缺少对 PC 微前端子项目的支持。要实现 PC/APP 多端同构需要对 R2X 的端能力进行扩展。PC 端本质上也属于 Web 端，因此 PC 微前端的端能力扩展可以复用大部分的 Webview 的端能力。整体架构图、技术设计要点、扩展流程图如下所示：

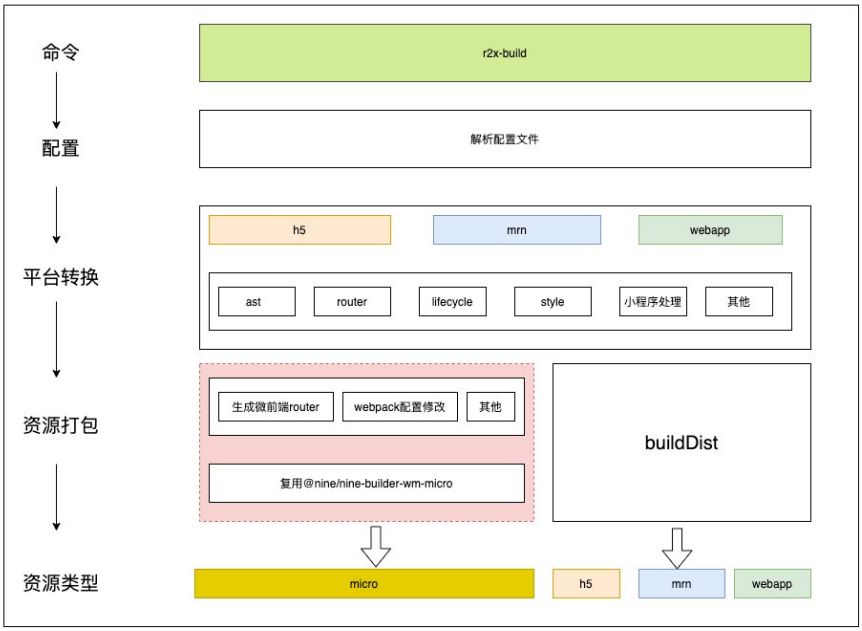


图 12

平台代码处理

在项目同构开发中，不可避免地会出现跟平台强相关的代码或者业务逻辑，比如某些 API 调用的是 App 的底层能力，只能在 React Native 中使用，在 Web 端肯定是不支持的。或者由于产品需求的原因，某些交互或者展示差异较大等等。而项目针对某一端进行编译、打包时，其他不相关的端代码是无用、多余的，如果保留的话，不仅会增加代码体积，甚至会出现编译报错，因此我们需要借助平台代码处理的能力来进行优化。平台代码的处理主要包含三部分：模块导入、组件展示、业务逻辑。

主要思路是使用注释和指定平台的方式，让特定的平台代码只在特定平台生效，注释关键字 `%%platform%%`，比如 `%%RN%%` 表示 React Native 端独有，`%%MICRO%%` 表示 PC 微前端独有，`%%MICRO|Webview%%` 表示 PC 微前端、Webview 两端生效。示例代码如下：

```
import A from '@r2x/r2x-a'; // %%RN%% 只在 React Native 端保留。
import B from '@r2x/r2x-b'; // %%MICRO%% 只在 MICRO 端保留。
import C from '@/utils/c'; // 这是所有端生效的公共模块。
import D from '@r2x/r2x-d'; // %%MICRO|Webview%% 在 MICRO、Webview 多端生效的模块。
```

4.2.4 小游戏容器场景同构

实现 react2x-game 同构方案主要做的两点：渲染层的兼容、业务层的兼容。

- **渲染层的兼容**：实现游戏引擎在多端环境下渲染能力的兼容（Canvas、WebGL）。
- **业务层的兼容**：实现基础 API、项目流程、公共模块的兼容，制定游戏差异的个性化定制规范。

渲染层兼容

在上文，我们提到过“无论是 Webview 游戏、小程序、小游戏、美团小游戏都为我们提供了 Canvas、WebGL 控件”，很大程度地降低了我们兼容渲染层的复杂度。下面表单，是各端对于语法以及 Canvas、WebGL、Document、Window 等基础功能的支持情况：

对象	Webview	微信小游戏	微信小程序	美团小游戏
语法	JavaScript	JavaScript	JavaScript	JavaScript
Canvas	支持	支持	支持	支持
Canvas（离屏）	支持	支持	不支持	支持
WebGL	支持	支持	>2.11.0	支持
Ducument	支持	不支持	不支持	不支持
Window	支持	不支持	不支持	不支持

可以看出，在语法层面各端都支持了 JavaScript 语法，但是在执行环境以及基础功能上的差异比较大，总结来说：

执行环境：小游戏、小程序不具备 DOM、BOM 的能力（渲染引擎中会大量使用）。

基础功能：小程序不支持离屏 Canvas，在 2.11.0 版本以后才开始支持 WebGL。

为了解决这些问题，我们设计开发了 adaptor 层，用来模拟 document、window 的能力。使游戏引擎可以在非 Webview 的环境下正常的执行和调用 BOM、DOM 的基础功能。同时，制定离屏 canvas 的适配方案，用来解决小程序无法支持离屏 canvas 的问题。为了获取到有效离屏 canvas，我们制作了**“r2x-add-wxml-loader”，在 .wxml 文件的 loader 阶段自动注入额外的 < canvas/> 控件**，并隐藏于手机屏幕之外，用于模拟游戏引擎中的离屏 canvas。

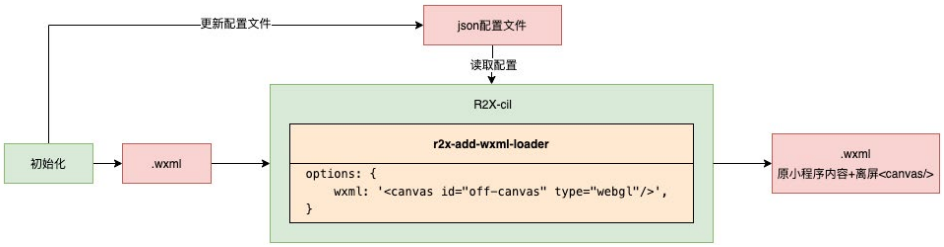


图 13

多端兼容构建

在构建层面，我们通过集成的多种个性化插件工具，对多端代码进行差异处理。如：环境变量注入、各端适配代码的混入、规范检测、代码解析和转化等。针对小游戏、

小程序代码和执行环境的特殊性，制作 wx-build-plugin、lwxapp-build-plugin 等用于处理小游戏和小程序的打包工作。结合上文中提到的各类差异的处理方案，制作 add-wxml-loader、transfrom-loader、wxss-loader 等工具协助完成项目构建。如下图 14 所示，构建之初会注入本次构建的环境变量，读取和分析配置文件，集成和初始化构建工具集合，为项目构建做准备。然后在构建环节，针对各端的差异进行差别处理，分析层针对不同文件进行解析，并在转换层进行转换和构建，最终生成各端需要的最终产物。

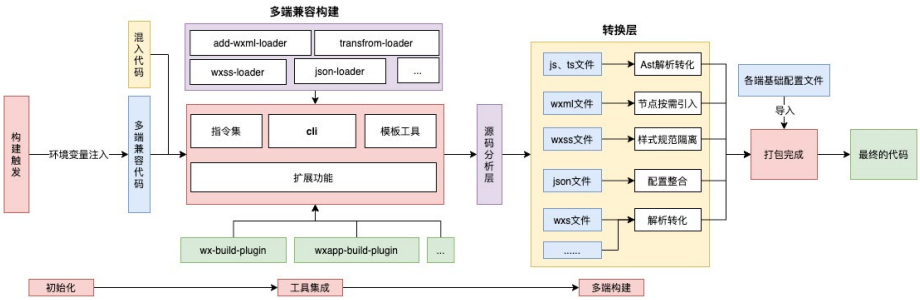


图 14

4.3 落地场景与效果



MRN效果图　　小程序效果图　　H5效果图

R2X- 推广页容器场景同构



R2X- 模块化容器场景同构



R2X- 小游戏容器场景同构

效果收益

R2X 在美团外卖业务中得到了广泛的应用。截止 2021 年 10 月，R2X 累计在美团内部已有二十多个部门在使用或者在调研中，总计落地了上百个工程、页面，框架下

载量达百万次，页面平均代码同构率达 90% 以上。R2X 生态体系在容器代码复用与运维层面，累计为美团节省成本上千人 / 日，并提升动态化页面转化 5%–8% 的成功率。

五、展望与总结

综上所述，在美团外卖多元化业务形态和容器多样性的情况下，跨容器复用成为了发展的必经之路。而 R2X 在经历了两年的迭代下也取得了阶段性的成果，在美团各个业务场景都完成了业务的落地覆盖，针对公司的生态环境接入也做出了不少的基础建设。我们相信跨容器多端代码复用依旧是当前缩减项目交付周期，减少研发成本，提升研发效率的重要一环。但目前我们在很多复杂的业务场景下做的不够完美，因此还有许多工作待完善，例如：

- **开发体验优化**，目前想接入或正在接入的兄弟部门已经越来越多，如何减少接入成本，丰富基础建设，优化开发体验，帮助大家快速迁移接入，将是下一阶段的重要课题。
- **渲染性能优化**，在美团外卖场景下性能优化一直是我们在兼顾高效生产的另一个重要指标。特别在小程序场景下，低端机型的性能体验一直是业界瓶颈，如何突破这一难关将会是同构方案全面推广的“敲门砖”。

最后，感谢各个相关研发团队对 R2X 建设过程中的鼎力支持，R2X 的发展离不开所有参与者日以继夜的投入和贡献，我们会持续基于 R2X 在终端容器领域进行更多探索。如果大家觉得 R2X 还不错，或者对美团的 R2X 框架比较感兴趣，欢迎跟我们一起交流探讨。

作者简介

正浩、宝石、彭震，均为美团外卖终端团队研发工程师。

招聘信息

美团外卖长期招聘 Android、iOS、FE、Java 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 lizhenghao@meituan.com。

一款可以让大型 iOS 工程编译速度提升 50% 的工具

作者：思琦　旭陶　霜叶

cocoapods-hmap-prebuilt 是什么?

cocoapods-hmap-prebuilt 是美团平台迭代组自研的一款 cocoapods 插件，以 Header Map 技术 为基础，进一步提升代码的编译速度，完善头文件的搜索机制。

虽然以二进制组件的方式构建 App 是 HPX（美团移动端统一持续集成 / 交付平台）的主流解决方案，但在某些场景下（Profile、Address/Thread/UB/Coverage Sanitizer、App 级别静态检查、ObjC 方法调用兼容性检查等等），我们的构建工作还是需要以全源码编译的方式进行；而且在实际开发过程中，大多是以源码的方式进行开发，所以我们将实验对象设置为基于全源码编译的流程。

废话不多说，我们来看看它的实际使用效果！

总的来说，以美团和大众点评的全源码编译流程为实验对象的前提下，cocoapods-hmap-prebuilt 插件能将总链路提升 45% 以上的速度，在 Xcode 打包环节上能提升 50% 以上的速度，是不是有点动心了?

为了更好的理解这个插件的价值和功能，我们不妨先看一下当前的工程中存在的问题。

为什么现有的项目不够好?

目前，美团内的 App 都是基于 CocoaPods 做包管理方面的工作，所以在实际的开发过程中，CocoaPods 会在 `Pods/Header/` 目录下添加组件名目录和头文件软链，类似于下面的形式：

```
/Users/sketchk/Desktop/MyApp/Pods
└──── Headers
```

```
    ├── Private
    │   └── AFNetworking
    │       ├── AFHTTPRequestOperation.h -> ./XXX/
AFHTTPRequestOperation.h
    │       ├── AFHTTPRequestOperationManager.h -> ./XXX/
AFHTTPRequestOperationManager.h
    │       ├── ...
    │       └── UIRefreshControl+AFNetworking.h -> ./XXX/
UIRefreshControl+AFNetworking.h
    └── Public
        └── AFNetworking
            ├── AFHTTPRequestOperation.h -> ./XXX/
AFHTTPRequestOperation.h
            ├── AFHTTPRequestOperationManager.h -> ./XXX/
AFHTTPRequestOperationManager.h
            ├── ...
            └── UIRefreshControl+AFNetworking.h -> ./XXX/
UIRefreshControl+AFNetworking.h
```

也正是通过这样的目录结构和软链，CocoaPods 得以在 Header Search Path 中添加如下的参数，使得预编译环节顺利进行。

```
$(inherited)
${PODS_ROOT}/Headers/Private
${PODS_ROOT}/Headers/Private/AFNetworking
${PODS_ROOT}/Headers/Public
${PODS_ROOT}/Headers/Public/AFNetworking
```

虽然这种构建 Search Path 的方式解决了预编译的问题，但在某些项目中，例如多达 400+ 组件的巨型项目中，会造成以下几点问题：

- 大量的 Header Search Path 路径，会造成编译参数中的 `-I` 选项极速膨胀，在达到一定长度后，甚至会造成无法编译的情况
- 目前美团的工程中，已经有近 5W 个头文件，这意味着不论是头文件的搜索过程，还是软链的创建过程，都会引起大量的文件 IO 操作，进而会产生一些耗时操作。
- 编译时间会随着组件数量急剧增长，以美团和大众点评有 400+ 个组件的体量为参考，全源码打包耗时均在 1 小时以上。
- 基于路径顺序查找头文件的方式有潜在的风险，例如重名头文件的情况，排在

后面的头文件永远无法参与编译。

- 由于 `${PODS_ROOT}/Headers/Private` 路径的存在，让引用其他组件的私有头文件变为了可能。

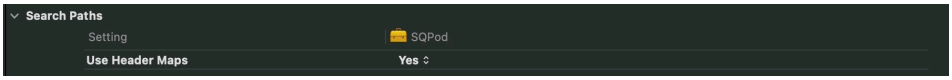
想解决上述的问题，好一点的情况下，可能会浪费 1 个小时，而不好的情况，就是让有风险的代码上线了，你说工程师会不会因此而感到头疼?

Header Map 是个啥?

还好 cocoapods-hmap-prebuilt 的出现，让这些问题变成了历史，不过要想理解它为什么能解决这些问题，我们得先理解一下什么是 Header Map。

Header Map 其实是一组头文件信息映射表!

为了更直观的理解 Header Map，我们可以在 Build Setting 中开启 Use Header Map 选项，真实的体验一下它。



然后在 Build Log 里获取相应组件里对应文件的编译命令，并在最后加上 -v 参数，来查看其运行的秘密：

```
$ clang <list of arguments> -c some-file.m -o some-file.o -v
```

在 console 的输出内容中，我们会发现一段有意思的内容：

```
#include "..." search starts here:
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-generated-files.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-project-headers.hmap (headermap)
#include <...> search starts here:
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-own-target-headers.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-all-non-framework-target-headers.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Products/Debug-iphonesimulator/PodA/PodA.framework/Headers
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/DerivedSources
```

通过上面的图，我们可以看到编译器将寻找头文件的顺序和对应路径展示出来了，而在这些路径中，我们看到了一些陌生的东西，即后缀名为 `.hmap` 的文件，后面还有个括号写着 headermap。

没错！它就是 Header Map 的实体。

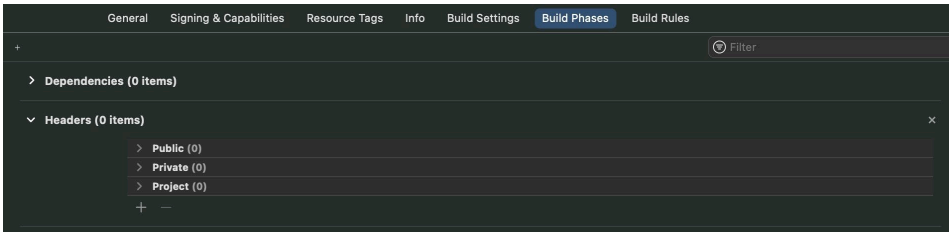
此时 Clang 已经在刚才提到的 hmap 文件里塞入了一份头文件名和头文件路径的映射表，不过它是一种二进制格式的文件，为了验证这个的说法，我们可以通过 milend 编写的 hmap 工具来查其内容。

在执行相关命令（即 `hmap print`）后，我们可以发现这些 hmap 里保存的信息结构大致如下，类似于一个 Key-Value 的形式，Key 值是头文件的名称，Value 是头文件的实际物理路径：

```
sketchk ~/Desktop/Host git main
▶ hmap print /Users/sketchk/Library/Developer/Xcode/DerivedData/Host-ewmvmvyzlhizmsau
byxgeyqcnucf/Build/Intermediates.noindex/Pods.build/Debug-iphonesimulator/PodA.build/
PodA-own-target-headers.hmap
ClassA.h -> PodA/ClassA.h
ClassB.h -> PodA/ClassB.h
ClassF.h -> /Users/sketchk/Desktop/PodA/PodA/Classes/C/ClassF.h
PodA-umbrella.h -> PodA/PodA-umbrella.h
PodA/ClassF.h -> /Users/sketchk/Desktop/PodA/PodA/Classes/C/ClassF.h
```

需要注意，映射表的键值内容会随着使用场景产生不同的变化，例如头文件引用是在 `"..."` 的形式下，还是 `<...>` 的形式下，又或是在 Build Phase 里 Header 的配置情况。例如，你将头文件设置为 Public 的时候，在某些 hmap 中，它的 Key 值就为

`PodA/ClassA`，而将其设置为 project 的时候，它的 Key 值可能就是 `ClassA`，而配置这些信息的地方，如下图所示：



至此我想你应该了解到 Header Map 到底是个什么东西了。

当然这种技术也不是一个什么新鲜事儿，在 Facebook 的 buck 工具中也提供了类似的东西，只不过文件类型变成了 `HeaderMap.java` 的样子。

此时，我估计你可能并不会对 buck 产生太多的兴趣，而是开始思考上一张图中 Headers 的 Public、Private、Project 到底代表着什么意思，好像很多同学从来没怎么关注过，以及为什么它会影响 hmap 里的内容?

Public，Private，Project 是个啥?

在 Apple 官方的 Xcode Help - What are build phases? 文档中，我们可以看到如下的一段解释：

> Associates public, private, or project header files with the target. Public and private headers define API intended for use by other clients, and are copied into a product for installation. For example, public and private headers in a framework target are copied into Headers and Private-Headers subfolders within a product. Project headers define API used and built by a target, but not copied into a product. This phase can be used once per target.

总的来说，我们可以知道一点，就是 Build Phases - Headers 中提到 Public 和

Private 是指可以供外界使用的头文件，而 Project 中的头文件是不对外使用的，也不会放在最终的产物中。

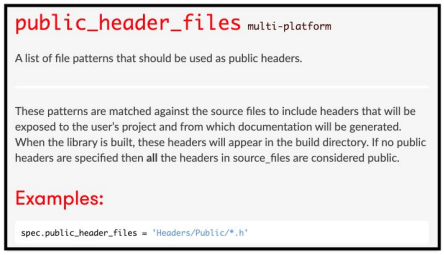
如果你继续翻阅一些资料，例如 StackOverflow – Xcode: Copy Headers: Public vs. Private vs. Project? 和 StackOverflow – Understanding Xcode's Copy Headers phase，你会发现在早期 Xcode Help 的 Project Editor 章节里，有一段名为 Setting the Role of a Header File 的段落，里面详细记载了三个类型的区别。

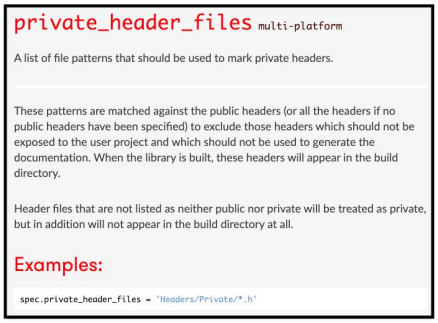
> **Public:** The interface is finalized and meant to be used by your product's clients. A public header is included in the product as readable source code without restriction. **Private:** The interface isn't intended for your clients or it's in early stages of development. A private header is included in the product, but it's marked "private". Thus the symbols are visible to all clients, but clients should understand that they're not supposed to use them. **Project:** The interface is for use only by implementation files in the current project. A project header is not included in the target, except in object code. The symbols are not visible to clients at all, only to you.

至此，我们应该能够彻底了解了 Public、Private、Project 的区别。简而言之，Public 还是通常意义上的 Public，Private 则代表 In Progress 的含义，至于 Project 才是通常意义上的 Private 含义。

此时，你会不会联想到 CocoaPods 中 Podspec 的 Syntax 里还有 `public_header_files` 和 `private_header_files` 两个字段，它们的真实含义是否和 Xcode 里的概念冲突呢?

这里我们仔细阅读一下官方文档的解释，尤其是 `private_header_files` 字段。





我们可以看到，`private_header_files` 在这里的含义是说，它本身是相对于 Public 而言的，这些头文件本义是不希望暴露给用户使用的，而且也不会产生相关文档，但是在构建的时候，会出现在最终产物中，只有既没有被 Public 和 Private 标注的头文件，才会被认为是真正的私有头文件，且不出现在最终的产物里。

看起来，CocoaPods 对于 Public 和 Private 的官方解释是和 Xcode 中的描述一致的，两处的 Private 并非我们通常理解的 Private，它的本意更应该是开发者准备对外开放，但又没完全 Ready 的头文件，更像一个 In Progress 的含义。

这一块是不是让你有点大跌眼镜？那么，在现实世界中，我们是否正确的使用了它们呢？

为什么用原生的 hmap 不能改善编译速度？

前面我们介绍了 hmap 是什么，以及怎么开启它（启用 Build Setting 中的 Use Header Map 选项），也介绍了一些影响生成 hmap 的因素（Public、Private、Project）。

那是不是我只要开启 Xcode 提供的 Use Header Map 就可以提升编译速度了呢？

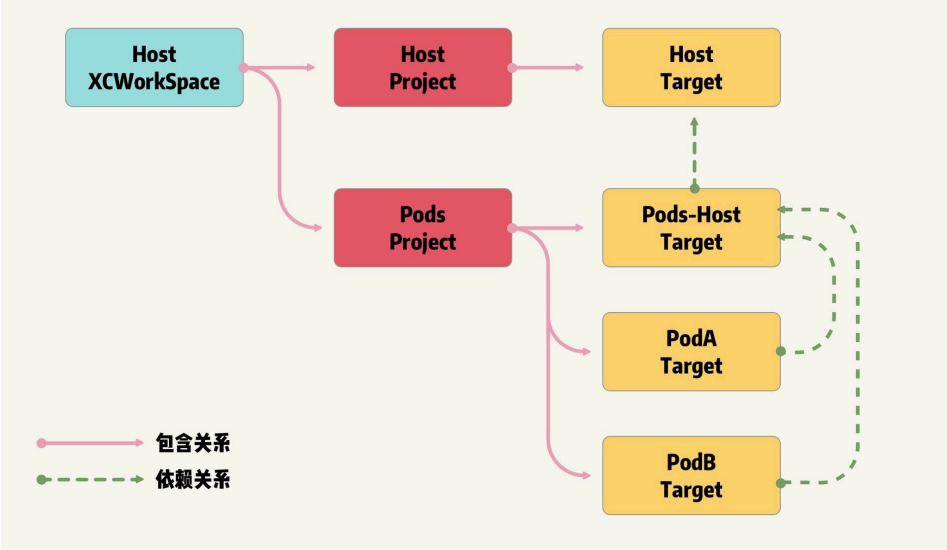
很可惜，答案是否定的！

至于原因，我们就从下面的例子开始说起，假设我们有一个基于 CocoaPods 构建的全源码工程项目，它的整体结构如下：

- 首先，Host 和 Pod 是我们的两个 Project，Pods 下的 Target 的产物类型为 Static Library。

- 其次，Host 底下会有一个同名的 Target，而 Pods 目录下会有 n+1 个 Target，其中 n 取决于你依赖的组件数量，而 1 是一个名为 Pods-XXX 的 Target，最后，Pods-XXX 这个 Target 的产物会被 Host 里的 Target 所依赖。

整个结构看起来如下所示：

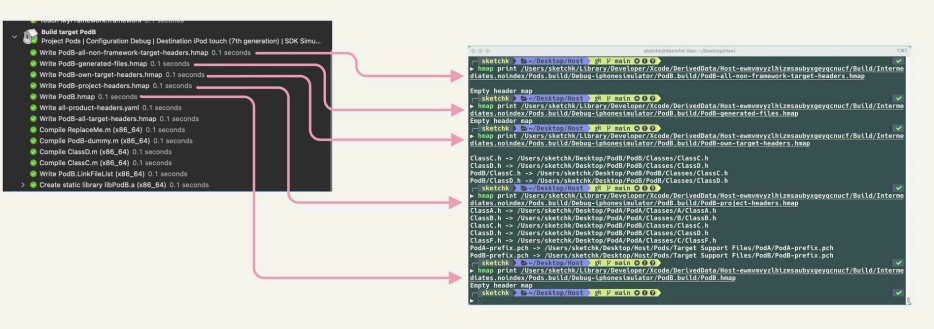


当构建的产物类型为 Static Library 的时候，CocoaPods 在创建头文件产物过程中，它的逻辑大致如下：

- 不论 podspec 里如何设置 `public_header_files` 和 `private_header_files`，相应的头文件都会被设置为 Project 类型。
- 在 `Pods/Headers/Public` 中会保存所有被声明为 `public_header_files` 的头文件。
- 在 `Pods/Headers/Private` 中会保存所有头文件，不论是 `public_header_files` 或者 `private_header_files` 描述到，还是那些未被描述的，这个目录下是当前组件的所有头文件全集。
- 如果 podspec 里未标注 Public 和 Private 的时候，`Pods/Headers/Public` 和 `Pods/Headers/Private` 的内容一样且会包含所有头文件。

正是由于这种机制，会导致一些有意思的问题发生。

- 首先，由于所有头文件都被当做最终产物保留下来，在结合 Header Search Path 里 `Pods/Headers/Private` 路径的存在，我们完全可以引用到其他组件里的私有头文件，例如我只要使用 `#import <SomePod/Private_Header.h>` 的方式，就会命中私有文件的匹配路径。
- 其次，就是在 Static Library 的状况下，一旦我们开启了 Use Header Map，结合组件里所有头文件的类型为 Project 的情况，这个 hmap 里只会包含 `#import "ClassA.h"` 的键值引用，也就是说只有 `#import "ClassA.h"` 的方式才会命中 hmap 的策略，否则都将通过 Header Search Path 寻找其相关路径，例如下图中的 PodB，在其 build 的过程中，Xcode 会为 PodB 生成 5 个 hmap 文件，也就是说这 5 个文件只会在编译 PodB 中使用，其中 PodB 会依赖 PodA 的一些头文件，但由于 PodA 中的头文件都是 Project 类型的，所以其在 hmap 里的 Key 全部为 `ClassA.h`，也就是说我们只能以 `#import "ClassA.h"` 的方式引入。



而我们也知道，在引用其他组件的时候，通常都会采用 `#import <A/A.h>` 的方式引入。至于为什么会用这种方式，一方面是这种写法会明确头文件的由来，避免问题，另一方面也是这种方式可以让我们在是否开启 clang module 中随意切换。当然，还有一点就是 Apple 在 WWDC 里曾经不止一次建议开发者使用这种方式来引入头文件。

接着上面的话题来说，所以说在 Static Library 的情况下且以 `#import <A/A.h>` 这

种标准方式引入头文件时，开启 Use Header Map 选项并不会帮我们提升编译速度。

但真的就没有办法使用 Header Map 了么?

cocoapods-hmap-prebuilt 诞生了

当然，总是有办法解决的，我们完全可以自己动手做一个基于 CocoaPods 规则下的 hmap 文件，正是基于这个想法，美团自研的 cocoapods-hmap-prebuilt 插件诞生了。

它的核心功能并不多，大概有以下几点：

- 借助 CocodPods 处理 Header Search Path 和创建头文件 soft link 的时机，构建了头文件索引表并以此生成 n+1 个 hmap 文件（n 是每个组件自己的 Private Header 信息，1 是所有组件公共的 Public Header 信息）。
- 重写 xcconfig 文件里的 Header Search Path 到对应的 hmap 文件上，一条指向组件自己的 private hmap，一条指向所有组件共用的 public hmap。
- 针对 public hmap 里的重名头文件进行了特殊处理，只允许保存组件名 / 头文件名方式的 Key-Value，排查重名头文件带来的异常行为。
- 将组件自身的 Ues Header Map 功能关闭，减少不必要的文件创建和读取。

听起来可能有点绕，内容也有点多，不过这些你都不用关心，你只需要通过以下 2 个步骤就能将其使用起来：

- 在 Gemfile 里声明插件。
- 在 Podfile 里使用插件。

```
// this is part of Gemfile
source 'http://sakgems.sankuai.com/' do
  gem 'cocoapods-hmap-prebuilt'
  gem 'XXX'
  ...
end

// this is part of Podfile
```

```
target 'XXX' do
  plugin 'cocoapods-hmap-prebuilt'
  pod 'XXX'
  ...
end
```

除此之外，为了拓展其实用性，我们还提供了头文件补丁（解决重名头文件的定向选取）和环境变量注入（无侵入的在其他系统中使用）的能力，便于其在不同场景下的使用。

总结

至此，关于 cocoapods-hmap-prebuilt 的介绍就要结束了。

回看整个故事的开始，Header Map 是我在研究 Swift 和 Objective-C 混编过程中发现的一个很小的知识点，而且 Xcode 自身就实现了一套基于 Header Map 的功能，在实际的使用过程中，它的表现并不理想。

但幸运的是，在后续的探索的过程中，我们发现了为什么 Xcode 的 Header Map 没有生效，以及为什么它与 CocoaPods 出现了不兼容的情况，虽然它的原理并不复杂，核心点就是将文件查找和读取等 IO 操作编变成了内存读取操作，但结合实际的业务场景，我们发现它的收益是十分可观的。

或许这是在提醒我们，要永远对技术保持一颗好奇的心！

其实，利用 Clang Module 技术也可以解决本文一开始提到的几个问题，但它并不在这篇文章的讨论范围中，如果你对 Clang Module 或者对 Swift 与 Objective-C 混编感兴趣，欢迎阅读参考文档中的《从预编译的角度理解 Swift 与 Objective-C 及混编机制》一文，以了解更多的详细信息。

参考文档

- Apple - WWDC 2018 Behind the Scenes of the Xcode Build Process
- Apple 的 HeaderMap.cpp 源码

作者

思琦，笔名 SketchK，美团 iOS 工程师，目前负责移动端 CI/CD 方面的工作及平台内 Swift 技术相关的事宜。

旭陶，美团 iOS 工程师，目前负责 iOS 端开发提效相关事宜。

霜叶，2015 年加入美团，先后从事过 Hybrid 容器、iOS 基础组件、iOS 开发工具链和客户端持续集成门户系统等工作。

从预编译的角度理解 Swift 与 Objective-C 及混编机制

作者：思琦　旭陶　霜叶

写在前面

本文涉及面较广，篇幅较长，阅读完需要耗费一定的时间与精力，如果你带有较为明确的阅读目的，可以参考以下建议完成阅读：

- 如果你对预编译的理论知识已经了解，可以直接从【原来它是这样的】的章节开始进行阅读，这会让你对预编译有一个更直观的了解。
- 如果你对 Search Path 的工作机制感兴趣，可以直接从【关于第一个问题】的章节阅读，这会让你更深刻，更全面的了解到它们的运作机制，
- 如果您对 Xcode Phases 里的 Header 的设置感到迷惑，可以直接从【揭开 Public、Private、Project 的真实面目】的章节开始阅读，这会让你理解为什么说 Private 并不是真正的私有头文件
- 如果你想了解如何通过 hmap 技术提升编译速度，可以从【基于 hmap 优化 Search Path 的策略】的章节开始阅读，这会给你提供一种新的编译加速思路。
- 如果你想了解如何通过 VFS 技术进行 Swift 产物的构建，可以从【关于第二个问题】章节开始阅读，这会让你理解如何用另外一种提升构建 Swift 产物的效率。
- 如果你想了解 Swift 和 Objective-C 是如何找寻方法声明的，可以从【Swift 来了】的章节阅读，这会让你从原理上理解混编的核心思路和解决方案。

概述

随着 Swift 的发展，国内技术社区出现了一些关于如何实现 Swift 与 Objective-C 混编的文章，这些文章的主要内容还是围绕着指导开发者进行各种操作来实现混编的效果，例如在 Build Setting 中开启某个选项，在 podspec 中增加某个字段，而鲜有文章对这些操作背后的工作机制做剖析，大部分核心概念也都是一笔带过。

正是因为这种现状，很多开发者在面对与预期不符的行为时，亦或者遇到各种奇怪的报错时，都会无从下手，而这也是由于对其工作原理不够了解所导致的。

笔者在美团平台负责 CI/CD 相关的工作，这其中也包含了 Objective-C 与 Swift 混编的内容，出于让更多开发者能够进一步理解混编工作机制的目的，撰写了这篇技术文章。

废话不多说，我们开始吧！

预编译知识指北

`#import` 的机制和缺点

在我们使用某些系统组件的时候，我们通常会写出如下形式的代码：

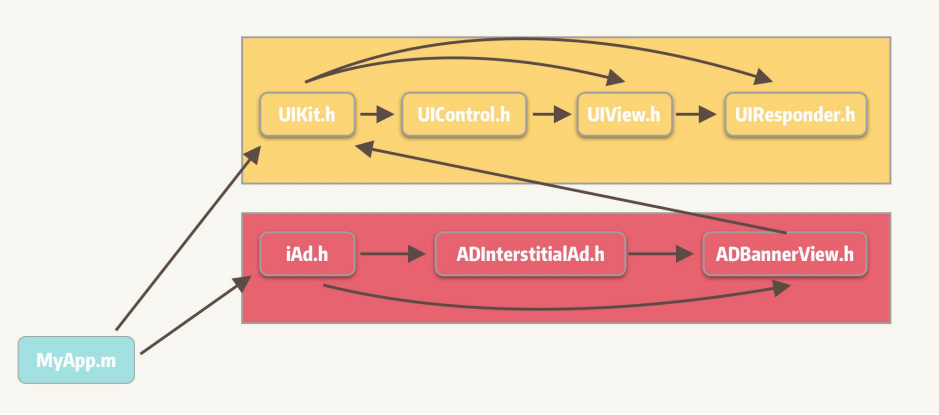
```
#import <UIKit/UIKit.h>
```

`#import` 其实是 `#include` 语法的微小创新，它们在本质上还是十分接近的。`#include` 做的事情其实就是简单的复制粘贴，将目标 `.h` 文件中的内容一字不落地拷贝到当前文件中，并替换掉这句 `#include`，而 `#import` 实质上做的事情和 `#include` 是一样的，只不过它还多了一个能够避免头文件重复引用的能力而已。

为了更好的理解后面的内容，我们这里需要展开说一下它到底是如何运行的?

从最直观的角度来看：

假设在 `MyApp.m` 文件中，我们 `#import` 了 `iAd.h` 文件，编译器解析此文件后，开始寻找 iAd 包含的内容（`ADInterstitialAd.h`，`ADBannerView.h`），及这些内容包含的子内容（`UIKit.h`，`UIController.h`，`UIView.h`，`UIResponder.h`），并依次递归下去，最后，你会发现 `#import <iAd/iAd.h>` 这段代码变成了对不同 SDK 的头文件依赖。



如果你觉得听起来有点费劲，或者似懂非懂，我们这里可以举一个更加详细的例子，不过请记住，对于 C 语言的预处理器而言，`#import` 就是一种特殊的复制粘贴。

结合前面提到的内容，在 AppDelegate 中添加 `iAd.h`：

```
#import <iAd/iAd.h>
@implementation AppDelegate
//...
@end
```

然后编译器会开始查找 `iAd/iAd.h` 到底是哪个文件且包含何种内容，假设它的内容如下：

```
/* iAd/iAd.h */
#import <iAd/ADBannerView.h>
#import <iAd/ADBannerView_Deprecated.h>
#import <iAd/ADInterstitialAd.h>
```

在找到上面的内容后，编译器将其复制粘贴到 AppDelegate 中：

```
#import <iAd/ADBannerView.h>
#import <iAd/ADBannerView_Deprecated.h>
#import <iAd/ADInterstitialAd.h>

@implementation AppDelegate
//...
@end
```

现在，编译器发现文件里有 3 个 `#import` 语句 了，那么就需要继续寻找这些文件及其相应的内容，假设 `ADBannerView.h` 的内容如下：

```
/* iAd/ADBannerView.h */
@interface ADBannerView : UIView
@property (nonatomic, readonly) ADAdType adType;

- (id)initWithAdType:(ADAdType)type

/* ... */
@end
```

那么编译器会继续将其内容复制粘贴到 AppDelegate 中，最终变成如下的样子：

```
@interface ADBannerView : UIView
@property (nonatomic, readonly) ADAdType adType;

- (id)initWithAdType:(ADAdType)type

/* ... */
@end
#import <iAd/ADBannerView_Deprecated.h>
#import <iAd/ADInterstitialAd.h>

@implementation AppDelegate
//...
@end
```

这样的操作会一直持续到整个文件中所有 `#import` 指向的内容被替换掉，这也意味着 `.m` 文件最终将变得极其的冗长。

虽然这种机制看起来是可行的，但它有两个比较明显的问题：健壮性和拓展性。

健壮性

首先这种编译模型会导致代码的健壮性变差！

这里我们继续采用之前的例子，在 AppDelegate 中定义 `readonly` 为 `0x01`，而且这个定义的声明在 `#import` 语句之前，那么此时又会发生什么事情呢?

编译器同样会进行刚才的那些复制粘贴操作，但可怕的是，你会发现那些在属性声明

中的 `readonly` 也变成了 `0x01`，而这会触发编译器报错！

```
@interface ADBannerView : UIView
@property (nonatomic, 0x01) ADAdType adType;

- (id)initWithAdType:(ADAdType)type

/* ... */
@end

@implementation AppDelegate
//...
@end
```

面对这种错误，你可能会说它是开发者自己的问题。

确实，通常我们都会在声明宏的时候带上固定的前缀来进行区分。但生活里总是有一些意外，不是么?

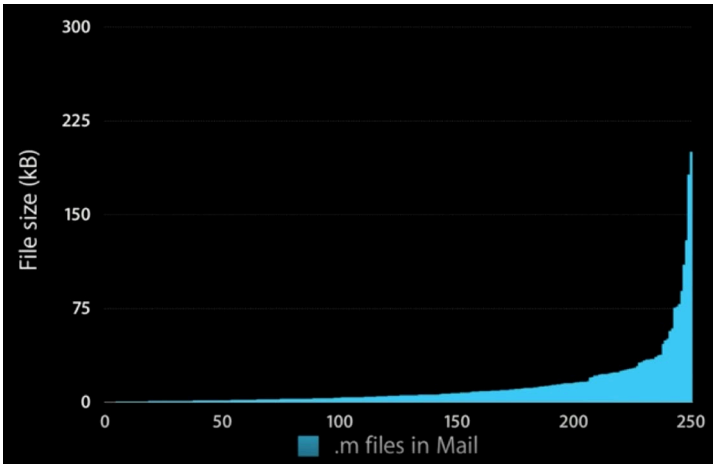
假设某个人没有遵守这种规则，那么在不同的引入顺序下，你可能会得到不同的结果，对于这种错误的排查，还是挺闹心的。不过，这还不是最闹心的，因为还有动态宏的存在，心塞 ing。

所以这种靠遵守约定来规避问题的解决方案，并不能从根本上解决问题，这也从侧面反应了编译模型的健壮性是相对较差的。

拓展性

说完了健壮性的问题，我们来看看拓展性的问题。

Apple 公司对它们的 Mail App 做过一个分析，下图是 Mail 这个项目里所有 `.m` 文件的排序，横轴是文件编号排序，纵轴是文件大小。

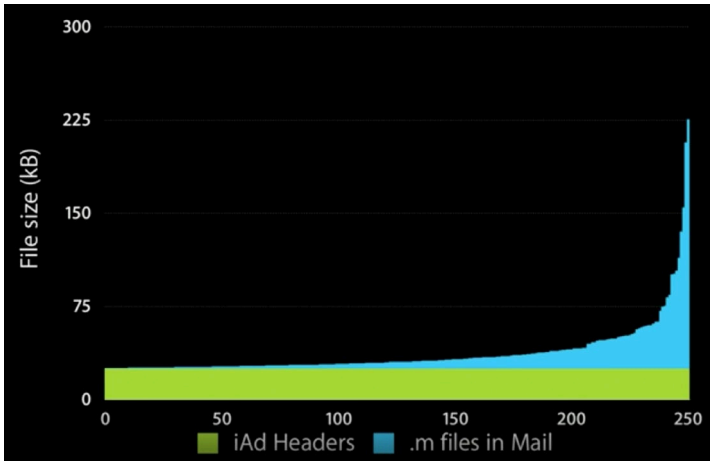


可以看到这些由业务代码构成的文件大小的分布区间很广泛，最小可能有几 kb，最大的能有 200+ kb，但总的来说，可能 90% 的代码都在 50kb 这个数量级之下，甚至更少。

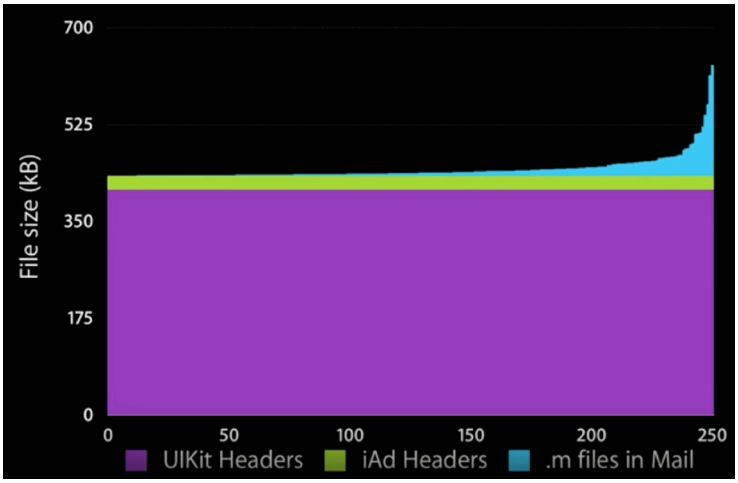
如果我们往该项目的某个核心文件（核心文件是指其他文件可能都需要依赖的文件）里添加了一个对 `iAd.h` 文件的引用，对其他文件意味着什么呢?

> 这里的核心文件是指其他文件可能都需要依赖的文件

这意味着其他文件也会把 `iAd.h` 里包含的东西纳入进来，当然，好消息是，iAd 这个 SDK 自身只有 25KB 左右的大小。



但你得知道 iAd 还会依赖 UIKit 这样的组件，这可是个 400KB+ 的大家伙



所以，怎么说呢?

在 Mail App 里的所有代码都需要先涵盖这将近 425KB 的头文件内容，即使你的代码只有一行 `Hello World`。

如果你认为这已经让人很沮丧的话，那还有更打击你的消息，因为 UIKit 相比于 macOS 上的 Cocoa 系列大礼包，真的小太多了，Cocoa 系列大礼包可是 UIKit 的 29 倍……

所以如果将这个数据放到上面的图表中，你会发现真正的业务代码在 File Size 轴上的比重真的太微不足道了。

所以这就是拓展性差带来的问题之一!

很明显，我们不可能用这样的方式引入代码，假设你有 M 个源文件且每个文件会引入 N 个头文件，按照刚才的解释，编译它们的时间就会是 M * N，这是非常可怕的!

> 备注：文章里提到的 iAd 组件为 25KB，UIKit 组件约为 400KB，macOS 的 Cocoa 组件是 UIKit 的 29 倍等数据，是 WWDC 2013 Session 404 Advances in Objective-C 里公布的数据，随着功能的不断迭代，以现在的眼光来看，这些数据可能已经偏小，在 WWDC 2018 Session 415 Behind the Scenes of the Xcode Build Process 中提到了 Foundation 组件，它包含的头文件数量大于 800 个，大小已经超过 9MB。

PCH(PreCompiled Header) 是一把双刃剑

为了优化前面提到的问题，一种折中的技术方案诞生了，它就是 PreCompiled Header。

我们经常可以看到某些组件的头文件会频繁的出现，例如 UIKit，而这很容易让人联想到一个优化点，我们是不是可以通过某种手段，避免重复编译相同的内容呢?

而这就是 PCH 为预编译流程带来的改进点!

它的大体原理就是，在我们编译任意 `.m` 文件前，编译器会先对 PCH 里的内容进行预编译，将其变为一种二进制的中间格式缓存起来，便于后续的使用。当开始编译 `.m` 文件时，如果需要 PCH 里已经编译过的内容，直接读取即可，无须再次编译。

虽然这种技术有一定的优势，但实际应用起来，还存在不少的问题。

首先，它的维护是有一定的成本的，对于大部分历史包袱沉重的组件来说，将项目中的引用关系梳理清楚就十分麻烦，而要在此基础上梳理出合理的 PCH 内容就更加麻烦，同时随着版本的不断迭代，哪些头文件需要移出 PCH，哪些头文件需要移进 PCH 将会变得越来越麻烦。

其次，PCH 会引发命名空间被污染的问题，因为 PCH 引入的头文件会出现在你代码中的每一处，而这可能会是多于的操作，比如 iAd 应当出现在一些与广告相关的代码中，它完全没必要出现在帮助相关的代码中（也就是与广告无关的逻辑），可是当你把它放到 PCH 中，就意味组件里的所有地方都会引入 iAd 的代码，包括帮助页面，

这可能并不是我们想要的结果！

> 如果你想更深入的了解 PCH 的黑暗面，建议阅读 4 Ways Precompiled Headers Cripple Your Code ，里面已经说得相当全面和透彻。

所以 PCH 并不是一个完美的解决方案，它能在某些场景下提升编译速度，但也有缺陷！

Clang Module 的来临！

为了解决前面提到的问题，Clang 提出了 Module 的概念，关于它的介绍可以在 Clang 官网 上找到。

简单来说，你可以把它理解为一种对组件的描述，包含了对接口（API）和实现（dylib/a）的描述，同时 Module 的产物是被独立编译出来的，不同的 Module 之间是不会影响的。

在实际编译之时，编译器会创建一个全新的空间，用它来存放已经编译过的 Module 产物。如果在编译的文件中引用到某个 Module 的话，系统将优先在这个列表内查找是否存在对应的中间产物，如果能找到，则说明该文件已经被编译过，则直接使用该中间产物，如果没找到，则把引用到的头文件进行编译，并将产物添加到相应的空间中以备重复使用。

在这种编译模型下，被引用到的 Module 只会被编译一次，且在运行过程中不会相互影响，这从根本上解决了健壮性和拓展性的问题。

Module 的使用并不麻烦，同样是引用 iAd 这个组件，你只需要这样写即可。

```
@import iAd;
```

在使用层面上，这将等价于以前的 `#import <iAd/iAd.h>` 语句，但是会使用 Clang Module 的特性加载整个 iAd 组件。如果只想引入特定文件（比如 `ADBannerView.h`），原先的写法是 `#import <iAd/ADBannerView.h>`，现在可以写成：

```
@import iAd.ADBannerView;
```

通过这种写法会将 iAd 这个组件的 API 导入到我们的应用中，同时这种写法也更符合语义化（semanitc import）。

虽然这种引入方式和之前的写法区别不大，但它们在本质上还是有很大程度的不同，Module 不会“复制粘贴”头文件里的内容，也不会让 `@import` 所暴露的 API 被开发者本地的上下文篡改，例如前面提到的 `#define readonly 0x01`。

此时，如果你觉得前面关于 Clang Module 的描述还是太抽象，我们可以再进一步去探究它工作原理，而这就会引入一个新的概念—— modulemap。

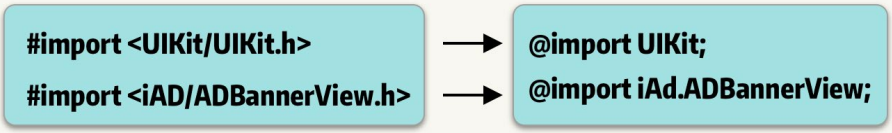
不论怎样，Module 只是一个对组件的抽象描述罢了，而 modulemap 则是这个描述的具体呈现，它对框架内的所有文件进行了结构化的描述，下面是 UIKit 的 modulemap 文件。

```
framework module UIKit {
  umbrella header "UIKit.h"
  module * {export *}
  link framework "UIKit"
}
```

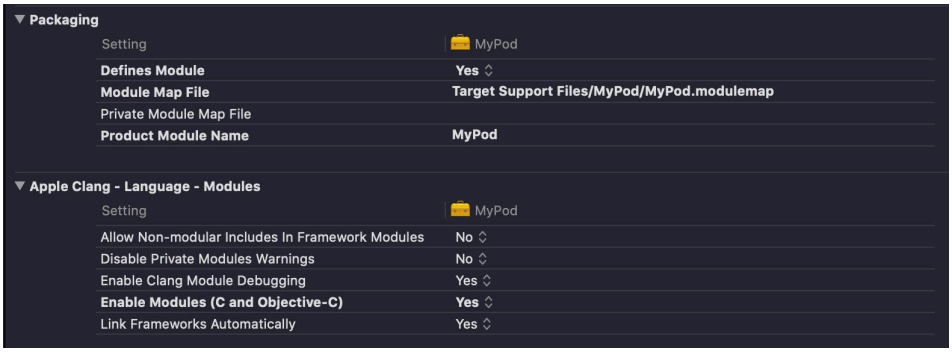
这个 Module 定义了组件的 Umbrella Header 文件（UIKit.h），需要导出的子 Module（所有），以及需要 Link 的框架名称（UIKit），正是通过这个文件，让编译器了解到 Module 的逻辑结构与头文件结构的关联方式。

可能又有人会好奇，为什么我从来没看到过 `@import` 的写法呢？

这是因为 Xcode 的编译器能够将符合某种格式的 `#import` 语句自动转换成 Module 识别的 `@import` 语句，从而避免了开发者的手动修改。



唯一需要开发者完成的就是开启相关的编译选项。



对于上面的编译选项，需要开发者注意的是：

`Apple Clang - Language - Modules` 里 `Enable Module` 选项是指引用系统库的的时候，是否采用 Module 的形式。

而 `Packaging` 里的 `Defines Module` 是指开发者编写的组件是否采用 Module 的形式。

说了这么多，我想你应该对 `#import`，`pch`，`@import` 有了一定的概念。当然，如果我们深究下去，可能还会有如下的疑问：

- 对于未开启 Clang Module 特性的组件，Clang 是通过怎样的机制查找到头文件的呢？在查找系统头文件和非系统头文件的过程中，有什么区别么？
- 对于已开启 Clang Module 特性的组件，Clang 是如何决定编译当下组件的 Module 呢？另外构建的细节又是怎样的，以及如何查找这些 Module 的？还有查找系统的 Module 和非系统的 Module 有什么区别么？

为了解答这些问题，我们不妨先动手实践一下，看看上面的理论知识在现实中的样子。

原来它是这样的

在前面的章节中，我们将重点放在了原理上的介绍，而在这个章节中，我们将动手看看这些预编译环节的实际样子。

`#import` 的样子

假设我们的源码样式如下：

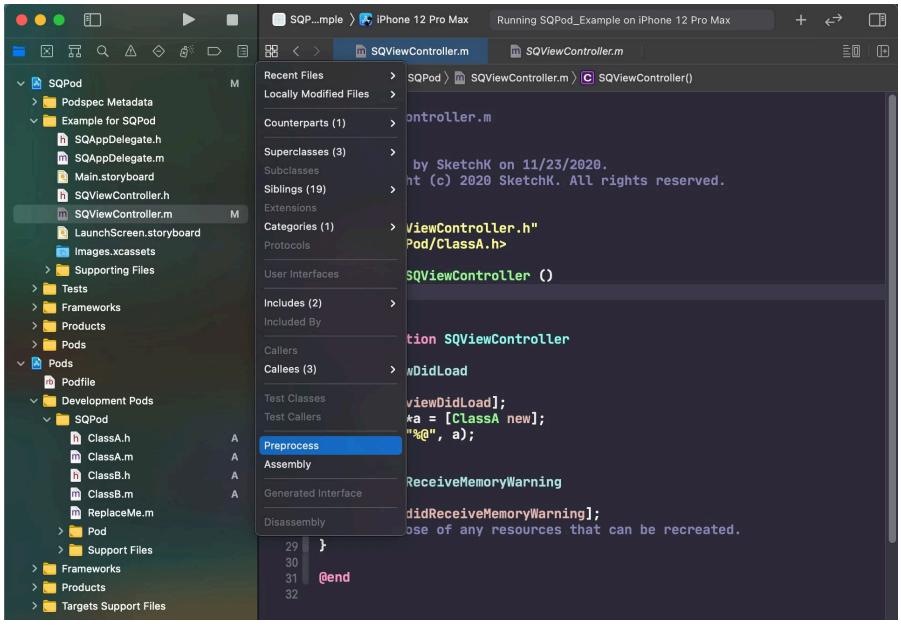
```
#import "SQViewController.h"
#import <SQPod/ClassA.h>

@interface SQViewController ()
@end

@implementation SQViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    ClassA *a = [ClassA new];
    NSLog(@"%@", a);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
@end
```

想要查看代码预编译后的样子，我们可以在 `Navigate to Related Items` 按钮中找到 `Preprocess` 选项



既然知道了如何查看预编译后的样子，我们不妨看看代码在使用 `#import`, PCH 和 `@import` 后，到底会变成什么样子?

这里我们假设被引入的头文件，即 ClassA 中的内如下：

```
@interface ClassA : NSObject
@property (nonatomic, strong) NSString *name;
- (void)sayHello;
@end
```

通过 preprocess 可以看到代码大致如下，这里为了方便展示，将无用代码进行了删除。这里记得要将 Build Setting 中 Packaging 的 `Define Module` 设置为 NO，因为其默认值为 YES，而这会导致我们开启 Clang Module 特性。

```
@import UIKit;
@interface SQViewController : UIViewController
@end

@interface ClassA : NSObject
@property (nonatomic, strong) NSString *name;
- (void)sayHello;
@end

@interface SQViewController ()
@end

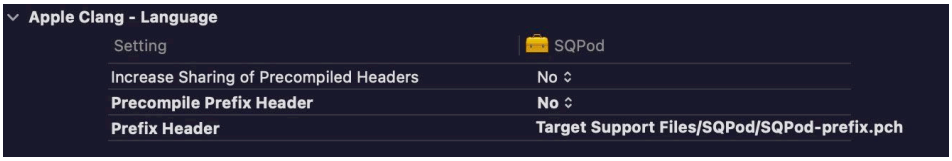
@implementation SQViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    ClassA *a = [ClassA new];
    NSLog(@"%@", a);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
@end
```

这么一看，`#import` 的作用还就真的是个 Copy & Write。

PCH 的真容

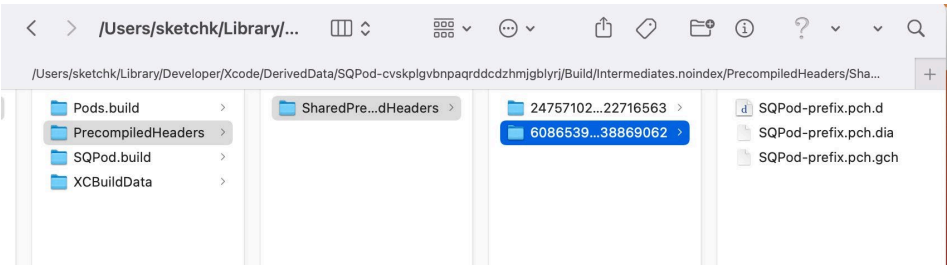
对于 CocoaPods 默认创建的组件，一般都会关闭 PCH 的相关功能，例如笔者创建的 SQPod 组件，它的 `Precompile Prefix Header` 功能默认值为 NO。



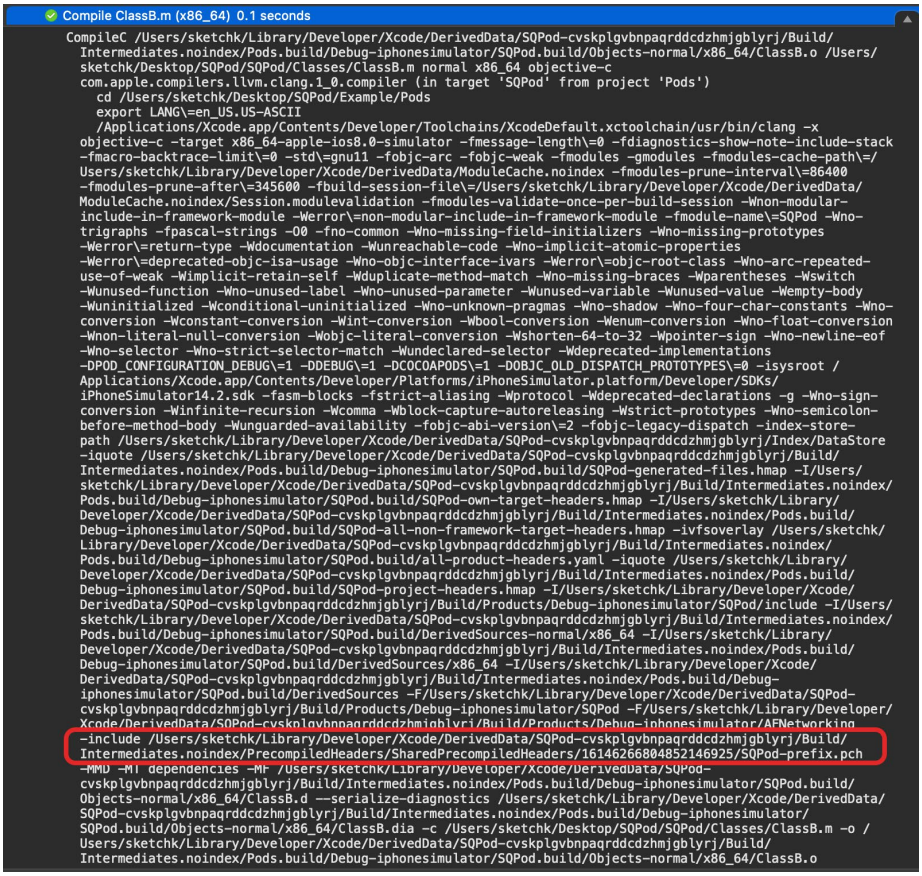
为了查看预编译的效果，我们将 `Precompile Prefix Header` 的值改为 YES，并编译整个项目，通过查看 Build Log，我们可以发现相比于 NO 的状态，在编译的过程中，增加了一个步骤，即 `Precompile SQPod-Prefix.pch` 的步骤。



通过查看这个命令的 `-o` 参数，我们可以知道其产物是名为 `SQPod-Prefix.pch.gch` 的文件。

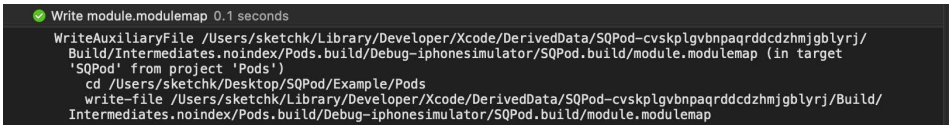


这个文件就是 PCH 预编译后的产物，同时在编译真正的代码时，会通过 `-include` 参数将其引入。



又见 Clang Module

在开启 Define Module 后，系统会为我们自动创建相应的 modulemap 文件，这一点可以在 Build Log 中查找到。



它的内容如下：

```
framework module SQPod {
  umbrella header "SQPod-umbrella.h"

  export *
  module * { export * }
}
```

当然，如果系统自动生成的 modulemap 并不能满足你的诉求，我们也可以使用自己创建的文件，此时只需要在 Build Setting 的 Module Map File 选项中填写好文件路径，相应的 clang 命令参数是 `-fmodule-map-file`。

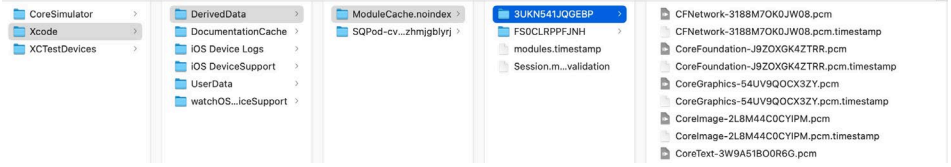


最后让我们看看 Module 编译后的产物形态。

这里我们构建一个名为 SQPod 的 Module ，将它提供给名为 Example 的工程使用，通过查看 `-fmodule-cache-path` 的参数，我们可以找到 Module 的缓存路径。



进入对应的路径后，我们可以看到如下的文件：



其中后缀名为 `pcm` 的文件就是构建出来的二进制中间产物。

现在，我们不仅知道了预编译的基础理论知识，也动手查看了预编译环节在真实环境下的产物，现在我们要开始解答之前提到的两个问题了！

打破砂锅问到底

关于第一个问题

> 对于未开启 Clang Module 特性的组件，Clang 是通过怎样的机制查找到头文件的呢？在查找系统头文件和非系统头文件的过程中，有什么区别么？

在早期的 Clang 编译过程中，头文件的查找机制还是基于 Header Seach Path 的，这也是大多数人所熟知的工作机制，所以我们不做赘述，只做一个简单的回顾。

Header Search Path 是构建系统提供给编译器的一个重要参数，它的作用是在编译代码的时候，为编译器提供了查找相应头文件路径的信息，通过查阅 Xcode 的 Build System 信息，我们可以知道相关的设置有三处 Header Search Path、System Header Search Path、User Header Search Path。



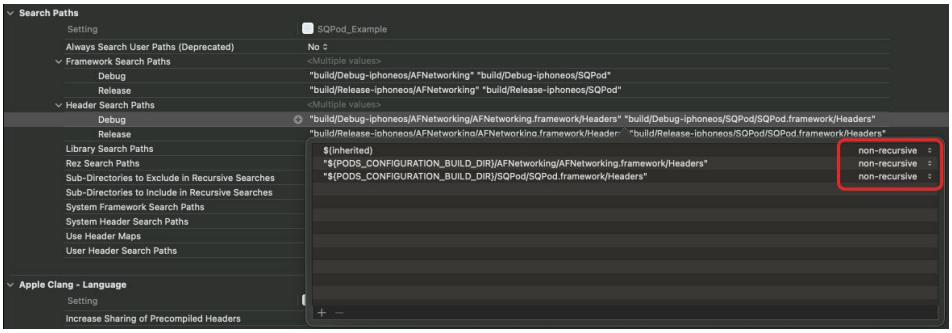
它们的区别也很简单，System Header Search Path 是针对系统头文件的设置，通常代指 `<>` 方式引入的文件，uUser Header Search Path 则是针对非系统头文件的设置，通常代指 `""` 方式引入的文件，而 Header Search Path 并不会有任何限制，它普适于任何方式的头文件引用。

听起来好像很复杂，但关于引入的方式，无非是以下四种形式：

```
#import <A/A.h>
#import "A/A.h"
#import <A.h>
#import "A.h"
```

我们可以两个维度去理解这个问题，一个是引入的符号形式，另一个是引入的内容形式。

- 引入的符号形式：通常来说，双引号的引入方式（`"A.h"` 或者 `"A/A.h"`）是用于查找本地的头文件，需要指定相对路径，尖括号的引入方式（`<A.h>` 或者 `<A/A.h>`）是全局的引用，其路径由编译器提供，如引用系统的库，但随着 Header Search Path 的加入，让这种区别已经被淡化了。
- 引入的内容形式：对于 `X/X.h` 和 `X.h` 这两种引入的内容形式，前者是说在对应的 Search Path 中，找到目录 A 并在 A 目录下查找 `A.h`，而后者是说在 Search Path 下查找 `A.h` 文件，而不一定局限在 A 目录中，至于是否递归的寻找则取决于对目录的选项是否开启了 `recursive` 模式

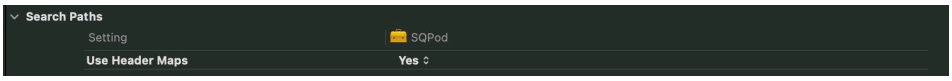


在很多工程中，尤其是基于 CocoaPods 开发的项目，我们已经不会区分 System Header Search Path 和 User Header Search Path，而是一股脑的将所有头文件路径添加到 Header Search Path 中，这就导致我们在引用某个头文件时，不会再局限于前面提到的约定，甚至在某些情况下，前面提到的四种方式都可以做到引入某个指定头文件。

Header Maps

随着项目的迭代和发展，原有的头文件索引机制还是受到了一些挑战，为此，Clang 官方也提出了自己的解决方案。

为了理解这个东西，我们首先要在 Build Setting 中开启 Use Header Map 选项。



然后在 Build Log 里获取相应组件里对应文件的编译命令，并在最后加上 -v 参数，来查看其运行的秘密：

```
$ clang <list of arguments> -c SQViewController.m -o SQViewcontroller.o -v
```

在 console 的输出内容中，我们会发现一段有意思的内容：

```
#include "..." search starts here:
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-generated-files.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-project-headers.hmap (headermap)
#include <...> search starts here:
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-own-target-headers.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/PodA_Example-all-non-framework-target-headers.hmap (headermap)
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Products/Debug-iphonesimulator/PodA/PodA.framework/Headers
 /Users/sketchk/Library/Developer/Xcode/DerivedData/PodA-crhgcfyqirwidfdliyxbsp
iprubx/Build/Intermediates.noindex/PodA.build/Debug-iphonesimulator/PodA_Exampl
e.build/DerivedSources
```

通过上面的图，我们可以看到编译器将寻找头文件的顺序和对应路径展示出来了，而在这些路径中，我们看到了一些陌生的东西，即后缀名为 `.hmap` 的文件。

那 hmap 到底这是个什么东西呢?

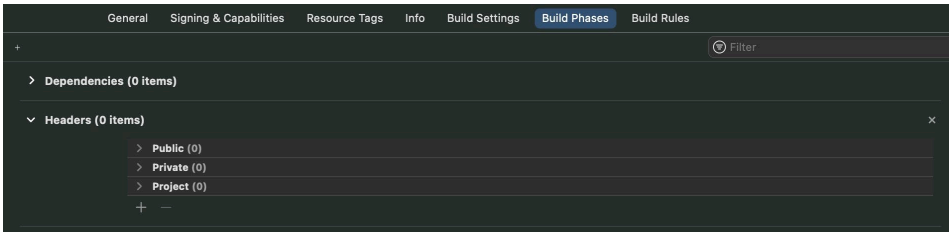
当我们开启 Build Setting 中的 Use Header Map 选项后，会自动生成的一份头文件名和头文件路径的映射表，而这个映射表就是 hmap 文件，不过它是一种二进制格式的文件，也有人叫它为 Header Map。总之，它的核心功能就是让编译器能够找到相应头文件的位置。

为了更好的理解它，我们可以通过 milend 编写的小工具 hmap 来查其内容。

在执行相关命令（即 `hmap print`）后，我们可以发现这些 hmap 里保存的信息结构大致如下：

```
sketchk ~/Desktop/Host git main
▶ hmap print /Users/sketchk/Library/Developer/Xcode/DerivedData/Host-ewmvmvyzlhizmsau
byxgeyqcnucf/Build/Intermediates.noindex/Pods.build/Debug-iphonesimulator/PodA.build/
PodA-own-target-headers.hmap
ClassA.h -> PodA/ClassA.h
ClassB.h -> PodA/ClassB.h
ClassF.h -> /Users/sketchk/Desktop/PodA/PodA/Classes/C/ClassF.h
PodA-umbrella.h -> PodA/PodA-umbrella.h
PodA/ClassF.h -> /Users/sketchk/Desktop/PodA/PodA/Classes/C/ClassF.h
```

需要注意，映射表的键值并不是简单的文件名和绝对路径，它的内容会随着使用场景产生不同的变化，例如头文件引用是在 `"..."` 的形式，还是 `<...>` 的形式，又或是在 Build Phase 里 Header 的配置情况。



至此，我想你应该明白了，一旦开启 Use Header Map 选项后，Xcode 会优先去 hmap 映射表里寻找头文件的路径，只有在找不到的情况下，才会去 Header Search Path 中提供的路径遍历搜索。

当然这种技术也不是一个什么新鲜事儿，在 Facebook 的 buck 工具中也提供了类似的东西，只不过文件类型变成了 `HeaderMap.java` 的样子。

查找系统库的头文件

上面的过程让我们理解了在 Header Map 技术下，编译器是如何寻找相应的头文件的，那针对系统库的文件又是如何索引的呢？例如 `#import <Foundation/Foundation.h>`

回想一下上一节 console 的输出内容，它的形式大概如下：

```
#include "..." search starts here:
XXX-generated-files.hmap (headermap)
XXX-project-headers.hmap (headermap)

#include <...> search starts here:
XXX-own-target-headers.hmap (headermap)
```

```
XXX-all-target-headers.hmap (headermap)
Header Search Path
DerivedSources
Build/Products/Debug (framework directory)
$(SDKROOT)/usr/include
$(SDKROOT)/System/Library/Frameworks(framework directory)
```

我们会发现，这些路径大部分是用于查找非系统库文件的，也就是开发者自己引入的头文件，而与系统库相关的路径只有以下两个：

```
#include <...> search starts here:
$(SDKROOT)/usr/include
$(SDKROOT)/System/Library/Frameworks.(framework directory)
```

当我们查找 `Foundation/Foundation.h` 这个文件的时候，我们会首先判断是否存在 Foundation 这个 Framework。

```
$SDKROOT/System/Library/Frameworks/Foundation.framework
```

接着，我们会进入 Framework 的 Headers 文件夹里寻找对应的头文件。

```
$SDKROOT/System/Library/Frameworks/Foundation.framework/Headers/
Foundation.h
```

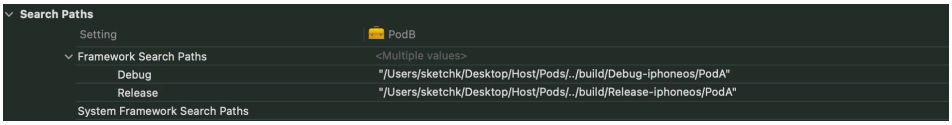
如果没有找到对应的文件，索引过程会在此中断，并结束查找。

以上便是系统库的头文件搜索逻辑。

Framework Search Path

到现在为止，我们已经解释了如何依赖 Header Search Path、hmap 等技术寻找头文件的工作机制，也介绍了寻找系统库（System Framework）头文件的工作机制。

那这是全部头文件的搜索机制么？答案是否定的，其实我们还有一种头文件搜索机制，它是基于 Framework 这种文件结构进行的。

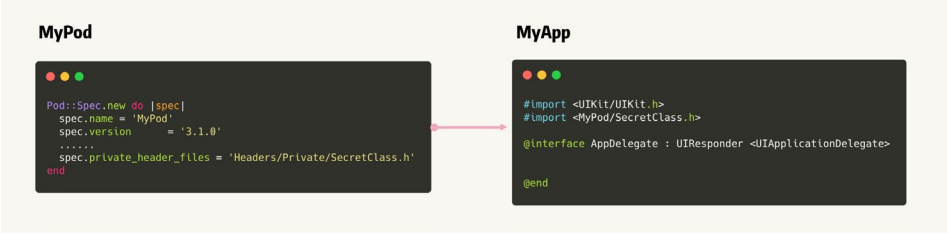


对于开发者自己的 Framework，可能会存在“private”头文件，例如在 podspec 里用 `private_header_files` 的描述文件，这些文件在构建的时候，会被放在 Framework 文件结构中的 PrivateHeaders 目录。

所以针对有 PrivateHeaders 目录的 Framework 而言，Clang 在检查 Headers 目录后，会去 PrivateHeaders 目录中寻找是否存在匹配的头文件，如果这两个目录都没有，才会结束查找。

```
$SDKROOT/System/Library/Frameworks/Foundation.framework/PrivateHeaders/
SecretClass.h
```

不过也正是因为这个工作机制，会产生一个特别有意思的问题，那就是当我们使用 Framework 的方式引入某个带有“Private”头文件的组件时，我们总是可以以下面的方式引入这个头文件！



怎么样，是不是很神奇，这个被描述为“Private”的头文件怎么就不私有了?

究其原因，还是由于 Clang 的工作机制，那为什么 Clang 要设计出来这种看似很奇怪的工作机制呢?

揭开 Public、Private、Project 的真实面目

其实你也看到，我在上一段的写作中，将所有 Private 单词标上了双引号，其实就是在暗示，我们曲解了 Private 的含义。

那么这个“Private”到底是什么意思呢?

在 Apple 官方的 Xcode Help – What are build phases? 文档中，我们可以看到如

下的一段解释：

> Associates public, private, or project header files with the target. Public and private headers define API intended for use by other clients, and are copied into a product for installation. For example, public and private headers in a framework target are copied into Headers and Private-Headers subfolders within a product. Project headers define API used and built by a target, but not copied into a product. This phase can be used once per target.

总的来说，我们可以知道一点，就是 Build Phases - Headers 中提到 Public 和 Private 是指可以供外界使用的头文件，且分别放在最终产物的 Headers 和 PrivateHeaders 目录中，而 Project 中的头文件是不对外使用的，也不会放在最终的产物中。

如果你继续翻阅一些资料，例如 StackOverflow - Xcode: Copy Headers: Public vs. Private vs. Project? 和 StackOverflow - Understanding Xcode's Copy Headers phase，你会发现在早期 Xcode Help 的 Project Editor 章节里，有一段名为 Setting the Role of a Header File 的段落，里面详细记载了三个类型的区别。

> **Public:** The interface is finalized and meant to be used by your product's clients. A public header is included in the product as readable source code without restriction. **Private:** The interface isn't intended for your clients or it's in early stages of development. A private header is included in the product, but it's marked "private". Thus the symbols are visible to all clients, but clients should understand that they're not supposed to use them. **Project:** The interface is for use only by implementation files in the current project. A project header is not included in the target, except in object code. The symbols are not visible to clients at all, only to you.

至此，我们应该彻底了解了 Public、Private、Project 的区别。简而言之，Public

还是通常意义上的 Public，Private 则代表 In Progress 的含义，至于 Project 才是通常意义上的 Private 含义。

那么 CocoaPods 中 Podspec 的 Syntax 里还有 `public_header_files` 和 `private_header_files` 两个字段，它们的真实含义是否和 Xcode 里的概念冲突呢?

这里我们仔细阅读一下官方文档的解释，尤其是 `private_header_files` 字段。

private_header_files multi-platform

A list of file patterns that should be used to mark private headers.

These patterns are matched against the public headers (or all the headers if no public headers have been specified) to exclude those headers which should not be exposed to the user project and which should not be used to generate the documentation. When the library is built, these headers will appear in the build directory.

Header files that are not listed as neither public nor private will be treated as private, but in addition will not appear in the build directory at all.

Examples:

```
spec.private_header_files = 'Headers/Private/*.h'
```

我们可以看到，`private_header_files` 在这里的含义是说，它本身是相对于 Public 而言的，这些头文件本义是不希望暴露给用户使用的，而且也不会产生相关文档，但是在构建的时候，会出现在最终产物中，只有既没有被 Public 和 Private 标注的头文件，才会被认为是真正的私有头文件，且不出现在最终的产物里。

其实这么看来，CocoaPods 对于 Public 和 Private 的理解是和 Xcode 中的描述一致的，两处的 Private 并非我们通常理解的 Private，它的本意更应该是开发者准备对外开放，但又没完全 Ready 的头文件，更像一个 In Progress 的含义。

所以，如果你真的不想对外暴露某些头文件，请不要再使用 Headers 里的 Private 或者 podspec 里的 `private_header_files` 了。

至此，我想你应该彻底理解了 Search Path 的搜索机制和略显奇怪的 Public、Private、Project 设定了！

基于 hmap 优化 Search Path 的策略

在查找系统库的头文件的章节中，我们通过 `-v` 参数看到了寻找头文件的搜索顺序：

```
#include "..." search starts here:
XXX-generated-files.hmap (headermap)
XXX-project-headers.hmap (headermap)

#include <...> search starts here:
XXX-own-target-headers.hmap (headermap)
XXX-all-target-headers.hmap (headermap)
Header Search Path
DerivedSources
Build/Products/Debug (framework directory)
$(SDKROOT)/usr/include
$(SDKROOT)/System/Library/Frameworks(framework directory)
```

假设，我们没有开启 hmap 的话，所有的搜索都会依赖 Header Search Path 或者 Framework Search Path，那这就会出现 3 种问题：

- 第一个问题，在一些巨型项目中，假设依赖的组件有 400+，那此时的索引路径就会达到 800+ 个（一份 Public 路径，一份 Private 路径），同时搜索操作可以看做是一种 IO 操作，而我们知道 IO 操作通常也是一种耗时操作，那么，这种大量的耗时操作必然会导致编译耗时增加。
- 第二个问题，在打包的过程中，如果 Header Search Path 过多过长，会触发命令行过长的错误，进而导致命令执行失败的情况。
- 第三个问题，在引入系统库的头文件时，Clang 会将前面提到的目录遍历完才进入搜索系统库的路径，也就是 `$(SDKROOT)/System/Library/Frameworks(framework directory)`，即前面的 Header Search 路径越多，耗时也会越长，这是相当不划算的。

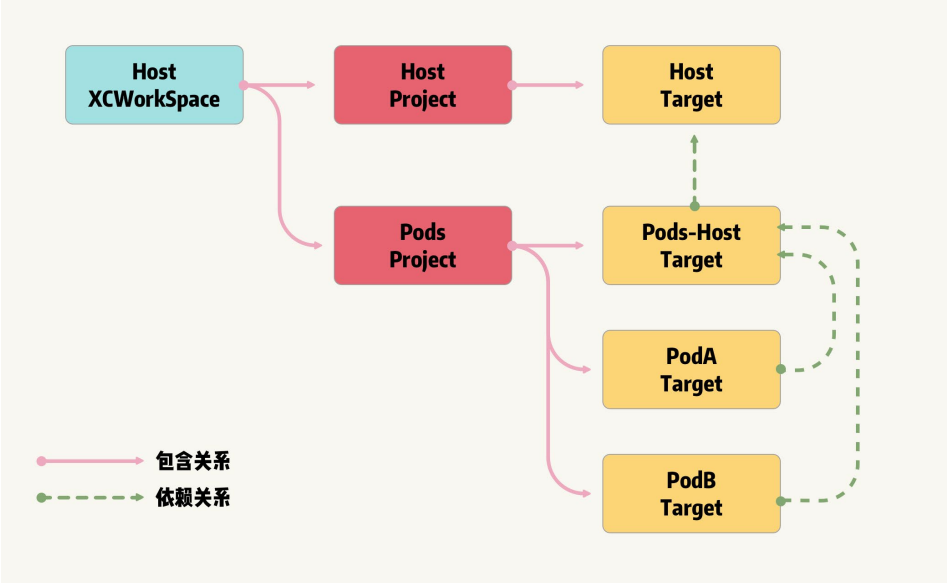
那如果我们开启 hmap 后，是否就能解决掉所有的问题呢?

实际上并不能，而且在基于 CocoaPods 管理项目的状况下，又会带来新的问题。下面是一个基于 CocoaPods 构建的全源码工程项目，它的整体结构如下：

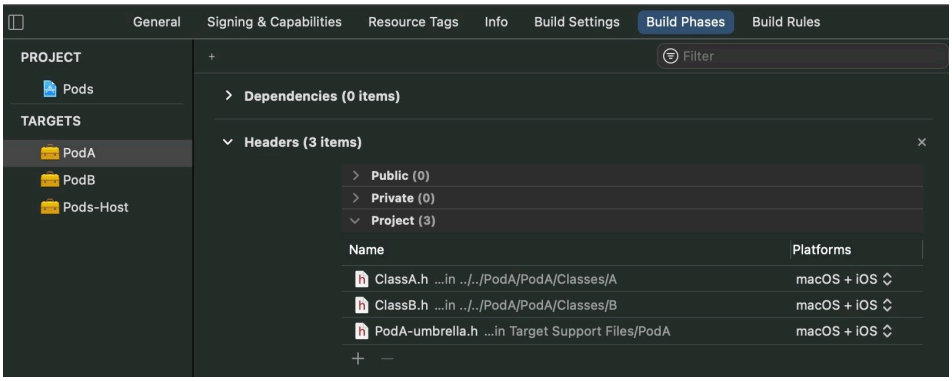
首先，Host 和 Pod 是我们的两个 Project，Pods 下的 Target 的产物类型为 Static Library。

其次，Host 底下会有一个同名的 Target，而 Pods 目录下会有 n+1 个 Target，其中 n 取决于你依赖的组件数量，而 1 是一个名为 Pods-XXX 的 Target，最后，Pods-XXX 这个 Target 的产物会被 Host 里的 Target 所依赖。

整个结构看起来如下所示：



此时我们将 PodA 里的文件全部放在 Header 的 Project 类型中。



在基于 Framework 的搜索机制下，我们是无法以任何方式引入到 ClassB 的，因为它既不在 Headers 目录，也不在 PrivateHeader 目录中。

可是如果我们开启了 Use Header Map 后，由于 PodA 和 PodB 都在 Pods 这个 Project 下，满足了 Header 的 Project 定义，通过 Xcode 自动生成的 hmap 文件会带上这个路径，所以我们还可以在 PodB 中以 `#import "ClassB.h"` 的方式引入。

而这种行为，我想应该是大多数人并不想要的结果，所以一旦开启了 Use Header Map，再结合 CocoaPods 管理工程项目的模式，我们极有可能会产生一些误用私有头文件的情况，而这个问题的本质是 Xcode 和 CocoaPods 在工程和头文件上的理念冲突造成的。

除此之外，CocoaPods 在处理头文件的问题上还有一些让人迷惑的地方，它在创建头文件产物这块的逻辑大致如下：

- 在构建产物为 Framework 的情况下
  - 根据 podspec 里的 `public_header_files` 字段的内容，将相应头文件设置为 Public 类型，并放在 Headers 中。
  - 根据 podspec 里的 `private_header_files` 字段的内容，将相应文件设置为 Private 类型，并放在 PrivateHeader 中。
  - 将其余未描述的头文件设置为 Project 类型，且不放入最终的产物中。
  - 如果 podspec 里未标注 Public 和 Private 的时候，会将所有文件设置为

Public 类型，并放在 Header 中。

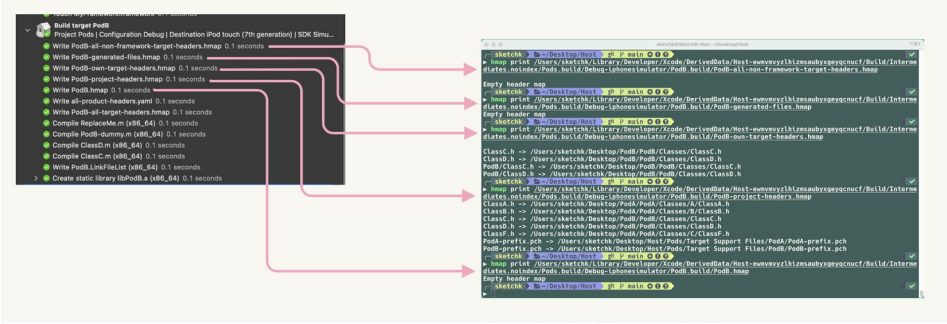
- 在构建产物为 Static Library 的情况下
  - 不论 podspec 里如何设置 `public_header_files` 和 `private_header_files`，相应的头文件都会被设置为 Project 类型。
  - 在 `Pods/Headers/Public` 中会保存所有被声明为 `public_header_files` 的头文件。
  - 在 `Pods/Headers/Private` 中会保存所有头文件，不论是 `public_header_files` 或者 `private_header_files` 描述到，还是那些未被描述的，这个目录下是当前组件的所有头文件全集。
  - 如果 podspec 里未标注 Public 和 Private 的时候，`Pods/Headers/Public` 和 `Pods/Headers/Private` 的内容一样且会包含所有头文件。

正是由于这种机制，还导致了另外一种有意思的问题。

在 Static Library 的状况下，一旦我们开启了 Use Header Map，结合组件里所有头文件的类型为 Project 的情况，这个 hmap 里只会包含 `#import "A.h"` 的键值引用，也就是说只有 `#import "A.h"` 的方式才会命中 hmap 的策略，否则都将通过 Header Search Path 寻找其相关路径。

而我们也知道，在引用其他组件的时候，通常都会采用 `#import <A/A.h>` 的方式引入。至于为什么会用这种方式，一方面是这种写法会明确头文件的由来，避免问题，另一方面也是这种方式可以让我们在是否开启 Clang Module 中随意切换，当然还有一点就是，Apple 在 WWDC 里曾经不止一次建议开发者使用这种方式来引入头文件。

接着上面的话题来说，所以说在 Static Library 的情况下且以 `#import <A/A.h>` 这种标准方式引入头文件时，开启 Use Header Map 并不会提升编译速度，而这同样是 Xcode 和 CocoaPods 在工程和头文件上的理念冲突造成的。



这样来看的话，虽然 hmap 有种种优势，但是在 CocoaPods 的世界里显得格格不入，也无法发挥自身的优势。

那这就真的没有办法解决了么?

当然，问题是有办法解决的，我们完全可以自己动手做一个基于 CocoaPods 规则下的 hmap 文件。

举一个简单的例子，通过遍历 PODS 目录里的内容去构建索引表内容，借助 hmap 工具生成 header map 文件，然后将 Cocoapods 在 Header Search Path 中生成的路径删除，只添加一条指向我们自己生成的 hmap 文件路径，最后关闭 Xcode 的 Ues Header Map 功能，也就是 Xcode 自动生成 hmap 的功能，如此这般，我们就实现了一个简单的，基于 CocoaPods 的 Header Map 功能。

同时在这个基础上，我们还可以借助这个功能实现不少管控手段，例如：

- 从根本上杜绝私有文件被暴露的可能性。
- 统一头文件的引用形式
- ...

目前，我们已经自研了一套基于上述原理的 cocoapods 插件，它的名字叫做 cocoapods-hmap-prebuilt，是由笔者与同事共同开发的。

说了这么多，让我们看看它在实际工程中的使用效果!

经过全源码编译的测试，我们可以看到该技术在提速上的收益较为明显，以美团和点

评 App 为例，全链路时长能够提升 45% 以上，其中 Xcode 打包时间能提升 50%。

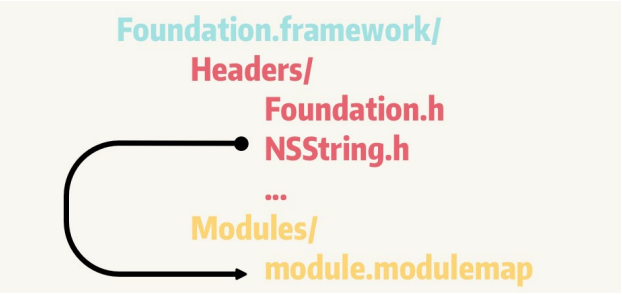
关于第二个问题

> 对于已开启 Clang Module 特性的组件，Clang 是如何决定编译当下组件的 Module 呢？另外构建的细节又是怎样的，以及如何查找这些 Module 的？还有查找系统的 Module 和非系统的 Module 有什么区别么？

首先，我们来明确一个问题，Clang 是如何决定编译当下组件的 Module 呢？

以 `#import <Foundation/NSString.h>` 为例，当我们遇到这个头文件的时候：

首先会去 Framework 的 Headers 目录下寻找相应的头文件是否存在，然后就会到 Modules 目录下查找 modulemap 文件。



此时，Clang 会去查阅 modulemap 里的内容，看看 NSString 是否为 Foundation 这个 Module 里的一部分。

```
// Module Map - Foundation.framework/Modules/module.modulemap
framework module Foundation [extern_c] [system] {
    umbrella header "Foundation.h"
    export *
    module * {
        export *
    }

    explicit module NSDebug {
        header "NSDebug.h"
        export *
    }
}
```

很显然，这里通过 Umbrella Header，我们是可以在 Foundation.h 中找到 NSString.h 的。

```
// Foundation.h
...
#import <Foundation/NSStream.h>
#import <Foundation/NSString.h>
#import <Foundation/NSTextCheckingResult.h>
...
```

至此，Clang 会判定 `NSString.h` 是 Foundation 这个 Module 的一部分并进行相应的编译工作，此时也就意味着 `#import <Foundation/NSString.h>` 会从之前的 textual import 变为 module import。

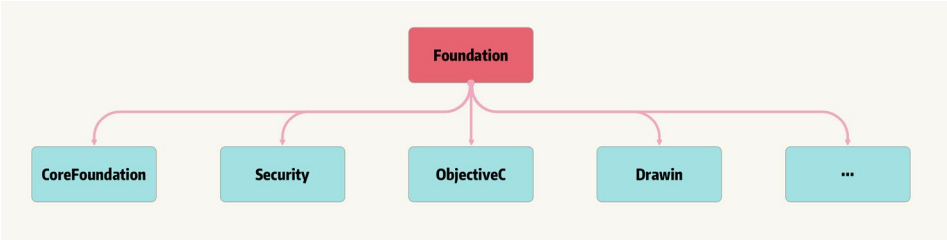
Module 的构建细节

上面的内容解决了是否构建 Module，而这一块我们会详细阐述构建 Module 的过程！

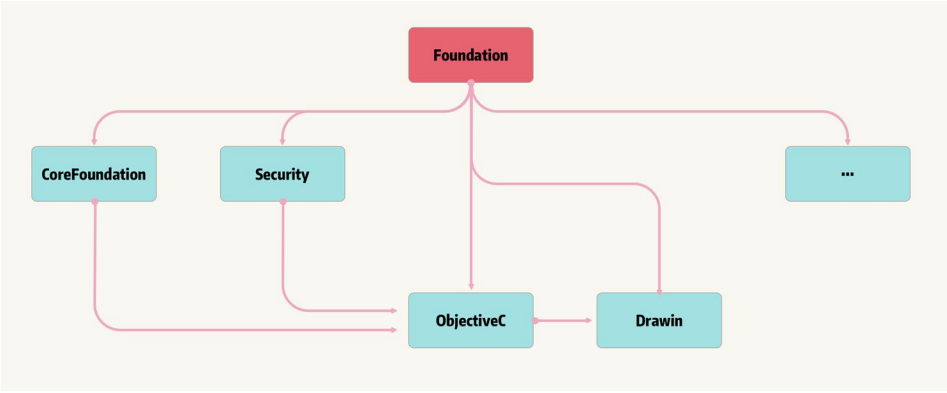
在构建开始前，Clang 会创建一个完全独立的空间来构建 Module，在这个空间里会包含 Module 涉及的所有文件，除此之外不会带入其他任何文件的信息，而这也是 Module 健壮性好的关键因素之一。

不过，这并不意味着我们无法影响到 Module 的唯一性，真正能影响到其唯一性的是其构建的参数，也就是 Clang 命令后面的内容，关于这一点后面还会继续展开，这里我们先点到为止。

当我们在构建 Foundation 的时候，我们会发现 Foundation 自身要依赖一些组件，这意味着我们也需要构建被依赖组件的 Module。



但很明显的是，我们会发现这些被依赖组件也有自己的依赖关系，在它们的这些依赖关系中，极有可能会存在重复的引用。



此时，Module 的复用机制就体现出来优势了，我们可以复用先前构建出来的 Module，而不必一次次的创建或者引用，例如 Drawin 组件，而保存这些缓存文件的位置就是前面章节里提到的保存 `pcm` 类型文件的地方。

先前我们提到了 Clang 命令的参数会真正影响到 Module 的唯一性，那具体的原理又是怎样的?

Clang 会将相应的编译参数进行一次 Hash，将获得的 Hash 值作为 Module 缓存文件夹的名称，这里需要注意的是，不同的参数和值会导致文件夹不同，所以想要尽可能的利用 Module 缓存，就必须保证参数不发生变化。

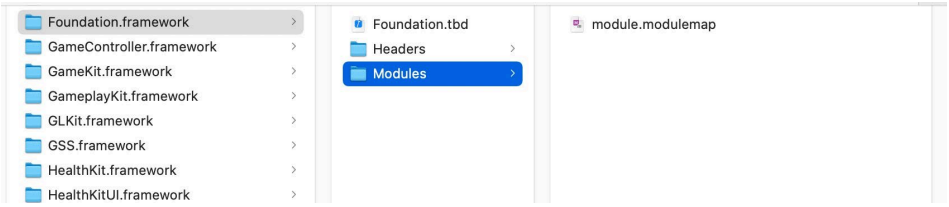
```
$ clang -fmodules —DENABLE_FEATURE=1 …
## 生成的目录如下
98XN8P5QH5OQ/
  CoreFoundation-2A5I5R2968COJ.pcm
  Security-1A229VWPAK67R.pcm
  Foundation-1RDF848B47PF4.pcm

$ clang -fmodules —DENABLE_FEATURE=2 …
## 生成的目录如下
1GYDULU5XJRF/
  CoreFoundation-2A5I5R2968COJ.pcm
  Security-1A229VWPAK67R.pcm
  Foundation-1RDF848B47PF4.pcm
```

这里我们大概了解了系统组件的 module 构建机制，这也是开启 `Enable Modules(C and Objective-C)` 的核心工作原理。

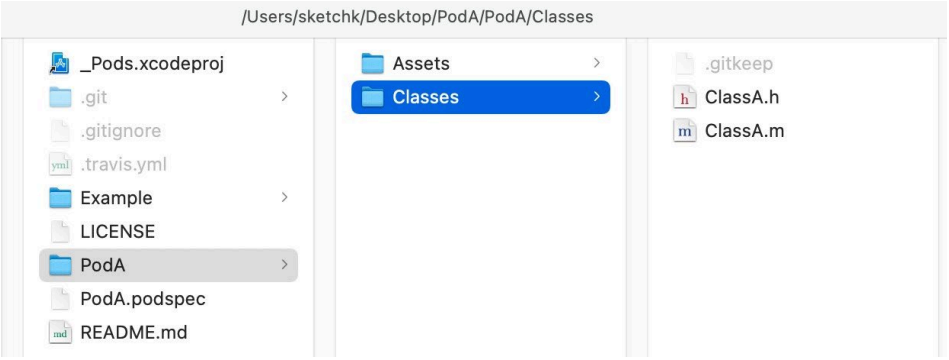
神秘的 Virtual File System（VFS）

对于系统组件，我们可以在 `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator14.2.sdk/System/Library/Frameworks` 目录里找到它的身影，它的目录结构大概是这样的：



也就是说，对于系统组件而言，构建 Module 的整个过程是建立在这样一个完备的文件结构上，即在 Framework 的 Modules 目录中查找 modulemap，在 Headers 目录中加载头文件。 那对于用户自己创建的组件，Clang 又是如何构建 Module 的呢?

通常我们的开发目录大概是下面的样子，它并没有 Modules 目录，也没有 Headers 目录，更没有 modulemap 文件，看起来和 Framework 的文件结构也有着极大的区别。



在这种情况下，Clang 是没法按照前面所说的机制去构建 Module 的，因为在这种文件结构中，压根就没有 Modules 和 Headers 目录。

为了解决这个问题，Clang 又提出了一个新的解决方案，叫做 Virtual File System（VFS）。

简单来说，通过这个技术，Clang 可以在现有的文件结构上虚拟出来一个 Framework 文件结构，进而让 Clang 遵守前面提到的构建准则，顺利完成 Module 的编译，同时 VFS 也会记录文件的真实位置，以便在出现问题的时候，将文件的真实信息暴露给用户。

为了进一步了解 VFS，我们还是从 Build Log 中查找一些细节！



在上面的编译参数里，我们可以找到一个 `-ivfsoverlay` 的参数，查看 Help 说明，可以知道其作用就是向编译器传递一个 VFS 描述文件并覆盖掉真实的文件结构信息。

```
-ivfsoverlay <value>    Overlay the virtual filesystem described by file
over the real file system
```

顺着这个线索，我们去看看这个参数指向的文件，它是一个 yaml 格式的文件，在将内容进行了一些裁剪后，它的核心内容如下：

```
{
  "case-sensitive": "false",
  "version": 0,
  "roots": [
    {
      "name": "XXX/Debug-iphonesimulator/PodA/PodA.framework/Headers",
      "type": "directory",
      "contents": [
        { "name": "ClassA.h", "type": "file",
          "external-contents": "XXX/PodA/PodA/Classes/ClassA.h"
        },
        ......
        { "name": "PodA-umbrella.h", "type": "file",
          "external-contents": "XXX/Target Support Files/PodA/PodA-
umbrella.h"
        }
      ]
    },
    {
      "contents": [
        "name": "XXX/Products/Debug-iphonesimulator/PodA/PodA.
framework/Modules",
        "type": "directory"
        { "name": "module.modulemap", "type": "file",
          "external-contents": "XXX/Debug-iphonesimulator/PodA.build/
module.modulemap"
        }
      ]
    }
  ]
}
```

结合前面提到的内容，我们不难看出它在描述这样一个文件结构：

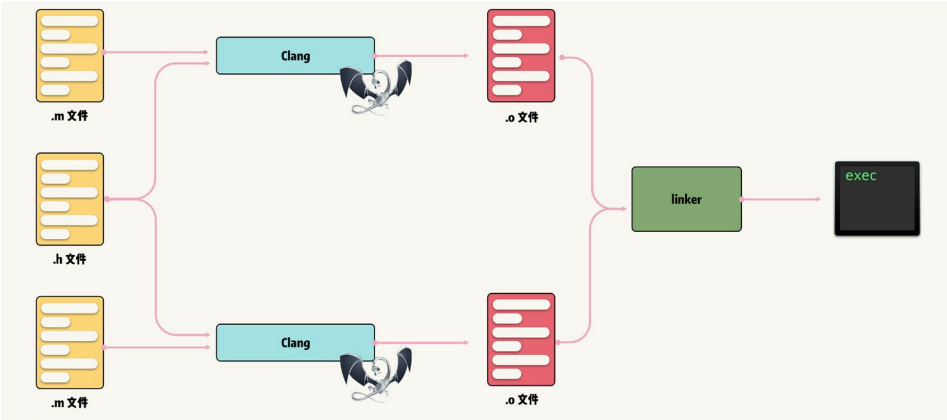
借用一个真实存在的文件夹来模拟 Framework 里的 Headers 文件夹，在这个 Headers 文件夹里有名为 `PodA-umbrella.h` 和 `ClassA.h` 等的文件，不过这几个虚拟文件与 `external-contents` 指向的真实文件相关联，同理还有 Modules 文件夹和它里面的 `module.modulemap` 文件。

通过这样的形式，一个虚拟的 Framework 目录结构诞生了！此时 Clang 终于能按照前面的构建机制为用户创建 Module 了！

Swift 来了

没有头文件的 Swift

前面的章节，我们聊了很多 C 语言系的预编译知识，在这个体系下，文件的编译是分开的，当我们想引用其他文件里的内容时，就必须引入相应的头文件。



而对于 Swift 这门语言来说，它并没有头文件的概念，对于开发者而言，这确实省去了写头文件的重复工作，但这也意味着，编译器会进行额外的操作来查找接口定义并需要持续关注接口的变化！

为了更好的解释 Swift 和 Objective-C 是如何寻找到彼此的方法声明的，我们这里引入一个例子，在这个例子由三个部分组成：

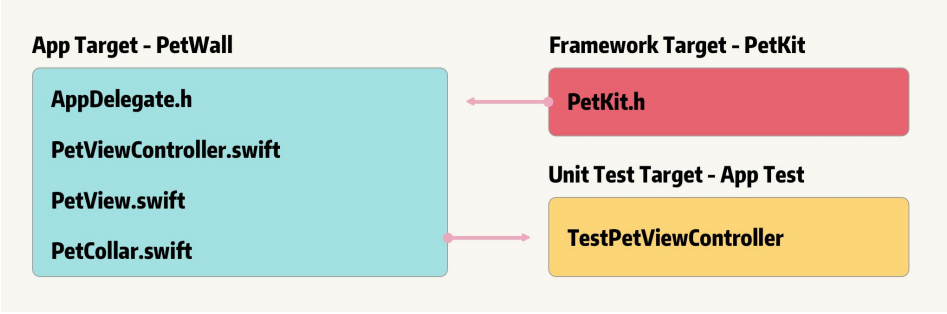
- 第一部分是一个 ViewController 的代码，它里面包含了一个 View，其中 PetViewController 和 PetView 都是 Swift 代码。
- 第二部分是一个 App 的代理，它是 Objective-C 代码。
- 第三个部分是一段单测代码，用来测试第一个部分中的 ViewController，它是 Swift 代码。

```
import UIKit
class PetViewController: UIViewController {
  var view = PetView(name: "Fido", frame: frame)
  …
}
```

```
#import "PetWall-Swift.h"
@implementation AppDelegate
…
@end
```

```
@testable import PetWall
class TestPetViewController: XCTestCase {
}
```

它们的关系大致如下所示：



为了能让这些代码编译成功，编译器会面对如下 4 个场景：

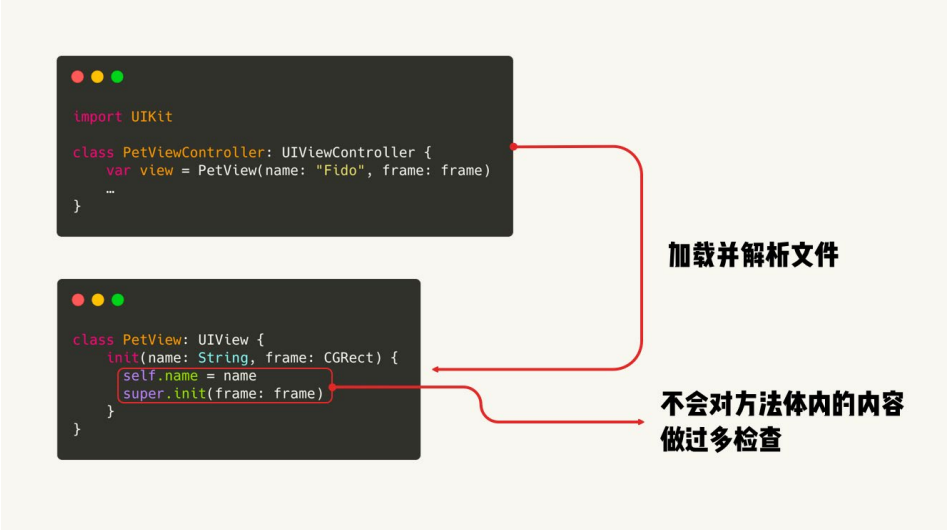
首先是寻找声明，这包括寻找当前 Target 内的方法声明（PetView），也包括来自 Objective-C 组件里的声明（UIViewController 或者 PetKit）。

然后是生成接口，这包括被 Objective-C 使用的接口，也包括被其他 Target（Unit Test）使用的 Swift 接口。

第一步 – 如何寻找 Target 内部的 Swift 方法声明

在编译 `PetViewController.swift` 时，编译器需要知道 PetView 的初始化构造器的类型，才能检查调用是否正确。

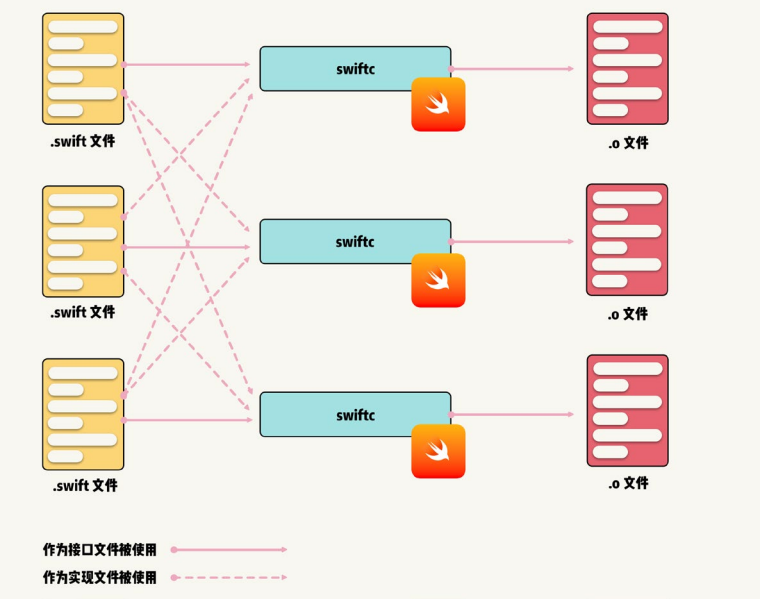
此时，编译器会加载 `PetView.swift` 文件并解析其中的内容，这么做的目的就是确保初始化构造器真的存在，并拿到相关的类型信息，以便 `PetViewController.swift` 进行验证。



编译器并不会对初始化构造器的内部做检查，但它仍然会进行一些额外的操作，这是什么意思呢?

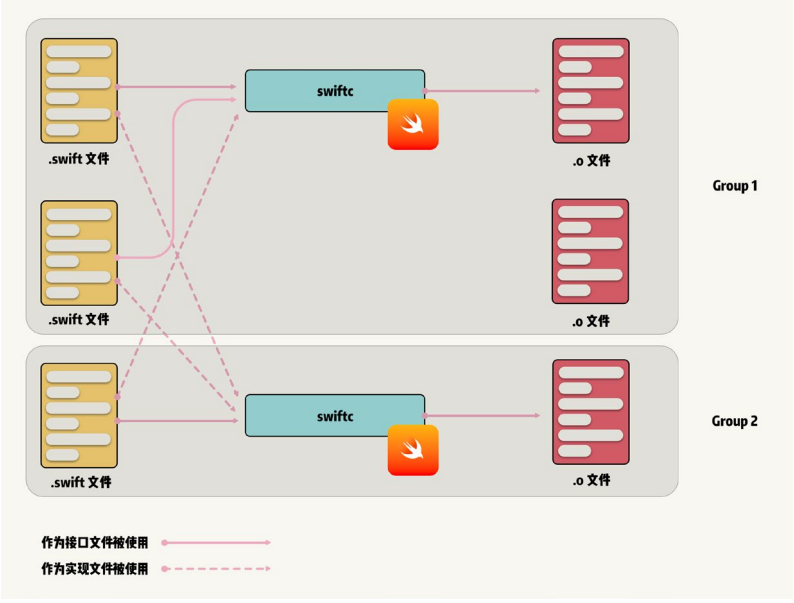
与 Clang 编译器不同的是，Swiftc 编译的时候，会将相同 Target 里的其他 Swift 文件进行一次解析，用来检查其中与被编译文件关联的接口部分是否符合预期。

同时我们也知道，每个文件的编译是独立的，且不同文件的编译是可以并行开展的，所以这就意味着每编译一个文件，就需要将当前 Target 里的其余文件当做接口，重新编译一次。等于任意一个文件，在整个编译过程中，只有 1 次被作为生产 `.o` 产物的输入，其余时间会被作为接口文件反复解析。



不过在 Xcode 10 以后，Apple 对这种编译流程进行了优化。

在尽可能保证并行的同时，将文件进行了分组编译，这样就避免了 Group 内的文件重复解析，只有不同 Group 之间的文件会有重复解析文件的情况。



而这个分组操作的逻辑，就是刚才提到的一些额外操作。

至此，我们应该了解了 Target 内部是如何寻找 Swift 方法声明的了。

第二步 – 如何找到 Objective-C 组件里的方法声明

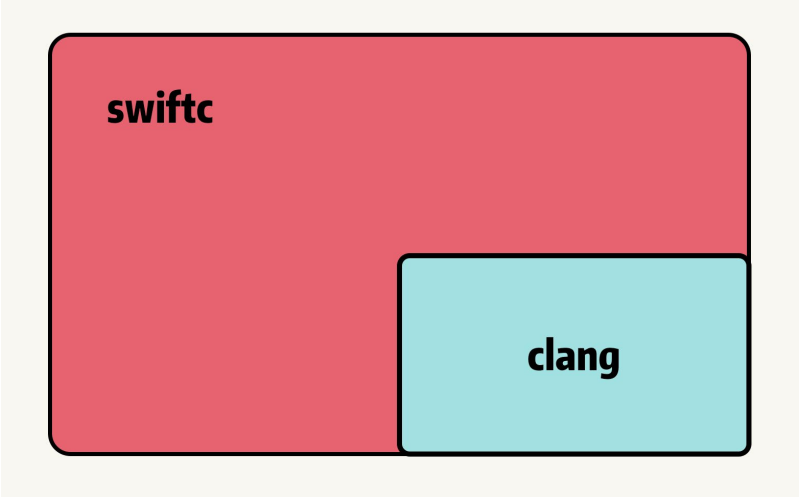
回到第一段代码中，我们可以看到 PetViewController 是继承自 UIViewController，而这也意味着我们的代码会与 Objective-C 代码进行交互，因为大部分系统库，例如 UIKit 等，还是使用 Objective-C 编写的。

在这个问题上，Swift 采用了和其他语言不一样的方案！

通常来说，两种不同的语言在混编时需要提供一个接口映射表，例如 JavaScript 和 TypeScript 混编时候的 `.d.ts` 文件，这样 TypeScript 就能够知道 JavaScript 方法在 TS 世界中的样子。

然而，Swift 不需要提供这样的接口映射表，免去了开发者为每个 Objective-C API 声明其在 Swift 世界里样子，那它是怎么做到的呢？

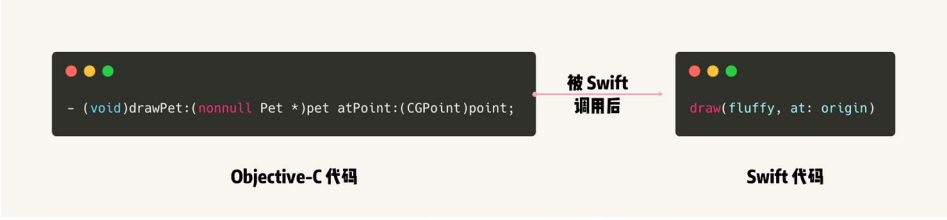
很简单，Swift 编译器将 Clang 的大部分功能包含在其自身的代码中，这就使得我们能够以 Module 的形式，直接引用 Objective-C 的代码。



既然是通过 Module 的形式引入 Objective-C，那么 Framework 的文件结构则是最好的选择，此时编译器寻找方法声明的方式就会有下面三种场景：

- 对于大部分的 Target 而言，当导入的是一个 Objective-C 类型的 Framework 时，编译器会通过 modulemap 里的 Header 信息寻找方法声明。
- 对于一个既有 Objective-C，又有 Swift 代码的 Framework 而言，编译器会从当前 Framework 的 Umbrella Header 中寻找方法声明，从而解决自身的编译问题，这是因为通常情况下 modulemap 会将 Umbrella Header 作为自身的 Header 值。
- 对于 App 或者 Unit Test 类型的 Target，开发者可以通过为 Target 创建 Briding Header 来导入需要的 Objective-C 头文件，进而找到需要的方法声明。

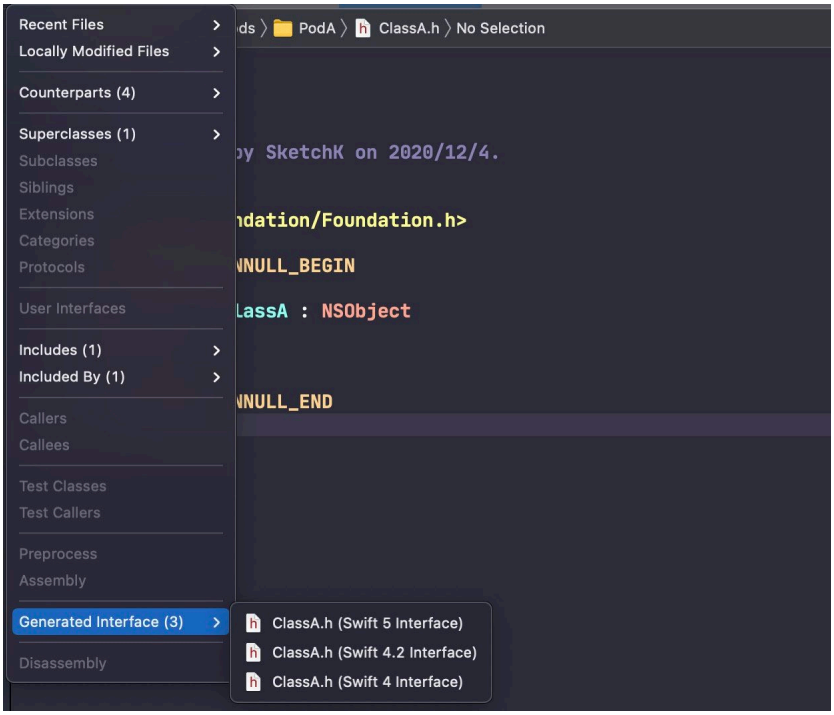
不过我们应该知道 Swift 编译器在获取 Objective-C 代码过程中，并不是原原本本的将 Objective-C 的 API 暴露给 Swift，而是会做一些“Swift 化”的改动，例如下面的 Objective-C API 就会被转换成更简约的形式。



这个转换过程并不是什么高深的技术，它只是在编译器上的硬编码，如果感兴趣，可以在 Swift 的开源库中的找到相应的代码 - PartsOfSpeech.def

当然，编译器也给与了开发者自行定义“API 外貌”的权利，如果你对这一块感兴趣，不妨阅读我的另一篇文章 - WWDC20 10680 - Refine Objective-C frameworks for Swift，那里面包含了很多重塑 Objective-C API 的技巧。

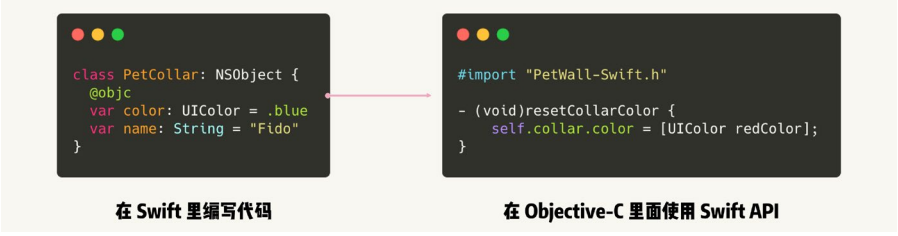
不过这里还是要提一句，如果你对生成的接口有困惑，可以通过下面的方式查看编译器为 Objective-C 生成的 Swift 接口。



第三步 – Target 内的 Swift 代码是如何为 Objective-C 提供接口的

前面讲了 Swift 代码是如何引用 Objective-C 的 API，那么 Objective-C 又是如何引用 Swift 的 API 呢?

从使用层面来说，我们都知道 Swift 编译器会帮我们自动生成一个头文件，以便 Objective-C 引入相应的代码，就像第二段代码里引入的 `PetWall-Swift.h` 文件，这种头文件通常是编译器自动生成的，名字的构成是 `组件名 -Swift` 的形式。



但它到底是怎么产生的呢?

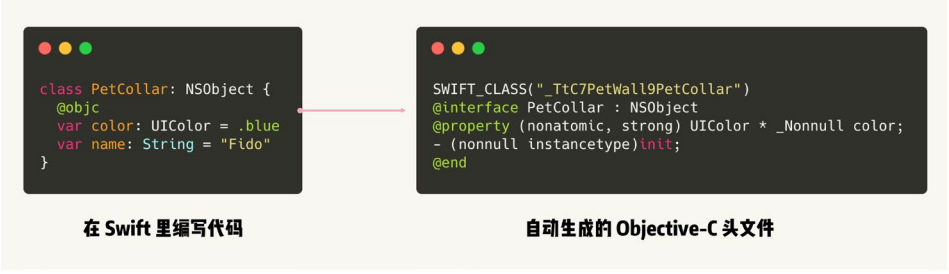
在 Swift 中，如果某个类继承了 NSObject 类且 API 被 `@objc` 关键字标注，就意味着它将暴露给 Objective-C 代码使用。

不过对于 App 和 Unit Test 类型的 target 而言，这个自动生成的 Header 会包含访问级别为 Public 和 internal 的 API，这使得同一 Target 内的 Objective-C 代码也能访问 Swift 里 internal 类型的 API，这也是所有 Swift 代码的默认访问级别。

但对于 Framework 类型的 Target 而言，Swift 自动生成的头文件只会包含 Public 类型的 API，因为这个头文件会被作为构建产物对外使用，所以像 internal 类型的 API 是不会包含在这个文件中。

> 注意，这种机制会导致在 Framework 类型的 Target 中，如果 Swift 想暴露一些 API 给内部的 Objective-C 代码使用，就意味着这些 API 也必须暴露给外界使用，即必须将其访问级别设置为 Public。

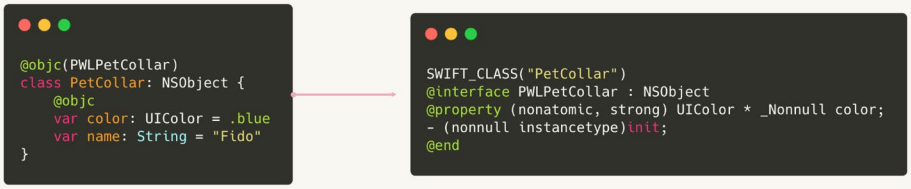
那么编译器自动生成的 API 到底是什么样子，有什么特点呢?



上面是截取了一段自动生成的头文件代码，左侧是原始的 Swift 代码，右侧是自动生成的 Objective-C 代码，我们可以看到在 Objective-C 的类中，有一个名为 `SWIFT_CLASS` 的宏，将 Swift 与 Objective-C 中的两个类进行了关联。

如果你稍加注意，就会发现关联的一段乱码中还绑定了当前的组件名（PetWall），这样做的目的是避免两个组件的同名类在运行时发生冲突。

当然，你也可以通过向 `@objc(Name)` 关键字传递一个标识符，借由这个标识符来控制其在 Objective-C 中的名称，如果这样做的话，需要开发者确保转换后的类名不与其他类名出现冲突。



自定义 Swift 在 Objective-C 的类名

这大体上就是 Swift 如何像 Objective-C 暴露接口的机理了，如果你想更深入的了解这个文件的由来，就需要看看第四步。

第四步 – Swift Target 如何生成供外部 Swift 使用的接口

Swift 采用了 Clang module 的理念，并结合自身的语言特性进行了一系列的改进。

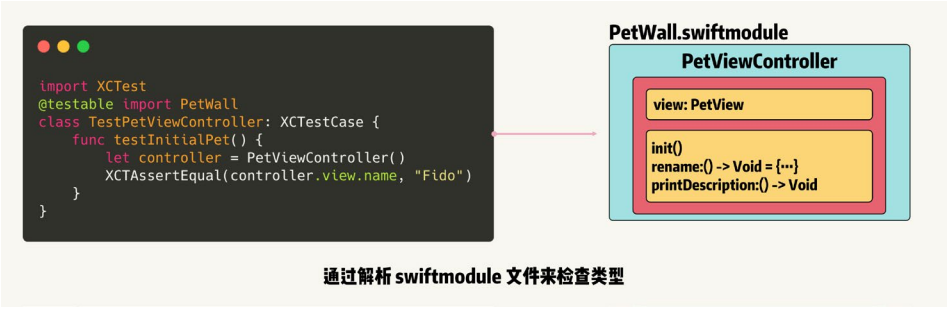
在 Swift 中，Module 是方法声明的分发单位，如果你想引用相应的方法，就必须引入对应的 Module，之前我们也提到了 Swift 的编译器包含了 Clang 的大部分内容，所以它也是兼容 Clang Module 的。

所以我们可以引入 Objective-C 的 Module，例如 XCTest，也可以引入 Swift Target 生成的 Module，例如 PetWall。

```
import XCTest
@testable import PetWall
class TestPetViewController: XCTestCase {
  func testInitialPet() {
    let controller = PetViewController()
    XCTAssertEqual(controller.view.name, "Fido")
  }
}
```

在引入 swift 的 Module 后，编译器会反序列化一个后缀名为 `.swiftmodule` 的文件，并通过这种文件里的内容来了解相关接口的信息。

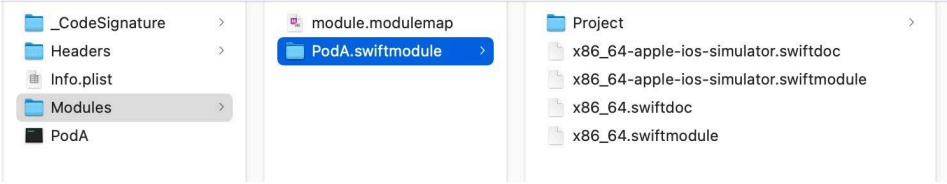
例如，以下图为例，在这个单元测试中，编译器会加载 PetWall 的 Module，并在其中找寻 PetViewController 的方法声明，由此确保其创建行为是符合预期的。



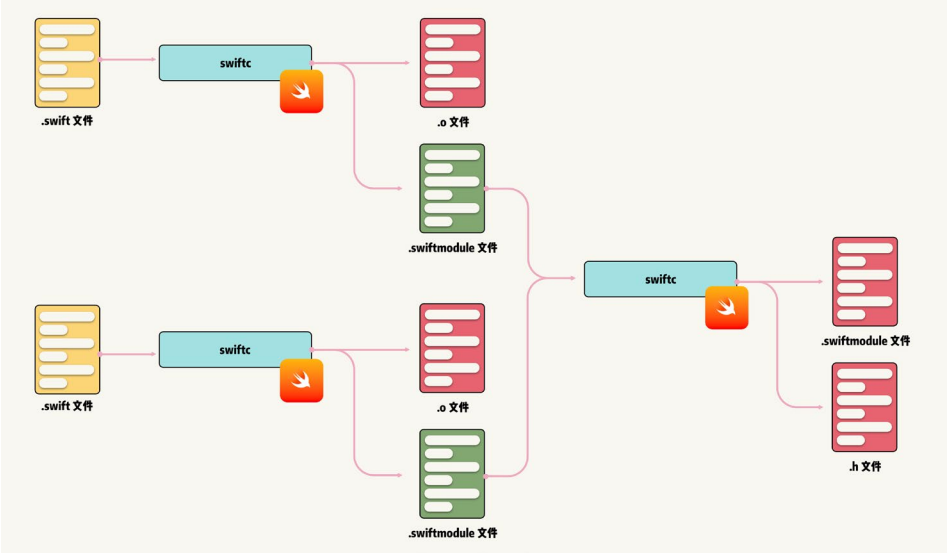
通过解析 swiftmodule 文件来检查类型

这看起来很像第一步中 Target 寻找内部 Swift 方法声明的样子，只不过这里将解析 Swift 文件的步骤，换成了解析 Swiftmodule 文件而已。

不过需要注意的是，这个 Swfitmodule 文件并不是文本文件，它是一个二进制格式的内容，通常我们可以在构建产物的 Modules 文件夹里寻找到它的身影。



在 Target 的编译的过程中，面向整个 Target 的 Swiftmodule 文件并不是一下产生的，每一个 Swift 文件都会生成一个 Swiftmodule 文件，编译器会将这些文件进行汇总，最后再生成一个完整的，代表整个 Target 的 Swiftmodule，也正是基于这个文件，编译器构造出了用于给外部使用的 Objective-C 头文件，也就是第三步里提到的头文件。

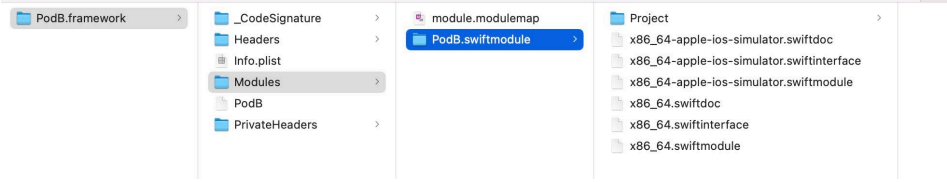


不过随着 Swift 的发展，这一部分的工作机制也发生了些许变化。

我们前面提到的 Swiftmodule 文件是一种二进制格式的文件，而这个文件格式会包含一些编译器内部的数据结构，不同编译器产生的 Swiftmodule 文件是互相不兼容的，这也就导致了不同 Xcode 构建出的产物是无法通用的，如果对这方面的细节感兴趣，可以阅读 Swift 社区里的两篇官方 Blog：Evolving Swift On Apple Platforms After ABI Stability 和 ABI Stability and More，这里就不展开讨论了。

为了解决这一问题，Apple 在 Xcode 11 的 Build Setting 中提供了一个新的编译参数 Build Libraries for Distribution，正如这个编译参数的名称一样，当我们开启它后，构建出来的产物不会再受编译器版本的影响，那它是怎么做到这一点的呢?

为了解决这种对编译器的版本依赖，Xcode 在构建产物上提供了一个新的产物，Swiftinterface 文件。



这个文件里的内容和 Swiftmodule 很相似，都是当前 Module 里的 API 信息，不过 Swiftinterface 是以文本的方式记录，而非 Swiftmodule 的二进制方式。

这就使得 Swiftinterface 的行为和源代码一样，后续版本的 Swift 编译器也能导入之前编译器创建的 Swiftinterface 文件，像使用源码的方式一样使用它。

为了更进一步了解它，我们来看看 Swiftinterface 的真实样子，下面是一个 `.swift` 文件和 `.swiftinterface` 文件的比对图。

.swift 文件

```swift
import UIKit

public class Spaceship {
  public let name: String
  private var currentLocation: Location

  public init(name: String) {
    self.name = name
    currentLocation = .launchpad
  }.

  public func fly(
    to destination: Location,
    speed: Speed) {
    currentLocation = destination
  }.
}.

public enum Speed {
  case leisurely
  case fast
}.

public struct Location {
  public var coordinates: Coordinates
}.
```

.swiftinterface文件

```
// swift-interface-format-version: 1.0
// swift-compiler-version: Swift version 5.1
// swift-module-flags:  -target arm64-apple-ios13.0
//                      -enable-library-evolution
//                      -swift-version 5 -O
//                      -module-name FlightKit

import Swift
import UIKit

public class Spaceship {
  public let name: Swift.String

  public init(name: Swift.String)
  public func fly(
    to destination: FlightKit.Location,
    speed: FlightKit.Speed)

  @objc deinit
}.

public enum Speed: Swift.Hashable {
  case leisurely
  case fast

  public static func ==(
    a: FlightKit.Speed,
    b: FlightKit.Speed) -> Swift.Bool
  public func hash(
    into hasher: inout Swift.Hasher)
}

public struct Location {
  public var coordinates: FlightKit.Coordinates
}
```

在 Swiftinterface 文件中，有以下点需要注意

- 文件会包含一些元信息，例如文件格式版本，编译器信息，和 Swift 编译器将其作为模块导入所需的命令行子集。
- 文件只会包含 Public 的接口，而不会包含 Private 的接口，例如 currentLocation。
- 文件只会包含方法声明，而不会包含方法实现，例如 Spacesship 的 init、fly 等方法。

- 文件会包含所有隐式声明的方法，例如 Spacesship 的 deinit 方法，Speed 的 Hashable 协议。

总的来说，Swiftinterface 文件会在编译器的各个版本中保持稳定，主要原因就是这个接口文件会包含接口层面的一切信息，不需要编译器再做任何的推断或者假设。

好了，至此我们应该了解了 Swift Target 是如何生成供外部 Swift 使用的接口了。

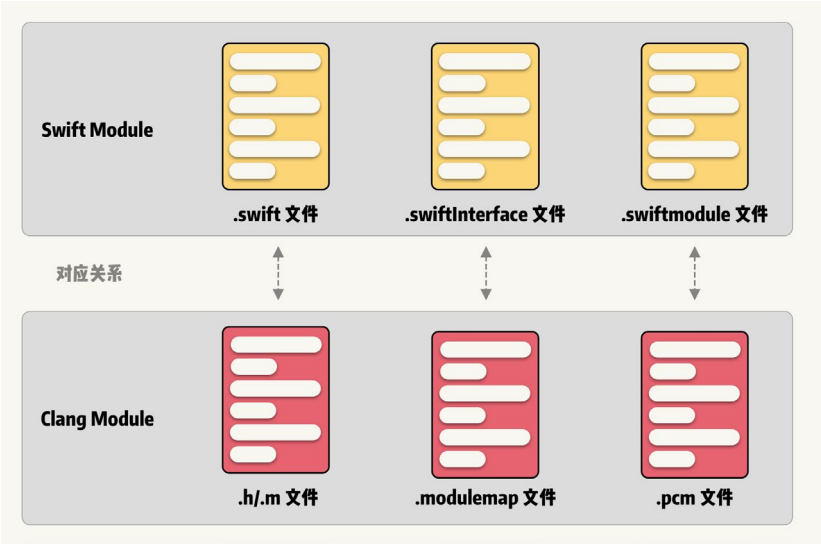
这四步意味着什么?

此 Module 非彼 Module

通过上面的例子，我想大家应该能清楚的感受到 Swift Module 和 Clang Module 不完全是一个东西，虽然它们有很多相似的地方。

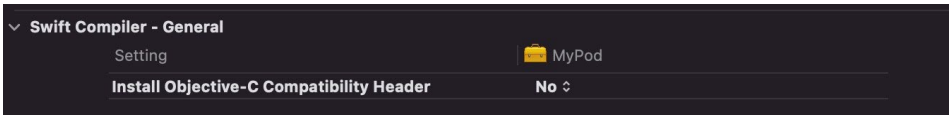
Clang Module 是面向 C 语言家族的一种技术，通过 modulemap 文件来组织 `.h` 文件中的接口信息，中间产物是二进制格式的 pcm 文件。

Swift Module 是面向 Swift 语言的一种技术，通过 Swiftinterface 文件来组织 `.swift` 文件中的接口信息，中间产物二进制格式的 Swiftmodule 文件。

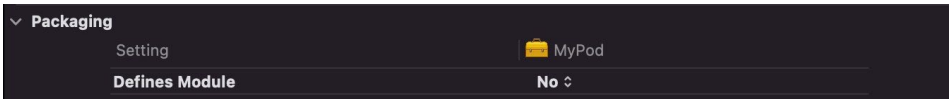


所以说理清楚这些概念和关系后，我们在构建 Swift 组件的产物时，就会知道哪些文件和参数不是必须的了。

例如当你的 Swift 组件不想暴露自身的 API 给外部的 Objective-C 代码使用的话，可以将 Build Setting 中 Swift Compiler - General 里的 Install Objective-C Compatiblity Header 参数设置为 NO，其编译参数为 `SWIFT_INSTALL_OBJC_HEADER`，此时不会生成 `<ProductModuleName>-Swift.h` 类型的文件，也就意味着外部组件无法以 Objective-C 的方式引用组件内 Swift 代码的 API。



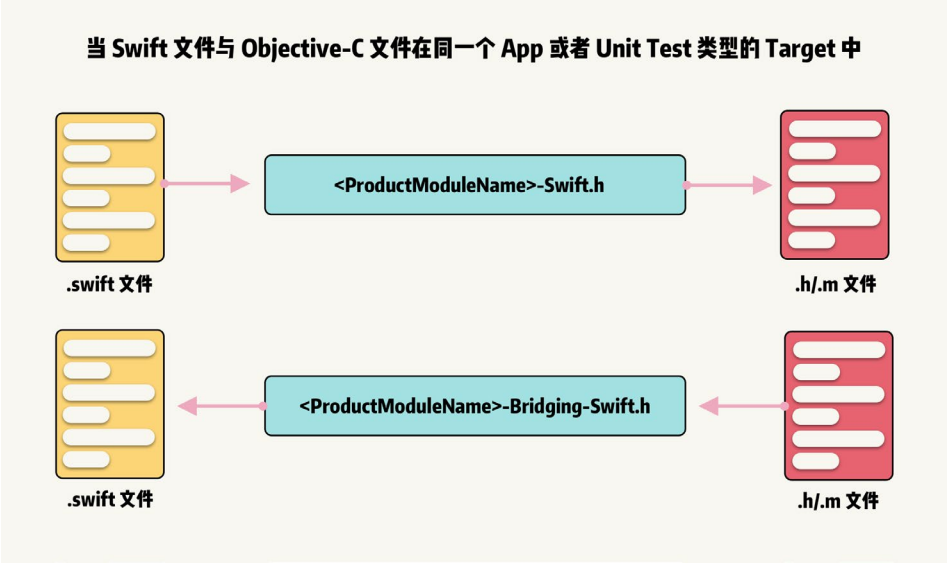
而当你的组件里如果压根就没有 Objective-C 代码的时候，你可以将 Build Setting 中 Packaging 里 Defines Module 参数设置为 NO，它的编译参数为 `DEFINES_MODULE`，此时不会生成 `<ProductModuleName>.modulemap` 类型的文件。



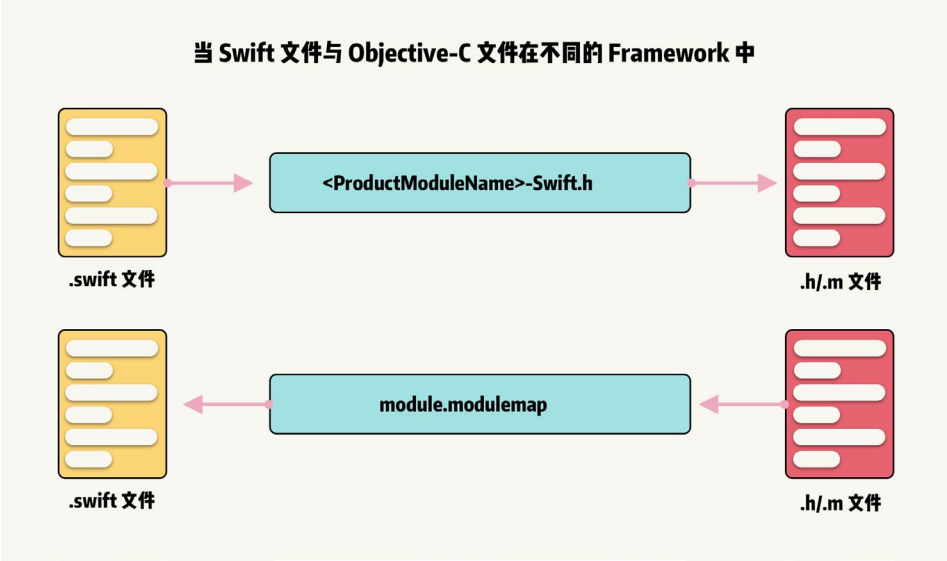
Swift 和 Objective-C 混编的三个“套路”

基于刚才的例子，我们应该理解了 Swift 在编译时是如何找到其他 API 的，以及它又是如何暴露自身 API 的，而这些知识就是解决混编过程中的基础知识，为了加深影响，我们可以将其绘制成 3 个流程图。

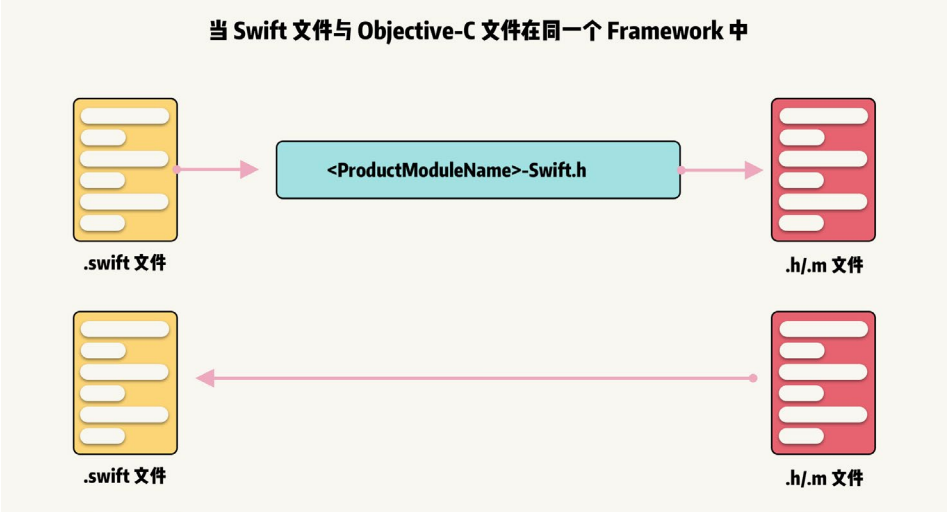
当 Swift 和 Objective-C 文件同时在一个 App 或者 Unit Test 类型的 Target 中，不同类型文件的 API 寻找机制如下：



当 Swift 和 Objective-C 文件在不同 Target 中，例如不同 Framework 中，不同类型文件的 API 寻找机制如下：



当 Swift 和 Objective-C 文件同时在一个 Target 中，例如同一 Framework 中，不同类型文件的 API 寻找机制如下：



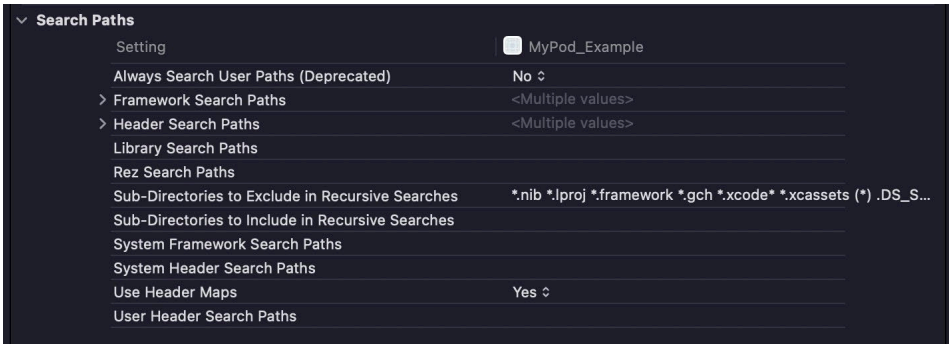
对于第三个流程图，需要做以下补充说明：

- 由于 Swiftc，也就是 Swift 的编译器，包含了大部分的 Clang 功能，其中就包含了 Clang Module，借由组件内已有的 modulemap 文件，Swift 编译器就可以轻松找到相应的 Objective-C 代码。
- 相比于第二个流程而言，第三个流程中的 modulemap 是组件内部的，而第二个流程中，如果想引用其他组件里的 Objective-C 代码，需要引入其他组件里的 modulemap 文件才可以。
- 所以基于这个考虑，并未在流程 3 中标注 modulemap。

构建 Swift 产物的新思路

在前面的章节里，我们提到了 Swift 找寻 Objective-C 的方式，其中提到了，除了 App 或者 Unit Test 类型的 Target 外，其余的情况下都是通过 Framework 的 Module Map 来寻找 Objective-C 的 API，那么如果我们不想使用 Framework 的形式呢?

目前来看，这个在 Xcode 中是无法直接实现的，原因很简单，Build Setting 中 Search Path 选项里并没有 modulemap 的 Search Path 配置参数。



为什么一定需要 modulemap 的 Search Path 呢?

基于前面了解到的内容，Swiftc 包含了 Clang 的大部分逻辑，在预编译方面，Swiftc 只包含了 Clang Module 的模式，而没有其他模式，所以 Objective-C 想要暴露自己的 API 就必须通过 modulemap 来完成。

而对于 Framework 这种标准的文件夹结构，modulemap 文件的相对路径是固定的，它就在 Modules 目录中，所以 Xcode 基于这种标准结构，直接内置了相关的逻辑，而不需要将这些配置再暴露出来。

从组件的开发者角度来看，他只需要关心 modulemap 的内容是否符合预期，以及路径是否符合规范。

从组件的使用者角度来看，他只需要正确的引入相应的 Framework 就可以使用到相应的 API。

这种只需要配置 Framework 的方式，避免了配置 Header Search Path，也避免了配置 Static Library Path，可以说是一种很友好的方式，如果再将 modulemap 的配置开放出来，反而显得多此一举。

那如果我们抛开 Xcode，抛开 Framework 的限制，还有别的办法构建 Swift 产物么?

答案是肯定有的，这就需要借助前面所说的 VFS 技术!

假设我们的文件结构如下所示：

```
├── LaunchPoint.swift
├── README.md
├── build
├── repo
│   └── MyObjcPod
│       └── UsefulClass.h
└── tmp
    ├── module.modulemap
    └── vfs-overlay.yaml
```

其中 `LaunchPoint.swift` 引用了 `UsefulClass.h` 中的一个公开 API，并产生了依赖关系。

另外，`vfs-overlay.yaml` 文件重新映射了现有的文件目录结构，其内容如下：

```
{
  'version': 0,
  'roots': [
    { 'name': '/MyObjcPod', 'type': 'directory',
      'contents': [
        { 'name': 'module.modulemap', 'type': 'file',
          'external-contents': 'tmp/module.modulemap'
        },
        { 'name': 'UsefulClass.h', 'type': 'file',
          'external-contents': 'repo/MyObjcPod/UsefulClass.h'
        }
      ]
    }
  ]
}
```

至此，我们通过如下的命令，便可以获得 LaunchPoint 的 Swiftmodule、Swiftinterface 等文件，具体的示例可以查看我在 Github 上的链接 – manually-expose-objective-c-API-to-swift-example

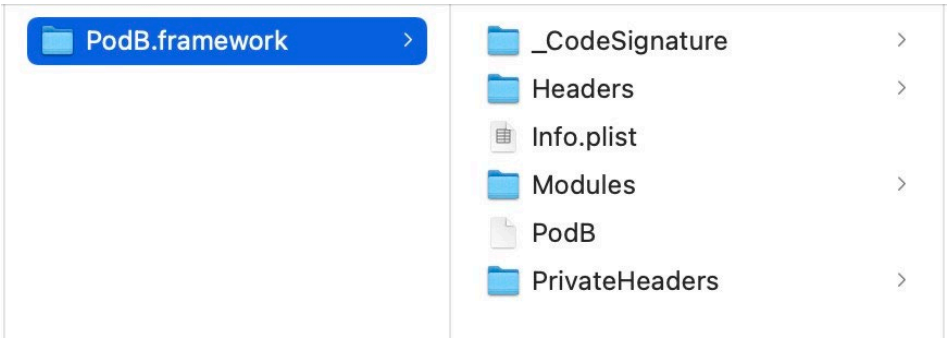
```
swiftc -c LaunchPoint.swift -emit-module -emit-module-path build/LaunchPoint.
swiftmodule -module-name index -whole-module-optimization -parse-as-library -o
build/LaunchPoint.o -Xcc -ivfsoverlay -Xcc tmp/vfs-overlay.yaml -I /MyObjcPod
```

那这意味着什么呢?

这就意味着，只提供相应的 `.h` 文件和 `.modulemap` 文件就可以完成 Swift 二进制产物的构建，而不再依赖 Framework 的实体。同时，对于 CI 系统来说，在构建产物时，可以避免下载无用的二进制产物（`.a` 文件），这从某种程度上会提升编译效率。

如果你没太理解上面的意思，我们可以展开说说。

例如，对于 PodA 组件而言，它自身依赖 PodB 组件，在使用原先的构建方式时，我们需要拉取 PodB 组件的完整 Framework 产物，这会包含 Headers 目录，Modules 目录里的必要内容，当然还会包含一个二进制文件（PodB），但在实际编译 PodA 组件的过程中，我们并不需要 B 组件里的二进制文件，而这让拉取完整的 Framework 文件显得多余了。



而借助 VFS 技术，我们就能避免拉取多余的二进制文件，进一步提升 CI 系统的编译效率。

总结

感谢你的耐心阅读，至此，整篇文章终于结束了，通过这篇文章，我想你应该：

- 理解 Objective-C 的三种预编译的工作机制，其中 Clang Module 做到了真正意义上的语义引入，提升了编译的健壮性和扩展性。
- 在 Xcode 的 Search Path 的各种技术细节使用到了 hmap 技术，通过加载映射表的方式避免了大量重复的 IO 操作，可以提升编译效率。

- 在处理 Framework 的头文件索引时，总是会先搜索 Headers 目录，再搜索 PrivateHeader 目录。
- 理解 Xcode Phases 构建系统中，Public 代表公开头文件，Private 代表不需要使用者感知，但物理存在的文件，而 Project 代表不应让使用者感知，且物理不存在的文件。
- 不使用 Framework 的情况下且以 `#import <A/A.h>` 这种标准方式引入头文件时，在 CocoaPods 上使用 hmap 并不会提升编译速度。
- 通过 `cocoapods-hmap-built` 插件，可以将大型项目的全链路时长节省 45% 以上，Xcode 打包环节的时长节省 50% 以上。
- Clang Module 的构建机制确保了其不受上下文影响（独立编译空间），复用效率高（依赖决议），唯一性（参数哈希化）。
- 系统组件通过已有的 Framework 文件结构实现了构建 Module 的基本条件，而非系统组件通过 VFS 虚拟出相似的 Framework 文件 结构，进而具备了编译的条件。
- 可以粗浅的将 Clang Module 里的 `.h/m`，`.moduelmap`，`.pch` 的概念对应为 Swift Module 里的 `.swift`，`.swiftinterface`，`.swiftmodule` 的概念
- 理解三种具有普适性的 Swift 与 Objective-C 混编方法
  - 同一 Target 内（App 或者 Unit 类型），基于 `<PorductModuleName>-Swift.h` 和 `<PorductModuleName>-Bridging-Swift.h`。
  - 同一 Target 内，基于 `<PorductModuleName>-Swift.h` 和 Clang 自身的能力。
  - 不同 Target 内，基于 `<PorductModuleName>-Swift.h` 和 `module.modulemap`。
- 利用 VFS 机制构建，可以在构建 Swift 产物的过程中避免下载无用的二进制产物，进一步提升编译效率。

参考文档

- Apple - WWDC 2013 Advances in Objective-C
- Apple - WWDC 2018 Behind the Scenes of the Xcode Build Process

- Apple – WWDC 2019 Binary Frameworks in Swift
- Apple – WWDC 2020 Distribute binary frameworks as Swift packages
- Swift org – Evolving Swift On Apple Platforms After ABI Stability
- Swift org – ABI Stability and More
- StackOverflow – #import using angle brackets < > and quote marks " "
- StackOverflow – Xcode: Copy Headers: Public vs. Private vs. Project?
- StackOverflow – Understanding Xcode's Copy Headers phase
- Xcode Help – What are build phases?
- Xcode Build Settings
- Big Nerd Ranch – Manual Swift: Understanding the Swift/Objective-C Build Pipeline
- Big Nerd Ranch – Build Log Groveling for Fun and Profit: Manual Swift Continued
- Big Nerd Ranch – Build Log Groveling for Fun and Profit, Part 2: Even More Manual Swift
- Quality Coding – 4 Ways Precompiled Headers Cripple Your Code
- try! Swift Tokyo 2018 – Exploring Clang Modules
- milen.me – Swift, Module Maps & VFS Overlays

作者简介

思琦，笔名 SketchK，美团点评 iOS 工程师，目前负责移动端 CI/CD 方面的工作及平台内 Swift 技术相关的事宜。

旭陶，美团 iOS 工程师，目前负责 iOS 端开发提效相关事宜。

霜叶，2015 年加入美团，先后从事过 Hybrid 容器、iOS 基础组件、iOS 开发工具链和客户端持续集成门户系统等工作。

美团民宿跨端复用框架设计与实践

作者：凯林　森伟　熙辰　戈弋　少元

从 PC 时代、移动时代到万物互联的 IoT 时代，伴随终端设备的日趋多样化，跨端复用的种子自此落地，开始生根发芽。从依靠容器能力、各类离线化预装包的 Hybrid 方案，到通过 JSC 连接 JavaScript 生态与原生控件，结合视图框架（React、Vue 等）寻找效率、动态性和性能更均衡的 Native 容器方案（React Native、Weex 等），接着由微信牵头的以多进程 WebView、容器标准化的小程序方案出世，各平台小程序随之春笋萌发，随后带来了国内 Taro、uni-app、Rax、Remax 等多端框架的百家争鸣。

从业务角度出发，跨端技术演进更多是在不同阶段、不同时间段内业务效率上的选择，美团民宿业务就是在大前端融合的浪潮中逐浪前行，不断探索和迭代抉择，为解决业务痛点而孵化出跨端框架技术。本文主要分享美团民宿在跨端复用技术探索层面以及业务实践过程中积累的经验，希望能给大家带来一些帮助或者启发。

1. 背景

1.1 美团民宿业务介绍

美团民宿专注为消费者提供“住得不一样”的旅居体验，提供的服务包括民宿、酒店、公寓、客栈、短租、宾馆、旅行住宿等，同时包括树屋、房车、INS 风等新奇的网红民宿。美团民宿自上线之后，业务发展迅猛，在供给侧，房源类型不断丰富，各类分销、直销、直连、境外陆续推出，房源信息维度不断扩展，筛选、推荐、信息呈现也不断变得复杂。同时伴随着营销方式的丰富、房东管理、经营、服务的不断扩充，民宿的业务也越来越复杂。美团民宿大前端伴随业务的发展不断自我迭代，移动端整体架构也随之不断调整、升级，以寻求匹配业务多样化、复杂化的发展诉求。

1.2 美团民宿移动端现状

业务的发展和跨端复用技术的不断演化，让美团民宿客户端从业务刚起步的单端 Native App，到跨 App（民宿 App、美团 App、点评 App ）的 Native 复用和以 SSR 弥补性能差距的 Hybrid 的结合方案，在这场性能和效率的博弈中，客户端最终落脚以 React Native（以下简称 RN）为核心的复用框架。在此同时，民宿小程序端也随着微信小程序的诞生、生态壮大、多平台化的趋势不断成长，逐渐形成多平台复用的小程序架构。

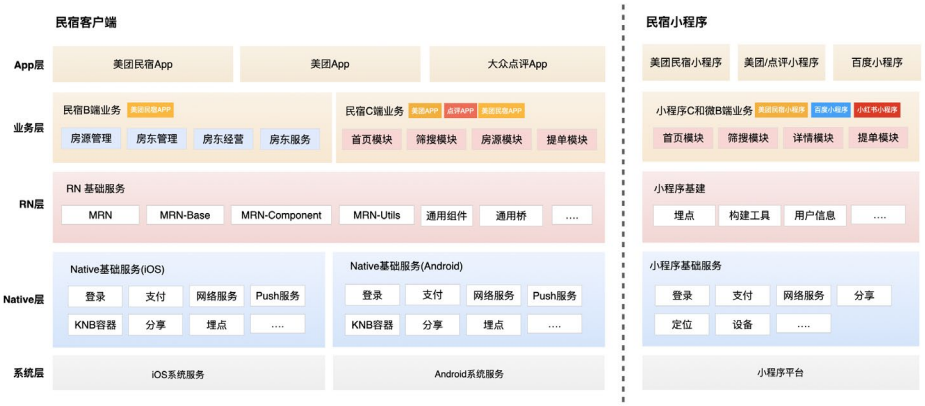


图 1　美团民宿移动端原始架构图

上图是美团民宿移动端原始架构图，左侧是客户端的技术架构，iOS 和 Android 系统层之上是独立的 Native 基建层，再往上通过了 RN 打开双端的复用之门，接着以 RN 容器标准化屏蔽了宿主应用间差异，保障了容器化的一致性，进而实现了业务层的复用和跨 App 的复用。右侧是民宿小程序当前简化的架构图，我们在基建层做了多端适配，通过多平台复用构建工具实现了各平台小程序的复用。当前客户端和小程端相关独立，开发维护也相互独立，团队各司其职。

尽管美团民宿 App 已经通过 RN 实现 iOS 和 Android 的跨端复用，但是由于 App 和小程序仍然需要投入双倍的人力成本进行业务迭代，所以我们思考一个问题：**是否可以更进一步，使用一套代码解决多端，把 iOS App、Android App、小程序进行大一统**。

2. 美团民宿跨端复用框架设计

2.1 行业现状

近几年，在微信小程序产品牵头下，业界也随之诞生出各种小程序应用，各端技术差异使得开发和维护成本都成倍增加。为了抹平原生开发、小程序开发、Web 开发等技术差异，一些优秀多端框架也就此诞生了。比如 Taro、uni-app、Rax、Remax 等，这些框架都是以自身定义 DSL（一般是 React DSL、Vue DSL）转换成各端应用（微信小程序、RN、H5 等），从而实现一套代码，多端运行。

在美团民宿业务中，App 的交易占比较大，从业务角度出发需优先保障 App 的性能体验和需求开发效率，而当前的民宿 App 已迁移至 RN 技术栈。基于这两点，我们希望跨端复用方案的是：RN 转到小程序平台方案，所以上述的多端框架并不能满足我们的 RN- 小程序跨端复用的诉求，为此美团民宿参考了业界多端设计方案，实现了基于 RN 转小程序复用的方案。

RN 采用的是 React 语法，因此如何将 RN 转换为小程序，首先要思考如何将 React 代码转换成小程序可运行的代码（简称小程序代码），其次是 RN 基础组件库的适配。随着这几年的发展，React 代码转换成小程序代码在业界实践也是层出不穷，业界方案分为编译时与运行时两类，以下是这两类方案的简单对比：

框架分类	重编译	重运行
典型代表	Taro2.0 / Rax 编译时	Taro Next / Remax
原理	编译时将 React 代码直接转换成小程序代码	运行是通过 React 自定义渲染器完成页面绘制
优势	性能损耗低	无语法限制
劣势	语法限制大	性能损耗大

对比来看，重编译方案有一个严重的问题：语法限制。因为大部分前端开发者们已经对灵活的语法有一定的依赖性，比如会使用高阶组件、在条件判断的时候写很多 return 等等，这种写法很难在编译过程被准确命中。因此，编译时方案就会制定一些语法规则来限制开发者的写法。重运行方案则没有语法限制问题，可以随意使用各种 React 特性。它的实现原理是通过 react-reconciler 实现小程序平台对应的 React

渲染器（以下简称 MP-Renderer），从而来渲染虚拟 DOM 树。不过小程序没有 DOM API 可以更新界面，所以生成的虚拟 DOM 树数据是通过小程序的 setData 触发渲染层的更新，在渲染层里有一个通用模板可以用来渲染这些数据。

因重编译语法限制的问题，我们决定采用重运行时方案来实现 RN 转小程序。但重运行方案存在性能问题，难以满足业务的要求，我们经不断探索后设计了对应的方案极大提升了性能，下文会详细描述如何解决这个问题的。

2.2 整体方案设计

2.2.1 RN 与小程序复用的技术方案

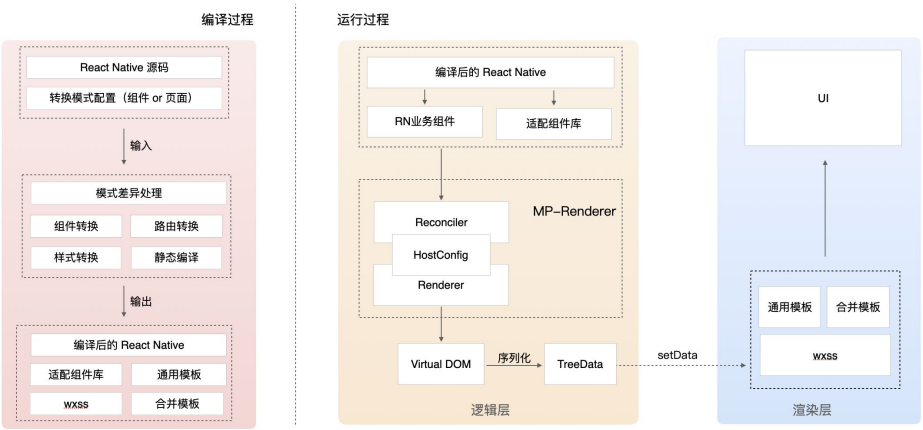


图 2　RN 与小程序复用技术方案图

整体架构分为两个部分：编译过程、运行过程。它的渲染方式与上文描述重运行时方案类似，都是通过 MP-Renderer 来处理 React 代码。下面我们来简要分析这两个过程：

(1) **编译过程**：该阶段对 RN 源码进行一定的转换处理，用于运行过程，编译后主要产生有以下产物：

- **编译后的 RN**：经过编译后产生 RN 代码，本质上还是 React 代码。

- **适配组件库**：RN 基础组件的适配库，是使用小程序自定义组件实现的。
- **通用模板**：由于小程序没有像 Web 有 DOM API 操作节点操作方法，所以这里通过一个通用模板来渲染 React 渲染出来的 TreeData（页面虚拟 DOM 树序列化后的 UI 数据）。
- **合并模板**：主要用于性能优化的，下文会详细分析这个模板的作用。
- **WXSS**：将 RN 代码的 Style 转换为 WXSS，这样可以减少页面的 TreeData 数据量，从而优化性能。

(2) **运行过程**：运行过程分为逻辑层和视图层两部分。

- **逻辑层**：编译后的 RN 源码包含 RN 业务组件和适配组件库，适配组件库是通过小程序自定义组件来进行适配。这样的方式既可以灵活使用小程序原生代码对齐 RN 组件功能，也可以提升转换后小程序的性能，因为小程序原生代码不会产生 TreeData 数据，从而使性能上得到提升。逻辑层有一个 MP-Renderer ，实现方式和上文讲述的是一样的，RN 代码经过渲染后，便产生对应的虚拟 DOM 树，虚拟 DOM 树数据再经过序列化便产生对应的 TreeData（描述页面的 UI 数据）。
- **渲染层**：当页面需要更新的时候，逻辑层通过 setData 将 TreeData 传输到渲染层里，TreeData 与通用模板、合并模板和对应样式结合在一起，便可以渲染出对应的 UI。

综上所述，上述整体设计与业界多端框架有点类似，但是也有不同点，主要体现在适配组件库和合并模板。适配组件库上文有解释比较好理解，而合并模板这里可能大家还是比较有疑惑的。其实这个合并模板内容是由编译过程的“静态编译”转换生成的，这样的处理方式是为提升转换后的小程序性能，接下来，我们会着重来讲述这个性能解决方案。

2.2.2 性能解决方案

重运行时方案性能损耗原因是什么？正如上文所说，重运行时方案会将所有 React 代码对应的 TreeData，再通过小程序 setData 传输到渲染层，当页面初始化或者大

数据更新的话，setData 就需要传递比较大的一个数据，因此也就会造成对应的性能问题。所以要解决这种方案的性能问题，核心就是要减少 TreeData 数据量。

在上述 RN 转小程序方案，有提到适配组件库、样式转换等是可以起到对应性能优化作用的，它的优化原理正是通过减少 TreeData 数据的方式。尽管这些方式可以优化性能，但是在页面比较复杂的时候，TreeData 数据量仍然会保留比较大，因此优化效果并不明显。为此，我们思考一种新的方式来进一步压缩 TreeData 的数据量，也就是前文所提到的结合静态合并树节点方案，在讲述该方案前我们先来看下一个 RN 代码转换为 TreeData 的例子：



图 3　RN 代码转换 TreeData 示例图

如上图所示，RN 代码转换后的 TreeData 是一个描述 UI 树的 JSON 数据，等同于右侧的 UI 树，将这颗树的节点进行分类，可以分为静态数据和动态数据，比如 View、Text 节点就是静态数据，而“Hello”、“World”则是动态数据。所谓静态数据，就是编译过程可预知的，因此这些数据是不是可以转换另一种形式来描述 UI 呢，从而减少 TreeData 的数据量。答案是肯定的，静态编译合并树节点正是通过这样的原理来实现的，如下流程所示：

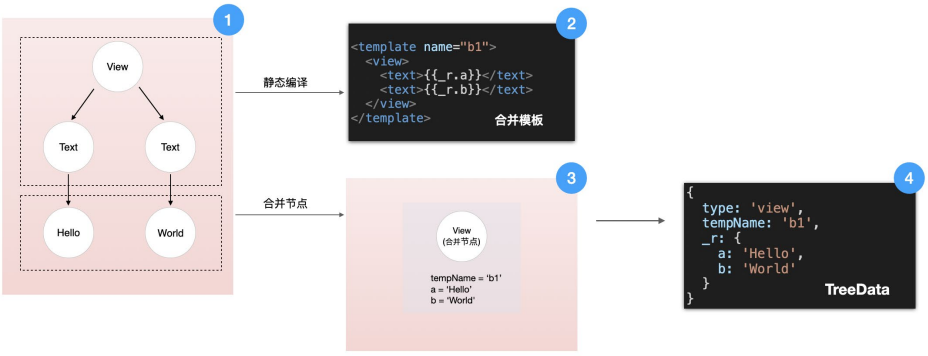


图 4　静态编译合并树节点原理图 -1

这个方案有两个动作，分别是静态编译和合并树节点，静态编译就将 RN 代码的转换成合并模板，如上图序号 2 代码所示，合并模板的名称为“b1”，内容就是一段与 RN JSX 代码对应的 WXML 结构片段。而合并节点是将已经静态编译的节点进行合并，如上图序号 2 至序号 3 流程所示，原本五个节点被合并到顶层的 View 节点，这个 View 节点称为合并节点，合并节点需要记录合并模板的名称和相关的动态数据，目的是为了渲染时让合并节点可以找到对应的合并模板进行渲染，经过这样合并节点后，最终生成的 TreeData，如上图序号 4 所示。可以看到 TreeData 相比之前的数据量就减少了 60% 左右！

看到这里，是不是有同学就有疑问了，上文不是提到静态编译会有语法限制，那这里是否会有语法限制？确实，如果是完全静态编译，是会有语法限制，而这里所说的结合静态编译是有选择性的编译，即在编译过程，首先会通过 AST 分析节点是否静态数据，如果是的话，再转换成对应的合并模板。如果遇到不可预测的动态节点，则按照运行时方案去处理。因此，最终生成的 UI 树节点即会包含合并节点、也会包含原本的组件节点，如下图所示：

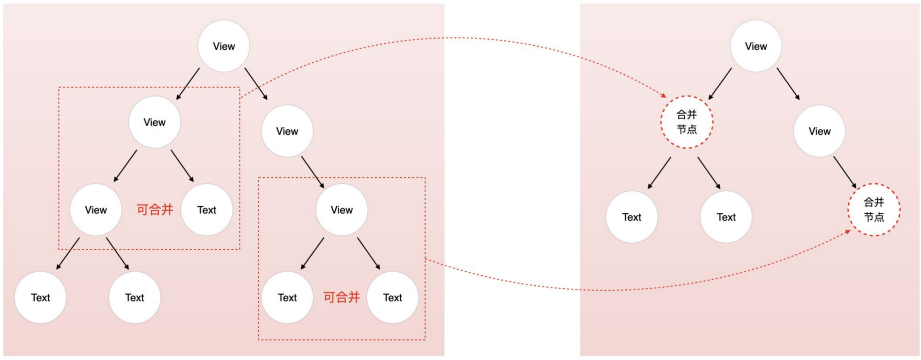


图 5　静态编译合并树节点原理图 -2

通过这样的方式，既可以保证语法无限制，又能通过编译结合的手段最大化优化性能。当然了这种方案也是有缺点，因为这种方案其实是用空间换性能的方式，生成的合并模板会影响会影响包大小，不过对于一些需要追求性能的页面，这点包大小的增加是值得付出的。

为了更好地衡量解决方案对性能的提升程度，我们参考 Taro 官网的实验（实验内容），对优化前后以及原生和 Taro 3.0 运行后的性能指标进行采集与比较。经过实验，统计出各框架在初始化、加载数据、加载大量数据的操作耗时，如下表所示：

操作耗时 \ 框架	优化前	优化后	原生	Taro 3.0.17
初始化（首屏渲染时间）	897ms	423ms	210ms	675ms
加载普通数据（20 条）	1124ms	198ms	110ms	640ms
加载大量数据（400 条）	5330ms	1041ms	470ms	3919ms

从上表中可以看出：性能优化后，得益于更少的渲染数据与更精简的节点树，加载数据的操作耗时比优化前减少 80% ，初始化耗时减少了 52%。与同类型的框架 Taro 3.0 相比，也有更好的性能表现。

与原生相比，优化后性能差距明显减少，但是由于运行时方案相对于原生需要更多的 setData 数据开销和更复杂渲染流程，所以从原理上运行时方案和原生性能差距客观存在。尽管如此，业务实践上两者差距并不会那么明显，因为在测评实验中测试数据

比较纯粹，setData 数据使用率较高，但在业务实践中原生开发 setData 数据难免冗余且难以优化，而运行时方案会默认优化冗余数据使得两者性能差距更接近，从我们历史业务实践数据上看，性能与原生差距在 10% 左右。

3. 美团民宿跨端复用实践

在跨端复用探索中，我们用创新的方案解决了性能和特性限制的难题，设计了 RN−小程序跨端复用框架。虽然跨端复用属于“利器在手”，但是这是一把“双刃剑”，用得其所则事半功倍，处理不当则隐患丛生。那么，如何在业务实践中驾驭好这把利刃呢？我们先介绍在业务实践中遇到的问题，然后介绍解决这些问题的方案。

3.1 跨端复用场景下的问题

- **复用场景下的问题**：小程序产品形态以轻、快、便为旨，用户可快速使用，用完即走，客户端产品相对全、精、稳，可以满足更多的用户需求，以用户留存、用户认知、用户体验为主，两者在产品功能上存在较大的差异，如何恰当地处理产品差异化问题是跨端复用的场景下的一个重要挑战。
- **跨端复用质量隐患**：实现了复用便要考虑两端的各种兼容性问题，这就会产生各种质量上的隐患。如何在复用组件不断迭代中，保障组件接口、输入、输出的兼容性问题？如何保障各个复用组件底层依赖的统一、适配层接口的统一？双端复用场景下，如何更好的做测试和监控？双端同学存在各自技术认知的边界，如何在出现问题时快速排查、及时止损？
- **跨端复用流程规范问题**：新的技术革命，必然打破旧的秩序，在当前跨端复用场景下，各种包括工程管理、代码规范、分支管理、需求同步的问题也会孕育而生，同需解决。

3.2 跨端复用应用架构

为了解决跨端复用在业务实践中遇到的各种问题，我们重新设计了跨端复用应用架构，从架构分层管理、复用方式设计、流程规范、质量保障方面入手，重点解决跨端差异化、质量隐患、流程规范各种问题，并寻求复用的最大化和性能上的均衡。

3.2.1 跨端复用应用架构演进

在这里，先贴出动态的架构演进过程，让大家有一个宏观的认识。我们先简单地描述下演进过程，后续会基于最终的架构图再做详细的介绍。大致演进过程如下：

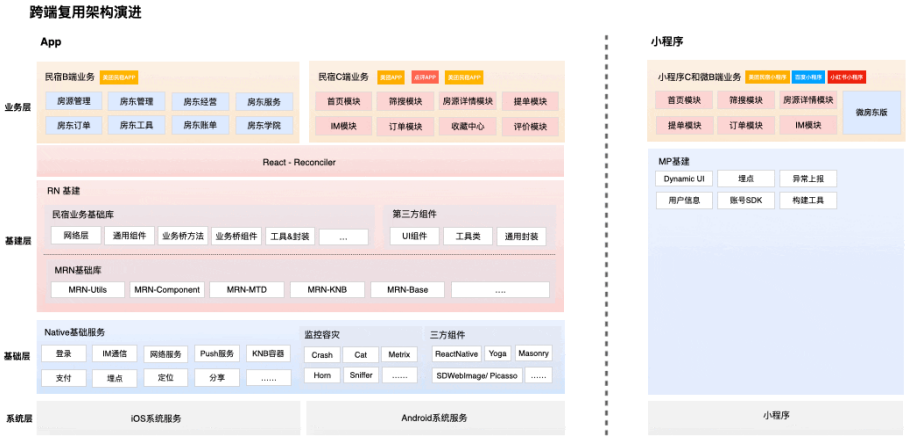


图 6　跨端复用架构演进动画图

- 起初，客户端分 Android App 和 iOS App 单独开发，引入 RN 技术实现了 Android 和 iOS 跨端复用，但是小程序端依然需要单独维护迭代。
- 为了跟进一步实现 RN- 小程序跨端复用，我们接入了自研的 RN- 小程序跨端复用框架，并基于框架的适配规范，以 RN 的基建为基准，打造出一个和 RN 基建统一接口的小程序适配层。
- 完成小程序渲染器接入（MP-Render）和小程序适配层后，React-Reconciler 这一层就可以打通到小程序侧，实现了 React 代码复用到小程序的能力。
- 实现 RN 与小程序间的复用后，就可以对存量的 RN 代码进行抽象、适配、整理，进而抽取出一个组件复用层，这个复用层可直接供上层业务层直接使用。
- 最后，为了解决跨端复用场景下各种流程、协作和质量隐患，我们配套了相应的流程规范和质量保障措施。

3.2.2 跨端复用应用架构整体介绍

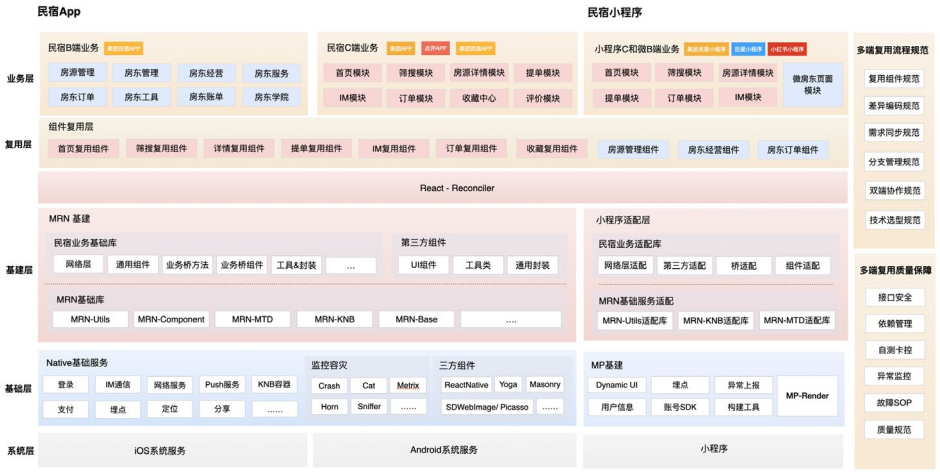


图 7　跨端复用应用架构图

整个民宿的 RN- 小程序跨端复用架构图如上，我们按照从下到上，从左到右的视角进行解读：

- **系统层**：最底层是系统服务，除了 iOS 系统和 Android 系统外，我们把小程序视为一个单独的系统模块。
- **基础服务层**：系统服务之上是基础服务层，这一层主要是集团基于 Native 和小程序建设的基建，全公司通用，覆盖了研发工程中方方面面的基础服务。在此基础上，我们在小程序基建中引入了基于 react-reconciler 实现的小程序运行时渲染器（MP-Render），这个渲染器能在运行时动态更新 vnode 以匹配编译转化的小程序 UI 模板，调用小程序原生 API，最终渲染出小程序组件，有了这个基于 React 的小程序渲染器便使得跨端复用成为可能。
- **基建层**：基础服务层之上是基建层，这块主要包括 MRN 基建和小程序适配层，我们以 MRN 的基建为标准，适配出一个统一标准和统一接口的小程序适配库，通过这一层适配，上层可以无感知、无差异地以同一标准实现复用组件。其中适配层分为 2 块，下半部分主要适配 RN 基础服务，上层是民宿业务独立封装的基础库和第三方库，这块我们单独引入一个名为 Mapping 的适

配库。一个独立的适配库可以让 RN 和小程序在业务迭代和技术变革过程中相互独立，互不干扰，如此就能保障技术的推进完全不会影响业务的迭代。基建层的最上方是 react-reconciler，React 框架本身就是把协调过程和渲染过程分开的，react-reconciler 是实现跨端复用的核心，所以我们把它单独展示出来，它真正打通了客户端和小程序的隔阂，只要有了一个独立的小程序渲染器，就可以全面、无限制的把 React 代码复用到小程序。

- **复用层**：基建层再往上是复用层，复用层主要以组件维度做复用，复用组件是基于存量 RN 组件做抽象和适配，然后抽取独立出来，复用层的组件以统一的标准和接口供上层业务使用。复用层是很重要的一块，好的复用机制能帮助我们解决前面提到的产品差异化问题和复用最大化问题。这块我们单独放到 `3.3 跨端复用方式设计` 来详细讲解。
- **业务层**：复用层之上就是业务层，业务层的各模块主要以页面容器来承接复用组件，基于不同的端和产品差异，可以灵活、动态配置页面的组件来满意业务的差异化需求。

3.3 跨端复用方式设计

差异化问题，一直是跨端复用场景中的一个痛点，双端的产品上、平台上、代码上的差异如何妥善的处理、适配，也是我们一直思考的问题。而好的差异化处理方案可以提升代码的可维护性、降低质量隐患、提升开发效率。我们从复用设计层面出发，探索出页面复用模式、组件复用模式、“组件 + 逻辑复用”模式等三种复用设计方式，并且根据不同的场景下采用不同的复用模式，可以较好地处理跨端差异化问题，同时能兼顾效率提升、性能体验和可维护性。

3.3.1 差异化下的复用方式

我们自研的复用框架提供两种复用模式，如下图所示：

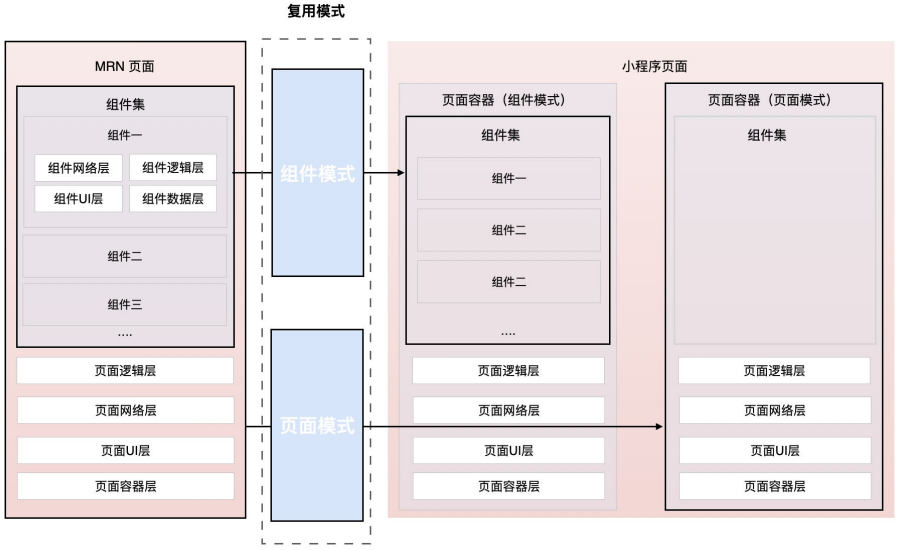


图 8　小程序复用方式原理图

页面复用模式：页面模式基于页面维度的，可以直接把页面的网络层、逻辑层、数据层以及页面内的组件集全部转换复用，这样可以达到复用的最大化，代码复用率能达到 90% 以上，人效提升明显。**组件复用模式**：组件模式是基于组件维度的，复用以页面中的业务组件为目标，把页面的所有组件抽象、解耦、规范化之后抽取为复用组件。组件模式只能复用组件内代码，对于页面容器的逻辑交互、网络层都需要小程序自己实现，代码复用率相对较低，但是组件复用更灵活、可控，可随意插拔、拼接、定制。

以下是两种复用模式的优劣分析。

页面复用模式

优势

1) 提效明显：整个页面包括所有组件、页面逻辑层网络层一并打包转换复用，代码复用率极高，开发效率提升幅度更大。 2) 接入成本低：整个页面直接转化同步复用，无需小程序同学协助接入，减少双端协助、接口沟通带来的出错风险。

劣势

1) 灵活性低：业务差异和小程序特性不易处理，双端差异适配只能在 RN 上做，代码易出错，维护成本高。 2) 性能劣势：整体页面由 RN 转换复用而来，页面一次性渲染，性能上会略差一些，而且做页面级的性能优化困难。 3) 包大小风险大：整页复用情况下包大小较大，且不能动态调配（比如页面内某一模块需求迭代较少，不想复用，但是页面模式做不到动态移除）。

组件复用模式

优势

1) 轻便灵活：组件如插件般可随意插拔、拼接、定制，可较好解决 App 和小程序双端的差异性问题，针对差异点双端可以独立实现，提高项目的可维护性。 2) 性能较好：页面容器依然是小程序原生组件，如滚动、滑动组件采用原生可减少性能损耗，另外组件分布式 setData 渲染有更好的性能，不会像整页一次性渲染导致 setData 数据量较大影响首屏加载性能。 3) 性能优化空间大：不会影响做页面维度的性能优化（如首屏优先、请求前置）。 4) 包大小可控：组件是否复用可以动态调配，比如把页面中迭代较少的组件不复用以减少包大小。

劣势

1) 提效有限：组件模式只能复用组件内的代码，代码复用率较低，页面容器、逻辑层、网络层小程序依然要自己维护一份代码。2) 复用组件维护成本高：组件的接口要考虑组件升级迭代的兼容性、可维护性问题，管理不当，容易产生质量隐患。 3) 接入成本较高：小程序需要实现 RN 的页面逻辑，然后按照组件接口进行接入，有更高的接入成本。

两组复用模式各有利弊，页面模式复用率高，但是灵活性低、性能欠佳；组件模式轻便灵活，性能可控，能较好的处理平台差异化问，但是复用率低、维护成本高。我们在想有没有一种方案能保留组件模式的灵活性，又能降低组件维护成本、提高复用程度。在业务实践中，我们探索出一套“组件 + 逻辑复用”的模式，可以较好地解决上

面提到的问题。

3.3.2 差异化下的逻辑复用

“组件 + 逻辑复用”模式依然保留组件复用的方式，但是在组件复用基础上增加了逻辑层（包括页面逻辑、网络、数据层）的复用，这样保留了组件灵活性，也增加了复用性。具体设计如下图：

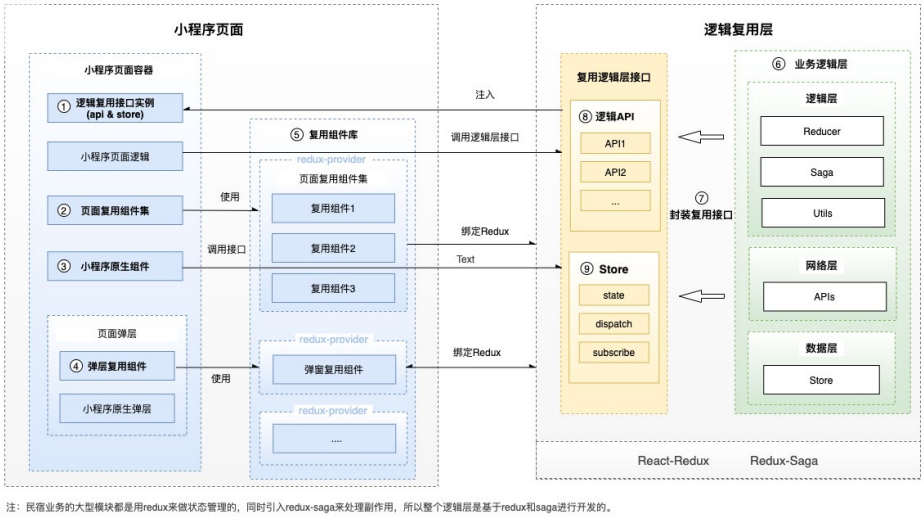


图 9　组件 + 逻辑复用模式原理图

整个组件 + 逻辑复用模式设计图如上，我们按照图片标注的序号进行一一解读：

1) **逻辑复用接口实例**：在小程序的页面容器中，通过注入的方式获取逻辑层复用的接口实例，通过这个实例便可以调用接口实现获取、更改、监听 Redux 的状态，实质上就达到了逻辑复用的效果。 2) **页面复用组件集**：页面可以自由使用复用组件，复用组件可大可小，可以虽然拼装布局，保留了组件模式良好的灵活性。 3) **小程序原生组件**：页面既可以使用复用组件，也可以用小程序原生组件来实现小程序差异化的功能和特性，这样能较好的处理双端差异性。小程序原生组件可以通过 逻辑复用接口实例 来调用逻辑层功能，进而达到逻辑复用的效果。 4) **弹窗复用组件**：弹窗复用组件和页面复用组件同理，这边主要说明可以按照各类维度把复用组件分类，进

而更好的做复用组件管理。 5) **复用组件库**：复用组件库的复用组件可多可少，可大可小，如果页面双端差异性小，一个大组件即可满足。每个复用组件集外层包一层 Reudx-Provider 并设置相同的 Store，便可以和逻辑层自动绑定上。因为 RN 组件本身就是基于 Redux 的，所以复用过程相对容易。 6) **业务逻辑层**：最右侧的业务逻辑层可以简单理解成 3 块，一块是基于 Redux Store 的数据层，这里存放整个页面模块所有的数据和操作、监听数据的接口，一块是包含页面内所有网络请求的网络层，另外就是用来流转状态和处理复用的 Reducer、Redux-Saga，以及配套的各种工具类。业务逻辑层可以根据双端的差异把 Reducer 与 Saga 分拆更小的单元实现差异化的逻辑复用，提升逻辑复用层的代码可维护性。 7) **封装复用接口**：业务逻辑层包含整个页面的业务逻辑，只要针对性开放接口给小程序，让小程序可能获取、更改、监听 Redux 的状态，那实质上就达到了逻辑复用的效果。开放接口给小程序有 2 种方式：逻辑 API 接口 和 Store。 8) **逻辑 API 接口**：基于 Store 给小程序提供小程序真正需要的逻辑 API 接口，通过这些 API 小程序可以来获取数据来渲染 UI（如：渲染没有复用的组件），也可以更新数据，也能监听复用组件内部的数据变化。 9) **Store**：把 Redux Store 暴露出去，小程序便利用 Store 实例可以通过 getState、dispatch、subscribe 来操作、监听状态机了，也就达到逻辑层复用的目的了。

这种方案的优势很明显，它保留组件模式的灵活特性，可以比较方便做差异化处理和性能优化。而逻辑复用层把 Redux 包含进来了，这样不仅转化容易、不易出错，而且逻辑复用接口基于 Redux 的 Store，接口较好设计，容易维护、不易出错。而对于逻辑层，可以根据业务上一些差异做 Reducer 与 Saga 分拆，把不需要复用的代码逻辑排查在外，逻辑层复用也可以做到像组件一样热插拔，按需引入，这样也比较好做差异化代码管理，挺高项目的可维护性，同时也能优先减少包大小风险。

3.4 跨端复用流程规范

为在代码跨端复用过程中尽可能提升开发效率并避免引入质量问题，我们制定了差异化编码规范、需求同步规范、复用组件规范等开发流程规范，以下将通过 RN 到小程

序产品需求同步过程进行简单的介绍。



图 10　跨端复用流程规范图

1. 评估业务需求是否需要同步

针对 PM 提出需要同步的需求，客户端尽量将 RN 业务代码复用至小程序，以提升开发效率。无需同步的需求将通过差异编码规范进行控制，避免同步至小程序后增加潜在风险与测试成本。通常可使用平台判断（如 iOS、Android、WX_Platform ）的方式控制业务代码是否打入复用组件包，也可通过 module.rn.js、module.wx.js 不同后缀文件方式完成相同接口不同逻辑的实现。

2. 评估是否有关联依赖需求

如明确业务需求需要同步，先判断该需求是否有前置需求依赖，再评估技术方案。如无依赖可直接开始复用适配工作；如有依赖，需判断前置需求能否一起同步或做适当降级，以此递推，避免因前置依赖需求未同步出现不符合预期的问题。

3. 制定 RN 组件适配与小程序接入方案

明确需求同步范围评估工作后，需完成以下技术评估工作：（1）明确需求是否需要新建复用组件还是在原有的复用组件上进行迭代。如需新建复用组件 NPM 包，需根据组件复用规范进行技术选型，确定使用“组件 + 逻辑复用模式”、“页面模式”还是“组件模式”，并制定相应的复用组件接口协议；（2）明确该需求是否需要开发 RN- 小程序映射方法、组件，并评估相应的开发量。完成技术评估后需提前与小程序侧沟通接入排期。

4. RN 组件适配开发

客户端完成 RN 侧需求开发后，便可进行复用组件适配小程序开发。完成适配开发工作后需在 RN 页面与小程序 Demo 页面中对复用组件同时进行测试，避免在适配小

程序过程中引入 RN 页面 Bug。复用组件测试完毕后将 NPM 包以及相应的接口文档提供给小程序接入，但在打包前需严格审查当前版本与上个版本间的 diff，避免不符合预期的代码也被同步至小程序。

5. 小程序接入 RN 适配组件

适配完成后将组件打包提供给小程序侧接入，接入后需在美团民宿小程序环境下再次进行自测。原则上客户端同学提供适配好的 RN 组件后，由小程序侧同学接入并测试，但我们也鼓励客户端在完成 RN 组件开发与复用适配后，一并完成小程序侧的组件接入工作，这样需求开发完整度更高，并能有效减少跨端开发下的沟通成本。后续随着大前端融合推进，RN- 小程序代码复用率将逐步提升，客户端（iOS、Android）与 小程序代码将倾向由一名同学完成多端开发。

6. RN 适配代码合入迭代分支

需求在小程序测试完毕后，将 RN 组件适配 Feature 分支代码合入 Release 迭代分支，并在客户端（iOS、Android）打包上线。

3.5 跨端复用质量保障

跨端复用场景下存在包括复用组件接口兼容性问题、组件间的依赖隐患问题、测试和监控的缺失问题，以及故障排查困难等各种质量隐患，我们在业务实践中，也探索出一系列解决这些隐患的质量保障措施，包括组件接口维护、组件依赖管理、双重自测卡控、异常监控融合、双端故障 SOP、跨端复用流程规范 。这些措施能有效保障复用场景下双端的线上质量，民宿业务在跨端复用推进中，因为这些措施的保障护航，没有出现任何的线上故障。

1. 组件接口维护

复用组件随着业务迭代会不断更新升级，组件升级过程中便会带来的组件接口、输入、输出的变动，进而产生兼容性隐患，比如组件输入参数类型变动，而小程序端或 RN 端没有及时兼容或者未知晓，非常容易引发线上质量问题。为此，我们制定了组件接口维护计划，包括复用组件接口规范、组件版本管理规范、组件接口文档建设

等。复用组件接口规范要求复用组件接口、参数必须严格按照规范来，如参数类型使用基础类型、只增少减原则、接口命名清晰、参数个数限制等等，减少双端的接入组件难度，避免参数频繁变动产生质量隐患。组件版本管理规范要求组件版本升级必须遵循語意化 2.0，并且有相应的版本升级文档。组件接口文档建设也是很重要的一环，每个复用组件都有相应的文档维护，记录参数的增删改查，接入方对组件接口变动一目了然，自然减少了接入风险。

2. 组件依赖管理

组件依赖主要存在两个问题，第一，复用框架本身也在不断升级优化、新的复用组件可能用新的编译版本转化而来并且依赖新的运行时渲染器，但是旧的复用组件可能会出现不兼容问题，因此我们开发相关的工具，如果组件依赖的运行时渲染器版本和小程序内置的不一致就会发出警告，提示组件兼容性问题。第二，因为不同的复用组件来自不同的 RN 模块，它们可能依赖不同版本的第三方库，容易产生版本不一致的质量问题。目前的解决方案是把这些依赖库分别打入各自的包里，这样复用组件间依赖相互独立，互不影响。再结合 Tree-Sharking 的优化，打入的依赖的真实包大小并不大，用小量的包大小换取更稳健的质量保证。

3. 双重自测卡控

在跨端复用场景下，一个复用模块的改动要考虑双端兼容和新旧版兼容问题，相比与之前有更高的出错风险，更全面的自测能帮我们尽早暴露问题，减少故障风险。所以我们在 App 侧和小程序侧做了代码自测覆盖率卡控，要求改动代码执行覆盖率超过 90% 才能提测和上线。复用组件既在 RN 侧自测过一遍，在小程序接入后又强制要求再自测一遍，双重自测卡控更能保障组件质量和线上质量。

4. 异常监控融合

RN 和小程序侧都有单独的异常监控机制，包括 JS 异常监控、API 异常监控、自定义异常监控等。但是双端的异常监控机制差别较大，在复用场景下两者交叉混用导致异常监控体系混乱，上报数据格式、策略、日志不统一而造成监控体系误告、漏

告、排查困难、运维混乱等问题。所以我们把双端的异常监控模块打通，适配了底层异常上报逻辑，统一了双端的上报规范，告警策略、日志、处理流程。异常监控体系双端融合后，异常上报、监控、运维都顺畅许多，也帮我们发现不少的线上异常，是RN- 小程序跨端复用场景线上质量的坚固屏障。

5. 双端故障 SOP

鉴于双端同学存在技术上的隔阂和信息不对称，当出现复用组件的故障或异常时，如何快速排查问题成为一个痛点，小程序的错误日志 RN 同学不熟悉，小程序同学不熟悉 RN 的业务代码实现，框架层面的错误更难排查。为此，我们整体了梳理双端故障SOP，这里面包括常见日志分析帮助鉴别是复用组件、小程序端、底层复用框架的问题和相应的解决方案，同时开发了 Source Map 错误反解工具协助 RN 同学反解小程序日志帮助快速排错等等。这些 SOP 和工具能够在第一时间帮助双端同学自主或协助排查相关故障，快速止损。

6. 跨端复用流程规范

流程规范包括前面提到的复用组件规范、编码规范、需求同步规范、分支管理规范等等也是质量保障的重要的一环，它让研发流水线每一环都有严格的法律约束，保障整条研发流水线最终能把完整的产品交付到用户手里。

3.6 成果

RN- 小程序跨端复用的设计方案在业务实践中不断完善，探索出效率相对最大化的复用模式。从开发效率角度来看，提升显著。我们总结了代码复用率与人效提升率来评估效率的提升，两个指标具体计算公式如下：

- **代码复用率**：∑ (RN 复用模块代码行数 − 模块中 RN 与小程序平台分支判断代码行数) / ∑ (RN 代码总行数 + 小程序原生代码行数)；其中，RN 复用模块代码行数是根据框架转换生成的组件来确定。
- **人效提升率**：∑ (RN 开发耗时 + 小程序开发耗时 − (RN 开发耗时 + 转换适配耗时)) / ∑ (RN 开发耗时 + 小程序开发耗时)；复用前需要 RN 与小程序侧两

端的开发耗时，复用后只需要 RN 开发与复用组件转换适配的耗时，根据复用前后的耗时可以得出人效提升率公式。

根据转换采用的模式不同，可以得出代码复用率与人效提升率，如下表所示：

转换模式	代码复用率	人效提升率
组件 + 逻辑复用模式	76%	42.36%
页面复用模式	91%	46.71%

从表中可以看出，页面转换模式复用了页面与组件的代码，代码复用率可以达到 90% 以上；组件复用模式复用了组件与部分业务逻辑代码，复用率也可以达到 76%。在人效提升方面，所有模式都能达到较高的人效提升率，代码复用率越高人效提升率也越高，页面转换模式可以复用页面与数据状态处理逻辑人效提升比组件转换模式更高。

4. 总结

民宿大前端团队为解双端研发效率之痛，倾力而寻跨端之技，浅尝百草、深谙其理而后自建之，举偏补弊、终解跨端框架性能之桎梏，青出蓝而胜于蓝。而后践于实业，瑕弊昭然若揭。为此，重设框架以谋其变（复用架构设计），寻之新式以尽其效（复用模式设计），立之新法以固其序（跨端复用流程规范），磨之利器以护其城（跨端复用质量保障），至此成果初成。然朝夕变化不休，路漫远兮，吾当持之求索以适其变、顺其道。跨端复用前行之鉴，故记以文，望有启示，文毕。

作者简介

凯林、森伟、熙辰、戈弋、少元等，均为美团民宿前端团队研发工程师。

招聘信息

美团民宿长期招聘 Android、iOS、FE 前端工程师，坐标在福建厦门。感兴趣的同学可将简历发送至：tech@meituan.com（邮件主题请注明：美团民宿大前端）。

美团跨端一体化富文本管理技术实践

作者：高瞻　宇立等

一、引言

在互联网圈，开发和产品经理之间相爱相杀的故事，相信大家都有所耳闻。归根结底，往往都是从简单的改需求开始，然后你来我往、互不相让，接着吵架斗嘴，最后导致矛盾不断升级，甚至带来比较严重的后果。

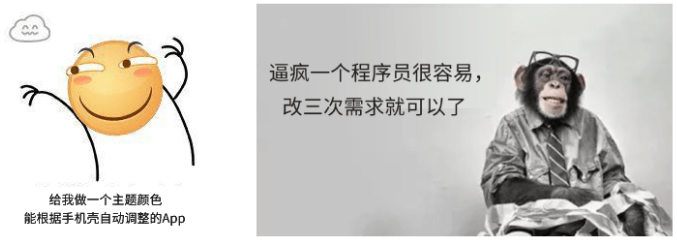


图 1

在这种背景下，如果把一些功能相对简单的、需求变动比较频繁的页面，直接交给产品或者运营自己去通过平台实现，是不是就可以从一定程度上减少产品和开发人员之间的矛盾呢?

二、背景

当然上述的情况，美团也不例外。近些年，美团到家事业群（包括美团外卖、美团配送、闪购、医药、团好货等）的各个业务稳步发展，业务前端对接的运营团队有近几十个，每个运营团队又有不同的运营规则，这些规则还存在一些细微的样式差别，同时规则内容还会随着运营季节、节日、地理位置等进行变化和更新。这些需求具体来说有以下几个特点：

- **需求量大**：业务稳步发展，业务需求不断叠加，甚至部分业务呈指数级增长，

且业务方向涉及到一些业务规则、消息通知、协议文档、规则介绍等需求。

- **变更频繁**：面对市场监管和法务的要求，以及新业务调整等因素的影响，会涉及到需求的频繁变更，像一些业务 FAQ、产品介绍、协议文档、业务规则、系统更新日志等页面，需要做到快速响应和及时上线。
- **复杂度低**：这些页面没有复杂的交互逻辑，如果能把这些简单的页面交给运营 / 产品去实现，开发人员就能有更多的时间去进行复杂功能的研发。
- **时效性高**：临时性业务需求较多，且生命周期较短，具有定期下线和周期性上线等特点。

基于以上特点，为了提高研发效率，美团医药技术部开始构建了一个**跨端一体化富文本管理平台**，希望提供解决这一大类问题的产研方案。不过，部门最初的目标是开发一套提效工具，解决大量诸如帮助文档、协议页、消息通知、规则说明等静态页面的生产与发布问题，让产品和运营同学能够以**所见即所得**的方式**自主**完成静态页面制作与发布，进而缩短沟通成本和研发成本。

但是，随着越来越多业务部门开始咨询并使用这个平台，我们后续不断完善并扩充了很多的功能。经过多次版本的设计和迭代开发后，将该平台命名为 **Page- 佩奇**，并且注册成为美团内部的公共服务，开始为美团内部更多同学提供更好的使用体验。

本文将系统地介绍 Page- 佩奇平台的定位、设计思路、实现原理及取得成效。我们也希望这些实战经验与总结，能给更多同学带来一些启发和思考。

三、跨端一体化富文本管理解决方案

3.1 平台定位

我们希望将 Page- 佩奇打造成一款为产品、运营、开发等用户提供快速一站式发布网页的产研工作台，这是对该平台的一个定位。

- 对产品运营而言，他们能够**可视化**地去创建或修改一些活动说明、协议类、消息类的文章，无需开发排期，省去向开发二次传递消息等繁琐的流程，也无需

等待漫长的发布时间，从而达到灵活快速地进行可视化页面的发布与管理。

- 对开发同学而言，他们能够**在线编写代码**，并实现**秒级**的发布上线，并且支持 ES 6、JavaScript 、Less、CSS 语法，我们还提供了基础的工具、图表库等，能够生成丰富多样的页面。帮助开发同学快速实现数据图表展示，设计特定样式，完成各种交互逻辑等需求。
- 对项目管理方而言，他们能够清晰地看到整个需求流转状态和开发日志信息，为运营管理提供强大的“抓手”。

一般来讲，传统开发流程是这样的：首先产品提出需求，然后召集研发评审，最后研发同学开发并且部署上线；当需求上线之后，如果有问题需要反馈，产品再找研发同学进行沟通并修复，这种开发流程也是目前互联网公司比较常见的开发流程。

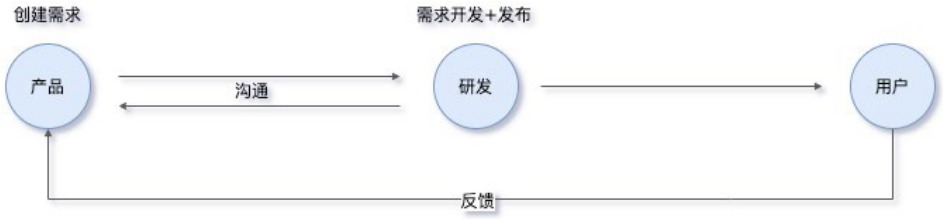


图 2　传统开发流程图

而美团 Page- 佩奇平台的开发流程是：首先产品同学提出需求，然后自己在 Page 平台进行编辑和发布上线，当需求上线之后有问题需要反馈，直接就能触达到产品同学，他们通常可自行进行修复。如果需求需要定制化，或者需要做一些复杂的逻辑处理，那么再让研发人员配合在平台上进行开发并发布上线。

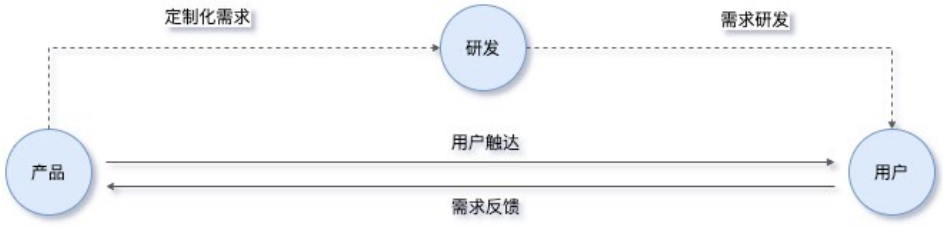


图 3　Page- 佩奇平台开发流程图

简单来说，对那些功能相对简单、需求变动比较频繁的页面，如果用传统的开发流程将会增加产研沟通和研发排期成本，因此传统方案主要适用于功能复杂型的需求。而 Page- 佩奇平台开发流程，并不适合功能复杂型的需求，特别适用于功能相对简单、需求变动比较频繁的页面需求。

综上所述，可以看出这两种开发流程其实起到了一个互补的作用，如果一起使用，既可以减少工作量，又可以达到降本提效的目的。

3.2 设计思路

我们最初设计 Page- 佩奇平台的初心其实很简单，为了给产品和运营提供一个通过富文本编辑器快速制作并发布网页的工具。但是，在使用的过程中，很多缺陷也就慢慢地开始暴露，大致有下面这些问题：1. 简单的富文本编辑器满足不了想要的页面效果，怎么办？ 2. 如果能导入想要的模板，是否会更友好？ 3. 怎么查看这个页面的访问数据？如何能监控这个页面的性能问题？ 4. 发布的页面是否有存在安全风险？

于是，我们针对这些问题进行了一些思考和调研：

- 当富文本编辑器满足不了想要实现的效果的时候，可以引入了 WebIDE 编辑器，可以让研发同学再二次编辑进行实现。
- 一个系统想要让用户用得高效便捷，那么就要完善它的周边生态。就需要配备完善的模板素材和物料供用户灵活选择。
- 如果用户想要了解页面的运行情况，那么页面运行的性能数据、访问的数据也是必不可少的。
- 如果发布的内容存在不当言论，就会造成不可控的法律风险，所以内容风险审核也是必不可少的。

实现一个功能很容易，但是想要实现一个相对完善的功能，就必须好好下功夫，多思考和多调研。于是，围绕着这些问题，我们不断挖掘和延伸出了一系列功能：

- **富文本编辑**：强大而简单的可视化编辑器，让一切操作变得简单、直观。产品同学可以通过编辑器自主创建、编辑网页，即使无程序开发经验也可以通过富

文本编辑器随意操作，实现自己想要的效果，最终可以实现一键快速发布上线。

- **WebIDE**：定制化需求，比如，与客户端和后端进行一些通信和请求需求，以及针对产品创建的 HTML 进行二次加工需求，均可以基于 WebIDE 通过 JavaScript 代码实现。具备专业开发经验的同学也可以选择通过前端框架 jQuery、Vue，Echarts 或者工具库 Lodash、Axios 实现在线编辑代码。
- **页面管理**：灵活方便地管理页面。大家可以对有权限的文档进行查看、编辑、授权、下线、版本对比、操作日志、回滚等操作，且提供便捷的文档搜索功能。
- **模板市场**：丰富多样的网页模板，简易而又具备个性。模板市场提供丰富的页面模板，大家可选择使用自己的模板快速创建网页，且发布的每个页面又可以作为自己的模板，再基于这个模板，可随时添加个性化的操作。
- **物料平台**：提供基础 Utils、Echart、Vue、jQuery 等物料，方便开发基于产品的页面进行代码的二次开发。
- **多平台跨端接入**：高效快捷地接入业务系统。通过通信 SDK，其他系统可以快速接入 Page- 佩奇平台。同时支持以 HTTP、Thrift 方式的开放 API 供大家选择，支持客户端、后端调用开放 API。
- **内容风险审核**：严谨高效的审核机制。接入美团内部的风险审核公共服务，针对发布的风险内容将快速审核，防止误操作造成不可控的法律风险。
- **数据大盘**：提供页面的数据监测，帮助大家时刻掌握流量动向。接入美团内部一站式数据分析平台，帮助大家安全、快速、高效地掌握页面的各种监测数据。
- **权限管理**：创建的每个页面都有相对独立的权限，只有经过授权的人才能查看和操作该页面。
- **业务监控**：提供页面级别 JavaScript 错误和资源加载成功率等数据，方便开发排查和解决线上问题。

功能流程图如下所示：

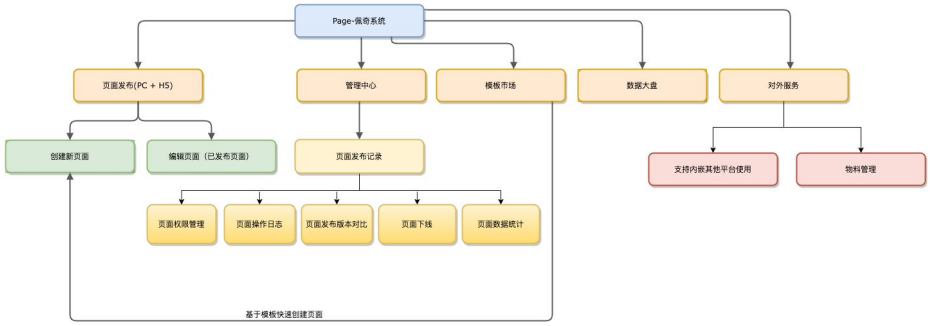


图 4　Page- 佩奇平台功能流程图

3.3 实现原理

3.3.1 基础服务

Page- 佩奇平台的基础服务有四个部分，包括物料服务、编译服务、产品赋能、扩展服务。

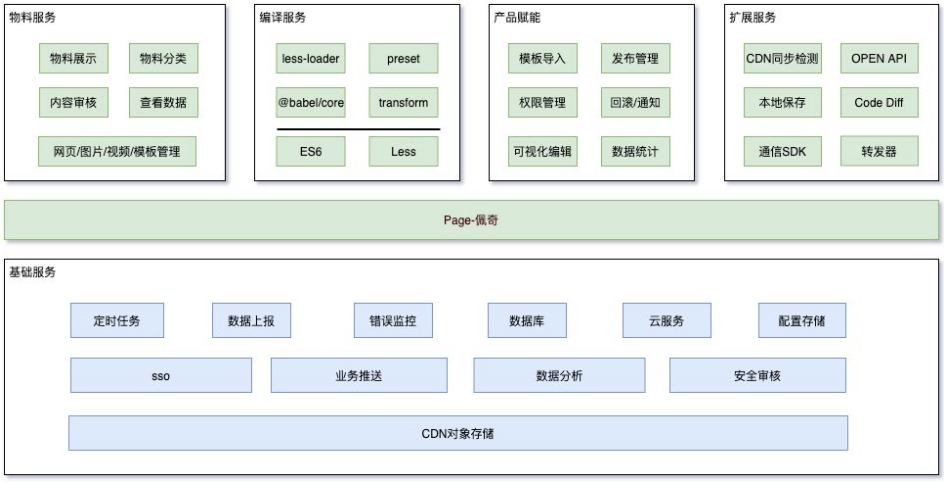


图 5　整体架构图

3.3.2 核心架构

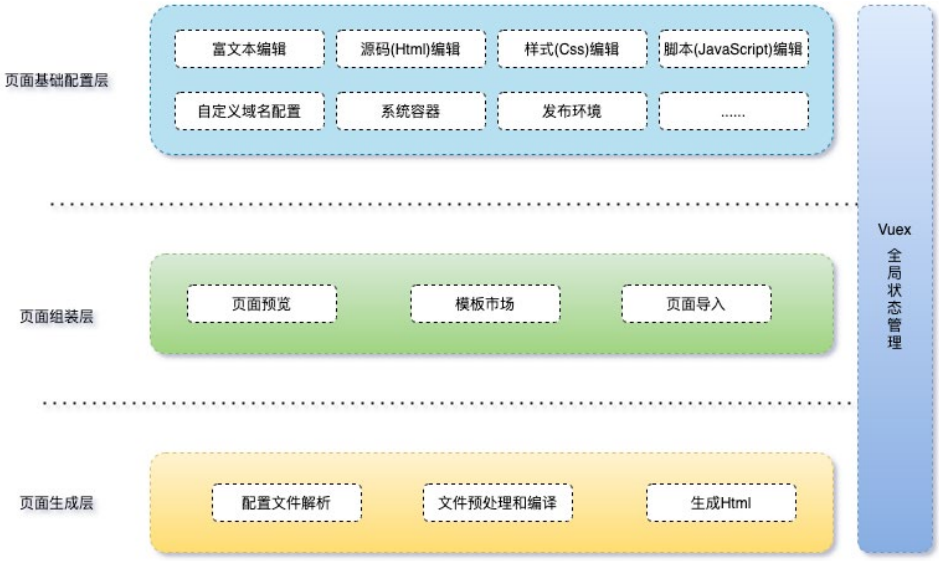


图 6　核心架构图

Page- 佩奇平台核心架构主要包含页面基础配置层、页面组装层以及页面生成层。我们通过 Vuex 全局状态对数据进行维护。

- 页面基础配置层主要提供生成页面的各种能力，包括富文本的各种操作能力、编辑源码（HTML、CSS、JavaScript）的能力、自定义域名配置、适配的容器（PC/H5）、发布环境等。
- 页面组装层则会基于基础配置层所提供的的能力，实现页面的自由编辑，承载大量的交互逻辑，用户的所有操作都在这一层进行。
  - 业务 PV 和 UV 埋点，错误统计，访问成功率上报。
  - 自动适配 PC 和移动端样式。
  - 内网页面显示外网不可访问标签。
- 页面生成层则需要根据组装后的配置进行解析和预处理、编译等操作，最终生成 HTML、CSS、JavaScript 渲染到网页当中。

3.3.3 关键流程

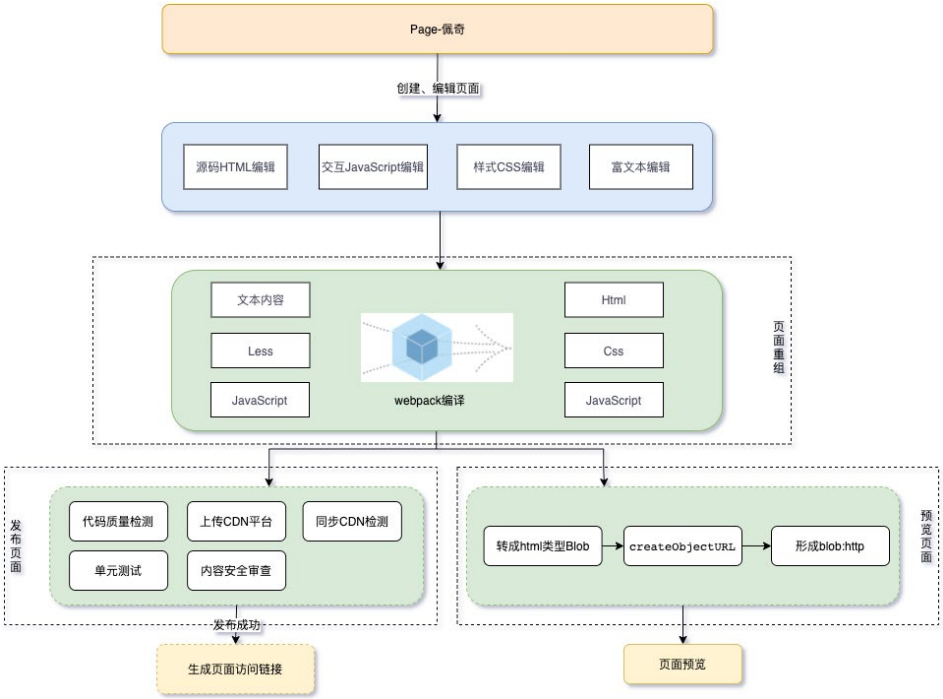


图 7　关键流程图

如上图 7 所示，平台的核心流程主要包含页面创建之后的页面预览、编译服务、生成页面。

- **页面预览**：创建、编辑之后的页面，将会根据内容进行页面重组，对样式和 JavaScript 进行预编译之后，对文本 +JavaScript+CSS 进行组装，生成 HTML 代码块，然后将代码块转换成 Blob URL，最终以 iframe 的方式预览页面。
- **编译服务**：文件树状结构和代码发送请求到后端接口，基于 Webpack 将 Less 编译成 CSS，ES 6 语法编译成 ES 5。通用物料使用 CDN 进行引入，不再进行二次编译。
- **生成页面**：当创建、编辑之后的页面进行发布时，服务端将会进行代码质量检测、内容安全审查、代码质量检测、单元测试、上传对象存储平台、同步 CDN 检测，最终生成页面链接进行访问。

3.3.4 多平台接入

Page- 佩奇平台也可以作为一个完善的富文本编辑器供业务系统使用，支持内嵌到其他系统内。作为消息发布等功能承载，减少重复的开发工作，同时我们配备完善的 SDK 供大家选择使用。通过 Page-SDK 可以直接触发 Page 平台发布、管理等操作，具体的流程如下图所示：

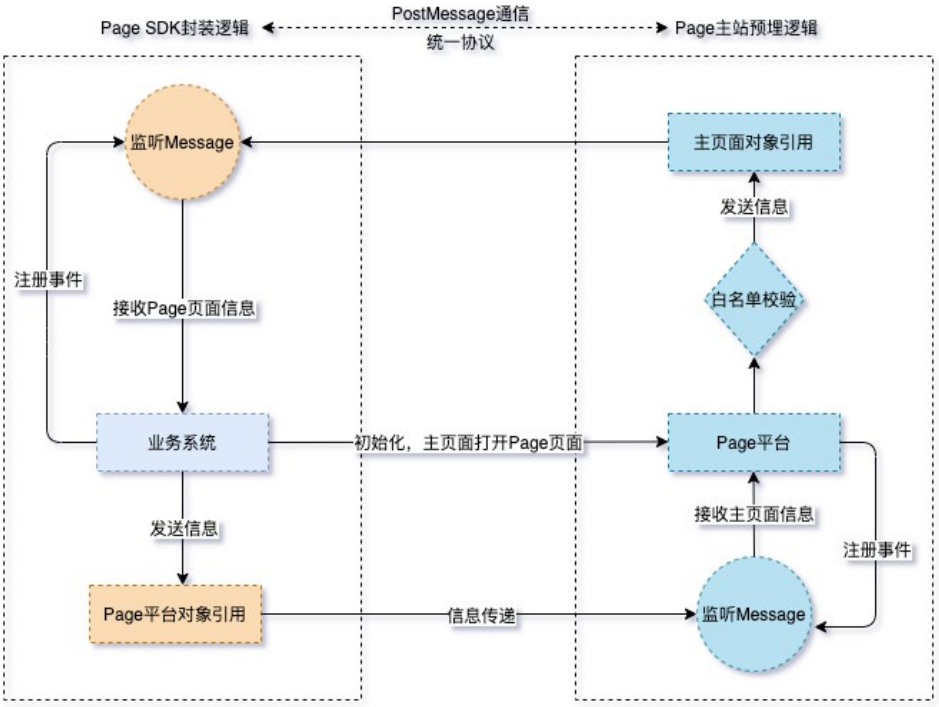


图 8　Page-SDK 流程图

3.3.5 Open API

在使用 Page- 佩奇平台的时候，美团内部一些业务方提出想要通过 Page- 佩奇平台进行页面的发布，同时想要拿到发布的内容做一些自定义的处理。于是，我们提供了 Open API 开放能力，支持以 HTTP 和 Thrift 两种方式进行调用。下面主要讲一下 Thrift API 实现的思路，首先我们先了解下 Thrift 整体流程：

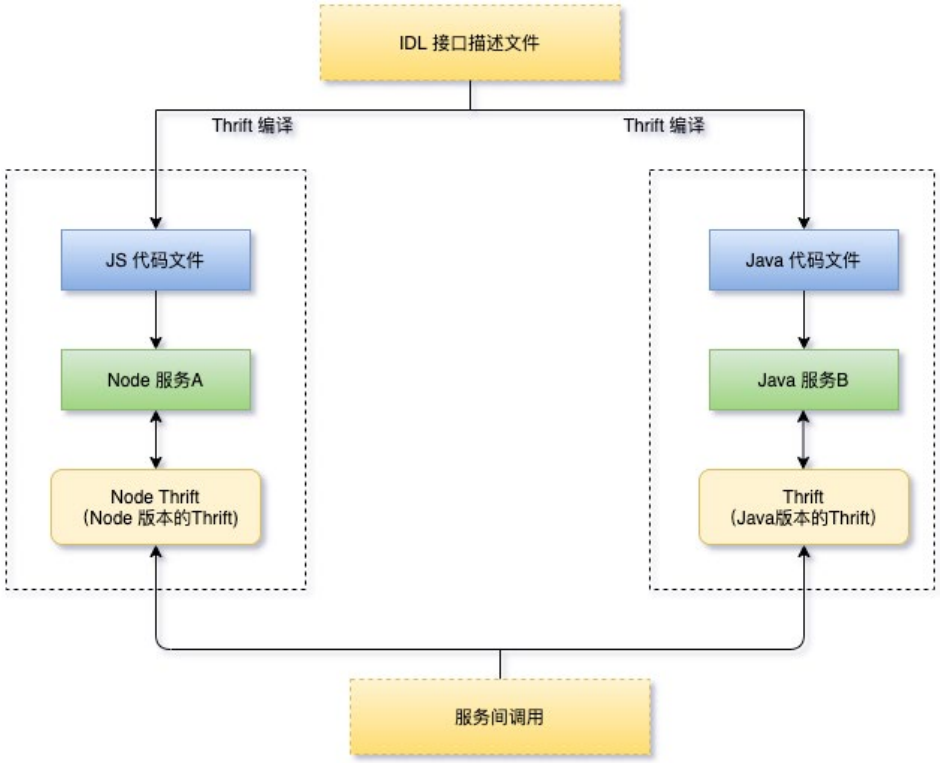


图 9　Thrift 整体流程图

Thrift 的主要使用过程如下：

- 服务端预先编写接口定义语言 IDL（Interface Definition Language）文件定义接口。
- 使用 Thrift 提供的编译器，基于 IDL 编译出服务语言对应的接口文件。
- 被调用服务完成服务注册，调用发起服务完成服务发现。
- 采用统一传输协议进行服务调用与数据传输。

下面具体讲讲，Node 语言是如何实现和其他服务语言实现调用的。由于我们的服务使用的 Node 语言，因此我们的 Node 服务就充当了服务端的角色，而其他语言（Java 等）调用就充当了客户端的角色。

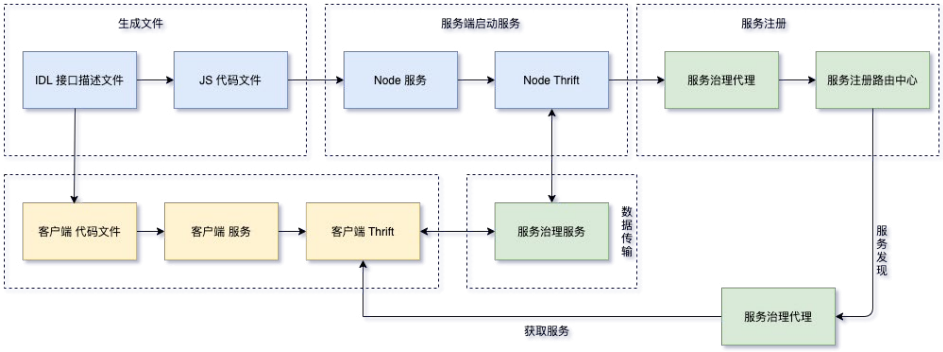


图 10　Thrift 使用详细流程图

- **生成文件**：由服务端定义 IDL 接口描述文件，然后基于 IDL 文件转换为对应语言的代码文件，由于我们用的是 Node 语言，所以转换成 JavaScript 文件。
- **服务端启动服务**：引入生成的 JavaScript 文件，解析接口、处理接口、启动并监听服务。
- **服务注册**：通过服务器内置的“服务治理代理”，将服务注册到美团内部的服务注册路由中心（也就是命名服务），让服务可被调用方发现。
- **数据传输**：被调用时，根据“服务治理服务”协议序列化和反序列化，与其他服务进行数据传输。

目前，美团内部已经有相对成熟的 NPM 包服务，已经帮我们实现了服务注册、数据传输、服务发现和获取流程。客户端如果想调用我们所提供的的 Open API 开放能力，首先申请 AppKey，然后选择使用 Thrift 方式或者 HTTP 的方式，按照所要求的参数进行请求调用即可。

3.4 方案实践

3.4.1 H5 协议

能力：富文本编辑。**描述**：提供富文本可视化编辑，产品和运营无需前端就可以发布和二次编辑页面。**场景**：文本协议，消息通知，产品 FAQ。

具体案例：

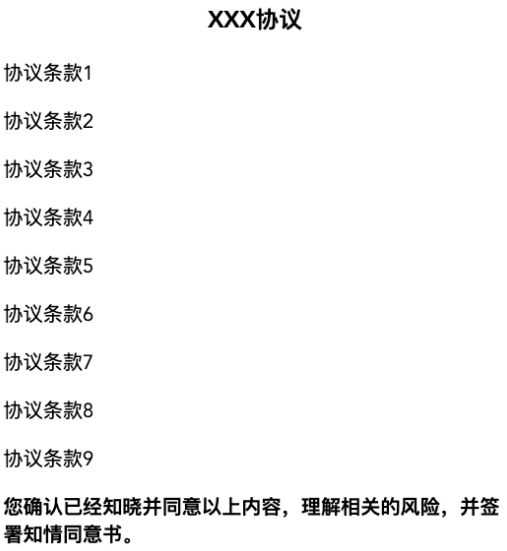


图 11　H5 静态文本协议案例

3.4.2 业务自定义渲染

能力：开放 API（Thirft + HTTP）。**描述**：提供开放 API，支持业务自定义和样式渲染到业务系统，同时解决了 iframe 体验问题。**场景**：客户端、后端、小程序的同学，可根据 API 渲染文案，实现动态化管理富文本信息。

具体案例：

小程序使用组件、Vue 使用 v-html 指令实现动态化渲染商品选择说明。

```
    {
    "code": 0,
    "data": {
      "tag": " 苹果，标准 ",
      "title": " 如何挑选苹果 ",
      "html": "<h1> 如何挑选苹果 </h1>><p> 以下标准可供消费者参考 </
p><ul><li> 酸甜 </li><li> 硬度 </li></ul>",
      "css": "",
      "js": "",
      "file": {}
    },
    "msg": "success"
}
```

3.4.3 投放需求

能力：WebIDE 代码编辑。**描述**：开发基于 WebIDE 代码开发工作，基于渠道和环境修改下载链接，能够做到分钟级支撑。**场景**：根据产品创建静态页面进行逻辑和样式开发。

具体案例：

```
var ua = window.navigator.userAgent
var URL_MAP = {
    ios: 'https://apps.apple.com/cn/app/xxx',
    android: 'xxx.apk',
    ios_dpmerchant: 'itms-apps://itunes.apple.com/cn/app/xxx'
}

if (ua.match(/android/i)) location.href = URL_MAP.android
if (ua.match(/(ipad|iphone|ipod).*os\s([\d_]+)/i)) {
    if (/xx\/com\.xxx\.xx\.mobile/.test(ua)) {
        location.href = URL_MAP.ios_dpmerchant
    } else {
        location.href = URL_MAP.ios
    }
}
```

3.4.4 客户端通信中间页

能力：WebIDE 代码编辑 + 物料平台。**描述**：通过物料平台，引入公司客户端桥 SDK，可以快速完成客户端通信需求。方便前端调试客户端基础桥功能。**场景**：客户端跳转，通信中间页。

具体案例：

```
// 业务伪代码
XXX.ready(() => {
    XXX.sendMessage({
            sign: true,
        params: {
            id: window.URL
        }
    }, () => {
        console.error('通信成功')
    }, () => {
        console.error('通信失败')
    })
})
```

3.4.5 业务系统内嵌 Page

能力：提供胶水层 Page-SDK，连接业务系统和 Page。**描述**：业务系统与 Page-佩奇平台可进行通信，业务系统可调用 Page 发布、预览、编辑等功能，Page 可返回业务系统页面链接、内容、权限等信息。减少重复前后端工作，提升研发效率。**场景**：前端富文本信息渲染，后端富文本信息管理后台。

具体案例：

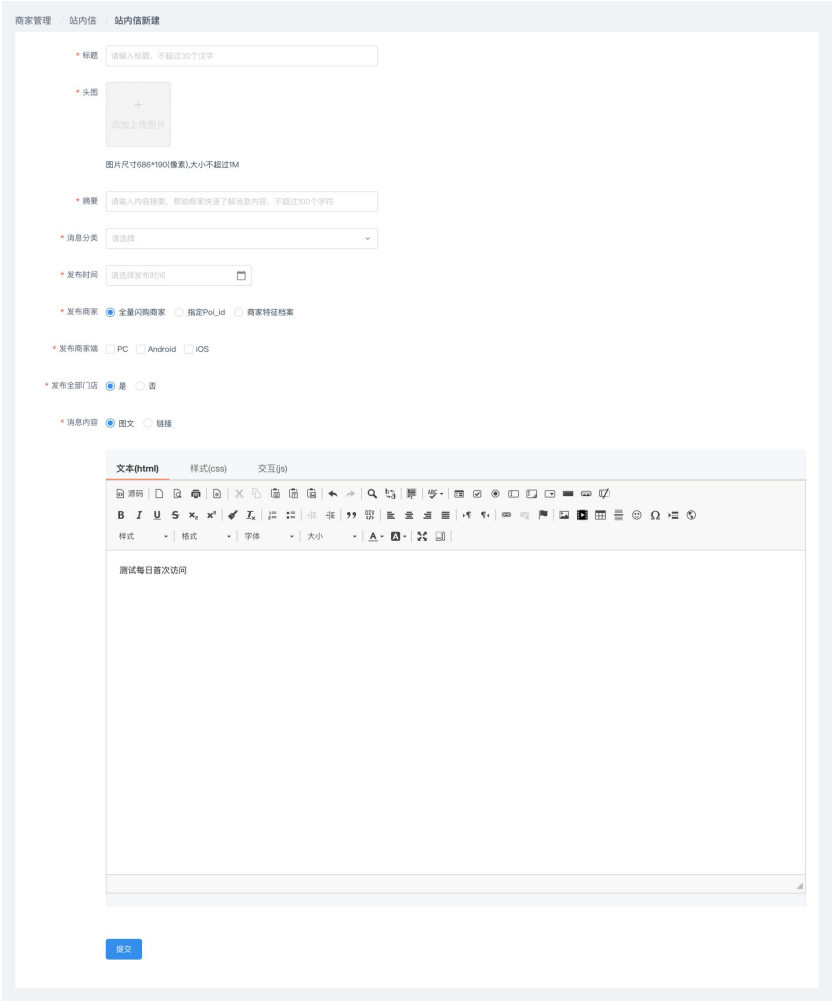


图 12　业务系统内嵌 Page 案例

3.5 业务成绩

截止目前数据统计，Page- 佩奇平台生成网页 5000 多个，编辑页面次数 16000 多次，累计页面访问 PV 超过 8260 万。现在，美团已经有十多个部门和三十多条业务线接入并使用了 Page- 佩奇平台。

图 13　Page 平台每日生成页面统计

四、总结与展望

富文本编辑器和 WebIDE 不仅是复杂的系统，而且还是比较热门的研究方向。特别是在和美团的基建结合之后，能够解决团队内部很多效率和质量问题。这套系统还提供了语法智能提示、Diff 对比、前置检测、命令行调试等功能，不仅要关注业务发布出去页面的稳定性和质量，更要有内置的一系列研发插件，主动帮助研发提高代码质量，降低不必要的错误。

经过长期的技术和业务演进，Page- 佩奇平台已经能够有效地帮助研发人员大幅提升开发效率，具备初级的 Design To Code 能力，但是仍有许多业务场景值得去我们探索。我们也期待优秀的你参与进来，一起共同建设。

- **WebIDE 融合**：完善基础设施建设和功能需求，更好地支持 Vue、React、ES 6、TS、Less 语法，预览模式采用浏览器编译，能有效地提高预览的速度，发布使用后端编译的模式。
- **研发流程链路**：针对代码进行有效评估，包括 ESlint、代码重复率、智能提示是否可以三方库替代。出具开发代码质量、业务上线的质量报告。
- **综合研发平台**：减少团队同学了解整体基建的时间成本，内置了监控、性能、任务管理等功能，提升业务开发效率。建设自动化日报、周报系统，降低非开发工作量占比。
- **物料开放能力**：接入公共组件平台，沉淀更多的物料，快速满足产品更多样化的需求。

五、作者简介

高瞻、宇立、肖莹、浩畅，来自美团医药终端团队。王咏、陈文，来自美团闪购终端团队。

六、招聘信息

美团医药长期招聘 Android、iOS、FE 前端工程师，坐标在北京和成都。感兴趣的同学可将简历发送至：sunyuli@meituan.com（邮件主题请注明：美团医药终端）。

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

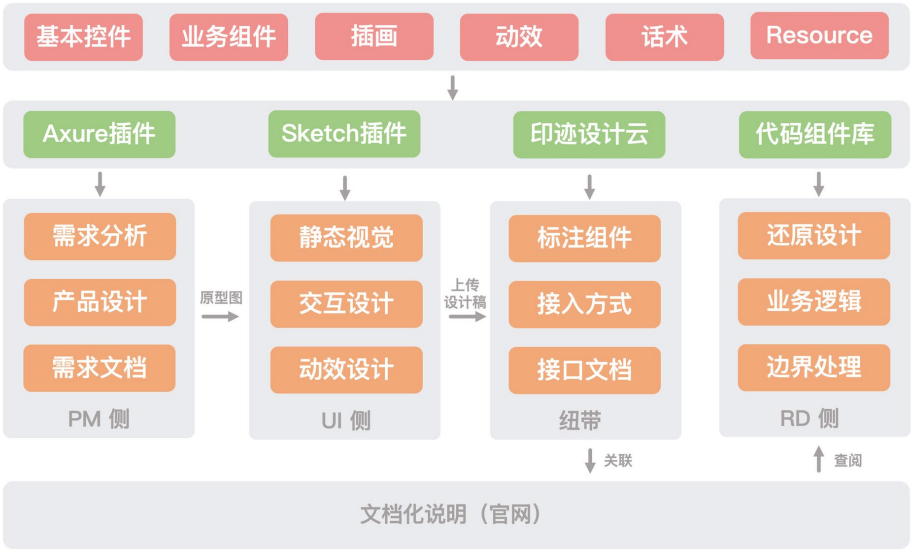
为了帮助更多的业务部门定制符合一致性原则的专属设计风格，外卖技术部在实践中不断总结经验，开发了一套通用的 UI 一致性解决方案。该方案通过 UI 一致性工具链落地项目建设，并打造一整套的闭环 UI 开发流程，目前已经取得了一定的成果，以下系具体方案的介绍。

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

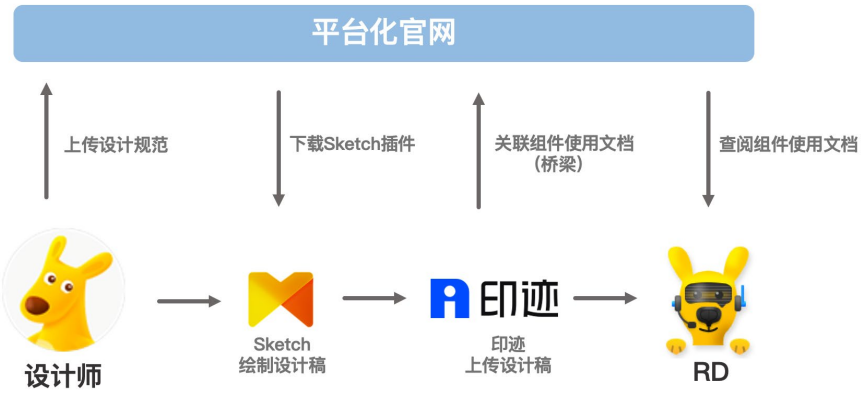
关联了官网中该组件的使用说明，是代码组件库与官网的纽带。



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

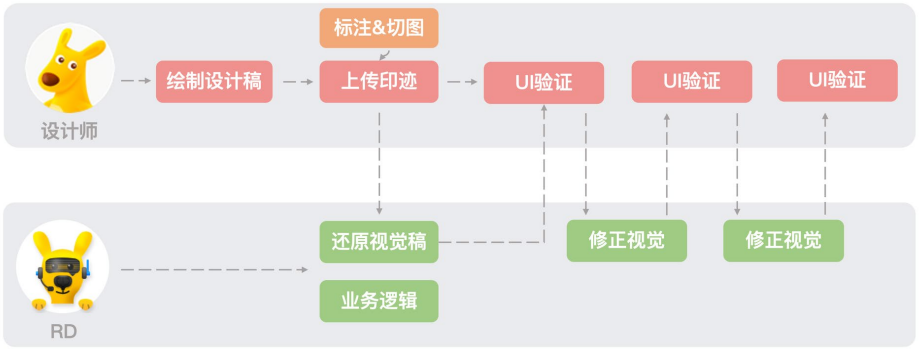
经过一段时间的 UI 一致性建设，在资源一致性方面，外卖 App 团队已经完成了近百个 Iconfont 的替换工作，有效减小了安装包的体积。在组件代码库建设方面，完成组件替换三十多处，中等业务需求平均节约 3pd 人力；在工具链方面，根据 UI/UE 提供的数据，对于强依赖设计资源的需求，在使用积木 Sketch 插件后，提效能够达到 30% 以上，对于 UI 资源依赖不强的流程需求，平均提效可以达到 50% 以上。

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。

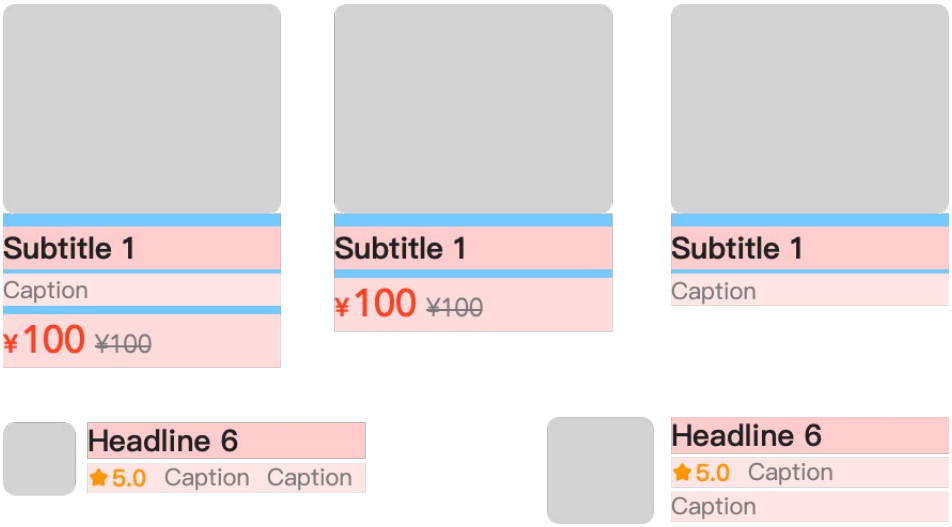
我们将走查结果与设计师反复交流，发现复用性较高的组件大体可以分为两类：第一类“基础控件”，也就是类似于标签、按钮、开关等具备基础功能的元素，对应原子理论中的原子；第二类“业务组件”，类似于商品卡片等，是由“基础控件”组成（比如商品卡片由“标签控件”与“图片控件”组成），同时“业务组件”还能相互组合，成为更高阶的“复杂组件”，类似于原子理论中的分子。“业务组件”的组合又是千变万化的，不同样式的业务组件可以组成类似“商家列表”、“菜品列表”等“模板”，而“模板”与“基本控件”组合在一起，就成为了“页面”。



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。



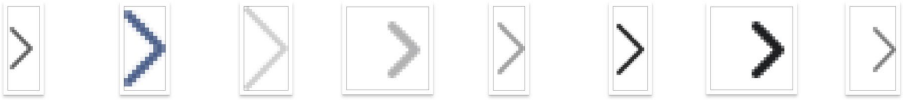
使用 iconfont 替换项目中的图片

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。

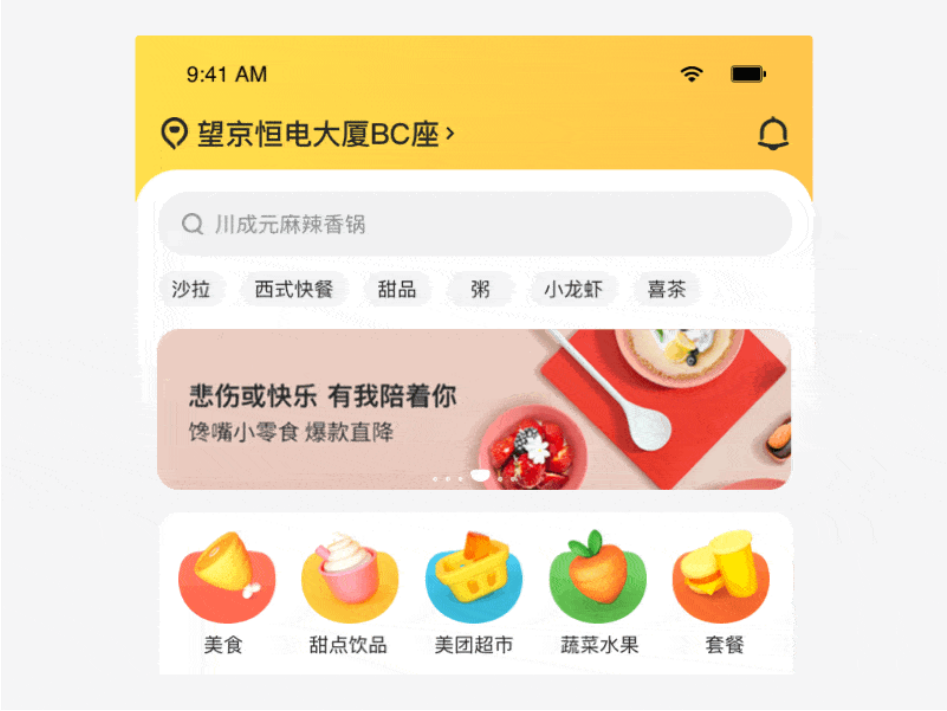


一致性项目实施前项目中的相似图片

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。

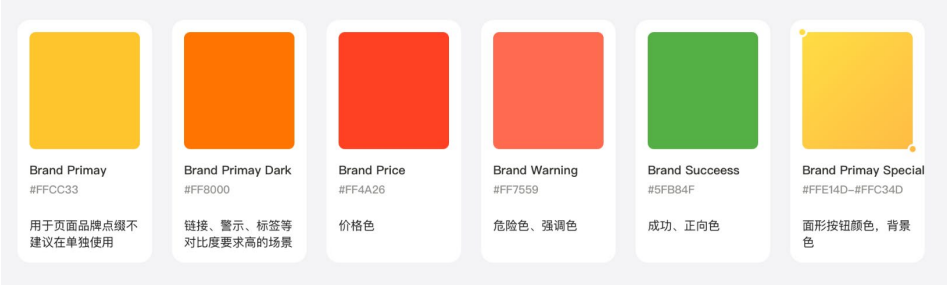
随着外卖业务的不断增加，动效的使用比重在不断增加，重要性日渐凸显，也是增强用户体验与竞品拉开差距的重要因素，因此，统一规范的使用动效尤为重要。通过动效库建设，UI 层面可以承载品牌、传递情感，加深用户对 App 的印象，让用户放松、愉悦；RD 层面同一组件可在多场景直接复用，还降低了研发成本。



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。



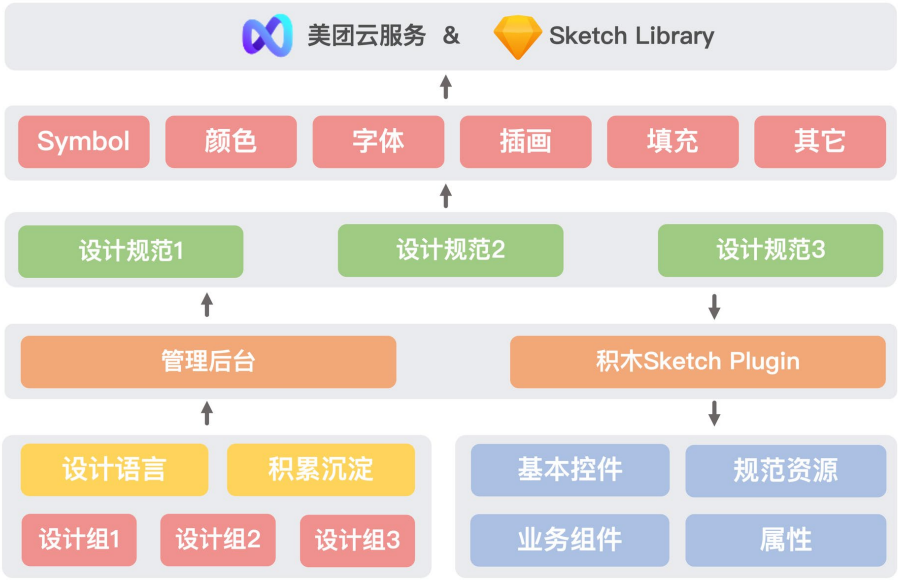
定义字体使用规范

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

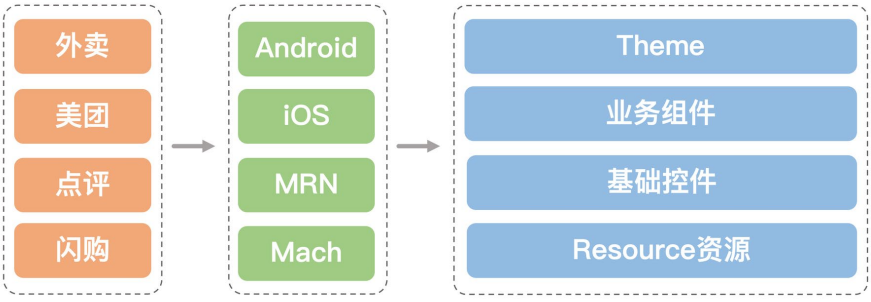
库上传至云端并与整个设计团队中成员共享，并可实现设计资产的版本管理。通过将 Sketch Library 存储在后台物料市场，设计师可以与团队成员共享组件（Symbol），Library 可以实现“一处更改，处处生效”，即使是关联了远程组件库历史的设计稿检测到更新时，也会收到 Sketch 通知，确保工作中使用的是最新组件。



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,
      }
  }
```

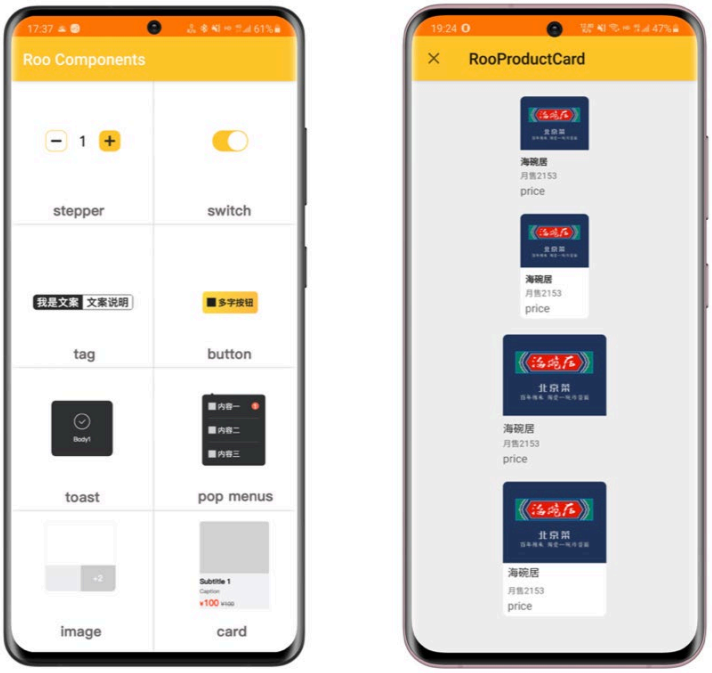
搭建全平台组件库

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。



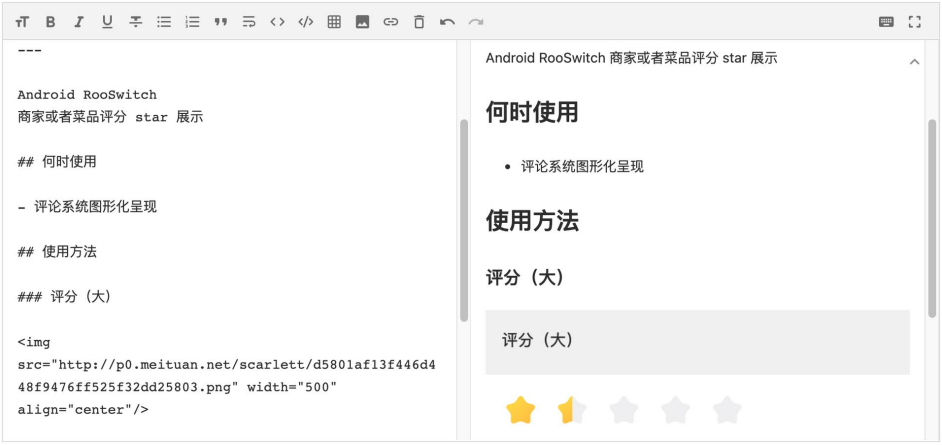
外卖平台化官网

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

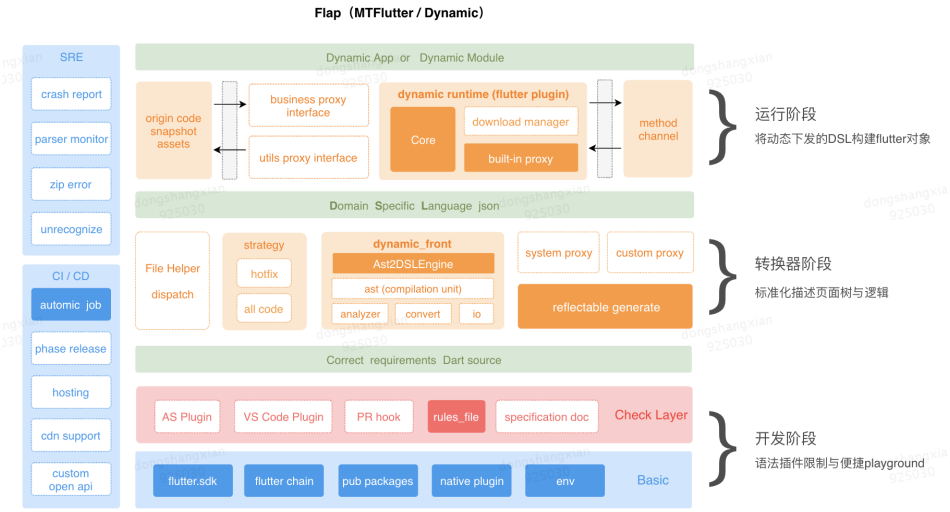


图 1　Flap 整体架构

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。

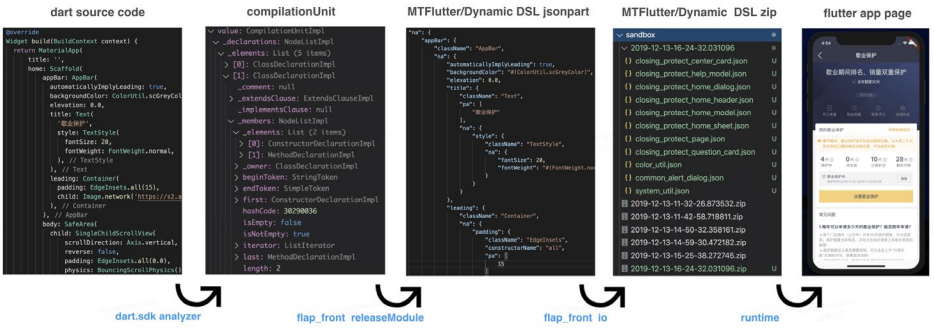


图 2　DSL-JSON 的转换步骤

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。

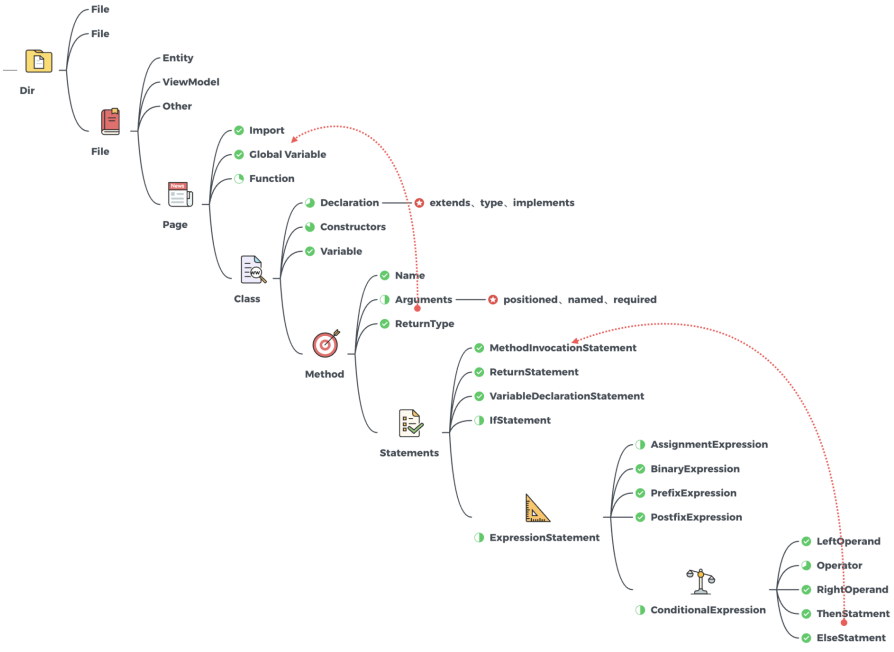


图 6　DSL 内部结构层级

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。

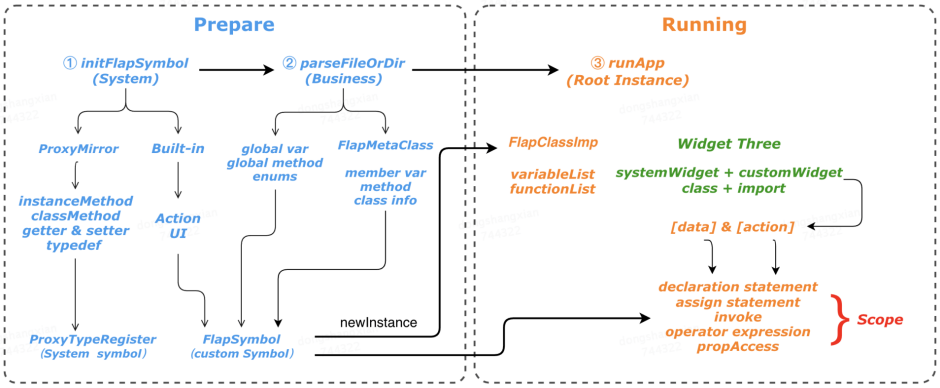


图 7　运行时原理的两大阶段

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。

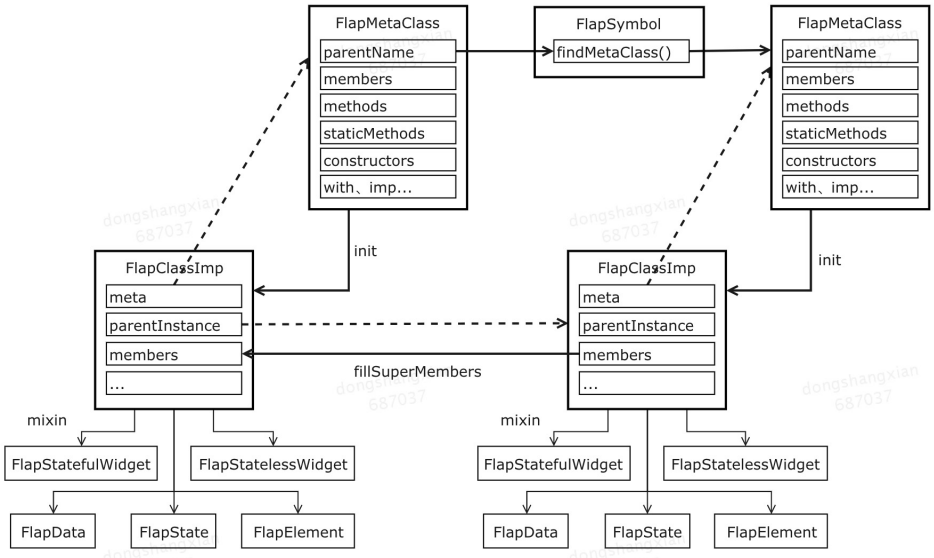


图 8　运行时模拟的元类系统

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：

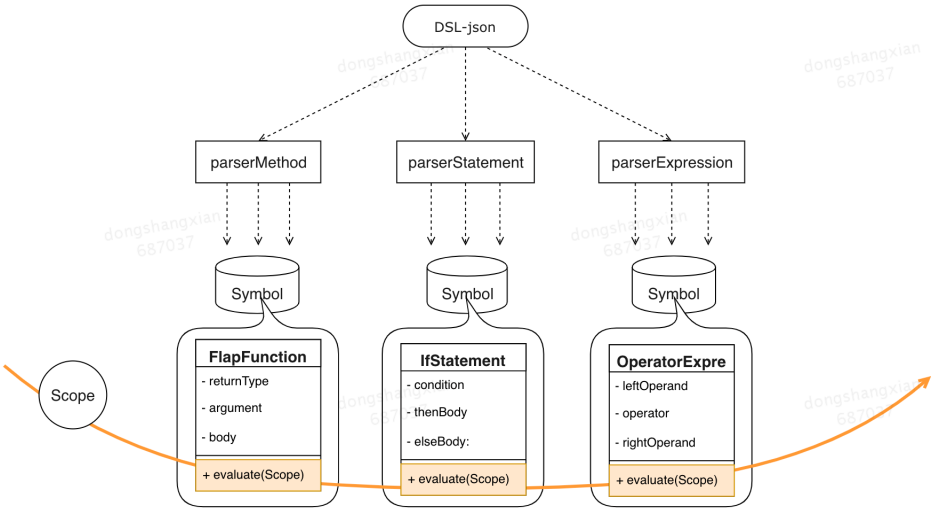


图 9　运行时 evaluate 触发链路

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。

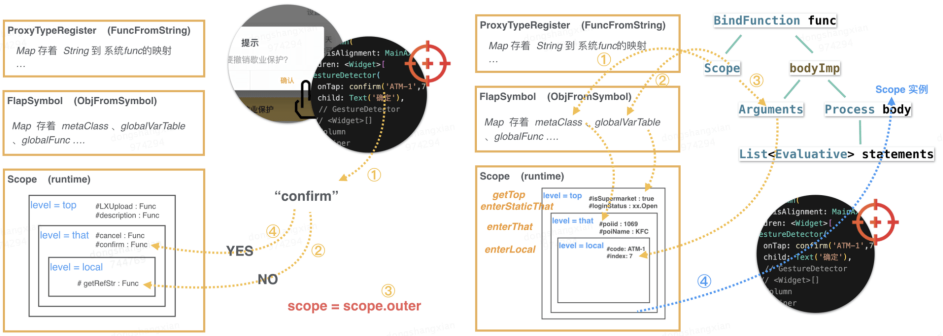


图 10　Scope 的寻找与构建

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。

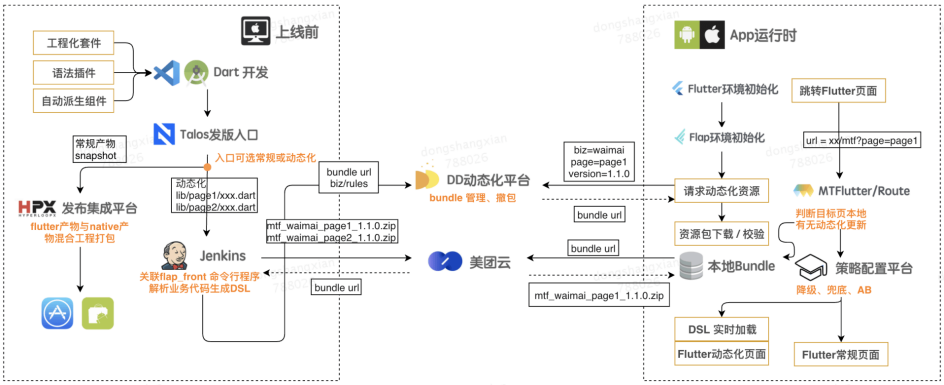


图 11　Flap 在美团内网生态

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。

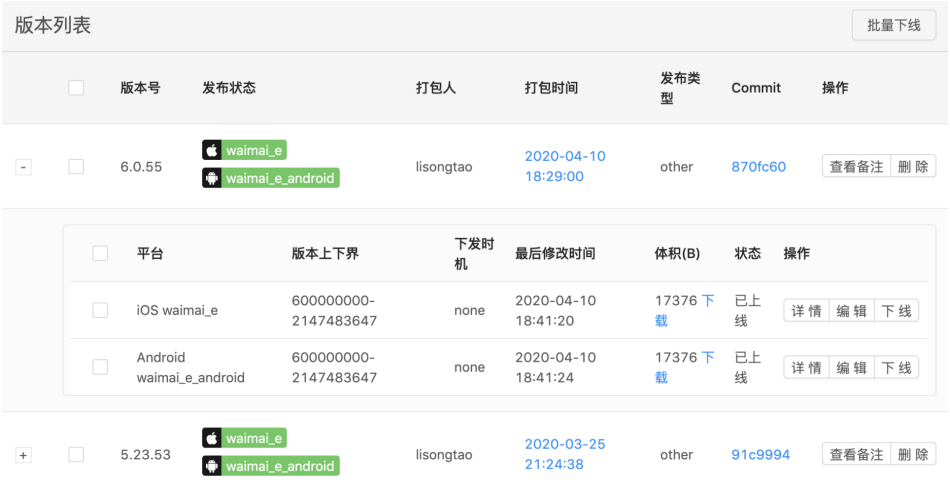


图 13　Bundle 发布系统的各项边界控制

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。

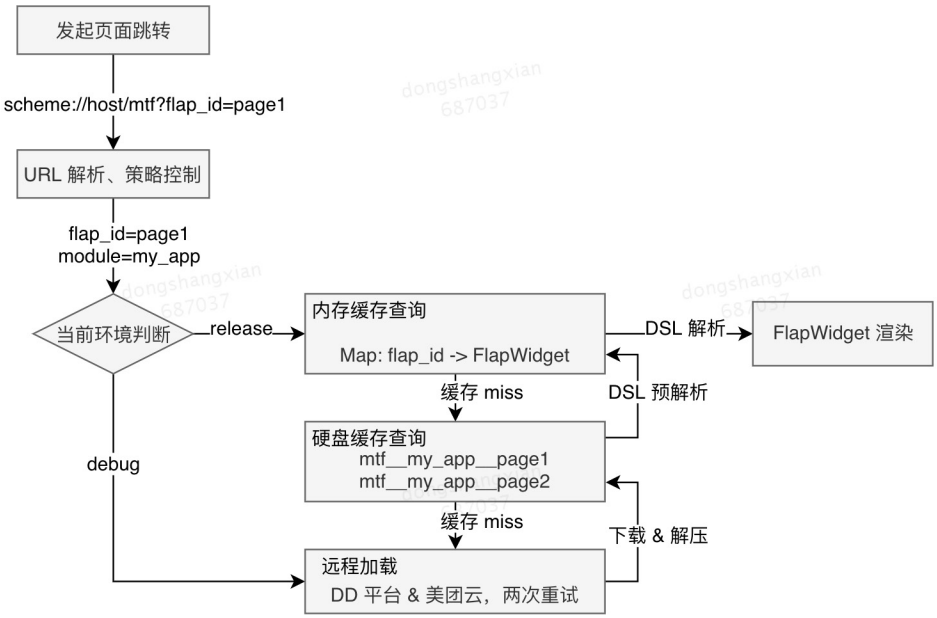


图 15　Flap 二级缓存策略

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。

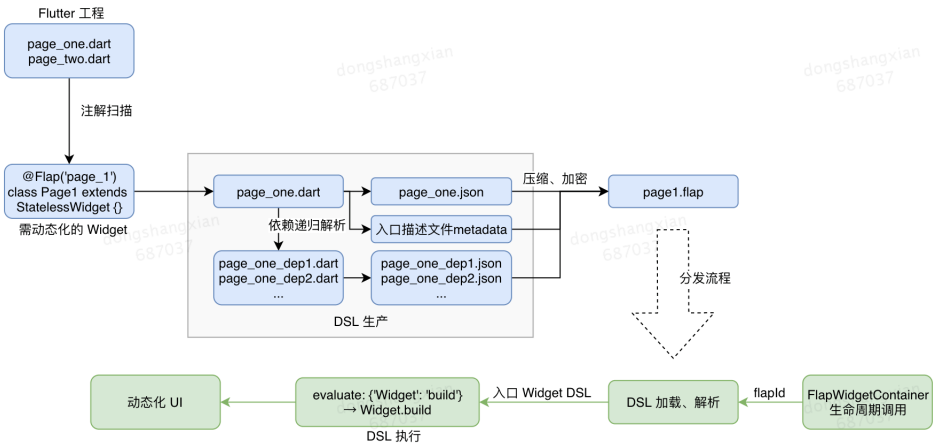


图 16　注解的扫描与 widget 构建

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。

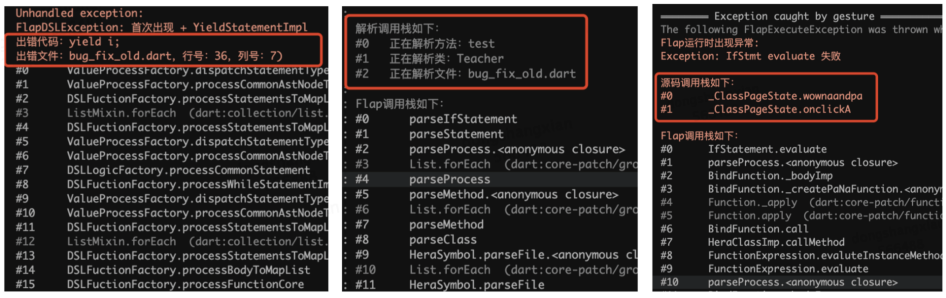


图 17　三个阶段的 Debug 定位

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。

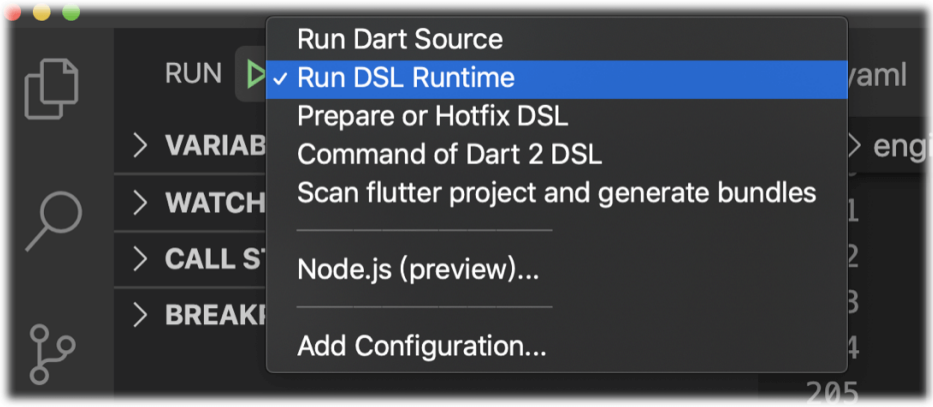


图 18　研发过程支持不同的运行模式

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

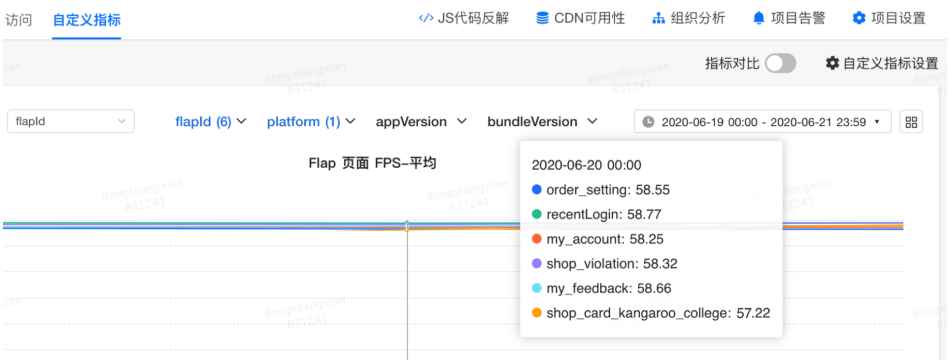


图 20　部分动态化页面 FPS 数据

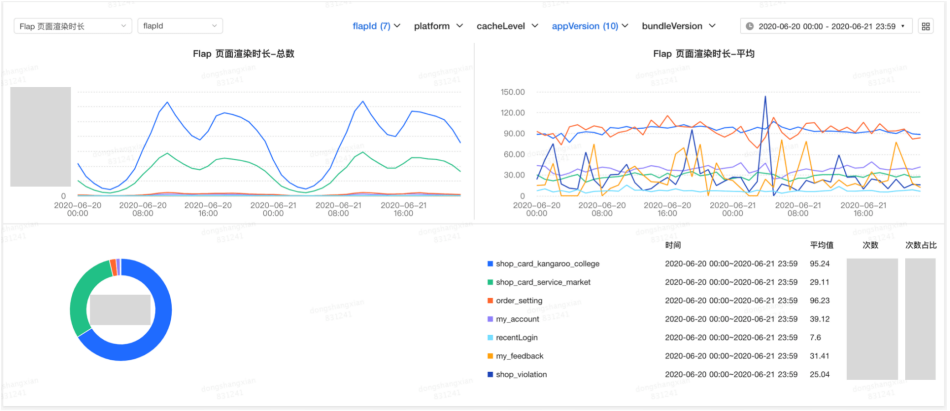


图 21　部分动态化页面渲染时长

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。

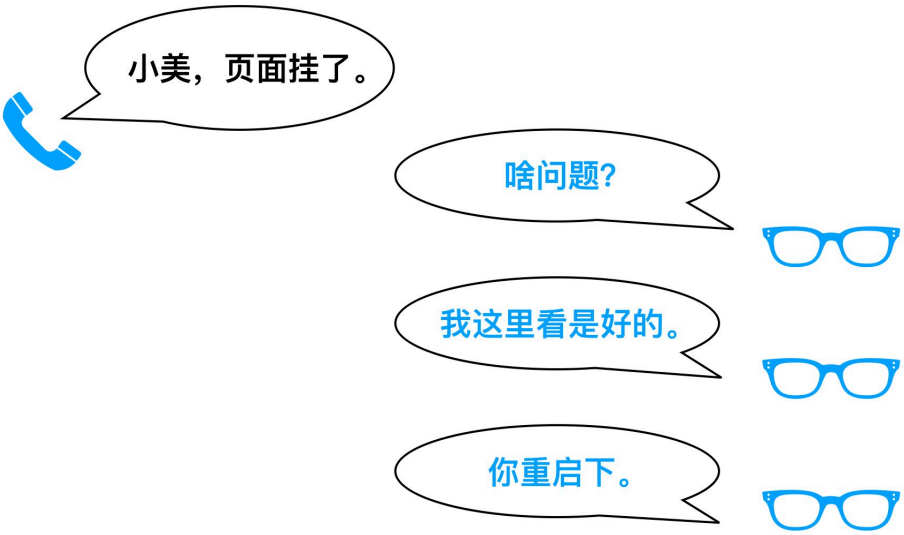
本文将围绕 Logan 在 Web 端的应用背景、技术实现、美团点评的实践、开源整体进展以及未来规划这几个方面展开介绍，以方便读者对 Logan 大前端日志系统有更加深入的了解。

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

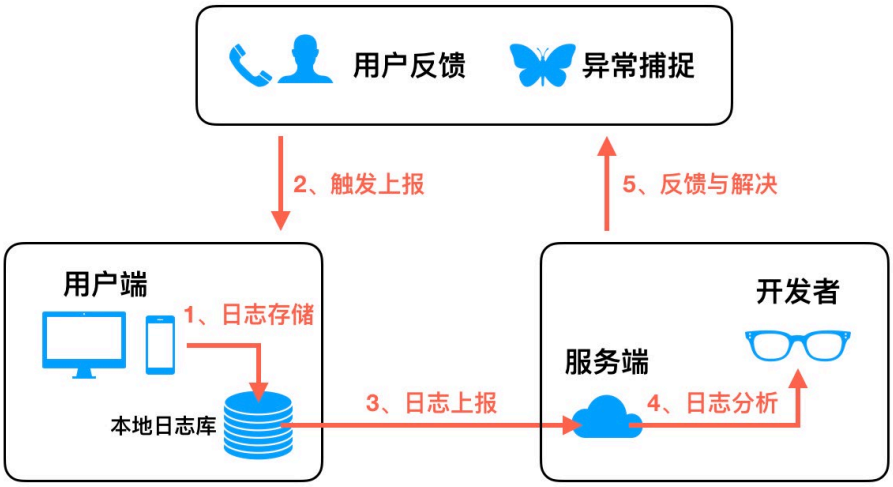
2.2.1 Logan 在各端实行的通用策略

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

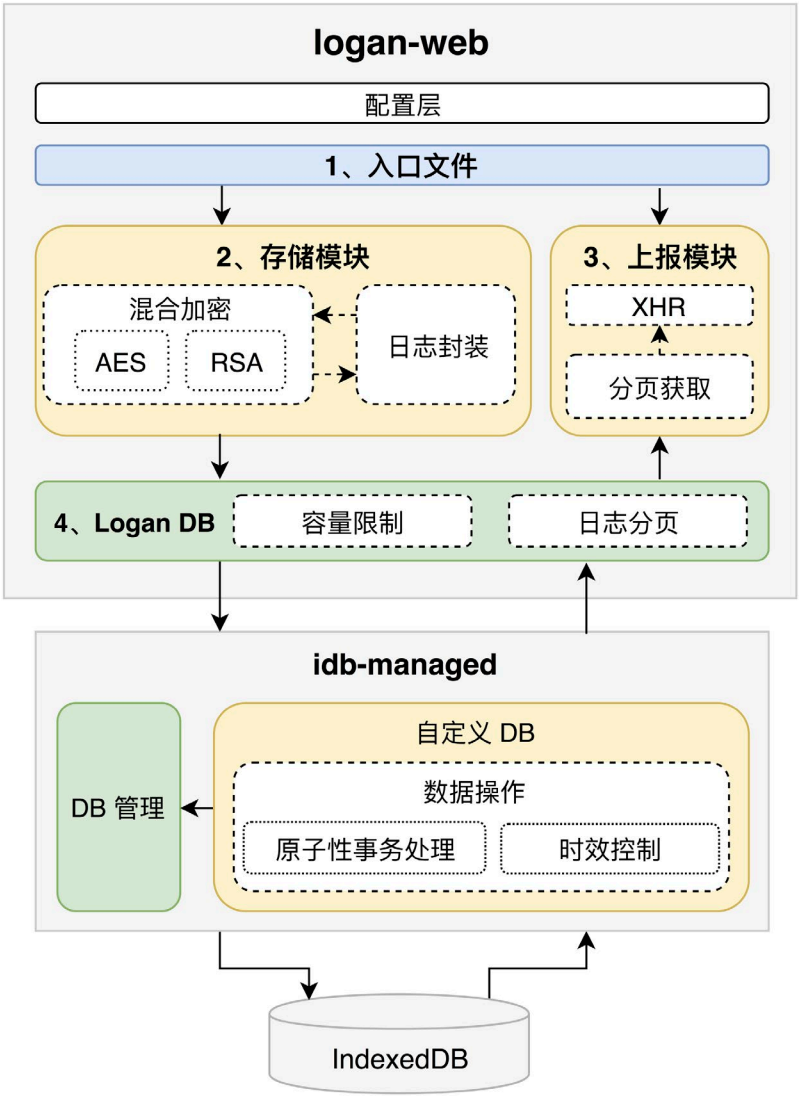
Logan Web 在底层利用了现有前端技术来实现大体积日志的安全存储：

- **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。

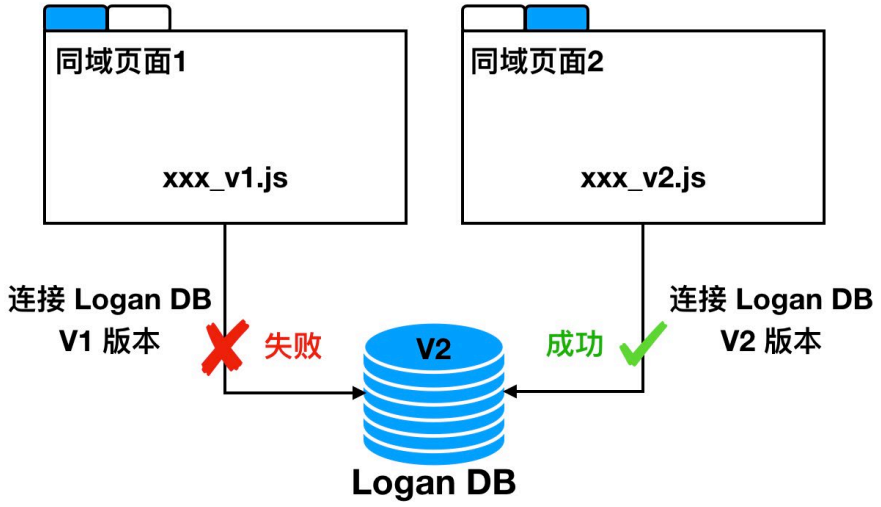
3.2 本地存储方案：idb-managed

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：

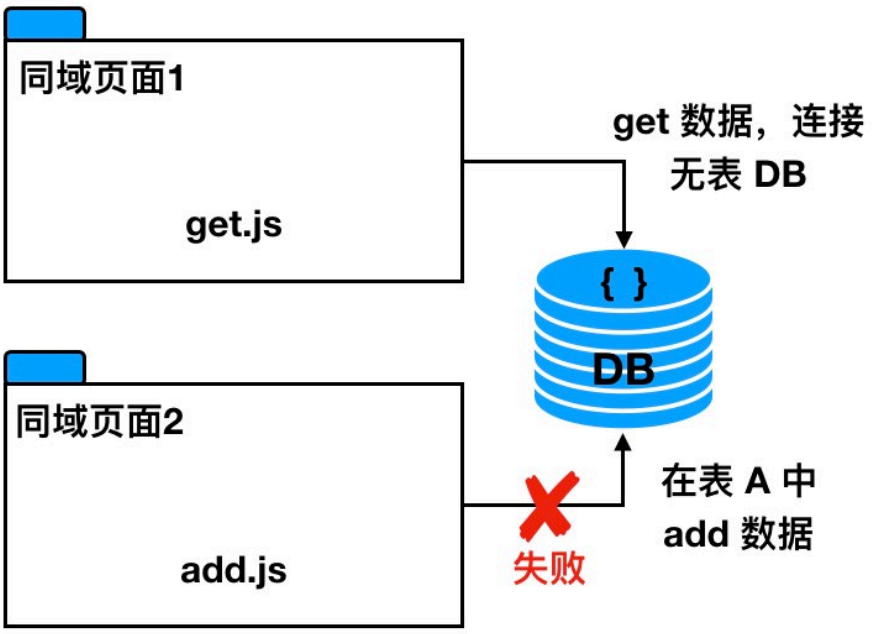
（1）DB 版本升级问题

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

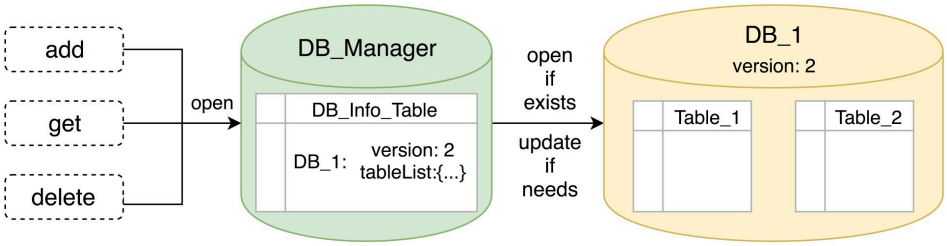
为了统一解决以上在使用 IndexedDB 时面临的困扰，我们额外封装了 idb-managed 包，该包随着 Logan Web 一起开源。

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：

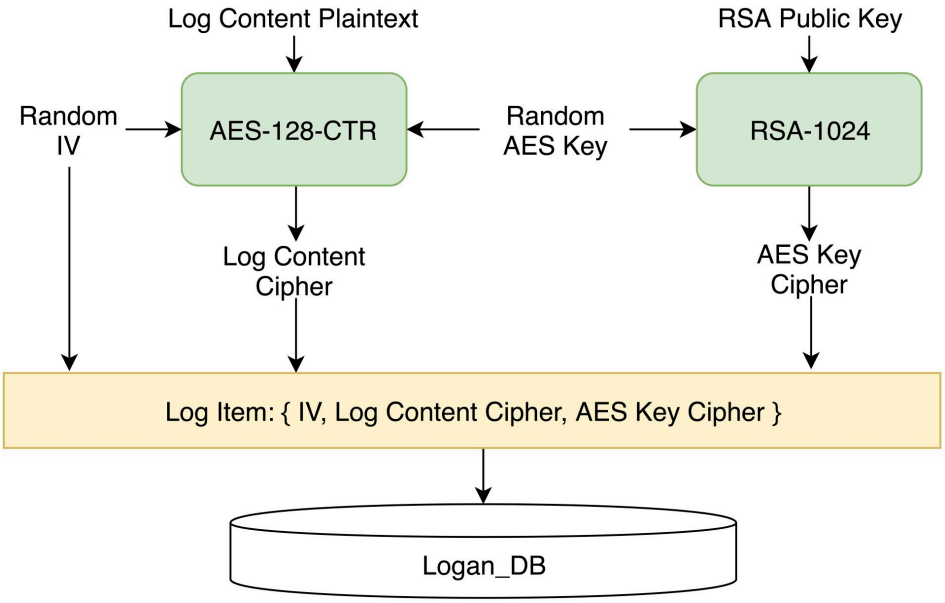
- 对称加密：保证对长内容加密的效率。
- 非对称加密：保证对称密钥的安全性。

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

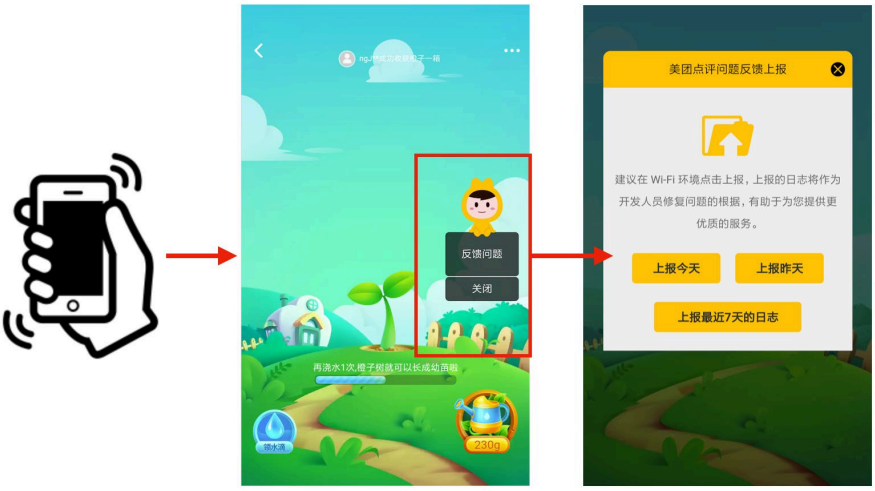
用户端的日志上报触发机制一般分为两大类：

- **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

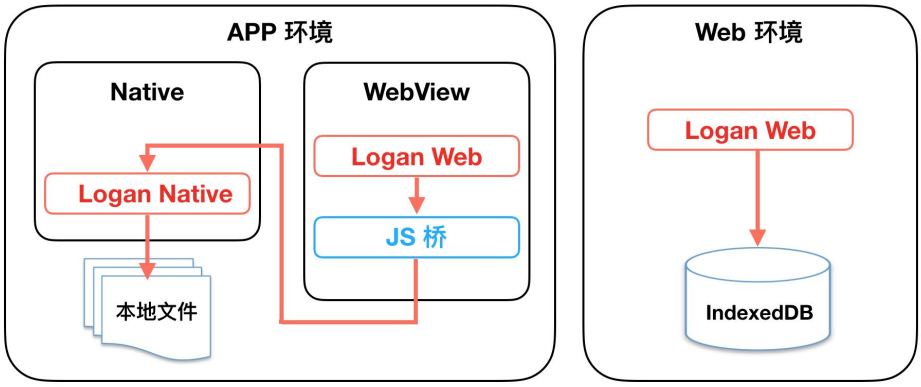
考虑到用户主动触发方式存在的弊端，Logan Web 提供了上报接口供使用者在代码层面调用。另外 Logan Web 也正在迭代实现内部的代码触发上报逻辑，提供在 Web 端内异常发生等特定场景下主动上报日志的能力。

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

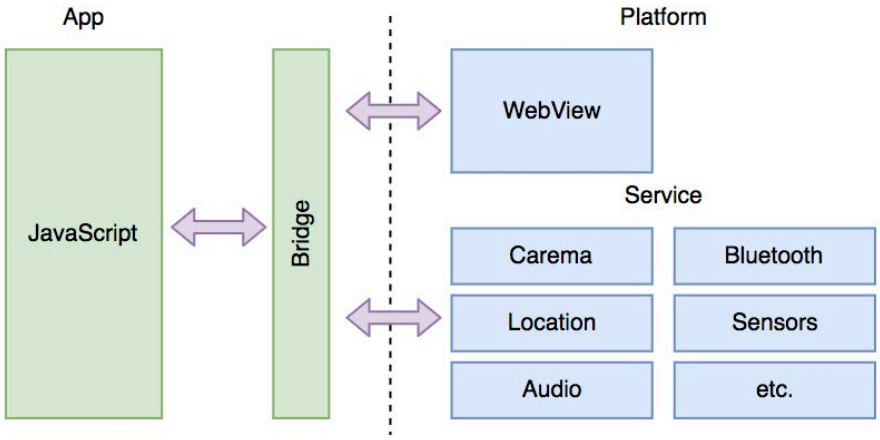
1. 背景

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。

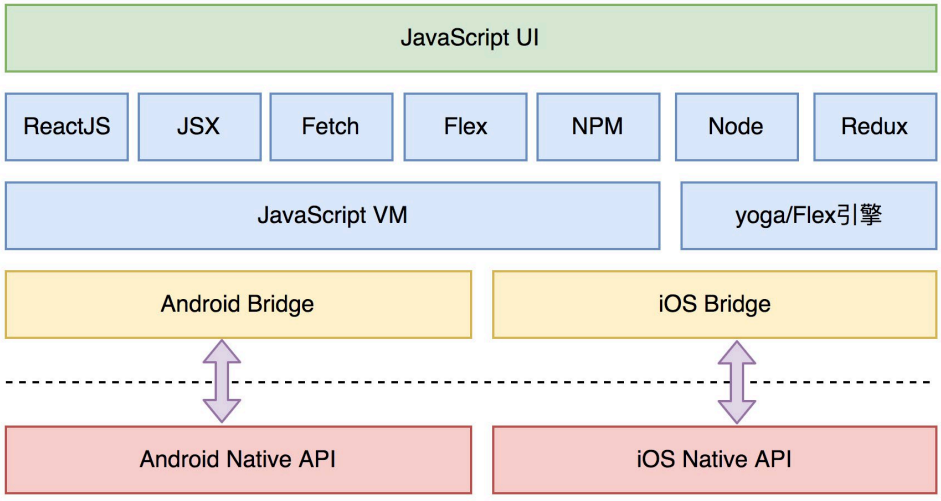
1.1.1 WebView 容器

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。



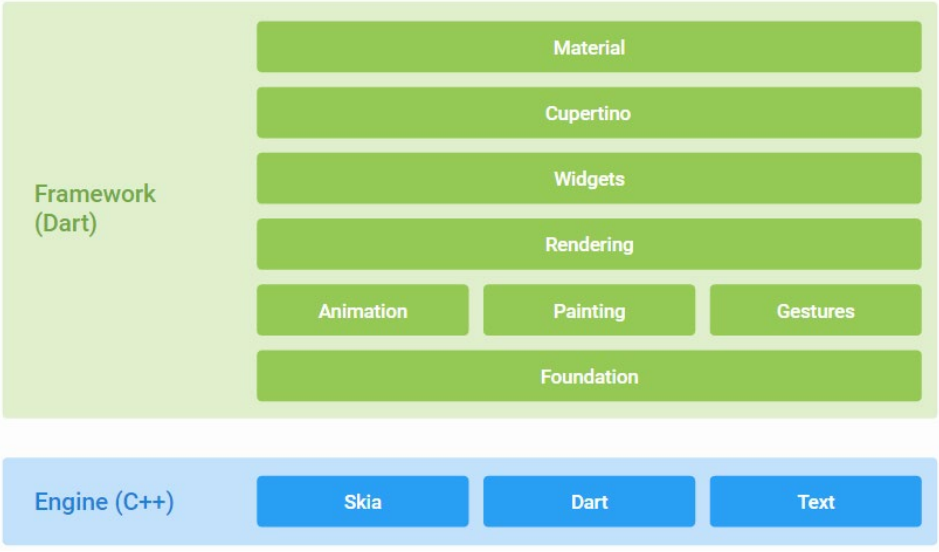
1.1.2 React Native

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。

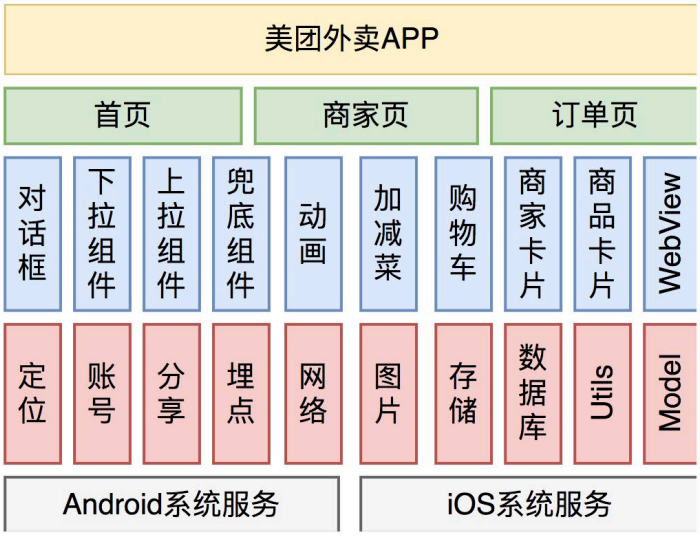
1.3 美团外卖移动端历史架构概述

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。

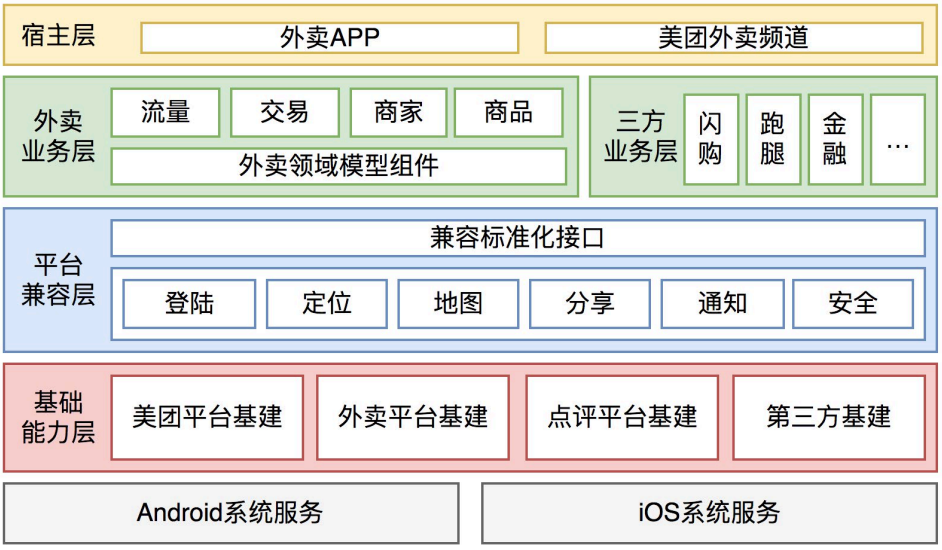
1.3.1 组件化架构

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：



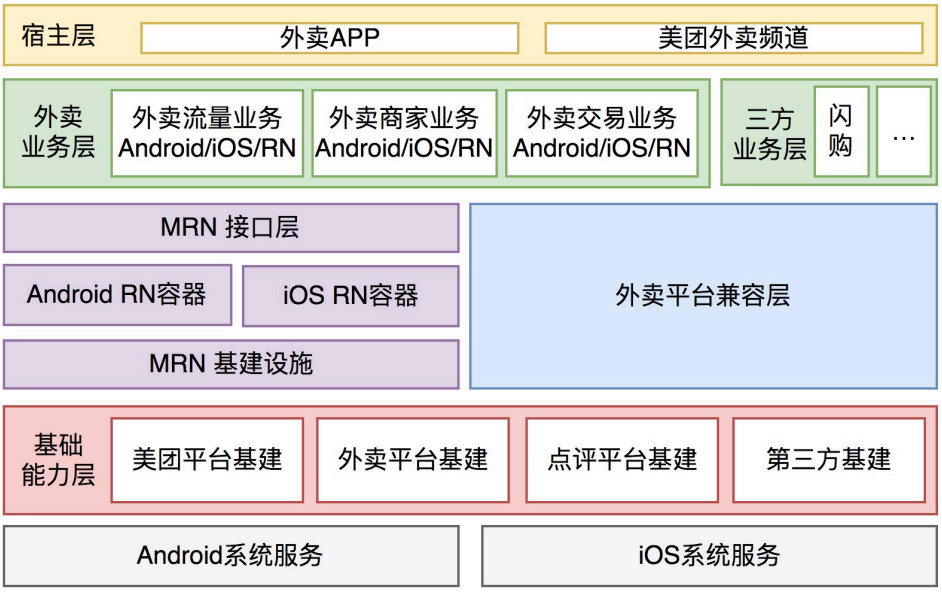
1.3.3 RN 混合架构

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

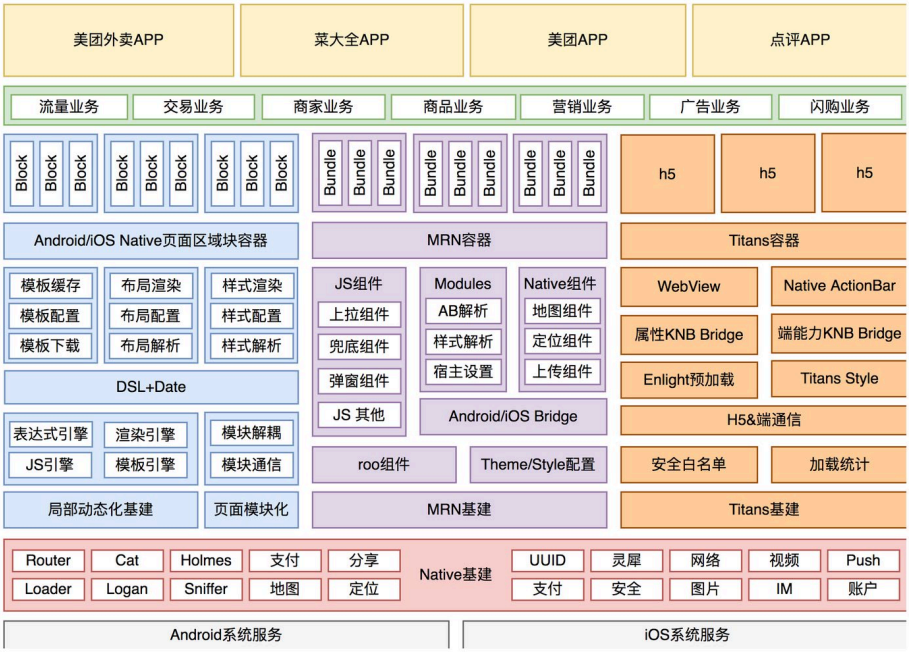
当我们把承载外卖业务的环境进行了抽象和标准化后，就可以获得以下若干点好处。首先动态化属性提升，我们可以把原有必须在客户端上写的业务放到了远端，业务的动态性得到很大的提升，具备随时上线业务的可能。对于开发过程而言，编译部署的

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。

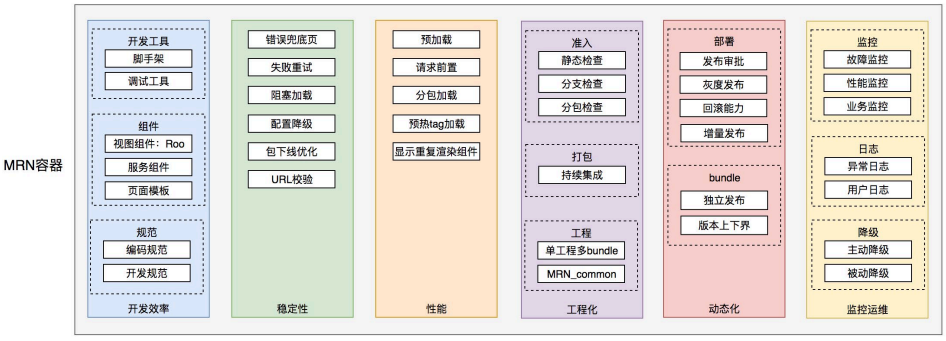
监控运维的挑战：以往的移动端架构，我们更加关注的是端本身的可用性，然而当我们演进到容器化架构的时候，仅仅关注端的可用性已经远远不能确定业务是可用的了。我们需要从端的可用性延伸出下载链路、加载链路，使用链路上的可用性，针对每个重要的环境，都做好监控运维。

3. 外卖跨端容器建设

3.1 MRN 容器

3.1.1 MRN 容器简介

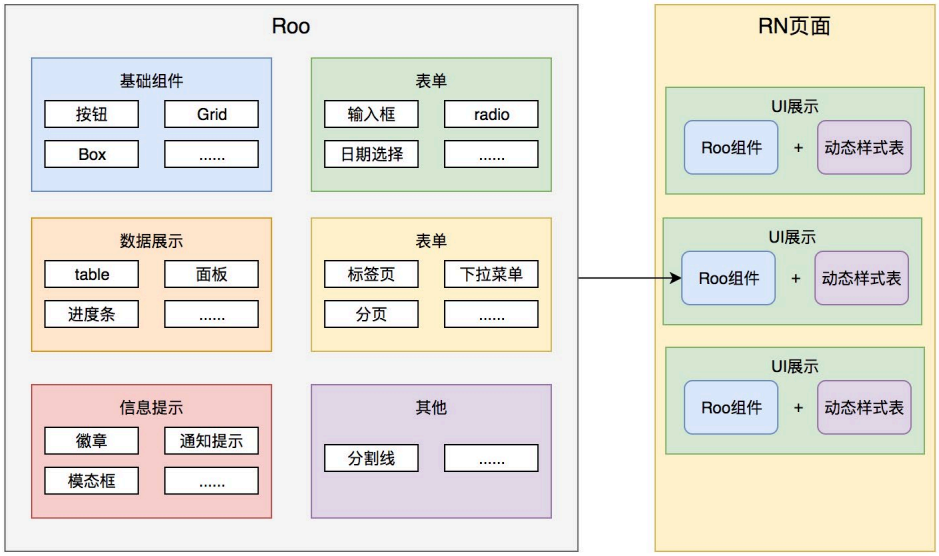
React Native 框架本身只是一个运行时环境中的渲染引擎，可以将同一套 JS 代码分别在 Android 和 iOS 系统上最终以 Native 的方式渲染页面，从而为 App 提供了基础的跨端能力。但从工程化的角度来看，如果想在 App 中大规模地应用 RN 技术，除了 RN 框架本身外，还需要在开发、构建、测试、部署、运维等诸多方面的配合。MRN（Meituan React Native）是美团基于 React Native 框架改造并完善而成的一套动态化方案，在 RN 的基础上提供了容器化能力、动态化能力、多端复用能力和工程化保障。MRN 在开发效率、稳定性、性能体验、动态化和监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。

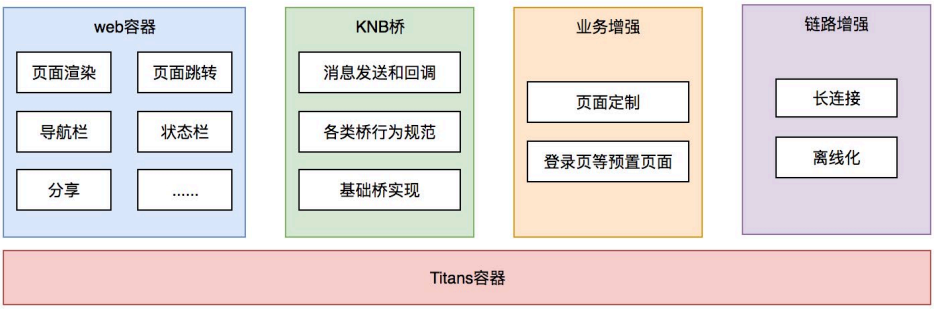


外卖在 2018 年底开始试验 MRN 容器在外卖业务上的应用，并在 2019 年上半年进行了大面积的页面落地。目前，外卖已有近 60 个 RN 页面上线，占外卖页面比例超 80%，其中包括 Tab 页面“我的”、提单选择红包页、订单评价页等高 PV 页面。MRN 容器的接入，给外卖 App 的容器化、动态化、人效提升、包大小瘦身等方面都做出了不小的贡献。

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。

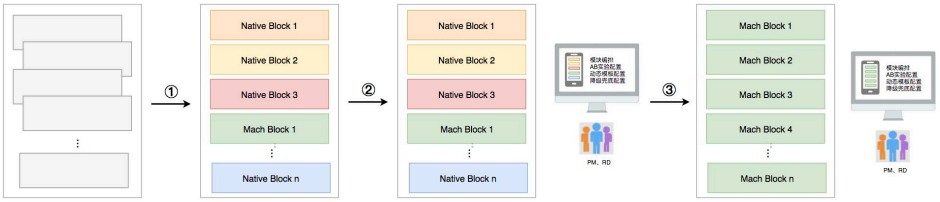
Titans 容器在外卖业务中的使用场景非常丰富，其中最重要的使用场景是各种运营页和活动页，例如点击首页顶部 Banner 的广告落地页、为你优选、限时秒杀等活动运营页等；还有客服页、帮助反馈页、商家入驻页、美团公益页等功能性页面；作为一级入口页面的美团会员页面，也是一个基于 Enlight 的 Titans 容器。

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

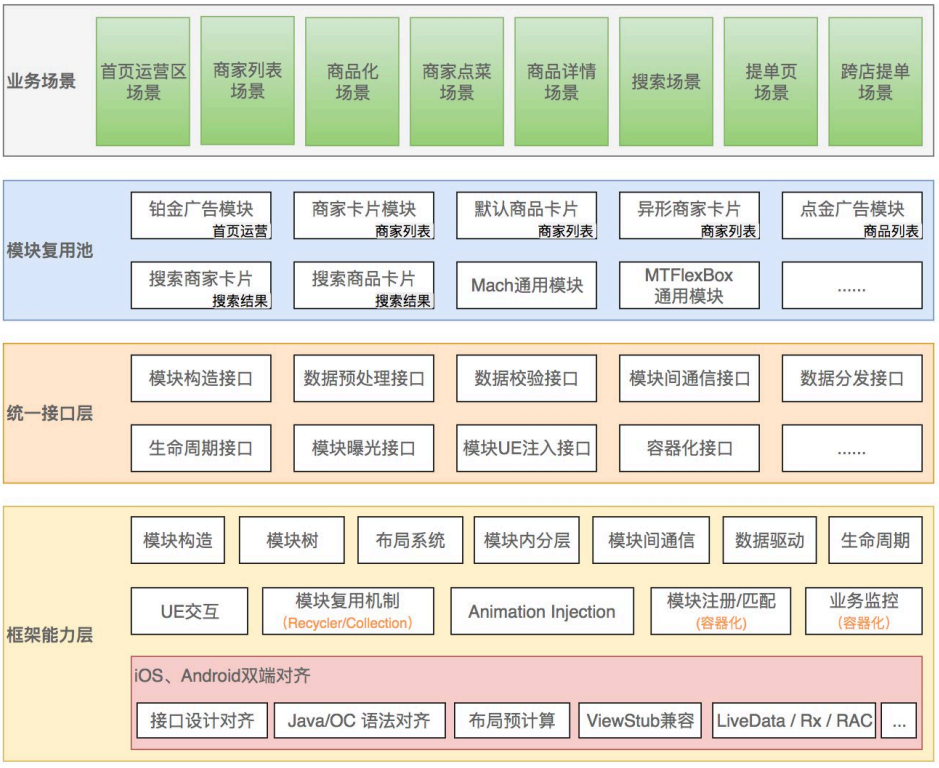
从 App 页面开发的角度看，一个完整的页面可以按照不同的功能及不同业务属性划分出多个不同的模块。

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。

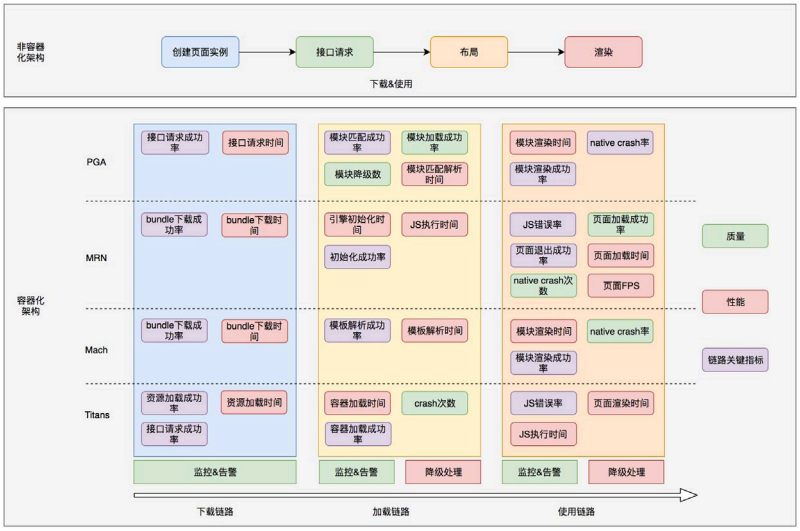
5. 外卖容器化架构的衡量指标

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

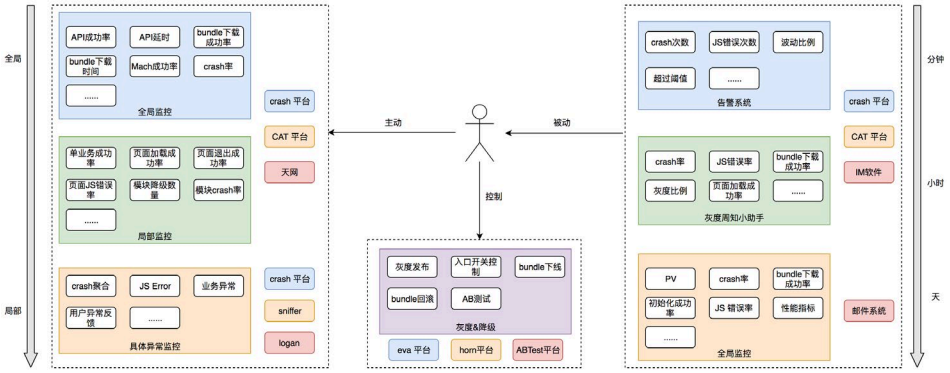
上面讲到了容器化架构的各项衡量指标，那么把这些指标具体落到实处的工作就是线上的运维监控工作。工欲善其事，必先利其器，对于监控运维工作，一定要有合适的监控工具辅助配合才能事半功倍，公司内有很多优秀的监控统计工具可供使用，这里的难点就是如何根据监控的需要判断选择合适的工具。还有就是合理的划分监控维度和数据指标的优先级，例如对于能够影响到链路稳定性的关键指标，我们需要做到分钟级的监控，一旦出现问题就能及时收到告警，对于非关键指标，则通过生成日报的方式，方便开发者的统计和分析。

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。

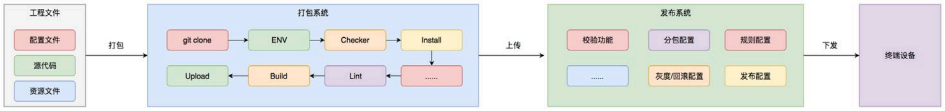


7.3 bundle 资源发布

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。

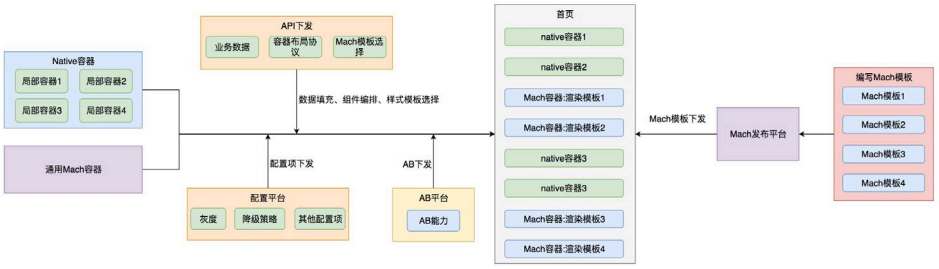
未来，我们还将引入美团住宿的 MRN-DevOps 来进一步的屏蔽当前多系统的问题，降低整个周期管理的成本，特别是发布前的人工检查成本，逐渐实现 RD 在一个平台上操作从研发到发布运维的所有实现。尽可能地减少人工成本，提升自动化。



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

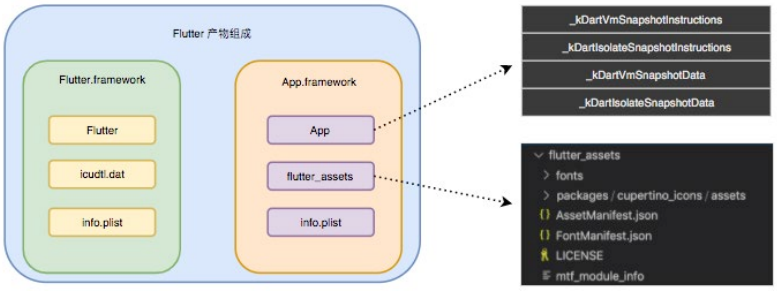


图 1　Flutter iOS 产物组成示意图

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

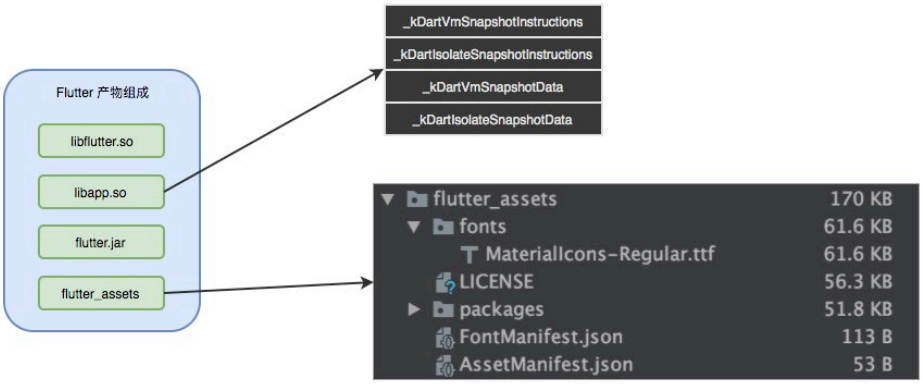


图 2　Flutter Android 产物组成示意图

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：

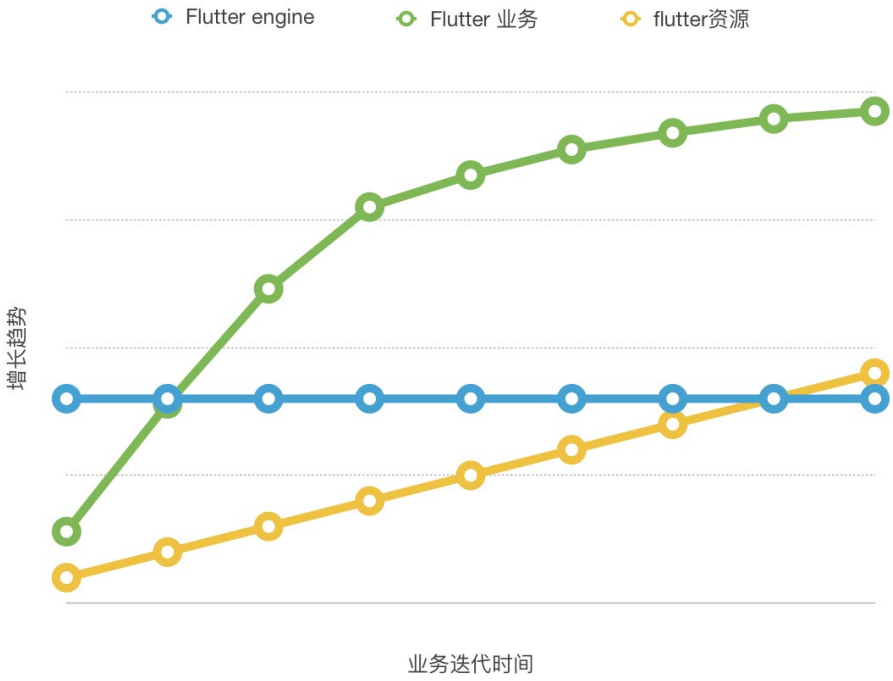


图 3　Flutter 各资源大小变化的趋势图

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。

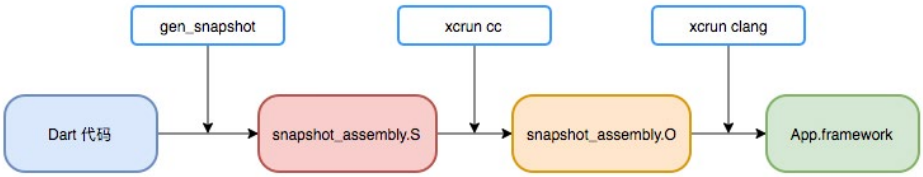


图 4　App.framework 生成流程示意图

产物加载流程：

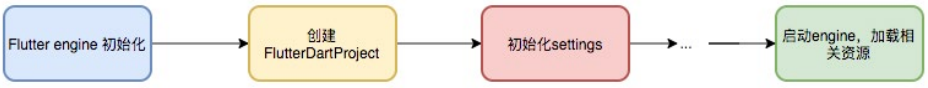


图 5　Flutter 产物加载流程图

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：

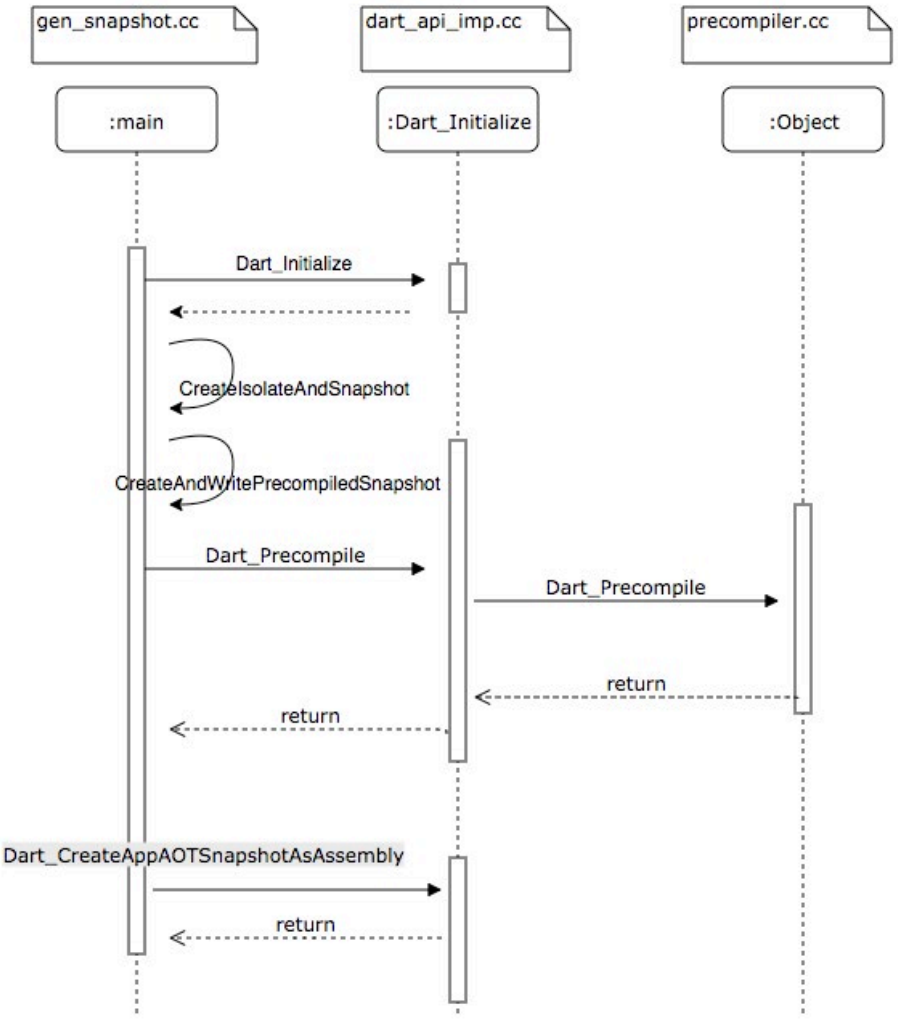


图 6　Flutter Data 段写入时序图

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 kDartVmSnapshotData、kDartIsolateSnapshotData 与 App 的分离。此部分流程示意图如下图 7 所示：

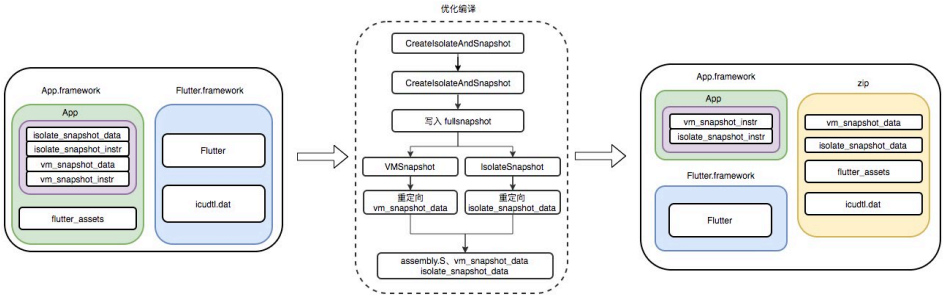


图 7　Flutter 产物拆分流程示意图

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。

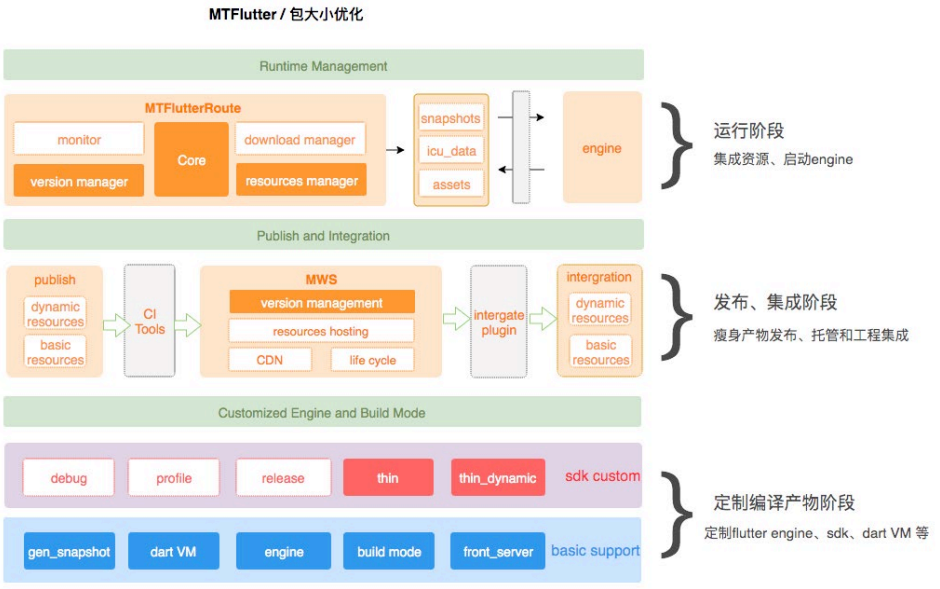


图 8　架构设计

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：

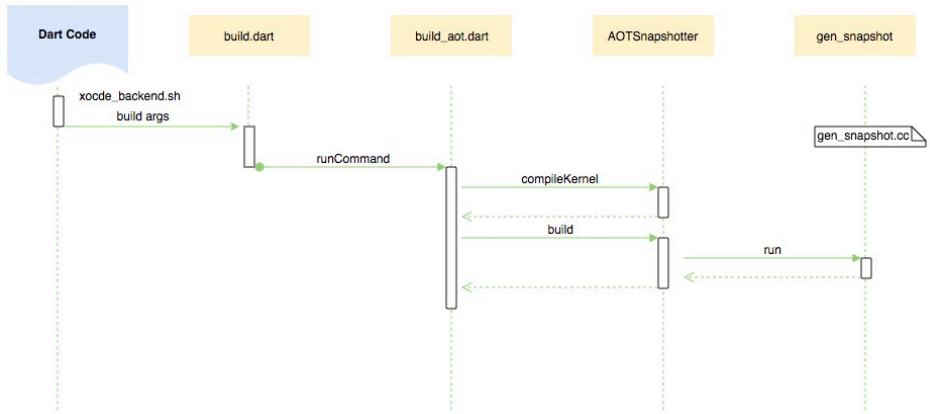


图 9　AOT 模式 Flutter 产物编译的关键工作流程图

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 -0z 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：

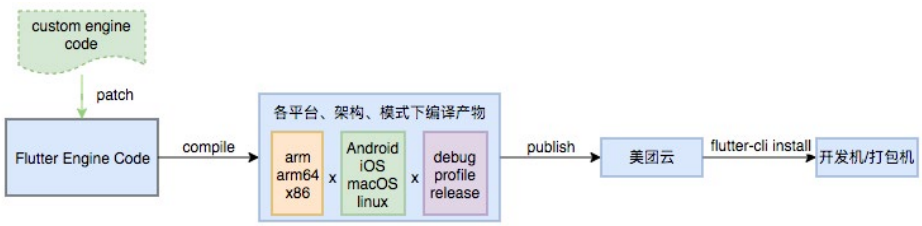


图 10　Flutter engine 自动化编译发布流程

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：

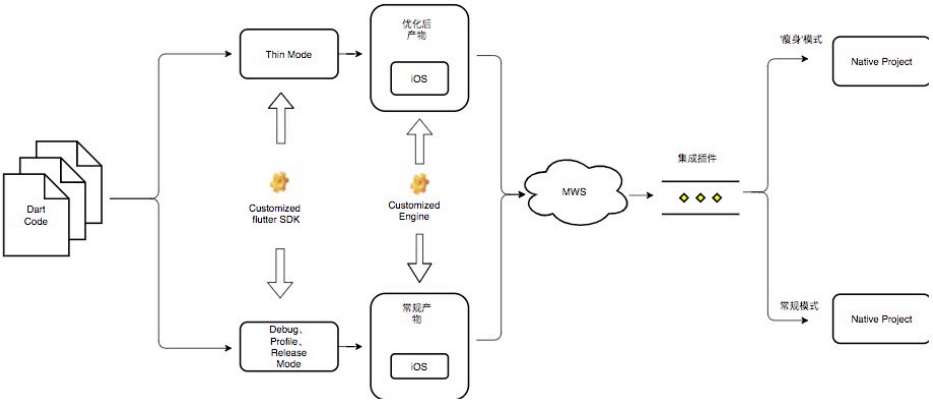


图 11　Flutter 产物发布集成流程示意图

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。

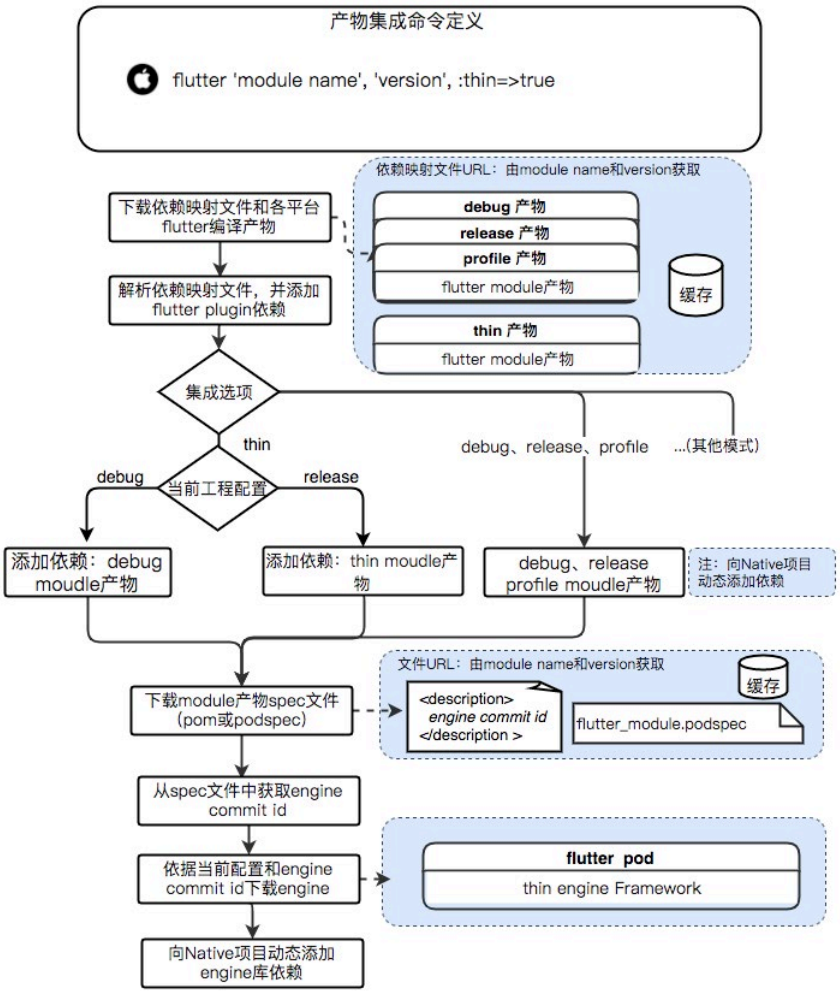


图 12　Flutter iOS 端集成插件修改

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。

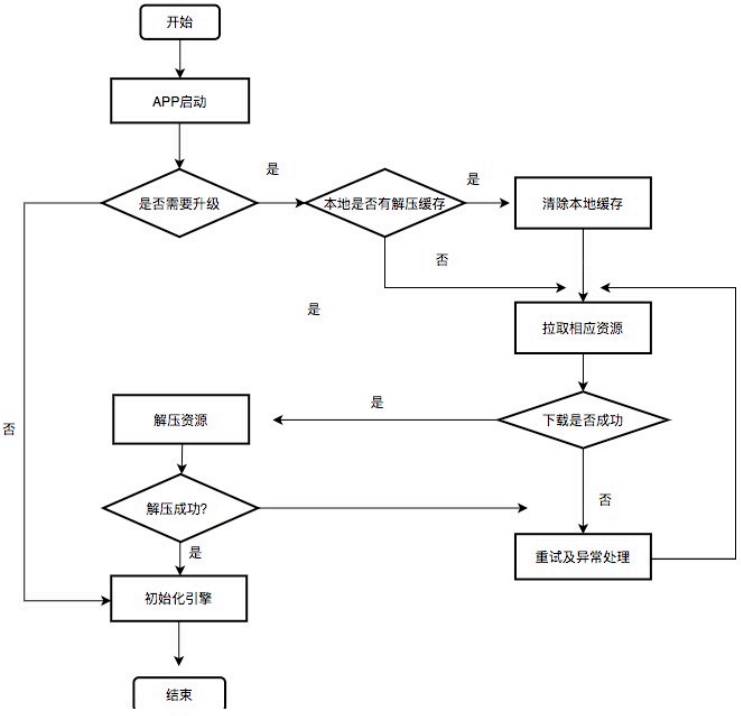


图 13　iOS 侧瘦身模式下 engine 启动流程图

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：

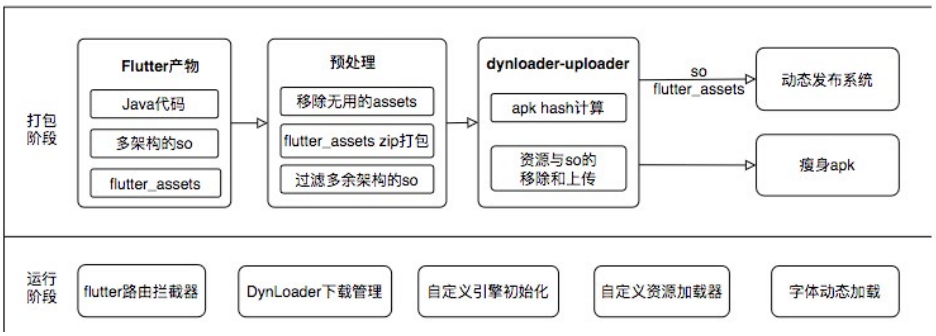


图 14　Android 侧 Flutter 包大小优化方案整体架构

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。

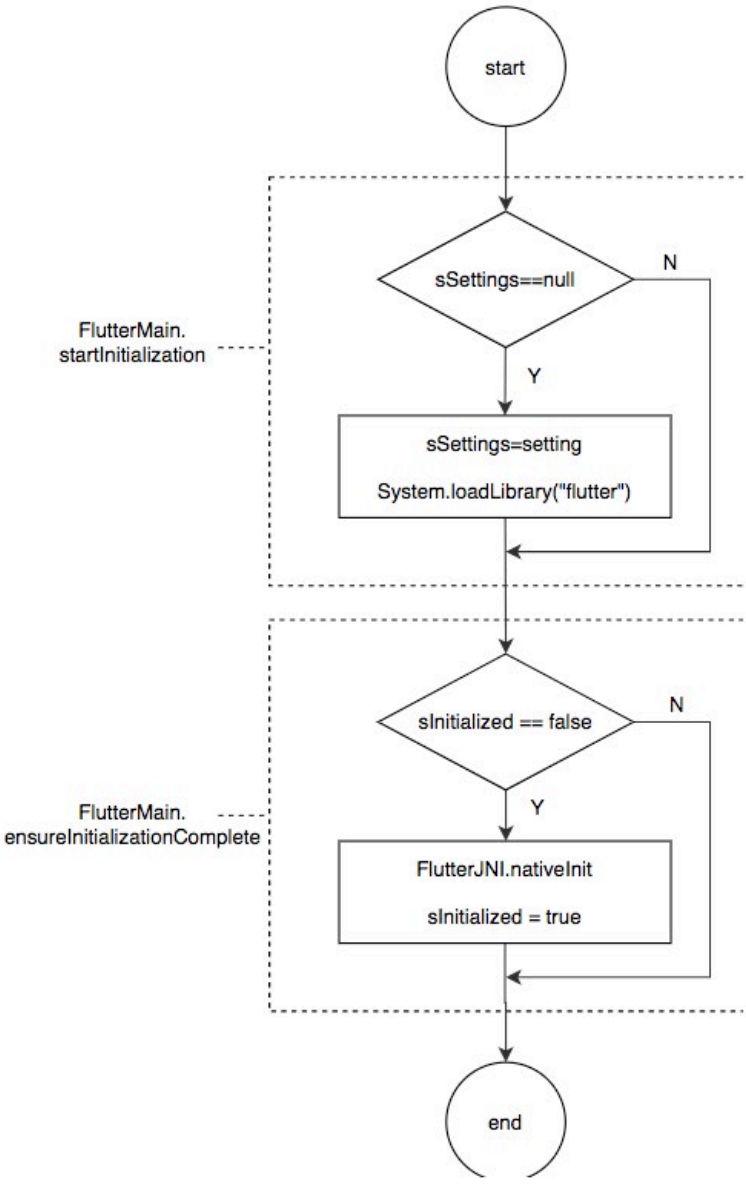


图 15　Android 侧 Flutter 引擎初始化流程图

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：

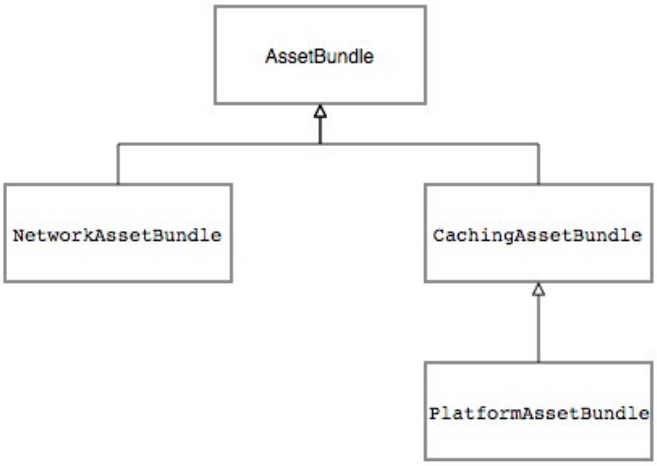


图 16　Flutter 资源加载相关类图

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。

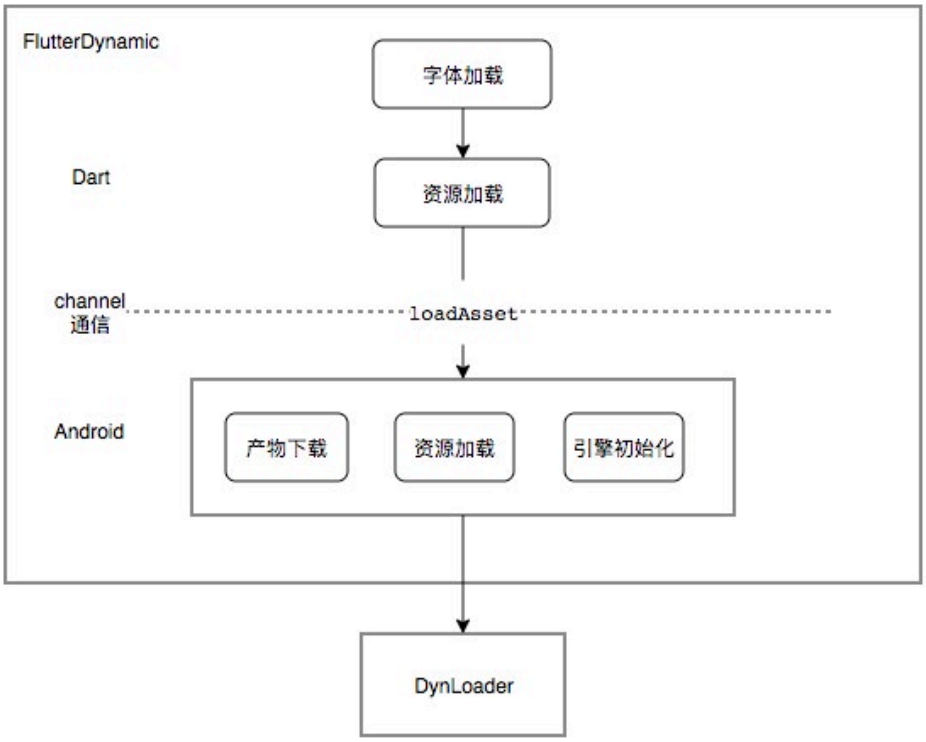


图 17　FlutterDynamic 结构图

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

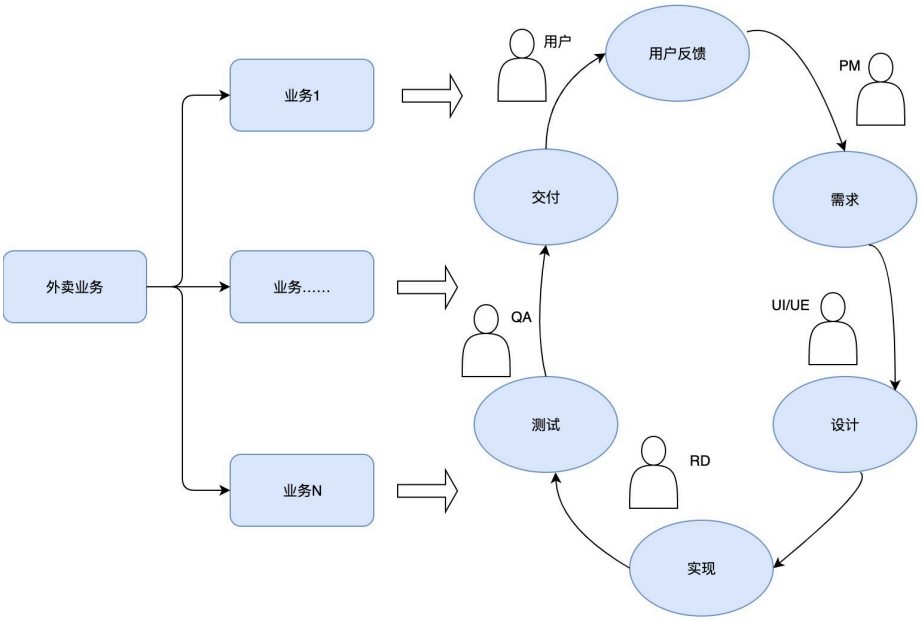
研究的话题。早在 20 世纪 50 年代，软件领域就在积极地探索设计什么样的模型可以解决这些问题。常见的包括瀑布流模型、迭代模型、螺旋模型、敏捷模型等等。由于篇幅原因，本文不再做详细的介绍。

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。

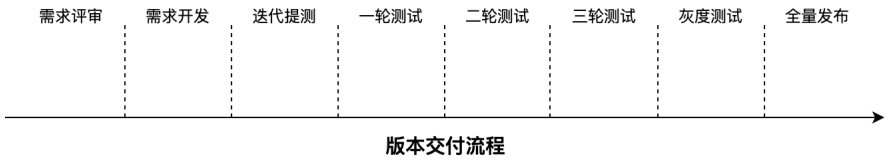
从美团外卖业务的角度，我们可以将持续交付定位为“外卖用户”和“外卖团队”之间的紧密反馈流。而外卖团队涉及到 PM、UI/UE、RD 和 QA 角色。如下图所示，产品同学从市场或用户的需求和反馈中收集到需求，转换为需求 PRD 文档，交由设计交互同学设计成期望用户所见的界面和行为，然后交给研发团队进行调研实现。研发团队实现后，再交给测试团队进行测试。等测试团队完成测试后，提交到应用市场，最终交付到用户手上，这个过程是本文所考虑的交付。“持续”指的是，外卖团队将外卖的业务拆分成不同维度的子业务，每个子业务持续通过这个迭代流程不断地优化各个子业务达到最优，从而使整个外卖业务达到最优。



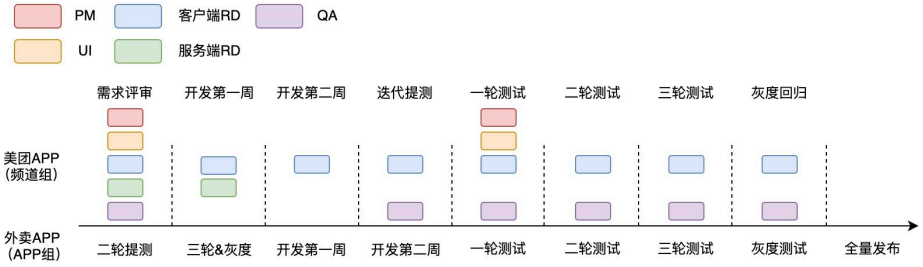
3. 外卖持续交付模型的变迁史

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。

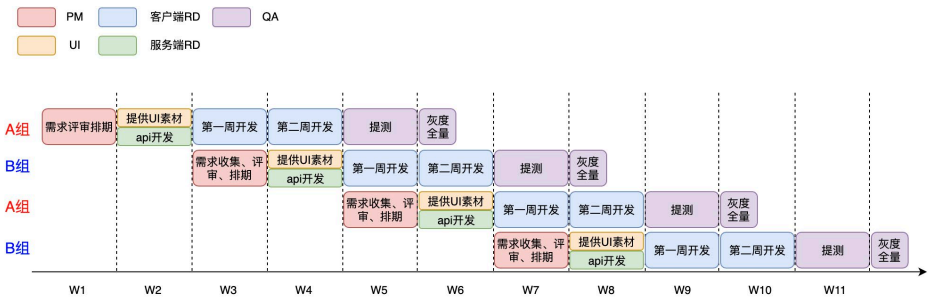
3.2 双周迭代的交付模型实践

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。

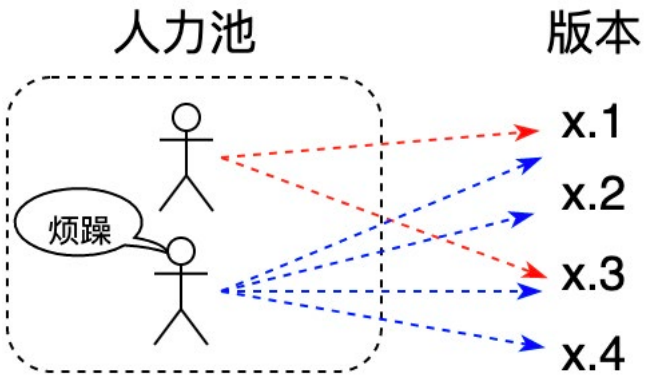
3.2.2 分组交付

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：

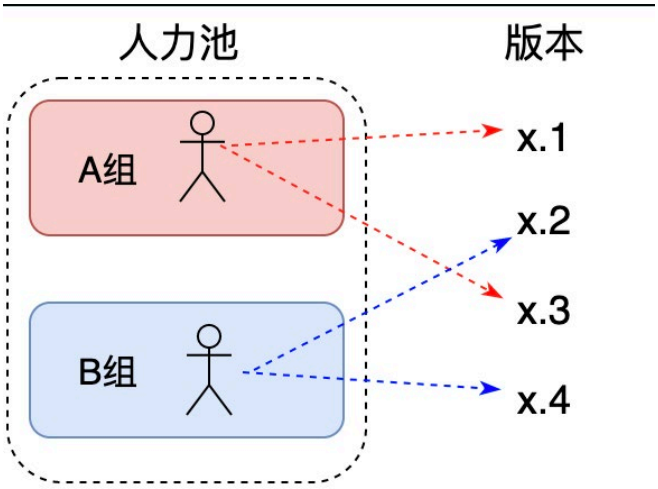


在双周迭代的过程中，客户端的交付是连贯的，理想状态是每个 RD 可以隔版进行需求开发。但是大家都知道，写完需求是不可能没有 Bug 的，需求的估时也不可能都是 5 的倍数，这样就会造成很多 RD 既要开发当前版本，又要解决上个版本的 Bug。这间接引入了 2 个问题：

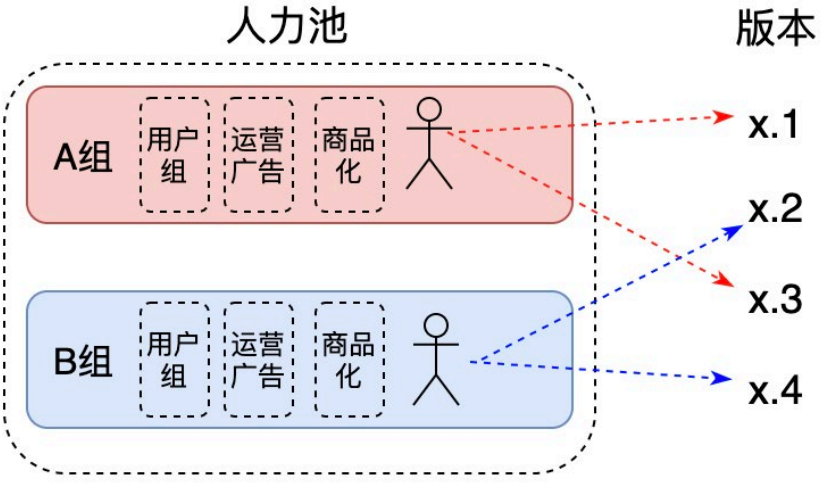
1. 个人成长方面：单个 RD 可能会在同一个时间段内，出现又要评审需求、又要开发需求、又要改 Bug，时间长了会堆积疲惫感。另外，需求的估时不是 5 的倍数，每个版本单个 RD 是否能满人力工作是不确定的，不利于资源的分配。
2. 人力分配合理性：版本的需求是存在不确定性的，而 App 的需求熟悉度是存在和 RD 一一对应的，当熟悉度最高的 RD 在上个版本的时候，当前版本的版本主 R 就会很头疼，是“能者多劳”还是“合理分配”，给人力分配的合理性又带来了新的挑战。



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。

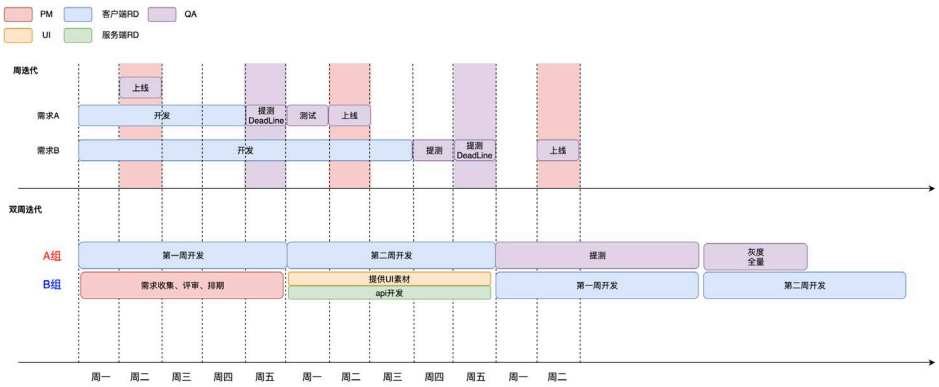
3.3.2 双周迭代 & 周迭代

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：

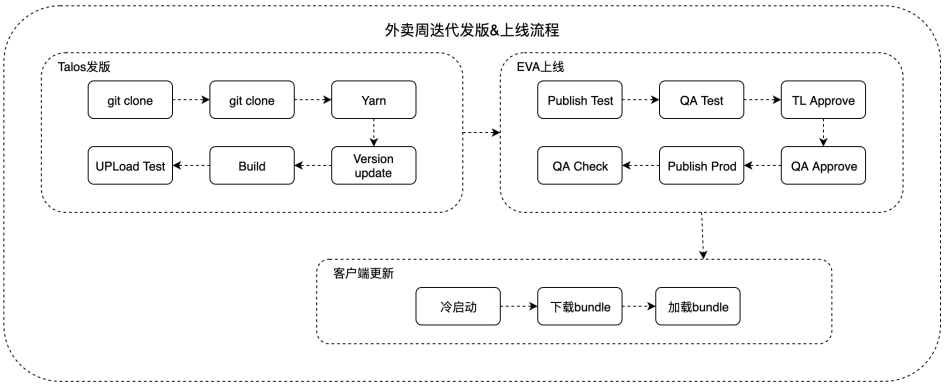
1. 纯动态化需求，根据需求开发及测试排期选择上线窗口，搭最近一次的上线窗口。
2. 动态化结合 Native 需求，跟随主版本交付流程，在提测周发版上线，主要面向敏捷迭代内容。

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

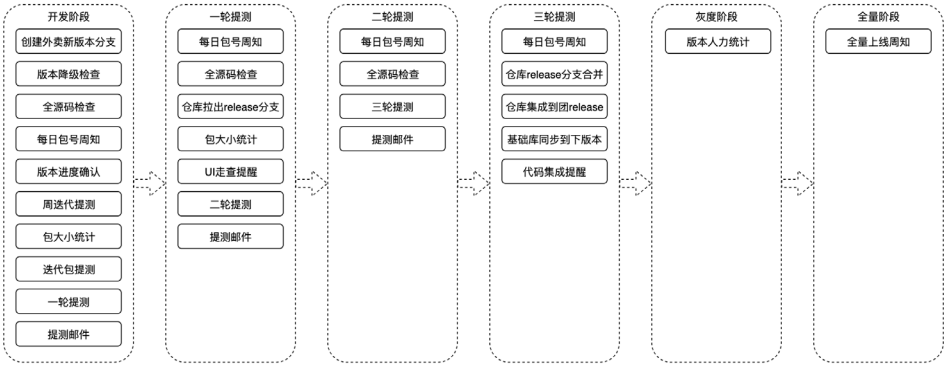
在施行了双周迭代结合周迭代后，大大缩短了需求的上线周期，但是同时也提升了版本流程管理的复杂度。怎么样保证版本流程的正常运作呢？外卖 C 端涉及开发人员分为 A、B 组及敏捷开发人力；同时涉及美团频道及外卖 App 双平台，需要考虑双平台的版本迭代周期；涉及阶段多，各阶段都需要人工操作，执行事件多，双端执行事件总计 40+；涉及双版本并行开发，仓库代码管理复杂。

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。

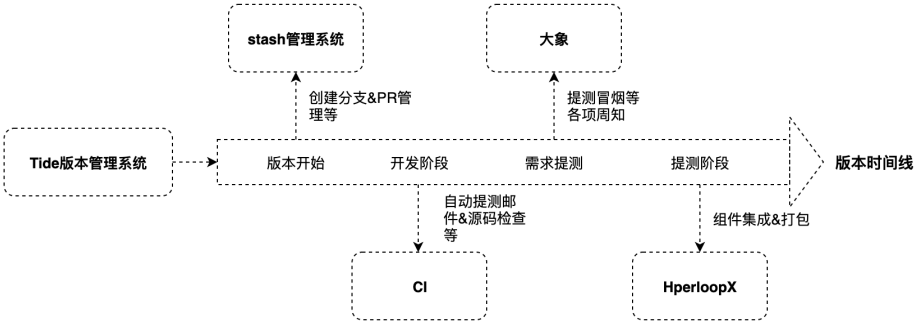
早期的版本管理，我们依托于 Gulf（美团内部日期管理系统）、QA-Assistant 系统（美团内部事件通知系统）、CI 及大象（美团内部通讯工具）消息，通过在 Gulf 系统提前配置好各版本各事件的时间，并通过 QA-Assistant 配置人员名单及消息，在固定时间节点，Gulf 去触发 QA-Assistant，通过上述方式，做到了各类提醒的自动化，并且在 CI 上部署上自动化 Job，由人工触发。

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。

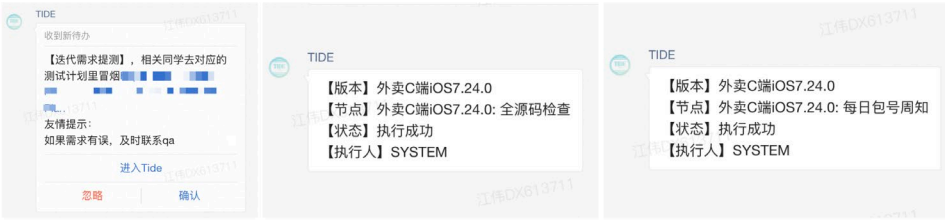
在版本开始前配置好当前版本，后续则会在版本的每个关键节点自动执行需要执行的事件。



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：

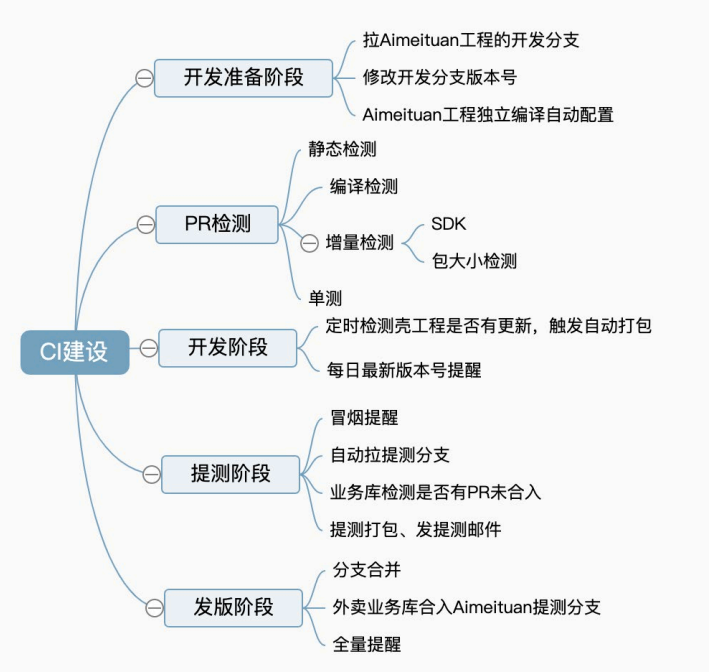


各类提醒确认结果即时展示：



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。



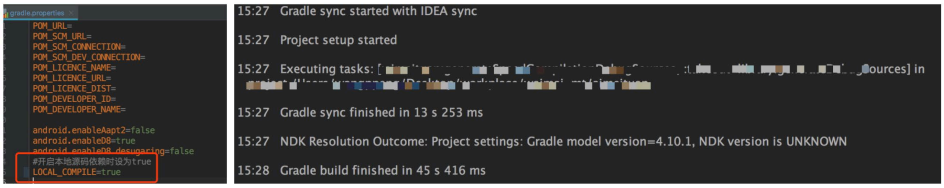
下面介绍三个 CI 工作中，外卖场景下的特定检查：

- 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。

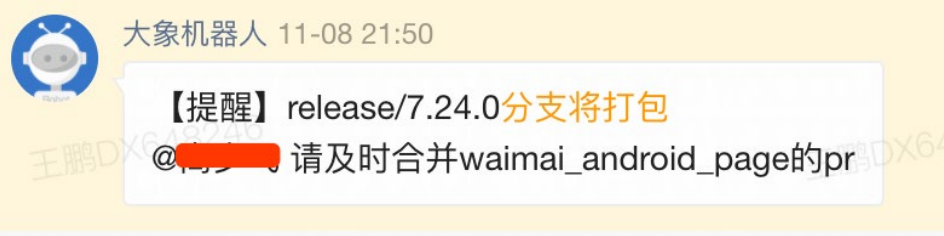
- 定时检测壳工程是否有更新，自动触发打包，保证 QA 第二天能回归前一天所有的修改，避免测试返工的问题，提高版本测试有效性。

总结

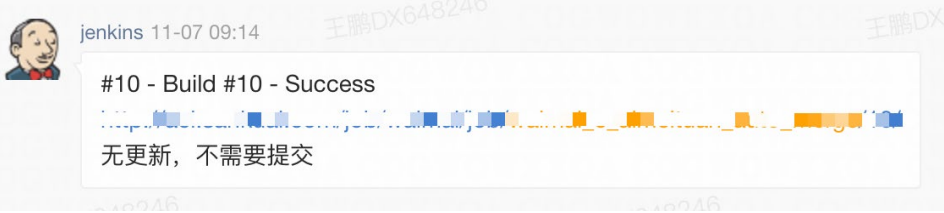
- Aimeituan 独立编译配置从 5 ~ 10 分钟（同事反馈从修改配置到同步完成的耗时）降低到 45s 左右，优化了 80% ~ 90%。



- 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。



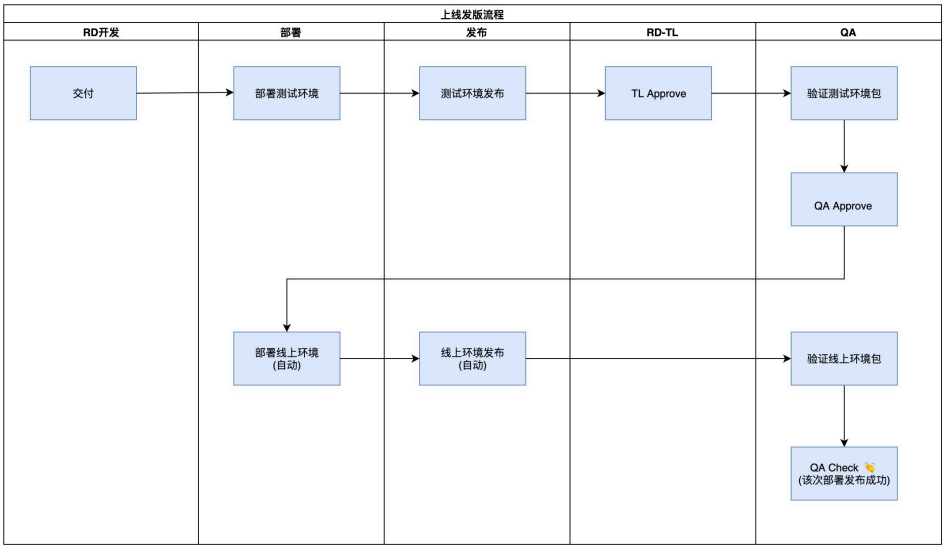


3.6 部署发布

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。

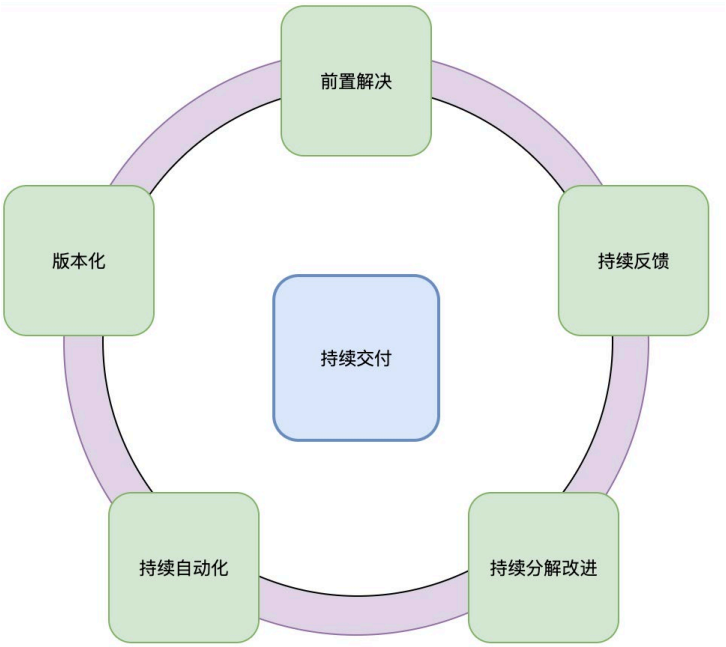
持续反馈

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

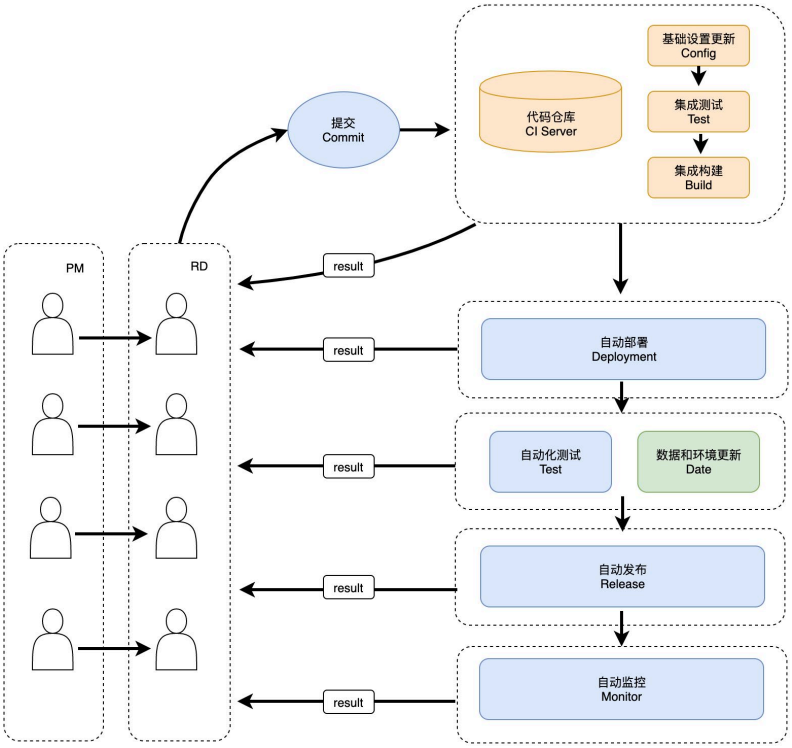
当完成代码集成后，就需要进入到测试环节了。那首先我们需要将代码部署到测试环境，因为外卖主要提供 App 供测试包，所以我们的部署，主要就是打出 App 包来，周知 QA 测试。目前外卖的测试，分位一轮新功能测试、二轮全回归测试、三轮主流程回归测试。新功能因为新引入，可能存在前置条件，而在本次测试的过程中，存在需要编写自动化用例和人工测试的用例。但对于老功能，我们希望未来能够尽可能地采用自动化为主的方式。

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

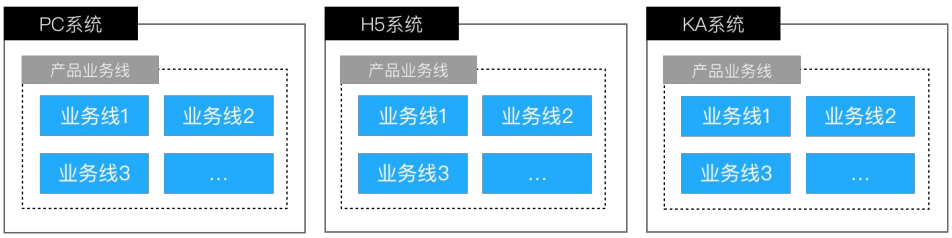
晓飞，2015 年加入美团，美团外卖团队技术专家。
王鹏，2017 年加入美团，美团外卖团队高级工程师。
江伟，2018 年加入美团，美团外卖团队高级工程师。

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

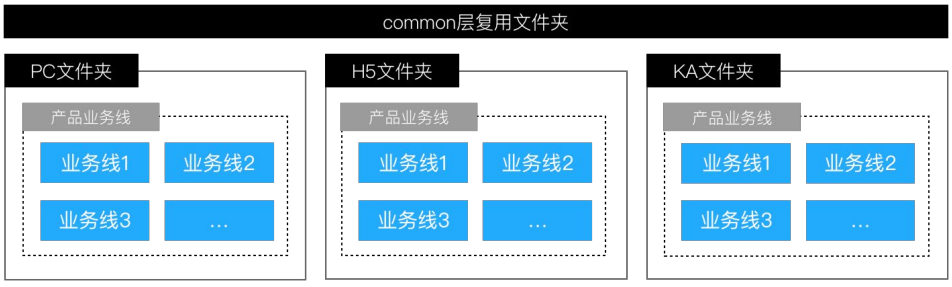
如上图所示，原始解决方案的三端由各自独立开发和维护，各自包含所有的业务线，而我们的业务开发情况是：

- 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：

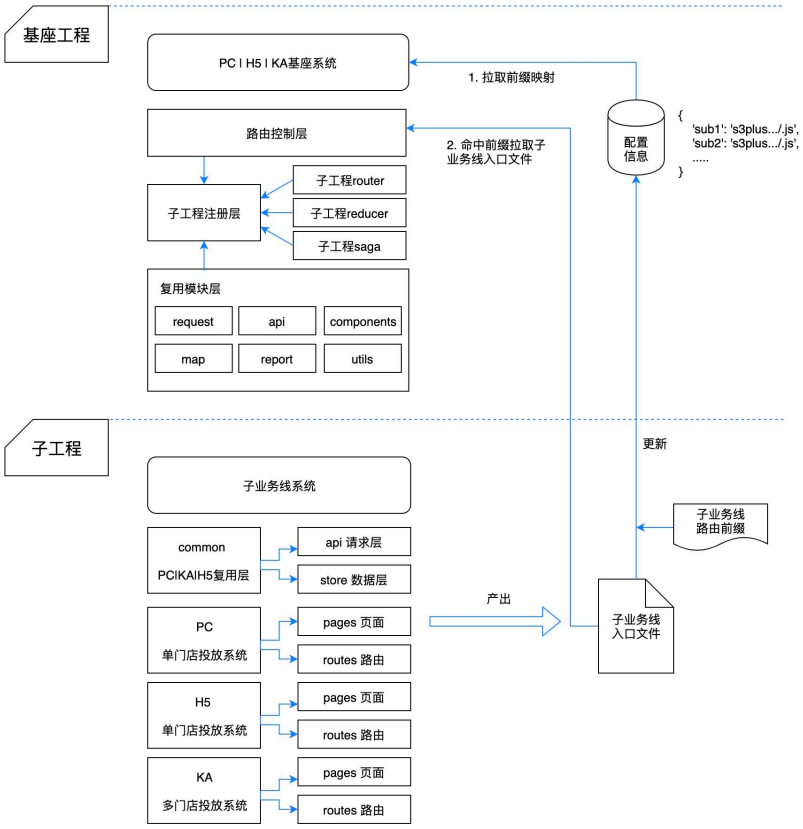
方案	技术栈是否能统一	单独打包	单独部署	打包部署速度	单页应用体验	子工程切换速度	工程间通信难度	现有工程侵入性	学习成本
NPM 式	是（不强制）	否	否	慢	是	快	正常	高	高
iframe 式	是（不强制）	是	是	正常	否	慢	高	高	低
通用中心路由基座式	是（不强制）	是	是	正常	是	慢	高	高	高
特定中心路由基座式	是（强制）	是	是	快	是	快	正常	低	低

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：



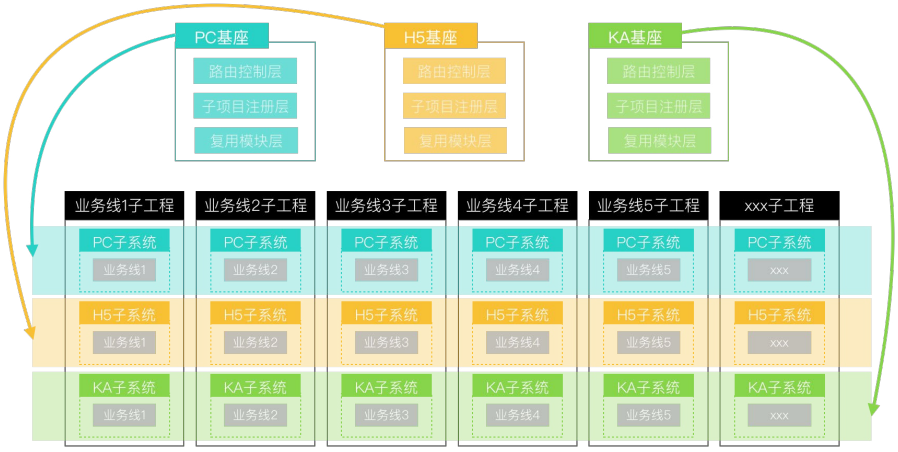
可以看到，整个方案非常简单明确，即按照业务线进行了**路由级别的拆分**。整个系统可分为两个部分：

- 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

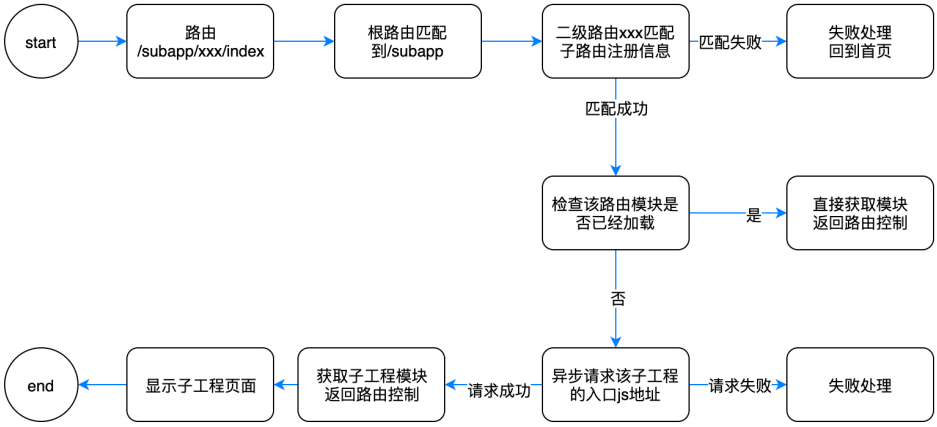
首先，我们需要**路由的管理方案**，使得子工程之间有能力互通切换。其次，我们需要 **Store 层的方案**，让子工程有能力使用全局 Store。并且，我们还需要 **CSS 的加载方案**，来加载子工程的样式布局。下面来详细说明这三个方案。

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

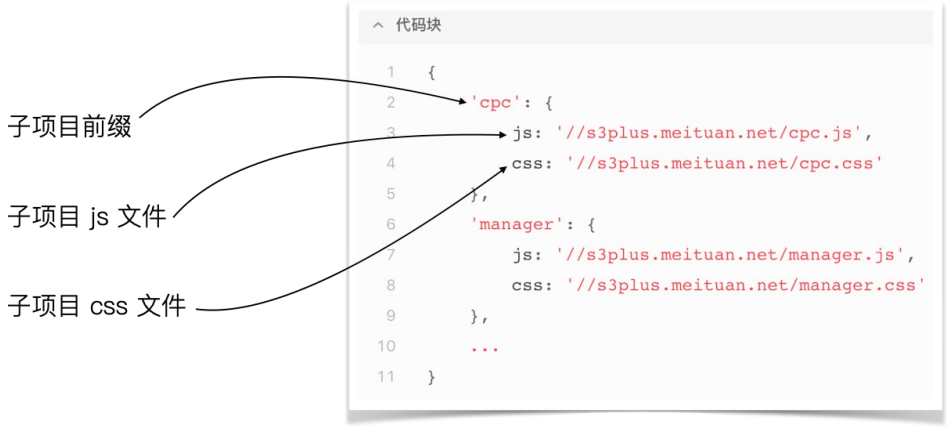
同样的对应子工程的样式布局，我们也需要通过某种途径加载到基座工程中来。这个很自然地用异步加载 CSS 文件通过 style 标签注入来完成，不过这里需要注意两个问题：

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;
```

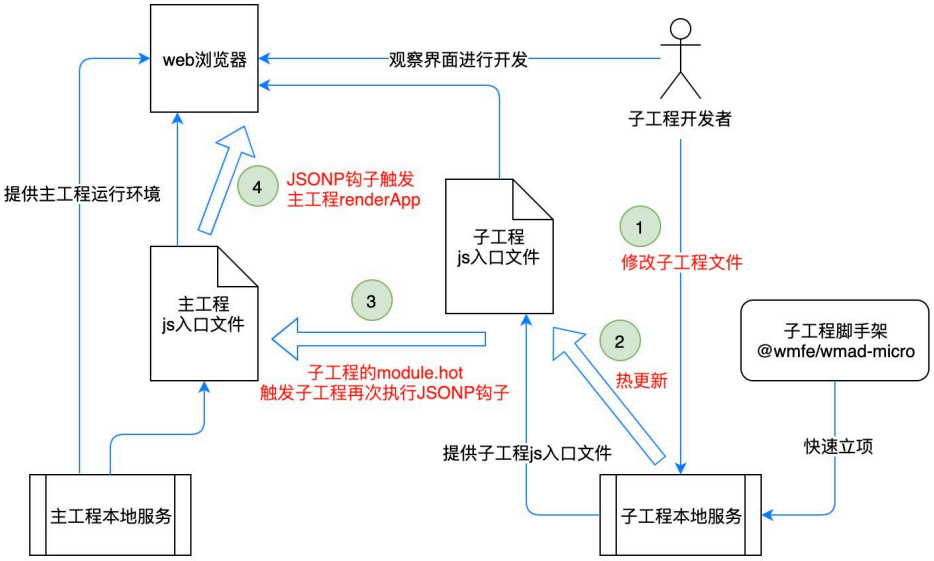
流程方案

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：

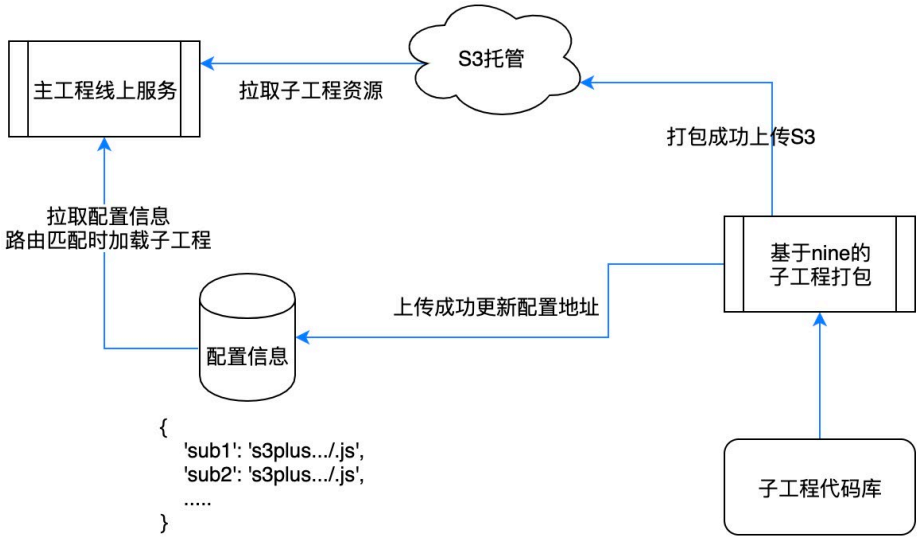
```
// 在子工程入口文件
import routes from './routes/index';
function registerApp(dep: any = {}): any {
    return {
        routes,
    };
}
if ((module as any).hot) {
    (module as any).hot.accept('./routes/index', (): any => {
        window.wmadSubapp(registerApp, true); // 支持子工程热加载的信息传递
    });
}
export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



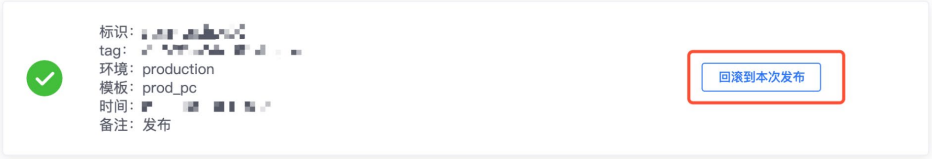
整个部署过程我们是托管到 Talos（美团内部自研的部署工具）上的，配置信息我们是托管到 Portm（美团内部自研的文件存储）上的（通过我们开发的 Talos 的插件

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

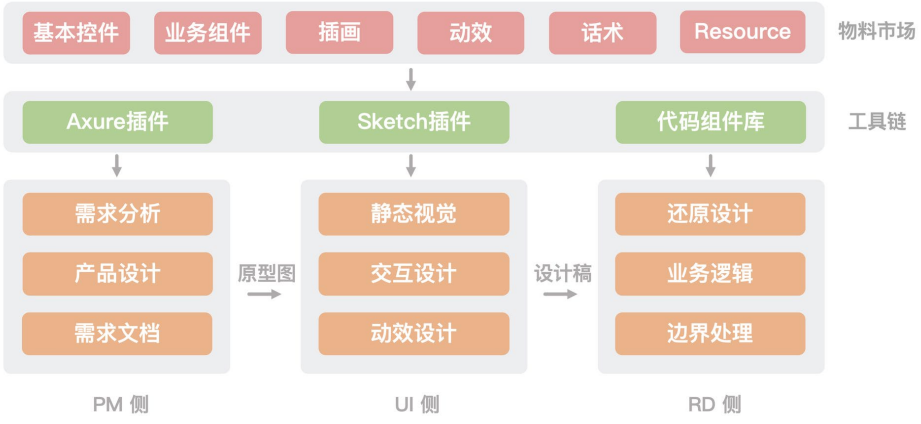
前段时间我们在美团技术团队公众号上发表了《积木 Sketch Plugin：设计同学的贴心搭档》一文，不曾想到，这个仅在美团外卖 C 端使用的插件受到了更多的关注，美团多个业务团队纷纷向我们抛出“橄榄枝”，表示想要接入以及并表达了愿意共同开发的意向，其它互联网同行也纷纷询问相关的技术，一时让我们有些“受宠若惊”。回想起写第一篇文章的时候，我们的内心还是有些不安的。作为 UI 同学的一个设计工具，有些 RD 甚至没有听说过 Sketch 这个名字，我们很认真地修改过上一篇文章的每一句措辞，争取让内容更丰富有趣，当时还很担心不会被读者接受。

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。



工具栏业务切换　　选择业务方　　变更物料市场内容

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')
);

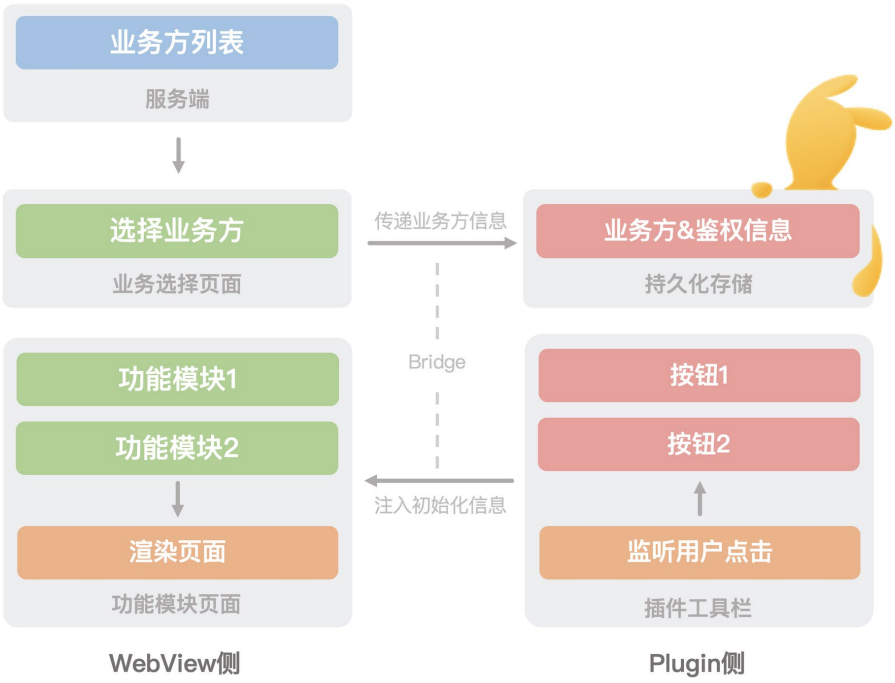
window.initBusinessData = data => {
  const businessItem = {};
  Object.keys(data).forEach(key => {
    businessItem[key] = { $set: initialData[key] };
  });
  store.dispatch(update(businessItem));
};

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。



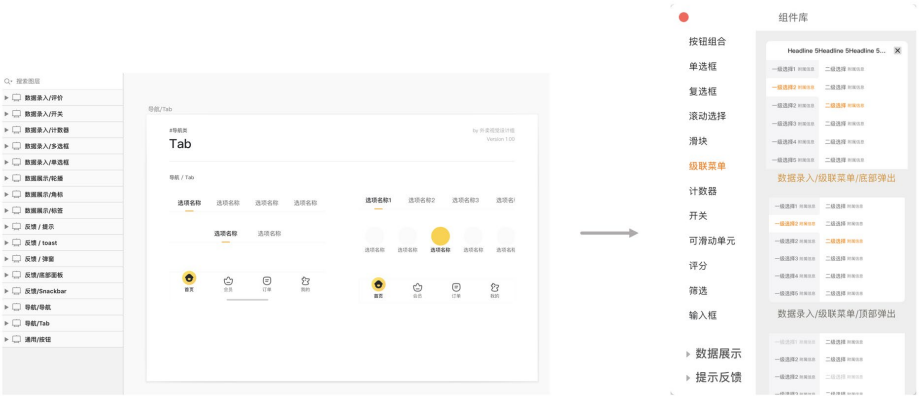
Library 库文件自动化处理

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。

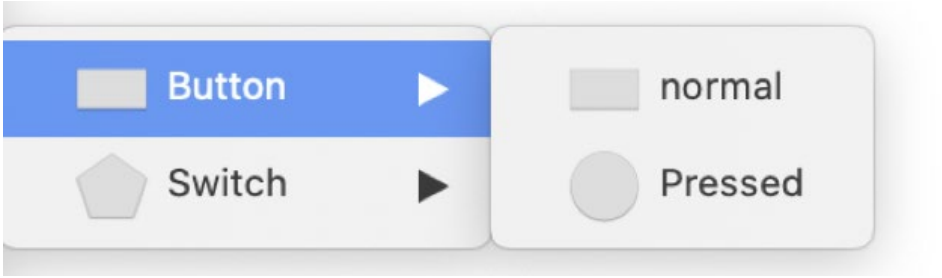
订阅云端组件库的方式很简单，首先创建一个云端组件库，具体可以参照上一篇文章，如果需要服务多个设计部门，则需要创建多个库，每个库有唯一的 RSS 地址；在插件中获取到这些 RSS 地址后，可以通过 Library.getRemoteLibraryWithRSS 方法对其进行订阅。

```
// 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';

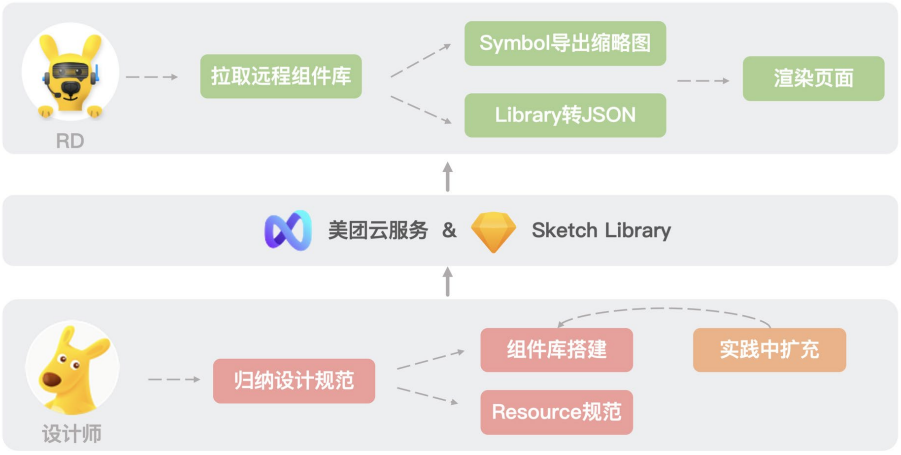
sketchDOM.export(symbolMaster, {
     overwriting: true,
     'use-id-for-name': true,
     output: path,
     scales: '1',
     formats: 'png',
     compression: 1,
});
```

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();
  }
});
```

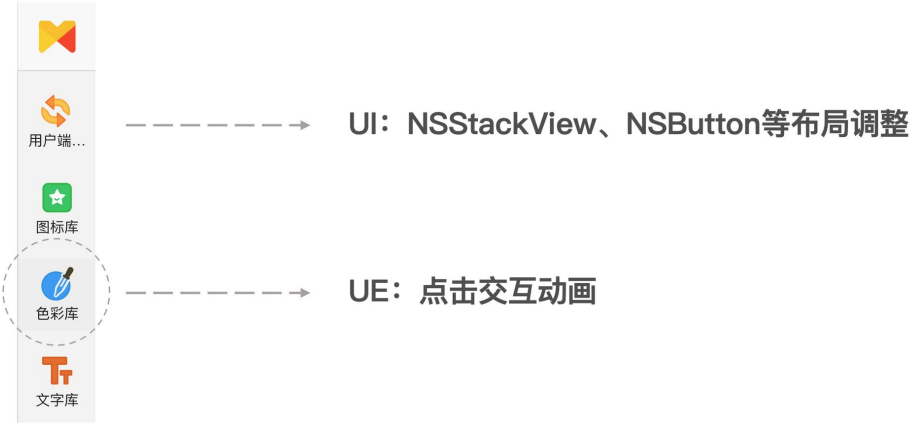
工具栏优化

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。



优化前



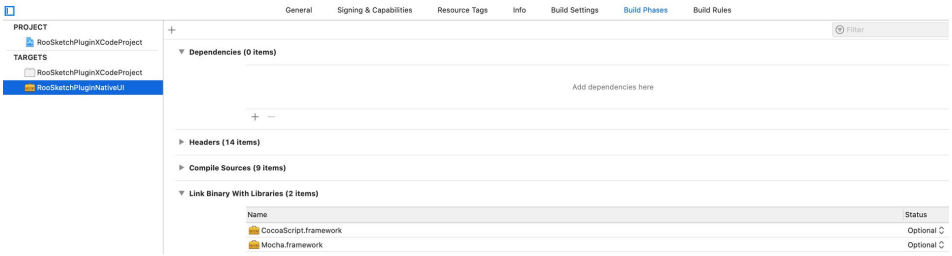
优化后

Sketch Inspector 面板操作区优化

Xcode 工程配置

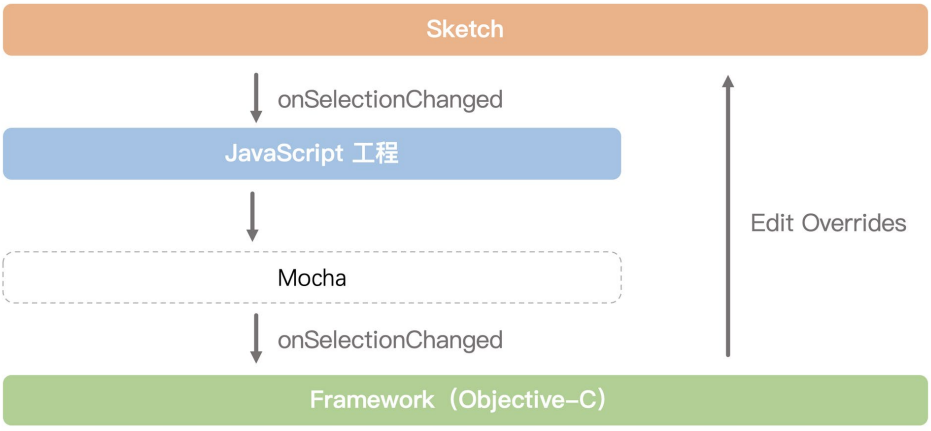
通过 Xcode 工程构建自定义属性面板，最终生成一个可以供 JS 侧调用的 Framework。可以参考上一篇文章介绍的方法创建 Xcode 工程，该工程在每次构建后会自动生成测试 Sketch 插件并放入对应的文件夹中。需要注意的一点是，这里生成的插件只是为了方便开发和调试，后面会介绍如何将 XCode 工程构建的 Framework 集

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"]];

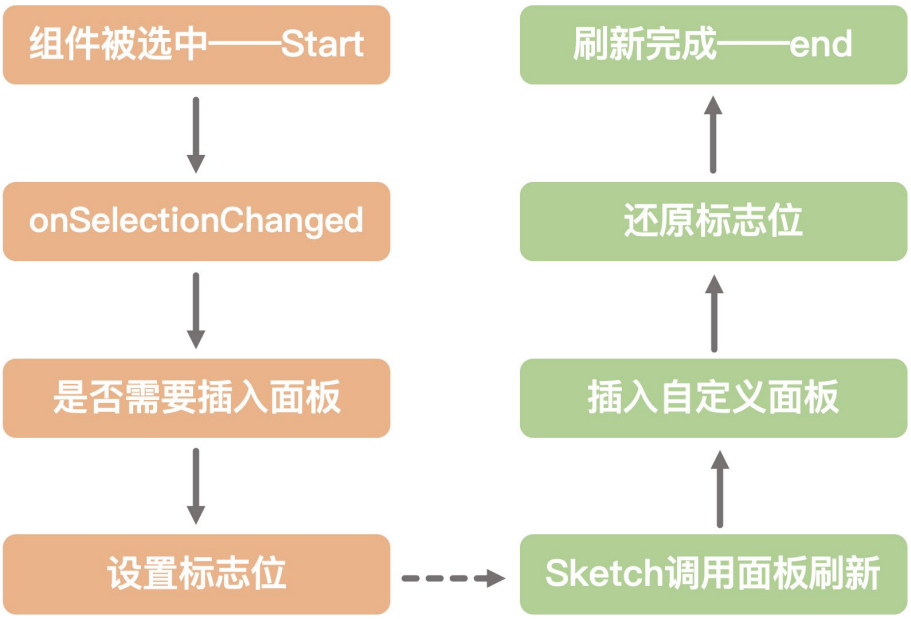
    NSString *key = [NSString stringWithFormat:@"%@-
RooSketchPluginNativeUI", [document description]];
    __block RooSketchPluginNativeUI *instance = [[Mocha sharedRuntime]
valueForKey:key];

    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。

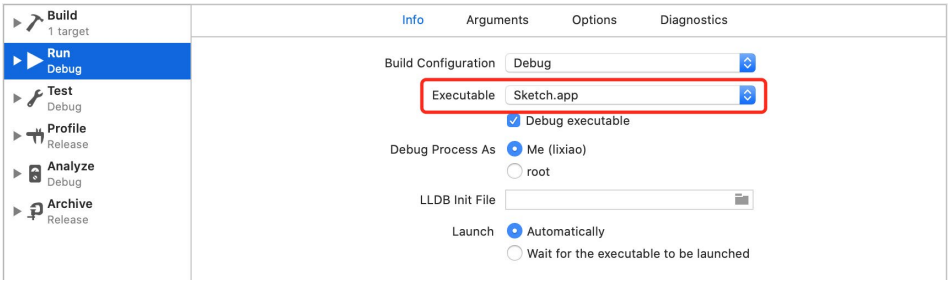
```
// 在更新图层上的组件之前，我们需要把组件导入到当前 document 对象中
id foreignSymbol = [libraryController
importShareableObjectReference:sharedObject intoDocument:documentData];

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

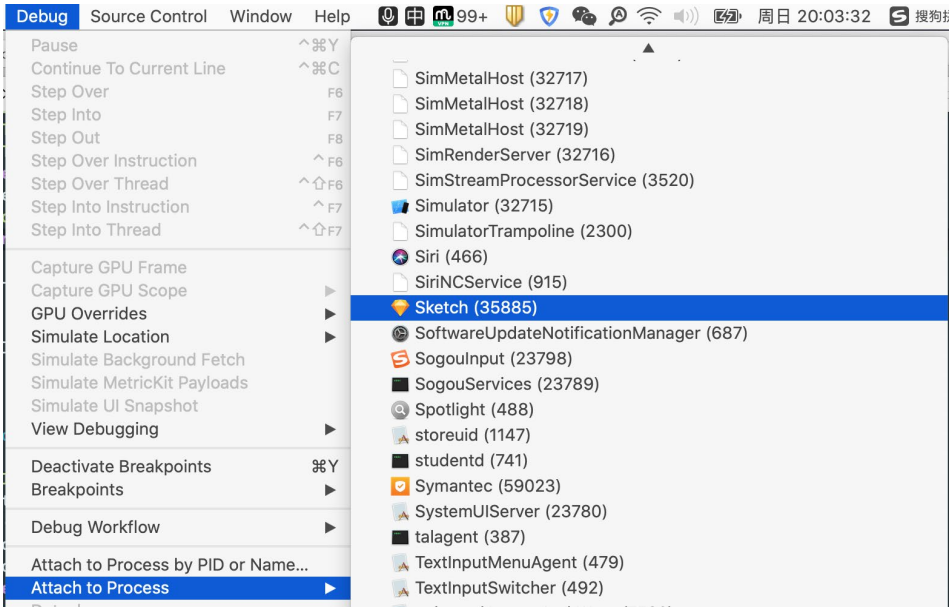
调试技巧

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。

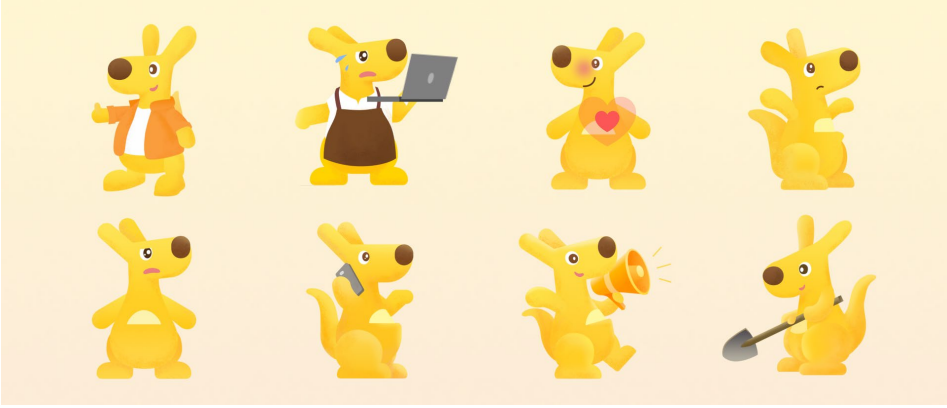
```
// 找到已经被加载的 framework
 const frameworkClass = NSClassFromString('RooSketchPluginNativeUI');
// 调用暴露的方法
frameworkClass.onSelectionChanged(context);
```

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

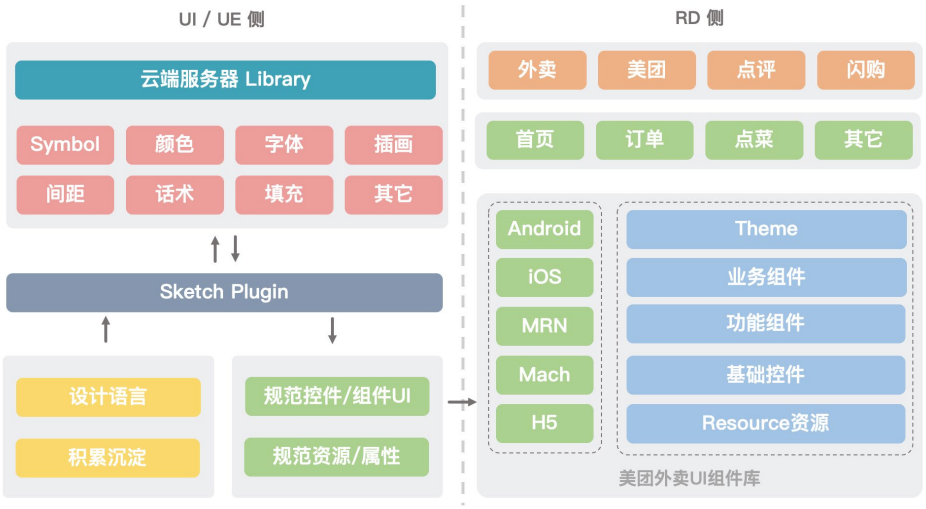
重复走查，频繁回归，每次发版均需验证组件质量。

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

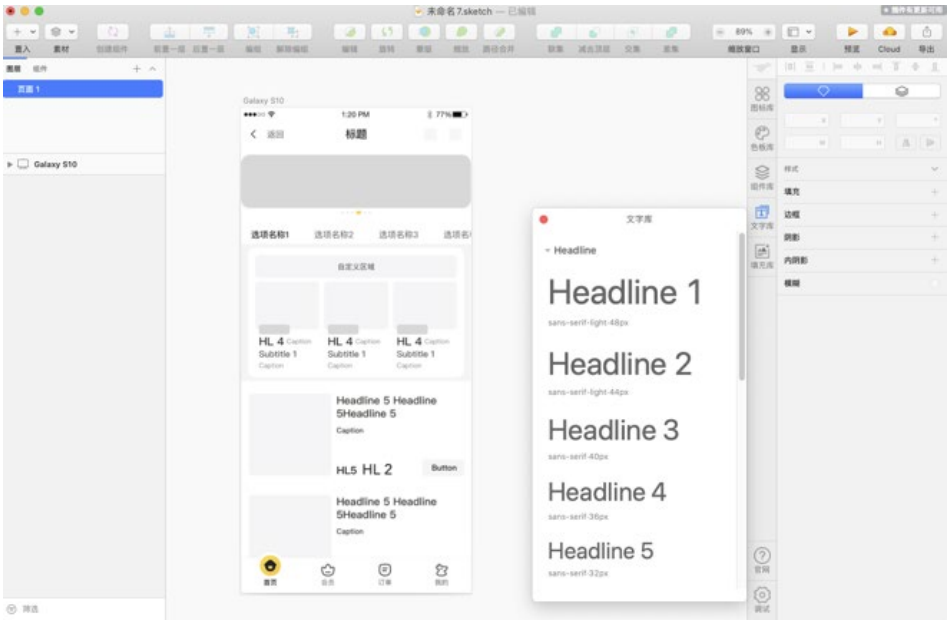
要想保持 UI 一致性，就不能打破规则。从设计阶段颜色的选择、字体的规范、控件的样式到 RD 开发阶段代码的统一管理、API 的制定、多端的实现方式，都必须遵守一套规则，而 Sketch Plugin 建设则是让规范落地执行的解决方案。

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

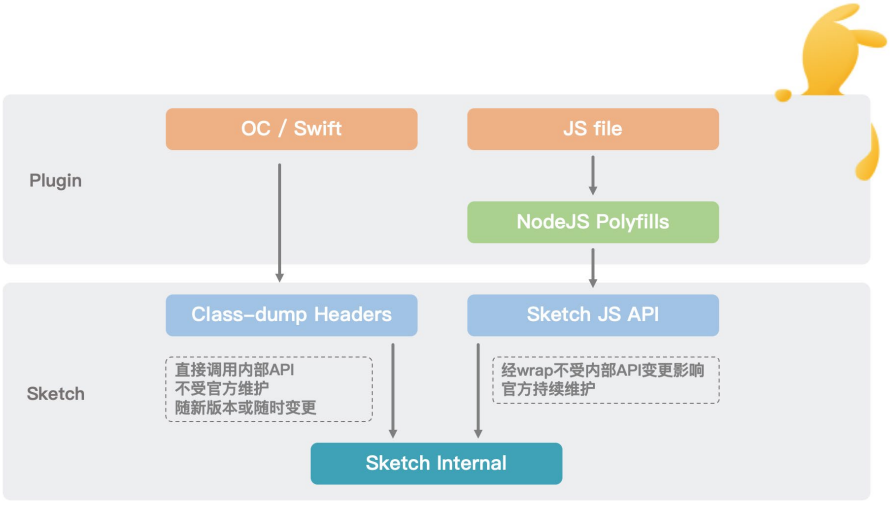
好，先别着急敲击键盘。毕竟我们连使用哪种语言去开发都没决定，这曾经也是困恼我们许久的一个问题。目前 Sketch Plugin 开发大概有两种方式：

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。

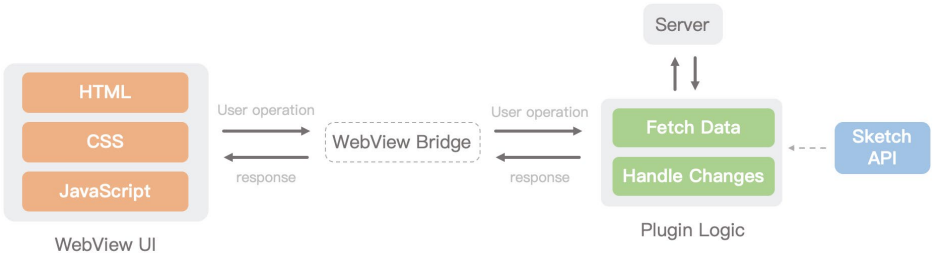
```
{
      "script": "action.js",
      "identifier": "my-action-listener-identifier",
      "handlers": {
        "actions": {
          "OpenDocument": "onOpenDocument"
        }
      }
}
```

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

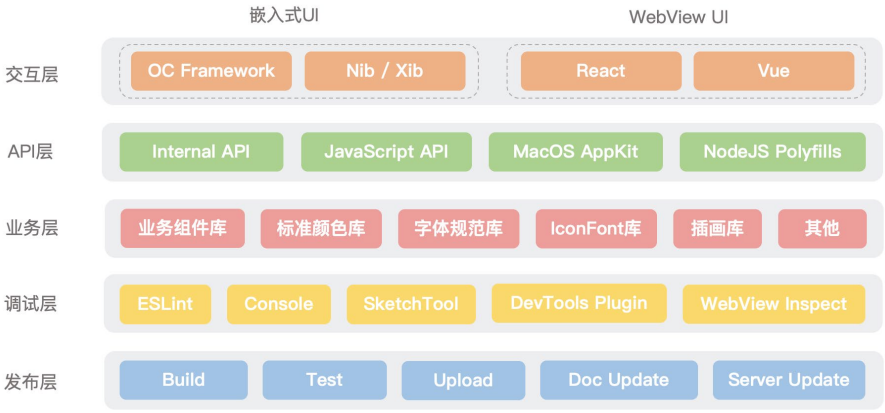
```
  browserWindow.webContents.on('fill-text-layer', function(a) {
   // 找到当前页面 document
  const document = context.document;
   // 获取用户选择的 layers
     const selection = Document.fromNative(document).selectedLayers;
        layers.forEach(item => {
          // 判断 layer 类型是否为文字
          if (item.type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

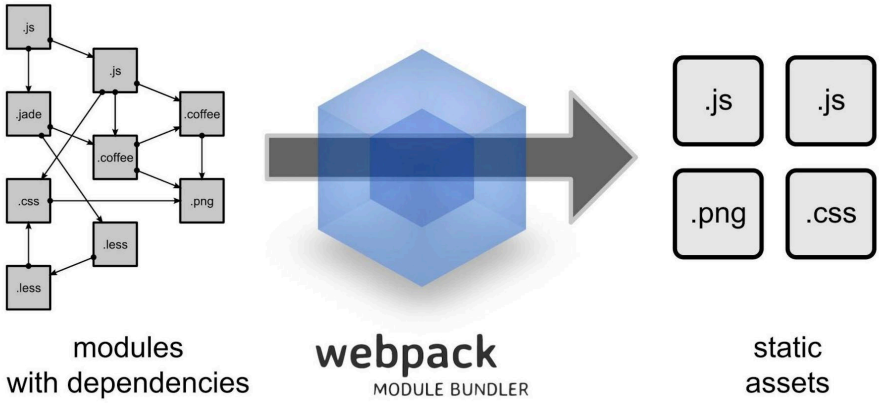
但除此之外，在代码编写过程中还需要 Lint 组件辅助开发，发现问题需要使用各类 Dev 工具进行调试，通过 QA 验证后，需要 Cli 工具打包并发布插件更新。这一小节，我们将简单介绍一些基本的调试层和发布层知识。



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

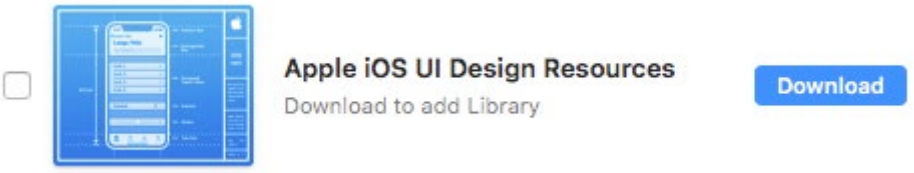
同学，可能对于 ESLint 的复杂配置并不熟悉。可以直接使用 Skpm 提供的 ESlint Config，里面配置了包含 Sketch 和 macOS 的头文件的全局变量，而代码格式化则推荐使用 Prettier。

```
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
```

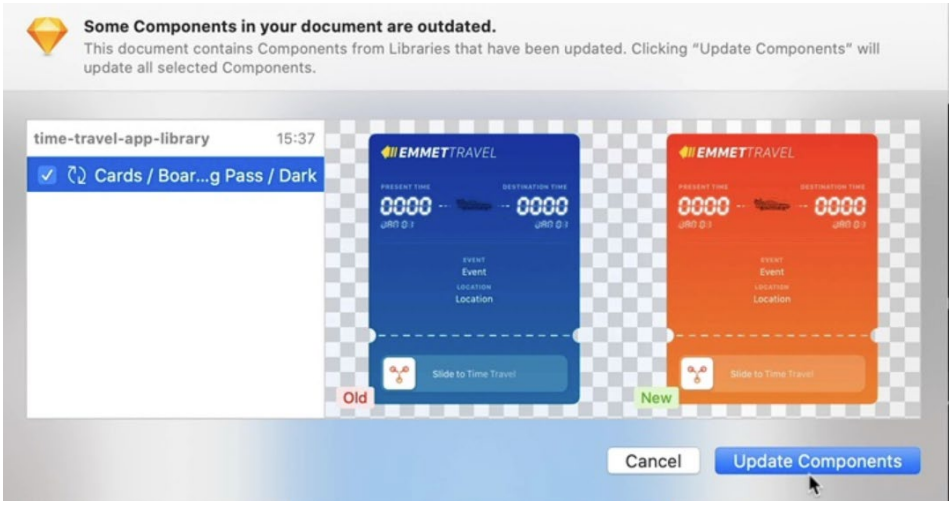
内容服务端化

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。

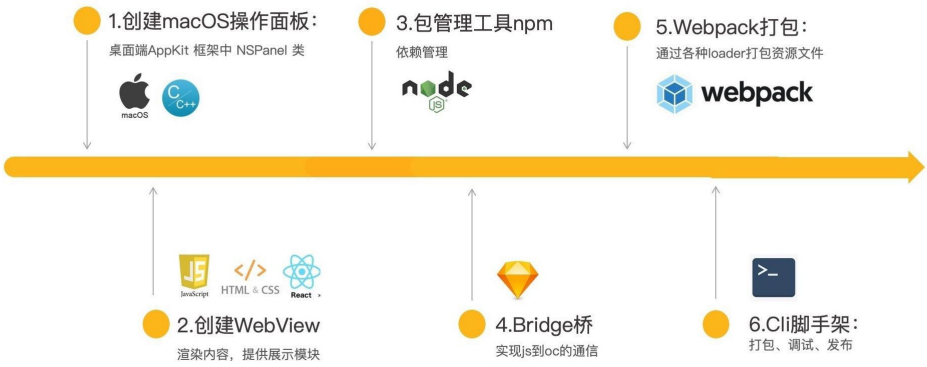
```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:content="http://purl.org/rss/1.0/modules/content/"
xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:sparkle="http://
www.andymatuschak.org/xml-namespaces/sparkle">
```

```
  <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：



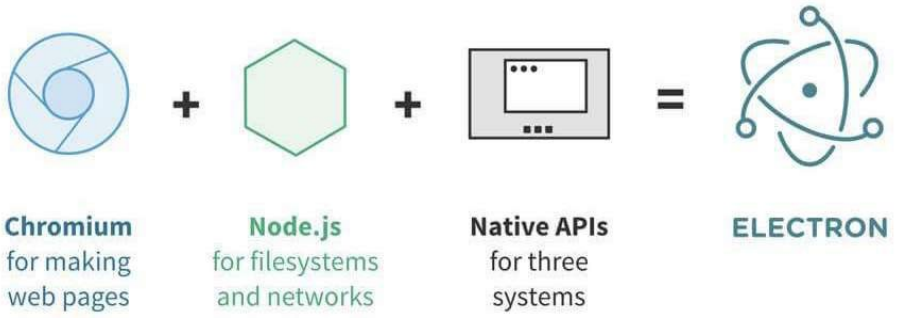
2. 了解 Electron

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）。



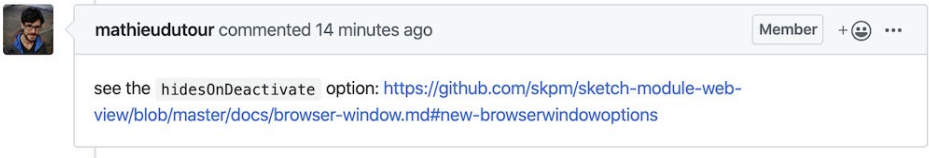
MacOS 版大象截图

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?



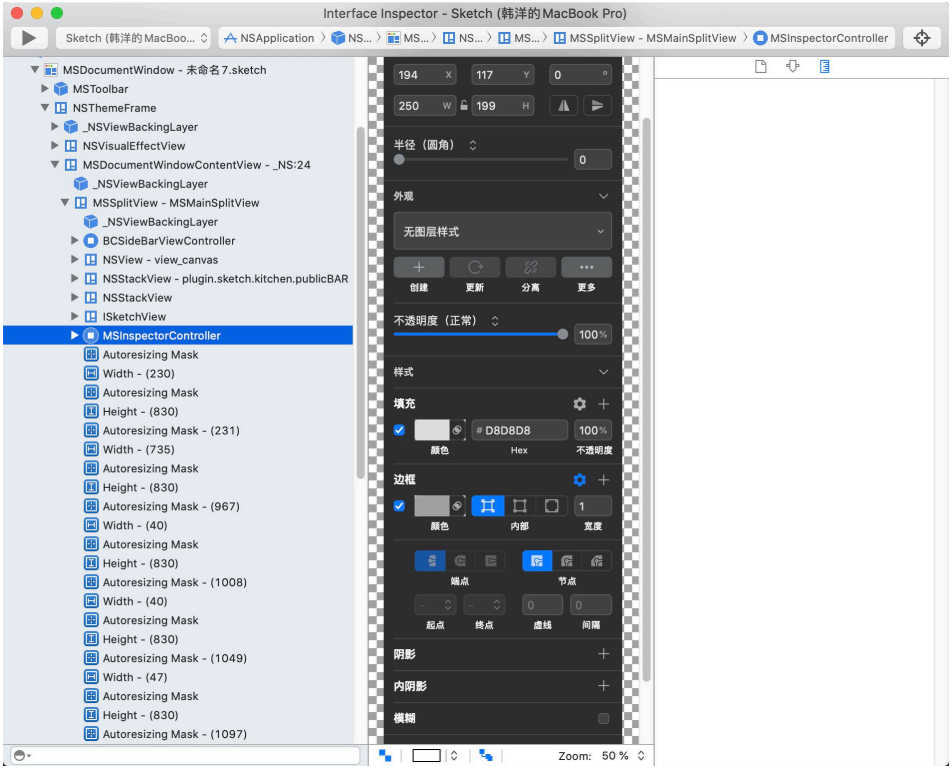
3. 更新原生属性面板

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。

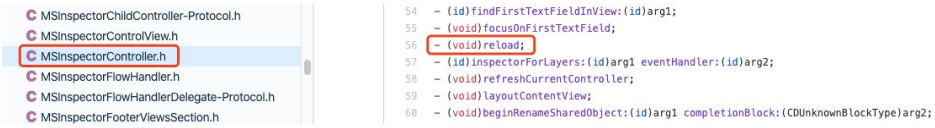
开心的下载下来后，发现这个软件上次的更新时间是 6 年前，忽然有了一种不祥的预感。果然 Attach 任何 App 时都会提示无法 Attach，在 macOS Catalina 版本已经无法运行。可是这怎么能难倒“万能”的程序员呢？我们查看系统报错，发现是 mach_inject_bundle_stub 错误，查阅发现 mach_inject_bundle_stub 是 Github 上的一个开源库，所以自己下载源码重新编译个 Bundle 包就可以了。

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

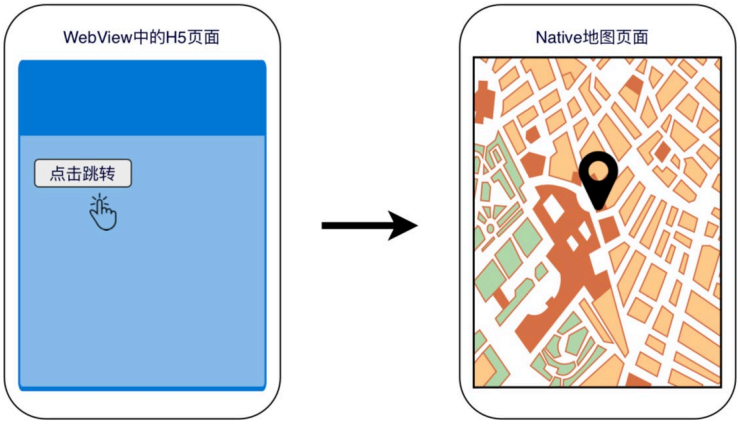
为了改善打车业务的地图体验，我们想到的方案是在展示地图的部分使用 Native 地图，而非地图部分使用 H5 页面来显示，这样既能追平与竞品的地图性能差距，又能充分发挥 H5 的开发效率。这种方案乍一看似乎是传统的 Hybrid 开发，没什么难度与新奇。比如地图使用预先内置到 App 中的地图 SDK 实现，H5 与 Native 的交互使用业界成熟的 JSBridge 技术。但从打车业务角度来看，因为打车业务有很多功能入口需要漂浮在地图之上，如起终点卡片、用户中心入口等，这种漂浮功能在技术上并不容易实现，而且还要保证用户触摸动作在漂浮元素与地图上发生时，分别派发给各自的事件系统，Hybrid 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。

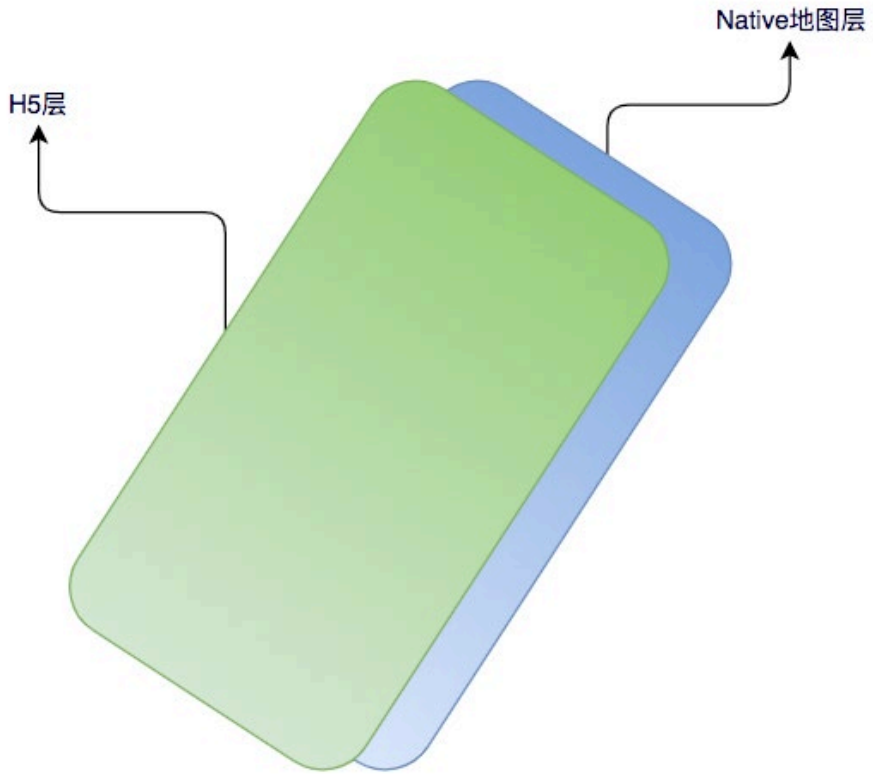
多个 WebView 组件构成的页面布局，由于内存空间不共享，它们之间信息的同步比

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

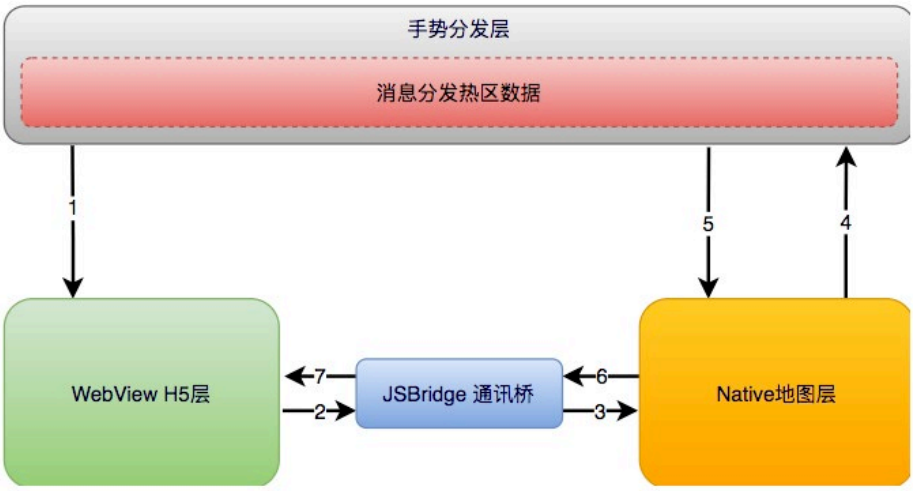
3.1 热区数据介绍



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

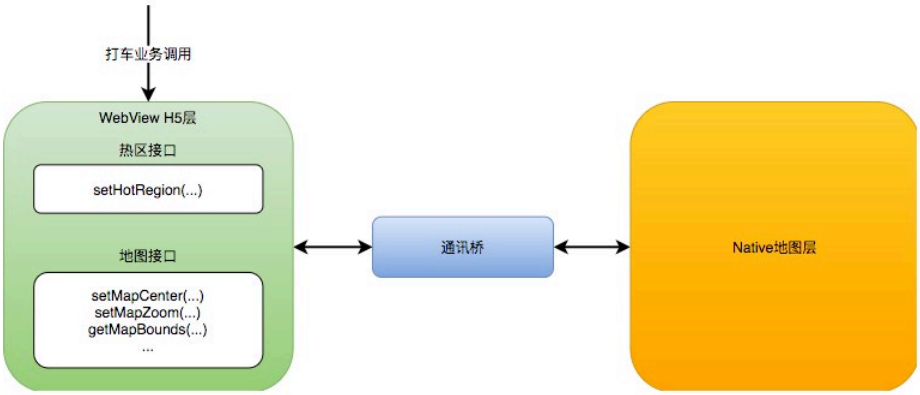
```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
   UIView *hitTestView = nil;
   // 分发层接收到手势触摸消息，通过 pointInHotspot 判断手势是否落在热区内，从而确定
消息分发的对象
   if ([self pointInHotspot:point]) {
     // 分发给 WebView H5 层
     hitTestView = [self.WebView hitTest:point withEvent:event];
   }else{
     // 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。

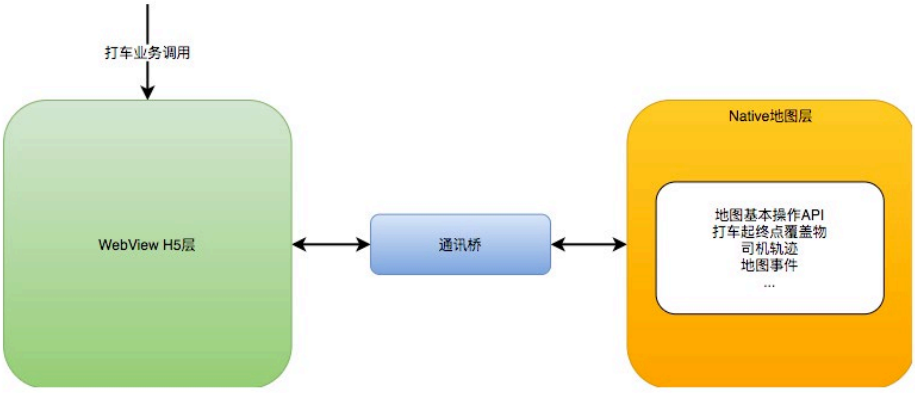


4.3 通讯桥简介

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。

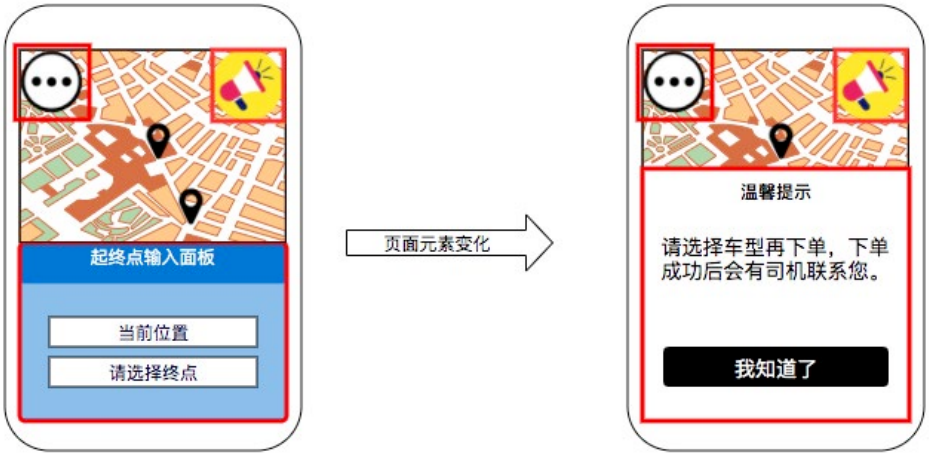
4.4 Native 地图层

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）

前端

溪流汇成海，梦站成山脉。

十余年，前端领域从打包在后端工程中的一段 HTML 和 JS 代码逐渐演化出具有完整技术体系的全新篇章。

十余年，美团点评前端团队已成为美团点评业务发展的重要技术支撑。

十余年，美团点评也成为了全球最大的生活服务电子商务平台。

十余年的技术积累，是美团技术团队继续成长的有力后盾。过去的一年间，美团技术博客继续践行「分享美团点评优质技术内容、传递美团点评技术文化」的宗旨，从美团点评内部提炼不少优质内容，分享出来与业界同仁一起学习交流。其中前端领域发布数十篇优质文章，包括动态化、代码质量及安全、App 流程管理及实践、前端基本功等。

希望同业界同仁一起交流切磋，共同成长！

MTFlexbox 自动化埋点探索

叶梓

背景

跨平台动态化技术是目前移动互联网领域的重点关注方向，它既能节约人力，又能实现业务快速上线的需求。经过十年的发展，美团 App 已经变成了一个承载众多业务的超级平台，众多的业务方对业务形态的快速迭代和更新提出了越来越高的要求。传统移动端“静态”的开发方式存在一系列问题，比如包体积增长过快、线上 Bug 修复困难、发版周期长等，已经不能满足高速发展的业务需要。因此，美团平台自研了一套跨平台动态化方案——MTFlexbox。

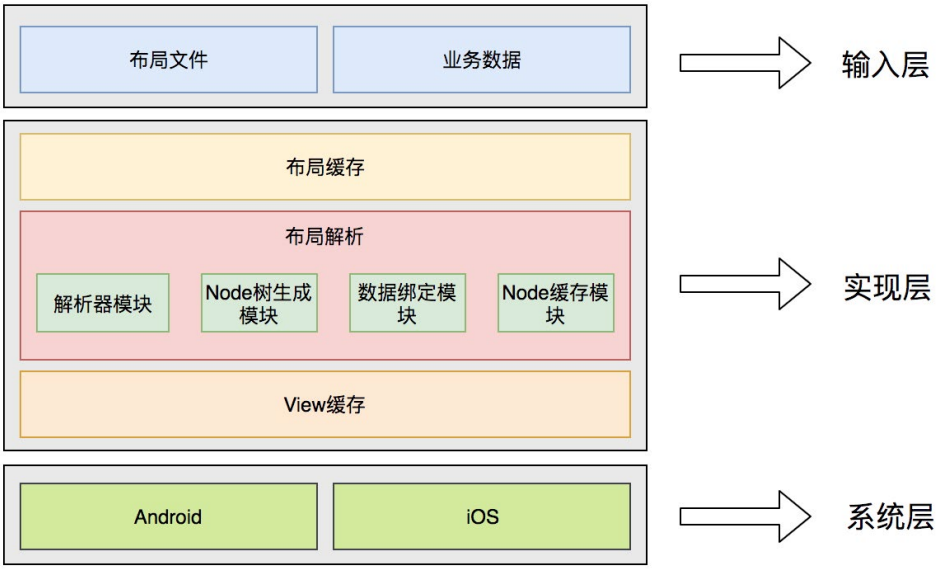
目前，MTFlexbox 已经广泛应用于美团首页、搜索、外卖等多个业务场景，并且已稳定运行两年有余。在 MTFlexbox 规范下，只需要写一份布局文件，就可以适用多端。在实际开发中，客户端开发同学开发布局的同时也要添加好埋点信息，帮助产品同学来评估上线后的效果。但现有布局埋点存在成本过高、准确率较低等痛点，为了解决这些问题，我们充分了解数据组开发人员和产品对数据统计的诉求，结合对 MTFlexbox 原理的深入理解，围绕 MTFlexbox 的埋点上报做了很多持续、有针对性的自动化工作，帮助多个项目的效率得到了显著提升。本文主要介绍美团在 MTFlexbox 自动化埋点方向所进行的一些探索，希望对大家能够有所帮助。

MTFlexbox 介绍

MTFlexbox 原理

MTFlexbox 是美团内部一套非常成熟的跨平台动态化解决方案，遵循了 CSS3 中提出的 Flexbox 规范，抹平了多平台的差异。MTFlexbox 首先按照 Flexbox 的规

范，定义了一套三端统一的 XML 布局文件，并将布局文件上传至后台；客户端下载带有布局文件的 JSON 数据后，解析布局并绑定 JSON 数据，最终交由 Native 渲染成视图。MTFlexbox 的整体架构图如下所示：



MTFlexbox 架构图

如果要用一句话来解释 MTFlexbox 的原理，就是按照约定的规则将 XML 内容映射成 Native 布局。从 Android 开发者的角度想，可以认为是把传统 XML 布局文件由内置改成从网络下发，实现展示样式动态改变的效果。上图第一层是 MTFlexbox 需要的输入，包括 XML 布局文件和展示的业务数据。其中 XML 布局文件中包括 UI 标签和埋点信息，每一种类型的埋点信息都作为一种属性和某一个 UI 标签相绑定。展示的业务数据可以通过后台下发或者写死在本地。为了将 XML 文件与具体的 View 进行解耦，MTFlexbox 在 XML 与 View 之间增加了一层 Node 层，即先将 XML 解析成 Node 树，再将 Node 树解析成 View 树。MTFlexbox 共有 3 层缓存：对 XML 文件的缓存、对 Node 节点的缓存、对 View 的缓存。其中缓存 View 指的是缓存一个 XML 创建的 View，通常只会缓存 rootView。在 Node 树生成了 View 树并绑定 JSON 数据后，才会最终渲染成 Native 控件。

MTFlexbox 适用场景

MTFlexbox 基本上支持 Native 上常用的基础控件的展示，对有 UI 定制化的需求支持度很高。但 MTFlexbox 的 XML 布局需要在运行前编写完成，只支持简单的三元表达式，逻辑能力有限。因此，MTFlexbox 特别适合布局样式复杂、变动频繁但交互简单的业务场景。例如美团 App 首页、搜索结果页等。这些业务场景都具备以下两个特点：

面向多业务方：各业务方有自己的个性化丰富样式，且不同时期可能需要不同的样式。

交互简单：点击跳转完成流量输送的简单交互。

下面是 MTFlexbox 使用场景的一些截图：



首页模板



搜索模板

MTFlexbox 自动化埋点前期工作

在美团实际的业务场景中，卡片的点击、曝光和加载数据是分析一个新产品形态上线效果好坏的最基本方式之一。相对应的，客户端的数据采集方式是洞察对于模块的点击、曝光和加载事件，然后结合上下文环境，比如页面标识、模块标识等，最后使用埋点上报工具和业务字段一起进行上报。MTFlexbox 作为模块级别的动态布局 UI 展示框架，对于数据采集方式的支持也是必不可少的。MTFlexbox 针对数据采集

的方式，做了以下两件事：

制定了一套双端统一的埋点标准化规范。

埋点类型定义成 Tag 标签属性，写入布局文件中。

MTFlexbox 结合美团自研的客户端数据上报工具，定义了多个专门针对埋点的特有属性字段，主要类型如下：

属性名	埋点类型
click-mge2-report	点击2.5埋点
click-mge4-report	点击4.0埋点
click-tag-report	点击Tag埋点
load-meg2-report	加载2.5埋点
load-meg4-report	加载4.0埋点
see-mge2-report	看见2.5埋点
see-mge4-report	看见4.0埋点

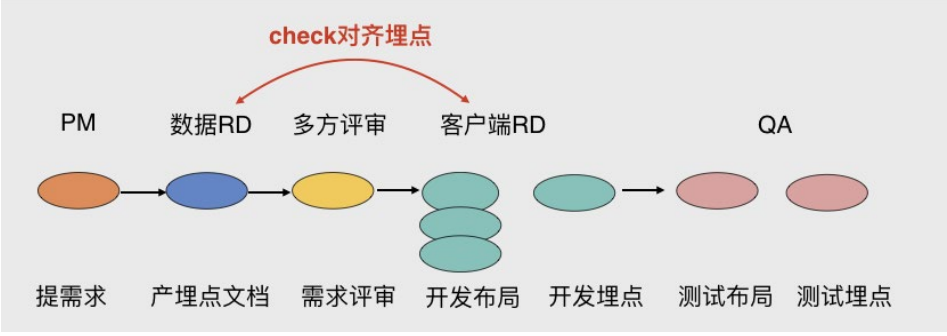
客户端开发人员在编写布局文件时，可以根据具体的产品需求，对不同控件的标签添加埋点属性，并且写入需要上报的业务字段。这样可以达到与 Native 埋点相同的效果，并且双端只需要配置一份埋点。以 see-mge4-report 埋点为例，布局埋点代码如下：

```
<Container style="width:360pt;justify-content:center;" >
    <Var name="see_MGE4" type="json">
            {
            "bid":"xxxxx",
            "cid":"yyyyy",
            "lab":{
                "isDynamic":true,
                "gather_index":"{extra.gather_index}",
                "index":"{extra.index}"
```

```
                }
            }
    </Var>
    <Container
              see-mge4-report="{see_MGE4}"
              click-url="{business.iUrl}"
              visibility="{{display.itemDynamic.
imageUrl}?visible:displaynone}" >

            <Img style="width:331pt;height:106pt;justify-content:center;"
                border-radius="5pt"
                scale-type="center-crop"
                src="{display.itemDynamic.imageUrl}"
                background="#FFF8F8F8" />
    </Container>
</Container>
```

MTFlexbox 动态化研发流程



MTFlexbox 动态化研发流程

从上述 MTFlexbox 动态化研发流程图中可以看出，数据需求和产品需求需要客户端开发人员在同一份布局文件中耦合在一起去实现，而且埋点属性和布局控件相绑定。这就导致在埋点过程中会出现很多问题，总结如下：

埋点成本过高

- 沟通成本较高：对于一个新的产品需求，首先产品需要将埋点需求提给数据组，数据同学理解了产品需求后产出埋点文档；然后产品、数据同学、客户端开发同学三方进行需求评审和埋点评审，沟通埋点需要上报的字段和时机等细节。

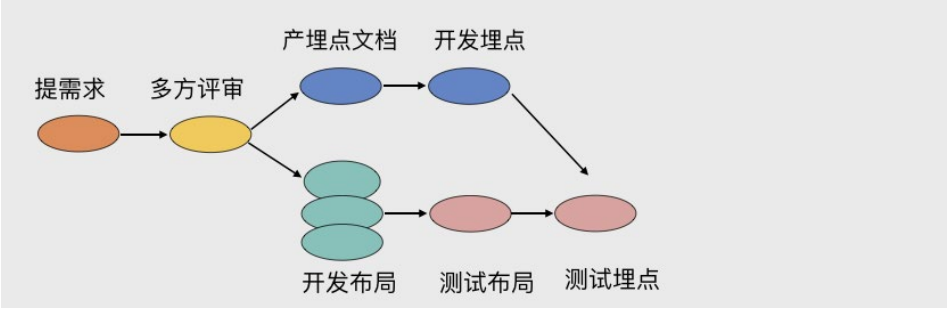
很多时候，一次沟通不到位，还需要反复沟通或者重新沟通，直到产品、数据同学和客户端开发人员三方对需求和埋点的理解一致为止。平均一个 5PD（5PD 指 5 个工作日）的需求需要消耗数据同学和客户端开发人员各 1PD 的时间来进行理解和沟通。

- 开发成本过高：客户端开发人员在编写 XML 布局文件时，往往要花 30% 左右的时间进行手动埋点和自测校验。

埋点线上事故多

- 因整个埋点缺乏自动化的埋点校验和预警机制，一旦开发人员出现人为的失误，导致错埋、漏埋现象，都有可能引发严重的线上故障。例如，客户端开发人员手动埋点时，出现人为失误引入了错误数据；产品验收阶段需要修改布局样式，客户端开发人员会出现“仅修改布局而遗漏埋点”的问题。

鉴于 MTFlexbox 存在埋点成本过高和线上问题较多的突出问题，我们迫切的希望通过一些手段来最大程度的规避和解决这类问题。埋点成本过高的原因在于 MTFlexbox 将布局和埋点耦合在一起编写，客户端开发人员需要做的事情过于“杂”和“多”。找到了这个痛点，很容易想到将埋点上报和布局编写解耦，让客户端开发人员只负责编写布局，数据同学只负责埋点配置，以此降低开发和沟通成本；同时通过自动化埋点校验手段提升埋点准确率，优化流程，减少线上事故的产生。基于此，产出我们理想的布局和埋点解耦之后的动态化研发流程，如下图所示：



新的动态化流程

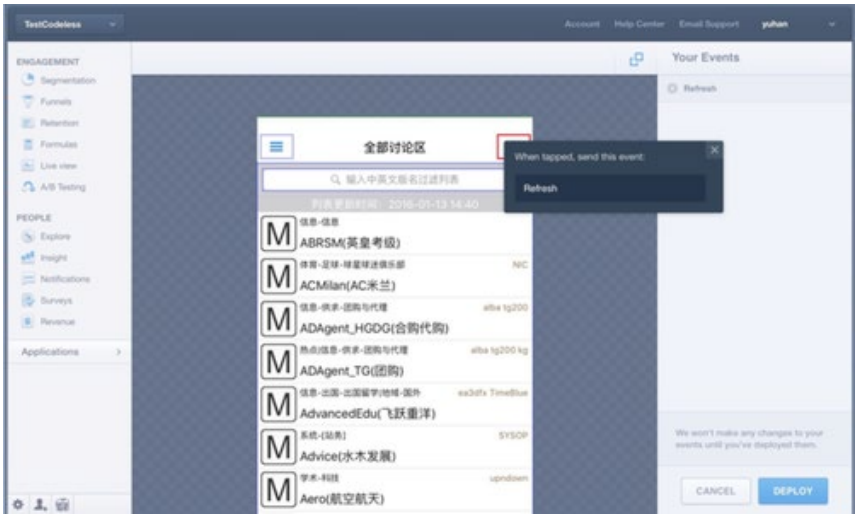
业内自动化埋点方案调研与参考

美团外卖前端无痕埋点实践

外卖团队在他们原有代码埋点方案的基础上，演化出了一套轻量的、声明式的前端埋点方案。详细内容可以参考博客:《美团点评前端无痕埋点实践》。此方案通过声明式埋点的方式实现了埋点代码与业务逻辑的解耦，并且支持对通用的业务数据的自动化上报。但此方案不能完全实现自动化埋点，并且实现成本较高。

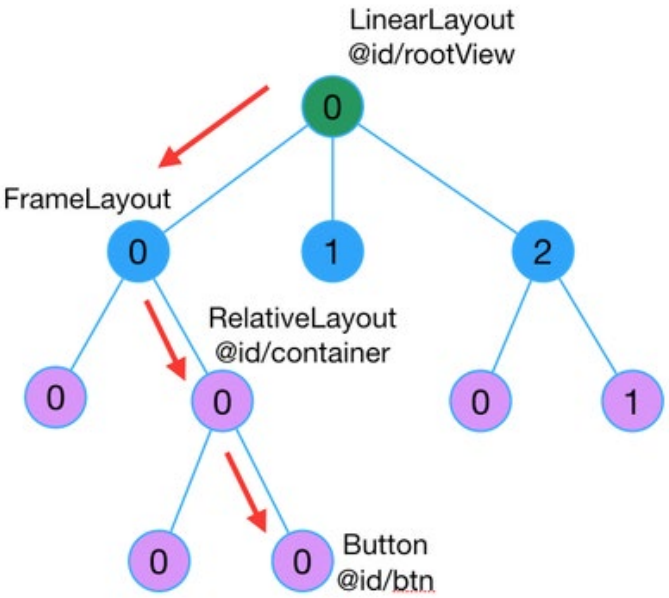
Mixpanel

Mixpanel 是一个已经商业化的可视化埋点方案，采用了截屏的方式在 IDE 中完成控件的圈选操作，体验较好值得我们借鉴。不过该方案主要面向非技术人员，不支持上报业务字段数据。



HubbleData

HubbleData 是网易开发的一个洞察用户行为的数据分析系统，提供一套完整的数据解决方案。



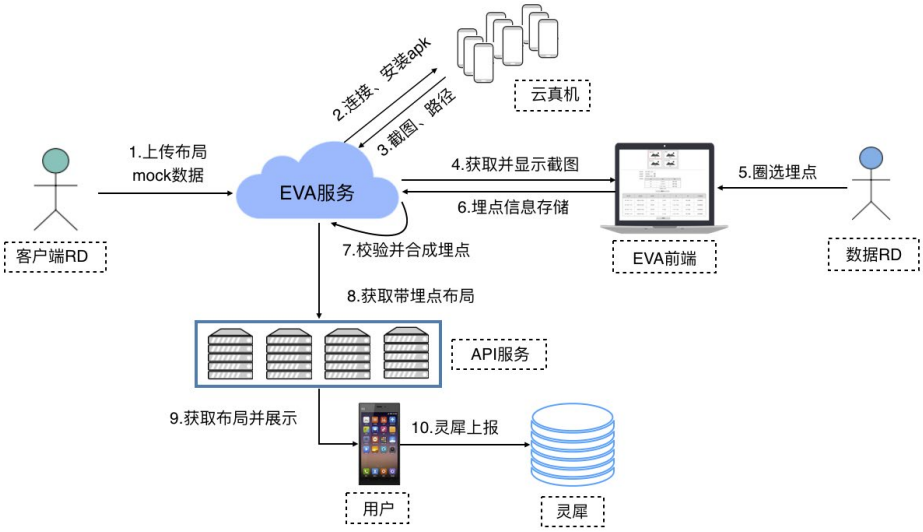
网易对 XPath 做了优化，主要体现在 View 索引的计算上：

- 原始 XPath 计算方式：每个 ViewGroup 下的所有 View 作为一个数组，索引从 0 开始。例如上图 Button 控件的 XPath 标识为：LinearLayout[0]/FrameLayout[0]/RelativeLayout[1]/Button[1]
- 网易 XPath 计算方式：每个 ViewGroup 下的所有 View 先按控件类型分类，然后再把每个类型中的控件按照数组的方式，从 0 开始。例如上图 Button 控件的 XPath 标识为：LinearLayout[0]#rootView/FrameLayout[0]/RelativeLayout[0]#container/Button[0]#btn

但是网易的这次优化，并没有解决由于同类型控件位置变更而引发的埋点错误问题，根源在于控件唯一标识不够准确。同时网易的修改仅限控件的一些固有属性，并没有搜集到更有价值的业务数据。

结合上述四种方案的优缺点，自动化埋点需要具备的几个条件，即：简洁直接的流程、友好可视化的前端配置界面、业务字段的可配置化、埋点有效性的检测。我们的方案就是基于这几个目标而诞生的。

我们的方案



整体流程

MTFlexbox 自动化埋点的核心流程，分为以下五步：

1. 客户端开发人员根据设计稿开发 XML 样式文件，自测通过后将 XML 样式文件与接口数据上传至 MTFlexbox 管理后台。
2. MTFlexbox 管理后台自动连接远程移动设备，并发送布局处理命令。远程移动设备将布局渲染结束后，抓取截图和布局层级信息（包括控件父子关系、控件位置、大小等信息）并上传至管理后台。
3. 前端页面从后台拿到 DPath 路径信息、坐标信息和截图信息，提供一套可视化的界面供数据同学进行模块内任一控件的埋点圈选配置。数据同学根据自身的需求，从目录树中圈选出自己希望配置埋点的控件。如下图所示，右侧模块中会出现红圈将选中的控件标出。



目录树圈选控件

4. 选中某个控件之后，数据同学对该控件进行埋点配置，元素类型支持当前元素和同类元素。其中同类元素可以节省数据同学对于同一种类型的控件的多次配置。对于已经圈选出的控件，列表中会详细展示出相关的信息，并附上控件对于的位置截图，能够方便数据 RD 定位埋点的控件具体位置。



埋点配置

5. MTFlexbox 管理后台根据前端上报的埋点信息，生成包含业务埋点的 XML 样式文件，供 C 端业务方后台调用。

```
<?xml version="1.0" encoding="UTF-8"?>
<Container>
    <Var auto-mge="true" name="ff510aa110844bb78c0b86fb04b26460"
type="json">
        {
            "bid" : "xxxxx",
            "cid" : "sssss",
            "lab" : {
                "index" : "{_index}",
            }
        }
    </Var>

    <!-- 整个容器 -->
    <Container background="#FFFFFF" border-radius="10pt"
        click-mge4-report="{ff510aa110844bb78c0b86fb04b26460}"
        click-url="{_iUrl}" padding-left="10pt" padding-right="10pt">
        <!-- 左半部分 -->
        <Container style="flex-direction:column;justify-content:flex-
start;margin-top:15pt;">
```

6. 当客户端请求业务后台时，业务后台将包含业务埋点的 XML 样式文件下发给客户端，客户端根据配置完成埋点信息上报。

总结与展望

目前 MTFlexbox 自动化埋点方案已经使用在美团首页、大搜等业务中，整体埋点成本降低了 80%，上线后且无埋点故障。在此埋点方案的实现过程中，我们也踩了很多在设计之初没有预想到的坑，遇到了一些难点，详细设计问题和解决方案稍后的博客中的详细介绍，敬请关注美团技术团队公众号。

目前，我们基于 MTFlexbox 实现了 View 与埋点的自动化绑定，后期我们规划通过规范标准化后台下发的数据，包括业务数据和埋点数据，进而实现埋点数据的动态化下发和自动化绑定，进一步节省在埋点配置阶段和测试阶段的人力投入。

参考资料

[1] 网易 HubbleData 之 Android 无埋点实践
[2] 商业化埋点实现方案 mixpanel
[3] 美团点评前端无痕埋点实践

作者简介

叶梓、腾飞、田贝、张颖，美团终端业务研发团队研发工程师。

招聘信息

美团终端业务研发团队的职责是保障平台业务高效、稳定迭代的同时，持续优化用户体验和研发效率。团队负责的业务主要包括美团首页、美团搜索等千万级 DAU 高频业务以及分享、账号、音 / 视频等基础业务，支撑和对接外卖、酒店等 30 多个业务方。团队通过动态化能力建设，加快业务上线速度，帮助产品（PM）快速验证业务选型，做出业务决策；架构 / 服务标准化体系建设，提升前后端以及平台与业务线的沟通、合作效率；业务监控和体验优化，有效保障核心业务服务成功率的同时，提升用户使用美团 App 过程中的稳定性和流畅性。团队开发技术栈包括 Android、iOS、ReactNative、Flexbox 等。

美团终端业务研发团队是一个活力四射、对技术充满激情的团队，现诚聘 Android、iOS 工程师，欢迎有兴趣的同学投简历至 tech@meituan.com。

Litho 在美团动态化方案 MTFlexbox 中的实践

少宽　腾飞　叶梓

MTFlexbox

MTFlexbox 是美团内部应用的非常成熟的一种跨平台动态化解决方案，它遵循了 CSS3 中提出的 Flexbox 规范来抹平多平台的差异。MTFlexbox 适用于重展示、轻交互的业务场景，与现有 HTML、React Native、Weex 等跨平台方案相比，MTFlexbox 具备着性能高、渲染速度快、兼容性高、原生功能支持度高等优势。但其缺点在于不支持复杂的交互逻辑，不适合复杂交互的业务场景。目前，MTFlexbox 已经广泛应用在美团首页、搜索、外卖等重要业务场景。本文主要介绍在 MTFlexbox 中使用 Litho 优化性能的实践经验。

MTFlexbox 的原理

MTFlexbox 首先定义一份跨平台统一的 DSL 布局描述文件，前端通过“所见即所得”的编辑器编辑产生布局，客户端下载布局文件后，根据布局中的描述绑定 JSON 数据，并最终完成视图的渲染。MTFlexbox 框架图如下图所示：

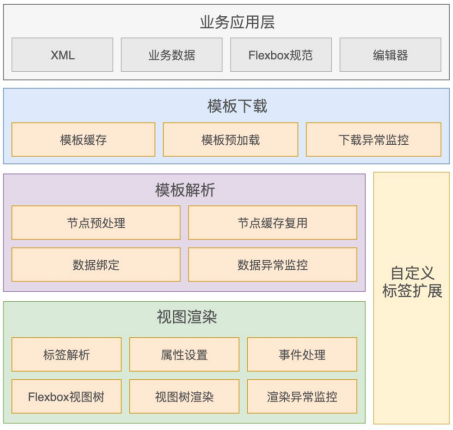


图 1　MTFlexbox 的架构

图中分为五层，分别是：

- 业务应用层：业务使用 MTFlexbox 的编辑器定义符合 Flexbox 规范的 DSL 文件（XML 模版）。
- 模版下载：负责 XML 模版下载相关的工作，包括模版缓存、预加载和异常监控等。
- 模版解析：负责模版解析相关的工作，包括标签节点的预处理、数据绑定、标签节点的缓存复用和数据异常监控等。
- 视图渲染：负责视图渲染相关的工作，包括把标签结点按照 Flexbox 规范解析成 Native 视图，并完成视图属性的设置、点击曝光事件的处理、视图渲染、异常监控等。
- 自定义标签扩展：提供支持业务扩展自定义标签的能力。

鉴于本篇博客主要涉及渲染相关的内容，下面将着重介绍 MTFlexbox 从模版解析到渲染的过程。如下图所示，MTFlexbox 首先会把 XML 模版解析成 Java 中的标签树，然后和 JSON 数据绑定结合成一颗具有完整数据信息的节点树。至此，模版解析工作就完成了。解析完成的节点树会交给视图引擎进行 Native 视图树的创建和渲染。

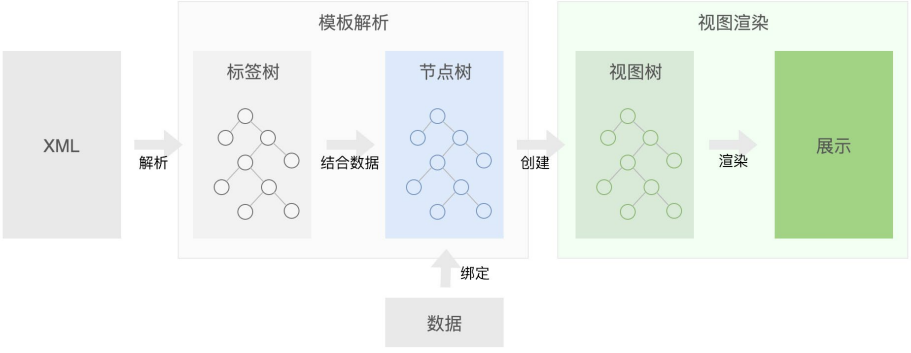


图 2　视图模版从解析到渲染

MTFlexbox 在美团动态化实践中面临的挑战

随着 MTFlexbox 在美团内部被广泛使用，我们遇到了两个问题：

- 复杂视图因层级过深，导致滑动卡顿问题。
- 生成视图耗时过长，导致滑动卡顿问题。

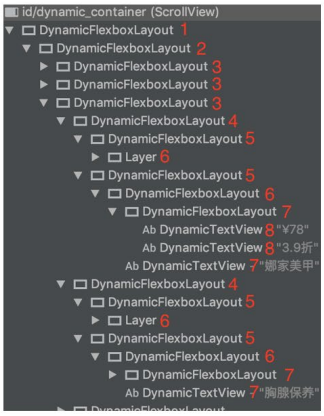
问题一：视图层级过深

原因分析

MTFlexbox 使用的是 Flexbox 布局，Flexbox 布局可以理解成 Android LinearLayout 布局的一种扩展。Flexbox 在布局过程中使用到大量的布局嵌套，如果布局酷炫复杂，无疑会出现布局层级过深、视图树遍历耗时、绘制耗时等问题，最终引发滑动卡顿。下图是美团正在使用的一个模版的视图层级情况（布局最深处有 8 层）：



显示布局边界下的视图效果



视图树

图 3　模版布局层级效果

影响

布局层级过深在布局的计算和渲染过程中会导致过多的递归调用，影响视图的绘制效率，引发页面滑动 FPS 下降问题，这会直接影响到用户体验。

问题二：生成视图耗时过长

原因分析

视图生成耗时原因如下图所示：RecyclerView 在使用 MTFlexbox 布局条目时，需要对条目模版进行下载并解析生成节点树，这样会导致生成视图的过程耗时过长。为了提高视图生成速度，我们增加了复用机制，但是滑动过程中，如果遇到新的布局样式仍然需要重新下载和解析。另外，MTFlexbox 绑定的数据是未经解析的 JSON 字符串，所以也要比正常情况下的数据绑定更耗时一些。 正是上面两个原因，导致了 MTFlexbox 生成视图耗时过长的问题，这也会导致滑动时 FPS 出现突然下降的现象，产生卡顿感。

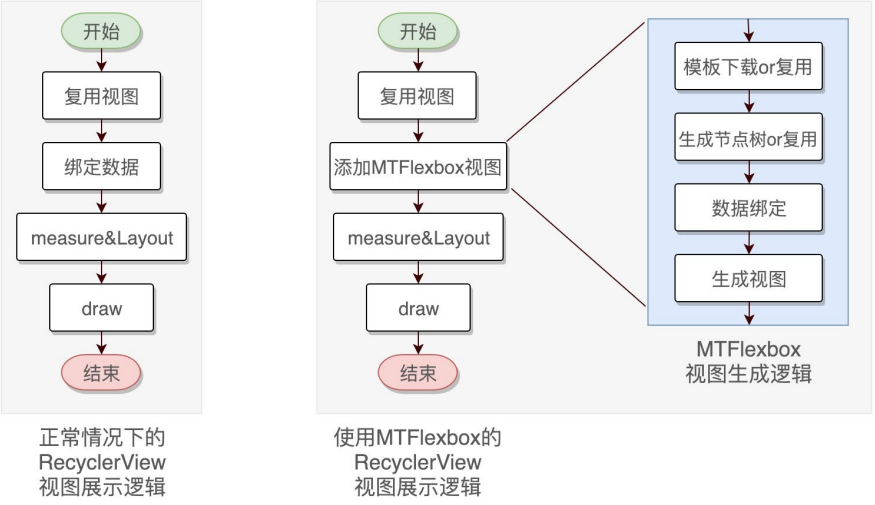


图 4　视图生成耗时原因分析

影响

由于视图的创建会阻塞主线程，创建视图耗时过长会导致 RecyclerView 列表滑动时卡顿感明显，也严重影响到了用户体验。

Litho

Litho 原理

Litho 是一套声明式 UI 框架，或者说是一个渲染引擎，它主要优化复杂 RecyclerView 列表的滑动性能问题。Litho 实现了视图的细粒度复用、异步计算布局和扁平化视图，可以显著提升滑动性能，减少 RecyclerView 滑动时的内存占用。详细介绍可以参考美团技术团队之前发布的另一篇博客：Litho 的使用及原理剖析。

Litho 的优势

通过对 Litho 原理的了解，我们可以看到 Litho 主要针对 RecyclerView 复杂滑动列表做了以下几点优化：

- 视图的细粒度复用，可以减少一定程度的内存占用。
- 异步计算布局，把测量和布局放到异步线程进行。
- 扁平化视图，把复杂的布局拍成极致的扁平效果，优化复杂列表滑动时由布局计算导致的卡顿问题。

扁平化视图刚好可以优化 MTFlexbox 遇到的视图层级过深的问题。异步计算布局虽然不能直接解决 MTFlexbox 生成视图耗时过长问题，但是给问题的解决提供了新的思路——异步提前完成视图创建。而且使用 Litho 还能带来一定程度的内存优化。所以如何将 Litho 应用到 MTFlexbox 中，进而来解决 MTFlexbox 现存的问题，是我们解下来要讨论的重点。

Litho + MTFlexbox 是怎么解决上述两个问题的?

解决问题一：视图层级过深问题

Litho 实现了布局的扁平化，所以最直接的方式就是使用 Litho 来替换 MTFlexbox 现有的视图引擎。视图引擎最主要的作用，是把 XML 文件解析出来的节点树变成 Litho 可以展示的视图，所以视图引擎替换的主要工作是把节点树转换成 Litho 能展示的视图。如下图所示。由于 Litho 使用的是组件化思想，需要先把节点转化成组

件，再把组件树设置给 LithoView，而 LithoView 是 Litho 用于兼容原生 View 的容器，它负责把 Litho 和系统视图引擎桥接起来。

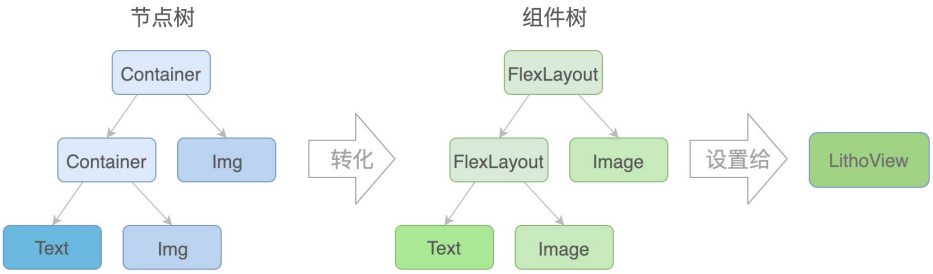


图 5　Litho 视图引擎从节点到视图的转换

不过视图引擎的替换并不是一帆风顺的，我们在替换过程中也遇到了 4 个比较大的挑战。

难点一：复用视图无法更新数据问题

问题描述：完成了节点树到组件树的转化以后，我们发现了一个严重的问题——复用的视图无法应用新的数据。

问题分析：当数据发生变化后，MTFlexbox 的节点树会对比新旧数据的变更，确定哪些结点需要更新并通知到具体的视图节点，然后更新显示内容（例如：新数据相比旧数据改变了 Text，那么只有 Text 对应的节点会通知对应的视图去更新内容）。Litho 组件的 Prop 属性是不允许更改的，而 Litho 组件中绝大多数属性都是 Prop 属性。

解决方案

方案一：使用 State 属性全局替换所有组件的 Prop 属性。这种方式的优点在于替换方式相对简单直接，缺点是侵入性强，替换工作量巨大且不符合 Litho 的思想（尽可能少的去改变组件的状态）。这种方案不是最优解，我们要降低侵入，简单快捷地实现数据更新，于是就产生了方案二，具体如下图所示。

方案二：封装一套 Updater 组件，用于创建真正展示的组件。Updater 组通过 State 属性监听对应节点的数据变更，当节点数据变化时，可以触发对应节点的更新。

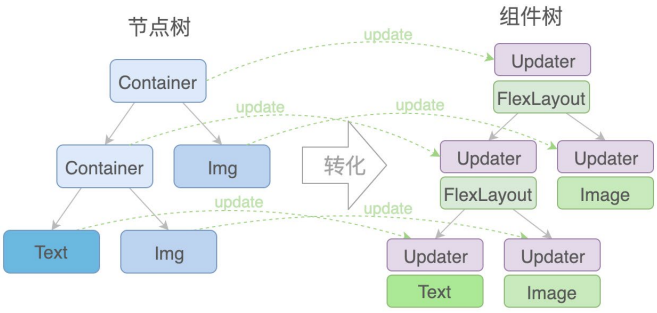


图 6　数据更新问题初版解决方案

但在后来的实践过程中，我们发现 Litho 整个组件树中只要有一个组件有状态更新，便会重新计算整个布局，而每次数据更新少说也会有几十个节点发生变化。频繁的重复计算反而导致性能变得很差。在经过了多种尝试以后，我们找到了最优的解决方案：

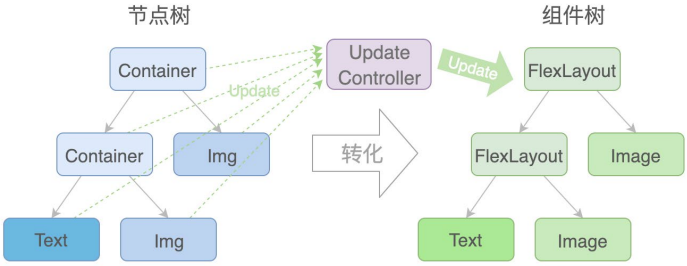


图 7　数据更新问题最终解决方案

如上图所示，状态更新控制器负责整个视图所有节点的更新操作。在所有数据都更新完成以后，统一交由状态更新控制器触发一遍组件更新。

难点二：Litho 不支持层叠布局问题

MTFlexbox 并没有完全严格的使用 Flexbox 布局规范，为了简单实现层叠效果，MTFlexbox 自定义了一种新布局规范——Layer 布局。Layer 布局具有以下两个特点：

- 特点一：Layer 的子视图在 z 轴上依次层叠展示。
- 特点二：Layer 的子视图默认且只能充满父布局。

原因分析：由于 Litho 严格遵守 Flexbox 布局规范，所以没有现成的 Layer 组件。

解决方案：自己实现 Layer 组件，满足第一个特点很容易，Flexbox 本身就支持层叠展示，只需要把子视图设为绝对布局就可以了。但是让子视图默认充满父布局就没有那么简单了，Flexbox 布局中没有任何一个属性可以达到这个效果。在经过了若干次组合多个属性的尝试以后，还是没能找到解决方案。既然 Layer 并不是 Flexbox 布局的规范，那么我们局限在 Flexbox 的束缚下，怕是很难找到完美的解决方案。那么，能不能在 Litho 中绕过 Flexbox 的约束，自己实现 Layer 效果呢？想在 Litho 中突破 Flexbox 布局的束缚，就需要了解 Litho 是如何使用 Flexbox 的。

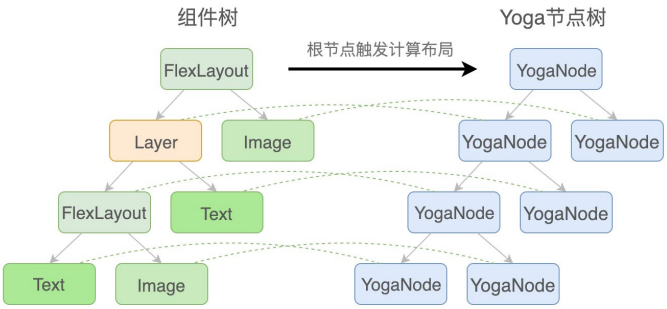


图 8　Litho 的布局计算原理

如上图，Litho 的 Flexbox 布局是由 Yoga 负责布局计算的。每一个 Litho 组件都会对应一个 Yoga 节点。但 Yoga 的布局计算过程是由根节点去统一触发的，子节点没有办法知道自己对应的 Yoga 节点是何时开始计算，及何时计算结束。这样以来，我们就没有时机去感知到 Layer 组件的布局是否计算完成，也就没有办法在 Layer 组件计算完成后去控制 Layer 子节点的计算。为了解决这个问题，我们做了两件事：

- 添加布局计算完成的回调，在布局计算完成后由根节点逐层通知子节点计算完成的消息。

- 拆分 Yoga 节点树，由 Layer 自己来控制子节点的计算。

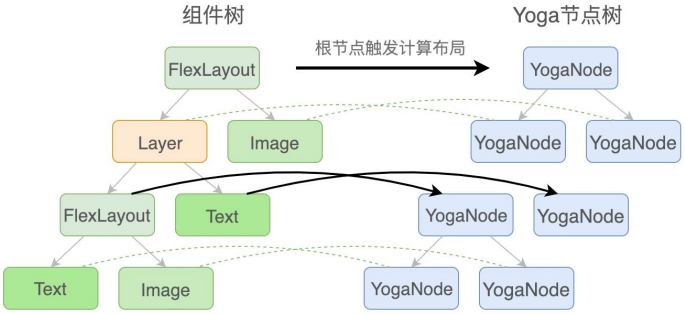


图 9　Layer 布局的实现原理

如上图所示，把 Layer 组件伪造成叶子节点，不把 Layer 组件的子节点设置给 Yoga，这样一个 Yoga 中的布局树就被 Layer 组件切割开了。当根节点计算完成以后，通知到 Layer 组件，Layer 组件再依次去设置子节点的宽高和位置属性，并触发子节点去完成各自子节点的布局计算。这样就完美地实现了 Layer 的布局效果。

难点三：Litho 图片组件不支持使用网络图片问题

原因分析：Litho 的组件是一个属性的集合，Litho 期望我们在组件创建时便确定了所有属性的值，所以 Litho 不支持网络图的展示。如果要支持从网络下载图片，就意味着图片组件用来展示的内容会发生变化。所以 Litho 自带的图片组件并不支持使用网络图片。

解决方案

方案一：用 State 属性解决网络图片下载带来的展示内容变化问题。我们在实践中发现，State 属性的更新会导致整个布局重新计算，其实替换图片资源不会导致图片组件的大小位置发生变化，根本不需要重新计算布局。为了减少使用 State 属性导致布局计算频繁的问题，就摒弃了这种方案。

方案二：Litho 官方额外提供的异步下载图片组件 FrescoImage 中使用的是图片代理方式。FrescoImage 使用 DraweeDrawable 来绘制视图，而 DraweeDrawable 实际上并不具备图片渲染的能力，只是在内部保存了一个真正的 Drawable 来负责渲

染。所以，DraweeDrawable 本质上是对真正要展示的图片做了一层代理，当从网络上下载下来真正要展示的图片后，只需要通过替换代理图片就可以完成视图的更新。美团下载图片使用的是 Glide，只需要按照这个思路实现自己的 GlideDrawable 就好了。

难点四：自定义标签扩展的接口不兼容问题

MTFlexbox 支持自定义标签的扩展，所以我们在完成基本视图标签的 Litho 实现以后，还需要支持自定义 Tag 的扩展，才算完成视图引擎的替换工作。

原因分析：MTFlexbox 在设计自定义标签接口时，只提供了允许使用 View 完成视图扩展的接口，但是 Litho 实现的视图引擎是使用组件作为视图单元和 MTFlexbox 对接的，所以接口不能兼容。

解决方案

方案一：重新提供使用 Litho 组件完成视图扩展的接口。其缺点是，需要 MTFlexbox 的使用方重新实现已经支持了的自定义标签，工作量较大，所以这种方案被抛弃了。

方案二：Litho 中使用业务方已经扩展好的 View。其优点是使用方对视图引擎的替换无感知。那么，怎样才能在 Litho 中使用业务方已经扩展好的 View 呢？可以先看下面这张图。

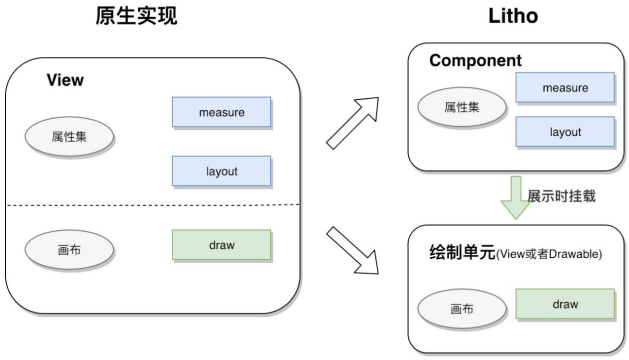


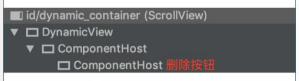
图 10　Litho 对 View 功能的拆分

我们可以简单的理解成 Litho 对 Android 的 View 做了一个功能拆分，把属性和

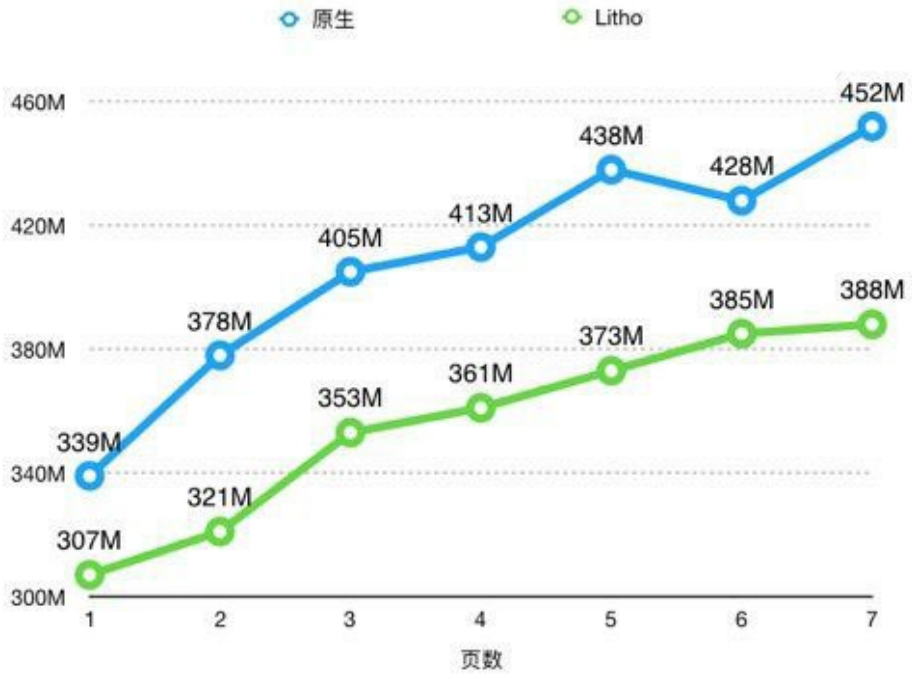
布局计算的能力放在了组件里面，每一种组件对应一个绘制单元来专门负责绘制。那么对于使用方扩展的标签，我们可以定义一个通用组件来统一承接。在挂载绘制单元时，再去调用使用方扩展的视图去绘制。

优化效果

至此，视图引擎的替换就完成了，整个视图引擎的替换做到了使用方无感知。完美解决了 MTFlexbox 视图层级深的问题，顺带还优化了部分性能。下面是布局层级优化效果的对比，可以看到相同样式下，使用 Litho 引擎实现的视图比使用 MTFlexbox 原生引擎的视图层级要浅很多。

视图引擎	显示布局边界下的视图效果	视图树
Litho		
原生		

除此之外，还有我们的内存优化成果。下图是美团首页使用 MTFlexbox 时，内存占用随滑动页数（一页为 20 条数据）增加而变化的趋势图。可以看到，使用 Litho 引擎实现的 MTFlexbox 比使用原生引擎的 MTFlexbox 在内存占用上能有 30M 以上的优化。

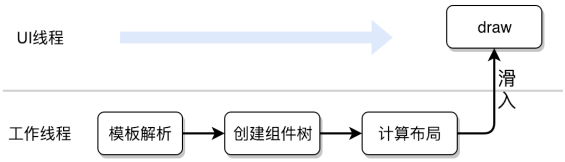


解决问题二：生成视图耗时过长

上文提到导致生成视图耗时过长的有两个原因：

（1）MTFlexbox 对布局模版的下载和解析耗时。（2）MTFlexbox 绑定时解析数据的耗时。

上文“自定义标签扩展的接口不兼容问题”中介绍过 Litho 的组件能够独立完成布局计算。另外，Litho 组件是轻量级的，所以我们直接把 Litho 组件作为 RecyclerView 适配器的数据源。这样就需要在数据解析时提前完成组件的创建，而组件的创建需要用到 MTFlexbox 的整个解析过程，也就是说，我们把 MTFlexbox 导致视图生成耗时过长的过程提前在数据层异步完成了。这样就不需要等到视图要展示时再去解析，从而规避了视图生成耗时过长的问题。具体的原理，可以参见 Litho 的使用及原理剖析一文中的 3.2 节“异步布局”。



如上图所示，在异步线程中提前完成 MTFlexbox 布局到 Litho 组件的转换。当视图真正要展示时，只需要把组件设置给 LithoView 就可以了。

优化效果

使用 Litho 引擎实现的滑动列表，在连续滑动过程中不会出现 FPS 波动问题，而使用 MTFlexbox 原生引擎实现的滑动列表则波动明显。（数据采集自魅蓝 2 手机，中低端手机优化效果明显）



总结

经过一段时间的实践，Litho + MTFlexbox 给美团 App 在性能指标上带来了较大的提升。但是还有很多问题待完善，我们后续还会针对以下几点进一步提升效果：

- 利用 Litho 组件属性不可变的特点，将提前异步布局进一步扩展为提前渲染出位图，在绘制时直接展示位图，可以进一步提升绘制效率。
- 优化 RecyclerView 相关的 API，降低侵入性。
- 解决有点击事件、埋点事件等属性的视图需要降级成 View 才能使功能生效的问题，进一步优化视图层级。

参考资料

Litho 官网　Flexbox 规范

作者简介

少宽、腾飞、叶梓，美团终端业务研发团队开发工程师。

招聘信息

美团终端业务研发团队的职责是保障平台业务高效、稳定迭代的同时，持续优化用户体验和研发效率。团队负责的业务主要包括美团首页、美团搜索等千万级 DAU 高频业务以及分享、账号、音 / 视频等基础业务，支撑和对接外卖、酒店等 30 多个业务方。

团队通过动态化能力建设，加快业务上线速度，帮助产品团队快速验证业务选型，做出业务决策；通过架构 / 服务标准化体系建设，提升前后端以及平台与业务线的沟通、合作效率；业务监控和体验优化，有效保障核心业务服务成功率的同时，提升用户使用美团 App 过程中的稳定性和流畅性。团队开发技术栈包括 Android、iOS、ReactNative、Flexbox 等。

美团终端业务研发团队现诚聘 Android、iOS 工程师，欢迎有兴趣的同学投简历至：tech@meituan.com（注明：美团终端业务研发团队）。

开源 React Native 组件库 beeshell 2.0 发布

小龙　宋鹏

引言

随着 React Native（以下简称 RN）技术的出现，大前端的发展趋势已经势不可挡，跨平台技术因其通用性、低成本、高效率的特点，逐渐成为行业追捧的热点。

为了进一步降低开发成本、提升产品迭代的效率，美团开始推广使用 RN 技术。随之而来，相关业务方开始提出对 RN 组件库的诉求。2018 年 11 月，公司内部发起了 RN 组件库建设，旨在提供全公司共用的组件库。鉴于我们团队在开源 beeshell 1.0（建议阅读 1.0 版本推广文章）时，积累了丰富的经验，于是就加入到了公司级 RN 组件库的项目共建中。完成组件库开发后，我们决定将共建的成果贡献出来，并以 beeshell 升级 2.0 的形式进行开源，共计开源 38（33 个组件与 5 个工具）个功能。在服务社区的同时，也希望借助社区的力量，进一步完善组件库。



图 D

beeshell 2.0 效果图

背景

前端开发（包括 Web 与 Native 开发）是图形用户界面（GUI）开发的一种。纵观各种 Web 以及 Native 产品界面，发现它们都是由一些基本组件（控件、元素）组合而成。受益于组件化、模块化的开发方式，前端 MV* 框架（如：AngularJS、React、Vue.js 等）也得以蓬勃发展。借助这些组件化框架，前端开发构建用户界面的过程，本质上是：开发组件，处理组件间的组合与通信。

这样看来，如果实现一套通用的组件并有效复用，便可以大幅减少开发组件的工作，进而提升前端开发的效率。各个业务方对 RN 组件库的诉求，目的也在于此：降低成本，提升效率。

- 然而，公司内不同事业部的业务场景和产品功能不尽相同，如何通过一套组件，来有效的支撑外卖、配送、酒旅及其他事业部的业务需求？这无疑是对组件库提出了更高的要求：
- UI 风格一致性。在同一个业务中，各个组件要有一致的 UI 风格，保证用户体验、塑造品牌形象。
- 通用性。可以支持不同的业务方，可以灵活定制不同的业务需求，最大化组件复用率，减少重复开发。
- 易用性。组件的功能、行为表现符合开发者的直观感受，易于学习和使用、减轻记忆负担；功能丰富，可以支持多种业务场景，支持特定业务场景的多种情况。
- 稳定性。RN 组件库需要同时支持 iOS、Android、Web 三个平台，组件要在三个平台可用、可靠、稳定。

简而言之，组件库的目标是通用、易用、稳定、灵活。

系统设计

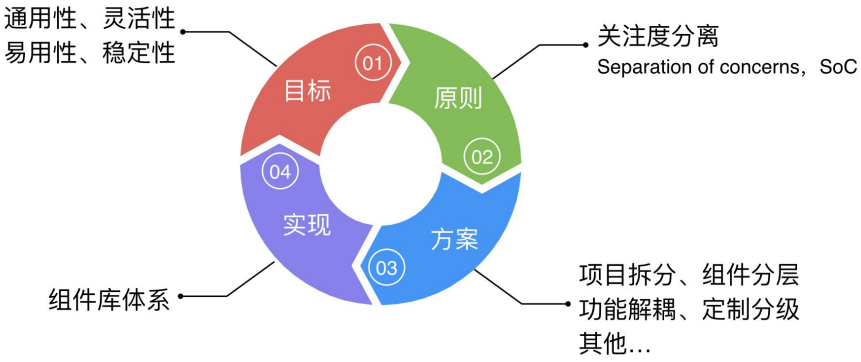


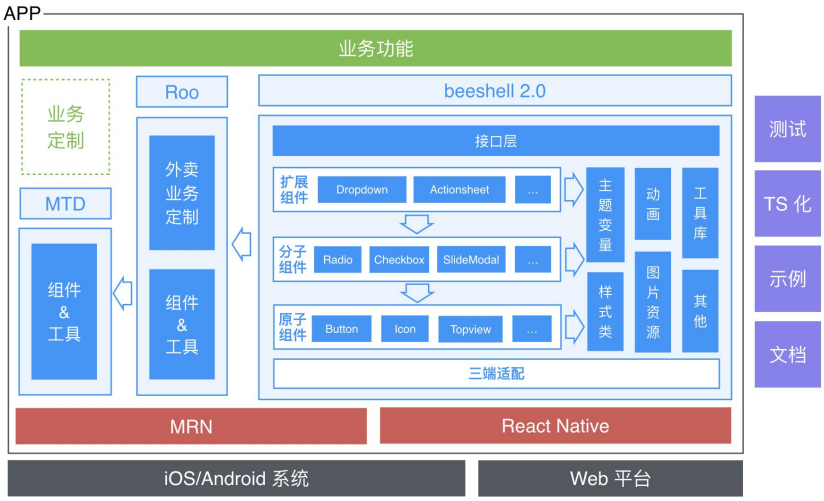
图 H

我们的目标是提供一套通用、易用、稳定、灵活的组件。然而，对一个组件来说，如果通用性、灵活性强，则易用性、稳定性势必较差。如何合理的处理这个矛盾呢?

为解决这个矛盾，我们使用了“关注度分离”的设计原则：首先将组件库需要支持的功能与特性进行分解；进而仔细研究特性的不同侧面（关注点），并提供相应的解决方案；最后整合各独立方案，解决冲突部分，形成整体的解决方案。

“关注度分离”的设计原则，贯穿整个组件库的设计与实现，是组件库的核心思想。以该原则为基础，衍生出了项目拆分、组件分层、功能解耦等具体方案，实现了一个组件库体系，保证了可以兼顾到相互矛盾的两个方面，实现最终目标。

架构升级


APP
业务功能
业务定制
Roo
beeshell 2.0
外卖业务定制
接口层
扩展组件
Dropdown
Actionsheet
...
主题变量
动画
工具库
MTD
分子组件
Radio
Checkbox
SlideModal
...
组件&工具
组件&工具
原子组件
Button
Icon
Topview
...
样式类
图片资源
其他
三端适配
测试
TS 化
示例
文档
MRN
React Native
iOS/Android 系统
Web 平台


beeshell 2.0
VS
beeshell 1.0

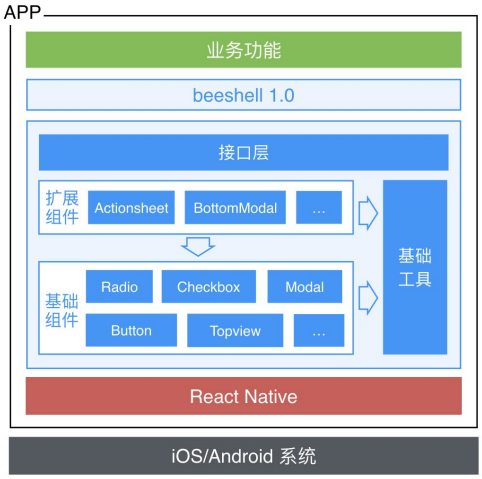

APP
业务功能
beeshell 1.0
接口层
扩展组件
Actionsheet
BottomModal
...
基础组件
Radio
Checkbox
Modal
Button
Topview
...
基础工具
React Native
iOS/Android 系统

图 F

beeshell 2.0 与 1.0 的架构在整体上保持一致，共分成四层：业务层、组件库（体系）、RN 层、系统层。而 beeshell 2.0 的架构升级，则主要体现在第二层与第三层：

第二层：组件库体系

组件库由 1.0 的一个项目演变成 2.0 的三个项目（版本），形成组件库体系。具体包含：MTD 公司通用版本、Roo 外卖定制版本和 beeshell 开源版本。

这种项目拆分的方式，符合“关注度分离”的设计原则，三个版本有各自不同的关注点：

- MTD 的关注点是通用性、灵活性，所以提供的是基础、通用的组件。组件的扩展能力极强，可以满足多个业务方的定制化需求。
- Roo 是对 MTD 的继承与扩展，定制了外卖业务的 UI 风格与功能，通用性减弱，功能性和业务性增强，关注易用性、业务性、一致性。
- beeshell（准确说是 2.0 版本）是对 Roo 的继承与扩展，与 Roo 相比，去除了过于业务化的组件与功能，纳入并整合社区的需求，关注通用性、易用性、稳定性。

组件库体系的三个版本，内部的架构设计一致，本文只详细介绍 beeshell 开源版本。

组件库使用了分层的架构风格，分成了接口层、组件层、工具层以及三端适配：

- 接口层。汇总所有组件，统一输出，使用全部引入的方式，方便组件使用。另外，为了避免引入过多无用组件，引起资源包过大，也支持组件的按需引入。
- 组件层。细分为原子、分子、扩展组件。 与 beeshell 1.0 相比，我们对组件在更细的粒度上进行拆分。同时，层次划分也更加精细、明确。如上图 F 所示：基础组件细分为分子、原子组件。这样，组件的继承、组合方式更加灵活，能够最大化代码复用。而且，原子、分子、扩展组件，各层次组件各司其职，“关注度分离”，兼顾通用性和易用性。

- 工具层。与 beeshell 1.0 相比，工具更加完备。不但新增了树形结构处理器、校验器等；而且工具的独立性更强，与组件完全解耦，与组件配合实现功能。这样，一方面，使得组件实现更加简洁，提升了组件的可维护性；另一方面，可以将工具提供给用户，方便用户开发，提升组件库的功能性、易用性。而且，工具与组件的解耦遵循“关注度分离”的原则。
- 三端适配。RN 在整体上实现了跨平台，iOS、Android、Web 三端使用一套代码，但是在一些细节方面，例如：特殊 API 的支持、相对位置计算等，各个平台有较大差异。为了更好的支持三端，保证跨平台稳定性，还需要在这一层做很多适配工作。

第三层：RN 层

这一层新增了 MRN，MRN 是对 RN 的二次封装，与 RN 底层实现保持一致。组件库在两个平台的表现一致。

MRN 是基于开源的 RN 框架改造并完善而成的一套动态化方案。MRN 从三个方面弥补 RN 的不足：

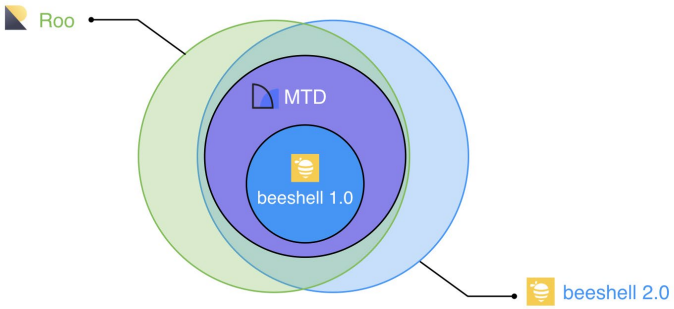
- 动态化能力。RN 本身不支持热更新，MRN 借助公司内发布平台，实现包的上传发布、下载更新、下线回滚等操作。
- 双端（未来三端）复用问题。从设计原则上保证开发者对平台的差异无明显感知。
- 保障措施。在开发保障方面：提供脚手架工具、模版工程、开发文档等，方便开发者日常的 MRN 开发工作。提供多级分包机制，业务内部和业务之间能够灵活组织代码；在运行保障方面，提供运行环境隔离，使得不同业务之间页面运行无干扰。提供数据采集和指标大盘，及时发现运行中的问题，同时提供降级能力以应对紧急情况。

除此之外，整个组件库体系，还具备以下特点：更加完善的测试方案；丰富的代码示例；使用 TypeScript（以下简称 TS）语言进行开发，充分利用 TS 的类型定义与检查；针对每个组件都有详细的文档说明。

协作模式

这里通过结构和流程两个方面，详细介绍 beeshell 1.0 与 beeshell 2.0 以及 MTD、Roo 的关系。三个版本之间通过 Git Fork 建立依赖关系，使用源码依赖的方式实现项目拆分。对于用户而言，不同版本的相同组件，底层依赖与实现都是一致的。

结构



流程

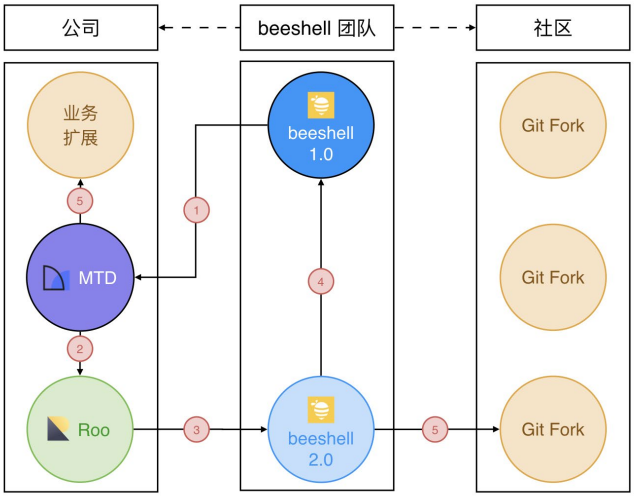


图 X

结构方面：MTD 包含 beeshell 1.0 的全部内容，并进行了组件数量的扩充、组件功能的增强；Roo 包含 MTD，进行了定制与业务扩展；beeshell 2.0 与 Roo 基本一致，去除业务部分，纳入社区需求。

流程方面：

- 首先，我们在 beeshell 1.0 的开发以及开源中，积累了丰富的经验。在建设 MTD 公司通用版组件库时，贡献了 50% 的组件；同时，贡献了许多设计模式与思路，大大加速了组件库的建设。
- 其次，在 MTD 建设完成后，为了更加方便、快速的接入外卖的相关业务，以 MTD 为基础，定制了外卖主题的组件库 Roo，提升了组件库的业务功能性和易用性。外卖相关的业务项目，在接入 Roo 后直接使用，无需再进行主题的定制与调整，在一定程度上节省开发成本。
- 第三，我们将共建的成果贡献出来，以 Roo 为基础，升级 beeshell 2.0 并开源。将部分过于业务化的组件移除，纳入了社区的相关需求，保证组件库的通用性、易用性与稳定性。
- 最后，对于公司内部，各个业务方可以以 MTD 为基础进行扩展，定制自己的业务主题组件库（Roo 就是第一个业务扩展）；对于社区，各个开发者可以根据实际的业务需求，以 beeshell 为基础，定制扩展组件库。

综上所述，我们以 beeshell 开发团队为桥梁，建立了美团公司与开源社区之间进行技术交流的通道，美团公司、beeshell 团队以及社区，可以在技术上互帮互助，共同建设、进步。

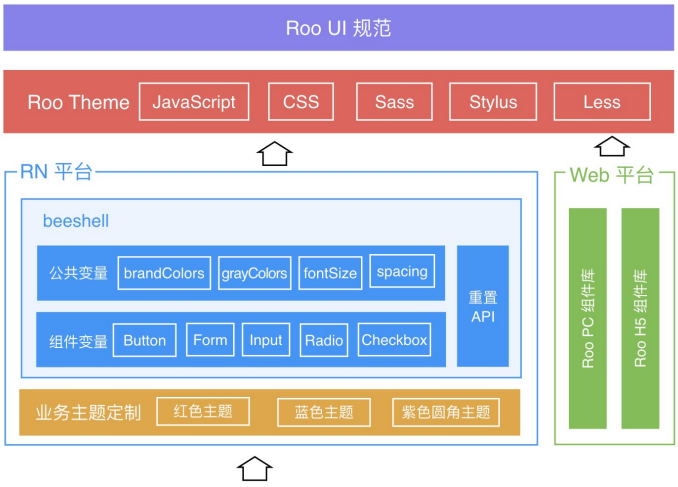
方案实现

UI 风格一致性

UI 风格一致性的重要性在于，对内可以保持平台统一性、提升团队效率、打磨细节体验；对外可以塑造品牌形象、减轻用户学习成本、保持产品的体验一致性。

UI 风格一致性的关键要素有很多，包括色彩、排版、字体、图标以及交互操作等。可以归纳为两类：样式一致性和动效一致性。

beeshell 延用了 Roo（袋鼠 UI）的 UI 设计规范，其内容涵盖了 PC 端与移动端、Web 平台与 RN 平台，对 UI 与交互给出了详细的视觉规范，旨在保证外卖事业部，全部产品的 UI 一致性。UI 规范的技术实现方式如下：



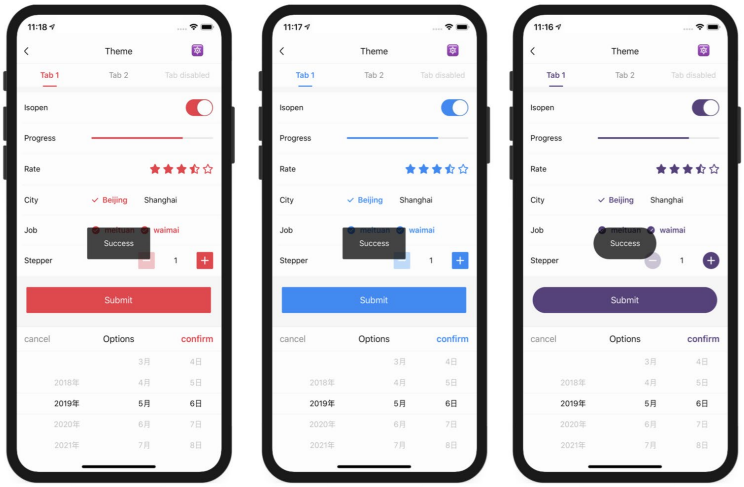


图 E

Roo Theme 向上实现了 UI 规范具体内容，将设计规范统一收敛，向下输出主题变量、组件样式类、通用样式工具等，供各个组件库以及业务方使用。

Roo Theme 主要使用 SasS 预处理器实现，提供了各个组件的统一样式类。为了满足不同技术栈的需求，主题变量的输出提供了 JavaScript（以下简称 JS）、CSS、Less、SasS、Stylus 多种语言，不同平台、技术栈可以根据情况自由选择。

beeshell 基于 Roo Theme 输出的 JS 主题变量，通过样式和动效两个方面，保证了 UI 一致性。

- 样式一致性。通过继承、扩展 Roo Theme，定义全局性的主题变量，用于组件的样式部分定义。主题变量范围涉及品牌色、灰度、字体尺寸、间距以及组件级变量，为组件以及组件之间样式一致性，提供了全面的保障。同时，组件库提供了自定义主题变量的接口，可以重置相关变量的值，对 UI 风格进行一致性调整，实现一键换肤。另外，使用“内部样式 < 主题 < 扩展样式”的样式优先级覆盖策略，保证了样式部分的定制能力（在下文“定制化能力分级设计”章节中详细介绍）。
- 动效一致性。一方面，依赖主题变量中定义的动画开关变量（主要考虑到一些低端 Android 机器的性能问题），用户可以关闭某个组件的动画；另一方面，依赖组件库的良好分层设计，我们将动画类独立实现，这样可以使用策略模式，将动画方便的集成到任意组件中。

下文详细介绍样式一致性和动效一致性：

样式一致性

样式一致性，可以从色彩和排版两个方面来保证。

首先，介绍下色彩部分。在 App 应用中，色彩元素扮演的角色仅次于功能。人与计算机的互动，主要是与图形用户界面的交互，而色彩在该交互中起着关键作用。它可以帮助用户查看和理解界面内容，与正确的元素互动，并了解相关操作。每个 App 都会有一套配色方案，并在主要区域使用其基础色彩。

正因为有无数种色彩组合的可能，在设计一个 App 时，人们的配色方案也有无数种选择。本文不纠结于如何选择一个好的配色方案，而是介绍一个配色方案应该具有哪些元素。一套完整的配色方案，应该包括品牌主色、品牌功能色、中性色。本文以 beeshell 的配色方案举例说明。

色彩：品牌主色

品牌主色应该是应用中出现最频繁的颜色，通常用来强调 UI 中关键部分的颜色。beeshell 的品牌主色色值为 #ffd800，如下图所示：

color-1	color-2	color-3	color-4	color-5	color-6	color-7	color-8	color-9	color-10
#ffffe6	#fffda3	#fff87a	#fff152	#ffe629	#ffd800	#d9b100	#b38c00	#8c6900	#664900

通常，一个产品的 UI 只会有一个品牌主色。然而，像 beeshell 这种品牌主色色值较浅的情况，一个品牌主色并不能够支撑所有的应用场景。此时，可以通过加深主色的方式，再增加几个色值，beeshell 的品牌主色还包括一个加深的色值 #ffa000，用于某些组件的激活状态，如下图所示：

color-1	color-2	color-3	color-4	color-5	color-6	color-7	color-8	color-9	color-10
#fffae6	#ffeaa3	#ffdc7a	#ffcb52	#ffb829	#ffa000	#d98200	#b36500	#8c4b00	#663300

对于品牌主色的个数，需要根据色值的情况而定，不必过于拘泥规则，只要能有一致性的用户体验即可。

色彩：品牌功能色

beeshell 的功能色内容与使用场景如下图所示：

Info 一般信息	#188afa
Success 成功	#61cb28
Warning 警告	#ff8400
Danger 危险	#f23244

图 U

这四个色值，分别用于一般信息、成功、警告、失败这四种业务场景，以及从这四种业务场景所衍生出的场景。在一定程度上，保证色彩的一致性。

色彩：中性色

beeshell 的中性色（灰度）的内容与使用场景如下图所示：

GrayBase 标题、正文	GrayDarker 副标题	GrayDark 补充信息	Gray 取消、禁用	GrayLight —	GrayLighter —	GrayLightest —
#111111	#333333	#555555	#888888	#aaaaaa	#cccccc	#ebebeb

图 G

中性色应用于界面主体文本的颜色，灰度范围的确定，可以大大提升色彩的一致性。

接下来介绍下排版，具体可以分为字体、间距、边线。

排版：字体

beeshell 的字体尺寸（Font Size）集，是基于 12、14、16、20 和 28 的排版比例，如下图所示：



图 B

这样的排版比例，可以使得界面的文字内容更加协调、流畅，进而提升了排版的一致性。

对于字重（Font Weight），beeshell 只使用正常（Normal）和加粗（Bold）两种：Normal 用于一般文本；Bold 用于强调，吸引用户注意力。只使用这两种字重，也避免了因为不同字体家族（Font Family），对字重的支持范围不同，而导致视觉差异。

除了字体尺寸和字重，影响排版的还有字体行高（Line Height）。为了达到适当的可读性和阅读流畅性，可以根据字体的大小和粗细，设定字体行高。默认情况下，RN 应用：行高 = 字体大小 * 1.2。如下图所示：



图 L

beeshell 使用了默认的字体行高，在一定程度保证了可读性和排版的一致性。

排版：间距

间距是 UI 元素与元素之间、父元素与子元素之间的空白区域，一个应用排版风格一致性，很大程度取决于间距。一个组件的最终宽高，应该由内容、内边距以及边框决定，是自适应的，而不应该直接定义宽高。

对于同一个 App，间距应该在一个合适的范围取值，可以通过定义『小号间距』、『中号间距』、『大号间距』等来划分信息层次。例如 beeshell 的 Button 组件，有三种尺寸。实现效果如下图所示：

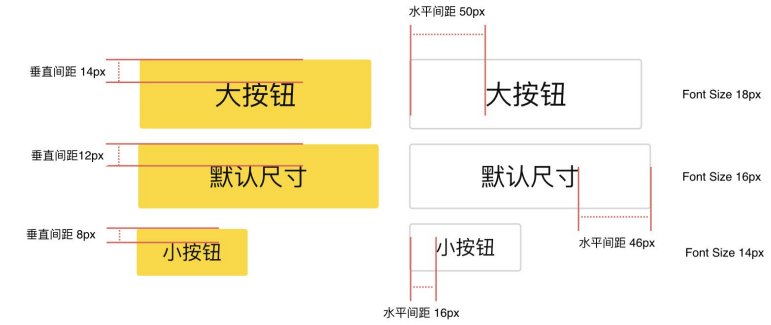


图 S

排版：边线

边线（边框）部分，需要统一元素的边框宽度、颜色和圆角，边线虽然对 UI 风格的影响较小，但是不可或缺。beeshell 使用的边框宽度为一个物理像素，使用 RN 提供的 StyleSheet.hairlineWidth 接口实现；定义了三种灰度的边框颜色；主要使用 2px 的圆角。

最后，样式的一致性，还涉及到图标、布局等相关内容，因为 beeshell 对这些内容的涉及较少，这里不做详细介绍。

动效一致性

动效展示了应用的组织方式和功能。

动效可以：

- 引导用户在视图中的视觉焦点。
- 提示用户完成手势操作后会发生什么。
- 暗示元素间的等级和空间关系。
- 让用户忽视系统背后发生的事情（比如抓取内容、或加载下一个视图）。
- 使应用更有个性、更优雅、体验更加一致。

beeshell 组件库基于 Animated 进行了二次封装，提供 FadeAnimated 和 SlideAnimated 两个动画类，支持淡入淡出动画和滑动动画，可以使用策略模式集成到任何组件中。

beeshell 将逐渐在所有的组件集成这两种动画，保证动效的一致性。下文展示已经实现了动画的组件，先睹为快。

Button 组件使用 FadeAnimated 类实现动画，动效如下图所示：



Modal 组件使用 FadeAnimated 类实现动画，动效如下图所示：



Dropdown 组件使用 SlideAnimated 类实现动画，动效如下图所示：



定制化能力分级设计

要开发一套全公司共用的组件，需要同时满足酒旅、外卖 C 端、外卖 B 端以及外卖 M 端等业务的需求，这对组件的定制化能力提出了更高的要求。

为了进一步增强组件的定制化能力，提升组件的通用性、灵活性；同时，避免属性（API）的无节制增加，进而导致组件难以维护，我们设计了定制化能力分级的策略。

这里以一个常见的业务场景：底部上滑弹框，来举例说明分级设计。



图 M

如上图所示：第一个例子比较通用、规范。“区域文字内容”的文案与样式需要支持自定义；第二个例子，需要支持多行文字以及图标，即“区域内容”需要支持自定义；第三个例子，自定义的重点，由区域以及区域内部，转移到区域之间的布局，“区域布局”需要支持自定义。

区域文字内容、区域内容、区域布局，三个层面的灵活性，可以涵盖一个业务场景所有定制化需求了。以当前业务场景为例，来讲解如何使用定制化能力分级设计，完成全部的定制化需求。

首先实现了一个 BottomModal 底部弹框组件，然后开始进行定制化设计。

第一级定制化：定制主题变量

“完成”文本的颜色，使用的是主题变量定义的品牌主色（Brand Primary Dark），beeshell 默认的品牌主色为黄色。

通过组件库提供的自定义主题变量接口，可以修改品牌主色的色值，进而修改了“完成”文本的颜色。同理，可修改“取消”、“标题”的颜色。

修改品牌主色，影响范围很大，所有组件的色彩风格统一变化。如果我只想把文本的颜色改成红色，但是并不想修改品牌主色，应该如何定制呢？可以使用第二级定制化。

第二级定制化：提供定制属性

这里提供 LabelText（类型为 String）和 LabelTextStyle（类型为 TextStyle）属性，如下图所示：



图 C-1

代码实现为：

```
<Text style={this.props.labelTextStyle}>{this.props.labelText || '完成
'}</Text>
```

LabelText 用于定制文案内容，将 LabelTextStyle 整体暴露出来，而不是只暴

露颜色单个属性，这样处有两点好处：

- 开发者都熟悉 Style 这个名称与用法，但并不知道 xxxColor 是什么，组件更加易用。
- Style 不仅可以定制 Color，还支持 fontSize、fontWeight 等属性，定制能力更强。

以上两级主要是样式部分的定制，使用了样式优先级覆盖的策略，扩展样式（labelTextStyle）覆盖主题，主题覆盖内部样式。

下一步，就是对于多行文字、图标的支持。



第三级定制化：开放渲染区域

提供 Label 属性，类型为 ReactElement 对象，任意定制 UI，如下图所示：



图 C-2

到这里，区域以及区域内部的定制化需求，就都可以满足了。但是区域布局的定制化，因为布局情况太多，并不容易实现。



如果再提供几个属性，用于定制布局方式、头部边框样式、底部按钮，按照这种方式，属性会无节制增加，势必造成组件难以维护，易用性也会大打折扣。那应该如何实现？我们设计了第四级。

第四级定制化：继承 / 组合基类

在 beeshell 1.0 的开源推广文章中也有讲到过，我们在组件库开发之初，对常见组件，进行了全面的梳理。在比较细的粒度，对组件进行拆分，以继承的方式，层层依赖，以功能渐进式增强的方式，实现各个组件。

这样使得开发者，可以在任意层级上继承、组合组件，进行定制化开发，提供了极强的扩展能力。具体实现如下：

首先，组件库实现一个 SlideModal 滑动弹框组件。这是一个比较底层、基础的组件，功能相对少，支持多个方向的滑动动画，内容完全由开发者自定义，通用性、定制化能力极强。实现效果如下：



然后，组件库实现了 BottomModal 组件，继承 SlideModal，固定滑动的方向和开始位置，弹框内容横向拉伸至全屏、纵向自适应，功能增强而定制化能力减弱。实现效果如下：



前文已经讲到，产品需求已经超出了 BottomModal 定制化的能力，强行实现只会带来不良后果。所以，我的方式是组合使用 SlideModal，开发一个新的组件，也就是第四级定制化。新组件的实现效果如下：



第四级定制化，是使用了新的思路，不再盲目的增加一个组件的功能，来帮助开发者满足产品需求，而是提供了基础工具。基础工具实现了底层、复杂的部分，表现层的部分则让渡给开发者，用户自己实现，“授人以鱼不如授人以渔”。

对比业界的开源 RN 组件库，也没有几个可以达到第四级的定制化能力。

beeshell 通过四个级别的定制化的能力，可以轻松搞定所有的产品的需求。



总之，定制化能力分级设计，是对定制化需求进行分类，针对每一类的定制化需求，设计了相应的策略，最终合成一个完整的解决方案，符合“关注度分离”的设计原则。

功能丰富

功能丰富旨在支持多种业务场景，支持特定业务场景的多种情况，进而提升组件库的易用性。功能丰富通过两个方面保证：组件丰富度、单个组件的功能丰富度。

组件丰富度

为了保证组件丰富度，我们通过多种渠道（既有组件整理、业务组件提取、参考标杆项目）来收集组件。最终规划了包括通用类、导航类、数据录入类、数据展示类、操作反馈类、基础工具在内的 6 个大类，共 50 多个功能，旨在覆盖尽可能多的业务场景。

目前，beeshell 已经实现了 38 个功能，与业界标杆项目的对比情况如下图：

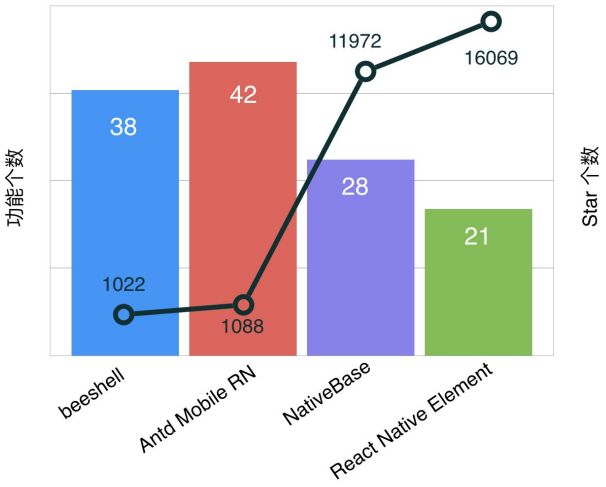


图 K

虽然，beeshell 的组件数量还比不上 Antd Mobile RN（用不了多久也会超过），但已经超过 NativeBase 和 React Native Element。beeshell 在组件数量上有很大优势，可以支持更多的业务场景，且支持全部引入和按需引入，用户无需担心打包过多无用代码的问题。

功能丰富度

功能丰富度针对的是单个组件所支持的功能，旨在覆盖特定业务场景的多种情况。

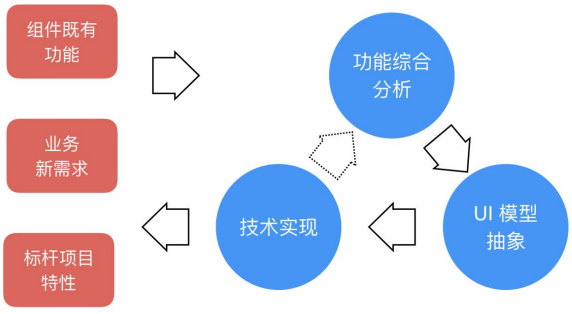


图 O

对于一个组件的功能丰富度，我们通过多方式收集、功能综合分析、UI 模型抽象、技术实现、验证反馈几个步骤来保证。

- 多方式收集。通过多种方式来收集组件功能，收集方式包括：组件既有功能、业务新需求、标杆项目特性。
- 功能综合分析。对收集的功能进行全面、综合分析与考虑，得出组件需要支持的功能特性。
- UI 模型抽象。对组件功能进行抽象设计，根据 UI 模型，明确抽象设计的合理性和有效性。
- 技术实现。根据平台特性、技术选型来完成代码实现。
- 验证反馈。组件实现后，会应用到业务或者示例工程来验证，如果组件并不能满足需求，需要重复执行以上几步。

下文以 SlideModal 组件的实现，举例说明：

首先，通过多种方式，收集到的滑动弹框场景如下：

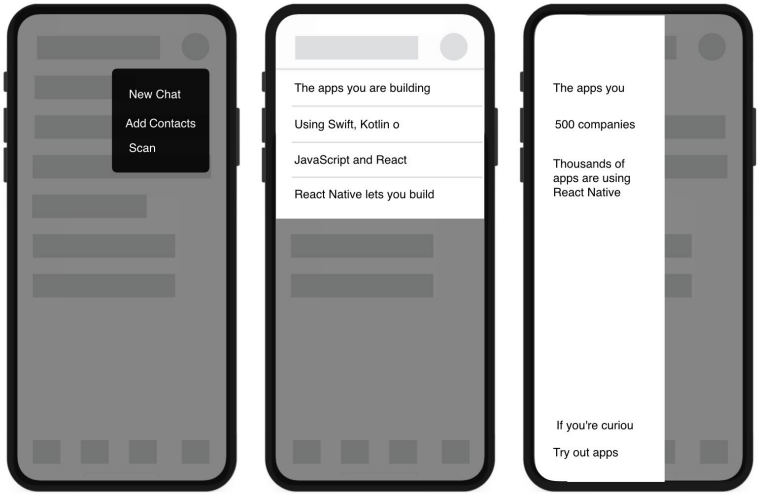


图 Q

其次，综合分析得出，SlideModal 组件需要支持的功能有：弹出位置自定义、滑动方向自定义、全屏 / 半屏自定义。

然后，进行 UI 模型抽象，抽象设计如下图所示：

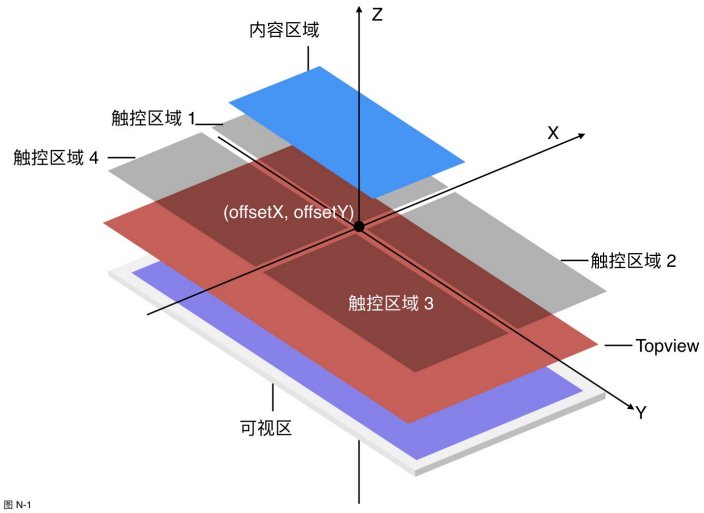


图 N-1

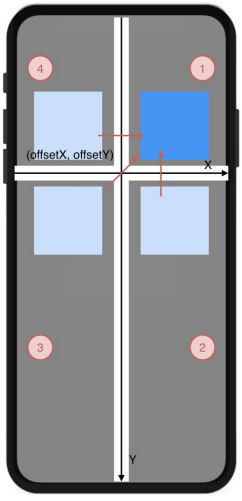


图 N-2

由 UI 模型得出，SlideModal 组件通过 (offsetX, offsetY) 坐标值来定义弹出位置；为了支持全屏 / 半屏效果，将屏幕分割为 4 个区域，分别自定义触控效果（阻断点击或者击穿）；弹框内容在一个区域展示，每个区域有 3 种滑动方向（图 N-2），共支持 12 种滑动动效。

最后，是基于 RN 平台的技术实现，并经过大量业务场景的验证反馈。Slide-

Modal 组件的最终实现效果如下：



对比业界开源 RN 组件库，针对滑动弹框场景，没有几个可以超过 SlideModal 的业务支持能力。

SlideModal 组件只提供底层支持，如果要应用到真实的业务场景，还需要基

于该组件进一步开发。beeshell 也提供了更高层次的定制组件，例如：Dropdown、Popover 等，可以直接使用。

除了 SlideModal 之外，还实现了其他功能强大的组件：Slider 滑块组件，支持纵向和横向滑动；Rate 评分组件，实现一套滑动评分的机制，支持定制任意 UI 元素。由于篇幅有限，在此不再赘述。

易用性提升

组件易用性的提升，通过命名、文档和示例这三个方面来保证。

命名

命名包括组件名、属性与方法名。

一个组件，实际上就是 Web 页面或者 App 中的元素、控件，通常因为原生控件的能力薄弱，而进行二次封装。所以组件名与原生控件名的名称，尽量保持一致。例如，Form 与 HTML Form 标签一致，Switch 从 iOS 控件 UISwitch 中得来。这样的命名，可以给与开发者更加直观的感受，通过名称就能知道组件大概的 UI 与功能，降低学习和使用的成本。

属性与方法的命名，既要考虑原生控件的属性名，又要考虑组件库命名的一致性。例如，表单录入的相关组件，包括 Input、Radio、Checkbox、Switch 等，组件的值要统一使用 Value 命名，值变化的回调使用 onChange，选中状态使用 Checked 布尔类型。这样符合用户的直观感受，更加易用，降低使用成本。

常用属性名举例如下：

属性名	类型	描述
Style	ViewStyle/TextStyle	组件样式，通常作为组件的第一个子节点的样式属性
Data	Any[]	数据源，数据源的元素通常是对象 { label: string, value: any, [props: string]: any } Label 作为展示文案，Value 作为元素唯一标志，以及其他属性
Value	Any	值
onChange	Function	值变化回调
onPress	Function	点击事件
renderItem	Function	自定义渲染项

文档

文档规定了统一的格式，旨在全方位介绍组件，方便开发者使用，格式内容如下：

- 组件名称。
- 组件描述。
- 引入方式，包括全部、按需两种引入方式。
- 示例演示，动图与静图。
- 示例代码，使用伪代码，言简意赅，能说明使用方式即可，同时，附有完整示例代码的链接。
- API 说明，分成 Props 和 Methods 两部分。
  - Props 包含 Name | Type | Required | Default | Description。
  - Methods 格式借鉴 RN 官方文档格式。

示例

beeshell 组件库遵循“关注度分离”的设计原则，对组件进行细粒度的拆分，进行了分类、分层，以及基础工具与组件实现的功能解耦。

这些设计虽然大大提升了组件库的灵活性，但是在一定程度上，对开发者提出了更高的要求。开发者需要理解各个组件与工具，灵活的组合各个元素，才能更好的完成业务需求。

为了方便开发者，更有效、合理的使用组件，我们将会实现一些经典的业务场景，以示例的形式开源出来。借助美团的平台服务，为用户提供在线演示的功能，用户可以下载美团 App（iOS 与 Android 都可以），扫描下图二维码，即可快速体验各种应用场景。



测试

代码开发的目标有两个：第一个是实现需求，第二个是保证研发质量。研发质量对公共组件库来说，尤为重要。测试是为了提高代码质量和可维护性，是实现代码开发第二个目标的一种方法。

beeshell 1.0 中已经集成了“黑盒测试”与“白盒测试”。beeshell 2.0 在原有的基础上，进行了一定程度的优化，代码的可靠性与安全性，仍然保持最高级别，而测试覆盖率则由原来的 70% 提升到了 80% 以上。

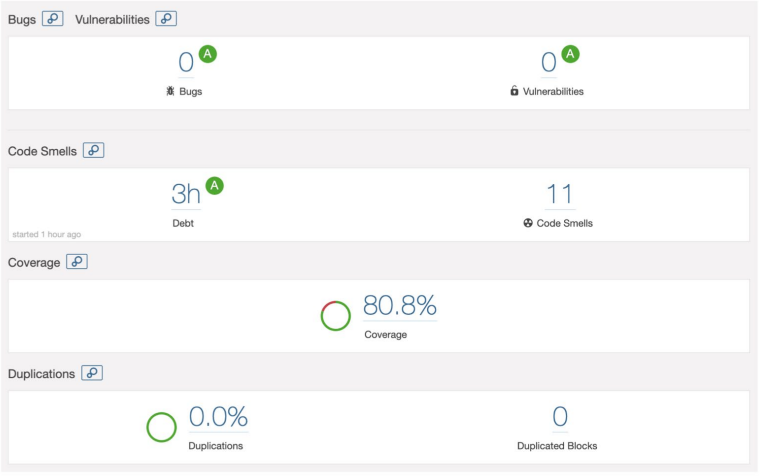
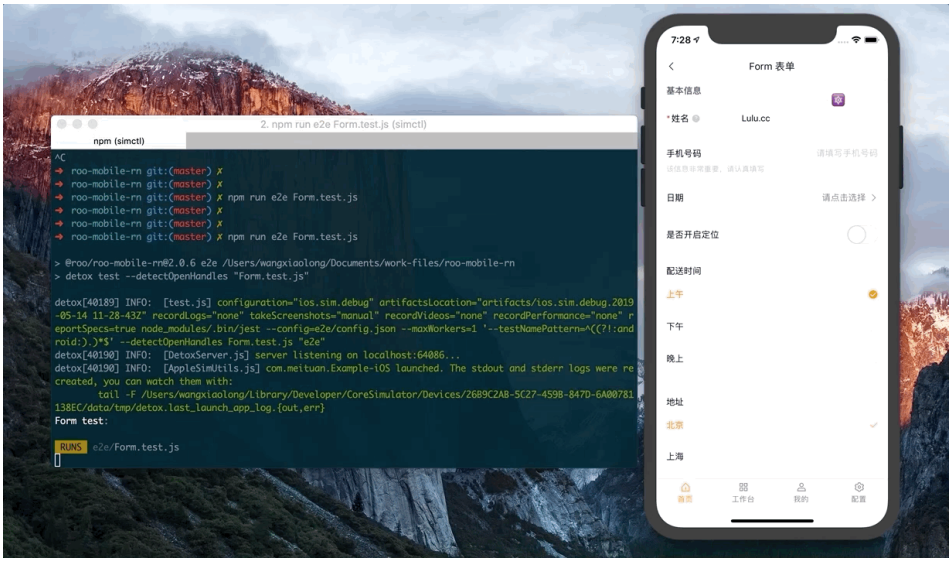


图 A

SonarQube 的分析统计结果

不仅如此，beeshell 2.0 在测试领域继续探索，集成了“灰盒测试”（基于开源方案 Detox 实现）。

> 灰盒测试，是介于白盒测试与黑盒测试之间的一种测试，灰盒测试多用于集成测试阶段，不仅关注输出、输入的正确性，同时也关注程序内部的情况。灰盒测试不像白盒那样详细、完整，但又比黑盒测试更关注程序的内部逻辑，常常是通过一些表征性的现象、事件、标志来判断内部的运行状态。



灰盒测试效果

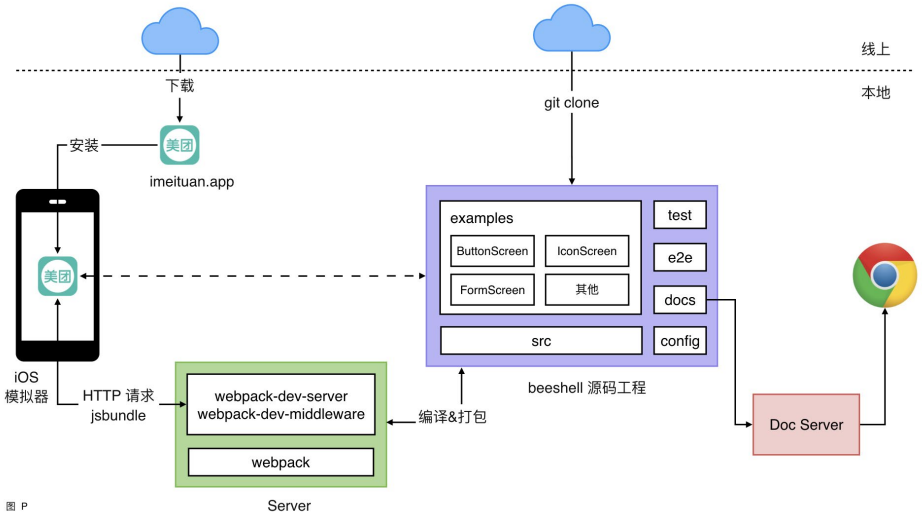
通过黑盒测试、白盒测试、灰盒测试，三种测试方案，更加全面的保证组件库的代码质量，大大提高了代码可维护性。

开发调试

受益于 MRN 平台，JS 代码与 Native 代码得以完全分离。

beeshell 源码工程，包含了包括组件源码、示例代码、测试文件在内的全部 JS 代码，Native 部分则只负责打包生成容器（本文以美团 APP 举例说明），通过下载并安装 .app（iOS）或者 .apk（Android）文件至模拟器，直接加载本地服务提供的 jsbundle，快速进入开发调试。

前端开发再也不用关心 Native 的部分，无需耗时耗力的维护 Native 环境、依赖，极大降低了前端开发成本。



开发调试流程

未来规划

我们的目标是把 beeshell 建设成为一个全面且完备的组件库，不仅会不断丰富 JS 组件，而且会不断加强复合组件去支持更多的底层功能。

我们为组件库发展规划了三个阶段：

- 第一阶段，beeshell 1.0 版本，开源 20+ 组件，主要提供基础功能。
- 第二阶段，对我们在开发 React Native 应用几年时间积累的组件进行整理，同时参考业界的标杆项目，开源 50+ 组件。
- 第三阶段，调研移动端 APP 常用的功能与场景，分析与整理，然后在 beeshell 中实现，开源 100+ 组件。

此次 beeshell 2.0 升级，共计开源 38 个功能，而且已经详细的规划了另外的 15+ 组件，也会在近期开源，目前处于第二阶段的收尾阶段。

第三阶段的建设，也在紧锣密鼓的筹备当中，要实现 100+ 组件任务十分艰巨，希望大家踊跃参与，共同建设。

开源相关

Github 地址

beeshell

核心贡献者

小龙，泽楠，轶超，宋鹏，孟谦

参考资料

- beeshell 1.0 开源推广文章
- MATERIAL DESIGN

团队介绍

外卖事业部终端团队，负责的多个终端和平台直接连接亿万用户、数百万商家和几万个运营人员，目标是在保障业务高稳定、高可用的同时，持续提升用户体验和研发效率。 – 在用户方向上，构建了全链路的高可用体系，客户端、Web 前端和小程序等多终端的可用性在 99% 左右；跨多端高复用的局部动态化框架在首页、广告、营销等核心路径的落地，提升了 30% 的研发效率；

– 在商家方向上，从提高进程优先级、VoIP Push 拉活、doze 等方面进行保活定制，并提供了 Shark、短链和 Push 等多条触达通道，订单到达率提升至 98% 以上；

– 在运营方向上，通过标准化研发流程、建设组件库和 Node 服务以及前端应用的管理与页面配置等提升 10% 的研发效率。

招聘信息

外卖事业部终端团队是由一群活力四射，对技术饱含热情，平均年龄不超过 26 岁的人共同组成。在这里，你可以看到大家对技术的追求，对产品的雕琢，对团队的认同，一切都是那么自然；在这里，你可以做一个纯粹的 FE，写写 JS，打磨一下 CSS；也可以做一个精致的猪猪女孩（男孩），优雅的调试 Android 和 iOS 代码。当然，你绝对可以做一个霸道总裁，肆意的拥抱跨端方案，Hybrid，Flutter，React Native 等，都是你的新战场。我们正在持续努力成为一个面向未来编程的团队，而这里还缺一个你。欢迎志同道合的同学发送简历到：tech@meituan.com（邮件标题注明：外卖事业部终端团队）。

React Native 在美团外卖客户端的实践

晓飞、唐笛、维康

MRN 简介

MRN（Meituan React Native）是基于开源的 React Native 框架改造并完善而成的一套动态化方案，在开发体验上基本能与原生 RN 保持一致，同时从业务需求的角度满足从开发、构建、测试、部署、运维的工程化需要。解决了一系列痛点问题：客户端版本审核及更新效率低、Android/iOS/Web 三端开发技术方案不一致、公共需求重复劳动、需求排期不敏捷、集成成本高等。目前 MRN 已接入美团数十个 App，在核心框架及生态工具上有超过百位代码贡献者，（每天）的总 PV 超过 1 亿次。

在项目成立之初，MRN 使用当时最新的 React Native 0.54.3 作为基础版本，然后进行了一系列的改造。React Native 官方稳定版已经升至 0.60.5，对 MRN 页面的质量性能、开发者体验都有了巨大的提升，包括 JSI 替换桥进行 JS 和 Native 通信、JS 引擎替换、React Hooks 等功能。最近，MRN 也做了一些升级适配和深度优化，在相关基础建设、融合过程、优化手段等方面，我们进行了很多的探索和思考，后续这些内容会陆续放出，希望能给大家一些启发。本文主要分享美团外卖 App 在业务实践和技术探索过程中的经验。

背景

美团外卖自 2013 年创建以来，一直处于高速发展期。美团外卖所承载的业务，也从单一的餐饮业务，发展到餐饮、超市、生鲜、果蔬、药品、鲜花、蛋糕、跑腿等十多个大品类业务。伴随着业务的快速发展，我们深切地感受到 3 个痛点：

（1）业务要求快速发版试错和原生迭代周期长

美团外卖业务长期使用 H5+Native 的技术栈。由于原生应用需要依托于应用市场进行更新，这样的话，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

周期变得很长，无法实现快速发版快速试错的诉求。H5 虽然具备随时发布的能力，但受限于内核的影响，平台兼容性并不好，性能也较差，而且调用 Native 的能力也受限，往往只能满足一定范围内的产品需求。

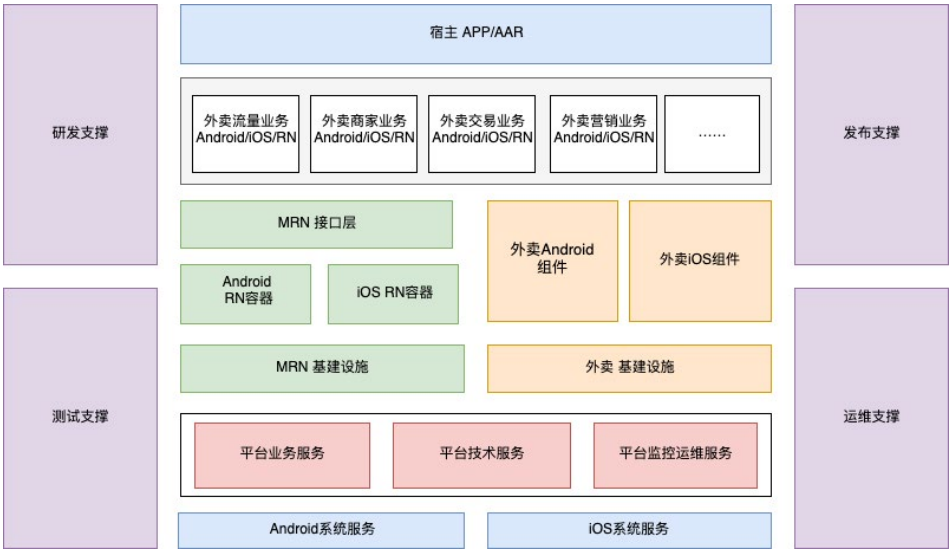
（2）有限的客户端研发资源无法满足日益增长的业务

业务的快速发展对客户端的开发效率不断提出挑战。如何通过技术手段，在有限的客户端人力资源下，支持更多的业务需求，解决有限的研发资源跟不断变大的业务需求量之间的矛盾呢？试想，如果能逐渐地磨平 Android 和 iOS 开发技术栈带来的问题，支持一套代码在 2 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。

（3）业务持续增长带来的安装包的大幅增长

业务的快速迭代，功能的持续增加，美团外卖客户端安装包也在持续增加，从 2018 年到 2019 年，已经累计增长 140%。如果没有切实有效的技术手段，安装包将变得越发臃肿。 业务层面的这些痛点，也在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。2015 年，Facebook 发布了非常具有颠覆性的 React Native（简称 RN）框架。从名字上就可以看出，这属于一种混合式开发的模式。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。作为一种跨平台的移动应用开发框架，RN 的特性非常符合我们的诉求。我们也在一直积极地探索 RN 相关的技术，并且基于 RN 在脚手架、组件库、预加载、分包构建、发布运维等多个维度进行了全面的定制及优化，大幅提升了 RN 的开发及发布运维效率，还打造了更适应于美团的 MRN 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 框架来解决业务层面的一系列问题。经过一年多的实践，我们积累了一些经验和结论，希望相关的经验和结论能够帮助到更多的个人或技术团队。

外卖混合式架构



上图是外卖 App 引入 MRN 后的架构全景图，接下来我们会从下到上、从左到右逐步介绍：

- 最下层是 Android/iOS 系统服务层，因为 MRN 是跨端的，所以需要引入这一层。相对单一平台来说，由于 MRN 的引入，整个 App 的架构不可避免地需要考虑 Android 和 iOS 平台本身的差异性。
- 倒数第二层是平台服务层，这一层相对与单一平台来说，并没有太大区别。
- 再往上一层是 MRN 基建层，这一层的工作主要是：(1) 尽可能地屏蔽 Android 和 iOS 系统的差异性；(2) 打通已有的平台基建能力，让上层业务不能感知到差异。
- 再上一层是业务组件层，这一层相对于单一平台来说，区别不大，主要是增加了 Android 和 iOS 的 RN 容器，同时业务组件是可以被 RN 调用的。
- 继续往上是 MRN 接口层，该层的主要任务是尽可能地屏蔽 Android 和 iOS 组件之间的差异，让上层页面使用的 RN 接口保持一致。
- 最后是业务层，这一层是用户可直接接触到的页面，页面的实现可以是

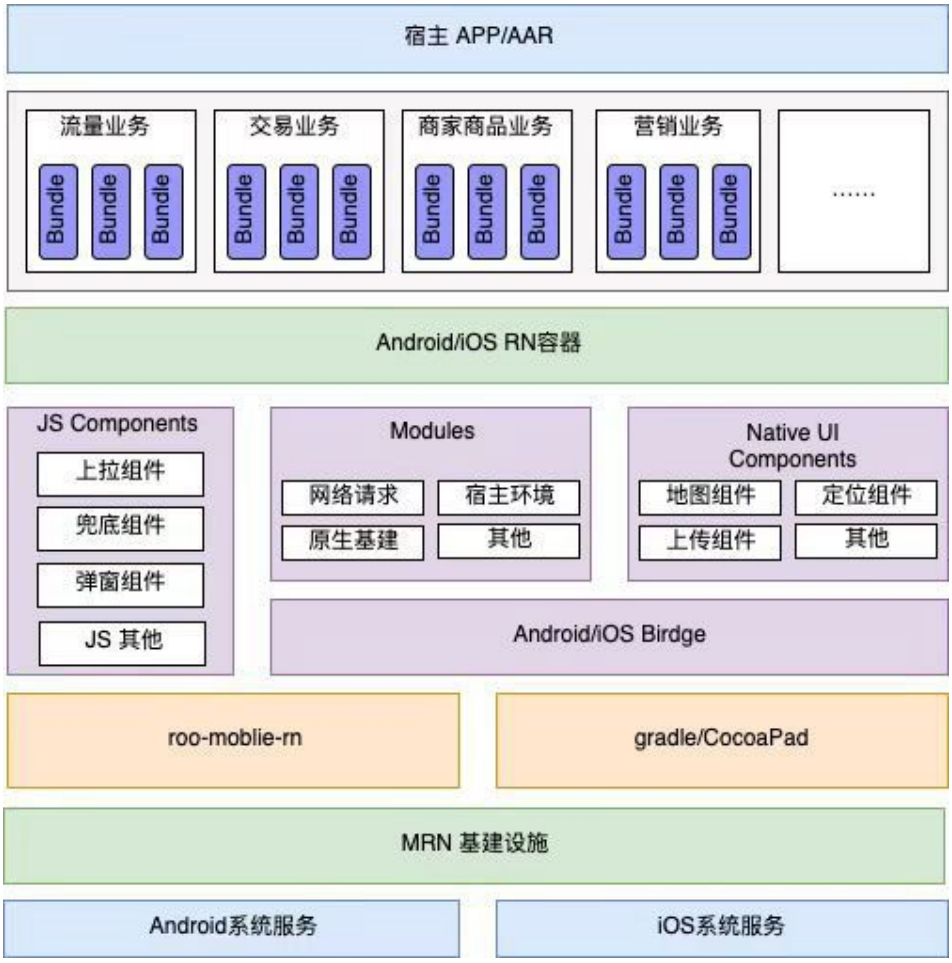
Android/iOS/RN。

- 左上角是研发支撑，主要包括代码规范、代码检查工具、Debug 插件、准入规范、准入检查工具、代码模板插件等。这块相对于单一平台来说，主要的差异体现在：由于编译器和语言不同，使用的工具有所区别，但工具要做的事情基本是一致的。
- 左下角是测试支撑，主要包括 UI 自动化测试、自测覆盖率检查、AppMock 工具、业务自测小助手、性能测试、云测平台等。这块相对于单一平台来说，基本也是一致的，主要的差异同研发支撑，主要是语言不同，使用的工具有所区别。
- 右上角是发布支撑，主要包括打包 Bundle 和 APK、打包检查、发布检查、发布 Bundle 和 APK 等。这块相对于单一平台来说，保持了打包发布平台的一致性，区别在于：需在原有的基础上，增加 MRN 的打包发布环节。
- 右下角是运维支撑，主要包括基建成功率监控、业务成功率监控、线上问题追踪、网络降级等。这块相对于单一平台来说，保持了一致性，区别在于：需在原有的基础上，增加 MRN 的监控运维。

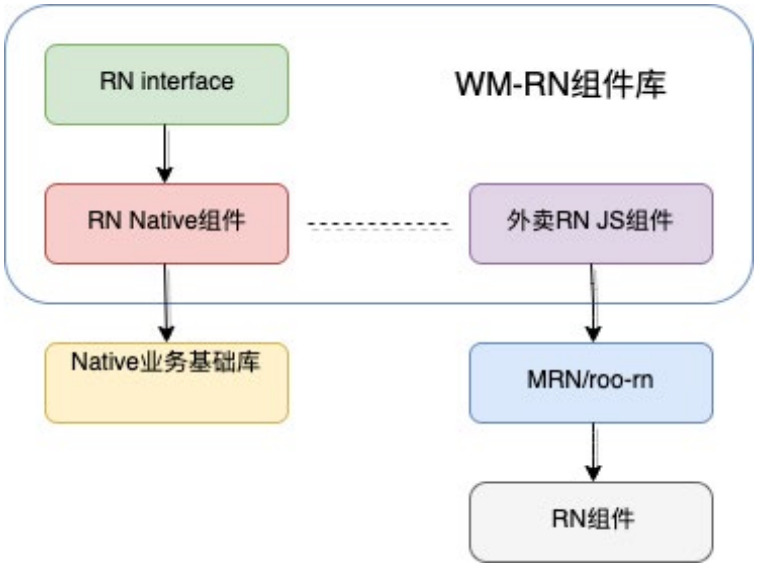
研发测试支撑

外卖业务 MRN 组件架构

RN 官方对双端只提供了 30 多个常用组件，与成熟的 Native 开发相比，天壤之别。所以我们在开发的过程中面临的一个很重要问题就是组件的缺失。于是，MRN 团队基于 RN 组件进行了丰富，引入了一些优秀的开源组件，但是源于外卖业务的特殊性，一方面需要业务定制，另一方面部分组件依然缺失。所以为了减少重复代码，提升外卖客户端 MRN 的研发效率，建设外卖组件库就变得非常有必要。



上图是我们外卖组件库的架构图，最底层依赖 Android 和 iOS 的原生服务；然后是 MRN 基建层，用于抹平 Android 和 iOS 系统之间的差异；再上一层则是外卖组件库及其依赖，如平台组件库和打包服务，组件库分为两类：纯 JS 组件和包含 JS 和 Native 的复合组件。再上一层则是 Android 和 iOS 的 MRN 容器，它提供了上层 Bundle 的运行环境。整个组件的架构思路，是利用中间层来屏蔽平台的差异，尽可能地使用 JS 组件，减少对原生组件的依赖。这样可以有效地减少上层业务开发时对平台的理解。接下来，我们主要讲一下 WM-RN 组件库：



如上图所示，WM-RN 组件库主要包含三部分：RN interface、RN Native 组件、外卖 RN JS 组件。RN Interface 主要包括 Native 组件的 Bridge 部分和 Native 组件在 JS 侧的封装，封装一层的好处是方便调用 Native 暴露出的接口，也可以用来抹平 Android 和 iOS 系统间的差异；RN Native 组件分为 Android 和 iOS 两端，依赖各自的业务模块，为 RN 提供外卖 Native 的业务能力，如购物车服务、广告服务；外卖 RN JS 组件则是纯 JS 实现，内部兼容外卖 App 与美团外卖频道间的差异、Android 和 iOS 平台间的差异，依赖现有的 MRN 组件库和外卖开源 Beeshell 组件库，减少组件的开发成本；从工程的物理结构来看，建议将 Native 组件、RN Interface 放在一个仓库进行管理，主要是因为 Native 与 JS 侧的很多通信都是通过字符串来匹配的，放在一起方便双端与 JS 侧的接口统一对齐，发布时也会更加方便。目前，外卖组件库已经扩展了几十个业务组件，支持了线上近百个 MRN 页面。

Native/MRN/H5 选型标准

目前，美团外卖 App 存在三种技术栈：Native、MRN、H5，面对业务持续增长和安装包不断变大的压力，选择合适的技术栈显得尤为重要。H5 在性能和用户体验方面相比 Native 和基于 Native 渲染的 RN 相对弱一些，所以目前大部分 H5 页面只

是用来承载需求变更频繁、需要即时上线的活动页面。那么 MRN 和 Native 的界限是什么呢？当有一个新的页面产生时，我们应该如何做取舍？通过实践，我们逐渐摸索了一套选型规则，如下：

- Native 选型规则，强交互（同时存在 2 种及以上手势操作），无法用二元函数描述的复杂动效，对用户体验要求极致的页面，类似首页、点菜页、提单页等。
- 对于强交互或强动画，MRN 技术栈支持效果不理想，不建议使用。其他情况下，建议使用 MRN。
- H5 适用于需要外链展示的轻展示页面，比如向外投放活动的运营页面等等。

具体选型细节可参考下表：

需求类型	细类	举例	Native	MRN	H5
展示类	静态展示类	列表、纯展示	不推荐	推荐	不推荐
	简单交互页	交互简单，业务复杂	不推荐	推荐	不推荐
	千人千面		不推荐	推荐	不推荐
交互类	复杂交互功能（可以RN配合Native）	键盘、嵌套滚动、频繁精细滑动、滑动冲突	推荐	不推荐	不推荐
	一般动画	轮播图、弹窗、进度条等	不推荐	推荐	不推荐
	精细动画	下拉刷新动画、加载动画	推荐	不推荐	不支持
组件类	通用业务类组件	商家相册	不推荐	推荐	不推荐
	页面内多个Native组件	公用部分不多，大部分都是Native组件堆砌的	推荐	不推荐	不支持
	原生能力组件	图片选择、地图定位等	推荐	不推荐	不支持
全局模块	高性能模块	购物车拖拽、用户频繁加减菜	推荐	不推荐	不支持
	强交互模块	智能点餐	推荐	不推荐	不支持
外链	需要外链展示	需要嵌入没有MRN环境的App	不支持	不支持	推荐

发布运维支撑

发布运维是一个成熟的软件项目中非常核心的部分，它保证了整个项目能够高效且稳定地运转。建立一个稳定可靠的发布运维体系是我们建设整个外卖 MRN 技术

体系的重要目标。但发布运维的建设上下游牵扯了众多基建：拥有一个合理的工程结构对发布运维来说至关重要。如果工程结构臃肿且混乱，将会引起的一系列的权限问题、管理维护问题，这样会严重制约整个发布运维体系的效率。所以 MRN 的工程架构演进优化也是发布运维体系建设的重要组成部分。

MRN 分库 & 工程结构演进

业务分库

任何一个大型、长期的前端技术项目，良好的工程结构都是研发发布支撑中非常核心的部分。从 2018 年 10 月份，外卖正式启动 MRN 项目以来，面临涉及近百个 MRN 和几十人参与的大规模 MRN 应用计划。从项目初期，我们就开始寻找一个非常适合开发维护的工程结构。

在最开始的时候，我们的目标是快速验证及落地，使用了一个 Git 库与一个 Talos 项目（美团自研发布系统）去承接所有页面的开发及发布工作，同时对权限进行了收缩，保证初期阶段的安全发布。然而随着页面的增多，每个版本的发布压力逐渐增大。发布 SOP 上的三大关键节点权限：Git 库操作权限、Talos 的发布权限、美团自研的线上降级系统 Horn 权限，互不相关，负责人也各异，导致发布时常因各个节点的权限审批问题，严重阻塞效率。

随着项目的大规模铺开，我们的页面数量、合并上线次数与初期已不可同日而语。为了解决逐渐臃肿的代码仓库问题及发布效率问题，我们将庞大而臃肿的 RN 库根据业务维度和维护团队拆分成了 4 个业务库，分别是订单业务、流量业务、商家业务、营销业务，并确认各库的主 R，建立对应的 Talos 项目，而主 R 也是对应 Talos 项目的负责人。同时所有的主 R 都有 MRN 灰度脚本的管控权限。这样一来，MRN 的工程结构和 Native 的工程结构完全对齐，每个责任人都非常明确自己的职责，不会来回地穿插在不同的业务之间，同时业务库任意页面的发布权限都进行了集中，RD 只需要了解业务的负责人，即可找到对应的主 R 完成这个业务的所有相关工作。

工程结构

在项目初期，对于每个库的工程结构，美团内部比较流行的工程结构有两种：一个是适合小型业务开发的单工程多 Bundle 方案，另一个是相对更适合中大型业务开发的多工程多 Bundle 方案。

单工程单 Bundle 方案

顾名思义，单工程单 Bundle 方案的意思就是一个前端工程承载所有的业务代码，最终的产物也只有一个 RN Bundle。通过入参决定具体加载哪个页面。

对于业务不多，参与人不多的团队，使用单工程单 Bundle 的方式即可快速完成开发、发布。因为通过一次发布就可以完成整个发布的工作，但是带来的弊端也是不可接受的：因为所有业务都耦合在一起，每次更新都会“牵一发而动全身”，增大了问题的隐患。如果多个业务需求同时提测的时候，在团队配合上也是一个极大的挑战，因为新版本号会覆盖旧版本号，导致两个需求提测时会出现相互覆盖的情况。所以我们在立项之初就排除了这种方案。

多工程多 Bundle 方案

多工程多 Bundle 方案的意思就是一个 Git 库中存放了多个页面文件夹，各个文件夹是完全独立的关系，各自是一个完整的前端工程。拥有自己独立的 MRN 配置信息、package.json、组件、Lint 配置等（如下图所示）。每个页面文件夹都输出一个独立的 RN Bundle。

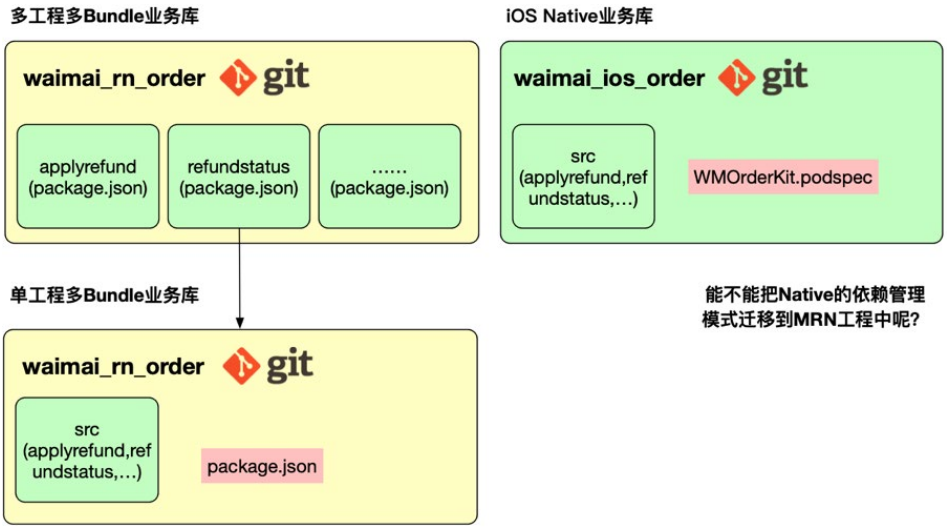
相比于单工程单 Bundle 方案，多工程多 Bundle 方案将页面进行解耦，使之基本可以满足中大型 MRN 项目的需求。在外卖 MRN 项目初期，一直都使用着这样的工程结构进行开发。但是我们也为之付出了相应的代价，即每个页面的依赖都需要对应 RD 去维护升级，依赖碎片化的问题日趋严重。同时在工程级别的管控，如统一 Lint 规则、Git Hook 等也变得更加复杂。

多工程多 Bundle 方案 => 单工程多 Bundle 方案

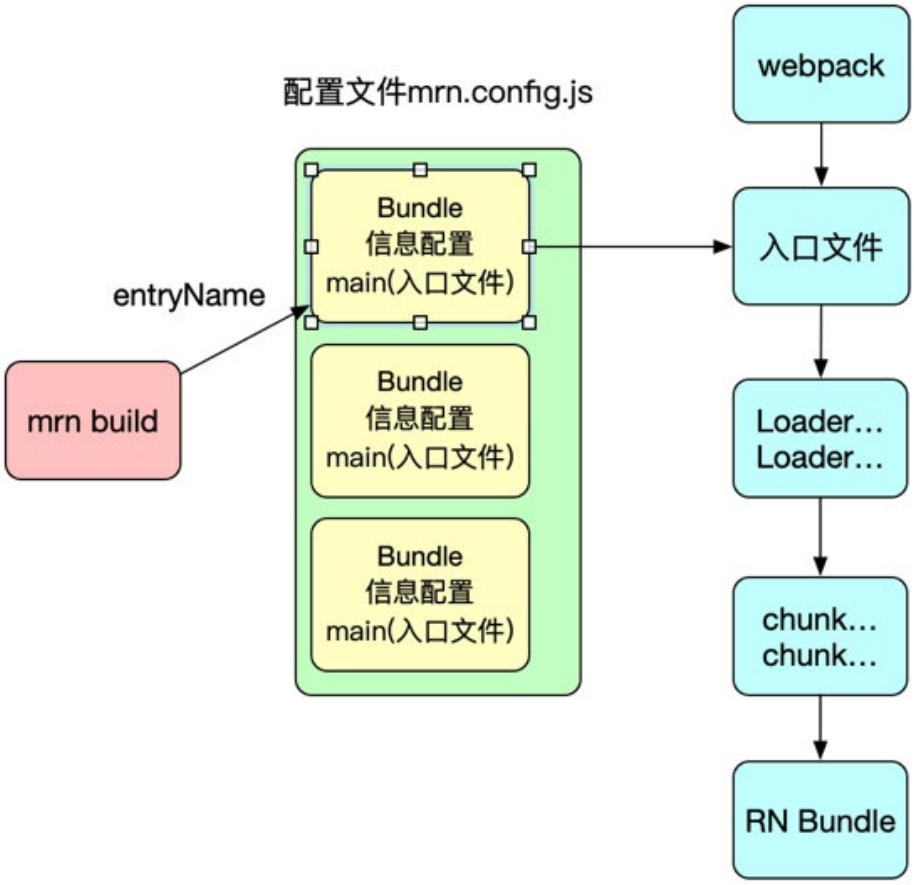
随着外卖 MRN 页面规模以及参与人规模的进一步增大，多工程多 Bundle 方案的缺点日益凸显。特别对于那些前端技术底子相对薄弱的团队来说，依赖管理问题会

变得很头疼。在这种情况下，单工程多 Bundle 的方案就应运而生了。

核心思路也很简单：观察一下单工程单 Bundle 方案和多工程多 Bundle 方案的优缺点可知，单工程单 Bundle 依赖管理方便的优点主要来自于“单工程”，而多工程多 Bundle 的业务解耦的优点主要来自于“多 Bundle”。所以结合这两种工程方案的核心优点，就可以设计一种新方案：单工程多 Bundle。即用一个工程去承接所有的页面代码，但是又可以让每个页面输出独立的 RN Bundle 来保证互不影响。其实，这种方式类似于 Native 一个静态库的管理，如下图所示：



通过分析 MRN 的打包原理可知，MRN 通过一个配置文件配置了一个 Bundle 的所有业务信息以及 mrn-pack2 的打包入口。所以我们只需要让配置文件支持多份 Bundle 信息的配置，通过打包命令与参数选择正确的 mrn-pack2 打包入口，即可打出我们最终所需要的业务 Bundle。如下图所示：



核心优势：

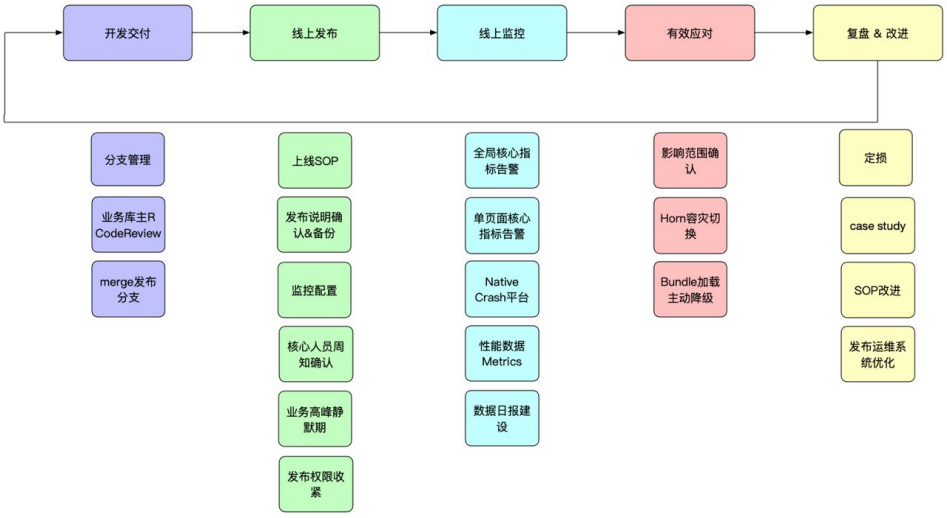
- 整个工程采用一个 package.json，管理业务库中所有的依赖。这样可以有效地解决各自页面去管理自己依赖时，必然产生的依赖版本碎片化问题，避免同一依赖库因为版本不一样，而导致页面表现不一样的问题。
- 从依赖角度去规范各自页面的使用工具规范，如 A 页面使用某一种三方库来实现某种功能，B 页面使用另一种三方库也实现了同一种功能，单一依赖管理就可以从库依赖的角度强制做技术选型，减少各个页面的实现差异，从而降低维护成本。
- 让业务同学可以更加专心地开发业务代码，不用关心复杂的依赖问题，大大提升了开发效率。

实现了工程级别的管控，如 Pre-Commit，脚手架方案管理将变得更加便捷。

这种工程组织形式也成为了 MRN 工程结构的最佳实践，而且美团内部也有多个团队采用了这种解决方案。目前已支撑超过几百个页面的开发和维护工作。

外卖发布运维体系

下图展示了我们的发布运维全景，共覆盖了开发交付、线上发布、线上监控、有效应对、复盘改进等五大模块。接下来我们会逐一进行介绍。



（1）开发交付

开发阶段，需求 RD 完成开发，提交到 Git 库的发布分支。对应的业务库主 R 角色（通常由 RN 经验较丰富的工程师来承担）进行 CodeReview，确认无误之后会执行代码的合并操作。顺便说一下，这也是外卖 RN 质量保障长征路的第一步。

（2）线上发布

合入发布分支之后，就可以正式启动一次 RN Bundle 发布。这里我们借助了美团内部的 Talos 完成整个发布过程，Talos 的发布模板与插件流水线规范了一次发布需要的所有操作，核心步骤包括发布准备（Git 拉代码、环境参数确认、本次发布说明填写）、发布自检（依赖问题检查、Lint、单元测试）、正式打包（Build、版本号自更新）、产物上传测试环境（测试 / 线上环境隔离、测试环境进行测试），双重确认

（QA、Leader 确认发布）、产物上传线上环境等等。

产物上传线上环境，实际上是上传到了美团内部的 CD 平台 –Eva。在 Eva 上，我们可以借助 RN Bundle 的发布配置去约束发布 App 的版本号、SDK 版本等，以及具体的发布比例及地区，去满足我们不同的发布需求。最终执行发布操作，将 RN Bundle 上传到 CDN 服务器，供用户下载，完成整个发布流程。

（3）运维监控

发布之后，运维是重中之重。首先我们的运维难点在于我们的业务横跨两个平台——美团 App 与外卖 App。由于它们在基建、扩展、网络部分都存在差异，所以我们选取指标的维度不仅要从业务出发，还要增加全局的维度，来确保外卖平台 MRN 的正常运转。基于这个层面的思考，我们选取了一系列 RN 核心指标（在下面的章节会详细列举），进行了全方位的监控。目前外卖客户端，已经做到分钟级监控、小时级监控和日级别监控等三档监控。

在监控手段上，首先我们使用了美团开源的 Cat 告警平台（这部分已经通过 Talos 插件完全自动化配置），确保当核心指标在线上出现波动、异常的时候，相关 RD、QA 以及业务负责人可以及时接受到报警，并由对应的 RD 主 R 负责，快速进入到“有效应对”的环节。同时为了能够分阶段、更好地处理问题，我们将核心指标报警分为【P1】与【P0】两个级别，分别代表“提高警觉，确认问题”与“大事不好，马上处理”。保障了一个问题出现之后能够及时被发现并快速进行处理。

除了监控报警手段之外，我们还会借鉴客户端高可用性保障的经验。用一些日常运维的手段去发现问题。比如使用灰度小助手、数据日报等手段从宏观角度主动去发现存在隐患的指标，及时治理，避免问题。

（4）有效应对

根据“墨菲定律”：如果事情有变坏的可能，不管这种可能性有多小，它总会发生。即便我们在发布管控和线上监控上做的再充分，线上问题最终还是无法避免的。所以当通过线上告、客诉等手段发现线上问题之后，我们需要及时的应对问题、解决问题，把问题带来的影响降低到最小，并以最快的速度恢复对用户的服务。

在有效应对的方面，我们主要靠两种手段。第一种是存在 B 方案兜底的情况，使

用 Horn 灰度配置，关掉 MRN 开关，短时间内恢复成 Native 页面或者 H5 页面继续为用户提供服务，同时通知相关 RD 和 QA 快速定位问题，及时修复，验证并上线。第二种是无兜底方案的情况，CDN 服务器（Eva）上撤掉问题 Bundle，实现版本回滚，接下来的问题定位过程跟手段保持一致。

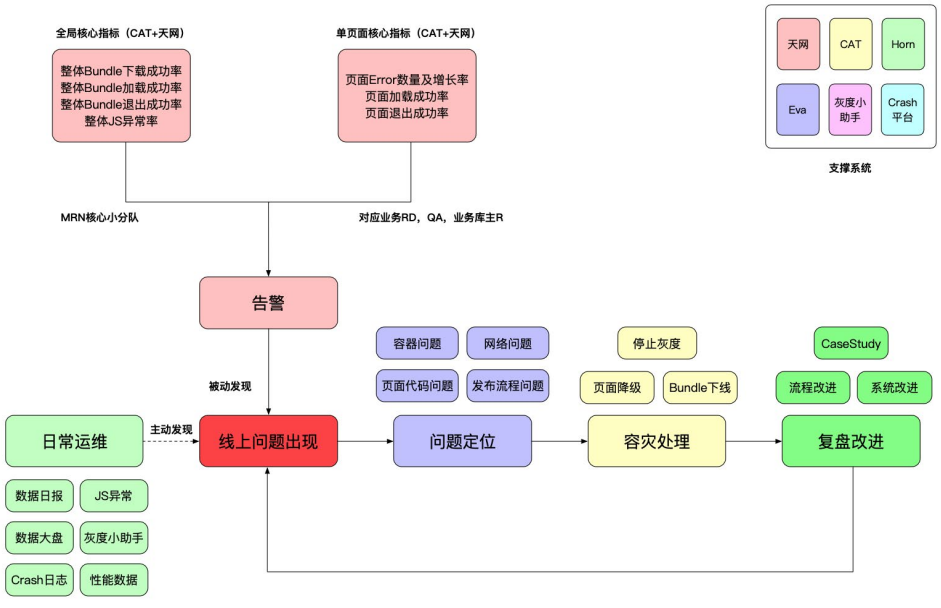
这两种备案保障了外卖 MRN 业务的整体高可用性。

（5）复盘改进

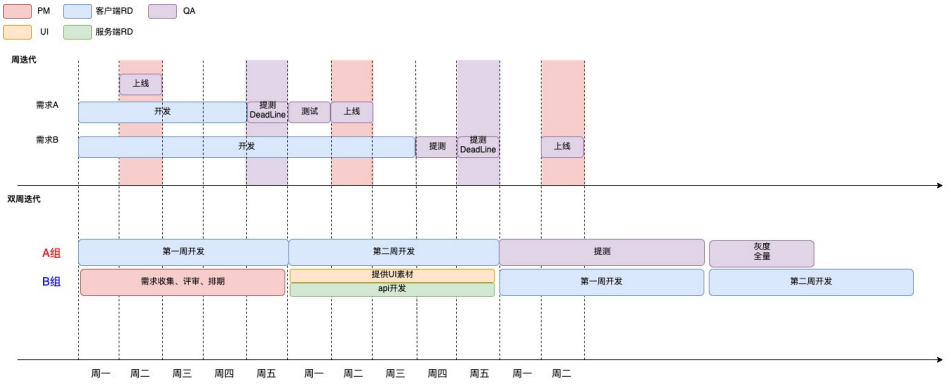
在以上四个大环节中，问题可能会出现在任意一个环节。除了及时发现问题与解决问题，我们还需要尽力避免问题。这一点主要是靠我们内部的例会、复盘会，对典型问题进行 Review，将问题进行归类，包括复盘流程规范问题、操作失误问题、框架 Bug 等，并力图通过规范流程、系统优化来尽力地避免问题。

在外卖 MRN 项目实施过程中，我们共推动了二十多项规范流程、系统优化等措施，大大保障了整体服务的稳定性。

最后，我们用一张图对外卖监控运维体系做一个总结，帮助大家有一个全局的认知。



混合式架构流程



针对混合式架构的流程，目前外卖技术团队采用的是正常双周版本迭代流程 + 周迭代上线流程。MRN 页面既可以跟版迭代，也可以不跟版迭代，这样可以有效地减少流程的复杂度和降低 QA 的测试成本，而周迭代流程可以有效地利用 MRN 动态发版的灵活性。混合式开发和原生开发应尽量保持时间节点和已有流程的一致。这种设计的好处在于，一方面随着动态化的比例越来越高，版本迭代将可以无限拉长，另一方面从双周迭代逐渐演变成周迭代的切换成本也得到大幅的降低。详细可分为下面几个阶段：

评审阶段

业务评审阶段在原有的流程上，增加了技术选型阶段。在技术选型时，明确是否会存在需要使用 MRN 页面的情况，如果页面可以完全不涉及到 Native 部分即可完成，就可以进入周迭代的发版流程。如果需求用 MRN 实现，但是又涉及到 Native 部分，仍然走周迭代的上线流程。除正常开发需求的时间外，RD 需综合考虑到双端上的适配成本。

开发阶段

客户端以周维度进行开发，每周确定下周可提测的内容，根据提测内容是否为动态化的业务、下周是否在版本迭代周期内，决定跟版发布或周发布。

提测阶段

提测前，为了保证 MRN 页面的提测质量，RD 首先需要按照 QA 提供的测试用例提前发现适配问题。提测时需要在提测邮件中注明：(1) 提测的 Bundle 名称和对应的版本号；(2) 标明哪些组件涉及 Native 模块；(3) 依赖变更情况，如是否升级了基础库，升级后的影响范围；(4) 重点测试点的建议。

上线阶段

MRN 由于其可动态发布的特性可以跟版发布，也可不跟版发布，但上线时间和灰度时间节点都保持了一致。不过版本还是动态发版，都默认周二上线，周四全量。

- 跟版发布：默认只对当前版本生效，需在双周迭代三轮提测节点，周二当天将 Bundle 上线服务器，MRN 的灰度开关全量打开。通过周四 App 的发版灰度比例来控制 MRN 的灰度比例，上线时需配置报警和灰度助手监控，实时掌握 MRN 的线上数据。
- 不跟版发布：也同样以周四作为全量发布窗口，Bundle 需在周二时上线指定线上版本，指定 QA 白名单。测试通过后，在周三按照比例逐步灰度，周四正式全量，和跟版发布一样，上线时需要配置报警和监控。

架构总结

引入 MRN 后，相对单平台而言，架构层级上，我们增加了 2 个 MRN 中间层去屏蔽 Android 和 iOS 平台、原生组件之间的差异。这样做的目的是为了让上层业务开发者可以很快地使用框架进行业务开发，完全不用关心平台和组件间的差异。通过引入 MRN 技术栈，带来的好处很明显：

(1) 使用 MRN 实现的页面理论上可以实现一套代码，部署到不同平台上，开发效率得到大幅度提升。(2) 采用 MRN 框架，无论是加载性能还是页面滑动性的用户体验上，都会比原来 H5 的方式要好。(3) 部分页面具备了快速编译、快速发布的能力。

但一个硬币总有两面，混合式架构增加了架构的复杂度，使得原本只要考虑一个平台的事情，逐渐转变成需要考虑三个平台，另外 Android 本身具备碎片化的问题，

这使得混合式架构的适配问题较为突出。当出现问题时，我们的第一反应由“这是什么问题”变成“它是否存在于两个平台，还是只在一个平台上?”、“如果仅在一个平台上，是在原生代码还是 React Native 代码出了问题?”、“历史版本的 MRN 是否存在问题，是否需要修复”、“修复的效果在 Android 和 iOS 上的表现是否一样”，这些问题增加了定位和修复工作的复杂性。另外，MRN 的适应场景也是有限的，并非所有的业务和页面都适合改造成 MRN，如何做选择也需要进行有效的判断，从而增加了决策成本。

针对上述问题，我们的建议是：

（1）减少分歧

- 在研发、测试、发布和运维环节，MRN 的页面尽可能对齐 Native 原有的环节，减少团队理解的成本。
- 在 Debug 开发环境下，利用页面浮层提示技术栈使用情况；Release 环境下，利用工具、MRN 自动化报表，及时的让开发同学明确知道是 Native 页面还是 MRN 页面，减少确认。
- MRN 页面尽可能地避免原生组件的使用，而使用纯 JS 代码实现，供 MRN 页面使用的原生组件的需要高质量的提供，减少下层组件的问题。
- 默认只修复当前的版本，出现严重问题时才考虑修复历史版本，减少多版本带来的复杂度提升。

（2）技术栈明确边界

- 做好 Native 和 MRN 技术栈使用的边界，尽可能用简单的选型标准，让合适的场景选用合适的技术栈，从而保证业务整体的可用性，让用户体验依然如初。

（3）单技术栈转向多技术栈团队

- 培养全栈工程师，当团队的同学都具备 iOS、Android 和 MRN 多个技术栈能力时，将会有效地提升开发的效率，短期内可选择 iOS、Android 和 MRN 工程师结伴编程的策略。

可用性体系

正如在“监控运维”章节中所讲到的那样，线上运维是我们工作的重中之重。这个章节我们就讲一下我们对于监控指标的选取。鉴于外卖业务的特殊性，除了美团的外卖频道之外，外卖业务还需要运行在独立的外卖 App 上。如下图所示：

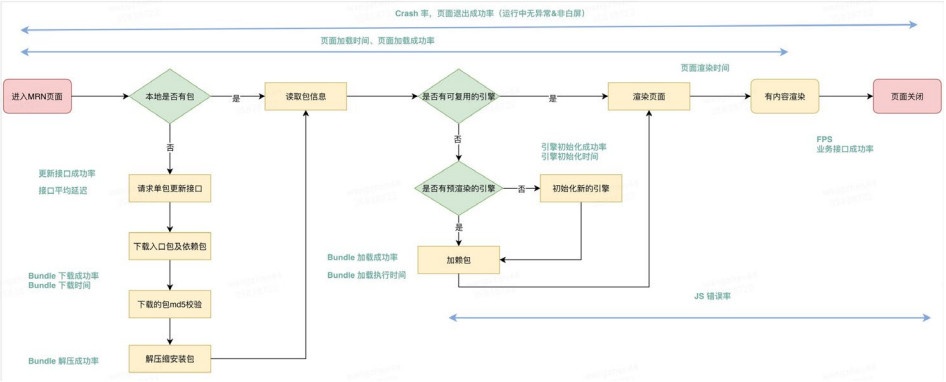


外卖 App 经过多年的发展，目前已逐渐成为一个平台级应用，承接了 C 端、闪购、跑腿等多个业务。与美团 App 相比，它们之间在很多基础建设、扩展、网络部分都存在差异。所以在监控核心指标的选取上，我们除了保证 C 端 MRN 业务在美团以及外卖两端的高可用性，还需要保证外卖 App 平台本身基建的稳定性，从而保证运转在外卖 App 上所有 MRN 业务的高可用性。

而从监控的大分类上来讲，我们分为了【可用性指标】以及【性能指标】，它们分别关注业务本身的可用性，以及页面的性能与用户体验。接下来，我们就依次进行讲解。

MRN 可用性指标

可用性指标也是我们关注的关键指标，它直接决定了我们的 MRN 页面是否能够正确、稳定地为用户提供服务。通过 MRN Bundle 加载全景，我们可以确定整个包加载的几个关键节点。可以说，MRN 业务的可用性就是取决于这些关键节点的成功率。



下载链路

MRN 是一个动态化的框架，所有的 MRN Bundle 都是从 CDN 节点上远程下载。所以下载成功是 MRN 业务可用的先决条件。有些普通的业务方是不需要关注这个指标的，而外卖 App 可能会因为网络库基建，出现启动下载线程拥堵、DNS 劫持等问题，所以我们把下载成功率作为外卖 App 监控的全局指标。目前，外卖 App 的下载成功率长期稳定在 99.9% 左右。

加载链路

加载链路可以细分为初始化引擎部分以及业务 Bundle 加载部分。前者跟基建有关，代表从引擎创建到加载完 Common 包加载成功这段的成功率。这部分主要依赖 MRN SDK 的稳定性，从我们的日报上看，稳定性基本保持在 99.99% 以上。

而业务 Bundle 加载成功率（MRN PageLoad Success），是 MRN 页面创建到业务视图内容渲染过程中，没有发生错误的比例。它与跟拉包时网络情况、MRN 框架稳定性和业务 JS 代码都有关系。这也是我们关注的核心指标，因为它直接决定了我们某个页面是否可以渲染成功，所以我们把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。目前，整个外卖业务的 Bundle 加载成功率稳定在 99.9% 以上。

使用链路

Bundle 加载成功之后，页面成功被渲染。但是在使用的过程中，可能会因为 JS

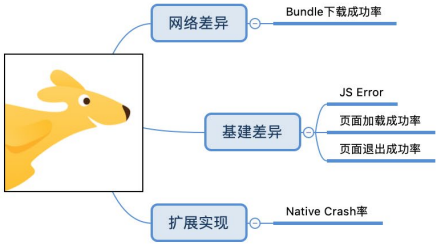
代码，Native 代码的 Bug 出现 JS Error、Native Crash 等问题，这样给用户带来的直观反馈就是应用闪退、页面白屏等，造成了服务的不可用。所以在使用链路上出现问题率，基本也可以直观反映出一个 RN 页面的质量以及它当前的运行状况。

在使用链路上，我们主要关注的是 JS Error 率、JS Error 个数以及页面退出成功率（MRN PageExit Success）等。

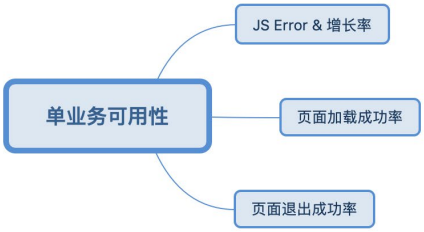
JS Error 很好理解，由于 RN 是由 JS 驱动的框架，所以一个页面的 JS Error 率基本上可以综合反映出一个页面的可用性、稳定性或者基建的稳定性，故我们同样把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。我们用上报上来的 JS Error 数量做分子，该页面的 PV 做分母，计算一个页面的 JS 错误率，当 JS Error 个数短时间内极速升高或者 JS Error 率有大幅上升时，就会触发我们的 JS Error 告警。目前外卖大盘的 JS Error 率保持在万分之一左右，略低于 Native Crash 率。

页面退出成功率（MRN PageExit Success），理解起来不如前面的指标那么简单，因为它表示的是用户在退出 MRN 页面时，业务视图内容已成功渲染的比例。它会包含所有已知和未知的异常，但是用户进入页面后快速退出的场景，也会被错误的统计在其中，因为用户退出时可能页面尚在加载中。相比于 JS Error，它是一个更加综合的指标，基本上涵盖了加载失败、渲染白屏、使用时出现错误等多个异常场景，基本上可以反映出一次 MRN 业务的单次可用性，相比于之前的指标会更加严格。我们把这个指标同时列为了外卖 App 监控告警的全局指标与单 Bundle 告警的指标。我们希望它永远能保持在 99.9% 以上，否则就会触发告警。目前外卖大盘的 MRN PageExit Success 基本稳定在万分之三左右，我们最终的目标是希望稳定在万分之一左右。

最后，我们希望通过两个“脑图”快速回顾一下外卖全局监控与单业务监控关注的核心指标。



全局监控



单业务监控

MRN 性能指标

除了可用性指标，性能指标也是我们重点关注的内容。如果加载时间过长，就会大大增加用户离开页面的概率。而页面卡顿，也会影响用户在使用层面的体验，从而引发客诉或者业务损失。

根据 Bundle 加载全链路图，我们也可以把性能指标分为两个大类，一个是加载时耗时与使用时性能指标。前者主要关注 Bundle 从 Load 到渲染整个链路的耗时，后者主要关注使用时的性能指标，在这里主要是指页面的 FPS。

加载链路耗时

如上述所说，整个加载链路分为引擎初始化的时间以及 Bundle 本身加载及渲染的时间的时间。

引擎初始化的时间在整条链路上占比是最长的，因为初始化的时候会加载比一般业务代码大得多的 CommonJS。经过观察，这部分的时间总体表现较差，在 iOS 上 50 分位和 90 分位分别是 0.3s 和 0.7s。在 Android 上表现更差，50 分位和 90 分位分别是 1.3s 和 1.8s。不过目前 MRN 已经使用了预加载方案，即在 App 刚启动时就初始化一个 JS 引擎，等实际使用时，直接复用该引擎即可，大大缩短了首次 Bundle 的整体加载时间。

页面加载时间和页面渲染时间是我们关注的第二类指标，从加载链路图也可以发现，页面加载时间代表从开始加载 Bundle 到 RN 内容渲染成功的整条时间，而页面渲染时间则是它的子集，代表 Bundle 解析完毕，从 JS StartApplication 开始加载

组件到渲染出第一帧的时间（iOS 和 Android 的统计口径不同）。区分这两项指标也可以更好地分析整个加载链路上的瓶颈在哪，有助于针对性的做性能优化。

以外卖 iOS 50 分位为例，我们发现页面整体的加载时间在 400ms 左右，JS 渲染时间只需要 100ms 左右，主要的性能瓶颈在 Bundle 加载以及 JS Bundle 的解析部分，这也是我们接下来需要重点研究课题。

使用时 FPS

衡量用户使用体验比较直观的一个指标就是 FPS，较高的 FPS 会让用户更加顺畅地体验功能，完成操作。

目前，MRN 在外卖侧业务总体落地页面复杂度适中，遇到复杂动画也使用了 BindingX 来提升性能。通过监控，外卖侧的页面总体表现良好，在 iOS 上几近满帧，在 Android 上表现稍差，平均在 55 帧左右，较深的视图层级与较低的 JS-Native 的通信效率都是 MRN FPS 的杀手。如何提升 MRN 特别是在 Android 上的页面性能也是我们下一阶段研究的课题。

目前，外卖性能指标 50 分位的性能指标基本满足线上需求，但是 90 分位的表现不尽如人意，特别是较低的 FPS 以及过长的页面加载时间。革命尚未成功，同志仍需努力。

效率衡量

引入 MRN，提升了本地的开发效率，但同时也增加了工程的复杂度，所以总体来说真的能提升实际开发效率吗？在完成几十个 RN 页面的开发后，总结了一些公式，希望可以给其他团队一些结论性的参考。首先设定三个方面去考量：人效提升、代码复用、维护成本衡量，将外卖的所有 MRN 页面加在一起，取平均值，可以得出较为准确的结论：

- 人效提升计算公式：∑（Android Native 总人日 +iOS Native 总人日 −RN 总人日）/ ∑（Android Native 总人日 +iOS Native 总人日）
- 代码复用率计算公式：∑（RN 行数 − 平台分支判断代码块）/ ∑（RN 行数 +Android native+iOS Native）

- 维护成本计算公式：∑（Android Native 原生总行数 +iOS Native 页面总行数 −RN 页面总行数）/ ∑（Android Native 页面总行数 +iOS Native 页面总行数）

根据页面的交互程度去进一步的划分，得到如下的表格：

页面	人效提升	代码复用率	维护成本
纯展示页面	60%	100%	81.77%
简单交互页面	58%	88.69%	55.16%
复杂交互页面	55%	73.99%	46.60%
总计	57.70%	87.56%	61.18%

如表所示：人效提升的方面，主要取决于页面是否存在复杂的交互，如果页面存在复杂交互，就会不可避免的导致涉及到 Native 的双端原生开发，如部分交互需要 Native Module 实现，最终的人效提升将大打折扣。而对于涉及较少的 Native Module 和展示型的页面，MRN 存在较大优势。但大家会很奇怪这种结果，为什么人效提升会大于 50%？ 逻辑上 Android 和 iOS 双端复用后，提升的效率理论上最大应该是 50%。这是由于 RN bundle 的热加载极大地节省了 Native 的编译时间，这一部分相对原生开发效率大概能提升 20% 以上，使得最终的人效提升大于 50%。双端复用率方面，对于纯展示型的页面，大概率可以完全由 JS 实现，双端复用率可以达到 100%，后续双端只需维护一份 JS 代码即可，极大的降低了维护成本。对于一些交互复杂的页面，需双端各自封装对应的 Native Module 实现，复用率下降，维护成本变高。

总结

随着业务的快速发展，工程复杂度的不断提升，在没有外力的情况下，开发效率必然会持续下降。如何在资源有限的情况下不断提升开发效率是一个永恒的话题。美团外卖客户端通过借助美团基建 MRN，推动混合式架构来提升效率。截至目前，美

团外卖业务已经有 60 多个 RN 页面上线，每天的 PV 高达上千万，为用户提供了稳定可靠的服务。

混合式开发带来的不仅仅是技术层面的挑战，更是对团队成员、团队组织能力的挑战。MRN 虽然能够做到跨端，但是有时候仍然需要针对特定平台单独编写代码来解决问题，这就间接要求工程师必须熟悉三个平台，团队也必须有效组织各技术栈人才共同协作，才能真正用好 MRN。

参考文献

- 京东 618：RN 框架在京东无线端的实践
- React Native 架构分析
- 点我达骑手 Weex 最佳实践
- State of React Native 2018
- 使用 React Native 的五个理由
- iOS 开发是否要采用 React Native
- 开源 React Native 组件库 beeshell 2.0 发布
- ESLint 在中大型团队的应用实践
- CAT 3.0 开源发布，支持多语言客户端及多项性能提升

作者简介

晓飞、唐笛、维康，均为美团外卖前端团队研发工程师。

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，Base 北京、上海、成都，欢迎有兴趣的同学投递简历到 tech@meituan.com（邮件标题注明：美团外卖前端团队）。

Android 静态代码扫描效率优化与实践

肖鸿耀

DevOps 实践中，我们在 CI(Continuous Integration) 持续集成过程主要包含了代码提交、静态检测、单元测试、编译打包环节。其中静态代码检测可以在编码规范，代码缺陷，性能等问题上提前预知，从而保证项目的交付质量。Android 项目常用的静态扫描工具包括 CheckStyle、Lint、FindBugs 等，为降低接入成本，美团点评集团内部孵化了静态代码扫描插件，集合了以上常用的扫描工具。项目初期引入集团内部基建时我们接入了代码扫描插件，在 PR(Pull Request) 流程中借助 Jenkins 插件来触发自动化构建，从而达到监控代码质量的目的。初期单次构建耗时平均在 1~2min 左右，对研发效率影响甚少。但是随着时间推移，代码量随业务倍增，项目也开始使用 Flavor 来满足复杂的需求，这使得我们的单次 PR 构建达到了 8~9min 左右，其中静态代码扫描的时长约占 50%，持续集成效率不高，对我们的研发效率带来了挑战。

针对以上的背景和问题，我们思考以下几个问题：

思考一：现有插件包含的扫描工具是否都是必需的?

扫描工具对比

为了验证扫描工具的必要性，我们关心以下一些维度：

- 扫码侧重点，对比各个工具分别能针对解决什么类型的问题；
- 内置规则种类，列举各个工具提供的能力覆盖范围；
- 扫描对象，对比各个工具针对什么样的文件类型扫描；
- 原理简介，简单介绍各个工具的扫描原理；

- 优缺点，简单对比各个工具扫描效率、扩展性、定制性、全面性上的表现。

维度	CheckStyle	FindBugs	Lint
扫描侧重点	代码风格，编程规范，圈复杂度等	针对Java工程，Java代码的编码习惯、糟糕代码、性能以及安全等问题。	针对Android工程，检查 Android 项目源文件是否有潜在的错误，以及在正确性、安全性、性能、易用性、无障碍性和国际化方面是否需要优化改进。
内置规则种类	[100+] 检测规则	[300+] 检测规则	300+ 检测规则
扫描对象	源代码文件	Java字节码	Manifest文件、XML、Java、Kotlin、Java字节码、Gradle文件、Proguard文件、Property文件、图片资源
原理简介	使用Antlr库对源码文件做词语法分析生成抽象语法树，遍历整个语法树匹配检测规则	基于BCEL库通过扫描字节码完成代码检查，主要做缺陷模式匹配和数据流分析	基于抽象语法树分析，经历了LOMBOK-AST、PSI、UAST三种语法分析器
优点	对规范类规则检查很细致，速度快、轻量、针对代码风格有优势，耗时相对较少	针对字节码，对JDK定制化程度高、能发现Java代码中的一些潜在错误和缺陷	官方支持、检测全面、扩展性极强，支持自定义规则、配套工具完善
缺点	检查的规则相对简单，无法检查潜在Bug	定制规则门槛高，需要了解字节码，依赖编译代码，扫描比较耗时	如果检测了字节码文件，依赖编译代码，并且全量检测比较耗时，同时API变动比较大，目前还在不断优化迭代

注：FindBugs 只支持 Java1.0~1.8，已经被 SpotBugs 替代。鉴于部分老项目并没有迁移到 Java8，目前我们并没有使用 SpotBugs 代替 FindBugs 的原因如下，详情参考官方文档。 同时，SpotBugs 的作者也在讨论是否让 SpotBugs 支持老的 Java 版本，结论是不提供支持。

Supported Java version

SpotBugs is built by JDK8, and run on JRE8 and newer versions.

SpotBugs can scan bytecode (class files) generated by JDK8 and newer versions. However, support for Java 11 and newer is still experimental. Visit issue tracker to find known problems.

SpotBugs does not support bytecode (class files) generated by outdated JDK such as 10, 9, 7 and older versions.

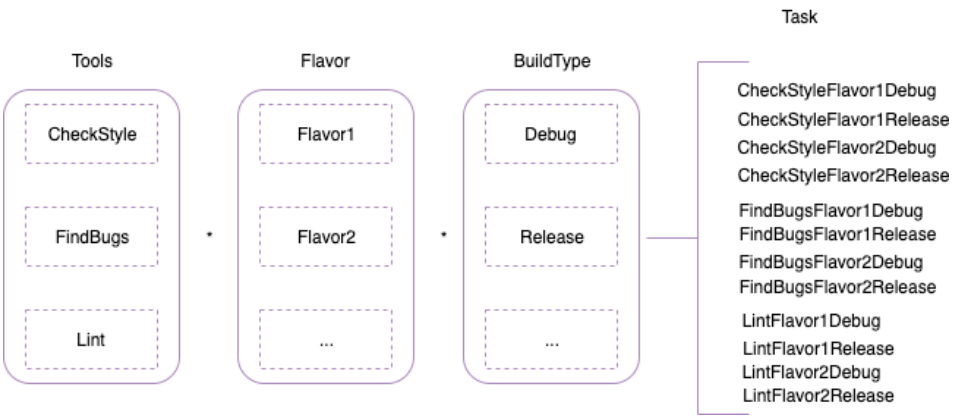
经过以上的对比分析我们发现，工具的诞生都能针对性解决某一领域问题。CheckStyle 的扫描速度快效率高，对代码风格和圈复杂度支持友好；FindBugs 针对 Java 代码潜在问题，能帮助我们发现编码上的一些错误实践以及部分安全问题和性能问题；Lint 是官方深度定制，功能极其强大，且可定制性和扩展性以及全面性都表现良好。所以综合考虑，针对思考一，我们的结论是整合三种扫描工具，充分利用每一个工具的领域特性。

思考二：是否可以优化扫描过程？

既然选择了整合这几种工具，我们面临的挑战是整合工具后扫描效率的问题，首先来分析目前的插件到底耗时在哪里。

静态代码扫描耗时分析

Android 项目的构建依赖 Gradle 工具，一次构建过程实际上是执行所有的 Gradle Task。由于 Gradle 的特性，在构建时各个 Module 都需要执行 CheckStyle、FindBugs、Lint 相关的 Task。对于 Android 来说，Task 的数量还与其构建变体 Variant 有关，其中 Variant = Flavor * BuildType。所以一个 Module 执行的相关任务可以由以下公式来描述：Flavor * BuildType *（Lint，CheckStyle，Findbugs），其中 * 为笛卡尔积。如下图所示：



可以看到，一次构建全量扫描执行的 Task 跟 Varint 个数正相关。对于现有工程的任务，我们可以看一下目前各个任务的耗时情况：（以实际开发中某一次扫描为例）



通过对 Task 耗时排序，主要的耗时体现在 FindBugs 和 Lint 对每一个 Module 的扫描任务上，CheckStyle 任务并不占主要影响。整体来看，除了工具本身的扫描时间外，耗时主要分为多 Module、多 Variant 带来的任务数量耗时。

优化思路分析

对于工具本身的扫描时间，一方面受工具自身扫描算法和检测规则的影响，另一方面也跟扫描的文件数量相关。针对源码类型的工具比如 CheckStyle 和 Lint，需要经过词法分析、语法分析生成抽象语法树，再遍历抽象语法树跟定义的检测规则去匹配；而针对字节码文件的工具 FindBugs，需要先编译源码成 Class 文件，再通过

BCEL 分析字节码指令并与探测器规则匹配。如果要在工具本身算法上去寻找优化点，代价比较大也不一定能找到有效思路，投入产出比不高，所以我们把精力放在减少 Module 和 Variant 带来的影响上。

从上面的耗时分析可以知道，Module 和 Variant 数直接影响任务数量，一次 PR 提交的场景是多样的，比如多 Module 多 Variant 都有修改，所以要考虑这些都修改的场景。先分析一个 Module 多 Variant 的场景，考虑到不同的 Variant 下源代码有一定差异，并且 FindBugs 扫描针对的是 Class 文件，不同的 Variant 都需要编译后才能扫描，直接对多 Variant 做处理比较复杂。我们可以简化问题，用以空间换时间的方式，在提交 PR 的时候根据 Variant 用不同的 Jenkins Job 来执行每一个 Variant 的扫描任务。所以接下来的问题就转变为如何优化在扫描单个 Variant 的时候多 Module 任务带来的耗时。

> 对于 Module 数而言，我们可以将其抽取成组件，拆分到独立仓库，将扫描任务拆分到各自仓库的变动时期，以 aar 的形式集成到主项目来减少 Module 带来的任务数。那对于剩下的 Module 如何优化呢？无论是哪一种工具，都是对其输入文件进行处理，CheckStyle 对 Java 源代码文件处理，FindBugs 对 Java 字节码文件处理，如果我们可以通过一次任务收集到所有 Module 的源码文件和编译后的字节码文件，我们就可以减少多 Module 的任务了。所以对于全量扫描，我们的主要目标是来解决如何一次性收集所有 Module 的目标文件。

思考三：是否支持增量扫描？

上面的优化思路都是基于全量扫描的，解决的是多 Module 多 Variant 带来的任务数量耗时。前面提到，工具本身的扫描时间也跟扫描的文件数量有关，那么是否可以从扫描的文件数量来入手呢？考虑平时的开发场景，提交 PR 时只是部分文件修改，我们没必要把那些没修改过的存量文件再参与扫描，而只针对修改的增量文件扫描，这样能很大程度降低无效扫描带来的效率问题。有了思路，那么我们考虑以下几个问题：

- 如何收集增量文件，包括源码文件和 Class 文件?
- 现在业界是否有增量扫描的方案，可行性如何，是否适用我们现状?
- 各个扫描工具如何来支持增量文件的扫描?

根据上面的分析与思考路径，接下来我们详细介绍如何解决上述问题。

全量扫描优化

搜集所有 Module 目标文件集

获取所有 Module 目标文件集，首先要找出哪些 Module 参与了扫描。一个 Module 工程在 Gradle 构建系统中被描述为一个“Project”，那么我们只需要找出主工程依赖的所有 Project 即可。由于依赖配置的多样性，我们可以选择在某些 Variant 下依赖不同的 Module，所以获取参与一次构建时与当前 Variant 相关的 Project 对象，我们可以用如下方式：

```
static Set<Project> collectDepProject(Project project, BaseVariant
variant, Set<Project> result
= null) {
  if (result == null) {
    result = new HashSet<>()
  }
  Set taskSet = variant.javaCompiler.taskDependencies.
getDependencies(variant.javaCompiler)
  taskSet.each { Task task ->
    if (task.project != project && hasAndroidPlugin(task.project)) {
      result.add(task.project)
      BaseVariant childVariant = getVariant(task.project)
      if (childVariant.name == variant.name || "${variant.
flavorName}${childVariant.buildType.
name}".toLowerCase() == variant.name.toLowerCase()) {
        collectDepProject(task.project, childVariant, result)
      }
    }
  }
  return result
}
```

目前文件集分为两类，一类是源码文件，另一类是字节码文件，分别可以如下处理：

```
projectSet.each { targetProject ->
  if (targetProject.plugins.hasPlugin(CodeDetectorPlugin) && GradleUtils.
hasAndroidPlugin(targetProject)) {
    GradleUtils.getAndroidExtension(targetProject).sourceSets.all {
AndroidSourceSet sourceSet ->
      if (!sourceSet.name.startsWith("test") && !sourceSet.name.
startsWith(SdkConstants.FD_TEST)) {
        source sourceSet.java.srcDirs
      }
    }
  }
}
```

注：上面的 Source 是 CheckStyle Task 的属性，用其来指定扫描的文件集合；

```
// 排除掉一些模板代码 class 文件
static final Collection<String> defaultExcludes =
(androidDataBindingExcludes +
androidExcludes + butterKnifeExcludes + dagger2Excludes).asImmutable()

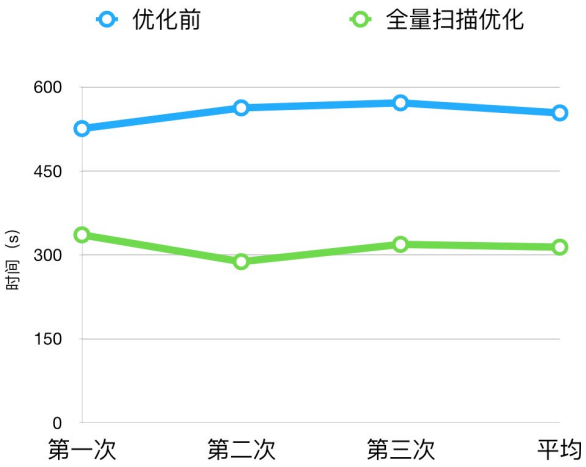
List<ConfigurableFileTree> allClassesFileTree = new ArrayList<>()
ConfigurableFileTree currentProjectClassesDir = project.fileTree(dir:
variant.javaCompile.
destinationDir, excludes: defaultExcludes)
allClassesFileTree.add(currentProjectClassesDir)
GradleUtils.collectDepProject(project, variant).each { targetProject ->
  if (targetProject.plugins.hasPlugin(CodeDetectorPlugin) && GradleUtils.
hasAndroidPlugin(targetProject)) {
    // 可能有的工程没有 Flavor 只有 buildType
    GradleUtils.getAndroidVariants(targetProject).each { BaseVariant
targetProjectVariant ->
      if (targetProjectVariant.name == variant.name ||
"${targetProjectVariant.name}".
toLowerCase() == variant.buildType.name.toLowerCase()) {
        allClassesFileTree.add(targetProject.fileTree(dir:
targetProjectVariant.javaCompile.
destinationDir, excludes: defaultExcludes))
      }
    }
  }
}
```

注：收集到字节码文件集后，可以用通过 FindBugsTask 的 Class 属性指定扫描，后文会详细介绍 FindBugs Task 相关属性。

对于 Lint 工具而言，相应的 Lint Task 并没有相关属性可以指定扫描文件，所以在全量扫描上，我们暂时没有针对 Lint 做优化。

全量扫描优化数据

通过对 CheckStyle 和 FindBugs 全量扫描的优化，我们将整体扫描时间由原来的 9min 降低到了 5min 左右。



增量扫描优化

由前面的思考分析我们知道，并不是所有的文件每次都需要参与扫描，所以我们可以通过增量扫描的方式来提高扫描效率。

增量扫描技术调研

在做具体技术方案之前，我们先调研一下业界的现有方案，调研如下：

工具	现有方案	备注	初步调研方案
checkStyle	https://github.com/yangziwen/diff-checkstyle	基于git命令对比提交记录的差异获取差异文件	可以从配置参数入手，指定扫描的Java源文件集合
FindBugs	暂无	暂无此方面资料	可以从配置参数入手，指定扫描的Class文件集合
Lint	Lint增量扫描（基于Android Gradle Plugin 2.X实现）	未开源，Android Gradle Plugin 3.x以后实现方式不一样，需要单独处理	需要了解内部实现原理，自定义增量扫描Client

针对 Lint，我们可以借鉴现有实现思路，同时深入分析扫描原理，在 3.x 版本上寻找出增量扫描的解决方案。对于 CheckStyle 和 FindBugs，我们需要了解工具的相关配置参数，为其指定特定的差异文件集合。

> 注：业界有一些增量扫描的案例，例如 diff_cover，此工具主要是对单元测试整体覆盖率的检测，以增量代码覆盖率作为一个指标来衡量项目的质量，但是这跟我们的静态代码分析的需求不太符合。它有一个比较好的思路是找出差异的代码行来分析覆盖率，粒度比较细。但是对于静态代码扫描，仅仅的差异行不足以完成上下文的语义分析，尤其是针对 FindBugs 这类需要分析字节码的工具，获取的差异行还需要经过编译成 Class 文件才能进行分析，方案并不可取。

寻找增量修改文件

增量扫描的第一步是获取待扫描的目标文件。我们可以通过 git diff 命令来获取差异文件，值得注意的是对于删除的文件和重命名的文件需要忽略，我们更关心新增和修改的文件，并且只需要获取差异文件的路径就好了。举个例子：git diff –name-only –diff-filter=dr commitHash1 commitHash2，以上命令意思是对比两次提交记录的差异文件并获取路径，过滤删除和重命名的文件。对于寻找本地仓库的差异文件上面的命令已经足够了，但是对于 PR 的情况还有一些复杂，需要对比本地代码与远程仓库目标分支的差异。集团的代码管理工具在 Jenkins 上有相应的插件，该插件默认提供了几个参数，我们需要用到以下两个：- ${targetBranch}：需要合入代码的目标分支地址；- ${sourceCommitHash}：需要提交的代码 hash 值；

通过这两个参数执行以下一系列命令来获取与远程目标分支的差异文件。

```
git remote add upstream ${upstreamGitUrl}
git fetch upstream ${targetBranch}
git diff --name-only --diff-filter=dr $sourceCommitHash
upstream/$targetBranch
```

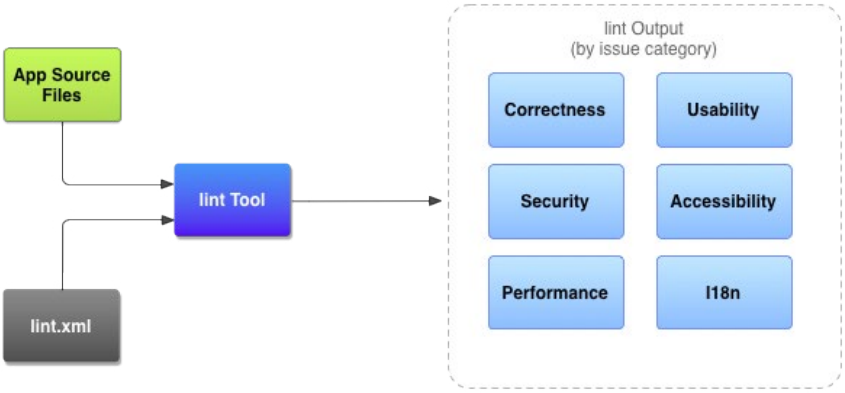
1. 配置远程分支别名为 UpStream，其中 upstreamGitUrl 可以在插件提供的配置属性中设置；
2. 获取远程目标分支的更新；

3. 比较分支差异获取文件路径。

通过以上方式，我们找到了增量修改文件集。

Lint 扫描原理分析

在分析 Lint 增量扫描原理之前，先介绍一下 Lint 扫描的工作流程：



App Source Files

项目中的源文件，包括 Java、XML、资源文件、proGuard 等。

lint.xml

用于配置希望排除的任何 Lint 检查以及自定义问题严重级别，一般各个项目都会根据自身项目情况自定义的 lint.xml 来排除一些检查项。

lint Tool

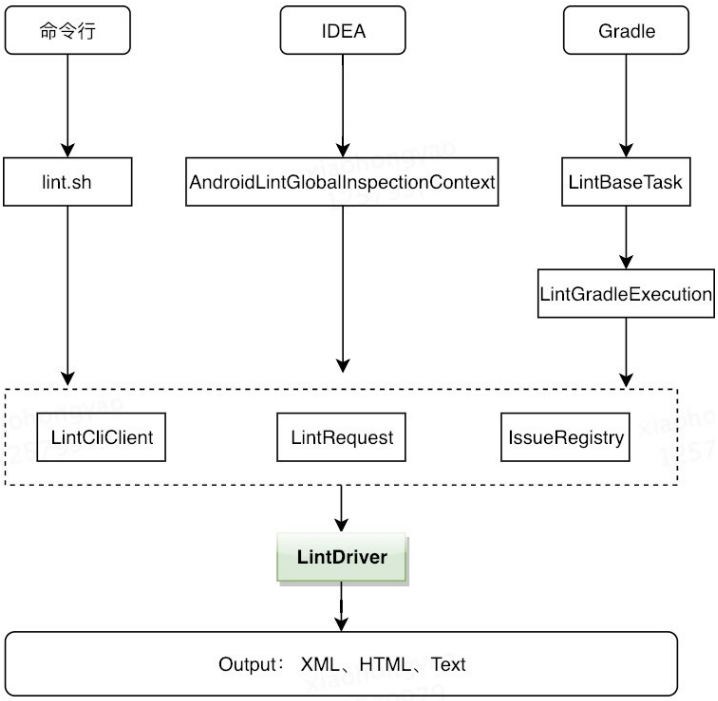
一套完整的扫描工具用于对 Android 的代码结构进行分析，可以通过命令行、IDEA、Gradle 命令三种方式运行 lint 工具。

lint Output

Lint 扫描的输出结果。

从上面可以看出，Lint Tool 就像一个加工厂，对投入进来的原料（源代码）进行加工处理（各种检测器分析），得到最终的产品（扫描结果）。Lint Tool 作为一个扫

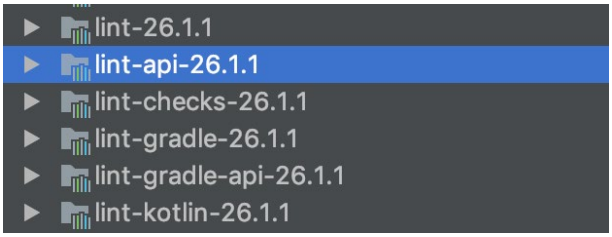
描工具集，有多种使用方式。Android 为我们提供了三种运行方式，分别是命令行、IDEA、Gradle 任务。这三种方式最终都殊途同归，通过 LintDriver 来实现扫描。如下图所示：



为了方便查看源码，新建一个工程，在 build.gradle 脚本中，添加如下依赖：

```
compile 'com.android.tools.build:gradle:3.1.1'
compile 'com.android.tools.lint:lint-gradle:26.1.1'
```

我们可以得到如下所示的依赖：



lint-api-26.1.1

Lint 工具集的一个封装，实现了一组 API 接口，用于启动 Lint；

lint-checks-26.1.1

一组内建的检测器，用于对这种描述好 Issue 进行分析处理；

lint-26.1.1

可以看做是依赖上面两个 jar 形成的一个基于命令行的封装接口形成的脚手架工程，我们的命令行、Gradle 任务都是继承自这个 jar 包中相关类来做的实现；

lint-gradle-26.1.1

可以看做是针对 Gradle 任务这种运行方式，基于 lint-26.1.1 做了一些封装类；

lint-gradle-api-26.1.1

真正 Gradle Lint 任务在执行时调用的入口；

在理解清楚了以上几个 jar 的关系和作用之后，我们可以发现 Lint 的核心库其实是前三个依赖。后面两个其实是基于脚手架，对 Gradle 这种运行方式做的封装。最核心的逻辑在 LintDriver 的 Analyze 方法中。

```
fun analyze() {

    ... 省略部分代码 ...

    for (project in projects) {
        fireEvent(EventType.REGISTERED_PROJECT, project = project)
    }
    registerCustomDetectors(projects)

    ... 省略部分代码 ...

    try {
        for (project in projects) {
            phase = 1

            val main = request.getMainProject(project)

            // The set of available detectors varies between projects
            computeDetectors(project)

            if (applicableDetectors.isEmpty()) {
```

```
                // No detectors enabled in this project: skip it
                continue
            }

            checkProject(project, main)
            if (isCanceled) {
                break
            }

            runExtraPhases(project, main)
        }
    } catch (throwable: Throwable) {
        // Process canceled etc
        if (!handleDetectorError(null, this, throwable)) {
            cancel()
        }
    }
    ... 省略部分代码 ...
}
```

主要是以下三个重要步骤：

- registerCustomDetectors(projects)

Lint 为我们提供了许多内建的检测器，除此之外我们还可以自定义一些检测器，这些都需要注册进 Lint 工具用于对目标文件进行扫描。这个方法主要做以下几件事情：

1. 遍历每一个 Project 和它的依赖 Library 工程，通过 client.findRuleJars 来找出自定义的 jar 包；
2. 通过 client.findGlobalRuleJars 找出全局的自定义 jar 包，可以作用于每一个 Android 工程；
3. 从找到的 jarFiles 列表中，解析出自定义的规则，并与内建的 Registry 一起合并为 CompositeIssueRegistry；需要注意的是，自定义的 Lint 的 jar 包存放位置是 build/intermediaters/lint 目录，如果是需要每一个工程都生效，则存放位置为 ~/.android/lint/。

- computeDetectors(project)

这一步主要用来收集当前工程所有可用的检测器。

checkProject(project, main) 接下来这一步是最为关键的一步。在此方法中，

调用 runFileDetectors 来进行文件扫描。Lint 支持的扫描文件类型很多，因为是官方支持，所以针对 Android 工程支持的比较友好。一次 Lint 任务运行时，Lint 的扫描范围主要由 Scope 来描述。具体表现在：

```
fun infer(projects: Collection<Project>?): EnumSet<Scope> {
            if (projects == null || projects.isEmpty()) {
                return Scope.ALL
            }

            // Infer the scope
            var scope = EnumSet.noneOf(Scope::class.java)
            for (project in projects) {
                val subset = project.subset
                if (subset != null) {
                    for (file in subset) {
                        val name = file.name
                        if (name == ANDROID_MANIFEST_XML) {
                            scope.add(MANIFEST)
                        } else if (name.endsWith(DOT_XML)) {
                            scope.add(RESOURCE_FILE)
                        } else if (name.endsWith(DOT_JAVA) || name.
endsWith(DOT_KT)) {
                            scope.add(JAVA_FILE)
                        } else if (name.endsWith(DOT_CLASS)) {
                            scope.add(CLASS_FILE)
                        } else if (name.endsWith(DOT_GRADLE)) {
                            scope.add(GRADLE_FILE)
                        } else if (name == OLD_PROGUARD_FILE || name ==
FN_PROJECT_PROGUARD_FILE)
{
                            scope.add(PROGUARD_FILE)
                        } else if (name.endsWith(DOT_PROPERTIES)) {
                            scope.add(PROPERTY_FILE)
                        } else if (name.endsWith(DOT_PNG)) {
                            scope.add(BINARY_RESOURCE_FILE)
                        } else if (name == RES_FOLDER || file.parent ==
RES_FOLDER) {
                            scope.add(ALL_RESOURCE_FILES)
                            scope.add(RESOURCE_FILE)
                            scope.add(BINARY_RESOURCE_FILE)
                            scope.add(RESOURCE_FOLDER)
                        }
                    }
                } else {
                    // Specified a full project: just use the full
project scope
                    scope = Scope.ALL
```

```
                    break
                }
            }
}
```

可以看到，如果 Project 的 Subset 为 Null，Scope 就为 Scope.ALL，表示本次扫描会针对能检测的所有范围，相应地在扫描时也会用到所有全部的 Detector 来扫描文件。

如果 Project 的 Subset 不为 Null，就遍历 Subset 的集合，找出 Subset 中的文件分别对应哪些范围。其实到这里我们已经可以知道，Subset 就是我们增量扫描的突破点。接下来我们看一下 runFileDetectors：

```
if(scope.contains(Scope.JAVA_FILE)||scope.contains(Scope.ALL_JAVA_FILES)){
  val checks = union(scopeDetectors[Scope.JAVA_
FILE],scopeDetectors[Scope.ALL_JAVA_FILES])
  if (checks != null && !checks.isEmpty()) {
    val files = project.subset
    if (files != null) {
      checkIndividualJavaFiles(project, main, checks, files)
    } else {
      val sourceFolders = project.javaSourceFolders
      val testFolders = if (scope.contains(Scope.TEST_SOURCES))
      project.testSourceFolders
      else
      emptyList<File> ()
      val generatedFolders = if (isCheckGeneratedSources)
      project.generatedSourceFolders
      else
      emptyList<File> ()
      checkJava(project, main, sourceFolders, testFolders,
generatedFolders, checks)
    }
  }
}
```

这里更加明确，如果 project.subset 不为空，就对单独的 Java 文件扫描，否则，就对源码文件和测试目录以及自动生成的代码目录进行扫描。整个 runFileDetectors 的扫描顺序入下：

1. Scope.MANIFEST
2. Scope.ALL_RESOURCE_FILES)|| scope.contains(Scope.RESOURCE_FILE) || scope.contains(Scope.RESOURCE_FOLDER) || scope.contains(Scope.BINARY_RESOURCE_FILE)
3. scope.contains(Scope.JAVA_FILE) || scope.contains(Scope.ALL_JAVA_FILES)
4. scope.contains(Scope.CLASS_FILE) || scope.contains(Scope.ALL_CLASS_FILES) || scope.contains(Scope.JAVA_LIBRARIES)
5. scope.contains(Scope.GRADLE_FILE)
6. scope.contains(Scope.OTHER)
7. scope.contains(Scope.PROGUARD_FILE)
8. scope.contains(Scope.PROPERTY_FILE)

与官方文档的描述顺序一致。

现在我们已经知道，增量扫描的突破点其实是需要构造 project.subset 对象。

```
/**
     * Adds the given file to the list of files which should be
checked in this
     * project. If no files are added, the whole project will be
checked.
     *
     * @param file the file to be checked
     */
    public void addFile(@NonNull File file) {
        if (files == null) {
            files = new ArrayList<>();
        }
        files.add(file);
    }

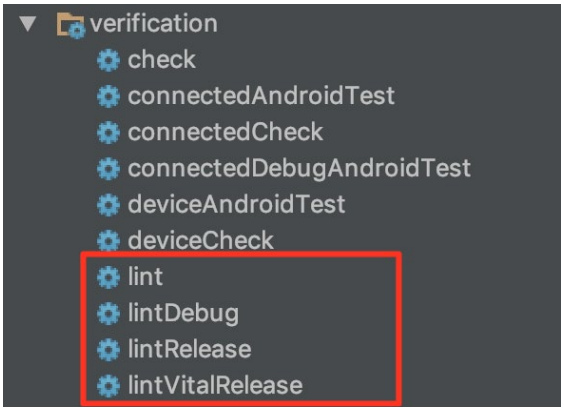
    /**
     * The list of files to be checked in this project. If null, the
whole
     * project should be checked.
     *
     * @return the subset of files to be checked, or null for the
whole project
     */
```

```
    @Nullable
    public List<File> getSubset() {
        return files;
    }
```

注释也很明确的说明了只要 Files 不为 Null，就会扫描指定文件，否则扫描整个工程。

Lint 增量扫描 Gradle 任务实现

前面分析了如何获取差异文件以及增量扫描的原理，分析的重点还是侧重在 Lint 工具本身的实现机制上。接下来分析，在 Gradle 中如何实现一个增量扫描任务。大家知道，通过执行 ./gradlew lint 命令来执行 Lint 静态代码检测任务。创建一个新的 Android 工程，在 Gradle 任务列表中可以在 Verification 这个组下面找到几个 Lint 任务，如下所示：



这几个任务就是 Android Gradle 插件在加载的时候默认创建的。分别对应于以下几个 Task：

- lint->LintGlobalTask：由 TaskManager 创建；
- lintDebug、lintRelease、lintVitalRelease->LintPerVariantTask： 由 ApplicationTaskManager 或者 LibraryTaskManager 创建，其中 lintVital-Release 只在 release 下生成；

所以，在 Android Gradle 插件中，应用于 Lint 的任务分别为 LintGlobalTask 和 LintPerVariantTask。他们的区别是前者执行的是扫描所有 Variant，后者执行只针对单独的 Variant。而我们的增量扫描任务其实是跟 Variant 无关的，因为我们会把所有差异文件都收集到。无论是 LintGlobalTask 或者是 LintPerVariantTask，都继承自 LintBaseTask。最终的扫描任务在 LintGradleExecution 的 runLint 方法中执行，这个类位于 lint-gradle-26.1.1 中，前面提到这个库是基于 Lint 的 API 针对 Gradle 任务做的一些封装。

```
/** Runs lint on the given variant and returns the set of warnings */
    private Pair<List<Warning>, LintBaseline> runLint(
            @Nullable Variant variant,
            @NonNull VariantInputs variantInputs,
            boolean report, boolean isAndroid) {
        IssueRegistry registry = createIssueRegistry(isAndroid);
        LintCliFlags flags = new LintCliFlags();
        LintGradleClient client =
                new LintGradleClient(
                        descriptor.getGradlePluginVersion(),
                        registry,
                        flags,
                        descriptor.getProject(),
                        descriptor.getSdkHome(),
                        variant,
                        variantInputs,
                        descriptor.getBuildTools(),
                        isAndroid);
        boolean fatalOnly = descriptor.isFatalOnly();
        if (fatalOnly) {
            flags.setFatalOnly(true);
        }
        LintOptions lintOptions = descriptor.getLintOptions();
        if (lintOptions != null) {
            syncOptions(
                    lintOptions,
                    client,
                    flags,
                    variant,
                    descriptor.getProject(),
                    descriptor.getReportsDir(),
                    report,
                    fatalOnly);
        } else {
            // Set up some default reporters
```

```
            flags.getReporters().add(Reporter.createTextReporter(client,
flags, null,
                    new PrintWriter(System.out, true), false));
            File html = validateOutputFile(createOutputPath(descriptor.
getProject(), null, ".html",
                    null, flags.isFatalOnly()));
            File xml = validateOutputFile(createOutputPath(descriptor.
getProject(), null, DOT_XML,
                    null, flags.isFatalOnly()));
            try {
                flags.getReporters().add(Reporter.
createHtmlReporter(client, html, flags));
                flags.getReporters().add(Reporter.
createXmlReporter(client, xml, false));
            } catch (IOException e) {
                throw new GradleException(e.getMessage(), e);
            }
        }
        if (!report || fatalOnly) {
            flags.setQuiet(true);
        }
        flags.setWriteBaselineIfMissing(report && !fatalOnly);

        Pair<List<Warning>, LintBaseline> warnings;
        try {
            warnings = client.run(registry);
        } catch (IOException e) {
            throw new GradleException("Invalid arguments.", e);
        }

        if (report && client.haveErrors() && flags.isSetExitCode()) {
            abort(client, warnings.getFirst(), isAndroid);
        }

        return warnings;
    }
```

我们在这个方法中看到了 warnings = client.run(registry)，这就是 Lint 扫描得到的结果集。总结一下这个方法中做了哪些准备工作用于 Lint 扫描：1. 创建 IssueRegistry，包含了 Lint 内建的 BuiltinIssueRegistry；2. 创建 LintCliFlags；3. 创建 LintGradleClient，这里面传入了一大堆参数，都是从 Gradle Android 插件的运行环境中获得；4. 同步 LintOptions，这一步是将我们在 build.gralde 中配置的一些 Lint 相关的 DSL 属性，同步设置给 LintCliFlags，给真正的 Lint 扫描核心库使

用；5. 执行 Client 的 Run 方法，开始扫描。

扫描的过程上面的原理部分已经分析了，现在我们思考一下如何构造增量扫描的任务。我们已经分析到扫描的关键点是 client.run(registry)，所以我们需要构造一个 Client 来执行扫描。一个想法是通过反射来获取 Client 的各个参数，当然这个思路是可行的，我们也验证过实现了一个用反射方式构造的 Client。但是反射这种方式有个问题是丢失了从 Gradle 任务执行到调用 Lint API 开始扫描这一过程中做的其他事情，侵入性比较高，所以我们最终采用继承 LintBaseTask 自行实现增量扫描任务的方式。

FindBugs 扫描简介

FindBugs 是一个静态分析工具，它检查类或者 JAR 文件，通过 Apache 的 BCEL 库来分析 Class，将字节码与一组缺陷模式进行对比以发现问题。FindBugs 自身定义了一套缺陷模式，目前的版本 3.0.1 内置了总计 300 多种缺陷，详细可参考官方文档。FindBugs 作为一个扫描的工具集，可以非常灵活的集成在各种编译工具中。接下来，我们主要分析在 Gradle 中 FindBugs 的相关内容。

Gradle FindBugs 任务属性分析

在 Gradle 的内置任务中，有一个 FindBugs 的 Task，我们看一下官方文档对 Gradle 属性的描述。

选几个比较重要的属性介绍：

- Classes 该属性表示我们要分析的 Class 文件集合，通常我们会把编译结果的 Class 目录用于扫描。
- Classpath 分析目标集合中的 Class 需要用到的所有相关的 Classes 路径，但是并不会分析它们自身，只用于扫描。
- Effort 包含 MIN，Default，MAX，级别越高，分析得越严谨越耗时。
- findBugs ClasspathFinbugs 库相关的依赖路径，用于配置扫描的引擎库。
- reportLevel 报告级别，分为 Low，Medium，High。如果为 Low，所有

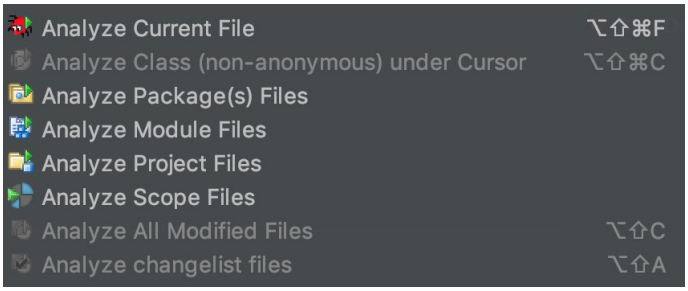
Bug 都报告，如果为 High，仅报告 High 优先级。

- Reports 扫描结果存放路径。

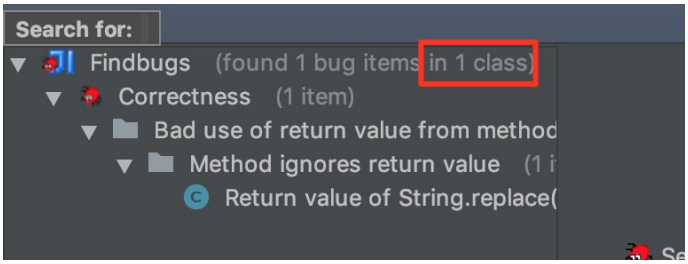
通过以上属性解释，不难发现要 FindBugs 增量扫描，只需要指定 Classes 的文件集合就可以了。

FindBugs 任务增量扫描分析

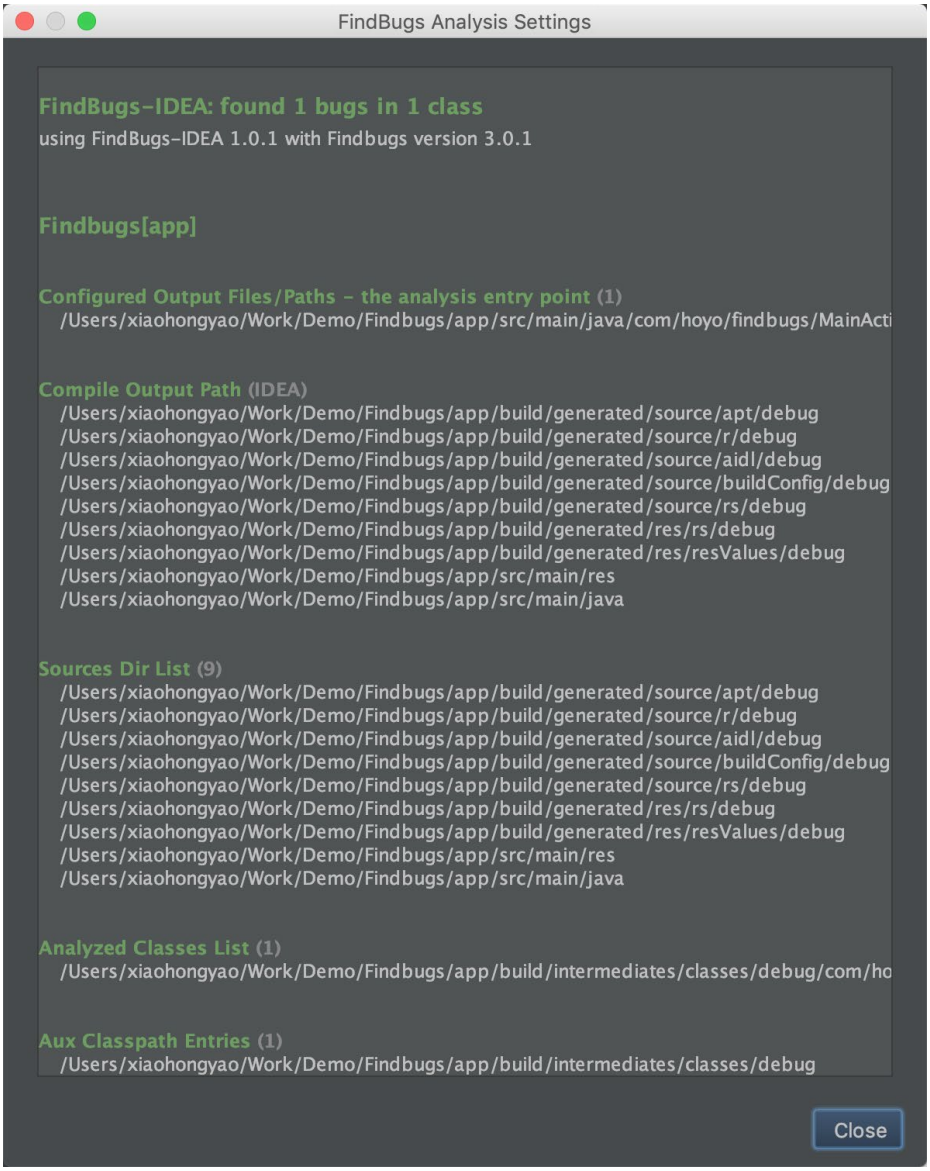
在做增量扫描任务之前，我们先来看一下 FindBugs IDEA 插件是如何进行单个文件扫描的。



我们选择 Analyze Current File 对当前文件进行扫描，扫描结果如下所示：



可以看到确实只扫描了一个文件。那么扫描到底使用了哪些输入数据呢，我们可以通过扫描结果的提示清楚看到：



这里我们能看到很多有用的信息：

- 源码目录列表，包含了工程中的 Java 目录，res 目录，以及编译过程中生成的一些类目录；
- 需要分析的目标 Class 集合，为编译后的 Build 目录下的当前 Java 文件对应的 Class 文件；

- Aux Classpath Entries, 表示分析上面的目标文件需要用到的类路径。

所以，根据 IDEA 的扫描结果来看，我们在做增量扫描的时候需要解决上面这几个属性的获取。在前面我们分析的属性是 Gradle 在 FindBugs lib 的基础上，定义的一套对应的 Task 属性。真正的 FinBugs 属性我们可以通过官方文档或者源码中查到。

配置 AuxClasspath

前文提到，ClassPath 是用来分析目标文件需要用到的相关依赖 Class，但本身并不会被分析，所以我们需要尽可能全的找到所有的依赖库，否则在扫描的时候会报依赖的类库找不到。

```
FileCollection buildClasses = project.fileTree(dir: "${project.
buildDir}/intermediates/classes/${variant.flavorName}/${variant.
buildType.name}",includes: classIncludes)

FileCollection targetClasspath = project.files()
GradleUtils.collectDepProject(project, variant).each { targetProject ->
    GradleUtils.getAndroidVariants(targetProject).each { targetVariant ->
        if (targetVariant.name.capitalize().equalsIgnoreCase(variant.
name.capitalize())) {
            targetClasspath += targetVariant.javaCompile.classpath
        }
    }
}

classpath = variant.javaCompile.classpath + targetClasspath +
buildClasses
```

FindBugs 增量扫描误报优化

对于增量文件扫描，参与的少数文件扫描在某些模式规则上可能会出现误判，但是全量扫描不会有问题，因为参与分析的目标文件是全集。举一个例子：

```
class A {
  public static String buildTime = "";
  ....
}
```

静态变量 buildTime 会被认为应该加上 Final，但是其实其他类会对这个变量赋值。如果单独扫描类 A 文件，就会报缺陷 BUG_TYPE_MS_SHOULD_BE_FINAL。我们通过 Findbugs-IDEA 插件来扫描验证，也同样会有一样的问题。要解决此类问题，需要找到谁依赖了类 A，并且一同参与扫描，同时也需要找出类 A 依赖了哪些文件，简单来说：需要找出与类 A 有直接关联的类。为了解决这个问题，我们通过 ASM 来找出相关的依赖，具体如下：

```
void findAllScanClasses(ConfigurableFileTree allClass) {
    allScanFiles = [] as HashSet
    String buildClassDir = "${project.buildDir}/$FINDBUGS_ANALYSIS_
DIR/$FINDBUGS_
ANALYSIS_DIR_ORIGIN"

    Set<File> moduleClassFiles = allClass.files
    for (File file : moduleClassFiles) {
        String[] splitPath = file.absolutePath.split("$FINDBUGS_
ANALYSIS_DIR/$FINDBUGS_
ANALYSIS_DIR_ORIGIN/")
        if (splitPath.length > 1) {
            String className = getFileNameNoFlag(splitPath[1],'.')
            String innerClassPrefix = ""
            if (className.contains('$')) {
                innerClassPrefix = className.split('\\$')[0]
            }
            if (diffClassNamePath.contains(className) ||
diffClassNamePath.contains(innerClassPrefix))
{
                allScanFiles.add(file)
            } else {
                Iterable<String> classToResolve = new
ArrayList<String>()
                classToResolve.add(file.absolutePath)
                Set<File> dependencyClasses = Dependencies.
findClassDependencies(project, new
ClassAcceptor(), buildClassDir, classToResolve)
                for (File dependencyClass : dependencyClasses) {
                    if (diffClassNamePath.
contains(getPackagePathName(dependencyClass))) {
                        allScanFiles.add(file)
                        break
                    }
                }
            }
        }
    }
}
```

通过以上方式，我们可以解决一些增量扫描时出现的误报情况，相比 IDEA 工具，我们更进一步降低了扫描部分文件的误报率。

CheckStyle 增量扫描

相比而言，CheckStyle 的增量扫描就比较简单了。CheckStyle 对源码扫描，根据官方文档各个属性的描述，我们发现只要指定 Source 属性的值就可以指定扫描的目标文件。

```
void configureIncrementScanSource() {
    boolean isCheckPR = false
    DiffFileFinder diffFileFinder

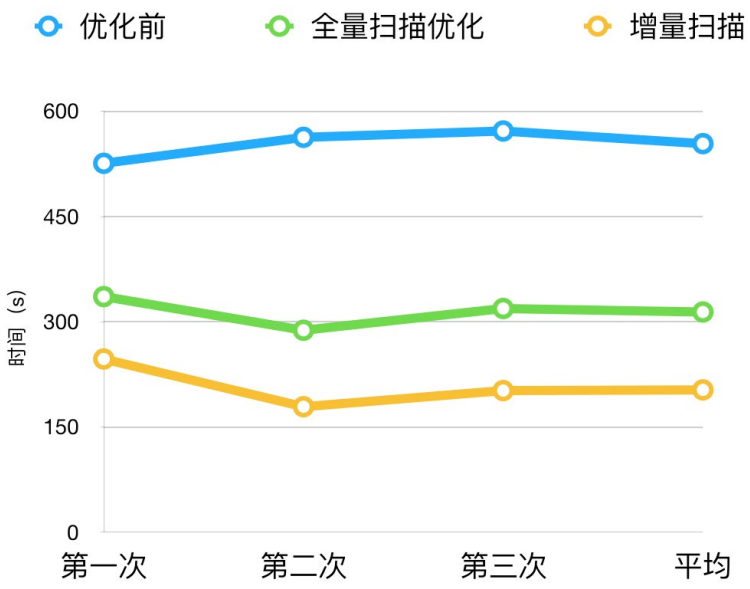
    if (project.hasProperty(CodeDetectorExtension.CHECK_PR)) {
        isCheckPR = project.getProperties().get(CodeDetectorExtension.
CHECK_PR)
    }

    if (isCheckPR) {
        diffFileFinder = new DiffFileFinderHelper.PRDiffFileFinder()
    } else {
        diffFileFinder = new DiffFileFinderHelper.LocalDiffFileFinder()
    }

    source diffFileFinder.findDiffFiles(project)

    if (getSource().isEmpty()) {
        println '没有找到差异 java 文件，跳过 checkStyle 检测'
    }
}
```

经过全量扫描和增量扫描的优化，我们整个扫描效率得到了很大提升，一次 PR 构建扫描效率整体提升 50%+。优化数据如下：



扫描工具通用性

解决了扫描效率问题，我们想怎么让更多的工程能低成本的使用这个扫描插件。对于一个已经存在的工程，如果没有使用过静态代码扫描，我们希望在接入扫描插件后续新增的代码能够保证其经过增量扫描没有问题。而老的存量代码，由于代码量过大增量扫描并没有效率上的优势，我们希望可以使用全量扫描逐步解决存量代码存在的问题。同时，为了配置工具的灵活，也提供配置来让接入方自己决定选择接入哪些工具。这样可以让扫描工具同时覆盖到新老项目，保证其通用。所以，要同时支持配置使用增量或者全量扫描任务，并且提供灵活的选择接入哪些扫描工具。

扫描完整性保证

前面提到过，在 FindBugs 增量扫描可能会出现因为参与分析的目标文件集不全导致的某类匹配规则误报，所以在保证扫描效率的同时，也要保证扫描的完整性和准确性。我们的策略是以增量扫描为主，全量扫描为辅，PR 提交使用增量扫描提高效率，在 CI 配置 Daily Build 使用全量扫描保证扫描完整和不遗漏。

我们在自己的项目中实践配置如下：

```
apply plugin: 'code-detector'

codeDetector {
    // 配置静态代码检测报告的存放位置
    reportRelativePath = rootProject.file('reports')

    /**
     * 远程仓库地址，用于配置提交 pr 时增量检测
     */
    upstreamGitUrl = "ssh://git@xxxxxxxx.git"

    checkStyleConfig {
        /**
         * 开启或关闭 CheckStyle 检测
         * 开启：true
         * 关闭：false
         */
        enable = true
        /**
         * 出错后是否要终止检查
         * 终止：false
         * 不终止：true。配置成不终止的话 CheckStyleTask 不会失败，也不会拷贝错误
报告
         */
        ignoreFailures = false
        /**
         * 是否在日志中展示违规信息
         * 显示：true
         * 不显示：false
         */
        showViolations = true
        /**
         * 统一配置自定义的 checkstyle.xml 和 checkstyle.xsl 的 uri
         * 配置路径为：
         *      "${checkStyleUri}/checkstyle.xml"
         *      "${checkStyleUri}/checkstyle.xsl"
         *
         * 默认为 null，使用 CodeDetector 中的默认配置
         */
        checkStyleUri = rootProject.file('codequality/checkstyle')
    }

    findBugsConfig {
        /**
         * 开启或关闭 Findbugs 检测
         * 开启：true
```

```
            * 关闭：false
            */
           enable = true
           /**
            * 可选项，设置分析工作的等级，默认值为 max
            * min, default, or max. max 分析更严谨，报告的 bug 更多 . min 略微少些
            */
           effort = "max"
           /**
            * 可选项，默认值为 high
            * low, medium, high. 如果是 low 的话，那么报告所有的 bug
            */
           reportLevel = "high"
           /**
            * 统一配置自定义的 findbugs_include.xml 和 findbugs_exclude.xml 的 uri
            * 配置路径为：
            *       "${findBugsUri}/findbugs_include.xml"
            *       "${findBugsUri}/findbugs_exclude.xml"
            * 默认为 null，使用 CodeDetector 中的默认配置
            */
           findBugsUri = rootProject.file('codequality/findbugs')
       }

       lintConfig {

           /**
            * 开启或关闭 lint 检测
            * 开启：true
            * 关闭：false
            */
           enable = true

           /**
            * 统一配置自定义的 lint.xml 和 retrolambda_lint.xml 的 uri
            * 配置路径为：
            *       "${lintConfigUri}/lint.xml"
            *       "${lintConfigUri}/retrolambda_lint.xml"
            * 默认为 null，使用 CodeDetector 中的默认配置
            */
           lintConfigUri = rootProject.file('codequality/lint')
       }
   }
```

我们希望扫描插件可以灵活指定增量扫描还是全量扫描以应对不同的使用场景，比如已存在项目的接入、新项目的接入、打包时的检测等。

执行脚本示例：

```
./gradlew ":${appModuleName}:assemble${ultimateVariantName}"
-PdetectorEnable=true
-PcheckStyleIncrement=true -PlintIncrement=true -PfindBugsIncrement=true
-PcheckPR=${checkPR} -PsourceCommitHash=${sourceCommitHash}
-PtargetBranch=${targetBranch} --stacktrace
```

希望一次任务可以暴露所有扫描工具发现的问题，当某一个工具扫描到问题后不终止任务，如果是本地运行在发现问题后可以自动打开浏览器方便查看问题原因。

```
def finalizedTaskArray = [lintTask,checkStyleTask,findbugsTask]
checkCodeTask.finalizedBy finalizedTaskArray

"open ${reportPath}".execute()
```

为了保证提交的 PR 不会引起打包问题影响包的交付，在 PR 时触发的任务实际为打包任务，我们将静态代码扫描任务挂接在打包任务中。由于我们的项目是多 Flavor 构建，在 CI 上我们将触发多个 Job 同时执行对应 Flavor 的增量扫描和打包任务。同时为了保证代码扫描的完整性，我们在真正的打包 Job 上执行全量扫描。

本文主要介绍了在静态代码扫描优化方面的一些思路与实践，并重点探讨了对 Lint、Finbugs、CheckStyle 增量扫描的一些尝试。通过对扫描插件的优化，我们在代码扫描的效率上得到了提升，同时在实践过程中我们也积累了自定义 Lint 检测规则的方案，未来我们将配合基础设施标准化建设，结合静态扫描插件制定一些标准化检测规则来更好的保证我们的代码规范以及质量。

参考资料

- CheckStyle 插件文档
- FindBugs 插件文档
- Android Lint 开发介绍
- Lint 增量扫描
- FindBugs 配置属性介绍
- 美团外卖 Android Lint 代码检查实践
- Gradle 源码
- 静态代码检测工具对比
- Android Gradle 插件源码

作者简介

鸿耀，美团餐饮生态技术团队研发工程师。

招聘

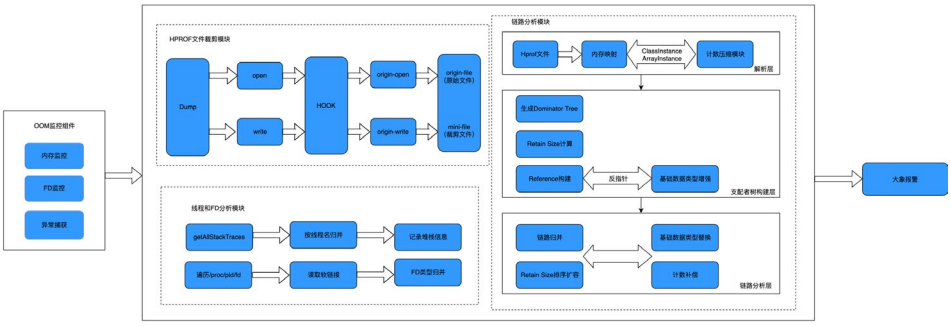
美团餐饮生态技术团队诚招 Android、Java 后端高级 / 资深工程师和技术专家，Base 成都，欢迎有兴趣的同学投递简历到 tech@meituan.com。

Probe：Android 线上 OOM 问题定位组件

逢搏　毅然　永刚

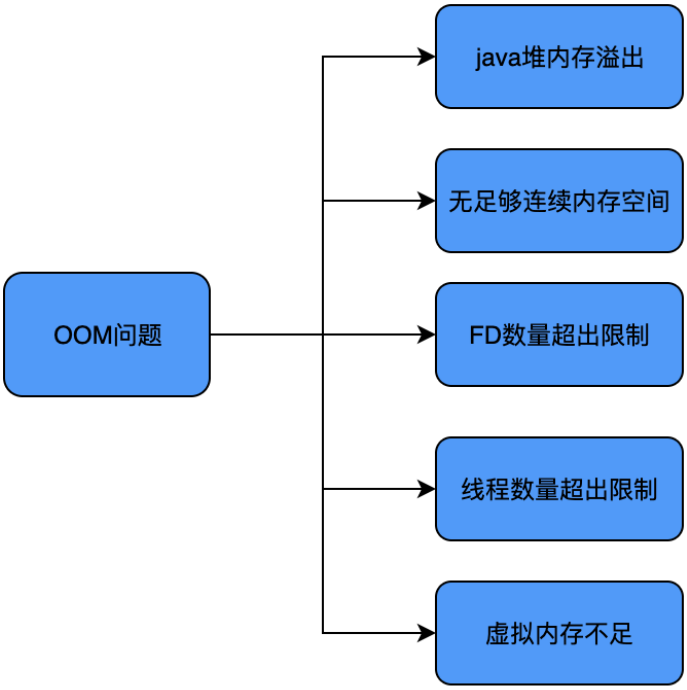
配送骑手端 App 是骑手用于完成配送履约的应用，帮助骑手完成接单、到店、取货及送达，提供各种不同的运力服务，也是整个外卖闭环中的重要节点。由于配送业务的特性，骑手 App 对于应用稳定性的要求非常高，体现 App 稳定性的一个重要数据就是 Crash 率，而在众多 Crash 中最棘手最难定位的就是 OOM 问题。对于骑手端 App 而言，每天骑手都会长时间的使用 App 进行配送，而在长时间的使用过程中，App 中所有的内存泄漏都会慢慢累积在内存中，最后就容易导致 OOM，从而影响骑手的配送效率，进而影响整个外卖业务。

于是我们构建了用于快速定位线上 OOM 问题的组件——Probe，下图是 Probe 组件架构，本文主要分享 Probe 组件是如何对线上 OOM 问题进行快速定位的。



OOM 原因分析

要定位 OOM 问题，首先需要弄明白 Android 中有哪些原因会导致 OOM，Android 中导致 OOM 的原因主要可以划分为以下几个类型：



Android 虚拟机最终抛出 OutOfMemoryError 的代码位于 /art/runtime/thread.cc。

```
void Thread::ThrowOutOfMemoryError(const char* msg)
参数 msg 携带了 OOM 时的错误信息
```

下面两个地方都会调用上面方法抛出 OutOfMemoryError 错误，这也是 Android 中发生 OOM 的主要原因。

堆内存分配失败

系统源码文件：/art/runtime/gc/heap.cc

```
void Heap::ThrowOutOfMemoryError(Thread* self, size_t byte_count,
AllocatorType allocator_
type)
抛出时的错误信息：
    oss << "Failed to allocate a " << byte_count << " byte allocation
with " << total_bytes_free
<< " free bytes and " << PrettySize(GetFreeMemoryUntilOOME()) << " until OOM";
```

这是在进行堆内存分配时抛出的 OOM 错误，这里也可以细分成两种不同的类型：

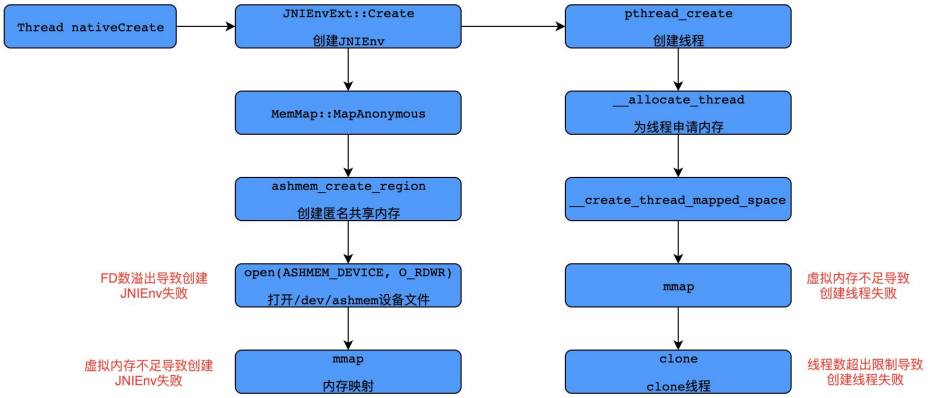
1. 为对象分配内存时达到进程的内存上限。由 Runtime.getRuntime.Max-Memory() 可以得到 Android 中每个进程被系统分配的内存上限，当进程占用内存达到这个上限时就会发生 OOM，这也是 Android 中最常见的 OOM 类型。
2. 没有足够大小的连续地址空间。这种情况一般是进程中存在大量的内存碎片导致的，其堆栈信息会比第一种 OOM 堆栈多出一段信息：failed due to fragmentation (required continguous free “<< required_bytes << “bytes for a new buffer where largest contiguous free” << largest_continuous_free_pages << “ bytes)”；其详细代码在 art/runtime/gc/allocator/rosalloc.cc 中，这里不作详述。

创建线程失败

系统源码文件：/art/runtime/thread.cc

```
void Thread::CreateNativeThread(JNIEnv* env, jobject java_peer, size_t
stack_size, bool is_
daemon)
抛出时的错误信息：
    "Could not allocate JNI Env"
  或者
    StringPrintf("pthread_create (%s stack) failed: %s",
PrettySize(stack_size).c_str(),
strerror(pthread_create_result)));
```

这是创建线程时抛出的 OOM 错误，且有多种错误信息。源码这里不展开详述了，下面是根据源码整理的 Android 中创建线程的步骤，其中两个关键节点是创建 JNIEnv 结构体和创建线程，而这两步均有可能抛出 OOM。



创建 JNI 失败

创建 JNIEnv 可以归为两个步骤：

- 通过 Andorid 的匿名共享内存（Anonymous Shared Memory）分配 4KB（一个 page）内核态内存。
- 再通过 Linux 的 mmap 调用映射到用户态虚拟内存地址空间。

第一步创建匿名共享内存时，需要打开 /dev/ashmem 文件，所以需要一个 FD（文件描述符）。此时，如果创建的 FD 数已经达到上限，则会导致创建 JNIEnv 失败，抛出错误信息如下：

```
E/art: ashmem_create_region failed for 'indirect ref table': Too many
open files
 java.lang.OutOfMemoryError: Could not allocate JNI Env
   at java.lang.Thread.nativeCreate(Native Method)
   at java.lang.Thread.start(Thread.java:730)
```

第二步调用 mmap 时，如果进程虚拟内存地址空间耗尽，也会导致创建 JNIEnv 失败，抛出错误信息如下：

```
E/art: Failed anonymous mmap(0x0, 8192, 0x3, 0x2, 116, 0): Operation
not permitted. See
process maps in the log.
java.lang.OutOfMemoryError: Could not allocate JNI Env
  at java.lang.Thread.nativeCreate(Native Method)
```

```
    at java.lang.Thread.start(Thread.java:1063)
```

创建线程失败

创建线程也可以归纳为两个步骤：

1. 调用 mmap 分配栈内存。这里 mmap flag 中指定了 MAP_ANONYMOUS，即匿名内存映射。这是在 Linux 中分配大块内存的常用方式。其分配的是虚拟内存，对应页的物理内存并不会立即分配，而是在用到的时候触发内核的缺页中断，然后中断处理函数再分配物理内存。
2. 调用 clone 方法进行线程创建。

第一步分配栈内存失败是由于进程的虚拟内存不足，抛出错误信息如下：

```
W/libc: pthread_create failed: couldn't allocate 1073152-bytes mapped
space: Out of memory
W/tch.crowdsourc: Throwing OutOfMemoryError with VmSize  4191668 kB
"pthread_create
(1040KB stack) failed: Try again"
java.lang.OutOfMemoryError: pthread_create (1040KB stack) failed: Try
again
        at java.lang.Thread.nativeCreate(Native Method)
        at java.lang.Thread.start(Thread.java:753)
```

第二步 clone 方法失败是因为线程数超出了限制，抛出错误信息如下：

```
W/libc: pthread_create failed: clone failed: Out of memory
W/art: Throwing OutOfMemoryError "pthread_create (1040KB stack) failed:
Out of memory"
java.lang.OutOfMemoryError: pthread_create (1040KB stack) failed: Out
of memory
  at java.lang.Thread.nativeCreate(Native Method)
  at java.lang.Thread.start(Thread.java:1078)
```

OOM 问题定位

在分析清楚 OOM 问题的原因之后，我们对于线上的 OOM 问题就可以做到对症下药。而针对 OOM 问题，我们可以根据堆栈信息的特征来确定这是哪一个类型的

OOM，下面分别介绍使用 Probe 组件是如何去定位线上发生的每一种类型的 OOM 问题的。

堆内存不足

Android 中最常见的 OOM 就是 Java 堆内存不足，对于堆内存不足导致的 OOM 问题，发生 Crash 时的堆栈信息往往只是“压死骆驼的最后一根稻草”，它并不能有效帮助我们准确地定位到问题。

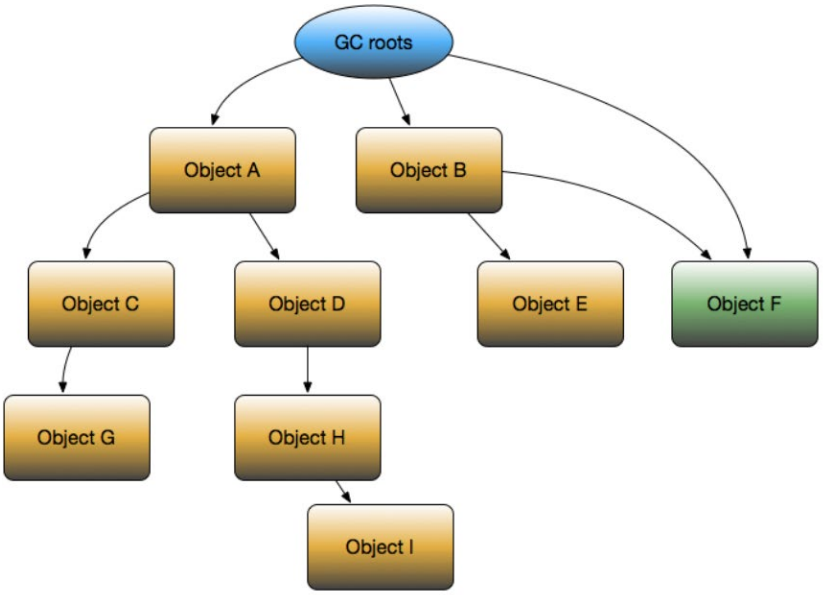
堆内存分配失败，通常说明进程中大部分的内存已经被占用了，且不能被垃圾回收器回收，一般来说此时内存占用都存在一些问题，例如内存泄漏等。要想定位到问题所在，就需要知道进程中的内存都被哪些对象占用，以及这些对象的引用链路。而这些信息都可以在 Java 内存快照文件中得到，调用 Debug.dumpHprof-Data(String fileName) 函数就可以得到当前进程的 Java 内存快照文件（即 HPROF 文件）。所以，关键在于要获得进程的内存快照，由于 dump 函数比较耗时，在发生 OOM 之后再去执行 dump 操作，很可能无法得到完整的内存快照文件。

于是 Probe 对于线上场景做了内存监控，在一个后台线程中每隔 1S 去获取当前进程的内存占用（通过 Runtime.getRuntime.totalMemory()-Runtime.getRuntime.freeMemory() 计算得到），当内存占用达到设定的阈值时（阈值根据当前系统分配给应用的最大内存计算），就去执行 dump 函数，得到内存快照文件。

在得到内存快照文件之后，我们有两种思路，一种想法是直接将 HPROF 文件回传到服务器，我们拿到文件后就可以使用分析工具进行分析。另一种想法是在用户手机上直接分析 HPROF 文件，将分析完得到的分析结果回传给服务器。但这两种方案都存在着一些问题，下面分别介绍我们在这两种思路的实践过程中遇到的挑战和对应的解决方案。

线上分析

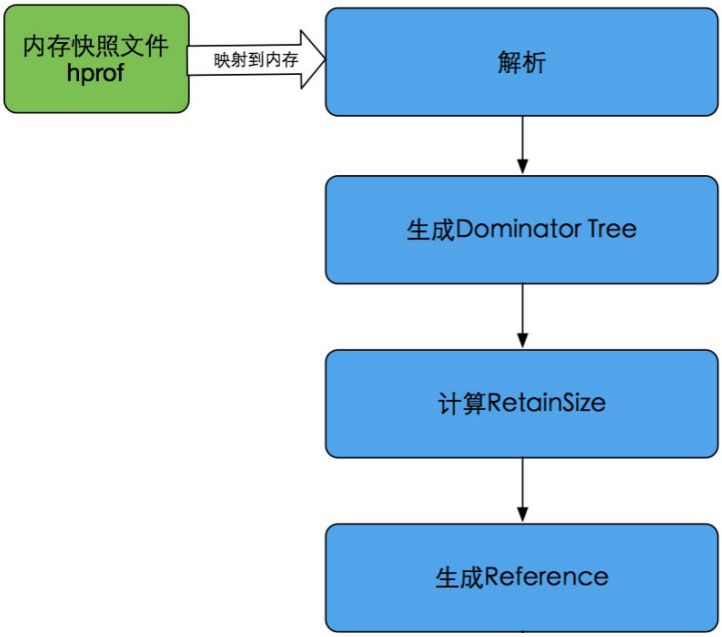
首先，我们介绍几个基本概念：



- **Dominator**：从 GC Roots 到达某一个对象时，必须经过的对象，称为该对象的 Dominator。例如在上图中，B 就是 E 的 Dominator，而 B 却不是 F 的 Dominator。
- **ShallowSize**：对象自身占用的内存大小，不包括它引用的对象。
- **RetainSize**：对象自身的 ShallowSize 和对象所支配的（可直接或间接引用到的）对象的 ShallowSize 总和，就是该对象 GC 之后能回收的内存总和。例如上图中，D 的 RetainSize 就是 D、H、I 三者的 ShallowSize 之和。

JVM 在进行 GC 的时候会进行可达性分析，当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说，就是从 GC Roots 到这个对象不可达）时，则证明此对象是可回收的。

Github 上有一个开源项目 HAHA 库，用于自动解析和分析 Java 内存快照文件（即 HPROF 文件）。下面是 HAHA 库的分析步骤：



于是我们尝试在 App 中去新开一个进程使用 HAHA 库分析 HPROF 文件，在线下测试过程中遇到了几个问题，下面逐一进行叙述。

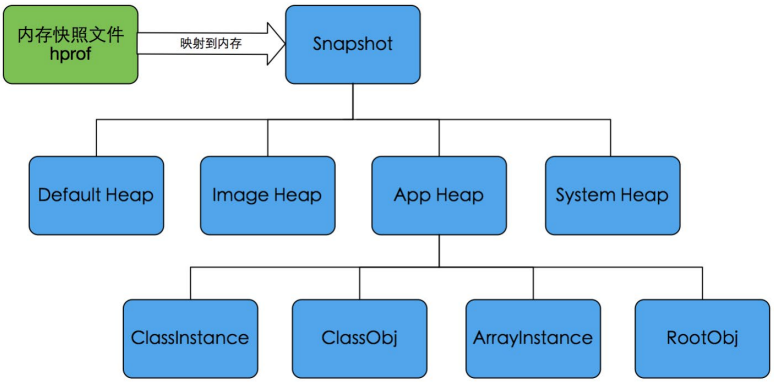
分析进程自身 OOM

测试时遇到的最大问题就是分析进程自身经常会发生 OOM，导致分析失败。为了弄清楚分析进程为什么会占用这么大内存，我们做了两个对比实验：

- 在一个最大可用内存 256MB 的手机上，让一个成员变量申请特别大的一块内存 200 多 MB，人造 OOM，Dump 内存，分析，内存快照文件达到 250 多 MB，分析进程占用内存并不大，为 70MB 左右。
- 在一个最大可用内存 256MB 的手机上，添加 200 万个小对象（72 字节），人造 OOM，Dump 内存，分析，内存快照文件达到 250 多 MB，分析进程占用内存增长很快，在解析时就发生 OOM 了。

实验说明，分析进程占用内存与 HPROF 文件中的 Instance 数量是正相关的，在将 HPROF 文件映射到内存中解析时，如果 Instance 的数量太大，就会导致 OOM。

HPROF 文件映射到内存中会被解析成 Snapshot 对象（如下图所示），它构建了一颗对象引用关系树，我们可以在这颗树中查询各个 Object 的信息，包括 Class 信息、内存地址、持有的引用以及被持有引用的关系。

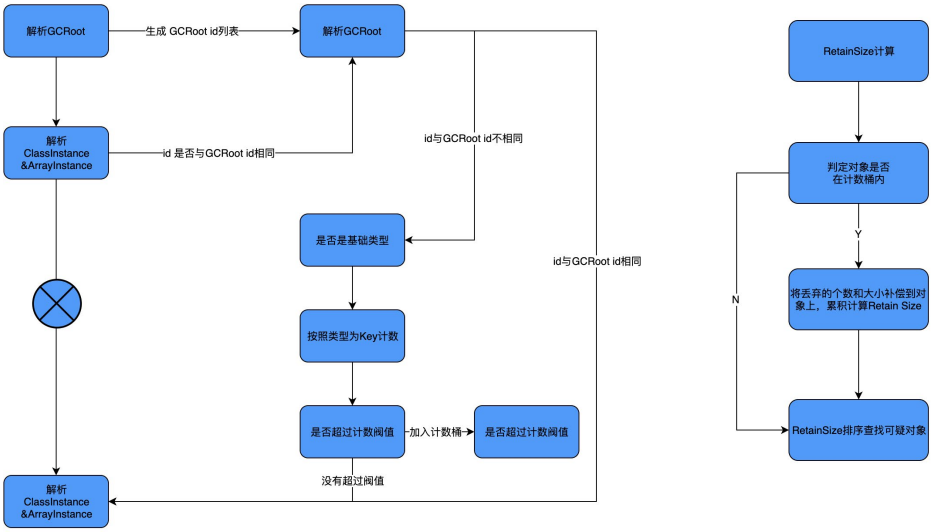


HPROF 文件映射到内存的过程：

```
// 1. 构建内存映射的 HprofBuffer 针对大文件的一种快速的读取方式，其原理是将文件流的通
道与 ByteBuffer 建立起关联，并只在真正发生读取时才从磁盘读取内容出来。
HprofBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);
// 2. 构造 Hprof 解析器
HprofParser parser = new HprofParser(buffer);
// 3. 获取快照
Snapshot snapshot = parser.parse();
// 4. 去重 gcRoots
deduplicateGcRoots(snapshot);
```

为了解决分析进程 OOM 的问题，我们在 HprofParser 的解析逻辑中加入了计数压缩逻辑（如下图），目的是在文件映射过程去控制 Instance 的数量。在解析过程中对于 ClassInstance 和 ArrayInstance，以类型为 key 进行计数，当同一类型的 Instance 数量超过阈值时，则不再向 Snapshot 中添加该类型的 Instance，只是记录 Intsance 被丢弃的数量和 Instance 大小。这样就可以控制住每一种类型的 Instance 数量，减少了分析进程的内存占用，在很大程度上避免了分析进程自身的 OOM 问题。既然我们在解析时丢弃了一部分 Instance，后面就得把丢弃的这部分找补回来，所以在计算 RetainSize 时我们会进行计数桶补偿，即把之前丢弃的相同类

型的 Instance 数量和大小都补偿到这个对象上，累积去计算 RetainSize。

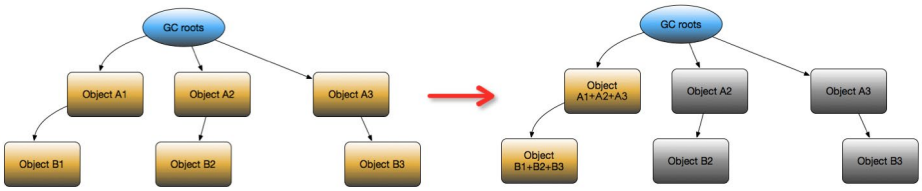


链路分析时间过长

在线下测试过程中还遇到了一个问题，就是在手机上进行链路分析的耗时太长。

使用 HAHA 算法在 PC 上可以快速地对所有对象都进行链路分析，但是在手机上由于性能的局限性，如果对所有对象都进行链路分析会导致分析耗时非常长。

考虑到 RetainSize 越大的对象对内存的影响也越大，即 RetainSize 比较大的那部分 Instance 是最有可能造成 OOM 的“元凶”。

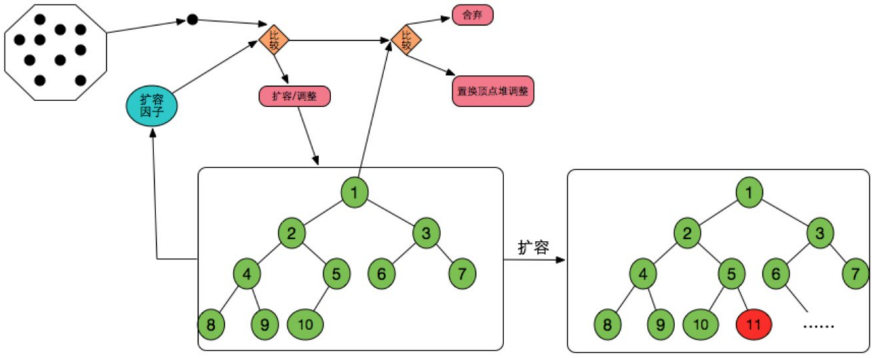


我们在生成 Reference 之后，做了一步链路归并（如上图），即对于同一个对象的不同 Instance，如果其底下的引用链路中的对象类型也相同，则进行归并，并记录 Instance 的个数和每个 Instance 的 RetainSize。

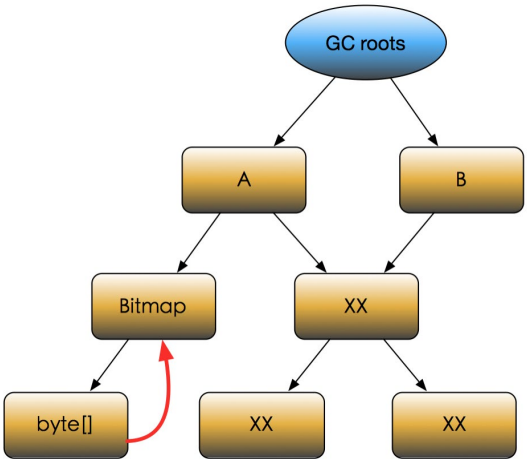
然后对归并后的 Instance 按 RetainSize 进行排序，取出 TOP N 的 Instance，

其中在排序过程中我们会对 N 的值进行动态调整，保证 RetainSize 达到一定阈值的 Instance 都能被发现。对于这些 Instance 才进行最后的链路分析，这样就能大大缩短分析时长。

排序过程：创建一个初始容量为 5 的集合，往里添加 Instance 后进行排序，然后遍历后面的 Instance，当 Instance 的 RetainSize 大于总共消耗内存大小的 5% 时，进行扩容，并重新排序。当 Instance 的 RetainSize 大于现有集合中的最小值时，进行替换，并重新排序。

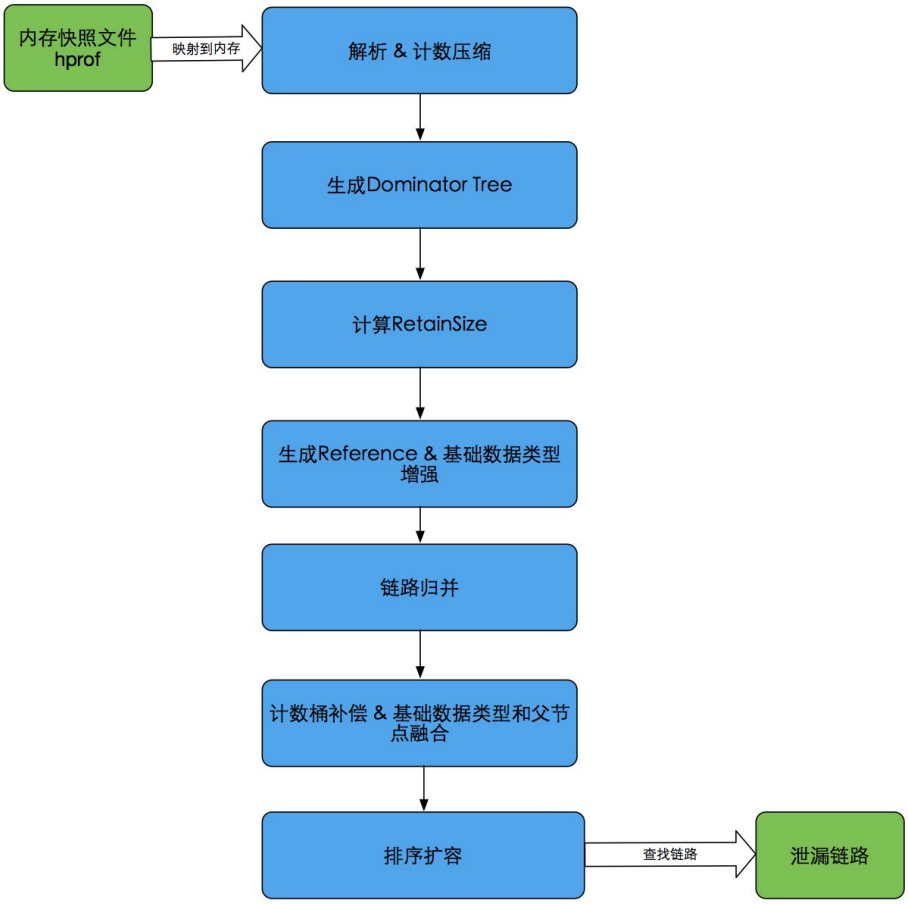


基础类型检测不到



为了解决 HAHA 算法中检测不到基础类型泄漏的问题，我们在遍历堆中的 Instance 时，如果发现是 ArrayInstance，且是 byte 类型时，将它自身舍弃掉，并将它的 RetainSize 加在它的父 Instance 上，然后用父 Instance 进行后面的排序。

至此，我们对 HAHA 的原始算法做了诸多优化（如下图），很大程度解决了分析进程自身 OOM 问题、分析时间过长问题以及基础类型检测不到的问题。



针对线上堆内存不足问题，Probe 最后会自动分析出 RetainSize 大小 Top N 对象到 GC Roots 的链路，上报给服务器，进行报警。下面是一个线上案例，这里截取了上报的链路分析结果中的一部分，完整的分析结果就是多个这样的组合。在第一段链路分析可以看到，有个 Bitmap 对象占用了 2MB 左右的内存，根据链路定

位到代码，修复了 Bitmap 泄漏问题。第二段链路分析反映的是一个 Timer 泄漏问题，可以看出内存中存在 4 个这样的 Instance，每个 Instance 的 Retain Size 是 595634，所以这个问题会泄漏的内存大小是 4*595634=2.27MB。



裁剪回捞 HPROF 文件

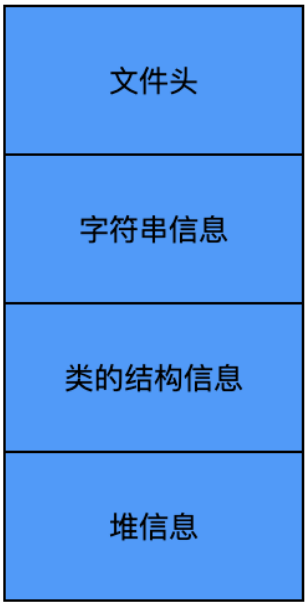
在 Probe 上线分析方案之后，发现尽管我们做了很多优化，但是受到手机自身性能的约束，线上分析的成功率也只有 65%。

于是，我们对另一种思路即回捞 HPROF 文件后本地分析进行了探索，这种方案最大的问题就是线上流量问题，因为 HPROF 文件动辄几百 MB，如果直接进行上传，势必会对用户的流量消耗带来巨大影响。

使用这种方案的关键点就在于减少上传的 HPROF 文件大小，减少文件大小首先想到的就是压缩，不过只是做压缩的话，文件还是太大。接下来，我们就考虑几百 MB 的文件内容是否都是我们需要的，是否可以对文件进行裁剪。我们希望对 HPROF 无用的信息进行裁剪，只保留我们关心的数据，就需要先了解 HPROF 文件的格式：

Debug.dumpHprofData() 其内部调用的是 VMDebug 的同名函数，层层深入最终可以找到 /art/runtime/hprof/hprof.cc，HPROF 的生成操作基本都是在这里执行的，结合 HAHA 库代码阅读 hrpof.cc 的源码。

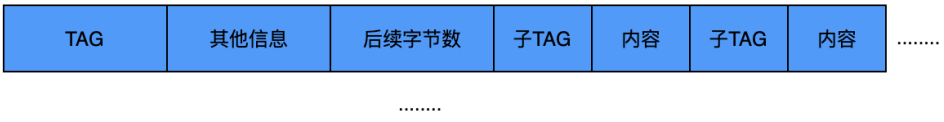
HPROF 文件的大体格式如下：



一个 HPROF 文件主要分为这四部分：

- 文件头。
- 字符串信息：保存着所有的字符串，在解析的时候通过索引 id 被引用。
- 类的结构信息：是所有 Class 的结构信息，包括内部的变量布局，父类的信息等等。
- 堆信息：即我们关心的内存占用与对象引用的详细信息。

其中我们最关心的堆信息是由若干个相同格式的元素组成，这些元素的大体格式如下图：



每个元素都有个 TAG 用来标识自己的身份，而后续字节数则表示元素的内容长度。元素携带的内容则是若干个子元素组合而成，通过子 TAG 来标识身份。

具体的 TAG 和身份的对应关系可以在 hrpof.cc 源码中找到，这里不进行展开。

```
// Values for the first byte of HEAP_DUMP and HEAP_DUMP_SEGMENT records:
enum HprofHeapTag {
  // Traditional.
  HPROF_ROOT_UNKNOWN = 0xFF,
  HPROF_ROOT_JNI_GLOBAL = 0x01,
  HPROF_ROOT_JNI_LOCAL = 0x02,
  HPROF_ROOT_JAVA_FRAME = 0x03,
  HPROF_ROOT_NATIVE_STACK = 0x04,
  HPROF_ROOT_STICKY_CLASS = 0x05,
  HPROF_ROOT_THREAD_BLOCK = 0x06,
  HPROF_ROOT_MONITOR_USED = 0x07,
  HPROF_ROOT_THREAD_OBJECT = 0x08,
  HPROF_CLASS_DUMP = 0x20,
  HPROF_INSTANCE_DUMP = 0x21,
  HPROF_OBJECT_ARRAY_DUMP = 0x22,
  HPROF_PRIMITIVE_ARRAY_DUMP = 0x23,

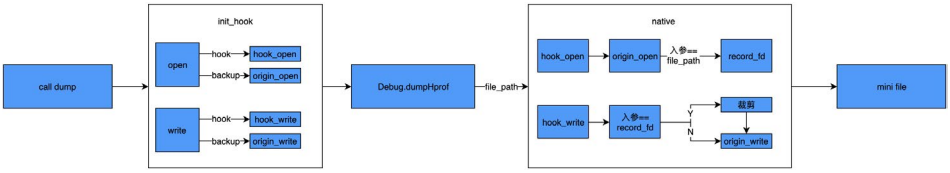
  // Android.
  HPROF_HEAP_DUMP_INFO = 0xfe,
  HPROF_ROOT_INTERNED_STRING = 0x89,
  HPROF_ROOT_FINALIZING = 0x8a,  // Obsolete.
  HPROF_ROOT_DEBUGGER = 0x8b,
  HPROF_ROOT_REFERENCE_CLEANUP = 0x8c,  // Obsolete.
  HPROF_ROOT_VM_INTERNAL = 0x8d,
  HPROF_ROOT_JNI_MONITOR = 0x8e,
  HPROF_UNREACHABLE = 0x90,  // Obsolete.
  HPROF_PRIMITIVE_ARRAY_NODATA_DUMP = 0xc3,  // Obsolete.
};
```

弄清楚了文件格式，接下来需要确定裁剪内容。经过思考，我们决定裁减掉全部基本类型数组的值，原因是我们的使用场景一般是排查内存泄漏以及 OOM，只关心对象间的引用关系以及对象大小即可，很多时候对于值并不是很在意，所以裁减掉这部分的内容不会对后续的分析造成影响。

最后需要确定裁剪方案。先是尝试了 dump 后在 Java 层进行裁剪，发现效率很低，很多时候这一套操作下来需要 20s。然后又尝试了 dump 后在 Native 层进行裁剪，这样做效率是高了点，但依然达不到预期。

经过思考，如果能够在 dump 的过程中筛选出哪些内容是需要保留的，哪些内容是需要裁剪的，需要裁剪的内容直接不写入文件，这样整个流程的性能和效率绝对是最高的。

为了实现这个想法，我们使用了 GOT 表 Hook 技术（不展开介绍）。有了 Hook 手段，但是还没有找到合适的 Hook 点。通过阅读 hrpof.cc 的源码，发现最适合的点就是在写入文件时，拿到字节流进行裁剪操作，然后把有用的信息写入文件。于是项目最终的结构如下图：



我们对 IO 的关键函数 open 和 write 进行 Hook。Hook 方案使用的是爱奇艺开源的 xHook 库。

在执行 dump 的准备阶段，我们会调用 Native 层的 open 函数获得一个文件句柄，但实际执行时会进入到 Hook 层中，然后将返回的 FD 保存下来，用作 write 时匹配。

在 dump 开始时，系统会不断的调用 write 函数将内容写入到文件中。由于我们的 Hook 是以 so 为目标的，系统运行时也会有许多写文件的操作，所以我们需要对前面保存的 FD 进行匹配。若 FD 匹配成功则进行裁剪，否则直接调用 origin-write 进行写入操作。

流程结束后，就会得到裁剪后的 mini-file，裁剪后的文件大小只有原始文件大小的十分之一左右，用于线上可以节省大部分的流量消耗。拿到 mini-file 后，我们将裁剪部分的位置填上字节 0 来进行恢复，这样就可以使用传统工具打开进行分析了。

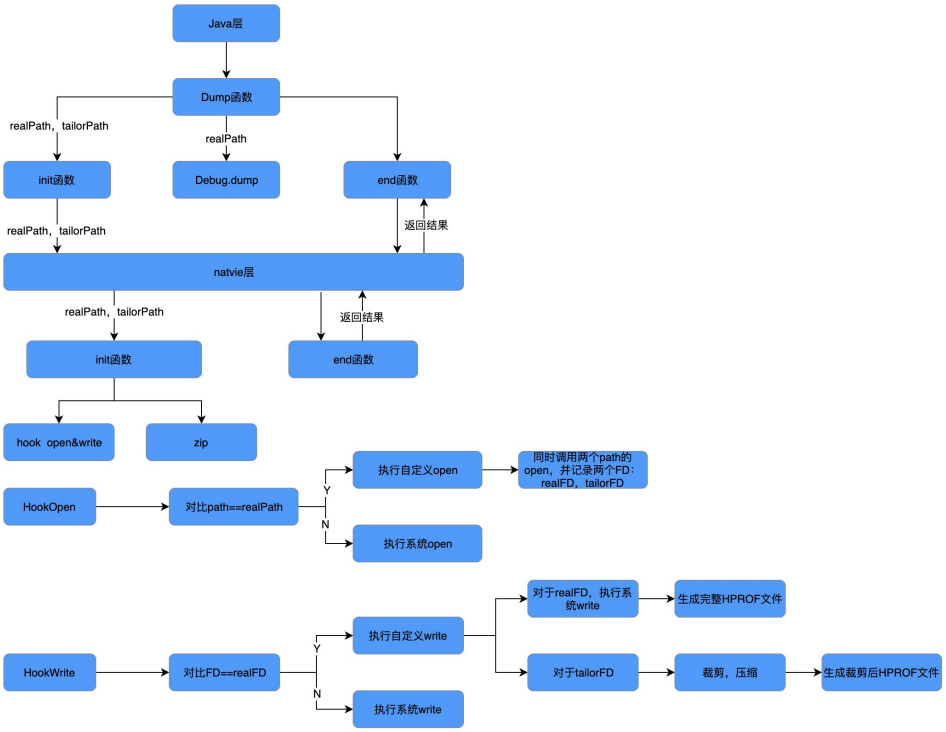
原始 HPROF 文件和裁剪后再恢复的 HPROF 文件分别在 Android Studio 中打开，发现裁剪再恢复的 HPROF 文件打开后，只是看不到对象中的基础数据类型值，而整个的结构、对象的分布以及引用链路等与原始 HPROF 文件是完全一致的。事实证明裁剪方案不会影响后续对堆内存的链路分析。

方案融合

由于目前裁剪方案在部分机型上（主要是 Android 7.X 系统）不起作用，所以在 Probe 中同时使用了这两种方案，对两种方案进行了融合。即通过一次 dump 操作

得到两份 HPROF 文件，一份原始文件用于下次启动时分析，一份裁剪后的文件用于上传服务器。

Probe 的最终方案实现如下图，主要是在调用 dump 函数之前先将两个文件路径（希望生成的原始文件路径和裁剪文件路径）传到 Native 层，Native 层记录下两个文件路径，并对 open 和 write 函数进行 Hook。hookopen 函数主要是通过 open 函数传入的 path 和之前记录的 path 比对，如果相同，我们就会同时调用之前记录的两个 path 的 open，并记录下两个 FD，如果不相同则直接调原生 open 函数。hookwrite 函数主要是通过传入的 FD 与之前 hookopen 中记录的 FD 比对，如果相同会先对原始文件对应的 FD 执行原生 write，然后对裁剪文件对应的 FD 执行我们自定义的 write，进行裁剪压缩。这样再传入原始文件路径调用系统的 dump 函数，就能够同时得到一份完整的 HPROF 文件和一份裁剪后的 HPROF 文件。



线程数超出限制

对于创建线程失败导致的 OOM，Probe 会获取当前进程所占用的虚拟内存、进程中的线程数量、每个线程的信息（线程名、所属线程组、堆栈信息）以及系统的线程数限制，并将这些信息上传用于分析问题。

/proc/sys/kernel/threads-max 规定了每个进程创建线程数目的上限。在华为的部分机型上，这个上限被修改的很低（大约 500），比较容易出现线程数溢出的问题，而大部分手机这个限制都很大（一般为 1W 多）。在这些手机上创建线程失败大多都是因为虚拟内存空间耗尽导致的，进程所使用的虚拟内存可以查看 /proc/pid/status 的 VmPeak/VmSize 记录。

然后通过 Thread.getAllStackTraces() 可以得到进程中的所有线程以及对应的堆栈信息。

一般来说，当进程中线程数异常增多时，都是某一类线程被大量的重复创建。所以我们只需要定位到这类线程的创建时机，就能知道问题所在。如果线程是有自定义名称的，那么直接就可以在代码中搜索到创建线程的位置，从而定位问题，如果线程创建时没有指定名称，那么就需要通过该线程的堆栈信息来辅助定位。下面这个例子，就是一个“crowdSource msg”的线程被大量重复创建，在代码中搜索名称很快就查出了问题。针对这类线程问题推荐的做法就是在项目中统一使用线程池，可以很大程度上避免线程数的溢出问题。

线程信息：

```
thread name: Thread[nio_tunnel_handler,5,main]   count: 1
thread name: Thread[OkHttp Dispatcher,5,main]   count: 30
thread name: Thread[process_read_thread,5,main]   count: 4
thread name: Thread[Jit thread pool worker thread 0,5,main]   count: 1
thread name: Thread[crowdSource msg,5,main]   count: 202
thread name: Thread[Timer-4,5,main]   count: 1
thread name: Thread[mqt_js,5,main]   count: 1

threadnames:Thread[Thread-5,5,main] count:1
trace:
java.lang.Object.wait(NativeMethod)
com.dianping.networklog.d.run(UnknownSource:28)
```

FD 数超出限制

前面介绍了，当进程中的 FD 数量达到最大限制时，再去新建线程，在创建 JNIEnv 时会抛出 OOM 错误。但是 FD 数量超出限制除了会导致创建线程抛出 OOM 以外，还会导致很多其它的异常，为了能够统一处理这类 FD 数量溢出的问题，Probe 中对进程中的 FD 数量做了监控。在后台启动一个线程，每隔 1s 读取一次当前进程创建的 FD 数量，当检测到 FD 数量达到阈值时（FD 最大限制的 95%），读取当前进程的所有 FD 信息归并后上报。

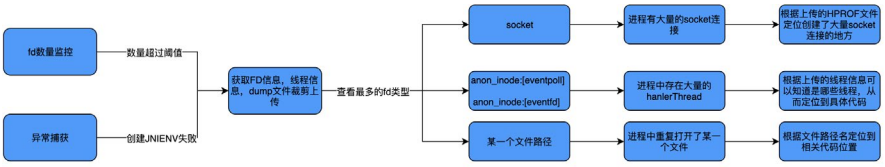
在 /proc/pid/limits 描述着 Linux 系统对对应进程的限制，其中 Max open files 就代表可创建 FD 的最大数目。

进程中创建的 FD 记录在 /proc/pid/fd 中，通过遍历 /proc/pid/fd，可以得到 FD 的信息。

获取 FD 信息：

```
File fdFile=new File("/proc/" + Process.myPid() + "/fd");
File[] files = fdFile.listFiles();
int length = files.length; // 即进程中的 fd 数量
for (int i = 0; i < length ; i++) {
  if (Build.VERSION.SDK_INT >= 21) {
        Os.readlink(files[i].getAbsolutePath()); // 得到软链接实际指向的文件
    } else {
     //6.0 以下系统可以通过执行 readlink 命令去得到软连接实际指向文件，但是耗时较久
  }
}
```

得到进程中所有的 FD 信息后，我们会先按照 FD 的类型进行一个归并，FD 的用途主要有打开文件、创建 socket 连接、创建 handlerThread 等。



比如像下面这个例子中，就是 anon_inode:[eventpoll] 和 anon_inode:[eventfd] 的数量异常的多，说明进程中很可能是启动了大量的 handlerThread，再结合回传

上来的线程信息就能快速定位到问题代码的具体位置。

FD 溢出案例：

FD 信息：

```
anon_inode:[eventpoll]   count: 381
anon_inode:[eventfd]   count: 381
pipe   count  26
socket   count  32
/system/framework/framework-res.apk   count: 1
 ..........
Thread 信息:
thread name: Thread[Jit thread pool worker thread 0,5,main]   count: 1
thread name: Thread[mtqq handler,5,main]   count: 302
thread name: Thread[Timer-4,5,main]   count: 1
thread name: Thread[mqt_js,5,main]   count: 1
```

总结

Probe 目前能够有效定位线上 Java 堆内存不足、FD 泄漏以及线程溢出的 OOM 问题。骑手 Android 端使用 Probe 组件解决了很多线上的 OOM 问题，将线上 OOM Crash 率从最高峰的 2‰降低到了现在的 0.02‰左右。我们后续也会继续完善 Probe 组件，例如 HPROF 文件裁剪方案对 7.X 系统的兼容以及 Native 层的内存问题定位。

作者简介

逢搏，美团配送 App 团队研发工程师。
毅然，美团配送 App 团队高级技术专家。
永刚，美团平台监控团队研发工程师。

招聘信息

美团配送 App 团队，负责美团骑手、美团众包、美团跑腿等配送相关 App 的研发，涉及技术领域包括但不限于 App 的稳定性建设、App 性能监控和优化、大前端跨平台动态化、App 安全。对上述领域感兴趣的请联系 tech#meituan.com。

活动 Web 页面人机识别验证的探索与实践

益国

在电商行业，线上的营销活动特别多。在移动互联网时代，一般为了活动的快速上线和内容的即时更新，大部分的业务场景仍然通过 Web 页面来承载。但由于 Web 页面天生“环境透明”，相较于移动客户端页面在安全性上存在更大的挑战。本文主要以移动端 Web 页面为基础来讲述如何提升页面安全性。

活动 Web 页面的安全挑战

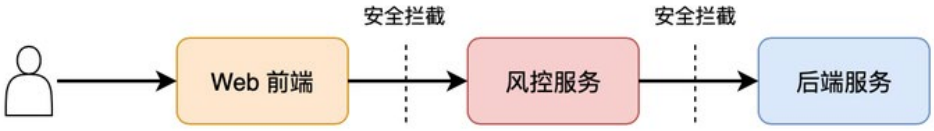
对于营销活动类的 Web 页面，领券、领红包、抽奖等活动方式很常见。此类活动对于普通用户来说大多数时候就是“拼手气”，而对于非正常用户来说，可以通过直接刷活动 API 接口的“作弊”方式来提升“手气”。这样的话，对普通用户来说，就变得很不公平。

对于活动运营的主办方来说，如果风控措施做的不好，这类刷接口的“拼手气”方式可能会对企业造成较大的损失。如本来计划按 7 天发放的红包，在上线 1 天就被刷光了，活动的营销成本就会被意外提升。主办方想发放给用户的满减券、红包，却大部分被黄牛使用自动脚本刷走，而真正想参与活动的用，却无法享受活动优惠。

终端用户到底是人还是机器，网络请求是否为真实用户发起，是否存在安全漏洞并且已被“羊毛党”恶意利用等等，这些都是运营主办方要担心的问题。

安全防范的基本流程

为了提升活动 Web 页面的安全性，通常会引入专业的风控服务。引入风控服务后，安全防护的流程大致如图所示。



- Web 前端：用户通过 Web 页面来参与活动，同时 Web 前端也会收集用于人机识别验证的用户交互行为数据。由于不同终端（移动端 H5 页面和 PC 端页面）交互形式不同，收集用户交互行为数据的侧重点也会有所不同。
- 风控服务：一般大公司都会有专业的风控团队来提供风控服务，在美团内部有智能反爬系统来基于地理位置、IP 地址等大数据来提供频次限制、黑白名单限制等常规的基础风控拦截服务。甚至还有依托于海量的全业务场景的用户大数据，使用贝叶斯模型、神经网络等来构建专业度较深的服务。风控服务可以为 Web 前端提供通用的独立验证 SDK：验证码、滑块验证等区分人机的“图灵验证”，也可以为服务端提供 Web API 接口的验证等。
- 后端业务服务：负责处理活动业务逻辑，如给用户发券、发红包，处理用户抽奖等。请求需要经过风控服务的验证，确保其安全性，然后再来处理实际业务逻辑，通常，在处理完实际业务逻辑时，还会有针对业务本身的风控防范。

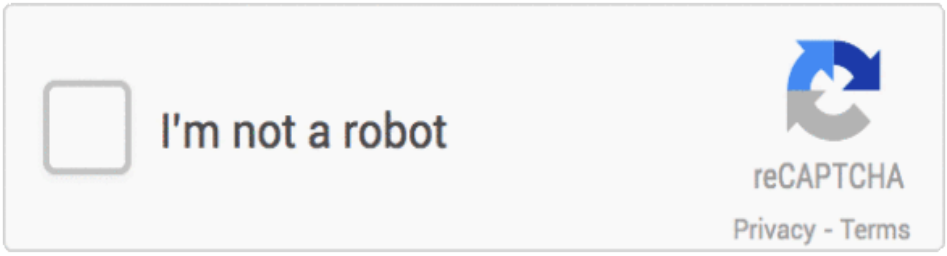
对于活动 Web 页面来说，加入的风控服务主要为了做人机识别验证。在人机识别验证的专业领域上，我们可以先看看业界巨头 Google 是怎么做的。

Google 如何处理人机验证

Google 使用的人机验证服务是著名的 reCAPTCHA（Completely Automated Public Turing Test To Tell Computers and Humans Apart，区分人机的全自动图灵测试系统），也是应用最广的验证码系统。早年的 reCAPTCHA 验证码是这样的：



如今的 reCAPTCHA 已经不再需要人工输入难以识别的字符，它会检测用户的终端环境，追踪用户的鼠标轨迹，只需用户点击“我不是机器人”就能进行人机验证（reCAPTCHA 骗用户进行数据标注而进行 AI 训练的验证另说）。



reCAPTCHA 的验证方式从早先的输入字符到现在的轻点按钮，在用户体验上，有了较大的提升。

而在活动场景中引入人机识别验证，如果只是简单粗暴地增加验证码，或者只是像 reCAPTCHA 那样增加点击“我不是机器人”的验证，都会牺牲用户体验，降低用户参加活动的积极性。

Google 的普通 Web 页面的浏览和有强交互的活动 Web 页面虽是不同的业务场景，但对于活动 Web 页面来说，强交互恰好为人机识别验证提供了用户交互行为数据收集的契机。

人机识别验证的技术挑战

理想的方案是在用户无感知的情况下做人机识别验证，这样既确保了安全又对用户体验无损伤。

从实际的业务场景出发再结合 Web 本身的环境，如果想实现理想的方案，可能会面临如下的技术挑战：

- （1）需要根据用户的使用场景来定制人机识别验证的算法：Web 前端负责收集、上报用户交互行为数据，风控服务端校验上报的数据是否符合正常的用户行为逻辑。
- （2）确保 Web 前端和风控服务端之间通信和数据传输的安全性。

• (3) 确保上述两大挑战中提到的逻辑和算法不会被代码反编译来破解。

在上述的三个挑战中，(1) 已经实现了人机识别验证的功能，而 (2) 和 (3) 都是为了确保人机识别验证不被破解而做的安全防范。接下来，本文会分别针对这三个技术挑战来说明如何设计技术方案。

挑战一：根据用户使用场景来定制人机识别验证算法

先来分析一下用户的使用场景，正常用户参与活动的步骤是用户进入活动页面后，会有短暂的停留，然后点击按钮参与活动。这里所说的“参与活动”，最终都会在活动页面发起一个接口的请求。如果是非正常用户，可以直接跳过以上的实际动作而去直接请求参与活动的接口。

那么区别于正常用户和非正常用户就是那些被跳过的动作，对实际动作进一步归纳如下：

1. 进入页面。
2. 短暂的停留。
3. 滚动页面。
4. 点击按钮。

以上的动作又可以分为必需的操作和可选的操作。对这一连串动作产生的日志数据进行收集，在请求参与活动的接口时，将这些数据提交至后端，验证其合法性。这就是一个简单的人机识别验证。

在验证动作的合法性时，需要考虑到这些动作数据是不是能被轻易模拟。另外，动作的发生应该有一条时间线，可以给每个动作都增加一个时间戳，比如点击按钮肯定是在进入页面之后发生的。

一些特定的动作的日志数据也会有合理的区间，进入页面的动作如果以 JS 资源加载的时间为基准，那么加载时间可能大于 100 毫秒，小于 5 秒。而对于移动端的按钮点击，点击时记录的坐标值也会有对应的合理区间，这些合理的区间会根据实际的环境和情况来进行设置。

除此之外，设备环境的数据也可以进行收集，包括用户参与活动时使用的终端类型、浏览器的类型、浏览器是否为客户端的容器等，如果使用了客户端，客户端是否会携带特殊的标识等。

最后，还可以收集一些“无效”的数据，这些数据用于障人耳目，验证算法会将其忽略。尽管收集数据的动作是透明的，但是验证数据合法性不是透明的，攻击者无法知道，验证的算法中怎么区分哪些是有效、哪些是无效。这已经有点“蜜罐数据”的意思了。

挑战二：确保通信的安全性

收集的敏感数据要发送给风控服务端，进而确保通信过程的安全。

1. Web API 接口不能被中途拦截和篡改，通信协议使用 HTTPS 是最基本的要求；同时还要让服务端生成唯一的 Token，在通信过程中都要携带该 Token。
2. 接口携带的敏感数据不能是明文的，敏感数据要进行加密，这样攻击者无法通过网络抓包来详细了解敏感数据的内容。

Token 的设计

Token 是一个简短的字符串，主要为了确保通信的安全。用户进入活动 Web 页面后，请求参与活动的接口之前，会从服务端获取 Token。该 Token 的生成算法要确保 Token 的唯一性，通过接口或 Cookie 传递给前端，然后，前端在真正请求参与活动的接口时需要带上该 Token，风控服务端需要验证 Token 的合法性。也就是说，Token 由服务端生成，传给前端，前端再原封不动的回传给服务端。一旦加入了 Token 的步骤，攻击者就不能直接去请求参与活动的接口了。

Token 由风控服务端基于用户的身份，根据一定的算法来生成，无法伪造，为了提升安全等级，Token 需要具有时效性，比如 10 分钟。可以使用 Redis 这类缓存服务来存储 Token，使用用户身份标识和 Token 建立 KV 映射表，并设置过期时间为 10 分钟。

虽然前端在 Cookie 中可以获取到 Token，但是前端不能对 Token 做持久化的

缓存。一旦在 Cookie 中获取到了 Token，那么前端可以立即从 Cookie 中删除该 Token，这样能尽量确保 Token 的安全性和时效性。Token 存储在 Redis 中，也不会因为用户在参与活动时频繁的切换页面请求，而对服务造成太大的压力。

另外，Token 还可以有更多的用处：

- 标识参与活动用户的有效性。
- 敏感数据对称加密时生成动态密钥。
- API 接口的数字签名。

敏感数据加密

通信时，传递的敏感数据可以使用常见的对称加密算法进行加密。

为了提升加密的安全等级，加密时的密钥可以动态生成，前端和风控服务端约定好动态密钥的生成规则即可。加密的算法和密钥也要确保不被暴露。

通过对敏感数据加密，攻击者在不了解敏感数据内容的前提下就更别提模拟构造请求内容了。

挑战三：化解纸老虎的尴尬

有经验的 Web 开发者看到这里，可能已经开始质疑了：在透明的前端环境中折腾安全不是白折腾吗？这就好比费了很大的劲却只是造了一个“纸老虎”，质疑是有道理的，但是且慢，通过一些安全机制的加强是可以让“纸老虎”尽可能的逼真。

本文一再提及的 Web 环境的透明性，是因为在实际的生产环境中的问题：前端的代码在压缩后，通过使用浏览器自带的格式化工具和断点工具，仍然具备一定的可读性，花点时间仍然可以理解代码的逻辑，这就给攻击者提供了大好的代码反编译机会。

如果要化解“纸老虎”的尴尬，就要对前端的代码进行混淆。

前端代码混淆

前端的 JS 代码压缩工具基本都是对变量、函数名称等进行缩短，压缩对于混淆的作用是比较弱。除了对代码进行压缩，还需要进行专门的混淆。

对代码进行混淆可以降低可读性，混淆工具有条件的话最好自研，开源的工具要慎用。或者基于 Uglify.js 来自定义混淆的规则，混淆程度越高可读性就越低。

代码混淆也需要把握一个度，太复杂的混淆可能会让代码无法运行，也有可能会影响本身的执行效率。同时还需要兼顾混淆后的代码体积，混淆前后的体积不能有太大的差距，合理的混淆程度很重要。



```
!function(e,u,A,l,p,O,k,g,h,j,v,d,w,x,U,m,F,I,D,E,R,C,P,S,T,q,z,B,G,H,r){module.exports=function(e)
{return function e(r,a,t){if(!(r instanceof Object))return r;var n;switch(Object[p][l][A](r)[u](O,-k))
{case g:n=[];break;case h:n=new Date(r[j]());break;case v:n=new RegExp(r);break;case d:break;case
w:case x:case U:case m:case F:case I:case D:case E:case R:n=r[C]();break;default:n={}}if(a[P](r),t[P]
(n),r instanceof Array)for(var c=S;c<r[T];c++)n[c]=e(r[c],a,t);for(var s=Object[z](r)[q](),i=Object[z]
(n)[q](),o=S;o<s[T];o++){var y=s[o];if(i[T]>S&&y===i[S])i[B]();else if(Object[p][G][A](r,y)){var
f=r[y],b=a[H](f);n[y]=b!==-k?t[b]:e(r[y],a,t)}}return n}(e,[],[])}}
(0,"slice","call","toString","prototype",8,1,"Array","Date","getTime","RegExp","Function","Uint8Array",
"Uint8ClampedArray","Uint16Array","Uint32Array","Int8Array","Int16Array","Int32Array","Float32Array","F
loat64Array","subarray","push",0,"length","sort","keys","shift","hasOwnProperty","indexOf");
```

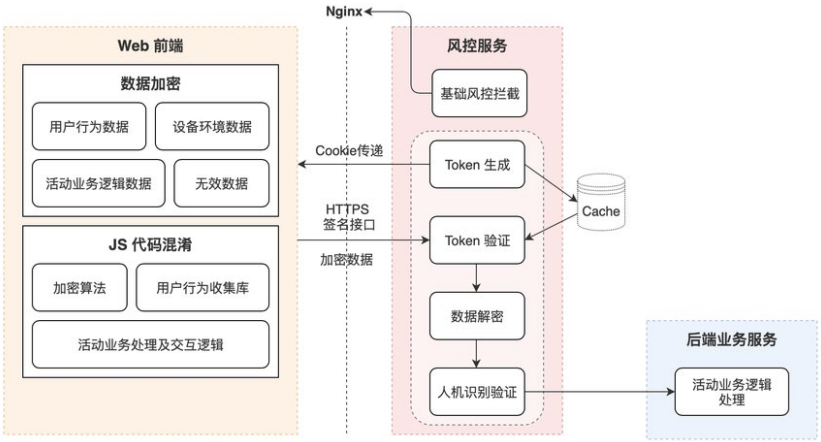
断点工具的防范会更麻烦些。在使用断点工具时通常都会导致代码延迟执行，而正常的业务逻辑都会立即执行，这是一个可以利用的点，可以考虑在代码执行间隔上来防范断点工具。

通过代码混淆和对代码进行特殊的处理，可以让格式化工具和断点工具变得没有用武之地。唯一有些小遗憾，就是处理后的代码也不能正常使用 Source Map 的功能了。

有了代码混淆，反编译的成本会非常高，这样“纸老虎”已经变得很逼真了。

技术方案设计

在讲解完如何解决关键的技术挑战后，就可以把相应的方案串起来，然后设计成一套可以实施的技术方案了。相对理想的技术方案架构图如下：



下面会按步骤来讲解技术方案的处理流程：

Step 0　基础风控拦截

基础风控拦截是上面提到的频次、名单等的拦截限制，在 Nginx 层就能直接实施拦截。如果发现是恶意请求，直接将请求过滤返回 403，这是初步的拦截，用户在请求 Web 页面的时候就开始起作用了。

Step 1　风控服务端生成 Token 后传给前端

Step 0 可能还没进入到活动 Web 页面，进入活动 Web 页面后才真正开始人机识别验证的流程，前端会先开始获取 Token。

Step 2　前端生成敏感数据

敏感数据应包含用户交互行为数据、设备环境数据、活动业务逻辑数据以及无效数据。

Step 3　使用 HTTPS 的签名接口发送数据

Token 可以作为 Authorization 的值添加到 Header 中，数据接口的签名可以有效防止 CSRF 的攻击。

Step 4　数据接口的校验

风控服务端收到请求后，会先验证数据接口签名中的 Token 是否有效。验证完 Token，才会对敏感数据进行解密。数据解密成功，再进一步对人机识别的数据合法

性进行校验。

Step 5 业务逻辑的处理

前面的步骤为了做人机识别验证，这些验证不涉及到业务逻辑。在所有这些验证都通过后，后端业务服务才会开始处理实际的活动业务逻辑。处理完活动业务逻辑，最终才会返回用户参与活动的结果。

总结

为了提升活动 Web 页面的安全性，使用了各种各样的技术方案，我们将这些技术方案组合起来才能发挥安全防范的作用，如果其中某个环节处理不当，都可能会被当作漏洞来利用，从而导致整个验证方案被攻破。

为了验证技术方案的有效性，可以持续观察活动 API 接口的请求成功率。从请求成功率的数据中进一步分析“误伤”和“拦截”的数据，以进一步确定是否要对方案进行调优。

通过上述的人机识别验证的组合方案，可以大幅提升活动 Web 页面的安全性。在活动 Web 页面应作为一个标准化的安全防范流程，除了美团，像淘宝和天猫也有类似的流程。由于活动运营的环节和方法多且复杂，仅仅提升了 Web 页面也不敢保证就是铁板一块，安全需要关注的环节还很多，安全攻防是一项长期的“拉锯升级战”，安全防范措施也需要持续地优化升级。

参考资料

- https://www.google.com/recaptcha/intro/v3.html
- https://segmentfault.com/a/1190000006226236
- https://www.freebuf.com/articles/web/102269.html

作者简介

益国，美团点评 Web 前端开发工程师。2015 年加入美团，曾先后负责过风控前端 SDK 和活动运营平台的研发，现负责大数据平台的研发工作。

React Native 工程中 TSLint 静态检查工具的探索之路

家正

背景

建立的代码规范没人遵守，项目中遍地风格迥异的代码，你会不会抓狂?

通过测试用例的程序还会出现 Bug，而原因仅仅是自己犯下的低级错误，你会不会抓狂?

某种代码写法存在问题导致崩溃时，只能全工程检查代码，这需要人工花费大量时间 Review 代码，你会不会抓狂?

以上这些问题，可以通过静态检查有效地缓解!

静态检查（Static Program Analysis）主要是以不运行程序的方式对于程序源代码进行检查分析的技术，而与之相反的就是动态检查（Dynamic Program Analysis），通过实际运行程序输入测试数据产生预期结果的技术。通过代码静态检查，我们可以快速定位代码的错误与缺陷，可以减少逐行阅读代码浪费的时间，可以（根据需要）快速扫描代码中可能存在的漏洞等。代码静态检查可以在代码的规范性、安全性、可靠性、可维护性等方面起到重要作用。

在客户端中，Android 可以使用 CheckStyle、Lint、Findbugs、PMD 等工具，iOS 可以使用 Clang Static Analyzer、OCLint 等工具。而在 React Native 的开发过程中，针对于 JavaScript 的 ESLint，与 TypeScript 的 TSLint，则成为了主要代码静态检查的工具。本文将按照使用 TSLint 的原因、使用 TSLint 的方法、自定义 TSLint 的步骤进行探究分析。

一、使用 TSLint 的原因

在客户端团队进入 React Native 项目的开发过程中，面临着如下问题：

1. 由于大家从客户端转入到 React Native 开发过程中，容易出现低级语法错误；
2. 开发者之前从事 Android、iOS、前端等工作，因此代码风格不同，导致项目代码风格不统一；
3. 客户端效果不一致，有可能 Android 端显示正常、iOS 端显示异常，或者相反的情况出现。

虽然以上问题可以通过多次不断将雷点标记出，并不断地分享经验与强化代码 Review 过程等方式来进行缓解，但是仍面临着 React Native 开发者掌握的技术水平千差万别，知识分享传播的速度缓慢等问题，既导致了开发成本的不断增加和开发效率持续低下的问题，还难以避免一个坑被踩了多次的情况出现。这时急需一款可以满足以下目标的工具：

1. 可检测代码低级语法错误；
2. 规范项目代码风格；
3. 根据需要可自定义检查代码的逻辑；
4. 工具使用者可以“傻瓜式”的接入部署到开发 IDE 环境；
5. 可以快速高效地将检查工具最新检查逻辑同步到开发 IDE 环境中；
6. 对于检查出的问题可以快速定位。

根据上述要求的描述，静态检查工具 TSLint 可以较为有效地达成目标。

二、TSLint 介绍

TSLint 是硅谷企业 Palantir 的一个项目，它是一款可以检查 TypeScript 代码可读性、可维护性以及功能性错误的静态检查工具，当前许多编辑器（Editors）和构建系统（Build Systems）支持这一工具，同时支持自定义编写 Lint 规则、配置、格式化等。

当前 TSLint 已经包含了上百条规则，这些规则构筑了当前 TSLint 检查的基础。在代码开发阶段中，通过这些配置好的规则可以给工程一个完整的检查，并随时可以提示出可能存在的问题。本文内容参考了 TSLint 官方文档 https://palantir.github.io/tslint/。

TSLint 常见规则

以下规则主要来源于 TSLint 规则，是某些规则的简单介绍。

规则	含义	错误示例

class-name	class的命名规则，主要是首字母大写的pascal命名法	interface someInterface {}
		~~ [0] //提示命名不规范
		class Another_Invalid_Class_Name {
		~~ [0] //提示命名不规范
		}

no-unsafe-finally	不许在finally里面使用return\continue\break\throws	function() {
		try {
		} finally {
		return;
		~~ [return] //提示不许在finally里面使用return
		}
		}

align 可配置参数 "arguments", "elements", "members", "parameters", "statements	垂直对齐，用于代码风格	var invalidConstructorArgsAlignment = new foo(10, "abcd"); ~~~~~~~ [arguments are not aligned] //提示未对齐
no-empty	不允许空块出现	if (x === 1) {} ~~ [block is empty] //提示块内为空

no-switch-case-fall-through	不允许switch语句中直接穿过case，意思为必须有break，防止没有break，导致代码进入下个case里面去	witch (foo) {
		case 1:
		bar();
		case 2:
		~~ [expected a 'break' before 'case'] //提示在case关键字前要有break
		bar();
		bar();
		case 3:
		~~ [expected a 'break' before 'case']
		case 4:
		default:
		break;
		}

...	...	...

TSLint 规则示例

常用 TSLint 规则包

上述 2.1 所列出的规则来源于 Palantir 官方 TSLint 规则。实际还有多种，可能会用到的有以下：

TSLint规则包	规则包内容
tslint	Palantir官方推出的规则，是最基础的标准规则包
tslint-react	Palantir官方推出的规则，主要用于React & JSX的检查
tslint-microsoft-contrib	微软官方推出的规则，用于微软项目的一个TSLint规则包
tslint-consistent-codestyle	主要用于统一代码风格的规则包
tslint-config-airbnb	适用于Airbnb 代码风格统一的工程，其中此工程没有新规则，主要是基于tslint-consistent-codestyle、tslint-eslint-rules、tslint-microsoft-contrib这三个规则包进行配置，根据自己需求开启或者关闭规则
tslint-eslint-rules	主要是补充对应于ESLint规则中TSLint规则缺失的部分
...	...

TSLint 规则示例

我们在项目的规则配置过程中，一般采用上述规则包其中一种或者若干种同时配置，那如何配置呢？请看下文。

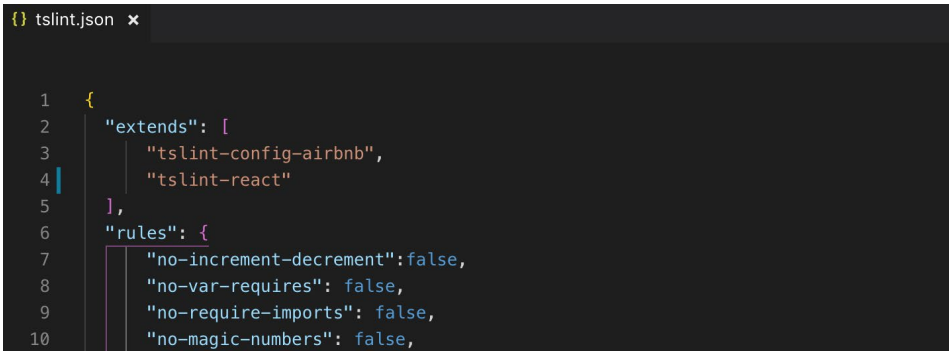
三、如何进行 TSLint 规则配置与检查

首先，在工程 package.json 文件中配置 TSLint 包：



TSLint 规则示例

在根目录中的 tslint.json 文件中可以根据需要配置已有规则，例如：



TSLint 规则示例

其中 extends 数组内放置继承的 TSLint 规则包，上图包括了 airbnb 配置的规则包、tslint-react 的规则包，而 rules 用于配置规则的开关。

TSLint 规则目前只有 true 和 false 的选项，这导致了结果要么正常，要么报错 ERROR，而不会出现 WARNING 等警告。

有些时候，虽然配置某些规则开启，但是某个文件内可能会关闭某些甚至全部规则检查，这时候可以通过规则注释来配置，如：

```
/* tslint:disable */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域关闭 TSLint 规则检查。

```
/* tslint:enable */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域开启 TSLint 规则检查。

```
/* tslint:disable:rule1 rule2 rule3... */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域关闭规则 rule1 rule2 rule3…的检查。

```
/* tslint:enable:rule1 rule2 rule3... */
```

上述注释表示本文件自此注释所在行开始，以下的所有区域开启规则 rule1 rule2 rule3…的检查。

```
// tslint:disable-next-line
```

上述注释表示此注释所在行的下一行关闭 TSLint 规则检查。

```
someCode(); // tslint:disable-line
```

上述注释表示此注释所在行关闭 TSLint 规则检查。

```
// tslint:disable-next-line:rule1 rule2 rule3...
```

上述注释表示此注释所在行的下一行关闭规则 rule1 rule2 rule3…的检查检查。

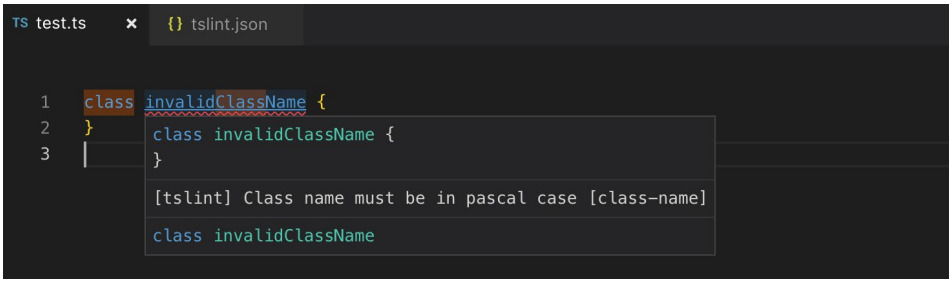
以上配置信息，这里具体参考了 https://palantir.github.io/tslint/usage/rule-flags/。

本地检查

在完成工程配置后，需要下载所需要依赖包，要在工程所在根目录使用 npm install 命令完成下载依赖包。

IDE 环境提示

在完成下载依赖包后，IDE 环境可以根据对应配置文件进行提示，可以实时地提示出存在问题代码的错误信息，以 VSCode 为例：



TSLint 规则示例

本地命令检查

VSCode 目前还有继续完善的空间，如果部分文件未在窗口打开的情况下，可能存在其中错误未提示出的情况，这时候，我们可以通过本地命令进行全工程的检查，在 React Native 工程的根目录下，通过以下命令行执行：

```
tslint --project tsconfig.json --config tslint.json
```

（此命令如果不正确运行，可在之前加入 ./node_modules/.bin/）即为：

```
./node_modules/.bin/tslint --project tsconfig.json --config tslint.json
```

从而会提示出类似以下错误的信息：

```
src/Components/test.ts[1, 7]: Class name must be in pascal case
```

在线 CI 检查

本地进行代码检查的过程也会存在被人遗忘的可能性，通过技术的保障，可以避免人为遗忘，作为代码提交的标准流程，通过 CI 检查后再合并代码，可以有效避免代码错误的问题。CI 系统可以为理解为一个云端的环境，环境配置与本地一致，在这种情况下，可以生成与本地一致的报告，在美团内部可以使用基于 Jenkins 的 Castle CI 系统，生成结果与本地结果一致：



TSLint 规则示例

其他方式

代码检查不止局限上述阶段，在代码 commit、pull request、打包等阶段均可触发。

- 代码 commit 阶段，通过 Hook 方式可以触发代码检查，可以有效地将在线 CI 检查阶段强制提前，基本保证了在线 CI 检查的完全正确性。
- 代码 pull request 阶段，通过在线 CI 检查可以触发代码检查，可以有效保证合入分支尤其是主分支的正确性。
- 代码打包阶段，通过在线 CI 检查可以触发代码检查，可以有效保证打包代码的正确性。

四、自定义编写 TSLint 规则

为什么要自定义 TSLint 规则

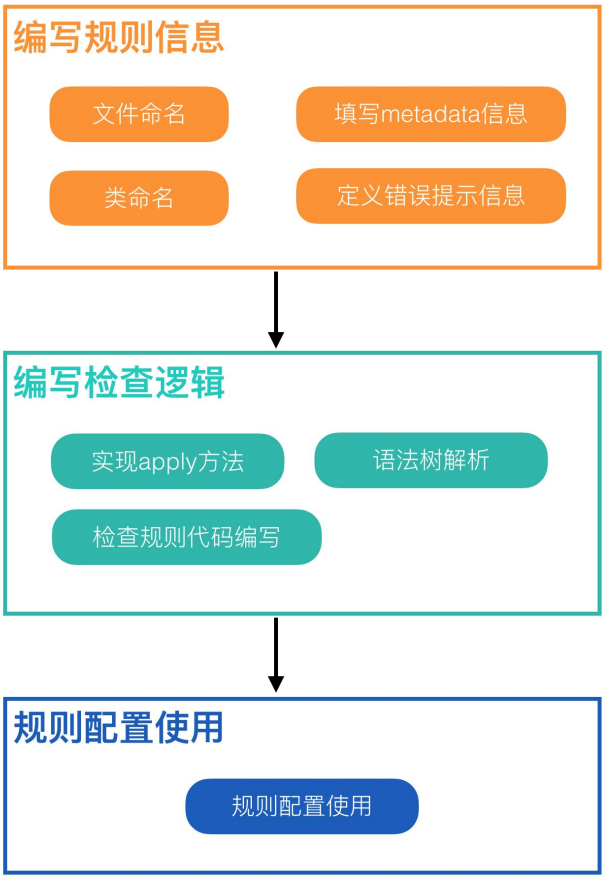
当前的 TSLint 规则虽然涵盖了比较普遍问题的一些代码检查，但是实践中还是存在一些问题的：

1. 团队中的个性化需求难以满足。例如，saga 中的异步函数需要在最外层加 try-catch，且 catch 块中需要加异常上报，这个明显在官方的 TSLint 规则无法实现，为此需要自定义的开发。
2. 官方规则的开启与配置不符合当前团队情况。

基于以上原因其他团队也有自定义 TSLint 的先例，例如上文提到的 tslint-microsoft-contrib、tslint-eslint-rules 等。

自定义规则步骤

那自定义 TSLint 大概需要什么步骤呢，首先规则文件根据规范进行按部就班的编写规则信息，然后根据代码检查逻辑对语法树进行分析并编写逻辑代码，这也是自定义规则的核心部分了，最后就是自定义规则的使用了。



TSLint 规则示例

自定义规则的示例直接参考官方的规则是最直接的，我们能这里参考一个比较简单的规则”class-name”。

“class-name”规则上文已经提到，它的意思是对类命名进行规范，当团队中类相关的命名不规范，会导致项目代码风格不统一甚至其他出现的问题，而”-

class-name”规则可以有效解决这个问题。我们可以看下具体的源码文件：https://github.com/palantir/tslint/blob/master/src/rules/classNameRule.ts。

然后将分步对此自定义规则进行讲解。

```
import { isClassLikeDeclaration, isInterfaceDeclaration } from "tsutils";
import * as ts from "typescript";

import * as Lint from "../index";
import { isUpperCase } from "../utils";

export class Rule extends Lint.Rules.AbstractRule {
    /* tslint:disable:object-literal-sort-keys */
    public static metadata: Lint.IRuleMetadata = {
        ruleName: "class-name",
        description: "Enforces PascalCased class and interface names.",
        rationale: "Makes it easy to differentiate classes from regular variables at a glance.",
        optionsDescription: "Not configurable.",
        options: null,
        optionExamples: [true],
        type: "style",
        typescriptOnly: false,
    };
    /* tslint:enable:object-literal-sort-keys */

    public static FAILURE_STRING = "Class name must be in pascal case";

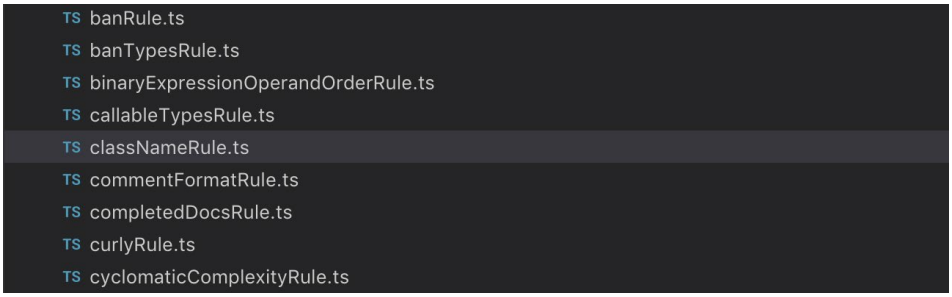
    public apply(sourceFile: ts.SourceFile): Lint.RuleFailure[] {
        return this.applyWithFunction(sourceFile, walk);
    }
}

function walk(ctx: Lint.WalkContext<void>) {
    return ts.forEachChild(ctx.sourceFile, function cb(node: ts.Node): void {
        if (isClassLikeDeclaration(node) && node.name !== undefined ||
            isInterfaceDeclaration(node)) {
            if (!isPascalCased(node.name!.text)) {
                ctx.addFailureAtNode(node.name!, Rule.FAILURE_STRING);
            }
        }
        return ts.forEachChild(node, cb);
    });
}

function isPascalCased(name: string): boolean {
    return isUpperCase(name[0]) && !name.includes("_");
}
```

TSLint 规则示例

第一步，文件命名



TSLint 规则示例

规则命名必须是符合以下 2 个规则：

1. 驼峰命名。
2. 以 'Rule' 为后缀。

第二步，类命名

规则的类名是 Rule，并且要继承 Lint.Rules.AbstractRule 这个类型，当然也可能有继承 TypedRule 这个类的时候，但是我们通过阅读源码发现，其实它也是继承自 Lint.Rules.AbstractRule 这个类。

```
import { AbstractRule } from "./abstractRule";
import { ITypedRule, RuleFailure } from "./rule";

export abstract class TypedRule extends AbstractRule implements ITypedRule {

    public apply(): RuleFailure[] {
        // if no program is given to the linter, show an error
        showWarningOnce(`Warning: The '${this.ruleName}' rule requires type information.`);
        return [];
    }

    public abstract applyWithProgram(sourceFile: ts.SourceFile, program: ts.Program): RuleFailure[];
}
```

TSLint 规则示例

第三步，填写 metadata 信息

metadata 包含了配置参数，定义了规则的信息以及配置规则的定义。

- ruleName 是规则名，使用烤串命名法，一般是将类名转为烤串命名格式。
- description 一个简短的规则说明。
- descriptionDetails 详细的规则说明。
- rationale 理论基础。
- options 配置参数形式，如果没有可以配置为 null。
- optionExamples 参数范例 ，如没有参数无需配置。
- typescriptOnly true/false 是否只适用于 TypeScript。
- hasFix true/false 是否带有修复方式。
- requiresTypeInfo 是否需要类型信息。
- optionsDescrition options 的介绍。
- type 规则的类型。

规则类型有四种，分别为：”functionality”、”maintainability”、”style”、”type-script”。

- functionality ：针对于语句问题以及功能问题。
- maintainability：主要以代码简洁、可读、可维护为目标的规则。
- style：以维护代码风格基本统一的规则。
- typescript：针对于 TypeScript 进行提示。

第四步，定义错误提示信息

```
public static FAILURE_STRING = "Class name must be in pascal case";
```

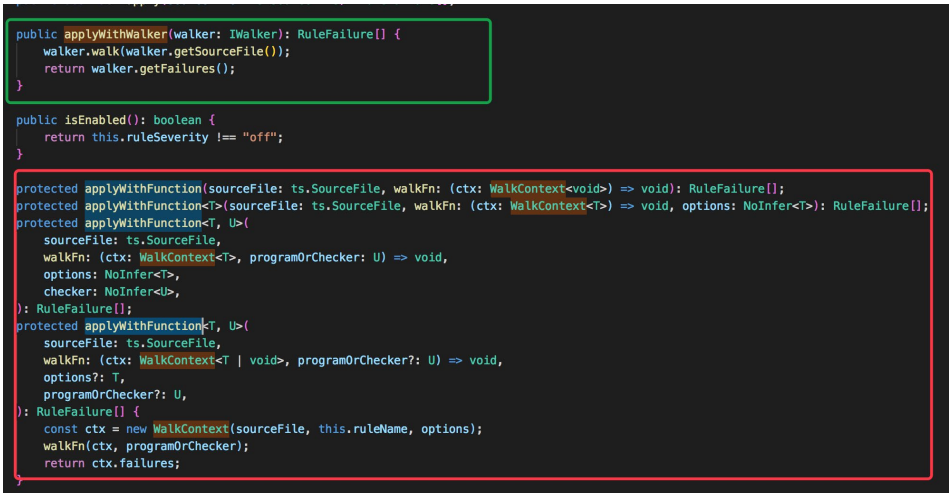
TSLint 错误信息

这个主要是在检查出问题的时候进行提示的文字，并不局限于使用一个静态变量的形式，但是大部分官方规则都是这么编写，这里对此进行介绍，防止引起歧义。

第五步，实现 apply 方法

apply 主要是进行静态检查的核心方法，通过返回 applyWithFunction 方法或者返回 applyWithWalker 来进行代码检查，其实 applyWithFunction 方法与

applyWithWalker 方法的主要区别在于 applyWithWalker 可以通过 IWalker 实现一个自定义的 IWalker 类，区别如下：



```
public applyWithWalker(walker: IWalker): RuleFailure[] {
    walker.walk(walker.getSourceFile());
    return walker.getFailures();
}

public isEnabled(): boolean {
    return this.ruleSeverity !== "off";
}

protected applyWithFunction(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<void>) => void): RuleFailure[];
protected applyWithFunction<T>(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<T>) => void, options: NoInfer<T>): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T>, programOrChecker: U) => void,
    options: NoInfer<T>,
    checker: NoInfer<U>,
): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T | void>, programOrChecker?: U) => void,
    options?: T,
    programOrChecker?: U,
): RuleFailure[] {
    const ctx = new WalkContext(sourceFile, this.ruleName, options);
    walkFn(ctx, programOrChecker);
    return ctx.failures;
}
```

TSLint

其中实现 IWalker 的抽象类 AbstractWalker 里面也继承了 WalkContext，

```
import * as ts from "typescript";

import { RuleFailure } from "../rule/rule";
import { WalkContext } from "./walkContext";

export interface IWalker {
    getSourceFile(): ts.SourceFile;
    walk(sourceFile: ts.SourceFile): void;
    getFailures(): RuleFailure[];
}

export abstract class AbstractWalker<T> extends WalkContext<T> implements IWalker {
    public abstract walk(sourceFile: ts.SourceFile): void;

    public getSourceFile() {
        return this.sourceFile;
    }

    public getFailures() {
        return this.failures;
    }
}
```

TSLint

而这个 WalkContext 就是上面提到的 applyWithFunction 的内部实现类。

```
protected applyWithFunction(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<void>) => void): RuleFailure[];
protected applyWithFunction<T>(sourceFile: ts.SourceFile, walkFn: (ctx: WalkContext<T>) => void, options: NoInfer<T>): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T>, programOrChecker: U) => void,
    options: NoInfer<T>,
    checker: NoInfer<U>,
): RuleFailure[];
protected applyWithFunction<T, U>(
    sourceFile: ts.SourceFile,
    walkFn: (ctx: WalkContext<T | void>, programOrChecker?: U) => void,
    options?: T,
    programOrChecker?: U,
): RuleFailure[] {
    const ctx = new WalkContext(sourceFile, this.ruleName, options);
    walkFn(ctx, programOrChecker);
    return ctx.failures;
}
```

TSLint

第六步，语法树解析

无论是 applyWithFunction 方法还是 applyWithWalker 方法中的 IWalker 实现都传入了 sourceFile 这个参数，这个相当于文件的根节点，然后通过 ts.forEachChild 方法遍历整个语法树节点。

这里有两个查看 AST 语法树的工具：

- AST Explorer：https://astexplorer.net/

  对应源码：https://github.com/fkling/astexplorer
- TypeScript AST Viewer：https://ts-ast-viewer.com/

  对应源码：https://github.com/dsherret/ts-ast-viewer

AST Explorer

优点：

在 AST Explorer 可以高亮显示所选中代码对应的 AST 语法树信息。

缺点：

1. 不能选择对应版本的解析器，导致显示的语法树代码版本固定。



TSLint

2. 语法树显示的信息相对较少。

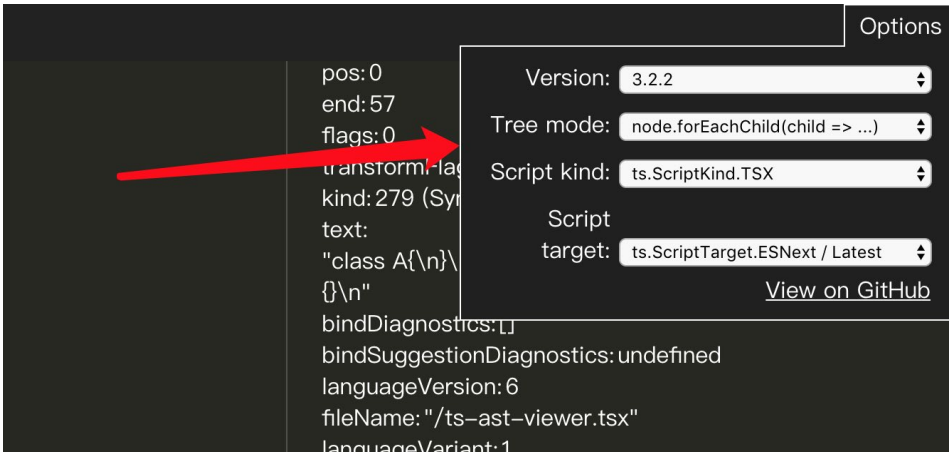
```
kind: 203
- name: Identifier = $node {
    pos: 41
    end: 53
    flags: 0
    escapedText: "invalidName"
    text: "invalidName"
    kind: 71
    *leadingComments: undefined
    *trailingComments: undefined
  }
  typeParameters: undefined
```

TSLint

TypeScript AST Viewer

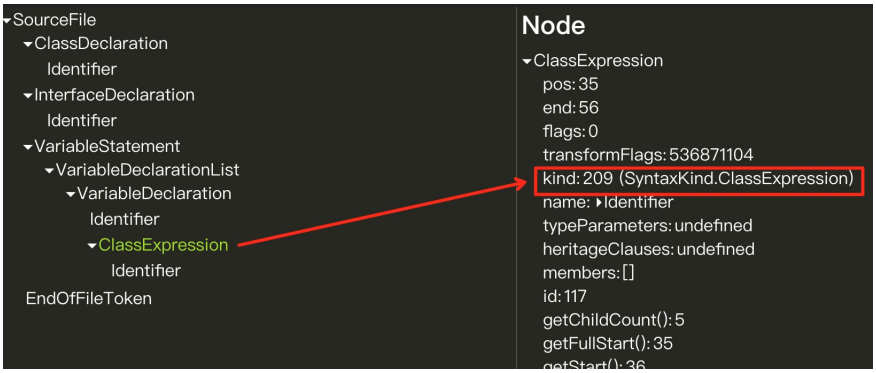
优点：

1. 解析器对应版本可以动态选择：



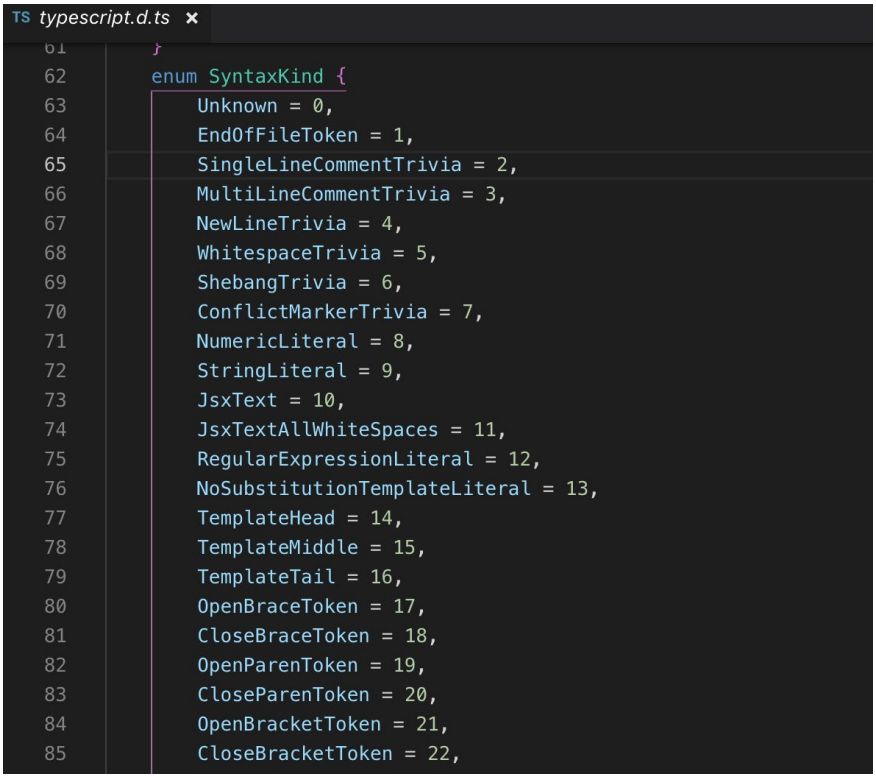
TSLint

2. 语法树显示的信息不仅显示对应的数字代码，还可为对应的实际信息：



TSLint

每个版本对应对 kind 信息数值可能会变动，但是对应的枚举名字是固定的，如下图：



TSLint

从而这个工具可以避免频繁根据其数值查找对应信息。

缺点：不能高亮显示代码对应的 AST 语法树区域，定位效率较低。

综上，通过同时使用上述两个工具定位分析，可以有效地提高分析效率。

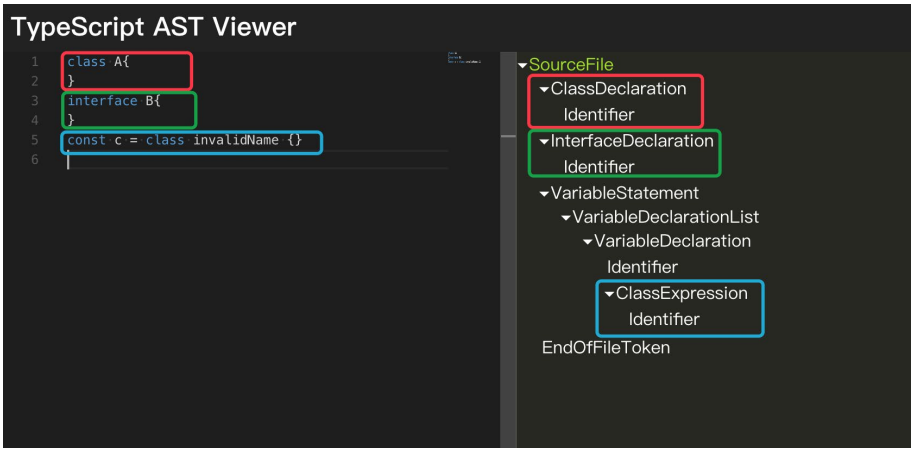
第七步，检查规则代码编写

通过 ts.forEachChild 方法对于语法树所有的节点进行遍历，在遍历的方法里可以实现自己的逻辑，其中节点的类为 ts.Node：

```
interface Node extends TextRange {
    kind: SyntaxKind;
    flags: NodeFlags;
    decorators?: NodeArray<Decorator>;
    modifiers?: ModifiersArray;
    parent?: Node;
}
```

TSLint

其中 kind 为当前节点的类型，当然 Node 是所有节点的基类，它的实现还包括 Statement、Expression、Declaration 等，回到开头这个 "class-name" 规则，我们的所有声明类主要是 class 与 interface 关键字，分别对应 ClassExpression、ClassDeclaration、InterfaceDeclaration，我们可以通过上步提到的 AST 语法树工具，在语法树中看到其为一一对应的。



TSLint

在规则代码中主要通过 isClassLikeDeclaration、isInterfaceDeclaration 这两个方法进行判断的。

```
function walk(ctx: Lint.WalkContext<void>) {
    return ts.forEachChild(ctx.sourceFile, function cb(node: ts.Node): void {
        if (isClassLikeDeclaration(node) && node.name !== undefined ||
            isInterfaceDeclaration(node)) {
            if (!isPascalCased(node.name!.text)) {
                ctx.addFailureAtNode(node.name!, Rule.FAILURE_STRING);
            }
        }
        return ts.forEachChild(node, cb);
    });
}
```

TSLint

其中 isClassLikeDeclaration、isInterfaceDeclaration 对应的方法我们可以在 node.js 文件中找到：

```
function isClassLikeDeclaration(node) {
    return node.kind === ts.SyntaxKind.ClassDeclaration ||
        node.kind === ts.SyntaxKind.ClassExpression;
}
exports.isClassLikeDeclaration = isClassLikeDeclaration;
```

TSLint

```
function isInterfaceDeclaration(node) {
    return node.kind === ts.SyntaxKind.InterfaceDeclaration;
}
exports.isInterfaceDeclaration = isInterfaceDeclaration;
```

TSLint

判断是对应的类型时，调用 addFailureAtNode 方法把错误信息和节点传入，当然还可以调用 addFailureAt、addFailure 方法。

```
export class WalkContext<T> {
    public readonly failures: RuleFailure[] = [];

    constructor(public readonly sourceFile: ts.SourceFile, public readonly ruleName: string, public readonly options: T) {}

    /** Add a failure with any arbitrary span. Prefer `addFailureAtNode` if possible. */
    public addFailureAt(start: number, width: number, failure: string, fix?: Fix) {
        this.addFailure(start, start + width, failure, fix);
    }

    public addFailure(start: number, end: number, failure: string, fix?: Fix) {
        const fileLength = this.sourceFile.end;
        this.failures.push(
            new RuleFailure(this.sourceFile, Math.min(start, fileLength), Math.min(end, fileLength), failure, this.ruleName, fix),
        );
    }

    /** Add a failure using a node's span. */
    public addFailureAtNode(node: ts.Node, failure: string, fix?: Fix) {
        this.addFailure(node.getStart(this.sourceFile), node.getEnd(), failure, fix);
    }
```

TSLint

最终这个规则编写结束了，有一点再次强调下，因为每个版本所对应的类型代码可能不相同，当判断 kind 的时候，一定不要直接使用各个类型对应的数字。

第八步，规则配置使用

完成规则代码后，是 ts 后缀的文件，而 ts 规则文件实际还是要用 js 文件，这时候我们需要用命令将 ts 转化为 js 文件：

```
tsc ./src/*.ts --outDir dist
```

将 ts 规则生成到 dist 文件夹（这个文件夹命名用户自定），然后在 tslint.json 文件中配置生成的规则文件即可。

```
{} tslint.json ×
1   {
2     "rulesDirectory": "./dist",
3     "rules": {
4       "check-if-number-zero": true
5     }
6   }
7
```

TSLint

之后在项目的根目录里面，使用以下命令既可进行检查：

```
tslint --project tsconfig.json --config tslint.json
```

同时为了未来新增规则以及规则配置的更好的操作性，建议可以封装到自己的规则包，以便与规则的管理与传播。

总结

TSLint 的优点：

1. 速度快。相对于动态代码检查，检查速度较快，现有项目无论是在本地检查，还是在 CI 检查，对于由十余个页面组成的 React Native 工程，可以在 1 到 2 分钟内完成；
2. 灵活。通过配置规则，可以有效地避免常见代码错误与潜在的 Bug；
3. 易扩展。通过编写配置自定义规则，可以及时准确快速查找出代码中特定风险点。

TSLint 缺点：

1. 规则的结果只有对与错两种等级结果，没有警告等级的的提示结果；
2. 无法直接报告规则报错数量，只能依赖其他手段统计；
3. TSLint 规则针对于当前单一文件可以有效地通过语法树进行分析判定，但对于引用到的其他文件中的变量、类、方法等，则难以通过 AST 语法树进行判定。

使用结果及分析

在美团，有十余个页面的单个工程首次接入 TSLint 后，检查出的问题有近百条。但是由于开启的规则不同，配置规则包的差异，检查后的数量可能为几十条到几千条甚至更多。现在已开发十余条自定义规则，在单个工程内，处理优化了数百处可能存在问题的代码。最终 TSLint 接入了相关 React Native 开发团队，成为了代码提交阶段的必要步骤。

通过团队内部的验证，文章开头遇到的问题得到了有效地缓解，目标基本达到预期。TSLint 在 React Native 开发过程中既保证了代码风格的统一，又保证了 React

Native 开发人员的开发质量，避免了许多低级错误，有效地节省了问题排查和人员沟通的成本。

同时利用自定义规则，能够将一些兼容性问题在内的个性化问题进行总结与预防，提高了开发效率，不用花费大量时间查找问题代码，又避免了在一个问题上跌倒多次的情况出现。对于不同经验的开发者而言，不仅可以进行友好的提示，也可以帮助快速地定位问题，将一个人遇到的经验教训，用极低的成本扩散到其他团队之中，将开发状态从“亡羊补牢”进化到“防患未然”。

作者简介

家正，美团点评 Android 高级工程师。2017 年加入美团点评，负责美团大交通的业务开发。

ESLint 在中大型团队的应用实践

宋鹏

引言

代码规范是软件开发领域经久不衰的话题，几乎所有工程师在开发过程中都会遇到，并或多或少会思考过这一问题。随着前端应用的大型化和复杂化，越来越多的前端工程师和团队开始重视 JavaScript 代码规范。得益于前端开源社区的繁盛，当下已经有几种较为成熟的 JavaScript 代码规范检查工具，包括 JSLint、JSHint、ESLint、FECS 等等。本文主要介绍目前较为通用的方案——ESLint，它是一款插件化的 JavaScript 代码静态检查工具，其核心是通过对代码解析得到的 AST（Abstract Syntax Tree，抽象语法树）进行模式匹配，定位不符合约定规范的代码。

ESLint 的使用并不复杂。依照 ESLint 的文档安装相关依赖，可以根据个人 / 团队的代码风格进行配置，即可通过命令行工具或借助编辑器集成的 ESLint 功能对工程代码进行静态检查，发现和修复不符合规范的代码。如果想降低配置成本，也可以直接使用开源配置方案，例如 eslint-config-airbnb 或 eslint-config-standard。

对于独立开发者，或者执行力较强、技术场景较为单一的小型团队而言，直接使用 ESLint 及其生态提供的一些标准方案，可以用较低成本来实现 JavaScript 代码规范的落地。如果再搭配一些辅助工具（例如 **husky** 和 **lint-staged**），整个流程会更加顺畅。但对于数十人的大型前端团队来说，面向数百个前端工程，规模化地应用统一的 JavaScript 代码规范，问题就会变得较为复杂。如果直接利用现有的开源配置方案，可能会使工作事倍功半。

问题分析

规模化应用统一的 ESLint 代码规范，会涌现各类问题，根源在于大型团队和小团队（或独立开发者）的差异性：

技术层面上：

- **技术场景更加广泛：**对于大型团队，其开发场景一般不会局限在传统 Web 领域内，往往还会涉及 Node.js、React Native、小程序、桌面应用（例如 Electron）等更广泛的技术场景。
- **技术选型更加分散：**团队内工程技术选型往往并不统一，如 React/Vue、JavaScript/TypeScript 等。
- **工程数量的增加和工程方案离散化导致 ESLint 方案的复杂度提升：**这样会进一步增加工程接入成本、升级成本和方案维护成本。

在团队层面，随着人员的增加和组织结构的复杂化：

- 人员风格差异性更大、沟通协调成本更高。
- 方案宣导更难触达，难以保证规范执行的落实。
- 执行状况和效果难以统计和分析。

因为存在诸多差异，我们在设计具体方案时，需要考虑和解决更多问题，以保证规范的落实。针对上述分析，我们梳理了以下需要解决的问题：

- 如何制定统一的代码规范和对应的 ESLint 配置?
  - **场景支撑：**如何实现对场景差异的支持？如何保证不同场景间一致部分（例如 JavaScript 基础语法）的规范一致性?
  - **技术选型支撑：**如何在支撑不同技术选型的前提下，保证基础规则（例如缩进）的一致性?
  - **可维护性：**具体到规则配置上，能否提升可复用性？在方案升级迭代时成本是否可控?
- 如何保证代码规范的执行?
  - 人员的增加和组织结构的复杂化，会导致基于管理的执行把控失效，这种情况应该如何保证代码规范的执行质量?
- 如何降低应用成本?
  - 在工程数量增加、工程方案离散化的情况，降低方案的接入、升级和执行

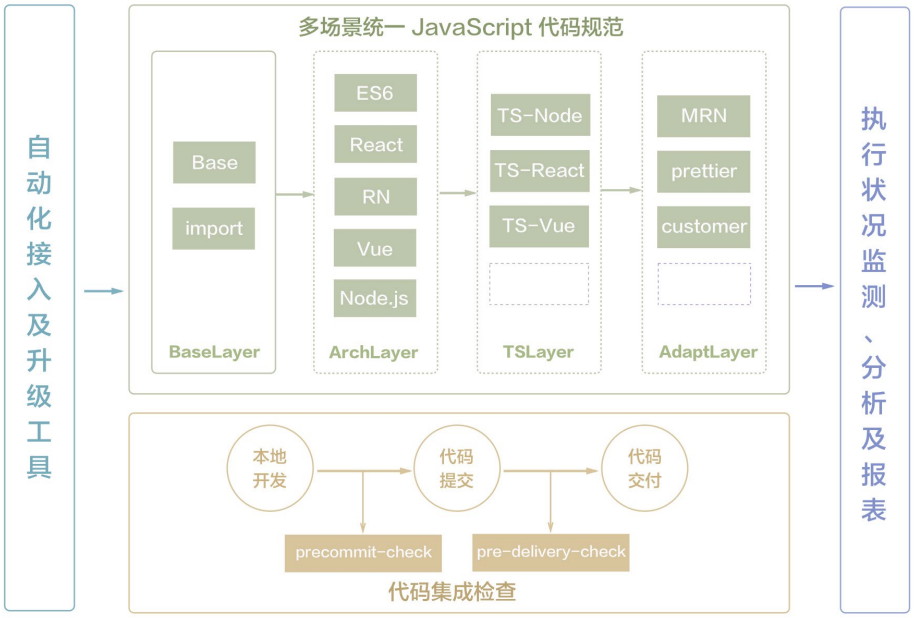
成本能节约大量的人力，同时也有利于方案落地推进。

- 如何及时了解规范应用状况和效果?

解决方案

为了能在团队内实现 JavaScript 代码规范的统一，在分析和思考团队规模化应用存在的问题后，我们设计了一套完整的技术解决方案。该方案包括多场景统一的 ESLint 规则配置、代码集成交付检查、自动化接入工具、执行状况监测分析等四个模块。通过各个模块协调配合，共同解决上文提出的问题，在降低维护成本、提升执行效率的同时，也保障了代码规范的统一。

整体方案的设计如下图所示：



1. **多场景统一的 JavaScript 规范：**该模块是整个方案的核心，借助 ESLint 的特性，通过分层分类的结构设计，在保证基础规则一致性的同时，实现了对不同场景、技术选型的支撑。
2. **代码集成交付检查：**该模块是方案落地执行的保障，将代码静态检查集成到持续交付工作流中。具体设计实现上，在保证交付质量的同时，也通过定制集

成检查工具降低了开发者的应用执行成本。

3. **自动化接入和升级方案：**通过命令行工具提供“一键”接入 / 升级能力，同时集成到团队脚手架中，大大降低了工程接入和维护的成本。
4. **执行状况监测分析：**通过对工具运行和代码集成交付检查过程进行埋点、检查结果收集和分析，了解方案的应用状态和效果。

方案实现

上文中提出的问题，通过各模块的协调配合能够得到有效地解决，但具体到各个模块的实现，仍然需要进一步深入思考，以设计出更加合理的实现方案。本章将对方案的四个核心模块进行详细介绍。

通用 ESLint 配置方案

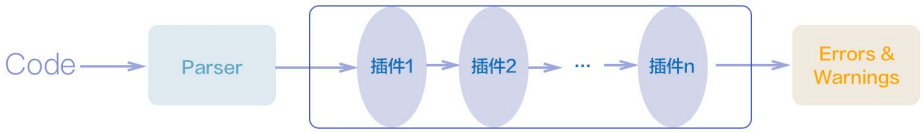
这一模块主要借助 ESLint 的基础特性，采用分层分类的结构设计，提供多场景、多技术方案的通用配置方案，并使方案具备易维护、易扩展的特性。

ESLint 特性简介

在进行 ESLint 配置方案设计前，我们先看一下 ESLint 的一些特点。

1. 插件化

下图简单地描述了 ESLint 的工作过程：



ESLint 的能力更像一个引擎，通过提供的基础检测能力和模式约束，推动代码检测流程的运转。原始代码经过解析器的解析，在管道中逐一经过所有规则的检查，最终检测出所有不符合规范的代码，并输出为报告。借助插件化的设计，不但可以对所有的规则进行独立的控制，还可以定制和引入新的规则。ESLint 本身并未和解析器强绑定，我们可以使用不同的解析器进行原始代码解析，例如可以使用

babel-eslint 支持更新版本、不同阶段的 ES 语法，支持 JSX 等特殊语法，甚至可以借助 **@typescript-eslint/parser** 支持 **TypeScript** 语言的检查。

2. 配置能力全面、可层叠、可共享

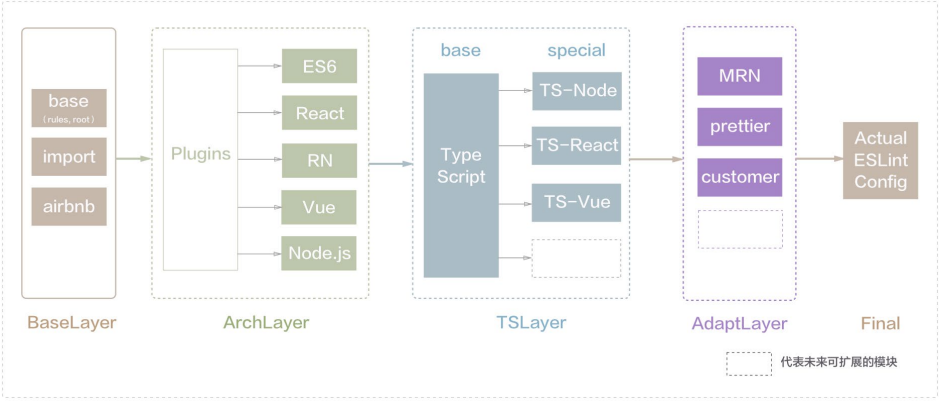
ESLint 提供了全面、灵活的配置能力，可以对解析器、规则、环境、全局变量等进行配置；可以快速引入另一份配置，和当前配置层叠组合为新的配置；还可以将配置好的规则集发布为 npm 包，在工程内快速应用。

3. 社区生态较为成熟

开源社区中基于 ESLint 的项目非常多，既有针对各种场景、框架的插件，也有各种 ESLint 规则配置方案，基本可以涵盖前端开发的所有场景。

规范配置方案设计

基于 ESLint 的插件化、可层叠配置特性，以及面向各种场景、框架的开源方案，我们设计了如下图所示的 ESLint 配置架构：



该配置架构采用了分层、分类的结构，其中：

- **基础层：**制定统一的基础语法和格式规范，提供通用的代码风格和语法规则配置，例如缩进、尾逗号等等。
- **框架支撑层（可选）：**提供对通用的一些技术场景、框架的支持，包括 Node.js、React、Vue、React Native 等；这一层借助开源社区的各种插件进行配

置，并对各种框架的规则都进行了一定的调整。

- **TypeScript 层（可选）：**这一层借助 typescript-eslint，提供对 TypeScript 的支持。
- **适配层（可选）：**提供对特殊场景的定制化支持，例如 MRN（美团内部的 React Native 定制化方案）、配合 prettier 使用、或者某些团队的特殊规则诉求。

具体的实际项目中，可以灵活的选择各层级、各类型的搭配，获得和项目匹配的 ESLint 规则集。例如，对于使用 TypeScript 语言的 React 项目，可以将基础层、框架层的 React 分支、以及 TypeScript 支撑层的 React 分支层叠到一起，最终形成适用于该项目的 ESLint 配置。如果项目不再使用 TypeScript 语言，只需要将 ts-react 这一层去掉即可。

最终，形成了如下所示的 ESLint 配置集：



考虑到维护、升级和应用成本，我们最终选择将所有配置放到一个 npm 包中，而不是每种类型分别设置。仍以使用 TypeScript 语言的 React 项目为例，只需在工程中进行如下配置：

```
// 需要安装 typescript、eslint-plugin-react、@typescript-eslint 等插件
module.exports = {
  root: true,
  extends: [
    // 因为基础层是必备的，所以框架层默认引入了对应的基础层，不需再单独引入 eslintrc.base.js
    'eslint-config-xxx/eslintrc.react.js',
    'eslint-config-xxx/eslintrc.typescript-react.js'
  ]
}
```

这种通过分层、分类的结构设计，还有利于后期的维护：

- 对基础层的修改，只需修改一处即会全局生效。
- 对非基础层某一部分的调整不会产生关联性的影响。
- 如需扩展对某一类型的支持，只需关注这一类型的特殊规则配置。

众所周知，TypeScript 类型的项目使用 TSLint 进行代码检查，也是一种简单、便捷的方案。但在本方案中我们依旧选择了：eslint + @typescript-eslint/parser + @typescript-eslint/eslint-plugin 的组合方案。主要有以下几点原因：

- ESLint 的规则配置更加详细全面，覆盖更加广泛。
- 采用了分层分类的架构，能够保证即使框架或语言不同，也能在基本语法、风格层面保持规则的一致性，这样有利于团队内不同技术选型项目的风格统一。
- @typescript-eslint 方案持续迭代，问题响应非常迅速，对 TSLint 相关的规则基本提供了对等的实现。

根据最新消息，TypeScript 在 2019 路线图 中明确表明后续对 Lint 工具的支持和建设会以对 ESLint 进行适配的方式为主。

代码集成检查

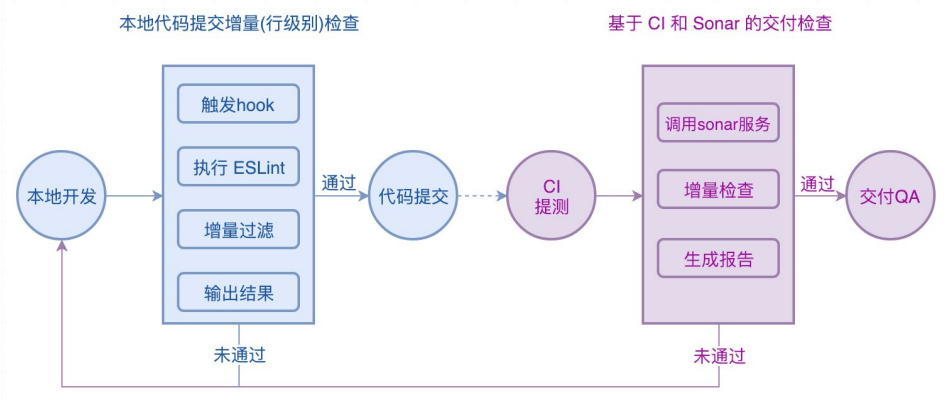
基于团队对工程化基础设施的建设，将代码规范静态检查与开发工作流集成，保证代码规范的落实。

通常而言，工程接入 ESLint 后，可以在开发的同时借助编辑器集成的 ESLint 检查提示能力（例如 VSCode 的 ESLint 插件），实时发现和修改不符合规范的语法错误和风格问题。但这仍不能避免因一些主观因素或疏漏造成的规范执行不到位，所以我们考虑在开发工作流的特定节点自动执行代码静态检查，阻断不合规范代码的提交或交付。

集成静态检查的开发工作流节点有很多，我们主要参考以下两种方案：

- **代码提交检查：**在代码 Commit 时，通过 githook 触发 ESLint 检查。其优点在于能实时响应开发者的动作，给出反馈，快速定位和修复问题；缺陷在于开发者可以主动跳过检查。
- **代码交付检查：**在代码交付（借助 CI 系统的交付流程功能）时，在代码检测平台中对代码进行 ESLint 检查，检测不通过则阻断交付。其优点在于能够强制执行，可在线追踪检测报告；缺陷在于离开发者的开发环境太“远”，开发者响应处理成本较高。

如果将两者进行结合，可能会事半功倍，效果如下图所示：



常用的代码提交检查方法一般是 husky 与 lint-staged 结合，在代码 Commit 时，通过 githook 触发对 git 暂存区文件的检查。但考虑到团队现有工程数量庞大、存在大量行数较多的文件，虽然 lint-staged 策略能够降低部分成本，但仍稍显不足。为此，我们对该方法进行优化，定制了本地代码提交检查工具 precommit-eslint，其核心特点是：

- 将增量检查执行到代码行这一粒度，支持 Warn 和 Error 两个检查级别。
- 只需将工具安装为工程的依赖，无需任何配置。
- 减少了 pre-commit hook 中植入脚本的侵入性。
- 进行了执行状况埋点和采集。

使用效果如下图所示：



在美团，我们使用自主开发的 CI 系统，并在独立部署的 Sonar 系统上定制化实现了相应规则，基本可以满足诉求，这里就不再赘述。对于独立的团队，基于 ESLint 提供的工具，可以很容易的实现使用 Node 快速搭建一个代码检测服务或平台，大家有兴趣不妨一试。

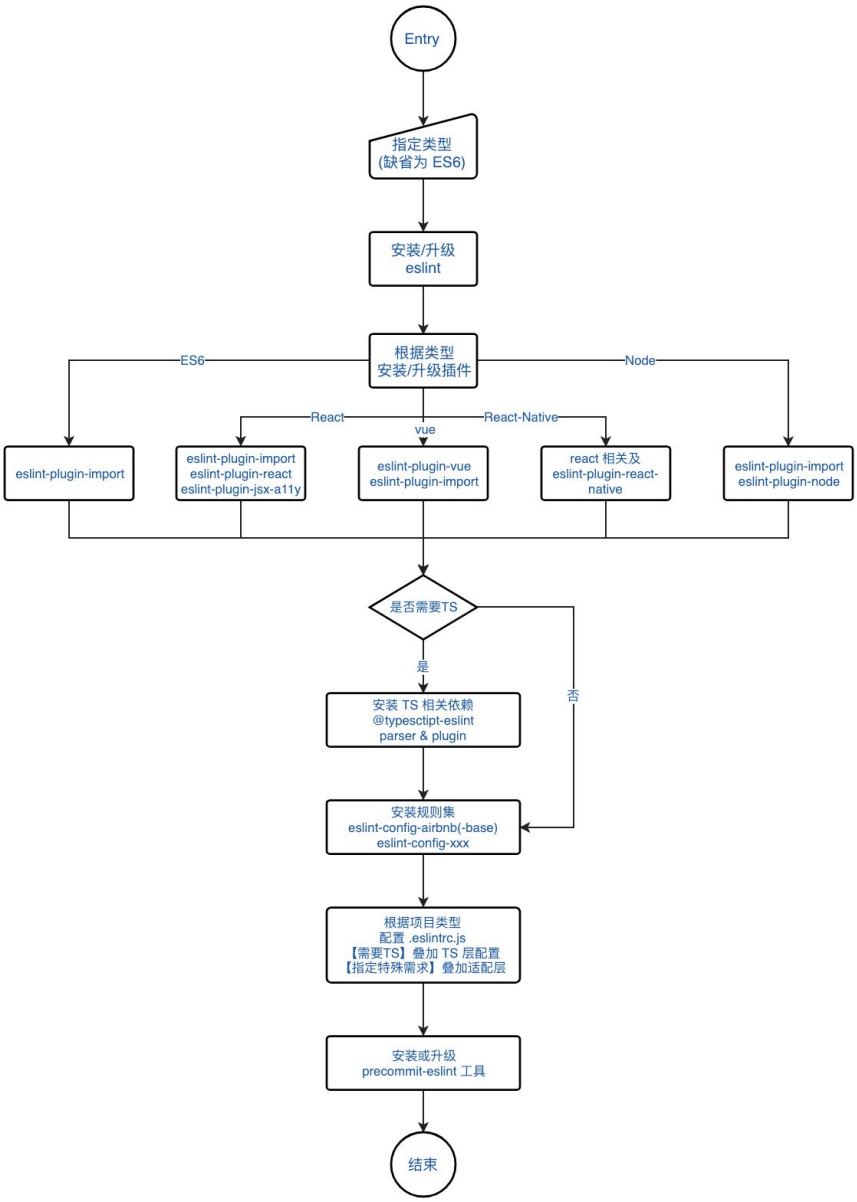
自动化接入工具

这个模块主要通过 CLI 工具提供方案自动化接入的能力，降低工程接入和升级的成本。如果不借助自动化工具，在工程中接入上述方案还是有一定的工作量和复杂度的，大致步骤如下：

1. 安装 Eslint。
2. 根据项目类型安装对应的 ESLint 规则配置 npm 包。
3. 根据项目类型安装相关的插件、解析器等。
4. 根据项目类型配置 .eslintrc 文件。
5. 安装代码提交检查工具。
6. 配置 package.json。
7. 测试及修复问题。

在这个过程中，特别需要注意依赖的版本问题：依赖之间的版本兼容性，例如 typescript 和 @typescript-eslint/parser 之间的兼容性；依赖对规则的支持性，比如某个版本的插件中去除了对某个规则的支持，但规则配置中仍然配置了该规则，此时配置就会失效。对于 ESLint 不熟悉的开发者而言，在配置的过程中都会或多或少遇到兼容性、解析异常、规则无效等问题，反复排查和定位问题会浪费大量的精力。

因此，在设计开发自动化接入工具时，我们综合考虑了操作步骤、依赖版本、规则集和工程方案的兼容性，设计了如下的工作流程：

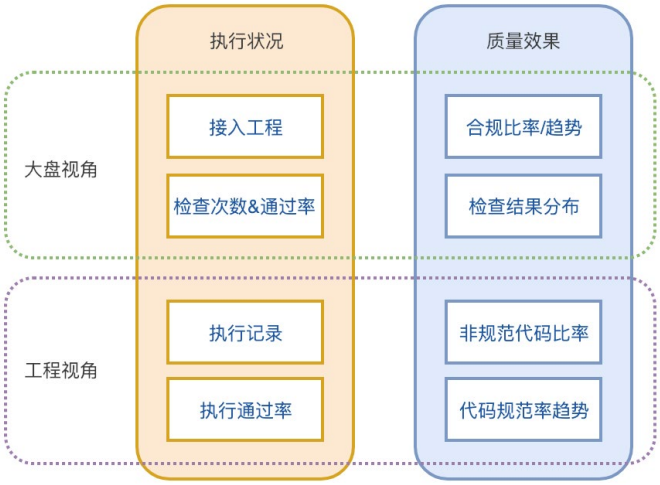


该工具流程简单，不管什么开发场景和框架选型，繁琐的接入流程都可以简化为一条命令，需要配合工程方案升级时同样如此。如下图所示，执行该命令后项目就完成了 ESLint 的接入，使用统一的规则规范编码，同是在代码提交时自动进行增量检查：

```
➜ test git:(master) ✗ eslint-init --react --ts --neverAutoInstall
正在安装 eslint 相关依赖 ...
eslint@5.12.1
babel-eslint@10.0.1
eslint-plugin-import@2.15.0
eslint-plugin-react@7.12.4
eslint-plugin-jsx-a11y@6.2.0
eslint-config-airbnb@17.1.0
@typescript-eslint/parser@1.1.1
@typescript-eslint/eslint-plugin@1.1.1
eslint 依赖安装完成
正在配置 eslint...
正在安装 eslint 配置集
eslint 配置集安装完成
检测到该项目尚无 eslintrc.js 配置文件
复制标准 eslintrc.js 配置模板到项目空间...
eslint配置完成
如果该项目中已经存在 eslintrc.js 之外的其他eslint配置文件，可以删除~
默认使用工程目录下的 .tsconfig.json 文件帮助 typescript 的检验
如配置文件非该路径，请自行配置：https://github.com/typescript-eslint/typescript-eslint/tree/master/packages/parser#configuration。
eslint 配置完成
正在设置持续集成检查方案
开始配置package.json...
持续集成检查方案配置成功
eslint初始化完成，happy coding~
```

埋点与统计分析

统计分析的主要目的是掌握方案应用执行状况和效果，理论上应当支持工程和大盘两个视角，如下图所示：

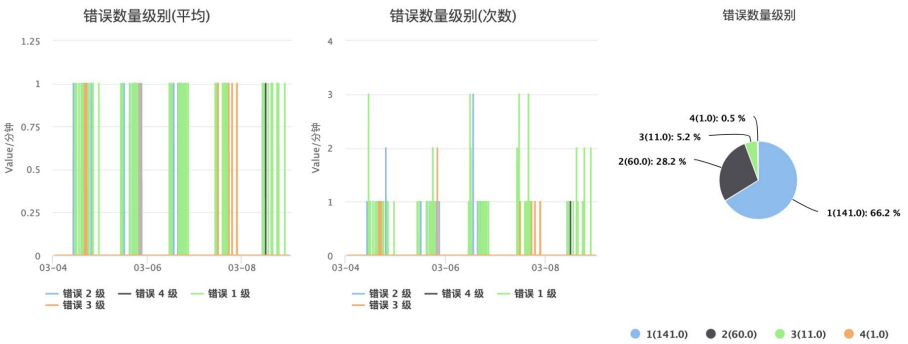


执行情况分析其实并不复杂，核心是信息采集和分析。在本方案中，信息采集通过 precommit-eslint 工具实现：在 git commit 触发本地代码检查后，脚本会把检查结果（包括检查是否通过、错误或警告信息的数量级别等）上报；信息的统计分析借助日志上报分析平台实现，美团使用的是 CAT 平台（如果团队或公司没有专门的平台，可以在上文提到的代码检测服务平台中实现这部分功能）。为了便于数据的聚合分析，我们将一次代码提交检查中出现的问题数量进行了分级：

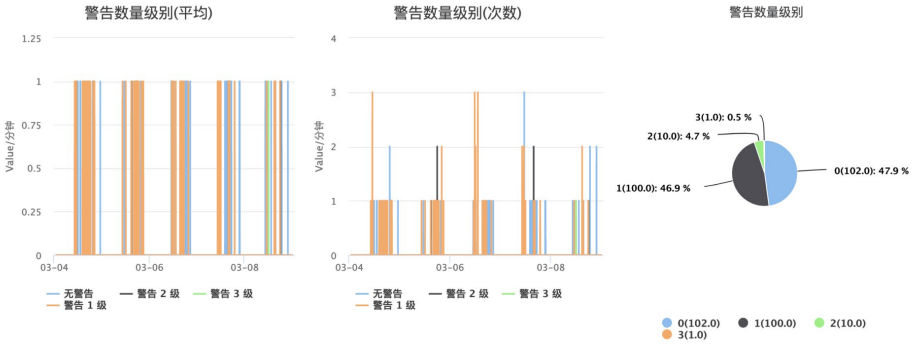
- 检查通过：检查无代码规范错误。
- 错误 1 级：检查出代码规范错误数量小于 10 个。
- 错误 2 级：检查出代码规范错误数量在 10 – 100 个之间。
- 错误 3 级：检查出代码规范错误数量在 100 – 1000 个之间。
- 错误 4 级：检查出代码规范错误数量大于 1000 个。

比如下图中，**201903 第一周**的代码提交检查结果统计（综合采样率 0.2），很明显，所有检查失败的提交中，错误数量在 10 个以内的占比最大，修复成本不高。

1. 提交检查异常分布（仅筛选检查未通过信息）



2. 提交检查警告信息分析



除此之外，还可以对单一工程，在更细的时间粒度上去观察提交检查的执行情况。

效果质量主要分析工程质量的变化：一方面可以通过代码检查执行通过率变化趋势、检查结果分布去看持续的生产流程中，代码质量是否有所提升；另一方面，由于代码检查采用增量模式，需要对工程代码进行整体分析，得到工程整体的不规范代码占比及变化趋势，从而从工程维度分析判断质量效果（涉及到权限相关问题，目前团队中未采用工程分析的方法）。具体的分析会在方案应用效果中一并进行介绍。

方案应用

除了上述整体方案外，为保证开发者使用更方便，我们还进行了一些配套工作：

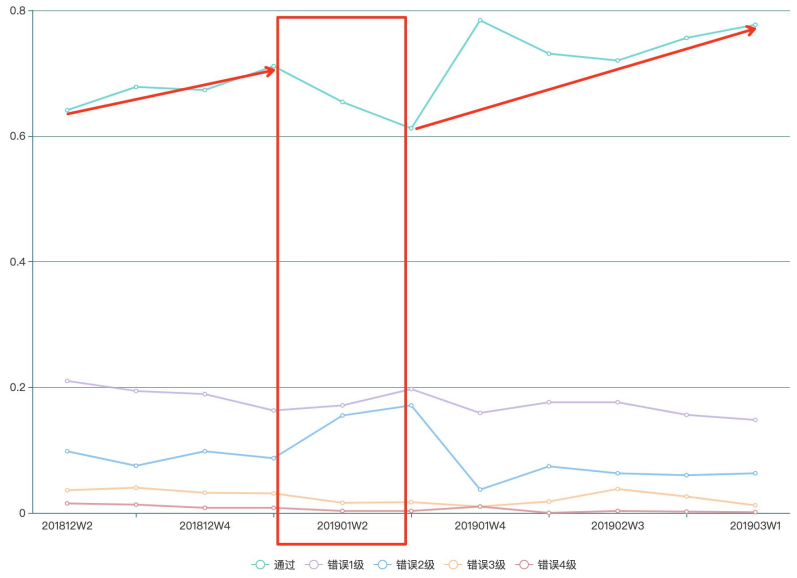
- 持续维护升级：以每月一版的方式持续迭代升级，解决应用中的问题、规则争议，以及支持新的规则或方案。
- 工程化集成：整套方案可以无缝接入到各个团队的脚手架工具中，自动成为团队的默认方案，在工程初始化阶段即可完成接入。
- 官网建设：提供详细的使用文档，包括规则信息、接入方法，并且对每个版本提供规则、环境依赖、changeLog 等详细说明。
- 常见使用问题：更新维护 FAQ，帮助后续接入者快速查找并解决问题。

目前，该套方案已经接入美团外卖、餐饮平台、闪购、榛果、金融等多个团队，基于埋点统计分析，我们（基于 2019 年 2 月份最后一周统计数据分析，综合采样率

0.2）得到了如下数据：

- 截止到 2019 年 2 月底，该方案已接入超过 200 个前端工程。
- 集成检查（增量）每天执行接近 1000 次。
- 集成检查（增量）平均每天检查出错误约 20000–25000 处。
- 集成检查代码质量：平均通过率为 75.562%，错误 1 级的比率为 15.644%，在所有未通过检查中占比 64.015%。

同时，我们持续统计上述数据的变化趋势，跟踪代码质量提升效果，以 2018 年 12 月到 2019 年 3 月的数据为例（截止 2019 年 3 月第一周，以周为时间统计尺度）：



从图中可以看出，最近三个月检查通过率整体呈上升趋势，但 2019 年 1 月的第 2 周和第 3 周集成检查通过率有明显下降。分析项目信息发现，在 2019 年 1 月的第 2 周有一批新项目接入，代码检查规范检查出几十个错误。但整体来看，目前集成检查通过率基本稳定在 75% – 80%，从变化趋势看仍有上升空间。

方案实施之后，我们做了一个用户调研，发现整体方案的运营正在发挥着正向的作用。一方面，在一定程度上提升了多人协作的效率，无论是共同维护一个工程还是

在多个工程间切换，避免了代码风格不一致带来的可读性成本和格式化风险；另一方面，会帮助大家发现和避免一些简单的语法错误。

规划和思考

该方案已经稳定应用，除了现有功能，我们还在思考是否可以更进一步的优化，提供更丰富的能力。由此规划了一些仍未落地的方向：

1. 扩展支持 HTML 和 CSS 的代码风格检查：虽然近几年前端框架、组件库的建设一定程度上减少了业务开发中（尤其是中后台业务）对 HTML 和 CSS 的需求，但是规范 HTML 和 CSS 的代码风格仍是必要的。基于此，可以用同样的思路将 HTML 和 CSS 的代码静态检查方案集成到当前的方案中，不再局限于 JavaScript（或 TypeScript）。
2. 进一步的封装：目前整体方案会将所有依赖和配置暴露在工程内，如果将其完全封装在一个工具内会更便于应用，但难点在于兼顾灵活性、对编辑器的支持等问题。
3. 增加工程维度的代码质量趋势分析：目前代码检查策略是增量检查，可以对接入的工程定期全量检查，基于时间线分析工程的代码质量变化趋势。
4. 进一步深入分析检查结果和统计数据，发现一些潜在问题，为推动开发质量提升提供辅助，如：
   - 统计开发者在工程中关闭或调整的规则，分析占比较高的规则被关闭的原因，进而调整规则或推动规则的执行。
   - 统计分布检查出错误的规则分布，梳理出最常出问题的代码规则，发布对应的最佳实践或手册。

以上是美团外卖团队在 ESLint 方案规模化应用过程中的一些实践，欢迎大家提出建议，一起沟通交流。

作者简介

宋鹏，美团外卖事业部终端研发工程师。

团队介绍

美团外卖事业部终端团队，负责的多个终端和平台直接连接亿万用户、数百万商家和几万名运营与销售，目标是在保障业务高稳定、高可用的同时，持续提升用户体验和研发效率。

在用户方向上，构建了全链路的高可用体系，客户端、Web 前端和小程序等多终端的可用性在 99% 左右；跨多端高复用的局部动态化框架在首页、广告、营销等核心路径的落地，提升了 30% 的研发效率；

在商家方向上，从提高进程优先级、VoIP Push 拉活、doze 等方面进行保活定制，并提供了 Shark、短链和 Push 等多条触达通道，订单到达率提升至 98% 以上；

在运营方向上，通过标准化研发流程、建设组件库和 Node 服务以及前端应用的管理与页面配置等提升 10% 的研发效率。

团队有多个岗位正在招聘，欢迎加入我们，联系邮箱 tech@meituan.com ，注明“外卖终端团队”。

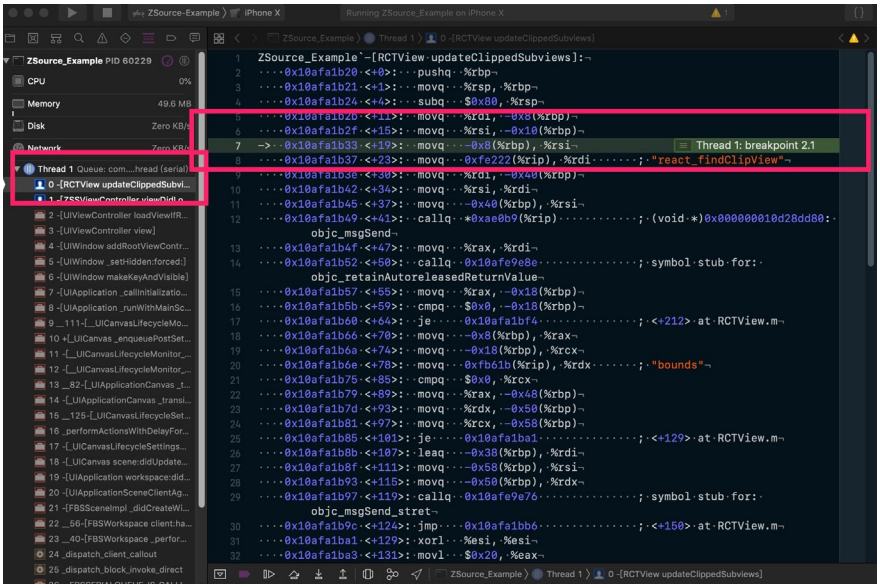
美团 iOS 工程 zsource 命令背后的那些事儿

宇杰

zsource 命令是什么?

美团 App 在 2015 年就已经基于 CocoaPods 完成了组件化的工作。在组件化的改造过程中，为了能够加速整体工程的构建速度，我们对需要集成进美团 App 的组件进行了二进制化，同时提供一个叫做 cocoapods-binary 的 CocoaPods 插件来支持本地工程使用二进制。因此，美团 App 的开发者在集成开发时，除了自己正在开发的组件，其他的组件都以二进制的形式存在。

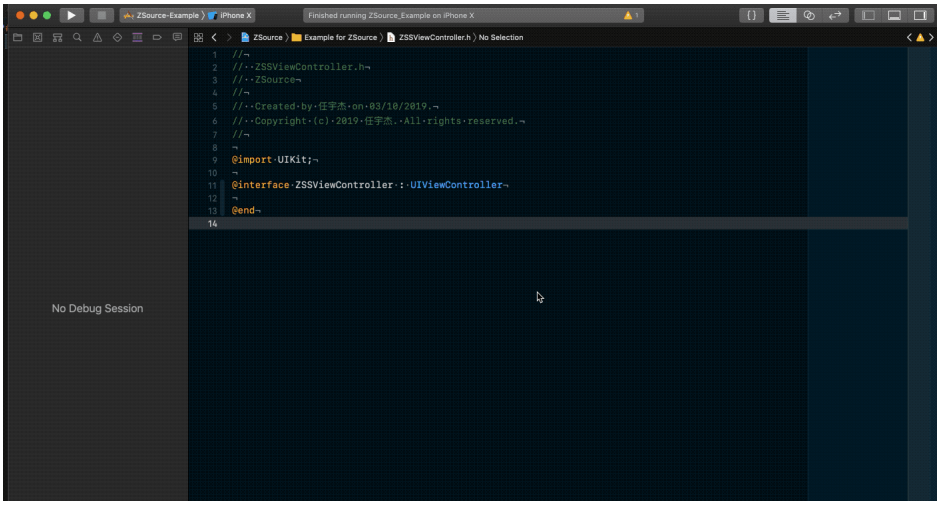
使用二进制，虽然会给工程带来构建速度的提升，但是会带来一个新的问题：在调试工程时，那些使用二进制的组件，无法像源码调试那样看到足够丰富的调试信息。例如，如果程序在二进制组件的代码中崩溃，我们只能看到该组件的堆栈信息和一些不明所以的汇编代码：



程序断点在二进制组件的代码中时的样子

和业界大多的组件化方案类似，美团 App 的组件化方案也提供了将一个组件从二进制切换到源码的机制。美团工程的开发者能够使用一系列配置和命令来切换组件的源码和二进制状态，但每次切换都需要重新执行 pod install。这种方式在组件化的初期是没有什么问题的。但随着美团 App 的组件数量不断增长，即便是只切换一个组件的状态，单次 pod install 的时间也增长到了分钟级。而且这种方式每切换一次就必须重新编译运行一次 App，在追查一些偶现崩溃问题时，开发体验非常不友好，也不利于崩溃问题的快速定位分析。

为了解决以上提到的这些问题，我们利用 CocoaPods 的插件机制，为 CocoaPods 的 pod 命令增加了 zsource 子命令，开发者可以在使用二进制构建工程的同时，非常快速地将一个组件调出源码进行调试，具体的使用效果可以看一下如下的屏幕录制：



zsource 的实际使用过程

zsource 命令的开发始末

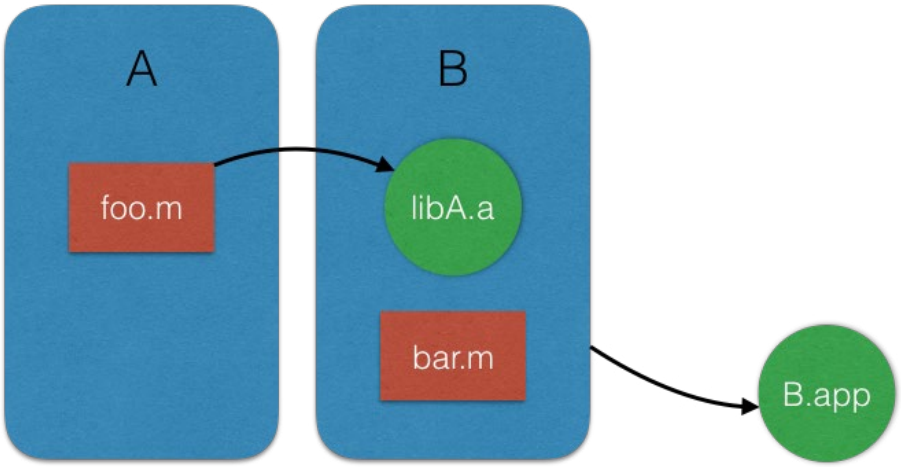
在推出 zsource 功能后，很多同学都对 zsource 背后的技术原理十分感兴趣。其实 zsource 整个功能的开发流程也十分的有趣，就像小说一样，分为几个不同的时期：

- 原理猜想
- 查阅资料
- 简单粗暴的尝试
- 柳暗花明
- 工程化

原理猜想

如果让我们猜想 Xcode 断点调试功能的实现原理，可能大部分人都会猜这样一种可能：Xcode 在编译 Debug 版本的二进制过程中，在二进制中某个字段存储了该二进制所对应的源码的文件地址。当我们在 Xcode 中打断点进行调试的时候，Xcode 会根据二进制中这个字段中存储的源码文件地址，打开对应的源码文件，并在 UI 上展示该源码文件。

道理好像没有什么问题，但是事实是这样吗？在某次团建回国的航班上，我们组成威和志宇两位同学在提出这种猜想后，拿出电脑，做了一个这样的小实验：



实验说明

实验中，他们分别创建了两个 Xcode 工程 A 和 B，工程 A 会产出一个二进制 libA.a。工程 B 中会接将 A 的产出 libA.a 拖到工程中，然后设置 A 中代码的符号断点，然后编译运行。结果发现，当断点断在 A 中的代码时，Xcode 会直接跳转到 A

的源文件中，并且可以继续增加断点以及正常的单步调试。

通过这个实验，成威和志宇同学确定了猜想的正确性。那么接下来需要做的，就是确定二进制中，这个源文件地址信息具体藏在哪一个字段中。

查阅资料

我们都知道苹果的 Mach-O 二进制文件使用的是 DWARF 这种格式来存放调试相关的数据的。但因为我们很难从这个问题中提炼几个精确的关键词在搜索引擎中检索，所以很难通过简单的几次检索就获取到我们想要的答案：二进制这个字段的名称，在初期甚至无法确定这个字段应该是从 Mach-O 的资料中检索还是从 DWARF 的资料中检索。

在没有太好的搜索结果的情况下，我们一度曾经想尝试去从头去啃一啃找到的一些二进制相关的文档：

- osx-abi-macho-file-format-reference
- Introduction to the DWARF Debugging Format
- DWARF 1.1.0 Reference

简单粗暴的尝试

然而，由于对二进制格式不是那么熟悉，也不太了解二进制相关的词汇和概念，所以阅读文档的速度就非常缓慢。

不过，技术的有趣之处就在于，有时候你可以基于我们的猜想，任意去尝试，跳过艰辛的文档阅读过程。在文档阅读遇到挫折后，我们猜想，二进制中很有可能也是用字符来存储这些源码信息的，那么如果我们就把二进制当做字符来看，是不是能搜到一些东西呢?

于是我们试着做了一个比较简单的二进制文件，二进制文件中仅仅包含一个 ZSCViewController，然后用 xxd 这个命令尝试读取二进制中的内容，考虑到 xxd 的输出会折行，我们选取了 ZSCViewController 字符串的子串进行过滤：

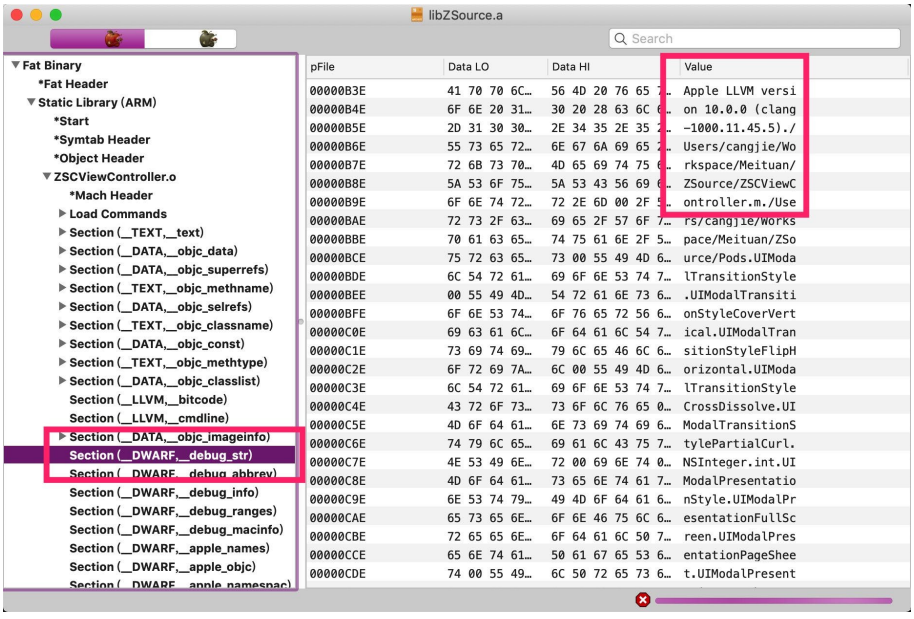
```
xxd ./libZSource.a | grep -C 5 'ZSCViewControlle'
```

果真得到了一些结果：

```
xxd ./libZSource.a | grep -C 5 'ZSCViewControlle'
00001dd0: 564d 2076 6572 7369 6f6e 2031 302e 302e  VM version 10.0.
00001de0: 3020 2863 6c61 6e67 2d31 3030 302e 3131  0 (clang-1000.11
00001df0: 2e34 352e 3529 002f 5573 6572 732f 6361  .45.5)./Users/ca
00001e00: 6e67 6a69 652f 576f 726b 7370 6163 652f  ngjie/Workspace/
00001e10: 4d65 6974 7561 6e2f 5a53 6f75 7263 652f  Meituan/ZSource/
00001e20: 5a53 4356 6965 7743 6f6e 7472 6f6c 6c65  ZSCViewControlle
00001e30: 722e 6d00 2f55 7365 7273 2f63 616e 676a  r.m./Users/cangj
00001e40: 6965 2f57 6f72 6b73 7061 6365 2f4d 6569  ie/Workspace/Mei
00001e50: 7475 616e 2f5a 536f 7572 6365 2f50 6f64  tuan/ZSource/Pod
00001e60: 7300 5549 4d6f 6461 6c54 7261 6e73 6974  s.UIModalTransit
00001e70: 696f 6e53 7479 6c65 0055 494d 6f64 616c  ionStyle.UIModal
```

xxd 命令的输出结果

通过这个实验，我们确定了二进制中源码文件的路径确实是用普通的字符来存储的；紧接着，我们用 MachOViewer 来查看二进制文件，以获取到更友好的二进制信息。利用 MachOViewer，我们可以发现这些信息都存在了二进制的“__debug_str”Section 中。



MacOViewer 的结果

虽然还是不确定这个地址所对应的字段叫什么，但研究到这里，我们还是有所进

展的，最起码我们可以假定这个路径一定是紧跟在“Apple LLVM version 10.0.0 ”字符后面的，然后利用一些读取 Mach-O 的 Ruby 库，比如 ruby-macho，基于这个假定来读取这个路径，为这个特性的工具化提供一丝可能性。

柳暗花明

简单的尝试没有得到想要的答案，但透过 Section 的名字，可以确定源码文件的路径信息和 DWARF 有关。

长时间和 CI 打交道的经验告诉我们，对于每一种二进制格式，苹果公司都会提供一个可以专门用于解析的命令行工具，所以我们就尝试找了找有没有解析二进制中 DWARF 格式的命令行工具。

功夫不负有心人，我们找到了 dwarfdump，那么用它来看看之前的那个二进制文件：

```
dwarfdump ./libZSource.a | grep 'ZSCViewContro'
```

果然有了更好的输出：

```
AT_name( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_name( "ZSCViewController" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.h" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
    AT_name( "-[ZSCViewController viewDidLoad]" )
    AT_decl_file( "/Users/cangjie/Workspace/Meituan/ZSource/ZSCViewController.m" )
        AT_type( {0x000011bd} ( const ZSCViewController* ) )
    AT_type( {0x000011c2} ( ZSCViewController* ) )
    AT_type( {0x000001e5} ( ZSCViewController ) )
```

dwarfdump 的输出

这里我们注意到了 AT_name 这个字段名。拿着这个字段名，去前面给出的 DWARF 1.1.0 Reference 文档中查阅，我们可以得知：

> An AT_name attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

进一步查询，我们可以找到另一个和他类似的字段 —— AT_comp_dir：

> An AT_comp_dir attribute whose value is a null-terminated string containing the current working directory of the compilation command that produced this compilation unit in whatever form makes sense Forelax the host system.

看起来，这两个字段就是我们所苦苦追寻的答案了。

工程化

通过实验，以及找到的这两个字段的描述，我们基本可以确定，即便工程是使用二进制构建，只要二进制 AT_name 字段中的路径存在对应的源码文件，App 一样可以使用源码进行断点调试。这种调试方式除了修改源码再次构建不能生效以外，其他的调试场景都和直接使用源码构建无异。考虑到我们日常的调试场景绝大多数都只需要查看其他组件的源码，并不需要修改，把这个功能工程化还是非常有意义的。

那接下来的事情就比较简单了：

1. 首先，我们需要确定大部分美团使用的组件二进制的编译目录是相同的。这样就方便我们在本地某个路径下统一管理下载的源码文件。
2. 接下来，我们通过 dwarfdump 这个命令获取源码文件应该在的路径，然后通过给 CocoaPods 增加命令，将源码文件下载并放置在对应的路径中。

幸运的是，查看完美团 App 的几百个组件后，我们发现只有少数近一年内没有制作过二进制的组件路径比较不同，其他都相同，因此可以先忽略这一小部分组件。如果这部分组件需要支持该功能，只要再制作一次二进制即可。

确定方案以后，写代码就很简单了，最终我们利用 CocoaPods，提供了 zsource 的三个命令：

```
pod zsource
Usage:

    $ pod zsource COMMAND

      [cocoapods-binary] 通过将二进制对应源码放置在临时目录中，让二进制出现断点时可以跳到对应的源码。 该命令想法由 Zangchengwei 以及
      wangZhiyu 提出，故名为 zsource

Commands:

    + add       [cocoapods-binary] 在不删除二进制的情况下为某个组件添加源码调试能力，多个组件名称用空格分隔
    + clean     [cocoapods-binary] 删除所有已经下载的源码
    + list      [cocoapods-binary] 展示所有已经下载的源码以及其大小

Options:

    --silent    Show nothing
    --verbose   Show more debugging information
    --no-ansi   Show output without ANSI codes
    --help      Show help banner of specified command
```

pod zsource 命令

总结

zsource 功能整体的开发过程基本上都是基于一个个的猜想和实验来完成的，整体的开发上线过程实际上只花了两个晚上。但如果在没有基础知识的情况下，选择把上文中提到的参考资料都看懂后再动手，可能会花费更多的时间。这一个有趣的验证过程也充分说明，有时候我们可以不拘泥于冗长的文档以及资料，通过类似逆向工程的方式，非常快速地拿到我们需要的答案。此时我们再回过头去看文档，可能会获得比直接看文档更好的效果。

最后，非常感谢成威老师和志宇同学对技术的崇高追求，即便在飞机上，也愿意拿出电脑验证自己的猜想，为 zsource 后续的工程化落地提供了更多的可能。

作者简介

宇杰，美团 iOS 工程师，2016 年加入美团，先后参与美团 App 持续集成平台建设、美团 App ReactNative 平台化等工作。目前在参与美团 App 工程效率提升和 Flutter 应用的相关工作。

客户端单周发版下的多分支自动化管理与实践

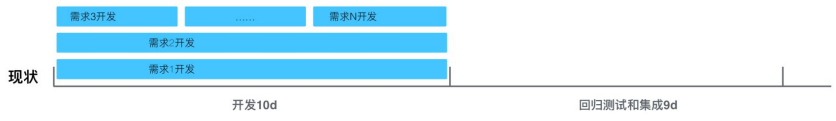
王坤

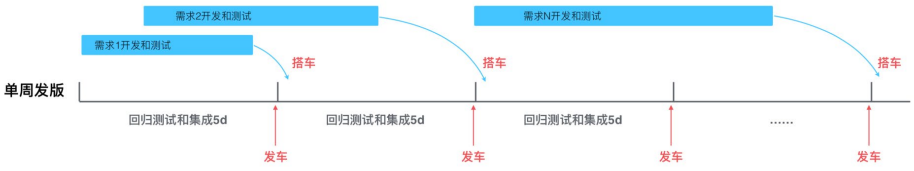
背景

目前，互联网产品呈现出高频优化迭代的趋势，需求方希望尽早地看到结果，并给予及时反馈，所以技术团队需要用“小步快跑”的姿势来做产品，尽早地交付新版本。基于以上背景，美团客户端研发平台适时地推行了单周发版的迭代策略。单周版本迭代的优点可以概括为三个方面：更快地验证产品创意是否符合预期，更灵活地上线节奏，更早地修复线上 Bug。

首先说一下美团平台的发版策略，主要变更点是由之前的每四周发一版改为每周都有发版。具体对比如下：

- （旧）三周迭代指的是 2 周开发 +1 周半测试，依赖固定的排期和测试时间，如果错过排期，则需要等待至少 20 天方可跟着下个版本迭代发布，线上验证产品效果的时间偏长。具体示例描述如下：





- （新）单周版本迭代指一周一发版，单周迭代版本排期、测试不再依赖固定时间节点，需求开发并测试完成，就可以搭乘最近一周的发版“小火车”，跟版发布直接上线。对于一般需求而言，这将会大大缩短迭代时间。

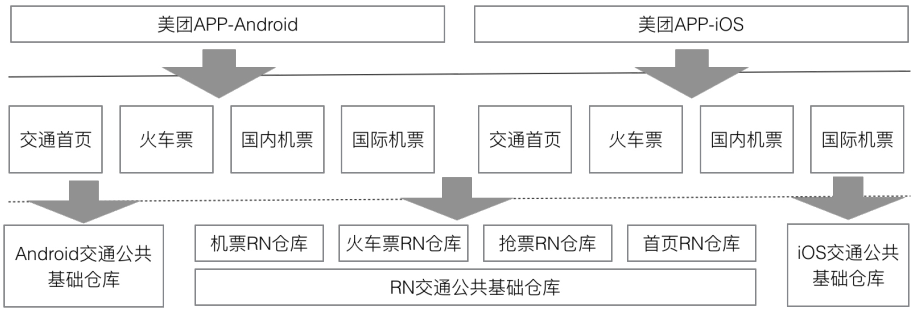
业务方研发人员的痛点

在之前按月发版的迭代节奏下，基本上所有的需求都属于串行开发，每个版本的开发流程比较固定。从“评审 – 开发 – 提测 – 灰度 – 上线”各个环节都处于一个固定的时间点来顺序执行，开发人力资源的协调方式也相对简单。全面推进单周发版之后，并不能把所有需求压缩到 5 天之内开发完成，而是会存在大量的并行开发的场景，之前的固定时间节点全部被打破，由固定周期变成了动态化调配，这给业务方的需求管理和研发人员人力管理都带来了指数式复杂度的提升。一旦进入并行开发，需求之间会产生冲突和依赖关系，版本代码也会随之产生冲突和依赖，这也大大提高了开发过程中的分支管理成本，如何规范化管理分支，降低分支冲突，把控代码质量，是本文接下来要讨论的重点。

下面描述了几种典型的单周发版带来的问题：

- **业务需求开发周期不固定，会存在大量的多版本、多需求并行开发。平台只提供了单周发版的基础策略，每 5 天发一版，业务方完成需求即可搭车发版。**
  对于各业务方来说，需求开发往往并不是都能在 5 天内完成，一般需求在 5~10 天左右，在之前串行发版模式下这个问题其实也存在，但并不突出，在单周发版的前提下，都要面临跨版本开发，意味着多个版本多个需求会同步并行开发，这给业务方的分支管理带来了极大的挑战。

- **业务方架构复杂，仓库依赖多，单周发版分支创建合并维护成本大。**
  交通业务线涉及火车票、国内机票、国际机票多条业务线，代码仓库除了业务线的独立仓库，还有交通首页，交通公共仓库，RN 仓库等多个仓库，Android 端 6 个 Git 仓库，iOS 端 5 个仓库，RN5 个仓库，共计 16 个 Git 仓库。
  多仓库频繁发版分支代码存在安全风险，容易漏合代码，冲掉线上代码。



交通业务线仓库结构示意图

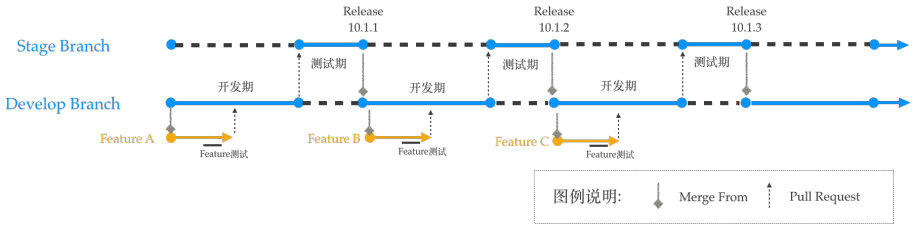
- **业务线自身的公共基础库需求变动频繁。也需要具备单周发版的能力。**

  例如交通公共基础仓库，承载了很多交通业务线的 UI 功能组件，这些公共组件的业务变化频繁，公共基础仓库变化的同时，可能会对使用组件的业务产生影响，需要同步的升级发版。美团平台的策略是公共服务组件每四个小版本统一升级一次，但对业务方自身组件这种策略限制较大，还是需要公共组件也要具备随时发版的能力。

单周发版分支管理解决方案

针对上面提出的问题，交通客户端团队通过技术培训、流程优化、关键点检测、自动化处理等方式保证分支代码的安全。技术培训主要是加强技术人员分支管理的基本知识，Git 的正确使用方法，这里不做过多描述。本文主要讨论关键点检测，以及如何进行自动化的分支管理。

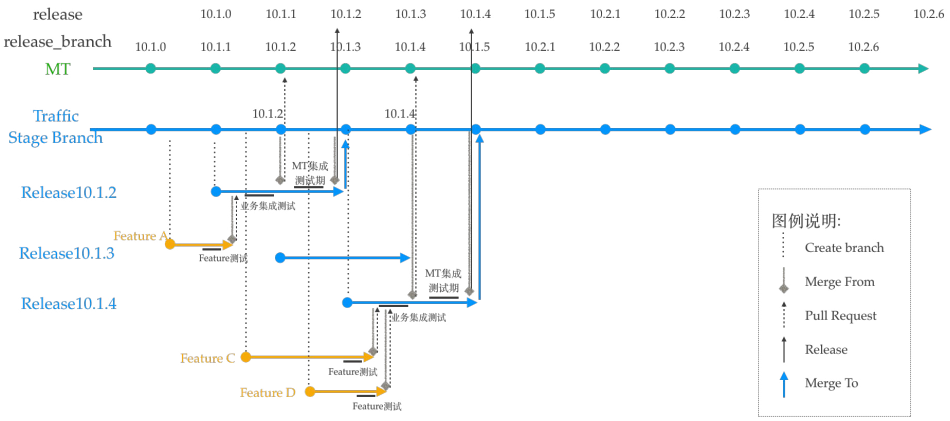
在实施单周发版之前，业务方代码仓库只有两个分支，Develop 分支，即开发分支；Stage 分支，即发版分支；开发流程基本在串行开发模式，每个版本 10 天开发，8 天测试，然后进入下一版本的开发。



之前的串行发版模式

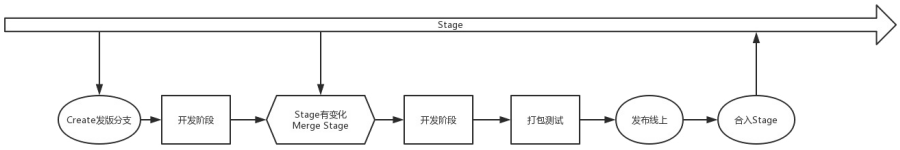
这种方式只能适用于节奏固定的长周期开发方式，对于多版本并行开发来说，有点力不从心，显然已经不能承载当前更灵活的发版节奏。

针对这些问题，我们推出了如下分支管理结构。总的来说，就是废除之前作为开发分支的 Develop 分支，建立对应的 Release 发版分支，每个版本打包从 Release 分支直接打包；同时 Stage 分支不再承担打包职责，而是作为一个主干分支实时同步所有已发布上线的功能，Stage 分支更像一个“母体”，孵化出 Release 分支和其它 Feature 分支；当 Release 分支测试通过、并且发版上线之后，再合入到 Stage 分支，此时所有正在开发中的其它分支都需要同步 Stage 分支的最新代码，保证下一个即将发布的版本的功能代码的完整性。



交通业务单周发版分支管理模型示意图

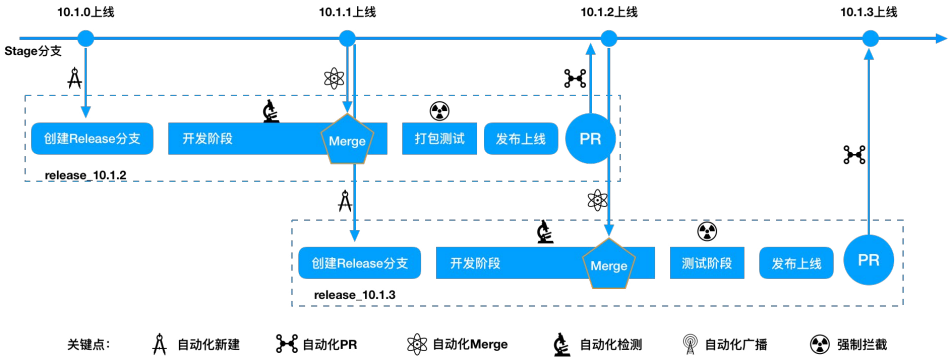
上面的流程描述可能有些复杂，下面是简化的流程图，每个版本都有自己的生命周期：



交通业务单周发版分支生命周期

1. 从 Stage 创建一个 Release 分支；
2. 进入开发阶段；
3. 如果 Stage 分支有变化，同步 Stage 分支；
4. 打包测试；
5. 测试通过，发布线上；
6. 发布线上之后，合回 Stage 分支。

为了适应单周发版，新的开发流程也引入了很多新的挑战。例如下图所示的一个 Branch 分支中涉及的六个关键点：创建分支、合入主干、主干变化通知、Merge 主干变化、检测主干同步、未同步拦截，除了这些还要考虑多仓库同步操作的问题，还有热修复版本的管理方式的问题。能否把这些关键节点合理的规范和把控起来，是我们当前应对多分支并行开发的主要难点：



交通业务单周发版关键节点

如何更高效的解决这些问题呢？结合我们当前使用的工具：Git + Atlassian Stash 代码仓库管理工具；Jenkins Build 打包工具；大象（美团内部通讯工具）内网通信工具。目前这三个开发工具已经非常成熟、稳定，并且接口丰富易于扩展，我们需要配合当前单周发版的分支管理模式，利用这些工具来进行扩展开发，正所谓“要站在巨人的肩膀上”。

1. 创建分支 Release 分支如何创建，何时创建，分支命名规范定义如何约束?

创建 Release 分支，本质上是从 Stage 新建一个分支，当前提前一个周期创建新的发版分支，例如在 10.1.1 版本 Release 后，创建 10.1.3 版本的分支，此时 10.1.2 版本处于开发测试阶段。业务方所有的分支命名和平台的分支命名保持一致，采用 Release/x.x.x 的格式，但同时需要升级成为即将发布的 Release 版本号，例如 10.1.3。

现在交通业务线多达十几个仓库，每个仓库每周都要操作一次需要耗费大量人力。之前每个分支的创建都是通过 Stash 或者手工创建，能不能自动化批处理的创建呢? 答案是肯定的。对此，我们采用了 Jenkins 的方式，需要建立一个 Jenkins Job, 基本原理就是通过命令行的方式进行 Branch 的创建，然后通过 Job 管理，批处理建立所有仓库的 Release 分支，这样就收敛了 Branch 的创建，即采用统一的命名规范，并且同时升级版本号。这就解决了创建分支的难点，实现了自动化创建分支，并且实现了规范化命名。

2. 如何知道 Stage 分支有变化，变化后需要做什么，不做会怎样?

一个好的开发习惯，就是每天写码之前都同步主分支，但是还是需要一个机制来确保同步。这里做了三个措施来确保各个分支和 Stage 是保持同步的：一个通知，一个警告，一次拦截。这三个步骤解决主干变化通知、检测主干同步、未同步拦截的问题。

一个通知：具体路径如下，建立了一个内部推送公众账号和一个 Jenkins 监听 Job，当所有交通业务仓库 Stage 分支有代码改动，通知所有对应的开发人员，该仓库有代码变化，请及时合入。

一次警告：本地开发过程中，每次提交代码到远端仓库时，会触发一个 Stage 分支代码同步检测的脚本，如果发现未同步，会通过内部通讯系统通知提交者存在未同步主分支问题。但这里目前并不做强制拦截，保证分支代码开发的整体流畅性。

最终拦截：在开发分支打包的过程中强制拦截，最终功能代码上线还是需要打包操作。在打包操作时统一收口，由于之前打包也是在 Jenkins 上来完成的，这里我们也是通过在打包 Jenkins 上接入了分支合并检测的插件，这样每次打包时会再次检测

和主分支的同步情况。如果发现未同步则打包失败，确保每次发版都包含当前线上已有代码的功能，防止新版本丢失功能。

3. 如何合并分支，如何保证漏合?

和上面提到的第一个如何创建分支的问题类似，通过 Jenkins Job 来进行批量操作，可以一键创建所有分支的 Pull Request；在每个版本发版之前，统一进行一次打包，合入美团的主分支，防止多个仓库有漏合的情况。

4. 公共基础库版本策略?

公共基础和业务分支保持同样的策略，通过批处理脚本同时建立分支，合并分支，监听分支变化，需要注意的是，每次版本升级，公共基础库也需要同步的打包，并且强制业务库升级。不然，如果基础仓库存在接口变动，有的业务升级了，有的业务没升级，最终会导致无法合入主分支，进而无法打出 App 包。

5. 热修复的版本管理策略?

热修复确实是一种非常规的处理方式。从原则上来讲，热修复需要在对应的 Release 分支上进行修改，然后把修改合入 Stage 分支，同时需要同步到其它正在开发的分支。实际的处理需要根据具体情况来分析，是否需要对线上多个版本热修复。如果多版本都要修复就不能再合入 Stage 分支，否则会导致 Stage 分支冲突，如果把 Stage 分支合入需要热修复的其它分支，会把线上当前最新代码带入历史旧版本，会导致版本兼容性问题。最终执行起来可能还是对热修复版本进行单独处理，不一定要进行 Stage 主分支的同步，热修复的版本管理成本会比较高，更多的需要人工介入。

未来展望

目前整体的分支发版流程已经基本完成，现在已经稳定运行了 10 个小版本，同时没有出现因为分支管理问题而引发的线上问题。

不过，当前整体流程的自动化程度还有待提高，每周需要人工去触发，很多代码合并过程中的冲突问题还需要人工去解决。未来我们希望能够自动化地根据平台的版本号自动创建分支，并且对于一些简单的冲突问题拥有自动化的处理能力。随着单周

发版的不断成熟，未来对于持续交付能力也将不断提升，发版节奏可以不限于单周，一周两版或是更快的发版节奏也成为一种新的可能。

作者简介

王坤，美团客户端开发工程师，2016 年加入美团，目前主要负责大交通业务的客户端架构、版本管理及相关工作。

美团外卖前端容器化演进实践

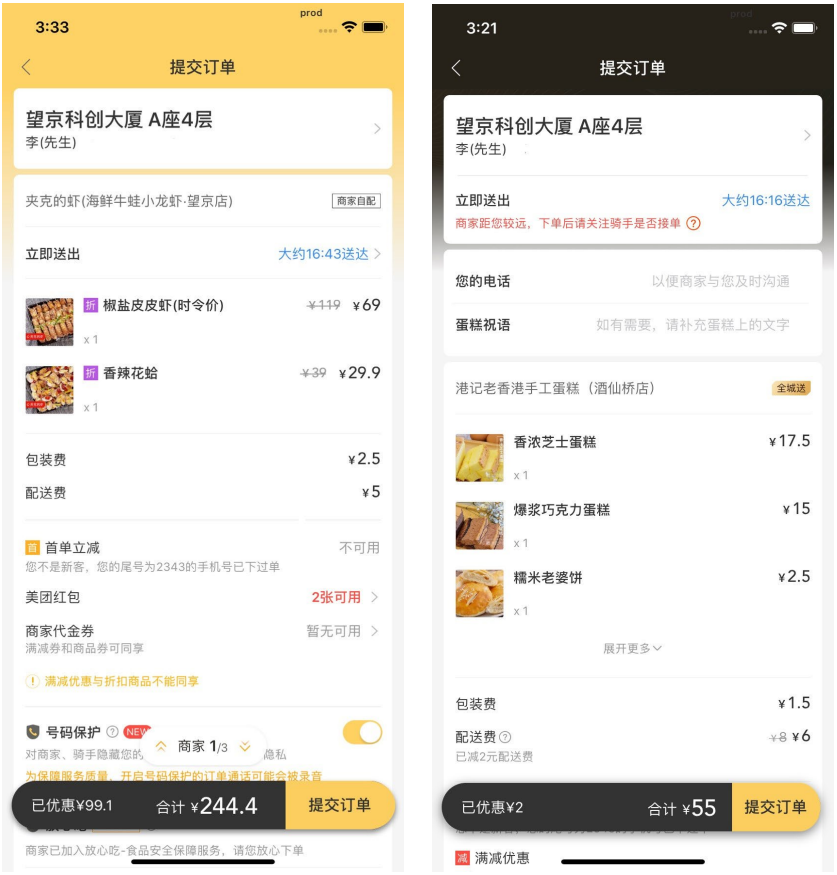
李肖　廷瑞　彦平　同同

背景

提单页的位置

提单页是美团外卖交易链路中非常关键的一个页面。外卖下单的所有入口，包括首页商家列表、订单列表页再来一单、二级频道页的今日推荐等，最终都会进入提单页，在确认各项信息之后，点击提交订单按钮，完成最终下单操作。





所支撑的业务

虽然提单页的代码统一放在外卖代码仓库中，但根据业务发展的需要，提单页上的模块分别由不同的业务部门去负责维护。主要包括以下业务方：

外卖侧业务

- 提单页绝大部分模块的需求开发和日常维护都是由外卖侧的研发同学在负责，包括地址模块、商家商品信息模块、折扣信息模块、准时宝、隐私号、发票备注等。

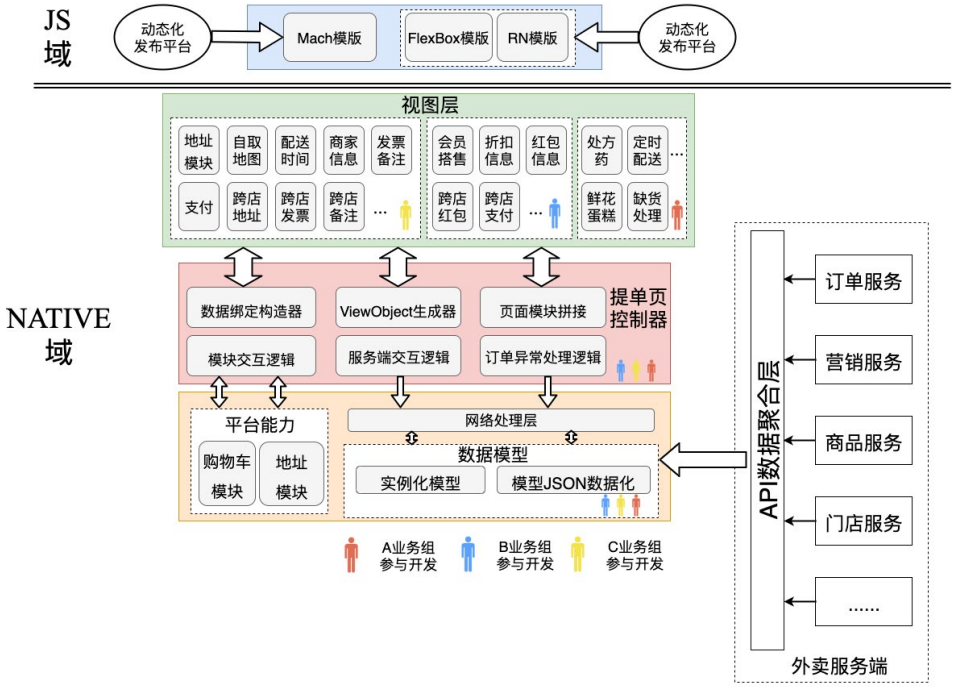
闪购侧业务

- 当从商超等频道进入提单页时，提单页生成的是闪购侧订单，闪购侧的订单在

配送方式、红包、下单路径上都与外卖订单有所区别，但又依赖于外卖的基础功能模块，因此与外卖侧功能存在严重的耦合问题。

其他业务

- 提单页上的部分模块对动态化配置能力有着很高的要求，这些模块使用 Mach 等动态化模版来实现相关的业务逻辑，由专门的业务组负责开发和维护。



容器化改造前提单页架构与开发人员分工

随着业务的不断迭代，提单页的模块也越来越多，逻辑的耦合也越来越重。现在提单页的 UI 展示模块已经超过 30 个，这些模块的展示与否基本上通过服务端的下发数据来决定。在不同的订单类型下，提单页所展示元素的差异越来越大，很多模块的代码已经不适合统一放在一起维护，代码拆分的需求十分强烈。此外，客户端包体积是衡量客户端性能的重要指标，为了解决业务发展带来的提单页代码量急剧增长的问题，同时实现页面元素的动态配置，我们希望能够实现提单页的动态化，而动态化需要基于容器来实现，所以我们提出了提单页的容器方案。

问题和挑战

提单页的容器化与外卖首页的动态化有以下几点不同：

1. 提单页整体动态化的需求不是很强烈，并且 API 改造的成本比较高，因此 API 接口字段保持不变，需要在客户端层面去做转换。
2. 首页模块基本仅作为展示用途，提单页模块的交互逻辑要复杂一些，比如发票模块，进入二级页面操作完成后还要更新提单页的数据。
3. 首页模块的 UI 展示各模块之间是完全独立的，而提单页的模块是根据功能聚合在一个组，这些模块条件出现的位置不同，展示的样式也不一致，如下图备注发票模块所示，最上层和最底层的模块上都带有圆角，所以提单页需要外层再添加一个模块组。



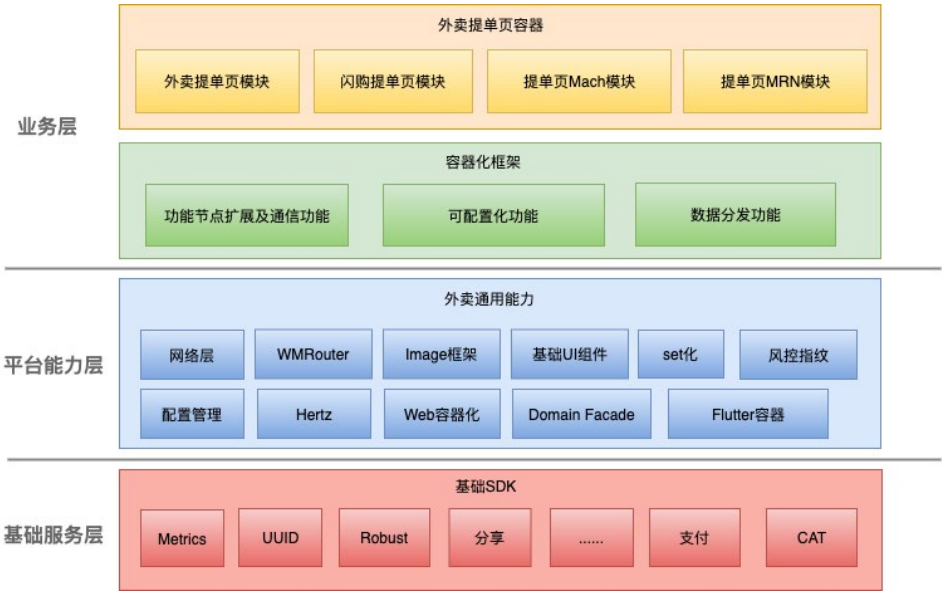
容器化后的提单页，需要实现模块之间的互相无感知，根据服务端的下发数据，客户端可以将闪购代码仓库内的模块和外卖代码仓库内的模块拼接起来组成完整的提单页展示给用户。当用户在提单页完成一系列操作时，各模块可以提供必要的参数给服务端。要想实现这一点，我们需要考虑以下几个问题：

- 模块注册问题，如何在无直接依赖的情况下，让提单页获取页面可用模块。
- API 数据分发问题，如何将服务端字段转换为模块可用数据，同时不侵入到模块这一层。
- 通信问题，模块之间如何实现联动效果。

- 页面更新和复用问题，在提单页刷新时如何提交数据给服务端以及如何完成模块的更新。

设计方案

1. 容器化整体的架构图设计

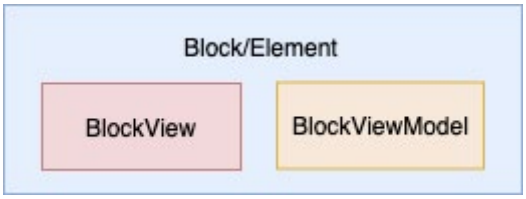


容器化是我们在外卖平台化之后对多方业务能力的支持和扩展，在不改变 API 数据源等前提下，我们保证其具有动态可配置化的能力。为了更好地支撑业务，我们在业务层面抽离出来容器化框架层，其所提供三个部分的核心功能：1. 功能节点扩展及通信功能；2. 可配置化功能；3. 数据分发功能。在最上层业务容器中，目前所支持外卖提单页面模块、闪购提单页模块、提单页 Mach（外卖动态化模板）模块、提单页 MRN（RN 页面）模块四种不同的业务。

1.1 概念解释

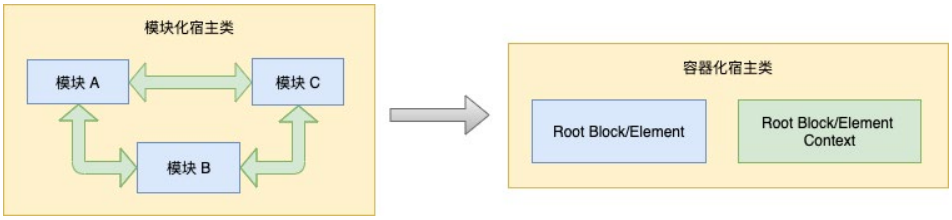
在容器化框架设计过程中，我们引入了一些新的定义，比如在 Android 端引入了 Block 的概念，这里的 Block 是一个功能模块的简称。在提单页页面中，我们可以理解为一个 Block 对应一个功能条目。在 iOS 端有与之对应的概念 Element（由于

两者没有差异，下文陈述中用 Block 代指两者）。



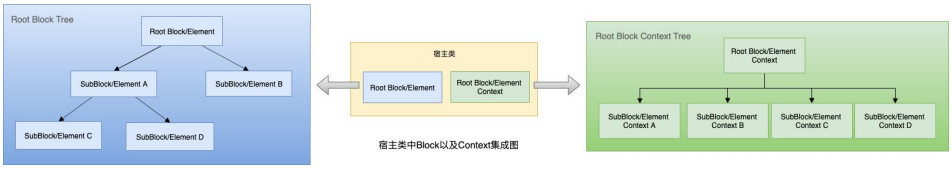
Block 有两种类型：其一是普通的 Block，其包含 BlockView（视图层）和 BlockViewModel（数据层）。BlockView（视图层）用来展示具体的视图以及内部的业务逻辑；BlockViewModel（数据层），用来数据解析。其二是 LogicBlock，是没有视图的 Block，单纯地用来做数据业务处理。

1.2 整体概述

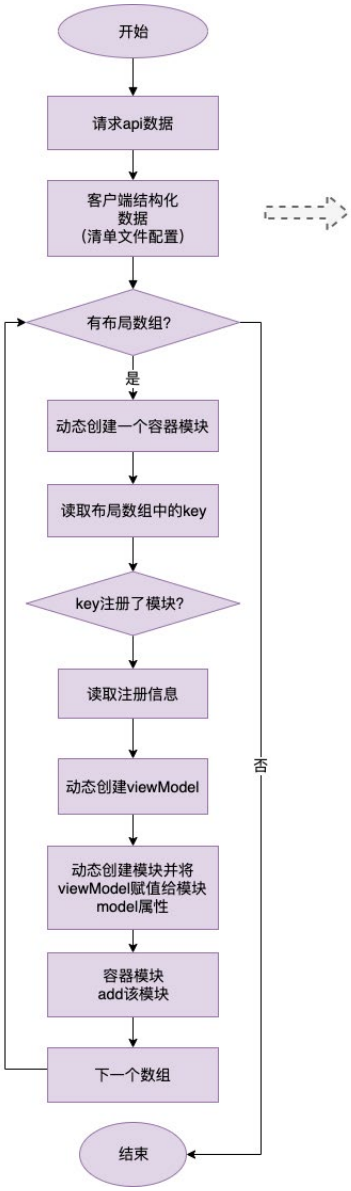


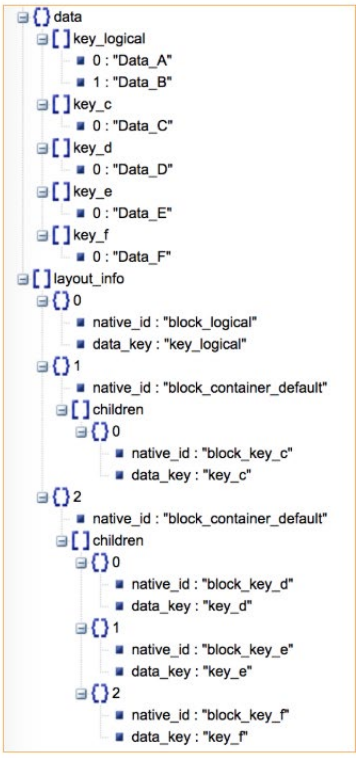
在容器化之前，我们的业务大多是模块化的结构，模块化宿主类是承载所有模块化的管理类，各个模块之间通过宿主类或者控制器进行数据交互。但在容器化改造中，我们将之前宿主类中管理的模块进行拆解，并重新定义了宿主类的职责。在容器化宿主类中，我们将不再持有各个功能模块的引用，而只要持有 Root Block 这一个实例，就可以完成对所有功能模块的管理。而 Root Block Context 则用来处理父 Block 与子 Block 之间的通信以及子 Block 之间的通信。

1.3 核心功能



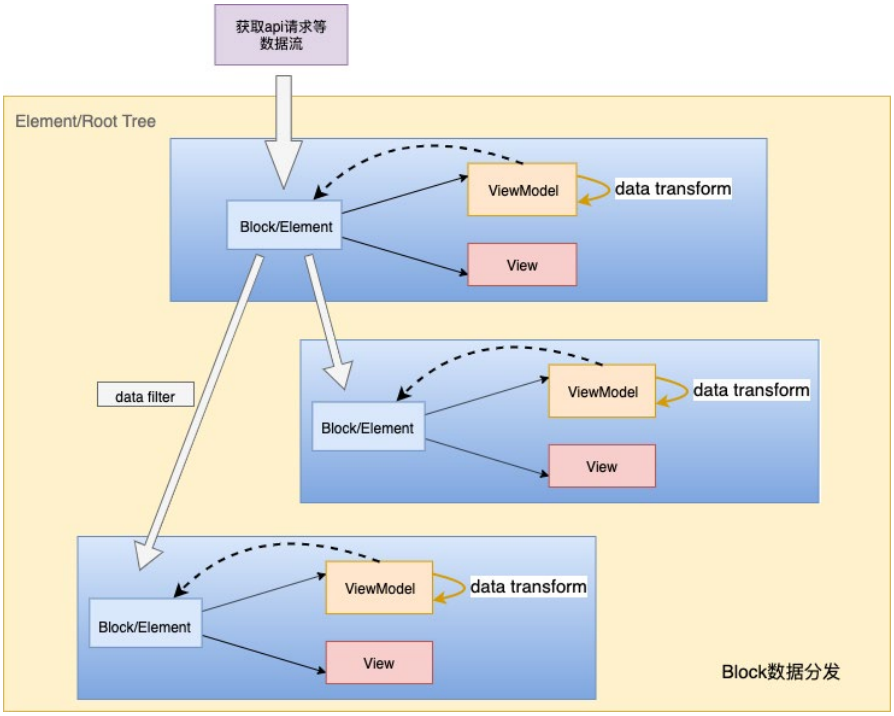
第一部分功能节点扩展及通信功能。主要是目前页面的集成和通信关系，其中 Root Block 是 Block Tree 的根节点，下面会挂载一些 SubBlock 子节点，Root Block 会控制整体的数据流的分发以及整体样式；Root Block Context 可以理解为上下文环境或通信的总线。每个模块都有自己的 Context，来维护自己向外部提供数据以及业务逻辑的能力，这些子 Context 会统一注册到 Root Context 中进行管理维护。





容器化创建Block流程图

第二部分可配置化功能。在发起数据请求成功之后，客户端根据注册的 Key 以及接收到的数据，动态创建 Block 的容器化能力。遍历解析数据以及配置文件，先动态创建 viewModel，将创建好的 viewModel 绑定到生成的 Block 模块上，动态添加到 Root Block 中。多业务方在完全不用相互感知的情况下，完成对新增模块的开发。



第三部分数据分发。既将解析之后的数据，由 Root Block 节点进行数据分发到各个子 Block，各子 Block 的 BlockViewModel 在更新数据之后并回传到 Block 中，Block 用更新后的数据更新 View 的展示。其中，数据可以自动完成分发，也可以手动的接管数据流进行相应的处理。

2. Block 注册问题

2.1 Android 注册的设计方案

Android 是在编译时期，通过 APT（注解处理器）的方式，将在指定模块上的注解信息和 Block 类关联起来，生成 Block 类对应的工厂类，然后将这些工厂类存在全局的 Map 集合中，并在运行时进行初始化操作。

```
@DynamicBinder(nativeId = "block_key_d", viewModel = blockDViewModel.
class, modelType = blockDInfo.class)
```

NativeID 是用来标识 Block 块的唯一 Key，viewModel 是用来绑定 View 视图的数据层，modelType 对应着 API 的数据 Model。

2.2 iOS 注册的设计方案

iOS 使用 Kylin 注册，Kylin 是美团平台开发的基建库，利用 Clang 提供的 section() 函数，在编译时 Kylin 将 {kylin_Key,kylin_Data} 格式的数据写入到可执行文件的特定数据段中，运行期就可以通过读取指定的 Key 值获取相应的数据。 使用这种方式，注册代码分散在每个组件内部。注册内容：组件 native_id、Element 名称、viewModel，其含义同上。

```
PGA_ELEMENT_REGISTER(order_action, OrderActionElement, OrderActionViewModel)
```

注册宏：

```
#define PGA_ELEMENT_REGISTER(NATIVE_ID, PGA_ELEMENT, PGA_VIEW_MODEL)  \
   KLN_STRING_EXPORT("AppKey_"#NATIVE_ID"", "{ \""#PGA_VIEW_MODEL"\"
: \""#PGA_ELEMENT"\"}");
```

3. API 数据结构化

由于 API 下发数据的不规范性，需要将数据按照 data_key 这种数据模式的方式进行整理，然后在获取数据之后，按照规则进行数据解析并创建相应的功能 Block。

目前 API 数据返回的格式：

```
{
  "data":{
        "xxx_pay_by_friend": true,
        "xxx_by_friend_tip": " 发给微信好友，让 TA 买单 ",
        "xxx_by_friend_bubble_desc": " 找微信好友买单 ",
        "xxx_friend_order": false
  }
  "code":0,
  "msg":""
}
```

由于这种格式是平铺分散的，没有将特定功能点的字段聚合在一起表示，不利于我们动态地将数据 Model 与 Block 绑定在一起。

需要我们将一个模块的数据统一在一个 JSON 对象中，整理之后 API 数据返回的格式如下：

```
{
  "data":{
       "pay_by_friend":{//key
          "xxx_pay_by_friend": true,
          "xxx_by_friend_tip": "发给微信朋友，让 TA 买单",
          "xxx_by_friend_bubble_desc": "找微信好友买单",
          "xxx_friend_order": false
       }
  }
  "code":0,
  "msg":""
}
```

将平铺的 API 数据整理成定制的结构化数据，将 Key 作为唯一的标识，那么就可以方便地用来对应指定模块化 Block 中所需的数据 Model。

布局及位置信息会对应相应的模块视图层，这由另外的 layoutInfo 字段给出。数组中的每条元素对应每一个 Block 模块 ，其中 native_id 的值是唯一的且与上面 Block 在注册时候的 Key 保持一致，data_key 的值映射上面整理之后的 API 数据的 Key，这样在编译时期生成 Block 的时候，就可以动态地关联相应的 ViewModel 以及数据模型。

```
{
  "layout_info":[
      {"native_id":"order_pay_by_friend","data_key":"pay_by_friend"},
      {
        "native_id":"block_container_default",//容器组
        "children":[
          {"native_id":"order_flower_cake","data_key":"flower_cake"}
        ]
      }
    ]
}
```

当然，这里可以以组为维度将一些功能相似的模块聚合在一起，native_id 的含

义同上，Children 是子 Block 结点的数组。

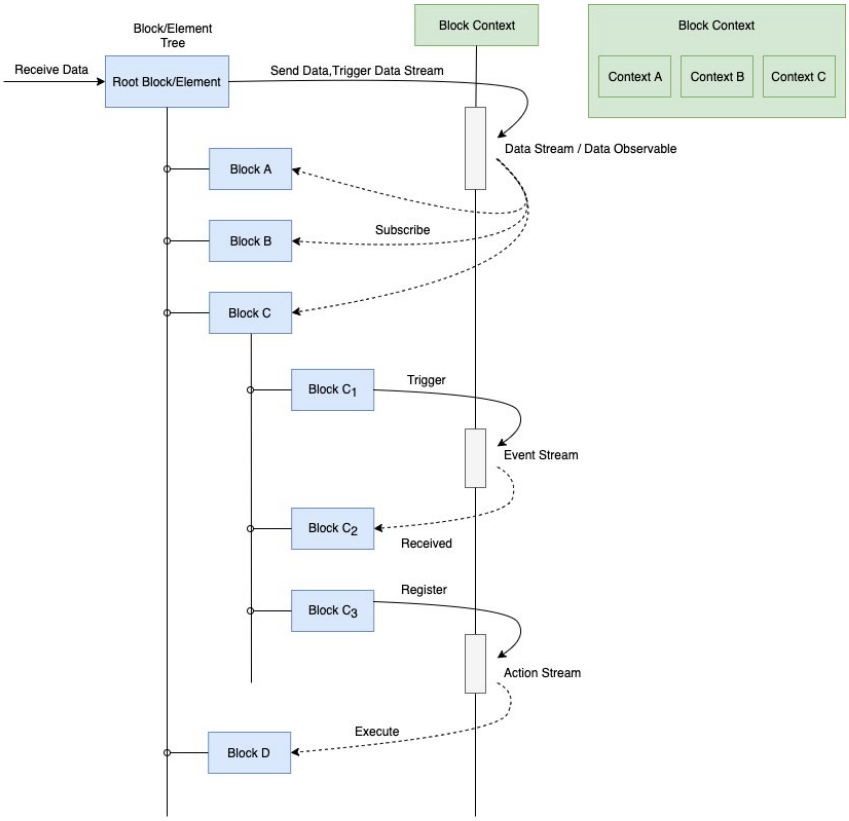
4. 模块间通信问题

由于之前模块化的时候，我们通过中间类的方式承载各个业务模块的通信逻辑。以 Android 为例，我们将多个子模块之间需要通信的逻辑，用接口的方式抛到 Activity 层，由 Activity 层进行业务逻辑的实现，但是由于子模块众多，最终导致该类的膨胀和模块的高耦合性，难以进行扩展和维护。

在容器化设计的时候，为了更好地使各个业务之间进行通信，降低耦合性，我们引入了 BlockContext，同上所述，理解为通信总线。

每个 Block 都有自己的 BlockContext，各个 BlockContext 汇总到 Root Block Context 中去实现，最终，各个 Block 就可以通过 BlockContext 进行数据传递。

整体的通信分发图如下：



图中展示的两种数据方式

4.1 Command 数据交互方式

将所需要的数据包装成事件，在指定的位置驱动事件的执行进而拿到需要的数据。

```
// 声明事件容器
private SupplierCommand<Object> mSupplierCommand = new SupplierCommand<>();
@Override
public SupplierCommand<Object> getSupplierCommand() {
    return mSupplierCommand;
}
// 注册实现
context().getSupplierCommand().registerCommand(new Supplier() {
   @Override
    public Object run() {
    }
});
// 获取相应的 Object 对象
context().getSupplierCommand().execute();
```

4.2 Event 数据交互方式

利用观察者的方式，订阅相应的事件，通过主动触发，从而完成数据分发等不同操作。

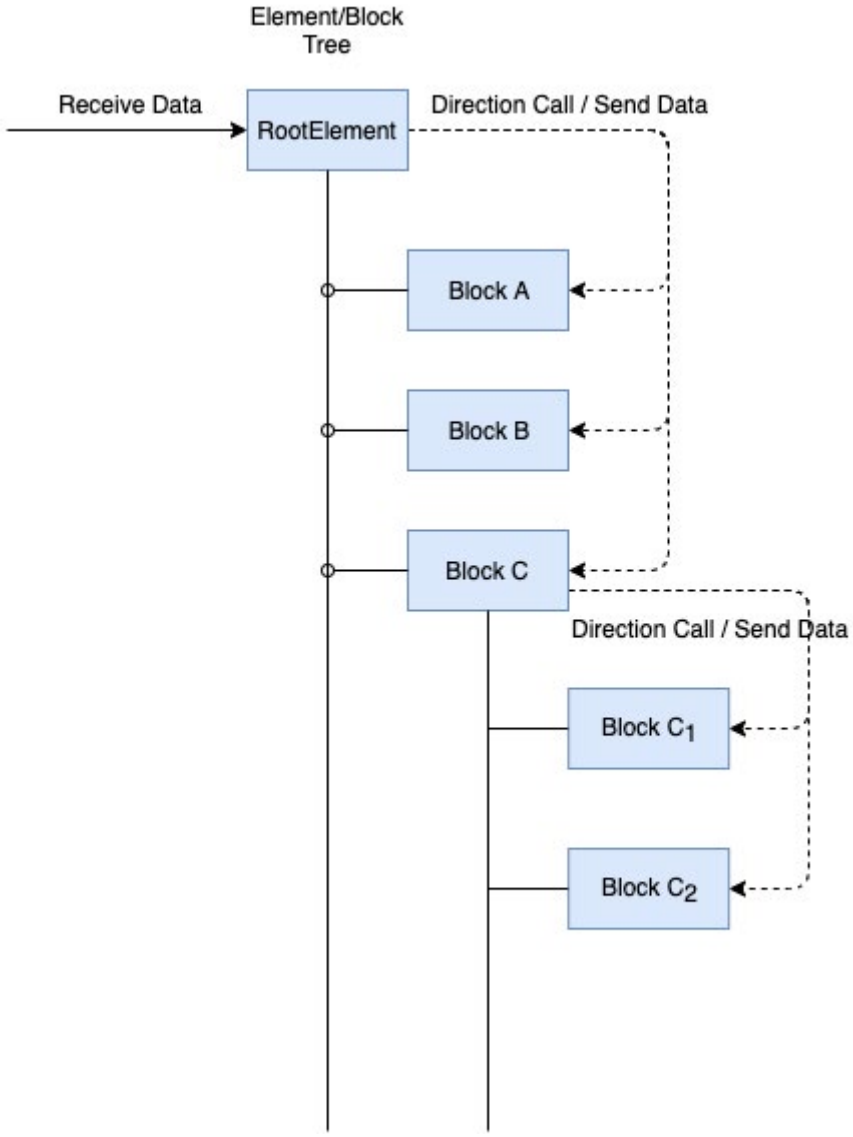
```
// 声明事件容器
private SupplierEvent mSupplierEvent = new SupplierEvent();
@Override
public SupplierEvent supplierResponseEvent() {
  return mSupplierEvent;
}
// 实现订阅
context().supplierResponseEvent().subscribe(new Action() {
   @Override
   public void action() {
   }
});
// 触发相应的操作
context().supplierResponseEvent().trigger();
```

5. Block 页面数据分发问题

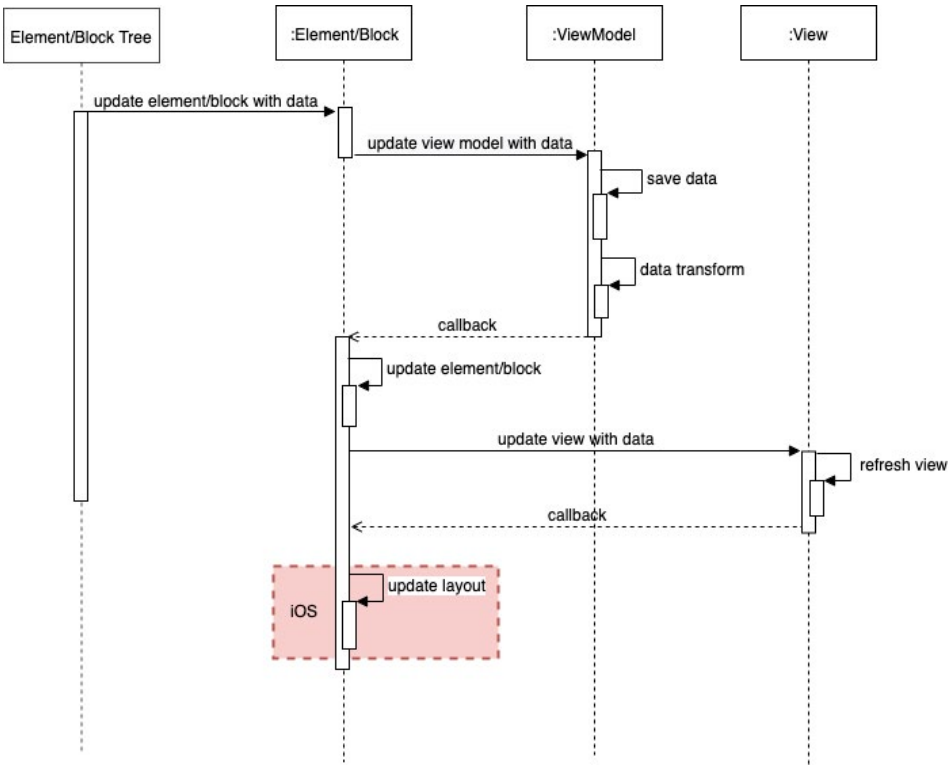
5.1 数据分发问题

Root Block 在接收数据的之后，会按照 Block 结点进行数据的分发。父 Block

将数据逐次的分发给子 Block。



Block Tree 数据分发逻辑简介图



Block 页面的刷新流程时序图

5.2 Block 创建的顺序

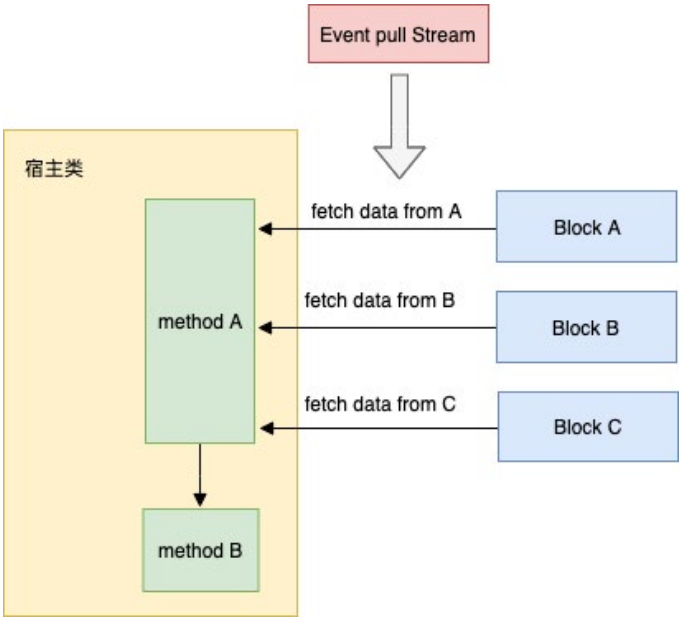
Block 创建的顺序由 API 结构化数据中的 layoutInfo 数组来决定，layoutInfo 数组的具体格式如第三节 API 数据结构化中内容所示。容器化后的提单页会根据 layoutInfo 数组的顺序，依次创建对应 native_id 的 Block 模块。因此，对于一些基础公共模块（比如 wm_confirm_order_logical 对应的 Block），我们可以将其放在 layoutInfo 数组的最前面让其提前加载，保证负责 UI 展示的 Block 创建时数据可用。

5.3 数据拉取问题

由于提单页的模块比较多，在页面曝光、页面刷新或提交请求时，需要从指定的模块获取相应的数据，作为请求的入参，那么如何做成在不感知其他业务方模块的情况下，完成数据的组装呢?

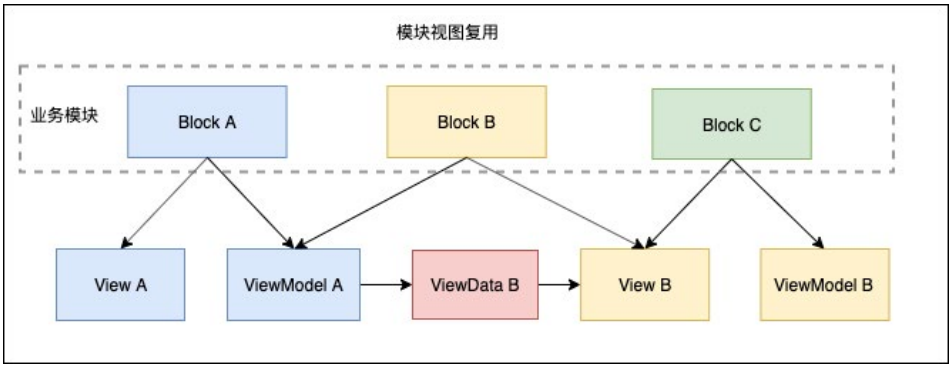
如上面的通信设计思路，我们利用 Event 数据交互方式，从各个模块中将需要

的数据取出来，完成数据的拼装。其中不同业务场景提取数据需要的校验工作，也分散在各个模块中进行处理。最终，即使在物理层面上隔离了对 Block 的感知，但是依然可以完成对请求所需数据的获取。



6. Block 页面的复用问题

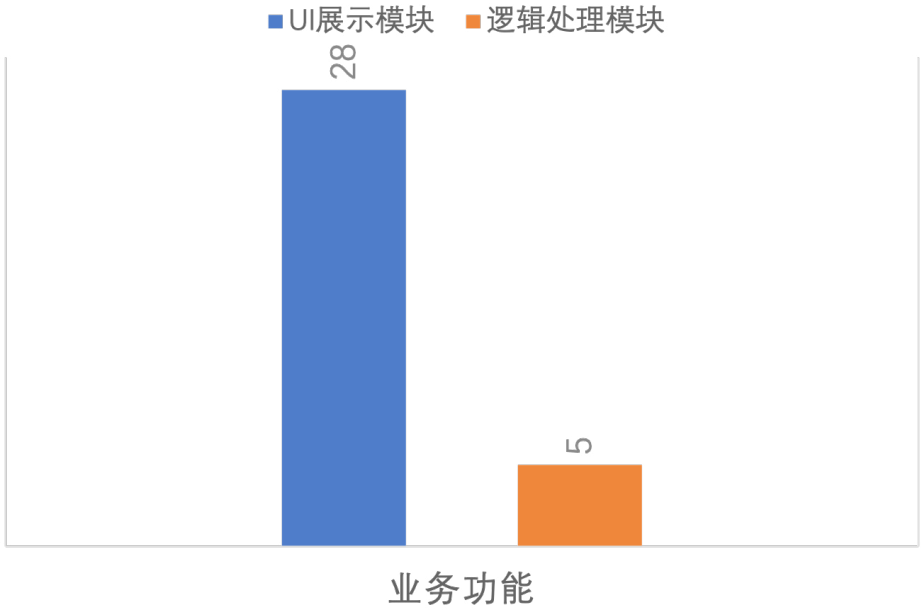
在实际的开发中，有些 Block 的页面 View 大致上相似，但是逻辑上有些细微的差异，为了快速开发，我们在设计上复用了其视图。Block、BlockView 以及 ViewModel 的关系：一个 Block 对应一个 ViewModel 和一个 BlockView，一个 ViewModel 和一个 BlockView 可以对应多个 Block。

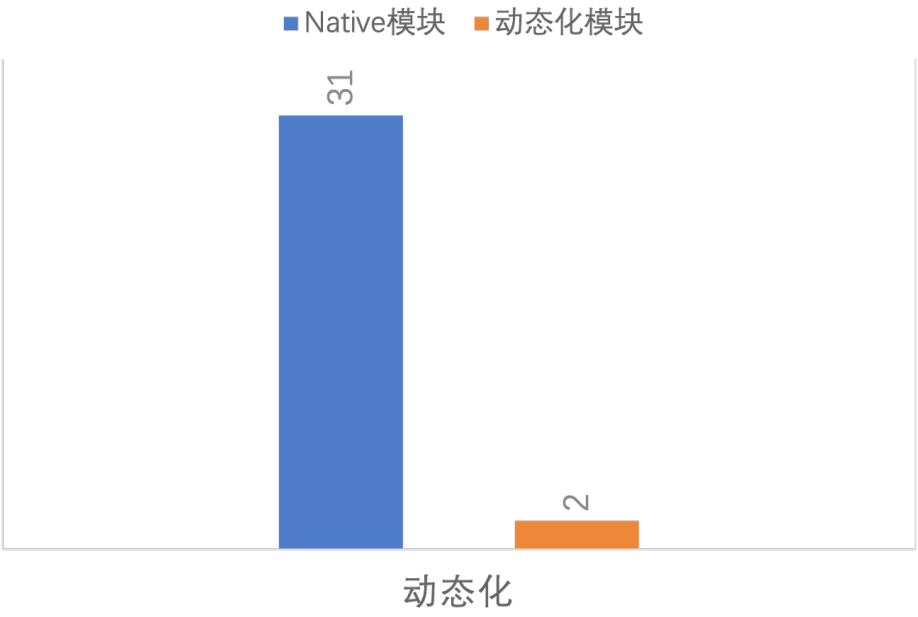


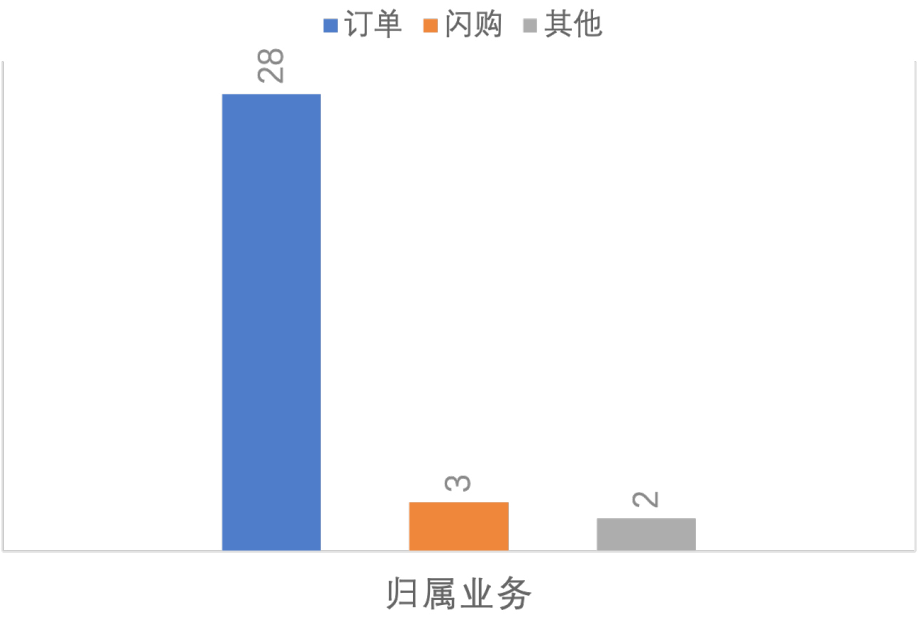
计算机界有一句名言："计算机科学领域的任何问题都可以通过增加一个中间层来解决。"（原始版本出自计算机科学家 David Wheeler）相似的，为了视图层的复用，屏蔽数据层的差异，我们在数据层的逻辑中转部分引入一个中间层 ViewData，ViewData 是为了更好地适配数据模型以及区别视图展示上的差异，这样就大大提高了代码的复用率。

收益

在开发过程中，我们将 iOS 和 Android 系统的模块进行了对齐和统一，容器化完成之后，两端同一 NativeID 对应的模块展示着相同的 UI 数据，也具有完全一样的业务逻辑。经过容器化后的提单页，相关代码被划分到了 33 个模块当中，这些模块分别承担着不同的职责。这里按照模块的业务功能、所采用的技术栈和所属业务线将这些容器化后的模块进行划分，得到如下的柱状直方图：







容器化之后的提单页完全由各模块组成，这些模块可以负责 UI 展示，也可以不展示任何 UI 模块，单纯地处理业务逻辑。模块内部的开发方案也可以根据业务形态

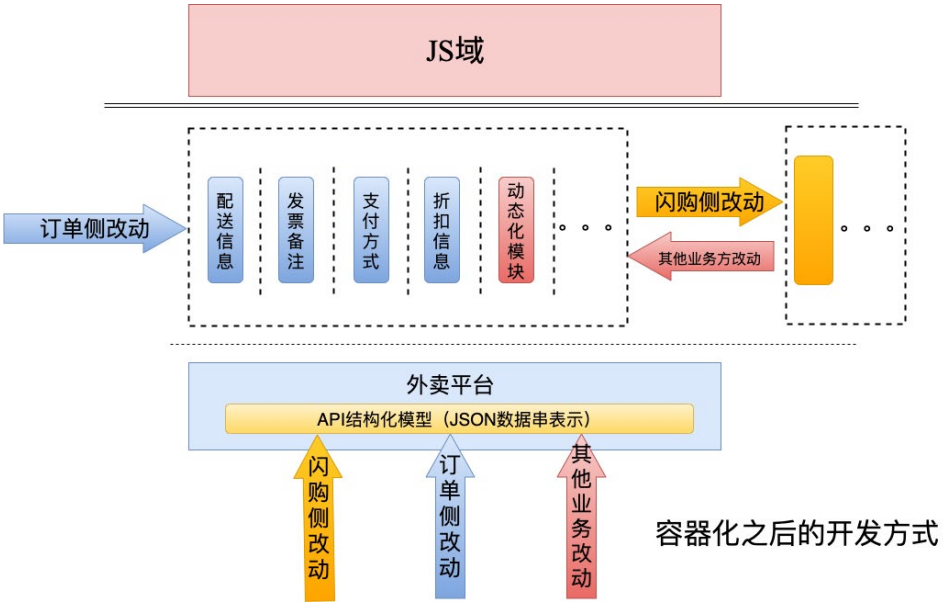
自由选择，相互之间做到了完全无感知。这些优点为后续提单页的业务迭代和技术优化都提供了很大的空间。

解耦的收益

开发效率提升

容器化之前的提单页，页面各部分共享同一个数据模型，服务端接口数据返回后，在提单页控制器内进行数据的更新、过滤和二次加工之后，再分发给页面上的各模块。当不同的 RD 同时开发提单页的需求时，这些放置在一起的业务逻辑会提高 RD 的开发成本，另外很容易出现代码层面的冲突，在需要 RD 手动解决的同时，也很容易因为开发流程的不规范出现 Bug。

容器化之后的提单页，开发模式也相应发生了改变，RD 在开发过程中，不会感知到别的模块内业务需求的改动，各业务可以在各自的容器内进行有效的推进迭代，而不用担心会影响到其他业务，从而让多人协作开发效率得到一定的提升。

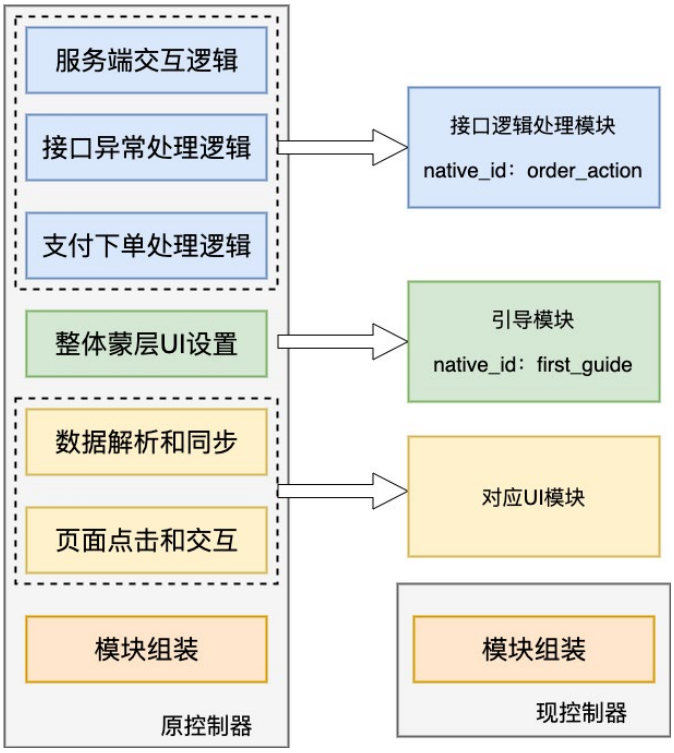


容器化之后的开发方式

控制器瘦身

在客户端业务开发的层面，MVC 架构得到了广泛应用。容器化重构之前的提单页，虽然也以模块化思想为基础做过多次重构，但是依然深受 MVC 思想的影响，在提单页控制器内保留了大量业务逻辑的代码。这些业务逻辑的代码最终会在业务迭代过程中逐渐变得臃肿和不可控，最终成为下一次代码重构计划中的业务背景。

本次提单页的容器化改造彻底解决了这一问题。基于 PGA 框架，包括接口异常处理、数据模型传递和二级页面跳转等业务逻辑代码都被收入到对应的 Element 和 Block 中，改造后的提单页中已经不存在业务逻辑相关的代码，彻底杜绝再次出现臃肿页面 VC 的可能。经统计，iOS 侧提单页控制器的代码行数从 2894 行减少到 289 行，控制器类中仅包含 Block 组装的业务逻辑。



包体积减少

提单页承载着美团的外卖业务和闪购业务，在未进行容器化之前，两个业务方需

要同时向订单库提交代码，在订单库整体“瘦身”的过程中，我们发现这种开发模式让包大小优化的工作多次出现反复，并且统计指标也难以统一和对齐。对提单页进行容器化改造之后，外卖和闪购分别维护各自模块内的代码，相互之间无依赖，闪购侧可以直接在自己的代码仓库内完成提单页模块的新增和修改，不需要再给外卖代码仓库提 PR，也就不会对外卖侧的包大小统计产生影响。

动态化的收益

动态化是整个外卖业务的发展方向。提单页的动态化建立在容器化的基础之上，在完成容器化之后，就具备了动态化的基础。当前提单页的动态化，所指的主要是模块层级的动态化，提单页的各模块展示顺序、展示与否，都可以完全由根据服务端下发的数据决定，各模块可以自由地进行组合、拼装，实现提单页的动态配置。

两端对齐的收益

之前因为历史原因，提单页很多的功能模块，Android 和 iOS 在实现上大相径庭，完全不一样的实现让两端在新业务需求到来时，在与服务端接口对接、开发工时和开发方案上都存在很大的差异，这些差异点对产品需求的排期、开发和测试上线上都产生了很多负面的影响。

提单页在容器化后的另外一项收益，就是 Android 和 iOS 在模块层级的代码实现，完成了统一。借助于 PGA 框架和 Element 注册机制，Android 和 iOS 具有大致相同的模块结构，相同 native_id 的模块获取的 API 接口返回字段完全一致；在页面请求接口数据时，相同 ID 的模块也提供同样的数据字段。在后续的开发过程中，两端对 API 接口字段的请求趋于一致，可以最大程度地减少因为两端不一致带来的合作方开发成本，也可以在一定程度上减轻下游的测试压力。

总结与展望

外卖客户端一直在推动核心页面的标准化，同时一直在探索尝试让核心页面也具备动态化能力。提单页作为下单路径上的核心页面，在 PGA 框架的基础上完成了容器化重构。至此，外卖首页、点菜页和提单页在页面这一层级都统一使用 PGA 框架

实现。统一化和标准化之后，可以让编程风格趋于一致，代码结构在不同平台保持统一，在后续的需求开发中，可以有效减少因为两端实现不一致出现的隐性开发成本。

提单页在容器化之后，让区域动态化的技术演进更容易推进。模块之间的解耦让不同模块可以自由选择模块内使用的技术栈而不会对其他模块产生影响。对于提单页的部分模块，完全可以通过 Mach 或者 RN 等动态化方案来实现，通过区域动态化来进一步减少开发成本，提高业务需求的开发效率。

在提单页之后，客户端会继续推进订单状态页使用 PGA 框架实现容器化，让标准化框架对用户下单路径上的核心页面实现 100% 覆盖。同时积极在提单页的商家商品信息展示、放心吃、准时保等模块探索页面的部分区域动态化，进一步缩减包大小，提高开发效率。

附录

1. Mach (马赫) 是外卖终端组自研的多终端跨平台级的局部动态化技术。
2. MRN 是美团基于 React-native 0.54.3 进行的二次封装，抹平了两端上的差异，并且提供了一些基础库和组件库供业务开发同学使用。
3. Metrics 是美团平台团队和外卖团队，开发的新一代 App 性能采集、监控、统计平台。
4. Hertz (赫兹) 是一个自动化的性能采集与监控 SDK，可以在开发、测试、灰度、运维各阶段，采集性能指标、检测卡顿、测量页面加载时间，帮助开发者监控和定位性能问题。

作者简介

李肖、廷瑞、彦平、同同均为美团外卖团队工程师。

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，Base 北京、上海、成都，欢迎有兴趣的同学投递简历到 tech@meituan.com。

Bifrost 微前端框架及其在美团闪购中的实践

雨甫

Bifrost（英 [ˈbi:frɔst]）原意彩虹桥，北欧神话中是连通天地的一条通道。而在漫威电影《雷神》中，Bifrost 是神域——阿斯加德（Asgard）的出入口，神域的人通过它自由穿梭于“九界”（指九个平行的宇宙）之间。借用“彩虹桥”的寓意，我们希望 Bifrost 可以成为前端不同 SPA（Single Page Application）系统之间的桥梁，使得不同的单页应用可以用这种方式实现功能的自由聚合 / 拆分。

项目背景

立项之初，闪购赋能企管平台（以下简称“企管平台”）仅仅是面向单个商家的 CRM 管理系统，采用常规的 Vue 单页应用方式来实现。随着项目的推进，它的定位逐渐发生了变化，从一个单一业务的载体逐渐变成了面向多种场景的商家管理平台。另一方面，由于系统由多个前端团队共同开发维护，越来越多的问题随之浮现：

- 异地协作时，信息同步不及时引起的代码冲突以及修改公共组件引入的 Bug。
- 不同的商家针对同一个页面存在定制化的需求。
- 已经实现的一些功能需要集成到企管平台中来。

因此，我们希望构建一个更高维度的解耦方案，使我们能够在开发阶段把互不干涉的模块拆成一个个类似后端微服务架构那样的子系统，各自迭代，在运行时集成为一个能够覆盖上述各种使用场景的完整系统。

方案选型

首先，我们整理了核心诉求，按优先级排序如下：

- 希望异地开发时不同的模块能够独立开发、独立部署。
- 对已在线上运行的项目，希望能够低成本地接入企管平台，而不需要对开发、部署流程做大规模的改动。

- 各个子系统独立运行，互不影响，但允许我们在开发阶段与其他子系统进行联调。
- 保持单页应用的体验。
- 由于现有项目都是基于 Vue 技术栈开发，因此，我们的框架并不需要做到技术栈无关，只要满足 Vue 的项目即可。

基于以上这些诉求，我们调研了目前市面上常用的微前端方案，最常见的方案有：

- 基于 Nginx 的路由分发。
- 使用 Iframe 将页面嵌入。

除此之外，还有美团集团内部的微前端实践——美团 HR 系统（用微前端方式搭建类单页应用）和业界比较知名的微前端框架——SingleSPA。

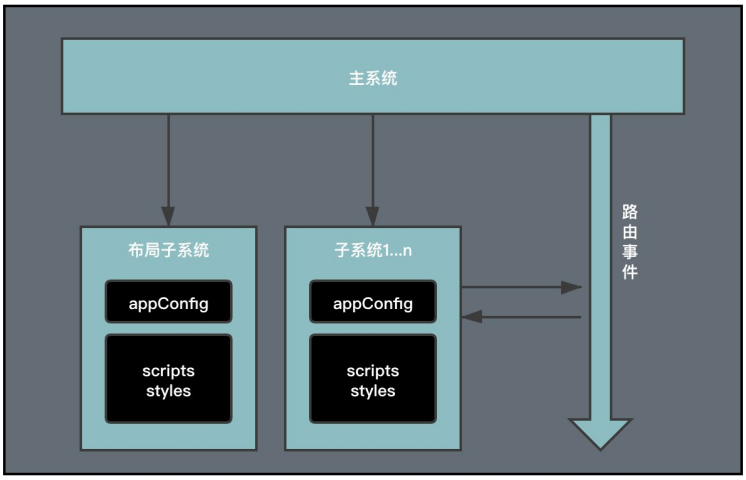
这些方案的优劣整理如下：

方案	实现	优势	缺点
基于Nginx的路由分发	根据路由前缀的不同将用户的请求指向不同的HTML入口文件。	实现简单，且各个子系统可以采用完全不同的技术栈进行开发。	页面跳转时，可能会闪动、白屏，无法做到单页应用的体验；新系统加入时，需要修改线上Nginx配置，风险较大。
Iframe嵌入	在现有系统中直接嵌入Iframe。	实现简单、子系统加载时依然保持单页应用体验。	页面刷新之后，无法保持子系统当前的路由状态；Iframe的适配存在一定问题。
HR系统方案	按模块对系统进行拆分。各个模块独立开发、构建、部署，在运行时，通过Webpack动态加载的方式进行集成。	能够保证单页应用的体验，且经过多次迭代，已经是一个经过生产验证的成熟解决方案。	本质上还是一个单页应用，各个子模块无法独立运行；同其他子系统进行联调的成本较高；新增子系统需要修改Nginx配置，存在一定风险。
SingleSPA	异步加载子系统的入口文件，并动态为子系统创建挂载点，Bifrost也参考了它的实现方式。	目前唯一落地的真正的微前端框架，可以做到技术栈无关。	子系统之间难以联调；需要修改目前的部署流程。

从用户体验角度出发，Nginx 和 Iframe 首先被否决；HR 系统的方案需要对现有的项目进行改造，把不同团队目前开发的项目整合到同一个单页应用中，在项目快速迭代的过程中，成本过高，所以也被否掉。SingleSPA 看起来完美，但它没有照顾到实际生产环境中的开发、部署的差异性，并不是 Product-Ready。综合多种因素考虑，我们最终决定采用自研的方式来实现微前端化 Bifrost。

核心架构

Bifrost 框架在设计的时候参考了 SingleSPA 的思路，将系统了分为主系统和子系统。

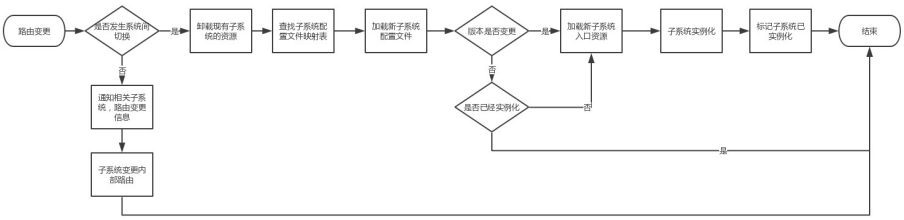


主系统是用来控制子系统的调度中心，职责包括：

- 维护子系统的注册表。
- 管理各个子系统的生命周期。
- 传递路由信息。
- 加载子项目的入口资源。
- 为子系统的实例提供挂载点。

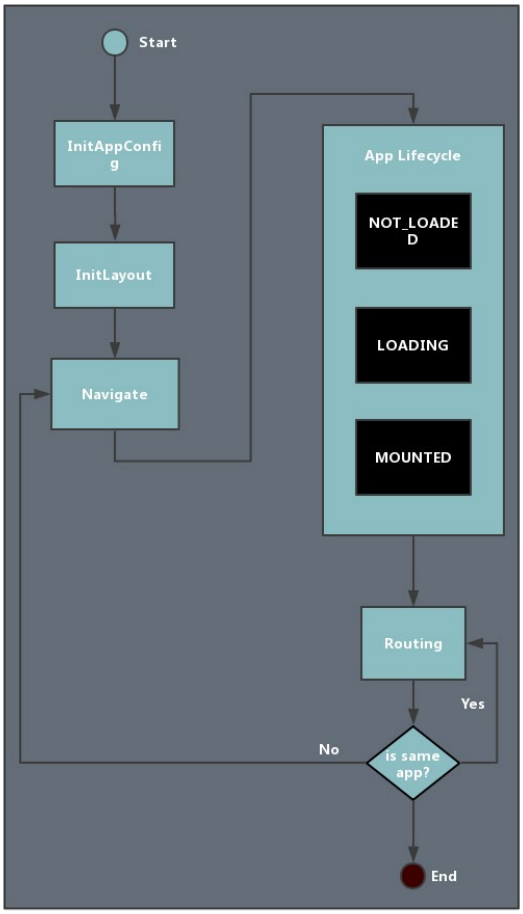
子系统只负责业务逻辑的实现。如果进一步细分的话，子系统可以分为业务子系统、实现公共菜单子系统、导航布局子系统，其中布局子系统会先于业务子系统加载。

Bifrost 采用路由消息分发的方式来控制子系统的加载和跳转。主系统维护了一条路由消息总线，当路由发生变化时，子系统会将路由事件推送给路由总线，然后由路由总线决定加载 / 跳转的目标子系统。如果路由不需要切换子系统，则交由当前子系统进行处理。



如果子系统发生切换，主系统会在 DOM 中添加对应子系统的挂载节点，并异步加载系统的静态资源。由于子系统都是完整的 Vue 实例，当子系统的代码加载并执行之后，子系统就会自动在其对应的挂载节点上渲染相应的内容。

整个系统的生命周期如下图所示：



具体实现

基于 Bifrost 实现的项目架构如下图所示：



这里，我们主要关注主系统、业务子系统和布局子系统的实现。

主系统

主系统的逻辑比较简单，主要是实例化 Bifrost 中定义的 Platform 对象，并注册各个子系统。子系统的注册信息包括：

- AppName：子系统名，与系统的路由前缀保持对应，同时也会作为子系统在 DOM 中挂载节点的 ID。

- Domain：非必填，如果出现多个路由前缀都对应同一个子系统，可以通过 Domain 进行映射。
- ConfigPath：对应子系统配置文件的 URL。

一个简单的主系统实现如下：

```
import { Platform } from '@sfe/bifrost'

new Platform({
  layoutFrame: {
    render () {
      // render layout
    }
  },
  appRegister: [
    { appName: 'app1', configPath: '/path/to/app1/config.js' }
  ]
}).start()
```

业务子系统

在设计方案时，我们始终保持一个理念，就是保证对业务代码的零侵入，因此业务系统改造的工作量很小。代码层面，只需要把原本子系统的初始化流程放到 AppContainer 对象的 Mounted 回调函数里即可：

```
import { AppContainer } from '@sfe/bifrost'
import Vue from 'vue'
import VueRouter from 'vue-router'

Vue.use(VueRouter)
const router = new VueRouter({})
new AppContainer({
  appName: 'app1',
  router,
  mounted () {
    return new Vue({
      router,
      components: { App },
      template: '<App/>'
    }).$mount()
  }
}).start()
```

另外，还需要修改子系统的构建流程，构建完成之后生成一个包含子系统入口资源信息的配置文件。一个典型的配置文件如下：

```
((callback) => callback({
  scripts: [
    '/js/chunk-vendors.dee65310.js','/js/home.b822227c.js'
  ],
  styles: [
    '/css/chunk-vendors.e7f4dbac.css','/css/home.285dac42.css'
  ]
}))(configLoadedCb.crm)
```

- 此处我们实现了 @sfe/bifrost-config-plugin 插件，在 Webpack 构建脚本中引入该插件就可以自动生成项目对应的配置文件。配置文件是一个立即执行函数，主系统可以通过 JSONP 的方式读取配置文件中的内容。

在实际生产环境中，我们可以将子系统发布到任意 CDN，只要能够保证配置文件的 URL 始终不变，那么无需依赖任何服务，主系统就可以感知到子系统的发布。

布局子系统

布局子系统是用来实现菜单和导航栏的 Vue 工程，本质上和一般的业务子系统没有区别。只需要注意，布局子系统使用的是 LayoutContainer 而非 AppContainer 进行包装。

```
import { LayoutContainer } from '@sfe/bifrost'
import Vue from 'vue'

import App from './app'

new LayoutContainer({
  appName: 'layout',
  router,
  onInit ({ appSlot, callback }) {
    Vue.config.productionTip = false
    const app = new Vue({
      el: '#app',
      router,
      store,
      render: (h) => (
        <App appSlot={appSlot} />
```

```
        )
      })
      callback()
    }
  }).start()
```

布局子系统作为主系统的一部分，既可以放在主系统中去实现，也可以像其他子系统一样通过异步的方式去加载。在我们的项目中，结合了上面两种方式（布局子系统既可以为作为常规的 Vue 项目构建，也可以发布成 NPM 包），每次发布时，会同时发布布局的静态资源和 NPM 包。主系统通过 NPM 包的方式引入布局子系统，将它打包到项目中，避免线上运行时，额外加载布局子系统的资源，减小项目体积，加快渲染速度。

本地开发时，我们则会通过 Bifrost 定义的 MockPlatform 异步加载布局子系统的静态资源，保证线上 / 线下运行效果的一致性，方便本地联调。

工程实践

代码层面的改动虽然不多，但要在实际的生产环境中落地，还需要解决一系列老生常谈的问题，包括：

- 本地开发时，如何保证与线上实际运行效果的一致性?
- 如何实现全局状态管理和子系统之间的通信?
- 如何对公共依赖和公共模块进行管理?
- 发布部署流程需要怎样调整?

根据闪购业务实践，我们总结了一套适用于 Bifrost 的解决方案。

本地联调

采用微前端的方式意味着子系统的完全隔离，这给我们的开发带来了一系列困扰：

- 本地开发时，无法看到当前开发的功能在主系统中实际运行的效果。
- 子系统之间有时会存在跳转关系，在开发阶段难以验证这种跳转逻辑的正确性。

为了解决这些问题，Bifrost 定义了 MockPlatform。MockPlatform 的思路很简单：既然主系统可以动态加载线上的子系统，那么我们只需要在开发时，模拟主系统的运行方式，去加载其他子系统的线上资源，之后就可以像调用后端 API 一样同各个子系统进行联调了。这也就解释了为什么布局子系统在输出 NPM 包的同时还维护了一份静态资源。

MockPlatform 的 API 同 Platform 对象的 API 是一致的，开发时，我们只需要按照主系统的方式引用布局或业务子系统的配置文件 URL 即可：

```
// ...others...

new AppContainer({
  // ...others...
  runDevPlatform: process.env.NODE_ENV === 'development', // 只在开发环境
下启动 mock platform
  devPlatformConfig: {
    layoutFrame: {
      mode: 'remote',
      configPath: 'path/to/layout/config.js'
    },
    appRegister: [{
      appName: 'app2',
      mode: 'remote',
      configPath: 'path/to/app2/config.js'
    }]
  },
  // ...others...
}).start()
```

借助 MockPlatform，我们项目在开发阶段的感受和开发普通的单页应用没有任何差异，如果某个我们依赖的子系统更新了功能，只需要让对应的 RD 发布一下，就可以在本地看到它的最新效果。

全局状态

除了本地联调，全局通信也是微前端项目中绕不开的一个话题。由于我们所有项目采用的都是 Vue 技术栈，所以会选择基于 Vuex 来实现全局通信。Bifrost 的主系统会维护一个全局的 Vuex Store，用于保存全局状态。

当子系统希望监听全局状态时，子系统并不是直接订阅全局 Store（Vue 的依赖收集机制也决定了子系统无法响应全局 Store 的变化），而是借助 Bifrost 提供的 syncGlobalStore 函数来订阅全局 Store。调用该函数后，任何全局状态的变化都会被同步到本地 Store 的 Global 命名空间下。之后，就可以像普通的单页应用那样，调用 Vuex 的 mapState 方法实现和全局状态的双向绑定。

```
import { AppContainer, syncGlobalStore } from '@sfe/bifrost'
import Vue from 'vue'
import Vuex from 'vuex'
// ...others...
Vue.use(Vuex)
const store = new Vuex.Store({})
new AppContainer({
  mounted () {
    // 同步全局 store 状态
    syncGlobalStore(store)

    // ...others...
    return new Vue({
      store,
      router,
      components: { App },
      template: '<App/>'
    }).$mount()
  }
}).start()
```

如果子系统自身的状态需要共享，Bifrost 还会提供 installGlobalModule 函数。该函数会将当前子系统需要共享的状态挂载到全局 Store 下，其他子系统可以通过前面提到的方式来同步这些状态。虽然 Bifrost 提供了子系统通信的能力，但在实际拆分子系统时，应该尽量避免这种情况发生。如果两个子系统之间需要频繁通信，那就应该考虑把他们划分到同一个子系统。

公共依赖

由于各个子系统都需要集成到企管平台，为了保证体验的一致性，大家都是基于同样的组件库进行开发。几乎所有项目都会依赖 lodash、Moment 等基础库，因此如果不对公共依赖进行管理，项目会加载大量冗余代码。

针对这个问题，我们采用的是 Webpack External 方式来解决。构建时，各个子系统会将公共依赖排除，主系统会打包一份包含所有这些公共依赖的 DLL 文件。子系统在运行时，直接从全局引用对应的依赖。如果子系统希望使用某些库的特定版本，也可以选择不排除这些依赖项。这在子系统希望升级某些依赖库的时候显得极为有用：通过子系统的局部升级，可以限制依赖库升级的影响范围，避免造成全局影响。

DLL 文件会包含大部分公共依赖，但有一个例外——我们不会将 Vue 打到 DLL 文件中。因为在实际开发中，很多库都喜欢向 Vue 的原型链上挂载方法和属性。如果不同团队开发时挂载的内容恰好用到同一个字段，就会带来不可预知的影响。

模块复用

除了底层的依赖，我们还需要考虑对公共的业务模块和工具函数进行复用。在企管平台，我们为公共业务组件库和公共函数库创建独立的 Git 工程，然后将所有的子系统和公共模块通过 Git Submodule 的方式引入到主系统的工程中。主系统采用 Lerna 的方式组织代码，各个子系统在开发时，可以通过软链直接引用到本地公共模块的代码，实现公共模块的复用。当公共模块发生更新，直接调用 Lerna Publish 就可以同时更新所有子系统 package.json 中依赖版本。

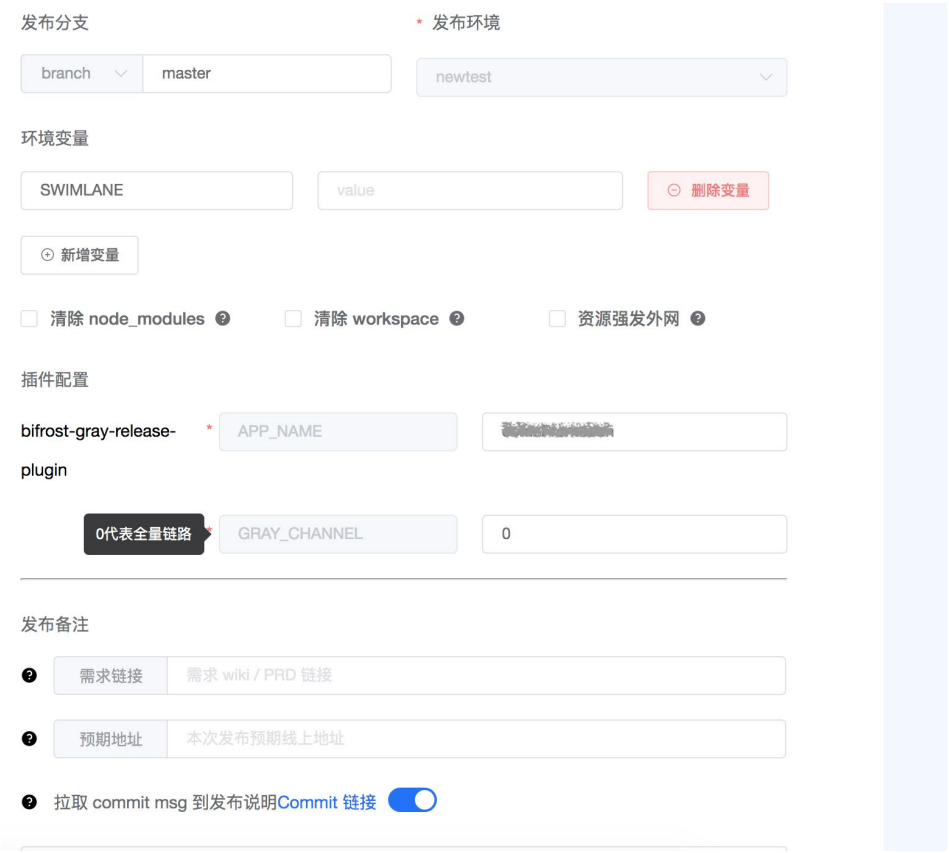
发布及部署流程

前面提到，主系统采用的是 JSONP 方式加载子系统的配置文件，整个发布过程都只需要发布静态资源，因此，Talos（美团内部自研的持续集成平台）提供的前端静态资源发布的能力就可以满足我们的需求。每次发布时，只需要构建有更新的项目，并将打包后的静态资源上传到 CDN 即可。

版本控制

采用微前端架构还有一个额外的好处：在 Nginx 和实际的业务层之间，多了一层主系统，我们可以像客户端一样，动态决定需要加载的子系统版本。基于此，我们实现了子系统的版本控制和定向灰度功能。发布时，我们通过参数确定本次发布是否是

灰度版本。在发布成功后，会记录本次发布的灰度信息、版本和配置文件 URL 等信息。



主系统每次启动时，首先会调用接口确定当前用户所处的链路（全量 / 灰度），再根据链路信息加载相应的子系统。我们记录了每次发布的资源 URL，所以也支持子系统的版本切换。只需要在版本服务中修改各条链路上需要激活的子系统版本，就可以轻松实现子系统版本切换。



埋点及错误上报

这里我们主要讨论 Bifrost 框架的埋点方案。在 Bifrost 项目中，可以借助主系统提供的一系列钩子函数实现针对子系统的埋点，包括：onAppLoading、onAppLoaded、onAppRouting、onError。每当子系统发生切换都会调用 onAppRouting 函数，因此我们可以在这里记录子系统加载的次数（PV）。onAppLoading 和 onAppLoaded 则会在子系统初次加载时调用，通过计算 Loading 和 Loaded 成功率的比值，我们可以得到子系统加载的成功率。子系统加载失败时，会调用 onError 函数，帮助排查子系统加载失败的原因。

收益

今年年初，我们对企管平台进行了微前端改造，目前系统已经在线上平稳运行半年时间，支持上百个零售商品牌，上千家门店业务的运转。

采用微前端架构，给我们项目带来的好处是显而易见的：

- 实现了异地合作开发时的完全解藕。采用微前端架构之后，两地团队在开发过程中再也没有遇到代码冲突的问题。
- 避免了单页应用发展成“巨石”应用。目前，企管平台总共实现了上百个页面，采用微前端的方式进行划分后，每个子系统包含的页面都不超过三十个，子系统的可维护性得到大大提高。
- 今年企管平台经历了两次大的组件库版本升级。第一次升级时，项目还是单页

应用，我们在暂停业务开发的基础上，耗费了大约一周的时间对所有的页面进行回归验证、完成升级。第二次升级时，我们已经完成了项目的微前端改造，可以通过增量的方式，先升级不常用的子系统，验证通过后再升级其他子系统。这样既不用中断正常的业务开发，也保证了依赖库升级时的影响范围和风险可控。

不是“银弹”

当然，同所有的架构方案一样，微前端这种模式也存在一些折衷和妥协。在获得低耦合和灵活性的同时，也引入了额外的复杂度。在微前端项目中，我们需要考虑多个工程的规范和代码质量的统一，需要引入更多的自动化工具来管理项目的发布部署流程，还需要处理多个前端工程运行在同一个域名下引起的 Cookie 覆盖等问题。

因此，在采用微前端架构之前，建议大家要谨慎的评估自己的项目是否真的适合采用微前端的方式，避免盲目引入微前端导致项目难以维护，得不偿失。

我们认为，如果项目中存在以下两个场景，比较适合采用微前端架构：

- 功能模块较多，且各个功能模块相对较为独立的中后台系统。
- 项目存在大量历史遗留问题，希望在保留已有功能的基础上，开发新的功能模块。

其他大部分项目都可以通过调整代码结构，构建单页应用，甚至采用最传统的多页应用等方式来进行优化、调整，从来达到降低耦合的目的。微前端并不是“银弹”。

期许

- 从去年 12 月立项至今，Bifrost 经历了近一年的迭代，发布了 2 个大版本和 38 个小版本。诞生之初，Bifrost 仅仅是针对企管平台这个特定业务场景的微前端方案。如今，已进化为面向 Vue 技术栈的通用微前端框架。期间，我们围绕 Bifrost，逐步完善了整个微前端技术体系的建设，实现了 Bifrost 主 / 子系统的脚手架工程和命令行工具、子系统的管理平台、灰度发布功能等一系列

平台和工具，完成了 Bifrost 微前端生态的雏形。



当然，Bifrost 依然还有很多可以提升的地方。未来，我们将会从以下几个方面进一步完善 Bifrost：

- 提供更加完善的前端微服务治理工具。
- 实现 JS 和 CSS 沙盒。
- 支持更多的技术栈。

结语

随着前端工程的日益复杂，我们对可扩展的前端架构的诉求也变得更加强烈。微前端作为一种前端解藕的方案，自然更加频繁地被大家所提及和应用。另一方面，虽然网上已经有了很多关于微前端的讨论，但依然缺乏真正落地到生产环境的案例。因此，我们希望通过对闪购团队近半年在微前端方案上的实践分享，帮助大家对微前端从概念到应用有一个更加清晰的认识，也期待与大家一起交流，碰撞出更多的火花。

作者简介

雨甫，美团闪购前端研发工程师。

招聘

美团闪购是美团点评旗下的零售到家业务，闪购专注于为消费者提供丰富、便捷的零售品类选择和及时配送服务，为零售商家提供线上、线下的整体解决方案，助力商家提升经营效率。
用户通过手机下单即可快速买到周边各类商家提供的丰富商品，涵盖食材生鲜、超市便利、鲜花绿植、母婴用品和健康护理等众多品类。美团闪购与美团外卖共享配送网络，平均 30 分钟配送上门，24 小时无间断配送，打造全品类一站式零售到家平台。
美团闪购前端团队诚招高级前端开发、前端开发专家。欢迎各位大佬的加入，共同打造极致的 LBS 电商体验。感兴趣同学可投递简历至：tech@meituan.com（邮件标题注明：美团闪购前端团队）

Litho 的使用及原理剖析

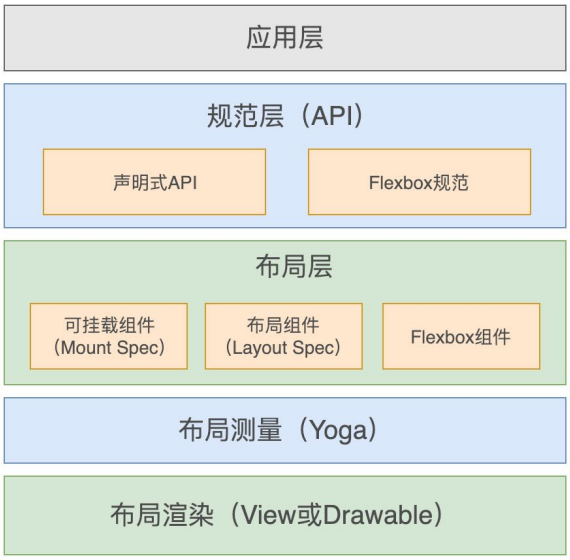
少宽　张颖

1. 什么是 Litho？

Litho 是 Facebook 推出的一套高效构建 Android UI 的声明式框架，主要目的是提升 RecyclerView 复杂列表的滑动性能和降低内存占用。下面是 Litho 官网的介绍：

> Litho is a declarative framework for building efficient user interfaces (UI) on Android. It allows you to write highly-optimized Android views through a simple functional API based on Java annotations. It was primarily built to implement complex scrollable UIs based on RecyclerView. With Litho, you build your UI in terms of components instead of interacting directly with traditional Android views. A component is essentially a function that takes immutable inputs, called props, and returns a component hierarchy describing your user interface.
>
> Litho 是高效构建 Android UI 的声明式框架，通过注解 API 创建高优的 Android 视图，非常适用于基于 Recyclerview 的复杂滚动列表。Litho 使用一系列组件构建视图，代替了 Android 传统视图交互方式。组件本质上是一个函数，它接受名为 Props 的不可变输入，并返回描述用户界面的组件层次结构。

Litho 是一套完全不同于传统 Android 的 UI 框架，它继承了 Facebook 一向大胆创新的风格，突破性地在 Android 上实现了 React 风格的 UI 框架。架构图如下：



应用层：上层 Android 应用接入层。

规范层（API）：允许用户使用声明式的 API（注解）来构建符合 Flexbox 规范的布局。

布局层：Litho 使用可挂载组件、布局组件和 Flexbox 组件来构建布局，其中可挂载组件和布局组件允许用户使用规范来定义，各个组件的具体用法下面的组件规范中会详细介绍。在 Litho 中每一个组件都是一个独立的功能模块。Litho 的组件和 React 的组件相类似，也具有属性和状态的概念，通过状态的变更来控制组件的展示样式。

布局测量：Litho 使用 Yoga 来完成组件布局的异步或同步（可根据场景定制）测量和计算，实现了布局的扁平化。

布局渲染：Litho 不仅支持使用 View 来渲染视图，还可以使用更轻量的 Drawable 来渲染视图。Litho 实现了大量使用 Drawable 来渲染的基础组件，可以进一步拍平布局。

除了上面提到的扁平化布局，Litho 还实现了布局的细粒度复用和异步计算布局的能力，对于这些功能的实现在 Litho 的特性及原理剖析中详细介绍。下面先介绍一

下大家比较关心的 Litho 使用方法。

2. Litho 的使用

Litho 的使用方式相比于传统的 Android 来说有些另类，它抛弃了通过 XML 定义布局的方式，采用声明式的组件在 Java 中构建布局。

Litho 和原生 Android 在使用上的区别

Android 传统布局：首先在资源文件 res/layout 目录下定义布局文件 xx.xml，然后在 Activity 或 Fragment 中引用布局文件生成视图，示例如下：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello World"
  android:textAlignment="center"
  android:textColor="#666666"
  android:textSize="40dp" />
```

```
public class MainActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.helloworld);
  }
}
```

Litho 布局：Litho 抛弃了 Android 原生的布局方式，通过组件方式构建布局生成视图，示例如下：

```
public class MainActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
      ComponentContext context = new ComponentContext(this);
      final Text.Builder builder = Text.create(context);
      final Component = builder.text("Hello World")
                .textSizeDip(40)
                .textColor(Color.parseColor("#666666"))
                .textAlignment(Layout.Alignment.ALIGN_CENTER)
```

```
                .build();
        LithoView view = LithoView.create(context, component);
        setContentView(view);
    }
}
```

Litho 自定义视图

Litho 中的视图单元叫做 Component，可以直观的翻译为“组件”，它的设计理念来自于 React 组件化的思想。每个组件持有描述一个视图单元所必须的属性和状态，用于视图布局的计算工作。视图最终的绘制工作是由组件指定的绘制单元（View 或者 Drawable）来完成的。

Litho 组件的创建方式也和原生 View 的创建方式有着很大的区别。Litho 使用注解定义了一系列的规范，我们需要使用 Litho 的注解来定义自己的组件生成规则，最终由 Litho 在编译期自动编译生成真正的组件。

组件规范

Litho 提供了两种类型的组件规范，分别是 Layout Spec 规范和 Mount Spec 规范。下面分别介绍两种规范的使用方式：

Layout Spec 规范：用于生成布局类型组件的规范，布局组件在逻辑上等同于 Android 中的 ViewGroup，用于组织其他组件构成一个布局。它要求我们必须使用 @LayoutSpec 注解来注明，并实现一个标注了 @OnCreateLayout 注解的方法。示例如下：

```
@LayoutSpec
class HelloComponentSpec {
  @OnCreateLayout
  static Component onCreateLayout(ComponentContext c, @Prop String name) {
    return Column.create(c)
      .child(Text.create(c)
        .text("Hello, " + name)
        .textSizeRes(R.dimen.my_text_size)
        .textColor(Color.BLACK)
        .paddingDip(ALL, 10)
        .build())
      .child(Image.create(c)
        .drawableRes(R.drawable.welcome)
```

```
            .scaleType(ImageView.ScaleType.CENTER_CROP)
            .build())
          .build();
    }
  }
```

最终 Litho 会在编译时生成一个名为 HelloComponent 的组件。

```
  public final class HelloComponent extends Component {

    @Prop(resType = ResType.NONE,optional = false) String name;

    private HelloComponent() {
      super();
    }

    @Override
    protected Component onCreateLayout(ComponentContext c) {
      return (Component) HelloComponentSpec.
onCreateLayout((ComponentContext) c, (String)
name);
    }

    ...

    public static Builder create(ComponentContext context, int
defStyleAttr, int defStyleRes) {
      Builder builder = sBuilderPool.acquire();
      if (builder == null) {
        builder = new Builder();
      }
    HelloComponent instance = new HelloComponent();
      builder.init(context, defStyleAttr, defStyleRes, instance);
      return builder;
    }

    public static class Builder extends Component.Builder<Builder> {
      private static final String[] REQUIRED_PROPS_NAMES = new String[]
{"name"};
      private static final int REQUIRED_PROPS_COUNT = 1;
      HelloComponent mHelloComponent;

      ...

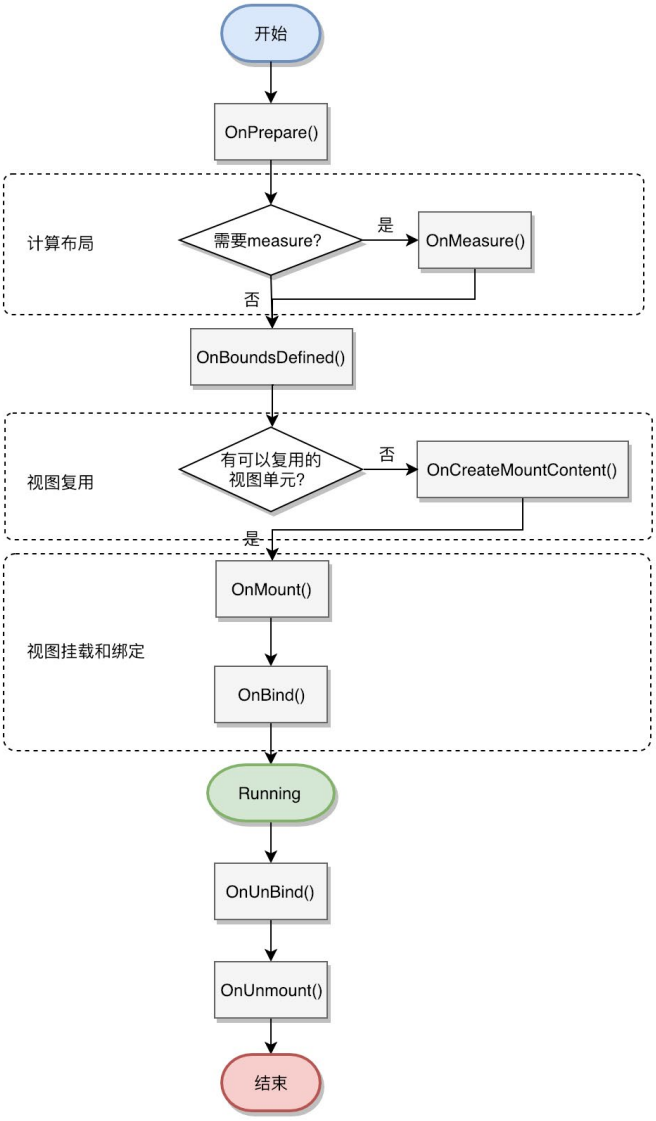
      public Builder name(String name) {
       this.mHelloComponent.name = name;
       mRequired.set(0);
```

```
        return this;
      }

      @Override
      public HelloComponent build() {
        checkArgs(REQUIRED_PROPS_COUNT, mRequired, REQUIRED_PROPS_NAMES);
        HelloComponent helloComponentRef = mHelloComponent;
        release();
        return helloComponentRef;
      }
    }
  }
```

Mount Spec 规范：用来生成可挂载类型组件的规范，用来生成渲染具体 View 或者 Drawable 的组件。同样，它必须使用 @MountSpec 注解来标注，并至少实现一个标注了 @onCreateMountContent 的方法。Mount Spec 相比于 Layout Spec 更复杂一些，它拥有自己的生命周期：

- @OnPrepare，准备阶段，进行一些初始化操作。
- @OnMeasure，负责布局的计算。
- @OnBoundsDefined，在布局计算完成后挂载视图前做一些操作。
- @OnCreateMountContent，创建需要挂载的视图。
- @OnMount，挂载视图，完成布局相关的设置。
- @OnBind，绑定视图，完成数据和视图的绑定。
- @OnUnBind，解绑视图，主要用于重置视图的数据相关的属性，防止出现复用问题。
- @OnUnmount，卸载视图，主要用于重置视图的布局相关的属性，防止出现复用问题。



除了上述两种组件类型，Litho 中还有一种特殊的组件——Layout，它不能使用规范来生成。Layout 是 Litho 中的容器组件，类似于 Android 中的 ViewGroup，但是只能使用 Flexbox 的规范。它可以包含子组件节点，是 Litho 各组件连接的纽带。Layout 组件只是 Yoga 在 Litho 中的代理，组件的所有布局相关的属性都会直接设置给 Yoga，并由 Yoga 完成布局的计算。Litho 实现了两个 Layout 组件 Row 和

Column，分别对应 Flexbox 中的行和列。

Litho 的属性

在 Litho 中属性分为两种，不可变属性称为 Props，可变属性称为 State，下面分别介绍一下两种属性：

Props 属性：组件中使用 @Prop 注解标注的参数集合，具有单向性和不可变性。下面通过一个简单的例子了解一下如何在组件中定义和使用 Props 属性：

```
@MountSpec
class MyComponentSpec {

    @OnPrepare
    static void onPrepare(
      ComponentContext c,
      @Prop(optional = true) String prop1) {
      ...
    }

    @OnMount
    static void onMount(
      ComponentContext c,
      SomeDrawable convertDrawable,
      @Prop(optional = true) String prop1,
      @Prop int prop2) {
        if (prop1 != null) {
        ...
        }
    }
}
```

在上面的代码中，共使用了三次 Prop 注解，分别标注 prop1 和 prop2 两个变量，即定义了 prop1 和 prop2 两个属性。Litho 会在自动编译生成的 MyComponent 类的 Builder 类中生成这两个属性的同名方法。按照如下代码，便可以去使用上面定义的属性：

```
MyComponent.create(c)
  .prop1("My prop 1")
  .prop2(256)
  .build();
```

State 属性：意为“状态”属性，State 属性虽然可变，但是其变化由组件内部控制，例如：输入框、Checkbox 等都是由组件内部去感知用户行为，并更新组件的 State 属性。所以一个组件一旦创建，我们便无法通过任何外部设置去更改它的属性。组件的 State 属性虽然不允许像 Props 属性那样去显式设置，但是我们可以定义一个单独的 Props 属性来当做某个 State 属性的初始值。

3. Litho 的特性及原理剖析

Litho 官网首页通过 4 个段落重点介绍了 Litho 的 4 个特性。

声明式组件

Litho 采用声明式的 API 来定义 UI 组件，组件通过一组不可变的属性来描述 UI。这种组件化的思想灵感来源于 React，关于声明式组件的用法上面已经详细介绍过了。

传统 Android 布局因为 UI 与逻辑分离，所以开发工具都有强大的预览功能，方便开发者调整布局。而 Litho 采用 React 组件化的思想，通过组件连接了逻辑与布局 UI，虽然 Litho 也提供了对 Stetho 的支持，借助于 Chrome 开发者工具对界面进行调试，不过使用起来并没有那么方便。

异步布局

Android 系统在绘制时为了防止页面错乱，页面所有 View 的测量 (Measure)、布局 (Layout) 以及绘制 (Draw) 都是在 UI 线程中完成的。当页面 UI 非常复杂、视图层级较深时，难免 Measure 和 Layout 的时间会过长，从而导致页面渲染时候丢帧出现卡顿情况。Litho 为解决该问题，提出了异步布局的思想，利用 CPU 的闲置时间提前在异步线程中完成 Measure 和 Layout 的过程，仅在 UI 线程中完成绘制工作。当然，Litho 只是提供了异步布局的能力，它主要使用在 RecyclerView 等可以提前知道下一个视图长什么样子的场景。

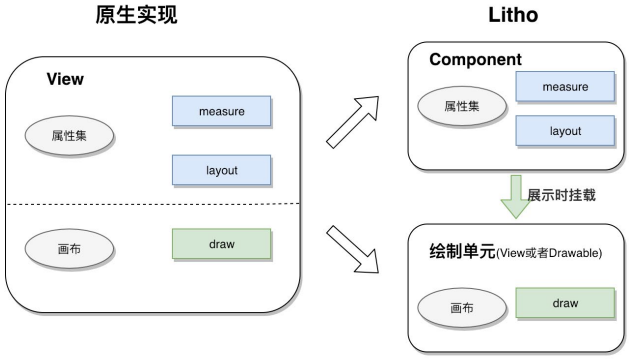
异步布局原理剖析

针对 RecyclerView 等滑动列表，由于可以提前知道接下来要展示的一个甚至多

个条目的视图样式，所以只要提前创建好下一个或多个条目的视图，就可以提前完成视图的布局工作。

那么 Android 原生为什么不支持异步布局呢？主要有以下两个原因：

- View 的属性是可变的，只要属性发生变化就可能导致布局变化，因此需要重新计算布局，那么提前计算布局的意义就不大了。而 Litho 组件的属性是不可变的，所以对于一个组件来说，它的布局计算结果是唯一且不变的。
- 提前异步布局就意味着要提前创建好接下来要用到的一个或者多个条目的视图，而 Android 原生的 View 作为视图单元，不仅包含一个视图的所有属性，而且还负责视图的绘制工作。如果要在绘制前提前去计算布局，就需要预先去持有大量未展示的 View 实例，大大增加内存占用。反观 Litho 的组件则没有这个问题，Litho 的组件只是视图属性的一个集合，仅负责计算布局，绘制工作由指定的绘制单元来完成，相比与传统的 View 显然 Litho 的组件要轻量的多。所以在 Litho 中，提前创建好接下来要用到的多个条目的组件，并不会带来性能问题，甚至还可以直接把组件当成滑动列表的数据源。如下图所示：



扁平化的视图

使用 Litho 布局，我们可以得到一个极致扁平的视图效果。它可以减少渲染时的递归调用，加快渲染速度。

下面是同一个视图在 Android 和 Litho 实现下的视图层级效果对比。可以看到，

同样的样式，使用 Litho 实现的布局要比使用 Android 原生实现的布局更加扁平。

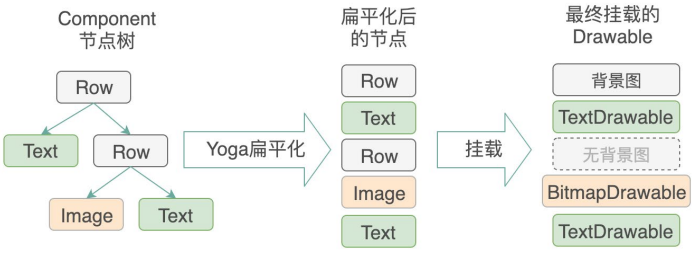


扁平化视图原理剖析

Litho 使用 Flexbox 来创建布局，最终生成带有层级结构的组件树。然后 Litho 对布局层级进行了两次优化。

- 使用了 Yoga 来进行布局计算，Yoga 会将 Flexbox 的相对布局转成绝对布局。经过 Yoga 处理后的布局没有了原来的布局层级，变成了只有一层。虽然不能解决过度绘制的问题，但是可以有效地减少渲染时的递归调用。
- 前面介绍过 Litho 的视图渲染由绘制单元来完成，绘制单元可以是 View 或者更加轻量的 Drawable，Litho 自己实现了一系列挂载 Drawable 的基本视图组件。通过使用 Drawable 可以减少内存占用，同时相比于 View，Android 无法检查出 Drawable 的视图层级，这样可以使视图效果看起来更加扁平。

原理如下图所示，Litho 会先把组件树拍平成没有层级的列表，然后使用 Drawable 来绘制对应的视图单元。



Litho 使用 Drawable 代替 View 能带来多少好处呢？ Drawable 和 View 的区别在于前者不能和用户交互，只能展示，因此 Drawable 不会像 View 那样持有很多变量和引用，所以 Drawable 比 View 从内存上看要轻量很多。举个例子：50 个同样展示“Hello world”的 TextView 和 TextDrawable 在内存占比上，前者几乎是后者的 8 倍。对比图如下，Shallow Size 表示对象自身占用的内存大小。

Class Name	Allocations	Deallocations	Total Count	Shallow Size
TextView *(android.widget)*	0	0	50	40,450
TextTrieMap$Node *(android.icu.impl)*	0	0	1	24
TextTrieMap *(android.icu.impl)*	0	0	1	13
TextPaint *(android.text)*	0	0	104	14,144
TextLine$DecorationInfo *(android.text)*	0	0	1	26
TextLine *(android.text)*	0	0	1	78
TextLayoutBuilder$Params *(com.facebook.fbui.textl*	0	0	1	95
TextLayoutBuilder *(com.facebook.fbui.textlayoutbui*	0	0	1	38
TextDrawable *(com.facebook.litho.widget)*	0	0	50	5,450

绘制单元的降级策略

由于 Drawable 不具有交互能力，所以对于使用 Drawable 无法实现的交互场景，Litho 会自动降级成 View。主要有以下几种场景：

- 有监听点击事件。
- 限制子视图绘出父布局。
- 有监听焦点变化。
- 有设置 Tag。
- 有监听触摸事件。
- 有光影效果。

对于以上场景的使用请仔细考虑，过多的使用会导致 Litho 的层级优化效果变差。

对比 Android 的约束布局

为了解决布局嵌套问题，Android 推出了约束布局（ConstraintLayout），使用约束布局也可以达到扁平化视图的目的，那么使用 Litho 的好处是什么呢?

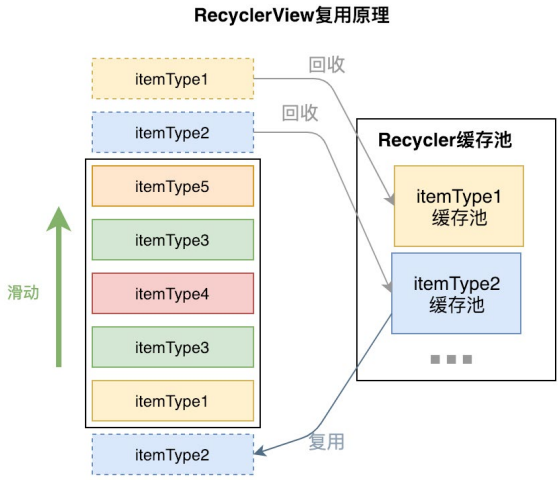
Litho 可以更好地实现复杂布局。约束布局虽然可以实现扁平效果，但是它使用了大量的约束来固定视图的位置。随着布局复杂程度的增加，约束条件变得越来越多，可读性也变得越来越差。而 Litho 则是对 Flexbox 布局进行的扁平化处理，所以实际使用的还是 Flexbox 布局，对于复杂的布局 Flexbox 布局可读性更高。

细粒度的复用

Litho 中的所有组件都可以被回收，并在任何位置进行复用。这种细粒度的复用方式可以极大地提高内存使用率，尤其适用于复杂滑动列表，内存优化非常明显。

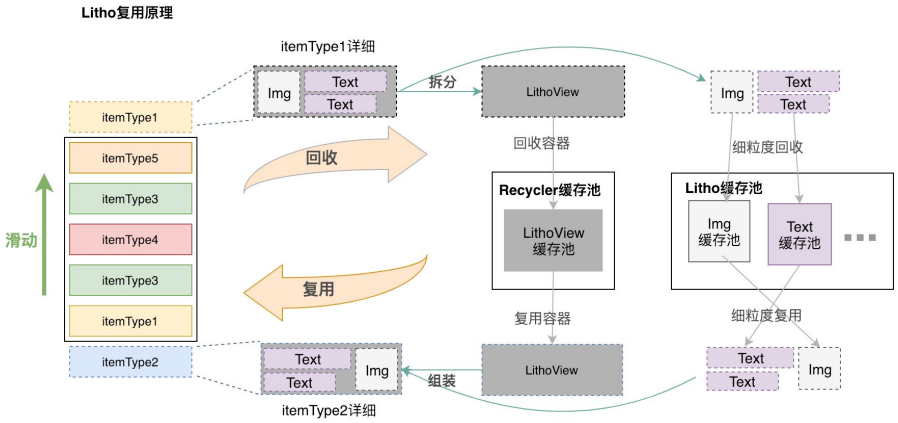
原生 RecyclerView 复用原理剖析

原生的 RecyclerView 视图按模板类型进行存储并复用，也就是说模板类型越多，所需存储的模板种类也就越多，导致内存占用越来越大。原理如下图。滑出屏幕的 itemType1 和 itemType2 都会在 Recycler 缓存池保存，等待后面滑进屏幕的条目的复用。



细粒度复用优化内存原理剖析

在 Litho 中，item 在回收前，会把 LithoView 中挂载的各个绘制单元拆分出来（解绑），由 Litho 自己的缓存池去分类回收，在展示前由 LithoView 按照组件树的样式组装（挂载）各个绘制单元，这样就达到了细粒度复用的目的。原理如下图。滑出屏幕的 itemType1 会被拆分成一个个的视图单元。LithoView 容器由 Recycler 缓存池回收，其他视图单元由 Litho 的缓存池分类回收。

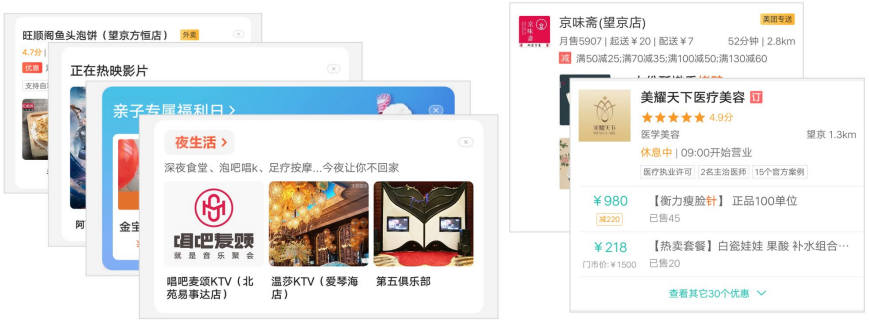


使用细粒度复用的 RecyclerView 的缓存池不再需要区分模板类型来缓存大量的视图模板，只需要缓存 LithoView 容器。细粒度回收的视图单元数量要远远小于原来缓存在各个视图模板中的视图单元数量。

实践

美团对 Litho 进行了二次开发，在美团的 MTFlexbox 动态化实现方案（简称动态布局）中把 Litho 作为底层 UI 渲染引擎来使用。通过动态布局的预览工具，为 Litho 提供实时预览能力，同时可以有效发挥 Litho 的性能优化效果。

目前 Litho+ 动态布局的实现方案已经应用在了美团 App 中，给美团 App 带来了不错的性能提升。后续博主会详细介绍 Litho+ 动态布局在美团性能优化的实践方案。

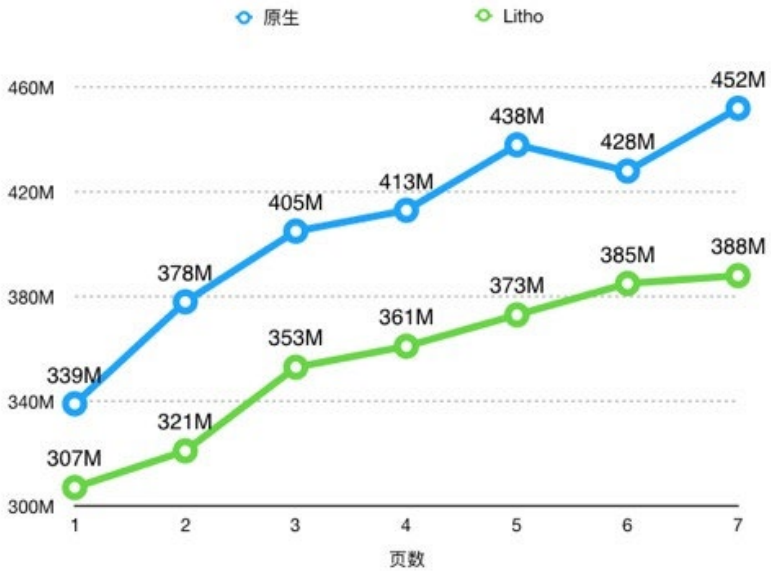


首页部分卡片　　**搜索部分卡片**

使用 Litho+ 动态布局实现的部分卡片

内存数据

由于 Litho 中使用了大量 Drawable 替换 View，并且实现了视图单元的细粒度复用，因此复杂列表滑动时内存优化比较明显。美团首页内存占用随滑动页数变化走势图如下。随着一页一页地滑动，内存优化了 30M 以上。（数据采集自 Vivo x20 手机内存占用情况）



FPS 数据

FPS 的提升主要得益于 Litho 的异步布局能力，提前计算布局可以减少滑动时的帧率波动，所以滑动过程较平稳，不会有高低起伏的卡顿感。（数据采集自魅蓝 2 手机一段时间内连续 fps 的波动情况）



5. 总结

Litho 相对于传统 Android 是颠覆式的，它采用了 React 的思路，使用声明式的 API 来编写 UI。相比于传统 Android，确实在性能优化上有很大的进步，但是如果完全使用 Litho 开发一款应用，需要自己实现很多组件，而 Litho 的组件需要在编译时生成，实时预览方面也有所欠缺。相对于直接使用 Litho 的高成本，把 Litho 封装成 Flexbox 布局的底层渲染引擎是个不错的选择。

6. 参考资料

1. Litho 官网

2. 说一说 Facebook 开源的 Litho
3. React 官网
4. Yoga 官网

7. 作者简介

何少宽，美团 Android 开发工程师，2015 年加入美团，负责美团平台终端业务研发工作。
张颖，美团 Android 开发工程师，2017 年加入美团，负责美团平台终端业务研发工作。

Android 兼容 Java 8 语法特性的原理分析

元合　朝旭

本文主要阐述了 Lambda 表达式及其底层实现（invokedynamic 指令）的原理、Android 第三方插件 RetroLambda 对其的支持过程、Android 官方最新的 dex 编译器 D8 对其的编译支持。通过对这三个方面的跟踪分析，以 Java 8 的代表性特性——Lambda 表达式为着眼点，将 Android 如何兼容 Java8 的过程分享给大家。

Java 8 概述

Java 8 是 Java 开发语言非常重要的一个版本。Oracle 从 2014 年 3 月 18 日发布 Java 8，从该版本起，Java 开始支持函数式编程。特别是吸收了运行在 JVM 上的 Scala、Groovy 等动态脚本语言的特性之后，Java 8 在语言的表达力、简洁性两个方面有了很大的提高。

Java 8 的主要语言特性改进概括起来包括以下几点：

- Lambda 表达（函数闭包）
- 函数式接口（@FunctionalInterface）
- Stream API（通过流式调用支持 map、filter 等高阶函数）
- 方法引用（使用 :: 关键字将函数转化为对象）
- 默认方法（抽象接口中允许存在 default 修饰的非抽象方法）
- 类型注解和重复注解

其中 Lambda 表达、函数式接口、方法引用三个特性为 Java 带来了函数式编程的风格；而 Stream 实现了 map、filter、reduce 等常见的高阶函数，数据源囊括了数组、集合、IO 通道等，这些又为 Java 带来了流式编程或者说链式编程的风格，以上这些风格让 Java 变得越来越现代化和易用。

Android 和 Java 关系

其实 Java 在 Android 的快速发展过程中扮演着非常重要的角色，无论是作为开发语言（Java）、开发 Framework（Android-SDK 引用了 80% 的 JDK-API），还是开发工具（Eclipse or Android Studio）。这些都和 Java 有着千丝万缕的关系。不过可能是受到与 Oracle 的法律诉讼的影响，Google 在 Android 上针对 Java 的升级一直都不是很积极：

- Android 从 1.0 一直升级到 4.4，迭代了将近 19 个 Android 版本，才在 4.4 版本中支持了 Java 7。
- 然后从 Android 4.4 版本开始算起，一直到 Android N(7.0) 共 4 个 Android 版本，才在 Jack/Jill 工具链勉强支持了 Java 8。但由于 Jack/Jill 工具链在构建流程中舍弃了原有 Java 字节码的体系，导致大量既有的技术沉淀无法应用，致使许多 App 工程放弃了接入。
- 最后直到 Android P(9.0) 版本，Google 才在 Android Studio 3.x 中通过新增的 D8 dex 编译器正式支持了 Java 8，但部分 API 并不能全版本支持。

可谓“历经坎坷”。特别是 Rx 大行其道的今天，Rx 配合 Java 8 特性 Lambda 带来简洁、高效的开发体验，更是让 Android Developer 望眼欲穿。

接下来，本文将从技术原理层面，来分析一下 Android 是如何支持 Java 8 的。

Lambda 表达式

想要更好的理解 Android 对 Java 8 的支持过程，Lambda 表达式这一代表性的“语法糖”是一个非常不错的切入点。所以，我们首先需要搞清楚 Lambda 表达式到底是什么？其底层的实现原理又是什么？

Lambda 表达式是 Java 支持函数式编程的基础，也可以称之为闭包。简单来说，就是在 Java 语法层面允许将函数当作方法的参数，函数可以当做对象。任一 Lambda 表达式都有且只有一个函数式接口与之对应，从这个角度来看，也可以说是该函数式接口的实例化。

Lambda 表达式

通用格式：

```
语法格式:
(parameters) -> expression  或者  (parameters) -> { statements; }

格式理解:
( 对应函数式接口的参数列表 ) -> { 对应函数接口的实现方法 }
```

简单范例：

```
package com.j8sample;

public class J8Sample {

    public static void main(String arg[]) {

        Runnable runnable = () -> System.out.println("xixi"); // Lambda表达式1
        new Thread(runnable).start();

        new Thread(() -> {
            System.out.println("haha"); // Lambda表达式2
        }).start();
    }
}
```

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

说明：

- Lambda 表达式中 () 对应的是函数式接口 –run 方法的参数列表。
- Lambda 表达式中 System.out.println（“xixi”）/ System.out.println（“ha–ha”），在运行时会是具体的 run 方法实现。

Lambda 表达式原理

针对实例中的代码，我们来看下编译之后的字节码：

```
javac J8Sample.java  ->  J8Sample.class

javap -c -p J8Sample.class
```

```
Compiled from "J8Sample.java"
public class com.j8sample.J8Sample {
  public com.j8sample.J8Sample();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokedynamic #2,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
       5: astore_1
       6: new           #3                  // class java/lang/Thread
       9: dup
      10: aload_1
      11: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      14: invokevirtual #5                  // Method java/lang/Thread.start:()V
      17: new           #3                  // class java/lang/Thread
      20: dup
      21: invokedynamic #6,  0              // InvokeDynamic #1:run:()Ljava/lang/Runnable;
      26: invokespecial #4                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
      29: invokevirtual #5                  // Method java/lang/Thread.start:()V
      32: return

  private static void lambda$main$1();
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #8                  // String haha
       5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  private static void lambda$main$0();
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #10                 // String xixi
       5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

从字节码中我们可以看到：

- 实例中 Lambda 表达式 1 变成了字节码代码块中 Line 11 的 0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable。
- 实例中 Lambda 表达式 2 变成了字节码代码块中 Line 20 的 21: invokedynamic #6, 0 // InvokeDynamic #1:run:()Ljava/lang/Runnable。

可见，Lambda 表达式在虚拟机层面上，是通过一种名为 invokedynamic 字节码指令来实现的。那么 invokedynamic 又是何方神圣呢?

invokedynamic 指令解读

invokedynamic 指令是 Java 7 中新增的字节码调用指令，作为 Java 支持动态类

型语言的改进之一，跟 invokevirtual、invokestatic、invokeinterface、invokespecial 四大指令一起构成了虚拟机层面各种 Java 方法的分配调用指令集。区别在于：

- 后四种指令，在编译期间生成的 class 文件中，通过常量池 (Constant Pool) 的 MethodRef 常量已经固定了目标方法的符号信息（方法所属者及其类型，方法名字、参数顺序和类型、返回值）。虚拟机使用符号信息能直接解释出具体的方法，直接调用。
- 而 invokedynamic 指令在编译期间生成的 class 文件中，对应常量池 (Constant Pool) 的 Invokedynamic_Info 常量存储的符号信息中并没有方法所属者及其类型，替代的是 BootstapMethod 信息。在运行时，通过引导方法 BootstrapMethod 机制动态确定方法的所属者和类型。这一特点也非常契合动态类型语言只有在运行期间才能确定类型的特征。

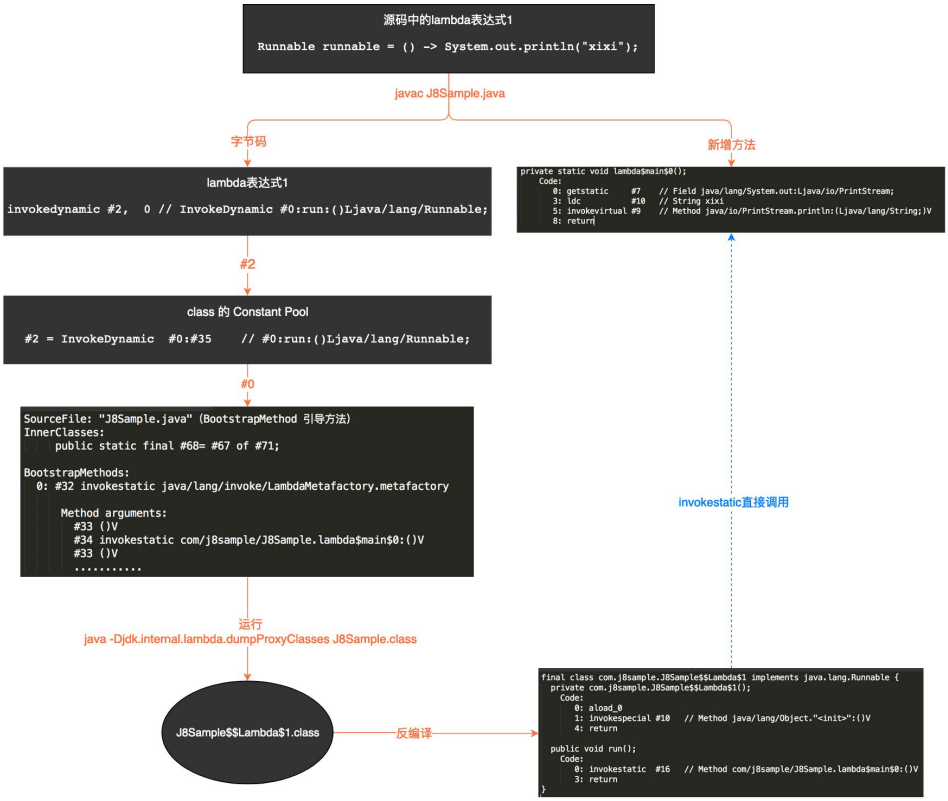
那么，invokedynamic 如何通过引导方法找到所属者及其类型？我们依然结合前面的 J8Sample 实例：

```
javap -v J8Sample.class
```

```
Constant pool:  (class 常量池)
   #1 = Methodref          #12.#30        // java/lang/Object."<init>":()V
   #2 = InvokeDynamic      #0:#35         // #0:run:()Ljava/lang/Runnable;
   #3 = Class              #36            // java/lang/Thread
   #4 = Methodref          #3.#37         // java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
   #5 = Methodref          #3.#38         // java/lang/Thread.start:()V
   #6 = InvokeDynamic      #1:#35         // #1:run:()Ljava/lang/Runnable;
   #7 = Fieldref           #40.#41        // java/lang/System.out:Ljava/io/PrintStream;
   #8 = String             #42            // haha
   #9 = Methodref          #43.#44        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #10 = String             #45            // xixi
  #11 = Class              #46            // com/j8sample/J8Sample
  #12 = Class              #47            // java/lang/Object
  #13 = Utf8               <init>
  ..........

SourceFile: "J8Sample.java" (BootstrapMethod 引导方法)
InnerClasses:
     public static final #68= #67 of #71;
    //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
                                                                      Ljava/lang/String;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodHandle;
                                                                      Ljava/lang/invoke/MethodType;)
                                                                      Ljava/lang/invoke/CallSite;
      Method arguments:
        #33 ()V
        #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
        #33 ()V
        ..........
```

结合 J8Sample.class 字节码，并对 invokedynamic 指令调用过程进行跟踪分析。总结如下：



依据上图 invokedynamic 调用步骤，我们一步一步做一个分析讲解。

步骤 1　选取 J8Sample.java 源码中 Lambda 表达式 1：

Runnable runnable = () -> System.out.println("xixi"); // lambda 表达式 1

步骤 2　通过 javac J8Sample.java 编译得到 J8Sample.class 之后，

Lambda 表达式 1 变成：0: invokedynamic #2, 0 // InvokeDynamic #0:run:()Ljava/lang/Runnable;

对应在 J8Sample.class 中发现了新增的私有静态方法：

```
private static void lambda$main$0();
  Code:
     0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #10                 // String xixi
     5: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
     8: return
```

步骤 3　针对表达式 1 的字节码分析 #2 对应的是 class 文件中的常量池：

#2 = InvokeDynamic #0:#35 // #0:run:()Ljava/lang/Runnable;

注意，这里 InvokeDynamic 不是指令，代表的是 Constant_InvokeDynamic_Info 结构。

步骤 4　结构后面紧跟的 #0 标识的是 class 文件中的 BootstrapMethod 区域中引导方法的索引：

```
BootstrapMethods:
  0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
                                                                      Ljava/lang/String;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodType;
                                                                      Ljava/lang/invoke/MethodHandle;
                                                                      Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #33 ()V
      #34 invokestatic com/j8sample/J8Sample.lambda$main$0:()V
      #33 ()V
```

步骤 5　引导方法中的 java/lang/invoke/LambdaMetafactory.metafactory 才是 invokedynamic 指令的关键：

```
public static CallSite metafactory(MethodHandles.Lookup caller,
                                   String invokedName,
                                   MethodType invokedType,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType)
        throws LambdaConversionException {
    AbstractValidatingLambdaMetafactory mf;
    mf = new InnerClassLambdaMetafactory(caller, invokedType,
                                         invokedName, samMethodType,
                                         implMethod, instantiatedMethodType,
                                         isSerializable: false, EMPTY_CLASS_ARRAY, EMPTY_MT_ARRAY);
    mf.validateMetafactoryArgs();
    return mf.buildCallSite();
}
```

```
public InnerClassLambdaMetafactory(MethodHandles.Lookup caller,
                                   MethodType invokedType,
                                   String samMethodName,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType,
                                   boolean isSerializable,
                                   Class<?>[] markerInterfaces,
                                   MethodType[] additionalBridges)
        throws LambdaConversionException {
    super(caller, invokedType, samMethodName, samMethodType,
          implMethod, instantiatedMethodType,
          isSerializable, markerInterfaces, additionalBridges);
    implMethodClassName = implDefiningClass.getName().replace( oldChar: '.',  newChar: '/');
    implMethodName = implInfo.getName();
    implMethodDesc = implMethodType.toMethodDescriptorString();
    implMethodReturnClass = (implKind == MethodHandleInfo.REF_newInvokeSpecial)
            ? implDefiningClass
            : implMethodType.returnType();
    constructorType = invokedType.changeReturnType(Void.TYPE);
    lambdaClassName = targetClass.getName().replace( oldChar: '.',  newChar: '/') + "$$Lambda$" + counter.incrementAndGet();
    cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    int parameterCount = invokedType.parameterCount();
    if (parameterCount > 0) {
        argNames = new String[parameterCount];
        argDescs = new String[parameterCount];
        for (int i = 0; i < parameterCount; i++) {
            argNames[i] = "arg$" + (i + 1);
            argDescs[i] = BytecodeDescriptor.unparse(invokedType.parameterType(i));
        }
    } else {
        argNames = argDescs = EMPTY_STRING_ARRAY;
    }
}
```

该方法会在运行时，在内存中动态生成一个实现 Lambda 表达式对应函数式接口的实例类型，并在接口的实现方法中调用步骤 2 中新增的静态私有方法。

步骤 6　使用 java -Djdk.internal.lambda.dumpProxyClasses J8Sample.class 运行一下，可以内存中动态生成的类型输出到本地：

```
public InnerClassLambdaMetafactory(MethodHandles.Lookup caller,
                                   MethodType invokedType,
                                   String samMethodName,
                                   MethodType samMethodType,
                                   MethodHandle implMethod,
                                   MethodType instantiatedMethodType,
                                   boolean isSerializable,
                                   Class<?>[] markerInterfaces,
                                   MethodType[] additionalBridges)
        throws LambdaConversionException {
    super(caller, invokedType, samMethodName, samMethodType,
          implMethod, instantiatedMethodType,
          isSerializable, markerInterfaces, additionalBridges);
    implMethodClassName = implDefiningClass.getName().replace( oldChar: '.',  newChar: '/');
    implMethodName = implInfo.getName();
    implMethodDesc = implMethodType.toMethodDescriptorString();
    implMethodReturnClass = (implKind == MethodHandleInfo.REF_newInvokeSpecial)
            ? implDefiningClass
            : implMethodType.returnType();
    constructorType = invokedType.changeReturnType(Void.TYPE);
    lambdaClassName = targetClass.getName().replace( oldChar: '.',  newChar: '/') + "$$Lambda$" + counter.incrementAndGet();
    cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    int parameterCount = invokedType.parameterCount();
    if (parameterCount > 0) {
        argNames = new String[parameterCount];
        argDescs = new String[parameterCount];
        for (int i = 0; i < parameterCount; i++) {
            argNames[i] = "arg$" + (i + 1);
            argDescs[i] = BytecodeDescriptor.unparse(invokedType.parameterType(i));
        }
    } else {
        argNames = argDescs = EMPTY_STRING_ARRAY;
    }
}
```

步骤 7　通过 javap -p -c J8Sample\$\$Lambda\$1.class 反编译一下，可以看到生成类的实现：

```
final class com.j8sample.J8Sample$$Lambda$1 implements java.lang.Runnable {
  private com.j8sample.J8Sample$$Lambda$1();
    Code:
       0: aload_0
       1: invokespecial #10                 // Method java/lang/Object."<init>":()V
       4: return

  public void run();
    Code:
       0: invokestatic  #16                 // Method com/j8sample/J8Sample.lambda$main$0:()V
       3: return
}
```

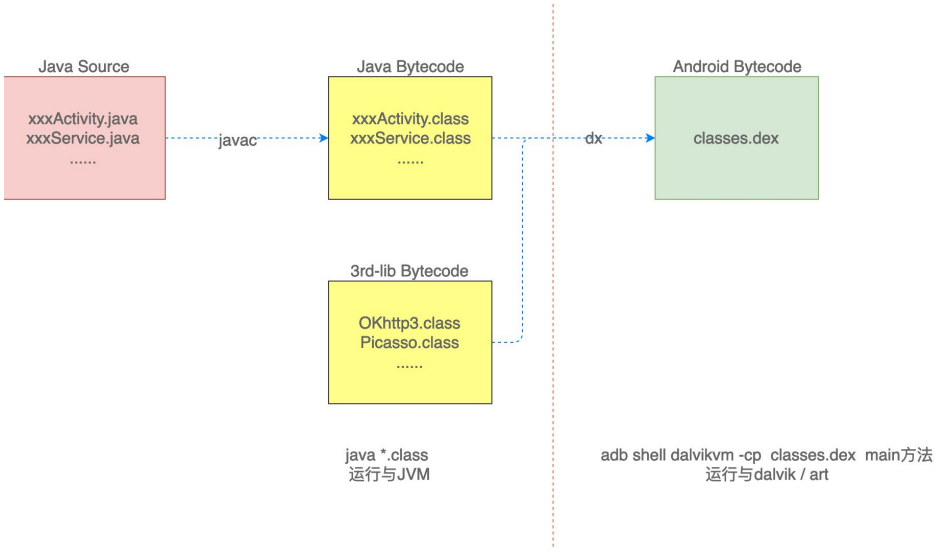
在 run 方法中使用了 invokestatic 指令，直接调用了 J8Sample.lambda$-main$0 这个在编译期间生成的静态私有方法。

至此，上面 7 个步骤就是 Lambda 表达式在 Java 的底层的实现原理。Android 针对这些实现会怎么处理呢?

Android 不能直接支持

回到 Android 系统上，Java-Bytecode（JVM 字节码）是不能直接运行在 Android 系统上的，需要转换成 Android-Bytecode（Dalvik/ART 字节码）。

如图：

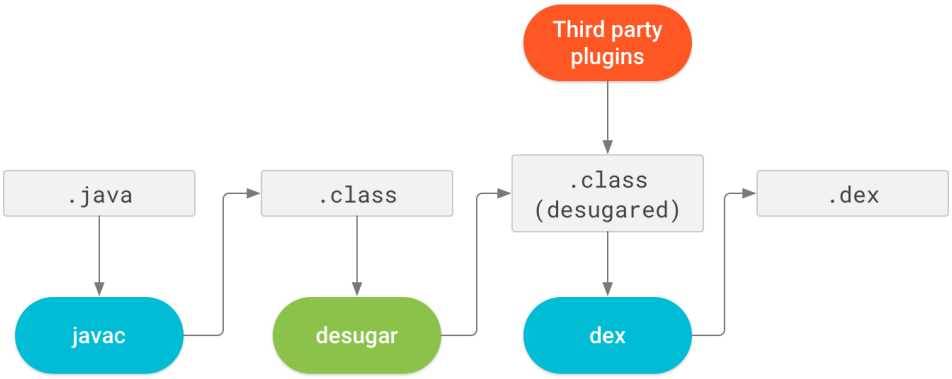


通过 Lambda 这节，我们知道 Java 底层是通过 invokedynamic 指令来实现，由于 Dalvik/ART 并没有支持 invokedynamic 指令或者对应的替代功能。简单的来说，就是 Android 的 dex 编译器不支持 invokedynamic 指令，导致 Android 不能直接支持 Java 8。

Android 间接支持

既然不能直接支持，那就只能在 Java-Bytecode 转换到 Android-Bytecode 这一过程中想办法，间接支持。这个间接支持的过程我们统称为 Desugar（脱糖）过程。

官方流程图：

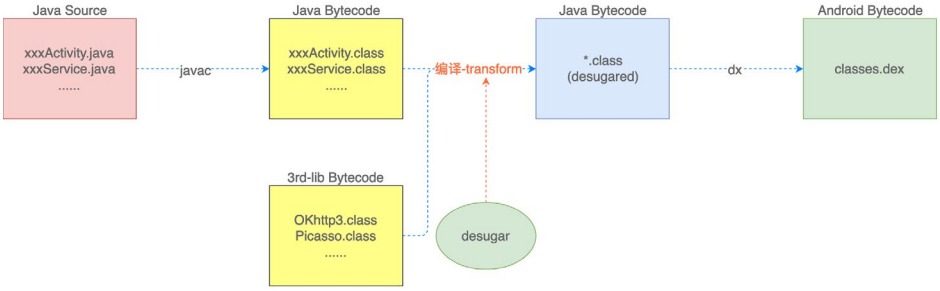


当前，无论是 RetroLambda，还是 Google 的 Jack & Jill 工具，还是最新的 D8 dex 编译器：

- 流程方面：都是按照如上图所示的官方流程进行 Desugar 的。
- 原理方面：却是参照 Lambda 在 Java 底层的实现，并将这些实现移至到 RetroLambda 插件或者 Jack、D8 编译器工具中。

下面我们逐个分析解读一下。

Android 间接支持之 RetroLambda



如图所示，RetroLambda 的 Desugar 过程发生在 javac 将源码编译完成之后，dx 工具进行 dex 编译之前。

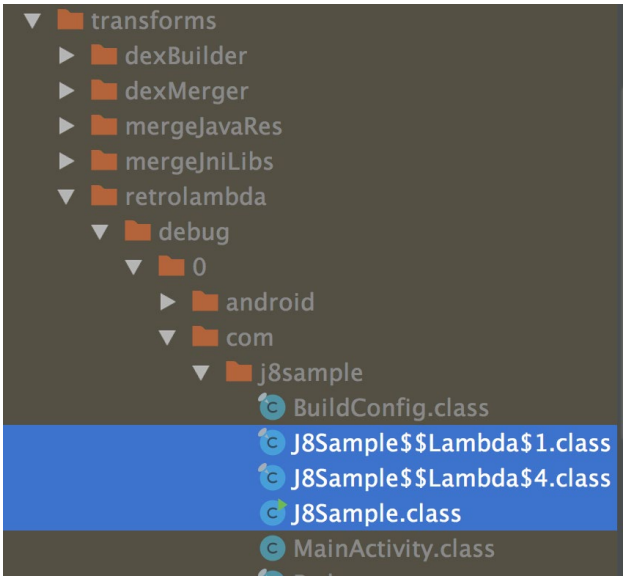
RetroLambda Desugar

其实就是参照 invokedynamic 指令解读一节中的步骤 5，根据 java/lang/invoke/LambdaMetafactory.metafactory 方法，直接将原本在运行时生成在内存中的 J8Sample\$\$Lambda\$1.class，在 javac 编译结束之后，dx 编译 dex 之前，直接生成到本地，并使用生成的 J8Sample\$\$Lambda\$1 类修改 J8Sample.class 字节码文件，将 J8Sample.class 中的 invokedynamic 指令替换成 invokestatic 指令。

将实例中的 J8Sample.java 放到一个配置了 Retrolambda 的 Android 工程中：



AndroidStudio -> Build -> make project 编译之后：



app:transformClassesWithRetrolambdaForDebug 任务发生在 app:compileDebugJavaWithJavac（对应 javac）之后，和 app:transformDexArchiveWithDexMergerForDebug（对应 dx）之前，同时在 build/intermediates/transforms/retrolambda 下面生产如图所示的 class 文件。

J8Sample.class 和 J8Sample$$Lambda$1.class 反编译之后的代码如下：

```
package com.j8sample;

public class J8Sample {
    public J8Sample() {
    }

    public static void main(String[] arg) {
        Runnable runnable = J8Sample$$Lambda$1.lambdaFactory$();
        (new Thread(runnable)).start();
        (new Thread(J8Sample$$Lambda$4.lambdaFactory$())).start();
    }
}
```

```
package com.j8sample;

// $FF: synthetic class
final class J8Sample$$Lambda$1 implements Runnable {
    private static final J8Sample$$Lambda$1 instance = new J8Sample$$Lambda$1();

    private J8Sample$$Lambda$1() {
    }

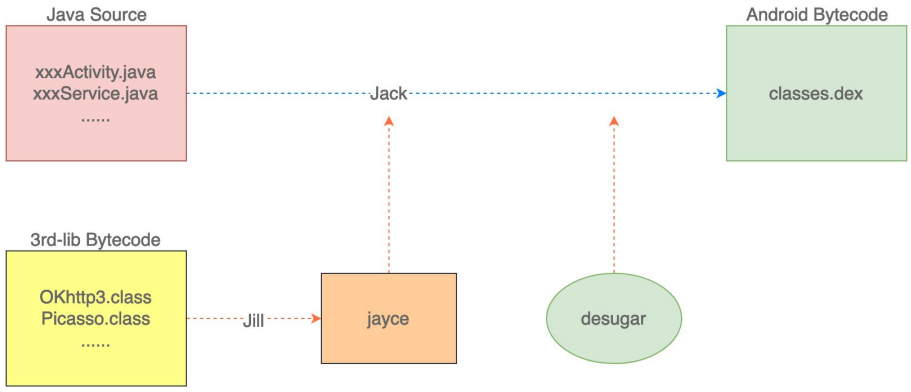
    public void run() {
        J8Sample.lambda$main$0();
    }

    public static Runnable lambdaFactory$() { return instance; }
}
```

通过反编译代码，可以看出 J8Sample.class 中 Lambda 表达式已经被我们熟悉的 1.7or1.6 的语句所替代。

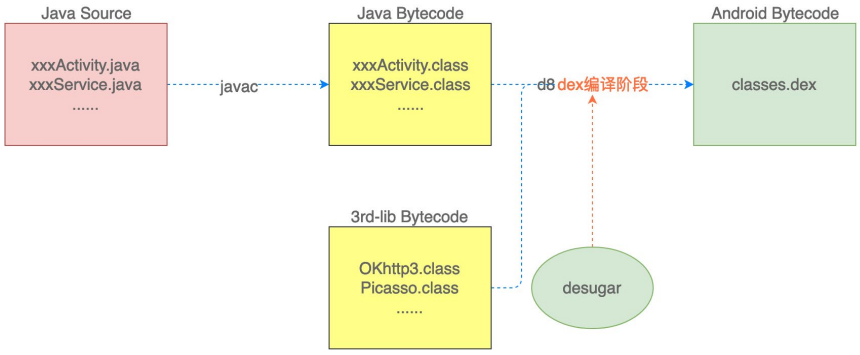
注意：右图中 J8Sample.lambda$main$0() 方法在左图中没有显示出来，但是 J8Sample.class 字节码确实是存在的。

Android 间接支持之 Jack&Jill 工具



Jack 是基于 Eclipse 的 ecj 编译开发的，Jill 是基于 ASM4 开发的。Jack&Jill 工具链是 Google 在 Android N(7.0) 发布的，用于替换 javac&dx 的工具链，并且在 jack 过程内置了 Desugar 过程。

但是在 Android P(9.0) 的时候将 Jack&Jill 工具链废弃了，被 javac&D8 工具链替代了。这里就不做 Desugar 具体分析了。



D8 是 Android P(9.0) 新增的 dex 编译器。并在 Android Studio 3.1 版本中默认使用 D8 作为 dex 的默认编译器。

D8 Desugar

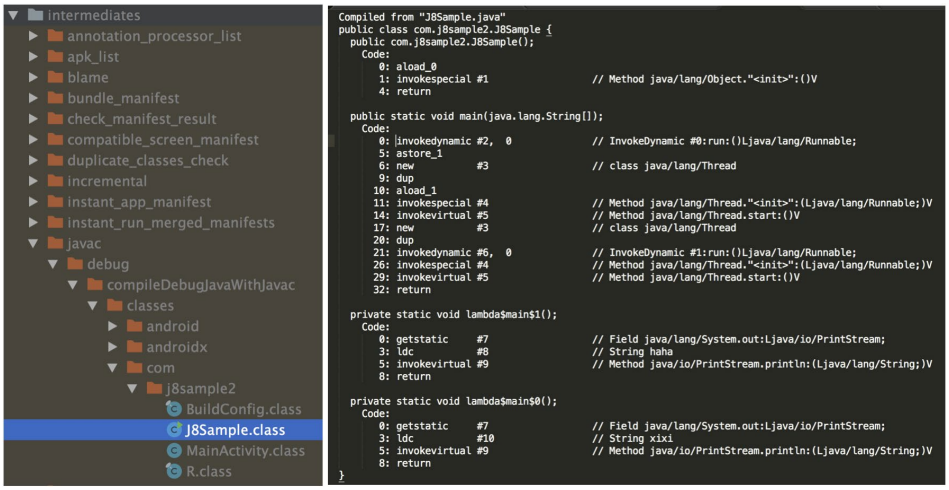
如图所示，Desugar 过程放在了 D8 的内部，由 Android Studio 这个 IDE 来实现这个转换，原理基本和 RetroLambda 是一样。

本质上也是参照 java/lang/invoke/LambdaMetafactory.metafactory 方法直接将原本在运行时生成在内存中的 J8Sample\$\$Lambda\$1.class，在 D8 的编译 dex 期间，直接生成并写入到 dex 文件中。

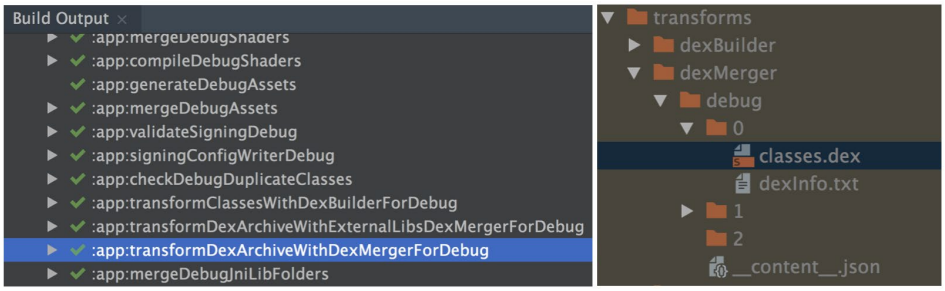
同样，将实例中的 J8Sample.java 放到支持 D8 的 Android 工程中：



同样，AndroidStudio -> Build -> make project 编译之后：



javac 编译之后的 J8Sample.class 还是使用 invokedynamic 指令，即这一步并没有 Desugar：



在 app:transformDexArchiveWithDexMergerForDebug（对应 dx）任务之后，再对应 build/intermediates/transforms/dexMerger 目录找第 0 个 classex.dex。

执行 $ANDROID_HOME/build-tools/28.0.3/dexdump -d classes.dex >> dexInfo.txt 拿到 dex 信息。

还是选取实例中 Lambda 表达式 1：Runnable runnable = () -> System.out.println("xixi"); 来进行分析。

这个 dexIno.txt 文件非常大，有 1.4M，我们通过 com.J8Smaple2.J8Sample 找到我们 J8Sample 在 dex 中位置。

```
Class #175            -
  Class descriptor  : 'Lcom/j8sample2/J8Sample;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
  Static fields     -
  Instance fields   -
  Direct methods    -
    #0              : (in Lcom/j8sample2/J8Sample;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
      registers     : 1
      ins           : 1
      outs          : 1
      insns size    : 4 16-bit code units
011bac:                                        |[011bac] com.j8sample2.J8Sample.<init>:()V
011bbc: 7010 e100 0000                         |0000: invoke-direct {v0}, Ljava/lang/Object;
011bc2: 0e00                                   |0003: return-void
```

新增方法：

```
|[011bc4] com.j8sample2.J8Sample.lambda$main$0:()V
|0000: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@154c
|0002: const-string v1, "xixi" // string@0829
|0004: invoke-virtual {v0, v1}, Ljava/io/PrintStream;.println:(Ljava/lang/String;)V
|0007: return-void
```

J8Sample.main 方法：

```
|[011c04] com.j8sample2.J8Sample.main:([Ljava/lang/String;)V
|0000: sget-object v0, Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc;.INSTANCE:L
|0002: new-instance v1, Ljava/lang/Thread; // type@00cd
|0004: invoke-direct {v1, v0}, Ljava/lang/Thread;.<init>:(Ljava/lang/Runnable;)V // method@00e2
|0007: invoke-virtual {v1}, Ljava/lang/Thread;.start:()V // method@00e3
|000a: new-instance v1, Ljava/lang/Thread; // type@00cd
|000c: sget-object v2, Lcom/j8sample2/-$$Lambda$J8Sample$WGbOO3DKt0AUOe3xQIFKehlVuRk;.INSTANCE:L
|000e: invoke-direct {v1, v2}, Ljava/lang/Thread;.<init>:(Ljava/lang/Runnable;)V // method@00e2
|0011: invoke-virtual {v1}, Ljava/lang/Thread;.start:()V // method@00e3
|0014: return-void
```

图中选中部分，对应就是 Lambda 表达式 1 desugar 之后的内容。

翻译成 Java 的话就变成了：new Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc 这个生成类的一个对象。

类 Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc 对应前面的生成的 J8Sample$$Lambda$1 类型，只不过数字 1 变成了 Hash 值。

```
Class #172            -
  Class descriptor  : 'Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc;'
  Access flags      : 0x1011 (PUBLIC FINAL SYNTHETIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
    #0              : 'Ljava/lang/Runnable;'
```

实现 Interface Ljava/lang/Runnable。Lcom/j8sample2/-$$Lambda$J8Sample$jWmuYH0zEF070TKXrjBFgnnqOKc.run 方法：

```
|[011b08] com.j8sample2.-..Lambda.J8Sample.jWmuYH0zEF070TKXrjBFgnnqOKc.run:()V
|0000: invoke-static {}, Lcom/j8sample2/J8Sample;.lambda$main$0:()V // method@00c9
|0003: return-void
```

到这里，是不是和前面 RetroLambda 就一样了。

总结

至此，Lambda 及其 invokedynamic 指令、RetroLambda 插件、D8 编译器各自的原理分析都已经结束了。

相比较 Lambda 在 Java8 自己内部的实现：即运行时，在内存中动态生成关联的函数式接口的实例类型，通过 BSM- 引导方法找到该内存类（字节码层面的反射）。

在 Android 上的其他三种 Desugar 方式，原理都是一样的，区别在于时机不同：

1. RetroLambda 将函数式接口对应的实例类型的生产过程，放在 javac 编译之后，dx 编译之前，并动态修改了表达式所属的字节码文件。
2. Jack&Jill 是直接将接口对应的实例类型，直接 jack 过程中生成，并编译进了 dex 文件。
3. D8 的过程是在 dex 编译过程中，直接在内存生成接口对应的实例类型，并将生成的类型直接写入生成的 dex 文件中。

探讨

无论是 RetroLambda，还是 D8，对 Java8 的特性也不是全都支持。

Java8 新增的许多 API（例如：新的 DataAPI），就 D8 编译器而言，只有在 Android P(9.0) 版本中能直接运行。低于 9.0 就不行了。如何能够全版本支持 Java 8。D8 还有很长的一段路要走。

如果我们在低版本需要使用新的 API，目前可以采取将这些 API 打包进去的临时办法。

写到这里，肯定有人要提出，为什么不直接使用 Kotlin 呢？确实 Kotlin 对 Lambda 表达式、函数引用等特性都做了很好的支持，但是现实的情况中，Kotlin 很难取代 Android 中的 Java。新业务、新工程还相对容易，对老业务来说，尤其是经过多年沉淀，工程结构复杂，迁移改造带来的收益，往往远远小于迁移改造带来的成本和不可控之风险。Kotlin 和 Java 同时存在的情况，长期来看是一个必然的结果。

至于 Java 8 的其他特性呢，D8 是如何实现的，也可以按照上面类似的方式去分析，甚至可以结合 Kotlin 实现的方式，一探究竟。

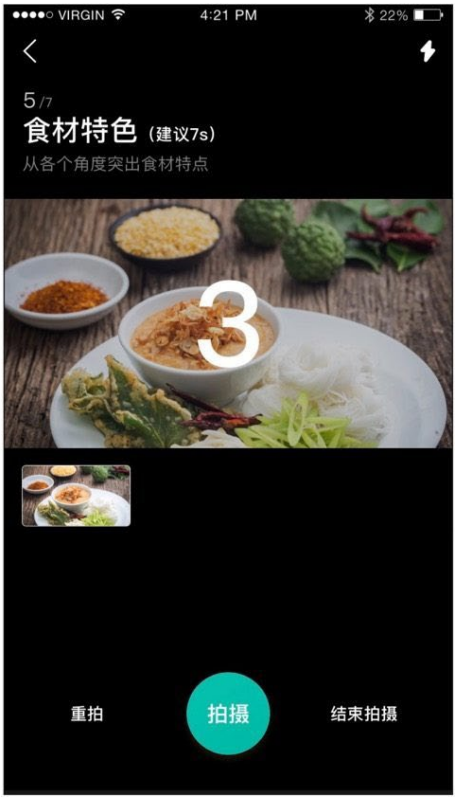
美团外卖商家端视频探索之旅

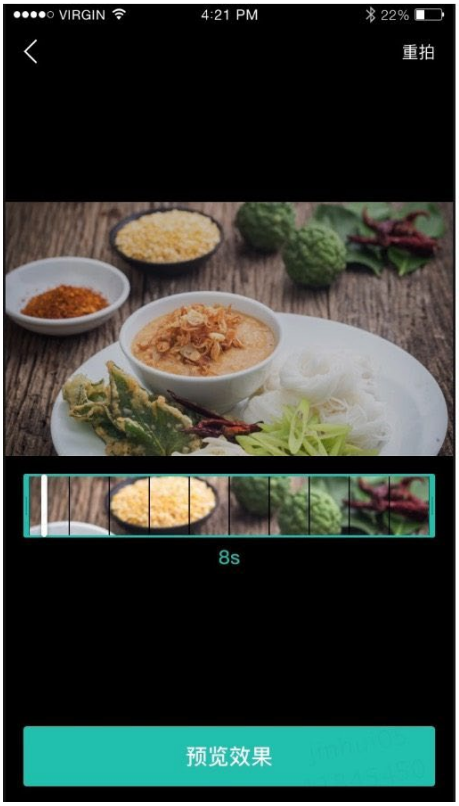
金辉　李琼

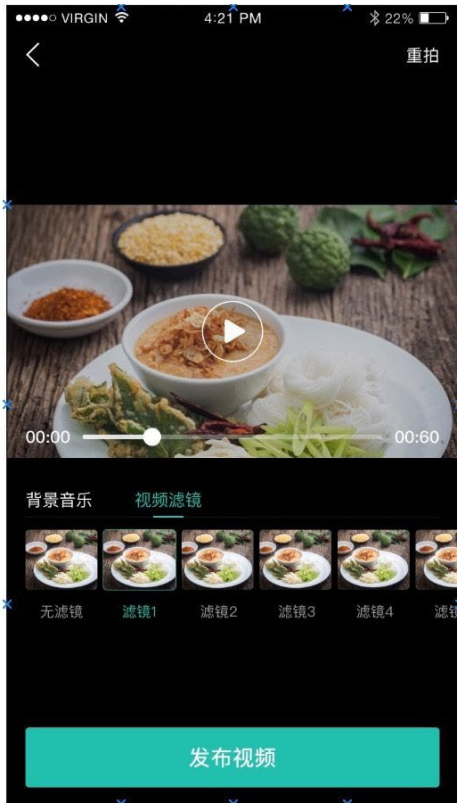
背景

美团外卖至今已迅猛发展了六年，随着外卖业务量级与日俱增，单一的文字和图片已无法满足商家的需求，商家迫切需要更丰富的商品描述手段吸引用户，增加流量，进而提高下单转化率和下单量。商品视频的引入，在一定程度上可以提升商品信息描述丰富度，以更加直观的方式为商家引流，增加收益。为此，商家端引入了视频功能，进行了一系列视频功能开发，核心功能包含视频处理（混音，滤镜，加水印，动画等）、视频拍摄、合成等，最终效果图如下所示：









自视频功能上线后，每周视频样本量及使用视频的商家量大幅增加，视频录制成功率达 99.533%，视频处理成功率 98.818%，音频处理成功率 99.959%，Crash 率稳定在 0.1‰，稳定性高且可用性强。目前，视频功能已在蜜蜂 App、闪购业务和商家业务上使用。

对于视频链路的开发，我们经历了方案选型、架构设计及优化、业务实践、功能测试、监控运维、更新维护等各个环节，核心环节如下图所示。在开发过程中，我们遇到了各种技术问题和挑战，下文会针对遇到的问题、挑战，及其解决方案进行重点阐述。



方案选型

在方案选型时，重点对核心流程和视频格式进行选型。我们以功能覆盖度、稳定性及效率、可定制性、成本及开源性做为核心指标，从而衡量方案的高可用性和可行性。

1. 核心流程选型

视频开发涉及的核心流程包括播放、录制、合成、裁剪、后期处理（编解码、滤镜、混音、动画、水印）等。结合商家端业务场景，我们有针对性的进行方案调研。重点调研了业界现有方案，如阿里的云视频点播方案、腾讯云视频点播方案、大众点评 App 的 UGC 方案，及其它的一些第三方开源方案等，并进行了整体匹配度的对比，如下图所示：

方案	能力	是否开源	收费	SDK大小	稳定性/性能	说明	结论
腾讯	丰富	否	10000/年	15M左右	高可靠，性能优 兼容性好	稳定性强，功能丰富，性能效率好，兼容性优	优点：软硬编码支持，稳定强，功能丰富 缺点：收费，定制化难度大
阿里	丰富	否	49000/年	20M左右	高可靠，性能优	功能丰富，稳定性强，性能效率好，安全性高	优点：性能优，安全性高 缺点：成本高，功能略臃肿，定制化难度大
点评方案	满足基本	否	否	集成方式待定	不保证	支持部分功能，业务场景有差异	优势：沟通协作良好可共建 缺点：业务场景差异大，部分功能不支持
其他开源	弱	是	否	待定	差	无完善的开源库，杂乱	开发代价太大，但可作参考

阿里和腾讯的云视频点播方案比较成熟，集成度高，且能力丰富，稳定性及效率也很高。但两者成本较高，需要收费，且 SDK 大小均在 15M 以上，对于我们的业务场景来说有些过于臃肿，定制性较弱，无法迅速的支持我们做定制性扩展。

当时的点评 App UGC 方案，基础能力是满足的，但因业务场景差异：

- 比如外卖的视频拍摄功能要求在竖屏下保证 16:9 的视频宽高比，这就需要对原有的采集区域进行截取，视频段落的裁剪支持不够等，业务场景的差异导致了实现方案存在巨大的差异，故放弃了点评 App UGC 方案。其他的一些开源方案（比如 Grafika 等），也无法满足要求，这里不再一一赘述。

通过技术调研和分析，吸取各开源项目的优点，并参考点评 App UGC、Google CTS 方案，对核心流程做了最终的方案选型，打造一个适合我们业务场景的方案，如下表所示：

核心能力	方案	效果
视频播放	ijkplayer+AndroidVideoCache	兼容性强 支持边缓存边播
视频录制	MediaRecorder+MediaCodec	采集区域的裁剪
视频合成	mp4parser	多文件拼接
视频裁剪	MediaCodec	误差微秒级
后期处理	MediaCodec，OpenGL，AAC	滤镜 动画 贴纸 混音等

2. 视频格式选型

文件格式	视频编码格式	视频分辨率	帧率	码率	音频压缩编码	音频采样率	音频码率
mp4	H.264	比例16：9	30fps	1200kb/s	AAC	44.1khz	128kb/s

- 采用 H.264 的视频协议：H.264 的标准成熟稳定，普及率高。其最大的优势是具有很高的数据压缩比率，在同等图像质量的条件下，H.264 的压缩比是 MPEG-2 的 2 倍以上，是 MPEG-4 的 1.5 ~ 2 倍。
- 采用 AAC 的音频协议：AAC 是一种专为声音数据设计的文件压缩格式。它采用了全新的算法进行编码，是新一代的音频有损压缩技术，具有更加高效，更具有”性价比“的特点。

整体架构

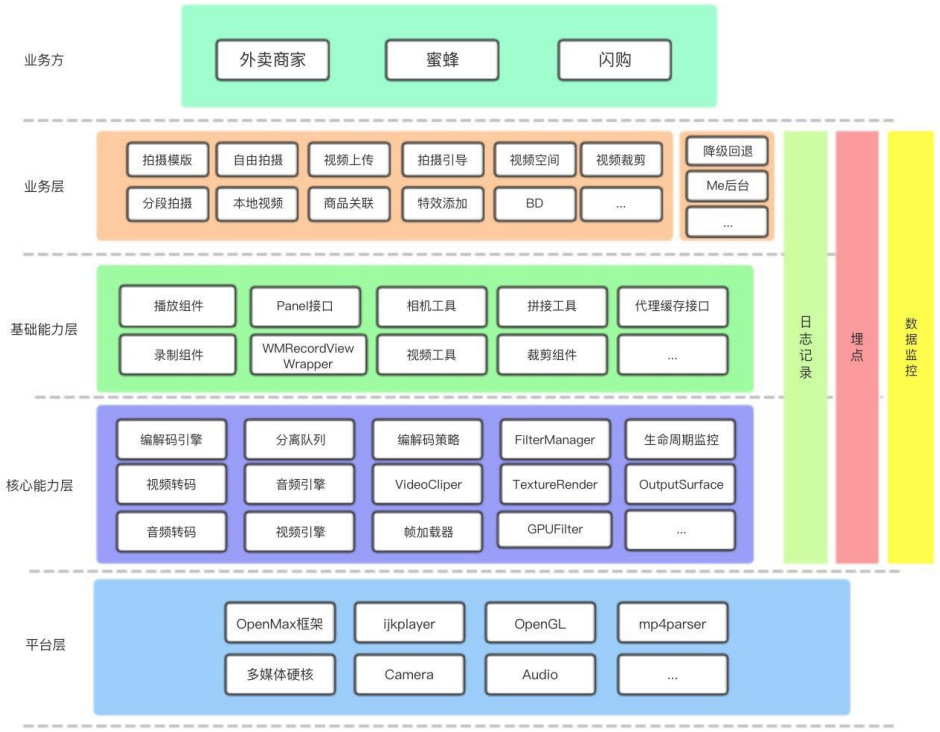
我们整体的架构设计，用以满足业务扩展和平台化需要，可复用、可扩展，且可快速接入。架构采用分层设计，基础能力和组件进行下沉，业务和视频能力做分离，最大化降低业务方的接入成本，三方业务只需要接入视频基础 SDK，直接使用相关能力组件或者工具即可。

整体架构分为四层，分别为平台层、核心能力层、基础组件层、业务层。

- **平台层**：依赖系统提供的平台能力，比如 Camera、OpenGL、MediaCodec 和 MediaMuxer 等，也包括引入的平台能力，比如 ijkplayer 播放器、mp4parser 。
- **核心能力层**：该层提供了视频服务的核心能力，包括音视频编解码、音视频的转码引擎、滤镜渲染能力等 。
- **基础能力层**：暴露了基础组件和能力，提供了播放、裁剪、录屏等基础组件和对应的基础工具类，并提供了可定制的播放面板，可定制的缓存接口等。
- **业务层**：包括段落拍摄、自由拍摄、视频空间、拍摄模版预览及加载等。

我们的视频能力层对业务层是透明的，业务层与能力层隔离，并对业务层提供了部分定制化的接口支持，这样的设计降低了业务方的接入成本，并方便业务方的

扩展，比如支持蜜蜂 App 的播放面板定制，还支持缓存策略、编解码策略的可定制。整体设计如下图所示：



实践经验

在视频开发实践中，因业务场景的复杂性，我们遇到了多种问题和挑战。下面以核心功能为基点，围绕各功能遇到的问题做详细介绍。

视频播放

播放器是视频播放基础。针对播放器，我们进行了一系列的方案调研和选择。在此环节，遇到的挑战如下：

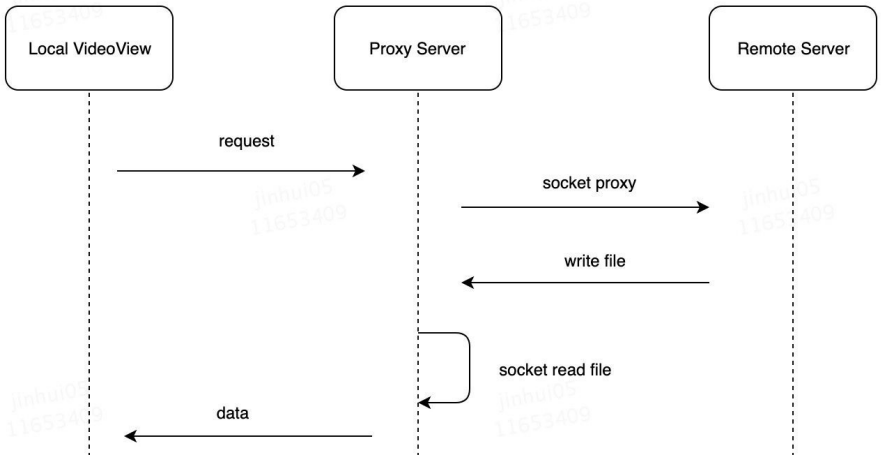
1. 兼容性问题

2. 缓存问题

针对兼容性问题，Android 有原生的 MediaPlayer，但其版本兼容问题偏多且支持格式有限，而我们需要支持播放本地视频，本地视频格式又无法控制，故该方案被舍弃。ijkplayer 基于 FFmpeg，与 MediaPlayer 相比，优点比较突出：具备跨平台能力，支持 Android 与 iOS；提供了类似 MediaPlayer 的 API，可兼容不同版本；可实现软硬解码自由切换，拥有 FFmpeg 的能力，支持多种流媒体协议。基于上述原因，我们最终决定选用 ijkplayer。

但紧接着我们又发现 ijkplayer 本身不支持边缓存边播放，频繁的加载视频导致耗费大量的流量，且在弱网或者 3G 网络下很容易导致播放卡顿，所以这里就衍生出了缓存的问题。

针对缓存问题，我们引入 AndroidVideoCache 的技术方案，利用本地的代理去请求数据，先本地保存文件缓存，客户端通过 Socket 读取本地的文件缓存进行视频播放，这样就做到了边播放边缓存的策略，流程如下图：



此外，我们还对 AndroidVideoCache 做了一些技术改造：

- 优化缓存策略。针对缓存策略的单一性，支持有限的最大文件数和文件大小问

题，我们调整为由业务方可以动态定制缓存策略；

- 解决内存泄露隐患。对其页面退出时请求不关闭会导致的内存泄露，我们为其添加了完整的生命周期监控，解决了内存泄露问题。

视频录制

在视频拍摄的时候，最为常用的方式是采用 MediaRecorder+Camera 技术，采集摄像头可见区域。但因我们的业务场景要求视频采集的时候，只录制采集区域的部分区域且比例保持宽高比 16:9，在保证预览图像不拉伸的情况下，只能对完整的采集区域做裁剪，这无形增加了开发难度和挑战。通过大量的资料分析，我们重点调研了有两种方案：

1. Camera+AudioRecord+MediaCodec+Surface
2. MediaRecorder+MediaCodec

方案 1 需要 Camera 采集 YUV 帧，进行截取采集，最后再将 YUV 帧和 PCM 帧进行编码生成 mp4 文件，虽然其效率高，但存在不可把控的风险。

方案 2 综合评估后是改造风险最小的。 综合成本和风险考量，我们保守的采用了方案 2，该方案是对裁剪区域进行坐标换算（如果用前置摄像头拍摄录制视频，会出现预览画面和录制的视频是镜像的问题，需要处理）。当录制完视频后，生成了 mp4 文件，用 MediaCodec 对其编码，在编码阶段再利用 OpenGL 做内容区域的裁剪来实现。但该方案又引发了如下挑战：

（1）对焦问题

因我们对采集区域做了裁剪，引发了点触对焦问题。比如用户点击了相机预览画面，正常情况下会触发相机的对焦动作，但是用户的点击区域只是预览画面的部分区域，这就导致了相机的对焦区域错乱，不能正常进行对焦。后期经过问题排查，对点触区域再次进行相应的坐标变换，最终得到正确的对焦区域。

（2）兼容适配

我们的视频录制利用 MediaRecorder，在获取配置信息时，由于 Android 碎片

化问题，不同的设备支持的配置信息不同，所以就会出现设备适配问题。

```
                        // VIVO Y66 模版拍摄时候，播放某些有问题的视频文件的同
时去录制视频，会导致 MediaServer 挂掉的问题
        // 发现将 1080P 尺寸的配置降低到 720P 即可避免此问题
        // 但是 720P 尺寸的配置下，又存在绿边问题，因此再降到 480
        if(isVIVOY66() && mMediaServerDied) {
            return getCamcorderProfile(CamcorderProfile.QUALITY_480P);
        }

        //SM-C9000，在 1280 x 720 分辨率时有一条绿边。网上有种说法是 GPU 对数据
进行了优化，
使得 GPU 产生的图像分辨率
        // 和常规分辨率存在微小差异，造成图像色彩混乱，修复后存在绿边问题。
        // 测试发现，降低分辨率或者升高分辨率都可以绕开这个问题。
        if (VideoAdapt.MODEL_SM_C9000.equals(Build.MODEL)) {
            return getCamcorderProfile(CamcorderProfile.QUALITY_HIGH);
        }

        // 优先选择 1080 P 的配置
        CamcorderProfile camcorderProfile =
getCamcorderProfile(CamcorderProfile.
QUALITY_1080P);
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_720P);
        }
        // 某些机型上这个 QUALITY_HIGH 有点问题，可能通过这个参数拿到的配置是
1080p，所
以这里也可能拿不到
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_HIGH);
        }
        // 兜底
        if (camcorderProfile == null) {
            camcorderProfile = getCamcorderProfile(CamcorderProfile.
QUALITY_480P);
        }
```

视频合成

我们的视频拍摄有段落拍摄这种场景，商家可根据事先下载的模板进行分段拍摄，最后会对每一段的视频做拼接，拼接成一个完整的 mp4 文件。mp4 由若干个 Box 组成，所有数据都封装在 Box 中，且 Box 可再包含 Box 的被称为 Container

Box。mp4 中 Track 表示一个视频或音频序列，是 Sample 的集合，而 Sample 又可分为 Video Smaple 和 Audio Sample。Video Smaple 代表一帧或一组连续视频帧，Audio Sample 即为一段连续的压缩音频数据。(详见 mp4 文件结构。)

基于上面的业务场景需要，视频合成的基础能力我们采用 mp4parser 技术实现(也可用 FFmpeg 等其他手段)。mp4parser 在拼接视频时，先将视频的音轨和视频轨进行分离，然后进行视频和音频轨的追加，最终将合成后的视频轨和音频轨放入容器里(这里的容器就是 mp4 的 Box)。采用 mp4parser 技术简单高效，API 设计简洁清晰，满足需求。

但我们发现某些被编码或处理过的 mp4 文件可能会存在特殊的 Box，并且 mp4parser 是不支持的。经过源码分析和原因推导，发现当遇到这种特殊格式的 Box 时，会申请分配一个比较大的空间用来存放数据，很容易造成 OOM(内存溢出)，见下图所示。于是，我们对这种拼接场景下做了有效规避，仅在段落拍摄下使用 mp4parser 的拼接功能，保证我们处理过的文件不会包含这种特殊的 Box。

```
java.lang.OutOfMemoryError: Failed to allocate a 671088632 byte allocation with 11496528 free bytes and 501MB until OOM, max

StackTrace:
,java.nio.HeapByteBuffer.<init>(HeapByteBuffer.java:54)
java.nio.HeapByteBuffer.<init>(HeapByteBuffer.java:49)
java.nio.ByteBuffer.allocate(ByteBuffer.java:261)
com.googlecode.mp4parser.AbstractBox.parse(AbstractBox.java:110)
com.coremedia.iso.AbstractBoxParser.parseBox(AbstractBoxParser.java:107)
com.googlecode.mp4parser.BasicContainer.next(BasicContainer.java:185)
com.googlecode.mp4parser.BasicContainer.hasNext(BasicContainer.java:161)
com.googlecode.mp4parser.util.LazyList.blowup(LazyList.java:30)
com.googlecode.mp4parser.util.LazyList.size(LazyList.java:77)
com.googlecode.mp4parser.BasicContainer.getBoxes(BasicContainer.java:80)
com.googlecode.mp4parser.authoring.samples.DefaultMp4SampleList.<init>(DefaultMp4SampleList.java:36)
com.coremedia.iso.boxes.mdat.SampleList.<init>(SampleList.java:33)
com.googlecode.mp4parser.authoring.Mp4TrackImpl.<init>(Mp4TrackImpl.java:64)
com.googlecode.mp4parser.authoring.container.mp4.MovieCreator.build(MovieCreator.java:57)
com.sankuai.meituan.video.utils.VideoUtils.composeVideo(VideoUtils.java:453)
com.sankuai.meituan.video.audio.AudioManager.composeAudio(AudioManager.java:273)
com.sankuai.meituan.video.audio.AudioManager.addAudio(AudioManager.java:60)
```

视频裁剪

我们刚开始采用 mp4parser 技术完成视频裁剪，在实践中发现其精度误差存在很大的问题，甚至会影响正常的业务需求。比如我们禁止裁剪出 3s 以下的视频，但是由于 mp4parser 产生的精度误差，导致 4-5s 的视频很容易裁剪出少于 3s 的视频。究其原因，mp4parser 只能在关键帧(又称 I 帧，在视频编码中是一种自带全部信息的独立帧)进行切割，这样就可能存在一些问题。比如在视频截取的起始

时间位置并不是关键帧，因此会造成误差，无法保证精度而且是秒级误差。以下为 mp4parser 裁剪的关键代码：

```
public static double correctTimeToSyncSample(Track track, double
cutHere, boolean next) {
        double[] timeOfSyncSamples = new double[track.getSyncSamples().
length];
        long currentSample = 0;
        double currentTime = 0;
        for (int i = 0; i < track.getSampleDurations().length; i++) {
            long delta = track.getSampleDurations()[i];
            int index = Arrays.binarySearch(track.getSyncSamples(),
currentSample + 1);
            if (index >= 0) {
                timeOfSyncSamples[index] = currentTime;
            }
            currentTime += ((double) delta / (double) track.
getTrackMetaData().getTimescale());
            currentSample++;
        }
        double previous = 0;
        for (double timeOfSyncSample : timeOfSyncSamples) {
            if (timeOfSyncSample > cutHere) {
                if (next) {
                    return timeOfSyncSample;
                } else {
                    return previous;
                }
            }
            previous = timeOfSyncSample;
        }
        return timeOfSyncSamples[timeOfSyncSamples.length - 1];
}
```

为了解决精度问题，我们废弃了 mp4parser，采用 MediaCodec 的方案，虽然该方案会增加复杂度，但是误差精度大大降低。

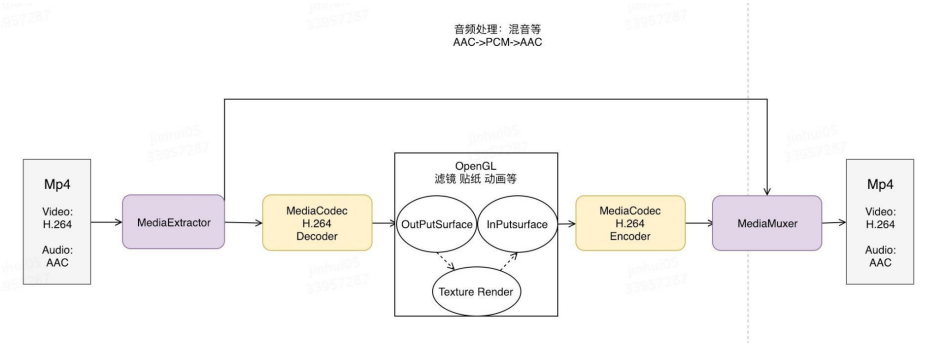
方案具体实施如下：先获得目标时间的上一帧信息，对视频解码，然后根据起始时间和截取时长进行切割，最后将裁剪后的音视频信息进行压缩编码，再封装进 mp4 容器中，这样我们的裁剪精度从秒级误差降低到微秒级误差，大大提高了容错率。

视频处理

视频处理是整个视频能力最核心的部分，会涉及硬编解码（遵循 OpenMAX 框

架)、OpenGL、音频处理等相关能力。

下图是视频处理的核心流程，会先将音视频做分离，并行处理音视频的编解码，并加入特效处理，最后合成进一个 mp4 文件中。



在实践过程中，我们遇到了一些需要特别注意的问题，比如开发时遇到的坑，严重的兼容性问题（包括硬件兼容性和系统版本兼容性问题）等。下面重点讲几个有代表性的问题。

1. 偶数宽高的编解码器

视频经过编码后输出特定宽高的视频文件时出现了如下错误，信息里仅提示了 Colorformat 错误，具体如下：

```
19624-19828 I/OMXClient: Treble IOmx obtained
19624-19820 I/VideoTrackTranscoder: mEncoder.name = OMX.qcom.video.encoder.avc
19624-19820 I/VideoTrackTranscoder: mOutputFormat :  {color-format=39, i-frame-interval=10, mime=video/avc, width=1080, bitrate=358
19624-19828 W/OMXUtils: do not know color format 0x7fa30c06 = 2141391878
19624-19828 W/OMXUtils: do not know color format 0x7fa30c04 = 2141391876
    do not know color format 0x7fa30c08 = 2141391880
    do not know color format 0x7fa30c07 = 2141391879
19624-19828 W/OMXUtils: do not know color format 0x7f000789 = 2130708361
19624-19828 E/ACodec: [OMX.qcom.video.encoder.avc] does not support color format 39
    [OMX.qcom.video.encoder.avc] configureCodec returning error -61
    signalError(omxError 0x80001001, internalError -61)
19624-19827 E/MediaCodec: Codec reported err 0xffffffc3, actionCode 0, while in state 3
19624-19820 E/MediaCodec: configure failed with err 0xffffffc3, resetting...
19624-19828 I/OMXClient: Treble IOmx obtained
19624-19820 W/System.err: android.media.MediaCodec$CodecException: Error 0xffffffc3
        at android.media.MediaCodec.native_configure(Native Method)
        at android.media.MediaCodec.configure(MediaCodec.java:1943)
        at android.media.MediaCodec.configure(MediaCodec.java:1872)
        at com.dianping.video.videofilter.transcoder.engine.VideoTrackTranscoder.setup(VideoTrackTranscoder.java:104)
```

查阅大量资料，也没能解释清楚这个异常的存在。基于日志错误信息，并通过系统源码定位，也只是发现了是和设置的参数不兼容导致的。经过反复的试错，最后确认是部分编解码器只支持偶数的视频宽高，所以我们对视频的宽高做了偶数限制。引

起该问题的核心代码如下：

```
status_t ACodec::setupVideoEncoder(const char *mime, const sp<AMessage> &msg,
        sp<AMessage> &outputFormat, sp<AMessage> &inputFormat) {
    if (!msg->findInt32("color-format", &tmp)) {
        return INVALID_OPERATION;
    }
    OMX_COLOR_FORMATTYPE colorFormat =
        static_cast<OMX_COLOR_FORMATTYPE>(tmp);
    status_t err = setVideoPortFormatType(
            kPortIndexInput, OMX_VIDEO_CodingUnused, colorFormat);
    if (err != OK) {
        ALOGE("[%s] does not support color format %d",
              mComponentName.c_str(), colorFormat);
        return err;
    }
    .......
}
status_t ACodec::setVideoPortFormatType(OMX_U32 portIndex,OMX_VIDEO_
CODINGTYPE
compressionFormat,
        OMX_COLOR_FORMATTYPE colorFormat,bool usingNativeBuffers) {
    ......
    for (OMX_U32 index = 0; index <= kMaxIndicesToCheck; ++index) {
        format.nIndex = index;
        status_t err = mOMX->getParameter(
                mNode, OMX_IndexParamVideoPortFormat,
                &format, sizeof(format));
        if (err != OK) {
            return err;
        }
     ......
}
```

2. 颜色格式

我们在处理视频帧的时候，一开始获得的是从 Camera 读取到的基本的 YUV 格式数据，如果给编码器设置 YUV 帧格式，需要考虑 YUV 的颜色格式。这是因为 YUV 根据其采样比例，UV 分量的排列顺序有很多种不同的颜色格式，Android 也支持不同的 YUV 格式，如果颜色格式不对，会导致花屏等问题。

3. 16 位对齐

这也是硬编码中老生常谈的问题了，因为 H264 编码需要 16*16 的编码块大小。

如果一开始设置输出的视频宽高没有进行 16 字节对齐，在某些设备（华为，三星等）就会出现绿边，或者花屏。

4. 二次渲染

4.1 视频旋转

在最后的视频处理阶段，用户可以实时的看到加滤镜后的视频效果。这就需要对原始的视频帧进行二次处理，然后在播放器的 Surface 上渲染。首先我们需要 OpenGL 的渲染环境（通过 OpenGL 的固有流程创建），渲染环境完成后就可以对视频的帧数据进行二次处理了。通过 SurfaceTexture 的 updateTexImage 接口，可将视频流中最新的帧数据更新到对应的 GL 纹理，再操作 GL 纹理进行滤镜、动画等处理。在处理视频帧数据的时候，首先遇到的是角度问题。在正常播放下（不利用 OpenGL 处理情况下）通过设置 TextureView 的角度（和视频的角度做转换）就可以解决，但是加了滤镜后这一方案就失效了。原因是视频的原始数据经过纹理处理再渲染到 Surface 上，单纯设置 TextureView 的角度就失效了，解决方案就是对 OpenGL 传入的纹理坐标做相应的旋转（依据视频的本身的角度）。

4.2 渲染停滞

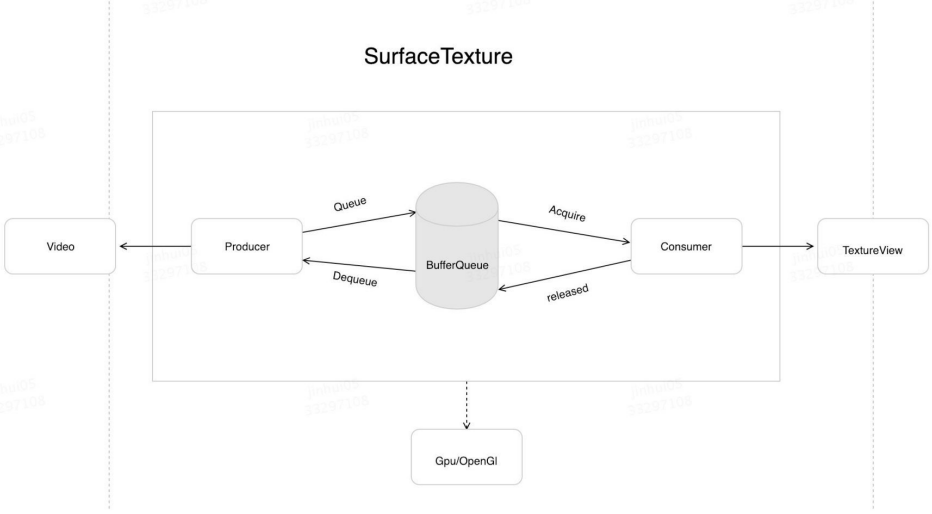
视频在二次渲染后会出现偶现的画面停滞现象，主要是 SurfaceTexture 的 OnFrameAvailableListener 不返回数据了。该问题的根本原因是 GPU 的渲染和视频帧的读取不同步，进而导致 SurfaceTexture 的底层核心 BufferQueue 读取 Buffer 出了问题。下面我们通过 BufferQueue 的机制和核心源码深入研究下：

首先从二次渲染的工作流程入手。从图像流（来自 Camera 预览、视频解码、GL 绘制场景等）中获得帧数据，此时 OnFrameAvailableListener 会回调。再调用 updateTexImage()，会根据内容流中最近的图像更新 SurfaceTexture 对应的 GL 纹理对象。我们再对纹理对象做处理，比如添加滤镜等效果。SurfaceTexture 底层核心管理者是 BufferQueue，本身基于生产者消费者模式。

BufferQueue 管理的 Buffer 状态分为：FREE，DEQUEUED，QUEUED，ACQUIRED，SHARED。当 Producer 需要填充数据时，需要先 Dequeue 一

个 Free 状态的 Buffer，此时 Buffer 的状态为 DEQUEUED，成功后持有者为 Producer。随后 Producer 填充数据完毕后，进行 Queue 操作，Buffer 状态流转为 QUEUED，且 Owner 变为 BufferQueue，同时会回调 BufferQueue 持有的 ConsumerListener 的 onFrameAvailable，进而通知 Consumer 可对数据进行二次处理了。Consumer 先通过 Acquire 操作，获取处于 QUEUED 状态的 Buffer，此时 Owner 为 Consumer。当 Consumer 消费完 Buffer 后，会执行 Release，该 Buffer 会流转回 BufferQueue 以便重用。BufferQueue 核心数据为 GraphicBuffer，而 GraphicBuffer 会根据场景、申请的内存大小、申请方式等的不同而有所不同。

SurfaceTexture 的核心流程如下图：



通过上图可知，我们的 Producer 是 Video，填充视频帧后，再对纹理进行特效处理（滤镜等），最后再渲染出来。前面我们分析了 BufferQueue 的工作流程，但是在 Producer 要填充数据，执行 dequeueBuffer 操作时，如果有 Buffer 已经 QUEUED，且申请的 dequeuedCount 大于 mMaxDequeuedBufferCount，就不会再继续申请 Free Buffer 了，Producer 就无法 DequeueBuffer，也就导致 onFrameAvailable 无法最终调用，核心源码如下：

```
status_t BufferQueueProducer::dequeueBuffer(int
*outSlot,sp<android::Fence> *outFence,
uint32_t width, uint32_t height,
       PixelFormat format, uint32_t usage,FrameEventHistoryDelta*
outTimestamps) {
       ......
       int found = BufferItem::INVALID_BUFFER_SLOT;
       while (found == BufferItem::INVALID_BUFFER_SLOT) {
            status_t status =
waitForFreeSlotThenRelock(FreeSlotCaller::Dequeue,
                      & found);
            if (status != NO_ERROR) {
                return status;
            }
        }
        ......
}
status_t BufferQueueProducer::waitForFreeSlotThenRelock(FreeSlotCaller
caller,
                    int*found) const{
        ......
        while (tryAgain) {
            int dequeuedCount = 0;
            int acquiredCount = 0;
            for (int s : mCore -> mActiveBuffers) {
                if (mSlots[s].mBufferState.isDequeued()) {
                    ++dequeuedCount;
                }
                if (mSlots[s].mBufferState.isAcquired()) {
                    ++acquiredCount;
                }
            }
            // Producers are not allowed to dequeue more than
            // mMaxDequeuedBufferCount buffers.
            // This check is only done if a buffer has already been
queued
            if (mCore -> mBufferHasBeenQueued &&
                    dequeuedCount >= mCore -> mMaxDequeuedBufferCount) {
                BQ_LOGE("%s: attempting to exceed the max dequeued
buffer count "
                        "(%d)", callerString, mCore ->
mMaxDequeuedBufferCount);
                return INVALID_OPERATION;
            }
        }
        .......
 }
```

5. 码流适配

视频的监控体系发现，Android 9.0 的系统出现大量的编解码失败问题，错误信息都是相同的。在 MediaCodec 的 Configure 时候出异常了，主要原因是我们强制使用了 CQ 码流，Android 9.0 以前并无问题，但 9.0 及以后对 CQ 码流增加了新的校验机制而我们没有适配。核心流程代码如下：

```
status_t ACodec::configureCodec(
       const char *mime, const sp<AMessage> &msg) {
      .......
      if (encoder) {
        if (mIsVideo || mIsImage) {
          if (!findVideoBitrateControlInfo(msg, &bitrateMode, &bitrate,
&quality)) {
               return INVALID_OPERATION;
           }
      } else if (strcasecmp(mime, MEDIA_MIMETYPE_AUDIO_FLAC)
          && !msg->findInt32("bitrate", &bitrate)) {
         return INVALID_OPERATION;
      }
   }
   .......
}
static bool findVideoBitrateControlInfo(const sp<AMessage> &msg,
        OMX_VIDEO_CONTROLRATETYPE *mode, int32_t *bitrate, int32_t
*quality) {
    *mode = getVideoBitrateMode(msg);
    bool isCQ = (*mode == OMX_Video_ControlRateConstantQuality);
    return (!isCQ && msg->findInt32("bitrate", bitrate))
         || (isCQ && msg->findInt32("quality", quality));
}
9.0 前并无对 CQ 码流的强校验，如果不支持该码流也会使用默认支持的码流，
static OMX_VIDEO_CONTROLRATETYPE getBitrateMode(const sp<AMessage>
&msg) {
    int32_t tmp;
    if (!msg->findInt32("bitrate-mode", &tmp)) {
        return OMX_Video_ControlRateVariable;
    }
    return static_cast<OMX_VIDEO_CONTROLRATETYPE>(tmp);
}
```

关于码流还有个问题就是如果通过系统的接口 isBitrateModeSupported（int mode），判断是否支持该码流可能会出现误判，究其原因是 framework 层写死了该返回值，而并没有从硬件层或从 media_codecs.xml 去获取该值。关于码流各硬件厂商

支持的差异性，可能谷歌也认为码流的兼容性太碎片化，不建议用非默认的码流。

6. 音频处理

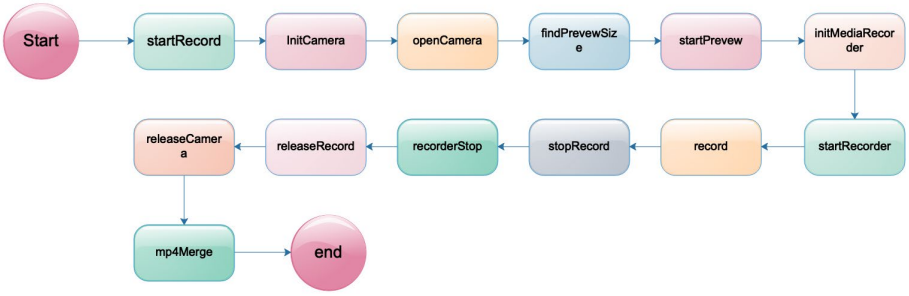
音频处理还括对音频的混音，消声等操作。在混音操作的时候，还要注意音频文件的单声道转换等问题。

其实视频问题总结起来，大部分是都会牵扯到编解码（尤其是使用硬编码），需要大量的适配工作（以上也只是部分问题，碎片化还是很严峻的），所以就需要兜底容错方案，比如加入软编。

线上监控

视频功能引入了埋点，日志，链路监控等技术手段进行线上的监控，我们可以针对监控结果进行降级或维护更新。埋点更多的是产品维度的数据收集，日志是辅助定位问题的，而链路监控则可以做到监控预警。我们加了拍摄流程，音视频处理，视频上传流程的全链路监控，整个链路如果任何一个节点出问题都认为是整个链路的失败，若失败次数超过阈值就会通过大象或邮件进行报警，我们在适配 Andorid 9.0 码流问题时，最早发现也是由于链路监控的预警。所有全链路的成功率目标值均为 98%，若成功率低于 92% 的目标阈值就会触发报警，我们会根据报警的信息和日志定位分析，该异常的影响范围，再根据影响范围确定是否热修复或者降级。如下以拍摄流程为例，其链路各核心节点的监控：

拍摄流程全链路，如下图（各关键节点监控）：



容灾降级

视频功能目前只支持粗粒度的降级策略。我们在视频入口处做了开关控制，关掉后所有的视频功能都无法使用。我们通过线上监控到视频的稳定性和成功率在特定机型无法保证，导致影响用户正常的使用商家端 App，我们支持针对特定设备做降级。后续我们可以做更细粒度的降级策略，比如根据 P0 级功能做降级，或者编解码策略的降级等

维护更新

视频功能上线后，经历了几个稳定的版本，保持着较高的成功率，但近期收到了 sniffer 的邮件报警，发现视频处理链路的失败次数明显增多，通过 sniffer 收集的信息发现大部分都是 Android 9.0 的问题（也就是上面讲的 Android 9.0 码流适配的问题），我们在商家端 5.2 版本进行了修复，该问题解决后我们的视频处理链路成功率也恢复到了 98% 以上。

总结和规划

视频功能上线后，稳定性、内存、CPU 等一些相关指标数据比较理想，我们建设的视频监控体系，也支撑着视频核心业务的监控，一些异常报警也让我们及时发现问题并迅速对异常进行维护更新，但视频技术栈也是远比本文介绍的要庞大，怎么提高秒播率，怎么提高编解码效率，还有硬编解码过程中可能造成的花屏，绿边等问题都是挑战，需要更深入的研究解决。

未来我们会继续致力于提高视频处理的兼容性和效率，优化现有流程，我们会对音频和视频处理合并处理，也会引入软编和自定义编解码算法。

美团外卖大前端团队将来也会继续致力于提高用户的体验，并且会将在实践过程中遇到的问题进行总结，沉底技术，积极的和大家分享，如果你也对视频感兴趣，欢迎加入我们。

参考资料

1. Android 开发者官网
2. Google CTS
3. Grafika
4. BufferQueue 原理介绍
5. MediaCodec 原理
6. 微信 Android 视频编码爬过的坑
7. mp4 文件结构
8. AndroidVideoCache 代理策略
9. ijkplayer
10. mp4parser
11. GPUImage

作者简介

金辉、**李琼**，美团外卖商家终端研发工程师。

招聘信息

美团外卖商家终端研发团队的主要职责是为商家提供稳定可靠的生产经营工具，在保障稳定的需求迭代的基础之上，持续优化 APP、PC 和 H5 的性能和用户体验，并不断优化提升团队的研发效率。团队主要负责的业务主要包括外卖订单、商品管理、门店装修、服务市场、门店运营、三方会话、蓝牙打印、自动接单、视频、语音和实时消息触达等基础业务，支撑整个外卖链路的高可用性及稳定发展。

团队通过架构演进及平台化体系化建设，有效支撑业务发展，提升了业务的可靠性和安全性；通过大规模落地跨平台和动态化技术，加快了业务迭代效率，帮助产品（PM）加快产品方案的落地及上线；通过监控容灾体系建设，有效保障业务的高可用性和稳定性；通过性能优化建设，保证 APP 的流畅性和良好用户体验。团队开发的技术栈包括 Android、iOS、React、Flutter 和 React Native。




虎年大吉