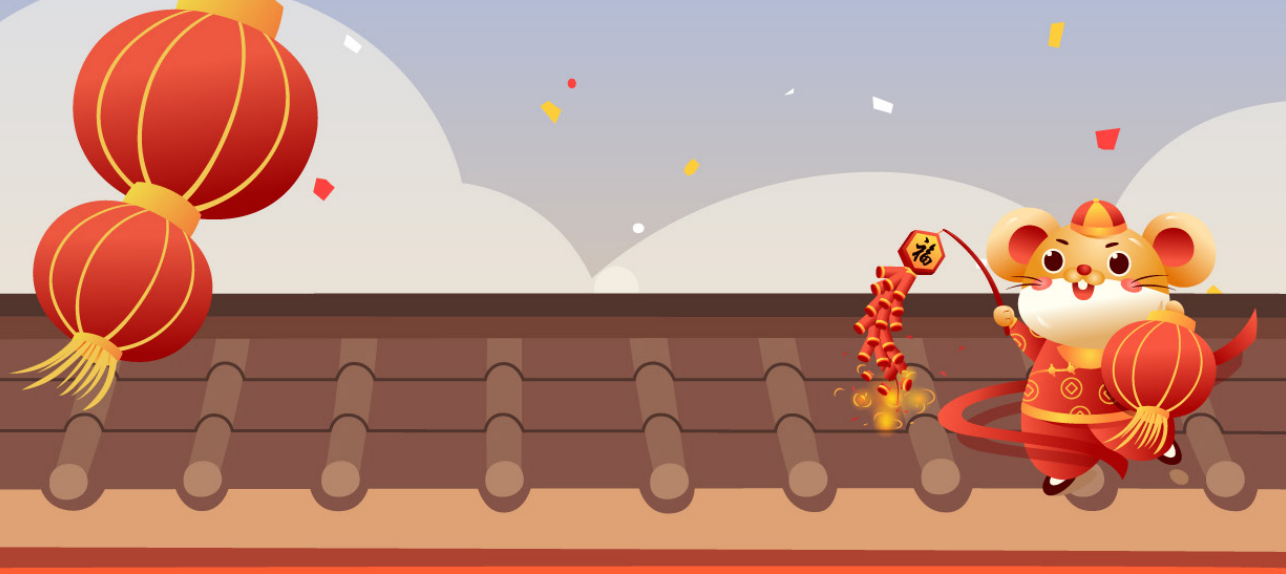


# 美团点评
# 2019 技术年货

CODE A BETTER LIFE

【前端篇】

美团点评


微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐

# 目录

# 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！

# 动态化

# 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。

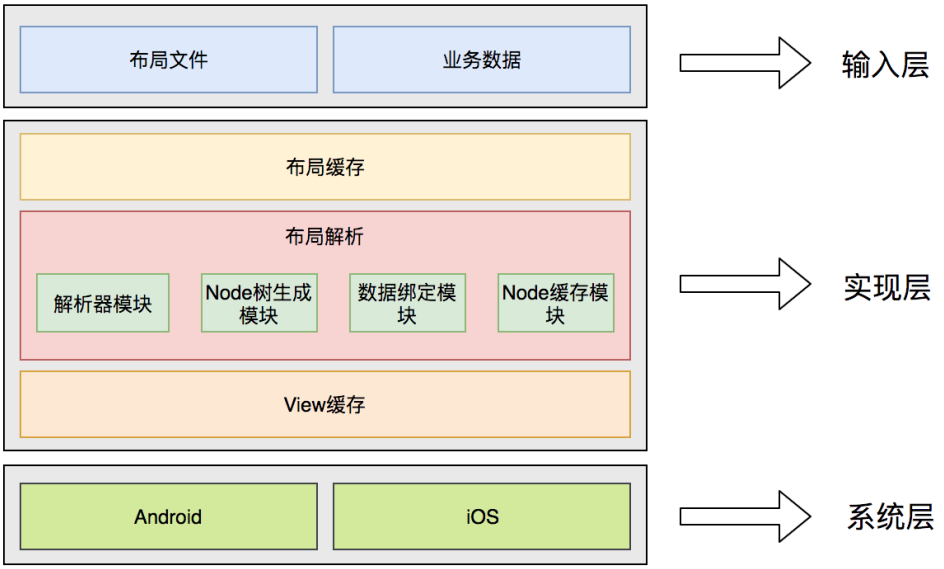
目前，MTFlexbox 已经广泛应用于美团首页、搜索、外卖等多个业务场景，并且已稳定运行两年有余。在 MTFlexbox 规范下，只需要写一份布局文件，就可以适用多端。在实际开发中，客户端开发同学开发布局的同时也要添加好埋点信息，帮助产品同学来评估上线后的效果。但现有布局埋点存在成本过高、准确率较低等痛点，为了解决这些问题，我们充分了解数据组开发人员和产品对数据统计的诉求，结合对 MTFlexbox 原理的深入理解，围绕 MTFlexbox 的埋点上报做了很多持续、有针对性的自动化工作，帮助多个项目的效率得到了显著提升。本文主要介绍美团在 MTFlexbox 自动化埋点方向所进行的一些探索，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：

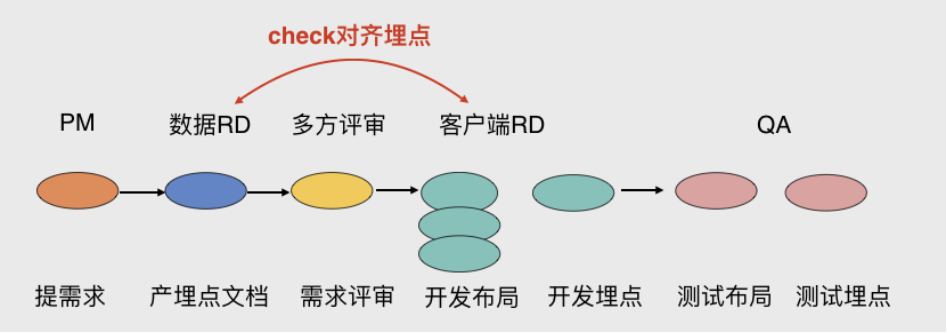
| 属性名 | 埋点类型 |
| --- | --- |
| click-mge2-report | 点击2.5埋点 |
| click-mge4-report | 点击4.0埋点 |
| click-tag-report | 点击Tag埋点 |
| load-meg2-report | 加载2.5埋点 |
| load-meg4-report | 加载4.0埋点 |
| see-mge2-report | 看见2.5埋点 |
| see-mge4-report | 看见4.0埋点 |

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。

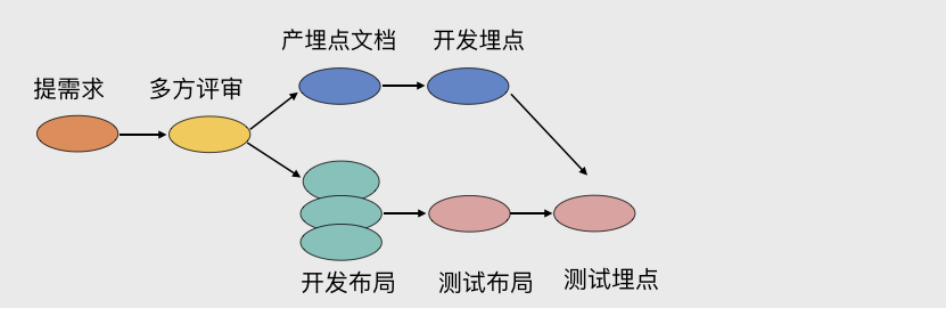
很多时候，一次沟通不到位，还需要反复沟通或者重新沟通，直到产品、数据同学和客户端开发人员三方对需求和埋点的理解一致为止。平均一个 5PD（5PD 指 5 个工作日）的需求需要消耗数据同学和客户端开发人员各 1PD 的时间来进行理解和沟通。

- 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

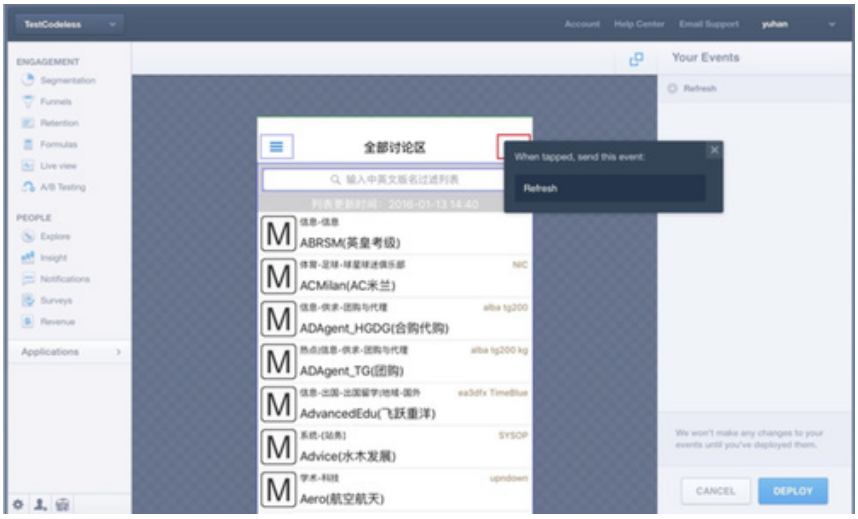
## 业内自动化埋点方案调研与参考

### 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。

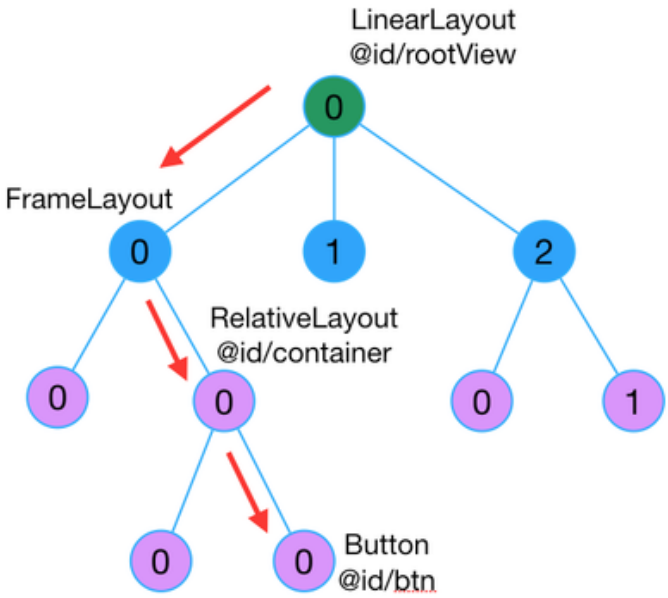
### Mixpanel

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。



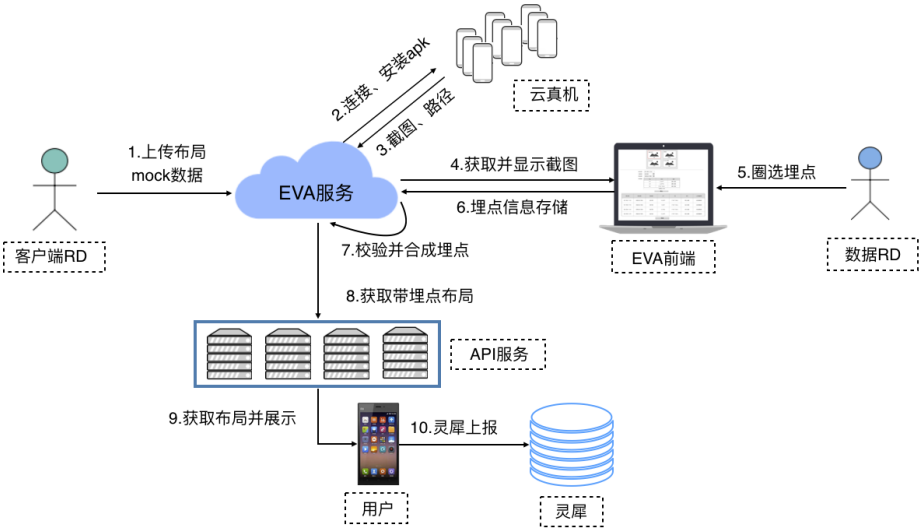

网易对 XPath 做了优化，主要体现在 View 索引的计算上：

- 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" : "ssssss",
            "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：

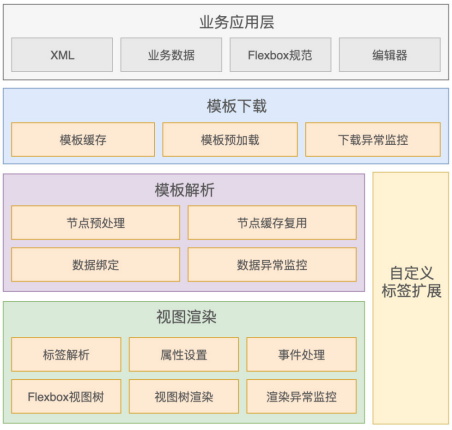



图 1　MTFlexbox 的架构

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。

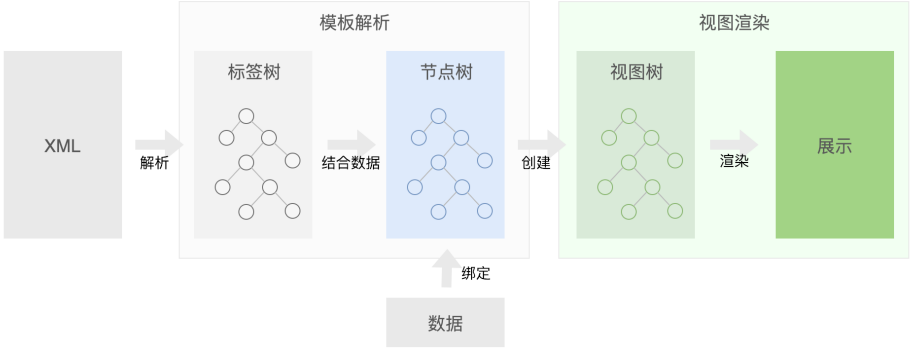



图 2　视图模版从解析到渲染

## 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：

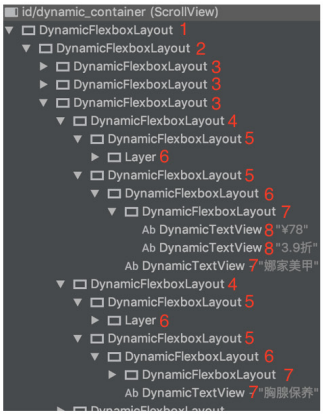
- 复杂视图因层级过深，导致滑动卡顿问题。
- 生成视图耗时过长，导致滑动卡顿问题。

### 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。

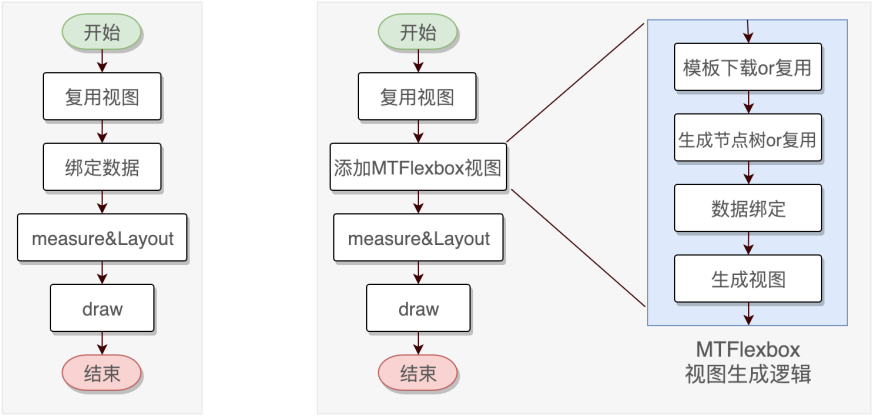



图 4　视图生成耗时原因分析

### 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。

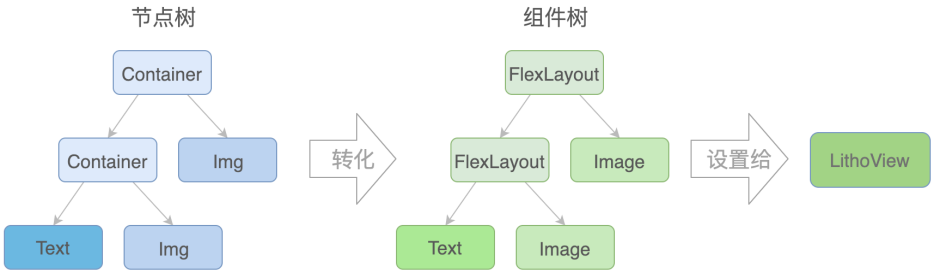



图 5　Litho 视图引擎从节点到视图的转换

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。

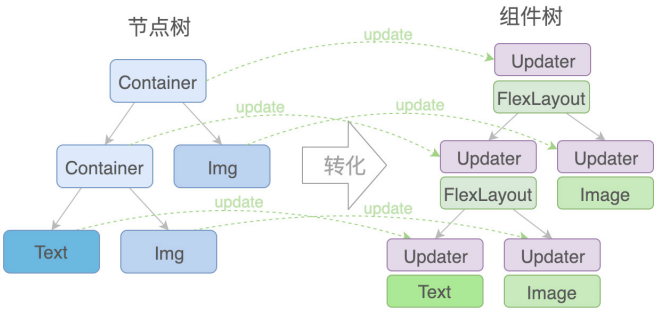



图 6　数据更新问题初版解决方案

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：

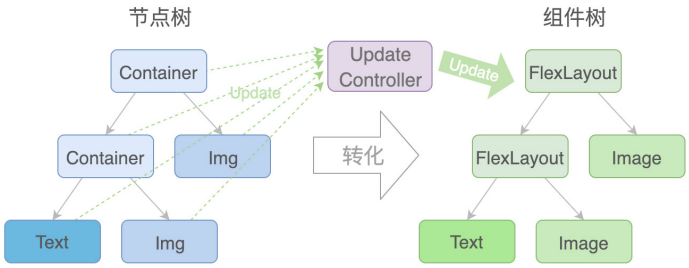



图 7　数据更新问题最终解决方案

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。

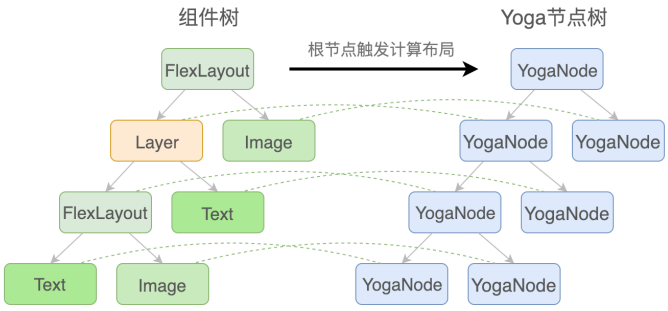



图 8　Litho 的布局计算原理

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。

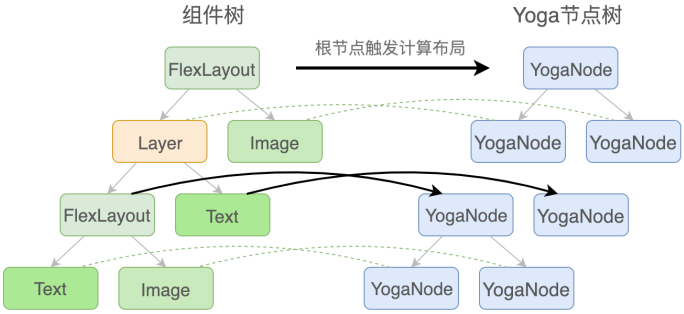



图 9　Layer 布局的实现原理

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。

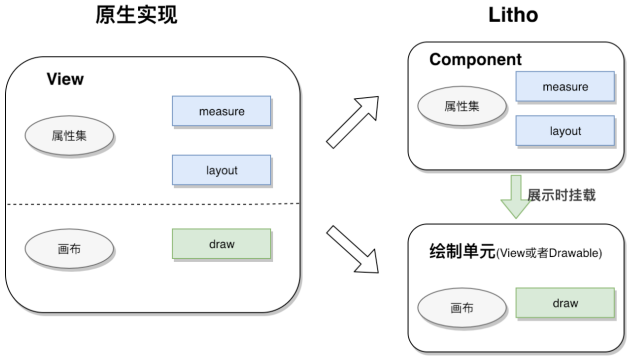


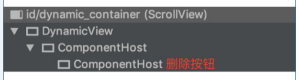

图 10　Litho 对 View 功能的拆分

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

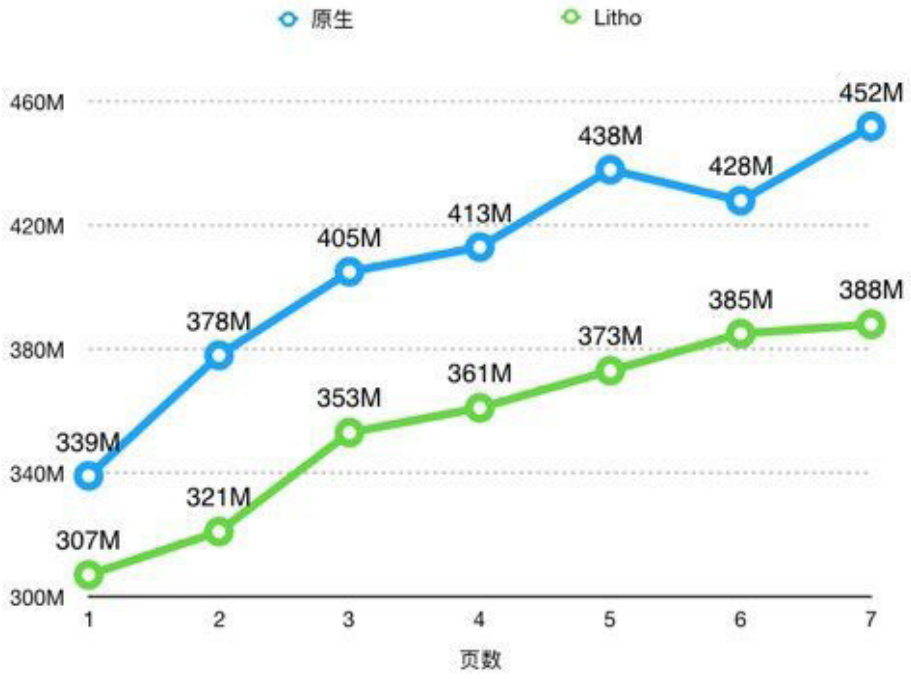
布局计算的能力放在了组件里面，每一种组件对应一个绘制单元来专门负责绘制。那么对于使用方扩展的标签，我们可以定义一个通用组件来统一承接。在挂载绘制单元时，再去调用使用方扩展的视图去绘制。

### 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。

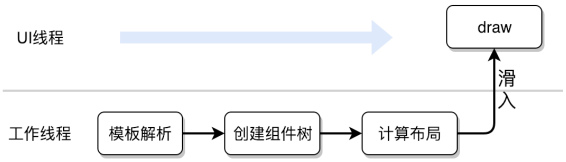



### 解决问题二：生成视图耗时过长

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

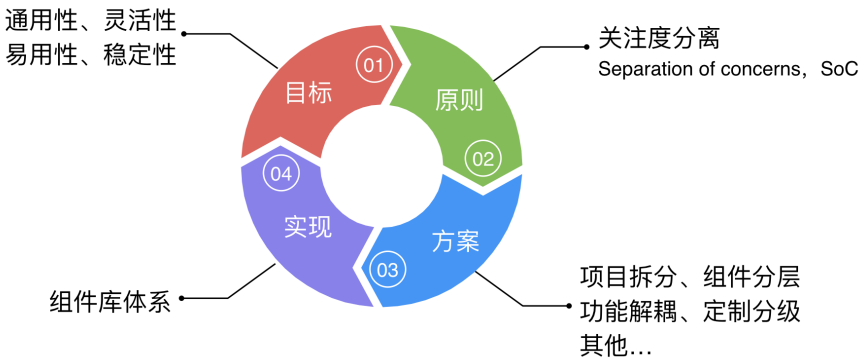


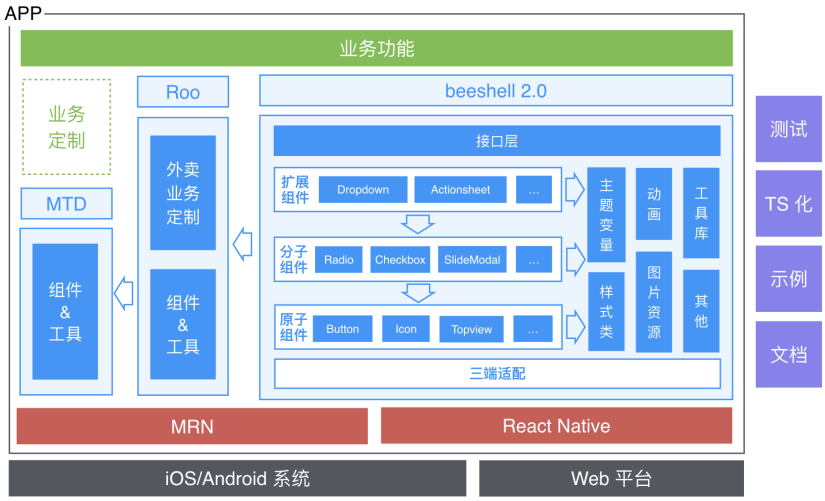

图 H

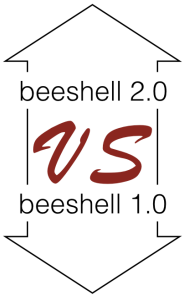
我们的目标是提供一套通用、易用、稳定、灵活的组件。然而，对一个组件来说，如果通用性、灵活性强，则易用性、稳定性势必较差。如何合理的处理这个矛盾呢?

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
MTD
组件&工具
Roo
外卖业务定制
组件&工具
beeshell 2.0
接口层
扩展组件
Dropdown
Actionsheet
...
分子组件
Radio
Checkbox
SlideModal
...
原子组件
Button
Icon
Topview
...
主题变量
样式类
动画
图片资源
工具库
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

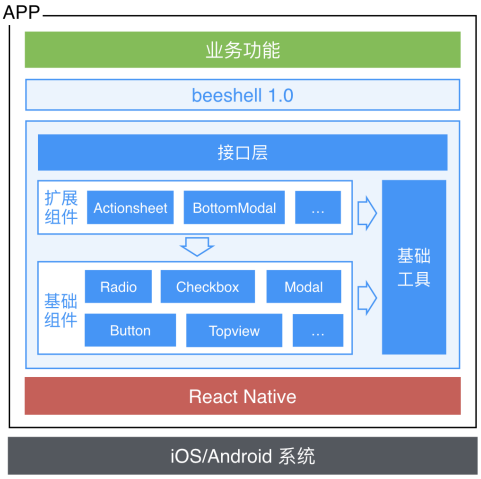

APP
业务功能
beeshell 1.0
接口层
扩展组件
Actionsheet
BottomModal
...
基础组件
Radio
Checkbox
Modal
Button
Topview
...
基础工具
React Native
iOS/Android 系统

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：

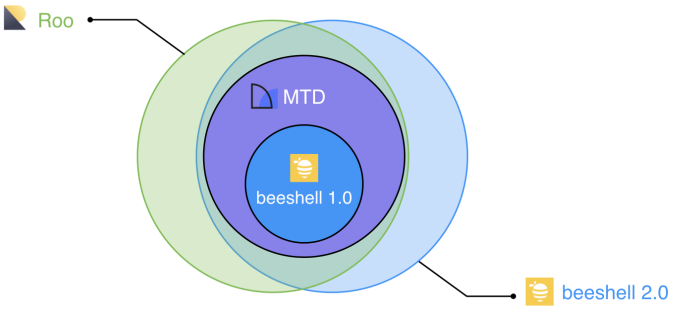
- 动态化能力。RN 本身不支持热更新，MRN 借助公司内发布平台，实现包的上传发布、下载更新、下线回滚等操作。
- 双端（未来三端）复用问题。从设计原则上保证开发者对平台的差异无明显感知。
- 保障措施。在开发保障方面：提供脚手架工具、模版工程、开发文档等，方便开发者日常的 MRN 开发工作。提供多级分包机制，业务内部和业务之间能够灵活组织代码；在运行保障方面，提供运行环境隔离，使得不同业务之间页面运行无干扰。提供数据采集和指标大盘，及时发现运行中的问题，同时提供降级能力以应对紧急情况。

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

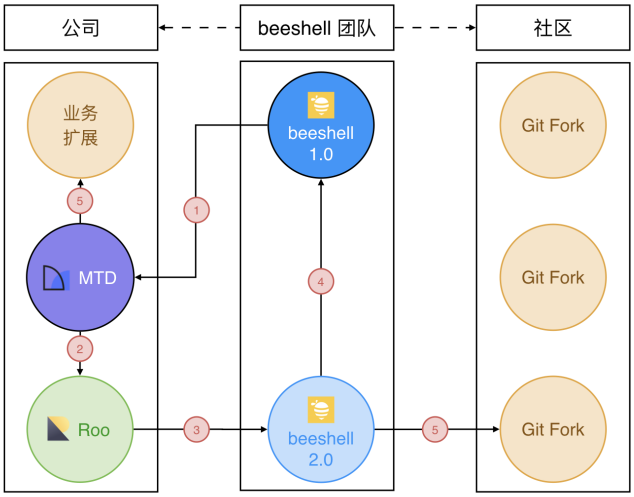



图 X

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：

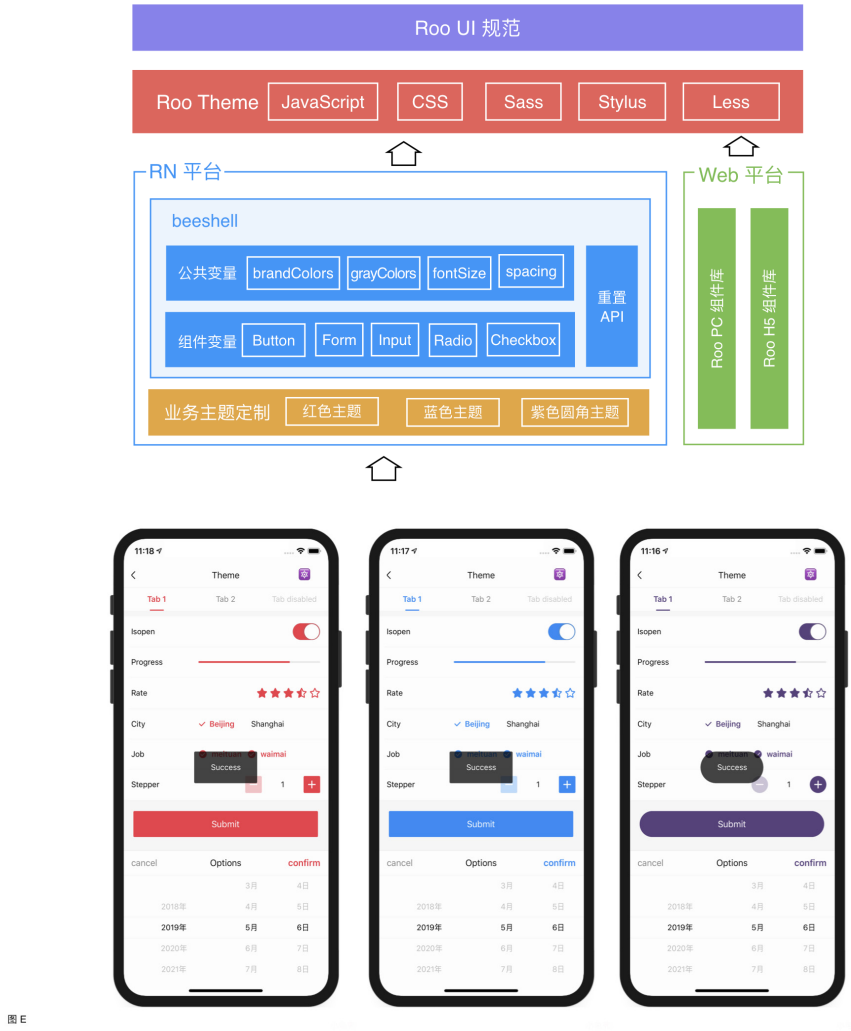



图 E

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<br>标题、正文 | GrayDarker<br>副标题 | GrayDark<br>补充信息 | Gray<br>取消、禁用 | GrayLight<br>— | GrayLighter<br>— | GrayLightest<br>— |
|---|---|---|---|---|---|---|
| #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：

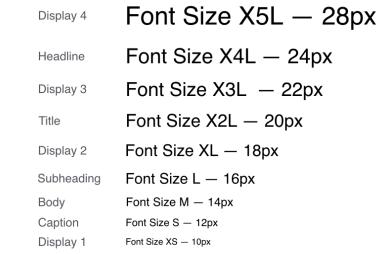



图 B

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：

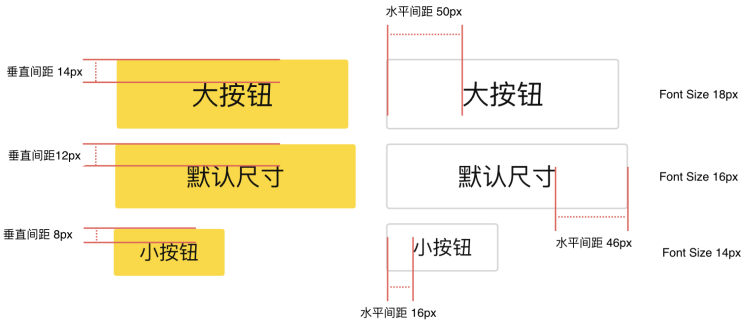



图 S

*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：

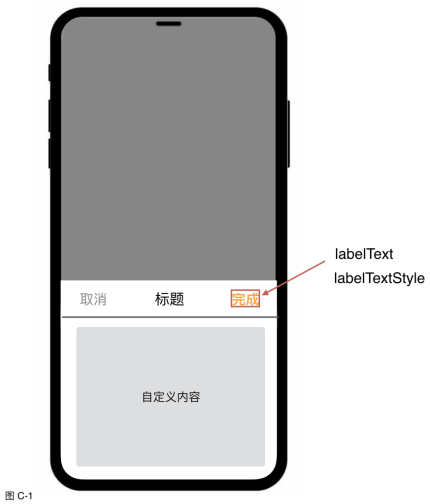



图 C-1

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：

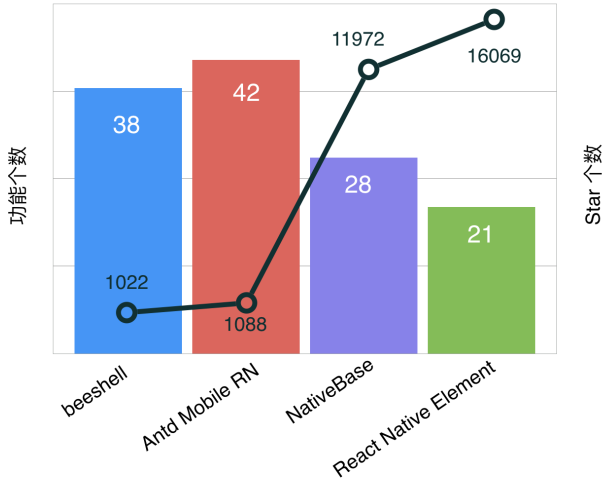



图 K

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。

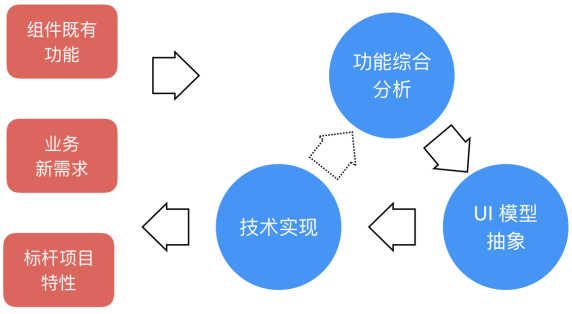



图 O

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：

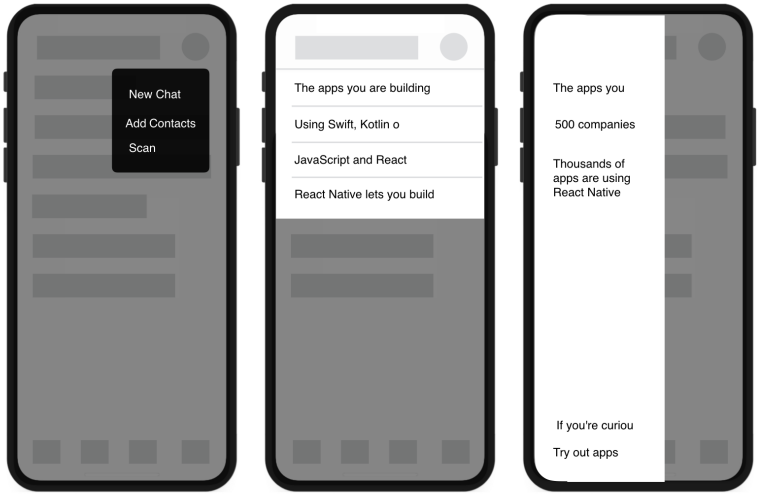



图 Q

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：

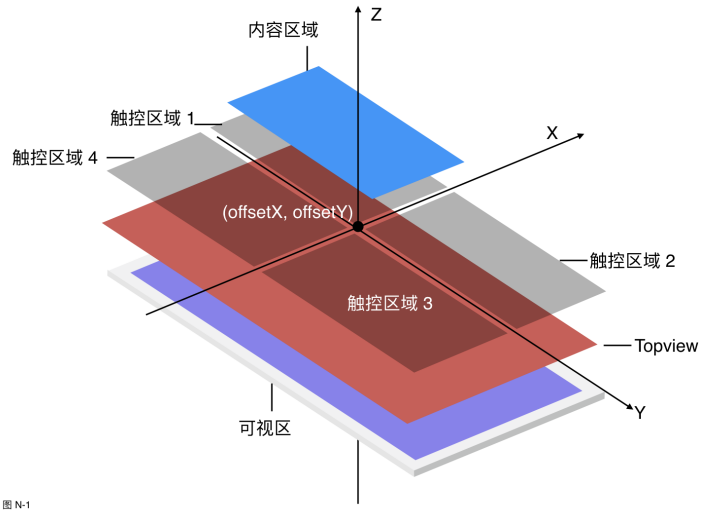



图 N-1

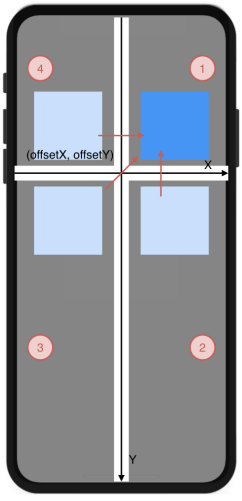



图 N-2

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。

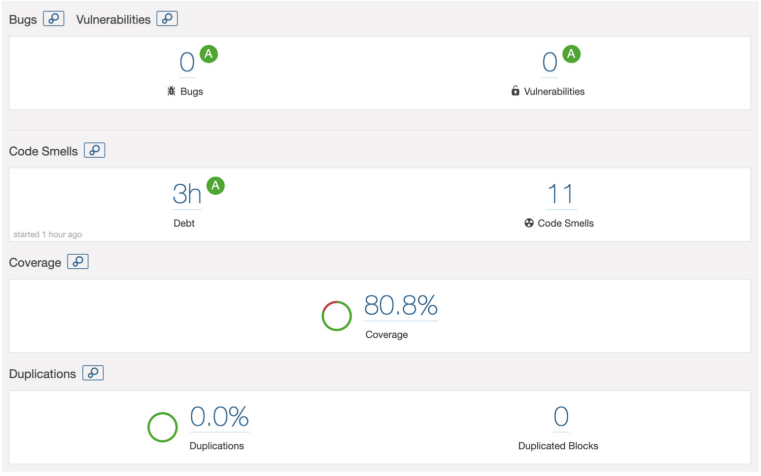
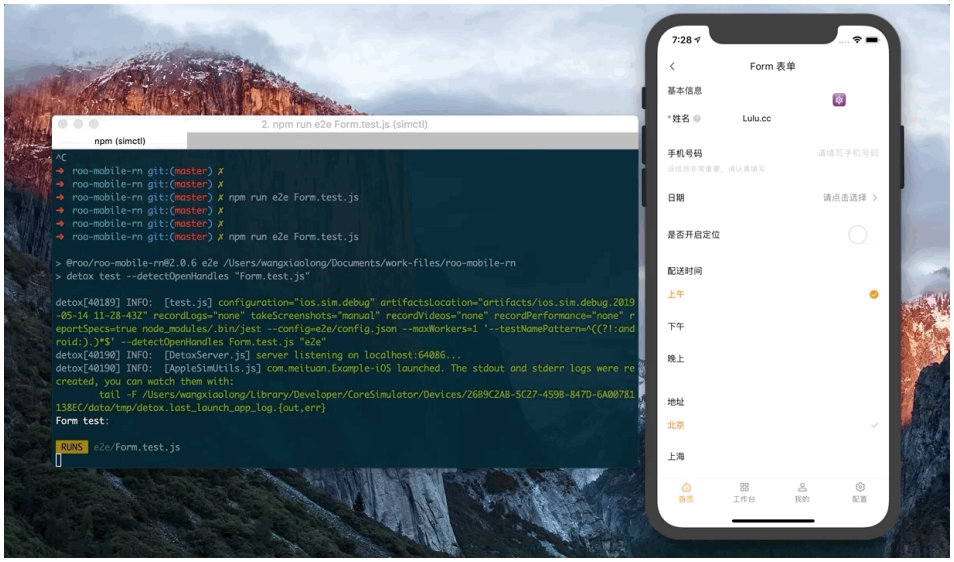



图 A

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

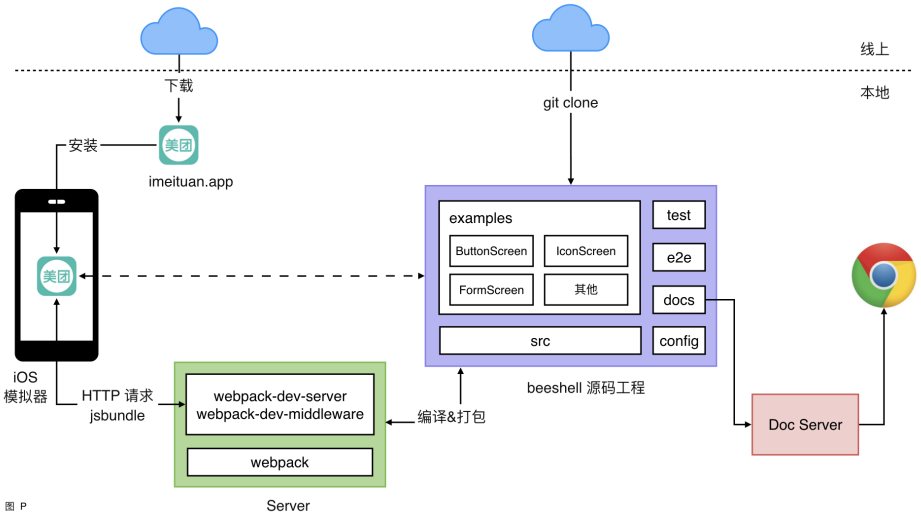
通过黑盒测试、白盒测试、灰盒测试，三种测试方案，更加全面的保证组件库的代码质量，大大提高了代码可维护性。

## 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。

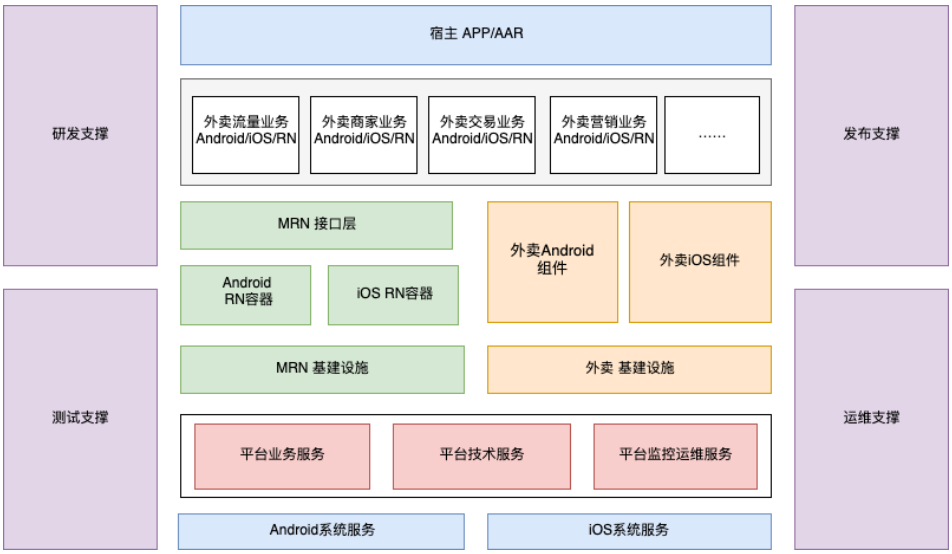
（2）有限的客户端研发资源无法满足日益增长的业务

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

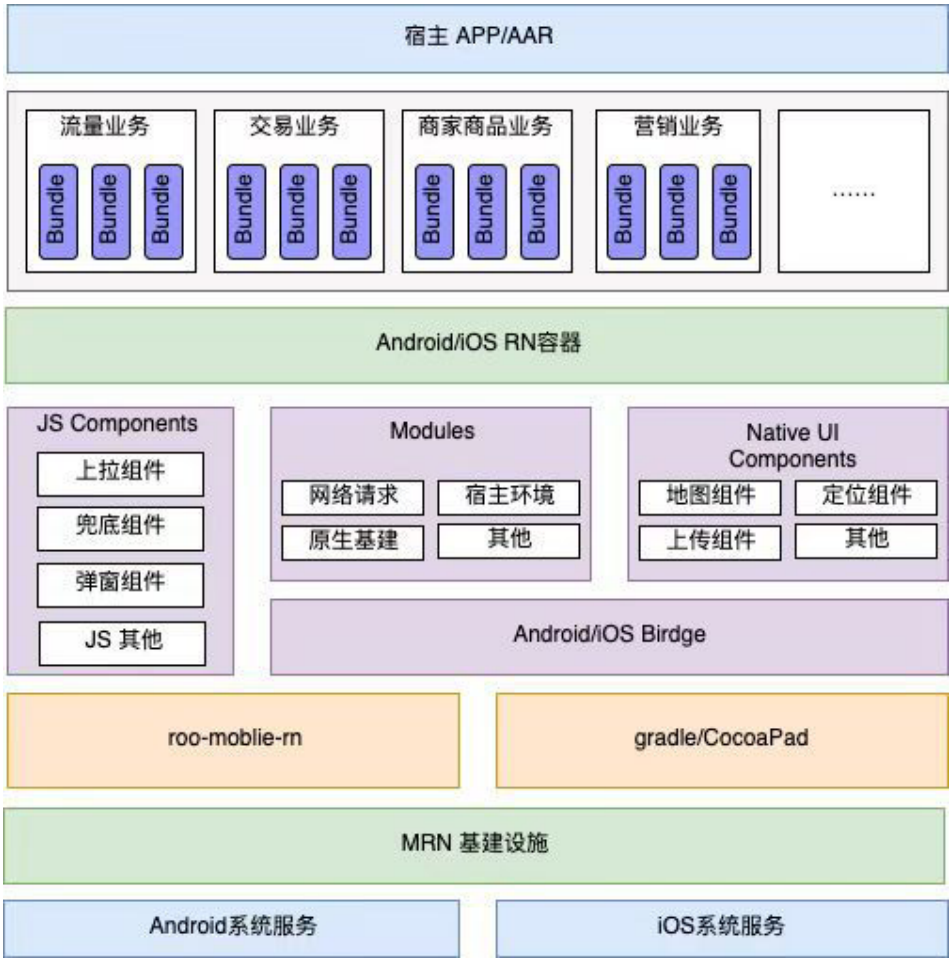
Android/iOS/RN。

- 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。

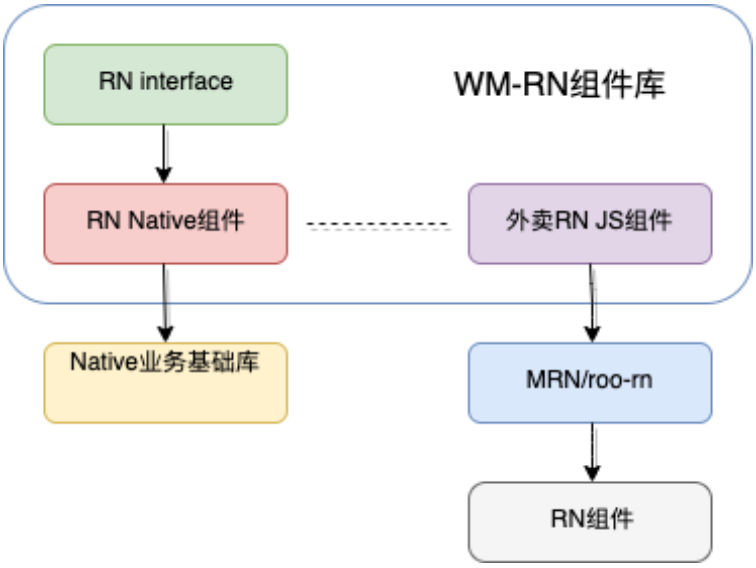
## 研发测试支撑

### 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。

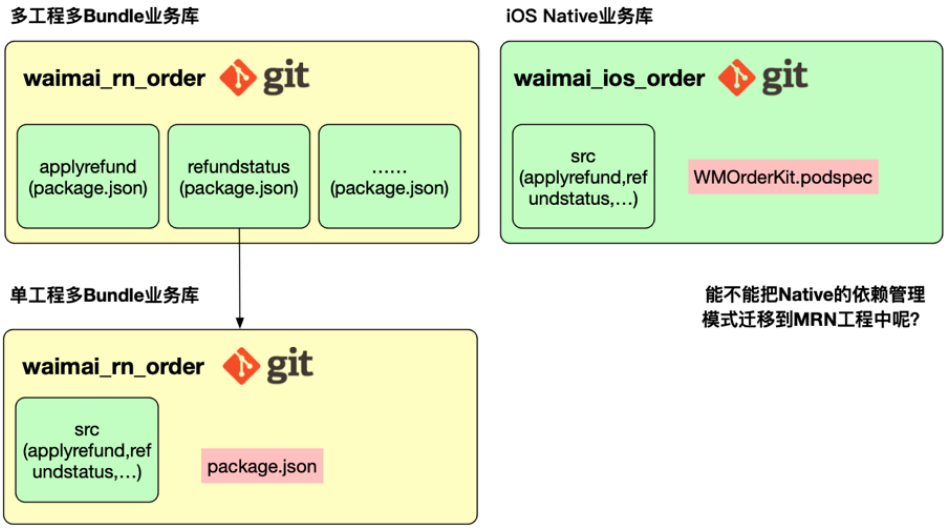
相比于单工程单 Bundle 方案，多工程多 Bundle 方案将页面进行解耦，使之基本可以满足中大型 MRN 项目的需求。在外卖 MRN 项目初期，一直都使用着这样的工程结构进行开发。但是我们也为之付出了相应的代价，即每个页面的依赖都需要对应 RD 去维护升级，依赖碎片化的问题日趋严重。同时在工程级别的管控，如统一 Lint 规则、Git Hook 等也变得更加复杂。

### 多工程多 Bundle 方案 => 单工程多 Bundle 方案

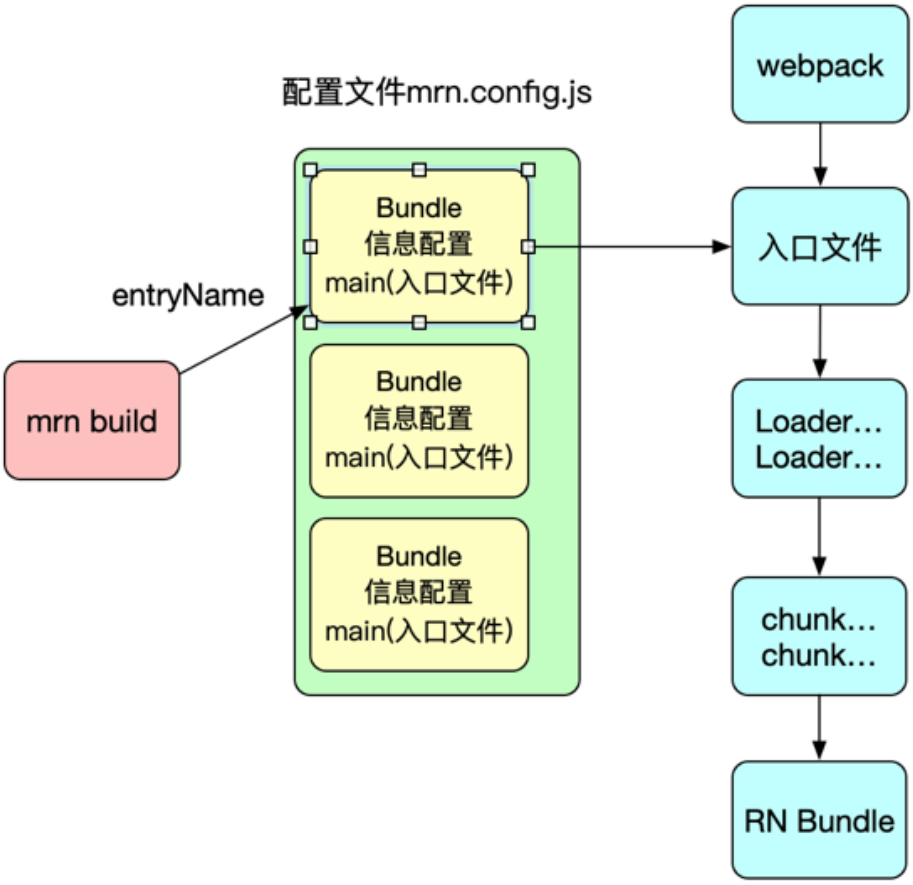
随着外卖 MRN 页面规模以及参与人规模的进一步增大，多工程多 Bundle 方案的缺点日益凸显。特别对于那些前端技术底子相对薄弱的团队来说，依赖管理问题会

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：

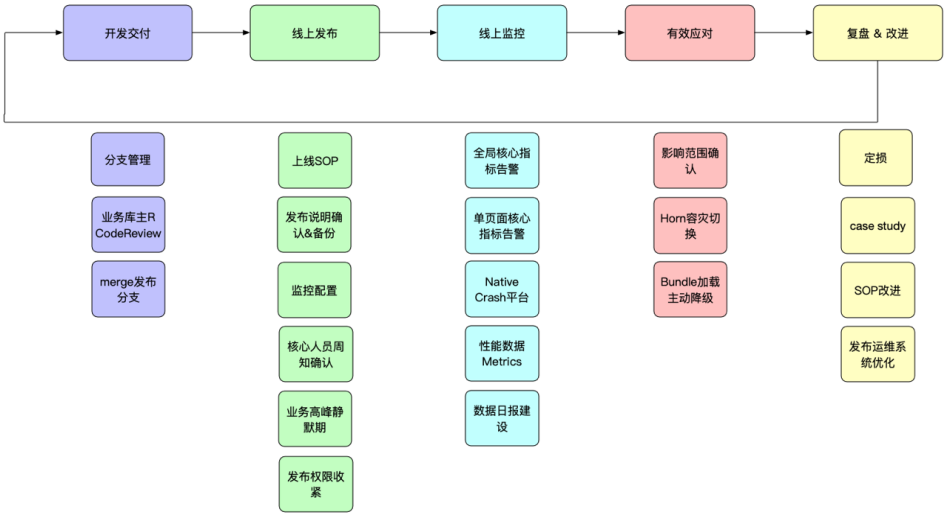
- 整个工程采用一个 package.json，管理业务库中所有的依赖。这样可以有效地解决各自页面去管理自己依赖时，必然产生的依赖版本碎片化问题，避免同一依赖库因为版本不一样，而导致页面表现不一样的问题。
- 从依赖角度去规范各自页面的使用工具规范，如 A 页面使用某一种三方库来实现某种功能，B 页面使用另一种三方库也实现了同一种功能，单一依赖管理就可以从库依赖的角度强制做技术选型，减少各个页面的实现差异，从而降低维护成本。
- 让业务同学可以更加专心地开发业务代码，不用关心复杂的依赖问题，大大提升了开发效率。

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。

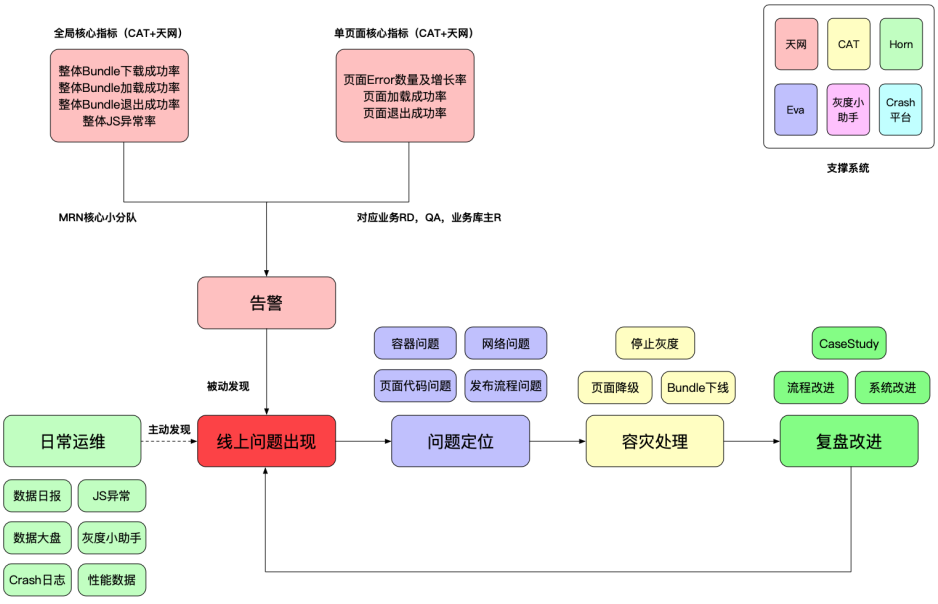
这两种备案保障了外卖 MRN 业务的整体高可用性。

（5）复盘改进

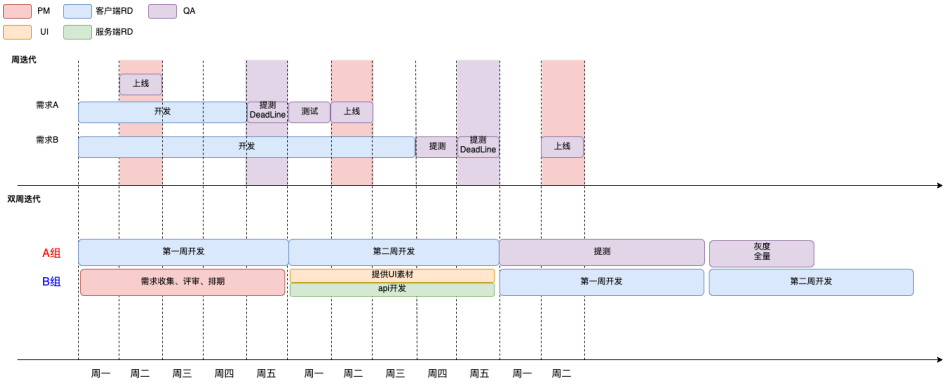
在以上四个大环节中，问题可能会出现在任意一个环节。除了及时发现问题与解决问题，我们还需要尽力避免问题。这一点主要是靠我们内部的例会、复盘会，对典型问题进行 Review，将问题进行归类，包括复盘流程规范问题、操作失误问题、框架 Bug 等，并力图通过规范流程、系统优化来尽力地避免问题。

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

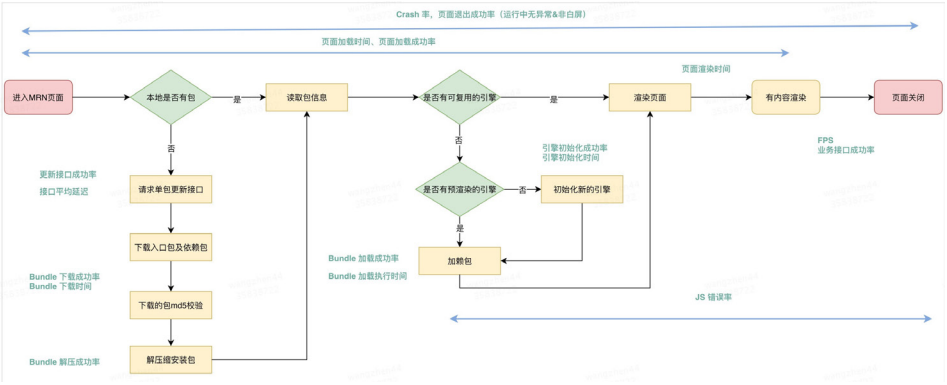
正如在“监控运维”章节中所讲到的那样，线上运维是我们工作的重中之重。这个章节我们就讲一下我们对于监控指标的选取。鉴于外卖业务的特殊性，除了美团的外卖频道之外，外卖业务还需要运行在独立的外卖 App 上。如下图所示：


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

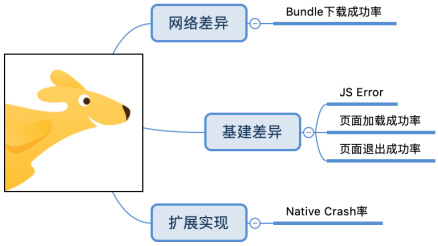
代码，Native 代码的 Bug 出现 JS Error、Native Crash 等问题，这样给用户带来的直观反馈就是应用闪退、页面白屏等，造成了服务的不可用。所以在使用链路上出现问题率，基本也可以直观反映出一个 RN 页面的质量以及它当前的运行状况。

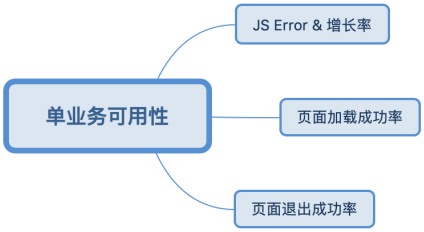
在使用链路上，我们主要关注的是 JS Error 率、JS Error 个数以及页面退出成功率（MRN PageExit Success）等。

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）。

# 代码质量与安全

## 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.

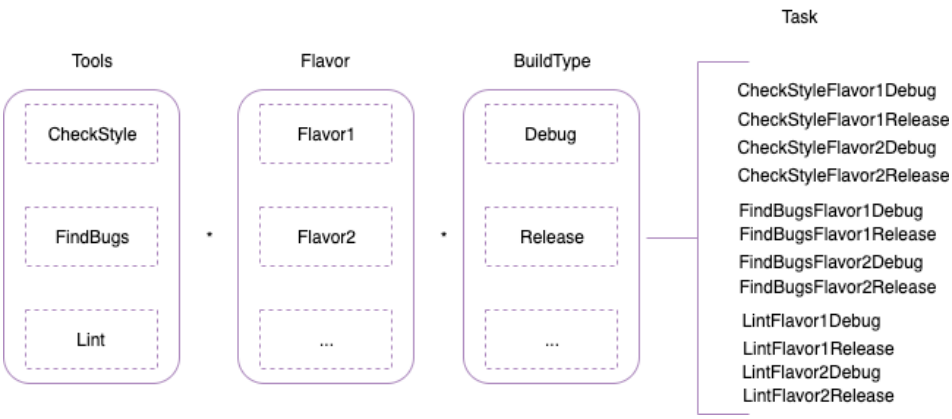
经过以上的对比分析我们发现，工具的诞生都能针对性解决某一领域问题。CheckStyle 的扫描速度快效率高，对代码风格和圈复杂度支持友好；FindBugs 针对 Java 代码潜在问题，能帮助我们发现编码上的一些错误实践以及部分安全问题和性能问题；Lint 是官方深度定制，功能极其强大，且可定制性和扩展性以及全面性都表现良好。所以综合考虑，针对思考一，我们的结论是整合三种扫描工具，充分利用每一个工具的领域特性。

## 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；

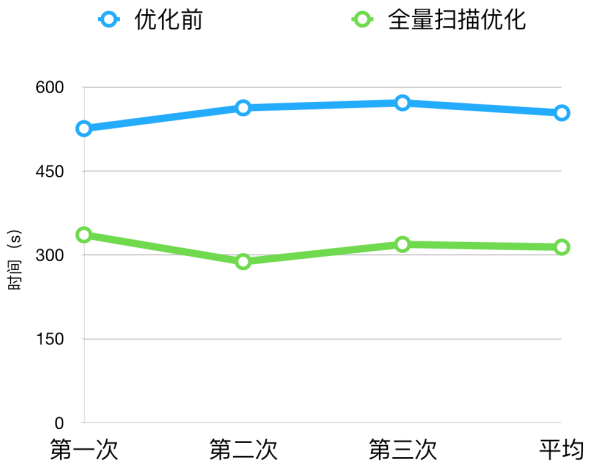
```
// 排除掉一些模板代码 class 文件
static final Collection<String> defaultExcludes =
(androidDataBindingExcludes +
androidExcludes + butterKnifeExcludes + dagger2Excludes).asImmutable()

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
```

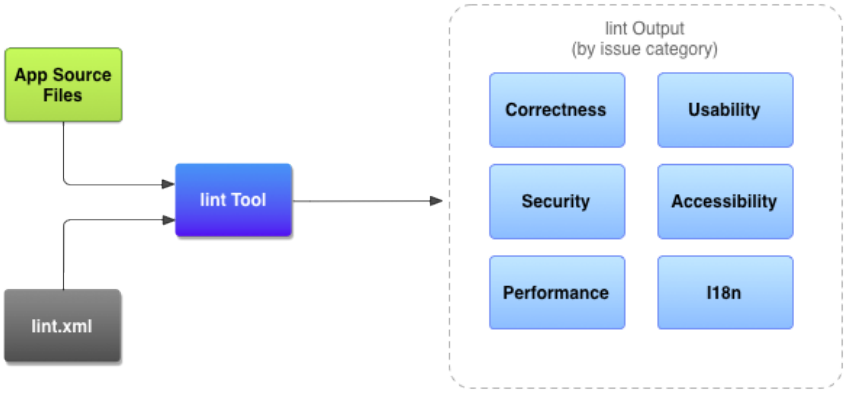
1. 配置远程分支别名为 UpStream，其中 upstreamGitUrl 可以在插件提供的配置属性中设置；
2. 获取远程目标分支的更新；

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

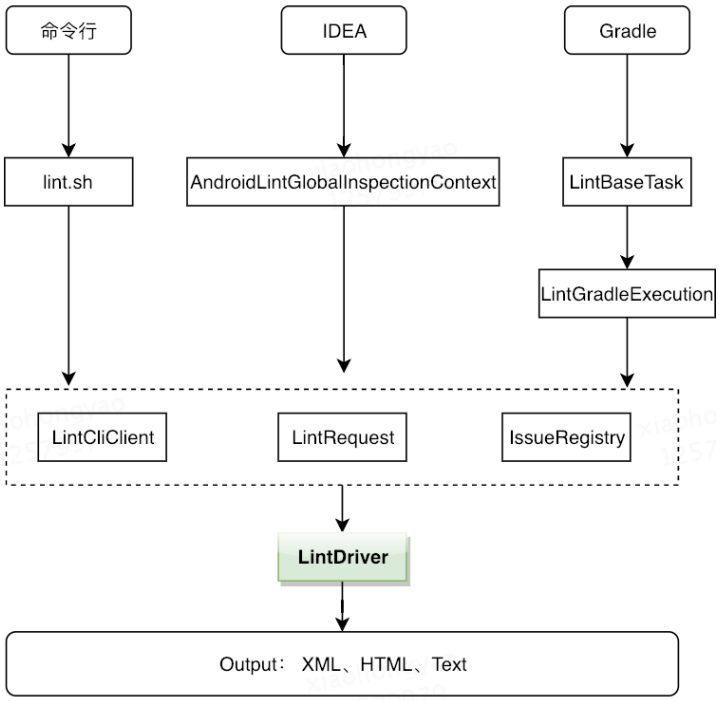
一套完整的扫描工具用于对 Android 的代码结构进行分析，可以通过命令行、IDEA、Gradle 命令三种方式运行 lint 工具。

### 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

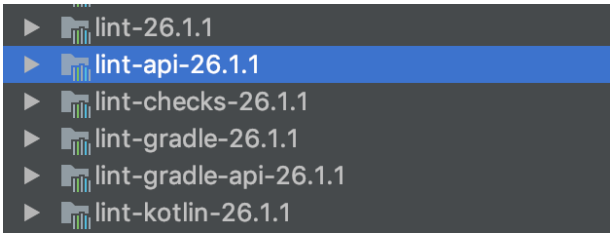
描工具集，有多种使用方式。Android 为我们提供了三种运行方式，分别是命令行、IDEA、Gradle 任务。这三种方式最终都殊途同归，通过 LintDriver 来实现扫描。如下图所示：




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。

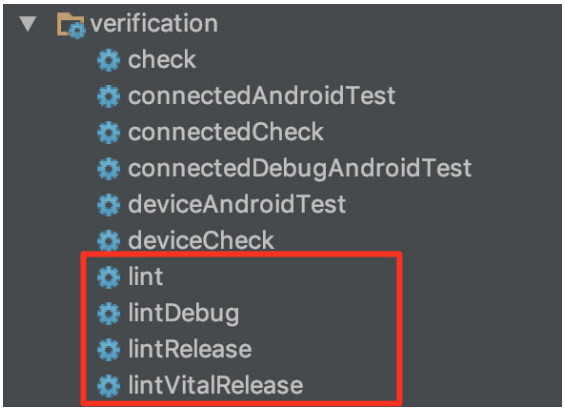
```
/**
     * Adds the given file to the list of files which should be
checked in this
     * project. If no files are added, the whole project will be
checked.
     *
     * @param file the file to be checked
     */
    public void addFile(@NonNull File file) {
        if (files == null) {
            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 lintVitalRelease 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

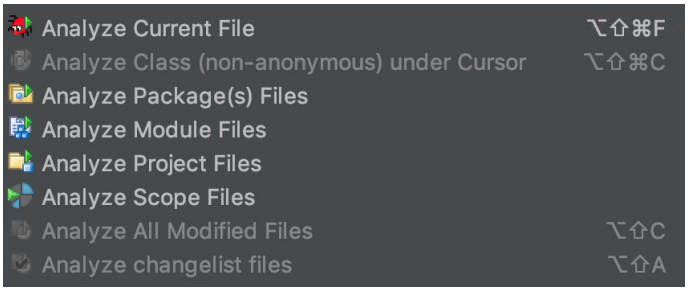
Bug 都报告，如果为 High，仅报告 High 优先级。

- Reports 扫描结果存放路径。

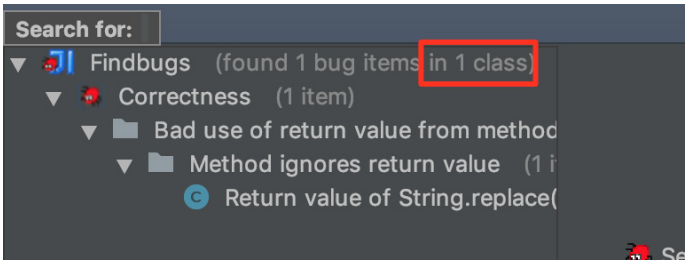
通过以上属性解释，不难发现要 FindBugs 增量扫描，只需要指定 Classes 的文件集合就可以了。

## 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
                    }
                }
            }
        }
    }
}
```

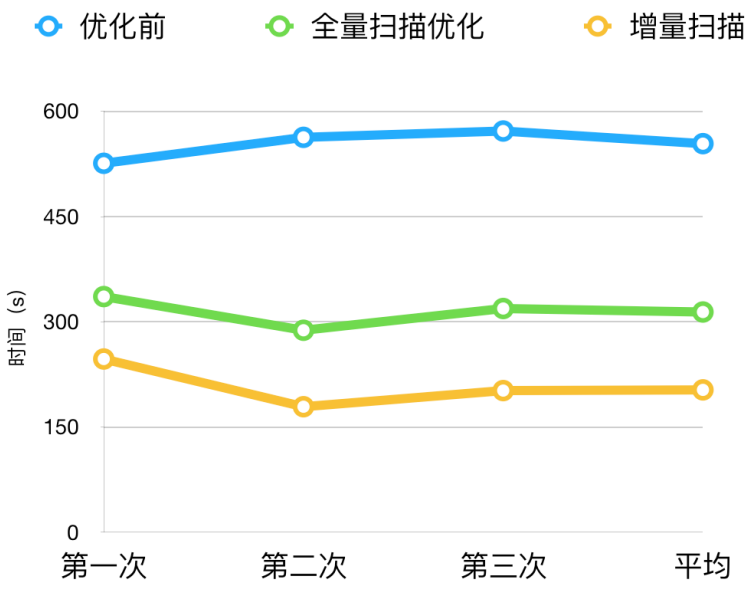
通过以上方式，我们可以解决一些增量扫描时出现的误报情况，相比 IDEA 工具，我们更进一步降低了扫描部分文件的误报率。

## 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

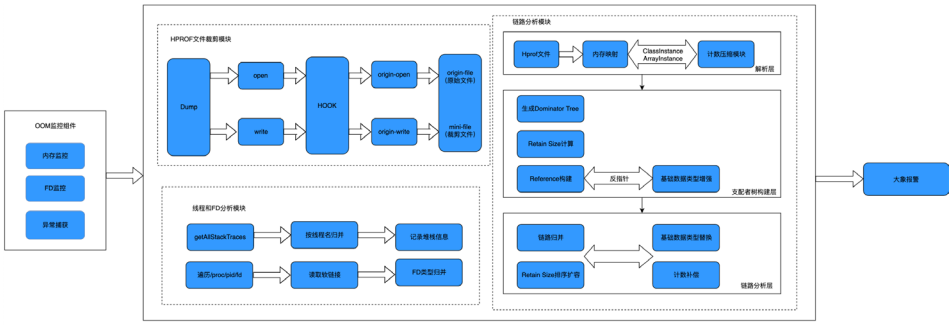
美团餐饮生态技术团队诚招 Android、Java 后端高级 / 资深工程师和技术专家，Base 成都，欢迎有兴趣的同学投递简历到 tech@meituan.com。

# Probe：Android 线上 OOM 问题定位组件

逢搏　毅然　永刚

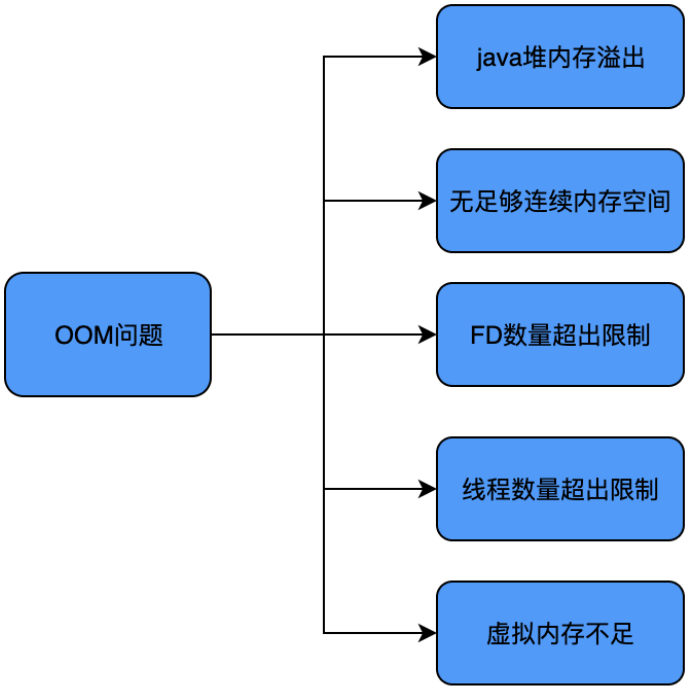
配送骑手端 App 是骑手用于完成配送履约的应用，帮助骑手完成接单、到店、取货及送达，提供各种不同的运力服务，也是整个外卖闭环中的重要节点。由于配送业务的特性，骑手 App 对于应用稳定性的要求非常高，体现 App 稳定性的一个重要数据就是 Crash 率，而在众多 Crash 中最棘手最难定位的就是 OOM 问题。对于骑手端 App 而言，每天骑手都会长时间的使用 App 进行配送，而在长时间的使用过程中，App 中所有的内存泄漏都会慢慢累积在内存中，最后就容易导致 OOM，从而影响骑手的配送效率，进而影响整个外卖业务。

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：

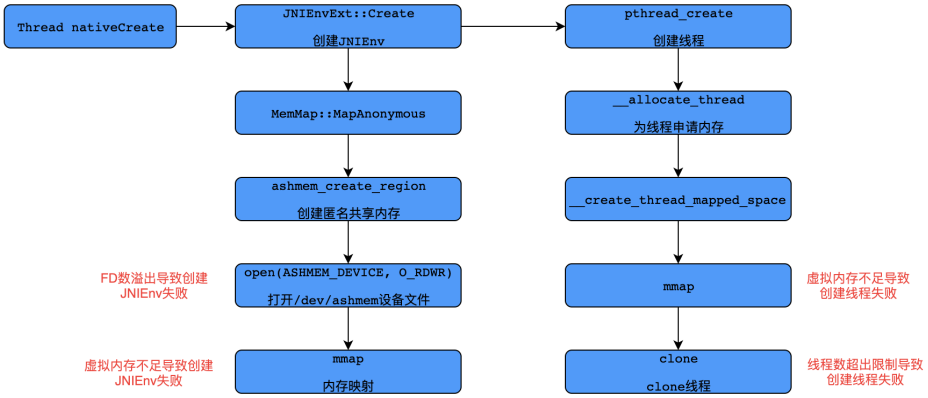
1. 为对象分配内存时达到进程的内存上限。由 Runtime.getRuntime.Max-Memory() 可以得到 Android 中每个进程被系统分配的内存上限，当进程占用内存达到这个上限时就会发生 OOM，这也是 Android 中最常见的 OOM 类型。
2. 没有足够大小的连续地址空间。这种情况一般是进程中存在大量的内存碎片导致的，其堆栈信息会比第一种 OOM 堆栈多出一段信息：failed due to fragmentation (required continguous free “<< required_bytes << “bytes for a new buffer where 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。

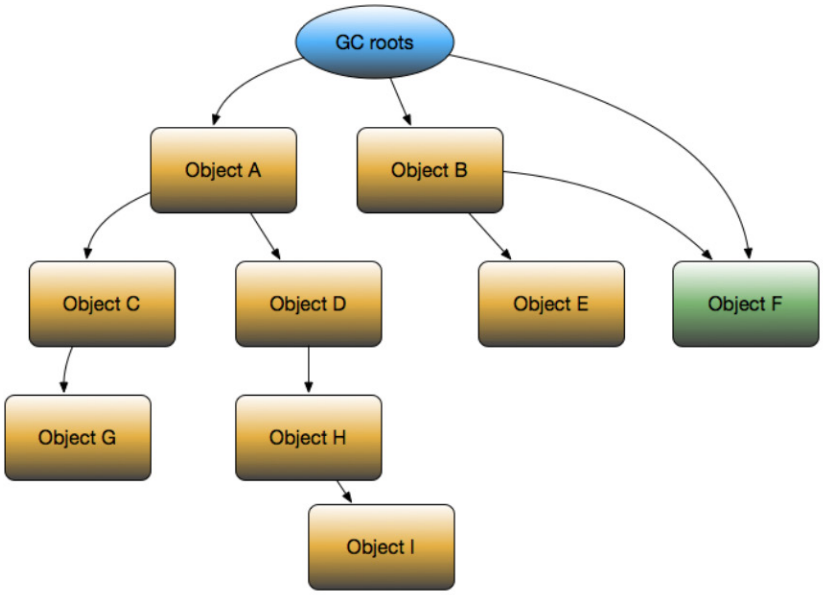
堆内存分配失败，通常说明进程中大部分的内存已经被占用了，且不能被垃圾回收器回收，一般来说此时内存占用都存在一些问题，例如内存泄漏等。要想定位到问题所在，就需要知道进程中的内存都被哪些对象占用，以及这些对象的引用链路。而这些信息都可以在 Java 内存快照文件中得到，调用 Debug.dumpHprof-Data(String fileName) 函数就可以得到当前进程的 Java 内存快照文件（即 HPROF 文件）。所以，关键在于要获得进程的内存快照，由于 dump 函数比较耗时，在发生 OOM 之后再去执行 dump 操作，很可能无法得到完整的内存快照文件。

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

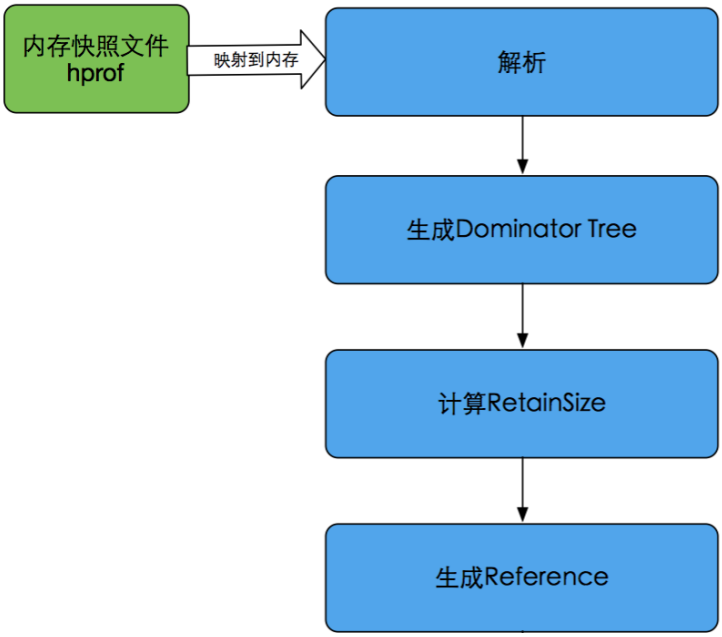
首先，我们介绍几个基本概念：




- **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。

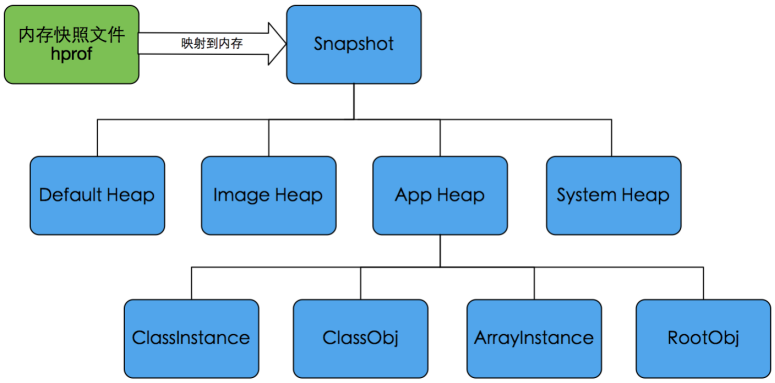
### 分析进程自身 OOM

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。

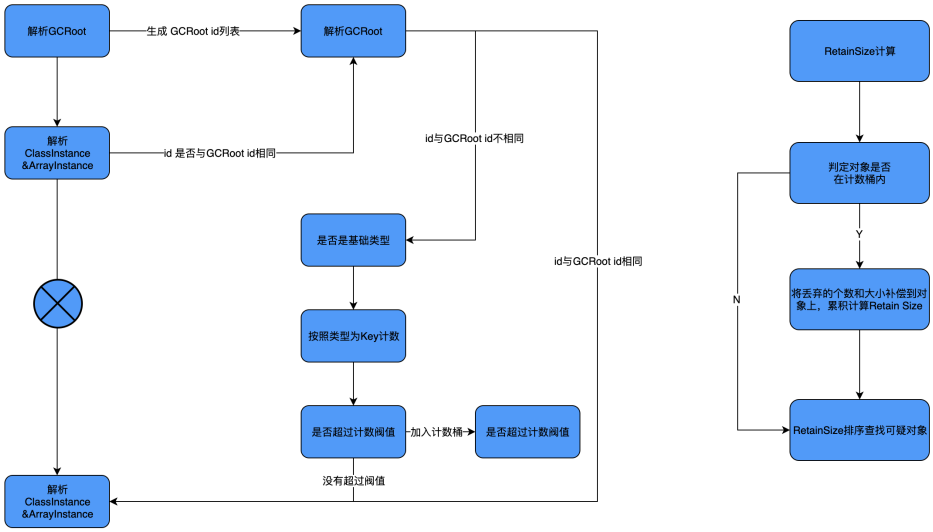



HPROF 文件映射到内存的过程：

```
// 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。

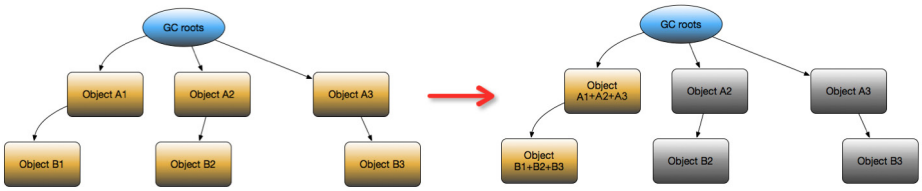



### 链路分析时间过长

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”。

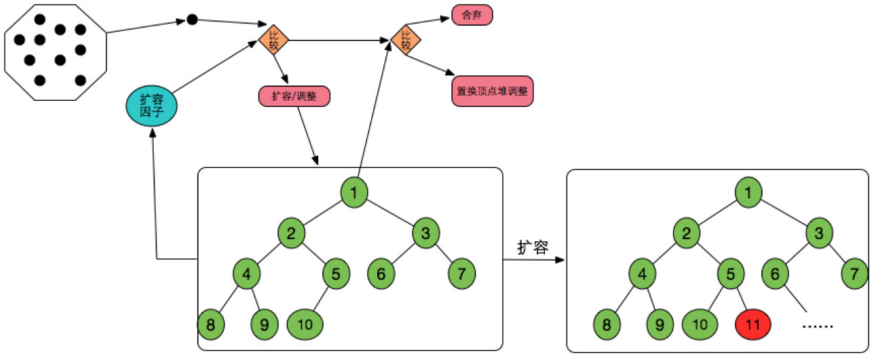



我们在生成 Reference 之后，做了一步链路归并（如上图），即对于同一个对象的不同 Instance，如果其底下的引用链路中的对象类型也相同，则进行归并，并记录 Instance 的个数和每个 Instance 的 RetainSize。

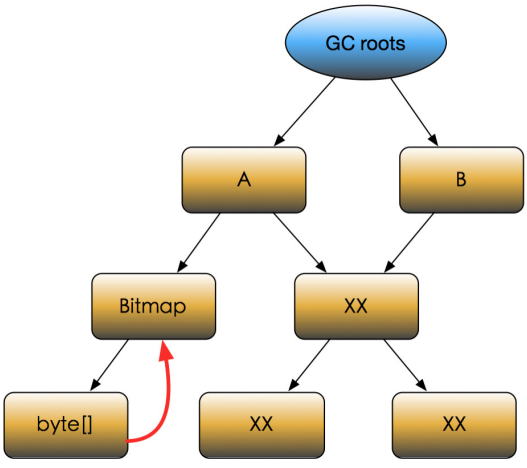
然后对归并后的 Instance 按 RetainSize 进行排序，取出 TOP N 的 Instance，

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。

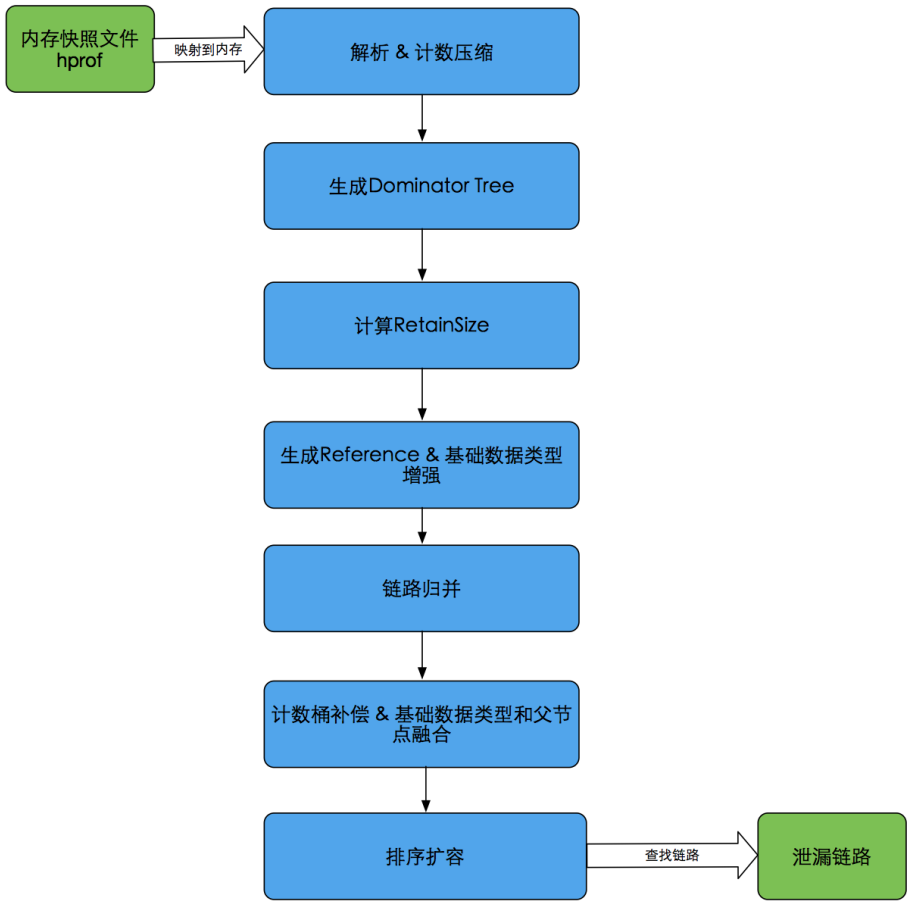



### 基础类型检测不到



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

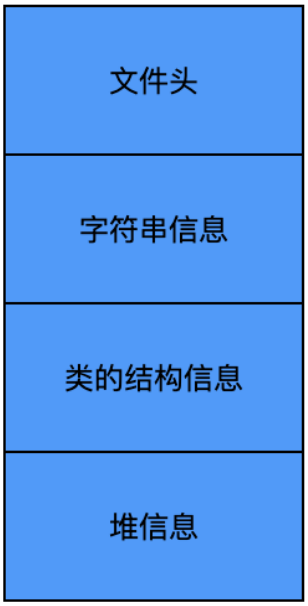
在 Probe 上线分析方案之后，发现尽管我们做了很多优化，但是受到手机自身性能的约束，线上分析的成功率也只有 65%。

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。

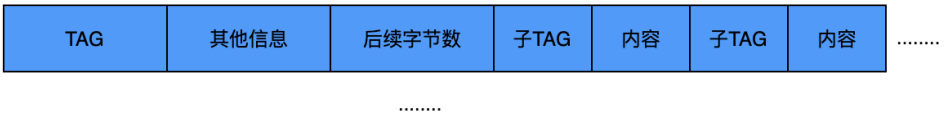
HPROF 文件的大体格式如下：




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。

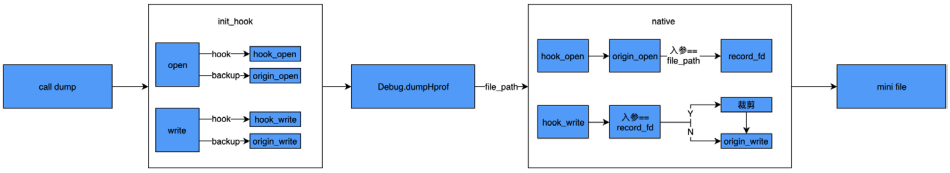
```
// Values for the first byte of HEAP_DUMP and HEAP_DUMP_SEGMENT records:
enum HprofHeapTag {
  // Traditional.
  HPROF_ROOT_UNKNOWN = 0xFF,
  HPROF_ROOT_JNI_GLOBAL = 0x01,
  HPROF_ROOT_JNI_LOCAL = 0x02,
  HPROF_ROOT_JAVA_FRAME = 0x03,
  HPROF_ROOT_NATIVE_STACK = 0x04,
  HPROF_ROOT_STICKY_CLASS = 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。

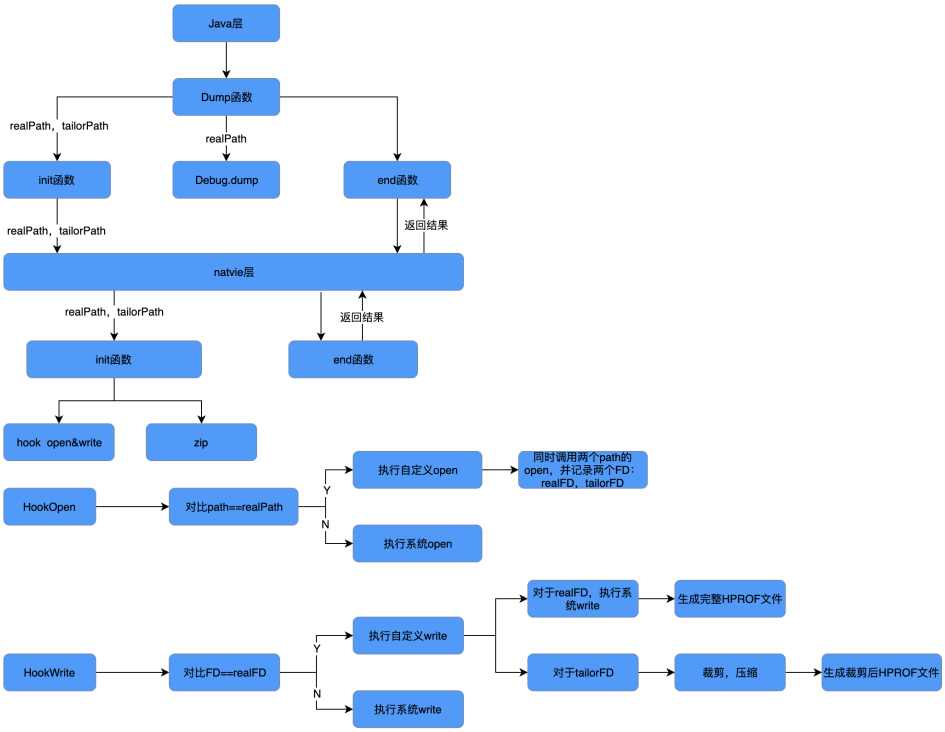
原始 HPROF 文件和裁剪后再恢复的 HPROF 文件分别在 Android Studio 中打开，发现裁剪再恢复的 HPROF 文件打开后，只是看不到对象中的基础数据类型值，而整个的结构、对象的分布以及引用链路等与原始 HPROF 文件是完全一致的。事实证明裁剪方案不会影响后续对堆内存的链路分析。

### 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。

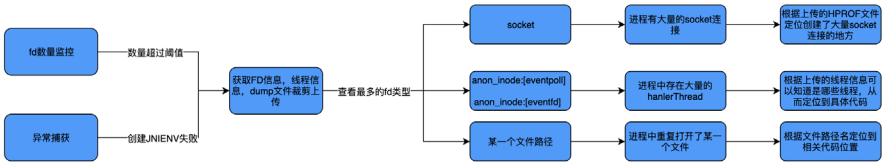
在 /proc/pid/limits 描述着 Linux 系统对对应进程的限制，其中 Max open files 就代表可创建 FD 的最大数目。

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

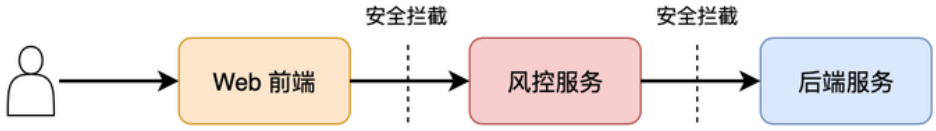
对于营销活动类的 Web 页面，领券、领红包、抽奖等活动方式很常见。此类活动对于普通用户来说大多数时候就是“拼手气”，而对于非正常用户来说，可以通过直接刷活动 API 接口的“作弊”方式来提升“手气”。这样的话，对普通用户来说，就变得很不公平。

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。



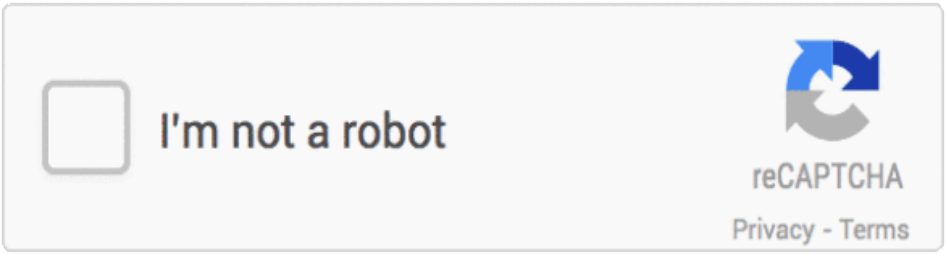
- Web 前端：用户通过 Web 页面来参与活动，同时 Web 前端也会收集用于人机识别验证的用户交互行为数据。由于不同终端（移动端 H5 页面和 PC 端页面）交互形式不同，收集用户交互行为数据的侧重点也会有所不同。
- 风控服务：一般大公司都会有专业的风控团队来提供风控服务，在美团内部有智能反爬系统来基于地理位置、IP 地址等大数据来提供频次限制、黑白名单限制等常规的基础风控拦截服务。甚至还有依托于海量的全业务场景的用户大数据，使用贝叶斯模型、神经网络等来构建专业度较深的服务。风控服务可以为 Web 前端提供通用的独立验证 SDK：验证码、滑块验证等区分人机的“图灵验证”，也可以为服务端提供 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](0,-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");
```

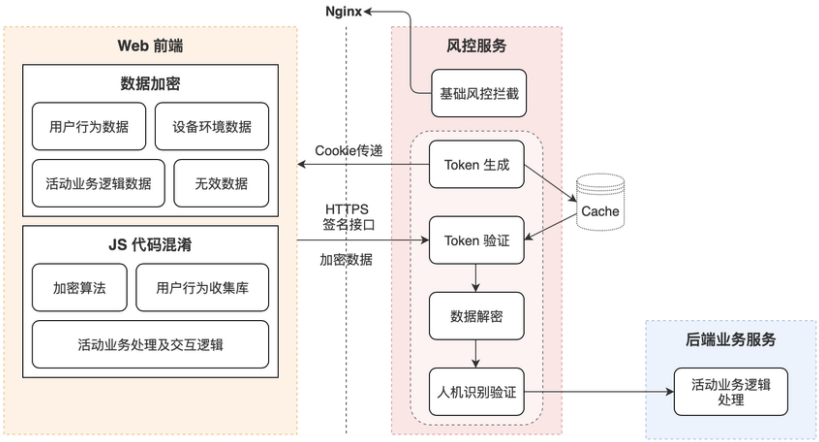
断点工具的防范会更麻烦些。在使用断点工具时通常都会导致代码延迟执行，而正常的业务逻辑都会立即执行，这是一个可以利用的点，可以考虑在代码执行间隔上来防范断点工具。

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 {}<br>~~~~~~~~~~~~~~~ [0] //提示命名不规范<br>class Another_Invalid_Class_Name {<br>~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ [0] //提示命名不规范<br>} |
| no-unsafe-finally | 不许在finally里面使用return\continue\break\throws | function() {<br>try {<br>} finally {<br>return;<br>~~~~~~~~ [return] //提示不许在finally里面使用return<br>}<br>} |
| 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,<br>"abcd");<br>~~~~~~~ [arguments are not aligned] //提示未对齐 |
| no-empty | 不允许空块出现 | if (x === 1) {}<br>~~ [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) {<br>case 1:<br>bar();<br>case 2:<br>~~~~ [expected a 'break' before 'case'] //提示在case关键字前要有break<br>bar();<br>bar();<br>case 3:<br>~~~~ [expected a 'break' before 'case']<br>case 4:<br>default:<br>break;<br>} |
| ... | ... | ... |

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

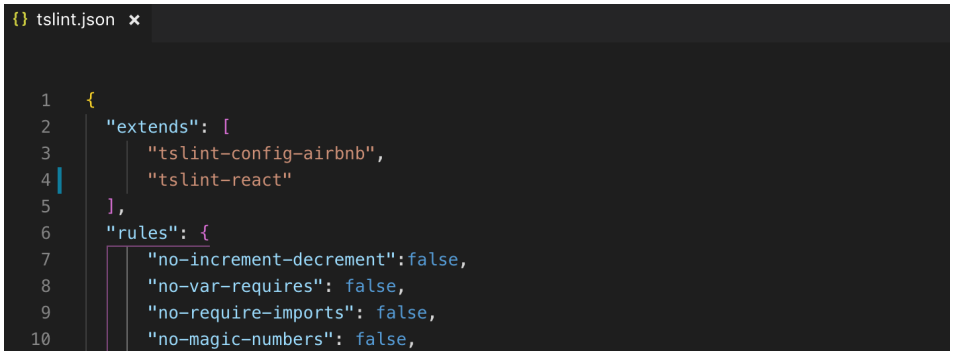
我们在项目的规则配置过程中，一般采用上述规则包其中一种或者若干种同时配置，那如何配置呢？请看下文。

## 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。

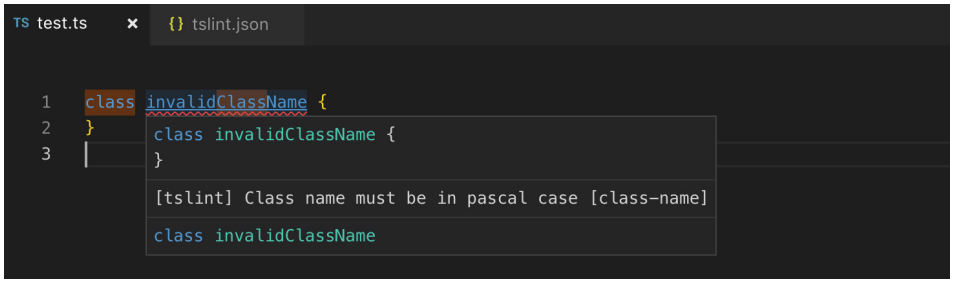
以上配置信息，这里具体参考了 https://palantir.github.io/tslint/usage/rule-flags/。

### 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

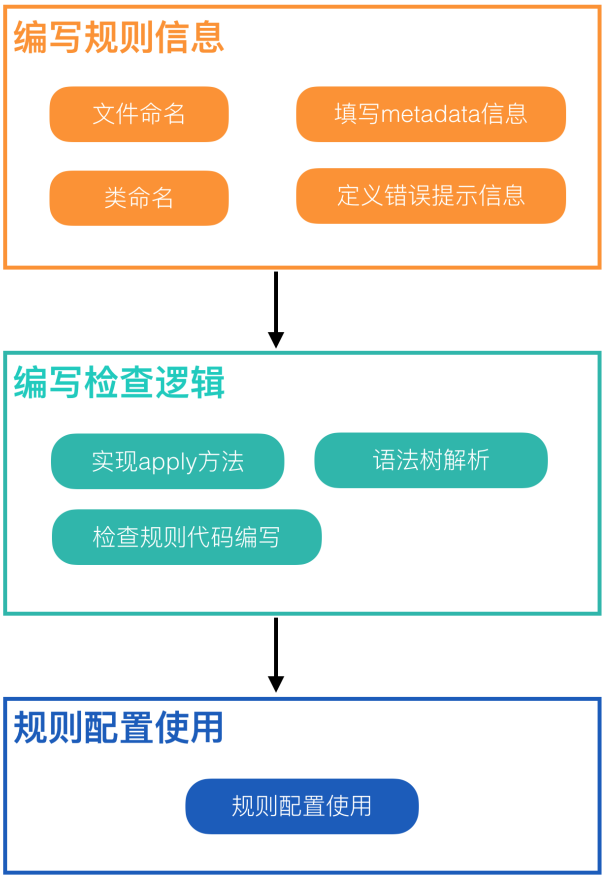
当前的 TSLint 规则虽然涵盖了比较普遍问题的一些代码检查，但是实践中还是存在一些问题的：

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”-

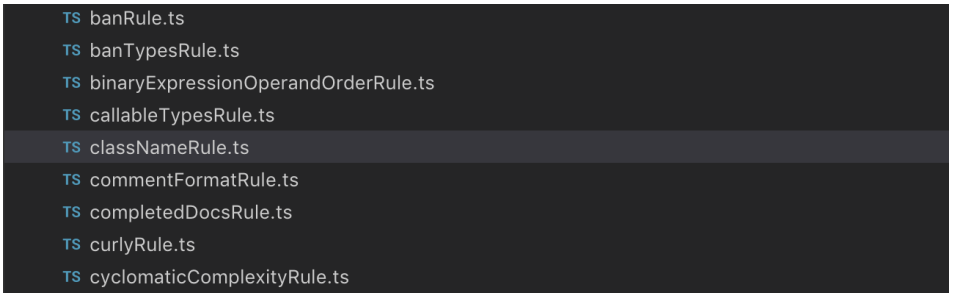
class-name”规则可以有效解决这个问题。我们可以看下具体的源码文件：https://github.com/palantir/tslint/blob/master/src/rules/classNameRule.ts。

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.forEach-Child 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

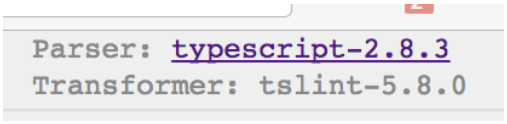
#### AST Explorer

**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。

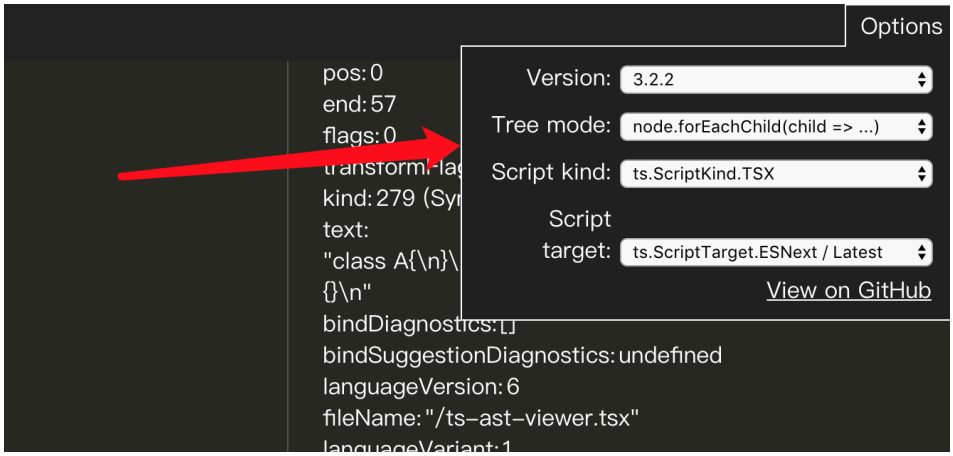
```
kind: 203
name: Identifier = $node {
   pos: 41
   end: 53
   flags: 0
   escapedText: "invalidName"
   text: "invalidName"
   kind: 71
   *leadingComments: undefined
   *trailingComments: undefined
}
typeParameters: undefined
```

TSLint

### TypeScript AST Viewer

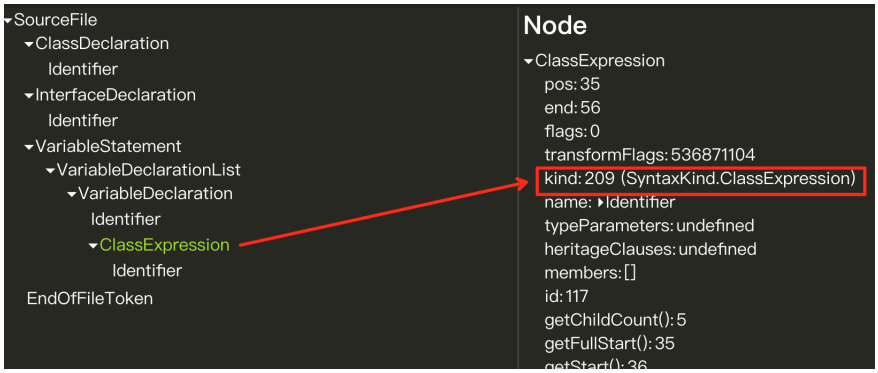
**优点：**

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：

typescript.d.ts

```
61    }
62    enum SyntaxKind {
63        Unknown = 0,
64        EndOfFileToken = 1,
65        SingleLineCommentTrivia = 2,
66        MultiLineCommentTrivia = 3,
67        NewLineTrivia = 4,
68        WhitespaceTrivia = 5,
69        ShebangTrivia = 6,
70        ConflictMarkerTrivia = 7,
71        NumericLiteral = 8,
72        StringLiteral = 9,
73        JsxText = 10,
74        JsxTextAllWhiteSpaces = 11,
75        RegularExpressionLiteral = 12,
76        NoSubstitutionTemplateLiteral = 13,
77        TemplateHead = 14,
78        TemplateMiddle = 15,
79        TemplateTail = 16,
80        OpenBraceToken = 17,
81        CloseBraceToken = 18,
82        OpenParenToken = 19,
83        CloseParenToken = 20,
84        OpenBracketToken = 21,
85        CloseBracketToken = 22,
```

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。

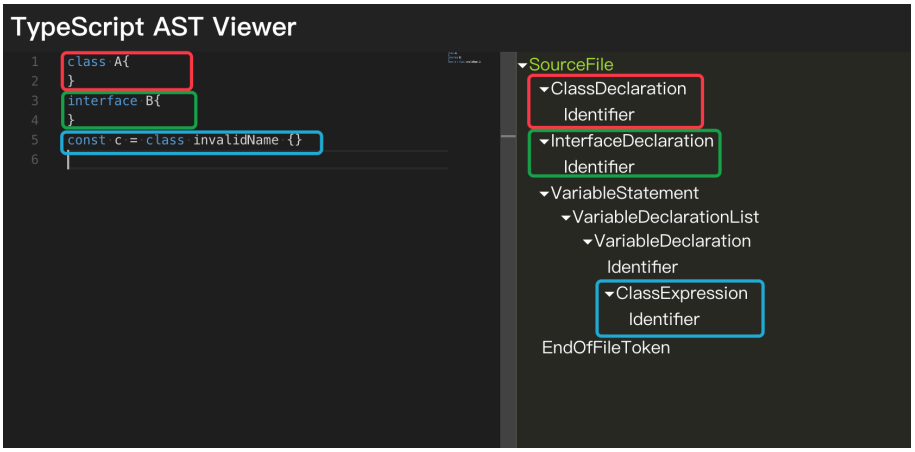
### 第七步，检查规则代码编写

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

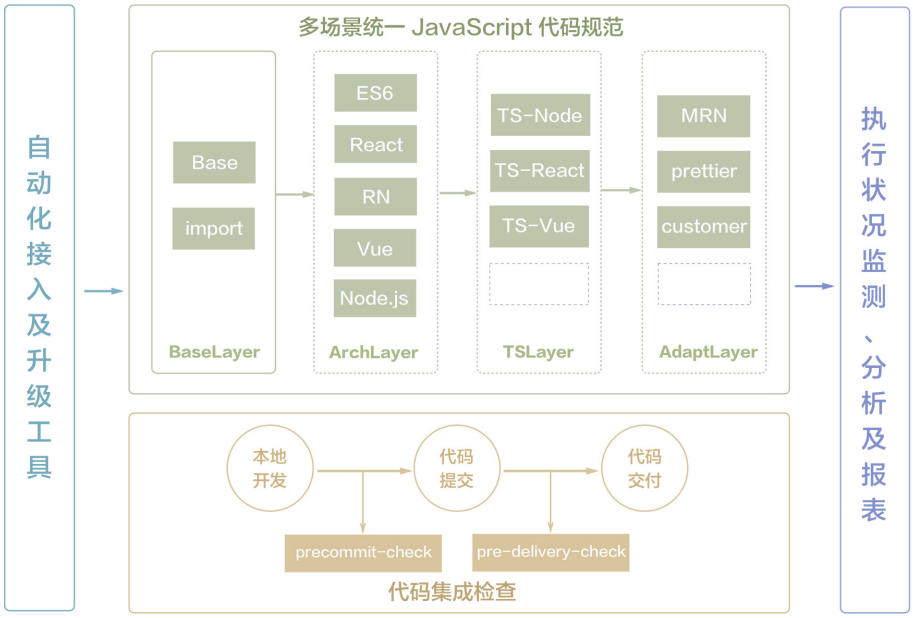
成本能节约大量的人力，同时也有利于方案落地推进。

- 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

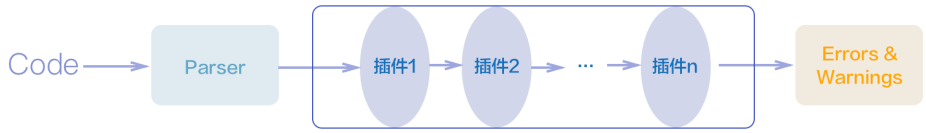
这一模块主要借助 ESLint 的基础特性，采用分层分类的结构设计，提供多场景、多技术方案的通用配置方案，并使方案具备易维护、易扩展的特性。

### 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

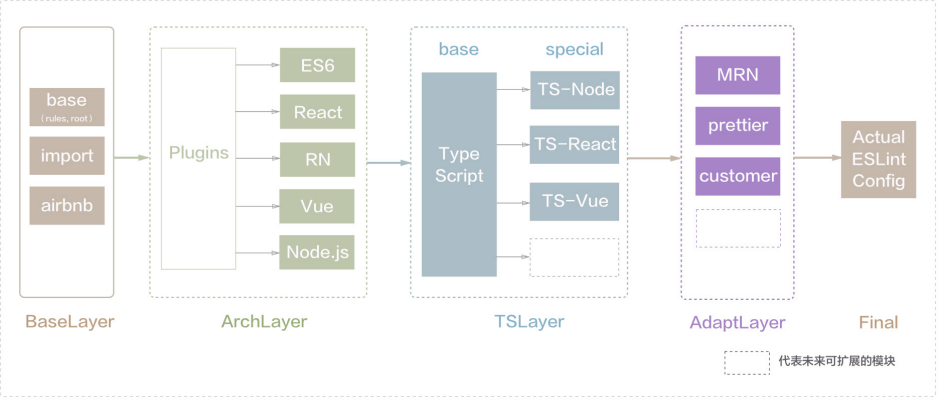
ESLint 提供了全面、灵活的配置能力，可以对解析器、规则、环境、全局变量等进行配置；可以快速引入另一份配置，和当前配置层叠组合为新的配置；还可以将配置好的规则集发布为 npm 包，在工程内快速应用。

### 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

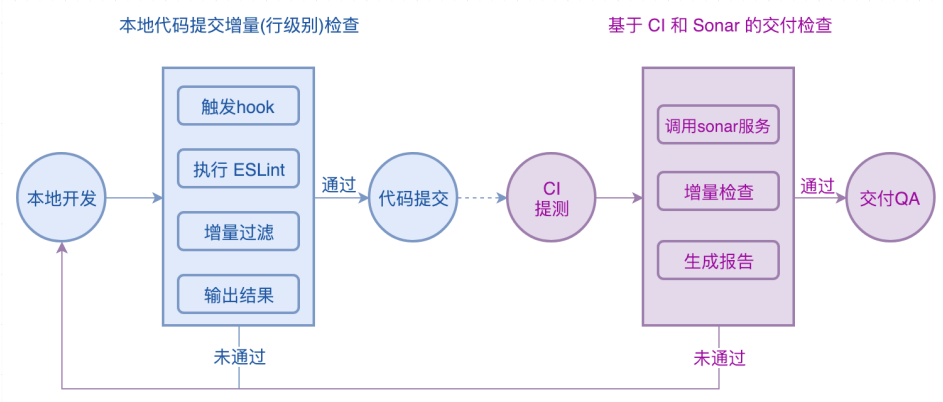
基于团队对工程化基础设施的建设，将代码规范静态检查与开发工作流集成，保证代码规范的落实。

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：

```
→  waimai_mfe_ems git:(feature/WAIMAISP-8833) ✗ git commit
**start eslint**


/Users/songpeng/mtcode/mfe/contract/waimai_mfe_ems/src/pages/taskList/constants.ts
  10:1   error  Missing return type on function  @typescript-eslint/explicit-function-return-type
  39:70  error  Missing semicolon                semi

✖ 2 problems (2 errors, 0 warnings)


eslint check info collected.

**eslint failed**
```

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。

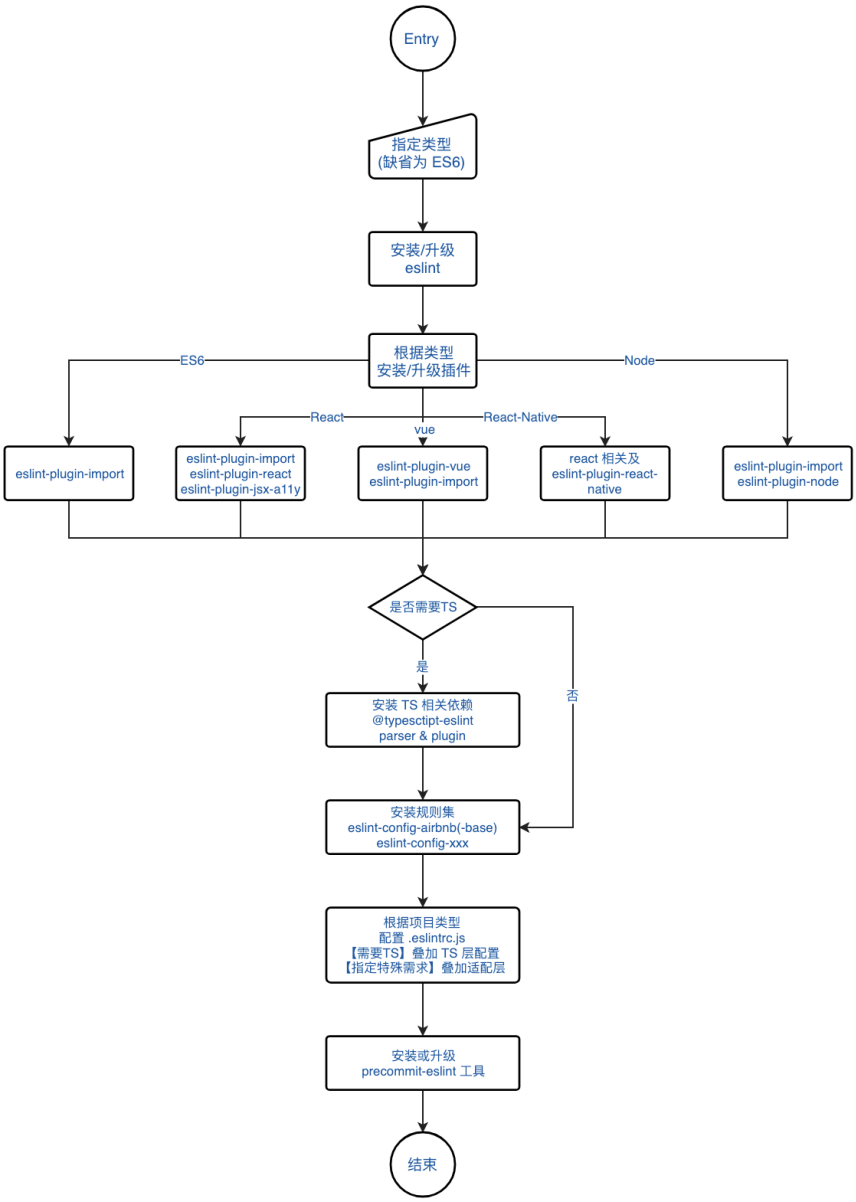
## 自动化接入工具

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：

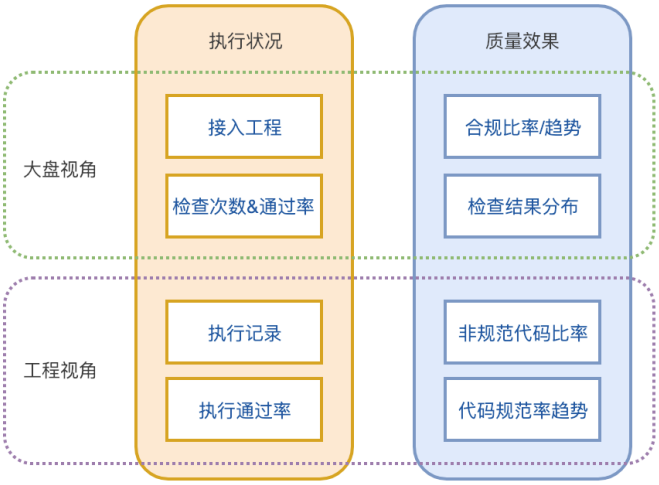



该工具流程简单，不管什么开发场景和框架选型，繁琐的接入流程都可以简化为一条命令，需要配合工程方案升级时同样如此。如下图所示，执行该命令后项目就完成了 ESLint 的接入，使用统一的规则规范编码，同是在代码提交时自动进行增量检查：

```
➜ 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-esli
nt/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：

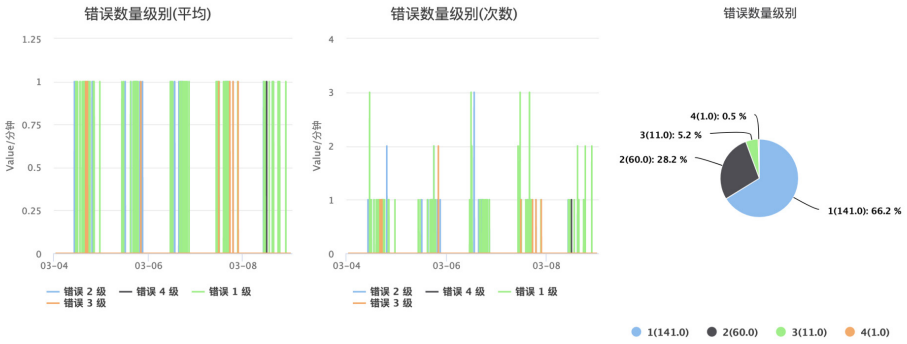


执行情况分析其实并不复杂，核心是信息采集和分析。在本方案中，信息采集通过 precommit-eslint 工具实现：在 git commit 触发本地代码检查后，脚本会把检查结果（包括检查是否通过、错误或警告信息的数量级别等）上报；信息的统计分析借助日志上报分析平台实现，美团使用的是 CAT 平台（如果团队或公司没有专门的平台，可以在上文提到的代码检测服务平台中实现这部分功能）。为了便于数据的聚合分析，我们将一次代码提交检查中出现的问题数量进行了分级：

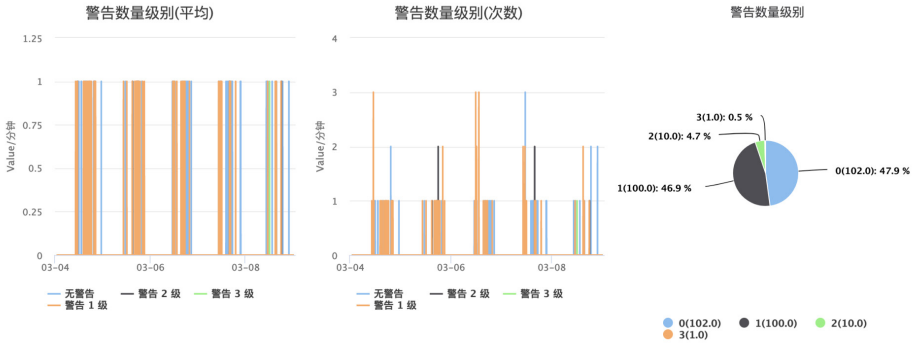
- 检查通过：检查无代码规范错误。
- 错误 1 级：检查出代码规范错误数量小于 10 个。
- 错误 2 级：检查出代码规范错误数量在 10 – 100 个之间。
- 错误 3 级：检查出代码规范错误数量在 100 – 1000 个之间。
- 错误 4 级：检查出代码规范错误数量大于 1000 个。

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：

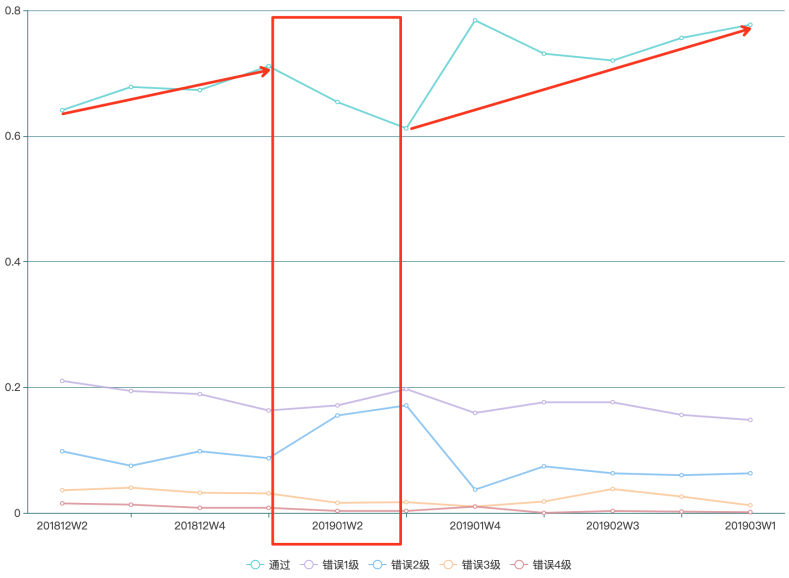
- 持续维护升级：以每月一版的方式持续迭代升级，解决应用中的问题、规则争议，以及支持新的规则或方案。
- 工程化集成：整套方案可以无缝接入到各个团队的脚手架工具中，自动成为团队的默认方案，在工程初始化阶段即可完成接入。
- 官网建设：提供详细的使用文档，包括规则信息、接入方法，并且对每个版本提供规则、环境依赖、changeLog 等详细说明。
- 常见使用问题：更新维护 FAQ，帮助后续接入者快速查找并解决问题。

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”。

# App流程管理及实践

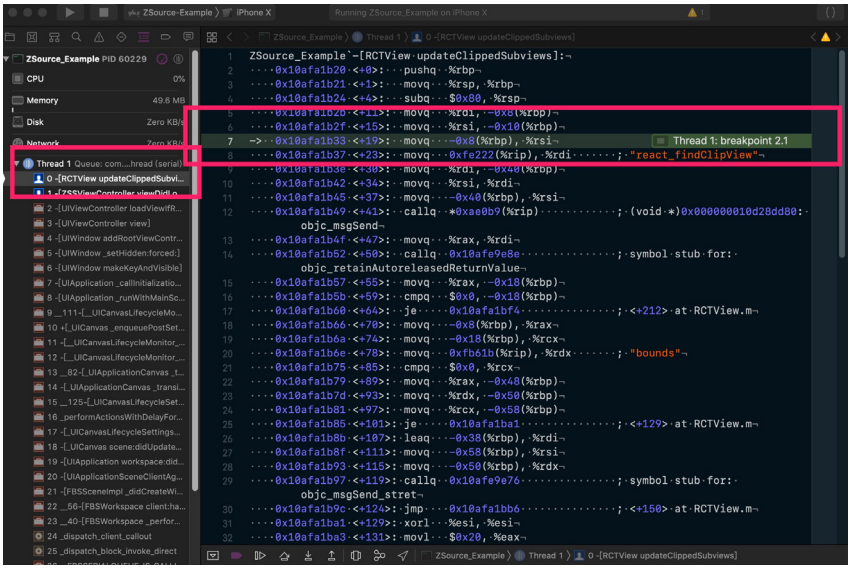
# 美团 iOS 工程 zsource 命令背后的那些事儿

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。

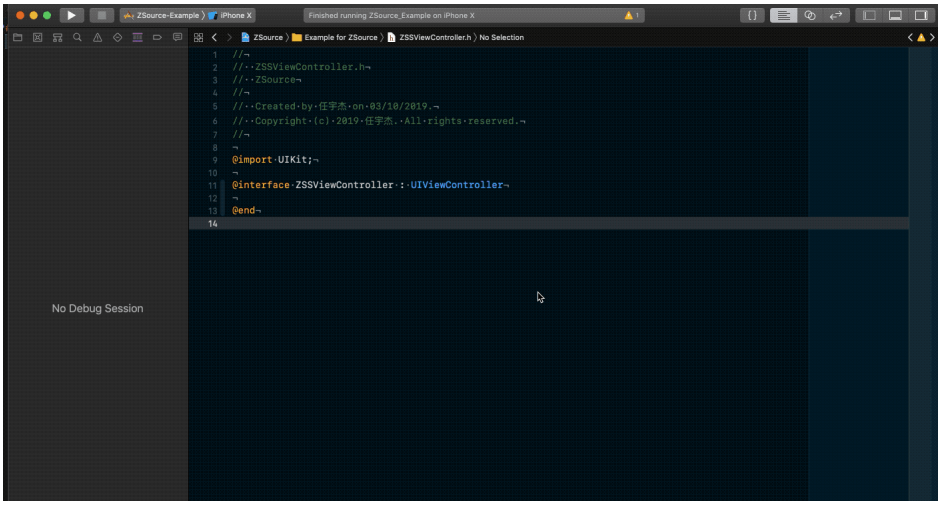
使用二进制，虽然会给工程带来构建速度的提升，但是会带来一个新的问题：在调试工程时，那些使用二进制的组件，无法像源码调试那样看到足够丰富的调试信息。例如，如果程序在二进制组件的代码中崩溃，我们只能看到该组件的堆栈信息和一些不明所以的汇编代码：




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

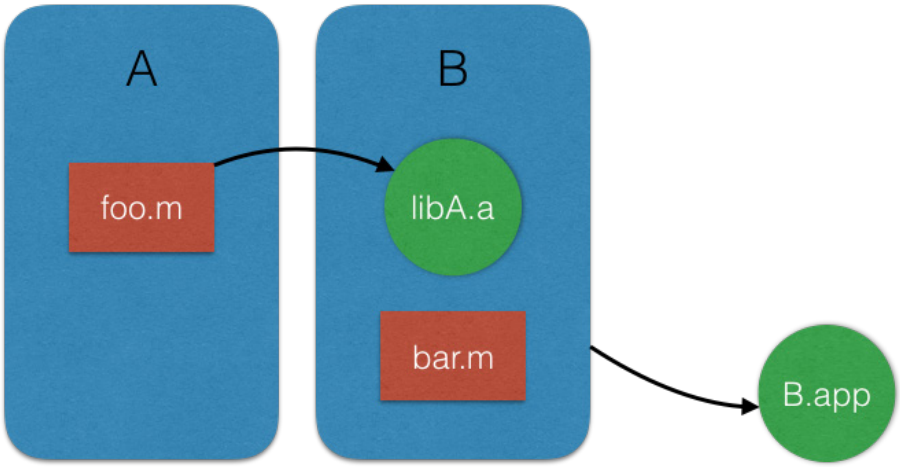
在推出 zsource 功能后，很多同学都对 zsource 背后的技术原理十分感兴趣。其实 zsource 整个功能的开发流程也十分的有趣，就像小说一样，分为几个不同的时期：

- 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：

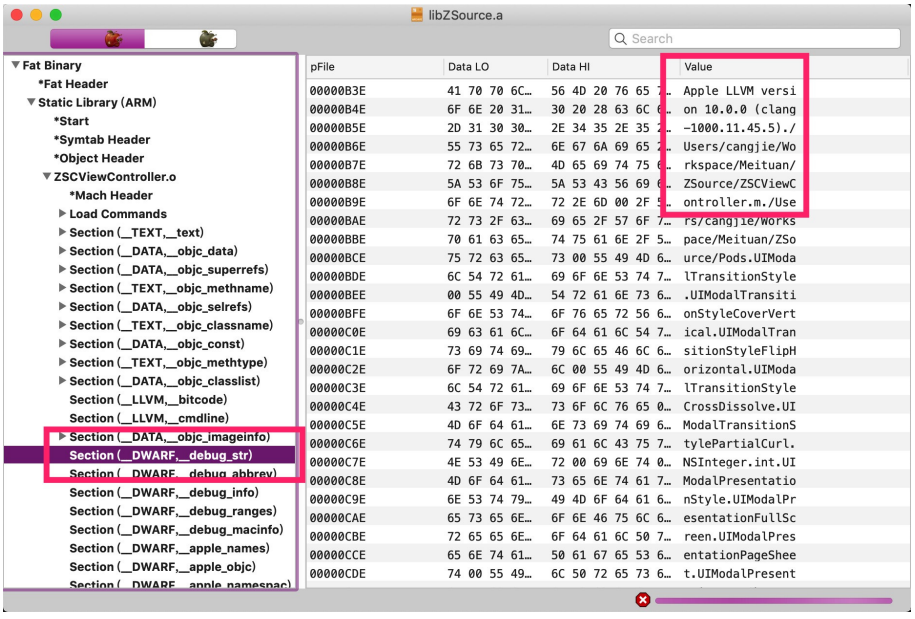
```
xxd ./libZSource.a | grep -C 5 'ZSCViewControlle'
```

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

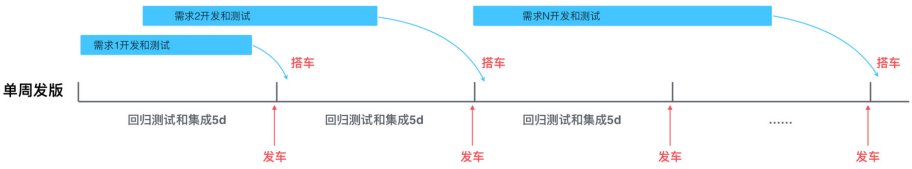
王坤

## 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。

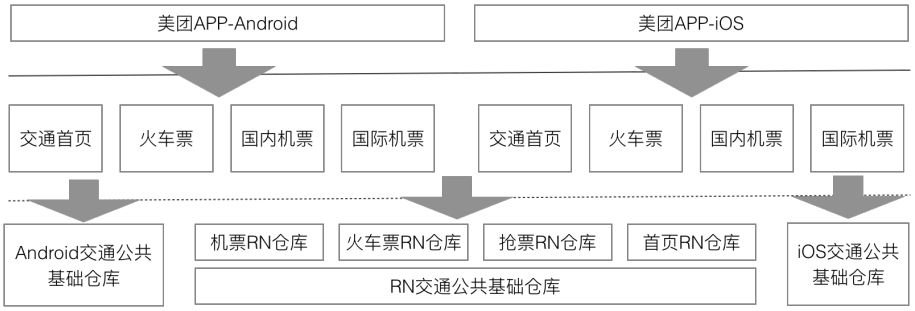
## 业务方研发人员的痛点

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

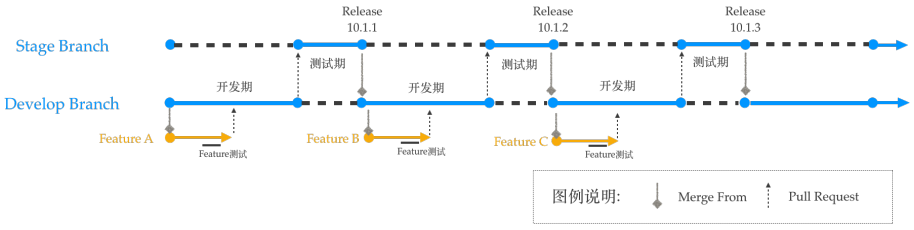
- **业务线自身的公共基础库需求变动频繁。也需要具备单周发版的能力。**

  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。

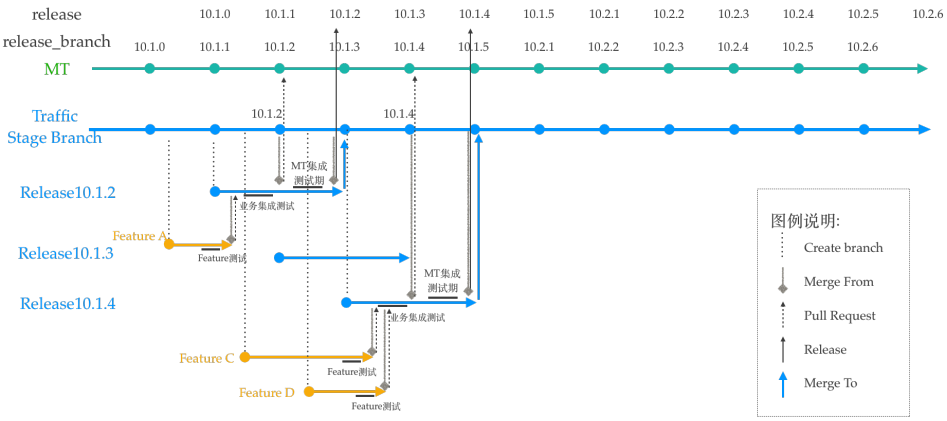
在实施单周发版之前，业务方代码仓库只有两个分支，Develop 分支，即开发分支；Stage 分支，即发版分支；开发流程基本在串行开发模式，每个版本 10 天开发，8 天测试，然后进入下一版本的开发。




之前的串行发版模式

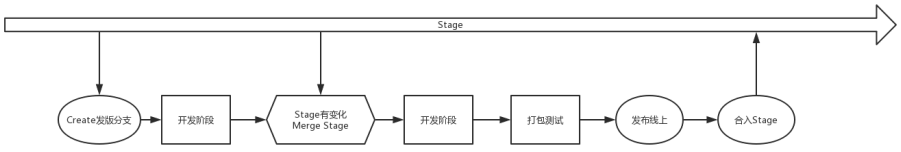
这种方式只能适用于节奏固定的长周期开发方式，对于多版本并行开发来说，有点力不从心，显然已经不能承载当前更灵活的发版节奏。

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

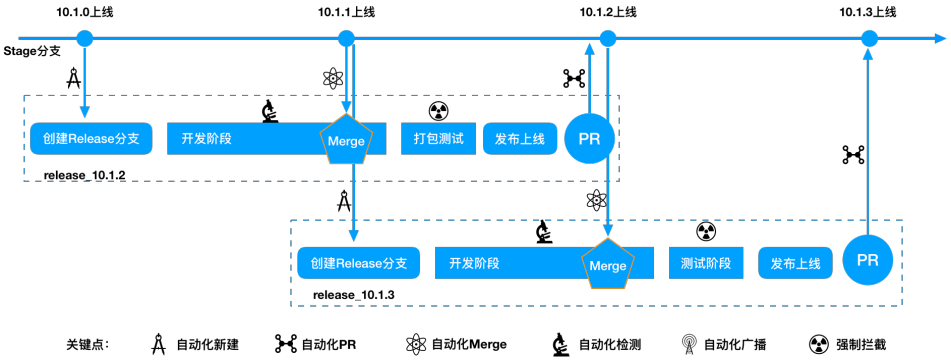
上面的流程描述可能有些复杂，下面是简化的流程图，每个版本都有自己的生命周期：




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。

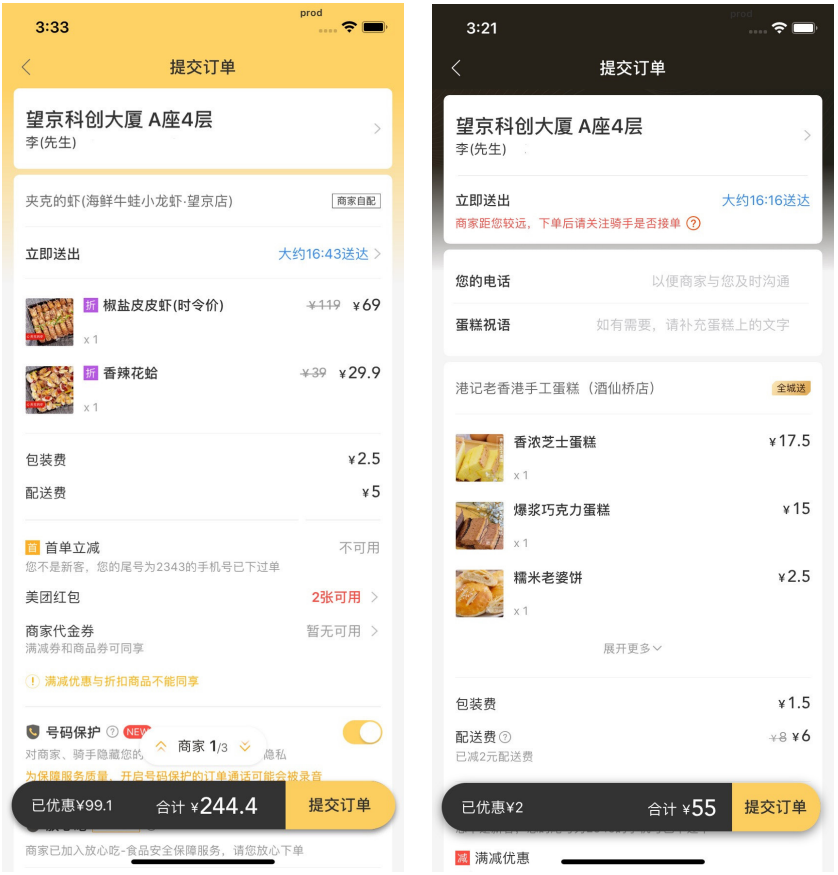
# 美团外卖前端容器化演进实践

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。

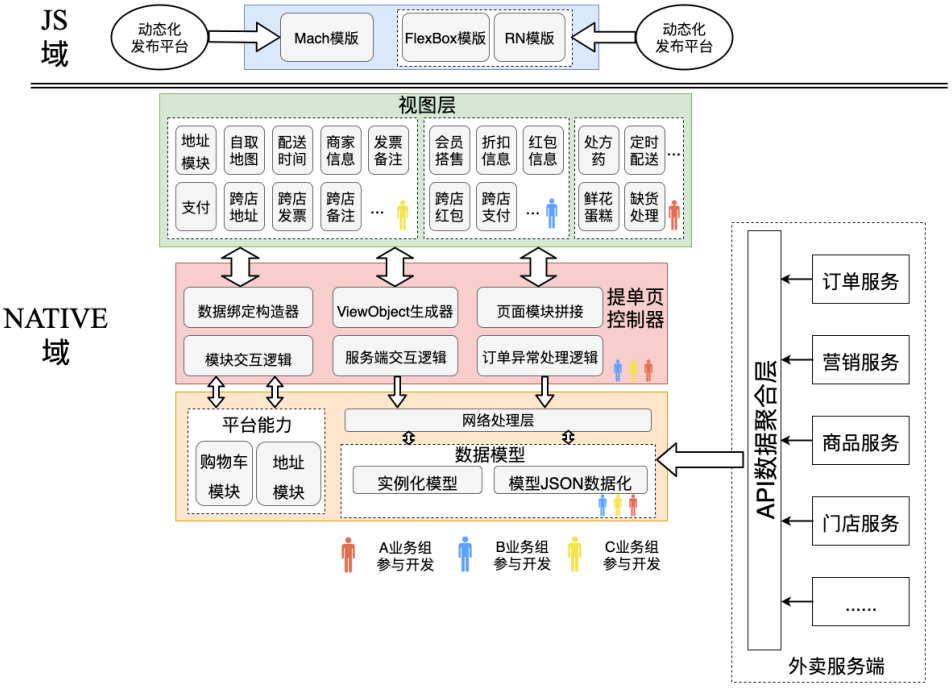
### 闪购侧业务

- 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。

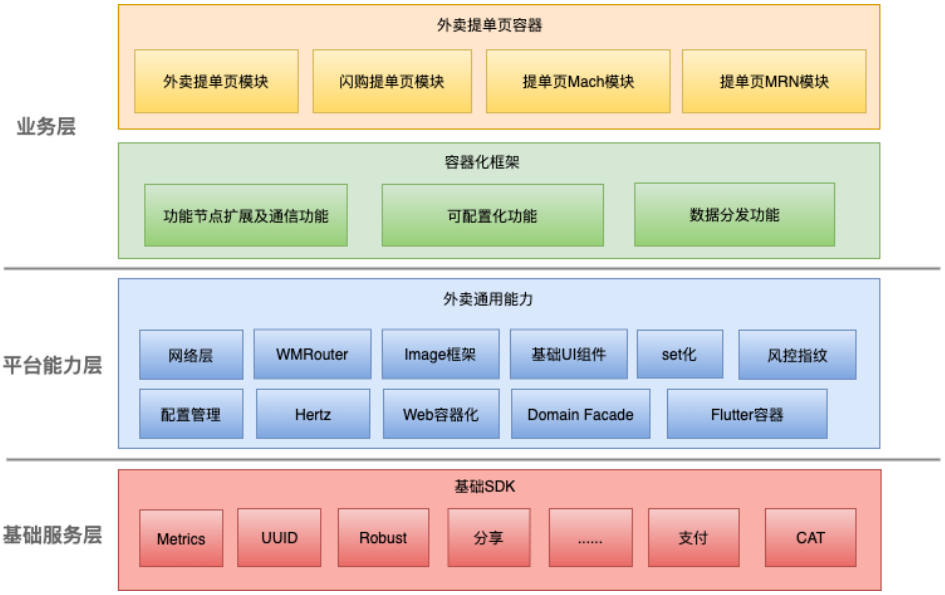

容器化后的提单页，需要实现模块之间的互相无感知，根据服务端的下发数据，客户端可以将闪购代码仓库内的模块和外卖代码仓库内的模块拼接起来组成完整的提单页展示给用户。当用户在提单页完成一系列操作时，各模块可以提供必要的参数给服务端。要想实现这一点，我们需要考虑以下几个问题：

- 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

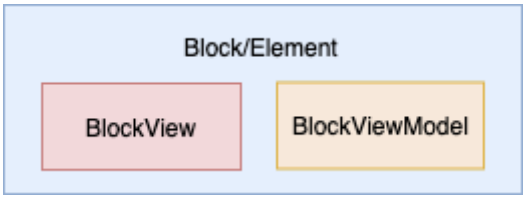



容器化是我们在外卖平台化之后对多方业务能力的支持和扩展，在不改变 API 数据源等前提下，我们保证其具有动态可配置化的能力。为了更好地支撑业务，我们在业务层面抽离出来容器化框架层，其所提供三个部分的核心功能：1. 功能节点扩展及通信功能；2. 可配置化功能；3. 数据分发功能。在最上层业务容器中，目前所支持外卖提单页面模块、闪购提单页模块、提单页 Mach（外卖动态化模板）模块、提单页 MRN（RN 页面）模块四种不同的业务。

#### 1.1 概念解释

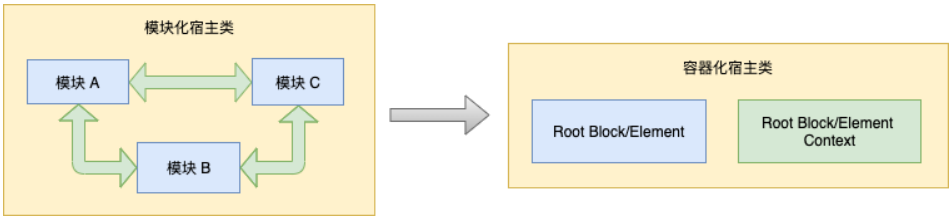
在容器化框架设计过程中，我们引入了一些新的定义，比如在 Android 端引入了 Block 的概念，这里的 Block 是一个功能模块的简称。在提单页页面中，我们可以理解为一个 Block 对应一个功能条目。在 iOS 端有与之对应的概念 Element（由于

两者没有差异，下文陈述中用 Block 代指两者）。



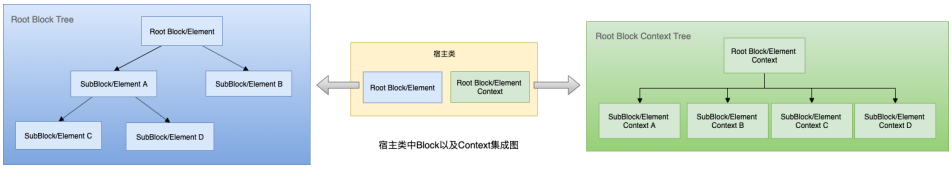

Block 有两种类型：其一是普通的 Block，其包含 BlockView（视图层）和 BlockViewModel（数据层）。BlockView（视图层）用来展示具体的视图以及内部的业务逻辑；BlockViewModel（数据层），用来数据解析。其二是 LogicBlock，是没有视图的 Block，单纯地用来做数据业务处理。

## 1.2 整体概述

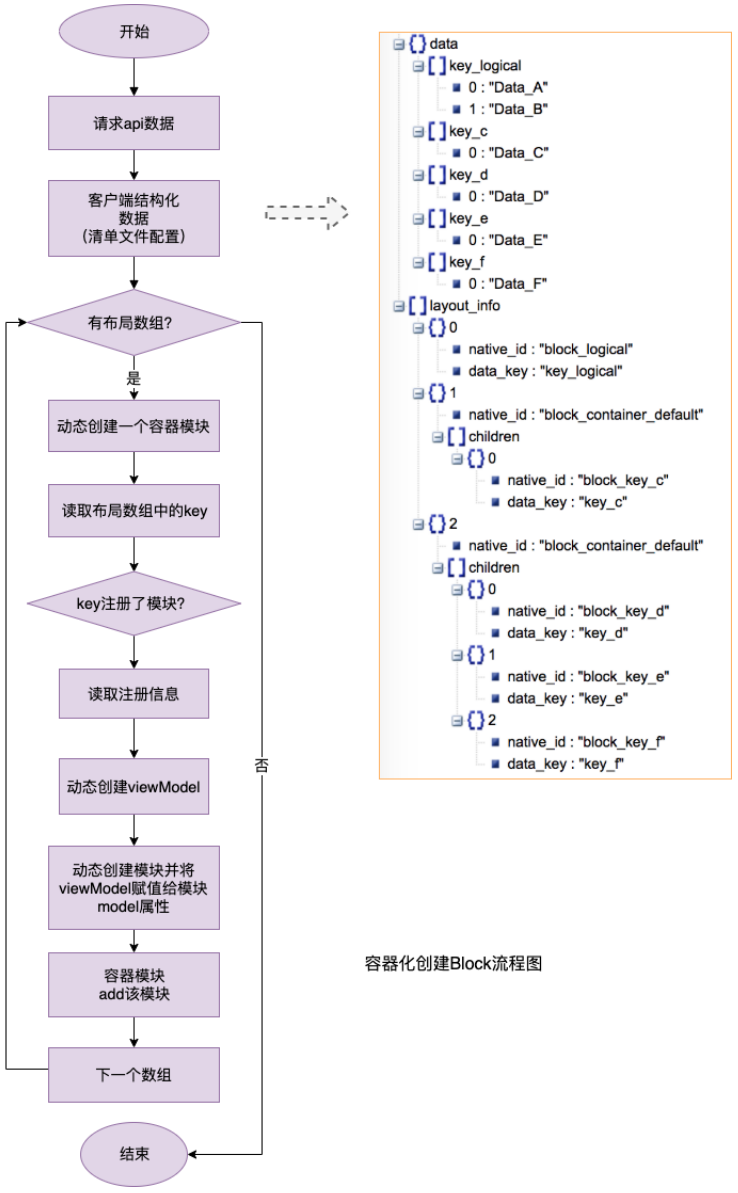



在容器化之前，我们的业务大多是模块化的结构，模块化宿主类是承载所有模块化的管理类，各个模块之间通过宿主类或者控制器进行数据交互。但在容器化改造中，我们将之前宿主类中管理的模块进行拆解，并重新定义了宿主类的职责。在容器化宿主类中，我们将不再持有各个功能模块的引用，而只要持有 Root Block 这一个实例，就可以完成对所有功能模块的管理。而 Root Block Context 则用来处理父 Block 与子 Block 之间的通信以及子 Block 之间的通信。

## 1.3 核心功能

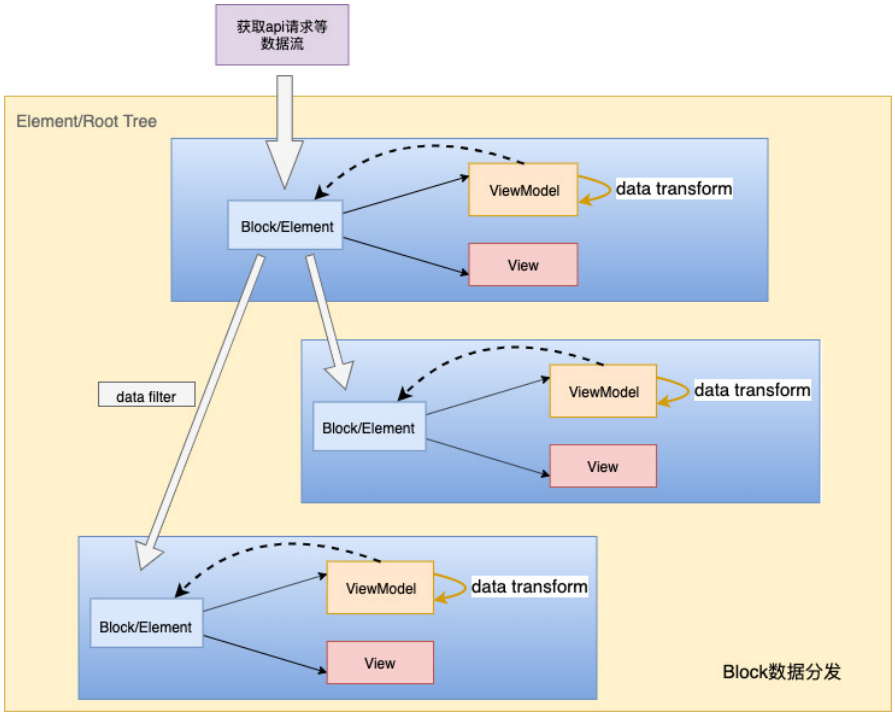


第一部分功能节点扩展及通信功能。主要是目前页面的集成和通信关系，其中 Root Block 是 Block Tree 的根节点，下面会挂载一些 SubBlock 子节点，Root Block 会控制整体的数据流的分发以及整体样式；Root Block Context 可以理解为上下文环境或通信的总线。每个模块都有自己的 Context，来维护自己向外部提供数据以及业务逻辑的能力，这些子 Context 会统一注册到 Root Context 中进行管理维护。




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。

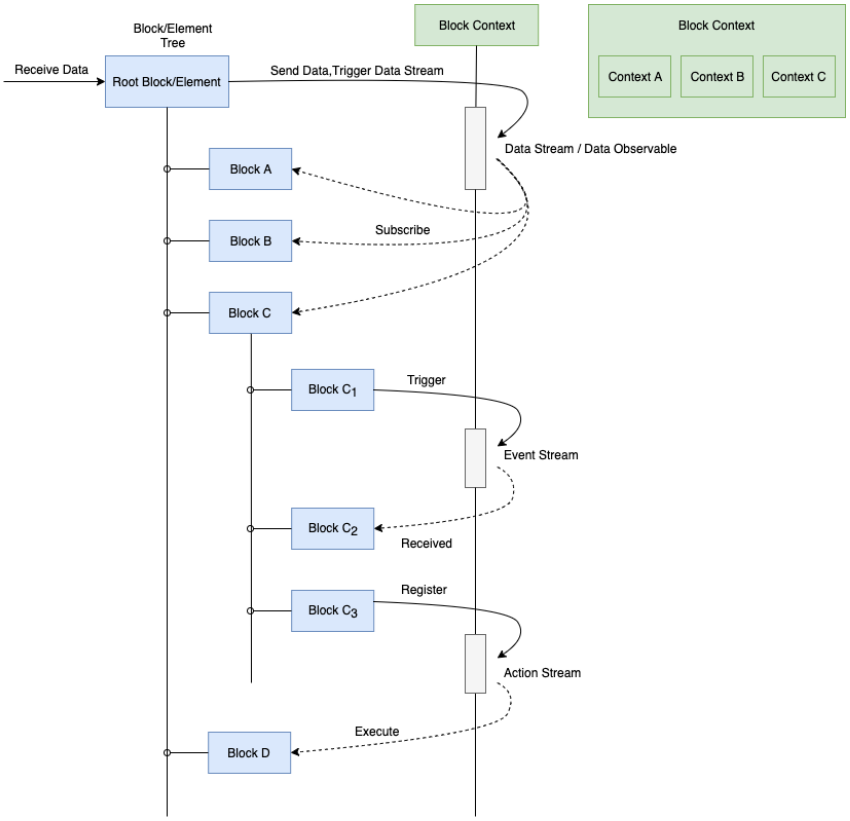
### 4. 模块间通信问题

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。

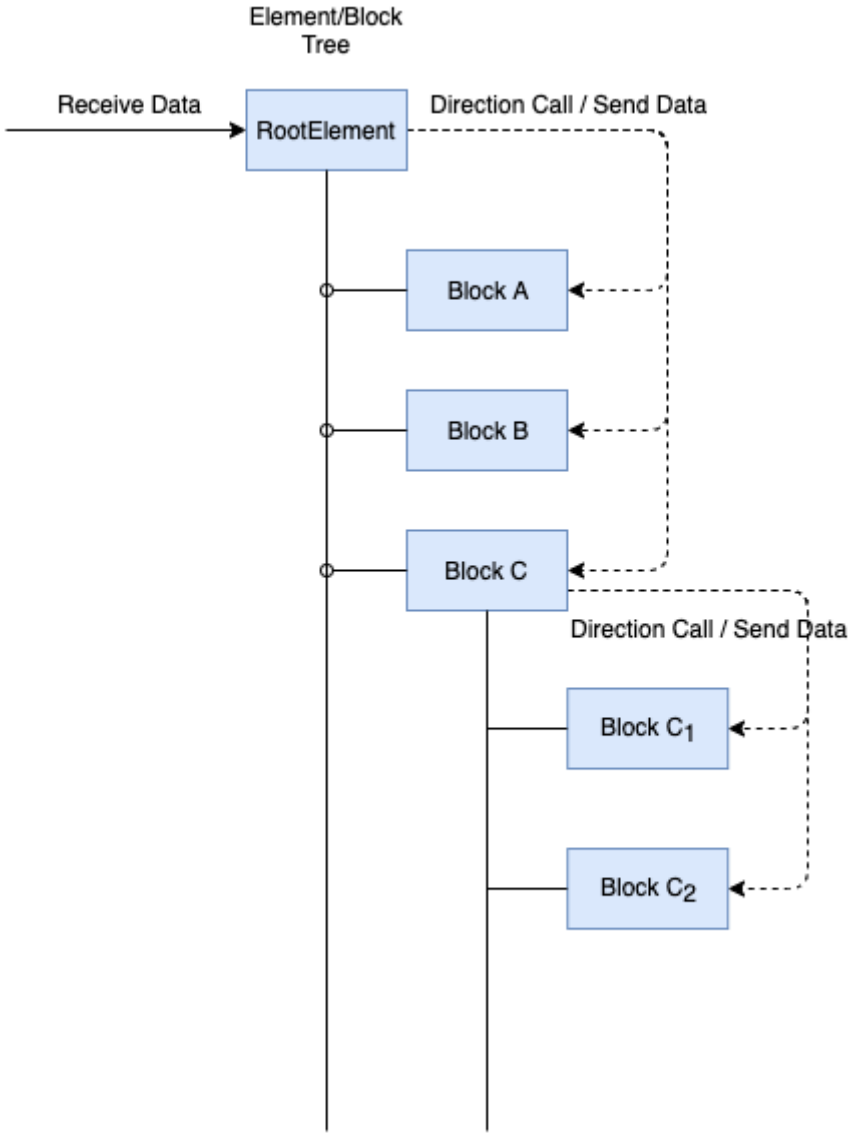
```
// 声明事件容器
private SupplierEvent mSupplierEvent = new SupplierEvent();
@Override
public SupplierEvent supplierResponseEvent() {
  return mSupplierEvent;
}
// 实现订阅
context().supplierResponseEvent().subscribe(new Action() {
   @Override
   public void action() {
   }
});
// 触发相应的操作
context().supplierResponseEvent().trigger();
```

## 5. Block 页面数据分发问题

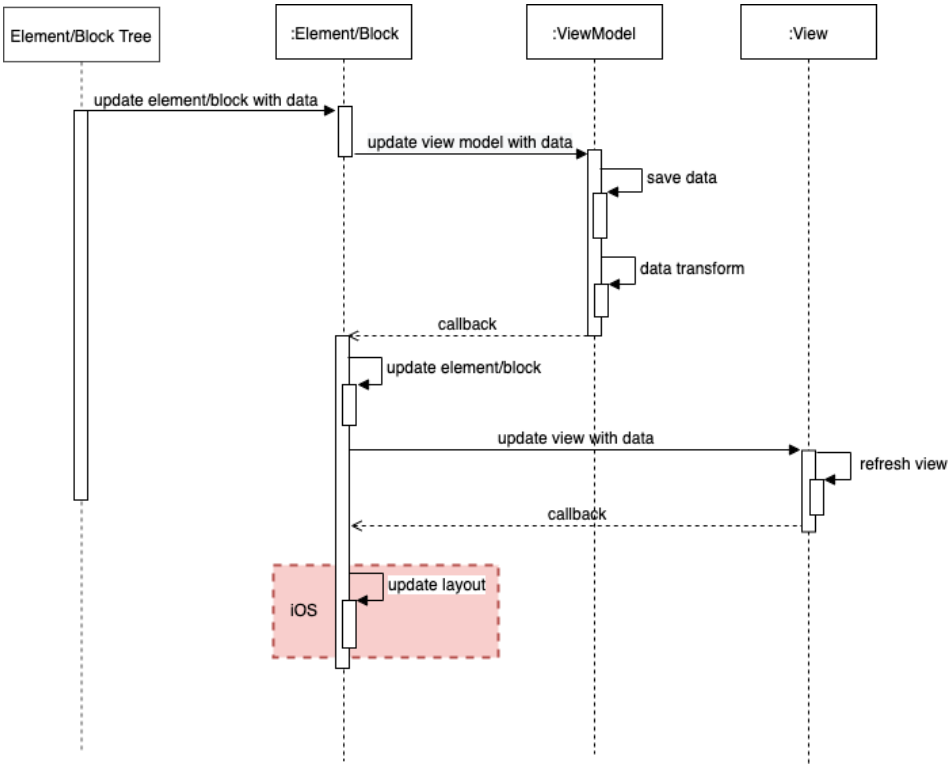
### 5.1 数据分发问题

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

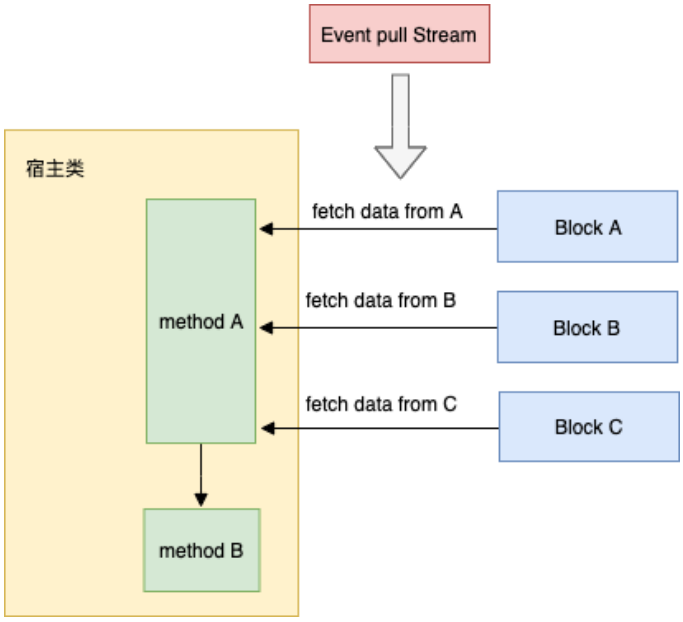
Block 创建的顺序由 API 结构化数据中的 layoutInfo 数组来决定，layoutInfo 数组的具体格式如第三节 API 数据结构化中内容所示。容器化后的提单页会根据 layoutInfo 数组的顺序，依次创建对应 native_id 的 Block 模块。因此，对于一些基础公共模块（比如 wm_confirm_order_logical 对应的 Block），我们可以将其放在 layoutInfo 数组的最前面让其提前加载，保证负责 UI 展示的 Block 创建时数据可用。

## 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?

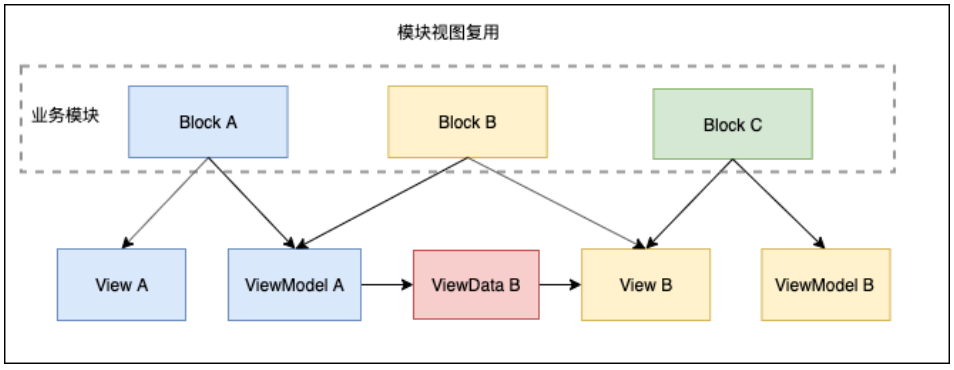
如上面的通信设计思路，我们利用 Event 数据交互方式，从各个模块中将需要

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

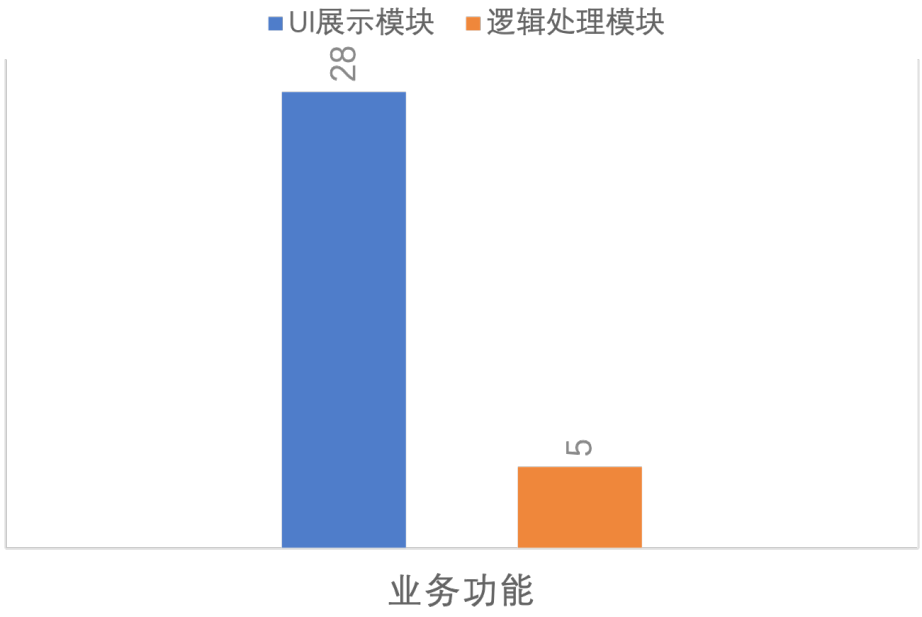
在实际的开发中，有些 Block 的页面 View 大致上相似，但是逻辑上有些细微的差异，为了快速开发，我们在设计上复用了其视图。Block、BlockView 以及 ViewModel 的关系：一个 Block 对应一个 ViewModel 和一个 BlockView，一个 ViewModel 和一个 BlockView 可以对应多个 Block。

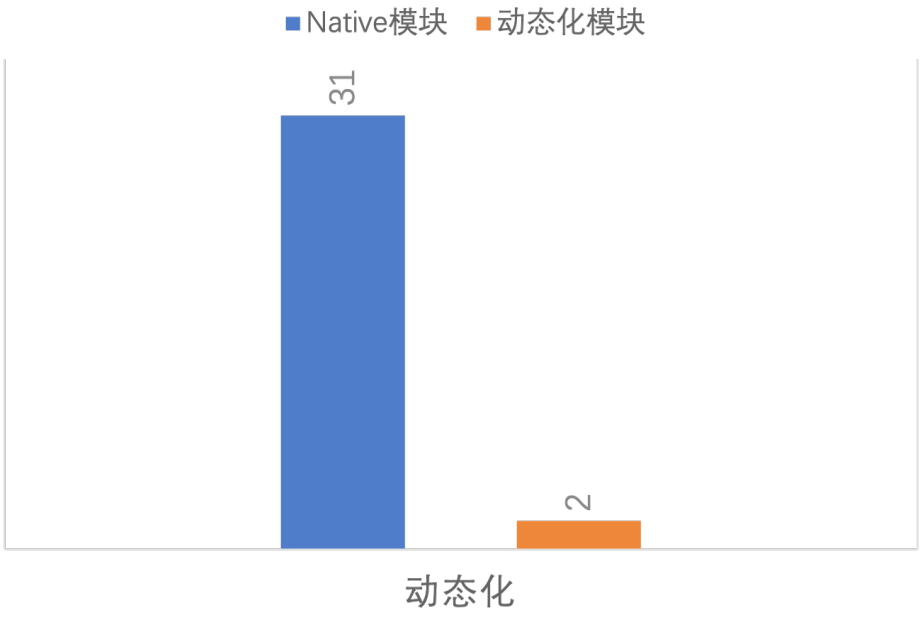


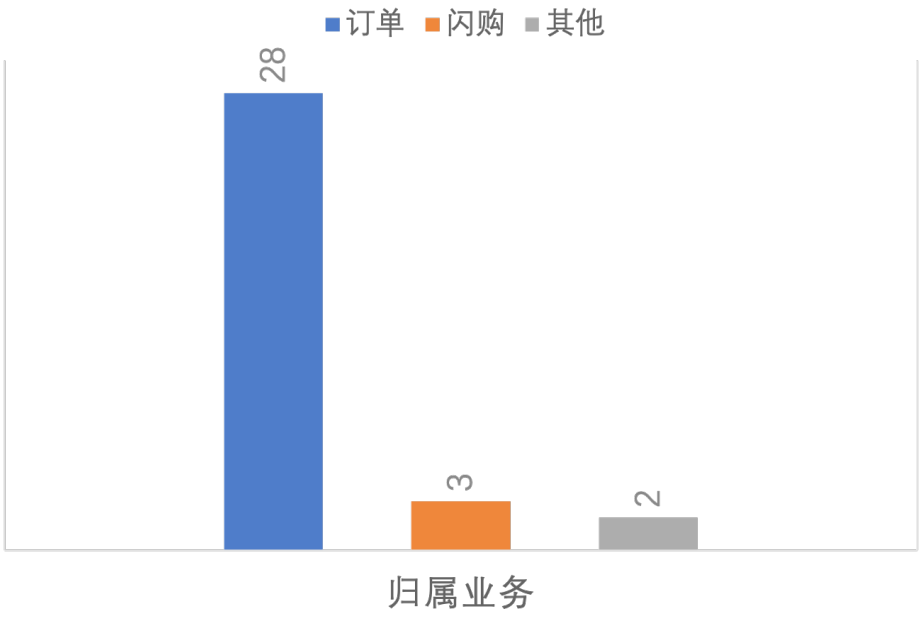
计算机界有一句名言：“计算机科学领域的任何问题都可以通过增加一个中间层来解决。”（原始版本出自计算机科学家 David Wheeler）相似的，为了视图层的复用，屏蔽数据层的差异，我们在数据层的逻辑中转部分引入一个中间层 ViewData，ViewData 是为了更好地适配数据模型以及区别视图展示上的差异，这样就大大提高了代码的复用率。

## 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

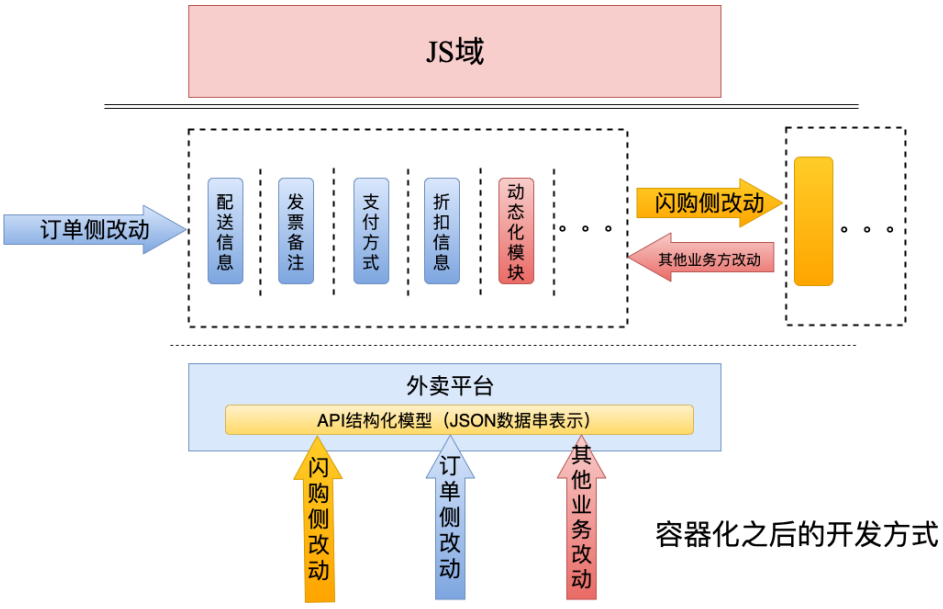
自由选择，相互之间做到了完全无感知。这些优点为后续提单页的业务迭代和技术优化都提供了很大的空间。

## 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。

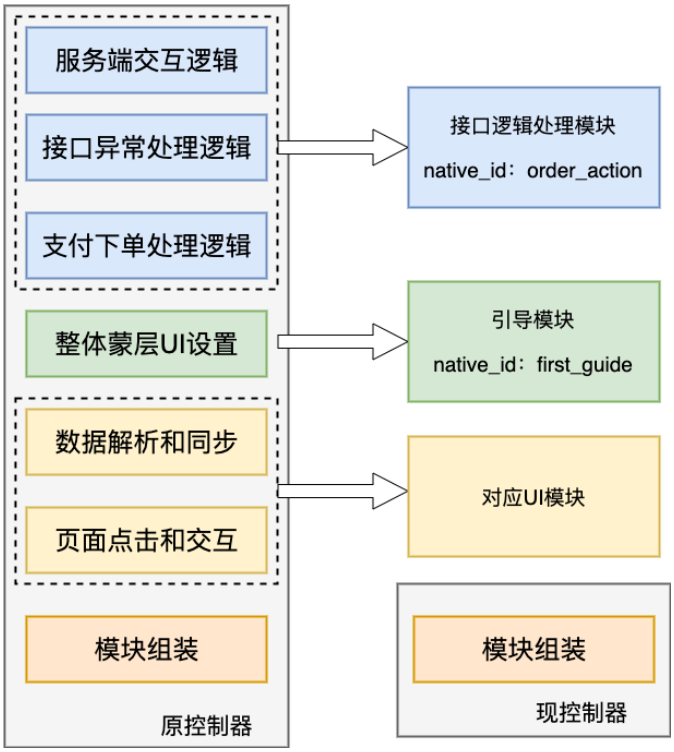



容器化之后的开发方式

### 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。

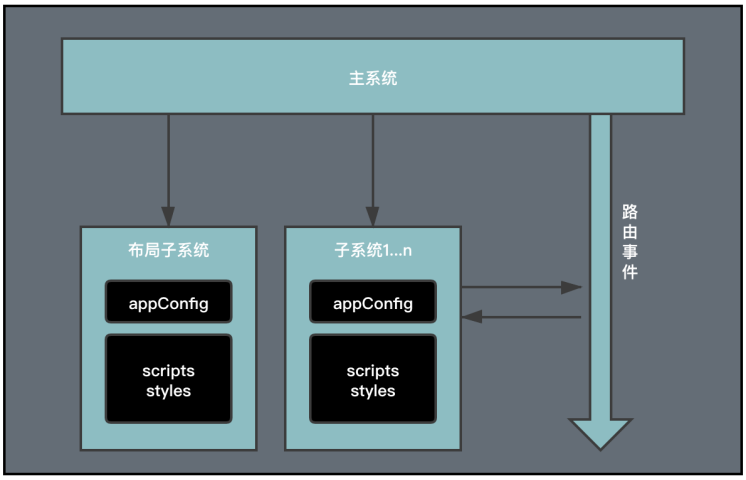
这些方案的优劣整理如下：

| 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。

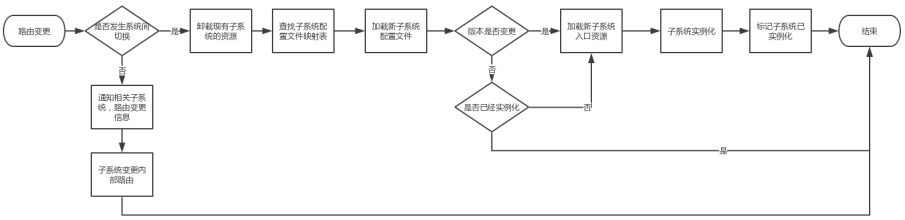



主系统是用来控制子系统的调度中心，职责包括：

- 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。

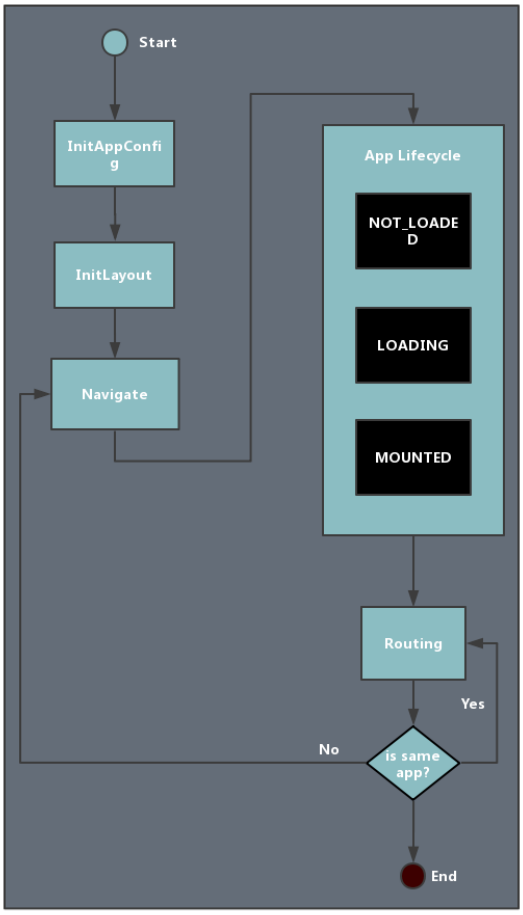
子系统只负责业务逻辑的实现。如果进一步细分的话，子系统可以分为业务子系统、实现公共菜单子系统、导航布局子系统，其中布局子系统会先于业务子系统加载。

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。



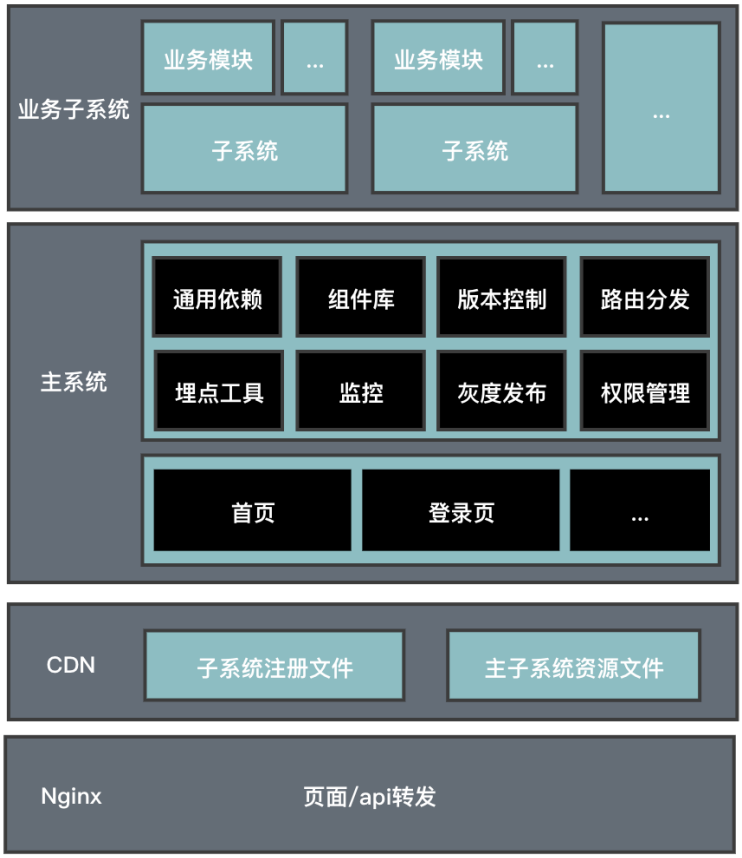

如果子系统发生切换，主系统会在 DOM 中添加对应子系统的挂载节点，并异步加载系统的静态资源。由于子系统都是完整的 Vue 实例，当子系统的代码加载并执行之后，子系统就会自动在其对应的挂载节点上渲染相应的内容。

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。

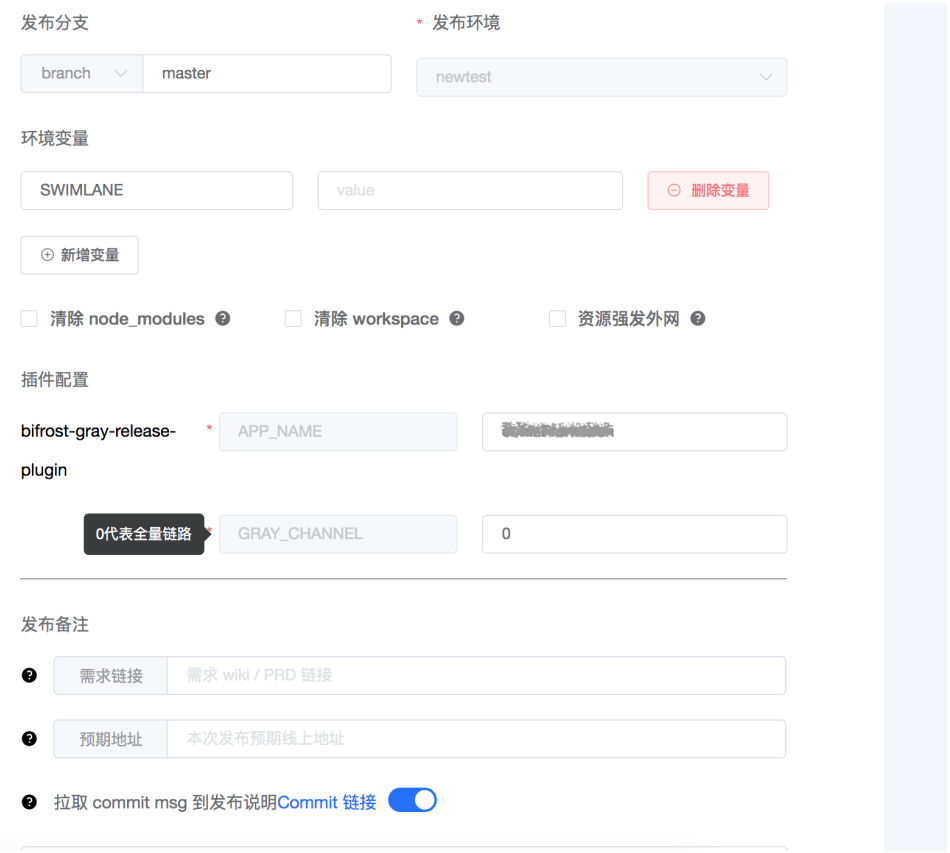
## 发布及部署流程

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）

# 基本功

# Litho 的使用及原理剖析

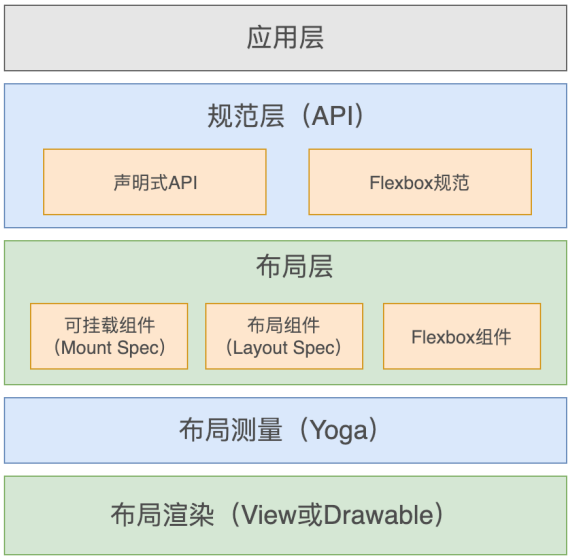
少宽　张颖

## 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();
    }
  }
```

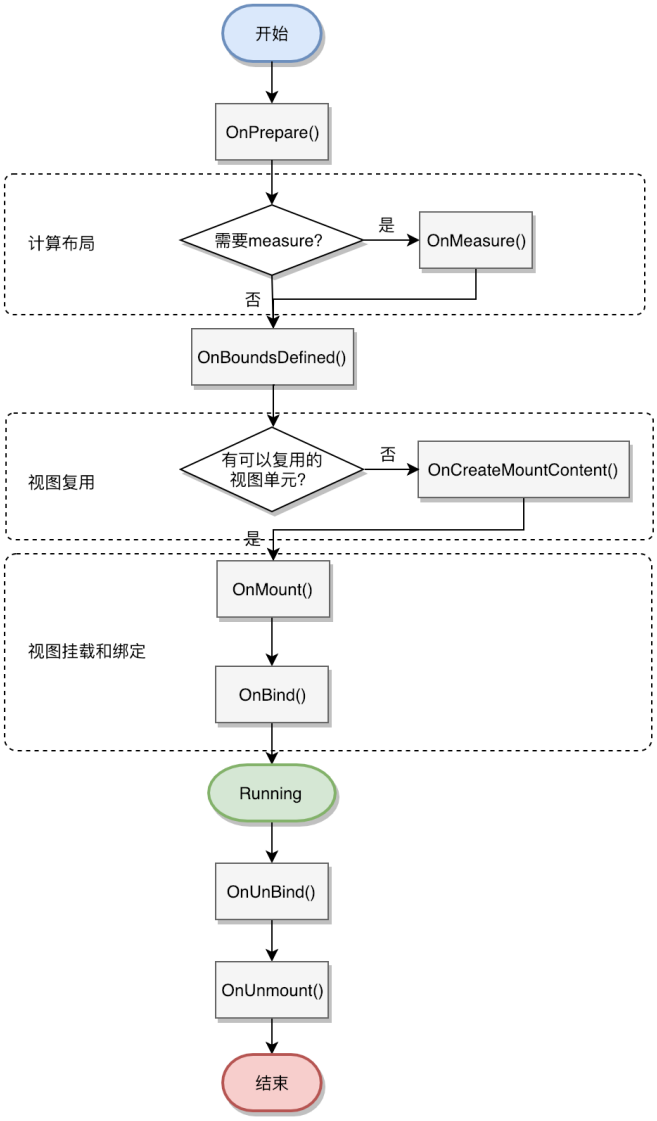
最终 Litho 会在编译时生成一个名为 HelloComponent 的组件。

```
  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。

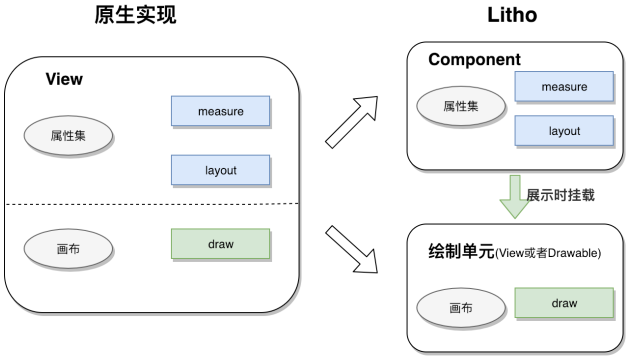
### 异步布局原理剖析

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，

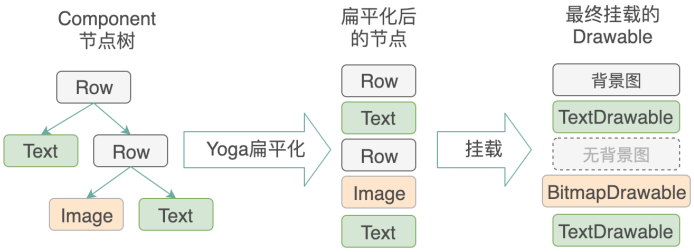
同样的样式，使用 Litho 实现的布局要比使用 Android 原生实现的布局更加扁平。


### 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?

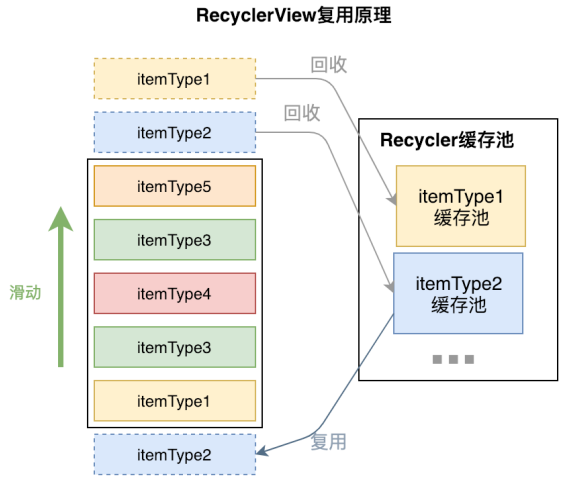
Litho 可以更好地实现复杂布局。约束布局虽然可以实现扁平效果，但是它使用了大量的约束来固定视图的位置。随着布局复杂程度的增加，约束条件变得越来越多，可读性也变得越来越差。而 Litho 则是对 Flexbox 布局进行的扁平化处理，所以实际使用的还是 Flexbox 布局，对于复杂的布局 Flexbox 布局可读性更高。

### 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。

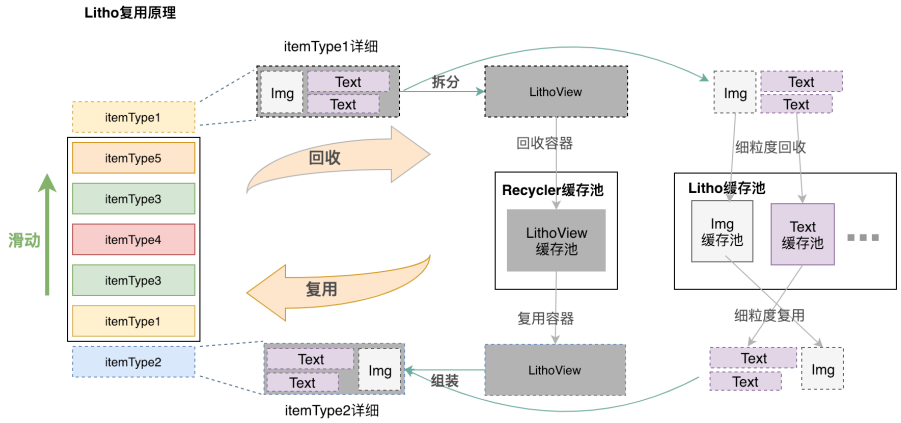
### 原生 RecyclerView 复用原理剖析

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：

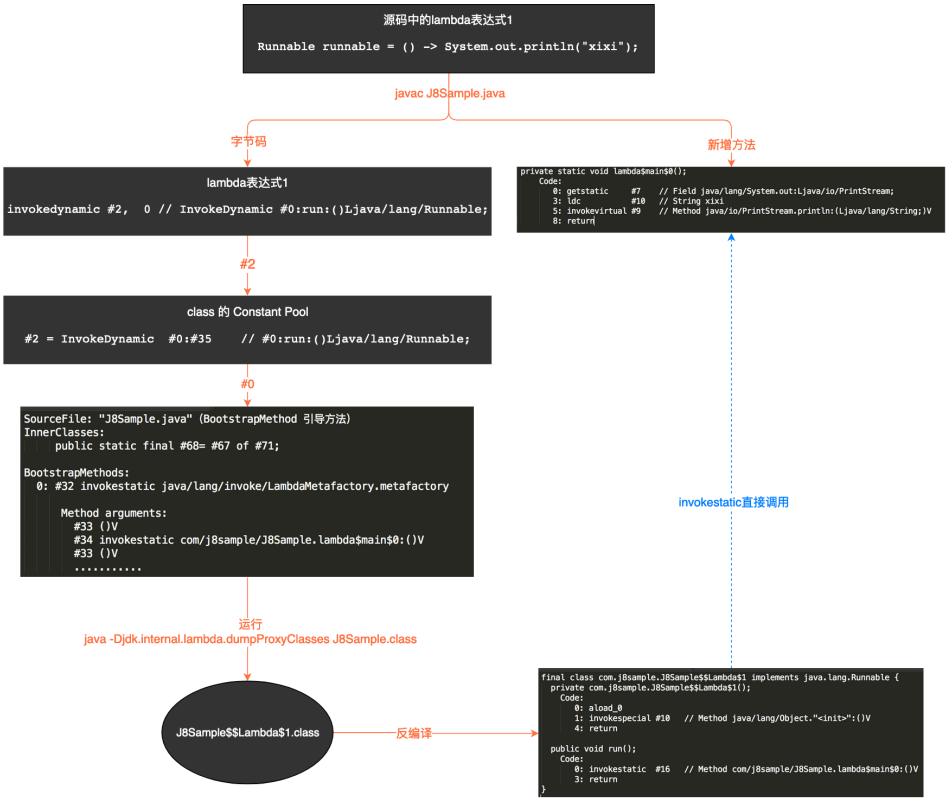
- 后四种指令，在编译期间生成的 class 文件中，通过常量池 (Constant Pool) 的 MethodRef 常量已经固定了目标方法的符号信息（方法所属者及其类型，方法名字、参数顺序和类型、返回值）。虚拟机使用符号信息能直接解释出具体的方法，直接调用。
- 而 invokedynamic 指令在编译期间生成的 class 文件中，对应常量池 (Constant Pool) 的 Invokedynamic_Info 常量存储的符号信息中并没有方法所属者及其类型，替代的是 BootstapMethod 信息。在运行时，通过引导方法 BootstrapMethod 机制动态确定方法的所属者和类型。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
}
```

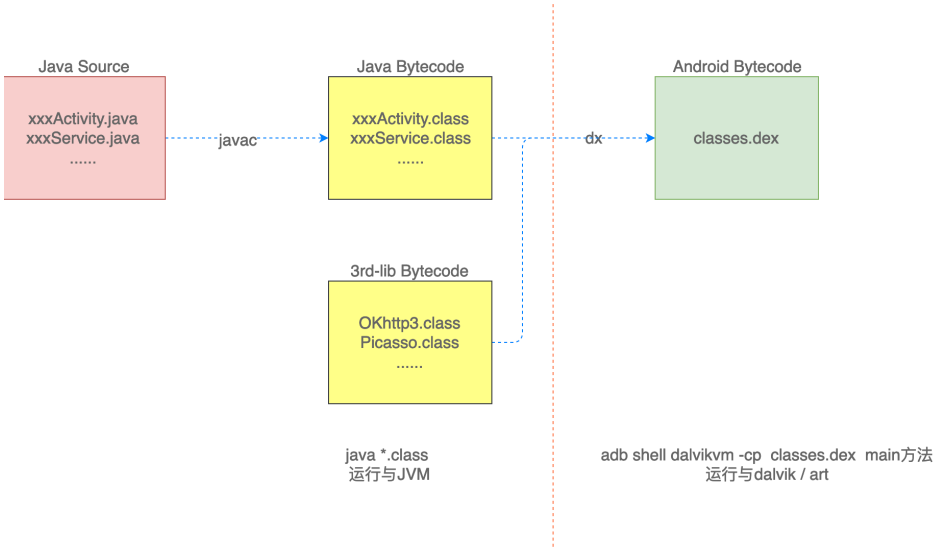
在 run 方法中使用了 invokestatic 指令，直接调用了 J8Sample.lambda$-main$0 这个在编译期间生成的静态私有方法。

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：

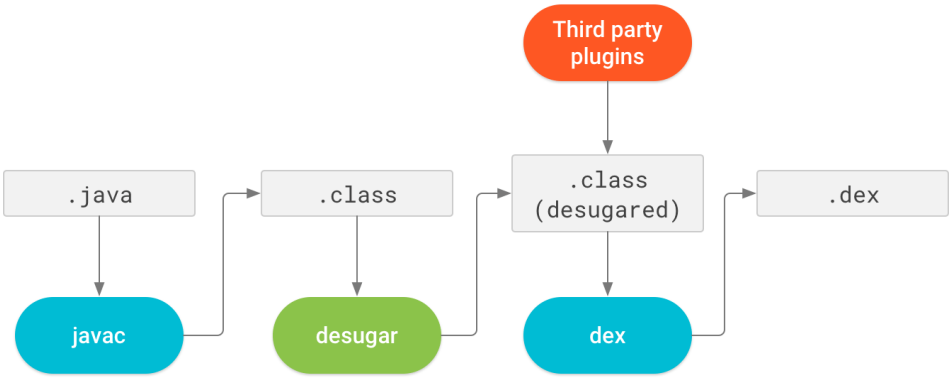


通过 Lambda 这节，我们知道 Java 底层是通过 invokedynamic 指令来实现，由于 Dalvik/ART 并没有支持 invokedynamic 指令或者对应的替代功能。简单的来说，就是 Android 的 dex 编译器不支持 invokedynamic 指令，导致 Android 不能直接支持 Java 8。

## 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：

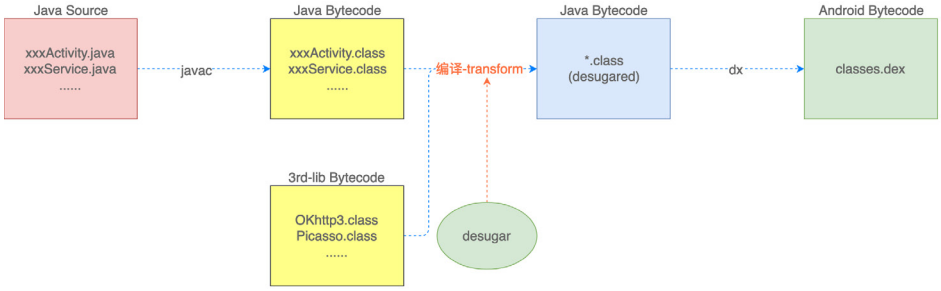



当前，无论是 RetroLambda，还是 Google 的 Jack & Jill 工具，还是最新的 D8 dex 编译器：

- 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



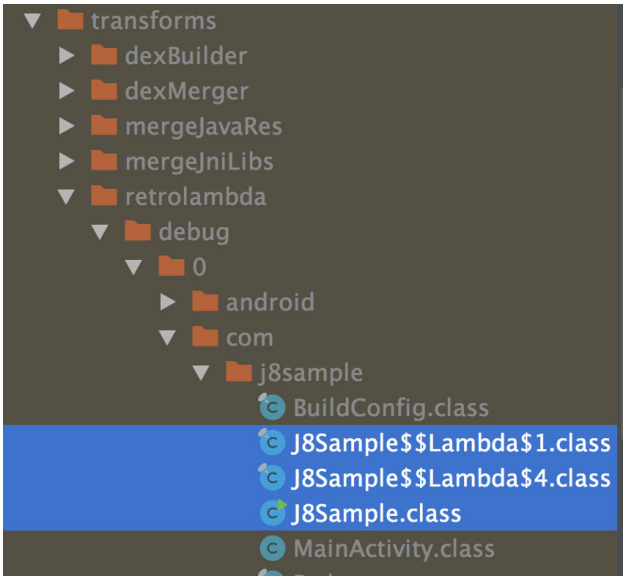

如图所示，RetroLambda 的 Desugar 过程发生在 javac 将源码编译完成之后，dx 工具进行 dex 编译之前。

### 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：

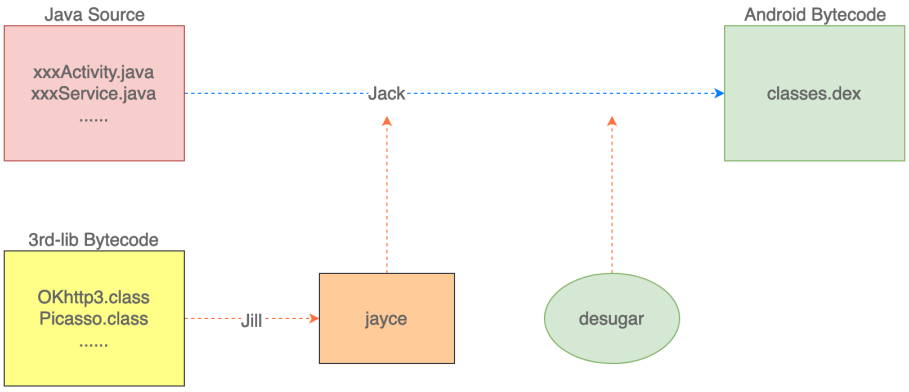
```
package com.j8sample;

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。

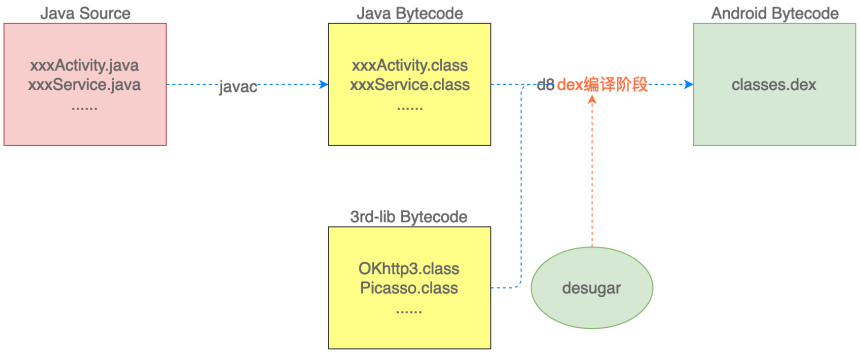
注意：右图中 J8Sample.lambda$main$0() 方法在左图中没有显示出来，但是 J8Sample.class 字节码确实是存在的。

## 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。

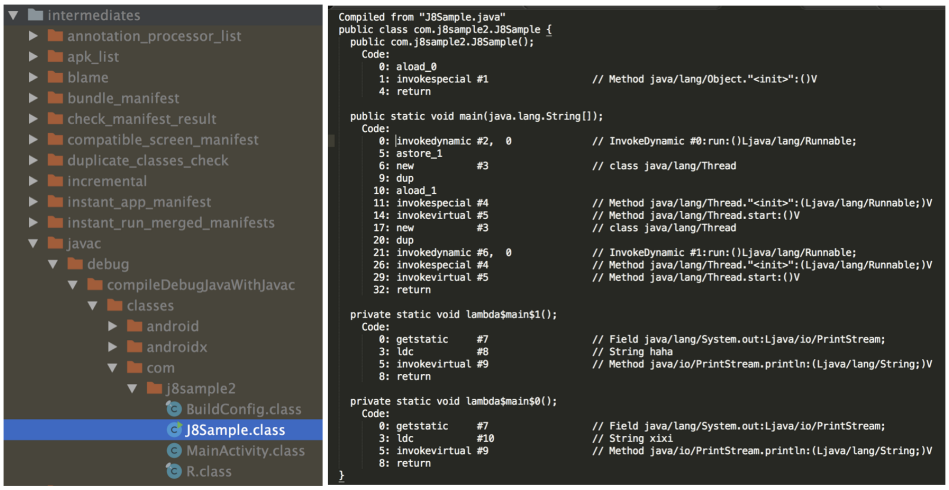
## D8 Desugar

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。

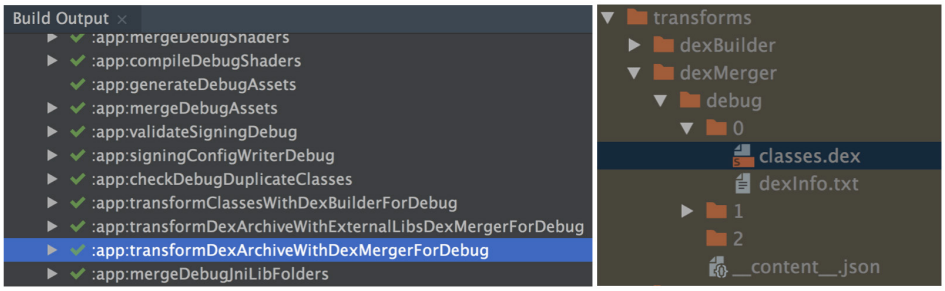
本质上也是参照 java/lang/invoke/LambdaMetafactory.metafactory 方法直接将原本在运行时生成在内存中的 J8Sample\$\$Lambda\$1.class，在 D8 的编译 dex 期间，直接生成并写入到 dex 文件中。

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。

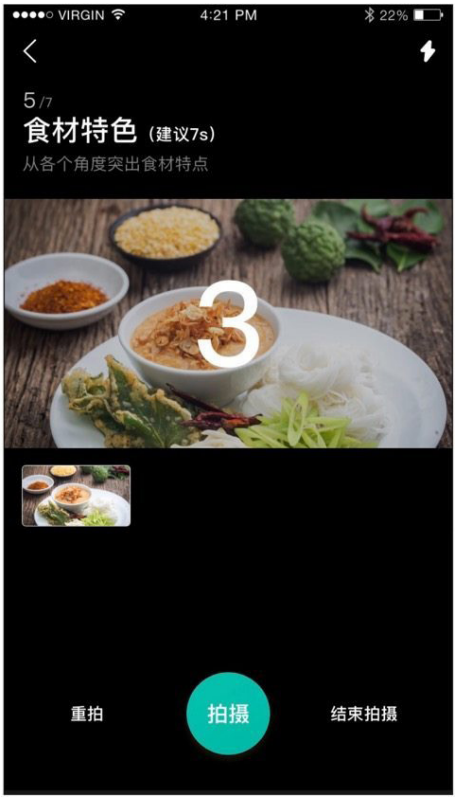
至于 Java 8 的其他特性呢，D8 是如何实现的，也可以按照上面类似的方式去分析，甚至可以结合 Kotlin 实现的方式，一探究竟。

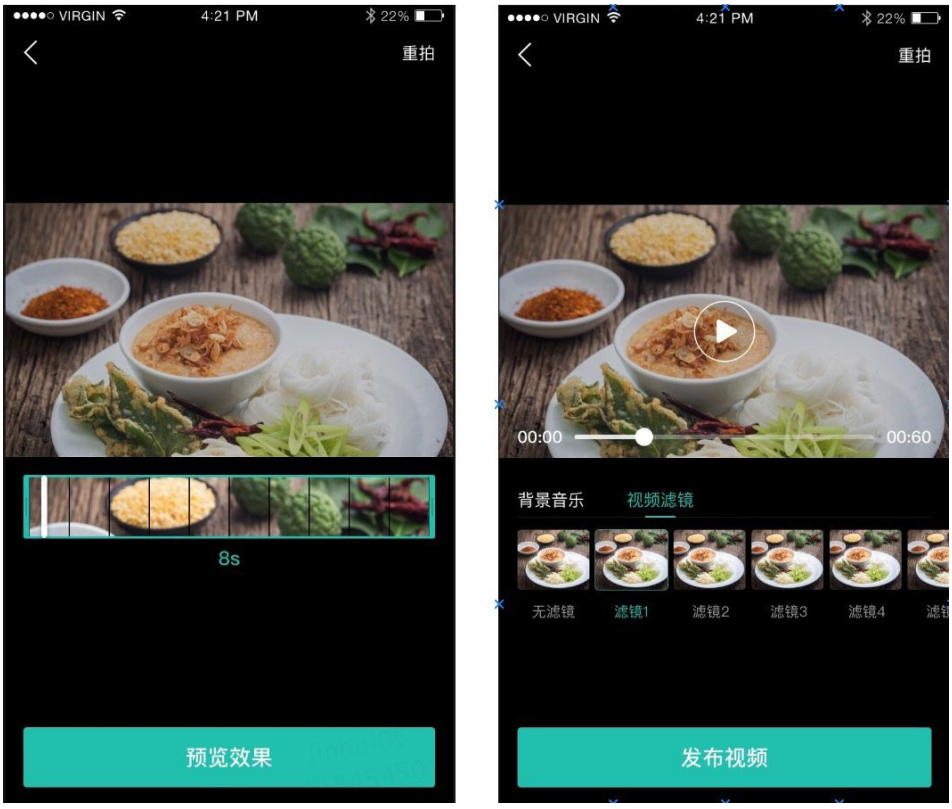
# 美团外卖商家端视频探索之旅

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<br>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<br>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<br>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<br>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

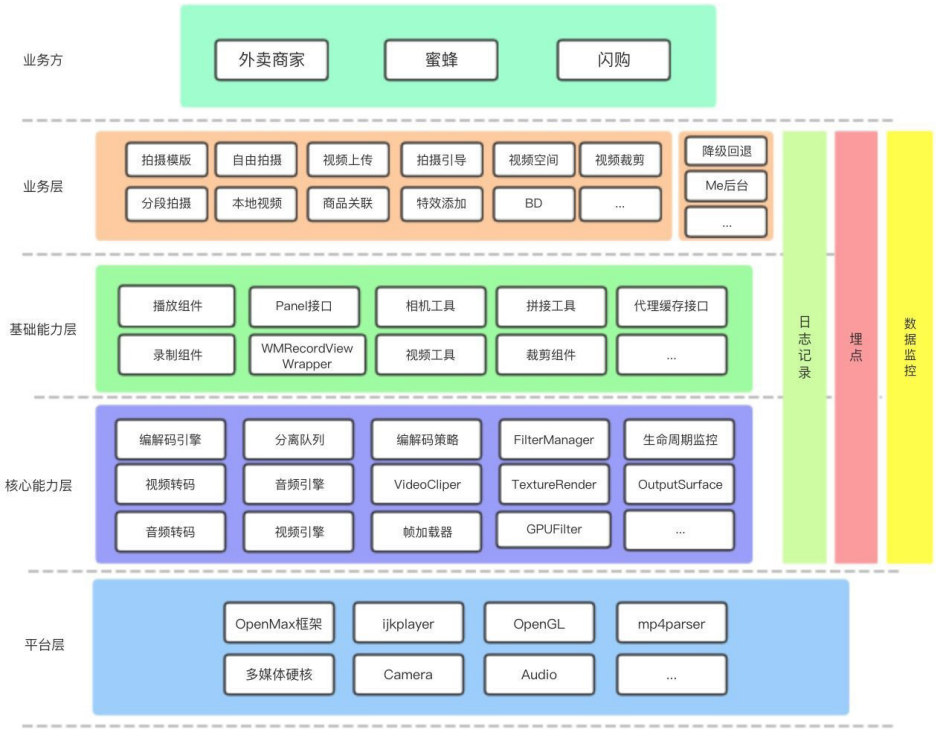
我们整体的架构设计，用以满足业务扩展和平台化需要，可复用、可扩展，且可快速接入。架构采用分层设计，基础能力和组件进行下沉，业务和视频能力做分离，最大化降低业务方的接入成本，三方业务只需要接入视频基础 SDK，直接使用相关能力组件或者工具即可。

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：

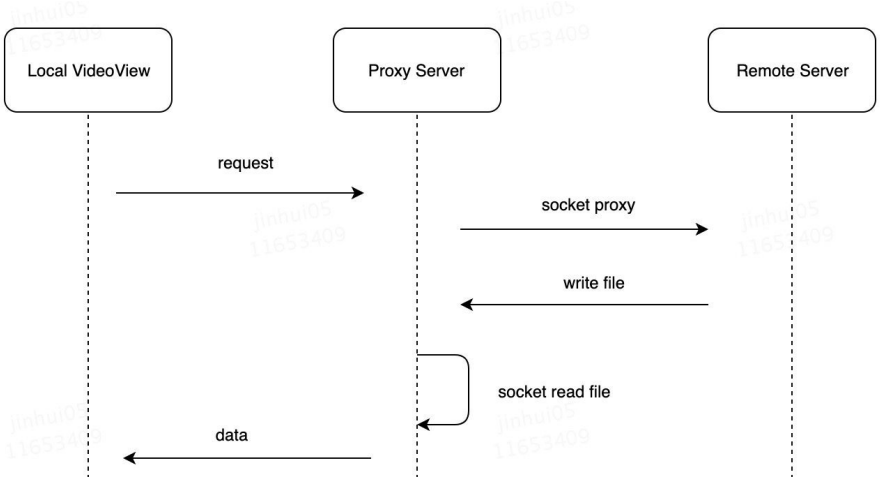
### 1. 兼容性问题

### 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);
        }
```

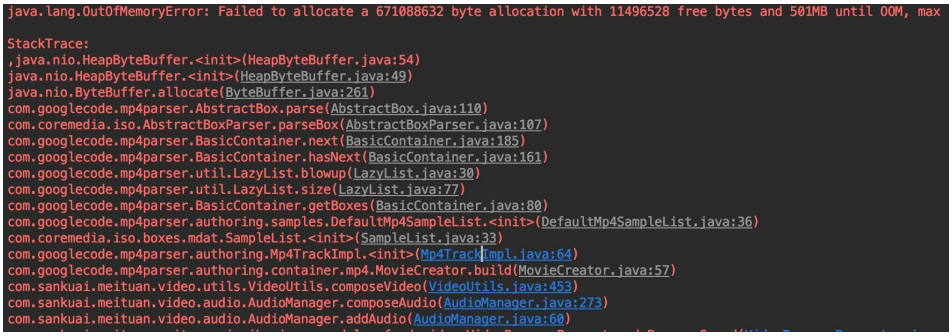
## 视频合成

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。




## 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。

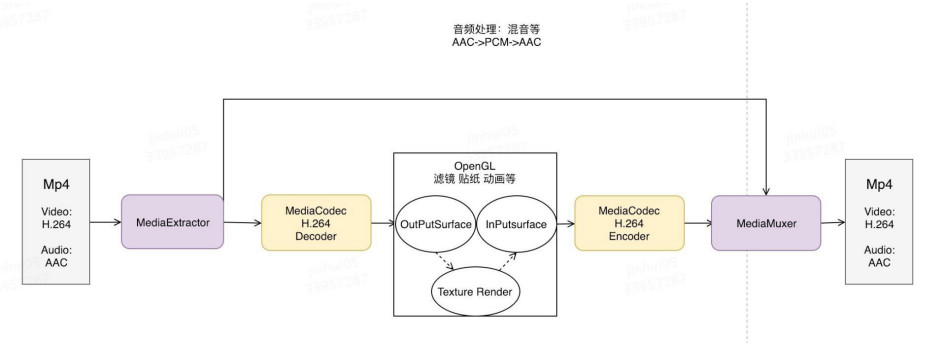
方案具体实施如下：先获得目标时间的上一帧信息，对视频解码，然后根据起始时间和截取时长进行切割，最后将裁剪后的音视频信息进行压缩编码，再封装进 mp4 容器中，这样我们的裁剪精度从秒级误差降低到微秒级误差，大大提高了容错率。

## 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 0xfffffffc3, actionCode 0, while in state 3
19624-19820 E/MediaCodec: configure failed with err 0xfffffffc3, resetting...
19624-19828 I/OMXClient: Treble IOmx obtained
19624-19820 W/System.err: android.media.MediaCodec$CodecException: Error 0xfffffffc3
        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

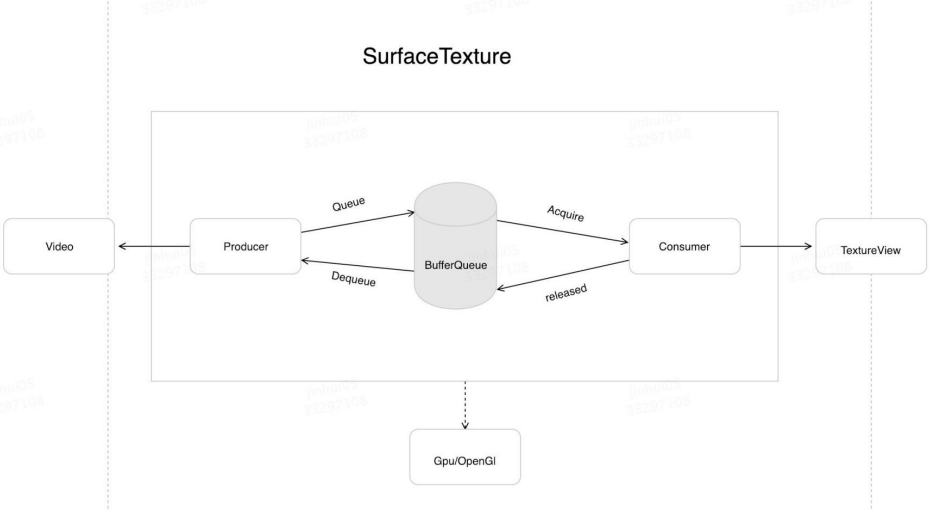
视频在二次渲染后会出现偶现的画面停滞现象，主要是 SurfaceTexture 的 OnFrameAvailableListener 不返回数据了。该问题的根本原因是 GPU 的渲染和视频帧的读取不同步，进而导致 SurfaceTexture 的底层核心 BufferQueue 读取 Buffer 出了问题。下面我们通过 BufferQueue 的机制和核心源码深入研究下：

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

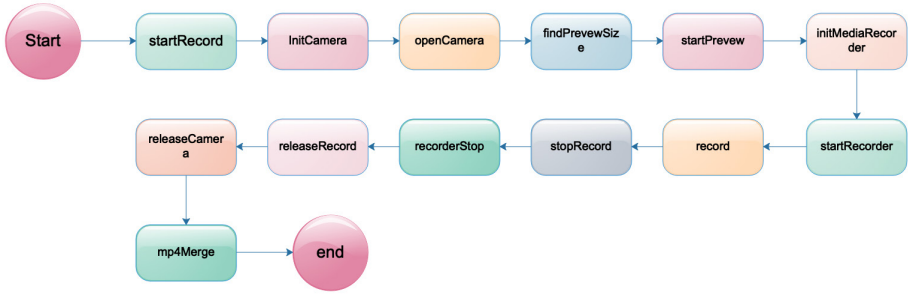
音频处理还括对音频的混音，消声等操作。在混音操作的时候，还要注意音频文件的单声道转换等问题。

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。


微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐