

大促背后的 前端核心业务实践

618技术亮点合集

618

每年的618是淘系前端的“年中大考”，通过618，来验证淘系前端的基础架构、研发体系、业务平台等发展是否满足预期；今年的618淘系前端在研发侧，通过imgcook和ATS实现了高质量的智能化研发，在终端侧通过PHA将Web和Native进一步融合，提供高性能和创新体验，在平台侧上通过搭投一体能力支持海量业务场景。这本电子书将详细分享这些体系的创新和实践。

——淘系技术部资深前端技术专家 姜凡（展炎）



钉钉扫码加入
「淘系互动交流群」
与万千技术爱好者共同进步



阿里云开发者“藏经阁”
海量免费电子书下载



微信扫码关注公众号
「淘系前端团队」
获取前端领域最新动态

目录

618 大促背后的淘系前端技术体系	6
生产力再提速，618 互动项目进化之路	14
淘宝大促页面性能监控和优化实践	24
「高稳定性」视频播放器养成计划	30
AST 代码扫描实战：如何保障代码质量	39
如何实现代码自动生成？	51
前端通用模块在手淘业务中的实践	67
2020 年，我们该如何学习 WEB 前端开发	73
附录	87

618 大促背后的淘系前端技术体系

2020 年 618 大促已经过去，作为淘系每年重要的大促活动，淘系前端在其中扮演着什么样的角色，如何保证大促的平稳进行？又在其中应用了哪些新技术？

本篇的作者是来自于营销活动团队的墨冥，为大家介绍 618 大促背后的淘系前端技术体系。

前言

持续近一个月的 618 电商大促终于落下帷幕。笔者有幸成为阿里这次 618 大促的前端负责人，借着这个机会跟大家分享一下支撑 618 大促背后的前端技术体系，作为阿里淘系 618 前端技术分享系列的开篇。

业务背景

2020 开年的前几个月，肆虐的新冠肺炎疫情把人们封印在家中，对中国的服务业、旅游业等线下经济产生了巨大的冲击，同时也大大促进了电商网购、线上办公、线上医疗等服务的发展。对电商平台、商家、消费者来说，今年的 618，成为了疫情之后国内最大的电商消费节点，对拉动内需，促进国内消费，带动就业起到非常关键的作用。

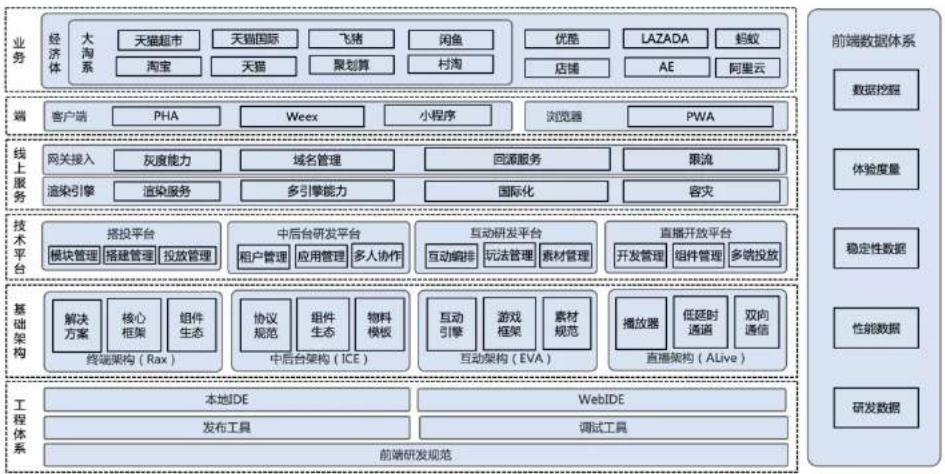
阿里作为电商领域的引领者，今年 618 大促一贯地保持了对友商的竞争优势，创造了新的数字消费记录，业务玩法和策略上也有了一些新的变化。例如：

- 超长的售卖周期：5.25 预售、5.29 开门红、6.4 多波段品类日、6.16 狂欢日。
- 消费券发放：平台、各地政府和商家一共发放了超 100 亿元的消费券和补贴。
- 直播带货：通过直播带货让用户更好地感知商品，提升流量变现效率，形成电商导购的新模式。

- 互动任务体系: 618 理想生活列车, 做任务, 赚喵币, 瓜分 10 亿平台和商家福利。
-

经过多年的沉淀和发展, 淘系前端已经构建出了一套较为完备的技术体系, 用以支撑阿里包含 618、双 11 在内的电商营销活动业务。接下来笔者将简单介绍淘系前端技术体系以及这个技术体系上基于 618 大促的场景诉求, 技术演进的创新点。

淘系前端技术体系



淘系前端技术体系大图

工程体系

- 前端研发规范: 统一的编码规范、组件规范、模块规范等, 确保跨业务, 跨团队之间的研发质量和协同效率。
- 发布工具: 云端构建的前端发布工具集。通过配套的规范、流程定义、权限管理以及数据日志提高前端开发效率, 保证团队开发过程的一致性和可复制性,

提升代码质量和安全。

- 调试工具：支持 source map、断点调试、本地代理、云真机等能力在内的调试工具集。
- 本地 IDE：集成发布工具、调试工具能力的本地开发环境。
- WebIDE：基于 Web，轻量化的集成开发环境，使用浏览器即可进行前端编码研发。

基础架构

- 终端架构：基于 **Rax**（已开源，超轻量，高性能，易上手的类 React 多端渲染引擎）的终端架构。Rax DSL 通过构建不同的产物可运行于 Web、Weex、小程序等容器，提供丰富的组件生态，做到一码多端，帮助前端高效研发无线页面。



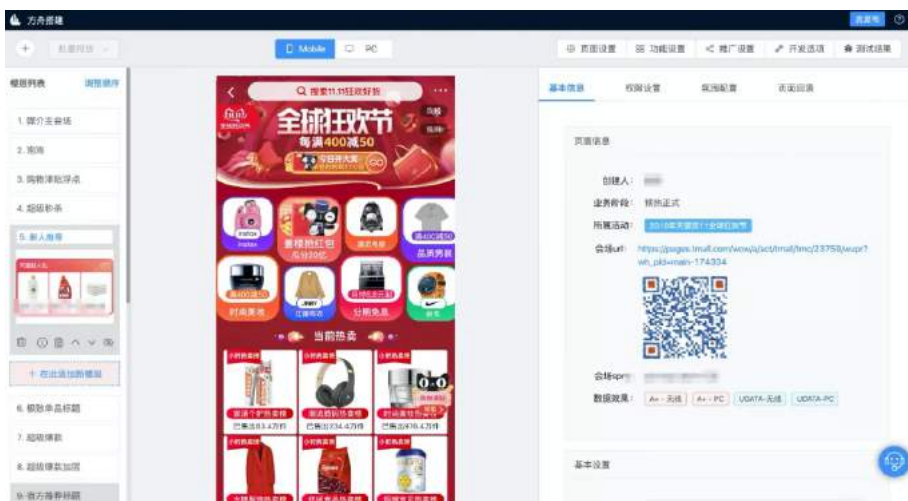
- 中后台架构：基于 **ICE**（飞冰，已开源）的中后台前端研发架构。支持微前端、丰富的中后台组件、领域模板，帮助前端快速构建中后台应用。



- 互动架构：基于阿里内部高效、高性能、可扩展的互动引擎 EVA（尚未开源）的互动架构。提供丰富的互动素材、组件生态，帮助前端高效研发互动玩法。
- 直播架构：基于阿里内部自研播放器、流媒体服务的直播架构。提供多端一致的播放器，低延时通道，可靠的双向通信机制，帮助前端高效研发丰富的短视频 / 直播玩法。

技术平台

- 搭投平台：基于丰富的模块体系和搭投服务，以 No Code 方式让业务能够搭建成千上万的页面，并提供可运营的数据投放管理能力。



- 中后台研发平台：基于 ICE 架构，提供 Pro Code、Low Code、No Code 三种方式让前端、开发高效研发中后台页面和应用。
- 互动研发平台：基于 EVA 互动架构，提供通过流程编排生成互动玩法的能力，并沉淀玩法库提供给业务直接使用。
- 直播开放平台：基于 ALive 直播架构，提供直播、互动、营销一体化解决方案，赋能二方、三方直播能力。

线上服务

- 网关接入：前端页面统一的网关接入层，提供域名管理、回源服务、限流、灰度等能力。
- 渲染引擎：基于 Node.js 编写的服务端渲染模版的容器，为阿里提供全平台的统一前端模版渲染引擎。

端

- 客户端：1. PHA: Progressive Hybrid App，渐进式混合应用，提供客户端内的辅助能力，提升 webview 渲染性能与体验。
- 2. **Weex**：一个可以使用现代化的 Web 技术开发动态高性能原生应用的框架。



- 浏览器：1. PWA: Progressive Web App，使用多种技术来增强 Web App 的能力，如通知推送，离线缓存等。

618 技术亮点

基于以上淘宝前端技术体系和这次 618 的业务诉求，淘系前端进一步做了许多技术和方案的亮点创新，例如 PHA、同层渲染、前端智能化探索、极致性能体验、营销互动提效、小程序、资损防控等，相关技术方案将以系列文章的方式做一些总结和沉淀，让我们先简单一睹为快。

互动生产力进化之路

今年 618 我们带来了名为“幸运列车”的互动游戏，携全国各地的特色农货和美食，让大家在这个夏天寻味中国。从 2019 年双十一的“盖楼”到今年 618 的“开列车”，在大促互动游戏背后，是业务多变性、产品稳定性和研发效率的多重博弈。这篇文章将介绍淘系互动前端团队如何应对研发效率 & 产品稳定性的挑战，内容涵盖“互动智能测试”&“弹窗规模化生产”这两个技术方案。

618 会场性能保障全揭秘

作为一名前端工程师，更高的性能、更流畅的体验是长久不变的追求目标。而作为大促锋线，会场页面的性能表现直接影响了亿万消费者的购买体验。那么在今天的 618，我们是如何让消费者们在上千张的会场页面里能够逛的知心、挑的称心、买的开心呢？这篇文章将简要介绍今年的 618，我们是如何通过预缓存、请求优化、监控测试等方案来保障会场页面体验的。

亿级用户高稳定性视频播放器养成计划

2020 直播带货是电商导购的新模式之一。PHA 框架的优秀性能，使大量业务回归跨平台和开放的 Web 体系，但原生系统的播放器对于直播 / 短视频来说，稳定性、

性能、播放能力支持均难以达到使用标准。这篇文章将介绍此背景下，我们如何通过同层渲染技术实现在 Web 中使用阿里淘系自研的 Native 播放器，做到期间 0 故障，整体无降级，端 crash 率稳定。

资损防控 – 代码扫描技术揭秘

现如今，日常业务的资损防控工作在生产环节中已经变得越来越重要。尤其是每逢大促活动（譬如本次 618 大促），一旦出现资损故障更容易引发重大损失。如果只是通过 code review 之类的方式，效率低且其质量参差不齐，无法得到保障。这篇文章将介绍我们如何通过引入代码扫描，在每次代码提交时都能自动检测出代码中的资损风险并给出告警，从而在研发阶段就能提前发现问题并及时修复。

P2C – 需求智能出码的思考

AI 技术的飞速发展，使得机器代替人编码产生了可能性。P2C，即通过智能算法由结构化产品需求文档直接生成可用前端代码的技术方案。实现 P2C，将给代码研发带来巨大的效率突破。这篇文章将主要围绕自动化生成代码的目标，分享我们在这一过程中的所思所想，以及我们在 618 会场中的实践。

旗舰店小程序升级，承上启下的一步

为了建设更加开放的生态能力、更加丰富的商家运营能力，我们在过去一段时间操刀了旗舰店小程序升级。旗舰店作为流量大、架构复杂、稳定性要求高的典型场景，技术挑战极大，整个过程遇到的问题非常多。在本次 618 大促，店铺是如何落地小程序技术方案？又是如何建设小程序的性能体验？最后又是如何保障店铺的大促稳定性？这篇文章会重点介绍店铺的小程序架构、性能优化方案、稳定保障措施等，围绕店铺在小程序上的实践，分享在整个过程中遇到的问题和经验。

频道业务黑科技 – 行业魔方

过去的一年，天猫行业的业务发展促使快速建场、高效用场的需求愈发强烈，而

行业前端的开发方式仍是劳动密集型，对行业频道这类长尾业务弊大于利。得益于淘系前端的积累，去年底开始，天猫行业与 UED、产品团队合作完成了 TaoUI 组件规范，并建设了织网组件中心来支撑行业沉淀下来的物料，那么，如果按照一定的规范，使用直接的数据模型直接驱动组件，是不是大部分普适的模块就不需要开发了？于是，行业魔方应运而生。在这次天猫服饰行业 618 会场的商品内容中支持了混排 Feeds 流，这篇文章将分享通过更通用、轻量的方案为营销会场带来了更丰富的体验，让用户不仅买得爽，还能看得爽。

结语

作为阿里淘系 618 前端技术分享系列的开篇，本文主要是抛砖引玉，从整体上介绍淘系前端技术体系以及在 618 大促中的技术亮点，请期待后续详细的各项专题详细分享文章。

生产力再提速，618 互动项目进化之路

前言

本篇来自于淘系技术部互动前端团队，今年我们带来了名为“幸运列车”的互动游戏，携全国各地的特色农货和美食，让大家在这个夏天寻味中国。

从 2019 年双十一的“盖楼”到今年 618 的“开列车”，在大促互动游戏背后，是业务多变性、产品稳定性和研发效率的多重博弈。本文介绍了淘系互动前端团队如何应对研发效率 & 产品稳定性的挑战，内容涵盖“互动智能测试”&“弹窗规模化生产”这两个技术方案。

互动智能测试

当前互动玩法愈发新颖多样，这给业务开发的效率带来很大的挑战，我们需要在视图模型中维护大量的状态。

以列车合成区域（下图红框）的状态为例，共有 10 个合成位可以放置车厢，每个车厢有 58 个等级，开发的时候需要模拟大量的数据。另外在互动过程中还有抽中红包等概率事件，异常状态情况的验证也有很高的成本。



618 互动游戏的玩法可以简单归纳为：用户通过各种行为获取喵币，消耗喵币升级列车换取惊喜红包，最后兑换现金红包的过程。

我们通过机器学习的手段，帮助我们模拟用户的行为，获得真实交互环境中产生的数据，而不是手动枚举方式造测试数据。同时还结合 Puppeteer 模拟真实客户端环境，在线上需要变更时，快速的进行前端功能回归验证，减少研发成本。

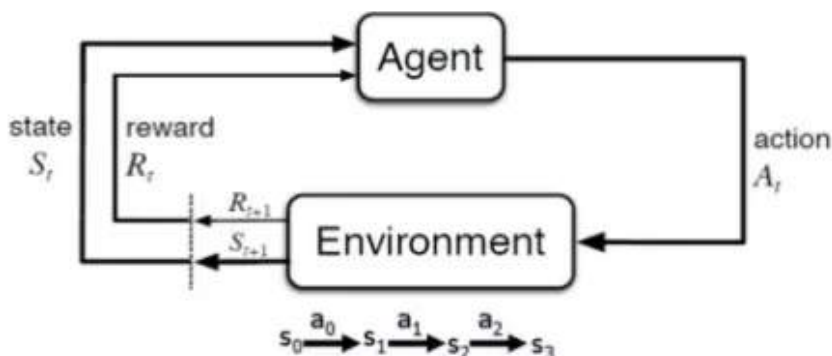
强化学习

强化学习是机器学习中的一个领域，强调如何基于环境行动，以取得最大化的预期利益，也意味着能够更快速的到达互动玩法的最终状态。

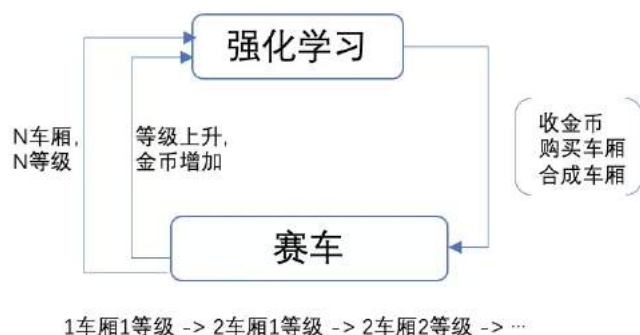
其中环境通常被规范视为马尔可夫决策过程 (MDPs)。首先介绍几个概念：

- Agent: 学习和做决策的主体
- Environment: Agent 交互的对象
- State: Agent 的状态
- Action: 行动, Agent 会根据当前的状态选择 Action
- Reward: 奖励

MDPs 简单说就是一个智能体 (Agent) 采取行动 (Action) 从而改变自己的状态 (State) 获得奖励 (Reward) 与环境 (Environment) 发生交互的循环过程。Agent 会持续的和环境交互，根据当前的状态选择 Action，而环境会给 Agent 新的状态和 Reward。



为了在项目中生成数据，我们定义了如下 Action：收金币、购买车厢、合成车厢等。Agent 不断与赛车玩法交互，Agent 从初始化状态开始，进行“发送 Action -> 更新状态”的循环，直到最终达到目标“抽大奖”，学习过程到此结束。



学习流程图

我们将学习过程中的状态快照记录了下来，作为服务接口的测试数据，帮助前端侧开发和调试。

ID	名称	车厢信息	剩余金币
State_1	初始化数据	2 x 车厢等级1	20000
State_2	购买车厢数据	3 x 车厢等级1	15000
....
State_N	抽奖数据		

自动回归测试

互动场景下的前端交互非常复杂，然而前端功能回归一直以人工方式为主。我们在项目中尝试自动生成测试用例，用于回归测试。

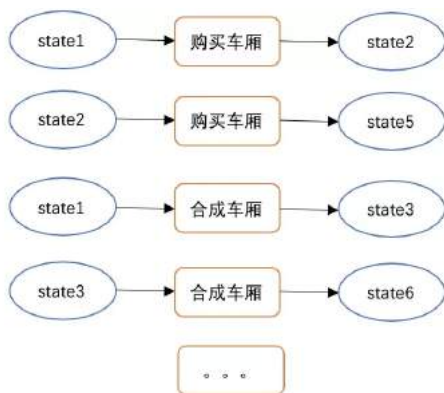
从用户交互的视角出发，收金币、购买车厢、合成车厢都对应用户的一次交互行为。我们在 Puppeteer 环境中运行页面，根据沉淀下来的数据，回溯到特定的状态。

并结合强化学习，择优触发前端 Action 事件，模拟真实的用户操作，最终形成用户的前端交互行为树。



用户交互行为树

得到用户交互行为树之后，我们会对行为路径再进行优化，排除无价值的链路，合并重复链路，并最终拆分成简短的片段便于测试。如合成车厢 N 次，会处理成为 N 个测试用例，尽量以同种状态下的最短路径作为最终的测试用例。



前端测试用例图

在需要回归测试时，我们可以在 Puppeteer 环境中回放测试用例，做到了前端功能的自动回归。这个过程中，我们把各个测试用例的 UI 快照保存了下来，利用图像识别技术进行最后的校验。

以倒计时浏览任务为例，我们需要验证在跳转后的页面上，是否正确的展示了某个组件，通过图像元素的对比，可以判断该功能点是否正常。



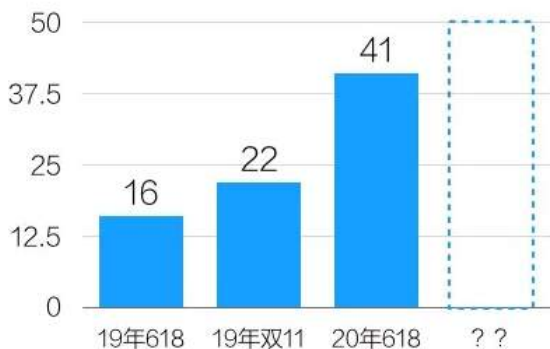
检测到组件，测试用例通过

互动项目的业务逻辑，是一系列用户行为带来的反馈的组合物，智能测试方案在本次 618 互动项目中，成为了前端开发测试阶段的提效利器。在线上阶段发生变更时，可以快速完成线上功能的全量回归和新功能的验证，保障线上业务的稳定。

弹窗规模化生产

今年的 618 的弹窗场景数量是去年 618 的 2.5 倍，弹窗由于可以避免触及到游戏区域的复杂变动，常常被用来满足各类支线乃至主线需求，帮业务完成各个细分领域的玩法覆盖，在无线营销互动领域中，弹窗需求一直处于持续增量的状态。

近几年大促养车玩法弹窗需求数



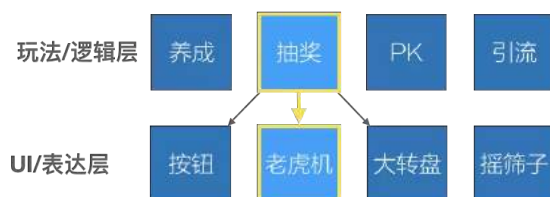
弹窗看似简单，但每个弹窗都有自己的意义以及属性，可复用范围非常受限，传统方式研发弹窗的成本较高，在业务快速发展的背景下，我们需要有更快更好的研发交付能力。

表达层与逻辑层的解耦

一般情况我们可能会将弹窗沉淀成包含 UI 的弹窗组件库，也会进一步会将弹窗细节抽象出 header、body、button、footer 等配置项。但这样会有一些问题，在互动领域下的一个按钮布局、一个图标形式都会让这个“组件”越来越臃肿，所以不要天真的试图用前端的设计思路，去预判设计师天马行空的设计理念。毕竟不同的玩法和品牌形象下，对 UI 的定制往往有较强的诉求，因此在营销互动中很难达到真正的 UI 可复用，因此我们要将表达层完全抽离出来，弹窗方案的逻辑层只负责模型的处理，表达层通过接受数据变化带来的“表达”变化。

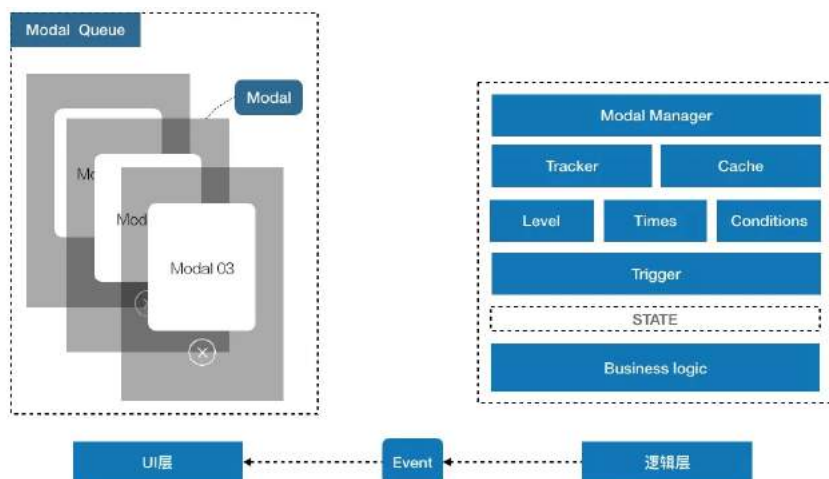


例如实现了一个抽奖玩法，逻辑层包含了数据模型、登录初始化请求数据以及抽奖事件的后续逻辑行为，那么该数据模型下最终表达层选用的是老虎机、大转盘还是直接点击抽奖按钮其实都是兼容的。



解耦下的弹窗逻辑层

参照上述的解耦方法，我们将弹窗的能力分为 UI 层跟逻辑层，大致结构是逻辑通过事件唤起弹窗，先抛开 UI 层那么先对逻辑进一步结构化，最终逻辑层的结构以及逻辑层跟 UI 层的关系如下图所示。



逻辑层通过监听业务数据层变换，初始化后 Trigger 管理器负责从配置队列中检索到匹配条件的行为，开发者几乎可将所有诉求类的弹窗根据 Conditions（触发条件）、Times（展示次数）、Level（层级面）等能力描述出来，并通过配套的 runtime 快速生成业务所需的逻辑，例如一个初始化进来后的弹窗只需要描述这样一个 DSL。

```
1  {
2    created: [{
3      id: 'eveningParty',
4      triggerEvent: 'init',
5      condition: [
6        {
7          key: 'state.key',
8          operation: '===',
9          value: true
10       }
11     ],
12     times: {
13       interval: 'day'
14     },
15     modalView: {
16       id: '2020618Pops',
17       sceneId: 'Other.PartyLottery'
18     },
19     tracker: {
20       key: '/2020618.rewards.night_rewards',
21     },
22     param: {
23       msg: 'state.msg',
24       text: 'state.count + 1'
25     },
26     level: 1
27   }]
28 }
```

解耦下的弹窗视图层

为了给予设计师更多的发挥空间，我们对 UI 进一步结构化拆解，直到要达到可以快速编排出 UI，以及支持动态下发。

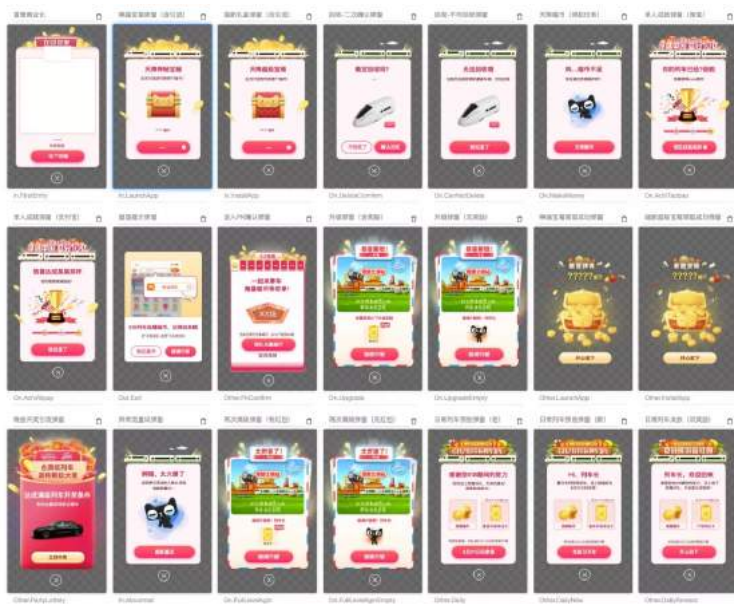
一个项目中往往会有多个弹窗，每个弹窗有许多图层组合，假设类型比较简单只会有组、图片、文案等类型的图层，图层上会有静态 & 动态的属性描述。

我们可以通过经验所得来的项目 > 场景 > 图层各个维度的拆解，把静态配置 + 动态绑定等能力对弹窗的 UI 进行结构化描述，如下图。



首先 UI 部分是纯函数式的，所以只需要支持 UI 描述以及动态参数，就可以展示实际的弹窗 UI，让业务开发专注在弹窗的逻辑描述上。

配合提供相应的 UI 设计器，开发者就可以根据需求绘制出所有弹窗，把弹窗的 UI 开发成本降到最低。



618 弹窗部分截图



UI 设计器

通过结构化 UI 层我们就很方便的做很多事情，首先弹窗 UI 跟逻辑方案都是 DSL+Runtime 的相同机制，所以只需要配合搭建平台或者异步接口，就能快速支持动态下发的能力。

以及面向开发者协同以及视觉还原的真实预览能力，在发布时，平台上的弹窗预览功能将原来可能需要近半小时的弹窗配置 review 环节，缩短到几分钟，且准确度更高。

淘宝大促页面性能监控和优化实践

本篇的作者是来自于淘系技术部 前端营销活动团队的森郎，为大家介绍今年的618会场是如何做到顺滑体验。

前言

作为一名前端工程师，更高的性能、更流畅的体验是长久不变的追求目标。而作为大促锋线，会场页面的性能表现直接影响了亿万消费者的购买体验。那么在今年的天猫618，我们是如何让消费者们在各个会场中能够逛的知心、挑的称心、买的开心呢？

本文将简要介绍今年的618，我们是如何通过缓存优化、请求优化等方案来应对性能挑战，并如何通过监控测试等手段来保障大规模的会场页面的性能。

变化与挑战

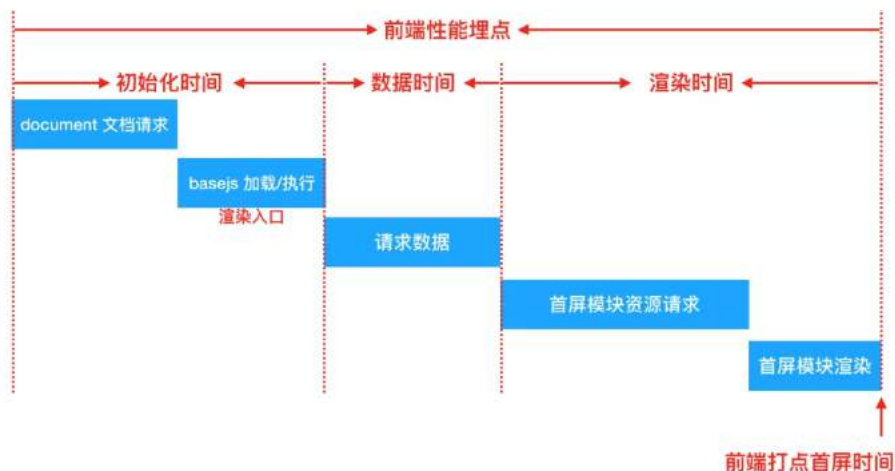
今年的618大促，在终端渲染方案上，淘系营销活动团队与终端架构团队深入合作，第一次大规模在会场域落地了PHA (Progressive Hybrid App) 方案，并在主会场使用了web方案。这一变化对页面的资源加载、执行耗时都带来了挑战。而在业务玩法上，今年的会场在内容化、本地化等方向发力，短视频、红人直播等在H5页面上同层播放，这对页面的内存占用等提出了一定的要求。



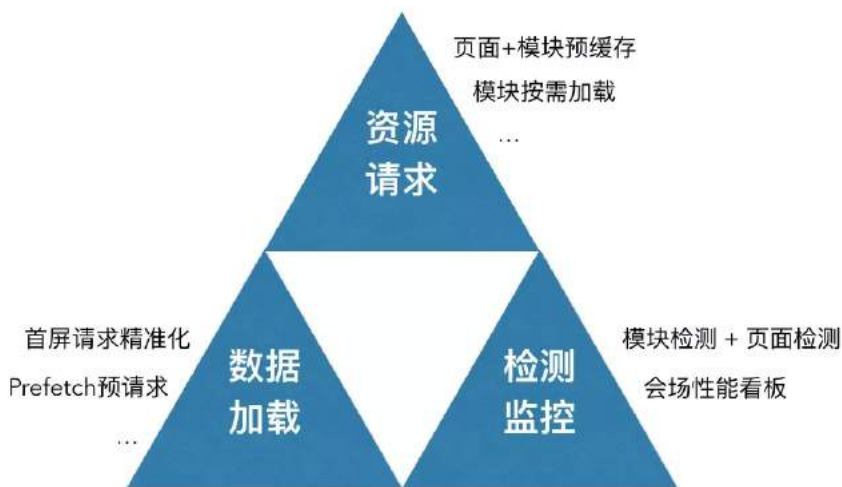
说到挑战,首先,新用户、低端机、弱网络等在大促期间的占比明显增加,带来了较大的劣化长尾影响。其次,天猫、淘宝的小二们在 618 期间创建了大量的会场页面,如何在大规模的体量下仍然整体保持高性能水位,这对业务配套的检测监控方案也提出了一定的要求。

方案详解

在性能优化的落地过程中,我们根据会场页面的前端时序(如下图所示),将会场的性能优化拆分为了资源加载、数据请求、模块渲染等环节。



当前的会场所使用的渲染方案是一套页面、模块、数据相分离的异步式方案。在资源请求上,可分为三个部分:一是 HTML 文档(不同页面之间相同),二是包含了 `rax`、`loader`、`render` 等在内的基础 JS(不同页面之间相同),三是页面所使用的各个 UI 模块的 JS+CSS 资源(不同页面之间不同)。数据请求由页面的 `render` 统一发起,并在数据网关层将当前页面的不同模块的商品列表、店铺信息等数据在组装后合并返回。



结合上述的变化挑战与实际线上的性能情况，今年 618 会场的性能保障方案可主要概括为上图的三个环节。下面将简要介绍各个环节的策略与实现。

资源请求优化

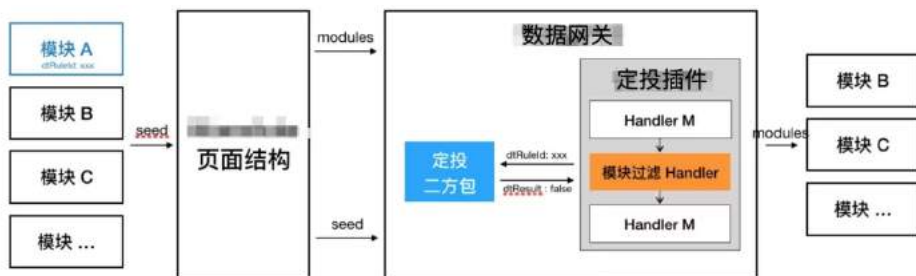
页面预缓存

在手机淘宝、天猫等 app 内，我们利用了端侧提供的静态资源缓存方案，将 HTML 和基础 JS 等资源，推送下发至客户端。在客户端的 webview 请求资源时，端侧可根据域名来匹配已下发的缓存包，在匹配成功后直接从本地缓存中读取对应的 HTML 和 JS 资源，而无需每次都请求网络、大大缩短了页面的初始化时间。例如，主会场的 html 加载耗时在推送缓存后可平均减少 70+%、命中率可达到 97%。

模块资源缓存

模块的 JS+CSS 资源，因为不同页面所使用的模块不同（例如男装会场和家装会场），并且总计有上百个模块，无法做到全量的提前缓存。这里我们通过捞取 top 流量页面的方式，仅将首屏相关的核心模块做了模块缓存下发，较好的缓解了模块请求的耗时。

模块按需加载



除了刚提到的模块缓存下发，今年的 618 会场还通过“模块按需加载”的优化方式，最小化的控制了当前页面的模块数量，这对首屏的 JS 资源请求、数据请求都有一定的缩减。在实现方案上，通过在数据网关层先读取服务端所缓存的定向投放条件，判断当前访问的 URL 参数、客户端信息等是否满足模块的展示条件（例如，搜索框模块仅在手淘内才展示）。不满足条件时，则直接从页面中移除这一模块。例如，在外部浏览器里打开 618 超酷数码会场时，页面所加载的 JS 资源大小可因此减少 40+%。

数据请求优化

为了给消费者们推荐最贴心的好物，现在的大促会场使用的更多是千人千面、个性化的算法推荐数据。但推荐所带来的算法耗时增加，使得数据请求成为了当前会场性能的耗时最长板。如果能够降低当前会场的数据请求耗时，那么首屏时间也会随之明显减少。

所以在今年的 618 会场中，我们落地了一套基于线上实时埋点的请求优化方案，通过更精简的首屏模块数量来实现了对首屏接口耗时的直接优化。

在方案中，通过采样上报的方式获取到了模块高度，并结合设备信息、页面模块排序、个性化数据等条件，个性化的控制了每次访问的首屏模块加载数量。例如，小明在访问数码会场时只需要加载 5 个模块，而小强只需要加载 4 个模块。以线上 RT 情况为例，针对于单个会场，方案可使得首屏耗时平均减少 100+ms。

监控检测

这次的 618 使用了两套检测机制来保障会场性能。第一套是模块级检测，通过与 H5 自动化测试平台等打通，批量化的对所有大促模块进行了性能检测，从中发现了例如部分模块的引用图片未压缩、资源域名未收敛等问题。

页面检测详情

Android性能评测	Chrome资源检测	适配前图
检测结果: 不通过 得分: 86		
UITest地址: UITest地址	测试环境: docker	更新时间: 2020-05-28 23:55:35
加载性能		
> 首屏外图片未懒加载数检测 检测标准: 0, 结论: 通过		
> 首屏图片display:none检测 检测标准: 0, 结论: 通过		
> 请求资源大小检测 检测标准: 50k, 扣分: 14		
> css中小于10k的png背景图检测 检测标准: 0, 结论: 通过		
> assets的未域名收敛数量检测 检测标准: 0, 结论: 通过		

第二套则是页面级检测，在小二们完成了会场页面的配置填写后，批量化的对所有的 618 会场进行了自动检测，使得一些页面加载、数据请求、模块渲染等相关性能问题，可在上线前得以被发现和修复。

而性能监控上，与双 11 整体保持一致，今年 618 使用了一套接近于 FMP 的性能指标，以自定义采集上报的方式、联动前端监控平台，实现了分钟级的性能看板，帮助我们更快的发现线上性能问题、并作出针对性的调整。

降级策略

同层播放组件的引入对会场页面的内存占用提出了较高的要求，尤其是低端机、老系统等。针对这个问题，今年的 618 会场域统一接入了客户端所提供的降级平台，可直接根据机型、系统版本、客户端版本等条件动态的执行降级策略。例如，可设置在手淘版本 $\leq 9.5.0$ + iOS 版本 ≤ 10 时不播放商品短视频等等。

今年的 618 会场，线上整体稳定性良好，未出现因会场内存占用过高而导致的客户端 crash 飙升情况。

整体回顾

性能优化，功在平时，成在大促。在今年的 618 大促中，通过上述的优化方案和保障手段，最终达成了预期的体验目标。



但在过程中，也发现了一些可继续深挖的性能优化点。例如，可以结合端智能等客户端的能力帮助大促主会场提前完成页面数据、依赖资源的预加载，甚至是页面的预渲染。我们会在后面的一场场大促中持续迭代优化，为消费者们提供没有最好、只有更好的购物体验。

总结

618，这一场持续了 20 多天的年中促销活动，对于亿万消费者们来说是一场狂欢盛宴，而对于在背后默默支撑这一场大促的所有工程师们，则更像是一场极客马拉松，痛并快乐着。

「高稳定性」视频播放器养成计划

本篇的作者是淘系技术部高级前端工程师 叶序，为大家介绍亿级用户高稳定性视频播放器养成计划。

背景

PHA 框架的优秀性能让大量业务、会场开始逐步转用 H5，但同时带来了一些挑战。以多媒体日常短视频 / 直播业务为例，H5 原生的播放器的稳定性、性能、播放能力支持均难以达到使用标准，在 H5 环境下没有一个业务可用的 H5 播放器。这时候就需要一个 H5 上能够流畅播放的播放器。

先提前体验一下 618 期间，H5 下播放器能力。

618 期间战果：618 主会场，猜你喜欢，直播会场，聚划算百亿补贴、行业会场。

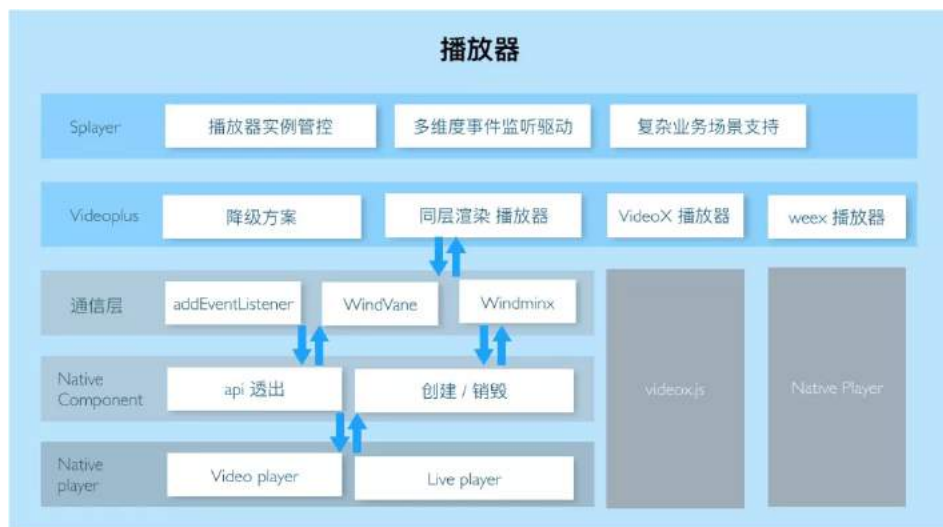


同层渲染播放器方案设计

也许你之前不了解什么是同层渲染？

同层渲染，是允许将 Native 组件和 WebView DOM 元素混合在一起进行渲染的技术，能够保证 Native 组件和 DOM 元素体感一致，渲染层级、滚动感受、触摸事件等方面几乎没有区别。

综合了性能、稳定性、易用性、灵活性上考虑，我们选择了同层渲染这种技术方案，以此渲染手淘 H5 下播放器。



底层 Native 播放器 拥有一套健全的播放能力，在早期 weex 及 Native 下，均有不俗表现。经过多年沉淀，api 健全、性能稳定性良好，Native 播放器能力已达到业务可用的标准。

在此基础上，需要针对 Native 播放器 做一层同层渲染组件的封装处理以适用于 webview (iOS wkwebview, android UC 内核) 上播放器的同层渲染。客户端封装组件 层需要监听同层渲染事件通知，在接收到创建 / 销毁消息时，实例化 / 销毁播放器。













监听的依赖于中间的通信层，淘系基础架构团队针对同层渲染的组件做了监听：在同层渲染组件创建节点时，通知到客户端播放器实例化渲染业务层播放器封装，使用了 `@ali/rax-composite-view-factory` 同层组件渲染工厂方法，该库将组件渲染成节点，当节点渲染出来时，前端会通知到监听层需要渲染何种组件，此时监听层收到后通知到客户端组件，客户端完成渲染，并持续接收来自前端的事件监听。（iOS 通过 `element.addEventListener` 通信，Android 通过 `WindVane` 通信）。

Videoplus 从业务角度来说，视频播放主要分为两种：直播和点播。

其中，直播中区分：直播流、回放、看点，点播则是视频播放形式。

从图中可以看到，对于如此复杂度的播放要求，在此之前，仅在 weex 和 web 端下针对复杂业务的播放器支持。手淘内在 PHA（即手淘 H5）场景下，没有一个适用的播放器。

618 前视频组件支持情况：

		WEEX	PHA	WEB
直播	直播流			
	回放			
	看点			
点播	视频			

Videoplus 是端内针对多场景播放器的一个组件，该组件在对不同场景下的播放器进行了抹平处理，对用户来说，无需关注当前场景，只关注播放器本身即可。

播放场景抹平：Videoplus 针对环境做了一层抹平处理，不同场景使用不同的播放器，本次 618 新增同层渲染播放器，能力上更加完善。

- weex：使用 weex 底层封装的播放器。
- 手淘 H5：使用同层渲染播放器播放 3、外部 web：使用多媒体前端团队 videox.js。

事件 / 属性 / 监听：Videoplus 针对播放器的时间、属性、监听进行一层抹平，符合 w3c 标准播放器的 api 标准，用户使用时，如果上手过 video 开发，接入成本大大降低。



Splayer

Splayer 严格意义上来说不是一个真正的播放器，是一个业务基于在 Videoplus 上的封装层组件。Splayer 不针对播放器本身功能做特殊处理，把重点放在了业务日常开发播放器中更关注的一些实用功能上。

在 Splayer 管控之前，业务背景十分复杂：

- 列表形式单列流双列流
- 轮播形式
- Tab 形式
- 版头模式

支持如此复杂且多样的交互上场景，Splayer 必须兼具各项能力，对于事件监听及分发处理逻辑十分复杂。

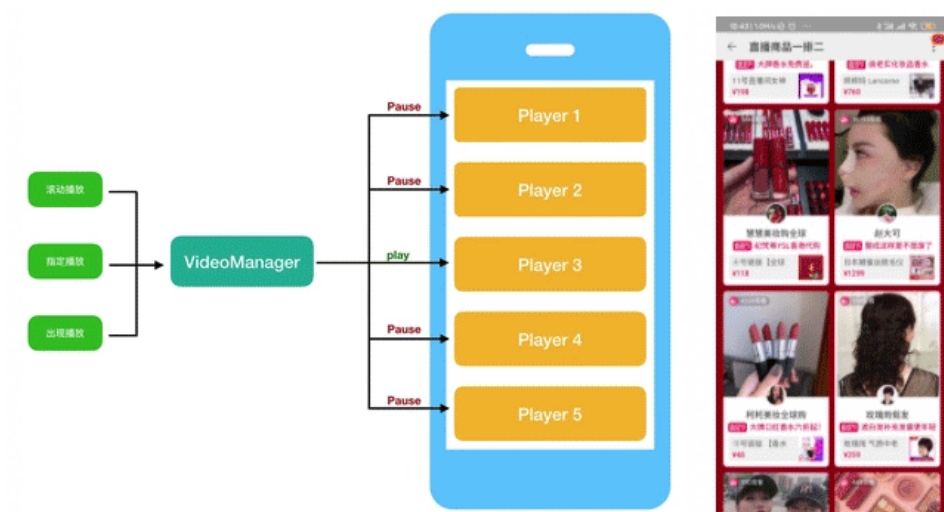
同时，需要整体考虑客户端性能稳定性，保持播放器单例管控。于是：

实例管控：Splayer 总能保证当前播放器只有一个播放器实例，避免因为实例无限增长导致 OOM。

对于类似会场模块组成形式的页面，Splayer 要求各个模块引用的版本一致，实例化出来的 SplayerEmitter 保证唯一性并能监听到其他模块发送的指令。

事件监听：Splayer 提供了一系列事件监听方案控制视频播放器，可以通过 Splayer 提供出来的 SplayerEmitter 完成播放器的销毁或者创建工作，并且无需担心其余播放器的处理（在每次操作播放器时，Splayer 会同时管理其他播放器实例）。

- 滚动监听：当滚动停止时，Splayer 会寻找播放器最佳位置播放。
- 指定 id 播放监听：存在播放器列表场景下，指定单个播放器监听播放，此时 VideoManager 会销毁其余播放器，并将指定 id 的播放器实例化。
- Appear/DisAppear 监听：依赖于 rax 的 appear/disappear 方法，在每个 Splayer Appear 时，VideoManager 会缓存当前的播放器，并从当前缓存的 Appear 播放器队列中选择最佳位置（居中）的开始播放。



稳定性保障方案

Videoplus 提供了多维度的降级能力，通过手淘 mt 配置降级方案。在大促期间，在预案平台提前报备。

Videoplus 的播放能力整体分为以下几个维度：

- 全量降级终极兜底方案，开启后，所有 Videoplus 播放器强制降级成 poster。
- 播放器场景降级方案 (H5/weex/PHA) 支持 H5/weex/PHA 场景，对应场景（直播或者点播）播放器出现意想不到的情况时应急采用。H5/weex/PHA 场景分别带有：1、禁用直播 2、禁用视频。
- 指定 bizFrom 降级方案在某一业务使用播放器时发生问题时，兜底使用。当开发业务接入播放器时，需要传入唯一业务指定 bizFrom，具体需要到播放器管理者处申请，如果未传入值，则播放器不可使用。可直接指定某项业务 (bizFrom) 关闭播放功能，一旦关闭，直播和视频在该业务场景下不可使用。
- 指定客户端 / 客户端版本号使用指定客户端版本及版本号使用。在手淘及天猫客户端版本中，如果播放器被发现某些 bug，可通过此开关降级。粒度控制到三位版本号，例如手淘 10.0.0 版本。

- 指定操作系统版本号此场景由于某些操作系统及版本下播放器出现问题，粒度控制到三位版本号，例如 iOS 10.0.0 版本。

大促技术和业务表现

淘系技术部 Native 播放器内核的多年沉淀，Native 底层播放器处于稳定状态。相较 H5 播放器来说，Native 播放器支持格式更广、解码能力更优、性能体验极佳，但灵活能力不足。基于 Native 播放器的同层渲染播放器，在体验上与 Native 播放器达到持平状态的同时，灵活能力兼具。

H5 的播放器 vs 同层渲染播放器

	H5 原生播放器	同层渲染播放器
自动播放	iOS 不支持 Android 支持	✓
H.265	✗	✓
HLS	iOS 支持 Android 不支持	✓
FLV	✗	✓

618 战果

业务数据：6 个以上 618 模块（行业、直播 1x2、猜你喜欢、头图等）接入，服务超过 200 个以上会场。

技术表现：sls 大盘监控底层播放器成功率 99% 以上；客户端监控稳定无 crash 新增，无故障，无回滚降级，整体灰度顺利。

同层渲染播放器遇到的问题

问题：前端移除之后，PHA Tab 切换挂起 webview，导致异步操作失效未将播放器实例销毁，主会场头图 / 猜你喜欢模块、行业商品模块、直播会场 1x2 模块。

解法：播放器监听 PHA 页面 disappear，强制停止播放，并 forceUpdate 同步销毁播放器解决此问题。

问题：Splayer 的复杂使用场景，在 1x2 模块播放器视频切换过快情况下，消失动画延时，封面图移除时机出错导致出现白窗。

解法：调整底层播放器封面图移除时机，在播放前移除。

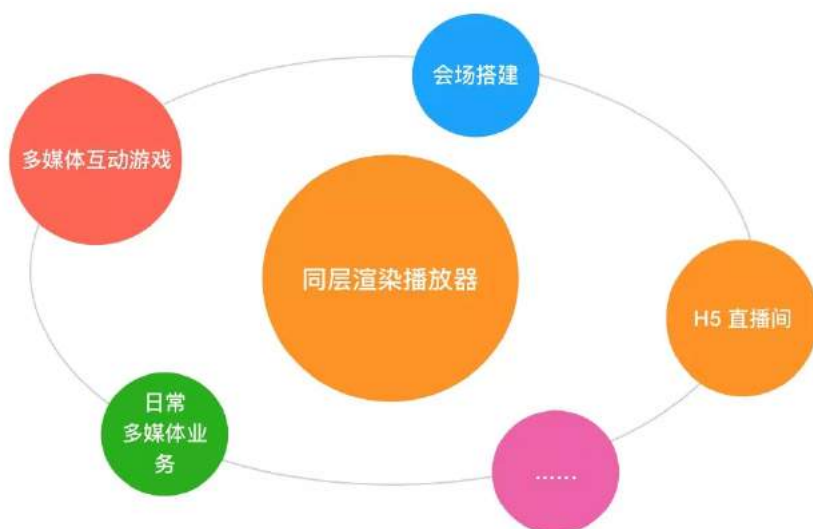
未来规划

对于本次 618 的实验与应用，同层渲染播放器仅仅是小试牛刀。本次大促同层渲染播放器的能力让业务尝到甜头，结合今年直播带货的火爆，后续手淘大量业务将会更积极拥抱 PHA 使用同层渲染播放器能力，对于我们而言，如何把握住风口将播放器大力发展走在业务前头，驱动业务发展是主要目标。

同层渲染播放器后续将会持续发展，提升播放体验、直播互动能力：

1. 降低接入标准，覆盖全量环境，业务 0 成本接入。
2. 打造端内体验对标客户端直播间的 H5 直播间，应对业务的快速迭代。
3. 提升会场播放器播放体验，带来更多 GMV 转换。
4. 未来直播互动游戏打造。

以同层渲染播放器为钩子，技术能力完善驱动业务能力升级，全面发展多媒体业务多点开花。



AST 代码扫描实战：如何保障代码质量

本篇来自于频道与 D2C 智能团队的慕竹，为大家介绍本次 618 大促中是如何用代码扫描做资损防控的。

前言

现如今，日常业务的资损防控工作在生产环节中已经变得越来越重要。尤其是每逢大促活动（譬如本次 618 大促），一旦出现资损故障更容易引发重大损失。就目前来说，有效的防控手段一般有：

- 项目上线前 code review，通过预演提前发现问题。
- 线上实时监控对账，出现问题时执行预案，及时止血。

由上可以看出，及时止血只能减小资损规模，要想避免资损还得靠人工 code review 在项目上线之前发现问题。

然而，一方面 code review 需要额外的人工介入，且其质量参差不齐，无法得到保障；另一方面，高质量的 code review 也会花费较多时间，成本较高。

那么有没有一种两全其美的方法：以一种低成本的方式，自动发现代码中存在的资损风险，从而保障代码质量？答案是：代码扫描！

我们希望每次代码提交时都能自动检测出代码中的资损风险并给出告警，从而在研发阶段就能提前发现问题并及时修复。接下来，本文就将介绍本次 618 资损防控中我们是如何用 AST 来做静态代码扫描的。

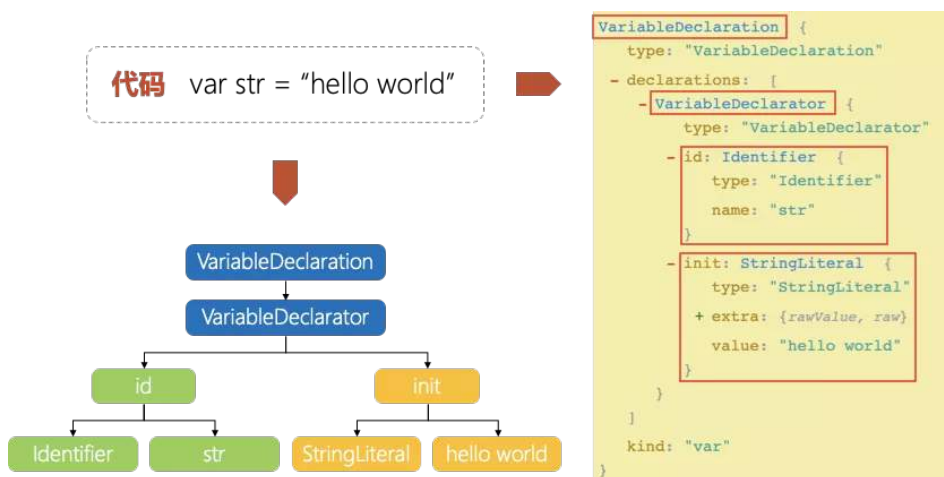
什么是 AST

在上文中，我们提到可以利用 AST 来做静态代码扫描，检测代码中是否存在某

些可能造成资损或者舆情的场景。那么问题来了，AST 是什么呢？

在计算机科学中，抽象语法树 (Abstract Syntax Tree, AST) 或简称语法树 (Syntax tree)，是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

这是一段引自百科上的解释，什么意思呢？让我们一起来看下面这个例子：



可以看到，非常简单的一句初始化赋值代码 `var str = "hello world"` 被拆解成了多个部分，并用一棵树的形式表示了出来。（如果想查看更多源代码对应的 AST，可以使用神器 [astexplorer](#) 在线尝试）

其实，我们每天日常工作都在使用的 js 代码编译工具 — Babel，它也离不开 AST。为了将 ES6 甚至更高版本的 js 语法转换成浏览器兼容性更好的 ES5 代码，Babel 每次都需要先将源代码解析成 AST，然后修改 AST 使其符合 ES5 语法，最后再重新生成代码。总结一下就是 3 个阶段：parse -> transform -> generate。

到这儿，也许你的心中会冒起一个想法："咦？前文就在说可以用 AST 做代码扫描，而 Babel 又恰好已经做了解析 AST 的工作，难道 ..." 没错，当你带着这个疑惑打开 Babel 官网 时，你会发现：真香 ~~~

Babel 不但完成了 AST 的解析工作，而且由于其编译 js 代码的使命，它还提供了一套完善的 visitor 插件机制用于扩展，而种种的这些都为我们的代码扫描工作创造了完美的条件。

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
// 编写自定义规则插件
const visitor = {};
// 源代码
const code = `var str = "hello world";`;
// code -> ast
const ast = parser.parse(code);
// 用自定义规则遍历 ast (即代码扫描)
traverse(ast, visitor);
```

如上所示，利用 Babel 提供的能力来做代码扫描就是如此简单，唯一要做的就是结合我们自定义的资损 / 舆情规则来编写 Babel 开放的 visitor 插件。(有关 "如何自定义 Babel 插件" 可以查看这份 Babel 插件手册，该文档介绍了如何编写一个 Babel 自定义插件，也是后文的基础)

解决问题

在简单介绍完 AST 后，让我们回归到本文的核心问题：面对以下这些可能发生资损故障的场景，什么 "千奇百怪" 的代码都可能出现，我们该如何做检测呢？

- 前端金额赋默认值
- 前端金额计算错误
- 前端写死固定金额 / 积分
- ...

寻找 "金额"

根据上文的描述，我们知道 "金额" 在前端就是一个高危分子，有关它的操作都容易造成资损。

一方面，这是因为 js 的数字 "精度" 问题 (老生常谈的 $0.1 + 0.2 = 0.3000000004$)

问题); 另一方面, 金额计算本就应该放在服务端更安全。

因此, 为了避免潜在的风险, 所有的金额计算操作都应该由服务端计算后下发给前端, 而前端只做展示作用。这也正是代码扫描的关键一步, 我们需要检测代码中是否含有金额的计算操作。

为了找出代码中的金额计算, 首先要做的就是识别代码中的 "金额变量"。对于这个问题, 我们可以使用简单粗暴却又行之有效的方法: 正则匹配。由于大家的金额变量名取得都比较有规律 (就比如 xxxPrice, PS: 可继续扩展), 我们可以用一个简单的正则进行匹配:

```
const WHITE_LIST = ['price']; // TODO: 可扩展
const PRICE_REG = new RegExp(WHITE_LIST.map(s => s + '$').join('|'), 'i');
```

根据 Babel 解析得到的 AST, 由于变量名均是 Identifier 类型的节点, 所以我们可以用一个简单的规则来匹配所有的金额变量:

```
const isPrice = str => PRICE_REG.test(str);
const visitor = {
  Identifier(path) {
    const {id} = path.node;
    if(isPrice(id.name)) {
      // 金额变量 匹配成功!
    }
  }
};
```

小试牛刀

解决金额变量的定位问题后, 我们再来看看 "金额赋默认值" 的检测问题。

```
// case 1: 直接赋默认值
const price = 10;
// case 2: ES6 解构语法赋默认值
const {price = 10} = data;
// case 3: "||" 运算符赋默认值
const price = data.price || 10;
// ...
```

如上所示，虽然金额赋默认值有多种写法，但是当它们被解析成 AST 后，我们却可以将其逐一击破。说到这，就不得不再次祭出 astexplorer 神器将上述代码分析一波。

case 1: 直接赋默认值

代码 `const price = 10;`



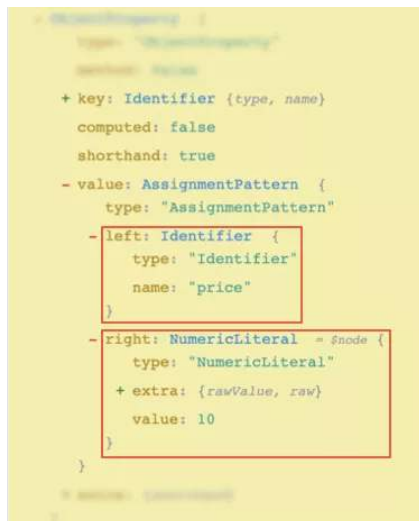
```
VariableDeclaration {
  type: "VariableDeclaration"
  - declarations: [
    - VariableDeclarator {
      type: "VariableDeclarator"
      - id: Identifier {
        type: "Identifier"
        name: "price"
      }
      - init: NumericLiteral = $node {
        type: "NumericLiteral"
        + extra: {rawValue, raw}
        value: 10
      }
    }
  ]
  kind: "const"
}
```

根据上面的 code vs AST 关系图可以看到，我们只要找到 VariableDeclarator 节点，且同时满足 id 是金额变量，init 是大于 0 的数值节点这两个条件即可。代码如下：

```
const t = require('@babel/types');
const visitor = {
  VariableDeclarator(path) {
    const {id, init} = path.node;
    if(
      t.isIdentifier(id) &&
      isPrice(id.name) &&
      t.isNumericLiteral(init) &&
      init.value > 0
    ) {
      // 直接赋默认值 匹配成功!
    }
  }
};
```

case 2: ES6 解构语法赋默认值

代码 `const {price = 10} = data;`



经过对上一个 case 的解析，我们其实已经初步掌握了如何用 AST 做代码扫描的要领，再来看 ES6 解构语法赋默认值的检测。观察上面的关系图，我们可以得出结论：找到 AssignmentPattern 节点，且同时满足 left 是金额变量，right 是大于 0 的数值节点这两个条件。代码如下：

```

const t = require('@babel/types');
const visitor = {
  AssignmentPattern(path) {
    const {left, right} = path.node;
    if (
      t.isIdentifier(left) &&
      isPrice(left.name) &&
      t.isNumericLiteral(right)
      && right.value > 0
    ) {
      // ES6 解构语法赋默认值 匹配成功!
    }
  }
};

```

case 3: “||” 运算符赋默认值

代码 `const price = data.price || 10;`

```
VariableDeclaration {
  type: "VariableDeclaration"
  - declarations: [
    - VariableDeclarator {
      type: "VariableDeclarator"
      - id: Identifier {
        type: "Identifier"
        name: "price"
      }
      - init: LogicalExpression {
        type: "LogicalExpression"
        + left: MemberExpression {type, ob;
        operator: "||"
        - right: NumericLiteral - $node {
          type: "NumericLiteral"
          + extra: {rawValue, raw}
          value: 10
        }
      }
    }
  ]
}
kind: "const"
```

经过上面的两个例子说明，想必 “||” 运算符赋默认值 的检测已经不在话下。不过这里需要特别注意一点：在实际的代码中，= 右侧的赋值表达式可能并不像例子中给的 “data.price || 10” 这般简单，而是可能夹杂着一定的逻辑运算。对于这类情况，我们需要改变策略：遍历右侧的赋值表达式中是否包含 “|| 正数” 的模式。

```
const t = require('@babel/types');
const visitor = {
  VariableDeclarator(path) {
    const {id, init} = path.node;
    if (t.isIdentifier(id) && isPrice(id.name)) {
      path.traverse({
        LogicalExpression(subPath) {
          const {operator, right} = subPath.node;
          if (
            operator === '||' &&
            t.isNumericLiteral(right) &&
            right.value > 0
          ) {
            // "||" 运算符赋默认值 匹配成功!
          }
        }
      });
    }
  }
};
```

变量追踪

根据上文的介绍，其实一些基础规则的代码扫描已经可以实现，然而现实中提交的代码往往会比上面给出的例子复杂得多。就拿金额计算来说，我们可以用下面的 visitor 来匹配任何有关金额的四则运算：

```
const t = require('@babel/types');
const Helper = {
  isPriceCalc(priceNode, numNode, operator) {
    return (
      t.isIdentifier(priceNode) &&
      isPrice(priceNode.name) &&
      (t.isNumericLiteral(numNode) || t.isIdentifier(numNode)) &&
      ['+', '-', '*', '/'].indexOf(operator) > -1
    );
  }
};
const checkPriceCalcVisitor = {
  BinaryExpression(path) {
    const {left, right, operator} = path.node;
    if(
      Helper.isPriceCalc(left, right, operator) ||
      Helper.isPriceCalc(right, left, operator)
    ) {
      // 金额计算 匹配成功!
    }
  }
}
```

然而，上面的规则却只能检测对金额变量的直接运算，一旦碰上函数调用就无效了。比如以下代码：

```
const fen2yuan = (num) => {
  return num / 100;
};
const ret = fen2yuan(data.price);
```

这是一个再简单不过的分转元金额单位换算函数，由于形参不具备金额变量名的特征，先前的规则将无法成功检测。为了解决“变量追踪”这个问题，我们还需引入 Babel 中的 Scope 能力。根据 官方文档 介绍，一个 scope 可以被表示成：


```
// 一个 scope
{
  path: path,
  block: path.node,
  parentBlock: path.parent,
  parent: parentScope,
  bindings: [...]
}
// 其中的一个 binding
{
  identifier: node,
  scope: scope,
  path: path,
  kind: 'var',
  referenced: true,
  references: 3,
  referencePaths: [path, path, path],
  constant: false,
  constantViolations: [path]
}
```

有了上面这些信息，我们就可以查找任何一个变量的声明以及任何一个绑定的所有引用。什么意思呢？

前文提到的变量追踪问题在于：原本是金额变量名的实参在函数调用时，形参可能变成了和金额无关的变量名。但是现在，我们可以借助 scope 顺藤摸瓜，先找到该函数的声明，然后根据参数的位置信息重新建立实参和形参之间的关系，最后再用 binding 检测函数体内是否含有对形参的四则运算。

```
const t = require('@babel/types');
const Helper = {
  // ...
  findScope(path, matchFunc) {
    let scope = path.scope;
    while(scope && !matchFunc(scope)) {
      scope = scope.parent;
    }
    return scope;
  }
};
const checkPriceCalcVisitor = {
  // ...
  CallExpression(path) {
```

```

const {arguments, callee: {name}} = path.node;
// 匹配金额变量作为实参的函数调用
const priceIdx = arguments.findIndex(arg => isPrice(arg));
if(priceIdx === -1) return;

// 寻找该函数的声明节点
const foundFunc = Helper.findScope(path, scope => {
  const binding = scope.bindings[name];
  return binding && t.isFunctionDeclaration(binding.path.node);
});
if(!foundFunc) return;

// 匹配实参和形参之间的位置关系
const funcPath = foundFunc.bindings[name].path;
const {params} = funcPath.node;
const param = params[priceIdx];
if(!t.isIdentifier(param)) return;

// 检测函数内是否有对形参的引用
const renamedParam = param.name;
const {referencePaths: refPaths = []} = funcPath.scope.
bindings[renamedParam] || {};
if(refPaths.length === 0) return;

// 检测形参的引用部分是否涉及金额计算
for(const refPath of refPaths) {
  // TODO: checkPriceCalcVisitor 支持指定变量名的检测
  refPath.getStatementParent().traverse(checkPriceCalcVisitor);
}
}
}

```

如上所示，借助 scope 和 binding 的能力，我们就基本解决了 " 变量追踪 " 问题。

检测效果

经过前文对基本原理介绍后，我们再来看下实际的检测效果。从代码扫描上线之后到本次 618 活动目前为止，我们对一批前端代码仓库进行了扫描，共有 1/7 的仓库都命中了规则。下面挑了几个例子来感受下藏在代码中的 " 毒药 " ~

Bad code 1:

```
let {  
  // ...  
  rPrice = 1  
} = res.data || {};
```

如上所示，当服务端返回的数据异常时，一旦 `res.data` 为空，那么 `rPrice` 就会获得默认值 1。经过代码分析后发现 `rPrice` 代表的就是红包面额，因此理论上就可能会造成资损。

Bad code 2:

```
class CardItem extends Component {  
  static defaultProps = {  
    itemPrice: '99',  
    itemName: '...',  
    itemPic: '...',  
    // ...  
  }  
  // ...  
}
```

如上所示，该代码应该是在开发初期 mock 了展示所需的数据，但是在后续迭代时又没有删除 mock 数据。一旦服务端下发的数据缺少 `itemPrice` 字段，所有的价格都将显示 99，这也是颗危险的定时炸弹。

Bad code 3:

```
const [price, setPrice] = useState(50);
```

如上所示，这个 hooks 的使用例子默认就会给 `price` 赋值 50，如果这是一个红包或券的面额，意味着用户可能就领到了这 50 元，从而也就造成了资损。

Bad code 4:

```
// price1 为活动价, price2 为原始价  
let discount = Math.ceil(100 * (price1 / 1) / (price2 / 1)) / 10;
```

如上所示，这是一个前端计算折扣的代码案例。按照前文提到的约定，凡是涉及到金额计算的逻辑都应该放在服务端，前端只做展示逻辑。因此，如果能检测出这类代码，还是可以从源头上避免不必要的风险。

Bad code 5:

```
Toast.show('恭喜您获得双 11 红包');
```

如上所示，这是一段字符串常量中包含大促关键字（双 11）的代码。由于目前是 618 大促，如果用户看到这个 toast 提示就不合适了，虽然不会造成资损，但可能会引发舆情。因此，原则上来说，前端使用的兜底文案就应该是通用型文案，凡是此类带“时效性”的文案要么走配置下发，要么服务端下发。

总结与展望

本文先对 AST 做了简单介绍，接着围绕资损防控问题介绍了如何用 AST 做代码扫描的基本原理，最后再以实际仓库的扫描结果验证检测效果。目前来看，针对一些通用型的问题，通过代码扫描确实能够发现一些藏在代码中的潜在资损 / 舆情风险。但是对于一些和业务逻辑强相关的资损风险目前仍不容易检测，这还需从其他角度点进行突破。

如何实现代码自动生成？

每年大促前一个月都是奋战与忙碌的时节，不仅业务上在不断迭代创新，技术上也在推陈出新，需求推动技术变革是一个正向演进的过程，但革新是需要成本的，每一次技术与标准的革新都带来一场翻天覆地的大改造。如果我们能将需求与产物划上等号：需求即代码，那么我们只要找到两者之间的关联关系即可通过需求自动产出代码了，那岂不是乐哉美哉 (diao zha tian le)。

本文主要围绕自动化生成代码的目标，讲述我们在这一过程中的所思所想，以及我们在 618 期间的阶段成果实践。

前言

自然语言是及其抽象且强依赖上下文的，其整体表现为不精准，而代码是多维具象且有强相关性的，其整体要求为精准。这样两个不同概念的东西怎么找到他们之间的相关性呢？

需求和代码之间没有一个标准的转换模式，通过标准转换器来实现是不可能的，唯一可以采用的方案就是运用机器学习，并通过大量的样本来增强之间的关联性。但正如前面所提到的，代码和需求都不是对局部的简单理解就能完成的，而是对一个整体的理解，因此对每一句话的理解都不能是独立存在的，而是相互交织在一起的，这就增高了对整个需要理解的纬度，据鄙人所了解这个难度是非常大的。

但有时一个问题的难度大小也是由当下技术边界所决定的，比如 3D 打印的出现，就彻底将需求和产出物划上了等号，过去很多繁琐的工序都可以舍弃了，只需要一种实现方式就能让人们在家就可以下载设计稿并自己完成产品的实现。

当然 3D 打印的出现并不能解决代码自动生成的问题，3D 打印只是在生产过程中运用了统一化的解决思路，而从需求到产出的过程依旧需要依靠设计图来对产出物

的各项尺标数据做约束。所以设计稿的标准化一定是统一化的前提，3D 打印更像是一个标准化出码的机器，而指挥这台机器如何生产的正式标准化下的设计稿，因此对我们来说需要思考两个问题：

1. 自然表达的需求如何转变为标准化的“设计语言”。
2. 如何通过标准化的设计语言产出代码。

我把第二个问题看作和 3D 打印一样，是一个标准化的问题，而有了标准化的设计语言再转到 AST 是很简单的事情，困难的是第一个问题，如何将需求转为标准化的设计语言。这是我们需要核心思考的问题，并且这个课题对于运用机器学习来解决也同样是非常困难的，那我们首先是否应该来了解一下目前机器学习的边界呢，并怎样拆分成具体目标去实现它。

聊聊 AI

当我最早了解人工智能的时候是这部著名的《AI》。



电影中的小男孩大卫是个机器人，但是他被赋予了情感，他历经万险，为了一个

注定幻灭的结局。“等我变成了真的小男孩，妈妈就会带我回家了”。

“当人工智能开始寻求自我认同时，他就已经和人类没有区别了”。如果这个世界是虚拟的，造物主一定不会让我们发现丝毫，这也是当时这部电影令人感同身受最恸哭内心的部分。这也正验证了人与机器的本质区别，即是否能感受自我的存在。

回到当下，现实的 AI 并不像电影里的那样，既没有情感也没有逻辑，更没有自我意识。比如知名的 AI AlphaGo，从技术上讲，它之所以能够击败人类的围棋高手，是因为它具有由“小”变“大”的能力，人类将十几万的围棋博弈输入到它的“大脑”中，然后它就会进行自我“对战”，进而产生几百万甚至几千万的围棋博弈或者叫做棋谱，所以它在应付人类的围棋博弈时，就能够“从容不迫”。但他产生的模型的过程不能举一反三，当改变棋盘的尺度或改变规则时 AI 就再无法做出有效应对了，它必须得进行重新的学习，而人类只需要通过理解规则即可做出应对改变。

与 AI 相比，人类似乎能够解释和理解其所生活的环境，而计算机只是一台从事匹配工作的机器。哲学家约翰·塞尔 (John Searle) 很久以前就通过他的“中文房间”思维实验 (Chinese Room thought experiment) 指出了这种差异。在本实验中，他将自己关在密闭的房间中。

这个实验是这样设计的：Searle 他既不会说也听不懂中文，但是在这个房间里有一本很大的手册，里面有一系列的假设陈述。通过门上的一个插槽，他可以接收汉字，查找手册中的字符串，并找到要通过插槽送出的字符串。显然，塞尔可能会让房间外的人相信他真的懂中文，但事实上他并不懂，他只是在匹配符号而已。



虽然从外部无法区分房内的人到底是否“懂得”中文，但是实质上房内的人是不懂中文的。因此以“是否无法区分‘懂得中文’和‘表现得完全像懂得中文’这两个现象”而将其两者认为是等同的，这个观点是不能够成立的。

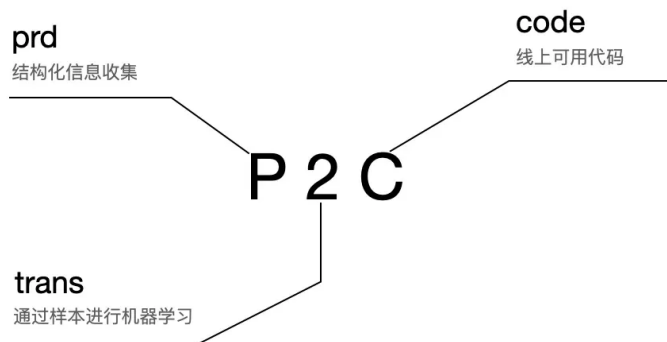
AI 并不神奇，它预测的基础甚至和《易经》和 中医理论 都没有太大区别，如果有，我觉得差在了灵魂和哲学的思考上。而对我们来说 AI 是否真的能产生情感还是是否能真正能够理解处理信息，对我们来说都不是重要的，我们只需要他看起来可以做到准确预测即可，并不需要一个真正的人工智能。

用传统机器学习进行建模预测，其本质更接近于搜索，这是一个看起来会比较傻的弱人工智能，并且所有的预置算法都是由人类来写的，AI 只是一个按算法规则来进行分类 / 聚类的机器，比如对单个图形或句子进行预测，这些数据都是相对结构化的数据，而对于整篇文章或全幅大图做分析则需要强人工智能来完成，一般我们会用到深度学习，并需要使用多层处理单元的巨神经网络利用强大计算能力和改进的训练技术来学习大量数据中的复杂模式。原理是计算机在学习特定问题时，需要大量输入这个问题相关的学习材料也就是数据，然后在计算机通过算法和模型来构建对这个具体问题的认知，也就是总结出一个规律，那么在以后遇到相似问题时，计算机会把收集的数据转成特征值，如果这个特征值符合这前面规律里面的特征值，那么这个事物、行为或者模式，就可以被识别出来。

背景

聊完 AI，我们回归正题，说一下什么是 P2C 以及为什么要做 P2C？

P2C 即通过 prd 自动产出 code，即是我们开头所讲的需求即代码的终极目标。



讲完 P2C 是啥，我们再回过头来聊聊我们为什么要做 P2C？

在开发链路中耗时较长的两个环节是「沟通成本」和「编码成本」，「沟通」一般是需求明确的过程，一般体现在各种需求评审上。而前端的「开发」成本则主要可看作两个部分：视图代码与逻辑代码两个部分的开发工作。目前我们团队的 D2C 产品（imgcook 设计稿到代码）已经能够帮我们准确的生成大部分的视觉代码了，甚至可以分析出部分简单的业务代码，因此对前端开发来说在开发中的成本主要花在不通用的逻辑表达部分，即粘合剂。

对于这些“粘合剂”逻辑，在产品的 prd 中我们可以找到准确答案，但有时产品的 prd 是不够结构化的，信息丢失也比较严重，这非常不利于机器学习进行学习和理解。因此我们需要一个平台来规范产品的需求，使产品能够按照要求进行结构化的录入。这就是我们做 P2C 的初衷。

挑战

前面我们对 AI 的能力已经有所知了，而现在我们要怎样一步步达成我们的终极目的“Auto Code”呢？

我们知道 AI 目前还不能够像人类一样来分析数据关系，而大部分 prd 都是以文本为主的非结构化信息，这时候我们就需要利用 NLP 技术对信息进行处理并根据意图翻译为可进行转化的逻辑描述 DSL，因此我们通过相对具有结构化文档的方式进行需求的收集，从而简化机器学习的难度。

虽然我们可以将原本海量信息的大文档进行结构化的信息表达，使局部信息清晰可解，但此降维操作也产生了新问题，就是关键点与关键点之间的联系变得更弱了。

我们的初衷是希望能降低需求沟通成本，并在此过程中积累智能化出码的能力，所以我们只是设计了简单的结构化收集方式，并不强制改变 pd 原有的需求录入方式，因此需求文档内容有着很高的不确定性，不仅存在上下文关系依赖，还有很多未描述的知识依赖，并且内容也都是允许其自由组合天马行空。

知乎有一篇[买苹果的例子](#)，很好的诠释了一个简单的需求背后所牵涉的复杂的关联关系。因此可见单纯的通过需求来推导出产出物是非常难的工作。既然如此为什么不直接走向另一个极端：严格的结构化需求录入呢？虽然严格的结构化信息录入可以帮助更清晰的将一个开域问题收拢回一个闭域问题，但是会让需求产生的过程变得更困难更具专业化，这和我们服务 pd 更好的进行需求录入的初衷是相违背的，另外我们也不希望通过标准化的方式去限制业务的创新。

因此如何既能让 pd 录入需求的工作变得简单，又用能借此发展我们的“Auto Code”事业，是我们最大的一个难题。另外对于整个出码目标的技术链路我觉得目前也存在两个难点：

1. 获取大量自然语言意图分析的训练样本：对于 pd 的同一个需求可以用一千个不同的句子来表达，同样一个需求表达也可以对应一千种实现方式。要想将自然语言的需求描述同实际产品表达进行准确关联需要对大量的样本数据进行学些。
2. 将所有自然语言逻辑使用统一的 DSL 描述：如开篇提到的 3d 打印，统一化的设计稿是实现转换的基石，因此一套描述需求的标准化 DSL 是必不可少的。

推导

既要让 pd 录入需求时能录入的舒服，又得要求能按需出码，似乎两者是矛盾的，但是困难也都不可能是一朝就能解决掉的。

所以我们打算分步骤实现目标，首先我认为短时间对需求的结构化信息分析还很难达到从局部到整体的突破，文本的灵活性也会带来诸多问题，比如模块信息的增删改造造成的信息错位，因此我们觉得第一步要从以 P2C 为核心出码的逻辑变为以 D2C（即 Design2Code - imgcook），因为视觉是产出是最终产出物比较接近终态的一个阶段，通过视觉我们将会更容易分析缺失部分的内容，然后结合 P2C 内容从中寻找补充的关键信息，比如一个视图是否是 ScrollView 单从视觉上很难辨认，需要结合从 prd 中获取的信息，甚至 prd 中没有详细描述时还需要跟产品属性进行分析，例如这是一个音乐播放器的界面，那么其中应该包含什么功能组成，哪些功能组成一定会是 ScrollView，这就需要一些知识依赖的输入，而这些宝贵数据资产都是需要脚踏实地不断积累的。

我觉的 D2C 和 P2C 的结合一定是 $1+1>2$ 的，对于 D2C 视觉出码有了知识依赖做背景，对于视觉稿的整体判断及组成可以有了很好的依据，另外 P2C 也能够基于准确的视觉做精准的逻辑补充。因此以视觉稿为主导出码，以 prd 为辅助出码的思想浮出水面。

即我们第一阶段重修了目标：将原本的 AutoCode 目标先降低为 NoCode = 通用逻辑组件 + ProCode(AI) 目标。

这个过程通俗来讲就是 通用组件 + 配置信息 的模式，配置信息即 ProCode 的部分，该部分即为需要 AI 自动产出的内容。因此想要完成自动出码，就需要对 ProCode 的部分进行样本制造和监督学习。

我们来总结一下我们的三步走计划：

1. 需求的结构化收集（为了降低 NLP 的分析难度）。

2. 创造结构化需求到标准逻辑表达的样本 (为机器学习 ProCode 部分提供充足样本)。
3. 通过机器学习对样本进行学习, 以达到 AutoCode 的目标 (长期目标)。

即我们需建立两个平台, 我们称之为 P2C 的 2.0 代和 1.0 代。P2C 1.0 负责打标需求数据, 并为 2.0 的智能出码提供庞大准确的数据。而 P2C 2.0 则主要进行需求的结构化收集并与 1.0 的样本进行关联学习, 训练出码模型, 最终达到创作和自动完成需求的目的。

两个平台所代表的关键词分别是:

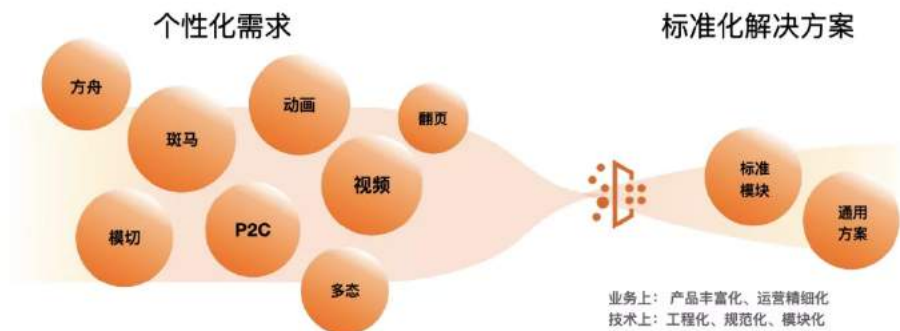
- P2C1.0: 精确收集, 精确产出
- P2C2.0: 开放收集, 创新产出

那么精确收集样本就成为我们 P2C 1.0 的主要工作方向。

打标样本一定要从实践出发, 并能在该阶段帮助需求真实的进行产出, 即我们提供的打标平台能真实的完成业务需求, 因此我们决定开发一个可视化编排的平台, 通过该平台来收集样本数据, 并且可以顺手把需求也消化掉。

那么问题来了, 这么多的样本打标的工作到底由谁来完成呢?

我们通过调研发现运营的角色在可视化编排方向上, 是有着非常高契合度的。为什么呢? 其一, 运营都具备模块搭建的能力。其二, 很多运营的需求都不一定有机会能够落地, 主要原因是开发资源不足导致的, 这也是多年来存在的业务与技术矛盾, 前端这十年来一直朝着工程化、规范化、模块化的方向发展, 本意是为了更有效的重用业务能力以达到解放生产力, 而产品却一直在朝着丰富化、精细化的方式来运作, 各个业务方不再是单单为了满足功能诉求而更讲求的是用户心智。最终致使现在对一个个性化需求的提出往往是用一个标准化方案来落地的。当业务方为资源不足妥协时, 其业务整体感官也会越来越平庸化、趋同化, 最终导致产品对用户的心智弱化。



因此对于运营的诉求是希望能将创新和业务思考带入到自己的产品中，而不是简单的拆解为一个个的标准实现。如何拉大与竞争对手的运营差异、交互差异、创新差异、视觉差异才是对于运营真正的核心价值。

在 AI 的介绍篇中我们讲了对于复杂的逻辑关系我们依然可以采用抽象的组件化来实现，而这部分实现对比传统搭建体系的组件颗粒度更细，传统搭建一般是对模块的自定义配置，而在我们的编排体系里最小的组件应该是原子化的不可再被拆解的，比如一个 Image 组件，而对组件的配置可以是一段编排好的逻辑实现，因此它能有着同代码一样的绝对灵活性。

拆解

在前面 3D 打印的内容中我们提到，自动化统一化生产的前提一定是要有一个标准化的表达做支撑的，这个中间 DSL 该如何设计才能承载住全量的需求内容呢？另外通过什么样的方式可以让运营能接受和理解这部分的逻辑编排？

1. 首先我们解决的问题一定是有边界的是处在一个业务领域的，并在这个业务域下进行分析。因此我们分析了一下我们最常见的业务形态 - 模块。而据统计“相同业务能力，不同技术实现”的模块在整个天马体系中的占比非常的高，单单拿通用商品坑来看就有上千个，而这些模块大部分都是相同的逻辑实现，也有很多是处于不同时期的技术产品。

2. 一个模块在前端的开发设计里, 大概可以分为 4 部分, 通用组件, 通用 Action, 视觉代码, 驱动逻辑。
3. 我们可以看到视觉和通用组件以及 Action 都是已经有了的, 变动周期比较低。而驱动逻辑部分是业务中的主要实现逻辑, 如何让这部分逻辑能让运营进行产出呢?
4. 首先我们将这部分能力进行一个拆解, 我们得出一个初步的结构, 就是由 运算符 + 表达式 + 基础 action + pipeline 组成的表达结构, 这个表达也可以组成一个新的 action, 再加上 trigger, 整体就完成一个 workflow 的能力了。



$\text{new Expression} = \text{Variable} + \text{Operator} + \text{Expression} + \text{Pipeline}$

$\text{new Action} = \text{Expression} + \text{Action}$

$\text{Workflow} = \text{Action} + \text{Expression} + \text{Trigger}$

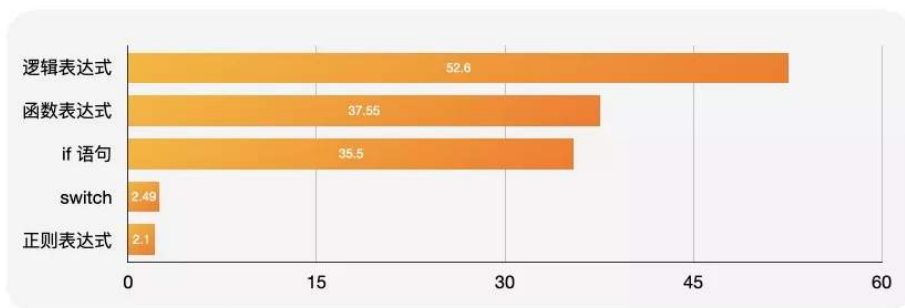
业务模块的四个部分中, 驱动逻辑就是我们主要核心产出的内容, 根据我们的经验这部分 ProCode 大部分都是表达式 Expression 内容。因此我们只要能让运营能

够自主的完成 ProCode 部分的编排即可高级定制需求的产出物。

让运营完成所有的 ProCode 也显然是不现实的，但是对于 IFTTT 这种简单的逻辑表达，只要将这部分内容的表达可以使用更接近自然语言的表达方式，自然会降低整个的理解难度。

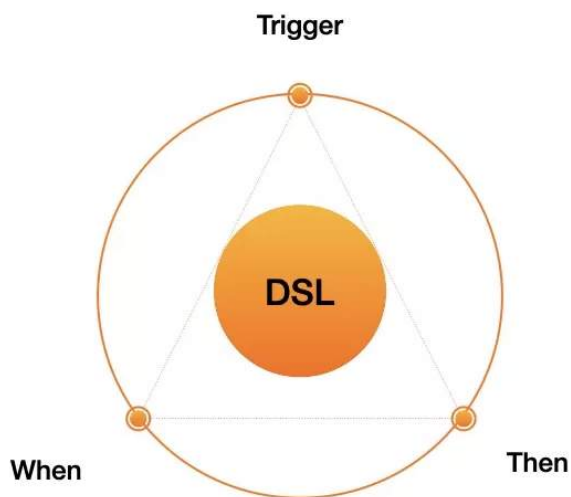
验证

为了验证我们的拆解过程，我们把 pmod 分组下的 9 千多个仓库进行一个拆解分析，发现平均每个模块都有逻辑表达 52.6 个，函数表达 37.55 个，if 语句 35.5 个，这三大部分实现是符合我们的拆解预期的。这部分的逻辑大部分都是可以用 IFTTT 的模型套用。



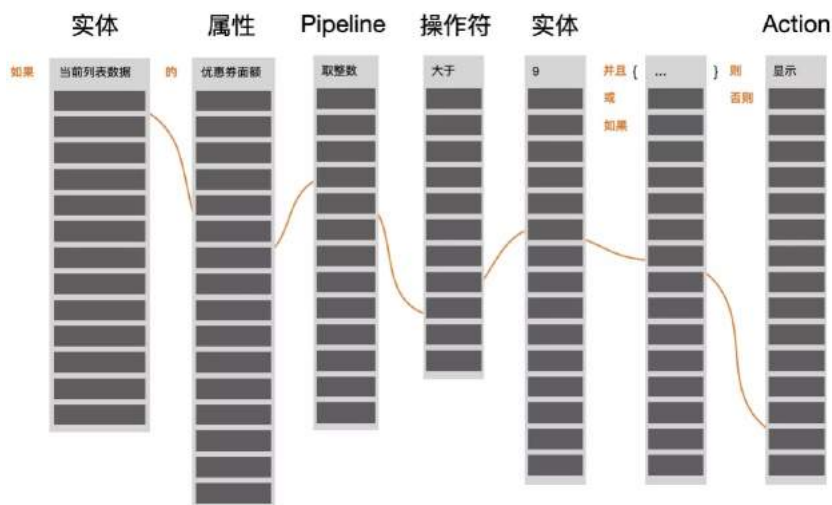
而一个复杂的模块，光有 IFTTT 是不够的，逻辑的表达是具有很强的依赖上下文结构的，但是为了降低整体的复杂度，我们需要对逻辑进行拍平表述，对拍平后的逻辑我设计了一套 StateLink 的工作流管理库，这里就不详细介绍了。

最后我们通过一个 When + Then + Trigger 三元素的 DSL 就能描述我们所有的业务逻辑部分了。这个 DSL 也为我们 2.0 版的出码提前做好了准备，同时也能应对因技术升级而导致模块需要重新开发的尴尬。



为了让运营能够用得懂用的轻松，我们采取了中文编排的方式，即首先根据选取的视觉对象进行一些具象的操作表达。

为什么我们决定采用中文编排呢？中文编排是怎样一个编排方案呢？



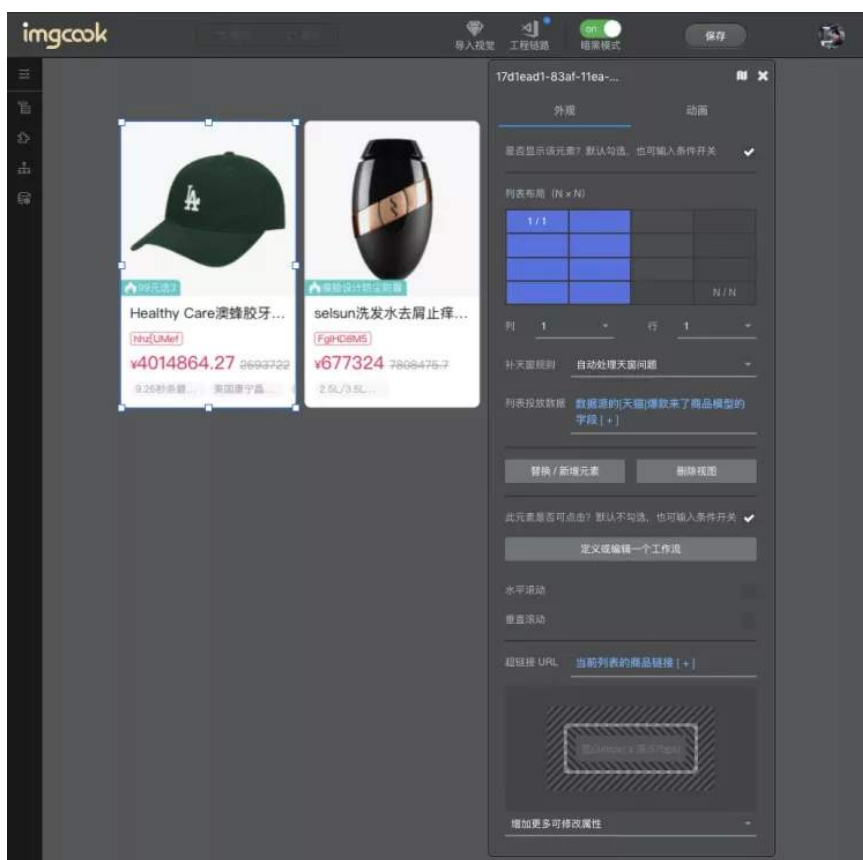
我们将 pd 需求的一句可以拆分为上图这样的表达，对于使用者只需要顺着决策树表达自己的意图即可，类似于输入法，当我们输入一句话的时候，输入法通过预测词可以表达所有的表达意图，比如输入“我”，那么一定能够枚举“我们”、“我的”、

“我 ..” 等词并可以一直预测下去，而每个词的预测都是一个有限的枚举，同样 pd 在描述一个需求时我们也可以根据语法和目的预测出所有的可能项。

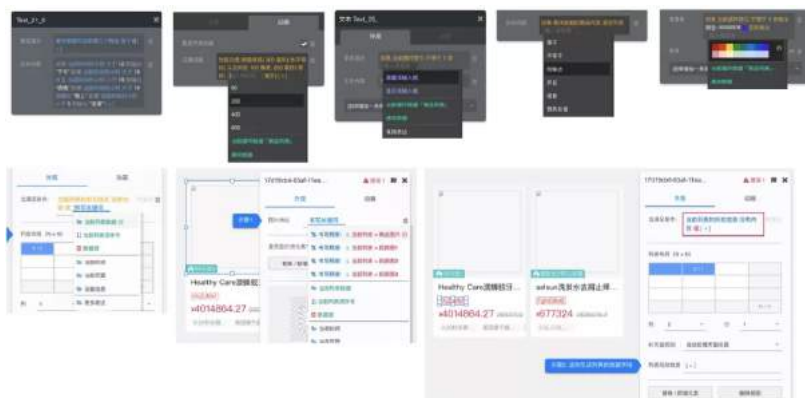
现状

我们将常用的逻辑表达拆解为中文关键词，通过关键词关系可以让运营通过编排一句话的形式来表达事物的逻辑关系。

我们目前已初步完成了第一个版本的运营逻辑编排平台，通过该平台可以先让运营能够进来满足一些简单的需求。



运营主要通过可视化的方式对元素上的逻辑进行中文关键词表达的编排。



案例

“爆款来了”会场中的 6 个模块，由运营自行编排发布上线。





未来

虽然人工智能目前还有很多的不足和局限性，但随着深度学习的不断进化，我相信从需求理解到需求实现全过程的自动化会离我们越来越近。



我相信未来不仅可以实现从需求到代码的智能生产，也将会从智能生产覆盖到智能视觉、智能测试、智能运营的全链路智能场景。

就像我们的大厨，一开始是需要人工切菜人工炒菜，后来是人工按菜谱备料再由机器辅助炒菜，再然后是直接输入菜谱机器就能自动根据菜谱备料和炒菜，而未来只要描述想吃什么口味就能自动做出符合用户的菜。

P2C 的未来就是将过程描述简化为目标描述，一个需求只需要表达诉求和目标而不再需要描述详细的生产过程及规则，只需简单的几句话我们就能够知道用户想要什么该产生什么样的交付物了，这就是我们未来的“智能大厨”。

前端通用模块在手淘业务中的实践

本篇的作者是来自于行业与工作台团队的玄扈，为大家介绍行业魔方项目。

前言

过去的一年，笔者在天猫行业支持了成批的行业频道前端建设，深刻体会到业务的发展促使快速建场、高效用场的需求愈发强烈；而行业前端的开发方式仍是劳动密集型，通过加大外包资源投入 + 玩命加班来完成越来越多的新业务需求，而外包和加班都会导致代码可维护性进一步下降，对频道这类长尾业务弊大于利。

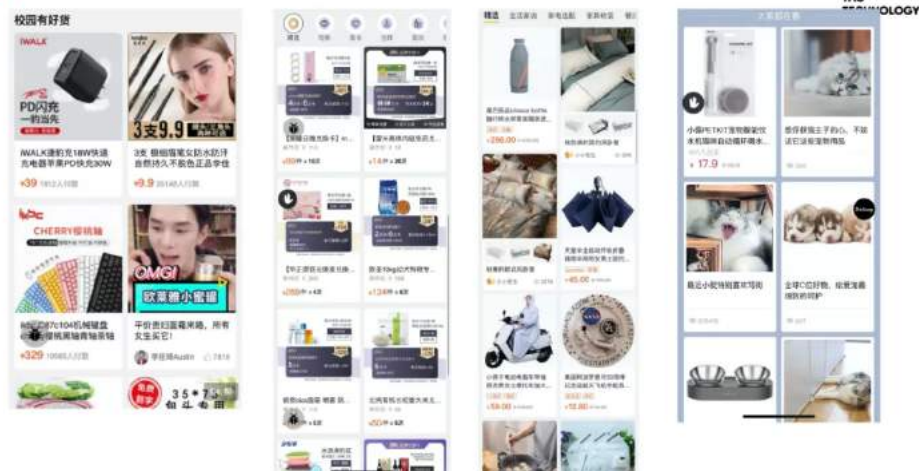
得益于淘系前端的积累，现在我们可以借助完善的天马搭建体系、Rax1.0 跨端开发框架、imgcook 智能生产这些贼棒的工具完成一个个模块的开发并搭建出一个完整的小程序频道，但在行业这样的生产关系下，我们希望能沉淀出一套更高效的生产体系来支持我们高 (hao) 效 (hao) 工 (shui) 作 (jiao)。

去年底开始，天猫行业已与 UED、产品团队合作完成了 TaoUI 组件规范，并建设了织网组件中心来支撑行业沉淀下来的物料，那么，如果按照一定的规范，使用直接的数据模型直接驱动组件，是不是大部分普适的模块就不需要开发了？于是，行业魔方项目应运而生。

■ 业务共性多，物料可被统一，但数据源复杂



可被编排



可被编排



TL; DR

我们想要提供一套供行业业务快速搭建出行业页面并高效运营维护的模块生产 & 使用平台。

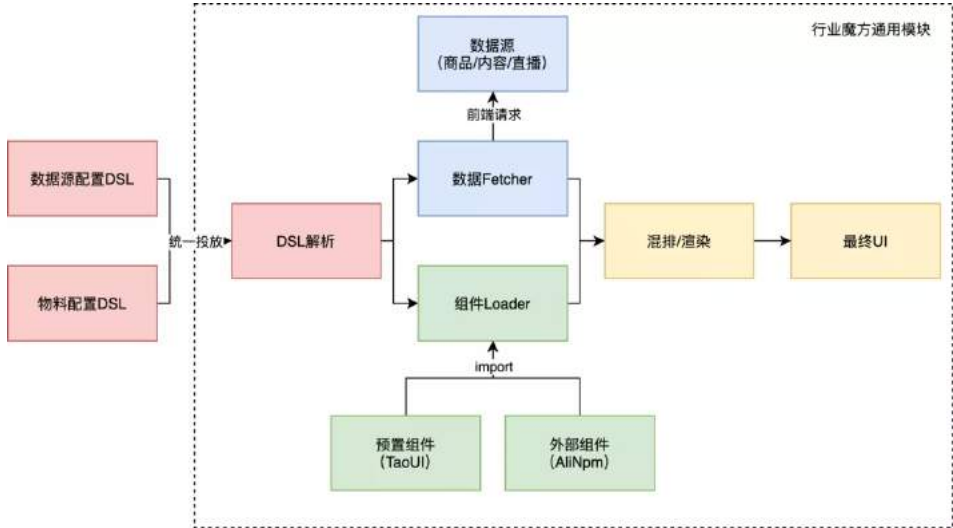
石器时代

去年底开始推进 TaoUI 组件落地时，正是行业频道需求的爆发期，我们借此机会沉淀了一批以 行业魔方 为名的通用模块用以支持业务。

模块名称	模块名称	特点	开发量	状态
rgemod-bc-bannerbar	搜索栏	主搜按钮、三搜按钮、搜索框	中低量	已交付
rgemod-bc-banner	Banner	支持轮播、跳转、多种样式	中低量	已交付
rgemod-bc-feeds	Feed 流	支持多种内容、商品、直播、视频	中低量	已交付
rgemod-bc-navigator	沉浸式侧边栏	三端兼容、支持多种交互	中低量	已交付
rgemod-bc-items	商品列表	支持轮播、跳转、多种样式	中低量	已交付
rgemod-bc-floor-title	楼层标题	支持轮播、跳转、多种样式	中低量	待开发
rgemod-bc-floor-slider	楼层轮播	支持轮播、跳转、多种样式	中低量	待开发
rgemod-industry-general-enrapp-1a3	通用一屏三		中低量	已交付
rgemod-category-entry	品类、店铺、频道		中低量	待开发
rgemod-industry-carousel	通用轮播		中低量	待开发



这批模块成功帮助我们在短期内支持了 7 个淘宝 / 天猫行业的频道业务，以及服饰行业新风尚的营销场景混合 Feeds 需求，有效释放了研发压力。在这批模块中，我们首次引入了数据驱动 UI 的形式，通过运营在搭建平台配置的数据源 DSL 来编排数据获取行为，并进行数据与组件物料的组装。



- 这套方案支持了我们几个月的业务开发，但我们也发现了其中的几个问题：
1. 运营需要在搭建平台同时配置数据源参数和物料信息，由于 schema 复杂，操作繁琐，扩展性极差。
 2. 随着 TaoUI 组件的应用持续增加，定制点亦越来越多，我们提出了基于主题

进行配置，主题允许业务自行定制，但通用模块不可能透传所有参数到组件。

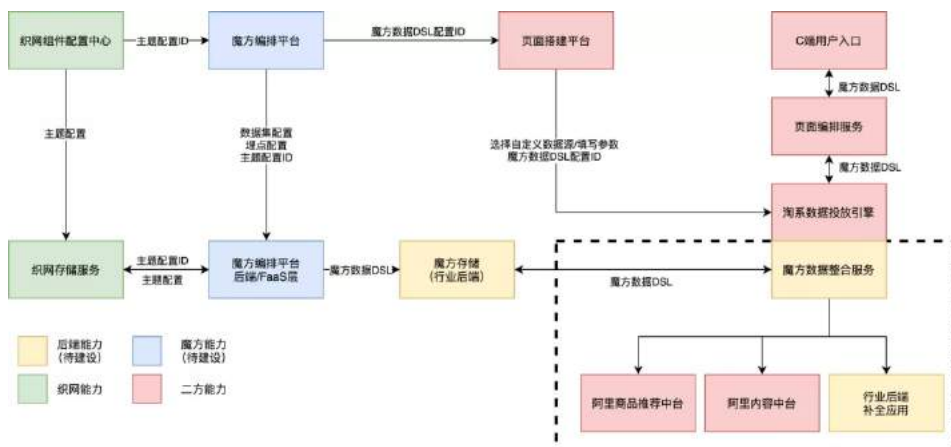
3. 现有方案的数据获取基于在前端发起请求，在数据源较多或混排规则复杂的场景下性能严重受限。
4. Rax 编译时小程序方案不支持动态 import 组件，通用模块引入业务定制（外部组件）能力实际无法落地。

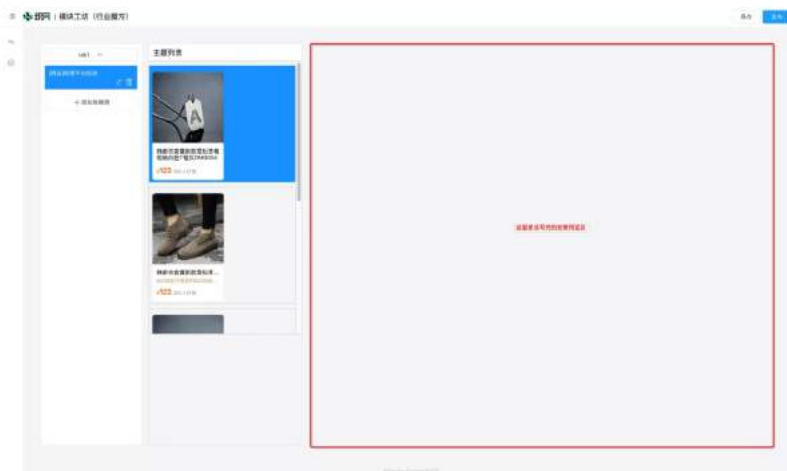
所以，我们决定发展到下个时代～

铜器时代

基于此前在行业魔方通用模块中沉淀的经验，我们决定做出以下几点调整：

1. 组件基于主题进行配置，主题（元素样式集合）支持业务定制，产出了织网组件配置平台。
2. 与行业后端共建数据链路，将对固定模型 DSL 的数据源获取、混排、分页等能力在服务端落地，前端模块回归渲染本职。
3. 自建织网模块工坊（行业魔方）平台，用于配置数据源、物料主题，并用于最终发布 DSL、生产模块。

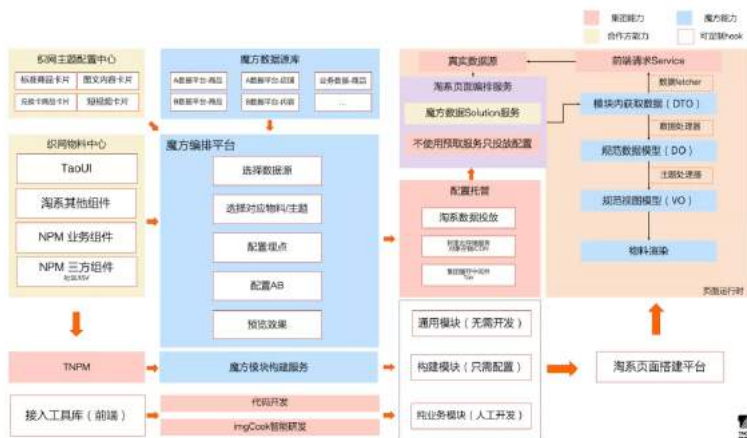




这套方案现已小步快跑支持了天猫服饰行业 618 会场的内容混排 Feeds 流，通过更通用、轻量的方案为营销会场带来了更丰富的体验，让用户不仅买得爽，还能看得爽。

钢铁时代（未来）

行业魔方的能力目前还是较为初级的阶段，我们希望在未来能够通过体系化的建设，帮助业务更快更高效的编排数据 -> 生产出符合规范且有调性的频道模块，一步到位，从而高效支持频道快速建场、频繁迭代的需求场景。



同时，这套方案因为完全依靠编排数据和物料来渲染频道模块，在基于数据进行物料的混排、对不同种类的数据、不同的物料进行 AB 等精细化运营场景下，具备天然的接入优势，可以完美实现算法控制数据，数据决定 UI。这一理念已在天猫服饰的内部数据平台基于行业魔方的方案有所落地。

2020 年，我们该如何学习 WEB 前端开发

本文作者小问，为大家讲解学习 web 前端开发中的几个注意事项。

我们可以把学习路线比作游戏中的段位上分，在不同的分段都有自己的定位和要锻炼的事情：

青铜—从零开始小学生：怀着满腔的热血，看到了这一个行业的希望和未来，准备开始学习 Web 开发知识。

- 先通过 w3cschool 等免费学习资源把 HTML、CSS 和 JavaScript 的基本操作学会了。
- 写一个简单的表白页面送给你的女 / 男朋友，展示一下自己努力的成果，如果没有就当我说。
- 白银 一懵懵懂懂初学者：懂得如何使用 HTML、CSS 和 JavaScript 三大件来实现基本页面开发功能。
- 选择一个可以覆盖多种场景、可以随自己意愿调整难度的项目尝试实现，如博客系统、记账本、Markdown 编辑器等。
- 从 React 和 Vue 这两个框架中选择一个进行学习。
- 黄金—轻车熟路新玩家：懂得使用框架来实现上面所举例项目。
- 学习 Redux、Vuex 或者 MobX 等状态管理工具，并将他们使用到前面的项目中。
- 思考状态管理工具为你的项目带来了什么好处。
- 铂金 V—初出茅庐新司机：懂得如何使用脚手架创建项目，并且能将代码结构根据模块化的思想进行安排。
- 学习 TypeScript，对前面的项目进行重写，注重对数据结构和类型的控制。
- 学习 Node.js，试着配合数据库实现一个比博客系统更为复杂的 CMS（内容管理系统），如图书馆管理系统、仓库管理系统。

- 铂金 I—基本上手好司机：如果是懂得如何利用 Node.js 或 TypeScript 编写业务代码的。
- 思考在前面使用框架开发的项目中，有哪些代码是重复冗余的，有哪些逻辑是可以在多个组件之间共用的。
- 学习利用 ES2015 或更新的 JavaScript 标准，逐步替换使用框架所编写的代码。
- 钻石 V—淡定自然老司机：如果是对逻辑抽象、模块封装有了一定的理解和经验的。
- 思考如何使用纯 JavaScript 对业务组件中的非渲染、非 DOM 相关代码进行抽象。
- 引入单元测试工具，对纯逻辑代码进行测试，争取覆盖率达到 80% 以上。
- 钻石 I—赛道新手初学者：如果上面的条件你都已经满足了。
- 思考不同的代码哲学（OO、FP 等）、不同的代码结构（MVC、MVVM 等）的区别。
- 思考不同的框架之间设计的初衷，思考不同的编程语言中对同一类问题不同解法的区别。

到这里我划了一条从 0 到高级前端工程师级别的纯技术路线。相信有不少有经验的同学会发现中间我省略了不少内容，但也不难发现路线中从前半段的“学习”逐步变成后半段的“思考”。优秀的工程师除了需要有在纯技术领域的沉淀以外，还需要更多对技术、团队、ROI（投资回报率）的思考，当然这依然不足以支撑我们平稳地渡过“程序员 35 岁危机”，前面的路还有很长，钻石往上还有王者呢，谁说程序员就是青春饭碗的？

回想起很多年前我也跟你一样是一个完全的新手，从 0 开始慢慢自学摸索 Web 开发，甚至后来我也没有进入科班学习计算机，那么来听听我作为一个“前人”是如何完全靠自学至今的故事吧。

我的从 0 开始

我是一个完全从自学开始的前端工程师，想起来第一次接触前端就是初中那会特别流行合租 VPS 然后注册一个 .tk 的免费域名。而作为一个刚入门 Web 开发不久的小屁孩来说，用这种方式一探“大人的世界”属实让人兴奋。而当时最流行的博客管理软件就是用 PHP 写的 WordPress，作为一个十分成熟的 CMS 软件来说 WordPress 当时就有了非常丰富的社区资源，比如主题、模板、插件等等。

而作为一个十分注重个性化的小屁孩来说，当然是要自己做一个主题的啊！于是我就从此踏上了 Web 开发的不归路，在此之前我所接触的都是 Visual Basic 这样的 Native 的语言。

以 WordPress 主题作为切入点，我开始学习 PHP 用于调用 WordPress 的 API 并输出内容、学习 HTML 用于写主题的模板、学习 CSS 用于“装潢”我的博客、学习 jQuery 用于实现页面动态效果。是的，那个时候基本上大部分人接触的是 jQuery 而不是 JavaScript，一个 \$ 函数就可以完成非常多的效果这让我第一次感受到了“框架”所带来的价值。于是便一步一步地发生了以下事情（不一定完全对，毕竟时间过太久了）：

1. 我发现页面上的一些样式效果无法在 IE 浏览器上正常显示，于是我就开始到网上学习 CSS 在 IE 的各种特殊处理，包括 reset.css、normalize.css 等工具的使用。
2. 每次点击链接都要刷新页面，在那个网速不怎么好的年代体验非常糟糕，于是乎就开始研究怎么用 jQuery/JavaScript 实现不需要刷新页面的情况下切换页面的内容；通过查看文档发现浏览器支持一种叫做 XMLHttpRequest 的技术，可以让我们不需要通过跳转的方式从服务器获取到信息，从这里开始了解到 HTML、XML 和 JSON 三种不同格式的区别；第一次知道了可以通过服务器传递 JSON 格式的纯数据，然后前端通过 JavaScript 对数据进行解析，并且结合前端的模板引擎渲染成完整的 HTML。

从这里又可以学习到如何通过 URL 中的 path、query、hash 以及 POST 和 PUT 请求正文等信息向服务器传递信息，服务器通过这些信息动态地对各种数据进行处理并返回结果。

SPA (Single Page Application) 开发习惯初见雏形。

这样我就来到了“白银”阶段了。

接触 Node.js

当我正在愉快地设计着 WordPress 的自定义主题时，偶然间我在某前端网站上了解到了一个新技术 —— Node.js。与它的相遇改变了我以后的学习路径，影响至今。

2009 年 Ryan Dahl 发布了一个基于 Chrome JavaScript V8 引擎开发的程序运行环境 Node.js，它允许开发者在除了浏览器以外的地方运行 JavaScript 语言，并且提供一些标准库允许 JavaScript 脚本启动进行启动一个 HTTP 服务端应用这种以前在浏览器无法完成的事情。

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

这一份代码是 2010 年写在 Node.js 官网的一段实例代码，机缘巧合之下我被这么一段简单的代码深深地吸引住了，虽然当时安装它仍需要从 GitHub 上克隆整个项目代码到本地并依次运行以下指令：

```
$ ./configure
$ make
$ make install
```

这一次编译就得花上至少十分钟，但完成安装后运行上面的一段代码，并在浏览器中打开 <http://127.0.0.1:8124/>，然后在浏览器上看到 Hello World 字样时仿佛新世界的门打开了。因为当时我所接触过的服务端程序只有 PHP，而 PHP 本质上就是一个模板引擎，它并不能很直观地处理请求本身而是借助 CGI 进行响应。能做更多的事情，这件事情对刚学习编程不久的新手来说是具有很大诱惑力的。

从这里开始，Node.js 配合 npm 便开始了长达 10 年的快速发展。从纯服务端应用开发，到开发工具、工程工具，再到如今的 FaaS (Function as a Service, Serverless) 开发方式。Node.js 已经成为 Web 工程师不可或缺的一项技能，不管是用来开发服务端应用还是开发工具类应用，甚至是使用 Electron 开发桌面端应用还是配合 React Native 开发移动端 App，Node.js 能让前端工程师了解更多系统级别的概念，如网络、I/O、内存、文件系统等等，这些很多都是原本在浏览器端上看不到的。而学习这些知识对你理解前端开发背后的一些原理有非常好的价值，就跟学习算法一样。

结论：请学习 Node.js 和其中涉及到的一些基本计算机原理。

框架时代

当我在做 WordPress 主题的时候，绝大部分的主题开发者都会在前端做一些简单的效果，甚至有甚者会通过 JavaScript 实现一些原本只能通过后端来完成的事情，比如文章列表、文章内容的加载和渲染。而当年这些主题开发者基本上都会使用 jQuery 来进行这些 JavaScript 的操作，因为纯手写 JavaScript 在当时来说非常的繁琐（ES4 时代，很多现在被广泛使用的原生 API 都仍未具备）所以当时 jQuery 就是大家的首选方案。

从非常早的 PrototypeJS、后来的 jQuery、进入 MVC 时代的 Backbone，AngularJS 开启 MVVM 模式，React 引入 FP 的概念，Vue 成功开启了渐进式开发体验的道路。一路下来一地的鸡毛，被各路人马诟病前端领域一个月开发一个新框

架，“学不动了”。然而作为一个也写过框架、写过工具类库的开发者，我很喜欢用一个经常用于泛科技领域的例子来类比前端领域。

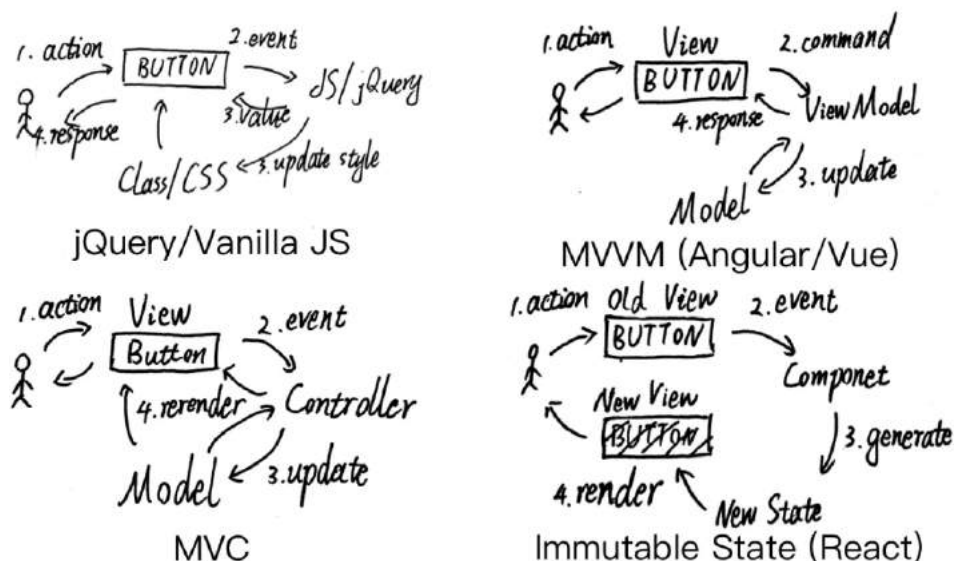
科技的终极目标，就是让人民感觉不到科技。

jQuery 时代，前端开发者使用 JavaScript 的模式是从页面中获取 DOM 元素，添加事件，然后通过 class 和 style 对页面进行动态地变更，以完成对用户行为的响应。

Backbone 时代，原本用在桌面端软件开发中的 MVC 模式被引入到了前端开发中，前端开发者们发现 Web 开发的复杂度已经需要用这些更成熟的开发模式进行管理了。

AngularJS 时代，从这里开始 Google 把数据双向绑定模式带到前端开发中，将原本需要通过 JavaScript 控制 DOM 元素这一繁琐的操作变成了只需要关心 Model 层需要改动什么内容即可。而 Vue 则将这种模式的开发成本降低到了一种相当可观的程度，让很多新手开发者也能很简单地入手这种便捷的开发模式。

React 时代，Facebook 的科学家们把函数式编程的思想引入到前端开发中，注重的是数据链路的可跟踪、可回溯、可管理，让整个数据链路是尽可能以单链路流转。



虽然前端领域常被说“一个月一个新框架”，但实际上每一个框架在迭代的过程中都是解决了它们所在业务场景的实际需求的，并不是“拍脑袋”地想要把每一个技术细节做出一个 break change。

而目前我目前推荐的学习的框架是 React 和 Vue：同样都是目前最流行的框架之一，而且可以预见未来 3~5 年内都是能满足找工作的需求的。

React：引入函数式编程 (Functional Programming) 的概念，使得写代码的思路更加严谨，更具有可维护性和逻辑可导性。

Vue：将 MVVM 模式变得非常简单易于入手，把 Progressive JavaScript Framework 的定位实践得非常到位。且如今 Vue 3.0 的 Composition API 更是在某种程度上将 Hooks 的玩法实现得比 React Hooks 更优。

结论：请不要害怕学习！不要惧怕新技术！

工程之路

虽然我在接触了框架和 Node.js 之后，发现 JavaScript 除了能实现一般只用于展示内容和呈现简单交互以外还能做更多的事情。但本质上还是围绕着多个页面进行页面上 DOM 元素的控制，而直到我打开了 Google 的一些网站时，我才发现原来网站除了能叫页面以外，还能称之为“应用”。

自从 Google 上线了一个完全不需要刷新页面就能完成所有事情而且体验很不错的 GMail 之后，我们发现网页原来也是可以承载那么复杂的逻辑和应用场景的。大家的热情异常地高涨，想着能不能让自己所负责的项目也有这么厉害高级的样子。但随着项目不断地复杂，代码规模也变得非常难以管理，而这个时候就需要工程化的引入。

工程化协作

对于企业来说除了研发效率要足够高以外，研发链路的安全、合规也是同样重要的。

什么叫安全合规？可管理的代码版本、可控制的发布流程、可管控的灰度机制，

都是大厂用于保证项目流程稳定进行的必要工具。

有很多初学者或者还没有大公司经验的同学在写项目时都是单打独斗的，但更多的一线项目都需要至少 2~3 个甚至更多的人员一同参与开发的。

而这种时候，因为每个人的水平和开发习惯都是不一致的，而这些不一致就直接导致整体研发效率和项目进度受到极大的影响。所以需要一种能够让大家在一个水平线上进行开发的模式，工程化需求便应运而生。

版本控制

- Git: GitHub、GitLab、Coding……
- SVN: BitBucket、Google Code……
- 代码样式检查工具 JavaScript/TypeScript: ESLint
- 测试工具
- 单元测试: Karma、Jest、Mocha……
- 持续集成: CI
- ……

工程化开发工具

从直接将 JavaScript 代码用 script 标签，到需要将 jQuery 文件和主要程序文件分别引入，再到 Node.js 出现后使用 npm 进行依赖库管理并使用 webpack 进行打包和压缩。工程类工具的发展见证着前端工程近十年的发展历史，对目前我们所常用的工程工具有更好的了解和实践，绝对是通往优秀路上不可或缺的一步。

- 依赖包管理工具: npm、yarn
- 打包工具: webpack、rollup
- 脚手架工具: create-react-app……

工程化开发语言

相信很多同学都听说过 JavaScript 诞生之初的一些轶事，比如根本没有特别多的严谨思考，或者在非常多的场景中十分地晦涩，比如隐性转换等。

有人认为 JavaScript 能发展到如今的地位跟它的这种“灵活度”或者“松散度”有关联，虽然在某种程度上确实因为这种特性造成的 JavaScript 学习门槛比较低而间接导致。但就如我上面所说，当项目规模和人员规模不断发展乃至膨胀过后，这些特性会逐渐表现出来非常糟糕的体验。

团队之间因为没有良好的技术文档沉淀，信息不对等的情况直接导致代码在没有良好的单元测试时出现逻辑冲突。

第三方依赖库的 API 在设计上大量使用了 JavaScript 松散的特性，导致使用方在引用时频繁出现“迷惑”的状态。

当需要使用 JavaScript 与其他语言（特别是强类型语言）进行交互时，JS 过于松散的习惯会让对接方感到非常迷惑，对于双方的实际接入成本会比前期预估的大得多。

为了解决这种情况，来自不同编程领域的大牛们都纷纷开始想办法，于是乎便诞生了非常多的“轮子”：

- Java 系: Scala.js、ClojureScript
- Go 系: GopherJS
- Microsoft: TypeScript
- Facebook: Flow、Reason

目前 TypeScript 已经影响了前端乃至整个 Web 领域的开发生态，TypeScript 之父 Anders Hejlsberg 创造过 Turbo Pascal、Delphi、C# 等在整个计算机科学领域都举足轻重的语言，而 TypeScript 又再次创造出翻天覆地的变化。

强类型的引入能让我们在写代码的时候从值优先的思维转变成类型优先。

强类型的引入能帮助开发工具（IDE 等）更好地为开发者提供便利性能力，如智能补全、类型检测、编译时检查等等。

TypeScript 可以让 JavaScript 更好地与其他语言进行交互，甚至转换为其他语言。

工程化通用组件

当需求不断变多后，“爱偷懒”的工程师们就会把经常用到的内容进行抽象，比如从很早以前就有的 ExtJS、Twitter 工程师发布的 Bootstrap 再到今天的 Ant Design、Element UI 等，都帮助我们更快更好更稳定地完成一些通用页面能力的开发。

- React: Ant Design、Fusion Design
- Vue: Element UI、iView、Ant Design of Vue

逻辑抽象能力

随着我对 JavaScript 应用的编写经验不断增加，我所尝试的技术和场景也在不断地变得更加复杂。而当逻辑代码变得越来越复杂时我也渐渐发现一个新的问题，很多时候我所编写的逻辑代码是相似的，但相似之余其中的一些细节不尽相同，而这些代码往往是后期维护成本最高的。

这就让我感到十分困惑，如何让我的代码写起来没有那么繁琐的同时，又不丢失原本代码的应有逻辑呢？这就让我想起了之前学习的框架，它们的实现原理不就是把原本我们写得非常繁琐的逻辑代码进行压缩，让我们写起来更加简洁直观吗？

这是我曾经面试过的一位校招候选人写的代码，其背景是用于快速判断自走棋类游戏中不同的增益能力（Buff）的成立状态。但显然这样的代码在实际开发中是绝对不允许存在的：代码逻辑过于冗余。

一旦通用判断逻辑出现变动，需要每一个都进行手动维护。

没有良好的可维护性。

所以我便提出如何让这些代码写得更加“优雅”和利于维护。

```
export default {
  beastBuff: (state) => {
    let arr = [];
    if (state.raceCount[0]['beast'] == 2 || state.raceCount[0]['beast'] == 3)
    {
      console.log(`you got 2 beast`)
      arr.push(state.racebuffdata[8])
    } else if (state.raceCount[0]['beast'] == 4 || state.raceCount[0]
['beast'] == 5) {
      console.log(`you got 4 beast`)
      arr.pop()
      arr.push(state.racebuffdata[9])
    } else if (state.raceCount[0]['beast'] == 6) {
      console.log(`you got 6 beast`)
      arr.pop()
      arr.push(state.racebuffdata[10])
    } else if (state.raceCount[0]['beast'] < 2 && arr.length == 1) {
      arr.pop()
    }
    return arr;
  },
  caveclanBuff: (state) => {
    let arr = [];
    if (state.raceCount[1]['caveclan'] == 2 || state.raceCount[1]['caveclan']
== 3) {
      console.log(`you got 2 caveclan`)
      arr.push(state.racebuffdata[11])
    } else if (state.raceCount[1]['caveclan'] == 4) {
      console.log(`you got 4 caveclan`)
      arr.pop()
      arr.push(state.racebuffdata[12])
    } else if (state.raceCount[1]['caveclan'] < 2 && arr.length == 1) {
      arr.pop()
    }
    return arr;
  },
  demonBuff: (state) => {
    let arr = [];
    if (state.raceCount[2]['demon'] == 1) {
      console.log(`you got 1 demon`)
      arr.push(state.racebuffdata[5])
    } else if (state.raceCount[2]['demon'] < 1 && arr.length == 1) {
      arr.pop()
    }
    return arr;
  }
}
// ...
}
```

我们不难发现这几个 xxxBuff 函数中的逻辑都非常接近，但也各有不同。那么如何将这段代码进行优化和抽象呢？我当时给 TA 提出了一份示例代码：

```
const beastConfig = [
  [2, [2, 3], 8],
  [4, [4, 5], 9],
  [6, [6], 10],
  [2]
]
```

这份代码中的每一个数字在上面的 beastBuff 函数中都可以一一找到，那么要怎么将它们复用到逻辑代码中，实现与原本的代码一样的功能呢？



我同样给他写了一份参考答案：

```
const generateBuff = (rate, configArr) => {
  return state => {
    const arr = []

    for (const [ output, conditions, index ] of configArr) {
      if (conditions && index) {
        // Buff calculating
        const isHit = conditions.some(condition => state.raceCount[0][race]
        == condition)
        if (isHit) {
          console.log(`you got ${output} ${race}`)
          arr.pop()
          arr.push(state.racebuffdata[index])
          break
        }
      } else if (state.raceCount[0][race] < output && arr.length === 1) {
        // Last condition
```

```
        arr.pop()
      }
    }

    return arr
  }
}

export default {
  beastBuff: generateBuff('beast', [
    [2, [2, 3], 8],
    [4, [4, 5], 9],
    [6, [6], 10],
    [2]
  ]),

  caveclanBuff: generateBuff('caveclan', [
    [2, [2, 3], 11],
    [4, [4], 12],
    [2]
  ]),

  // ...
}
```

原本代码里面通过 hard code 实现的判断逻辑，通过观察其中的共同点，并思考能否通转换为可抽象部分，这同样也是一名优秀的工程师所必须具备的能力。Be D.R.Y.! (Don't repeat yourself)。

更高层次的思考能力

随着我对不同业务、不同场景和不同代码难度的不断探索和研究，我发现在前端领域乃至整个编程领域里，不同的框架和架构层出不穷地发展，其实在根本上就是各种实际业务场景在寻找更合适的 Better Practice (更好实践)。就如前面的所说的那样，不同的框架作者在开发的时候会采取不同的代码结构甚至代码哲学，这些不同的思维角度可能在框架的源码中并不会直接表现出来。但我不会说研读源码完全没有用！因为研读源码最起码可以学习其中的一些 trick 或者代码习惯。

但更重要的是理解从 API、系统架构上进行思考，因为只有多思考了，你才能逐

渐变得比其他人更加对不同的技术游刃有余。

EOF

这一个流程并不是严谨的学习路线，更多的是我个人的一些经验总结。当然除了我所提到的学习知识点以外，还有很多不同的分支对应着不同的实际业务和场景，比如配合 Electron 开发桌面端应用、配合 React Native/Flutter 开发移动端应用、配合 Node.js/QuickJS/FibJS 开发嵌入式应用、配合 TensorFlow.js 开发适用于前端甚至适用于边缘计算的机器学习应用、配合 WebAssembly 将 Web 应用的使用体验提升到接近原生应用的境界……

关于 JavaScript 有一个很有名的预言：

凡是能用 JavaScript 重写的，终将会使用 JavaScript 重写。

无论这句话会不会最终完全实现，但目前我们已经能看到很多应用逐渐通过 Web 应用的形式云端化，比如 Photoshop、音视频编辑软件、代码编辑器甚至是大型游戏等等原本我们完全没想到可以运行在浏览器中。

前端开发困难吗？不困难、门槛相对较低。简单吗？不简单，通过相信看到这里的你也已经有所体会了。当然实际要如何选择路线和方向，还是你自己所遇到的经历和机遇来决定的。

附录

淘系前端团队

618 虽然已经结束，但更大规模的双 11 全球狂欢节马上就要启动。高复杂度、大规模的营销活动业务场景持续推动着淘系前端技术体系朝着极致方向迭代演进，期待更多的同学加入阿里淘系前端团队，一起来创造 618、双 11 的新历史，此时此刻，非你莫属！

简历投递：moming.lq@alibaba-inc.com



钉钉扫码加入
「淘系互动交流群」
与万千技术爱好者共同进步



阿里云开发者“藏经阁”
海量免费电子书下载



微信扫码关注公众号
「淘系前端团队」
获取前端领域最新动态