阿里文娱技术 阿里云 开发者社区

覆盖全端业务的大前端技术

工程效能 | 多端同构 | 互动渲染

—— 阿里文娱技术精选系列 ——

大前端技术

关注我们

（阿里文娱技术公众号）

关注阿里技术

扫码关注「阿里技术」获取更多资讯

加入交流群

1）添加“文娱技术小助手”微信

2）注明您的手机号 / 公司 / 职位

3）小助手会拉您进群

By 阿里文娱技术品牌

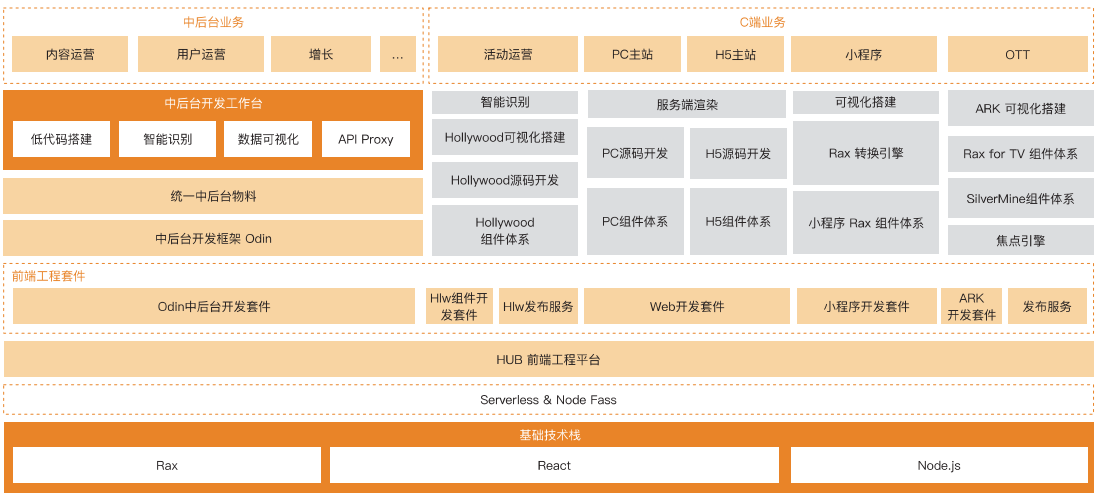
更多电子书

扫码获取更多技术电子书

目 录

序

前端技术一直在快速变化，回顾阿里技术从 PC 时代到 All in 无线，几年间新技术不断涌现又被快速替代。与此同时，优酷前端也在不断的进化迭代中，业务从 Web 站逐步扩展到移动端 APP、OTT、小程序，以及营销系统搭建和运营中后台建设。团队形态也从最初分布在多个业务的小型闭环，逐步集结，融合为面向整体业务横向支撑的“大”前端。



上图是优酷前端技术体系的概况，业务场景多、技术栈繁杂，对前端工程能力的要求不仅限于提效，我们基于阿里工程基础服务之上构建了面向文娱的工程平台，为不同业务技术栈和开发流程的收敛统一提供通用的工程能力；随着 Node.js 的发展，前端同学不只聚焦在端上的交互和展现，开始接管 Web 主站服务中间层，并且借助 Node.js 服务端渲染的优势，在优酷 Web 主站性能和体验上取得了跨越式的提升；活动营销对于页面高频大量的运营诉求驱动了页面生产方式从源码朝着搭建、智能识别等低代码方式演化；对于互动能力的需求，也催生着前端在渲染领域的发展，从 JS 驱动 HTML 结构到借助端渲染能力的 RN、Weex，再到逐步回归 W3C 子集的 Canvas，渲染侧解决问题的维度也在从研发效能、性能、展现能力不断交融变化。

我们将团队遇到的技术挑战以及解决过程做详细的展开，希望由解决方案的推演抽丝剥茧，一探优酷前端团队在支撑业务过程中的技术思考和沉淀，为读者带来一些启发。

阿里文娱高级前端技术专家 倪欧

2020.02.08

优酷前端技术：如何支撑营销活动？

作者| 阿里文娱前端技术专家 听鸿

优酷前端团队自 2017 年开始承担优酷营销活动的开发工作，到 2020 年初，已经有 90%以上的营销活动前端是由平台化、组件化的搭建方式来支撑。在支撑营销活动的进程中，技术方案也进行了几次迭代和建设，本文将把这些演变进行总结。

一、2017 年及以前：原生 JavaScript 支撑的 TV 营销活动，初识搭建的魅力

挑战：低性能的设备、不一样的交互方式

关键词：原生开发、焦点管理引擎、初识搭建

相对于手机端和 PC 端的前端页面开发，TV 端的开发有很大的不同：

首先是性能和兼容性，相比价格更高的手机和性能优秀的 PC 浏览器，200 元上下的网络电视机顶盒在硬件水平上有很大的差距；加上系统的高度定制，在某些特性上已经失去了较多的兼容性。因此，TV 端活动营销的开发方式在当时是比较原生的前端开发方式，无框架的约束可以让开发人员更底层更针对性的对页面进行优化，以便能达到较好的交互性能和兼容性。

其次：用户交互方式的不同使得 TV 端开发有额外的工作，其中最主要的不同就是对焦点的管理，相对于触屏交互和鼠标，TV 端的交互输入设备是遥控器，用户能做的操作是上、下、左、右、确定、返回（当然，现在的电视智能系统已经把语音控制作为更广泛的交互方式了）。对此，团队创造了一套通用的焦点引擎来统一对焦点进行管理及自动智能切换，该方案至今仍在部分 TV 端开发中应用。

开发方式确定&重要方案解决后，对于研发同学的效率提升已经有了很大的保证，但是活动的增加仍然需要大量的研发低价值投入，没有解决能力的复用，因此：“组件化搭建”便成了释放生产力的最重要的抓手，也是此时，团队正式初步开始构建属于我们自己的搭建系统。

“搭建”作为阿里巴巴集团前端委员会四大方向之一，而营销活动也是搭建最重要的落地业务方之一，以此为契机，团队顺势而为构建了最具特色的 TV 端活动搭建系统“ARK 系统”：将可复用的组件通过拖拽的方式放置到面板上，生成最终页面。

“搭建”系统的出现极大的解放了前端研发同学对低价值重复活动的开发投入，让技术同学能够更多的从可复用、可配置的方向去思考活动的实现，做到做一次任意用，在业务上也让玩法得到沉淀，让经验得到共享。

二、2017 双 11：引入框架的源码开发的移动端 H5 活动支撑

挑战：埋点、跳转、唤端

关键词：埋点规范化、跳转&唤端统一、开发规范

对于团队来说，2017 年是一个比较挑战的一年，团队从 TV 端营销活动开发开始承接优酷部分重要运营平台的开发工作，到 2017 年双 11，优酷第一次“真正意义”上的参与双 11：猫晚优酷独播、双 11 大促之后的免单返场营销活动等，第一次找到了优酷 X 天猫的玩转双 11 的更好的方式。

在支撑 2017 年双 11 营销活动的过程中，由于 React、Vue 等框架的崛起，团队在营销活动领域开始尝试性的引入部分框架来代替纯原生开发的方式，加上之后的优酷周年活动、2018 春节战役、三月战役等，框架在为团队的营销活动的支撑中做出了重要的贡献。几次战役打下来，团队成功积累了优酷移动端营销活动的基础能力如埋点规范、跳转唤端规范等，加上对集团搭建体系的进一步调研，我们也在继续寻求建设更适合优酷业务方向的移动端搭建技术解决方案和系统。

（每个活动顶部高度定制，但是首屏之后的内容导流区、售卖区是可以进行强复用的）

三、2018 世界杯战役：初具搭建能力的 Weex 页面

挑战：高时效性、高灵活性

关键词：CMS 数据分发+搭建、weex 云构建

2018 年俄罗斯世界杯对于优酷来说是一次意义重大的赛事，网络独播带来的大量流量需要有更加灵活、时效性更高、更新更快速的页面。因此基于优酷的 CMS 内容分发平台，针对特殊的“世界杯”频道，定制了基于 CMS 提供组件搭建能力和数据分发能力+云构建生成最终 weexbundle 的初具搭建能力的解决方案，该方案在一个月的世界杯期间稳定的支撑了“世界杯”频道页及部分其他页面的运营需求。组件化+分离运营+分离开发，为后续构建更适用于营销活动的搭建平台积累了丰富的经验。

四、2018 双 11：新搭建平台的第一次战役练兵

挑战：稳定性、规范

关键词：Weex 开发规范、看齐集团技术体系

基于世界杯战役中团队对 Weex 搭建的成功经验和积累的相关能力，世界杯战役之后，团队正式开始快速推进符合优酷特色的新的营销活动搭建平台“统一活动营销平台”的开发工作，经过 1 个多月的开发工作，于 2018 年 9 月正式上线，并在 10 月在站内推广，并在 2018 年双 11 战役中作为主要承接平台高效承接了双 11 的战役需求。新平台在基础能力上没有做太多重

复的建设，而是在集团搭建方案的服务之上针对优酷和文娱的特色，做了更多符合自身业务的定制。

之后在 2019 春节战役，2019 暑期战役中，统一活动营销平台一直作为营销活动的主力搭建平台，为高效高质的优酷营销活动服务。

五、2019 双 11：属于文娱的活动营销平台

挑战：提效

关键词：流程规范、精细化、智能化、文娱向

到 2019 年双 11 战役，经过一年迭代优化的统一活动营销平台”已经成为具备较为灵活的精细化投放能力、全方面解决多人共同运营、多端（Phone、PC、Pad）搭建能力、智能识别搭建能力等高效运营提效的平台，同时平台所支撑的业务也从优酷向文娱各业务方扩充，借助平台，让整个文娱的营销活动能力通、数据通、运营通。

以上便是关于优酷营销活动的关键节点和里程碑。回顾下来，这些方案的演进和落地还是有着必要的“刻意设计”，每年优酷的核心三战役是：春节战役、暑期战役、双 11 战役，基于这些将团队方案、平台进行提前设计，建设在平日、练兵在战役中，技术团队只有提前设想、落实计划、跟踪进度、确认结果，才能在一个个的时间点上完成应有的建设，拿到该拿到的结果。

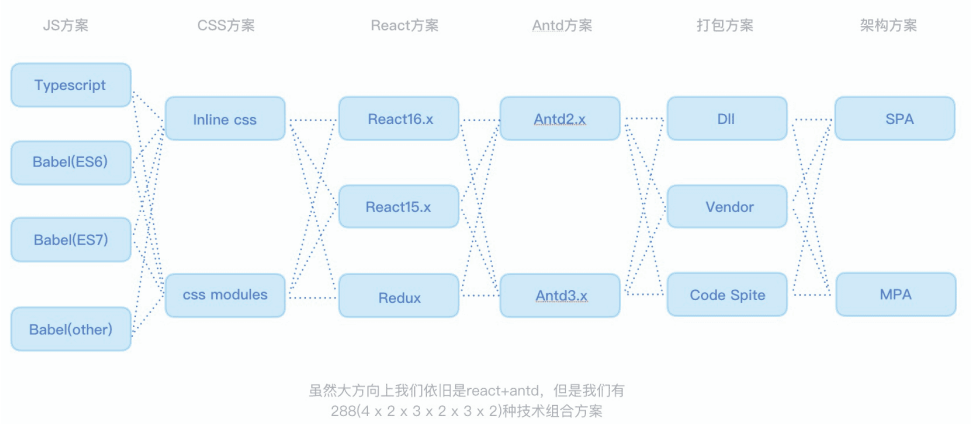
技术方案多、复用难？且看前端工程管理实践

作者| 阿里文娱前端技术专家 铉咏

一、背景

近两年，前端复杂度持续攀升，从框架到开发模式都衍生出了无数的技术方案。单点的小规模尝试，导致团队内部技术栈以及实现方案出现分化，间接造成了知识库之间的隔离、项目之间模块复用率下降，人员在不同项目中的学习成本大大增加，技术管理成本被大量转嫁到人员管理上。

例如，同一种平台因为技术方案管控问题，导致执行路径产生偏差，细节复用艰难。



那么，如何实现人与工程的规模化管理，并优雅的驱动项目？本文将详解阿里文娱的实践及思考，尤其是整合各业务方向、技术实现拉齐、能力复用方面。

二、我们该怎么做?

经过详细的技术及收益评估，我们决定在阿里基础前端工程服务的基础上，搭建文娱自己

的前端工程研发系统，用以高效承接及处理在前端工程领域上所遭遇的挑战。

1. 收敛&聚焦

大部分问题的诱因是缺乏统一的工具入口，而工具作为工程开发模式的重要指导急需统一的工具开发规范。

在此之上以“收敛&聚焦”为目标，我们开启了“终端+服务”的执行思路，落地了两个工具：

1）工具集成工具 Hub Cli：集成其他二次开发工具，提供统一的工具接入接口；

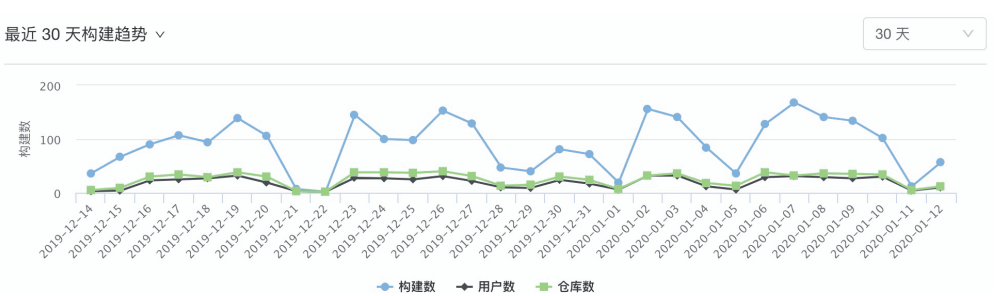
2）工具服务平台 Hub Service：提供工具依赖的基础服务。

2. 强化&赋能

通过工具服务平台对一些能力型问题进行针对性增强，如用户身份校验，从而将通用型能力赋能到各业务开发场景。

1）自动接入云构建

基于 Hub Cli 开发的工具默认支持开启云构建，采用阿里集团云构建方案，进行统一的数据采集。



2）用户识别

基于 Hub Cli 开发的工具自带用户识别接口，用以快速识别用户针对性下发配置。

3）工具灰度控制

传统工具版本控制基于 Npm 管理，无法快速的进行回退或者废弃版本；基于 Hub Cli 开发的工具，则可以通过工具服务平台平台，快速实现对人/系统/特定环境的快速灰度(对云构建同样生效)。

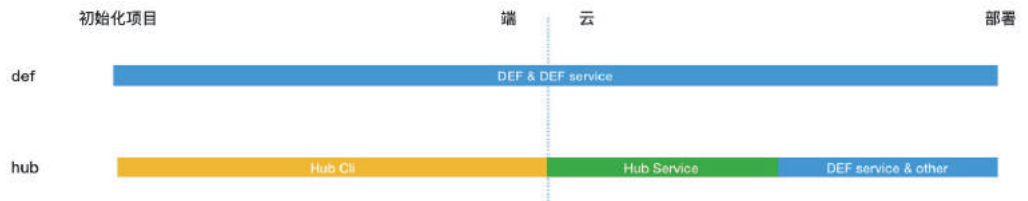
4）日志采集

基于 Hub Cli 开发的工具自动识别工具上报错误信息，帮助工具开发者快速定位工具问题。

id	类型	包名	信息	用户	系统信息	hub版本	操作时间	栈
6624	错误	@ali/hub-kit-hlw2	[31m请选择或填写必须的选项或值 [39m	晓玉	darwin/18.2.0 nodejs/v8.11.2	3.3.13	2020-01-12 22:49:42	详细信息
6623	错误	@ali/hub	connect ETIMEDOUT 106.11.209.164:443	言连	darwin/18.7.0 nodejs/v11.12.0	3.3.13	2020-01-12 22:13:17	详细信息
6622	错误	@ali/hub-kit-hlw2	[31m请选择或填写必须的选项或值 [39m	言连	darwin/18.7.0 nodejs/v11.12.0	3.3.13	2020-01-12 21:17:37	详细信息

三、为什么不直接使用集团前端工程工具

我们通过比较后决定下沉集团前端工程工具相关服务能力，以期在文娱层面达到更深的管理能力、灵活的方案定制能力。同时通过入口的收敛，降低各子团队对于集团基础前端工程服务的直接诉求，降低集团基础前端工程服务团队直接对接诸多一线业务团队的压力。



1）大部分工程诉求不再依赖集团基础前端工程服务团队支持，方案制定更加灵活；

2）对文娱前端工程状态首次有了可观测的可能；

3）系统化的工具研发方案，大大加强工具能使用的能力，比如用户信息的获取，比如更新控制；

4）自上而下的流程管控，便于随时增加工程卡口。

四、工程抽象

为了更好的实现一个工具集成平台，则需要对整个工程流程进行定义与抽象。

1. 工程生命周期

在工具入口层面定义了五个基础的工程生命周期，覆盖整个开发流程，降低工具间的学习成本，解决开发流程的规范问题。

1）初始化项目：hub init

2）将代码部署至日常：hub daily

3）讲代码部署至线上：hub publish

4）构建项目：hub build

5）启动开发服务：hub server

6）启动代码测试：hub test

2. 可编排流程

对于发布流程的抽象，可帮助工具开发者更好的定义工具用户的发布行为，下图展示了对与 Assets 发布流程中的流程定义，实现了对分支的自动维护；工具开发者可自行定义流程从而降低使用者的操作成本。

3. 流程钩子

从生产到完成发布，整条流程线上分为三部分，分别提供流程钩子，用于做中间状态的校验。钩子校验采取阻断式。

名称	钩子类型	备注	可用钩子数量(个)
Hub cli hook	端	终端工具的命令执行前后的钩子	6
Git hook	端	git 相关操作的钩子	18
Hub service hook	云	云端流程相关钩子	6

4. 质量卡口

通过增加生产发布前的强制卡口。

五、流程管控

1. 工程方案下放

Hub Cli 会在执行命令时启动，启动会经过一轮 Check 流程，包含两部分内容更新:

1）主程序更新检测

2）工具更新检测

当发现存在新版本时，会通过 Hub Cli 内置更新模块进行自动更新，确保所有模块的运行版本为最新版本。

更新流程为强制更新，不可手动关闭是出于对工程方案覆盖率的考虑。及时覆盖就意味着版本断层的问题会被削弱，集中更新，集中处理。

2. 自动化发布流程

以改动提交场景为例，传统流程操作下，至少需要三步操作：

```
$ git add .
$ git commit -m '提交信息'
$ git push origin daily/0.0.1
```

步骤越多，出错的可能性越高。我们通过流程编排，将操作完全自动化:

```
$ hub daily -q
```

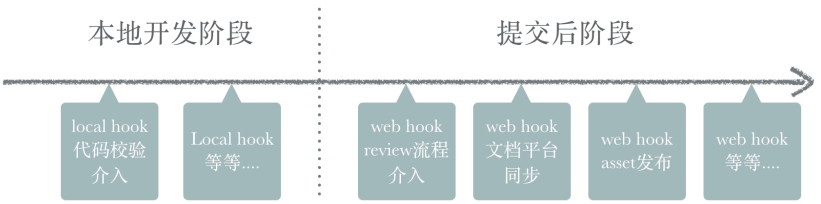
3. Commit 规范化

通过内置的 commit 管理模块，简化的同时取规范化 commit 提交数据，实现开发数据的有效沉淀，培养技术同学的优质编码习惯。

```
~/工作/阿里巴巴/Gitlab.Alibaba-Inc/中后台/youku-interactive-video-platform  daily/0.0.26 ●  hub daily
Hub 3.3.13 - Project: git
[Kit] 当前项目关联套件信息：odin(开发版)
? 请选择当前Commit类型 (Use arrow keys)
  feat     : 新功能 (feature)
  fix      : 修补bug
  docs     : 文档 (documentation) 相关的改动
  style    : 对代码的格式化改动，代码逻辑并未产生任何变化
  refactor : 重构代码或其他优化举措
  test     : 增加测试
❯ chore    : 构建过程或辅助工具的变动
```

4. 平台联动

基于钩子的多平台联动，通过 hook 设置的方式，实现发布后对其他平台的调用通知。

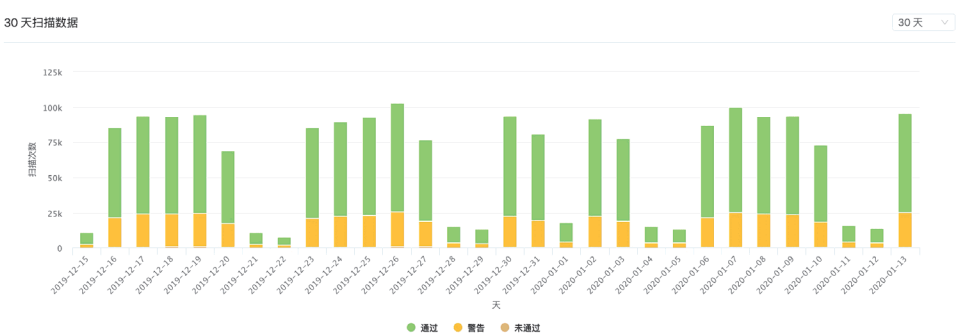


5. 代码质量检测&Code Review

使用 Hub 自动接入发布系统的项目将默认开启以下门神检测插件：

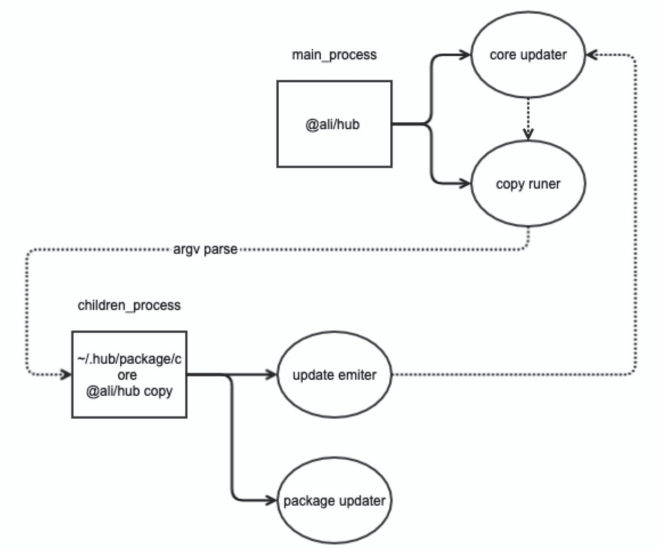
- HTTPS 协议检查
- 文件元信息检查
- 内部域名检查
- 代码注释检查
- NPM 模块 License 检查

发布的代码需要经过自动校验通过后才可发布至生产环境。除开自动检测外，还可以选择打开 Code Review 流程，只有经过 Code Review 流程后才可进行发布至生产的操作。



六、特定场景

1. 自动化更新



工具多是全局安装，因为全局环境权限问题，导致全局工具不能自更新，所以目前主流终端方案是只进行版本检查提示更新。

这种方案虽然解决了新版本通知的问题，但是弊端是依旧要人主动更新。

为此我们设计了一套基于全局环境的更新流程。主程序启动过程中会进入一个预先检查环节，预检查过程中会先检测存在的程序副本，若副本存在则启动子进程执行副本，而副本中会先进行自生版本检查，如果存在版本更新则通知主进程进行副本版本更新，当自身版本检查通过后，才会进入程序正式执行阶段。

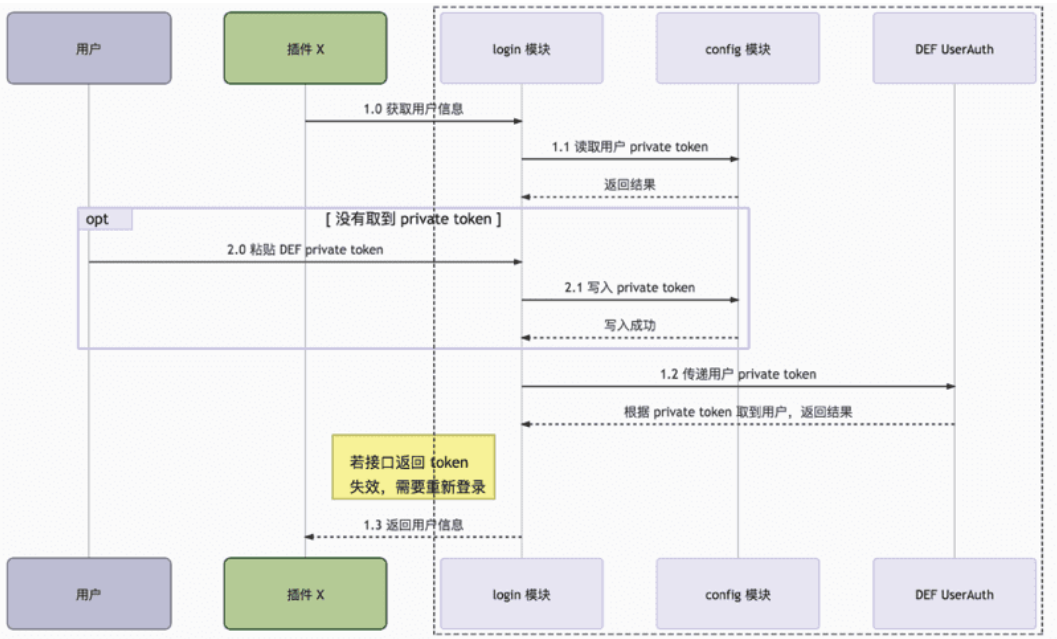
通过对副本储存目录的控制，绕过了全局的权限校验，实现了任意环境下的自更新操作。

2. 模板引擎

对于模板引擎，我们简化了相关设计，基于 Gitlab 托管，结合 Ejs 模板引擎，实现了一套轻量化的模板引擎系统。

3. 终端用户识别

基于阿里基础前端工程服务的终端授权模式，实现了 hub 的终端用户授权能力，并实现了与阿里基础前端工程服务登录状态的完全拉通。



4. 千项（目）千面

通过对项目内配置文件.hubrc 的识别，自动挂载对应的工具；根据工具的不同改变 Hub Cli 的能力。

~/ 目录下 hubr -h结果

```
~ hubr -h
hubr 3.0.9 - no active project

Usage:
  hubr [command] <args> [options]

Options for me hearties!
  --help, -h      获取帮助
  --version, -v   版本信息
```

~/…/youku-material 目录下 hubr -h结果

```
~/工作/阿里巴巴/Gitlab.Alibaba-Inc/youku-material  daily/0.0.9  hubr -h
hubr 3.0.9 - project: youku-metarial (silk:app)

Usage:
  hubr [command] <args> [options]

Choose yer command:
  hubr template <name>   创建子模板
  hubr create <name>     创建子模块(只用于应用类项目)
  hubr snippet [code]    获取代码片段到粘贴板
  hubr dll               创建 Vendor.dll.js
  hubr demo              创建规范站点用Demo
  hubr readme            创建规范站点用Readme
```

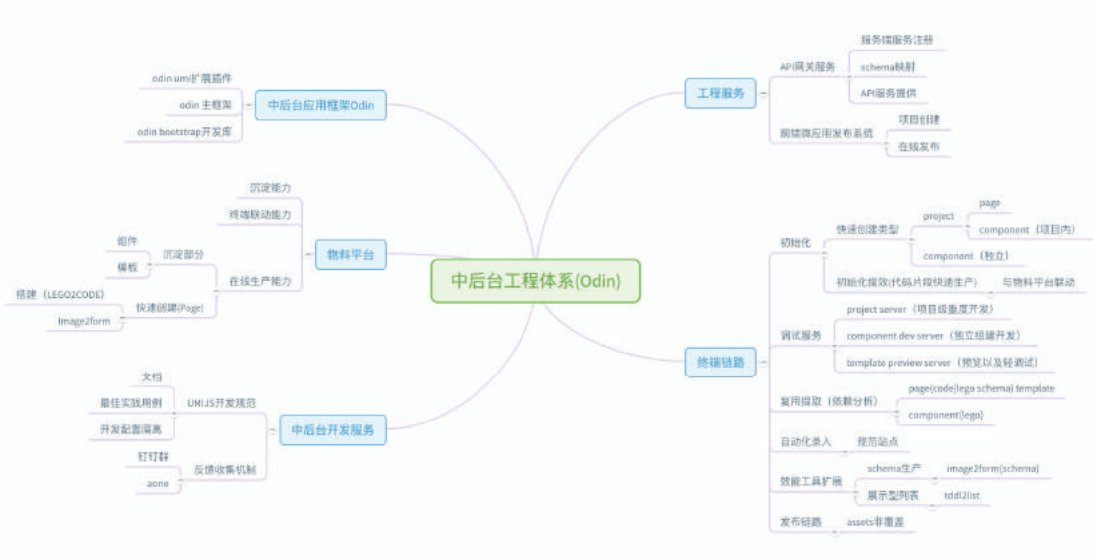
七、垂线领域沉淀

通过基础工程平台的沉淀赋能特定领域工程方案，加速特定领域研发效率，让工具开发者更专注于工程需求本身以及领域内的工程诉求。

1. 中后台场景

基础平台完善后，中后台技术方案开始聚焦，开始系统化设计/解决中后台场景下的提效问题。

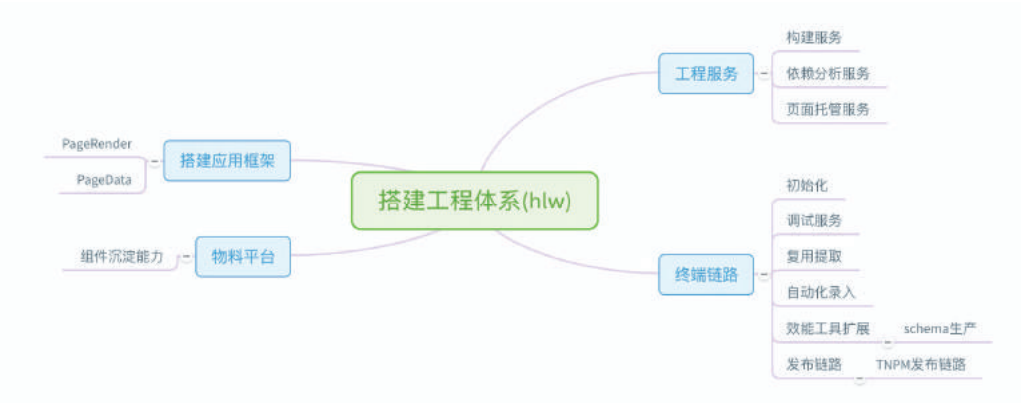
- 物料平台（用于沉淀通用型物料）
- 中后台研发中心（用于沉淀开发/设计规范）
- API 网关服务（用于前后端解耦，接口快速编排）
- 应用托管平台（用于联动发布链路自动化部署）



2. 搭建平台场景

文娱活动运营搭建平台的工程诉求基于 Hub 孵化出的工程体系。

- 物料平台（用于沉淀通用型物料）
- 依赖分析服务（用于在线搭建动态渲染）
- 构建服务（用于平台特殊场景部署下的动态构建）



八、经验复用

1. 采集：将在日常开发中产生的错误收集起来，并统一解答归类存档。即可自动归类生成一个错误的知识库。

2. 消费：在错误发生时即可从错误的知识库中查找出对应结果处理。

九、工程数据化（图形化示例）

通过流程收敛，使得整个团队的工程状态可以被监控，让数据的沉淀产生价值，优化/指导工程迭代，让每个技术人的努力可以被量化。

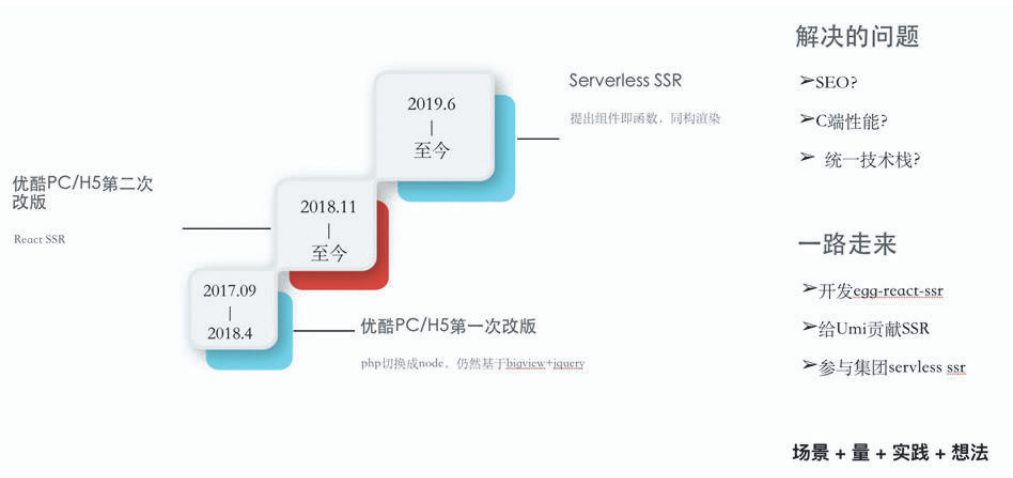
十、小结

上面是我们对前端工程如何规模化管理的思考与实践，整体来说就是工程入口的收敛&管控，加上领域方案的优胜劣汰，通过领域内经验的横向复用实现整体前端工程能力的提效。

这！就是优酷 Node.js 重构之路

作者| 阿里文娱前端技术专家 狼叔

在 2017 年底，优酷只有 Passport 和土豆的部分页面用 Node.js，PC 和 H5 核心页面还都是 PHP 模板渲染。而最近 2 年，基于阿里巴巴的技术体系，我们对 PC、H5 多端进行了技术改造。在 2019 年，React SSR 第一次且成功地扛起双 11 重任，具有一定意义。



本文将对这一技术演进之路进行总结和展望，有 2 点突出变化：

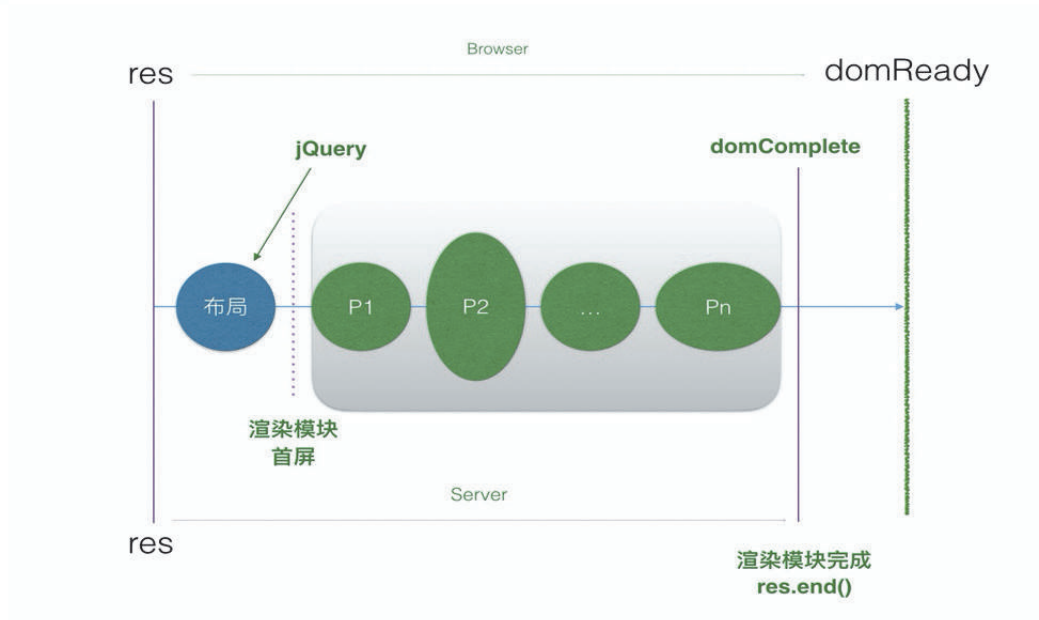
1）我们将优酷 C 端核心页面全部用 Node 重写，完成了 PHP 到 Node.js 的迁移。在没有 PHP 同学的情况下，前端可以支撑业务；

2）从 Bigpipe+JQuery 到 React SSR，性能明显提升。PC 页面首屏渲染降到 150ms、播放器起播时间从 4.6 秒优化到 2 秒。H5 站上了 React SSR 后，性能提升 3 倍。

一、演进之路

优酷前端团队在之前 PC 首页/频道页面改造的基础上，将 React 服务端渲染沉淀出了一个技术框架，该项目已经在 Github 开源，截止 2020 年 1 月底已有 657 star，具体使用方式和源码请查看 egg-react-ssr 项目地址（https://github.com/ykfe/egg-react-ssr）

第一版：Bigpipe + JQuery



1. 原理：

核心还是采用 Bigpipe + Nunjucks 模板编译实现的。模板 tpl 和 data 编译，生成 html，这个部分其实 Biglet 的 render 方法里做的。然后在请求写入的生命周期上，将 html 分块写入浏览器。

1）采用 Bigpipe 有 2 大核心优势：

- 兼容 IE6 等低版本浏览器，不得不用。现代 Web 框架只支持主流浏览器。之前开发对 JQuery 都比较熟悉，二者结合是当时最完美的选择；
- 业务属性决定的，新老接口拼凑（PHP 的，Java 的，HTTP 的），采用 Bigpipe，利用 Node.js 并发，将接口请求从前端转移到服务端，即可以保证业务快速迭代，也可以保证页面性能；

将页面进一步抽象，以前理解是每个模块都 Biglet。但实际使用中，进行分级处理会更好。

- 首先渲染布局；
- 核心：渲染首屏，对于播放页重点是播放器和剧集内容；
- 其他：看不到的可以偷偷加载，及时不展示也没问题；

2）性能

PageSpeed	起播时间	JS 总大小	Request 数量	首屏渲染时间	资源加载结束时间
50	2.4s	885kb	173	180 ms	12.8s

2. 当前：React SSR

采用 React SSR，是在 Node.js Bigpipe 做了一些之后才考虑的。稳定性有了，低版本浏览器兼容有了，接下来就要考虑团队成长和架构升级的问题。

- 会 JQuery，不会 React，大家是非常想用想学的；
- Bigpipe 虽好，但依赖模板编译，在内存和 CPU 使用上非常高，优化不易；
- 老系统慢慢下掉，之前混着新老接口一起用的方式也需要重新梳理。为了快速迭代而产生的“狗粮”还是要吃的；

其实，如果把 React 当模板，结合 Bigpipe，可以达到和之前解决方案一样的效果。问题不能很好的利用 React 的优势，React 自身也提供 SSR 基本能力。所以，我们需要做的是如何结合 React 的 SSR，构建出符合业务特点的 SSR 框架。

3. 组件级同构：支持 CSR/SSR 一键切换

在技术调研期，我们分别看了 next.js 和 easywebpack。

- easywebpack 在使用上，我不认可使用 egg 的 worker 来做。easy-team 的方案本地开发采用了在 Node 中启动 webpack 编译 bundle 的方案，将服务端 bundle 打包在内存中，agent 进程通过 memory-fs 提供的 api 来读取文件内容，并且通过 worker 与 agent 进程的通信，来让 worker 进程可以获取到文件的字符串内容。然后采用了 Node 的 runInNewContext api，类似于 eval 的方式去执行 js 字符；
- next.js 写法上是完美的。结合上面讲过的 Biglet，请求方法抽取成静态方法，可以复用。另外在服务端渲染，先执行请求方法，是最高效的方式；

写法上，最终定了采用 next.js 式的写法。

该技术方案具有以下优点:

- 实现方式简单优雅
- SSR/CSR 无缝切换
- 全面拥抱 JSX
- 拥抱阿里生态

我们的代码既可以在服务端运行又可以在客户端运行。这里需要重点讲一下 CSR 和 SSR，我们是业内第一个这样做的。它也带给我们一个意外惊喜，那就是在服务降级上提供了一个更好的方案。

播放页的服务支持两种渲染方式，当服务端有渲染压力时或集群不稳定时，可以切换为客户端渲染，服务端只负责页面的 SEO 数据获取，渲染工作交给浏览器。另外一种方式是当数据源服务出现问题时，可以使用 CDN 数据兜底进行页面容灾。

4. 组件: 多模块升级配合改造

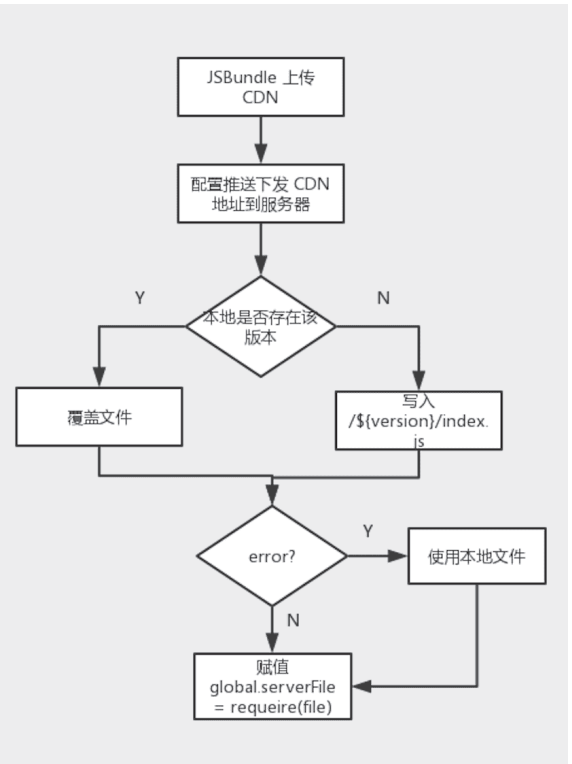
播放页的页面组件包括首屏、分发区、弹幕、播放器、评论等等一系列的组件，此次播放页升级对应的组件也做了重构，其中弹幕、评论已随新版播放页一起灰度上线。

5. 服务端 JSBundle 发布: 变更无需发布应用

在 SSR 改造中我们总结出了一套基于配置下发的服务端和客户端文件的流程。该流程有以下优点:

- 构建方式一致
- 发布方式一致
- 服务端 bundle 变化无需走服务端发布
- 及时生效

下面是发布的流程图:



6. 性能对比

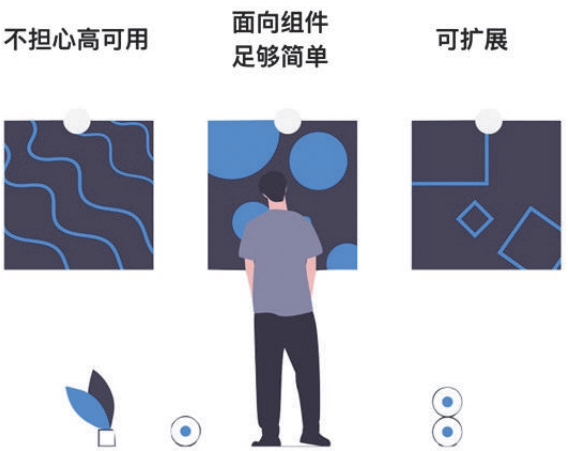
以预发版本播放页为例，使用 chrome 无痕模式测试。

优酷 PC 播放页：150ms 首屏，首屏文档大小 37kb。文档到达时间平均为 120ms。

优酷 H5 播放页性能提升 3 倍。

二、未来：Serverless SSR

从 beidou、next.js、egg-react-ssr 到 Umi SSR，可以看出服务端渲染是很重要的端侧渲染组成部分。无论如何，React SSR 都是依赖 Node.js Web 应用的。那么，在 Serverless 时代，基于函数即服务（Functions as a Service，简写为 FaaS）做 API 开发相关是非常简单的：1）无服务，不需要管运维工作；2）代码只关系函数粒度，面向 API 变成，降低构建复杂度；3）可扩展。



在 FaaS 下，如何做好渲染层呢？直出 html，做 CSR 很明显是太简单了，其实我们可以做的更多，Serverless 时代的渲染层具有如下特点：

- 采用 next.js/egg-react-ssr 写法，实现客户端渲染和服务端渲染统一；
- 采用 Umi SSR 构建，生成独立 umi.server.js 做法，做到渲染；
- 采用 Umi 做法，内置 webpack 和 React，简化开发，只有在构建时区分客户端渲染和服务端渲染，做好和 CDN 配合，做好优雅降级，保证稳定性；
- 结合 FaaS API，做好渲染集成。

通过提供 ctx.ssrRender 方法，读取 dist 目录下的 Page.server.js 完成服务端渲染。

核心要点：

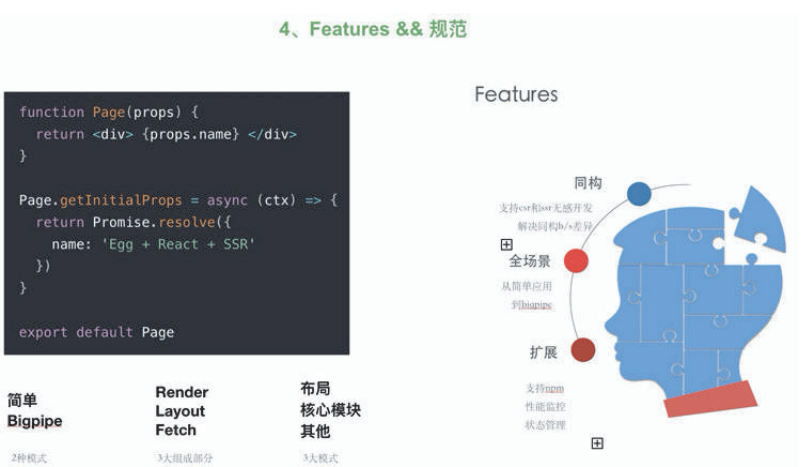
- ssrRender 方法比较容易实现；
- 采用类似 Umi SSR 的方式，将源码打包到 Page.server.js 文件中；
- 在发布的时候，将配置，app.server 函数和 Page.server.js 等文件上传到 Serverless 运行环境即可。

综上所述，我们大致可以推出架构升级的 4 个阶段。

- 在 CSR 中，开发者需要关心 React 和 webpack；
- 在 SSR 中，开发者需要关心 React、webpack 和 egg.js；
- 在 Umi SSR 同构中，开发者需要关心 React 和 egg.js，由于 Umi 内置了 webpack，开发者基本不需要关注 webpack；
- 在未来，基于 FaaS 的渲染层，开发者需要关心 React，不需要关心 webpack 和 egg.js。

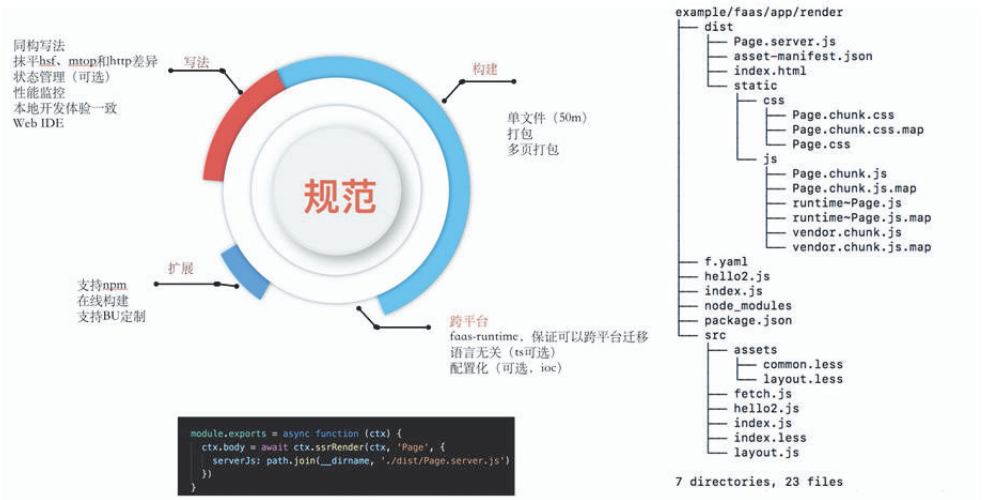
在这 4 个阶段中，依次出现了 CSR 和 SSR，之后在同构实践中，对开发者要求更高，甚至是全栈。所有这些经验和最佳实践的积累，沉淀出了更简单的开发方式，在未来 Serverless 环境下，希望前端更加简单、高效。

1. 规范



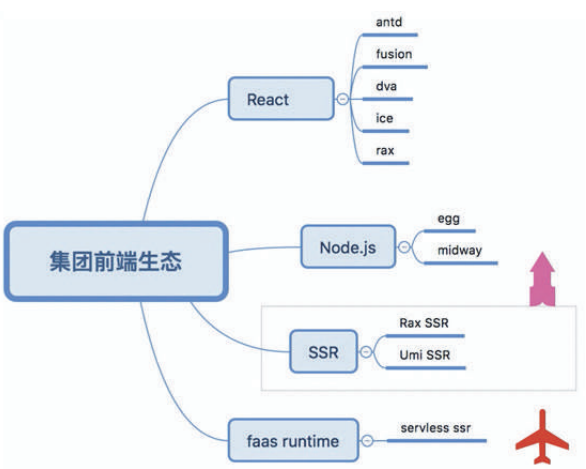
上图的最小示例代码中描述了一个页面如何进行数据获取和展示，其中 render(必选)、getInitialProps(可选) 均为方法，在 SSR 模式下页面数据传输模式可以为简单模式和 Bigpipe 模式。

2. 实现



实现上基于 FaaS 规范，使用 React 同构写法，构建后的目录结构如上图，最终通过调用 ssrRender 方法进行服务端渲染，将页面 HTML 返回客户端。

3. 生态



目前集团沉淀的前端生态已经被开发者大量使用，从 React 的组件库、框架、多端方案，到 Node.js 的 WEB 框架，相信在未来的 Serverless 环境下，会有更多优秀的方案或框架加入到集团生态，使前端开发更加的快捷、高效。

三、致敬所有多端坚守者

感谢苹果，将用户体验提升到了前无古人的位置。移动互联网兴起后，PC Web 日渐没落。在 AI 时代，没有“端”的支持可以么？明显是不可以的。不只是“端”的概念，而是真真正的用户体验。

1）我们可以利用 PC/H5 快速发版本的优势，快速验证 AI 算法，继而为移动端提供更好的模型和数据上的支撑；

2）多端对齐，打好组合拳。既然不能在移动端有更大的突破，大家只能在细节上血拼。按照木桶原理，哪个地方是短板，整体水位就在那里。

今天的大前端，除了 Web 外，还包括各种端，比如移动端、OTT，甚至是新的物联网设备。我们有理由相信 Chrome OS 当年的远见：“给我一个浏览器，我就能给你一个世界”。

OTT 端登录态设备穿透：扫码登录与反登录

作者| 阿里文娱高级前端技术开发工程师 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“私有-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。

但是到了 OTT 大屏电视场景，扫码登录是如何实现的呢？本文将以优酷“CIBN 酷喵影视”为例，详解 OTT 大屏中的“扫码登录“与“扫码反登”的技术实现策略。

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”。

OTT端

手机端

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：

1）登录：OTT 端酷喵应用登录二维码；

2）客服：各种业务类型客服小蜜二维码；

3）互动：OTT 端活动投放，与小屏配合互动-扫码；

4）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，那么我们先对两种同步登录态实现逻辑进行解读。

1. 扫码正登录

早在数年前已经普及，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：一个光秃秃的二维码，是如何完成识别被哪个手机扫码，又是通过什么逻辑实现 OTT 端/PC 端登录的？下面我们进行逻辑拆解：

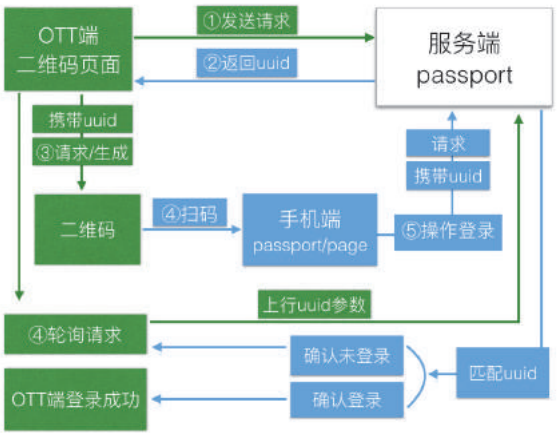
1）Tips

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。

2）原理图：



3）逻辑解读

①当二维码扫码登录页面被打开时，OTT 端会想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；

④现在登录状态轮询已开启，而二维码则开始等待手机扫码；

⑤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，服务端跟据 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；

⑥在以上④-1 中已经提到，OTT 端一直在轮询接口登录状态，在完成上文第⑤后，轮询接口再次携带 uuid 请求 passport 服务端，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⑤项中生成的 token 一并返回。

至此六大过程形成闭环，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。

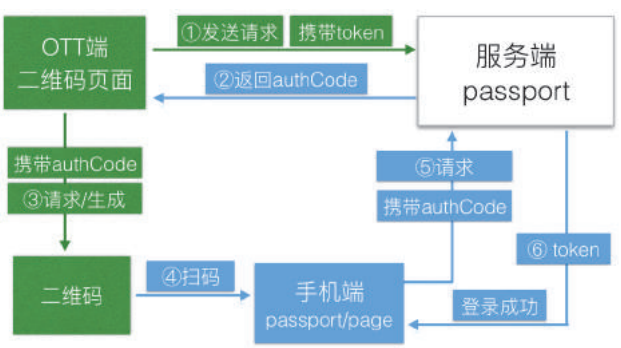
1）Tips

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

2）原理图：



3）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。

四、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，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。

五、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。

小而美的 egg-react-ssr 开源实现方案

作者| 阿里文娱前端工程师 十忆

本文基于优酷 PC/H5 业务改造的经验心得，提炼出了一套基础的 egg-react-ssr 服务端渲染应用开发的基础骨架。为了更好的对比 Next.js 以及市面上等其他各种实现方案，我提炼出了比较精华的几点优势。

- 小：实现方式简洁；
- 全：功能齐全，配套结合多种热门模块的 example；
- 美：基于 React 和 Eggjs 框架，拥有强大的插件生态，配置非黑盒，非常方便加入业务个性化逻辑。

项目地址：https://github.com/ykfe/egg-react-ssr

一、快速开始

这里我们提供了一个脚手架，方便你创建快速项目。

```
$ npm install yk-cli -g
$ ykcli init <Your Project Name>
$ cd <Your Project Name>
$ npm i
$ npm start
$ open http://localhost:7001
```

二、特色

本应用的特色是实现方式简单优雅，相比于 Next.js 的代码非黑盒，让你清晰的了解 SSR(Server Side Render)服务端渲染应用的执行流程，并且方便加入业务自定义需求。如果你需要更直接地控制应用程序的结构，Next.js Nuxt.js 并不适合这种使用场景。同时本应用最大的特

色是可以实现 SSR/CSR（Client Side Render）的无缝切换，只需要通过修改配置文件，或者通过 Query 参数的方式，在线上应用流量过大或者服务端渲染过程中出现错误时，迅速降级为客户端渲染。

在实现方案上，我们抛弃了传统的模版引擎的做法，页面的一切元素包括基础 Html 骨架皆采用 jsx 的方式来编译生成而不需要使用 react-helmet 这种额外的库。

同时我们正在接入集团的 Ginkgo Faas Runtime，之后就可以使用 Serverless 的方式来开发 SSR 应用。蚂蚁的前端框架 umi.js 与我们的实现方式类似。

综上：我们自己用 React 提供的 api 去实现了一套简洁优美全面的 SSR 方案，并且核心代码十分简洁，主要配置皆暴露出来，这样构建工程的思路更清晰些，我们尽可能避免了你去学习新的轮子新的概念，一切功能都用社区现有的热门模块来完成。

本项目已经过很多外部开发者使用以及内部其他部门使用，反馈开发体验好过 Next.js,Nuxt.js 等方案很多。

三、组件编写方式

我们遵循 Next.js 的规范，在页面级组件上定义静态方法 getInitialProps 来做页面初始化的数据获取操作。在首次访问页面以及前端路由切换时会调用该方法。

```
Page.getInitialProps = async (ctx) => {
  return Promise.resolve({
    name: 'Egg + React + SSR'
})
```

四、如何兼容两种渲染模式?

SSR 的详细原理和执行机制在官方文档中都有详细解释，大家比较感兴趣的可能是，如何实现 SSR/CSR 无缝切换。在本地开发时，你可以同时启动两种渲染模式来观察区别。在生产环境时，可以通过 config 配置文件或者 URL Query 参数，来随时切换两种渲染模式

1. 实现原理

下面来介绍我们的详细做法，我们使用 JSX 来编写前端组件同时作为 CSR 模式下的页面骨架的生成来源，抛弃 Html 文件以及模版引擎。

2. 为什么选用 JSX?

更加详细的对比可以查看下列文章 https://medium.freecodecamp.com/angular-2-versus-react-there-will-be-blood-66595faafd51#.v4y4euy1r

中文翻译：https://zhuanlan.zhihu.com/p/20549104?refer=FrontendMagazine

3. 编写 layout

```
const Layout = (props) => {
  if (__isBrowser__) {
    return commonNode(props)
  } else {
    const { serverData } = props.layoutData
    const { injectCss, injectScript } = props.layoutData.app.config
    return (
      <html lang='en'>
        <head>
          <meta charSet='utf-8' />
          <meta name='viewport' content='width=device-width, initial-scale=1, shrink-to-fit=no' />
          <meta name='theme-color' content='#000000' />
          <title>React App</title>
          {
            injectCss && injectCss.map(item => <link rel='stylesheet' href={item} key={item} />)
          }
        </head>
        <body>
          <div id='app'>{ commonNode(props) }</div>
          {
            serverData && <script dangerouslySetInnerHTML={{
              __html: `window.__USE_SSR__=true; window.__INITIAL_DATA__ =${serialize(serverData)}`
            }} />
          }
          <div dangerouslySetInnerHTML={{
            __html: injectScript && injectScript.join('')
          }} />
        </body>
      </html>
    )
  }
}
```

我们可以直接使用 JSX 来编写我们的 Meta 标签。同时插入我们在配置文件中配置的脚本文件和样式文件并且注入服务端预取的数据给浏览器。

4. 编译 layout

我们提前将 layout 组件打包为可以在 Node.js 环境中调用的模块。这样我们运行的时候就不需要动态使用 Webpack 来进行编译。

```
module.exports = merge(baseConfig, {
  devtool: isDev ? 'eval-source-map' : '',
  entry: {
    Page: paths.entry,
    Layout: paths.layout
  },
  target: 'node',
  externals: nodeExternals({
    whitelist: /\.(css|less|sass|scss)$/
  }),
  output: {
    path: paths.appBuild,
    publicPath: '/',
    filename: '[name].server.js',
    libraryTarget: 'commonjs2'
  },
  plugins: plugins
})
```

5. 以 SSR 模式渲染页面

SSR 模式下我们可以直接渲染包含子组件的 layout 组件即可以获取到完整的页面结构。

```
const serverRender = async (ctx) => {
  // 服务端渲染 根据ctx.path获取请求的具体组件，调用getInitialProps并渲染
  const ActiveComponent = getComponent(Routes, ctx.path)()
  const Layout = ActiveComponent.Layout || defaultLayout
  const serverData = ActiveComponent.getInitialProps ? await ActiveComponent.getInitialProps(ctx) : {}
  ctx.serverData = serverData
  return <StaticRouter location={ctx.req.url} context={serverData}>
    <Layout layoutData={ctx}>
      <ActiveComponent {...serverData} />
    </Layout>
  </StaticRouter>
}
```

6. CSR 模式

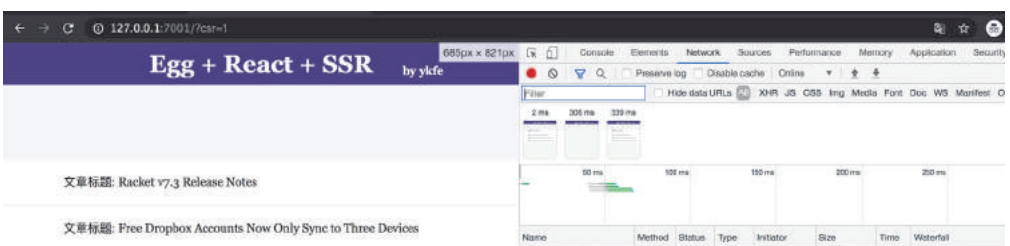
为了应对大流量或者 SSR 应用执行错误，需要紧急切换到 CSR 渲染模式下，我们可以通过 config.type 来控制。实现方式如下：

```
if (config.type !== 'ssr' || csr) {
  const renderLayout = require('yk-cli/lib/renderLayout').default
  const str = await renderLayout(ctx, config)
  return str
}
```

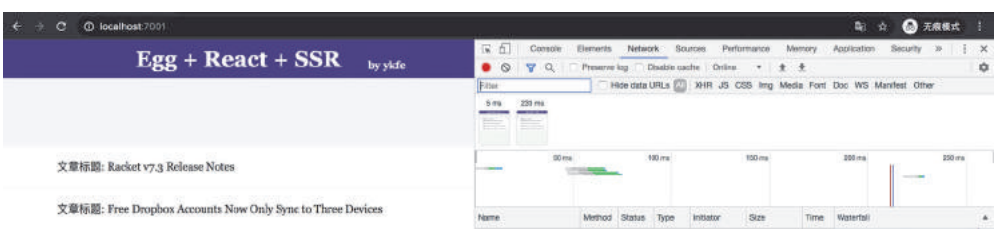
```
const renderLayout = async (ctx: any, config = {}) => {
  const dist = process.env.FE_BUILD || 'dist'
  const layoutPath = resolve(cwd, `./${dist}/Layout.server.js`)
  if (isDev) {
    delete require.cache[layoutPath]
  }
  const Layout = require(layoutPath).default

  const props = ctx ? {
    layoutData: ctx
  } : {
    layoutData: {
      app: {
        config: config
      }
    }
  }

  const str = reactToStream(Layout, props)
  return str
}
```

在非 SSR 渲染模式下，服务端直接返回一个只包含空的 </app> 的 Html 文档

7. 效果展示



CSR 页面首屏时间为 339ms



SSR 页面首屏时间为 233ms

五、取得的开源成果

https://github.com/ykfe/egg-react-ssr

- 使用公司已知 10+个，集团内部包括：优酷视频 、Vmate 短视频、阿里影业、口碑、新零售。外部公司包括：火炽星原 CRM、牛牛搭、希沃帮助中心、腾讯微卡、微脉；
- 18 个贡献者；
- PR(Pull Request)共 85 个，外部有效 PR 在 5 个左右（比如 mpa、bug 修复），文档等小修改较多；
- 使用 jest 做单测，NightWatch 做 e2e，使用 Circleci 做 ci 集成、测试覆盖率 100%；
- 大型外部分享会 1 次；
- 口碑较好，用过的都说比 Next.js 好。

整体

- 代码质量有提升，完善了测试，Bug 有降低，很多没有考虑到的 case 都被完善了；
- 由此衍生出 Serverless-Side-Render。结合当下火热的 Serverless 技术，按照 yml 文件中对于页面渲染层的规范，来轻易的开发并部署出一个具有 SSR/CSR 功能互相切换的应用。

双 11 猫晚互动前端容器化之路

作者| 阿里文娱前端专家 沐南

一、猫晚简介

天猫双 11 狂欢夜，以下简称“猫晚”。晚会有丰富的节目以及多样的互动，用户可以在互动中得到双 11 所需的购物权益。在这三年中，互动也从简单的 H5 互动逐渐演变成为依赖 Weex、H5、游戏引擎的多样互动。

二、互动技术挑战

1. 多平台同时直播

2019 年猫晚在优酷、淘宝、天猫同时播出，同时附带了 13 种互动，其中不乏“叠叠乐”等流程复杂的游戏。开发会面临两个问题：

a）渠道多、平台多。如何保证多平台开发效率且体验一致？

b）互动类型多，每个互动都有独立逻辑，直播间需要将所有互动做串联。这些看似独立又有关联的互动，如何能高效、健壮地落地在不同平台呢？

2. 年年创新、迭代速度快

我们需要在既定的时间内，实现快速迭代，并且要保证用户的良好体验。那么又会碰到一些问题：

1）迭代速度快。开发期只有几个月，客户端发版节奏无法满足。如何处理“发版”“快速迭代”这两个互相冲突的要求？

2）PC 端、H5 如何跟上移动端的脚步？

3. 如丝般顺滑的体验

“双 11 天猫狂欢节”的品牌有多响亮，“猫晚”同时在线人数就有多高。人数多就等于“机型多”，每年做猫晚几乎什么型号的手机都能碰到。

从服务端的角度来看，高同时在线就意味着服务端需要承担更高的 QPS，部署更多的机器。又会引出几个问题：

a）同时在线高的情况下，如何保证每个用户的体验？

b）高并发下如何保证服务稳定？

三、技术策略

为了更好的解决上述问题，我们提出了组件化、容器化方案。同时针对一些用户体验，从技术方面进行了优化。

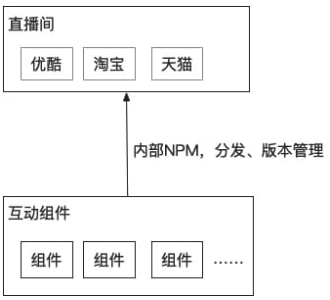
1. 组件化

需要在优酷、淘宝、天猫至少三个平台同时落地互动，的前提下如何保证同时进行多个平台开发的效率呢？

1）互动组件化

经过需求分析发现，每种互动其实是个较为封闭的环，每种互动都在内部有套独立的逻辑。

那是不是可以使用“组件化”的思想把不同互动拆分成“互动组件”然后分配出去，让每个开发只负责某一种或几种互动，通过内部 NPM 分发，在不同平台统一依赖这些“互动组件”，最终组装成直播间。



2）抹平平台差异

组件化要想同时运行在多个平台上，就要处理平台差异，有两种选择：

a）组件适配不同平台；

b）统一标准，平台去兼容组件。

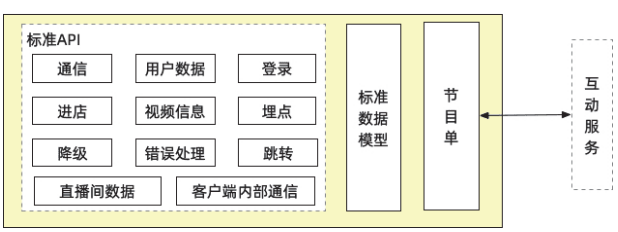
两种方案没有好坏之分，但由于今年猫晚有着多达十多种互动模式，为了减少适配的工作量，选择了第二种方案。

我们在平台上实现了一套“适配框架”，抹平不同平台的差异，它主要有三个职责：

a）提供标准化 API，整合容器能力；

b）提供标准化数据模型，不同平台组件可以获取相同的数据；

c）提供节目单与互动服务通信，获取数据，控制互动组件的渲染逻辑。



这样，互动组件只需要对接“适配框架”，无需关心平台。

3）组件标准化

决定“适配框架”方案后，就需要统一组件的整个生命周期，使得每个组件都有统一的标准。在此基础上，我们提供了统一的“脚手架”，把控组件从创建到部署的过程：

a）自动化创建组件托管仓库；

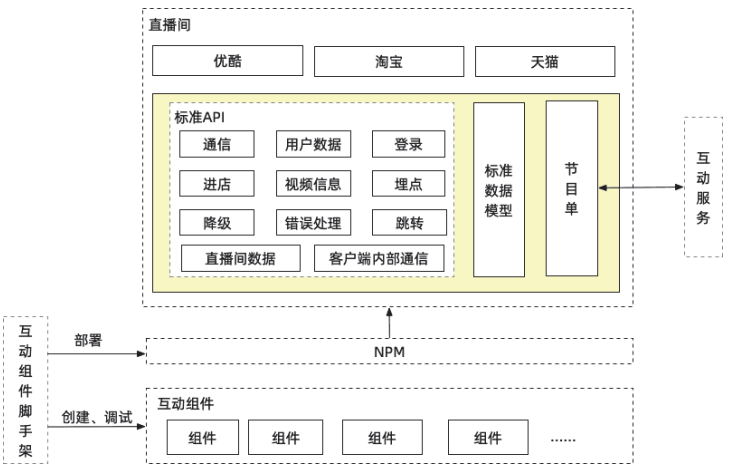
b）自动生成初始化目录和模版；

c）统一编码风格，统一 ES 版本；

d）提供开发调试能力；

e）自动化部署 NPM。

4）方案



跨平台输出效果

2. 容器化

每年的用户诉求都有所不同，每年的产品方案也在变化，必须要做好充足的准备。我们使用“容器”支撑不断变化的需求。

1）热更新与选型

客户端依赖版本发布、用户更新的策略来分发，比 H5 的分发效率低。但使用 H5 在移动浏览器环境下渲染复杂互动，会遇到性能瓶颈。选型方面，我们需求:

a）热更新

b）高性能渲染

c）前端可以轻松上手

框架上我们选择了 Weex，引用一则 Weex 官网的介绍：

Weex 致力于使开发者能基于通用跨平台的 Web 开发语言和开发经验，来构建 Android、iOS 和 Web 应用。简单来说，在集成了 WeexSDK 之后，你可以使用 JavaScript 语言和前端开发经验来开发移动应用。

Weex 最终渲染的是 Native UI, 性能上要优于浏览器渲染。借助 Weex 的渲染远程 jsBundle 的能力，实现实时更新。

Weex 的渲染引擎与 DSL 是分开的，在 DSL 选型上有两个选择 Rax 或 Vue，考虑到阿里集团内复用性上，选择了 Rax。Rax 在几届双 11 中也有很好的表现。同时，我们也保留了 Webview，渲染一些 H5 游戏。

2）容器化的能力

Weex 有两种方式来扩展能力:

a）Module 可以封装一些接口，调用原生能力；

b）Component 可以使用原生实现 Weex 组件。

有了手段以后，容器能力需要封装到哪种程度？有两个选择：

a）提供完整的业务组件，开箱即用。业务只需摆放位置，拼装；

b）只提供基础能力，业务需要在此基础上实现业务逻辑。

显而易见，想要支撑不断变化的需求，容器的能力不能太定制化。所以我们选择了第二种方案：

a）在 Module 方面，容器提供了包括“流媒体、下载、相册、通信、埋点、键盘、支付、屏幕旋转”等 31 个 Module，给容器扩展出近几十种原生能力；

b）在 Component 方面，封装了“Webview、播放器、动画、进度条、聊天列表、弹幕”等 18 个 Component，扩展出了一些通用的 Native View，提高渲染性能。

最后，我们抽离了直播间逻辑。使播放，通信等基础功能功能可以单独剥离出来，做为“直播间基础逻辑”，业务层可以选择继承方式来扩展自己的能力。

3）模版，创造更多的搭配

在日常的业务中，容器需要支撑多种多样的直播。有普通直播，也有猫晚这种大型互动直播。

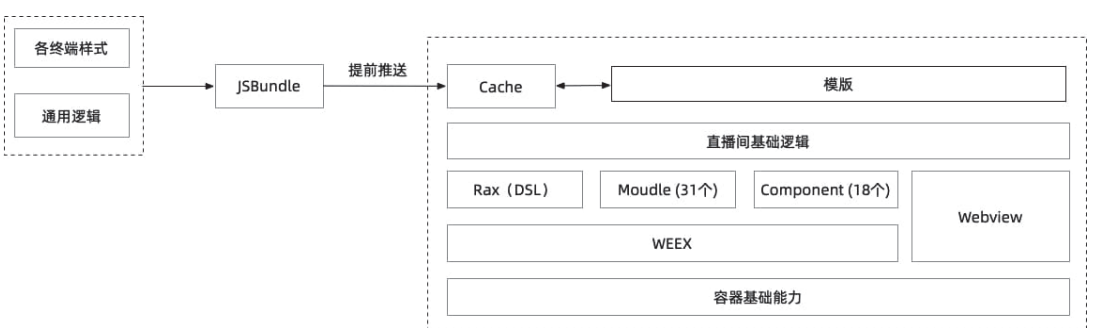
我们在容器中增加了“模版”的概念，把模版和直播类型建立起关系，通过不同的模版来支撑不同类型的直播。例如猫晚直播间会渲染“2019 猫晚模版”，展现给用户相应的界面。
在模版加载过程中，加入了提前推送 Cache 的能力，使得大部分用户可以直接从本地加载模版，大大加快了首屏渲染的速度。

4）输出到多终端

搞定了移动端后，我们还需要处理 PC 和 H5，每个终端单独开发有些费时费力。
在多端输出方案上，容器借助了 Rax 的跨平台渲染能力。剥离了组件的样式和业务逻辑，使得各终端可以共用一份业务逻辑，然后实现不同的样式，最后分别构建后运行在各终端的容器中。

5）方案

最终形成了容器化的方案：



最终渲染结果：

phone　　H5　　PC

3. 体验优化

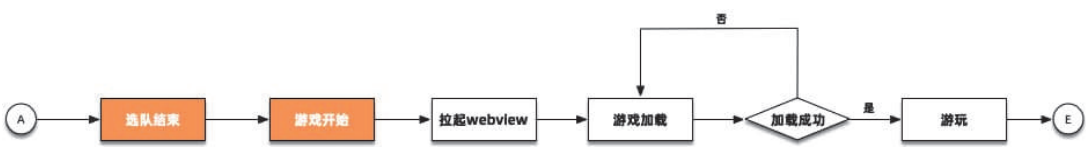
在保证了产品需求落地后，还需要考虑如何保证互动有“丝般顺滑”的体验，下面介绍其中几个优化方案。

1）互动游戏秒开优化

今年双 11，用户在直播间内和现场进行了游戏互动。互动开始后，会进入选队阶段，用户在此阶段可以选择自己喜欢的队伍。选队阶段结束后，线上会和现场连线，一起进行一次互动游戏。

游戏本身是依赖 H5 游戏引擎实现的，代码体积较大。在浏览器环境中，会遇到加载速度的挑战。网络层面上，每个用户网络环境不同，必然有网络不好的情况出现。用户可能会等待很长时间才能玩到游戏，体验非常不好。

我们来看一下游戏加载的流程：



可以看到，服务器会在“选队阶段”开始一段时间后，发送“选队结束”的信号，让所有用户停止选队。等待晚会现场可以进行游戏后，会发送“游戏开始”信号，客户端会拉起 webveiw

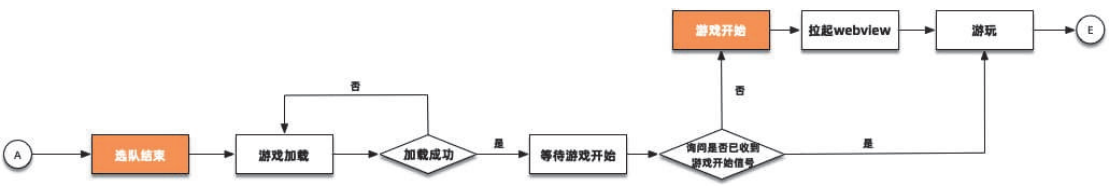
面板去加载游戏。

有两个阶段用户无法操作互动，需要等待。一个是“选队结束”信号收到后，直到收到“游戏开始”信号。随后会拉起 webveiw 面板；另外一个是“游戏开始”信号收到后，直到游戏加载成功。

第一个阶段服务器发送“选队结束”、“游戏开始”这两个信号之间所需时间，是依据晚会现场的节奏，这个优化空间比较小。

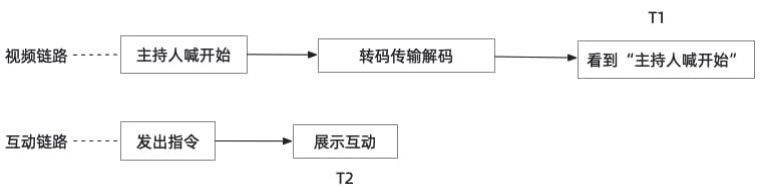
第二个阶段，游戏本身会做一些资源压缩，资源整合，可以从体积层面提高游戏加载速度。但是这也有局限性，游戏的图片、特效等很丰富，优化后的资源体积也是远远大于普通 H5 页面的，再叠加游戏初始化耗时，在普通网络环境下无法做到“秒开”。

在“空间”层面上，我们无法再优化的情况下，我们是否可以从“时间”层面上做一些优化？从上面的流程图中可以看到，选队结束后，就已经可以做游戏的加载初始化工作了。随后游戏初始化完成，收到“游戏开始”指令后，只需拉起面板即可。这样游戏就会是提早准备好，大大缩短了等待时间。



2）互动与视频对齐

晚会虽然是同步直播给用户的，但综合每个用户的网络情况以及客户端解码性能，每个用户在同一时刻看到的画面是不同的。以“互动开始”信号为例，“互动指令”是在主持人说出“开始”等关键信号时，下发给客户端。



“视频”和“互动”是两条互不相干的链路。我们把用户看到“视频画面”的时刻记为 T1，看到“互动”的时刻记为 T2。可以看出，“T1”与“T2”是没有任何关系的，除非凑巧，理论

上用户是无法在看到“主持人喊开始”时，同时看到相应的互动。这样就会缺失“现场感”，互动直播的质量会大打折扣。

那如何让用户看到相应视频时，同时看到相应互动呢？

视频编码中的 SEI（附加增强信息)，可以附带一些自定义数据。如果使用 SEI 附带互动标识，真正的互动数据则提早一些下发到客户端。互动可以实时监听解码视频时的 SEI 标识，发现当前互动的标识时，执行相应的操作。

这样用户就会在看到“主持人喊开始”时，同时看到相应的互动展示，就有了“现场感”。

3）高并发下的服务端稳定性优化

在互动开发中，避免不了客户端与服务端做一些数据的通信，其中又有许多，是程序自动触发的。尤其在互动直播场景下，用户会集中在某个时间段内做一些操作。QPS 监控上，往往可以看到一根尖刺，请求时机都比较集中。同时在线只有几千个人的情况下，没有什么优化的价值。但是如果像猫晚这种百万级别同时在线的互动直播中，就很有优化的必要了。

这样会带来几个好处：

a）可以大大降低 QPS，减少部署工作成本；

b）减少集群机器数量，节省开支。

同时在线越高，这些好处就越明显。在开发大型直播互动时，提前要建立起流量评估模型，方案细化到每一个接口。如果一些请求过于集中，就通过一些手段打散掉（在一个时间段内取一个随机时间点，发送请求）。后续演练场次里面，不断地验证模型的准确性。如果有不符合的地方及时修正。

四、大型直播互动经验

最后分享在参与大型直播互动中的经验：

a）技术设计、选型，以稳定性为标准，客户第一，保证用户权益、用户体验；

b）多进行演练，并且搜集数据，重点分析预期外指标，调整对应技术参数；

c）多平台统一架构，一套代码，多平台部署，保证一致性；

d）建立流量评估模型，消峰策略，降级策略；

e）监控体系要齐全，预案要细化。

关注我们

（阿里文娱技术公众号）

关注阿里技术

扫码关注「阿里技术」获取更多资讯

加入交流群

1）添加“文娱技术小助手”微信

2）注明您的手机号 / 公司 / 职位

3）小助手会拉您进群

By 阿里文娱技术品牌

更多电子书

扫码获取更多技术电子书