微服务 | Martin Fowler

作者：Martin Fowler & James Lewis 译者：伍斌

原文：http://martinfowler.com/articles/microservices.html

转载自：https://mp.weixin.qq.com/s/dqk7CTX1W3FNCE7GyCgbSQ

https://mp.weixin.qq.com/s/h4p2_-RmCFJzXmusQOWbcw

https://insights.thoughtworks.cn/microservices-martin-fowler/

（注：仅调整了译文的文字排版）

微服务

有关这个新的技术架构术语的定义

> “微服务架构”（Microservice Architecture）这个术语最近几年横空出世，来描述这样一种特定的软件设计方法，即以若干组可独立部署的服务的方式进行软件应用系统的设计。尽管这种架构风格尚无精确的定义，但其在下述方面还是存在一定的共性，即围绕业务功能的组织、自动化部署、端点智能、和在编程语言和数据方面进行去中心化的控制。

2014年3月25日

作者：

James Lewis 是 ThoughtWorks 首席咨询师，而且是该公司的技术顾问委员会成员。James 对于采用相互协作的小型服务来构建应用系统的兴趣，源自于他的整合大规模企业系统的工作背景。他已经使用微服务构建了许多系统，而且几年以来已经成为正在成长的微服务社区的积极参与者。

Martin Fowler 是一位作者和演讲者，并且在软件开发行业中，他通常是最能说的那一位。他长期以来一直困惑于这样的问题，即如何才能将软件系统进行组件化。那些声称已将软件进行组件化的声音他听到了很多，但是很少有能让他满意的。他希望微服务不要辜负其倡导者们对它的最初的期望。

目录

- 一个微服务应该有多大？（How big is a microservice?）
- 微服务与 SOA（Microservices and SOA）
- 多种编程语言，多种选择可能（Many languages, many options）
- “实战检验”的标准与“强制执行”的标准（Battle-tested standards and enforced standards）
- 让“方向正确地做事”更容易（Make it easy to do the right thing）
- “断路器”与“可随时上线的代码”（The circuit breaker and production ready code）
- “同步调用”有害（Synchronous calls considered harmful）

“微服务”（Microservices）——这是在软件架构这条熙熙攘攘的大街上出现的又一个新词儿。我们自然会对它投过轻蔑的一瞥，但是这个小小的术语却描述了一种引人入胜的软件系统的风格。最近几年，我们已经看到许多项目使用了这种风格，而且至今其结果都是良好的，以至于对于我们许多 ThoughtWorks 的同事来说，它正在成为构建企业应用系统的缺省的风格。然而，很不幸的是，我们找不到有关它的概要信息，即什么是微服务风格，以及如何设计微服务风格的架构。

简而言之，微服务架构风格[1]这种开发方法，是以开发一组小型服务的方式来开发一个独立的应用系统的。其中每个小型服务都运行在自己的进程中，并经常采用 HTTP 资源 API 这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术。对这些微服务我们仅做最低限度的集中管理。

在开始介绍微服务风格之前，将其与单块（monolithic）风格进行对比还是很有用的：一个单块应用系统是以一个单个单元的方式来构建的。企业应用系统经常包含三个主要部分：客户端用户界面、数据库和服务端应用系统。客户端用户界面包括 HTML 页面和运行在用户机器的浏览器中的 JavaScript。数据库中包括许多表，这些表被插入一个公共的且通常为关系型的数据库管理系统中。这个服务端的应用系统就是一个单块应用（monolith）——一个单个可执行的逻辑程序[2]。对于该系统的任何改变，都会涉及构建和部署上述服务端应用系统的一个新版本。

> 在我的“微服务资源指南”中能找到有关微服务最好的文章、视频、图书和播客媒体。

这样的单块服务器是构建上述系统的一种自然的方式。处理用户请求的所有逻辑都运行在一个单个的进程内，从而能使用编程语言的基本特性，来把应用系统划分为类、函数和命名空间。通过精心设计，就能在开发人员的笔记本电脑上运行和测试这样的应用系统，并且使用一个部署流水线来确保变更被很好地进行了测试，并被部署到生产环境中。通过负载均衡器运行许多实例，来将这个单块应用进行横向扩展。

单块应用系统可以被成功地实现，但是渐渐地，特别是随着越来越多的应用系统正被部署到云端，人们对它们开始表现出不满。软件变更受到了很大的限制，应用系统的一个很小的部分的一处变更，也需要将整个单块应用系统进行重新构建和部署。随着时间的推移，单块应用开始变得经常难以保持一个良好的模块化结构，这使得它变得越来越难以将一个模块的变更的影响控制在该模块内。当对系统进行扩展时，不得不扩展整个应用系统，而不能仅扩展该系统中需要更多资源的那些部分。

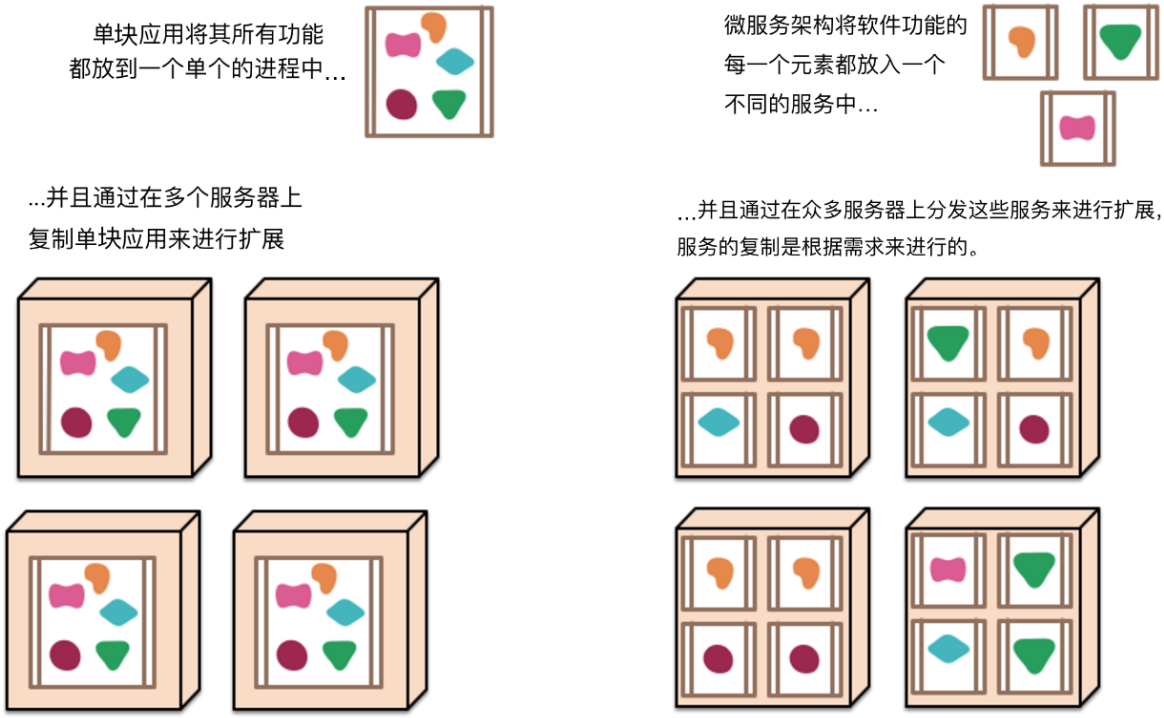

图1: 单块应用和微服务（Monoliths and Microservices）

这些不满导致了微服务架构风格的诞生：以构建一组小型服务的方式来构建应用系统。除了这些服务能被独立地部署和扩展，每一个服务还能提供一个稳固的模块边界，甚至能允许使用不同的编程语言来编写不同的服务。这些服务也能被不同的团队来管理。

我们并不认为微服务风格是一个新颖或创新的概念，它的起源至少可以追溯到 Unix 的设计原则。但是我们觉得，考虑微服务架构的人还不够多，并且如果对其加以使用，许多软件的开发工作能变得更好。

微服务架构的九大特性

虽然不能说存在微服务架构风格的正式定义，但是可以尝试描述我们所见到的能够被贴上微服务标签的那些架构的共性。下面所描述的所有这些共性，并不是所有的微服务架构都完全具备，但是我们确实期望大多数微服务架构都具备这些共性中的大多数特性。尽管我们两位作者已经成为这个相当松散的社区的活跃成员，但我们的本意还是试图描述我们两人在自己和自己所了解的团队的工作中所看到的情况。特别要指出，我们不会制定大家需要遵循的微服务的定义。

特性一：“组件化”与“多服务”

自我们开始从事软件业已来，发现大家都有一个把组件插在一起来构建系统的愿望，就像在物理世界中所看到的那样。在过去几十年中，我们已经看到，在公共软件库方面已经取得了相当大的进展，这些软件库是大多数编程语言平台的组成部分。

当谈到组件时，就会碰到一个有关定义的难题，即什么是组件？我们的定义是，一个**组件**（component）就是一个可以独立更换和升级的软件单元。

微服务架构也会使用软件库，但其将自身软件进行组件化的主要方法是将软件分解为诸多服务。我们将**软件库**（libraries）定义为这样的组件，即它能被链接到一段程序，且能通过内存中的函数来进行调用。然而，**服务**（services）是进程外的组件，它们通过诸如 web service 请求或远程过程调用这样的机制来进行通信（这不同于许多面向对象的程序中的 service object 概念[3]）。

以使用服务（而不是以软件库）的方式来实现组件化的一个主要原因是，服务可被独立部署。如果一个应用系统[4]由在单个进程中的多个软件库所组成，那么对任一组件做一处修改，都不得不重新部署整个应用系统。但是如果该应用系统被分解为多个服务，那么对于一个服务的多处修改，仅需要重新部署这一个服务。当然这也不是绝对的，一些变更服务接口的修改会导致多个服务之间的协同修改。但是一个良好的微服务架构的目的，是通过内聚的服务边界和服务协议方面的演进机制，来将这样的修改变得最小化。

以服务的方式来实现组件化的另一个结果，是能获得更加显式的（explicit）组件接口。大多数编程语言并没有一个良好的机制来定义显式的**发布接口**（Published Interface）。通常情况下，这样的接口仅仅是文档声明和团队纪律，来避免客户端破坏组件的封装，从而导致组件间出现过度紧密的耦合。通过使用显式的远程调用机制，服务能更容易地避免这种情况发生。

如此这般地使用服务，也会有不足之处。比起进程内调用，远程调用更加昂贵。所以远程调用 API 接口必须是粗粒度的，而这往往更加难以使用。如果需要修改组件间的职责分配，那么当跨越进程边界时，这种组件行为的改动会更加难以实现。

近似地，我们可以把一个个服务映射为一个个运行时的进程，但这仅仅是一个近似。一个服务可能包括总是在一起被开发和部署的多个进程，比如一个应用系统的进程和仅被该服务使用的数据库。

特性二：围绕“业务功能”组织团队

当在寻求将一个大型应用系统分解成几部分时，公司管理层往往会聚焦在技术层面上，这会导致组建用户界面团队、服务器端团队和数据库团队。当团队沿着这些技术线分开后，即使要实现软件一个简单的变更，也会导致跨团队的项目时延和预算审批。在这种情况下，聪明的团队会进行局部优化，“两害相权取其轻”，来直接把代码逻辑塞到他们能访问到的任意应用系统中。换句话说，这种情况会导致代码逻辑散布在系统各处。这就是康威定律（Conway's Law）[5]在起作用的活生生的例子。

> 任何设计（广义上的）系统的组织，都会产生这样一个设计，即该设计的结构与该组织的沟通结构相一致。
> ——梅尔文•康威（Melvyn Conway），1967年

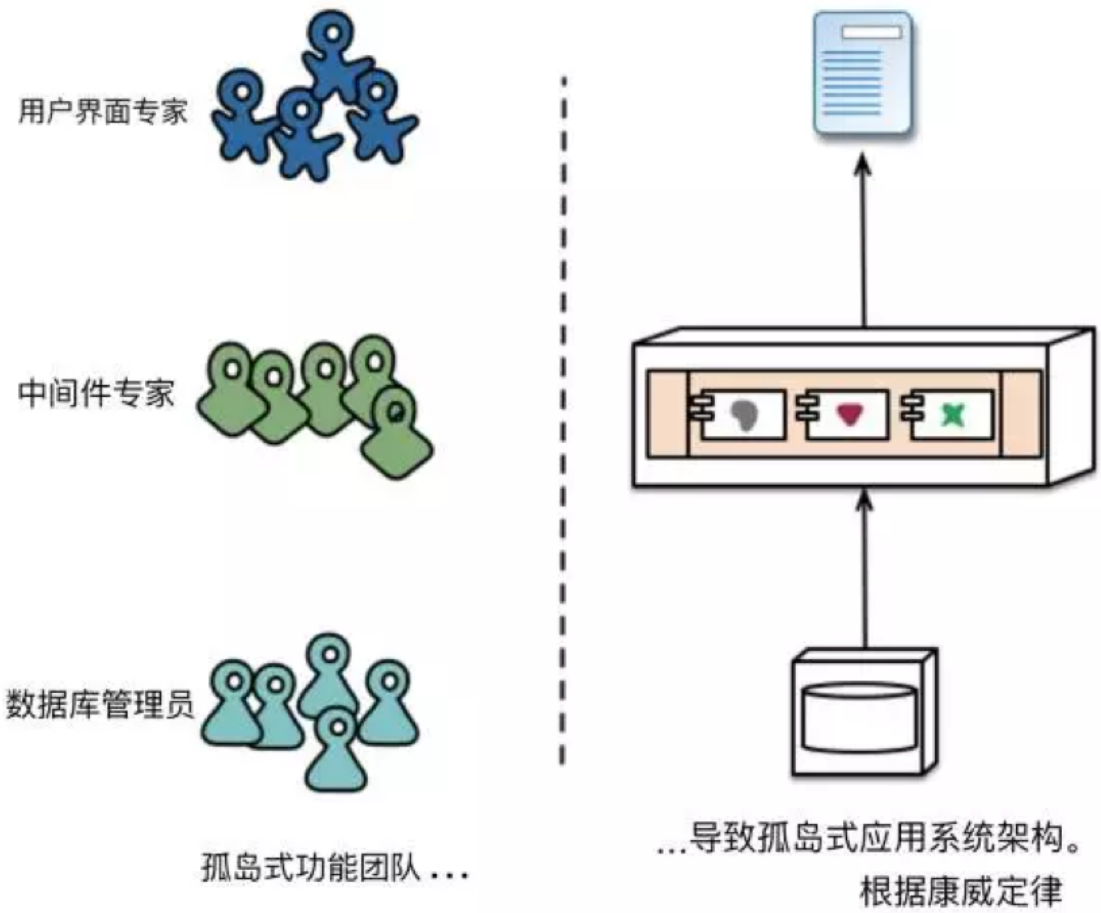

图2：康威定律在起作用

微服务使用不同的方法来分解系统，即根据**业务功能**（business capability）来将系统分解为若干服务。这些服务针对该业务领域提供多层次广泛的软件实现，包括用户界面、持久性存储以及任何对外的协作性操作。因此，团队是跨职能的，它拥有软件开发所需的全方位的技能：用户体验、数据库和项目管理。

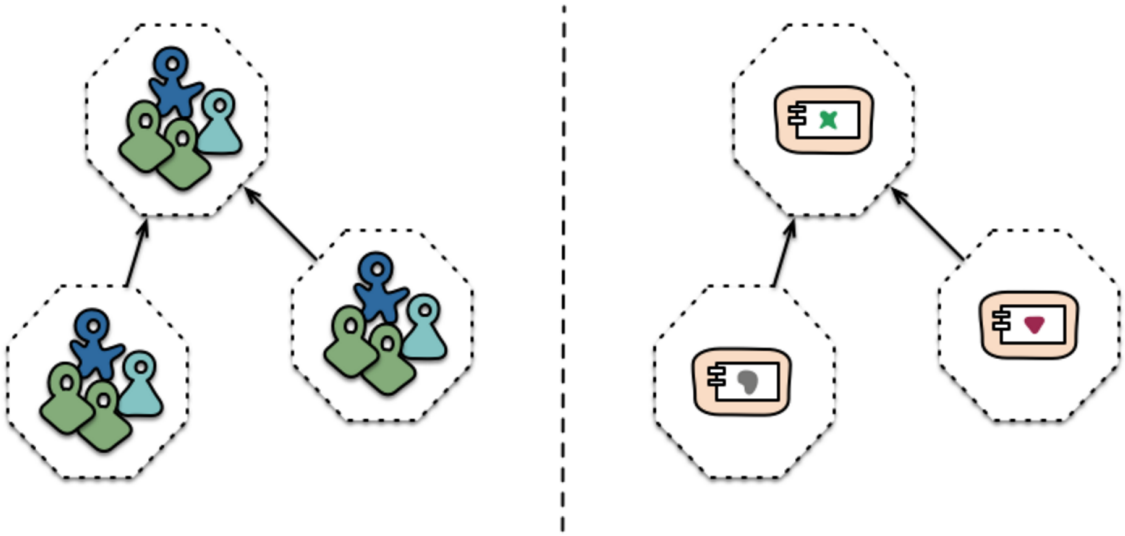

图3：被团队边界所强化的服务边界

以上述方式来组织团队的公司是 www.comparethemarket.com。跨职能团队负责构建和运维每个产品，而每个产品被拆分为多个独立的服务，彼此通过一个消息总线来通信。

> ## 一个微服务应该有多大?
>
> 尽管对于这种架构风格，“微服务”已经成为一个流行的名字，但是这个名字确实会不幸地导致大家对服务规模的关注，并且产生了有关什么是“微”（micro）的争论。在与微服务从业者的交谈中，我们看到了有关服务的一系列规模。所听到的最大的一个服务的规模，是遵循了亚马逊的“两个比萨团队”（即一个团队可以被两个比萨所喂饱）的理念，这意味着这个团队不会多于 12 人。对于规模较小的服务，我们已经看到一个 6 人的团队在支持 6 个服务。
>
> 这引出了一个问题，即“每 12 人做一个服务”和“每人做一个服务”这样有关服务规模的差距，是否已经大到不能将两者都纳入微服务之下？此时，我们认为最好还是把它们归为一类，但是随着进一步探索这种架构风格，绝对有可能我们将来会改变主意。

大型单块应用系统也可以始终根据业务功能来进行模块化设计，虽然这并不常见。当然，我们会敦促构建单块应用系统的大型团队根据业务线来将自己分解为若干小团队。在这方面，我们已经看到的主要问题是，他们往往是一个团队包含了太多的业务功能。如果这个“单块”跨越了许多模块的边界，那么这个团队的每一个成员都难以记忆所有这些模块的业务功能。此外，我们看到这些模块的边界需要大量的团队纪律性来强制维持。而实现组件化的服务所必要的更加显式的边界，能更加容易地保持团队边界的清晰性。

特性三：“做产品”而不是“做项目”

我们所看的的大部分应用系统的开发工作都使用项目模型：目标是交付某一块软件，之后就认为完工了。一旦完工后，软件就被移交给维护团队，接着那个构建该软件的项目团队就会被解散。

微服务的支持者们倾向于避免使用上述模型，而宁愿采纳“一个团队在一个产品的整个生命周期中都应该保持对其拥有”这样的理念。通常认为这一点源自亚马逊的“谁构建，谁运行”（you build, you run it）的理念，即一个开发团队对一个在生产环境下运行的软件负全责。这会使开发人员每天都会关注软件是如何在生产环境下运行的，并且增进他们与用户的联系，因为他们必须承担某些支持工作。

这样的“产品”理念，是与业务功能的联动绑定在一起的。它不会将软件看作是一个待完成的功能集合，而是认为存在这样一个持续的关系，即软件如何能助其客户来持续增进业务功能。

当然，单块应用系统的开发工作也可以遵循上述“产品”理念，但是更细粒度的服务，能让服务的开发者与其用户之间的个人关系的创建变得更加容易。

特性四：“智能端点”与“傻瓜管道”

当在不同的进程之间构建各种通信结构时，我们已经看到许多产品和方法，来强调将大量的智能特性纳入通信机制本身。这种状况的一个典型例子，就是“企业服务总线”（Enterprise Service Bus, ESB）。ESB 产品经常包括高度智能的设施，来进行消息的路由、编制（choreography）、转换，并应用业务规则。

微服务社区主张采用另一种做法：**智能端点**（smart endpoints）和**傻瓜管道**（dumb pipes）。使用微服务所构建的各个应用的目标，都是尽可能地实现“高内聚和低耦合”——他们拥有自己的领域逻辑，并且更多地是像经典 Unix 的“过滤器”（filter）那样来工作——即接收一个请求，酌情对其应用业务逻辑，并产生一个响应。这些应用通过使用一些简单的 REST 风格的协议来进行编制，而不去使用诸如下面这些复杂的协议，即“WS-编制”（WS-Choreography）、BPEL 或通过位于中心的工具来进行编排（orchestration）。

微服务最常用的两种协议是：带有资源 API 的 HTTP “请求－响应”协议，和轻量级的消息发送协议[6]。对于前一种协议的最佳表述是：

> 成为 Web，而不是躲着 Web (Be of the web, not behind the web)
> ——Ian Robinson

这些微服务团队在开发中，使用在构建万维网（world wide web）时所使用的原则和协议（并且在很大程度上，这些原则和协议也是在构建 Unix 系统时所使用的）。那些被使用过的 HTTP 资源，通常能被开发或运维人员轻易地缓存起来。

最常用的第二种协议，是通过一个轻量级的消息总线来进行消息发送。此时所选择的基础设施，通常是“傻瓜”（dumb）型的（仅仅像消息路由器所做的事情那样傻瓜）——像 RabbitMQ 或 ZeroMQ 那样的简单实现，即除了提供**可靠的异步机制**以外不做其他任何事情——智能功能存在于那些生产和消费诸多消息的各个端点中，即存在于各个服务中。

在一个单块系统中，各个组件在同一个进程中运行。它们相互之间的通信，要么通过方法调用，要么通过函数调用来进行。将一个单块系统改造为若干微服务的最大问题，在于对通信模式的改变。仅仅将内存中的方法调用转换为 RPC 调用这样天真的做法，会导致微服务之间产生繁琐的通信，使得系统表现变糟。取而代之的是，需要用更粗粒度的协议来替代细粒度的服务间通信。

特性五：“去中心化”地治理技术

使用中心化的方式来对开发进行治理，其中一个后果，就是趋向于在单一技术平台上制定标准。经验表明，这种做法会带来局限性——不是每一个问题都是钉子，不是每一个方案都是锤子。我们更喜欢根据工作的不同来选用合理的工具。尽管那些单块应用系统能在一定程度上利用不同的编程语言，但是这并不常见。

如果能将单块应用的那些组件拆分成多个服务，那么在构建每个服务时，就可以有选择不同技术栈的机会。想要使用 Node.js 来搞出一个简单的报表页面？尽管去搞。想用 C++ 来做一个特别出彩儿的近乎实时的组件？没有问题。想要换一种不同风格的数据库，来更好地适应一个组件的读取数据的行为？可以重建。

当然，仅仅*能*做事情，并不意味着这些事情就*应该*被做——不过用微服务的方法把系统进行拆分后，就拥有了技术选型的机会。

> ## 微服务和 SOA
>
> 当我们谈起微服务时，一个常见的问题就会出现：是否微服务仅仅是十多年前所看到的“面向服务的架构”（Service Oriented Architecture, SOA）？这样问是有道理的，因为微服务风格非常类似于一些支持 SOA 的人所赞成的观点。然而，问题在于 SOA 这个词儿意味着太多不同的东西。而且大多数时候，我们所遇到的某些被称作“SOA”的事物，明显不同于本文所描述的风格。这通常由于它们专注于 ESB，来集成各个单块应用。
>
> 特别地，我们已经看到如此之多的面向服务的拙劣实现——从将系统复杂性隐藏于 ESB 中的趋势[7]，到花费数百万进行多年却没有交付任何价值的失败项目，到顽固抑制变化发生的中心化技术治理模型——以至于有时觉得其所造成的种种问题真的不堪回首。
>
> 当然，在微服务社区投入使用的许多技术，源自各个开发人员将各种服务集成到各个大型组织的经验。“容错读取”（Tolerant Reader）模式就是这样一个例子。对于 Web 的广泛使用，使得人们不再使用一些中心化的标准，而使用一些简单的协议。坦率地说，这些中心化的标准，其复杂性已经达到令人吃惊的程度。（任何时候，如果需要一个本体（ontology）来管理其他各个本体，那么麻烦就大了。）
>
> 这种常见的 SOA 的表现，已使得一些微服务的倡导者完全拒绝将自己贴上 SOA 的标签。尽管其他人会将微服务看作是 SOA 的一种形式[8]，也许微服务就是以正确的形式来实现面向服务的 SOA。不管是哪种情况，SOA 意味着如此之多的不同事物，这表明用一个更加干净利落的术语来命名这种架构风格是很有价值的。

相比选用业界一般常用的技术，构建微服务的那些团队更喜欢采用不同的方法。与其选用一组写在纸上已经定义好的标准，他们更喜欢编写一些有用的工具，来让其他开发者能够使用，以便解决那些和他们所面临的问题相似的问题。这些工具通常源自他们的微服务实施过程，并且被分享到更大规模的组织中，这种分享有时会使用内部开源的模式来进行。现在，git 和 github 已经成为事实上的首选版本控制系统。在企业内部，开源的做法正在变得越来越普遍。

Netflix 公司是遵循上述理念的好例子。将实用且经过实战检验的代码以软件库的形式共享出来，能鼓励其他开发人员以相似的方式来解决相似的问题，当然也为在需要的时候选用不同的方案留了一扇门。共享软件库往往集中在解决这样的常见问题，即数据存储、进程间的通信和下面要进一步讨论的基础设施的自动化。

对于微服务社区来说，日常管理开销这一点不是特别吸引人。这并不是说这个社区并不重视服务契约。恰恰相反，它们在社区里出现得更多。这正说明这个社区正在寻找对其进行管理的各种方法。像**“容错读取”**（Tolerant Reader）和**“消费者驱动的契约”**（Consumer-Driven Contracts）这样的模式，经常被运用到微服务中。这些都有助于服务契约进行独立演进。将执行“消费者驱动的契约”做为软件构建的一部分，能增强开发团队的信心，并提供所依赖的服务是否正常工作的快速反馈。实际上，我们了解到一个在澳洲的团队就是使用“消费者驱动的契约”来驱动构建多个新服务的。他们使用了一些简单的工具，来针对每一个服务定义契约。甚至在新服务的代码编写之前，这件事就已经成为自动化构建的一部分了。接下来服务仅被构建到刚好能满足契约的程度——这是一个在构建新软件时避免 YAGNI[9] 困境的优雅方

法。这些技术和工具在契约周边生长出来，由于减少了服务之间在时域（temporal）上的耦合，从而抑制了对中心契约管理的需求。

> ### 多种编程语言，多种选择可能
>
> 做为一个平台，JVM 的发展仅仅是一个将各种编程语言混合到一个通用平台的最新例证。近十年以来，在平台外层实现更高层次的编程语言，来利用更高层次的抽象，已经成为一个普遍做法。同样，在平台底层以更低层次的编程语言编写性能敏感的代码也很普遍。然而，许多单块系统并不需要这种级别的性能优化，另外 DSL 和更高层次的抽象也不常用（这令我们感到失望）。相反，许多单块应用通常就使用单一编程语言，并且有对所使用的技术数量进行限制的趋势[10]。

或许去中心化地治理技术的极盛时期，就是亚马逊的“谁构建，谁运行”的风气开始普及的时候。各个团队负责其所构建的软件的所有方面的工作，其中包括 7 x 24 地对软件进行运维。将运维这一级别的职责下放到团队这种做法，目前绝对不是主流。但是我们确实看到越来越多的公司，将运维的职责交给各个开发团队。Netflix 就是已经形成这种风气的另一个组织[11]。避免每天凌晨 3 点被枕边的寻呼机叫醒，无疑是在程序员编写代码时令其专注质量的强大动力。而这些想法，与那些传统的中心化技术治理的模式具有天壤之别。

特性六：“去中心化”地管理数据

去中心化地管理数据，其表现形式多种多样。从最抽象的层面看，这意味着各个系统对客观世界所构建的概念模型，将彼此各不相同。当在一个大型的企业中进行系统集成时，这是一个常见的问题。比如对于“客户”这个概念，从销售人员的视角看，就与从支持人员的视角看有所不同。从销售人员的视角所看到的一些被称之为“客户”的事物，或许在支持人员的视角中根本找不到。而那些在两个视角中都能看到的事物，或许各自具有不同的属性。更糟糕的是，那些在两个视角中具有相同属性的事物，或许在语义上有微妙的不同。

上述问题在不同的应用程序之间经常出现，同时也会出现在这些应用程序内部，特别是当一个应用程序被分成不同的组件时就会出现。思考这类问题的一个有用的方法，就是使用领域驱动设计（Domain-Driven Design, DDD）中的“限界上下文”（Bounded Context）的概念。DDD 将一个复杂的领域划分为多个限界上下文，并且将其相互之间的关系用图画出来。这一划分过程对于单块和微服务架构两者都是有用的，而且就像前面有关“业务功能”一节中所讨论的那样，在服务和各个限界上下文之间所存在的自然的联动关系，能有助于澄清和强化这种划分。

> ### “实战检验”的标准与“强制执行”的标准
>
> 微服务的下述做法有点泾渭分明的味道，即他们趋向于避开被那些企业架构组织所制定的硬性实施的标准，而愉快地使用甚至传播一些开放标准，比如 HTTP、ATOM 和其他微格式的协议。
>
> 这里的关键区别是，这些标准是如何被制定以及如何被实施的。像诸如 IETF 这样的组织所管理的各种标准，只有达到下述条件才能称为标准，即该标准在全球更广阔的地区有一些正在运行的实现案例，而且这些标准经常源自一些成功的开源项目。
>
> 这些标准组成了一个世界，它区别于来自下述另一个世界的许多标准，即企业世界。企业世界中的标准，经常由这样特点的组织来开发，即缺乏用较新技术进行编程的经验，或受到供应商的过度影响。

如同在概念模型上进行去中心化的决策一样，微服务也在数据存储上进行去中心化的决策。尽管各个单块应用更愿意在逻辑上各自使用一个单独的数据库来持久化数据，但是各家企业往往喜欢一系列单块应用共用一个单独的数据库——许多这样的决策是被供应商的各种版权商业模式所驱动出来的。微服务更喜欢让每一个服务来管理其自有数据库。其实现可以采用相同数据库技术的不同数据库实例，也可以采用完全不同的数据库系统。这种方法被称作“多语种持久化”（Polyglot Persistence）。在一个单块系统中也能使用多语种持久化，但是看起来这种方法在微服务中出现得更加频繁。

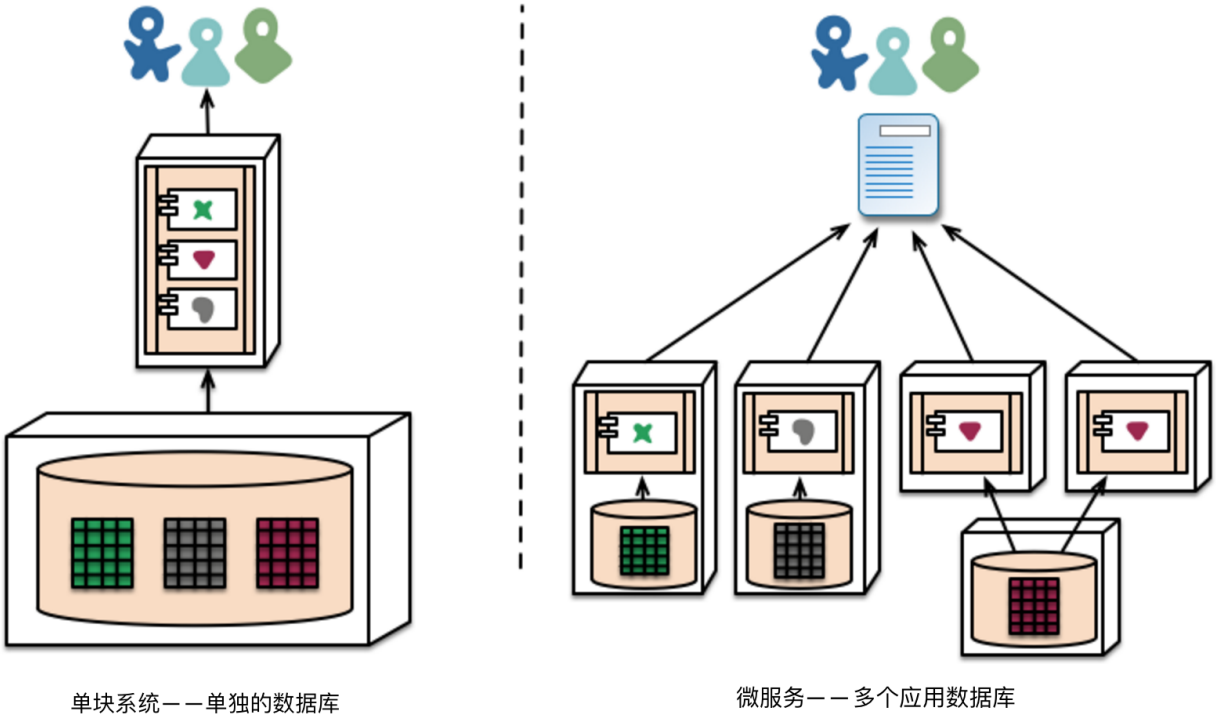

图4：微服务更喜欢让每一个服务来管理其自有数据库

在各个微服务之间将数据的职责进行“去中心化”的管理，会影响软件更新的管理。处理更新的常用方法，是当更新多个资源的时候，使用事务来保证一致性。这种方法经常在单块系统中被采用。

像这样地使用事务，有助于保持数据一致性。但是在时域上会引发明显的耦合，这样当在多个服务之间处理事务时会出现一致性问题。分布式事务实现起来难度之大是臭名远扬的。为此，微服务架构更强调在各个服务之间进行“无事务”协作。这源自微服务社区明确地认识到下述两点，即数据一致性可能只要求数据在最终达到一致，并且一致性问题能够通过补偿操作来进行处理。

对于许多开发团队来说，选择这种方式来管理数据的“非一致性”，是一个新的挑战。但这也经常符合在商业上的实践做法。通常情况下，为了快速响应需求，商家们都会处理一定程度上的数据“非一致性”，来通过做某种反向过程进行错误处理。只要修复错误的成本，与在保持更大的数据一致性却导致丢了生意所产生的成本相比，前者更低，那么这种“非一致性”地管理数据的权衡就是值得的。

特性七：“基础设施”自动化

基础设施自动化技术在过去几年里已经得到长足的发展。云的演进，特别是 AWS 的发展，已经降低了构建、部署和运维微服务的操作复杂性。

许多使用微服务构建的产品和系统，正在被这样的团队所构建，即他们都具备极其丰富的“**持续交付**”（Continuous Delivery）和其前身“**持续集成**”（Continuous Integration）的经验。用这种方法构建软件的各个团队，广泛采用了基础设施的自动化技术。如下图的构建流水线所示：

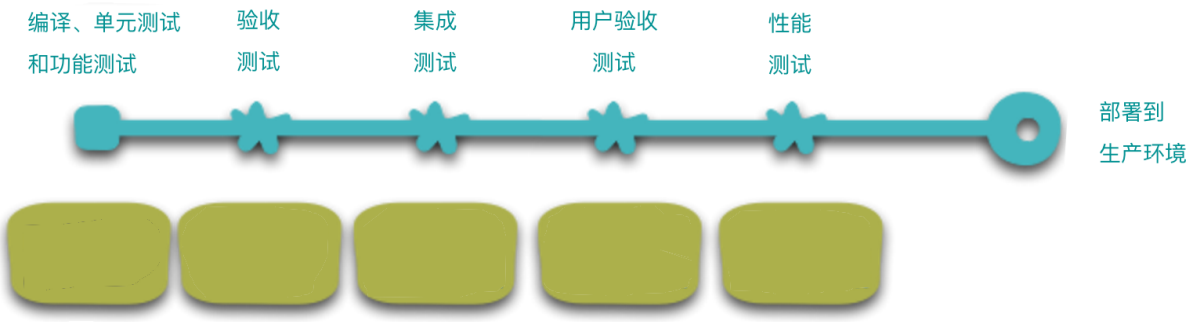

图5：基本的构建流水线

由于本文并不是一篇有关持续交付的文章，所以下面仅提请大家注意两个持续交付的关键特点。为了尽可能地获得对正在运行的软件的信心，需要运行大量的**自动化测试**。让可工作的软件达到“晋级”（Promotion）状态、从而“推上”流水线，就意味着可以在每一个新的环境中，对软件进行**自动化部署**。

一个单块应用程序，能够相当愉快地在上述各个环境中，被构建、测试和推送。其结果是，一旦在下述工作中进行了投入，即针对一个单块系统将其通往生产环境的通道进行自动化，那么部署更多的应用系统似乎就不再可怕。记住，持续交付的目的之一，是让“部署”工作变得“无聊”。所以不管是一个还是三个应用系统，只要部署工作依旧很“无聊”，那么就没什么可担心的了[12]。

> ## 让“沿着正确的方向做事”更容易
>
> 那些因实现持续交付和持续集成所增加的自动化工作的副产品，是一些对开发和运维人员有用的工具。现在，能完成下述工作的工具已经相当常见了，即创建工件（artefacts）、管理代码库、启动一些简单的服务、或增加标准的监控和日志功能。Web 上最好的例子可能是 Netflix 提供的一套开源工具集，但也有其他一些好工具，包括我们已经广泛使用的 Dropwizard。

我们所看到的各个团队在广泛使用基础设施自动化实践的另一个领域，是在生产环境中管理各个微服务。与前面我们对比单块系统和微服务所说的正相反，只要部署工作很无聊，那么在这一点上单块系统和微服务就没什么区别。然而，两者在运维领域的情况却截然不同。

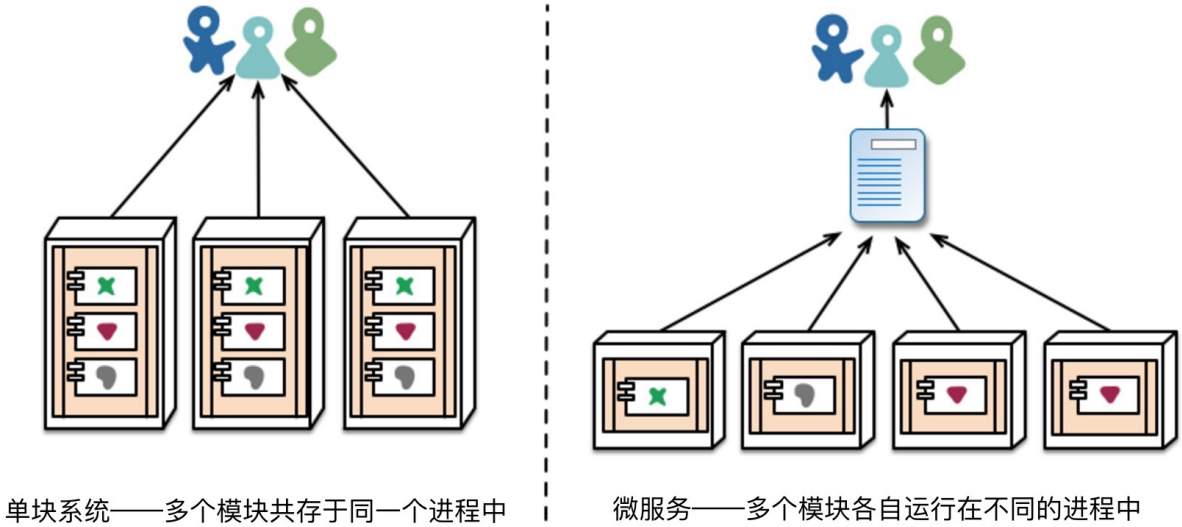

图6：两者的模块部署经常会有差异

特性八：“容错”设计

使用各个微服务来替代组件，其结果是各个应用程序需要被设计的能够容忍这些服务所出现的故障。如果服务提供方不可用，那么任何对该服务的调用都会出现故障。客户端要尽可能优雅地应对这种情况。与一个单块设计相比，这是一个劣势。因为在处理这种情况时会引入额外的复杂性。为此，各个微服务团队在不断地反思：这些服务故障是如何影响用户体验的。Netflix 公司所研发的开源测试工具 Simian Army，能够诱导服务发生故障，甚至能诱导一个数据中心在工作日发生故障，来测试该应用的弹性和监控能力。

这种在生产环境中所进行的自动化测试，足以让大多数运维组织兴奋得浑身颤栗，就像即将迎来一周的长假那样。这并不是说单块架构风格不能构建先进的监控系统——只是根据我们的经验，这在单块系统中并不常见罢了。

> ### “断路器”与“可随时上线的代码”
>
> “断路器”（Circuit Breaker）一词与其他一些模式一起出现在《发布！》（Release It!）一书中，例如隔板（Bulkhead）和超时（Timeout）。当构建彼此通信的应用系统时，将这些模式加以综合运用就变得至关重要。Netflix 公司的这篇很精彩的博客解释了这些模式是如何应用的。

因为各个服务可以在任何时候发生故障，所以下面两件事就变得很重要，即能够快速地检测出故障，而且在可能的情况下能够自动恢复服务。各个微服务的应用都将大量的精力放到了应用程序的实时监控上，来检查“架构元素指标”（例如数据库每秒收到多少请求）和“业务相关指标”（例如系统每分钟收到多少订单）。当系统某个地方出现问题，语义监控系统能提供一个预警，来触发开发团队进行后续的跟进和调查工作。

这对于一个微服务架构是尤其重要的，因为微服务对于服务编制（choreography）和事件协作（event collaboration）的偏好，会导致“突发行为”。尽管许多权威人士对于偶发事件的价值持积极态度，但事实上，“突发行为”有时是一件坏事。在能够快速发现有坏处的“突发行为”并进行修复的方面，监控是至关重要的。

单块系统也能构建得像微服务那样来实现透明的监控系统——实际上，它们也应该如此。差别是，绝对需要知道那些运行在不同进程中的服务，在何时断掉了。而如果在同一个进程内使用软件库的话，这种透明的监控系统就用处不大了。

> ### “同步调用”有害
>
> 一旦在一些服务之间进行多个同步调用，就会遇到宕机的乘法效应。简而言之，这意味着整个系统的宕机时间，是每一个单独模块各自宕机时间的乘积。此时面临着一个选择：是让模块之间的调用异步，还是去管理宕机时间？在英国卫报网站 www.guardian.co.uk，他们在新平台上实现了一个简单的规则——每一个用户请求都对应一个同步调用。然而在 Netflix 公司，他们重新设计的平台 API 将异步性构建到 API 的机制（fabric）中。

那些微服务团队希望在每一个单独的服务中，都能看到先进的监控和日志记录装置。例如显示“运行/宕机”状态的仪表盘，和各种运维和业务相关的指标。另外我们经常在工作中会碰到这样一些细节，即断路器的状态、当前的吞吐率和延迟，以及其他一些例子。

特性九：“演进式”设计

那些微服务的从业者们，通常具有演进式设计的背景，而且通常将服务的分解，视作一个额外的工具，来让应用开发人员能够控制应用系统中的变化，而无须减少变化的发生。变化控制并不一定意味着要减少变化——在正确的态度和工具的帮助下，就能在软件中让变化发生得频繁、快速且经过了良好的控制。

每当试图要将软件系统分解为各个组件时，就会面临这样的决策，即如何进行切分——我们决定切分应用系统时应该遵循的原则是什么？一个组件的关键属性，是具有独立更换和升级的特点[13]——这意味着，需要寻找这些点，即想象着能否在其中一个点上重写该组件，而无须影响该组件的其他合作组件。事实上，许多做微服务的团队会更进一步，他们明确地预期许多服务将来会报废，而不是守着这些服务做长期演进。

英国卫报网站是一个好例子。原先该网站是一个以单块系统的方式来设计和构建的应用系统，然而它已经开始向微服务方向进行演进了。原先的单块系统依旧是该网站的核心，但是在添加新特性时他们愿意以构建一些微服务的方式来进行添加，而这些微服务会去调用原先那个单块系统的 API。当在开发那些本身就带有临时性特点的新特性时，这种方法就特别方便，例如开发那些报道一个体育赛事的专门页面。当使用一些快速的开发语言时，像这样的网站页面就能被快速地整合起来。而一旦赛事结束，这样页面就可以被删除。在一个金融机构中，我们已经看到了一些相似的做法，即针对一个市场机会，一些新的服务可以被添加进来。然后在几个月甚至几周之后，这些新服务就作废了。

这种强调可更换性的特点，是模块化设计一般性原则的一个特例，通过“变化模式”（pattern of change）[14]来驱动进行模块化的实现。大家都愿意将那些能在同时发生变化的东西，放到同一个模块中。系统中那些很少发生变化的部分，应该被放到不同的服务中，以区别于那些当前正在经历大量变动（churn）的部分。如果发现需要同时反复变更两个服务时，这就是它们两个需要被合并的一个信号。

把一个个组件放入一个个服务中，增大了作出更加精细的软件发布计划的机会。对于一个单块系统，任何变化都需要做一次整个应用系统的全量构建和部署。然而，对于一个个微服务来说，只需要重新部署修改过的那些服务就够了。这能简化并加快发布过程。但缺点是：必须要考虑当一个服务发生变化时，依赖它并对其进行消费的其他服务将无法工作。传统的集成方法是试图使用版本化来解决这个问题。但在微服务世界中，大家更喜欢将版本化作为最后万不得已的手段来使用。我们可以通过下述方法来避免许多版本化的工作，即把各个服务设计得尽量能够容错，来应对其所依赖的服务所发生的变化。

未来的方向是“微服务”吗?

我们写这篇文章的主要目的，是来解释有关微服务的主要思路和原则。在花了一点时间做了这件事后，我们清楚地认识到，微服务架构风格是一个重要的理念——在研发企业应用系统时，值得对它进行认真考虑。我们最近已经使用这种风格构建了一些系统，并且了解到其他一些团队也在已经使用并赞同这种方法。

我们所了解到的那些在某种程度上做为这种架构风格的实践先驱包括：亚马逊、Netflix、英国卫报、英国政府数字化服务中心、realestate.com.au、Forward 和 comparethemarket.com。在 2013 年的技术大会圈子充满了各种各样的正在转向可归类为微服务的公司案例——包括 Travis CI。另外还有大量的组织，它们长期以来一直在做着我们可以归类为微服务的产品，却从未使用过这个名字（这通常被标记为 SOA ——尽管正如我们所说，SOA 会表现出各种自相矛盾的形式[15]）。

尽管有这些正面的经验，然而并不是说我们确信微服务是软件架构的未来的方向。尽管到目前为止，与单块应用系统相比，我们对于所经历过的微服务的评价是积极的，但是我们也意识到这样的事实，即能供我们做出完整判断的时间还不够长。

通常，架构决策所产生的真正效果，只有在该决策做出若干年后才能真正显现。我们已经看到由带着强烈的模块化愿望的优秀团队所做的一些项目，最终却构建出一个单块架构，并在几年之内不断腐化。许多人认为，如果使用微服务就不大可能出现这种腐化，因为服务的边界是明确的，而且难以随意搞乱。然而，对于那些开发时间足够长的各种系统，除非我们已经见识得足够多，否则我们无法真正评价微服务架构是如何成熟的。

> 我们的同事 Sam Newman 花了 2014 年的大部分时间撰写了一本书（“微服务设计”，豆瓣），来记述我们构建微服务的经验。如果想对这个话题深入下去，下一步就应该是阅读这本书。

有人觉得微服务或许很难成熟起来，这当然是有原因的。在组件化上所做的任何工作的成功与否，取决于软件与组件的匹配程度。准确地搞清楚某个组件的边界的位置应该出现在哪里，是一件困难的工作。演进式设计承认难以对边界进行正确定位，所以它将工作的重点放到了易于对边界进行重构之上。但是当各个组件成为各个进行远程通信的服务后，比起在单一进程内进行各个软件库之间的调用，此时的重构就变得更加困难。跨越服务边界的代码移动就变得困难起来。接口的任何变化，都需要在其各个参与者之间进行协调。向后兼容的层次也需要被添加进来。测试也会变得更加复杂。

另一个问题是，如果这些组件不能干净利落地组合成一个系统，那么所做的一切工作，仅仅是将组件内的复杂性转移到组件之间的连接之上。这样做的后果，不仅仅是将复杂性搬了家，它还将复杂性转移到那些不再明确且难以控制的边界之上。当在观察一个小型且简单的组件内部时，人们很容易觉得事情已经变得更好了，然而他们却忽视了服务之间杂乱的连接。

最后，还有一个团队技能的因素。新技术往往会被技术更加过硬的团队所采用。对于技术更加过硬的团队而更有效的一项技术，不一定适用于一个技术略逊一筹的团队。我们已经看到大量这样的案例，那些技术略逊一筹的团队构建出了杂乱的单块架构。当这种杂乱发生到微服务身上时，会出现什么情况？这需要花时间来观察。一个糟糕的团队，总会构建一个糟糕的系统——在这种情况下，很难讲微服务究竟是减少了杂乱，还是让事情变得更糟。

我们听到一个合理的说法，是说不要一上来就以微服务架构做为起点。相反，要用一个单块系统做为起点，并保持其模块化。当这个单块系统出现了问题后，再将其分解为微服务。（尽管这个建议并不理想，因为一个良好的单一进程内的接口，通常不是一个良好的服务接口。）

因此，我们持谨慎乐观的态度来撰写此文。到目前为止，我们已经看到足够多的有关微服务风格的事物，并且觉得这是一条有价值去跋涉的道路。我们不能肯定地说，道路的尽头在哪里。但是，软件开发的挑战之一，就是只能基于“目前手上拥有但还不够完善”的信息来做出决策。

参考资料

尽管下面不是一个详尽的清单，但是它们是微服务从业者们获取灵感的一些来源，或者是那些倡导的理念与本文所述内容相似的一些资料。

博客和线上文章

- Clemens Vasters' blog on cloud at microsoft
- David Morgantini's introduction to the topic on his blog
- 12 factor apps from Heroku
- UK Government Digital Service design principles
- Jimmy Nilsson's blog and article on infoq about Cloud Chunk Computing
- Alistair Cockburn on Hexagonal architectures

图书

- Release it，“发布！软件的设计与部署”，豆瓣
- Rest in practice，“REST实战”，豆瓣
- Web API Design (free ebook). Brian Mulloy, Apigee.
- Enterprise Integration Patterns，“企业集成模式”，豆瓣
- Art of unix programming，“UNIX编程艺术”，豆瓣
- Growing Object Oriented Software, Guided by Tests，“测试驱动的面向对象软件开发”，豆瓣
- The Modern Firm: Organizational Design for Performance and Growth
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation，“持续交付”，豆瓣
- Domain-Driven Design: Tackling Complexity in the Heart of Software，“领域驱动设计”，豆瓣

演讲

- Architecture without Architects. Erik Doernenburg.
- Does my bus look big in this?. Jim Webber and Martin Fowler, QCon 2008
- Guerilla SOA. Jim Webber, 2006
- Patterns of Effective Delivery. Daniel Terhorst-North, 2011.
- Adrian Cockcroft's slideshare channel.
- Hydras and Hypermedia. Ian Robinson, JavaZone 2010
- Justice will take a million intricate moves. Leonard Richardson, Qcon 2008.
- Java, the UNIX way. James Lewis, JavaZone 2012
- Micro services architecture. Fred George, YOW! 2012
- Democratising attention data at guardian.co.uk. Graham Tackley, GOTO Aarhus 2013
- Functional Reactive Programming with RxJava. Ben Christensen, GOTO Aarhus 2013 (registration required).
- Breaking the Monolith. Stefan Tilkov, May 2012.

论文

- L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", 1978 http://research.microsoft.com/en-us/um/people/lamport/pubs/implementation.pdf
- L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", 1982 (available at) http://www.cs.cornell.edu/courses/cs614/2004sp/papers/lsp82.pdf
- R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 2000 http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- E. A. Brewer, "Towards Robust Distributed Systems", 2000 http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed", 2012, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

---

1. 2011 年 5 月在威尼斯附近的一个软件架构工作坊中，大家开始讨论“微服务”（microservice）这个术语，因为这个词可以描述参会者们在架构领域进行探索时所见到的一种通用的架构风格。2012 年 5 月，这群参会者决定将“微服务”（microservices）作为描述这种架构风格的最贴切的名字。在 2012 年 3 月波兰的克拉科夫市举办的“33rd Degree”技术大会上，本文作者之一 James 在其“Microservices - Java, the Unix Way”演讲中以案例的形式谈到了这些微服务的观点，与此同时，Fred George 也表达了同样的观点。Netflix 公司的 Adrian Cockcroft 将这种方法描述为“细粒度的 SOA”，并且作为先行者和本文下面所提到的众人已经着手在 Web 领域进行了实践——Joe Walnes, Dan North, Evan Botcher 和 Graham Tackley。 ↩

2. “单块”（monolith）这个术语已经被 Unix 社区使用一段时间了。它出现在“UNIX编程艺术”（The Art of Unix Programming）一书中，来描述那些变得庞大的系统。 ↩

3. 许多面向对象的设计者，包括我们自己，都使用领域驱动设计中 service object 这个术语，来描述那种执行一段未被绑定到一个 entity 对象上的重要的逻辑过程的对象。这不同于本文所讨论的“service”的概念。可悲的是，service 这个术语同时具有这两个含义，我们必须忍受这样的多义词。 ↩

4. 我们认为一个应用系统是一个社会性的构建单元，来将一个代码库、功能组和资金体（body of funding）结合起来。 ↩

5. 原始论文参见梅尔文•康威的网站：http://www.melconway.com/Home/Committees_Paper.html ↩

6. 在极度强调高效性（scale）的情况下，一些组织经常会使用一些二进制的消息发送协议——例如 protobuf。即使是这样，这些系统仍然会呈现出“智能端点和傻瓜管道”的特点——来在易读性（transparency）与高效性之间取得平衡。当然，大多数 Web 属性和绝大多数企业并不需要作出这样的权衡——获得易读性就已经是一个很大的胜利了。 ↩

7. 忍不住要提一下 Jim Webber 的说法：ESB 表示 Egregious Spaghetti Box（一盒极烂的意大利面条）。 ↩

8. Netflix 让 SOA 与微服务之间的联系更加明确——直到最近这家公司还将他们的架构风格称为“细粒度的 SOA”（fine-grained SOA）。 ↩

9. "YAGNI" 或者 "You Aren't Going To Need It"（你不会需要它）是极限编程的一条原则和劝诫，指的是“除非到了需要的时候，否则不要添加新功能”。 ↩

10. 单块系统使用单一编程语言，这样讲有点言不由衷——为了在今天的 Web 上构建各种系统，可能要了解 JavaScript、XHTML、CSS、服务器端的编程语言、SQL 和一种 ORM 的方言。很难说只有一种单一编程语言，但是我们的意思你是懂得的。 ↩

11. Adrian Cockcroft 在他 2013 年 11 月于 Flowcon 技术大会所做的一次精彩的演讲中，特别提到了“开发人员自服务”和“开发人员运行他们写的东西”（原文如此）。 ↩

12. 这里我们又有点言不由衷了。很明显，在更复杂的网络拓扑里，部署更多的服务，会比部署一个单独的单块系统要更加困难。幸运的是，有一些模式能够减少其中的复杂性——但对于工具的投资还是必须的。 ↩

13. 事实上，Dan North 将这种架构风格称作“可更换的组件架构”，而不是微服务。因为这看起来似乎是在谈微服务特性的一个子集，所以我们选择将其归类为微服务。 ↩

14. Kent Beck 在《实现模式》（Implementation Patterns）一书中，将其作为他的一条设计原则而强调出来。 ↩

15. SOA 很难讲是这段历史的根源。当 SOA 这个词儿在本世纪初刚刚出现时，我记得有人说：“我们很多年以来一直是这样做的。”有一派观点说，SOA 这种风格，将企业级计算早期 COBOL 程序通过数据文件来进行通信的方式，视作自己的“根”。在另一个方向上，有人说“Erlang 编程模型”与微服务是同一回事，只不过它被应用到一个企业应用的上下文中去了。 ↩