架构师

ARCHITECT

特刊

进击的618

SPECIAL ISSUE

June,2017

架构师特刊

Geekbang. 极客邦科技 | InfoQ

序言

在京东作为技术总指挥经历了这么多次 618、双 11，我一直有个愿望——看到我们技术团队的同事们在大促当天能够轻轻松松地浏览商城页面，享受一下促销。这个 618，这个愿望开始实现了。

对于京东这样体量的互联网平台而言，在 618 凌晨那 10 分钟面临的技术压力在整个 IT 技术领域也是比较罕见的。各种“秒杀”、“疯狂 2 小时”等活动让海量用户和订单迅速涌入，给交易系统的每一个环节都带来了巨大的压力。在指挥中心的大屏前，看着代表各个板块业务情况的曲线保持迅猛爬升，没有丝毫迟滞，对于一个技术人，这可能是最欣慰的时刻。

大促对于京东研发体系是一项巨大的系统工程，我们从 4 月就开始了第一次备战会，一共有五千多名研发同事参与备战，涉及到 3000 多个系统和一万多个应用。从 6 月 1 日起，各系统安排 24 小时值班；6 月 16 日起，900 余名核心人员开始集中值班，确保系统万无一失。

为扛住订单洪峰的压力，提供最佳的用户体验，这个 618 京东共制定、更新了 6000 多个应急预案，绝大部分应急预案处理时间均要求在 60 秒以下；各团队共进行了上千次的各种形式的应急演练和压测。

技术应用在于积累、实践，在于团队的责任感、响应能力，如果回望大促中技术体系的出色表现，我觉得除了越来越坚实的基础架构、稳妥的

应急预案、成熟的调度指挥，在这个 618 让京东技术人能放松下来的，还有全面得到应用的人工智能技术。

就像年初京东集团董事局主席兼首席执行官刘强东先生在年会上谈到的，京东的未来 12 年将以技术驱动，我们会用技术来再造每一个环节。在技术本身也是如此，我们正在用人工智能等技术再造每一个环节。

例如在系统压测方面，以往大促前京东要通过憋单等方式进行多次压测，但这些方式也无法准确模拟真实场景。今年 618 我们引入了“军演机器人”，可以相当逼真地模拟大促开始时海量订单涌入的情景，让每个系统都得到充分测试。在客服领域，京东的人工智能客服机器人 JIMI 也在大促中挑起了大梁。通过人工智能算法实现的精准用户意图识别，大数据积累实现的对用户和商品充分了解，JIMI 承担了过半的在线咨询量，应答准确率超过 80%。

这还仅仅是开始。我相信，随着人工智能等技术在京东的全面应用，在技术领域越来越多以往需要人海战术才能解决的问题会逐步被机器人接手，让京东技术人可以发挥更大的创造性，不断提升运营效率、降低成本，提升用户体验。在接下来的大促中，我会看到京东技术人越来越从容地应对，甚至有一天，我们可能不会再把大促当做一次重大技术挑战。

我坚信，这就在不远的未来。

目录

容器技法日趋娴熟，60% 业务已切换至 Kubernetes

作者 鲍永成

容器技术火遍技术界，很多公司包括传统行业企业都已经从观望者转变为采用者。作为最早期采用容器技术的一批先锋者，京东从 2015 年的 9 千多实例扩大到如今容器作为业务上线默认选项，支撑全部业务运行以及中间件、数据库等。此外，在经历了从 OpenStack 到 Kubernetes 的迁移转变之后，京东容器引擎平台已经了从 1.0 迭代到 2.0 版本，并且于今年陆续开源数个项目。

罗马不是一天建成的。

积累如此久并且支撑过 618 大促的京东容器技术是怎样的？有哪些革新又有哪些值得业界学习呢？已经开源的项目是怎样的呢？

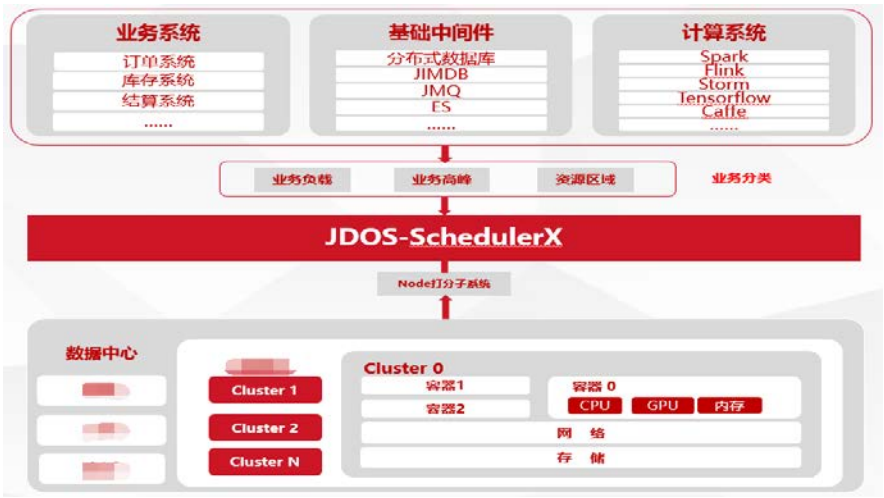
容器技术整体概况

今年随着京东业务的飞速发展，京东容器数量上也对应迅速增加。不仅在去年完成京东业务全面运行在JDOS 容器之上，并且在数据库，中间件等系统也全面容器化。同时，在这一年，京东上线了 JDOS 2.0 系统，开始了从 OpenStack 向 Kubernetes 的迁移。截止到 6 月 7 日，已经有 60% 的业务运行在了 JDOS 2.0 平台中。

此外不得不提及发生的主要变化：业务系统全面基于容器镜像全量上线发布；全面使用集群编排；在生产环境尝试和运行抢占式调度，并自研单层单体全局调度器 SchedulerX; 让业务系统与硬件解耦与资源完全解耦，海量资源，从容大促。

自研单层单体全局调度器

正如上文所述，京东在生产环境尝试和运行抢占式调度，并自研单层单体全局调度器 SchedulerX。



SchedulerX 属于 JDOS 弹性计算项目，主要目的是从更高的层面来加强计算资源调度，以提供服务更强的弹性计算能力，提升数据中心资源利用率。

JDOS2.0 主要通过以下三个维度对业务进行优先级分类归集，并实施

抢占式调度。

1）从业务负载层面来对业务分类，根据业务是长时间运行的任务 (long time running) 还是离线计算任务 (off line)，采用不同的资源占用优先级和调度模式

2）从业务高峰时间段会对各个业务进行归类，例如区分是否白天高峰期还是夜间高峰期，将任务进行混合调度，实现资源的错峰利用。

3）从资源区域层面对业务分类，例如 GPU 资源、CPU 资源、SSD 资源等。根据业务对于资源的实际需求进行调度。

在保证各个业务 80% 的资源的情况下，20% 的资源在不同时间段可以互相抢占借用。（此比例可以根据实际运营进行调配。例如 618、双 11 大促时，则不允许业务相互抢占，保证资源足够）

在当前没有空闲资源的情况下，JDOS 会根据每个机器上运行的业务的分类对机器打分，如果该机器的分数较低，那么抢占就会发生，低优先级的业务首先会被驱逐 (Eviction) 抢占。

被抢占的作业重新回到 PENDING 队列里等待重新调度。

SchedulerX 确保关键业务不会由于资源不足而停止运行，也会重新调度其他业务使其获得更好的安置。

重大决策：OpenStack No, Kubernetes Yes!

应用容器化遇到的瓶颈

JDOS 1.0 解决了应用容器化的问题，但是依然存在很多不足。

首先是编译打包、自动部署等工具脱胎于物理机时代，与容器的开箱即用理念格格不入。容器启动之后仍然需要配套工具系统为其分发配置、部署应用等等。应用启动的速度受到了制约。

其次线上线下环境仍然存在不一致的情况，应用运行的操作环境，依赖的软件栈在线下自测时仍然需要进行单独搭建。线上线下环境不一致也造成了一些线上问题难于在线下复现。更无法达到镜像的“一次构建，随

处运行”的理想状态。

再次，JDOS 1.0 时代的容器体量太重，应用需要依赖工具系统进行部署，导致业务的迁移仍然需要工具系统人工运维去实现，难以在通用的平台层实现灵活的扩容缩容与高可用。

另外，容器的调度方式较为单一，只能简单根据物理机剩余资源是否满足要求来进行筛选调度。在提升应用的性能和平台的使用率方面存在天花板。

OpenStack PK Kubernetes

Kubernetes 方案与 OpenStack 方案相比，架构更为简洁。OpenStack 整体运营成本较高，因为牵涉多个项目，每个项目各自有多个不同的组件，组件之间通过 RPC(一般使用 MQ) 进行通讯。为提高可用性和性能，还需要考虑各个组件的扩展和备份等。这些都加剧了整体方案的复杂性。问题的排查和定位难度也相应提升，对于运维人员的要求也相应提高。

与之相比，Kubernetes 的组件较少，功能清晰。其核心理念（对于资源，任务的理解），灵活的设计（标签）和声明式的 API 是对 Google 多年来 borg 系统的最好总结。而其提供的丰富的功能，使得京东可以投入更多精力在平台的整个生态上，比如网络性能的提升、容器的精准调度上，而不是容器管理平台本身。尤其是，副本控制的功能受到了业务线上应用运维工程师的追捧，应用的扩容缩容和高可用实现了秒级完成。

改造之路

有了 1.0 的大规模稳定运营作为基础，业务对于使用容器已经给予了相当的信任和支持。但是平台化的容器和基础设施化的容器对于应用的要求也不尽相同。比如，平台化的应用容器 IP 并不是固定的，因为当一个容器失效，平台会自动启动另一个容器来替代。新的容器 IP 可能与原 IP 不同。这就要求服务发现不能再以容器 IP 作为主要标识，而是需要采用域名，负载均衡或者服务自注册等方式。因此，在 JDOS 2.0 推广过程中，

京东也推动了业务的微服务化，服务框架的升级改造等。

在近两年随着大数据、人工智能等研发规模的扩大，消耗的计算资源也随之增大。因此，京东将大数据、深度学习等离线计算服务也迁移进入 JDOS 2.0。目前是主要采用单独划分区域的方式，各自的服务仍然使用相对独立的计算资源，但是已经纳入 JDOS 2.0 平台进行统一管理。未来，京东将在此基础上，通过调度将离线计算服务在集群资源充足（如夜晚）时给予计算资源扩充，提高计算的效率。

研发成果，两大开源项目

1 分布式高性能DNS项目

JDOS 是如何支持业务的弹性伸缩的？

对于业务的扩展，直接通过调整副本数，横向扩充容器的实例个数。业务如果是 L4/L7 类型的， 使用一个负载均衡来进行流量的分导。 负载均衡项目 ContainerLB 是京东自研的一套基于 DPDK 实现的高性能 L4 负载均衡服务，主要负责 JDOS2.0 的 service 中 LoadBalancer 的实现。

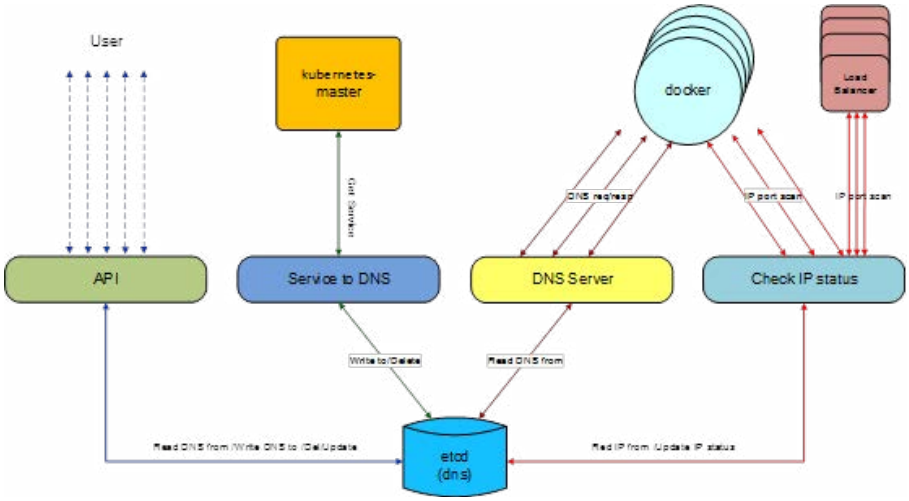
而与 ContainerLB 项目非常密切的还有分布式高性能 DNS 项目 ContainerDNS。(https://github.com/ipdcode/skydns) 为容器提供了内部的 DNS 解析服务。业务如果是微服务类型京东叫 JSF，即需要在 JSF 上进行服务注册与发现的类型，京东则是在容器扩充后，通过服务中间层监听到容器已经启动成功，则对应 Notify JSF。

ContainerDNS，作为京东商城软件定义数据中心的关键基础服务之一，具有以下特点：

- 高可用
- 支持自动发现服务域名
- 支持后端IP+Port，以及URL探活
- 易于维护和横向动态扩展

ContainerDNS 包括四大组件 DNS server、service to DNS 、user API

、IP status check。这四个组件通过 etcd 数据库集群结合在一起，彼此独立，降低了耦合性，每个模块可以单独部署。DNS server 用于提供 DNS 查询服务的主体，目前支持了大部分常用的查询类型（A、AAAA、SRV、NS、TXT、MX、CNAME 等）。service to DNS 组件是 k8s 集群与 DNS server 的中间环节，会实时监控 k8s 集群的服务的创建，将服务转化为域名信息，存入 etcd 数据库中。



user API 组件提供 restful api，用户可以创建自己的域名信息，数据同样保持到 etcd 数据库中。IP status check 模块用于对系统中域名所对应的 ip 做探活处理，数据状态也会存入到 etcd 数据库中。如果某一个域名对应的某一个 ip 地址不能对外提供服务，DNS server 会在查询这个域名的时候，将这个不能提供服务的 ip 地址自动过滤掉。（关于 ContainerDNS 的更多内容详见本系列的另外一篇文章《京东商城分布式智能容器 DNS 实践》）。

2 分布式共享存储 ContainerFS 项目

JDOS 是如何支持有状态服务和无状态服务的？

无状态业务的支持相对容易一些，可以直接通过调度自动调整副本数来实现服务的弹性伸缩。对于有状态的业务，原生的 Kubernetes 有 StatefulSet 进行支持，但是 StatefulSet 需要容器一个个启动， 另外社区在

这个方面开发进度缓慢。因此京东选择了自己定制 Kubernetes 进行支持，主要是为本集 (RC/RS/deployment) 提供了 IP 保持不变和存储自动迁移的功能来进行支持。

通过对于每个副本集维护一个小的 IP 池。当副本数调整时，也对应增加或者减少 IP 池中的 IP 的数量。副本集中的容器创建时，则使用这个 IP 池中的 IP 进行创建；容器删除时，则将 IP 返回到副本集的 IP 池中。

存储也是类似， 对于一个副本集有一个对应的持久化存储 (persistent volume) 的集合。当副本集中的容器创建时，则使用这个 PV 集合中的一个 PV 进行绑定核存储挂载。容器删除时，则对应进行卸载和解除绑定。

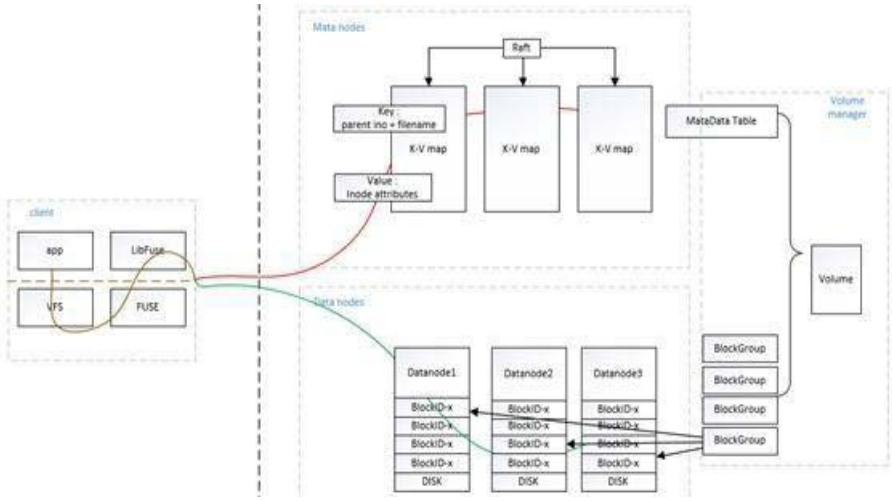
针对于容器的存储， 京东没有选用社区已有的 glusterfs 等方案。而是自研一套分布式共享存储 ContainerFS 的项目来专门提供容器的存储。

Container File System （简称 ContainerFS）是为 JDOS2.0 系统针对性开发的一个分布式文件系统，同时适用于原生 Kubernetes 集群以及其他应用场景。

ContainerFS 的核心概念是：

```
a volume = a metadata table + multiple block groups
```

ContainerFS 的架构图如下：



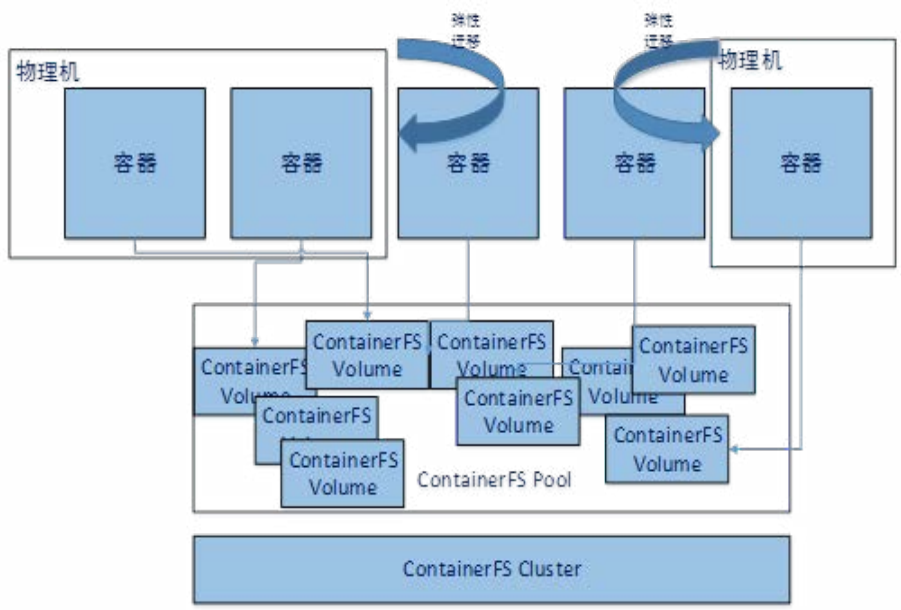
ContainerFS 的产品特性：

- 无缝集成：支持标准的文件访问协议，支持fuse挂载，业务应用无

需任何修改即可无缝使用；

- 共享访问：共享访问帮助多个业务应用获得相同的数据来源；
- 弹性伸缩 ： 可满足业务增长对文件存储的容量诉求；
- 线性扩展的性能：线性扩展的存储性能，非常适合数据吞吐型的应用；

ContainerFS 典型应用：



做为 JDOS2.0 的数据存储引擎，ContainerFS 提供了独享、共享等类型的 volume，并通过 PV 机制挂载给 POD 或者容器使用。

使用效果：

目前 ContainerDNS， ContainerFS 已经开源，ContainerLB 会近期在 GitHub 上开源。

写在最后

为什么要将京东底层技术开源呢？主要两个方面原因。

在底层技术方面，开源是大势所趋。Google 的 borg 系统在过去十余年间一直处于保密状态，但是现在不但公开了，而且利用起核心思想，孵化出了 Kubernetes 项目。而 Kubernetes 项目一经发布，也立即受到了热捧。同时，社区的完善也为 Kubernetes 和 Google 的 borg 提供了更为有益的建议和帮助。当然，不仅仅是 Google，CoreOS、OpenStack、Docker 等等公司和项目的开源大热也说明了这一趋势。

在容器平台实践路上，京东是走的比较早也是比较坚定的。在实践过程中有很多理解和技术视野。比如我们认为容器技术本质是 linux kernel 技术，容器技术需要数据中心底层基础软件全力配合，如分布式域名解析，高性能负载均衡，分布式共享存储，精确授时，等等。

因此京东在这方面不希望闭门造车，而是能够更多的同业界来分享我们的经验。一方面，为许多底层技术还在摸索中的业内同仁提供一点借鉴和帮助，另一方面，也是希望获取业界的指导，提升京东的基础平台系统和技术思路。

作者简介

鲍永成，京东商城 基础平台部技术总监。2013 年加入京东，负责京东容器集群平台（JDOS）研发，带领团队完成京东容器大规模落地战略项目，有效承载京东全部业务系统和 80% 数据库，特别在大促期间 scale up 秒级弹性应对高峰流量。目前聚焦在京东容器集群 JDOS 2.0 以及京东敏捷智能数据中心研发。服务过土豆网（TUDOU.COM），思科（CRDC）等，在分布式、虚拟化、容器、数据中心建设有丰富的实践经验。

一个中心五个原则，谈谈物流系统的大促优化实践

作者 者文明

在京东的订单流链路中，可以简单的划分为订单前和订单后两部分，我们在京东主站上搜索商品、浏览商品详情、把商品加入购物车、提交并支付订单等环节属于订单前，订单提交之后，订单信息流就进入订单后的物流系统部分。每逢 618 大促期间，大家可能会更多的聚焦到网站 PV、秒杀系统、交易数据、广告收入等等。其实对于京东来说，其很核心的优势来源于精准的时效承诺、极速的送货体验和极致的售后服务，在大促期间，其物流系统的表现对客户体验至关重要。

京东物流系统简介

京东物流系统属于订单生产系统，主要包括订单履约、仓储、配送、客户服务和逆向处置中心等等。图 1 示意了一个简单的正向订单生产流程，逆向生产流程主要由逆向处置中心发起，主要包括售后服务单（退换货等）和安装维修单。

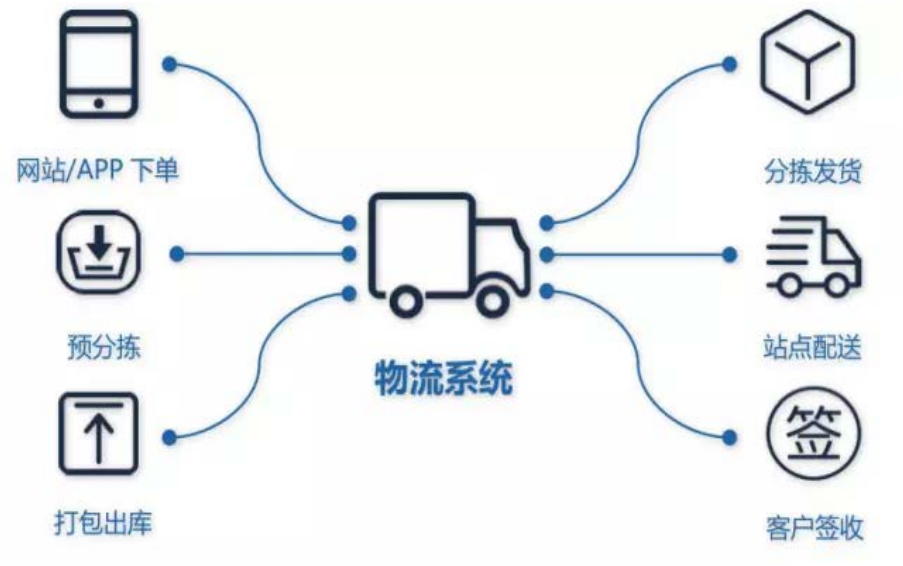


图 1 订单生产流程

京东物流系统有如下 3 大特性：

- 90% 以上为OLTP系统，承载着订单生产相关的所有核心交易流程；
- 领域模型和业务逻辑复杂；
- 强依赖关系型数据库。

以上特性也决定了物流系统的大促备战和电商网站、订单交易、秒杀、搜索推荐、广告等系统会大有不同，在很大程度上系统 70% 以上的性能（容量）取决于 DB 的性能（容量）。因此，DB 是我们每次大促备战的重点。围绕 DB 侧的备战工作，主要聚焦在慢 SQL、垂直和水平拆分、读写分离、

生产库和报表库分离、连接池优化、参数调优等方面。

打不死的小强—慢 SQL

记得刚加入京东第一次负责 618 的时候，在 618 当天就遇到了两次业务反馈系统卡顿的现象，紧急排查发现 DB 中大量连接堆积，再通过查看当前线程发现是一个慢 SQL（耗时 10 多秒）导致了连接堆积，后来把慢 SQL 紧急优化上线后系统恢复正常。从那天以后，我深深感受到了慢 SQL 对我们系统的影响，同时也明白了一点，一个慢 SQL 对我们的系统总是致命的，我们不能放过任何一个慢 SQL。为了说明一个慢 SQL 对系统的影响，截取了两张数据库 CPU 使用率在一个慢 SQL 优化前后的对比图（如图 2），从图中也可以看出，前后对比是非常明显的。

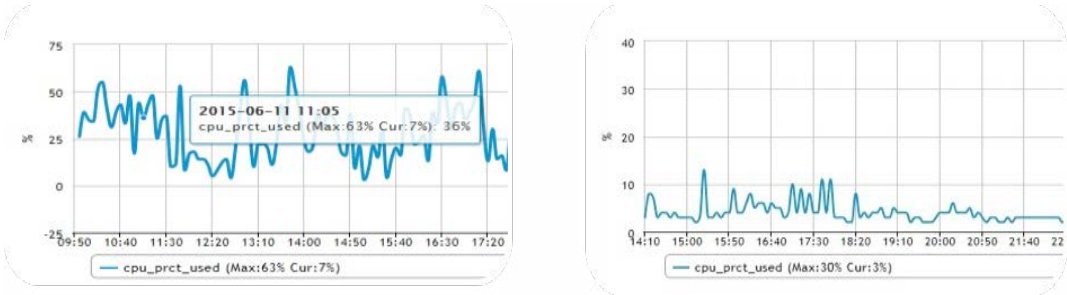


图 2 一个慢 SQL 优化前后 CPU 负载对比

在数据库优化方面，慢 SQL 优化是最重要且效果最好的一项工作，如果要用一个比喻去形容慢 SQL，打不死的小强是再贴切不过的了，慢 SQL 在我们的系统中是灭了一茬又一茬，似乎永远消灭不完。通常情况下，慢 SQL 的出现可能是因为过滤条件中没有索引、SQL 语句写的过于复杂、表中数据量过大，做了全表扫描等等，因此我们在进行慢 SQL 优化时，优先会通过添加索引解决，索引解决不了的才会去优化语法，拆解 SQL 语句，将大事务化小，通过适当冗余来减少关联，优化数据模型，通过历史数据结转减少数据量等等。总之优化慢 SQL 的方法很多，各系统要根据各自的特性和场景选择最优且成本最低的方案。

近几年来，京东的业务一直处于持续膨胀之中，系统中总会不断涌入

很多新的业务需求，这样也就不可避免的引入了新的慢 SQL，所以每次大促，慢 SQL 优化是一大备战重点。

数据库垂直和水平拆分

跟传统的企业应用系统一样，京东的仓储系统也经历过 C/S 和 B/S 时代，V3.0 之前用的是 SQLServer 和 .Net 平台，而且整个仓储管理是一个系统，包括基础资料、库存、入库、出库、在库等，随着京东业务规模的迅速增长，每次大促的单量峰值也由早期的万级增长到了现在的亿级，这中间仓储系统进行了垂直拆分，将基础资料、库存、入库、出库、在库等拆分为独立系统独立部署（如图 3）。垂直拆分之后仓储系统一分为多，系统的容量也就成倍上升。

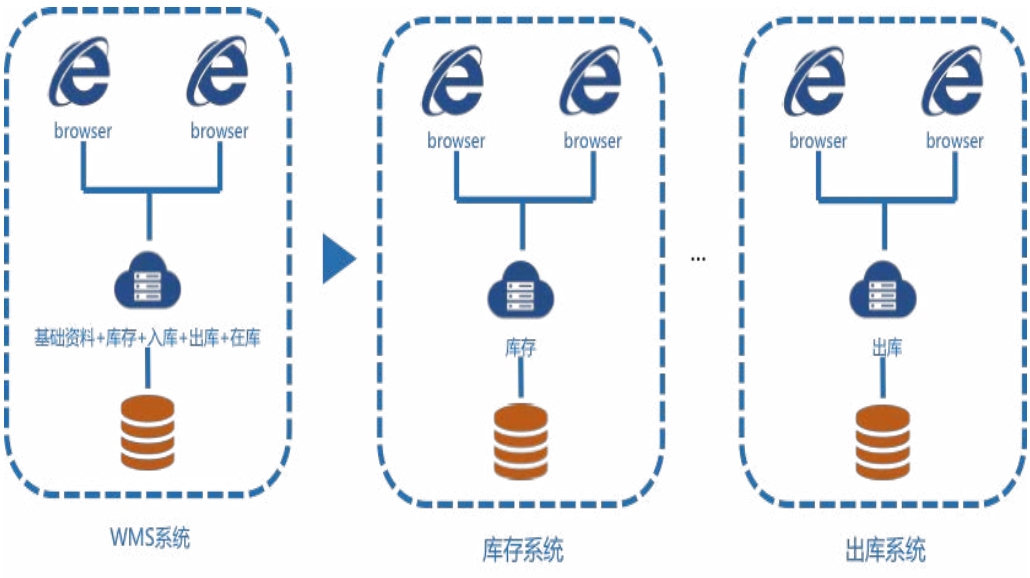
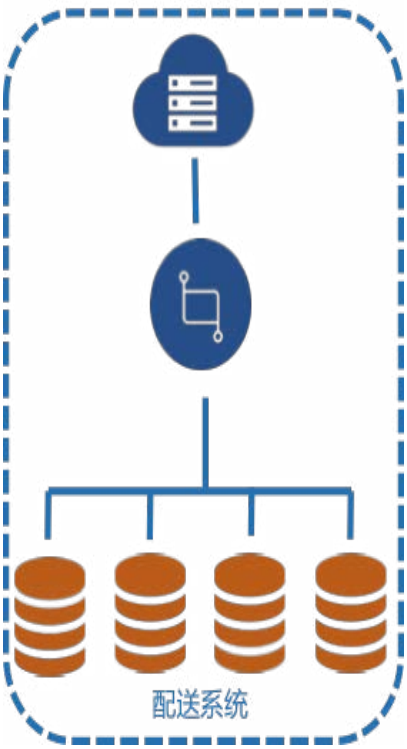


图 3 仓储系统数据库垂直拆分

除了仓储系统，其他很多系统（包括配送系统）都经历了垂直拆分的过程，垂直拆分不但可以很好的解耦系统，还能成倍提升系统容量。

京东的配送系统流量比仓储系统还要大，垂直拆分之后的系统容量不足以支撑大促期间的单量冲击，于是在垂直拆分的基础上又做了水平拆分，水平拆分除了常用的分库分表之外，还有部分复杂业务表的模型水平拆分，

比如运单表，拆分成基础数据、扩展数据和状态管理三个表，有的表也会按读写比例进行拆分，比如将读多写少的列放一张表，读少写多的列放另一张表。图 4 是配送系统进行水平拆分的一个示意图。水平拆分之后，目前系统可以轻松应对大促期间的亿级单量，流量还远远未到系统的容量上限。见下图 4， 配送系统数据库水平拆分。



分离技术

分离技术也是我们每次大促备战中的常用方法，主要包括读 / 写分离，生产 / 监控分离和在线 / 离线分离。

我们大部分系统读写比例大约 10:1，对于关系型数据库来说，主要消耗来源于查询，尤其是复杂查询，所以为了提升数据库端的总体容量，必须尽可能的将查询 SQL 分离到从库上，主库只提供写服务和一些必要的读服务，图 5 中 B 为备份库，R 为从库，所有从库均可提供读服务，一个主库下可能会挂多个从库，多个从库根据业务场景需求可以做成负载均衡，也可以按业务优先级进行隔离并支持灵活切换。这样主库就只负责生产，避免了那些比较消耗性能的复杂查询影响到生产，同时系统的总体容量也会得到大大提升。

生产 / 监控分离指的是生产报表和监控报表必须分离开来，所谓生产报表就是业务生产过程中强依赖的报表，比如仓储系统中的积压类报表（拣货、复核、打包等各环节积压数量），配送系统中的分拣差异报表、配送差异报表，等等。

这两类报表业务优先级不一样，生产报表是要优先保障的，所以在系统中需要将这两类报表进行隔离，避免监控类报表影响到生产类报表。监控报表是一个独立系统，数据来源有两种路径，一种是从生产库通过

binlog 复制过来（我们用的是自研的 Decomb 总线），另一种是从生产库通过消息方式先进入 kafka，再从 kafka 消费到监控系统。因为监控报表业务场景的多样性和复杂性，监控系统的数据库会采用多种技术，比如 MySQL、Elasticsearch、HBase、Cassandra 等等。

在线 / 离线分离指的是在线报表和离线报表分离，在线报表是实时或准实时报表，查看的是 24 小时之内的业务数据，离线报表多为分析类报表，查看的是 24 小时之前的业务数据。因为二者的业务优先级和技术方案都不尽相同，所以必须要进行分离，避免相互影响。

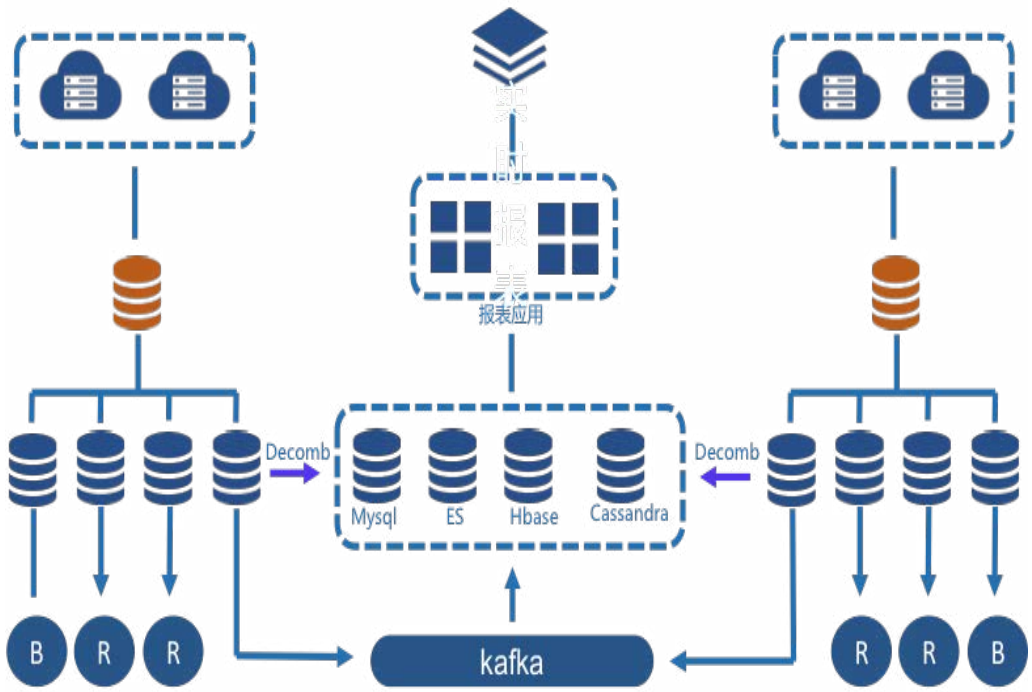


图 5 分离技术

DB+ 技术

经历过多次大促备战之后，给我们最大的感触就是业务规模的增长速度总是快于我们系统的迭代速度，业务规模总是在驱动着系统的迭代升级。面对亿级单量，单纯的引入前面提到的技术已经无法让系统容量发生质的变化，系统容量容易受制于数据库，所以，除了通过分库分表来实现

数据库写的分布式，还需要引入一些 NoSQL 技术，所谓的 DB+，也就是 DB+NoSQL+ 分布式，主要包括如下几个方面的改进：

1. 引入KV引擎，将一些数据从关系型数据库（MySQL）迁移到KV引擎中来存储和处理，这样不仅可以大大降低关系型数据库（MySQL）的负担，还能提升数据的读写性能。京东的物流系统中，引入的KV引擎主要包括Redis、HBase、Elasticsearch和Cassandra，Redis用于缓存相对静态的热点数据，HBase是存储，主要存储海量的业务数据和历史数据，Elasticsearch主要存储查询条件相对复杂的数据，Cassandra主要存储一些日志、流水类数据。
2. 引入数据库分库分表中间件，实现数据库写的分布式，做到数据库读写的水平可扩展，真正实现从Scale up到Scale out的转变。
3. 追求BASE模型，容忍分区失败，弱化事务，大事务化小事务，甚至是无事务，舍强一致性取最终一致性。

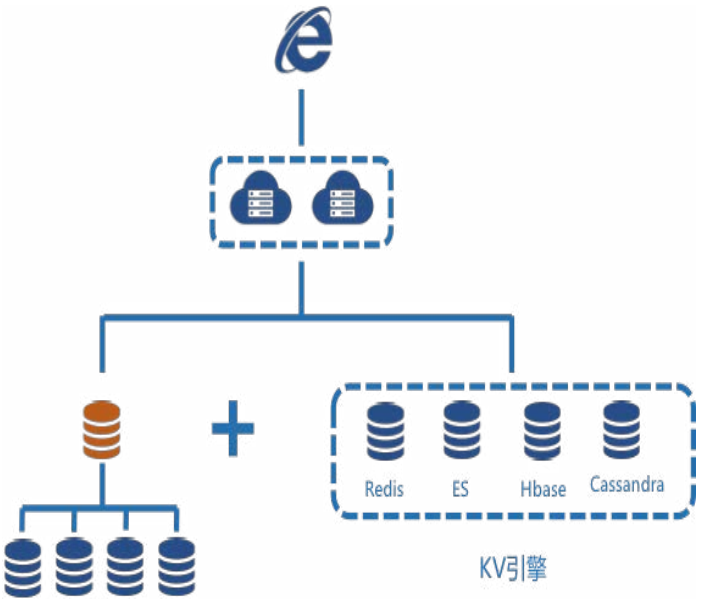


图 6 DB+ 技术

图 6 能简单说明 DB+ 的基本思路，系统的存储分两部分，一部分是

传统的关系型数据库（MySQL），用来存储结构化，强事务数据，数据库做了 Sharding，读写均为分布式，支持弹性扩展。另一个是 KV 引擎，KV 引擎主要包括 Redis、HBase、ElasticSearch 和 Cassandra，Redis 主要用来做热点缓存，HBase 用来存储数据量级大而且 rowkey 又比较固定的数据，ElasticSearch 用来存储查询条件比较复杂的报表、查询类数据，Cassandra 主要用来存储日志、流水类数据，这类数据量级大，读写性能要求也比较高，但是大多都是按 key 查询。

思考总结

在经历过多次大促备战之后，最大的感触是每次大促的业务规模总是在驱动着系统的技术不断的升级。不同的业务量级所需要使用的技术也大不一样，前面介绍的都是每次大促备战的一些技术实践。简而言之，对于 OLTP 类系统来说，面对大促的优化可以总结为一个中心和五个基本原则。

一个中心就是要以数据库为中心，优化数据库性能为先，从数据库端出发来提升系统容量。五个基本原则就是大系统小做原则、大事务化小原则、分离原则、分布式原则和数据库弱依赖原则。下面分别介绍下：

- 大系统小做讲的就是合理的垂直拆分，将一个业务系统按照合理的领域模型拆分成多个可以独立部署的子系统，一方面解耦，一方面提升系统的容量和可扩展能力；
- 大事务化小指的是在业务允许的前提下尽可能将大事务拆成小事务，大事务会严重影响数据库的性能而且容易造成死锁；
- 分离原则就是要根据业务的不通场景和要求和数据的冷热程度等进行数据的分离，避免不同优先级的业务相互影响；
- 分布式原则主要说的是要将数据库的写进行分布式，并且真正做到写库可动态扩展；
- 数据库弱依赖原则简单说就是要尽可能减少对关系型数据库的依赖，能用NoSQL解决的就不用关系型数据库，能异步写库的就不同步写，能最终一致性的就不追求强一致性，等等。

现阶段正处于电商高速发展的黄金时期，业务规模还将持续保持快速增长，京东的物流系统也还将持续迭代和演进。

作者介绍

者文明，中科院硕士，清华大学学士，15 年电子商务 / 企业应用领域研发、架构经验，涉及电子商务、互联网、大数据、人工智能等领域，专注电商物流系统架构、实时大数据、智慧物流等解决方案。2012 年初加入京东，主要负责京东物流系统架构。

商城交易平台的高可用架构之路

作者 郭蕾

据腾讯科技报道，6 月 18 日零点，京东全民年中购物节拉开了高潮的序幕。第一个小时的销售额超过去年同期的 250%。从凌晨开始的海量订单让 6 月 1 日就拉开序幕的京东年中购物节奏出最强音，大量用户瞬间涌入，峰值订单被不断刷新。为了应对如此大规模的流量增长，京东研发团队几乎全年都在高筑墙、广积粮，一直着力从技术层面为用户提供流畅的交易体验，以保证在峰值交易时期系统的高可用性。在京东整个电商体系中，交易系统占据着其中的半壁江山，购物车、结算、库存、价格等相关的环节都包含在其中，可以说交易系统的高可用能力基本上决定了整个京东商城的高可用能力。在过去的一年时间里，京东的交易系统做了哪

些迭代和优化？今年又有哪些创新？整体的交易系统规划是怎么样的？InfoQ 记者带着这些问题采访了京东商城交易平台高级总监王晓钟。

InfoQ：能否整体介绍下交易平台目前的架构体系？

王晓钟：交易平台负责商品、价格、用户、库存、订单等电商核心基础信息的中心化管理，以及对购物车、结算页、优惠券 / 礼品卡、订单中心等黄金交易流程的管控和平台化服务。交易平台致力于技术改变生活，打造智慧营销的交易平台。为用户提供黄金交易流程；为客户提供智慧营销解决方案包含促销建议、智能库存定位等智慧营销工具；为研发团队提供稳定、可靠的交易服务。

渠道是交易的流量入口来源，目前主要包含几大部分，PC、APP、微信、手 Q 等。目前 APP 入口已经占据了整体流量的 70% 以上。

组件完成对现有基础服务的抽象与整合，将现有服务资源以多元化的方式展示给外界，灵活的组织并支持多种协议的交互，最终实现了系统的模块化、服务平台化、功能配置化。组件最大限度的减少外界对内部逻辑的耦合，从而实现对需求快速响应。

基础服务位于整个黄金流程的最底层，其扮演者交易平台心脏的角色。其中商品服务、价格服务、库存服务、用户服务、购物车等更是核心中的核心。

中间件、基础设施是基础服务的基石，对业务系统提供高性能，高可用的技术支撑。

InfoQ：过去一年，交易平台在保证底层的基础平台稳固方面做了哪些事情？有哪些点读者是可以参考学习的？

王晓钟：除了我们一直在做的、已经形成常规的工作，比如线上压测、性能优化、扩容、故障切换、限流、降级之外，过去一年，我们在系统维稳方面做了一些精细化的工作。

核心调用链监控。在黄金交易流程中的各个服务入口点和服务相关依赖、调用方等进行联合监控。当服务性能下降、可用率下降时，可以快速的定位到故障点。把监控和故障解决方案联动起来，比如一键切换、服务

降级、限流等，可以快速的发现和解决问题。

1. **自动切换**。对于成熟的切换流程，比如数据库、缓存、服务等节点的客户端，当检测到故障时，可以根据策略自动切换到健康的节点，同时在故障节点恢复后自动切换回来，减少人工操作的错误和耗时，提高系统的可用率。
2. **异步化编程模式**。部分服务通过彻底的异步化改造来提升吞吐量，还是有一些效果。但是由于纯异步化对于现有系统的改造还是挺大的，所以目前还在尝试前行阶段。
3. **共享资源池**。提前准备一些资源共享池，各服务混用，平时设置较低的权重。当某个服务的常规资源组不足时，则增加其在共享池中的权重，这样可以快速的使用资源，而不用临时扩容。
4. **全链路压测**。从入口开始模拟用户的行为进行压测，流量通过依赖传递，从浏览、搜索，到提交订单以及最后的生产，自动覆盖到链路中的所有环节。配合上面提到的核心调用链监控，解决以往只是单服务的压测，覆盖面不全的问题。

随着业务的发展，功能的复杂度也在不断增加，定位故障原因变的困难了起来，很多时候线上发生故障大部分的时间都在定位问题，故障的解决只要有预案就可以很快处理。调用链监控就很重要，可以站在全局的角度，快速的定位问题，和故障预案处理结合可以解决我们的痛点。

随着服务的不断扩容，机器数量的增加，出现问题时，故障修复的速度变慢，自动化的故障切换可以使人工解放出来，处理更重要的事情，可以让大家不用总是在半夜起来处理故障。

InfoQ：目前交易平台的服务是依据什么维度进行划分的？

王晓钟：目前交易平台主要依据业务能力来划分服务的：购物车、结算页、促销、价格、库存、商品、用户等，为 PC，手机，微信等渠道提供高可靠的大中台服务。

这种划分模式好处在于：

- 架构稳定，因为业务能力相对稳定和相互独立。

- 开发团队是自主的，围绕着交付业务价值而不是技术特性来组织。
- 服务之间共同合作，松耦合。

InfoQ：能否分别从业务、系统、基础设施三个层面谈谈你们的监控体系方案？

王晓钟：在京东这样的大规模分布式系统面前，每时每刻服务器可能都宕机，网络随时可能都在抖动，大量接口调用量日均过亿，同时具有流量聚集效应的促销每天都会有好几波，如果没有一套强大的监控体系，我们就像睁眼瞎一样。经过多年的努力，京东目前已经形成多套监控系统，建立了比较完善的监控体系，时刻监视着系统的健康状态，并在发现问题时第一时间进行预警：

1. 业务层面的监控，主要是核心业务指标，比如实时订单量，并按渠道、省份、运营商、机房、品类、活动等各个维度进行细分，从而在及时发现核心业务指标变化的同时，能够快速定位、排查问题，并做出应急响应。
2. 系统层面的监控，主要是方法或代码块的调用量、成功率以及响应时间。同时，不同语言平台有特定的监控指标，例如Java应用，我们也非常关心JVM GC 情况。这些指标我们会按实例、集群、机房等进行逐级汇总计算。对于响应时间，我们更关心的是TP99甚至TP999任何一指标低于预设阈值都会触发报警。在采集单一接口性能数据的基础上，我们将请求访问链经过的一系列子调用串起来，包括RPC服务之间、访问缓存、访问数据库等等，实现调用链条薄弱环节的快速发现，快速解决。
3. 基础设施的监控，主要是网络质量和机器健康度的监控，像常规的带宽、丢包率、重传、连通性，CPU、内存、磁盘等等。在网络方面，除了内网，我们也非常关心公网网络质量，一旦发现运营商或者区域故障，就会做立即出预案响应，7*24小时确保用户购物体验。

在监控指标完善的同时，我们更多在解决监控自身的延时性。京东自身访问量大，所以在提高监控的延时性同时又不能影响业务自身性能，本身就是就一个挑战。目前我们在业务层面、系统层面都做到了秒级粒度，基础设施方面的重要指标也有了秒级数据。在预警方面，除了传统的邮件、短信，我们集成了京东内部 IM 工具，同时还有手机语音呼叫。

在这么多指标，这么精细的数据面前，传统的监控仪表盘也会让我们再度迷失，因此我们开发了天眼系统进一步将各个监控子系统进行集成，结合前述的调用链，在一个大屏上多个核心主流程的各环节、各调用层次的当前健康状况一览无遗，一旦有故障我们可以在短时间内快速响应并恢复。

InfoQ：对于恶意的流量攻击，京东做了哪些准备工作？准备如何预防？

王晓钟：恶意流量攻击，是每个互联网企业都必须面对的难题。目前我们把流量攻击分为两大类：网络协议层和应用逻辑层。

网络协议层的，主要是 SYNFlood、UDPFlood、DNSFlood、HTTPFlood 这些 4 层或 7 层协议的各种流量攻击，主要以带宽或服务资源消耗为主。目前我们通过京东云平台自研的流量分析和清洗系统能够防御主流的恶意流量攻击。除此之外，信息安全部门也会联合外部力量进行上百 G 的流量攻防演练，确保合作和联动等实战能力。

应用逻辑层的恶意流量的范围和影响则比较广泛。狭义上，恶意流量利用应用系统的软件漏洞，做拒绝服务攻击；广义上，能够利用应用的实现逻辑或规则漏洞，非法实现各种商业利益的，无论流量大小，都属于恶意流量攻击。这一大类型攻击由京东的多个部门配合进行整体防御。

1. 信息安全部门会通过开展安全自查和外部合作报告漏洞的方式，由各业务研发部门实施安全漏洞消除，比如SQL注入、代码执行、水平越权、信息泄露等。
2. 风控部门会通过数据分析，建立各种等级的风险控制模型，形成动态的不同风险等级的账户池，供业务系统使用。

3. 业务研发部门则根据业务特性、用户风险等级、系统压力等因素，提供不同策略的限流实现。

InfoQ：以商品的实时价格为例，聊聊你们的读逻辑和写逻辑流程？

王晓钟：京东实时价格面临几大挑战：一是数据量大，几十亿的商品；二是调用量大，日峰值上百亿；三是实时性要求高；最后是业务复杂度高，并不是单一的京东价，不仅要综合计算各类促销规则，还要对 PC、手机、第三方合作渠道以及区域进行差异化运营。这里，我们运用读写分离、异步化策略，选择支撑大并发、高性能的开源组件进行设计，确保可水平扩展、高稳定性。

1. **写逻辑流程**：当采销在后端调整价格或建立促销时，同步写入MySQL数据库，然后通过异步任务更新促销主Redis数据，并同时更新价格主Redis的过期时间戳，通过Redis自身复制机制，将数据传播到从节点。
2. **读逻辑流程**：当用户在前端浏览商品列表、详情等页面时，异步访问价格实时服务，此时内嵌Nginx 的Lua程序直接读取本地Redis（从）中的价格数据，无过期则直接返回用户；若过期或不存在，则回源访问价格实时计算服务，即刻返回最新价格给用户。
3. **回源写逻辑**：价格实时计算服务读取促销主Redis，在返回最新价格给用户的同时，异步写价格主Redis集群，价格主Redis同步数据至前置Nginx节点的从Redis节点。

InfoQ：今年 618 京东的交易平台都做了哪些技术上的改进或者创新，以及未来将会考虑哪些优化和升级方向？

王晓钟：除了上面提到的主要用来维护系统稳定的技术改造之外，今年交易平台也投入了更多的精力在做提升用户体验、提升 GMV 的改进和创新工作。比如利用大数据技术和机器学习模型，来提供千人千价、千人千促的体验。

我们也在尝试利用大数据和机器学习等在系统维稳上做一些工作，比如：

1. SQL注入和恶意代码执行方面引入了机器学习模型，通过对已有的攻击行为进行学习，训练特征。引入半监督学习，让模型可以通过学习，自动发现新型的攻击。大大提高了攻击的发现效率和新攻击的识别能力。各项指标已经完全超越传统的规则识别。
2. 使用有向图模型对恶意攻击进行溯源检测，更加准确快速的进行溯源分析，并且得到了非常好的效果。

下一步，我们会继续尝试在这个方向上做一些创新，比如：

1. 在人机行为检测方面进行优化。使用聚类和nlp模型对恶意刷单行为进行识别，提高恶意刷单行为的验证级别，从而极大地降低后台接口压力。
2. 评论价值评定模型，识别真实评论和刷出来的评论。让评论产生更大的价值。
3. 我们将在故障智能预测上进行探索。目前很多监控和预警都是事后的，我们希望能做到事前。通过分析历史性、周期性故障数据，结合当前实时健康度，快速识别出“濒死”的机器、实例，真正做到监控预警智慧化。

受访嘉宾介绍

王晓钟，京东商城交易平台高级总监，京东交易黄金流程与智慧营销生态系统的掌舵人，带领的产品与研发团队为京东商城提供了核心交易的系统保证。

如何配合业务打造 JDReact 三端融合开发平台

作者　沈晨

一、诞生背景

1. 无线开发的痛点

React Native 最近两三年之内整个框架在业界应该说是非常热门，很多团队、大公司都在做 RN 的一些研究开发工作。先一起回想下在 React Native 框架出现之前，互联网 APP 开发是一种什么样的模式。最初，大多数同学应该是用原生开发 Android 或者 iOS，再加上 HTML5 内嵌的方式，即 Web APP。之后又衍生出了 Hybrid APP，基于 PhoneGap/Cordova

框架实现了 WebView 的能力强化。不知道大家在做这种开发的时候，有没有遇到过一些瓶颈或者一些痛点，反正我们的团队是遇到了很多。这里总结一下之前传统的方式有哪些问题。

第一，效率低下。因为无论是 Android 还是 iOS，使用传统的原生开发都有一定的开发门槛。而且代码上不能复用，这意味着任何一个业务要在 Android 和 iOS 各做一次开发，测试和业务开发工作都不能复用。

第二，性能比较差。用传统的 H5 开发方式，受限于 WebView 容器的一些瓶颈，导致无论在页面加载还是用户体验上，相比原生应用有比较大的差距。

第三，灵活性不够。因为传统原生开发意味着任何改动都需要发版，在 Android 上因为像国内应用商店非常多，而且涉及到各种不同的渠道包，所以发版成本很大；在 iOS 则受限于苹果的审核机制。对我们来讲，任何这种线上问题处理起来都非常痛苦。

最后，接入困难。因为 Android 和 iOS 平台有差异，所以任何一种垂直业务接入 APP 的成本非常高，很多业务代码和业务流程并不能复用，造成业务团队的开发、接入成本非常高。

2. React Native 登场

说了这么多痛点，我们也在反思到底需要一种什么样的框架来解决这些问题。非常幸运，我们在 2015 年的时候注意到 Facebook 发布了非常具有颠覆性的 RN 框架。简单来说，这是一种跨平台的移动应用开发框架。在当时它非常有颠覆性，因为它最大的特点就是完全用 JavaScript 进行应用的开发，但是最终会渲染成原生的组件。对开发者来说，这意味着你拥有了 Web 开发的效率，同时兼顾了原生的性能。这对我们当时业务的吸引力非常大，这个框架一经推出，国内外很多公司都在用，像 Facebook 自己也在用。在国内，手机百度、手机 QQ、京东 APP 也很早就进行了开发。RN 对我们团队来讲都有哪些优点，或者说为什么要用它，这里大概总结了以下四个原因。

第一，学习成本低。因为它的开发基于 JavaScript，JS 语言本身在开发者当中有非常良好的群众基础，任何一个有经验的前端团队可以快速地上手 RN 开发。

第二，多端代码复用。因为所有的业务都用 JavaScript 开发完之后只有一份代码，然后通过编译打包机制直接部署到不同的平台，如 Android、iOS 甚至 Windows 平台。

第三，接近原生的性能。开发者使用 JavaScript 进行 RN 框架开发，开发完之后在再通过中间的虚拟 DOM。这个实际上是它核心所在，传统的 H5 的应用是跑在 Web View 的容器当中，容器中需要维护一个真实的 DOM，而真实的 DOM 上每一次操作都会有回流 (reflow) 和重绘 (repaint)，效率并不高。Facebook 最有颠覆性的一点就是提出了一个虚拟 DOM 的概念，把整个 DOM 放在内存当中，然后通过高效的 diff 算法来计算比较哪些 UI 组件需要更新，最终只对这些需要更新的组件进行真实操作。经过测试，采用 RN 框架，无论是加载性能还是页面滑动性的用户体验上，都比原来 H5 的方式要好很多。

最后，社区活跃。除了 Facebook 之外，GitHub 上有很多第三方的团队、个人、公司开发贡献了很多非常优秀的第三方组件，它的社区是非常健康、非常活跃的。

3. React Native 的局限

不过现实是残酷的，即便确定了用 RN 框架做业务开发，在实际的开发当中也发现了 RN 的一些不足。对我们的业务来讲，最不能接受的主要是以下四个方面。

第一点，RN 框架原生并不支持 Web 端。这意味着如果一个业务需要同时上 Android、iOS 和 H5 页面的话，那除了用 RN 之外，还需要用传统的 H5 或用 ReactJS 框架再做一次开发，这样效率是非常低的。

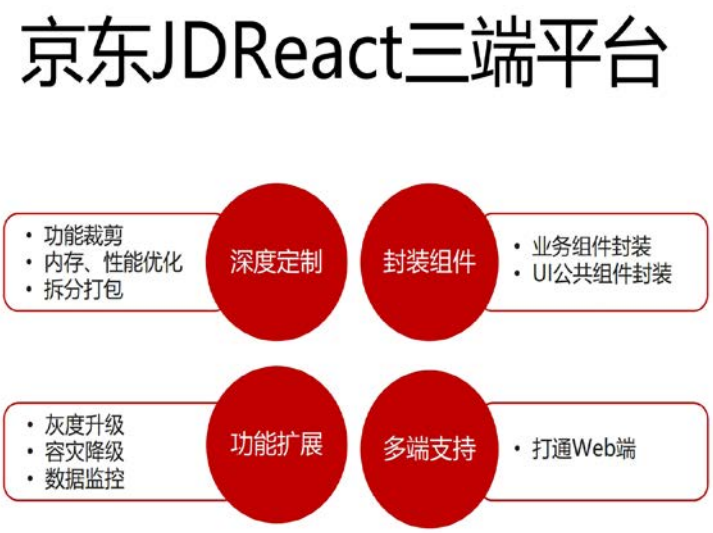
第二点，RN 框架官方并不支持热更新。虽然现在有很多第三方方案，比如微软的 CodePush，但是官方并不原生支持热更新，而热更新对我们

的业务来说也是非常重要。

第三点，Facebook 给出的官方 RN API 不能完全满足业务快速的发展。它只给了一些很基础的 API，但业务中经常会用到的一些多媒体，比如录音、录像、视频播放文件以及文件上传、压缩、加密等等，这些都没有提供。

最后，前面提到 RN 框架性能非常不错，比 H5 好很多。实际上经过真正的业务开发后，发现 90% 的场景下 RN 的性能非常棒，可以满足我们的业务需求；但是在另外的 10% 的场景下，特别是一些交互非常复杂、页面非常复杂、需要频繁的更新、需要一些手势交互的场景，RN 仍有些内存跟性能的瓶颈。

4. 解决方案：JDReact 三端融合平台



既然 RN 有优点也有缺点，那怎么办?

我们的解决方案是 基于 RN 框架进行了深度定制和二次开发，逐步打造了符合京东业务的 JDReact 三端融合平台，主要的工作是以下四大方面：

- 第一，把 RN 的核心 Base 库拿来做裁剪和二次开发，把不需要的功能删减掉，把性能、兼容性、稳定性的问题修复，包括也支持

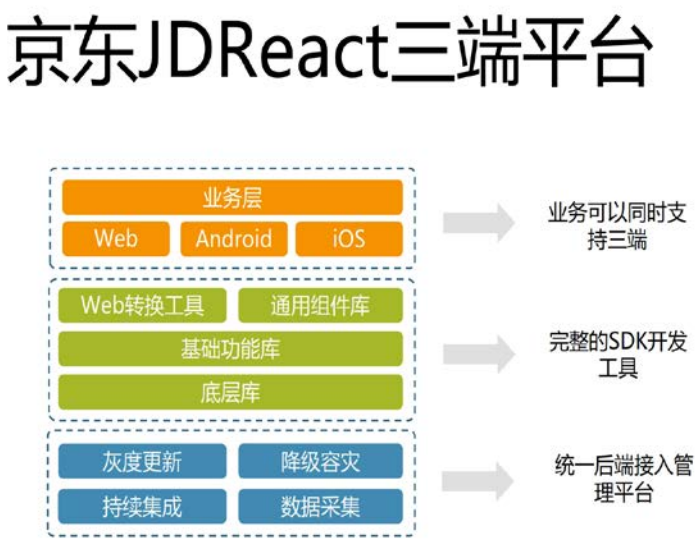
了拆分打包。

- 第二，在后端搭建了一个功能支撑平台，帮 RN 框架增加了灰度更新升级、数据监控以及降级容灾功能，这些对业务来说是非常重要的。
- 第三，基于整个 RN 框架，结合京东的一些业务特点，封装了一套自己的业务组件，包括 UI 公共组件库。 目的是为了让垂直业务开发者可以很快地使用框架进行业务开发，完全不用关心设计的样式跟交互，可以快速接入业务。
- 第四，打通 Web 端，实现了一套 RN 框架向 ReactJS 转换的工具。 可以做到一次代码编写，直接部署到 Android、iOS 跟 Web 三端。

二、JDReact 三端融合平台全解析

1. 整体架构

下图所示就是整个 JDReact 三端融合平台的架构图。



最下面是一个后端接入平台，包含刚刚提到的灰度更新、降级容灾、数据采集和持续集成，这些是由服务端提供的一套服务。中间这一层是提

供给内部开发者的一套完整的 SDK 开发工具，里面除了一些 API 之外，也封装了大量的京东定制功能组件，包括 UI 公共组件。其中还有一块是 Web 转换工具，提供了一套 RN 转换的脚本。业务开发者完全不需要关注这些细节，只要关心他自己的业务逻辑，就可以直接开发出覆盖三端的应用。最上面的业务层就是京东 APP 所有使用三端融合平台开发的业务，这些都可以直接部署到 Android、iOS 和 Web。

2. 改进和优化实践

前面主要介绍了整体的平台架构，现在开始来分享一些干货，就是我们在开发过程当中团队遇到的 RN 的一些问题，包括如何改进跟优化的一些实践。我列了一些功能点跟大家一起分享。

功能裁剪

有同学抱怨过 RN 库太大了，所以拿到 RN 的第一件事就是裁剪。对 Android 平台来讲，除了把 RN 的基础库裁剪以外，很重要一点就是要把方法数减少。因为 Android 平台 dex 有方法数限制，一旦超过 65K 就需要拆分成多个 dex，整个应用的安装跟加载都会有性能问题。所以，要对 Android 方法数进行严格控制，我们的做法就是根据业务情况，把一些用不到的组件方案中的功能组件删除。其中最重要改动就是把 Android 中 support-v7 和 stetho 库依赖给去掉，去掉之后不仅大小减小了很多，而且方法数减少了将近 7000。除了移除这个功能库，很重要一点，因为不是一个全新的 RN 应用，需要跟现有的体量很大的 APP 做集成整合，所以尽量让一些依赖库复用主站中依赖库，比如 fresco、okhttp 等。一来缩减包的大小，二来避免包的冲突。但是主站中的版本很可能跟 RN 中引用的版本有差异，需要中间做一层适配层，把这些差异尽量抹平，保证这些功能和方法都能工作。

加载性能优化

虽然说 RN 框架号称比 H5 的加载性能快很多，但实际开发中发现在 Android 的一些低端机型上，加载速度还是达不到原生体验，极端情况下甚至会出现白屏。主要原因是业务 jsbundle 比较大，RN 框架在加载

jsbundle 和通过 JSCore 解析 jsbundle 时耗时太长。当用户看到真正业务页面之前会出现长时间的空白页面。

当时提出了两个解决方案，第一种方式是实现一套预加载机制。预加载机制就是在用户真正进入业务之前，把 jsbundle 提前加载解析，提前把 RootView 生成。简单来说就是用空间换时间。但这样做并不是所有的业务都适合，因为会带来一些内存增长，所以一般在很核心很重要的业务采取预加载机制。第二种方式是修改了 RN 框架底层库，在 RN 框架开始加载 jsbundle 文件时，显示一个 loading 的进度提示用户正在做加载的动作。当 JS 文件加载并且解析渲染完成之后，把进度条去掉，最终被页面展现给用户。这样虽然等待时间并没有减少，但是用户体验会好很多，整体的时间从收到的反馈来看还是比 H5 要好很多，这是我们做的一个优化点。

内存优化

我们还做了一件很重要的事情，就是内存优化。在 RN 框架开发中碰到的最大的坑就是内存这块，因为业务中会经常碰到 ListView 的使用，根据这些业务的需要，可能要加载很多页，两页、三页、甚至可能会无限加载。这种方式在早期的 RN 版本当中肯定会引起 OOM（OutOfMemory）崩溃，原因是在 RN 的早期版本当中并没有对 ListView 做内存复用。这意味着 ListView 滚多少，图片都会在内存中，当页面加载地越多，出现 OOM 崩溃的几率也越大，这是一个非常不能接受的问题。

在 RN 的早期版本，我们团队在 JS 层实现了一套内存回收。它的原理跟原生当中的原理也差不多，就是当页面划出两个屏幕之后，会强制把图片和内容进行回收，用一个空白的 View 替换。当内容划到用户可见的屏幕范围之后，再把图片给加载出来，这也是原生常用的一种内存回收的方式。修改后的效果很好，无论页面加载再多，都不会出现卡顿和 OOM 崩溃。在 RN 的新版本（0.43 之后），引入了一个新的 FlatList 组件。这个组件完全解决了 ListView 的内存回收问题。它的实现机制和我们的方案类似也是在 JS 层中做内存回收的动作。所以给大家建议，如果开发中碰到类似的问题，完全可以升级到最新的 RN Base 0.43 以上使用 FlatList 组

件。如果版本比较低的话，那就需要自己实现这套机制。

第二个比较大的内存问题就是图片，iOS 平台可能相对好一些，在 Android 问题会相对多一些。RN 的底层图片框架库用的是 Fresco，而我们主 App 中用的也是 Fresco 底层库，这里就会有些问题。第一个就是重复初始化，这也是当时业务开发当中碰到的问题。当主 App 中的 Fresco 进行初始化之后，如果 RN 中也进行一次初始化，实际上之前那部分内存并没有被释放，会出现内存泄漏。我们做了专门的检测，避免 RN 重复初始化的问题。第二个也是跟 RN 框架里面的实现有关系，因为它采用的图片编解码用的是 ARGB_8888，这种方式支持 Alpha 通道。但实际上大部分情况下可以采用 RGB_565 编码，虽然丢失了 Alpha 通道，但是图片在内存当中的大小可以减少 50%。不过有些业务可能也真的需要一些透明的背景，需要 Alpha 通道，所以也提供了一些 API 来针对特殊图片，让它采用 ARGB_8888 进行编解码。这样既解决内存问题，也满足了业务的需求。

最后一个经验就是在所有的 RN 页面退出之前，建议强制调用 Fresco 框架的 clearMemoryCache 方法，通知 Fresco 清除内存缓存。可以保证 GC 及时地把这些图片内存给回收掉，避免整个 APP 的内存占用过高，经过实践验证这也很有效地解决了内存问题。

拆分打包

关于拆分包，因为目前我们采取的方式是每一个业务打成一个 jsbundle 文件，这意味着业务越多，jsbundle 文件会越大。而这些 jsbundle 文件当中，业务的代码其实占比很小。百分之七八十都是 Facebook 提供的一些公共组件库。我们的做法是在编译打包之前，把这些公共组件库先抽取出来，放在一个 common jsbundle 里面，然后业务只保留业务相关的一些 jsbundle 文件。最终在真正的加载之前，做一个简单的合并动作，这样业务越多，这种优化的效果就越好，可以有效减缓 jsbundle 文件大小的增长速度。

性能优化

除了内存之外，最关心的就是性能。前面也提了 RN 的性能其实比

H5 要好很多，可以满足我们 90% 的场景，但实际上还有 10% 的场景，RN 做的并不是很好。主要也是因为整个 RN 的机制，它虽然是最终渲染成原生的组件，但是 UI 的控制还是在 JS 中做的。受限于 JS 单线程一些限制，当有一些很复杂的交互、很复杂的手势或者快速的滑动，很有可能引起 JS 中的阻塞，造成动画的一些渲染的数据不能及时同步到原生当中，造成了整个页面的卡顿。

建议的方案有三种，第一个做 RN 的 Base 升级，把 RN 升级到最新的 0.45，它会采用了一个新的叫 Yoga 的引擎。这种引擎是完全用 native 实现的，可以把大部分的动画渲染和交互放在原生的线程中做。经过测试，采用了 Yoga 引擎，整体的渲染性能可以提升 30% 以上。

第二种方式，有一些非常复杂的一些交互，比如左右滑动结合上下滑动一些手势，如果用单纯用 RN 做，很容易碰到一些手势冲突的问题。所以把这种组件原生化，完全用原生实现，所有的交互跟手势控制全在原生做。这样做就可以达到非常完美的性能，但同时也需要原生开发团队介入。

最后一个经验就是尽量使用 Animated 这种动画类，减少 JS 控制的 UI 数据同步，避免 JS 线程阻塞。

版本检测

另外在 jsbundle 文件当中增加了一个 version 文件，解决版本冲突检测。因为要支持线上更新，就意味着需要把每一个业务 jsbundle 文件做一套完善的版本控制。需要知道当前这个 jsbundle 文件的版本号是多少，可以跑在哪个客户端的版本当中，可以支持的这它的 RN 底层库是多少。这些信息都会记录下来，然后在每一次的升级之前做版本检测。这可以有效地避免线上不同客户端和不同 RN 版本之间的版本冲突问题，可以支持线上的灰度升级。

兼容检测

RN 其实有最低版本支持，像它的早期版本在 Android 是支持 API 16 以上，iOS 是 iOS7 以上。其实我们的主 APP 要支持的版本会比他更低一些，所以需要在主 APP 中做一些保护和判断，一旦检测到用户的版本不支持

RN，就需要做一些降级处理，比如说把入口关闭或者跳转 M 页。这样最大的程度避免不支持 RN 版本的用户出现崩溃的情况。RN 其实可以支持 x86 芯片，但是考虑到如果要支持 x86 的话，需要增加一套基于 x86 的 so 文件，会对包大小有影响，所以对所有的 x86 做了降级。

原生能力扩展

前面刚才也提到了我们的业务非常多样，很多的能力 RN 并不支持。所以基于 RN 框架我们扩展很多业务上用到的原生组件，比如做了整个多媒体的视频播放、视频录制、音频播放、音频录制等组件，还有一些文件上传、语音识别等。在 RN 提供了这套 JS 的接口，给垂直业务团队快速开发和使用。

3. 通用组件库封装

我们也结合自己的业务做了一套通用组件库的封装，例如京东当中的用户登录、购物车、收银台等等业务，在 RN 中做了一套组件的封装。把所有的接口都提供了 JS 的 API，样式和交互像常用的下拉刷新、对话框、按钮等等，也提供了一套通用的样式组件给开发者。在做业务开发的时候，完全不需要关心这些样式怎么画、颜色怎么搭配，只需要关注业务逻辑。剩下的事情由框架做，这可以提升整个业务开发的效率。

4. 三端融合

刚才前面提到了很重要的一项工作就是克服了 RN 不支持 Web 端的问题。我们做了一套 Web 转换的工具，打通了三端。其实在业内三端融合也有广泛的研究，方案主要有三种。

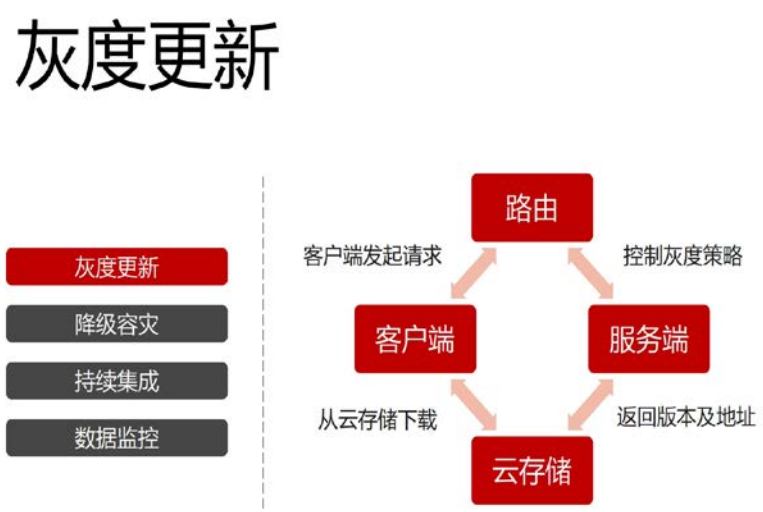
第一种方式，就是在 RN 跟 ReactJS 之上再封装一套轻量的跨平台的抽象层，像微软发布的 ReactXP 就类似于这样的架构。使用这种架构，意味着所有 API、类、组件都不能用 RN API，必须要用新的定义的接口，而且目前 API 支持也不是太多，还在完善中，所以没有采用这种方式。

第二种就是 ReactJS 做开发，之后通过工具转换成 RN，这种方案适合于比较偏重 H5 业务的一些团队，因为他优先需要上的是 H5 页面，用户体验比较偏重 H5。通过工具向 RN 转换其实是个有损转换，因为 RN 支持的样式实际比 CSS 样式少。从 ReactJS 向 RN 转换的话，可能会丢掉一些属性和布局。

第三种方案就是先用 RN 做开发，开发完之后再通过 WebPack 工具向 ReactJS 进行转换。 这种方式的好处是可以优先保证 RN 中的体验，而且 RN 的样式支持是 CSS 的一个子集，这意味着从 RN 向 ReactJS 转换不会丢失功能和属性，所以业内更多的方案也是采用这种方式。GitHub 上

有一些类似的开源框架。但它们支持的组件并不是太全，不能完全覆盖我们的业务，所以我们自己实现了一套。包括之前说的所有的原生组件，它只有原生部分，我们也增加了 JS 部分的实现，使我们的框架可以完整、功能完全没有丢失地转化为 Web 页面。

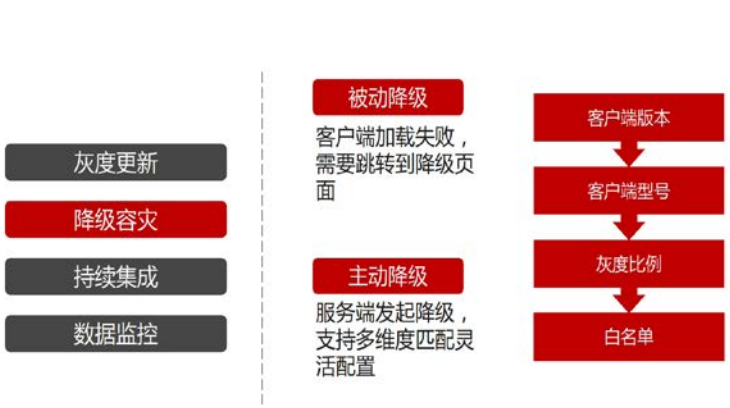
5. 灰度更新



下面简单介绍我们后端的接入平台在服务端增加的灰度更新控制。发版之后，如果需要做一些 RN 组件更新，可以通过后台的更新服务器做一次支持这种灰度的更新。它大概的流程就是首先由 APP 端发起更新的请求，发送到路由控制，路由控制负责控制用户是不是在灰度比例范围之内。如果符合灰度策略，把这个请求转到服务端处理，服务端根据客户端上报的 jsbundle 文件的版本跟服务端部署的版本做一次比较，看有没有适合这个业务的新版本。如果有，把这个升级的版本号以及下发地址回传给客户端，客户端会直接根据下发的下载地址从云存储上下载升级包，完成整个升级过程。因为用户量比较大，所以每一次更新一定要有一个灰度策略，根据灰度比例逐渐放到全网，这是非常重要的。

6. 降级容灾

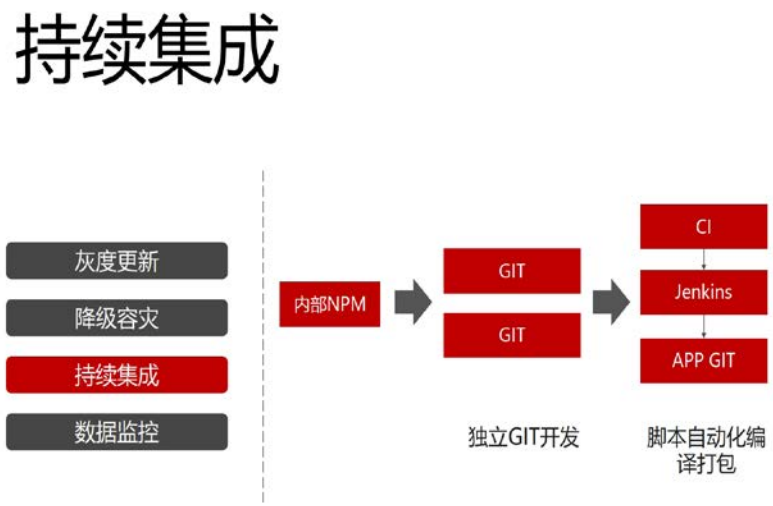
我们把降级容灾定义为两种：一种叫被动降级，一种叫主动降级。所



谓的被动降级是指客户端确实不支持 RN 框架，每次加载 RN 框架都会出现问题，那必须要进行被动的降级，跳转到对应的 H5 页面，使得对业务的影响降到最低。这种降级逻辑是在客户端当中做处理的，就是前面介绍的兼容性检测。第二种是主动降级，很可能在业务开发的时候会发现一些上游的接口出现了问题，导致客户端中的某项业务不能正确地运行，这个时候就需要由服务端控制对这项业务进行精准的降级。我们会支持多个维度灵活的配置，可以根据客户端的版本号、客户端的型号，配置灰度比例、白名单，精准地对某些用户的某些业务或者某些地区的某些用户进行降级，减少业务上的损失。在一些非常大的促销的时候，像双 11、618 这种峰值非常高的时候，可能会采用这种方式。

7. 持续集成

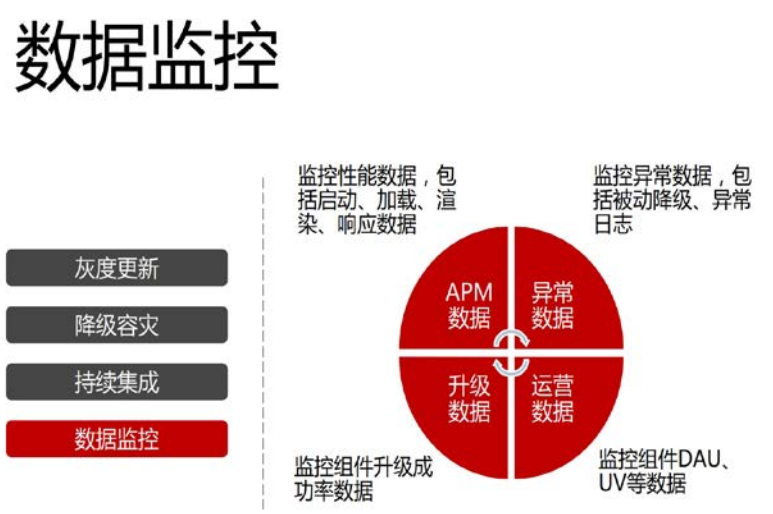
这主要是我们内部的一个开发模型，把所有 RN 的基础库，包括自己提供的一些公共组件库、公共 UI 组件库都部署在内部的 NPM Server 上。每一个接入的业务开发者、每个业务都会有一个独立的 GIT。因为在我们内部，其实业务开发团队可能会很多，我们的团队是负责维护框架，而业务开发团队各个部门各个地区都会有，他们会有申请自己独立的 GIT，然后从 NPM Server 上下载最新的 SDK 包进行业务开发。业务开发完成调试



之后，会通过 CI 打包平台发起打包命令，然后触发 Jenkins 当中的 job，从对应的业务的规则拉取代码，进行编译打包。编译打包成功之后，把它部署到对应的 Android 或者 iOS 客户端版本当中进行整个发布。这种方式对业务开发者来说，最大的好处就是打包编译完全是脚本自动化，不需要获取客户端的源码就可以做到这个业务的开发和上线。

8. 数据监控

JDReact 三端融合平台我们也做了一套非常完善的数据监控中心。因



为需要知道所有 RN 页面启动的时间、加载页面的时间、服务端返回的响应时间、界面渲染的时间。需要把这些 APM 数据上报，通过上报的数据分析，不断地优化性能。第二，也会把业务开发当中碰到的一些异常日志进行上报，可以帮助我们快速的定位问题，发现问题并部署相应的升级包。第三，因为灰度升级更新这个机制，需要有数据埋点来统计升级的成功率。有多少的用户真的是发布升级之后可以成功升级到这个版本。最后，也会针对 DAU、UV、PV 等基础数据做统计，这个主要也是帮业务方搜集一些运营数据，好做业务上的决策。

三、总结

这个框架推出有一年多的时间，到目前为止，京东 APP 当中已经有 20 多个业务正在使用这套框架，其中也有一些比较重要和常用的业务。我们整个平台也经历了去年的双 11、618，今年即将到来的 618，我们也会做更多的后台保障，在稳定性，包括降级上做一些处理，确保业务能够正常地推进。未来也希望能够不断的完善这个平台，不光是京东内部在用，一些很好的组件框架也可以开放出来，跟大家一起学习进步。

六年历程步步为营，京东商城的安全保卫战

作者　李学庆

电商网站在为广大用户提供网购便利的同时，在安全方面也不可以掉以轻心。那么作为一家高流量的电商，京东是怎样做安全防护的？在 618 备战期间又需要特别注意哪些事项？京东安全的现状和未来是怎样的？为此，InfoQ 采访了京东安全方向第一人李学庆，采访内容如下。

InfoQ：电商安全有几个层面？（分类如数据安全、账号安全、业务诈骗安全）

李学庆：电商安全从大类可分为业务安全、应用安全、系统安全、网络安全、数据安全、办公安全。

- 业务安全：风控安全、帐号安全、支付安全、交易安全；
- 应用安全：网站安全、组件安全、框架安全；
- 系统安全：帐号安全、补丁管理、安全加固；
- 网络安全：DDoS 攻击、DNS 攻击、链路劫持、DNS 劫持、异常行为流量感知；
- 数据安全：数据存储、数据传输、数据控制；
- 办公安全：准入（网络安全）、授权（帐号安全）、内网（系统安全）、设备（终端安全）、流程（安全治理）。

InfoQ：电商网站需要注意哪些安全隐患？（后台被攻击？ DDos, DNS 挟持等等）

李学庆：从电商角度来看，首先要确保外部业务不会出现重大安全漏洞，例如基础漏洞、管理后台、弱口令、逻辑漏洞、配置不当等等，对于外部的安全风险做到可控、可快速清除。展开来说，外部业务要建立一套完整的扫描机制，这个扫描机制不是我们传统的上一套扫描器就可以的，需要我们在基础扫描引擎上进行扩展，增加端口、后台监控、banner 监控、页面监控、水平权限监控、爆破探测、应用配置监控、重大漏洞监控等等，然后通过一定的依赖关系进行串联，形成一套完整的外部监控体系。

其次的安全隐患就是通过各渠道导致的数据泄漏问题，如果第一步做到足够坚固，那由于应用层在外部导致的泄漏风险基本可以降到最低，其他的风险源可以通过数据的走向梳理数据链，在关键节点做加固、监控、和审计。最后需要关注的是网络流量问题，也就是 DDOS 攻击。对于抗 D 来说京东通过不同层级分解的方案应对不同类型和量级的流量攻击，并初步具备流量的溯源能力。当然我们也在把 DDOS 的演练工作慢慢推向例行，把漏洞导致的 DOS 和模拟外部攻击的流量对上线的不同业务进行攻击测试，也就是后面我们会让系统上线就具备抗 D 的能力。

InfoQ：具体到京东的 618 时期有哪些尤其凸显的安全挑战？

李学庆：对于今年 618 我们早在 4 月就开始启动，今年安全团队也将有近 50 人规模的安全保障，跨部门对接上百人，我们联合相关业务板块

共同保障京东商城金融、一号店的 PC 端、移动端等所有平台的安全。信息安全部今年落地了 10 个方向的安全预案，对于重大的安全风险提前进行了安全演练。

InfoQ：能否比较详细地谈谈京东的“安全决策蜂窝模型”？

李学庆：京东安全决策蜂窝模型是为了帮助管理者决策公司安全方向的模型。每个行业的安全方向都是不一样的，所以我们要有个能够参考的模型做标准。

- 战略：CXO 从年度的战略方向中需要分析出由于安全风险可能导致的问题，是否需要扩展新的安全技术投入和安全人才的储备。
- 趋势：从公司的发展方向联合公司相关业务部门共同定义出易出现安全隐患的范围，以及行业内对资产的最新技术和最新漏洞，需要提前从技术角度做好调整。
- 影响：针对出现的安全风险需要快速定义是否为核心业务，内外部分界，漏洞级别。多个纬度进行判断影响，从而确定决策。
- 特征：特征阶段可以通过历史风险数据去判断新风险属于重发还是频发，是属于严重还是属于不严重典型类。
- 业务：对现有业务进行清晰分级，辅助影响和特征
- 形象：出现安全风险后需要启动公关、内控、党委以及公共事务相关进行不同层面的决策
- 价值：所有安全风险价值进行直观的分析，用价值的形式进行展现。

InfoQ：能否回顾下京东这六年来走过的路。

李学庆：到现在为止已经 6 年头，一步一个扎实的脚印，一步一个坑的踩了过来。

可以简单的把京东这六年概括为：

萌芽期（2011 年）

大环境下各家公司对于安全的理解和认识还是一个萌芽阶段，京东在 11 年以前已经在开发的每个环节增加安全的检查点，例如 code review。

但整体来说属于初级，大风险能够覆盖。并在我刚去的 2 个月做了一次大型的安全培训，培训场次 60 余场，培训人次 2000 余人，当时最有趣的就是培训完了大家回去都去把电脑密码设置更加复杂，但等到第二天有人打电话过来：李老师，我睡了一晚上觉，电脑密码忘了。到现在，设置安全密码并定时更换已经成为日常工作中大家最基本的安全常识。

起步期（2012 年）

2012 年开始组建京东的安全部门，当时叫做安全管理部。当年做的最多的就是怎么让大家把安全因素放到开发中并形成流程，所以当年做的最多的就是规范研发体系的开发流程、开发规范、以及重大漏洞的设计方案。做的最成功的可能就是自己定义了一套应对 Struts2 漏洞的防御方案，并全部嵌入到了上线流程中。

成长期（2013 年）

2013 年最受行业关注的莫过于撞库问题，由于前两年行业中出现的各种数据泄漏，用户通常的习惯又是在多个网站注册相同的用户名和密码，所以导致很多电商网站出现诈骗、批量刷券的问题。我们通过努力，将撞库风险降到最低。并于在 2013 年推出了京东 JSRC，联手行业白帽子一起，帮助京东查缺补漏，直至现在 JSRC 为京东作出了很大的贡献。

发展期（2014 年）

通过 3 年的积累，我们对于基础的安全风险已经可以做到可控，14 年我们主要聚焦到怎么保障业务上的安全，很多业务上的安全风险通过不同检测方法、逻辑判断、智能的识别相对完整的做了体系化。

对标期（2015 年）

对于公司业务发展迅速、上线系统繁多，我们开始考虑到把现有的很多能力通过平台化的形式做好管理，提高效率。所以后边慢慢延伸出来不同纬度的安全管理平台。

扩张期（2016 年）

2016 年是一个京东安全团队一个扩张期，职责变得更大，随着京东业务体量的不断增长，安全团队的责任也越来越大，已从几十人的小团队

到现在数百人的团队。

创新期（2017 年）

今年京东信息安全部聚焦的业务和技术范围更加广阔。我们开始针对移动痛点的自研解决方案，希望可以开放给行业；不断提升威胁感知能力；攻防团队慢慢升级为红蓝对抗；IoT 安全方向研究；公有云的安全赋能等等。

InfoQ：京东应急安全响应中心的工作职责和具体的工作内容包括？

李学庆：京东安全响应中心的工作职责是为京东与白帽子之间搭建一个以安全为中心的沟通桥梁。白帽子可以通过挖掘京东的漏洞、情报、扫描插件、0day 提交至京东安全响应中心（http://security.jd.com），京东安全团队会根据漏洞级别发放等同价值的积分，白帽子可以使用积分兑换想要的商品。

自京东安全响应中心开张以来在行业中是首家创办安全小课堂、开创系列诈骗宣传活动、首家启用白帽子为大促保驾护航、京东安全公益以及首家联合行业 SRC 共同倡议白帽子懂法、守法单位。

InfoQ：能否给我们讲讲现在京东的前沿安全趋势？

李学庆： 对于传统安全厂商来说，京东做安全相关的产品具备的优势就是场景。我们也在从这个角度看是否可以做些更具有价值的落地产品。

脉象平台：脉象平台是基于京东的资产平台（大海）进行的升级改造。大海平台目前已具有京东重要资产信息，并可随时获取最新的资产数据。在前段时间的 struts2 漏洞大海系统起到了至关重要的作用，快速定位风险范围，对升级效果的快速检验。整体下来比行业修复漏洞至少快了 10 个小时。针对现有的漏洞定位已经解决的，但对于数据泄漏的溯源、定位还是个很难的问题，特别采用数据关系链的思路把数据泄漏问题进行溯源定位。

京东脉象平台分成三层：基础数据层、关系链层、查询接口层。

- 基础数据层包括京东的基础资产、资产的安全漏洞、资产的威胁情报；

- 关系链层把所有的资产做关联，从基础资产类、漏洞类、情报类、框架类、用户信息类、订单信息类，每一类都会把相关资产进行关联，并把路径展现清晰；
- 查询接口层通过不同维度既可以定位漏洞影响范围，同样输入泄漏的用户数据或订单数据。

脉象平台会把相关数据的使用部门、存储方式、存储服务器资产、是否出现过安全漏洞、对外服务器是否出现过入侵痕迹，使用人的基本信息（是否为新员工），之后从脉象中定位具体范围。

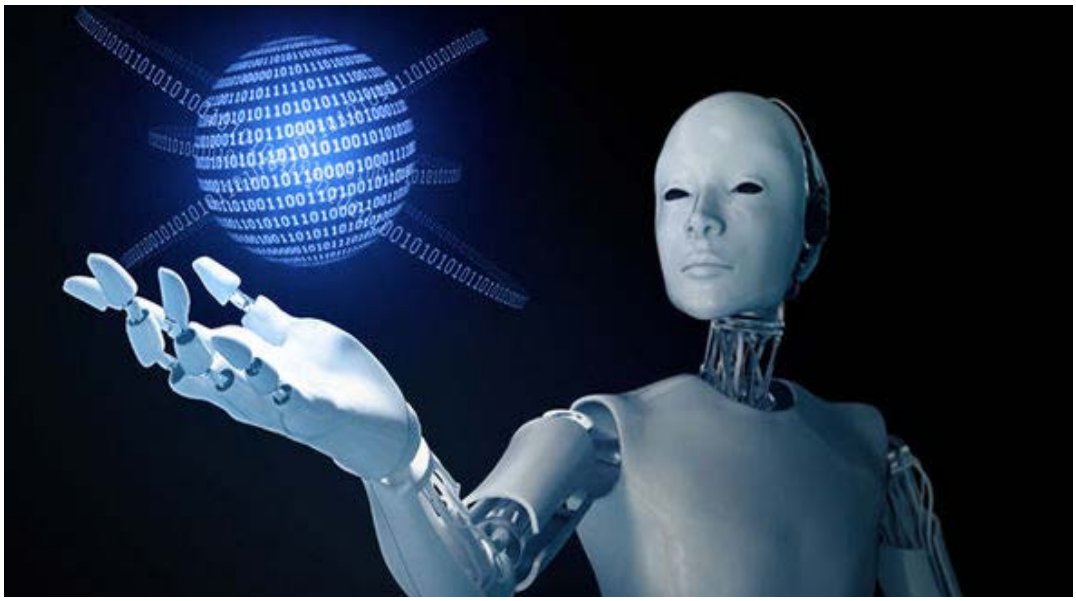
最后喊句口号：技术引领，正道成功！所有的付出都将成为个人生涯中的一个重要里程碑！

受访嘉宾介绍

李学庆，京东安全方向第一人，618、双 11 安全保障总舵手，安全领域中 SELC 发起者。2011 年加入京东商城，担任公司安全攻防、安全响应以及安全体系规划建设工作，京东安全响应中心建设及运营等。曾参与京东涉及的所有行业重大漏洞响应、京东相关的安全事件等。2016 年带领安全团队保障京东安全，间接避免京东损失高达 4.3 亿元。曾根据多年安全行业经验总结并分享“安全决策蜂窝模型”、“信息安全体系建设三部曲”、“安全行业人才培养计划”等内容。

智能机器人 JIMI 的进击之路

作者　刘丹

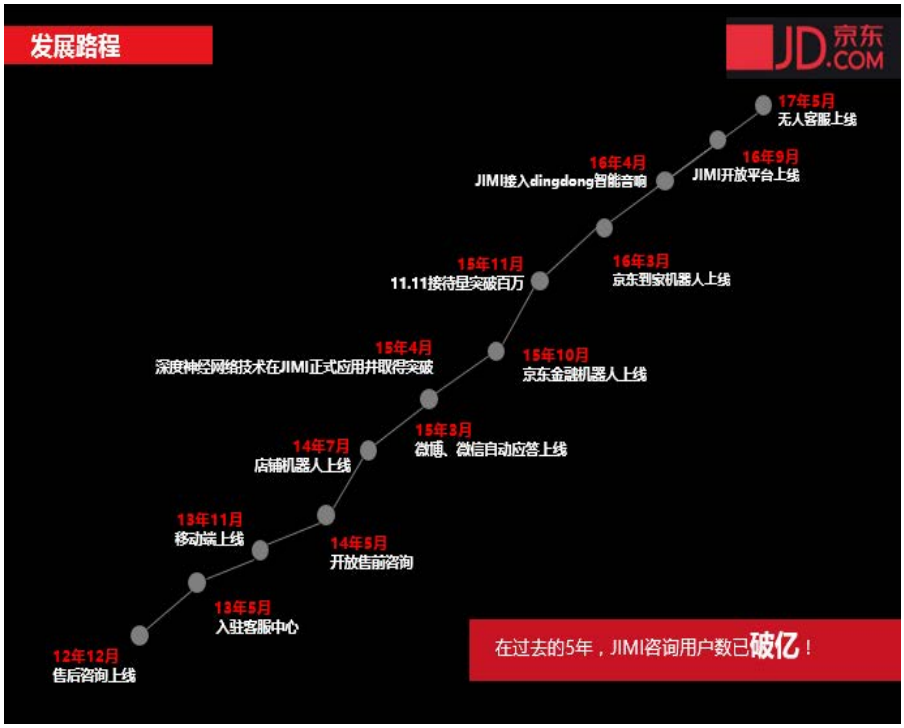


目前，人工智能正在以前所未有的姿态汹涌而来，快速杀入人们的视野。京东一直致力于用技术驱动业务成长，全面提高用户体验，基于对未来客服人力成本可能的提升，以及人工智能技术的发展趋势，早在 2012 年，京东就决定研制智能机器人以应对业务不断拓展带来的客服成本和压力。

JIMI 的发展：智能进化，全面拓展

2012 年 JIMI 顺势诞生，初期以售后服务为主，14 年 5 月开放售前服务，逐渐拓展到移动端、微博、微信等多平台端口，为用户提供推荐商品、告

知优惠、砍价、下单、直接支付的售前全流程闭环体验，让用户可以边咨询边购物，成为用户贴心的购物助手。同时，我们也将智能机器人拓展到各个业务层面，店铺JIMI、京东金融JIMI、京东到家JIMI相继诞生，此外，我们还将JIMI的服务能力平台化，推出了JIMI开放平台，接入长虹、华西等外部企业。

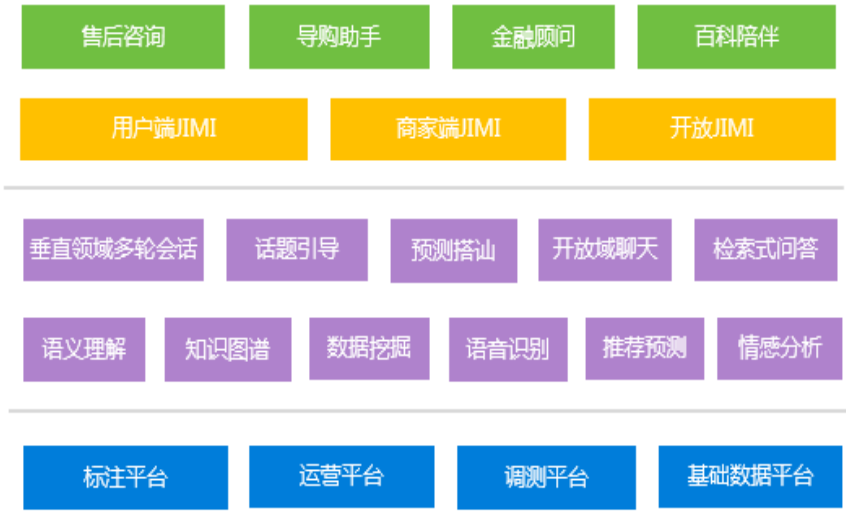


在这全面应用和不断推广的过程中，JIMI也为京东商城节约了数千万的人力成本。日接待量高达百万次，覆盖京东10亿+的商品，应答准确率90%以上，用户满意度高达80%以上，减少客服压力的同时为用户提供更好的服务，也帮助外部企业也减少了至少50%的人力成本。

JIMI产品技术体系揭秘

JIMI整体产品架构如下图所示。

目前JIMI推出的覆盖全平台的用户端产品，商家和企业一旦启用，也就将JIMI的能力赋能给商家和企业，让商家和企业可以定制自己的智能机器人。JIMI在多领域多终端以拟人化的交互体验为用户进行服务，



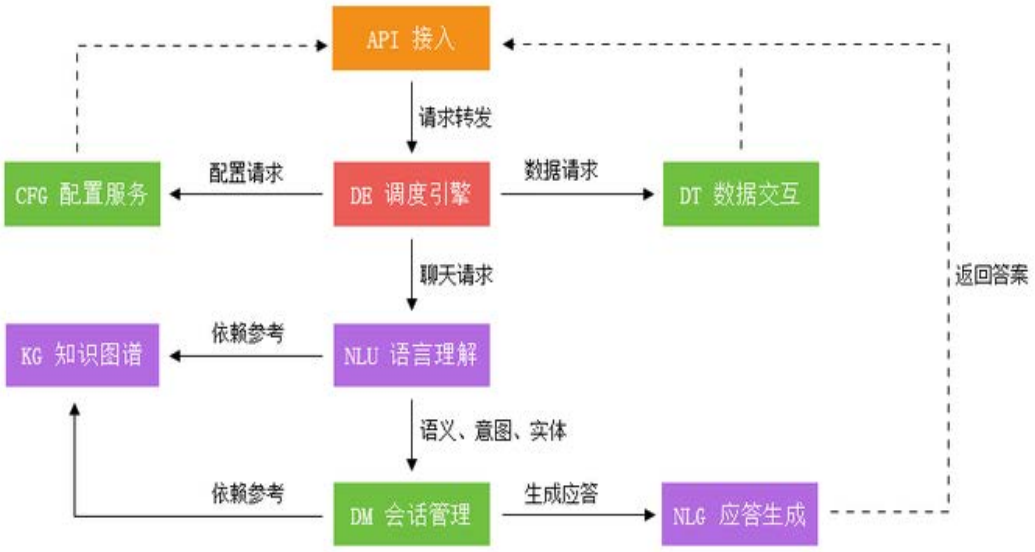
多领域语义识别、情感分析和领域知识图谱等能力是 JIMI 的核心。基础平台支撑起 JIMI 服务能力，让 JIMI 可以快速高效进行数据标注、清洗、挖掘，以及模型训练等。

为了更灵活高效的支撑 JIMI 产品发展路线，在技术上 JIMI 采用了平台服务化架构技术体系，如下图：

从上到下，顶部展示的是目前 JIMI 支撑的所有业务场景，包括：京东自营业务的导购、售后和虚拟业务，京东 POP 店铺机器人，京东金融机器人等。其下，是统一接入层。界面按照终端接入组件化思路，所有界面交互统一把交互和展示逻辑封装，按照 Web（PC）、SDK、M 页面分三类。前后端数据交互通道，浏览器走 HTTPs 和 WebSocket 方式，移动 SDK 走 TCP 方式，若是服务端接口，走京东内部标准化 JSF RPC 方式。这样就整合了所有来自不同终端不同业务场景机器人请求，所有请求统一转发到“调度引擎”服务进行请求调度分发。

调度引擎相当于一个请求路由服务，根据终端机器人请求类型的不同，会调度分发给后端不同服务流程处理。后端的服务整体分为“算法”、“工程”、“数据”三类。对于聊天机器人来说，“算法”是大脑、“工程”是躯体、而“数据”是血液。下面，分别简单介绍下三类服务的作用与协作方式。

工程是系统的躯体，它负责了在线问答请求各服务的内部流转，如下图所示的服务交互流程：



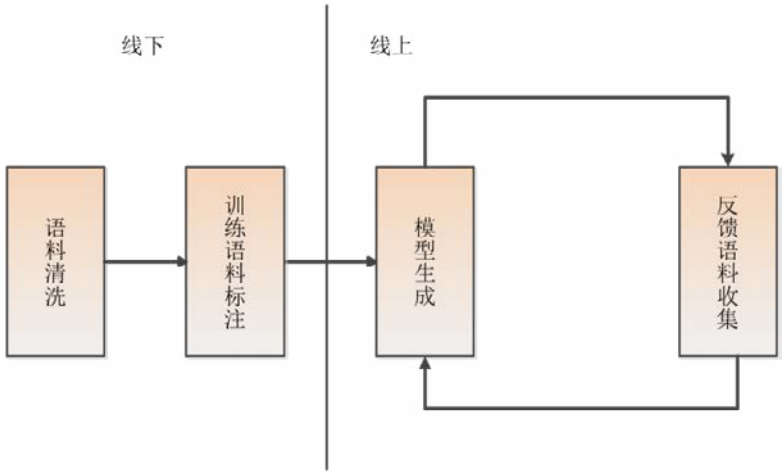
算法是系统的大脑，它的核心是自然语言处理（NLP）。NLP 负责对用户的问题进行分析并产生答案，包括：用户的意图识别，关键词的识别、歧义分析等。对于问答机器人而言，语言理解的关键在于用户意图识别。当用户存在明确意图时，结合相应商品等信息就能给出准确的答案。在实

际应用中，意图识别往往看作机器学习的多分类问题。

为了让JIMI更精准地理解用户的提问，从而给出针对性更强的回答，2015年，京东成立DNN实验室（深度神经网络实验室），将NLP和DNN进行结合，这种新算法具有一定的上下文识别能力，相对于传统的分类算法，会更准确。

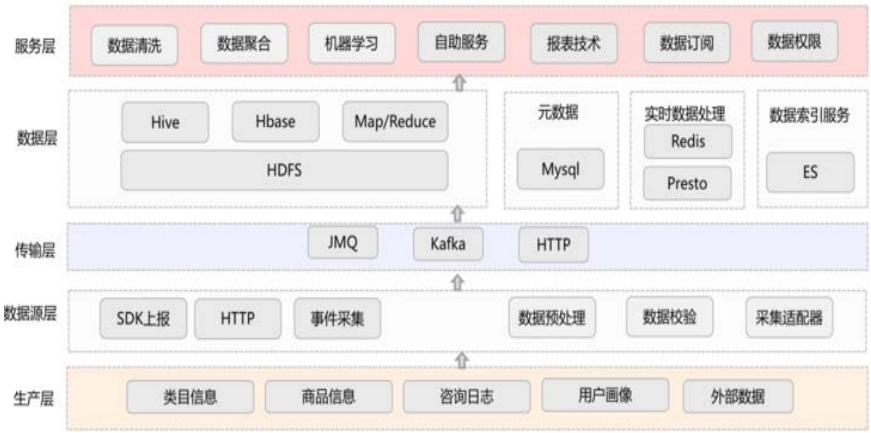
我们只要将京东客服能够回答的问题归纳成一个个的业务点：如退换货相关、运费相关、商品属性相关等。意图识别需要做的就是对用户问题进行分类，决定用户意图到底是在哪一个业务点上。即分类模型输入为用户问题，输出为当前咨询所属业务点。在引入深度神经网络模型后，JIMI意图识别整体准确率由原先的76% 提升至84.1%。在命名实体识别（关键词识别）环境上，结合深度神经网络，JIMI抽取用户问题关键信息的准确率比传统方法提高了6.6%。这也是JIMI与其它同类产品的本质区别。

意图识别模块除了分类外，后续的维护工作也相当重要。一方面需要优化当前的分类效果，解决线上的 bad case；另一方面随着时间迁移，会出现新的业务点，用户提问方式也会发生改变。我们的分类模型也需要随之做出调整。为了减少人力，做到模型自学习，还需要引入线上数据收集模块，整个意图识别框架如下：



这样线下和线上的工作联动运作，就能保证JIMI的意图识别不断优化，跟得上业务的不断升级。

数据是系统的血液，其架构体系如下：



数据经过采集（采集方式包括：SDK、HTTP 和事件采集）、预处理、数据格式校验，通过传输通道（JMQ、Kafka 消息队列异步传输、HTTP 同步传输）进入数据存储层。通过建立的元数据进行实时或者离线加工处理，并建立索引服务，供查询搜索使用。对于清洗后的数据，进行数据聚合，供机器学习语料训练；并提供数据自助查询，报表展现。能支撑 10 亿级数据实时上报，做到秒级延迟、秒级查询，提升了 JIMI 的应答效果。

用户画像也是 JIMI 重要的技术手段之一，目前我们主要致力于挖掘用户与用户、用户与商品之间的深层次联系，通过深度学习、大数据处理等核心技术，建立起复杂的用户、商品特征关联网络，用于精准的刻画用户特性。我们把每个用户的特征，都加入到模型的训练和预测中，比如根据画像数据，当前用户是有小孩的，且历史有过购买奶粉的记录，那我们预测时就会结合用户画像特征，推测用户可能会问奶粉购买相关的问题，并直接将问题展示出来供客户点选，用户点击感兴趣的问题进行咨询即可。

JIMI 的未来：全能的智能 AI 与开放的平台

前不久，AlphaGo 对战柯洁获全胜的新闻引起了大众的广泛讨论，对于 JIMI 是否会像 AlphaGo 一样在某些方面进行训练，以达到人类水平或超过人类这个问题，我们已经可以说，JIMI 在一些品类的客服满意度已超过人工客服。目前在日常情况下，JIMI 的接待量已与人工客服接待量持平，大促期间甚至会超越人工客服。

未来 JIMI 也会继续不断进化，除客服行业，也会积极拓展其它领域的深度学习，提升服务质量，推动人工智能技术成长。同时，我们也会通过我们的人工智能开放平台，给不同行业的商家或机构提供智能咨询服务解决方案，让智能 JIMI 的身影能够进入各个垂直领域，实现京东技术能力的开放与经验共享。

作者介绍

刘丹，京东智能通讯部总监，京东深度神经网络实验室（DNN-Lab）核心成员之一，资深电商专家。在实时通讯、测试架构、稳定性框架、智能电商服务等领域均有涉猎，精通业务通用 性架构、用户行为、智能客服、稳定性提升等多项理论及实践，京东咚咚、智能聊天机器人和无人客服系统研发负责人。

升级全链路压测方案，打造军演机器人 ForceBot

作者　张克房

准备电商大促就要准备好应对高流量，全链路压测无疑是必不可少的一个环节，但是同时也涉及到很繁重繁琐的工作。京东研发设计了军演机器人，这为今年备战 618 减负不少。最初，军演机器人 ForceBot 正式立项并组建了一个虚拟的研发团队，彼时计划是基于开源的 nGrinder 项目进行二次开发，随后实现部署即可；随后在深入研发之后，又根据京东的业务场景对 nGrinder 进行了优化，以满足功能需求。

传统的压测方案

传统的系统压测基本都是部署在内网环境，和被压测的系统部署

在一个局域网内，比较常用的工具有：loadrunner、jmeter、nGrinder、gatling、iperf 等等，通过这些工具，模拟生产环境中的真实业务操作，对系统进行压力负载测试，同时监控被压测的系统负载、性能指标等不同压力情况下的表现，并找出潜在的性能优化点和瓶颈，目前流行的压测工具，工作原理基本都是一致的，在压力机端通过多线程或者多进程模拟虚拟用户数并发请求，对服务端进行施压。以往在京东，备战 618 大促要提前 3 个月准备，需要建立独立系统进行线上的压力评测，这为各个性能压测团队带来了很大的工作量。

除了工作量大的问题之外，传统的压测数据与线上对比，并不准确。以前对某个系统性能的压测需要在内网线下进行多次，压测出的结果各项指标与线上相比差异太大；这是因为线下服务器配置和上下游服务质量不可能与线上一模一样，只能作为一个参考，不能视作线上系统的真实表现。压测后需要思考如何进行各系统容量规划，每次大促前备战会进行基础服务资源的分配，如果不能精确预估容量需求，会发生扩容行为的浪费。

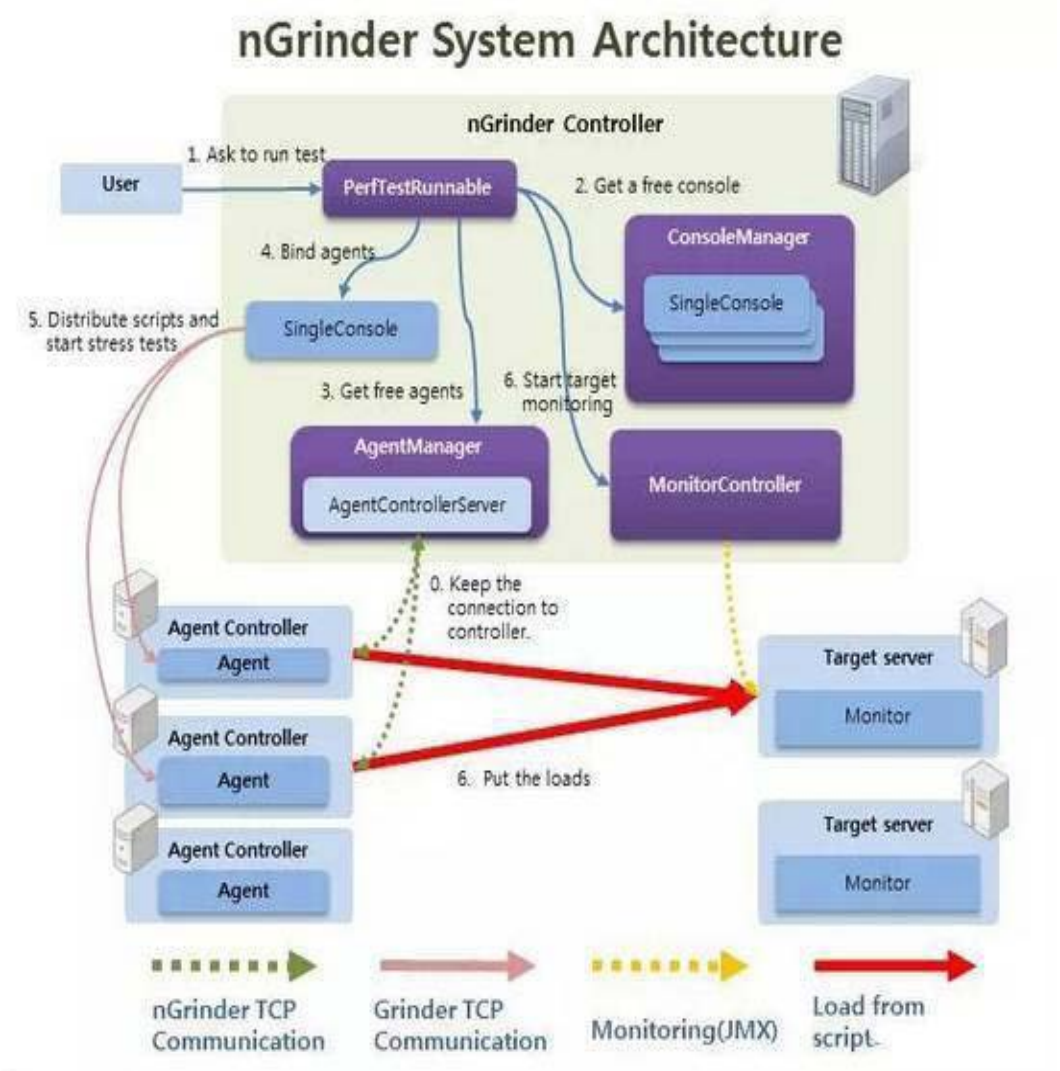
为了更贴近线上真实结果，有些系统也会通过直接内网压测线上系统，这样做需要上下游同步协调，费事费力，这样下来至少一周时间才能搞定。使用了 ForceBot 全链路军演压测系统后，目前只需要 2 天左右就可以完成所有黄金链路系统的性能评测。

变革，制造一个军演机器人

最开始的时候京东并没有直接投入研发力量，京东 618 年中全民购物节技术执行总指挥刘海锋首先提出了 ForceBot 这个想法，京东内部则面向各研发和性能团队进行大量的调研，了解他们现在的痛点和使用的工具情况，同时谈及了 ForceBot 的想法，这个想法获得了大家的赞同并且收集了很多宝贵意见和建议。于是 ForceBot 正式立项并组建了一个虚拟的研发团队，最开始计划是基于开源的 nGrinder 项目进行二次开发，随后实现部署即可。

nGrinder 基于开源的 Java 负载测试框架 grinder 实现，并对其测试引

擎做了功能提升，支持 python 脚本和 groovy 脚本；同时提供了易用的控制台功能，包括脚本管理、测试计划和压测结果的历史记录、定时执行、递增加压等功能。



根据京东的业务场景对 nGrinder 进行了优化，以满足我们的功能需求。比如：提升 Agent 压力，优化 Controller 集群模式，持久化层的改造，管理页面交互提升等。

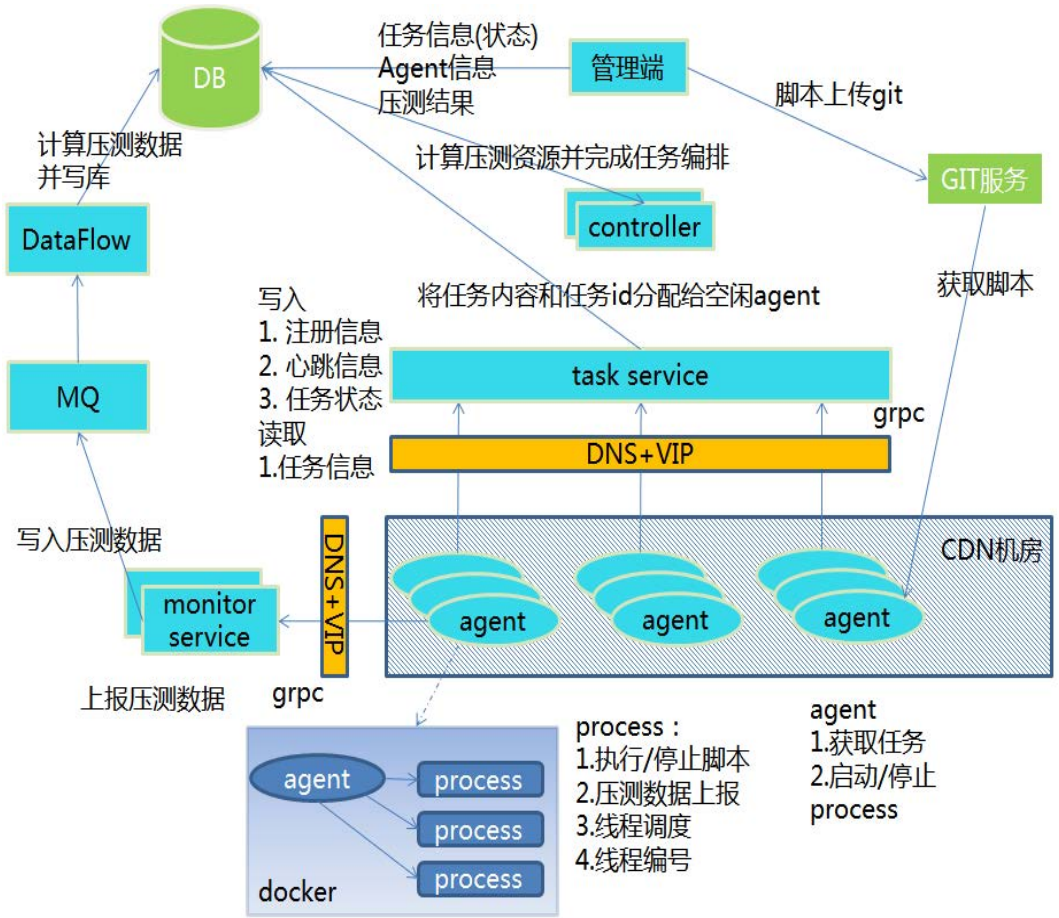
nGrinder 能胜任单业务压测，但很难胜任全链路军演压测。分析其原因是 Controller 功能耦合过重，能管理的 Agent 数目有限。原因如下：

- Controller 与 Agent 通讯是 BIO 模式, 数据传输速度不会很快；

- Controller 是单点，任务下发和压测结果上报都经过 Controller，当 Agent 数量很大时，Controller 就成为瓶颈了。

也就是说问题出现在：Controller 干的活又多又慢，整体压力提升不上去。

尽管我们优化了 Controller 集群模式，可以同时完成多种测试场景。但是，集群之间没有协作，每个 Controller 只能单独完成一个测试场景。即 nGrinder 整体构架无法满足设想的军演规模和场景，也算是走了一些小弯路，最终在其基础上开始新的架构设计和规划，规避已知瓶颈点，着手研发全链路军演压测系统（ForceBot），为未来做好长远打算。



ForceBot

ForceBot 平台在原有功能的基础上，进行了功能模块的解耦，铲除系

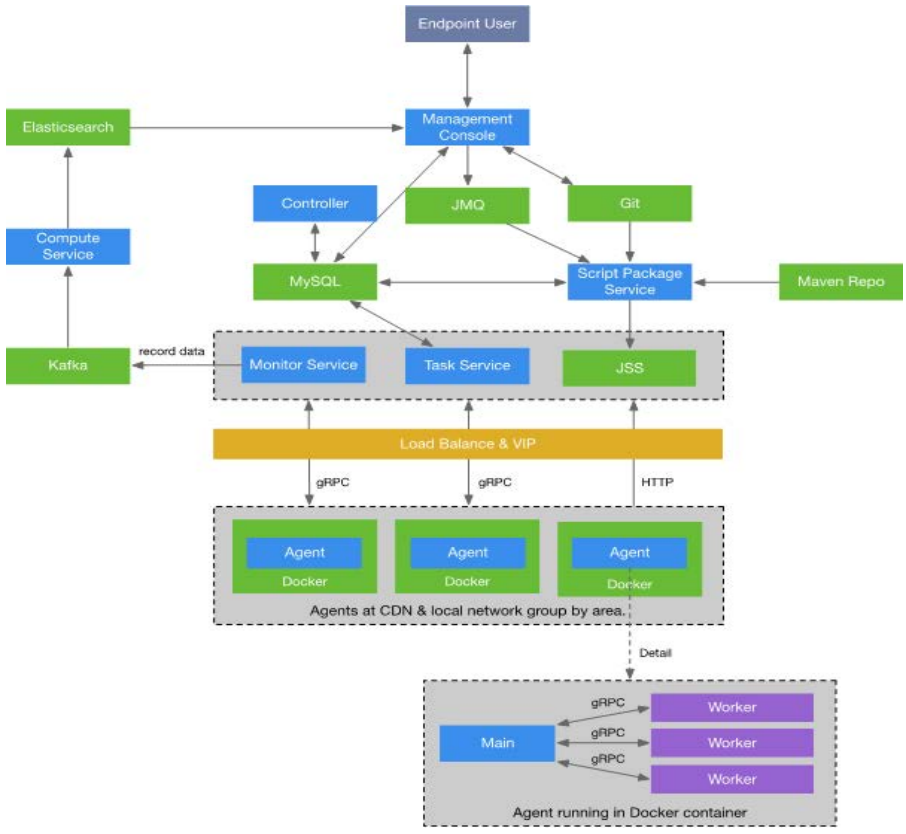
统瓶颈，便于支持横向扩展。

对 Controller 功能进行了拆解，职责变为单一的任务分配；

- 由 Task Service 负责任务下发，支持横向扩展；
- 由 Agent 注册心跳、拉取任务、执行任务；
- 由 Monitor Service 接受并转发压测数据给 Kafka；
- 由 Dataflow 对压测数据做流式计算，将计算结果持久化至 DB 中；
- 由 Git 来保存压测脚本和类库。GIT 支持分布式，增量更新和压缩。

这样极大的减轻了 Controller 的负载压力，并且提升了压测数据的计算能力，还可以获取更多维度的性能指标。

在此基础上，还融合进来了集合点测试场景、参数下发、TPS 性能指标等新特性。



持续改进

ForceBot 平台在上线提供服务后，在受到好评的同时也发现了一些问题。所以我们对架构和功能实现进行了调整。对问题进行了合理化解决，重新设计了架构中的部分功能实现，并对依托 nGrinder 和 Grinder 的功能，针对京东的使用习惯和场景，进行了自研，至此，ForceBot 平台由基于 nGrinder 基础上的深度改造升级为完全自研的性能测试平台。

Git 作为先进的版本控制系统，用来构建测试脚本工程的资源库再合适不过，但是直接使用它作为资源分发服务就不太合适了。Git 的每次操作如 Clone、Pull 等都是一组交互，传输效率不高。同时一代平台使用 GitLab 构建的 Git 服务集群依赖于一个集中的 NFS 网络共享存储，这就带来性能瓶颈和单点故障的可能。

架构调整中针对 ForceBot 平台的资源分发方式进行了重新设计，借助京东基础平台自研的京东分布式文件系统（JFS）的云存储服务（JSS）进行资源的分发。

新的架构调整中，增加了 Script Package Service 为性能测试脚本提供统一的构建打包支持，并通过对 Maven 的支持，与公司内网 Maven 私有服务交互，为脚本提供更灵活可靠的依赖管理。Script Package Service 将脚本及其依赖类库、配置、数据进行打包，处理成一个统一的 Gzip 压缩文件，并上传至 JSS 以向 Agent 进行分发和归档。

平台在使用过程中，由于调试环境和实际运行环境有所差别，用户有查看 Agent 执行日志的需求。考虑到日志落地带来的磁盘 IO 对性能性能的影响，以及日志内容给平台管理带来的开销，新的架构中提升了日志的收集和处理功能。Agent 的日志不再落地本地磁盘，改为写入到内存一块固定大小的区域，最后经处理切割成一条一条的日志实体并结构化的存入 Elasticsearch 中供用户查询。

技术细节

1. 核心功能

平台核心功能在于需要向性能测试人员提供一个高效的可操作环境用于准确的描述其测试逻辑，并兼容大部门公司业务服务调用方式和场景。京东内部业务系统大部分使用 Java 语言开发，使用基础平台中间件技术部开发的 JSF、JMQ 等中间件进行服务调用，为此提供一个与 Java 语言高度兼容的脚本语言执行环境作为性能测试逻辑编写基础尤为关键。

系统选用了兼容 JSR223 规范的 Groovy 作为主要脚本语言，并效仿 Java 下著名的单元测试框架 Junit 的设计哲学设计了一套高效并友好的测试逻辑开发和执行环境。

平台核心脚本引擎为性能测试脚本设计了多种生命周期控制，以适用不同的场景，并使性能最优化。在脚本编写过程中，用户仅需要在 Groovy 脚本中使用内置的几种注解便可对脚本的执行和数据采集进行精确灵活的控制，如测试类生命周期、事物、执行权重等，大大提升脚本开发效率。

2. 容器部署

为了快速的创建测试集群，Agent 采用 Docker 容器通过镜像方式进行自动化部署。这样做好处如下：

- 利用镜像方式，弹性伸缩快捷；
- 利用 Docker 资源隔离，不影响 CDN 服务；
- 运行环境集成，不需要额外配置运行所需类库；
- 每个 Agent 的资源标准化，能启动的虚拟用户数固定，应用不需要再做资源调度。

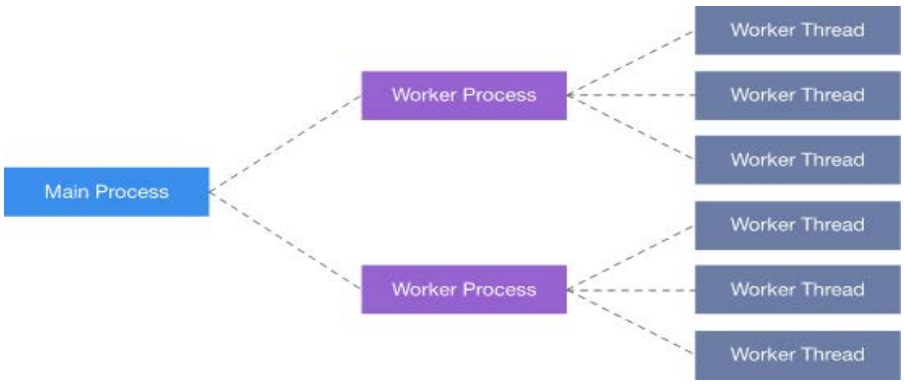
3. 服务通信

Task Service 采用了 gRPC 与 Agent 进行通信，通过接口描述语言生成接口服务。gRPC 是基于 http2 协议，序列化使用的是 protobuf3，并在

除标准单向 RPC 请求调用方式外，提供了双向流式调用，允许在其基础上进一步构建带状态的长链接调用，并允许被调用服务在会话周期内主动向调用者推送数据， 其 java 语言版采用 netty4 作为网络 IO 通讯。使用 gRPC 作为服务框架，主要原因有两点。

- 服务调用有可能会跨网络，可以提供 http2 协议；
- 服务端可以认证加密，在外网环境下，可以保证数据安全。

4. Agent 实现



Agent 采用多进行多线程的结构设计。主进程负责任务的接收、预处理和 Worker 进程的调度。将任务的控制和执行进行进程级别的分离，这样可以为测试的执行提供相对独立且高度灵活的类库环境，使不通的任务之间的类库不会产生冲突，并有益于提升程序运行效率。

Agent 与 Task Service 保持通信，向系统注册自身并获取指令。根据任务需要启动 Worker 进程执行任务，主进程负责管理 Worker 进程的生命周期。Worker 进程启动后会通过 gRPC 与主进程保持通信，获取新的变更指令，如线程数变化通知，及时进行调整。

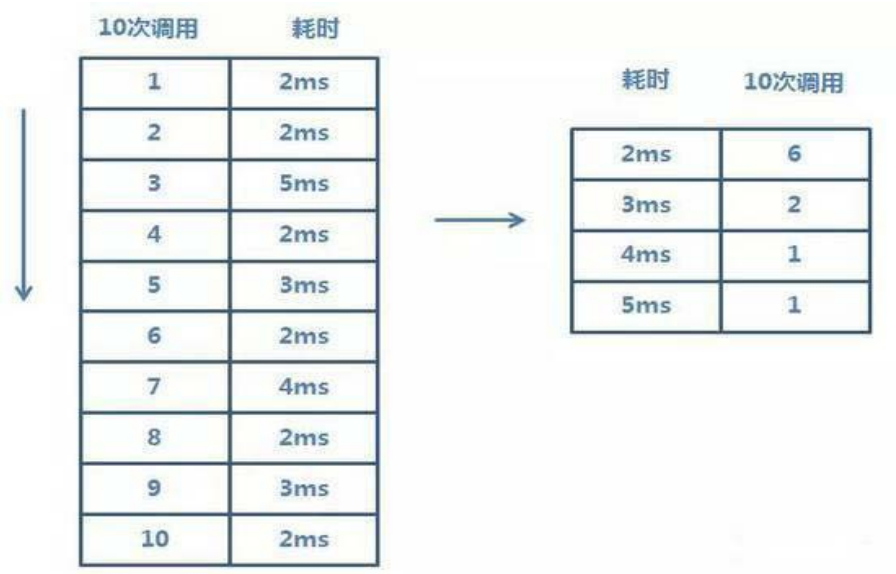
5. 数据收集和计算

实现秒级监控。数据的收集工作由 Monitor Service 完成，也是采用 GRPC 作为服务框架。Agent 每秒上报一次数据，包括性能，JVM 等值。

Monitor Service 将数据经 Kafka 发送给 Compute Service，进行数据的流式计算后，产生 TPS，包括 TP999，TP99，TP90，TP50，MAX，MIN

等指标存储到 ES 中进行查询展示。

为了计算整体的 TPS，需要每个 Agent 把每次调用的性能数据上报，会产生大量的数据，Agent 对每秒的性能数据进行了必要的合并，组提交到监控服务以进行更有效有的传输。



新式全链路压测的打开方式

ForceBot 的工作基本原理就是站在真实用户角度出发，从公网发起流量请求，模拟数百万级用户 0 点并发抢购和浏览，由于压力机分布在全国各地，不同的运营商，所以最接近真实用户的网络环境，对于读服务通过回放线上日志形式模拟用户请求，数据更分散、更真实，避免热点数据存在，对于写服务则模拟用户加入购物车、结算、更改收货地址、使用优惠券、结算、支付等核心链路环节，每次军演都会发现新的问题和瓶颈，对后期的资源扩容、性能调优都会有针对性的备战方向，最终让研发兄弟对自己系统放心，让消费者购物无忧。

全链路压测方案最重要的部分是被压系统 如何去适应这种压测，如何精准的识别测试流量，如何协调整个研发系统统一打标透传压测流量，并做相应的处理，尤其对脏数据的隔离和清理尤其关键，被压测的系统要

和线上环境保持一致，又要隔离这些数据，需要确保万无一失，尤其订单部分环节，出现差错是不可恢复的。

压测数据读的服务都是来自于线上服务器的日志真实数据，数据最终会由各个系统通过标记进入回收站，最终被清理。军演平台本身不会侵入到系统去清理垃圾数据，每次军演压测首先会有一个目标值，比如按照历史峰值的一个倍数去动态加压，如到了这个目标值，各系统性能表现都非常好，那么继续加压，直至有系统出现瓶颈为止。

军演机器人的过去与未来

全链路压测可以理解为网络链路 + 系统链路，网络链路是用户到机房的各个网络路由延迟环境，系统链路是各个系统之间的内部调用关系和强依赖性。 其实去年双 11，京东就采用了 ForcBot，只不过仅仅针对重要链路；因为如果没有核心系统首先参与进来，小系统是搞不起来的，因为小系统强依赖核心系统。

今年备战 618，ForceBot 技术架构没有太大调整。我们工作一部分是平台用户体验的一个优化和性能的优化，让有限的压力机产生更大的压力请求；另外一个是整合被压测的系统监控，实时展示。

京东未来希望 ForceBot 可以实现“人工智能预言”。现在还在逐步的做，

我们希望未来的全链路压测引入 AI 技术，通过人工智能预言各个系统的流量值和资源分配建议，根据线上的系统军演数据预言未来的大促各系统场景，举个简单的例子，在 ForceBot 平台上录入每秒一千万并发订单场景，以现在的系统去承载，各系统是一个什么样的性能指标和瓶颈点在哪？这就是我们思考要做的。

作者介绍

张克房，京东商城资深架构师，2010 年 3 月份加入京东商城，2010 年 -2016 年负责京东核心数据中心 IDC 网络、负载均衡、DNS、自建 CDN、DevOps 等多项运维架构和运维管理负责人，是京东早期运维体系架构变革和发展的直接参与者，奠定了京东运维现行的多项技术架构和运维模式，伴随京东从小到大，从无到有、从有到优的飞速变化。

版权声明

InfoQ 中文站出品

出版：北京极客邦科技有限公司
北京市朝阳区洛娃大厦 C 座 1607
欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。
网 址：www.infoq.com.cn