

阿里文娱 **技术**  阿里云 开发者社区

# 一次看懂 文娱B端核心技术

抢票技术 | 赛事保障 | 实时大数据

—— 阿里文娱技术精选系列 ——  
电影演出及制片宣发技术

### 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

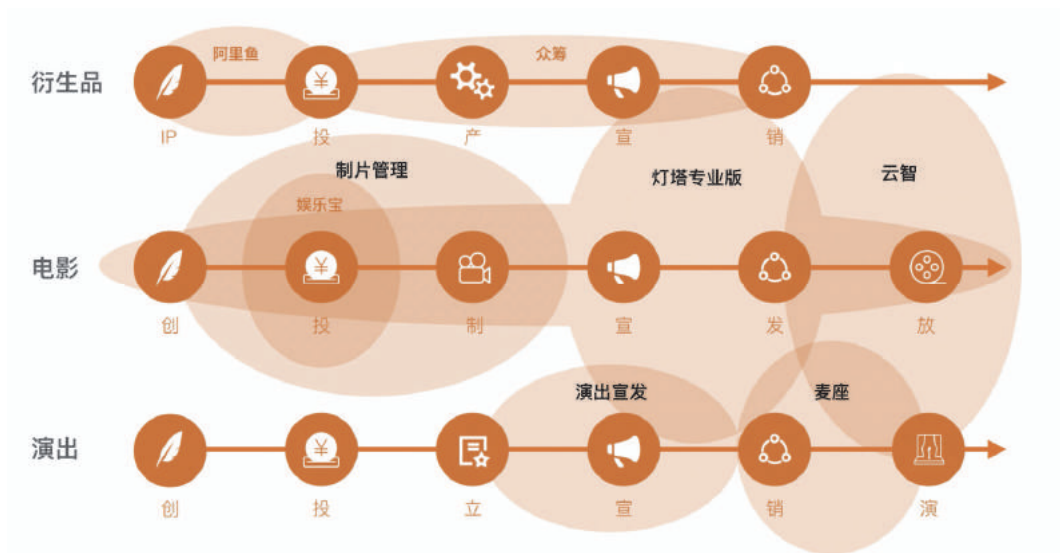
# — | 目 录 | —

<b>1 电影</b>	<b>3</b>
电影垂直行业的云智开放平台如何炼成？	4
阿里工程师带你了解 B 端垂类营销中心如何设计？	14
云智前端技术如何赋能场馆院线？	20
60 秒售出 5 万张票！电影节抢票技术揭秘	25
电影行业提升 DCP 传输效率，还能这样做！	33
<b>2 演出</b>	<b>38</b>
超大型场馆的绘座选座解决方案	39
大型赛事稳定性保障：Dpath 为世界军人运动会护航	48
世界顶级赛事的票务支撑：百万座位与限时匹配	52
前端技术：Webpack 工程化最佳实践	59
<b>3 宣发</b>	<b>68</b>
电影票房数据查询服务高性能与高可用实践	69
基于 webpack 的应用治理	75

## 序

阿里文娱作为阿里巴巴 Double H 战略的重要组成部分,需要为文娱 B 端客户持续创造价值,主要为片方、宣发方、影院、场馆、IP 版权方等客户在内容创作、投资、制作、宣传、发行、票务运营等环节提供产品解决方案,帮助客户进行数字化转型升级,提升多端跨场景运营效率。

在内容创作和制作环节,为制片方提供数字化制片管理软件“云尚制片”,为主办方和片方提供金融服务平台“娱乐宝”;在宣发环节为主办方、片方和宣发方提供一站式宣发服务平台“灯塔”;为场馆和院线客户在演出和电影票务运营环节提供数字化经营管理服务平台“麦座”和“凤凰云智”。



2020 年,文娱 B 端在依托阿里巴巴基础技术能力之上,在大数据和开放平台两个方向重点推进产品建设,推动行业数字化、智能化转型升级,打造文娱 B 端行业新基础设施,成为行业“水电煤”。

阿里文娱 B 端产品技术中心总监 修墨

2020.1.6

1

电 影



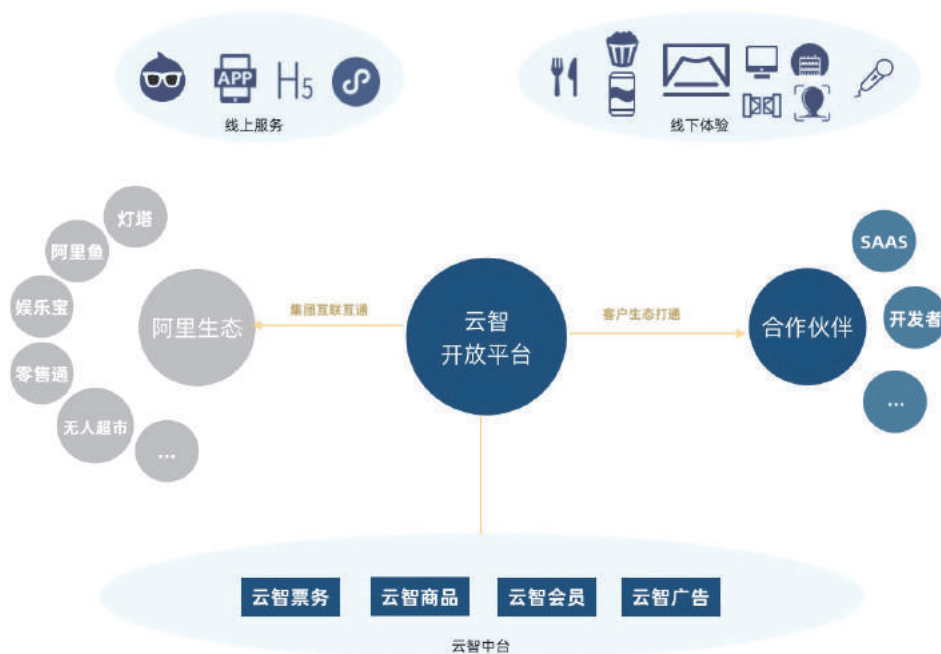
## 电影垂直行业的云智开放平台如何炼成？

作者| 阿里文娱高级开发工程师 念贤、阿里文娱高级开发工程师 宵征

### 一、前言

云智是阿里影业旗下的影院数字化经营管理开放平台，主要负责影院管理及影票卖品的售卖。

本文以云智开放平台为例，将为您揭开 B 端垂类电影行业开放系统的高性能 API 网关、高可靠消息服务、高安全性数据服务等技术内幕。



（图 1.1 云智开放平台生态）

## 二、开放平台架构大图

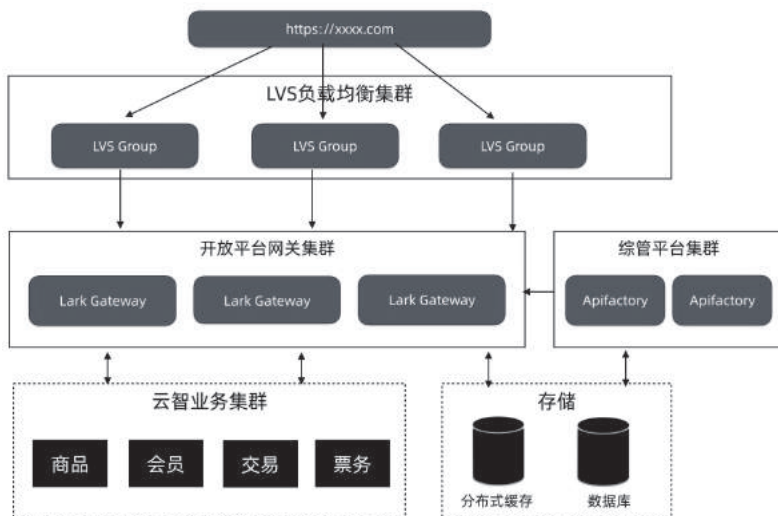
### 1. 开放平台技术大图



(图 2.1 开放平台技术大图)

开放平台的组成包括：泛 ISV 的调用方、综管门户、网关和业务系统，业务系统通过网关开放业务给到 ISV 调用方，网关与业务系统使用泛化调用，综管主要包括对 API 生命周期管理等功能。

### 2. 开放平台部署架构图



(图 2.2 部署架构图)

部署架构包括：网关集群、综管平台集群、云智各个业务集群，对于开放平台，核心关注网关和综管集群，为了网关的高性能，在云数据存储这块引入分布式缓存。

### 三、如何搭建高性能的 API 网关

#### 1. 网关总体技术架构图



(图 3.1 API 网关技术架构图)

架构图中的 5 层是网关真正的应用层功能，每一层都解决一个核心问题：

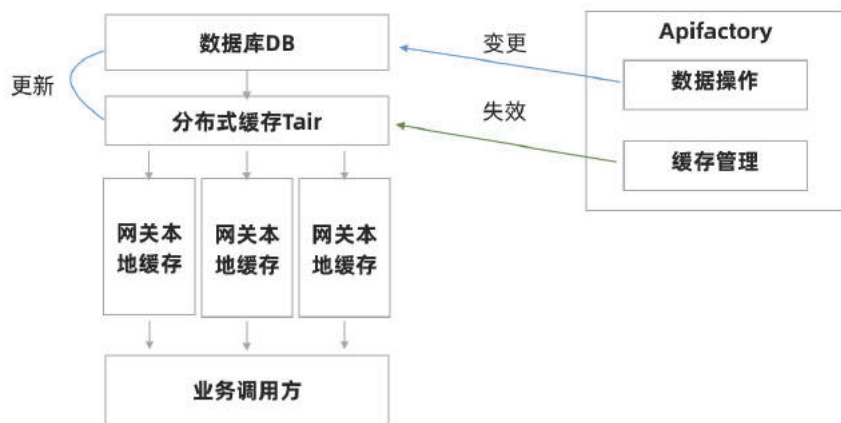
- 1) 协议层：解决客户端/调用方如何来连接网关的问题，主要做的事情，按照某种协议去监听，数据协议是怎麼样的，最终转换成网关内部私有协议；
- 2) 控制层：一个请求过来之后，网关如何认证这个请求是合法的，是经过认证的请求，保证每个请求都是安全的请求；



- 3) 调度层：针对一些异常处理，流程路由的处理，这里做的是要怎么样去处理的问题；
- 4) 服务编排：API 和 Server 之间的映射和编排，复杂的场景都是在这层进行处理的；
- 5) 调用执行：拿到这个请求之后，最终是调用外部的一个服务，是如何调用的，在这层实现。

## 2. 网关的缓存模式

### 1) 网关缓存模式：



(图 3.2 网关缓存示意图)

### 2) 流程说明：

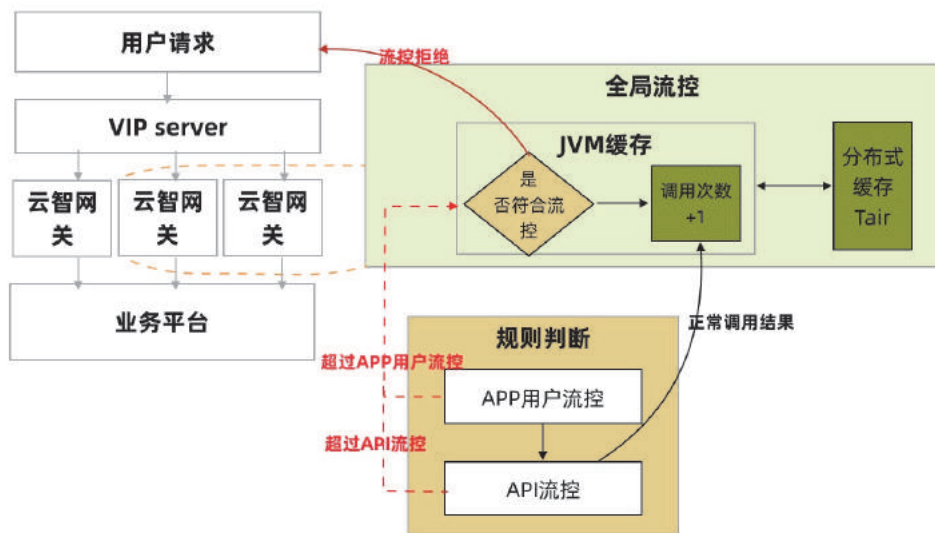
a) 网关会对缓存进行分级存储，以保证最快的访问速度，数据访问的时候，访问顺序依次是本地缓存->分布式缓存->数据库；

b) 在保证数据一致性方面，由于缓存的是元数据，元数据的特性决定了其变动少，时效性要求低的特性,因此,网关采取了分布式缓存主动失效，本地缓存被动失效的策略,已减少代码复杂度。

## 3. 多维度的流量控制

流量控制：既控制 API 在单位时间内允许被通过调用的次数，简称流控。

### 1) 流量控制的原理



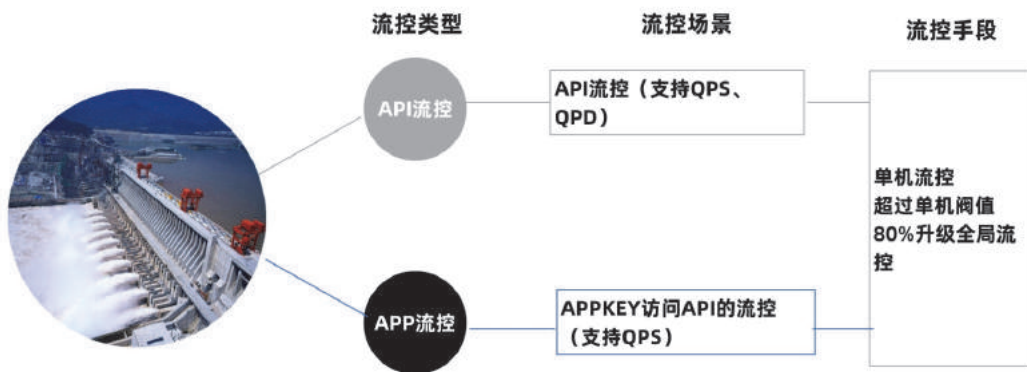
(图 3.3 流量控制流程)

## 2) 工作流程

单机阈值策略：总阈值/机器数 \* 80%；

全局流控策略：低于单机阈值：走内存计数；高于单机阈值：走分布式缓存计数。

## 3) 流控类型



(图 3.4 流量控制类型)

目前云智开放平台提供了以下流控方式，(1) API 的滑动窗口时间内的全局流量控制；(2) API 支持定制 APPKEY 进行流量控制；(3) 基于 APP 维度的流控，对 APPKEY 访问平台 API 的流量进行控制。

## 四、高可靠的消息服务

### 1. 总体架构



(图 4.1 消息服务架构图)

### 2. 消息模式

1) 网关的消息通知是通过异步的 HTTP 回调+消费方的主动确认两种方式来确保消息的可达，异步的 http 回调可以理解为消息的推模式，而消费方主动确认则可以理解为消息的拉模式；

2) 目前消息模式的使用场景主要是在异业会员之间信息拉通，例如会员消息注册，会员信息变更，积分变更等，均会使用消息来进行内部与外部系统信息的同步。

### 3. 如何保证消息不丢失？

- 1) 利用消息中间件本身的重试机制，确保网关能至少正确消费一次消息(at lease once)；
- 2) 接收消息后，持久化到磁盘，留待后续追溯与重试；
- 3) 根据消息的订阅关系投递到对应的订阅者，成功则更新消息状态，失败则留待定时任务扫描；
- 4) 对投递失败的消息，进行间隔 10，20，40 分钟的重试，三次重试均失败后，则标记为

失败，不再主动投递；

5) 消息消费方根据业务需要定期主动发起失败消息的查询以防止消息的遗漏。

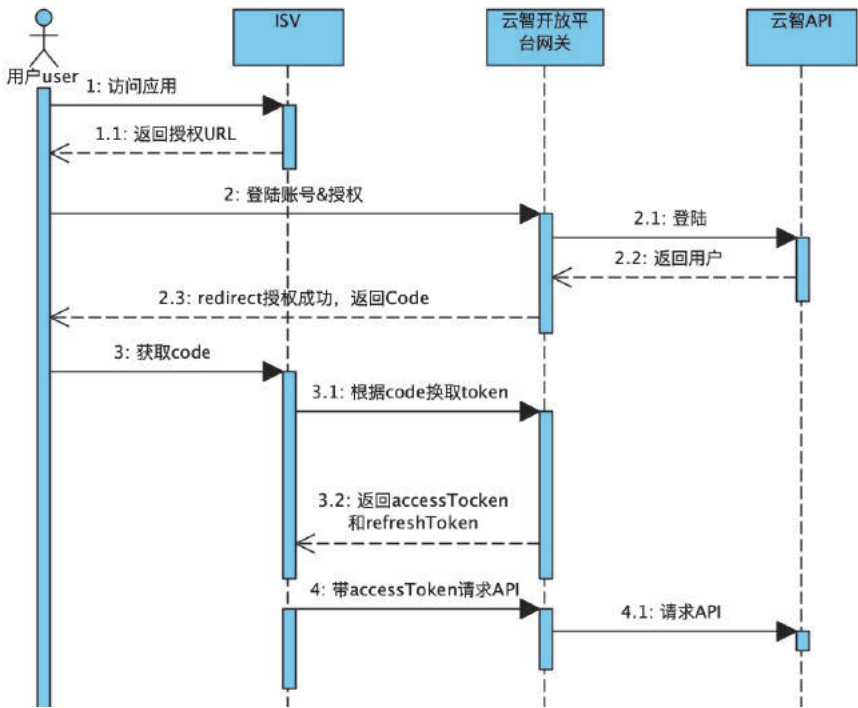
## 五、高安全性的数据服务

### 1. API 授权方式

云智开放平台采用 OAuth2.0 作为授权协议，授权流程可以简单归纳为：

- 1) 获取临时令牌 code;
- 2) 用临时令牌 code 换取长时令牌(refreshToken)以及访问令牌(accessToken);
- 3) 访问令牌过期后用长时令牌(refreshToken)刷新访问令牌(accessToken)。

授权及使用授权时序图如下：



(图 5.1 API 授权时序图)

## 2. API 访问控制

用于 API 访问权限控制，可限制 API 只对部分商户开放，或者不允许 APPKEY 跨商户、跨渠道等访问。

在云智开放平台中，访问控制主要由：权限组和访问控制水平鉴权配合组成，开放网关增加了访问控制水平鉴权的功能，达到控制商户访问隔离、数据隔离。访问控制原理如下：



（图 5.2 API 访问控制图）

对 APPKEY 进行访问控制的设置，配置访问策略，访问策略可以配置为 API 接口的请求参数，开放平台网关会根据访问策略判断用户的请求是否合法，不合法则抛出错误。

配置示例：

```
{"商户":["yunzhi"],"影院":["test1","test2"],"渠道":["H5]}
```

## 3. API 返回数据控制

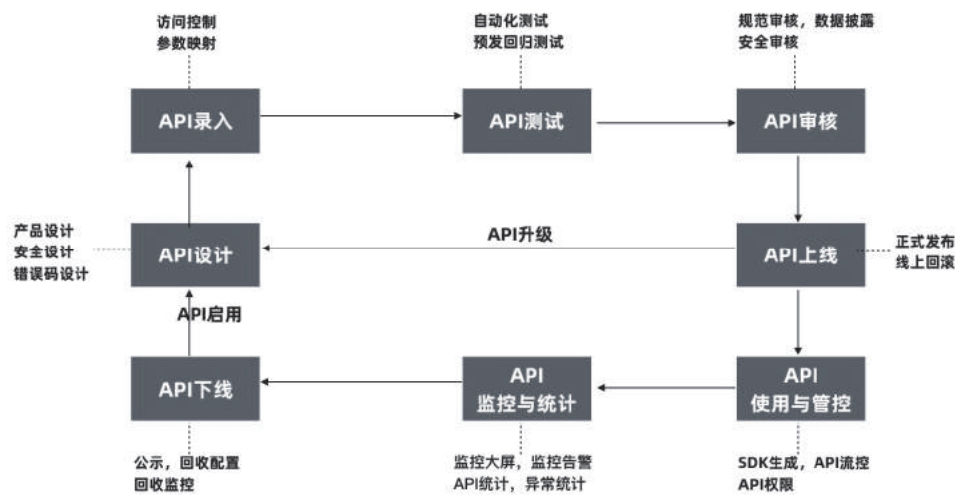
1) 对于分配的每个 APPKEY，网关都会给其定义一个相应的数据访问等级，目前一共存在从 L1~L4 四个级别，安全等级从低到高递增；

2) 对于需要严格管控的高危接口，在配置的时候，需要明确每一个字段的定义，从低到高同样分别为 L1~L4；

3) 通过插件机制对每个出参进行字段匹配的过滤，不符合权限等级的出参将被直接摘除。

六、可扩展、可维护

1. 可维护性-API 的生命周期

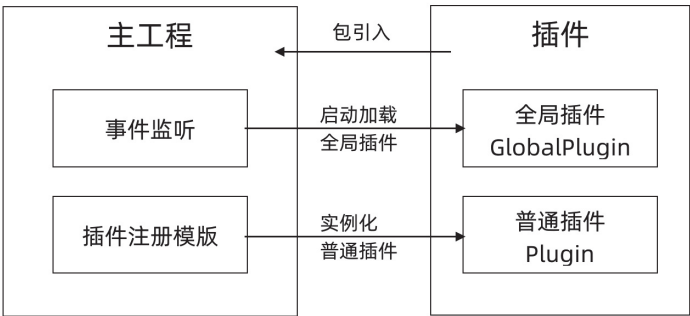


(图 6.1 API 生命周期管理图)

2. 可扩展性-API 的插件机制

网关的协议监听与解析层、API 逻辑处理层、调用协议与执行层，都作为组件，可插拔的集成到网关内核中。

1) 插件原理-类加载



(图 6.2 插件机制类加载图)

云智开放平台新增的，有电影行业特色的插件包括：API 访问权限控制-水平鉴权、支持云效环境项目标。

## 七、总结

云智开放平台是 https 通讯协议，多级缓存，消息中间件等技术的融合，专门为高性能数据访问与数据安全性把控而生的系统。达到了极速的访问与全方位的数据安全管控，回顾从系统的选型到诞生的整个过程，网关的演化也经历了下面三个阶段：

- 具备基础核心能力，基础核心 = 服务访问 + 稳定运行；
- 具备平台化能力，平台化 = 高性能 + 数据安全 + API 规范与审核；
- 垂类平台特色能力，垂类化 = 插件化定制。

在这个过程中，沉淀核心技术，深入了解和落地这些技术细节，在每一次的演进中，都思考是否可以在现有技术上面，再进一步地优化，让网关可以不但高效安全，并且易于维护，虽然过程困难重重，但是只要不畏艰难，必定可以攀上高峰，在这个过程中，团队的技术思想也在不断的变化，系统也炼出了垂类平台的特色。

# 阿里工程师带你了解 B 端垂类营销中心如何设计？

作者| 阿里文娱 B 端技术专家 和同

云智是阿里影业旗下的影院数字化经营管理开放平台，主要负责影院管理及影票卖品的售卖。本文以云智营销中心为例，为您揭秘 B 端垂类营销中心的高复用性、强扩展性的技术架构内幕。

## 一、架构设计方向

### 1. 营销中心设计

在业务架构设计上，将玩法通用特性进行抽象，实现营销业务和规则能力分离，静态管理和动态运行分离，营销中心可划分为能力平台和业务平台两大部分，如图 1 所示：

- 1) 业务平台：负责营销工具生命周期的维护、资产管理和其他各种业务场景的实现；
- 2) 能力平台：负责规则数据的标准化和规则关系的配置，将能力进行领域划分。

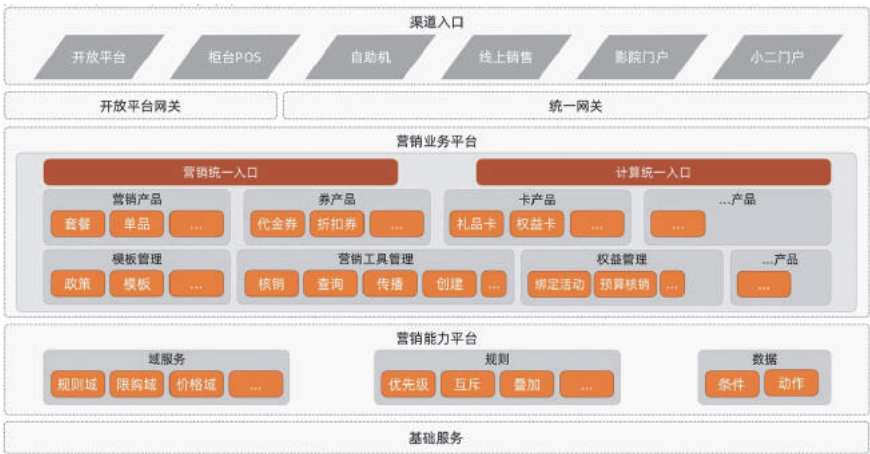


图 1 营销中心业务架构图



在系统架构设计上，业务平台按照业务管理视角组织数据，能力平台按照规则运行视角进行领域规划和服务暴露，如图 2 所示：

1) 业务平台把业务规则按照能力平台定义出的规则模型标准化，将规则数据同步到数据库中，通过标准化定义，实现规则祛业务化；

2) 能力平台对外统一暴露服务，并在优惠域进行统一参数组装，并根据对应的业务身份执行不同的脚本引擎，进而调用不同的领域服务，同时能力平台针对规则数据按照其作用范围进行领域划分，例如：负责计算的价格域、负责过滤的规则域等，针对不同工具特有规则由其对应产品扩展点实现。

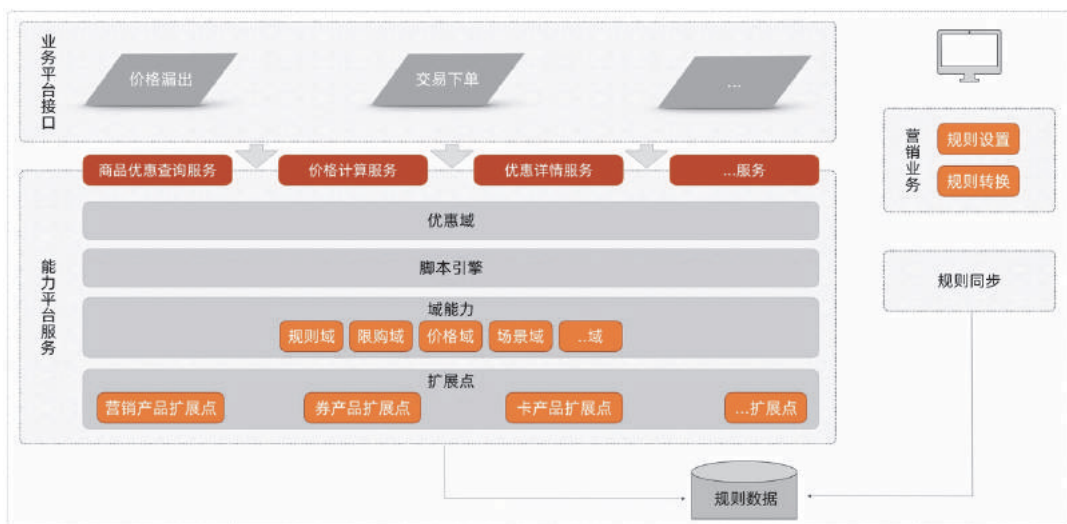


图 2 营销中心系统架构图

## 2. 营销中心解决了那些问题？

B 端营销具有工具多，玩法复杂的特点。垂直业务上可以分为如卡、券、积分等超过 10 种营销工具，每一种工具包含立减、打折等超过 7 类玩法，每一类玩法会在不同的工具中有差异化。因此，在传统的设计思路下就会出现“烟囱式”的建设模式，如图 3 所示：



图3 传统系统建设模式

这种“面向业务”的“烟囱式”建设方式会对业务和系统带来三大弊端：

- 1) 重复功能建设和维护带来的重复投资。单单从开发和运维两方面成本投入的角度，对于业务都是一种显性的成本和资源浪费；
- 2) 打通“烟囱式”系统间交互的集成和协作成本高昂。随着营销业务的发展，各“烟囱”之间不得不开始打通，涉及到大量的协同和开发成本；
- 3) 不利于业务沉淀和持续发展。受限于之前服务设计时的通用性和业务前瞻性不足，业务领域的数据和业务被打散到不同的系统中，这样无法满足业务快速响应和模式创新的需求，同时无法从更高维度上去观察和设计整个领域。

传统建设模式的主要问题是将业务和玩法规则混杂在一起，使得玩法规则和业务耦合极强，进而导致系统复用性低，可扩展性弱。营销中心充分针对这些问题，在架构设计和结构分层上，实现了高复用性和强扩展性。

## 二、如何实现高复用性

营销中心在面对大量带有业务特性的数据时，主要通过模型标准化，通用领域服务平台化，平台能力自进化三种方式实现高复用性：

- 1) 模型标准化：通过标准定义规则描述、统一计算模型，实现底层能力和逻辑祛业务化，将所有规则及商品数据按照标准模型进行重塑，即将各业务规则按照其本质拆解为条件&动作

的映射，不同类型商品按照标准模型转换，如图 4 所示；

2) 通用领域服务平台化：在领域服务能力实现过程中，将通用常见的能力抽象为平台能力，如果业务玩法没有特殊设置，则可以快速复用默认的平台能力。例如打折玩法，平台将四舍五入作为默认能力，新的业务接入时可以直接使用平台提供的打折能力，无需二次开发；

3) 平台能力自进化：当扩展点能力逐渐被更多业务使用时，可以将扩展点能力上升为平台通用能力，实现平台能力等级动态调整，满足业务对通用能力变化的要求，进而加强营销中心的复用性。

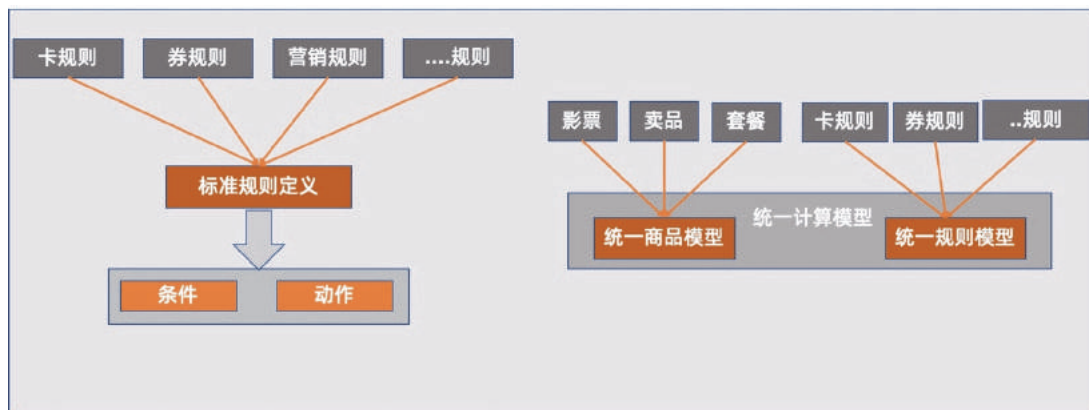


图 3 模型标准化示意图

### 三、如何实现强扩展性

营销中心的能力平台需要满足各种特性的业务玩法接入，所以在设计上我们通过产品扩展包和流程编排的方式实现强扩展性：

1) 产品扩展包：当平台提供的默认能力无法满足业务需求时，则由相应的产品扩展包来扩展实现。例如打折玩法，平台将四舍五入作为默认能力，但是在折扣券业务中，要求打折后取整，平台将这类玩法的特殊处理逻辑在折扣券扩展包中实现，当折扣券计算时，平台通过工具标识，将请求路由到对应的扩展点执行相应的特殊逻辑，从而解决规则的定制化问题。图 5 为营销中心能力平台调用流程图；

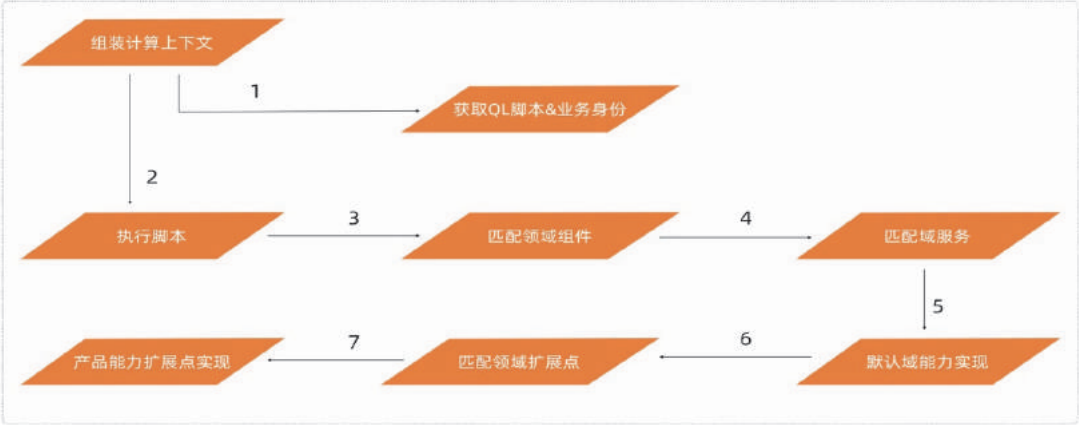


图5 能力平台调用流程图

2) 流程编排，通过流程编排的方式，达到允许业务按照场景进行自定义功能选择的效果，实现用户自定义需要。能力平台的流程编排引擎采用 QLEExpress 技术，如图 6 所示。

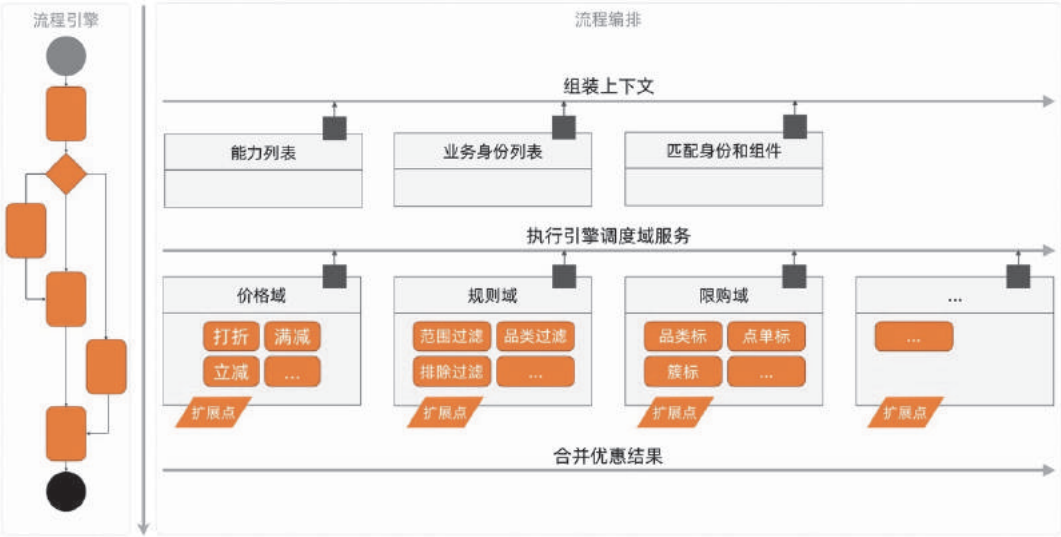


图6 流程调度域服务示意图

#### 四、总结

云智营销域从“烟囱式”架构演进成“平台化”架构，主要参考了 TMF 框架的分层、领域

划分思想，使用 QLEExpress 脚本引擎实现服务编排和服务调度。

在架构选型时，与 TMF 同类框架的还有 NBF 框架，相比之下，TMF 更重视业务抽象，但 TMF 中有一些分层是 B 端营销业务不需要的，所以营销中心参考 TMF 的设计思想，形成了适用于 B 端营销的 BEF 框架，上文所述的解决方案都是 BEF 的一部分。在 BEF 落地过程中，对优惠计算模型和规则模型进行了抽象，以满足对业务对象的需要。因此抽象成为了平台化的关键所在，抽象程度决定平台深度。

在流程引擎选择上，QLEExpress 同类技术还有 Drools，但虑到 QLEExpress 较 Drools 在性能方面有明显优势，可读性较强，开发门槛较低，营销中心最终采用了 QLEExpress。

# 云智前端技术如何赋能场馆院线？

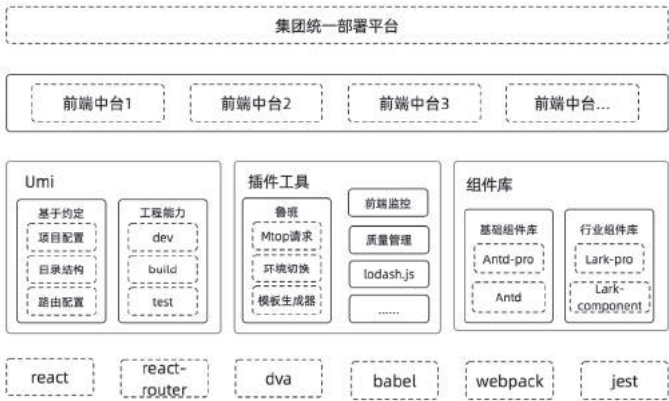
作者| 阿里影业高级前端工程师 余缺

## 一、背景

场馆院线的互联网化，就是将场馆院线实体，通过互联网的手段，实现票务的全网分发营销。前端在该进程中，扮演着多系统操作的可视化、简易化，协调调度的流程化、规范化。本文将为你揭秘前端技术和组件库技术如何解决场馆院线前端面临的版本碎片化、需求个性化、迭代效率低、维护成本高的问题。

## 二、场馆院线的前端技术

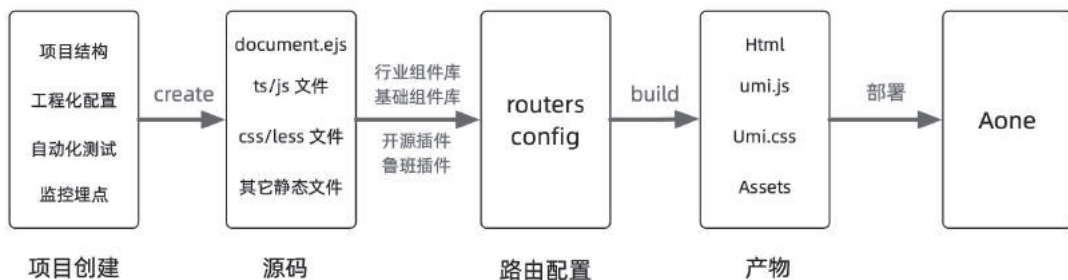
一个垂类行业的系统，往往是多系统协调工作，开发和维护成本都很高，但垂类行业系统都有着较高的相似性，大多数都是由查询列表页、编辑页、详情页组成。同行业的前端系统相似性更高，需要结合行业特有的业务场景，形成行业前端解决方案，方便多套垂类系统复用，减少维护和开发成本。



（图：阿里影业云智业务前端系统架构图）

通过在开发过程中沉淀经验和对开源软件的封装，影业前端团队形成了高效的研发体系和开发规约，基于约定大于配置的理念，工程化的思想渐入业务。实现了多个项目的项目结构、文件路径、路由配置，发布和上线方式全部打通，统一管理，减少了配置时间和学习成本。

同时我们在四个部分进行了流程上的改进：

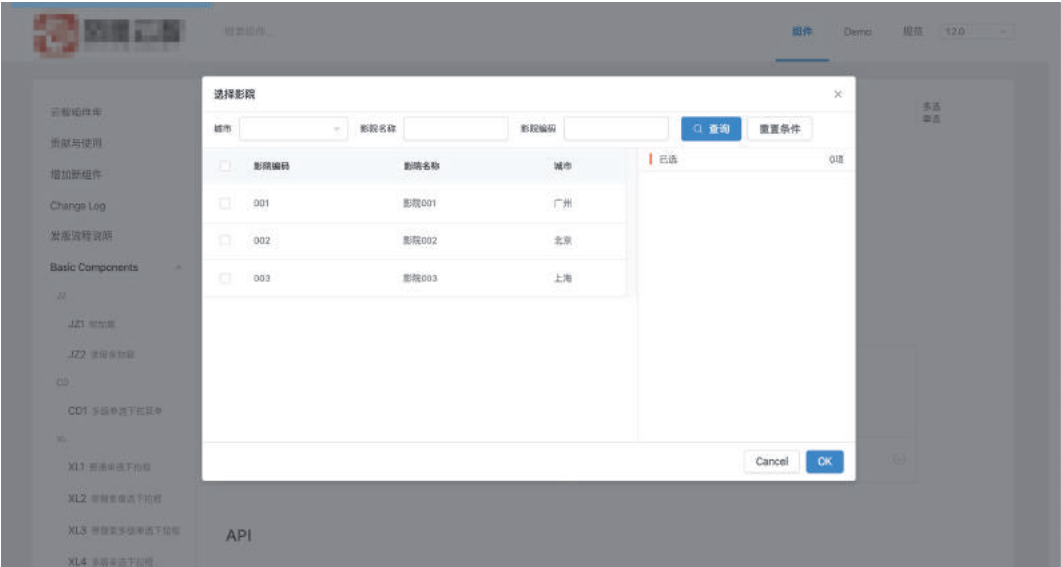


（图：阿里影业云智业务前端系统流程图）

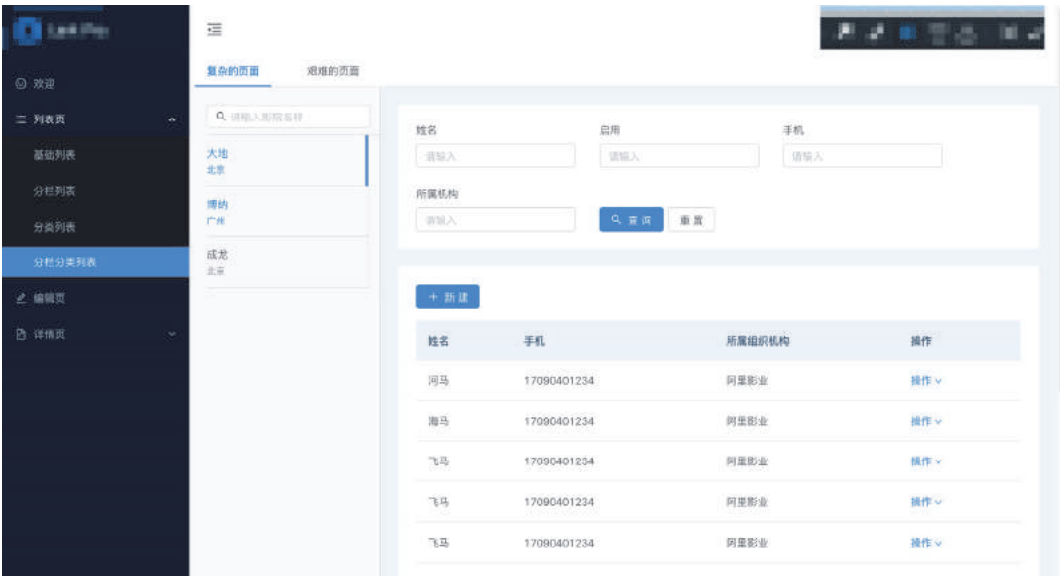
- 1) 前端工程化：使用 Umi 框架提供项目的基本结构和开发规范以及基础工程能力；
- 2) 插件和工具：根据团队需要，在开源插件工具库不能满足需求时，开发了鲁班系统，一个前端工具集合；
- 3) 行业组件库：在视觉层面积累行业经验，沉淀了行业视觉组件库，共同支撑多套前端垂类系统，助力业务的快速迭代，覆盖绝大多数业务场景；
- 4) 前端监控和质量管埋：统一的监控插件和质量管埋标准，在项目生成时内嵌，无需开发时手动接入。

### 三、场馆院线前端组件库

在业务的开发上，我们大量使用组件库提供的交互和视觉，目前主要的基础组件库是 Antd 和 Antd Pro，通用程度较高，但对于特定的行业和业务场景还有些不够。我们深耕场馆院线和积累业务场景，在原有的组件库基础上进行了拓展，沉淀出了两套对应的场馆院线前端组件库，分别是元素级别的组件库 Lark-component 和页面级别的组件库 Lark-pro。



(图：Lark-component 组件库实例)



(图：Lark-pro 组件库实例)

在场馆院线前端系统中，如根据城市或者影院名称筛选影院的场景，基本覆盖到所有的场馆院线系统，我们对这一场景进行抽象，封装在页面级别的组件库 Lark-pro 中。





(图：日期选择视觉图)

经过对用户日期选择行为的数据进行分析，不断的优化交互体验，沉淀出满足不同需求的日期选择类型，如单日的日期选择，带左右箭头的简洁型日期选择，再对细节进行像素级别的不断打磨，落地到元素级别的组件库 Lark-component 中。

#### 四、总结

我们试着用前端水平横向方案和组件库方案，去解决前端场馆院线问题，基于约定大约配置的理念，最大化的统一需要开发人员手动配置的内容，如项目编译配置、路由配置、镜像配

置、发布管理。同时基于行业背景下，产出更加丰富行业前端组件库，去覆盖高频率使用场景。

通过不断积累和沉淀，我们在前端系统上可复用的内容会越来越多，逐步完善场馆院线的前端基础建设。在 2020 年，面对 10000+ 的影院市场，前端系统会朝着智能化搭建、积木式搭建方向演进。快速、高效、稳定的服务用户，助力阿里场馆院线实现观影人次、影院数量双第一目标。

# 60 秒售出 5 万张票！电影节抢票技术揭秘

作者| 阿里文娱高级开发工程师 念贤

## 一、背景介绍

对于电影爱好者来说，每次的电影节、影展活动，都是抢票大战的开启，出票速度几乎可以用“秒空”来形容，例如上海国际电影节线上开售的记录是 60 秒售出 5 万张。

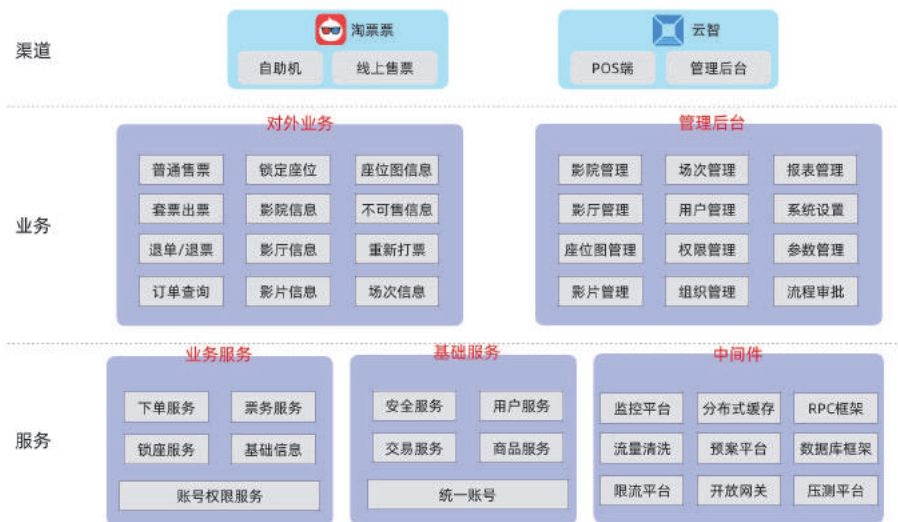
本文主要围绕售票环节，讲述阿里文娱的云智系统是如何支撑高流量并发，保障系统的稳定，不出现重卖等实现方案背后的技术。

先简单分析一下电影节的抢票业务，典型特征是在大流量抢购、高并发的场景下，让用户极快的锁定座位然后出票，特别是热门的影片，会异常的火爆。第一道压力是查询已售座位列表和锁座，需要能快速的支撑用户的锁座请求，且实时查询到已售卖的座位列表，避免发起无效的锁座请求；第二道压力是出票，如果锁座成功，但一直出票失败，会给用户带来很不好的体验。

## 二、架构设计思考的方向

### 1. 让业务赢

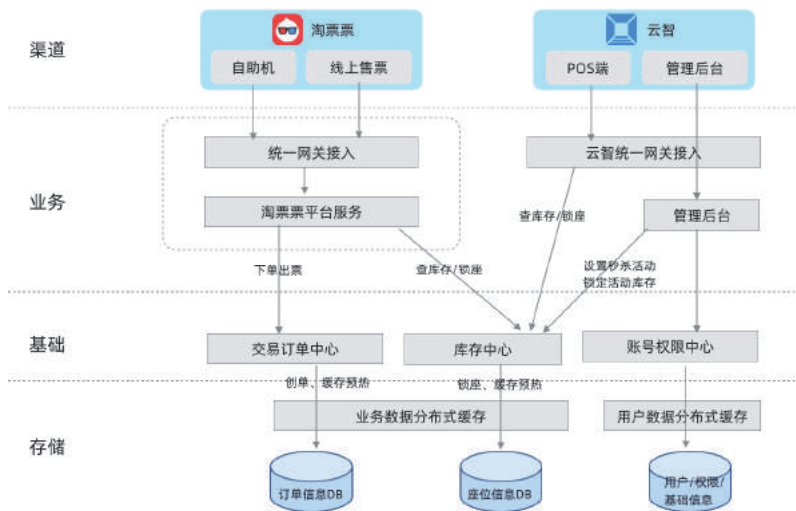
在分层设计上，分成渠道接入层、业务层和服务层。在业务层，对外业务和管理后台功能独立，职责清晰，快速支撑业务；服务层沉淀基础服务，构成稳定的业务和基础服务。



(图 1 业务技术大图)

## 2. 让系统稳定

在架构设计上，接入统一网关让系统安全，有限流，对库存中心和订单中心进行数据隔离，且加入多级缓存方案，让系统稳定。



(图 2 技术架构图)

### 三、实现方案与技术解析

#### 1. 高并发流量如何抗？

电影节的流量是非常典型的秒杀场景，瞬时流量非常高，对于系统的高性能要求就注定很高，在云智中，我们是如何抗高并发流量的？我们通过以下三点来进行阐述：热点数据隔离、流量削峰漏斗、多级缓存。

##### 1) 热点数据隔离

在热点隔离这块，云智选择的策略包括：数据隔离和业务隔离。

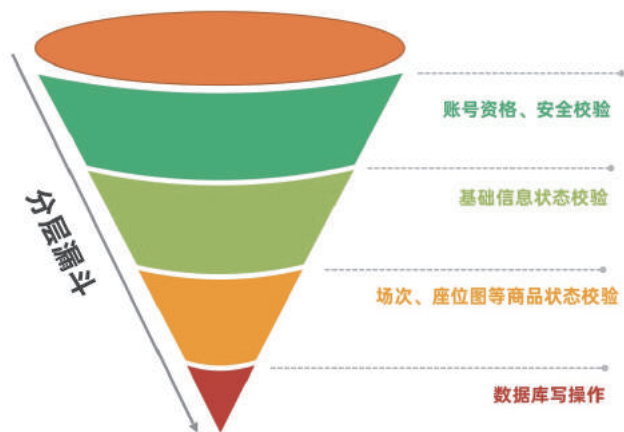
数据隔离：是把查询已售卖座位和已锁定座位等库存相关的热点数据，隔离出来，单独业务数据库，且使用分库分表，减少系统性能压力，提高吞吐量。

业务隔离：电影节的业务数据，独立的业务数据生成能力，圈定参与活动的业务数据，进行缓存预热，起到隔离的效果。

##### 2) 流量削峰漏斗

关键词是“分层削峰”，漏斗式的减少请求流量，在业务链路的过程中，我们会进行业务校验，层层过滤，如用户的账号安全、购买资格，影院、影厅等基础信息状态是否正常，要购买的商品信息状态是否正常、秒杀是否已经结束等，每个层次都尽可能的过滤掉非法的请求，只在最后端处理真正有效的请求，最终减少请求到数据库 DB 的写操作流量，保证系统处理真正有效的请求。

以锁座流程为例子：



(图3 流量削峰漏斗示例图)

### 3) 多级缓存

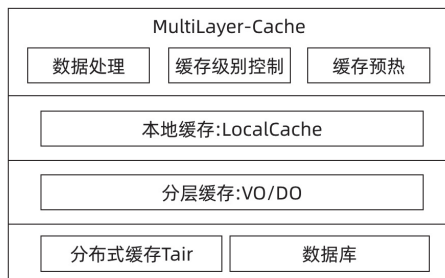
在分层漏斗的前提下，云智采用分布式缓存和本地缓存 LocalCache 多级缓存的方案来抵抗高并发流量，以下简要介绍一下在系统中使用的策略：

a) 缓存预热。在指定参加活动的场次后，会在限定时间内停止变更，在开售前，会自动进行预热缓存，避免激增流量击穿缓存；

b) 缓存失效时长控制，对基础数据实体的 VO 对象和 DO 对象采用失效时间长短的缓存控制，静态数据和 DO 实体使用长失效时长的策略：不失效或 24H；动态数据和实体 Info 使用比较短的失效时长策略：分钟级，比如幂等性 KEY 的缓存时间为 2min；

c) 本地缓存 LocalCache 使用的缓存时长策略分 3 种：2s，60s，122s。优先读本地的缓存，其次读远程分布式的缓存，使得系统可以抵抗瞬间的高并发流量。

示例图如下所示：



(图 4 多级缓存示例图)

将缓存分 2 层结构：

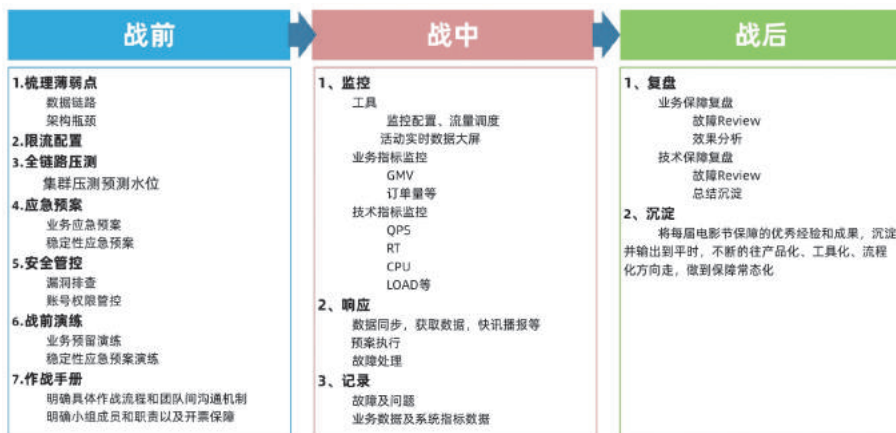
第一层是本地缓存结构：用户、权限、基础信息等静态数据，我们优先选择本地缓存；

第二层是全量的缓存实体信息的 DO 和 VO 信息，这层采用的是 Tair 分布式缓存。

## 2. 系统的稳定性、高可用性如何保证？

对于任何档期或者活动，系统的稳定性都是第一要素，针对电影节的活动场景，我们使用了很多设计上的稳定性模式，其中比较核心的有：多轮全链路压测、限流、降级、动态扩容、流量调度、减少单点、依赖简化等方式；除了以上几点，本节我们重点聊一聊我们在电影节过程中是如何保障备战的？

### 1) 保障备战体系



(图 5 保障备战体系图)

### a) 在战前阶段

这个阶段的工作会比较多，只有做到事前充分准备，才能有更好的保障结果，主要包括以下几个部分：

- (1) 梳理薄弱点，包括系统架构、系统薄弱点、核心主流程，识别出来后制定应对策略；
- (2) 全链路压测，对系统进行全链路压测，找出系统可以承载的最大 QPS；
- (3) 限流配置，为系统配置安全的、符合业务需求的限流阈值；
- (4) 应急预案，收集各个域的可能风险点，制作应急处理方案；
- (5) 安全保障，主要聚焦在账号权限管控，以最小够用原则为准，防止权限滥用，安全无小事；
- (6) 战前演练，通过演练来检验保障体系是否完善，演练开票现场，提高团队响应和处理能力；
- (7) 作战手册，制定作战手册，明确作战流程和关键点节点的任务以及沟通机制。

### b) 在战中阶段

活动开售，我们也称为战中，整个项目组主要专注三件事情，即“监控”、“响应”和“记录”。项目组的同学都必须要保持作战状态，严格按照应用 owner 机制，负责巡检应用情况，及时同步技术数据和业务数据是否有异常。同时，在战中，我们临时组建“保障虚拟小组”，用于应对大促期间可能出现的紧急客诉等问题，及时做出决策，控制影响范围，同时也能提高整体作战能力。记录，是在战中过程中必须要记录下各应用的峰值，及时沉淀技术数据，为后续系统建设，流量评估等提供参考借鉴。

### c) 在战后阶段

这个阶段的主要工作是项目复盘，复盘的内容主要包括：项目结果、项目回顾、项目沉淀和改进，将项目过程中收集到的问题和故障进行详细分析，并将项目过程中沉淀出来的，关于系统稳定性保障的经验沉淀到日常，让活动保障的常态化逐步落地。

## 2) 最佳实践

### a) 精准监控

通过监控，实时发现各个服务是否触发限流值，及时进行 Review，调整限流值，保证业务



成功率和系统稳定。

对系统基础值班和业务量指标进行精准监控，如 load，内存，PV，UV，错误量等，避免因内存泄露或代码的 Bug 对系统产生影响，精准监控，提前感知内存泄露等问题。

#### b) 数据大盘

通过数据大盘，实时汇总数据，展示业务数据，为系统、为业务提供更加直观的业务支持，也可以更加有效的进行业务备战

### 3. 如何保证不出现重卖？

在业务过程中，我们实现了很多业务，解决了很多困难，我们重点阐述以下两个痛点，一个是恶意锁座，一个是防止超卖。

#### 1) 如何解决恶意锁座？

首先我们采用的扣减库存方式是预扣库存，用户操作锁定座位时即锁定库存，那我们如何解决恶意锁座呢？

- a) 锁座订单中会生成一个“库存失效时间”，超过该时间，锁座订单会失效释放库存；
- b) 限制用户购买数量，一人最多只能购买 6 张票；
- c) 接入黄牛防控系统。

#### 2) 如何防止库存超卖？

电影票不同于电商业务普通的标品，是不允许出现超卖的情况，否则会出现重票，从而引发客诉舆论问题，所以在库存数据一致性上，需要保障在高并发情况下不出现重票，我们的解决方案是：

- a) 使用分布式缓存，在分布式缓存中预减库存，减少数据库访问；
- b) 使用数据库唯一键，在锁座表中，设定场次 Id 和座位 Id 做为唯一键。锁定座位时，如果座位已经售卖，会报出数据库异常，不允许某一个座位重复售卖。

## 四、总结

回顾电影节抢票，我们首先想到的是能抗高并发流量，能让系统稳定。通过上述章节我们揭开了高性能、高可用等背后的技术，展示了一个典型抢票大战的技术方案，核心技术包括：

- 让业务赢 = 完整的业务应用 + 支撑核心业务；
- 高性能、高可用 = 流量削峰 + 限流降级 + 多级缓存；
- 平台成熟化 = 完善的监控 + 保障方案。

在这个过程中，我们沿着让系统稳定、让业务赢的设计思想，不断的思考和落地这些技术细节，沉淀核心技术，以达到让用户体验流畅的抢票过程。

## 电影行业提升 DCP 传输效率，还能这样做！

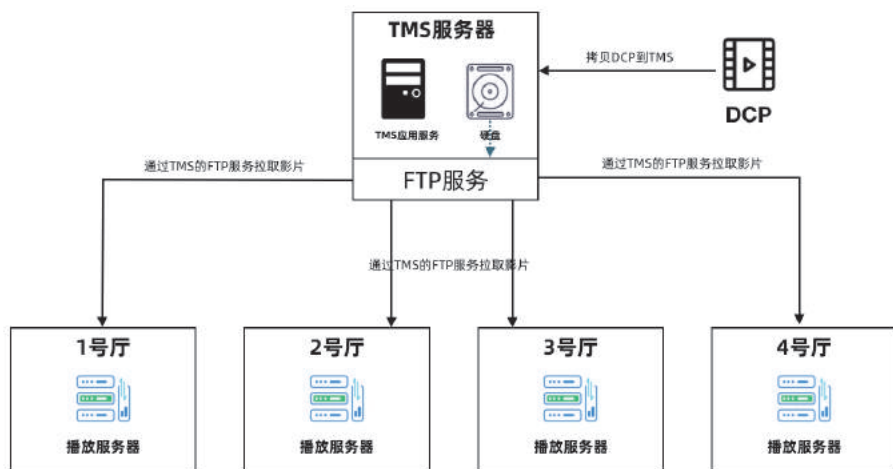
作者| 阿里文娱开发工程师 雨衡

### 一、背景

DCP 全拼是 Digital Cinema Package，中文是数字电影包，用于存储和转换数字影像的音频、图像和数据流，是影院放映设备使用的媒体文件包。一部普通 2D 电影的 DCP 大小一般在 40G~60G 之间，一部普通 3D 电影要乘以 2 倍，如果是 IMAX 或者 4k 的电影，DCP 的大小达到 200G 以上也是正常的。

本文揭秘超过 200G 的超大数字电影包如何高效通过 TMS 传输到各个影厅。

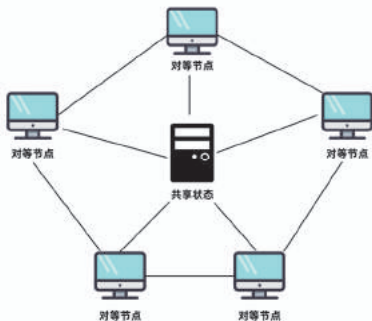
现在影院拷贝 DCP 到各影厅的方式主要是使用 TMS（影院放映管理系统）的传输影片功能，由 TMS 负责把 DCP 传输到各个影厅，但是这种传输的效率不高，数据源只有 TMS，所以各影厅拷贝影片都要到 TMS 上拉取，带宽就成为了瓶颈。



（图 1 目前使用 TMS 向播放服务器传输 DCP 的模式）

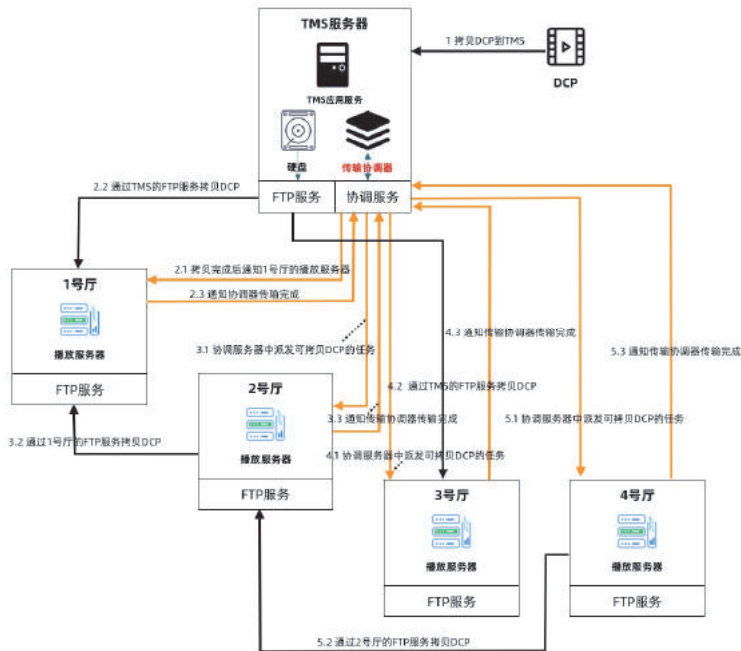
## 二、使用类 P2P 方式传输

从图 1 可以看出, 影厅的播放服务器拉取 DCP 后, 它们的带宽就处于空闲状态, 那么我们可以使用类似于 P2P (对等网络传输) 传输方式解决, 这样就可以用现有设施提高影院内 DCP 的分发效率, 起到降本提效的效果。



(图 2 P2P 示意图)

根据上面的 P2P 方式，改造影院内传输 DCP 的模型：



(图 3 改进后的 TMS 向播放服务器传输 DCP 的模式)

上图主要叙述的是传输协调器协调各个影厅寻找传输源拉取 DCP 的过程。本改进方案的特点是增加了一个传输协调器作为共享状态机，协调各影厅拷贝 DCP 的路径，计算出传输路径最优解。传输协调器的核心功能是：

- 1) 收集各厅播放服务器网络情况；
- 2) 标记 DCP 在各厅播放服务器的存储情况；
- 3) 根据网络情况，计算并派发传输任务到各厅播放服务器。

而且使用了本方案的传输方式，传输效率会有极大的提升。例如有一个 DCP 的文件总大小为 400GB，总共 10 个影厅，带宽为  $1000\text{Mbps}\approx 125\text{MB/s}$ ，那么使用传统 TMS 传输方式，起码要  $400\times 1024\times 10\div 125\div 60\div 60=9.10$  小时，差不多一个工作日的时间。而如果使用新方式，仅需要 3 小时，可以提升 3 倍，而且随着影厅的增多，效率提升指数增加。

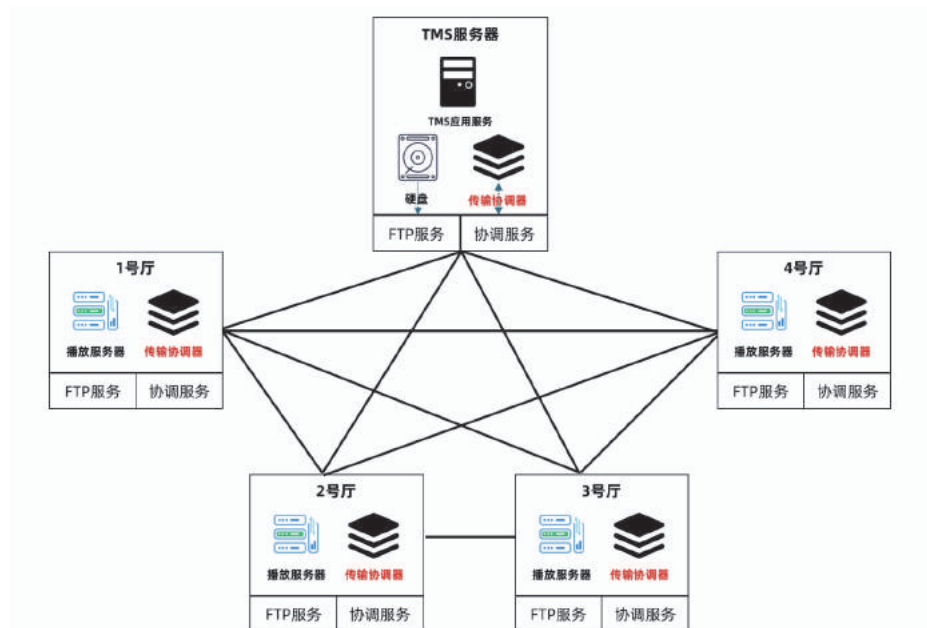
### 三、进一步改进方案

由图 3 了解到，虽然上述方案把带宽浪费的问题解决了，但是架构上还有一些问题：传输协调器就是一个单点，它挂了，传输就出问题了；数据只能有一个数据来源，来源挂了，传输就停止了，而且重新传输要从头开始。这时候我们要如何解决？

这两个问题可以使用传输协调器去中心化部署及文件分片断点续传方式下载解决：

#### 1. 传输协调器去中心化部署

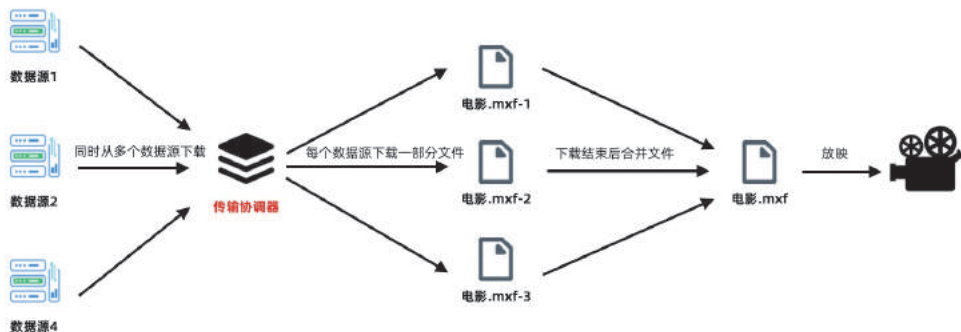
我们可以把传输协调器部署到各个影厅，每个传输协调器是一个几乎无状态节点，节点之间无任何信息同步，每当一个影厅的 DCP 传输完成后，就广播到各个传输协调器中。但发送拷贝指令的方式就需要改造一下，由 TMS 统一发送下载影片的指令到各个影厅的传输协调器，然后传输协调器就负责询问邻近节点是否有可下载的 DCP，存在则下载，不再需要 TMS 的传输协调器为影厅指定下载地址。新的网络拓扑图如下：



(图4 进一步改进后的网络拓补图)

## 2. 文件分片断点续传方式下载 DCP

上一节的方案中，拷贝 DCP 还是使用播放服务器原生指令操作的，限制很大，不支持多数据源及文件分片操作。既然播放服务器不支持，那么我们就需要自己开发一个中介角色，需要支持多数据源及文件分片操作，并且具备拷贝 DCP 到播放服务器硬盘的能力，而部署在影厅的传输协调器恰好可以承担这个职责。多数据源及文件分片方案示意图如下：



(图5 多数据源及文件分片方案示意图)

上述两种技术方案可以合并使用。

小结：虽然这种方式并不能提升多大的速度，但是在系统容错性方面有所提升，用户体验更好了。

## 四、总结

通过上述章节可以看出，我们通过将 P2P、FTP、断点续传、文件分片等技术的融合，产生了一个专用于局域网传输 DCP 的技术方案。我们借鉴 P2P 的思想，实现了影厅的片源在局域网内共享的效果，克服了传统 TMS 传输 DCP 单数据源的缺点；使用 FTP 作为传输手段，兼容现有影厅的传输模式；使用断点续传、文件分片提升系统的容错性。

这个方案其实是很典型的组合创新法，用的技术都是已有并且是很经典的，但通过将它们重新梳理整合，使其在性能上发生质的变化，以产生出新的价值。本文的方案正是使用这种方法诞生的，在设计这个方案的过程中，我也学会了组合创新法的一些皮毛，以后还要继续努力学习这种方法。

2

演出





## 超大型场馆的绘座选座解决方案

作者| 阿里文娱高级前端工程师 弋辰

超大型场馆座位图的加载与渲染在行业内始终是个不小的挑战，很多团队在面对绘座选座技术研发时，都难以支撑超大型场馆的座位图在浏览器端快速、稳定地渲染。如果你是奋斗在现场娱乐、剧院演出和电影院线相关行业内的是一名开发者，那么文章将帮助你更多的了解大场馆座位图技术，同时可以帮助你扩展绘座选座核心功能开发的思路。

### 一、业务场景

麦座是一套完善的数字化运营系统，连接内容和场馆，提供票务生产、票务销售、现场管理和演出运营等能力，绘座选座是麦座的核心能力之一，从建设场馆到规划票房再到门店销售。目前可支持的十万座位量级的场馆，包括剧院、演唱会、体育赛事等多个垂类。



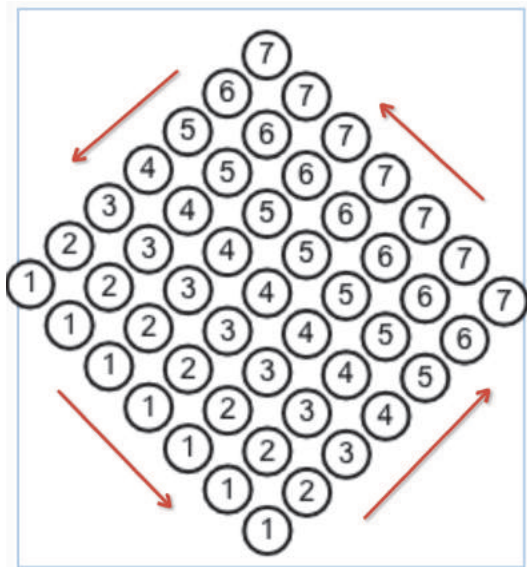
本文将介绍麦座的绘座、票房规划、锁座和门店销售等场景是如何通过技术手段实现业务需求的，以及超大型场馆的座位图是如何打破性能边界，给用户呈现出更流畅的交互体验。

## 二、功能场景

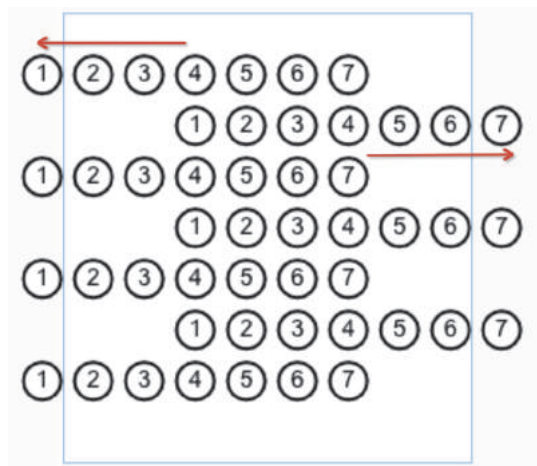
麦座的座位图相关的业务可以分为“绘座”和“选座”两大功能场景。选座就是通过用户事件绑定选中的座位进而完成接下来的业务逻辑，而绘座包括绘制座位、设置楼层、看台、舞台、出口等信息，绘制座位同时又涉及到如何给块级座位进行变形处理。那么，绘制的座位是如何填满看台区域的呢？又是如何将现场千变万化的座位排布展示出来的呢？下面我们带着疑问来看看麦座的解决方案。

### 1. 绘座

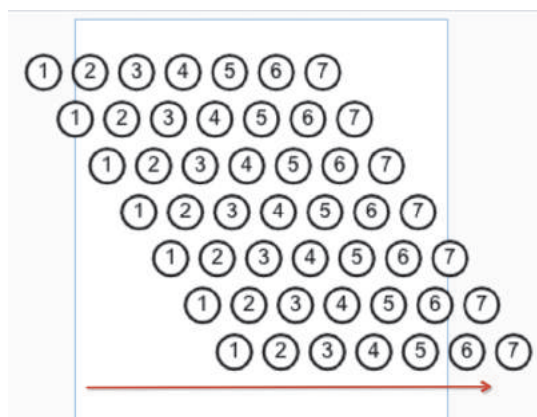
- Svg 导入、校验；
- 编辑场馆信息：可对舞台、楼层、看台、区域和场馆入口等进行设置；
- 添加座位：主要针对座位的物理属性进行设置，包含但不限于座位类型、所在区域、楼层信息、排数、列数、排序规则；
- 块级座位变形：间距调整，可通过座位  $x$ ,  $y$  轴的数值进行调整；
- 旋转：将所选座位集合设置成为一个临时块，将其作为矩形矩阵。向上取整，获取离矩阵中心最近的座位，将其设做圆心，其他座位以圆心座位进行 `rotate` 旋转；



- 错位：根据角度进行奇偶排位移错位；

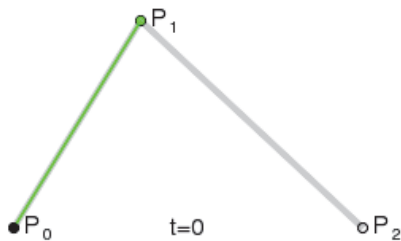


- 倾斜：将所选座位集合设置成为一个临时块，将其作为矩形矩阵。第一排不改变位移，从第二排开始，每一排根据上一排进行等比例位移；



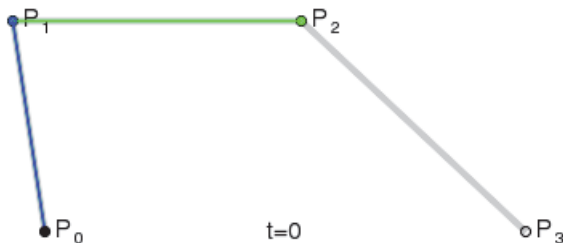
- 弧形：
  - 应用贝塞尔曲线：贝塞尔曲线是用一系列点来控制曲线状态的，我将这些点简单分为两类：数据点、控制点。通过调整控制点，贝塞尔曲线形状会发生变化；
  - 中弧算法：二阶曲线：

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2, t \in [0, 1]$$



- 左/右弧算法：三阶曲线：

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3, t \in [0, 1]$$

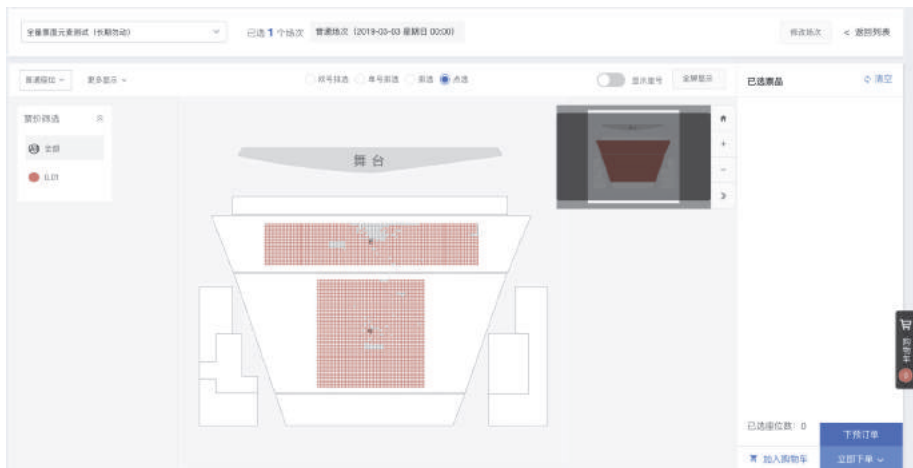


## 2. 选座

- 票房规划：规划票档、保留票、套票和打印座位图；



- 锁票：根据业务场景可分为单场次锁票和跨场次锁票，跨场次锁票可支持在相同场馆的不同场次内进行锁票；
- 门店销售：彩虹图、高级筛选、购物车和下单；



### 三、效率提升（按区域加载方案）

麦座的选座场景在两万以上座位量的大型体育赛事上，座位图页面加载缓慢、操作座位时出现卡顿等情况。

实现座位按区域加载，使用户在拖拽、缩放的同时惰性加载所需要展示区域内的座位，加快首屏渲染时间，在用户无感知的情况下加载其余区域的座位。解决因数据量庞大引起的页面加载缓慢、卡顿等体验问题。

技术实现（以规划票房场景为例）：

1) 展示当前浏览器窗口内所有看台/区域的座位按区域加载。

看台内座位的关键技术是获取到当前浏览器窗口内所展示出来的看台/区域。那么如何做呢？首先我们设定一个矩形区域，同步浏览器视窗的范围，然后可以通过 `getBBBox()` 获取矩形区域内部的看台元素，最后遍历看台区域并与矩形区域进行对比，可以根据业务需要来设定重叠多少面积触发加载内部座位的逻辑。

2) 预加载浏览器窗口附近的看台/区域的座位，保证在用户移动底图时可以更快速地浏览视窗以外的其他座位。

预加载的初衷是减少异步加载的耗时，提高用户体验。这里的预加载并非 `applicationCache`，而是一种巧妙的提前加载方案：如之前说过的如何实现获取浏览器视窗内的所有看台区域，在这个基础上我们可以在对比的过程中将实际的 SVG 面板缩小一定的比例，比如 80%，那么得到

的看台区域和视窗矩形重叠的部分会增加，更会有实际视窗外的看台被视作与矩形有重叠，就是这样我们可以提前拿到视窗外且遍布在视窗周边的看台区域，把这些区域提前加载出来从而实现预加载。

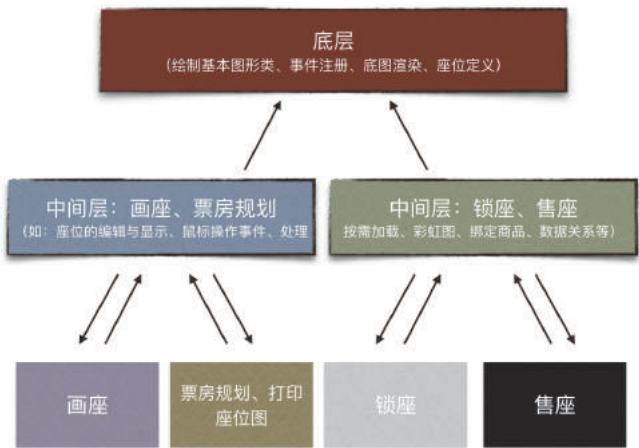
3) 以座位维度单点绘制座位。

早期麦座座位图场景绘制座位的维度是看台，也就是从服务端获取的数据是看台集合，而看台到座位需要遍历看台、区域、块、排。在浏览器渲染座位之前，做了很多数据包装的工作，而且操作的数据量是很庞大的，性能与绘制效率均受到了不同程度的影响。

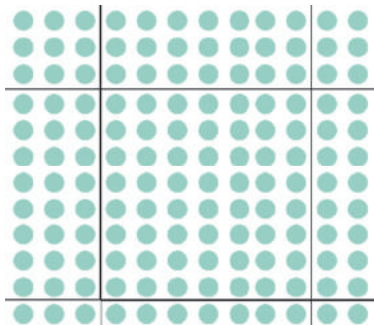
	升级前（全量加载）	升级后（按区域加载）
可支持座位量	小于 3 万	可支持 10 万
首屏渲染时间	10 秒以上	2 秒以内
组件化	无	技术组件封装，降低业务耦合度，降低开发、维护成本
数据结构	各场景不一致	统一座位、商品模型，统一各场景数据结构
CPU 性能	峰值 145.9	峰值 69.1
接口响应时长	10 秒以上	2 秒以内
6 万座位渲染总时长	几十秒	3 秒以内

四、架构设计

业界普遍采取的技术实现方案有三种：Dom 渲染、Svg 渲染和 Svg+Canvas 渲染。基本的架构设计思路如下：



本文解决方案采用 Grid 思路：将座位坐标进行按照一定范围，比如 2000（经测算这个范围 2000\*2000 比较合适）进行网化，也就是将元素坐标在 0 到 2000 的放到一个网格、2001 到 4000 的放到一个网格，以此类推。



## 五、性能提升

### 1. 座位图快照

- 目的：提高页面加载速度；
- 产物：利用 canvas 快照能力，将非焦点区域的座位图以图片的形势展示出来，减少不必要的 canvas 渲染。

### 2. 全场景数据打通

- 目的：加载与渲染速度提升；
- 产物：画座、票房规划、单场次/跨场次锁座、门店销售、打印座位图打通获取场次、票档信息、座位数据的结构，座位由原来的看台维度的加载变化成座位单点加载，规避了大量的深层 map 结构解析与数据包装。

### 3. 局部异步渲染

- 目的：降低引起重绘的 cpu 使用率；
- 产物：设置票档、保留票、套票、高亮等操作，将所选中的座位设置成临时方形矩阵，重绘矩阵内部的座位。

### 4. Canvas 拆分

- 目的：减少页面加载渲染卡顿；



- 产物：Canvas 宽高大于一定数值（经过测试这个数值在大多数浏览器下是 5000）时，就会出现卡顿问题，demo 之所以不会比较流畅，是因为模拟的十万座位数据比较密集，Canvas 没有设置到那么大，将多个小的 Canvas 拼接，然后再绘制座位，按照座位坐标确定座位该绘制到具体的 Canvas 上。

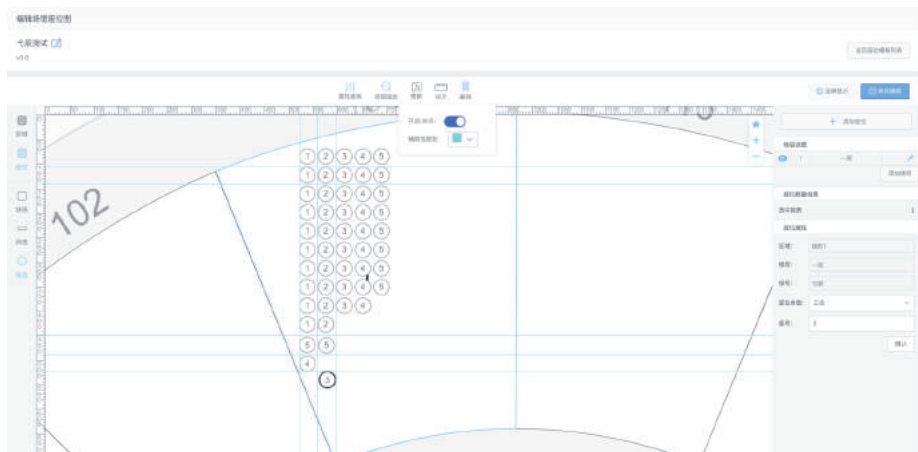
## 5. 组件化沉淀

- 目的：提高多处相似能力的可复用性，降低技术成本。低耦合，高维护性；
- 产物：鹰眼图组件、座位变形组件、票房规划相关组件、创建座位组件等等。

## 六、辅助工具

- 1) 圈选：通过鼠标右键圈选自定义矩形块，选中矩形块内的座位集合；
- 2) 套索：类似于 Photoshop 套索工具，可以通过打点完成不规则图形的圈选，选中多边形区域内的座位集合；
- 3) 增量选座：增量添加已选座位，可以通过不同的选座工具做增量添加；
- 4) Svg 校验：团队内部根据一套完善的 Svg 规范定义，对新绘制的 Svg 进行校验，目的是过滤掉一些不符合规范的 Svg 底图，避免到线上出现比例有误影响缩放效果和底图显示出现问题的情况，同时避免出现因为 Svg 底图的问题导致已生产完毕的场次重新生产的情况；
- 5) 标尺。

辅助绘座对其排、列的标尺工具，可随意切换颜色。





## 七、新领域：“全景、3D”思路与初探

现有的一些技术资源介绍：

1) ThreeJs: 被称作 3D 大哥的 threejs 封装了 webgl 很多核心方法, 通过做三个核心“场景(scene)”、“相机(camera)”和“渲染器(renderer)”完成 3D 效果, 需要自己造轮子、自己搬砖, 看案例性能上需要做很多优化工作, 坑也需要自己一点点去踩;

2) Aframe: 入手相对容易, 还可以支持 VR 模式观看, 但比较鸡肋, 不支持放大缩小, 移动端视角不支持移动;

3) jQuery-VRview: 开源插件很多, 但功能大多都未必不合适我们要的场景, 需要探索一下资源库碰碰运气;

4) Pannellum: 国外的一个插件, 图片宽度最大限制 4086, 图片放大后有失真、像素感严重的情况, 尤其是移动端;

5) Krpano: 但是据说很好用, 市面上很多全景 VR 公司用的都是这个。缺点是本地生产, 如果需要上传, 有 linux 版本, 可能需要 java 帮助, 是收费的。

3D 全景展示目标看台的视角/视野效果: 图片资源要求比较严格, 可能需要专业的设备和“摄影师”, 具体是受 webgl 规范影响, 图片的宽度有要求, 所以涉及到图片的裁减与拼接, 图片质量直接影响分辨率和抗锯齿程度。

用户视角: 鼠标悬浮展示当前看台的视觉效果, 主要是图片展示给用户一个直观感受, 类似于国外网站 seatsMaster 的效果。

# 大型赛事稳定性保障：Dpath 为世界军人运动会护航

作者| 阿里文娱技术专家 司楚

## 一、背景

第七届世界军人运动会是中国第一次承办综合性国际军事赛事，有 100 多个国家、一万多现役军人参与竞技。武汉军运会票务系统建设和票务运营由大麦网承接，使用阿里集团基础技术能力，通过大麦麦座票务系统提供票务管理、售卖和现场换验服务。

大麦麦座票务系统（简称“麦座”）定位于现场娱乐的一站式行业服务平台，结合阿里的大数据、云计算等能力，为场馆向数字化、智能化的升级提供行业解决方案。麦座基础架构经历了从单机软件部署到阿里云的商户独立部署，再到 SaaS 化的演进过程。

目前麦座系统利用阿里巴巴的 pouch 容器，采用微服务的架构，为大量商户提供统一的服务。多用户共享服务的架构，实现了资源利用的最大化，方便版本的统一维护和升级。当承载军运会等大型赛事，或者面对头部 KA 客户时，从安全、稳定性角度考虑，需要做独立部署和流量隔离，从而避免突发流量影响到 KA 客户。

## 二、目标

针对军运会的场景，我们需要满足：独立部署、流量隔离。在满足独立部署和流量隔离的基础上，需要满足：

1. 对开发尽量透明：尽可能少的定制和代码侵入，对开发透明，不能因为独立部署增加额外的开发和运维成本；
2. 独立部署的容器与普通分组的容器完全等价，且可以动态的切流：某个环境宕机的极端情况下，能够快速切换，保证业务正常运行。

### 三、方案选型

为了实现独立部署和流程隔离的目标，同时支持动态切流，我们调研了 单元化、独立应用、环境配置项、通过流量打标进行路由的方式。

	单元化	独立应用	环境配置项	流量标路由
介绍	独立机房部署，通过域名等信息进行路由；可动态切流；实现了存储隔离和异地容灾	单独的应用和代码库，可实现多版本的共存，无法代码复用	同一个应用和代码库，独立的分组，对应独立的配置项。通过配置项维护 RPC 版本号等信息，实现隔离	根据商户信息，对流量打标。RPC 调用、消息队列等根据打标信息进行路由，形成分组的隔离
部署成本	需要独立的单元化环境，成本高	独立应用需要测试、预发、线上环境，成本偏高	申请独立的分组，成本低	申请独立的分组，成本低
开发成本	对代码无侵入，成本低	维护多套代码，成本高	新增配置时需要修改多套配置项，成本偏高	RPC、消息队列等中间件支持，对开发完全透明

通过对四个方案的比较，我们最终选择流量打标路由的方式。阿里巴巴中间件 Dpath(Dedicated path)，在请求中添加流量标识，通过 Diamond(持久配置管理中间件)把 Dpath 的打标规则，推送到各个应用容器。RPC、消息队列等中间件根据流量标和 Dpath 规则，进行路由，选择相应的应用分组。

1. Dpath 实现的需求：

- 1) 针对特殊流量可以圈定一些特殊机器作为他的专属的服务器，以便对特殊流量进行特殊保障或者测试；
- 2) 普通流量不应该使用专属服务器，特殊流量可以按需使用普通服务器；
- 3) 整个链路上的专属服务器组成了特殊流量的专属通道，类似公交专用道。

2.基于 Dpath，我们需要实现：圈定专属服务、识别特殊流量、在链路上引导流量到对应的服务器。

### 四、HTTP 流量路由

我们为每个应用申请独立的应用分组，专门为军运会提供服务(下文称为军运会分组)。然后在 Dpath 创建独立的环境，选择对应的分组，组成军运会的集群环境。通过统一接入层，为

军运会分组配置独立的域名。在 **VipServer** 进行配置，军运会域名的请求关联军运会的分组。当某个分组宕机等原因导致服务不可用时。修改 **VipServer**，把域名的关联切换到可用分组，保证流量的动态切换。

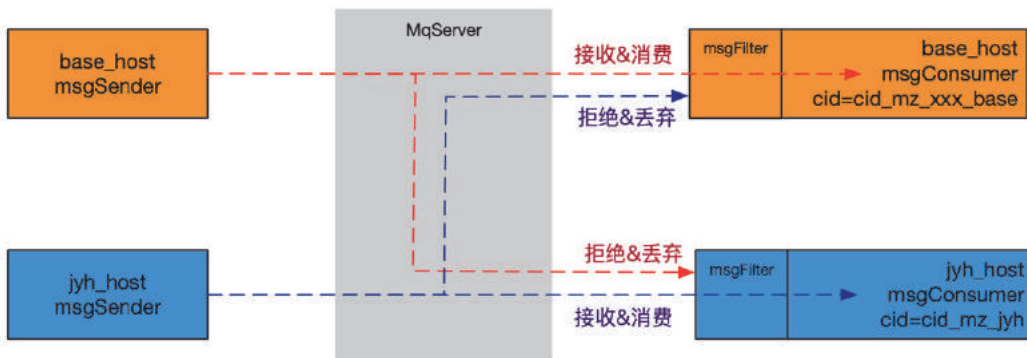
## 五、RPC 流量路由

RPC 的流量可以分为麦座内部系统的相互调用和麦座外系统的调用。针对麦座的内部流量，当请求经过某个分组时，自动在链路的上下文中打标。常见的 RPC 中间件(如 **Dubbo**)支持路由规则的配置。针对外部的流量，在 **Diamond** 中配置路由规则。**Diamond** 把 **Dpath** 的路由规则推送到各个容器，包括流量标识和应用分组的映射关系。RPC 中间件根据 **Dpath** 规则和流量标识，选择对应的分组。

```
class DPathTagRule {
    boolean isOverwriteOn() {
        // 是否使用当前规则进行覆盖，默认为 True
        return true
    }
    // 接口/方法级别规则
    String methodTagRule(String serviceName, String methodName, String[] paramTypeStrs) {
        return null
    }
    // 参数级别
    Object argsTagRule(String servicenName, String methodName, String[] paramTypeStrs)
{
    return {
        Object[] args ->
        if (JYH_TENANT_ID.equals(args.getTenantId())) {
            return "jyh"
        } else {
            return null
        }
    }
}
```

## 六、MQ 消息路由

Dpath 支持消息队列的路由。Dpath 通过为消息消费者(Consumer)配置消息 Filter 方式实现。不同分组的 Consumer，注册时在用户配置的 ConsumerId(cid)基础上添加后缀。普通分组和军运会分组都会接收到所有的消息。消息的生产者(Producer)发送消息时根据自己的机器分组，在消息体内打标。Consumer 在消费消息之前，添加消息的过滤器。根据 Consumer 自己的机器分组，过滤掉非同分组的 Producer 发送的消息，仅消费同分组的 Producer 发送的消息。



## 七、效果

通过 Dpath，我们用低成本的部署，实现了独立部署和流量隔离。保障业务功能快速升级，同时兼顾最低维护和运行成本，保障流量和风险隔离。整个军运会从上线到闭幕期间，整个售卖过程非常顺利，无任何问题和故障。

# 世界顶级赛事的票务支撑：百万座位与限时匹配

作者| 阿里文娱技术专家 展恒

## 一、背景

麦座，是大麦旗下的票务系统。去年，我们承接了2019年国际篮联篮球世界杯(2019FBWC)，核心目标是完成三种套票的运营及售卖。用户可选择的套票方案为：

1. 球队套票：可以观看指定球队比赛；
2. 城市套票：可以观看指定城市比赛；
3. 单日套票：可以观看指定日期（同时也指定城市）的比赛。

要知道，2019FBWC 有 92 场比赛，共计 100 多万的座位，两场比赛间最短时间间隔仅有 10 多个小时，只有上一场比赛打完，才能确定下场比赛对手和城市。我们的技术难点是，面对套票本身对阵关系及场次的不确定性，如何在百万座位中，快速为套票用户匹配座位。以下是我们的技术解法和总结，希望对大家有借鉴。

## 二、套票解法

### 1. 套票问题抽象

我们按照目标场次是否存在，把三种套票分为两类：

1) 球队套票定金、城市套票、单日套票。特点在于目标场次一定存在，但是对阵关系和座位图不确定，可以抽象为无座场次下单模型，即在不确定具体场次和座位情况下通过售卖数字库存，为用户锁定某种资格（入场、后续购票等等）；

2) 球队套票后续阶段。特点在于目标场次不一定存在，败者球队不一定有后续比赛。但是随着赛程推进，一轮打完必然知道下一轮对手以及比赛城市，此时只有已经交了球队套票定

金的观众可以继续以较便宜的价格购买球队套票下一阶段的门票。

针对球队套票赛程阶段，我们需要提供的是锁定资格后不断购买指定球队后续比赛场次门票的能力，即给客户“配单”。而无论何种套票，用户最终还是要进场观赛，需要要在验票入场前告知用户实际座位，因此我们提供将无座订单转化为有座订单的能力，即给客户“配座”。

为了解决配单、配座问题，麦座设计了两个工具：

- 1) 配单工具：球队套票，每个阶段之间通过工具，生成下一阶段待支付订单，延续用户购买套票资格；
- 2) 配座工具：具体比赛场次出来后通过工具，将无座套票映射到有座场次，给用户机选座位，最终实现用户座位分配。

## 2. 解法一：配单

配单功能适用于球队套票的分阶段售卖，世界杯的球队套票分为六个阶段。

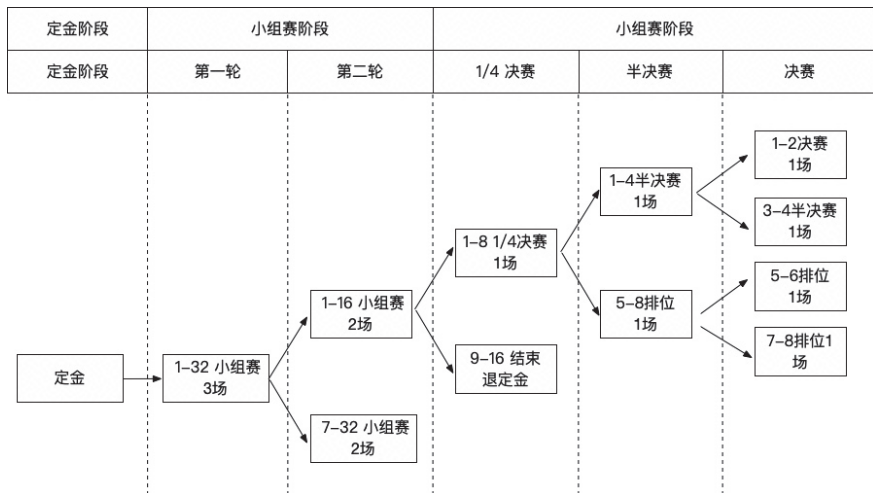
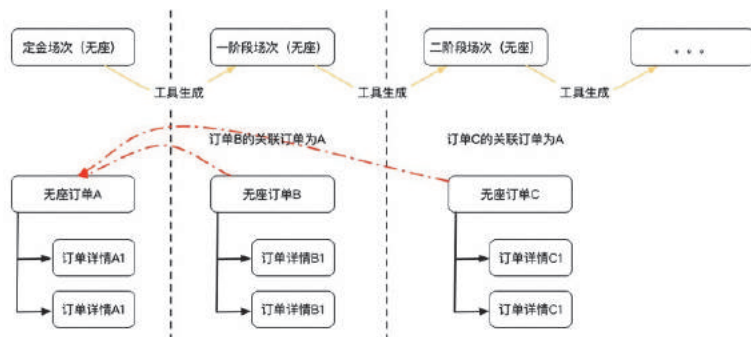


图 1 赛程抽象

第一阶段为定金支付，支付定金后，获得后续球队套票购买资格，后续共分为 5 个阶段，每个阶段对应小组赛到决赛的实际场次。配单工具要解决的核心问题在于：

- 1) 利用当前阶段的订单信息生成下一阶段待支付订单，同时通知用户支付；
- 2) 保持套票订单之间的关联关系，方便报表统计。



如图所示，配单时基本上是按照原始订单的订单结构信息，复制出目标订单的结构信息，订单结构保持不变，同时所有后续阶段的订单都与最原始的定金订单建立关联关系，后续通过定金场次订单可以统计出套票生命周期的所有订单，方便报表统计。球队套票每个阶段都生成一个待支付的无座订单，并提醒用户及时支付。

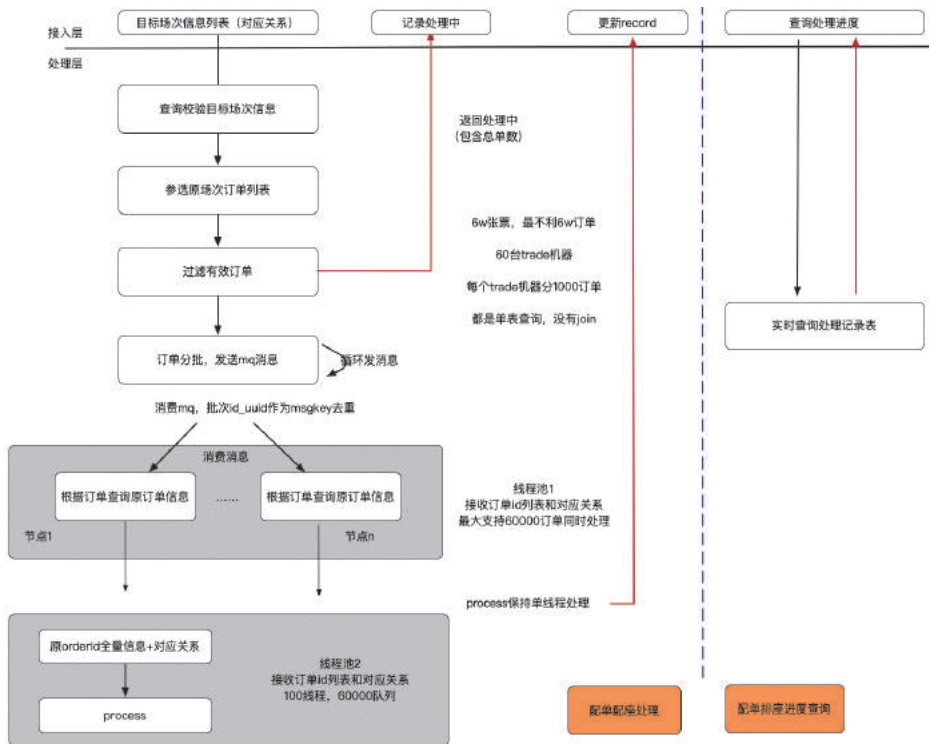


图3 处理流程



配单处理流程如图所示，由于处理时间较长，为了更好的用户体验和性能，需要在订单开始处理后及时返回用户“处理中”的状态，后台采取异步处理。通过消息中间件将处理任务分发到集群处理节点，另一方面由于底层下单接口 TPS 承载有限，需要每个处理节点（应用实例）控制处理速率，做好限流，图中是通过线程池控制处理速率。

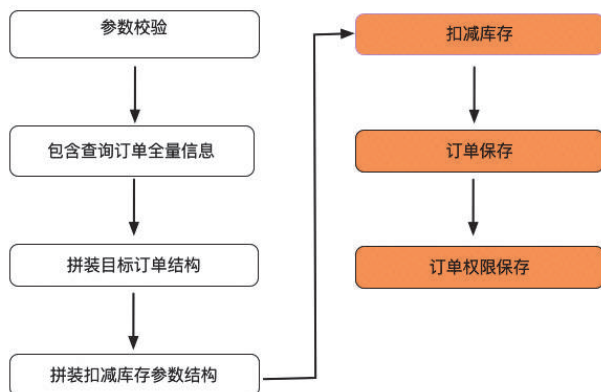


图4 process 分解

单个 process 的内容如图所示，其中橙色框中的步骤标识有数据库写操作，需要做事务处理。扣减库存、订单保存、订单权限保存基本上是复用的麦座下单 process 中的的 step。

### 3. 解法二：配座

配座工具适用于所有套票映射最终有座场次的场景。从用户体验上讲，配座与配单不同，配单工具生成的订单需要用户支付，因此必然让用户感知到新订单的生成，而配座工具生成的订单实际上不需要用户支付，而且由于配座的场次非常多，会出现一个原订单生成最多十个目标订单的场景，因此如果让用户感知到其实是不合理的（考虑一个用户某天打开订单列表，可能看见若干莫名其妙的订单），所以订单结构上配座与配单也不尽相同。

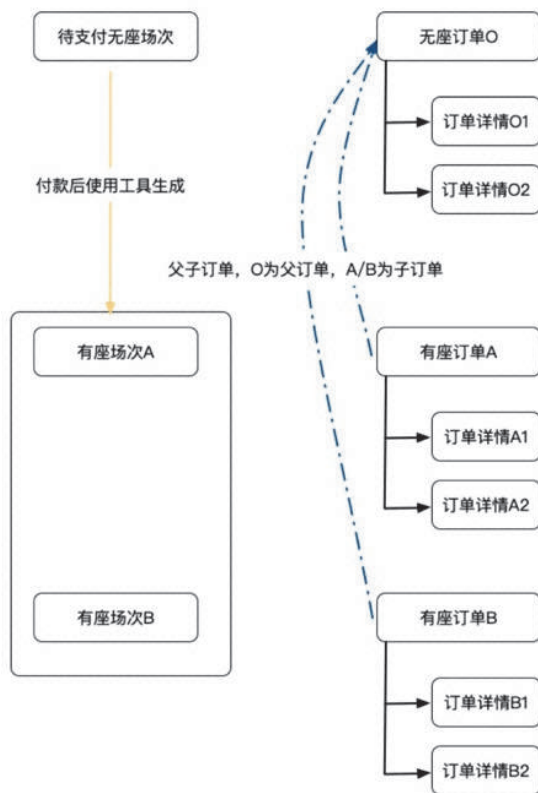


图5 配座模型

配单工具通过关联关系记录套票生命周期，而配座工具通过父子关系，更偏向于传统的购物车订单，即父订单收钱，子订单无需再次收钱。套票无座订单记为父订单，实际的有座场次与套票订单建立父子关系。另外由于配座时不需要用户支付，因此子订单都记为零元订单，以免后续报表和对账结算金额无法抹平。

球队套票每个阶段、城市套票、单日套票最终要映射到有座场次，座位的选择是通过机选实现，配座流程与配单流程大部分是可以复用的，不同之处在于配座需要为用户机选座位，因此需要插入机选步骤，如图6所示。同样，图中的橙色框标识有数据库写操作，需要做事务处理。

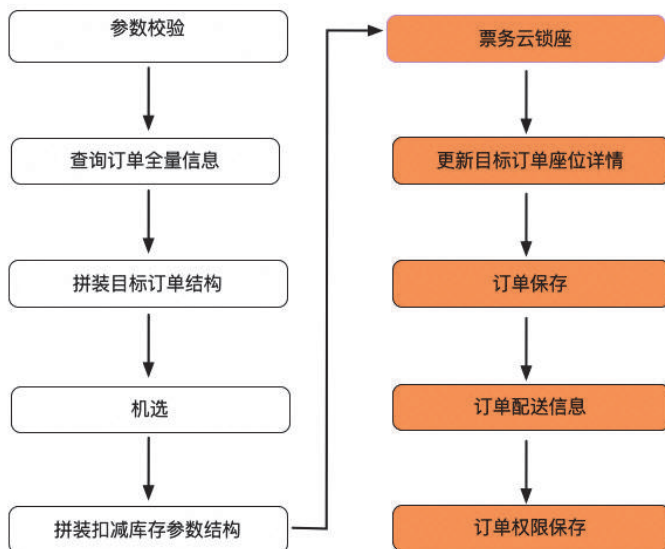


图6 process 分解

配座是麦座与大麦票务中台融合的一次实践，也是大麦多年以来票务能力的资源整合。未来大麦票务中台的机选能力可以开放给更多的业务方使用。

## 5. 如何做的更加通用？

设计之初，我们考虑到了业务的通用性，虽然不同票务系统的差异，不太可能通过一套实现满足麦座、世界杯、票务云的需求，但是玩法是可以借鉴的，通过一套接口流程，可以将套票玩法沉淀下来，后续其他票务系统有类似的玩法需求，仅需要实现接口就可以方便快速的克隆套票玩法，因此我们将接口定义抽取了独立的业务插件（jar 包）。另一方面，使用插件而不是独立应用，是因为配单和配座不可避免的要去做权限校验，jar 包可以复用接入系统的权限体系，这样工具只做核心的玩法部分，可以更聚焦。

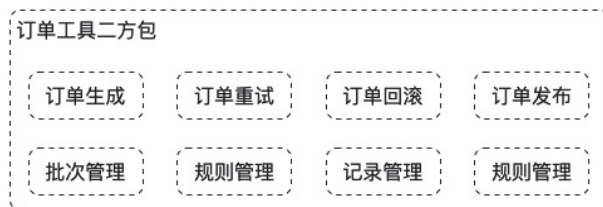


图7

### 三、效果以及未来展望

2019FBWC 总共 92 场比赛，共计进行了 5 轮配单配座，通过后续的业务数据检查，所有座位配置正确，无一错漏，配单配座工具将运营同学原本一周甚至更久的工作在十几分钟内搞定，达到了产品要求的准确、高效，效果得到业务方的认可。

大规模的配单配座在体育赛事中可能成为行业常态，事实上 2019FBWC 的玩法也同时得到了亚运会、冬奥会等赛事的关注，配单配座工具采用的技术并不是阿里特有，同时由于我们采用插件设计，也在一定程度上实现了可复用，方便的推广到其他票务平台。

# 前端技术：Webpack 工程化最佳实践

作者| 阿里文娱前端开发专家 芄苏

## 一、引言

### 1. 前端构建工具的演变

回想在 2015-2016 年的时候，开发者们开始渐渐把视线从大量使用 Task Runner 的 Grunt 工具，转移到 Gulp 这种 Pipeline 形式的工具。Gulp 还可以配合上众多个性化插件（如 gulp-streamify），从而使得整个前端的准备工作链路，变得清晰易控，如刷新页面、代码的编译和压缩等等。自动化“流水线”工具取代了很多繁杂的手动工作，可以说，是具有跨时代意义的。之于 Webpack 而言，其本质是是基于“模块化”思想的一个“JS 预编译”解决方案，诞生初期，和其相似的方案还有 Browserify，和 Webpack 属于同门不同派别的还有 sea.js 或 require.js，这二者需“在线依赖”解释器编译。

时至今日，多数日常工作接触的项目，已经可以完全的舍弃 Gulp 了。但工作中有时还会接触一些老项目，其中 Gulp 的使用和维护屡见不鲜。2019 年初之时，通过一个老项目（gulp 3.x + webpack 3.x）的技术升级，借机了解了 gulp 4.x 的动态，又不禁让人回想起 gulp-browserify，和 gulp-webpack（五年前发布，目前改名为 webpack-stream）。所以，Webpack 做为某一个垂直方向的解决方案，当然可以 manually built-in Gulp 中。在拿 Webpack“方案”和 Gulp 类“工具”去做正面比较的时候，需要明晰两者解决问题的范围和思路。如今再次回顾历史，对技术的发展演变顺序，能有一个基本客观的概念。

在 2017 年的时候，Gulp 和 Webpack 在用户的使用率和“将继续使用”的意向上，还不分伯仲。但从《State of Javascript 2019》中可以看到，Webpack 已经完全碾压了其它工具和类库，成为了首屈一指被大家广泛使用、讨论的 Build Tool。2018 年 2 月 25 日 Webpack 发布了 4.0.0 正式版本；对不少项目进行了 Webpack 4.10.2 版本的升级后，又将部分项目升级到了 4.29.0 最新版本。这一系列的“跟进式升级”中，一方面是在不断融入 Webpack 对于模块构建的新思路

和理念，为了能够更好的适应其未来的变化，另一方面是在一个好的方案中不断尝试，结合项目的基础设施优化，从而提高效能，保障产品稳定。

## 2. 本次回顾

Webpack 工具虽说只是前端项目 CI 流程的一个小部分（构建 build），就它自身而言，所涉及到的 Node 知识和包依赖管理经验，是一整块技能。细节来看，里面涉及了 Webpack 自己的包和第三方 plugin 生态，还要配合恰当的 babel、typescript、flow.js、eslint 配置等多个生态，去处理 Javascript 语言本身的编译/转译。以及，正确管理本地静态资源文件和远端 CDN 资源文件路径（打包配置决定打包结果），涉及到了跨域知识和 Node 层服务配置、模板配置知识。更进一步还有，NPM 众多包的版本管理等让人头疼的问题。其中琐碎细节数不胜数，当所有第三方工具正确使用的前提下，也许还有些 plugin 小工具，需要开发者去自主研发。知识谱系之大，可见一斑。

本文不描述 Webpack Docs 使用指南，也不描述第三方插件的使用“指北”。更多的是结合过往项目经验，记录实践得出的使用技巧，也记录一些走过的弯路所带来的问题，希望对其它众多的前端技术人能够起到一点借鉴作用。（Package Checking List: React: 16.3.2, Babel: 7.0.0, Webpack: 4.29.0, Node: 11.8.0）

## 二、文件结构

在 4.x 版本中的早期，CLI 工具集里的命令是 Webpack 主包自带的，但在 Webpack 4.x 后期的版本，将 webpack-cli 作为独立包剔除出去，需要手动单独安装才可以执行 `tnpm run start` 这样的脚本命令。其次，对于开发/日常环境（dev）和预发/生产环境（prod）来说，打包的策略是截然不同的：

### 1. 对于 dev 日常环境：

- 1) 方便的 debug 和 troubleshooting，有比较强的 source mapping;
- 2) 希望能够得到颗粒度较小、且有根据变动代码针对性的的加载（live reloading/hot module replacement）;
- 3) 希望可以做一些代理 Proxy 相关的调试;
- 4) 可以方便的根据开发者的情况，对本地的 dev-server 进行配置等。

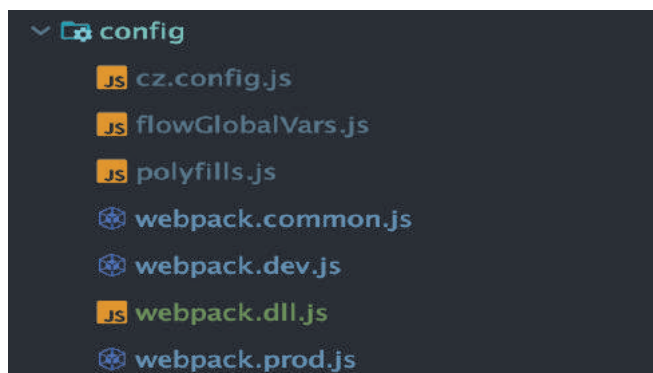
## 2. 对于 Prod 生产环境：

- 1) 通过压缩 Javascript/CSS 代码，获取更小的文件加载体积；
- 2) 通过包的拆解来得到更优的加载策略，从而降低 load time；
- 3) 比较轻量的 source mapping（当然，当你需要一些 trace 信息做日志和报警的时候是另外一番情景）；
- 4) 线上的产品的一些个性诉求（比如，对同一份 Javascript 代码也许要匹配不同的样式文件）等。

## 3. 通常评估效率维度主要有以下几个，文中提到的数据来源主要属于前三个：

- 本地开发 compile(w/ DLL or NO DLL)；
- 本地开发 re-compile(w/ DLL or NO DLL)；
- 本地测试 build（webpack analyse 分析的重点部分）；
- 云构建时长 (NO DLL or 配置化 OSS 支撑 DLL)。

在 Webpack 的新版本中，webpack-merge: 4.2.1 这个独立包的使用，开发者使用 webpack.common.js 文件对开发和生产环境中的公共部分进行配置，webpack.dev.js 针对开发环境，webpack.prod.js 针对生产环境。区分后，两种环境的配置差异，一目了然：



（图：webpack 配置文件结构）

关于 cz.config.js 和 flowGlobalVars.js 里面“话题点”颇多，不在此处重点描述。

如果需要 DLL 配置(在后面的优化部分会重点讲)，还需要单独加入一个 webpack.dll.js 打包的配置文件。当然，dll 其实也是一个普通的文件 Output，我们可以在 webpack.common.js 文件

中 `module.exports` 时，写两个区分开。通过这种不是很常见的灵活写法（Exporting multiple configurations），可以更多的去理解文件的 I/O 和 `module` 模块的概念。

## 三、基础/自定义配置

### 1. CommonsChunkPlugin 被取代

被移入到了 `webpack.optimization.splitChunks` 中。有关拆包切分和颗粒度控制，这个其实从 Webpack 的层面已经为我们做了很多优化，自身也是有一套基础默认的优化策略的。类比来看，React 生态里面 `diff` 算法本身也是有策略机制的，更多的优化，使用者可以在这个对象里面加入回调方法，自己去细化控制。

这里需要特别注意的是 `cacheGroups`，当不明确哪些内容需要被 `cache` 时，或者是颗粒度不好把控时，这样的切分会给我们带来非常多的冗余文件。定义一个 `vendors` 对象，那么我们的 `output` 文件（不包含 `chunksFiles`）的每一个都会生成一个 `cache` 文件。加入 `output` 的有 `app.bundle.js` 和 `polyfill.bundle.js`，一旦加入这个 `vendors` 对象，打包的时候会额外的生成两份文件，分别是 `vendors-app.js` 和 `vendors-polyfill.js`。虽然不用担心这两个文件内容会重新打包代码进去，里面只是放一些 `cache` 索引，但这两个文件如果在不确定要用他们来做什么的时候，`cacheGroups` 的设置，需要重新认真去考虑。

### 2. OccurrenceOrderPlugin

本身不再是一个 `webpack` 类下面的构造器，而是被重新命名（之前的名称因为单词拼写错误了），然后放入到新的位置，调用起来需要重新去书写：`new webpack.optimize.OccurrenceOrderPlugin()`。

### 3. terser（默认的内置压缩工具包）

`webpack.optimization.minimizer` 的新版本中，`default built-in` 的工具已经由旧有的 `uglifyJS` 变成了 `terserJS`，旧的 `uglify` 已经被 `depreacted` 处理，相信不久之后的状态就会变成 `legacy`，新的 `terser` 更好的性能，对 ES6+ 的语法支持的更多，也同时兼容了 `babel 7` 的生态，同步其它第三方库代码压缩后的诉求。目前我在使用的是 `terser-webpack-plugin`，和普通的 `terser` 配置的参数上有一些差异，需要自己手动引入（官方文档推荐）。



#### 4. module.rules.exclude[0]

module.rules.exclude[0]的文件地址书写，要求更加严格（4.11.0 以后的版本）。

以往我们在对 module.rules 做配置时，有些文件不希望被遍历到，那么我们通过 exclude 这个参数配置，将其跳过，有时候会使用'src/contianer/xx.jsx'这样的写法，如果是多个 path 索引，那就放到一个 Array 中就好。但这种写法，在新版本中是不被允许的，我们只能使用 path.resolve() 或/RegExp/的写法去声明文件路径地址。（Bonus Basic Tips，如何用正则书写并集和特定路径，如我希望 include 所有 src 加上一个指定的 npm 包：  
/(src\\.\*)|(node\_modules\\.\*@ali\\lark-components)/）

#### 5. alias 和绝对路径

webpack 在打包的时候，通常需要对文件的路径去做查找、搜索，它需要明确知道文件的引用位置和引用关系，从而能够完整的知道整个映射 mapping 关系。减少这方面的开销，我们可以考虑去配置 alias，从而以绝对路径的写法代替大量相对路径写法。好处的话，一方面是帮助 webpack 更快的去定位文件位置，另一方面书写起来，也不用再用被输入 '../..'/\* 还是 '../..'/\*' 而困扰。

- Webstorm 寻找绝对路径：在配置里面对 webpack 配置项加入 webpack 文件路径就好，Webstorm IDE 会自己找到对应的 alias 关系；
- VSCode 寻找绝对路径：插件层面没有发现太好的办法，如果项目正在使用 typescript，可以在 tsconfig.json 里面配置相关的编译项，可以达到和上面 Webstorm 同样的效果。

#### 6. 大图片上传 CDN

上传 CDN 后可以大幅减小包体积。另外，webpack 也不需要再去关注那些图片的文件索引路径了。项目稍微大一些，本地图片 5Mb ~ 10Mb 的情况非常普遍，亟待优化。

#### 7. devServer Proxy 的代理能力

去调研这个能力，得益于一次请求层的改造。诉求是希望 Token 不再显示传递，而是通过塞到 Header 去实现。在本地开发的环境，我们通常使用 jsonp 去解决跨域问题，但其本质其实是在网页中嵌入一段<script />，自然也就不能写入 Header 信息，这个和我们的初衷并不相符，无法满足诉求。所以对于这样的跨域问题，我们通过几个简单的参数配置，在请求发起和请求返回的两端，分别做了代理配置，从而“欺骗”了“源 Origin”，得以解决本地开发的跨域问题：

```
devServer: {
  // ...
  headers: {
    'Access-Control-Allow-Origin': '*', // CORS
  },
  proxy: { // for ajax cors
    '/h5/ajaxObj': {
      target: 'http://xxx.xxx.xxx.com',
      onProxyReq: (proxyReq) => {
        proxyReq.setHeader('Origin', 'http://xxx.xxx.com');
      },
      onProxyRes: (proxyRes) => { // ...
      },
    },
  },
},
```

## 四、优化性能 by Node / Happypack

基础配置和需要的自定义配置已经有了，整个项目的构建时间有可能还是非常不理想的，当前本文提及的测试项目，大概有 57s 的时间，还是有很多地方没有补足的，可优化的空间非常大。

第一步可以先关注下 Node 版本，经过测试，是对整体速度可以至少提升 30%的事情，尤其是在 Node V8 版本到 V10 的时候，以下是之前在另一个项目做技术改造时记录到的数据：

Node 版本	v 8.x	v 10.x
compile	32s - 36s	26s
re-compile	8s - 9s	4s

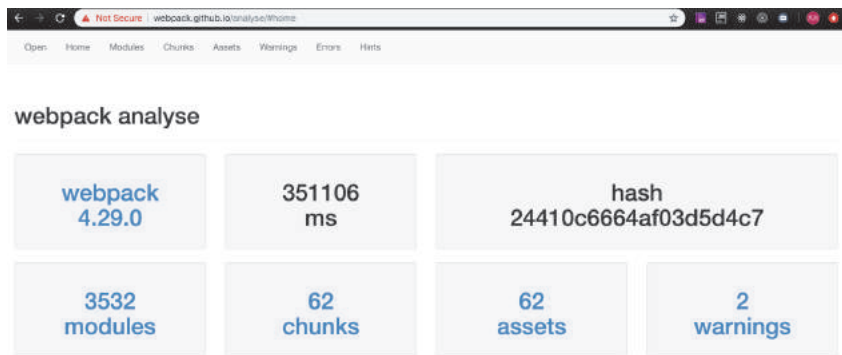
但是这次，在把项目直接升级到了 v 11.x 后发现，有带 node-sass 的项目编译构建都崩溃了。才意识到，node-sass 的版本也需要相应的版本更新。也测试了 Babel v 6.x 到 v 7.x 版本的升级效果，本来以为 babel 的大版本升级会带来显著的编译速度提高，实际上却并不理想（基本可以忽略不计）。

打算开启多线程能力，去处理模块化打包里面那些本是单线程执行的 loaders 们的工作。Happypack 的提升效率对整个项目的首次编译而言，效果是 20%左右，比较明显。加入 Happypack 能力的时候，有两点需要注意：

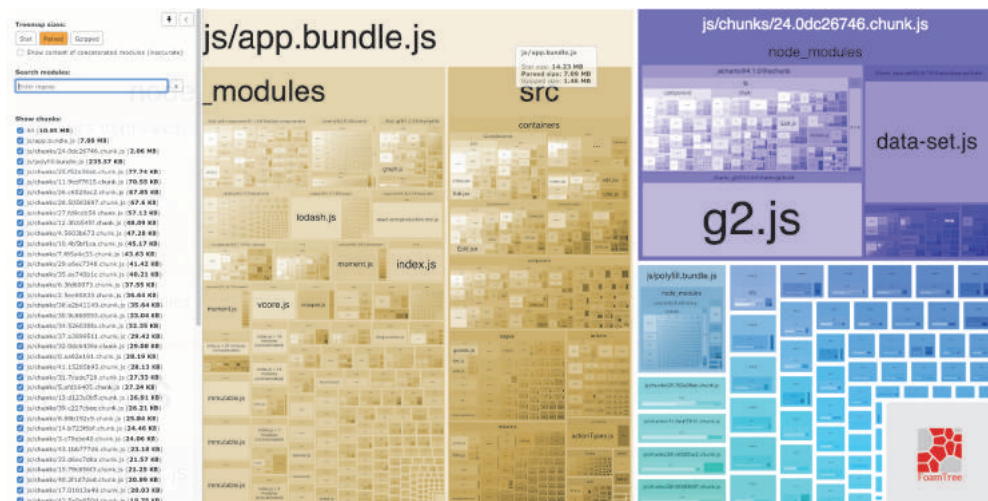
- 其对 file-loader 和 url-loader 的支持不好，可以考虑不加，毕竟我们项目里面图片类（最好上传 CDN）的和非常规格式的文件只是小部分；
- 这次也尝试了把 ts-loader 加入到多线程中，但是也出现了不少编译问题。大概率怀疑是我个人的配置问题，但过程中去看 issues 见到了不少 ts-loader 和 ts 生态依赖兼容性的问题。目前这个项目.ts 只是少数文件，作为一种尝试，大部分文件还都是.jsx 和.js，所以针对 ts 也先不加入 Happypack 能力了。

## 五、优化性能 by DLL/ Optimization

首先需要借助一些工具来进行分析，如：webpack-bundle-analyzer，通过这个工具我们可以对整个构建（用于生产，Webpack Analyse 针对的 build 过程，不是 compile）过程和结果进行数据、图形上的分析，从而得知问题具体出现在了哪里。进而得知 DLL 所需拆分的内容是什么。以下内容是在第一次分析时得出的：

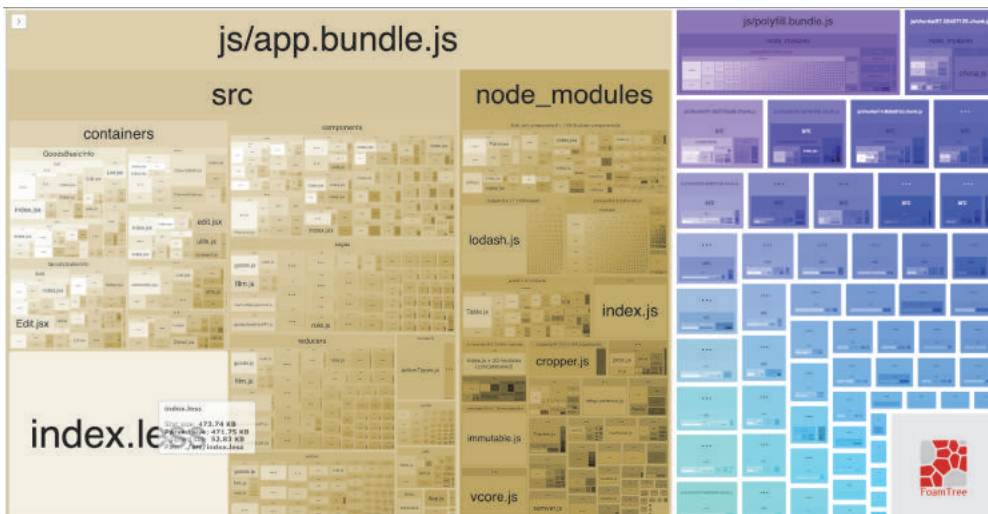


这个图片的 3532 modules 和 62 chunks 可以看到具体的模块以及 chunks 划分后的情况。更加直观的我们来看下面这张图，可以看到Parsed的尺寸，入口文件(7.09MB)和主 chunk(2.04MB，主要是一些首页就需要加载的 node\_module)的大小都很夸张，并且 node\_modules 里面的包基本上是一一打包、整整齐齐：



有了这些分析结果，对应解法的思路就很清晰了：首先要抽离常用的 `node_modules`（这是 DLL 的意义），然后要逐个分析，把不被经常用到的 `node_module` 们（仅被某些页面使用，不具有公共特点）也抽出去。

对于 React 项目中的 React, React-Dom, React-router, Redux 等，还要一些第三方比较大的库，比如 antv 或者 G2 相关的，也要进行 DLL 抽离了：



（modules 数量由 3532 降低到 1500，编译时间缩短了三倍）

在做了上述 DLL 的抽离后其实效果已经很明显了,进一步的提升空间,可以对 optimization 进行了配置 (用法详见官方文档):

- 1) terser;
- 2) chunksAll;
- 3) no minimizer sourceMap。

## 六、结尾

本文大概主要介绍了一些工具衍变背景、基础的组织结构和自定义配置,以及如何通过分析工具去来做性能优化,其中很多小的细节没办法一一提到,比如我们看到加载的 chunk 都是 hash 值的时候,如何能够辨别是什么组件呢:解法是在路由处通过配置 moduleName 的方式去做:

```
() => import (/* webpackChunkName: "chunkNameDisplay"
* /'../containers/UserList/chunkNameDisplay')
```

诸如此类,实在繁多。随着 Webpack 5.x 版本的陆续发布和众多团队使用之后,也许很多东西又会有大的改变。并且各种框架的集成已经越来越丰富,更多的解放程序员在工程化维护上的双手,我们关注工程化的演进,看看 Webpack 生态会给我们带来什么样的惊喜。

3

宣 发



## 电影票房数据查询服务高性能与高可用实践

作者| 阿里大文娱高级开发工程师 奋氛

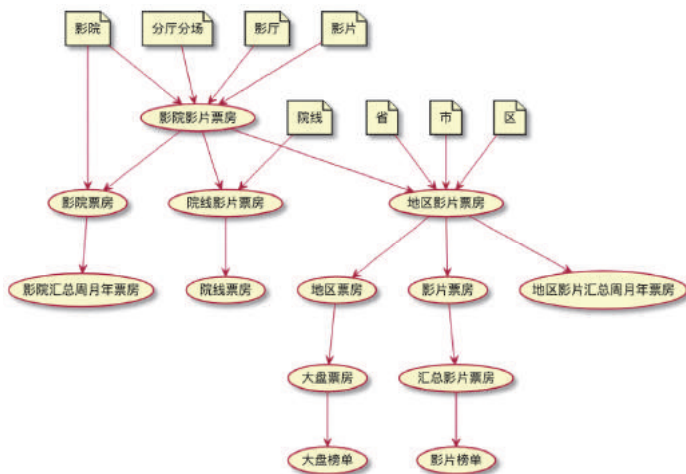
灯塔是阿里大文娱旗下一站式宣发平台，同时也是阿里巴巴为数不多对外提供数据的数据平台。作为数据平台，数据的时效性和准确性一直技术需要突破的重点和难点。

### 一、技术挑战

灯塔数据系统（前身淘票票专业版）从 2017 年开始建设，最开始采用 MYSQL 作为数据存储，基础数据定时计算写入数据库，经过 2 年多的建设，产品已经基本成形，但对于数据的实效性有了更高的要求，由于影院单日售票在 3000W 张，预售将近 1 亿张票，计算量大，写入频次高，从感知影院售票到客户端呈现数据，采用什么样的方案，什么样的技术，能够通过最小的改动让数据最快的呈现出来，成了技术考虑的难点。

### 二、技术策略

首先是缩小数据量，找出数据规律，实现数据的实时计算，各维度数据汇总如图：

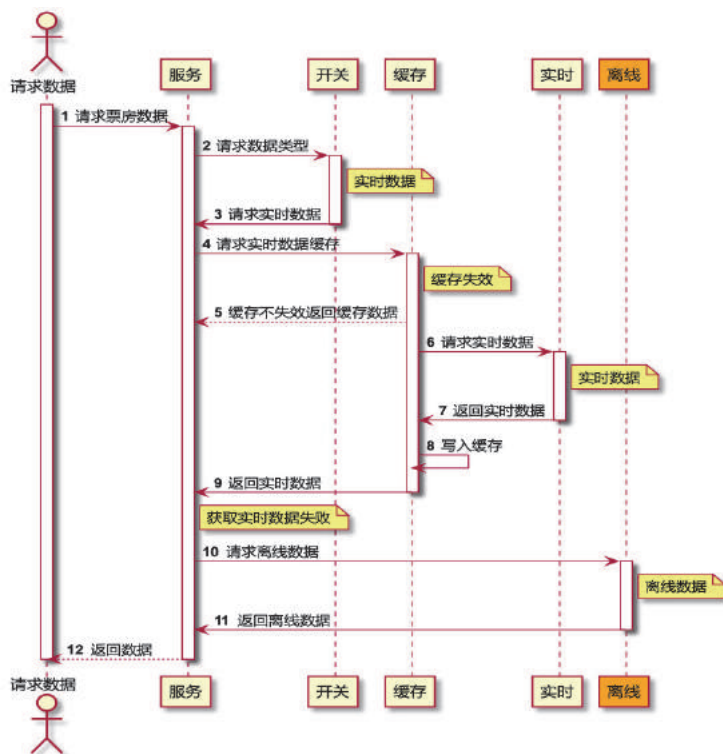


（图片：灯塔专业版数据汇总关系图）

各维度数据在业务应用的场景中，均可以按照时间、地区、业务主键进行检索，根据这个特征，我们生成了天然的 Key 组合，时间、地区、业务主键，并排列组合出三种 Key：时间\_地区\_业务主键,时间\_业务主键\_地区，地区\_业务主键\_时间。按照以上三种 Key 组合，在已知任何两个条件的情况下，均能实现对业务数据的检索。此时我们已经锁定了数据的存储平台 HBase。剩下的就是如何改造系统实现实时化。

### 三、落地方法

数据源有了，MYSQL 和 HBase，HBase 是实时数据，MYSQL 是离线数据，为了让上层业务无感知，特在底层数据做处理，实现离线数据和实时数据的结合，数据处理流程如图：



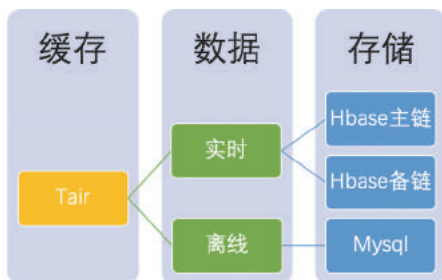
(图片：灯塔专业版数据处理流程图)

用户在请求票房数据的时候，先根据业务开关，决定请求实时数据还是离线数据，离线数据直接请求 MYSQL。实时数据，优先查询缓存，若缓存存在且不过期，直接返回缓存数据。



缓存数据失效的情况下，查询 HBase，重新写入缓存。

系统日常还是有上千的 QPS，为了防止缓存击穿，对数据源造成压力，需要对热数据进行缓存预热。由于数据的特殊性，T 日为最热数据，占到总流量的 80% 以上，这时候，缓存预热成了承受高并发访问的关键。定时任务每秒将 T 日数据整体刷入缓存，防止缓存失效击穿（因为都是 key-value 存储，后续考虑热数据直接写入缓存，直接替代预热方案）。其实为了防止击穿，这部分数据是 24 小时不过期的，数据的更新是依赖定时任务的，一旦数据链路故障、HBase 故障或算法异常，只需要停止定时任务，就能暂时止血，给技术留出处理时间不至于故障升级。同时 HBase 做了主备链路，而且主链路和备用链路的算法略有不同，保证主备链路不会同时出问题。这样的架构，对于应用而言，就有了 4 套不同的数据源做保证的。架构上线至今，数据未曾出现一次问题。而且无形中解决了高 QPS 的问题，数据的提供主要依靠 TAIR，而缓存应对 QPS 就显得简单的多了。



（图片：灯塔专业版数据源关系图）

系统的难点在于实时数据和离线数据的结合。数据结合共分为以下几类：

1. T 日查询，非实时即离线，如查询今日大盘票房；系统首先定义了一个方法，根据日期判定数据应该查询实时还是查询离线，由于行业数据是按照 6 点到 6 点，即 T 日数据，在第二日 6 点后才变为离线数据，且由于专资办数据回刷的问题，防止数据回跳，会在数据回刷后才切换为离线数据。当查询单日数据时，针对查询日期，判定数据源，进行数据转换；

2. 日期范围查询，即有实时又有离线，如查询影片每日票房情况；当查询日期范围时，由于日期范围时连续的，特将范围日期拆散成每一天，按照方法 1 中的判定规则，切分日期范围为离线日期和实时日期，然后数据源根据日期范围取最小日期和最大日期进行范围查询，查询后进行数据组合后返回数据；

3. 范围统计，离线+实时，如本周票房；当进行范围统计查询时，首先去离线数据，然后

根据日期判定，如果 T-1 数据还未回刷，则去 T-1 和 T 日数据，否则只去 T 日数据即可，将实时数据和离线数据进行加和，返回查询数据结果；

4. 榜单查询，离线补实时，如影片榜单。榜单查询由于榜单数据范围不好确定，范围查询有可能查询数据太多，所以在查询排行榜时，先取离线数据 2 倍的数据量，然后根据离线数据返回业务主键，查询当前的实时数据，将实时数据覆盖到离线数据后，进行内存排序和截取，最终返回榜单数据。榜单数据略有不同，比如院线影片，由于全国院线一共 49 家，此时不做离线查询，直接查询所有实时数据进行排序。

针对以上数据整合的各种可能，参照以往出现的各种问题，封装代码如下：

```

1 /**
2  * 查询地区影片单日票房示例
3  */
4 List<GwAdsAreaShowBoxOfficeDO> list = ResultUtil.listHBaseTemplate(
5     //查询操作的类名和方法名对象
6     TemplateBase.CallId.create(this, "queryByCityAndDate"),
7     //查询常用操作开关控制
8     HBaseConfig.hBase_show_box_office_config,
9     //查询数据日期
10    showDate,
11    //昨日回刷标识缓存标记值
12    cacheCoreService.getCacheObject(ResultUtil.getYesterdayReturnKey(), Boolean.FALSE),
13    /**
14     * 实时查询方法
15     */
16    () -> {
17
18        List<GwAdsAreaShowBoxOfficeDO> resultList = newShowBoxOfficeHbaseHelper
19            .listShowByDateAndCity(showDate, finalAreaId);
20        //内存排序截取
21        return sortAndLimit(resultList, finalOrderBy, finalLimit);
22    },
23    /**
24     * 离线查询方法
25     */
26    () -> {
27        GwAdsTfproAreaShowBoxOfficeQuery query = new GwAdsTfproAreaShowBoxOfficeQuery();
28        query.setShowDate(showDate);
29        query.setCityId(finalAreaId);
30        query.setOrderBy(finalOrderBy);
31        query.setPager(new Pager(1, finalLimit));
32        return gwAdsAreaShowBoxOfficeDAO.query(query);
33    }
34 );
35 return list;

```

（图片：示例代码 1）

```

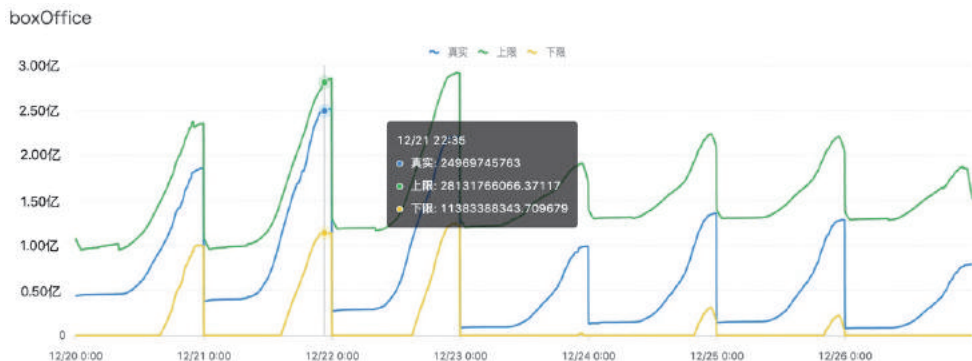
1 public static <T> List<T> listHBaseTemplate(TemplateBase.CallId callId,
2     BoxOfficeConfig config,
3     Integer showDate,
4     Boolean yesterdayIsReturn,
5     Supplier<List<T>> hbase,
6     Supplier<List<T>> mysql) {
7     //判定数据查询日期是否要走实时
8     if (isRtOn(config, showDate, yesterdayIsReturn)) {
9         try {
10            //查询实时数据
11            return hbase.get();
12        } catch (Exception e) {
13            HBASE_LOG.error("{} - listHBaseTemplate showDate:{} yesterdayIsReturn:{} HBase
14                callId.getId(), showDate, yesterdayIsReturn, e);
15            //实时数据异常检索离线数据
16            return mysql.get();
17        }
18    } else {
19        //查询离线数据
20        return mysql.get();
21    }
22 }

```

(图片：示例代码 2)

通过以上处理，在上层业务无感知的情况下，下游数据实现了整体实时化的切换。而且通过 switch 开关控制针对数据源能够实现单业务主备切流，实时离线数据转换，使得数据的稳定性更可靠，为底层数据改造和升级留下了充足的扩展空间。

有了架构还不够，还需要感知能力，业务异常感知还是比较简单的，但是对于灯塔来说，有数不代表正常，数据到底对不对，这是问题感知的关键，这时候需要一个智能化的监控系统。针对票房数据，在不改造代码的情况下，我们设计了一个切面，引入脚本代码，针对特定数据来源做数据动态处理，将返回数据整合在一起，并提取出来，通过算法识别票房数据行为趋势，如图：



(图片：灯塔专业版票房监测趋势图)

针对数据的趋势做智能化监控，当数据异常变化或者超过业务限定范围，就会通知告警，以此来有效的规避数据异常的情况，并能及时感知问题。

## 四、总结沉淀

随着 B 端业务的发展，数据的作用越来越大，在海量数据存储和更新的需要下，关系型数据库已经越来越无力，各种类型的数据存储起到了不同的作用，多数据源的整合也越来越重要。本文介绍了灯塔为了做到实时的数据系统，是如何组合 Mysql、Hbase、Tair 三个数据源来实现高写入，高并发、高可靠的数据系统，希望能给后续更多的业务系统提供参考和指导。

## 基于 webpack 的应用治理

作者| 阿里影业高级开发工程师 百命

当前市面上大部分前端应用都是基于 webpack 进行构建，而随着应用日益庞大，webpack 应用就会出现构建速度慢，构建结果体积大等一系列问题。

### 一、webpack 应用治理应该从哪个方向入手？

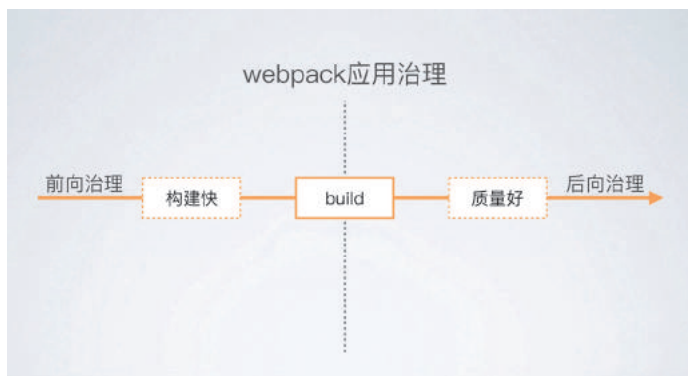
随着应用的不断迭代，webpack 应用最常见的两个问题就是：

1. 构建速度慢；
2. 构建体积大。

有一个很简单的划分方式，就是以构建（build）为分界线，分成前向治理和后向治理：

- 前向治理：提升构建速度；
- 后向治理：保证构建结果质量。

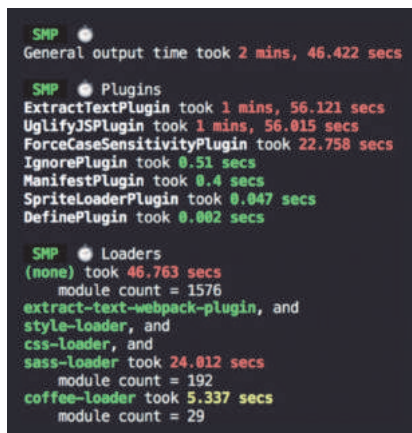
我们的治理方向，就是围绕前向治理和后向治理。



## 二、前向治理包含哪些内容？

前向治理的核心概念，就是一个字‘快’，目的就是提升构建速度，市面上大部分 webpack 优化文章都是这一类提升构建速度的文章，所以这里就简单介绍一些不错的实践

### 1. 利用 SMP 采集 webpack 数据指标



```
SMP ●
General output time took 2 mins, 46.422 secs

SMP ● Plugins
ExtractTextPlugin took 1 mins, 56.121 secs
UglifyJSPlugin took 1 mins, 56.015 secs
ForceCaseSensitivityPlugin took 22.758 secs
IgnorePlugin took 0.51 secs
ManifestPlugin took 0.4 secs
SpriteLoaderPlugin took 0.047 secs
DefinePlugin took 0.002 secs

SMP ● Loaders
(none) took 46.763 secs
  module count = 1576
extract-text-webpack-plugin, and
style-loader, and
css-loader, and
sass-loader took 24.012 secs
  module count = 192
coffee-loader took 5.337 secs
  module count = 29
```

数据先行，通过 speed-measure-webpack-plugin 采集性能指标，可以得到 webpack 在整个编译过程中在 loader、plugin 上花费的时间，基于该数据可以专项的进行优化和治理。

### 2. 开启缓存

如果通过 SMP 分析得知在 loader 编译过程耗时较多，那么可以在核心 loader，例如 babel-loader 中添加缓存。

```
{
  loader: 'babel-loader',
  options: {
    cacheDirectory: true
  }
}
```

### 3. 开启 happyPack 多线程编译

如果通过 SMP 分析得知在 loader 编译过程耗时较多，还可以通过使用 happyPack，开启多线程编译，提升开发效率。

#### 4. 使用 dll 技术

dll 可以简单理解成提前打包，例如 lodash、echarts 等大型 npm 包，可以通过 dll 将其提前打包好，这样在业务开发过程中就不用再重复去打包了，可以大幅缩减打包时间。

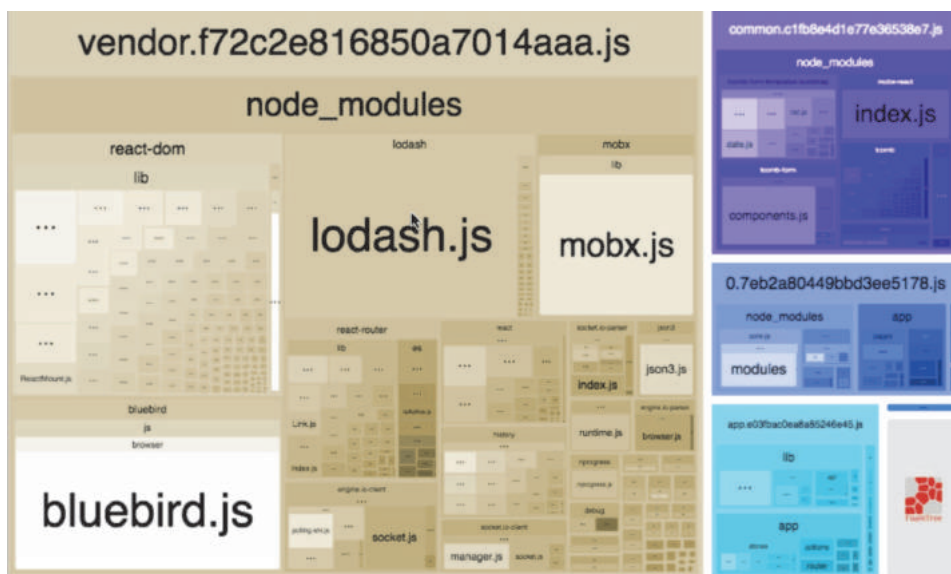
#### 5. 升级到 webpack5

webpack5 利用‘持久缓存’来提高构建性能，或许升级 webpack 后，前述的各种优化，都将成为历史。

### 三、后向治理包含哪些内容？

后向治理主要保证构建结果的质量

#### 1. 可视化分析构建结果



很常见的就是 webpack-bundle-analyzer，提供打包结果的可视化展示，如上图给予的决策帮助是：

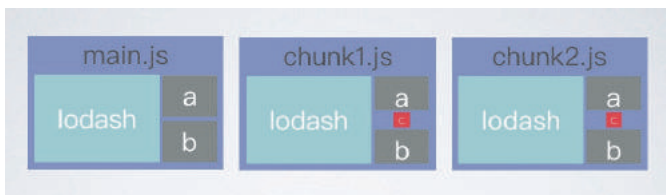
- 是否需要按需加载；
- 是否需要提取公共代码；
- 是否需要制定 cacheGroup 的策略。

## 2. 清理 deadcode

业务开发过程中，随着业务迭代，经常有些文件、模块及代码被废弃，这些废弃代码随着时间推移，将逐渐变为历史包袱，所以针对构建后结果，我们要做的就是清理其中的 deadcode。

前面 webpack-bundle-analyzer 虽然是最常用的插件，但依旧有一些缺陷：

1) 体积超小的 deadcode 模块引用，无法被准确识别。



例如上图：

lodash 体积大一下子就能被发现，就会意识到重复引用或者是未使用

但 deadcode 模块 c 体积很小，即便被 chunk1、chunk2 都引用了，也不一定能立刻发现，很容易被带到线上

而且这种 deadcode 也无法通过 splitchunk 来进行优化，因为 splitchunk 根据引用次数提取公共代码，无法分辨是否是废弃代码，所以对模块 c.js 这种的 deadcode 就无力了

2) tree-sharking 只保留有用的代码，但 deadcode 还在那里。

tree-sharking 大家都了解，摇掉不需要的代码，做为最终的输出结果，但反过来说，这些废弃代码依旧在本地真实不虚的存在着。

所以如何能准确的清理掉 deadcode 呢？这就需要通过 webpack 的 `统计信息(stats)` 来进行更细节的分析

## 3. 统计信息(stats)

stats 是通过 webpack 编译源文件时，生成的包含有关于模块的统计数据的 JSON 文件，这些统计数据不仅可以帮助开发者来分析应用的依赖图表，还可以优化编译的速度。

```
webpack --profile --json > compilation-stats.json
```

通过上述全局命令即可输出统计信息,例如：



```

{
  "version": "1.4.13", // Version of webpack used for the compilation
  "hash": "11593e3b3ac85436984a", // Compilation specific hash
  "time": 2469, // Compilation time in milliseconds
  "filteredModules": 0, // A count of excluded modules when `exclude` is passed to the
  `toJson` method
  "assetsByChunkName": {
    // Chunk name to emitted asset(s) mapping
    "main": "web.js?h=11593e3b3ac85436984a",
    "named-chunk": "named-chunk.web.js",
    "other-chunk": [
      "other-chunk.js",
      "other-chunk.css"
    ]
  },
  "assets": [
    // A list of asset objects
  ],
  "chunks": [
    // A list of chunk objects
  ],
  "modules": [
    // A list of module objects
  ],
  "errors": [
    // A list of error strings
  ],
  "warnings": [
    // A list of warning strings
  ]
}

```

其中：modules：表示 module 的集合

- module：webpack 依赖树中的真实模块；
- chunks：表示 chunk 的集合；
- chunk：包含 entry 入口、异步加载模块、代码分割（code splitting）后的代码块。

通过对 modules 和 chunks 加以分析，就可以得到 webpack 完整的依赖关系，从而梳理出废弃文件及废弃代码，同时也可以根据业务形态进行定制。

#### 4 webpack-deadcode-plugin

前面提到分析 stats.json，但因为是原始数据，数据量比较大，有一定处理和清洗成本，所以可以使用开源的 webpack-deadcode-plugin 这个插件

```
----- Unused Files -----  
  
/Users/mquy/Projects/webpack-deadcode-plugin/samples/start.js  
/Users/mquy/Projects/webpack-deadcode-plugin/samples/webpack.config.js  
There are 2 unused files (~?~)~.  
Please be careful if you want to remove them.  
  
----- Unused Exports -----  
  
/Users/mquy/Projects/webpack-deadcode-plugin/samples/unused.js  
  → abcdefgh  
  
/Users/mquy/Projects/webpack-deadcode-plugin/samples/app.css  
  → abcdef  
  
There are 2 unused exports (~?~)~.
```

通过 webpack-deadcode-plugin，可以快速筛选出：

- 1) 未使用的文件；
- 2) 未使用的已暴露变量。

#### 5. 结合 eslint、tslint 进行治理

lint 可以快速的扫描出未使用的变量，这能够极大的提升我们的 deadcode 清理效率。

- 1) 首先通过 lint 对未使用变量进行清理；
- 2) 再通过 webpack-deadcode-plugin 再扫描出未使用文件和未使用的导出变量。

顿时整个应用干干净净，舒舒服服！

## 四、参考

speed-measure-webpack-plugin:

<https://github.com/stephencookdev/speed-measure-webpack-plugin>

happyPack:

<https://github.com/amireh/happypack>

webpack-bundle-analyzer:

<https://github.com/webpack-contrib/webpack-bundle-analyzer>

stats:

<https://webpack.js.org/api/stats/>

webpack-deadcode-plugin:

<https://github.com/MQuy/webpack-deadcode-plugin>

## 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