微服务

作者 YYGCui 日期 2015-07-22

Categories: 技术积累 (/categories/技术积累/) Tags: microservices (/tags/microservices/) 微服务 (/tags/微服务/)

翻译自Martin Fowler的microservices (http://martinfowler.com/articles/microservices.html), 翻译于2015年7月22日。翻译尽量贴近原文，减少意译带入的个人观点，如有不当之处，请指正。

在过去几年中，“微服务架构”这一术语如雨后春笋般涌现出来，它描述了一种将软件应用程序设计为一组可独立部署的服务的特定方式。虽然这种架构风格没有明确的定义，但在组织、业务能力上有一些共同的特征：自动化部署，端点智能化，语言和数据的去中心化控制。

“微服务” - 软件架构拥挤大街上的有一个新术语。虽然我们自然的倾向是轻蔑的一瞥将它一带而过，然而我们发现这一术语描述了一种越来越吸引人的软件系统风格。我们已看到，在过去的几年中有许多项目使用了这种风格，并且到目前为止结果都还不错，以致于这已变成了我们同事在构建企业级应用程序时默认使用的架构风格。然而，遗憾的是并没有太多的信息来概述什么是微服务风格以及怎样用这种风格。

简单来说，微服务架构风格[1]是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通信采用轻量级通信机制(通常用HTTP资源API)。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术。

与单体风格作对比有助于开始解释微服务风格：单体应用程序被构建为单一单元。企业级应用程序通常由三部分组成：客户端侧用户接口(由运行于开发机上的浏览器里的HTML页面和Javascript组成)，数据库(由插入到通用关系型数据库管理系统中的许多数据表格组成)，服务端应用程序。服务端应用程序处理HTTP请求，执行领域逻辑，从数据库中检索、更新数据，选择、填充将要发送到浏览器的HTTP视图。服务端应用程序是一个单一的逻辑可执行单体[2]。系统的任何改变都将牵涉到重新构建和部署服务端的一个新版本。

这样的单体服务器是构建这样一个系统最自然的方式。处理请求的所有逻辑都运行在一个单一进程中，允许你使用编程语言的基本特性将应用程序划分类、函数和命名空间。你认真的在开发机上运行测试应用程序，并使用部署管道来保证变更已被正确地测试并部署到生产环境中。该单体的水平扩展可以通过在负载均衡器后面运行多个实例来实现。

单体应用程序可以是成功的，但人们日益对他们感到挫败，尤其是随着更多的应用程序被部署在云上。变更周期被捆绑在一起 —— 即使只变更应用程序的一部分，也需要重新构建并部署整个单体。长此以往，通常将很难保持一个良好的模块架构，这使得很难变更只发生在需要变更的模块内。程序扩展要求进行整个应用程序的扩展而不是需要更多资源的应用程序部分的扩展。

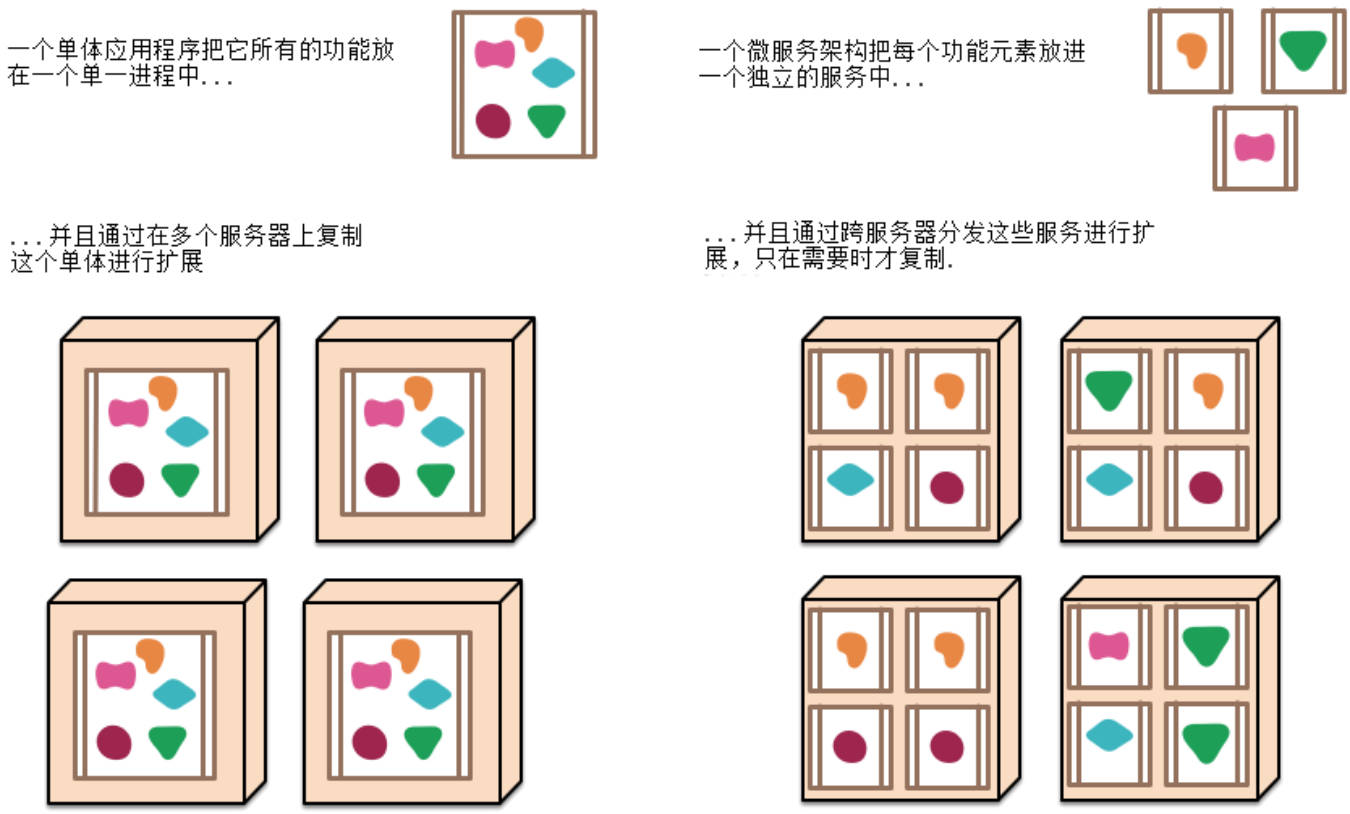


图1: 单体和微服务

这些挫败导向了微服务架构风格：构建应用程序为服务套件。除了服务是可独立部署、可独立扩展的之外，每个服务都提供一个固定的模块边界。甚至允许不同的服务用不同的的语言开发，由不同的团队管理。

我们不会声称微服务风格是新颖的、创新的，其本质至少可以回溯到Unix的设计哲学。但我们的确认为没有足够的人仔细考虑微服务架构，并且如果使用它很多软件实现将会更好。

微服务架构的特征

我们无法给出微服务架构风格的一个正式定义，但我们可以尝试去描述我们看到的符合该架构的一些共性。就概述共性的任何定义来说，并非所有的微服务架构风格都有这些共性，但我们期望大多数微服务架构风格展现出大多数特性。虽然本文作者一直是这个相当松散的社区的活跃用户，我们的目的是试图描述我们工作中和我们知道的一些团队的相似努力中的所见所闻。特别是我们不会制定一些可遵守的定义。

通过服务组件化

只要我们一直从事软件行业，一个愿望就是通过把组件插在一起构建系统，如同我们看到的现实世界中事物的构造方式一样。在最近的二十年中，我们看到作为大多数语言平台一部分的公共库的大量汇编工作取得了很大的进展。

当谈到组件时，我们遭遇困难的定义：组件是什么。我们的定义是：组件是一个可独立替换和独立升级的软件单元。

微服务架构将使用库，但组件化软件的主要方式是分解成服务。我们把库定义为链接到程序并使用内存函数调用来调用的组件，而服务是一种进程外的组件，它通过web服务请求或rpc(远程过程调用)机制通信(这和很多面向对象程序中的服务对象的概念是不同的[3]。)

使用服务作为组件而不是使用库的一个主要原因是服务是可独立部署的。如果你有一个应用程序[4]是由单一进程里的多个库组成，任何一个组件的更改都导致必须重新部署整个应用程序。但如果应用程序可分解成多个服务，那么单个服务的变更只需要重新部署该服务即可。当然这也不是绝对的，一些变更将会改变服务接口导致一些协作，但一个好的微服务架构的目的是通过内聚服务边界和按合约演进机制来最小化这些协作。

使用服务作为组件的另一个结果是一个更加明确的组件接口。大多数语言没有一个好的机制来定义一个明确的发布接口 (http://martinfowler.com/bliki/PublishedInterface.html)。通常只有文档和规则来预防客户端打破组件的封装，这导致组件间过于紧耦合。服务通过明确的远程调用机制可以很容易的避免这些。

像这样使用服务确实有一些缺点，远程调用比进程内调用更昂贵，因此远程API被设计成粗粒度，这往往更不便于使用。如果你需要更改组件间的责任分配，当你跨进程边界时，这样的行为动作更难达成。

直观的估计，我们观察到服务与运行时进程一一映射，但这仅仅是直观的估计而已。一个服务可能由多进程组成，这些进程总是被一起开发和部署，比如只被这个服务使用的应用进程和数据库。

围绕业务能力组织

当想要把大型应用程序拆分成部件时，通常管理层聚焦在技术层面，导致UI团队、服务侧逻辑团队、数据库团队的划分。当团队按这些技术线路划分时，即使是简单的更改也会导致跨团队的时间和预算审批。一个聪明的团队将围绕这些优化，两害取其轻 - 只把业务逻辑强制放在它们会访问的应用程序中。换句话说，逻辑无处不在。这是Conway法则[5]在起作用的一个例子。

> 任何设计系统(广泛定义的)的组织将产生一种设计，他的结构就是该组织的通信结构。
> **-- Melvyn Conway** — *1967*

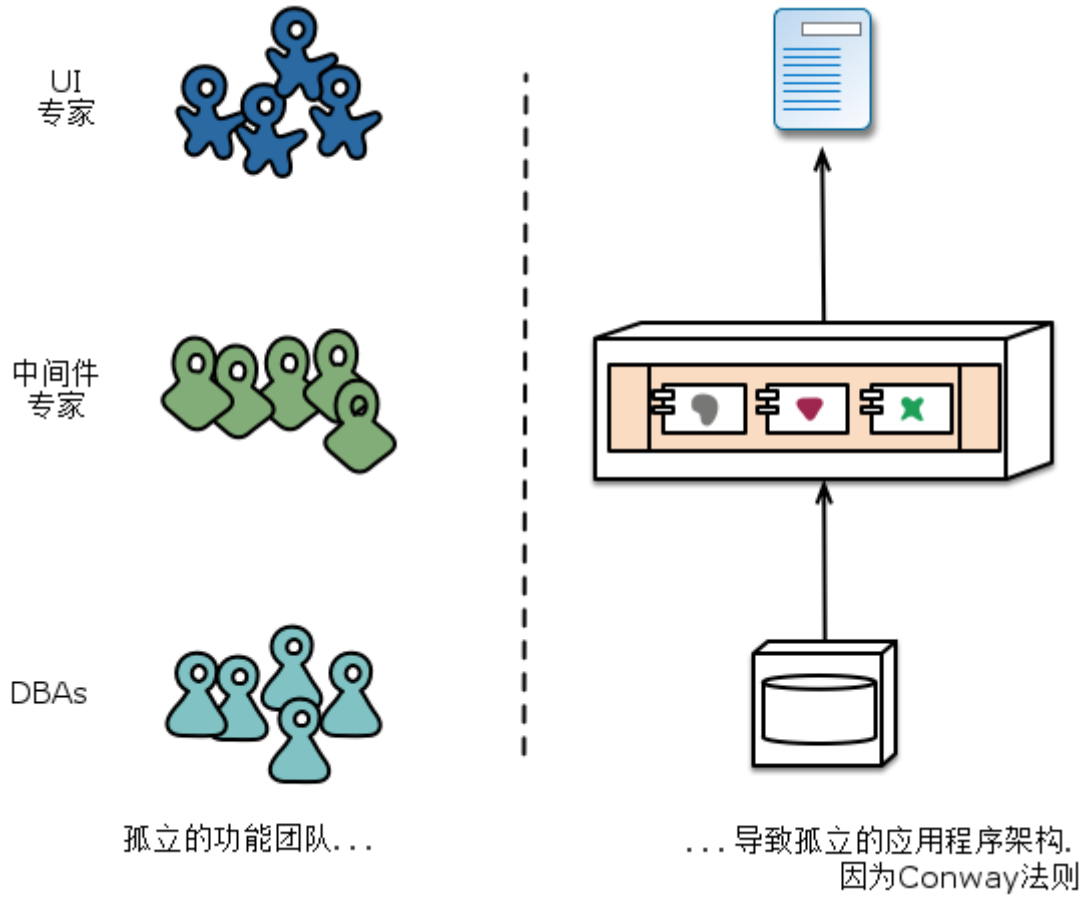


图2: Conway法则在起作用

微服务采用不同的分割方法，划分成围绕业务能力组织的服务。这些服务采取该业务领域软件的宽栈实现，包括用户接口、持久化存储和任何外部协作。因此，团队都是跨职能的，包括开发需要的全方位技能：用户体验、数据库、项目管理。

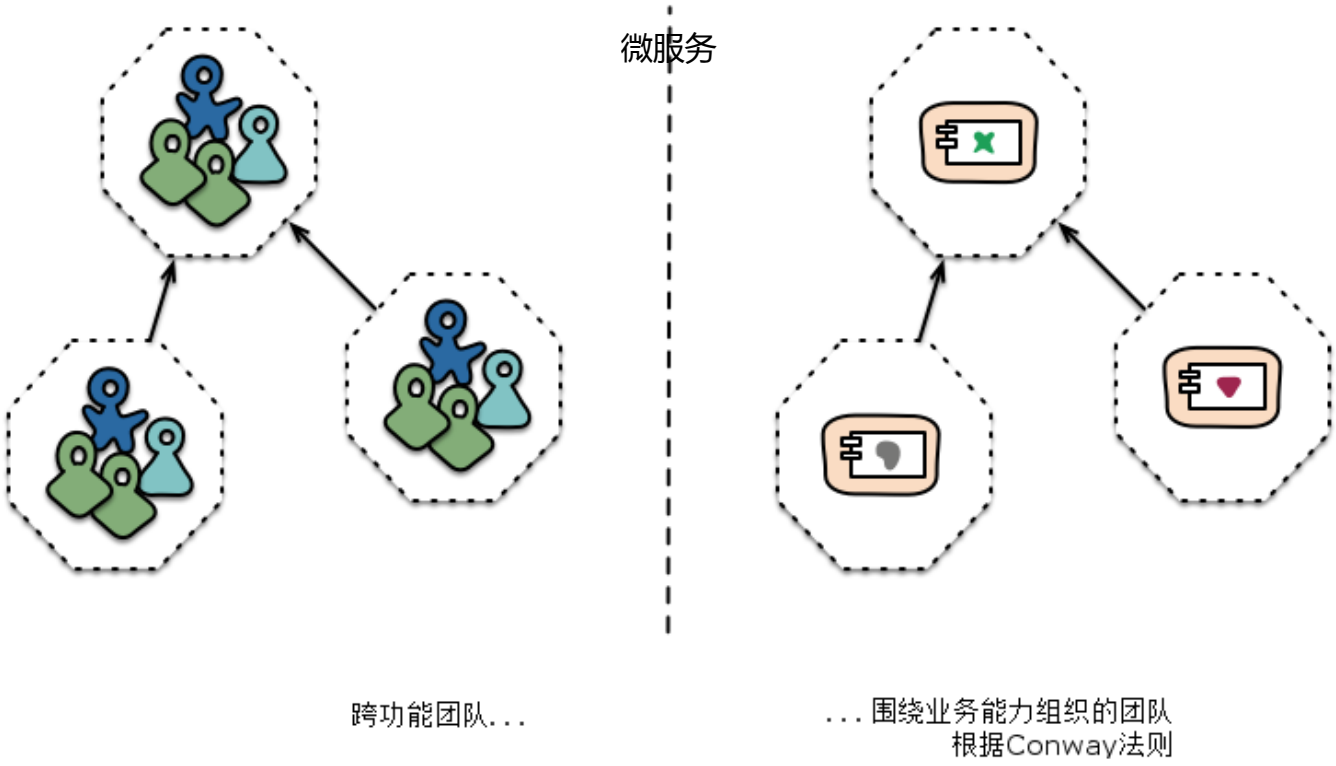


图3: 团队边界增强的服务边界

www.comparethemarket.com (www.comparethemarket.com)是按这种方式组织的一个公司。跨职能团队负责创建和运营产品，产品被划分成若干个体服务，这些服务通过消息总线通信。

大型单体应用程序也总是可以围绕业务能力来模块化，虽然这不是常见的情况。当然，我们将敦促创建单体应用程序的大型团队将团队本身按业务线拆分。我们看到这种情况的主要问题是他们趋向于围绕太多的上下文进行组织。如果单体横跨了多个模块边界，对团队个体成员来说，很难把它们装进他们的短期记忆里。另外，我们看到模块化的路线需要大量的规则来强制实施。服务组件所要求的更加明确的分离，使得它更容易保持团队边界清晰。

侧边栏：微服务有多大？

虽然，“微服务”已成为这种架构风格的代称，这个名字确实会导致不幸的聚焦于服务的大小，并为“微”由什么组成争论不休。在与微服务实践者的对话中，我们发现有各种大小的服务。最大的服务报道遵循亚马逊两匹萨团队(也就是，整个团队吃两个披萨就吃饱了)的理念，这意味着团队不超过12个人。在更小的规模大小上，我们看到这样的安排，6人团队将支持6个服务。
这导致这样一个问题，在服务每12个人和服务每1个人的大小范围内，是否有足够打的不同使他们不能被集中在同一微服务标签下。目前，我们认为最好把它们组合在一起。但随着深入探索这种风格，我们一定有可能改变我们的看法。

是产品不是项目

我们看到大多数应用程序开发工作使用一个项目模式：目标是交付将要完成的一些软件。完成后的软件被交接给维护组织，然后它的构建团队就解散了。

微服务支持者倾向于避免这种模式，而是认为一个团队应该负责产品的整个生命周期。对此一个共同的启示是亚马逊的理念 “you build, you run it” (https://queue.acm.org/detail.cfm?id=1142065)，开发团队负责软件的整个产品周期。这使开发者经常接触他们的软件在生产环境如何工作，并增加与他们的用户联系，因为他们必须承担至少部分的支持工作。

产品思想与业务能力紧紧联系在一起。要持续关注软件如何帮助用户提升业务能力，而不是把软件看成是将要完成的一组功能。

没有理由说为什么同样的方法不能用在单体应用程序上，但服务的粒度更小，使得它更容易在服务开发者和用户之间建立个人关系。

智能端点和哑管道

当在不同进程间创建通信结构时，我们已经看到了很多的产品和方法，把显著的智慧强压进通信机制本身。一个很好的例子就是企业服务总线(ESB)，在ESB产品中通常为消息路由、编排(choreography)、转化和应用业务规则引入先进的设施。

微服务社区主张另一种方法：智能端点和哑管道。基于微服务构建的应用程序的目标是尽可能的解耦和尽可能的内聚 - 他们拥有自己的领域逻辑，他们的行为更像经典UNIX理念中的过滤器 - 接收请求，应用适当的逻辑并产生响应。使用简单的REST风格的协议来编排他们，而不是使用像WS-Choreography或者BPEL或者通过中心工具编制(orchestration)等复杂的协议。

最常用的两种协议是使用资源API的HTTP请求-响应和轻量级消息传送[6]。对第一种协议最好的表述是

> 本身就是web，而不是隐藏在web的后面。
>
> *-- — Ian Robinson (http://www.amazon.com/gp/product/0596805829?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0596805829)*

微服务团队使用的规则和协议，正是构建万维网的规则和协议(在更大程度上，是UNIX的)。从开发者和运营人员的角度讲，通常使用的资源可以很容易的缓存。

第二种常用方法是在轻量级消息总线上传递消息。选择的基础设施是典型的哑的(哑在这里只充当消息路由器) - 像RabbitMQ或ZeroMQ这样简单的实现仅仅提供一个可靠的异步交换结构 - 在服务里，智能仍旧存活于端点中，生产和消费消息。

单体应用中，组件都在同一进程内执行，它们之间通过方法调用或函数调用通信。把单体变成微服务最大的问题在于通信模式的改变。一种幼稚的转换是从内存方法调用转变成RPC，这导致频繁通信且性能不好。相反，你需要用粗粒度通信代替细粒度通信。

去中心化治理

集中治理的一个后果是单一技术平台的标准化发展趋势。经验表明，这种方法正在收缩 - 不是每个问题都是钉子，不是每个问题都是锤子。我们更喜欢使用正确的工具来完成工作，而单体应用程序在一定程度上可以利用语言的优势，这是不常见的。

把单体的组件分裂成服务，在构建这些服务时可以有自己的选择。你想使用Node.js开发一个简单的报告页面？去吧。用C++实现一个特别粗糙的近乎实时的组件？好极了。你想换用一个更适合组件读操作数据的不同风格的数据库？我们有技术来重建它。

当然，仅仅因为你可以做些什么，而不意味着你应该这样做 - 但用这种方式划分系统意味着你可以选择。

团队在构建微服务时也更喜欢用不同的方法来达标。他们更喜欢生产有用的工具这种想法，而不是写在纸上的标准，这样其他开发者可以用这些工具解决他们所面临的相似的问题。有时，这些工具通常在实施中收获并与更广泛的群体共享，但不完全使用一个内部开源模型。现在git和github已经成为事实上的版本控制系统的选择，在内部开放源代码的实践也正变得越来越常见。

侧边栏：微服务和SOA

当我们谈论微服务时，一个常见问题是它是否仅仅是十年前我们看到的面向服务的架构(SOA)。这一点是有可取之处的，因为微服务风格和SOA赞同的某些主张十分相似。然而，问题是SOA意味着很多不同的东西 (http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html)，而大多数时候，我们遇到的所谓的SOA和这里我们描述的风格明显不同，这种不同通常由于SOA专注于用于集成单体应用的ESB。

特别是我们已看到太多的搞砸的服务导向的实现，从趋向于隐藏ESB中的复杂性[7]，到花费数百万并不产生任何价值的失败的多年举措，到积极抑制变化的集中治理模型，这有时很难看到过去的这些问题。

当然，微服务社区用到的许多技术从开发者在大型组织机构整合服务的经验中成长。Tolerant Reader (http://martinfowler.com/bliki/TolerantReader.html)模式就是这样的一个例子。使用简单协议是衍生自这些经验的另一个方法，使用网络的努力已做出远离中央标准的反应，坦率地说，中心标准已达到令人叹为观止 (http://wiki.apache.org/ws/WebServiceSpecifications)的复杂性。(任何时候，你需要一个本体来管理你的本体，你知道你深陷困境。)

SOA的这种常见表现使得一些微服务倡导者完全拒绝SOA标签，尽管其他人认为微服务是SOA的一种形式[8]，也许服务导向做得对。无论哪种方式，事实上，SOA意味着如此不同的事情，这意味着有一个术语来更清晰地定义这种架构风格是有价值的。

Netflix是遵守这一理念的很好的例子。尤其是，以库的形式分享有用的且经过市场检验的代码，这激励其他开发者用类似的方式解决相似的问题，同时还为采用不同方法敞开了大门。共享库倾向于聚焦在数据存储、进程间通信和我们接下来要深入讨论的基础设施自动化的共性问题。

对为服务社区来说，开销特别缺乏吸引力。这并不是说社区不重视服务合约。恰恰相反，因为他们有更多的合约。只是他们正在寻找不同的方式来管理这些合约。像Tolerant Reader (http://martinfowler.com/bliki/TolerantReader.html)和消费者驱动的契约(Consumer-Driven Contracts (http://martinfowler.com/articles/consumerDrivenContracts.html))这样的模式通常被用于微服务。
这些援助服务合约在独立进化。执行消费者驱动的合约作为构建的一部分，增加了信心并对服务是否在运作提供了更快的反馈。事实上，我们知道澳大利亚的一个团队用消费者驱动的合约这种模式来驱动新业务的构建。他们使用简单的工具定义服务的合约。这已变成自动构建的一部分，即使新服务的代码还没写。服务仅在满足合约的时候才被创建出来 - 这是在构建新软件时避免"YAGNI"[9]困境的一个优雅的方法。围绕这些成长起来的技术和工具，通过减少服务间的临时耦合，限制了中心合约管理的需要。

> ### 侧边栏：许多语言，许多选项
>
> JVM作为平台的成长就是在一个共同平台内混合语言的最新例子。几十年来，破壳到高级语言利用高层次抽象的优势已成为一种普遍的做法。如同下拉到机器硬件，用低层次语言写性能敏感的代码一样。然而，很多单体不需要这个级别的性能优化和常见的更高层次的抽象，也不是DSL的。相反，单体通常是单一语言的并趋向于限制使用的技术的数量[10]。

也许去中心化治理的最高境界就是亚马逊广为流传的build it/run it理念。团队要对他们构建的软件的各方面负责，包括7*24小时的运营。这一级别的责任下放绝对是不规范的，但我们看到越来越多的公司让开发团队负起更多责任。Netflix是采用这一理念的另一家公司[11]。每天凌晨3点被传呼机叫醒无疑是一个强有力的激励，使你在写代码时关注质量。这是关于尽可能远离传统的集中治理模式的一些想法。

去中心化数据管理

数据管理的去中心化有许多不同的呈现方式。在最抽象的层面上，这意味着使系统间存在差异的世界概念模型。在整合一个大型企业时，客户的销售视图将不同于支持视图，这是一个常见的问题。客户的销售视图中的一些事情可能不会出现在支持视图中。它们确实可能有不同的属性和(更坏的)共同属性，这些共同属性在语义上有微妙的不同。

这个问题常见于应用程序之间，但也可能发生在应用程序内部，尤其当应用程序被划分成分离的组件时。一个有用的思维方式是有界上下文(Bounded Context (http://martinfowler.com/bliki/BoundedContext.html))内的领域驱动设计(Domain-Driven

Design, DDD)理念。DDD把一个复杂域划分成多个有界的上下文，并且映射出它们之间的关系。这个过程对单体架构和微服务架构都是有用的，但在服务和上下文边界间有天然的相关性，边界有助于澄清和加强分离，就像业务能力部分描述的那样。

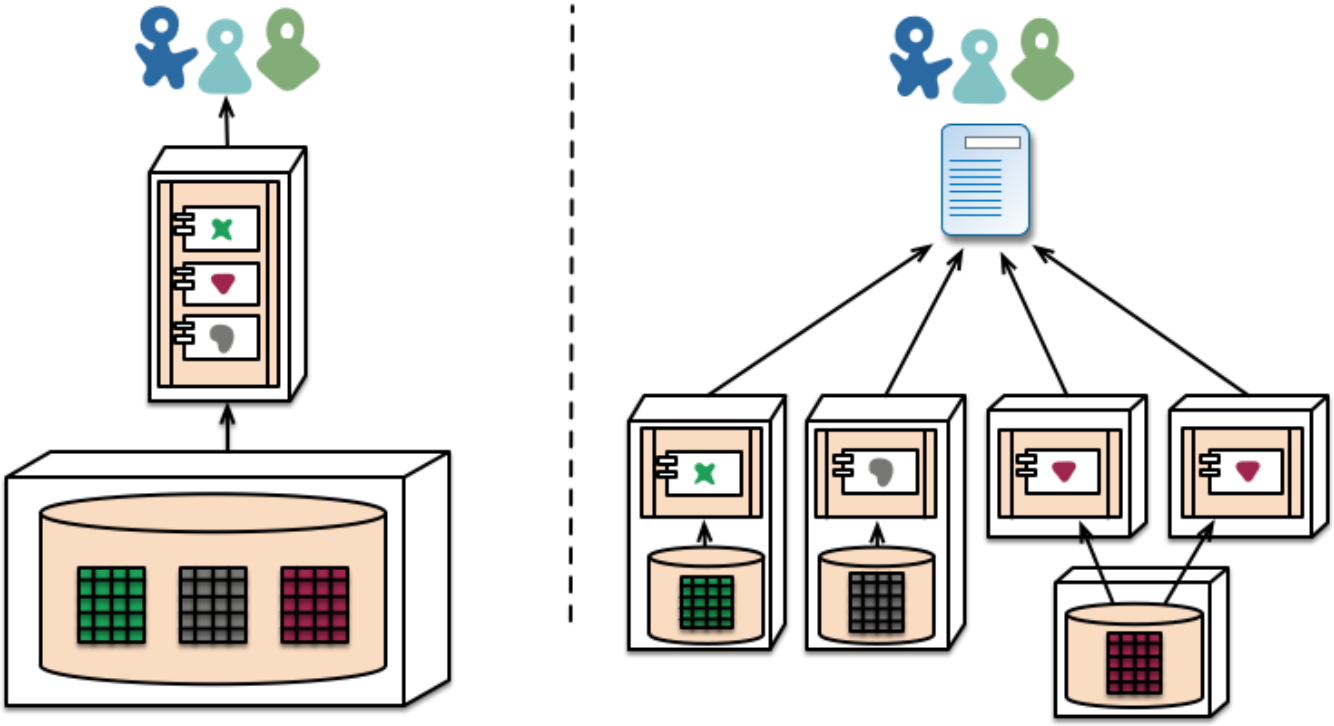
侧边栏：久经考验的标准和执行标准

这有一点分裂，微服务团队倾向于避开企业架构组规定的那种严格的执行标准，但又很乐意使用甚至传教开放标准，比如HTTP、ATOM和其他威格士。

关键的区别是如何定制标准和如何执行。由诸如IETF等组织管理的标准仅当在世界范围内有几个有用的实现时才变成标准，这往往会从成功的开源项目成长起来。

这些标准是远离企业世界的标准。往往被一个几乎没有近期编程经验的或受供应商过度影响的组织开发的。

和概念模型的去中心化决策一样，微服务也去中心化数据存储决策。虽然单体应用程序更喜欢单一的逻辑数据库做持久化存储，但企业往往倾向于一系列应用程序共用一个单一的数据库 - 这些决定是供应商授权许可的商业模式驱动的。微服务更倾向于让每个服务管理自己的数据库，或者同一数据库技术的不同实例，或完全不同的数据库系统 - 这就是所谓的混合持久化(Polyglot Persistence (http://martinfowler.com/bliki/PolyglotPersistence.html))。你可以在单体应用程序中使用混合持久化，但它更常出现在为服务里。



单体 – 单一数据库　　微服务 – 应用程序数据库

对跨微服务的数据来说，去中心化责任对管理升级有影响。处理更新的常用方法是在更新多个资源时使用事务来保证一致性。这个方法通常用在单体中。

像这样使用事务有助于一致性，但会产生显著地临时耦合，这在横跨多个服务时是有问题的。分布式事务是出了名的难以实现，因此微服务架构强调服务间的无事务协作(http://www.eaipatterns.com/ramblings/18_starbucks.html)，对一致性可能只是最后一致性和通过补偿操作处理问题有明确的认知。

对很多开发团队来说，选择用这样的方式管理不一致性是一个新的挑战，但这通常与业务实践相匹配。通常业务处理一定程度的不一致，以快速响应需求，同时有某些类型的逆转过程来处理错误。这种权衡是值得的，只要修复错误的代价小于更大一致性下损失业务的代价。

基础设施自动化

在过去的几年中，基础设施自动化已经发生了巨大的变化，特别是云和AWS的演化已经降低了构建、部署和运维微服务的操作复杂度。

许多用微服务构建的产品或系统是由在持续部署(http://martinfowler.com/bliki/ContinuousDelivery.html)和它的前身持续集成(http://martinfowler.com/articles/continuousIntegration.html)有丰富经验的团队构建的。团队用这种方式构建软件，广泛使用了基础设施自动化。如下面的构建管线图所示：

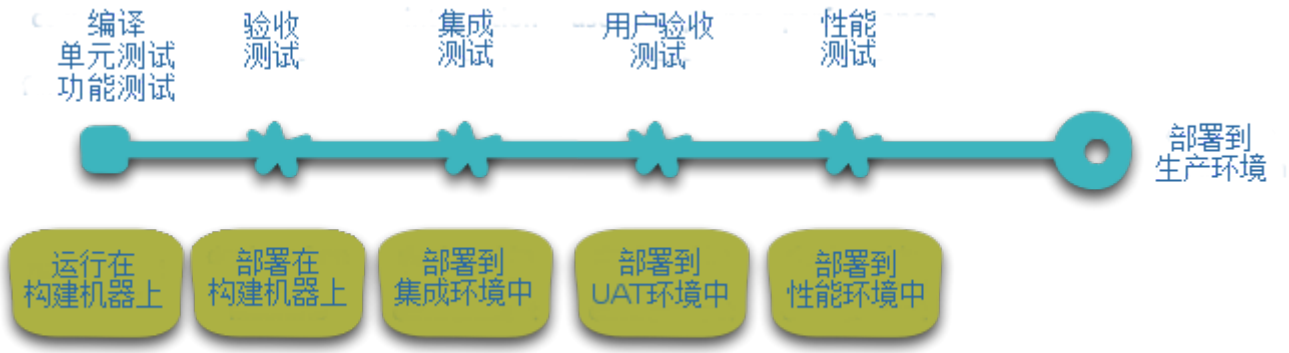


图5: 基础构建管道

因为这不是一篇关于持续交付的文章，我们这里将之光住几个关键特性。我们希望有尽可能多的信心，我们的软件正在工作，所以我们运行大量的**自动化测试**。促进科工作软件沿管道线“向上”意味着我们**自动化部署**到每个新的环境中。

一个单体应用程序可以十分愉快地通过这些环境被构建、测试和推送。事实证明，一旦你为单体投入了自动化生产之路，那么部署更多的应用程序似乎也不会更可怕。请记住，持续部署的目标之一是使部署枯燥，所以无论是一个或三个应用程序，只要它的部署仍然枯燥就没关系[12]。

侧边栏：使它容易做正确的事情 微服务

我们发现，作为持续交付和持续部署的一个后果，增加自动化的一个副作用是创造有用的工具，以帮助开发人员和运营人员。用于创造人工制品、管理代码库、起立(standing up)简单服务或添加标准监控和日志记录的工具现在都是很常见的。web上最好的例子可能是Netflix的开源工具集 (http://netflix.github.io/)，但也有其他我们广泛使用的工具，如Dropwizard (http://dropwizard.codahale.com/)。

我们看到团队使用大量的基础设施自动化的另一个领域是在生产环境中管理微服务时。与我们上面的断言(只要部署是枯燥的)相比，单体和微服务没有太大的差别，各运营场景可以明显不同。

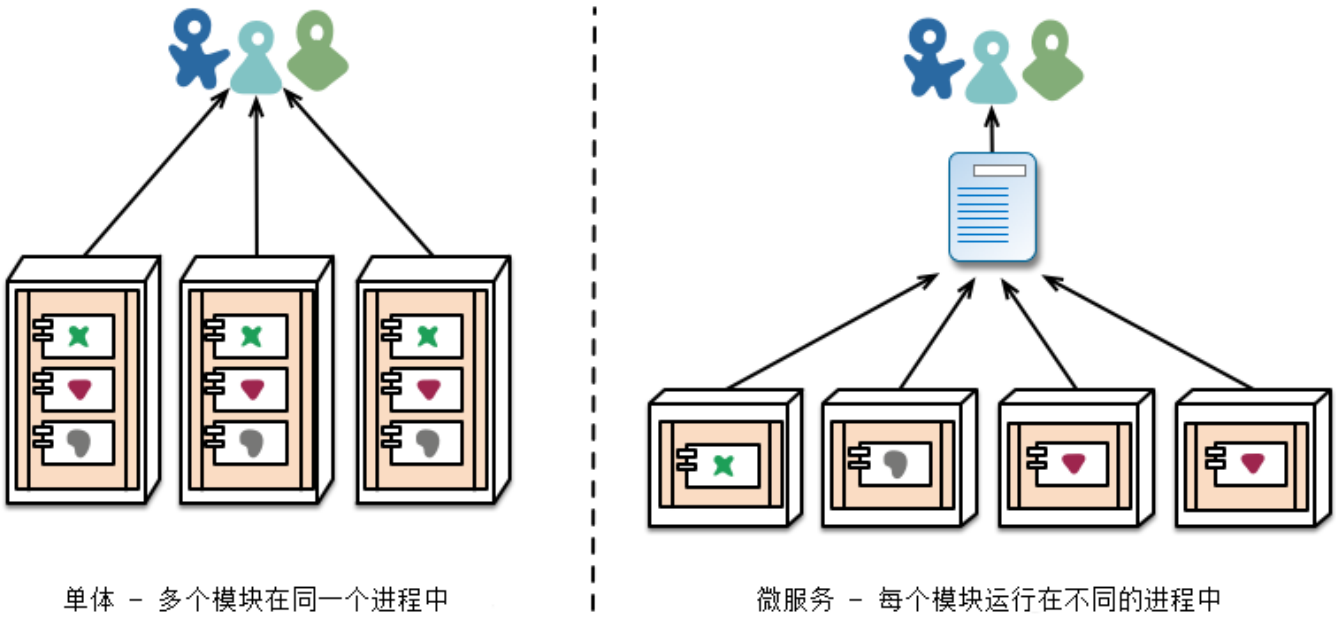


图6: 模块部署常常不同

为失效设计

使用服务作为组件的一个结果是，应用程序需要被设计成能够容忍服务失效。任何服务调用都可能因为供应者不可用而失败，客户端必须尽可能优雅的应对这种失败。与单体应用设计相比这是一个劣势，因为它引入额外的复杂性来处理它。结果是，微服务团队不断反思服务失效如何影响用户体验。Netflix的Simian Army (https://github.com/Netflix/SimianArmy)在工作日诱导服务甚至是数据中心故障来测试应用程序的弹性和监测。

在生产环境中的这种自动化测试足够给大多数运营团队那种不寒而栗，通常在结束一周的工作之前。这不是说单体风格不能够进行完善的监测设置，只是在我们的经验中比较少见。

侧边栏：断路器和产品就绪代码 微服务

断路器(Circuit Breaker) (http://martinfowler.com/bliki/CircuitBreaker.html)与其他模式如Bulkhead和Timeout出现在《Release it!》(http://www.amazon.com/gp/product/B00A32NXZO?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=B00A32NXZO)中。这些模式是被一起实现的，在构建通信应用程序时，它们是至关重要的。这篇Netflix博文(http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html)很好的解释了使用这些模式的应用程序。

既然服务随时都可能失败，那么能够快速检测故障，如果可能的话，能自动恢复服务是很重要的。微服务应用程序投入大量比重来进行应用程序的实时监测，既检查构形要素(每秒多少次数据请求)，又检查业务相关指标(例如每分钟收到多少订单)。语义监测可以提供一套早期预警系统，触发开发团队跟进和调查。

这对微服务架构特别重要，因为微服务偏好编排和事件协作，这会带来突发行为。虽然很多专家称赞偶然涌现的价值，事实的真相是，突发行为有时可能是一件坏事请。监测对于快速发现不良突发行为是至关重要的，所以它可以被修复。

单体可以被构建成和微服务一样透明 - 事实上，它们应该是透明的。不同的是，你绝对需要知道在不同进程中运行的服务是否断开。对同一进程中的库来说，这种透明性是不大可能有用的。

侧边栏：同步调用被认为是有害的

任何时候，在服务间有大量的同步调用，你将遇到停机的乘法效应。简单地说，就是你的系统的停机时间编程各个组件停机时间的乘积。你面临一个选择，让你的调用变成异步或者管理停机时间。在www.guardian.co.uk (http://xn--www-lp6e.guardian.co.uk)，他们已在新平台实现了一个简单的规则 - 每个用户请求一个同步调用，而在Netflix，他们的平台API重设计成在API交换结构(fabric)建立异步性。

微服务团队希望看到为每个单独的服务设置的完善的监控和日志记录，比如控制面板上显示启动/关闭状态和各种各样的运营和业务相关指标。断路器状态、当前吞吐量和时延的详细信息是我们经常遇到的其他例子。

进化式设计

微服务从业者，通常有进化式设计背景并且把服务分解看做是进一步的工具，使应用程序开发者能够控制他们应用程序中的变更而不减缓变更。变更控制并不一定意味着变更的减少 - 用正确的态度和工具，你可以频繁、快速且控制良好的改变软件。

当你试图把软件系统组件化时，你就面临着如何划分成块的决策 - 我们决定分割我们的应用的原则是什么？组件的关键特性是独立的更换和升级的理念[13] - 这意味着我们要找到这样的点，我们可以想象重写组件而不影响其合作者。事实上很多微服务群组通过明确地预期许多服务将被废弃而不是长期演进来进一步找到这些点。

卫报网站是被设计和构建成单体应用程序的一个好例子，但它已向微服务方向演化。网站的核心仍是单体，但他们喜欢通过使用调用单体API构建的微服务添加新功能。这种方法对天然临时性的特性特别方便，比如处理体育赛事的专题页面。网站的这样一部分可以使用快速开发语言迅速的被放在一起，并且一旦赛事结束立即删除。在金融机构中，我们看到类似的方法，为一个市场机会添加新服务，并在几个月甚至几周后丢弃掉。

强调可替代性是模块设计更一般原则的一个特例，它是通过变更模式来驱动模块化的[14]。你想保持在同一模块中相同时间改变的事情。系统中很少变更的部分应该和正在经历大量扰动的部分放在不同的服务里。如果你发现你自己不断地一起改变两个服务，这是它们应该被合并的一个标志。

把组件放在服务中，为更细粒度的发布计划增加了一个机会。对单体来说，任何变更都需要完整构建和部署整个应用程序。而对微服务来说，你只需要重新部署你修改的服务。这可以简化和加速发布过程。坏处是，你必须担心一个服务的变化会阻断其消费者。传统的集成方法试图使用版本管理解决这个问题，但是微服务世界的偏好是只把版本管理作为最后的手段 (http://martinfowler.com/articles/enterpriseREST.html#versioning)。我们可以避免大量的版本管理，通过把服务设计成对他们的提供者的变化尽可能的宽容。

微服务是未来吗？

我们写这篇文章的主要目的是讲解微服务的主要思想和原则。通过花时间做这件事情，我们清楚地认为微服务架构风格是一个重要的思想 - 它值得为企业应用程序认真考虑。我们最近用这种风格构建了一些系统，也知道别人用这种风格并赞成这种风格。

那些我们知道的以某种方式开拓这种架构风格的包括亚马逊，Netflix，卫报 (http://www.theguardian.com/)，英国政府数字服务部门 (https://gds.blog.gov.uk/)，realestate.com.au (http://martinfowler.com/articles/realestate.com.au)，前锋和comparethemarket.com (http://www.comparethemarket.com/)。2013年的会议电路中全是正向微服务类别转移的公司 - 包括Travis CI。此外还有大量的组织长期以来一直在做可归为微服务类别的事情，但是还没有使用这个名字。(这通常被称为SOA - 虽然，正如我们说过的，SOA有许多矛盾的形式。[15])

尽管有这些积极的经验，但是，我们并不认为我们确信微服务是软件架构的未来发展方向。虽然到目前为止，与单体应用程序相比，我们的经验是正面的，但我们意识到这样的事实，并没有经过足够的时间使我们做出充分的判断。

通常，你的架构决策的真正后果是在你做出这些决定的几年后才显现的。我们已经看到对模块化有强烈愿望的一个好团队用单体架构构建的项目，已经衰败了多年。很多人相信微服务是不太可能出现这种衰败的，因为服务界限是明确的，并且很难围绕它打补丁。然而，知道我们看到经过足够岁月的足够的系统，我们不能真正评估微服务架构有多么成熟。

人们当然有理由希望微服务是多么不成熟。在组件化中做任何努力，成功取决于软件在多大程度上适用于组件化。很难弄清楚组件边界在哪里。进化式设计承认获取正确边界的困难性和使它们易于重构的重要性。但当你的组件是带有远程通信的服务时，那么重构它比重构带有进程内库的服务难很多。跨服务边界移动代码是很困难的，任何接口变更都需要在参与者之间进行协调，需要添加向后兼容层，并且测试也变得更加复杂。

> 侧边栏：《构建微服务》
>
> 我们的同事Sam Newman花费2014年的大部分时间写了一本书(http://www.amazon.com/gp/product/1491950358?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=1491950358)，捕捉了我们构建微服务的经验。如果你想深入到这个话题中，这应该是你的下一步。

另一个问题是，如果组件不组成的干净利索，那么所有你做的是将复杂度从组件内部转移到组件之间的连接。不仅仅是把复杂性移到周围，它将复杂性移动到一个不太明确、难以控制的地方。在没有服务间的凌乱连接的情况下，当你在看一个小的、简单的组件内部时，你可以很容易的认为事情是更好的。

最后，有团队技能的因素。更熟练的团队倾向于采用新技术。但是对更熟练的团队更有效的一种技术不一定适合于不太熟练的团队。我们已经看到大量的例子，不太熟练的团队构建了凌乱的单体架构，但这需要时间去看当微服务发生这种凌乱时会发生什么。一个差的团队总是创建一个差的系统 - 很难讲在这个例子中微服务会减少这种凌乱还是使它更糟糕。

我们听到的一个合理的说法是，你不应该从微服务架构开始。相反，从单体开始，使它保持模块化，一旦单体成为问题时把它分解成微服务。(虽然这个建议是不理想的，因为一个好的进程内接口通常不是一个好的服务接口。)

所以我们怀着谨慎乐观的态度写了这篇文章。到目前为止，我们已经看到关于微服务风格足以觉得这是一条值得探索的路。我们不能肯定地说，我们将在哪里结束，但软件开发的挑战之一是，你只能基于目前能拿到手的不完善的信息作出决定。

1. 2011年5月在威尼斯召开的软件架构研讨会上，“微服务”这一术语被讨论用来描述参与者一直在探索的一种常见的架构风格。2012年5月，该研讨会决定使用“微服务”作为最合适的名字。2012年3月在波兰克拉科夫市举办的33届Degree大会上，James介绍了这些想法作为一个案例研究微服务 - Java，Unix方式 (http://2012.33degree.org/talk/show/67)，

Fred George也差不多在同一时间 (http://www.slideshare.net/fredgeorge/micro-service-architecure)提出。Netflix的Adrian Cockcroft把这种方法描述为“细粒度的SOA”，在网域级开拓了这一风格，还有在该文中提到的许多人 - Joe Walnes, Dan North, Evan Botcher 和 Graham Tackley。 ↩

2. 单体这一术语已被Unix社区使用了一段时间，在《Unix编程艺术》(http://www.amazon.com/gp/product/B003U2T5BA?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=B003U2T5BA)中用它来描述非常大的系统。 ↩
3. 很多面向对象的设计人员，包括我们自己，在领域驱动设计(http://www.amazon.com/gp/product/0321125215?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321125215)意义上使用服务对象术语，该对象不依赖于实体执行一个重要进程。这和我们在本文中如何使用“服务”是不同的概念。不幸的是，服务这个词有两个含义，我们不得不忍受这个多义词。 ↩
4. 我们认为应用程序是一个社会结构(http://martinfowler.com/bliki/ApplicationBoundary.html)，它由代码基、功能组、资金体组合在一起。 ↩
5. 原文可在Melvyn Conway的网站上找到，在这里(http://www.melconway.com/Home/Committees_Paper.html)。 ↩
6. 在极端规模下，组织通常移至二进制协议并权衡规模的透明度。例如protobufs ()。使用二进制协议的系统仍旧展现出智能端点、哑管道。大多数网站，当然绝大多数企业不需要做这种权衡，透明度可以是一个很大的胜利。 ↩
7. 我们忍不住提起Jim Webber的说法，ESB全称是“令人震惊的意大利面条盒”(http://www.infoq.com/presentations/soa-without-esb) ↩
8. Netflix使这种联系清晰起来 - 直到最近作为细粒度SOA提及他们的架构风格。 ↩
9. “YAGNI”也就是“You Aren’t Going To Need It(你将不需要它)”是一个XP原则(http://c2.com/cgi/wiki?YouArentGonnaNeedIt)和劝诫，在你知道你需要它们时才添加特性 ↩
10. 我们声称单体是单一语言的，这有一点不诚实 - 要在现在web上构建系统，你可能需要知道JavaScript、XHTML、CSS、选择的服务器语言、SQL和ORM方言。很难只用单一语言，但是你知道我的意思。 ↩
11. 在2013年11月的Flowcon大会上提交的这个出色演讲中(http://www.slideshare.net/adrianco/flowcon-added-to-for-cmg-keynote-talk-on-how-speed-wins-and-how-netflix-is-doing-continuous-delivery)，Adrian Cockcroft特别提到“开发者自助服务”和“开发者运行他们自己写的代码”(原文如此)。 ↩

12. 我们这里有一点不诚实。显然在更复杂的拓扑结构中部署更多的服务要比部署单一单体更困难。幸运的是，模式减少了这种复杂性 - 在工具上的投资仍是必须的。 ↩
13. 事实上，Dan North提到这种风格是可更换的组件架构而不是微服务。因为这似乎是在讨论我们更喜欢的后者的一个特征子集。 ↩
14. Kent Beck强调这是他《实现模式》(http://www.amazon.com/gp/product/0321413091?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321413091)一书中的设计原则之一。 ↩
15. SOA几乎是这段历史的根源。我记得当SOA这一术语出现在本世纪初时，有人说“多年来我们一直这样做”。一个理由是，这种风格看其根源是在企业计算早期COBOL程序通过数据文件通信的方式。在另一个方向，有人可能会说微服务和Erlang编程模型相同，但被应用于企业应用程序上下文。 ↩

← PREVIOUS POST (/BLOG/2015/09/20/COMPILE-AND-DEPLOY-USING-DOCKER/)

NEXT POST → (/BLOG/2015/05/12/NOTES-ABOUT-THE-LONGTAIL-THEORY/)

FEATURED TAGS (/tags/)

microservices (/tags/microservices/) 微服务 (/tags/微服务/)

(https://www.zhihu.com/people/cui-yan-bao)

(http://weibo.com/yygccui) (https://github.com/YYGCui)

(https://www.linkedin.com/in/yanbaoc)