

全球架构师峰会[特刊]

揭秘 双11 背后的技术较量

The Technology Combat
Behind Nov 11th Online Shopping Festival

InfoQ



扫描关注InfoQ官方微信

CONTENTS

目录



揭秘双十一背后的技术较量

本期主编 郭蕾

流程编辑 丁晓昀

发行人 霍泰稳

联系我们

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

1号店 11.11 / 04

从应用架构落地点谈高可用高并发高性能

天猫 11.11 / 14

手机淘宝 521 性能优化项目揭秘

京东 11.11 / 25

商品搜索系统架构设计

当当 11.11 / 32

促销系统与交易系统的重构实践

苏宁 11.11 / 41

系统拆分的一些经验谈

蘑菇街 11.11 / 45

私有云平台的Docker应用实践

唯品会 11.11 / 50

峰值系统应对实践

蚂蚁金服 11.11 / 57

支付宝和蚂蚁花呗的技术架构及实践





InfoQ

www.infoq.com/cn



软件 正在改变世界！

InfoQ 促进软件开发领域知识与创新的传播

软件正在改变世界！InfoQ是一个在线新闻/社区网站，旨在通过促进软件开发领域知识与创新的传播，为软件开发者提供帮助。为达到这个目的，InfoQ基于实践者驱动的社区模式建立平台，提供新闻、文章、视频演讲和采访等资讯服务，所有的这一切也都是为了研发团队中那些有创新精神的人群：团队领导者、架构师、项目经理、工程总监和高级软件开发人员等。

InfoQ目前在全球有四种语言版本，分别是英文、中文、日文、葡文。

另外，InfoQ在伦敦、北京、东京、纽约、圣保罗、上海、杭州等城市举办过QCon全球软件开发大会。

InfoQ中文站将和InfoQ全球网站一样，秉承“扎根社区、服务社区、引领社区”的经营理念，与中国技术社区的专家一起，为中国软件企业和个人提供及时、高质量的技术资讯，成为连接中国企业软件技术高端社区与国际主流技术社区的桥梁。



ArchSummit全球架构师峰会

International Architect Summit

全球架构师峰会（International Architect Summit，简称ArchSummit）是InfoQ关于架构垂直领域的技术峰会，邀请处于企业IT架构设计核心领域的架构师，探讨不同行业、不同领域的架构发展演变，探讨架构师们在设计和实现时的心得体会，也探讨架构师的自身成长和职业发展。每次活动我们都邀请国内外的知名技术专家来分享交流，为国内的技术人员搭建一个信息沟通的桥梁。

郭蕾

InfoQ主编，负责InfoQ网站内容的输入和输出，关注容器、开源等技术领域。我狂妄，我自负，我信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。



卷首语

双十一可以说是各个电商平台一年中最繁忙的一天，也是他们系统压力最大的一天。在高频、高额、高密度的交易场景下，如何能为用户提供稳定而流畅的购物流程，成为了各个电商一年中的工作重点，而双十一，正是对这一年工作的一次检验。去年，我们报道了京东、天猫、蘑菇街三个电商平台的双十一促销活动，其中蘑菇街坦言自己考试没及格，而天猫的表现最为抢眼，已经突破了之前的交易记录，创下历史新高。今年双十一，苏宁向京东开战，号称平京战役，而阿里更是搞出大动作，与湖南卫视联袂在水立方上演了一场精彩的双十一晚会。双十一，俨然已经成为电商的火拼战场。

在这样的购物狂欢节中，消费者不仅关心商品的质量、送货速度，更多地会关心平台是否有流畅的购物体验，而这也正是电商技术的关键所在。纵观国内的电商公司，技术已经成为其绝对的核心竞争力。以阿里巴巴为例，在过去七年的双十一活动

中，交易额、交易峰值、支付峰值这三项指标连年增高，对比2009年和2015年，订单创建峰值增长了350倍。在这持续的增长背后，毫无疑问需要强有力的技术支撑和保障。正如阿里云总裁胡晓明所说，他们用中国的计算力量支撑起了商业史上的又一个新纪录。

这几年，InfoQ一直从技术视角报道双十一背后的技术力量，试图向读者揭秘电商的技术架构，以让更多的后来者从中受益。2015年的双十一，我们报道了阿里巴巴、京东、1号店、苏宁易购、蘑菇街、唯品会、当当网等电商的关键技术架构和实践经验，并在ArchSummit全球架构师峰会上策划了双十一技术专题，期望通过线上和线下的方式全方位解读双十一背后的技术较量，并促进国内的技术交流，推动技术创新。

参与过几次阿里巴巴双十一狂欢节的报道，在现场发现国内几百家的媒体当中，专注做技术报道的媒体不超过5家。究其原因，我觉得有几点，一是技术对于业务的重要性这几年才开始逐步被重视，二是技术报道门槛太高，就内容生产来说难度比较大。在技术日益重要的今天，很庆幸我们能成为为数不多的技术媒体中的一员，因为少，因为难，因为不容易，所以我们要更专业，更努力。

推动技术的创新和发展，我们一直在路上。

1号店11.11：从应用架构落地点谈高可用高并发高性能



作者 张立刚

1. 背景

1.1 三高是电商核心交易系统的基础

电商核心交易系统有很多特点，如分布式、高可扩展等，在众多特性中，高可用、高并发、高性能是基础。大到技术峰会、论坛、研讨会，小到一场面试，高可用、高并发、高性能始终是焦点，是技术大牛、技术追随者永远津津乐道的话题，成为他们毕生的追求。

另，ArchSummit全球架构师峰会北京站将于2015年12月18日~19日在北京国际会议中心召开，大会设置了《揭秘双十一背后的技术较量》专题来深入解读双十一背后的技术故事，欢迎关注。

1.2 三高首先依赖的是架构

日常和同行、同事的交流过程中，大家经常讨论的问题就是，你们是如何做到高可用的？访问峰值达到了什么级别，系统怎么撑住的？高并发下怎么保证数据一致性？性能如何提升？采用了什么新技术？

尽管大家的答案各有不同，从硬件到软件、从程序到SQL、从静态到动态、从C到JAVA，但大家最终总能达成一致，高可用、高并发、高性能依靠的不是某个硬件、某种技术、某种DB，而是好的架构。

1.3 能落地的架构才是好架构

好架构很多，网上随便一搜，微软、Google、阿里、京东等众多大牛的架构图很多，都是好架构。

当然今天我们不是来谈什么是好架构，因为我们从不缺架构图。架构图是形，怎么落地是神；就如军用材料，技术大家都懂，工艺才是关键。所以，能落地的架构才是好架构，当然我们更缺的是好架构的落地点。

2.1号店如何做三高

1号店技术部从1个人做起到现在千人级别的规模，系统支持每天亿级的访问量、单Service支持每天亿级的请求、订单支持每分钟几万单级别、Service服务可用性达到99.9999%，架构上也经历了历次演进，

今天我们就从应用架构历次演进的落地点谈起。

1号店应用架构的演进大致经历了以下历程：
强依赖→ Service化→业务解耦→读写分离→异步→水平/垂直拆分→服务逻辑分组等。

当然这个过程从不是串行的，永远是并行的，并且这个过程永远是在1号店这辆系统大巴行进过程中进行的，因为我们不能停车也不能刹车，而且还必须不断提速。

2.1 应用架构的最大演进 - SOA治理

早期的1号店系统，遵循简单的MVC架构，Controller层处理了所有的业务逻辑包括与DB的交互，在系统初期这种Simple的架构方便快捷，成本低业务响应快。但当业务开始变得复杂、人员规模爆发式增长，这种强耦合强依赖带来的弊端就成了巨大的瓶颈，代码耦合度高互相冲突、出错概率和事故概率明显提升，业务需求不能快速响应，SOA治理迫在眉睫，解耦和去依赖成为第一需求，于是Service化成为第一前提。

2.2 SOA治理 - Service日志是保障

1. 做Service首先是规划，Service规划的第一步首先考虑什么？大家可以先自行“考虑”下：

- 很多人想的是采用什么RPC框架、采用什么技术，怎么让性能更高；也有人首先想的是业务怎么拆分，怎么才能更合理。
- 我们首先想到的是如何做监控和问题定位。
- 高可用不是一步做到的，我们的Service可用性不是一步达到99.9999%的，在这过程中一定会有很多的问题出现，怎么提前发现这些问题、出现问题后如何快速定位，这才是最重要的。
- 这只能依赖日志，这是监控和问题定位的基础。

2. 下单接口业务性强，其对可用、并发、性能的要求作为技术人你懂的。我们就从这个接口开始下手，但我们没有先去分析业务，首先想的是如何定义日志系统，让以后的监控和问题定位更简单更快捷。事实证明这个决定是对的，直到现在1号店的大部分订单相关的监控、预警、问题排查定位等完全依赖这个日志。



• 编码详解--GOS-10111001

- 第一组3位数字代表Service名，如101代表createOrder
- 第二组2位数字代表错误类型(第一位代表大类，第二位可细分小类)
 - 1-系统级错误**
 - 2-应用级错误(前端参数错误)** ,
 - 3-业务级错误(Service自身处理出错)** ,
 - 4-依赖级错误(Service内部调用第三方服务出错)**
 - 5-交互级业务提醒**(正常业务逻辑，非错误，需告知用户，如库存不足)
 - 99-未知异常**
- 第三组3位数字代表错误明细，如001代表参数值为空

www.yhaodian.com 誓信 顾客 执行 创新

3. 日志系统的设计基于以下：一是进数据库、持久化有序化，二是分类化、层次化、错误code唯一。

进数据库、持久化有序化这个大家都理解，我曾经接手过的一个应用系统，一天下来Tomcat的log文件大小超过1G，会让你崩溃的。

分类化、层次化、特别是错误code唯一这个是关键，它是大海航行中的那盏灯塔，让你可以瞬间定位问题位置，它给监控预警带来的好处同样伟大，可以从各个维度去做监控预警。

4. 1号店现在有了很好的SOA中间件 - Hedwig，可支持百亿级的访问请求，它有SOA级别的日志，也很完善。但应用级别的日志我们还是建议各应用系统自己做，它的业务性、个性化是公共架构永远代替不了的。

2.3 应用架构演进之落地

一定有人要问1号店采用的什么RPC框架，好吧，是Hessian，这不是什么秘密。为什么要用Hessian？我偷偷告诉你，PHP是最伟大的的开发语言。

2.3.1 业务垂直拆分

万事具备，草船已借箭，要从业务角度规划Service了。

我们从产品、用户、订单三个维度上对Service进行了规划，构成1号店应用架构上的三架马车，确立了SOA治理的框架基础。

在此基础上，又陆续衍生出促销、积分、支付等众多Service业务，在三架马车中同样会细分至如

文描、价格、库存、下单、订单查询等垂直服务。

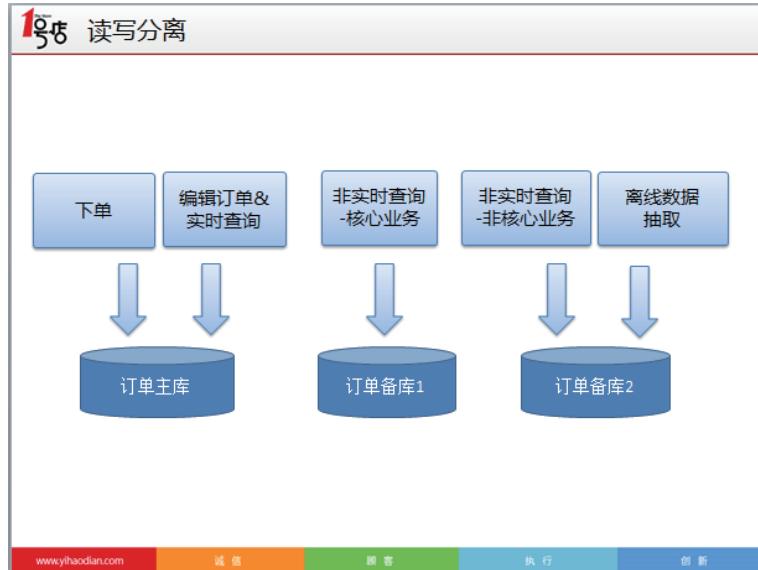
Service化是前提，Service化完成后，后面可以大刀阔斧地干了，因为业务独立了、DB读写权限收回了，哈，好像整个天下都是我的了。但，得天下易治天下难，真正的大戏在后面。



2.3.2 读写分离

读写分离的重要性不需多讲，除了最简单的读写分离，写可以从应用层面继续细分，

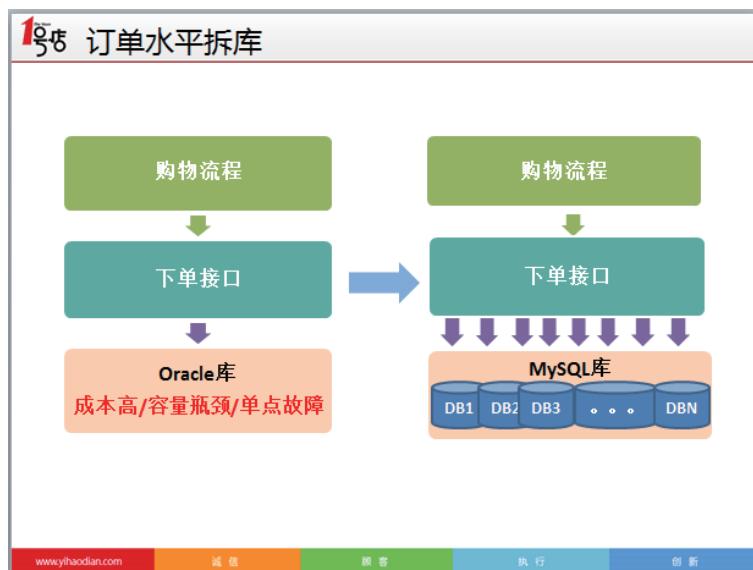
读也可以从应用和DB层面再细分，如订单的读写分离：



2.3.3 水平拆分

早在2013年，1号店就实现了库存的水平拆分，2014年又一举完成订单水平拆库&去Oracle迁Mysql项目。

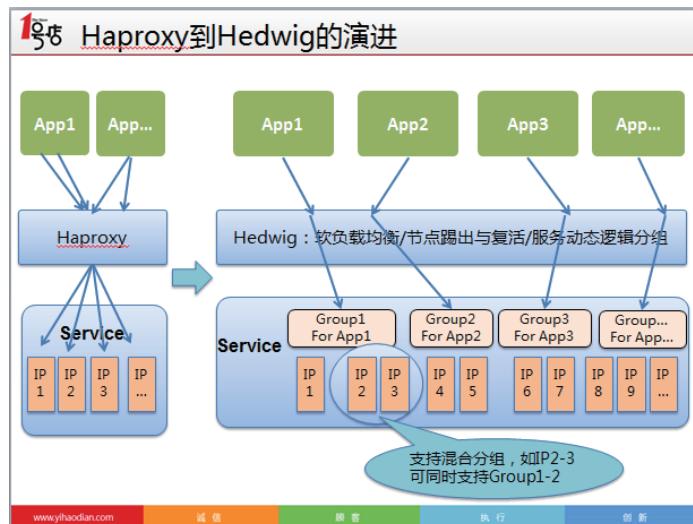
订单水平拆库&去Oracle迁Mysql两个重量级的大项目合并为一个项目并完美无缝上线，在经济上节省的是成本，在技术上体现的1号店的整体技术实力和水平。



2.3.4 服务逻辑分组

前面说到了读写分离，大家也注意到了，这是物理隔离。物理分离好处明显，但其硬件成本、维护成本高的弊端也显而易见。这时候，我们的大杀器—Hedwig又上场了，有兴趣多来了解下我们SOA中间件—Hedwig，这可是获总裁奖的项目。

Hedwig提供了服务自动注册发现、软负载均衡、节点踢出与复活、服务动态逻辑分组、请求自动重试等众多SOA框架所需的强大功能，支持并行请求、灰度发布，其背后提供的调用链路及层次关系、日志分析、监控预警等更是为SOA治理提供了强大的后勤保障。



2.3.5 业务解耦/异步

上面提到的读写分离、水平拆分、逻辑分组等主要是从技术层面保障，也是技术人员最喜欢的话题。业务流程的梳理、优化、改造往往不被重视，但作为应用架构，我们最不能忽视的是业务。

1. 我们的一个核心Service服务，性能从2年前的近200ms到现在的几十ms，从代码上、sql上的优化提升顶多占到10%，更多地优化都在业务流程的梳理改造上。

2. 我们曾经花费两周多的时间将一个系统的整体性能优化提升了近10倍，最后总结下来，纯技术的优化（代码或sql质量导致的性能差）几乎没有。

优化主要在两方面，一是架构上，如使用缓存、单SKU的循环查询改成批量查询等，这个能优化的也并不多，因为我们的架构整体还是比较合理的；最大的优化还是在业务梳理上，很多地方我们使

用了重接口，接口里很多逻辑计算和DB查询，这些逻辑并不是所有的业务场景都需要的，但开发人员为了简单没有将业务场景细分，导致大量不合理的调用，既浪费了服务器资源又严重影响用户体验；同样，很多地方为了一个不重要的展示或逻辑也产生大量不合理的调用，反而影响了核心业务，这些都是最需要优化的，也是优化效果最明显的。

3. 异步本身不是什么高深的技术，关键是哪些业务可以走异步，这更体现架构师的业务理解能力和综合能力。

如下单成功后给用户的消息通知、发票详细信息的生成等都可以异步，我们在这方面做了很多工作，包括和各业务方的大量沟通制定方案等，在不牺牲用户体验又保证业务流程完整的情况下，尽量走异步解耦，这对可用、性能都是极大的提升。

3.追求极致

开放、共享、追求极致是1号店技术人的理念。我们在追求极致上做了很多，简单举几个例子：

3.1 一个下单接口定义了135个错误编码

前面提到过日志和错误编码的定义，大家一定想象不到，我们仅一个下单接口就定义了135个错误编码。接口上线后至今出现的错误编码在50-60个，也就是说有50-60处不合理或错误的地方，这个不合理或错误既有业务的又有程序的也有我们对编码定义的不合理。

出现一个我们就解决一个，系统自然越来越健壮和稳定，目前日常每天下单出现3-5个错误编码，主要为业务性如特价商品已售完无库存等。

那没有出现过的几十个编码是不是就意味着我们白做了？

功夫下对了永远不会浪费，在下单接口上线近2年后，一个之前从未出现过的错误编码跳出来了，是一个很难出现的业务场景，但通过这个编码，我们马上定位了问题，都不用去看代码。

我们永远不能保证系统没有bug，bug可以藏的很深埋的很久，但我们不怕，因为我们的伏兵也一直在，你一跳我们立马抓，毫不犹豫。

3.2 Service服务可用性99.9999%

6个9的Service服务可用性，可能有人不信，看看我们真实的监控邮件，这是每天亿级的调用量。

功夫永远在戏外，结果仅仅是一个结果，一步步踏实走过来的旅程才是我们收获最大的。



3.3 销售库存准确率99.9999%，超卖率为0

做过电商核心系统的人都明白库存控制的难度，库存不准甚至超卖的问题至今还有很多电商公司没有完全解决。

这个问题也曾经困扰我们，为此还专门写了一个库存刷子的程序来刷数据，现在这个程序已基本宣告废弃了，因为我们的库存准确率达到了6个9，超卖率为0。

怎么做到的？业务、业务、业务，重要的事说三遍。

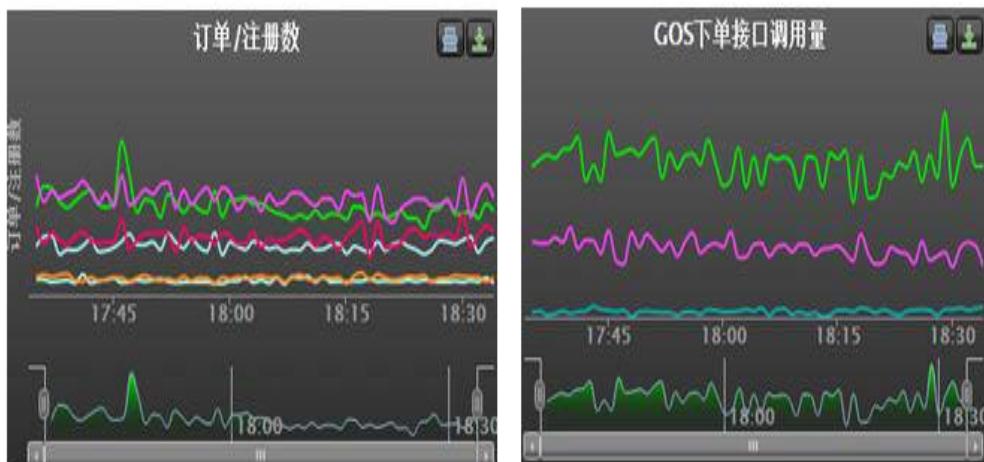
我们团队花了3个多月的时间，对所有库存应用场景逐一排查，最终梳理出16个有问题的库存场

景，并逐一协调解决，让库存形成严格的闭环线路，保证了库存的准确性。在这过程中，对库存闭环方案和对业务的理解成为关键，纯靠技术，我们能做的并不多。

4.应用场景

4.1 业务监控

业务监控首提订单监控，对订单我们从实际订单数据和Service接口调用量两个维度去做监控，保证了监控系统的稳定和准确（监控系统也会出错的），其中下单接口调用量使用的就是Service日志数据。



4.2 服务监控

依赖查询

依赖治理

	ProviderApp	ServiceName	ServiceMethodName	CallerApp	ClientMethodName
1	yihodian/mandy	mandy	MandyService getProvinceProductsInfo	yihodian/front-search	MandyCacheService getProvinceProductsInfo
2	yihodian/mandy	mandy	MandyService search	yihodian/front-search	MandyCacheService search
3	yihodian/mandy	mandy	MandyService getRelatedCategory	yihodian/mobile-servi...	MandyCacheService getRelatedCategory
4	yihodian/mandy	mandy	MandyService searchProducts	inshop/store-service	MandyCacheService searchProducts
5	yihodian/mandy	keyword	KeywordService keywordAutoComplete	yihodian/mobile-servi...	SearchKeywordServiceImpl.getSearchKeyword
6	yihodian/mandv	mandv	MandvService search	yihodian/mobile-servi...	MandvCacheService search

服务监控

集群列表 ServRegCluster | 自动刷新:禁用
更新时间: 2014-10-13 17:49:48 状态: GOOD

Node Ip	Role	连接数	Watched	Watched	Total Path	Send	接收	Config	状态
1 10.4.1.50.2181	follower	296	432670	432670	18200	373297566	360578738	360578738	OK
2 10.4.1.40.2181	follower	294	442592	442592	18200	363465077	352019812	352019812	OK
3 10.4.1.48.2181	follower	261	437678	437678	18200	306035485	296126188	296126188	OK
4 10.4.1.46.2181	leader	677	480536	480536	18200	3412455896	3342804769	3342804769	OK
5 10.4.1.69.2181	follower	634	480400	480400	18200	2380115465	2317909006	2317909006	OK
6 10.4.1.38.2181	follower	641	522311	522311	18200	2135083274	2074901170	2074901170	OK
7 10.4.1.55.2181	follower	650	540821	540821	18200	2286354315	2227794325	2227794325	OK



5.后言

电商核心交易系统的高可用、高并发、高性能不是一朝一夕的，需要好的技术，更需要好的架构，如何找到落地点并一步步踏实落地，这是关键。有想法、有目标、有执行力，必有所成。

我们是技术人，但我们的很多工作并不一定要

多高深的技术，业务和技术的平衡点才是最重要的。业务敏锐性对应用架构师和开发人员来说都尤为重要，因为更多的时候我们要的是解决方案而不是技术方案。

谨以此献给那些在追求高可用、高并发、高性能道路上飞奔的同学们！祝你们早日三高：



天猫11.11：手机淘宝 521 性能优化项目揭秘



作者 手机淘宝技术团队

又是一年双十一，亿万用户都会在这一天打开手机淘宝，高兴地在会场页面不断浏览，面对琳琅满目的商品图片，抢着添加购物车，下单付款。为了让用户更顺畅更方便地实现这一切，做到“如丝般顺滑”，双十一前夕手机淘宝成立了“521”（我爱你）性能优化项目，在日常优化基础之上进行三个方面的专项优化攻关，分别是1) H5页面的一秒法则；2) 启动时间和页面帧率提升20%；3) Android内存占用降低50%。优化过程中遇到的困难，思考后找寻的方案，实施后提取的经验都会在下面详细地介绍给读者。

第一章 一秒法则的实现

“1S法则”是面向Web侧，H5链路上加载性能和体验方向上的一个指标，具体指：1) “强网”(4G/WIFI)下，1秒完全完成页面加载，包括首屏资源，可看亦可用；2) 3G下1秒完成首包的返回；3) 2G下1秒完成建连。

在移动网络环境下，http请求和资源加载与有线网络或者PC时代相比有着本质区别，尤其是在2G/3G网络下，往往一个资源请求建连的时间都会是整个Request-Response流程里面的大头，一

些小资源上拖累效应尤其明显。例如一个1k的图片，即使在10k/s 的极慢网速下，理论上0.1秒可下载完毕，但由于建立连接的巨大消耗，这样一个请求会要耗上好几秒。

仅仅“建连”这一个点，就能说明移动时代的Web侧性能优化和PC时代目标和方式都相去甚远，要求我们必须从更底层，更细致的去抓，才能取得看起来相对有效的结果。

15年初的性能情况：

平均 LoadTime-WIFI	平均 LoadTime - 4G	平均 LoadTime - 2G
3. 35s	3. 84s	14. 34s

可以看到优化前，平均时间很难接近1秒。为了实现优化目标，在技术和实施抓手层面，由底层往上，做了四方面事情：

1. 网络节点：HttpDNS 优化
2. 建连复用：SSL化，SPDY建连高复用
3. 容器层面：离线化和预加载方案
4. 前端组件：请求控制，域名收敛，图片库，前端性能CheckList

网络节点：HttpDNS优化

DNS解析想必大家都知道，在传统PC时代DNS Lookup基本在几十ms内。而我们通过大量的数据采集和真实网络抓包分析（存在DNS解析的请求），DNS的消耗相当可观，2G网络大量5-10s，3G网络平均也要3-5s。

针对这种情况，手淘开发了一套HttpDNS一面向无线端的域名解析服务，与传统走UDP协议的DNS不同，HttpDNS基于HTTP协议。基于HTTP的域名解析，减少域名解析部分的时间并解决DNS劫持的问题。

手淘HttpDNS服务在启动的时候就会对白名单的域名进行域名解析，返回对应服务的最近IP(各运营商)，端口号，协议类型，心跳等信息。

优点

1) 防止域名劫持

传统DNS由Local DNS解析域名，不同运营商的Local DNS有不同的策略，某些Local DNS可能会劫持特定的域名。采用HttpDNS能够绕过Local DNS，避免被劫持；另外，HttpDNS的解析结果包含HMAC校验，也能够防止解析结果被中间网络设备篡改。

2) 更精准的调度

对域名解析而言，尤其是CDN域名，解析得到的IP应该更靠近客户端的地区和运营商，这样才能有更快的网络访问速度。然而，由于运营商策略的多样性，其推送的Local DNS可能和客户端不在同一个地区，这时得到的解析结果可能不是最优的。HttpDNS能够得到客户端的出口网关IP，从而能够更准确地判断客户端的地区和运营商，得到更精准的解析结果。

3) 更小的解析延迟和波动

在2G/3G这种移动网络下，DNS解析的延迟和波动都比较大。就单次解析请求而言，HttpDNS不会比传统的DNS更快，但通过HttpDNS客户端SDK的配合，总体而言，能够显著降低解析延迟和波动。HttpDNS客户端SDK有几个特性：预解析、多域名解析、TTL缓存和异步请求。

4) 额外的域名相关信息

传统DNS的解析结果只有ip，HttpDNS的解析结果采用JSON格式，除了ip外，还支持其它域名相关的信息，比如端口、spdy协议等。利用这些额外的信息，APP可以启用或停止某个功能，甚至利用HttpDNS来做灰度发布，通过HttpDNS控制灰度的比例。

建连复用：SSL化，SPDY建连高复用

出于安全目的，淘宝实现了全站SSL化。本身和H5链路性能优化没有直接的关系，但是从数据层面看，SSL化之后的资源加载耗时都会略优于普通的Http连接。

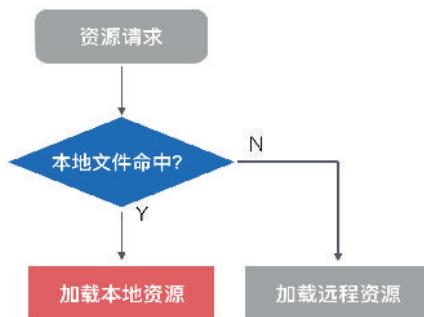
有读者会有疑惑，SSL化之后每个域名首次请求会额外增加一个“SSL握手”的时间，DNS建连也会比http的状态下要长，这是不可避免的，但是为什么一次完整的RequestResponse 流程耗时会比http状态下短呢？

合理的解释是：SSL化之后，SPDY可以默认开启，SPDY协议下的传输效率和建连复用效益将最大化。SPDY协议下，资源并发请求数将不再受浏览器webview的并发请求数量限制，并发100+都是可能的。

同时，在保证了域名收敛之后，同样域名下的资源请求将可以完全复用第一次的DNS建连和SSL握手，所以，仅在第一次消耗的时间完全可以被SPDY后续带来的资源传输效率，并发能力，以及连接复用度带来的收益补回来。甚至理论上，越复杂的页面，资源越多的情况下，SSL化+SPDY之后在性能上带来的收益越大。

容器层面：离线化和预加载方案

收益最明显，实现中遇到困难最多的就是离线化或者说资源预加载的方案。预加载方案是为了在用户访问H5之前，将页面静态资源（HTML / JS/CSS/IMG...）打包预加载到客户端；用户访问H5时，将网络IO拦截并替换为本地文件IO；从而实现H5加载性能的大幅度提升。



手淘实现要比上面的通用示意图复杂：因为Android和iOS安装包已经很大，所以预加载Zip包（以下简称“包”）都是从服务器端下载到客户端；本地需要记录整体包状态，并在合适的时机与服务器通信并交换状态信息。在包发布更新的过程中要注意，本地版本和服务端最新包之间的差量同步，必要的网络判断，WiFi下才下载等。

面对亿级UV，并且在服务器资源很有限的情况下搞定这个流程，需要借助CDN来扛住压力，实际上CDN扛住了约98%的流量。

需要注意的是预加载实际上也是一种缓存，更新比H5稍慢一些，主要受几个因素影响：推送到达率（用户是否在线，用户所在网络质量），总控，服务端策略等，所以需要通过推拉结合的触发策略并优化下载包的体积（增量包）来提升到达率。

除了优化到达率，手淘还做了url解CDN Combo后再映射的优化工作，若 URL 是 Combo URL，那么会对 URL 解 Combo，解析出其中包含的资源。然后尝试从本地读取包含的资源，如果所有资源

都在本地存在，那么将本地文件内容拼装为一份完整文件并返回；否则 URL 直接走线上，不做任何操作。

提升到达率和解CDN Combo再映射，这两个容器侧对于离线化方案的优化对于本次H5链路上整体性能的提升有着至关重要的意义。

前端组件：请求控制，域名收敛，图片库，前端性能CheckList

严格执行性能方面的CheckList，主要有三个点：

1. 图片资源域名全部收敛到gw.alicdn.com；
2. 前端图片库根据强弱网和设备分辨率做适配；
3. 首屏数据合并请求为一个。

在执行中，性能的检查和校验一定要纳入到发布阶段，否则就不是一个合理的流程。性能的工具和校验一定应该是工程化，研发流程里面的一部分，才能够保障性能自动化，低成本，不退化。

通过以上优化方案，H5页面的平均Loadtime在Wifi, 4G下均如期进入1秒，3G和2G也有80%多达成1s法则的目标。

第二章 启动时间和页面帧率20%的提升

很多App都会遇到以下几个常见的性能问题：启动速度慢；界面跳转慢；事件响应慢；滑动和动画卡顿。

手机淘宝也不例外。我们分为两部分来做，第一部分是启动阶段优化，目的解决启动任务繁多，缺乏管控的问题，减少启动和首页响应时间。第二部分是针对各个界面做优化，提升界面跳转时间和滑动帧率，解决卡顿问题。双十一性能优化目标之一就是将启动时间和页面帧率在原有基础上继续优化提升20%，接下来就从这两部分的优化过程来做一一介绍。

一. 启动阶段的优化

手机淘宝作为阿里无线的航母，接入的业务Bundle超过100个，启动初始化任务超过30个，这些任务缺少管控和性能监控。

那么首要任务就是：

建立任务管理机制

所有的初始化任务可以用两个维度来区分：

1) 任务必要性：

有些任务是应用启动所必需的，比如网络、主容器；有些任务则不是必需的，仅仅实现单个业务功能，甚至是为了业务自身体验和性能而考虑在启动阶段提前执行，其合理性值得推敲。

2) 任务独立性：

将应用的架构简单分成基础库、中间件、业务三层，这三层中业务层最为庞大，其初始化任务也最多。对于中间件来说，其初始化可能依赖于另外一个中间件。但对于一个独立的业务模块来说，其初始化任务应该也具有独立性，不存在跟其他业务模块依赖关系。

启动阶段任务管理机制包含了如下几方面的内容：

1) 任务可并行

既然很多初始化任务是独立的，那么并行执行可以提高启动效率。

2) 任务可串行

虽然我们期望所有初始化任务都相互独立，但是在实际中不可避免会存在相互依赖的初始化任务。为了支持这种情况，我们设计任务的异步串行机制，这里主要借鉴了前端的Promise思想实现。

3) 任务可插拔

面对这么多不同优先级的初始化任务，任何一个出现异常都会导致应用不能启动，给稳定性带来严重挑战。因此我们设计了可插拔机制，当某一项初始化任务出现问题时能够跳过该任务，从而不影响整个应用的启动使用。这里我们根据初始化任务的必要性做了区分，只有非必要的初始化任务才会应用可插拔的特性，这也是为了防止出现不执行一

个必要的初始化任务导致应用启动使用出现问题。

4) 任务可配置

在ios上通过plist指定每一项启动任务，其中字段optional表示该项是否是必需的，当之前运行出现crash或者异常时，若值为YES则可以不执行该项。

有了任务管理机制，并引入懒加载的理念，可以持续地合理有效管控启动阶段的各项初始化任务，是大型app必不可少的环节。

检测超时方法，优化主线程

性能优化前，初始化代码都在主线程中执行，为了启动性能已将部分初始化任务放入后台线程或者异步执行。但是随着业务发展和人员变更，还是出现了在主线程中执行很重的初始化任务。为此，在ios实现了一套应用运行时方法耗时检测机制，能够对应用中所有类的方法调用做耗时统计。方便的找到超时的方法调用之后，就可以有针对性的做出修改，或删除或异步化。这种方法调用耗时检测机制同样适用于APP运行过程中，从而找到导致应用卡顿的根本原因，最后做出对应修改。

多线程治理

分析各个模块的线程数量，检查线程池的合理性。通过对掉不必要的线程和线程池，再控制线程池的并发数和优先级。进一步通过框架层的线程池来接管业务方的线程使用，以减少线程太多的问题。

减少IO读写

从自身业务出发，去除若干初始化阶段不必要的文件操作，以及将若干非实时性要求的文件操作延后处理。Android上对于频繁读写数据库和SharedPreference以及文件的模块，通过增加缓存和降低采样率等手段减少对IO的读写。对于SharedPreference进行了专门的优化，减少单个文件的大小，将毫无联系的存储键值分开到不同文件中，并且防止将大数据块存储到SharedPreference中，这样既不利于性能也不利于内存，因为SharedPreference会有额外的一份缓存长期存在。

降级部分功能

例如摇一摇功能，测试发现应用场景不频密，但业务使用了高频率的游戏模式，会耗电及占用主线程时间。对该功能做了降级处理，降低检测频率。同理，对于其他非必须使用但又占据较多资源的模块也都做了适当的降级处理。

热启动时间的缩短

在安卓手机上我们把启动分为两类进行检测和优化：冷启动和热启动。冷启动是程序进程不存在的情况下启动，热启动是指用户将程序切换到后台或者不断按Back键退出程序，实际进程还存在的情况下点击图标运行。

之前安卓手淘在按Back键退出时整个首页Activity销毁了，热启动会经过一个比较长的过程。优化后首页在退出的时候并不销毁Activity，但是会释放图片等主要资源，在下次热启动时就能更快的进入。另外，将手淘欢迎页的界面从其它bundle转移到首页的模块，在进入欢迎页时就开始初始化首页资源，做到更快展示。

在经过一系列的优化后，启动方面已经有了明显的改善，在进入首页的时候不会卡顿，GC次数也减少了一半以上。

二.各个界面的优化

各界面优化我们也是围绕着提高帧率和加快展现而展开的，手淘的几个主链路界面，都是相对比较复杂的，既使用多图，也使用了动态模板的技术。功能越复杂，也越容易产生性能问题，所以常遇到布局复杂、过渡绘制多、Activity主要函数耗时、内容展示慢、界面重新布局（Layout）、GC次数多等问题。

优化GPU的过渡绘制

通过开发者选项的GPU过渡绘制选项检查界面的过渡绘制情况。该优化并不复杂，通过对掉层叠布局中多余的背景设置、图片控件有前景内容的时候不显示背景、界面背景定义到Activity的主题中、减少Drawable的复杂Shape使用等手段就可以

基本消除过渡绘制，减少对GPU和CPU的浪费。

优化层级和布局

层级越多，测量和布局的时间就会相应增加，创建硬件列表的时间也会相应增加。有时我们会嵌套很多布局来实现原本只要简单布局就可以实现的功能，有时还会添加一些测试阶段才会使用的布局。通过删除无用的层级，使用Merge标签或者ViewStub标签来优化整个布局性能。比如一些显示错误界面、加载提示框界面等，不是必须显示的这些布局可以使用ViewStub标签来提升性能。

另外要灵活使用布局，并不是层级越多就会性能越差，有时候1层的RelativeLayout会比3层嵌套的LinearLayout实现的性能更糟糕。

除了灵活使用布局，另外我们还通过提前inflate以及在线程中做一些必要的inflate等来提前初始化布局，减少实际显示时候的耗时。对于一些复杂的布局，我们还会自己做复用池，减少inflate带来的性能损耗，特别是在列表中。

加快界面显示

1. 可以通过TraceView工具找出主线程的耗时操作和其他耗时的线程并作优化。另外减少主线程的GC停顿，因为即使并行GC，也会对heap加锁，如果主线程请求分配内存的话，也会被挂起，所以尽量避免在主线程分配较多对象和较大的对象，特别是在onDraw等函数中，以减少被挂起的时间。另外可以通过去掉ListView，ScrollView等控件的EdgeEffect效果，来减少内存分配和加快控件的创建时间。
2. 利用本地缓存，主要界面缓存上次的数据，并且配合增量的更新和删除，可以做到数据和服务端同步，这样可以直接展示本地数据，不用等到网络返回数据。
3. 减少不必要的数据协议字段，减少名字长度等，并作压缩。还可以通过分页加载数据来加快传输解析时间。因为JSON越大，传输和解析时间也会越久，引发的内存对象分配也会越多。

- 注意线程的优先级，对于占用CPU较多时间的函数，也要判断线程的优先级。

优化动画细节

通过TraceView工具发现，一些Banner轮播广告和文字动画在移出可视区域后，仍然存在定时刷新，不仅耗电也影响帧率。优化措施是在移出可视区域后停止动画轮播。

阻断多余requestLayout

在ListView滑动，广告动画变化等过程中，图片和文字有变化，经常会发现整个界面被重新布局，影响了性能。尤其布局复杂时，测量过程很费时导致明显卡顿。对于大小基本固定的控件和布局例如TextView, ImageView来说，这是多余的损耗。我们可以用自定义控件来阻断，重写方法requestLayout、onSizeChanged，如果大小没有变化就阻断这次请求。对于ViewPager等广告条，可以设置缓存子view的数量为广告的数量。

优化中间件

中间件的代码被上层业务方调用的比较频繁，容易有较多的高频率函数，也容易产生细节上的问题。除了频繁分配对象外，例如类初始化性能，同步锁的额外开销，接口的调用时间，枚举的使用等都是不能忽视的问题。

减少GC次数

安卓上的GC会引起性能卡顿，必须重点优化。除了第三章会详细介绍对于图片内存引起GC的优化，我们还做了如下工作：

- 减少对象分配，找出不必要的对象分配，如可以使用非包装类型的时候，使用了包装类型；字符串的+号和扩容；Handler.post(Runnable r)等频繁使用。
- 对象的复用，对于频繁分配的对象需要使用复用池。
- 尽早释放无用对象的引用，特别是大对象和集合对象，通过置为NULL，及时回收。

- 防止泄露，除了最基本的文件、流、数据库、网络访问等都要记得关闭以及unRegister自己注册的一些事件外，还要尽量少的使用静态变量和单例。
- 控制finalize方法的使用，在高频率函数中使用重写了finalize的类，会加重GC负担，使得性能上有几倍的差别。
- 合理选择容器，在性能上优先考虑数组，即使我们现在习惯了使用容器，也要注意频繁使用容器在性能上的隐患点：首先是扩容开销，HashMap扩容时重新Hash的开销较大。其次是内存开销，内存开销，HashMap需要额外的Map.Entry对象分配，需要额外内存，也容易产生更多的内存碎片。SparseArray和ArrayList等在内存方面更有优势。再次是遍历，对于实现了RandomAccess接口的容器如ArrayList的遍历，不应该使用foreach循环。
- 用工具监控和精雕细琢：在页面滑动过程中，通过Memory Monitor查看内存波动和GC情况，还可通过Allocation Tracker工具观察内存的分配，发现很多小对象的分配问题。
- 利用Trace For OpenGL工具找出界面上导致硬件加速耗时的点，例如一些圆角图片的处理等。通过多种工具和手段配合，手淘各个界面性能上有了较大的提高，平均帧率提高了20%，那么内存节省50%又是如何实现的哩，请看下文。

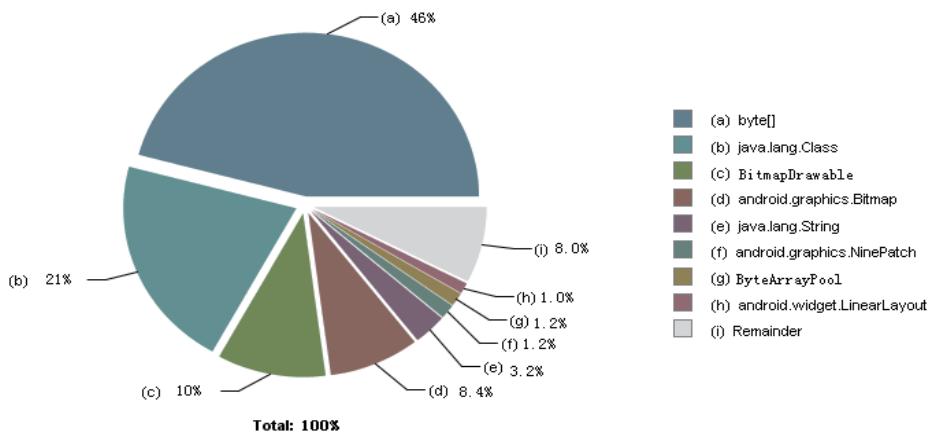
第三章 Android手机内存节省50%

Android上应用出现卡顿的核心原因之一是主线程完成绘制的周期过长引起丢帧。而影响主线程完成绘制时间的主要有两方面，一方面是主线程处于运行状态时需要做的任务太多但CPU资源有限，另外一方面是GC时Suspend时直接挂起了所有线程包括主线程。GC对总体性能的影响在4.x的系统上尤为突出，一部分是单次GC pause总时长，一部分是用户操作过程中GC发生的次数。而决定这两部分的因素就是Dalvik内存分配。那么在手淘这样的大型应用中到底是谁占用了内存大头呢？

谁占用了内存

基于双11前的手淘Android版本，我们在魅蓝note1（4.4 OS）上滑动完首页后，dump出其Dalvik Heap，整体内存占用的分布情况如下图。可以看出，byte数组（a）占用空间最大，绝大多数是用来存放Bitmap的像素数据（Pixel Data）。另外（c）与（d）一起占用了18.4%，byte数组加上

Bitmap、BitmapDrawable总共占用了64.4%，成为内存占用的主体。这也从侧面说明了手淘是以图片为浏览主体内容的大型应用。而往往图片需要较大的内存块，在分配时引起GC的可能性也往往最大。那我们能不能将图片这部分需要的内存移走而不在Dalvik Heap分配呢？如果能，那么不单GC会明显减少，同时Dalvik Heap总大小也会下降50%左右，对整体性能会有显著的提升。



何处安放的Pixel Data

Ashmemp即匿名共享内存，使用的核心过程是创建一个/dev/ashmem设备文件，控制反转设置文件的名字和大小，最终把设备符交给mmap就得到了共享内存。在Android系统中Binder进程间通信的实现就是依赖Ashmem完成不同进程间的内存共享。但此处并不利用其共享特性，而是使用它在Native Heap完

成内存分配。图片空间如何才能使用Ashmem，答案在Facebook推出的Fresco中已有提及，那就是解码时的purgeable标记，这样在系统底层解码位图时会走Ashmem空间分配，而非Dalvik Heap空间。这样就解决了像素数据存放由Dalvik到Native的问题了吗？

```

1. BitmapFactory.Options options = new BitmapFactory.Options();
2. /*
3. * inPurgeable can help avoid big Dalvik heap allocations (from API level 11 onward)
4. */
5. options.inPurgeable = true;
6. Bitmap bitmap = BitmapFactory.decodeByteArray(inputByteArray, 0, inputLength, options);

```

小心Bitmap空包弹

事实并非那么简单，最后实际解出来Bitmap没有像素数据（没有到Ashmem分配任何空间），根本没有去完成jpeg或者png解码。此时的Bitmap是个空包弹！它所做的只是把输入的解码前数据拷贝到了native内存，如果把这个Bitmap交给ImageView渲染就糟了，在View.draw()时Bitmap会在主线程进行图片解码。

而且不要天真的以为Bitmap解码一次之后再多次使用都不会引起二次解码，在系统内存紧张时底层可能回收Ashmem里这部分内存。回收后该Bitmap再次渲染时又将在主线程完成一次解码。如果就这样直接使用该机制，性能上无疑雪上加霜。

那么怎样才能避免这个隐形炸弹呢？还好SDK预留了一个C层方法AndroidBitmap_lockPixels。而lockPixels底层完成的工作大致如下图所示。

第一步是prepareBitmap完成真正的数据解码，在工作线程调用AndroidBitmap_lockPixels避免了在主线程进行数据解码；第二步是完成对分配出来的Ashmem空间的锁定，这样即使在系统内存紧张时，也不会回收Bitmap像素数据，避免多次解码。



貌似解决了Bitmap渲染的所有问题，但在手淘中则不然。为了兼容低版本系统以及提升wevp解码性能，我们使用了自己的解码库

libwevp.so，怎样把它解码出来的数据也存放到Ashmem呢？

libwevp借鸡生蛋

如果自有解码库libwevp.so要解码到Ashmem，通过SkBitmap、ashmem_create_region实现一套类似的机制是不太现实的。一方面Skia库的源码编译兼容会存在很大问题，另一方面很多系统层面的核心接口并没有对外。所以实现这点的关键还是要借助系统已经提供的purgeable到Ashmem的机制，借鸡生蛋，稳定性和成本上都能得到保证：

1. 依据图片宽高生成空JPEG。
2. 走系统解码接口完成Ashmem Bitmap生成。
3. 覆写Pixel Data地址在libwevp完成解码。

更进一步，迁移解码前数据

上面谈到的内存迁移都是针对Decoded像素数据的，而Encoded图像数据在解码时会在Dalvik Heap保存一份，解码完成后再释放；Ashmem方式解码时在底层又会拷贝一份到Native内存，这份数据直到整个Bitmap回收时才释放。那能否直接将网络下载的Encoded数据存放到Native内存，省去Dalvik Heap上的开销以及解码时的内存拷贝呢？的确可以，将网络流数据直接转移到MemoryFile可实现，但遗憾的是真机测试中发现，小米及其他国产“神机”（自改ROM），多线程使用MemoryFile获取fd到BitmapFactory解码，会出现系统死机，怀疑是在并发情况下系统代码级别的死锁造成。手机淘宝放弃了这种方案，改用ByteArrayPool复用池技术来减少Dalvik Heap针对Encoded Image的内存分配，效果也不错。如果应用能接受单线程解码，还是MemoryFile方案更具优势。

是放手的时候了

上文提到Bitmap像素数据存放到Ashmem，有读者可能担心数据回收问题，其实还是由GC来触发Ashmem内存的回收。在Dalvik层如果一

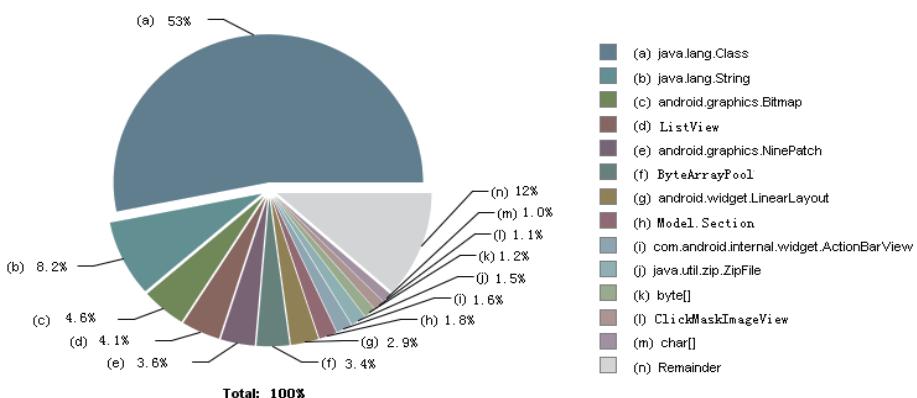
个Bitmap已经不被任何地方引用，那么在下一次GC时该Bitmap就会从Ashmem中回收，大致流程示意如下图。



再看内存占用

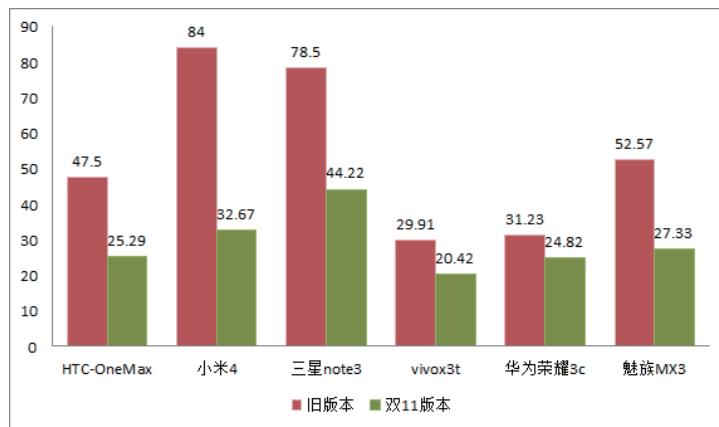
我们再次在魅蓝note1中dump出首页滑动后的内存，如下图可以看出，原来byte数组(k)

大量占用已经不存在了，Bitmap (c) 与 BitmapDrawable (已不在前14名当中) 的占用也急剧下降。应用的总体内存下降近60%。



在双11版本上，针对一些热门机型在搜索结果页不断滚动使用，进行了不同版本的内存占用对比分析，如下图。可以看出，除华为3c和

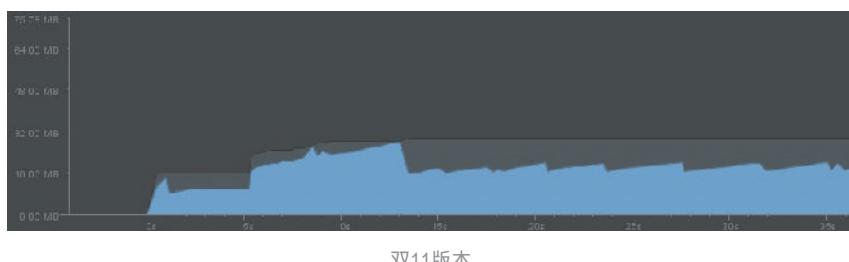
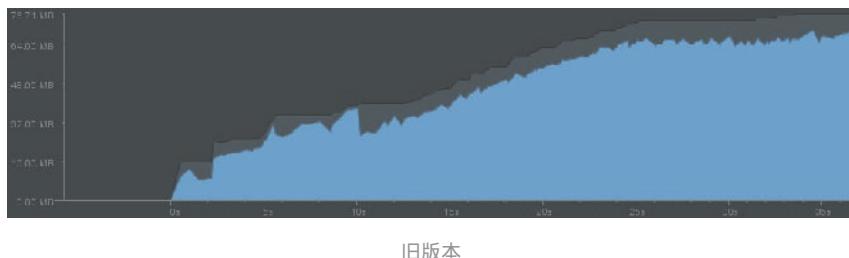
vivo这类系统内存偏小使用上一直受到控制、内存较为紧张的外，大部分机型内存的下降幅度都达到45%以上。



挠走GC之痒

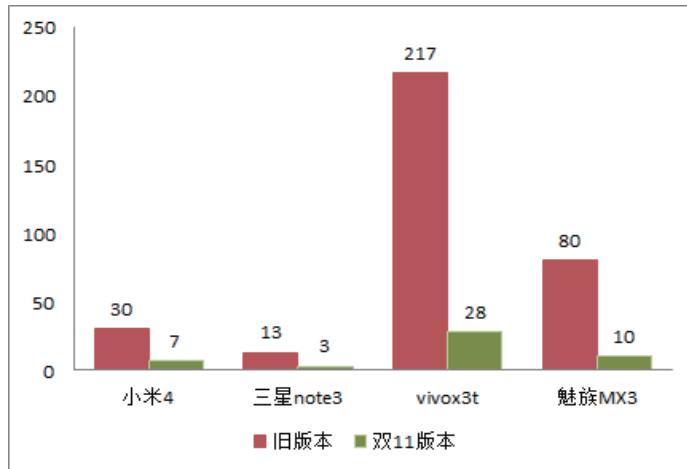
内存下降不是最终目的，最终要将GC对性能的影响降到最低。仍然以魅蓝note1打开首页后滑动到底的内存堆叠图来做对比。可以看到旧版本内存占用上升趋势相当明显，一路带有各式“毛刺”

直奔70MB，每形成一个毛刺就意味着一次GC。而双11版本中，内存只在初期有上升，而后很快下降到21MB左右，后期也显得平滑得多，没有那么多的“毛刺”，就意味着GC发生的次数在明显减少。



同时使用一些热门机型，针对双十一版本在首页不断滑动，进行前后版本的GC_FOR_ALLOC次数对比。

热门机型GC次数下降了4~8倍，效果非常明显。



通过上文描述的各个优化方案，手机淘宝于双十一前在大部分机型上达到了521目标—Android手机内存节省50%，启动时间和页面帧率提升20%，H5页面实现1s法则。

从持续不断的优化中，我们也得到了一套优化的经验闭环，由观察问题现象到分析原因，建立监控，定下量化目标，执行优化方案，验证结果数据再回到观察新问题。每一次闭环只能解决部分问题，只有不断抓住细微的优化点“啃”下去，才能得到螺旋上升的良好结果。

当然，随着手机机型的日益碎片化，程序功能的复杂化多样化，性能调优是没有止境的，在部分低端机和低内存手机上手淘性能问题依然不容乐观。欲穷千里目，还需更上一层楼，接下来我们还会努力通过更多更细致的优化方案来达到“如丝般顺滑”。

手机淘宝技术团队黎明（雷曼）、陈虹如（岑安）、吕承飞（吕行）、徐凯（鬼道）、王曜东（雪鹭）、赵密（正凡）、倪天雪（晓田）等同学参与本文创作。

京东11.11：商品搜索系统架构设计



作者 刘尚堃

京东商品搜索简介

京东商品搜索引擎是搜索推荐部自主研发的商品搜索引擎，主要功能是为海量京东用户提供精准、快速的购物体验。虽然只有短短几年的时间，我们的搜索引擎已经经过了多次618店庆和双11的考验，目前已经能够与人们日常使用的如谷歌、百度等全文搜索引擎相比，我们的产品与其有相通之处，比如涵盖亿级别商品的海量数据、支持短时超高并发查询、又有自己的业务特点：

1. 海量的数据，亿级别的商品量；
2. 高并发查询，日PV过亿；
3. 请求需要快速响应。

搜索已经成为我们日常不可或缺的应用，很难想象没有了Google、百度等搜索引擎，互联网会变成什么样。京东站内商品搜索对京东，就如同搜索引擎对互联网的关系。他们的共同之处：

1. 海量的数据，亿级别的商品量；
2. 高并发查询，日PV过亿；

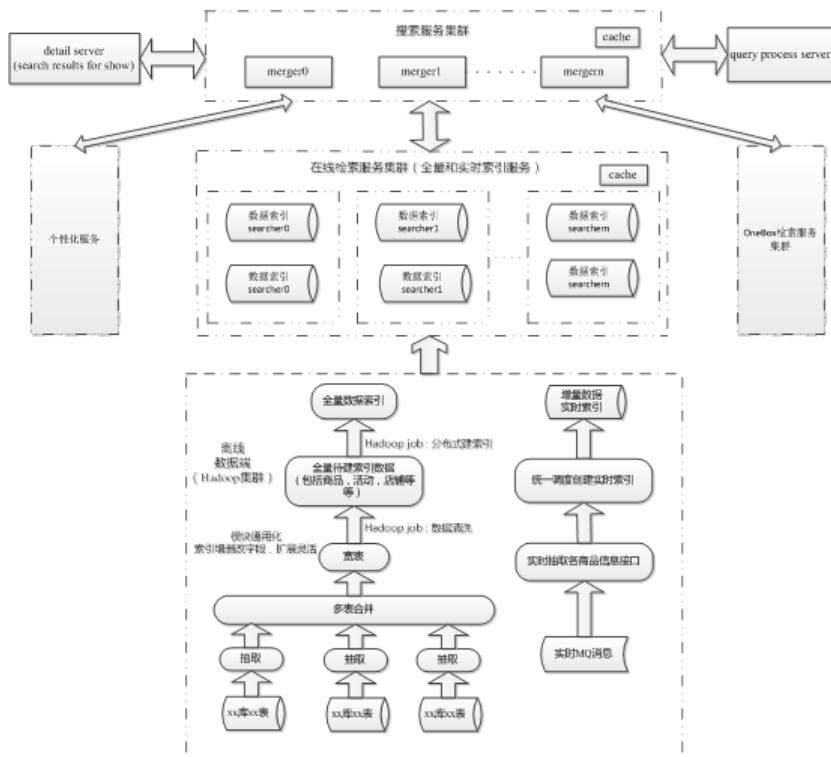
3. 请求需要快速响应。这些共同点使商品搜索使用了与大搜索类似的技术架构，将系统分为：
1. 离线信息处理系统；
2. 索引系统；
3. 搜索服务系；
4. 反馈和排序系统。

同时，商品搜索具有商业属性，与大搜索有一些不同之处：

1. 商品数据已经结构化，但散布在商品、库存、价格、促销、仓储等多个系统；
2. 召回率要求高，保证每一个正常的商品均能够被搜索到；
3. 为保证用户体验，商品信息变更（比如价格、库存的变化）实时性要求高，导致更新量大，每天的更新量为千万级别；

4. 较强的个性化需求，由于是一个相对垂直的搜索领域，需要满足用户的个性化搜索意图，比如用户搜索“小说”有的用户希望找言情小说有的人需要找武侠小说有的人希望找到励志小说。另外不同的人消费能力、性别、对配送时间的忍耐程度、对促销的偏好程度以及对属性比如“风格”、“材质”等偏好不同。以上这些需要有比较完善的用户画像系统来提供支持。

一、总体架构图



搜索服务集群：由很多个merger节点组成的集群。接收到查询query后，将请求通过qp触发有策略地下发到在线检索服务集群和其他服务集群，并对各个服务的返回结果进行合并排序，然后调用detail server包装结果，最终返回给用户。

query processor server：搜索query意图识别服务。

在线检索服务集群：由很多个searcher节点组成，每个searcher列对应一个小分片索引（包含全量数据和实时增量数据）。

detail server：搜索结果展示服务。

索引生产端：包含全量和增量数据生产，为在线检索服务集群提供全量索引和实时索引数据。

二、 离线信息处理系统

由于商品数据分布在不同的异构数据库当中有KV有关系型数据库，需要将这些数据抽取到京东搜索数据平台中，这分为全量抽取和实时抽取。

对于全量索引，由于商品数据散布于多个系统的库表中，为了便于索引处理，对多个系统的数据在商品维度进行合并，生成商品宽表。然后在数据平台上，使用MapReduce对商品数据进行清洗，之后进行离线业务逻辑处理，最终生成一份全量待索引数据。

对于实时索引，为了保证数据的实时性，实时调用各商品信息接口获取实时数据，将数据合并后采用与全量索引类似的方法处理数据，生成增量待索引数据。

三、 索引系统

此系统是搜索技术的核心，在进入这个系统之前，搜索信息仍然是以商品维度进行存储

的。索引系统负责生成一种以关键字维度进行存储的信息，一般称之为倒排索引。

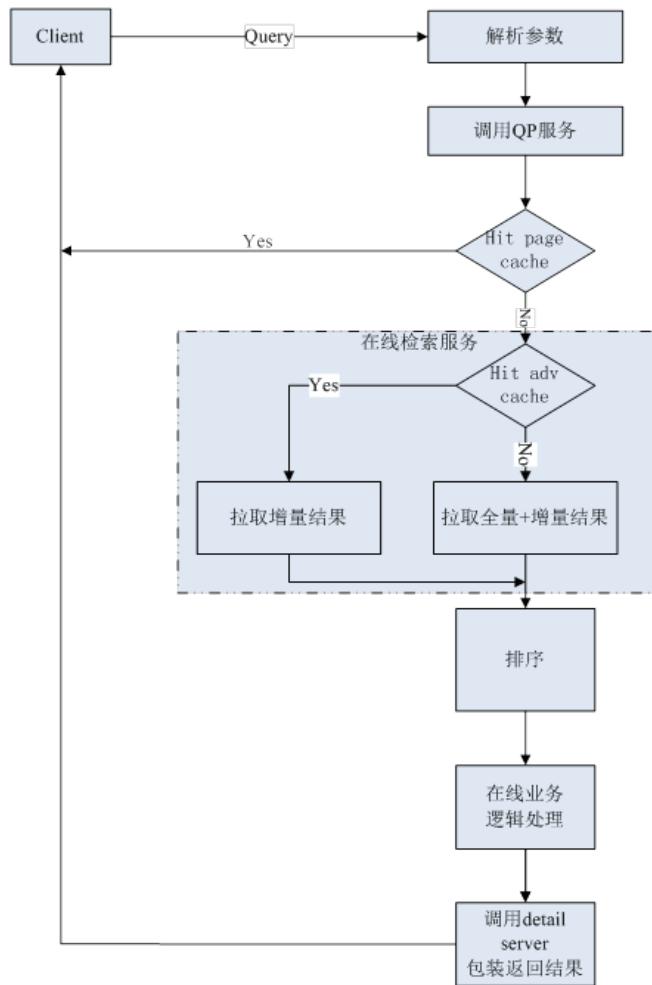
此系统对于全量和增量的处理是一致的，唯一的区别在于待处理数据量的差异。一般情况下，全量数据索引由于数据量庞大，采用hadoop进行；实时数据量小，采用单机进行索引生产。

四、 搜索服务系统

搜索服务系统是搜索真正接受用户请求并响应的系统。这个系统最初只有1列searcher组成在线检索服务。由于用户体验的需要，首先增加Query Processor服务，负责查询意图分析，提升搜索的准确性。随着访问量的增长，接着增加缓存模块，提升请求处理性能。接着随着数据量（商品量）的增长，将包装服务从检索服务中独立出去，成为detail server服务。数据量的进一步增长，对数据进行类似数据库分库分表的分片操作。这时候，在线检索服务由多个分片的searcher列组成。自然而然，需要一个merger服务，将多个分片的结果进行合并。至此，搜索基础服务系统完备。

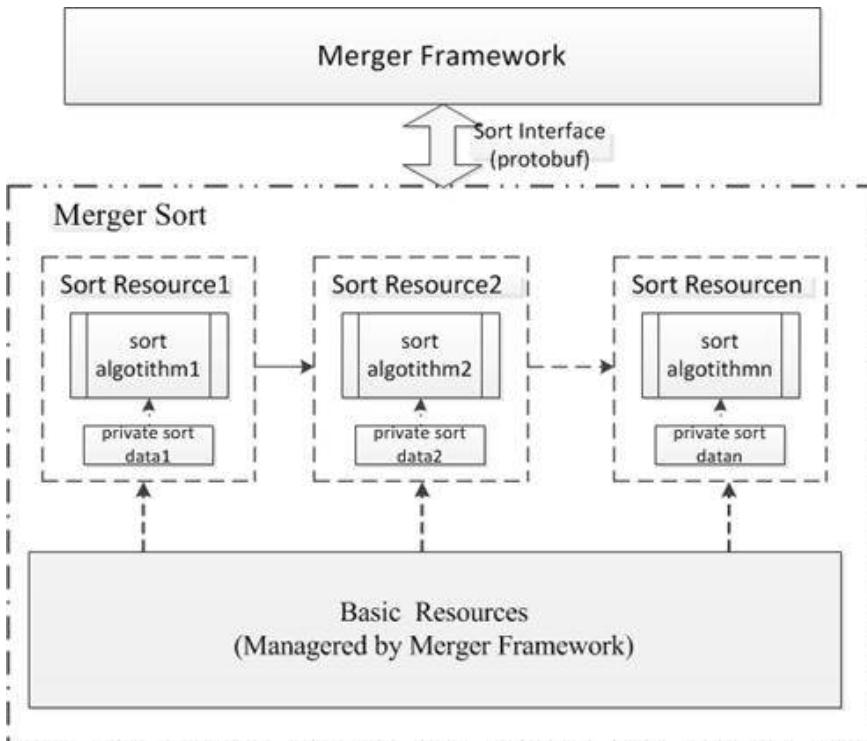
之后，无论是搜索量的增长或者数据量的增长，都可以通过扩容来满足。对于618、1111之类的搜索量增长，可通过增加每个searcher列服务器的数量来满足。而对于商品数据的不断增加，只需要对数据做更多的分片，相应地增加searcher列来满足。

搜索服务系统内部的处理流程如下：



在这个流程中，缓存模块和拉取结果模块非常稳定。而排序模块和在线业务逻辑处理模块经常需要改动。架构需要稳定，高效和通用。排序业务特点是实验模型多，开发迭代速度快，讲求效果。为了解决这一冲突，需要将排序业务与架构分离，以动态链接库的方式集成到搜索整体架构中，具体包

括文本策略和其他策略两个维度的相关性，文本策略相关性集成在searcher当中；其他策略相关性（包括反馈，个性化和业务调权等等）集成在merger当中。实现架构与排序业务各司其职，互不影响干扰。



五、反馈和排序系统

反馈系统主要包含用户行为数据的实时收集、加工，并将数据存储到数据集市当中，并对这些数据进行特征提取，排序最主要考核的线上指标是UV价值和转化率，所以还会利用这些数据根据优化目标构建起标注数据。然后基于机器学习的排序系统会针对特征构建出模型。京东排序模型是每天更新的训练之前大概半年的数据。京东搜索在基于模型的排序基础之上，上层还会有一层规则引擎，比如保障店铺和品牌的多样性，以及京东战略扶持的品牌等都通过业务引擎来实现。一般基于机器学习的排序模型需要较长期的投入但是模型更加健

壮不容易被作弊手段找到漏洞，并且可以让转化率和UV价值可持续的提升。规则引擎主要是为了快速反应市场的变化，起到立竿见影的效果。二者一个像中药一个像西药，中西结合疗效好。

六 针对今年双11的搜索系统性能优化

1. 故障秒级切换

今年搜索集群做到了三机房部署，任何一个机房出现断网、断电等问题可以秒级将流量切换到其它机房。并且搜索的部分应用部署到了弹性云上，可以进行动态扩容。

2. 大促期间索引数据实时更新

每年大促由于商品内容等信息更改频繁，涉及千万级的索引写操作，今年针对索引结构进行了调整彻底消灭掉了索引更新存在的一切锁机制，商品新增和修改操作变为链式更新。使大促期间商品的索引更新达到了妙极。

3. 大促期间的个性化搜索不降级

往年大促期间由于流量在平时5倍以上，高峰流量会在平时的7倍，为了保障系统稳定，个性化搜索都进行了降级处理。今年针对搜索的缓存进行了针对性的优化，实现了三级缓存结构。从底向上分别是针对term的缓存，相关性计算缓存和翻页缓存。最上层的翻页缓存很多时候会被用户的个性化请求击穿，但是底层的相关性缓存和term缓存的结果可以起到作用，这样不至于使CPU负载过高。

七、京东在电商搜索方面产品和技术的创新

1. 个性化搜索

个性化之前的搜索对于同一个查询，不同用户看到的结果是完全相同的。这可能并不符合所有用户的需求。在商品搜索中，这个问题尤为特出。因为商品搜索的用户可能特别青睐某些品牌、价格、店铺的商品，为了减少用户的筛选成本，需要对搜索结果按照用户进行个性化展示。

个性化的第一步是对用户和商品分别建模，第二步是将模型服务化。有了这两步之后，在用户进行查询时，merger同时调用用户模型服务和在线检索服务，用户模型服务返回用户维度特征，在

线检索服务返回商品信息，排序模块运用这两部分数据对结果进行重排序，最后给用户返回个性化结果。

2. 整合搜索

用户在使用搜索时，其目的不仅仅是查找商品，还可能查询服务、活动等信息。为了满足这一类需求，首先在Query Processor中增加对应意图的识别。第二步是将服务、活动等一系列垂直搜索整合并服务化。一旦QP识别出这类查询意图，就条用整合服务，将对应的结果返回给用户。

3. 情感搜索

情感搜索在于尽可能满足更多的搜索意图，这需要在后台构建一个强大的知识库体系。比如从海里评论中挖掘有意义的标签“成像效果好的相机”、“聚拢效果好的胸罩”、“适合送丈母娘”等，将这些信息一同构建到索引中去比如搜索“适合送基友的礼物”结合搜索意图分析相关的结果可以搜索出来。另外也可以从外部网站抓取有价值信息辅助构建知识库体系。

4. 图像检索

很多时候用户并不知道如何描述一个商品。通过搜索意图分析、情感分析可以尽可能挖掘搜索意图，很多时候用户根本无法描述，比如在超市看到一个进口食品或者一件时尚的衣服，可以通过拍照检索迅速在网上找到并比较价格，另外看到同事穿着一件比较喜欢的衣服也可以通过拍照检索来找到。目前京东正在开始展开这方面的开发。

离线方面主要通过CNN算法，对图片进行主题提取、提取相似特征、相同特征提取。引擎端主要是和搜索引擎类似的技术。图像搜索未来将可以开辟一个新的电商购物入口。京东目前正在研发新的图像检索引擎。

作者介绍

刘尚堃，京东推荐搜索部技术总监，ArchSummit全球架构师峰会顾问，有丰富的团队管理经验以及推荐、搜索、广告产品研发经验。



扫描二维码阅读更多关于京东内容

当当11.11：促销系统与交易系统的重构实践



作者 史海峰

电商行业近年来发展势头迅猛，诸多巨头成功上市，业务模式不断升级，促销手段花样百出。双十一成为各路电商运营能力的年度大考，同时也是对电商技术平台能力的极限测试，每年进行了重大改版升级的系统只有经过双十一的枪林弹雨才能浴火重生。

在当当，2015年的双11，面临考验的是促销系统和交易系统，两者都是电商体系的核心组成部分。此次双11专题，InfoQ特别邀请EGO会员、当当架构部总监史海峰先生，为大家讲述当当双11背后的技术故事。

促销系统重构

如今大规模促销已经成为大大小小的电商平台及入驻商家运营的常态。随着业务的复杂化、运营的精细化，以及品类、平台、渠道的不断丰富，各种新的促销形式也层出不穷，贯穿从商品展示、搜索、购买、支付等整个流程，电商对于精细化、精准化促销运营的需求也越来越强烈。

一次促销活动几十万商品，一天之内几十个、上百个促销活动已是家常便饭，至于入驻商家的常

态促销更是不胜枚举。双十一期间，电商平台和商家更是会使出浑身解数，火力全开，无品不促销。

促销规则支持分时段设置，多个活动能够叠加，促销系统中的数据量甚至会超过商品信息系统，而且促销内容会根据执行效果快速调整，这些都对促销系统提出了更高的要求，促销系统越强大，促销活动才能玩得越疯狂。

我们在重构前面临的状况，是促销模型比较陈旧、扩展性差，促销系统成熟度低、与其他系统耦合严重，新增一个促销类型可能牵动从单品展示、搜索、推荐、购物车、交易、订单、退换货、库存、价格、促销自身等一系列产品线的变更。

因此，促销系统的重构势在必行，数据模型与运营的贴合度决定的扩展性、灵活性，系统解耦和更强大的数据处理能力，是核心改进点。最基本的促销模型很简单，如下图：



在当当，有一些“类促销”业务，从广义上可以归入促销范畴，但业务与数据均不属于促销系统，在设计中，我们考虑将这类业务逐渐回收；另外，促销系统能不能承担一些营销的功能？带着这两点考虑，在促销基础上进一步抽象出活动模型。

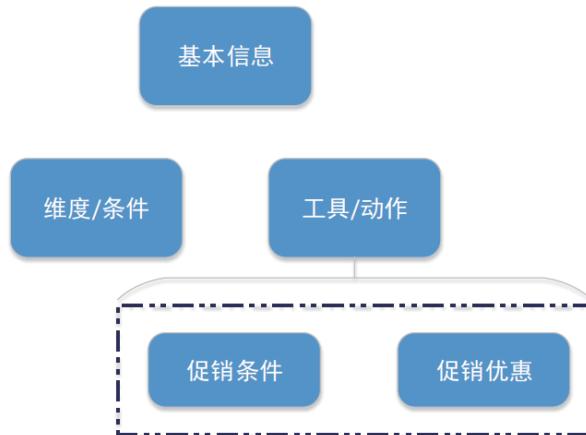
什么是活动？我们认为任何一个有时间范围的事件/动作均可称为活动，活动则抽象为三要素组成：基础信息、维度（条件）、工具（动作）。

例如，在11月1日10:00-12:00在第一会议室开

双十一准备会，讨论双十一各系统需要准备的事项，需要各系统负责人参加；那么这个活动的基础信息包括时间（11月1日10:00-12:00）、主题（双十一准备会），维度包括地点（第一会议室）、与会人员（各系统负责人），工具（动作）包括议题以及讨论本身。

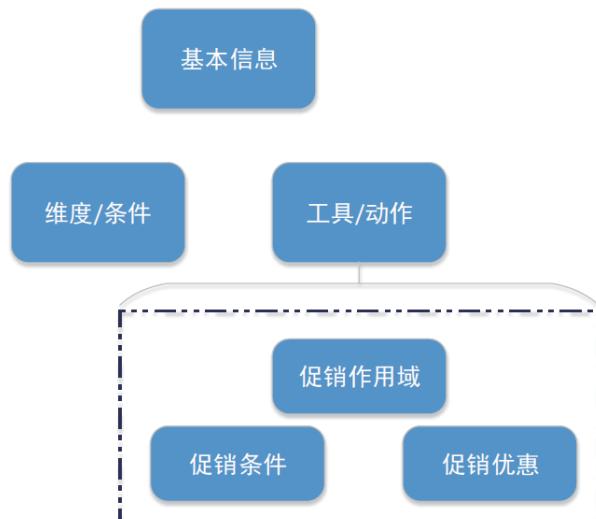
那么推而广之，理论上，只要有相应的工具对接，可以用这个极简的活动模型，去管理任何一类活动，这样模型就变为了两层：





实际业务中我们遇到过的一些关于促销计算单元的头疼问题。买了一堆商品，到底哪几个应该作为一组计算条件和优惠，在促销叠加的场景这一点显得更为复杂。所以我们引入作用域来定义这个计算单元的范围。例如常规的限时抢促销，每个SKU有自己的价格，那么SKU就是这个促销的计算单

元，也就是促销的作用域；例如第二件5折，可能会按SPU来做，你买一个红的一个蓝的，还是能享受促销，那么SPU成为了这个促销的计算单元；诸如此类，现有及未来可扩展的还有店铺、品类、品牌等等。简言之，这个作用域成为促销计算引擎进行计算单元分组的依据。于是模型又变成了这样：



举个例子，我们要在11月11日11:00-12:00针对IT技术类图书进行满200元减100元促销，购买过此类图书的客户每本书每人限购一册。那么这个活动的基础信息包括时间（11月11日11:00-12:00）、主题（程序猿光棍节福利）；维度包括商品品类

（IT技术）、用户范围（购买过此类图书的客户）；工具是满额减促销、以金额满200元为条件、减100元为优惠，此外还有限购策略为限购1本，作用域为参与活动的所有商品。



可能这里会引发困扰，基础信息的时间为何不能算做时间维度？维度也定义了一些限制条件，那和促销工具模型里的条件有什么区别？时间之所以不归入维度，是基于前面对活动的定义，时间范围是

必须的，而维度是可选的；促销模型中的条件只对于促销工具有效和有意义，而维度则有更广泛的普适性，例如平台、渠道、地区、用户、商品等，与工具是什么并无关系。基础模型定型之后，我们开始

基本信息	活动名称	活动时间				
活动维度	地区	平台	用户等级	渠道	商品	
促销条件	金额	数量				
促销优惠	送赠品	打N折	减N元	以N元销售		
促销作用域	SKU	SPU	品牌	品类	店铺	ALL
促销规则属性	限购策略	库存策略	阶梯/滚动			

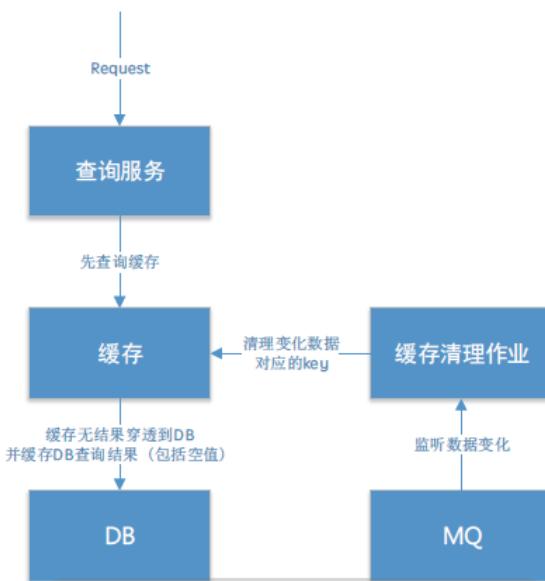
着手解耦方面的设计：

首先是系统交互解耦，将直读DB和存储冗余促销数据的系统修改为调用服务及监听MQ；然后是逻辑回收，包括将促销校验与促销计算提取为交易服务，将原先由购物车、交易系统自行处理的促销逻辑回收；从业务上，将促销工具的属性进行提取，诸如类型枚举、促销标签、限购策略、库存策略等，以期外围系统尽量少的关注促销类型，通过促销ID拿到所需信息直接使用；未来则关注于业务层面的梳理与整合，逐步回收适用于活动模型的其他“类促销”业务。

系统解耦后，促销系统需要提供各系统所需要的服务，必须具备更强大的数据处理能力和更好的

性能表现。应用架构实现上，从前端页面到后端逻辑，尽量避免有逻辑与促销类型直接绑定，全部以插件化方式与促销模型对接，完全根据促销类型的配置进行组装。针对不同维度、条件、优惠、促销属性，定制页面模板及业务逻辑，使得新增一种促销类型（在已有维度、条件、优惠下）仅需配置即可完成。

促销系统的查询服务需要同时为多个系统提供数据，对TPS要求很高，同时促销的时效性又要求很高的实时性。我们采用的方式是在数据库前加Redis缓存，提高响应速度，同时监听MQ，根据事件清理相应的缓存数据。



这种设计方案也有一些可能的坑，例如Redis缓存虽然减轻了DB压力，但对于计算密集型应用并未减轻应用服务器压力，IO没有节省还增加了序列化的开销；事件驱动清理缓存在读写分离场景下，有可能比主从同步更快，造成缓存数据错误。这也是具体应用中需要注意的地方。

促销系统重构上线后，使多渠道（终端）、多区域化营销成为简单易行的配置操作，显著提高了当当运营能力，当当双十一呈现出更多的想象空间。

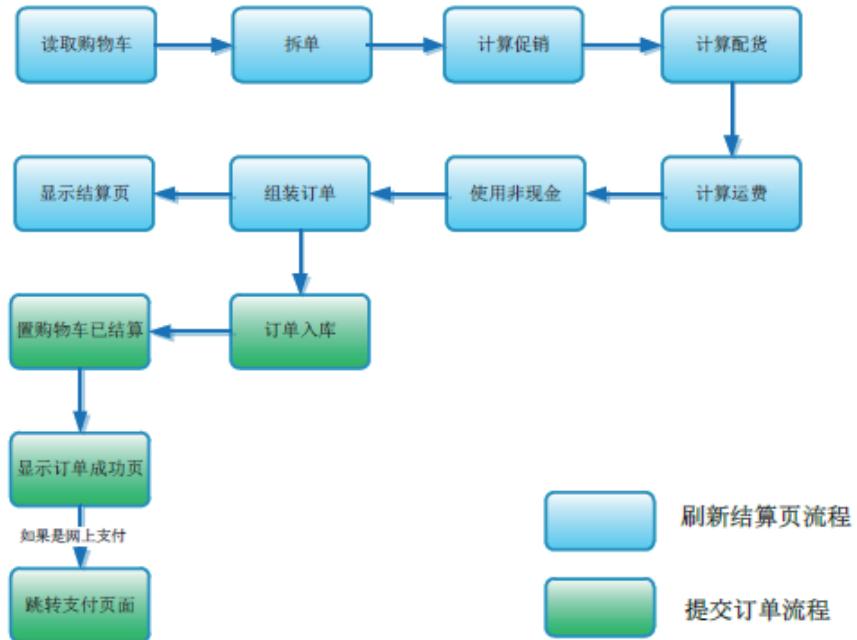
交易系统重构

交易系统是客户购物流程中最重要的环节，主要任务是完成购物车中商品信息获取、拆单、促销计算、配货计算、运费计算、非现金支付的使用以及生成订单等操作，聚合各方面业务逻辑，计算

非常复杂，而且响应速度影响购买转化率，一旦出现故障，直接影响营业收入，可谓电商最为敏感的核心系统，决定对其进行重构需要极大的魄力。当当原有交易系统采用.NET技术框架，运行多年，很好的支撑了购买流程，但是弊端逐渐显露。首先是技术体系属于微软系，每年要花费大量成本购买服务；其次是随着业务需求的不断叠加，其结构及可维护性逐年下降，尤其是众多小版本结算的存在，使得功能扩展异常艰难。

基于以上因素，交易系统团队在2014年底启动重构项目，2015年10月底新老版本完成切换。此次重构耗费约1500人天，重构代码17万行，全部切换至Java开源技术架构，为公司节约大量成本，并进行了架构优化，整体性能平均提升25%。

交易系统业务主流程图如下：



交易系统重构引入了许多业界成熟的技术实现方案，主要有以下几点：

1. 集中化配置

集中化配置方式，一点配置，所有实例可见，更易于管理，而且配置修改后，通过热加载方式，立刻生效，快速便捷。而原有交易系统修改配置后，必须重启系统才能生效。

2. 页面缓存技术

用户请求一次交易结算页面，会调用各种后端服务，而由于逻辑的复杂性，每次服务调用都会调用订单计算大流程，导致页面刷新缓慢。新交易系统将大流程计算结果进行缓存，在一次页面请求范围内，后续调用直接用缓存结果，极大提高了页面的刷新速度。

3. 小版本合并

由于历史原因，交易系统存在很多版本的结算逻辑。最常用的是统一结算，还有一些特殊类型的结算，如秒杀、一键下单、补发货等等，逻辑与统一结算稍有不同，统称为小版本结算。因小版本结算与统一结算大部分逻辑相同，因此新交易系统将二者合到了一起，共享基础逻辑，而不同的逻辑则单独处理，极大提高了可维护性。

4. 灰度发布、无缝切换

借助了Nginx在运行状态下可以reload配置，而基本不影响对外提供服务的能力。每个Nginx负载两台应用服务器，灰度发布时，将Nginx配置更改为只负载一台应用服务器，即可对另一台进行部署。用户请求不会导向正在部署中的服务器，从而不影响用户下单。

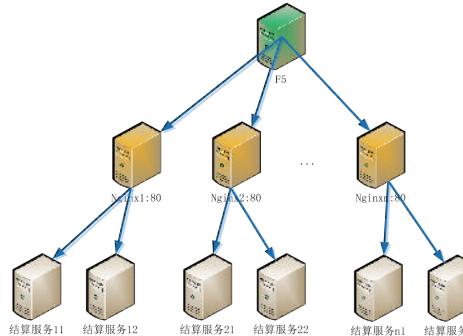


图1. 初始状态

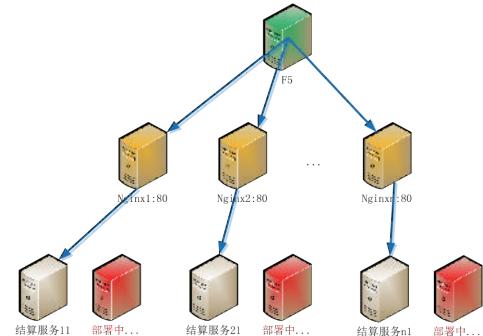


图2. 部署中

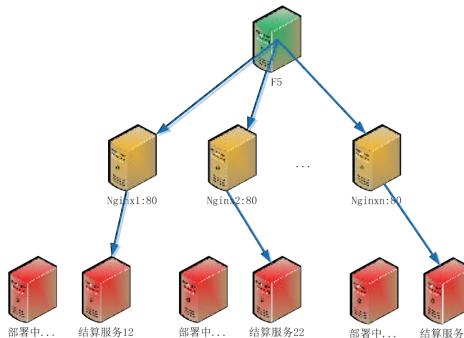


图3. 部署中

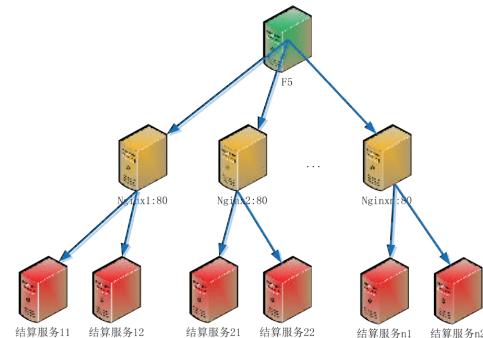


图4. 部署完成

5. 并行比对

交易系统重构后，尽管进行了大量的测试，仍不能放心部署上线。因为交易系统的计算都和金钱有关，必须慎之又慎，我们提出了线上并行比对方案，根据老交易系统比对新交易，保证其逻辑正确。原理如下：

1. 用户请求到达老交易系统
2. 根据条件将部分请求数据复制，发送至调用 mock 服务的新交易系统
3. 新老交易同时计算，结果存入各自的数据库，但只有老交易结果对用户公开
4. 对新老计算结果进行比对。这样，既实现了比对目的，又不会影响线上环境。

6. 分流

比对之后，新交易系统也不能立即全面上线，那样可能有巨大风险。我们开发了分流功能，按照用户 id 来分流，正式分流前，先使用测试白名单中的用户进行预验证。预验证通过后，再按比例由低至高逐步切换。

7. Web服务器按需伸缩

基于前面所讲的灰度发布技术，新交易系统很容易做到按需伸缩。正常情况下，每个 Nginx 负载两台应用服务器。双十一需要扩容时，将待扩服务器 ip 地址加入 Nginx 配置，重新 reload，该 Nginx 就可负载更多台应用服务器了。

交易系统上线后为备战双十一，确保万无一失，利用老交易系统还未下线的条件进行了线上压测。为达到最准确的测试效果，且不影响正常系统运行，进行了以下的准备：

1. 测试环境、数据与生产环境一致

在准备测试环境方面，为了保证测试结果更接近真实结果，本次测试使用线上环境，通过大量测试账号对线上的真实促销商品进行测试，这样也对新交易系统所依赖的各系统服务做了检验。

2. 线上业务运行保障

测试阶段线上的请求切到老交易系统，压测请求发送到新交易系统，使测试和生产业务进行分离，保证了各自服务器资源的独立性。在应用服务器层面使测试过程不会影响到线上正常交易。

3. 拦截订单回收库存

由于使用线上环境测试，需要对测试订单进行拦截，避免进入生产流程，并回收占用的库存，

使商品不会耗尽库存，采用的方法是在自动审单系统中将测试账户加入黑名单，测试订单提交后会被拦截，然后取消订单释放库存。为防止测试过程占用了商品的全部库存而影响线上销售，测试商品池基数很大，并且过滤掉了库存数量较少的商品，在执行测试过程中控制每个商品的使用次数，保证可销售库存占用在安全范围内。

4. 恶意用户策略控制

因为在交易下单过程中包含恶意用户判定的策略，会对用户进行隔离，禁止连续大量下单，线上压测时根据实际情况短时间内调整了安全级别，保证订单成功率。

经过多轮线上压测，新交易系统的高可用性得到验证，得到了实际性能指标，并根据双十一流量估算进行了扩容部署，将以崭新的面貌为广大客户提供稳定可靠的网购服务。

（感谢周裕杰、王胜军、马铁强对本文内容提供支持）



史海锋是EGO会员，什么是EGO？关注了解

苏宁11.11：系统拆分的一些经验谈



作者 杨学增

“平京战役”一发布使本来就热闹的电商促销大战呛出了火药味，也为双11的大促增添了许多谈资，更让消费者享受到实实在在的优惠。而在技术上这种竞争则温和许多。

技术上的压力来源于业务的需求。苏宁阿里战略合作后，易购赢得了社会的广泛关注，系统的流量在苏宁的传统促销节8.18显现出来；加上苏宁的双11销售目标，使得我们系统承担的压力更大了。

技术上的准备不是一蹴而就的，尤其像易购这样的大系统，更需要长期的积累和演变。历经多年的大促，目前苏宁在技术线上的准备变得也非常清晰和严谨。接下来我们将分享下在系统拆分、基础平台、研发流程和系统保障四个方面的经验。

一、 系统拆分方面

目前苏宁易购在线业务有相当多的系统，很多系统的TPS（每秒钟的请求数量）都非常高，而且做到了水平扩展。这些都是不断拆分和重构的结果，几年以前则是另一番景象。

以前苏宁易购的主站系统都在基于IBM的Commerce开发的B2C平台上。为了方便扩容，线上独立部署了很多套这样的系统。当用户浏览时，会根据用户ID选择其中的一个系统。这个系统非常庞大，它包含了所有的业务代

码、单个包超过几百MB、启动时间非常长、部署这样的系统花费了更多时间。每当大促来的时候，秒杀、抢购和正常的交易蜂拥而至，宕机就在所难免了。驾驭庞大的系统意味着高超的能力，大牛也许能够轻松地理清各种逻辑关系，但是对普通工程师而言变有相当难度。修改频繁出错、那种错牵一发而毁全身的痛使得工程师们对业务的变更畏手畏脚。大促中的几次宕机、新业务难以在老系统上扩展促使我们走上了系统拆分的道路。

进行系统拆分是需要充分准备的，没有一个合理的规划只会让系统的职责扯不断、理还乱，外加各种抱怨。系统拆分要分析主流程、分离主干系统和枝叶系统、把主干系统根据业务的内聚性独立出来，做到分别部署。我们分析了电商的主要功能和重运营的特点。我们根据业务功能拆分了几大核心系统：会员、商品、库存、价格、购物车、交易、订单和内容管理等，并且组建了对应的研发中心来维护。根据交易环节的系统压力，独立出抢购和秒杀系统，还拆分出购物券、云钻等营销类的系统，并组建了营销中心。

系统职责明确了，能够独立发布了，进而快速响应新业务。系统变小了，新人只要了解很少的逻辑就能上手，而且模块小了、关注度高了，新人更能深入了解和优化系统。

二、基础平台方面

系统拆分只是把系统的职责理顺了，接踵而来的是系统间的通讯问题和各个系统重复造轮子的问题。针对这些问题，苏宁成立了基础研发中心。基础研发中心属于一级中心，这也显示了苏宁对基础研发的重视。

苏宁的基础平台主要集中解决了以下方面的问题：基础架构方面包括自建CDN、云计算和云存储；通用系统方面包括短信、邮件、验证等系统；系统集成包方面包括系统之间的通讯、统一验证和内部管理系统的统一权限；中间件方面包括Session共享、分布式调用链、Redis分片存储、数据库的分库分表中间件、统一配置管理和流控；平台方面：运维监控平台，持续集成平台，大数据分析平台；还有针对安全的风控系统等。因为内容繁多，这里选取其中几个做简要介绍。

云计算平台、云存储平台已经成为互联网的标准配。云计算平台提供了KVM、Docker的中间件服务，

用来部署着苏宁大量的系统。云存储支持文件存储和对象存储两种方式，支撑着苏宁的海量存储需求。苏宁易购内部通讯方式主要有两套系统：MYESB和RSF。MYESB是一个安全的集中消息转发服务系统，提供了同步和异步两种服务接口。其中同步基于HTTP+XML，异步基于MQ。RSF类似阿里的Dubbo，提供远程调用的机制，支持HTTP和TCP通讯服务。它的优点是使消费方直接调用服务方，不需要像MYESB需要代理转发，减少了系统延迟。RSF的TCP通讯是长链的，可以减少网络开销。由于历史的原因苏宁有很多异构的系统，这些系统通常需要MYESB来解决异构环境通讯问题；一般系统内部都是用RSF作为通讯方式。

持续交付平台主要包括了代码基础框架自动生成、代码质量分析、代码的自动化部署和代码权限管理。有了上面的准备，构建一个新系统就变得容易起来。

监控是系统的眼睛，尤其是针对线上的系统。监控能让我们知道系统的运行状态，在系统出问题时，也能让我们知道系统那时发生了什么。目前采集监控数据的手段基本都是定时采样、日志收集分析和及时消息触发。分析监控数据手段基本上有实时和离线两种。实时监控分析偏重于实时性，分析的纬度有局限性；离线的监控分析系统偏重于一些定制统计和分析，能够对系统做出比较全面的分析。苏宁的监控平台包括了对硬件资源、操作系统、中间件以及业务系统的监控。苏宁目前有实时日志系统，偏向分析的LogMonitor系统以及针对移动端的监控系统。实时日志分析系统基于ELK技术，可以实时监测请求状态、系统错误和进行多维度查询分析；LogMonitor可以统计分析接口最大、平均处理时间和历史接口的性能对比；新上线的CloudyTrace能更好地识别tp90、tp99等高级功能。

目前苏宁自建的CDN服务承担的流量也相当的多。当流量比较大的时候，自身的业务会受到第三方CDN的质量和价格约束。选择自建也是大型电商系统选择的必经之路。

基础平台解决了共性问题后，上层业务系统只需要集中关注业务领域。这样一来业务系统开发的周期大大缩短，所以苏宁很多系统的开发周期都很短。当基础平台支撑能力越强，它为业务系统提供的可扩展性、可靠性、性能就越好，也能够保证业务系统的飞速发展，从而使业务真正站在巨人的肩上。从以往的资料来看，阿里、京东的业务和人员也都是在建立了基础平台之后快速发展起来的。

三、研发流程

近两年苏宁进入了快速发展期，各种系统不断地被拆分和研发。仅消费者研发中心就有上百个系统，各个系统的质量参差不齐。越靠近用户端的系统，越容易成为用户反馈和抱怨的密集地。如何把控系统的质量？对此，中心在研发流程上下了大力气。总结起来就是在关键的研发点制定了对应的检查单和组织会议对其检查。经得起推敲的产品才是好产品。对于产品的评审，力度也是相当大。不单是对产品需求，大到产品的运营指标，小到产品的文字都会反复评审。通过架构设计模版和设计检查单，确保这些问题在设计系统时候都认真思考过。这样一来架构设计变得简单、有序。

除了通过代码走查、sonar平台、各种测试等手段，中心也采用代码飞检来确保代码质量。代码飞检就是定期快速评审系统的核心代码。与面向项目组内的代码走查不同的是参加代码飞检人员的级别比较高，都是各个系统负责人、架构师以及总监。当各个系统裸露在大家面前的时候，系

统的美与丑很容易被区分出来。通过这种方式，我们发现了很多优秀的代码，也发现了幕后的高手。意想不到的效果是优秀的人才很快浮出水面，而不是靠挖掘了。

流程发布检查单是系统的最后一关，它需要经过产品负责人、开发负责人、QA、测试负责人、DBA、运维人员和线上验证人员对各个环节进行确认，让系统上线过程少出问题，出现了问题也能及时下架。

其实很多公司有这样的流程，但是受抵触情绪或者投入力度不够的影响，难免流于形式。而经验告诉我们，这些实践起来真的不难，而且非常有效。

四、系统保障方面

双11期间，除了开发新业务，我们主要的工作就是确保系统能够顶住流量的压力。为此我们做了两手准备：一个是提高系统的负载能力，另一个就是做应急预案准备。

系统的压力来自流量。首先我们根据历史数据对双11的流量进行了预估，细化到每个系统的PV、UV、峰值TPS，要求每个系统要努力达到这些指标。然后我们对目前系统的压力、容量和相关指标进行统计，按照预期的流量判断系统的服务是否满足要求。如果不满足就需要通过优化和扩容来完成，能优化则优先通过优化解决，因为扩容意味更多的服务器资源。

优化方面，我们对系统进行自上而下的体检，针对体检发现的问题制定了相关的方案。具体是对系统架构梳理、关键代码优化以及中间件调优。好的架构可以节约很多资源。架构梳理主要对重点业务的处理流程和处理的链路进行审核。系统依赖问题是经常遇到，一个系统经常依赖多个系统，一个业务需要多次调用第三方服务，流

程链相当长。有时候这种依赖不是必须的，可以通过依赖调用改成第三方主动推送数据来消除这种依赖。

在压测过程中，我们经常发现接口的性能不够高。主要是因为长长的调用链、不分层次垒代码、没有良好的处理模型。很多人因为经验和时间的原因，验证完功能就认为搞定了，造成性能有很大的问题，更别说可扩展性和可维护性。做代码优化要静下心来，深入了解业务特点，构建优化方案。消息推送系统是我们重点优化的一个系统。我们对其进行改造：从数据库中取用户列表，改成把用户列表存储在Redis缓存中，性能一下子提高很多；以前都是一股脑的随机发送，现在则是首先推送重点城市，保证重点城市的用户在几分钟内接收到数据。中间件主要针对JVM策略、各种链接池、系统链接数、缓存和读写分离做一些调正。通过压测最容易找到性能的瓶颈，为了让数据更真实，重要系统在夜深人静的时候还在生产系统上直接压测，帮助了我们快速发现系统的真实能力和系统瓶颈。优化和测试验证是个反复的过程，这期间的过程也相当辛苦。

系统如果能够按照预期流量来，我们大部分系统是可以支撑的。但问题是不清楚实际会来多少，尤其是峰值的时候；而且这些流量除了正常用户的访问流量，还有一些异常的流量。为此我们准备了应急预案和相应的操作手册。我们的应急预案主要包括几个方面黑名单、限流和降级。

黑名单主要是拒绝恶意的系统访问，如：
IP黑名单、用户黑名单。

限流则是在流量超过系统负载警戒线时，主动丢弃相关的请求，以保全系统。现在的系统的都是分层部署的，限流可以通过防火墙流

控功能、中间件的流控功能和流控组件来实现。苏宁的流控组件还支持IP、用户、URL三个纬度来控制的访问频度，以防止过度请求。

降级可以让系统临时承担更大的流量压力。我们经常通过下面策略进行降级：屏蔽非关键业务的入口、关闭影响性能非关键业务功能、页面静态化、开启验证码策略延缓系统压力、延长缓存的时间牺牲实时性、放弃后端的补偿机制以减少调用链时间等。在多大压力的情况下开启什么的降级策略，需要再定义和演练。在实践过程中，我们定义了不同级别的降级策略、每个级别对应不同的流量压力。

另外很重要的一点是这些应急预案一定要演练，否则预案就变得不可靠，也极有可能出现重大问题。技术上的准备要经得起考验。双11就要到了，正是我们大考的时候，我们将严阵以待，确保系统平稳运行。双11之后，相信有更多的文章和数据会分享出来，让大家更多了解我们是如何一路走来的。

蘑菇街11.11：私有云平台的Docker应用实践



作者 郭嘉

对于蘑菇街而言，每年的11.11已经成为一年中最大的考验，考验的是系统稳定性，容灾能力，紧急故障处理，运维等各个方面的能力。蘑菇街的私有云平台，从无到有，已经经过了近一年的发展，生产环境上经历了3次大促，稳定性方面得到了初步验证。本文我将从架构、技术选型、应用等角度来谈谈蘑菇街的私有云平台。

蘑菇街的私有云平台（以下简称蘑菇街私有云）是蘑菇街面向内部上层业务提供的基础性平台。通过基础设施的服务化和平台化，可以使上层业务能够更加专注在业务自身，而不是关心底层运行环境的差异性。它通过基于Docker的CaaS层和

KVM的IaaS层来为上层提供IaaS/PaaS层的云服务，以提高物理资源的利用率，以及业务部署和交付的效率，并促进应用架构的拆分和微服务化。

在架构选型的时候，我们认为Docker的轻量化，秒级启动，标准化的打包 / 部署 / 运行的方案，镜像的快速分发，基于镜像的灰度发布等特性，都十分适合我们的应用场景。而Docker自身的集群管理能力在当时条件下还很不成熟，因此我们没有选择刚出现的Swarm，而是用了业界最成熟的OpenStack，这样能同时管理Docker和KVM虚拟机。相对来说，Docker适合于无状态，分布式的业务，KVM适合对安全性，隔离性要求更高的业务。

对于上层业务来说，它不需要关心是运行在容器中，还是KVM虚拟机里。今后的思路是应用的微服务化，把上层的业务进行拆分，变成一个个微服务，从而对接PaaS基于容器的部署和灰度发布。

技术架构

在介绍双十一的准备工作之前，我先简单介绍一下蘑菇街私有云的技术架构。

我们采用的是OpenStack+novadocker+Docker的架构模式，novadocker是StackForge上一个开源项目，它做为nova的一个插件，通过调用Docker的RESTful接口来控制容器的启停等动作。每个Docker就是所谓的“胖容器”，它会有独立的IP地址，通过supervisord来管理容器内的子进程，常见的如SSHD、监控agent等进程。



我们在IaaS的基础上自研了PaaS层的编排调度等组件，实现了应用的弹性伸缩、灰度升级，支持一定的调度策略。我们通过Docker和Jenkins实现了持续集成（CI）。Git中的项目如果发生了git push等动作，便会触发Jenkins Job进行自动构建，如果构建成功便会生成Docker Image并push到镜像仓库。基于CI生成的Docker Image，可以通过PaaS的API或界面，进行开发测试环境的实例更新，并最终进行生产环境的实例更新，从而实现持续集成和持续交付。

网络方面，我们没有采用Docker默认提供的NAT网络模式，NAT会造成一定的性能损失。通过OpenStack，我们支持Linux bridge和openvswitch，不需要启动iptables，Docker的性能接近物理机的95%。

准备工作

稳定性

迎战双11，最重要的当然是确保稳定性。通过近一年的产品化和实际使用，我们积累了丰富的提高稳定性的经验。

对于那些已遇到过的问题，需要及时采用各种方式进行解决或者规避。

比如说，CentOS6.5对network namespace支持不好，在Docker容器内创建Linux bridge会导致内核crash，upstream在2.6.32-504中修复了这个bug，因此线上集群的内核版本，必须升级至2.6.32-504或以上。

又比如，CentOS6.5自带的device mapper存在dm-thin discard导致内核可能随机crash，这个问题我们早在四月份的时候已经发现并解决了，解决的办法是关闭discard support，在docker配置中添加“--storage-opt dm.mountopt=nodiscard --storage-opt dm.blkdiscard=false”，并且严格禁止磁盘超配，因为磁盘超配可能会导致整个device mapper无法分配磁盘空间，而把整个文件系统变成只读，从而引起严重问题。

监控

我们在双11前重点加强的是针对容器的监控。在此之前，我们已经自研了一套container tools。主要功能有两个：一是能够以容器为粒度计算load值，可以根据load值进行容器粒度的qps限流。二是替换了原有的top、free、iostat、uptime等命令，确保运维在容器内使用常用命令时看到的是容器的值，而不是整个物理机的值。双十一之后我们还会把lxcfs移植到我们的平台上。

在宿主机上，我们增加了多维度的阈值监控和报警，包括对关键进程的存活性监控 / 语义监控，内核日志的监控，实时pid数量的监控，网络连接跟踪数的监控，容器oom的监控报警等等。

实时pid数量监控

为什么要监控实时的pid数量呢？因为目前的Linux内核对pid的隔离性支持是不完善的。还没有任何Linux发行版能做到针对pid按照容器粒度进行pid_max限制。

曾经发生过一个真实的案例是：某个用户写的程序有bug，创建的线程没有及时回收，容器中产生了大量的线程，最后在宿主机上都无法执行命令或者ssh登陆，报的错是“bash: fork: Cannot allocate memory”，但是此时通过free命令看到空闲的内存却是足够的。

为什么会这样呢？根本原因是内核中的pid_max(/proc/sys/kernel/pid_max)是全局共享的。当一个容器中的pid数目达到上限32768，会导致宿主机和其他容器无法创建新的进程。最新的4.3-rc1才支持对每个容器进行pid_max的限制。

内存使用监控

值得一提的是，我们发现cgroup提供的内存使用值是不准确的，比真实使用的内存值要低。因为内核memcg无法回收slab cache，也不对dirty cache量进行限制，所以很难估算容器真实的内存使用情况。曾经发生过统计的内存使用率一到70-80%，就发生OOM的情况。为此，我们调整了容

器内存的计算算法，根据经验值，将cache的40%算做rss，调整后的内存计算比之前精确了不少。

日志乱序

还有一个问题是跑Docker的宿主机内核日志中经常会产生字符乱序，这样会导致日志监控无法取到正确的关键字进行报警。

经过分析发现，这个跟我们在宿主机和Docker容器中都跑了rsyslogd有关。由于内核中只有一个log_buf缓冲区，所有printk打印的日志先放到这个缓冲区中，docker host以及container上的rsyslogd都会通过syslog从kernel的log_buf缓冲区中取日志，导致日志混乱。通过修改container里的rsyslog配置，只让宿主机去读kernel日志，就能解决这个问题。

隔离开关

平时我们的容器是严格隔离的，我们做的隔离包括CPU、内存和磁盘IO，网络IO等。但双十一的业务量可能是平时的十几倍或几十倍。我们为双十一做了不少开关，在压力大的情况下，我们可以为个别容器进行动态的CPU，内存等扩容或缩容，调整甚至放开磁盘iops限速和网络的TC限速。

健康监测

我们还开发了定期的健康监测，定期的扫描线上可能存在的潜在风险，真正做到提前发现问题，解决问题。

灾备和紧急故障处理

除了稳定性，灾备能力也是必须的，我们做了大量的灾备预案和技术准备。比如我们研究了不启动Docker Daemon的情况下，离线恢复Docker中数据的方法。具体来说，是用dmsetup create命令创建一个临时的dm设备，映射到Docker实例所用的dm设备号，通过mount这个临时设备，就可以恢复出原来的数据。

我们还支持Docker容器的冷迁移。通过管理平台的界面可以一键实现跨物理机的迁移。

与已有运维系统的对接

Docker集群必须能与现有的运维系统无缝对接，才能快速响应，真正做到秒级的弹性扩容/缩容。我们有统一的容器管理平台，实现对多个Docker集群的管理，从下发指令到完成容器的创建可以在7秒内完成。

性能优化

我们从系统层面也对docker做了大量的优化，比如针对磁盘IO的性能瓶颈，我们调优了像vm.dirty_expire_centisecs, vm.dirty_writeback_centisecs, vm.extra_free_kbytes这样的内核参数。还引入了Facebook的开源软件flashcache，将SSD作为cache，显著的提高docker容器的IO性能。

我们还通过合并镜像层次来优化docker pull镜像的时间。在docker pull时，每一层校验的耗时很长，通过减小层数，不仅大小变小，docker pull时间也大幅缩短。

镜像	文件层数	文件大小	docker pull时间
原镜像	13	1. 051GB	2m13
新镜像	1	674. 4MB	0m26

总结

总的来说，双11是对蘑菇街私有云的一次年终大考，对此我们已有了充分的准备。随着Docker集群部署的规模越来越大，我们还有很多技术难题有待解决，包括容器本身的隔离性问题，集群的弹性调度问题等等。同时我们也很关注Docker相关的开源软件Kubernetes、Mesos、Hyper、criu、runC的发展，未来将引入容器的热迁移，Docker Daemon的热升级等特性。如果大家想了解更多的话，可以关注我们的技术博客<http://mogu.io/>。蘑菇街期待你的加入，一起来建设大规模的Docker集群。

作者介绍

郭嘉，蘑菇街平台技术部架构师，虚拟化组负责人。2014年加入蘑菇街，目前主要专注于蘑菇街的私有云建设，关注Docker、KVM、OpenStack、Kubernetes等领域。

邮箱：guojia@mogujie.com。



扫描二维码下载Docker迷你书

唯品会11.11：峰值系统应对实践



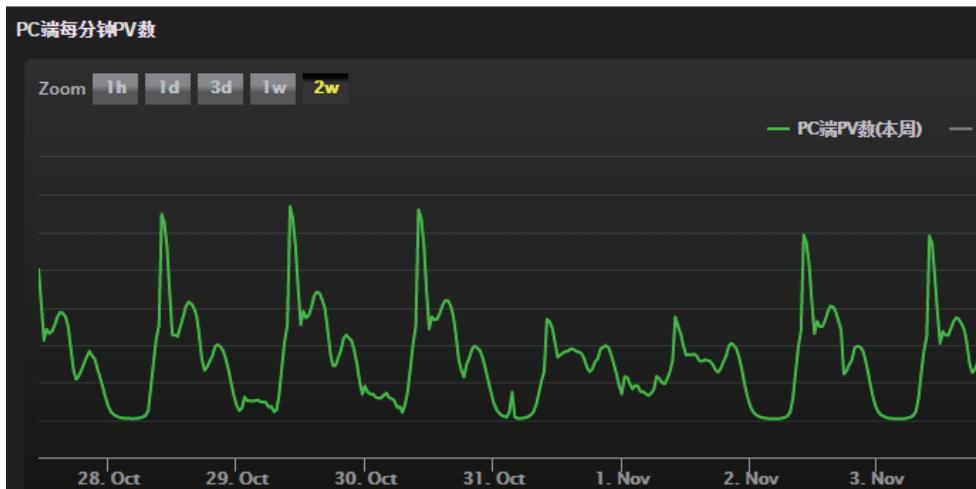
作者 张广平

区别于其他网购品牌唯品会定位是“一家专门做特卖的网站”，商业模式为“名牌折扣+限时抢购+正品保险”，即“闪购”（flash sales）模式。每天上新品，以低至1折的深度折扣及充满乐趣的限时抢购模式，为消费者提供一站式优质购物体验，

这种闪购限时特卖业务特点决定了网站随时都需要处理高并发、大流量的用户请求。大量买家在每次新的品牌档期上线后，大量涌入，抢购商品，造成网站承担大量流量。尤其碰到热门商品，网站并发访问剧增，会造成整个网站负载过重，响应延迟，严重时甚至会出现服务宕机的情况。

另外唯品会有众多的业务销售模式a，如自营销售模式、JIT、直发、海淘、O2O等等，这些业务销

售模式导致系统非常复杂。系统外部需要通过开放平台对接供应商系统和第三方物流系统。内部系统包括诸多系统，如供应商管理、商品选品、商品交易、支付系统、物流仓储、客服系统、商品配送等等。这些系统功能模块之间关联性非常强，逻辑扩展非常复杂，如何快速满足业务发展的需要，是一个非常迫切的问题。



为了保证系统在高并发、大流量访问下工作，并且使系统有较强的扩展性，我们的设计主要从以下几个方面展开：

- 系统模块有效切分
- 服务化解耦，集中服务治理
- 增加异步访问
- 多阶段缓存，降低后端压力
- 优化数据库访问
- 加强系统监控

系统模块有效切分

唯品会整个业务系统虽然已经拆分成几个相对独立的子系统如交易平台（B2C）、VIS、WMS、TMS、ODS、CS、EBS等，但是这些业务系统在实际运作中业务耦合严重。碰到新业务逻辑加入，

就需要每个模块做大量修改，各个开发团队之间为了业务逻辑放在那里争论不休，浪费了大量的时间，导致开发效率比较低。这主要由于模块划分不合理，导致模块之间边界不清楚。所以我们架构团队从整个系统角度出发，梳理整个流程，重新做系统定位，将不同业务子系统做物理分离，减少彼此之间的依赖，使各个子系统独立部署，出现问题后能快速采取措施，隔离出问题模块，将故障影响降到最低。

服务化解耦，集中服务治理

服务化设计已经被主流电商系统证明是一个切实可行的方向。通过SOA服务化改造，实现了服务使用者和服务提供者的分离，使系统间的服务解耦和系统内高内聚，大大简化了系统复杂性，具有更强的伸缩性和扩展性，满足了业务快速发展的需要。

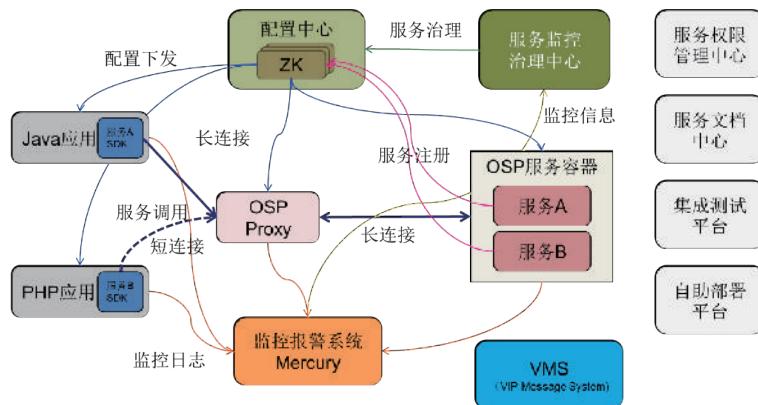
我们怎么有效管理这些服务呢？

Venus是唯品会自开发的一款基于Spring的Java开发框架，以降低开发的复杂度，提高开发人员的开发效率，提升代码质量，规范开发流程。

Venus框架涵盖了以下内容：

- 数据库访问层封装，支持分库、分表，连接池优化

- 缓存（Redis/Memcached）接口封装，连接池优化
- CRUD服务代码自动生成（包含数据库操作）
- OSP/REST服务调用接口封装及异步支持
- ValidateInternals
- 单元 / 集成测试模板代码自动生成
- 配置中心
- 中央文档中心



Venus生态体系

其中开放服务平台（OSP）的主要目标是提供服务化的核心远程调用机制。契约化的服务接口保证系统间的解耦清晰、干净；基于Thrift的通信和协议层确保系统的高性能；服务可以自动注册并被发现，易于部署；配合配置中心，服务配置

可以动态更新；客户端与治理逻辑的分离使服务接入得到极大简化；除此之外，OSP提供了丰富的服务治理能力，如路由、负载均衡、服务保护和优雅降级等，通过OSP，有效的实现了流量控制。

服务分流

首先OSP Proxy具有软负载的作用，系统不需要硬件负载均衡，可以将服务请求均衡地分配到不同服务主机上。另外OSP可以配置服务的路由，服务请求可以被分配到不同版本的服务中处理，这样很容易实现灰度发布。

服务限流

在系统流量达到极限时的情况，有自动熔断机制。熔断器是在服务或者周边环境（如网络）出现了异常后主动断开客户端后续的使用，从而

避免服务崩溃无法恢复。但是在后续时间熔断将使用小量请求尝试侦测服务是否已经恢复，如果恢复则将服务再次提供给客户端调用。熔断器的机制即保护了服务也减少了人工干预。相关的阀值都在是在配置中心中配置，并支持动态修改生效。限流一定要谨慎使用，要使用恰当的限流策略，区分正常访问和恶意请求，不能将正常的用户请求抹杀掉。如果无法区分是否是恶意请求，需要将应用分级，确保优先级最高的应用能被访问到，比如所有上线的商品信息。而对于下线的商品信息，可以根据请求容量作适当的限流。

Nginx Rate Limiter是一个自主开发的防刷工具，通过Nginx上的LUA脚本插件，实现在Nginx上对本机的HTTP访问进行限流控制的工具，以提高在促销等高业务量环境下保障系统稳定运行的能力。Nginx Rate Limiter通过RESTful API接口进行配置以及信息查看，可以对全局进行开关等配置，也可以针对指定URL分别添加多个限流配置，包括全局的限流。限流配置可以选择以下一种方式：

1. 最大访问请求速率，超出则丢弃请求；
2. 按比例丢弃请求。

服务降级

对于电商系统，为了保证用户体验，在资源有限的条件下，我们必须保证关键系统的稳定性。通过对不同业务级别定义不同的降级策略，对除核心主流程以外的功能，根据系统压力情况进行有策略的关闭，从而达到服务降级的目的，例如在线商品信息，我们必须保证优先访问，而对于下线的商品信息，我们可以容许在访问容量受限情况下，容许关闭下线商品详情页面的访问等。

增加异步访问

对于系统间实时性要求不高的操作，如果执行时比较耗时，可通过异步处理提高调用者性能，提高响应能力，尤其通过异步调用通知非主要流程，加快了系统主要业务流程的反应速度和性能，异步处理机制可起到缓冲的作用，被通知的下游系统可依据自身能力控制处理数据量，避免遭受超负荷的冲击，保证系统稳定运行，增加了系统可用性。

分布式异步消息队列服务器可在宕机后确保消息不丢失，异步系统有重试机制，从而提高系统

可用性、健壮性和扩展性。

在用户下单后，其他系统如物流、供应商系统、配送、财务等系统需要获取订单详情、订单状态，订单系统通过异步消息方式将订单的变化通知其它系统，异步调用实现系统间隔离解耦，上下游系统业务逻辑分离，下游系统只需要解析异步消息进行处理，不需要依赖上游系统的业务逻辑，从而降低了系统之间的依赖。即使下游系统出现异常，订单系统仍然能正常处理数据。

多阶段缓存，降低后端压力

1、动静分离，静态化

静态化可降低后端压力，一方面通过用户浏览器缓存静态资源，失效时间通过cache-control来控制。另外一方面通过CDN缓存，例如商品详情页面，为了提高缓存效率，可将商品详情页面伪静态化，将URL后缀显示为HTML，商品描述信息等静态信息在有用户访问情况下，缓存到靠近用户的CDN节点，另外为了提高CDN效率，提前将商品图片推送到CDN。其它商品动态数据可动态加载，如商品运营信息、商品库存、尺码表等，从而降低了避免不必要的后台访问。

2、分布式缓存

引入分布式缓存，对缓存数据服务节点做统一集中管理，可支持缓存集群弹性扩展，通过动态增加或减少节点应对变化的数据访问负载，通过冗余机制实现高可用性，无单点失效，不会因服务器故障而导致缓存服务中断或数据丢失。应用端使用统一的API接口访问缓存服务器。

通过分布式缓存，可做应用对象缓存、数据库缓存、会话状态及应用横向扩展时的状态数据缓存。

3、巧用应用服务器本地缓存

分布式缓存虽然有效解决了访问压力，但由于缓存服务器分布在不同网络端、不同数据中心部署，随着访问量增大将导致I/O和带宽瓶颈。为此可将那些基本不修改的配置数据、全局数据可以在应用服务器本地缓存，减少对后端缓存服务器实例的峰值冲击。本地缓存需要谨慎使用，如果大量使用本地缓存，可能会导致相同的数据被不同的节点存储多份，对内存资源造成较大的浪费。

使用缓存对提高系统性能有很多好处，但是不合理使用缓存非但不能提高系统的性能，反而成为系统的累赘，甚至影响系统运作，产生很大的风险。对于频繁修改的数据、没有热点的访问数据、数据一致性要求非常高的数据，不建议使用缓存。

优化数据库访问

在高并发大数据量的访问情况下，数据库存取瓶颈一直是个令人头疼的问题。如果数据库访问出现性能问题，整个系统将受到影响。

为此需要优化数据库访问，从以下几个方面解决高并发问题。

优化复杂查询，提高数据库查询效率，找出关键模块的数据库慢查询进行优化。例如减少数据库表之间的Join、重构数据库表相关索引、对where子句进行优化等。

保证在实现功能的基础上，尽量减少对数据库的访问次数；通过查询参数，尽量减少对表的访问行数，最小化结果集，从而减轻网络负担；能够分开的操作尽量分开处理，提高每次的响应速度，查询时用到几列就选择几列，降低数据库访问I/O负载压力。

基于电商系统读写比很大的特性，采用读写分离技术，通过一主多从，写操作只发生在主表，多操作发生在从表上，可以大大缓解对主数据库的访问压力。

对业务和系统的细分，对数据库表进行垂直拆分。将数据库想象成由很多个“数据块”（表）组成，垂直地将这些“数据块”切开，然后把它们分散到多台数据库主机上面，从而可以分散单位时间内数据库访问压力。垂直切分的依据原则是：将业务紧密，表间关联密切的表划分在一起。

垂直切分后，需要对分区后表的数据量和增速进一步分析，以确定是否需要进行水平切分。对于核心数据如订单，采取水平分区的方式，通过一致性哈希算法，利用用户ID把订单数据均匀的分配在各个数据库分区上，在应用系统查询时，以用户ID调用哈希算法找到对应数据库分区，从而将高峰期的数据库访问压力分散到不同数据库分区上，从而实现数据库的线性伸缩，极大提升数据库的承载能力。

借助于分布式缓存，缓存提供了远大于数据库访问的性能。当某一应用要读取数据时，会首先从缓存中查找需要的数据，如果找到了则直接执行，找不到的话则从数据库中找。在设计时，需要防止缓存失效带来的缓存穿透压力。

容许一定程度的数据冗余，对于关键模块，为了防止对其他模块的依赖而影响当前模块的性能和可靠性，可适度保存其他模块的关键数据，减少由于访问其他模块服务带来的系统损耗和可靠性压力。

使用NoSQL数据库对海量大数据进行存储和处理。

加强系统监控

业务系统通常由众多分布式组件构成，这些组件由web类型组件，RPC服务化类型组件，缓存组件，消息组件和数据库组件。一个通过浏览器或移动客户端的前端请求到达后台系统后，会经过很多个业务组件和系统组件，并且留下足迹和相关日志信息。但这些分散在每个业务组件和主机下的日志信息不利于问题排查和定位问题的根本原因。这种监控场景正是应用性能监控系统的用武之地，应用性能监控系统收集，汇总并分析日志信息达到有效监控系统性能和问题的效果。通过监控信息，可以清晰地定位问题发生原因，让开发人员及时修复问题。

唯品会有三级监控，系统/网络层面监控、应用层面监控和业务层面监控。

系统/网络层面监控，主要是对下列指标进行监控：服务器指标，如CPU、内存、磁盘、流量、TCP连接数等；数据库指标，如QPS、主从复制延时、进程、慢查询等。

业务层面监控，通过两种方法，第一种在指定页面做埋点，第二种方法从业务系统的数据库中，将需要监控的数据抽取出来，做必要的分析处理，存入运维自己维护的数据库中；然后通过浏览器页面，展示监控数据，页面同时提供各种时间维度上的筛选、汇总。对一些业务的关键指标如PV、UV、商品展示、登录/注册、转化率、购物车、订单数量、支付量、发货量和各仓订单数据。可自定义告警范围，通知相关人以便做出响应。



Telescope

业务层面监控，如PV，UV，订单量，购物车等，并产生和上报业务层告警



Mercury

应用层面监控，如应用错误数，异常数，响应时间，访问量和调用链等，并产生和上报应用层告警



Zabbix

系统/网络层面监控，如CPU，内存，磁盘，流量，tcp连接数等，并产生和上报系统层面告警

应用层面监控系统Mercury，是唯品会独立研发的应用性能监控平台。通过在应用程序中植入探针逻辑来实现对应用代码、关系数据库、缓存系统的实时监控。Mercury通过收集日志、上报日志的方式即时获取相关性能指标并进行智能分析，及时发现分布式应用系统的性能问题以及异常和错误，为系统解决性能和程序问题提供方便可靠的依据。同时通过Mercury数据展现平台，用户可轻松便捷的获取应用360度监控信息。

在唯品会体系中，Mercury提供的主要功能包括：

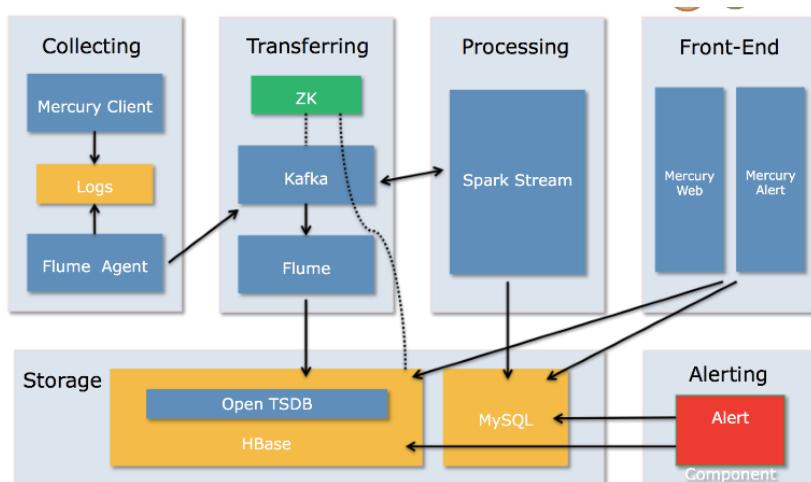
- 定位慢调用：包括慢Web服务（包括Restful Web服务），慢OSP服务，慢SQL；
- 定位错误：包括4XX, 5XX, OSP Error；
- 定位异常：包括Error Exception, Fatal Exception展现依赖和拓扑：域拓扑，服务拓扑，trace拓扑；
- Trace调用链：将端到端的调用，以及附加在这次调用的上下文信息，异常日志信息，每一个调用点的耗时都呈现给用户；
- 应用告警：根据运维设定的告警规则，扫描指标数据，如违反告警规则，则将告警信息上报到唯品会中央告警平台Mercury架构主要分以下几大模块。

日志由客户端传输到服务端后，分二条路径。第一条路径，裸日志（Trace log / Exception log）通过kafka，再通过flume直接落地到HBase。这些裸日志用来查询trace调用链信息和异常日志。另一条路径，日志信息通过kafka直接送到spark stream，通过spark分析后计算后，产生data points性能指标数据，再通过flume写入OpenTSDB。整个传输过程最重要的就是保证数据消费不要丢失和积压。

一旦满足通过运维人员事先配置的告警规则，告警模块可触发告警动作。告警信息可在第一时间将故障上报给中央告警平台。

结语

以上几点是我们对高并发系统的一些体会，我们在不断改进系统，为将唯品会做大做强持续努力，也希望通过本次分享给大家带来一定收获。



蚂蚁金服11.11：支付宝和蚂蚁花呗的技术架构及实践



作者 贺岩

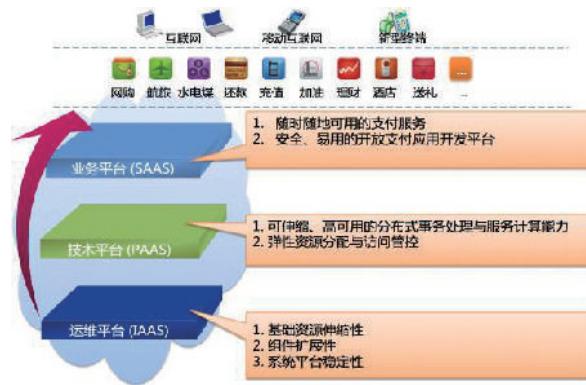
每年“双11”都是一场电商盛会，消费者狂欢日。今年双11的意义尤为重大，它已经发展成为全世界电商和消费者都参与进来的盛宴。而对技术人员来说，双十一无疑已经成为一场大考，考量的角度是整体架构、基础中间件、运维工具、人员等。

一次成功的大促准备不光是针对活动本身对系统和架构做的优化措施，比如：流量控制，缓存策略，依赖管控，性能优化……更是与长时间的技术积累和打磨分不开。下面我将简单介绍支付宝的整体架构，让大家有个初步认识，然后会以本次在大促中大放异彩的“蚂蚁花呗”为例，大致介绍一个新业务是如何从头开始准备大促的。

因为涉及的内容要深入下去是足够写一个系列介绍，本文只能提纲挈领的让大家有个初步认识，后续可能会对大家感兴趣的专项内容进行深入分享。

架构

支付宝的架构设计上应该考虑到互联网金融业的特殊性，比如要求更高的业务连续性，更好的高扩展性，更快速的支持新业务发展等特点。目前其架构如下：



整个平台被分成了三个层：

1. 运维平台 (IAAS)：主要提供基础资源的可伸缩性，比如网络、存储、数据库、虚拟化、IDC等，保证底层系统平台的稳定性；
2. 技术平台 (PAAS)：主要提供可伸缩、高可用的分布式事务处理和服务计算能力，能够做到弹性资源的分配和访问控制，提供一套基础的中间件运行环境，屏蔽底层资源的复杂性；
3. 业务平台 (SAAS)：提供随时随地高可用的支付服务，并且提供一个安全易用的开放支付应用开发平台。

架构特性

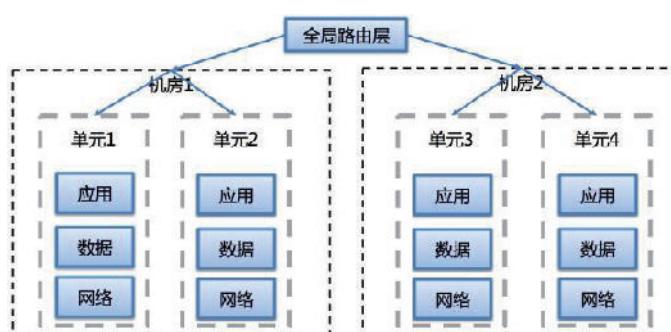
逻辑数据中心架构

在双十一大促当天业务量年年翻番的情况下，支付宝面临的考验也越来越大：系统的容量越来越大，服务器、网络、数据库、机房都随之扩展，

这带来了一些比较大的问题，比如系统规模越来越大，系统的复杂度越来越高，以前按照点的伸缩性架构无法满足要求，需要我们有一套整体性的可伸缩方案，可以按照一个单元的维度进行扩展。能够提供支持异地伸缩的能力，提供N+1的灾备方案，提供整体性的故障恢复体系。基于以上几个需求，我们提出了逻辑数据中心架构，核心思想是把数据水平拆分的思路向上层提到接入层、终端，从接入层开始把系统分成多个单元，单元有几个特性：

1. 每个单元对外是封闭的，包括系统间交换各类存储的访问；
2. 每个单元的实时数据是独立的，不共享。而会员或配置类对延时性要求不高的数据可共享；
3. 单元之间的通信统一管控，尽量走异步化消息。同步消息走单元代理方案。

下面是支付宝逻辑机房架构的概念图：



这套架构解决了几个关键问题：

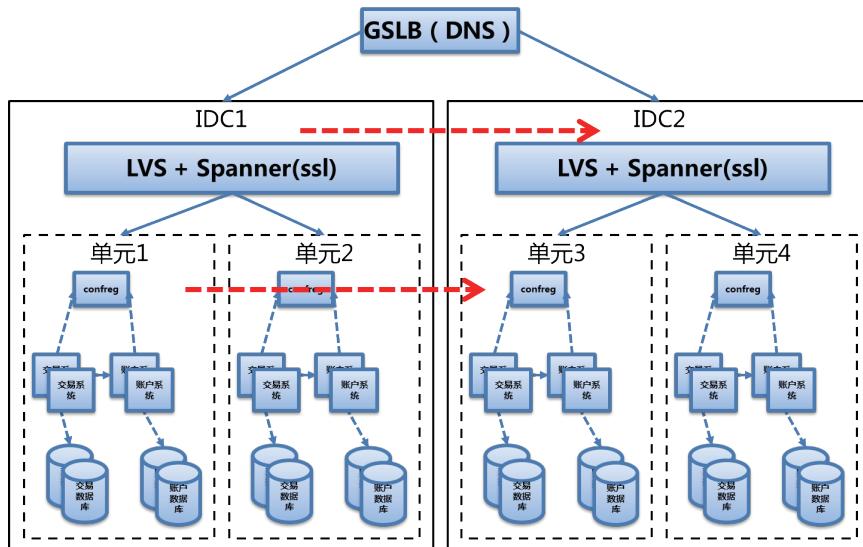
1. 由于尽量减少了跨单元交互和使用异步化，使得异地部署成为可能。整个系统的水平可伸缩性大大提高，不再依赖同城IDC；
2. 可以实现N+1的异地灾备策略，大大缩减灾备成本，同时确保灾备设施真实可用；
3. 整个系统已无单点存在，大大提升了整体的高可用性；同城和异地部署的多个单元可用作互备的容灾设施，通过运维管控平台进行快速切换，有机会实现100%的持续可用率；
4. 该架构下业务级别的流量入口和出口形成了统一的可管控、可路由的控制点，整体系统的可管控能力得到很大提升。基于该架构，线上压测、流量管控、灰度发布等以前难以实现的运维管控模式，现在能够十分轻松地实现。

目前新架构的同城主体框架在2013年已经完成，并且顺利的面对了双十一的考验，让整套架构的落地工作得到了很好的证明。

在2015年完成了基于逻辑机房，异地部署的“异地多活”的架构落地。“异地多活”架构是指，基于逻辑机房扩展能力，在不同的地域IDC部署逻辑机房，并且每个逻辑机房都是“活”的，真正承接线上业务，在发生故障的时候可以快速进行逻辑机房之间的快速切换。

这比传统的“两地三中心”架构有更好的业务连续性保障。在“异地多活”的架构下，一个IDC对应的故障容灾IDC是一个“活”的IDC，平时就承载着正常线上业务，保证其稳定性和业务的正确性是一直被确保的。

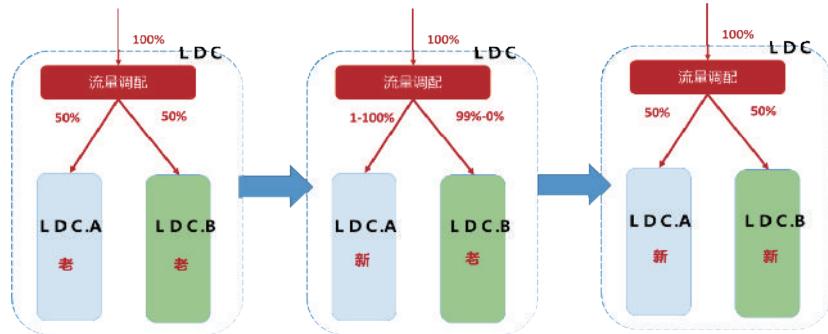
以下是支付宝“异地多活”架构示意图：



除了更好的故障应急能力之外，基于逻辑机房我们又具备的“蓝绿发布”或者说“灰度发布”的验证能力。我们把单个逻辑机房（后续简称LDC）内部又分成A、B两个逻辑机房，A、B机房在功能上完全对等。日常情况下，调用请求按照对

等概率随机路由到A或B。当开启蓝绿模式时，上层路由组件会调整路由计算策略，隔离A与B之间的调用，A组内应用只能相互访问，而不会访问B组。

然后进行蓝绿发布流程大致如下：



Step1. 发布前，将“蓝”流量调至0%，对“蓝”的所有应用整体无序分2组发布。

Step2. “蓝”引流1%观察，如无异常，逐步上调分流比例至100%。

Step3. “绿”流量为0%，对“绿”所有应用整体无序分2组发布。

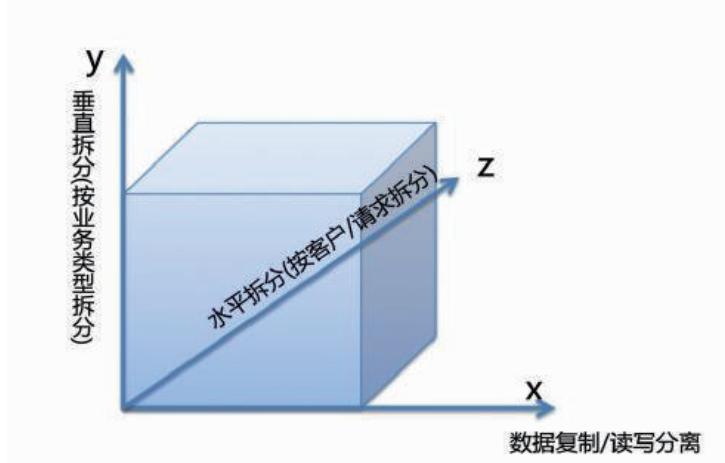
Step4. 恢复日常运行状态，蓝、绿单元各承担线上50%的业务流量。

支付宝已经是全球最大的OLTP处理器之一，对事务的敏感使支付宝的数据架构有别于其他的互联网公司，却继承了互联网公司特有的巨大用户量，最主要的是支付宝对交易的成本比传统金融公司更敏感，所以支付宝数据架构发展，就是一部低成本、线性可伸缩、分布式的数据架构演变史。

现在支付宝的数据架构已经从集中式的小型机和高端存储升级到了分布式PC服务解决方案，整体数据架构的解决方案尽量做到无厂商依赖，并且标准化。

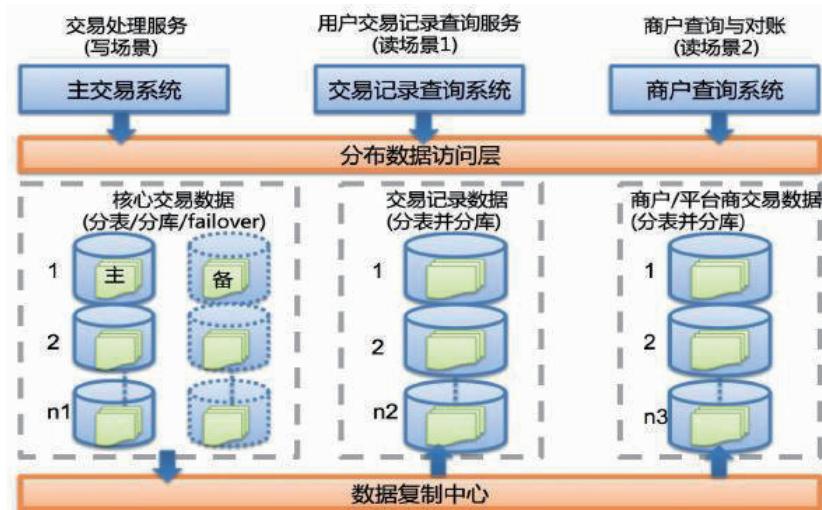
分布式数据架构

支付宝在2015年双十一当天的高峰期间处理支付峰值8.59万笔/秒，已经是国际第一大系统支付。



1. 按照业务类型进行垂直拆分
2. 按照客户请求进行水平拆分（也就是常说的数据的sharding策略）
3. 对于读远远大于写的数据进行读写分离和数据复制处理

下图是支付宝内部交易数据的可伸缩性设计：



交易系统的数据主要分为三个大数据库集群：

1. 主交易数据库集群，每一笔交易创建和状态的修改首先在这里完成，产生的变更再通过可靠数据复制中心复制到其他两个数据库集群：消费记录数据库集群、商户查询数据库集群。该数据库集群的数据被水平拆分成多份，为了同时保证可伸缩性和高可靠性，每一个节点都会有与之对应的备用节点和failover节点，在出现故障的时候可以在秒级内切换到failover节点。
2. 消费记录数据库集群，提供消费者更好的用户体验和需求；
3. 商户查询数据库集群，提供商户更好的用户体验和需求；

对于分拆出来的各个数据节点，为了保证对上层应用系统的透明，我们研发一套数据中间产品来保证交易数据做到弹性扩容。

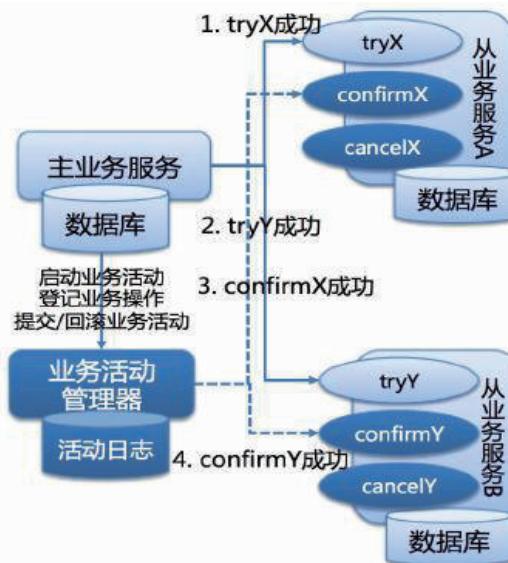
数据的可靠性

分布式数据架构下，在保证事务原有的ACID（原子性、一致性、隔离性、持久性）特性的基础上，还要保证高可用和可伸缩性，挑战非常大。试想你同时支付了两笔资金，这两笔资金的事务如果在分布式环境下相互影响，在其中一笔交易资金回滚的情况下，还会影响另外一笔是多么不能接受的情况。

根据CAP和BASE原则，再结合支付宝系统的特点，我们设计了一套基于服务层面的分布式事务框架，他支持两阶段提交协议，但是做了很多的优化，在保证事务的ACID原则的前提下，确保事务的最终一致性。我们叫做“柔性事物”策略。原理如下：



以下是分布式事务框架的流程图：



实现：

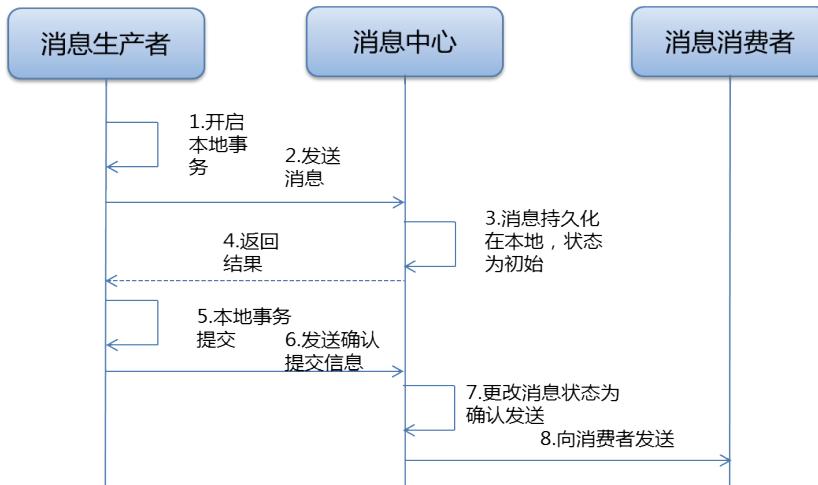
1. 一个完整的业务活动由一个主业务服务与若干从业务服务组成。
2. 主业务服务负责发起并完成整个业务活动。
3. 从业务服务提供TCC型业务操作。
4. 业务活动管理器控制业务活动的一致性，它登记业务活动中的操作，并在活动提交时

确认所有的两阶段事务的confirm操作，在业务活动取消时调用所有两阶段事务的cancel操作。”

与2PC协议比较：

1. 没有单独的Prepare阶段，降低协议成本；
2. 系统故障容忍度高，恢复简单。

其中关键组件异步可靠消息策略如下：



其中一些关键设计点：

1. 若在第2、3、4步出现故障，业务系统自行决定回滚还是另起补偿机制；若在第6、7步出现异常，消息中心需要回查生产者；若在第8步出现异常，消息中心需要重试。第6步的确认消息由消息中心组件封装，应用系统无需感知。
2. 此套机制保障了消息数据的完整性，进而保障了与通过异步可靠消息通讯的系统数据最终一致性。
3. 某些业务的前置检查，需要消息中心提供指定条件回查机制。

技术体系经过不断演进、已经完全依托于蚂蚁金服的金融云架构。

在2014年12月，蚂蚁花呗团队完成业务系统优化，按照标准将系统架设到了金融云上，依次对接了渠道层、业务层、核心平台层、数据层，使得用户对蚂蚁花呗在营销、下单和支付整个过程中体验统一。

2015年4月，蚂蚁花呗系统同步金融云的单元化的建设，即LDC，使得数据和应用走向异地成为了现实，具备了较好的扩展性和流量管控能力。在可用性方面，与金融云账务体系深度结合，借用账务系统的failover能力，使得蚂蚁花呗通过低成本改造就具备了同城灾备、异地灾备等高可用能力。任何一个单元的数据库出了问题、能够快速进行容灾切换、不会影响这个单元的用户进行蚂蚁花呗支付。在稳定性方面，借助于云客户平台的高稳定性的能力，将蚂蚁花呗客户签约形成的合约数据迁移进去，并预先写入云客户平台的缓存中，在大促高峰期缓存的命中率达到100%。同时，结合全链路压测平台，对蚂蚁花呗进行了能力摸高和持续的稳定性测试，发现系统的性能点反复进行优化，使得大促当天系统平稳运行。在之前的架构中，系统的秒级处理能力无法有效衡量，通过简单的引流压测无法得到更加准确、可信的数据。立足于金融云，支撑蚂蚁花呗业务发展的顺畅。

蚂蚁花呗

蚂蚁花呗是今年增加的一个新支付工具，“确认收货后、下月还”的支付体验受到了越来越多的消费者信赖。跟余额和余额宝一样，蚂蚁花呗避开了银行间的交易链路，最大限度避免支付时的拥堵。据官方数据披露，在今天的双十一大促中，蚂蚁花呗支付成功率达到99.99%、平均每笔支付耗时0.035秒，和各大银行渠道一起确保了支付的顺畅。

蚂蚁花呗距今发展不到一年，但发展速度非常快。从上线初期的10笔/秒的支付量发展到双十一当天峰值2.1w笔/秒。支撑蚂蚁花呗业务发展的

系统很快通过全链路压测得到了每秒处理4w笔支付的稳定能力。

蚂蚁花呗业务中最为关键的一环在于买家授信和支付风险的控制。从买家下单的那一刻开始，后台便开始对虚假交易、限额限次、套现、支用风险等风险模型进行并行计算，这些模型最终将在20ms以内完成对仅百亿数据的计算和判定，能够在用户到达收银台前确定这笔交易是否存在潜在风险。

为了保证蚂蚁花呗双11期间的授信资金充足，在金融云体系下搭建了机构资产中心，对接支付清算平台，将表内的信贷资产打包形成一个一定期限的资产池，并以这个资产池为基础，发行可交易证券进行融资，即通过资产转让的方式获得充足资金，通过这一创新确保了用户能够通过花呗服务顺利完成交易，并分流对银行渠道的压力。通过资产证券化运作，不仅帮助100多万小微企业实现融资，也支撑了蚂蚁花呗用户的消费信贷需求。蚂蚁小贷的资产证券化业务平台可达到每小时过亿笔、总规模数十亿元级别的资产转让。

总结

经过这么多年的高可用架构和大促的准备工作，蚂蚁金融技术团队可以做到“先胜而后求战”，主要分为三方面技术积累：“谋”，“器”，“将”。

“谋”就是整体的架构设计方案和策略；

“器”就是支持技术工作的各种基础中间件和基础组件；

“将”就是通过实践锻炼成长起来的技术人员。

纵观现在各种架构分享，大家喜欢谈“谋”的方面较多，各种架构设计方案优化策略分享，但实际最后多是两种情况：“吹的牛X根本没被证实过”（各种框架能力根本没经过实际考验，只是一纸空谈），“吹过的牛X一经实际考验就破了”（说的设计理念很好，但是一遇到实际的大业务的冲击系统就挂了），最后能成功的少之又少。这些说明虽然架构上的“心灵鸡汤”和“成功学”技术人员都已经熟的不行，但是发现一到实践其实根本不是那么

回事。从此可以看出，其实最后起决定作用的不是“谋”方面的理论层面的分析设计，最重要的是落地“器”和“将”的层面。有过硬高稳定性的各种基础设施工具的和身经百战被“虐了千百次”的技术人员的支撑才是最后取胜的关键。而这两个层面的问题是不能通过分享学到的，是要通过日积月累的，无数流血流泪趟雷中招锻炼出来的，没有近路可抄。

而目前从业务和市场的发展形势来看，往往就是需要技术在某个特定时间有个质的能力的提升和飞跃，不会给你太多的准备技术架构提升的时间，在技术积累和人员储备都不足的时候，如何构建平台能力，把更多的精力放在业务相关的开发任务中，是每个技术团队的希望得到的能力。

过去我们是通过某个开源或者商业组件来实现技术共享得到快速解决谋发展技术的能力的，但是随着业务复杂性，专业性，规模的逐步变大，这种方式的缺点也是显而易见的：

1. 很多组件根本无法满足大并发场景下的各种技术指标；
2. 随着业务的复杂和专业性的提高，没有可以直接使用的开源组件；
3. “人”本身的经验和能力是无法传递的。

所以现在我们通过“云”分享的技术和业务的能力的方式也发展的越来越快，这就我们刚才介绍的“蚂蚁花呗”技术用几个月的时间快速的成功的达到“从上线初期的10笔/秒的支付量发展到双十一当天峰值2.1w笔/秒，快速走完了别人走了几年都可能达不到的能力。类似的例子还有大家熟知的“余额宝”系统。

这些都是建立在原来蚂蚁金服用了10年打磨的基础组件和技术人员经验的云服务上的，通过目前基于这种能力，我们目前可以快速给内部和外部的客户组建，高可用、安全、高效、合规的金融云服务架构下的系统。