7种微服务反模式

作者 **Vijay Alagarasan** ，译者 **谢丽** 发布于 2015年9月14日 | 讨论

什么是微服务

流行术语为那些逐步形成的、需要一个好的“标签”来方便交流的概念提供了一个上下文。微服务就是这样的一个新“标签”，它定义了一个领域，这个领域我自己也发现了，并且现在已经使用了一段时间。我慢慢认识到，相关文章和会议所描述的东西，我已经从自己过去几年的个人经历中引申出来。行业和专家对微服务的讨论让Netflix、亚马逊、谷歌等已经成功实现微服务的公司成为了焦点，而我有一些个人经验，可以为成功实现微服务提供一些启发。

以下是任何架构都遵循的三个标准和最常见的业务驱动力：

- 提高敏捷性——及时响应业务需求，促进企业发展
- 提升用户体验——提升客户体验，减少客户流失
- 降低成本——降低增加产品、客户或业务方案的成本

实际上，在日常工作中，所有人都试图这样做。SOA创建了一种业务一致的软件框架，使企业可以达成上述目标。已经出现了几家大型的软件供应商，宣称他们的产品套件可以推动企业实现SOA。

如果没有合适的人员、文化和投入，那么SOA会无法实现业务价值。微服务同SOA并没有根本的不同，它们的目标和目的是相同的，但微服务方法更精炼。事实上，简单来说，微服务就是可扩展的SOA。对于迫切需要由单体实现转变成分布式、去中心化的服务平台并为许多应用程序提供服务的应用程序/系统，微服务提供了这种可能。微服务是独立的，它拥抱敏捷，并允许应用程序随企业的数字化转型进化。微服务的成功取决于服务独立性和灵活度。

我会将微服务定义为“一种实现SOA的方法，它通过构建细粒度的服务支持分布式的、按功能域组织的业务能力”。没有哪种模式是魔法棒或银弹。你应该专门针对一个企业构思和定制模式。企业应该重点解决那些可以为建立自适应平台的架构提供支持的必要事项。

非常不幸的是，一些企业在实现SOA时失败了——因为他们没有充分分析他们的业务能力模型，认为开发Web服务就是SOA，或者从大型供应商那里购买一个SOA套件就实现了SOA，或者他们没有能力阐述SOA同其业务驱动力/目标的一致性。

例子

我经历过的一个例子也许可以说明这一点。在以前的一个岗位上，企业的目标是提高敏捷性、提升用户体验以及降低成本。我们决定构建一个标准的多租户SOA平台。我们选择的方法是开发细粒度的服务，以便我们能够经常修改，并将便于管理的小变更部署到平台。如果今天我们采用了同样的方法，那么我们会称其为微服务架

构。当时还没有这个术语，但它就是有效。

服务根据业务能力建模，初次发布进展顺利。这是些基于JMS的XML同步服务，主要用于为面向代理商、Web和语音通道应用程序的索赔平台提供其所需的业务能力。它为我们提供了敏捷性，使我们可以频繁部署小变更，使我们的应用程序可以完美支持A/B功能。

当需求逐步增加（需求总是会增加）时，由于应用程序同消费者之间集成复杂度很高，所以难以实现方案的快速发布。集成、功能测试、产品发布需要紧密协作。随着业务开始扩展，与初次发布相比，变更多了10多倍，而且，由于交付周期中的大部分任务都是手工的，所以推向市场的时间无法达到企业预期。很快，糟糕的微服务自动化和生命周期管理导致了“交付熵（delivery entropy）”，我们的目标一个也实现不了了。

经验教训——不要做这些事，而是……做其它事

这促使我想要分享一些经验教训，它们是我微服务之旅的一部分，希望你在踏上微服务旅途的时候能够时刻留意这些事项。

1）“内聚混乱（Cohesion Chaos）”

我们开发了一个服务，用于获取客户信息。按照设计，该服务可以获取客户策略信息、个人资料和他们报名参加的计划。过了一段时间，该服务所做的事情开始超出获取客户信息。随着新需求的到来，这个服务经历了频繁的变更和部署。它已经无法扩展和满足可用性要求，变成了众所周知的“大泥球”。它是怎么走到这一步的？首先，我们没有针对功能上相互隔离的问题采取治理措施。如果一个有影响力的服务消费者要求将不相关的逻辑添加到这个服务中，理由是减少往返，那么毫无疑问，那个功能就遭到了破坏。也许网关或BPM层可以避免这种情况的出现，但我们没有时间那样做……仅有匆匆构建另一个业务功能点的时间。

预防措施是抑制与服务不相关的业务功能。服务必须清楚地与业务能力保持一致，不应该试图做一些超出范围的事。功能隔离问题对架构治理而言至关重要，否则，它会破坏敏捷性、性能和可扩展性，最后创建了一个松耦合的架构，导致交付熵和内聚混乱。

2）不重视自动化

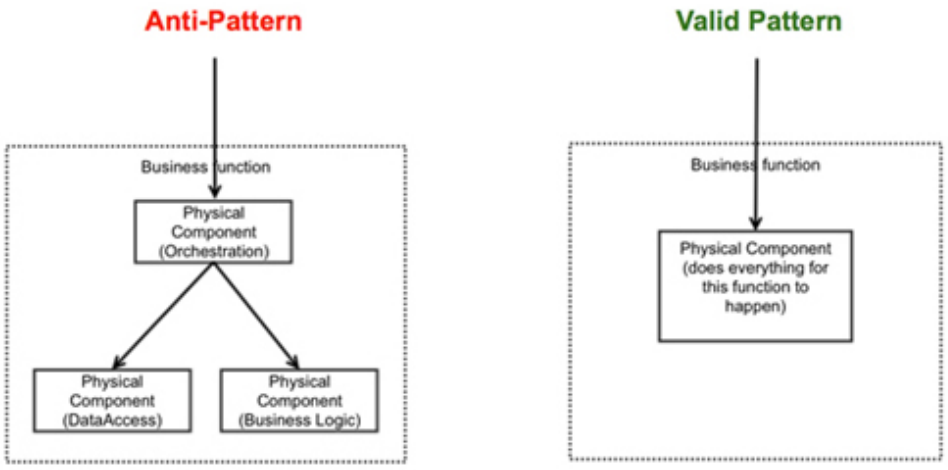
我们没有服务自动部署和运维监控策略（运行时QoS指标）。这明显增加了运维成本和人为部署错误。有几次生产部署因为配置错误导致了宕机。服务总是以HA模式部署，因此容器数量是全部服务总数的3倍。运维团队无法人工处理每个服务的配置。一段时间之后，运维团队开始抱怨架构低效，因为他们无法处理日益增加的容器数量。

什么疫苗能预防这种情况？处方包含多种材料。如果你还没有这样做的话，持续部署是一项必须的投资，也是每个企业都应该追求的一种文化变革。至少，如果没有自动测试和部署方法，就不要使用微服务。微服务的目标是驱动敏捷，为我们提供所需的变更速度；质量保证需要每个服务都有自动的单元、功能、安全和性能测试。当我们开发的服务需要与不受我们控制的服务集成时，服务可视化是另一个功能强大的概念。

3）服务架构分层

对于SOA，人们常犯的一个错误是误解了如何实现服务重用性。团队主要关注技术层面的内聚，而不是功能相关的重用。例如，多个服务充当数据访问层（ORM），将表暴露为服务；他们认为该服务有很高的重用性。这样，他们就创建了一个由横向团队管理的人造物理层，导致交付依赖。创建的任何服务都应该是高度自治的——就是彼此独立。

创建多个技术上的服务物理层只会导致交付复杂、运行低效。最后，我们有包装器服务、编排服务、业务服务和数据服务。这些服务模型均是服务于技术上关切的问题。每个服务层都会形成一个管理团队，最终的结果是，业务逻辑无计划的扩展，功能没有单独的所有者，效率低下，团队之间总是相互指责。



同一个服务内实现逻辑隔离很好，不过不应该有任何过程调用。尽力将每个服务视为一个原子业务实体，它必须实现与所需业务功能有关的一切。与分层服务相比，自包含的服务有更高的自治性和可扩展性。最好能重写可以跨多个服务使用的公共代码，在保持一定的自治水平的前提下，这是一种很好的权衡。关键是不要根据技术上关切的问题隔离服务，一定要根据业务功能来隔离。容器化概念的兴起就是因为这种特性。

4）依赖客户签字

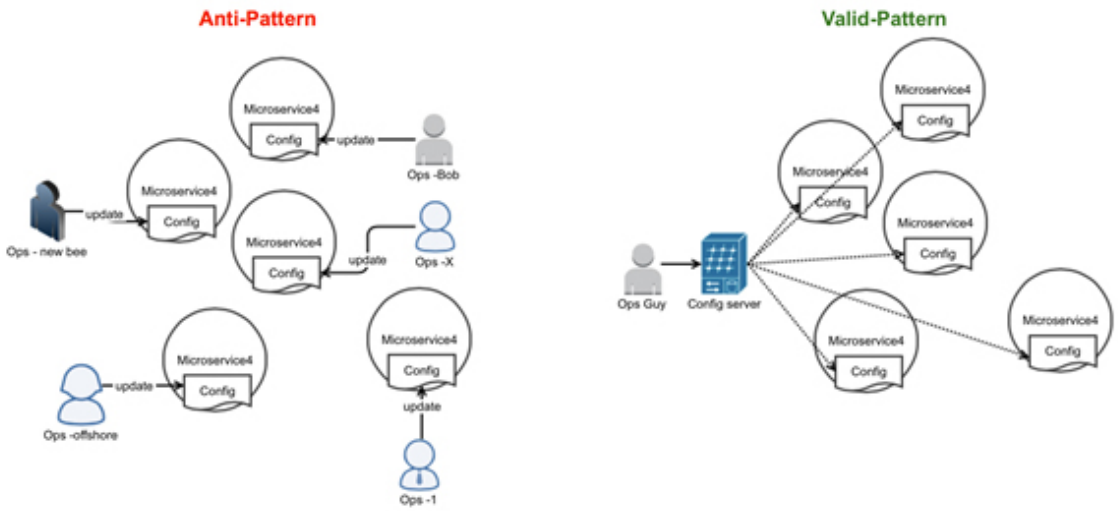
我们有一个供多个应用程序消费的服务，它有三种不同的消费通道，包括代理、Web和语音。代理通道是最根本的，因此，服务在进入生产环境前必须获得他们的签字。这会延迟语音和Web应用程序的生产发布。是什么将那三个通道紧紧地绑在了一起？

当涉及到特定于通道的功能时，服务就不是松耦合了。要让服务有独立性。交付的每个服务都必须有一个测试套件，该套件应该涵盖所有的服务功能、安全、性能、错误处理以及针对每一个现有消费者和未来消费者的消费驱动测试。这必须作为构建自动化回归测试通道的一部分。

5）手动配置管理

当我们开始创建更多的服务（并且由于缺少服务生命周期治理而导致了无计划的扩展），服务的配置管理失控了。大部分生产部署不顺利的情况都是由于配置错误，如错误的密码、URL和值。手动管理配置信息越来越困难。要是我们将应用程序配置管理工具作为PaaS或CD的一部分……但我们没有。

（点击查看大图）

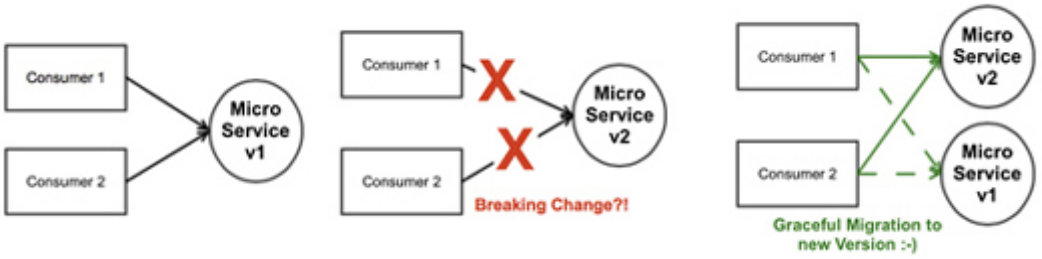


6）未使用版本控制

我们天真地以为，一个服务只会需要一个版本。然后，为了适应多样的消费者和频繁的变更，我们开始增加大、小版本。最后，由于服务依赖客户签字，所以每个版本都不得不是个大版本。结果，容器数量增长得非常快，管理它们成了一个巨大的痛苦。缺少运行时治理是导致这个问题的另一个原因。部分企业愚蠢地试图避免版本控制。如果变更不可避免，服务就需要有良好的架构设计。制定一种策略，管理向前兼容的服务变更，使消费者可以轻松升级。否则，消费者会紧紧地绑定到一个服务版本，当服务变更时，就会破坏这种绑定关系。

在微服务的世界里，可以预见，复杂度会随着服务数量的增加而增加。制定一个版本控制策略可以使消费者轻松迁移，并且服务提供者可以透明地部署变更而不影响其他任何人。限制生产环境中并行的大版本数量，并治理它们。

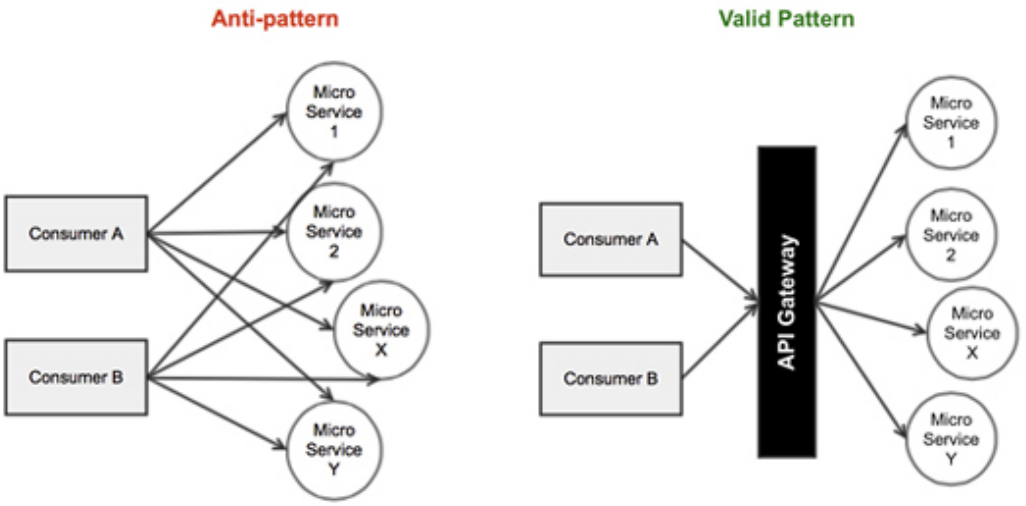
（点击查看大图）



7）在每个服务中构建网关

我们没有一个API网关，也没有运行时治理（我们不知道谁在什么时间以什么速率消费什么服务）。我们开始在每个服务中实现终端用户身份验证、限流、编排、转换和路由等操作。这增加了每个服务的复杂度，而且失去了服务之间实现的一致性，导致我们不知道谁在哪里实现了什么功能。此外，我们构建的部分服务是为了满足一个特定消费者的非功能需求，而不是其它消费者的。如果我们有一个网关，并运用一些数据过滤和增强模式，那原本是可以做到的。要真是这样就好了。

在API管理方案上投入精力，集中化部分非功能性问题的管理和监控，这同时也会减少消费者管理若干微服务配置的负担。API网关可以编排跨功能的微服务，可以减少Web应用程序的请求往返次数。



小结

微服务的目标是解决这三个最常见的问题，即提升客户体验、非常敏捷地响应新需求和以更细粒度的服务交付业务功能降低成本。微服务不是银弹，它需要一个合乎经验规律的平台，使以敏捷方式交付高质量服务成为可能。从他人（我）的错误中吸取教训，在架构设计和交付过程中避免上述模式。这是我们可以谈论容器化、云实施等之前的关键一步。我希望这篇文章能够提供一些信息，让你可以针对自己的企业进行思考，争取在将它们引入到你的架构之前解决这些反模式。上述各项大部分都会推动组织的文化变革，仅仅靠你自己是无法完成的，务必要同主管及高层领导合作。

关于作者

Vijay Alagarasan是Asurion企业架构和策略部门的首席架构师。这是一家全球领先的技术支持和安全公司。他主要从事企业级服务、API和集成（A2A、B2B和B2C）。他定义并管理这些领域的原则、模式、技术选择和标准。他通过制订完整的解决方案或开发原型来提供参考实现，使交付组织可以比平时更快地学习新概念和技术。他还评估和引入新技术产品，使企业更快地实现战略目标，并在市场中处于领先地位。他最初是一名Java/J2EE应用程序开发人员，然后是作为一名EAI开发人员，之后不久就成为了 SOA实施者/架构师。他的工作主要涉及TIBCO的产品、Java、Weblogic、JMS及一堆其它技术（LinkedIn用户资料）。他能胜任多种角色，如软件开发人员、团队负责人、交付经理、集成架构师、解决方案架构师和领域架构师。近年来，他一直致力于Web API（像Layer 7、Apigee、AWS API Gateway、Azure API Management这样的API管理工具）、云技术（AWS、Azure）、容器、服务可视化和图数据库等方面的工作。

查看英文原文：Seven Microservices Anti-patterns