

目錄

简介	1.1
1.微服务简介	1.2
2.构建微服务：使用API网关	1.3
3.构建微服务：微服务架构中的进程间通信	1.4
4.微服务架构中的服务发现	1.5
5.事件驱动的数据管理微服务	1.6
6.选择微服务部署策略	1.7
7.重组重构为微服务	1.8

微服务：从设计到部署

本文所有内容翻译自：[nginx](#)

1. 微服务简介
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的英文文章集，以及使用[NGINX Plus](#)实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

微服务目前得到了很多关注：文章，博客，社交媒体讨论和会议演讲。他们迅速走向[Gartner炒作周期](#)的高峰期。与此同时，软件界也有怀疑者，认为微服务没有什么新意。[Naysayers](#)声称这个想法只是[SOA](#)的一个重塑。然而，尽管有炒作和怀疑，[微服务架构模式](#)具有显着的好处 - 特别是当涉及到实现敏捷开发和交付复杂的企业应用程序。

这七个系列的文章现在完成：

1. 构建微服务：使用API网关
2. 构建微服务：微服务架构中的进程间通信
3. 微服务架构中的服务发现
4. 事件驱动的数据管理微服务
5. 选择微服务部署策略
6. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

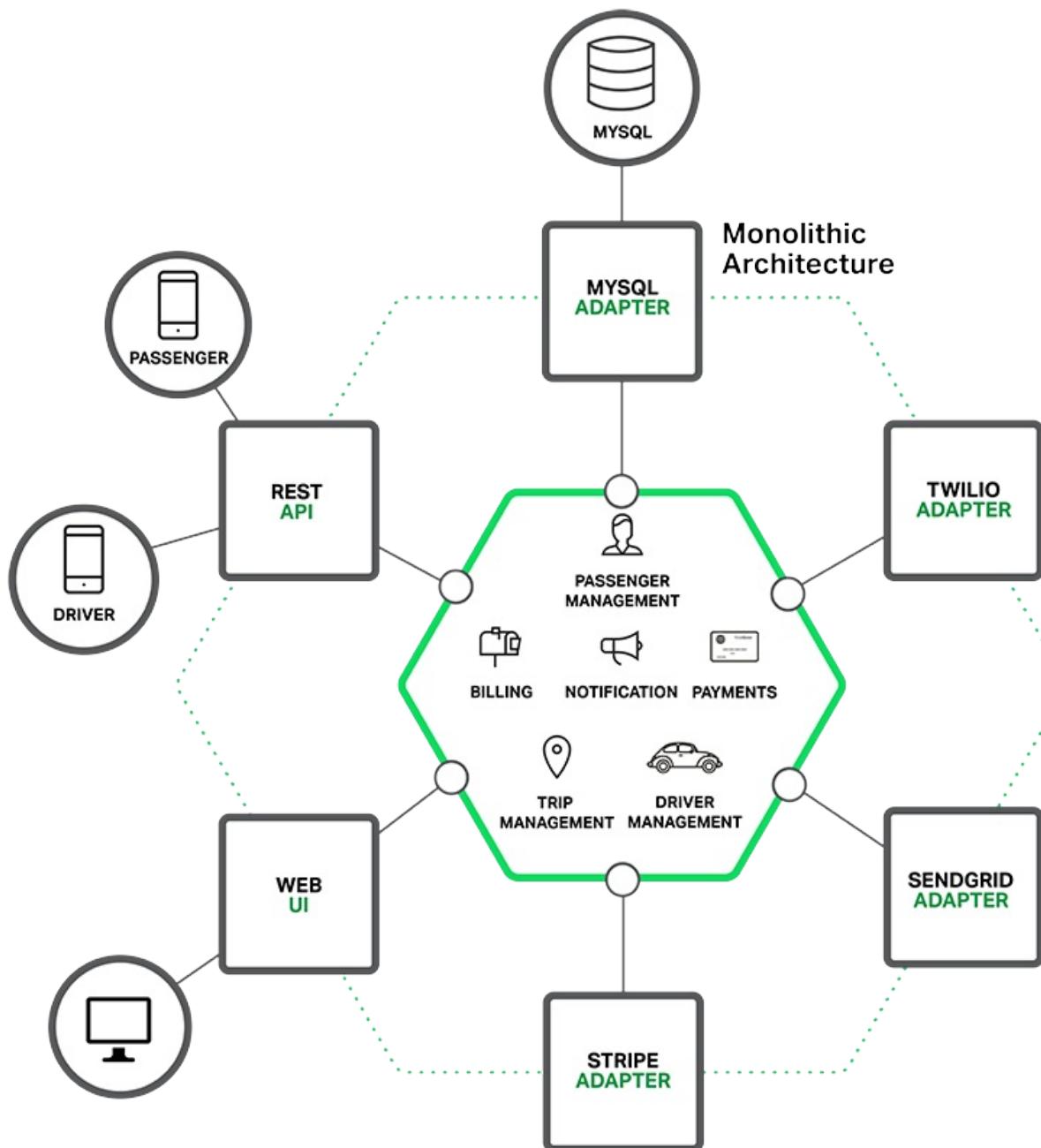
微服务目前得到了很多关注：文章，博客，社交媒体讨论和会议演讲。他们迅速走向Gartner炒作周期的高峰期。与此同时，软件界也有怀疑者，认为微服务没有什么新意。Naysayers声称这个想法只是SOA的一个重塑。然而，尽管有炒作和怀疑，微服务架构模式具有显着的好处 - 特别是当涉及到实现敏捷开发和交付复杂的企业应用程序。

这篇博文是关于设计，构建和部署微服务的七部分系列中的第一篇。您将了解该方法以及如何与更传统的单片建筑模式进行比较。本系列将描述微服务架构的各种元素。您将了解Microservices Architecture模式的优点和缺点，无论它对您的项目有意义，以及如何应用它。

让我们先来看看为什么你应该考虑使用微服务。

建筑单片应用

让我们想象你开始建立一个全新的出租车应用程序，旨在与Uber和Hailo竞争。在一些初步会议和需求收集之后，您可以手动创建一个新项目，或者使用Rails，Spring Boot，Play或Maven附带的一个生成器。这个新的应用程序将具有模块化的六角形架构，如下图所示：



应用程序的核心是业务逻辑，它由定义服务，域对象和事件的模块实现。围绕核心的是与外部世界接口的适配器。适配器的示例包括数据库访问组件，生成和使用消息的消息传递组件以及公开API或实现UI的Web组件。

尽管具有逻辑模块化架构，但应用程序被打包并部署为一个整体。实际格式取决于应用程序的语言和框架。例如，许多Java应用程序打包为WAR文件，并部署在应用程序服务器（如Tomcat或Jetty）上。其他Java应用程序打包为自包含可执行JAR。类似地，Rails和Node.js应用程序打包为目录层次结构。

以这种风格编写的应用程序是非常常见的。它们很容易开发，因为我们的IDE和其他工具专注于构建单个应用程序。这些类型的应用程序也很容易测试。您可以通过简单地启动应用程序并使用Selenium测试UI来实现端到端测试。单片应用也很容易部署。您只需将打包的应用程

序复制到服务器。您还可以通过在负载平衡器后运行多个副本来自扩展应用程序。在项目的早期阶段，它工作得很好。

走向单片地狱

不幸的是，这种简单的方法有很大的局限性。成功地应用程序有一个随着时间增长的习惯，并最终变得巨大。在每个sprint期间，您的开发团队实现了更多的故事，这当然意味着添加许多代码行。经过几年，你的小，简单的应用程序将成长为一个巨大的巨石。为了给出极端的例子，我最近谈到一个开发人员，他正在编写一个工具来分析其数百万行代码（LOC）应用程序中的数千个JAR之间的依赖关系。我相信它花了许多开发人员多年来一起努力创造这样的野兽。

一旦你的应用程序成为一个庞大，复杂的单片，你的开发组织可能在一个痛苦的世界。任何敏捷开发和交付的尝试都会岌岌可危。一个主要的问题是应用程序是非常复杂的。对于任何一个开发者来说，它太大了，不能完全理解。因此，修正错误和实现新功能正确变得困难和耗时。更重要的是，这往往是一个向下的螺旋。如果代码库很难理解，那么更改将不会正确。你最终会得到一个怪异，不可理解的大泥球。

应用程序的庞大规模也会减慢开发速度。应用程序越大，启动时间越长。例如，在最近的一项调查中，一些开发人员报告启动时间长达12分钟。我也听说过应用程序需要长达40分钟才能启动的轶事。如果开发人员经常需要重新启动应用程序服务器，那么他们大部分的时间都会等待，他们的生产力将受到影响。

大的，复杂的单片应用的另一个问题是它是连续部署的障碍。今天，SaaS应用程序的最新技术是每天将更改推入生产多次。这对于复杂的整体来说是非常困难的，因为您必须重新部署整个应用程序以更新它的任何一个部分。我之前提到的冗长的启动时间也不会有帮助。此外，由于更改的影响通常不是很好理解，很可能您必须进行广泛的手动测试。因此，连续部署是不可能做到的。

当不同模块具有冲突的资源需求时，单片应用也可能难以扩展。例如，一个模块可能实现CPU密集型图像处理逻辑，并且将理想地部署在Amazon EC2计算优化实例中。另一个模块可能是内存数据库，最适合用于EC2内存优化实例。但是，由于这些模块一起部署，您必须妥协选择硬件。

单片应用的另一个问题是可靠性。因为所有模块都在同一进程中运行，任何模块中的错误（例如内存泄漏）都可能会导致整个进程崩溃。此外，由于应用程序的所有实例是相同的，该错误将影响整个应用程序的可用性。

最后但并非最不重要的是，单片应用程序使得采用新的框架和语言非常困难。例如，让我们想象你有两百万行代码使用XYZ框架。将整个应用程序重写以使用较新的ABC框架将是非常昂贵的（在时间和成本上），即使该框架明显更好。因此，采用新技术有巨大的障碍。你在项目开始时所做的任何技术选择都被困住了。

总而言之：你有一个成功的关键业务应用程序已经成长为一个巨大的巨型，很少，如果说有的话，开发人员理解。它是使用过时的，非生产性的技术，使招聘有才华的开发人员困难。该应用程序难以扩展，并且不可靠。因此，敏捷开发和应用程序的交付是不可能的。

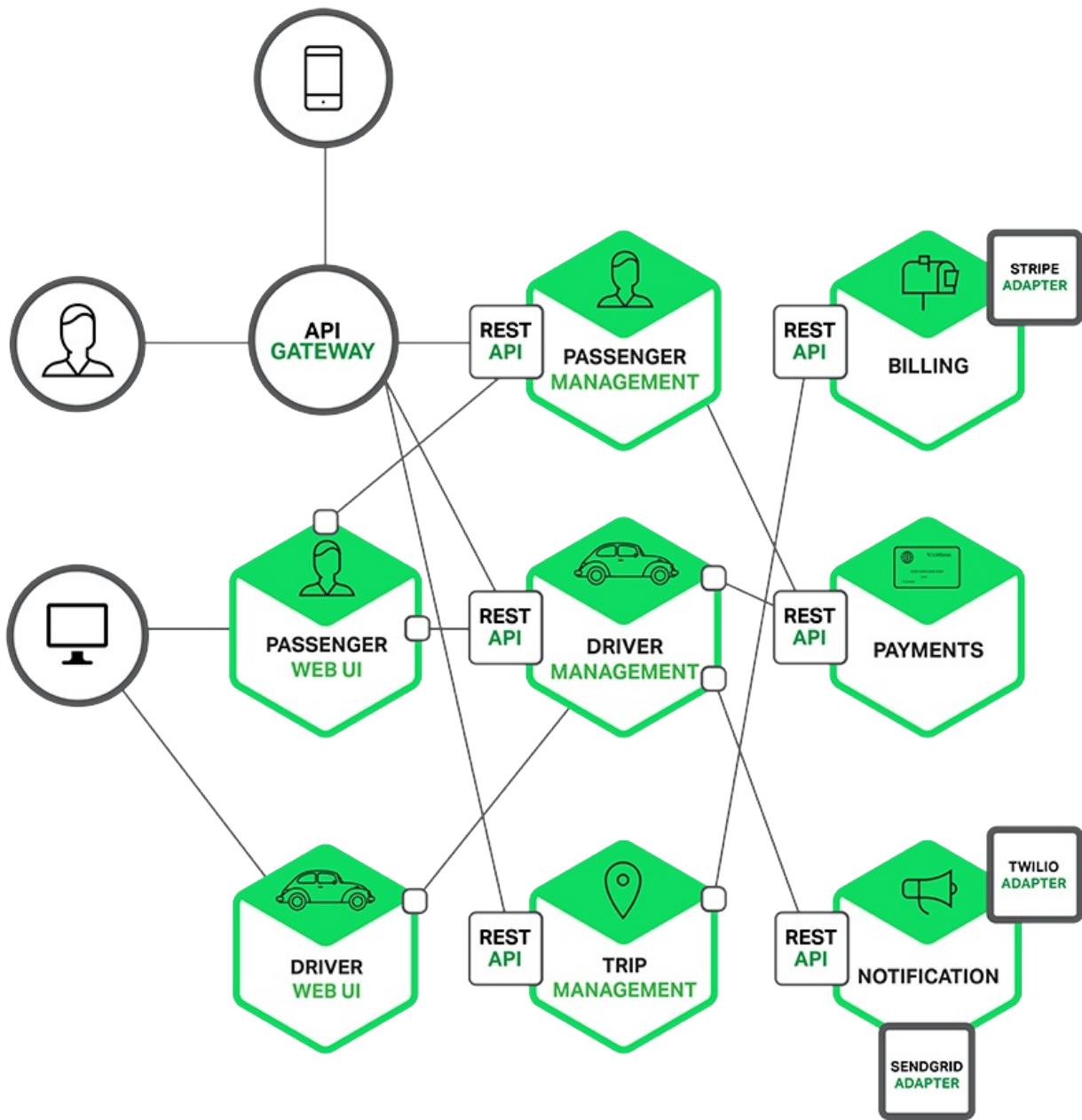
那么你能做什么呢？

微服务 - 解决复杂性

许多组织，如亚马逊，eBay和Netflix，通过采用现在所谓的[微服务架构模式](#)解决了这个问题。而不是构建一个怪异的，单一的应用程序，这个想法是将您的应用程序分成一组较小的，互连的服务。

服务通常实现一组不同的特征或功能，例如订单管理，客户管理等。每个微服务是具有其自己的六边形架构的小型应用，该六角形架构包括业务逻辑以及各种适配器。一些微服务会暴露其他微服务或应用程序客户端消耗的API。其他微服务可能实现Web UI。在运行时，每个实例通常是云VM或Docker容器。

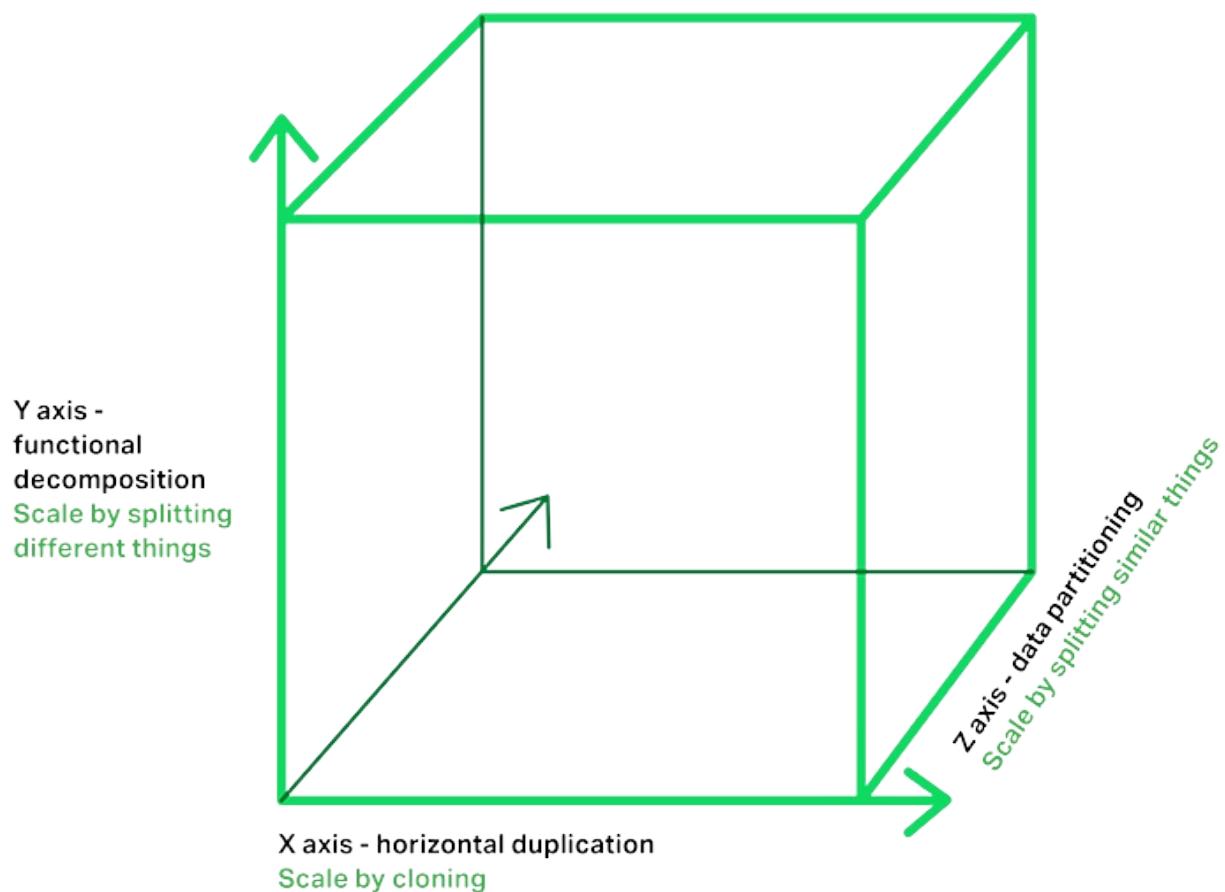
例如，上述系统的可能分解如下图所示：



应用程序的每个功能区域现在由其自己的微服务实现。此外，Web应用程序被分成一组更简单的Web应用程序（例如一个用于乘客和一个用于我们出租车示例中的司机）。这使得更容易为特定用户，设备或专用用例部署不同的体验。

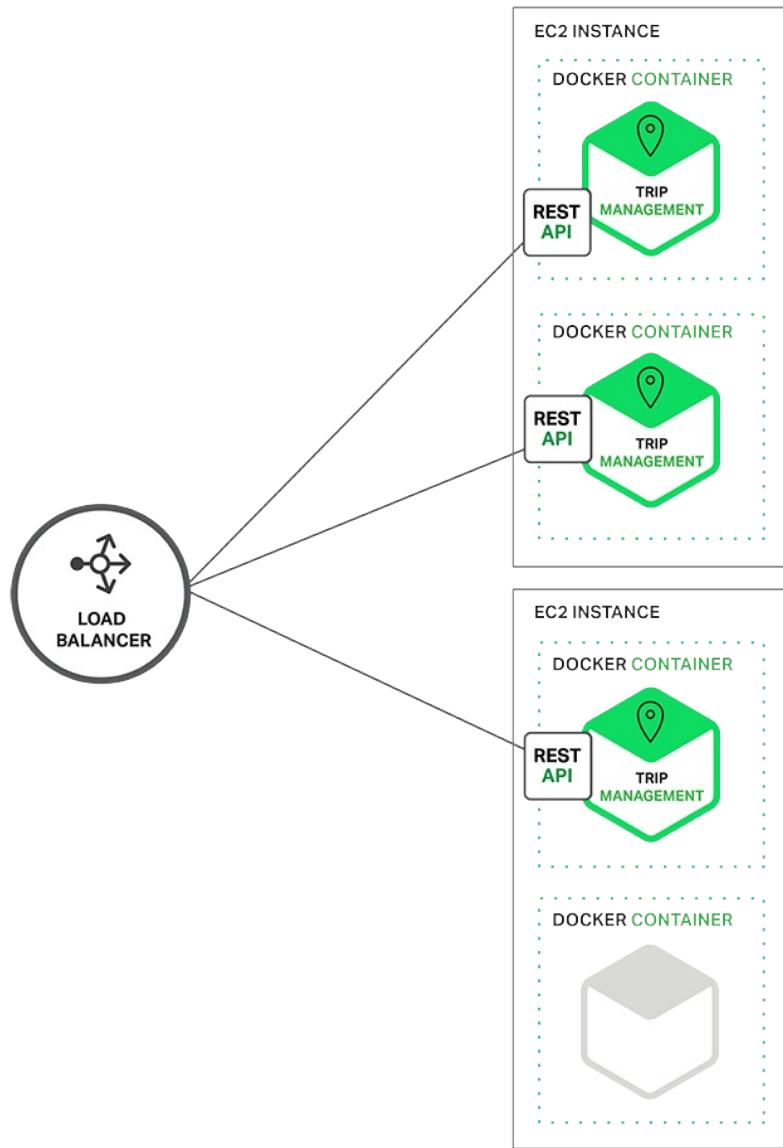
每个后端服务公开一个REST API，大多数服务使用其他服务提供的API。例如，驱动程序管理使用通知服务器告诉可用的驱动程序有关潜在的行程。UI服务调用其他服务以便呈现网页。服务还可以使用异步的基于消息的通信。服务间通信将在本系列后面更详细地介绍。

一些REST API还暴露给驱动程序和乘客使用的移动应用程序。但是，应用程序不能直接访问后端服务。相反，通信由被称为[API网关](#)的中介调解。API网关负责负载平衡，缓存，访问控制，API计量和监控等任务，可以使用[NGINX](#)有效实施。本系列的后续文章将涵盖[API网关](#)。



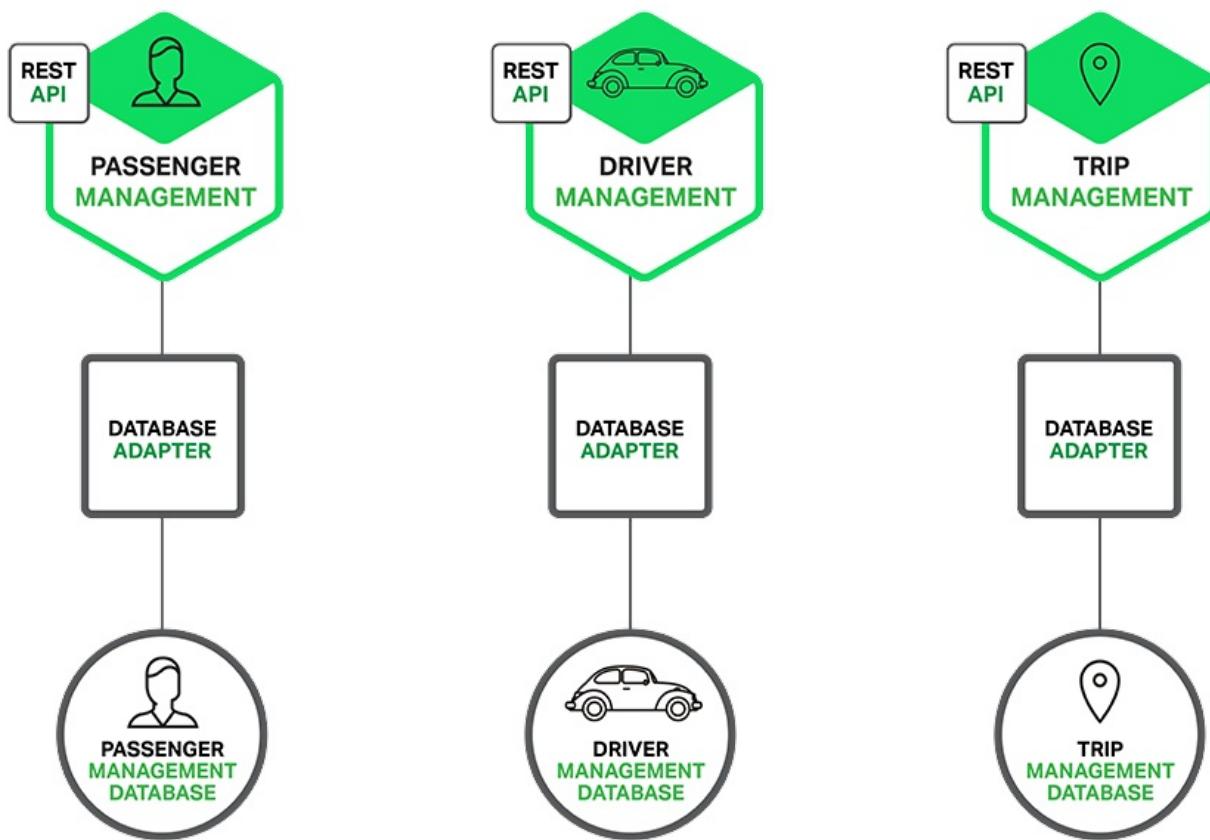
微服务架构模式对应于缩放立方体的Y轴缩放，缩放立方体是来自优秀的可扩展性艺术的可扩展性的3D模型。其他两个缩放轴是X轴缩放，其包括在负载均衡器后运行应用程序的多个相同副本，以及Z轴缩放（或数据分区），其中请求的属性（例如，主键的行或客户的身份）用于将请求路由到特定服务器。

应用程序通常一起使用三种类型的缩放。Y轴缩放将应用程序分解为微服务，如上面本节第一幅图所示。在运行时，X轴扩展在负载均衡器后运行每个服务的多个实例，以实现吞吐量和可用性。某些应用程序还可能使用Z轴缩放来分割服务。下图显示了在Amazon EC2上运行的Docker如何部署Trip Management服务。



在运行时，Trip Management服务由多个服务实例组成。每个服务实例都是一个Docker容器。为了实现高可用性，容器在多个云VM上运行。在服务实例前面是负载均衡器，例如NGINX，通过实例分配请求。负载均衡器还可以处理其他问题，如缓存，访问控制，API计量和监控。

微服务架构模式显著影响应用程序和数据库之间的关系。不是与其他服务共享单个数据库模式，每个服务都有自己的数据库模式。一方面，这种方法与企业级数据模型的想法不一致。此外，它经常导致一些数据的重复。但是，如果您希望从微服务中受益，则必须为每个服务提供数据库模式，因为它确保了松散耦合。下图显示了示例应用程序的数据库体系结构。



每个服务都有自己的数据库。此外，服务可以使用最适合其需要的类型的数据库，所谓的多语言持久性架构。例如，查找靠近潜在乘客的驱动程序的驱动程序管理必须使用支持高效地理查询的数据库。

表面上，微服务架构模式类似于SOA。使用这两种方法，该体系结构包括一组服务。然而，考虑微服务架构模式的一种方式是它是没有商业化和[Web服务规范（WS- *）](#) 和企业服务总线（ESB）的感知行为的SOA。基于微服务的应用程序更喜欢简单，轻量级的协议，如REST，而不是WS- *。他们也非常避免使用ESB，而是在微服务本身中实现类似ESB的功能。微服务架构模式也拒绝SOA的其他部分，例如规范模式的概念。

微服务的好处

微服务架构模式有很多重要的好处。首先，它解决复杂性的问题。它将一个怪异的单片应用程序分解为一组服务。虽然功能的总量不变，但应用程序已分解为可管理的块或服务。每个服务具有以RPC或消息驱动的API的形式的明确定义的边界。微服务架构模式实施了一个模块化级别，在实践中使用单片代码库极难实现。因此，个别服务的开发速度更快，更容易理解和维护。

第二，这种架构使得每个服务能够由专注于该服务的团队独立开发。开发人员可以自由选择任何技术有意义，前提是该服务符合API合同。当然，大多数组织希望避免完全无政府状态并限制技术选择。然而，这种自由意味着开发人员不再需要使用在新项目开始时存在的可能过

时的技术。当编写新服务时，他们可以选择使用当前技术。此外，由于服务相对较小，使用当前技术重写旧服务变得可行。

第三，微服务架构模式使每个微服务能够独立部署。开发人员从来不需要协调对他们的服务本地的更改的部署。这些类型的更改可以在测试完成后立即部署。UI团队可以，例如，执行A|B测试，并快速迭代UI更改。微服务架构模式使得连续部署成为可能。

最后，微服务架构模式使每个服务能够独立扩展。您只能部署满足其容量和可用性约束的每个服务的实例数。此外，您可以使用最符合服务资源要求的硬件。例如，您可以在EC2计算优化实例上部署CPU密集型图像处理服务，并在EC2内存优化实例上部署内存数据库服务。

微服务的缺点

正如弗雷德布鲁克斯几乎30年前写的那样，没有银子弹。像其他技术一样，Microservices架构也有缺点。一个缺点是名称本身。微服务一词过分强调服务大小。事实上，有些开发人员主张构建极其细粒度的10-100 LOC服务。虽然小型服务是更好的，但重要的是要记住，这是一种手段，而不是主要目标。微服务的目标是充分分解应用程序，以便于敏捷应用程序的开发和部署。

微服务的另一个主要缺点是微服务应用程序是分布式系统的事实导致的复杂性。开发人员需要选择和实现基于消息传递或RPC的进程间通信机制。此外，他们还必须编写代码来处理部分失败，因为请求的目的地可能很慢或不可用。虽然这些都不是火箭科学，但它比在单片应用中复杂得多，在单片应用中模块通过语言级方法/过程调用彼此调用。

微服务的另一个挑战是分区数据库架构。更新多个业务实体的业务事务很常见。这些类型的事务在单片应用程序中实现很简单，因为存在单个数据库。但是，在基于微服务的应用程序中，您需要更新由不同服务所拥有的多个数据库。使用分布式事务通常不是一个选项，而且不仅仅是因为CAP定理。它们根本不受许多当今高度可扩展的NoSQL数据库和消息传递代理的支持。您最终必须使用最终一致性方法，这对开发人员更具挑战性。

测试微服务应用程序也要复杂得多。例如，使用诸如Spring Boot的现代框架，编写用于启动单片Web应用程序并测试其REST API的测试类非常简单。相反，服务的类似测试类将需要启动该服务及其所依赖的任何服务（或至少为这些服务配置存根）。再次，这不是火箭科学，但重要的是不要低估这样做的复杂性。

微服务架构模式的另一个主要挑战是实施跨多个服务的更改。例如，假设您正在实现一个需要更改服务A，B和C的故事，其中A取决于B，B取决于C。在单片应用程序中，您可以简单地更改相应的模块，集成更改，并一次部署它们。相比之下，在微服务架构模式中，您需要仔细规划和协调对每个服务的更改的推出。例如，您需要更新服务C，然后是服务B，最后是服务A。幸运的是，大多数更改通常只影响一个服务，需要协调的多服务更改相对较少。

部署基于微服务的应用程序也要复杂得多。单个应用程序只是部署在传统负载均衡器后面的一组相同的服务器上。每个应用程序实例都配置有基础架构服务（如数据库和消息代理）的位置（主机和端口）。相反，微服务应用通常由大量服务组成。例如，根据Adrian Cockcroft，Hailo有160种不同的服务，Netflix有超过600种。每个服务将有多个运行时实例。这是需要配置，部署，扩展和监控的更多移动部件。此外，您还需要实现一个服务发现机制（在后面的文章中讨论），使服务能够发现与其通信所需的任何其他服务的位置（主机和端口）。传统的故障单和基于手动的操作方法无法扩展到这种复杂程度。因此，成功部署微服务应用程序需要开发人员更好地控制部署方法以及高水平的自动化。

一种自动化的方法是使用现成的PaaS，如Cloud Foundry。PaaS为开发人员提供了一种轻松部署和管理其微服务的方法。它使他们免受诸如采购和配置IT资源等问题的影响。同时，配置PaaS的系统和网络专业人员可以确保遵守最佳做法和公司策略。自动化微服务部署的另一种方法是开发基本上是您自己的PaaS。一个典型的起点是使用诸如Kubernetes的集群解决方案，结合诸如Docker的技术。在本系列的后面，我们将看看如何基于软件的应用程序交付方法，如NGINX，轻松处理缓存，访问控制，

概要

构建复杂应用程序本质上很困难。单片式架构仅对简单，轻量级应用程序有意义。如果你将它用于复杂的应用程序，你将会陷入一个痛苦的世界。微服务架构模式是复杂的，不断发展的应用程序的更好的选择，尽管有缺点和实施挑战。

在后面的博客文章中，我将深入了解微服务架构模式的各个方面的细节，并讨论诸如服务发现，服务部署选项以及将整体应用程序重构为服务的策略等主题。

敬请关注...

编辑 - 这七个系列的文章现在完成：

1. 微服务简介（本文）
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

1. 微服务介绍
2. 构建微服务：使用API网关（本文）
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

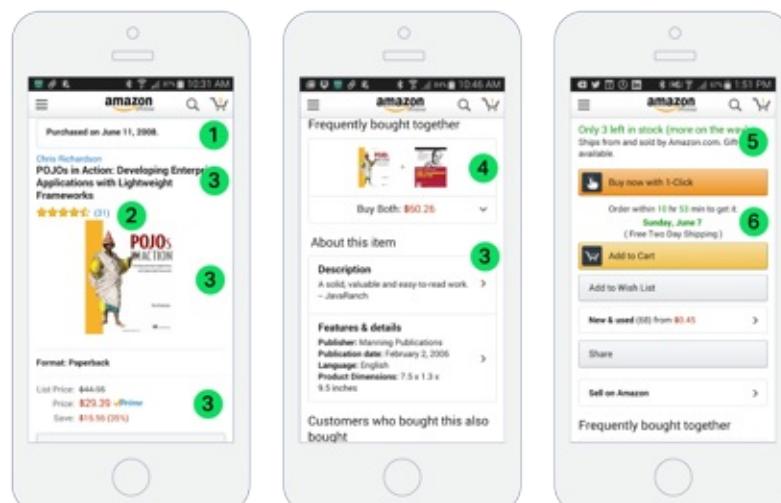
在这七部分组成的系列第一篇文章中有关设计，构建和部署微服务推出的微服务架构模式。它讨论了使用微服务的优点和缺点，以及尽管微服务的复杂性，它们通常是复杂应用程序的理想选择。这是本系列的第二篇文章，将讨论使用API网关构建微服务。

当您选择将应用程序构建为一组微服务时，您需要决定应用程序的客户端将如何与微服务进行交互。对于单片应用程序，只有一组（通常是复制的，负载平衡的）端点。然而，在微服务架构中，每个微服务暴露一组通常是细粒度端点的。在本文中，我们研究这如何影响客户端到应用程序的通信，并提出使用API网关的方法。

介绍

让我们假设您正在为购物应用程序开发本地移动客户端。您可能需要实施产品详情页面，其中显示有关任何指定产品的信息。

例如，下图显示了在Amazon的Android移动应用程序中滚动浏览产品详细信息时将会看到的内容。



1. ORDER HISTORY
2. REVIEWS
3. BASIC PRODUCT INFO
4. RECOMMENDATION
5. INVENTORY
6. SHIPPING

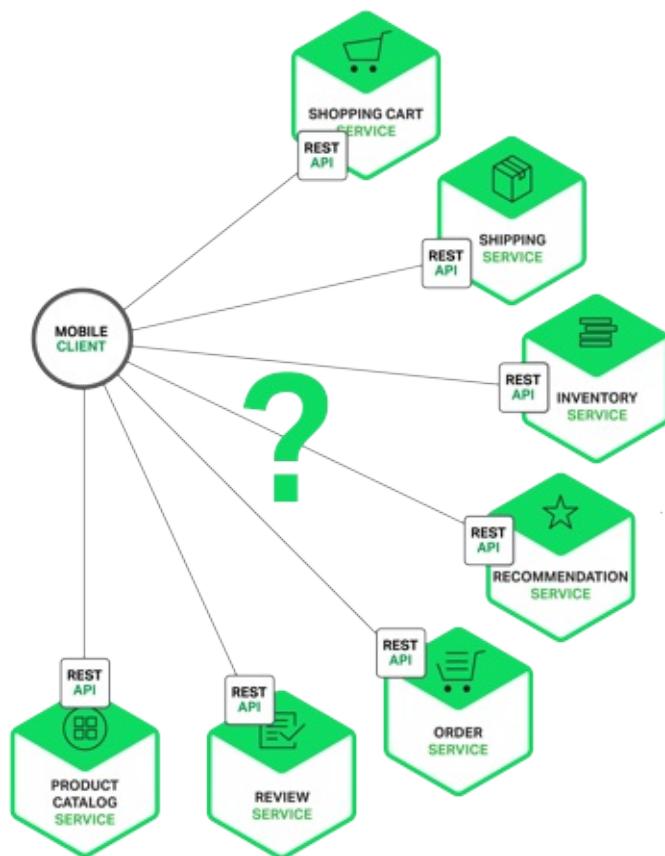
即使这是一个智能手机应用程序，产品详细信息页面显示了很多信息。例如，不仅有基本的产品信息（如名称，说明和价格），但此页面还显示：

- 购物车中的商品数量
- 订单历史
- 顾客评论
- 低库存警告
- 运输选项
- 各种建议，包括该产品经常购买的其他产品，购买此产品的客户购买的其他产品，以及购买此产品的客户查看的其他产品
- 其他采购选择

当使用单片应用程序架构时，移动客户端将通过对应用程序进行单个REST调用（）来检索此数据。负载平衡器将请求路由到N个相同的应用程序实例之一。然后应用程序将查询各种数据库表，并将响应返回给客户端。`GETapi.company.com/productdetails/productId`

相反，当使用微服务体系结构时，在产品详细信息页面上显示的数据由多个微服务拥有。以下是一些潜在的微服务，它们拥有示例产品详细信息页面上显示的数据：

- 购物车服务 - 购物车中的商品数量
- 订单服务 - 订单历史记录
- 目录服务 - 基本产品信息，如其名称，图像和价格
- 评论服务 - 客户评论
- 库存服务 - 低库存警告
- 装运服务 - 装运选项，截止日期和费用与装运提供商的API分开绘制
- 推荐服务 - 建议的项目



我们需要决定移动客户端如何访问这些服务。让我们看看选项。

直接客户端到微服务通信

在理论上，客户端可以直接向每个微服务器发出请求。每个微服务都有一个公共端点（<https://serviceName.api.company.name>）。此URL将映射到微服务的负载均衡器，负载均衡器在可用实例之间分配请求。要检索产品详细信息，移动客户端将向上面列出的每个服务发出请求。

不幸的是，这个选项有挑战和限制。一个问题是客户端的需求与每个微服务公开的细粒度API之间的不匹配。在这个例子中的客户端必须做七个单独的请求。在更复杂的应用程序中，它可能需要做更多。例如，Amazon描述了如何在呈现其产品页面中涉及数百种服务。虽然客户端可以通过LAN进行这么多请求，但是它可能在公共互联网上太低效，并且对于移动网络肯定是不切实际的。这种方法也使得客户端代码更复杂。

客户端直接调用微服务的另一个问题是，有些人可能使用不友好的网络协议。一个服务可能使用Thrift二进制RPC，而另一个服务可能使用AMQP消息传递协议。这两种协议都不是浏览器或防火墙友好的，最好在内部使用。应用程序应在防火墙外使用HTTP和WebSocket等协议。

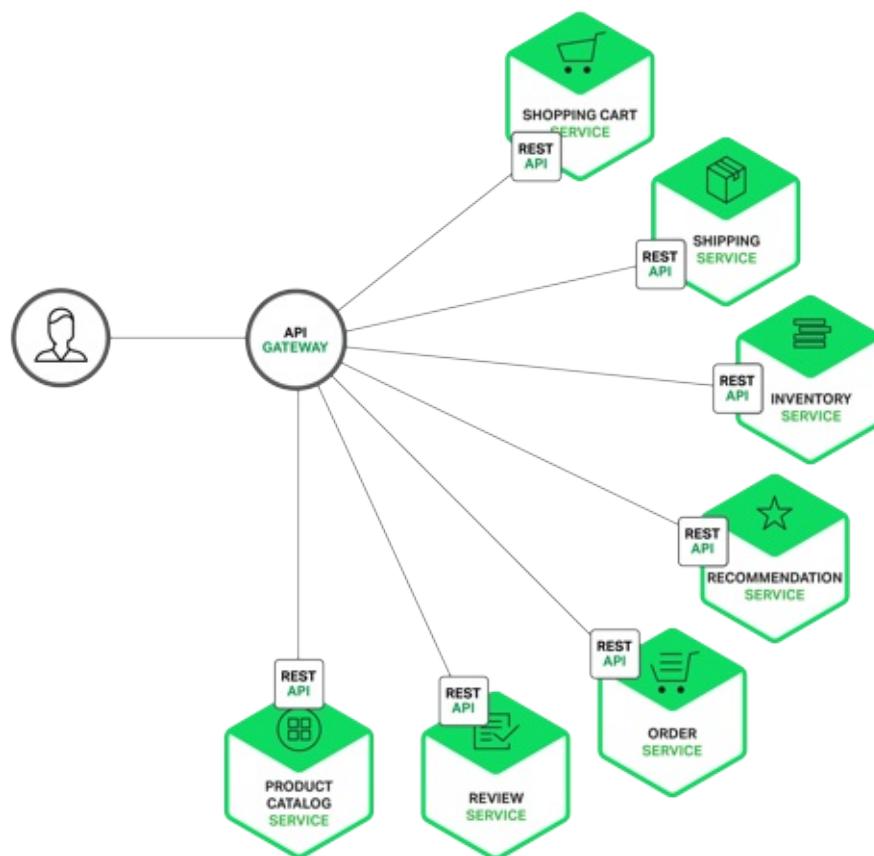
这种方法的另一个缺点是它使重构微服务变得困难。随着时间的推移，我们可能希望改变系统如何划分为服务。例如，我们可以合并两个服务或将服务分为两个或更多服务。然而，如果客户直接与服务通信，则执行这种重构可能是非常困难的。

由于这些类型的问题，客户端很少直接与微服务交谈。

使用API网关

通常一个更好的方法是使用所谓的**API网关**。API网关是一个进入系统的单一入口点的服务器。它类似于面向对象设计的**门面模式**。API网关封装了内部系统架构，并提供了针对每个客户端定制的**API**。它可能具有其他职责，例如认证，监视，负载平衡，缓存，请求整形和管理以及静态响应处理。

下图显示了API网关通常如何适应该架构：



API网关负责请求路由，组合和协议转换。来自客户端的所有请求首先通过API网关。然后它将请求路由到适当的微服务。API网关通常会通过调用多个微服务并聚合结果来处理请求。它可以在Web协议（如HTTP和WebSocket）和内部使用的Web不友好协议之间进行转换。

API网关还可以为每个客户端提供自定义API。它通常为移动客户端提供粗粒度的API。例如，考虑产品详细情况。API网关可以提供端点（`/productdetails?productid=xxx`），使移动客户端能够通过单个请求检索所有产品详细信息。API网关通过调用各种服务（产品信息，建

议，评论等）并结合结果来处理请求。

API网关的一个很好的例子是[Netflix API网关](#)。Netflix流媒体服务可用于数百种不同类型的设备，包括电视，机顶盒，智能手机，游戏系统，平板电脑等。最初，Netflix试图为其流媒体服务提供一种[一刀切的API](#)。然而，他们发现，它不能很好地工作，因为各种各样的设备及其独特的需求。今天，他们使用API网关，通过运行特定于设备的适配器代码为每个设备提供定制的API。适配器通常通过调用平均六到七个后端服务来处理每个请求。Netflix API网关每天处理数十亿次请求。

API网关的优点和缺点

正如您所期望的，使用API网关具有优点和缺点。使用API网关的一个主要好处是它封装了应用程序的内部结构。客户端只需要与网关通话，而不必调用特定的服务。API网关为每种类型的客户端提供特定的API。这减少了客户端和应用程序之间的往返次数。它还简化了客户端代码。

API网关也有一些缺点。它是另一个高度可用的组件，必须开发，部署和管理。还有一个风险是API网关成为开发瓶颈。开发人员必须更新API网关才能公开每个微服务的端点。重要的是，更新API网关的过程尽可能轻量级。否则，开发人员将被迫排队等待以更新网关。尽管有这些缺点，然而，对于大多数现实世界的应用程序，使用API网关是有意义的。

实现API网关

现在我们已经研究了使用API网关的动机和权衡，让我们来看看您需要考虑的各种设计问题。

性能和可扩展性

只有少数公司在Netflix的规模运营，每天需要处理数十亿的请求。然而，对于大多数应用程序，API网关的性能和可扩展性通常非常重要。因此，在支持异步，非阻塞I/O的平台上构建API网关是有意义的。有多种不同的技术可以用于实现可扩展的API网关。在JVM上，您可以使用基于NIO的框架之一，如Netty，Vertx，Spring Reactor或JBoss Undertow。一个流行的非JVM选项是Node.js，它是一个基于Chrome的JavaScript引擎的平台。另一个选择是使用[NGINX Plus](#)。NGINX Plus提供了一个成熟的，可扩展的高性能Web服务器和反向代理，易于部署，配置和编程。

使用反应式编程模型

API网关通过将它们简单地路由到适当的后端服务来处理一些请求。它通过调用多个后端服务并聚合结果来处理其他请求。对于某些请求，例如产品详细信息请求，对后端服务的请求是彼此独立的。为了最小化响应时间，API网关应同时执行独立请求。然而，有时，请求之间存

在依赖关系。API网关可能首先需要在将请求路由到后端服务之前通过调用身份验证服务来验证请求。类似地，为了获取关于顾客的愿望清单中的产品的信息，API网关必须首先检索包含该信息的顾客的简档，然后检索每个产品的信息。

使用传统的异步回调方法编写API组合代码快速导致回调地狱。代码将会纠缠，难以理解，并且容易出错。一个更好的方法是使用反应式方法以声明样式编写API网关代码。反应抽象的例子包括[未来在Scala中](#)，[CompletableFuture在Java中8](#)，并[承诺在JavaScript中](#)。还有[反应式扩展](#)（也称为Rx或ReactiveX），它最初是由Microsoft为.NET平台开发的。Netflix为专门在其API网关中使用的JVM创建了RxJava。还有用于JavaScript的RxJS，它在浏览器和Node.js中都运行。

服务调用

基于微服务的应用是分布式系统，并且必须使用进程间通信机制。有两种风格的进程间通信。一个选项是使用基于消息传递的异步机制。一些实现使用消息代理，例如[JMS](#)或[AMQP](#)。其他，如[ZeroMQ](#)，是无代理的，服务直接通信。另一种类型的进程间通信是诸如[HTTP](#)或[Thrift](#)之类的同步机制。系统通常使用异步和同步样式。它甚至可以使用每个样式的多个实现。因此，API网关将需要支持各种通信机制。

服务发现

API网关需要知道它与之通信的每个微服务的位置（IP地址和端口）。在传统的应用程序中，您可能可以将位置固定连接，但在现代的基于云的微服务应用程序中，这是一个非常重要的问题。基础设施服务（例如消息代理）通常具有静态位置，可以通过OS环境变量指定。然而，确定应用服务的位置不是那么容易。应用程序服务具有动态分配的位置。此外，服务的实例集合由于自动缩放和升级而动态地改变。因此，API网关与系统中的任何其他服务客户端一样需要使用系统的服务发现机制：[服务器端发现](#)或[客户端发现](#)。一个[以后的文章中](#)会详细描述了服务发现。现在，值得注意的是，如果系统使用客户端发现，那么API网关必须能够查询[服务注册表](#)，它是所有微服务实例及其位置的数据库。

处理部分故障

您在实现API网关时必须解决的另一个问题是部分失败的问题。每当一个服务调用另一个响应缓慢或不可用的服务时，在所有分布式系统中都会出现此问题。API网关不应阻止无限期地等待下游服务。然而，它如何处理故障取决于具体情况和哪个服务失败。例如，如果推荐服务在产品细节场景中没有响应，则API网关应将其余产品细节返回给客户端，因为它们对用户仍然有用。建议可以是空的，或者由例如硬连接的十大列表替换。然而，如果产品信息服务无响应，那么API网关应该向客户端返回错误。

API网关还可以返回缓存数据（如果可用）。例如，由于产品价格很少更改，如果定价服务不可用，API网关可以返回高速缓存的定价数据。数据可以由API网关本身缓存或存储在外部缓存（如Redis或Memcached）中。通过返回默认数据或缓存数据，API网关确保系统故障不会影响用户体验。

[Netflix Hystrix](#)是一个非常有用的库，用于编写调用远程服务的代码。Hystrix会超出超过指定阈值的呼叫。它实现了断路器模式，其阻止客户端不必要的等待无响应的服务。如果服务的错误率超过指定的阈值，Hystrix将断开断路器，并且所有请求将在指定的时间段内立即失败。Hystrix允许您在请求失败时定义回退操作，例如从高速缓存读取或返回默认值。如果你使用JVM，你一定要考虑使用Hystrix。并且，如果您在非JVM环境中运行，则应使用等效库。

概要

对于大多数基于微服务的应用程序，实现一个API网关是有意义的，它作为进入系统的单个入口点。API网关负责请求路由，组合和协议转换。它为每个应用程序的客户端提供自定义API。API网关还可以通过返回缓存或默认数据来屏蔽后端服务中的故障。在本系列的下一篇文章中，我们将介绍服务之间的通信。

编辑 - 这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关（本文）
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

这七个系列的文章现在完成：

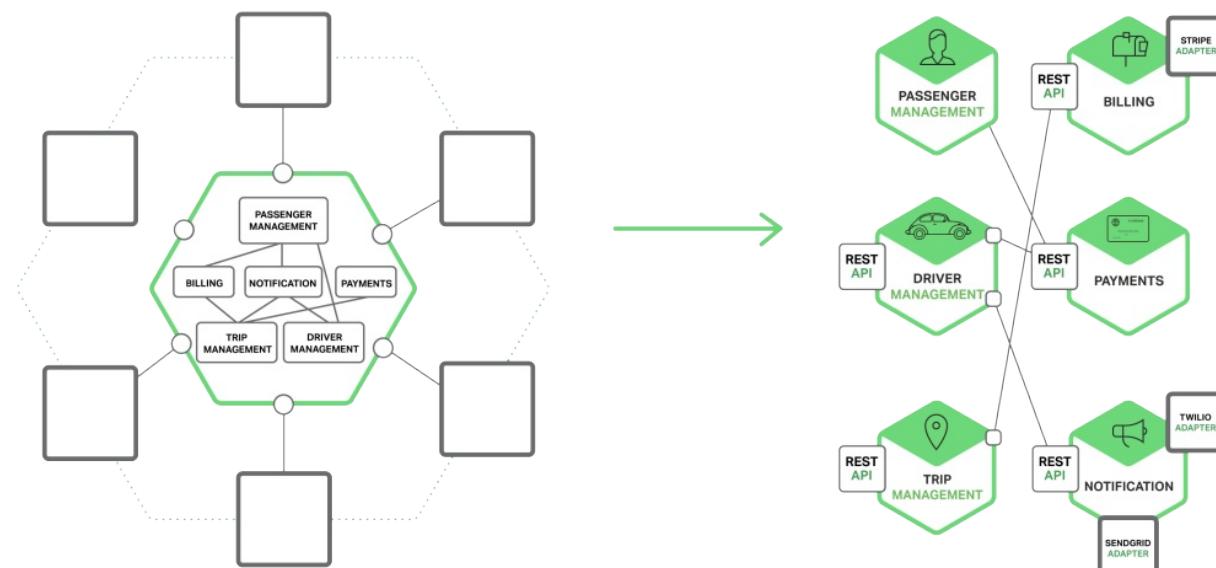
1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信（本文）
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这是我们关于使用微服务架构构建应用程序的第三篇文章。在[第一篇文章](#)介绍了[微服务架构模式](#)，它与比较[单片架构模式](#)，并讨论了好处和使用微服务的缺点。在[第二篇文章](#)介绍了如何应用程序的客户端通过被称为中介与微服务通信[API网关](#)。在本文中，我们将了解系统中的服务如何相互通信。在[第四篇文章](#)探讨了服务发现的密切相关的问题。

介绍

在单片应用程序中，组件通过语言级方法或函数调用互相调用。相比之下，基于微服务的应用程序是在多个机器上运行的分布式系统。每个服务实例通常是一个进程。因此，如下图所示，服务必须使用进程间通信（IPC）机制进行交互。



稍后我们将讨论具体的IPC技术，但首先让我们探讨各种设计问题。

互动样式

当为服务选择IPC机制时，首先考虑服务如何交互是有用的。有各种client↔service交互风格。它们可以沿着两个维度分类。第一个维度是交互是一对一还是一对多：

- 一对一 - 每个客户端请求都由一个服务实例处理。
- 一对多 - 每个请求由多个服务实例处理。

第二个维度是交互是同步还是异步：

- 同步 - 客户端期望来自服务的及时响应，并且甚至可以在其等待时阻塞。
- 异步 - 客户端在等待响应时不阻止，并且响应（如果有）不一定立即发送。

下表显示各种交互样式。

	一对一	一对多
同步	请求/响应	--
异步	通知	发布/订阅
	请求/异步响应	发布/异步响应

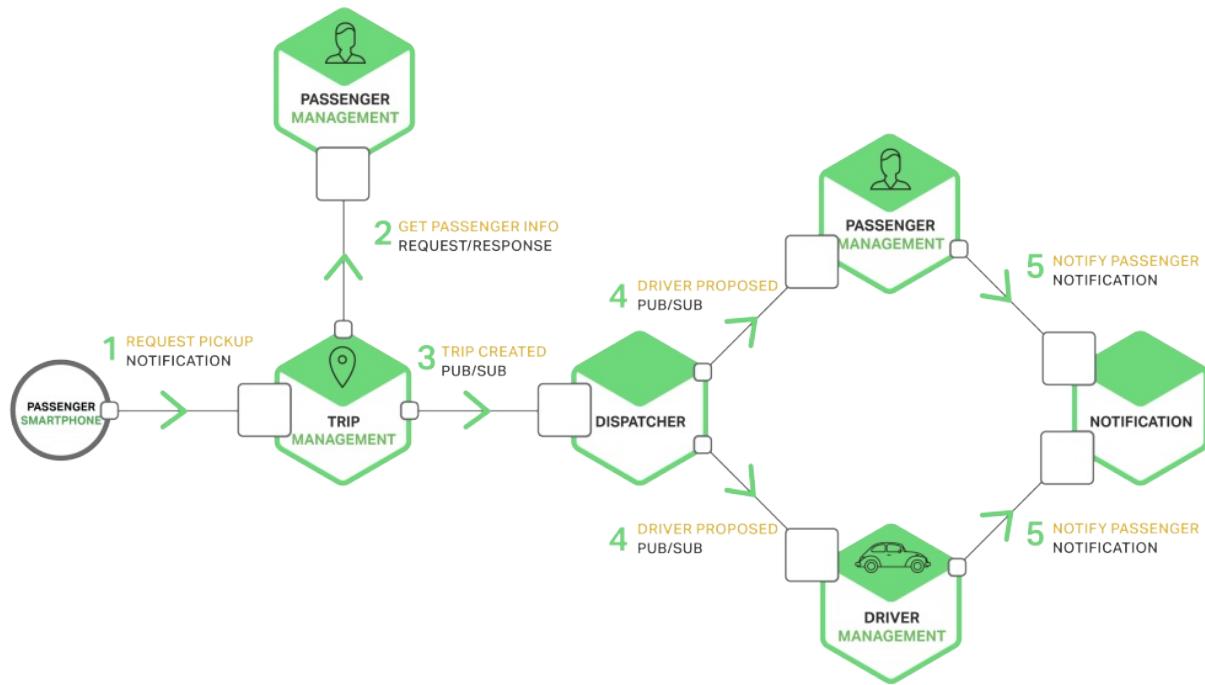
有以下种类的一对一交互：

- 请求/响应 - 客户端向服务器发出请求并等待响应。客户期望响应及时到达。在基于线程的应用程序中，发出请求的线程甚至可能在等待时阻塞。
- 通知（也称为单向请求） - 客户端向服务器发送请求，但不期望或发送回复。
- 请求/异步响应 - 客户端向服务异步回复的服务发送请求。客户端在等待时不阻塞，并且设计有假设响应可能不会到达一段时间。

有以下种类的一对多交互：

- 发布/订阅 - 客户端发布通知消息，由零个或多个感兴趣的服务使用。
- 发布/异步响应 - 客户端发布请求消息，然后等待一定量的时间用于感兴趣的服务的响应。

每个服务通常使用这些交互风格的组合。对于某些服务，单个IPC机制就足够了。其他服务可能需要使用IPC机制的组合。下图显示了当用户请求旅行时，出租汽车应用程序中的服务可能如何交互。



服务使用通知，请求/响应和发布/订阅的组合。例如，乘客的智能手机向旅行管理服务发送通知以请求提取。旅行管理服务通过使用请求/响应来调用乘客服务来验证旅客的帐户是否活动。旅行管理服务然后创建旅行并使用发布/订阅来通知其他服务，包括定位可用驱动程序的调度程序。

现在我们已经看了交互风格，让我们来看看如何定义API。

定义API

服务的API是服务与其客户之间的合同。无论您选择IPC机制，使用某种接口定义语言（IDL）精确定义服务的API非常重要。使用[API优先方法](#)来定义服务甚至有很好的理由。您通过编写接口定义并与客户端开发人员一起审查来开始服务的开发。只有在对API定义进行迭代之后，才能实现服务。在前面进行此设计可提高您构建满足其客户需求的服务的机会。

正如你将在本文后面看到的，API定义的性质取决于你使用的IPC机制。如果您使用消息传递，API由消息通道和消息类型组成。如果您使用HTTP，API由URL和请求和响应格式组成。稍后我们将更详细地描述一些IDL。

演进API

服务的API总是随时间变化。在单片应用程序中，通常可以直接更改API并更新所有调用者。在基于微服务的应用程序中，它是更困难很多，即使你的API的所有消费者是同一个应用程序中的其他服务。您通常不能强制所有客户端与服务一起锁定升级。此外，您可能会[逐步部署](#)

新版本的服务，以使旧版本和新版本的服务将同时运行。有一个处理这些问题的战略是很重要的。

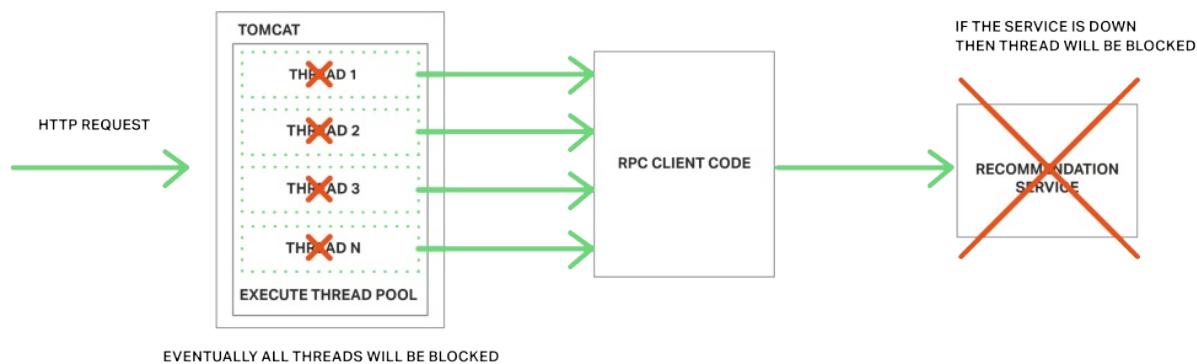
如何处理API更改取决于更改的大小。一些更改是轻微的，向后兼容以前的版本。例如，您可以向请求或响应添加属性。设计客户端和服务以使其遵守[鲁棒性原则是有意义的](#)。使用旧API的客户端应继续使用新版本的服务。该服务为缺少的请求属性提供默认值，并且客户端忽略任何额外的响应属性。重要的是使用IPC机制和消息传递格式，使您能够轻松地改进您的API。

然而，有时，您必须对API进行主要的，不兼容的更改。由于您无法强制客户端立即升级，因此服务必须在一段时间内支持API的旧版本。如果您使用基于HTTP的机制（如REST），一种方法是在URL中嵌入版本号。每个服务实例可能同时处理多个版本。或者，您可以部署各自处理特定版本的不同实例。

处理部分故障

如上一篇关于[API网关的文章所述](#)，在分布式系统中，存在部分故障的永远存在的风险。由于客户端和服务是单独的进程，因此服务可能无法及时响应客户端的请求。由于故障或维护，服务可能会关闭。或者服务可能过载并且对请求响应非常慢。

例如，考虑该文章中的[产品详细信息方案](#)。让我们假设推荐服务没有响应。客户端的简单实现可能会阻止无限期地等待响应。这不仅会导致糟糕的用户体验，但在许多应用程序中，它会消耗宝贵的资源，如线程。最终，运行时将用完线程并变得无响应，如下图所示。



为了防止这个问题，你必须设计你的服务来处理部分失败。

一个好的方法来跟随是[Netflix描述的](#)。处理部分故障的策略包括：

- 网络超时 - 不要无限期地阻止，并且在等待响应时始终使用超时。使用超时可确保资源永远不会无限制地关闭。
- 限制未完成请求的数量 - 对客户端可能拥有的特定服务的未完成请求的数量施加上限。如果已达到限制，则可能没有做出额外的请求，这些尝试需要立即失败。

- [断路器模式](#)

- 跟踪成功和失败请求的数量。如果错误率超过配置的阈值，请断开断路器，以便进一步尝试立即失败。如果大量请求失败，则表明服务不可用，并且发送请求是无意义的。超时时间后，客户端应再次尝试，如果成功，请关闭断路器。
- 提供回退 - 请求失败时执行回退逻辑。例如，返回缓存数据或默认值，例如空的推荐集。

[Netflix Hystrix](#)是一个实现这些和其他模式的开源库。如果你使用JVM，你一定要考虑使用Hystrix。并且，如果您在非JVM环境中运行，则应使用等效库。

IPC技术

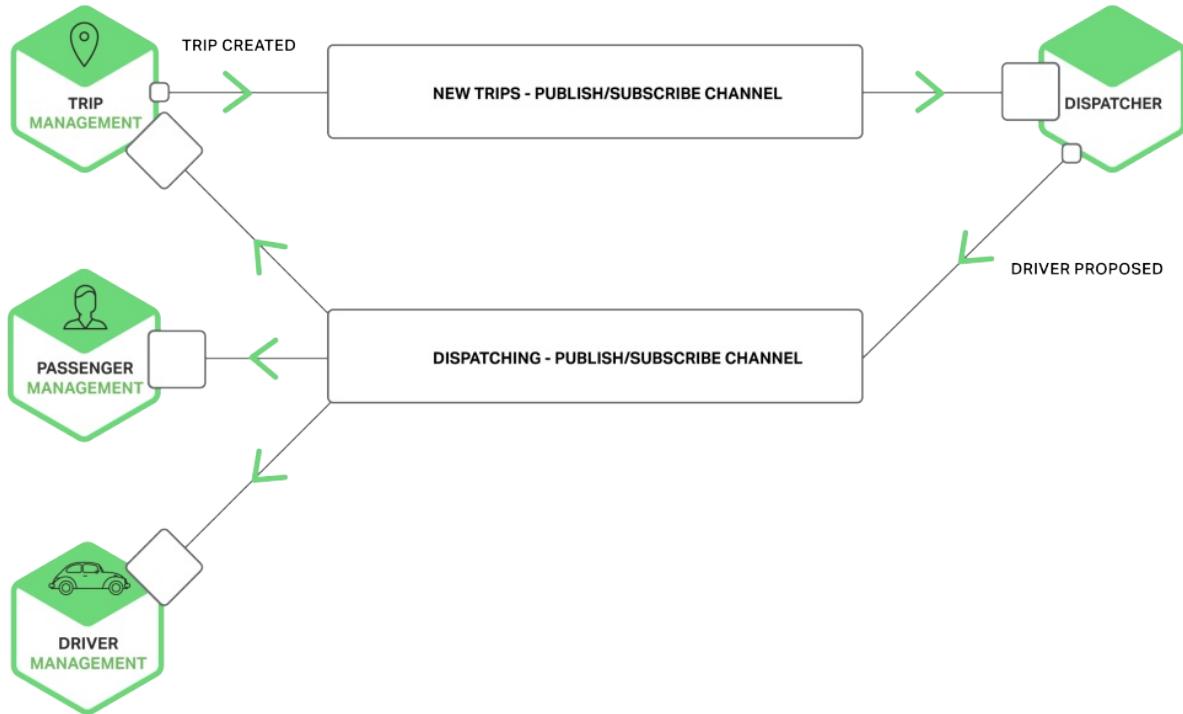
有很多不同的IPC技术可供选择。服务可以使用基于同步请求/响应的通信机制，例如基于HTTP的REST或Thrift。或者，它们可以使用异步的基于消息的通信机制，例如AMQP或STOMP。还有各种不同的消息格式。服务可以使用人类可读的基于文本的格式，例如JSON或XML。或者，他们可以使用二进制格式（更高效），如Avro或协议缓冲区。稍后我们将讨论同步IPC机制，但首先让我们讨论异步IPC机制。

异步，基于消息的通信

当使用消息传递时，进程通过异步交换消息进行通信。客户端通过向服务发送消息向服务发出请求。如果服务期望回复，它通过发送单独的消息回到客户端这样做。由于通信是异步的，所以客户端不会阻塞等待应答。相反，客户端被写为假定不会立即接收到答复。

一个[消息](#)由头（元数据，例如发件人）和消息体。消息通过[通道](#)交换。任何数量的生产者可以向一个信道发送消息。类似地，任何数量的消费者可以从频道接收消息。有两种类型的通道，[点对点](#)和[发布订阅](#)。点对点信道向正在从信道读取的消费者中的一个消费者传递消息。服务使用点对点通道用于前面描述的一对一交互风格。发布 - 订阅频道将每个消息传递给所有附加的消费者。服务使用发布 - 订阅频道来实现上述的一对多交互方式。

下图显示了出租汽车应用程序如何使用发布订阅频道。



旅行管理服务通过向发布订阅频道写入旅行创建消息来通知感兴趣的服务，例如调度员关于新旅行。Dispatcher找到一个可用的驱动程序并通过向发布订阅通道写入一个驱动程序建议消息来通知其他服务。

有许多信息系统可供选择。你应该选择一个支持各种编程语言。一些消息系统支持标准协议，如AMQP和STOMP。其他消息系统具有专有的但记录的协议。有大量的开源消息系统可供选择，包括RabbitMQ，Apache Kafka，Apache ActiveMQ和NSQ。在高水平，他们都支持某种形式的消息和渠道。他们都力求可靠，高性能和可扩展性。然而，每个代理的消息传递模型的细节存在显著差异。

使用消息传递有很多好处：

- 将客户端与服务解耦 - 客户端通过向相应的通道发送消息来发出请求。客户端完全不知道服务实例。它不需要使用发现机制来确定服务实例的位置。
- 消息缓冲 - 使用同步请求/响应协议（如HTTP），客户端和服务都必须在交换期间可用。相反，消息代理将写入信道的消息排队，直到消费者可以处理它们。这意味着，例如，即使在订单履行系统较慢或不可用时，在线商店也可以接受来自客户的订单。订单消息只是排队。
- 灵活的客户端 - 服务交互 - 消息传递支持之前描述的所有交互方式。
- 显式进程间通信 - 基于RPC的机制试图使调用远程服务看起来与调用本地服务相同。然而，由于物理定律和部分失效的可能性，它们实际上是完全不同的。消息传递使得这些差异非常明确，因此开发人员不会陷入虚假的安全感。

但是，使用消息传递有一些缺点：

- 附加操作复杂性 - 消息系统是必须安装，配置和操作的另一个系统组件。消息代理必须

高度可用，否则系统可靠性会受到影响。

- 实现基于请求/响应的交互的复杂性 - 请求/响应式交互需要一些工作来实现。每个请求消息必须包含回复信道标识符和相关标识符。服务将包含相关ID的响应消息写入回复通道。客户端使用关联ID将响应与请求进行匹配。使用直接支持请求/响应的IPC机制通常更容易。

现在我们已经研究了使用基于消息的IPC，让我们检查基于请求/响应的IPC。

同步，请求/响应IPC

当使用基于同步，基于请求/响应的IPC机制时，客户端向服务发送请求。服务处理请求并发出响应。在许多客户端中，在等待响应时使请求块阻塞的线程。其他客户端可能使用异步，事件驱动的客户端代码，可能由Futures或Rx Observables封装。但是，与使用消息传递不同，客户端假设响应将及时到达。有很多协议可供选择。两种流行的协议是REST和Thrift。让我们先来看看REST。

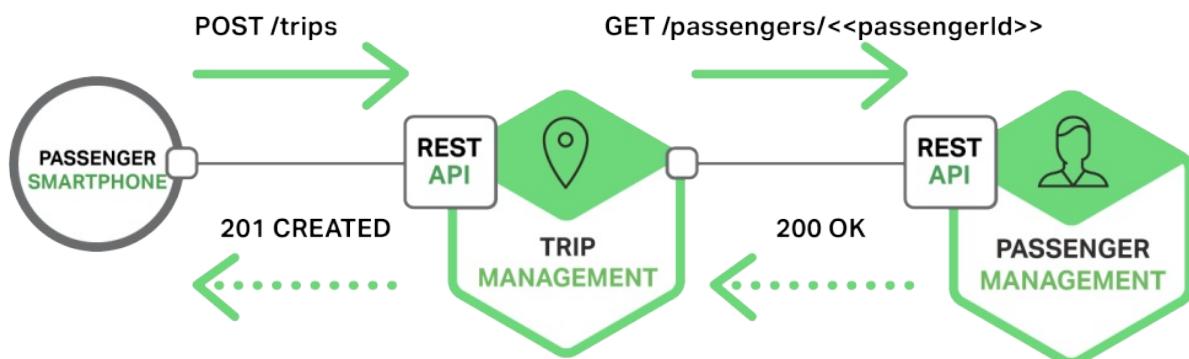
休息

今天，以RESTful风格开发API是时尚。REST是一种（几乎总是）使用HTTP的IPC机制。REST中的一个关键概念是资源，它通常表示业务对象（如客户或产品）或业务对象集合。REST使用HTTP动词来处理使用URL引用的资源。例如，`GET`请求返回资源的表示形式，可能是XML文档或JSON对象的形式。一个`POST`请求创建一个新的资源和`PUT`请求更新的资源。引用REST的创建者Roy Fielding：

“REST提供了一套架构约束，当整体应用时，强调组件交互的可扩展性，接口的一般性，组件的独立部署和中间组件，以减少交互延迟，实施安全性并封装传统系统。”

- 架构，建筑风格和基于网络的软件架构的设计

下图显示了出租汽车应用程序可能使用REST的一种方式。



乘客的智能手机通过向旅行管理服务 POST 的 /trips 资源发出请求来请求旅行。此服务通过向 GET 乘客管理服务发送有关乘客信息的请求来处理请求。在验证乘客被授权创建旅行之后，旅行管理服务创建旅行并且 201 向智能电话返回响应。

许多开发人员声称他们的基于HTTP的API是RESTful的。但是，正如Fielding在这篇[博客](#)中描述的，并不是所有的都是。Leonard Richardson（无关）定义了一个非常有用的[REST成熟度模型](#)，由以下级别组成。

- 级别0 - 级别0的客户端通过 POST 向其唯一的URL端点发出HTTP请求来调用服务。每个请求指定要执行的操作，操作的目标（例如业务对象）和任何参数。
- 第1级 - 第1级API支持资源的想法。要对资源执行操作，客户端会发出一个 POST 请求，指定要执行的操作和任何参数。
- 第2级 - 第2级API使用HTTP动词执行操作：GET 检索，POST 创建和PUT 更新。请求查询参数和正文（如果有）指定动作的参数。这使服务能够利用Web基础架构，如缓存 GET 请求。
- 级别3 - 3级API的设计基于可怕的命名HATEOAS（超文本作为应用程序状态引擎）原理。基本思想是由 GET 请求返回的资源的表示包含用于对该资源执行允许的动作的链接。例如，客户端可以使用响应于 GET 发送的请求返回的Order表示中的链接来取消订单以检索订单。[HATEOAS的好处](#) 包括不再需要将URL硬编码到客户端代码中。另一个好处是，因为资源的表示包含用于允许的动作的链接，所以客户端不必猜测对其当前状态下的资源可以执行什么动作。

使用基于HTTP的协议有很多好处：

- HTTP是简单和熟悉的。
- 您可以使用扩展名（例如 Postman）或从命令行使用 curl（假设使用JSON或其他文本格式）在浏览器中测试HTTP API。
- 它直接支持请求/响应式通信。
- HTTP当然是防火墙友好的。
- 它不需要中间代理，这简化了系统的架构。

使用HTTP有一些缺点：

- 它只是直接支持请求/响应式的交互。您可以对通知使用HTTP，但服务器必须始终发送HTTP响应。
- 因为客户端和服务直接通信（没有中间体来缓冲消息），它们必须都在交换期间运行。
- 客户端必须知道每个服务实例的位置（即，URL）。如上一篇[关于API网关的文章所述](#)，这是现代[应用程序](#)中的一个非平凡问题。客户端必须使用服务发现机制来定位服务实例。

开发人员社区最近重新发现了RESTful API的接口定义语言的价值。有几个选项，包括[RAML](#)和[Swagger](#)。一些IDL（如Swagger）允许您定义请求和响应消息的格式。其他如RAML要求您使用单独的规范，如[JSON模式](#)。除了描述API之外，IDL通常还具有从接口定义生成客户端存根和服务器框架的工具。

节约

[Apache Thrift](#)是REST的一个有趣的替代品。它是一个用于编写跨语言[RPC](#)客户端和服务器的框架。Thrift提供了一个C风格的IDL来定义你的API。您使用Thrift编译器来生成客户端存根和服务器端骨架。编译器生成各种语言的代码，包括C ++，Java，Python，PHP，Ruby，Erlang和Node.js。

Thrift接口由一个或多个服务组成。服务定义类似于Java接口。它是一个强类型方法的集合。Thrift方法可以返回一个（可能为void）值，也可以定义为单向。返回值的方法实现了交互的请求/响应风格。客户端等待响应，可能会抛出异常。单向方法对应于交互的通知风格。服务器不发送响应。

Thrift支持各种消息格式：JSON，二进制和紧凑二进制。二进制比JSON更高效，因为它更快地解码。而且，顾名思义，紧凑二进制是一种节省空间的格式。JSON是，当然，人类和浏览器友好。Thrift还为您提供了传输协议的选择，包括原始TCP和HTTP。原始TCP可能比HTTP更有效。但是，HTTP是防火墙，浏览器和人性化的。

消息格式

现在我们来看看HTTP和Thrift，让我们来看一下消息格式的问题。如果您正在使用消息传递系统或REST，则可以选择消息格式。其他IPC机制，如Thrift可能只支持少量的消息格式，也许只有一个。在任一情况下，使用跨语言消息格式很重要。即使您今天使用单一语言编写您的微服务，也有可能在将来使用其他语言。

有两种主要的消息格式：文本和二进制。基于文本格式的示例包括JSON和XML。这些格式的优点是，它们不仅是人类可读的，而且是自我描述的。在JSON中，对象的属性由一组名称 - 值对表示。类似地，在XML中，属性由命名的元素和值表示。这使得消息的消费者能够挑选其感兴趣的值并忽略其余值。因此，对消息格式的微小改变可以容易地向后兼容。

XML文档的结构由[XML模式](#)指定。随着时间的推移，开发人员社区已经意识到JSON也需要类似的机制。一个选择是使用[JSON Schema](#)，独立或作为IDL的一部分，如Swagger。

使用基于文本的消息格式的缺点是消息往往是冗长的，尤其是XML。因为消息是自描述的，所以每个消息都包含属性的名称以及它们的值。另一个缺点是解析文本的开销。因此，您可能需要考虑使用二进制格式。

有几种二进制格式可供选择。如果你使用Thrift RPC，你可以使用二进制Thrift。如果你可以选择消息格式，流行的选项包括Protocol Buffers和Apache Avro。这两种格式都提供了一个类型化的IDL来定义消息的结构。然而，一个区别是协议缓冲器使用标记字段，而Avro消费者需要知道该模式以解释消息。因此，使用协议缓冲区的API演化比使用Avro更容易。这篇[博文](#)是Thrift，Protocol Buffers和Avro的一个很好的比较。

概要

微服务必须使用进程间通信机制进行通信。在设计服务如何进行通信时，您需要考虑各种问题：服务如何交互，如何为每个服务指定API，如何演进API以及如何处理部分故障。微服务可以使用两种IPC机制，异步消息和同步请求/响应。在本系列的下一篇篇文章中，我们将讨论微服务架构中服务发现的问题。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信（本文）
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现（本文）
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

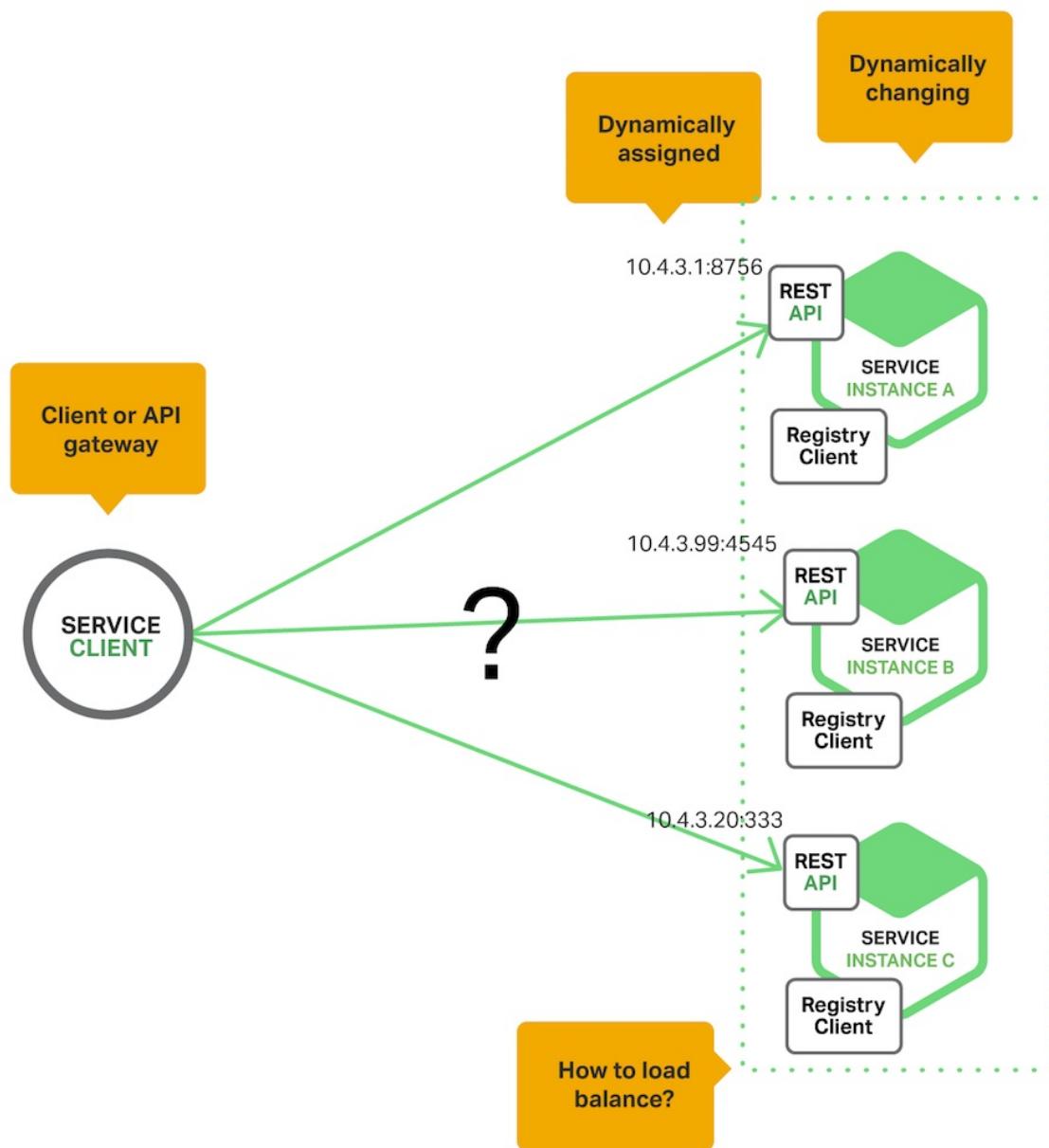
您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这是我们关于使用微服务构建应用程序的第四篇文章。在[第一篇文章](#)介绍了[微服务架构模式](#)和讨论的好处和使用微服务的缺点。的[第二](#)和[第三](#)系列中的文章描述了一个微服务架构内的通信的不同方面。在本文中，我们探讨密切相关的服务发现问题。

为什么使用服务发现？

让我们假设你正在编写一些代码来调用具有REST API或Thrift API的服务。为了发出请求，您的代码需要知道服务实例的网络位置（IP地址和端口）。在运行在物理硬件上的传统应用中，服务实例的网络位置是相对静态的。例如，您的代码可以从偶尔更新的配置文件中读取网络位置。

然而，在现代的基于云的微服务应用程序中，这是一个更难解决的问题，如下图所示。



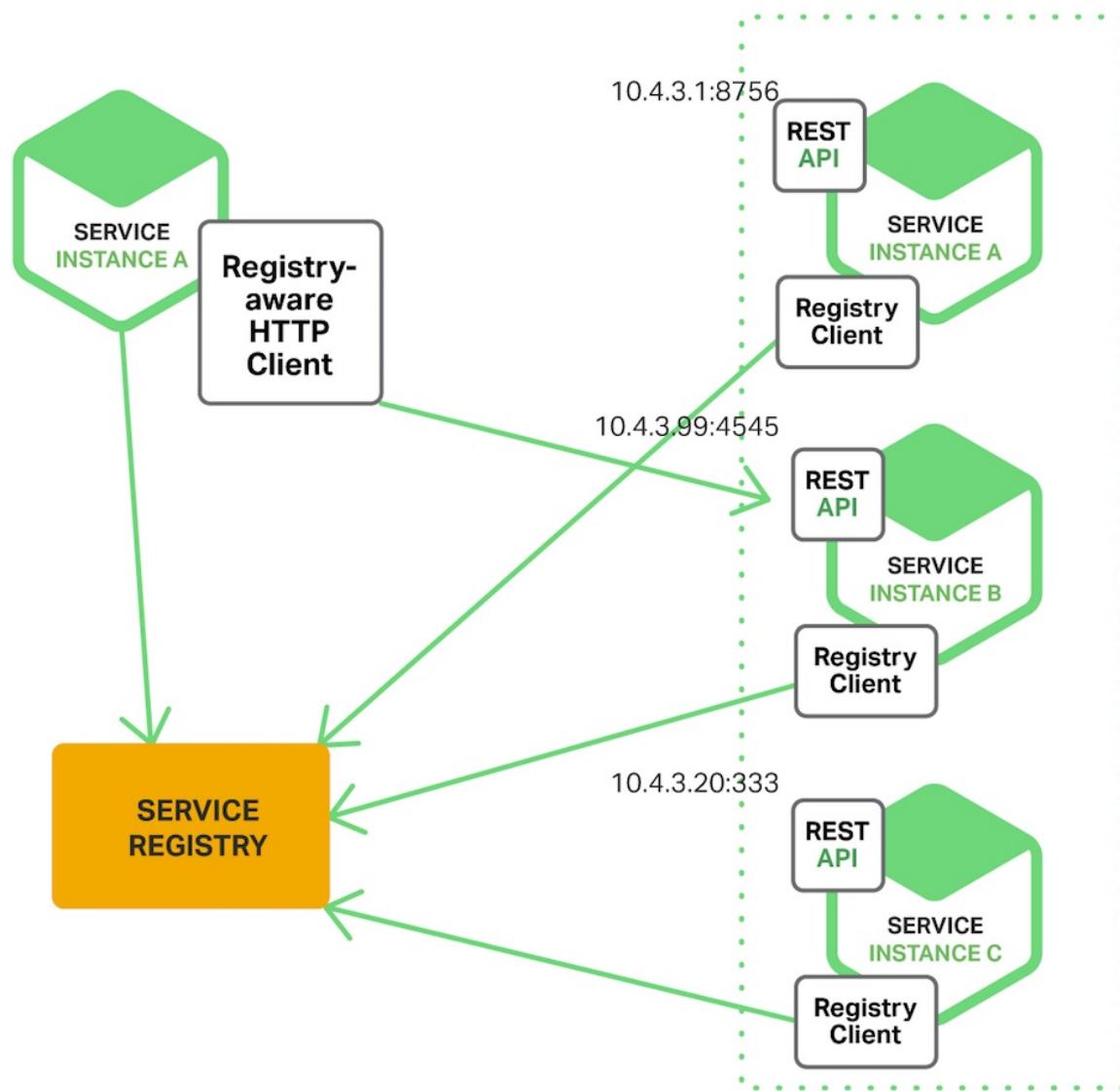
服务实例具有动态分配的网络位置。此外，服务实例集合由于自动缩放，故障和升级而动态地改变。因此，您的客户端代码需要使用更精细的服务发现机制。

有两种主要的服务发现模式：[客户端发现](#)和[服务器端发现](#)。让我们先来看看客户端发现。

客户端发现模式

当使用[客户端发现](#)时，客户端负责确定可用服务实例的网络位置和跨它们的负载平衡请求。客户端查询服务注册表，该服务注册表是可用服务实例的数据库。然后，客户端使用负载平衡算法来选择可用服务实例之一并发出请求。

下图显示了此模式的结构。



服务实例的网络位置在启动时向服务注册表注册。当实例终止时，它将从服务注册表中删除。通常使用心跳机制周期性地刷新服务实例的注册。

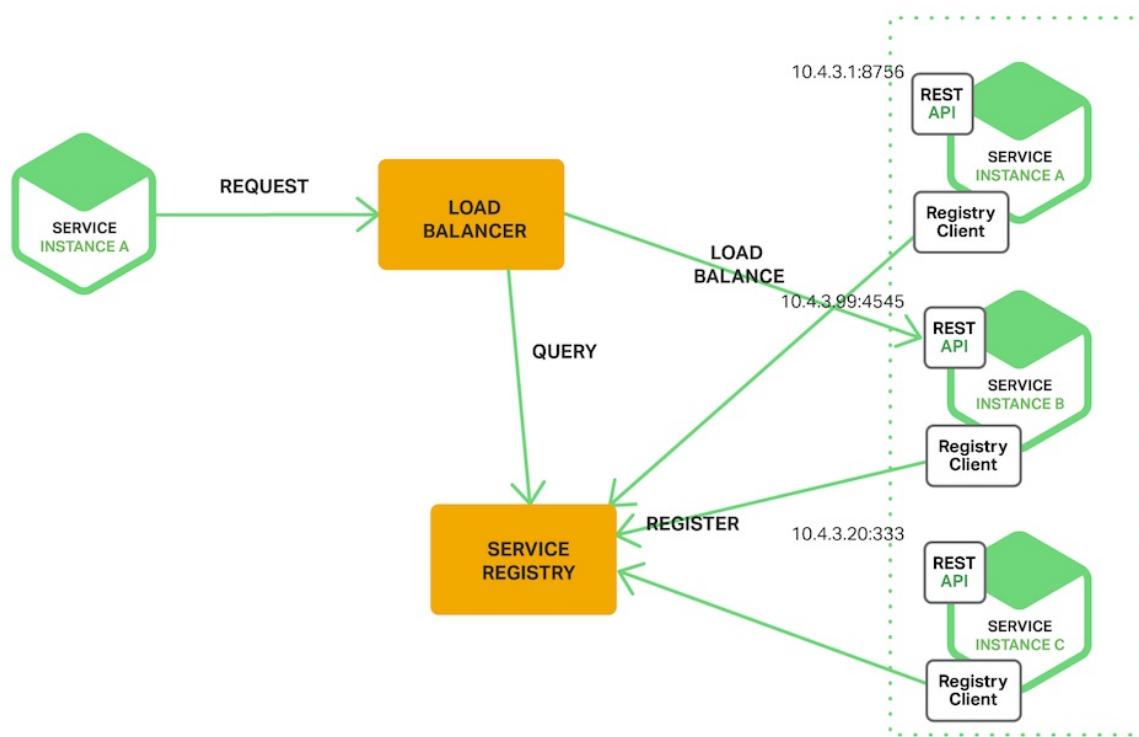
[Netflix OSS](#)提供了客户端发现模式的一个很好的例子。[Netflix Eureka](#)是一个服务注册表。它提供了一个用于管理服务实例注册和查询可用实例的REST API。[Netflix功能区](#)是一个IPC客户端，与Eureka一起工作，在可用的服务实例之间进行负载均衡请求。我们将在本文后面更深入地讨论Eureka。

客户端发现模式具有各种好处和缺点。这种模式相对直接，除了服务注册表，没有其他移动部分。此外，由于客户端知道可用的服务实例，它可以做出智能的，特定于应用程序的负载平衡决策，例如一致地使用哈希。这种模式的一个显着缺点是它将客户端与服务注册表耦合。您必须为服务客户端使用的每种编程语言和框架实现客户端服务发现逻辑。

现在我们已经研究了客户端发现，让我们来看看服务器端发现。

服务器端发现模式

服务发现的另一种方法是服务器端发现模式。下图显示了此模式的结构。



客户端通过负载均衡器向服务器发出请求。负载均衡器查询服务注册表并将每个请求路由到可用的服务实例。与客户端发现一样，服务实例将通过服务注册表注册和注销。

在[AWS弹性负载均衡（ELB）](#)是一个服务器端的路由器发现的一个例子。ELB通常用于负载平衡来自因特网的外部流量。但是，您也可以使用ELB对虚拟私有云（VPC）内部的流量进行负载平衡。客户端使用其DNS名称通过ELB发出请求（HTTP或TCP）。ELB负载平衡一组注册的弹性计算云（EC2）实例或EC2容器服务（ECS）容器之间的流量。没有单独的服务注册表。相反，EC2实例和ECS容器向ELB本身注册。

HTTP服务器和负载均衡器（如[NGINXPlus](#)和[NGINX](#)）也可用作服务器端发现负载均衡器。例如，本[博客文章](#)描述了使用[Consul Template](#)来动态重新配置NGINX反向代理。Consul模板是一种工具，它定期从存储在[Consul](#)服务注册表中的配置数据重新生成任意配置文件。每当文件更改时，它运行一个任意的shell命令。在博客文章描述的示例中，Consul Template生成一个[nginx.conf](#)文件，该文件配置反向代理，然后运行一个命令，告诉NGINX重新加载配置。

一些部署环境（如[Kubernetes](#)和[Marathon](#)）在集群中的每个主机上运行代理。代理扮演服务器端发现负载均衡器的角色。为了对服务进行请求，客户机使用主机的IP地址和服务的分配端口经由代理路由请求。然后，代理将该请求透明地转发到在集群中某处运行的可用服务实例。

服务器端发现模式有几个好处和缺点。这种模式的一个很大的好处是发现的细节被抽象出远离客户端。客户端只向负载均衡器发出请求。这消除了为服务客户端使用的每种编程语言和框架实施发现逻辑的需要。此外，如上所述，一些部署环境免费提供此功能。这种模式也有

一些缺点，但是。除非负载均衡器由部署环境提供，否则它是您需要设置和管理的另一个高可用性系统组件。

服务注册表

该[服务注册中心](#)是服务发现的一个关键部分。它是一个包含服务实例的网络位置的数据库。服务注册表需要高度可用并且是最新的。客户端可以缓存从服务注册表获取的网络位置。但是，该信息最终会过时，并且客户端无法发现服务实例。因此，服务注册表由使用复制协议来维护一致性的服务器集群组成。

如前所述，[Netflix Eureka](#)是服务注册表的一个很好的例子。它提供了一个用于注册和查询服务实例的REST API。服务实例使用 `POST` 请求来注册其网络位置。每30秒，它必须使用 `PUT` 请求刷新其注册。通过使用`HTTP DELETE`请求或通过实例注册超时来删除注册。如您所料，客户端可以通过使用`HTTP GET`请求来检索注册的服务实例。

[Netflix](#)通过在每个Amazon EC2可用区域中运行一个或多个Eureka服务器来[实现高可用性](#)。每个Eureka服务器在具有[弹性IP地址](#)的EC2实例上运行。DNS `TEXT` 记录用于存储Eureka集群配置，这是从可用区到Eureka服务器的网络位置列表的映射。当Eureka服务器启动时，它查询DNS以检索Eureka集群配置，定位其对等端，并为自己分配一个未使用的弹性IP地址。

Eureka客户端 - 服务和服务客户端 - 查询DNS以发现Eureka服务器的网络位置。客户端更喜欢在同一可用区域中使用Eureka服务器。但是，如果没有可用区域，则客户端在另一个可用区域中使用Eureka服务器。

服务注册管理机构的其他示例包括：

- [etcd](#)
 - 用于共享配置和服务发现的高可用性，分布式，一致的键值存储。使用etcd的两个值得注意的项目是[Kubernetes](#)和[Cloud Foundry](#)。
- [领事](#) - 一种用于发现和配置服务的工具。它提供了一个API，允许客户端注册和发现服务。领事可以执行健康检查以确定服务可用性。
- [Apache Zookeeper](#)
 - 用于[分布式](#)应用程序的广泛使用的高性能协调服务。Apache Zookeeper最初是[Hadoop](#)的子项目，但现在是一个顶级项目。

此外，如前所述，一些系统（如[Kubernetes](#)，[Marathon](#)和[AWS](#)）没有明确的服务注册表。相反，服务注册表只是基础结构的内置部分。

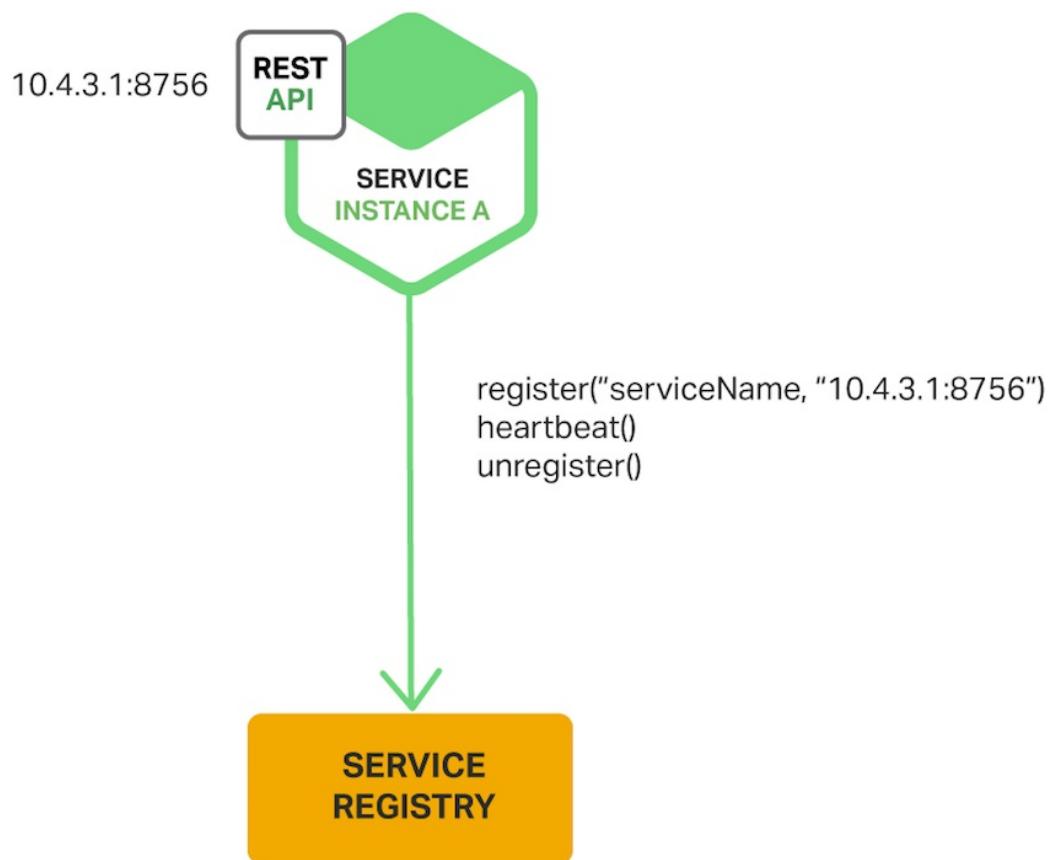
现在我们已经看了服务注册表的概念，让我们看看服务实例如何注册到服务注册表。

服务注册选项

如前所述，服务实例必须向服务注册表注册和注销。有两种不同的方式来处理注册和注销。一个选项是服务实例注册自己，[自注册模式](#)。另一个选项是对于某些其他系统组件来管理服务实例的注册，[第三方注册模式](#)。让我们先来看看自我注册模式。

自注册模式

当使用[自注册模式](#)时，服务实例负责向服务注册表注册和注销自身。此外，如果需要，服务实例发送心跳请求以防止其注册到期。下图显示了此模式的结构。



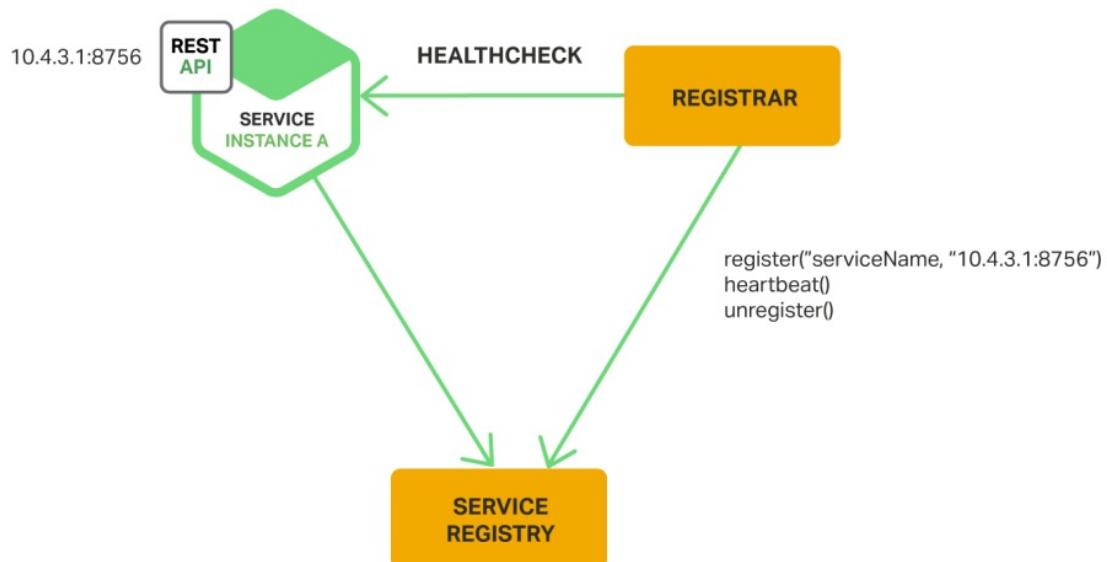
这种方法的一个很好的例子是[Netflix OSS Eureka客户端](#)。Eureka客户端处理服务实例注册和注销的所有方面。在[春天的云项目](#)，它实现了多种模式，包括服务发现，可以很容易地与尤里卡自动注册服务实例。您只需用注释注释您的Java配置类 `@EnableEurekaClient`。

自注册模式具有各种益处和缺点。一个好处是它相对简单，并且不需要任何其他系统组件。然而，主要的缺点是它将服务实例耦合到服务注册表。您必须在您的服务使用的每种编程语言和框架中实现注册码。

将服务与服务注册表分离的替代方法是[第三方注册模式](#)。

第三方注册模式

当使用[第三方注册模式](#)时，服务实例不负责向服务注册表注册自己。相反，被称为服务注册器的另一个系统组件处理注册。服务注册器通过轮询部署环境或订阅事件来跟踪对运行实例集的更改。当它注意到新的可用服务实例时，它向服务注册表注册该实例。服务注册器还注销终止的服务实例。下图显示了此模式的结构。



服务注册器的一个示例是[开源注册器](#)项目。它自动注册和注销部署为Docker容器的服务实例。注册器支持多个服务注册表，包括etcd和Consul。

服务注册商的另一个例子是[Netflix OSS Prana](#)。主要用于以非JVM语言编写的服，它是与服务实例并行运行的边路应用程序。Prana使用Netflix Eureka注册和注销服务实例。

服务注册器是部署环境的内置组件。由Autoscaling组创建的EC2实例可以自动注册到ELB。Kubernetes服务自动注册并可用于发现。

第三方注册模式具有各种好处和缺点。主要优点是服务与服务注册表断开连接。您不需要为开发人员使用的每种编程语言和框架实现服务注册逻辑。相反，在专用服务内以集中方式处理服务实例注册。

这种模式的一个缺点是，除非它内置在部署环境中，它是另一个高度可用的系统组件，您需要设置和管理。

摘要

在微服务应用程序中，运行服务实例集会动态更改。实例具有动态分配的网络位置。因此，为了使客户机向服务发出请求，它必须使用服务发现机制。

服务发现的关键部分是[服务注册表](#)。服务注册表是可用服务实例的数据库。服务注册表提供了管理API和查询API。服务实例使用管理API从服务注册表注册和注销。系统组件使用查询API来发现可用的服务实例。

有两种主要的服务发现模式：客户端发现和服务端发现。在使用[客户端服务发现的系统中](#)，客户端查询服务注册表，选择可用实例，并发出请求。在使用[服务器端发现的系统中](#)，客户端通过路由器进行请求，路由器查询服务注册表并将请求转发到可用实例。

服务实例有两种主要方式向服务注册表注册和注销。一个选项是服务实例向服务注册表注册[自己的自注册模式](#)。另一个选项是用于一些其他系统组件代表服务处理注册和注销，[第三方注册模式](#)。

在某些部署环境中，您需要使用服务注册表（如[Netflix Eureka](#)，[etcd](#)或[Apache Zookeeper](#)）设置自己的服务发现[基础结构](#)。在其他部署环境中，内置了服务发现。例如，[Kubernetes](#)和[Marathon](#)处理服务实例注册和注销。他们还在担任[服务器端发现](#)路由器角色的每个群集主机上运行代理。

HTTP反向代理和负载平衡器（如[NGINX](#)）也可以用作服务器端发现负载平衡器。服务注册表可以将路由信息推送到[NGINX](#)并调用优雅的配置更新；例如，您可以使用[领事模板](#)。[NGINX Plus](#)支持[附加的动态重新配置机制](#) - 它可以使用[DNS](#)从注册表中提取有关服务实例的信息，并为远程重新配置提供API。

在将来的博客文章中，我们将继续深入了解微服务的其他方面。注册[NGINX邮件列表](#)（表格如下），以通知系列中未来的文章的发布。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现（本文）
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用[NGINX Plus](#)实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 微服务的事件驱动数据管理（本文）
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用[NGINX Plus](#)实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这是关于使用微服务构建应用程序的第五篇文章。在[第一篇文章](#)介绍了微服务架构模式，并讨论的好处和使用微服务的缺点。的[第二](#)和[第三](#)系列中的文章描述了一个微服务架构内的通信的不同方面。在[第四篇文章](#)探讨了服务发现的密切相关的问题。在本文中，我们改变齿轮，并看看在微服务架构中出现的分布式数据管理问题。

微服务和分布式数据管理的问题

单片应用程序通常具有单个关系数据库。使用关系数据库的一个关键好处是您的应用程序可以使用[ACID事务](#)，这提供了一些重要的保证：

- 原子性 - 原子性地进行改变
- 一致性 - 数据库的状态始终一致
- 隔离 - 即使事务被并发执行，它们似乎被顺序执行
- 耐久性 - 一旦事务已提交，它不会撤消

因此，您的应用程序可以简单地开始事务，更改（插入，更新和删除）多个行，并提交事务。

使用关系数据库的另一个好处是它提供了SQL，它是一种丰富的，声明性的和标准化的查询语言。您可以轻松地编写组合来自多个表的数据的查询。然后，RDBMS查询计划器确定执行查询的最佳方法。您不必担心低级别的详细信息，例如如何访问数据库。并且，因为您的应用程序的所有数据都在一个数据库中，所以很容易查询。

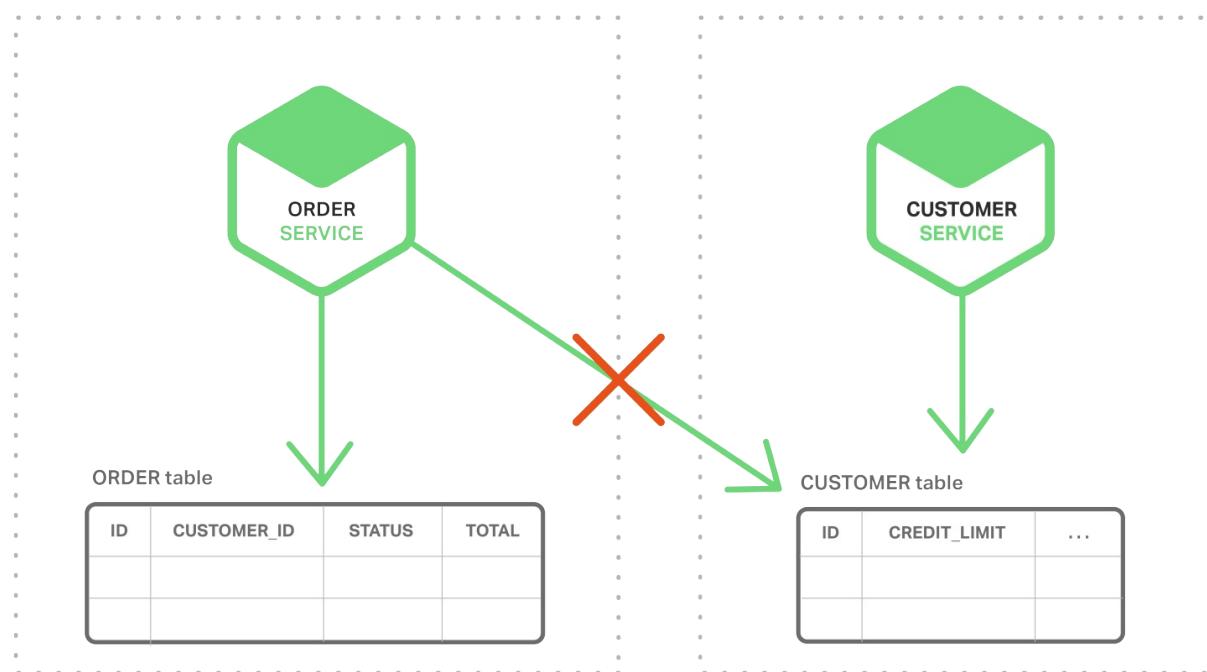
不幸的是，当我们转向微服务架构时，数据访问变得更加复杂。这是因为每个微服务拥有的数据对于该微服务是私有的，并且只能通过其API来访问。封装数据确保微服务松耦合并且可以彼此独立地演进。如果多个服务访问相同的数据，则模式更新需要耗费时间，协调地更新所有服务。

更糟糕的是，不同的微服务通常使用不同种类的数据库。现代应用程序存储和处理各种各样的数据，关系数据库并不总是最好的选择。对于某些使用情况，特定的NoSQL数据库可能具有更方便的数据模型，并提供更好的性能和可扩展性。例如，存储和查询文本的服务使用文本搜索引擎（例如Elasticsearch）是有意义的。类似地，存储社交图数据的服务应当使用图形数据库，例如Neo4j。因此，基于微服务的应用程序通常使用SQL和NoSQL数据库的混合，所谓的多语言持久化方法。

用于数据存储的分区的多磁盘持久性架构具有许多优点，包括松耦合服务和更好的性能和可扩展性。然而，它确实引入了一些分布式数据管理挑战。

第一个挑战是如何实现在多个服务之间保持一致性的业务事务。要了解为什么这是一个问题，让我们看看一个在线B2B商店的例子。客户服务维护有关客户的信息，包括他们的信用额度。订单服务管理订单，并且必须验证新订单不超过客户的信用额度。在此应用程序的单片版本中，订单服务可以简单地使用ACID事务来检查可用信用额并创建订单。

相比之下，在微服务架构中，ORDER和CUSTOMER表对它们各自的服务是私有的，如下图所示。



订单服务无法直接访问CUSTOMER表。它只能使用由客户服务提供的API。订单服务可能使用分布式事务，也称为两阶段提交（2PC）。然而，2PC在现代应用中通常不是可行的选择。该CAP定理需要您的可用性和酸风格的一致性之间做出选择和可用性通常是更好的选择。此外，许多现代技术，如大多数NoSQL数据库，不支持2PC。维护服务和数据库之间的数据一致性至关重要，因此我们需要另一个解决方案。

第二个挑战是如何实现从多个服务检索数据的查询。例如，让我们假设应用程序需要显示一个客户和他最近的订单。如果订单服务提供用于检索客户订单的API，那么您可以使用应用程序端加入来检索此数据。应用程序从客户服务中检索客户，并从订单服务检索客户的订单。

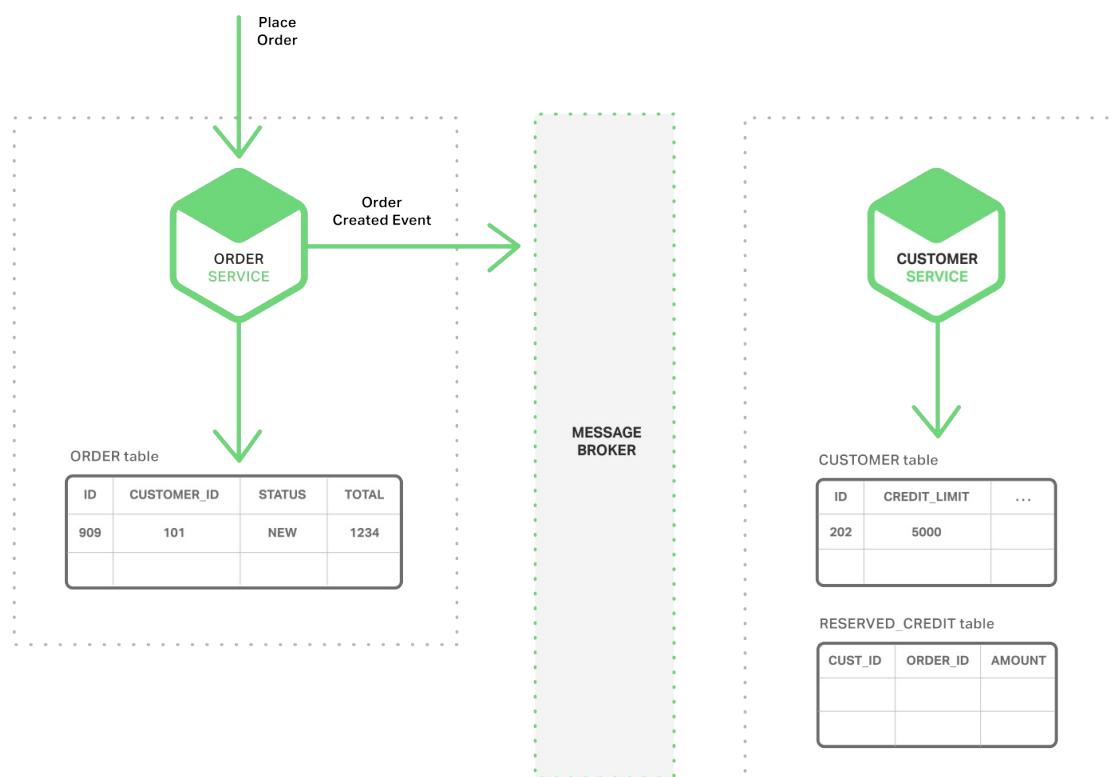
然而，假设订单服务仅支持通过其主键查找订单（也许它使用仅支持基于主键的检索的 NoSQL 数据库）。在这种情况下，没有明显的方法来检索所需的数据。

事件驱动架构

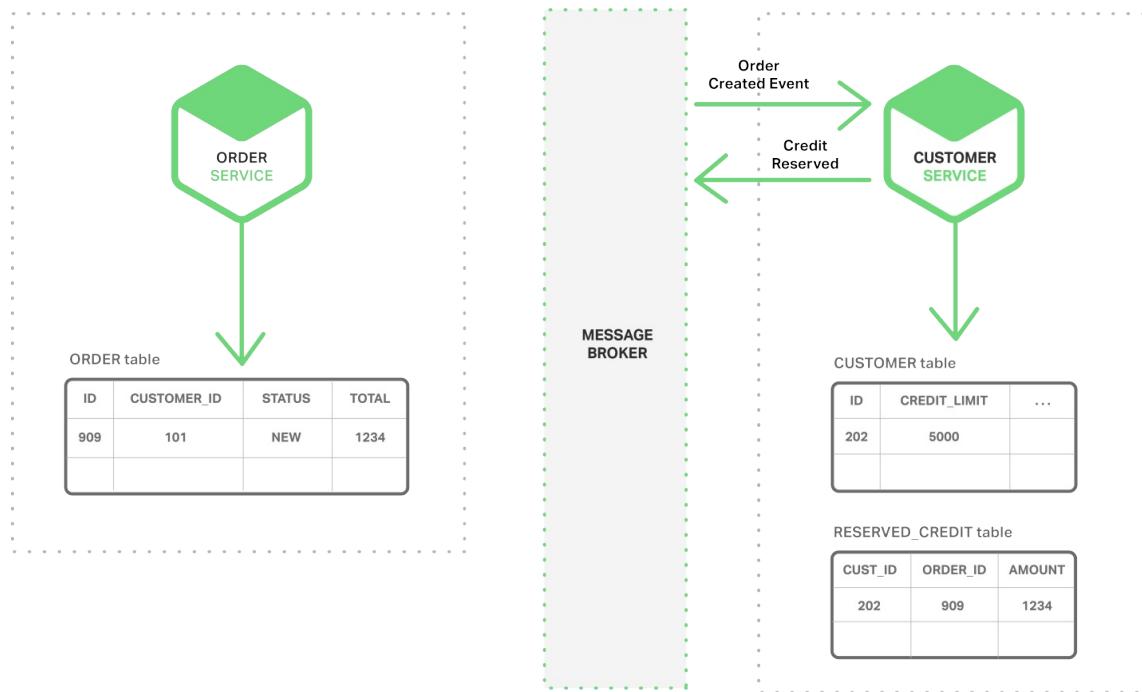
对于许多应用程序，解决方案是使用[事件驱动架构](#)。在此体系结构中，微服务在发生显着事件时发布事件，例如更新业务实体时。其他微服务订阅这些事件。当微服务接收到事件时，它可以更新其自己的业务实体，这可能导致更多的事件被发布。

您可以使用事件来实现跨多个服务的业务事务。事务包括一系列步骤。每个步骤包括微服务更新业务实体和发布触发下一步骤的事件。以下图表序列显示了如何使用事件驱动方法在创建订单时检查可用信用额。微服务通过Message Broker交换事件。

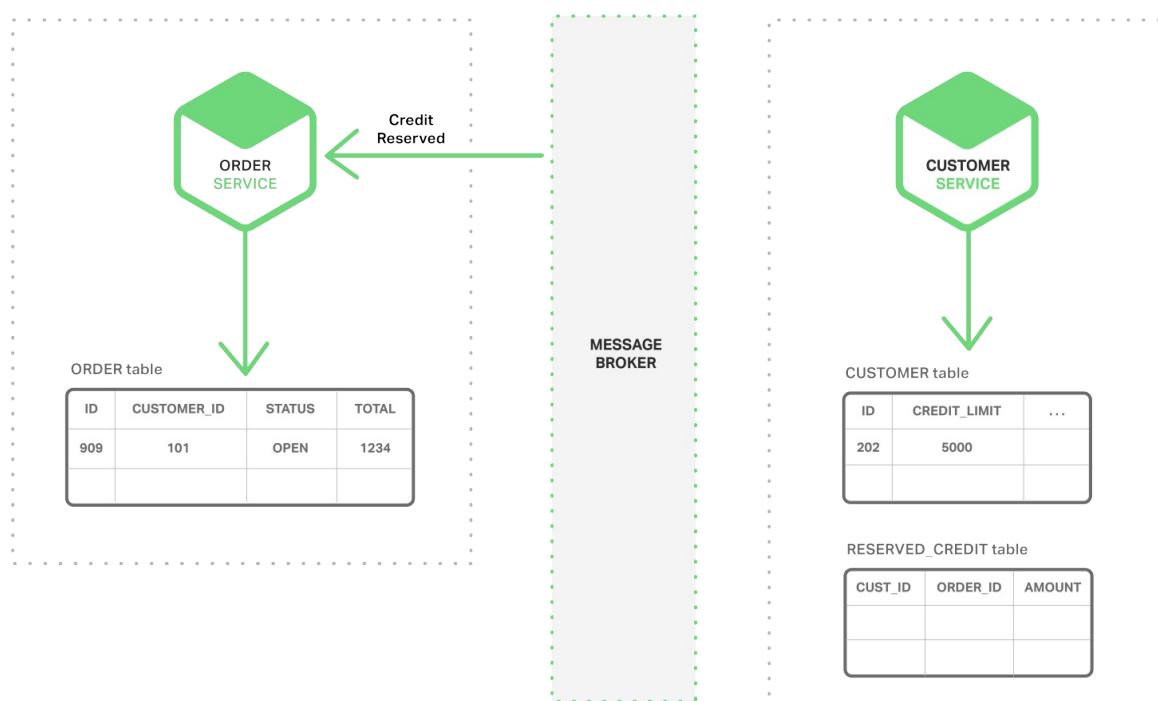
1. 订单服务创建状态为 NEW 的订单，并发布订单创建事件。



2. 客户服务使用订单创建事件，为订单预留信用，并发布信用预留事件。



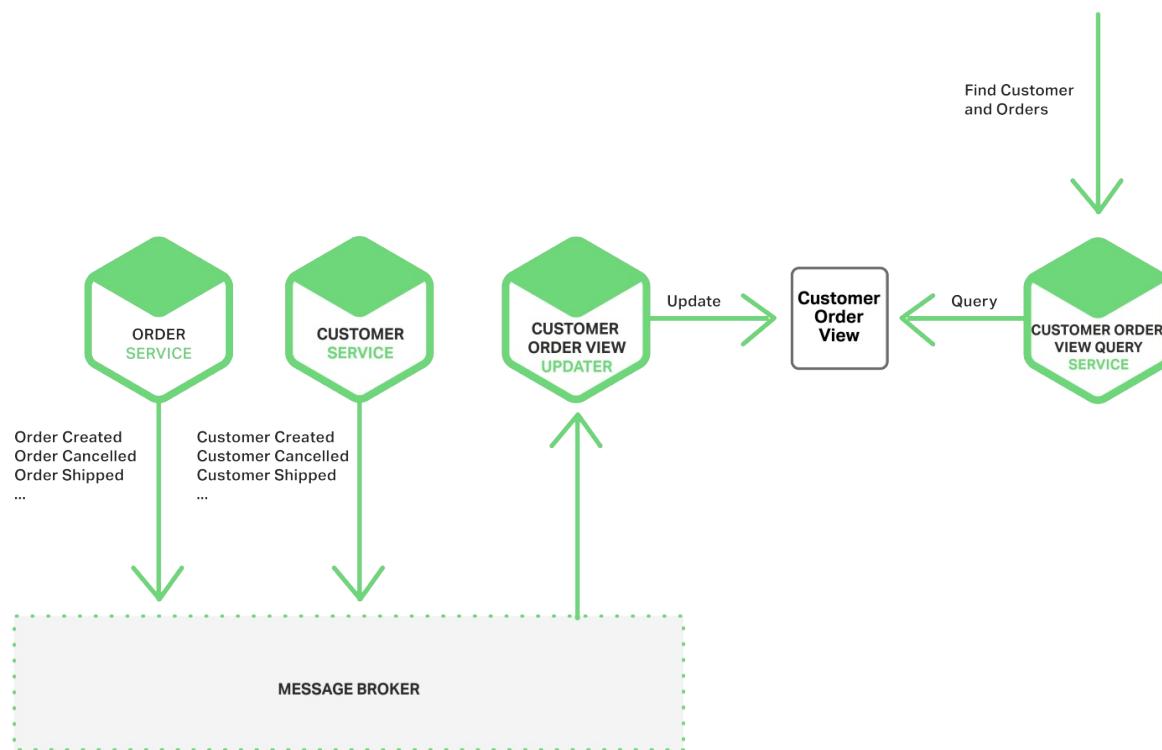
3. 订单服务消耗预留信用事件，并将订单状态更改为OPEN。



更复杂的情况可能涉及额外的步骤，例如在检查客户信用的同时预留库存。

如果 (a) 每个服务原子性地更新数据库并发布事件（更多），(b) Message Broker保证事件至少传递一次，那么您可以实现跨多个服务的业务事务。重要的是要注意，这些不是ACID事务。它们提供了更弱的保证，例如[最终的一致性](#)。这个交易模型被称为**BASE模型**。

您还可以使用事件来维护预先加入由多个微服务拥有的数据的物化视图。维护视图的服务订阅相关事件并更新视图。例如，维护“客户订单”视图的“客户订单视图更新程序”服务订阅客户服务和订单服务发布的事件。



当客户订单视图更新程序服务收到客户或订单事件时，它会更新客户订单视图数据存储。您可以使用文档数据库（如MongoDB）实现客户订单视图，并为每个客户存储一个文档。客户订单视图查询服务通过查询客户订单视图数据存储来处理客户和最近订单的请求。

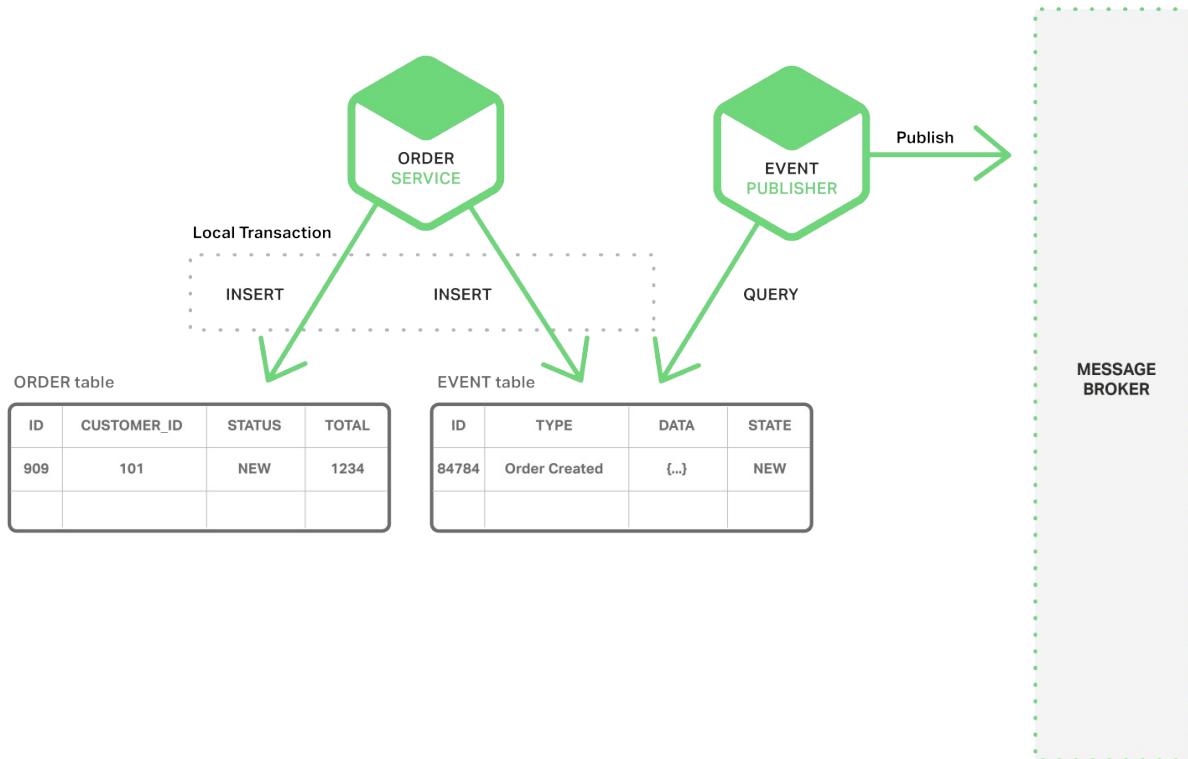
事件驱动架构有几个好处和缺点。它实现跨多个服务的事务的实现，并提供最终的一致性。另一个好处是，它还使应用程序能够维护物化视图。一个缺点是编程模型比使用ACID事务时更复杂。通常，您必须实现补偿事务以从应用程序级故障中恢复；例如，如果信用检查失败，您必须取消订单。此外，应用程序必须处理不一致的数据。这是因为飞行中交易所做的更改是可见的。如果应用程序从尚未更新的实例化视图读取，那么应用程序也可以看到不一致。另一个缺点是订户必须检测和忽略重复的事件。

实现原子性

在事件驱动的架构中，还存在原子级更新数据库和发布事件的问题。例如，订单服务必须在ORDER表中插入一行，并发布订单创建事件。这两个操作必须原子性地完成。如果服务在更新数据库之后但在发布事件之前崩溃，则系统会不一致。确保原子性的标准方法是使用涉及数据库和Message Broker的分布式事务。然而，由于上述原因，如CAP定理，这正是我们不想做的。

使用本地事务发布事件

实现原子性的一种方式是应用程序使用仅涉及本地事务的多步骤过程来发布事件。诀窍是在存储业务实体的状态的数据库中有一个EVENT表，该表用作消息队列。应用程序开始（本地）数据库事务，更新业务实体的状态，将事件插入EVENT表，并提交事务。单独的应用程序线程或进程查询EVENT表，将事件发布到Message Broker，然后使用本地事务将事件标记为已发布。下图显示了设计。



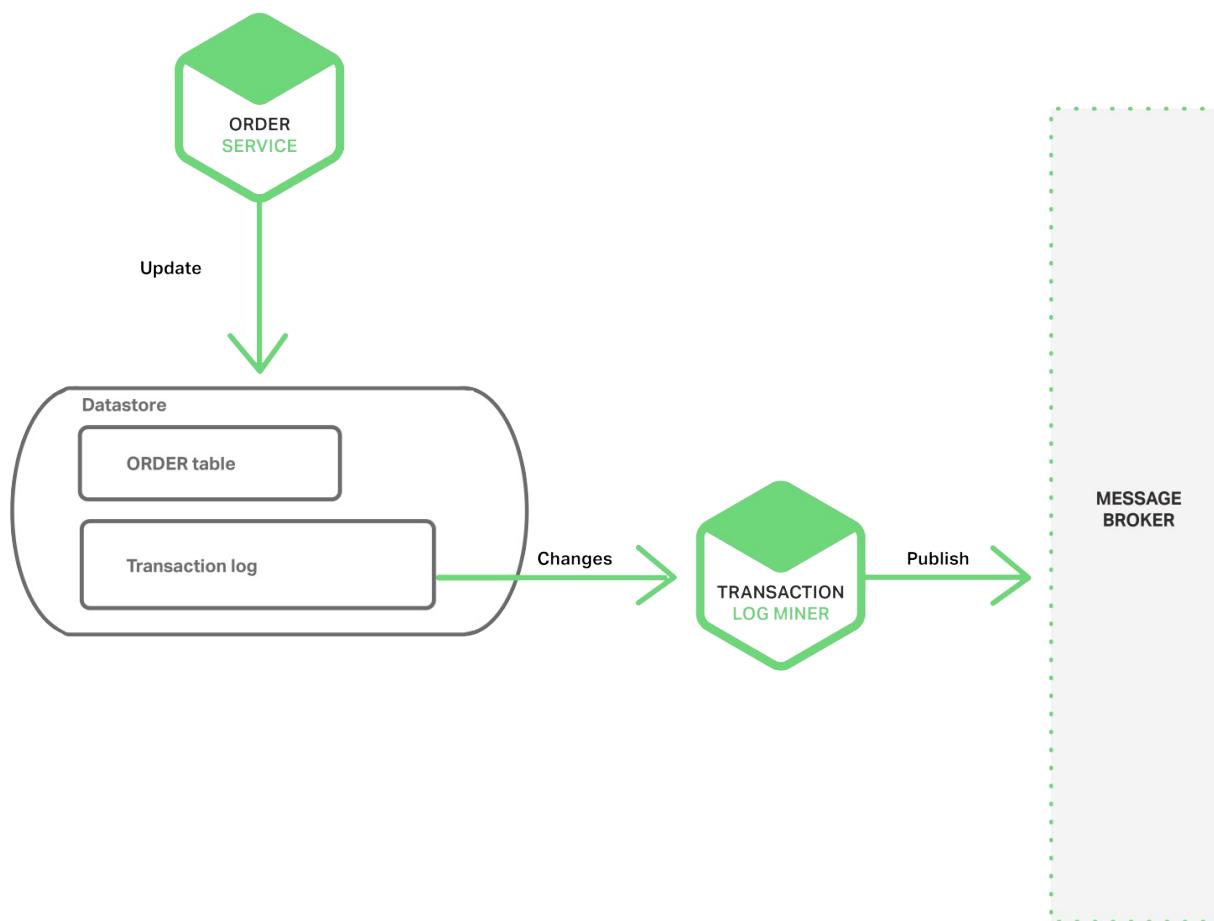
Order Service在ORDER表中插入一行，并将一个Order Created事件插入EVENT表中。事件发布器线程或进程查询EVENT表以查找未发布的事件，发布事件，然后更新EVENT表以将事件标记为已发布。

这种方法有几个好处和缺点。一个好处是，它保证为每个更新发布事件而不依赖于2PC。此外，应用程序发布业务级事件，这消除了推断它们的需要。这种方法的一个缺点是它潜在地容易出错，因为开发人员必须记住发布事件。这种方法的局限性在于，当使用一些NoSQL数据库时，由于它们的有限事务和查询能力，实施是有挑战性的。

此方法通过让应用程序使用本地事务来更新状态和发布事件，从而消除了对2PC的需要。让我们来看看通过让应用程序简单地更新状态来实现原子性的方法。

挖掘数据库事务日志

另一种实现原子性而不使用2PC的方法是通过挖掘数据库事务或提交日志的线程或进程发布事件。应用程序更新数据库，导致更改记录在数据库的事务日志中。事务日志Miner线程或进程读取事务日志并将事件发布到Message Broker。下图显示了设计。



这种方法的一个例子是开源的[LinkedIn数据库](#)项目。Databus挖掘Oracle事务日志并发布与更改相对应的事件。LinkedIn使用Databus来保持各种派生数据存储与记录系统一致。

另一个示例是[AWS DynamoDB](#)中的[流机制](#)，这是一个托管NoSQL数据库。DynamoDB流包含在过去24小时内对DynamoDB表中的项目进行的更改（创建，更新和删除操作）的按时间排序的顺序。应用程序可以从流中读取这些更改，例如，将其作为事件发布。

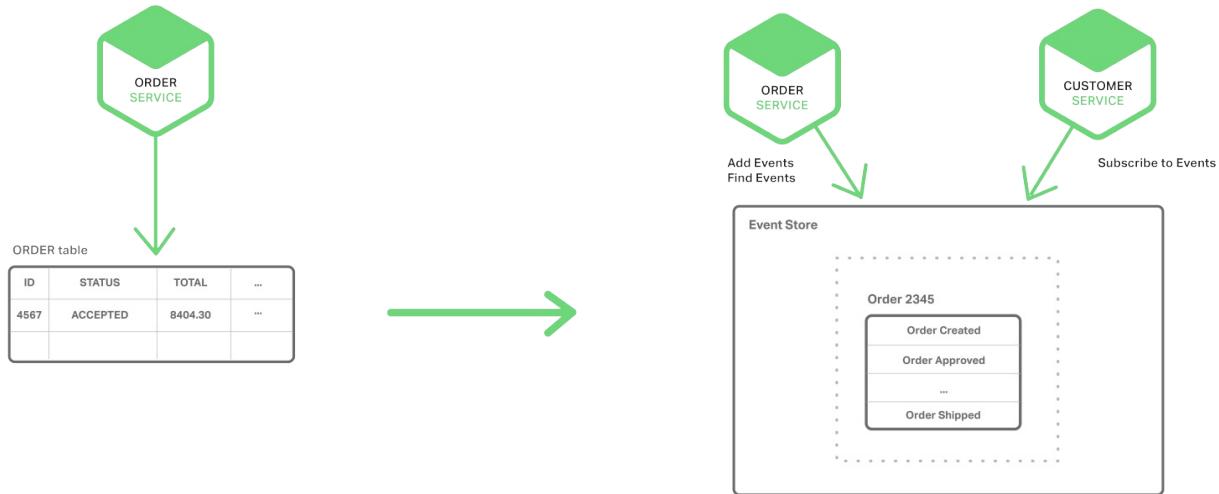
事务日志挖掘具有各种好处和缺点。一个好处是，它保证为每个更新发布事件，而不使用2PC。事务日志挖掘还可以通过将事件发布与应用程序的业务逻辑分离来简化应用程序。主要缺点是事务日志的格式是每个数据库专有的，甚至可能在数据库版本之间更改。此外，可能难以从记录在事务日志中的低级更新逆向工程化高级业务事件。

事务日志挖掘消除了对2PC的需要，让应用程序做一件事：更新数据库。现在让我们看一个不同的方法，消除更新，并完全依赖事件。

使用事件源

[事件源](#)通过使用完全不同的，以事件为中心的方法来持久化商业实体来实现没有2PC的原子性。不是存储实体的当前状态，而是存储状态改变事件的序列。应用程序通过重放事件来重建实体的当前状态。每当业务实体的状态改变时，新事件被附加到事件列表。由于保存事件是单个操作，它本质上是原子的。

要查看事件源如何工作，请考虑Order实体作为示例。在传统方法中，每个顺序映射到ORDER表中的一行以及例如ORDER_LINE_ITEM表中的行。但是当使用事件源时，订单服务以其状态改变事件的形式存储订单：已创建，已批准，已发货，已取消。每个事件包含足够的数据来重建Order的状态。



事件存储在事件存储中，事件存储是事件的数据库。商店具有用于添加和检索实体的事件的API。事件存储器的行为类似于我们之前描述的架构中的**Message Broker**。它提供了一个API，使服务能够订阅事件。事件存储将所有事件提供给所有感兴趣的订阅者。事件存储是事件驱动的微服务架构的主干。

事件源有几个好处。它解决了实现事件驱动架构中的一个关键问题，并且使得每当状态改变时可靠地发布事件成为可能。因此，它解决了微服务架构中的数据一致性问题。此外，因为它持久化事件而不是域对象，它大部分避免了**对象关系阻抗失配问题**。事件源还提供对业务实体所做的更改的100%可靠的审计日志，并且使得可以在任何时间点确定实体的状态的时态查询。事件源的另一个主要优点是，您的业务逻辑包括交换事件的松散耦合的业务实体。

事件源也有一些缺点。它是一种不同的和不熟悉的编程风格，所以有一个学习曲线。事件存储器仅直接支持通过主键查找业务实体。您必须使用**命令查询责任分离（CQRS）**来实现查询。因此，应用程序必须处理最终一致的数据。

概要

在微服务体系结构中，每个微服务都有自己的私有数据存储。不同的微服务可能使用不同的SQL和NoSQL数据库。虽然此数据库架构具有显着的优势，但它会产生一些分布式数据管理挑战。第一个挑战是如何实现在多个服务之间保持一致性的业务事务。第二个挑战是如何实现从多个服务检索数据的查询。

对于许多应用程序，解决方案是使用事件驱动架构。实现事件驱动架构的一个挑战是如何以原子方式更新状态以及如何发布事件。有几种方法可以实现这一点，包括使用数据库作为消息队列，事务日志挖掘和事件源。

在将来的博客文章中，我们将继续深入了解微服务的其他方面。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 微服务的事件驱动数据管理（本文）
6. 选择微服务部署策略
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略（本文）
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这是关于使用微服务构建应用程序的第六篇文章。在[第一篇文章](#)介绍了微服务架构模式，并讨论的好处和使用微服务的缺点。以下文章讨论微服务架构的不同方面：[使用API网关](#)，[进程间通信](#)，[服务发现](#)和[事件驱动的数据管理](#)。在本文中，我们讨论部署微服务的策略。

动机

部署单片应用程序意味着运行单个通常是大型应用程序的多个相同副本。您通常配置N个服务器（物理或虚拟），并在每个服务器上运行应用程序的M个实例。单片应用程序的部署并不总是完全直接的，但它比部署微服务应用程序简单得多。

一个微服务的应用程序包含几十个甚至数百个服务的。服务以各种语言和框架编写。每个都是具有自己的特定部署，资源，扩展和监视要求的小型应用程序。例如，您需要根据该服务的需求运行每个服务的一定数量的实例。此外，必须为每个服务实例提供适当的CPU，内存和I/O资源。更有挑战性的是，尽管这种复杂性，部署服务必须快速，可靠和具有成本效益。

有几种不同的微服务部署模式。让我们首先看一下每个主机的多个服务实例模式。

每个主机模式的多个服务实例

部署微服务的一种方法是使用[每个主机的多个服务实例](#)模式。使用此模式时，您可以配置一个或多个物理或虚拟主机，并在每个主机上运行多个服务实例。在许多方面，这是传统的应用部署方法。每个服务实例在一个或多个主机上的公知端口运行。主机通常像[宠物一样治疗](#)。

下图显示了此模式的结构。

Host (Physical or VM)



Host (Physical or VM)



有这种模式的几个变体。一个变体是每个服务实例是进程或进程组。例如，您可以将Java服务实例部署为Apache Tomcat服务器上的Web应用程序。一个Node.js的服务实例可能包括父进程和一个或更多的子进程。

此模式的另一个变体是在同一进程或进程组中运行多个服务实例。例如，您可以在同一Apache Tomcat服务器上部署多个Java Web应用程序，或在同一OSGI容器中运行多个OSGI捆绑软件。

每个主机模式的多个服务实例具有优点和缺点。一个主要好处是其资源使用相对有效。多个服务实例共享服务器及其操作系统。如果进程或进程组运行多个服务实例（例如，共享同一Apache Tomcat服务器和JVM的多个Web应用程序），则效率更高。

这种模式的另一个好处是部署服务实例相对较快。您只需将服务复制到主机并启动它。如果服务是用Java编写的，那么您将复制JAR或WAR文件。对于其他语言（如Node.js或Ruby），您可以复制源代码。在任一种情况下，通过网络复制的字节数相对较小。

此外，由于缺乏开销，启动服务通常非常快。如果服务是它自己的进程，你只需启动它。否则，如果服务是在同一容器进程或进程组中运行的几个实例之一，则可以将其动态部署到容器中或重新启动容器。

尽管它的吸引力，每个主机的多个服务实例具有一些显着的缺点。一个主要的缺点是很少或没有隔离服务实例，除非每个服务实例是单独的进程。虽然可以准确监视每个服务实例的资源利用率，但不能限制每个实例使用的资源。运行错误的服务实例可能会占用主机的所有内存或CPU。

如果多个服务实例在同一进程中运行，则根本没有隔离。例如，所有实例可能共享同一个JVM堆。错误的服务实例可能容易打破在同一进程中运行的其他服务。此外，您无法监视每个服务实例使用的资源。

这种方法的另一个重要问题是部署服务的操作团队必须知道如何做的具体细节。服务可以用多种语言和框架编写，因此开发团队必须与操作共享许多细节。这种复杂性增加了部署期间错误的风险。

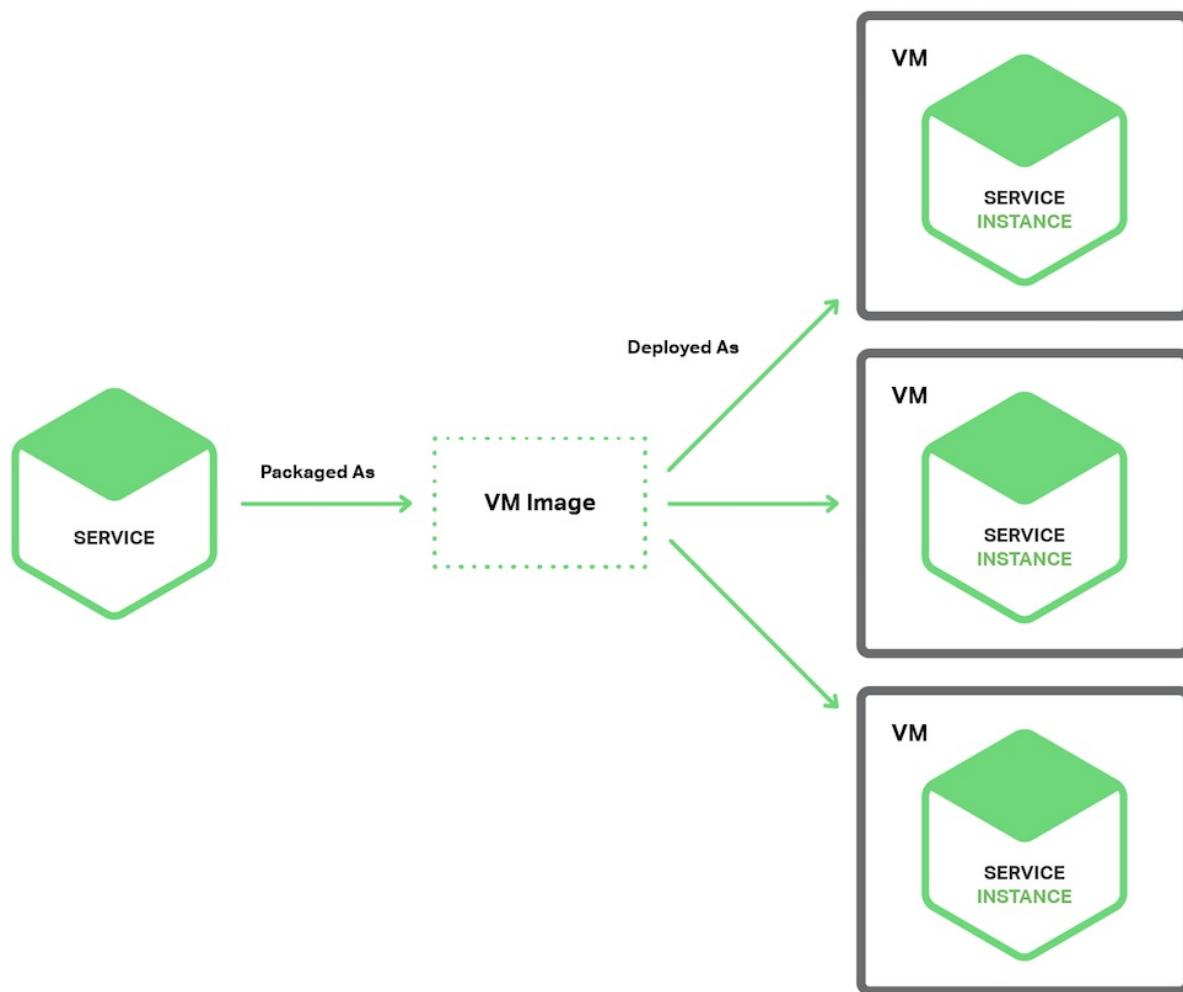
正如你所看到的，尽管它熟悉，每个主机模式的多个服务实例有一些显着的缺点。现在让我们看看部署微服务的其他方法，避免这些问题。

每个主机模式的服务实例

部署[微服务的](#)另一种方法是[每个主机服务实例](#)模式。使用此模式时，您可以在其自己的主机上单独运行每个服务实例。此模式有两种不同的特殊化：每个虚拟机的服务实例和每个容器的服务实例。

每个虚拟机模式的服务实例

当您使用[每个虚拟机服务实例](#)模式时，您将每个服务打包为虚拟机（VM）映像，例如[Amazon EC2 AMI](#)。每个服务实例是使用该VM映像启动的VM（例如，EC2实例）。下图显示了此模式的结构：



这是Netflix用来部署其视频流服务的主要方法。Netflix使用[Aminator](#)将其每项服务打包为EC2 AMI。每个正在运行的服务实例都是EC2实例。

有多种工具可用于构建自己的VM。您可以配置持续集成（CI）服务器（例如[Jenkins](#)）以调用Aminator将您的服务打包为EC2 AMI。[Packer.io](#)是创建自动VM映像的另一个选项。与Aminator不同，它支持各种虚拟化技术，包括EC2，DigitalOcean，VirtualBox和VMware。

[Boxfuse](#)公司有一个令人信服的方式来构建VM映像，克服了我在下面描述的VM的缺点。Boxfuse将您的Java应用程序打包为最小的VM映像。这些图像快速构建，快速启动，更安全，因为它们暴露了有限的攻击面。

公司[CloudNative](#)有面包店，一个用于创建EC2 AMI的SaaS产品。您可以将CI服务器配置为在对微服务传递进行测试后调用面包店。然后面包店将您的服务打包为AMI。使用SaaS产品（如面包店）意味着您不必浪费宝贵的时间来设置AMI创建基础架构。

每个虚拟机服务实例模式具有许多优点。VM的一个主要优点是每个服务实例以完全隔离的方式运行。它具有固定数量的CPU和内存，不能从其他服务窃取资源。

将微服务部署为VM的另一个好处是，您可以利用成熟的云基础架构。AWS等AWS提供了有用的功能，如负载平衡和自动扩展。

将服务部署为VM的另一个好处是它封装了您的服务的实现技术。一旦服务被打包为VM，它将变成一个黑盒子。VM的管理API成为部署服务的API。部署变得更简单和更可靠。

然而，每个虚拟机服务实例模式有一些缺点。一个缺点是资源利用效率较低。每个服务实例都有整个VM的开销，包括操作系统。此外，在典型的公共IaaS中，VM具有固定大小，并且VM可能未充分利用。

移动，公共IaaS通常对VM收费，而不管它们是忙还是空闲。诸如AWS之类的IaaS提供自动缩放，但是[很难对需求变化做出快速反应](#)。因此，您经常必须过度配置VM，这增加了部署成本。

这种方法的另一个缺点是部署新版本的服务通常很慢。VM映像由于其大小通常缓慢构建。此外，VM通常很慢实例化，也是因为它们的大小。此外，操作系统通常需要一些时间来启动。然而，请注意，这并不是普遍真实的，因为存在轻量级VM，如由Boxfuse构建的VM。

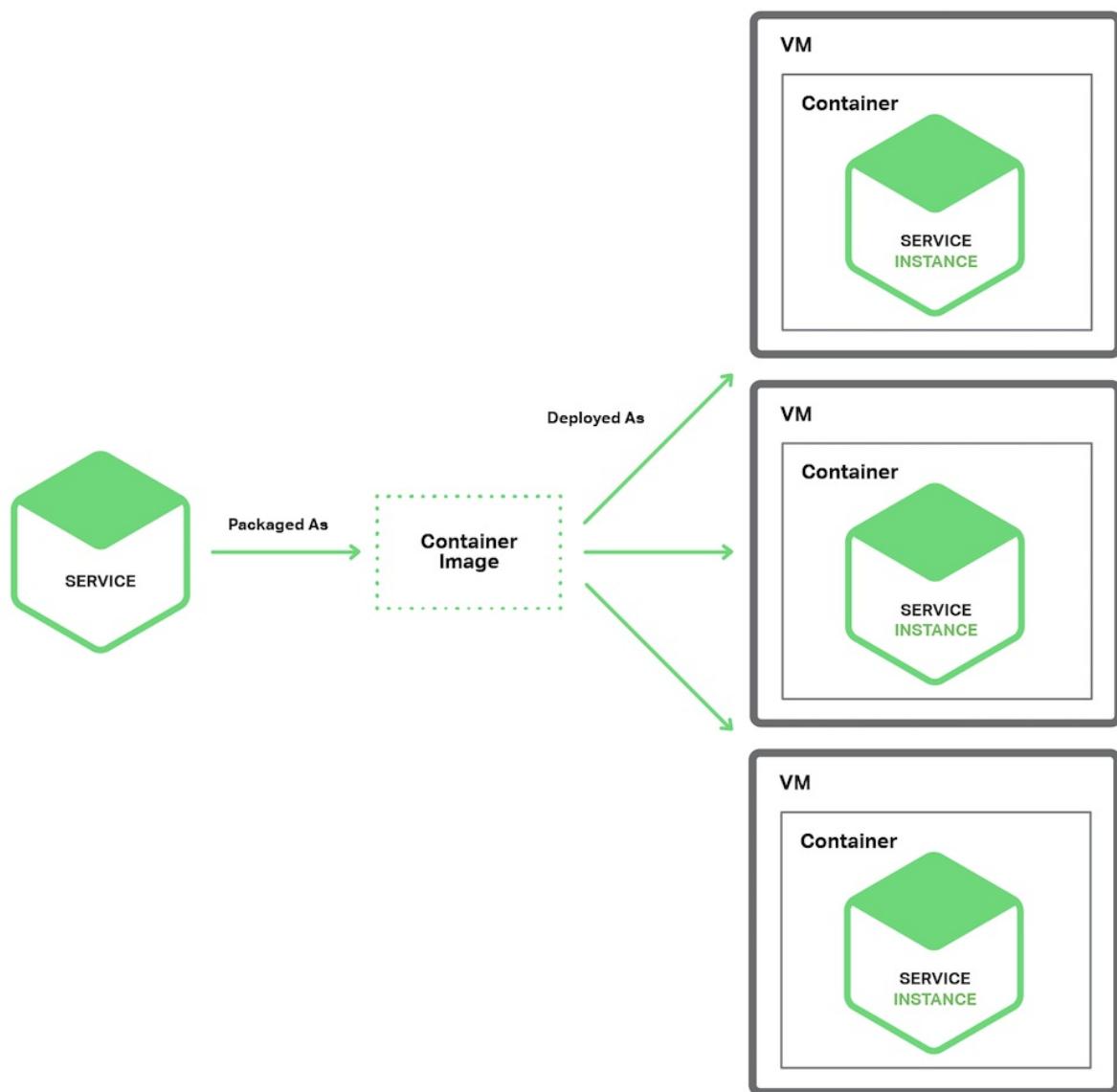
每个虚拟机服务实例模式的另一个缺点是，通常你（或你的组织中的其他人）负责许多未分化的繁重工作。除非您使用Boxfuse等工具来处理构建和管理VM的开销，否则这是您的责任。这种必要但耗时的活动分散了您的核心业务。

现在让我们看一个部署微服务的替代方法，它更轻量，但仍然具有虚拟机的许多好处。

每个容器模式的服务实例

当您使用[每个容器的服务实例](#)模式时，每个服务实例都在其自己的容器中运行。容器是[操作系统级别的虚拟化机制](#)。容器由在沙箱中运行的一个或多个进程组成。从进程的角度来看，它们有自己的端口命名空间和根文件系统。您可以限制容器的内存和CPU资源。一些容器实现也具有I/O速率限制。容器技术的例子包括Docker和Solaris Zones。

下图显示了此模式的结构：



要使用此模式，请将服务打包为容器图像。容器映像是由运行服务所需的应用程序和库组成的文件系统映像。一些容器镜像包含一个完整的Linux根文件系统。其他更轻量级。例如，要部署Java服务，您将构建一个包含Java运行时（可能是Apache Tomcat服务器）和已编译的Java应用程序的容器映像。

将服务打包为容器映像后，即可启动一个或多个容器。通常在每个物理或虚拟主机上运行多个容器。您可以使用集群管理器（如[Kubernetes](#)或[Marathon](#)）来管理容器。集群管理器将主机视为资源池。它根据容器所需的资源和每个主机上可用的资源，决定放置每个容器的位置。

每个容器服务实例模式具有优点和缺点。容器的优点与VM类似。它们将服务实例彼此隔离。您可以轻松地监视每个容器使用的资源。此外，像VM一样，容器封装了用于实现您的服务的技术。容器管理API也用作管理您的服务的API。

然而，与VM不同，容器是一种轻量级技术。容器图像通常构建起来非常快。例如，在我的笔记本电脑上，将[Spring Boot](#)应用程序打包为Docker容器只需5秒钟。容器也启动非常快，因为没有冗长的操作系统启动机制。当容器启动时，运行的是服务。

使用容器有一些缺点。虽然容器基础设施正在快速成熟，但它不像虚拟机的基础设施那么成熟。另外，容器不如VM安全，因为容器与主机OS共享内核。

容器的另一个缺点是，你负责管理容器图像的未分化繁重。此外，除非您使用托管容器解决方案（如[Google容器引擎](#)或[Amazon EC2容器服务（ECS）](#)），否则您必须管理容器基础架构以及可能运行的基础架构。

此外，容器通常部署在具有每个VM定价的基础设施上。因此，如前所述，您可能会承担过度配置虚拟机的额外成本，以处理负载峰值。

有趣的是，容器和VM之间的区别很可能模糊。如前所述，Boxfuse VM快速构建和启动。该[清除容器](#)项目旨在打造轻量级虚拟机。人们对[unikernels](#)也越来越感兴趣。Docker最近收购了Unikernel Systems。

还有更新的和日益流行的无服务器部署概念，这是一种解决必须在部署容器或VM中的服务之间进行选择的问题的方法。让我们来看看下一个。

无服务器部署

[AWS Lambda](#)是无服务器部署技术的一个示例。它支持Java，Node.js和Python服务。要部署微服务，请将其打包为ZIP文件，并将其上传到AWS Lambda。您还提供元数据，其中包括指定调用以处理请求（也称为事件）的函数的名称。AWS Lambda自动运行足够的微服务实例来处理请求。根据所用时间和内存消耗，您只需为每个请求计费。当然，魔鬼在细节，你会很快看到AWS Lambda有局限性。但是，作为您的组织中的开发人员或任何人都不需要担心服务器，虚拟机或容器的任何方面的概念是令人难以置信的吸引力。

一个lambda函数是一个无状态的服务。它通常通过调用AWS服务处理请求。例如，当图像上传到S3存储桶时调用的Lambda函数可以将项目插入DynamoDB图像表，并将消息发布到Kinesis流以触发图像处理。Lambda函数还可以调用第三方Web服务。

有四种方法来调用Lambda函数：

1. 直接，使用Web服务请求
2. 自动响应由AWS服务（如S3，DynamoDB，Kinesis或简单电子邮件服务）生成的事件
3. 自动通过AWS API网关处理来自应用程序客户端的HTTP请求
4. 定期，根据 cron 类似的时间表

如您所见，AWS Lambda是部署微服务的一种方便的方法。基于请求的定价意味着您只需为您的服务实际执行的工作付费。此外，因为您不负责IT基础架构，您可以专注于开发应用程序。

然而，存在一些显着的限制。它不用于部署长时间运行的服务，例如使用来自第三方消息代理的消息的服务。请求必须在300秒内完成。服务必须是无状态的，因为在理论上AWS Lambda可能为每个请求运行单独的实例。它们必须以支持的语言之一编写。服务也必须快速

启动;否则，它们可能超时并终止。

概要

部署微服务应用程序具有挑战性。有几十甚至几百种服务以各种语言和框架编写。每个都是一个小型应用程序，有自己的特定部署，资源，扩展和监控要求。有几种微服务部署模式，包括每个虚拟机的服务实例和每个容器的服务实例。部署微服务的另一个有趣的选择是 AWS Lambda，一种无服务器方法。在本系列的下一部分和最后一部分中，我们将讨论如何将单片应用程序迁移到微服务体系结构。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略（本文）
7. 将重组重构为微服务

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 重构一个Monolith到微服务（本文）

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。

这是我的系列中关于使用微服务构建应用程序的第七篇和最后一篇文章。在[第一篇文章](#)介绍了[微服务架构模式](#)，并讨论的好处和使用微服务的缺点。以下文章讨论微服务架构的不同方面：[使用API网关](#)，[进程间通信](#)，[服务发现](#)，[事件驱动的数据管理和部署微服务](#)。在本文中，我们将讨论将单片应用程序迁移到微服务的策略。

我希望这个系列的文章给你一个很好的理解微服务架构，其优点和缺点，以及何时使用它。也许微服务架构是一个很适合你的组织。

但是，有一个很好的机会，你正在一个大型，复杂的单片应用程序。您开发和部署您的应用程序的每日经验是缓慢和痛苦。微服务似乎是一个遥远的涅。。幸运的是，有一些策略，你可以使用从整体地狱逃脱。在本文中，我将描述如何将单片应用程序逐步重构为一组微服务。

重构到微服务概述

转化单片应用到微服务的过程是一种形式[应用现代化](#)。这是开发人员已经做了几十年的事情。因此，我们可以在将应用程序重构到微服务时重用一些想法。

不使用的一个策略是“大爆炸”重写。这就是当你把所有的开发工作集中在从头开始构建一个新的基于微服务的应用程序。虽然听起来很吸引人，但它是非常危险的，并且很可能会以失败告终。正如Martin Fowler所说，“大爆炸重写保证的唯一的东西是大爆炸！”

而不是大爆炸重写，你应该递增重构你的单片应用程序。您逐步构建一个由微服务组成的新应用程序，并与您的单片应用程序一起运行。随着时间的推移，由单片应用程序实现的功能量收缩，直到它完全消失或者它只是另一个微服务。这种策略类似于在70英里/小时的速度下行驶在高速公路上时为您的车辆提供服务 - 具有挑战性，但远远没有尝试大爆炸重写的风险。

Martin Fowler将此应用程序现代化策略称为“[Strangler应用程序](#)”。这个名字来自扼杀者的藤蔓（aka strangler fig）在热带雨林中发现。扼制者藤蔓生长在一棵树上，以便到达森林树冠上的阳光。有时，树死了，留下一棵树形的藤。应用程序现代化遵循相同的模式。我们将构建

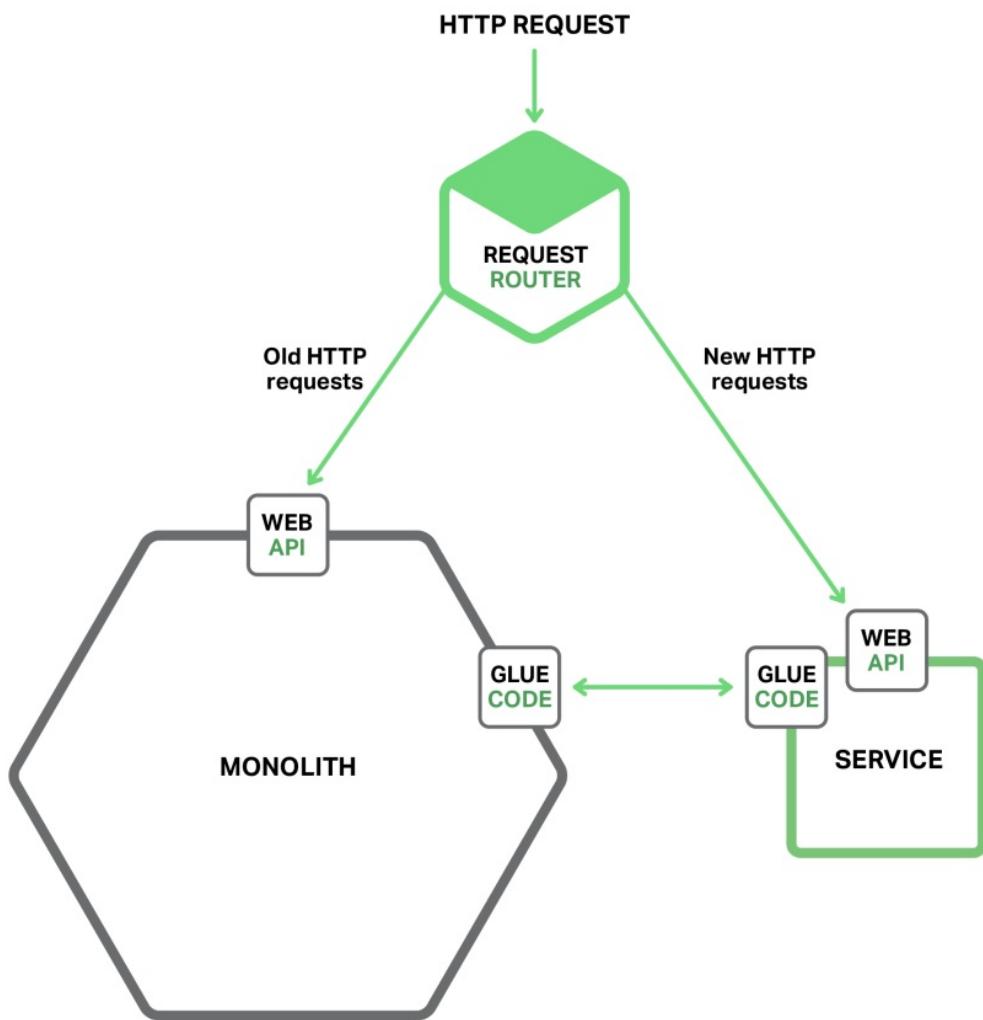
一个由遗留应用程序周围的微服务组成的新应用程序，这将最终死亡。



让我们看看这样做的不同策略。

策略1 - 停止挖掘

该孔的法说，每当你在一个洞，你应该停止挖掘。当你的整体应用程序变得难以管理时，这是很好的建议。换句话说，你应该停止使巨石更大。这意味着当你实现新的功能，你不应该添加更多的代码到巨石。相反，这个策略的一个大想法是将这个新代码放在一个独立的微服务中。下图显示了应用此方法后的系统架构。



除了新服务和传统的整体外，还有两个其他组件。第一个是请求路由器，它处理传入（HTTP）请求。它类似于前面的文章中描述的API网关。路由器向新服务发送对应于新功能的请求。它将遗留请求路由到整体。

另一个组件是粘合代码，它将服务与整体集成。服务很少孤立存在，并且通常需要访问整体所拥有的数据。粘合代码，位于单片机，服务或两者，负责数据集成。该服务使用胶水代码来读取和写入整体所拥有的数据。

服务可以使用三个策略来访问巨庞的数据：

- 调用整体提供的远程API
- 直接访问巨庞的数据库
- 维护自己的数据副本，这是与巨量的数据库同步

胶水代码有时被称为反腐败层。这是因为粘合代码阻止了具有其自己的原始域模型的服务被遗留巨作的领域模型的概念污染。粘合代码在两个不同的模型之间进行转换。术语反腐败层首先出现在必读的[域名驱动设计](#)由埃里克·埃文斯，然后在一个[白皮书](#)精炼。发展反腐败层可以是一项不小的工作。但是，如果你想成长你的方式从整体地狱是必要的创造一个。

将新功能实现为轻量级服务具有几个好处。它防止巨石变得更加难以管理。该服务可以独立于整体开发，部署和扩展。对于您创建的每个新服务，您都会体验到微服务架构的优势。

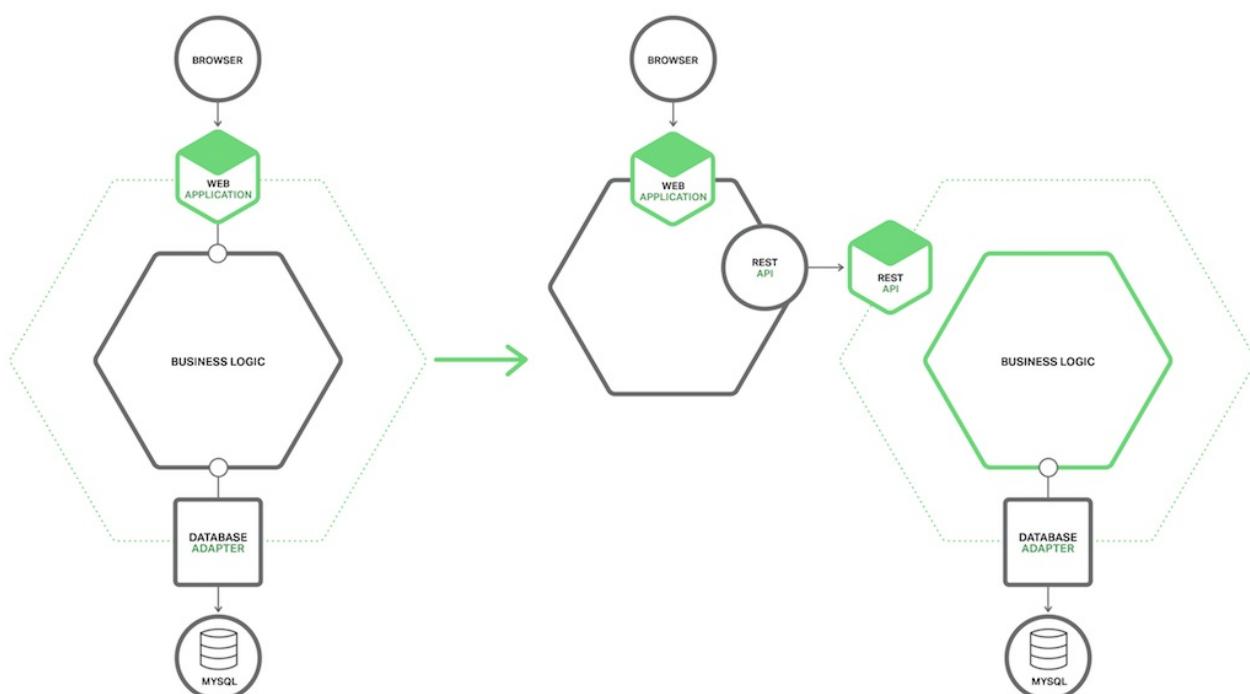
然而，这种方法没有解决整体的问题。要解决这些问题，你需要打破巨石。让我们看看这样做的策略。

策略2 - 拆分前端和后端

缩小整体应用程序的策略是将表示层与业务逻辑和数据访问层分离。典型的企业应用程序由至少三种不同类型的组件组成：

- 表示层 - 处理HTTP请求并实现（REST）API或基于HTML的Web UI的组件。在具有复杂用户界面的应用程序中，表示层通常是大量代码。
- 业务逻辑层 - 作为应用程序核心并实现业务规则的组件。
- 数据访问层 - 访问基础结构组件（如数据库和消息代理）的组件。

通常一方面的表示逻辑与另一方面业务和数据访问逻辑之间存在清晰的分离。业务层具有包含一个或多个外观的粗粒度API，其封装业务逻辑组件。这个API是一个自然缝，您可以沿着这个缝将整块分割成两个较小的应用程序。一个应用程序包含表示层。另一个应用程序包含业务和数据访问逻辑。分割后，表示逻辑应用程序对业务逻辑应用程序进行远程调用。下图显示了重构之前和之后的架构。



以这种方式分割整体有两个主要好处。它使您能够独立于彼此开发，部署和扩展这两个应用程序。特别地，其允许表示层开发者在用户界面上快速迭代，并且例如容易地执行A|B测试。这种方法的另一个好处是，它暴露了一个远程API，可以由您开发的微服务调用。

然而，这种策略只是部分解决方案。很可能一个或两个应用程序将是一个不可管理的整体。您需要使用第三个策略来消除剩余的整体或整体。

战略3 - 提取服务

第三个重构策略是将单片机中的现有模块转换为独立的微服务。每次你提取一个模块，并把它变成一个服务，这个单块收缩。一旦你已经转换了足够的模块，这个单块将不再是一个问题。它或者它完全消失或者变得足够小，它只是另一个服务。

确定将哪些模块转换为服务的优先级

一个大的，复杂的单片应用程序包括几十或几百个模块，所有这些模块都是提取的候选。确定首先转换哪些模块通常是具有挑战性的。一个好的方法是从一些容易提取的模块开始。这将给你提供微服务的经验，特别是提取过程。之后，你应该提取那些将给你最大的好处的模块。

将模块转换为服务通常是耗时的。你想通过你将收到的好处排序模块。提取频繁变化的模块通常是有益的。将模块转换为服务后，您可以独立于整体开发和部署模块，这将加速开发。

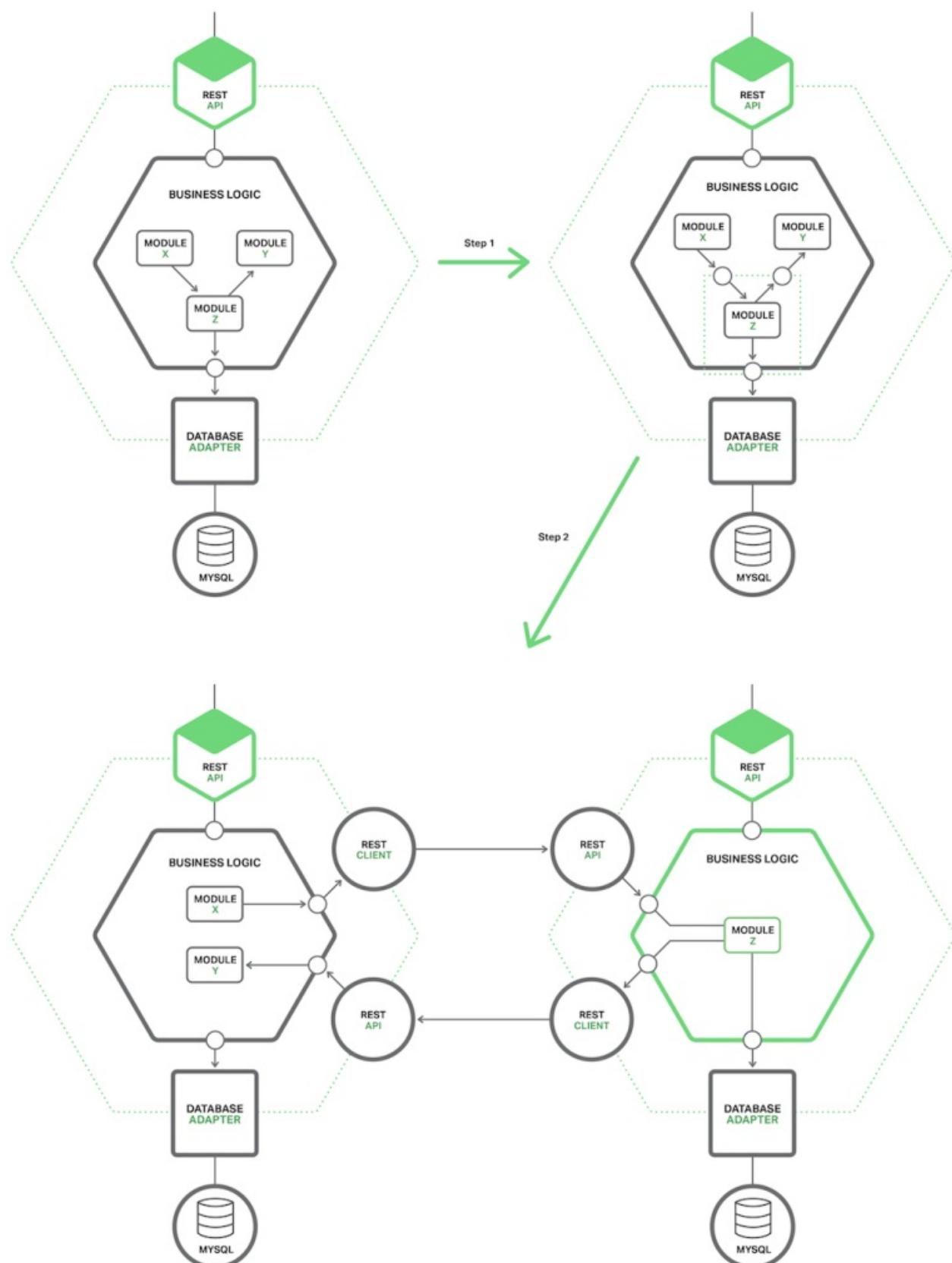
提取具有明显不同于整块的其余部分的资源需求的模块也是有益的。例如，将具有内存数据库的模块转换为服务是有用的，然后可以将服务部署在具有大量内存的主机上。类似地，可以值得提取实现计算昂贵的算法的模块，因为服务然后可以部署在具有大量CPU的主机上。通过将具有特定资源需求的模块转换为服务，您可以使您的应用程序更容易扩展。

当确定要提取哪些模块时，查找现有的粗粒边界（又名接缝）是有用的。它们使得将模块转换为服务变得更容易和更便宜。这种边界的示例是仅通过异步消息与应用程序的其余部分通信的模块。它可以相对便宜并且容易地将该模块转换成微服务。

如何提取模块

提取模块的第一步是在模块和单块之间定义粗粒度接口。它很可能是一个双向API，因为这个单片将需要服务拥有的数据，反之亦然。由于模块和应用程序的其余部分之间的纠缠的依赖性和细粒度的交互模式，实现这样的API通常是具有挑战性的。使用[域模型模式](#)实现的业务逻辑对于重构是特别具有挑战性的，因为领域模型类之间存在大量关联。您经常需要进行重要的代码更改以打破这些依赖关系。下图显示了重构。

一旦实现粗粒度接口，然后将模块转换为独立服务。为此，您必须编写代码以使单片和服务通过使用[进程间通信（IPC）](#)机制的API进行通信。下图显示重构之前，期间和之后的体系结构。



在本示例中，模块Z是要提取的候选模块。它的组件由模块X使用，它使用模块Y.第一个重构步骤是定义一对粗粒度的API。第一个接口是模块X用于调用模块Z的入站接口。第二个接口是模块Z用于调用模块Y的出站接口。

第二重构步骤将模块转换为独立服务。入站和出站接口由使用IPC机制的代码实现。您很可能需要通过将模块Z与[微服务机箱框架相结合](#)来构建服务，该服务框架处理诸如服务发现的交叉关注。

一旦你提取了一个模块，你还有另一个服务，可以开发，部署和独立于巨石和任何其他服务进行扩展。你甚至可以从头重写服务；在这种情况下，将服务与整体集成的API代码成为在两个域模型之间转换的反损坏层。每次提取服务时，都会沿着微服务的方向迈出第一步。随着时间的推移，巨石将缩水，你将有越来越多的微服务。

概要

将现有应用程序迁移到微服务的过程是应用程序现代化的一种形式。您不应该从头重写您的应用程序移动到微服务。相反，你应该递增地将你的应用程序重构为一组微服务。有三个策略可以使用：实现新的功能作为微服务；从业务和数据访问组件分离呈现组件；并将整体中的现有模块转换为服务。随着时间的推移，微服务的数量将会增长，您的开发团队的敏捷性和速度将会增加。

这七个系列的文章现在完成：

1. 微服务介绍
2. 构建微服务：使用API网关
3. 构建微服务：微服务架构中的进程间通信
4. 微服务架构中的服务发现
5. 事件驱动的数据管理微服务
6. 选择微服务部署策略
7. 重构一个Monolith到微服务（本文）

您还可以下载完整的文章集，以及使用NGINX Plus实现微服务的信息，作为电子书 -[微服务：从设计到部署](#)。