

架构师

ARCHITECT



热点 | Hot

AWS re:Invent 2017 特别报道

推荐文章 | Article

阿里巴巴正式开源容器技术 Pouch

理论派 | Theory

从 CQS 到 CQRS

专题 | Topic

中小型研发团队架构实践



CONTENTS / 目录

热点 | Hot

Werner Vogels 脑中的未来世界 @AWS re:Invent 2017

理论派 | Theory

从 CQS 到 CQRS

推荐文章 | Article

阿里巴巴正式开源其自研容器技术 Pouch

观点 | Opinion

为什么说 Service Mesh 是微服务发展到今天的必然产物？

专题 | Topic

中小型研发团队架构实践：电商如何做企业总体架构？



架构师 2018 年 1 月刊

本期主编 徐 川

提供反馈 feedback@cn.infoq.com

流程编辑 丁晓昀

商务合作 sales@cn.infoq.com

发行人 霍泰稳

内容合作 editors@cn.infoq.com

助力人工智能落地

2018.01.13 – 01.14 · 北京国际会议中心

根据Gartner的预测，AI在2018年已经不是遥不可及的东西，每家公司都可以碰得到。到2020年，AI将成为CIO首要投资目标，深度神经网络跟机器学习会有100亿美元的市场。

那么，2018年，你是否已经做好准备转战AI了？应该去哪里学习现成的落地案例和实践经验呢？

InfoQ中国团队为大家梳理了目前AI领域的最新动态，并邀请到了来自Amazon、Snap、Etsy、BAT、360、小米、京东等40+公司AI技术负责人前来AiCon分享他们的机器学习落地实践经验，肯定可以给你一些启发和思考。

演讲嘉宾



颜水成
360人工智能研究院
院长及首席科学家



山世光
中科院智能信息处理重点实验室
常务副主任



喻友平
百度
AI技术生态部总经理



徐盈辉
菜鸟
人工智能部资深总监



洪亮劼
Etsy
数据科学主管



老师木
一流科技
创始人



王刚
小米
小爱语音交互系统负责人



林添
Google软件工程师
Tensorflow中国团队成员



张清
浪潮
AI首席架构师



杨建朝
Snap
研究院任主任科学家



胡时伟
第四范式
首席架构师



蔡超
Amazon
中国研发中心首席架构师

精彩案例 先睹为快

Amazon 机器学习在工程项目中的应用实践

第四范式 如何利用大规模机器学习技术解决问题并创造价值

菜鸟

双11：如何运用机器学习等AI技术实现物流优化

TutorABC

如何利用大数据和AI提升学习效果

小米

语音识别和NLP在智能音响中的实践

苏宁

智能机器人平台应用实践

爱奇艺

自然语言处理和视频大数据分析应用

淘宝

智能写手——智能文本生成在双十一的应用

售票倒计时进行中！

截至2018年1月12日前

团购享受更多优惠

售票咨询(电话)：18514549229 (同微信)

售票咨询(邮箱)：goupiao@geekbang.org



售票咨询



扫码关注大会官网

卷首语

程序员的中年危机还未到来

作者 徐川

2017年12月10日，中兴研发主管欧建新，在公司大楼一跃而下，在程序员群体里也掀起了对中年危机的讨论。

需要指出的是，当事人所处的行业是通信业，并不是互联网与IT业。

真正席卷行业的中年危机，是整个行业高度成熟，进入洗牌阶段、不需要那么多的研发来开拓进取才会发生，其它的，只能看做是危机管理没有做好的个案。

国内互联网高速发展已经有二十余年，在可预见的将来，还会如火如荼的发展下去。越来越多的行业主动拥抱互联网，越来越多的创业公司发生在互联网，只要这样的趋势持续，我们仍然有大量的编程需求，手里有真功夫的程序员不会愁没有饭吃。

现在那么多的程序员中年危机论，要么是无病呻吟，要么是危言耸听。

不过，程序员需要具备危机意识，未雨绸缪。

一年经验可以用十年，但二十年三十年则必然被抛弃。互联网的技术更新很快，程序员必须终身学习，这是选择了这个职业的宿命。



架构对于程序员来说是一座山，入行时只关心自己的几亩地，只有到处多走，才有机会得窥山的全貌。只有站在山顶，程序员的修行才算圆满。很多思想和理念是通用的，掌握之后，学习新领域和新知识可以省很多力气。

在修行的途中，也要注意软技能的锻炼。当前，社会对于程序员群体的偏见逐渐固定：不修边幅、不善言辞、不喜社交，这样的群体在社会上的地位可想而知，甚至体现在公司里就是话语权低，即使成为总监，也是高管会议上的点缀，是咨询专业问题时的顾问。要打破这样的偏见，需要程序员多走出去与不同背景的人交流，抓住表达机会，参与各种活动，培养电脑以外的爱好。

最近，某招聘平台发布报告，显示35岁以上的程序员仍然有很强的需求，所谓的中年危机并不存在。对于程序员，现在无疑是一个美好的时代。

所以，忘记中年危机吧，想想我们应该做些什么，才不辜负这美好的时代？

聚焦最新技术热点 沉淀最优实践经验

[北京站]2018

北京·国际会议中心

演讲：2018年4月20-22日 培训：2018年4月18-19日

精彩案例 先睹为快

《Netflix的工程文化：是什么在激励着我们？》

Speaker: Katharina Probst

Netflix 工程总监

《Apache Kafka的过去，现在，和未来》

Speaker: Jun Rao

Confluent 联合创始人

《人工智能系统中的安全风险》

Speaker: 李康

360网络安全北美研究院负责人，IoT安全研究院院长

《从C#看开放对编程语言发展的影响》

Speaker: Mads Torgersen

微软 C#编程语言Program Manager

《Lavas：PWA的探索与最佳实践》

Speaker: 彭星

百度 资深前端工程师

《浅谈前端交互的基础设施的建设》

Speaker: 程劭非（寒冬）

淘宝 高级技术专家

《深入Apache Spark流计算引擎：Structured Streaming》

Speaker: 朱诗雄

Databricks软件开发工程师，Apache Spark PMC和Committer

《AI大数据时代电商攻防：AI对抗AI》

Speaker: 苏志刚

京东安全 硅谷研究中心负责人

《QUIC在手机微博中的应用实践》

Speaker: 聂永

新浪微博 技术专家

《阿基米德微服务及治理平台》

Speaker: 张晋军

京东 基础架构部服务治理组负责人，架构师

8折 优惠报名中，立减1360元
团 购 享 受 更 多 优 惠

访问官网获取更多前沿技术趋势

2018.qconbeijing.com

如有任何问题，欢迎咨询

电话：15110019061，微信：qcon-0410



Werner Vogels 脑中的未来世界 @AWS re:Invent 2017

作者 杨赛



2017年11月30日，拉斯维加斯。这是Werner Vogels第六次站在AWS re:Invent主题演讲的舞台上。作为AWS的CTO，多年来他发起、参与并见证了无数的变化，但是有一件事情一直没有变过……

他一直在想象一件事：未来的开发会发生怎样的变化？

“虽然我无法预测五年后的世界会变成什么样子，正如同五年之前的我无法预测今天一样。但是有一点我们非常清楚：靠我们自己是无法把AWS建设成今天这样的。不是我们去教育开发者们应该如何开发软件，而是他们教会我们应该去开发怎样的软件。”

“举个例子，当年我们之所以开发DynamoDB，是因为我们听到了开发者对于另一种数据库的需求。然而当我们发布DynamoDB的时候，我们

并不知道开发者们会需要IAM层面的访问控制功能。是开发者们教会我们DynamoDB的服务应该要怎样去做。”

“我们的发展路线图掌握在你们手中。”

以后的App开发会是怎样的？

想象的能力建立在观察的积累之上，观察的积累建立在用心的基础上。技术提供方与技术需求方之间的交流，本质上是人与人之间的交流。当客户有意或无意间说出一句话的时候，身为希望去帮助客户解决问题的这一个人，他听到了什么？他听见了什么？

“一位GE客户曾经说过一句话：我们晚上睡觉的时候还是一家制造公司，结果一觉睡醒，变成一家数据分析公司了。”

Vogels举了这个例子之后，说了两件事：

1. 机器学习领域过去两三年的发展，最大的意义在于两个字：“实时”。实时在线数据和离线历史数据完全是两码事。TensorFlow和MXNet这样的神经网络框架，有它们和没有它们最大的区别就在于是以天为单位出结果，还是以秒为单位出结果。以天为单位出结果，就限制了只能对那些离线的历史数据进行学习。以秒为单位出结果，才使得我们有能力对现在正在发生的事情进行学习。

Real time这个词组在Vogels大叔的这段话里至少出现了三次。我想他应该是希望我们听见什么。

2. 大家为什么要花费这么大精力来到我的Keynote现场呢？为什么你的爷爷奶奶除了Skype之外不会用别的App？为什么菲律宾的贫困农民不会用智能手机App来改进他们的生产？为什么同样的人机交互方式，我们就可以无障碍的使用，他们却不行？

有的观点会说，如果他们真的需要某个App，他们肯定还是能学会的。之所以学不会，是因为他们还并没有那么想要这个App的功能。

Vogels绝对不认同这个观点。

“他们怎么可能会不想要？”

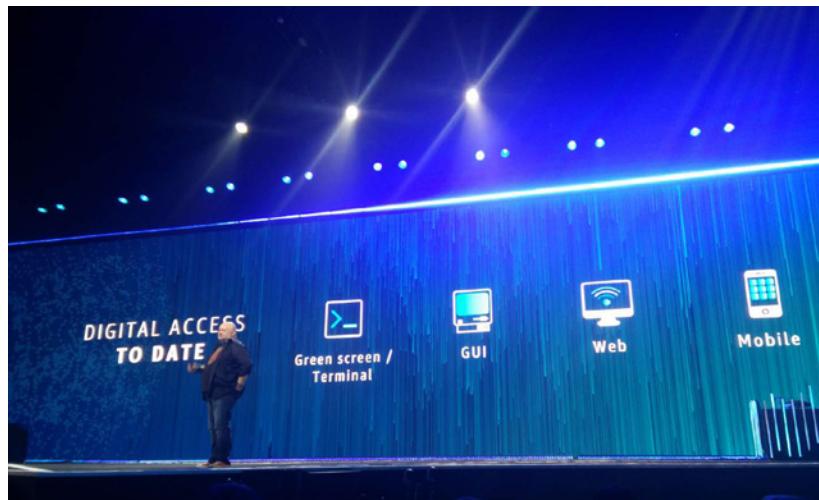


图1 迄今为止的人机交互方式

“我们之所以具备了人机交互的能力，是因为我们去适应了机器的交互方式。我们学会了使用键盘、鼠标和屏幕，是因为我们具备了特定的学习条件，但这些并不是人类的自然交互方式。大家来到我的Keynote现场，难道会是为了来看我背后这块屏幕的吗？肯定不能！你们是来听我说话的。”

“未来的软件一定要去适应人类天然的交互方式。人类天然的交互方式就是说话、倾听。这是你的爷爷奶奶和菲律宾的农民们都会用的交互方式。”



图2 以人类为中心的交互方式

的确，Amazon 是已经发布了 Echo 设备，发布了 Polly、Lex、Transcribe 和 Comprehend，以及基于其所构建的 Alexa for Home、Alexa for Business 服务，但这些只不过是微不足道的小小几步。这个领域需要更多开发者们的关怀，需要开发者们持续不断的进行更多的观察和想象。

以上是 Vogels 对于机器技术栈的最上层——人机交互接口层面的未来思考。

“语音能力一定会是下一代系统的构建核心，这也是深度学习技术将发挥重大意义的第一个阵地。”

其实从技术背景来说，Vogels 的关注点主要是在底层技术栈，应用层的工作算不上 AWS 的主业。如果是强调语音交互能力的战略重要性，完全可以由 AWS CEO Andy Jassy 而不是 Vogels 来讲。如果只是产品发布的话，这次连 GuardDuty 这个量级的安全服务都没上主题演讲（GuardDuty 这次是安排了一次晚场活动做的发布，该服务利用机器学习技术实现了一些威胁自动报警的功能），Transcribe 和 Comprehend 在 Andy 的演讲里都只是在中间小小的露了一个脸，Alexa for Business 身为一个应用层服务，为什么能排在 Vogels 演讲的第一部分、而且还占了这么大篇幅呢？InfoQ 编辑认为，Vogels 来讲语音交互这个事情是想表达他的一个态度：

他觉得开发者们现在对语音交互领域的关注度实在是太不够了。

所以，Vogels 要动用自己技术领导者的影响力建议开发者们赶紧往这个方向多跑跑。

回到底层

讲完语音交互，Vogels 把话题收回到了 IT 架构的底层领域。

底层的几件事主要是在实践层面，即所谓的“最佳实践”推荐。“最佳实践”在 AWS re:Invent 期间有很多专门的课程，此类课程特别受到一线工程师们的欢迎。其实这些最佳实践，Vogels 想必已经说了不知道多少遍了，可是他还是继续说。也许他觉得听进去的人还不够多吧？

技术细节在主题演讲的有限时间也讲不了太深，InfoQ 编辑在这里给

Vogels的分享做个简单的综述。

第一件事：别让你的架构被最初的计划限制死了。你还在预测“我两年后会需要多少容量”这件事吗？那已经是老黄历了，现在再这样做是要坑死自己的！请从一开始就以“可以伸展的方式”设计你的架构（extensive architecture）。

第二件事：安全必须在所有工作之前进行。安全的优先级高于任何一个特性开发！你的每一位开发者都应该是一个安全工程师。与此同时，如果你想确保你的数据只有你自己能访问，那么加密是唯一的保证（Vogels在这里提到了本次re:Invent发布的KMS服务——bring your own keys）。当然，你一定会需要各种自动化工具来记录系统的一切变化、检测系统的任何异常——Macie和GuardDuty就是做这事儿的。

第三件事：你需要更好的开发工具。也许你可以试试我们的Cloud9 IDE？

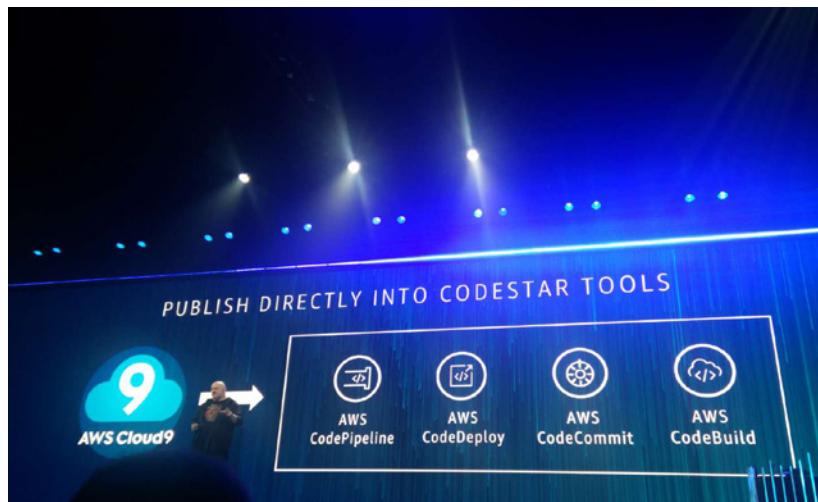


图3 与AWS各服务深度集成的Cloud9 IDE

第四件事：关于可用性的一些常识。如果你在一个可用性区域（AZ）上部署，其基础架构的理论可用性是99%。如果你想要四个九，你可能会需要三个AZ，这意味着三倍的成本。如果你想要五个九，你还需要跨区域部署（region），这意味着六倍的成本。所以在讨论可用性之前，请搞清楚你想要什么。比如，在AWS的工程师们就很清楚的知

道Route 53服务是绝对不能挂的，所以这个服务的可用性做到了绝对的100%，不骗人。而其他的服务，则根据各自的情况来进行各自的决策。

第五件事：关于测试的一些常识。你还在“测试环境”里做测试吗？别逗啦。这年头，生产环境才应该是你的测试场地。强烈推荐大家都学习一些“混乱工程学”（Chaos Engineering），并在自己的生产系统上赶紧用起来。这一段邀请了Netflix的美女工程师、《Chaos Engineering》的作者之一Nora Jones为大家上台做了一次科普。这本书的英文电子版可以在他们的网站[Principles of Chaos](#)上查看。

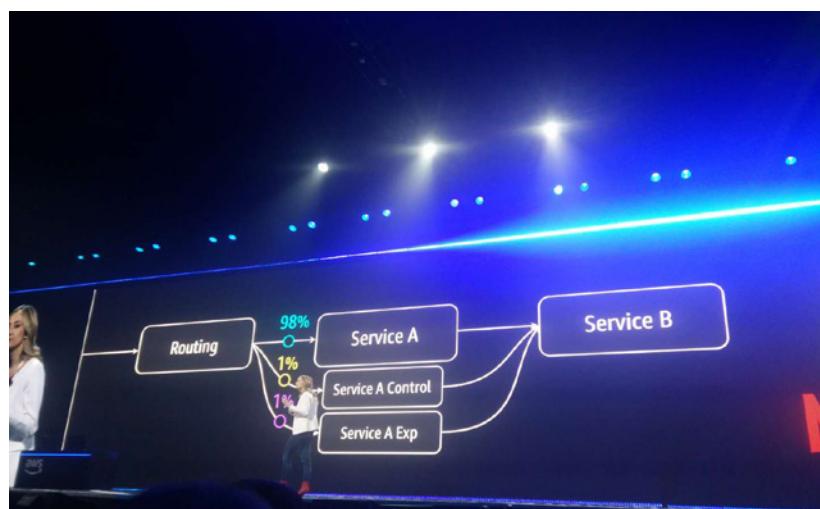


图4 Chaos Engineering是怎样做的

第六件事：Gall's Law——可用的复杂系统在一开始都是可用的简单系统。如果你的起点是一个复杂系统，那么最可能得到的结果是一个不可用的复杂系统。你问我怎么才能让系统不那么复杂？我的第一个建议是微服务——不仅仅是容器，同时还需要正确的微服务设计思路、以及好用的容器管理工具。欢迎早点来试试我们的EKS（Kubernetes）和Fargate！第二个建议是无服务。Vogels在这里介绍了Lambda最近的一些新功能，包括API Gateway VPC集成、并发控制、3GB的内存、以及.NET core 2.0语言的支持，同时还发布了[AWS Serverless Application Repository](#)。

第七件事：SageMaker！

当Andy Jassy正式把[SageMaker](#)的消息对外发布的时候，Vogels在

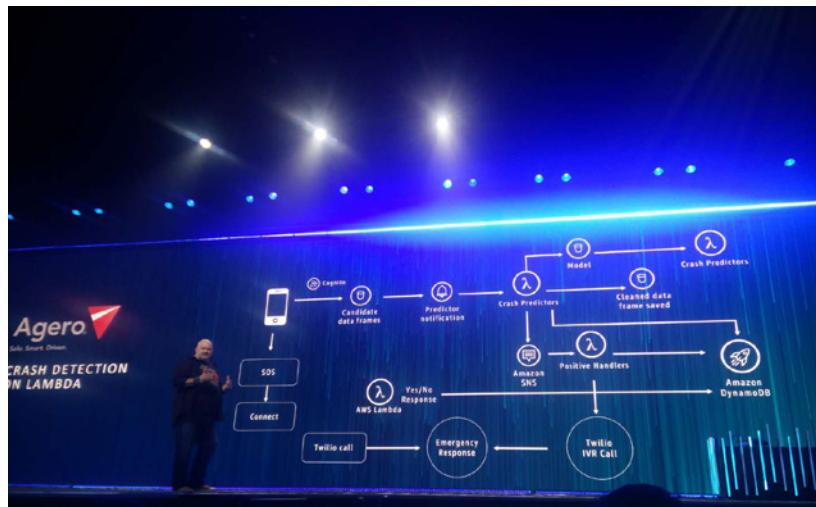


图5 Vogels举例说明无服务（Lambda）的一些实际用法

Twitter上发了一条推：

“SageMaker有多么重大的意义呢？那就是无论我描述它有多大的意义都不算过分。”

按照AWS的惯例，新产品正式对外发布之前都是已经有内部选定的客户秘密使用过一段时间的，对于SageMaker而言，DigitalGlobe就是其中的一个客户。他们是做卫星地图数据的，数据采集了17年下来攒了有100PB。因为这个数据量太大，所以他们首先是成了[AWS Snowmobile](#)的用户——对，你没想错，就是在2016年re:Invent上台的那辆卡车：

100PB的图片在AWS上放着，成本实在不低，所以很自然的当作冷数据放进Glacier。但是这些图片数据还需要拿来做分析，怎么办？所以Glacier Select相当于也是AWS配合他们家的需求做的。分析、预测的需求场景那就比较多了，比如从他们自己的层面，需要提升图片缓存命中率；从他们客户的层面，比如某国运营商要部署5G网络，会问他们要全国树木的分布情况；某国森林大火要做紧急疏散，会问他们要疏散方案等等。现在，他们在SageMaker上每天分析的图片量有80TB。

SageMaker的意义还不止如此。如果你已经试用过SageMaker（如果没有，建议去注册一个AWS账号试试，这个服务现在可以免费试用），

你大概已经知道SageMaker上的训练建模代码是写在[Jupyter Notebook](#)里面的。然后，Jupyter Notebook是可以分享的！DigitalGlobe就把他们的一个GBDX Notebooks分享了出来，这套代码是用来从图片里抽取特征的。我想，他们会很希望看到有人能对这套代码进行一些改善。



图6 在某城市卫星地图上标记所有的树木

总结

看完了Vogels的分享，你的收获如何？关于最佳实践部分的内容，除了本次re:Invent大会上的现场课程之外，有条件的同学也可以了解他们在各地的[AWSome Day](#)、[培训课程](#)，或者是[白皮书](#)。

从 CQS 到 CQRS

作者 hgraca 译者 薛命灯



一个以数据为中心的应用程序，它实现了基本的CRUD操作，并把业务流程留给用户去操心，那么用户就可以在不修改应用程序的情况下改变业务流程。但反过来看，用户必须了解所有业务流程的细节，如果我们有很多繁杂的业务流程，而且需要很多人了解它们，那么就会成为一个大问题。

以数据为中心的应用程序对业务流程一无所知，所以除了修改原始数据，它们什么也做不了。于是它们就变成了数据模型的抽象体，而业务流程仅存在于应用程序用户的脑子里。

一个真正有用的应用程序应该能够为用户分忧解难，通过捕捉用户的意图将“业务流程”的重担从用户肩上移走，它不仅能存储数据，还应该

具备处理业务流程的能力。

有一些技术为应用程序提供了准确的领域映射，它们组合在一起，演化成CQRS，突破了技术上常见的一些局限。

命令查询分离

Bertrand Meyer在1988年出版的《面向对象软件架构》一书中提出了“命令查询分离”的概念，这本书被认为是早期面向对象领域最具影响力的著作之一。

Meyer说，原则上一个方法不应该既修改数据又返回数据，所以我们就有了两类方法：

1. 查询：返回数据，但不修改数据，所以不会产生副作用；
2. 命令：修改数据，但不返回数据。这与单一职责原则（Single Responsibility Principle）一脉相承。

不过，仍然会有一些模式逃逸于这个原则之外。比如，正如Martin Fowler所说的，从一个队列或栈弹出一个元素时，不仅改变了这个队列或栈，还返回了一个元素。

命令模式

命令模式旨在将以数据为中心的应用程序变成以流程为中心，以流程为中心的应用程序具备了领域知识和流程知识。

从应用层面看，用户不需要为了注册一个账号而执行一系列动作，如“创建用户”、“激活用户”和“发送邮件通知”，他们只需要执行一个“注册用户”命令就可以了，其他步骤已经被封装到业务流程里。

举一个更有趣的例子，假设我们要填写一份表单来修改客户数据，我们可以修改客户的名字、地址、电话号码以及他是否是一个有优先权的客户。我们再假设只有已经支付过账单的客户才能成为有优先权的客户。在一个CRUD应用程序里，我们会先接收客户的数据，检查这个客户是否支付过账单，然后决定是接受还是拒绝修改这个客户的数据。我们还有另一

个业务流程：不管客户是否支付过账单，都应该能修改客户的名字、地址和电话号码。如果使用命令模式，我们可以创建两个命令，分别代表这两个不同的业务流程：一个用于修改客户数据，一个用于更新客户的状态，这两个流程都可以通过同一个UI来触发。

在修改数据时，命令提供了恰到好处的粒度和意图。

——《CQRS详解》，Udi Dahan, 2009

命令模式里仍然可以存在像“创建用户”这样的简单命令。CRUD可以与带有意图的操作共存，形成复杂的业务流程，只要不滥用它们就可以了。

从技术角度来看，《Head First Design Patterns》一书中所描述的命令模式把所有相关的动作封装了起来。如果我们有一系列不同的业务流程（也就是命令）需要在同一个位置上执行，那么这样做就很有用，前提是它们必须要有同样的接口。例如，所有的命令都需要有execute()方法，这样它们就可以在适当的时候被执行。业务流程（命令）就可以被加入队列，在适当的时候再执行，既可以同步执行也可以异步执行。

《Head First Design Patterns》在解释命令模式时，以远程控制房子里的电灯为例。我在这里也会使用这个作为例子，虽然我觉得它并不是非常恰当。

假设我们有一个控制面板用于控制房子里的电灯，面板上有一个按钮用来打开厨房的灯，另一个用来关闭它们。每一个按钮都代表了一个命令，用于控制房子里的电灯。

这个系统可以设计成这样，见图1。

当然，这样的设计其实是不成熟的，它甚至都没有考虑使用DIC（密度指示控制器），也没有使用合适的UML。不过，我们假设它可以实现我们的目的：初始化LightController，传入构造参数CommandInvoker，并触发控制器动作KitchenLightOnAction。这个动作将初始化KitchenLight和KitchenLightOnCommand，并将KitchenLight对象作为构造参数传给KitchenLightOnCommand。CommandInvoker将会在某个时间点执行

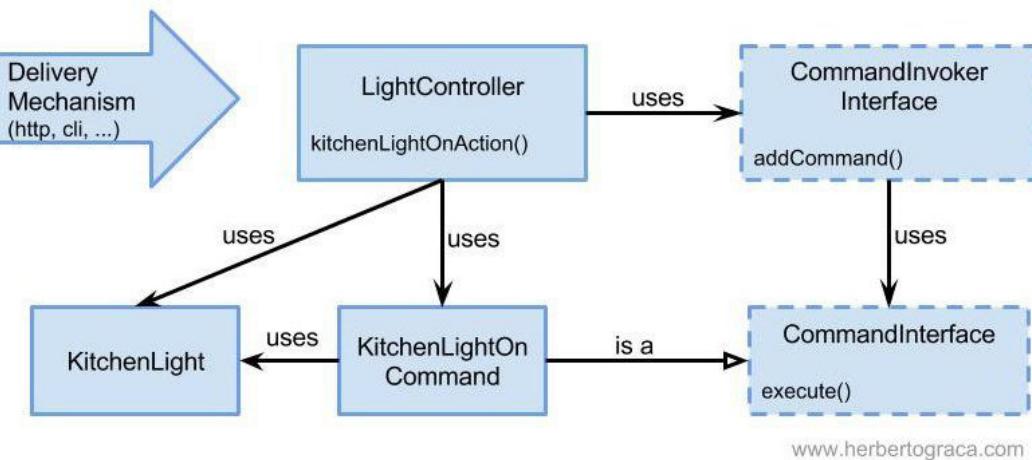


图 1

KitchenLightOnCommand。我们需要创建另一个Action和Command来关闭电灯，不过设计过程基本上是一样的。

这样我们就有了分别用于打开和关闭电灯的两个命令。如果我们要设置50%的亮度该怎么办？我们需要创建另一个命令！而如果要设置25%或75%的亮度呢？我们需要创建更多的命令！如果我们使用调光器代替按钮来设置任意亮度又该怎么办？我们总不能创建不计其数的命令吧！

这种实现方式的问题在于，命令是一样的，但数据（也就是亮度）一直在变。所以，我们应该只使用一个包含相同逻辑的命令，然后使用不同的数据来执行，但问题是命令接口的execute()方法是不接受参数的。如果它接受参数，就会破坏命令模式的设计初衷（封装所有的业务逻辑，不需要知道具体要执行的是什么）。

当然，我们可以将数据作为构造参数传给命令，但这样也不优雅。实际上，这样做带有侵入性，因为数据变成了执行业务逻辑的前提，也就是说，数据成为这个方法的依赖项。

命令总线

为了打破命令模式的局限性，我们能够做的是使用最古老的面向对象原则：让变化部分和不变部分相分离。

在这种情况下，数据就是会发生变化的部分，命令里的逻辑就是不变的部分，所以我们可以把它们分别放到两个类里面。一个是简单的包含了数据的DTO对象（我们把它叫作命令），另一个则包含了要执行的逻辑（我们把它叫作处理器），它有一个用于触发执行逻辑的方法，也就是 execute(CommandInterface \$command):void。我们还把CommandInvoker变成可以接收命令并为命令分配处理器的实体，我们称之为命令总线。

另外，通过修改用户接口，很多命令可以不需要马上执行，它们可以被放到队列里等待异步执行。这样可以让系统更健壮：

1. 返回给用户的响应会更快，因为我们不需要立即执行命令；
2. 如果系统出现bug或数据库离线导致命令执行失败，用户并不会感知到，我们可以在系统修复之后重放命令。

我们在一个中心位置触发逻辑（触发处理器），同时可以在启动处理器之前或执行完处理器之后加入逻辑。例如，我们可以在将数据传给处理器之前对其进行验证，或者将处理器放进一个数据库事务里，我们也可以让命令总线支持复杂的队列操作、异步命令或异步处理器。

命令总线通常使用装饰器（decorator）来实现这些功能，装饰器就像俄罗斯套娃一样层层包装命令总线。

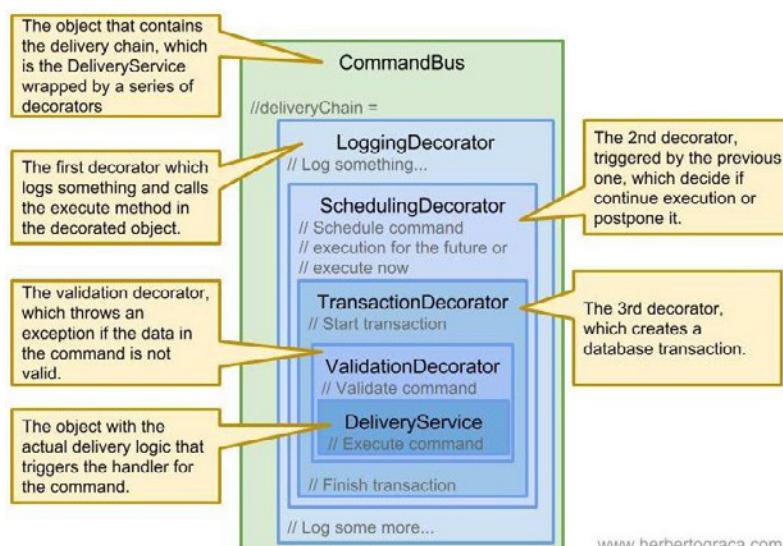


图 2

我们可以创建自己的装饰器，按照任意的顺序来包装命令总线，并添加自定义功能。如果要用到命令队列，就加入一个装饰器来管理命令队列。如果没有用到事务性数据库，就不需要装饰器。

命令查询责任分离

将CQS、命令、查询和命令总线放在一起，我们就得到了CQRS。CQRS有不同的实现方式，可以只有命令端，也可以不使用命令总线。图3展示了一个完整的CQRS实现。

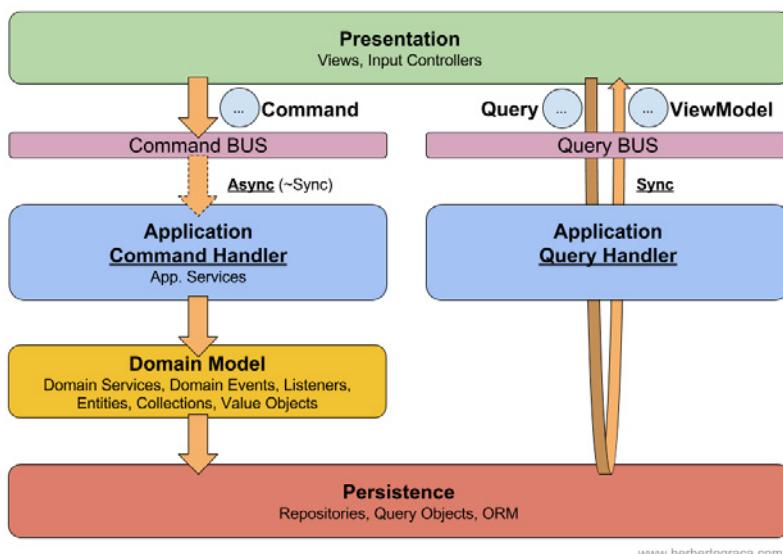


图 3

查询端

在CQS里，查询端只返回数据，完全不修改数据。因为我们不打算在这些数据上执行业务逻辑，所以不需要业务对象（也就是实体），也没必要使用ORM框架。我们只需要查询原始数据，并把它们嵌到视图模板展示给用户！

这在性能方面具有一定的优势：在查询数据时我们不需要经过业务层，我们只获取必要的数据。

这里还存在一种优化的可能性——将数据完全独立地保存到两个数据

存储里：一个为写优化，另一个为读优化。举个例子，如果我们正在使用一个RDBMS：

- 读操作不需要做数据完整性检查，因为在写入数据时已经做过数据完整性检查，所以连外键约束也不需要了。我们可以从读数据库中移除数据完整性约束。
- 我们还可以结合使用数据库视图和视图模板来加快查询速度（尽管在修改模板时需要同步数据库视图，从而给系统增加一定的复杂性）。

如果每个模板都有对应的数据库视图，那我们还需要专门用于读操作的RDBMS吗？或许我们可以改用文档数据库，比如MongoDB或Redis。但谁知道呢，我只是觉得在遇到性能问题时可以多想想其他的解决方案。

查询本身可以通过查询对象实现，查询对象返回一组数据，并应用在模板上。我们也可以使用更复杂的查询总线，它接收模板名称，使用查询对象查询数据，并返回模板所需要的视图模型（ViewModel）。

这种方式可以解决Greg Young提出的一些问题：

- 读取数据操作通常包含分页和排序信息；
- Getter方法会暴露领域对象的内部状态；
- 使用预加载路径会导致ORM加载更多的数据；
- 通过聚合的方式构建DTO会导致不必要的查询；
- 最大的问题是查询优化变得相当困难：因为查询先是作用在对象模型上，然后被转译成数据模型（比如使用了ORM框架），难以对其进行优化。

命令端

通过使用命令，应用程序从以数据为中心的设计变成以行为为中心的设计，这与领域驱动设计一脉相承。

将读操作从处理命令的代码和领域中移出之后，Greg Young之前所说的问题也就不存在了：

- 领域对象不需要暴露内部状态；
- 除了 GetById 之外，只需要少量的查询方法；
- 聚合边界聚焦在行为上。

实体之间的“one-to-many”和“many-to-many”关系对ORM的性能有重大影响。所幸的是，在处理命令时很少需要用到这些关系，它们一般用于查询，而我们已经将查询移出了命令处理流程，所以也就可以移除这些关系。当然，这里所说的关系并不是指数据库的表间关系，表间的外键约束仍然会存在，这里所说的关系指的是ORM层面的实体之间的关系。

业务流程事件

在成功处理完一个命令后，处理器会触发一个事件，用于通知应用程序的其他部分。事件的名称应该要与其对应的命令一样，而作为一个事件，应该使用过去式。

总结

CQRS让读模型和写模型完全分离，因此可以对读操作和写操作进行优化。这样会带来性能上的提升，同时也会让代码更清晰、更简单，代码反映了领域模型，提升了代码的可维护性。

这一切都是关于封装、低耦合、高聚合和单一职责原则。

不过，尽管CQRS可以让应用程序变得更健壮，但并不是说所有的应用程序都要使用CQRS：我们应该在必要的时候选择正确的方案。

参考资料

- 1994 - Gamma, Helm, Johnson, Vlissides - [Design Patterns: Elements of Reusable Object-Oriented Software](#)
- 1999 - Bala Paranj - [Java Tip 68: Learn how to implement the Command pattern in Java](#)
- 2004 - Eric Freeman, Elisabeth Robson - [Head First Design Patterns](#)
- 2005 - Martin Fowler - [Command Query Separation](#)
- 2009 - Udi Dahan - [Clarified CQRS](#)

- 2010 - Greg Young - [CQRS, Task Based UIs, Event Sourcing agh!](#)
- 2010 - Greg Young - [CQRS Documents](#)
- 2010 - Udi Dahan - [Race Conditions Don't Exist](#)
- 2011 - Martin Fowler - [CQRS](#)
- 2011 - Udi Dahan - [When to avoid CQRS](#)
- 2014 - Greg Young - [CQRS and Event Sourcing - Code on the Beach 2014](#)
- 2015 - Matthias Noback - [Responsibilities of the command bus](#)
- 2017 - Martin Fowler - [What do you mean by "Event-Driven"?](#)
- 2017* - Doug Gale - [Command Pattern](#)
- 2017* - Wikipedia - [Command Pattern](#)

阿里巴巴正式开源其自研容器技术 Pouch

作者 孙宏亮



11 月 19 日上午，在中国开源年会现场，阿里巴巴正式开源了基于 Apache 2.0 协议的容器技术 Pouch。Pouch 是一款轻量级的容器技术，拥有快速高效、可移植性高、资源占用少等特性，主要帮助阿里更快的做到内部业务的交付，同时提高超大规模下数据中心的物理资源利用率。开源之后，Pouch 成为一项普惠技术，人人都可以在 GitHub 上获取，GitHub 项目地址。

Pouch 的开源，是阿里看好容器技术的一个信号。时至今日，全球范围内，容器技术在大多数企业中落地，已然成为一种共识。如何做好容器的技术选型，如何让容器技术可控，相信是每一个企业必须考虑的问题。Pouch 无疑使得容器生态再添利器，在全球巨头垄断的容器开源生态中，为中国技术赢得了一块阵地。

Pouch 技术现状

此次开源 Pouch，相信行业中很多专家都会对阿里目前的容器技术感兴趣。到底阿里玩容器是一个侠之大者，还是后起之秀呢？以过去看未来，技术领域尤其如此，技术的沉淀与积累，大致可以看清一家公司的技术实力。

Pouch 演进

追溯 Pouch 的历史，我们会发现 Pouch 起源于 2011 年。当时，Linux 内核之上的 namespace、cgroup 等技术开始成熟，LXC 等工具也在同时期诞生不久。阿里巴巴作为一家技术公司，即基于 LXC 研发了容器技术 t4，并在当时以产品形态给集团内部提供服务。此举被视为阿里对容器技术的第一次探索，也为阿里的容器技术积淀了最初的经验。随着时间的推移，两年后，Docker 横空出世，其镜像技术层面，极大程度上解决了困扰行业多年的“软件封装”问题。镜像技术流行开来后，阿里没有理由不去融合这个给行业带来巨大价值的技术。于是，在 2015 年，t4 在自身容器技术的基础上，逐渐吸收社区中的 Docker 镜像技术，慢慢演变，打磨为 Pouch。

带有镜像创新的容器技术，似一阵飓风，所到之处，国内外无不叫好，阿里巴巴不外如是。2015 年末始，阿里巴巴集团内部在基础设施层面也在悄然发生变化。原因很多，其中最简单的一条，相信大家也不难理解，阿里巴巴体量的互联网公司，背后必定有巨大的数据中心在支撑，业务的爆炸式增长，必定导致基础设施需求的增长，也就造成基础设施成本的大幅提高。容器的轻量级与资源消耗低，加上镜像的快速分发，迅速让阿里巴巴下定决心，在容器技术领域加大投入，帮助数据中心全面升级。

阿里容器规模

经过两年多的投入，阿里容器技术 Pouch 已经在集团基础技术中，扮演着极其重要的角色。2017 年双 11，巨额交易 1682 亿背后，Pouch 在

“超级工程”中做到了：

- 100% 的在线业务 Pouch 化
- 容器规模达到百万级

回到阿里集团内部，Pouch 的日常服务已经覆盖绝大部分的事业部，覆盖的业务场景包括：电商、广告、搜索等；覆盖技术栈包括：电商应用、数据库、大数据、流计算等；覆盖编程语言：Java、C++、NodeJS 等。

Pouch 技术优势

阿里巴巴容器技术如此之广的应用范围，对行业来说实属一大幸事，因为阿里已经用事实说明：容器技术已经在大规模生产环境下得到验证。然而，由于 Pouch 源自阿里，而非社区，因此在容器效果、技术实现等方面，两套体系存在差异。换言之，Pouch 存在不少独有的技术优势。

隔离性强

隔离性是企业走云化之路过程中，无法回避的一个技术难题。隔离性强，意味着技术具备了商用的初步条件；反之则几乎没有可能在业务线上铺开。哪怕是阿里巴巴这样的技术公司，实践容器技术伊始，安全问题都无法幸免。众所周知，行业中的容器方案大多基于 Linux 内核提供的 cgroup 和 namespace 来实现隔离，然后这样的轻量级方案存在弊端：

- 容器间，容器与宿主间，共享同一个内核；
- 内核实现的隔离资源，维度不足。

面对如此的内核现状，阿里巴巴采取了三个方面的工作，来解决容器的安全问题：

- 用户态增强容器的隔离维度，比如网络带宽、磁盘使用量等；
- 给内核提交 patch，修复容器的资源可见性问题，cgroup 方面的 bug；
- 实现基于 Hypervisor 的容器，通过创建新内核来实现容器隔离。

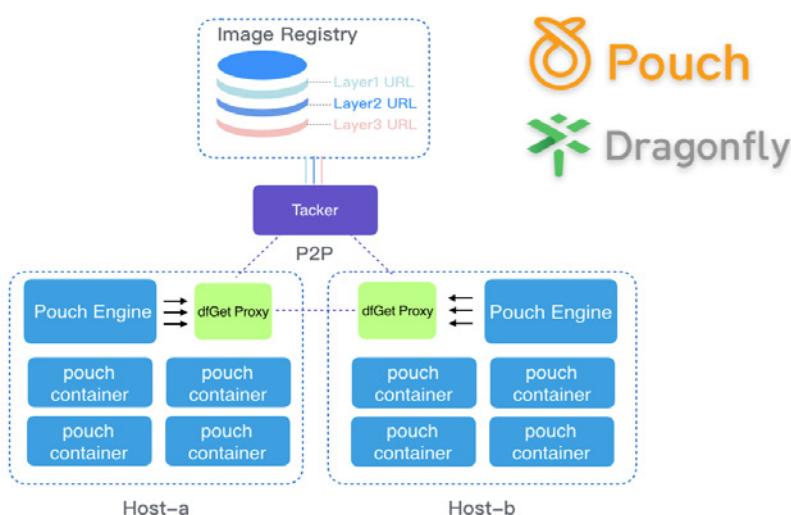
容器安全的研究，在行业中将会持续相当长时间。而阿里在开源 Pouch 中，将在原有的安全基础上，继续融合 lxcfs 等特性与社区共享。同时阿里巴巴也在计划开源“阿里内核”，将多年来阿里对 Linux 内核的增强回馈行业。

P2P 镜像分发

随着阿里业务爆炸式增长，以及 2015 年之后容器技术的迅速普及，阿里容器镜像的分发也同时成为亟待解决的问题。虽然，容器镜像已经帮助企业在应用文件复用等方面，相较传统方法做了很多优化，但是在数以万计的集群规模下，分发效率依然令人抓狂。举一个简单例子：如果数据中心中有 10000 台物理节点，每个节点同时向镜像仓库发起镜像下载，镜像仓库所在机器的网络压力，CPU 压力可想而知。

基于以上场景，阿里巴巴镜像分发工具“蜻蜓”应运而生。蜻蜓是基于智能 P2P 技术的通用文件分发系统。解决了大规模文件分发场景下分发耗时、成功率低、带宽浪费等难题。大幅提升发布部署、数据预热、大规模容器镜像分发等业务能力。目前，“蜻蜓”和 Pouch 同时开源，项目地址。

Pouch 与蜻蜓的使用架构图如下：



富容器技术

阿里巴巴集团内部囊括了各式各样的业务场景，几乎每种场景都对 Pouch 有着自己的要求。如果使用外界“单容器单进程”的方案，在业务部门推行容器化存在令人难以置信的阻力。阿里巴巴内部，基础技术起着巨大的支撑作用，需要每时每刻都更好的支撑业务的运行。当业务运行时，技术几乎很难做到让业务去做改变，反过来适配自己。因此，一种对应用开发、应用运维都没有侵入性的容器技术，才有可能大规模的迅速铺开。否则的话，容器化过程中，一方面得不到业务方的支持，另一方面也需要投入大量人力帮助业务方，非标准化的实现业务运维。

阿里深谙此道，内部的 Pouch 技术可以说对业务没有任何的侵入性，也正是因为这一点在集团内部做到 100% 容器化。这样的容器技术，被无数阿里人称为“富容器”。

“富容器”技术的实现，主要是为了在 Linux 内核上创建一个与虚拟机体验完全一致的容器。如此一来，比一般容器要功能强大，内部有完整的 init 进程，以及业务应用需要的任何服务，当然这也印证了 Pouch 为什么可以做到对应用没有“侵入性”。技术的实现过程中，Pouch 需要将容器的执行入口定义为 systemd，而在内核态，Pouch 引入了 cgroup namespace 这一最新的内核 patch，满足 systemd 在富容器模式的隔离性。从企业运维流程来看，富容器同样优势明显。它可以在应用的 Entrypoint 启动之前做一些事情，比如统一要做一些安全相关的事情，运维相关的 agent 拉起。这些需要统一做的事情，倘若放到用户的启动脚本，或镜像中就对用户的应用诞生了侵入性，而富容器可以透明的处理掉这些事情。

内核兼容性

容器技术的井喷式发展，使得不少走在技术前沿的企业享受到技术红利。然后，“长尾效应”也注定技术演进存在漫长周期。Pouch 的发展也在规模化进程中遇到相同问题。

但凡规模达到一定量，“摩尔定律”决定了数据中心会存有遗留资

源，如何利用与处理这些物理资源，是一个大问题。阿里集团内部也是如此，不管是不同型号的机器，还是从 2.6.32 到 3.10+ 的 Linux 内核，异构现象依然存在。倘若要使所有应用运行 Pouch 之中，Pouch 就必须支持所有内核版本，而现有的容器技术支持的 Linux 内核都在 3.10 以上。不过技术层面万幸的是，对 2.6.32 等老版本内核而言，namespace 的支持仅仅缺失 user namespace；其他 namespace 以及常用的 cgroup 子系统均存在；但是 /proc/self/ns 等用来记录 namespace 的辅助文件当时还不存在，setns 等系统调用也需要在高版本内核中才能支持。而阿里的技术策略是，通过一些其他的方法，来绕过某些系统调用，实现老版本内核的容器支持。

当然，从另一个角度而言，富容器技术也很大程度上，对老版本内核上的其他运维系统、监控系统、用户使用习惯等实现了适配，保障 Pouch 在内核兼容性方面的高可用性。

因此综合来看，在 Pouch 的技术优势之上，我们不难发现适用 Pouch 的应用场景：传统 IT 架构的迅速容器化，企业大规模业务的部署，安全隔离要求高稳定性要求高的金融场景等。

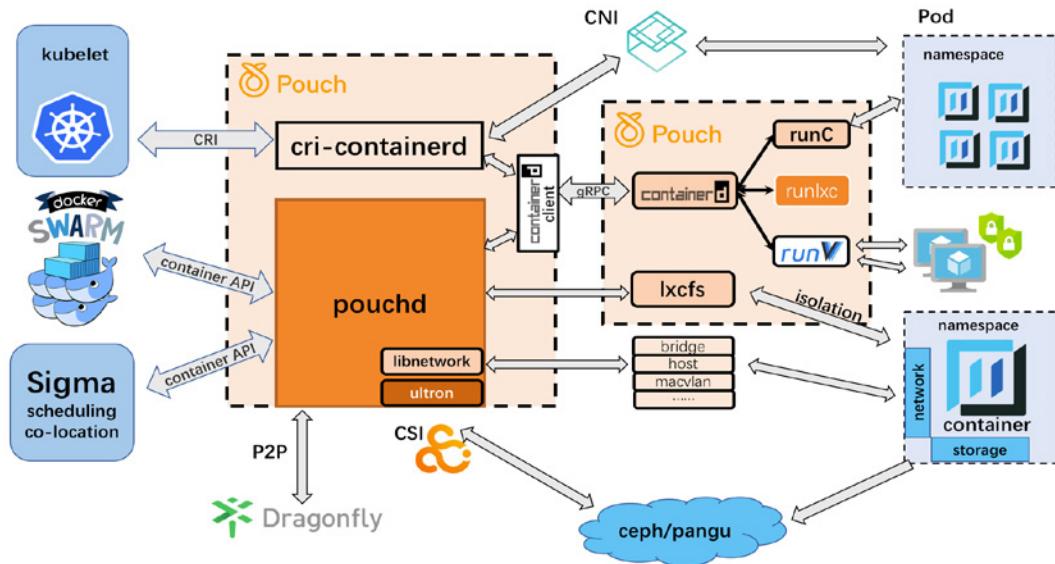
Pouch 架构

凭借差异化的技术优势，Pouch 在阿里巴巴大规模的应用场景下已经得到很好的验证。然而，不得不说的是：目前阿里巴巴内部的 Pouch 与当前开源版本依然存在一定的差异。

虽然优势明显，但是如果把内部的 Pouch 直接开源，这几乎是一件不可能的事。多年的发展，内部 Pouch 在服务业务的同时，存在与阿里内部基础设施、业务场景耦合的情况。耦合的内容，对于行业来说通用性并不强，同时涉及一些其他问题。因此，Pouch 开源过程中，第一要务即解耦内部依赖，把最核心的、对社区同样有巨大价值的部分开源出来。同时，阿里希望在开源的最开始，即与社区站在一起，共建 Pouch 的开源社区。随后，以开源版本的 Pouch 逐渐替换阿里巴巴集团内部的 Pouch，最终达成 Pouch 内外一致的目标。当然，在这过程中，内部 Pouch 的解耦工作，

以及插件化演进工作同样重要。而在 Pouch 的开源计划中，明年 3 月底会是一个重要的时间点，彼时 Pouch 的 1.0 版本将发布。

从计划开源的第一刻开始，Pouch 在生态中的架构图就设计如下：

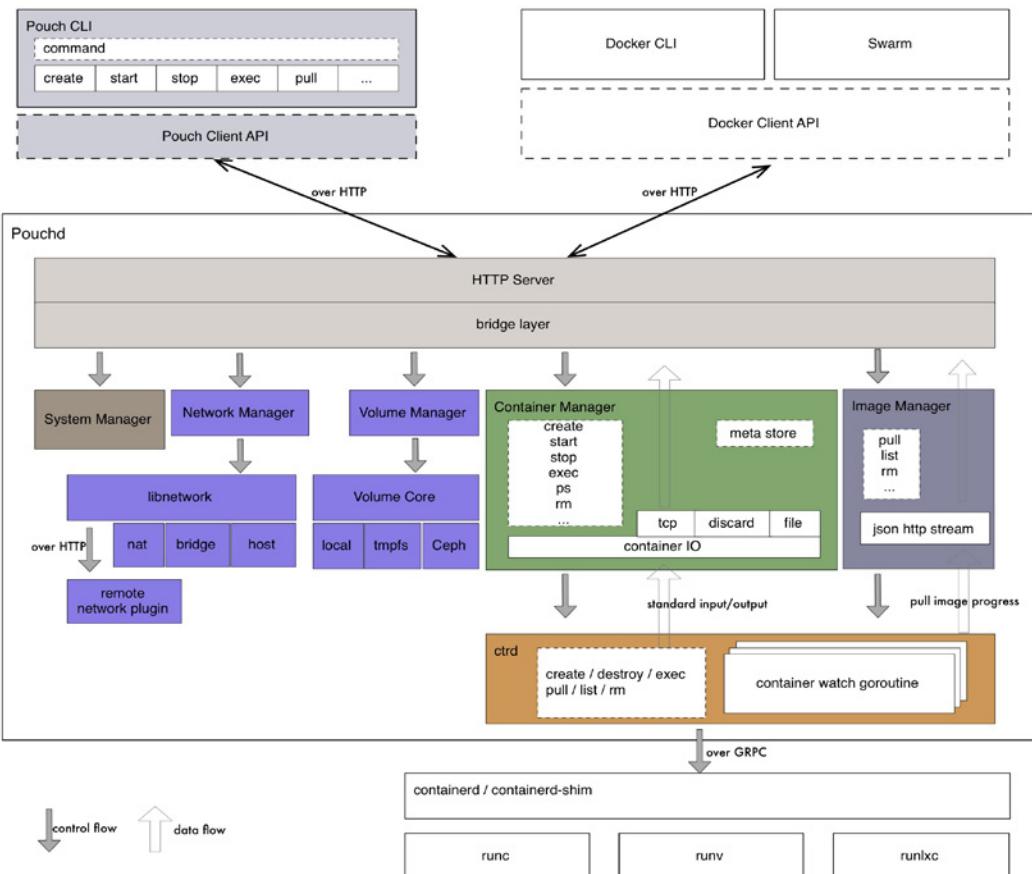


Pouch 的生态架构可以从两个方面来看：第一，如何对接容器编排系统；第二，如何加强容器运行时。

容器编排系统的支持，是 Pouch 开源计划的重要板块。因此，设计之初，Pouch 就希望自身可以原生支持 Kubernetes 等编排系统。为实现这点，Pouch 在行业中率先支持 container 1.0.0。目前行业中的容器方案 containerd 主要停留在 0.2.3 版本，新版本的安全等功能还无法使用，而 Pouch 已经是第一个吃螃蟹的人。当前 Docker 依然是 Kubernetes 体系中较火的容器引擎方案，而 Kubernetes 在 runtime 层面的战略计划为使用 cri-containerd 降低自身与商业产品的耦合度，而走兼容社区方案的道路，比如 cri-containerd 以及 containerd 社区版。另外，需要额外提及的是，内部的 Pouch 是阿里巴巴调度系统 Sigma 的重要组成部分，同时支撑着“混部”工程的实现。Pouch 开源路线中，同样会以支持“混部”为目标。未来，Sigma 的调度（scheduling）以及混部（co-location）能力有望服务行业。

生态方面，Pouch 立足开放；容器运行时方面，Pouch 主张“丰富”与“安全”。runC 的支持，可谓顺其自然。runV 的支持，则表现出了和生态的差异性。虽然 docker 默认支持 runV，然而在 docker 的 API 中并非做到对“容器”与“虚拟机”的兼容，从而 Docker 并非是一个统一的管理入口。而据我们所知，现有企业中仍有众多存量虚拟机的场景，因此，在迎接容器时代时，如何通过统一的运维入口，同时管理容器和虚拟机，势必会是“虚拟机迈向容器”这个变迁过渡期中，企业最为关心的方案之一。Pouch 的开源形态，很好的覆盖了这一场景。runlxc 是阿里巴巴自研的 lxc 容器运行时，Pouch 对其的支持同时也意味着 runlxc 会在不久后开源，覆盖企业内部拥有大量低版本 Linux 内核的场景。

Pouch 对接生态的架构如下，而 Pouch 内部自身的架构可参考下图：



和传统的容器引擎方案相似，Pouch 也呈现出 C/S 的软件架构。命

令行 CLI 层面，可以同时支持 Pouch CLI 以及 Docker CLI。对接容器 runtime，Pouch 内部通过 container client 通过 gRPC 调用 containerd。Pouch Daemon 的内部采取组件化的设计理念，抽离出相应的 System Manager、Container Manager、Image Manager、Network Manager、Volume Manager 提供统一化的对象管理方案。

写在最后

如今 Pouch 的开源，意味着阿里积累的容器技术将走出阿里，面向行业。而 Pouch 的技术优势，决定了其自身会以一个差异化的容器解决方案，供用户选择。企业在走云化之路，拥抱云原生（Cloud Native）时，Pouch 致力于成为一款强有力的软件，帮助企业的数字化转型做到最稳定的支持。

Pouch 目前已经在 GitHub 上开源，欢迎任何形式的开源参与。GitHub 地址。

作者介绍

孙宏亮，阿里巴巴技术专家，毕业于浙江大学，目前在阿里巴巴负责容器项目 Pouch 的开源建设。数年来一直从事云计算领域，是国内第一批研究和实践容器技术的工程师，在国内起到极为重要的容器技术布道作用。拥有著作《Docker 源码分析》，个人崇尚开源精神，同时是 Docker Swarm 项目的全球 Maintainer。

为什么说 Service Mesh 是微服务发展到今天的必然产物？

作者 Sean



过去三年里，架构领域最火的方向非微服务莫属。

微服务的好处显而易见，它本身所具备的可扩展性、可升级性、易维护性、故障和资源的隔离性等诸多特性使得产品的生产研发效率大大提高。同时，基于微服务架构设计风格，研发人员可以构建出原生对于“云”具备超高友好度的系统，让产品的持续集成与发布变得更为便捷。

但是，世界上没有完美无缺的事物，微服务也是一样。著名软件大师，被认为是十大软件架构师之一的 Chris Richardson 曾一针见血地指出：“微服务应用是分布式系统，由此会带来固有的复杂性。开发者需要在 RPC 或者消息传递之间选择并完成进程间通讯机制。此外，他们必须写代码来处理消息传递中速度过慢或者不可用等局部失效问题。”

在云原生模型里，一个应用可以由数百个服务组成，每个服务可能有数千个实例，而每个实例可能会持续地发生变化。这种情况下，服务间通信不仅异常复杂，而且也是运行时行为的基础。管理好服务间通信对于保证端到端的性能和可靠性来说是无疑是非常重要的。

以上种种复杂局面便催生了服务间通信层的出现，这个层既不会与应用程序的代码耦合，又能捕捉到底层环境高度动态的特点，让业务开发者只关注自己的业务代码，并将应用云化后带来的诸多问题以不侵入业务代码的方式提供给开发者。

这个服务间通信层就是 Service Mesh，它可以提供安全、快速、可靠的服务间通讯（service-to-service）。

在过去的一年中，Service Mesh 已经成为云原生技术栈里的一个关键组件。很多拥有高负载业务流量的公司都在他们的生产应用里加入了 Service Mesh，如 PayPal、Lyft、Ticketmaster 和 Credit Karma 等。

那么，对于 Service Mesh 这种新兴事物，它有哪些显而易见的优点？现在是否是企业落地 Service Mesh 的好时机？市面上有哪些好用的开源解决方案？InfoQ 记者就这些问题采访了普元云计算架构师、微服务专家宋潇男。

InfoQ：Service Mesh 现在国内基本都翻译为“服务网格”了，它的出现也没多长时间。根据我们的了解，它今年年中开始迅速在社区中流行并获得关注。能否谈谈你对 **Service Mesh** 的理解？它的出现最终是为了解决什么问题？

宋潇男：Service Mesh 这个词本身出现的时间确实不长，但是它所描绘的事情存在的时间可不短，其本质就是分布式系统的动态链接过程，眼下最大众化的分布式系统就是微服务，所以可以简单地说，Service Mesh 就是微服务的动态链接器（Dynamic Linker）。

让我们回忆一下单机程序的运作方式，源代码被编译器编译为一系列目标文件，然后交由链接器将这些目标文件组装成一个可执行文件，链接过程就是将各个目标文件之间对符号（方法、变量、函数、接口等）的

引用转化为对内存地址的引用，由于这个过程在生成可执行文件时就完成了，所以被称为静态链接。

后来为了程序的模块化和功能上的解耦与共用，开始把一些常见的公共程序剥离出来，制作成库文件供其他程序使用，在引用这些库文件的程序运行时，操作系统上的动态链接器会在库文件中查询到被引用的符号，然后将这些符号的内存地址映射到该程序的虚拟内存空间之中，由于这个过程是在程序运行时完成的，所以被称为动态链接。

再后来出现了分布式系统，程序被散布在网络中的不同主机上，那么如何链接这些程序呢？业界走过了和链接单机程序类似，但是却艰难得多的一段历程。因为这个访谈是在微服务的大背景下进行的，为了叙述方便，我们从现在开始把这些程序称为服务。业界最开始是把这些服务的网络地址写在配置文件中，这个方案显然问题太多、很不靠谱。所以接下来自然而然地出现了服务注册中心来统一记录这些服务的网络地址并维护这些地址的变化，服务通过注册中心提供的客户端 SDK 与注册中心通信并获得它们所依赖的服务的网络地址。由于网络通信远没有内存通信稳定，为了保证可靠的服务调用，又出现了用于流量控制的 SDK，提供流量监控、限流、熔断等能力。

在大型系统中，被依赖的服务往往以多实例的方式运行在多个主机上，有多个网络地址，所以又出现了用于负载均衡的SDK。现在问题貌似是解决了，但是我们手里多了一堆 SDK，我们手上已有的应用，必须用这些 SDK 重新开发，这显然可行度不高，而对于新开发的应用，我们又发现这些 SDK 体积过大，以 Netflix OSS 提供的 SDK 为例，依赖包动辄上百兆，在做微服务开发时，经常发现 SDK 的体积比程序本身还大很多倍，如果你使用容器技术，你会发现你的程序和基础容器的体积加起来还没 SDK 大，所以经常有人吐槽说现在的这些所谓的微服务框架实际上不是为微服务设计的。另外，SDK 还会带来性能伸缩性的问题，在性能要求较高的系统中，SDK 往往成为了性能瓶颈。再回头看一下单机上动态链接过程的顺畅，这种基于 SDK 的微服务动态链接方案简直是难用的不得了。

这时业界才开始关注已经存在了一段时间的 Service Mesh 方案，Service Mesh 的基础是一个网络代理，这个网络代理会接管微服务的网络流量，通过一个中央控制面板进行管理，将这些流量转发到该去的地方，并在这个代理的基础之上，扩展出一系列的流量监控、限流、熔断甚至是灰度发布、分布式跟踪等能力，而不需要应用本身做出任何修改，让开发者摆脱了 SDK 之苦，也避免了由于 SDK 使用不当造成的一系列问题，同时这个代理工作在网络层，一般情况下也不会成为性能瓶颈。怎么样，是不是有一些单机上动态链接过程的感觉了？

InfoQ：那在 Service Mesh 没有出现之前，微服务架构之间的通信是用什么方式解决的？都有哪些解决方案？

宋潇男：之前的方案也不少，比如刚刚提到的 Netflix OSS 的 SDK 方案，一些大型互联网公司在解决自身内部遇到的问题后，也开源出了一些方案，还有一些规模不太大、不太复杂的系统通过 Nginx 反向代理做了一些服务发现方案，值得注意的是，现在 Nginx 官方也推出了基于 Nginx 的 Service Mesh 方案。

InfoQ：采用 Service Mesh 是否需要对正在使用的一些微服务框架作出改动？企业落地 Service Mesh 有哪些难点？现在是开始转向 Service Mesh 的好时机吗？

宋潇男：目前成熟的微服务框架大多是 SDK 方案，如果你的应用已经用了这些 SDK 进行开发，那么引入 Service Mesh 肯定要做出很多改动的，简单地说，就是改回去。

目前企业要落地 Service Mesh，我觉得并不是一个好时机，先不说落地的难点，首先要面对的问题是 Service Mesh 框架的选型难题，目前最多生产部署的 Service Mesh 方案是 Linkerd，但是由 Google 和 IBM 牵头、众多新老 IT 厂商支持的 Istio 方案似乎又更有前景，可惜 Istio 现在刚刚 0.3 版本，还不能生产使用。所以与其在两种方案间摇摆不定，不如再等等看。而近期 Linkerd 又发布了一个新的 Service Mesh 方案 Conduit，使得局势进一步扑朔迷离。所以我的建议是，再等等看。

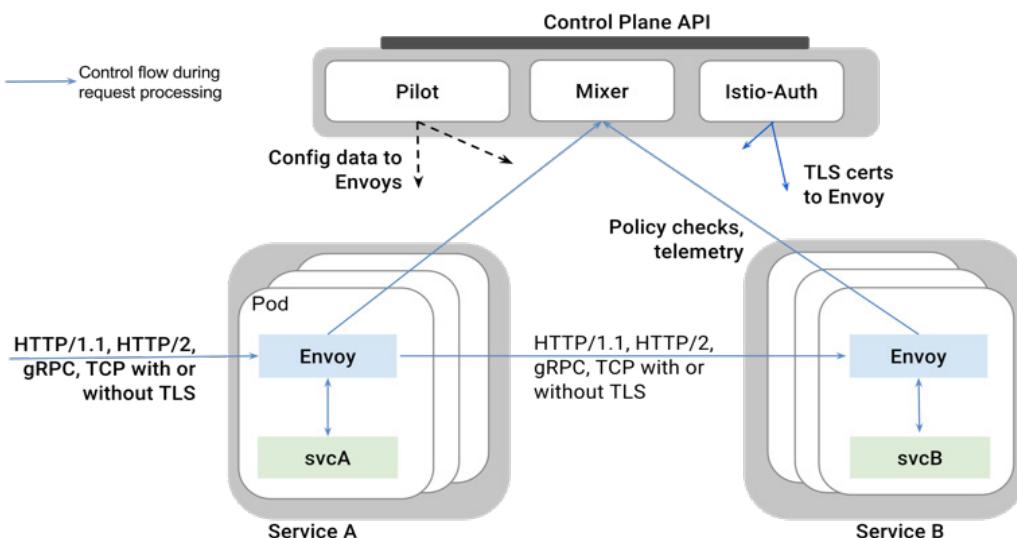
InfoQ: 目前社区都有哪些 Service Mesh 的开源解决方案呢？（可否介绍下？Linkerd、Envoy、Istio）

宋潇男：主要是由 Buoyant 公司推出的 Linkerd 和 Google、IBM 等厂商牵头的 Istio。Linkerd 更加成熟稳定些，Istio 功能更加丰富、设计上更为强大，社区相对也更加强大一些。所以普遍认为 Istio 的前景会更好，但是毕竟还处于项目的早期，问题还很多。

InfoQ：Istio 是目前最为流行的开源解决方案，也有大厂加持，可否解释下 Istio 的架构设计？以及它的社区发展情况？

宋潇男：Istio 的架构并不复杂，其核心组件只有四个。首先是名为 Envoy 的网络代理，用来协调服务网格中所有服务的出入站流量，并提供服务发现、负载均衡、限流熔断等能力，还可以收集大量与流量相关的性能指标；其次是名为 Mixer 收集器，用来从 Envoy 代理收集流量特征和性能指标；然后是名为 Pilot 的控制器，用来将流量协调的策略和规则发送到 Envoy 代理；最后是名为 Istio-Auth 的身份认证组件，用来做服务间访问的安全控制。

详见下图：



至于社区的发展，其实并不是特别火热，毕竟项目正式启动才半年

多，而且与大多数开发者的距离比较远，所以关注者并不是很多。我想让大多数开发者不去关心它，也正是 Service Mesh 的意义所在吧。想想单机时代，有多少人会去关注动态链接器呢？我们总是说应该让开发人员更多的关心业务，可是看看现在的微服务、DevOps 方案，把多少程序运行期的事情推给开发人员解决，这其实是不对的，而 Service Mesh 正是要逆转这个不好的趋势。

InfoQ：现在 Service Mesh 开始逐步成为一个大家“炒作”的新技术，那在你看来，Service Mesh 的引入又会带来哪些新的问题呢？

宋潇男：我以前总是开玩笑说，新技术总是解决一个问题，再带来一堆问题，就好像汽车解决了出行问题，然后带来了修车、修路、建加油站、卖车险等一堆问题，世界就是在这个过程中进步的。可是到了 Service Mesh 这儿，确实想不出它会引入些什么问题，当然这是假设在 Service Mesh 成熟稳定之后。

在我看来，在三到五年之后，Kubernetes 会成为服务器端的标准环境，就像现在的 Linux，而 Service Mesh 就是运行在 Kubernetes 上的分布式应用的动态链接器，届时开发一个分布式应用将会像开发单机程序一样简单，业界在分布式操作系统上长达三十多年的努力将以这种方式告一段落。

受访嘉宾简介

宋潇男，普元信息云计算架构师。目前在普元负责容器相关的技术架构工作，曾在华为云计算部门负责分布式存储和超融合基础设施的产品与解决方案规划管理工作。曾负责国家电网第一代云资源管理平台和中国银联 OpenStack 金融云的技术方案、架构设计和技术原型工作。在云计算的萌芽期曾参与 Globus、HPC 等分布式系统的研究工作，拥有十余年的 UNIX 和分布式系统技术经验。

中小型研发团队架构实践：电商如何做企业总体架构？

作者 张辉清



企业总体架构是什么？有什么用？具体怎么做？以我曾任职的公司为案例，一起来探讨这个问题。

这家公司当时有 200 位研发人员和 200 多台服务器，我刚进这家公司时，系统已经玩不下去了，总是出现各种问题，例如日常发布系统时或访问量稍微过大时，系统就会出现很多故障，而且找不到故障发生的根本原因。

我进公司后主要的任务就是对这个系统进行升级改造，花了一个半月的时间写了份企业总体架构文档，文档共有 124 页，直接指导了之后的技术改造，下图是那份文档的目录，文末有相关资料下载地址。

企业商务模型

- ▷ 1 系统概述
- ▷ 2 企业商务模型
- ▷ 3 信息系统模型
- ▷ 4 应用架构
- ▷ 5 数据设计
- ▷ 6 物理架构
- ▷ 7 接口架构
- ▷ 8 领域模型
- ◀ 9 架构规划
 - 9.1 顶层架构
 - ▷ 9.2 网站功能规划
 - ▷ 9.3 应用规划
 - 9.4 数据库规划
 - ▷ 9.5 物理规划
 - ▷ 9.6 其它
- ▷ 10 架构实施

企业商务模型的内容主要包括主营业务、商务模式、商务主体、竞品分析、组织架构、商务运作模型和业务流程等。

主营业务即公司做什么业务。

商业模式即公司怎么赚钱。

商务主体即哪几个人在一起做这门生意。

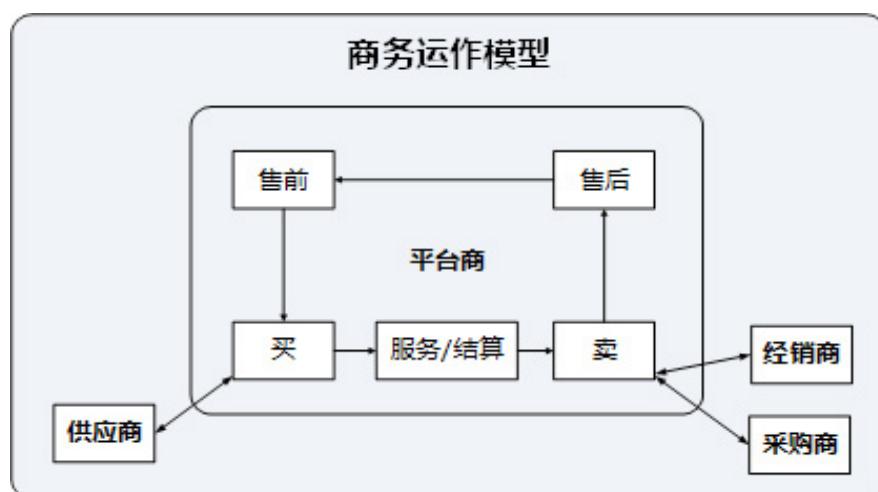
竞品分析即了解竞争对手的情况。

组织架构即公司部门是怎么划分的，组织架构图中标出人数，根据系统与业务之间对应关系，可以了解系统中哪些模块使用频率高，以及

业务与其对应模块的复杂度。

商务运作模型即公司是如何运作的，售前做计划，找供应商买东西进来后，经过服务和结算，再卖给我们的经销商和采购商，使我们获得利润，售后进行大数据分析最后又指导着我们的售前，整个过程形成良性循环。

可以把一家公司想象成一台机器，输进去的是钱，转一转后，又能够生出更多的钱出来。



最后是业务流程和附档资料，业务流程包括预订流程、订单处理流程、产品供应流程、财务结算流程、账户管理流程。企业商务模型的建立，指导着整个应用系统模型的建立，它是整个应用系统建设的基础和前提，毕竟应用系统是为业务服务的。

架构现状

架构现状的内容主要包括：功能架构、应用架构、数据设计和物理架构。

功能架构

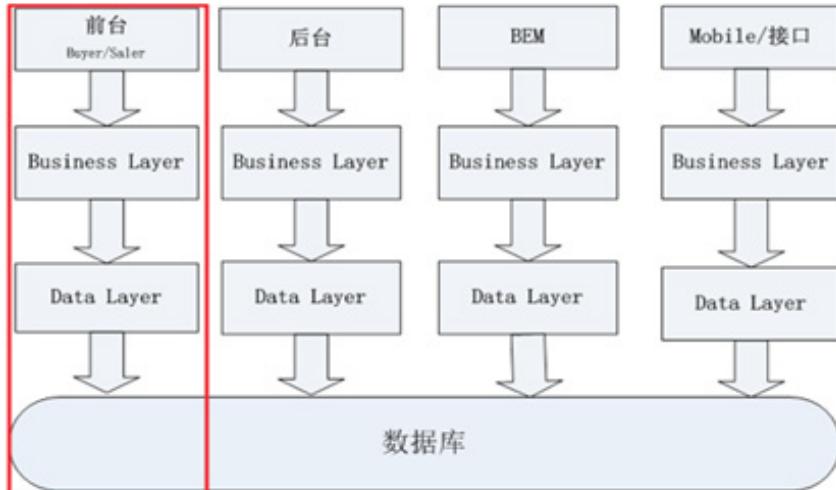
采购商的功能：

模块	功能	备注
系统管理	资料信息	
	投诉，建议	
机场服务	代换登机牌供应管理	
	已确认订单代换处理	
	换登机牌管理	
	代换登机牌	
采购机票管理	PNR 导入创建订单	
	航班查询及预订	
	团队票申请	
	申请改签及升舱查询	
	三字代码查询	
在线订单管理	当日最新订单	
	所有订单查询	
	采购报表下载	
	自由转账	
	退废票相关查询	
	申请行程单及查询	
	行程单领用及管理	
	保险管理	
短信平台	短信充值	
	短信发送	
	短信发送历史	
常旅客管理	常旅客添加	
	常旅客查询/修改	
常用软件下载	机票软件下载	

功能架构主要包括功能、角色和权限三部分。功能是企业服务，用户使用的每一个功能，就是企业的每一个服务。角色是用户操作的归类，功

能与角色的对应关系即权限。了解系统架构的现状，从功能架构开始。

应用架构



业务逻辑	应用	重复次数
预订逻辑	前台、接口、BEM、Mobile	4 次
订单处理逻辑	前台、后台、BEM、作业小应用、Mobile	5 次
产品供应逻辑	前台、后台、BEM、作业小应用、Mobile	5 次
财务结算逻辑	前台、后台、BEM、作业小应用、Mobile	5 次
公共逻辑	前台、后台、BEM、作业小应用、Mobile	5 次

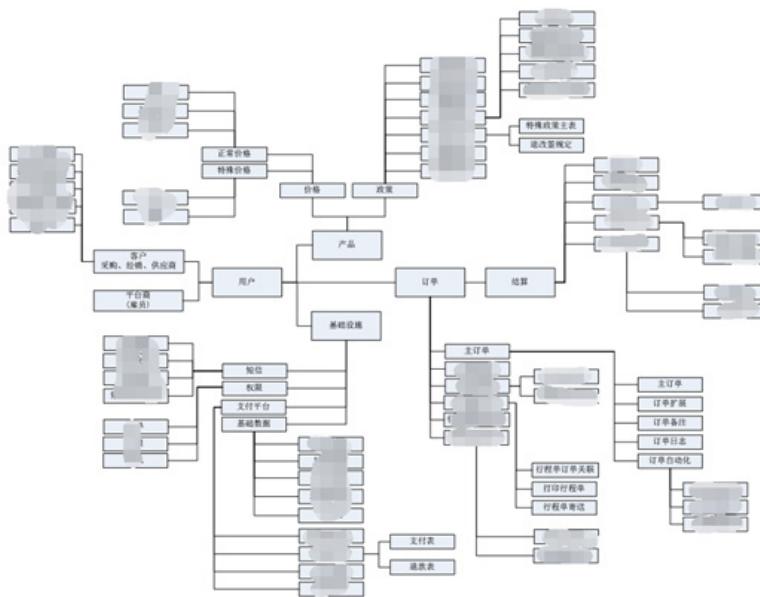
应用就是处理器，应用架构的内容包括现有架构图、Web 应用现状、作业小应用（Job）现状和接口架构。其中，接口是应用层面的关键，它是一个程序与另外一个程序交互的部分。

应用架构图表列出了哪些业务逻辑没有被重用，换句话说业务逻辑被多少个应用调用，就需要被重复开发多少次，一旦改了一个地方，就要同时改多个地方，导致系统开发效率非常低下。

各业务逻辑如预订逻辑，虽然被多个应用调用，但它们与应用是没有关系的，业务逻辑可以独立的存在，也可以寄宿于多个应用。业务逻辑是一个业务操作的抽象，而业务应用与业务部门共同完成了业务操作。

数据设计

100 多个数据库，一万多张表，能否使用一张 E-R 图来表示呢？它是可以的。



数据设计依赖于企业的数据，而不是数据库的设计，对企业数据适当做归类，会直接导致数据设计，最终画出 E-R 图，数据设计完成后，数据库设计就自然而然出来了。

超越库、超越表去看这张 E-R 图，可以看出它包括产品、订单、结算、用户和基础设施这五类数据。低层的 E-R 图可以变，但是高层的 E-R 图一般不会变化，因为它是根据你的业务模型而定，业务模型稳定，高层 E-R 图也是稳定的。

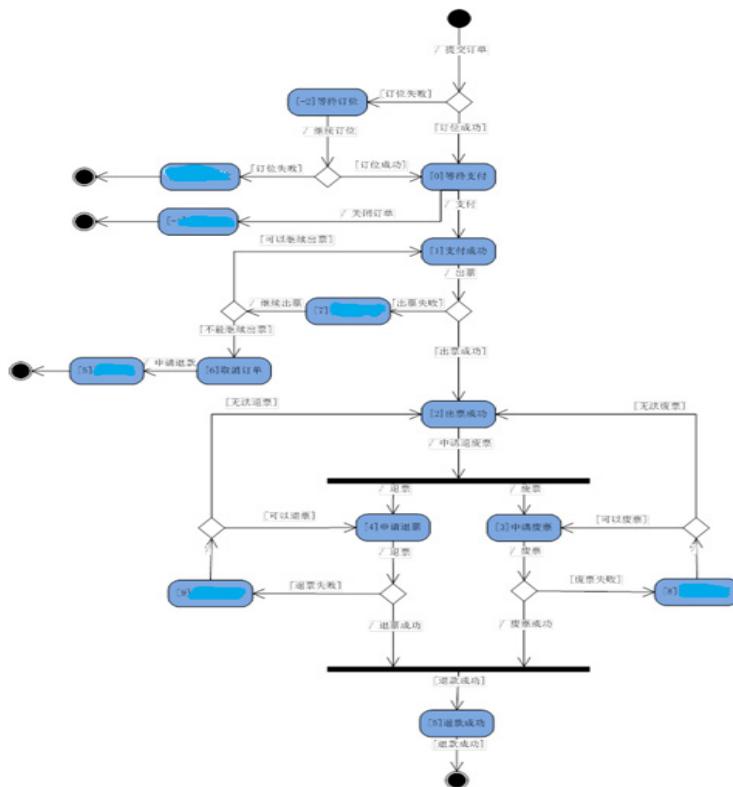
数据库只要早期设计得好，是可以做到易伸缩、易拆分的。下图从内往外看，一个框既可以是一个库，也可以是一个模块，还可以是一个表。

在业务发展的早期它可以是一个库，里面有 5 个模块，中期可以分为 5 个库，后期以更低级别可以分为更多的库，这与业务阶段及系统复杂度相关。在数据的设计完成后，数据库的设计也就很容易规划和调整。

以上是数据库、数据表之间的静态关系，接下来我们介绍数据的流转状态即状态图。通过数据状态图去了解现有数据流转变迁，如国内订单状态变迁图，这种图的价值不仅在于数据库层，还在于服务化。

图中的从等待支付到支付成功，中间有个支付行为，通过这个支付行为把数据状态变更为支付成功，否则继续等待，直到超时关闭订单。这个支付行为可以做成一个微服务，然后由不同的应用去调用。

物理架构



物理架构的内容主要包括 IDC 机房、机房之间访问关系、机房内服务器物理部署图、机房与业务分布、网站架构、数据库架构、集群清单和域名清单。

将这些内容以列表和图形方式整理出来，就会很容易了解发现问题，只有发现问题才能解决问题，特别是在全局体系架构方面，这也是表和图的价值所在。

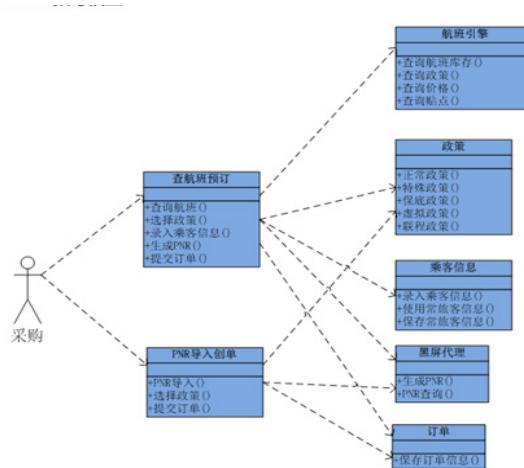
当时这家公司共有 5 个地区、8 个机房，虽然只有 200 多台服务器，但分布很散，导致物理结构复杂，通讯也很复杂。技改前故障不断，其主要的一个原因就是物理架构不合理，运维要占 60%、70% 的责任，当时却把责任归咎为应用架构，这是个错误的方向。

物理架构的不合理，应用架构是很难合理的，因为物理架构是我们的基础设施，位于最底层，下层为上层服务，运维要为应用服务，应用要为业务服务，业务要为客人服务。

领域模型

领域模型关注概念，关注职责、关注边界、关注交互，只有先确定职责和边界，交互才会很清晰。领域模型是针对现有问题域提出一个系统解决方案，然后在图表上建立完整的模型，如同用 AutoCAD 画的施工图纸一样。

领域模型属于概要设计阶段，对于单个应用架构设计，首先需要了解业务和功能需求、用例图、用例活动图，然后才是领域模型。业务流程图是对业务操作的抽象，领域图是对业务逻辑代码的抽象。



建立领域词汇是建立领域模型的第一步，它能统一词汇明确概念，以减少一词多义、一义多词的情况。概念一旦确定，再扩展属性和行为，然后把它当作一个单元与其它事物构建在一起，就会很容易形成模型，领域

模型与企业商务模型中的业务流程图有参考对应关系。

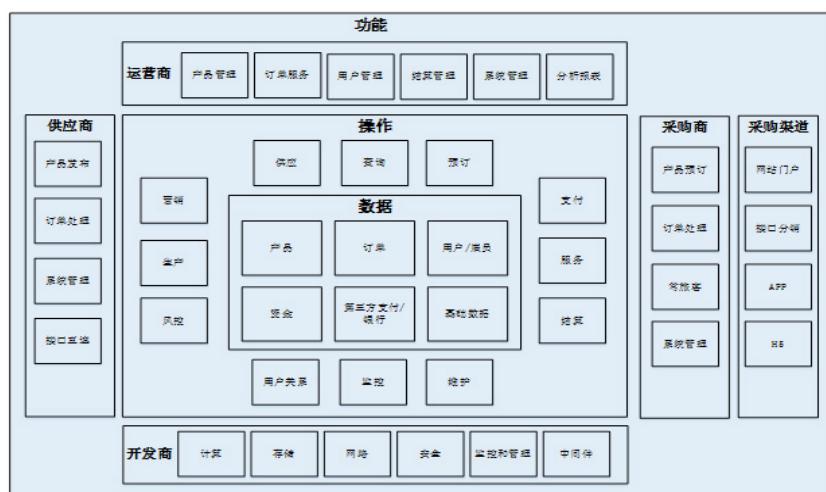
领域模型在实现时可大可小，在业务的早期，在系统比较小的情况下，它有可能是一个类。当系统做大了以后，它可能是个 DLL 库。再做更大一点的时候，它可能是一个服务，给不同的应用去调用。

每一个方法都有成为服务的潜质，特别是在系统中后期。领域模型是业务逻辑代码的施工图纸，它不仅有利于对现在系统业务逻辑的了解，同时也指导未来的架构改造。

架构规划

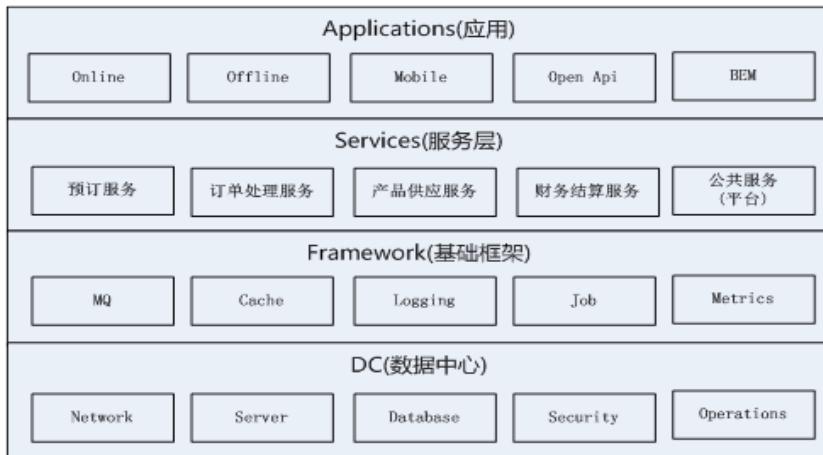
当我们了解了业务、了解了架构的现状，发现现有架构的问题，接下来就可以做中远期架构规划，以及架构的调整和具体实施。架构规划内容包括：顶层架构规划、网站功能规划、应用规划、SOA 规划、分层架构规划、数据库规划和物理规划等。

顶层架构规划



上图是顶层架构的俯视图和侧视图。第一张图是俯视图，坐在飞机上看，整个顶层架构最外层的是功能，中间的是业务操作，内层的是数据。

功能对应业务系统的用户界面，操作对应业务系统里的服务，数据对应业务系统的数据存储如数据库。



第二张图是剖面图，切一刀来看，上层是应用，中层是服务和框架，下层是基础设施数据中心。从图中的服务层可以看出，服务的归类跟业务流程的归类有很大关系。

网站功能规划

网站功能规划就是功能的重新划分，对照着架构现状，未来功能应该如何调整？如案例中的国内网站功能规划，分别画出了全局功能图、采购商功能图、平台商功能图和供应商功能图。

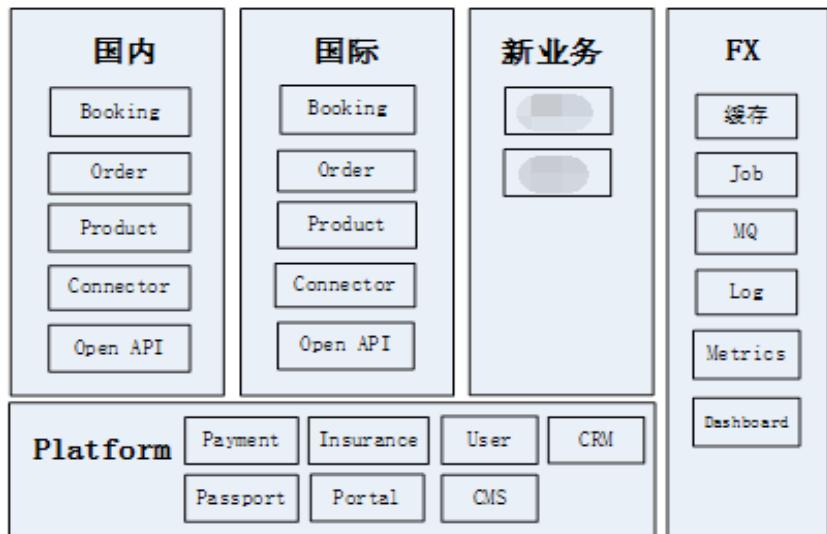
其实在做网站功能规划的时候，更多需要考虑现状，而不是未来调整的部分，如果没有很大问题，则不做调整，尊重历史。因为有些东西（如名称）用户已经使用很久了，调整往往比较难，合理大于准确。

应用规划

系统是什么？系统 = 元素 + 关系。应用架构是什么？应用架构 = 应用 + 架构。应用就是系统的最小单元，应用分类和应用编号则构成了应用关系即应用的架构。

如下图中的案例，应用分类新建了框架 FX 和公共业务系统 CBS，在原有的 200 多个应用中并没有这两个产品线，而是分布在了不同的业务线中，从而导致重复建设。

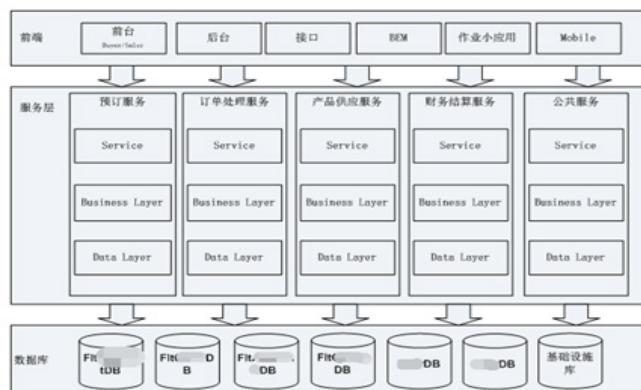
应用编号是给每个应用分配一个六位的数字 ID，就如同我们的身份证一样，头两位表示产品线，中间两位表示子系统，最后两位表示应用，



如 100206。应用编号是应用管理、依赖和追踪的基础，集中式日志和监控框架都有使用到应用编号。

SOA 规划

SOA 规划就是接口规划，它的归类与商务模型中的业务流程有参考对应关系。上图案例有五个服务中心：预订服务、订单处理服务、产品供应服务、财务结算服务和公共服务。



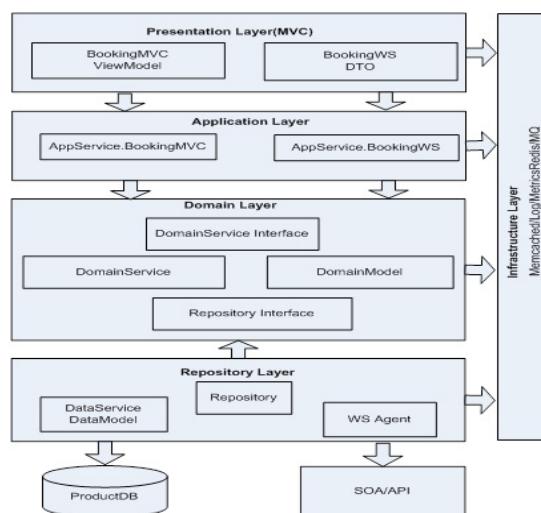
前端应用 ⁽¹⁾	调用的服务 ⁽²⁾	备注 ⁽³⁾
前台（采购商） ⁽⁴⁾	预订服务、订单处理服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
前台（供应商） ⁽⁴⁾	订单处理服务、产品供应服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
后台（平台商） ⁽⁴⁾	订单处理服务、产品供应服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
接口 ⁽⁴⁾	预订服务、订单处理服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
BEM ⁽⁴⁾	预订服务、订单处理服务、产品供应服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
作业小应用 ⁽⁴⁾	订单处理服务、产品供应服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾
Mobile ⁽⁴⁾	订单处理服务、产品供应服务、财务结算服务、公共服务 ⁽⁵⁾	⁽⁶⁾

每个服务只需要实现一套自己的逻辑，我们的前台、后台、接口、作业小应用等都可以调用，服务的逻辑跟我们的业务逻辑是一致的，修改代码的时候只需要改一个地方就可以影响到所有调用到这服务的前端应用。

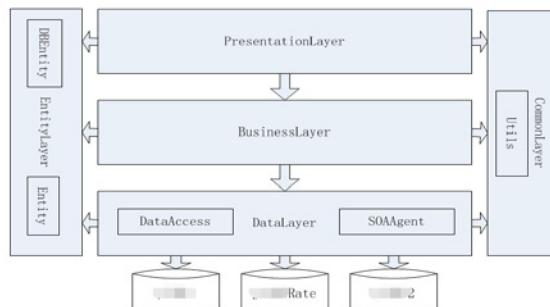
分层架构

分层架构看似很简单，但保证整个研发中心都使用统一的分层架构就不容易了。那么要如何去做，以达到提高编写代码效率、保证工程统一性的目的呢？

先简单介绍下当前两种比较流行的分层架构体系，一种是领域架构：仓储层（Repository Layer）、领域层（Domain Layer）、应用服务层（Application Layer）、表现层（Presentation Layer）和基础公共层（Infrastructure Layer），见下图。



另一种是相对传统地分为三层：数据层（Data Layer）、应用逻辑层（Business Layer）和表现层（Presentation Layer），见下图。



领域架构和三层架构之间有什么区别？我们是这样认为的，在早期我们做三层架构的时候，大都以表来做驱动的，在做领域架构的时候，大都以业务逻辑来驱动的，两者的区别确实比较明显。

但到了现在，如果都以业务逻辑为中心的话，实际上两者并没有本质区别。当时，我所在公司采用了第二种分层法，我们希望把分层做得极简，也就是说哪怕刚毕业进来的员工，在分层时基本上也不会乱。

而相对第一种分层法，第二种分层法简单很多。每一个应用的代码量都不应该很大，一旦工程变得过大，我们就会把它适当拆分，而不是全部放在一个单块应用里。

总之，我认为分层越简单，整个软件结构就越清晰，代码就越容易统一。把工程做得极简，才有利于复制，有利于业务的快速构建，有利于规模化、稳定可靠。

数据库规划

数据库	说明	备注
FT\DB	国内机票产品库	
FT\DB	国内机票订单库	
FT\DB	国内机票公共库	
FT\DB	国内机票结算库	
FT\LogDB	国内机票产品日志库	
FT\LogDB	国内机票订单日志库	
FT\LogDB	国内机票结算日志库	
IFI\DB	国际机票产品库	
IFI\DB	国际机票订单库	
IFI\DB	国际机票公共库	
IFI\DB	国际机票结算库	
IFI\LogDB	国际机票产品日志库	
IFI\logDB	国际机票订单日志库	
IFI\LogDB	国际机票接口日志库	
DB	用户库	将来合成为一个库，现在有国内、国际两个库
DB	雇员库	同上
DB(ip\city\mobile\airLine)	基础主数据	同上
DB	网站内容管理库	同上
DB	客户关系管理库	同上
DB	支付库	同上
LogDB	支付日志库	同上
DB	保险库	同上
DB(cache\job\config)	框架库	同上

数据库是整个信息系统中生命周期最长、最难修改的部分，所以要加

强规划。数据库的设计至少要提前两步，具体根据高层 E-R 图和数据设计来新建数据库，早建要比晚建好。

数据库调整的代价大、周期长，长时间产生的问题，需要长时间来解决，先在新库里解决新表，再根据当前业务和应用的需求，逐步调整旧表。

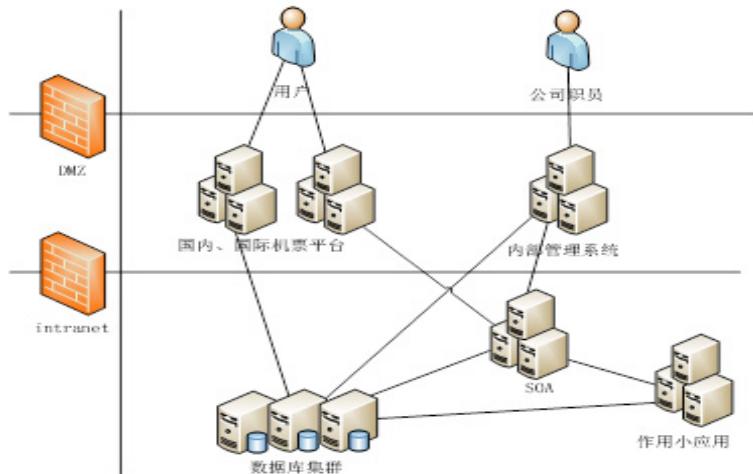
物理规划

物理架构的规划内容包括集群规划和域名规划。首先是集群规划。

20 倍规划、5 倍设计和 1.5 倍实施：规划和设计要大一些，但实施时小一些，这样不仅便于将来的扩展，也节省了当前的费用。

两个逻辑网络：一个内网和一个外网，两个负载均衡，两个防火墙，安全隔离内外网。

四条产品线：国际、国内、新业务以及公共业务，单点登录和企业支付网关等公共业务也属于一条产品线。



六个集群：Web 集群、SOA 集群、中间件集群、数据库集群、Job 集群和 ITD 集群。

以上横向集群与纵向产品线形成了一个矩阵结构，也基本确定了网络基础架构。对于域名规划。对内的域名该改的改，该停用的停用，该合

并的合并。对外的域名要尽量少改，要改的话也要有历史继承性（如跳转），要尽量减小对用户的影响。

其他

除以上架构规划外，还有一些其它重要项，如源代码管理规划、文档管理规划、技术选型和团队分工。为什么还要做这些呢？因为统一了源代码怎么放、每个部门的文档怎么放、将来要用什么工具版本，才利于团队的协作，基于统一的环境才能有更高层次地提升。

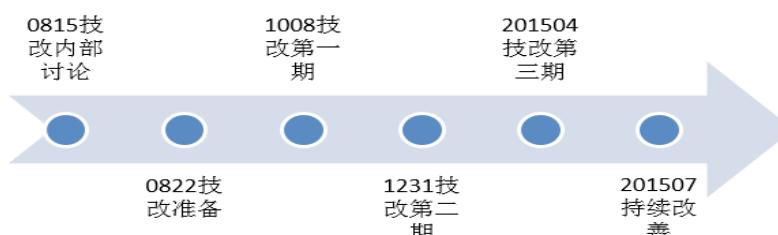
对于团队分工，需要逐步对齐组织架构与系统的架构规划。对于技术选型，需要注意中间件的引进，要有节奏性，力量要相对集中，要小规模试点，找非核心项目，试用成功后再进行大规模推广。

架构实施

做完架构规划后，就是架构实施落地了。我们的架构实施整体思路是：树目标、给地图、立榜样、抓重点、造文化、建制度、整环境、组建架构部。

架构部内招几名老程序员，外招几个架构师。内部走出去，提高眼界。外部牛人请进来，落地了解历史和业务。技术建议是：SOA 服务化、基础设施平台化、公共业务服务化、加强项目概要设计。

当研发团队达到 200 多人、有了几百个应用，且在故障不断的情况下，不能与以前一样没有设计就开始编码，而是做加强项目概要设计及评审。后面的补与前面的防，两手都要抓，两手都要硬。



具体计划是：Roadmap 分步实施，改造一期、改造二期、改造三期，

近细远粗、实事求是、逐步细化、逐步完善。不断立技术改造项目，不断将技改与业务研发项目相结合，技改即是工单、工单即是技改。避免对业务过多地影响，并不断有业务价值输出，这是架构改造得以持续实施的关键！

以上简单地介绍了总体架构的编写方法，我们的编写思路是先了解业务，建立企业商务模型，主要包括静态的商务主体、组织架构和动态的商务运作模型和业务流程。

接着了解架构现状，建立现有信息系统模型，主要包括功能架构、应用架构、数据设计和物理架构。一个是商务，一个是电子，两者即是整个公司的电子商务系统。

然后在企业商务模型和现有系统模型之上建立领域模型，领域模型它相对稳定，直接指导着接下来的架构规划，最后一定要落地即架构实施。

附档是去掉敏感信息后的真实案例，它的价值如下：

- Big Picture，全局蓝图，起到方向性和指导性；
- 将隐性知识显性化，方便传达、广而告之；
- 对于新员工的价值，快速入门；
- 对于老员工的价值，了解全局，过程梳理，然后专注于自己的部分。

关于企业总体架构，你可以参考标准 TOGAF(开放组体系结构框架)。其实，我们是在完成那份文档后才知道 TOGAF，它们之间有很多相似之处和不同之处。

TOGAF 的内容主要包括业务架构、应用架构、数据架构和技术架构，而我们当时只是以解决公司系统架构问题为导向、以时间为主线，内容有企业商务模型、架构现状、领域模型、架构规划和架构实施。

方法论很重要，但看到事物本身的特点，深入问题以及找到解决办法更为重要。欢迎点赞和拍砖！

案例参考：<https://github.com/das2017/TopArchDemo>

本系列文章涉及内容清单如下，其中有感兴趣的，欢迎关注。

- 开篇：[中小型研发团队架构实践三要点](#)
- 缓存 Redis：[Redis快速入门及应用](#)
- 消息队列 RabbitMQ：[如何用好消息队列RabbitMQ？](#)
- 集中式日志 ELK：[中小型研发团队架构实践之集中式日志ELK](#)
- 任务调度 Job：[中小型研发团队架构实践之任务调度Job](#)
- 应用监控 Metrics：[应用监控怎么做？](#)
- 微服务框架 MSA：[这是你心心念念的.NET栈的微服务架构实践](#)
- [搜索利器 Solr](#)
- [分布式协调器 ZooKeeper](#)
- 小工具：Dapper.NET/EmitMapper/AutoMapper/Autofac/NuGet
- 发布工具 Jenkins
- 总体架构设计：电商如何做企业总体架构？
- 单个项目架构设计
- 统一应用分层：[如何规范公司所有应用分层？](#)
- 调试工具 WinDbg
- 单点登录
- 企业支付网关
- 结篇

作者介绍

张辉清，10 多年的 IT 老兵，先后担任携程架构师、古大集团首席架构、中青易游 CTO 等职务，主导过两家公司的技术架构升级改造工作。现关注架构与工程效率，技术与业务的匹配与融合，技术价值与创新。



在微信上关注我们



InfoQ

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



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



聊聊架构

以架构之“道”为基础，呈现更多的务实落地的架构内容。

关注「聊聊架构」

和百位架构师共聊架构



细说云计算

探讨云计算的一切，关注云平台架构、网络、存储与分发。这里有干货，也有闲聊。

关注「细说云计算」

回复“群分享”，
看云计算实践干货分享文章



AI前线

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

关注「AI前线」

回复“AI”，下载《AI前线》系列迷你书



前端之巅

紧跟前端发展，共享一线技术，不断学习进步，攀登前端之巅。

关注「前端之巅」

回复“京东”，看京东如何做网站前端监控



移动开发前线

关注移动开发领域最前沿和第一线开发技术，打造技术分享型社群。

关注「移动开发前线」

回复“群分享”，看移动开发实践干货文章



高效开发运维

常规运维、亦或是崛起的DevOps，探讨如何IT交付实现价值。

关注「高效开发运维」

回复“DevOps”，四篇精品文章领悟DevOps





架构师 月刊 2017年12月

本期主要内容：Java 10 新特性前瞻；阿里重启维护 Dubbo 了；为什么原生应用开发者需要关注 Flutter；商汤科技杨帆：AI 落地的关键是算法闭环；Linus 怒喷谷歌安全工程师。



深度学习利器 TensorFlow程序设计

本书介绍 TensorFlow 程序设计中的关键技术。



架构师双十一特刊 电商大促技术探秘

今年我们尝试将技术内容转换为语音并推出了极客时间 APP，思考和创新不会止步于此，希望来年双十一专题能以更棒的内容形式与大家再见面。



架构师特刊 范式大学

本期主要内容：构建商业 AI 能力的五大要素；判别 AI 改造企业的 70 个指标；用最小成本 验证 AI 可行性；企业技术人员如何向人工智能靠拢？打造基于机器学习的推荐系统。