



# 架构师

ARCHITECT

QCon 特刊

## ■ 关于QCon4Kids QCon4KIDS

QCon4Kids 代码捣鼓日 /The Tinkerthon Day, 是由极客邦联合网易卡搭编程共同发起的亲子少儿编程活动, 旨在通过 Scratch 帮助孩子培养良好的解决问题能力和探索尝试的习惯。

### 什么是代码捣鼓日?



什么是 Hackathon 大家都不陌生, 很多技术公司都会进行“黑客马拉松”。我们的孩子或许还不够 hack, 但 tinkering 可是他们与生俱来的能力。

Tinkering, 捣鼓是一种态度。通过“试验 – 找出错误 – 调整 – 再试验……”这样不断迭代的过程, 创造出一些独特的东西, 解决实际问题。当一群 tinker 凑到一起, 共同捣鼓, 就是一场会诞生无数创意作品的 Tinkerthon 了。



[了解更多](#)

### “代码捣鼓日”活动安排

- 时间: 10月20日 09: 00–17: 00
- 地点: 大宁会议中心



## ■ 关于 QCon4Educators QCon4Educators

QCon4Educators 是给中小学信息科技老师的论坛。极客邦邀请教育工作者一起来参与 QCon 盛会, 期待老师们能从技术展台和百余场讲座中感受到技术人改变世界的雄心。老师们和工程师们将在这个下午一起探讨信息科技课的 **WHAT、WHY** 与 **HOW**, 共同推进青少年计算机科学教育的发展。

### 活动安排

- 时间: 10月20日 13: 00–15: 30
- 地点: 大宁会议中心



嘉宾演讲

ET 结桥社成立仪式

主题讨论 1:

主题讨论 2:

面向未来的计算机科学教育

通过编程教学培养计算思维

## ■ 关于结桥社 结桥社

2018 年 8 月 14 日, TGO 鲲鹏会发起了“ET 之桥”的公益项目。我们要在老师和工程师之间架起一座桥梁, 带孩子们去认识这个被计算机改变了的世界, 教孩子们通过编程去表达与创造。

# 卷首语

黄丹

近期，CNCF（Cloud Native Computing Foundation）用中文开展了关于云原生技术的调研。该项调查每年进行两次，以洞察各技术社区的发展，更为清晰地了解云原生技术的使用情况。通过调查发现，在生产环境中，超过 49% 的受访者使用 Kubernetes。作为 Docker 生态圈中重要一员，无论是公有云还是私有云甚至混合云，Kubernetes 在容器管理框架环境中无处不在。

今年 7 月份，Istio 1.0 版本被官宣，距离最初的 0.1 版本发布已经过了一年多时间。我们看到，这个开源平台的 1.0 正式版旨在简化“已部署服务网络的创建，借助负载均衡、服务到服务的身份验证、监控等，而不需要修改任何服务代码”。该版本的主要新特性包括跨集群 Mesh 支持、细粒度流量控制以及在一个 Mesh 中增量推出 Mutual TLS 的能力。正是 Istio 的出现使“Service Mesh”这一概念开始流行起来，很多企业正在采用云原生应用程序架构，Service Mesh 俨然已经成为云原生技术栈里的一个关键组件。它提供了一种透明的、与编程语言无关的方式，使网络配置、安全配置以及遥测等操作能够灵活而简便地实现自动化。当然网上有很多资料在介绍 Service Mesh，但 Service Mesh 的本质是什么？如何更好地应用？估计并不容易通过简单的一句话理解清楚。这次 QCon 上海站，我们挖掘了一些 Service Mesh 大规模落地探索案例，总结了 Service Mesh 落地过程中会遇到的常见问题，以及平衡架构和性能之间的关系的方法，希望能更好地帮助你结合公司的业务规模和发展阶段去考察这一新技术对于公司的价值，带给你一些启发与灵感。

金秋十月，收获的季节。有“诗豪”之称的唐代文学家刘禹锡在《秋词》中以“自古逢秋悲寂寥，我言秋日胜春朝”热情讴歌了秋天的美好。愿正值美好秋天参加 QCon 的你，也能收获前沿技术思路和解决方案。

# 目录

05 Apache Kylin在卷皮网大数据平台的运用

09 后Kubernetes时代的微服务

12 2018年《DevOps 现状报告》

18 闲鱼基于Flutter的移动端跨平台应用实践

32 聊聊工程师的影响力

36 SOA旅程：从了解业务到敏捷架构

45 AIOps实践思考：AIOps如何与APM结合？

本期主编 徐 川

架构师特刊 流程编辑 丁晓昀

发行人 霍泰稳

QCon是由InfoQ主办的综合性技术盛会，每年在伦敦、北京、东京、纽约、圣保罗、上海、旧金山召开。自2007年3月份开始举办以来，已经有超万名有多年从业经验的技术人员参加过QCon大会。QCon内容源于实践并面向社区，演讲嘉宾依据热点话题，面向5年以上工作经验的技术团队负责人、架构师、工程总监、开发人员分享技术创新和实践。

# 卷皮OLAP平台进化史

## Apache Kylin在卷皮网大数据平台的运用

作者 许湘楠



**导读：**“卷皮网”是一家专注高性价比商品的移动电商，日活跃高达 1000 多万，随着卷皮网的快速发展，数据规模快速增长，集群数据存储量成指数倍增大，服务器规模达到 100+ 台，与此同时公司的运营成员急剧增加，数据需求也随着业务的发展落地不断增长，如统计分析、运营报表、取数需求任务日益增大。为了节省取数工作的时间和人员开支，及时响应运营等部门同学数据需求的快速响应，于是开发了以自助数据分析为目标的 OLAP 平台。本文将详解 Apache Kylin 在卷皮网大数据平台的运用。

### 前言

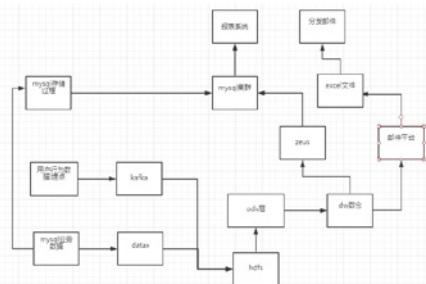
在开始案例分享前，先简单介绍一下“卷皮网”以及“卷皮网”的大数据团队“卷皮网”是一家专注

高性价比商品的移动电商，日活跃高达 1000 多万“卷皮网”的大数据团队规模在 40 人左右，主要负责公司的底层数据仓库建设、OLAP 平台、报表系统等数据可视化工具，以及数据挖掘在搜索排序推荐上的应用、爬虫物流平台的建设、鹰眼风控系统、拨云日志系统等。

随着卷皮网的快速发展，数据规模快速增长，集群数据存储量成指数倍增大，服务器规模达到 100+ 台，与此同时公司的运营成员急剧增加，数据需求也随着业务的发展落地不断增长，如统计分析、运营报表、取数需求任务日益增大。为了节省取数工作的时间和人员开支，及时响应运营等部门同学数据需求的快速响应，我们于是开发了以自助数据分析为目标的 OLAP 平台。随着公司业务的日益扩增，平台经历了如下发展过程。

## 早期的 ROLAP

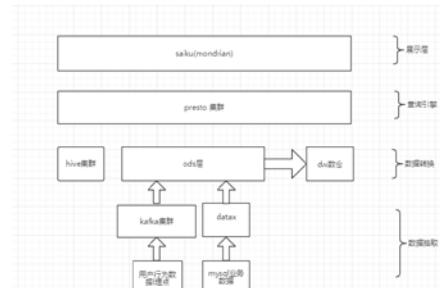
起初，数据规模较小，业务线比较简单，而且需求比较碎，故主要采取如下 ROLAP 引擎支撑：



- 具体流程：通过埋点采集用户行为数据，通过 Datax 和 Otter 同步数据到 Hive 集群和 MySQL 集群，数据开发工程师通过 Etl 脚本 (Hive 脚本和 MySQL 存储过程) 两种方式将最终结果数据落地到 MySQL 数据库，最终呈现给业务方使用，还有一部分灵活定制的是通过邮件平台每日生成 Excel 附件，邮件推送给业务方

## 以 Presto+Mondrian 为核心的 MOLAP 平台

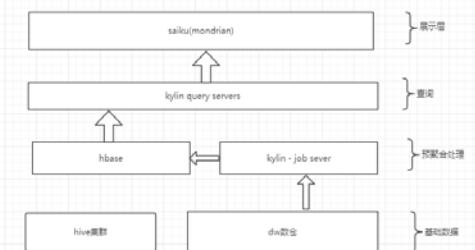
随着数据规模的增长和需求的增多，瓶颈逐渐显现。每个需求都要开发数据脚本，维度增加，开发周期拉长，同时需要耗费更多的人力，无法快速产出数据和响应需求变化。我们采用了 Saiku+Mondrian+Presto+Hive 的技术架构，通过分隔不同的业务线，最终生成若干个 Cube，提供给运营的同学使用，基本满足了业务方 90% 的数据需求。



## 使用 Kylin 解决超大规模数据分析

### Kylin v1.6

由于 Presto 是在线运算执行查询的，在日增上亿数据查询的时候，表现极为吃力。于是我们于 2016 年 8 月份开始引入 Apache Kylin（以下简称 Kylin），将用户行为数据等超大规模数据迁移到 Kylin 上，同时大大缓解 Presto 集群的压力。



由于 Kylin 的基本原理是通过预算实现空间换时间，Presto 需要在线查询源数据，所以 Kylin 的性能远远好于 Presto，Kylin 和 Presto 的查询性能对比。

	Kylin	Presto
节点数量	2 个	23 个
每日数据量	1 亿+	3000 万
平均查询时间	1 秒以内	10 秒+

该版本只支持星型模式，在 MR 上进行构建 Cube。起初我们根据业务线设计 Cube，其中最大的一个 Cube，维表 20+，其中包含若干高基数维，我们在预聚合的时候发现该 Cube 处理时间非常长，甚至造成内存溢出。于是我们对 Cube 进行了优化，以下是用到的一些优化手段。

- 我们将该 Cube 根据业务细分成若干个 Cube，同时对高基数维度做了优化。
- 使用了 Cube 构建的高级设置。这些高级设置包括聚合组、联合维度、层级维度和必要维度等。基于这些设置，我们对拆分后的 Cube 进行了进一步的优化。

	优化前	优化后
Cube 数量	1	3 个
维度数量	21	平均 12
最大维表基数	2000 万+	300 万+
预聚合时间	280min	平均 100min
膨胀率	977	123
存储空间	1.3T	150g

### (1) Mandatory Dimension

一般设置查询时经常使用的维度，我们使用了日期作为必需维度

### (2) Hierarchy Dimension

如果维度关系有一定层级性、基数由小到大的情况可以使用层级维度。比如年月日，省市区，一级类目二级类目三级类目等等

### (3) Joint Dimension

如果维度之间是同时出现的关系，即查询阶段，绝大部分情况都是同时出现的。可以使用联合维度。

根据高级配置后，虽然牺牲了部分查询的查询性能，但是极大的优化了预聚合的性能。以下是优化之后的性能指标：

	优化前	优化后
预聚合时间	133min	50min
膨胀率	166	8
存储空间	150g	10g

## Apache Kylin v2.x 版本升级

2017 年 4 月 30 号 Kylin v2.0 版本发布，不到三个月的时间，v2.1 版本正式发布。这两个版本主要有使用 Spark 做预聚合，支持雪花模型等新特性，对于我们解决 OLAP 预聚合慢的需求可以提供更多支持，并解决老版本的 Cube 构建时长、构建不稳定等问题。以下是 v1.6 版本和 v2.1 版本的一个对比：

### 场景 1：新版本 MR 构建性能对比

版本	时长(分钟)	产出数据量
1.6	50+	80g
2.1	15+	3g

事实事导入 2kw(2g) 测试数据，使用 19 个维度（维度基数在 1w 以下），6 个 Count Distinct(Bitmap)，8 个普通指标，聚合方式 : $2^{10}$  （无 Join Dimension，无 Mandatory Dimensions），以 MR 引擎进行构建。

### 场景 2：Spark 构建性能以及 Join Dimension

版本	引擎	时长(分钟)	产出数据量
1.6	MR	24+	0.2g
2.1	MR	8+	0.05g
2.1	SPARK	5+	0.05g

事实事导入 1kw(1.4g) 测试数据，使用 16 个维度（维度基数在 1w 以下），4 个普通指标，聚合方式： $(2^3 + 2^4 + 2)$  （有 Join Dimension，有 Mandatory Dimensions），以 Spark 和 MR 引擎分别进行构建。

### 场景 3：维表数量对性能的影响

版本	时长(分钟)	产出数据量
1.6	280+	15g
2.1	40+	1.5g

事实事导入 3kw(3g) 测试数据，使用 15 个维表，30+ 维度（维度基数在 1w 以下），12 个 Count Distinct(bitmap), 8 个普通指标，聚合方式： $2^5 + 2^5 + 2^5$  （有 Join Dimension，无 Mandatory Dimensions），以 MR 引擎进行构建。

由于这两个版本的 MR 构建性能差异较大，单独对比各阶段的耗时，发现 v2.1 有了全面提升。

## 应用场景

我们的业务场景根据数据规模和业务复杂度来使用不同的技术框架。趋势如下：

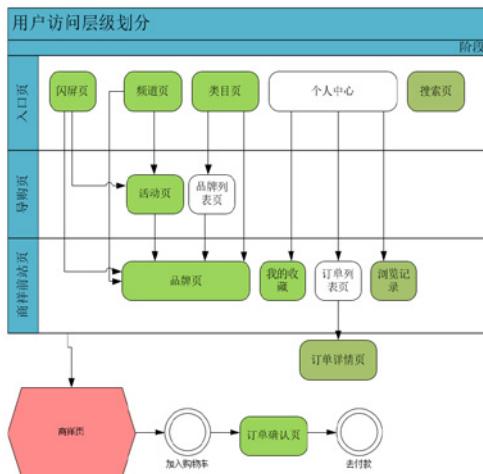


## 数据业务需求可视化结构

曝光转化分析是对平台坑位的曝光点击率做多维分析的 Cube，日增数据量在数亿级别。用户画像分析是对基于平台所有用户的属性做多维分析的 Cube，日增数据在三千万左右。

以下我们成单路径分析为例做详细介绍：

**简介：**成单路径是围绕用户从浏览页面到最终下单到支付的整个生命周期的用户行为路径分析。采用的是归因算法。我们采用的归因方法是事先对我们平台的页面进行划分层级，用户在返回上一层级的时候重新覆盖。这样我们在计算最终转化率能达到，一个订单最终只归到一条下单路径，具体的页面划分层级如下：



下表是成单路径的维度和指标说明。

初期我们使用 Presto 来做成单路径分析，当时数据量日增还在 1 千万左右，90% 查询在 10 秒以内。随着公司用户规模的增长，行为数据呈指数级增加，数据高峰期日增达到上亿级别，Presto 的查询显得有点力不从心，我们引进了 Kylin，大大得缓解了问题，90% 的查询的性能回归到 1 秒以内。后记：

卷皮 OLAP 一年多时间经历了三次重大的变革，目前平台采用 Presto 和 Kylin 两种引擎并用，

维度	备注
日期	年, 季度, 月, 周, 日
入口页	页面 id, 页面类型, 页面名称
导航页	页面 id, 页面类型, 页面名称
前站页面	页面 id, 页面类型, 页面名称
渠道	用户的来源渠道
站点	九块邮或者卷皮折扣
终端	iOS, Android, 小程序, 微信, PC
商品商务	负责运营商品的商务
商品来源	入库商品, 普通商品
类目	商品的一级二级三级类目
商家	商家名称, 公司名称
活动类型	商品参加的活动类型
品牌	商品的品牌
商品	商品标题, 价格
指标	备注
pv	页面事件的访问次数或者点击事件的访问次数
uv	页面事件的访问人数或者点击事件的访问人数
商详 uv	商品详情页的访问人数
支付人数	支付人数
订单量	支付订单量
销量	订单的支付销量
销售总额	订单的支付金额
优惠金额	订单的优惠金额
满减金额	订单的满减金额
支付转化率	$\langle \text{支付人数} \rangle / \langle \text{商详 uv} \rangle$
商详到达率	$\langle \text{商详 uv} \rangle / \langle \text{uv} \rangle$

事实表日增数量级在千万级别或以下，维表数多在 15 张以上最好采用 Presto，而事实表日增数量级在千万级别以上乃至上亿，维表数小于 15 个时候可以采用 Kylin。采用 Kylin 一定要将模型提前设计周全，不要频繁变更，因为每次模型变更数据都需要重刷，重新聚合，费时费力。

# 后Kubernetes时代的微服务

作者 Bilgin Ibryam 译者 盖磊



微服务的关注热度起源于一大堆极端的想法，涉及组织的结构、团队的规模、服务的规模、重写和抛出服务而不是修复、避免单元测试等。依我的经验看，其中大部分想法已被证明是错误的、不实用的，或者至少在一般情况下是不适用的。当前能残存下来的微服务原则和实践，大部分是非常通用和宽松定义的。虽然它们适合未来许多年内的发展，但在实践中并没有多大的意义。

早在 Kubernetes 横空出世的几年前，微服务理念就得到了采用，目前，它仍然是一种最流行的分布式系统架构风格。Kubernetes 和云原生运动已大规模地重定义了应用设计和开发的一些方面。在本文中，我试图提出一些原始的微服务理念。我将指出，这些理念在后 Kubernetes 时代已不再像以前那样强大。

## 服务不仅应可观测，而且应是自动化的

可观测性 (Observability) 是微服务自一开始就提出的一个基本原则。可观测性虽然适用于一般的分布式系统，但是当前，尤其是在 Kubernetes 上，可观测性的主要涉及平台层的开箱即用，例如进程运行状况检查、CPU 和内存消耗等。应用能以 JSON 格式登录控制台，这就是可观测性的最低要求。此外，平台应可以在无需过多服务层开发的情况下，实现跟踪资源的消耗、开展请求追踪、收集全部类型的指标、计算错误率等。

在云原生平台上，服务仅具备可观测性是不够的。更基本的先决条件是使用检查健康、响应信号、声明资源消耗等手段实现微服务的自动化。任何应用都可以置于容器中并运行。但是要创建

一个可通过云原生平台自动化和协调编排容器的应用，则需要遵循一定的规则。遵循这些原则和模式，可确保所生成的容器作为云本地成员在大多数容器编排引擎中表现为优秀，并支持对容器进行自动化的调度、扩展和监视。

我们希望平台不仅可观测服务中发生的情况，而且希望平台能检测到异常，并按照声明情况做出协调。纠正措施可以是通过停止引导流量到服务实例、重新启动、向上 / 向下扩展，也可以是将服务迁移到另一台健康的主机、重试失败的请求或是其它一些操作。如果服务实现了自动化，那么所有这些纠正措施都会自动做出，我们只需要描述所需的状态，而不是去观测并做出响应。服务应该是可观测的，但也应在无需人工干预的情况下由平台实现问题整改。

## 具备正确职责的智能平台和智能设备

在从 SOA 转向微服务的过程中，在服务交互上发生的另一个根本转变就是“智能端点哑管道（smart endpoints and dumb pipes）”这一理念。在微服务领域，服务不依赖于所具有的集中式智能路由层，而是依赖于具有某些平台级功能的智能端点。服务的实现是通过在每个微服务中嵌入传统 ESB 的部分功能，并转为使用不具有业务逻辑元素的一些轻量级协议。

这仍然是一种惯常采用的方法，即在不可靠的网络层（使用诸如 Hystrix 之类的库）实现服务交互，但在当前的后 Kubernetes 时代，服务交互已完全被服务网格（Service Mesh 技术）取代。服务网格吸引人之处在于，它甚至要比传统的 ESB 更智能。网格可以执行动态路由、服务发现、基于延迟的负载平衡、响应类型、指标和分布式跟踪、重试、超时，以及我们所能想到的所有特性。

与 ESB 的不同，服务网格只有一个集中路由器层，每个微服务通常都具有自己的路由器，即一个使用额外中央管理层执行代理逻辑的“跨斗模式容器”（Sidecar Container）。更重要的是，管道（即平台和服务网格）中并不维持任何业务逻辑。管道完全聚焦于基础架构问题，而让服务

聚焦于业务逻辑。下图表示了为适应云环境的动态和不可靠特性，ESB 和微服务在认知上的演变情况。

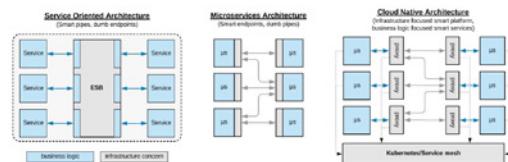


图 从SOA到MSA和CNA

如果查看服务的其他一些方面，我们就会注意到云原生不仅影响了端点和服务交互。Kubernetes 平台（及其所有附加技术）还负责资源的管理、调度、部署、配置管理、扩展和服务交互等。与其再次称之为“智能代理哑管道”，我认为更好的描述应是一种具备正确职责的智能平台和智能服务。它不仅是与端点相关，而且也是一个完整的平台，实现主要聚焦于业务功能的服务在所有基础架构上的自动化。

## 设计不应针对“故障”，而应针对“恢复”

毫无疑问，要在基础架构和网络本身并非可靠的云原生环境中运行微服务，我们必须针对故障做出设计。但是越来越多的故障是由平台检测并处理的，而人们对如何从微服务中捕获故障的考虑较少。相反，我们应通过考虑从多个维度实现幂等性，设计我们的恢复服务。

容器技术、容器编排和 Service Mesh 可以检测许多故障，并从中进行恢复。例如无限循环（分配 CPU 份额）、内存泄漏和 OOM（运行状况检查）、磁盘占用（配额问题）、Fork 炸弹（进程限制），批量处理和进程隔离（限制内存份额）、延迟和基于响应的服务发现、重试、超时、自动扩展等。同样，在过渡到无服务器模型后，服务必须在几毫秒内处理一个请求。看上去对垃圾回收、线程池、资源泄漏等问题的关注，越来越成为一些毫不相关的问题……

使用平台处理所有诸如此类的问题，会将服务视为一个密封黑盒子。该黑盒子应支持多次启

动和停止（支持服务重新启动）、服务按比例的放大和缩小（通过将服务成为无状态的以支持安全扩展）、假定许多传入请求最终会超时（使端点具有幂等性）、假定许多传出请求将暂时失败并且平台将会做出重试（确保我们使用了幂等服务）。

为实现自动化在云原生环境中适用自动化，服务必须满足下列条件：

- 对重启的幂等（服务支持多次被杀掉并启动）。
- 对向上/向下扩展幂等（服务可实现多个实例的自动扩展）。
- 对服务生成者幂等（其它服务可重试调用）。
- 对服务消费者幂等（服務或服务网格可以重试传出请求）。

如果服务在一次或是多次执行上述行为中总是表现出同一方式，那么平台就可以在无需人工干预的情况下从故障中恢复服务。

最后请记住，平台提供的所有恢复只是一些本地优化。正如 Christian Posta 所指出的，分布式系统中应用的安全性和正确性仍然是应用的责任。对于设计一个整体稳定的系统，业务流程整体范围中的思维模式（可能跨越多个服务）十分重要的。

## 双重开发职责

越来越多的微服务原则已被 Kubernetes 及一些补充项目实施和提供。因此，现代开发人员必须做到不仅要精通编程语言去实现业务功能，而且同样也要精通云原生技术去完全满足一些非功能性基础架构层上的需求。

业务需求和基础架构（操作上的需求、跨功能的需求，或是一些系统质量属性）之间的界限通常是模糊不清的，我们不可能只采取其中的某个方面，而期望其他人去实现另一个方面。例如，如果要在服务网格层实现重试逻辑，那么必须使服务中的业务逻辑或数据库层所使用的服务具有幂等性。如果在服务网格层使用超时，那

么必须在服务中实现服务使用者超时的同步。如果必须要实现服务的定期执行，那么必须配置 Kubernetes 作业去按时间执行。

展望未来，一些服务功能应作为业务逻辑实现在服务中，而其它一些服务功能则应作为平台功能提供。虽然使用正确的工具去完成正确的任务是一种很好的责任分离，但新技术不断出现极大地增加了整体的复杂性。要在业务逻辑方面实现简单的服务，我们需要很好地理解分布式技术堆栈，因为开发职责是分散在各个层上的。

事实证明，Kubernetes 支持向上扩展到数千个节点，数万个 Pod 和每秒数百万事务。但它是否同样支持向下扩展？对我来说，我并不清楚应用的规模、复杂性或关键性的阈值应该是多少，才能证明我们引入复杂的“云原生”是正确的。

## 结论

看到微服务运动为采用 Docker 和 Kubernetes 等容器技术提供了如此巨大的动力，这是非常有意思的。虽然一开始是微服务实践推动了这些技术的发展，但现在是 Kubernetes 重新定义了微服务架构的原则和实践。

从最近的一些实例看，我们将很快采纳功能模型作为有效的微服务原语，而不是将微服务视为纳米（nanoservice）服务的反模式。我们并没有充分质疑云原生技术对于中小型案例的实用性和适用性，而是出于兴奋有些随意地投身到这个领域中。

Kubernetes 从 ESB 和微服务中汲取了大量经验，因此它将会成为最终的分布式系统平台。它是一种用于定义建筑风格的技术，而不是反之。究竟是好是坏，只有时间才能证明。

Bilgin Ibryam (@bibryam) 是 Red Hat 的首席架构师、提交者和 ASF 成员。他也是一名开源布道师、博客作者，《Camel Design Patterns》和《Kubernetes Patterns》等书的作者。在他的日常工作生活中，Bilgin 喜欢指导、编码和领导开发人员成功地构建云解决方案。

# 2018年《DevOps 现状报告》行业脉动与团队发展指导一览

作者 Sarah Schlothauer 译者 邵思华



DORA 与 Google Cloud 合作发布的 2018 年《DevOps 现状报告》已经新鲜出炉，今年的报告中有哪些新的结论？你又应当如何根据这些结论，促使你的团队发展为精英水平？

## 关于《DevOps 现状报告》

DORA 公司（DevOps Research and Assessment, DevOps 研究与评估）从 2014 年起每年都会发布一份关于 DevOps 现状的报告。在过去 4 年，《DevOps 现状报告》都是 DORA 和 Puppet 合作针对软件开发人员开展原创性研究，今年报告的主要合作者已从 Puppet 转为 Google Cloud。《DevOps 现状报告》的目的是为了对改进技术交付团队的资源管理、生产力和质量提供指导。DORA 通过使用数据驱动的研究方法，并

创立基于证据的工具，帮助企业了解如何改变自身的实践，从而为软件开发、产品管理和组织转型提供科学方法。

8 月 29 日，DORA 发布了《2018 DevOps 现状报告》。来自各行各业的 1800 多名调查者提交了问卷，调查内容涵盖了云基础设施、领导力与学习氛围、交付效能、数据库实践等等。

今年是 DORA 连续第五年发布该报告，在报告中可看到新的关注点以及更广泛的调查方向。在 2014 年时，只有 16% 的调查参与者表示自己在 DevOps 团队。而在 2018，这个数字已经增长到 27%。

今年的报告中有哪些新的结论？你又应当如何根据这些结论，促使你的团队发展为精英水平？让我们来解读一下这份报告，看看报告中的

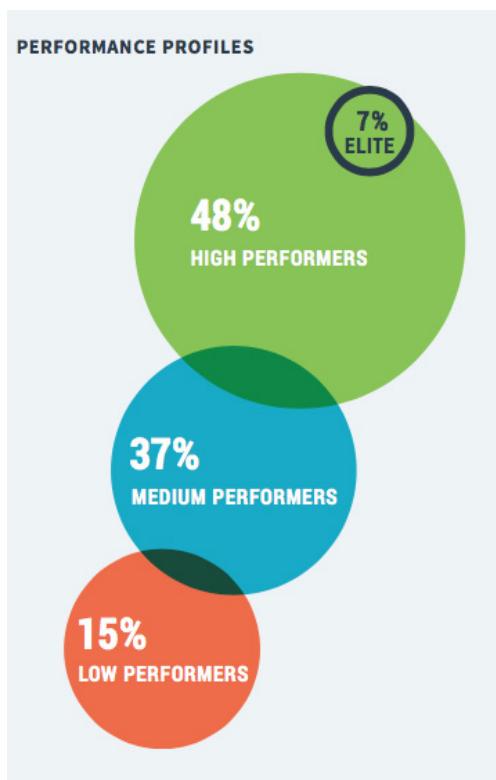
结论对于持续发展的 DevOps 领域具有怎样的意义。

## 新定义的 DevOps 精英级团队

根据 DORA 的软件交付效能基准，团队被划分为三种类型：高效能、中效能与低效能团队，对团队的评价取决于他们的总体产出。发布频率、变更响应时间、服务恢复时间，以及变更故障率等指标是划分的参考标准。

## 表格

在今年的调查中，有 15% 的团队被划分为低效能团队、37% 为中效能，而 48% 的团队属于高效能团队。其中在高效能团队中的 7% 可归为精英级团队，他们的表现可称卓越。



## 团队效能情况

精英级执行团队在以下几个方面有着突出的表现：

- 代码发布频率高 46 倍
- 代码提交至发布的速度快 2555 倍
- 变更故障率少 7 倍
- 事故恢复时间快 2604 倍



与之前的报告相比，高效能团队的比例逐年增长，表明行业正在持续改善。DORA 发现，表现最好的团队（如“精英团队”）拥有最高的软件开发和交付水平，正如过去几年中的观察一样。同时调查结果显示，表现欠佳的团队正在努力跟上，但与高效能团队之间的差距在不断拉大。

那么，是哪些关键的因素影响了普通团队的效能呢？报告中指出，低效能团队往往对于软件开发与发布往往采取过于谨慎的做法。

## 团队的专注度是成功的关键因素

低效能团队如何转变为高效能团队，乃至成为精英级团队呢？

报告指出，“研究表明，对于各家软件供应商来说，高效能的执行团队与低效能团队相比，手工操作的比例减少了许多，能将更多地时间用于新工作的开展，而在修复安全漏洞或缺陷的时间上则减少许多。”

改进效能的一大关键是自动化能力，自动化能够加速任务的完成，改进质量与一致性。其结果是团队能够更多地投入在更有价值的工作任务上。由于自动化能力处理了各种低级别工作任务，使人力得到了解放。

现实情况是，团队对时间的利用方式造成了低效能的结果。尤其值得注意的是，各种计划外的工作、安全问题、干扰、客户支持任务以及缺陷是阻碍低效能团队向前进步的绊脚石。

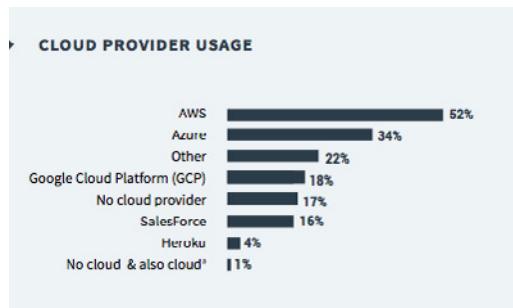
精英团队能够将 50% 的时间投入在新工作中，相比之下，低效能团队只能投入 30% 的时间。低效能团队还需要投入 15% 的时间用于客户支持工作，与之相比，精英团队在这方面只需投入 5% 的时间。

## 云计算持续发展

报告中显示，Forrester（一家独立的技术和市场调研公司）预测 2018 年全球公有云市场总量将达到 178 亿美元，比 2017 年增长 22%。福布斯报告称，到 2020 年，83% 的企业工作负载将在云端。在 DORA 的调查中，67% 的受访者表示他们正在开发的主要应用程序或服务是托管在某种云平台上的。

毋庸置疑，云计算市场仍在不断增长之中。从调查结果来看，只有少部分用户仍然游离在外。

报告中有 17% 的调查者仍然没有使用云厂商的服务。与此同时，AWS 在最受欢迎的云平台中占据了头把交椅，投票率为 52%，Azure 屈居次席，占比为 34%。



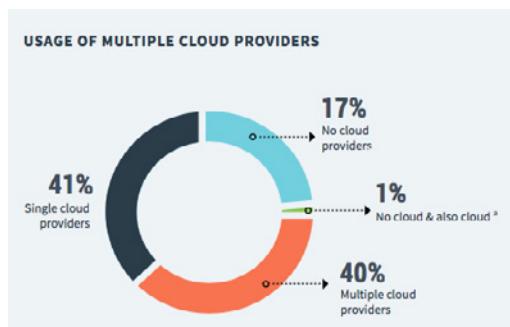
云服务商占比

此外，采用多云的方式已经逐渐普及。41% 的调查者表示他们选择了单一云服务，与之相比，有 40% 的调查者使用了多个云平台的服务。造成这一现象的原因有以下几点：可用性、灾难恢复计划、对于单一服务商信心不足，以及法律合规性原因。

那么，精英团队在用云方面有哪些共性呢？

“选择 使用 PaaS（平台即服务）的调查者与

其他调查者相比有 1.5 倍的可能性是精英用户，遵循了云原生最佳实践的用户有 1.8 倍的可能性是精英团队。选择通过基础设施即代码（IaC）方式管理云发布的用户有 1.8 倍的可能性是精英，最后是选择容器技术的用户有 1.5 倍的可能性是精英团队。”



多云的采用情况

不过，云平台的应用情况似乎遇到了某些障碍。用户并没有充分利用云平台所带来的决定性的特点，这对软件交付的效能产生了影响。DORA 列举了 5 项云计算的基本特征，但这几方面大家都还没有达到要求。

以下是在调查者反馈的自己多大程度上符合云平台的特性：

- 按需分配，自主服务：46%
- 广泛的网络访问性：46%
- 资源池：43%
- 快速的弹性能力：45%
- 符合标准的服务：48%

以上这些特征的评分都没有超过半数的选择，这也反映出团队在效率方面的不足。

## “文化是 DevOps 的重要组成部分”

组织的文化如何才能健康地发展？DORA 引用了社会学家 Ron Westrum 所提出的看法。Westrum 发现“企业的文化可通过安全感与效能产出进行预测”。

企业的文化可能是病态的（权力导向），官僚的（制度导向），也可能是生机勃勃的（效能导向）。每一种文化在处理协作、职责、风险、

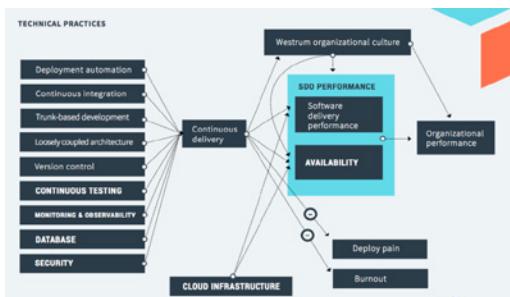
新事物与面对问题时的反应都是不同的。为了 DevOps 能够成功实施，应当满足某些基本的文化条件，包括打破团队壁垒，以及实施新的方式方法。

如何利用这些已知的内容使你的团队得到改进呢？请记住其中的精华部分：“谷歌的研究者对 180 个工程团队进行了研究分析，他们发现，高效能的团队通常在心理上有安全感，或者说在团队中不会害怕冒风险。除此之外，其他因素还包括可依赖性、工作的结构化与清晰度、工作的意义与个人影响力。”

高效能团队的其他重要特点还包括浓厚的学习氛围，学习是一种有价值的投资，它能够带来效能的回馈。为了减轻团队的负担，学习任务应当在工作时间完成，而不是在周末或加班后作为额外的任务。工作中学习的方式能够让团队更快地适应未知的变化。

## 对技术转型至关重要的实践

DORA 的研究强调了对技术转型至关重要的实践。这些重要的实践包括版本控制，自动化部署，持续集成，基于主干的开发以及松散耦合的架构。今年我们还发现，有助于持续交付的实践包括：使用监控和可观察性解决方案，持续测试，将数据库更改集成到这样的软件交付流程中，以及关注安全性。



### 至关重要的技术实践

其中，持续测试的实践包括：

- 不断审查和改进测试套件，以更好地发现缺陷并控制复杂性和成本；
- 允许测试人员在整个产品开发和交付过程中

与开发人员一起工作；

- 在整个交付过程中（包括如探索性测试，可用性测试和验收测试等）执行手动测试活动；
- 在为代码库的所有更改写生产代码之前编写单元测试，来练习测试驱动的开发；
- 能够在不到十分钟的时间内从本地工作站和 CI 服务器获得自动化测试的反馈。

## 受众面的提高

最后，2018 年的报告得到了更多女性从业者的反馈，这也反映出行业性别分布状况的变化。今年，调查者参与者有 12% 为女性，并且按调查者的反馈，他们的团队中有 25% 的成员为女性。这是个非常大的变化：去年仅有 6% 的调查参与者为女性，进入 IT 业的女性队伍正在不断壮大。

此外，有 4% 的调查者并未留下性别信息，还有不到 1% 选择了非二元性别的选项。

今年，DORA 首次在问卷中间及调查者的伤残状况，有 6% 的调查者表示他们有某部分的身体残疾，另有 9% 的调查者未作出选择。

## 效能方面的关键结论

如果你需要更多信息来促进团队进入精英领域，可以参考以下报告结果：

- 开源非常重要，“高效能团队深入应用开源软件的可能性是其他类型软件的 1.75 倍，而这些团队在未来提高开源软件使用度的可能性也是其他团队的 1.5 倍。”
- 外包会带来效能下降，“低效能团队将整部分功能进行外包的可能性几乎是高效能团队的 4 倍，这些外包功能包括测试或运维等等。”
- 在任何行业中都存在高效能的团队，“我们在具有高合规性要求及无合规性要求的行业中都能看到高效能团队的存在。”
- 团队的信任促进公司的发展，“我们发现，当团队主管让成员能够自治地开展工作时，就能够提高团队信任感。”

# CNUTCon

全球运维技术大会

8折报名中，立减720元

团购享更多优惠！

## 聚焦12大专题

- AIOps实践与探索
- 大规模系统的性能优化
- 微服务架构与实践
- 自动化运维平台实践
- 智能+时代下的运维破局
- Kubernetes实战
- 监控与分析（APM）
- 数据库运维
- 运维新技术
- 日志处理
- CI/CD实践
- SRE实践与思考

## 部分演讲嘉宾



陈云  
百度 智能云事业部  
资深研发工程师



不畏  
阿里云  
视频云运维专家



刘伟  
腾讯互娱  
技术运营部高级工程师



夏舰波  
京东商城  
技术架构部资深架构师



顾宇  
前ThoughtWorks  
高级咨询师



谭宇（茂七）  
阿里巴巴  
高级技术专家



闫鹏  
阿里云  
ARMS技术负责人



刘林  
搜狗  
资深软件工程师



李浩  
eBay 主管工程师



周辉  
美团点评  
资深移动架构师



郭忆  
网易 数据库平台开发专家  
网易云 数据库产品负责人



茹炳晟  
极客时间App专栏作者  
eBay中国研发中心  
测试基础架构技术主管



李越敏  
Twitter总部  
软件工程师



巨震  
华为 软件工程师



会议：2018.11.16–17 | 培训：2018.11.18–19

地址：上海 光大会展中心大酒店

售票咨询(电话): 13269078023

售票咨询(邮箱): piaowu@geekbang.org

# AiCon

全球人工智能与机器学习技术大会

8折报名中，立减720元

团购享更多优惠！

## 大会聚焦

- 机器学习应用和实践
- 搜索推荐与算法
- 计算机视觉
- 数据智能驱动业务
- NLP和语音技术
- AI+行业案例



## 大咖助阵



颜水成  
360人工智能研究院  
院长及首席科学家



华先胜  
阿里巴巴达摩院机器  
智能技术实验室副主任  
城市大脑人工智能技术负责人



裴健  
京东集团副总裁  
大数据与智能供应链事业部总裁



马维英  
今日头条副总裁  
人工智能实验室负责人



崔宝秋  
小米人工智能云平台  
副总裁 首席架构师

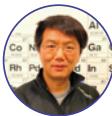
## 部分分享嘉宾



薄列峰  
京东金融  
AI实验室首席科学家



张俊林  
新浪微博  
AI Lab资深算法专家



陈博兴  
阿里巴巴达摩院  
资深算法专家



袁进辉 (老师木)  
一流科技创始人



王兴星  
美团点评技术总监  
外卖商业技术负责人

• • •

会议：2018年12月20–21日

培训：2018年12月22–23日

地址：北京·国际会议中心

联系热线（同微信）：18514549229



# 闲鱼基于Flutter的移动端跨平台应用实践

作者 王树彬



## 闲鱼为什么使用 Flutter

Flutter 作为 Google 新一代的跨平台框架，有较多的优点，但跟其他跨平台解决方案相比，最吸引我们的是它的高性能，可以轻松构建更流畅的 UI。虽然各跨平台方案都有各自的特点，但 Flutter



的出现，给闲鱼、给大家都提供了一种新的可能性。

那么，Flutter 为什么会有高性能呢？

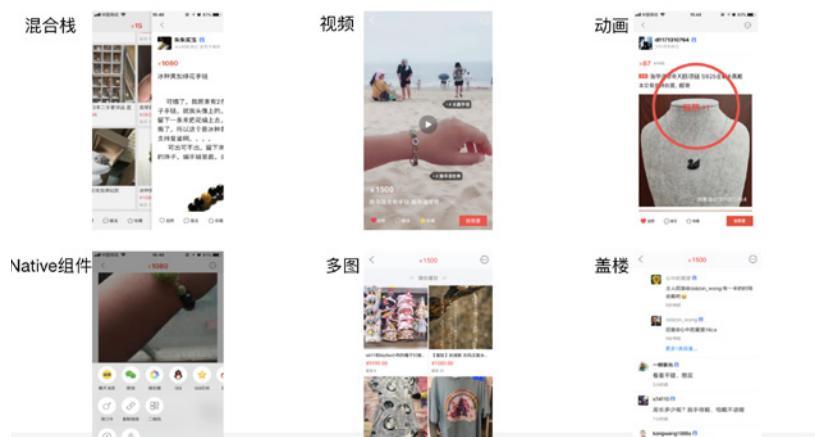
首先，Flutter 自建了一个绘制引擎，底层是由 C++ 编写的引擎，负责渲染，文本处理，Dart VM 等；上层的 Dart Framework 直接调用引擎。避免了以往 JS 解决方案的 JS Bridge、线程跳跃等问题。

第二，引擎基于 Skia 绘制，操作 OpenGL、GPU，不需要依赖原生的组件渲染框架。

第三，Dart 的引入，是 Flutter 团队做了很多思考后的决定，Dart 有 AOT 和 JIT 两种模式，线上使用时以 AOT 的方式编译成机器代码，保证了线上运行时的效率；而在开发期，Dart 代码以 JIT 的方式运行，支持代码的即时生效（HotReload），提高开发效率。

第四，Flutter 的页面和布局是基于 Widget 树的方式，看似不习惯，但这种树状结构解析简单，布局、绘制都可以单次遍历完成计算，而原生布局往往要往复多次计算，“simple is fast”的设计效果。

下面截图是目前闲鱼已经上线的商品详情页面。



商品详情页包含混合栈、视频、动画、原生组件、多图、留言盖楼等功能，页面较复杂，有代表性，也是闲鱼最重要的页面之一。选择商品详情页做为第一个 Flutter 页面，是闲鱼能成功快速使用起 Flutter 的重要因素。

接下来介绍一下，闲鱼的实践过程和总结。

## Flutter 与 Native 混合开发实践

### Flutter Hybrid 工程实践（研发时）

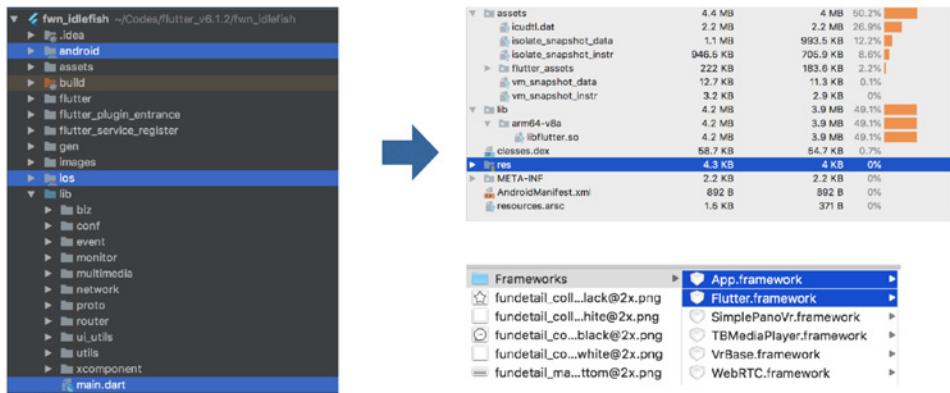
我们把 Flutter 和闲鱼现有的 APP 做渐进式的整合，App 中会同时有 Native、Flutter 和 H5 页面。现有的 Flutter Demo 和应用，都是独立的 Flutter 应用，而当把它和 Native 混合的时候，会碰到很多的困难。

首先是研发时的问题，怎么让 Flutter 在现有的 Native 工程中开发起来。下面这个要从这张图说起：

闲鱼 Flutter 工程结构如图，三个蓝色背景的目录分别是安卓工程、iOS 工程和 main.dart 入口。

编译产物中以 iOS 为例，APP Framework 是 Flutter 应用页面代码，Flutter Framework 是 Flutter 引擎。

这个过程，需要重点考虑几个问题：如何基于现有工程搭建混合工程？如何支持过渡期的 Flutter

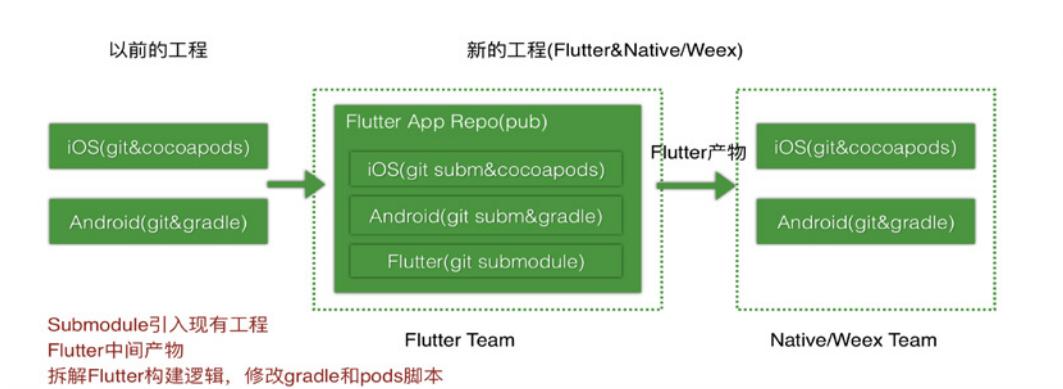


开发及纯 Native 开发的双开发模式？如何让 Flutter 与现有持续集成、构建工具集成？

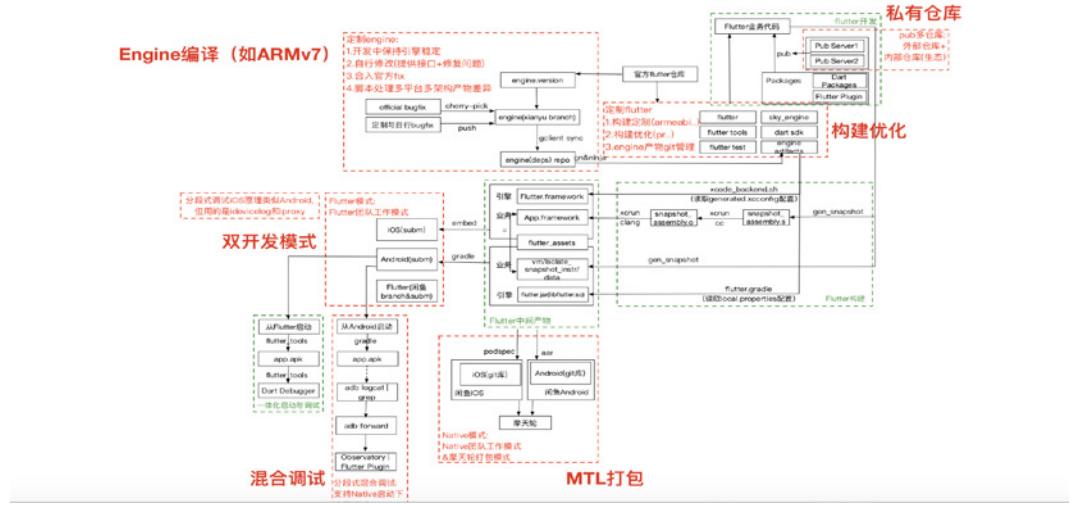
## Hybrid工程的问题



首先，现有的 Native 工程并不符合 Flutter 默认的规范，两者不能完全匹配，需要修改打包脚本，甚至修改 Flutter 的打包 Tool 来解决。另外，我们通过 Submodule 将现有工程引入到 Flutter 父工程中。



纯 Native 开发同学，不需要引入 Flutter 工程，直接在 iOS 或 Android 工程下开发，Flutter 以产物



的方式集成到 Native 中运行，Flutter 的开发同学引入 Submodule。

上图是工程上的修改点。绿色虚线部分是 Flutter 默认的结构，红色虚线是闲鱼在 Flutter 基础上做的定制。Flutter 的构建工具 gen\_snapshot，会把业务代码、Flutter 框架、引擎编译成中间产物，以 so 或 Framework 的方式变成 Native 的一部分。

几个主要的改动点：

- 第一，构建私有的仓库，用来管理阿里私有包，如 CDN、无线网关等中间件适配 Package。
- 第二，构建工具和引擎的优化。
- 第三，跟现有的构建工具打通，混合调试等。

## 闲鱼iOS工程修改示例

- 修改工程名为Runner(包括主代码文件夹)
- 修改Podfile以处理Flutter相关逻辑
- 添加Flutter目录，并指定Runner的Debug Config为Flutter/Debug.xcconfig，Release.config为Flutter/Release.xcconfig
- 修改Runner的Build Phases
- 添加xcode\_backend.sh用于拷贝Flutter.framework，构建App.framework(如果Debug还有snapshot\_blob.bin)，以及构建后的thin。
- 嵌入App.framework和Flutter.framework(对应代码+引擎)
- 嵌入flutter\_assets(资源等)

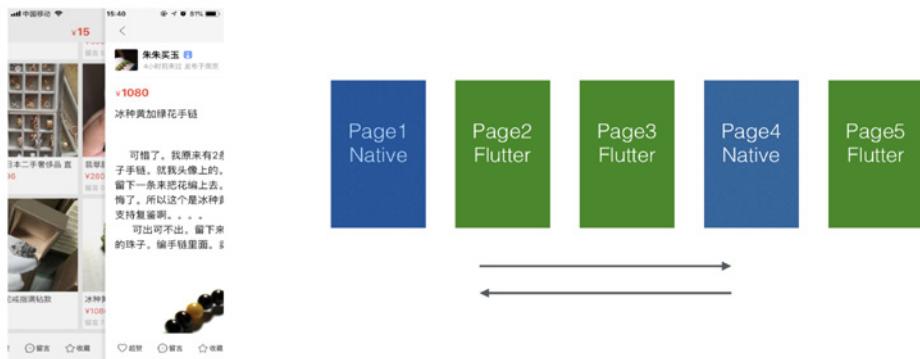
## Flutter Hybrid 栈管理

除了上述的研发时问题，接下来就是让它跑起来，解决运行时问题。其中最重要的是实现混合栈。

# 闲鱼Android工程修改示例

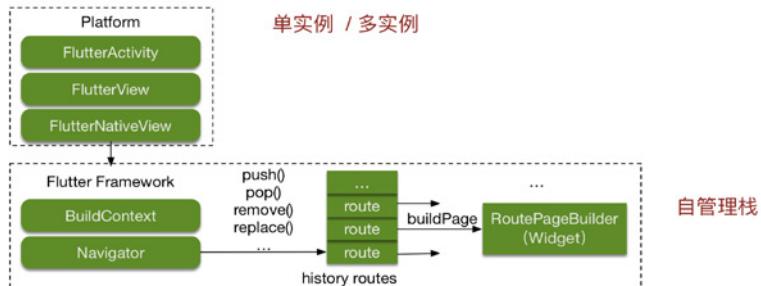
- 添加包文件夹`android/app/src/main`以存储`GeneratedPluginRegistrant.java`
- `local.properties`存储Flutter相关信息
- 引入`flutter.gradle`
- 闲鱼Android支持armeabi
- 修改`flutter.gradle`, 将arm64的处理同arm
- 修改`libflutter.so`目录结构为`lib/armeabi/libflutter.so`
- 修改Android启动Activity启动逻辑以保证多个Launchable-Activity的时候的正确启动。

## 混合栈的定义



在混合工程中，Native 页面，Flutter 页面之间会以多种可能的顺序混合入栈，出栈。要怎么去做？先看一下 Flutter 内部栈的管理默认下是怎么做的：

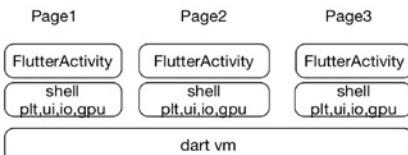
整个 Flutter 运行在一个单例的 Activity 容器里（用安卓举例），Flutter 内部的所有页面都在这个



容器中管理。对安卓来说，怎样把这样容器里面的栈与 Native 栈混合起来，直接的一个想法就要把栈自己托管起来，把这个容器在 Android 的栈中来回移动。但 Android 里想这样操作非常难。

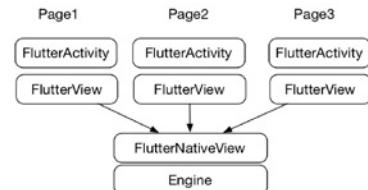
所以解决这个事情，就主要有两个问题要考虑，首先就是混合栈要在哪里管理？是在 Hybrid 栈管理，还是在 Flutter 管理，第二个就是关于实例剥离的问题，既然移动单例很复杂，那就把单例剥离出来，在上面 Wrap 出多个实例，这样就方便管理了。下面是两个对比方案

方案一



更简单  
页面隔离性好

方案二



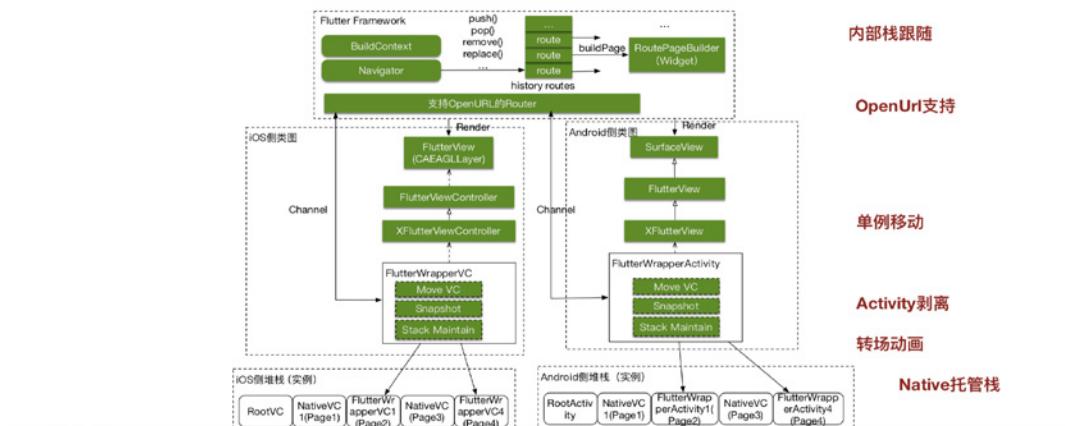
更快的加载速度  
数据多页面复用

这两种方案都是可选的，方案一就是把 Flutter 直接变成多例，每个 Flutter 页面重新启动一个 Flutter 的容器，每个 Flutter 页面就像通常使用 WebView 一样，这个方便我们做了实测，发现它的启动速度有影响，能感觉到一些卡顿，另外，还有一个问题，当我想在两个页面之间去复用数据的时候，那两个引擎之间是完全隔离的，最后数据不好复用。这个方案的好处是很简单，如果喜欢隔离性，也可以变成优点。

第二种方案，就是做浅层的单例剥离，尽量多的遵守 Flutter 的标准运行方式，以最小的影响把单例剥离出来，Wrap 成多例。

这种方案是在 Flutter View 这一层剥离，关于 Flutter View 的概念看一下源码很容易理解。

这种解决方案的好处是可以实现多页面复用，因为不用每次都取一个新的实例，加载速度会更快，因为对闲鱼来讲，我们追求的就是性能，最后我们的选择就是方案二。



上图是具体的实现方式。

把下面的 View 复用，在多个 Activity 之间移动，切换到下一个页面的时候，把这个可复用的 View 从前一个 Activity 移走，放到下一个 Activity，这是它的主要的思路。

在这个思路下也会遇到一些需要解决的问题：

两个页面转场动画由于 View 在 Activity 间移动，会有一个短暂的白色闪屏，体验不好，解决闪屏的办法，就是做一个截图，从 A 页面到 B 页面的时候，对 A 页面做个截图，同时把 Flutter 自带栈的转场动画禁止掉，有这个截图，转场时就不会有闪屏的感觉了。

考虑对统一 OpenUL 支持，把 Flutter 和 Native 的 URL 统一。

由于 Flutter 容器内部有个栈管理，对这个栈需要与 Native 做同步的跟随。

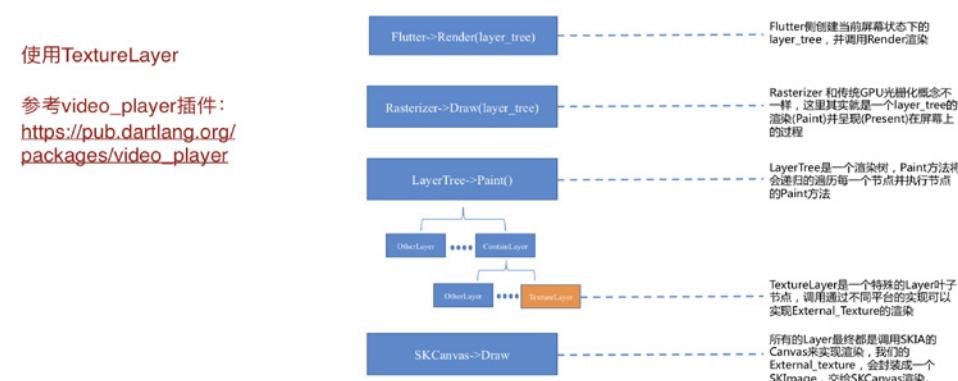
到此，混合栈的方案就简单介绍完了。

## 基于 Texture 的自定义视频播放器

接下来，如果 Flutter 页面中想复用已有的 Native 组件，怎么办？

一种情况是视频播放器，Native 中我们做过很多优化的播放器，希望能复用到 Flutter 页面中。

首先，还是先看原理：



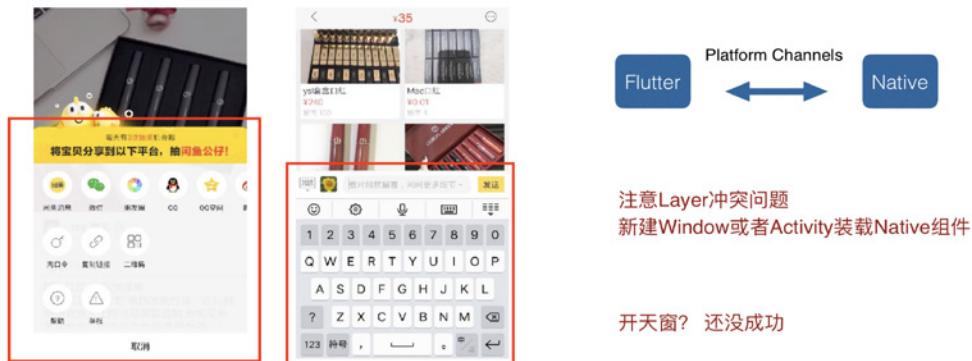
Flutter 内部的渲染，与通常的做法一样，有 layer。其中一种 Layer 叫 Texture Layer，可以把任何



其他地方计算出来的纹理直接贴到 Flutter 的 Texture Layer 上。不管是视频，还是图片，如果有需要，都可以用 Texture Layer。

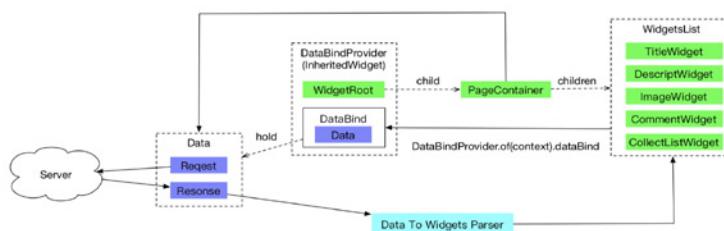
在这个实现的方式中，Flutter 侧负责展示这个播放器 UI，接收对播放器做控制交互，而 Native 侧负责视频的渲染，通过 TextureLayer 展示到 Flutter 侧。而控制协议，通过 Flutter 特有的 MethodChannel 来控制。

除了视频，还有没有其他类型的 Native 组件能复用到 Flutter 中？像下图这样，把 Native 控件放在 View/Window 中与 Flutter 混合，是可以的。但截止演讲时，Flutter 还无法做到在 Flutter 中挖个小天窗嵌入 Native 组件。不过这种方式 Google Flutter 团队已经在做尝试，未来可能做有办法支持，大家可以关注。



## Flutter 通用问题实践

接下来，介绍一下 Flutter 商品详情页的页面的开发框架。



### 简单的页面：

- 动态组装能力
- InheritedWidget 共享数据
- 统一协议（下页详述）

### 更复杂的交互页面：

- 考虑Redux
- Stream

## 页面框架

右边边绿色的这一部分，就是整个页面的结构，整个详情页面是一个大列表，由商品的描述、图片、评论、个性化推荐等组成。这里简单概括几个特点：

通过 Server 端返回的数据驱动 UI 界面，可以一定程度上获得页面内容的动态能力。Flutter 本身

不支持动态更新，无法像 JS 那样，所以这种设计方式可以一定程度上弥补这方面的短板。

Widget 树结点间（或者说页面的不同组件间）的数据如何共享？这里大家知道 InheritedWidget 这个类就好了，这是解决数据共享的很有用的类。

如果页面再复杂些，有很多交互，希望将视频、交互、数据等分离怎么办？也可以考虑引入 Redux 框架。

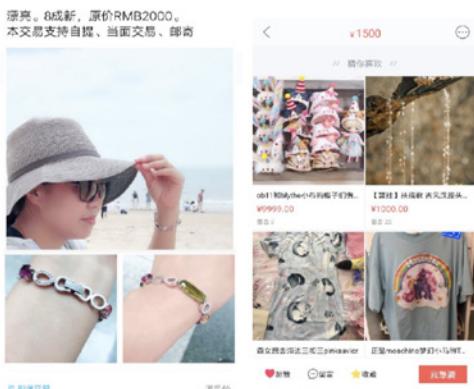
## 统一协议



Flutter 不支持 Dart 的反射（mirror），所以在开发 Flutter 页面时，解析服务端返回的数据，生成 Flutter 对象时，可能会很不习惯，需要有较多的硬编码。Flutter 不支持反射，请大家理解，这样可以获得 tree shaking 能力，减少 Flutter 包的大小。

既然不支持反射，怎么去解决刚才说的数据转换问题？我们实现了一个统一协议层，把 Serve 端和客户端的请求接口和数据模型，都通过协议统一生成代码，避免了手工编码。

## 图片缓存方案



当前默认的图片缓存策略（官方在优化中）：

```
class ImageCache
```

Implements a **least-recently-used cache** of up to 1000 images. The maximum size can be adjusted using [maximumSize].

问题：  
内存占用，容易 Crash

闲鱼的页面中有大量图片，但 Flutter 默认的图片缓存策略比较简单，截止演讲时，如上图所示，默认图片缓存策略是按照图片数量，以 1000 为上限，LRU 的方式置换。当大图片较多时，这会占用过多的内存，容易造成 Crash 或 Abort。

方案1，只有单页面时，简单处理方案：

1，修改最大缓存图片的数量，提供自定义Cache的Hook点

```
class XWidgetsFlutterBinding extends WidgetsFlutterBinding {
  @override
  ImageCache createImageCache() {
    ImageCache cache = new ImageCache();
    cache.maximumSize = 110; // the maximum size
    of cached images.
    return cache;
  }
}
```

2，图片尺寸自适应剪裁

- 图片宽度统一泛化为一系列标准尺寸，如  
60, 128, 234 ... 640, 720, 960, ...
- WebP
- Quality

在我们只有详情页一种页面时，解决这个问题可以用简单粗暴的方式，首先把 1000 这个数量调小。一种修改方式如图所示，通过 WidgetsFlutterBinding 来修改（WidgetsFlutterBinding 是 Flutter 中很重要的一个机制，有兴趣可以深入了解）。

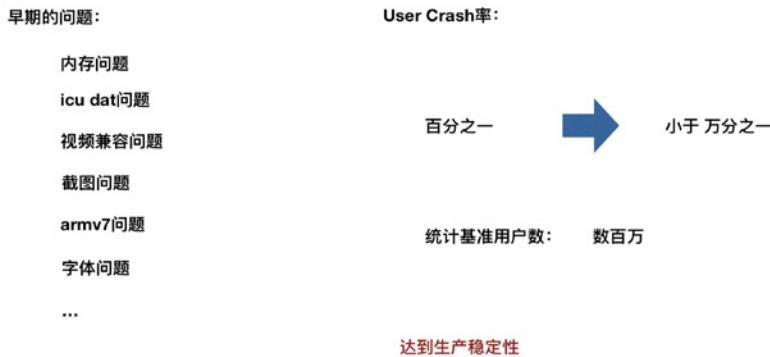
此外，还要注意图片尺寸自适应剪裁，支持 WebP 等，这些对节省图片内存和网络流量都很关键。

第二种解决方案，是官方正在做的优化，按照整个空间的大小来做缓存策略，具体可以关注图中的链接。

第三种方案，更加完善，加一层持久层的缓存，以实际的经验来看，闲鱼的场景下，持久层缓存时，通常可以提高缓存命中率 10% 到 30%。

## 上线效果

### 线上 Crash 率



大家可能会关心 Flutter 在生产环境的稳定性、兼容性等表现。闲鱼使用 Flutter 的前期阶段，这方面确实有很大的问题。前期在真实环境中发现了很多问题，第一次灰度测试时 Crash 率有百分之一的量级，主要的 Crash 问题包括内存、GPU、icu data、视频播放、截图接口、armv7、字体缺失等。

我们和 Google 团队一起，通过几个版本的灰度迭代，用了一个半月的时间，把问题逐步解决了，

目前 Crash 率收敛稳定，达到万分之一的量级，已经达到了生产标准。

## Flutter 与 Native 详情页性能对比

我们对 Flutter 与 Native 的详情页做了简单的性能对比，并不严谨，仅供参考。

测试场景：进入宝贝详情页后快速浏览到页面底部，从猜你喜欢进入第二个宝贝，重复进行访问 10 个不同宝贝详情。对比 Native 版详情页和 Flutter 版详情页。

测试机型，以低端机型为主（高端机型区分不明显）：

Android 4.x, 5.x...

iPhone 5c, 6s...

安卓的对比：

Xiaomi 5.0.2			Huawei 7.0		
	Native	Flutter		Native	Flutter
CPU	32.32	7.31	CPU	16.74	7.91
dvm memory	228.00	111.86	dvm memory	395.92	197.08
native memory	134.00	130.80	native memory	143.00	233.24
total memory	362.00	242.57	total memory	539.08	431.05
FPS	22.13	44	FPS	38.22	46.2
SM	17.68	58.29	SM	45.78	58.09

Lenovo 4.4.4			Oppo 4.3		
Lenovo 4.4.4	Native	Flutter		Native	Flutter
CPU	16.12	5.43	CPU	32.00	26.02
dvm memory	55.00	90.83	dvm memory	111.71	289.10
native memory	96.50	91.43	native memory	104.00	94.07
total memory	152.00	182.58	total memory	216.14	383.60
FPS	42.60	32.62	FPS	16.97	30.46
SM	2.3	31.14	SM	4.78	38.56

下面两行是体现流畅度的，LPS 或者 MS 是腾讯提出的一种流畅度的表达方式，在流畅度是 OK 的，比 Native 详情页做的好，在技术的指标上也还不错。

iOS 的结果：

iphone 5c 9.0.1			iphone 6s 10.3.2		
	Native	Flutter		Native	Flutter
CPU	50.27%	47.3%	CPU	30.31%	27.42%
Memory	149.4	127.28	Memory	247.24	231.66
Traffic total	29349	16757	Traffic total	24868	17686
FPS	39.71	53.08	FPS	47.47	50.08
GPU	4.36%	8.78%	GPU	21.89%	26.25%

注：以新老商品详情页为测试对象，相同的研究团队和相似的开发周期。

iOS 上，也是 Flutter 会更流畅一些，测下来，发现在 GPU 的使用率上，Flutter 会更高一些，Flutter 在这上有更进一步的优化空间。

说到这里，可能大家也会疑惑，这个对比结果，是不是因为以前 Native 写的详情页太复杂了？确实有这种可能。但主要分享的是，两种页面是相同团队成员开发的，并且没有针对 Flutter 做专门

的性能优化，这个性能测试可以确定的结论是，使用 Flutter 还是比较容易就能开发出与 Native 性能相近的页面。

最后，说一下大家可能会关于的成本问题。对于混合开发，初期接入成本是有。如果是全新的 Flutter 独立应用，接入成本会很低。首次接入完成后，后面开始会顺利很多，可以享受跨端统一编程，一套代码带来的效率快感。另外，关于学习成本，还好，因为 Dart 语言跟 Java 很像，跟 JS 也很像，另外 Flutter 的 UI 框架遵循响应式，声明式设计原则，个人感觉，较容易上手。谢谢大家，由于水平有限，可能会有错误，请大家指正。

王树彬，阿里巴巴闲鱼无线技术专家，毕业于浙江大学，2009 年加入阿里巴巴，现任阿里巴巴闲鱼架构负责人，负责闲鱼从端到云的整体架构升级。有十余年互联网研发经验。曾负责移动端 LBS 技术，是淘宝位置归一、地理围栏等技术的开拓者，为个性化、O2O 等业务提供基础能力。也曾负责淘宝的商家系统，建立商家十亿级大数据下的实时在线查询、挖掘服务。

# 聊聊工程师的影响力

作者 CHARITY.WTF 译者 无明



让我们来聊聊工程师的影响力。作为一名工程师，你是如何获得影响力的？什么是影响力，它的根源是什么，你该如何运用它或者怎样会失去它？它与管理者的权力和影响力有什么不同？

这个话题通常与那些迫切希望成为管理者以便获取更多信息和决策影响力工程师有关。这是一个危险的信号，但糟糕的是，这种情况却很普遍。

如果出现这种情况，你需要做一些自我反省。你的公司是否为高级工程师提供参与领导和决策的机会？是否有一个与管理者并行的工程师职位轨道，至少和总监一个级别？他们之间是否平等互补？公司是否有为工程师量身定制的职业发展阶梯？对于非管理者来说，公司的决策过程是否太过神秘？你认为理所当然的东西，别人可能不

这么想，所以一定要去问问他们的想法。

或许他们说的只是个人的想法。或许他们不相信你。或许他们只在管理者独裁的公司工作过。或许他们在很多公司工作过，还和你一样认为工程师可以拥有巨大的影响力，但其实这是彻头彻尾的谎言，以致于到最后希望彻底破灭。或许他们因为各种原因不习惯于掌握权力。

不管怎样，那些想要成为管理者以便能够长久利用权力的人，到最后一定不希望成为管理者。

那么什么是工程师的影响力？它是如何体现出来？

我不想重复讨论有关性别、种族和阶级的相关问题，我只是认为，从某种程度上讲，某些人掌握权力要比其他人更难。

## 创造的力量

行动是工程师的超级神力。我们在用笔记本电脑和自己的大脑创造事物！这太不可思议了！我们不必说服、忽悠或强迫他人为我们做事情，我们亲力亲为。

这看起来很浅显，但却很重要。创造是一切力量的源泉。除非我们同意，否则什么都不会发生。

Facebook 的一张海报上写着“CODE WINS ARGUMENTS（代码胜于雄辩）”。这句话说得太过绝对了。不过，有多少次，一场又一场的技术争辩都是因为有人愿意动手去做而得到解决的？行动避免了争辩，行动是对理论最好的证明，行动就是力量。（当然，“做”并不仅仅是指“写代码”。）

此外，开发软件是一项创造性的活动，而大规模的开发协作就更是一项非常具有共通性的活动。作为一种创造性行为，当我们对自己的工作充满动力、灵感和热情时（与砍伐木材相比），才会成为更好的创造者。作为一种协作行为，当我们拥有高度信任感和社交凝聚力时，才会做得更好。

工程能力和判断力、自主性和目标感、社会信任感和合作行为，这些都是伟大工程的基础。每个人都有一两种他们感到最舒适的模式，我们可以将这些模式大致分为几种原型。

## 影响力原型

“选择极其困难且迫在眉睫的工作（也往往极度无聊）”。SOC2 合规性、备份和恢复、可怕的重构、认证集成：只要能够推动业务发展，他们才不管这些工作无不无聊。如果你是这样的工程师，那么你将获得别人的尊重和感激。“杀手锏调试器”。通常是工作时间最长或最初构建系统的老工程师。如果你对自己的历史和背景感到乐观，那么它们将成为你的一笔巨大的资产。实际上，人们倾向于高估这种人的不可或缺性，但我不鼓励这样做。

“专家”。如果你是某个方面的技术专家，那么

你在自己的领域里将会产生巨大的影响。你应该在自己的领域内与时俱进，这样才能保持你的优势。

有些人持续提供输出，强大到令人感到费解，有时甚至会在多个方面同时取得进展。有些人长时间工作，有些人对如何最大限度地发挥影响力有着本能的反应。没有人想要惹恼这些人。他们的参与通常会加快一个项目的进度，或者让项目在终点线上挣扎。

并非所有的影响力都源于原始技术实力或产出，只是一小部分创意、协作、人际关系是占据优势的：

有些人怀有好奇心，并且总能比其他人提早一步嗅到新事物的味道。他们似乎在玩弄一些毫无意义的东西，你很想骂他们，但他们却可能救你于灾难之中，所以你不得不学会珍惜他们的这种“玩物丧志”。

有些人通过社交的方式来解决问题，比如交友、交换技能、互相帮忙等。不要低估了这种社交方式，它往往是为问题找到正确答案的最快途径。

有些人非常懒惰，并且通过他们所谓的优雅的捷径来打击你。

有些人是招募磁铁，这种人是值得重金聘请的，因为所有人都喜欢与他们共事。

- 有些人擅长推动利益相关者达成共识。
- 有些人擅长表达、讲故事或教育他人。
- 有些人是人人都想要成为的榜样。
- 有些人很会画饼，让每个人都心甘情愿照着大饼去做事。
- 有些人将代码评审变成了教学艺术形式。
- 有些人会让身边的每个人都更有成效和更有效率。有些人创造了无限的前进动力。有些人善于说不。

还有一些特殊的影响力是通过以下这些形式表现出来的。

- 曾经做过管理者的工程师是非常珍贵的。他们懂得如何为初级工程师解释业务目标，并让他们深信不疑（在初级工程师看来，这些



是纯粹的管理者所不具备的东西）。他们拥有强大的技术领导力，他们可以将项目分解为组件，让其他工程师能够赶在截止日期之前完成项目却不会让他们精疲力竭。

- 有些工程师是怎么也甩不掉的顽疾，他们质疑和挑战每个系统和组织架构。但他们也可能是可以打磨成优秀人才的好苗子，只是需要强有力的领导把他们的能量引导到富有成效的对话和改进上，并防止他们影响整个团队。
- 我们不要忘了轮班待命的工程师。如果你的公司有健康的轮班待命文化，那么产品所有权就会形成一种权力和道德权威——提出需求、推动变革、优先安排事项。轮班待命不应该是人人避之不及的东西，而应该是每个代码工程师应该肩负的荣誉勋章。（但它不应该让人感到痛苦不堪或影响他们的生活。）

这样的例子我可以说上一整天……工程包含了如此强大的角色和技能，所以我们有必要揭开影响力的面纱，并了解他人如何看待你的影响力。

大多数影响力的形式可归结为“影响和被影响”只是会写代码是不够的。你可能拥有信誉度，

但拥有它与使用它是不一样的。要将影响力转化为真实的力量，必须使用它，而使用它的最好方式是沟通。

藏在你脑子里的东西对其他人是没有影响力的，你必须把它们表达出来。

你可以通过多种方式实现这一目标：通过写作、小组谈话、公开招募盟友、说服权威人士、公开表达想法等等。

因为工程是一项创造性的活动，所以独裁主义实际上是非常脆弱和具有破坏性的。唯一可持续的权力形式是所谓的“软实力”，如影响力和激励能力，这就是为什么优秀的管理者喜欢使用软实力，而非不情愿使用权力。如果你的领导经常强调他们的权威，那就是一种反模式。

如果你不发声，就无法发挥你的影响力。在他人面前说出真实想法可能也会把自己的弱点暴露出来，但有时候也不一定是这样。

## 这不是一个“零和游戏”

你们当中的大多数人拥有的潜能远远超出自己的想象，因为你们感觉不到它们的存在，或者意识不到自己在做什么。

管理者可能拥有硬实力和权威，但是有关技



术交付的具体决策通常是由他们身边的工程师做出的。这些工程师都属于行动者，因为他们就是需要为这些决策后果提供支持的人。

权力倾向于流向管理者，因为他们了解更多信息。因此聘请了解这一点的管理者并借助它来向其他人行使权力就变得非常重要。

在健康的支配与臣服关系中，臣服一方通常拥有最终权力。同理，在健康的团队中，工程师实际上拥有最终权力。因为你有最终的否决权：你可以拒绝参与贡献。你通常可以另寻更好的下家，或许很多人都应该这样做。

当技术和管理发生冲突，谁会赢？理想情况下，他们需要一起努力为业务和人员寻找最佳的解决方案。在身处水深火热的团队中，他们两者之间反而能够保持紧密的联系。

## 选择你的战斗

如果你能够正确地培养和发挥自己的影响力，你就可以对构建的内容及其构建的方式有很多发言权。但你不可能对一切事情都有发言权，这不合逻辑。

你越是将影响力用在好的产出上，就会积累越多的影响力。但它是一种非常精确的工具，需

要以精确的方式来运用。想象一下，在按摩时，按摩师整个人压在你的背上，而不是用他们的肘部或手掌来按压某个部位。过于宽泛的目标会分散你的注意力，并限制你的潜在影响力。

所以，要正确地运用你的注意力。

一旦你有了影响力，别忘了帮助别人提升影响力。注意那些被忽视的人，帮助他们提升他们的影响力。借出你的时间和信誉度，把那些让你变得强大的技能也教给需要它们的人。



## SOA旅程：从了解业务到敏捷架构

作者 Vadim Samokhin 译者 姚佳灵

我想读者们都很清楚为何、何时和是否应该拆分一个单体。但是，为了以防万一，给大家提个醒：我们的共同目标是业务敏捷。如果这个单体无法满足业务的需求，如果开发的步伐减慢了，那么肯定要做些什么来解决问题。但是，在这之前，很显然，你需要找出原因。从我的经验来看，原因总是相同的：高耦合和低内聚。

如果你的系统属于一个单独的边界上下文，如果它不是足够大（听起来有点模棱两可，后面我会详细说明），那么你要做的就是以正确的方式把系统分解成模块。否则，你需要引入更加自主和独立的概念，我称之为“服务”。这个术语也许已经在整个软件行业中被用得最多的，因此，我要澄清一下我的意思。

我将进一步给出更严格的规定，但就目前而言，我想指出的是，首先，服务具有逻辑边界而非物理边界。它可以包含任何数量的物理服务器，这些服务器可以拥有后端代码和UI数据。在这些服务中，可以有任意数量的数据库，它们可以具有不同的模式。

### 识别服务边界的错误方法

在深入研究什么样才是良好的服务之前，请让我指出在定义它们的边界时最常遇到的错误方法。

#### 实体服务

将一个系统拆分为实体服务是一种经常被用到的方法，它被称为反模式。有时候，这个方法源于对重用的痴迷：所有跟某个实体相关的东西，要完全可重用，那是天赐良福！但是，在作为一个主要动机，代码重用其实是个悖论。我相信对于服务来说也是一样的。

因此，以下是这种方法的缺点：

- 紧密耦合：每个服务都是其他服务的客户端。因此，如果有一项服务发生变化，你就必须测试整个系统。
- 通信频繁：它们有大量的内部通信，通常是同步的。这让系统变得脆弱，让速度变慢。
- 大量的服务：难以理解整体情况，也难以跟踪请求。
- 糟糕的封装：业务规则遍布整个系统。通

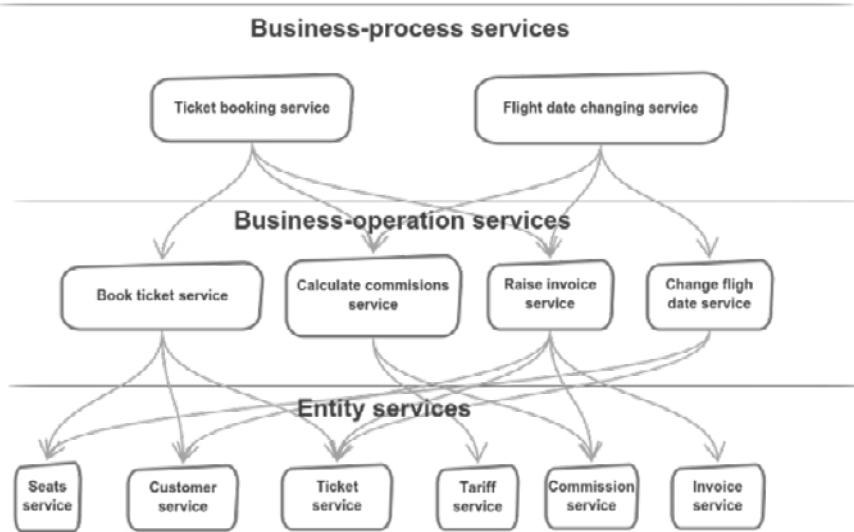


图1 这是实体服务通常看起来的样子

常，每个客户端在更新其他服务的数据前要做一些检查。同时，更新是与更新查询一起进行的，数据和行为被分开了。

- 同步：特别是基于http时，后面我会仔细讲它的缺点。

### 盲目将模糊的业务架构映射到技术架构

我常常遇到的一种方法是：“我们的任务就是要写代码交付产品，让我们开干吧！”好吧，对于小型项目，这是可以的，但是，对大一点的项目就不可行。它不可避免地带来业务和IT之间的阻抗，从而导致业务敏捷性降低。并且，如果你不够敏捷，那么你就出局了。

下图是与业务功能不一致的技术架构。

这里有两个问题特别引人注目。首先是技术功能1的紧密耦合。由于其中有两个业务功能，它们有可能具有不同的开发步骤、不同的可扩展性和可用性需求，而且很难把它们分开。其次，在技术服务之间会有一个间隙（使用“Oops!”标记的地方）。当在这个间隙中出现新的需求时，这两个技术服务会耦合得非常紧密。我会把这个系统称为分布式的整体，尽管这个术语已经被预留

给另一个不同的问题。

从缺点数量上讲，这个反模式绝对是领先的，甚至不知道该从何说起。不管怎样，我们还是试着来讲讲吧：

1. 同步通信是资源密集型的：接受请求的服务等待第2个服务，而第2个服务等待第3个服务，依次类推。
2. 昂贵的扩展：需要扩展每一个服务，而不是真正有需要的服务。
3. 分布式计算的8个谬误：Nuff在这里讲过。
4. 它是不可靠的：只要有一个服务宕机，整个系统也跟着宕机。
5. 一致性问题：整个系统可能最终处于一个不一致的状态。我认为你不会想使用分布式交易，对吧？如果用到了，那么请记住下面的内容（假设是两阶段事务）：
  - 两阶段提交事务本质上是脆弱的。
  - 整体可用性降低：如果一个子事务被拒绝，那么整个请求也就被拒绝。
  - 额外的操作复杂性。真的，去问问你的系统管理员，如果他们在这方面有经验的话。

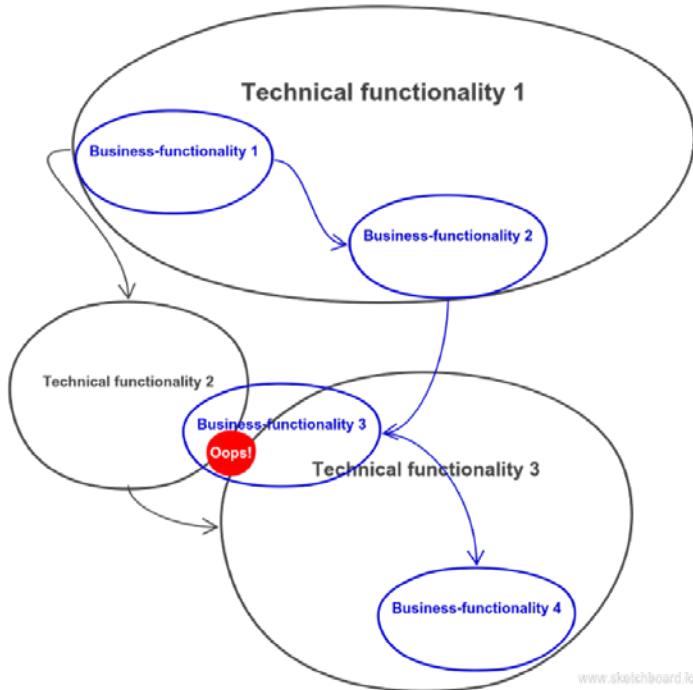


图2 与业务功能不一致的技术架构

- 不断增长的通信延迟。
- 资源被锁定在各阶段之间。如果第2个阶段一直没有发生呢？这代表了严重的扩展问题。无论怎样扩展，等待锁定资源释放的时间一直存在。

为了公平起见，我应该提一下三阶段提交事务和共识协议在解决上述问题上取得了不错的进展。

### 带有命令消息传递的服务

与 HTTP 相比，从可靠性和资源消耗来看，消息传递向前迈出了巨大的一步。但是，由此产生的服务仍然是耦合的。为什么呢？

首先，当服务 A 通知服务 B 去做某事时，服务 A 显然是知道服务 B 的。因此，如果服务 A 要通知服务 C 去做某事时，我们当然要修改服务 A。

其次，可能是最重要的，服务 A 希望一些来自服务 B 的行为。基于这类通信的特点，服务 B

在服务 A 的上下文中执行某些工作。因此，如果对服务 A 的需求发生变更的话，服务 B 也要做出变更才行。现在，假设服务 D 需要使用服务 B 的功能。但是，服务 D 有其自身的上下文，对服务 B 有自己的需求。很可能服务 B 需要做出一些变更来满足服务 D。在这一切完成之后，我们需要确保这个变更没有破坏服务 A 的功能。这是一个经典的紧耦合噩梦。

### 中心化的数据

在最简单的场景中，“中心化”这个术语是指具有任意数量服务的单个数据库。它的主要缺点是，对以某种方式修改中心化数据的服务逻辑做出变更，有可能会破坏其他请求这些数据的服务。是什么原因呢？因为，现在这个基础数据是根据不同的规则在操作，所以数据流不同了。

但是，一般说来，这个术语适用于不同逻辑的服务访问彼此数据的情形。它常常和一个实体服务反模式一起使用。它们的缺点非常相似。

我相信，一个良好的服务和一个良好的类有一些共同之处。其中一个共性是数据和行为永远在一起。这导致不可能绕过由类接口提供的行为进行随机的数据变更。同样的原则也适用于服务。通过中心化数据，我们通常把服务的接口变成CRUD（C-creat, R-retrieve, U-update, D-delete）。我们把行为和它的数据分开，把中心化的数据服务转变成数据库。

## 服务编排

这个方法意味着只有一个地方可以路由所有入站的消息或请求。这意味着该服务与其他服务之间的通信是同步的，以及随之而来的所有缺点。此外，业务逻辑必然会渗透进来。因此，它不属于自己某个单独的服务，而是分布在两个服务之间。它类似于智能管道方法，这个方法已经被证明完全不是个好方法。

这是常见问题的一个例子。有一种名为“进程管理器”的模式被证明在小范围内运行良好。服务编排尝试将这种模式应用在系统范围内。同样，CQRS（Command Query Responsibility Segregation）不应该是高级模式，事件源也一样。SOA应该也是。

## 根据组织结构定义服务边界

严格地说，这样做不一定是错误的。在一个理想世界中，企业以最有效的方式组织而成，它运作良好。但是，在现实生活中，政治悄然而至。因此，要非常谨慎地使用这种方式。不过它可以为我们带来一些好的线索，至少有一个好的开始。

## 围绕层次定义服务

层次本质上是紧密耦合的。因此，围绕层次而组织的服务在本质上也是紧密耦合的。可以考虑一下垂直切片。

## 我希望我的服务具备什么样的性质

因此，在讨论识别服务边界的方法之前，让我们先快速勾勒出要达到的目标；

在深入研究构成我对服务理解的价值观（遵

循Kent Beck的极限编程方法论），也即定义我对服务边界思考的主要动力之前，我想提出一些我的服务一直在遵守的原则。我把它们分为主要和推断两类。让我们从主要原则开始。事实上有两个主要原则。

## 松散耦合

几乎所有服务都存在的一个问题问题是紧密耦合。你无法自由地修改任何服务的实现，因为你永远不知道哪个服务依赖于此服务。这就是所谓的松散耦合，这也正是我一直想摆脱的，不只是在高级架构层面，也包括在代码层面。

## 高度内聚

在“识别服务边界的错误方法”一节中所描述服务都具有低内聚性。这些服务的数据和行为遍布整个系统。内聚性都跟功能有关：当我提到“内聚”时，我的意思是“暴露非常具体的行为”。维基百科也是这么说的。而且服务还具有封装性，封装通常也指信息隐藏，但是这种概念并没有被普遍接受。

推断原则如下。

## 正确的粒度

当粒度太粗时，一个服务可以被分为若干个其他内聚服务。当粒度太细时，一个服务迫切需要其他服务数据或功能来进行操作，因此，耦合变得紧密起来。

## 高度自治

松散耦合带来了服务自治的概念。自治是指一个服务的可用性不依赖其他服务的可用性。为了完成自己的工作，一个服务不需要其他服务的功能或数据。此外，一个服务甚至可能不知道其他服务（在大多数情况下不应该知道）。

## 通过事件进行通信的服务

服务并不是存在于真空中，因此，它们之间需要通信。如何实现呢？我主张使用以行为为中心和基于业务驱动的事件消息类型，反对使用同步请求和命令消息。这样的架构被称为基于事件

驱动的架构。发布的事件应该反映业务概念，在领域中已经发生过的真实事件：订单已完成、交易已处理、票据已付款。

## 分散的数据

当服务是松散耦合、高度内聚的，那么也就是自治的，它们根本不需要彼此的数据，数据自然而然地变得分散了。

## 服务编排

服务编排是拒绝使用同步通信、采用业务事件和拒绝使用中心化数据存储的自然产物。

## 如何定义服务边界

那么，如何识别服务，让它们最终可以成为松散耦合和高度内聚的呢？换句话说，可维护的、可靠的和具有业务一致性的架构的核心价值是什么呢？

## 业务能力和业务服务

首先，我们来介绍业务能力的概念。业务能力是组织为保持自身运作和运营能力而做的事。它是对整个企业功能的具体贡献。它是组织为实

现其目标而拥有的具体功能或能力。采购经理购买货物、货仓保存货物，销售员销售货物，财务人员计算利润。因此，业务能力对涉及相同业务的不同公司来说几乎是一样的。它们的实现是不同的。业务能力实现所在的逻辑边界称为业务服务。那么，这其中有些什么呢？有业务策略、业务规则、业务流程、参与其中并做出具体决定的人员、这些人员使用的应用程序。下图是意象图。

在业务能力和相应的业务服务之间应该始终存在一个双向关系。

我认为，比较业务服务边界和接口是非常容易的。这两者都是通过功能的声明性描述进行定义的：它们不说它们会如何完成它们的工作，而是告诉我们它们能做什么。

总而言之，业务服务边界和业务能力这两者都是声明性的概念，很少会发生变化。

## 业务服务如何与其他服务交互

业务服务的通信，或者，更具体地说，这些业务服务的业务流程的交互是通过业务领域事件来定义的。业务服务和其他业务交换的事件形成了它们之间的接口或合约。这些事件可以以电话、电邮或简单对话的形式实现。

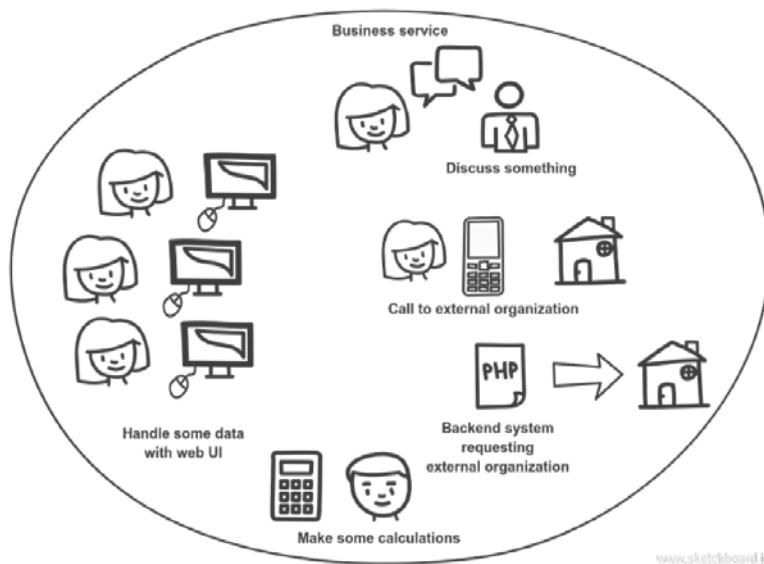


图3 意象图

## 业务服务和技术服务

因为业务服务边界在整个企业中是最稳定的，所以围绕它们构建技术服务就很有意义了。电话用 RabbitMQ 事件替代，纸质文档以 web 界面替代，计算由 HTTP 调用替代，而其他一些乏味的工作则由 ML 来做。通过这种方法，我们获得稳定的服务合约：业务服务很少发生变化，因此，由相应的技术部门实现的合约也很少会发生变化。

以这种方式确定的技术服务充分尊从单一事实来源的概念。这个来源代表了特定的事件，也代表存储在单个位置的数据。这很重要：在技术服务中，应该有单个逻辑位置来发布具体事件。这些事件不应该包含大量数据，重点不是要通过这些事件来获得数据。它们只是通知，告知我们某个事件发生了。此外，如果你的事件包含大量的数据，那么，你的服务边界可能出错了。事实上，这些有着大量数据交换的服务应该单独成一个服务。另一方面，这些事件应该包含足够的数据，即它们应该是完全自足的。但是，它们不应该有任何引用，因为这样会引入紧密耦合的共享数据库。

Udi Dahan 恰如其分地总结了这一点：“服务是具体业务能力的技术实现，任何数据或规则都只能由一个服务拥有。”

这让一个常见的错误浮出水面——物理和逻辑架构必须是一样的。实际上，它们不必一样。如果有个需要我们去集成的 web 服务，它不一定是一个成熟的业务服务。

## 业务能力映射

这是一种旨在促进服务边界识别的技术。

首先，你应该确定你的高级功能，它们一般是组织的核心功能。由此产生的服务可能是单独的业务。因此，它们可以被外包出去，或者相反，被收购。在定义这个服务时，你应该问“什么”和“如何”这两个问题，并自己确定声明性的功能。要做到这一点，必须和不同的利益相关者进行大量的沟通，以听取所有观点。和那些做底层工作的人沟

通，找出他们所参与的主要流程以及他们实际上在做什么。

每个服务应该对应单独的短语，就像“我<动词><名词>”这样。比如，“我处理付款”，或者“我检查欺诈付款”。每个这样的服务都是一个交付业务价值的步骤。

了解企业如何赚钱，可以为我们提供一些线索。如果你对一个组织如何寻找潜在客户、这些潜在客户如何成为真正的客户、他们何时以及如何购买何物有一个清楚的认识，那么，你也许已经完成了高级服务识别。

组织结构虽然可能具有误导性，但也有可能会带来帮助。只是不要期望一个企业能够安排得完美无缺，但更高层次的整体组织视图肯定是有用的。

寻找能够对某些功能进行自动化的方法也是有帮助的。这样可以让我们远离实施细节，有助于找到有用的抽象。

不要忘了找出业务服务之间的交互方式。因此，正如我已经提到的，每件事都重要：电话、电邮、消息、普通的对话。

在完成这个更高层次的概览之后，深入到每个服务。这个过程本质上是一样的。但是，我有个建议，A 永远不能将整个服务边界识别过程看成是一个主要步骤或阶段，因为这样就变成瀑布模型了。这个过程与开发很接近。尽管在编码前都需要做一些初始的工作来识别顶层服务，但如果我没有参与编码，我就不会深入其中。

因此，业务服务交互的总体情况（用箭头表示事件）看起来如下图所示。

## 价值链分析

我用这种方法来处理业务能力映射。基本上，它可以归结为以下几个方面。首先，将你的组织看成是一组从具体实现中抽象出来的业务功能。其次是我已经提到过的：这些功能是实现企业主要业务目标的步骤。

可以沿着这条线进行发现服务的对话（按照相反的顺序，可以从业务目标回到最初开始的地方）：

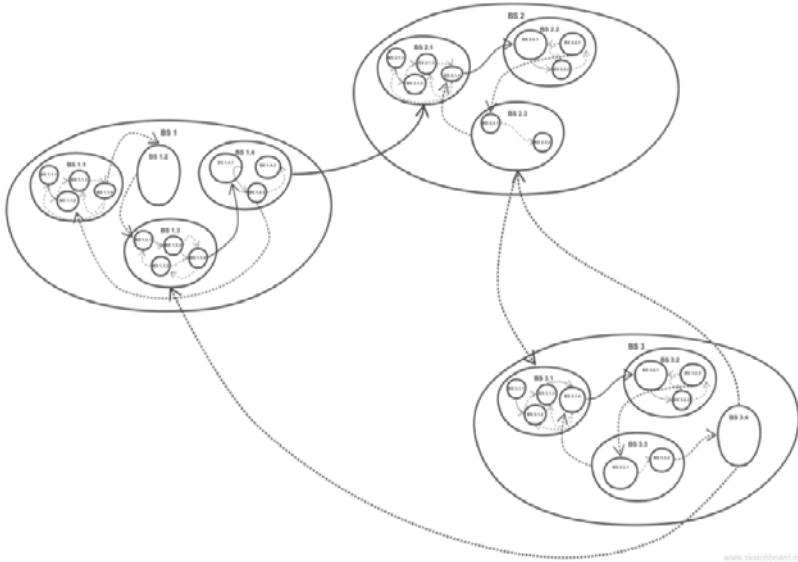


图4 业务服务交互的总体情况

- 你的主要业务目标是什么? (或者, 换句话说, 你怎样赚钱?)
- 我们销售家具。
- 你给客户送货吗?
- 没有。
- 你从哪里获得这些家具? 从哪里购买或自己生产的吗?
- 我们生产家具。我们在米兰有工厂。
- 原材料从哪里来?
- 我们的家具是用我们自己种植的树木制作的, 用到的紧固件是从别人那里采购的。

这里的主要能力可能是下面这几个: “提供原材料”、“制造家具”、“销售家具”。我已经省略了“家具仓储”和“市场营销”, 因为它们已经无处不在。如果公司提供送货服务, 那么还需要加入“家具送货”服务。

在这些类型的业务中存在两个非常普遍的链: 供应链和需求链。这两个名词本身已经很直观了。此外, 还有一个方法混合了业务能力映射和价值链。毫无疑问, 它就是能力链, 它主要体现在成果图表上。这里有个很好的例子。

## SOA和DDD

我假设你很知道什么是边界上下文。但让我感到困惑的是, 我不知道如何定义它们。例如众人皆知的产品目录总是让我一头雾水。我想知道这概念背后的理由。现在我能够回答这个问题了: 边界上下文就是一个业务服务。需要补充说明一下, 只显示某些商品的产品目录很少会是一个成熟的服务, 它不具备任何行为。业务服务肯定要实现一些行为或一些业务功能。目录实际上只是 Backend for Frontend 模式一个例子。它只是从其他服务中获取数据, 并展示出来。

**Backend for Frontend**示例。在 SOA 领域中, 它不是一个成熟的服务。

## SOA和康威定律

在尝试影响组织结构时, 我把康威定律当做目标。该定律表明, 我们注定要产生一种设计结果, 也就是组织沟通结构的副本, 因此, 我的目标是识别并优化那些沟通路径。所谓最优结构, 就是指最具内聚性的结构。理想情况下, 这些沟通路径应该属于同一业务服务中关系非常密切的

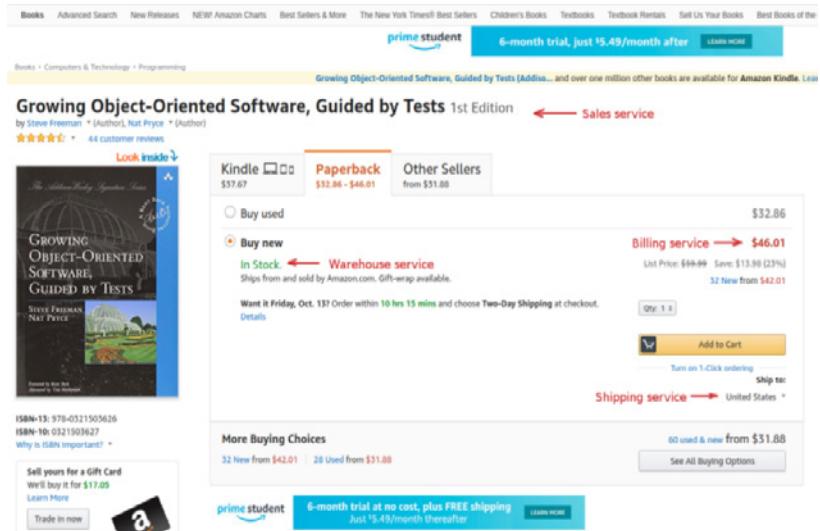


图5 Backend for Frontend示例

一组人。因此，围绕业务服务形成组织结构就完全合理了。这个方法根本不是什么新东西，但很少会遇到。按这种方法构建的组织，其内部单元会是内聚的、自治的和可替换的。同时，企业会变得更敏捷。

高度内聚、松散耦合和封装是自然的基本特征

我在 OOP 领域工作了很长一段时间，实践 XP，并创建 SOA 架构，我总是觉得它们有一些共通的东西。

瀑布模型中的阶段对应软件中层，这些层之间需要依赖彼此的数据，因此，它们天生是紧密耦合的。相反，XP 放弃了阶段的概念，使用了非常短的周期，把所有的活动结合在一起，其中包括和领域专家的交流、开发、单元测试、功能测试、客户反馈。Sprint 是高内聚的，而且可以做到不那么紧耦合。它们持续形成了 bizDevOps 文化。

过程编程是指实现一系列计算步骤的程序，并操作数据：进行这个操作，然后进行那个操作，接着进行另一个操作。这种方法无视数据封装。整个概念意味着数据和程序是分离的。这种方法

与实体服务反模式产生了共鸣，不是吗？相反，OOP 就是封装。适当的对象就像有责任能力的成年人一样，具备完成工作所需要的一切。我的业务服务也一样。它们不暴露数据，而是暴露行为，并通过事件来通知他们的工作进展。

康威定律意味着我们要围绕业务服务创建内聚的沟通结构，而不是由开发人员、QA、业务分析人员等组成的层级组织单元。

所有这些特征都是与生俱来的。原子由质子、中子和电子组成，它们都有自己的行为和需要遵守的规律，但是只作用于一个原子，这样形成了非常内聚的封闭微系统，与其它东西之间只有松散的耦合。而 XP 是一种持续的改进，可以用像 OOP 和 SOA 这样的工具来加强，进化是大自然的本质。

最后，关于软件的核心价值，我认为关键的是构建正确的抽象。它表现在各个层面、各个阶段上。也就是从高级别的 SOA 开始，通过 XP 的持续改进和反馈，识别主要的业务能力，并围绕它们形成技术服务，获得能够反映域驱动设计（Domain-Driven Design）和 OOP 的正确抽象。这就是如何在日常工作中通过业务 -IT 的一致性



来达到业务敏捷的方式。

例子？

第一个例子属于支付服务提供商领域。这篇博文包含了一些对伸缩性、sagas、聚合边界、组合UI和CQRS的一些想法。第二个例子属于比较普遍的电子商务领域，提供了一些正确识别边界的技巧。第三个例子是更加底层的示例，以RabbitMQ为主。

随着时间的流逝，我注意到我的服务的粒度变得越来越粗，saga包含了实体的整个生命周期。对于我的第一个例子中的金融交易，我可能会做一些与众不同的事。我会创建一个saga，它属于一个单独的服务，具有以下高级的阶段或状态：交易已注册、已通过交易欺诈检测、交易已处理、交易已协调、交易已完成。可能有一天，我甚至会写一篇属于我自己的“单体保卫战”博文。

Vadim Samokhin 是俄罗斯领先的临床研究公司Gemotest开发部门的负责人。他曾经涉及的领域包括电子商务、支付解决方案和卡片采集。他热衷于OOP、SOA、敏捷方法、通过业务-IT的协调实现业务敏捷。他偶尔会在medium.com/@wrong.about上分享跟上述主题有关的想法。



## AIOps实践思考：AIOps如何与APM结合？

作者 高驰涛

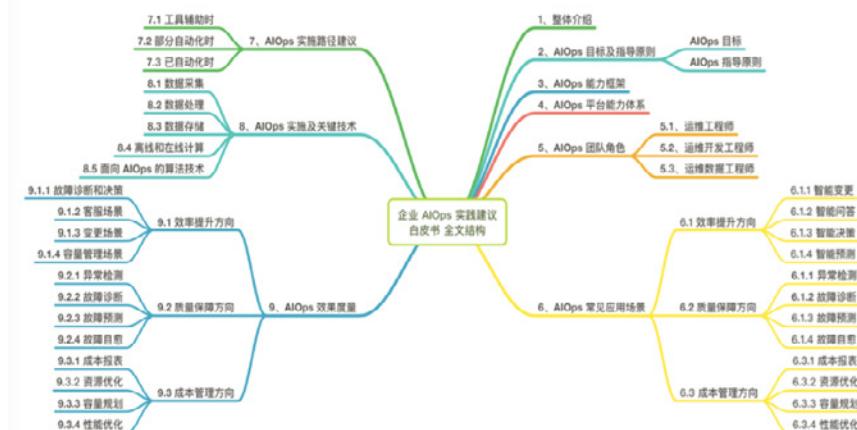
### 背景

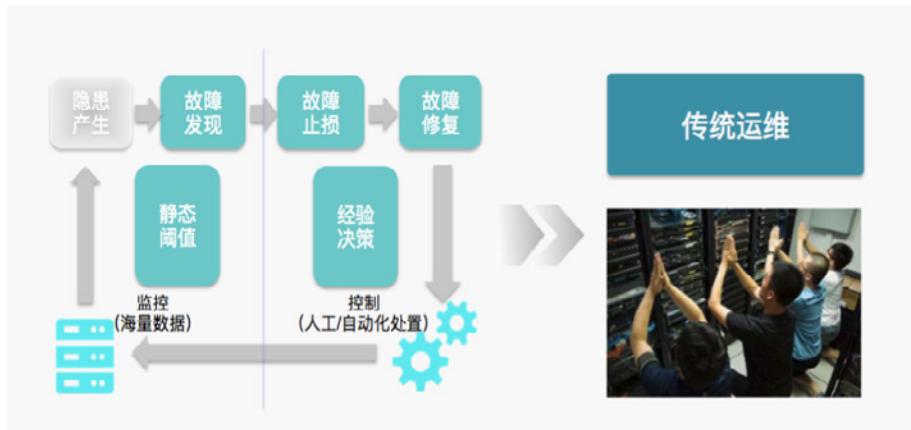
2018年4月13日由BATJ、360、华为、云智慧等众多互联网企业参与标准制定工作的《企业级 AIOps 实践建议》白皮书中提到：AIOps 即智能运维，其目标是，基于已有的运维数据（日志、监控信息、应用信息等），通过机器学习的方式进一步解决自动化运维所未能解决的问题，提高系统的预判能力、稳定性、降低 IT 成本，并

提高企业的产品竞争力。Gartner 在 2016 时提出了 AIOps 概念，并预测到 2020 年，AIOps 的采用率将会达到整个运维行业的 50%。

为什么要使用 AIOps 呢？它的价值到底在哪里？我们先来看一下通常情况下运维同学们是怎么处理一个故障的。

- 18:00 开始出现隐患；
- 18:30 业务正常，隐患影响范围在系统可承





受范围内：

- 19:00 隐患超出了系统可承受的正常范围，开始转变为故障；
- 19:30 部分业务出现异常，部分客户受到了影响；
- 19:40 异常指标超出监控阈值，监控系统发出告警；
- 19:50 运维人员登录系统开始排查问题；
- 20:20 运维人员找到故障原因，开始处置，此时该故障可能已经影响了大量用户；
- 20:30 故障恢复。业务恢复。

这是传统运维的处理模式。在整个故障从隐患产生到恢复的过程中，有几个明显问题：

- 监控数据获取的问题  
监控工具繁杂、无效数据充斥  
大量人工配置、缺乏问题的第一级识别
- 隐患发现的问题  
静态阈值：潜在隐患不能及时发现、重大隐患处理延迟

信息检索和甄别难度大：海量信息难以迅速定位原因

- 告警的问题  
事件处理效率低：历史告警无模型  
缺少事件压缩和升降级：大故障时消息洪水带来副作用
- 处理的问题

缺乏有效的数据依据：严重依赖人脑和经验  
处置后无法评价：只存在故障的是与否，一旦波动束手无策

## 实践AIOps的常见场景

除去这四类明显问题之外，当然还有更多的问题。比如：复杂多变的软硬件环境中，如何利用海量且有高价值的监控数据来保障业务高效安全运转？在这过程中，运维规则同样也是灵活多变的，又如何有效地进行管理和维护？针对这些传统运维的痛点，不难抽象出以下几个典型的场景。

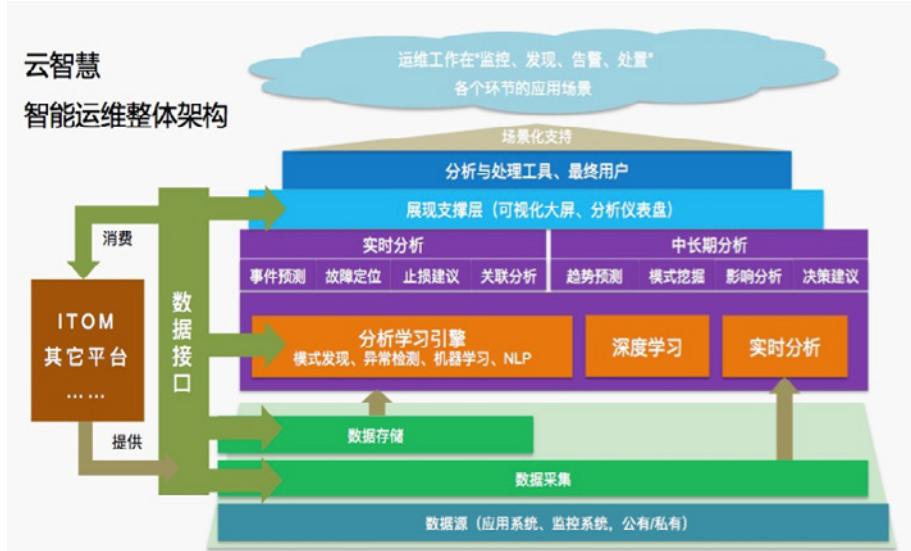
### 智能异常检测

这里我们提到的异常检测，特指从海量的运维监控数据指标中，针对时间序列类型数据指标的不正常问题发现。简单说，即发现历史数据中与大部分对象不同的离群对象，这不同于依靠人来判断的指标评价，能够更有效地提升发现问题的准确性和时效性。

### 智能告警

使用历史数据学习得到的动态阈值替代静态阈值，更及时地发现重大隐患或故障。

智能的告警消息相关性分析和收敛，解决故障发生时，告警风暴带来的副作用。通过对告警



消息的相关性分析，可以识别出告警的模式，将多条相关告警合并或转化成一条具有更多信息的告警，从而帮助更快更准确地诊断故障。

### 智能故障根因分析

在故障管理的检测、定位与识别的三个阶段中，故障的识别和诊断尤为重要。根因分析也被称为故障定位、故障隔离或警报 / 事件相关性，是推断产生一组给定症状的一组故障的过程。根因分析要求必须使用一个解释故障和症状之间关系的模型来执行这个推理过程。

### 智能时间序列预测

基于海量的历史数据习得模型，对未来的趋势的变化进行预测，并在生产过程中持续不断的进行模型的补偿修正，同时可以实现故障或事故发生前较准确的提前预警。

在这一系列典型的场景中，可预期的输出结果有影响范围、原因概率和影响概率、具体的某个类型的对象实体。而要求输入的数据能够满足以下几个方面。

- 足够存量的数据以及足够的数据增量
- 只有足够存量的数据才有条件去进行模型的

### 训练

只有足够的数据增量才有条件补偿修正训练得到的模型

- 数据维度覆盖度要（时间维度、地域维度、系统级维度、应用级维度等）足够

时间维度的提高有助于发现指标的时间周期性和预测的准确度，而没有足够的时间跨度将因损失周期性而大大降低数据价值

- 地域维度

从业务角度讲，大规模复杂系统，网络层面的故障往往造成区域性的影响。如 CDN 的故障影响、骨干网络质量的影响以及区域性网络质量的影响。通过分布式主动网络监测和真实用户体验的监测手段，这引起看似不可预知的故障发生，也可以在刚开始影响用户时就及时地感知到隐患的直接影响范围，从而降低故障影响时间。

- 系统级维度

时至今日，业务系统的软硬件规模日趋庞大，而软硬件的组合和调用关系也更加复杂，大规模复杂系统一旦发生故障往往是一点带面的影响。同时随着子业务每的变更频率增大，相关联上下游的业务系统的性能和稳定性也会受到严峻的挑

战。在复杂系统中，因为某个或某些子系统的突变事件，往往形成多米诺骨牌型的连锁效应，而采集的数据如果拥有完善的系统级维度，也势必为隐患和故障的发现定位和预测带来便利。

- 应用级维度

一旦应用系统发生隐患或故障时，在问题的定位过程中，除去因物理和网络因素导致外，Server 端故障多因程序 Bug 或 SQL 使用不当而引起。在分析问题所需的数据中，出现程序运行缓慢或异常、错误发生时的代码执行栈、运行切片、SQL 调用详情、进程或线程锁等等信息，都将为故障的定位速度和预测结果带来直接影响。

- 数据间归属和关联标记

非监督算法得到的数据有时需要人工监督或半监督的方式来进行修正，而此时对数据分析工程师或业务人员的要求都会非常高，这只能在小数据规模或简单业务场景中可行，当面临大模的或复杂的业务场景时，是决不可取的。那么假如数据产生时就具备某种天然的联系，则必然会为模型的训练、校验和修正带来事半功倍的效果。

## AIOps为什么可以与APM相结合

前面已经列出了本文要解决的问题和需求，下面试图论述 AIOps 必须与 APM 相结合的必要性。

APM 的全称是 Application Performance Management（应用性能管理），是由 Gartner 归纳抽象出的一个管理模型。APM 管理模型中要求做到这样五个层次：终端真实用户体验、运行时应用架构映射、应用事务分析、深度应用诊断和分析报告。为了支撑这五个层次的管理需求，一套完整的 APM 系统需要采集至少以下几种类型的数据：

### 用户端体验数据

包括各种浏览器和 APP（原生或 H5 或混合模式）客户端下的首屏时间、DNS 解析时间、首包时间等指标、JS 错误、IP 运营商数据、崩溃分析、卡顿分析、HTTP 和 Socket 链接与请求时

间与错误率、资源加载等数据。更为重要的是基于 Session Path 得到的用户行为路径数据，这使得从客户端采集到的所有数据天然地获得了用户行为的属性 – 这对于海量客户端倾刻间产生的海量数据，是一个多么棒的消息！

### 应用性能数据

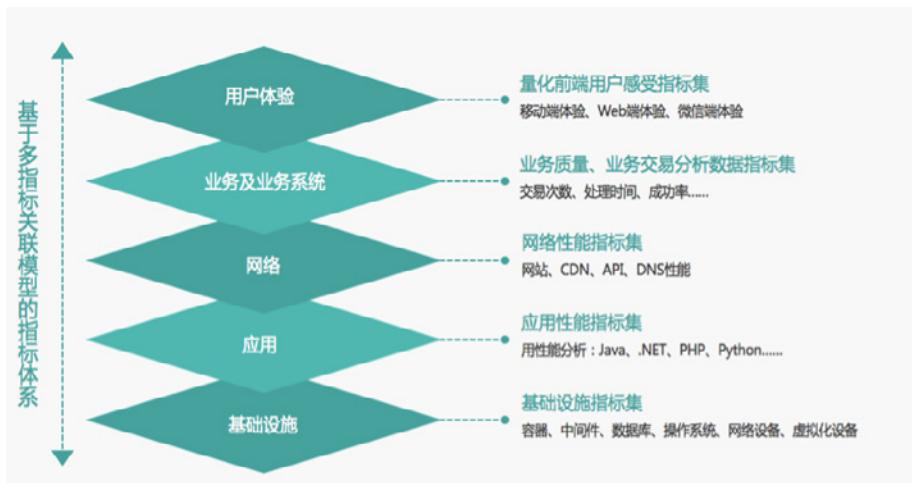
包括应用受访时的请求参数、执行时间、错误详情和错误率、异常详情和异常率、调用外部资源的时间和详情（如 SQL、MQ、API、RPC 等）、类和方法的执行时间与运行栈、虚机的状态数据（GC、Heap、线程池等）。更为重要的是基于 Trace 模型构造得到的数据间关系，这将发现应用与应用、应用与资源之间的关联关系。显而易见，可以通过这个关系快速地得到应用运行时的逻辑拓扑，同时也为即将进行的数据挖掘和预测提供强有力的数据依据。常见 Trace 模型包括 Google Dapper 和 Twitter Zipkin 方案，尤以 Google Dapper 最为常用。

### 服务器和服务的状态数据

这不同于通常的 Zabbix 等监控产品收集的数据，而是与应用、业务相关联，状态数据在产生时除去时间维度与应用业务间接相关之外，同样的由于 Trace 模型而变得与应用和业务直接相关。

### 端到端的数据联系

利用 APM 系统得到的数据实践于 AIOps，最有利的武器便是 Trace 模型。利用 Trace 模型得到的数据是具备了天然的数据联系的。基于 Trace 模型也很容易进行扩展，即将浏览器和 APP 客户端也加入到模型中。端到端指客户应用端到 Server 应用端的数据，在用户发起请求的最贴近用户侧产生唯一的 Trace ID、并通过 Request Header 或其他请求属性向 Server 端应用传递，于是用户端体验数据、应用性能数据、服务器和服务的状态数据，这三大类数据便有了天然的关系标记。



## AIOps如何与APM结合

通过 APM 和 APM 采集数据的简单介绍，不难看出实践 AIOps 所需要的数据需求，以及 APM 系统提供的各维度数据。在这个供求关系中，APM 系统提供的数据存量和增量足够满足、数据维度的覆盖度足够满足、数据间的归属和关联标记堪称完美。

我们再回过头来看，针对传统运维的痛点抽象出的几个典型场景，APM 系统提供的数据能否很好的应对：

### 智能异常检测

在 APM 系统中，关键事务是一个重要的需求场景。通过用户指定或系统习得的具备高频访问或至关重要的关键业务被称为关键事务，由于数据产生的时序性，在异常检测场景中，不仅可以很好地进行异常检测，也可以基于调用链的关系和用户行为来做故障的范围预测。

### 智能告警和智能时间序列预测

这两个典型 AIOps 场景对于 APM 系统提供的数据同样适用，并且由于数据间的系统级与应用级关系，模式识别变得更加简单高效，关系模型可以直接应用于告警模型的训练中，成功规避了场景中监督或半监督里最头疼的人为干预的难

题。应用智能告警收敛，AIOps 系统可以提供闪断、高频、阴断等多种告警压缩规则，基于算法削减无价值消息，缩短问题发现时间排除消息洪水中干扰。

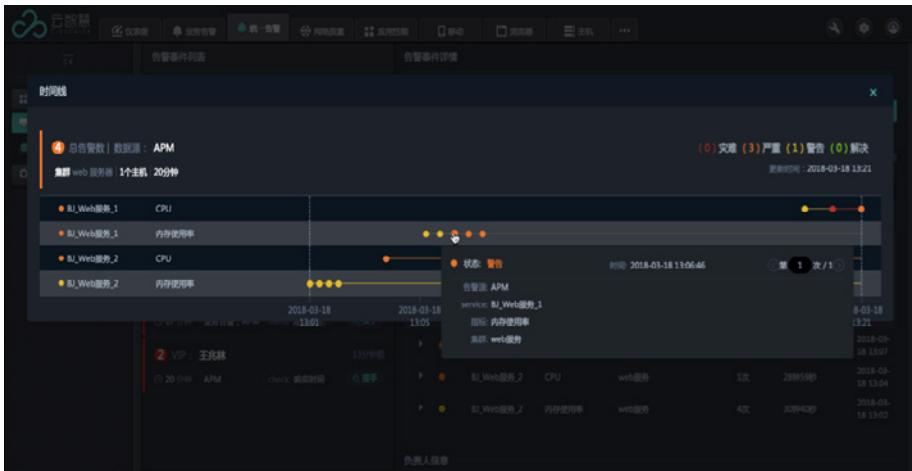
### 智能故障根因分析

前面已经较为详细地介绍了多种 Trace 模型，并且论证了因 Trace 模型而带来的数据间的天然关系。据 Gartner 多名分析师称：APM 系统实践 AIOps 最有利的武器便是 Trace 模型，它为分析问题提供了主线条。如果不用 APM 的话，应该怎么做呢？通常会根据人员经验或根据特定的业务场景，在应用程序中埋入追踪代码，即通称的“打点”法，这具有很大的局限性并因业务变更具有很大的操作难度，几乎不可能或很难进行标准化和产品化。

利用 APM 系统提供的数据实践 AIOps，从应用健康、用户体验或业务表现的外部视角来审视故障，如发现到某个具体的关键事务非常缓慢、某地域的用户受到了严重影响，关联诊断到最可能影响性能的代码段或 SQL 语句、应用服务器或中间件的某个节点 Load 或 IO 情况。

### 小结

通过本文的阅读，您可以获知实践 AIOps 时



利用 APM 系统的数据相较于传统的运维数据更为快速有效。APM 系统不仅向 AIOps 实践过程中提供足够丰富的数据以让 AIOps 平台更快适应企业的应用场景、为 AIOps 的实践过程提供了采集、处理和存储的关键技术基础，并可以为 AIOps 的实践效果验证和评价。

高驰涛 (Neeke)，云智慧研发总监，是 PHP 开发组成员，同时也是 PECL/SeasLog 的作者。早期从事大规模企业信息化研发架构，曾先后任职于易车集团和某大型微博营销平台，09年涉足互联网数字营销领域并深入研究架构与性能优化。2014年加入云智慧，致力于 APM 产品的架构与研发，对业务运维、智能运维有着独到的见解，崇尚敏捷，高效，GettingReal。

# ArchSummit 全球架构师峰会

## ▶ 实践

菜鸟网络 / 菜鸟仓储物流平台架构演进及技术挑战

京东 / 分布式BaaS的设计与实践

LinkedIn / 大规模机器学习在LinkedIn预测模型中的应用

阿里巴巴 / 深度学习在智慧餐厅中的应用

Pinterest / Build a dynamic and responsive Pinterest on AWS: a system engineer's perspective

Netflix / Netflix API Gateway that handles 2M requests per second

满帮集团 / 货车帮云原生平台架构设计思路和实践

## ▶ 分享嘉宾



Susheel Aroskar  
Netflix  
Software Engineer



李刚  
百度 可用性工程  
技术负责人



谷雪梅  
菜鸟网络  
CTO, 副总裁, 技术产品负责人



长纪  
阿里巴巴  
高级技术专家



刘波  
Pinterest  
Engineering Manager



郭平 (坤宇)  
阿里巴巴  
高级技术专家



张镭  
LinkedIn  
Engineering Manager



陈皓 (左耳朵耗子)  
Megaease  
创始人&CTO



杨巍威  
Hortonworks  
YARN/Staff Engineer

8折 报名中 现在报名立减1360元

联系我们: 电话: 17326643116 (灰灰) 微信: aschina666

会议: 2018年12月07–08日 培训: 2018年12月09–10日 地址: 北京·国际会议中心



▶ 程立 / CHENG LI

持续关注InfoQ好多年了。由于工作繁忙没有很多时间泡技术社区，我一直选择坚持精品与原创路线的InfoQ作为获得业界信息的主要来源。当遇到难题时也会到InfoQ上寻找灵感并常常有所收获，可以说InfoQ是我的老师、智囊和朋友，借此机会向InfoQ说声谢谢！

▶ 冯大辉 / FENG DAHUI

InfoQ，技术人都喜欢。几年下来，通过InfoQ网站获得了许多有价值的资讯，通过InfoQ的电子杂志借鉴到很多技术思路，而通过InfoQ举办的数次QCon大会，又结识了不少业界朋友。期待InfoQ 坚持自己的特色，期待越办越好！

▶ 洪强宁 / HONG QIANGNING

InfoQ是我获取业内最先进的技术和理念的重要渠道。在InfoQ的帮助下，我也得以与国内外众多技术高手交流切磋，获益匪浅。感谢InfoQ！

▶ 卢旭东 / LU XUDONG

是非成败转头空，青山依旧在，惯看秋月春风。一壶浊酒喜相逢，古今多少事，滚滚长江东逝水，浪花淘尽英雄。几度夕阳红。白发渔樵江渚上，都付笑谈中。

▶ 王文彬 / WANG WENBIN

InfoQ办的QCon大会是一个高质量的盛宴，对于最新的互联网技术和最佳实践一直在做探讨。除了邀请国内的牛人，也会有国外的大牛来做分享，对技术人员是一个不可错过的大会。

▶ 杨卫华 / YANG WEIHUA

InfoQ每年遍布全球的QCon大会是技术界的盛会，给业界很多研发方向上的启发，新浪微博的技术架构也从往届QCon大会演讲中获取了不少宝贵经验。

▶ 吴永强 / WU YONGQIANG

接触InfoQ，包括QCon，已经有好几年了，我非常喜欢它的风格，灵动、快速、实用，Moq网站、QCon、《架构师》杂志都能够紧贴互联网技术的发展前沿，带来大量的最佳实践，对我们这样发展中的公司的帮助非常大。希望InfoQ能够越做越好！

▶ 毛新生 / MAO XINSHENG

InfoQ社区是架构师的一流资讯来源，也是大家交流的桥梁。



扫描二维码  
了解QCon大会更多信息