



架构师

ARCHITECT

ArchSummit 特刊



全球人工智能与机器学习技术大会

助力人工智能落地

2018.01.13 – 01.14 · 北京国际会议中心

人工智能已不再停留在大家的想象之中，各路大牛也都纷纷抓住这波风口，投入AI创业大潮。那么，2017年，到底都有哪些AI落地案例呢？机器学习、深度学习、NLP、图像识别等技术又该如何用来解决业务问题？

由InfoQ举办的全球人工智能与机器学习技术大会上，一些大牛将首次分享AI在金融、教育、电商、外卖、搜索推荐、人脸识别、自动驾驶、语音交互等领域的最新落地案例，以及在落地过程中的那些痛点和难点，一些技术细节该如何操作，有哪些避坑经验，应该能学到不少东西。

部分演讲嘉宾



颜水成
360人工智能研究院
院长及首席科学家



山世光
中科院智能信息处理
重点实验室常务副主任
中科视拓董事长/CTO



袁进辉（老师木）
一流科技
创始人



刘海峰
京东商城
总架构师&技术VP



洪亮劫
Etsy
数据科学主管



于磊
携程
基础大数据产品团队总监



张浩
饿了么
技术副总裁



尹大朏
摩拜单车
首席科学家

精彩案例抢先看

摩拜 | 如何使用人工智能实现单车精细化运营

知乎 | 如何使用机器学习实现News Feed正向
交互率提升100%

国美 | 推荐引擎与算法持续部署实践

微博 | 深度学习在红豆Live直播推荐系统中的应用

Tutorabc | 大数据和AI之路

饿了么 | 机器学习和运筹优化在外卖行业的应用实践

第四范式 | 如何利用大规模机器学习技术解决
问题并创造价值

微信小程序 | 商业智能技术应用实践

爱奇艺 | 自然语言处理和视频大数据分析应用

8折 限时优惠进行中，每张立减720元
截至2017年12月15日前 团购享受更优惠

邮箱: hedy.hu@geekbang.org 电话: 18510377288 (同微信)



购票请联系



扫码关注大会官网
获取更多大会信息

卷首语：聊聊超能力

ArchSummit组委会

有人说：编程就是我们这个世界的超能力。

私以为非常同意这个观点，我们幼时幻想是否能瞬间凭空制造一些工具来满足我们的需求，如今，借助互联网这个虚拟世界媒介，与目前软硬件日益发达的能力和早已普遍的云计算，我们现在的需求能够不断被满足，乃至受限于编程的想象力。

但这次，我们想聊聊编程之上的一种超能力，而它应该是我们架构师所必备的：“连接”。

乔布斯在斯坦福大学的演讲三故事之一，聊的就是连接的故事：“...The first story is about connecting the dots...”

回顾我们架构师的知识从何而来？有来自自身的独立探索，不断 review 公司、团队和自己写的代码并从中获得源源不断的灵感；也有来自外部的交流，大家互相借鉴并默默记住前沿应用中目瞪口呆的经验感悟。

看上去架构师的知识是线性的，在探索和讨论中不断沉淀经典且耐用的经验，继而反复加入需要验证且有待落地的前沿知识。

简而言之，对于架构师而言，他不仅需要连接架构的过去与未来，而且需要连接优秀的沉淀内容与出色的启蒙思想。先别着急嗤笑，这本特刊和乔布斯一样，聊的也是连接的故事。

在这场 ArchSummit 全球架构师峰会北京站开幕之前，我们采访了不少前来分享的讲师，他们早已是各自企业的技术核心，我们挖掘了他们背后的一些故事，看看是否能让你注意到处处连接的力量。

我们尝试以 7 月份 ArchSummit 深圳站的精彩演讲稿为引，连接我们现在北京站讲师的预热采访。目录如下，各位前来演讲的嘉宾，可在大会日程里了解他们现场的演讲时间及地点，并和他们进一步交流。

InfoQ

在微信上关注我们



InfoQ

国内最好的原创技术社区，一线互联网公司核心技术人员提供优质内容。订阅 InfoQ，看全球互联网技术最佳实践。做技术的不会没听过 QCon，不会不知道 InfoQ 吧？——冯大辉
从事技术工作，或有兴趣了解 IT 技术行业的朋友，都值得订阅。——曹政



关注「InfoQ」回复“二叉树”，看十位大牛的技术初心，不同圈子程序员的众生相。



聊聊架构

以架构之“道”为基础，呈现更多的务实落地的架构内容。

关注「聊聊架构」
和百位架构师共聊架构



细说云计算

探讨云计算的一切，关注云平台架构、网络、存储与分发。这里有干货，也有闲聊。

关注「细说云计算」
回复“群分享”，
看云计算实践干货分享文章



AI前线

提供最新最全AI领域技术资讯、一线业界实践案例、业界技术分享干货、最新AI论文解读。

关注「AI前线」
回复“AI”，下载《AI前线》
系列迷你书



前端之巅

紧跟前端发展，共享一线技术，不断学习进步，攀登前端之巅。

关注「前端之巅」
回复“京东”，看京东
如何做网站前端监控



移动开发前线

关注移动开发领域最前沿和第一线开发技术，
打造技术分享型社群。

关注「移动开发前线」
回复“群分享”，看移动
开发实践干货文章



高效开发运维

常规运维、亦或是崛起的DevOps，探讨如何
IT交付实现价值。

关注「高效开发运维」
回复“DevOps”，四篇精品
文章领悟DevOps





极客时间

重拾极客精神·提升技术认知

「专栏订阅」

每天 10 分钟，邀请顶级技术专家，探究技术本质，解读科技动态。

「极客新闻」

每天早上 8 点，朝闻技术天下事。

「热点专题」

最前沿的专题，最独特的视角，最风趣的解读。

「二叉树视频」

一档属于技术人的直播和短视频节目，记录与时代并行的技术人。



关注我，获取更多干货



目录

08 阿里微服务之殇及分布式链路追踪技术原理

22 Pegasus：小米开源分布式KV存储系统架构设计

33 Web协议优化指南

45 李维博士：NLP智能化的台前幕后

52 解读阿里全球运行指挥中心背后的运转与实践

57 聊聊双11背后京东的虚拟商品系统

架构师 ArchSummit 特刊

本期主编 薛 梁

流程编辑 丁晓昀

发行人 霍泰稳

60 世界正在走向实时化，谈谈Twitter对流处理的理解与思考

65 流量小生DDOS攻击下，微博如何保证系统稳定不再挂

70 微信视频通话技术的演进之路

73 当AI遇上社交网络：专访Tumblr数据科学总监李北涛

76 Feed是什么？在知乎上如何应用？

81 musical.ly技术副总裁：短视频的泛服务和融合架构实践

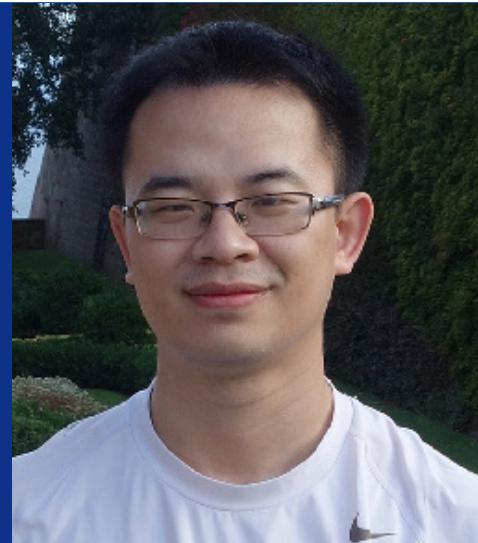
[联系我们](#)

提供反馈 editors@cn.infoq.com

商务合作 hezuo@geekbang.org

内容合作 editors@cn.infoq.com

阿里微服务之殇及分布式链路追踪技术原理



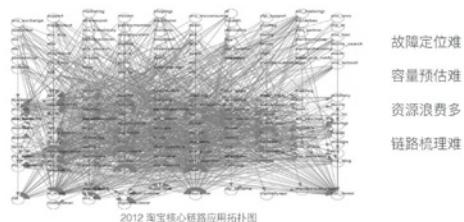
分布式链路追踪技术原理

微服务之熵

微服务的好处已经不用多说，而微服务的坏处，大家看这张图就明白了。这张图是 2012 年淘宝核心业务应用关系的拓扑图，还不包含了其他的非核心业务应用，所谓的核心业务就是和交易相关的，和钱相关的业务。这张图大家可能看不清楚，看不清楚才是正常的，因为当时的阿里应用数量之多、应用间关系之混乱靠人工确实已经无法理清楚了。

基于微服务体系之下构建的业务系统存在的问题基本上分为四类，第一个是故障定位难，今

微服务之“熵”



天我们淘宝下单的动作，用户在页面上点购买按钮，这么一个简单操作，其实它背后是由十几个甚至数十个的微服务去共同完成的，这十几个甚至几十个微服务也由不同的团队去负责，这是微服务的过度协同带来的结果，一旦出现问题，最坏情况下我们也许就要拉上十几个团队一起来看问题。

第二个问题是容量预估难，阿里每年要做若干次大促活动，在以前的巨石系统当中做容量预估是非常容易的，因为我们大促时候按照预估的流量与当前系统的单机压测容量做一个对比，把所有的系统按比例去扩容就可以了。而实际上在大促的场景下，每一个系统在核心链路当中的参与度、重要性都是不一样的，我们并不能对每一个系统做等比例的扩容，所以微服务架构下的容量预估也是一件难事。

第三个问题是资源浪费多，资源浪费多首先是容量预估不准的一个后果，同时资源浪费多背后隐含的另一个问题就是性能优化难，为什么这么说呢？我们当打开一个页面发现它慢的时候，我根本不知道这个页面慢在哪里，瓶颈在哪里，怎么去优化，这些问题累积下来，资源的浪费也成为了一个巨大的问题。

第四个是链路梳理难，我们一个新人加入阿里的时候，老板让他负责一个系统，他在这个复杂的微服务体系中，就像人第一次在没有地图没有导航的情况下到一个大城市一样，根本不知道自己身在何处。应用负责人不知道自己的系统被谁依赖了，也不知道自己的系统下游会影响其他哪些人。

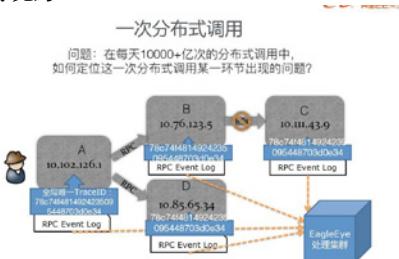
整体来看，我们所说的微服务之熵主要就包含了这四个问题。

鹰眼是什么

鹰眼就是主要的目的就是解决上面所说的这四个问题，我们首先来定义一下鹰眼这个系统，它是一个以链路追踪技术为核心的监控系统，它主要的手段是通过收集、存储、分析、分布式系统中的调用事件数据，协助开发运营人员进行故障诊断、容量预估、性能瓶颈定位以及调用链路

梳理。它的灵感是来自于 Google 的 Dapper。

基本实现原理



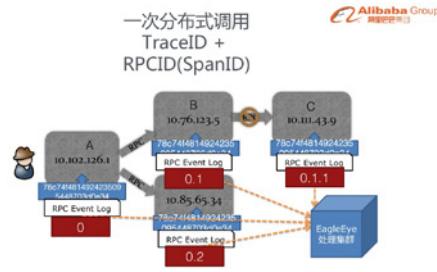
我们来看一下它的基本实现原理。在阿里巴巴每天有超过一万亿次的分布式调用，这个数据其实也是很早之前统计的，如果在一万亿次调用当中出现了一个问题，我们怎么去定位？看一下这个例子，系统 A 调用 B，B 调用 C，在这之后 A 又调用了 D，如果 B 调 C 出了问题的话，那么负责维护 A 系统的开发人员根本不知道问题到底出在哪里，他只知道这次调用失败了，那么我们怎么样解决这个问题？虽然现在的很多大公司都在重复造很多轮子，但还好在阿里巴巴中间件这个东西没有被重复造出两个，基础设施还是相对比较统一的。所以我们可以在一套中间件里做统一埋点，在分布式调用框架、分布式消息系统、缓存系统、统一接入层、Web 框架层的发送与接收请求的地方做统一埋点，埋点的数据能够被一套中间件在系统之间进行无缝透传。



问题：如何还原实际调用“栈”？
RPCID

当用户的请求进来的时候，我们在第一个接

收到这个请求的服务器的中间件会生成唯一的 TraceID，这个 TraceID 会随着每一次分布式调用透传到下游的系统当中，所有透传的事件会存储在 RPC log 文件当中，随后我们会有个中心化的处理集群把所有机器上的日志增量地收集到集群当中进行处理，处理的逻辑比较简单，就是做了简单清洗后再倒排索引。只要系统中报错，然后把 TraceID 作为异常日志当中的关键字打出来，我们可以看到这次调用在系统里面经历了这些事情，我们通过 TraceID 其实可以很容易地看到这次调用是卡在 B 到 C 的数据库调用，它超时了，通过这样的方式我们可以很容易追溯到这次分布式调用链路中问题到底出在哪里。其实通过 TraceId 我们只能够得到上面这张按时间排列的调用事件序列，我们希望得到的是有嵌套关系的调用堆栈。

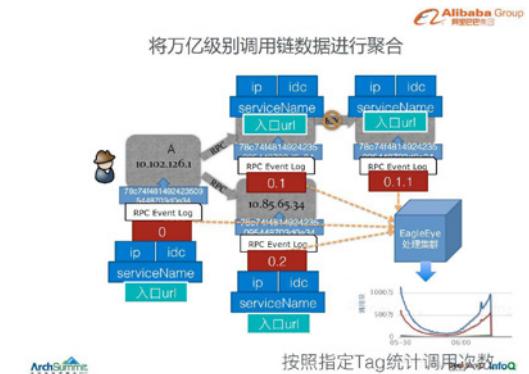


要想还原调用堆栈，我们还需要另外一个东西叫做 RPCId（在 OpenTracing 中有类似的概念，叫做 SpanID），RPCId 是一个多维序列。它经过第一次链路的时候初始值是 0，它每进行一次深入调用的时候就变成 0.1，然后再升就是 0.1.1，它每进行一次同深度的调用，就是说 A 调完 B 以后又调了 D 就会变成 0.2，RPCId 也随着本次调用被打印至同一份 RPC Log 中，连同调用事件本身和 TraceId 一起被采集到中心处理集群中一起处理。

一次分布式调用
TraceID + RPCID

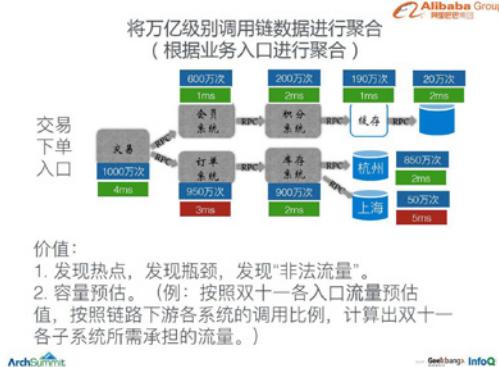


收集完了以后，我们对所有调用事件按照 RPCId 进行一个深度遍历，我们就可以获得这样的一个调用堆栈，上图中的调用堆栈实际上就是真实的淘宝交易系统里面进行下单的交易调用堆栈，可以看到这次调用经历了很多系统。但大家在鹰眼的视角上面来看，就好像是在本地发生的一样，我们可以很容易地去看到如果一次调用出现了问题，那问题的现象是出现在哪里，最后问题的根因又是发生在了哪里。除了调用异常的返回码之外，我们在右边其实还可以看到每次调用的耗时是多少，我们也可以看到每一次调用如果慢了它是慢在哪里。我们从这张图中解释了鹰眼是如何解决微服务四大问题中的故障定位难的问题，它可以通过倒排索引，让用户反查出每一次调用的全貌是怎样的。



如果我们对万亿级别的调用链数据进行聚合，是否能够获得更有价值的信息？我们可以看

一下，每一次调用除了它唯一标识 TraceID 和 RPCID 之外，还包含了一些标签信息（Tag），什么是标签呢？就是具备共性的，大家都会有的这么一些信息，比如说这次调用它分别经历了这些系统，这些系统它每次调用的 IP 是什么，经过哪个机房，服务名是什么？有一些标签是可以通过链路透传下去的，比如入口 url，它透传下去以后我就知道这次请求在下去之后发生的每一次事件都是由通过这个入口去发起的，那么如果把这些标签进行聚合计算，我们可以得到调用链统计的数据，例如按某机房标签统计调用链，我们就可以得到每个机房的调用次数的趋势图。



这样做有什么好处呢？实际上在容量预估当中这样做的好处是非常明显的。我们来看一个例子，假如我们有一个交易下单入口标签，我们对这样的标签做了聚合以后，不光能看到单次调用它的情况是如何，还能看到将这些调用链数据聚合以后，它的总调用次数是多少，平均耗时是多少，我可以发现系统当中热点瓶颈在哪里，同时我们可以发现一些非法流量，也就是说我之前不知道的事。

比如说我们今天看到交易下单入口，订单系统耗时比以往要长了，以往都是一毫秒，但现在是三毫秒，我们通过完整链路可以看到，实际上

这次交易下单应该是一个单元内的操作，但是有一部分流量因为业务逻辑的原因，本该到杭州的数据库最后跑到上海的数据库，这么一个跨集群的网络调用，实际上会导致整体的耗时增加，我们可以通过这种方式看到它热点是由什么原因引起的。同时，我们在做双 11 流量预估的时候，机器预估其实可以很容易做，因为我们只要知道交易下单入口我们预期的峰值是多少，可以按照平时的调用比例知道这次下单入口，在下游的每一个调用环节分摊到的基线流量是多少，根据基线流量来做等比例的扩容，就可以得到双 11 相对来说比较精准的扩容机器数量。



这是真实的我们系统当中的下单链路聚合结果图。除了我刚才说的一些点，我们还可以获得一些什么呢？比如说易故障点，也就是说这个系统看上去大面上没什么问题，但是偶尔它会出一些小问题。为什么我们说是易故障点，因为一旦它出现问题，下游的链路就终止了，也就是说调用对下游系统其实是强依赖的关系，那我们可以认为这个链路如果有一天下游系统真的挂了，那么整条链路的稳定性很可能出现问题，我们可以根据这些数据帮助用户提前发现这些问题。

第一节就是对基础功能的小结，它分为两部分的功能，第一个是通过 TraceId 和 RPCId 对分布式调用链的堆栈进行还原，从而实现故障定

位的功能。同时通过调用链数据分析，将这些入口、链路特征、应用、机房这么一些 tag 进行聚合统计，可以做到容量预估、性能瓶颈的定位以及调用链的梳理，所有这些都包含在我们在 2012 年发布的第一版鹰眼系统中，我们接下来看一下这套系统背后的技术演进过程。

EagleEye 架构演进

鹰眼 2012 年架构

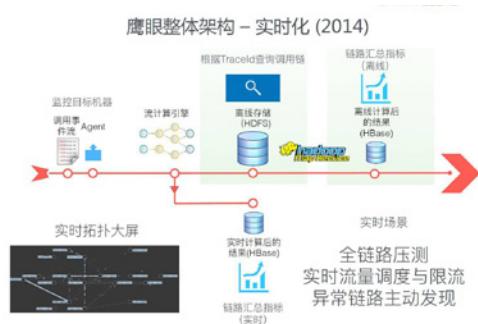
这是我们 2012 年架构，这一套架构完成了刚才我在第一章里面所说的这些功能。其实后端的技术架构看上去非常简单，但实际上是非常“重”的，我们整体的技术架构实现的大概是这样一个流程，中间件会打出调用事件流，统一在所有机器上安装的 agent 把数据上报到收集集群，然后数据进行一些简单的清洗以后，将它存到 HDFS 当中。后面我们做两件事。



第一个就是把调用链数据按照 TraceId 去做一个排序，这样用户去查找的时候就可以通过类似像二分查找法的这种方式，定位到 TraceId 经历的所有调用事件。第二件事是按照 tag 进行聚合得到了一个统计报表。2012 年的这个架构初步满足了功能，但可以看到明显的问题是第一个不够实时，因为整个 Hadoop 的这一套系

西是批次执行的，我们批次越大，它整体的吞吐量就越大，那么它显而易见的问题就是不够实时。我记得当时我们的延迟大概是一个多小时左右，也就是说发生了问题一个多小时以后才可以查到。第二个是不够轻量，如果我们在多个机房输出的话，每个地方都要部署这么重的一套东西，我们越来越深刻地意识到对于监控数据来说，它的价值随着时间的推移迅速地衰减，如果我们不能第一时刻给用户提供到这些数据的话，实际上我们的监控数据没有发挥出它最大的价值。

鹰眼 2014 年架构



也因为这样的原因，我们在 14 年的时候完成了这么一个实时化的改造，我们把原来比较简单的、职责较为单一的负责链路调用收集的中心集群改造成了用现在大家都知道的流计算引擎去完成这件事。我们把一部分核心的，我们认为比较重要的，用户认为比较重要的数据通过流计算引擎增量计算，第一时间经过统计和聚合存储到 HBase 当中，通过大屏幕可以第一时间实时地看到用户整体的微服务之间的调用流量情况。通过实时化的改造我们也催生出了一些场景，比如说 14 年，在全链路压测的时候鹰眼成为了不管是运维人员还是开发人员非常重要的数据支撑工具，它可以实时看到流量压测下来以后每个系

统的响应情况和链路之间的调用关系。同时实时流量调度与限流也是会基于这么一个调用链的情况去追溯到，比如说我发现有个下游系统扛不住了，那么我可以根据这个链路数据反过来追溯到上游，去看到链路数据的发源地是哪里，是手机淘宝还是天猫的交易。可以根据这样的链路追溯实时地去响应去做一些流量调度和限流的工作，异常链路整个的数据必须以非常高的实时性呈现给用户。

鹰眼 2016 年架构



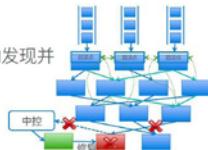
那么 2016 年我们又做了哪些呢？这套系统足够实时，但是还不够轻量，因为 Hadoop 还在那，那么 2016 年我们彻底把 Hadoop 那一套给抛弃掉，把所有的统计计算全部变到了实时引擎当中。每天几万亿查询的调用链事件存在 HBase 当中显然是扛不住的，我们是放到一个叫做 HiStore 的 MPP 列式数据库当中，这一套列式存储数据库不但能够完成原来按照 TraceId 的事情，同时它还可以通过耗时超过多少毫秒，错误码是多少各种各样的多维查询条件，把 TraceId 给查出来，更加方便用户去查找问题。

流计算挑战

我们在实施这套流计算的过程当中遇到一些

监控系统 - 流计算的挑战

- 持续计算稳定性
24x7持续计算，无法停机维护。
宕机自愈能力。
- 解决方案
具备高可用中控节点，自动发现并
重启计算节点的流计算引擎
(JStorm)。
自带Exactly-Once语义。



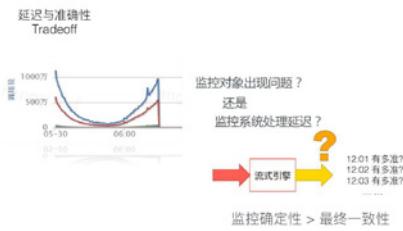
坑。第一个我们回顾一下离线计算实时计算区别，离线计算实际上是通过批次计算这么一个手段，对静态的文件进行分而治之的 map-reduce 的工作，流计算实际上是对无限的流数据切成很小的批次，对每一个批次进行的一个增量的计算，也就是说数据是不断上来的，监控数据不断产生的。以往的做法是把它存到一个大的存储当中，后面悠哉悠哉地去跑一下，计算任务挂了也没有关系。但是流计算我们不能这么做，我们必须保持 24 x 7 的稳定性，没有办法停机维护的。因为一停机的话整个监控数据掉下来，因此系统必须具备自愈能力。解决方案其实是我们跟 JStorm 团队进行了紧密的合作，做了不少优化，比如说把中控节点改成了高可用，自带 Exactly-Once 功能，当数据出现问题的时候，我们可以永远通过上一回滚点去回滚到上批次的数据，这样可以做到监控数据永不停息，永不丢失。

流计算第二个挑战我刚才也提到它的中间结果是尽量不落地的，那么不落地带来的问题就是说它实际上要比离线计算要难得多。举一个例子，统计全量数据，如果我们是统计一个入口 UV 这件事，其实我们在离线计算当中，这是非常容易做的。按照访问 Id 排个序，遍历得到它的 UV 就可以了，但是在流计算里面是比较难做的。这边简单提两点，一个就是全量计算到增量计算的

过程，这个过程我们实际上是用了一些开源流计算库，这边给大家推荐 StreamLib，它可以很容易地帮你通过近似计算估算的方式完成一些看上去在流计算里面比较难的事情。流计算就跟离线计算一样，热点问题在离线计算当中，因为我预先知道热点在哪里，其实很容易去做数据的拆分，但是在流计算当中我们预先是不知道的。所以其实我们的做法也非常简单，我们前置了一个 LocalReduce 节点，也就是说在所有的监控数据进来之前先在本地监控一次，实际上再发到下游的时候它的流量就是均匀的，它就不会对下游的节点产生影响。

监控系统

监控系统 – 确定性保证

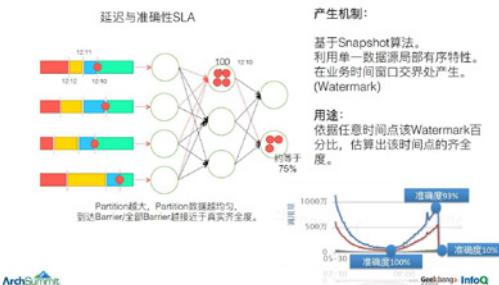


做监控系统，大家只要做过监控系统底层设施建设的其实都可能碰到这样一个问题，也就是说那这个问题，其实我在刚开始做鹰眼的时候也是碰到比较痛苦的一个问题，就是说凌晨两点钟突然监控的曲线掉下来了，这时候我应该给运维人员发报警，还是不发报警，为什么我很难做这个抉择，因为我不知道监控的对象真的出了问题，还是监控系统本身的流计算部分出了问题，流计算系统是不是哪里卡住了，还是数据收集通道出现了问题。其实当流计算系统被应用到了监控领域中，它提出了更高的要求，一个叫做“确

定性”的要求。“确定性”要求在监控系统里面，是远远大于最终一致性的，所以为了解决这个问题，我们看了一圈现在的流计算引擎提供的解法，实际上没有很好的现成方案去解决这个问题。

相关优化

相关优化 – 齐全度优化



我们自己实现了一个齐全度算法来完成确定性的保证。它与 Apache Flink 中采纳的 Snapshot 算法有些近似，它其实是在整个流的过程当中去安插一些屏障 (Barrier)，我们可以做一个假设，在所有流计算数据产生的地方，数据都是有序的。也就是说我们在收集数据的时候，通过第一探测数据源的深度，第二个在数据时间间隔出现交替的时候，比如说从 12:10 跑到 12:11 的时候去安插一个屏障随着流做传递，流计算过程中保证 FIFO，下游可以把这些屏障聚合起来，这样最终数据落地的时候，我就知道数据源当中有多少个屏障已经达到了最终的存储当中。通过这种方式，我们其实可以预测出当一个系统在任意一个时刻它的齐全度是多少，这样的话我们其实可以很容易得到一件事，就是说这个线掉下去了，但是因为在某一个时间点的屏障还没有完全到齐，所以我现在可以预测是因为监控系统卡住了，没有办法给下游的运维人员

发报警。因为我知道现在数据还没有收齐，我没有具备确定性的阈值保证，是没有办法去发报警的。这个齐全度的算法，也就是我们在流计算当中，针对监控系统报警场景作出的确定性保证的优化。

存储层优化

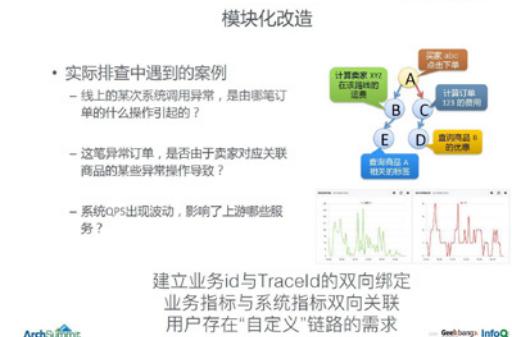
存储层的优化也说一下，我们一开始是用 HBase 来做倒排索引的，其实我们现在在测试环境当中还是用 HBase 来存，因为简单可靠。第二个阶段也就是我们在 2012 年的时候，把这个 TraceId 进行分片，按 TraceId 排序，反过来用户通过二分查找方式进行查找。实际上我们现在是用了列式存储的技术，区别是把同一列的内容用物理相邻的方式进行存储，这样可以换来更好的压缩比。

因为我们知道对于调用链的数据来说，大部分的调用链数据都是一些非常相近的数值型类型，比如说 2ms, 3ms, 1ms 这样的数据，如果物理上能够连续地存储在一起的话，它的压缩比是非常高的，我们实际通过我们内部叫做 HiStore 这么一个产品，经过这个产品的压缩以后，我们整体调用链的压缩比最高能达到 20:1，也就是说每天如果是 100G 调用链日志，我们最终压缩下来只有 5G。

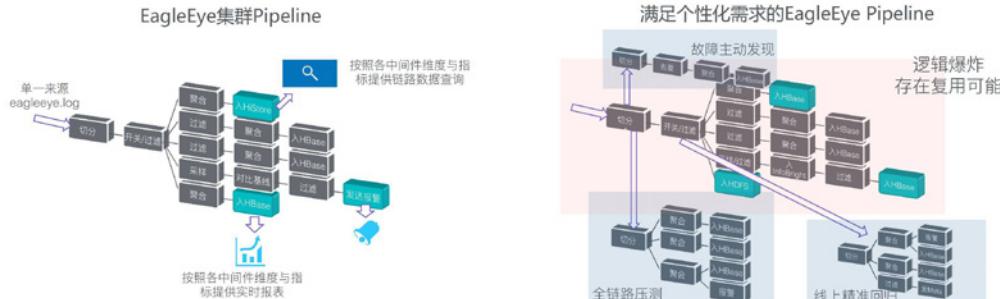
指标存储，因为大家对现在对时间序列数据库的开源的实现应该也有所了解，我们对 OpenTSDB 也做了一些改造，第一个把 Proxy 层直接干掉了，降低了网络开销和一些不确定性。第二个 OpenTSDB 必须让你的数据一次算完指标，直接丢到里面存储，没有办法再进行更新的操作。而在一个真实的流计算场景下，我们知道数据有早来的、迟来的，因此业务上原本应该属

于同一个时间窗口的两条数据有可能实际却在相差非常遥远的两个系统时间内到达流计算引擎，晚到的数据对于原先指标的更新的操作就很难去做。我们在基于 HBase 的时序存储的背后增加了应对这种场景的 Co-Processor，对于这些晚来的数据背后可以进行一个合并操作，变相地解决了流计算当中数据和时间窗口不对齐的问题。

模块化改造



下面一个议题上是我们对鹰眼进行的模块化改造，我先讲一下背景，大家如果对分布式调用链系统非常熟悉的话，实际上它的核心还是在于 TraceId，也就是说我要查什么问题必须有 TraceId 才行，但实际业务当中碰到的问题，都不是开发人员或者运维人员发现的，而是一线的客服发现的，是业务操作人员发现的，他会跟你说某某订单号的订单有问题，某某用户发现某某商品状态不对。其实这些问题都是跟业务紧密相关的，实际上最终所有用户向我们提出了一个需求，需要能够将业务 Id 和 TraceId 双向绑定，第二个是业务指标和系统指标双向关联。这两个需求的本质都是用户需要“自定义”自己的链路。这样的需求来了，我们按照 2012 年的架构来做，就出现了一些问题。



按照我们之前所描述的架构，一种简单解决自定义链路数据的解决方法是，让所有的用户都把自定义的数据按照规范打同一份日志，所有的这些计算方式我们沿用原来的那一套，用统一的方式去处理用户的自定义链路日志，把它和另外的中间链路进行关联就可以了，实际上我们发现在阿里的环境下这套方案没法实施。第一个原因用户的日志量根本是不可控的，存储和计算的开销很大。

第二是用户的这些数据，从各个方面都充满了自定义的需求，比如说它数据来源不一定是日志，可能是队列，可能是数据库的修改的 binlog，可能是巡检、拨测的结果。数据过滤方式也不一样，它聚合的维度更是五花八门，就算是我们系统的数据，聚合维度都是不一样的。持久化方式也是不一样的，因为我们原生的方案只提供两种，一种是列式存储，它是基于写多读少这么一个场景，有的用户说我要毫秒级查询，我想用某某存储来存这些数据。部署方式也存在各种各样，比如说跨机房、独立部署、专有云输出的需求等。

这样的自定义链路监控需求我们是没有办法满足的。当需求无法满足时，大家都喜欢造轮子，监控系统里面的这些轮子造起来还真的蛮有意思，它有流计算，高压缩比的存储，有各种各

样的数据收集，海量数据挑战。所以我们在想，与其我们订一套规范让你们强制接入，然后你回过头跟我们说我们的这套规范满足不了你们的需求，所以你们得自己造一个轮子，还不如把轮子直接给用户，让用户自己通过这些轮子造出自己的汽车来。一开始我们尝试着自己在 pipeline 里面写代码。全链路压测、线上精准回归、故障主动发现的这些场景我们就在原来那套逻辑里面去写，写完了以后发现真的是扛不住了，因为我们光是解决中间自己的问题就已经连轴转了。

后来我们发现在这套 pipeline 里虽然需求五花八门，但组成 pipeline 的各个组件其实存在大量复用的可能，所有的这些对于监控数据的计算逻辑都是可复用的。

积木块系统

如果我们能够把这些可被复用的逻辑变成积木块，让用户按照自己的需求去拼装监控系统的 pipeline，对用户来说是一件很爽的事情。所以我们干脆就做了一套积木块的系统，其实是把整个监控系统当中这些模块，进行了抽象和分类，比如说切分、逻辑判断、聚合、持久化、告警，还有一些用户自定义的逻辑，这一块也是借助了这套叫做 Google Blockly 开源可视化的框架，我们对它进行二次开发。用户可以直接在我们的

界面上，拼出自己的监控数据处理流程。



日志来源: buy

日志路径: /home/admin/logs/buy.log

```
2016-07-25 17:25:00|0|a48514414249449347162339e|0.1|18661234567|天猫|300|  
ERROR  
2014-07-25 17:25:00|0|a48514414249449347162339e|0.3|18661234567|淘宝|400|  
SUCCESSFUL  
2014-07-25 17:25:00|0|a48514414249449347162339e|0.15|18661233445|聚划算|500|  
53160|SUCCESSFUL
```

...
竖线分割，分别表示：

日期|TraceId|RPCId|电话|来源|价格|操作结果

目标: 1. 如果操作出现失败，将该业务事件与鹰眼调用链产生关联。
2. 分别统计各来源的交易额度

我们来看一个例子，假如有一个交易系统，

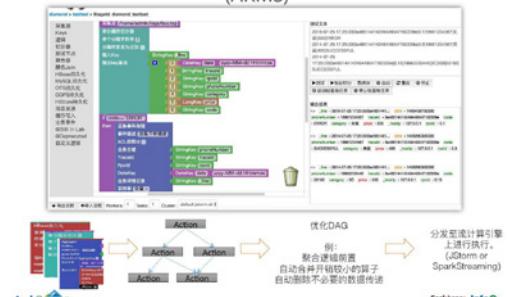
现在系统里面有一份日志，日志按照竖线分隔分别代表不同含义，它对我们的要求是，第一：如果操作出现失败，将该业务事件与鹰眼调用链产生关联，这样根据交易的 ID 我们能查到底下链路到底失败在哪里。第二个是现在原生的鹰眼里虽然有交易下单的微服务调用曲线，但实际上想知道与之对应的各个来源交易额度，想跟微服务的调用曲线放在一个视图上进行关联的对比。

我们首先定义数据来源，然后定义切分规则竖线分隔，分别这些字符是什么意思。当返回码为“ERROR”时，去记录一个鹰眼事件，这个事件是什么呢？交易下单错误，然后我把 TraceId 和用户的手机号进行关联，同时 RPCId、日期都关联上去，我们再定义一个 Aggregator 积木块来做一个聚合计算，比如按照比如入口去进行一

个聚合，聚合的计算是对价格的累加，聚合后顺序存到 ODPS 阿里的大数据存储平台，可能后面这个用户还需要对这份日志去做一个离线的分析。



将“积木块”转化为流计算
(ARMS)



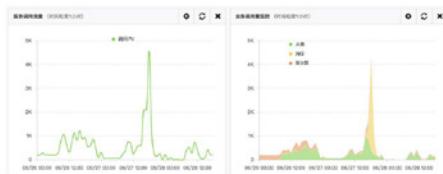
这一套积木块的框架，只要在页面上面简单拖拽就可以完成，同时这个框架还提供了一个所见既所得编辑模式，也就是说你把样例日志放在右上角，左边拼出你的积木块以后点一个测试，右边结果就出来了，来验证你这个积木块的逻辑是否符合预期。点击启动这个积木块就会被翻译成流计算的逻辑，把它提交到当前比较空闲的一个流计算 Slot 去执行，当然这个翻译的过程我们也是有一些优化。自动合并开销比较小的算子，比如说简单的一些切分、逻辑判断，把它压到一个 action 当中去执行。不必要的数据传递，也就是说下一步不需要的 key 和 value，其实我

们帮它自动删掉了，这样用户不会进行那些没有必要的网络开销，点击启动以后这个逻辑就直接跑到我们鹰眼计算逻辑集群里面去执行了。

将业务与系统链路双向绑定



将业务与系统指标双向关联

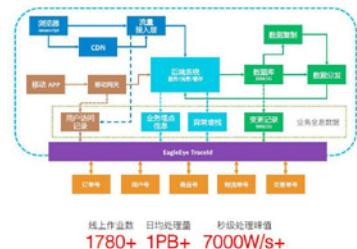


用户对这套积木块系统比较熟悉的话，基本上五分钟就可以完成一个自定义链路的过程。自定义链路完成后的结果在图中我们可以看到，在是鹰眼的系统调用堆栈之上，会穿插显示出在与系统调用并行出现的业务事件，另外，由于 TraceId 与业务 Id 形成了双向关联，客服人员在进行排查问题的时候，只要输入一个手机号，对应的所有系统链路就出来了。除了 Id 关联之外，业务指标和系统指标也实现了双向关联，我们可以在一个视图中看到左边是服务用量，右边是各个业务的图，在上面这张图的例子中我们可以看到这个时间点的整体调用量上去了，我们可以知道大部分是来自于淘宝业务，其他的两个天猫和聚划算的业务量是没有什么变化的。

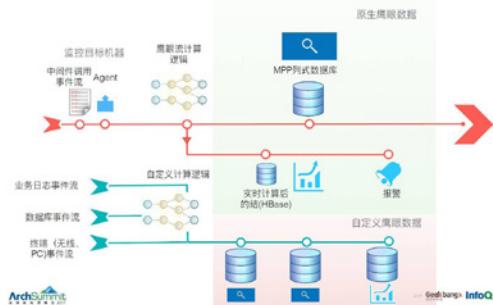
模块化改造的结果，总结一下就是 2014 的第一个季度以前鹰眼是一个只能接受固定来源格

式、按固定方式清洗、聚合、持久化的监控系统，到了 2014 年的 Q1 以后，实际上它也变成了一个自定义流式数据采集、清洗、计算、持久化的系统，它可以让用户自定义的链路数据和原来的鹰眼链路数据进行双向的关联，很快的，这套系统就在用户当中推广起来了。

全息排查全景图



截止七月前两天还统计了一下，整个鹰眼的集群线上作业数超过了 1780 了，实际上其中只有 2 个是鹰眼原生的链路统计，其他都是由用户自己配置的，可以说我们是提供了一套接入平台，是用户帮助我们构建了完整的鹰眼全链路监控体系。每天日均处理量是一个 PB 以上，秒级处理峰值是 7000 万每秒。除了原先中间那块蓝色中间件之外，我们还接入了下面橘黄色的用户业务上面的数据以及一些其他上下游的基础设施的链路数据，比如说移动用户终端、浏览器等设备的链路数据，还有数据库层面的数据，我们把鹰眼整体架构 - 模块化 (2016)



TraceId 和数据库的 binlog 打通了，当然这个也是依赖于我们数据库团队的一些改造。

上面红色的部分实际上是模块化之前，模块化之后蓝色部分就是用户自定义的链路，日志事件流、数据库事件流、终端事件流都可以通过自定义的逻辑存到指定的存储当中，同时它的这些数据一定要通过 TraceId 和我们原生的这些鹰眼数据进行关联，这样就算大家再怎么重复造轮子，最终都可以关联在一起，为一辆汽车所用，这个才是造轮子的最终目的。

从被动到主动

我们聊一下从监控系统如何从被动报警转化为主动发现，这一转变是我们从去年开始启动的一个项目。刚才大家看到的鹰眼系统目前已经到了 7000 万每秒的处理峰值，在这个之前其实我们不断地做优化，2016 年以前我们自己一直陶醉在技术优化的享受当中。到了去年的时候，我们开始反思问自己一些问题，这些问题包括：我们真的需要存储每一条链路吗？就算我们的压缩比很高，但实际上随着阿里流量不断的增长，不管怎么优化存储量也是相当惊人的。第二个是就算我们有那么多数据，用户在上面配了这么多报表，我们给用户算出这么多报表，用户真的理解每个报表的含义吗？第三个是我们能不能比用户更早发现问题，在这个之前实际上鹰眼还是一个被动的用户出现问题了过来查问题的系统，我们能不能领先一步，在用户发现问题之前提前把问题暴露，给用户预知问题，甚至提示用户问题的最终原因在哪里。

识别、关联、定位

我们解决问题的方法基本上也是分为三个步



骤，第一个是识别，第二个是关联，第三个是定位，具体的说一下每一个步骤。

识别什么呢？识别每一个链路当中出现异常的情况。时序指标当中的异常点，也就是突然高上去那一瞬间，那个时间点我们要识别出来。指标中的一些离群点，比如说机房当中所有的机器，某些机器出现和其他机器不一样的行为，我们把它识别出来。



非结构化日志中的离群点与异常

我们还能识别什么？其实今年我们做了一个挺有意思的项目，用户接了这么多的自定义日志进来，我们实际上把用户的这些日志进行了自动的归纳和整理。平时大家做应用负责人的时候可能都会有一个经历，我们常常会带着很强的“主观性”去观察我们的日志。特别是在发布的时候会盯着日志看，看日志跟以前长得差不多，这次发布好像是没什么问题，换一个人来发布的时候就完全懵了，因为他不知道日志以前长什么样子。

我们其实做这件事的目的就是帮你把日志每一个种类当中的次数归纳整理出来。

当你的日志出现异动，什么叫异动，如果你的日志里面长期出现某一类的异常，这个不算异动，这个叫破罐子破摔，这个异常没有人修的，或者是前面开发的人都已经离职走了，没人知道怎么回事。异动是什么，昨天没有，今天有了，昨天每小时只有两百条的，今天每小时有两万条，这是一个异动。

再说一下识别，我们会把这些异动给识别出来，也作为识别的一个依据，和之前的所有异常点结合在一起，进行一个关联，怎么关联呢？



第一个是按点关联：同一个部署单元，比如说同一台机器上面或者同一个应用里面的这些东西可以相互关联。第二个是按线关联，时间点上靠近的，比如说这件事在某个时间点发生的，在前一分钟发生了另外一件可疑的事，这两件事其实可以关联起来。第三个是按链路关联，下面是一个链路，在 A 点上发生的事在 B 点上同一时刻也发生了，这两件事就可能有关系。最终通过关联的结果我们可以定位原因了，通过这三步识别、关联以及最后定位，我们可以定位到最终这些相关联的事到底什么是现象，什么是原因，可以把现象和原因做出有可能的关联。

举个例子，我们前段时间通过这套系统发现

了一个案例，某个核心入口应用 A 的业务指标出现了小幅的波动，另外一个系统应用卖家中心日志出现了新增的异常堆栈，这两件事通过识别把它识别出来，当然在同一个时间点上识别出了一堆其他乱七八糟的异常，我们是怎么把这两件看似不相关的事情关联起来呢？

第一，在波动前一分钟，我们同时发现了做数据库拆分有分库分表的规则推送的变更，在这个变更发生前一分钟我们发现了业务指标的波动。按点关联是什么，因为应用 B 实际上是受到了配置变更的，那么按时间线关联，实际上这两个事情，时间线上面是非常接近的。链路关联是什么呢？应用 A 是强依赖应用 B 的，两个应用强依赖，其实是可以通过一些基线去得到。那么最终我就可以认为这一次的分布分表配置变更是引发 A 应用指标波动的高嫌疑原因，实际上我们通过识别出来异动新增的异常堆栈，人工验证以后确实如此。也就是说这一套识别加上关联再加上定位的手段，我们认为有可能是行之有效的，我们也准备将这套系统继续开发下去，今后有什么新的进展也可以跟大家一起同步。

总结

阿里巴巴鹰眼系统演进史



总结一下之前讲的一些东西，实际上我之前讲的这几部分内容的顺序，也就正好吻合鹰眼版

本的演进史，第一版本实现基本的链路定位以及指标分析的能力。第二个是平台化，通过积木块的方式把这些轮子给用户，让用户自定义我们的链路，用模块化平台化的手段让所有用户把不同的数据接入进来，共建我们的全链路系统。第三个版本实际上是全息排查，同时我们对流计算以及底下的一些存储做了优化。

实际上如果我们去看整个监控领域的话，基础运维层面是大概五年前大家都做得滚瓜烂熟的事情，再往上是应用层以及链路层的监控，这些监控通常是通过调用链的数据或者调用链聚合的数据来完成的。业务层面，我们希望通过业务和应用系统层面双向的 TraceId 的关联以及指标的关联来完成一件事，就是业务出现了问题，我能第一时间定位到底是哪个系统出了问题，哪个系统的变更可能导致业务层面给用户感知的问题。

17 年我们的鹰眼系统也正式在公有云向用户开放，目前 EDAS 中的鹰眼包含了对阿里中间件（Aliware）体系之上构建的微服务系统提供

鹰眼3.0 – 一个端到端的APM实现



了开箱即用的链路监控能力。而更加完整的自定义链路监控功能、前端浏览器监控以及其他相关的 APM 功能在阿里云的 ARMS (Application Realtime Monitoring System) 上面也开放了，如果将 EDAS/PTS/ARMS 配合在一起使用的话基本可以让一个企业在微服务治理、容量规划、应用链路监控以及全链路压测方面做到和阿里相同的水准。同时我们内部也在做一些智能诊断相关的一些事情，包括业务链路自动梳理和自动根因定位的能力，在合适的时机也会在云产品上开放给大家使用。

周小帆，阿里巴巴高级技术专家。8 年金融 + 电商 + 中间件工作经验，目前就职于阿里巴巴中间件技术部，整体负责中间件的监控与数据化运营工作，同时为阿里商品、菜鸟、安全、国际站等多个业务部门提供了完整监控方案。

参与了阿里近五年来监控体系的建设及演进。阿里云“业务实时监控（ARMS）”技术负责人。兴趣是业务架构、分布式计算与数据存储技术。

小米开源分布式KV存储系统 Pegasus

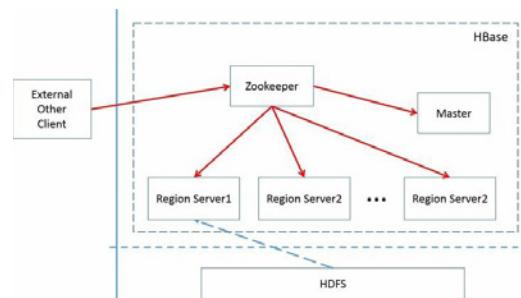


Pegasus 的产生

在 Pegasus 立项之前，小米的分布式存储主要使用 HBase。经过 10 多年的发展，HBase 本身还是比较稳定的，并且功能接口上也较为方便。但由于设计和实现上的一些问题，我们在使用上还是会有一些坑。

记一次 HBase 事故

HBase 的架构，我相信大家应该对其也比较熟悉了。如上图所示，HBase 由一个 Master 节点来管理整个集群的状态。在 Master 节点的统筹管理之下，Region Server 节点负责客户端的读写请求；Region Server 的数据，存在一



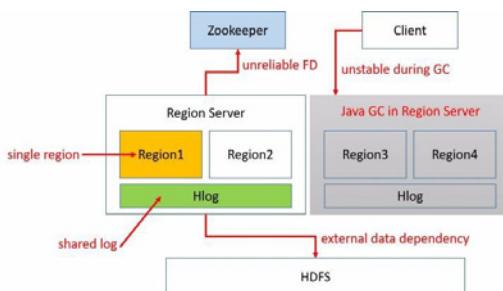
个外部的分布式文件系统 HDFS 之上。Region Server 的心跳探活，以及 Master 的高可用都由 Zookeeper 来负责。

我们在实际使用时，很多问题都由 Zookeeper 而起。Zookeeper 在设计之初，它是一个要求负载比较低的系，所以在小米这边，

Zookeeper 大多都由多个业务混合使用。除了 HBase 之外，很多其他业务的节点探活，服务注册等都依赖 Zookeeper。但一些业务在使用 Zookeeper 时，会将超额的压力加到 Zookeeper 上，从而导致 Zookeeper 的不稳定。除此之外，一些对 Zookeeper 的误用，如访问时不共用 socket 连接，也是 Zookeeper 服务不稳定的来源。

各种误用因素的存在，会导致 Zookeeper 经常遇到崩溃。对 HBase 而言，Zookeeper 的崩溃就意味着 Region Server 也随之崩溃掉。而小米的很多业务都依赖 HBase，一旦 HBase 崩掉，小米的很多业务也随之无法提供服务了。

HBase 的不足



通过对 HBase 的一系列问题进行复盘，我们认为有几点值得提一下：

1、对于 HBase 而言，它将“节点探活”这一重要的任务交给 Zookeeper 来做，是可以商榷的。因为如果运维不够细致的话，会使得 Zookeeper 成为影响 HBase 稳定性的一个坑。

在 HBase 中，Region Server 对“Zookeeper 会话超时”的处理方式是“自杀”。而 Region Server 上“多个 Region 合写一个 WAL 到 HDFS”的实现方式会使得“自杀”这一行为的成

本比较高，因为自杀之后 Server 重启时会拆分和重放 WAL。这就意味着假如整个 HBase 集群挂了，想要将 HBase 重新给拉起来，时间会比较长。

2、即使我们能保证 Zookeeper 的稳定性，“节点探活”这一功能也不能非常稳定的运行。因为 HBase 是用 Java 实现的。GC 的存在，会使得 Zookeeper 把正常运行的 Region Server 误判为死亡，进而又会引发 Region Server 的自杀；在其之上的 Region，需要其他的 Server 从 HDFS 上加载重放 WAL 才能提供服务。而这一过程，同样也是比较耗时的。在此期间内，Region 所服务的 Key 都是不可读写的。

对于这一问题，可以通过将“节点探活”的时间阈值拉长来解决。但这会使得真正的“Region Server 死亡”不能被及时发现，从而另一个方面引发可用性的问题。

3、GC 的另一个问题是，HBase 在读写延时上存在毛刺。我们希望在广告、推荐这种业务上能够尽量避免出现这种毛刺，即能够有一个比较稳定的延时。

把上面的三点概括一下，我们认为 HBase 在可用性和性能延时方面还是存在一些瑕疵的。对于这些问题，通过修修补补、调整参数的方式能够得以缓解。但想从根本上解决，还是不太容易。

Pegasus 的定位

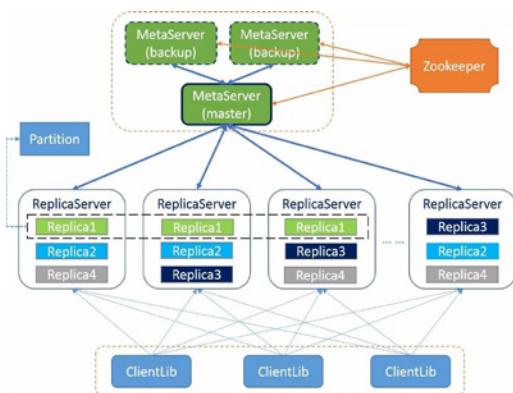
既然搞不定 HBase，那么我们只好自己重新造一个。由于已有 HBase 这样一个标杆，我们在定位上也非常明确：对 HBase 取长补短。具体来看：

HBase 的一致性的视图和动态伸缩，是存储

系统里非常好的两个特质，我们希望可以保留；对于 HBase 的性能延时和可用性问题，我们应该通过架构和实现着力弥补，从而将系统覆盖给更多的业务使用。

有了这几个定位之后，我们架构也比较清晰的浮出了水面：

Pegasus 架构一览



总体来说，该架构借鉴了不少 HBase 的内容：

从总体来看，Pegasus 也是一个中心化管理的分布式存储系统。MetaServer 负责管理集群的全局状态，类似 HBase 的 HMaster；ReplicaServer 负责数据读写，类似 HBase 的 RegionServer。

为了扩展性考虑，我们也对 key 分了不同的 partition。

和 HBase 不同之处在于以下几点：

心跳并没有依赖 Zookeeper，而是单独抽出来，直接由 MetaServer 进行管理

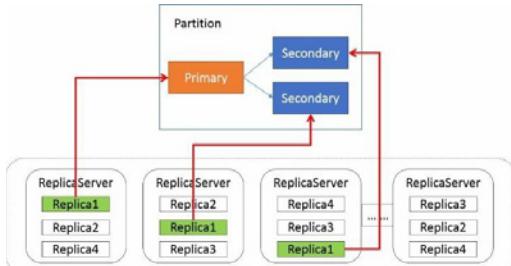
数据不写到第三方的 DFS 上，而是直接落入 ReplicaServer。

为了对抗单个 ReplicaServer 的失效，每

个 Partition 都有三个副本，分散到不同的 ReplicaServer 上。

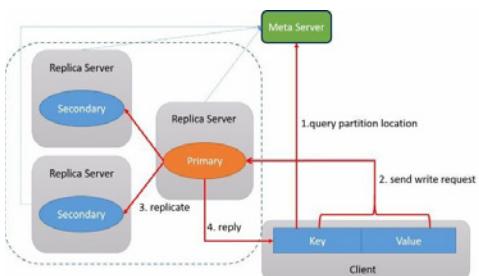
目前 MetaServer 的高可用依赖于 Zookeeper，这是为了项目开发简单上的一个权宜之计。后面可能引入 Raft，从而彻底消除对 Zookeeper 的依赖。

多副本的一致性协议



前面提过，我们的每个 Partition 都是有副本的。为了满足多副本情况下的强一致性，我们必须得采用一致性协议算法。我们使用的是 MSRA 发表的 PacificA。PacificA 和 Raft 的对比可以参考我们在 github 项目中的文档，这里不再详细展开。总的来说，我们认为依照 PacificA 的论文来实现一个可用的存储系统，其难度要比 Raft 更低。

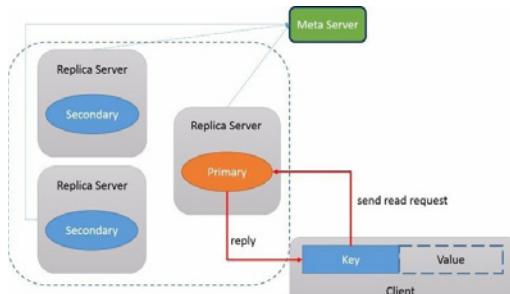
Pegasus 写请求流程



在 PacificA 协议里边，负责接受读写请求的叫做 primary，相当于 Raft 中的 leader；另外两个接受 replication 请求的

是 secondary，相当于 Raft 中的 follower。在此架构下，对一个 partition 写请求也比较简单：如果客户端有一个写请求，首先需要向 MetaServer 来查询 Key 的位置，之后再向 primary 所在的 ReplicaServer 发起写请求，然后 primary 将其同步给 secondary，二者都成功后再返回客户端写请求成功的回复。

Pegasus 的读请求流程图



读请求操作更加简单，客户端直接和 primary 发起读请求，primary 因为拥有全部数据，所以直接进行响应。

实现上的那些坑

前面大致回顾了 Pegasus 的设计考虑和总体架构。现在进入我们的正题，来看看这样的架构在实现上会有哪些问题？

扩展性

首先来看扩展性。扩展性分为两点：

- partition schema 的问题：一个完整的 key 空间，怎么样把它分成不同的 partition；
- load balance(负载均衡) 的问题：如果 partition schema 定了之后，怎么样把这些分片用一个比较好的算法分到不同的机器上。

Partition schema 的选取

对于 partition schema，业界解决方案一般有两种，第一种是哈希，第二种是排序。

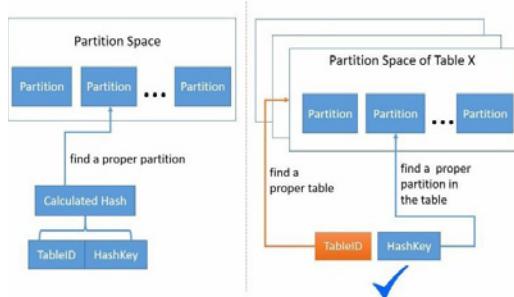
对于哈希，就是将所有的 key 分成很多桶，一个桶就是一个 partition，然后根据 key 的哈希值来决定分配到某个桶中；对于排序而言，所有 key 一开始都在一个桶里边，然后依据桶的容量进行不停地动态分裂、合并。

对比这两种方案，排序比哈希多一个天生的优点是排序方案有全局有序的效果。但由于排序需要不停地做动态分裂，因此在实现上更麻烦。

另外，如果采用 hash 的 partition schema，数据库不太容易因为业务的访问模式而出现热点问题。排序由于将所有的 key 都按序排列，相同前缀的请求很有可能会被集中在一个或者相邻的 partition 中，而这些请求则很有可能落在同一台机器上。而对于哈希来说，所有的 key 已经预先都打散了，它自然而然就没有这个问题。

不过，单纯对比两个方案的优劣意义并不大，还是要从业务出发。我们认为，在互联网的业务场景中，不同的 key 之间并不需要有一个偏序关系（比如两个用户的名字），所以在权衡之后我们采用哈希方案。

Hash schema 实现方式



在实现 HashSchema 的时候，我们遇到的第一个问题就是“数据怎么存储”的问题。“把一个 key 依据 hash 值落到一个桶中”，这只是一个理想化的抽象而已。在真实的系统实现上，你必须还得提供一层“表”的概念把不同的业务给分割开。有了表的概念后，我们在实现上就开始产生分歧了。具体来说，我们在存储上一共有“多表混存”和“分开存”两种方案。

所谓多表混存，就是将表 ID 以及 hash key 合起来算一个新的哈希值，根据这个哈希值从全局唯一的 partition space 里选取一个 partition 做存取。如上图的左半部分所示。

而对于分开存而言，把表的语义下推到了存储层。如上图的右半部分所示：每个表都有一个单独的 partition space，当有一个读写请求时，要先根据表 ID 找到对应的 partition space，再根据 hash key 去找对应的 partition。

那么这两种方案到底哪一个更好呢？从理论上来说，我们认为多表混存方案更好，因为它更符合软件工程中分层的思想；混存也更容易实现，因为只需要管理一份元数据，负载均衡也更为简单。但是从业务角度来看，采用各表分开存的方案更好。因为分开存意味着更简单的表间资源限制、表级别的监控和删除表的操作。考虑到运维上的误操作，分开存储也更有优势，即使你误删了一个 partition，该错误操作扩散给不同业务的影响也要少一些。

下表给出了我们对两种方案的对比。

两者相比，一个理论上更优美，一个实践上对业务更友好。最后我们还是从业务出发，我们放弃了理论更优美也更容易实现的方案，而选择了对业务友好的方案。

Hash schema 的负载均衡

	多表混存	各表分开存
元数据管理	容易	不容易
负载均衡	容易	不容易
表间的资源限制(quota)	不容易	容易
表级别的监控	不容易	容易
删表的操作	不容易	容易
误操作带来的影响	高概率影响多个业务	低概率影响多个业务

说完 hash schema，接下来就是负载均衡的问题。下面列出了一般分布式 KV 存储在负载均衡上的目标：

- 单个 Key 的请求过热：不论哪种存储方案都不太容易解决。如果是读，可以考虑再加上层的只读缓存；如果是写，只能让业务将 key 做进一步拆分
- 单个 Partition 的请求过热：在 hash schema 下无需考虑，因为 key 是 hash 分散的。
- Partition 的容量分布不均：在 hash schema 下无需考虑，因为 key 是 hash 分散的。
- Partition 的个数在不同机器上分布不均匀：这一点需要处理，而且这一点处理好之后，读写请求在不同 ReplicaServer 的请求就比较均衡了。

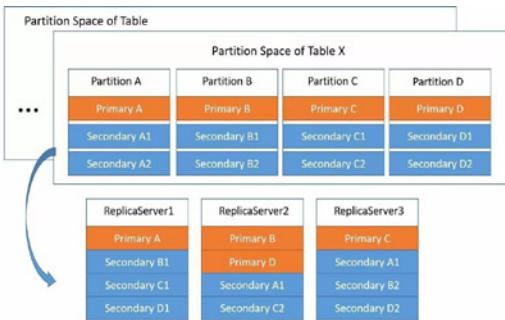
负载均衡 - 目标

具体来看，负载均衡的目标有两点：

A. primary 和 secondary 不能共享 ReplicaServer

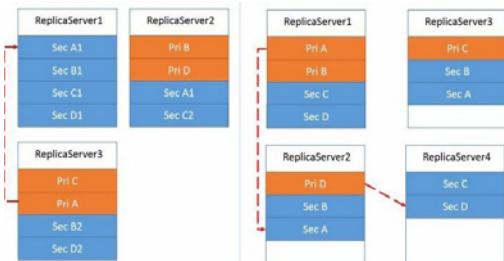
B. 对于每个表，primary 和 secondary 都能在不同的 ReplicaServer 上平均分配

上图是对目标 B 的一个简单说明：假如一张表有四个 Partition，而一共有三个 ReplicaServer，我们希望 12 个 Replica 的分布情况是 (1, 3), (2, 2), (1, 3)。



负载均衡 - 算法

在实现我们的负载均衡算法时，有一个很重要的注意点是：角色切换要优于数据的拷贝，因为角色切换的成本非常低。



对于这点的说明，可以参考上图中的两种情况：

- 在左图里，4个 Primary 分布在 ReplicaServer2 和 ReplicaServer3 上，而 ReplicaServer1 上没有任何 Primary。这时候，通过把 ReplicaServer3 上的 Primary A 和 ReplicaServer1 上的 Secondary A 做角色对调，就可以满足 Primary 的均衡。
- 在右图中，4个 Primary 在四个 ReplicaServer 上的分布是 (2, 1, 1, 0)。如果想要做 Primary 的均衡，需要把 ReplicaServer1 上的一个 Primary 迁移到 ReplicaServer4 上，但直接迁

移 Primary 需要拷贝数据。此时，我们如果引入中间节点 ReplicaServer2，先把 ReplicaServer1 的 Primary A 和 ReplicaServer2 的 Secondary A 角色互换，再把 ReplicaServer2 的 Primary D 和 ReplicaServer4 的 Secondary D 互换，就能实现 Primary 的均衡。

为了处理这些情况，我们对集群中 Primary 可能的流向建立了一个有向图，然后利用了 Ford-Fulkerson 的方法进行 Primary 的迁移调换。具体的算法不再展开，可以参见我们的开源项目代码。

在真实的负载均衡中，还有很多情况需要考虑：

为了更好的容错，不应该把一个 Partition 的几个副本都放在同一个机架上。

不同的 ReplicaServer 可能有不同的存储容量，副本个数的绝对均衡可能不很合理。

在做负载均衡的数据拷贝时，一定要注意限流，绝对不能占用过多的系统资源

在当前 Pegasus 的开源项目上，有部分情况还没有做考虑。后面在负载均衡上我们会持续做优化，大家可以持续保持关注。

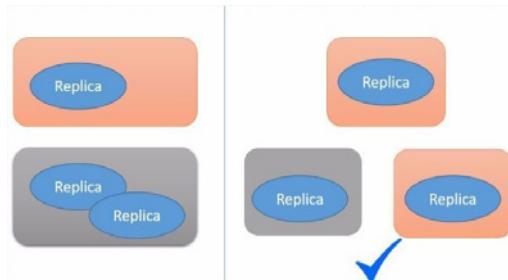
一致性和可用性

前面介绍了扩展性，接下来是一致性和可用性。我们在设计上的考虑前面已经介绍过了：

- 减少对 zookeeper 的依赖
- 数据不写在 DFS 上
- 多副本、用 PacificA 算法来保证强一致

当我们按照这些设计目的将系统实现出来，并准备找业务小试牛刀时，业务一句话就将我们顶了回来：“你们有双机房热备么？”

为什么要单独说这一点？因为对于一个强一致性的分布式存储系统而言，跨机房的容错是一件比较麻烦的事情：



一方面，假如把一个 Partition 的多个副本部署到两个机房中，一定会有一个机房拥有多数派。而这个机房的整体宕机，意味着集群不可服务（因为要保证强一致性）。

另一方面，假如把 Partition 的多副本部署的三个机房中，跨机房的安全性的确得到了保障。但是这意味着每一条写请求都一定会有跨机房的 replication，从而影响性能。

通过对这个问题进行反思，我们认为我们其实钻了一个“要做完美系统”的牛角尖。就业务的角度来看，作为一个完善的存储系统，的确要对各种各样的异常情况都需要做好处理。但随着异常情况发生概率的降低，业务对一致性的要求其实也是逐步放宽的：



- 在了解了业务的需求后，我们为 Pegasus 设计了多级的冗余策略来应对不同的风险：

- 一致性协议：用于单机房部署，抵御单机失效的风险
- 跨机房 replication：异步复制每个 partition 的 log，双机房都可写，基于 NTP 时间戳的最终一致性
- Snapshot 的定期冷备份：跨地域复制，发生极端故障时的兜底策略，可能会丢失部分数据

对上述冗余策略的几点说明：

- NTP 的时间戳可能不够精确。两个机房对同一个 key 的先后写，最终结果可能会是前者覆盖后者。目前我们并没有更进一步的做法来规避这点，因为我们的目标是“最终一致性”，即“随着时间的流逝，两个机房最后的结果是一样的”，所以即便是先写入的覆盖后写入的，最终在两个机房的状态表现也是相同的。
- 就我们在维护 HBase 的经验来看，业务两个机房同时写一个 key 的可能性是很小的。所以 NTP 带来的“先写覆盖后写”的问题，需要业务自己做注意。
- 多级的冗余策略可能会造成数据的备份数比较多。首先，并不是所有业务都需要这么多的冗余级别，而是根据具体需求进行在线配置的。所以这些限制冗余目前我们做成了表级别的可以修改的参数，而非集群级别强制的。再者，有些备份，如 snapshot 冷备，其存储介质是非常廉价的。
- 跨机房 replication 和 snapshot 冷备份的功能还在内部测试阶段，开源版本中还没有开放出来。

延时保证

最后介绍一下保证延时性能方面的问题。对于这个问题，有两点需要强调一下：

- 选择什么样的实现语言
- 怎样高效的实现一致性协议

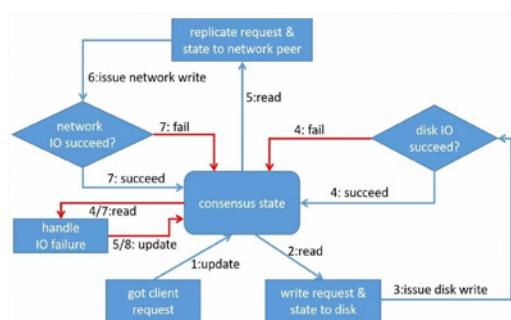
在实现语言上，我们选择了 C++。原因前面也说过，为了性能保障，我们必须采用没有运行时 GC 的语言。另一个选择可能是 Rust。在为什么选 C++ 而不选 Rust 上，我们的考虑如下：

- 当时项目立项时 Rust 还不太火，第三方库也远没有 C++ 完善。
- 只要遵守一些编程规范，C++ 还不算特别难以驾驭。
- 使用 C++ 招人更方便些

当然这个选择是偏保守型的，在语言层面的探讨也到此为止。重点还是说下我们一致性协议实现的问题。

就实践来看，想要把一致性协议实现的正确、高效，并且还要保证代码是易维护、可测试的，是一件比较难的事情，主要原因就在于一个完整的请求会涉及很多个阶段，阶段之间会产生 IO，再加上并发上的要求，往往需要对代码进行很细粒度的加锁。

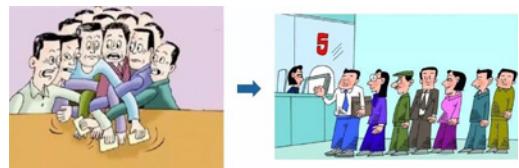
下图给出了一个完整写请求的流程简图：



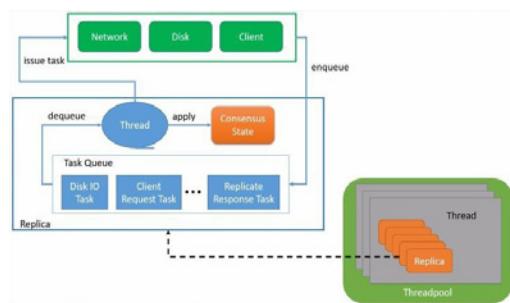
从上图可以看出，当客户端发起一个写请求后，这条请求会先后产生写本地磁盘、以及发送

网络 RPC 等多个事件，无论事件成功还是失败，都需要对公共的一致性状态进行访问或修改。这样的逻辑，会使得我们要对一致性状态进行非常繁琐的线程同步，是非常容易产生 bug 的。

那么这种问题该怎么处理呢？就我们这边的经验来看，是要把代码的组织方式从“对临界区的争抢”到“无锁串行化的排队”。我先用一张图来说明一下我们的这种代码架构：



具体来看，就是以“一致性状态”为核心，把所有涉及到状态更改的事件串到一个队列中，通过单独的线程取队列事件执行。所谓执行，就是更改一致性状态。如果执行过程中触发了 IO，就用纯异步的方式，IO 完成后的响应事件又会串到队列中来。如下图所示：



另外，为了避免创建过多的线程，我们采用了线程池的做法。一个“一致性状态”的数据结构只会被一个线程进行修改，但多个结构可能共享一个线程；replica 和线程的，是多对一的关系。

总的来说，通过这种事件驱动和纯异步的方

式，我们得以在访问一致性状态时避免细粒度的锁同步。CPU 没有陷入 IO 等待中，以及线程池的使用，性能方面也是有保障的。此外，这种实现方式由于把一个写请求清晰的分成了不同的阶段，是非常方便我们对读写流程进行监控的，这对项目的性能分析是非常有好处的。

Deterministic 测试

接下来我们重点讲一下测试是怎么做的。

分布式系统稳定的问题

当我们把系统做下来之后，发现真正长期困扰着我们的其实是如何将系统做稳定。

这个难题主要体现在哪几个方面呢？总结之后有以下三点：

- 难以测试，没有较有效的方法测试系统的问题。现在的经验便是将它当成一个黑盒，读写的同时杀任务，想办法使它的某一个模块出现问题，再去查看全局是否出现问题。然而这样的测试方法只能去撞 bug，因为问题是概率性的出现的。
- 难以复现。因为 bug 是概率性出现的，就算通过测试发现了问题，这个问题也不太容易复现，从而进一步对调试带来困扰。
- 难以回归。假如通过看 log、观察现象、分析代码找到了问题的症结。你的修复方法是不是有效也没有说服力。这样修是不是能解决问题？会不会引发新的问题？因为没有稳定复现的方法，这两个问题是很难回答的。

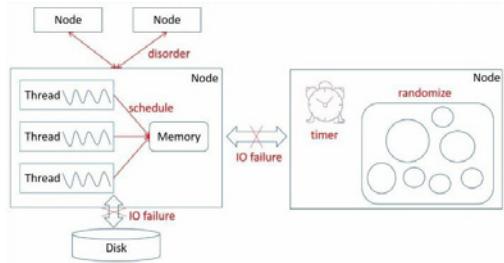
根源：不确定性

那么造成以上那些难点的根源在哪里？总结一下，我们认为是程序自身的不确定性。该不确

定性体现在两个方面：

程序自身的随机：系统 API 如调度、定时器、随机数的使用，以及多节点间的并行。

IO：程序可能要涉及到外设的 IO。IO 的概率性失败、超时、丢包、乱序等情况，也导致了程序不确定性。



用一个公式概括一下：

小概率的 IO 错误 + 随机执行路径 = 不容易复现的异常状况

那么对于这个问题，我们应该怎么解决？

既然在线上很难复现问题，那么能不能构造一种模拟的场景：在这个模拟场景里边，我们可以模拟 IO 错误的概率，也可以控制程序的执行顺序。再在这样的模拟场景里运行代码，如果逻辑真的出现了问题，我们就可以按照相同的执行顺序把问题复现出来。逻辑修改后，还可以进一步做成单元测试，这样难以回归的问题也就解决了。

当然这样描述这个问题，还是非常抽象。这里我举个简单的例子来说明下：

假设有这样的一个账户系统，里面有 Alice 和 Bob 两个人，两人账户的余额各为 100 元，分别存储在两台机器上，两人均可以向对方发起转账的交易。现在 Alice 要向 Bob 转账 5 元，最简单的实现是 Alice 把自己账号上的余额扣减 5 元，然后向 Bob 所在的机器发起一个增加

5 元的请求。等 Bob 的账号增加 5 元后，就可以通知 Alice 说转账成功了。

如果机器不会宕机、磁盘不会出故障、网络也不会出问题，那么这种简单的实现方式并无不可。但这样的假设绝对不会成立，所以为了应对这些方面的问题，我们可能加入一系列的手段来让账号系统变得可靠起来，诸如增加交易日志，把交易和账户信息备份到多个机器上，引入一些分布式事务的技术。这些手段的引入，会使得我们的系统变得复杂起来。

面对这样一个复杂的系统，任何一个异常的数据库状态都会使我们抓耳挠腮，比如 Alice 和 Bob 的账户信息的如下变迁：

(Alice_100, Bob_100) -> (Alice_105, Bob_105)

程序没有 bug 时，我们可以假定 Alice 和 Bob 账户的余额之和是等于 200 的。现在余额变成了 210，一定是某个环节出了问题。也许是两人同时向对方转账时，然后触发了什么 bug；也许是数据包被发送了两次。进入非法状态的可能情况有很多种，但硬件概率性的失败以及 Alice、Bob 间复杂（可能是并行）的转账记录会使得我们不太容易把产生非法状态的原因定位并复现出来。

而对这样的一个问题，我们的武器是模拟和伪随机。通过这两种手段，我们希望程序能按着可复现的事件序列运行。这样假如程序因为进入非法状态而 crash，我们可以对该状态进行复现、调试和回归。

还拿上面 Alice 和 Bob 的交易为例。一个双方同时转账的流程在模拟的环境里可能是这样运行的：Alice 发起转账 -> Bob 发起转账 -> Alice 发起写盘 -> Alice 发起 RPC -> Alice

RPC 成功 -> Bob 发起写盘 -> Alice 写盘失败
换句话说：

- 原本在并行执行的两套逻辑可能在模拟器里变成了并发交错的方式；
- 交错时候的调度选择是伪随机的，运行不同的模拟器实例，它们的状态变迁流程是完全不同的；但对于任何一个实例，只要选定了相同的随机数种子，其变迁流程就是确定的；
- IO 的失败也都是按照一定的伪随机概率进行错误注入的。

假设能有这样的一个环境，分布式业务逻辑的 debug 难度就一定会降很多。一方面，随机的存在使得我们有测试各种执行序列的能力；另一方面，伪随机和错误注入，使得我们可以复现遇到的问题。

控制不确定性

那么，具体的控制不确定是怎么做的呢？

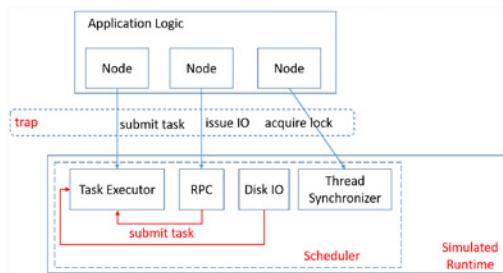
- 产生不确定性的接口提供抽象层。当前系统一共提供了线程池、RPC、磁盘 IO、线程同步和环境操作 5 种抽象接口。每种接口都有模拟实现（测试）和非模拟实现（部署运行），分布式业务逻辑（Application Logic）的代码只调用系统的抽象接口（Runtime）
- 系统提供能在单进程中模仿多个服务节点的能力，从而达到模拟分布式系统的效果。

Runtime 层的模拟实现是一个难点，重点包

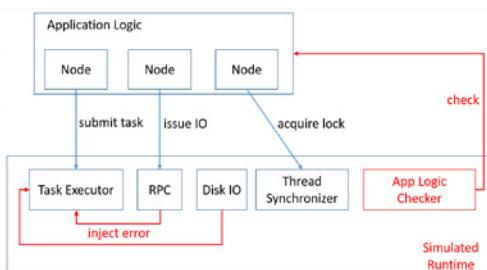
括：

- 只提供纯异步的编程接口，节点的运行模式就是线程执行事件队列中任务的过程。
- 当业务逻辑通过 Runtime API 调用到

Runtime 的模块内部时，Runtime 会将当前节点挂起，并按照伪随机的方式选取一个新的节点进行执行。整个过程请类比单核多任务的操作系统，runtime 就像内核态，分布式逻辑的 Node 就像用户态进程，runtime API 就像系统调用。如下图所示：



对于提交到事件队列中的 IO 事件，我们按照一定的概率伪随机的注入错误。另外，引入了一些全局的状态检测模块，在每次状态变化之时检测全局状态是否合法（比如检查 Alice 和 Bob 的账户余额之和是不是小于 200）；同时，Application Logic 的代码也加入了很多的 assert 点来检查 Node 状态是否合法。程序 crash 后，就可以用相同的随机数种子进行复现和调试。如下图所示：



Pegasus 的单元测试也是利用这套测试框架进行的。在单元测试中，我们会用一个脚本来描述为一个场景施加的动作以及预期的状态，上图的 App Logic Checker 的功能就是加载脚本，

然后按照脚本的方式检测程序状态是否符合预期。

Pegasus 所采用的这套测试框架，其理念和实现均来源于微软的开源框架 rDSN。整个框架较难以理解，这里也只是简述下其原理。大家如果感兴趣可以直接查看源代码。

现状和计划

现在 Pegasus 在小米内部已经稳定服务将近一年左右，服务了近十个业务。有关 Pegasus 的存储引擎、性能以及设计方面的更多阐述，大家可以移步 Arch Summit 2016 的另一次分享（文后有链接），也可以参考我们在 github 上的相关文档。

后续我们会把数据的冷热备份、Partition 分裂等功能都开源出来，请持续保持关注。也欢迎各路人士加入小米，加入我们团队。

写在最后

最后概括一下全篇的内容。

当我们在调研或者实现一个项目时，一定有三点是要关注的。

- 关注业务：项目是否满足业务需求。
- 关注架构：项目的架构是否合理，对于分布式存储系统，就是一致性、可用性、扩展性、性能这几点。
- 关注软件工程：项目的测试是否完善，代码的可维护性是不是很好，项目能不能进行监控。

Web协议优化指南



前言：Web 性能优化的问题

提到 Web 性能优化，那也就意味着我们遇到了很多性能方面的问题。

如下图所示，看到这一行字，我相信大家的脑海里一定会浮现出很多的场景画面，比如说无法接入网络，又比如说页面一直在加载，或者说内容可能一直卡在 99% 加载不出来，包括视频的卡顿、短视频以及直播，还有从 WiFi 切到 4G 网络出现重新加载的情况，那遇到这么多的 Web 问题，其主要原因是什么呢？我进行简单罗列一下，归纳为有以下两种情况。

直接和页面相关的页面大小，页面的元素类

型的多少，也就是说一个页面越大，元素越多，动态的交互越频繁，那相应的性能可能就会受到更大影响。页面优化问题也是前端优化的主战场。

网络环境以及端配置，包括用户手机硬件的性能、软件的操作系统版本，也会对性能产生影





响。而这两个主要和运营商以及用户的配置有关，也是我们很难施加影响的一部分。

网络协议

今天，我将着重介绍网络协议。网络协议是一个非常重要、非常关键、但又很容易被大家忽略的一个因素。那么提到网络协议，我们会遇到或者会用到哪一些网络协议呢？

接下来，我以现在最主流的互联网下一代 HTTP2 协议为例，来简单地介绍一下。



如上图所示，Web 请求需要经过的协议栈。该图左边是客户端，右边是数据中心，或者说服务端。

请求从客户端发出之后，首先会经过 HTTP1.1 协议。那为什么标题是 HTTP2 请求，但却提到的是 HTTP1.1 ? 这是因为 HTTP2 虽然

是一个全新的协议，但是它还是沿用了大部分 HTTP1.1 的语义。比如说 GET 请求、POST 请求，都是使用 HTTP1.1 的语义。对于页面或者对于 JavaScript 来讲，我们仍然使用着 HTTP1.1 的语义。HTTP1.1 下一层就到了 HTTP2，HTTP2 使用全新的帧控制语义，换句话说，HTTP2 frame 进行封装 HTTP1.1 的语义，比如说 GET 请求，它通过使用 HTTP2 的 HEADERS 实现封装；POST 请求的 body 数据使用 HTTP2 的 data frame 来实现封装。然后到达 TLS 协议，传输加密层，那这里为什么会有加密呢？

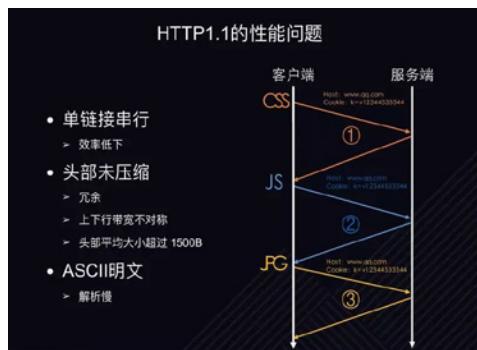
其主要原因是在 HTTP2 RFC7540 协议规定 HTTP2 有两种实现，一种是 H2，强制使用 TLS 进行加密；另外一种是 H2C，这里的 C 是 clear，就是明文，不需要加密。虽然协议是这么规定的，但是所有的主流实现，包括所有的浏览器、所有的操作系统、到目前为止都是使用 TLS 加密。也就是说，如果使用 HTTP2 就一定要使用 TLS 加密，所以在这过程中遇到 TLS 的问题。然后再往下是 TCP 层，再继续往下是 IP 网络层，以及以太网、运营商网络物理层，之后到达右边服务端协议上。两边端口可以认为是对称的，因此不再进行介绍。

那这么多协议，我们应该需要关注哪一些呢？或者说更明确地说，对我们大部分 Web 应用开发者来讲，我们需要关注哪一些呢？上图中白色虚线往上的部分是客户端可以施加影响或者优化的一部分，即协议上提出的包括 TCP、TLS 协议，以及再往上的 HTTP。明确了我们需要优化的协议，那么接下来我来分析一下他们都会有哪些问题。

HTTP1.1 的性能

HTTP1.1 的性能问题

以现在使用最广泛、历史最悠久的 HTTP1.1 协议说起。



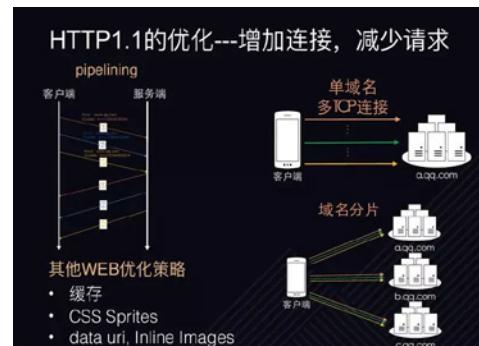
接触 Web 开发的同学都知道，HTTP1.1 最大的性能问题就是单链路串行。当一个 TCP 链接上如果有多个请求。如上图所示。请求只能一个接着一个发送，这是它最大的问题。

第二个问题是头部未压缩。头部，尤其当 HTTP 请求 Cookie、Host，其浏览器型号都一样，如果每次请求携带相同的信息，这显然是重复的。这是冗余，浪费带宽，并且从性能问题上会影响用户的访问速度。为什么呢？因为运营商的网络上下行带宽严重不对称，上行带宽很可能只有下行带宽 $1/10$ 甚至小于 $1/10$ ，也就是说，若其带宽有十兆或者一兆，那么它上行链路可能只有 100K 甚至几十 K。如果头部平均大小是 1500 个字节，其一次传出可能有十个头部，并且在带宽层面会影响用户访问性能。因此在 3G、4G 的移动网络环境下，头部未压缩问题显得更为重要。

HTTP1.1 的优化

优化的手段、优化的策略非常多，也非常有效，那么有哪些呢？

概括来讲，HTTP1.1 的优化方向主要是两点。



第一点是增加并发连接数量；第二点是减少往外发出的请求数量。如上图，不管是单域名多 TCP 连接，还是域名分片，使用多个域名都是为了提升并发连接的数量。需要注意的是并发连接不是并发请求。因为 HTTP1.1 尝试在单链接上实现并发请求，但是失败了。比如 pipelining，因为它存在队头阻塞。所以它只能通过提升并发连接的数量来提升用户的访问性能。此外，也有其他一些优化策略，包括缓存、CSS Sprite、data uri，图片内联等。其本质都是为了减少发出请求的数量，即将多个响应放在一个响应里面返回，以此减少请求数量。

以上这些 HTTP1.1 优化策略和手段在 HTTP1.1 时代取得非常好的效果，并且也行之有效。

HTTPS/HTTP2 加速 HTTP1.1 的淘汰

随着 HTTPS 的全站趋势越来越明显，HTTP2 正式发布渐成主流，HTTP1.1 的优化策略随之失效了。为什么这么说呢？

HTTPS 性能的一个特点或者说缺点是连接成本非常高，如果 HTTP1.1 使用增加并发连接的方式来提升性能，由于其连接成本很高，那么并发连接的并发成本也就会高，反而降低了性能。

HTTP2 的特性是支持多路复用。在一个链接上允许发出多个请求，允许并发请求。那 HTTP1.1 减少请求数量就显得有点多余，相反地，有可能会降低缓存命中率，降低性能。

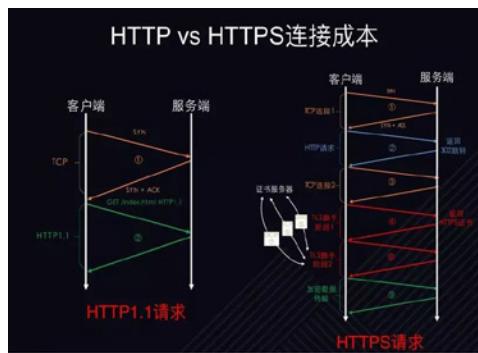
所以从这个层面来讲，HTTP1.1 的优化策略有可能行不通，并且产生副作用的。

HTTP 与 HTTPS 的比较



HTTP VS HTTPS 连接成本

接下来我简单介绍一下 HTTPS 的连接成本为什么会高？从协议层面，它的连接成本高在哪里？



如上图，左边图是一次 HTTP1.1 的请求过程或者说是成本。HTTP1.1 请求非常简单，只需要建立 TCP 连接，然后发送 HTTP 请求和响应

就完成了。总过程只需要两个 RTT 就可以实现一次 HTTP 请求。

右边图是一次 HTTPS 的请求。首先 HTTPS 通过三次握手建立 TCP 连接；然后发送 HTTP1.1 请求。这里为什么是 HTTP1.1 的数据呢？因为如果让用户主动输入 URL，大多数用户不会主动输入 HTTPS，比如说，访问腾讯网，他们不会输入 <https://www.qq.com>，而是输入 www.qq.com 或者 <http://www.qq.com>。为了强制用户使用 HTTPS 就会返回一个 302 跳转，关于 302 跳转，HTTPS 使用的端口是 443，HTTP 是 80；新的端口就必须重新建立一个新的 TCP 连接，这样又多了一次 TCP 连接的建立过程；建立 TCP 连接之后，开始进行 TLS 握手阶段。

TLS 握手有两个阶段，完全握手阶段一，协商协议版本、密码套件、请求证书、校验证书；客户端拿到证书之后即使证书的签名没有问题，证书时间也有效，但是仍然需要校验证书的状态，因为可能存在主动撤销证书的情况，证书的私钥也有可能被泄露或者说 CA 被攻击，所以查询证书状态是必要的。查询证书状态就是 OCSP，在线证书状态检查，查询证书状态需要解析个 CA 的三点 DNS，又需要建立 TCP 连接，然后又需要建立包括 OCSP 的请求响应，通过 HTTP 请求完成，三次握手、状态没问题之后，开始进行 TLS 完全握手阶段二，协商对称密钥，即非对称密钥交换计算；完成之后，整个 TLS 过程协商完毕，开始发送 HTTP 加密数据。至此达到第九个 RTT，相比 http1.1 多出 7 个 RTT，这是什么概念呢？

未经优化的 HTTPS 速度明显慢于 HTTP

如下图，现在最好的 WiFi 网络环境下平均

RTT 是 70 毫秒，7 个 RTT 就是 490 毫秒，4G 是 100 毫秒，3G 是 200 毫秒，7 个 RTT 就是一秒多钟。且这仅仅是一次请求，一个页面一般会有多个请求，并且请求之间还可能有依赖，产生阻塞。以此情况下，一个页面多出几秒钟是非常正常的现象，且该耗时仅为网络耗时；而由于协议的规定，它必须要进行网络交互，还包括很多其他的耗时，比如说计算耗时，证书校验、密钥计算、内容对称的加解密等计算，由于移动端性能相对要弱，所以在计算层面多出几百毫秒也是非常正常的现象。



经过以上分析，我们可以得出一个基本结论，就是未经优化的 HTTPS 要比 HTTP 慢很多，那么 HTTPS 是不是就是 slow，就是慢的意思呢？

为了找到慢的原因或者说找到性能的瓶颈，以上我们仅仅从网络协议理论进行分析，而事实上，在实验环境下、业务真实环境下、用户场景下面，它是不是还是这么慢，又慢在哪里呢？

为此，我们进行了两个层面的分析。

线下模拟测试

第一个就是线下模拟测试。线下模拟测试主要针对移动端，因为移动端手机性能是流量越来越多的、都是往移动端倾斜。移动端有两个特点，第一，手机屏幕比较小，查看数据或者查看页面

性能，调试日志也不太方便；第二，不太方便运行数据的分析脚本、控制脚本。所以我们将手机和 PC 使用 USB 连接起来，通过远程调试协议来控制手机上的浏览器来不断的重复访问我们所构造的不同的页面。该页面所涉及的元素一定是非常多的，不同的类型也都有。

因此我们进行线下模拟测试一定要自动化，靠人为实现是没有效率的，并且也测试不了那么多的页面和场景。而且要注意数据的同比、环比，如果没有，即使测试拿到了一百条数据，一千条数据也是不能说明问题的，因为数据误差尤其多，周期性的同比、环比要求数据超过一万条才认为它可能有一定的说明意义。

另外一个大误差是 WiFi。即使是在洲际酒店、腾讯大厦，或者朗科大厦，使用的 WiFi 还是经常出现网络抖动的情况，这对数据、对协议的分析有巨大影响，因为协议很可能仅仅就是那么一个 IT，那么几百毫秒的差异，在抖动的网络环境下很容易出现问题干扰结论。所以为了排除这些误差，我们也经常将手机和 PC 用 USB 连接起来，绕开 WiFi 直接使用有线网络，因为有线相比无线，特别是 WiFi，要稳定非常多。

关于工具的话就是 traffic control，因为我们真实用户的网络环境是千差万别的。不同 IP，不同的带宽，不同的丢包率，因此我们采用该工具来模拟在实验室里模拟环境。线下模拟数据，显然不能代表我们真实业务的性能。

线上业务速度数据采集

第二个部分是线上分析数据。

在服务端进行数据采集的方案有两个优势。

第一点它能采集到客户端采集不到的数据，采集到更低层信息和业务信息。如上图，左边是客户端，即可用手机、STW 或者是负载均衡器，

右边是业务服务器。关于业务信息，通过 LB 和业务服务器的交互可以获得业务整体处理时间，这是客户端拿不到的。第二点关于协议信息，包括 TCP 的 RTT、TCP 是否连接复用、是否 TLS SESSION 复用、TLS 协议版本，密码套件、握手时间以及非对称密钥交换计算时间，HTTP2 头部压缩比等，这些都是客户拿不到的信息。

Why Slow? 线下模拟测试

- 自动化
- 消除误差
 - 同比, 环比, 10000条
- 工具
 - Chrome Remote debug
 - Linux traffic control
 - performance timing api



有些信息客户端能采集到，但客户端的开发成本很高，因为安卓有不同的版本，比如有 IOS、有 windows，需要针对不同操作系统进行开发，而在服务端只需要一次开发即可，将数据放到 COOKIE 里返回给客户端，客户端通过 JS 或者普通页面就能获取到信息，拿到信息之后，将协议相关的信息组合起来放到一条数据内统一上报到数据平台进行分析。

那么拿出这么多数据，我们数据平台如何去分析呢？

多维数据分析

如下图，由于数据庞大，仅截取几条进行分析。左边是数据维度，比如 TCP-reuse 表示 TCP 连接复用，TLS1.2 表示 TLS 协议版本是 1.2；右边是与用户性能相关的监控指标，比如说开始加载时间，页面可活动时间，业务处理时间等维度，这些维度也可互相组合并得出三四百个维度。如上图所示，腾讯 X 五内核浏览器在

4G 网络下使用 HTTP2 并且是使用 TLS1.2 协议并且使用 ECDHE 并且没有复用 TLS Session 的首屏时间是多少？通过多维度对比，我们很容易就可以区别出它慢的因素，X5 在 4G 是这么多，那么在 3G 下面是多少，WiFi 下面是多少呢？通过多维度综合的对比就能从表面看出瓶颈在哪里。

Why Slow? 多维数据分析

维度	start_time	end_time	load_time	idle_time	first_paint	idle_time	idle_time
tcp_reuse	705	710	3181	3635	2730	147	147
TLSv1.2	966	982	1987	1152	2391	145	145
tcp_first_use	1492	1495	2956	1618	3044	148	148
ecdh-rsa-aes128-gcm-sha256	975	984	1973	1140	2368	143	143
android_tcp_first_use	1774	1844	3024	1772	3419	142	142
android5_tcp_first_use	999	1040	2648	1217	2461	89	89
ios8_tcp_reuse(http)	398	402	1571	443	897	180	180

腾讯X5内核浏览器在4G网络下使用HTTP2并且是TLS1.2协议并且使用ECDHE并且没有复用tls session的首屏时间是多少？

因此多维度数据分析，最终目的就是为了找到性能瓶颈以及速度优化方向，那么有哪些优化方向呢？

Web 访问速度优化方向

优化方向有三个。第一个协议；第二个资源；第三个用户行为。

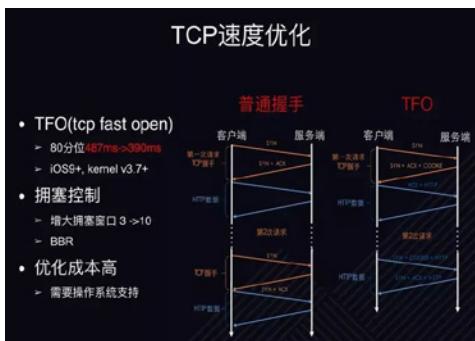
协议

TCP 速度优化

那么协议层面如何进行优化呢？协议的最底层是 TCP，而 TCP 最显而易见的性能问题是需要三次握手才能建立一个 TCP 连接，然后发送应用层数据，与普通握手相比，浪费一个 RTT。如上图左边是普通握手。每个 TCP 的连接需要建立握手，并且发送 SYN 包没有任何运动组数据，RTT 就浪费了。

TFO 是 TCP FAST OPEN。在 SYN 包发出的

同时将应用层数据发出来，然而它的前提是第一次建立握手、建立连接的时候也需要发 SYN 包，不同的是，服务器返回 SYN+ACK 时会返回一个 COOKIE，在这之后用户发起 TCP 连接，发出 SYN 包的同时会带上这个 COOKIE，然后可以直接带上 HTTP 数据。换句话说，在 SYN 包发出的同时将应用层数据一起发出来，减少了一个 RTT 对性能的影响。TCP FAST OPEN 性能是一个收益数据，我们发现 80 分的数据提升了将近一百毫秒。然而它的缺点是需要操作系统的支持，需要 iOS9+ 或者 linux kernel3.7+，并且不支持 Windows 系统。另一个优化方案是拥塞控制，将三个拥塞窗口增加到十个。



总的来说，在 TCP 协议层面来提升速度优化的空间有限，因为优化成本非常高，而高在哪里呢？它需要操作系统、需要内核的支持，如果仅仅是服务端升级，我们支持与研发，其成本还能够控制，但是最重要的是需要用户的操作系统升级，需要广大网络设备上的软协议栈或者操作系统升级，而在这个层面部署的压力非常大。所以 TCP 层面优化成本非常高。

TLS 速度优化: session reuse

TLS 是加密层，加密层最大的问题是完全握手。关于完全握手，接触过的同学应该很清楚它的 session ID，它的过程我就不做介绍了。



在这里，我主要分享两点。如上图。第一点，关于 iOS Qzone 优化，通过 session id 握手时间大概能够提升 50%；第二点，session ticket 的机制虽然更加先进，因为它不需要服务端提供缓存去提供内存存储，发送 ticket，服务端进行校验即可。而 session ID 需要提供内存进行存储，然后查找到 session cache 是否命中，由于 iOS 不支持 session ticket，所以大家一定要注意通过分布式 session cache 来提升 iOS 的 session ID 的缓存命中率。很多场景是没有办法实现简化握手的，比如说用户第一次打开 APP、打开浏览器，又比如说用户退出了再重新重启浏览器，或者说把浏览器页面关闭掉，再打开，都可能会导致 session cache 简化握手无法实现，因为 session ticket 都是基于内存的，而不是存储在硬盘里面。

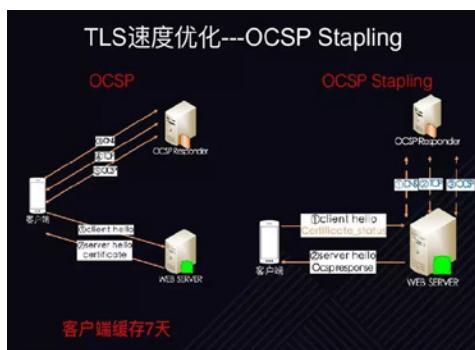
TLS 速度优化: False Start

针对完全握手的优化方法是 false start，即抢跑。与 TFO 思路类似，如上图左边是普通握手，必须要进行 TLS 的两个 RTT 四次握手才能建立连接，发送应用层数据。false start 是在 clientKeyExchange 没有交换的时候，即在完全握手的第二个阶段将应用层数据一起发出来，提升了一个 RTT。那怎么启用 false start



呢？其实现在客户端基本都已经支持了，如果服务端要启用需要注意将支持 PFS（完美前向密码）加密的算法配置在前，比如说 ECDHE、DHE 算法配置在前即可，并且其握手时间将提升 30%。

TLS 速度优化——OCSP Stapling OCSP Stapling。刚才提到的一些场景就是主动撤销证书，所以一定要进行证书状态的检查，因为在某些情况下，单纯地进行校验证书的签名、校验证书的时间是发现不了证书的状态，它需要更加实时的检查。我们证书颁布 3 年，中间可能出问题，所以他会进行周期性的检查。



他普通的过程与之类似，client hello 获取到证书时，③④⑤这过程中增加的 3 个 RTT 需要去 CA 站点请求 OCSP 的状态和内容。如上图，右边是 OCSP Stapling，它的过程相当于服务器代理实现了 CA 的证书状态颁发的功能，它提前向 CA 站点请求 OCSP 内容，并保存在本地

服务器端，然后在握手的同时，将像签名的内容返回给客户端，然后客户端对该内容进行校验。它不需要直接和 CA 站点进行交互，因此这样看来 OCSP Stapling 减少了三个 RTT，效果应该非常明显。

但是，事实上不是这样的，为什么？因为客户端会对 OCSP 进行缓存，没请求一次就可能可以缓存七天，也就是在这七天时期，如果用户访问该网站次数非常多，可能成千上万次，因为一个页面有非常多的请求，所以可能只有 1% 的概率来增加这三个 RTT。大家需要注意的，并不是一定就有这个效果。

TLS 速度优化——dynamic record size

现在来分析一下 dynamic record size，即记录块大小的动态的调整。



先简单讲解一下 TLS 协议层面的队头阻塞，即 head of line blocking 问题产生的背景，因为 TLS 协议传输的最基本单位是 record，一个记录块最大不能超过 16K，而一些服务端的实现，比如说 Nginx 最早期版本默认大小 16K。那么 16K 会产生什么情况呢？如果你中间丢了一个字节，或者说由于 TCP 的筹码丢了一个 segment，将导致这 16K 没有办法被校验，就算接收到 15.99K 的数据，也没有办法处理，因为一个数据的丢失、一个包的丢失，导致整个

record 没办法被处理。

那怎么解决呢？方法有两个。

第一个就是适当的调小 buffer，比如说改成 4K，不用实名制，即使丢失了也只影响这 4k 数据，而不是 16K 数据。

第二个，借鉴 TCP 的 slow start 原理。cloud flare 提供一个 patch，在 TLS 连接刚刚建立的时候，由于不知道网络状况以及拥塞情况，将 record 设置缩小，比如变成 1K，在发送速度提升之后，再将 record size 设置为 16K。它大概是这么个思路，也有开源代码，大家有兴趣的话可以了解一下。

刚才提到这些优化策略都是针对 TLS1.2 以及之前的版本。接下去介绍 TLS1.3 版本，这是一个革命性的、非常创新的、具有里程碑意义的协议。

TLS1.3 速度优化

TLS1.3 速度优化——ORTT Handshake

在性能方面的创新点是它能实现 1RTT 的完全握手，以及实现 ORTT 的简化握手。也就是说，TLS 连接层面没有握手原则不会增加 RTT 的延时。TLS1.3 现在仍处于草稿阶段，这里介绍它，是因为如果大家想尝鲜已经可以使用了，因为 OpenSSL1.1.1, Nginx1.13.0 已经分别支持 TLS1.3 的最新的草稿 draft20。而且 TLS1.3 上的最新的讨论，可能将于今年秋季正式发布。如果客户端是自己可以控制的，那可以尝试，包括 Firefox 也支持 TLS1.3。当然这些都是草稿版本，所以这里给大家介绍一下，具体的过程其实也比较复杂，我这里就不做详细介绍了。

HTTPS 速度优化

HTTPS 速度优化——HSTS 减少 302 跳转

HSTS 其实是 HTTP 的头部，它的作用是服务端告诉客户端，你以后访问我的网站，在指定的时间内，你必须使用 HTTPS，你不要使用 HTTP。然而他存在一个问题，HSTS 是通过 HTTP 返回的，很有可能被中间者所劫持。也就是说，客户端可能永远不知道服务端曾经发送 HSTS，也就没有办法使用 https。这样的话，有一个薄弱的预加载名单机制，比如说 chrome 浏览器将把你的网站内置在白名单里，当你发出访问请求时，它会默认使用 HTTPS；客户端同理。

HTTPS 速度优化——SPDY&&HTTP2

SPDY 和 HTTP2。为什么会提到 SPDY？主要有两点。

第一点，因为现在 HTTP2 很流行，并且正式发布了，但是它的所有特性和最主要特性都是沿用于 SPDY，除了头部压缩算法，所以我最开始研究的也是 SPDY 的协议，而且现在可能很多人只知道 HTTP2 而不知道 SPDY。所以也是为了向该协议致敬，因为它是鼻祖。

第二点，大多数老客户端只支持 SPDY，比如说安卓 4.4 以前版本，以及 IOS 现在还支持 SPDY，为了兼容老版本，提升他们性能同时支持 SPDY 和 HTTP2。

它有三个特性：

第一个，多路复用。在单个链接上同时发送多个请求，并且请求和请求之间在协议层面可以不依赖也可以依赖。比如说第二个请求先处理完了可以提前返回，相比较 HTTP1.1，它也做过 pipelining 的努力，它允许客户端同时发出多个请求，但是服务器必须按照严格顺序返回，否则就会乱套。它不知道哪个请求和哪个响应对

应的。这样会导致即使第三个请求先处理完了也不能提前返回；或者说四个请求都处理完，但是第三个请求丢了，那整个顺序也就出问题了。HTTP2 最强大的特性就是多路复用，它能将一百个请求甚至设置一百多个请求在一起发出去。

关于 SPDY 和 HTTP2，大家可能容易忽略的两点。第一点头部压缩，压缩的数据、宣传的页面大概会有 89% 或者 90% 的压缩比，但事实上是这样的吗？通过测试发现在一个 TCP 连接发送第一个请求的时候，当时的 HTTP2 压缩比只有 30%，发生第二个请求的时候压缩比能达到 60%，发生第三个请求以及之后才可能达到 90%，因为 SPDY 使用的是基于 zlib 压缩算法，而 HTTP2 使用的是基于 Hpack 压缩算法，他们都是基于状态空间进行压缩，因此如果信息越冗余、越重复，当你在这个连接上重复性阅读，压缩比才会越高。所以第一个请求发生时，头部没有那么冗余，压缩比可能就只有 30%。

这给我们性能优化方面的一个启发就是如果一个页面要放到下一个页面，我们可以提前给页面或者域名的连接发送两个空请求，积累数据。当真实的用户请求发起的时候已经达到了第三个合作之后了，以至于用户的头部压缩比就能达到 90%。

第二个是 server push。由于 Nginx 不支持 server push，所以现在这个应用在国内用得还很少，但它作用确实很大。比如说客户端请求一个 html，html 里面含有 css 和图片，按照正常来讲，解析 html 之后要分别发出 CSS 的请求和图片的请求，但是如果服务端得知页面支持 server push，客户端便只需要发出 http 请求，而服务器直接将 css 和图片一起发出去，以致请求多个响应未发先至。这就是 server

push 的作用，和 inlining 有点类似，但是相比 inlining 有两个好处，inlining 会影响缓存，会增大 html 的体积，包括后台模板的维护，这也便增加开发和维护成本，对于客户而言仅仅是多个请求。

HTTP2 实践建议

关于 HTTP2P 的实践建议以及原理性的知识，在网上资料已经非常多了，所以我主要是告诉大家一些我们自己在实践中遇到的一些经验和心得。

第一，使用一个连接或者说使用更少的链接。不用使用很多连接，为什么？第一，连接少，握手成本就少。第二，压缩比高，因为一个连接上积累的信息很多，压缩比会更高。第三，更好地利用 TCP 的特性，为什么？因为 TCP 的拥塞控制流量控制都是基于单个连接的，如果使用很多个连接，特别是在网络拥塞的情况下，会放大拥塞的系数，加剧网络拥塞，如果使用一个连接，当得知该窗口已经拥塞，响应很慢便不会继续发送了，但是多个连接的情况下，可能大家都会尝试发一下，而导致加剧网络拥塞。

第二，使用更少的域名，这也是为了更好地使用连接，并且减少了 DNS 解析时间。

第三，如果一定要使用很多域名，那就尽量将多个域名解析到相同的 ip，使用相同的证书。因为客户端实现的角度标准就是如果多个域名能够复用相同的 ip 和相同的证书，那我就会复用这个连接，我不会新建连接。比如 chrome 浏览器；灵活运用 server push，代替 inlining；关于 TLS1.2，若其默认使用 TLS1.2 之前的版本，HTTP2 便没有办法使用。TLS1.2 并不是支持所有的密码套件，它有很多密码套件属于黑名单

单，这也是在 HTTP2 现在的使用率比较低的一个原因；HTTP2 并不能解决所有问题，它适用于多元素、多连接的场景，如果你的页面本身很简单，只有几个请求，就用普通 TLS 也没问题，它并不能提升该场景下的访问性能。但是总的来说肯定还是推荐大家使用。

用户行为

预建连接

预建连接是一个最简单、最有效的速度优化的方案，为什么？预建连接的思路是在用户发起真正的连接之前，提前建立好连接。这样的话，对于用户而言，由连接所导致的成本和延时是感觉不到的。之前的一系列优化方案，即使是 0 RTT 握手，也存在一些计算，比如 ticket 的校验。那怎么样来预建连接呢？共有两点方法。

第一点通过 link 头部标签，比如说连接告诉浏览器，即服务端，可以返回；或者告诉浏览器，页面可能需要访问下一个页面，那提前跟我建立好预连接。这是支持 link 头部的浏览器可以实现的。如果不支持该特性，那怎么办呢？

第二点通过控制页面 JS。比如说，后台页面上有个 JS，对于即将访问的页面，可以提前与它建立好连接。当用户发起请求时，JS 已经提前给它预建连接了。还有很多场景，第一个是首页可以提前预建子页面连接，比如说百度首页，百度首页很简单，它的搜索结果千差万别，但是搜索结果的网站也可以固定好 AA.com，这样便于建立连接；第二个 QQ 空间，有的用户打开 QQ 空间可能会相册，有些用户会经常访问商城挂件、商城访问礼物，那我们就可以根据用户经常访问的页面给它提前建立好不同的直接预连接，预建好的连接也有可能会超时断开。

那么如何避免呢？可以使用长连接保持。比如说浏览器 HTTP2 超过三分钟就会断掉，HTTP 超过五分钟就会断掉，为了维持住长连接，我们可以使用 JS 周期性给这些页面发起长连接保持的请求，这样的话，连接相当于一直会保持住用户在访问时候建立好的状态，这便是预建连接的一个好处。



关于 HTTPS 速度的一个基本结论是 HTTPS 的访问速度可以超越 HTTP1.1。其最关键、最核心的一点是 HTTPS 可以使用多路复用，而 HTTP1.1 不行。关于优化手段节点，如上图，这是两年前的数据，它们的优化策略是一样的，包括 HTTP2、SPDY 都没有本质区别。

HTTP2 是未来吗？

之前提到 HTTP2 的一些强特性，对我们非常有帮助。如今 HTTP2 被认为是互联网下一代协议，但是它真的是我们未来吗？或者说，HTTP2 是下一个十年最具有性能权威、最具有统治力的一个协议吗？回答是肯定的。是。因为它的很多特性，比如多路复用、头部压缩、server push、优先级等，设计非常先进，性能也及其优良，解决了很多性能问题。

但回答也可以说不是。为什么？因为现在的 HTTP2 是基于 TCP 协议和 TLS 协议而构建，存在着些问题。

首先 TCP 协议 3 次握手。虽然它可以支持 TFO，但是 TFO 需要操作系统支持。并且 TFO 第一次建立连接获取 COOKIE 时还需要一次额外的 RTT 才能实现握手。此外 TLS1.3 支持 ORTT 握手，但 TLS 对 TCP 头部没有进行验证，可能存在被篡改的风险，比如修改窗口数、修改序列号等，都存在被劫持的可能。



而最大的问题是 TCP 的队头阻塞，比如说当我们传输 segment 时，如果第二个 segment 丢了，那便传输失败，为了保证它的可靠性和有序性，需要等到第二个 segment 到达才能往上传给应用层，而糟糕的是，由于 HTTP2 多路复用的特性，如果将假设的 50 个请求在一个 TCP 平台上发送，它会加剧队头阻塞的问题。关于 TCP 的重传，重传的序列号和最原始的序列号是同一个序列号，这会导致重传的模糊性问题。关于拥塞控制，需要操作系统升级，但是 TCP 的升级成本高，需要用户、网络设备中间商去升

级来支持。

所以从协议层面来讲，HTTP2 不是下一个十年最具有统治力或者最有心的、有权威的一个协议，那什么才是非常具有竞争力的一个协议呢？它就是 QUIC 协议，即使用 UDP 实现 HTTP2。以下是它的特性。

拥抱 QUIC 协议



它支持 HTTP2 的所有特性，比如多路复用、队头阻塞、头部压缩、server push；支持使用 TLS 1.3 的 ORTT 握手；使用 UDP 传输使用；基于 packet 加密，对 packet 头部进行一致性的验证，一旦发生修改即可发现。以上我所提及到的协议优化，腾讯云也都进行了很好的支持。另外，我们腾讯云 CLB 也正式支持了 QUIC 协议，平均性能提升了 15% 以上，欢迎大家试用。

罗成，腾讯 TEG 基础架构部高级工程师。目前主要负责腾讯 stgw（腾讯安全云网关）的相关工作。对 HTTPS, SPDY, HTTP2, QUIC 等应用层协议、高性能服务器技术、云网络技术、用户访问速度、分布式文件传输等有较深的理解。

李维博士：NLP助力电商智能化的台前幕后



NLP 入门

自然语言很复杂，自然语言处理（NLP）没有捷径。所谓 NLP 技能速成训练，除非指的是浅尝辄止，或所面对的是浅层的粗线条任务，否则基本上是自欺欺人。我有一个五万小时成精的定律，是这样说的：

“NLP 这玩意儿要做好（精准达到接近人的分析能力，鲁棒达到可以对付社交媒体这样的 monster，高效达到线性实现，real time 应用），确实不是一蹴而就能成的。这里有个 N 万小时定律。大体是：

NLP 入门需要一万小时（大约五年工龄）；

- 找到感觉需要两万小时；

- 栽几个有意义的跟头需要三万小时；
- 得心应手需要四万小时；
- 等你做到五万小时（入行 25 年）还没被淘汰的话，就可以成精了。”

——摘自李维博客《聊聊 NLP 工业研发的掌故》

对于急功近利的人，这仿佛天方夜谭，但我想说的是，这是一条非常漫长的道路，然而并非深不见底。作为“励志”故事，《梦想成真》描述了我的真实经历和心路历程。我曾自嘲说：“不知道多少次电脑输入 NLP，出来的都是‘你老婆’。难怪 NLP 跟了我一辈子，or 我跟了 NLP 一辈子。不离不弃。”

其他关于我自己与 NLP 的故事，我有个专

门系列，可以在【立委 NLP 频道】查看《关于我与 NLP》。那里还有 NLP 历史上的一些有趣掌故，有兴趣的同学也可以浏览。

NLP 要做深做透，要接近或达到类似人的深度解析和理解是一个艰难但并非不可能的历程，但我并不否定速成培训的功效和可能。毕竟并不是每一位想做点 NLP 的 AI 后学或同好，都有那个时间条件和需要去成为 NLP 的资深专家，很多时候就是要解决一个具体的浅层任务，譬如粗线条的分类和聚类。

这时候，通过开源资源和标准测试集自我培训的方法至少可以训练一个人使用开源工具的能力，如果赶上面对的任务相对简单，而且不乏大量带标数据（labeled data），也可能会很快做出可用的结果。典型的例子有对于影评做舆情分类，这种限定在狭窄领域的任务，利用开源工具也可以做得很好。

事实上，18 年前我的两位实习生，现在也都是业界非常有成就的人物了，他们的暑期实习项目就做到了非常漂亮的影评舆情分类结果，当时用的就是基本的贝叶斯机器学习算法。对于后学，除了拿开源练手外，也不妨浏览一下我开设的《NLP 网上大学》，或可开阔一点眼界，看到一些潮流以外的 NLP 风景。

NLP 架构

这次我参加的大会是 ArchSummit 全球架构师峰会，咱们可以多从 NLP 架构角度说说。

对于自然语言处理及其应用，系统架构是核心问题，我在《立委科普：NLP 联络图》里面给了四个 NLP 系统的体系结构的框架图，从核心引擎直到应用。

最底层最核心的是 deep parsing，就是对自然语言的自底而上层层推进的自动解析器，这个工作最繁难，但是它是 NLP 系统的基础赋能技术。解析的关键是把 非结构的语言结构化。面对千变万化的语言表达，只有结构化了，句型（patterns）才容易抓住，信息才好抽取，语义才好求解。这个道理早在乔姆斯基 1957 年语言学革命提出表层结构到深层结构转换的时候，就开始成为（计算）语言学的共识了。

接下来的一层是抽取层（extraction），这一层已经从原先的开放领域的 parser 进入面向领域应用和产品需求的任务了。值得强调的是，抽取层是面向领域语义聚焦的，而前面的解析层则是领域独立的。因此，一个好的架构是把解析做得很深入很逻辑，以便减轻抽取的负担，为领域转移创造条件。

有两大类抽取，一类是传统的信息抽取（IE），抽取的是事实或客观情报：实体、实体之间的关系、事件等，可以回答 who did what when and where（谁在何时何地做了什么）之类的问题。这个客观情报的抽取就是如今火得不能再火的知识图谱（knowledge graph）的技术基础，IE 完了以后再加上一层挖掘里面的整合（业内叫 IF：Information Fusion），就可以构建知识图谱了。

另一类抽取是关于主观情报，舆情挖掘就是基于这一种抽取。细线条的舆情抽取不仅仅是褒贬分类，竖大拇指还是中指，还要挖掘舆情背后的理由来为决策提供依据。这是 NLP 中最难的任务之一，比客观情报的抽取要难得多。抽取出来的信息通常是存到某种数据库去。这就为下面的挖掘层提供了碎片情报。

很多人混淆了抽取和下一层的挖掘，但实际

“今后的 10 年才真正是 NLP 的黄金时代，全面开花结果可以期待，尤其在情报挖掘、知识图谱、人机交互和智能搜索方面。”

上这是两个层面的任务。抽取面对的是一颗颗语言的树，从一个个句子里面去找所要的情报。而挖掘面对的是一个 corpus，或数据源的整体，是从语言大数据的森林里面挖掘提炼有统计价值的情报。

挖掘最早针对的是交易记录这样的结构数据，容易挖掘出那些隐含的关联（如，买尿片的人常常也买啤酒，原来是新为人父的人的惯常行为，这类情报挖掘出来可以帮助优化商品摆放和销售）。如今，自然语言也结构化为抽取的碎片情报在数据库了，当然也就可以做隐含关联的挖掘来提升情报的价值，这也是我们京东 NLP 在电商领域着力要做的任务之一。

第四张架构图是 NLP 应用（Apps）层。在这一层，解析、抽取、挖掘出来的种种情报可以

支持不同 NLP 产品和服务。从问答系统到知识图谱（包括对于电商领域具有核心价值的产品图谱和用户画像及其之间的关联），从自动民调到客户情报，从智能助理到自动文摘等，这些都是 NLP 可以发力的地方。

NLP 团队

具体到目前的工作，我领导的京东硅谷 NLP 团队还是有很多与众不同的特色。

我们的主核是 把语言结构化然后支持应用，而不是主流 NLP 的绕过显性结构解析来做的端到端深度学习。为此我们结合了人工智能领域的两大流派，以创新的多层符号逻辑（包括利用本体知识和常识的 ontology）和语言学模块作为精准分析的基础，以统计学习作为 backoff，使

得两种方法互补，取长补短。

这样设计的好处不仅照顾了 NLP 的精准 (precision) 和召回 (recall) 两方面的需求，而且使得系统调控变得比较透明，容易 debug。相较端对端系统，结构化的最大优势是不依赖海量的带标数据，因为深度解析的 NLP 应用是在知识和结构理解的基础上进行的知识工程项目，而不是从表层的标注好的冗余案例中学出来的模型。

这对于京东的一些场景有特别的意义。京东不乏业务场景和各种 NLP 应用的领域需求，这些场景和领域往往没有现成的带标数据，为这些多方面的场景组织人力进行深度学习所需要的海量标注，常常不是一件现实的事情。我们的目的就是 打造具有核武器威力的 NLP 深度解析平台，克服这个带标数据的知识瓶颈，为 NLP 多方面的电商场景的应用落地开辟道路，尤其是京东智慧供应链对市场需求客户情报的洞察挖掘以及产品舆情的意图挖掘，构建对于电商智能化至关重要的商品图谱 (product knowledge graph) 和用户画像 (user profile) 的知识引擎。

这条道路初期比较艰辛，需要深厚的计算语言学的功力和大数据驱动的研发，但 NLP 深度核心引擎打造出来以后就是另一番天地，这是一个赋能的核心技术 (enabling technology)。你想想，千变万化的语言表达一旦有规模的结构化以后，那会是一种什么情形：各种 NLP 任务在结构的显微镜下变得有迹可循，模式清晰并逻辑化，无论是情报挖掘还是其他应用都可以做到以不变应万变，以有限的句型把握无穷的语言现象。这就是我说的 “深度解析是 NLP 应用的核武器” 的本意。我在演讲中会通过多方面的 NLP 应用场景来展示和论证这一主题。

深度解析

所谓深度解析 (deep parsing)，就是把非结构的文本语句 (unstructured text) 自动解析成为深层的结构化数据（学界也称为 logical form），就是在自然语言与数据库之间建立自然语言理解 (natural language understanding) 的桥梁。

主流的文本情报挖掘 (text mining) 是绕过结构和理解的，依靠的是端对端的自动抽取挖掘的机器学习和深度神经。在具有海量带标大数据的情况下，由于数据的丰富和冗余，端对端的有监督学习系统也可以达成很好的挖掘效果。然而，一旦领域挖掘任务变了，必须重新标注和重新学习，这里面临一个巨大的知识瓶颈，就是说，领域带标数据往往严重不足，为每一个领域的每一个挖掘任务组织人力标注一个大数据训练集来克服稀疏数据的困难往往是不现实的。这是当前 AI 和 NLP 主流面临的一个巨大挑战。

我们的对策就是融合深度解析 (deep parsing) 和深度学习 (deep learning)，结合人工智能的理性主义和经验主义方法论，各取所长，利用深度解析来保证数据挖掘的精准度 (precision)，利用深度学习来提高数据挖掘召回率 (recall)。

以社交媒体舆情挖掘为例，面对以短消息作为压倒多数的开放领域 (open domain) 社媒大数据，缺乏结构分析的主流舆情分类方法面临一个精准度瓶颈 (业界公认 65% 是难以逾越的天花板)，而利用深度解析的结构化舆情挖掘，我们可以达到 85% 以上的精准度，整整 20 个百分点的差距，这样的精度才真正能为舆情挖掘基础上的决策和智能化应用提供可靠的保障。

在智慧供应链的选品环节，从全网数据挖掘出可靠的用户需求及其对于产品的舆情反馈（点赞抱怨及其背后的原因）是非常重要的决策情报。这是我们目前的深度分析平台落地的主要目标之一。

NLP 作用

语言的奥秘在于，语句的呈现是线性的，我们人类说话或写文章，都是一个词接着一个词表达出一个一个的语句；但语言学的研究揭示，语句背后是有语法结构的。我们之所以能够理解语句的意思，是因为我们的大脑语言处理中枢能够把线性语句下意识解构（decode）成二维的结构：语法学家常常用上下颠倒的树形图来表达解构的结果，这个过程就是深度解析（deep parsing）。

深度解析被公认为是自然语言处理和理解的核心任务，但长期以来大多是科学家实验室的玩具系统（toy systems），其速度（speed）、精准度（precision）、覆盖面（recall）和鲁棒性（robustness）都不足以在真实语料的大数据应用场景。而这一切已经不再是梦想，高精准度和高召回率（作为指标，精准召回的综合指标 F-score 要达到 90% 以上，接近语言学专家的分析水平）、符合线速要求的鲁棒的深度自动解析已经得到验证和实现，这是大数据时代的 NLP 技术福音。

再强调一遍，语言为什么要结构化？盖因语言是无限的，但结构是有限的，只有结构化，有限的模式才能捕捉变化多端的语言。话句话说，结构化是语言理解应用之本，现代的 deep parser 就是结构化的核武器。

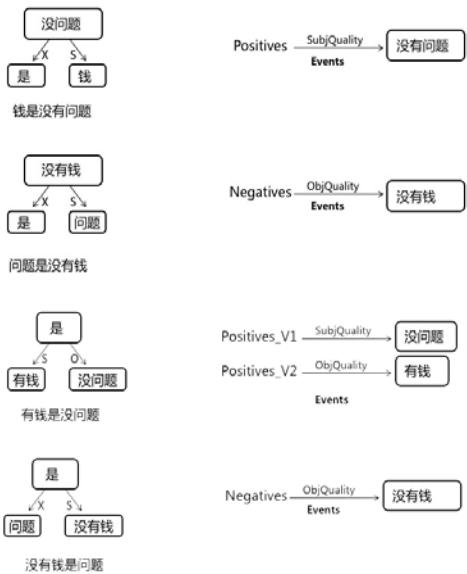
总体而言，我们面对的是不断变化的 NLP 任务，变化中的不同业务场景和情报需求。靠谱的深度解析结果反映在语法语义的结构图上，它离领域的信息抽取和情报挖掘只有一步之遥，离情感分析或舆情挖掘可以说是两步之遥（深度舆情的确需要一些苦功夫，舆情语言的复杂多变和模糊不确定，使得舆情挖掘比起传统的以事实作为抽取对象的情报挖掘要困难）。

结构化信息抽取的作用是巨大的，有多少产品的想法，就可以定义多少种不同的抽取任务。但万变不离其宗，只要抽取面对的是自然语言，它就必然总体上服从这个语言的文法，因此深度解析成为核心引擎的系统就顺风顺水。知识瓶颈因此被很大程度地克服了，不再需要那么多的带标数据。有了 parser，只要一些示意性的带标数据就够了，开发者可以根据示意举一反三。大多数信息抽取的开发任务，在有靠谱 parser 的支持下，可以在 2-4 周内开发完成，满足应用的基本需要，后面就是维护和根据反馈的 bugs 报告，做增量修补而已。

所以说 deep parser 打开了通向应用的大门和无限可能性。对于京东的智慧供应链和电商平台的业务场景，我们的愿景是让深度解析落地开花在多个 NLP 方向上，包括提升自动客服的语言理解水平，构建商品的知识图谱和用户画像，这当然也包括客户需求和商品舆情的挖掘和应用。

中文与 NLP

比起英语和其他欧洲语言，中文的语法具有相当程度的灵活性，成为自动分析的难题。与业界同仁的交流中，我们把中文叫做“裸奔”的语言，就是说中文的表达缺乏显性的形式标志，因为中



文没有形态（词尾），常常省略功能词（譬如介词），而且词序其实也相当灵活。

这些语言学的特点，加上不同地区的人的不同语言习惯，再加上社交媒体中反映出来的大量别字以及语言不规范，使得很多人对中文自动分析有很深的怀疑。这是好事儿，正因为它看上去如此复杂多变，才更需要对语言学的深刻认识和对语言工程的架构和方法有独特的创新。中文自动处理和理解提高了技术竞争的门槛。

这对我们而言，就意味着要突破乔姆斯基理论为基础的传统流行的上下文自由文法（CFG）的单层 chart-parsing，代之以自底而上的多层次语言处理系统，从而穿越乔姆斯基层级体系（Chomsky hierarchy）的围墙，在机制上有所创新（formalism innovation）。这一切需要深厚的计算语言学的素养和积累，才有希望。这方面的理论和实践，可参见白硕老师的《白硕 - 穿越乔家大院寻找“毛毛虫”》以及我的《乔姆斯基批判》和《语言创造简史》。

举例就举老友转来挑战我的所谓“2016 年最佳语文组词能力”，他给我发微信说：“钱是没有问题”，就这六个字的组词成句，可以变成不同意思的句子！哈哈，伟大的语文能力！parsing 请：

钱是没有问题；问题是没钱；有钱是没问题；没钱是问题；问题是钱没有；钱没有是问题；钱有没有问题；是有钱没问题；是没钱有问题；是钱没有问题；有问题是没有钱；没问题是有钱；没钱是有问题。

老友说的是中文词的不同的组合产生不同的意义，给人感觉是如此微妙，机器如何识别？其实仔细研究可以发现，这样的语言事实（现象）并非想象的那样玄妙不可捉摸。先看一下机器全自动分析出的样子吧！

这里面的 know-how 的细节就不赘述了，总之结果虽然仍有少数不尽如意尚有改进空间的结构分析，但几乎每个 parse 都可以站得住，说得出来道理。作为设计者，我自己都有点吓倒了。

（摘自《一日一 parsing：“钱是没有问题”》，更多参见【立委科普：自动分析《伟大的中文》】，关于中文自动分析的很多有意思的案例和深入的讨论，也可参看白硕老师与我就中文 NLP 的华山论剑似的《李白对话录系列》）。

NLP 场景与未来

要问是技术驱动业务，还是业务来驱动技术？我们坚持业务驱动，毕竟 NLP 是一个应用学科，再深的分析研究最终还是要落实到业务场景，解决业务痛点才能显示其价值。

在这个基础上，我们尝试从业务场景的点开始，逐渐借助深度解析的 NLP 平台技术，扩展

到多项业务场景，发挥结构化技术的跨领域核武器的作用，帮助克服领域数据的不足，以期快速领域化。

我这个小组的成员在业界有多年的 NLP 和机器学习专业经验，但成立迄今才刚半年，一切还是刚刚开始。随着深度解析平台的建立和打磨，在京东电商的各个场景只要找准 NLP 的切入点和大数据的场景，就会有实际的效益，对这一点我们充满信心。

大数据时代的信息过载，使得人类个体消化和利用信息的能力受到严重限制，只有借助电脑的自动分析和挖掘，情报才能从噪音的海洋中被有效挖掘和利用。

那么 10 年后 NLP 会怎样呢？

回顾 NLP 的历史，语言技术真正落地开花结果还局限于少数几个方向，如机器翻译、语音系统和文本分类。今后的 10 年才真正是 NLP 的黄金时代，全面开花结果可以期待，尤其在情

报挖掘、知识图谱、人机交互和智能搜索方面。NLP 是人工智能从感知全面进入认知的桥梁。

我这么说不是廉价迎合多少已经带有泡沫的 AI 现状，而是作为第一线 AI 从业人员的真实的有感而发。为什么这么说？我的根据主要有四点：

- 深度解析技术业已成熟，接近或达到人的水平；
- 深度解析与深度学习的融合和合力可以取长补短；
- 大数据可以弥补 NLP 技术的不够完善之处；
- 信息过载的大数据时代，不缺乏 NLP 的用武之地。

我的观点是，NLP 面对大数据时代，想不乐观都不成。深度解析是 NLP 应用的核武器。

李维博士，现任京东硅谷研究院主任研究员，领导 Y 事业部硅谷 NLP 团队，研发自然语言深度分析平台及其 NLP 应用，目前聚焦于大数据情报和舆情挖掘，以及智慧供应链应用。NLP 深度分析平台具有广阔的应用前景，方向还包括客户情报、信息抽取、知识图谱、问答系统、智能搜索、智能客服、自动文摘等。

直击阿里全球运行指挥中心 双11的隐形战场



全球运行指挥中心

全球运行指挥中心 (Global Operations Center, 下文简称 GOC) 是阿里巴巴集团基础架构事业群下属的事业部，是阿里经济体业务稳定运行的核心团队，负责生产系统全局性应急决策与指挥。GOC 团队通过为电商、金融、云计算等各项业务提供及时准确的告警、生产环境故障的全生命周期管理、重大故障时的快速切换以及线上问题的升级支持，在缩短系统灾难时长和提升消费者体验等方面做出了贡献。

之所以称之为 全球运行指挥中心，是因为阿里经济体的业务中有很多国际化的业务（如 AliExpress 速卖通、国际支付、Lazada 等），

业务的用户和受众遍布全球。

一直以来，GOC 从预防、快速恢复到复盘检验等环节全面推进业务连续性建设。

首先，GOC 持续推动系统的容灾和快速恢复的建设，确保各个机房都有同城或者异地容灾的方案，并通过日常演练来检验集群的容灾能力。

同时，经由与各个业务部门的密切合作，GOC 把各核心系统在极端情况下快速逃生的开关接入统一的平台，真正实现了快速恢复。

其次，在业务流量发生波动时，通过自建的嵌入机器学习模型的智能基线系统，GOC 能第一时间发现故障并判断处理方式。

如果该故障需要人工介入，则会迅速通知相关开发人员上线处理，并实时跟踪进展。在故障

处理完毕后，GOC 会与业务团队一起进行深度复盘，制定明确的改进措施，并通过模拟故障来检验系统是否已经具备了对类似的问题的免疫能力。

经过长期的技术积累，今天的 GOC 已经拥有了从覆盖从故障管理、应急响应、容灾演练、变更管理等平台化产品，打造出了一整套业界领先的业务连续性建设解决方案。

挑战与背后团队

除了由于时差等造成因素、全球化业务之间形态的差异、业务指标变化趋势的差异、以及各数据中心之间物理距离的加大，对我们准确地发现、判断、通告故障以及收集相关信息等工作带来了巨大的挑战。面对这种挑战，我们主要是通过智能化的手段，利用算法和数据提升故障发现和故障辅助定位等能力，提升服务的效率和准确率，以平台化的产品来支持我们提供的服务。

另一方面，我们也在美国的加州设立了子团队，实时响应和支持全球业务的监控发现和故障应急支持工作。

阿里集团一直倡导 DevOps 的理念，传统意义上的应用运维团队已经融入了各事业部的研发团队中，同时有专门的运维中台团队来负责研发通用的监控、部署等运维平台。

目前，阿里集团的各个运维团队也已经在向智能化的无人值守运维方向演进。GOC 团队作为横向团队，整体负责所有团队的业务指标异常发现，故障级别确定和故障通告。对于符合快速切换条件的故障，由 GOC 团队通过平台化产品进行服务快速切换来恢复服务。

对于其它类型的故障，由 GOC 通过平台化

产品来进行故障恢复过程中的信息流转和应急指挥。故障定位和恢复之后，由 GOC 团队通过工具和平台协同故障相关的同学进行故障的复盘。

故障在 GOC

GOC 负责管理全集团所有生产环境的故障。故障一词，在阿里集团内部有着明确的定义和分级，根据不同事业部的业务范围和特征，我们有上千条明确的故障等级定义。这些等级定义精确地界定了故障的现象、波及范围和相应的故障级别。

例如，淘宝交易量下跌 $\%X$ 且 Y 分钟未恢复，会被定级为 Pn 故障 ($n=1^4$)。P1 故障最为严重，而 P4 故障相对较为轻微。我们会根据故障的等级来决定我们的处理顺序和优先级。这些故障等级定义可以和我们的业务监控项（通过实时数据采集、流式计算得出的代表业务指标的时间序列数据）建立关联，方便后续的故障自动化和智能化的处理过程。

故障的处理包括几个环节：故障发现，故障定级，故障快恢，故障定位，故障复盘。

- 在故障发现环节，接近一半的故障可以被我们的智能化异常检查算法自动化地发现（其它不能自动化发现的故障主要是业务故障的现象不能直接反应在某个时间序列数据上，比如淘宝首页乱码之类）。
- 在故障定级环节，所有可以表征为时间序列的故障等级定义对应的故障都可以自动化地进行定级。
- 在故障快恢环节，目前符合快速恢切换条件的故障场景主要集中在阿里集团核心的交易链路相关的故障上。

- 在故障定位环节，我们能够自动化地推荐可能与故障原因相关的系统 / 应用层面的各类事件（如可疑变更、系统 / 应用的指标突变、网络抖动等）以辅助相关人员进行故障定位。
- 故障复盘环节仍然是一个需要相关负责同学人工参与的环节，我们可以通过平台和工具自动化地收集故障发生过程中的相关信息，方便相关人员进行故障复盘。

正确率从40%提升到 80%的背后

首先需要明确的是，我们对于业务指标监控的正确率的评测是在开放数据集上持续进行的，其判别标准主要基于上面提到的故障等级定义（但不是完全和故障等级定义相同，因为我们往往要在故障还没有严重到构成故障时就要检测出异常）。这和我们去判断一个集群的 CPU、IO 或一个微服务的 QPS 这样的指标是否异常，其尺度和标准是不同的。

由于业务指标时间序列数据本身具有周期性、趋势性规律，同时会受到内部系统层面和外部用户层面的多重影响进而存在很多噪声，因此基于静态阈值（或分段阈值）或同、环比的监控策略的正确率仅有 40% 左右。经过项目团队的努力，我们通过引用异常检测算法，把监控的正确率从 40% 提升到了 80% 左右。这相关于我们利用智能化的力量，每周为团队节省了 29 小时（因误报警而造成的）操作时间。

在这个过程中，我们主要在算法、工程框架和运营三个层面进行了优化和提升。

一、在算法层面，我们的主要优化集中在三个方面：

- 通过数据预处理以降低输入数据中的噪声；
- 通过时序分解来进行正常值的预测；
- 通过复合的异常判别方法和自适应的参数学习来提升报警的有效性。

二、在工程框架层面，我们支持对不同类型的曲线（甚至同一个曲线的不同时间片段）可以存储和更新不同的算法参数。

三、在运营层面，我们通过对于标注数据的分析，发现制约算法效果的若干因素，并制定针对性的策略来解决。

关于未来的 20% 如何攻克的问题：一方面我们会持续针对算法仍然存在的长尾的误报情况研发针对性的算法策略，另一方面我们也在通过与高校进行合作，探索利用深度学习等算法来解决业务指标异常检测问题的方案。目前，相关工作正在进展之中，让我们拭目以待。

随着算法优化工作的推进，特别是利用人的反馈（标准）来进行自动化反馈算法的研发过程中，我们也发现人对于什么是异常的标准也并非完全清晰。因此，我们也会加强对人工标注质量的定量统计和评估，厘清判断标准中模糊的部分，让算法专注于解决应该由算法解决的问题。

算法如何对比、选型，及应用？

在算法应用的业务场景确定的情况下，决定同类算法选型中最重要的因素或许是算法输入数据的特性和质量。

在异常序列的预测（拟合）这个环节，我们调研了时序序列分析算法中的 ARIMA, Holt-Winters 以及 STL 方法。最后，由于我们的输入数据本身间距周期性和趋势性，根据实验的结果我们选择了 Holt-Winters 算法和 STL 算法。

又由于我们输入数据中有较多的噪声，而 STL 算法的鲁棒性更好，因此我们根据实验结果又选择了以 STL 算法为主作为我们的时序分解算法。

当然，对于不同的业务数据，这个选择也不相同。我们的业务数据有上千条，因此我们也设计了一套基于分类的算法，来自动选择具体的分解算法及算法分解之后具体的参数。

我们在算法中采用的自适应的异常判定的技术主要是指我们能够通过对于数据本身特征的分析，以及人对算法结果的标注这两个因素共同动态地决定异常检测的参数。这其中的难点一方面在算法需要根据不同数据的不同特征，以及相同数据不同时间段的特征来大致确定初始的算法参数。一方面在于需要根据人对算法输出的评价来动态调整并最终稳定收敛出一组确定的反馈参数，并将初始参数和反馈参数组合起来，共同决定算法最后的输出。

最终算法策略实现层面的难度并不大，更多工作的是我们在理论上分析出这两类因素的作用，然后在工程框架上予以支持。然后就是不断地利用开放地实际业务的曲线进行验证和迭代，对算法的策略进行微调，最终让算法的效果达到预期的水平。

大促

从故障本身对应的时间序列指标的角度看，平时工作时、平时节假日、法定节假日、大促日（双 11，双 12 等）这四类日期的时间序列的趋势特征都各不相同。因此，在实践上，我们采用四组不同的策略来分别对这四类日期的数据进行拟合和异常判定。

除此之外，GOC 团队也会利用平台化的工具

产品，支持集团业务团队为备站双 11 而进行容量验证、故障演练等相关工作。

实际上，智能运维产品的功能和价值更多地体现在日常的运维工作中，而双 11 则是检验一切线上业务和运维产品的一次大考。很多基于智能算法的容量预估和弹性调度策略会在双 11 发挥巨大的作用。

对于很多基于时序分析或机器学习的智能监控系统来说，往往需要将从历史数据中学习到趋势规律用于对当前数据进行监控和调度。而双 11 的特殊之处就在于，几乎每次双 11 都不同于往年。双 11 的数据一定不是往年历史的简单重复，而是对历史的突破和创新。因此，如何在双 11 期间保持智能监控策略本身的有效性仍然是一个巨大的挑战。

今年双 11，阿里 GOC 团队和阿里监控中台团队合作，基于异常检测算法和运维数据仓库，共同设计和研发了双 11 事件大屏幕功能。我们希望能够在双 11 及以后的工作中，当研发及运维同学发现自己的应用存在异常时，我们的产品能够自动化地找到最显著的异常趋势（不依赖于人为配置的监控策略），并且推荐最可能导致这种异常的可疑事件，帮助大家快速定位业务运行时遇到的问题。

另一层面，双 11 是全体阿里技术体系的一场战役，GOC 团队也专门为类似双 11 这样的重大活动保障而设计和研发了专门的产品，保障和支持双 11 期间阿里内部各团队之间信息和问题的流转。

身在百度与阿里

在加入阿里巴巴之前，我曾作为百度智能运维团队的架构师及核心项目负责人，其实从技术本身的层面来看，百度的运维和阿里的运维对技术的要求没有本质的区别。而两家公司文化、技术架构层面的差异也决定了运维技术团队在发展理念和过程上的差异。

从近年来运维技术发展来看，运维技术在微观上向基础架构（基础软件、中间件）领域延伸，宏观上向智能化方向发展，则是包括阿里、百度在内的互联网运维团队共同的技术发展趋势。

在智能运维的方向，百度和阿里的运维团队都在进行不断的实践和探索：

- 百度智能运维团队在异常检测、容量预测、流量调度等智能运维领域有非常多优秀的实践；

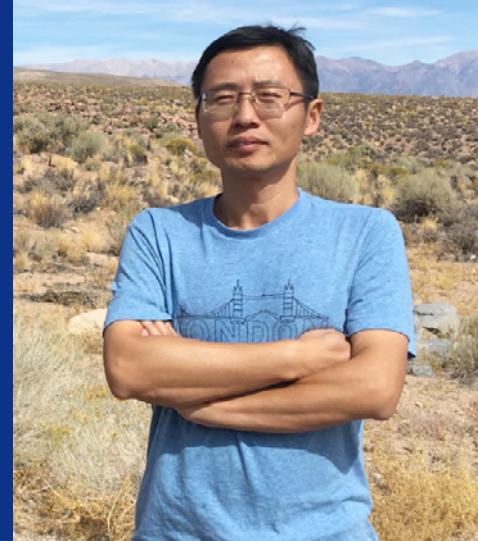
- 阿里集团各运维团队的同学，也在不同细分领域由自动化向智能化演进。

阿里巴巴集团内的各个业务的形态差异较大，业务之间的关联也相对较为复杂，这也给基于数据、算法的智能运维的效果提出了更高的要求，同时也为智能化运维产品提供了广阔的发展空间。阿里集团在智能运维中的业务指标异常检测、容量管理、异常关联分析、智能调度、硬件优化等层面也取得了很多优秀的成果。

我们相信，追求自动化、智能化是互联网运维从业人员的普适价值观和发展方向。我们也希望能够通过 InfoQ 组织的各种线下、线上场合和活动，更多地加强互联网企业间在智能运维领域的技术交流，互相学习，共同推进智能运维技术的发展，为业务带来更多可见的价值。

王肇刚，阿里巴巴高级技术专家。阿里巴巴集团基础设施事业群—全球运行指挥中心（GOC）高级技术专家。负责阿里巴巴集团业务指标监控、业务故障管理工作。在时间序列异常检测、业务故障定位及影响面分析、运维数据仓库和其它相关的智能运维相关领域有丰富的技术经验积累和成果产出。

双11进行时！京东虚拟商品系统的高可用架构设计



京东虚拟商品系统

虚拟商品系统和实物体系从架构体系上来说比较类似，都是服务化的、分布式的，都要满足高可用、高并发和高扩展性的架构设计目标。

从整个交易流程上来说，虚拟商品不涉及到仓储物流配送，但虚拟业务有很多自己的特点也就决定了在架构上需要一些特别的设计。比如交易过程需要与第三方供应商进行多次交互，这对系统的高可用性提出更多的挑战。

以机票系统为例，查询机票需要调用多个供应商的接口，在接口不稳定或外部网络延迟过长的时候，如何通过缓存、降级保证系统的可用性和用户体验。

随着公司不断扩展虚拟业务种类，并把核心的业务做深做强，对系统的架构设计也提出了更高的要求。虚拟商品系统架构大概可以分为三个阶段：

- 初始百花齐放
- 中期业务系统平台化 + 基础的京米平台
- 目前的组件化、业务赋能阶段。

虚拟业务刚成立的时候为了快速上线新的业务，业务系统各自开发。随着业务种类和业务量的迅猛增加，为了更好的支持新的业务，提高系统的稳定性，我们做了两方面的升级：把业务系统平台化，相同的业务在一个平台里完成，同时把通用的功能比如商品、下单、虚拟风控、退款等抽取出来构成一个基础的京米平台，这样虚拟

业务只需要关心本领域高度相关的业务逻辑，新的业务可以基于京米平台快速搭建。

今年以来，除了在平台化上继续深化，我们进行系统组件化设计，通过组件的组合和配置完成新系统新功能的构建，同时可以更迅速的将我们的业务能力开放出去。

大促

大促时期，虚拟商品系统每日有海量的订单。我们会关注像数据库、外部接口、网络等多个地方。

以数据库为例，应用无状态可以快速伸缩，但数据库目前还比较难做到。在架构设计时，对访问量高的调用尽量减少对数据库的依赖，同时排查长事务、检查索引、备份并清理过期数据等等，同时对数据库进行监控，并和数据库运维工程师紧密合作，如果出现数据库瓶颈（比如发现慢 SQL），快速定位问题，并进行相应的处理。

从架构设计上有很多降级的方案，可以自动降级，比如我们的充值系统，如果某个供应商的接口不可用或者不稳定，可以根据配置自动切换到其他供应商。

我们对故障处理的时间要求是在分钟级别，然后会迅速升级。大部分情况下都能比较快的解决，对用户的影响降到最小。为了保证能快速发现和解决故障，我们会提前架构进行梳理，我们的手段有：

- 技术改造系统薄弱的、有风险的地方；
- 加强系统全方位监控（分钟级和秒级）；
- 模拟真实用户压测发现问题；
- 准备各种场景应急方案并进行多次演练等等。

谈谈故障及应急手段

基本上可以自动处理的故障，都是根据业务的需求，系统可能出现的问题，或者是总结以前的经验教训，然后在架构设计中考虑进去并进行演练过。

比如在正常的业务系统设计中，用户支付后，支付系统会发送 MQ 消息，业务系统接收到支付成功确认消息会触发后续的处理。但支付系统可能因为高压力，或者网络的问题，MQ 消息延迟会比较长时间。

虚拟某些系统对此很敏感（如果不能快速获得支付确认并进行后续步骤，系统要取消订单），我们在设计的时候下单后如果一定时间没有支付确认，会按一定的时间间隔进行反查，同时与收到的 MQ 消息进行幂等处理。

关于大促，我们一直认为平时多流汗，战时才能少流血。我们把这些准备在日常的开发工作中就进行完成，比如直接与用户相关的系统，我们都进行压力测试后才上线，这样可以对系统的承载能力做到心中有底。

当然，大促中我们最关注的是系统的稳定性和出现故障能快速解决，11.11 前会进行多次的全链路压测，主要目的是确保重点业务流程中系统没有薄弱环节，上下游系统间人员的沟通通畅。

今年有全场景故障演练的工具，开发人员就可以非常方便的进行比如 CPU 类的系统故障、接口异常的服务故障等几十种场景的演练。

应急手段方面，我们会提前与业务部门沟通了解促销的情况，然后根据历史的数据，对大促的流量进行预测。最近几年的预测结果和实际的情况基本上差不多，比较准确。这样也确保我们的备战更加有的放矢，特别是像硬件资源的分配

上也更加具有针对性，应对起来更加有信心。

促销当日我们会有专门的值班室，各个业务系统核心开发人员都会进驻 24 小时值守，结合各种监控系统，如果出现故障，大家可以快速进行沟通交流，迅速定位问题，同时根据提前准备好的经过反复演练的应急预案来解决问题。

未来会怎么做？我们系统的架构设计会基于现有的情况，根据业务的战略目标，同时权衡成本和收益，确保架构在不用太大的修改也可以满足业务未来几年快速增长的需要。就虚拟系统来说，未来会更多的侧重于组件化，并在合适的场景引入 AI 人工智能，帮助更好的提升用户体验，同时进行业务赋能。

如何看待重构？

软件的架构应该根据业务的具体场景进行，在遵循通用的一些设计准则基础上，进行权衡。

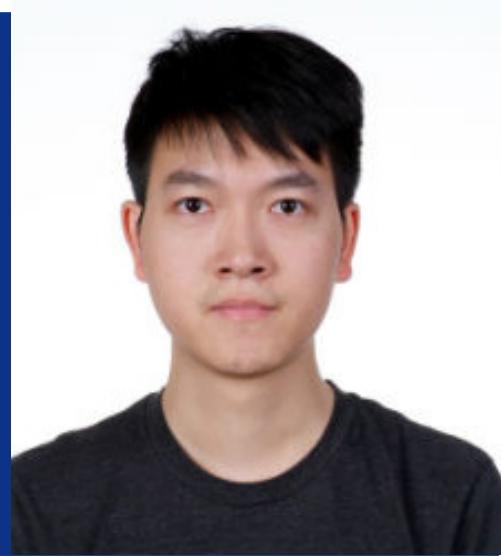
比如对于初始的新业务项目，为了快速上线抢占市场，开始就进行细粒度划分，采用严格的微服务架构往往不是最好的方式，而在项目中使用模块化可能会更好。因为这时对业务领域的理解并不太深刻，服务的划分不合理，很可能会使服务之间紧耦合，变得复杂，数据一致性保证困难，降低开发效率，增加维护成本。

所以对于这类型项目，当系统上线后，随着业务的增长和对业务领域更深入的了解，根据不同的业务和业务需求，进行合适的重构，不要为了重构而重构。每次重构都需要有具体的可衡量的目标 / 需求解决痛点和问题，比如增加一个业务规则需要修改的地方太多，又比如系统需要支持每秒多少次的访问而目前架构满足不了。

周光，京东商城研发部架构师。14 年软件开发相关工作经验，8 年以上软件架构设计经验。

擅长 SOA，微服务架构，工作流系统。对互联网高并发、高性能、高可用性分布式系统有丰富的架构设计和开发经验。

世界正在走向实时化，谈谈 Twitter 对流处理的理解与思考



当今时代，数据不再昂贵，但从海量数据中获取价值变得昂贵，而要及时获取价值则更加昂贵，这正是大数据实时计算越来越流行的原因。

我们可以将每一次用户点击，每一个数据库更改，每一条日志的生成，都转化成实时的结构化数据流，更早的存储和分析它们，并从中获得价值。同时，越来越多的企业应用也开始从批处理数据平台向实时的流数据平台转移。关于企业如何使用数据的一次技术革命拉开了序幕。

Twitter 每天要接收和处理用户发送的数十亿条推文。实时分析这些推文是一个巨大的挑战。从 Twitter 实时计算框架的演进可以看出：提高计算的时效性，更快的从数据中挖掘出信息和知识就意味着能够获取更大的价值。

对于实时数据技术栈（包括流计算引擎、数据存储引擎、编程语言和工具）的最前沿现状是什么样子？在这其中遇到了哪些技术挑战？以及这些前沿技术怎么影响流计算的架构和应用呢？带着这些疑问，InfoQ 采访了 Twitter 数据平台组吴惠君博士。

流计算成为大数据实时计算的事代名词

实时计算一般都是针对海量数据进行的，一般要求为秒级。实时计算主要分为两块：

- 数据的实时入库
- 数据的实时计算

数据源是实时的不间断的，要求用户的响应

时间也是实时的（比如对于大型网站的流式数据：网站的访问 PV/UV、用户访问了什么内容、搜索了什么内容等，实时的数据计算和分析可以动态实时地刷新用户访问数据，展示网站实时流量的变化情况，分析每天各小时的流量和用户分布情况）。

当谈及大数据实时计算方面当前的挑战时，吴惠君表示这个挑战就如同其名字一样：大数据的挑战，实时计算的挑战。其中，大数据的挑战包括很多方面：

- 比如‘数据的获得’上，现在移动端的数据分布广，按照一般的思路是收集手机数据到云上，那么这个收集的系统就是第一个要解决的挑战，一般 Kafka 比较适用，或者类似的 Pulsar 也有很多人尝试。
- 比如‘数据的存储’上，由于这些收集的数据要被实时计算使用，所以这个存储系统要响应快速，并能很好的跟实时计算的系统结合，一般 Druid 可以适用在这个场景，另外它还能跟批处理的历史数据存储结合。除此之外，还有比如‘数据的备份’等等。

在大数据挑战基础上的实时计算的挑战，是个非常复杂的问题。传统的硬实时或者软实时系统难以应用在大数据的情况下，然后在现有大数据平台的基础上构建新的实时系统才是普遍采用的方法，其中流计算就处在这样一个位置。

流计算有几个适应的点，它可以方便的建立在大数据的平台或者云上，并且能够通过一些扩容方法来满足各种大小数据量，凭借它的低延迟的特性，它可以满足一定的实时要求它和其他大数据处理的项目和社区共享先进的理念和经验，这些让流计算当仁不让的成为大数据实时计算的事实代名词。

可见，数据的价值随着时间的流逝而降低，所以事件出现后必须尽快地对它们进行处理，而不是积累。对于这点和传统的 MapReduce 有差异，即传统的分布式计算往往是首先拿到一个大的积累后的数据，再进行数据拆分和聚合。而流处理的重点是通过事件机制，类似流管道一样，接收都消息马上就进行处理。

流式大数据的实时处理

大数据行业核心技术面临的挑战仍然存在，并将在可预见的未来持续下去。随着数据呈指数级增长，企业组织和服务于其的技术公司将继续处在一场持续的战斗中，使其变得易于管理。

大数据通常被分为两类：一类是批式大数据，另一类是流式大数据。

如果把数据当成水库的话，水库里面存在的水就是批式大数据，进来的水是流式大数据。

其中，流式数据的实时分析，一定是有规则、模型的东西。复杂的分析计算，加上实时这两个结合起来，如果能做的好，一定能够加速大数据在各个行业的应用。

当前，越来越多的企业选择流处理。流处理打破了传统的数据分析和处理的模式，即数据最终积累和落地后再针对海量数据进行拆分处理，然后进行分析统计，传统的模式很难真正达到实时性和速度要求。

而实时流处理模型的重点正是既有类似 Hadoop 中 MapReduce 和 PIG 一样的数据处理和调度引擎，又解决了直接通过输入适配接到流的入口，流实时达到实时处理，实时进行分组汇聚等增量操作。

由于流数据实时到达，实时处理，有具体的

数据处理的流处理引擎，因此具备低延迟，高可靠性和容错能力。

Twitter 如何实现对实时系统的异常检测

据吴惠君介绍：Heron 项目就是 Twitter 提供的流处理或者叫做大数据实时计算的典型。

一个稳定可靠的系统离不开监控，我们不仅监控服务是否存活，还要监控系统的运行状况。运行状况主要是对这些组件的核心 metrics 采集、抓取、分析和报警。

吴惠君描述：

在异常检测方面，Heron 采用的 Dhalion 框架，在 Dhalion 框架基础上开发了 Health manager 模块来检测和处理系统异常。其中，Dhalion 框架是由 Microsoft 和 Twitter 联合发明专用于实时流处理系统的异常检测和响应框架，它主要应对三个在流处理系统中的常见场景：

- self tuning
- self stabilizing
- self healing

Dhalion 由一个 policy 调度器和若干个 detector 和 resolver 来合作完成异常检测和响应。

policy 管理 detector 和 resolver：

- detector 检测系统各种指标的异常；
- resolver 根据 detector 的结果进行对应的处理。

另外，还有些辅助的模块来配合这个主过程，比如 metrics provider 喂给 detector 检测数据；action 等完成 resolver 和 detector 之间的协同。

在 Dhalion 基础上的 Health manager 实现了几个 Heron 常用的 detector 和 resolver。比如检测 slow instance 的 detector。有些时候，某些容器云调度的容器由于各种原因会比较慢，那么根据这个容器的各种指标，比如队列长度，响应时间，反压 backpressure 标志等，判断比较然后得出这个容器结点相比其他容器是不是异常。

其他一些 detector 还包括检测是不是整体上资源不够，是不是可以压缩资源池等等。在 resolver 方面，从最简单的重启容器，到扩容 / 缩容，特殊操作等等都可以用 resolver 的形式实现应用。

Heron 增加了新功能

自去年 Twitter 开源了大数据实时分析系统 Heron 以来，一年多的时间，Heron 社区开发了许多新的功能。据吴惠君介绍：特别是今年 Heron 增加了 elastic runtime scaling, effectively once, functional API, multi-language topology, self-regulating 等。

elastic runtime scaling

根据 storm 的数据模型，topology 的并行度是 topology 的作者在编程 topology 的时候指定的。很多情况下，topology 需要应付的数据流量在不停的变化。

topology 的编程者很难预估适合的资源配置，所以动态的调整 topology 的资源配置就是运行时的必要功能需求。直观地改变 topology 中结点的并行度就能快速的改变 topology 的资源使用量来应付数据流量的变换。Heron 通过 update 命令来实现这种动态调整。

effectively once

Heron 在原有 tuple 传输模式 at most once 和 at least once 以外，新加入了 effectively once。原有的 at most once 和 at least once 都有些不足之处，比如 at most once 会漏掉某些 tuple；而 at least once 会重复某些 tuple。所以 effectively once 的目标是使得计算结果精确可信。

Functional API

函数式编程是近年来的热点，Heron 适应时代潮流在原有 API 的基础上添加了函数式 API。Heron 函数式 API 让 topology 编程者更专注与 topology 的应用逻辑，而不必关心 topology/spout/bolt 的具体细节。Heron 的函数式 API 相比于原有底层 API 是一种更高层级上的 API，它背后的实现仍然是转化为底层 API 来构建 topology。

multi language topology

以往 topology 编程者通常使用兼容 Apache Storm 的 Java API 来编写 topology。现在 Heron 提供 Python 和 C++ 的 API，让熟悉 Python 和 C++ 的程序员也可以编写 topology。

Python 和 C++ 的 API 设计与 Java API 类似，它们包含底层 API 用来构造 DAG，将来也会提供函数式 API 让 topology 开发者更专注业务逻辑。

self-regulating

Heron 结合 Dahlion 框架开发了新的 health manager 模块。

- Dahlion 框架是一个读取 metrics 然后对 topology 进行相应修复的框架。
- health manager 由 2 个步骤组成，detector/diagnose 和 resolver。
detector/diagnose 读取 metrics 探测 topology 状态并发现异常，resolver 根据发现的异常解决让 topology 恢复正常。healthmgr 模块的引入，形成了完整的反馈闭环。

倚仗开源社区，大大节约流式处理成本

流式处理是一种成本和效费比都高的计算模式。企业要如何权衡成本与人员支出的呢？为此，吴惠君提出了自己的看法，借助云平台和开源社区的力量，可以大大节约成本。

据吴惠君介绍，企业中使用流计算的成本分两部分：机器和人员。

- 机器方面，很多企业都是基于容器云平台上搭建流计算系统，对于这种情况机器池大小，部署等都可控。那么，机器的成本基本就是实打实的成本，意思是要处理这么多这么快的数据就是要这么多机器。对于具体的业务，根据实际场景测试能知道真实要多少机器，然后再看业务设计再来决策是不是值得。
- 人员方面，以 Heron 为例，Heron 开源以后形成了一定规模的社区，很多新的功能和日常项目维护很多都交给社区来做。依托社区的力量，企业实际的人员开销并不大。

流数据处理引擎如何选？

我们可以对比一下现在流行的几种流处理项

目：Storm，Flink，Spark Streaming，Kafka Streams 和 Heron。

- 首先看 Storm。它是适用于需要快速响应中等流量的场景。Storm 和 Heron 在 API 上兼容，在功能上基本可以互换；Twitter 从 Storm 迁移到了 Heron，说明如果 Storm，Heron 二选一的话，没有啥意外 情况需要考虑的话，一般都是选 Heron。
- 然后看 Kafka Streams。它与 Kafka 绑定，如果现有系统是基于 Kafka 构建的，可以考虑使用 Kafka Streams，减少各种开销。
- 再看 Spark Streaming，一般认为 Spark Streaming 的流量是这些项目中最高的，但是它的响应延迟也是最高的。对于响应速度要求不高，但是对流通量要求高的系统，可以采用 Spark Streaming；如果把这种情况推广到极致就可以直接使用 Spark 系统。
- 最后，Flink 使用了流处理的内核，同时提供了流处理和批处理的接口。如果项目中需

要同时兼顾流处理和批处理的情况，Flink 比较适合。同时因为需要兼顾两边的取舍，在单个方面就不容易进行针对性的优化和处理。

可见，Spark Streaming，Kafka Streams，Flink 都有特定的应用场景，其他一般流处理情况下可以使用 Heron。

写在最后

世界正在走向实时化，越来越多的应用场景需要以很低的延迟来分析实时数据。随着实时数据的流行，流处理会是很重要处理方式。在许多快速扩展的实时用例中大多都应用了 Heron，其中包括异常和欺诈检测、物联网和万物互联应用、嵌入式系统、虚拟现实和增强现实、广告投放、金融、安全和社会媒体等。在实时流处理过程中，如何选择一款适合的流数据处理引擎？如何更快的采集数据并实时的处理数据？都是企业亟须突破的。

吴惠君博士，现任 Twitter 工程师，致力于实时流处理引擎 Heron 的研究和开发，Apache Heron committer。毕业于 Arizona State University，专攻大数据处理和移动云计算，曾在国际顶级期刊和会议发表多篇学术论文，著有《Mobile Cloud Computing: Foundations and Service Models》，并有多项专利。

流量小生DDOS攻击下，微博如何保证系统稳定不再挂？



写在前面

微博作为当今中文社交媒体的第一品牌，拥有超过 3.6 亿的月活用户，也是当前社会热点事件传播的最主要平台。热点事件的发生具有不可预测性和突发性，以 9 月 26 日上午“谢娜宣布怀孕”事件为例，从图 1 可以看出微博评论的流量在短短 10 分钟内迅速上涨，并在 20 分钟后达到了日常晚高峰的 2 倍之多。可见，为了应对突发事件带来的流量冲击，确保线上服务的稳定性，能够进行随时随地的快速弹性扩容十分重要。

传统的面对而传统的人工值守，手工扩容的运维手段，显然无法满足这一需求。为此，我们的目标是做到系统的自动扩容，在流量增长达到

系统的警戒水位线时自动扩容，以应对任意时刻可能爆发的流量增长，确保服务的高可用性。

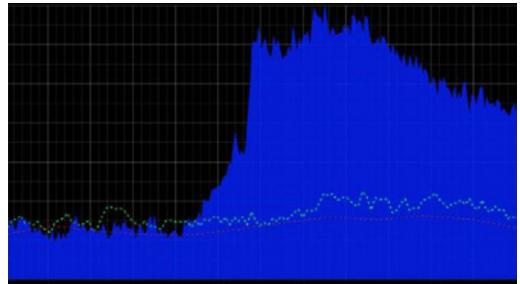


图 1. 9 月 26 日上午“谢娜宣布怀孕”事件微博评论流量

微博 Web 弹性调度演进

接下来，将为大家介绍微博 Web 系统如何

从人工值守的手工扩缩容一步步演进到无人值守的自动扩缩容。

第一代：人为触发扩缩容

- 人工根据监控系统的 QPS、AvgTime、load 等判断是否扩容；
- 根据经验值人为预估扩容机器数；
- 支持 PC、手机等多渠道触发。



图 2. 人工值守扩缩容流程

问题：需要人为介入确认扩容时机和扩容数量。

第二代：无人值守定时扩缩容

- 每天晚上 8 点定时扩容，12 点前定时缩容；
- Web 与依赖的 RPC 可以依赖扩容。



图 3. 无人值守定时扩缩容流程

问题：只能解决晚高峰问题，无法应对突发事件。

第三代：智能触发自动扩缩容



图 4. 智能触发自动扩缩容流程

- 自动压测评估线上服务池最大承载量
- 实时评估线上服务池冗余度

- 冗余度不足则触发扩容，充足则触发缩容
- 下图展示了在 # 薛之谦与前妻复合 # 事件中，智能触发自动扩缩容的实际效果。

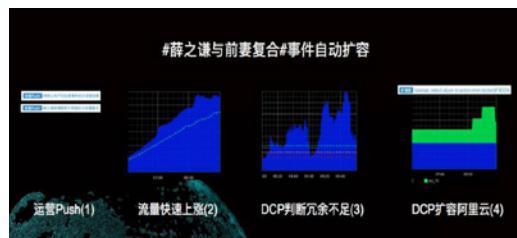


图 5. 薛之谦与前妻复合 # 事件自动扩容效果图

下面对智能弹性调度系统进行详细介绍，图 6 展示了这一系统包括的几个主要组成部分。



图 6. 智能弹性调度系统架构图

- 全数据日志分析 Profile，生成业务系统的各种指标并采集汇报给实时监控系统 Graphite。
- 实时监控功能系统 Graphite，实时汇聚并计算多维度业务指标数据，并提供 API 给在线容量评估以及智能弹性调度系统已作决策。
- 在线容量评估系统 Diviner，对在线服务池进行压测，以评估服务池的最大承载能力。
- 智能弹性调度系统 DCOS，根据系统实时的水位线情况，决策是否需要进行扩容以及扩容机器数。
- 混合云平台 DCP，向私有云和公有云申请机器，进行弹性扩容。

Profile- 全数据日志体系

在实际的扩容决策中，需要以一些关键指标如 QPS、AvgTime 或多种指标叠加作为判断依据，所以自动扩缩容系统首先要解决的问题就是关键指标的生成和采集。

指标的生产

一般情况下，指标的生产有两种方式：一种是在业务代码里以特定格式打印各业务关心的业务指标日志，如 API 的 QPS、AvgTime 等，但这种方式对业务代码的侵入性强，不建议采纳；一种是在框架的关键路径上埋点，统一打印 metric 日志，不侵入业务代码，对业务开发更加友好。我们就采用了这种方式，如在 motan 服务化框架上埋点，来记录 RPC 调用的 QPS、AvgTime、P99 等指标。

指标的采集

指标的采集主要涉及两个问题，一个是如何规范 metric 日志，便于在不同系统间传递，一个是如何传输的问题，有多种途径如 scirbe、kafka、udp 等。为了解决第一个问题，我们制定了规范的 profile 日志格式，各种 metric 信息均以标准的格式记录，如下图 7 所示。为了简化系统和传输效率，我们通过 udp 方式将各种 metric 信息传递给监控系统。

实时监控系统

有了关键业务指标的 Profile 日志，就可以对它们进行实时监控，其工作原理如图 8 所示。

从图 8 可以看出，实时监控系统的核心就是

```
 {  
   "type": "SERVICE",  
   "name": "SERVICE_A",  
   "slowThreshold": "200",  
   "totalCount": "1174",  
   "slowCount": "3",  
   "avgTime": "2.56",  
   "interval1": "1162",  
   "interval2": "5",  
   "interval3": "4",  
   "interval4": "1",  
   "interval5": "2",  
   "p75": "1.00",  
   "p95": "4.00",  
   "p98": "7.00",  
   "p99": "10.00",  
   "p999": "206.86",  
   "bizExcp": "0",  
   "otherExcp": "0",  
   "avgTps": "117",  
   "maxTps": "158",  
   "minTps": "80"  
 }
```

图 7. Profile 标准日志格式

Graphite。它主要包含两方面的功能：

将实时传输的 profile 日志进行聚合计算，产生各种维度的数据存储到时序数据库中。

还需要提供 API 接口，以提供 dashboard 展示，压测系统以及智能调度系统调用以用于扩



图 8. 实时监控系统架构图

缩容决策。下面将对 Graphite 进行详细介绍，其架构如图 9 所示。

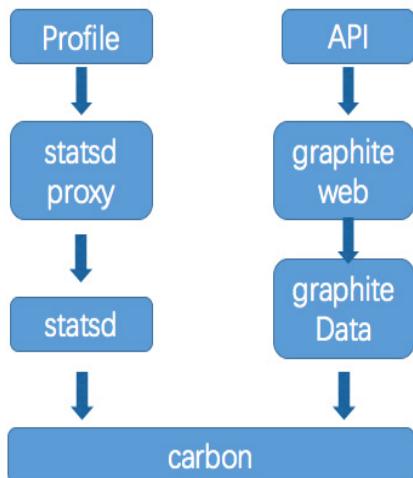


图 9. Graphite 架构图

为了减少延迟，我们对 graphite 系统中的关键模块进行了优化，主要包括两个方面：

- 用 go 重写了 statsd-proxy 和 statsd 模块
- carbon 中添加缓存，7 天的数据存储在缓存中，7 天以外的数据存到 SSD。

在线容量评估系统

有了关键业务指标的实时监控，就可以对系统进行压测，以评估系统的最大支撑能力。而如何对系统进行压测，以合理评估系统的承载能力主要取决于两个方面：

- 合理选择业务指标来衡量系统健康度；
- 精确定位性能拐点以确定系统临界值。

下面分别就上面两个问题进行阐述。

合理选择业务指标来衡量系统健康度

常见的可作为衡量系统健康度的指标有 avgTime、load、5xx、连接数等，对于单一业务模型的系统来说，选取其中一个指标即可。但对于复杂的业务系统，以微博 web 系统为例，既包含了 CPU 和带宽消耗较高的 feed 接口，又包含了低延迟和高并发的计数器接口，单一接口健康并不能代表整个系统健康。除此之外，单一指标正常也不能代表系统正常，比如我们经常遇到 feed 接口 avgTime 正常，但延迟大于 1s 的比率超过了 1%，这时候会直接造成 1% 用户刷新失败，影响了用户体验。

为此，我们建立了多接口多指标的健康度评估模型。

多接口，是指选取服务池中多个具有代表性的接口，并且还会考虑整体服务池中接口的情况，比如我们在微博 Web 系统中不仅选取了 feed 接口，还选取了计数器接口等。

多指标，是指不仅仅选取单一指标作为参照，比如我们在实际压测过程中，会考虑接口的平均耗时、5XX 以及慢速比（延迟超过 1s 的比率）。

精确定位性能拐点以确定系统临界值

常用的系统压测方案主要包括两种：

- 在线缩减机器数量以增加单机承载量，从而压测到单机承载能力的最大值。
- 模拟洪峰，对服务池进行全链路压测，以模拟出峰值流量下系统的承载能力。

目前，我们主要采用方案 1 进行压测。通

过减少在线机器数，以增加单机承载量来压测，直到服务池的健康度到达临界点时暂停压测，待系统恢复后，继续缩减在线机器数，否则则停止压测并记录服务池的临界值。

智能弹性调度系统

有了系统的最大承载能力，就可以根据实时监控系统中服务池当前的流量，来决定是否需要扩缩容以及扩容机器数。智能弹性调度的智能主要体现在两个方面：

快

智能弹性调度并不等到系统的承载量已经濒临临界值时才进行扩容，因为此时可能流量很快上涨超过水位线，在扩容完成之前就已经把系统压垮了。为此，我们给系统设置了三条线：致命线、警戒线、安全线，如图 10 所示。

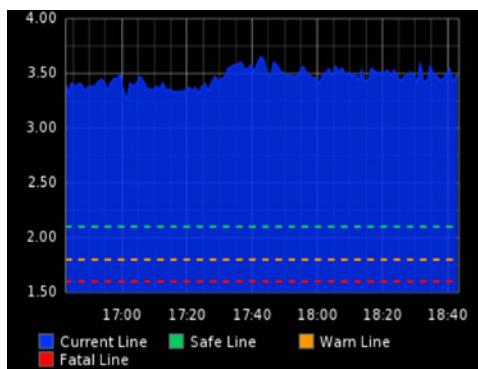


图 10. 智能弹性调度系统水位线

致命线。从字面意义上即可理解，当服务池的流量一旦触及致命线，就应当立即扩容。通常致命线的设定与业务的重要程度有关，一般核心系统的致命线设定要高一些，保证足够的冗余度，目前微博核心 web 系统的致命线设定为 1.6 倍冗余。

警戒线。当服务池的流量到达警戒线时，再结合业务指标综合考量是否需要扩容。以微博核心 web 系统为例，如果流量到达警戒线，并且 1s 的慢速比超过了 1%，也需要进行扩容。

安全线。主要用于决策缩容，当服务池的流量在安全线以上，则可以进行缩容。

准

主要体现在两个方面：防抖动和合理扩容。

防抖动。当系统的承载量到达临界值时，理论上要进行扩容。但还要考虑在实际线上系统运行时，经常出现系统偶发的抖动现象，避免因为偶发抖动触发临界值进行扩容带来不必要的机器成本。为此，我们在智能弹性调度系统中，设定了 1min 的采集周期，并设置了 5min 的滑动窗口。在这个滑动窗口内采集的 5 个点中，任意满足 3 个点即判断需要扩容。

合理扩容。智能弹性调度系统会根据系统当前的水位线，以及系统当前机器数评估出合理的扩容机器数，按需扩容，避免浪费机器成本。

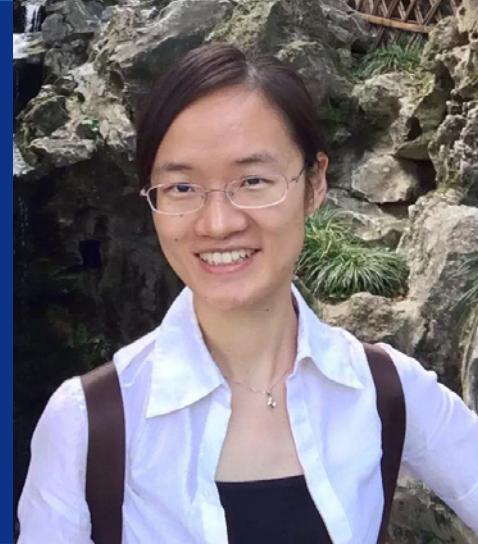
混合云平台 DCP

混合云平台 DCP 主要职能是向内部私有云和外部公有云申请机器，并部署服务扩容到线上服务池。目前微博混合云 DCP 平台，具备 15 分钟内扩容 1000+ 服务器的能力，关于这部分的介绍不是本文的重点，感兴趣的童鞋关注微博 OpenDCP 的 Github 主页：<https://github.com/weibocom/opendcp>。

胡忠想，微博服务化项目架构师、技术负责人。

2012 年加入微博工作至今，承担过微博计数器架构升级，春晚和奥运服务保障，以及微博 feed 业务架构升级等工作。

微信视频通话技术的演进之路



在 2017ArchSummit 全球架构师峰会上，腾讯专家研究员、微信视频技术负责人谷沉沉将以《数亿微信视频通话背后的视频技术二三事》为题发表演讲。她将介绍微信视频通话的基本框架，以及在微信视频通话技术发展不同阶段的关键视频技术环节，探讨如何打造一个适合移动端视频通话的实用视频编解码器，如何适应不同网络下的传输，如何适应不同的设备和内容场景进行视频图像处理，以及如何评价海量用户的视频质量等问题。本次对谷沉沉老师的采访是对大会演讲的一次预热。

InfoQ：谷老师，请向我们介绍一下微信视

频通话功能的发展历程，每个阶段解决了什么问题？

谷沉沉：微信视频通话功能最早是在 2012 年 7 月的微信 4.2 版本发布的，从开始研发微信视频到现在，我觉得大致可以分为三个阶段：

第一个阶段主要是为了让视频通话在手机上“跑起来”。在第一个微信视频通话版本发布前后的那段时间，当时的移动设备的计算能力还比较低，单核主频 1.0GHz 的手机已经算市面上比较好的手机了，所以初期的视频通话版本对技术的计算复杂度和实现优化要求非常高。我们团队凭借早期在移动端的技术积累，搭建了一个轻

量的、适合移动端视频通话的框架，并通过提升视频编解码速度、视频引擎代码性能优化，解决了“跑不动”的问题，当时我们在同等设备、同等视频清晰度下所能达到的通话帧率也是处于业界领先水平的。

第二个阶段主要是设备和网络适应性提升。随着设备处理能力不断增强，视频通话的帧率、分辨率、码率也都在不断提升，而且移动网络的带宽等服务质量等也在不断提升，同时微信海量用户的设备和网络的差距也在拉大，因此网络和设备的适配策略就变得非常重要了，这也是一个长期投入不断优化迭代的难点问题，通过这几年我们在码率控制、传输策略、容错保护等方面的持续研究，目前海量用户整体视频通话质量都有了非常明显的提升，当然很多细节策略我们现在也仍然在不断调整优化，力求为不同设备和网络下的每一位用户都提供尽可能优质的视频通话。

第三个阶段是视频压缩效率、主观质量提升。由于设备硬件处理能力的提升，目前很多手机都是四核、八核等，这样第一阶段仅仅为了“跑得动”而设计的技术跑在这些高性能设备上，CPU 还是有富余计算能力的，所以我们开始研究加入一些复杂度较高的视频编码、图像前后处理等技术，提高压缩效率，在同等带宽下获得更高的视频质量。由于设备硬件不断升级，这类高压缩率的编码算法、高质量的图像前后处理技术研发也在持续升级，将来我们的视频通话会更加清晰、更加流畅。

InfoQ：由于随着硬件设备和带宽质量的提高，用户对视频的分辨率的要求，故现在的编码器，更倾向于支持高分辨率，那对于未来，腾讯的编解码器会更倾向于解决用户的哪个痛点？或者说编解码打算往哪个技术方向发展呢？

谷沉沉：视频编码器本身都是可以支持不同分辨率的，用什么分辨率属于编码器应用策略的问题，关于视频编解码器技术本身，腾讯的视频编解码器关注的核心其实跟标准组织、做商业编解码器的公司等是一致的，都是为了获得更高的压缩效率（即同等质量下尽量降低视频码率，或者说同等视频码率下提升视频质量），同时控制编解码的计算复杂度。当然不同应用对编解码技术的需求侧重点是不同的，比如实时视频通话有低延时、低复杂度的要求，而视频文件的离线存储应用更关心压缩效率，编码复杂度可以很高，我在 12 月份 archsummit 的分享内容里也准备了一些不同视频应用对视频技术的需求对比，到时候欢迎大家一起探讨。

InfoQ：目前微信视频通话不清晰、不流畅的问题仍然不时会出现，这是什么问题造成的？为此，你们打算通过什么技术方法解决呢？

谷沉沉：由于微信视频通话面向数亿微信用户，网络环境、设备性能差异非常大，这些因素都会影响通话质量，“众口难调”——也是海量用户互联网视频通话应用的一大难点。因此我们在视频编解码、图像处理、传输容错等方面做了很多与网络状况、设备性能相适应的技术，为各种不同网络和设备的用户提供尽可能优质的音视频通话质量。最近几年我们针对弱网、中低端设备通话质量差的难点问题做了很多优化，可以为这部分用户提供基本质量的视频通话。虽然我们会从宏观统计数据上监控和检验每次优化效果，但是不排除小部分视频质量异常的场景，所以大家如果有发现自己的网络、设备良好的情况下，微信视频通话存在不清晰、不流畅的问题，欢迎反馈给我们，帮助我们进一步完善。

InfoQ：随着实时通信 RTC 技术栈快速演

进，苹果宣布 Safari 11 对 WebRTC 的支持，实时通信技术在主流浏览器端实现统一，那么微信或 QQ 现在是否已经使用了 WebRTC 组件？有这个打算吗？

谷沉沉：我们微信这边的多媒体技术团队在音视频编解码、前后处理、传输等技术上有着十多年的技术积累，因此没有使用 WebRTC。目前微信这边暂时还没有考虑过使用 WebRTC，以后要看产品应用和技术发展情况再做评估。

InfoQ：您怎么看待 WebRTC 的技术发展前景？

谷沉沉：WebRTC 包含了一套完整的实时音视频应用的开源解决方案，之前我也了解到很多做音视频通信、直播等业务的团队可以基于 WebRTC 快速搭建起自己的产品应用平台，而且 WebRTC 中涉及的音视频格式和通讯协议等都是公开的、标准的，平台系统兼容性较好，未来也有利于不同产品的互联互通。

但是考虑到不同的业务应用场景，如果要影音视频服务做得更好，比如从 80 分到 90 分，甚至 100 分，应用方还是需要做很多针对性的优化来提升性能和产品体验。

谷沉沉，腾讯专家研究员，微信视频技术负责人。目前主要负责微信视频通话、朋友圈视频图片等业务相关的视频图像技术研发和团队技术管理。拥有丰富的视频技术研究与应用实战经验，在国际视频领域知名学术会议刊物上发表多篇论文，数十项视频技术领域的发明专利在国内外获得授权，其中两件独立发明的专利荣获中国专利奖。

当AI遇上社交网络：专访 Tumblr数据科学总监李北涛



Tumblr，一款很有特色的社交 App，自诞生以来，就吸引了不少社交达人下载使用。不少人只知道这款 App 简约易用，却不知道这背后应用了怎样复杂的技术。2017 年 Arch Summit 全球架构师峰会北京站，Tumblr 数据科学工程总监李北涛先生将出席，为我们解读 Tumblr 背后的技术。

InfoQ 对李北涛先生进行了专访，当 AI 遇上社交网络，将会碰撞出怎样的火花？

AI 技术支撑的新型社交网络

AI 时代，人工智能已经一点一滴渗透进了人们的日常生活。除了那些看上去高大上的“黑科技”产品，比如无人车，智能机器人之外，每天都离不开的社交网络 App，为了给用户带来更好

的使用体验，也下了不少功夫，用 AI 来武装自己。

Tumblr 就是这样一款 App，在应用了当下很火的相关性反馈技术之后，一天之内的推荐效率甚至提高了 100% 以上。

据 Tumblr 数据科学工程总监李北涛先生介绍，相关性反馈，即 Information Retrieval（信息检索），它的原理其实很简单：主要是通过用户反馈优化搜索排序。由于用户反馈数据的价值，特别是含量用户的反馈数据，Tumblr 将这些数据实时的应用在推荐系统的优化上，才能够达到如此惊艳的成果。

在此之前，李北涛说，Tumblr 的算法还比较初级，也因为没有海量用户反馈信息，所以基本比较固定。在加入了实时反馈信息之后，算法相

“ AI+社交网络大数据已经极大的改变了高科技产业，甚至对社会产生了深远的影响。 ”

当可以自动进化，用户信息越多，进化越快。这和 Reinforcement Learning（加强学习）也有联系。相当于让算法在实践中不停尝试，让它不停学习，自动优化。

而针对不同的应用场景，Tumblr 做出的推荐也是不同的，譬如当用户离开 Tumblr 一段时间之后，推荐系统会根据用户在离开之前的使用情况进行预测，当用户返回之后，在首页出现的“当你离开”标签下，会推荐一些经过预测用户应该会感兴趣的内容。李北涛表示，推荐的内容有时候是通过算法进行排序的，不一定完全按照时间排序，这也就是为什么用户在首页看到的内容不全都是关注人发布的最新的内容。李北涛说：“这也是 Facebook，Twitter 等大多数社交网络公司的通用做法。这也是推荐的一种。其实单按时间排序也是很自然，很简单，很美的一种算法。我

觉得轻易是不要打破这种顺序。只有在有很强信号的基础上，才可以使用其他信号来调整这种排序。”

Tumblr推荐系统背后的男人

作为 Tumblr 推荐系统背后的男人，我们对他的经历也进行了简单的采访。

李北涛先生毕业于中国科学技术大学少年班，后来出国先去斯坦福读化学物理。97 年的斯坦福，应该算是 CS 的中心。那时候雅虎风头正劲，谷歌在酝酿中。李北涛自言是沾了点斯坦福的仙气，先辅修了一个 CS 的硕士，在学了几门课，并且都取得不错的成绩之后，他对计算机的兴趣越来越大。同时他也渐渐觉得辅修时间不够用，得全部时间投入才行。于是他坚定了自己的想法，李北涛说：“正好我后来导师当时斯坦福博士毕

业，要去加州大学做教授，我就去和他讨论，帮着做一点小项目。最终我就成了他的第一个博士生。这应该是我的一个重大转折。方向定了以后，坚持并不难，因为觉得这是一条适合自己发展的路。”

正如李北涛所说，这是一条适合他发展的道路，有了扎实的学术基础，李北涛在求职之路上也可以昂首阔步地走着：他先在一家叫 Teoma 的公司做搜索，搜索质量一度和谷歌不相上下；后来在 Etsy 做电商，李北涛自己开发了一套新的算法显著提高了搜索效率；几年前他加入 Tumblr——当时在纽约首屈一指的 Startup。“在纽约地区 Tumblr 的企业文化算是很好的。工程师文化浓烈。我主要负责推荐系统。在三到四年中我们的推荐系统得到很大发展。”李北涛及其团队研发的推荐系统对公司的社交网络构建的贡献从不到 1% 上升到超过 50%，推荐成功率增加了

二十倍左右。成为举足轻重的产品。目前，李北涛负责整个数据科学的 R&D。

AI与社交网络

社交网络已经大量的深入的使用了 AI，社交网络的海量数据和用户关系极大的促进了 AI 的发展。AI 让社交网络可以精确勾画用户图谱，精确推荐内容，其他用户和广告。

李北涛认为，AI 的加入让社交网络成为继搜索之后的一大广告金矿，这也是 Facebook 能够挑战 Google 广告地位的基本原因。所以 **AI+ 社交网络大数据已经极大的改变了高科技产业，甚至对社会产生了深远的影响。**

李北涛，Tumblr 数据科学总监。大学就读于中国科学技术大学少年班。后赴美国加州大学圣芭芭拉分校留学。获计算机工程博士学位。主要研究领域在机器学习，数据挖掘，和多媒体技术。毕业后在多家 Startup 任职。涉足搜索，电子商务，社交网络等领域。现任 Tumblr 数据科学工程总监。擅长于创造性开发产品和算法。对用户体验和产品方向有较深理解。

Feed是什么？在知乎上如何应用？



我们每天都在刷知乎、刷微博、刷Twitter，推文一条接一条地从你的指尖流过，信息就这样被你获取了。可你有没有想过背后支撑着这信息流的技术是怎样的呢？

12月份ArchSummit大会上，来自知乎首页面组的技术负责人姚钢强将进行演讲，主题是“知乎Feed流架构演进”，在此之前，我们对姚老师进行了一次采访，请他为我们简单介绍了一下Feed的相关知识，同时分享了知乎Feed的发展历程。以下内容由采访整理而成。

什么是 Feed

什么是Feed？你现在手机上的知乎、微信、微博等App中就存在着Feed。

不太严谨地划分，用户获取信息的方式可以分为Pull和Push，即“拉” / “推”。再详细一些，用户主动去获取、并且明确指定自己希望获取的信息的过程是Pull，而在不明确表述自己的信息需求的情况下，被动接受信息的过程是Push。

举个例子：一个人走到图书馆，按照字母表找到某一本书，获取这本书的内容，可以看作是主动寻找信息；当这个人坐到电视前，他并不知道可能会有什么内容出现，这可以看作是被动接受内容。

Feed从英文翻译过来是【投喂；供给；满足】的意思，Feed流产品形象地将用户等待接受信息的场景描绘为用户被信息“投喂”。Feed流



在中文里可以翻译为“信息流”。如果将 Feed 描述为一个信息准备被投喂，那么 Feed 流则是多个准备投喂的“信息”像流水线一样按照一定规则排列好。

Feed 的特点

现在，当大家讨论 Feed，一般来说会特指一列可以不断向下滑动不断加载的信息列表。Feed 具备以下几个特点：

- 使用简单，主要操作只有点击、向下滑动加载。
- 信息量大，每个 Feed 条目都是一个独立的信息。
- 兼容性好，可以在 Feed 中展示文字、图

片、视频、甚至是可操作的卡片。

-

它被大量地应用在不同领域的不同应用上，也成为了用户上网消费大量内容时，最熟悉的交互模式。

Feed 流产品的发展历史

Feed 流是一个非常笼统的说法。在我看来，将一类信息按照一定规律进行排序就可以称为 Feed 流产品。由于其同时兼顾了大量信息下的展示与消费，如今 Feed 流产品越来越被大多数互联网产品所使用。

早期互联网中，门户网站上一列列的文章标题，可以算作最早的 Feed。由编辑筛选，每个

人看到的都是一致的内容。随着媒体式的中心化发声方式从线下纸媒，理所当然般继承到了互联网上，不论中美，在最初的互联网世界上都出现了许多门户巨头。

此时门户主页可以看作是最原始的 Feed，这个时候并没有后来流行的“订阅”及“个性化”等信息分发方式，所有人接受到的信息是相同的。

互联网上基于“订阅”的个性化 Feed 开始被较大规模使用是从 RSS (Really Simple Syndication，一个互联网信息来源格式规范) 开始的。在 Web 时代，每个站点会独立发布消息。由于网站数量的爆发式增长，RSS 协议开始被用户用来订阅独立站点发布的消息，这样，用户可以在一个 RSS 阅读器上看到所有订阅站点发布的消息，而不需要再去不同的网站接受信息。

从 RSS 阅读器开始，这种“持续更新不同来源的信息，并呈现给用户的信息组织形式”也就正式拥有了“Feed 流”的名字。

而接下来，随着互联网的发展，“订阅”行为已经不再局限于 RSS 协议这个媒介。一系列产品的出现，开始允许用户以极低的成本发布信息，并非常容易的在一个产品中关注其他发布信息的人，比如：Facebook、Twitter、微博、知乎、微信的朋友圈等。

由于 Feed 流产品可以将所有订阅的信息源（信息创作者 / 信息类型 / 信息发布者等）发布的信息汇集到一个地方，同时按照一定的规则排序展示出来，大大提高了用户接受信息的效率，这也成为了这些产品的主要消费场景。

随着互联网产品和技术的演进，将 Feed 的使用成本再次降低。之前主流的 Feed 大都依赖“订阅”这一关注关系实现，而此时可以在取消了关注行为后，直接根据用户的信息消费历史，

使用数据挖掘的手段，计算使用者可能对哪些内容感兴趣，从而进行 Feed 生成。

这将 Feed 的使用成本再次降低，使用者的门槛变得无比之低，越来越多的产品开始向此方向尝试，例如 YouTube、Facebook、知乎。

总的来说，Feed 中的内容来源有以下几种：

- 由少部分人发布。
- 基于订阅行为获取。
- 由机器计算。

当然，也可以是他们 3 者的任意混合所组成。

Feed 流排序规则演进

最开始 Feed 是一类信息按照一定规则排列，上面讲述了 Feed 的来源与发展，那么排序规则同样存在这个“进化”的过程。

最初的 Feed 是编辑按照人主观的意识，来决定不同文章的排序，就像编辑一本杂志，所有会看到相同的内容与序列。

在“订阅”行为变得普及以后，不同的人由于订阅内容的不同，会有完全不一样的 Feed 流产生，显然这时不会再有编辑为每一个人排序。

在各种类型的信息订阅产品中，信息的时效性都是一个比较重要的考量，一个最简单也是比较符合使用者需求的做法是：把所有订阅的内容按照时间倒序排列，将最新生产的内容展示给用户。早期的 Facebook、微博、知乎，现在的微信朋友圈皆是如此。

而当用户订阅数量突破一个阈值时，订阅所产生的内容量会远远大于用户可消费的量，如果此时仍然按照时间倒序排列，显然是一个非常低效的做法：用户最关心的信息并不一定是最新产

生的信息。

帮助用户把最可能有价值的信息筛选出来则可以大大提高消费效率。所以有人在 Feed 排序中引入了“排序算法”。排序算法在初期是非常简单且粗暴的：

- 认为来自某些高价值订阅源的信息更重要——提升排序中的权重；
- 认为带有图片的文章对用户更有吸引力——提升排序中的权重；
- 认为提升排序中的权重；
-

也就是全靠主观判断，觉得哪些地方重要就提升权重。

第二阶段，会根据用户的行为动态地调整内容权重，不再是全靠产品运营者拍脑袋决定。比如你经常点击某个订阅源生产的内容，给某些文章点赞，评论一些人的回答等等。因为具体产品形态不同而产生不同的各种交互行为都在暴露你的喜好，甚至不点击任何内容也是一种反馈信号。

但即使是在第二阶段中引入了用户行为作为排序的依据，却依然有产品运营者在对各种行为下主观判断，这种主观判断是否和用户的真正需求匹配？根本无从分析。

而随着产品复杂度的提高以及用户量及分发量的急剧扩大，越来越多的信息开始被收集到：内容的长度、视频阅读时间、包含的链接、好友有没有读过、跟你比较相似的用户是否喜欢看、标题中的关键字等等，人类大脑没有办法为如此多的变量设定一个权重，将排序交给“机器学习”来进行则是一个必然的结果。

做好 Feed 的关键要素是什么？

1. 快：新生产的内容能第一时间出现在 Feed 流中
2. 准：尽量推荐和用户的兴趣、调性相关的内容
3. 优：保证内容优质，过滤掉谣言、反智、低俗的内容
4. 多：保证多样性，帮助用户发现更大的世界

知乎 Feed 的发展过程与规划

知乎 Feed 的发展经历了以下一系列过程：

第一阶段

2011 年，知乎上线，在初期用户积累阶段，Feed 是基于用户间的关注关系，将每个用户的动态按照时间倒序的形式展示。这是起步时采用的方式，简单易懂。但随着用户量的上涨，刷屏、新内容量过多等问题很快就暴露出来。

技术架构上采用了推模型，当用户产生新的动态时会向他的每个关注者进行推送。这种构架逻辑简单，实现容易，响应时间快。但是随着用户量的增加，推送方式资源占用多的劣势随之显现出来，特别是对于关注了很多人的用户，动态推送需要占用大量的资源，推送的速度也随之变得非常慢。

第二阶段

2013 年 11 月，知乎参考 Facebook 所提出的 EdgeRank 算法模型，上线了新的 Feed 流产品。此时的知乎 Feed 流主要根据 Affinity Score（用户与 Feed 源的亲密度），Edge Weight（权重），Time Decay（时间衰减）来进行排序。

随着用户量的增加，技术构架上从“推”转

换成了“推” / “拉”结合的方式，节省了大量资源占用。但是由于 Feed 的计算逻辑都放到了在线，响应时间急剧变慢。

为了解决这个问题，我们对用户按照活跃度进行分群，为每个用户计算活跃度，然后根据用户的活跃度离线提前计算，只有非活跃用户实时计算。这样活跃用户访问的都是缓存，速度很快。但是这种架构也存在问题：

- 用户产生的动态不能实时分发；
- 算法策略不能实时调整；
- 离线计算策略维护复杂；

相比第一阶段，知乎 Feed 的 CTR (Click-Through-Rate) 和 Duration 都有了 20% 左右的提升。

第三阶段

2017 年初至今，知乎上线了基于机器学习的排序算法，采用 GBDT 算法，根据用户维度的特征、内容维度的特征、以及交叉特征来进行排序，更重要的是加入了内容质量的判断，质量高

的内容会得到更好的分发。

技术构架上采用了计算接近存储的设计方案，使用 Redis Module 将部分固定的计算逻辑放到 Redis 中进行计算，去掉了所有的离线计算，使用户的动态能够快速分发，算法模型能够实时调整。

相比第二阶段，这次进化取得了较大的成果。Feed 的响应时间的 P95 降低了 45%，资源占用减少了 40%，CTR 和 Duration 分别有 100% 和 40% 的提升。高质量内容分发占比提升 10%。

未来规划

在算法角度正在基于深度学习搭建更加个性化的推荐模型。技术构架上使用 Redis Module 后的 Redis 的动态扩容和比较多内存占用也是正在解决的问题。产品上我们会在 Feed 上做进一步升级，将会尝试将关注关系产生的 Feed 内容和推荐引擎产生的 Feed 内容做一些隔离，这样能更好地满足用户不同的需求。

姚钢强，知乎首页组技术负责人，2013 年加入知乎，担任首页 Feed 流技术负责人。专注于改进 Feed 流的稳定性和性能，有丰富的编码和工程攻坚经验。在负责 Feed 流项目期间，通过构架优化使响应时间 p95 从 1.6s 降低到 700ms，通过开发规范使稳定性由 99.9% 提升到 99.99%。

musical.ly技术副总裁：短视频的泛服务和融合架构实践



11月10日，今日头条正式与北美知名短视频社交产品 musical.ly 签署协议，将全资收购 musical.ly，在此之前，musical.ly 的全球DAU 超过 2000 万，但国内鲜有人知道这家公司和其业务。

2014年初，从易宝离职的阳陆育 Louis 拉上同样热爱音乐的前同事朱骏一起创办了 musical.ly。这是一个强调音乐元素的短视频应用，用户可以先拍摄一段生活中的视频，然后在乐库中配上音乐，从而快速地创建时长 15 秒的 music video；或者从喜爱的歌曲中获得灵感，配上一段有意思的画面，然后分享给自己的朋友。

InfoQ 很荣幸地在会前采访到他，请他聊聊 musical.ly 发展的这几年在架构上的迭代经历，

以及未来支撑庞大业务背后的数据平台架构该如何搭建等思路。本文基于采访整理而成。

musical.ly功能模块及架构

木喜老师首先简单介绍了 musical.ly。他说，musical.ly 是一个短视频社交应用，从某种角度看，可以类比为短视频的微博，包含视频发布，视频信息流，评论，关注，点赞，用户主页，私信，通知，话题等功能。

短视频社区是一个与娱乐内容分享和观看为主要特征的社区，所以短视频的信息流和推荐无疑是用户用得最多的功能，实际上用户进入应用直接就到信息流页面，musical.ly 是一个内容创建非常活跃的社区，平日每天创建的短视频数

接近 1000 万，周末更超过 1500 万。视频的生产和消费是驱动社区发展的主干功能，而支撑社区互动的点赞，评论和关注则是紧随其后的用户用得最多的功能。

紧耦合架构的优势和劣势

我们关注到，musical.ly 最初的运营系统采用了传统的紧耦合架构，这样的架构固然存在优势和劣势，木喜老师回忆说，早期 musical.ly 的架构不是为大规模站点设计的，而且从今天的观点来看，这也是当时最能适应业务早期快速迭代和功能验证的架构。

当时采用的方案是经典的 tomcat 加上 SSH 架构，数据库用 MySQL，缓存用 memcached，简单可靠，不需要开发人员有复杂系统经验，所有的功能用一个 codebase，业务稍微成长也能通过简单扩机器解决。实际上，在创业早期进行复杂的架构设计带来了额外的成本，对快速推进业务有害无益。简单，快速，满足需要，是这种单一架构的优点。

musical.ly 这套架构一直支撑到接近 100 万日活，但是这样的架构劣势也比较明显：

- 随着业务的发展，功能变得更加复杂，人员也逐渐扩充，代码变动频繁，问题变得越来越多，各种功能同时在上面修改，发布导致的问题越来越多，代码极度臃肿；
- 用户规模的扩大，数据库不堪重负，各种问题导致线上故障频发甚至宕机。

正是在这样的情况下，musical.ly 团队开始进行全站微服务化改造。

现在回顾起来，这样的演进路径适合绝大多数的创业公司，就如软件工程强调“不过度、不

过早设计”一样，早期在业务不确定性很大的情况下，把精力放在功能打造上，在业务成长起来后有足够的时间去做更适合当前要求的设计。况且一般的早期创业团队很难吸引高水平的技术人员加入，过早考虑规模问题更不可取。

微服务架构改造

musical.ly 技术团队都有 Java 背景，刚开始进行改造的时候框架选用的是 SpringCloud 框架：

- 一方面是该框架以 Netflix 的软件栈为基础，而 Netflix 则一直被奉为微服务实践的典范；
- 另外一方面 SpringCloud 框架的功能也相对全面，所以在跟 Dubbo 进行比较之后选择了前者。

在微服务架构基本成型之后，根据经验，团队对框架本身做了较多的改造，替换了更友好的注册中心 Consul，采用了 gRPC 作为远程调用框架，用 Protobuf 作为序列化框架，替换了熔断和限流方案，集成了故障诊断和追踪功能等等，这些改造对业务是透明的。

musical.ly 的应用层架构从上到下分为：

- API 网关
- API 接口层
- 聚合服务层
- 业务层
- 基础服务层

遵循的基本原则是：服务只能从上往下进行调用，同层之间不能互相调用，同层和底层业务对上层业务的耦合只能通过异步消息进行。这保证了接口调用深度的可控和最快的响应速度

社交应用本质是事件驱动的系统。当用户出发一个动作，会在各个子系统之间形成一系列后续动作，就跟涟漪效应一样，事件会在系统范围内扩散。所以 musical.ly 的架构是面向异步的，尽可能简化同步调用所做的功能，而且尽量减少前端系统对后续动作的感知，把耦合的位置后置到数据库之后。

举个例子，用户对一个视频点了赞，对前端系统来说，做的是一个最简单的操作，就是简单验证用户对视频是否可以点赞，然后就直接添加点赞记录，这保证了用户操作的快速响应。

musical.ly 技术团队在系统中尽可能地使用变化捕捉技术 CDC，实际上就是尽可能地利用 MySQL 强大的 Binlog 功能，通过对 Binlog 的监控和处理，让前端业务无需关注后续业务。CDC 捕获变化后，会把数据库的变化转化为目标事件，注入消息队列，采用了 Kafka，其最好的特性之一是单一事件可由不同的消费者消费多次，这样无论扩展了多少种后续业务，只需要发出不同的消费端即可，对原有系统不需做任何改变，很好地实践了单一责任原则，极大地提升了系统的扩展性和稳定性。

木喜老师说，技术团队在微服务的基础上，提出了泛服务的概念，提出一切皆服务，服务只有协议、功能和有无状态的不同，通过服务注册和发现完成耦合。数据库、缓存、中间件、业务服务都是微服务，这极度简化了 musical.ly 系统的概念模型。

基于云的系统高效部署和优化

目前，musical.ly 在系统运维上也尝试了 AIOps，并做了很多努力，目标是让系统在低风

险的情况下尽可能地实现自动化运维，提高系统的可用性，降低手工干预的强度。

前面讲到泛服务，其实 musical.ly 内部还有另一个概念，叫融合架构，就是从系统设计上，把业务架构、平台架构、数据架构作为一个整体来考虑，把一切运行的程序都 API 化，这就为全面自动化和数据驱动以及更高大上的 AI 留下了切入的接口。

CI、部署、监控、伸缩容都是 API 化的，加上 musical.ly 的系统部署在云上，云本身也是可编程的（这是云最大的优点之一）。这样 musical.ly 系统可以分为两大类，业务服务和为业务服务提供支撑的系统服务。结合服务的全面 Docker 化，实践了 IaC (Infrastructure as Code) 设计。通过 SPEC 来完成对部署和状态的定义，而非过程化的脚本。基于目前的系统服务，实现了两个比较有特色的功能：

一部分是泳道部署，通过把服务分泳道部署实现了前台服务，异步 consumer，核心和非核心服务对依赖服务的相对隔离，很好的实现了柔性和有损系统，极大提高了整站的可用性；

另一部分是系统动态伸缩容，云的特点是资源按使用付费，系统的高峰和低谷的资源需求量差别很大，整套服务可编程化之后，musical.ly 团队可以全动态的来实现资源的申请和编排，甚至充分利用 AWS spotinstance 来实现对保留实例的动态替换来追求成本的极致节约。

接下来，实现系统的单元化部署，使系统能快速实现在全球不同数据中心的快速部署和复制是技术团队的下一个目标。

国际化架构挑战

musical.ly 的用户遍布国内外，那么对于这种国际化业务场景，musical.ly 是如何克服网络、本地化、部署及架构优化等等困难的？

木喜老师说，在世界的不同国家和地区，网络情况差别很大，特别是新兴市场国家，有些地区网络情况非常复杂，运营商众多。主要问题包括 DNS 解析延迟高，跨境距离长导致天然延时长，TLS 握手时间过长，传统 TCP 缺陷导致的初始拥塞控制窗口小、存在慢启动、无线弱网环境丢包概率大等。

musical.ly 技术团队做了很多的努力来优化网络连接，以部分实践为例。

一、网络协议

为了降低 DNS 解析延迟，增加 HttpDNS 部署节点来降低延迟，同时规避 DNS 劫持；

通过修改丢包重传算法，减少重传超时，自适应的初始窗口和更小的连接超时时间来改善弱网体验；

根据移动 App 的特点，预置公钥，去掉 RTT 交互来优化 TLS 握手；

另外，通过多供应商来进行 API 的动态加速和音视频的静态加速，同时在客户端埋点收集每次 API 和互联网访问的耗时都上报的服务器，全面了解各个地区网络的健康情况，对关键指标做实时监控，确保能及时处理网络故障。

二、本地架构

在存储架构上支持多存储架构，把多媒体资源尽量存储到本地，确保最优的访问速度。

对图片资源和视频资源，根据数据分析，压缩成多个不同码率，提高压缩强度，降低了同样

画质所需要的码率，而通过码率自适应，在网络状态不太好的情况下最大程度降低下载失败的比例和加载时间；

在客户端大量使用预加载和缓存，最大程度优化网络流量和提升起播速度，改善使用体验。同时，musical.ly 正在加快推进数据中心的全球异地多活部署，以进一步提升可用性，缩短网络时延。

如何能和 Facebook 一样支持10亿日活用户量？

木喜老师认为：支撑 10 亿日活，挑战来自于服务容量和数据处理能力两个方面。

最基础的还是业务的高并发以及大规模分布式数据服务，如何实现服务的可线性扩展和业务数据的海量存储。

这是泛服务的延续话题。微服务的可伸缩性，业务服务器是无状态的，理论上只有增加服务实例就可以实现业务容量的提升。所以关键在于实现有状态服务的可扩展。musical.ly 的数据库和缓存服务，在泛服务的理念下，也都是通过服务注册进行耦合的。在这块系统的设计上，团队遵循组合优先及尽可能避免重复造车轮的原则，使用一套功能分离的组合服务来实现一套功能完善的分布式数据服务。在几乎不改动开源软件源代码的情况下，通过胶水组件，完成系统的集成和耦合。

数据服务包含存储，分布式，动态伸缩，高可用等模块。以关系数据存储为例，存储是 MySQL，使用 Kinghsard 来实现分布式访问，MHA 用来做 HA，通过 musical.ly 自己开发的一些小组件来实现 MySQL 实例状态和分片信息

再 Consul 的动态更新，以及集群信息更新监控和 Kingshard 配置文件的映射，实现每个组件都干自己最擅长的部分，彼此互不感知，彼此分离，因而实现了最好的稳定性。

努力减少与系统正常进行无关组件与线上系统的耦合，集群的伸缩几乎以离线的方式进行，只发生极短时间的切换，保证了系统的简单可靠。目前的架构设计满足支撑亿级日活没有太大问题，后续会随着日活的增长进一步改进。

张木喜，2015年底加入 musical.ly 蜜柚网络科技，任高级工程副总裁，主持进行全面技术转型。致力于高可用和柔性系统，以及从架构上促进业务和数据算法的融合，通过技术推动产品、运营。在此之前的工作经历是点评架构师，阿里音乐天天动听 CTO，有多年的系统架构、应用架构从业经验，在微服务架构，分布式缓存，数据库和大规模系统监控领域都有丰富经验。2014年加入天天动听，领导产品技术团队，进行天天动听的基础技术架构改造和产品研发。

QCon

全球软件开发大会2018

主办方

Geekbang InfoQ
极客邦科技

[北京站]

北京·国际会议中心

演讲：2018年4月20–22日 培训：2018年4月18–19日

纵览20大热门专题

最低优惠7折进行时

现在报名每张立减2040元

团购享受更多优惠 截至2017年12月31日

大会官网：www.qconbeijing.com
访问官网获取更多前沿技术趋势

如有任何问题，欢迎咨询
电话：151 1001 9061
微信：qcon-0410





Geekbang | **InfoQ**
极客邦科技