微服务

一个新架构术语的定义

“微服务架构”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分散控制。

2014年3月25日

James Lewis

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

Martin Fowler是一位作家、演说家，也是软件开发领域具备影响力的人。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

分散治理

分散式数据管理

基础设施自动化

“容错”设计

演进式设计

微服务是未来吗?

拓展

一个微服务应该有多大?

微服务与SOA

多种语言，多种选择

实战检验的标准与强制执行的标准

让做正确的事变得容易

”断路器“和”可随时上线的代码“

同步调用被认为是有害的

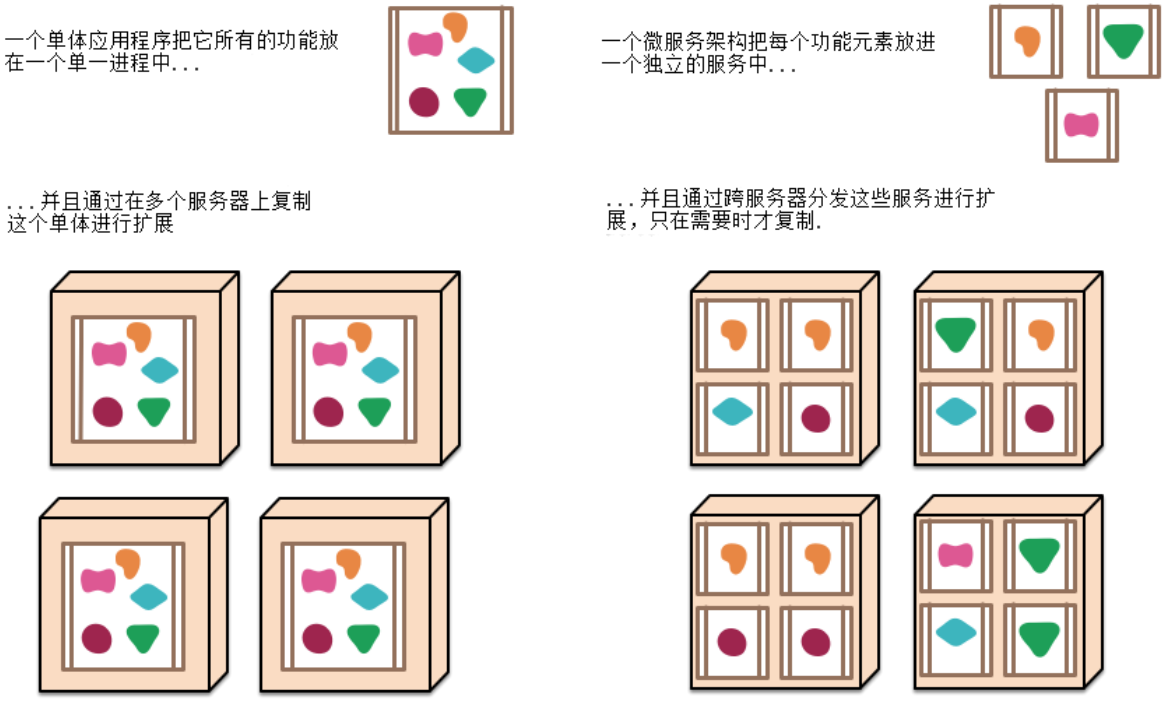
“微服务”——又一个出现在拥挤的软件架构领域的新术语。我们自然会对它投过轻蔑的一瞥，但是这个小小的术语却描述了一种愈发引人入胜的软件系统的风格。在过去的几年中，我们已经看到许多项目使用这种风格，到目前为止效果很好，以至于对于我们的许多同事来说，这种风格正在成为构建企业应用程序的默认风格。然而，遗憾的是，没有太多的信息概述什么是微服务风格以及如何实现它。

简而言之，微服务架构风格[1]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，每个服务在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）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。

在解释微服务风格之前，有必要将其与单体应用风格进行比较：单体应用是作为单个单元构建的应用程序。企业应用程序通常由三个主要部分组成：客户端用户界面（由在用户机器上的浏览器中运行的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）和服务器端应用程序。服务器端应用程序将处理HTTP请求、执行领域逻辑、从数据库检索和更新数据，并选择和填充要发送给浏览器的HTML视图。这个服务器端应用程序是一个整体——一个单一的逻辑可执行[2]。对系统的任何更改都涉及构建和部署服务器端应用程序的新版本。

这样的单体服务器是构建这样一个系统的自然方式。所有处理请求的逻辑都在一个进程中运行，允许你使用语言的基本特性将应用程序划分为类、函数和命名空间。只要稍加注意，就可以在开发人员的笔记本电脑上运行和测试应用程序，并使用部署流水线确保正确地测试了更改并将其部署到生产环境中。你可以通过在负载均衡器后面运行多个实例来水平扩展单体应用。

单块应用系统可以被成功地实现，但越来越多的人对单体应用感到失望——特别是随着越来越多的应用被部署到云端。更改周期被捆绑在一起——对应用程序的一小部分进行更改，就需要重新构建和部署整个单体应用程序。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。



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。除了服务是可独立部署和可扩展的这一事实之外，每个服务还提供了一个固定的模块边界，甚至允许用不同的编程语言编写不同的服务。这些服务也能被不同的团队来管理。

我们并不是说微服务风格新颖或创新，它的根源至少可以追溯到Unix的设计原则。但是我们确实认为没有足够的人考虑微服务架构，如果对其加以使用，许多软件的开发工作能变得更好。

微服务架构的特征

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。

通过服务划分实现系统组件化

自从我们涉足软件行业以来，就一直有一种愿望，就是通过将组件组装在一起来构建系统，就像我们在现实世界中看到的那样。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，组件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，服务则是进程外组件，它们通过web服务请求或远程过程调用等机制进行通信。（这不同于许多面向对象的程序中的service object概念[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软件库组成，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。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种情况发生。

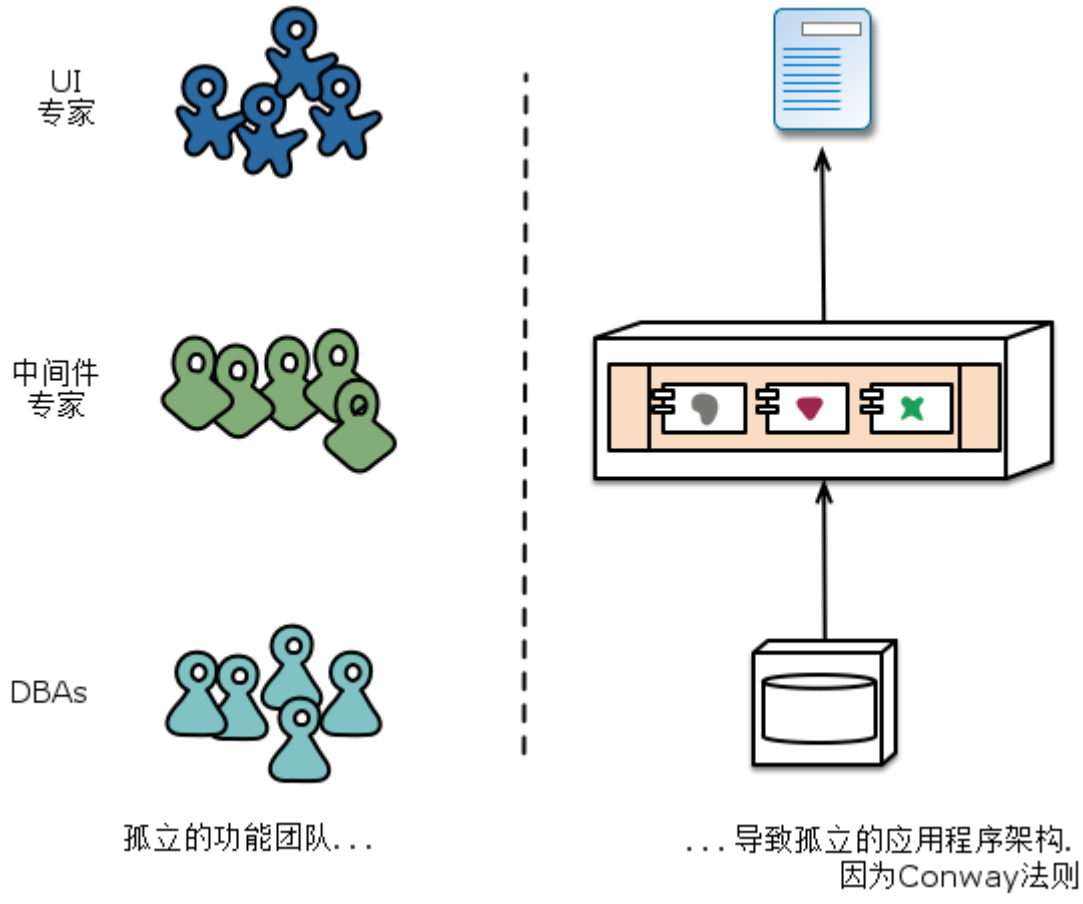
像这样使用服务确实有一些缺点，比起进程内调用，远程调用更加昂贵。所以远程调用API接口必须是粗粒度的，而这往往更加难以使用。如果需要修改组件间的职责分配，那么当跨越进程边界时，这种组件行为的改动会更加难以实现。

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。一个服务可能由多个总是一起开发和部署的进程组成，例如一个应用程序进程和一个仅由该服务使用的数据库。

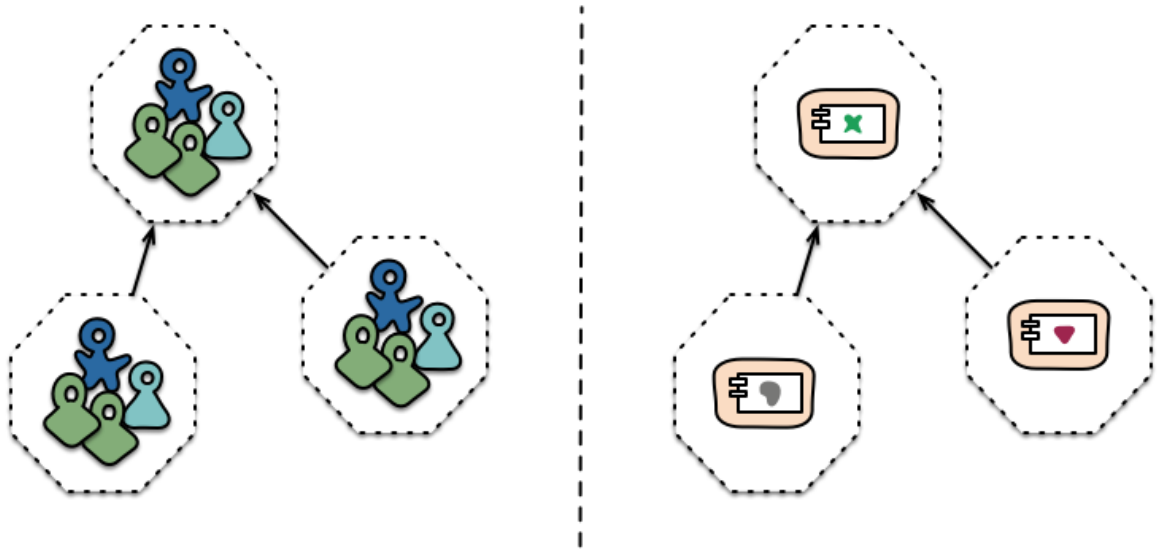
围绕业务功能组织团队

当在寻求将一个大型应用系统拆分成几部分时，通常管理层聚焦在技术层面，导致UI团队、服务器端逻辑团队和数据库团队的划分。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项目花费时间和预算审批。聪明的团队会围绕这一点进行优化，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。这是一个康威定律在起作用的一个例子。

> 任何设计系统（定义广泛）的组织，其设计的结构都与该组织的通信结构相一致。
>
> *-- Melvin Conway, 1968*



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。此类服务采用了该业务领域软件的广泛堆栈实现，包括用户界面、持久存储和任何外部协作。因此，团队是跨职能的，包括开发所需的所有技能：用户体验、数据库和项目管理。



跨功能团队...

...围绕业务能力组织的团队
根据Conway法则

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功能来模块化，虽然这不是常见的情况。当然，我们会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。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。如果单体横跨了多个模块边界，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。另外，我们看到模块化的路线需要大量的规则来强制实施。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。

> 拓展：一个微服务应该有多大?
>
> 尽管“微服务”已经成为这种架构风格的一个流行名称，但它的名称确实导致了对服务规模的关注，以及关于什么是“微”的争论。在我们与微服务从业者的对话中，我们看到了各种规模的服务。据报道，最大的披萨数量遵循亚马逊的两个披萨团队的概念（即整个团队可以吃两个披萨），这意味着不超过12人。在更小的规模上，我们看到一个6个人的团队可以支持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，我们当然有可能改变想法。

做产品而不是做项目

我们看到的大多数应用程序开发工作都使用项目模型：目标是交付一些软件，然后认为已经完成。软件完成后，将其移交给维护组织，并解散构建软件的项目团队。

微服务的支持者倾向于避免这种模式，他们倾向于认为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。对此的一个常见启发是亚马逊的"you build, you run it"概念，即开发团队对生产环境中的软件负全部责任。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，与业务功能的联系紧密。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。

> 拓展：微服务与 SOA

> 当我们谈到微服务时，一个常见的问题是，这是否是我们十年前看到的面向服务的架构（SOA）。这一点是有好处的，因为微服务风格非常类似于SOA的一些倡导者所支持的。然而，问题是SOA意味着太多不同的东西，而且大多数时候我们遇到的所谓的"SOA"与我们在这里描述的风格明显不同，这通常由于它们专注于ESB，来集成各个单体应用。
>
> 特别是，我们看到了如此多拙劣的面向服务实现——从将系统复杂性隐藏于ESB中的趋势[5]，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集中式治理模型——以至于有时觉得其所造成的种种问题真的不堪回首。
>
> 当然，微服务社区中使用的许多技术都来自于开发人员在大型组织中集成服务的经验。Tolerant Reader模式就是这样的一个例子。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SOA的常见表现形式导致一些微服务倡导者完全拒绝将自己贴上SOA的标签，尽管其他人认为微服务是SOA[6]的一种形式，也许面向服务的做法是正确的。无论如何，SOA意味着如此之多的不同事物，意味着有一个更清晰地定义这种体系结构风格的术语是有价值的。

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，来强调将大量的智能特性纳入通信机制本身。企业服务总线（Enterprise Service Bus, ESB）就是一个很好的例子，在ESB产品中通常为消息路由、编排、转化和应用业务规则引入高度智能的设施。

微服务社区倾向于另一种方法：智能端点（smart endpoints）和哑管道（dumb pipes）。由微服务构建的应用程序的目标是尽可能地实现"高内聚和低耦合"——它们拥有自己的领域逻辑，更像是经典Unix意义上的过滤器——接收请求，对其应用业务逻辑，并产生一个响应。它们使用简单的RESTish协议进行编排，而不是复杂的协议，如WS-Choreography或BPEL或通过一个中心工具进行编排（orchestration）。

最常用的两种协议是带有资源API的HTTP请求-响应协议和轻量级消息传递[7]协议。对于前一种协议的最佳表述是：

> 本身就是web，而不是隐藏在web的后面。
>
> -- Ian Robinson

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）。从开发者和运维人员的角度讲，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（仅仅像消息路由器所做的事情那样傻瓜）——像RabbitMQ或ZeroMQ这样的简单实现，除了提供可靠的异步机制（fabric）之外，并没有做更多的事情——智能仍然体现在生产和消费消息的端点上，即存在于各个服务中。

在单体应用中，各个组件在同一个进程中运行，它们之间的通信通过方法调用或函数调用。将单体应用改成微服务的最大问题在于通信模式的改变。从内存中的方法调用到RPC的简单转换会导致通信繁琐，性能不佳。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。

分散治理

集中式治理的后果之一是单一技术平台的标准化趋势。经验表明，这种方法是有局限性的——不是每个问题都是钉子，不是每个解决方案都是锤子。我们更喜欢使用正确的工具，虽然单体应用程序可以在一定程度上利用不同的语言，但这种情况并不常见。

如果能将单体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。你想用Node.js创建一个简单的报表页面？大胆尝试吧。用C++来实现一个特别复杂的近实时组件？很好。你想换一种更适合某个组件读取行为的数据库？我们有重建他的技术。

当然，仅仅因为您可以做一些事情，并不意味着您应该做——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。

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。与其使用一套写在纸上的定义好的标准，他们更喜欢开发有用的工具，让其他开发人员可以使用它们来解决他们面临的类似问题。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，有时，但不仅限于使用内部开源模型。现在Git和GitHub已经成为事实上的版本控制系统。在企业内部，开源的做法正在变得越来越普遍。

Netflix就是遵循这一理念的一个很好的例子。首先，将有用的、经过实战考验的代码作为软件库来分享，可以鼓励其他开发人员以类似的方式解决类似的问题，但如果需要，也为选择不同的方法敞开了大门。共享软件库倾向于关注数据存储、进程间通信等常见问题，以及我们在下文进一步讨论的基础设施自动化。

对于微服务社区来说，开销尤其没有吸引力。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“Tolerant Reader”和“Consumer-Driven Contracts”这样的模式经常应用于微服务。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。事实上，我们知道澳大利亚有一个团队，他们使用消费者驱动的契约来构建新服务。他们使用简单的工具来定义服务的契约。在编写新服务的代码之前，这就成为自动构建的一部分。然后，服务只被构建到满足契约的程度——这是一种在构建新软件时避免“YAGNI”[8]困境的优雅方法。这些技术和围绕它们发展起来的工具通过减少服务之间的时域（temporal）耦合限制了对集中契约管理的需求。

> 拓展：多种语言，多种选择
>
> 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。几十年来，使用更高级别的语言来利用更高级别的抽象已经成为一种常见的做法。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，许多单体应用并不需要这种级别的性能优化，DSL和更高级别的抽象也不常见（这令我们感到失望）。相反，许多单体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]。

也许分散治理的巅峰就是亚马逊的“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。这些想法与传统的集中式治理模式相去甚远。

分散式数据管理

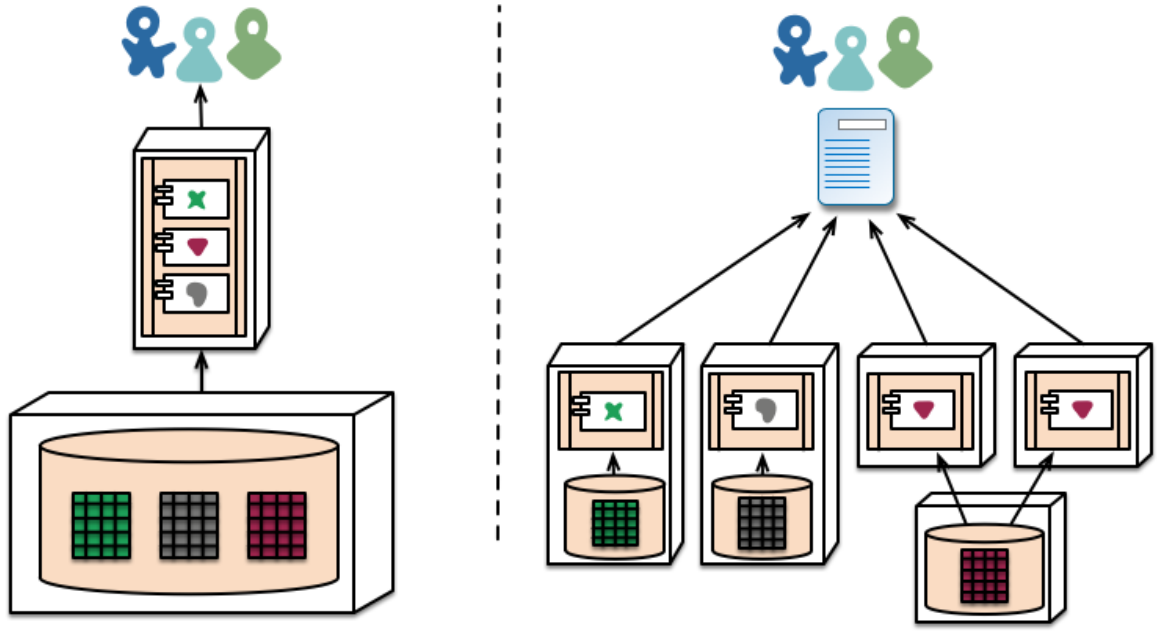
数据管理的分散化以多种不同的方式表现出来。在最抽象的层次上，这意味着世界的概念模型将因系统而异。当跨大型企业进行集成时，这是一个常见的问题，客户的销售视图将与支持视图不同。一些在销售视图中被称为客户的东西可能根本不会出现在支持视图中。它们可能具有不同的属性，（更糟糕的是）那些在两个视角中具有相同属性的事物，或许在语义上有微妙的不同。

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中的“限界上下文”的概念。DDD将复杂域划分为多个有界上下文，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体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。

> 拓展：实战检验的标准与强制执行的标准
>
> 这有点像二分法，微服务团队倾向于避免企业架构团队制定的那种严格执行的标准，但会乐于使用甚至宣传使用开放标准，如HTTP、ATOM和其他微格式。

> 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。
>
> 这些标准与企业世界中的许多标准截然不同，企业世界中的许多标准通常是由最近几乎没有编程经验或过度受供应商影响的团队开发的。

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。虽然单体应用程序更喜欢使用单一的逻辑数据库来持久化数据，但企业通常更喜欢跨一系列应用程序使用单一数据库——许多决策都是由供应商围绕许可的商业模型驱动的。微服务更喜欢让每个服务管理自己的数据库，要么是相同数据库技术的不同实例，要么是完全不同的数据库系统——这种方法称为多语言持久化。你可以在单体应用中使用多语言持久化，但它在微服务中出现得更频繁。



单体 – 单一数据库　　微服务 – 应用程序数据库

跨微服务分散数据责任对管理更新有影响。处理更新的常用方法是使用事务来保证更新多个资源时的一致性。这种方法经常用于单体应用中。

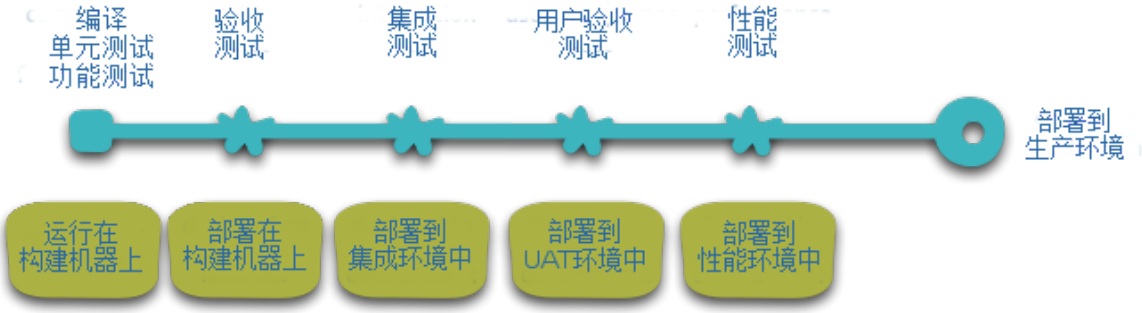
像这样使用事务有助于保持一致性，但会带来严重的时域耦合，这会在多个服务之间造成问题。众所周知，分布式事务很难实现，因此微服务架构强调服务之间的无事务协调，明确认识到一致性可能只是最终的一致性，并且一致性问题能够通过补偿操作来进行处理。

选择以这种方式管理不一致性对许多开发团队来说是一个新的挑战，但它通常与业务实践相匹配。为了快速响应需求，企业通常会处理一定程度的不一致性，同时有某种逆转过程来处理错误。只要修复错误的成本低于在更大的一致性下失去业务的成本，这种权衡是值得的。

基础设施自动化

基础设施自动化技术在过去几年中有了很大的发展——云的发展，特别是AWS，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或系统都是由具有丰富持续交付经验的团队构建的，持续交付是持续集成的前身。以这种方式构建软件的团队可以广泛地使用基础设施自动化技术。如下图的构建流水线所示：

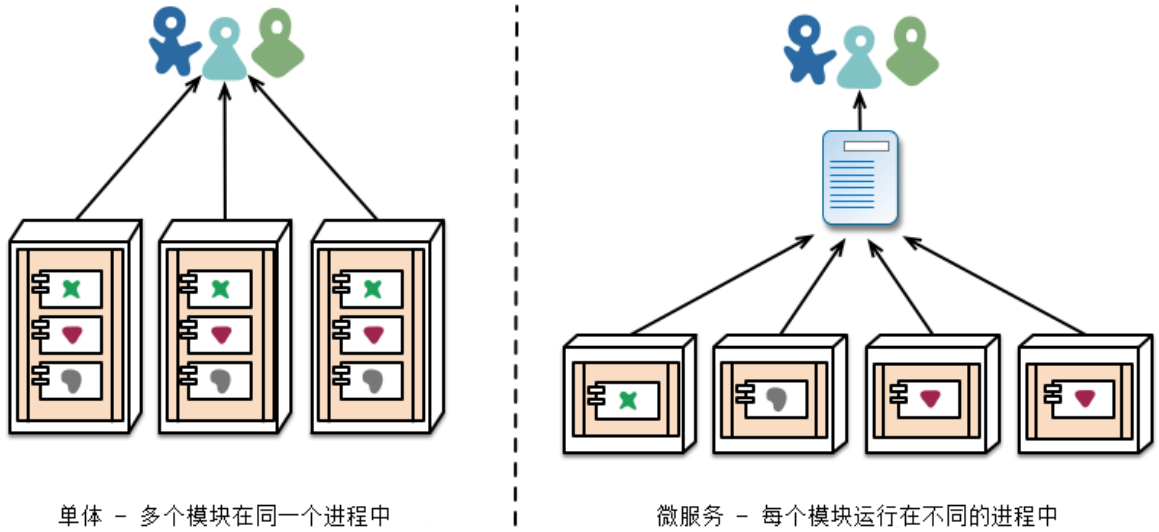


由于这不是一篇关于持续交付的文章，我们将在这里提请注意几个关键特性。为了尽可能地获得对正在运行的软件的信心，需要运行大量的自动化测试。让可工作的软件达到“晋级”（Promotion）状态从而“推上”流水线，就意味着可以在每一个新的环境中，对软件进行自动化部署。

在这些环境中，我们可以轻松地构建、测试和运行单体应用程序。事实证明，一旦在下述工作上进行了投入，即针对一个单体应用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[11]。

> 拓展：让做正确的事变得容易
>
> 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，是创建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。与前面我们对比单体应用和微服务所说的正相反，只要部署工作很无聊，那么在这一点上单体应用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。



单体 – 多个模块在同一个进程中

微服务 – 每个模块运行在不同的进程中

“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设计成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。客户端必须尽可能优雅地响应。与单体式设计相比，这是一个缺点，因为这会引人额外的复杂性来处理这种情况。因此，微服务团队不断反思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，以测试应用程序的弹性和监控能力。

这种在生产环境中所进行的自动化测试，能足以让大多数运维组织兴奋得浑身颤栗，就像在一周的长假即将到来前那样。这并不是说单体架构风格无法实现复杂的监控设置——只是在我们的经验中不太常见。

> 拓展："断路器"和"可随时上线的代码"
>
> 断路器与其他模式如Bulkhead和Timeout出现在《Release it!》中。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的这篇博客很好地解释了这些模式是如何应用的。

由于服务随时可能出现故障，因此能够快速检测故障并在可能的情况下自动恢复服务非常重要。微服务应用程序非常强调应用程序的实时监控，检查架构元素指标（数据库每秒接收多少个请求）和业务相关指标（例如每分钟接收多少个订单）。语义监控可以提供一个早期预警系统，提醒开发团队进行后续跟进和调查。

这对微服务架构特别重要，因为微服务对服务编排和事件协作的偏好会导致"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，偶然出现的行为有时可能是一件坏事。监控是至关重要的，可以快速发现不良的突发行为，以便修复。

单体应用也能构建得像微服务那样来实现透明的监控系统——事实上，它们也应该如此。区别在于，您绝对需要知道运行在不同进程中的服务何时断开连接。在同一个进程中使用的软件库中，这种透明的监控系统就用处不大了。

> 拓展：同步调用被认为是有害的
>
> 任何时候，当您在服务之间有许多同步调用时，您将遇到宕机带来的倍增效应。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，他们重新设计的平台API将异步性构建到API的机制中。

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。

进化式设计

微服务从业者通常具有进化式的设计背景，并将服务分解视为一种进一步的工具，使应用程序开发人员能够控制应用系统中的变更，而无须减少变更的发生。变更控制并不一定意味着要减少变更——通过正确的态度和工具，您可以对软件进行频繁、快速和良好控制的变更。

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？组件的关键特性是具有独立更换和升级的特点[12]——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。

《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体应用的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。

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"[13]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。

将组件放入服务为更细粒度的发布计划提供了机会。对于单体应用，任何更改都需要完整地构建和部署整个应用程序。然而，使用微服务时，您只需要重新部署您修改的服务。这可以简化和加快发布过程。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。传统的集成方法是尝试使用版本控制来处理这个问题，但在微服务世界中，首选的是将版本控制作为最后的手段。我们可以避免大量的版本管理，通过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。

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——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（这通常被称为SOA——虽然，正如我们说过的，SOA会表现出各种自相矛盾的形式。[14]）

尽管有这些正面的经验，但是，然而并不是说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，架构决策的真实结果只有在做出决策几年后才会显现出来。我们曾经看到过这样的项目：一个对模块化有着强烈需求的优秀团队构建了一个单体架构，但随着时间的推移，这个架构已经衰落了。许多人认为，微服务不太可能出现这种衰退，因为服务边界是明确的，并且很难围绕它打补丁。然而，在我们看到足够多的系统和足够长的时间之前，我们无法真正评估微服务架构的成熟程度。

我们当然有理由认为微服务不成熟。在组件化的任何努力中，成功与否取决于软件与组件的契合程度。很难准确地确定组件的边界应该在哪里。进化式设计认识到正确定义边界的困难，因此也认识到易于重构边界的重要性。但是，当你的组件是具有远程通信的服务时，重构要比使用进程内软件库困难得多。跨越服务边界的代码移动就变得困难起来。任何接口的更改都需要在参与者之间进行协调，需要添加向后兼容层，测试也变得更加复杂。

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。这不仅仅是将复杂性转移，它将复杂性转移到一个不那么明确且难以控制的边界之上。当你看到一个小的、简单的组件的内部，而忽略了服务之间混乱的连接时，很容易认为事情会更好。

最后，还有团队技能的因素。更熟练的团队倾向于采用新技术。但是，对于技术较强的团队更有效的技术并不一定适用于技术较差的团队。我们已经见过很多不太熟练的团队构建混乱的单体架构的案例，但我们需要时间来了解当这种混乱在微服务中发生时会发生什么。一个糟糕的团队总是会创建一个糟糕的系统——在这种情况下，很难判断微服务是减少了混乱还是使其变得更糟。

我们听到的一个合理的论点是，你不应该从微服务架构做为起点。相反，从单体应用做为起点，保持模块化，一旦单体应用出现问题，再将其拆分为微服务。（虽然这个建议并不理想，因为良好的进程内接口通常不是良好的服务接口。）

因此，我们持谨慎乐观的态度来撰写此文。到目前为止，我们已经对微服务风格有了足够的了解，觉得这是一条值得走的道路。我们不能确定我们将在哪里结束，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。

附注

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，Fred George也差不多在同一时间提出。Netflix的Adrian Cockcroft把这种方法描述为“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：单体这一术语已被Unix社区使用了一段时间，在《Unix编程艺术》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：我们不得不提到Jim Webber的说法，即ESB代表 "一盒极烂的意大利面条（Erroneous Spaghetti Box）"。

6：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”。

7：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“智能端点和哑管道”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。

8："YAGNI" 或 "You Aren't Going To Need It" 是极限编程的一个原则，告诫人们不要添加功能，除非你知道你需要它们。

9：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，但你知道我们的意思。

10：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）。

11：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。

12：事实上，Daniel Terhorst-North 将这种风格称为“可替换组件架构”，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。

13：Kent Beck在《实现模式》一书中，将其作为他的一条设计原则而强调出来。

14：SOA很难讲是这段历史的根源。当SOA这个词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。

参考文献

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。

博客和在线文章

- 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 blogand article on infoq about Cloud Chunk Computing
- Alistair Cockburn on Hexagonal architectures

书籍

- Release it
- Rest in practice
- Web API Design (free ebook). Brian Mulloy, Apigee.
- Enterprise Integration Patterns
- Art of unix programming
- Growing Object Oriented Software, Guided by Tests
- The Modern Firm: Organizational Design for Performance and Growth
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
- Domain-Driven Design: Tackling Complexity in the Heart of Software

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

原文：Microservices (martinfowler.com)

参考：“微服务”博客中译完整版