

架构师

ARCHITECT

特刊 |

微服务与DevOps技术内参

技术选型

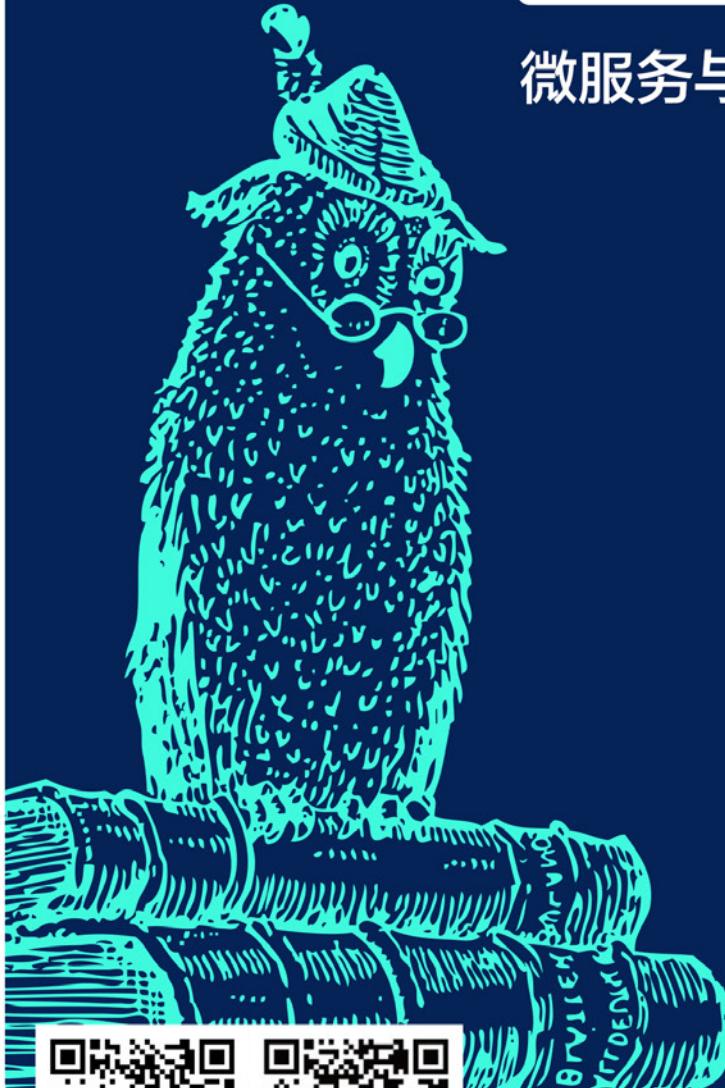
云平台的微服务治理框架
云平台的基础消息处理架构

技术洞察

谈DEVOPS对于企业IT的价值
谈元数据驱动的微服务架构

技术实践

基于微服务架构的技术实践
谈API网关的背景、架构以及落地方案
实施DEVOPS从哪里开始？



Geekbang 极客邦科技 | InfoQ

PRIMETON 普元

■ 卷首语

大道至简—微服务时代的技术美学

从结绳记事到云端计算，计算工具的演化经历了由简单到复杂、从低级到高级的不同阶段，贯穿始终的是人们一直在试图用人力以外的力量，简化人的工作，提升生产与生活的效能。然而，“科学每解决一个问题，都要引发十个新问题”。软件定义了一个无限美好的未来，却将人们拖入充满泥淖的现实。

虽然摩尔定律早就揭示了硬件进化的潜力，《没有银弹》却告诉我们一个冰冷的现实：没有任何一项技术或方法可以让软件工程的生产力在十年内提高十倍。数以百万计的软件从业者每天创造着浩如烟海的软件代码，却使得软件这一计算机的灵魂变得越来越纷繁复杂。

其实软件本应是美的，而美的软件应该是简单的，正所谓道生一，一生二，二生三，三生万物。

简单获得的体验之美

“（软件之美）在于它的功能，在于它的内部结构，在于团队创建它的过程。”然而美的价值最终在于体验，风景在旁人看不到它的时候，便

不能算是“风景”。

Gartner 为我们预言了一个人、物和商业深度连接的世界，在这样的世界中，因为云计算技术和架构的演进和发展，软件的体验正在如同水电一样无关基础，即用即得；移动终端的广泛应用，让人们可以无关时空，随时随地的获得功能和应用的交付，而在这些功能和交付的背后，不再需要动辄以千万计的庞大团队作为支撑，DevOps 和微服务满足了应用的碎片化，时间的碎片化，人们关注度的碎片化，使软件的生产能力可以给予客户所需要的即得感，企业和组织可以以“周”或以“天”甚至以“分”为单位快速实验和探索，在不断尝试中获得更加契合客户需求的软件。让软件表达出简单获得的体验之美。

简单创造的价值之美

程序员们日以继夜，诠释着披星戴月的含义，却不断的沉沦在重复搭建环境、重复系统部署、重复环境验证、重复代码开发等等的炼狱之中，“感觉身体被掏空”的绝望如影随形。人类发明并建造计算机，努力对其开发和优化，是为了让计算机可以更好地理解人类行为，模拟人类的学习和表达过程；在智能时代，计算机的认知能力开始超越人类的个体经验，甚至可以为人类行为提供帮助和预测。DevOps、ChatOps、OpsDev 让生产运营者做到聊天式的开发运维一体化，让提供者做到自动化的部署供给，让消费者做到自助式的开通使用，让程序员轻松地超越极限，充分释放自己的智力、想象力和创造力，从码农成为商业价值的创造者，让软件迸发出简单创造的价值之美。

简单架构的设计之美

《营造法式》为中国古建筑的结构设计、工程管理奠定了基础，灿若

星辰的中国古典园林体现了建筑执着于简单的美学理念，砖、瓦、琉璃与建筑构件，以最简化的分解让没有生命的木方泥胎诞生了横亘古今的美感。软件体系的“营造法式”却仍旧“立而望之，偏何姗姗其来迟。”。

出于项目实施时间、投入资源等方面的限制，大型软件往往以实现若干个具体的用户功能需求为目标。日复一日，随着用户功能要求的变化，软件变得面目全非。任何系统，在自然情况下，都是从有序到无序，但生物可以通过和外界交互，主动进行新陈代谢，继续生存。软件系统随着功能越来越多，调用量急剧增长，整个系统逐渐碎片化，越来越无序，最终无法维护与扩展，人们没有时间，也没有精力去追求软件的美学目标。所以系统在一段时间后必须即时干预，避免野蛮生长。然而大型软件项目已成为大量代码的随机而无序的堆积。工程师一旦完成项目，就恐避之不及，不愿再去碰自己几个月来夜以继日的劳动成果。

架构的本质就是让系统开发、维护变得有序，而好的架构一定是高度抽象的、易于理解的、合理定位的、整合有机的、面向未来的。微服务架构模式将大型的、复杂的、长期运行的应用程序构建为一组相互配合的服务，每个服务都可以很容易得到局部改良。让软件折射出简单架构的设计之美。

简单协作的生态之美

“我住长江头，君住长江尾。日日思君不见君，共饮长江水。”企业 IT 中，战略与实现的鸿沟，业务与技术的鸿沟，开发与运维的鸿沟，让协作的各方难以互相理解和交流。对于开发团队与运维团队而言，世界上最远的距离，不是我站在你的对面你却听不到我的话，而是我和你要了苹果 7 代，你却给了我 7 袋苹果。DevOps 提倡开发和 IT 运维之间的高度协同，从而

在完成高频率部署的同时，提高生产环境的可靠性、稳定性、弹性和安全性。

通过纵向协作，DevOps 强调的重点是跨工具链的「自动化」，最终实现全部人员的「自助化」服务。

通过横向协作，DevOps 强调的重点是跨团队的「线上协作」，也即是通过 IT 系统，实现信息的「精确传递」。

DevOps 不仅打通了开发运维之间的部门墙，更实现了应用全生命周期的工具链路打通、跨团队的线上协作能力，让软件衍生出简单协作的生态之美。

软件本应是美的，消费者的体验，工程师的价值，架构师的设计，部门间的生态，无不因简而实，因简而预，因简而美。因此，本期《架构师》，我们试图在技术的极简美学方面作一些尝试和分享。如何通过元数据这种“数据的数据”来抽象和简化微服务架构认知的维度；如何用简单的几种模式，看透数据的最终一致性；如何将 DevOps 软件交付的过程和环节，映射为最简单的企业 IT 生产元素；如何用几条简单的规则，来衡量选择开源架构的得失，简而言之如何确立数字化时代的软件技术架构。

我们今年一直在尝试做一件事，用开放和分享，来简化技术创新的过程，我们将面向商用的企业级云计算平台的设计文档，过程文档，技术思考，通过社群和公众平台的方式全面开放和分享，期待更多的身处企业 IT 环境的工程师和架构师，可以和我们一起，追寻软件的初心，寻求软件之美，合作创新，让未来触手可及。

至言不繁，大道至简。

普元信息 CTO 焦烈焱

目录 | Contents

技术选型

- 8** 云平台的微服务治理框架

- 16** 云平台的基础消息处理架构

技术洞察

- 26** 谈 DevOps 对于企业 IT 的价值

- 32** 谈元数据驱动的微服务架构

技术实践

- 42** 基于微服务架构的技术实践

- 57** 谈 API 网关的背景、架构以及落地方案

- 64** 实施 DevOps 从哪里开始?

技术选型

云平台的微服务治理框架

宋潇男

如何做开源技术选型？

很多同学在做技术选型的时候，往往过于关注技术 / 功能上的比较，陷入技术细节和功能特性上的争论。比如 A 产品有个 X 功能，看起来很棒，B 产品有个 Y 功能，也不错，选哪个，好纠结……或者 A 产品的当前版本看起来不错，B 就很一般，可是 B 的 Roadmap 里写，下一个版本会有个很强大的功能出来，是不是要再等等看，好纠结……

有时候勉强选了 A，又看到 B 发展的也不错，心里不踏实。

其实在我们看来，技术 / 功能只是技术选型过程中需要考量的诸多维度中的一个，只要这些开源产品大体上能满足我们的需求，架构上没有明显的缺陷，开发语言和现有团队比较匹配、Roadmap 比较完善，就没什么大问题，就可以进入其他维度的考量。

我们认为，技术 / 功能在技术选型中的权重，可能只有四分之一。有很多技术 / 功能上非常领先的开源项目，并没有得到很好的发展，比如 ZFS，比如 CloudStack，令人惋惜，这里不一一列举。那么技术选型过程中，

除了技术 / 功能之外，我们还应该关注哪些事情呢？

项目的运作模式

开源项目的运作模式中，我们重点关注以下三点：

- 1、使用哪种开源 License；
- 2、开发模式和测试模式；
- 3、被一家公司掌控还是松散的社区决策？

其中**最重要的是 License**。在很多技术人员的眼中，License 问题依然没有得到足够的重视。（对这个问题感兴趣的同学可以移步 <http://choosealicense.com/> 查看各种开源 License 的差异。）

开发模式，值得关注，但是一般知名的开源项目，在开发模式上都不会有太大的问题，**更重要的是测试模式**。很多开源项目自身不重视测试，把填坑的事情丢给参与厂商，这种项目，如果贵司不是动不动就能派出几十人上百人的大厂，必须慎用。会认真做测试框架、定期发测试报告的项目，必须好评。

项目运作是被一家公司掌控还是松散的社区决策，以前都说大教堂不如集市，那是因为以前大教堂基本上不搞开源，搞开源的大教堂和集市，就难说哪个好了，参与过 OpenStack 厂商扯皮的同学，应该对这个深有体会。

技术提供者的产业背景

在项目技术提供者的产业背景中，我们重点关注以下三点：

- 1、技术提供者的产业经验。
- 2、自己有没有大规模使用？
- 3、是从自身需求沉淀出来的产品还是按设想的需求开发的产品？

现在都讲“吃自己的狗粮”，很多最为成功的产品，都是在自己内部的长期的、大规模的使用中反复锤炼，再发布给公众使用的，最好的例子

就是 AWS。从自身的实际需求出发的、给自己做的产品，往往会比按设想需求出发的、给客户做的产品更好。所以我们往往更加青睐大型互联网公司释放出来的开源项目，比如 Netflix 的一系列开源项目。

生态环境

在项目所处于的生态环境中，我们重点关注以下三点：

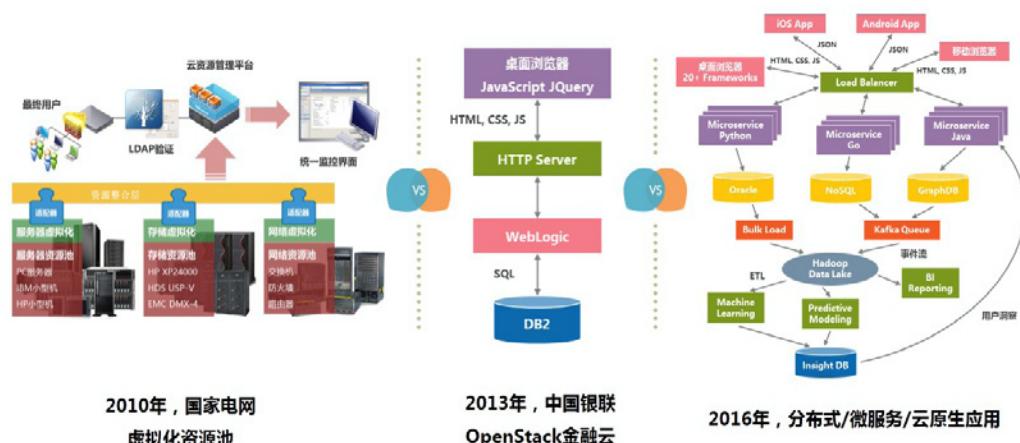
- 1、技术和技术提供者在产业链中的位置
- 2、与友商的合作 / 竞争关系
- 3、是众望所归还是单打独斗？

互联网时代，没有生态、就等于没有未来。良好的合作、清晰的分工界面，会让项目得到更好的发展，总想占点上下游的便宜，动别人的奶酪，或者妄图以一己之力对抗全行业，当然也有成功的，但是概率真的很低。

我们的选择

现在回到故事的大背景中，看下我们为什么选择 Kubernetes 作为普元新一代云平台的微服务治理框架。

首先，以下为我们在云计算项目中遇到的需求：

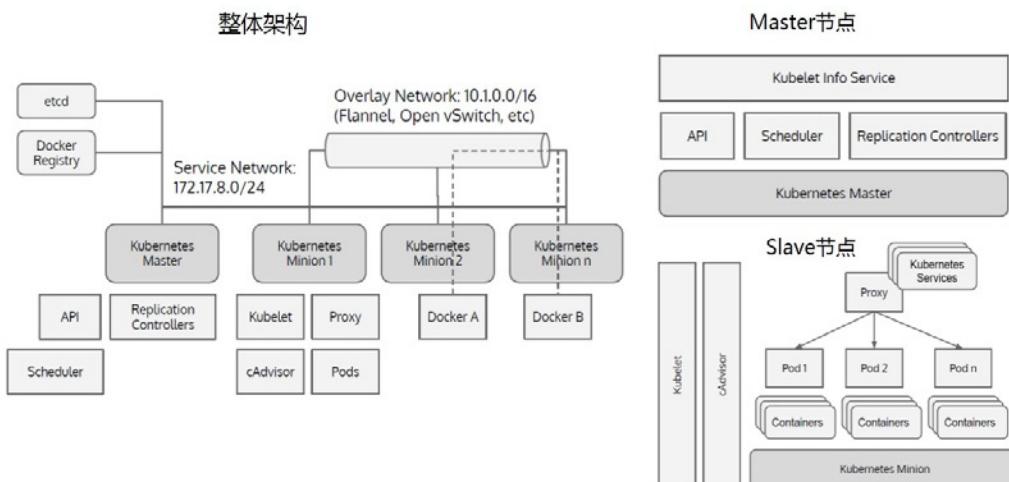


云计算技术经过近十年的变迁，需求重点已经从早期的虚拟化资源池管理、单体应用的“栈”管理，过渡到了微服务的“图”管理。

管理微服务时，我们需要对这些微服务和它们的调用关系进行注册、为其分配资源、创建一定数量的节点副本、并发布到集群中去，同时还要为其配置好网络和负载均衡，使这些微服务能够被外部访问；在这些微服务的运行过程中，需要始终保持其可用性，一旦有节点出现问题，需要立即创建新的节点将其替换掉；运行过程中需要对这些微服务进行监控和日志收集；在负载发生变化的时候，还要能够迅速调整资源分配。

大体上能满足这些需求的开源项目有 Kubernetes、Mesos Marathon、Docker Swarm、OpenShift、Cloud Foundry 等等。

我们重点看一下 Kubernetes：



可以看到，Kubernetes 使用了较为常见的 Master-Slave 架构，在容器之上又封装了一层 Pod 结构，很好地适应了多容器服务，也符合 Unix 的进程模型。Pod 通过 Label 标识为服务（Service），概念简洁明了而且非常灵活。

经常有人问，Mesos 资源调度能力更加强大，而且产品推出的时间比较久，更加成熟、稳定，为什么不选择 Mesos？还有人拿出了下面这张对比图，证明 Mesos 的功能更为强大。

需要注意的是，这张图只对比了资源调度，而 Kubernetes 提供的是从资源调度，到服务发放、变更、退休，到网络管理的全方位能力，而

Framework	Architecture	Resource granularity	Multi-scheduler	Pluggable logic	Priority preemption	Re-scheduling	Over-subscription	Resource estimation	Avoid interference
Open Source Orchestrators	Kubernetes	monolithic	multi-dimensional	N[v1.2, DD	Y[DD	N[Issue	N[Issue	Y[DD	N N
	Swarm	monolithic	multi-dimensional	N	N	N[Issue	N	N	N N
	YARN	monolithic/two-level	RAM/CPU slots	Y	N[app-lvl. only]	N[DRA	N	N[DRA	N N
	Mesos	two-level	multi-dimensional	Y	Y[framework-lvl.]	N[DRA	N	Y[v0.23, Doc	N N
	Nomad	shared-state	multi-dimensional	Y	Y	N[Issue	N[Issue	N[Issue	N N
	Sparrow	fully-distributed	fixed slots	Y	N	N	N	N	N N
Closed-Source Orchestrators	Borg	monolithic ^[7]	multi-dimensional	N ^[7]	N ^[7]	Y	Y	Y	N N
	Omega	shared-state	multi-dimensional	Y	Y	Y	Y	Y	N N
	Apollo	shared-state	multi-dimensional	Y	Y	Y	Y	N	N N

Figure 2: Architectural classification and feature matrix of widely-used orchestration frameworks, compared to closed-source systems.

Mesos 仅仅是资源调度，服务管理要借助 Marathon，而网络管理能力聊胜于无。其他类似的技术，同理，这里就不一一展开了（上面那张图来源于剑桥大学 CambridgeSystems at Scale 博客的一篇文章，感兴趣的同学可以移步 <http://www.cl.cam.ac.uk/research/srg/netos/camsas/blog/2016-03-09-scheduler-architectures.html>）。

再来看一下 Kubernetes 的 Roadmap:

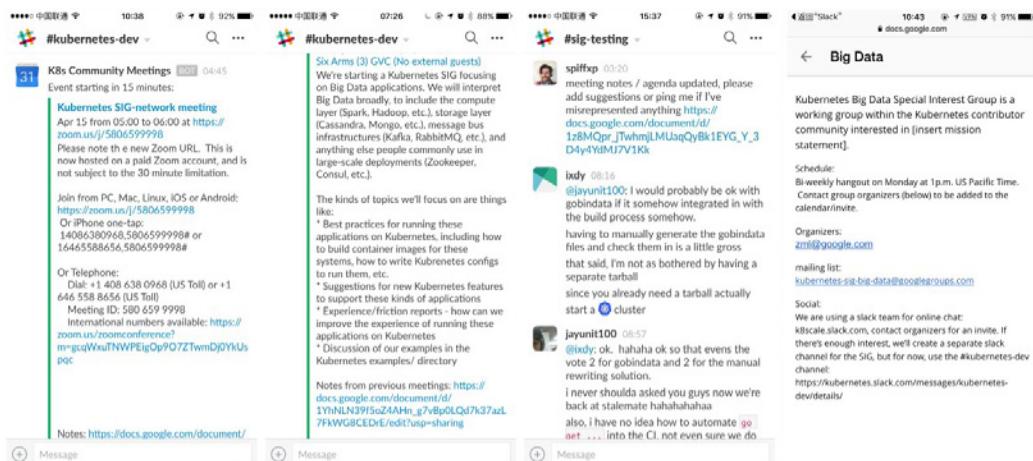
New and coming soon

- Cron (scheduled jobs)
- Custom metrics
- “Apply” a config (even more declarative)
- Interactive containers
- Bandwidth shaping
- Third-party API objects
- Scalability: 1000 nodes, 100+ pods/node
- Performance
- Machine-generated Go clients (less deps!)
- Volume usage stats
- Multi-zone (AZ) support
- Multi-scheduler support
- Node affinity and anti-affinity
- Multi-cluster federation
- API federation
- More volume types
- Private Docker registry
- External DNS integration
- Volume classes and auto-provisioning
- Node fencing
- DiY Cloud Provider plugins
- More container runtimes (e.g. Hyper)
- Better auth{n,z}
- Network policy (microsegmentation)
- Big data integrations
- Device scheduling (e.g. GPUs)

我们关注的功能，例如 CustomMetrics、Multi-zone、Multi-scheduler、Node affinity 等，都在 Roadmap 之中，还有 Device Scheduling 这种意想不到的功能，三个月一个版本，进度和我们自己的产品规划也比较吻合。

技术这关到这就算是过了，接下来我们看下 Kubernetes 项目的运作模式。

Kubernetes 使用 Apache License，没有对参与厂商做太多的限制。虽然很多厂商在积极贡献代码，但是控制权还是在 Google 手上，不会出现某些“集体决策”项目因为要不要做一个特性一堆厂商反复扯皮的问题。开源项目的研究过程做到开放透明自然不必多说，在此之上，Kubernetes 还做到了接地气，在沟通协作上没有使用所谓的“极客工具”，而是直接使用了 Slack、Zoom 这样的流行 App，每次会议的会议纪要都会发布到 Google Docs 上，让开发人员之外的技术爱好者也能很方便的获取项目进展的第一手资料。至于测试，有一伙人专门在搞测试框架，对测试的重视程度高于一般开源产品。



评估过了项目的运作模式，我们再来看一下技术提供者的产业背景。大家都知道近两年这一波容器浪潮的推手是 Docker，Docker 俨然成为容器的代名词，那么 Google 的位置又在哪里呢？以下为来自 Wikipedia 的截图。

很多同学可能没注意过，容器的两个根本核心技术之一，cgroups，发起者就是 Google，最初的目的是用来管理 Google 复杂庞大的互联网服务，后来贡献给 Linux，至今已经快十年了。而 Kubernetes 就是源

cgroups

From Wikipedia, the free encyclopedia

cgroups (abbreviated from **control groups**) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

Engineers at Google (primarily Paul Menage and Rohit Seth) started the work on this feature in 2006 under the name "process containers".^[1] In late 2007, the nomenclature changed to "control groups" to avoid confusion caused by multiple meanings of the term "container" in the Linux kernel context, and the control groups functionality was merged into the Linux kernel mainline in kernel version 2.6.24, which was released in January 2008.^[2] Since then, developers have added many new features and controllers, such as support for kernfs,^[3] firewalls,^[4] and unified hierarchy.^[5]

cgroups

Original author(s)	Paul Menage, Rohit Seth
Developer(s)	kernel.org (Tejun Heo et al.) and freedesktop.org
Initial release	2007. 9 years ago
Written in	C
Operating system	Linux
Type	System software
License	GPL and LGPL
Website	www.kernel.org/doc/Documentation/cgroup-v1/ and www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/

自于 Google 内部使用、经过多年的锤炼的集群容器管理系统 Borg，所以 Kubernetes 才能做到架构简洁、适应性强而且极具扩展性。有句话说好的架构不是设计出来的，而是进化出来的，这里的反面教材就是 Docker，有些同学可能了解 Docker 的 C/S 架构和网络设计，带来了多少麻烦，这就是有没有产业经验的区别。

前段时间的一个新闻.....

Intel与合作伙伴CoreOS、Mirantis联手打造“通用资源调度平台”

Intel在容器和虚拟化一直走在实践的最前方。虚拟化技术有两大分支：完全虚拟化和容器 Container，两者各有优劣，去年Intel 推出的 Clear Linux 项目，却声称同时拥有两者的特点。

CoreOS Collaborates with Intel to Deploy and Manage OpenStack with Kubernetes

CoreOS' Tectonic to Help Make OpenStack on Kubernetes Ready for the Enterprise; Provides a Path to GIFFE



CoreOS的CEO Alex Polvi对Intel云开放日的与会者发表了自己的意见，他觉得客户需要的是一个更广阔的，包括所有工作负载的部署，可以将 Kubernetes作为管理层，OpenStack作为服务的提供者。“这是专门提供给需要一套基础设施管理所有服务的公司”，Polvi还说，“他们在容器上和虚拟机上都会受益良多。”

Intel、CoreOS、Mirantis这一伙伴关系的三个成员并没有透漏调度程序的技术细节。但是基于Intel之前发布的Clear Containers雏形，再联系到今天发布会上简要的发布报告，Intel是想减少硬件基础设施，其VT的目标就一目了然了。原始的设计是只提供微指令资源到中间件，完全跳过操作系统，VT就会有机会提供资源调度到容器平台，尤其是像CoreOS这样的设计架构。

Clear Containers设计使用了VT技术，但是容器行业的规范还是围绕着OCI来铺开的。从Intel的角度来看，Clear Container可能被当做一個可替换的容器系统，这还是蛮危险的。这样来看，CoreOS的定位是十分妥帖的。

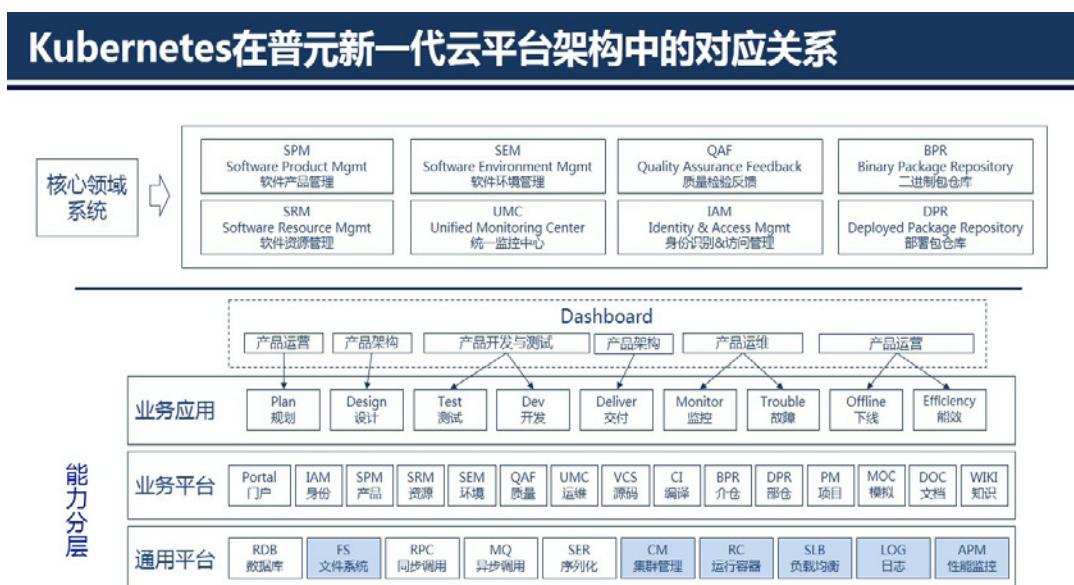
所以这一新型的合作伙伴关系可能会导致Mirantis OpenStack部署在 Kubernetes上。至少，所有形式的容器负载，以及CoreOS的rkt，Intel的Clear Container会在同一台服务器上部署是最好的。最新公布的Intel至强系列服务器处理器将提供使用者自主控制内存的新特性。这一特性会更好的满足扩展实时的工作负载——像的达克交易满足实时性需求一样。

最后，我们来看生态。Kubernetes 首先是和 CoreOS 联合，形成名为 Tectonic 的端到端容器管理解决方案，而 CoreOS 又是 etcd、rkt、fleet、flannel 等一系列优秀开源项目的领头羊，这些开源项目彼此之

间做到了很好的对接，避免了使用开源软件时经常碰到的“裁剪”问题。而前段时间的一个新闻更是将 Kubernetes 所在的生态环境进一步强化。

总结

经过上文的分析，可以看到，Kubernetes 在技术 / 功能、运作模式、产业背景、生态等四个维度有着较为均衡的优势，所以我们选择 Kubernetes 作为普元新一代云平台的微服务治理框架。下图中浅蓝色的模块基于 Kubernetes 的能力开发：



作者简介

宋潇男，现任普元云计算架构师，曾任华为云计算产品技术总监。曾负责国家电网第一代云资源管理平台以及中国银联基于 OpenStack 的金融云的技术方案、架构设计和技术原型工作。

云平台的基础消息处理架构

臧一超

我们身处在一个数字化商业的时代，作为一名 IT 工作者，如何保证我们所设计的系统、开发的服务在面对复杂不确定的网络环境中，还要去交付准确可靠稳定的服务？

我们在数以千计微服务支撑的云计算平台下，怎么考虑不确定性的并发请求、超量请求、请求的不断积压？

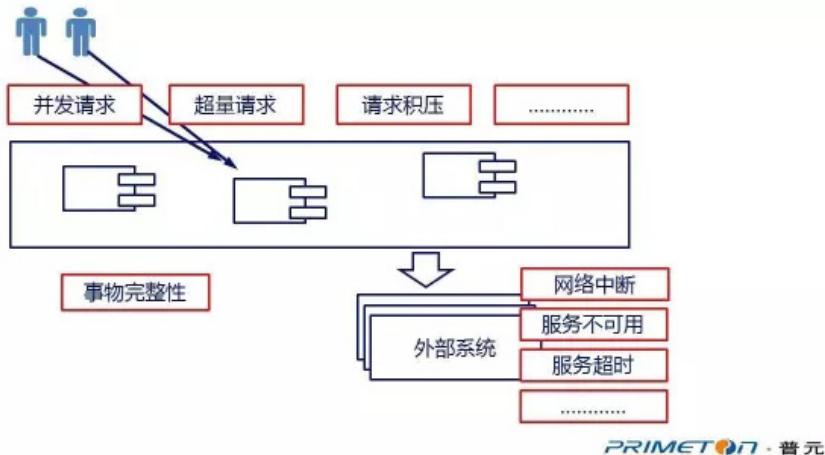
同时，我们还要兼顾外部所连接的商业功能网络中断、服务不可用、服务超时、事务完整性等等一系列的问题。那么，如何来解决现实世界中的这些问题呢？

传统的并发编程模型主要有两种：**一种是 Thread-based concurrency，另一种是 Event-driven concurrency。**

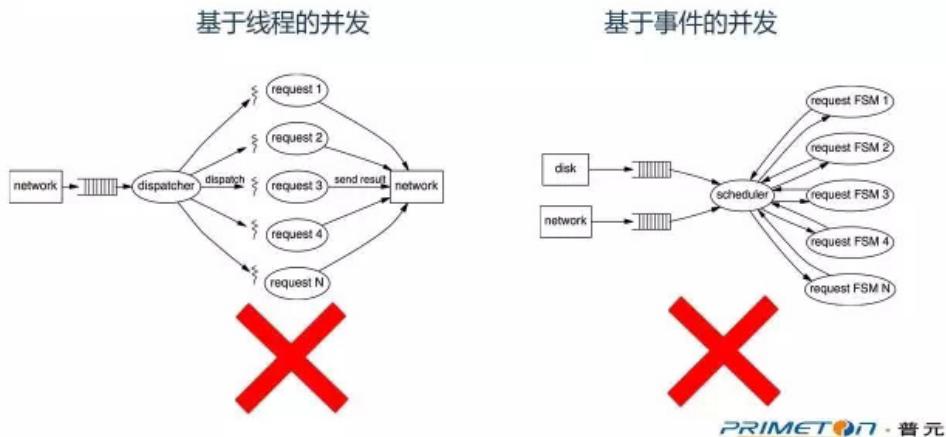
总结下两种模式的特点：

基于线程的并发：每个任务一线程直线式的编程使用资源高昂，context 切换代价高，竞争锁昂贵，太多线程可能导致吞吐量下降，响应时间暴涨；

如何在不确定的环境中交付准确的服务？



基于线程与事件的并发模型无法适应现有需求

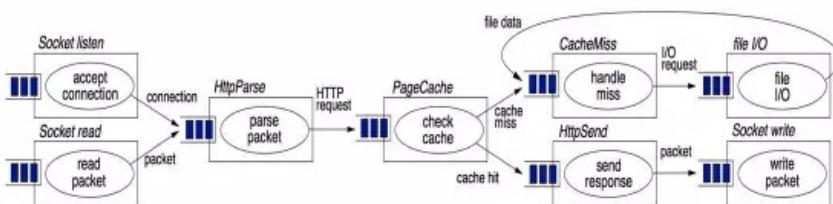


基于事件的并发：单线程处理事件的每个并发流实现为一个有限状态机应用直接控制并发负载增加的时候，吞吐量饱和响应时间线性增长。

SEDA 架构是目前云计算、微服务时代下一种优秀的消息处理架构，而且历经考验，稳定可靠。

SEDA 架构的核心思想：把一个请求处理过程分成几个 Stage，每个 Stage 可由不同的微服务进行处理，不同资源消耗的 Stage 使用不同数量的线程来处理，微服务之间采用异步通讯的模式。

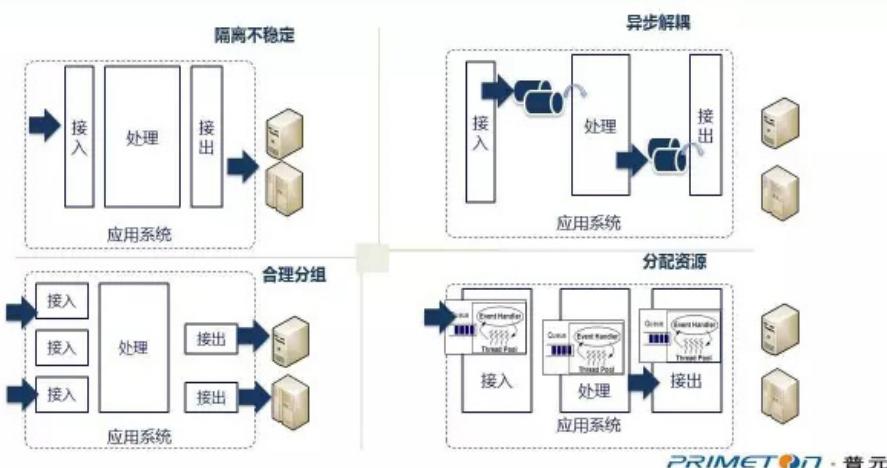
基于SEDA架构是理想的解决方案



PRIMETON · 普元

时代在变，我们的技术架构也在变，顺时针看架构的如下演进过程：

SEDA消息处理架构的演进过程

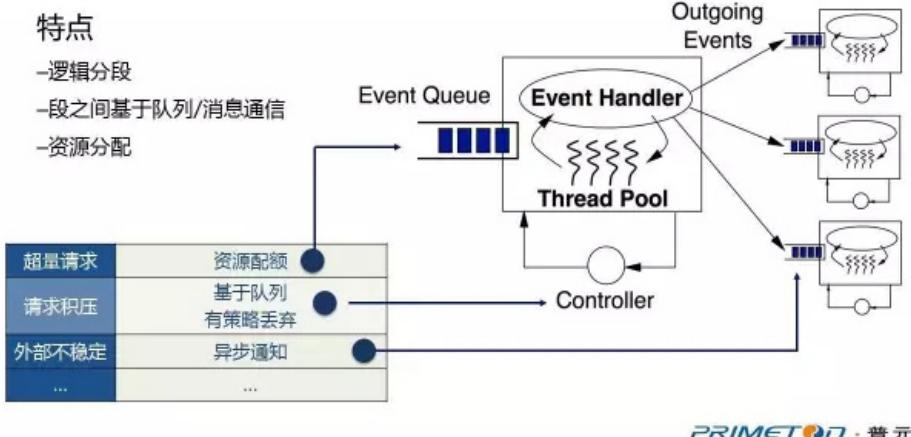


PRIMETON · 普元

应用逻辑封装到 Event Handler，接收到许多事件。处理这些事件，然后派发事件加入其他 Stage 的 queue。对 queue 和 threads 没有直接控制 Event queue 吸纳过量的负载，有限的线程池维持并发。

Stage 控制器：负责资源的分配和调度；控制派发给 Event Handler 的事件的数量和顺序；Event Handler 可能在内部丢弃、过滤、重排序事件。

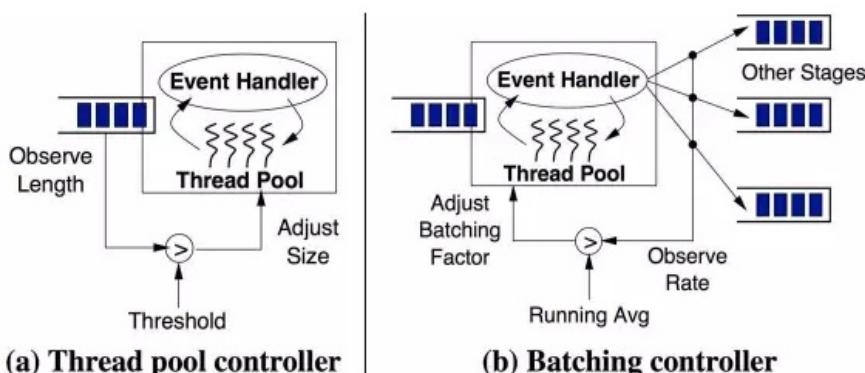
分阶段异步处理是可靠的构建基础



接下来，说一下这种架构里的动态资源控制器，一个是线程池管理，另一个是批量管理。

线程池管理器决定了每个微服务处理下的 Stage 合理的并发程度，通过观察队列长度，超过阈值就添加线程并移除空闲线程。

动态资源控制器：线程池管理

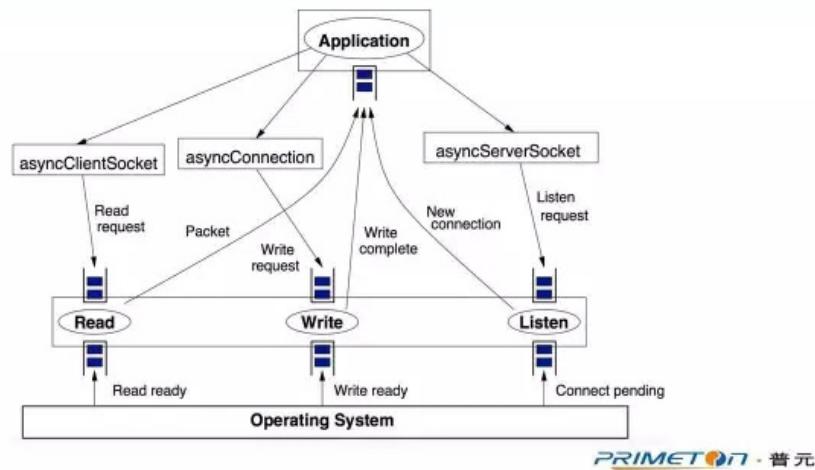


PRIMETON · 普元

SEDA 架构中 SEDA 并发模型依赖于 Event-driven concurrency 模型来支持高并发。利用一组线程来处理每个请求，而不是每一个请求一个线

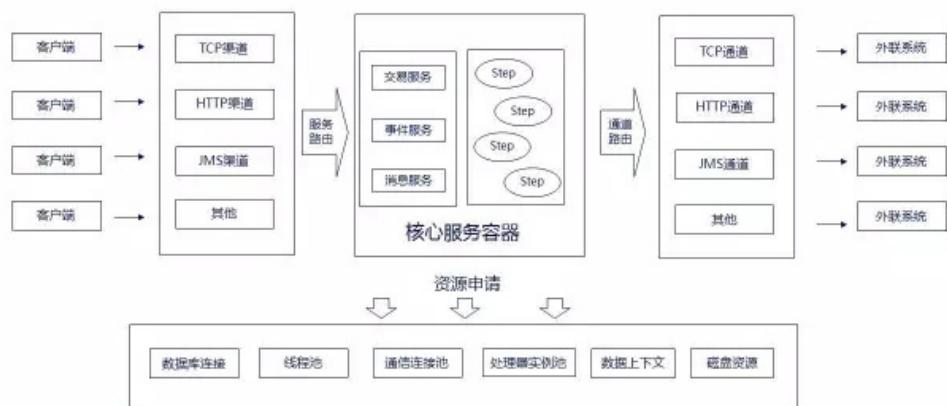
程，利用 Nonblocking I/O 来避免资源的阻塞。

全异步Socket处理架构



它的核心是，所有的逻辑处理以及接入、接出全部进行隔离，根据不同的业务操作类型分别对待合理进行分组。实际上，基于微服务架构的云平台在实现这一理念的时候有先天性的优势。

核心：逻辑处理、接出隔离，不同处理分别对待



SEDA 框架根据业务系统可靠性要求、消息框架可以采用内存队列或者持久化队列，满足不同 SLA 要求下的可靠性保障。

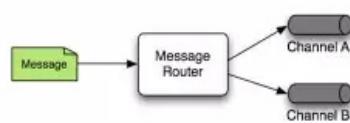
SEDA 架构下提供服务宿主处理的容器，容器是服务运行的基础环境，容器负责资源分配管理、服务超时管理、服务流量控制、服务路由控制。

消息框架

Channel使用Java内存队列
或者 外部持久化队列

多种类型的Channel

- DirectChannel
- PollableChannel
- SubscribableChannel
- PriorityChannel
- ExecutorChannel



消息编程不等于使用JMS

PRIMETON · 普元

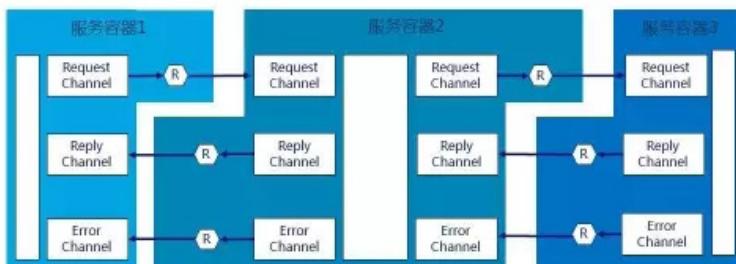
服务容器

容器具有三种形态

- ✓ 具备输入输出
- ✓ 只有输入
- ✓ 只有输出

容器提供基本能力

- ✓ 资源管理
- ✓ 超时管理
- ✓ 流量控制
- ✓ 路由控制



PRIMETON · 普元

每个容器有 2 组 channel（每组有请求、响正常应、错误响应）组成，根据是否具有输入、输出，容器分为三种类型。

接下来，我们来看看容器的路由。

作为独立的 stage，channel 根据不同业务负载提供路由，根据路由规则和服务元数据对服务进行路由。路由提供本地路由及远程路由能力，支持服务的横向扩展。

为了支撑远程路由，需要平台提供分布式的调度框架能力支撑，一个

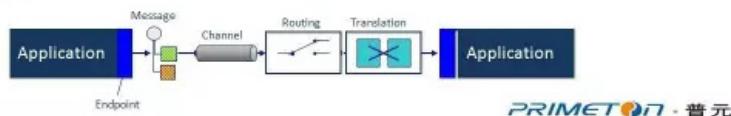
路由与分组

路由的定位

- 通过已经定义规则，将指定服务的消息发送到相应服务容器的 requestChannel

主要特征

- 支持远程路由与本地路由两种情况
- 默认支持根据服务码范围、服务类型两种路由规则
- 一个服务容器实例可以配置多个规则
- 对于不支持的规则，可以通过扩展方式支持



PRIMETON · 普元

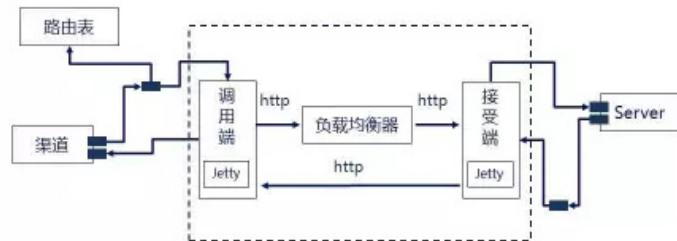
简化的分布式调度框架如下：

分布式调度框架

具体实现协议可替换，默认采用 Http 方式

提供同步、异步两种调用模式

发送和接受的是 Message，Message Header 保存远程调用的路由信息



PRIMETON · 普元

容器提供资源管理能力，对服务性资源、对象资源提供池化调度能力，精确匹配服务请求量。

在这种架构下，我们可以很方便的对云环境下的微服务（商业功能）实现多种控制、保障机制。

举个例子：分布式云环境下的流量控制机制在 SEDA 架构下的实现。

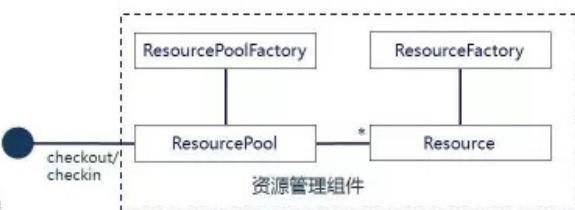
资源管理

两类资源

- 服务性资源
- 对象资源

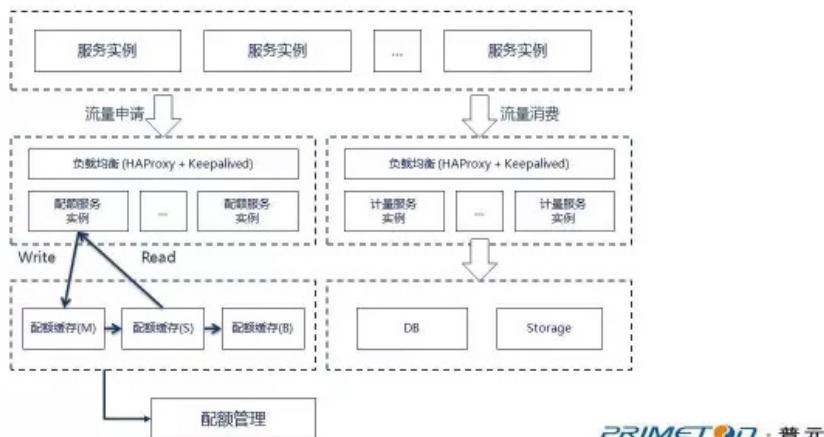
资源控制

- 在各自的定义中声明
- 在容器的定义中声明
- 使用时从资源管理器申请



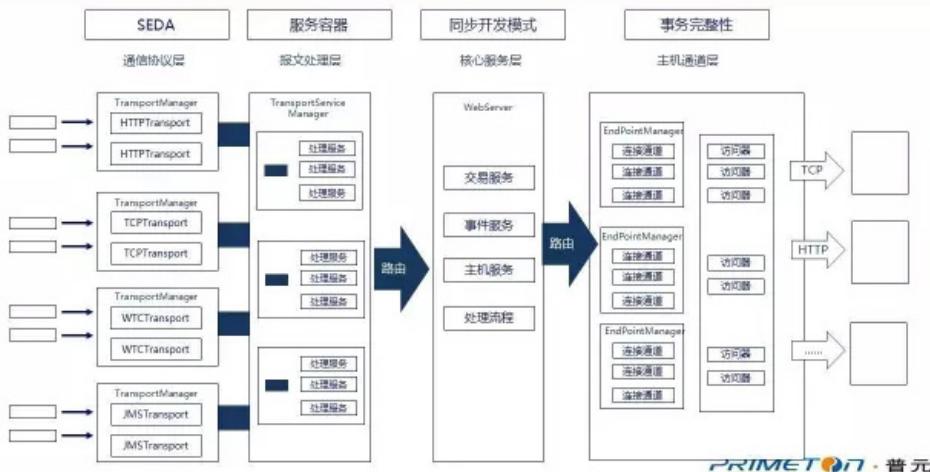
PRIMETON · 普元

分布式云环境下的流量控制机制



最后给大家奉上云平台下基础消息处理架构的 SEDA 架构总体设计。

总结



作者简介

臧一超, 现任普元大数据产品线副总, 基于微服务的企业架构实践者。十余年 IT 行业经验, 专注于 SOA、分布式计算、大数据处理、企业架构设计等领域。曾指导带领技术团队完成航天科工四院协同数据交换平台、上海移动 ESB 集成平台、华夏人寿服务治理平台等项目的系统实施以及方案撰写。

技术洞察

谈 DevOps 对于企业 IT 的价值

王延炯

其实从敏捷延展开的 DevOps 概念很早就已经被提出，不过由于配套的技术成熟度水平层次不齐，DevOps 的价值一直没有有效地发挥出来。现如今，随着容器技术的发展，DevOps 在企业中的实践难度大幅降低，其价值也得以体现。

1. DevOps 概念的发展历史

现在是 2016 年 6 月初，DevOps 作为一个 IT 圈的热词，几乎已经是无人不知，无人不晓。

如果你是一个有心人的话，可能会好奇 DevOps 这个概念到底是在什么情况下诞生的。

在 Wikipedia: DevOps^[1] 的词条里，可以发现 DevOps 相关概念已经早在 2008 年就被提出了。

At the Agile 2008 conference, Andrew Clay Shafer and Patrick Debois discussed “Agile Infrastructure”. The term “DevOps” was

popularized through a series of DevOps days starting in 2009 in Belgium. Since then, there have been DevOps days conferences held in many countries worldwide.

即便是在 2006 年 Amazon 发布了 ECS，微软在 2008 年和 2010 年提出和发布了 Azure，DevOps 的重要性似乎没那么强烈。

可是在 8 年前的那个时候，为什么 DevOps 没有迅速走红呢，第一个很重要的原因是由于那时候云计算（Wikipeida: Cloud computing^[2]）还是小众产品，更多地与虚拟化相关。第二个很重要的原因是 Docker 还没有横空出世，直到 2013 年 7 月。第三个很重要的原因是，Martin Fowler 在 2014 年 3 月提出了 Microservices^[3]。

可以看出，当前 DevOps 概念的深入人心，离不开云计算、容器 / Docker、微服务、敏捷等相关概念和实施的成熟发展。

2. 企业 IT 对于企业的价值

谈到企业 IT，就没有办法回避两种迥然不同的企业，一种是以传统制造业或者服务业为基础的，对生产资料进行加工的「传统企业」；另一种是以「信息互联」为基础的，对「人与人关系、人与物关系、物与物关系」进行信息加工的「互联网企业」。

这两类，是两类极端的企业，一类企业的日常运行，可以没有信息系统；另一类企业，完全离不开信息系统。

一般的信息系统，对于企业的价值，主要有三类渐进过度的典型类型。

第一类，是将信息系统定位于「辅助和支撑」企业的生产制造以及企业运营部门，因为这类企业的生产资料系、生产力、生产关，都以实体制造为主，不以信息加工和处理作为企业产品核心。

第二类，是将信息系统作为数据加工、传输作为主体，但业务模式来自于传统行业，信息系统主要完成已有业务规则的虚拟化，例如金融、电

信行业。这类企业的信息或者数据，主要来自于业务受理，或者说数据的生产者和使用者是企业自身。

第三类，是将信息系统作为企业唯一生产工具，并将企业的客户（个人或企业）所自发贡献的信息、数据，作为生产资料，形成新兴的业务模式。这里企业的典型，就是互联网企业。

随着新一轮「数字化」的概念席卷全球，非互联网企业所面临的更多针对用户和客户的思考和探索，都需要有更快交付能力的信息系统进行支撑，这也是传统企业互联网化，打开企业边界围栏迈出的第一步。

3. DevOps 对于企业 IT 的价值

通过前文的分析，可以看到，企业 IT 对于三类不一样的企业价值体现各有不同。

对于互联网企业，信息系统是企业产品的命脉，企业对于软件的价值观以及投资组合，对于其他两类企业要高出很多。

DevOps 的核心价值，是能够帮助企业快速交付变更，以便于快速响应企业对于市场的变化、用户的需求。

- 代码
- 构建
- 测试
- 打包
- 发布
- 配置
- 监控

以上 7 个过程，是 DevOps 站在软件生命周期平台化运营的视角，为企业 IT 所建立的一个「IT for IT / IT4IT^[4]」的业务平台。

如果说，软件开发、交付、运维是一个传统行业，那么 DevOps 就是

映射了这个传统行业的一个软件平台。

通过 DevOps 可以助力企业软件交付的效率提升，帮助企业 IT 实现数字化运营。

可以用一句话定位以下三者在价值链上的关系：

- 「信息」（数据）
- 「信息系统」（处理数据的工具）
- 「DevOps」（制造与维护处理数据的工具的工具）

在「信息」（数据）成为企业「生产资料」一部分的条件下，信息系统快速演进的业务驱动力、DevOps 的价值，才能够清晰地得以体现。

企业 IT 系统的从「业务支撑型」走向「业务驱动型」的转折点，是由企业产品的最终用户，直接以低成本为企业提供生产资料——信息（数据）——并使之成为企业产品的一部分。

如果信息是企业产品的全部，那么这种类型的企业就是第三类——互联网企业。

4. DevOps 在企业中的实践

和众多源于互联网的理念一样，在 DevOps 的概念被炒热之前，众多互联网公司其实已经实践了 DevOps。其中的原因也正是因为信息系统，是这些公司的生产工具，没有人比互联网公司的人更明白提高自身的办公效率，提高团队、企业的生产力，就是为提高企业产品的生产力进行有效的保障。

除了前文提的 DevOps 覆盖企业软件生命周期中的 7 个过程，DevOps 在企业落地过程中，传统企业或多或少都已经建设了一部分配套系统，尤其是 AAAA 与监控系统，其他的代码管理、持续集成工具也或多或少的有所积累。

DevOps 更多的是把 IT 服务产品化的平台，企业中的任何一个 IT

能力，都应当能够在 DevOps 中面向其用户提供自服务的能力。

例如，系统的开发者，能够自主地在 DevOps 上联机分析生产环境的日志，而不必经过层层壁垒，一台台登录生产环境的主机，检索日志。再例如，项目组的快速变更、员工的快速入职、离职，都应当能够在 DevOps 中实现自动化的账号开通和注销。

需要注意的是，相比传统企业尤其是制造业的产品制造工艺和制造流程，软件产品的制造，IT 服务的交付，更多的是交付一些无形的软件产品和知识工作。正因为这些无形产品受制于不同的人认知所产生的多变，其管理复杂度远比制造业来的复杂，企业软件的设计、开发、发布、上线，缺乏标准化的管理过程。

对于如今的非互联网企业而言，能够快速见效的 DevOps 实践，应当从（环境）配置的管理，以及自动化部署。在实施难度上，配置的管理要低于自动化部署。因为非互联网企业的技术路线由于供应商的竞争（甚至是恶意竞争），变得极其多样，架构离散化程度也很高。

对比互联网企业，（环境）配置管理和自动化部署，由于 IT 技术从硬件到虚拟化 / 容器的自主可控，企业整体技术架构的收敛性就比较理想。

5. 当前 DevOps 对于企业架构支撑的不足

从 代码 到 监控，可以看出 DevOps 对于企业尚有以下不足（或者说 DevOps 本来就不考虑这些）：

1. 缺少对企业 IT 战略规划、企业 IT 业务架构规划、企业 IT 系统架构设计、企业 IT 系统需求管理以及 IT 项目管理进行有效的平台化、数字化支撑。
2. 缺少对企业 IT 部门、信息系统运营效能的系统性评估和优化模型。
3. 缺少对企业不同 IT 系统供应商的技术架构、IT 产品的标准化、基线化管理。

4. 缺少对企业数据类应用的支撑，更偏于交易型应用的交付。

对于第 1, 2 两点，任何企业都有这方面的需求，但由于企业在投资组合上对这个领域投资较少，投资规模较大、实施难度较大，企业主更愿意进行面向企业产品「开源」投资。

对于第 3 点，几乎不会在秉承「自主建设」互联网公司存在，但在其他企业中也逐步会通过精益运营的方式进行落地实施。

6. 参考资料

1. Wikipedia:DevOps: <https://en.wikipedia.org/wiki/DevOps>
2. Wikipedia: Cloud Computing: https://en.wikipedia.org/wiki/Cloud_computing
3. Microservices: <http://martinfowler.com/articles/microservices.html>
4. IT4IT: <http://www.opengroup.org/IT4IT>

作者简介

王延炯，现任普元信息主任架构师。密码学博士，毕业于北京邮电大学。曾先后任职于西门子（中国）研究院、垂直行业互联网公司。带领团队交付了移动、金融、电信等多个行业、众多 IT 系统的咨询、设计、研发、实施、维护、优化工作。对分布式架构，企业架构，以及企业 IT 平台化运营有深入的研究和理解。

谈元数据驱动的微服务架构

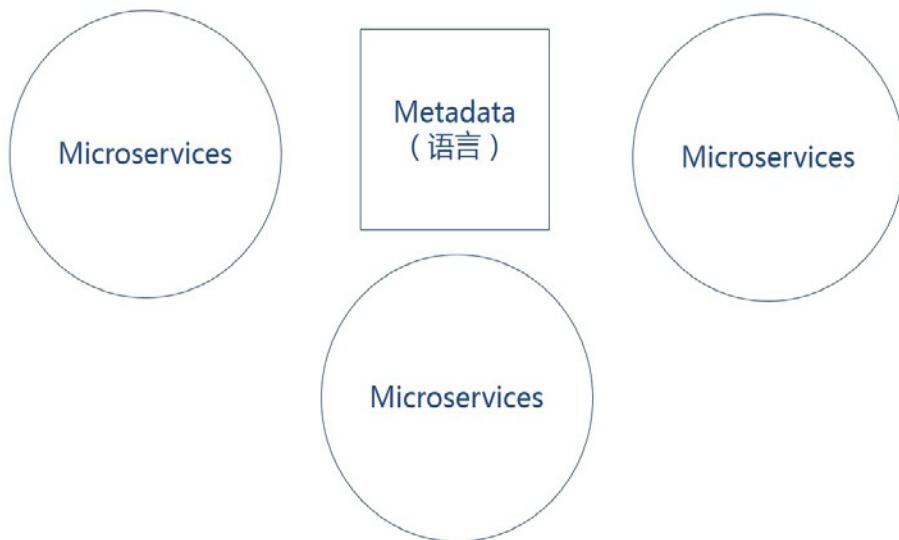
王轩

企业 IT 架构已经发展了多个阶段，一方面是服务化架构的发展，在 SOA 阶段主要解决应用间集成问题，但随着企业业务的发展，单个应用逐渐成为“巨石型”应用，难以扩展也难以维护。微服务架构应运而生，微服务架构专注于单个应用的内部，将“巨石”应用拆分成为多个微服务，以微服务为单独单元开发运营。

另一方面是模型化架构式 MDA（模型驱动架构）的发展，模型驱动工程也在不断发展，从全面的完全模型自动化，到 DSM（特定领域建模）针对特定领域的建模，再发展到 DDD（领域模型驱动设计），模型的作用变得更加特定化和轻量化。

服务化架构和模型化架构其实是统一的。在微服务架构中微服务的粒度小，数量多，微服务的设计与微服务之间的连接需要一套规范，同时需要一套可以对话的统一“语言”。传统的模型方式的核心目标是能够自动生成代码，故定义过于复杂。而微服务间的“语言”的目标与传统不同，

用元数据作为“语言”驱动整个微服务架构是不错的选择。

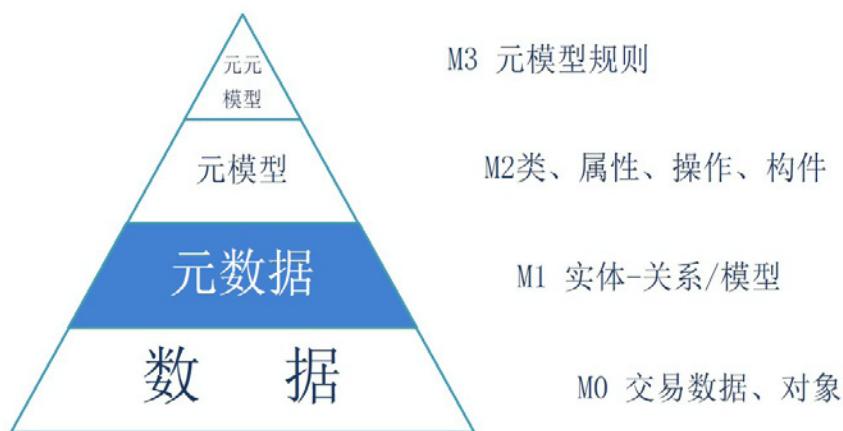


我们看看元数据表示了什么内容，元数据就是计算机的认知维度，可以说，掌握了元数据就掌握了信息的维度，只有充分利用好元数据（也就是信息的维度），通过合理的元数据建模（维度整合），对元数据进行科学管理（维度完善），才能让计算机更好地认知企业系统。元数据管理的核心内容是，信息的概念和信息之间的连接。概念表示对某个业务所有维度的集合，连接是对业务维度之间关系的描述。通过这样的描述，能够使元数据成为微服务直接对话的“语言”，还能够通过“语言”规范服务体系的设计。

从模型到微服务——元数据与微服务的关系

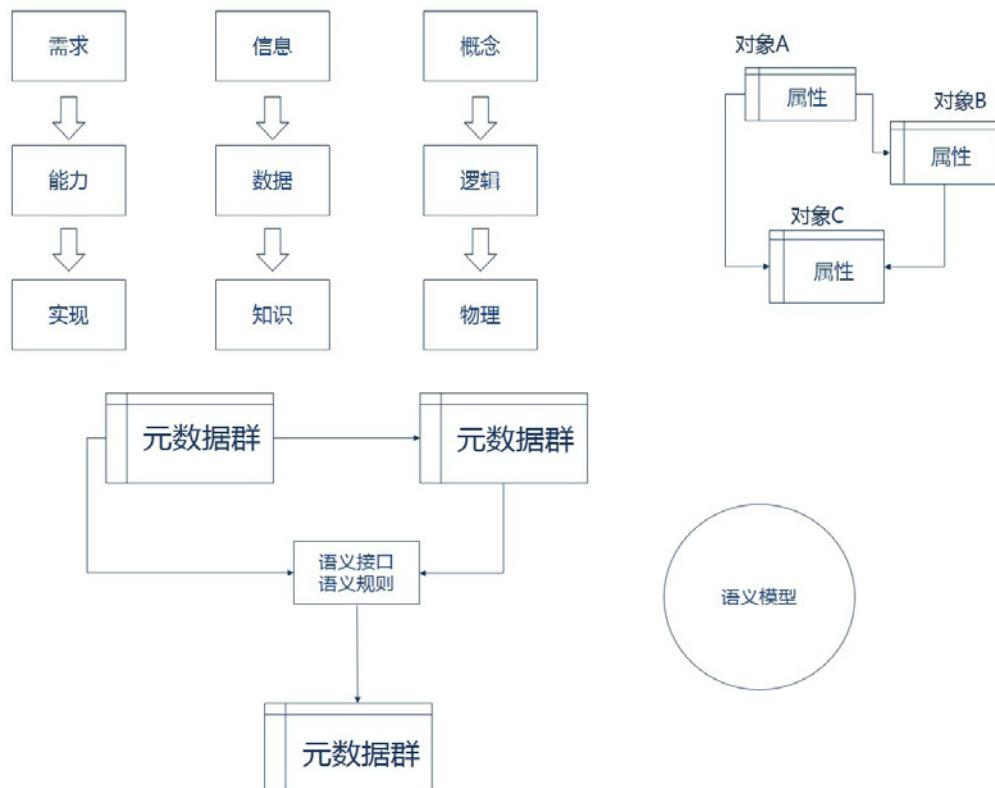
定位模型与元数据的概念之前，我们不得不提到 **MOF (元对象设施)** 或者 **元对象机制 MetaObject Facility**，它是 OMG (国际标准化组织) 元模型和元数据的存储标准，提供在异构环境下对元数据知识库的访问接口。我们可以看到每个层次的上一层是下一层的模型，本层次的描述语言在它的上一层模型中。**M1 层**元数据，也就是通常说的“**数据模型层**”。M1 层(元数据)对应在日常我们项目开发过程中进行的 ER 模型图模型设计。

也就是说我们进行所有的数据层设计其实都是元数据的建模过程。

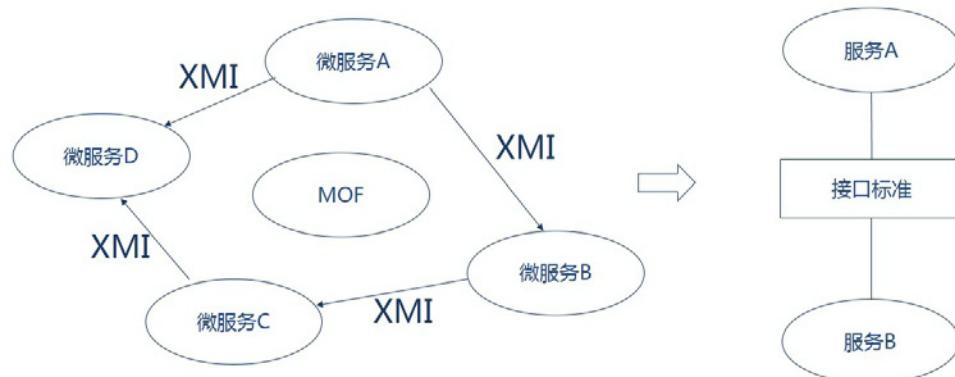


再看如何进行模型设计，也就是进行元数据的设计，数据的描述。我们一般建模过程会将其分解成更小的、更简单的元素，通过多个模型之间的关系描述复杂的事物逻辑。一般从需求开始，无论是用户的需求还是技术需求，能力是实现需求的桥梁与纽带，借助现有的技术手段进行实现的支撑。随着建模过程逐渐深入，建模可以分为概念模型 – 逻辑模型 – 物理模型，层层递进最终物理模型会确定数据库实现方式，将数据表固化到数据库中。建模最有效的简化方式是图形建模，也就是我们通常所说的 ER 图建模。多数建模方法都建立在可视化语言的基础上。比如 UML 实体 – 关系图建模，这就是最常见的语义模型建模方法。基于语义分析模型的元数据模型，主要是建立模型的实体与实体之间的关系，包括元数据模型之间关系的建立，元数据之间的输入输出接口等。

不同数据模型之间统一标准相互访问的机制是其中的关键点，可以使用 XMI 规范来做模型互相访问，这里 XMI 是 OMG 在元数据交换方面的最重要标准之一，同时也是 W3C 认可的标准。允许 MOF 元数据（即遵从 MOF 或基于 MOF 的元模型的元数据）以流或文件的形式按照 XML 的标准格式进行交换。



在微服务中每个服务都有自己的数据库。这种思路与企业级的传统数据建模过程不同，每个微服务中需要建立自己的数据模型。各微服务的接口 API 需要定义元数据，接口需要清晰的元数据模型，对象、属性。也就是我们需要元数据的原因，**我们需要建一套完整的元数据模型机制，这也就是元数据与微服务之间的关系所在。**



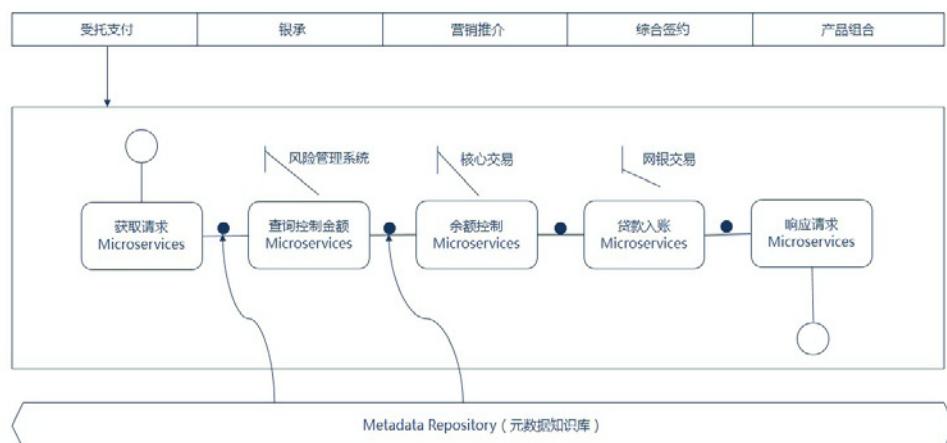
微服务架构需要元数据，即微服务与元数据的关系，那么微服务中的元数据中具体如何应用，有哪些应用场景？我们接下来看一下——微服务中元数据的价值，了解元数据在微服务中具体的应用价值点。

微服务中元数据的四大价值

元数据驱动的微服务价值体现为：一、提供微服务边界交互模型；二、规范微服务开发和使用；三、分析微服务的脉络；四、管理微服务的全生命周期。

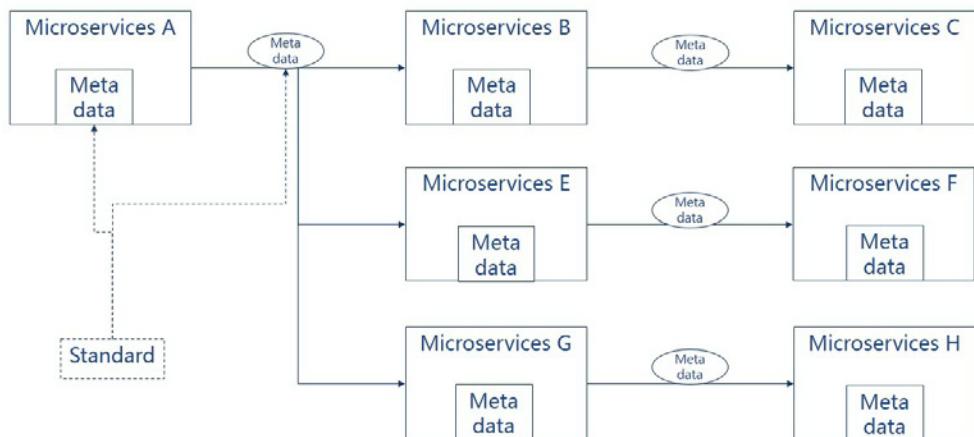
价值一，提供微服务边界交互模型

每个微服务代表一个业务可交付的应用，不同微服务在进行数据互通的时候需要一个业务实体作为信息承载。而这个信息载体通常称为微服务的交互信息，这些信息是通过元数据进行的，它代表在微服务间游走的数据载体信息，不同功能的微服务所使用的载体根据业务不同是不太一样的，在微服务架构下最好进行事前定义。边界，也就是元数据，这个元数据就像网络传输的数据包的概念，有它的协议和格式。**微服务下的元数据主要解决：模型复杂性和模型变更、涉众沟通、设计沟通、提炼并捕获领域知识等相关问题。**



上图服务数流程是企业申请贷款审核流程中便指定贷款款项相应划款

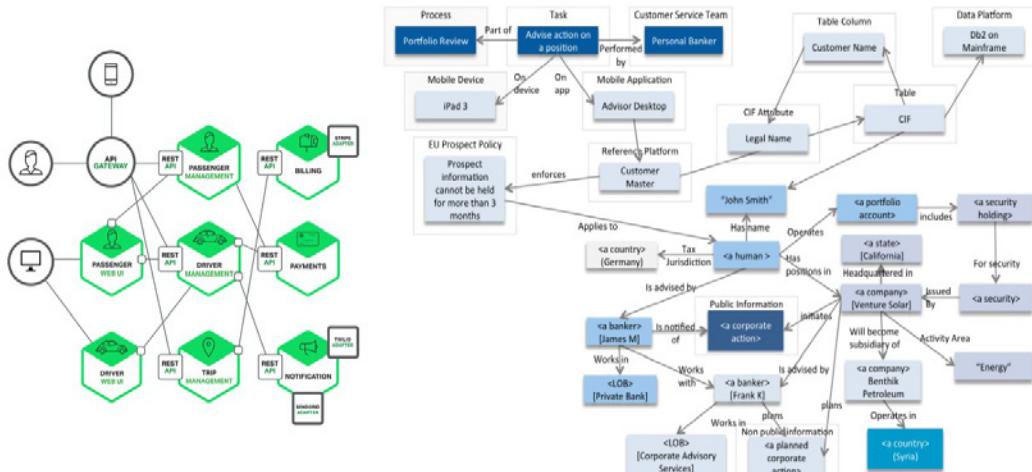
对象（收款人），贷款发放后，银行按照合同约定将贷款款项直接划转至指定收款人的操作。本场景中涉及到 5 个服务的串联整合，服务间的连接就是通过元数据驱动完成的。微服务的设计和实现需要遵照组织级的元数据知识库进行开发实例化，组织也有职责对微服务的合规性进行审查，而这一切是通过元数据来驱动的。



微服务包含两种元数据：一、**微服务本体的元数据驱动规范模型**；二、**微服务间数据载体的元数据驱动规范**。微服务本体元数据信息一般包含：业务信息，功能信息，数据信息，管理信息，逻辑信息。这些信息是基于元数据知识库中定义的元数据规范实例化得到的。微服务间的元数据信息包含，依赖信息，参数信息，过程信息，同样，这些信息也是来源于元数据知识库，用来解决微服务间连接的关系、映射、依赖、整合约束。

价值二，元数据标准规范微服务开发和使用

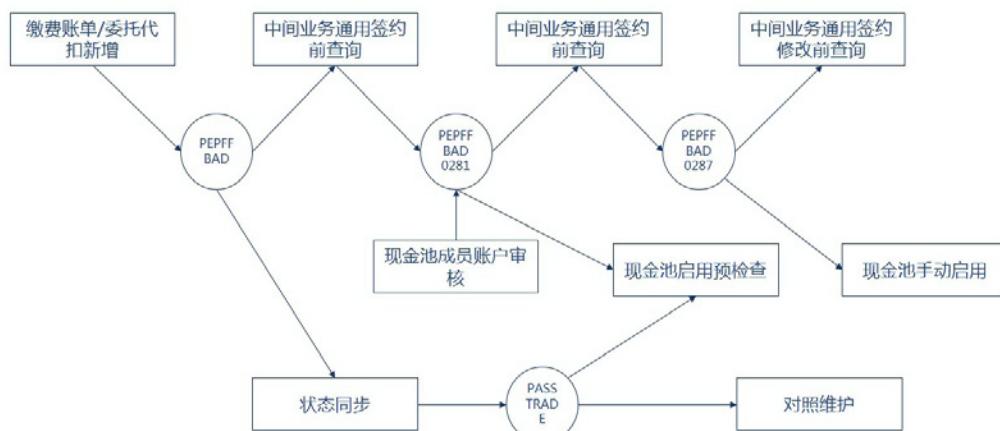
上面两幅图所呈现的是实际使用服务时的服务整合情景，不同微服务在不同的业务场景下被依赖和引用的程度不同，每个微服务提供的能力数据由业务复杂程度决定。成千上万个微服务在运营环境下高效地运转，这就要求微服务必须有标准规范。**标准规范分为两个方面**，一方面是微服务的数据标准，另一方面是微服务的服务标准。而这两种标准都需要元数据定义。



价值三，分析微服务脉络

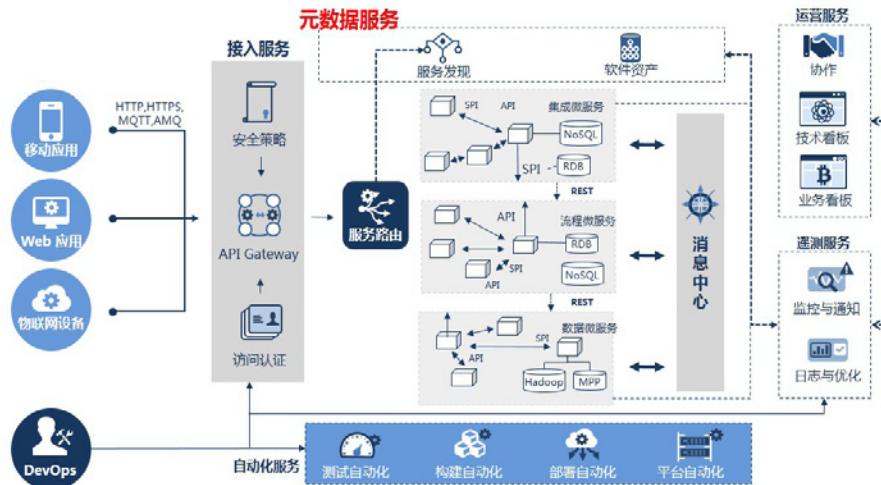
微服务架构模式应用的改变将会波及多个服务。微服务架构模式就要考虑相关改变对不同服务的影响。比如，你需要更新服务 C，然后是 B，最后才是 A，下图是一个场景示例。一般微服务的改变会影响多个服务联动调整，不同区域、不同团队在进行微服务的开发势必会带来协调上的不便。基于元数据驱动的微服务实现会带来天生的脉络分析优势，从**任何一个微服务可以分析出整个调用关系图谱，我们称之为“微服务地图”**。

价值四，微服务的全生命周期管理

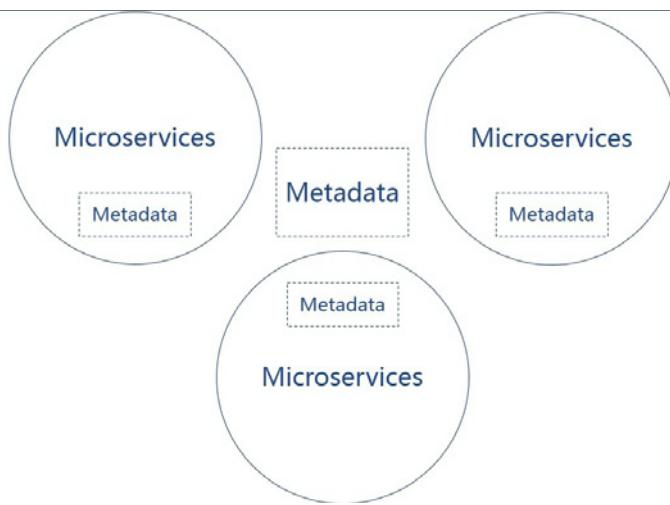


微服务的生命周期有多个阶段，在前期需要与多个微服务协同考虑，

上架后也会有多个使用者，在这种复杂的状况下需要管理微服务的全生命周期。



在**规划阶段**，提供标准元数据规范微服务。在**设计阶段**，提供连接其他微服务的元数据信息。在开发阶段，使用元数据协助开发测试。在**运营阶段**，上线后分析微服务的使用情况，并协助维护微服务的变更。最后微服务下架时将微服务的元数据存档，并确保对目前体系不产生影响。



最后，**未来元数据将是微服务的中枢神经**。元数据驱动的微服务架构还需要进一步思考和研究。

作者简介

王轩，普元信息软件产品部副总、大数据产品线总经理，2010 年加

入普元，全面主持普元大数据产品的研发、拓展及团队管理工作。十年大型企业信息化架构设计与建设经验，曾任中国人民银行核心平台架构师。主持参与了国家开发银行大数据项目、中国人民银行软件开发平台、国家电网云计算平台等大型项目建设。王轩对大数据行业有着深入的研究和洞察，并对企业信息化平台建设，企业云计算及大数据平台建设有着丰富经验。

技术实践

基于微服务架构的技术实践

顾伟

大家好，今天分享的是“基于微服务架构的技术实践”，标题有点土，希望内容对大家有用。这个是我上周在 CSDN 北京沙龙上分享的内容的改版，加入了一些设计部分，才见了一些理念部分和自身平台的具体内容，篇幅稍长，我们尽快进入主题。

内容分了三节，重点是第二部分，包括架构选型参考、6 个关键模块设计、以及通用的概念模型、部署模型图等。

对微服务的认识

先进入第一部分：包括两块内容，说说历史，说说问题。

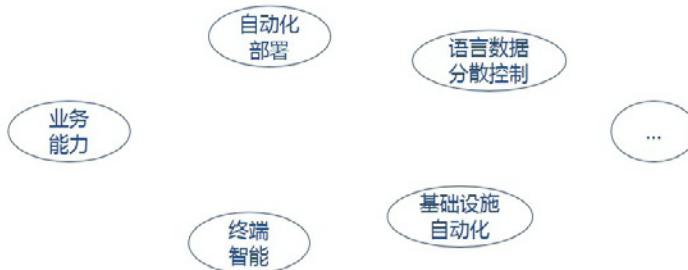
微服务的说法由来已久，但真真风靡起来是 2012 年 martin flower 提出的，但为什么现在大家一谈微服务就容易吵吵呢，因为 martin flower 给了给多参考原则，但并没有给出最佳实践，是的现在大家一讨论就会说应该按什么什么拆分，应该怎么考虑交互与安全，等等。

微服务架构的出现



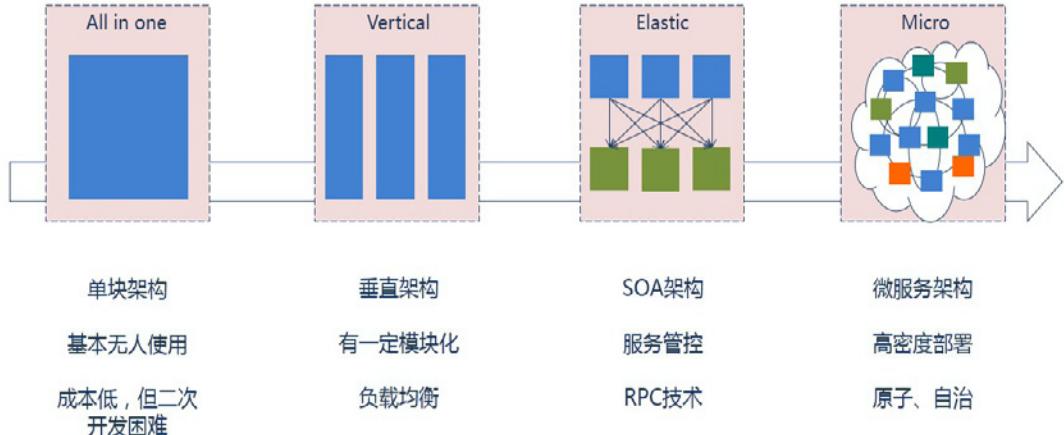
<http://martinfowler.com/articles/microservices.html>

2012



妨我们先从企业应用架构来看看，微服务与原来有什么区别：

微服务架构的演进史



第一个就不说了，第二个垂直架构，典型的比如 SSH 框架，帮大家考虑了模块化、MVC 等，但并没有考虑服务化。第三个是分布式架构，以 SOA 为代表的这类技术已经热了很多年，现在也是企业架构的主体支撑部分。而第四个以微服务架构为支撑的技术虽然在一些先进企业和互联网公司已经运用，但从生态上来看，还有很长一段时间要走。

所以会有下面的这些争吵：

对微服务的常见认知误区



曾经被人问的最多的就是，微服务应该按什么拆分，以前会和别人说 DDD，说康威，现在更多是不想回答了，因为怎么说都有理，怎么说也都有实际难以解决的问题。

再比如，微服务让开发变得更简单，是完全无依据的。这个往往建立在你已经把微服务通信框架、管控框架、部署框架、分布式事务、开发规范等一系列都完全制定好后，再结合技术与业务分离等实践后才能给出答案，切勿认为微服务可解决开发问题。

微服务架构实践

回到今天的主体第二部分：包括了技术和架构参考，平台主要架构设计，以及关键的模块设计。

参考 1：平台品质属性，这个图大家应该不陌生，是在说一个平台或者模块的设计，很难把各个维度面面俱到，往往是有取舍的支持，图中“+”代表可同时享用，而“-”代表会有冲突，著名的 CAP 原理其实也是这个意思。

参考 2：扩展立方体，这个是在说平台提升扩展性的几个维度，X 轴

参考 -> 平台的品质属性

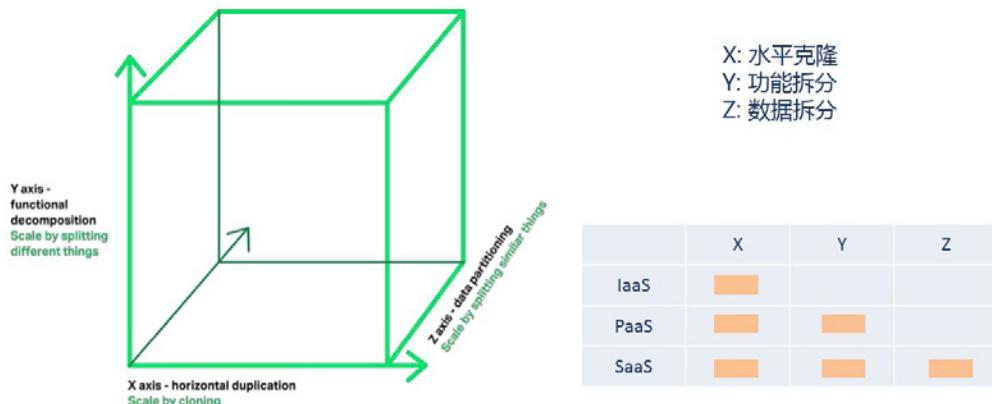
用户属性：	有效性	高效性	灵活性	完整性	互操作性	可维护性	移植性	可靠性	可重用性	健壮性	可测试性	可用性
有效性	+							+		+		
高效性		+	-	-	-	-	-	-	-	-	-	-
灵活性	-		+	-		+	+	+			+	
完整性	-			+	-				-		-	-
互操作性	-	+	-		+							
可维护性	+	-	+			+		+			+	
移植性	-	+	-	+	-	+	+		+	+	-	
可靠性	+	-	+			+		+		+	+	+
可重用性	-	+	-	+	+	+	+	-	+	+	+	
健壮性	+	-						+		+		+
可测试性	+	-	+			+		+			+	+
可用性		-							+	-	+	

用户属性：	X: 水平克隆	Y: 功能拆分	Z: 数据拆分
有效性	+		
高效性		+	
灵活性	-	+	
完整性	-		+
互操作性	-	+	-
可维护性	+	-	+
移植性	-	+	+
可靠性	+	-	+
可重用性	-	+	+
健壮性	+	-	+
可测试性	+	-	+
可用性		-	+

是通过克隆的方式，Y 轴是通过业务功能拆分的方式，Z 轴是数据拆分的方式。这个在系统设计中是一个进阶参考，但在设计之初尽量要考虑清楚一定时间段里，采用哪种模式作为默认方式。

参考 3: Heroku 的 12Factor，这个以前提过，这些因素为系统的

参考 -> 扩展性立方体



CloudNative 目标考虑，让云上云下得到同样的体验。

参考 4: 谷歌的优秀框架 Borg，谷歌内部的管理调度核心，支撑着谷歌上万台机器及业务的运行，这个虽然是不开源的，但其设计思路和架构是很容易被找到和学习的。参考这个的原因是谷歌本身内部就是容器与微

服务架构的生产运用，是一个真正大规模的实践参考。

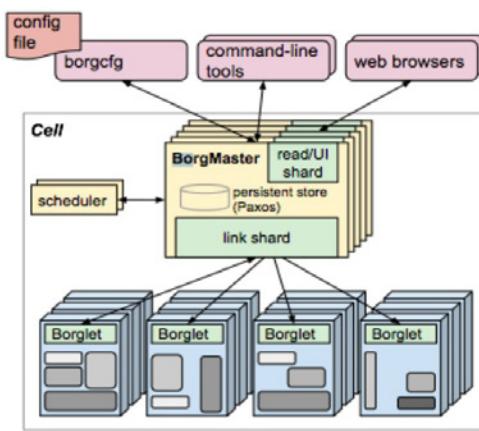
基于上述的参考等，我们验证过上述的技术栈，用于支撑微服务架构，

参考 -> 12Factor

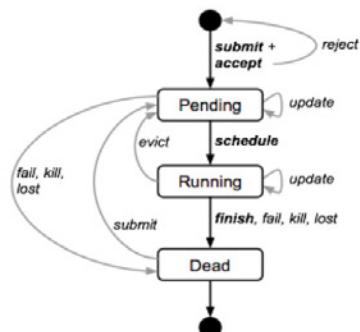
I. 基准代码
一份基准代码，多份部署
II. 依赖
显式声明依赖关系
III. 配置
在环境中存储配置
IV. 后端服务
把后端服务当作附加资源
V. 构建、发布、运行
严格分离构建和运行
VI. 进程
以一个或多个无状态进程运行应用
VII. 端口绑定
通过端口绑定提供服务
VIII. 并发
通过进程模型进行扩展
IX. 易处理
快速启动和优雅终止可最大化健壮性
X. 开发环境与线上环境等价
尽可能的保持开发、预发布、线上环境相同
XI. 日志
把日志当作事件流



参考 -> Borg



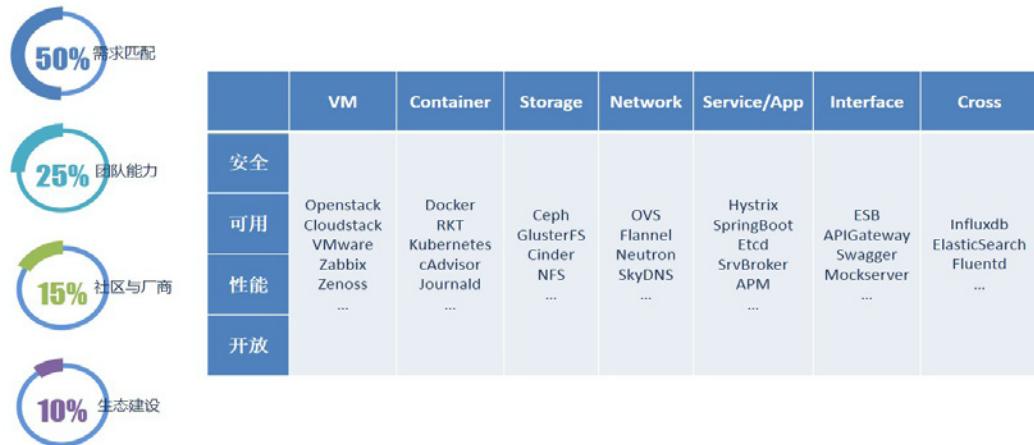
Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.



当然这里面已经有不少被我们放弃了，上图是从对象类型（横向）和功能要求（纵向）给技术做了一些罗列，也是我们平台使用的开源图谱，大家有兴趣可基于某几个来探讨。

紧接着我们开始尝试微服务支撑平台研发，因为前面的误区提到了，微服务其实对支撑（开发、部署、运维等）平台的要求比传统更高了，所以这个正是我们的市场定位之一。

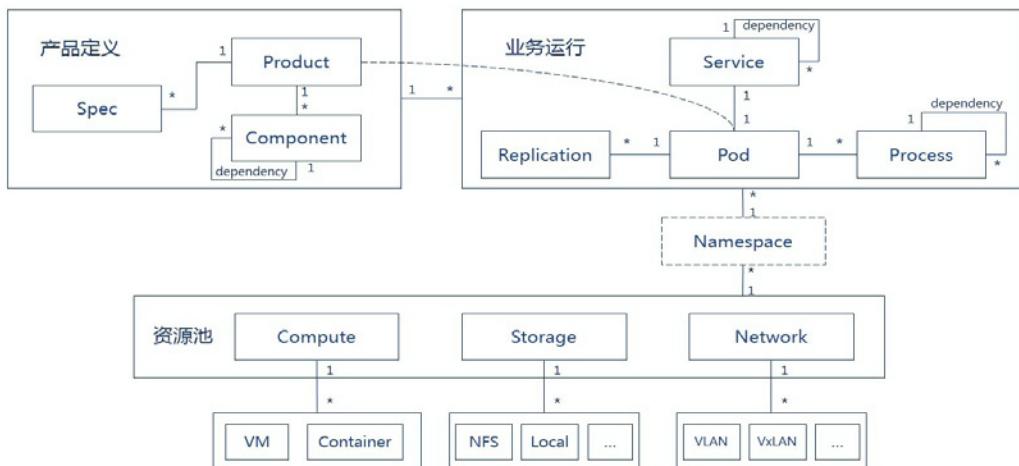
经过验证的可参考技术栈



在做的过程中，我简单总结：有 3 个思考和 6 个关键设计，这里其实篇幅考虑，省略了很多，比如自动化测试、比如一体化监控、比如安全控制等，后面应该会有同事往一些聚焦的话题分享。

思考 1：我们要一个兼容企业情况的模型，比如说：

过程思考(1) -> 概念模型建立



- 有些企业会考虑生产上VM，开发测试上容器；
- 有些企业需要开发测试预发上线四套环境，有些只要两套；
- 有些企业环境间要求完全隔离，有些只要逻辑隔离；

- 有些企业要求对下层资源池，有些则完全没有资源池的概念。

那概念模型上我们怎么办？上图是一个片段，核心是业务运行及 namespace，我们认为下层无论资源有没有池化，都需要加一层 Namespace 的管理，这个管理有很多目标，比如隔离，再比如池化。

然后紧接着是 pod，这个概念参考了 kubernetes，在微服务运行时，一直强调一个业务的独立性，比如一个业务，其应用及数据库是绑定的，那我们认为这种就是一个 pod（豌豆荚），体现的是一个独立业务（服务）。

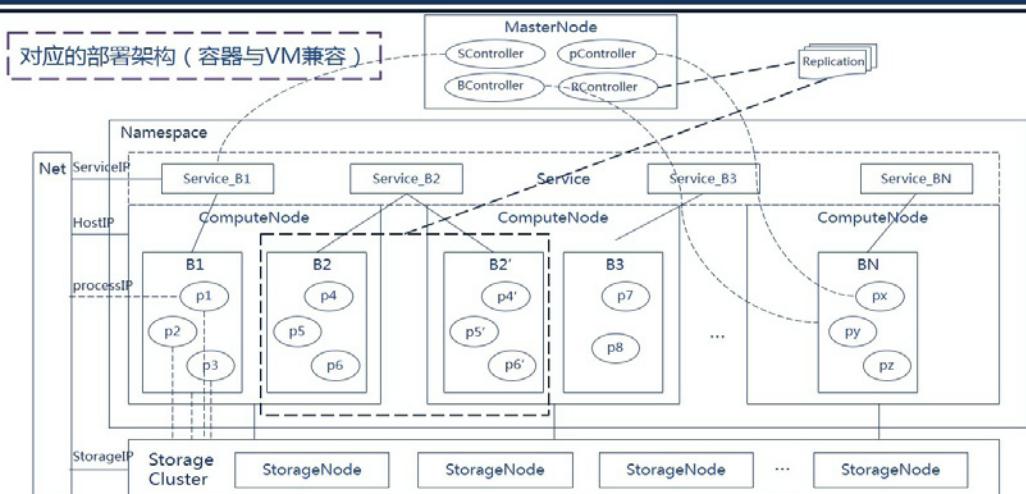
再从 pod 往下（一个 pod 无论内部如何，一般是跑在一台宿主机的，业务内部尽量本地调用），pod 可以包括多个进程，也可以包含多个容器，也就是上图的 pod 与 process 的关系。

再接着 pod 要求是集群的，所以一个 pod 可以有多个副本，也就是上图的 pod 与 replication 的关系。

最终业务要能对外提供能力，类似一个集群对外的统一发布地址，也就是上图的 service 的概念，service 是外部的访问入口，并拥有一定的敷在分发能力。

上图其实是一个详细的对于概念模型实际运行的部署架构图，这里我就不展开讲了，有兴趣可以结合上一张图一起看。

过程思考(1) -> 概念模型建立



思考 2：设计原则现在人人都会提，但微服务架构里，我觉得最重要的几个原则是这些：

过程思考(2) -> 设计原则

隔离失败

宽进严出

PDCA (重在反馈)

MVP

隔离失败：微服务架构下，相互间的通信越来越多，不像单块架构那样，要坏一起坏，如何不受别人影响，以及自己不破坏别人，是要考虑的重点。

宽进严出：男总老拿 Unix 的设计来批我，这个其实也是其中一项重要原则，运用到微服务上，每个微服务要足够宽容，让更多人，更多方法可以接入，但返回的结果一定要严谨，要对别人负责。

PDCA：微服务下要以服务为级别快速迭代，迭代的依据是什么，要能够收集的来自上下游的足够反馈。

MVP：毕竟前面提到了，微服务架构的最佳实践太少，不要试图一上来就做大而全的东西，一步一步走。

思考 3：数据是最有效的依据，现在支撑微服务框架的开源技术很多，太多，用一个东西要有对比，要对比就要有测试数据，不是只有资料。

接着我们谈谈关键的设计（这里列了 6 个）

关键 1：屏蔽异构环境，尤其像我们这种公有云和私有云同时发展的，要考虑各类环境（VM、容器、KVM、XEN、EXSI、NAS、SAN、CEPH、

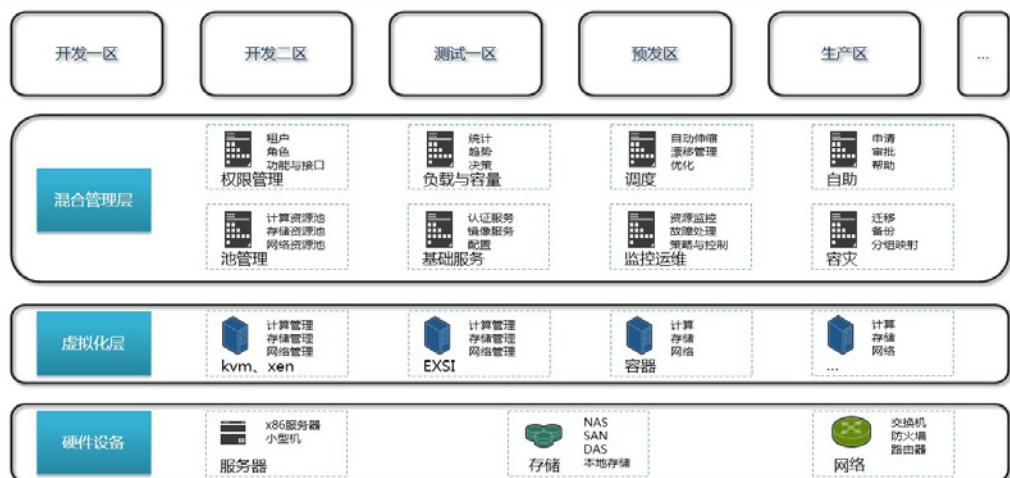
过程思考(3) -> 数据说话



GlusterFS、SDN），这个也没什么特别的好办法，就是看经验，能力积累，架构一致。

关键 2：微服务的隔离与互通，拿跑在容器里举例，互通要求的是“可达、快速可达、安全并快速可达”。一个微服务内部可采用本地方式，微

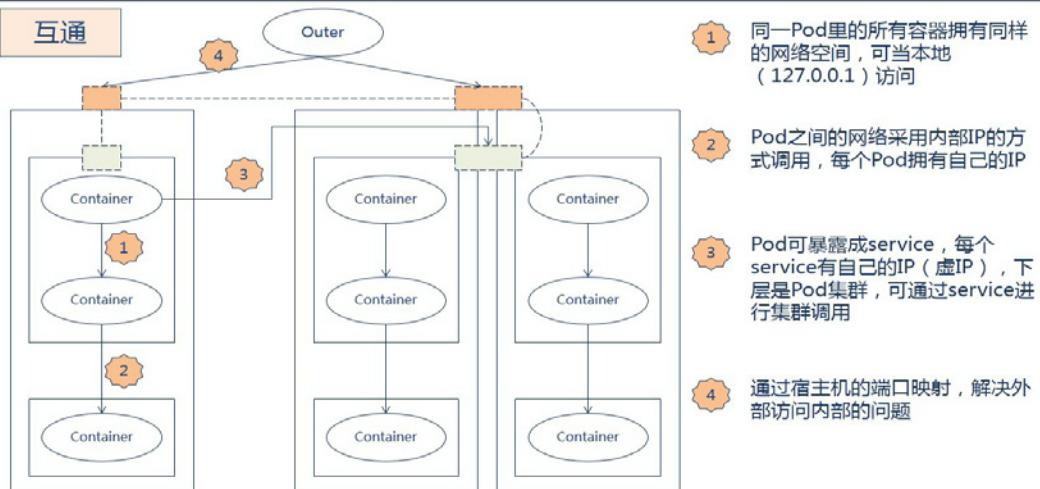
关键能力(1) -> 异构环境管理



服务之间采用 service 地址（无论内部怎么漂移伸缩，对外地址不变），当然外部分为大集群的微服务，以及以太网的客户端，对于公网上的，采用宿主机端口映射出去是个可选方式，当然要结合底层硬件基础设施，比如阿里还有 EIP 之类。

同样是关键 2 里的：隔离，隔离有逻辑有物理，大家可结合实际需要来选择，从宿主机隔离，到 Namespace 隔离，到建立逻辑子网，以及内部

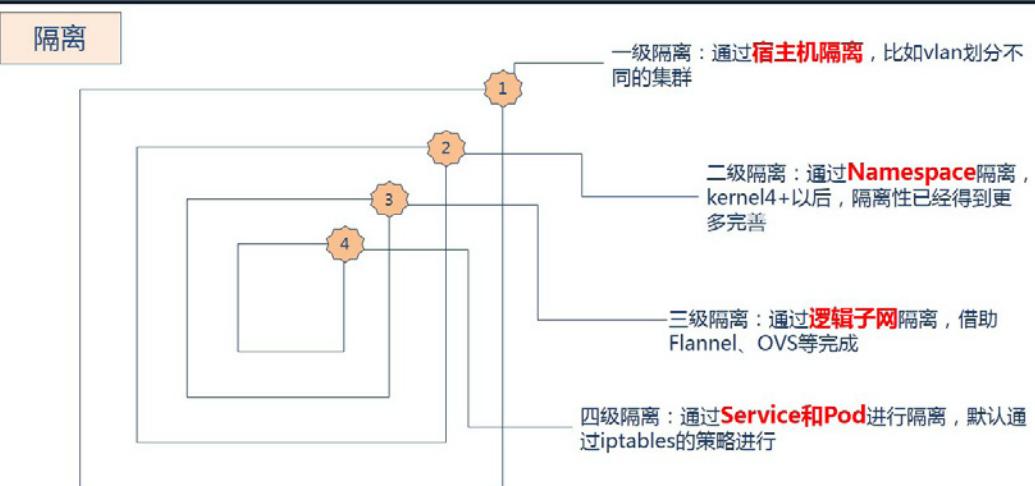
关键能力(2) -> 容器资源的隔离与互通



的 iptables 策略配置等。

关键 3：微服务伸缩与漂移，上图里万年不变的公式。以漂移为例子，漂移有很多触发可能，有因为故障的，有基于优化考虑的，像优化这种，

关键能力(2) -> 容器资源的隔离与互通



就要求定义很多维度，结合对微服务的监控打分加权。

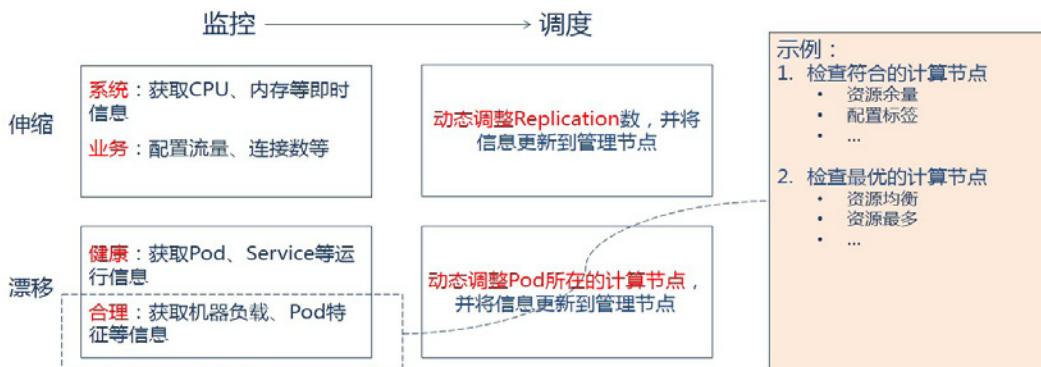
同样是关键 3，要求的两个基础能力，第一是对于微服务点线面结合的监控，第二是监控数据的存储分析（微服务散落各地，要汇总合并，要

能串接分析)

关键 4：微服务的升级与回退，既然要求快速迭代，那平台设计上需要考虑：

关键能力(3) -> 伸缩与漂移

*原则 : Result = func1*weight1 + func2*weight2 + ... + funcn*weightn*



关键能力(3) -> 伸缩与漂移

关键要求：快、准、灵



发布要原子化，可编排。

标签设计，让每个动作、每个状态、每个资源都可以标识。

状态设计，部署是原子化的操作，而内部的状态设计同样重要，可结合状态做挂起、唤醒等诸多操作。

版本规范，这个无论是微服务还是传统的都很重要。

路由能力，微服务这种快速迭代发布，伴随试错试对，快速变更，灰度等，对流量的出入动态性要求很高。

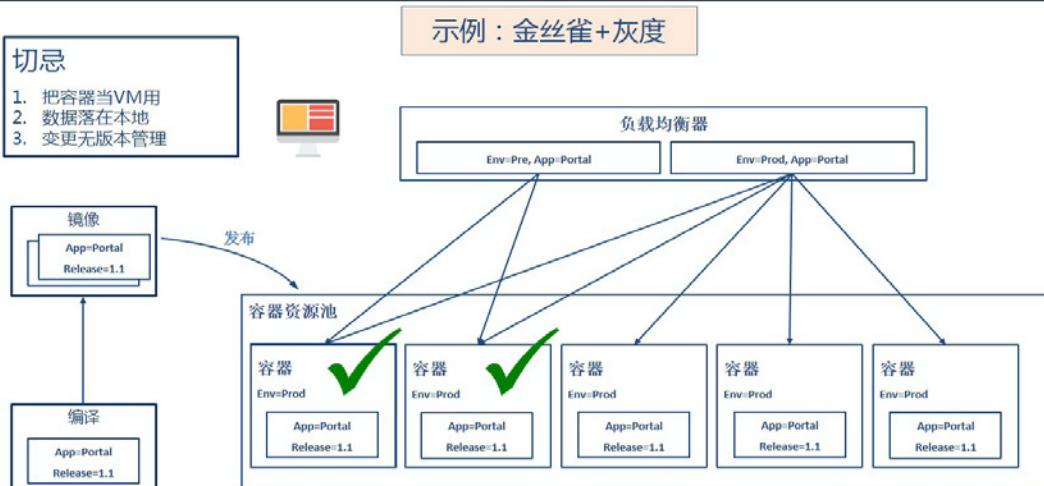
同样是关键 4，这个其实是个有动态效果的图，这没贴上来有点怪，就不赘述了。

关键能力(4) -> 升级与回退



关键 5：前面的原则里大家是否还记得提到了隔离失败，就是这个东西来做的，股市要熔断，微服务同样要，而熔断器的设计参考了标准的三

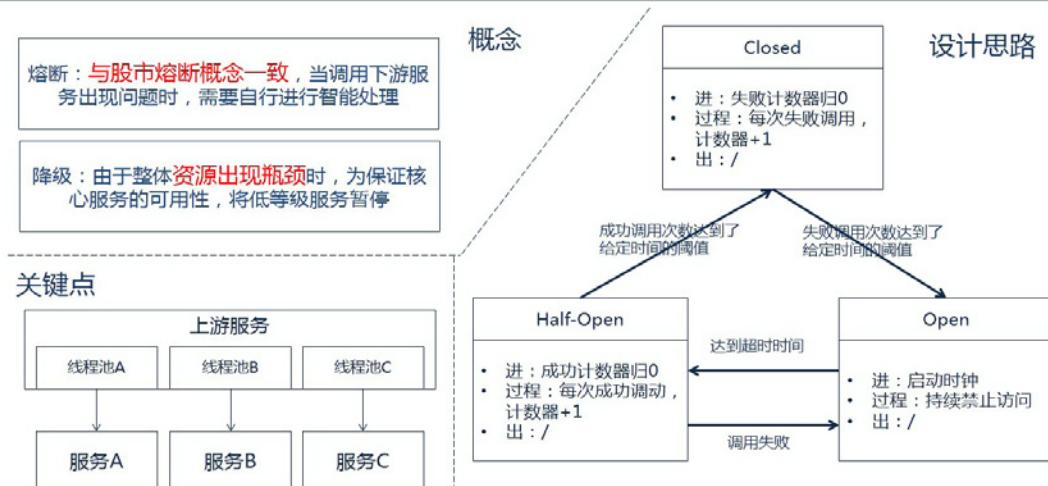
关键能力(4) -> 升级与回退



态设计，默认关闭，调用出错率到一定程度半开，半开时，允许一部分流量继续，如果一定时间还是出错（这个时间结合 MTTR），就全开；

同时还有一个关键，就是上下游调用时，拿上游来说，对于不同微服务的调用，本身要采用不同线程池，防止影响。

关键能力(5) -> 熔断与降级

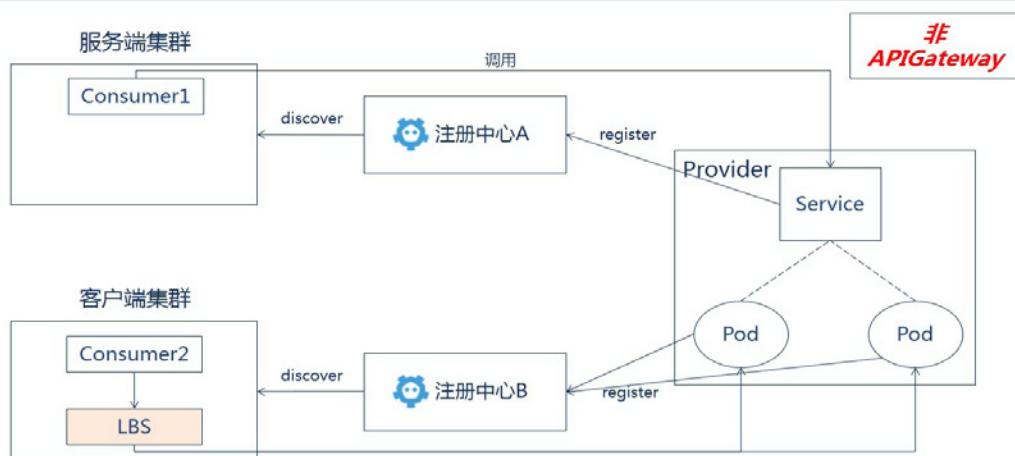


关键 6：微服务注册与发现，和上期的 APIGateway 不一样，这个更多是微服务之间的，通过服务的注册，支撑最终的客户端和服务端集群模式，客户端集群类似 Dubbo，服务端类似传统 Nginx；不同于 dubbo 或 motan，注册中心我们用 etcd 实现。

总结与展望

接着就是第三部分了：我们现在已经在客户那边做一些私有云下的微

关键能力(6) -> 服务注册与发现



服务架构实施，同时对今天的分享做个总结。

跑出去实施才发现，你认为的往往真不是你认为的，客户传统服务也希望拥有微服务的管理能力（比如持续发布、熔断与升级），那需要怎么做；客户的很多小工具都希望在新架构中同样发挥作用，比如像脚本管理这种，那怎么与平台结合。这个一言难尽，建议安排的同学是不是后面我们可以安排分享一些客户案例。

我们在微服务支撑平台的实践过程中，有些经验各位可参考：

尤其最后一条，前些天从饿了吗的架构师那边学习，发现跟开源这件事情，真的是个持续过程，用开源不是简单的使用开源，而是如何 cover

一些过程经验分享

吃自己的狗粮是最好的实践

不要试图挑战别人的总结，往往是自己理解不到位

做任何事都很难一步到位，时刻记得设计要能够得到反馈

跟开源是一件很累的事情

开源。

总结下，今天分享（时间关系做了一部分片子的裁剪），从架构演进及认知误区开始，讲了我们的参考架构，我们的微服务支撑平台的概念模型和关键设计，最后做了简单的实施问题描述和平台时间总结。

最后如果大家对微服务有兴趣，可以看看图灵出版的《微服务设计》这本书，也可以持续关注我们，分享的同时，我们年底也会以书的形式呈现给各位，谢谢！

总结

01

四代架构演进
四个认知误区

02

技术需求来源
主要技术参考
核心概念模型
六块关键设计
领域系统全貌

03

六个实施原则
四个经验总结

作者简介

顾伟，现任普元信息主任架构师。长期致力于 IT 技术研究、产品设计与开发、架构咨询等工作，擅长 Web、OSGI、CI/CD、服务治理、云计算等领域技术。对 DevOps、自动化运维、微服务架构有着浓厚的兴趣。

谈 API 网关的背景、架构以及落地方案

王延炯

Chris Richardson 曾经在他的博客上详细介绍过 API 网关，包括 API 网关的背景、解决方案以及案例。对于大多数基于微服务的应用程序而言，API 网关都应该是系统的入口，它会负责服务请求路由、组合及协议转换。如 Chris 所言，在微服务的应用程序中，客户端和微服务之间的交互，有如下几个挑战：

微服务提供的 API 粒度通常与客户端的需求不同，微服务一般提供细粒度的 API，也就是说客户端需要与多个服务进行交互。

不同的客户端需要不同的数据，不同类型客户端的网络性能不同。

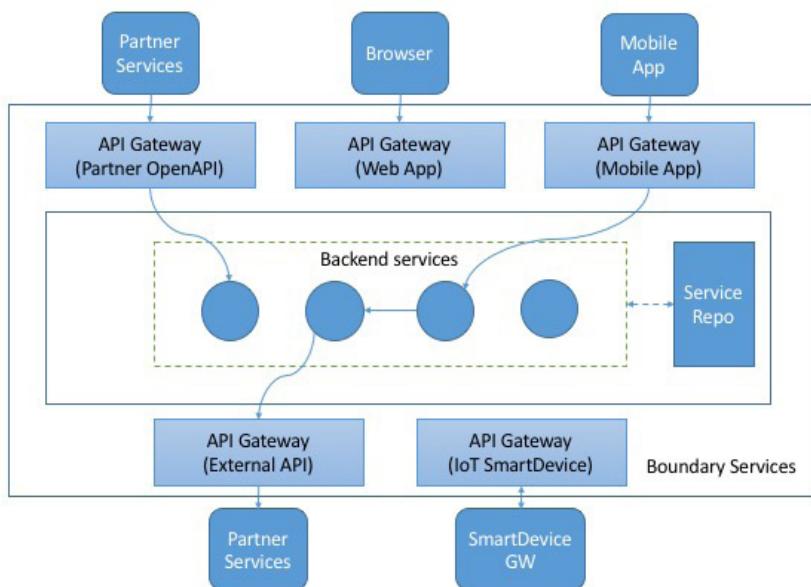
服务的划分可能会随时间而变化，因此需要对客户端隐藏细节。

那 API 网关具体是如何解决这些问题的，在 API 网关的落地时，需要注意哪些地方，就这些问题，InfoQ 编辑采访了普元主任架构师王延炯，与他一起探讨了 API 网关的来龙去脉。

InfoQ：谈谈你所理解的 API 网关，以及 API 网关出现的背景？

王延炯：API Gateway (API GW / API 网关)，顾名思义，是出现在系统边界上的一个面向 API 的、串行集中式的强管控服务，这里的边界是企业 IT 系统的边界。

在微服务概念的流行之前，API GW 的实体就已经诞生了，这时的主要应用场景是 OpenAPI，也就是开放平台，面向的是企业外部合作伙伴，对于这个应用场景，相信接触的人会比较多。当在微服务概念流行起来之后，API 网关似乎成了在上层应用层集成的标配组件。



其实，在我所经历过的项目中，API GW 的定位主要有五类：

1. 面向 Web App

这类场景，在物理形态上类似前后端分离，此时的 Web App 已经不是全功能的 Web App，而是根据场景定制、场景化的 App。

2. 面向 Mobile App

这类场景，移动 App 是后端 Service 的使用者，此时的 API GW 还需要承担一部分 MDM（此处是指移动设备管理，不是主数据管理）的职能。

3. 面向 Partner OpenAPI

这类场景，主要为了满足业务形态对外开放，与企业外部合作伙伴建立生态圈，此时的 API GW 需要增加配额、流控、令牌等一系列安全管控功能。

4. 面向 Partner ExternalAPI

这类场景，业界提的比较少，很多时候系统的建设，都是为了满足企业自身业务的需要，实现对企业自有业务的映射。当互联网形态逐渐影响传统企业时，很多系统都会为了导入流量或者内容，依赖外部合作伙伴的能力，一些典型的例子就是使用「合作方账号登录」、「使用第三方支付平台支付」等等，这些对于企业内部来说，都是一些外部能力。此时的 API GW 就需要在边界上，为企业内部 Service 统一调用外部的 API 做统一的认证、（多租户形式的）授权、以及访问控制。

5. 面向 IoT SmartDevice

这类场景，业界就提的更少了，但在传统企业，尤其是工业企业，传感器、物理设备从工业控制协议向 IP 转换，导致具备信息处理能力的「智能产品」在被客户激活使用直至报废过程中，信息的传输不能再基于 VPN 或者企业内部专线，导致物理链路上会存在一部分公网链路。此时的 API GW 所需要满足的，就是不是前三种单向的由外而内的数据流，也不是第四种由内而外的数据流，「内外兼修」的双向数据流，对于企业的系统来说终端设备很多情况下都不是直连网关，而是进过一个「客户侧」的集中网关在和企业的接入网关进行通信。

InfoQ：在一个微服务架构中，API 网关会在架构中的那一层？他主要的作用是什么？

王延炯：接续前一个话题，我把 API GW 分为了五类，对于当前的企业而言被关注的是前三类或者前四类 API GW。显然，它们都会出现在企业系统的边界上，也就是和企业外部交互的「独木桥」上。

它们除了保证数据的交换之外，还需要实现对接入客户端的身份认证、防报文重放与防数据篡改、功能调用的业务鉴权、响应数据的脱敏、流量与并发控制，甚至基于 API 调用的计量或者计费。

InfoQ：你有研究过 Netflix 的 API 网关吗？在实现方式上，你觉得他们的方式有什么巧妙之处吗？

王延炯：Netflix 的 API GW，主要是指 Zuul，Netflix 将他们用于自己的三大场景：Website Service，API Service，Streaming Service。其中前两个定位与我的前两个分类：Web App，Mobile App 比较类似，第三个 Streaming Service 主要是 netflix 的核心视频业务所形成的特有形态。

Netflix 在 Zuul 的实现上，主要特色是：Filter 的 PRE ROUTING POST ERROR (PRPE 模型)，以及采用 Groovy 脚本的 Filter 实现机制、采用 Cassandra 作为 filter repository 的机制。

Filter 以及 Filter 的 PRPE 模型，是典型的「前正后反模型」的实现，为集成的标准化做好了框架层面的铺垫。

Netflix 其实并没有对 API GW 进行深入的功能实现（或者说面向业务友好的相关功能），整体上它只提供了一个技术框架、和一些标准的 filter 实例实现，相信了解过 filter chain 原理的分布式中间件工程师也能搭出这样的框架。这么做的原因，我认为很大原因是 API GW 所扮演的角色是一个业务平台，而非技术平台，将行业特征很强的业务部分开源，对于受众意义也不是特别大。另外，除了 Netflix Zuul，在商业产品上还有 apigee 公司所提供的方案，在轻量级开源实现上还有基于 Nginx 的 kong，kong 其实提供了 19 个插件式的功能实现，涵盖的面主要在于安全、监控等领域，但缺少对报文转换的能力（为什么缺也很显而易见——避免产生业务场景的耦合，更通用）。

另外，还有基于 TCP 协议的 GW，比如携程无线应用的后端实现有

HTTP 和 TCP 两种，有兴趣的读者也可以深入关注。

InfoQ: 在 API 网关的设计上，需要包含哪些要素？

王延炯: 从三个方面说吧，API 网关本身以及 API 网关客户端，还有配套的自助服务平台。具体如下：

API GW 本身

- NIO 接入，异步接出
- 流控与屏蔽
- 秘钥交换
- 客户端认证与报文加解密
- 业务路由框架
- 报文转换
- HTTP DNS/ Direct IP
- API GW 客户端 SDK / Library

基本通信

- 秘钥交换与 Cache
- 身份认证与报文加解密

配套的在线自助服务平台

- 代码生成
- 文档生成
- 沙盒调测

InfoQ: 在 API 网关的落地上，你有可行的方案吗？在 API 网关的落地上，难点是什么？

王延炯: 在我所服务过的阿里系、非电商互联网公司里，内部的分布式服务调用采用的是 Dubbo，但移动应用是 iOS 和 Android，基本上没有 PC Web 端的客户端，在这种条件下，API GW 所承担的一个重要角色就是

报文转换，并且是跨语言、跨运行平台的报文转换。报文就是数据，在跨平台、跨语言的条件下，对于数据的描述——元数据——也就是类定义，对于 API GW 的系统性挑战是巨大的：传输时，报文内不能传输类定义，跨语言的类定义转换、生成与加载。

API GW 的落地技术基本贯通没有太大的难度，但形成最佳的实践，有一些外围的前置条件，比如：

后端 API 粒度

能和原子业务能力找到映射最好，一定要避免「万能接口」的出现。

业务路由的实现和含报文转换的 API 不停机发布

尽可能的在报文头里面存放业务路由所需要的信息，避免对报文体进行解析。

API GW 上线后，面临的很大问题都是后端服务如何自助发布到外部，同时不能重启网关服务，以保障业务的连续。在此过程中，如果涉及到报文格式的转换，那对 API 网关实现的技术要求比较高。如果让网关完成报文转换，第一种方案，网关需要知道报文的具体格式（也就是报文的元数据，或者是类定义），这部分要支持热更新。第二种方案，需要客户端在报文内另外附加元数据，网关通过运行期加载元数据对报文进行解析在进行报文的转换，这种方案性能不会很好。第三种方案，就是在运行期首次报文转换的时候，根据元数据生成报文转换代码并加载，这种方案对技术实现要求比较高，对网关外围平台支撑力度要求也不低。

客户端的秘钥管理

很多人都会把安全问题简单的用加密算法来解决，这是一个严重的误区，很多时候都存在对秘钥进行系统性管理的短板。打个比方，加密算法就好比家里的保险箱，而秘钥是保险箱的钥匙，而缺乏秘钥管理的安全方案，就好比把钥匙放在自家的客厅茶几上。更何况，安全方案里加解密也只是其中的一部分。

InfoQ：你认为一个设计良好的 API 网关应该做到什么？

王延炯：目前业界关注的 API GW，主要是在前三类，下文对于 API 网关的设计上，侧重于「面向接入」的 API GW。

在 API 网关的设计上，仅仅有类似 Zuul 这样的「面向接入」的运行期框架是远远不够的，因为一个完整的、「面向接入」的 API GW 需要包含以下功能：

面向运行期

- 对客户端实现身份认证
- 通信会话的秘钥协商，报文的加密与解密
- 日常流控与应急屏蔽
- 内部响应报文的场景化裁剪
- 支持「前正后反模型」的集成框架
- 报文格式的转换
- 业务路由的支撑
- 客户端优先的超时机制
- 全局流水号的生成与应用
- 面向客户端支持HTTP DNS / Direct IP

面向开发期

- 自助的沙盒测试环境
- 面向客户端友好的 SDK / Library 以及示例
- 能够根据后端代码直接生成客户端业务代码框架
- 完善的报文描述能力（元数据），支撑配置型的报文裁剪

面向运维与运营

- 支持面向接入方的独立部署与快速水平扩展
- 面向业务场景或合作伙伴的自助API开通
- 对外接口性能与线上环境故障定位自助平台

实施 DevOps 从哪里开始？

刘相

今天分享的主题是加速企业敏捷的 DevOps 平台。DevOps 在 2009 年提出，经过云计算、微服务、容器等技术概念的推动，DevOps 已经被大多数的企业接受并开始付诸实践。

根据我们的实践，接下来我从四个维度为大家分享 DevOps。

一、DevOps 的认知

我们先来看个问题，企业实施 DevOps 的情况：应用上线（哪怕是改动一行代码）需要多长时间？

大家的周期通常是月、周、天、小时？

如果大家发布周期在周级别，还有大量的工作靠人工执行，我们需要尽快引入 DevOps 了。

目前业界对 DevOps 的了解可谓千人千面，我们先看下维基百科给出的定义。



DevOps 是一组过程、方法与系统的统称，用于促进开发、运维部门之间的沟通、协作与整合。

DevOps 是提倡开发和 IT 运维之间的高度协同，从而在完成高频率部署的同时，提高生产环境的可靠性、稳定性、弹性和安全性。

从广义的角度来讲，我们认为 DevOps 应该从支持项目敏捷到支撑企业敏捷。



我们认为：DevOps 不仅是打通开发运维之间的部门墙，更多的需要从应用的全生命周期考虑，实现应用全生命周期的工具链路打通、跨团队的线上协作能力。

纵向集成中 DevOps 强调的重点是跨工具链的「自动化」，最终实现

全部人员的「自助化」服务。

横向集成中 DevOps 强调的重点是跨团队的「线上协作」，也即是通过 IT 系统，实现信息的「精确传递」。

对于 DevOps 的理解，目前业界存在不少的误区，希望大家在实践的时候能够快速跳过这些误区，成功实施 DevOps。



采用了云计算（IaaS、容器）才能开展 DevOps，确切的讲应该是采用云计算有助于加速 DevOps 的落地，云计算决不是实施 DevOps 的先决条件，传统的基础设施一样可以支撑 DevOps 的落地。

微服务架构开发的应用才适合；实施 DevOps 与应用架构无关，无论是采用微服务架构、SOA 架构，都可以开展 DevOps 工作。

采用自动化工具本身不是 DevOps，只有将这些工具与持续集成、持续交付、持续的反馈与优化进行端到端的整合时，这些工具才成为 DevOps 的一部分。

设置独立的 DevOps 部门，在责任没有清晰定义的情况下，这么做会导致更多的竖井，创造更多的混乱。

自动化是 DevOps 非常重要的一部分，但不是唯一的部分。

我们认为，实施 DevOps 需要从敏捷、持续、协作、系统性、自动化五个维度进行建设与改进。

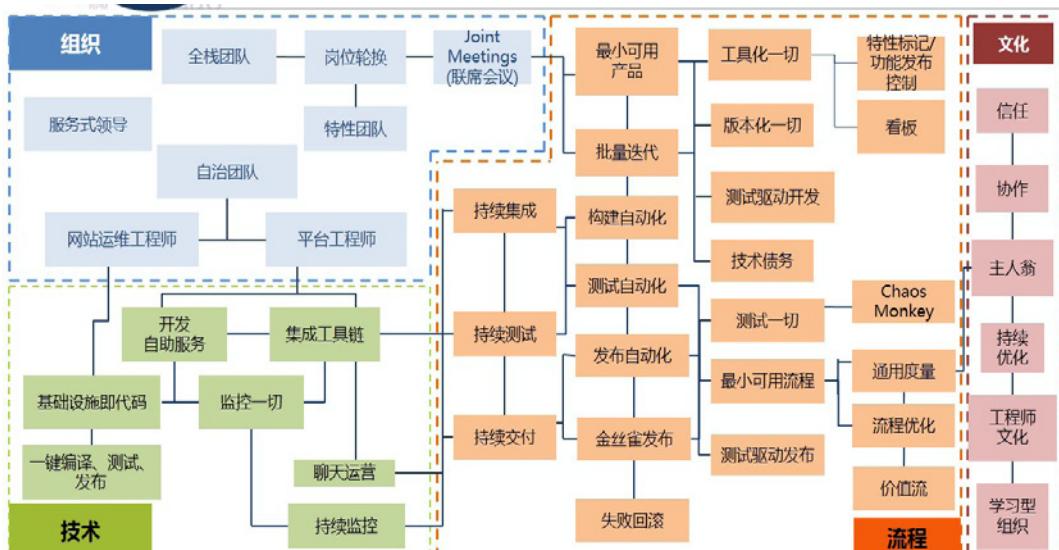


敏捷、自动化大家已经比较熟悉，大部分企业也已经付诸了实践工作。

另外我们还需要实现跨部门与组织的协作，从技术、流程维度实现系统的改进；最后我们认为实施 DevOps 是一个持续的过程，需要不断的进行总结、反馈、优化。

二、DevOps 实践总结

接下来给大家看下我们在组织、技术、流程方面的一些实践。



实施 DevOps，可以参考总结的“DevOps 实践模型”，从组织、技术、流程三个维度中选择部分开始实践。

根据我们的实施经验，在传统企业中，技术方面的实践最容易在团队

中实现、流程次之、组织的优化与变革最为艰难；大家尝试的时候，可以由易入难。

接下来我们看如何在组织方面实现敏捷。



全栈团队



特性团队



自治团队

全栈团队，而非全栈员工，按照「两个披萨原则」进行团队组建；团队分组是需要基于特性而非技术维度进行团队划分，确保每个团队开发出来的都是可用产品；

特性团队，是指在大型项目中，根据功能特性进行团队的划分与组件，而不是根据技术特性，根据功能特性组件的团队每次交付的都是用户可用的产品，可以提前进行确认，避免项目结束时候发现交付的产品是不可用的。

通过全栈和特性团队的磨练，逐渐形成**自治的、自交付的团队组织**。



基础设施
即编码



一键编译、
测试、部署



Chat DevOps

在技术层面，我们实施了基础设施即编码的能力，将基础环境可编程

化，项目团队成员可以自助获取；

形成持续编译、自动化测试、持续部署的能力；

另外我们正在做一件比较有意义的事情，**ChatDevOps**。ChatDevOps 是基于对话驱动的，将开发、运维工具植入对话中的，一批的开发运维机器人为我们提供各种服务。大家如果有兴趣，可以关注下 hubot。



看板



MVP



发布



软件度量

在制品管理
量化组织生产能力

快速迭代
市场快速验证

自动化、自助化能力
灰度、金丝雀、蓝绿、回滚

从感性到理性
优化组织瓶颈点
掌控组织生产力

在流程方面，我们实施了**看板文化**。看板通常被当作任务协调沟通的机制；我们把看板作为在制品管制平台，量化组织生产能力；

在产品交付上采用**MVP 模式**，快速交付产品原型，通过市场来验证，修正产品，最终适应市场的需求；

每个项目必须建立持续发布机制，形成自动化、自助化两种能力；

建立度量体系，让数据说话，建立组织的各种数据基线，一方面可以掌控组织的生产力水平，另一方面通过度量数据，反向优化组织瓶颈点；

基于上面的最佳实践，总结了企业**DevOps 宣言**。

三、构建 DevOps 平台

我们认为实施 DevOps 的终极目标是加速企业的敏捷转型，从根本上提升 IT 的生产效率，加速部门、企业的业务创新能力。让团队从 IT 支撑部门，转向为 IT 创新部门。

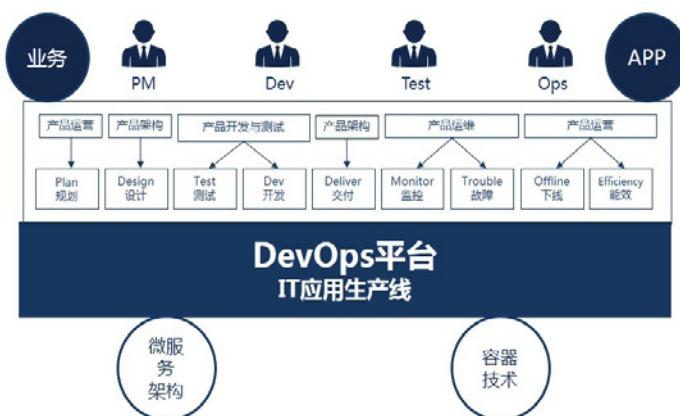
针对技术、流程我们通过平台进行了最佳实践的固化，形成了支持

DevOps 的平台。

	目标	加速企业敏捷转型
	组织	充分授权的自组织的团队 全栈团队而不是全栈工程师
	流程	自动化一切 (所有资源变更通过自动化流程发布) MVP交付
	技术	Everything is code (配置、脚本、数据、测试代码、基础设施均是代码)
	文化	建立信任、协作关系 学习型组织

在平台建设时，一个非常重要的思路是建设“**以应用为中心的 DevOps 平台**”。大家如果关注业界 DevOps 平台的话，会发现市面上的 DevOps 平台更多的是偏向“以资源为中心的”，提供更多的是创建容器、VM 的能力。

- 以应用为中心
- 统一工作台
- 工具链打通
- 生产力度量
- 支持多种环境
- 支撑高可靠



DevOps 平台通常具备的几个核心特性：

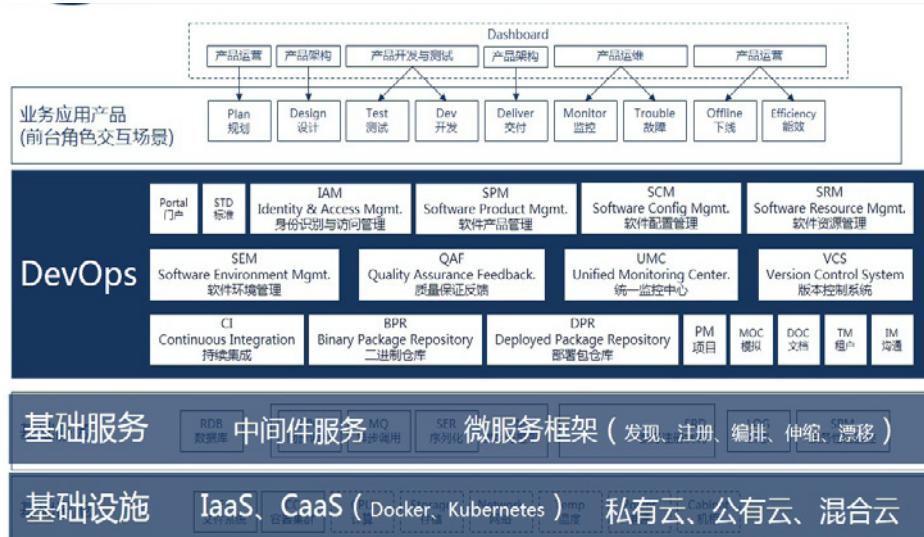
1. 完整的 DevOps 平台至少提供统一的工作台，支持部门的协同工作；
2. 打通工具链，做到自动化和自助化；
3. 实现研发过程的度量，建立组织基线数据；

4. 无缝支持多种环境公有云、私有云，常见的容器、VM；
5. 运行期提供应用高可靠、伸缩漂移等能力。

大家可以看下我们在 DevOps 平台打通的工具链。



如果团队要自主掌握庞大的工具需要大规模的团队，而使用统一的工作台可以简化整个工具的使用。



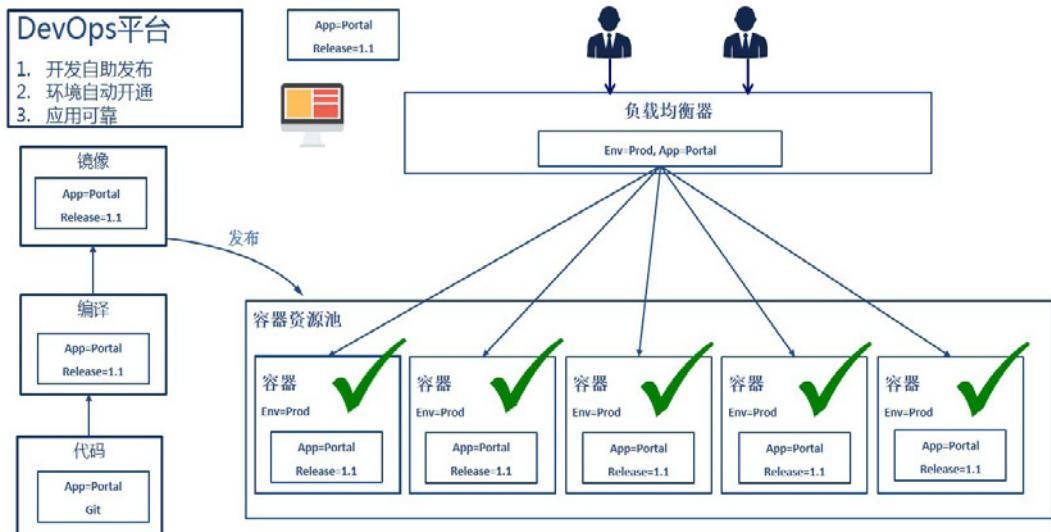
基于容器云的 DevOps 平台主要分为三层：

1. 基础设施层：包括 IaaS, CaaS，我们分别是基于 Kubernetes、

Docker 实现，上层有一层不同环境的适配，可以无缝对接私有云、公有云、混合云；

2. 基础服务层：包括服务管理与调度的基础能力，如注册中心，编排，伸缩漂移；还有一堆具体的企业级或互联网式的云服务；
3. DevOps 层：提供支撑全生命周期的 18 大领域系统更多的是工作流程（需求、设计、开发、测试、发布等）的串接，看板等文化的体现。

为大家展示下一键发布能力，通过 DevOps 平台，可以一键从源代码获得可访问的环境（自动根据应用的部署编排，实现了自动化的编译、集成、打包、部署、启动等）。



实施 DevOps 后的改变，首先团队变得更自治，成为使命型组织；沟通协作更顺畅；实现了开发人员的自助化服务；开发运维机器人提供更多的辅助功能。

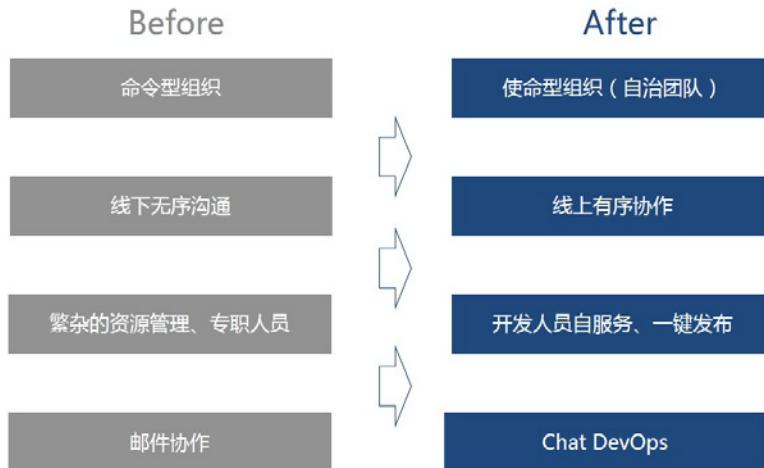
四、实施 DevOps 从哪里开始？

大家可能非常关心，如何在各自的企中如何落地 DevOps 平台呢？

在企业进行 DevOps 落地时，我们给大家推荐两个原则：

1. 寻找痛点，从痛点入手；

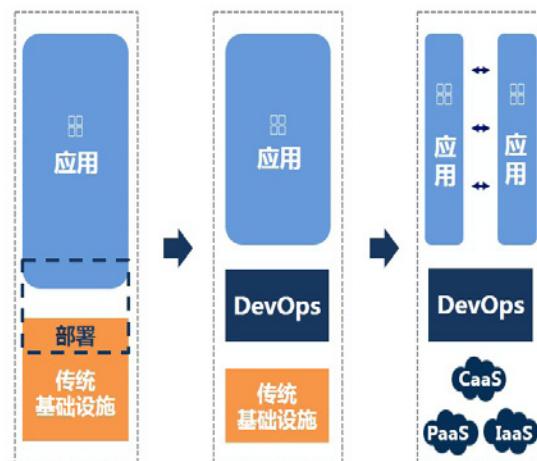
2. 将重复的、无价值的事情尽快自动化。



基于这两个原则，我们认为持续的部署，是目前企业实施最大的痛点，因此推荐实施 DevOps 从持续发布开始，后续可以建设量身定制的 DevOps 平台。逐渐搭建企业的云计算平台、采用微服务架构进行应用的拆分。

- 寻找痛点，以痛点切入
- 重复、无价值的尽快自动化
- 从持续发布开始
- 建设量身定制DevOps平台

实施云计算从 DevOps 开始，
实施 DevOps 从持续发布开始



最后，我们回顾下今天的分享，一共分享了三方面的知识。

- 第一：我们对 DevOps 的狭义和广义的理解；
- 第二：在实施 DevOps 过程中，需要从组织、技术、流程三个维度进行改进；
- 第三：我们讨论了实施 DevOps 从持续发布开始。

对DevOps的 2个认知

狭义
广义

实施DevOps 3个维度

组织
技术
流程

实施DevOps 从哪里开始

实施DevOps从持续发布
开始
搭建DevOps平台

作者简介

刘相，计算机应用技术硕士，现任普元软件产品部副总兼 SOA 产品线总经理。十年 IT 行业经验，专注于企业软件平台，在 SOA、分布式计算、企业架构设计等领域。先后主导公司 EOS7、Portal、云 PAAS 平台、云流程平台、BPM 等系列产品的开发和设计工作。著有国内首本解析 SpringBatch 的中文原创图书《SpringBatch 批处理框架》。个人爱好：阅读，慢跑。

授权说明

本刊内容来自 **EAII 企业架构创新研究院** 原创，授权极客邦科技发布。

普元是国内领先的软件基础平台与解决方案提供商，主要面向大中型企业、政府机构及软件开发商提供 SOA、大数据、云计算三大领域的软件基础平台及解决方案，用以满足上述组织信息化建设对关键技术的需求，帮助上述组织的业务在云计算和移动互联时代向数字化转型。

2016 年，普元全面开展 InsideOut——普元云计算研发设计开放计划，通过对普元数字化企业云平台“*The Platform*”的研发过程、研发文档、研发技术的全面开放，邀请产品用户从软件的研发设计阶段就深度参与，提需求共探讨，来形成软件行业的“众筹模式”，从而降低技术的转化成本和创新的社会化成本。目前 InsideOut 研发开放计划已经吸引超过 23000 名来自中国企业信息部门的技术主管和高层管理者的加入。

与此同时，InsideOut 计划中众多技术专家、技术内容和观点通过 InfoQ 这一全球性技术社区平台被更多技术人了解，为此普元与 InfoQ 共同将这些优质技术内容整理成该特刊，希望分享给更多技术人，共同交流。

关注企业架构创新研究院微信公众账号

由普元赞助设立的 EAII (Enterprise Architecture Innovation

Institute) 企业架构创新研究院（微信公众账号：eeworld），是专注于企业架构与业务创新领域的研究机构，EAII 托管运行着整个普元的 InsideOut 研发开放设计计划，并将实现普元云计算平台知识和体验的汇集。



加入由本刊作者主持的普元云计算研发设计群

在公众账号中，回复“**TK+微信号**”，即可报名加群。普元云计算研发设计群，是普元研发团队与 架构师团队共同探讨云计算平台架构设计的 讨论群，现已全面对外开放。

版权声明

InfoQ 中文站出品

微服务与 DevOps 技术内参

©2017 极客邦控股（北京）有限公司

本书版权为极客邦控股（北京）有限公司所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：极客邦控股（北京）有限公司

北京市朝阳区洛娃大厦 C 座 1607

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn