

阿里文娱 技术

阿里云开发者社区  
ALIBABA CLOUD DEVELOPER COMMUNITY

# 优酷OTT互联网大屏 前端技术实践

阿里巴巴文娱前沿技术精解



阿里云开发者电子书系列

随着客厅娱乐的不断升级，OTT几乎成为除手机外的第二块屏幕。为了提升基于遥控器的交互体验，优酷前端技术团队通过Web技术自研了焦点引擎，结合Weex体系打破了原生系统长版本周期的限制。这本合集将分享前端工程师在这个特殊平台上遇到的技术挑战及技术实践。

—— 阿里巴巴文娱技术 倪欧



【阿里巴巴文娱技术】  
公众号



阿里巴巴文娱技术  
钉钉交流群



阿里云开发者“藏经阁”  
海量电子书免费下载

## 目录

概述	4
优酷OTT大屏的前端建设之路	5
业务支撑	14
OTT端登录态设备穿透：扫码登录与扫码反登录	15
OTT端性能优化建设之Weex实践之路	22
能力建设	28
OTT端技术赋能之前端收单能力建设	29
我在OTT做自动化制图尝试	37
创新落地	53
不一样的烟火：记OTT端半屏互动能力建设	54
OTT端性能优化建设之本地缓存设计	63



# 优酷OTT大屏的前端建设之路

作者| 阿里巴巴文娱技术 默吉

转眼间2020年已过半，我投身到OTT端开发已经5年有余，回首OTT端（酷喵APP）前端建设历程，感慨良多。经历了从0到1的艰难起步，也经历了从1到N的合力前行；有过通宵达旦的debug之夜，也有过阳光和顺的高光时刻。以下和大家分享优酷OTT端前端的技术历程。

## 一、背景

### OTT是什么？

简单来说，“OTT”名词解释为over the top，原为篮球运动名词，“过顶传球”之意，后引申为通过互联网向用户提供各种应用服务，这种服务主要由运营商之外的第三方提供。

### OTT端上都有什么？

OTT端开发规范和逻辑与手机端APP属于一脉，一般由Android系统做底层支持，由APP开发人员开发不同类型软件应用。本文中的OTT端建设，为优酷“OTT端酷喵APP”建设。

### 前端在OTT端扮演什么角色？

说起来，前端开发与APP Native开发为互补关系，OTT端酷喵APP主框架为Native开发实现，而活动坑位投放、部分APP功能页、快速发版补位项目则需要前端介入开发，尤其以“活动坑位投放”最为常见，因其内容形式多样化，发版频率高等特点，并不适合使用Native开发，二者恰巧是前端优势所在。

### 这几年我们都做了什么？

自2015年的“OTT影视大厅”APP，到后来的“华数牌照方影视”APP，再到现在的“酷喵APP”，历经多轮端APP衍生发展，多次迭代升级优化前端相关能力。

## 那么OTT端与其它端有很大不同吗？

是的，纵观M端、PC端以及Pad端，用户可以通过鼠标、键盘、触控方式进行页面交互，但在OTT端所有基本操作均依赖“遥控器”进行操作，这也就决定了其人机交互与其它端存在本质差异，“焦点引擎”概念应运而生。

## 二、焦点引擎

我们先来看看TV用户交互的现状，我们只有一个遥控器，上下左右和ok键能分别映射到键盘的四个方向键和enter键；我们可以监听document的key事件来做焦点的切换逻辑。那么问题来了，作为一个业务开发者，难道要为每个页面花精力去处理“按了一下右键，下一步应该选中哪一块区域”诸如此类的问题？难道就不能像鼠标操作那样指哪打哪？

其实，所谓的“指哪打哪”是认知学上的问题。而无论在什么设备下，用户和开发者对人机交互都有符合心理预期的诉求。只是PC和手机上页面的人机交互都已经做得十分到位，而TV则不然。所以，我们希望通过一些手段，使TV上的页面交互更顺畅自然，尤其对于业务开发者，你的代码逻辑、风格，还有代码量能够趋近于PC和手机端的H5页面就是我们最大的愿景。

作为专注于TV前端开发的团队，我们设计了一套组件树管理和焦点管理的机制，来对TV上的交互行为进行抽象。使它托管页面中焦点的切换，帮助开发者从繁琐的焦点处理中释放出来，开发者只需要专注于业务代码的书写，提高开发效率。

接下来，我们从组件树、焦点管理两个方面对“焦点引擎”进行简单介绍：

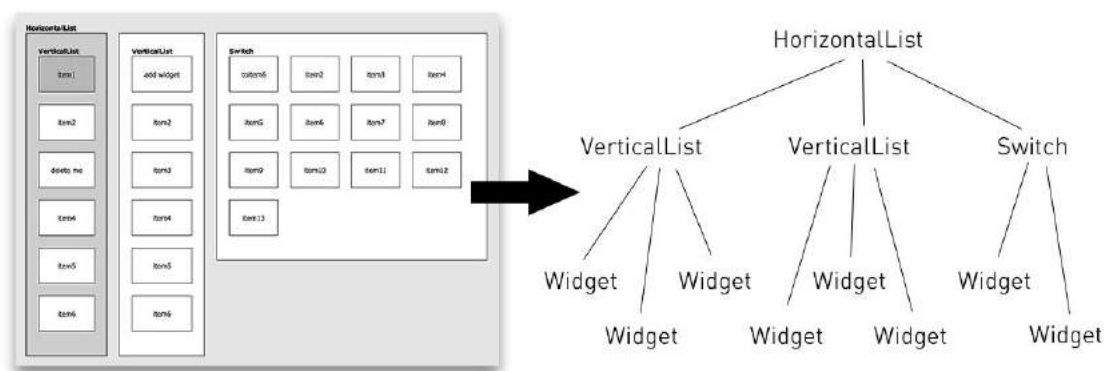
### 组件树

通过对TV端页面的切分，我们大致确定了使用频次比较高的一些模块：水平/垂直列表、多方向无序排列模块、滚动列表等。

我们希望把这些模块抽象成组件，组件内部可以自动实现各个元素间的焦点切换。同时，一个页面中的各个组件间的切换也应该对开发者透明。为此，我们把这些组件以树形结构组织，组件的层次关系有助于生命周期管理和事件流传递，为焦点管理的实现提供良好基础。

Widget是组件的基础类，拥有指向其父节点的属性parentWidget，同时实现增删改查方法管理子节点数组childWidgets。上述模块都是基于基础类的扩展。具体而言，基础类

并不直接实现其子节点间的焦点切换功能，而是交由扩展类内部实现，如水平列表实现左右按键的焦点切换，多方向无序排列模块实现了多种焦点感知算法，滚动列表实现了TV端焦点切换时的页面滚动逻辑。下面的例子展现了一个页面的模块划分到组件树的映射过程：



需要强调的是，虽然html天然就是树形结构，但由于TV中焦点管理相关的逻辑和事件并没有原生实现，所以我们仍然需要把dom树映射成组件树以方便功能注入。

## 焦点管理

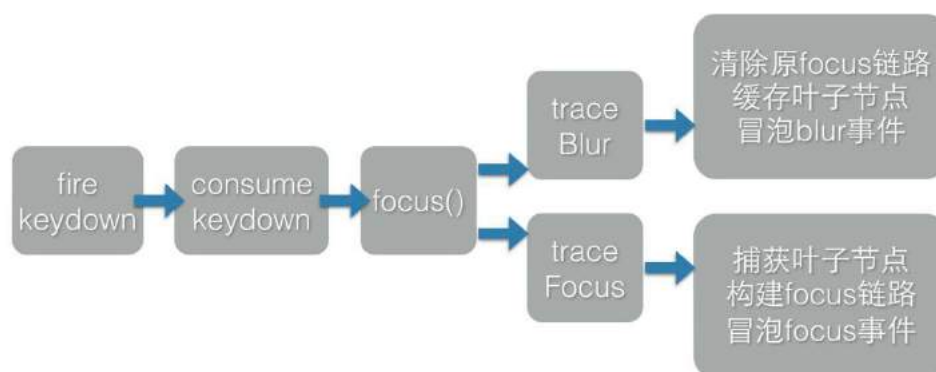
焦点管理主要有两个技术点，其一是事件模拟和组件树中的事件流管理，其二是焦点链路的构建。

尽管js里支持keydown、focus、blur这样的事件，但这些事件只能够在可选中的元素（focusable elements）或document上触发。TV上对于可选中的定义则要复杂得多，几乎所有的视觉元素都是可选中的，默认的切换行为往往无所适从。

我们对组件模块增加on、detach、fire这样的事件管理方法，事件的触发允许在组件树中冒泡。另外实现了Event.Base事件类，可以方便地扩展出事件流中传递的事件，如Event.KeyEvent、Event.FocusEvent、Event.SelectEvent等。

有了事件的模拟，焦点情况已然明朗开来：如果是组件切换逻辑的触发，整个流程应该从最左端开始；如果通过调用组件的focus()方法进行聚焦，则从focus这一步开始。焦点切换的执行流程大致如下图所示，





以上就是OTT端的特色点“焦点引擎”介绍，这属于技术层面的基础架构支撑。回归到业务中，正如上文所讲，“活动坑位投放”开发为前端开发业务重头戏，我们以“可视化搭建”为方向，提高开发效率、提升页面质量。并在其它业务并行情况下，进行能力升级，积极拓展前端赋能。

前端能力覆盖也由最早期活动投放页面开发，延伸到现阶段的OTT端页面搭建、半屏互动、APP功能单元、质量监控等多个领域。而在前端加载体验方面，也取得质的提升，尤其是秒开率、crash率等指标。在这个漫长的建设过程中，我们大体经历了三个阶段“石器时代”、“农耕时代”、“工业时代”。

### 三、时代建设

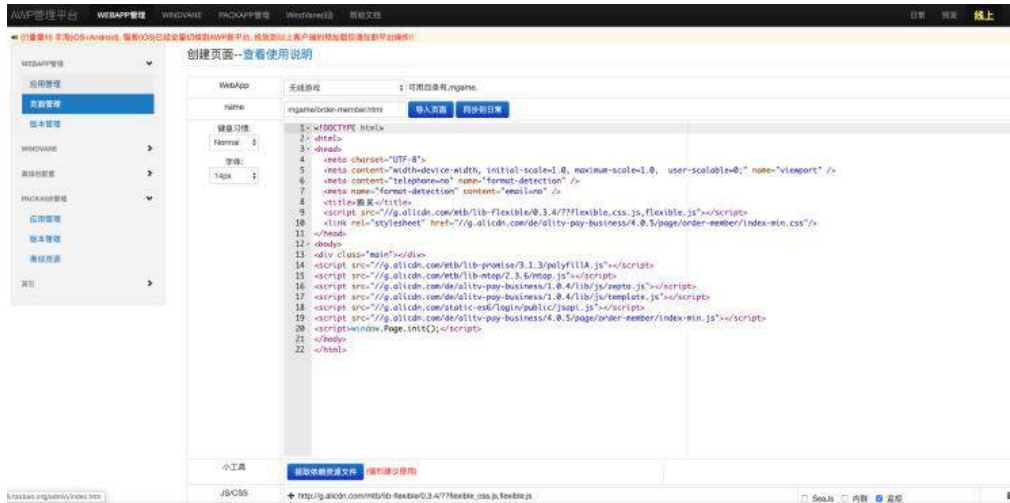
#### 石器时代---源码开发时期（AWP）

最开始主要以源码开发为主，每个活动都需要开发同学从零开始搭建，效率低，当活动积累了一定阶段后，我们抽离出通用组件，开发效率上稍有好转，比如3天的开发变成了1.5天。但本质上没有解决问题，尤其是当运营的活动页面需要频繁调整时，前端同学被折磨的死去活来。运营一点内容的修改，比如更换图片的位置大小等，前端都会发布新的版本，硬编码的这种方式确实是硬伤。尤其到了双11，前端集体上阵，被活动页面折磨的好生痛苦……这个时期持续了半年多，大家寻找着新的方式。

问题：源码开发比较方便，符合前端开发习惯，通过常用组件的沉淀效率有所提升，但无法解决活动页面频繁调整的问题。



## 【AWP源码开发】

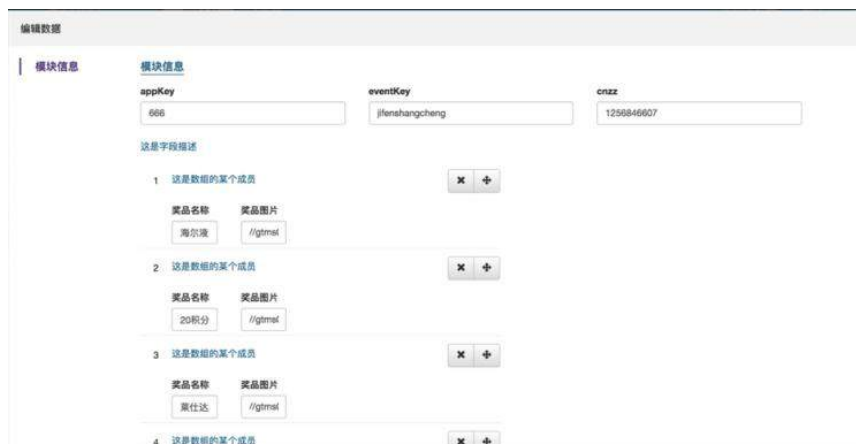


## 农耕时代---模版开发时期（TMS/Zbra）

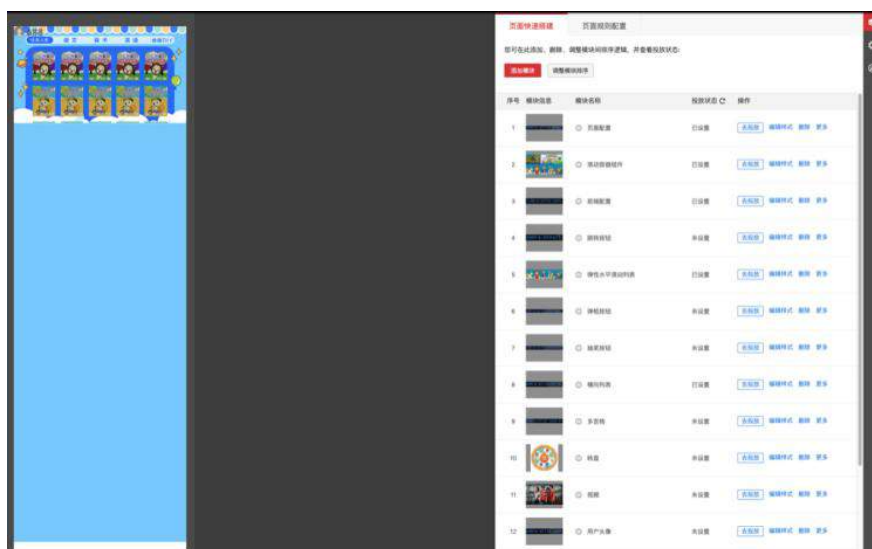
这个时期主要是模版化时期，通过坑位的配置解决开发效率的问题，技术同学与UED沟通制定出通用的OTT端通用型模版，例如每周大片专题模版、棋牌抽奖活动、倒计时模版等等，基本解决了运营效率的问题。

问题：这种方式依然有局限性，需要用约定的形式规范化内容，限制运营同学的创造力。这一阶段持续了半年之久，可以满足运营的基础需求，但面对不断推陈出新的活动形式，模版方式显得越来越力不从心.....

## 【TMS活动模版】



## 【Zebra斑马模版】



## 工业时代---可视化搭建时期（ARK搭建平台）

最终，我们参考了阿里集团的可视化搭建系统，2017年尝试打造一套适合TV端的搭建方案。如果从零开始，肯定是一条漫长之路，涉及很大的开发量。由于之前一直在使用阿里集团内B搭建系统，我们发现这些能力B系统都基本具备，那么我们为什么不借力，站在巨人的肩膀上？于是我们将页面的托管能力收敛到B系统的底层服务上。接下来，我们开始了研发阶段，当时投入的同学还历历在目，服务端开发震动负责对接B系统的所有node服务，我主要负责前端部分，经过2个多月的努力，我们发布了第一个beta版本「2016-11-04」。

## 【ARK可视化搭建】



## 超越时代---前端多元化赋能

在完成标准化、高效化建设后，满足日常运营需求已不在话下，而我们则将方向转移至技术多元化赋能上，逐步完成多项创新建设，推动业务建设。

### 半屏互动

依托现行weex技术栈灵活的发版及快读覆盖能力，通过前端、客户端、播放器等多方共同努力，借助OTT酷喵APP自身硬件特性及用户使用习惯，开创性的完成“半屏互动”建设。该能力建设既满足了用户线运营“大量级曝光”的诉求，也满足会员线运营“精准投放、提高渗透”的诉求。根据数据结果来看非常成功，尤其在曝光UV量级上，实现端内投放曝光UV提升约200%的惊人数据。



### 同步登录态

OTT从开发语言、操作理念等都遵循手机APP相关规范，不同之处在于手机APP为触屏操作，OTT一般为遥控器操作；OTT端的最常见登录，与常见的PC端扫码登录一致，由OTT端展示登录二维码，手机端扫码操作确认登录，即“扫码登录”；我们希望将OTT端登录态同步至手机端，由手机端完成相应操作，同时避免手机端无登录态、登录账号不统一等带来业务操作上的偏差，即“扫码反登”。



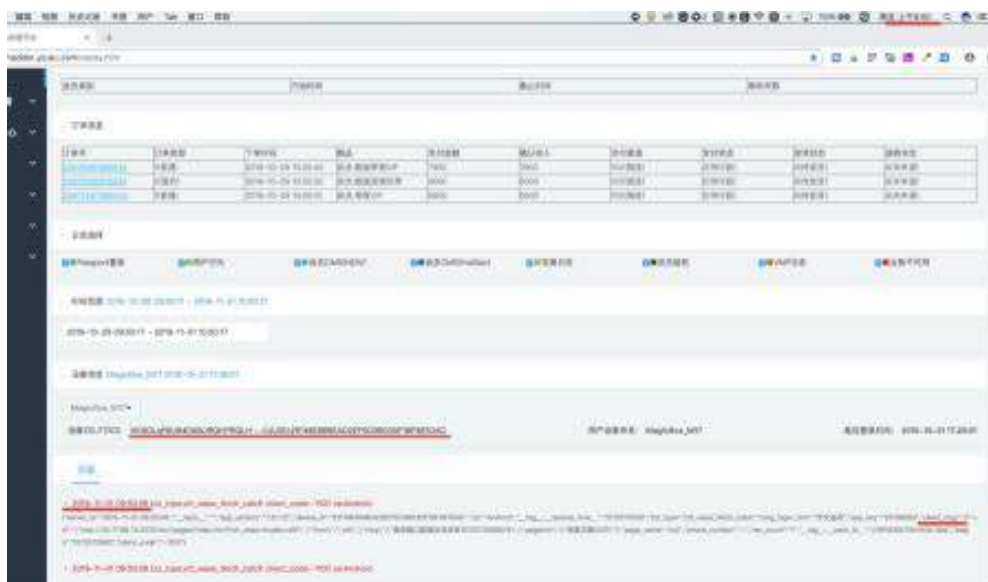
OTT端



手机端

## 质量监控

长期以来，TV端投放页面，存在用户投诉-bug定位困难的情况，客服及技术线同学难以获取用户相应错误数据；使用客户端隐藏debug又存在操作繁琐，沟通困难情况。经过调研后，APP端质量平台能力与OTT端该诉求吻合，因此集合Rax开发方、质量平台、Rax容器方三方共建，完善OTT端质量平台，即实时日志监控。



## 前端可视化搭建收单

OTT酷喵会员线-“收单能力”升级已经取得一定成果，借助同步登录态、单商品收银台等前端创新能力的落地，直接推动了OTT端登录率近10%提升，并简化购买链路，实现可视化搭建，“一步扫码付费”，该组合能力领先业内。另外，OTT半屏收银台的开发上线，使OTT酷喵用户触达能力迎来重大升级，依托播放页，实现一步扫码付费，较大程度简化付费链路，提高了转化率。



## 积分商城

OTT积分商城是围绕积分的发放与兑换，建立符合大屏端运营特点的用户管理体系，用户通过多种行为获取积分，再通过不同玩法或商品兑换消耗积分，从而为用户增长、用户活跃与留存提供有效抓手。

积分商城二期功能已全部完成，支持积分的有序发放，酷喵自身权益、（阿里生态）二三方虚拟权益与电视淘宝权益的兑换，为OTT用户运营的基础能力打下了基础。之后，与电视淘宝打通，完成账号绑定、加购、下单、付费等能力建设，后续将实现双方资源互补。

以上就是优酷OTT大屏的前端建设之路，希望能够为大家带来启发。



# OTT端登录态设备穿透：扫码登录与扫码反登录

作者| 阿里巴巴文娱技术 魏家鲁

## 一、背景

手机设备相较于台式电脑、笔记本电脑，具备更强的“私密性”，因此在智能手机中安装APP应用后，登录态可长期保存，这也直接导致了用户对于应用/站点密码记忆度大幅下降，“扫码登录”技术随之被普及。

常规“扫码登录”业务，一般由PC端和手机端两部分组成，而且PC端一般作为“被扫码方”，登录态则由手机端进行操作及确认。那以“CIBN酷喵影视”为代表的OTT大屏电视场景，其扫码登录实现情况又是怎样的呢？

### 1、OTT大屏特点

OTT从开发语言、操作理念等都遵循手机APP相关规范，不同之处在于手机APP为触屏操作，OTT一般为遥控器操作；

OTT端的最常见登录，与常见的PC端扫码登录一致，由OTT端展示登录二维码，手机端扫码操作确认登录，即“扫码登录”；

我们希望将OTT端登录态同步至手机端，由手机端完成相应操作，同时避免手机端无登录态、登录账号不统一等带来业务操作上的偏差，即“扫码反登”。

### 2、扫码登录和扫码反登的示例：

常规扫码登录：大屏端未登录、小屏端操作登录(如下图)





OTT端



手机端

OTT特殊场景：大屏端已登录、登录态同步小屏端(如下图)



OTT端



手机端

既然以上两种扫码路数相反，当然实现理念也有较大差别，那么我们就“扫码登录”和“扫码反登”两种情况，对其技术实现进行剖析解读。

## 二、OTT端登录需求

OTT从开发语言、操作理念等都遵循手机APP相关规范，区别在于APP为触屏操作，OTT为遥控器操作。自然地，很多年轻人更习惯在手机上完成操作，实现优酷手机APP与OTT APP（CIBN酷喵影视）的无缝衔接。这其中就涉及到了账号登录态同步的问题，而且二者存在众多登录场景的互动。

所以，除了常规的“扫码登录”（通过手机操作，将手机端登录态同步至PC/OTT端），在OTT场景下，还衍生出了“扫码反登”。

扫码反登：指将OTT端账号登录态，同步至手机端，使手机端在扫码相关操作完成后，跳转至指定页面。优势是，由手机端完成OTT相应操作，同时避免手机端无登录态、登录账号不统一等带来业务操作上的偏差。

具体的“扫码反登”业务场景可分为四类：

登录：OTT端酷喵应用登录二维码；

客服：各种业务类型客服小蜜二维码；

互动：OTT端活动投放，与小屏配合互动-扫码；

支付：OTT端二维码收银台。

### 三、同步登录态

在OTT端主要有“扫码登录”及“扫码反登录”两类场景业务需求，那么我们先将两种同步登录态实现逻辑进行解读。

#### 1、扫码正登录

这一快捷登录方式，早在数年前已经普及，目前不光在PC端，在OTT端也是首发登录方式。虽然这是个程序技术，但是也有不少同学感到好奇：这里只有一个光秃秃的二维码，它是如何完成识别被那个手机扫码，之后又是通过什么逻辑实现OTT端/PC端本身登录的？下面我们进行逻辑拆解。

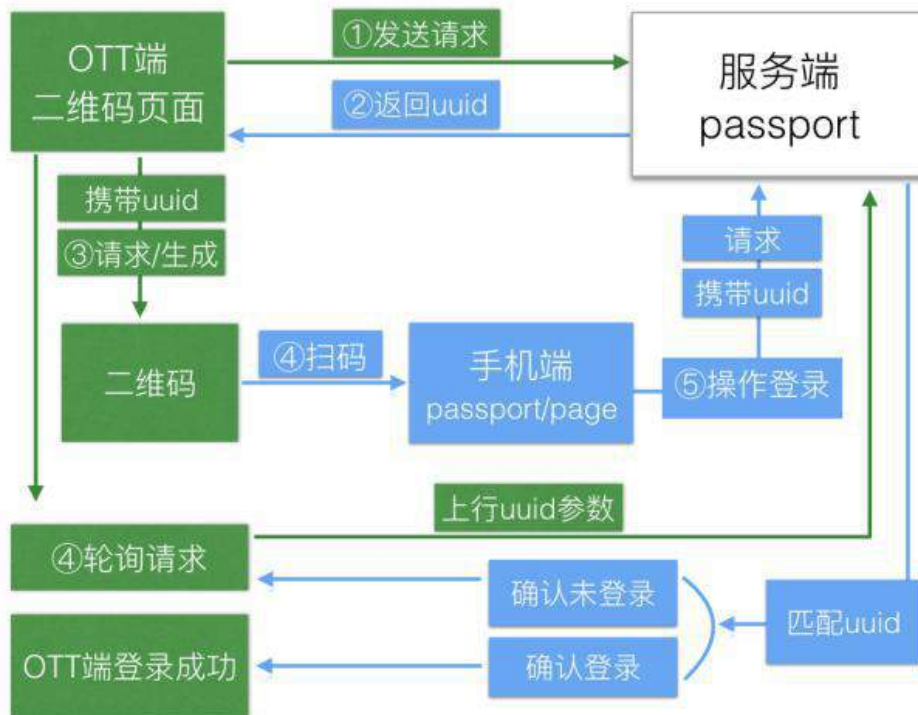
#### Tips

因为需要将小屏登录同步至OTT端，那么该情景OTT端与手机端登录情况分别为：

a) OTT端-未登录

b) 手机端-未登录/已登录均可

原理图



### 逻辑解读

①当二维码扫码登录页面被打开时，OTT端会向服务端发送一个请求，用以获取平台唯一标识值（当然也可以有客户端根据算法自行得出后通知服务端，但严密性存在问题），我们姑且称之为“uuid”，该值作为本次登录的核心参数贯穿整个登录始终；

②服务端在生成这个uuid之后，会将其记录（redis服务器），并建立查找索引逻辑，除了uuid通过接口返回前端外，接口返回中“登录验证二维码”url一并返回，此时uuid布设与该url参数中；

③前端页操作页面视图显示该二维码，该二维码实际为passport登录验证页面；

④在展示二维码的同时，前端将以uuid为参数进行登录状态接口验证轮询；

⑤现在登录状态轮询已开启，而二维码则开始等待手机扫码；

⑥当用户手机端扫码后，会出现两种情况：用户已经登录、用户未登录，那么此时就面临两种逻辑：

a) 用户已登录：则uuid及用户cookie随校验页面直接到达passport服务端，服务端根据cookie校验登录状态，并根据本次上行uuid去匹配先前（第①项）已记录的uuid，一经查找成功，则通过内部方法将cookie解析生成的token，与先前uuid形成键值关系；

b) 用户未登录：则手机端扫码后，一经校验无登录态，则该“passport登录验证页面”跳转至登录页面，此时，登录页url参数中，依旧携带有uuid参数，以及扫码登录标识，用于passport服务端后续逻辑处理，用户在键入账号密码后点击登录，则登录页跳转回“passport登录验证页面”，后续流程与前段落一致；

⑦在以上⑤中已经提到，OTT端一直在轮询接口登录状态，此时，在完成上文第⑥后，轮询接口再次携带uuid请求passport服务端，此时，经由uuid查询匹配，可以确定本次状态为“手机端已操作确认登录”，那么该次轮询接口返回中已经明确登录状态，并且将第⑥项中生成的token一并返回。

至此七大过程形成闭环，宣告登录成功，并显示相应已登录视图，扫码正登录也已经完成。

## 2、扫码反登录

OTT端存在一种特有场景：OTT端本身存在交互操作及功能上的局限，例如付费购买，需要将大屏登录态同步至手机端，在手机端完成强登录操作。

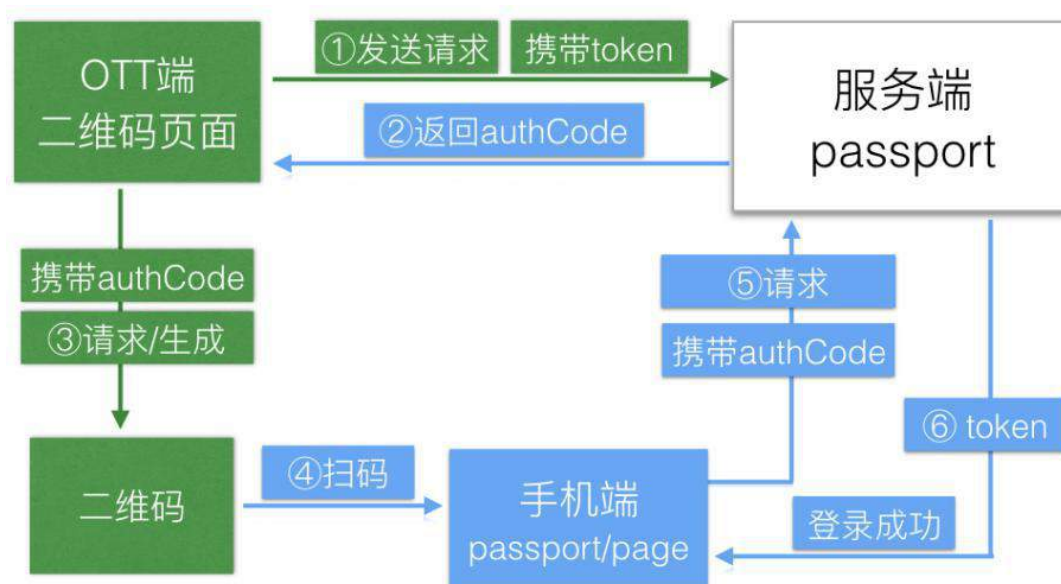
### Tips

因为需要将OTT屏登录同步至手机端，那么该情景OTT端与手机端登录情况分别为：

a) OTT端-已登录

b) 手机端-未登录/已登录均可

## 原理图



## 逻辑解读

- ①当二维码扫码登录页面被打开时，OTT端会向服务端携带token发送一个请求，用以获取平台唯一标识值“authCode”，该值作为本次反登录的核心参数贯穿整个登录始终；
- ②服务端生成authCode之后，会将authCode与请求上行token记录（redis服务器），并建立查找索引逻辑。除了authCode通过接口返回前端外，接口返回中“登录验证二维码”url一并返回，此时authCode布设与该url参数中；
- ③前端页操作页面视图显示该二维码，该二维码实际为“passport登录态种植页面”；
- ④当用户手机端扫码后，“passport登录态种植页面”被打开，此时该页面url中authCode参数，随请求发送与passport服务端；
- ⑤服务端接收到请求并且获取到authCode，并与先前存储的authCode进行匹配，已经匹配成功，则将先前存储的token返回，此时手机端登录态已经种植成功。

## 3、指定跳转

扫码正登与反登是OTT端登录业务中的核心环节，但这还没有完整满足OTT场景实际需求，我们还需要在手机端登录操作完成之后，跳转到指定页面。这相对于登录态同步要简单得多，我们可以在passport页面后追加一个callback参数，参数值为指定页面url，借助

passport相关逻辑，在手机端确认登录成功后跳转至该url，从而实现指定页面跳转。

在上述逻辑实现之后，则完整满足了OTT场景需求，而借助同步登录态以及指定跳转这两种能力，则可以进行进一步能力扩展、封装。

## 四、能力沉淀及扩展场景

优酷“CIBN酷喵影视”的业务量级非常大且场景相对复杂，在APP不断的业务迭代及能力升级过程中，扫码登录的能力也在积累沉淀。目前，“酷喵APP”扫码登录相关能力已实现以下能力：

- 1、能力组件化封装：完成“同步登录态-二维码”能力封装，实现相关业务零开发，可视化平台搭建；
- 2、二维码收银台封装：完成“CIBN酷喵影视”活动收银台能力封装，扫码直达付款，零开发，可视化搭建；
- 3、客服小蜜搭建化：借助1，使反馈不再匿名无序，实现问题排查有“账号”可寻；
- 4、半屏收银台封装：全屏播放中查半屏，实现超大流量精准引导付费。



# OTT端性能优化建设之Weex实践之路

作者| 阿里巴巴文娱技术 默吉

追求极致的用户体验是个永恒的话题。无论在PC端、移动端，还是IOT端，大家都在尝试着各种技术方案，如提高秒开率，降低白屏时间等等。

在OTT端进行营销活动开发的我们，也面临这一挑战，尽管PC端和Mobile端都有成熟的技术方案，但是到了大屏端，由于终端的差异性，很多技术方案不能完全照搬照抄。

回顾优酷在OTT端的用户体验探索之路，经历了三个阶段：webview时期、自定义内核Blitz时期、weex阶段。

当前weex(RAX)是OTT端的主要技术方案，它贯穿着整个前端页面的开发，无论是活动页面、半屏互动，还是常驻页面的开发，我们都采用这种技术方案，它带来了类似客户端的用户体验，相对H5的页面流畅度，体验效果上有明显的提高。

## 一、发展历程

### 1、webview时期：

活动页面采用原生的webview进行渲染，性能体验上勉强够用，但由于OTT设备多为弱硬件环境，而且在视频播放这块无法得到定制扩展，所以原生的webview有一定受限。





## 2、自定义内核时期：

Blitz web引擎是一套类似webview的渲染引擎。引擎前期投入人力很大，后期由于业务的调整引擎处于停滞状态，很多性能需求无法实现，如本地缓存，zcache介入等。在这个过程中我们尝试了很多提升用户体验的方案，如根据页面load加载时机，结束后再做整体展示，减少loading时间，设置1%分辨率的背景作为模糊图；其它优化方案如：异步加载js、懒加载图片、预加载等等，前端能做的方式都尝试了一遍，但结果都收效甚微。

性能优化方案：【loading加载时间，预加载，静态文件缓存】



## 3、weex时期：

weex技术在阿里集团的广泛推广，为OTT端带来曙光。客户端同学尝试将weex能力输出到OTT端，最终页面的性能体验发生质的变化。

由于OTT端的交互方式与传统端存在差异，所以在weex的引入过程中，客户端同学做了很多差异化处理。如jscore的剥离（OTT端只有andriod），焦点引擎的引入（OTT端靠遥控器操作），自定义组件等等。

性能优化方案：【首屏渲染效果，多屏滚动展示，视频播放，图片渲染】



## 二、weex实施方案

### 1、实施步骤:

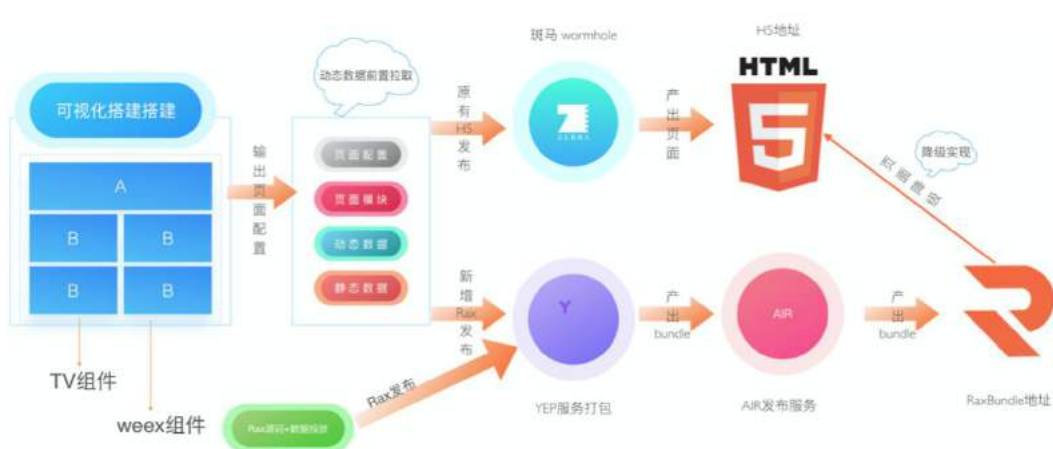
- 增加可视化搭建weex输出能力（原有ARK平台只支持H5页面的输出，平台侧增加weex的打包能力），将原有40+个H5组件进行weex化，这部分可以解决80%的业务场景问题。
- 源码weex开发，解决20%的定制化场景，如双11等大型活动及战役，这部分针对定制化源码开发，将H5源码开发的方式切换到weex的技术栈上。

### 2、技术细节:

- OTT端weex的差异性，一图胜千言。



- 可视化搭建技术改造方案



### 3、相关数据：

- 数据指标：【页面加载提升】【页面渲染流畅】【用户体验改善】

#### ■ 性能对比-页面加载时间（首次）

页面类型	M2	M13	M16S
H5	4462ms	9107ms	4143ms
Weex	1103ms	1797ms	698ms

#### ■ 渲染成功率-流失率

页面类型	渲染成功率	流失率
H5	98%	15%
Weex	99.5%	5%

#### 4、效果展示:

- 全屏weex



- 半屏互动

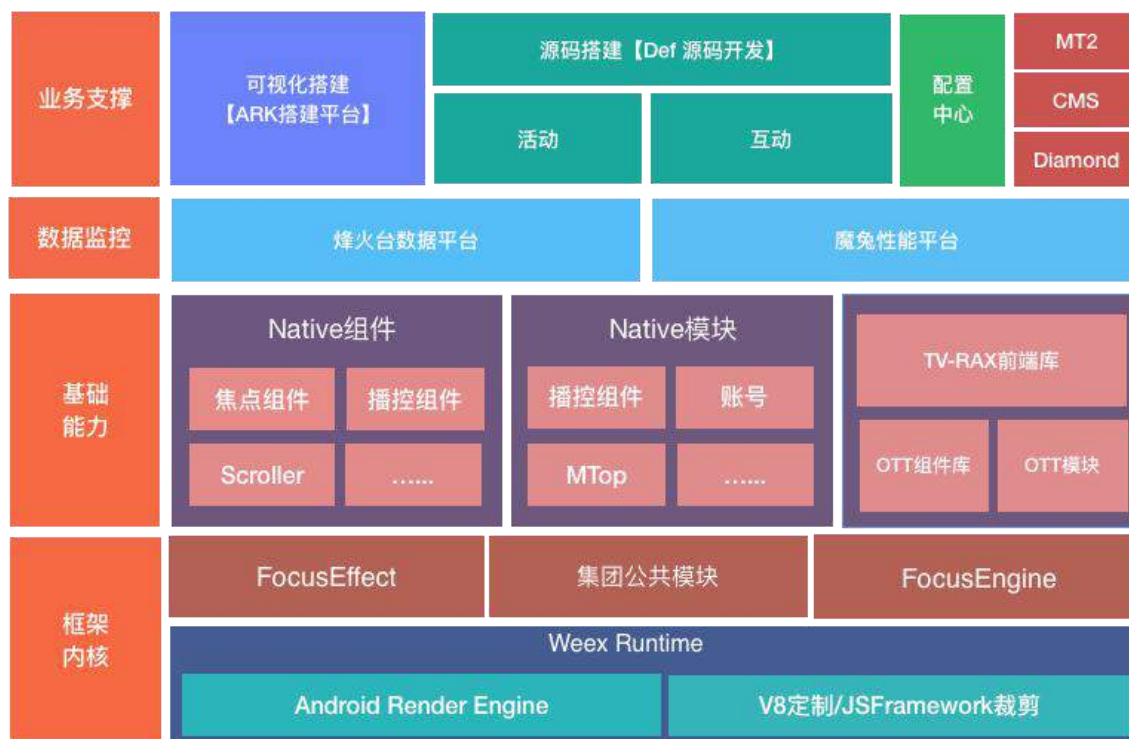


- 页面APP化



### 三、共建

输出OTT端的Rax能力到阿里集团，提供详尽的开发文档。



经过技术栈升级与优化，weex成为阿里文娱OTT端-前端业务的主要开发技术栈。相较于HTML5技术栈，weex稳定性及性能均有了长足进步。以页面加载效率为例，使用weex后，页面平均加载时间缩短了50%以上，效果显著。





# OTT端技术赋能之前端收单能力建设

作者| 阿里巴巴文娱技术 魏家鲁

## 一、背景

阿里巴巴文娱板块的OTT端业务，是由天猫魔盒硬件、酷喵APP、TV厂商专供APP等组成的大屏影视业务结合。业务平台为会员制，涉及付费会员、付费点播等多种业务类型。

相对于其它端，OTT端在体验上存在一定的局限性，一是受限于遥控器操作尺度较小；二是OTT端页面元素需要依赖焦点切换，深层级展示效果受限。于是在OTT端业内，各厂商APP页面均是走“扁平路线”：没有下拉展开、没有悬浮露出、没有多级菜单……在OTT页面建设开发过程中，用户体验与操作复杂度直接影响用户参与率。

会员营收业务作为OTT端（酷喵APP）核心目标之一，前端开发也紧紧围绕体验升级和收单能力升级两大方向进行技术探索，接下来我们就近一年“OTT端-酷喵APP收单能力升级建设”进行总结介绍。

## 二、原能力

**关键词：**Native收银台、综合收银台

**描述：**

以往OTT端-酷喵APP收单业务由Native开发完成，OTT端只有Native收银台一项收单能力，因原生Native开发，整体加载效率较好，并在持续迭代过程中得到升级优化。但是收银台作为Native独立页面，所有下单均需要由原页面跳转至该收银台页面，并由手机端扫码后操作下单，用户操作起来非常繁琐。





收单链路：OTT投放页面-->跳转Native收银台-->手机扫码-->自动下单-->支付

### 思考

就OTT端影视类APP而言，各大厂商均使用该逻辑链路，而使用中，也似乎没有什么不妥或者硬伤。但是，集合以往财年数据，我们发现OTT端在付费转化率方面，与其它端相比，水位较低。于是OTT产研团队，一直在思考，如何改变这一困局？经过多维度调研和测算，明确一个路线，即：简化付费链路。

与客户端相比，前端具备发版灵活、快速覆盖两大特性，也正是基于这一点，我们业务团队内达成方案统一，由前端进行能力创新升级，客户端予以必要能力支持。于是，收单能力升级正式立项下面对这三个阶段性成果，进行分析讲解。

## 三、能力1.0

**关键词：**同步登录态、前端简易收银台、可视化搭建

### 描述：

在《OTT端登录态设备穿透：扫码登录与扫码反登录》一文中提到，在OTT端，我们实现了账号登录态的同步（相关原理图：传送门），为大小屏互动提供了先决条件。我们前期将该能力使用在大小屏各大战役主会场连通承接互动上，诸如“快快扫码，手机端抽大奖”、“扫码查看奖品”等等。

之后，面临越来越多的会员产品优惠促销活动，我们转换玩法，在OTT端投放的酷喵会员售卖相关页面，放置同步登录态二维码，手机端扫码后，实现账号登录态同步，之后指定跳转至手机端H5收银台，由用户在H5收银台操作下单。而此时OTT端页面一直存在，并未跳转，围绕会员产品售卖的宣传文案和引导一直呈现在用户面前，势必可以加强下单刺激，提高付费量。

于是，一条新的付费收单链路产生：

- 1) 借助会员商品后台，通过配置优惠活动和指定会员产品；
- 2) 将1中相关参数添加至H5收银台URL中，获得指定商品和优惠的收银台商品链接；
- 3) 通过OTT搭投平台，使用前端同步登录态可视化搭建组件，将H5收银台URL配置其中；
- 4) 配置其它搭投组件，发布页面。



至此，OTT端独立于Native收银台，单独开辟出一条无需跳转的收单链路。

**收单链路：**OTT投放页面-->手机扫码-->操作下单-->支付

#### 特点

- 同步登录态设备穿透；
- 下单账号统一无偏差；
- 扫码下单，由H5收银台承接，业务稳定；

## 四、能力2.0

**关键词：**同步登录态、前端单商品收银台、订单回显、可视化搭建

**描述：**

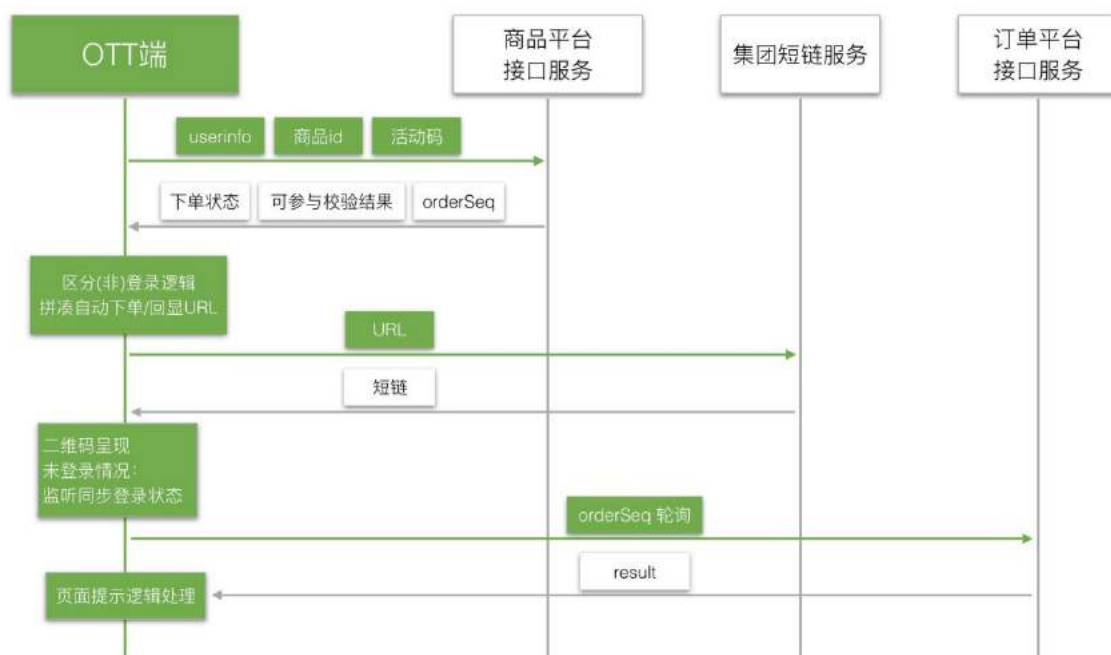
在实现能力1.0之后，用户下单转化率得到了一定的提升，可视化搭建投放也节约了运营时间成本，但是正如该项能力的初衷，是为解决同步登录态问题，收单能力只是衍生品，部分收单体验存在一定局限性，比如：

- 1) H5下单后退出，二次扫码无法回显订单；
- 2) 一经下单，优惠被占用，未支付情况下，二次扫码无指定优惠。

综合以上两点缺陷，我们通常在H5收银台配置订单提醒，以规避因第一次扫码下单使用了优惠，未支付情况下，二次扫码带来问题。当然这是一种保守方案，且在能力1.0中用户操作步骤仍为3步，这点与Native收银台并无提升。那么如果进一步简化链路，同时弥补能力1.0中的订单回显和优惠占用呢？我们对业务进行拆解：

- a) 必须具备大小屏两端同步登录态能力；
- b) 采用可视化搭建组件方案，在OTT搭投平台，使用该组件并经过配置后，在投放页面生成收单二维码；
- c) 用户扫码后，无需操作，自动下单，一步直达付款；
- d) 原订单未支付情况下，用户二次扫码，能回显订单；
- e) 原订单已支付情况下，二次进页面，提示购买成功，或者提供新商品展示；

参照该逻辑用例，其复杂性远高于能力1.0，但是一经实现，则可以真正实现简化收单付费链路，提高付费转化。那么既然业务拆解已经明确，且无明显硬伤，接下来，我们对OTT酷喵APP前后端现有收单能力和逻辑进行了解读和分析，最终，经过多方验证，该方案具备可行性，同时也梳理明确了开发链路：



围绕该开发逻辑图我们如期完成了真正意义上的前端会员商品收银台开发，实际效果展示：



收单链路：OTT投放页面-->手机扫码-->自动下单-->支付

特点：

- 同步登录态设备穿透；
- 下单账号统一无偏差；
- 扫码自动下单，一步至付费；
- 订单可回显，已购状态可区分；

## 五、能力3.0

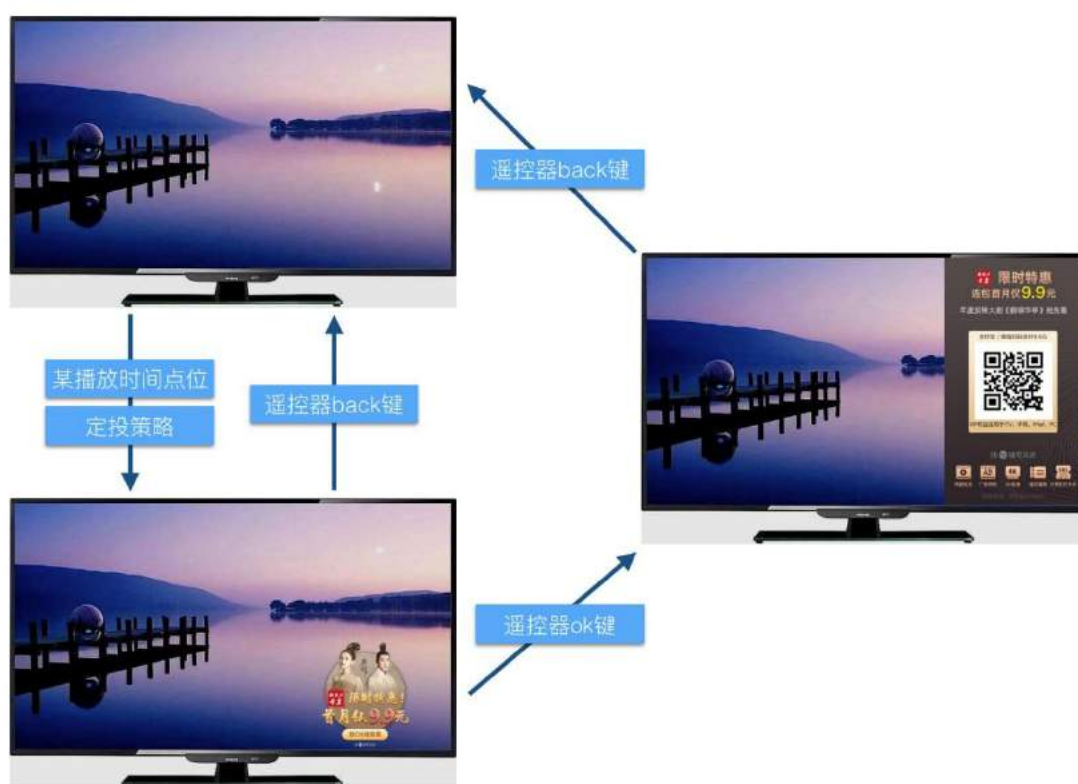
**关键词：**半屏收银台、半屏互动、圈人定投、可视化搭建、超高曝光量级

**描述：**

如果说，能力1.0和2.0只是简短了付费链路，提高了付费转化率；那么，能力3.0的建设则具备里程碑式的意义。

在《**不一样的烟火：记OTT端半屏互动能力建设**》一文中，我们简单介绍了，OTT端半屏投放能力的建设，结尾处我们提到，半屏收银台。大体描述为，在特定视频、特定播放时间点、屏幕指定区域露出引导元素、遥控器OK键拉出半屏页面、半屏页面呈现引导文案及收单二维码。虽然在手机APP端也存在“半屏”概念，也建设有半屏收银台，但是与OTT端的半屏有相当大的差异，

首先我们看下OTT端半屏交互示意图：



上述页面中的，播放器和半屏页面是存在于两种层级的视图，更准确的说，属于两种语言技术栈的视图，播放器属于原生Native，而半屏页面则是前端Weex，凭借高层级，覆盖于播放器之上，而左边镂空设置，实现“半屏假象”。当然，这也并不是简单的页面视图叠加，逻辑处理中包括了抢夺页面焦点等等OTT端特有的情况处理。



说回收单能力，3.0能力建设，引入了“精细化圈人定投”逻辑，与能力2.0的“坑位定投”相比，半屏定投具备更多维度以及更精细化逻辑处理，实现账号权益、软硬件、指定视频及播放点位、疲劳度等多方位综合圈人投放。其原理图如下：



实际效果展示：



收单链路：OTT半屏投放页面-->手机扫码-->自动下单-->支付

特点：

- 1) 同步登录态设备穿透；
- 2) 下单账号统一无偏差；
- 3) 扫码自动下单，一步至付费；
- 4) 订单可回显，已购状态可区分；
- 5) 半屏场景，渗透量级巨大；
- 6) 具备定投圈人配置能力，渗透曝光量级巨大。

六、总结与展望



经过OTT端酷喵会员线团队共同努力，在一年时间内，对现有能力进行升级扩展，取得一定成绩。尤其在场景适配方面，体验更为良好，经过数据测算，付费转化率方面也取得较大程度提升。此外，多种形式收单能力，也反向促进运营同学的营销方案思路扩散，开展了多种营销方案创新，整体效果良好。



# 我在OTT做自动化制图尝试

作者| 阿里巴巴文娱技术 罄天

## 一、背景

图片作为网页中的重要组成元素，广泛存在于各种站点中，有些站点中的图片内容已经远远超过了其他网页内容总和。如何高效的、快速的制作业务图片就被广泛的提出来。阿里有很多自动化图片生产平台，如海棠，鲁班等。

谈到自动化制图，主要有两种模式：

一是自动化模式：依赖于服务化能力包装，将核心制图能力进行抽取，任何三方通过直接调用服务能力即可完成图片的合成，此种模式完全自动化，无需任何人工干预即可制作出符合指定条件的业务图片；

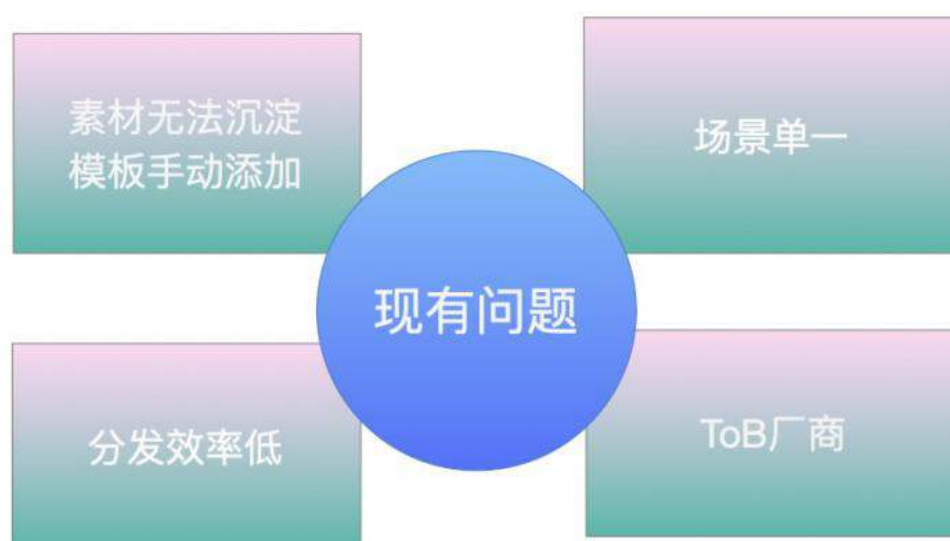
二是半自动化模式：主要依赖于业务共性的提取与升华，将繁琐的重复的业务流程通过统一的范式来解决，或多或少的需要人工干预。人工干预一方面需要人力投入，另一方面意味着可以发挥人的主观创造性。成品图除了满足指定的共性外，也可以保证输出个性，这种个性与共性并存的方式不失为一种好的折中。

## 二、OTT自动化制图

### 1、OTT自动化制图的起源

以前，优酷OTT侧有自动化制图的雏形，主要依据前端提供制图的工具平台，并将OTT主要合作厂商坑位图配置信息进行固化，然后运营在工具平台上做相应的坑位图合成。此模式在一定程度上满足了运营的合图需求。

在2019年，OTT侧开始尝试自动化制图。深入自动化制图的目的也很明显，主要基于以下痛点：



### 1) 内容源与成品沉淀

我们观察到，阿里集团开始大面积的做自动化制图的尝试，有影响力的包括鲁班系统。从OTT的业务场景出发，这些平台存在一些不足：比如，所生产物料的最终落地形式是一次性的，忽略了物料的源和产物最核心的内容价值。从渠道维度来讲，OTT侧坑位图具有高度的同质性，不仅包括内容源如节目，而且从产物的角度也如此。例如《爱我就别想太多》，某电视厂商对资源位有明确的要求，而且优酷等平台方也需要这一内容源，内容源的价值就应该被放大。另外，优酷需要将同一个成品图，投放到不同渠道上，如果能从产物角度做沉淀，成品图才能发挥最大价值。

### 2) 场景单一

制图工具解决的问题非常有限，主要是节目图的制作。前文提到，从内容源到成品图都缺少相应沉淀，这使得自动化制图服务的场景单一，和当前平台能力相比有明显差距。

现阶段OTT自动化制图涵盖了节目图、轮播、专题、动图等常见的制图场景，并结合自动化与半自动化双轨模式。半自动化场景下可以充分发挥运营创造性，基于原材料做创新性尝试；在自动化场景下，直接对接专题轮播系统，使得依赖于特定模板的制图无需人工参与。更进一步，在大数据推荐上也有相应场景，比如依赖于客户端唯一标识实现个性化的专题类推荐，这使得自动化制图一改之前“自动化”包装的外表。

### 3) 分发效率

平台建立之初，从素材源到成品的完整链路都在本地环境完成，依赖于工具平台下载的成品图对接给运营、第三方进行投放。严重依赖于运营的人力投入，分发效率低，其原因是从内容源到成品库，到最终的分发链路缺乏一致性，所以使得整个系统没有“活”起来。

所以从平台建立开始，我们就在探索一条“活”的完整链路。通过链路的升级，不仅能完成内容源、成品库的沉淀，也包括最终分发链路的升级，从而摆脱传统工具平台面临的点状而非面状链路。

#### 4) TOB厂商

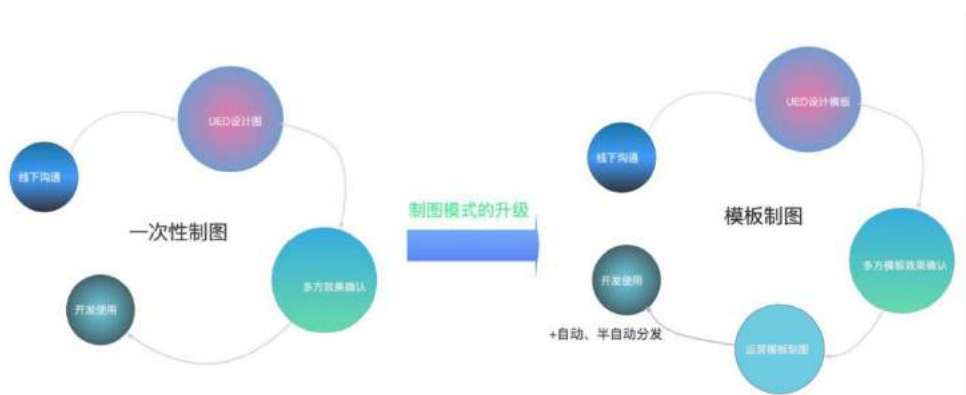
“巧妇难为无米之炊”，在自动化制图场景下，所谓的“米”就是模板，模板的格式包括但不限于PSD、SKETCH等，也包括如SVG，比如海棠。“米”是最核心的内容，如果不能从特定的场景中提取内容共性，那么自动化制图领域下的效率提升就非常有限，因此共性才是最应该被重视的特征。

并且，这种共性不仅存在于单个内容提供商，也存在于提供商之间。

## 2、OTT自动化制图的模式

自动化制图的优势非常明显：

第一是链路的升级：通过自动化制图的流程，可将原来“线下讨论-UED设计-效果确认-开发使用图片”的一次性长链路，转化为“UED设计模板-运营制图-自动/半自动分发”的非一次性完整流程，缩短沟通成本。



第二，内容共享红利：基于系统设计的高质量毛料库，产品库可进一步沉淀，打通多方，减少任何三方重复设计的可能性，将内容价值从原来的一方扩展到整个合作方生态，形成完整的内容回流链路。

第三，分发链路的建设：基于系统设计的成品图无需任何人工对接，可直接输出到渠道三方或者对接任何系统。现有可行的尝试，如专题系统自动化，大数据的个性化推荐场景等，完成从内容生产到内容自动分发的一体化能力建设。

制图效率环节：在半自动化制图场景，站外投放时间可从原来周级缩短为小时级，计件的

时间成本减低50%~60%。而在自动化制图场景，无需任何人工干预的方式，使得自动化制图能真正发挥价值。

### 三、OTT自建自动化制图平台

可能会有人问，阿里已经有鲁班系统等自动化制图平台，为什么优酷OTT要自建？

#### 1、为什么考虑自建？

与鲁班系统服务的（侧重商品化属性）场景不同，OTT场景更侧重媒资属性。图片主体维度的偏差，使得很多原本在商品制图场景下的规则被打破。比如一个简单的元素缩放操作，常规是按照等比加移动的方式解决。但是在以人物为中心的场景，这可能并不合理。因为在以人为主体的场景中，所有模板的指定需要考虑“人物”，而非人物所在元素图片的缩放，简单的说就是人脸，否则可能面临着人物缩放不满足模板要求的场景。因此，我们需要更进一步引入算法或人工打标的新思路。而且，除了元素缩放的特有场景，其实这种区别还很多。比如特定场景的模糊效果、动图绘制、自动化能力输出等等，这种基于特定业务特性的需求海棠确实难以做到定制。

#### 2、商品图使用海棠

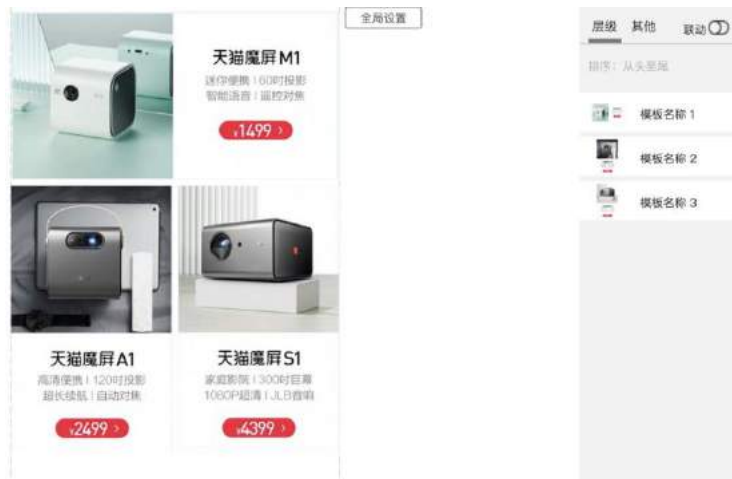
除了生产媒资海报图以外，OTT业务也需要生产商品图。比如天猫魔盒：



在没有自建系统之前，所有天猫魔盒的图片生产全部依赖于海棠。设计师从海棠后台传入模板，然后基于此模板在海棠上做相应的坑位图。但自建系统后也慢慢尝试切回到自有系统，主要原因有两个：

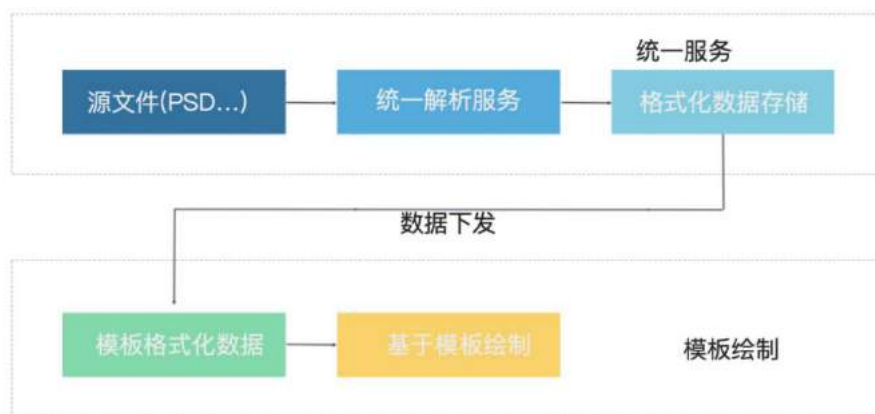
第一，内容回流：OTT业务本身是一闭环系统，媒资图和商品图缺一不可。这使得内容隔离渐渐不被接受。内容回流到自有系统后可以做统一管理、分发，享受现有系统提供的能力；

第二，定制化能力：设计师的需求往往依据业务的迭代渐渐变化，渐进增强成为一个常态。因此，当自己的诉求不被满足或者不会被满足后可能会渐渐的丧失信心，进而转向新的平台。而自建系统正好提供了这样一个契机。下图提供了一个基于多模板拼接的述求。任何模板可以随意拖进工作区，然后对模板元素加工并设置模板特性，比如模板间距等。



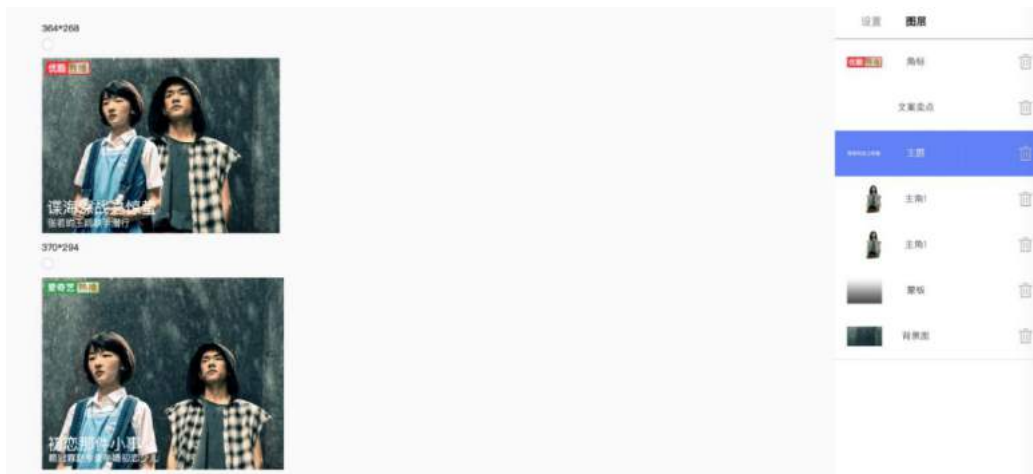
## 四、OTT自动化制图的流程

关于自动化制图我们是如何实现的？整体流程可以简化为下图：



## 1、模板解析与数据格式化

自动化制图的前提是内容的共性，共性点更进一步提取，就是模板，模板格式包括但不限于PSD、SKETCH、SVG等。任意一张成品图都可看做是基于某一种模板生产出来的，这张成品图由不同图层构成，这和Photoshop中的图层是同一个道理。比如以下示例图就包含了角标、文案卖点、主题、主角、蒙版、背景图等诸多图层。通过对不同图层的自由组合、编辑，得到最终的成品图。



上面讲到模板是由不同的图层所描述的，更形象的，模板就是解析后得到的格式化数据，描述了模板中的元素信息，如位置、尺寸、数量等等。而成品图可以看做是基于这些约定（也可以看做是图纸）生产出来的最终元素。





上图就是对PSD的解析后获取到的模板缩略图。当然，正如前文所讲，解析后图片的存在形式也不再是图片本身，而是格式化的数据，一般是对应的JSON。该JSON包含了模板指定的所有元素信息：

```
1.  const templateInfo = {
2.    "modelBase": {
3.      // 包括模板本身的信息，如尺寸等
4.    },
5.    "psdLayerInfoList": null,
6.    "person": [
7.      // 模板包含的主体个数，主体可以是人或者其他元素
8.      // 也包括主体的人脸信息等
9.    ],
10.   "bgPic": {
11.     //背景图
12.   },
13.   "themeGradientMask": [{
14.     //主题渐变
15.   }],
16.   "showNameH": [{
17.   }],
18.   "showNameV": [{
19.   }],
20.   "mMask": [{}],
21.   "mAppLogo": [{
22.   }],
23.   "mFontAppName": [{}]
24. }
```

## 2、前端依据模板绘制流程

服务端做了统一的数据格式化后，给到前端的是格式化的数据，数据指定了模板包含的所有元素信息，前端基于该元素信息进行绘制。

## 1) 分层绘制

HTML5页面中图层的概念大家应该已经很熟悉了。讲到HTML5的网页分层就需要深入理解Chrome渲染原理中的RenderObject、RenderLayer、Graphiclayer等几棵树。

除了网页会分层以外，Canvas中绘制的元素也可以分层，分层绘制有很多优势。比如在游戏场景中，很多背景类的图层需要重绘的可能性远比动态元素，如障碍物低得多。因此，在每一帧的绘制行为中，可以绕过相应背景图的绘制，直接绘制当前场景变化的图层即可，这与Chrome网页分层要解决的问题是一样的。

```
1. class SmallMultiLayerCanvas {
2.   constructor(id) {
3.     this.id = id;
4.     this.canvas = document.getElementById(id);
5.     this.ctx2d = this.canvas.getContext('2d');
6.     this.layers = [];
7.   }
8.   static extend = function (defaults, options) {
9.     var extended = {}, prop;
10.    for (prop in defaults) {
11.      if (Object.prototype.hasOwnProperty.call(defaults, prop))
12.        extended[prop] = defaults[prop];
13.    }
14.    for (prop in options) {
15.      if (Object.prototype.hasOwnProperty.call(options, prop))
16.        extended[prop] = options[prop];
17.    }
18.    return extended;
19.  };
20.  // 添加图层
21.  addLayer(obj) {
22.    const layer = SmallCanvas.extend({
23.      id: Math.random().toString(36).substr(2, 5),
24.      show: true,
25.      render: function (canvas, ctx) { }
26.    }, obj);
27.    if (this.getLayer(layer.id) !== false) {
28.      return false;
```

```
29. }
30. this.layers.push(layer);
31. return this;
32. };
33. //渲染所有图层
34. render() {
35. var canvas = this.canvas;
36. var ctx = this.ctx2d;
37. this.layers.forEach(function (item, index, array) {
38. if (item.show)
39. item.render(canvas, ctx);
40. });
41. };
42. //获取一个图层
43. getLayer(id) {
44. var length = this.layers.length;
45. for (var i = 0; i < length; i++) {
46. if (this.layers[i].id === id)
47. return this.layers[i];
48. }
49. return false;
50. };
51. // 移除一个图层
52. removeLayer(id) {
53. var length = this.layers.length;
54. for (var i = 0; i < length; i++) {
55. if (this.layers[i].id === id) {
56. removed = this.layers[i];
57. this.layers.splice(i, 1);
58. return removed;
59. }
60. }
61. return false;
62. };
63. }
```

以上是一个简单的分层绘制的类，通过该类可以随意新增、移除、获取、渲染任意的图层。这也是很多复杂的多图层绘制框架的最核心思想，比如fabric.js或者konvajs。有了这样的图层管理框架，就可以根据服务端下发的格式化数据来绘制模板中指定的任意元素，以模板为图纸，生产出符合设计规定的核心产品。例如：

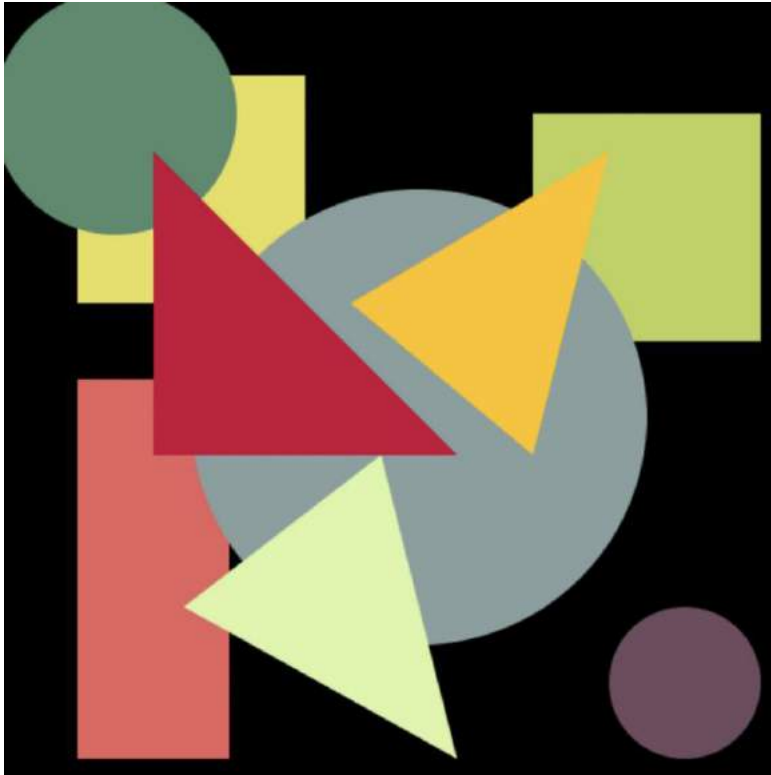
```
1. <canvas id="theCanvas" width="512" height="512"></canvas>
2. 在该Canvas上，我们绘制出几个指定的图形看看效果如何:

3. const myCanvas = new SmallMultiLayerCanvas("theCanvas");
4. myCanvas.addLayer({
5.   id: 'background',
6.   render: function (canvas, ctx) {
7.     ctx.fillStyle = "black";
8.     ctx.fillRect(0, 0, canvas.width, canvas.height);
9.   }
10. })
11. .addLayer({
12.   id: 'squares',
13.   render: function (canvas, ctx) {
14.     ctx.fillStyle = "#E5E059";
15.     ctx.fillRect(50, 50, 150, 150);
16.     ctx.fillStyle = "#BDD358";
17.     ctx.fillRect(350, 75, 150, 150);
18.     ctx.fillStyle = "#E5625E";
19.     ctx.fillRect(50, 250, 100, 250);
20.   }
21. })
22. .addLayer({
23.   id: 'circles',
24.   render: function (canvas, ctx) {
25.     ctx.fillStyle = "#558B6E";
26.     ctx.beginPath();
27.     ctx.arc(75, 75, 80, 0, 2 * Math.PI);
28.     ctx.fill();
29.     ctx.beginPath();
30.     ctx.fillStyle = "#88A09E";
```

```
31. ctx.arc(275, 275, 150, 0, 2 * Math.PI);
32. ctx.fill();
33. ctx.beginPath();
34. ctx.fillStyle = "#704C5E";
35. ctx.arc(450, 450, 50, 0, 2 * Math.PI);
36. ctx.fill();
37. }
38. })
39. .addLayer({
40. id: 'triangles',
41. render: function (canvas, ctx) {
42. ctx.fillStyle = "#DAF7A6";
43. ctx.beginPath();
44. ctx.moveTo(120, 400);
45. ctx.lineTo(250, 300);
46. ctx.lineTo(300, 500);
47. ctx.closePath();
48. ctx.fill();

49. ctx.fillStyle = "#FFC300";
50. ctx.beginPath();
51. ctx.moveTo(400, 100);
52. ctx.lineTo(350, 300);
53. ctx.lineTo(230, 200);
54. ctx.closePath();
55. ctx.fill();
56. ctx.fillStyle = "#C70039";
57. ctx.beginPath();
58. ctx.moveTo(100, 100);
59. ctx.lineTo(100, 300);
60. ctx.lineTo(300, 300);
61. ctx.closePath();
62. ctx.fill();
63. }
64. });
65. myCanvas.render();
```

上面的代码通过链式调用在Canvas中添加了4个对象，id分别为background、squares、circles、triangles。而且每一个对象都有相应的render方法，该方法指定了元素本身在Canvas的上下文是如何绘制的。基于以上代码可看到在Canvas中绘制的效果：

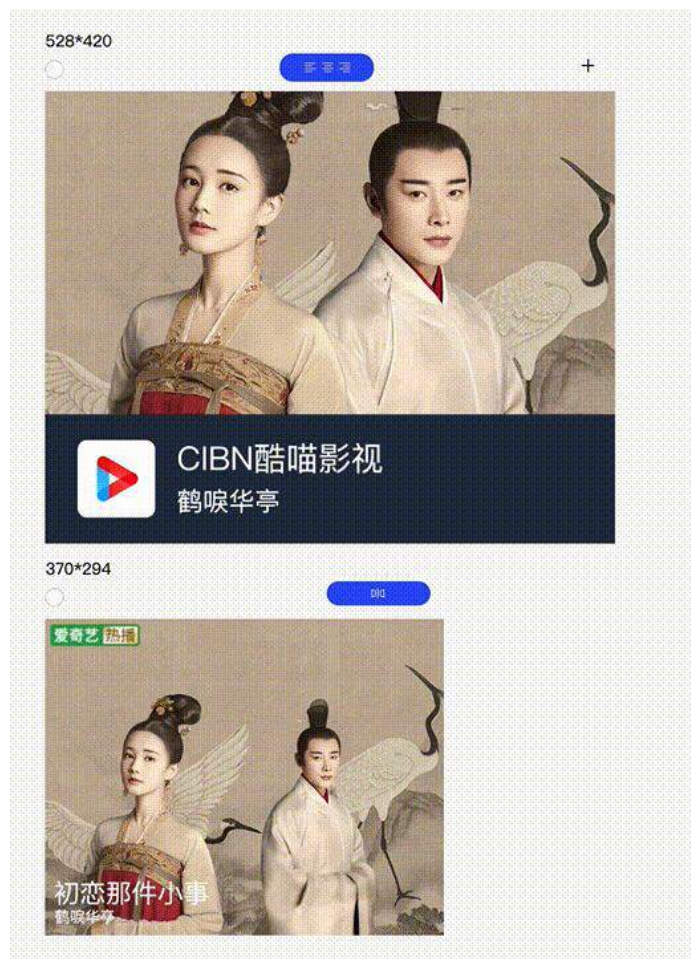


## 2) 单量与批量模式实时渲染

本小节将讲述首轮绘制后如何基于用户输入做相应的更新。其实所有的半自动化绘图场景，运营不可能只根据某一个模板进行绘制，换句话说，多模板同时绘制的场景必须加以考量。

比如，大多数场景下，运营需要同时基于模板来绘制海信、康佳、歌华、LG的所有坑位图然后导出或者投放，进而摆脱每次只能单独绘制导出单模板的低效模式。因此基于多模板实时绘制渲染的方式就亟待解决。基于此，在半自动化绘制场景中，天生支持多模板实时渲染绘制的模式，正如下面的动图所示：

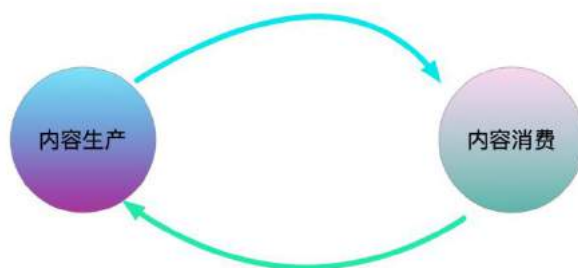




上图展示的是多模板实时修改的场景，也就是所谓的联动模式。但在未开启联动模式的场景下，所有的修改只针对单模板生效。因此在满足成品图共性的同时又保证模板的个性。

### 3、数据统一存储在服务端

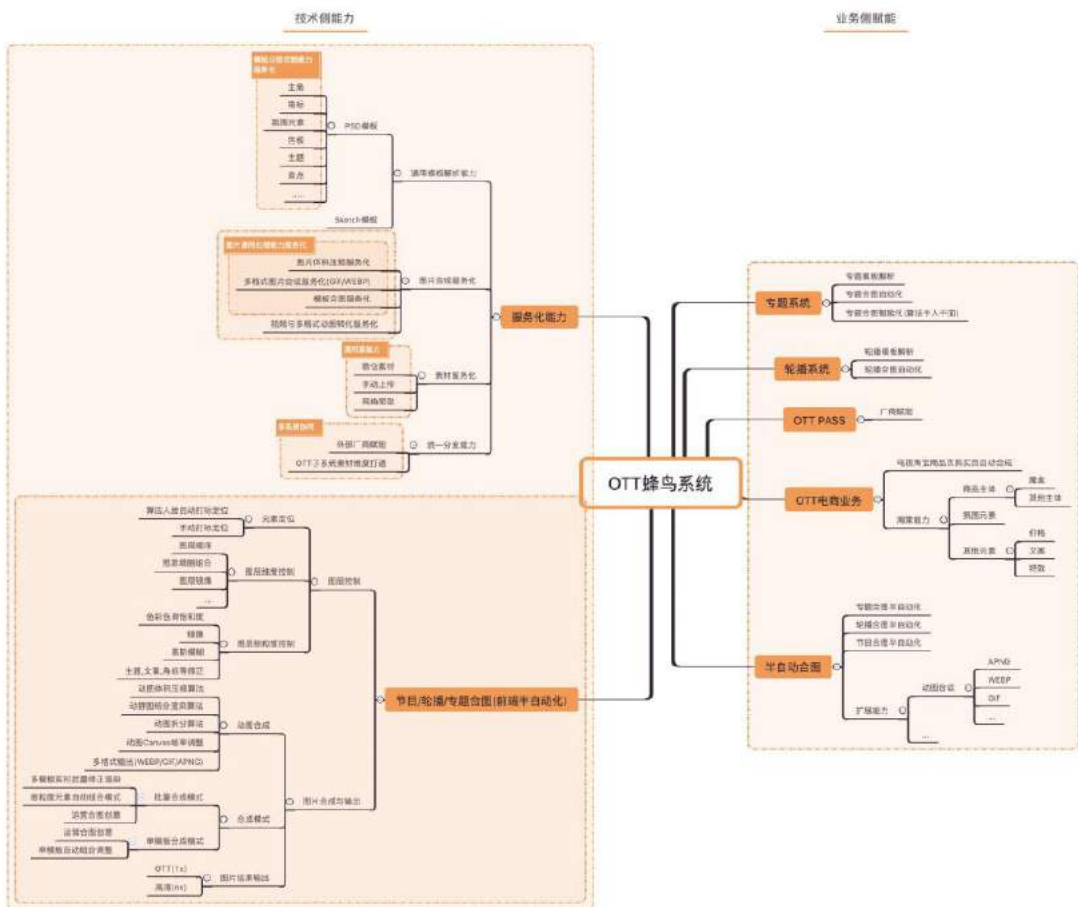
前面讲过，工具化平台的设计思路没法做到链路的完整串联，进而完成内容的回流，这在平台化的思路下是行不通的。平台化解决问题的思路是：从内容生产到内容消费的完整链路串联。



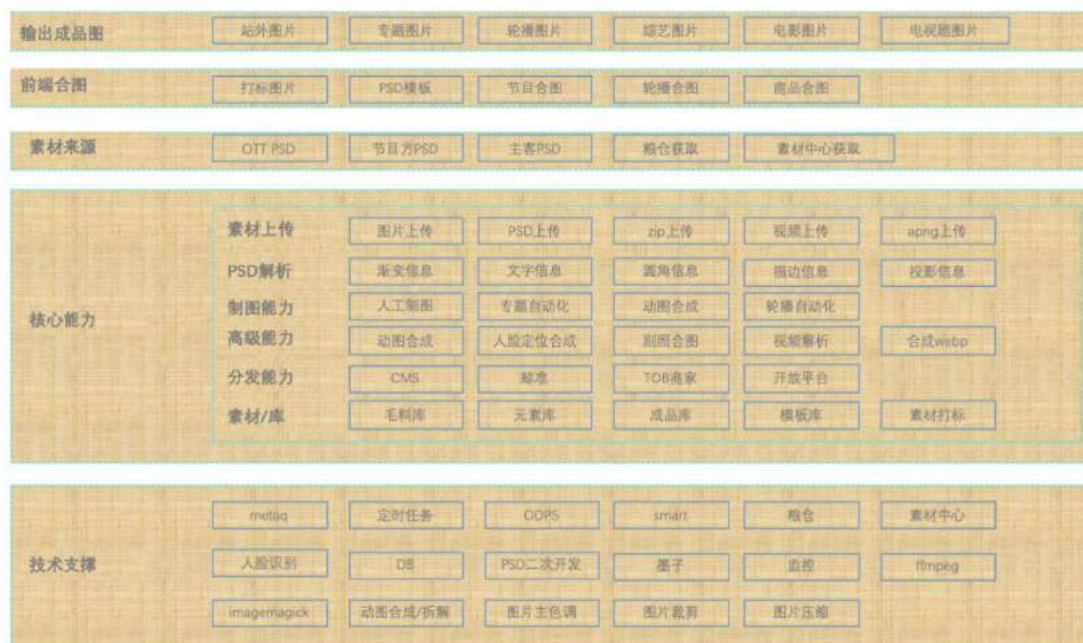
基于此，从输入到输出，到最终的分发都需要流动起来，这一切的前提都基于数据的存储，从输入源到输出结果。基于存储的结果，完成了从自动化生产，个性化等自由推荐。

#### 四、OTT蜂鸟制图场景的主要输出与输入

下图展示了制图平台在业务上所做的尝试，也总结了在支撑业务的同时，如何在技术上做范式的探索。



从1.0到2.0，是整个系统架构的升级。系统现有支持能力在OTT场景下已经逐渐完善起来。



## 1、技术侧主要输出

### 1) 服务化的能力



服务化的能力是上面所述的解决问题范式的具体形式，而制图平台服务化的能力已经涵盖了包括：通用模板解析能力，图片合成服务化能力，素材服务化能力，统一分发能力等。

- 通用模板解析能力，不仅适用于制图场景的数据格式化，在智能化领域也有涉及；
- 图片合成服务化能力，不仅可用于制图场景，对于动画合成场景也有渗透；
- 至于素材的服务化，分发能力的联合在业务赋能的同时，也能谋求业务发展方向的新思路。

### 2) 半自动化合图尝试

服务化能力将制图的触角做了极大的延展，但在半自动化的制图场景，依然需要探究新的制图范式。基于此，我们产出了自有的Canvas合图尝试，这与鲁班、海棠的模式有极大的差异。这条路没有太多的参考，很多问题需要自己去挖坑和填坑。

## 2、无人工自动化制图

自动化制图具有极大的吸引力，无需运营任何手动制图干预，直接在系统中选择相应的内容集合即可。因此，这种业务赋能尝试也被极大的重视起来。在OTT业务范围内比较成功的案例就是专题系统：

\* 专题来源: ☐ 手动生成 ☒ 关联节目SCG ☐ 推荐理由 ☐ 旧专题

\* scgid: 9-经典港片 X

\* 封面图: 刘德华经典港片

自动生成 规则/效果

专题背景图: 

专题页播放器: 

节目数量: 

状态: 

过频预告片: 

大片连连看

1.取专题下前5个节目竖向海报组合  
2.当专题下节目数量小于5个。不会生成海报图  
3.封面图生成时间大约3分钟

通过指定内容集合(scgid)以及相应的封面图生成规则就可生成相应的专题推荐内容，极大的节省了人力成本。

创新落地

# 不一样的烟火：记OTT端半屏互动能力建设

作者| 阿里巴巴文娱技术 魏家鲁

## 一、背景

酷喵APP，一方面拥有巨大的DAU量级，一方面受限于种种原因，营销投放渗透率还有很大提升空间。在技术上如何寻求某些突破口，实现端内投放渗透提升？

经过多方讨论，我们将注意点放在了酷喵APP“播放页”上。作为视频服务APP，依托千万级的强劲日活人群支持，“视频播放页”达到PV上亿规模，在端内PV量级排名中无可争议的第一。

如何建设OTT播放页，问题如下：

OTT播放页将如何承载投放？

互动及交互方式是怎样？

容错机制如何处理？

定向投放如何实现？

用户观影体验如何保障？

共建开发如何分工？

.....

## 二、横向调研

相比较而言，手机APP端、PC端、OTT端在播放页中，均有相应营销投放，在“播放器区”均建设有“互动投放”，经情况了解及数据检测，我们发现播放器区投放渗透效果远远优于其它区域坑位。我们判断，如果OTT端新开辟播放器区投放能力，将在相当程度上提升投放渗透率。

下面我们对手机APP端、PC端、OTT端播放器区投放，进行简单介绍。

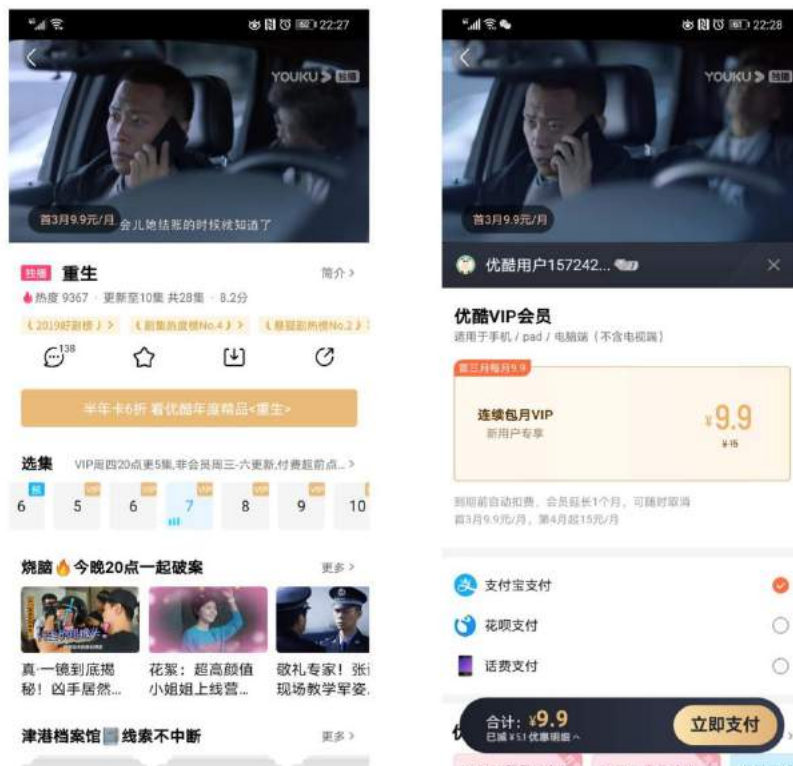


## 1、PC端



PC端在播放页-播放器中，策略投放主要是“付费引导”，如图所示，PC播放器经过UPS鉴权，获取该用户之于该视频的权益数据，根据数据情况确定是否露出引导文案及链接，点击交互链接拉起“PC弹框收银台”。

## 2、手机APP端



手机APP端与PC端类似，也是在不影响用户观看体验情况下，经过鉴权，判断是否露出付费引导按钮。不同的是，无论全屏还是小窗播放情况下，手机APP端在用户点击付费引导按钮后，都是拉起半屏Native收银台(右侧图)，此时，仍然不影响用户观看视频。

3、OTT端



与PC端和手机APP端一致，均为在不影响用户观看体验情况下，经过鉴权，判断是否露出付费引导按钮。而用户遥控器OK键，则新打开收银台页面，此时视频观看被阻断。

4、总结

我们对比了PC端、手机APP端、OTT端播放器区运营投放能力：

端平台	露出形式	打开形式	投放内容	定投查询维度	支持
PC端	播放器区-文案/链接露出	打开弹框	付费引导/收单	视频id/userId	H5开发
手机app端	播放器区-按钮露出	拉起半屏	付费引导/收单	视频id/userId	native原生开发
OTT端	播放器区-文案/按钮露出	新开页面	付费引导/收单	视频id/userId	native原生开发

经过调研发现，PC端、手机APP端、OTT端播放页-播放器区互动投放大致特点：

- 1) 播放器区互动投放，基本不影响用户观感体验(OTT端除外)；
- 2) 播放器区互动投放分两步，先露出引导内容，经由用户操作，展示完整投放内容；
- 3) 用户可通过操作，将该区域投放关闭，恢复初始状态(OTT端除外)；
- 4) 具备特定维度定投能力（圈人设定售卖商品）；
- 5) 目前三端投放能力主要为付费引导/收单；

纵观看来，PC端、手机APP端、OTT端的播放页“投放”更像是一种“配套能力”，紧紧围绕视频付费引导展开，并未涉及活动投放，且形式固化：通过权益判别，实现在站/端内每一个需付费视频(相对用户权益)，精准付费引导。而在OTT端我们更希望的是同时具备“付费引导/收单”和“活动投放”两种能力。因此，在OTT端播放互动建设规划中，势必与先行思路有所区别。

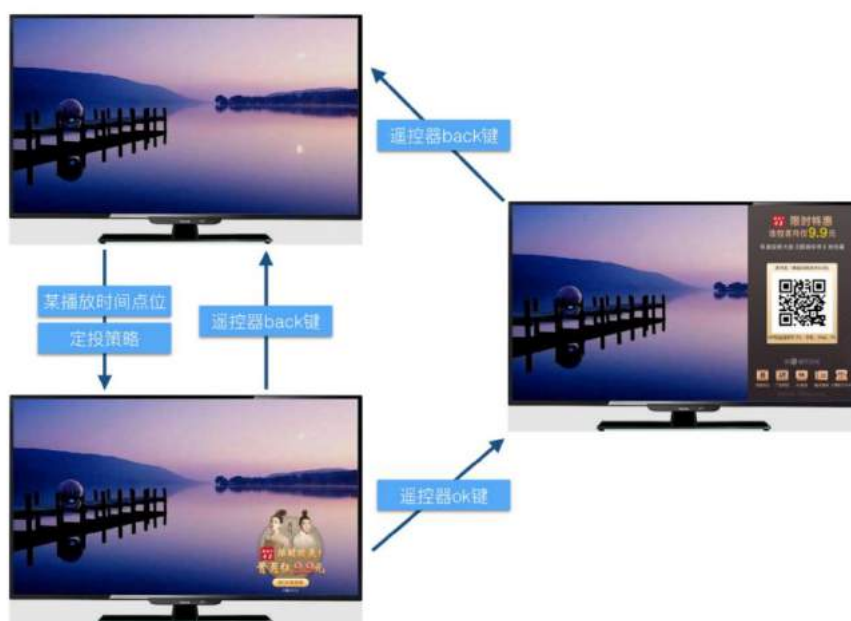
### 三、建设方案

#### 1、方案维度

OTT端最终确定OTT播放页半屏互动以“设备硬件/APP版本/账号权益状态”综合维度进行付费引导投放和活动投放，投放内容本身独立于播放页/播放器。实现多种维度，批量定投收单以及活动投放。

#### 2、交互方案

经过多方沟通，我们逐渐梳理出一条OTT端播放页-播放器区营销投放交互方案。即在OTT播放页-全屏播放状态下，通过定向投放，在特定播放时间点位，露出“引导卡片图”，用户遥控器OK键后，由右侧拉出半屏页面，在该半屏页面完成既定营销交互，我们称之为“OTT半屏互动”。交互图如下：



### 3、技术支持方案

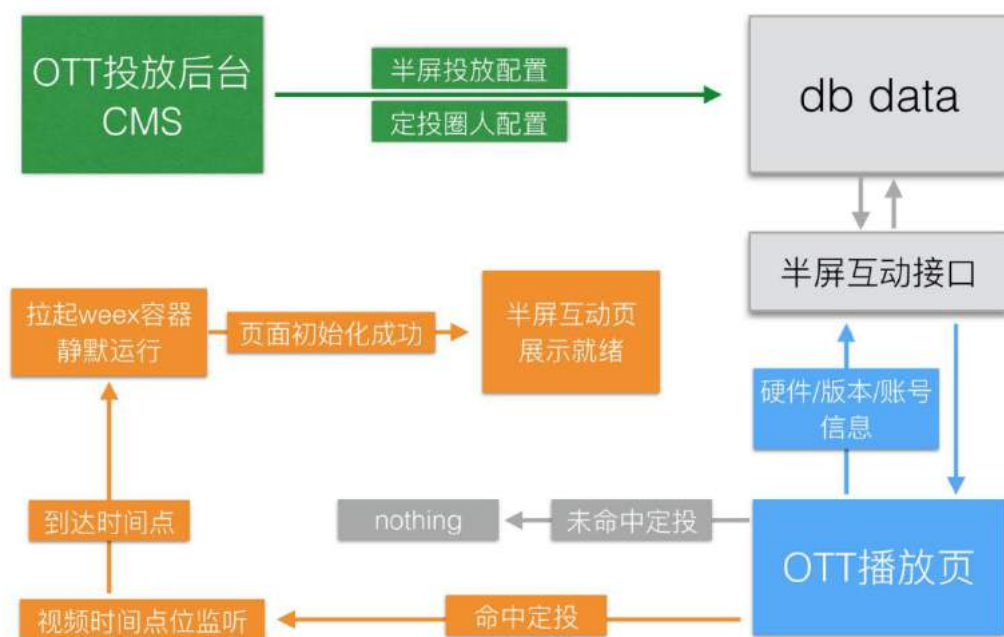
目前在OTT端酷喵APP开发过程中，存在3种技术栈，即原生Native、weex、H5。原生Native作为酷喵APP主要开发项；weex一方面作为坑位投放页面开发技术栈，一方面作为能力补充，补位Native开发；H5为早期投放页面开发技术栈，本财年已逐步被weex取代。

实现方案	优点	缺点
native原生开发	加载性能好、稳定	发版/覆盖周期长、灵活性欠佳
webview承载H5开发	开发成本小、发版/覆盖周期短、页面开发灵活性好	页面加载效率较低
weex容器承载Rax开发	开发成本较小、发版/覆盖周期短、页面开发灵活性好	页面加载效率相较native略低

综合评比，我们得出方案3为最优方案，后续的共建分工及实现逻辑也相应浮出水面。

### 4、共建分工

交互设计与技术支持方案已经确定，接下来梳理业务开发链路，并进行拆解，链路逻辑图如下：



按照上方链路逻辑图，多方共建分工已然明确，本次共建涉及：

- 1) Native方 - OTT weex容器、播放页/播放器能力支持；
- 2) CMS方 - OTT投放CMS能力支持
- 3) 后端 - 接口支持
- 4) 前端 - 交互调度/weex开发

## 四、体验保障

### 1、观感保障

虽然OTT全屏播放，播放画面尺寸较大，但半屏互动页由屏幕右侧拉出，势必会遮住部分播放画面，影响用户观看体验；此外，OTT页面操作，涉及“焦点切换”（OTT端页面有且最多只有一个焦点选中）。这两点需要寻求突破点解决，经过与播放页面/播放器同学沟通，得到一个相对完美的解决方案，即：

- 1) 半屏互动定投，“引导卡片”在特定时间点位露出（本质为weex页面），此时由于该卡片尺寸较小，读用户观感体验营销较小，不做处理；
- 2) weex页面高层级浮在播放器之上，为保证用户遥控器交互，weex页面通过Native api，执行抢夺页面焦点逻辑；
- 3) 用户遥控器点击“OK键”，半屏页（开发/设计约定宽600px）由屏幕右侧拉出，在该节点，前端weex层调用Native bridge方法，通知播放器，收缩播放器尺寸；
- 4) 随后用户遥控器“back键”，则weex层立即通知播放器回复全尺寸，之后weex页面自行执行页面实例销毁，页面焦点重新回归播放器；

### 2、前端容错

前端作为最下游展示业务，直接面向观影用户，上游任意一环节出现故障，都可能导致前端业务出错，为保证该环节健壮性，增强前端容错势在必行。在weex容器与前端共建过程中，根据架构设计，前端在两个方面进行了容错处理：

#### 1) 静默运行-错误防控

上文“共建分工”链路图中，在命中定投且到达时间点位后，weex容器被拉起，且为静默运行，并未展示在界面中。此时前端weex半屏页被打开，在实例化weex页面过程中，前端逻辑需要通过Native api方法获取半屏配置信息、查看屏幕播放状态等操作，此阶段，一经出现业务错误，则前端weex逻辑终止实例化，并执行weex实例销毁，将业务错



误规避于未展示阶段：



2）前端紧急开关

考虑到播放页作为酷喵APP最核心页面之一，且PV量级巨大，OTT投放平台配置有状态开关。此外，为保证开关状态实时性，在前端层面增设紧急开关逻辑，以应对突发故障，双重保险。



在CMS及前端MT平台，两道开关保险，确保半屏互动投放，在故障发生时，迅速止血，且实现用户无感知，不影响用户体验。

五、场景建设

在半屏互动能力支持下，前端线完成两大场景建设，即：半屏收银台、半屏活动投放。

1、半屏收银台

在最初，OTT端仅有Native收银台，后续借助《OTT 端登录态设备穿透：扫码登录与反登录》一文中介绍的同步登录态能力，在前端开发完成weex版“单商品收银台”，并将该收银台组件化，实现可视化搭建。

在后续的“OTT半屏收银台”建设过程中，复用“单商品收银台”能力，完成与半屏互动能力的结合，“OTT半屏收银台”就此诞生。

区别于手机APP端半屏收银台，OTT端半屏收银台为独立于播放页/播放器业务，以“设备硬件/APP版本/账号权益状态”多维度进行付费引导查询，实现多种维度、批量定投收



单，描述为：

特定硬件类型、特定APP版本、特定会员情况、指定会员商品、指定视频内容播放中曝光。



如上图，OTT酷喵APP在完成该半屏能力建设后，端内三种收银台能力形成：Native综合收银台、weex单商品收银台、weex半屏收银台：

名称	描述
native综合收银台	能力齐全、策略丰富、加载性能好； 单独页面，需跳转
前端weex单商品收银台	使用灵活，可视化搭建，任意坑位投放，无需跳转，一步扫码至付费； 不支持多商品
前端weex半屏收银台	使用灵活，可视化搭建、用户渗透量级大，无需跳转，一步扫码至付费； 不支持多商品

## 2、半屏活动投放

前文提到，OTT端播放页半屏互动投放为独立存在，并非集成式，且具备多维度定投能力，站在运营同学角度来，半屏互动投放无异于一个新的、高曝光量级的“定投坑位”。

与“半屏收银台”相比，“半屏活动页”偏向于业务开发，不再拘泥于商品、付费、收单的固定范畴，而是面向于个性化的业务诉求，因此在开发阶段也更加多样化。例如下图：双11预约猫晚。



## 六、总结

借助OTT酷喵APP自身硬件特性及用户使用习惯，创新完成“半屏互动”建设，该能力既满足用户线运营“大量级曝光”的诉求，满足会员线运营“精准投放、提高渗透”的诉求。根据数据结果，在曝光UV量级上，实现端内投放曝光UV提升200%。

半屏互动能力已经具备，也经受住了实际考验，作为技术开发人员，除了考虑业务支撑外，还需要考虑开发效能提升。因前端半屏业务相对常规业务而言，开发与调试都更加复杂。在之前的前端半屏业务开发过程中，尤其在拉取配置数据、抢取焦点等环节存在诸多“弯弯绕”，调试/测试过程也不同于常规项目，且新手上手开发成本较高。

在上述痛点之下，寻求突破势在必行，在经过多轮方案验证，最终敲定并完成两种优化建设：

### 1、半屏可视化搭建

前端开发一套“半屏可视化搭建套件”，集成于阿里文娱的可视化搭建平台，套件内置播放中调用监听、抢取焦点、获取配置、实时监控、紧急开关等逻辑。由运营同学在平台直接可视化搭建发布，应对常规半屏业务，无需开发介入；

### 2、半屏投放-能力模板

主要应对高复杂度/个性化业务，能力模板完成播放中调用、抢取焦点等逻辑，具体业务实现由业务方<如电淘>自行开发。双方逻辑解耦，降低开发难度与时间成本，提高半屏业务投放稳定性。

# OTT端性能优化建设之本地缓存设计

作者| 阿里巴巴文娱技术 魏家鲁

## 一、背景

目前，做2C业务的应用，更多强调SSR、客户端缓存以及PWA等，以实现首屏加载体验优化、秒开等性能指标，相比较而言，这些策略更加“综合”“强壮”，如果合理运用以及借助端能力，实现冷启动提速、首屏加载优化、秒开等不在话下。

但是笔者业务服务于“OTT端酷喵APP”前端业务，主要是酷喵APP的HTML5投放(目前更换使用Rax)，而端内浏览器并不支持service worker (PWA)，且受制于端及浏览器内核，并无zcache类似能力。至此，大写的无奈涌上心头，这种情况还能不能抢救一把？答案是：可以，localStorage迂回包抄方案。也鉴于此，本文方案诞生，虽不完美，但是终究有闪光所在。

## 二、方案

在localStorage出现之前，浏览器层面可用的本地存储只有一个：cookie。作为前端本地存储的独苗，cookie在很长一段岁月里扮演了极其重要的角色，如账号数据存储、状态标记等等。然而，4kb的容量让cookie在资源缓存道路上无力前行。随着HTML5的兴起，一个崭新的名词出现“localStorage”，容量比cookie大千倍、同步读写的特点，一经面世就被认定为实用型本地存储策略，因而被广大开发者关注。

## 三、优势与局限






### 1、localStorage的优势

- 1) localStorage拓展了cookie的4K限制，5mb(各浏览器略有不同)；
- 2) 键值对格式，同步读写，使用简单；
- 3) 持久存储，不随请求发出，不影响带宽资源。

## 2、localStorage的局限

- 1) 浏览器支持不统一，比如IE8以上的IE版本才支持localStorage这个属性；
- 2) 目前所有的浏览器中都会把localStorage的值类型限定为string类型，这个在对我们日常比较常见的JSON对象类型需要一些转换；
- 3) localStorage本质上是对字符串的读取，大量读写影响浏览器性能。

### 兼容情况

						
IE	Firefox	Opera	Chrome	Safari	iPhone	Android
8.0+	3.0+	10.5+	4.0+	4.0+	2.0+	2.0+

### 本地缓存分析

首先5MB的容量，对于存储前端部分常用的js、css等，如果经过系统逻辑筛选存储，虽然不从容，但也算能用，这也是localStorage可以作为前端本地缓存的第一要素。当然这5MB的容积，“大”是相对于cookie而言，真正存储资源则需要筛选和过滤的。比如一个网页中，包含大量js、css、img、font文件，这个量级是不可预估的，如果选择全部存储，5MB空间可能瞬间爆满。一般而言，我们更倾向于存储js、css这种资源（尤其是阻塞式加载的js资源）。

其次，localStorage只能存储字符串类型数据，这就决定了我们无法使用< script src="static/a.js">的形式加载js资源，这种情况下，我们无法拿到“文件句柄(字符)”，同时也无法赋予其本地缓存的资源。如果要拿到js文件句柄则需要转换思路，使用xhr或则fetch的形式得到文件字符。

再次，得到字符后，我们可以选择将其存储至localStorage的同时，进行eval或者new Function，使js代码执行。

最后，当页面再次打开，我们可以根据js路径情况，判断本地是否有缓存资源，如果有，则取出并且eval/new Function，使代码执行；如果没有，则使用xhr/fetch进行请求，文件资源返回后，存储至localStorage并执行eval/new Function。

至此，整个缓存逻辑即告成功。下面我们以实际代码形式进行技术方向解析。

## 四、缓存技术实现

客户端本地请求/缓存/加载器

### 1、localStorage api

设置localStorage 项，如下：

```
localStorage.setItem('keyName', 'keyValue');
```

读取 localStorage 项，如下：

```
localStorage.getItem('keyName');
```

移除 localStorage 项，如下：

```
localStorage.removeItem('keyName');
```

移除所有localStorage 项，如下：

```
localStorage.clear();
```

### 2、实现原理图：



### 3、主要代码实现：

```
1. (function () {
2.   constoHead=document.getElementsByTagName('head')[0];
3.   const _localStorage=window.localStorage|| {};
4.   //创建ajax函数letajax=function (_options) {
5.     constoptions=_options|| {};
6.     options.type= (options.type||'GET').toUpperCase();
7.     options.dataType=options.dataType||'javascript';
8.     constparams=options.data?formatParams(options.data) : '';
9.     //创建-第一步letxhr;
10.    if (window.XMLHttpRequest) {
11.      xhr=newXMLHttpRequest();
12.    } else {
13.      xhr=ActiveXObject('Microsoft.XMLHTTP');
14.    }
15.    //在响应成功前设置一个定时器（ 响应超时提示 ）
16.    consttimer=setTimeout(function () {
17.      //让后续的函数停止执行xhr.onreadystatechange=null;
18.      console.log('timeout:'+options.url);
19.      options.error&&options.error(status);
20.    }, options.timeout||8000);
21.    //接收-第三步xhr.onreadystatechange=function () {
22.      if (xhr.readyState==4) {
23.        clearTimeout(timer);
24.        conststatus=xhr.status;
25.        if (status>=200&&status<300) {
26.          options.success&&options.success(xhr.responseText, xhr.responseXML);
27.        } else {
28.          options.error&&options.error(status);
29.        }
30.      }
31.    }
32.    //连接和发送-第二步if (options.type=='GET') {
33.      xhr.open('GET', options.url+'?' +params, true);
34.      //xhr.setRequestHeader("Accept-Encoding", "gzip");xhr.send(null);
35.    } elseif (options.type=='POST') {
```

```
35. xhr.open('POST', options.url, true);
36. //设置表单提交时的内容类型xhr.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
37. xhr.send(params);
38. }
39. }
40. //格式化参数letformatParams=function (data) {
41. letarr= [];
42. for (letnameindata) {
43. arr.push(encodeURIComponent(name)
    +'='+encodeURIComponent(data[name]));
44. }
45. //arr.push(('v=' + Math.random()).replace('.', ''));returnarr.join('&');
46. }
47. //时间戳转日期格式 letformatDateTime=function (time, format) {
48. constt= (time&&newDate(time)) ||newDate();
49. lettf=function (i) {
50. return (i<10?'0': '') +i
51. };
52. returnformat.replace(/YYYY|MM|DD|hh|mm|ss/g, function (a) {
53. switch (a) {
54. case'YYYY':
55. returntf(t.getFullYear());
56. break;
57. case'MM':
58. returntf(t.getMonth() +1);
59. break;
60. case'DD':
61. returntf(t.getDate());
62. break;
63. case'hh':
64. returntf(t.getHours());
65. break;
66. case'mm':
67. returntf(t.getMinutes());
68. break;
```



```
69. case'ss':
70. returntf(t.getSeconds());
71. break;
72. }
73. })
74. };
75. lethandleError=function (url, callback, _send) {
76. letscript=document.createElement('script');
77. script.type='text/javascript';
78. script.onload=script.onreadystatechange=function () {
79. if
    (!this.readyState||this.readyState==="loaded"||this.readyState==="complete")
    {
80. console.log('create script loaded : '+url);
81. callback&&callback();
82. _send&&_send();
83. script.onload=script.onreadystatechange=null;
84. }
85. };
86. script.onerror=function () {
87. console.log('create script error : '+url);
88. _send&&_send();
89. script.onload=null;
90. }
91. script.src=url;
92. oHead.appendChild(script);
93. }
94. //eval js字符串代码let_eval=function (fnString) {
95. window['eval'].call(window, fnString);
96. }
97. letautoDel=function () {
98. if (!_localStorage.setItem) {
99. return;
100.}
101.for (letkeyin _localStorage) {
102.//console.log('第'+ (i+1) +'条数据的键值为: ' + _localStorage.key(i) + ', 数据
```

```
为: ' + _localStorage.getItem(_localStorage.key(i)); //let key =
_localStorage.key(i);constvalue=key.indexOf('##') > -1?_localStorage.getItem(key) :
";
103.constisSetExpire=value.split('##').length>1;
104.constdate=isSetExpire&&value.split('##')[1];
105.constisExpire=date&&nowDateNum> +date.replace(/-/ig, '');
106.if (isExpire) {
107._localStorage.removeItem(key);
108._localStorage.removeItem(key+'_data');
109.console.log('DEL:'+key+'|'+key+'_data');
110.}
111.}
112.}
113.letonIndex=-1;
114.letsend=function (list, index) {
115.constnum=list.length;
116.if (!num||num<index+1) {
117.autoDel();
118.return;
119.}
120.if (index<=onIndex) {
121.return;
122.}
123.onIndex=index;
124.constitem_url=list[index].url;
125.constitem_aliases=list[index].aliases;
126.constcallback=list[index].callback;
127.constisStorage=list[index].storage!==false ;
128.// if (!_localStorage) { //  handleError(item_url, callback); //  send(list, index +
    1); //  return; // }constisDone=item_url===
    (_localStorage.getItem&&_localStorage.getItem(item_aliases));
129.if (isDone) {
130.constfnString=_localStorage.getItem(item_aliases+'_data');
131.try {
132._eval(fnString);
133.} catch (e) {
```

```
134.console.log('eval error');
135.}
136.callback&&callback();
137.console.log('local:'+item_aliases);
138.send(list, index+1);
139.return;
140.}
141.ajax({
142.url: item_url,
143.success: function (response, xml) {
144.//请求成功后执行try {
145._eval(response);
146.} catch (e) {
147.console.log('eval error');
148.}
149.//window['eval'].call(window, response);// ( window.execScript ||
    function( script ) { //    window[ 'eval' ].call( window,
        script );// } )( response );callback&&callback();
150.console.log('ajax:'+item_aliases);
151.constisSetExpire=item_url.split('##').length>1;
152.constdate=isSetExpire&&item_url.split('##')[1];
153.constisExpire=date&&nowDateNum>+date.replace(/-/ig, '');
154.if (isStorage&&_localStorage.setItem&&!isExpire) {
155.try {
156._localStorage.setItem(item_aliases, item_url);
157._localStorage.setItem(item_aliases+'_data', response);
158.} catch (oException) {
159.if (oException.name=='QuotaExceededError') {
160.console.log('超出本地存储限额! ');
161._localStorage.clear();
162._localStorage.setItem(item_aliases, item_url);
163._localStorage.setItem(item_aliases+'_data', response);
164.}
165.}
166.}
167.send(list, index+1);
```

```
168.},
169.error: function (status) {
170.//失败后执行console.log('ajax '+item_aliases+' error');
171.handleError(item_url.replace('2580', ''), callback, function () {
172.send(list, index+1);
173.});
174.setTimeout(function () {
175.send(list, index+1);
176.}, 300)
177.}
178.});
179.}
180.constnowDateNum=+formatDateTime(false, 'YYYYMMDD');
181.send(window.__page_static, 0);
182.})();
```

//一期实现: localStorage缓存/存储; //后续加强: PWA/indexeddb(待定); //后续加强: 引入前端静态资源版本diff算法, 局部更新本地版本;

### 3、使用方法:

#### ●4.1-配置静态资源别名

```
1. window.__page_static= [
2. {
3.   aliases: 'FocusEngine',
4.   url:'//g.alicdn.com/de/focus-engine/2.0.20/FocusEngine.min.js'
5. },
6. {
7.   aliases: 'alitv-h5-system',
8.   url: '//g.alicdn.com/de/alitv-h5-system/1.4.9/page/main/index-min.js',
9.   callback: ()=>{ window.Page.init() };
10. }
11. ]
```

#### ●4.2-引入种子文件:

```
1. <script charset="utf-8"  
   src="//g.alicdn.com/de/local_cache/0.0.1/page/localStorage/index-  
2. min.js"></script>
```

#### ●4.3、过期设置:

```
1. window.__page_static= [  
2. {  
3.   aliases: 'FocusEngine',  
4.   url:'//g.alicdn.com/de/focus-engine/2.0.20/FocusEngine.min.js##2018-08-10'  
5. },  
6. {  
7.   aliases: 'alitv-h5-system',  
8.   url: '//g.alicdn.com/de/alitv-h5-system/1.4.9/page/main/index-min.js##2018-  
   08-30',  
9.   callback: ()=>{ window.Page.init() };  
10. }  
11. ]
```

tips: 每次加载TVcache后, TVcache通过xhr方式请求得到js资源, 并自动根据“##”分割得到有效期:

- 1、当无“##日期”时, 则默认长期缓存;
- 2、当“##日期”大于请求日期时, 则请求会资源后, 正常存入local;
- 3、当“##日期”小于请求日期时, 则请求会资源后, 不存入local, 并在所有请求结束后, 检索local, 删除之;

#### ●4.4、禁用本地缓存:

```
1. window.__page_static= [  
2. {  
3.   aliases: 'FocusEngine',  
4.   url: '//g.alicdn.com/de/focus-engine/2.0.20/FocusEngine.min.js##2018-08-10',  
5.   storage: false  
6. },
```

```
7. {  
8.   aliases: 'alitr-h5-system',  
9.   url: '//g.alicdn.com/de/alitr-h5-system/1.4.9/page/main/index-min.js##2018-  
    08-30',  
10.  storage: false, callback: () => { window.Page.init() };  
11. }  
12. ]
```

#### ●4.5、删除本地缓存：

方法1：同别名文件，修改文件版本号如

前期布设：

```
1. window.__page_static = [  
2.   {  
3.     aliases: 'alitr-h5-system',  
4.     url: '//g.alicdn.com/de/alitr-h5-system/1.4.9/page/main/index-min.js'  
5.   }  
6. ]  
7. 更新布设：  
8. window.__page_static = [  
9.   {  
10.    aliases: 'alitr-h5-system',  
11.    url: '//g.alicdn.com/de/alitr-h5-system/1.4.10/page/main/index-min.js',  
12.   }  
13. ]
```

方法2：手动设置删除数组（不推荐）

`window.__page_static_del = ['alitr-h5-system']`

业务应用

OTT端酷喵APP-H5投放页

## 五、总结

我们上文提到，该方案将资源文件缓存到本地，之后项目页面再次请求时与本地版本号比对，之后进行决定是用本地缓存还是重新fetch。而在笔者所从事的OTT端HTML5业务，是将“焦点引擎文件”、“框架文件”、“公共组件文件”默认存储于本地，业务文件视情况配置。



经过该部署方案的投放，我们监测显示，冷启动平均提速30%，尤其在弱网情况下更加明显，而相应使load超时情况大大降低。

然而，技术总是在进步的，近一两年阿里集团内2C业务，H5逐渐被Weex/Rax技术栈取代，相应的在OTT端H5缓存的技术探索也失去了业务依托，当然这并不是结束。换句话说，这是新的探索的开始，在OTT端Rax能力创建建设中，我们也开启了新的缓存方案建设，并且取得了一定的成绩，后续笔者将继续行文介绍，Rax在OTT端的缓存建设。



【阿里巴巴文娱技术】  
公众号



阿里巴巴文娱技术  
钉钉交流群



阿里云开发者“藏经阁”  
海量电子书免费下载