Microservice 微服务的理论模型和现实路径

两年前接触到了微服务的概念，面对日益膨胀的系统感觉豁然开朗。之后的两年逐步把系统按微服务的架构理念进行了重构，并将业务迁移到了新架构之上。感觉现在差不多是时候写一篇关于微服务的总结文章了。

定义

在 Martin Fowler & James Lewis 的文章(参考[1])里给出了微服务架构的一个定义：

> 微服务架构即是采用一组小服务来构建应用的方法。
> 每个服务运行在独立的进程中，不同服务通过一些轻量级交互机制来通信， 例如 RPC、HTTP 等。
> 服务围绕业务能力来构建，并依赖自动部署机制来独立部署。

这个定义相对还是模糊，但还是勾勒出了微服务的一些关键概念：小，独立进程，自动化。

起源

从微服务的定义，我们感觉似曾相识。早在 1994 年 Mike Gancarz 曾提出了 9 条著名原则(参考[4])，其中前 4 条和微服务架构理念特别接近。微服务就像把 UNIX 哲学应用到了分布式系统（参考[3]）。

> 1. Small is beautiful.
> 2. Make each program do one thing well.
> 3. Build a prototype as soon as possible.
> 4. Choose portability over efficiency.

- 小即是美：小的服务代码少，bug 也少，易测试，易维护，也更容易不断迭代完善的精致进而美妙。
- 一个程序只做好一件事：一个服务也只需要做好一件好，专注才能做好。
- 尽可能早地创建原型：尽可能早的提供服务 API，建立服务契约，达成服务间沟通的一致性约定，至于实现和完善可以慢慢再做。
- 可移植性比效率更重要：服务间的轻量级交互协议在效率和可移植性二者间，首要依然考虑兼容性和移植性。

可见微服务其实不是凭空产生的，它自有其历史的渊源。而在微服务之前的十年，大家经常谈论的是一个叫 SOA（面向服务）的架构模式，它和微服务又是什么关系？在 Sam Newman 的《Building Microservices》（参考[2]）一书中，作者对 SOA 和 Micorservices 的区别给出了定义：

> You should instead think of Microservices as a specific approach for SOA in the same way that XP or Scrum are specific approaches for Agile software development.

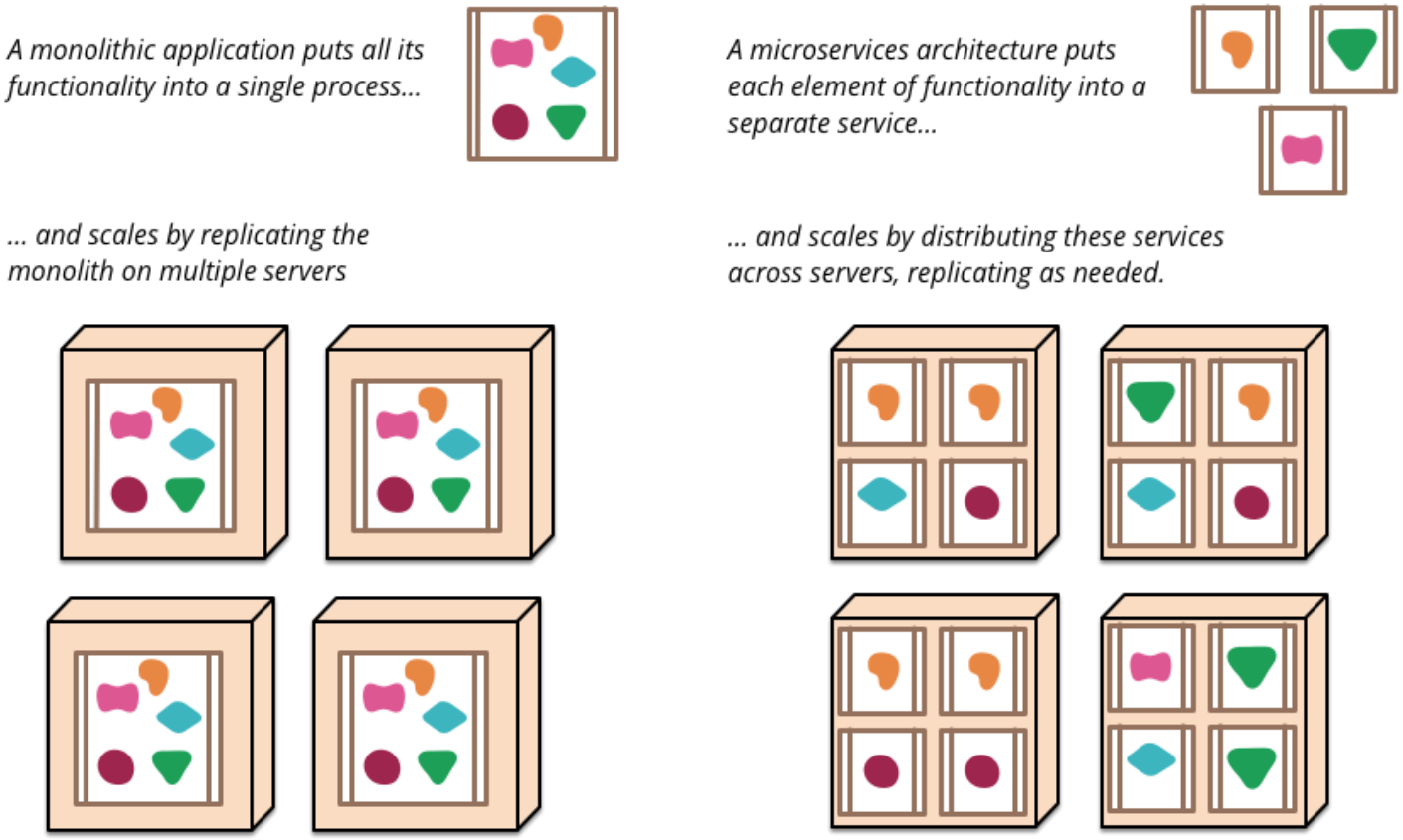
你可以把微服务想成是 SOA 的一种实践方式，正如 XP 或 Scrum 是敏捷软件开发的实践方式。我对这个定义是认同的，面向服务架构（SOA）的概念已有十多年，它提出了一种架构设计思想， 但没有给出标准的参考实现，而早期企业软件业界自己摸索了一套实践方式 —— 企业服务总线（ESB）。 但历史证明 ESB 的实现方案甚至在传统企业软件行业也未取得成功，Martin Fowler 在文中说正是因为 ESB 当年搞砸了很多项目， 投入几百万美金，产出几乎为零，因此 SOA 这个概念也蒙上了不详的标签，所以当微服务架构出现时， 其拥护者开始拒绝使用包裹着失败阴影的 SOA 这个标签，而直接称其为微服务架构（Microservices Architecture Style）， 让人以为是一套全新的架构思想，但事实上它的本质依然是 SOA 的一种实践方式。

特征

一个按微服务架构理念构建的系统应该具备什么样的特征呢？Martin 在其文章（参考[1]）中做了详尽的阐述，我这里简单归纳下。

组件服务化

传统实现组件的方式是通过库（library），库是和应用一起运行在进程中，库的局部变化意味着整个应用的重新部署。 通过服务来实现组件，意味着将应用拆散为一系列的服务运行在不同的进程中，那么单一服务的局部变化只需重新部署对应的服务进程。



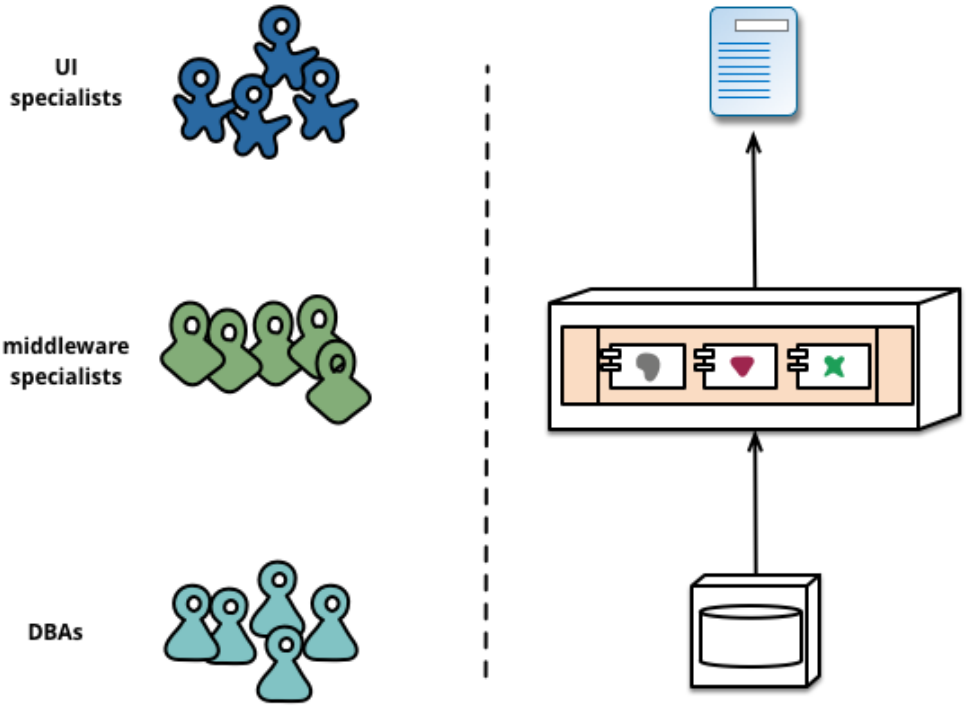
按业务能力组织服务

按业务能力组织服务的意思是服务提供的能力和业务功能对应，比如：订单服务和数据访问服务，前者反应了真实的订单相关业务，后者是一种技术抽象服务不反应真实的业务。所以按微服务架构理念来划分服务时，是不应该存在数据访问服务这样一个服务的。

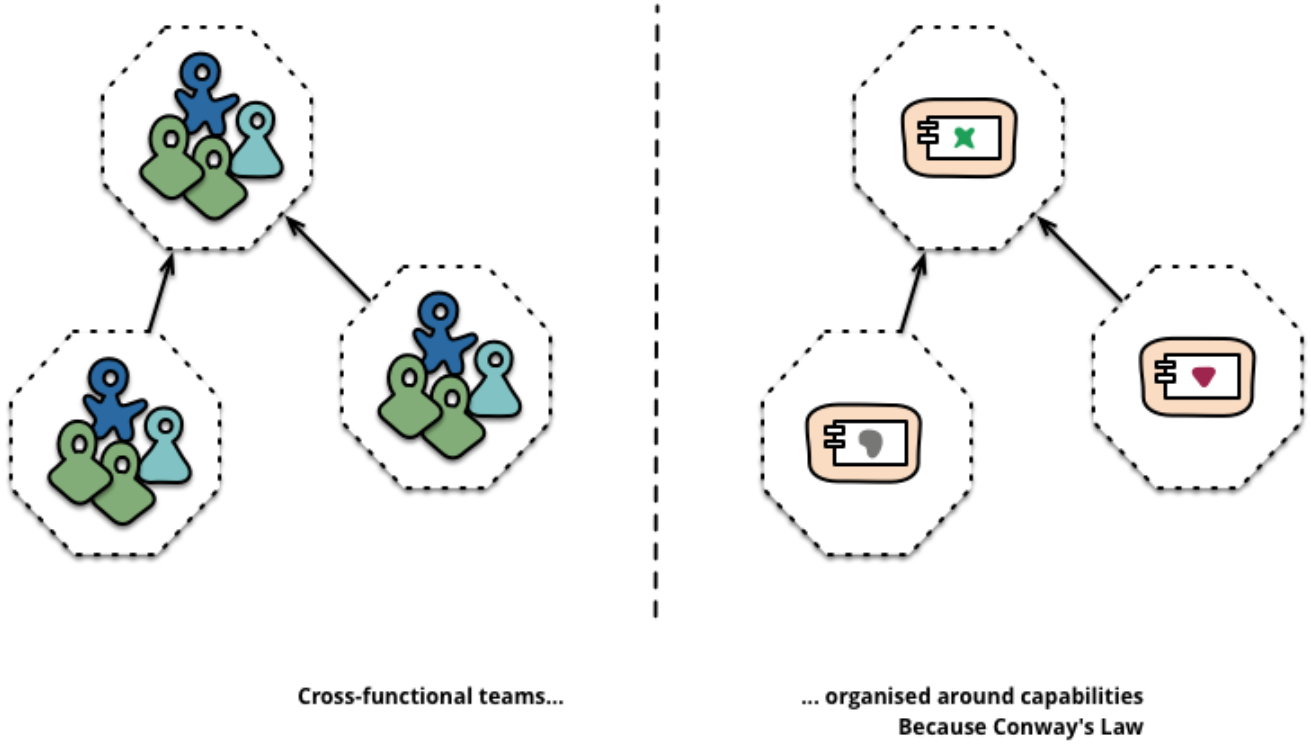
Melvin Conway 在 1967 年观察到一个现象并总结出了一条著名的康威定律（参考[5]）：

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

设计系统的组织，最终产生的设计等价于组织的沟通结构。传统开发方式中，我们将工程师按技能专长分层为前端层、中间层、数据层，前端对应的角色为 UI、页面构建师等，中间层对应的角色为后端业务开发工程师，数据层对应着 DBA 等角色。



事实上传统应用设计架构的分层结构正反应了不同角色的沟通结构。所以若要按微服务的方式来构建应用，也需要对应调整团队的组织架构。每个服务背后的小团队的组织是跨功能的，包含实现业务所需的全面的技能。



服务即产品

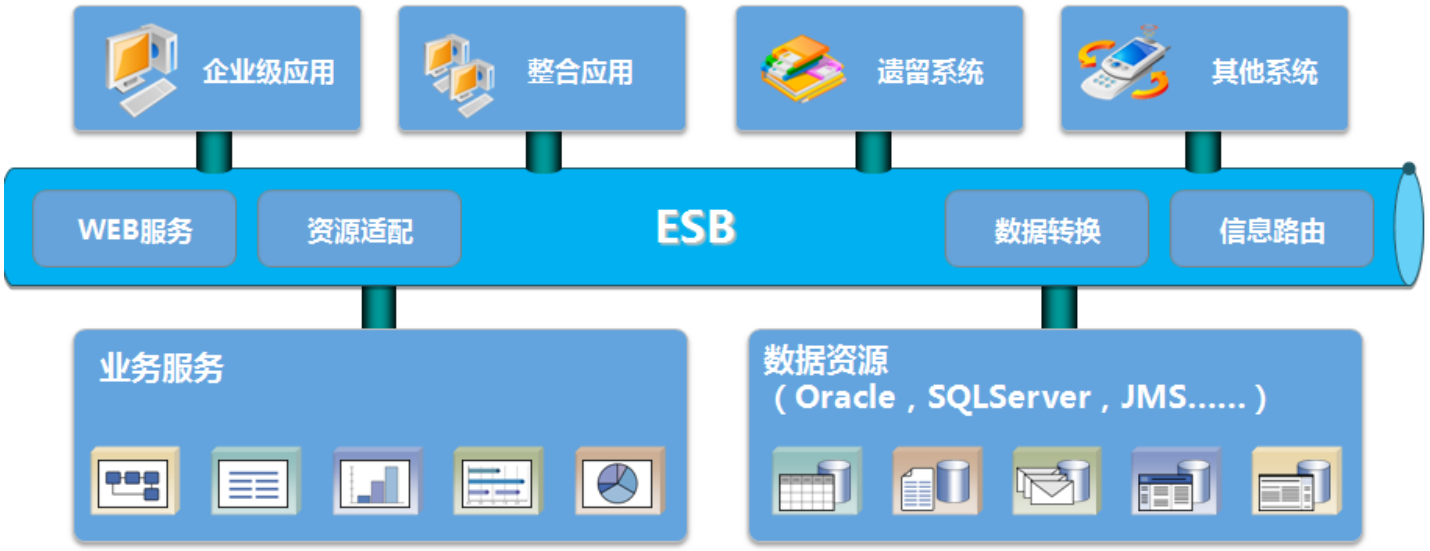
传统的应用开发都是基于项目模式的，开发团队根据一堆功能列表开发出一个软件应用并交付给客户后，该软件应用就进入维护模式，由另一个维护团队负责，开发团队的职责结束。 而微服务架构建议避免采用这种项目模式，更倾向于让开发团队负责整个产品的全部生命周期。Amazon 对此提出了一个观点：

> You build it, you run it.

开发团队对软件在生产环境的运行负全部责任，让服务的开发者与服务的使用者（客户）形成每日的交流反馈，来自直接客户的反馈有助于开发者提升服务的品质。

智能终端与哑管道

微服务架构抛弃了 ESB 过度复杂的业务规则编排、消息路由等。 服务作为智能终端，所有的业务智能逻辑在服务内部处理，而服务间的通信尽可能的轻量化，不添加任何额外的业务规则。所以这里的智能终端是指服务本身，而哑管道是通信机制，可以是同步的 RPC，也可以是异步的 MQ，它们只作为消息通道，在传输过程中不会附加额外的业务智能。



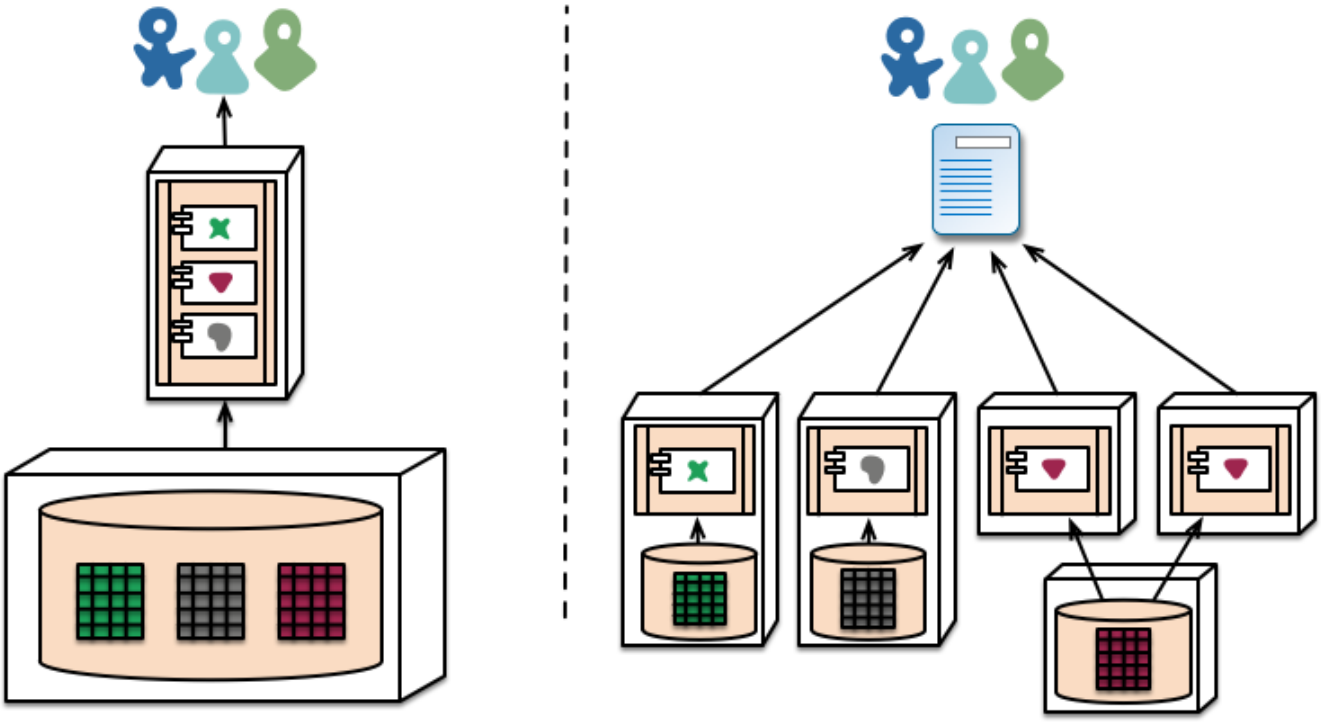
去中心化

去中心化包含两层意思：

1. 技术栈的去中心化。
2. 数据去中心化。

每个服务面临的业务场景不同，可以针对性的选择合适的技术解决方案。但也需要避免过度多样化，结合团队实际情况来选择取舍，要是每个服务都用不同的语言的技术栈来实现，想想维护成本真够高的。

每个服务独享自身的数据存储设施（缓存，数据库等），不像传统应用共享一个缓存和数据库，这样有利于服务的独立性，隔离相关干扰。



基础设施自动化

无自动化不微服务，自动化包括测试和部署。单一进程的传统应用被拆分为一系列的多进程服务后，意味着开发、调试、测试、监控和部署的复杂度都会相应增大，必须要有合适的自动化基础设施来支持微服务架构模式，否则开发、运维成本将大大增加。

容错设计

著名的 Design For Failure 思想，微服务架构采用粗粒度的进程间通信，引入了额外的复杂性和需要处理的新问题，如网络延迟、消息格式、负载均衡和容错，忽略其中任何一点都属于对“分布式计算的误解”。

兼容设计

一旦采用了微服务架构模式，那么在服务需要变更时我们要特别小心，服务提供者的变更可能引发服务消费者的兼容性破坏，时刻谨记保持服务契约（接口）的兼容性。一条普适的健壮性原则（伯斯塔尔法则，参考[6]）给出了很好的建议：

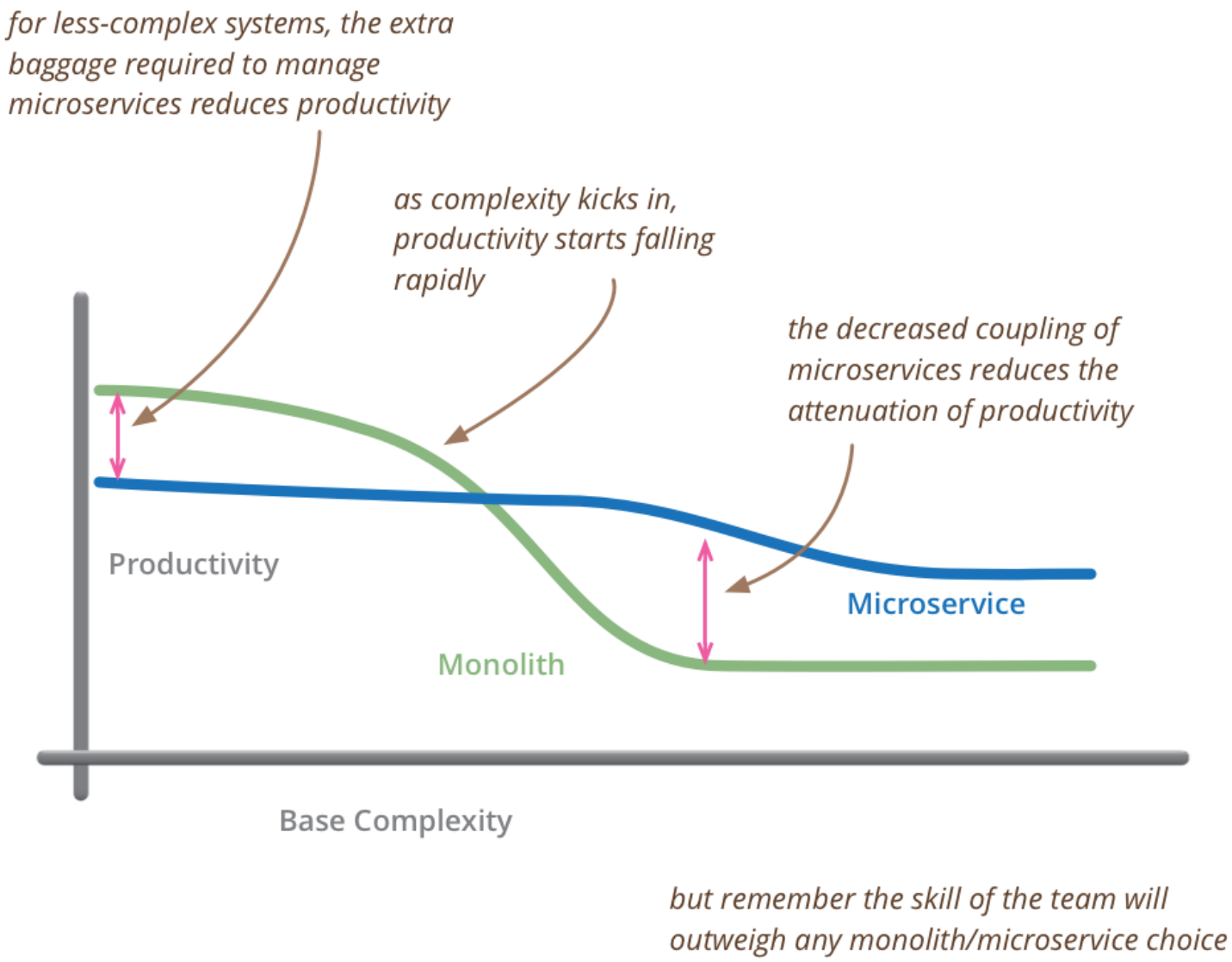
> Be conservative in what you send, be liberal in what you accept.

发送时要保守，接收时要开放。按照伯斯塔尔法则的思想来设计和实现服务时，发送的数据要更保守，意味着最小化的传送必要的信息，接收时更开放意味着要最大限度的容忍冗余数据，保证兼容性。

实施

前提

微服务似乎是一个近年很热门的架构选择，但什么时候该选择微服务架构，这是有一定前提的。



上面的图来自 Martin Fowler 的文章(参考[7])，揭示了生产率和复杂度的一个关系。在复杂度较小时采用单体应用（Monolith）的生产率更高，复杂度到了一定规模时，单体应用的生产率开始急剧下降，这时对其进行微服务化的拆分才是合算的。

图上标明了复杂度和生产率拐点的存在，但并没有量化复杂度的拐点到底是多少？或者换种说法系统或代码库的规模达到具体多大才适合开始进行微服务化的拆分。在一篇有趣的文章《程序员职业生涯中的 Norris 常数》（参考[9]）中提到大部分普通程序员成长生涯的瓶颈在 2 万行代码左右。

> 当代码是在 2,000 行以下，你可以写任何混乱肮脏的代码并依靠你的记忆拯救你。深思熟虑的类和包分解会让你的代码规模达到 20,000 行。

两万行是作者经历过并反复碰到的一个瓶颈点，于我也有同感。

> 初级程序员，学会了爬行，接着蹒跚学步，然后行走，然后慢跑，然后再跑步，最后冲刺，他认为，"以这样加速度前进我可以赶上超音速喷气式飞机的速度！" 但他跑进了 2,000 行的极限，因为他的技能不会再按比例增加。他必须改变移动方式，比如开车去获得更快的速度。然后，他就学会了开车，开始很慢，然后越来越快，但又进入到了 20,000 行的极限。驾驶汽车的技

术不会让你能够开喷气式飞机。

所以每一个瓶颈点的突破意味着需要新的技能和技巧，而结合我自己的经历和经验，微服务的合适拆分拐点可能就在两万行代码规模附近，而每个微服务的规模大小最好能控制在一个普通程序员的舒适维护区范围内。借用前面的比喻，一个受过职业训练的普通程序员就像一个拿到驾照的司机，一般司机都能轻松驾驭 100 公里左右的时速，但很少有能轻松驾驭 200 公里或以上时速的司机，即使能够风险也是很高的。而能开喷气式飞机的飞行员级别的程序员恐怕在大部分的团队里一个也没有。

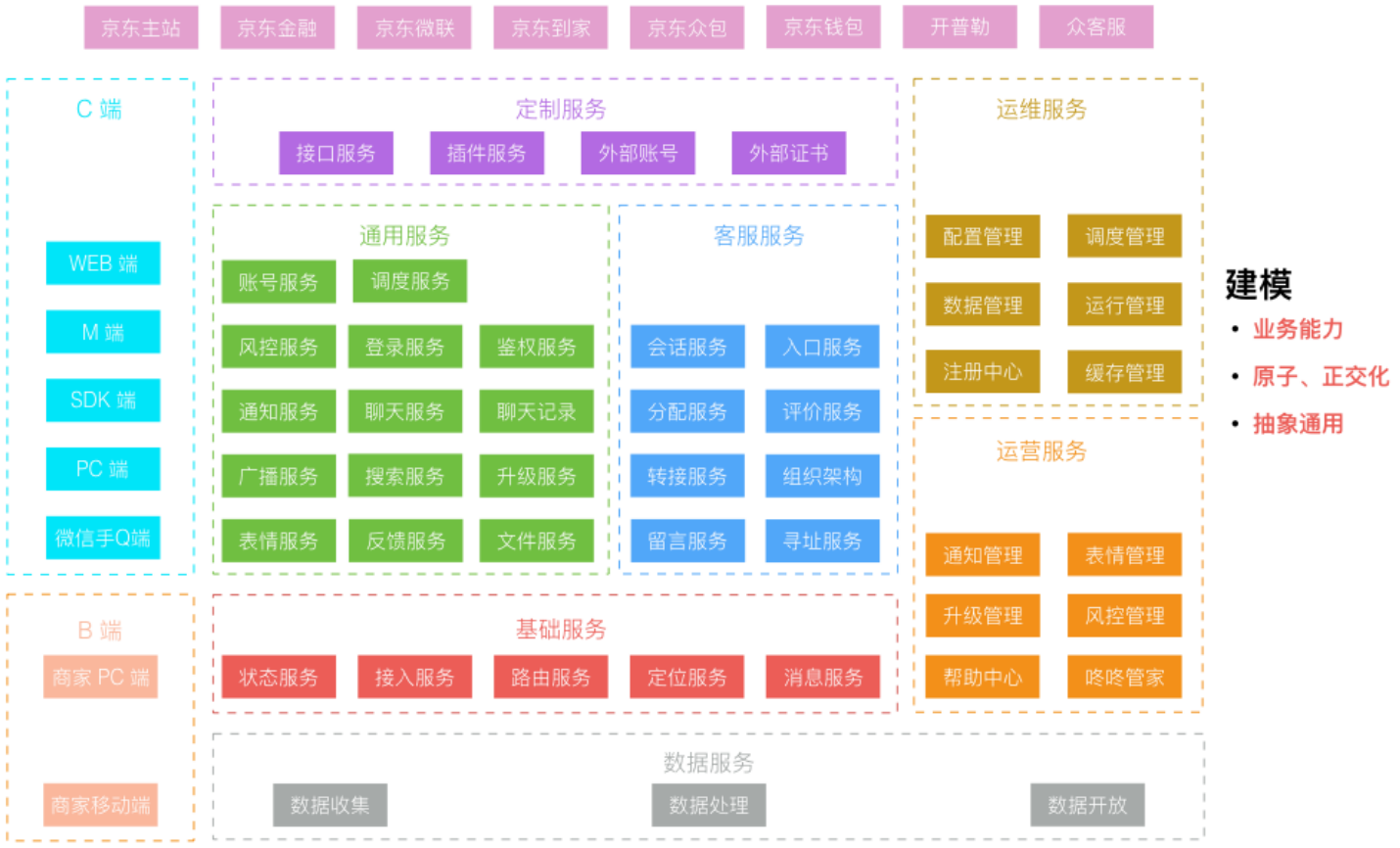
另外一个实施前提是基础设施的自动化，把 1 个应用进程部署到 1 台主机，部署复杂度是 1 x 1 = 1，若应用规模需要部署 200 台主机，那么部署复杂度是 1 x 200 = 200。 把 1 个应用进程拆分成了 50 个微服务进程，则部署复杂度变成了 50 x 200 = 10000，缺乏自动化设施，光部署就会把人搞死。所以前面微服务的特征才有基础设施自动化，这和规模也是有关的，这也是因为其运维复杂度的乘数级飙升， 从开发之后的构建、测试、部署都需要一个高度自动化的环境来支撑才能有效降低边际成本。

维度

实施微服务架构，可以从下面一些维度来做全面考量。

建模

服务围绕业务能力建模，下图是我在《京东咚咚架构演进》（参考[10]）一文中写到的咚咚向微服务架构演进中对服务拆分后得到的一个服务矩阵图。从服务名称就可以很容易看出服务比较清晰的反应了业务能力。

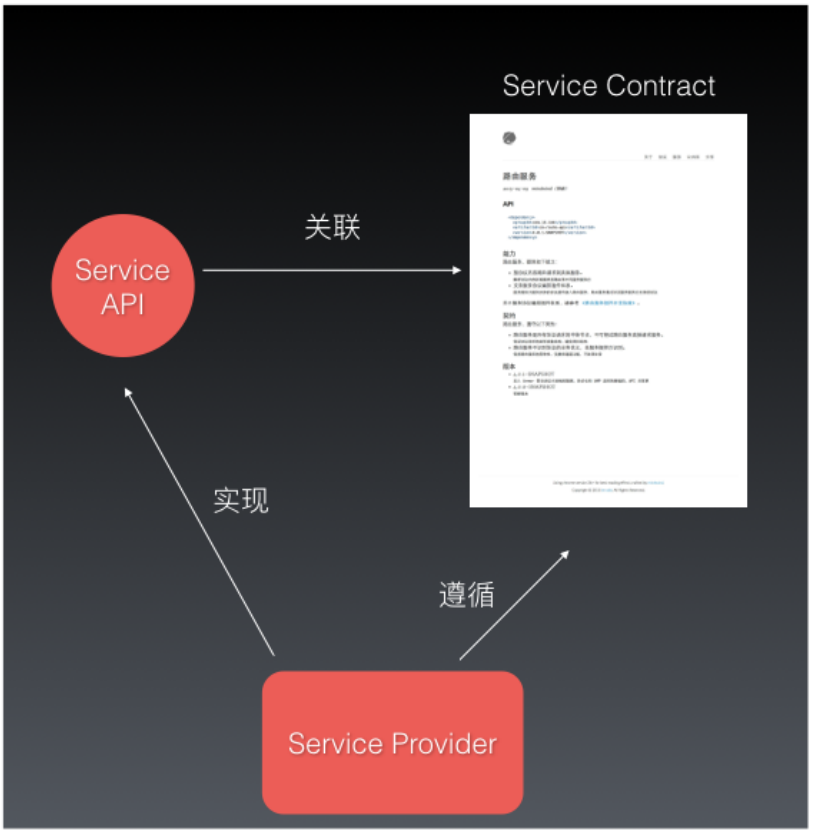


协作

采用微服务架构模式后，开发和运行的协作模式都会发生变化，还是以我们实践的经验为例来讲下。

按微服务的组织方式，不同人或小团队负责一个或一组微服务，服务之间可能存在相互调用关系，所以在服务之间也完全采用了像面向外部开放的契约化开发模式。

- 服务契约
  - API
  - 能力
  - 契约
  - 版本

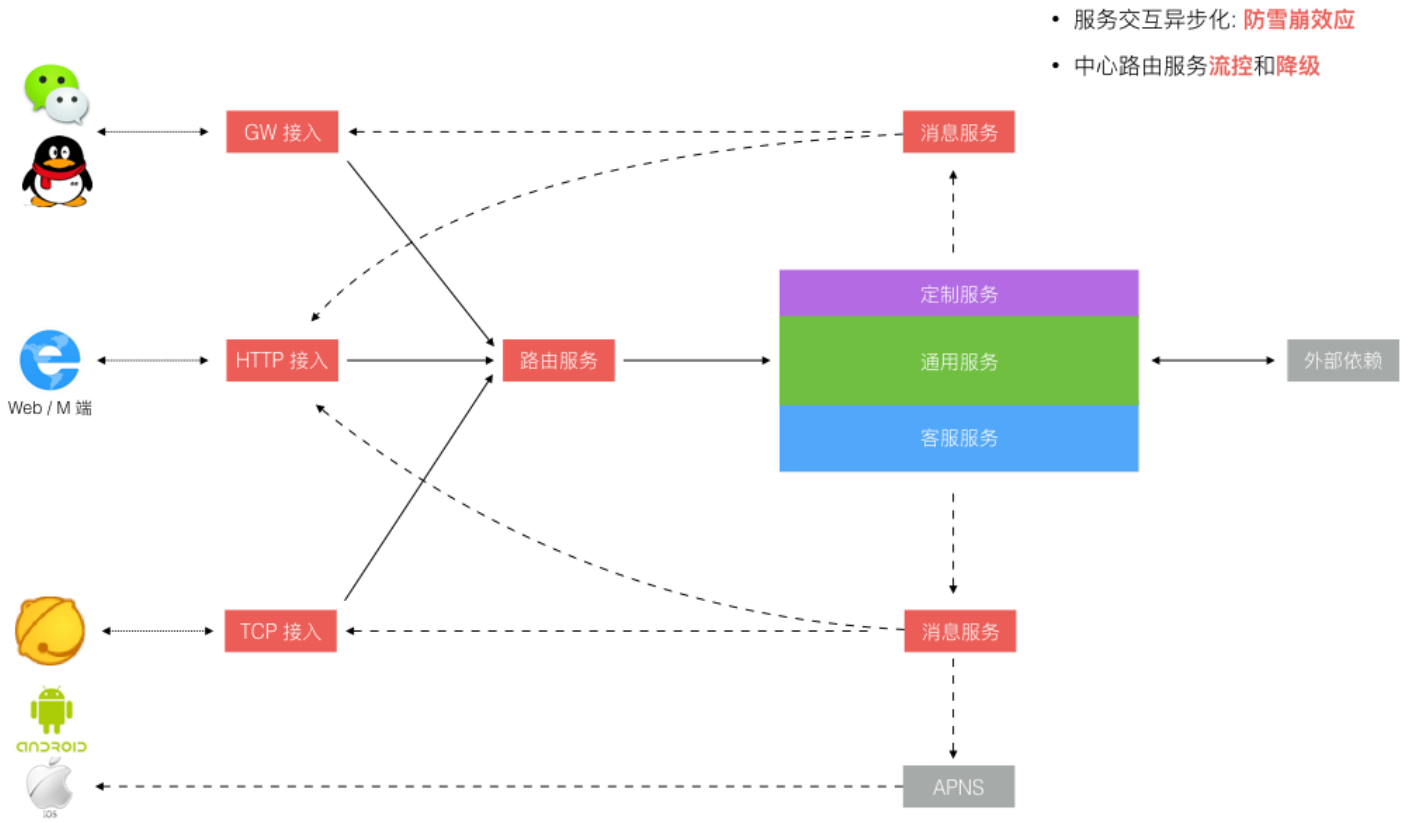


每一个服务都提供了一份契约文档，发布到公开的内部 wiki，方便服务干系人可自由获取查看。契约文档要求至少对服务的几个基本方面作出说明，如下：

- API，具体接口的 API 接入技术说明。
- 能力，服务能力的描述。
- 契约，提供这些能力所约定的一些限制条件说明。
- 版本，支持的最新和历史的版本说明。

使用契约文档来减少多余且可能反复重复的口头沟通，降低协作成本。

采用微服务后一个业务功能的调用会涉及多个服务间的协同工作，由于服务间都是跨进城的调用通信，一个业务功能的完成涉及的服务调用链条可能较长，这就涉及到服务间需遵循一些规则来确保协作的可靠性和可用性。我们采用的原则是：长链条的内部服务之间的调用异步化。若一个调用链条中的个别服务变慢或阻塞可能导致整个链条产生雪崩效应，采用异步化来规避调用阻塞等待导致的雪崩情形。

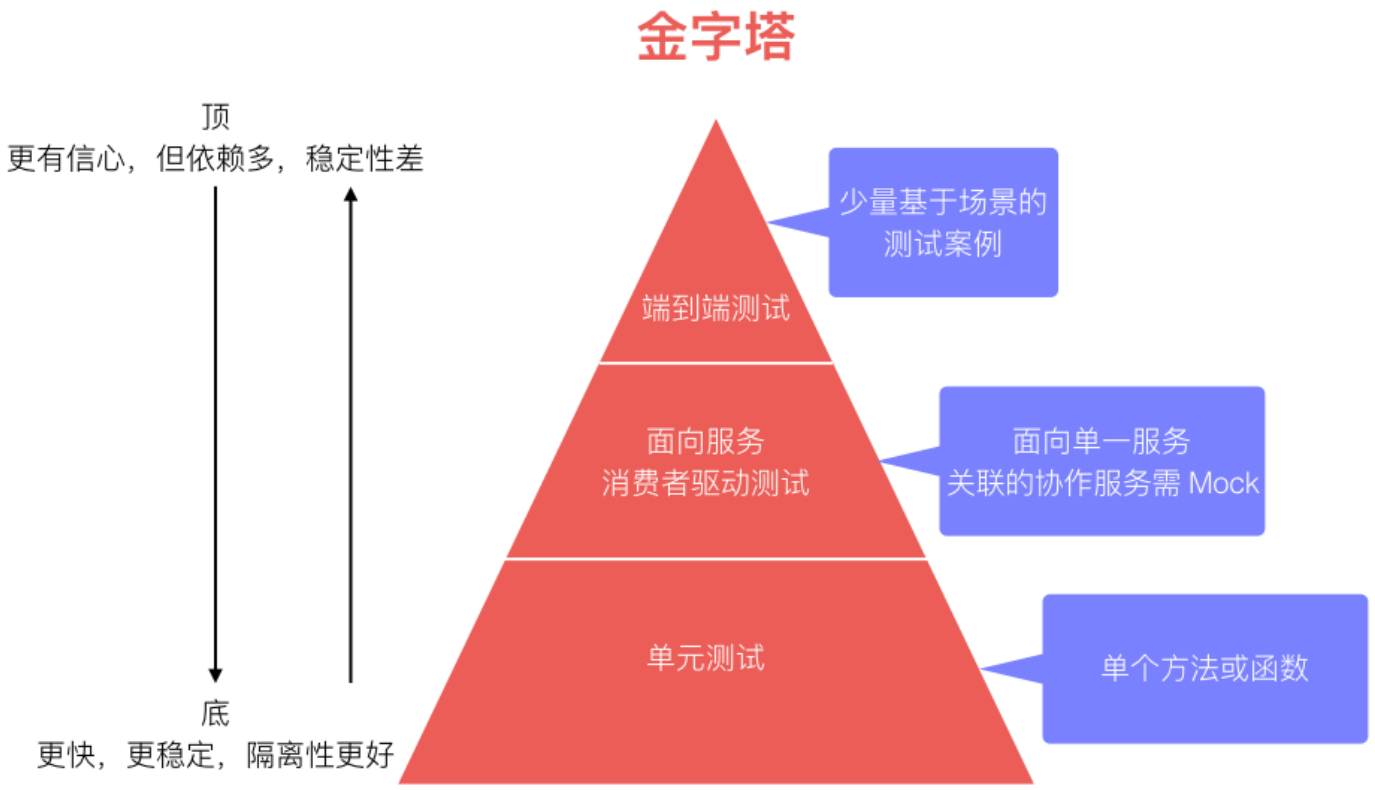


上图展示了咚咚请求调用链的一个异步化过程，若终端的请求是需要同步等待响应结果的（比如 HTTP 请求），只在最外层的接入点持有请求连接，内部服务的传递过程依然是异步化的。

测试

测试从不同的维度可以划分（参考[2]）如下四个象限，四个象限从不同维度视角对测试做了���察和判断，从中可以看出除了体验和探索性测试需要人工介入，其他维度的测试都可以通过自动化来实现，以降低测试人工成本和重复性工作。

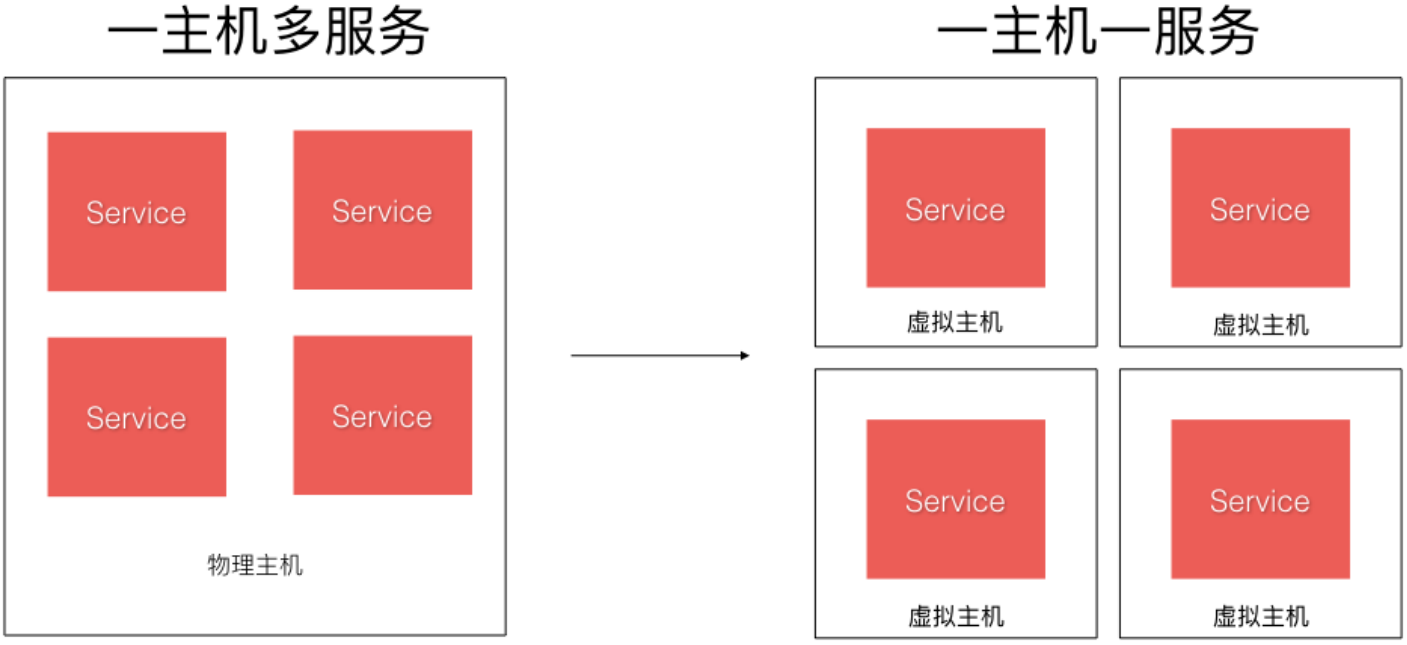
而从测试所处的层次，又可以得到下面这样个一个测试金字塔：



而微服务的测试，服务开发和运营人员专注于做好服务实现层面的单元测试和服务契约层面的接口测试。而面向业务功能的端到端测试，更多是依赖自动化脚本完成。而为了维护好这些自动化测试脚本，也需要保持服务接口和契约的兼容性和稳定性，这些自动化测试脚本也属于服务的消费方之一。

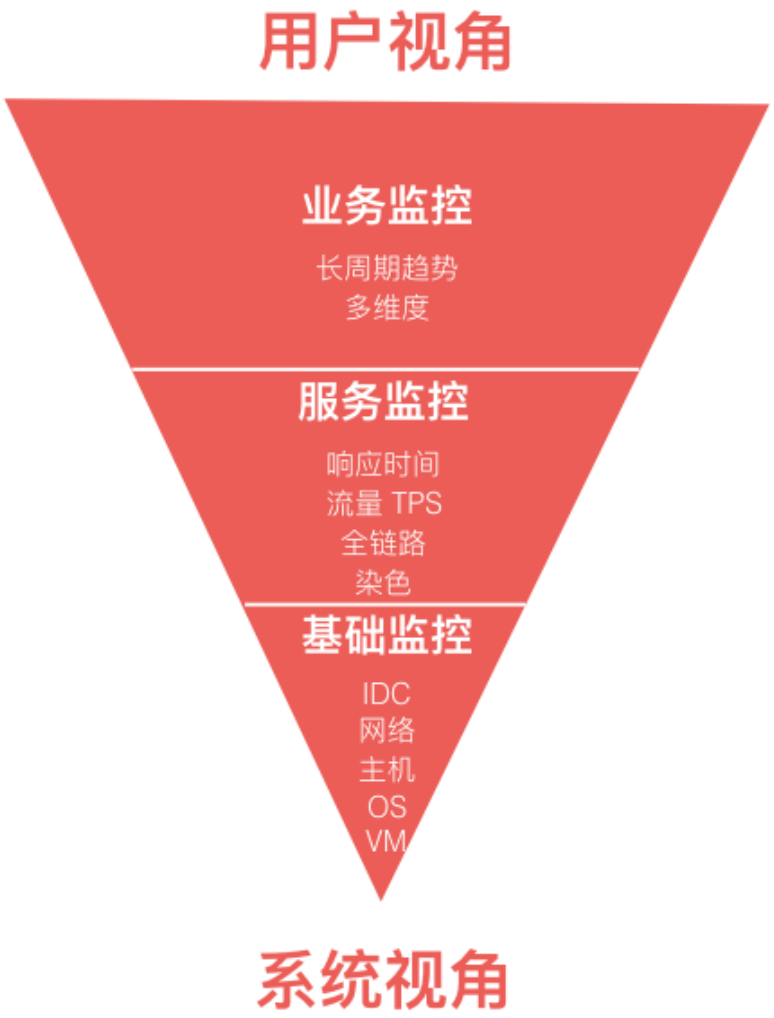
部署

借助于虚拟化或容器等隔离技术，每个服务感觉都是独享资源，不必考虑额外的资源使用冲突。



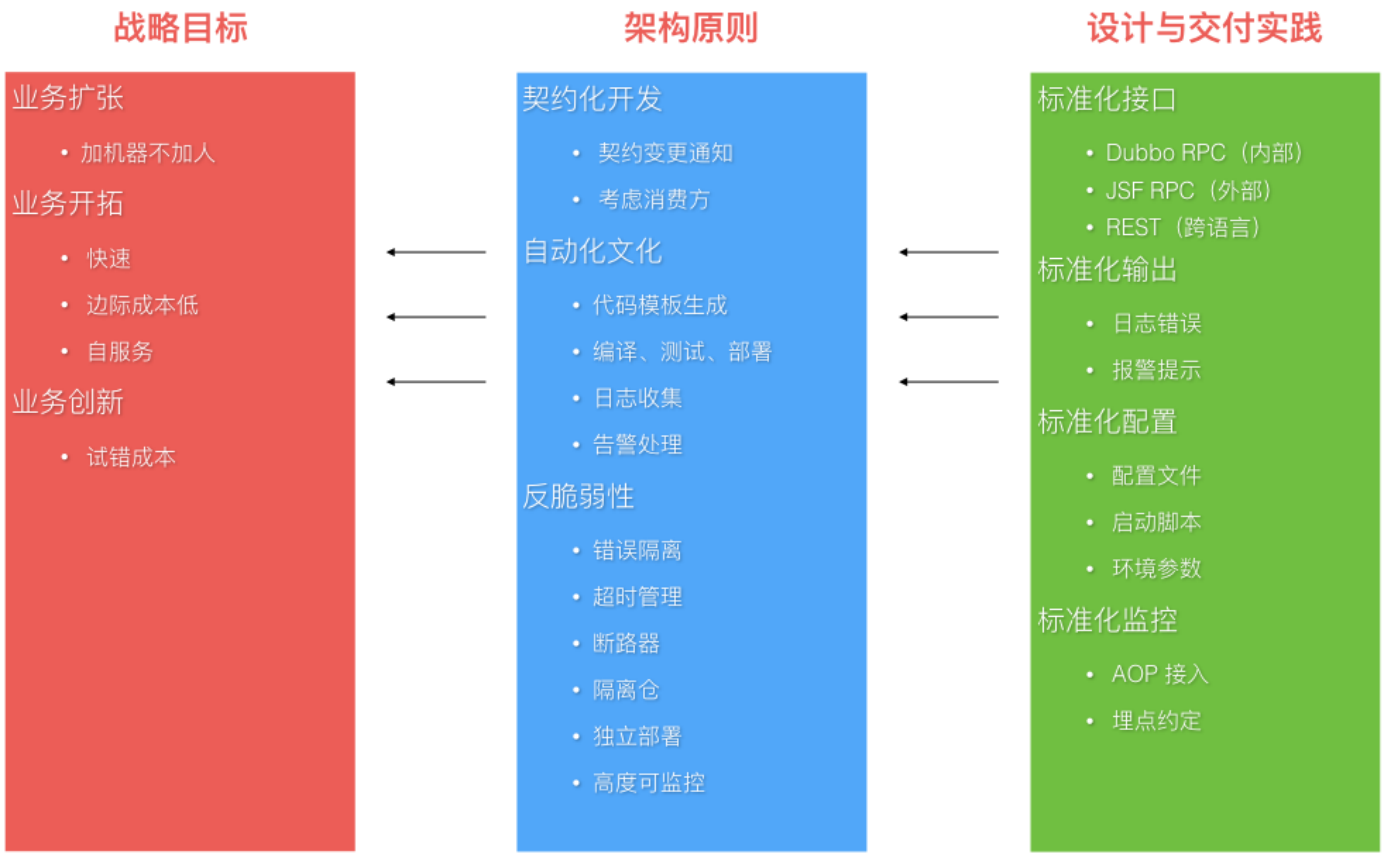
监控

大量松耦合的微服务通过相互协作来完成业务功能的流程处理，在这样一个复杂的生产环境中，出现异常或错误是很难迅速定位的。这就需要一套成体系的监控基础设施，在我们的实践中借助了公司统一的监控基础设施，对监控进行了分层，顶层的监控站在用户视角，底层的监控站在系统视角，形成更完善的反馈链路。



原则

在实施微服务架构的过程中，通过不断的迭代、摸索和修正得到了一些良好的实践模式，对这些良好的实践模式进行抽象提炼总结就得到了架构原则。而对架构原则的把控是为了更好的服务于业务的战略目标。原则的普及带来整体效率的提升和边际成本的下降，以便更有效的支持组织业务战略目标的快速达成。下面这个图结合了微服务架构实施过程中，演示了关于「交付实践」-「架构原则」-「战略目标」之间的一个升维演化和支撑关系。



角色

实施微服务后关于团队人员角色会发生什么样的变化？

按微服务拆分系统后，按照「服务即产品”」的思路，人员角色将发生变化。 普通工程师从仅仅开发功能转变为开发、运营服务，工作性质的转变将带来思路和关注点的变化。 每个服务至少有一个工程师作为负责人，当然能力更强的人可能会负责更多的服务。 大量拆分的微服务带来开发人员交集的减少，对于大规模的团队并行开发好处明显。 而服务负责制对个人能力要求更高，自驱动和自学习能力更强的人会得到更多的成长机会，个人成长路线的发展也打开了空间。

这时团队的构成会变得类似 NBA 球队的组成，工程师的角色类似球员，架构师或技术经理类似教练，而部门经理则是球队经理。球员只管打好球，教练负责球员训练、培养、战术安排和比赛全场把控，经理则掌握着人事权，控制着球员的薪水升迁，招聘到优秀的球员以及想办法带领球队去更受欢迎的比赛上打球。

总结

从接触微服务的概念到今天写下本文正好两年了。本文从微服务的定义出发，追溯它的起源，分析它的特征，然后到实施微服务的前提、维度和原则，最后是实施微服务过程中带来的一些人员角色属性的变化，比较全面的梳理总结微服务架构的各方面。

微服务是一个近年的新概念，但却真不是一个原创性的新东西。它帮助大型应用打散和转移了复杂性，使其可以被更高效的并行解决，但并没有减少任何复杂性，甚至还引入了额外的分布式计算固有的复杂性。我们需有一个清晰的认识，才能更好的认识和实践微服务架构。

参考

[1] Martin Fowler & James Lewis. Microservices. 2014.03
[2] Sam Newman. Building Microservices. 2014.12

[3] Peter Lawrey. Micro-services for performance. 2016.03
[4] Mike Gancarz. The UNIX Philosophy. 1994
[5] Melvin Conway. Conway's law. 1967
[6] Jon Postel. Robustness principle. 1980
[7] Martin Fowler. MicroservicePremium. 2015.05
[8] Martin Fowler. MicroservicePrerequisites. 2014.08
[9] 左手的灵魂. 程序员职业生涯中的 Norris 常数. 2014.06
[10] mindwind. 京东咚咚架构演进. 2015.12

分类: 破阵子·道

« 上一篇：刚挣钱的程序员同学该如何花钱？

» 下一篇：程序员的成长阶梯和级别定义

posted @ 2016-04-24 21:48 mindwind 阅读(612) 评论(3) 编辑 收藏