QCon 10th
全球软件开发大会

十周年特刊 广州站

TENTH ANNIVERSARY SPECIAL ISSUE

主办方 Geekbang. 极客邦科技 InfoQ

目录
Contents

2019极客邦科技会议推荐

5

QCon 北京
全球软件开发大会
大会：5月6–8日
培训：5月9–10日

QCon 广州
全球软件开发大会
培训：5月25–26日
大会：5月27–28日

6

GTLC GLOBAL TECH LEADERSHIP CONFERENCE 上海
技术领导力峰会
时间：6月14–15日

GMTC 北京
全球大前端技术大会
大会：6月20–21日
培训：6月22–23日

7

ArchSummit 深圳
全球架构师峰会
大会：7月12–13日
培训：7月14–15日

10

QCon 上海
全球软件开发大会
大会：10月17–19日
培训：10月20–21日

11

GMTC 深圳
全球大前端技术大会
大会：11月8–9日
培训：11月10–11日

AiCon 北京
全球人工智能与机器学习大会
大会：11月21–22日
培训：11月23–24日

12 （月份）

ArchSummit 北京
全球架构师峰会
大会：12月6–7日
培训：12月8–9日

查看会议详情

Geekbang. 极客邦科技 InfoQ

卷首语 | 永远保持好奇

QCon 在今年步入了它的第十个年头，这十年间，QCon 经历了翻天覆地的变化，而也正是在这样一个特殊的关键节点，QCon 选择拥抱广州。作为国内最大、最具影响力的综合性技术大会，进入到广州后，根据华南地区互联网的特点，在本次迷你书内特意收录了微服务、移动与大前端以及大数据处理这三个热门专题下 5 位大神的见解，希望能为我们的开发者带来更多变的思路与不一样的收获。

微服务的火热

这十年间，企业的后台业务架构发生了翻天覆地的变化。在过去，一体式架构已经能满足业务的绝大多数需求，很少有人会意识到需要把服务拆分开来。但随着互联网的飞速发展，云计算与容器技术的普及，一款产品的功能日益庞大甚至臃肿，动辄就要对整体进行更新维护，实在是太过伤筋动骨。

这两年的大红大紫的微服务恰好是该问题的最优解，提到微服务每个人似乎都能说上两句。像 Spring Cloud、Dubbo、Service Mesh 等都是当前最火热的微服务框架。

其中的 Service Mesh 更是微服务中的当红炸子鸡，被视为微服务的未来发展趋势。再加上一线互联网公司的尝试与带头作用，很多企业包括开发者都在尝试这个框架。这些框架在性能上各有优劣，对于开发者来说，如何选择最适合业务发展的微服务框架？向微服务转型，需要面临哪些问题？使用微服务，是选择自研、开源还是基于开源的微创新？ Service Mesh 这种独特的思考模式以及工程化，能给我们开发者带来怎样不同的灵光与思路？

大数据的实时流计算之争

说起微服务，就不得不提我们的大数据实时处理技术。在后端服务架构运转效率提高的同时，对数据的处理效率的要求也在同步跟进。在互联网越来越快的今天，用户的“耐性”正在变差，企业对数据服务实时化的需求也日益增多，打车、外卖、网购、在线视频等场景下，用户已经不能忍受较长时

间的等待，企业对于大数据实时决策的要求也越来越严苛。

而且各家厂商都在不遗余力地试用新的流计算框架，Flink、Storm、Kafak 等，这些大数据流计算的框架之争，我们从中又该如何抉择？

大前端与移动开发的融合

与其它领域下技术之间的争斗不同，前端领域有着融合统一的态势。随着前端技术的发展以及移动互联网普及，大前端的概念应运而生。由于业务的快速增长，单纯使用 HTML5+CSS 的组合已经无法适应当下前端与移动端上的各种应用，PC 端与移动端上各种跨平台开发的方案一时间百花齐放。

原因就在于 Web 平台才是真正意义上的跨平台，所有主流操作系统都能通过浏览器来访问相同的网页。但是由于 PWA、WebAssembly 等进一步增强 Web 能力的技术和标准并未成熟，导致当下前端局面“混乱”的状态。前端技术日新月异，在技术的演进过程中，有哪些设计思想和架构经验值得前端工程师学习呢？

再回到移动开发上来，从早期的 PhoneGap，到后来的 React Native，再到现在的 Flutter，众多跨平台开发框架的应用实践，与原生技术展开了一场博弈。用户该如何在众多选择中，做好技术选型及落地实践？使用跨平台开发框架，应该注意哪些问题？对其通病有何应对方案？

我们可以看到，开发圈子里，无论是哪一个领域，都存在着创新与抉择，没有一款开发语言、或者是框架能够完美适配全部的应用场景。那面对不同的场景与需求，我们又该如何进行选择？互联网，这个圈子实在是太大太深了，所以说，好奇是伴随程序员完整职业生涯的命题。喜欢尝试新鲜事物，喜欢用不同的方法、不同的语言来实现相同的目的，相信这是每一名优秀程序员都有的心态。

恕我直言，你可能误解了微服务

作者 刘超

随着云计算和容器技术的普及，互联网 IT 基础设施已经发生了很大的变革，也推动了微服务技术的大量采用和落地。现在的技术人，不谈微服务已经要跟不上形势了。但是你真的对微服务有正确的理解吗？要向微服务转型，有哪些问题和挑战摆在面前？如何拨开现代各种技术栈的迷雾看清微服务的发展趋势，选择最适合团队的技术方向？本次 InfoQ 记者采访了网易杭州研究院云计算技术部的首席架构师刘超，为大家分享他对这些问题的看法。刘超也是今年 5 月份 QCon 全球软件开发大会广州站「微服务实战」专题的出品人，将为大家策划几场微服务相关的内容丰富的分享。

InfoQ：刘超老师，请先介绍一下自己吧。

刘超：我是网易云的首席架构师，主要负责两部分工作，对内支撑网易核心业务上云，例如考拉，云音乐，云课堂，对外输出网易的微服务经验，帮助客户搞定容器化与微服务化架构，已经在银行、证券、物流、视频监控、智能制造等多个行业落地。

InfoQ：网易云在微服务方面的探索有哪些？落地过程中有哪些难点？

刘超：网易云的技术团队在博客时代就开始探索互联网架构，是在支撑博客用户量、访问量就爆发式增长的过程中，构建了聚焦微服务的网易云轻舟平台，并支撑内部考拉、云音乐、云课堂等核心业务。

在实施微服务的过程中，难点层出不穷，可谓见山开路，遇水搭桥。

实施服务化架构之后，首先实现的功能是进行统一的注册发现和 RPC 的透明封装，但是服务拆分多了，在应用层面就遇到以下问题：

- 服务雪崩：即一个服务挂了，整个调用链路上的所有的服务都会受到影响；
- 大量请求堆积、故障恢复慢：即一个服务慢，卡住了，整个调用链路出现大量超时，要长时间等待慢的服务恢复到正常状态。

在基础设施层面，还有另外的问题：

- 服务器资源分配困难，服务器机型碎片化：服务多了，各个团队都要申请服务器，规格不一，要求多样，管理十分困难；
- 一台服务器上多个进程互相影响、QoS 难以保障：采用虚拟机或者物理机的部署，往往会多个进程放在一台服务器上，高峰期影响严重；
- 测试环境数量大增，环境管理、部署更新困难：每个团队都有反复部署测试环境，手动部署或者脚本部署过于复杂。

为了解决这些问题，我们在应用层面实施了以下方案：

- 通过熔断机制，当一个服务挂了，被影响的服务能够及时熔断，使用 Fallback 数据保证流程在非关键服务不可用的情况下，仍然可以进行。
- 通过线程池和消息队列机制实现异步化，允许服务快速失败，当一个服务因为过慢而阻塞，被影响服务可以在超时后快速失败，不会影响整个调用链路。

在基础设施层面，我们实施了以下的方案：

- 统一基础设施，拥抱容器标准，解决服务器碎片化和服务之间的隔离问题；
- 统一编排和弹性伸缩平台，2015 年拥抱 Kubernetes 标准，解决了部署困难，环境不一致的问题；
- 打造 CI/CD 服务，抽象出产品、环境等多级概念，实现从代码到测试到上线的自动部署。

随着我们支撑的内部业务越来越多，就进一步遇到了以下问题：

- 微服务框架选型不一，技术无法积累，面向业务定制化严重，上手成本高；
- 传统依赖于应用运维的排障复杂度高，传统监控服务无法满足需求；
- 故障演练手段不一，硬编码随处可见；
- API 版本管理混乱，无统一的监控，治理，无开发标准；
- 分布式事务支持方式不一，和业务绑定严重。

为了解决这些问题，我们实施了以下方案：

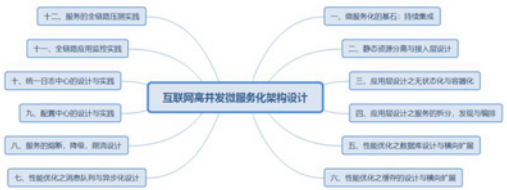
- 微服务框架与开源技术栈统一，将服务治理逻辑抽离、以无侵入方式实现、支持 Spring Cloud、Dubbo 等开源技术栈；
- 全链路跟踪服务与日志服务依据 ID 进行联系，以发现故障点上下文；
- 在 Agent 引入故障注入服务，可统一进行故障演练；
- 服务通过 API 网关暴露，引入 API 管理、测试平台，自动 Client SDK 生成；
- 实现 TCC 中间件、事务消息队列等标准中间件。

InfoQ：你如何理解微服务？微服务在当前技术形势下处于一个什么样的位置？

刘超：微服务是一个非常复杂的问题，在业内会有一些误解：

- 微服务主要的工作是服务拆分，主要考虑将服务拆分成什么粒度以及如何进行拆分；
- 微服务是一个运动式的过程，把大家关起门来封闭开发一个月，就能把架构修改好了，以后就万事大吉了；

- 微服务仅仅是一个技术问题，交给开发团队或者运维团队去搞就可以了。



微服务绝不仅仅是服务拆分，就像上图所示，拆分只是实施微服务十二个要点之一，因为拆分了服务之后，会面临上面我们遇到的所有问题，没有相应的工具和平台，拆分的越细，越是一场灾难。

微服务绝不是一个运动式的过程，而是应该渐进的过程，一旦实施了微服务，就处于业务系统不断更新和迭代的状态中，也处于不断的拆分和组合中。所以不建议一开始就拆的特别细，不建议一劳永逸，而是随着慢慢的拆成几个，十几个，几十个，上百个的过程，将十二个要点所需要的工具、团队、员工能力慢慢匹配到微服务状态。

微服务绝不仅仅是个技术问题，牵扯到 IT 架构、应用架构、组织架构多个方面。微服务必定带来开发、上线、运维的复杂度的提高，如果说单体应用复杂度为 10，实施了微服务后的复杂度将是 100，配备了相应的工具和平台后，可以将复杂度降低到 50，但仍然比单体复杂的多。

所以实施微服务是有成本的，只有在业务层面遇到不微不行的痛点，例如痛到影响收入，痛到被竞争对手甩在后面，所以微服务往往是业务驱动或者高管驱动的，而实施微服务的结果又必然会影响到组织架构的变化，例如运维和开发的界限模糊——DevOps，专门中间件和架构师团队的成立，数据中台和业务中台组的建立，小团队自主决策等。

目前，大多数企业都意识到了微服务的重要性，但是各处的阶段不同，我把微服务分成三个阶段：

- 微服务 1.0，仅使用注册发现，基于 SpringCloud 或者 Dubbo 进行开发，目前意图实施微服务的传统企业大部分处于这个阶段，或者正从单体应用，向这个阶段过渡，处于 0.5 的阶段；
- 微服务 2.0，使用了熔断，限流，降级等服务治理策略，并配备完整微服务工具和平台，目前大部分互联网企业处于这个阶段。传统企业中的领头羊，在做互联网转型的过程中，正在向这个阶段过渡，处于 1.5 的阶段；
- 微服务 3.0，Service Mesh 将服务治理作为通用组件，下沉到平台层实现，使得应用层仅仅关注业务逻辑，平台层可以根据业务监控自动调度和参数调整，实现 AIOps 和智能调度。目前一线互联网公司在进行这方面的尝试。

InfoQ：你怎么看微服务未来的发展趋势?

刘超：前面大概谈了一下微服务 3.0，这里详细说一下我眼中的微服务的发展趋势。

第一个就是 Service Mesh，他的主要作用就是将服务治理下沉到平台层，进行统一的治理。

为什么会这样呢？因为无论是在我们内部，还是在外部企业，都能看的这样的趋势。

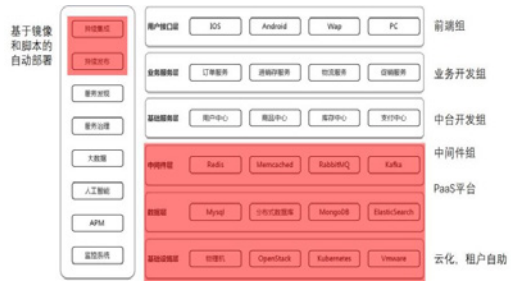
最初只有物理机，虚拟机是放在云平台上，由运维组统一管理的。

后来因为能力复用和开发速度的需要，数据库、中间件成为了 PaaS 平台用于部署通用的组件，持续发布也成了 PaaS 平台，用于部署客户的业务，所以这两部分也平台化了。

随着越来越多的业务需要进行服务治理，微服务框架，APM，也成为了平台的一部分。

但是微服务框架的统一，涉及多语言的问

题，也涉及和应用层绑定的问题，无论是 Spring Cloud 还是 Dubbo，都很难完全平台化，所以需要 Service Mesh，通过 sidecar 的方式，将控制面和数据面隔离，通过非侵入的模式进行流量拦截，实现真正的治理平台化。



第二个就是 AIOps 和智能调度，就是通过对于海量数据中心收集的监控数据和业务数据，实现业务的自动调度和参数调整。

这个看起来很遥远，其实不然，如果大家感兴趣的话，可以在网上搜索一下，Google 在 2011 年就公布了自己数据中心收集的监控数据，并在 2014 年发表论文《Machine Learning Applications for Data Center Optimization》，使用 AI 技术优化数据中心的效率。

而国内一线互联网公司也在 2018 年公布 4000 台服务器真实数据集，也在干和 Google 类似的事情。

我们观察到，当数据中心的机器规模突破十万台的时候，效率的提高就变成了一件能够节省大量成本的事情，所以开始引起重视。而能做到这件事情，往往依靠的就是数据驱动的智能调度。

为了支撑强大的调度功能，Google 开发了 Borg，Twitter 壮大了 Mesos，并通过将这些调度平台和机器学习相结合，实现自动化的智能调度，国内一线互联网公司也在进行着积极的尝试。

随着微服务化和容器化，服务的数量会十分的庞大，从而运维难度大幅度提高，原来仅仅会运维物理机和虚拟化的技术人员是不够的，而运维 Kubernetes 和 Docker 的人会比较贵，使得人力成本大幅度提高。

很多组织从物理机时代，到虚拟化时代，到云时代，再到容器时代，运维团队的规模是越来越大的，每个人的薪资也越来越高。

所以将来只运维少量节点的私有化容器平台，势必从成本上来讲是不划算的，当出现有公信力的公有云平台，则势必使用公有云成为节约成本的理智选择。

例如亚马逊、谷歌等公有云平台就没有问题，谷歌里面的运维工程师相当贵，他们掌握最先进的技术是没有任何问题的，但是他们会通过各种自动化，甚至智能化的技术，管理全球的几百万台机器，这样成本摊下来就不是很高了。如果你只是运维一个几十个节点，最多几百个节点的容器平台，同样需要招一些这么贵的人，一般的企业肯定受不了。所以将来要么是大规模公有云平台，要么是土豪如电信金融行业的自建云平台，都会出现超大规模的场景，基于 AIOps 和智能调度节约成本，就是势在必然的了。

InfoQ：QCon 广州的「微服务实战」专题下设置了 4 个演讲，作为出品人，你如何策划这 4 个演讲，想给参会者呈现微服务的哪些方面？

刘超：基于我们自己的微服务实践，和对于微服务发展阶段的理解，作为「微服务实战」专题的出品人，我计划全方位展示微服务在主流公司的主流技术方向的实践和未来方向。

第一个方面就是基于 Dubbo 的大规模微服务实践的场景，Dubbo 是应用范围非常广的微服务框架，很多企业都是基于 Dubbo 做的，Dubbo 的实践是微服务实施过程中绕不过去的一环，这个主题能够解决很多技术人员实施海量 Dubbo 服务的时候遇到的问题。

第二个方面就是基于 Spring Cloud 的大规模微服务实战的场景，Spring Cloud 是近年来新兴的微服务框架，很多新实施微服务的，会选择基于 Spring Cloud，但是 Spring Cloud 虽然组件丰富，可选项多，但是也很复杂，学习曲线高，如何再海量场景下进行改进和适配，是经常遇到的问题，这个主题能够给予技术人员实施 Spring Cloud 微服务的时候以借鉴意义。

第三个方面就是 Service Mesh 在高并发场景下的实践场景，前面说了 Service Mesh 是一个趋势，一线互联网公司都在尝试，但是这个技术太新了，很多坑还在踩，这个主题能够带给技术人员最前沿微服务技术的落地实践，给想试试 Service Mesh 的技术人员以借鉴意义。

第四个方面就是微服务框架各个方向的发展与未来趋势，微服务涉及范围广，技术选型难，很多没有实施微服务的技术人员面临如此多的技术名词，无所适从，选稳定的，会不会过时被淘汰，选先进的，会不会冒进出线上事故，选错了技术方向，万一开源的不维护了就麻烦大了，这个主题会讲解微服务发展的技术趋势和各个方向的优劣对比，给选型困难的技术人员以参考。

跨平台开发框架的大旗，究竟谁能扛起来？

作者 刘 超

近年来，移动端上各种跨平台开发方案百花齐放，一方面是因为随着移动互联网的迅猛发展，纯原生开发无法满足业务快速增长的需求；另一方面，跨平台可以增加代码复用，降低开发成本。在移动终端设备的软硬件、操作系统、开发工具链和技术社区等日趋成熟的今天，众多开发者对“造轮子”跃跃欲试。

从早期的 PhoneGap，到后来的 React Native，再到现在的 Flutter，众多跨平台开发框架的应用实践，与原生技术展开了一场博弈。用户该如何在众多选择中，做好技术选型及落地实践？使用跨平台开发框架，应该注意哪些问题？对其通病有何应对方案？

InfoQ 记者带着这些问题，采访到了腾讯微信客户端工程师方秋枋老师。另有结合微信团队目前采用的跨平台方案分析，希望能对大家有所启发！

另外，方秋枋老师也将在 QCon 全球软件开发大会（广州站）分享题为《基于 C++ 构建微信客户端跨平台开发框架》的话题，感兴趣的同学也可以关注下。

InfoQ：从早期的 PhoneGap，到后来的 React Native，再到现在的 Flutter，移动端跨平台开发框架一直是圈子里的热点。能否谈谈你对于跨平台开发框架的认识？这些年技术在不断迭代和演讲，你怎么看待他们背后的发展逻辑？

方秋枋："Write once, run anywhere" 一直以来就是开发者的梦想。在移动客户端领域，主流的跨平台开发框架大体经历了三个阶段。

第一阶段，主要通过 WebView 绘制界面，并通过 JavaScript Bridge 将系统的一部分能力暴露给 JS 调用。 PhoneGap、Cordova 都可以归属于这一类。

第二阶段，大家发现用 WebView 承载界面有性能等各种问题。于是将绘制交回给原生，通过 JS 来调用原生控件，编写业务代码。Weex、RN 就是其中的佼佼者，这也是现在绝大部分跨平台框架的思路。

第三阶段，虽然使用原生控件承载 UI 解决不少渲染的问题。但是处理平台差异性仍然需要耗费极大精力，效果也不尽如人意。这个时候，Flutter 提出的解决方案，就是连绘制引擎也自研，尽可能减少不同平台之间的差异性 , 同时保持和原生开发一样的高性能。因此目前业界对 Flutter 的关注度也是最高的。

对于这样的发展路线，个人认为根本原因还是 Web 标准和能力在移动客户端得不到很好的扩展与支持。实际上，Web 平台才是真正意义上的跨平台 , 我们在所有主流操作系统上都能通过浏览器来访问相同的网页。目前第二，第三阶段方兴未艾，而 PWA、WebAssembly 等进一步增强 Web 能力的技术和标准并未成熟。因此，在可预见的未来，移动端跨平台开发还是主要以当前的形式为主。但是，随着 Web 标准和能力的增强，最终绝大多数应用，只需要使用 Web 栈技术即可构建。

InfoQ：据了解，用户在选择跨平台开发框架时，比较看重其是否具备热更新的能力。但在 2017 年 3 月，苹果下发了警告邮件，禁止 JSPatch 等 iOS App 热更新方案，你如何看待“热更新”这件事?

方秋枋：前段时间围绕热更新， InfoQ 主编徐川已经写了一篇非常好的文章《移动开发的罗曼蒂克消亡史》。因此在这里就不再赘述。

目前 iOS 平台的热更新更多地是用于修复 Bug，而不是直接拿来进行业务需求的开发和更新。实际上对业务体验是无害的，但是也违反了苹果的规则。因此，大家目前在 iOS 平台基本只敢偷偷摸摸地做，没有新的开源项目公布。而安卓就百花齐放，有非常多的热更新框架可以选择。然而，安卓平台侧拥有了非常强的热更新能力，用户体验有变得更佳么？目前看来，好像还是没有的，因为安卓平台权限控制等相关问题的存在，反而厂家会利用漏洞来骚扰用户，获取隐私。

从用户角度来说，怎么真正利用好热更新能力来服务用户，提升应用体验，才是开发者真正需要考虑的问题。但是这也需要取舍。更开放的平台策略更允许热更新能力的存在，同时也会带来其他的一些问题。

从开发者角度，我希望平台能够允许和放开热更新技术。但从一个普通用户的角度，我更喜欢当前苹果的策略。

InfoQ：有人说，国外 App 开发偏重于“原生”，而国内 App 讲究的是“迭代”。你认同这句话吗？为什么?

方秋枋：我不太认同这句话。其实国内国外的应用，都在不断地迭代更新。只不过当下情况，国外 App 开发更倾向于使用原生技术进行开发，而国内则对跨平台的热更新能力非常关注。

InfoQ：2018 年，Airbnb 放弃使用 React Native，回归使用原生技术，究其原因主要是：项目庞大后，维护困难；第三方库良莠不齐；兼容上需要耗费更多精力……这是否是跨平台框架存在的通病？有什么解决办法?

方秋枋：这的确是跨平台框架存在的通病。跨平台意味着需要花费很多时间来解决平台差异性问题，同时要面临第三方库不够原生平台丰富健壮的现状。跨平台其实是牺牲部分功能和体验，换取开发速度和一致性的权衡，并不是业务开发的银弹。

目前并没有一个能完善解决这些问题的解决方案。 Flutter 可能相对来说在兼容上会做得好一点，通过自底向上自研框架来尽可能减少平台差

异，但是可靠性仍需进一步检验，引入 Flutter 需要大家对自己公司业务有非常清晰的认知，并权衡好利弊。

对于初创公司和普通开发者，我个人的建议是可以利用小程序来进行跨平台开发，把这些繁琐的兼容问题抛给小程序开发框架的开发者处理，这样就可以把更多精力放在业务开发，也省却了维护两个平台发版的工序。

InfoQ：目前，微信团队采用的是哪种跨平台开发框架？基于哪种核心语言？是出于什么考虑选择这种语言的？聊一聊该框架的实现原理？

方秋枋：目前微信团队并没有统一使用一套跨平台开发框架。我们也在积极探索各种可能性。

微信小程序，是前面提到的第一阶段和第二阶段的融合做法，利用 Webview 作为渲染容器，通过规定 Web 技术栈的子集（WXML、WXSS），让客户端获得更多的渲染性能优化空间，同时通过规范化组件的使用，逐步引入原生控件代替 Webview 渲染。

除此之外，我们还探索了基于 C++ 进行跨平台开发。实质上还是第二阶段的做法，只不过是使用 C++ 作为中间语言，去编写业务代码和调用原生控件。为什么要采用 C++ 呢？因为 C++ 安全性会更高一点，同时性能也会更佳。可以应用在一些对安全性要求比较高的场景。如果大家对基于 C++ 构建跨平台框架和业务开发有兴趣。可以关注一下 QCon 全球软件开发大会（广州站），我会更详细地阐述如何基于 C++ 构建跨平台开发框架。

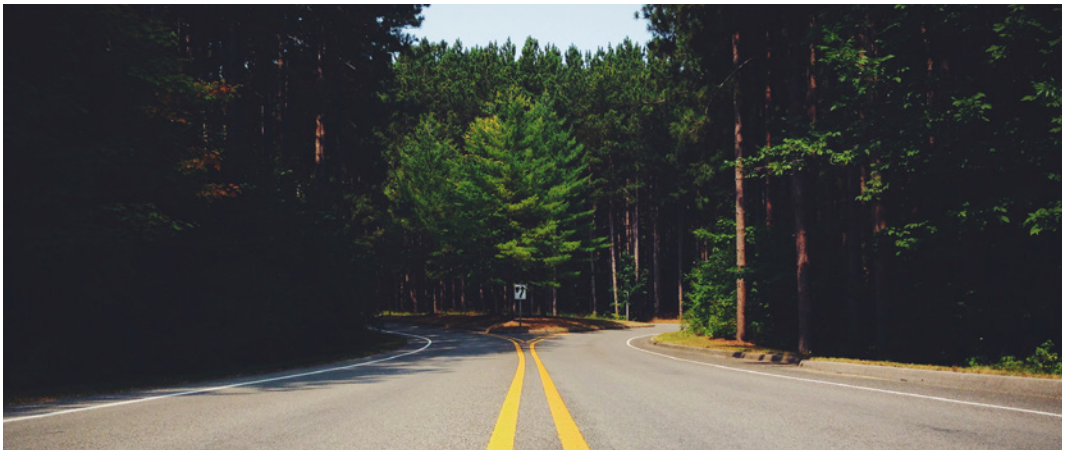
InfoQ：你认为跨平台开发的难点是什么？Flutter 是如何解决这些问题的？你如何看待 Flutter 的发展？

方秋枋：从框架开发者的角度，我认为跨平台开发的难点就在于处理平台差异性。从框架使用者的角度，难点在于如果框架出问题了，维护成本将会变得非常高。

Flutter 通过自底向上自研框架来尽可能减少平台差异，同时将 Dart 语言，渲染引擎等组成部件都进行完整开源，任何人都可以调试源码，当然维护门槛也是很高的。我认为如果谷歌愿意继续大力投入 Flutter 的开发，Flutter 会是跨平台开发框架非常好的一个选择。

大前端时代，如何做好 C 端业务下的 React SSR？

作者 狼叔



React 在中后台业务里已经很好落地了，但对于 C 端（给用户使用的端，比如 PC/H5）业务有其特殊性，对性能要求比较苛刻，且有 SEO 需求。另外团队层面也希望能够统一技术栈，小伙伴们希望成长，那么如何能够完成既要、也要、还要呢?

本次分享主要围绕 C 端业务下得 React SSR 实践，会讲解各种 SSR 方案，包括 Next.js 同构开发，并以一次优化的过程作为实例进行讲解。其实这些铺垫都是在工作中做的 Web 框架的设计而衍生出来的总结。这里先卖个关子，自研框架基于 Umi 框架并支持 SSR 的相关内容留到广州 QCon 上讲。下面开始正题。

曾和小弟讨论什么是 SSR？他开始以为 React SSR 就是 SSR，这是不完全对的，忽略了 Server-side Render 的本质。其实从早期的 cgi，到 PHP、ASP，jsp 等 server page，这些动态网页技术都是服务器端渲染的。而 React SSR 更多是强调基于 React 技术栈进行的服务器端渲染，是服务器端渲染的分类之一，本文会以 React SSR 为主进行讲解。

1、为什么要上 SSR？

对于 SSR，大家的认知大概是以下 3 个方面。

- SEO：强需求，被搜索引擎收录是网站的基本能力。
- C 端性能：至少要保证首屏渲染效率，如果秒开率都无法保证，那么用户体验是极差的。
- 统一技术栈：目前团队以 React 为主，无论

从团队成长，还是个人成长角度，统一技术栈的好处都是非常明显的。

诚然，以上都是大家想用 SSR 的原因，但对笔者来说，SSR 的意义远不止如此。在技术架构升级的过程中，如果能够同时带给团队和小伙伴成长，才是两全其美的选择。目前我负责优酷 PC/H5 业务，在优酷落地 Node.js，目前在做 React SSR 相关整合工作。玉伯曾讲过在 All in Mobile 的时代的尴尬——对于多端来说是毁灭性的灾难。押宝移动端在当时是正确的选择，但在今天获客成本过高，且移动端增速不足，最好的选择就是多端在产品细节上做 PK，PC/H5 业务的生机也正在于此。

然而历史包袱如此的重，有几方面原因：

1. 页面年久失修；
2. 移动端在 All in Mobile 时代并没有给多端提供技术支持，PC/H5 是掉队的，需要补齐 App 端的基本能力；3）技术栈老旧，很多页面还是采用 jQuery 开发的，对于团队来说，这才是最痛苦的事儿。

其实所有公司都是类似的，都是用有限资源做事，希望最少的投入带来最大化的产出。可以说，通过整合 SSR 一举三得，将 Node.js 和 React 一同落地，顺便将基础框架也落地升级，这样的投入产出是比较高的。

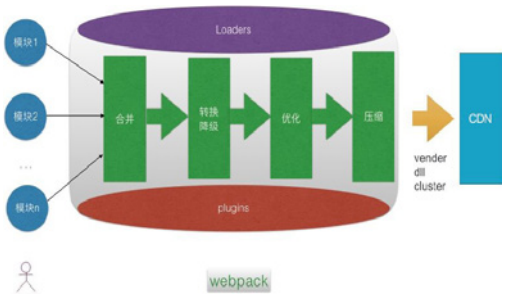
2、从 CSR 到 SSR 演进之路

SSR 看起来很简单，如果细分一下，还是略微负责的，下面和我一起看一下从 CSR 到 SSR 演进之路。

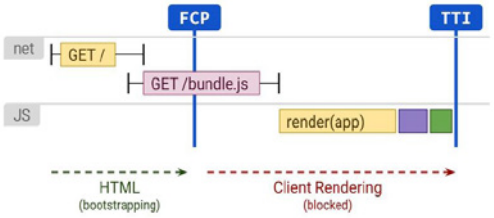
客户端渲染

客户端渲染是目前最简单的开发方式，以 React 为例，CSR 里所有逻辑，数据获取、模板编译、路由等都是在浏览器做的。

Webpack 在工程化与构建方便提供了足够多便利，除了提供 Loader 和 Plugin 机制外，还将所有构建相关步骤都进行了封装，甚至连模块按需加载都内置，还具备 Tree-shaking 等能力，外加 Node cluster 利用多核并行构建。很明显这是非常方便的，对前端意义重大的。开发者只需要关注业务模块即可。



常见做法是本地通过 Webpack 打包出 bundle.js，嵌入到简单的 HTML 模板里，然后将 HTML 和 bundle.js 都发布到 CDN 上。这样开发方式是目前最常见的，对于做一些内部使用的管理系统是够的。



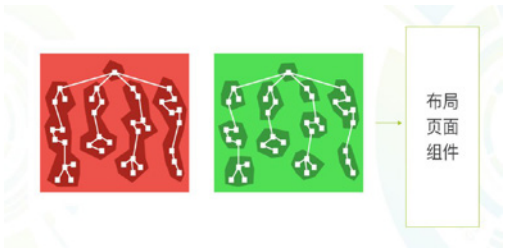
CSR 缺点也是非常明显的，首屏性能无法保障，毕竟 React 全家桶基础库就很大，外加业务模块，纵使按需加载，依然很难保证秒开的。

为了优化 CSR 性能，业界有很多最佳实践。在 2018 年，笔者以为 React 最成功的项目是 CRA（create-react-app），支付宝开发的 Umi 其实也是类似的。他们通过内置 Webpack 和常见 Webpack 中间件，解决了 Webpack 过于分散的问题。通过约定目录，统一开发者的开发习惯。

与此同时，也产生了很多与时俱进的最佳实践。使用 react-router 结合 react-loadable，更优雅的做 dynamic import。在页面中切换路由时按需加载，在 Webpack 中做过代码分割，这是极好的实践。

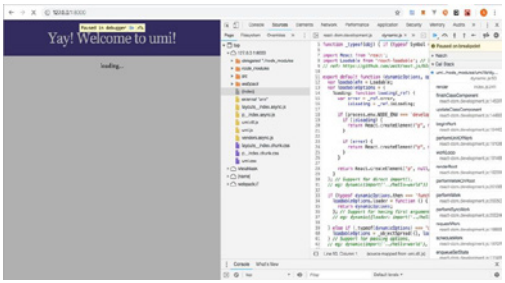
以前是打包 bundle 是非常大的，现在以路由为切分标准，按需加载，效率自然是高的。

Umi 基于 react-router 又进步增强了，约定页面有布局的概念。




```
export default {
  routes: [
    { path: '/', component: './a' },
    { path: '/list', component: './b',
Routes: ['./routes/PrivateRoute.js'] },
    { path: '/users', component: './
users/_layout',
      routes: [
        { path: '/users/detail',
component: './users/detail' },
        { path: '/users/:id', component:
'./users/id' }
      ]
    },
  ],
};
```

这样做的好处，就有点模板引擎中 include 类似的效果。布局提效也是极其明显的。为了演示优化后的效果，这里以 Umi 为例。它首先会加载 index 页面，找到 index 布局，先加载布局，然后再加载 index 页面里的组件。下图加了断点，你可以很清楚的看出加载过程。



在 create-react-app（cra）和 Umi 类似，都是通过约定，隐藏具体实现细节，让开发者不需要关注构建。在未来，类似的封装还会有更多的封装，偏于应用层面。笔者以为前端开发成本在降低，未来有可能规模化的，因为框架一旦稳定，就有大量培训跟进，导致规模化开发。这是把双刃剑，能满足企业开发和招人的问题，但也在创新探索领域上了枷锁。

预渲染

SPA(单页面应用) 的主要内容都依赖于 JavaScript(bundle.js) 的执行，当首页 HTML 下载下来的时候，并不是完整的页面，而是浏览器里加载 HTML 并 JavaScript 文件才能完成渲染。用户在访问的时候体验会很好，但是对于搜索引擎是不好收录的，因为它们不能执行 JavaScript，这种场景下预渲染 (Prerending) 就派上用场了，它可以帮忙把页面渲染完成之后再返回给爬虫工具，我们的页面也就能被解析到了。

CSR 是由 bundle.js 来控制渲染的，所以它外层的 HTML 都很薄。对于首屏渲染来说，如果能够先展示一部分布局内容，然后在走 CSR 的其他加载，效果会更好。另外业内有太多类似的事件了，比如用 less 写 css，coffee 写 js，用 markdown 写博客，都是需要编译一次才能使用的。比如 Jekyll/Hexo 等著名项目，它们都非常好用。那么基于 React 技术，也必然会做预处理的，Gatsby/Next.js 都有类似的功能。将 React 组建编译成 HTML，可以编译全部，也可以只编译布局，对于页面性能来说，预渲染是非常简单的提升手段。其原理 JSX 模板和 Webpack stats 结合，进行预编译。

编译全部：纯静态页面。

只编译布局：对于 SPA 类项目是非常好，当然多页应用也可以只编译布局的。

生成纯 HTML，可以直接放到 CDN 上，这是简单的静态渲染。如果不直接生成 HTML，由 Node.js 来接管，那么就可以转换为简单的 SSR。

无论 CSR 还是静态渲染，都不得不面对数据获取问题。如果 bundle.js 加载完成，Ajax 再获取的话，整个过程还要增加 50ms 以上的交互时间。如果预先能够得到数据，肯定是更好的。

类似上图中的数据，放在 Node.js 层去获取，并注入到页面，是服务器端渲染最常用的手段，当然，服务器端远不止这么简单。

服务器端

纯服务器渲染其实很简单，就是服务器向浏览器写入 HTML。典型的 CGI 或 ASP、PHP、JSP 这些都算，其核心原理就是模板 + 数据，最终编译为 HTML 并写入到浏览器。

第一种方式是直接将 HTML 写入到浏览器，具体如下。

上图中的 renderToString 是 react SSR 的 API，可以理解成将 React 组件编译成 HTML 字符串，通俗点，可以理解 React 就是当模板使用。在服务器向浏览器写入的第一个字节，就是 TTFB 时间，然后网络传输时间，然后浏览器渲染，一般关注首屏渲染。如果一次将所有 HTML 写入到浏览器，可能会比较大，在编译 React 组件和网络传输时间上会比较长，渲染时间也会拉长。

第二种方式是就采用 Bigpipe 进行分块传输，虽然 Bigpipe 是一个相对比较“古老“的技术，但在实战中还是非常好用的。在 Node.js 里，默认 res.write 就支持分块传输，所以使用 Node.js 做 Bigpipe 是非常合适的，在去哪儿的 PC 业务里就大量使用这种方式。

以上 2 种方法都是服务器渲染，在没有客户端 bundle.js 助力的情况下，第一种情况除了首屏后懒加载外，客户端能做的事儿不多。第二种情况下，还是有手段可以用的，比如在分块里写入脚本，可以做的的事情还是很多的。

```
res.write("<script>alert('something')</
script>")
```

渐进混搭法

渐进混搭法是将 CSR 和 SSR 一起使用的方式。SSR 负责接口请求和首屏渲染，并客户端准备数据或配合完成某些生命周期的操作。

首先，在服务器端生成布局文件，用于首屏渲染，在布局文件里会嵌入 bundle.js。当页面加载 bundle.js 成功后，客户端渲染就开始了。通常客户端渲染过程都会在 domReady 之前，所以优化效果是极其明显的。

Bigpipe 可以使用在分块里写入脚本，在 React SSR 里也可以使用 renderToNodeStream 搞定。React 16 现在支持直接渲染到节点流。渲染到流可以减少你的内容的第一个字节（TTFB）的时间，在文档的下一部分生成之前，将文档的开头至结尾发送到浏览器。当内容从服务器流式传输时，浏览器将开始解析 HTML 文档。渲染到流的另一个好处是能够响应。实际上，这意味着如果网络被备份并且不能接受更多的字节，则渲染器会获得信号并暂停渲染，直到堵塞清除。这意味着您的服务器使用更少的内存，并更加适应 I / O 条件，这两者都可以帮助您的服务器处于具有挑战性的条件。

最简单的示例，你只需要 stream.pipe(res, { end: false })。

```
// 服务器端
// using Express
import { renderToNodeStream } from
"react-dom/server"
import MyPage from "./MyPage"
app.get("/", (req, res) => {
  res.write("<!DOCTYPE
HTML><HTML><head><title>My Page</title></
head><body>");
  res.write("<div id='content'>");
  const stream =
renderToNodeStream(<MyPage/>);
  stream.pipe(res, { end: false });
  stream.on('end', () => {
    res.write("</div></body></HTML>");
    res.end();
  });
});
```

当 MyPage 组件的 HTML 片段写到浏览器里，你需要通过 hydrate 进行绑定。

```
//浏览器端
import { hydrate } from "react-dom"
import MyPage from "./MyPage"
hydrate(<MyPage/>, document.
getElementById("content"))
```

至此，你大概能够了解 React SSR 的原理了。服务器编译后的组件更多的是偏于 HTML 模板，而具体事件和 vdom 操作需要依赖前端 bundle.js 做，即前端 hydrate 时需要做的事儿。可是，如果有多个组件，需要写入多次流呢？使用 renderToString 就简单很多，普通模板的方式，流却使得这种玩法变得很麻烦。

React SSR 里还有一个新增 API：renderToNodeStream，结合 Stream 也能实现 Bigpipe 一样的效果，而且可以有效的提高 TTFB 时间。

伪代码：

```
const stream1 =
renderToNodeStream(<MyPage/>);
const stream2 =
renderToNodeStream(<MyTab/>);
res.write(stream1)
res.write(stream2)
res.end()
```

如果每个 React 组件都用 renderToNodeStream 编译，并写入浏览器，那么流的优势就极其明显了，边读边写，都是内存操作，效率非常高。后端写入一个 React 组件，前端就 hydrate 绑定一下，如此循环往复，其做法和 Bigpipe 如出一辙。

Next.js同构开发

Node.js 成熟的标志是以 MEAN 架构开始替换 LAMP。在 MEAN 之后，很多关于同构的探索层出不穷，比如 Meteor，将同构进程的非常彻底，使用 JavaScript 搞定前后端，开创性的提出了 Realtime、Date on the Wire、Database Everywhere、Latency Compensation，零部署等特性，其核心还是围绕 Full Stack Reactivity 做的，这里不展开。简言之，当数据发生改变的时候，所有依赖该数据的地方自动发生相应的改变。本身这些概念是很牛的，参与的开发者也都很牛，但问题是过于超前了。熟悉 Node.js 又熟悉前端的人那时候还没那么多，所以前期开发是非常快的，但一旦遇到问题，调试和解决的成本高，过程是非常难受的。所以至今发布了 Meteor 1.8 也是不温不火的情况。

Next.js 是一个轻量级的 React 应用框架。这里需要强调一下，它不只是 React 服务端渲染框架。它几乎覆盖了 CSR 和 SSR 的绝大部分场景。Next.js 自己实现的路由，然后 react-loadable 进行按照路由进行代码分割，整体效果是非常不错的。Next.js 约定组件写法，在 React 组件上，增加静态的 getInitialProps 方法，用于 API 请求处理之用。这样做，相当于将 API 和渲染分开，API 获得的结果作为 props 传给 React 组件，可以说，这种设计确实很赞，可圈可点。

Nextjs 式的一键开启 CSR 和 SSR，比如下面这段代码。

```
import React from 'react'
import Link from 'next/link'
import 'isomorphic-unfetch'
export default class Index extends React.
Component {
  static async getInitialProps () {
    // eslint-disable-next-line no-undef
    const res = await fetch('https://api.
github.com/repos/zeit/next.js')
    const json = await res.json()
    return { stars: json.stargazers_count
  }
  render () {
    return (
      <div>
        <p>Next.js has {this.props.stars}
</p>
        <Link prefetch href='/preact'>
          <a>How about preact?</a>
        </Link>
      </div>
    )
  }
}
```

在 scr/pages/*.js 都是遵守文件名即 path 的做法，内部使用 react-router 封装。在执行过程中：

- loadGetInitialProps()，获得执行 getInitialProps 静态方法的返回值 props；
- 将 props 传给 src/pages/*.js 里标准 react 组件的 props。

优点：

1. 静态方法，不用创建对象即可直接执行；
2. 利用组建自身的 props 传值，与状态无关，简单方便；
3. SSR 和 CSR 代码是一份，便于维护。

Next.js 的做法成为行业最佳实践并不为过，通过简单的用法，可有效的提高首屏渲染，但对于复杂度较高的情况是很难覆盖的。毕竟页面里用到的 API 不会那么理想，后端支持力度也是有限的，另外前端自己组合 API 并不是每个团队都有这样的能力，那么要解此种情况就只有 2 个选择：1）在 SSR 里实现，2）自建 API 中间层。

自建 API 中间层是最好的方式，但如果不方便，集成在 SSR 里也是可以的。利用 Bigpipe 和 React 做好 SSR 组合，能够完成更强大的能力。限于篇幅，具体实践留在 QCon 全球软件开发大会（广州站）上分享吧。

3. 性能问题

用 SSR 最大的问题是场景区分，如果区分不好，还是非常容易有性能问题的。上面 5 种渲染方式里，预渲染里可以使用服务器端路由，此时无任何问题，就当普通的静态托管服务就好，如果在递进一点，你可以把它理解成是 Web 模板渲染。这里重点讲一下混搭法和纯 SSR。

混搭法通常只有简单请求，能玩的事情有限。一般是 Node.js 请求接口，然后渲染首屏，在正常情况性能很好的，TTFB 很好，整体 rt 也很短，使用于简单的场景。此时最怕 API 组装，如果是几个 API 组合在一起，然后在返回首屏，就会导致 rt 很长，性能下降的非常明显。当然，也可以解，你需要加缓存策略，减少不必要的网络请求，将结果放到 Redis 里。另外将一些个性化需求，比如千人千面的推荐放到页面中做懒加载。

如果是纯服务器渲染，那么要求会更加苛刻，有时 rt 有 10 几秒，甚至更长，此时要保证 QPS 还是有很大难度的。除了合并接口，对接口进行缓存，还能做的就是对页面模块进行分级处理，从布局，核心展示模块，以及其他模块。

除了上面这些业务方法外，剩下的就是 Node.js 自身的性能调优了。比如内存溢出，耗时函数定位等，cpu 采样等，推荐使用成熟的 alinode 和 node-clinic。毕竟 Node.js 专项性能调优模块过多，不如直接用这种套装方案。

4、未来

Node.js 在大前端布局里意义重大，除了基本构建和 Web 服务外，这里我还想讲 2 点。首先它打破了原有的前端边界，之前应用开发只分前端和 API 开发。但通过引入 Node.js 做 BFF 这样的 API Proxy 中间层，使 API 开发也成了前端的工作范围，让后端同学专注于开发 RPC 服务，很明显这样明确的分工是极好的。其次，在前端开发过程中，有很多问题不依赖服务器端是做不到的，比如场景的性能优化，在使用 React 后，导致 bundle 过大，首屏渲染时间过长，而且存在 SEO 问题，这时候使用 Node.js 做 SSR 就是非常好的。

当然，前端开发使用 Node.js 还是存在一些成本，要了解运维等技能，会略微复杂一些，不过也有解决方案，比如 Servlerless 就可以降级运维成本，又能完成前端开发。直白点讲，在已有 Node.js 拓展的边界内，降级运维成本，提高开发的灵活性，这一定会是一个大趋势。

未来，API Proxy 层和 SSR 都真正的落在 Servlerless，对于前端的演进会更上一层楼。向前是 SSR 渲染，先后是 API 包装，攻防兼备，提效利器，自然是趋势。

作者简介

狼叔（网名 i5ting），现为阿里巴巴前端技术专家，Node.js 技术布道者，Node 全栈公众号运营者，曾就职于去哪儿、新浪、网秦，做过前端、后端、数据分析，是一名全栈技术的实践者。目前负责 BU 的 Node.js 和基础框架开发，即将出版 Node.js《狼书》3 卷。

从观望到落地：新浪微博 Service Mesh 自研实践全过程

作者 周 晶

2017 年 4 月 25 日 Buoyant 公 司 的 CEO William Morgan 给 Service Mesh 下了个完整的定义，简言之，“Service Mesh 是一个专用的基础设施层，用于使服务到服务的通信安全、快速和可靠……”

其实更确切的说 Service Mesh 应该诞生于 2016 年，因为回看很多 Service Mesh 方案早在 2016 年就已经开始摸索前行。随后的几年如雨后春笋般出现了很多实现，逐渐形成了其基本事实标准：负责服务间通信的数据面，以及控制整体流量在体系内流转的控制面。Service Mesh 可谓是当下微服务领域最火的模式。被认为是下一代的微服务。甚至 2018 年被称为 Service Mesh 元年。

本文，我试图从自己实践 Service Mesh 过程中的一些感悟出发去探寻 Service Mesh 的本质，希望能给正在关注 Service Mesh 或者计划实践、落地 Service Mesh 的你带来一些我的理解，毕竟大家已经观望 2 年多，也该考虑落地了。

Service Mesh 到底是什么?

为什么要讨论“Service Mesh 到底是什么？”这样一个话题，因为我发现一个有趣的现象，现在大家谈论 Service Mesh 时，几乎就把它与 Istio 画了等号，尤其是越往后关注这个技术方向的人这种情况越明显。

从另一个角度来看，这也算一种技术垄断，大公司背书的 Istio 经过这两年的发展与其专业的开源运营，几乎掩盖了所有其它方案的声音，让新进的开发者误以为 Service Mesh 就是 Istio。

这会带来一个直接的问题：Istio 并不一定适

应所有的场景。我觉得 Mesh 社区一个同学说的一句话特别有道理“垄断使得包括其在内的竞品并未通过足够长时间的竞争达到相对的成熟，就过早形成了某种心理意义上的垄断预期”。导致当下可能很多团队可能在自己的场景强推 istio，忽视了自己真正的痛点或者假装自己有 istio 预设那样的痛点，如此复杂的 istio 体系如果方向选不对，折腾到最后可能吃力不讨好。

那么 Service Mesh 究竟是什么？

Service Mesh 并不是什么新的技术，它所关注的问题域比如请求的高效、可靠传输，服务发现和治理等有服务化的一天就已经存在。成体系的服务治理方案我们早前很多 SOA、API 网关等就早有深入并已形成标准方法论和行业规范。社区也不乏这方面的最佳实践。那为什么到如今才催生出如此火爆的 Service Mesh？这一切的发生我认为关键点在于两个方面：

- 首先微服务架构模式成为大规模分布式应用的主流架构被大家所熟知与认可；
- 其次云计算、容器化技术的规模化落地。越来越多的团队和组织开始实践微服务。这个时机很重要。

大家很清楚，微服务架构的核心在于基于分而治之理念的服务拆解，而拆解往往使原先可能一次内部方法调用就获取的结果，转换为一次或多次网络调用，我们都知道网络本身是不可靠的（这也正是为何 Service Mesh 解决的核心问题域中始终将请求的可靠传输放在至关重要位置的原因），下面我们简单梳理服务的拆解给我们带来的挑战：

- 拆解后，服务间依赖不可靠的网络进行通信；
- 缺少一套整体的方案来解决所谓东西向（横向）服务连通性的问题；
- 如何保证拆解后整体业务的稳定性（一次业务处理可能依赖数目不定的微服务调用）；
- 如何来治理这些拆解后的服务（甚至是异构的微服务体系）；
- 如何规模化、标准化更高效的实施、运维整个微服务体系。

过去通常使用微服务治理框架、API 网关等方案来解决上面的问题。比如微博早在 2013 年就开始基于 Motan RPC 框架来解决服务治理的问题，而更早的 API 网关方案也具备相应的服务发现和治理能力甚至比微服务框架出现的还要早。传统微服务框架通过胖客户端的方式提供了客户端实现的服务发现与治理能力。但是这种方式有诸多弊端：

- 与业务无关的服务治理逻辑与业务代码强耦合；
- 框架、SDK 的升级与业务代码强绑定；
- 多语言的胖客户端支持起来性价比极地；
- 各种语言的服务治理能力天差地别，服务质量体系难以统一。

而 API 网关通常部署在集群的边缘，作为业务接入层，虽具备一些通用的服务治理能力，然而其本身的实现相对更重、且往往性能低下，因为它专注于 API 管理，流经 API 网关的流量都会经过很多通用逻辑，比如鉴权、验证、路由等，在传统的 API 网关场景，长达几十甚至过百毫秒的延迟对南北向的流量来说基本可以接受，但这在处理微服务体系内东西向流量时，这样的性能是不能容忍的，而且服务的拆分势必导致整个体系内东西向流量随着微服务规模的扩大而剧增。所以 API 网关并不能胜任微服务场景下的流量管理任务，它也不是为此而生的。

再补充一点很多人对 Service Mesh 和 API 网关分不清楚，因为正好这两种方案我都经历过，其实很简单，它们关注的点不一样，Mesh 关注内部服务间的互联互通，核心在于 Sidecar 构建的那张网格以及对应流量及服务的治理，而 API

网关关注的是 API 的管理，虽然它们都有服务治理的能力，但彼此专注的点不一样。

所以本质上 Service Mesh 的出现就是为了解决大规模微服务架构下，请求的高效可靠传输与服务治理的问题。它将原先实现在微服务框架胖客户端或者 API 网关接入层的服务治理功能下沉到一个统一的 Sidecar 代理以一个独立进程的方式部署在业务进程旁边，通过控制面来保障服务间通信的横向流量按照业务的真实需求在整个微服务体系内高效流转，因此 Service Mesh 不是服务治理框架，不是 API 网关更不是 Istio。它是一种新兴的微服务实施模式。一种微服务治理的标准规范。它有很多种实现，Istio 是目前最为大家所熟知的实现。

不过下面我们要看的是微博自研的实现 -- WeiboMesh。

WeiboMesh 的理想与现实

了解过 WeiboMesh 的同学可能知道，它基于微博早期服务治理 RPC 框架 Motan 演变而来，（没了解过的同学可以 Google“微博 Service Mesh”，这里不再赘述）。微博平台的微服务体系建设做得比较早，平台内部技术主要是 Java 栈，基于服务治理框架和混合云有一整套服务化体系。同时平台还给其它兄弟团队提供底层数据与存储资源支撑。数据通过 Restful 接口提供。是一个典型的异构体系服务化整合的场景。随着平台微服务规模的增大以及业务复杂度的提高，我们必须面对以下三个难题：

- 热点、突发事件所引发流量洪峰（几倍、十多倍的突增）的应对（要求高效、完备的服务治理体系）；
- 混合云微服务架构下的复杂拓扑带来的冗长的调用链路、低效的请求长尾与及其复杂的故障排查；
- 各语言服务治理能力差异性引入的异构系统服务治理体系建设难题。

为了应对这些难题，我们摸索了一套 Service Mesh 实现方案。这里我先对整个方案的几个核心点做个简单描述（希望大家能从微博的场景中体会这种方案选取的出发点）。

- 将以往在 Motan RPC 胖客户端实现的服务发现和治理功能下沉到 Motan-Agent 这个 Sidecar 代理上，通过对服务的订阅和发现来建立服务间依赖的逻辑网络，作为 WeiboMesh 的数据面。
- 复用 Motan RPC 的 Filter 机制，在 Cluster（服务集群）和 Endpoint（节点）两个维度实现了 Service Mesh 的控制面逻辑，完成对服务间通信流量的管控，作为 WeiboMesh 的控制面。
- 服务注册和发现依赖微博故有的自研 Vintage 命名和配置服务。WeiboMesh 不与任何平台耦合，可运行在裸机、公 / 私有云、混合云的场景。
- 实现全新语言无关的通信协议 Motan2 和跨语言友好的数据序列化协议 Simple 来应对跨语言。通过为所支持的语言开发标准的 Motan2 客户端，完成与 Mesh 的通信（对业务较低的侵入性带来平台能力的整体输出，比如我们提供的各种数据服务化、存储资源服务化能力。这个方案的性价比极高，保证整体架构向前演进而非推倒重来，这在微博大体量的业务需求下显得格外重要，而且我们跨语言的 SDK 只保留极薄的一层，仅保证使用 Simple 序列化在 Motan2 协议上传输请求而已，这很好的控制了整体维护成本）。

这是我们探索了各种跨语言服务化方式后，最终第一版上线时的方案（之前的文章有对此详尽的描述，请自行 Google，同样不再赘述），也是一个迄今为止我们依然认为比较理想（符合微博现状）的 Service Mesh 方案。最大程度上复用了我们多年来在 Java 体系以及混合云架构下积累的服务化经验。最大化的保证了平台技术体系的稳固、高效同时兼顾了极低的接入、升级、维护成本。

为什么说理想很丰满，现实很骨感。整个WeiboMesh项目组最初由服务框架组与Feed组主导协同多个业务方共建（立项时叫跨语言服务化小组），参与方涉及五、六个团队，且参与各方都对跨语言服务化有重度需求，一点不夸张的讲已经是痛不欲生苦不堪言才逼到这条路上来（经常为排查一个线上问题，要追踪无数跳的链路，打日志，加监控，费时费力效果还不理想；四七层的转发也增大了请求时延；因为请求双方服务池的不一致导致跨云转发，请求长尾严重等）。

所以最初参与的各方，状态和心境都非常积极，大家很主动的推进整个方案演进、调研、试错。服务端改造过GRPC方案、Yar方案、Servlet转Motan2的方案等等，客户端也改造过很多方案，比如服务在Client端的描述、Client的构造方式、请求的组织形式、各种兜底方案等等。所以第一版上线时，我们其实已经经历了很多尝试与磨合。算是比较成功的，我们原型版早在16年底就初步线上试水，17年已经直接抵御春晚的高峰。

但是往下想把这种理想的方案铺开来时，现实开始向我们展现出它的骨感。可谓一步一坎：

- 如何说服项目组外其他各方在新业务上应用Mesh方案；
- 如何将老业务接入Mesh体系，现有方案Server、Client两端都没做到完全零侵入；
- Agent引入的运维复杂度和权责划分。

我们除了需要消除大家对Mesh性能和稳定性的顾虑，还要保证方案足够简单，避免对已有的运维体系，服务开发、部署模式造成太大的冲击，同时要让大家认可Mesh改造的收益，用数据和效果说话。

虽然有搜索、微博主站Page、热门、话题这样的一些重量级典型业务方给我们的方案背书，从以往的Restful方案接入Mesh后，享受到性能提升的同时也收获了平台服务治理体系的红利，因为有些业务方专注于自己的业务领域，很正常在服务治理及相关体系建设方面会相对薄弱。

对于业务方接入我们的方案已经足够简单，只需要多部署一个Sidecar代理，将以往的Restful调用迁移到我们的跨语言SDK上即可，业务方不需要关注Sidecar的任何事情，因为当Sidecar出问题时，我们可以退化为Restful调用，并立即报警。

同时我们Mesh体系提供了缓存、队列等服务化资源，通过SDK既可以轻松访问，业务方可以由此对自己依赖的服务和资源都有整体治理的能力。对于SDK接入这样的改造，显然性价比是极高的。然而现实情况是，很多时候可能因为业务开发压力等原因，他们并没有太多的精力来完成这样的改造。

而另一方面，平台内部已有的大量老业务如何接入Mesh体系？现今平台业务已经处于一个相对稳定的阶段，老业务才是改造的重中之重，我们必须在Mesh体系为业务方准备好他们依赖的服务。但是Mesh方案能为平台服务方提供的唯一好处可能就是接入后，他们不再需要同时维护RPC和Restful两套服务，但是对于本来就已经历经风雨稳定运行的平台服务来说，已有的Restful服务没什么不好，为什么要去改造？显然大家是不乐意的，虽然我们已经在Motan RPC框架层面做了诸多努力，甚至做到只需修改一行配置，多导出一个Motan2服务即可，哪有如何？服务端的变更要求非常严谨，一行配置同样需要经过整个上线流程。婉拒的话术通常是诸如“目前业务比较忙，往后有空再具体规划”此类。

这迫使我们一直在思考一个问题“如果要让xxx接入Mesh，需要xxx做什么？我们需要支持什么？”，这也促使WeiboMesh迎来了全新的一次演进 -- HTTPMesh。HTTPMesh的目标在于，不需要修改任何一行代码，真正零侵入接入，具体做法：

- 将现有Restful接口自动导入Mesh体系；
- 通过HTTP_PROXY将客户端流量指向Sidrcax。

我们通过解析平台 Restful 服务对应的 Nginx 配置，将 Restful 接口自动对应到 Mesh 体系的 RPC 服务上，Nginx 的 upstream 与 RPC 的 Service 对应，Restful 服务池与 PRC 服务分组对应，这样一来原来的 Restful 服务在 Mesh 体系就有了相应的表示。新增业务只需提供一套 Motan2 服务，而非以往的 RPC 和 Restful 两套。老业务的场景，服务端 Agent 充当反向 HTTP 代理，将进入的 RPC 请求转换为 HTTP 请求转发到之前的 Restful 接口；客户端我们在 Sidecar 代理上提供了 HTTP 的正向代理支持，通过 HTTP_PROXY 将本地出口的 HTTP 请求都指向 Sidecar，这样如果 Sidecar 发现请求已经导入 Mesh 体系则会将代理的 HTTP 请求转换为 RPC 请求转发进入 Mesh 网格，否则将 HTTP 请求透传发出；而新开发的采用了 SDK 方案的服务可以同时享有平台所有资源、服务及治理体系。这样双方接入都不需要做任何改动。

解决了双方改造成本的大问题，接下来就是引入 Sidecar 对运维流程的影响和权责划分的问题。在业务进程旁运行 Sidecar 来处理进出流量，需要在固有的运维流程中添加对 Sidecar 的运维，这看似简单的操作，实则涉及多方面的问题：

- Mesh 进程与业务进程的启停（时序、优雅停机）；
- Mesh 进程的升级维护；
- Mesh 进程的监控、报警；
- 故障处理与恢复。

运维方面可能各公司情况不一，包括我们内部，不同团队可能姿势都不完全一样，不具有通用性，所以这里不展开太多，只想说明一点我们为了应对各种场景和姿势，我们在 Mesh 上提供了管理端口、后端探测等功能，比如运维可以通过向 Mesh 管理端口发起上下线请求来触发 Mesh 自动上下线服务，也可以通过配置探测地址及状态来对业务进程进行健康检查指导该节点上下线或告警，另外我们提供了多种发布方式，镜像、RPM 包等一应俱全，便于运维挑选适合自己生产的部署方式，我们还将 Mesh 配置做了统一的管理，因为以往接入的经验来看，80% 场景使用通用配置即可，只有很少的 20%，甚至更少的场景需要个性化配置，那样也减少大家理解和我们作为方案提供方的指导成本。所有 Mesh 相关的包、库和配置都由我们提供，这样也有了明晰的权责划分，我们为 Mesh 负责，降低业务方的心智负担。

就这样，我们在 WeiboMesh 的路上继续前行。因为我们深知微服务、云计算才是稳定服务的正道。确实给我们带来了很多现实收益，而且我们相信这些收益仍在不断放大。

如何落地 Service Mesh

前面我们探讨了 Service Mesh 的本质，分享了一些 WeiboMesh 实践过程中的经验，下面我们来讨论下如何落地 Service Mesh。

Service Mesh 很好的解决了规模化微服务架构下服务通信和治理的问题，给我们带来了很多确确实实的好处，网络上相关的解读很多了我就不再次啰嗦。这里我只想根据自己的实践和理解，梳理出一些我认为很适合引入 Service Mesh 的场景，希望能帮助大家更好的做出判断：

- 还未进行任何形式服务化；有的初创团队可能先期并没有太多精力来进行技术建设，但是技术的稳定性是业务增长的基石，Service Mesh 的整套的微服务治理方案正好是个不错的选择，先期可能不需要对服务做太细粒度的拆分，可以基于 Mesh 体系来构建，逐步接入、分解；
- 有跨语言互通的需求；这不用多说，前面分享的 WeiboMesh 就是这方面的实践的好例子；
- 内部服务依赖重（东西向流量）；有些业务可能同时依赖很多服务，一次请求对应数次 Restful 请求，有繁重的网络 I/O 需要处理，可以引入 Service Mesh 来保障这些请求的可靠性；

- 依赖传统的服务化框架做服务治理；这里主要指服务治理与业务逻辑强耦合的场景，前面也讨论过这种方案的各种弊端，引入 Service Mesh 将服务治理逻辑与业务逻辑解耦，应用开发的同学只需专注自己的业务，这是最好的选择；
- 技术建设与前瞻性；云计算与微服务为提高服务构建效率、降低投入成本提供了可能性，这里面的想象空间是巨大的，从大厂在此方面的投入可见一斑，俗话说的好，钱在哪儿方向就在哪儿。具有前瞻性的技术储备将为组织迎接各种挑战提供坚实后盾。

想清楚我们确实需要落地 Service Mesh，认可其带来的巨大收益，剩下的就是 Service Mesh 方案的选择了。我们需要制定一个可执行的 Service Mesh 实施方案。还是那个老生常谈的话题：“选择自研、开源还是基于开源微创新？”。在这方面我只有一条建议：“因地制宜，不盲目跟风”。选择的过程中，头脑一定要清晰，仔细分析自己的场景和业务痛点，调研与自己场景最贴合的开源方案，过程中掂量自己可支配的资源以及项目周期同时关注方案复杂度。尽量避免重复造轮子。

不过事实上开源方案一般考虑比较通用的场景，很难有能与自己场景完全吻合。这时候为了控制成本更多会选择基于开源进行改造。这就引入了一个与开源共建的问题。应该避免独立 fork 导致后期离开源越走越远。

Service Mesh 从出现到现在两年多的时间里可谓发展迅猛，这离不开社区大家的热情，抛开大厂在前面带风向不说，技术本身确实有很多先进性，不管是否入坑 Service Mesh，作为一个工程师我觉得都应该深入了解，学习这种 Mesh 思考模式和工程化、标准化思路。希望本文能给你带来一些思考。

作者简介

周晶，新浪微博平台研发技术专家，负责微博跨语言微服化框架开发以及 ServiceMesh 技术落地等，OpenResty 社区委员会成员，高性能 OpenResty 开发框架 Vanilla 作者，开源爱好者，关注微服务、云原生等技术。

OPPO 基于 Flink 构建实时计算平台的思路、演进与优化

作者 张 俊

在互联网越来越快的今天，用户的“耐性”正在变差，企业对数据服务实时化的需求也日益增多，打车、外卖、网购、在线视频等场景下，用户已经不能忍受较长时间的等待，企业对于大数据实时决策的要求也越来越严苛。

在这样的背景下，OPPO 基于 Flink 打造了实时计算平台 OStream，对 Flink 进行了系列的改进和优化，探索了实时流计算的行业实践以及变化趋势。为此，OPPO 大数据平台研发负责人接受了 InfoQ 的专访，深度解析了实时流计算的行业实践以及变化趋势。

InfoQ：目前 OPPO 有在什么样的场景中使用 Flink？（如果方便透露规模的话也可以具体介绍下）你们是什么时候开始使用 Flink 的？当时为什么要选择 Flink？

张俊：OPPO 的互联网服务拥有 2.5 亿的全球活跃用户，涵盖了应用商店、信息流、搜索等各类业务。2017 年开始，陆续有业务开始尝试实时计算，主要是基于 Storm 和 Spark Streaming。2018 年开始逐步尝试转向 Flink，核心有两点考虑：第一，相对 Spark Streaming 来说，Flink 是原生的流处理引擎，能带来更低的处理延迟；第二，相对 Storm 来说，Flink 具备内置的状态管理与更低成本的容错机制。

目前，Flink 已经应用在实时 ETL、实时报表、实时标签等场景，目前广泛服务于浏览器、信息流、应用商店等业务，集群规模达到 200 台机器。到下半年，预计集群规模会超过 500 台，推荐、

搜索、广告等业务也都在规划接入。

InfoQ：能否整体介绍下你们大数据平台的架构？基于 Flink 构建的实时计算平台主要做了哪些工作？

张俊：OPPO 大部分的数据来源是手机终端，因此我们基于 NiFi 构建了数据接入系统，将数据同时落地 HDFS 与 Kafka，分别应用于离线与实时场景：

离线方面，基于 Hive 构建了一整套数仓体系，跑批任务通过 Airflow 来统一调度，报表、多维分析主要利用 Kylin 来加速，即席查询由 Presto 来承载；

实时方面，构建了以"Kafka->Flink->Kafka->Druid/Elasticsearch"为核心的实时数据流，Druid 主要用于实时报表，Elasticsearch 用于实时标签。

目前，自建的实时计算平台 OStream 主要基于 Flink SQL 来构建，提供一站式的 WEB 控制台来创建库表、编辑并提交 SQL、查看作业日志、展示作业指标、设置告警规则。未来，平台还将支持画布拖拽与 JAR 包提交两种使用模式，进一步满足不同用户人群的诉求。

InfoQ：在使用 Flink 构建实时计算平台的过程中遇到过哪些难点？是怎么解决的？你们做了哪些定制化的改造与增强？（出于什么原因改造、如何优化、优化后的效果）

张俊：主要遇到有三个难点，社区都有相关的讨论，但官方目前还没有成熟的解决方案：

第一，如何与外部元数据打通。我们的离线数仓元数据（包括库表与 UDF）由自研的元数据中心管理，信息统一存储在 MySQL 上，实时元数据也希望能统一起来维护。Flink SQL API 虽然可以从 ExternalCatalog 来搜索外部表定义，但内置只有基于 memory 的实现。我们通过扩展 ExternalCatalog，从 MySQL 查询外部表，实现了与元数据中心的打通。

第二，如何提交与管理 SQL 作业。目前，Flink SQL 只支持嵌入代码中来提交，REST-based SQL gateway 一直是社区的热门需求。Uber 开源的 AthenaX 填补了这个空白，实现了以 SQL REST 的方式来提交与作业管理。但是，由于原生的 AthenaX 以及关键模块的缺失，根本无法运行起来，所以我们扩展了它的 TableCatalog 和 JobStore，实现了完整的元数据与作业管理，同时进行大量功能上的优化与修复，使其达到生产可用的状态。

第三，如何实现数据流与维表 join。实际流处理场景中，实时数据经常需要关联 MySQL/Hive 维表，比如根据用户 ID 获取特征标签。我们利用 Flink 对 stream-table 二元性的支持，在 SQL 编译阶段进行改写，将维表 join 转换为中间 stream，从而自动加载维表数据到内存来与中间 stream 关联。这样的实现方式很轻量，无需改动 Flink 内核代码。

InfoQ：OPPO 流计算技术的演进过程是怎样的？反过头来看，Flink 是否能够完美支撑平台迈向智能化的需求？未来你们会有什么样的迭代计划？

张俊：OPPO 流计算发展较晚，因此反而没有太多的历史包袱，使得我们可以更顺利地转向 Flink 这样的先进框架。当前，OPPO 流计算的应用主要偏向实时数仓方向，推荐实时化以及监控实时化正在萌芽中，未来会有更多的发展。

所谓平台智能化，我的理解是不断加深自动化程度，最小化人工的使用与维护成本。这个仅依靠 Flink 单体系统是无法实现的，需要生态的协作与合力。Flink 目前的生态发展势头不错，已经与越来越多的系统打通，自身框架的架构也愈加完善，能够支撑智能化场景的探索与演进。未来，我们计划从两个层面去尝试：在使用层面，自动报表、标签以及 ETL 的生成；维护层面，智能配置优化、自动系统告警等。

InfoQ：取之于开源，用之于开源，您个人或您所在的团队为 Flink 社区做过哪些贡献？开

发者如何参与到 Flink 社区贡献中来?

张俊：目前所做的贡献可以参考 FLINK-9384、FLINK-9444、FLINK-10061、FLINK-10079、FLINK-10170 这几个 issue。

学习 Flink 的初期一直有关注社区动向，但真正提交贡献是源于研发过程中偶然遇到的问题。因为属于基本的功能点，当时不太敢相信是 Flink 内核 bug，更多地以为是使用方式不当。后来，反复确认了 API 文档，还写了额外的示例都重现了问题，才深入源码去探究，最终找到和修复了 bug。现在总结下来，关于参与开源社区有几点感悟：

平时多关注 mailing list 与 github pull request，那里有很多问题和话题的探讨，可以提供方向性地指引。当然，真正的切入点还是源于深入应用中遇到的问题。

平时使用过程中，不要仅仅停留在 API 层面，可以通过某个 API 方法为切入点深入 Runtime，刚开始不用陷入细节，跟踪主体流程就好，必要时把 code path 记录一下，真正遇上问题时可以快速地定位。

发现问题，要大胆提出，社区多数人都很 nice，会耐心地给出建议与想法，即便最终没有 merge，探讨的过程也是一种学习与积累。当然，提交 PR 时有一些规范还是要注意，比如 PR 问题描述要尽量详细与清晰，问题修复要有单元测试佐证，确保修改后全局单元测试都能通过。

InfoQ：之前 DA（Flink 创始公司）的 CTO 发表演讲称，当前存在的一个趋势是，Flink 将朝着批流融合计算方面发展。您怎样看这样的发展趋势？这将对 Flink 的应用带来哪些影响?

张俊：Flink 或 Spark 这样的平台型框架所带来的核心优势，就是在 API 层面统一各个分布式计算场景。当前，虽然 Flink 在流计算领域所向披靡，但离线批处理领域仍然是 Hive 和 Spark 的天下。我认为最主要的原因是 Runtime 能力和生态没有跟上，前不久我们有尝试用 Flink 做批计算，但与 Hive 元数据打通这样最基本的特性都有缺失，前方死路一条。当然，我很期待 Flink 在 Runtime 层面的批流融合方向。数据开发者能够共同维护同一套代码，平台开发维护统一的计算引擎，相信是大家喜闻乐见的。

作者简介

张俊，OPPO 大数据平台研发负责人，主导了 OPPO 涵盖“数据接入 - 数据治理 - 数据开发 - 数据应用”全链路的数据中台建设。2011 年硕士毕业于上海交通大学，曾先后工作于摩根士丹利、腾讯，具有丰富的数据系统研发经验，目前重点关注数仓建设、实时计算、OLAP 查询方向，同时也是 Flink 开源社区贡献者。

滴滴基于 ElasticSearch 的一站式搜索中台实践

作者 张 亮

ElasticSearch 是基于 Lucene 实现的分布式搜索引擎，提供了海量数据实时检索和分析能力。滴滴从 2016 年 4 月开始组建团队，解决 ElasticSearch 在使用过程中遇到的性能问题。并且，随着业务体量的发展，滴滴构建了基于 ElasticSearch 的一站式搜索平台。

ElasticSearch 在滴滴的应用场景

滴滴自 2016 年 4 月开始组建团队，解决 ElasticSearch 在使用过程中遇到的性能问题。搜索平台的建设是随着业务体量的发展逐步演进的，如今已经发展到有超过 3500+ ElasticSearch 实例， 5PB 的数据存储，峰值写入 TPS 超过了 2000W/S 的超大规模，每天近 10 亿次检索查询。

ElasticSearch 在滴滴有着非常丰富的应用场景：

- 为线上核心搜索业务提供引擎支持；
- 作为 RDS 从库，海量数据检索需求；
- 解决公司海量日志检索问题；
- 为安全场景提供数据分析能力。

不同场景业务方对写入的及时性、查询的 RT、整体稳定性的要求都是不一样的，我们对平

台提供的服务抽象为索引模板服务，用户可以自助开通相应的服务。

我们内部经过压测、线上调优以及引擎的一些优化，已经将最佳实践，沉淀到标准的 Docker 镜像中，个性化的需求都在索引模板的服务级别进行设置与管控，部分优化如下：

- 优化master任务处理流程的耗时环节，处理效率提高5倍
- 优化es search流程在索引膨胀时的模糊索引查询对节点整体的影响
- 优化es单节点挂载多磁盘时，索引shard的均匀分配
- 修复master更新元数据超时导致的内存泄露异常
- 修复es在source为false时的响应结果返回null的bug
- 离线导入优化，单副本，关闭translog，导入完成后添加副本，CPU节约60%
- 去掉_all索引，存储空间节约30%
- 写入时使用es自动生成的id，减少get version环节，写入性能提升50%
- 结构化索引的字符串，限制索引长度，储存空间大约节约10%
- 去掉ttl的过期方式，改为删除过期索引，CPU节约50%
- 只索引不存储的字段优化，为外卖自配送场景，存储空间节约70%，查询性能提升3倍
- clientnode层面增加es内部各阶段、各shard的查询、写入耗时
- ……

平台稳定性面临的风险与挑战

超大的集群规模和丰富的场景给滴滴 ElasticSearch 平台带来了极大的风险与挑战。主有以下几个方面：

线上业务场景

- 稳定性要求至少 99.99%，对查询的 90 分位性能抖动敏感；
- 架构层面需要支持多活的需求，对数据的一致性与及时性都有要求，必须保证数据的最终一致性，数据更新秒级可见；
- 不同线上业务，插件需求、索引分片规则都是多样化的；
- 众多独立集群如何快速平滑地进行滚动升级，保障的线上业务无影响。

准线上业务场景

- 离线快速导入时效性要求分钟级，实时导入 10 亿条数据需要 5 个小时，导入时在线资源消耗严重，线上服务基本不可用，导入成本消耗过大；
- 查询的多样性，14W+ 查询模板，单索引最高有 100+ 应用同时查询，在多租户场景下，如何保证查询的稳定性。

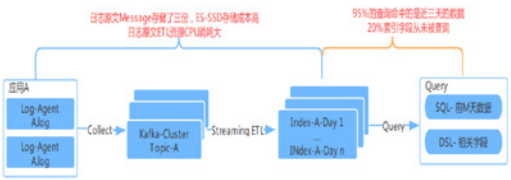
安全与日志场景

- 千万级别数据每秒的实时写入，PB 级日志数据的存储，对大规模 ElasticSearch 的集群提出诉求，但 ElasticSearch 有自己的元信息瓶颈；
- 查询场景不固定，单个索引几百亿级别的数据体量，需要保障不合理查询对集群与索引的稳定性风险可控；
- PB 级存储，查询频率低，但查询的时效性要求 S 级别返回，全部基于 SSD 盘，成本太高，需要在查询体验没有太大变化的情况下，降低整体的存储成本。

那么，如何解决这些问题呢？

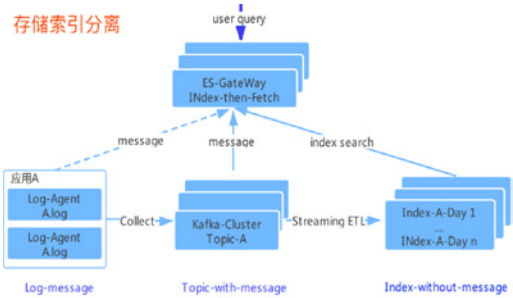
如何打造“存储成本低”的搜索中台

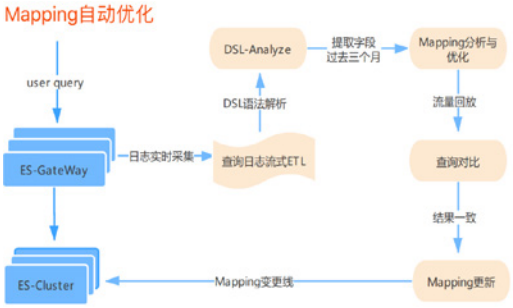
目前，在日志与安全分析场景下，存储成本压力很大，属于典型的“写多查少”的场景，我们对存储成本的耗散点进行了深入的分析，整体情况如下：



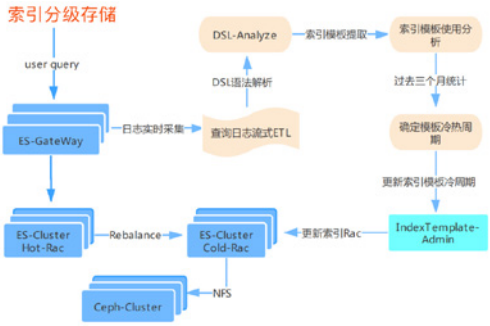
针对资源耗散点，我们在架构层面进行了优化，整体成本降低了 30%，累积节省了 2PB 的存储，分别从以下几个方面进行了优化

- 存储索引分离：日志原文与索引进行分开存储。
- 不合理的索引字段 Mapping 自动优化。

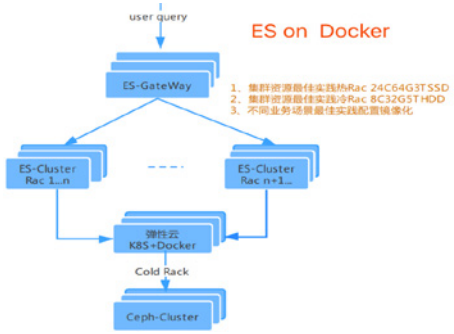




- 冷热数据进行了分级存储



- ES On Docker&Ceph 改造



未来发展规划

基于 ElasticSearch 的搜索中台给用户带来的收益

- 服务了超过 1200+ 平台业务方，其中 20+ 线上 P0 级应用，200+ 准实时应用；
- 索引服务接入效率从原来的两周降低到 5 分钟；
- 服务稳定性有保障：线上场景 99.99%，日志场景 99.95%；
- 高频运维操作一键自助完成，90% 的问题，5 分钟完成定位；
- 整体存储成本是业内云厂商的 1/3。

不足点

- 目前滴滴 90% 的集群还是在 ElasticSearch 2.3.3 版本，内部修复的 BUG 与优化，无法跟社区进行同步；
- 目前通过 ES-GateWay 的方式支持了多集群方案很好的满足了业务发展的需求，但是集群变多之后的，版本维护与升级、整体资源利用率提升、容量规划都变得非常艰难。

发展规划

解架构之“熵”

- 突破引擎元数据瓶颈，提升运维效率，降低成本 ->ES - Federation;
- GateWay 能力插件式下沉引擎，减少中间环节，与社区融合，优化性能。

提引擎迭代效率

- 100 个节点集群滚动重启时长从 2 天提升至 1 小时；
- 架构层面解决跨大版本升级之痛”2.2.3 -> 6.6.1 http restful。

聚焦价值问题

- 多租户查询、CBO、RBO 的查询优化器建设;
- 数据体系化 -> 数据智能化;
- 基于 Ceph、Docker 改造 ElasticSeach，支持 Cloud Native 的存储计算分离。

作者简介

张亮，滴滴出行高级专家工程师。曾任华为南研所网盘研发工程师；2014 年 4 月至今任职滴滴出行大数据架构部高级专家工程师。在滴滴任职 5 年，经历从无到有组建团队，主持构建过任务调度系统、业务监控系统、链路跟踪与诊断系统、数据同步中心等架构设计与研发工作，目前在负责数据通道、kafka 服务、数据检索的引擎建设工作，具有丰富的高并发、高吞吐场景的架构设计与研发经验。

Geekbang. 极客邦科技 | TGO 鲲鹏會

TGO 鲲鹏会是极客邦科技旗下高端科技领导者聚集和交流的组织，以 CTO、CPO、COO、技术 VP 等科技领导者为服务对象，采用实名付费会员制，严格审核会员资格，旨在组建全球最具影响力的科技领导者社交网络，线上线下相结合，联结杰出的科技领导者学习和成长。

12 个城市成立分会

会员超 850 人

使命 Mission

为社会输送更多优秀的科技领导者

愿景 Vision

构建全球领先的有技术背景优秀人才的学习成长平台

扫描二维码，了解更多内容