

通往创新之巅



互联网技术架构 创 新 案 例 和 实 践



■ 卷首语

让技术创新如繁花般盛开



2017 年，微服务架构完成了对传统单体和 SOA 架构的革新，敏捷开发和快速交付不再宿本无根；2017 年，AI 方兴未艾，以深度学习、神经网络为基石的预测平台也在悄然冒头……

这一年，架构、运维和前端等技术领域的变化，也对互联网企业提出了一项最重要的考验，即：如何向「主流」发起挑战。当 Spring Cloud、Dubbo 占据微服务框架的主导，K8S 成为容器界的新兴骄子，Python、Erlang 发起在编程语言领域的直追时，这些所谓的主流技术真的适合各家不同的软件架构吗？答案是未必。

主流并不意味着最佳。真正的高级玩家们，多是走在改造的前沿，评估企业原有的软件系统及结构，因地制宜地规划微服务、大数据、SPA 等体系，设计及搭建更合适自身的系统、框架或平台，以实现更好的用户体验。

围绕着用户体验的提升，根植于视频行业的 FreeWheel 也亲历了新一代互联网技术的成长。在其成立的十年间，新技术的扩增与迭代、互联网视频的全面爆发与极速进击，使 FreeWheel 在架构演进层、技术栈选型及更新层持续不断地寻求最佳解决方案，与时俱进、「生息繁衍」。

十年，是一种历史的积淀、创新的变现，同样也是对未来的覩见。在 FreeWheel 林林总总的各类技术创新背后，我们见证了其摆脱 Hadoop 而完成的系统进化、容器化和服务化管理平台的构建，微服务框架从 0 到 1，UI 系统前后端分离，数据处理架构和方法创新，以及用 AI 和区块链技术加码广告全管理系统升级……当然，花开繁盛，生息不止。用 FreeWheel 总架构师 Jack 的话来说，一致性和可预测性是其架构设计目前最关注的两点，好的架构是能够持续积极地面对不确定性和响应变化。

在这本 FreeWheel 与 InfoQ 联合推出的年终特刊中，我们试图以 7 篇站在技术前沿视角倾力奉献的文章，来记录当下的一些最佳技术实践，抛砖引玉，与更多企业一起探讨究竟该如何实现技术创新或转型。

开启这本书的阅读之旅，您将亲历 FreeWheel 十年整体架构演进、核心数据库从单点到跨 DC PXC 集群演进、PB 级海量数据的处理查询之道、前端组件框架 SparkUI 的诞生与成长、支持超级赛事直播广告的架构创新、亿级视频广告事件预测系统构建思路以及高质效技术团队的管理良方。**在本书中，您也同样可以看到 FreeWheel 是如何刷新「主流」，发起对自我的挑战的。**

赠人玫瑰，手有余香。希望这些如花般的技术实践，能让您看到创新的春天，走上新的技术旅途。

目录 | Contents

- 5 写在第十年：聊聊 FreeWheel 的架构演进历程**
- 14 从单点到跨 DC PXC 集群：核心数据库演进详谈**
- 27 PB 级海量数据的处理查询之道：大数据平台最佳实践**
- 44 亿级视频广告事件预测系统构建之道**
- 62 SparkUI：前端单页应用演进与实践**
- 83 FreeWheel 如何支持超级赛事的直播广告**
- 91 容力：如何打造更高质效的技术团队**

写在第十年： 聊聊 FreeWheel 的架构演进历程

第一个产品——MRM 的上线

FreeWheel 成立于 2007 年初，致力于解决电视行业在互联网领域的货币化问题。当时互联网视频刚刚兴起，正处于短平快的高速发展阶段。在国际上，Google 刚刚收购了 YouTube；而国内几家 P2P 流媒体公司则打得如火如荼，优酷也刚刚开始运营。热火朝天的表象背后，行业缺乏清晰的策略和商业模式，版权诉讼此起彼伏。

FreeWheel 的目标是提供一个方案，让整个视频行业的产业链——包括从内容制作、版权销售到传播渠道的所有参与者，都能在一个平台上，公平灵活地管理各个参与者的权限和合作分成等业务需求，从而维护视频生态系统，促使行业健康成长。因此，FreeWheel 的第一个产品，也是目前的主营产品，被命名为 MRM（Monetization Rights Management）——货币化权限管理系统。该产品极具行业前瞻性，虽然很多人都能预见到电视行业互联网化的趋势，但是没人知道这个确切的节点何时会到来。当时，

在美国包括电视台在内的诸多视频内容公司，数字部门的收入尚不及传统部门的零头，视频生态系统应该包含哪些业务模式，各方应该如何合作等细节亦尚未清晰。

电视行业主要包括两种业务模式，即广告和订阅。FreeWheel 的创始人均具有美国 DoubleClick 公司的工作背景，加之互联网广告的热潮，选择广告模式可谓水到渠成（后来 FreeWheel 也被形容为视频领域的 DoubleClick，这是后话）。MRM 的本质就是广告服务以及权限 / 分成管理。权限 / 分成管理是一个很好的切入点，因为版权诉讼的根本冲突来自于钱，解决了钱的问题，版权就不再是问题。然而这个切入点的背后，首先需要建立一个广告服务系统。于是，2007 年 7 月 FreeWheel 北京办公室成立，Diane Yu 女士组建工程师团队，开始为该系统的搭建编写代码。广告服务系统通常包括几个基本模块：管理业务的 UI 系统，执行广告投放、权限管理、分成等的广告服务器，后端的报表系统，以及前端视频播放器的集成类库；后来还增加了一个预测系统来帮助销售和监控广告投放，毕竟广告投放的数量与利润直接相关。最初的团队构建也采用了类似的组织结构，这也是最初的架构。Conway（康威）定律提到，组织架构定义了你的系统架构，亦是相似的道理。

重大的决定：推翻之前的广告服务器设计

与此同时，FreeWheel 已经签下了几家试水的小型客户，潜在的大客户依然处于观望状态。广告服务系统第一个版本的发布日期定在了 2008 年 1 月。虽然具体的产品需求还在变化，有时甚至客户连自己的具体需求都尚未明确，但是同所有创业公司一样，最终决定先确定发布日期并初步整合出一套系统再谈其他细节，毕竟公司存活才是第一位的。FreeWheel 的前期或多或少是处于此种境况，从架构设计的角度来说，即求快求简。因此我们做的第一个比较大的决定，就是推翻了之前的广告服务器设计，

毕竟“留给中国队的时间不多了”。

原先的设计有以下几个关键点：

- 使用 Apache Module,
- 多个进程（类似于 MicroServices），
- 并且 C 语言和 Java 混用。

这种设计主要会带来两个问题：高复杂性和测试 / 维护的难度大。而新的设计则采用 C++，且只有一个进程，用 FastCGI 和 Web 服务器进行通信（后来很快甚至放弃使用 FastCGI，直接内嵌一个 http 服务器）。基于新的设计，总架构师 Jack Deng 和同事共同连续奋战两个星期，加上另外两个同事写的测试用例，第一个可以 Demo 的版本成功问世，加班工作也就此告一段落。不仅如此，美国的运营同事也已蓄势待发，公司购进了几台 AMD 的服务器，开始准备上线。总体来说，运营部门对调整后的设计是比较满意的，部署操作简单，只需将可执行文件和配置文件拷贝即可运行。值得一提的是，Jack 还做出了另外一项重要决定，虽然当时使用内存成本较高，但公司直接选择了使用 64 位，并且把数据都放到了内存里来简化系统。得益于之后内存颗粒的降价，该点设计原则到现在大体没有改变——能放进内存的就不会放在其他地方。在这段时间里，基于 Ruby on Rails 的 UI 系统、Hadoop 的报表系统，以及一套 Flash ActionScript 2 的类库做完了第一个版本，2008 年实现了准时上线。

架构的从 0 到 1 与从 1 到 10：演进与创新

但上线了几个客户以后，Hadoop 系统开始出现性能问题。最先发现的是 Log 的 Parser 竟然用了 Reflection，内存也出现了空间不足的征兆。调整了一段时间以后，Diane 找到了 Hadoop 方面的专家来帮忙调优。专家开门见山就问：“你们的 Cluster 有几百台？”我们看了看机房里三台廉价服务器，对话就没能有效地继续下去。当时报表系统的业务需求相对

简单，用 Hadoop 有点大材小用，何况当时 Hadoop 的版本是 0.15，稳定性差。于是仅一个周末的时间，我们用 Python 做了个原型直接将结果统计出来，大概有 100 行代码。我们的 Log 格式起初是文本的，而业务比较复杂，结构化的格式会更加适用。于是我们扩充了重构的范围，让这个改变更大一些，用 Protocol Buffers 记 Log，同时用 C++/Python 写数据统计。这个 C++/Python 的程序基本上类似于简化的 Hadoop，在固定的节点上做预处理和预统计（类似 Map），然后在一个中央节点汇总到最终的结果（类似 Reduce）。这个过程比之前提到的广告服务器设计要困难一些，因为需要换掉一个在线系统。写代码的工作相对简单，在两三个星期之内完成几千行代码。而这个很简单的架构撑到了现在，直到我们换成基于 Hadoop 和 Presto 的新数据平台。这个简单架构的优点和缺点都很明显——简单可操控，但也带来了很多功能和灵活性方面的限制。

在我们忙于报表系统的同时，预测系统也上线了。

第一个版本的预测系统基于 Erlang，毕竟 Mnesia、高并发和 Process Supervisor 模型看起来很漂亮。虽然函数式编程有些不一样，但是代码写得也很快。而上线以后发现 Mnesia 性能存在问题，更重要的是 Erlang 标准库 OTP 某些代码的质量实在令人担忧。系统崩溃几次之后，同时系统变得复杂使 Erlang 代码读起来有些困难了，同时我们也希望采用一个采样模拟的方案来提高准确度，需要重用广告服务器的代码来测试模拟的请求，最终我们决定用 C++ 重写一个版本。这个重写的版本参照了 Erlang 版本的结构，比较复杂，维护起来依然有些困难。于是我们在 2011 年又推倒重来了一次，抛开历史负担，写出了一个更简单的版本，整个重写过程用去了一个多月时间。这个版本的架构一直维持到今天。总体来说后端几个系统的大重构还算顺利，原因主要在于重构的比较早，较好地控制了代码数量，人员相对稳定，以实现速战速决。

与此同时，客户端的类库也经历了一些波折。首先我们尝试用 Haxe

来同时支持 Flash 的 AS2 和 AS3。紧接着随着 iPhone 的崛起，又做了 iOS 的类库，接着是 Android 的类库。同时有客户要用 Silverlight 的集成，我们又做了 Silverlight 的库。随着技术的发展，Flash 开始走下坡路，用户普遍切换到 HTML5，从而 JavaScript 的类库应运而生。不仅如此，业界开始制定标准，从 VAST 的各个版本到 VMAP 的各个版本，还有 VPAID 的支持。加之各种设备的出现和普及，比如 Roku 和 Sling Box，每个细分领域都想要集成。如果 Android 的生态系统是碎片化的，那么这个生态系统可以说是颗粒化了。支持这么多的平台基本是不可能的，投入产出比就更低了。可想而知，如果战线铺得太宽，代码质量下降得也比较明显。

综上，我们采取了以下策略：第一，标准化。鼓励客户使用标准协议；第二，服务端化。把一些功能转移到服务器端来实现，简化客户端的逻辑，避免在每个平台都要实现一套类似的逻辑，举例来说工程师设计了一个在客户端做 HLS Proxy 的方案，来支持向 HLS 视频流里插播广告的功能，我们把这个逻辑搬到了服务器端；第三，尽量尽快终止不常用的集成，例如 AS2 和 Silverlight；最后是更加激进地简化系统，比如我们最初设计了一套复杂的 Android 自动升级方案，客户端先导入一个 Loader，然后由 Loader 导入最新的库版本，结果很多 Bug 都出现在这个 Loader 上，网络延迟故障、安全和各种时序等导致了很多不必要的问题，因此我们决定把这个功能取消，让用户直接链接库。

与此同时，基于 Ruby on Rails 的 UI 系统也在快速发展。代码数量甚至超过了其他几个系统的总和，而且 Rails 版本一直都未升级。2010 年左右我们遇到了第一个挑战：需要升级到当时比较新的 Rails 3 版本。除了大量的代码改动，还包括库的变动和升级。这项工作耗时数月，但总体上对代码质量的改善却十分有限。之后我们甚至举行了一个简化代码的比赛，哪个团队每个版本删除的代码行数最多，就能得到额外的奖励。结果这个活动也无疾而终。随着大型代码库增长而来的，是产品质量的下降

和团队成员的不稳定。在 2012 年前后，尝试模块化整个 UI 系统，形成几个小系统，并且把公用部分抽取出来做 UI Foundation，进展都不太顺利。结果是原来一个有问题的系统，反而变成了多个有问题的系统。从中亦获得了不少经验教训：最重要的问题是没有尽早对代码的质量进行监控和管理；其次是业务变化快、人员不稳定；Rails 框架本身存在问题，以及制定计划的时间线铺得过长等等。当然这几个因素也是互相影响的。

总之，FreeWheel 初期发展面临的挑战是各个创业公司所共有的，即各种不确定性，包括快速变化的业界图景、业务需求、技术方案以及团队组织。在此背景下，技术架构应相应迅速地进行调整。我们积累的主要经验在于，满足需求的情况下尽量简化设计去实现，并且尽快响应变化，同时为了避免技术债务的累积，要控制好代码总量。有一段时间我们只强调一个评判标准并力求使其深入人心，即代码行数——一方面要实现功能需求，另一方面要压缩代码行数。结果也有些小插曲，比如某一次一个工程师为了避免返工，把 Python 代码全堆到了一行上。

架构再升级：向一致性与可预测性的进击

这套策略当然也是有局限性的。业务需求有其内在的复杂性，技术上能做的简化同样有限，而且随着多年的技术债务积累，（重要）客户越来越多，团队也越来越壮大，快速简单的原则有时就不太可行了。一致性和可预测性则愈发重要。因为同时变动的模块有很多且经常互相依赖，如果不能按时按质的完成，就整合不到一起。因此，架构的策略也逐渐从“求快求简”向“求稳求准”过渡。2013 年左右是 FreeWheel 的分水岭。最后的一个快速项目是用 LevelDB 和 Memcached 做了一个 KV store，来存储用户投放数据，后来因为性能问题存储换成了 RocksDB，最后又换到了带有商业支持的 Aerospike，因为集群管理、复制和容错之类的工作没有太简单的方案。

架构策略的改变还有一个原因，就是技术基础设施的进一步发展而走向成熟。数年内我们见证了不少技术上的起伏，比如 Flash 的消亡，Chrome 的崛起，移动端更是喊出了 Mobile First 的口号，Rails 已经不再热门，业内将展示都迁移到客户端，而 JavaScript 就更不必说了，jQuery 也已经成为过去时，AngularJS 经历了一个起伏的周期，ReactJS 已经成为了半个缺省配置，GCC 的版本已经升级到 7，新的项目都至少用上了 C++ 11。在大数据方面，Hadoop 2 也推出一段时间了。在 Cloud 和集群管理方面，AWS/Docker 已经非常流行，而 Kubernetes 也抢占了很大的份额。唯一的例外可能是 Python 2，基本上还活着。要适应新的架构策略，我们也需要升级系统的基础设施。

FreeWheel 在 2014 年开始讨论和准备基础技术架构的升级，之后就开始逐步实施。相应的方向选择其实相当主流化，无非就是搜索引擎上最为热门的几个：

- 容器化和服务化管理平台。前者是 Docker，后者在 Kubernetes 出现以后就很自然了。同时我们还开始尝试把系统迁移到 AWS 上。
- UI 系统的前后端分离，前端往浏览器端展示迁移，后端则 API 服务化。前者选用了 ReactJS，关于后者曾经有过讨论。我们从 2012 年开始尝试 Golang。Golang 从语言、库、社区、支持工具各个方面都不错，但是有工程师提议想尝试 Elixir。考虑到一致性以及之前对 Erlang 的体验，这个基本没有争论，很快就决定采用 Go，并且同样是出于一致性的考虑，决定使用 gRPC 和 Protocol 来标准化。
- 数据处理，Hadoop 显然是缺省选择了。这里值得一提的有两点：第一是缺省用 Golang 而不是 Java，这点是考虑到 Golang 是公司层面的缺省选择，并且我们没有太多 Java 的经验；第二是尽量避免

使用 ETL 流程，而是存储最细粒度的数据，动态地通过SQL生成结果。这点考虑主要是基于之前对数据维护成本的经验。维护大量的中间数据（ETL 的结果）对于数据一致性的验证和检查，需求变动的响应，以及数据的导入，都有很高的要求。而随着计算成本的降低，在原始数据上查询是很自然的想法。我们用 Presto 搭建 Log 查询系统从而验证了这个思路。此系统应该是内部使用最多的系统。

- 广告服务器和预测系统。该方面的技术基础改变不大，主要是模块化，并且尽量使用更新的工具，比如编译器和静态检查工具。我们从 GCC 4.1 升级到 GCC 4.8，之后更是直接切换到了 Clang。同时新的代码都用上了 C++ 11。预测系统也有一些改进，包括对机器学习的利用，C++11 以及对基于 Hadoop 数据平台的尝试。同时，一些新的内部支撑服务，我们都开始缺省用 Golang 来实现。

架构升级还在进行中，大致到 2018 年年底完成。风险和质量显然是这次升级的最关键考虑因素。如前所述，一致性和可预测性是我们目前最关注的两点。对于前者，我们使用一致的技术栈（缺省使用 Golang），更多形式化的定义（数据库 Schema，Log 和 API 的 Protocol Buffer 定义等），更多的测试（包括单元测试和回归测试等），以及更强的 Code Review 来保证。而对于后者，主要是通过更多的量化，包括更多的指标评估和更多的监控来保证，比如延迟、吞吐量，数据大小等。更实时的数据，更多对系统的洞察，其实都是业界的趋势。当然，架构升级在进行中，也遇到了不少问题，比如需求的变动、计划的变化、依赖服务的不稳定性，还有人员的变化等等。我们所做的，也就是在设计的时候尽量简化，留有余量，在执行的时候开阔思路及时调整。

好的架构是持续积极的面对不确定性和响应变化

FreeWheel 曾经的创始人 Doug Knopper 在公司的大会上提到：“有同事问我，工作和家庭怎么平衡？我的观点很简单，这不是个问题，因为家庭永远是第一位的。我们做的只是广告，并不是火箭，看不到广告人们也不会怎么样。”我觉得好的架构演进背后也应该有类似的平常心：业务是第一位的，能支撑业务并可持续发展，就是好的架构。好的架构不是奇迹，也不能解决所有问题，它的本质是持续积极的面对不确定性和响应变化，有所取舍，并最终保证业务有序的发展。

从单点到跨 DC PXC 集群：核心数据库演进详谈

背景

OLTP 主数据库集群是 FreeWheel 各产品模块所需要的最主要数据来源和数据接口之一，改造前的数据库架构是比较传统的单点 MySQL Master 带几十个 Slave + 一个后备主库的模式，并且公司整体的架构设计很大程度上需要通过数据库集群作为各模块的数据接口，同时作为数据上下游以及跨数据中心的同步机制，目前的架构短时间内无法完成大的改动，因而需要数据库层面在保证性能和以上功能性需求的前提下，进行高可用性方面的升级改进，主要需求具体包括：

1. 尽量减少或者完全去除对维护窗口的依赖。
2. 当主节点单机硬件变更、故障时希望能够避免对业务的影响，增加集群的整体健壮性。
3. 随着业务推动集群的不断增长，急需提高能够挂接更多 Slave 的能力。面对以上的需求，DBA 团队本着“扎实求稳、小步快跑”的原则，

作为 Owner 与开发测试团队一起，用了大概一年的时间，最终把公司生产环境核心数据库从 MySQL5.1 非 GTID 集群，顺利迁移为基于 PXC+GTID 的集群。

本文主要对我们在整个过程中遇到的问题、解决方案和相关思考进行总结，希望对准备使用 PXC 或者已经使用 PXC 的同行们能具有一定的借鉴意义。

方案选型

业界类似方案调研

MySQL 集群高可用方案、方式目前有很多种，前期我们广泛考量并调研了以下方案，在技术上做了简单对比，最终选定 MHA 和 PXC 进行进一步可行性验证。

1. 存储层的 HA – DRBD/SAN：存储层块设备同步方案，数据更新频繁，需要专门（相对昂贵）的设备；DRBD 在测试阶段出现了性能问题，相对的高成本和性能问题使我们最终放弃。
2. MMM/MHA：目前比较流行的方案，尤其 MHA，相对轻量级，可控、可自行修改源代码，但缺乏商业支持，一旦 Binary Log 补齐出问题导致转接失败则会非常被动，尤其在 Slave 多的情况下稳定是第一需求。所以在准生产部署实施后选择了放弃。
3. Galera MySQL (PXC/MariaDB Galera Cluster)：有好的商业支持，在性能和 HA 上平衡得很好。其中 PXC 的外围工具相对较为成熟，成为了我们最终的选择。

4. MySQL Fabric：刚刚 GA 不久的产品，功能还在完善中，并且需要结合多主进行使用，不能直接提供想要的 HA 功能。有 Oracle 公司的支持，将来或许是个好的产品。

5. MySQL Cluster (NDB)：类似 RAC 的方案，对网络压力较大，成熟

度方面口碑欠佳，业界生产环境使用较少。而且不是乐观锁机制，笔者认为其更适用于少写多读。

锁定 PXC

PXC 的优势还是显而易见的：

1. Galera 的 WriteSet 协议的乐观锁机制是在性能和可用性上是一个很好的平衡点——比 MySQL 的同步复制性能更好，比异步复制更能保证数据一致性。当然前提条件是要按照最佳实践正确地使用 PXC，合理控制 Deadlock 的回滚带来的额外成本，或者从根本上慎重考虑是否启用多节点来编写。

2. 成熟靠谱的商业支持。在前期，针对公司集群的具体情况，DBA 团队和 Percona 的远程技术工程师一起做了细致完备的准备工作，Percona 工程师的技术支持亦高效合理，这一点在后来我们创建更多的技术问题 Ticket 时也得到进一步验证。

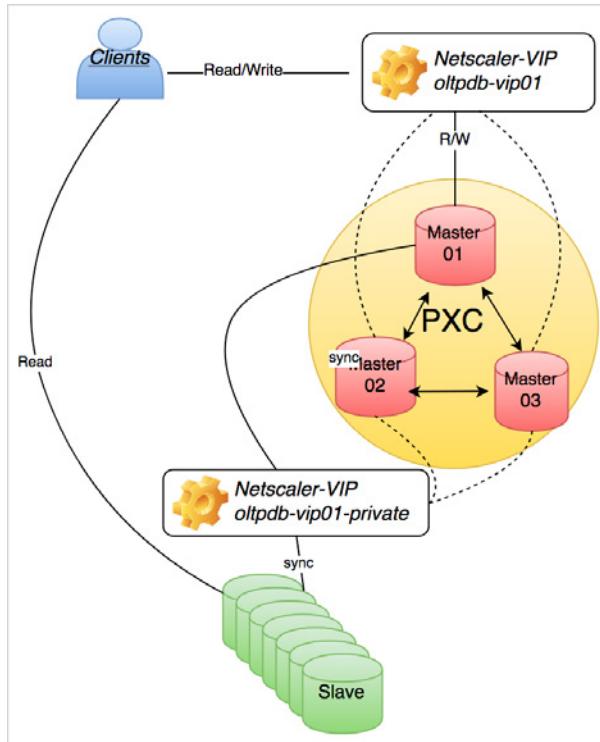
3. 因为三节点较好的一致性保证，相比 MHA 的补差量的方式，PXC 造成大面积 Slave 同步同时中断的可能性较低。由于 DBA 团队人员相对较少，一旦 Slave 大面积出现问题时，即使使用自动化脚本，人手也难以应付。因此，能够保证大多数 Slave 的正常工作是我们现阶段必须时刻考虑的因素。

最终方案

总体上来看，目前主 DC 的集群建设已经完成并运转正常；跨 DC 的方案也已经确定并在实施的准备过程中。最终我们要实现的是 PXC 集群 + 跨 DC 的 Master-Master 同步，可以最小限度地减少对维护窗口的依赖，并最大程度地提高集群可用性。

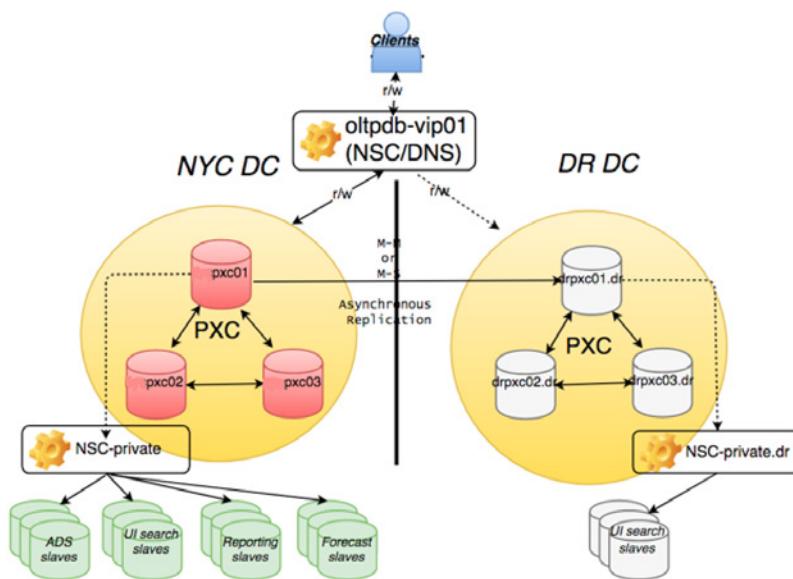
更多细节如下：

单 DC 方案：



细节方面请参照后面的讨论。

跨 DC 方案：



在讨论跨 DC 方案时，最终我们没有采用单 PXC 集群跨 DC 的方式，而是计划采用 Master-Master 模式，并构建两个 PXC 集群之间的双主复制关系，这样既可以更好的保证 Slave 的平滑切换，又可以防止跨 DC 网络抖动引起的性能问题，甚至是集群脑裂的发生。

PXC 从 POC 到落地需要解决的技术细节

5.5 to 5.6 兼容性和升级

Oracle 公司确实在 MySQL 兼容性方面做了很多工作，这里要提的一点是，切记纸上谈兵，要认真做兼容性回归测试之后才能做大面积升级，尤其是对于最重要的和运行次数最多的模块。案例 1，在测试一切顺利，大家都认为多此一举的时候，Point Release 中的一个重要模块的报错让我们警醒起来，经过仔细排查和分析，出现该问题的根本原因是 JDBC Driver 在一个 GROUP_CONCAT 的返回数据类型上有细微的差别，导致报出异常程序停止；案例 2，我们的环境中用到的比较老版本的 ODBC 同样存在 5.6 兼容性问题，好在这个问题会导致数据直接连接不上，得以在上线前发现并解决。

NIC Bonding还是双网卡分流

由于三节点之间的流量直接决定整个集群的性能，我们曾经考虑让 IST/SST/Flow Control/Internal Communication 的流量单独使用一块网卡，使其和数据访问隔离分流。虽然最终我们采用万兆网卡 Bonding 方案，这里仍要提醒大家的是：分流是一个可行方案；并需要注意不要因为网络原因触发 Flow Control。

是否启用GTID

最初考虑过不想一次做过多改动，先上 PXC 然后 Enable GTID。但经过仔细考虑，最终决定先上 GTID，然后基于 GTID 上 PXC，关键点是：只有启用了 GTID 才可能有 Slave 的自动无缝 Switch Master。GTID 可以

保证在不同 Master 之间，即使 Binary Log 的文件名和 Position 不同，但只要逻辑意义上的 Transaction 点一致就能通过 GTID 定位，如果没有 GTID，Slave 从一个 Master 切换到另外一个 Master 在技术上虽然仍可行，但需要解析 Binary Log，通过 Xid 定位反推不同 Master 的不同 Binary Log 和 Position，细节此处不过多讨论，可行但操作相对困难且易错。当在 PXC 启用 GTID 时，我们发现 GTID 中的 UUID: xxxx-xxx-xxxx 的 UUID 部分，并不是 PXC 节点中的任何一个节点的 UUID，而是 PXC 以 Bootstrap 方式启动时生成的 UUID，这一点也需要注意。查阅文档后也得到了验证。

性能VS单点Master

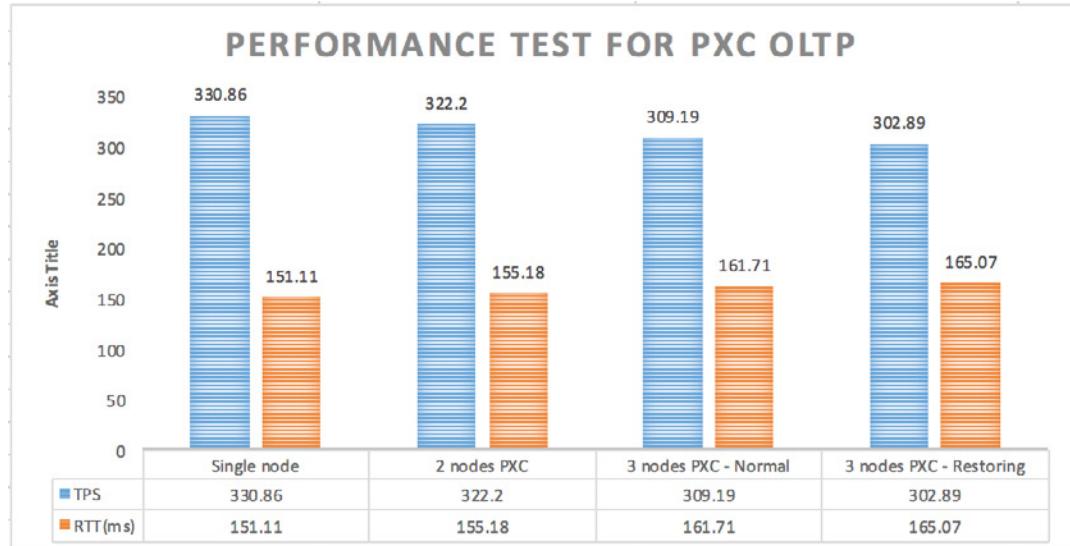
翻阅了官方测试资料，也参考了其它 DBA 人员的测试结果后，我们还是要结合 FreeWheel 公司的业务特点做自己的测试，在四种场景下使用 Sysbench 做的压测对比结果如下：

1. 单节点
2. 两节点
3. 三节点
4. 三节点 + 其中一个节点在做 SST 全量恢复

Test Scenario for OLTP		Single node - Single point	2 nodes PXC cluster - Temporary Status	3 nodes PXC cluster - Normal Status	3 nodes PXC cluster - Restoring Status
Description					
Baseline data	oltp-tables-count:	10	10	10	10
	oltp-table-size:	10,000,000	10,000,000	10,000,000	10,000,000
	max requests:	100,000,000	100,000,000	100,000,000	100,000,000
	total time:	3600.1158s	3600.1145s	3600.1535s	3600.2037s
	Number of threads:	50	50	50	50
Queries performed	read:	16,576,128	16,339,482	15,583,834	15,266,608
	write:	4,764,608	4,639,852	4,452,524	4,361,888
	other:	2,382,304	2,319,926	2,226,262	2,180,944
	total:	23,723,040	23,199,260	22,262,620	21,809,440
Transactions: (TPS)		11191152 (330.86 per sec.)	1159963 (322.20 per sec.)	1113131 (309.19 per sec.)	1090472 (302.89 per sec.)
read/write requests: (OPS)		21440736 (5955.57 per sec.)	20879334 (5799.63 per sec.)	20036358 (5565.42 per sec.)	19628496 (5422.02 per sec.)
Other operations:		2382304 (661.73 per sec.)	2319926 (644.40 per sec.)	2226262 (618.38 per sec.)	2180944 (605.78 per sec.)
total number of events:		1,191,152	1,159,963	1,115,131	1,080,477
total time taken by event execution:		180000.2899s	180000.7730s	180002.2801s	180003.0501s
Response time	min:	28.98ms	29.09ms	31.31ms	31.89ms
	avg:	151.11ms	155.18ms	161.71ms	165.07ms
	max:	374.51ms	374.60ms	514.29ms	716.89ms
approx. 95 percentile:		193.61ms	201.96ms	207.29ms	206.61ms
Threads fairness:		events (avg/stddev): 23823.040/0.78	23199.760/1.34	22262.670/1.35	21809.440/1.17
Threads fairness:		execution time (avg/stddev): 3600.0058/0.04	3600.0155/0.03	3600.0456/0.05	3600.0610/0.06

TPS (Transaction Per Sec) 越大性能越好。

RTT (Average Response Time) 越小性能越好。



可以看到，在我们单节点写入的各场景下，PXC 的性能损失并不大。

利用 NSC 实现 Public VIP 和 Private VIP 的设计

基础架构上 FreeWheel 有专业的 NetScaler 设备，也有网络专家支持，在综合各方面的因素之后，最终我们决定大胆启用双 VIP 的方式，三节点的 PXC01/02/03 前面挂一个 Public VIP：VIP01 给所有应用访问使用，01 宕机自动转接 02，以此类推到 03；同时再挂一个 Private VIP：VIP01-Private 给所有的 Slave 连接到 Master 时使用，VIP 转接逻辑同上。这里在考虑 VIP 探活机制上有过几次反复讨论和调整，使用过的方案有：

- i. TCP 3306 探活，最基础的探测；MySQL hang 或者网络特别繁忙、极端情况下容易误判。
- ii. MySQL 协议层面只读 “select 1;” 探活；粒度适中，但 FC 发生时有小概率可能转接到不可写节点。
- iii. MySQL 心跳表 Update 写成功探活；会产生不必要的 Binary Log 记录进而造成 GTID 的复杂化，这个方式主要用以应对小概率 FC 场景。

依次尝试下来，最终我们稳定在方案 ii 作为线上方案。目前一切平稳。

关于横向扩展的考虑

首先要说明，PXC 一般来讲是 HA 方案而并不是 Scale Out 方案，但实际上如果安排合理，也是可以考虑借力于 PXC 的。

1. 关于读

只要是 OLTP 的读，尤其是更新的热数据集，在 02/03 节点的读应该不影响 01 节点的性能，或者说不伤害整个集群的性能。换言之就是可以提高整个集群的吞吐，毕竟 02/03 节点只是 Standby 资源。

2. 关于写

不同于读，这里必须考虑不同节点更新数据时的锁冲突问题，一旦 WriteSet 协议下的二阶段 Authentication 失败，就会造成一阶段已执行事务回滚，得不偿失。铁律是必须保证不同节点的更新数据没有交集。

3. FreeWheel 特有的多 Slave 扩展的需求

由业务特征决定，我们的一集群目前有几十个 Slaves，个别 Slave 还在第三层复制关系中，三级复制，基本已经达到一个 MySQL 实例能带动 Slave 的极限。但是随着业务的增长和新功能的开发，对于 Slave 的需求还在不断增长，PXC 强一致性的三节点给了我们机会，可以顺理成章地在 02/03 节点上挂接 Slave，雪中送炭，极大缓解了 DBA 小组满足 Slave 需求的后备方案的压力。

受益于 PXC

1. 不再担心主节点硬件或者电源问题等的宕机事件：

在 Staging 环境，由于托管机房的误操作问题，发生过 01/02 节点的电源被误掉电的情况，令人欣喜的是我们的 DBA 并没有被 Call，第二天上班看监控邮件才发现 VIP 自动做了切换，一切如常。

2. 零宕机前提下硬件 /OS 层面的变更成为可能：

PXC 刚刚上线我们就遇到了内存上的压力，研究后发现同等情况下 PXC 确实会占用稍高些的内存。没有犹豫，直接进行滚动宕机硬件内存升级，整个集群对外服务零宕机，半个小时一切顺利完成。

3. 软件的滚动升级和部分 Schema 大表更改的滚动更成为可能：

将来，无宕机升级 5.7 可以采用滚动升级的方式进行；一些类似索引操作或者 Schema Default 值的更改。

4. 提供了可满足大量 Slave 需求的可能

比如 01 节点用于写操作，02/03 节点用于带 Slave，或者其他类似的变通方案，尤其是临时需要生产环境流量的测试 Slave 需求，可以毫不犹豫地挂接在 02 或者 03 上。

5. 多读多写的潜在能力：

当对实时性数据读取需求高的应用过多时，我们可以考虑启用 02/03 作为读节点。当写数据可以在业务上保证无冲突时，同样可以考虑启用 02/03 节点。

来自实践的教训

以下每段都是对实践中遇到的真实问题的总结，个别内容还有待更深入地探索。

自增列步长兼容性问题

准生产上线的当天，发现一个重要模块的某些功能随机报错，错误信息大致为“Primary Key not Found”，经过各层面的调查，最后终于发现是 PXC 的自调整自增列的默认行为，与公司使用的 OR-Mapping 的 Ruby 类库 Active Record 不匹配。PXC 默认会开启 wsrep_auto_increment_control，每个节点的自增列步长会根据 PXC 节点数量的变化自动设置，而 Active Record 的某些函数假设新增列的步长为 1，并没有考虑到 >1 的情况。最终关闭 wsrep_auto_increment_control 并保持单点写入作为

解决方案。

关于 GTID 空洞的坑

目前我们的实践经验是，GTID 或者 PXC 单独使用都可以，如果二者同时使用，某些情况下会导致 GTID 产生空洞，再切换 Slave 时不能自动转接，需要 DBA 人工干预。具体细节如下：正常 GTID 为 UUID:X1-Y1 的格式，但有时会有：UUID:X1-X2:X3-Y1（X3>X2+1，也就是不连续）的情况发生。这里提醒读者，一定注意小心验证，防止 Slave 不能 Switch Master 的尴尬问题发生。具体原则就是 Slave 的 UUID 一定要是 Master GTID 的子集。一般说来可以通过 SELECT GTID_SUBSET(“uuid:xxxxxx”, “uuid:yyyyyy”) 来提前判定，源代码层面以下代码段的检查需要正确返回才可以：

```
mysql-server/sql/rpl_binlog_sender.cc(5.6):707:
int Binlog_sender::check_start_file()
{
    .....
    if (!m_exclude_gtid->is_subset_for_sid(&gtid_executed_and_owned,
                                             gtid_state->get_server_
                                             sidno(),
                                             subset_sidno))
    {
        errmsg= ER(ER_SLAVE_HAS_MORE_GTIDS_THAN_MASTER);
        global_sid_lock->unlock();
        set_fatal_error(errmsg);
        return 1;
    }
    if (!gtid_state->get_lost_gtids()->is_subset(m_exclude_gtid))
    {
        errmsg= ER(ER_MASTER_HAS_PURGED_REQUIRED_GTIDS);
        global_sid_lock->unlock();
        set_fatal_error(errmsg);
        return 1;
    }
}
```

不兼容 SQL 的问题

这里不再多说，GTID 启用备战时，以下两个场景测试都被我们命中，得以提前推动开发团队进行完善。这里提醒读者的就是一定要采用尽量完

备的测试用例，不然上线后会非常被动。

- i. 显示事务内的 create/drop temp table 不支持
- ii. create table... select statement 不支持

SST/IST 对性能的影响

为了尽量避免 PXC 的 SST(全量恢复)，我们做了定制化设置：

```
wsrep_provider_options="gcache.size=1G"
wsrep_sst_donor = pxc02
[sst]
rlimit=70m
```

gcache.size 用于尽量使用 IST，wsrep_sst_donor 用于保护 PXC01 的 IO 性能；

rlimit 用于在 IST/SST 时三节点之间的网络限速。而且必须安装 linux PV command。

监控设计和实施

Public/Private VIP 无损非预期跳转的预警

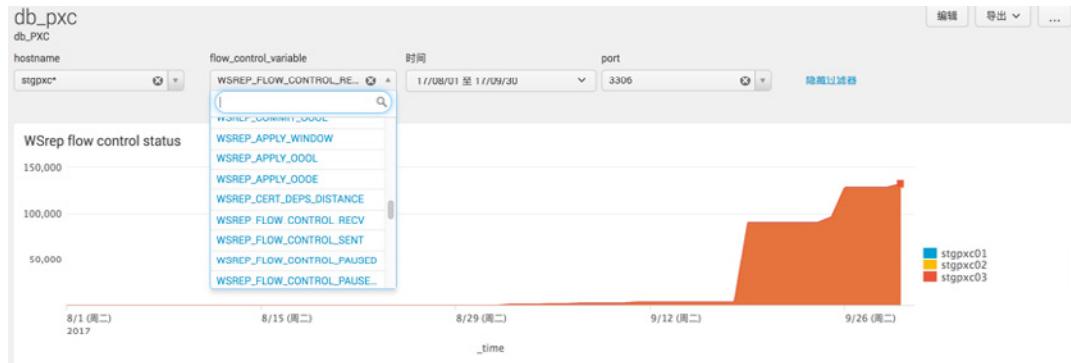
这里我们使用 Nagios 调用“select @@hostname”监控 VIP 可达的 MySQL 为节点 PXC01，否则报警，使用 Splunk 2 分钟一次获取当前可达节点保存为历史数据；



提早预警无损的 Flow Control

首先，Flow Control (FC) 不严重的情况下并不需要过度紧张，但需要持续严密监测趋势、发起者、被动接收者等，我们采用 Splunk 监控了

所有 wsrep_flow_control* 的参数值变化，为后期分析做好了准备。当突然有 wsrep_flow_control 收发次数，或者 wsrep_flow_control_paused 等参数出现逐渐升高时，需要注意密切监控。



集群架构的反思

PXC 上线平稳运行一段时间后，我们逐步开始认识到 PXC 在一些痛点上并不能给我们提供帮助，比如：

- i. PXC 并非 Scale Out 方案：当单机的容量发生瓶颈时，很难真的依靠 PXC 做读写能力扩容，因为它和 HA 的本意是矛盾的。目前我们使用的规模为 3 台集群，还没有试验过更大规模。
- ii. 一个大集群 VS 多个小集群：笔者的理解倾向于把大集群拆分成小集群，标准化做到极致的很多个小集群，在运维代价和灵活性上都有很大优势，笨重的大集群终究是整个系统的瓶颈和要害。
- iii. 集群数据准入原则：核心集群要尽量精简，包括数据进入、数据访问、复制 Slave 的引入等。一般的演化是“合久必分，分久必合”，但差别在于后边的合并和集中是更理智，精确可控的集中。
- iv. 数据访问层的需求：DAL 可以作为数据访问的统一入口，方便审计、权限管理、SQL 分析、性能统计、下层数据库变更的透明化。
- v. 非关系性数据的分离：Log、文件、图片、JSON、映射表等非关系型数据，一定要考虑转接到 Key-value、文档存储型数据库或服务上。

vi. 脑裂：分布式系统必须要考虑脑裂问题。PXC 采用 Majority Quorum 的策略，集群节点数必须为奇数，必须配合 VIP 的可写探测，才能达到脑裂时自动转接的能力。在平衡复杂性和实际网络稳定性的基础上，我们做了取舍，只使用了只读可达的探测。但当三节点跨 DC 时，这其中的风险一定要充分考虑。

DBA 团队在最初驱动这个项目时真的未曾想到，随着调研和项目的深入，会出现诸多潜在问题和拓展知识需要深入探究，中间更是驱动了全公司北京和纽约等全球各个部门（开发、测试、网络、甚至系统层面）的同事一起协同、有序工作，才真正得以实现。除了技术层面的挑战以外，对于人员之间的交流沟通能力，项目的协调安排，甚至是忍耐和对压力的承受能力都是极大的考验。DBA 团队也在这个过程中主动或被动地收获了个人的进步，提高了应对复杂项目的能力。某种程度上来说，改造现有的线上体系甚至比重新建立一个新体系要困难得多。

PB 级海量数据的处理查询之道： 大数据平台最佳实践

随着大数据技术和业务需求的发展变化，数据应用对实时性要求越来越高，以流计算和离线处理为代表的 Lambda 架构成为大数据平台的事实标准。如何借助大数据基础组件快速开发流计算框架与查询应用成为这其中的核心问题。本文将从以下三个方面展开论述：Lambda 架构构建大数据平台的实践经验；AWS 部署实践；OLAP 查询演进。

一、Lambda 架构构建大数据平台的实践经验

1. Lambda 架构概述

我们的数据来源于广告的投放和展示，我们的服务部署在全球 6 个数据中心，每天有近 10 亿的展示日志，累计产生超过 3TB 的视频广告数据，并且有 18 个月的数据需要持久化。

广告数据在视频过程中通过投放和展示产出，这种类型的数据是一个 Request 跟随多个 ACK 记录。在数据生成和传输过程中，我们使用 Google

Protocol Buffers 来作为数据存储格式，嵌套数据结构，经过我们的处理框架，以 Parquet 文件形式存储到 HDFS。

根据我们的数据内容，我们提供 4 种类型的数据应用。一是报表统计类，负责为客户产生各种形式的统计报表；二是追踪业务流量的变化，为客户提供定制化实时日志；三是分析查询服务，也就是 OLAP；四是预测类应用，为我们线上广告服务器提供反馈。

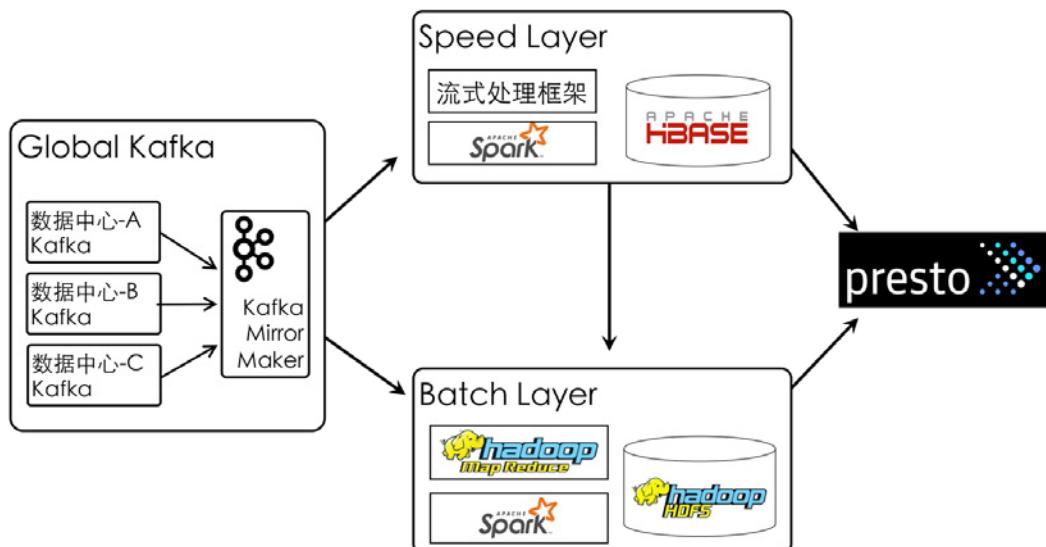


图 1 以 Hadoop/Kafka/HBase 为核心的 Lambda 框架

基于我们的数据特点和应用需求，我们在内部采用一套以 Hadoop、Kafka、HBase 为核心的 Lambda 处理架构（如图 1 所示）。各个数据中心的广告服务器实时地传输数据到 Kafka Cluster，然后 MirrorMaker 会把多个数据中心的数据聚集到一个全局的 Kafka Cluster。我们的流式处理框架会从全局的 Kafka Cluster 消费数据，处理之后写入 HBase Table，并由 Hadoop 离线作业将 HBase Table 转换成 Parquet 文件。OLAP（Presto）服务会同时查询 HBase 和 HDFS，对外提供数据的实时查询。针对 Kafka Topic，HBase Table 和 HDFS 上 Parquet 文件，还支持 Spark 和 MapReduce 作业进行处理。

2. 分布式资源管理平台

基于 Lambda 架构的需求，我们将分布式平台的服务分为两类：一类是计算服务，一类是核心服务（如图 2 所示）。计算服务追求吞吐和速度，核心服务关注服务的延迟和稳定性。计算服务包括各种类型的批处理、流式计算、内存计算以及 OLAP 查询相关的计算。为了统一管理，所有的计算服务都要部署在 YARN 上。核心服务包括分布式存储 HDFS、HBase、监控组件和系统内核服务。

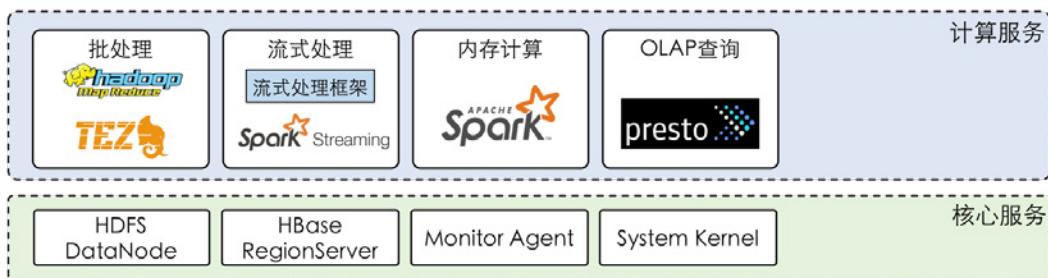


图 2 分布式平台服务划分

为了共享资源、提高资源利用率，这些服务在每台机器上都会部署。在计算服务这一层，我们统一使用 YARN 来管理。那么，在这样的一个背景下，如何解决计算服务与核心服务的资源竞争，在保证服务稳定的前提下，更有效地使用资源，成为了 Lambda 架构面临的重要的挑战。

(1) CPU 资源的竞争

不受限制的计算密集型服务负载变化存在不确定性，我们可以看到机器存在一些时刻，基本全部的 CPU 被计算服务占用，直接影响底下核心服务的质量。如图 3 所示，这台机器在持续高 CPU 情况下，直接导致了机器宕机。计算服务导致服务器异常高负载以及宕机，影响核心服务的稳定性。

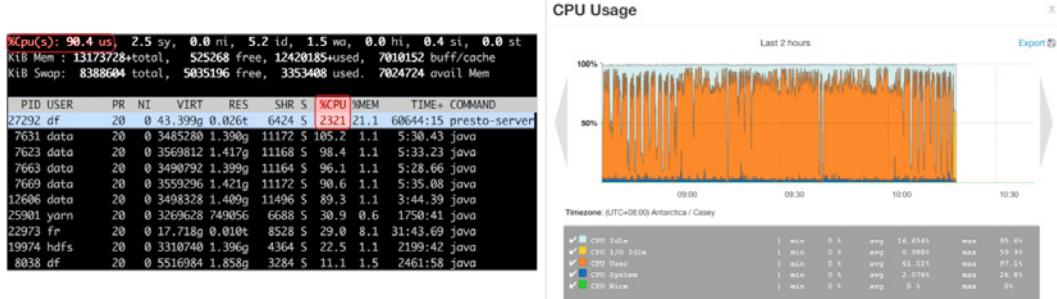


图 3 计算服务高负载的影响

(2) 网络 I/O 资源竞争

Presto 是我们分布式平台的数据分析服务，它负责对超过 18 个月的数据进行 SQL 查询，图 4 是 ATOP 工具记录的各项系统指标，可以看出 Presto-server 占用了超过 83% 的网络资源。对于网络 I/O 资源的竞争，会直接影响到 HDFS DataNode 传输 Block 的效率以及 HBase 的延迟。图 5 可以看到 HBase RPC 的服务延迟受到了带宽的影响，RPC 延迟会随着节点网卡带宽的负载的升高而变大。

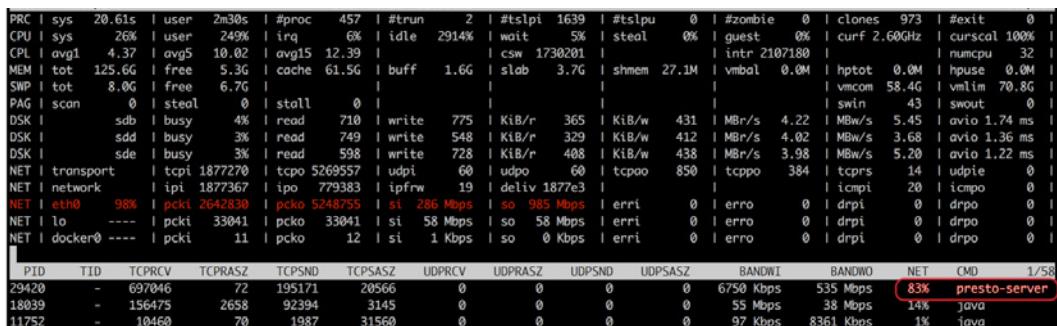


图 4 计算服务网络高负载的 ATOP 截图

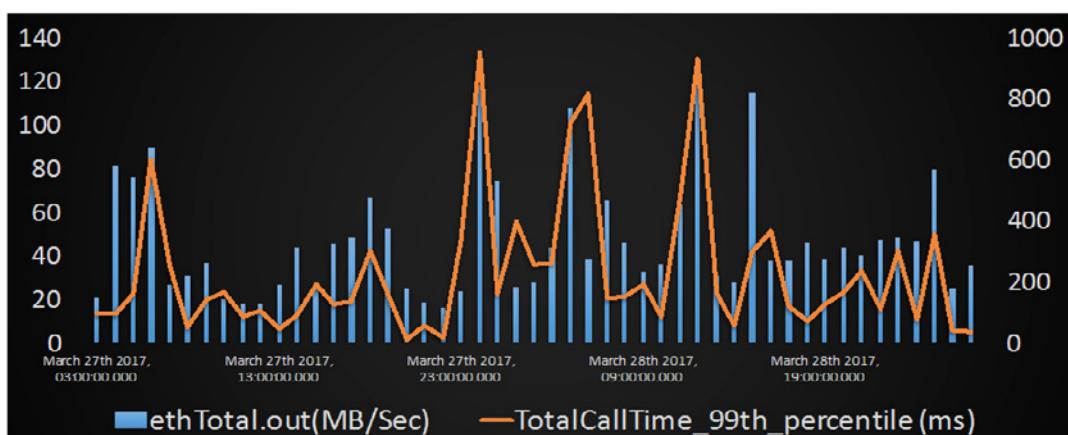


图 5 网络带宽和 HBase RPC 延迟的关系图

因此，基于 CPU 和网络 I/O 资源的隔离，是计算服务和以存储为目的的核心服务共存的必要条件。为了解决资源竞争的问题，我们采用了 Linux CGroup 技术，分别从 CPU 资源和网络资源进行隔离。

(1) CPU 资源隔离

计算服务和核心服务使用 cpuset cgroup 实现隔离。例如，我们服务器有 32 CPU 核心，我们将前 8 个 CPU 核分配给核心服务使用，后面 24 个核分配给计算服务。

在每一层服务内部，我们使用 cpu. shares 实现资源竞争共享资源。在这一点上，YARN 本身支持基于任务的 vcore 设置 cpu. shares，灵活实现了共享式竞争。

如图 6 所示，对比测试显示了基于 CPU 隔离能够有效控制机器的资源利用率。机器 B 是设置了 cpuset 隔离，机器 A 是没有设置 cpuset，可以看出在同样计算任务负载的情况下，机器 B 可以很好地控制资源利用率，保证了核心服务的稳定性。

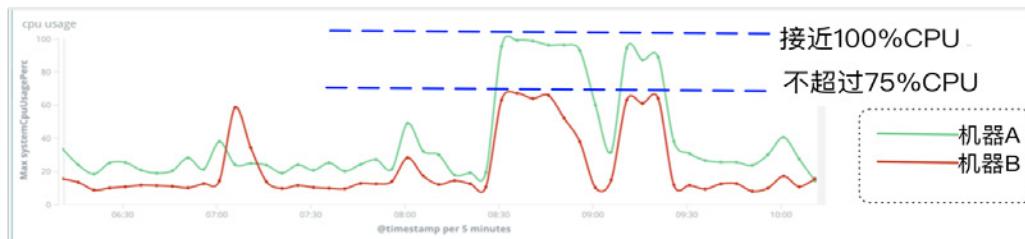


图 6 系统 CPU 利用率对比图

通过粗粒度 cpuset 隔离和分层内部细粒度 cpu. shares 的竞争，使得我们在保证核心服务的 CPU 资源的同时，允许我们的计算任务在允许的范围内竞争资源。

(2) 网络 I/O 资源隔离

在网络资源的隔离上，我们是通过 net_cls cgroup 对于进程按照网络资源进行分组，然后 tc(htb 模式) 限制网络 I/O 资源。例如，在我们的分布式平台，我们分别对于 HDFS、HBase、YARN 计算服务进行网络带宽限制。

节点网卡总带宽 1000mbit，我们对于 YARN、HBase、HDFS 按照 480mbit、260mbit、260mbit 进行分配。设置代码示例如下。

```
#设置tc htb类别和流量控制规则
$ tc class add dev eth0 parent 1: classid 1:1 htb rate
1000mbit
$ tc class add dev eth0 parent 1:1 classid 1:10 htb rate
480mbit ceil 1000mbit
$ tc class add dev eth0 parent 1:1 classid 1:11 htb rate
260mbit ceil 1000mbit
$ tc class add dev eth0 parent 1:1 classid 1:12 htb rate
260mbit ceil 1000mbit

#创建cgroup net_cls分组(yarn-service, hbase-service, hdfs-
service)
$ cgcreate -g net_cls:/yarn-service
$ cgset -r net_cls.classid=0x10010 yarn-service
$ cgcreate -g net_cls:/hbase-service
$ cgset -r net_cls.classid=0x10011 hbase-service
$ cgcreate -g net_cls:/hdfs-service
$ cgset -r net_cls.classid=0x10012 hdfs-service

#设置tc filter按照cgroup进行过滤
$ tc filter add dev eth1 parent 1: protocol ip handle 1: cgroup

#启动服务进程并加入对应的cgroup
$ cgexec -g net_cls:{$group_name} start-service-command
```

经过调整之后，YARN 计算服务在整个环境下的网络带宽使用被得到更合理的控制，这样在核心服务存在较高网络 I/O 需求时，服务能够获得稳定的吞吐和延迟。例如，我们通过实验测试了在 YARN 启动数据密集作业的同时，提交大规模的 HDFS/HBase 读写的场景下，对比了设置网络资源隔离带来的效果，如图 7 所示。在不影响整体网络 I/O 资源使用率的前提下，有效保护了 HDFS、HBase 核心服务的带宽。结合 CPU 资源隔离，有效地实现了核心服务、计算服务共置机器，资源高效共享。

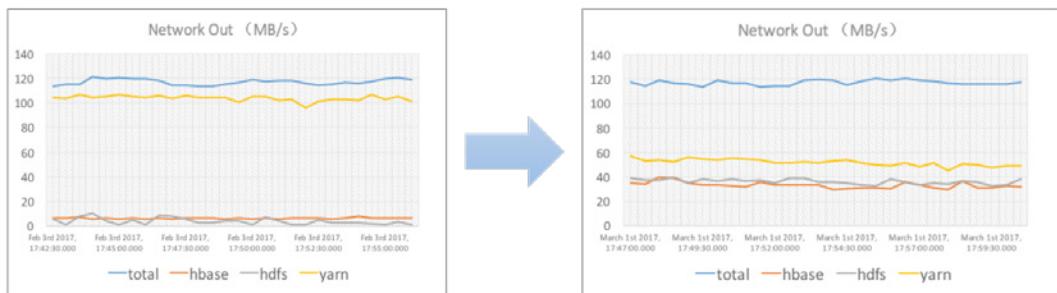


图 7 网络资源隔离效果图

3. 应用编排和 CD

通过 Linux CGroup 技术解决了 Lambda 架构的服务如何在同一个环境中更好地共享资源的问题，我们的计算服务由 YARN 统一调度和部署。那么如何让服务类型的作业部署到 YARN，成为了我们需要解决的一个问题。

服务类型的分布式应用，它需要解决如下几个关键问题：

第一是维护应用每个组件的状态。我们认为部署在 YARN 环境的应用，需要满足 Share-nothing 的分布式架构，从而可以保证在机器宕机等不可抗拒的因素的干扰下，服务的正常执行。

第二是维护内部逻辑拓扑。流式处理作业，一般包含多个层次的逻辑关系，需要一种语言来描述不同层次之间的依赖，保证服务的正常执行。

第三是支持弹性伸缩。在逻辑和依赖允许的情况下，支持按需增加和减少实例个数。

基于此，我们采用了 Apache Slider。Slider 是一种分布式应用框架，它帮助一个分布式应用可以灵活地部署到 YARN 上。分布式应用可以使用 Slider 描述出资源需求和内部的逻辑层次，Slider 可以监控服务的状态，并结合 YARN 的特性，实现自动恢复。

本质上，Slider 应用就是运行在 YARN 上的 Application，它也是有一个 AppMaster 作为中心任务，Slider Agent 被部署到 YARN 的工作节点，用来启动子任务。我们 Slider Client 可以通过 API 向 Slider

AppMaster 提交请求，通过 Slider Agent 执行相关管理工作。

Slider 支持 Docker 化的应用，每个工作节点会通过 Docker Pull 从 Docker Registry 请求指定版本的 Image，并按照 Agent 的指令启动 Docker Container。Slider 部署 Application 的架构图如图 8。

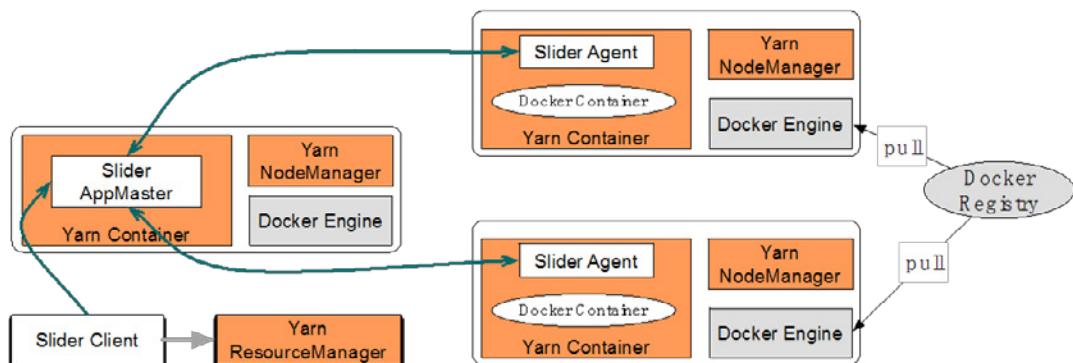


图 8 容器类应用 Slider 部署

在我们的流数据处理框架里，通过 Slider 来管理作业的状态和逻辑，Docker Image 负责业务逻辑和环境的组装发布。在实现按需动态伸缩的过程中，需要在指定节点关闭或者增加实例。这是在 Slider 的系统不支持的，因此，我们扩展了 Slider Flex 功能，实现指定节点启动 / 关闭某个组件的实例。在 Docker 部署实践上，我们解决了 Slider Agent 记录的 Docker 容器状态和 Docker Container 本身状态不一致的问题。为了实现灰度发布，增加支持 App 不退出更新 Docker Image 的特性，动态重启部分实例，实现部分升级的功能。

在完成了对于 Slider 的改造和优化之后，我们将 Slider 用于我们的持续集成（CD）流程中。现在 Slider 支持整体发布和灰度发布两种，可以基于 Application、Component、Instance 进行升级，极大地提升了我们 CD 的流程的效率。

经过 Slider+Docker 对于服务进行组装和改造之后，持续集成（CD）在大数据处理系统里面的流程包括如下几个步骤（如图 9）：

- 1) 通过 Git 提交代码。2) Jenkins 执行 Regression Test。3)

完成 Regression Test 的代码，打包成 Docker Image，提交到 Docker Registry 上。4) 部署到 Testing 环境。5) Testing 环境验证无误之后，提交到 Production 环境。

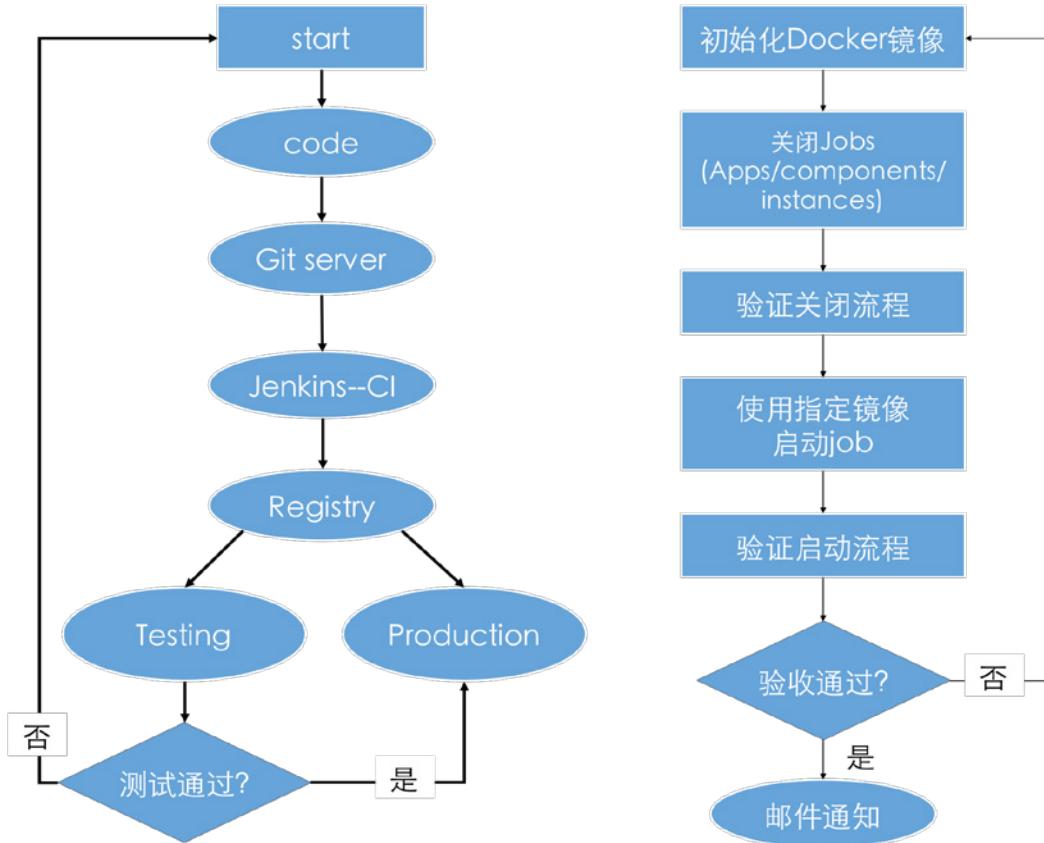


图 9 Docker 容器化的持续集成

在 Testing 和 Production 环境中，根据升级的需求选择合适的发布策略，例如，如果 Application 升级之后的输出数据和输入数据有变化，会做整体升级，对于模块代码 Bug Fix 或者内部优化，在不改变处理逻辑的情况下，对 Component 会进行灰度发布。

二、AWS 部署实践

在 FreeWheel 部署环境上，为了获取资源使用的弹性，我们引入了 AWS 部署方案。实现了一套代码，多套环境的灵活部署。

公有云有实时供应，灵活配置和资源隔离（多租户）等优点。根据公有云发展的趋势报告，混合云部署成为了当下流行的趋势。在这种趋势下，我们迁移到新的集群仍然要考虑如下因素：应用的可移植性、公有云开销、功能和数据容量。

对于我们以 Hadoop 生态系统的大数据架构而言，在 AWS 部署有两种选型：一种是使用 AWS EMR 的服务，另外一种是使用 EC2 和 EBS 创建 Hadoop 集群的服务。

AWS EMR 是 Hadoop 组件的集合，它包括硬件资源和 Hadoop 集群系统。EMR 的优点是易于管理，但是我们在使用过程中，发现它目前还存在单点故障，HBase on S3 极其不稳定。对于我们需要长时间稳定运行服务和作业而言，服务的稳定性要求较高。

因此，我们采用了第二种方案。考虑到存储服务（HDFS、HBase）和计算服务（YARN）它们之间的影响，我们从设计之初就使用不同类型的机器分别部署这两类服务，实现存储和计算的分离。在解决单点故障的问题上，我们使用了 Hadoop 开源系统已经比较完善的 HA 方案。在存储方案的选择上，我们使用 S3 和 HBase（HDFS）分别存储历史和实时数据。

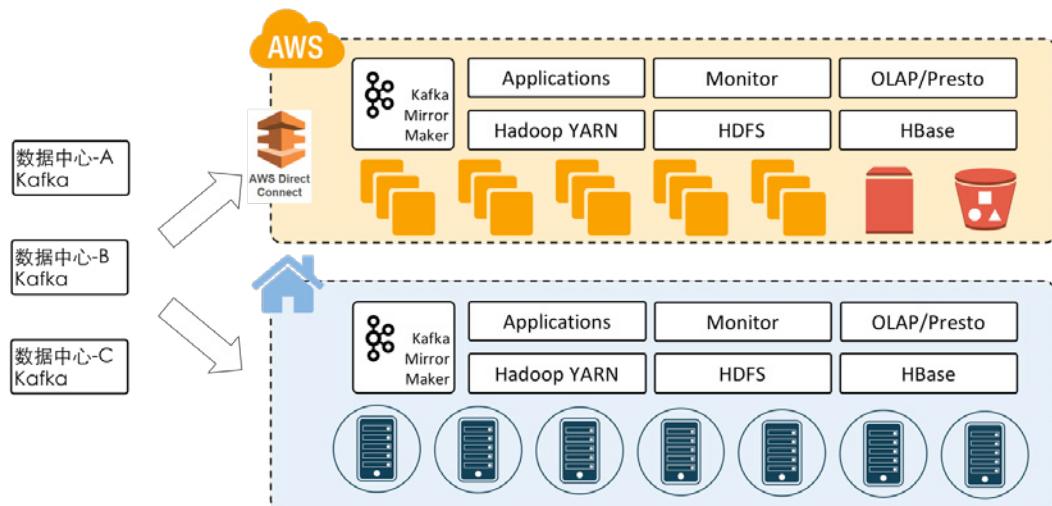


图 10 AWS 和 On-premise 混合部署

我们的数据处理流程和查询服务在 AWS 和 On-premise 分别部署（如

图 10），两者实际上是 Active-Active 的模式，两者之间也互为容灾。由于两套环境都是采用相同的技术栈和部署方案，这给我们后续服务的扩展和研发带来了极大的便利。我们可以先在 AWS 验证业务的可行性和做资源规划，然后再选择 On-premise 和 AWS 的环境部署。

三、OLAP 查询演进：基于 Presto 的数据查询服务

在 FreeWheel，每天会产生超过十亿条数据，这些数据包括视频广告的投放信息，观看点击信息等。对这些数据，我们需要保留过去 18 个月的数据，所以目前超过 200T 的数据需要进行查询分析，并且还在增长。

数据上，我们摊平的表有将近 600 多列，并且上游数据还在不断增加新的数据列。为了查询效率，存储上我们使用 Parquet 进行列式存储。

大概三年前，FreeWheel 就开始关注并选择了 Presto 作为分布式查询引擎，当时我们也注意到很多公司也在用 Presto。我们选择了 Presto 作为查询引擎因为 Presto 可以满足我们很多要求，首先我们看一下 Presto 是如何工作的。

1.Presto 的架构及优势

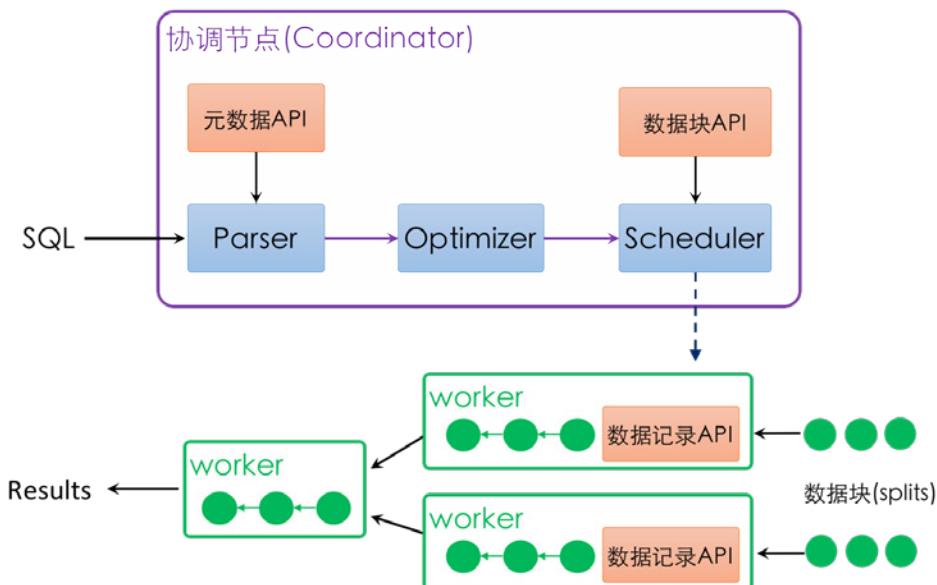


图 11 Presto 架构图

Presto 是一个由 Facebook 开源的大规模分布式查询引擎。从图 11 可以看出，Presto 主要有两种角色的节点，分别是 Coordinator 和 Worker。Coordinator 就类似系统中的 Master 节点，当用户提交一条 SQL 到 Presto 时，实际是 Coordinator 最先开始工作，Parser 负责分析和检查 SQL 的合法性，生成抽象语法树（AST）。然后 Parser 会把 AST 交给优化器，优化器一方面负责根据一些特定的规则对 SQL 进行优化，一方面生成逻辑执行计划，将 SQL 执行划分为若干个阶段。最后由调度器将任务划分成更细粒度的 Task 任务，并分发给工作节点。

工作节点一旦收到任务后就会立刻开始加载数据和计算，整个过程是全内存流水线式的，也就是说 Presto 不会像 Hive 一样把 SQL 规划成多个 MapReduce Job，而是一条内存中的数据流水线。这样一来从两个方面提高了执行速度，一是 Presto 在执行任务时不会把中间结果写磁盘，节省了很多磁盘的 I/O 操作；二是流水线模式下，各个阶段之间并不是完全的强依赖关系，先计算完的数据可以尽快进入下一个阶段的计算。这是 Presto 效率高的重要原因。

此外，Presto 作为查询引擎，在设计时考虑了对多种数据源的支持，在计算框架之外提供了一系列接口用来和数据打交道，图中红色的部分就是可以针对不同数据源进行扩展的接口（部分）。这是我们喜欢 Presto 的一个重要因素，它允许我们自己实现元数据提供的方法，数据读取的方法等。

此外，Presto 可以支持多种数据源 join 在一起，这对我们也很有帮助。因为我们在做数据分析的时候，经常需要 join 纬度表，这些数据可能在其他的关系数据库中，甚至可能就是 HDFS 上一个文件。Presto 这种对多数据源的支持非常方便我们做这种操作。

最后，Presto 支持标准 SQL，这使得用户的学习成本变得很低。

2.Presto 的使用

当选定了 Presto 作为我们的查询引擎之后，我就有了如图 12 的 Presto 服务。

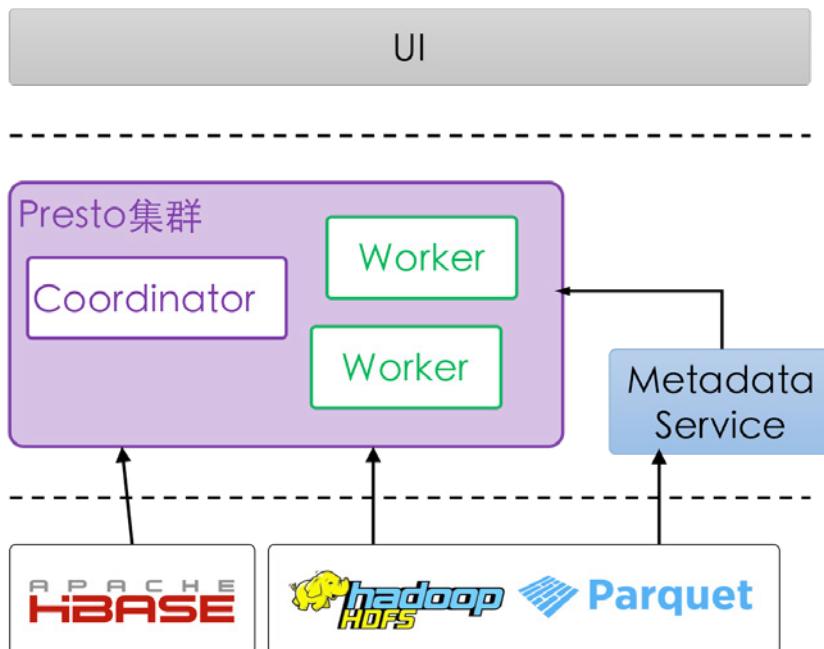


图 12 Presto 服务架构图

数据存储在 HBase 和 HDFS 上，HBase 里存的是动态的最新的数据，而 HDFS 中存储的是相对稳定的数据 Parquet 文件。这是我们的流式架构和业务共用决定的，但是对上层用户，我们不希望有这种差别，也就是说我们希望对这两种数据的融合查询应该是对用户透明的。

为了达到这个目的，我们实现自己的 Presto Plugin，实现了对两种数据的共同读取和解析。

此外，为了提高查询速度，增加查询的谓词下推功能（Predicate Pushdown）。也就是说，我们可以根据查询条件，比如时间范围、客户 ID 等，查询元数据来减少数据扫描的范围，按照查询需求设置 Presto 读入的数据内容。

要实现这一目标，我们需要知道当前存储数据的元数据。我们每天产生上万个 Parquet 文件，虽然 Parquet 文件中存储了数据块的信息和进行谓词下推优化所必须的信息，但是每次执行都要打开文件去读元数据显然是效率低下的。因此我们建立了一个独立的元数据服务，用于预先读取、缓存、更新 HDFS 上文件的元数据，对外提供 RESTful 查询接口，用来接收 Coordinator 的查询，和提供准确的数据块元数据。

3.Presto 多集群部署

在内部使用了一段时间后，我们统计了 Presto 的使用情况。从图 13 里可以看出，超过一半的查询可以在 1 分钟内完成，80% 可以在 5 分钟内完成。然而，我们也发现，大概 3% 的查询是超过了 30 分钟才完成的。

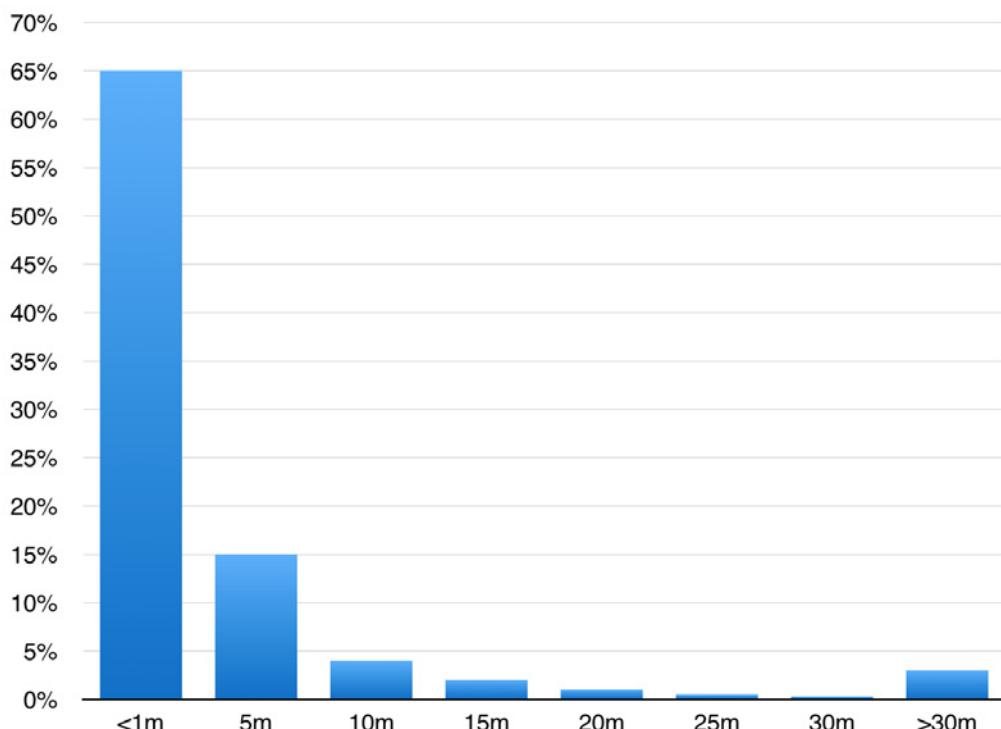


图 13 Query 执行时间的分布图

根据我们平时运维的观察和总结，长耗时查询的原因主要是：

- 1) 基数很大的 group by，或者 join 了很多纬度表；

- 2) SQL 条件复杂;
- 3) 时间范围比较长, 或者涉及很多列, 也就是需要扫描加载的数据比较大。

比较糟糕的是, 我们发现在这种情况下, Presto 集群的响应速度会变慢, 并发执行的其他查询的效率也会有所降低, 查询失败的风险也会增加。并且由于 Presto 不容错 (执行机制是: 一旦查询中某一个环节出错, 整个查询都会失败), 所以长耗时查询的失败重试的成本也会增加。

通过分析我们发现, 这些长耗时查询很大程度上是可以预测的, 换言之, 是有能力把这些长耗时拎出来的, 因此我们就希望由多个集群代替原来的单集群模式。

根据 Slider on YARN 的扩展功能, 这里我们做的一个改造就是把 Presto 也做成 Slider 应用放到 YARN 里去。我们把 Presto 的启动脚本、参数配置、包括资源申请的配置、还有组件间的启动依赖都做到了 Slider 里。这样做首先是对运维比较友好, 因为所有东西都在一个 Zip 包里, 简化了部署, 并且可以通过 YARN 容易调配和申请资源。

更重要的, 这种部署提高了应用的可伸缩性。这里的可伸缩性一方面是指运行中的集群, 我们可以运行一条指令就动态地增加或减少工作节点, 另一方面, 我们可以快速地在 YARN 上启动一个新的集群来执行查询服务。而且所有这些集群都会共享元数据服务中的元数据, 不会有数据不一致的情况发生。启动的集群可以应用不同的配置, 可以实现资源的隔离。这样一来, 我们可以把不同应用, 不同需求的 SQL 调度到不同的集群上去运行。

4. OLAP 统一查询平台

有了多集群部署, 很自然的我们就希望有一个统一的查询入口, 在这里我们可以做很多事情:

- **查询预测与限制:** 可以在这一层对查询有一个检查, 看一下查询的

执行代价，根据权限等设置来拒绝或者接受。

- 负载均衡：在可用的集群中挑选负载最低的去执行当前查询，或者根据查询执行代价去调度不同的集群去执行。
- 容错：如果查询在集群中发生内部错误，服务可以自动去重试，而对用户透明。
- 权限控制：可以根据不同类型查询的标志，分配到专属的集群中去。
- 灰度发布：升级时，可以逐个逐批去替换正在运行的集群，而通过服务设置和容错机制，保证上层用户不会感知到升级动作的存在，降低停服时间。
- 审计：记录查询的提交详情。

最后在这层查询服务上，我们会有 Presto 的 UI，可以接受 API 的调用，可以接入 BI 工具等等。以 Presto 为核心服务的统一查询平台如图 14 所示。

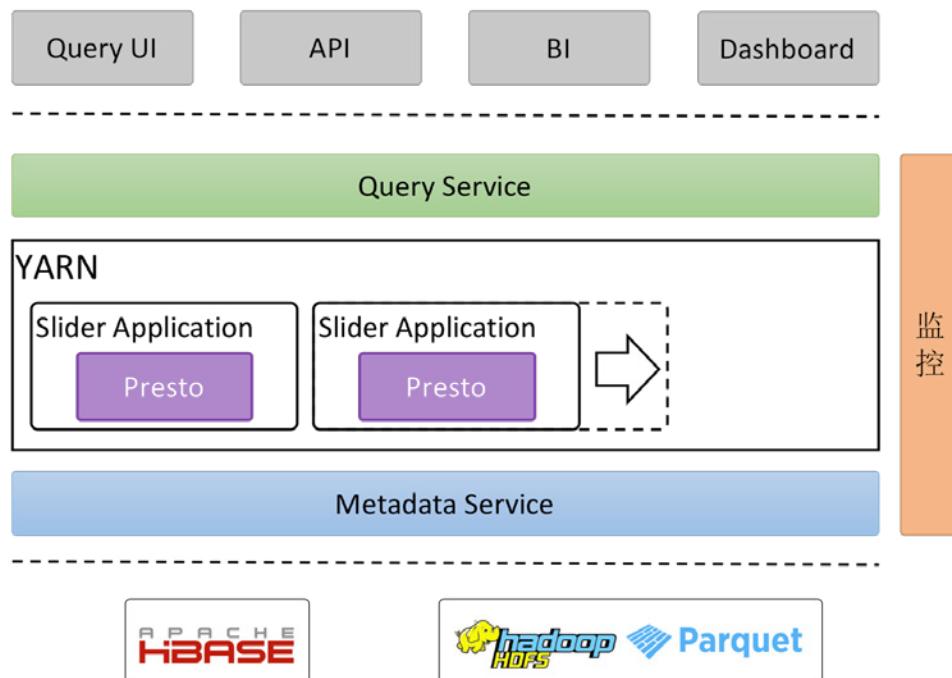


图 14 OLAP 统一查询平台

目前，在FreeWheel内部，这一套OLAP系统正在多种场景下服务于不同的团队：无论是交互式查询、调试系统问题或是生成报表，基于Presto的查询服务都可以满足我们绝大多数需求。

亿级视频广告事件预测系统构建之道

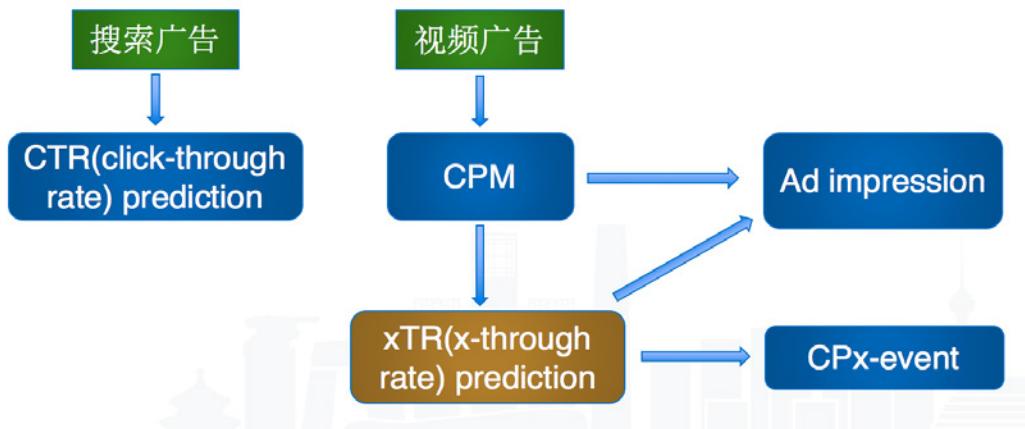
CTR 预估问题广泛存在于互联网主流应用的方方面面，是计算广告，特别是在线视频广告领域的核心问题。它涉及到视频广告的投放、预测和很多的增值业务。

FreeWheel 在此基础上，结合自身的业务特点，将计费模式抽象为 X-event，将传统的 CTR 预估问题转化为 xTR 问题，并构建了一个强大的 xTR 预测系统，用于优化在线广告投放策略。

本文将详细介绍 FreeWheel 在构建一套广告领域 xTR 预测系统的全过程，以实例解答怎么解决在系统构建过程中遇到的机器学习问题，并总结了在实现 xTR 预测系统中的一些经验与教训。全文分四部分：首先是 xTR 问题的由来，然后介绍 xTR 系统架构，最后针对 xTR 的问题定义，介绍我们如何利用机器学习技术实现特征工程，以及模型训练与优化。

xTR 问题的由来

在搜索广告领域，大多会采用点击计费（CPC）的方式进行流量变现，



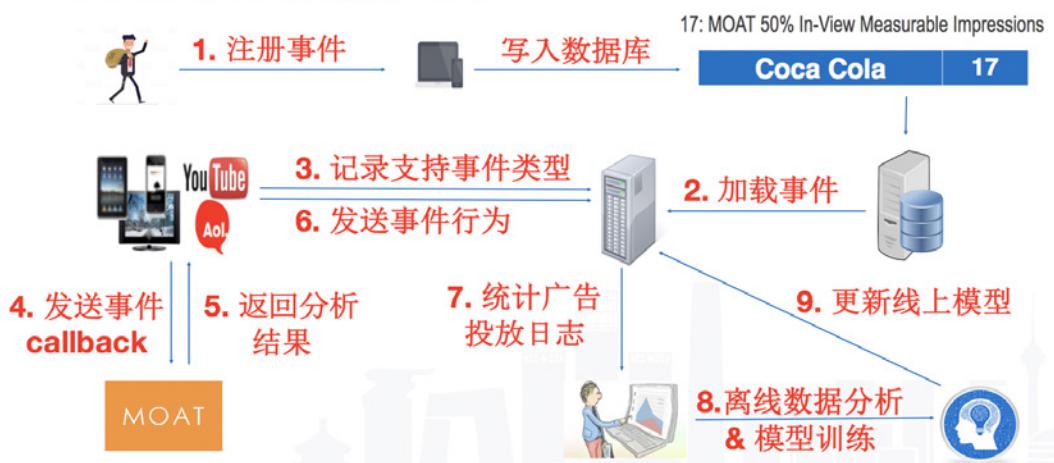
由此衍生出了 CTR 的预估问题。相对于搜索广告，视频广告的产品形态和计费方式都有着很大的不同。传统的视频广告大多会以 CPM 的方式来计费，也就是说会按照用户观看广告的次数来结算。

对于 FreeWheel 来说，广告订单分为合约广告与竞价广告两种，为适应不同客户不同的结算方式需求，从初始的按照 CPM 计费发展到在广告投放过程中按照不同事件发生次数方式计费，即 CPx-event 方式。因此，我们给出 xTR 定义：相对于 CTR，我们将点击泛化为任意一种事件，既包括传统 CPM 中的 Impression 事件，也包括在广告投放过程中发生的任意一种事件，对这些事件发生概率进行估计，就是 xTR (x-through rate) 预估问题。

首先，CPx-event 需要预先注册到整个系统之中。假设我们的客户可口可乐公司希望按照 MOAT 50% In-View Measurable Impressions 方式结算（MOAT 是一家广告分析公司，提供的主要服务是追踪广告的真实触达，即网络广告是否真的会被用户看到）。50% In-View Measurable Impressions 表示可口可乐广告至少 50% 部分被用户看到，才算一次有效触达。

CPx-event 通过以下方式注册：

CPx-event流程图

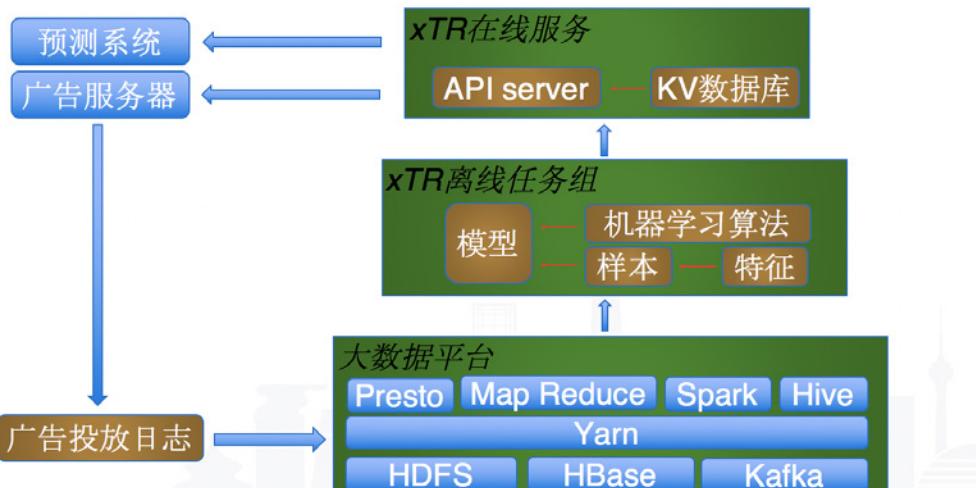


1. 可口可乐公司希望按照 MOAT 提供的监控方式结算，首先他要到广告主管理系统中注册该事件。注册之后，会在数据库中写入这样一条数据，17 是对应事件编号；
2. 广告服务器每天会增量同步数据库信息，将可口可乐公司新的结算方式加载进来；
3. 前端视频播放器与广告服务器建立第一次连接时，会将播放器类型等信息发送给广告服务器，广告服务器根据播放器类型等信息记录当前 Context 支持事件类型；
4. 在视频播放过程中，广告服务器利用 xTR 模型估计某个广告 17 号事件发生概率，并且不断向用户投放广告。在投放某个广告期间，如果 17 号事件发生，播放器会向第三方分析公司发送事件 Callback，经过分析，播放器再决定是否向 FreeWheel 的广告服务器发送事件 Callback；
5. 当视频播放结束，广告服务器统计所有行为并将广告投放日志存储到大数据平台。一方面与广告主结算，另一方面，xTR 基于这些数

据建模和离线训练结果决定是否更新线上 CPx-event 模型，新的线上模型会改进下一次广告投放策略。

xTR 系统架构与流程

xTR 系统架构



FreeWheel 拥有海量的数据资源，日访问量达 15 亿次，在如此庞大的数据上进行 CPx-event 的预估首先需要大数据平台的支持。FreeWheel 大数据平台采用 YARN 管理不同应用，提供 MapReduce、Spark 等分布式计算服务。

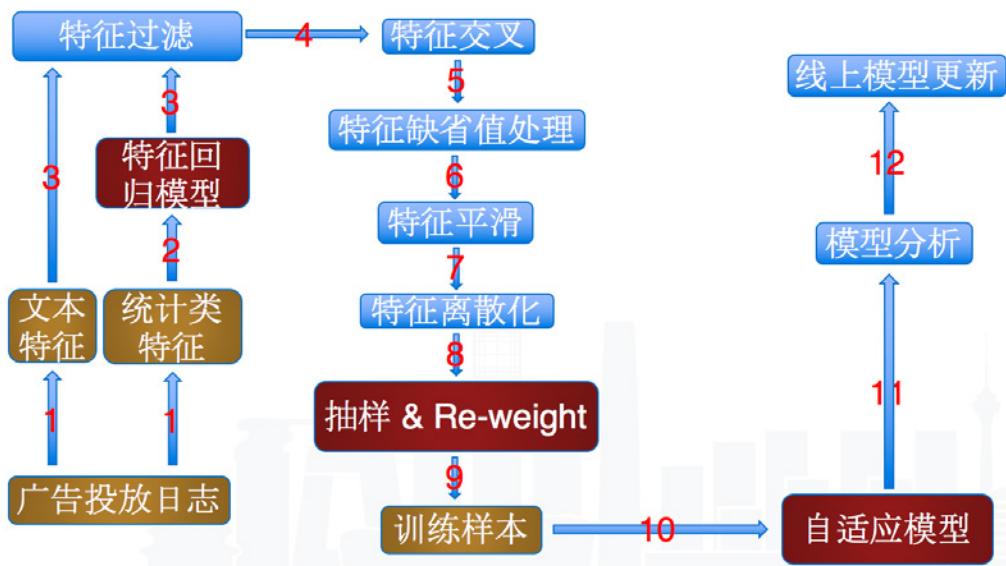
xTR 分为离线任务组与在线服务两部分。离线任务在分布式计算平台处理广告投放日志，对一些上下文信息（比如视频信息、网页信息、LBS、设备信息）抽取特征，根据前端视频播放器返回的事件 Callback 抽取样本。当收到事件 Callback 时认为事件发生，没有收到事件 Callback，则认为事件没有发生，并且选择合适的机器学习算法训练模型，最终将特征信息写入 Key-Value 存储中。

xTR 在线服务对外部系统提供服务请求，我们选择 Thrift 作为 Server 架构。预测系统与广告服务器会实时访问 xTR server，xTR server 经过运算返回预测结果，预测的对象就是某个事件发生概率。

在机器学习方法被应用之前，预测系统通过统计方法计算事件发生概率，即利用之前一段时间事件发生概率平均值估计下一时间点事件发生概率；使用机器学习方法之后，建模时考虑了更多上下文信息，xTR 系统帮助离线预测更加准确，同时帮助在线广告服务器更加精确地预测广告投放过程中不同事件发生概率。

xTR 算法流程

xTR 的算法流程大致如下：



1. 从广告投放日志中解析出文本特征与统计类特征，用特征回归模型解决特征量巨大问题，这是我们第一个优化点；
2. 特征过滤之后，筛选出优质特征，再利用特征交叉丰富特征空间，缺省值处理/特征平滑是我们特征工程主要做的处理；
3. 对于不同事件，使用抽样方法处理正负样本不均匀问题，这是我们第二个优化亮点；
4. 获得训练样本后，我们利用在线学习与强化学习设计出自适应模型算法，这是我们的第三个优化亮点；

5. 最终经过模型分析，更新线上模型。

对于任意一种事件，xTR 的目标是判断该事件是否发生，因此将 xTR 问题转化为机器学习中的二类分类问题。更进一步，可以将 xTR 算法问题分解为特征工程与模型构建两部分。

xTR 特征工程

特征提取

xTR 采用上下文统计类特征与文本特征作为基础特征，上下文信息有视频，网页，位置信息，视频运营商等。

项目初始阶段我们用 One-hot 表示上下文特征。如有 n 个视频，视频的特征空间就是 n 维，其中某一维代表纸牌屋第一季第一集。这种静态特征组织方式的缺点是特征空间巨大，不易训练。

因此我们改进了特征组织方式，选择用上文 + 事件方式组织特征。如当前视频仍然是纸牌屋第一季第一集，在之前一段时间内在该视频上的广告观看率是 0.53，直接利用 0.53 来作为视频特征之一。

这背后的算法原理是，利用不同上下文的弱模型，拟合观看率这样一个终极目标。相对于静态特征，动态特征的特征空间只与事件个数成正比，大大缩小了存储空间，更容易训练。

除了统计类特征，xTR 从视频描述信息中训练出视频分类特征，将视频的分类特征补充到特征空间中。

1. 首先 xTR 离线服务会抽取出视频标题与媒体上传的视频描述；
2. 利用 Dmoz 开源标签体系，对一部分数据做人工标注；
3. 接下来用一些开源自然语言处理工具做分词，常用词过滤等 NLP 处理，经过自然语言处理后，每个视频对应一段由高质量词表示的文本；
4. 将每段文本转换为 One-hot 表示，即映射到 n 维向量空间中，n 为词

表大小，相应位置 1，构成训练样本；

5. 选择合适的分类模型做模型训练。



因为视频分类数量有限，所以我们直接将视频分类结果转换为 One-hot 表示，补充到特征空间中，未来我们也有计划做一些 Embedding 的工作来优化。

加入视频分类特征后，在不同事件中 xTR 预测效果有不同程度提升。

特征组织方式

上下文信息丰富是视频广告的一个天然特性，会导致特征量大，如果将全部特征存储在 Key-value 数据库中，不仅会浪费大量存储空间，同时也会带来线上高并发访问问题。在实际业务中，xTR 在线服务会接受每秒百万级规模并发访问，每一次访问需要几千个特征，即使采用批量读方案，Key-value 数据库也很难支撑。

我们的解法是：只存储被频繁访问到的特征，将大量不经常访问的特征从 Key-value 数据库中移除，对这些特征建模，命名这种模型为特征回归模型。

关于特征回归模型，学习目标是每个特征的特征值。特征值是连续的，所以模型是回归模型，定义特征的特征为“子特征”，由上下文信息的 One-hot 表示组成子特征。

但 One-hot 表示的特征空间巨大，样本非常稀疏，即使优化到特征回归模型这一步，我们还是没有绕开这个问题。

我们的解法是：对从多个上下文组合上抽取出的特征，利用 Factorization Machine 建立回归模型，学习两个子特征之间关系。

One hot vector : n

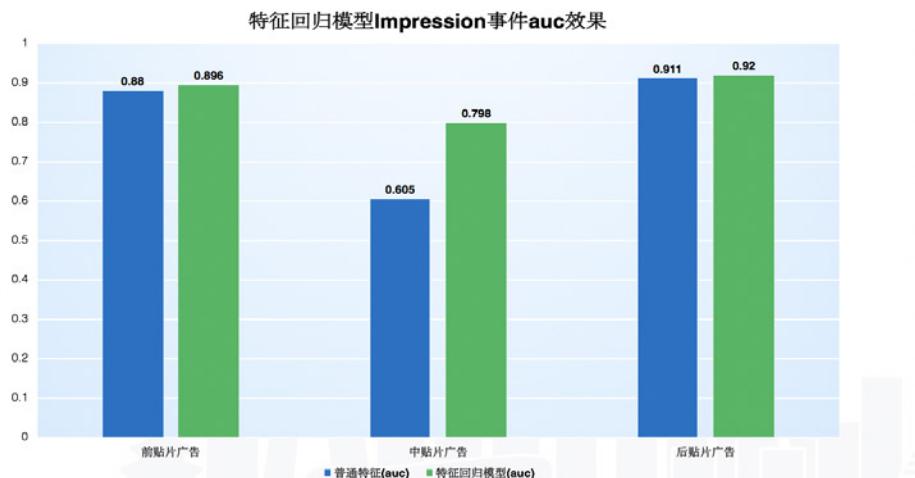
$$\bar{x} = (x_1, \dots, x_n)$$

$$\bar{V} = (v_1, \dots, v_k)$$

$$f(\bar{x}) = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^n (V_i^T V_j) x_i x_j$$

假设子特征空间长度为 n，FM 模型学习出的两个子特征关系保存在两个 V 向量乘积之中，向量长度是实验前预先设置好的超参数。

有了特征回归模型之后，在 Key-value 数据库中只需要存储少量特征即可，大量长尾特征通过模型拟合。



通过对比原始特征与特征回归模型效果，在前贴片广告、中贴片广告、后贴片广告上，二分类模型的 AUC 分别有一定的提升（其中前贴片广告指在视频播放前出现的广告，中贴片广告指在视频播放过程中出现的广告，后贴片广告指在视频播放结束后出现的广告）。

特征筛选

上下文统计特征加上视频分类特征，xTR 的基础特征空间只有不到 100 维。我们利用 GBDT 筛选特征，通过特征的另一种表达丰富特征空间。这种方法与深度学习有异曲同工之妙，不同之处是深度学习利用神经网络学习出高维表示，GBDT 利用决策树做特征筛选。



具体做法是：将 GBDT 中每棵树的叶子节点 0/1 结果作为新特征，补充到原始特征空间中。

加入 GBDT Binary 表示之后的特征在不同场景下 AUC 有不同程度的提升。

$$f(x) = \frac{1}{1 + e^{-(\sum_{i=1}^n w_i x_i + b)}}$$

$$f_{crossfeature}(x) = \frac{1}{1 + e^{-(\sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j + b)}}$$

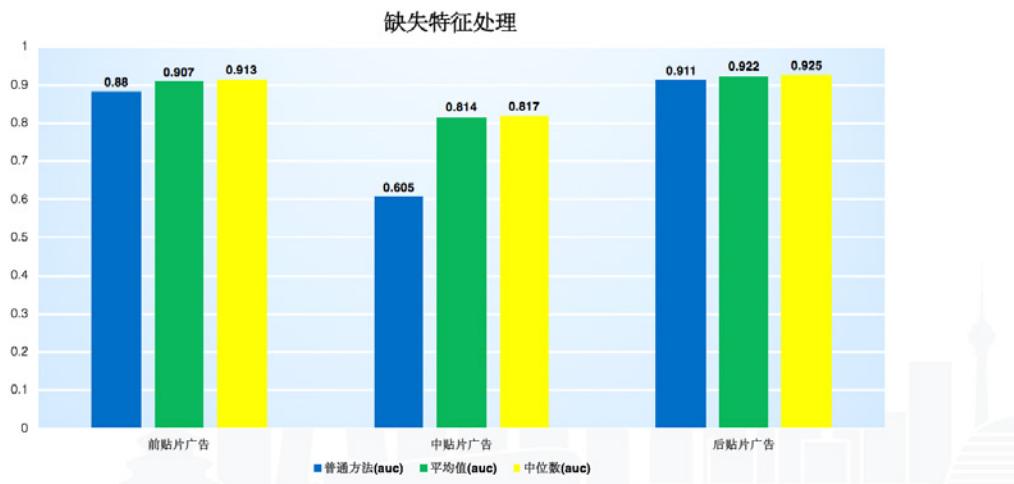
为了泛化模型能力，xTR 从不同特征之间相似性角度挖掘新特征。业界主要采用交叉特征与 Factorization Machine 这两种方法，交叉特征学习出任意两个特征相似值，FM 方法学习出任意两个特征的相似向量，FM 方法更强调解决特征稀疏问题。在我们的业务需求中，样本稀疏没有那么

严重，不需要用向量拟合特征之间关系，因此交叉特征更加适用。

特征预处理

在特征采集过程中 xTR 会根据特征出现次数过滤，即去掉出现次数较少的特征。背后的原理是出现次数越少则特征值置信度越小，所以在样本中会出现有些特征是缺失的，xTR 主要用中位数 / 平均数两种方法取代缺失特征。

不同缺失特征处理方法，效果如图。



不同特征的取值范围不同，取值范围大的特征无形中对模型作用更大，所以用特征平滑来将所有特征限制在相似取值范围内。

Min – max normalization :

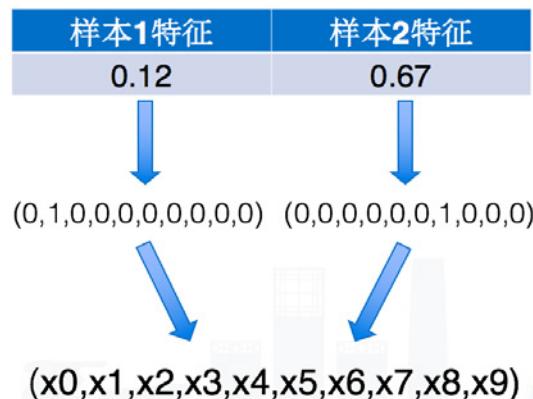
$$x_i(j) = \frac{x_i(j) - x_i^{\min}}{x_i^{\max} - x_i^{\min}}$$

Gaussian normalization :

$$x_i(j) = \frac{x_i(j) - \frac{1}{n} \sum_{k=1}^n x_i(k)}{\sqrt{\frac{1}{n} \sum_{k=1}^n (x_i(k) - \frac{1}{n} \sum_{k=1}^n x_i(k))^2}}$$

在 xTR 中，主要用 Min-max 与 Gaussian 平滑。Min-max 平滑可以使新特征值在 (0, 1) 区间内，Gaussian 平滑可以使新特征符合高斯分布。

对于某些连续特征，不能直接喂给分类模型，需要做离散化。从效率的角度来说，原因是稀疏向量内积乘法运算更快；从模型的角度来说，原因是离散化之后可以增强广义线性模型的表达能力，学习出一个特征不同取值范围的不同权重。



如当前有两个样本，其中的某一个特征值分别为 0.12 与 0.67，特征取值范围是 0 到 1，将 0.12 与 0.67 分别映射到两个 10 维向量中，用这个 10 维向量作为新特征取代原始特征，加以更多样本训练出不同权重。

特征更新策略



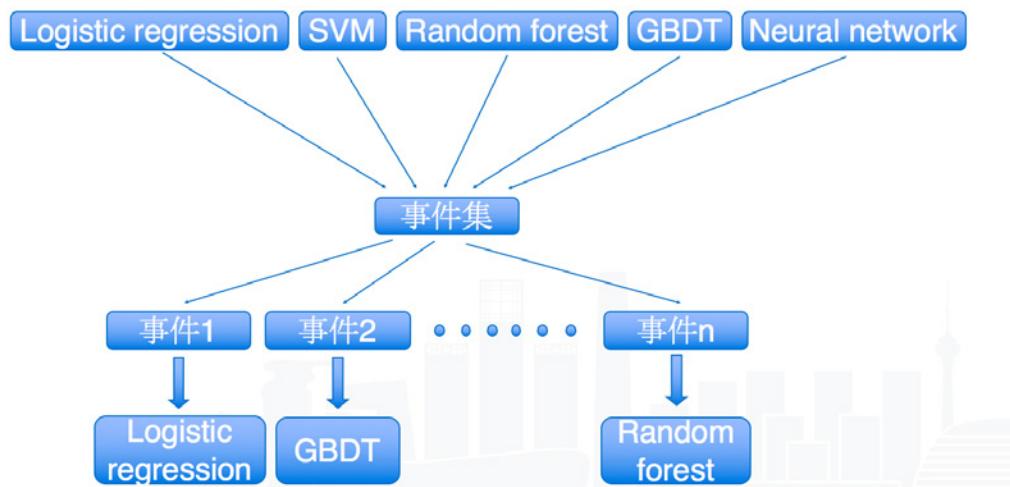
$$\frac{\alpha \times \text{impression}_{\text{pre}} + \text{impression}_{\text{now}}}{\alpha \times \text{delivery}_{\text{pre}} + \text{delivery}_{\text{now}}} \quad \alpha \in (0,1)$$

在实验中，另一个非常有意思的现象是，我们发现模型表现与特征采集周期不是严格成正比，即特征采集周期越长，模型表现不一定更好，这也与业界中的“模型做的大不如模型做的快”这一说法不谋而合。所以我们设计了一个衰减策略，即用一个衰减因子 Alpha 衰减距离当前时间长的数据，这样计算出的特征更加有效。

模型构建

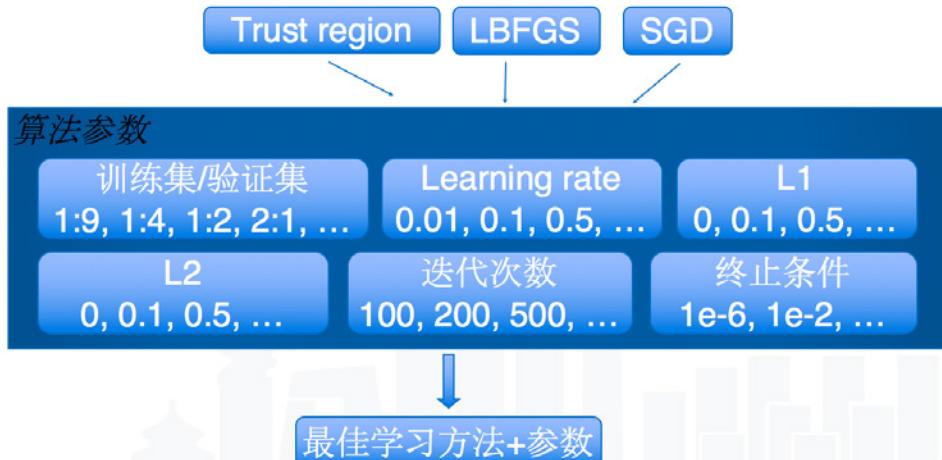
枚举方法

在机器学习领域，解决分类问题的算法很多，比如经典的逻辑回归、支持向量机、随机森林、GBDT、神经网络等等，不同模型适用的数据分布不同。



在实际场景中，不同事件的数据分布不同。如果用同一种方法训练，大部分事件的训练效果会很差，因此需要一种方法为不同事件量身订制最合适的模型。

从统计学角度，对同一种事件分别用不同算法计算出分类结果，对比预测出的分类结果与真实分类结果，选出平均表现最好的模型作为该事件的线上模型。



除了确定模型，还需要确定训练参数与训练方法，我们枚举了不同训练方法与训练参数组合。

比如当前模型是基于梯度学习的，分别用 Trust Region、LBFGS、SGD 三种方法学习，并且枚举训练集、验证集比例，Learning Rate 等训练参数，选出最佳学习方法与参数组合。

这里我们在 Spark 分布式计算框架实现了分布式训练，加速了训练速度。

L1/L2 正则化：用 L1 筛选稀疏特征，当训练数据 / 全量数据比例大，用 L2 防止模型过拟合。

枚举这种方法的缺点是需要事先定义好训练方法与训练参数候选集，完全凭经验来训练，因此需要更好的办法优化模型筛选过程。

在线强化学习筛选模型

对于一个新注册事件，没有训练样本，我们选择用在线学习与强化学习思路解决该问题。强化学习在每次迭代中不仅仅趋向局部最优解，而是在探索(Exploration)与利用(Exploitation)中做出抉择。如果选择探索，则按照某种相对随机策略选择一种解法；如果选择利用，则在候选集中选择最优解法。

1. 首先初始化一些基础模型，比如逻辑回归模型、SVM、随机森林等

等，基本模型的参数使用某种分布进行初始化，将这些基础模型加入到模型空间；

2. 当收到一个新的预估请求，根据 Epsilon-greedy 策略决定选择哪个模型进行预估，也就是图中的 Action 过程；
3. 定义预估值与实际表现之间差距绝对值为 Reward，表示当前估计的偏差，实际表现的定义是：事件发生为1，事件未发生为0；
4. 根据 Reward 信息，利用 FTRL 在线学习算法，更新相应模型参数，也就是 Policy 过程；
5. 同时定义模型的收敛条件是在模型处理一定数量预估请求条件下一段时间内 Reward 变化率，Reward 变化趋于稳定时，认为模型收敛；
6. 模型收敛之后，通过平均 Reward 表现筛选出最优模型。



除了新注册事件，已经积累训练数据的事件也可以按照该方法优化模型筛选过程。

相对于传统有监督学习方法，强化学习思路并不给模型强加很多假设，比如我们之前提前设定的先验模型参数，而是根据真实系统的实时反馈调整 Agent 的策略，这也是强化学习方法能够很好地应用在围棋算法，无人机驾驶等不能用规则囊括所有先验知识场景的原因之一。

这里用数学推导简单描述下我们的算法，首先有 n 个基础模型：

n model space

$$\gamma_{avg} = \frac{1}{t} \sum_{i=1}^t \gamma_i$$

$$\epsilon = \frac{e^{\gamma_{avg}}}{\sqrt{t}}$$

if $\epsilon \geq \mu_{threshold}$ then exploration
else exploitation

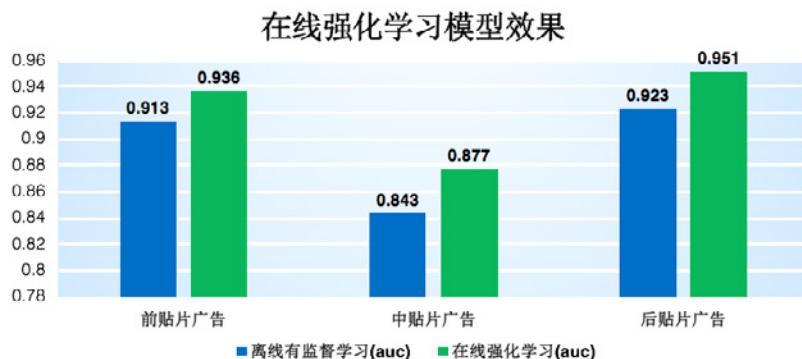
exploration(softmax):

$$p(m_t^i) = \frac{e^{\frac{\gamma_{avg}^i}{\lambda}}}{\sum_{j=1}^n e^{\frac{\gamma_{avg}^j}{\lambda}}}$$

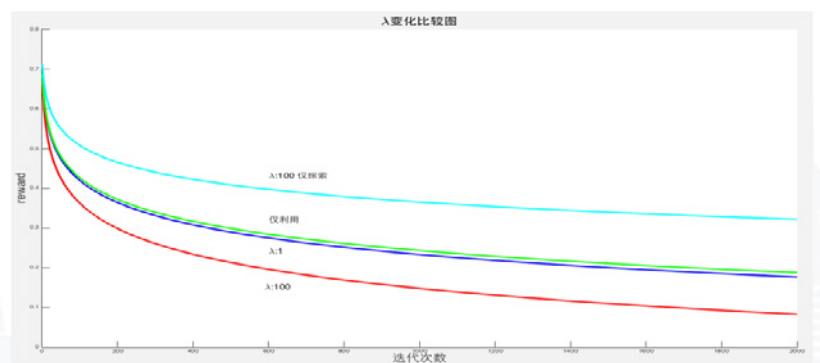
Policy : FTRL update

1. 当进行到时间t的迭代时，根据时间与平均 Reward 计算 Epsilon，用 Epsilon 来决定探索还是利用，Epsilon 代表探索发生概率，随着时间而变，Threshold 是不变的；
2. 通过公式可知，探索发生概率与时间成反比，和平均 Reward 成正比。也就是说在初始阶段更希望去探索而不是在模型集合中找局部最优解。当 Agent 逐渐收敛时，更希望用最优解去预测；
3. 当平均 Reward 较大时，模型表现比较差，我们希望多去探索。当 Reward 较小时，模型表现比较好，我们希望多去利用；
4. 探索发生时，我们利用一种概率分布来筛选模型。在统计学中这个分布叫做 Boltzman 分布， $p(m(t)(i))$ 表示在时间 t 第 i 个模型被选中的概率， $r(avg)(i)$ 表示到目前为止 i 模型的平均 Reward，Lambda 一般较大，用来突出模型筛选随机性。

Reward 比较大的模型被选中的概率会比较小，但通过指数函数，实际上削弱了平均收益的作用，使模型筛选过程更加随机。



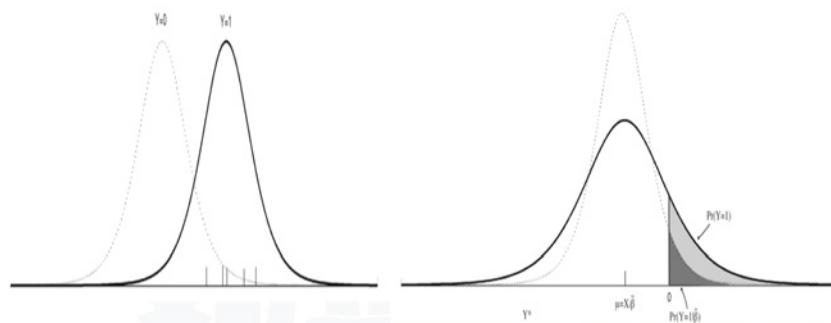
通过对比离线有监督学习与在线强化学习，不同场景下 Impression 事件预估有了不同程度的提升。



对比不同参数下 Agent 的收敛情况，抽取了前 2000 次迭代。最上青色的曲线是 Lambda 为 100 时的仅探索过程，表现最差，因为是仅探索，并且 Lambda 很大，这种方法接近随机；下面的绿色曲线是仅利用；蓝色曲线是 Lambda 为 1 时 Agent 的表现，当 Lambda 为 1 时，效果与仅利用差不多，因为这时只考虑了平均 Reward，相当于每次迭代都选择最优解；红色曲线是 Lambda 为 100 时 Agent 的收敛情况，表现最好，因为它同时考虑了探索与利用两条路，并且在探索中可以做到一定的随机性。

正负样本比例不均衡

模型训练部分还有一个问题，即在不同事件中训练数据的正负样本分布不均衡问题，这个问题其实很直观：出现在视频播放结束的广告能够获得 Impression 的数量很少，出现在视频播放开始的广告能够获得较多的 Impression。



从数学角度来解释这个问题：假设正负样本的特征分布均符合高斯分布，高斯分布方差的最大似然估计是有偏的。为了得到方差的无偏估计，需要将样本数目减 1 来计算方差，而这一偏差导致的后果是对方差有所低估，并且样本数目越少低估越严重。从图形上分析，方差变小后，图像变窄，导致的结果是更多的样本被分到大样本量所在 Label 集合中。

一些论文证明，在二分类问题中，当两类样本的比例小于等于 1:35 时，大部分分类算法基本上就无效了，所以说正负样本比例不均衡这个问题对于模型伤害还是很大的。

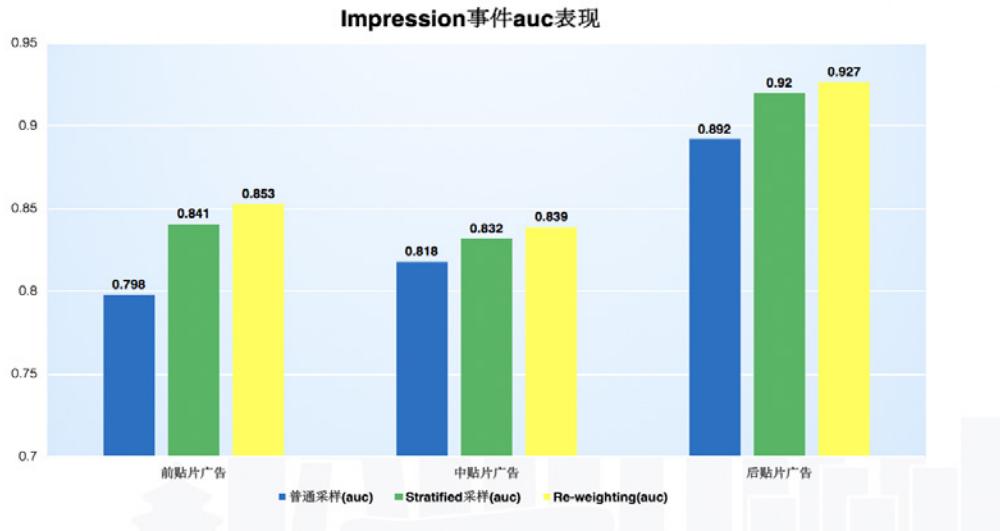
我们主要通过采样与重定义权重方法来解决正负样本比例不均衡问题。采样主要有两种策略，第一种是有偏采样，即小样本量数据全部采样，大样本量数据抽样采样；第二种策略为分层采样，这种采样方法不是根据样本直接采样，而是根据特征来采样，采样的原则是使样本尽量覆盖特征空间，即在一个特征维度中使取值尽量多样化。还有另一种思路是 Re-weighting，通过修改损失函数来突出小样本数据对模型的贡献，原理是当小样本数据被误分时，在损失函数中的惩罚更大。

$$\alpha > 1$$

$$L_i = \begin{cases} \frac{\alpha}{2}(y_i - p_i)^2, & y_i = 1 \\ \frac{1}{2}(y_i - p_i)^2, & y_i = 0 \end{cases}$$

用数学公式来解释，当我们发现训练样本中正样本很少负样本很多时，

定义 Alpha 为大于 1 的实数，当正样本被分错时给它的惩罚更大，以此来减少将正样本误分到负样本的概率。



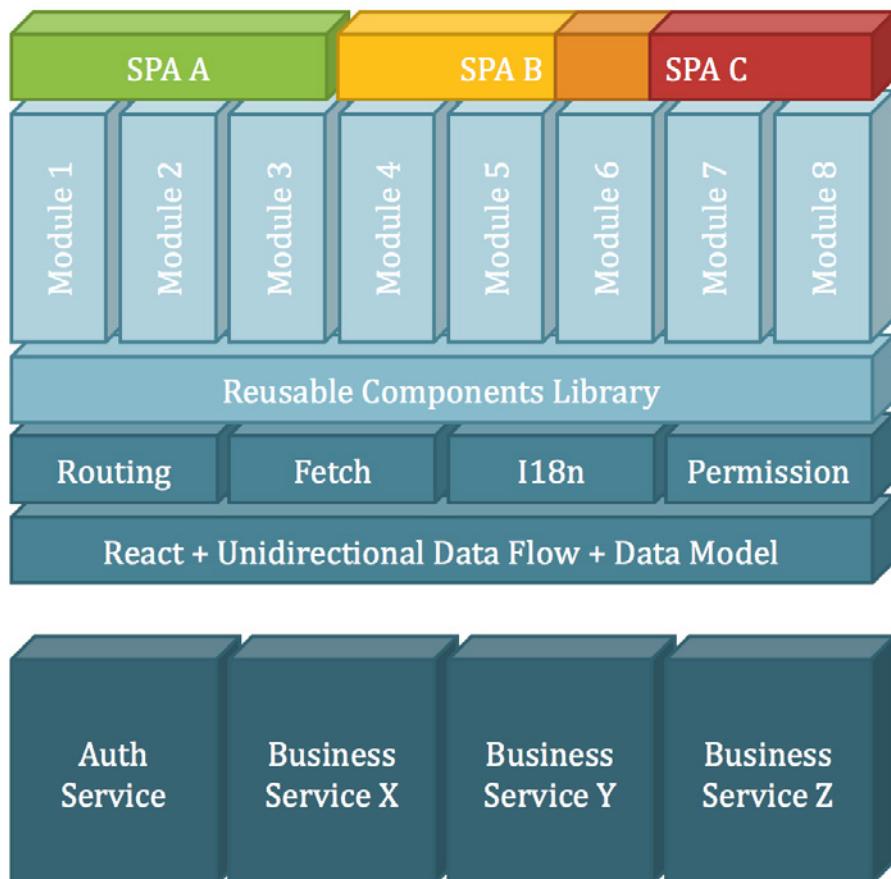
通过分层采样方法与 Re-weighting 方法，分类模型的 AUC 在不同场景中都有所提高。

SparkUI：前端单页应用演进与实践

在现代前端应用的工程实践中，单页应用 SPA (Single Page Application) 会为用户提供更极致的体验，加之更具灵活性的前后端分离的架构，已成为主流趋势。与之相对的，传统的单体 Web 应用 (Monolithic Web Application) 往往是基于多页面模式的，且前后端代码放在一起，虽然耦合性较强，页面给用户的体验也不够连续，但在产品研发的特定阶段仍具有较强的优势，早期的 ASP.NET、Java Spring MVC，以及 Ruby on Rails 都是单体应用的代表性框架。本文将以 FreeWheel 从单体应用改造为前后端分离的单页应用的实践为例，着重介绍前端演进期间所遇到的挑战和解决方案。

相较消费者，商业用户对前端应用的需求更具复杂性，且更强调质量。FreeWheel 深耕企业级的视频广告领域 10 年，其基于 Ruby on Rails 框架为广告主打造的 Web 管理应用已经历多轮迭代和演进，目前已达到 20 多个产品模块，1200+ 页面，代码量已达到 143.5 万行代码，其中包含 39

万行基于 jQuery 的传统 JS 代码。为保证其质量，其中包含了 20.2 万行单元测试代码，除此以外，还有独立的近 2 万个自动化测试脚本。在两年前，我们感受到了单体应用的局限性，并决定将其改造成为前后端分离的架构。



技术选型

FreeWheel 前端展现的业务多种多样，但其用户体验强调高效性和高一致性。为辅助业务研发团队进一步提升前端开发的效率和效果，我们在改造前期订立了组件化的目标，力求将统一的用户体验和复杂的内部交互逻辑封装进组件，通过自动化测试保证其质量，并最终在业务模块中广泛复用。

针对以上目标，我们选择 React 作为新前端核心技术，以 ES6 作为开发语言，利用 Webpack 和 Babel 进行编译打包。以 Mocha 全家桶加 Enzyme 作为单元测试框架保证组件质量。每个业务模块均开发、打包并发布为一个独立的 SPA，多个模块 SPA 之间，除了以统一的 SSO 服务保证用户认证外，并无更多的耦合，这一点保证了多个业务模块团队的工作不会互相制约。

在单向数据流框架选择上，我们基于 Facebook 的 Flux 推进了相当长的一段时间。在上线两个业务模块后，我们认识到 FreeWheel 的业务对前端数据流需求的复杂度远高于常见的 TodoMVC 样例，Flux 实现这些需求时会遇到较多困难。我们评估了当时的社区新秀 Redux，它能一定程度缓解我们遇到的问题但仍有局限性。我们最终决定以 Redux 和 ImmutableJS 为基础，开发一套新的单向数据流框架 Spark-modula。这点在下一节会有详细描述。

类似的还有前端路由。我们初期的选型是 React-router，随后根据项目需要开发了新的前端路由框架 Spark-router。

更详细的前端框架选型及新旧对比如下：

	原有前端架构	新前端架构
服务器	Rails + Unicorn	Nginx
开发语言	Ruby + ES5	ES6 + JSX
SSR服务器端渲染	Rails MVC + ERB/HAML	无
浏览器端JS	jQuery jQueryUI Bootstrap Handlebars Underscore	React.js SparkUI Redux Immutable.js React-router Lodash
请求数据	jQuery.ajax	Fetch API
JS组件	基于jQuery定制开发	基于React定制开发
CSS	SCSS	CSS Modules + PostCSS
路由	Rails Router	Spark-router替代react-router
I18n	Rails I18n	定制化的gettext方案
打包工具	Rails Assets Pipeline	Webpack + Babel
Lint	N/A (Rubocop for Ruby)	ESLint + customized rules
UT	Rspec	Mocha + Chai + Sinon + Enzyme
UT Coverage	N/A	Babel-istanbul

至于后端，我们的选型是以 Golang 开发的微服务。借此契机，团队

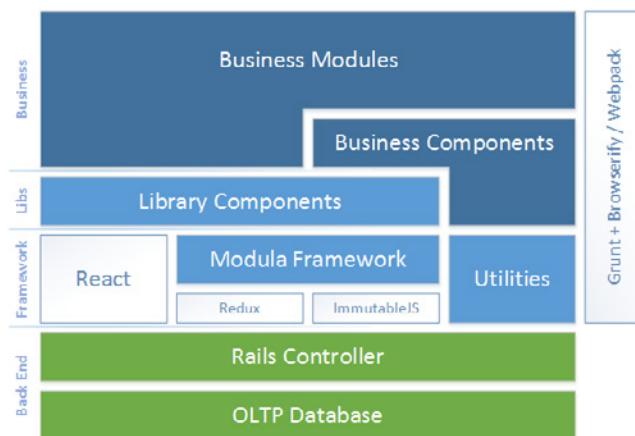
将原来内置于单体应用中的后端服务重新做了一次梳理，并逐步重构为微服务架构中的若干个微服务。前端在通过 SSO 验证后，以 JSON 格式与微服务交换数据。这些微服务除了满足前端使用，也会通过 Gateway 作为 API 暴露给我们的客户，更会为公司内部的其他微服务提供基础。后端架构不是本文重点，故不赘述，有兴趣的读者请参见 FreeWheel 发表的其他文章。

新轮子 SparkUI

为了推进 SPA 改造，我们成立了一个专门的前端小团队，与业务模块开发团队紧密合作，经历数十个迭代，开发并完善了一套基于 React 的前端框架，内部名称为“SparkUI”（这一名称与 Apache Spark 或 Java Spark 无关）。

SparkUI 是一套完整且灵活的前端开发解决方案。该方案基于 React，由 Modula 应用状态管理框架、一系列可重用的前端组件、以及构建 SPA 所需的各类支持库组成。该方案重视可重用性、灵活性、可测试性以及开发效率，解决了前端社区常见的一些针对商业前端应用开发的痛点，如复杂状态、Side Effect，组件拆分等，更在工程实践、文档化、本身代码质量等方面达到较高标准，为前后端分离架构下的商业前端应用开发提供了坚实的基础。目前 SparkUI 已成功应用在 FreeWheel 的前端项目中。

下图是 SparkUI 框架的简要架构。



其中上游的 React、Redux、ImmutableJS 等框架为 SparkUI 的直接依赖，下游的 Business Components 业务组件、Business Modules 业务模块则为基于 SparkUI 框架开发业务代码的产出；衔接上下游的，则是 SparkUI 的核心组成部分。

可重用组件 Library Components

SparkUI 截止至截稿日已积累了 40 个子 Package，其中很大一部分为可重用的 UI 组件，我们称之为 Library Components，例如 Spark-loading、Spark-calendar、Spark-raw-grid 等。凡是业务模块提出的对前端组件的需求，只要与业务并不直接相关的，我们都会设计并迭代开发相应的可重用组件。

我们在设计可重用组件时，遵循的一些要点包括：

1. 无状态组件（Stateless Component）优于状态化组件（Stateful Component）。例如下面的 <LinkGroup> 组件，用户调用该组件时需要将是否展开的状态传入 expanded 属性，并在 onClick 中传入处理函数以修改该状态。

```
<LinkGroup
  label="More"
  expanded={ model.get('linkGroupExpanded') }
  onClick={ model.sendLinkGroupToggle }
>
  {/* <Link>s */}
</LinkGroup>
```

我们在当初设计 API 的时候完全可以利用组件内部的 State 维护其展开状态，点击修改状态的逻辑也藏在组件内部，用户就无需传入这两个属性。这样确实能减少用户的代码量，但也大大限制了该组件的应用场景。试想，如果 LinkGroup 的子组件是在点击时才去服务器端获取的，用户该如何使用这一组件？

2. 组合组件 (Composing Components) 优于具有 DSL (Domain Specific Language) 属性的单一组件。在下面的例子中，`<Tooltip>` 被设计为一个独立的组件，封装了鼠标悬停则显示 tooltip 的交互逻辑，其 `Content` 属性除了支持简单的字符串，更是支持传入任意 `Element`；任何其他组件需要加入 tooltip 支持时，只要将其作为 `<Tooltip>` 的子组件即可：

```
<Tooltip content={ <p> I am a <strong>tooltip</strong></p> }>
  <div>My Content</div>
</Tooltip>
```

反观另一种设计(我们并没有这样做)，任一组件需要加入 tooltip 时，需要修改该组件以加入 tooltip 以及相关的一系列属性，这样并做不利于组件复用：

```
<Label tooltip="I am a tooltip" tooltipFontWeight="bold">My Content</Label>
```

以上只是简单的例子，其实我们在这一点上交过很高额的学费。我们最初版的 Grid 是这样的：

```
<Grid
  columns={[
    {
      name   : 'number',
      display: 'Number',
      size   : 'fixed-medium',
      align  : 'left',
      widget : false,
      tip    : {
        1: {
          direction: 'right',
          title    : 'This is level 1 tip',
          content  : () => { return 'This is level 1 tip'; }
        },
      }
  ]}
```

```
2: {
    direction: 'right',
    title     : 'This is level 2 tip----level 2',
    content   : () => { return 'This is level 2 tip----level 2'; }
}
},
{
    name      : 'month',
    display   : 'Month',
    size      : 'fixed-large',
    sortable: false,
    widget   : false
},
{
    name      : 'url',
    display   : 'URL',
    size      : 'fixed-medium',
    align     : 'center',
    sortable: false,
    renderer: {
        fallback: 'url'
    }
}
]
data={ this.data() }
/>
```

其中的 Column 属性我们称之为 DSL 属性。这样的 DSL 描述一个 Grid 对于用户而言还是比较友好的，但对于组件内部逻辑实现而言，其可维护性相当差，尤其是在项目初期，不断有用户提过来新的需求，比如“想改变特定单元格点击行为”，“特定行可以展开并加载子数据”等等，导致我们在维护这套 DSL 语法时会经常陷入前后矛盾，举步维艰的状况。

针对这样的困难，我们最终淘汰 DSL，开发了 Spark-raw-grid 包，将 Grid 拆分为从大到小十几个组件，由用户根据需要将他们组装在一起。限于篇幅，不在这里介绍细节。

3. 高阶组件 (HOC, Higher-Order Components) 优于混合属性 (Mixins)。在设计 Spark-modula-form 的时候，我们使用了 Mixin 属性：

```
<Checkbox label="Subscribe to news" {...subscribe.getCheckboxProps()} />
```

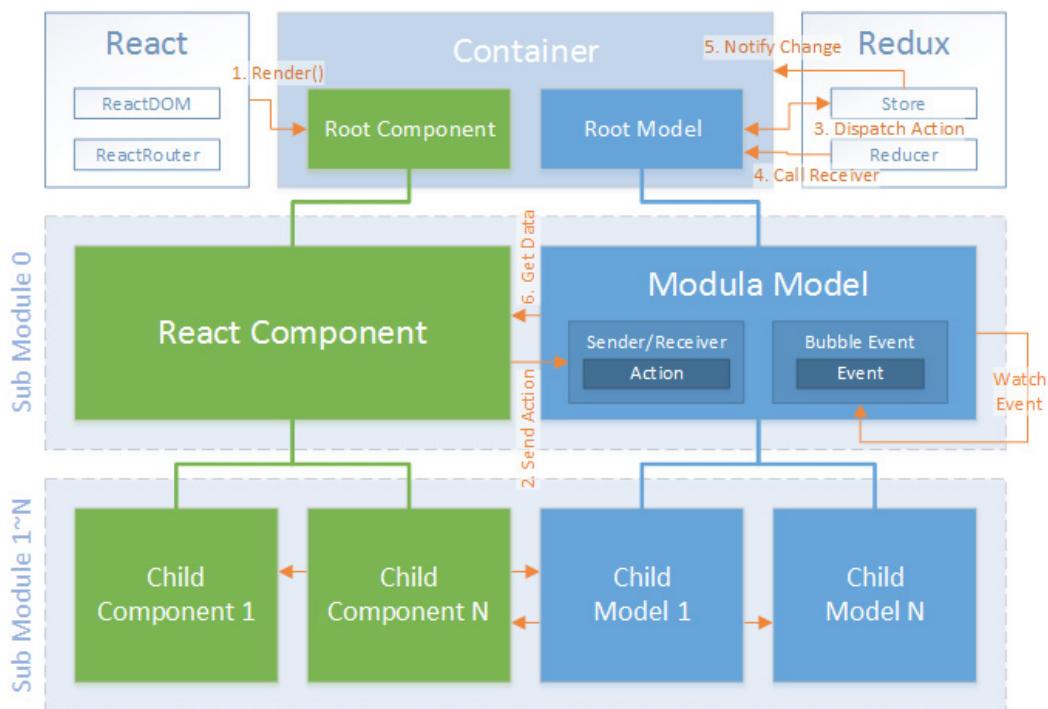
用户在加入表单输入组件时，需要显式的加入预制 Mixin 属性以享受表单接口的便利。然而在用户需要定制类似 onChange 这样属性时，Mixin 属性会产生冲突。这时 HOC 方式会提供更高的灵活性。我们在 SparkUI 中广泛使用了 recompose.pure() 这样的 HOC，也开发了一些自有的 HOC。限于篇幅，省略细节。

应用状态管理框架 Spark-modula

上一节提到 Flux 所提供的单向数据流不能完全满足我们的业务需求。我们在对比了 Flux 和 Redux 后，决定自主开发一只新轮子。当时面对的挑战包括但不限于：

- 在 Ruby on Rails 应用中，开发团队曾设计并开发了大量的 Model，这不仅是因为要遵守 RoR 的 MVC 实践，更是因为业务的复杂程度客观要求有完整的建模，并基于模型推进前端的开发。我们的新轮子需要以类似的方式来消化业务的复杂性；
- 在业务的前端需求中，常常有一个页面内包含 2 个甚至多个 Grid，这些 Grid 之间会互相影响。比如一个典型场景：“Grid A 在自动加载后，如果只包含一条记录，则自动选中这条记录，并按该记录 ID 读取 Grid B”。这样的交互在 React 社区被称为副作用即 Side Effects。我们的新轮子需要用相对简单的方式支持 Side Effects 的处理。

这只应用状态管理的新轮子我们起名为 **Modula**，并入 Spark-modula 包。Modula 框架基于 Redux 但并不限于 Redux，与部分 Redux 生态（如 Redux-devtools）兼容，且已完整封装并隐藏了底层的 Redux。下图简要介绍了 Modula 与 React、Redux 的关系：



Modula 应用状态管理框架

例如，在 Redux 里，应用状态是完全平展开的结构且不存在任何的层级关系，因为缺乏一个对象化的组织，所以要在状态众多的情况下，在 Redux 的 Store 上找到某个状态就只能依靠纯记忆。而 Modula 引入了对象树（Model Tree）后，所有的状态都可以被对象化，即通过预先定义好的结构来组织状态。尽管是比较复杂的组件，在页面上的展示可能只是一个表单或 Table。

如果给这个 Table 设定一个较为复杂的状态——加一个搜索条，搜索条本身有简单搜索和复杂搜索的区分，上面还有复杂的工具栏、动作条，其本身或许还需要支持翻页等。如此多的状态之下，用 Redux 的方式可

能会有好几百个状态在一个 Store 里，于是管理起来就会非常困难；但 Modula 就可以组织得更好，下面是 Modula 主要的设计理念：

1. Application State = Initial State + Deltas，其中 Delta 是由 Actions 触发的（借鉴 Flux, Elm）；
2. Application State 可以由一棵 Model Tree 来描述，这棵树的每个节点都是一个可以描述有效业务实体的 Model（借鉴 Redux, Elm）；
3. 由一个给定的 Application State 到另一个 State 的 Transition 可以由 Model Tree 提供的 Reactions 所描述，一次成功的 Action 到 Reaction 的匹配会将 Model Tree 演变为下一个状态（原创）；
4. Side Effect 是上述 State Transitions 的结果，它包含了一个更新的 Model 实例，以及零至多个 Callback Functions（借鉴 Elm）。

对于 Modula 中 Side Effect 问题的处理，Modula 模块中的 Receiver 可以返回 Side Effect，一个 Side Effect 可以是 Sender 或 Bubble Event 的引用，也可以是一段匿名函数（箭头函数）；List-A 读取完成时会根据 List-A 中包含的 ID，自动触发读取 List-B。

经过快速迭代，Modula 框架已正式替代早期的 Flux，应用于业务模块开发。Modula 包括 Model 模型、Constants 常量、Container 容器、Test Utility 测试工具四个组成部分，其中 Model 包含 Props/Hierarchy、Context、Sender/Receiver、Delegates、Bubble Event、Lifecycle Methods、Services、Local Props 等概念 / API。以下是一个典型的 Model 例子：

```

const EmployeeModel = createModel({
  propTypes: {
    id: PropTypes.string,
    name: PropTypes.string,
    age: PropTypes.number,
    todoList: ImmutablePropTypes.listOf(PropTypes.instanceOf(TodoModel))
  },
  defaults: {
    id: null,
    name: 'Employee',
    age: 0,
    todoList: () => new List()
  },
  sendAddTodo() {
    this.dispatch({ type: ActionTypes.ADD_TODO });
  },
  recvAddTodo() {
    return {
      type: ActionTypes.ADD_TODO,
      update(model, action) {
        const newModel = model.set('todoList', todoList => (
          todoList.add(new TodoModel())
        ));
        return [ newModel ];
      }
    };
  }
});

```

可以看出上半部分相当于 Model 的 Schema, Props/Hierarchy/Context 基于 Immutable 数据结构实现了数据模型; 而下半部分相当于 Model 的行为, Sender/Receiver + Modula Container 实现了单向数据流。

前端路由框架 Spark-router

此外, 为了能够支持构建典型的 SPA, 我们基于 Modula 开发了一个 Spark-Router 组件。相比于 React-Router (其状态并不存储在 Redux 的 Store 上), Spark-Router 中路由的状态管理能够与应用中其他部分的状态管理采用同样的机制。

此前, 应用状态都分散在 React-Router 的状态与 Modula Model (底层对应 Redux Store) 里, 两者经常遇到同步问题, 我们的解决方案是将路由相关的状态也合并进 Modula 中。Spark-Router 基于 Model 配置路由,

Component 可根据 Model 切换相应界面，这样就不必再在路由的状态和应用中其他部分的状态之间添加同步设计，也让程序变得更简单。

SparkUI 开源计划

到截稿日为止，SparkUI 在 FreeWheel 的生产环境里使用已经超过一年了。我们无论是在开发 SparkUI 还是开发其他前端基础设施的过程中，都利用了大量开源社区的成果，我们也希望我们的工作和成果能回馈到开源社区，为更多的开发者提供新的选择。

我们开发 SparkUI 的初衷，是为了应对 FreeWheel 前端应用较高的复杂性。FreeWheel 核心应用之一是广告资源的管理系统，这个系统所针对的客户非常专业，对应的、有着非常复杂的工作流是要通过 UI 来实现的。这导致该应用在前端的状态很多、很复杂。SparkUI 框架的特点，就是擅长于用来构建具有复杂前端状态的应用。这也是核心商业系统普遍具有的一个特点，我们认为该框架在类似应用中会有较高的使用价值。

SparkUI 目前还只是在 FreeWheel 公司内部使用，而其开源事宜已提上日程。FreeWheel 和其母公司 Comcast 都对此非常鼓励，目前已经开始对开源 SparkUI 走相关法务流程。FreeWheel 首席架构师张晗表示：

“SparkUI 可能并不会全部开源，组件库这类属于产品特定需求的部分会被拿掉，像 Modula 这样的通用模型部分会属于开源的范畴。如果你的应用需要复杂的前端功能，特别是需要对具有相互关系的状态进行较多维护时，就可以考虑使用我们的 Modula。”

前后端整合

从单体应用改造成前后端分离的架构后，理想状态下，前端可以分别独立开发、测试、部署，然而若想实现整体业务，则需要将前后两端整合。本节将介绍我们开展改造工作以来，在前后端整合领域积累的部分最佳实践。

RESTful 接口

后端接口均按照社区 RESTful 接口标准定义：

- 语义化 URL，活用 GET/POST/PUT/DELETE 四种 HTTP 方法；
- 支持 JSON 与 XML 两种数据呈现格式，默认情况下，HTTP 请求和响应均使用 JSON，加入 XML 参数，请求和响应改为使用 XML；
- 优先使用 HTTP 状态码（Status Code）表现后端成功状态或各类常见错误，如 HTTP 200(OK)、401(Unauthorized)、422(Unprocessable Entity) 等；
- 统一业务错误码和错误消息；
- 以 ISO 8061 标准输入输出日期时间，如：2015-09-08T01:55:28Z。

在前端我们基于浏览器 Fetch 接口，封装了 Spark-fetch 包，提供如下功能：

- 浏览器 Fetch 的所有功能；
- JSON 序列化、反序列化；
- 为 HTTP 错误统一显示对话框，其中 401 状态会跳转至登录页面；
- 根据用户需要缓存特定资源；
- 防止 Cross-Site Request Forgery (CSRF)。

我们为前端开发了一套简单的 Discover 服务发现，以 Key-value 方式描述前端中会用到的 RESTful 服务，Spark-fetch 包在发起 HTTP 请求时只要传入 Key 和相关参数即可。目前主要用来防止前端代码里 Hard-code 服务 URL，之后会与整个公司级别的服务发现整合起来。

除此之外，我们还在后端开发了一套 API Gateway，提供认证（Authentication）、限流（Throttling）、跨域等公共功能。上述 RESTful 接口本身无须处理认证等逻辑。在部署后端服务后，只有 API

Gateway 开放给外网访问，其他 RESTful 接口均限于机房内网访问，经由 API Gateway 的反向代理提供给外网。即前端在调用这些接口时，必须经过 API Gateway 调用。

认证授权

文章一开始提到的单体 Web 应用其实在 FreeWheel 有多套，分别对应于多个业务线或产品线。这些单体应用开发的阶段有先有后，架构和实现的设计也存在着差别，其中很重要的一点就是认证方式的差别，为了满足多个应用联合登录的需求，尤其是向后兼容 SPA 的联合登录，我们在后端以 Golang 开发了新的 SSO 服务。SPA 在登录页面调用 SSO 接口，登录成功则获取 Token 并存入 Cookie，这样后续的接口请求就会将 Cookie 传入 API Gateway 以获取认证信息。

至于授权（Authorization），我们在现有的 Ruby on Rails 应用中大部分是基于 CanCan 框架实现的，改造为前后端分离架构后，我们将与导航、功能入口相关的授权信息从后端完整传回前端，用前端代码判断特定导航或组件是否显示、是否禁用。当然，RESTful 接口中仍有完整的授权判断逻辑。如果有恶意用户通过 Hack 的方式修改了前端授权信息访问了本不能访问的界面，他依旧无法获得列表数据、也无法提交数据修改。

后端 Docker 容器化

在业务模块开发过程中，开发人员需要在开发前端代码的同时能访问到后端接口及测试数据。如果是单体应用的开发，开发人员只要配置一套开发环境即可达到这个目标，但在前后端分离后，前端开发人员除了配置前端开发环境，还要配置后端。后端代码有更新时，需要及时检出代码并顺利编译，数据库有更新时也需要执行相应的 SQL 脚本。这些日常工作成为前端开发人员的痛点。

后端 Docker 容器化有效解决了这一痛点。我们目前的 CI (Continuous

Integration) Pipeline 会在后端代码检入远程 Git 后触发编译，编译成功后会创建一个包含该编译版本的 Docker Image 并上传至公司内部的 Docker Image 仓库，类似的还有数据库，以及其他中间件的 Image。前端开发人员不再需要搭建后端开发环境，只需在开发机上安装 Docker（如 Docker for Mac），在前端工程内会维护一个 docker-compose.xml，声明了前端工程所需要的后端 Docker Image，每次该文件更新后，前端开发人员只需要运行 docker-compose up -d 即可启动一系列 Docker Container，在本机运行完整的后端服务，这里甚至包含了适用于开发的部分测试数据。

整合测试

前后端的分离和整合对质量保证提出了新的要求。我们在前端编写 Fetch 逻辑时，会以 Mock 方式编写对应的单元测试。后端每个接口也有响应的单元测试。而这两端分别的单元测试还不足以保证软件质量，理论上讲，纵使两者单元测试覆盖率均达到 100%，也不能保证覆盖所有用例。作为质量保证的关键环节，在两端的单元测试都通过后，我们的 CI 会执行端到端的自动化测试。这些自动化测试模仿了用户的使用场景，完整的覆盖了前端、后端、数据库乃至其他中间件。

渐进改造

SparkUI 的产生为 SPA 改造提供了坚实的基础。如果按最理想的方式推进，只要业务开发团队基于 SparkUI 对现有的 Ruby on Rails 的单体应用的前端部分、基于 Golang 微服务方式对其后端部分进行重构改写、践行前后端整合的最佳实践，即可达成前后端分离的目标。而文章开头曾提到，现存的 Rails 应用体积大、复杂度高，纵使有着业务开发团队的全力支持，我们也很难在一个较短时间内彻底完成 SPA 改造。更何况市场千变万化，在业务部门服务老客户、获取新客户过程中，产品经理们也会不断

地提出新的产品需求给我们的开发团队，技术演进和业务推进两者需要取得一个平衡。我们为达成这一平衡，所提出的方案是：渐进改造。

混合工程结构

我们的业务模块在 Ruby on Rails 工程中是以 Module 方式存在的，除了公共的 MVC 和资源放在统一的 Module 里，每个业务 Module 都有自己的 MVC 和资源（这里的资源特指 JavaScript 和 CSS）。我们以业务 Module 作为改造的单元。

由于资源等限制，前后端分离改造在前端、后端的推进节奏并不一致。比较多的情况是 Module 前端改造先行，后端依旧沿用 Rails 原有的 Controller（也有部分适配工作）。在这种情况下，Module 经 SparkUI 改写的 SPA 前端独立于 Rails 工程之外进行打包部署所带来的好处并不明显，故将这部分 SPA 前端代码的源码依旧放在 Rails 工程 Module 目录下，通过 Webpack 打包的 bundle JS / CSS 也按照 Module 对资源文件的约定（Convention）放在 modules/my_module/app/assets/javascripts/my_module/compiled 目录下，并藉由 Rails Asset Pipeline 打包进 Rails 工程发布包进行统一部署。

对于上述 bundle JS/CSS，我们仍使用 Rails 页面模版作为入口，以期减少对 Rails 工程的影响：

```
<%= javascript_include_tag "my_module/compiled/my_module" %>
<%- @js_module_alias = "my_module" %>
<div id="spa"></div>
<script>
(function() {
    var React = require('react');
    var ReactDOM = require('react-dom');
    var AppContainer = require('<%= @js_module_alias %>').AppContainer;
```

```

ReactDOM.render(
  React.createElement(AppContainer),
  document.getElementById('spa')
);

})();
</script>

```

至于路由，既然我们已经在 SPA 中实现前端路由，那在 Rails 端的后端（页面）路由就可以委托给前端：

```

scope 'spa' do
  get '/', :to => 'spa#index', :as => 'spa'
  get '*pages', :to => 'spa#index'
end

```

经由以上方案，我们在尽量短的周期改写了更多的业务模块，对运维的影响也非常小。对于这些业务模块，我们预期在其改写后端微服务时将前端代码从 Rails 里彻底分离出来，完成该模块的前后端分离。

在上述 Ruby on Rails 项目之外，FreeWheel 也启动了若干个新项目。这些项目一步到位，直接按照前后端分离架构设计开发，其前端均为完全基于 SparkUI 的 SPA。

静态资源服务器

我们也基于 Nginx 开发了一套轻量的静态资源服务器，前端利用 Webpack 编译打包成 Tar 包并独立上线。

前端资源，包括 JS、CSS、图片、字体等文件，在打包时会在文件名上追加一段 SHA1 值作为文件指纹，文件指纹相同意味着文件内容没有变化，这就保证了多版本静态资源文件可以共存，由入口页面决定使用哪个版本。基于这样的设计，前端代码在 CI (Continuous Integration, 持续集成) 的基础上，最终进一步实现了 CD (Continuous Deployment, 持续部署)。

文件名带有指纹的文件，在 Nginx 上可以设置长效的客户端缓存，对

于大文件，我们也采用了预压缩，通过 Gzip 格式减小文件传输的体积。更进一步，我们为静态资源配置了 CDN，并将静态资源服务器指定为 CDN 的源。这些措施都有效提升了 SPA 页面在用户浏览器端的加载速度。

SparkUI 独立工程

在小步快跑阶段，我们将 SparkUI 源码直接放在 Rails 公共 Module 中，令我们可以快速验证可重用组件的设计是否满足业务需要。然而这样的结构会带来几方面问题：

- 版本管理。任何对 Spark 的迭代都会直接影响到业务模块；
- 开发效率。SparkUI 是纯 JS 库，Rails 工程开发环境给 SparkUI 开发带来一定负担；
- 源码权限。任何业务模块开发人员均可修改 SparkUI 代码，带来潜在代码冲突；
- 跨工程复用。任何 Rails 工程之外的工程在利用 SparkUI 时都会比较繁琐。

我们在 SparkUI 推出 1.0 版本时，将其源码从 Rails 工程中摘出，移入一个新的纯前端工程。SparkUI 在这个新工程中，仍由 Babel 和 Webpack 打包，但会作为 Library 发布到公司 Nexus 上私有 NPM Repository 里。Rails 工程或其他纯前端工程在其 package.json 和 .npmrc 配置中声明对特定版本 SparkUI 的依赖，执行 npm install 后则可以在前端代码中使用 SparkUI。

这一改变大大解放了 SparkUI 和业务模块两方的生产力：

- 独立的代码库可以隐藏部分 SparkUI 的内部 API 或工具代码，防止业务模块中滥用；
- 不同的发版节奏令 SparkUI 可以追逐更高的代码质量，目前其源代

码已超 10 万行，单元测试覆盖率高达 99.81%；

- 业务模块代码可以更有计划地升级 SparkUI 版本，在此之前无须反复回归测试。

新老 JS 代码混用

对于 Rails 工程的部分功能模块，其前端实现有很大一部分是基于 jQuery 开发的 JS。虽然这些代码并不是基于 React 或 SparkUI 开发的，但它们也可以直接在 SPA 中独立使用。我们在统一的粒度下，创建了一层对 React 友好的适配器 Spark-adapter，对原有 jQuery JS 接口进行了封装和隔离。业务模块开发人员可以自行决定对于这一部分 JS 代码是基于 SparkUI 重写还是放在 Adapter 中以继续沿用。

质量保证

作为商业应用，其软件质量是绝不能妥协的。前后端分离和 SPA 改造不能成为降低软件质量的理由。我们保证质量的核心是测试：

- SparkUI 组件库本身要具有最高标准的单元测试覆盖率；
- 业务模块改写为新前端时，也要基于 SparkUI 提供的基础设施编写单元测试；
- 对于 Rails 工程原有的自动化测试脚本，在业务模块改造为基于 SparkUI 的新前端时，也要同时更新；
- 将测试加入 CI (Continuous Integration) Pipeline，一有 Merge Request 提交就执行测试，测试成功才允许 Merge；
- 各组 Lead 在 Merge Request 上做代码审查时严格把关。

另外一个有效实践是为新上线新前端的模块提供回滚机制。因为在这一步骤，Rails 工程里特定功能模块的新老前端代码可以同时存在，只需在功能入口处设置一个开关，就可以在线上执行 SPA 遇到严重问题时随时

切换回老前端。

专职前端团队

我们知道，任何一个技术产品的架构都会与其开发团队的组织架构相互影响，上述演进过程中涉及的一系列工作也不例外。尤其是 SparkUI 这样创新型的项目，需要有专职团队来支持。以 SparkUI 为契机，FreeWheel 在 2015 年底成立了一个专职前端团队，这个团队并不直接实现业务功能，而是专注于前端框架、基础设施、最佳实践等的搭建、演进和推广。最初在开发早期版本时，团队成员有两人，目前已发展至 6 人。

现阶段，团队的工作主要集中在 SparkUI 上，其中两到三位同学负责框架的核心开发和升级，比如 Modula 这样的核心组件；其余同学主要专注在扩展可重用组件库上。目前 FreeWheel 大多数的业务应用开发者，都是基于这个组件库去开发，而他们会不断地提出新的组件需求，例如新业务的开发需要新组件支持，或是原有组件需要追加新功能。

同时也有一些公共的前端模块或是基础设施，会由该团队开发维护。比如前文提到的静态资源服务器、SPA 页面通用的页头（Header）和页尾（Footer）等等。

他们在开发工作之余，也要撰写文档，为业务模块开发团队提供内部培训、日常答疑。帮助整个 FreeWheel 的研发团队更好的掌握并应用 React 和 SparkUI，进而推进 SPA 的演进。

总结

前后端分离的单页应用架构是诸多前端应用系统的必经之路，而现实情况往往需要顾及诸多历史架构。本文以单体应用为背景，设计开发可重用组件库为手段，在保证效率与质量的基础上，逐步演进为 SPA。希望文章对同样面对这一情况的读者有所帮助。

正如上文提到的，作为 SPA 基础的 SparkUI 框架，其中与 FreeWheel

业务并不直接相关的纯技术部分，比如 Spark-modula、Spark-router 等包，我们已计划将其逐步开源。希望届时能与更多的前端技术专家和群体深入探讨、共同进步，并最终对前端社区有所贡献。

FreeWheel 如何支持超级赛事的直播广告

FreeWheel 助力多家全球顶级媒体和娱乐公司开展视频广告业务，其中全美国 90% 以上的主流电视媒体和运营商均在使用 FreeWheel 的服务。在过去的几年里，FreeWheel 支持过许多超级赛事，包括 2016 年里约奥运会、2014 年世界杯、2012 年伦敦奥运会等。在这些超级赛事中，FreeWheel 为客户提供了大量的直播广告支持。在此背景下，这些超级赛事直播广告背后的技术架构和技术设计显得尤为重要。

直播广告的业务背景和技术挑战

所有的架构设计和技术设计都源于业务需求。因此，FreeWheel 的业务背景介绍以及直播广告的业务特点分析成为了架构设计和技术设计的前提。

业务背景

FreeWheel 的客户规模很大，多数都是市值几百亿甚至上千亿美元的公司。对于这些客户来说，广告收入是其收入的重要来源。换而言之，

FreeWheel 为重量级的客户提供其核心业务的技术支撑。

以 NBCU 为例，NBCU 是全美三大商业广播电视台之一，在 2016 年里约奥运会期间，NBCU 的广告收入超过了 12 亿美元，其中 FreeWheel 承担了 NBCU 全部数字视频广告支持业务。FreeWheel 的广告服务器在里约奥运会期间为 NBCU 投放了超过 15 亿次广告，总价值超过 1 亿美元，峰值时广告量接近百万次广告 / 秒。

为类似 NBCU 的大客户提供超级赛事的直播广告支持，无疑给 FreeWheel 广告服务器的技术设计带来了较大的挑战，而这些挑战主要源自于以下几个方面：

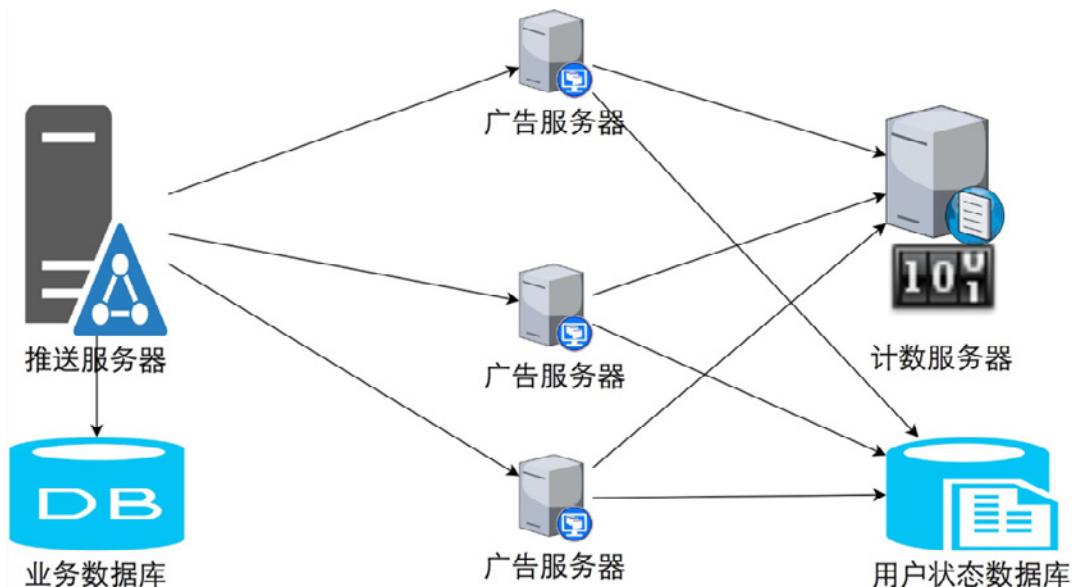
技术上的挑战：高并发，实时响应，高可用性

- **高并发：**在直播比赛过程中，所有的用户几乎同时发出广告请求，因此广告服务所面临的瞬间并发量非常高，峰值时广告服务器处理广告请求的 QPS 高达数十万/秒。
- **实时响应：**广告经常在比赛暂停时插入，但是比赛暂停的时机并不可预测。因此，从获知可以插入广告到广告开始播放，可能只有短短数秒时间。在此如此短的时间内，如果广告服务器不能及时做出广告决策，就会影响播放器的正常播放。实际上，FreeWheel 承诺给客户的广告响应时间是 300ms，而在这 300ms 中，还包括了一些高延时的外部系统交互，真正留给广告服务器做决策的时间只有短短几十毫秒。
- **高可用性：**在比赛期间，如果广告服务在决策或者播放过程中出现任何错误，将给客户带来不可估量的损失，直接影响到客户的广告收入。因此，这对广告服务器的高可用性提出了高标准的要求。

为应对以上这些挑战，FreeWheel 在广告服务器的一些核心架构和技

术设计上具有独到的创新之道和演进经验。

广告服务器核心架构



广告服务器核心架构图

FreeWheel 的广告服务器核心架构主要由广告服务器、推送服务器、计数服务器、业务数据库和用户状态数据库组成。

广告服务器：广告服务器负责广告决策，主要包括以下过程：

- 根据用户的IP、Cookie、ID、所观看的视频信息以及网站等信息，选出候选广告集合。
- 根据一系列算法计算广告权值，根据权值大小依次尝试填入广告位。
- 在填入广告过程中，检测广告各种设置，避免广告投放违反广告设置，比如广告频次控制（Frequency Cap）、广告互斥控制等。
- 将广告决策结果发送给播放器。
- 在广告被播放时，更新广告投放计数。

推送服务器：推送服务器负责将数据库中的各种信息，包括视频信息、

网站信息、广告数据、广告相关配置导出到镜像文件，然后将镜像文件推送到各个广告服务器。推送服务器是广告服务器系统的核心服务器之一，后面会进一步详细介绍。

计数服务器：计数服务器用来记录各种广告的具体投放数据，这些数据将被用于做广告投放预算控制。

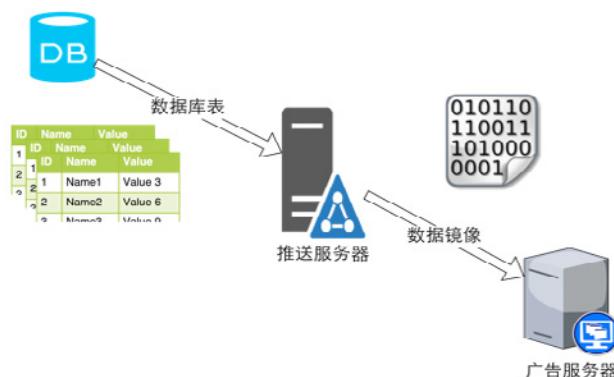
业务数据库：用于存放所有的广告相关的业务数据，包括视频信息、网站信息、广告数据、广告相关配置等。由于要向不同的客户提供各种定制化服务，而这些定制化服务需要各种不同的数据来支撑，所以FreeWheel的业务数据库有超过1,000个不同的数据库表。

用户状态数据库：对于不支持Cookie的客户端集成，原本应该保存在Cookie里面的状态数据将会被保存到用户状态数据库。

推送服务器

推送服务器是广告服务器核心架构的最重要组成之一。推送服务器是业务数据库和广告服务器的桥梁，主要用于加速广告服务器对业务数据库的访问。

推送服务器的主要功能是将存储在数据库中的各种信息，导入到数据镜像中，然后将数据镜像推送到广告服务器；广告服务器将数据镜像加载到内存中，通过访问所加载的内存数据，就可以访问到原来存放在数据库中的数据。（如下图所示）



一般在高并发的系统中，通常会使用 Redis 或者 Memcached 等内存数据库来加速对业务数据库的访问速度，跟 Redis 或 Memcached 相比，推送服务器模式到底有哪些优势呢？

为什么需要推送服务器

首先，FreeWheel 有许多重量级的客户，这些客户会有不同的定制化业务需求，导致业务数据库非常复杂——包含超过 1,000 个数据库表。每个广告请求可能需要访问其中数百个数据库表，有些数据库表还可能需要访问多条数据。

其次，对于一个广告请求来说，到底要访问哪些数据，事先是不明确的，只有在广告决策过程中，才能逐步明确后续的计算需要哪些其他数据。因此，很难设计一种简单的并行访问数据库的方法。

如果要顺序访问大量的数据库表，即使采用内存数据库做加速缓存，所带来的访问延迟也会非常显著，会严重影响广告服务器的响应时间；此外，如果大量顺序地访问内存数据库，内存数据库在性能上的轻微抖动，会造成广告处理延时较大地偏离正常值。

采用推送服务器模式，因为所有的数据都被加载到广告服务器内存中，广告服务器通过访问内存就可以访问到数据库中的数据，相当于用进程内的内存访问，取代了外部的内存数据库访问，因此极大降低了广告服务器的响应时间。实际上，FreeWheel 的广告响应时间大部分在 10–20ms 左右，如果只是通过内存数据库做加速缓存，是很难达到的。

除了加速响应，推送服务器带来的另外一个好处是稳定性。由于所有的数据都存放在内存中，在广告服务过程中就不需要大量的外部 I/O 操作。众所周知，外部的 I/O 操作和纯内存操作相比，稳定性不在一个级别，尤其是在超高并发的业务场景下。因此，推送服务器也极大提高了广告服务器在处理高并发请求时的稳定性。

控制数据镜像大小

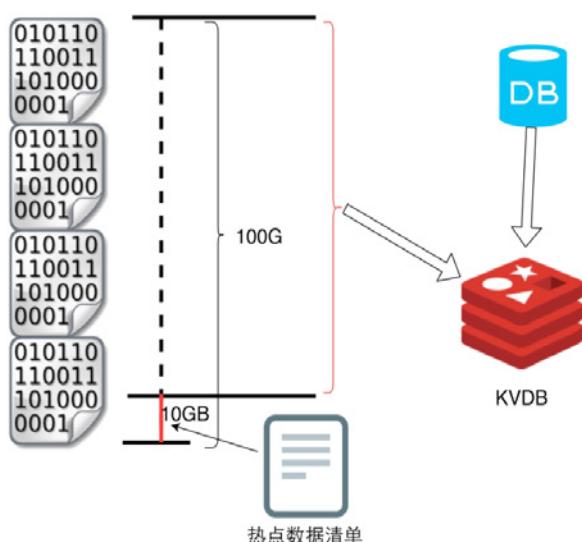
推送服务器虽然带来了性能上的提升，但其自身也有局限。由于所有的数据都加载在内存中，随着业务的增长，所需要的内存量也会越来越高。而在实际情况中，服务器的物理内存数量总是有限制的，所以需要严格控制数据镜像的大小。

FreeWheel 通过热点数据清单模式来解决数据镜像大小的问题，热点数据清单基于以下事实：

- 从数据来看，虽然有超过 1,000 张表，但是大部分表都非常小，对于这些小的表格，可以完全加载到数据镜像中。
- 只有极少数表的数据量比较大，这些大表主要是一些视频信息相关的内容。在 FreeWheel 数据库中，有超过 7,000 万的视频信息数据。
- 对于这些大表的数据，虽然数据非常多，但是也只有少数是被高频访问的。

如果把热点数据放入热点数据清单，之后推送服务器只加载热点数据清单中的数据，数据镜像大小将得到有效的控制。实际上，FreeWheel 虽然有 7,000 万视频数据，但是热点清单不超过 100 万，而这些热点数据覆盖了 95% 以上的广告请求。因此，通过热点数据清单模式，在没有本质影响服务器效率的情况下，有效压缩了内存数据镜像的大小。

对于没有在热点数据清单上的数据，依然采用通过内存数据库作为加速缓存的方式来访问数据库。



数据实时更新

推送服务器所带来的另外一个问题是数据的实时更新。由于推送服务器周期性地重新加载数据，在广告服务的过程中有些数据需要被实时更新，FreeWheel 通过以下方式来完成：

1. Purge 方式：需要更新的数据不放入热点数据清单。当用户修改数据库时，也一并修改内存数据库的缓存。广告服务器需要访问数据时，就可以直接访问到更新后的数据。
2. 按需访问：在广告请求里，直接指明哪些数据需要实时更新。广告服务器解析到这些信息时，直接读取数据库来获取内容。

计数服务器

计数服务器用于记录每个广告的投放情况。每一个广告服务涉及到数百乃至上千个备选广告，每个广告都需要预算控制。因此，广告服务器对计数服务器的访问频次非常高。在峰值情况下，单个广告服务器需要访问数十万数量级 / 秒的计数服务。

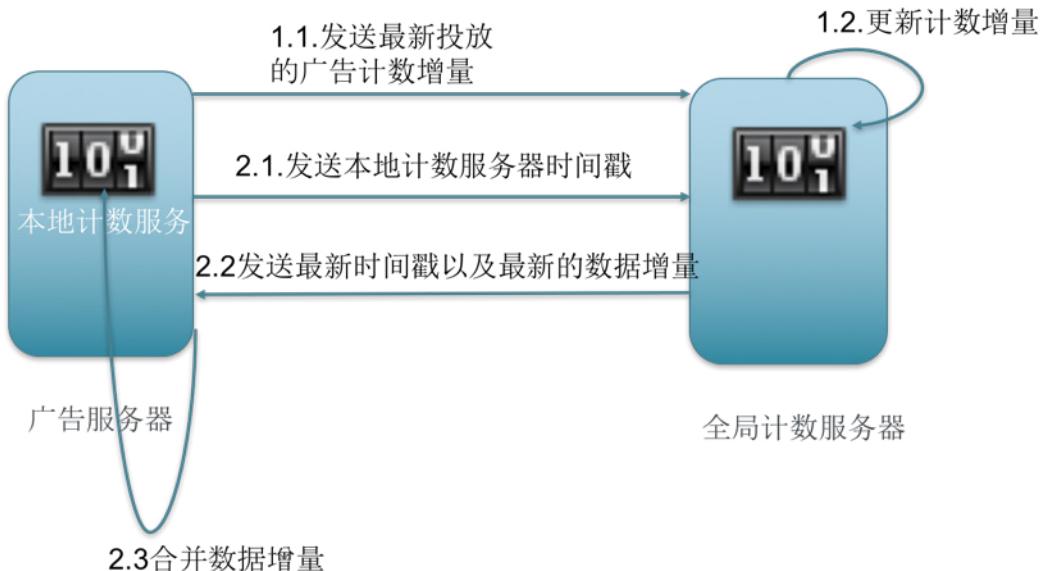
如何设计计数服务器，才能满足如此高的并发访问请求？

1. 首先，FreeWheel 有一个全局计数服务器，维护所有广告服务器的广告计数情况。
2. 其次，为了降低广告服务器对全局计数服务器的依赖，广告服务器内部设计了一个本地计数服务，本地计数服务是全局计数服务的一个镜像。
3. 广告服务器的所有投放决策，都通过访问本地计数服务完成。由于只需要访问本地服务，因此可以充分保障速度，以及高并发情况下的实时响应。

同时，由于广告决策时没有外部依赖，因此能保障服务器在高并发时

的高可用性。

本地计数服务器同步



为了保证本地计数服务器的准确性，需要高效地将全局服务器的数据同步到本地计数服务器，同步过程如上图所示：

- 全局计数服务器计数更新

- 广告服务器定期将本地计数服务的修改增量发送给全局计数服务器。
- 全局计数服务器收到修改增量后，合并到自己的计数服务里，从而得到最新的计数情况。

- 本地计数服务器计数同步

- 广告服务器定期发送自己的时间戳给全局计数服务器。
- 全局计数服务器计算所收到的时间戳和最新时间戳之间的计数增量，然后将计数增量发送给广告服务器。
- 广告服务器合并所收到的计数增量，从而达到和全局计数服务器数据同步。

容力：如何打造更高质效的技术团队

编者按

技术团队的管理包罗万象。但归结到底，无非在于二字：道、术。

听过很多人讲二者间的关联，却大多未全然说透。道与术的关系本质上就是指导思想与具体手段的关系，是心法与招式的关系。从技术角度来看，管理中的“道理”、“心法”或可界定为技术定位、技术与业务的融合、企业文化、技术与哲学等；“方法”和“招式”，则就如技术团队组织架构、产品开发流程、制度规范的建立、架构设计等。

怎样将“心法”与“招式”融合在一套特殊的理念和决策之下，或是困惑多数技术管理者的一道难关。带着对如何打造顶尖技术团队、如何在技术、文化以及技术人成长间搭建更优桥梁、如何培养团队荣誉感及价值感等问题，InfoQ 专访了 FreeWheel 高级副总裁容力及其所领导的技术团队管理者（技术副总裁党政法、技术副总裁王强），了解在这样一支跨国团队协同合作的环境背景下，作为 300+ 人团队的总负责人，他是如何塑

造高效敏捷的行动力、团队方向感、追逐感、凝聚力，以使庞大（甚至是跨国界、跨时区）的队伍能拥有更好的创新力和生命力。

曾在知乎上看过一个提问：“互联网公司，如何管理一个 8-10 人的技术团队？”下列的跟评有着林林总总各式不同的看法，其中一大致列了三条最简单也最复杂的见解，即了解人、了解事和了解业务。所以，我们和 FreeWheel 技术团队的交谈也从这几点开始。

万丈高楼平地起——「合理扩张」与「Hire the Best」

公司在高速发展的時候非常需要在不同的阶段引入适时的管理。从刚刚成立发展到数百人的规模，这是每个公司发展需要经历的一个门槛。有些会出现企业疯狂式的增长，这是欠缺对现实的考慮。我（FreeWheel 高级副总裁容力，以下均以第一人称“我”代之）观察到过去许多大公司人员的迅猛增长，并非是由公司的效益或业务需求的预期所驱动的，而是完全取决于领导层的话语权。

FreeWheel 在 2007 年刚刚成立时只有 10 个人左右，到 2015 年 3 月我加入时已发展至 150 余人，而截至目前，规模又增加了一倍，达到 300 余人。经历十余年的发展，整个创业和探索的过程非常曲折和艰辛。但一直以来，团队的扩建都是由业务所驱动的（FreeWheel 每年保持大约 70% 的营收增长率），因此决定了我们更为良性的增长动因和态势。

人员持续扩张的背后，“Hire the Best”是在招贤纳士时秉持的最重要的基准。何谓 Best，如何定义？表层意义上来说，Best 就是顶尖的，具体分析主要体现在两个方面：一个主要是体现在校招中，学历与毕业院校的卓越性，另一个更重要的是在技术上的扎实积淀和积累，放在不同的招聘环境下具有不同的侧重点。但同时由于我们所处的行业日新月异，Hire the Best 的难点在于，你不能只看他现在会什么，而是要预测他将来是否能更快地学到更多。技术人的学习能力，其个人长期的潜质和潜力

应该在面试环节得到更多的加分。

我个人有一个理念，不管是在应聘还是在招聘的时候，都会秉承一个非常浅显的道理：我们看待一位应聘者适合不适合这家公司就看三点，第一，是应聘者能为公司带来什么价值；第二，公司能带给他 / 她什么，双方一定是一个互利的过程，仅仅看这个人给公司带来什么，不看公司带给他 / 她什么，这个交易也谈不成；第三，这个人能够在公司做多久。招聘是公司对人才的投资，而工作也是员工生命中贡献时间和青春很重要的一部分，所以是否能够长期在一家企业效力很重要。因行业而异，因公司而定。

打造具有良好工程师基因团队的三点要素

- 管理人员必须要从技术第一线提拔而来

如何从一线的工程师转成技术管理者，对个人和工程师团队文化来说都是非常重要的一环。我非常看重的一点是，技术公司的管理人员一定需要从技术第一线提拔而来，这样才能让公司保持工程师团队文化，而且这种文化才具有与生俱来的技术创新性。

我当时来 FreeWheel 就跟一线管理人员说，如果你的技术能力不能让你在团队里服众的话，那你还能给团队带来什么。在我的定义和 FreeWheel 的文化中，想转变为团队领导的人首先要在他的团队里具有最顶尖的技术水平。我们一般会以 10 人左右的团队为一个评价标准，如果他是技术大拿，只要他说了别人就觉得有信心，愿意按照他说的去做，那么他才能成为一个合格的团队领导者。很多硅谷的互联网公司也是这样，并不是按职级的高低或者资历来选择最适合的管理者。

- 培育一种开放的文化：信息和思维双维度

加入 FreeWheel 之前，我曾在包括创业公司和诸如微软、雅虎这样的大型外企工作了十几年，从最早的编程开始，到做一线的技术管理，再到

管理 100 人、300 人的团队，这种锻炼是一步一步做起来的。十多年里，从一线的管理逐渐转变到管理更大的团队，我也有了更扎实的体会。关于如何建设团队；如何做长期的规划；对于一个团队，以至于一个比较大的团队组织，怎样获得长期且稳定发展；如何在良性竞争中胜出等问题，微软和雅虎都为我提供了很好的学习平台，所以在加入 FreeWheel 之后，可以很自然地将这些理念和思路应用在具体的技术管理场景中。

在雅虎工作期间，我能感觉到那种硅谷所具有的沸腾的氛围，而且在硅谷不同公司之间的交流也非常紧密、频繁。在硅谷，不管是大公司还是小公司，大家就像是在一起创业的大家庭，时间长了就形成了所谓的“工程师文化”，说的简单点是自由、平等、开放。这种工程师文化会让每个人都持续保持进取的态度，很少有人抱怨，心态也会更加开放。

都说技术老大的角色决定了一家公司技术团队建设的模式，FreeWheel 联合创始人兼 CTO Diane 女士在工程师开放文化的构建上起了非常关键的带头作用，我们的开放主要体现为在信息维度的充分共享和在思维维度的创造激发。整个公司发展中，技术层面和商业模式层面的信息，可以在领导层和员工之间做到充分的沟通和共享，使得全员在任务处理上能更好地把握整体方向、理清事情的优先级，做对公司最重要的事情。另外，大家做事情都是本着平等开放的原则，不会因为说话的人不同，就对他提出的问题或者方案有不同的态度，不因人废言，更不因言废人。这样做的目的就是要激发大家的积极性，充分发挥每一位员工的创造力。

- 对 Engineering Excellence 的追求

追求 Engineering Excellence，是近期 FreeWheel 整个工程师团队的最大变化之一。在公司整体度过了生存期的挑战并进入到加速生长期时，我们所关注的事情，不再是到处救火，而是要追求卓越，要打造一个可以在未来几年里，支撑业务发展的优秀技术平台。在这个新的目标

下，FreeWheel 工程师团队也发生了不少变化，包括对 Full Life Cycle Engineer 和 DevOps 理念的倡导，对 CI/CD（持续集成 / 持续交付）等高效开发流程的探索和精进等。

敏捷开发模式在工程师团队的实践与落地

敏捷开发模式跟传统的开发差别之一是对变化的拥抱。传统的开发相对会拒绝变化，即计划制定好之后不希望有任何变化，有变化会对开发流程造成很大的影响。敏捷文化是拥抱变化，当有问题发生的时候需要会主动做调整，当然这对团队的组织和行为方式也带来了很大挑战。这里，FreeWheel 有一些比较好的实践经验可以分享。

第一是团队的组织。FreeWheel 一开始按照敏捷开发的原则将团队组织成 Scrum，在这种管理方式下，所有相关人员（主要是测试人员、开发人员和在美国的产品团队）会形成一个比较紧密的团队，消除相互间的壁垒。所有人从一开始就会融入到产品的设计、开发、测试里去，通过快速反馈和及时沟通，能最迅速地解决项目过程中各种问题。

第二是团队的管理。敏捷开发与传统模式相比，最显著的特点就是人和流程的关系有很大不同。传统开发中会制定一个固定的流程和周密的计划，人被添在了这个流程和计划中。敏捷开发的变化多、迭代快，如果套用一种过去的管理模式并强加给团队，往往会造成生产力的倒退。所以管理的方式应该是给团队更大的自主性，注重引导而非控制。一个公司往往会有多个敏捷团队，我们最终是让各个团队用他们的聪明才智解决问题，但同时各个团队之间又可以互通有无、取长补短。有的时候你会是第一个踩坑的人，但这个踩坑的过程是有价值的，得到的经验教训可以跟大家分享。

第三是做事的方式。当把任务拆分的更细时，反馈就能更及时。如果有一大块的东西挡在你的工作流里，会对你的敏捷性造成障碍。所以我们

一开始就会对所做的事情进行拆分，把任务拆分得更细，使得任务更容易被排期。同时在技术上实现 CI / CD，对产品持续不断地做测试，并且把它集成到研发流程中。当工程师做了某些改动后，整个系统会立刻产生结果评价，告知你做的是对还是错，然后大家基于这个结果再看如何继续推进。

1.SAFE (Scaled Agile Framework) 模式的尝试

与此同时，上面提到的“Hire the Best”理念也会对团队建设和管理带来一定挑战。如果团队人员的水平和能力呈现为梯队型，自然而然管理会相对容易；但如果大家都处于较为平均的基线范围内，就会面临更多协调、平衡及取舍方面的工作。最近 FreeWheel 正在尝试 SAFE，即将团队都分成不同的 Squad，一个 Squad 即为对产品有贡献的垂直的功能性团队，需要挑选并重组在前端、后端、数据库方面具有差异化优势的小组成员。但将优秀的成员都放在一起做一件事，很容易有人做的多，有人做的少，有人满意，有人不满意，这就跟项目管理、软技能训练等相关。

通过实行敏捷模式，工程师团队的整体趋势会越来越扁平化，但是扁平化主要还是发生在一个模块内部。比如，目前 FreeWheel 中国的工程师主要被划分为五个分支技术团队——AdServing、Forecasting、Data、UI 和 SRE，其中，AdServing 模块内部被大概分成了 6、7 个小型团队；Forecasting 被分成 3 个小型团队。其他四大模块情况也类似。在团队组织上，部分小型团队成员可以直接汇报给一线研发经理，例如，UI（负责核心业务系统的研发与测试）模块的 80 人团队大概有 6 个研发经理，他们各自领导着十多人的团队。因此，分支技术团队总负责人和普通工程师之间只有一级，向上的沟通和汇报会更加通畅。而且，负责管理的同事有更多的机会直接与小型业务开发团队进行协作，第一手掌握他们真实的想法、他们的困难和反馈意见，以便缩短做决策的时间周期。

2. 对 Full Life Cycle Engineer 理念的倡导

对于 Full Life Cycle Engineer，现在业界有两种声音，一种是 Full Stack（全栈）Engineer，指能够掌握前端，能写 Web，能写后端服务器，还能开发 Mobile 程序等，要掌握不同的技术。这种声音存在一定的争议。另外一种方向是 Full Life Cycle Engineer。比如写后端 C++ 的程序员就不需要掌握 Web 编程技术，但要能对使用 C++ 开发软件的整个生命周期熟稔于心：当新的功能需求提交出来时能用 C++ 对它进行设计，写完之后能够用相关的工具测试并将服务进行发布，后续还需要支持服务的运维。举个很简单的例子，如果测试、运维的环节被技术管理者忽略，随之而来的问题便是产品质量不可控、Bug 一堆、发布成功率低、运维人为事故频发。但对 Full Life Cycle Engineer 理念的倡导可以有效减少此类问题的发生。

因此，FreeWheel 非常倡导 Full Life Cycle Engineer，其直接优势在于可以消除上下游间明显的界限。比如一开始设计的时候就会将测试的实施方案、后期发布、运维上的问题一并考虑进去，而不是执行一半之后再去想怎么“填坑”。这是敏捷模式下最为可行的方式：如果整个软件交付过程中还要上下游切换，沟通成本就会很高，整个团队不可能敏捷起来。

3. 最快速地定位技术问题出现的根源

研发团队之间的协作非常重要，协作和沟通的效率很多时候都是解决复杂技术问题的关键。开发过程中，如果我们在上线环节亦或开发测试环境中发现了一个具体的技术问题，不同的研发团队负责人或该领域的资深专家会组成一个临时小团队（War Room），集中调研，分析和研究后认定问题根源并划定应该由哪个团队、哪个工程师具体跟进和解决。解决方案落地且开始执行后，大家还会回过头来进一步验证，确定目前的解决方案是否有其他风险存在。

4. 轮岗制的跨地域协作

对 FreeWheel 而言，最大的挑战之一是跨地域、跨时区的沟通与协作。FreeWheel 除了在美国、中国、法国有研发机构以外，在欧、美、亚洲的多个国家也会有自己的办公室。这些国家的团队规模及成熟度较高，但都存在较大的时差。

相比通过电话、电子邮件进行跨时区的交流，我们会尽可能地采用很多其他的方式优化跨时区、跨地域的合作和沟通。比如北京的各模块的研发团队会持续派轮岗人员，去美国等地不同的实验室和不同部门，与产品经理、客服经理在一起紧密地合作一到三个月。这样做的好处之一就是当产品经理面对更多的客户、分析客户需求的时候能得到研发团队第一手的支持，比如从技术角度看业务设计是否合理、能否实现、实现难度大小等。

另外，美国的产品经理会每个季度飞到北京来与这里的同事工作 1-2 周的时间商议未来 3-4 个月的研发及项目规划。这类协同工作的目的都是尽可能地减少跨时区、跨地域沟通的难度和障碍。如果团队之间没有很好的沟通，做完以后发现有很多问题，但由于一开始没有识别出来，最后就会演化得愈发严重。

让每个人感觉到自己的价值所在

通常情况下，人才培养关注个体，团队建设则更关注集体。在 FreeWheel，团队一方面要做事，另一方面也要育人，人才是团队的核心资产。团队的领导者必须要给所有技术人以存在感和价值感，让他们能看到自己的付出能改变产品、改变公司，甚至改变整个行业的走向。

1. 保持团队的方向感，让团队成员知道自己在做什么，将来又要做什么，能感觉到自己的价值所在

在 FreeWheel，不论每个同事的职务或工作职责如何，技术管理层都会首先注重帮助所有人在公司找准自己的定位，也会帮助他们分析清楚自

己的长处和短板，并有针对性地制定相应的工作计划及提高个人能力和适应职业发展的规划，帮助所有人找到他对公司的价值所在。这样做一方面会让每一位同事在工作上形成满足感，另外一方面也保证了他做出的贡献和体现的价值能被公司所认可，因而能有更好的职业发展机遇。

2. 保持团队的进步感，让团队成员感觉到自己每隔一段时间都能学到新的东西，从而值得为之付出努力

对工程师来说，发挥价值的地方仍在于自己的工作与产品的强关联。我们非常提倡让技术人深入到第一线工作中，让大家有机会去直接解答客户的问题。在 FreeWheel，销售和客户服务团队是直接做售前、售后服务的人，他们每个季度也会聚集到北京参加 PI Planning (Program Increment Planning)，参与制定下一季的产品计划，从而去打破技术人员、产品经理、销售这一套人员固定模式上的隔阂，让工程师团队能感觉到自己做的东西所带来的变化、对用户所产生的影响，直接参与到用户的意见反馈环节中去。

另外，随着业务的发展和体量的增大，FreeWheel 也在实行大规模的技术重构工作。当每天的工作中都有可能面临到新的技术挑战时，就能更好地保持团队的进步感。

以 UI 模块团队为例，从 2007 年到现在，前端技术框架经历了多次演变，去年开始 UI 团队（同时也包括其他团队）都在推行微服务化 SOA 架构。后端的业务逻辑会通过 Golang 服务封装成一个一个不同粒度的微服务，这样我们整个前端的框架会更多专注在业务交互上，跟核心业务逻辑直接相关的实现都会封装在底层的微服务框架上。但对于像 Go（包括我们前端目前正在使用的 React.js 框架）都是比较新的技术框架，会在技术细节方面面临更多的挑战，很难通过参考别人的经验就能获得有意义或有帮助的答案。所以我们更多是依赖于自己的研发团队，深入地具体分析某一

些技术问题产生的原因：为什么在我们这样的产品环境下会发生；能不能做一些实验，尝试找到一些解决办法；论证我们的解决办法，确保不会带来其他的衍生问题；对我们的系统稳定性、可靠性是否造成影响等。

很多的问题都是挑战，而克服挑战、解决问题的过程才能让大家感受到每天的进步以及明天未知的新奇。

3. 保证团队成员的归属感和自豪感，这样的团队才有凝聚力

FreeWheel 常常说我们的企业文化是 Proudly Unique、Deeply Caring 和 Purposeful，虽然是几个形容词，但是在日常工作中会遇到很多事情，大家会经常在一起谈解决问题的方法，遇到矛盾的时候又怎么解决，这会让所有人感觉与他人连同在一起，是有意义的存在。

另外，企业的定位和愿景同样会对技术人的价值感形成带来很大的影响。我入职前，跟 FreeWheel 的联合创始人兼 Co-CEO 进行了一轮电话面试，谈到公司的发展前景时，他用十年前的苹果和诺基亚做了一个类比，并说：

“虽然在公司规模上，我们在业界和最主要的竞争对手相比依然弱小，但是如果把竞争对手比作鼎盛时期的诺基亚，那我们就应该做苹果。”举这个例子的目的在于说明，即使目前的你尚且弱小，但只要你的产品有非常清晰的、适合市场发展的战略，也一定能够打败当时占领市场 80% 份额的巨无霸。

在这样的目标之下，我们也相信，优秀的工程师会逐渐认识到企业和自身的使命，认识到他在做的这项技术的无限的前景与价值。

版权声明

InfoQ 中文站出品

通往创新之巅：互联网技术架构创新案例和实践

©2018 飞维美地信息技术（北京）有限公司

本书版权为飞维美地信息技术（北京）有限公司和北京极客邦科技有限公司共同所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营容和路叶青大厦（北园）五层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn