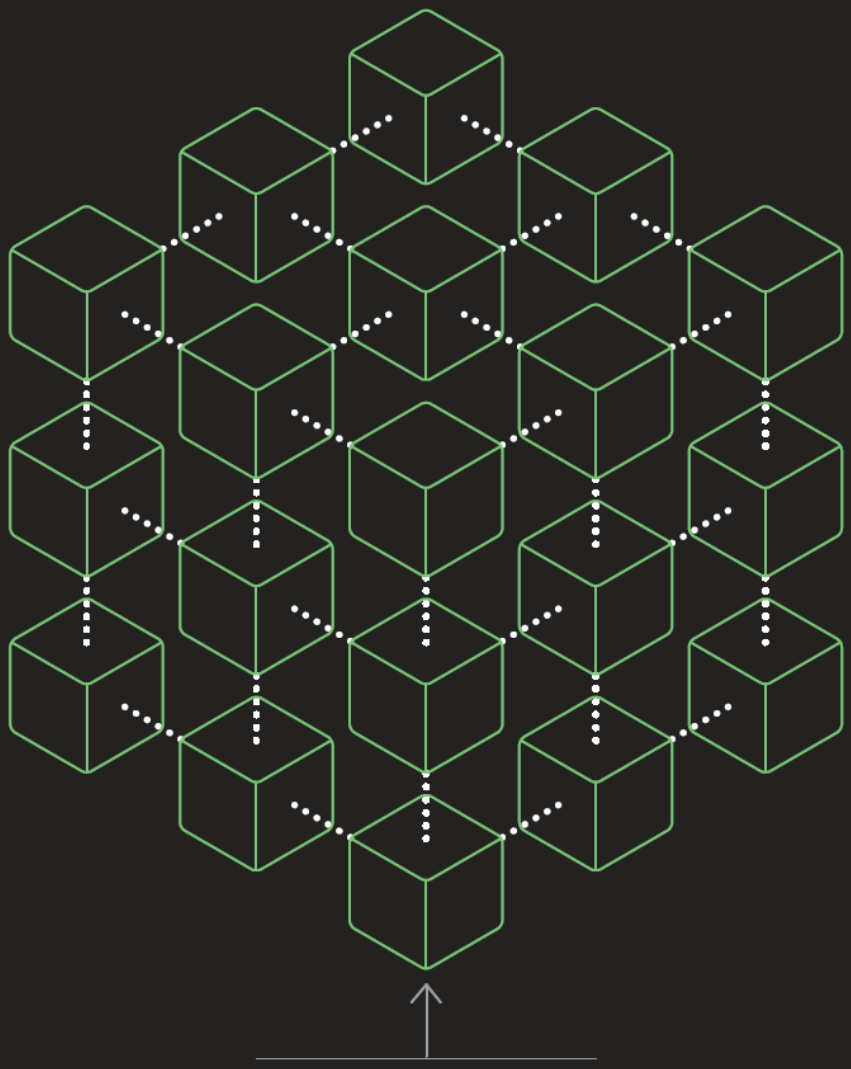
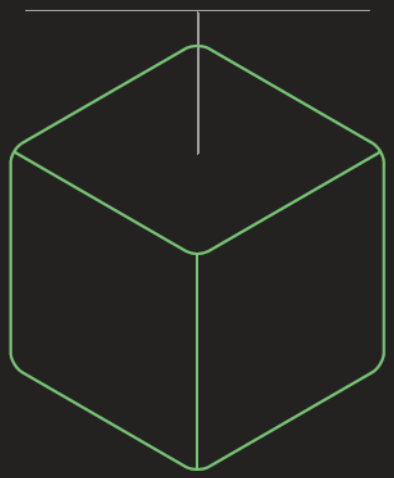


DESIGNING AND DEPLOYING MICROSERVICES

微服务：从设计到部署



NGINX

作者：Chris Richardson　Floyd Smith　　译者：Oopsguy

目录

关于本书

Oopsguy

本书为 Chris Richardson 和 Floyd Smith 联合编写的微服务电子书 Designing and Deploying Microservices 中文版，其从不同角度全面介绍了微服务：微服务的优点与缺点、API 网关、进程间通信（IPC）、服务发现、事件驱动数据管理、微服务部署策略、重构单体。

- github 仓库：https://github.com/oopsguy/microservices-from-design-to-deployment-chinese
- 在线阅读：http://oopsguy.com/books/microservices

本书对 Nginx 的描述不是很多，主要针对微服务领域。如果您想了解更多关于 Nginx 的内容，请参阅正在更新的 Nginx 中文文档。

这是本人第一次尝试书籍翻译，由于个人外文基础不太好，中文文笔功底尚欠火候，书中难免存在错误与遗漏，欢迎大家批评指正，互相学习进步。如果本书存在错误之处或者您在书中发现有侵犯您权益的内容，请联系我进行处理。

- 昵　称：向阳
- 邮　箱：oopsguy@foxmail.com
- 微　信：oopsguy
- 博　客：http://oopsguy.com
- Github：http://github.com/oopsguy

写于 2017 年 10 月 07 日

前言

Floyd Smith

近年来，微服务在应用开发和部署方面取得了显著的进步。将应用开发或者重构成微服务以分离服务，通过 API 以明确的方式来相互“对话”。例如，每个微服务都是自包含（self-contained），各自维护自己的数据存储（这非常有意义），可以独立更新其他服务。

使用基于微服务的方式使得应用程序开发变得更快更容易管理，它只需要较少的人力就能实现更多的功能，可以更快更容易地部署。把应用程序设计成一套微服务，更加容易在多台具有负载均衡的服务器上运行，使其能够轻松应对需求高峰、由于时间推移而平稳增长的需求和由于硬件或者软件问题导致的宕机事故。

微服务的最大进步在于改变了我们的工作方式。敏捷软件开发技术、应用迁移云端、DevOps 文化、持续集成与持续部署（CI/CD）和容器应用都使用了微服务来革新应用开发与交付。

无论是作为反向代理还是高性能的 web 服务器，NGINX 软件都与微服务和上述列出的所有技术有着紧密联系。NGINX 使得基于微服务的应用更加易于开发，确保了微服务解决方案能顺利运行。

随着 NGINX 与微服务之间的关系日渐紧密，我们已经在 NGINX 网站上运行了一个由 Chris Richardson 所写的七部分系列微服务。他很早就参与了设计与实现，他的博文主要涵盖了微服务应用设计与开发方面的内容，包括了如何从单体应用迁移至微服务。博文提供了关于微服务问题的全面概述，非常受欢迎。

在本书中，我们已经将全篇博文转换成章节，并在每一章节添加了尾栏以展示 NGINX 实现微服务的相关内容。如果您认真听取建议，您将解决许多潜在的开发时甚至是在编写代码之前可能遇到的问题。此书在关于

NGINX 微服务参考架构 方面也是一本非常不错的书籍，其实现了以下提出的大部分理论。

本书章节：

- **微服务简介**
  从被夸大的微服务概念到如何在创建和维护应用时部署微服务进行简单介绍。

- **使用 API 网关**
  API 网关是整个微服务应用的单入口，它为每一个微服务提供了 API。NGINX Plus 可以很好地应用于 API 网关，提供了负载均衡和静态文件缓存等功能。

- **微服务架构中的进程间通信**
  当把一个单体应用分解成几部分（微服务），他们就需要相互通信。事实上有许多进程间通信的方案可供您选择，包括表述性状态转义（REST）。本章将给出详细介绍。

- **微服务架构中的服务发现**
  当服务运行在一个动态环境中，想要找到他们并不是一件简单的事情。

- **微服务事件驱动数据管理**
  每个微服务维护着自己特有的数据展示与存储，而不是共享一个统一、跨越一个（或两个）单体应用的数据存储。虽然这能给予您很大的灵活性，但也可能导致变得复杂。本章可以帮助您理清这些问题。

- **选择微服务部署策略**
  在 DevOps 世界中，您怎样做与您最初要做的事一样重要。Chris 讲解了微服务部署的主要模式，以便您可以为您的应用作出合理的选择，

- **重构单体应用为微服务**
  在理想世界里，我们不会缺少时间与金钱，因此可以将核心软件转化为最新最好的技术、工具和方法。而您可能会发现自己正在将一个单体应用转化为微服务，而且进展非常缓慢……。Chris 在本章将为您讲解明智的做法。

我们认为您将会发现本书的每一章都是值得阅读的，我们希望当您在开发自己的微服务应用时，能应用到本书的内容。

Floyd Smith，NGINX 公司

1 微服务简介

如今微服务倍受关注：文章、博客、社交媒体讨论和会议演讲。微服务正在迅速朝着加德纳技术成熟度曲线（Gartner Hype cycle）的高峰前进。与此同时，也有持怀疑态度的软件社区人员认为微服务没什么新鲜可言。反对者声称它的思想只是面向服务架构（SOA）的重塑。然而，无论是炒作还是怀疑，不可否认，微服务架构模式具有非常明显的优势 — 特别是在实施敏捷开发和复杂的企业应用交付方面。

本书的七个章节主要介绍如何设计、构建和部署微服务，这是本书的第一章。在此章节中，您将了解到微服务的由来和与传统单体应用模式的对比。这本电子书描述了许多关于微服务架构方面的内容。无论是在项目意义还是实施方面，您都能了解到微服务架构模式的优点与缺点。

我们先来看看为什么要考虑使用微服务。

1.1 构建单体应用

我们假设，您开始开发一个打车应用，打算与 Uber 和 Hailo 竞争。经过初步交流和需求收集，您开始手动或者使用类似 Rails、Spring Boot、Play 或者 Maven 等平台来生成一个新项目。

该新应用是一个模块化的六边形架构，如图 1-1 所示：

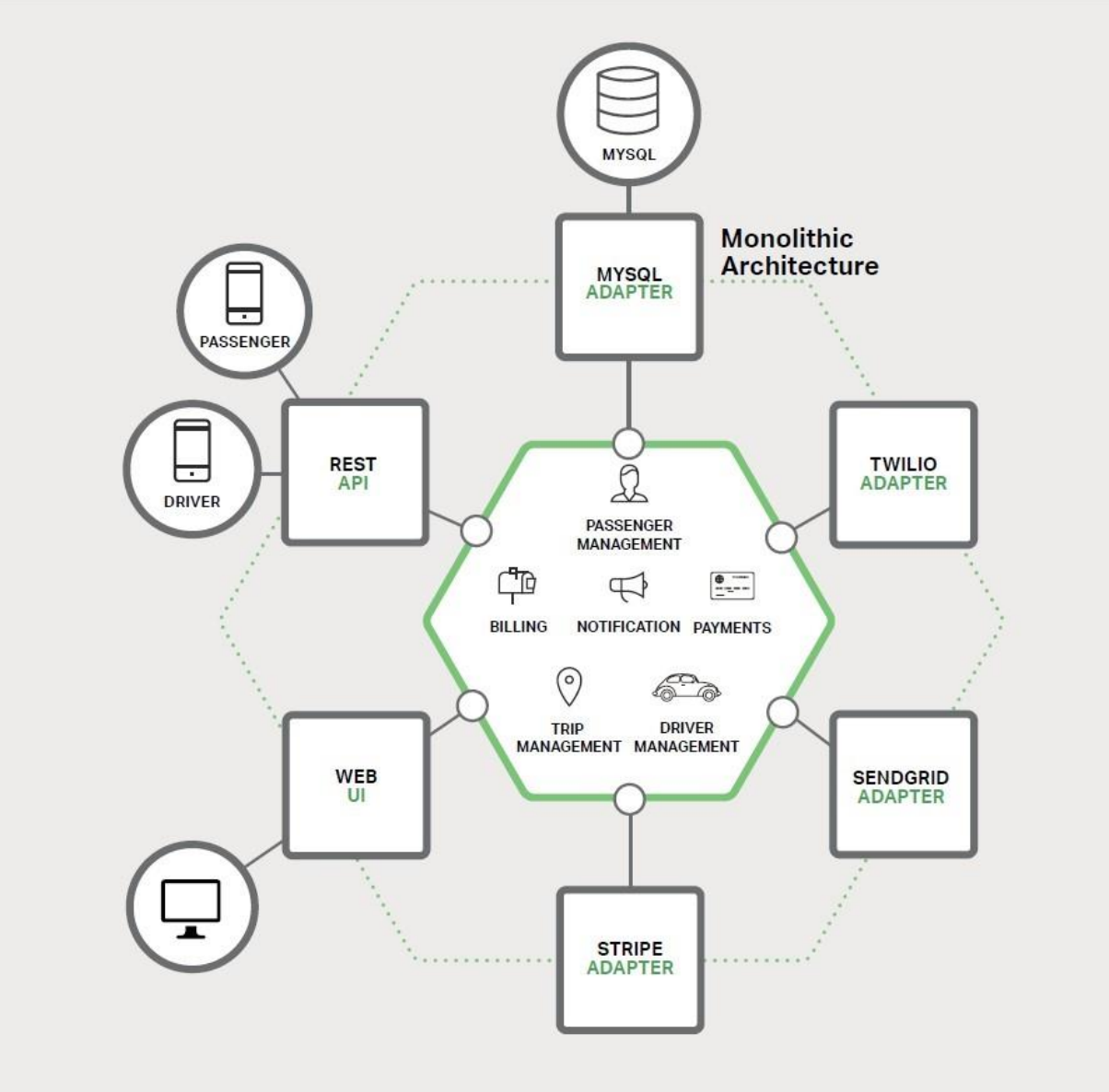


图 1-1、一个简单的打车应用

该应用的核心是由模块实现的业务逻辑，它定义了服务、领域对象和事件。围绕核心的是与外部世界接口对接的适配器。适配器示例包括数据库访问组件、生产和消费消息的消息组件和暴露了 API 或实现了一个 UI 的 web 组件。

尽管有一个逻辑模块化架构，但应用程序被作为一个单体进行打包和部署。实际格式取决于应用程序的语言和框架。例如，许多 Java 应用程序被打包成 WAR 文件部署在如 Tomcat 或者 Jetty 之类的应用服务器上。其他 Java 应用程序被打包成自包含（self-contained）的可执行 JAR。类似地，Rails 和 Node.js 应用程序被打包为有目录层次的结构。

以这种风格编写的应用是很常见的。他们很容易开发，因为我们的 IDE 和其他工具就是专注于构建单体应用。这些应用程序也很容易测试，您可以通过简单地启动并使用如 Selenium 测试包来测试 UI 以轻松地实现端到端（end-to-end）测试。单体应用

同样易于部署。您只需拷贝打包好的应用程序到服务器上。您还可以通过运行多个副本和结合负载均衡器来扩展应用。在项目的早期阶段，它可以良好运作。

1.2 走向单体地狱

不幸的是，这种简单的方法有很大的局限性。成功的应用有一个趋势，随着时间推移而变得越来越臃肿。您的开发团队在每个冲刺阶段都要实现更多的用户需求，这意味着需要添加了许多行代码。几年之后，小而简单的应用将会逐渐成长成一个庞大的单体。为了给出一个极端示例，我最近和一位开发者做了交谈，他正在编写一个工具，该工具用于从他们的数百万行代码（lines of code，LOC）应用中分析出数千个 JAR 之间的依赖。我相信这是大量开发者在多年齐心协力下创造出了这样的野兽。

一旦您的应用程序成为了一个庞大、复杂的单体，您的开发组织可能会陷入了一个痛苦的境地，敏捷开发和交付的任何一次尝试都将原地徘徊。一个主要问题是应用程序实在非常复杂。对于任何一个开发人员来说显得过于庞大，这是可以理解的。最终，正确修复 bug 和实现新功能变得非常困难而耗时。此外，这种趋势就像是往下的螺旋。如果基本代码都令人难以理解，那么改变也不会变得正确，您最终得到的将是一个巨大且不可思议的大泥球。

应用程序的规模也将减缓发展。应用程序越大，启动时间越长。我调查过开发者们的单体应用的大小和性能，一些报告的启动时间为 12 分钟。我也听说过应用程序启动需要 40 分钟以上的怪事。如果开发人员经常要重启应用服务器，那么很大一部分时间都是在等待中度过，他们的生产力将受到限制。

另一个大问题是，复杂的单体应用本身就是持续部署的障碍。如今，SaaS 应用发展到了可以每天多次将变更推送到生产环境中。这对于复杂的单体来说非常困难，因为您需要重新部署整个应用程序才能更新其中任何一部分。联想到我之前提到的漫长启动时间，这也不会是什么好事。此外，因变更所产生的影响通常不是很明确，您很可能需要做大量的手工测试。因此，持续部署是不可能做到的。

当不同模块存在资源需求冲突时，单体应用可能难以扩展。例如，一个模块可能会执行 CPU 密集型图像处理逻辑，理想情况下是部署在 Amazon EC2 Compute Optimized 实例中。另一个模块可能是一个内存数据库，最适合部署到 EC2 Memory-optimized 实例。然而，由于这些模块被部署在一起，您必须在硬件选择上做出妥协。

单体应用的另一个问题是可靠性。因为所有模块都运行在同一进程中。任何模块的一个 bug，比如内存泄漏，可能会拖垮整个进程。此外，由于应用程序的所有实例都是相同的，该错误将影响到整个应用的可用性。

最后但同样重要，单体应用使得采用新框架和语言变得非常困难。例如，我们假设您有 200 万行代码使用了 XYZ 框架编写。如果使用较新的 ABC 框架来重写整个应用，

这将非常昂贵（在时间和成本方面），即使框架非常好。因此，这对于采用新技术是一个非常大的障碍。在项目开始时，您无论选择何种新技术都会感到困扰。

总结一下：您有一个成功的关键业务应用程序，它已经发展成为一个只有少数开发人员（如果有的话）能够理解的巨大单体。它使用了过时、非生产性技术编写，这使得招聘优秀开发人员变得非常困难。应用程序变得难以扩展，不可靠。因此敏捷开发和应用交付是不可能的。

那么您能做些什么呢？

1.3 微服务 — 解决复杂问题

许多如 Amazon、eBay 和 Netflix 这样的组织，已经采用现在所谓的微服务架构模式解决了这个问题，而不是构建一个臃肿的单体应用。它的思路是将应用程序分解成一套较小的互连服务。一个服务通常实现了一组不同的特性或功能，例如订单管理、客户管理等。每一个微服务都是一个迷你应用，它自己的六边形架构包括了业务逻辑以及多个适配器。

一些微服务会暴露一个供其他微服务或应用客户端消费的 API。其他微服务可能实现了一个 web UI。在运行时，每个实例通常是一个云虚拟机（virtual machine，VM）或者一个 Docker 容器。

例如，前面描述的系统可能分解成如图 1-2 所示：

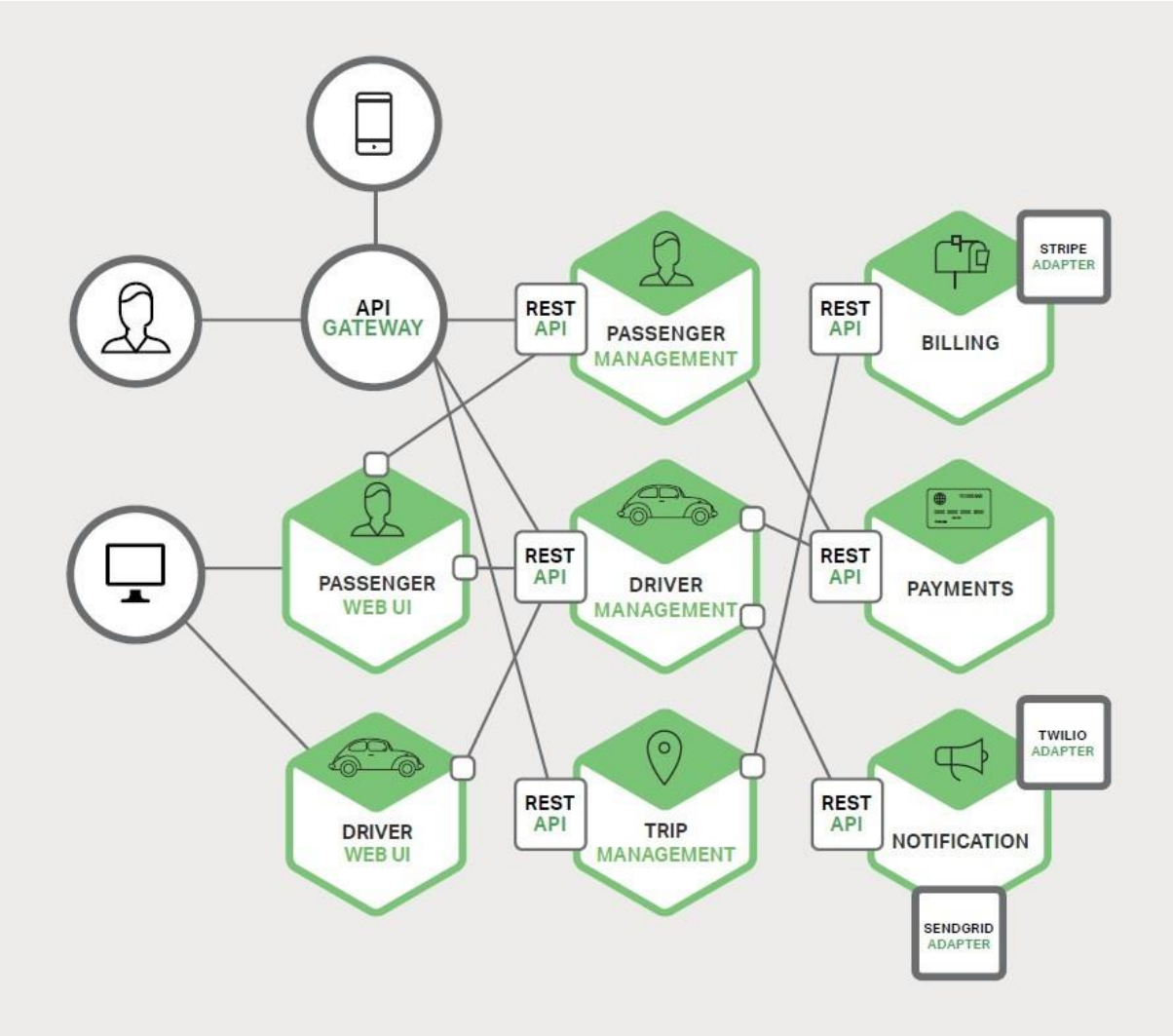


图 1-2、一个单体应用分解成微服务

应用程序的每个功能区域现在都由自己的微服务实现。此外，Web 应用程序被划分为一组更简单的 Web 应用程序。例如，以我们的出租车为例，一个是乘客的应用，一个是司机的应用。这使得它更容易地为特定的用户、司机、设备或者专门的用例部署不同的场景。每个后端服务暴露一个 REST API，大部分服务消费的 API 由其他服务提供。例如，Driver Management 使用了 Notification 服务器来通知一个可用司机一个可选路程。UI 服务调用了其他服务来渲染页面。服务也可以使用异步、基于消息的通信。本电子书后面将会更加详细介绍服务间通信。

一些 REST API 也暴露给移动端应用以供司机和乘客使用。然而，应用不能直接访问后端服务。相反，他们之间的通信是由一个称为 API 网关（API Gateway）的中介负责。API 网关负责负载均衡、缓存、访问控制、API 计量和监控，可以通过使用 NGINX 来实现。第二章将详细讨论 API 网关。

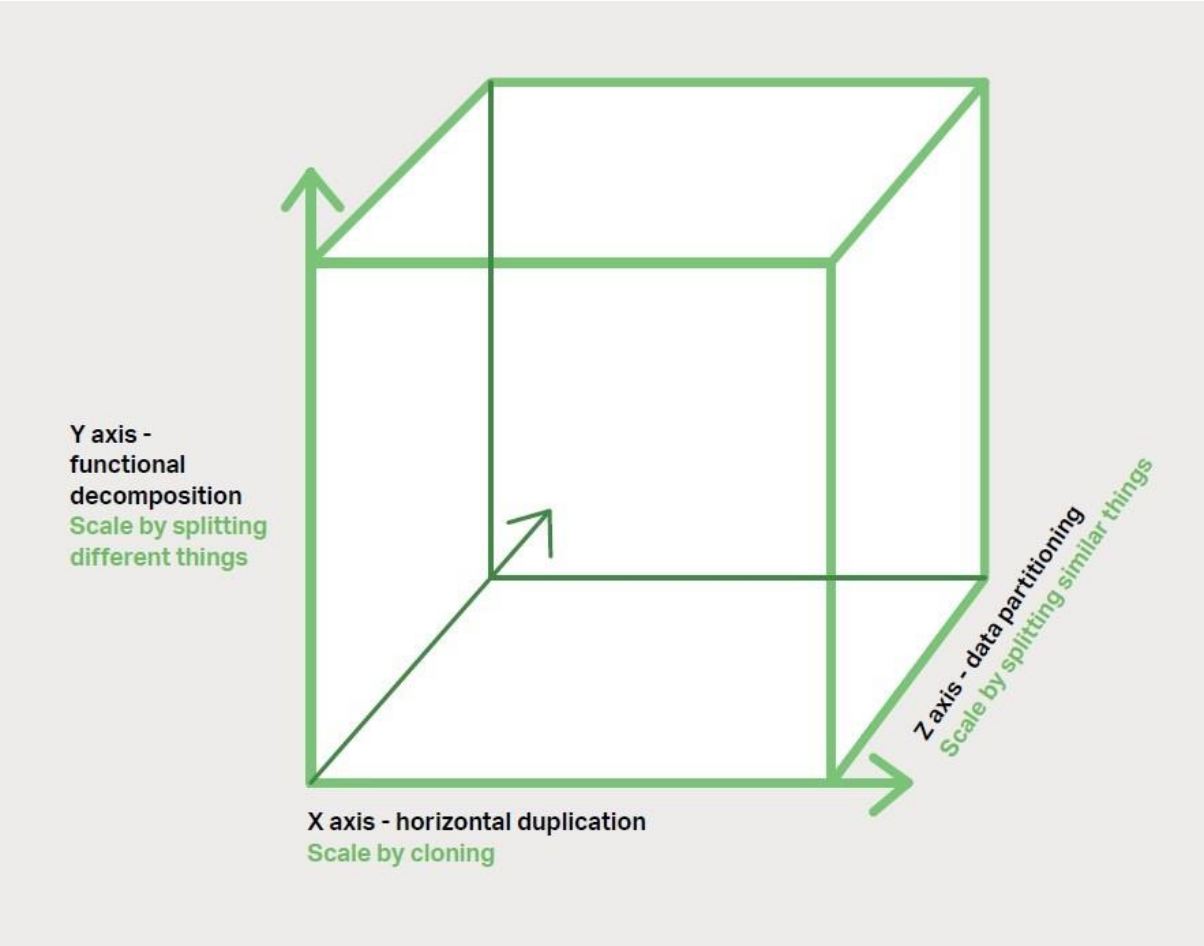


图 1-3、开发和交付中的伸缩立方（Scale Cube）

微服务架构模式相当于此伸缩立方的 Y 轴坐标，此立方是一个来自《架构即未来》的三维伸缩模型。另外两个坐标轴是由运行多个相同应用程序副本的负载均衡器组成的 X 轴坐标和 Z 轴坐标（或数据分区），其中请求的属性（例如，一行记录的主键或者客户标识）用于将请求路由到特定的服务器。

应用程序通常将这三种类型的坐标方式结合一起使用。Y 轴坐标将应用分解成微服务，如图 1-2 所示。

在运行时，X 坐标轴上运行着服务的多个实例，每个服务配合负载均衡器以满足吞吐量和可用性。某些应用程序也有可能使用 Z 坐标轴来进行分区服务。图 1-4 展示了如何用 Docker 将 Trip Management 服务部署到 Amazon EC2 上运行。

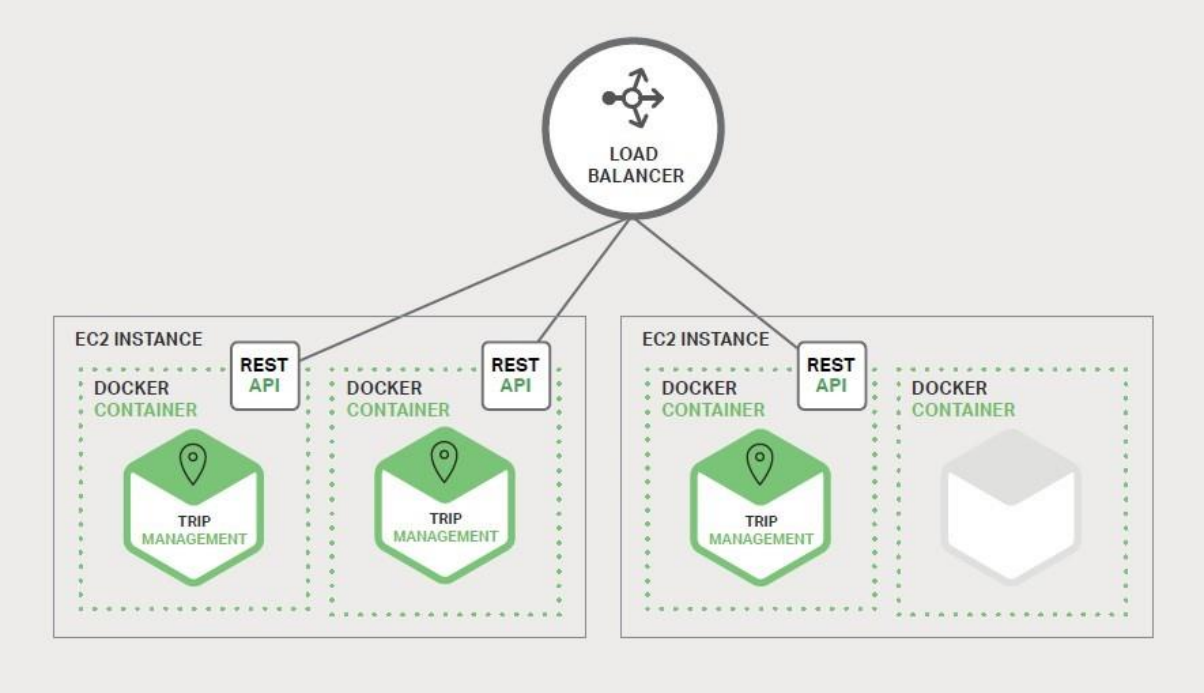


图 1-4、使用 Docker 部署 Trip Management 服务

在运行时，Trip Management 服务由多个服务实例组成，每个服务实例是一个 Docker 容器。为了实现高可用，容器是在多个云虚拟机上运行的。服务实例的之前是一个类似 NGINX 的负载均衡器，用于跨实例分发请求。负载均衡器也可以处理其他问题，如缓存、访问控制、API 度量和监控。

微服务架构模式明显影响到了应用程序与数据库之间的关系。与其他共享单个数据库模式（schema）服务有所不同，其每一个服务都有自己的数据库模式。一方面，这种做法与企业级数据库数据模型的想法相背，此外，它经常导致部分数据冗余。然而，如果您想从微服务中受益，每一个服务都应该有自己的数据库模式。因为它能实现松耦合。图 1-5 展示了数据库架构示例应用程序。

每个服务都拥有各自的数据库。而且，服务可以使用一种最适合其需求、号称多语言持久架构（polyglot persistence architecture）的数据库。例如，Driver Management，要找到与潜在乘客接近的司机，就必须使用支持高效地理查询的数据库。

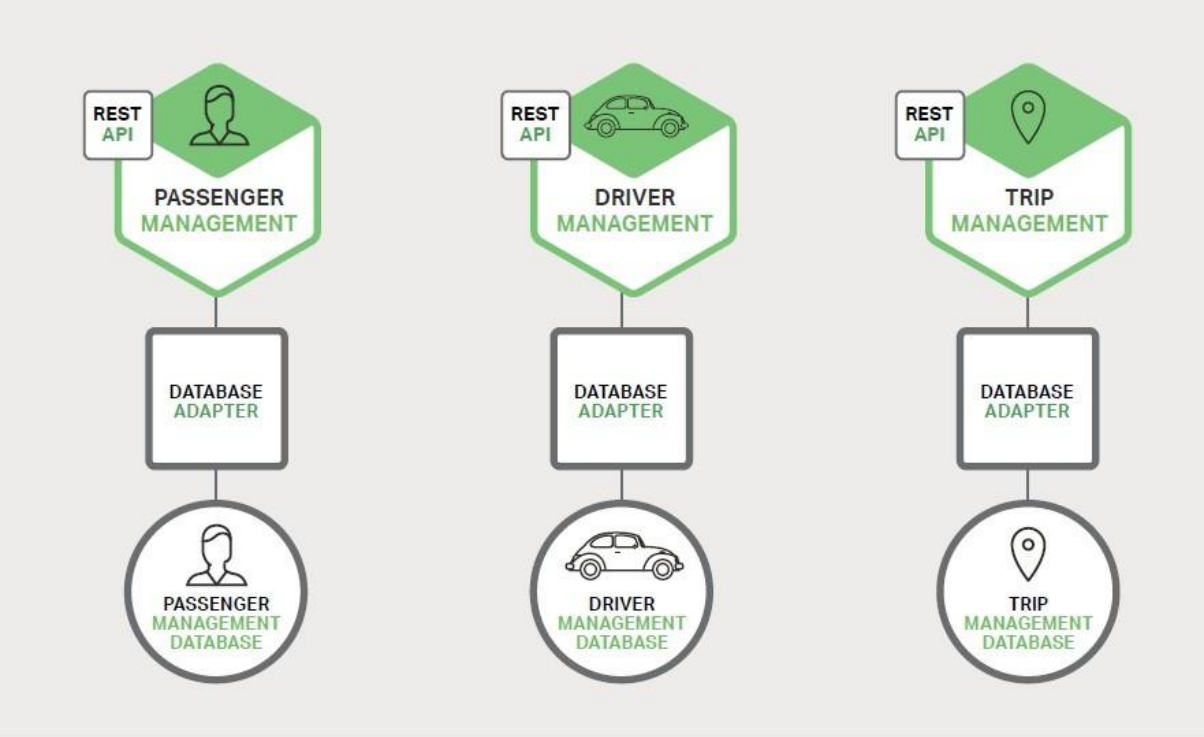


图 1-5、打车应用的数据库架构

从表面上看，微服务架构模式类似于 SOA。微服务是由一组服务组成。然而，换另一种方式去思考微服务架构模式，它是没有商业化的 SOA，没有集成 Web 服务规范（WS-*）和企业服务总线（Enterprise Service Bus，ESB）。基于微服务的应用支持更简单、轻量级的协议，例如，REST，而不是 WS-*。他们也尽量避免使用 ESB，而是实现微服务本身具有类似 ESB 的功能。微服务架构也拒绝了 SOA 的其他部分，例如，数据访问规范模式概念。

1.4 微服务的优点

微服务架构模式有许多非常好的地方。第一，它解决了复杂问题。它把可能会变得庞大的单体应用程序分解成一套服务。虽然功能数量不变，但是应用程序已经被分解成可管理的块或者服务。每个服务都有一个明确定义边界的方式，如远程过程调用（RPC）驱动或者消息驱动的 API。微服务架构模式强制一定程度的模块化，实际上，使用单体代码来实现是极其困难的。因此，使用微服务架构模式，个体服务能被更快地开发，并更容易理解与维护。

第二，这种架构使得每个服务都可以由一个团队独立专注开发。开发者可以自由选择任何符合服务 API 契约的技术。当然，更多的组织是希望通过技术选型限制来避免完全混乱的状态。然而，这种自由意味着开发人员不再有可能在这种自由的新项目开始时使用过时的技术。当编写一个新服务时，他们可以选择当前的技术。此外，由于服务较小，使用当前技术重写旧服务将变得更加可行。

第三，微服务架构模式可以实现每一个微服务独立部署。开发人员根本不需要去协调部署本地变更到服务。这些变更一经测试即可立即部署。比如，UI 团队可以执行 A|B 测试，并快速迭代 UI 变更。微服务架构模式使得持续部署成为可能。最后，微服务架构模式使得每个服务能够独立扩展。您可以仅部署满足每个服务的容量和可用性约束的实例数目。此外，您可以使用与服务资源要求最匹配的硬件。例如，您可以在 EC2 Compute Optimized 实例上部署一个 CPU 密集型图像处理服务，并且在 EC2 Memory-optimized 实例上部署一个内存数据库服务。

1.5 微服务的缺点

就像 Fred Brooks 大约在 30 年前写的《人月神话》中说的，**没有银弹**。与其他技术一样，微服务架构模式也存在着缺点。其中一个缺点就是名称本身。微服务这个术语的重点过多偏向于服务的规模。事实上，有些开发者主张构建极细粒度的 10 至 100 LOC（代码行） 服务，虽然对于小型服务可能比较好，但重要的是要记住，小型服务只是一种手段，而不是主要目标。微服务的目标在于充分分解应用程序以方便应用敏捷开发和部署。

微服务另一个主要的缺点是由于微服务是一个分布式系统，其使得整体变得复杂。开发者需要选择和实现基于消息或者 RPC 的进程间通信机制。此外，由于目标请求可能很慢或者不可用，他们必须要编写代码来处理局部故障。虽然这些都不是很复杂、高深，但模块间通过语言级方法/过程调用相互调用，这比单体应用要复杂得多。

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是相当普遍的。这些事务在单体应用中的实现显得微不足道，因为单体只存在一个单独的数据库。在基于微服务的应用程序中，您需要更新不同服务所用的数据库。通常不会选择分布式事务，不仅仅是因为 CAP 定理。他们根本不支持如今高度可扩展的 NoSQL 数据库和消息代理。您最后不得不使用基于最终一致性的方法，这对于开发人员来说更具挑战性。

测试微服务应用程序也很复杂。例如，使用现代框架如 Spring Boot，只需要编写一个测试类来启动一个单体 web 应用程序并测试其 REST API。相比之下，一个类似的测试类对于微服务来说需要启动该服务及其所依赖的所有服务，或者至少为这些服务配置存根。再次声明，虽然这不是一件高深的事情，但不要低估了这样做的复杂性。

微服务架构模式的另一个主要挑战是实现了跨越多服务变更。例如，我们假设您正在实现一个变更服务 A、服务 B 和 服务 C 的需求，其中 A 依赖于 B，且 B 依赖于 C。在单体应用程序中，您可以简单地修改相应的模块、整合变更并一次性部署他们。相反，在微服务中您需要仔细规划和协调出现的变更至每个服务。例如，您需要更新服务 C，然后更新服务 B，最后更新服务 A。幸运的是，大多数变更只会影响一个服务；需要协调的多服务变更相对较少。

部署基于微服务的应用程序也是非常复杂的。一个单体应用可以很容易地部署到基于传统负载均衡器的一组相同服务器上。每个应用程序实例都配置有基础设施服务的位

置（主机和端口），比如数据库和消息代理。相比之下，微服务应用程序通常由大量的服务组成。例如，据 Adrian Cockcroft 了解到，Hailo 拥有 160 个不同的服务，Netflix 拥有的服务超过 600 个。

每个服务都有多个运行时实例。还有更多的移动部件需要配置、部署、扩展和监控。此外，您还需要实现服务发现机制，使得服务能够发现需要与之通信的任何其他服务的位置（主机和端口）。比较传统麻烦的基于票据（ticket-based）和手动的操作方式无法扩展到如此复杂程度。因此，要成功部署微服务应用程序，需要求开发人员能高度控制部署方式和高度自动化。

一种自动化方式是使用现成的平台即服务（PaaS），如 Cloud Foundry。PaaS 为开发人员提供了一种简单的方式来部署和管理他们的微服务。它让开发人员避开了诸如采购和配置 IT 资源等烦恼。同时，配置 PaaS 的系统人员与网络专业人员可以确保达到最佳实践以落实公司策略。自动化微服务部署的另一个方式是开发自己的 PaaS。一个普遍的起点是使用集群方案，如 Kubernetes，与 Docker 等容器技术相结合。在本书最后我们将看到如 NGINX 的基于软件的应用交付方式是如何在微服务级别处理缓存、访问控制、API 计量和监控，这些可以帮助解决此问题。

1.6 总结

构建复杂的微服务应用程序本质上是困难的。单体架构模式只适用于简单、轻量级的应用程序，如果您使用它来构建复杂应用，您最终会陷入痛苦的境地。微服务架构模式是复杂、持续发展应用的一个更好的选择。尽管它存在着缺点和实现挑战。

在后面的章节中，我将介绍微服务架构的方方面面并探讨诸如服务发现、服务部署方案以及将单体应用重构为服务的策略。

微服务实战：NGINX Plus 作为反向代理服务器

By Floyd Smith

10000 个网站中有超过 50% 使用 NGINX，这主要是因为它具有作为反向代理服务器的能力。您可以把

NGINX 放在当前应用程序甚至是数据库服务器之前以获取各种功能 — 更高的性能、更高的安全性、可扩展性、灵活性等。您现有的应用程序只需要配置代码和作出很少或无需改变。然而，对于存在性能压力的站点，或者预计未来存在高负荷，使用 NGINX 的效果看起来似乎没那么神奇。

那么这与微服务有什么关系呢？实现一个反向代理服务器，并使用 NGINX 的其他功能来为您提供架构灵活性。反向代理服务器、静态和应用文件缓存、SSL/TLS 和 HTTP/2

都会从您的应用程序剔除。让应用程序只做它该做的事，NGINX 还可作为负载均衡器，这是微服务实施过程中的一个关键角色。先进的 NGINX Plus 的功能包含了复杂的负载均衡算法、多种方式的会话持久和管理监控，这些对微服务尤其有用（NGINX 最近还增加了使用 DNS SRV 记录的服务发现支持，这是一个顶尖的功能）。而且，如本章所述，NGINX 可以自动化部署微服务。

此外，NGINX 还提供了必要的功能来支撑 NGINX 微服务参考架构中的三大模型。代理模型使用 NGINX 作为 API 网关；网格路由模型使用一个额外的 NGINX 作为进程间通信中枢；Fabric 模型中的每个微服务使用一个 NGINX 来控制 HTTP 流量，在微服务之间实现 SSL/TLS，这非常具有突破性。

2 使用 API 网关

本书的七个章节是关于如何设计、构建和部署微服务。第一章介绍了微服务架构模式。它阐述了使用微服务的优点与缺点，以及尽管如此，微服务通常是复杂应用的理想选择。该系列的第二章将探讨使用 API 网关构建微服务。

当您选择将应用程序构建成为一组微服务时，您需要决定应用程序客户端将如何与微服务进行交互。单体应用程序只有一组端点（endpoint），通常使用复制（replicated）结合负载均衡来分配流量。

然而，在微服务架构中，每个微服务都暴露一组通常比较细颗粒的端点。在本文中，我们将研究如何改进客户端通信，并提出一个使用 API 网关的方案。

2.1 简介

我们假设您正在为一个购物应用开发一个原生移动客户端。您可能需要实现一个产品详细信息页面，用于展示给定商品的信息。

例如，图 2-1 展示了在 Amazon 的 Android 移动应用中滚动产品信息时所看的内容。

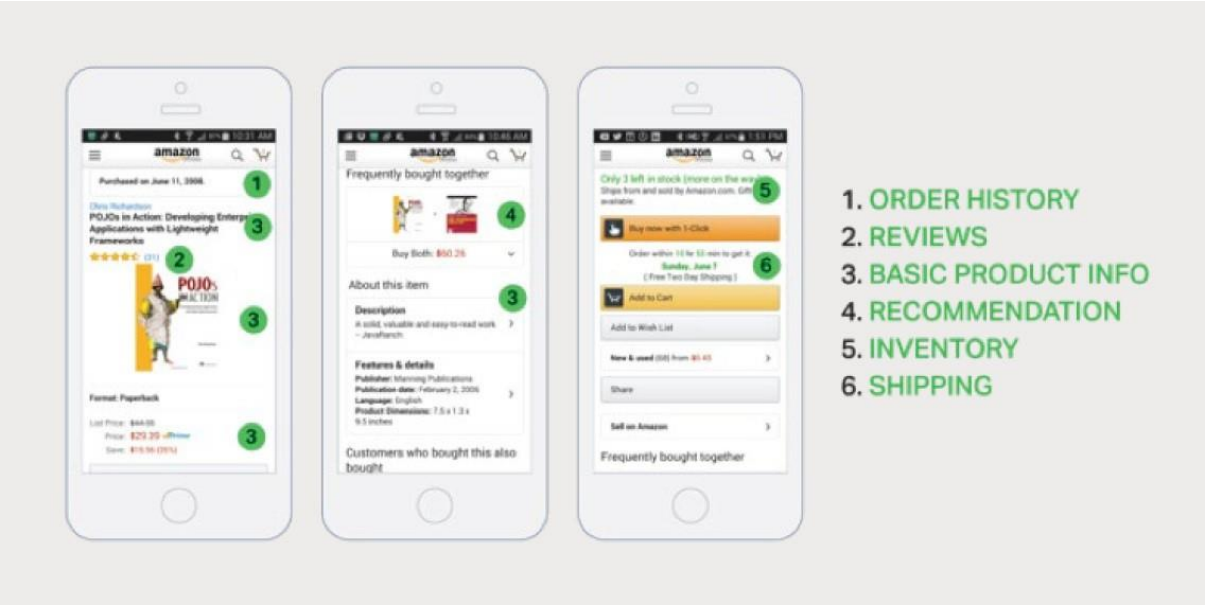


图 2-1、一个简单的购物应用

这是一个智能手机应用，产品详细信息页面展示了许多信息。不仅有基本的产品信息，如名称、描述和价格，页面还展示了：

1. 购物车中的物品数量
2. 订单历史
3. 客户评价
4. 低库存警告
5. 配送选项
6. 各种推荐，包括了购买此产品的客户购买的其他产品
7. 选择性购买选项

在使用单体应用架构的情况下，移动客户端通过对应用程序进行单个 REST 调用来检索此数据，例如：

```
GET api.company.com/productdetails/productId
```

负载均衡器将请求路由到几个相同应用程序实例中的其中一个。之后，应用程序查询各个数据库表并返回响应给客户端。相比之下，当使用微服务架构时，产品详细页面上展示的数据来自多个微服务。以下是一些微服务，可能拥有特定产品页面展示的数据：

- **订单服务** 订单历史
- **目录（catalog）服务** 基本的产品信息，如产品名称、图片和价格
- **评价服务** 客户评价
- **库存服务** 低库存警告
- **配送服务** 配送选项、期限和费用，由配送方的 API 单独提供
- **推荐服务** 推荐类目

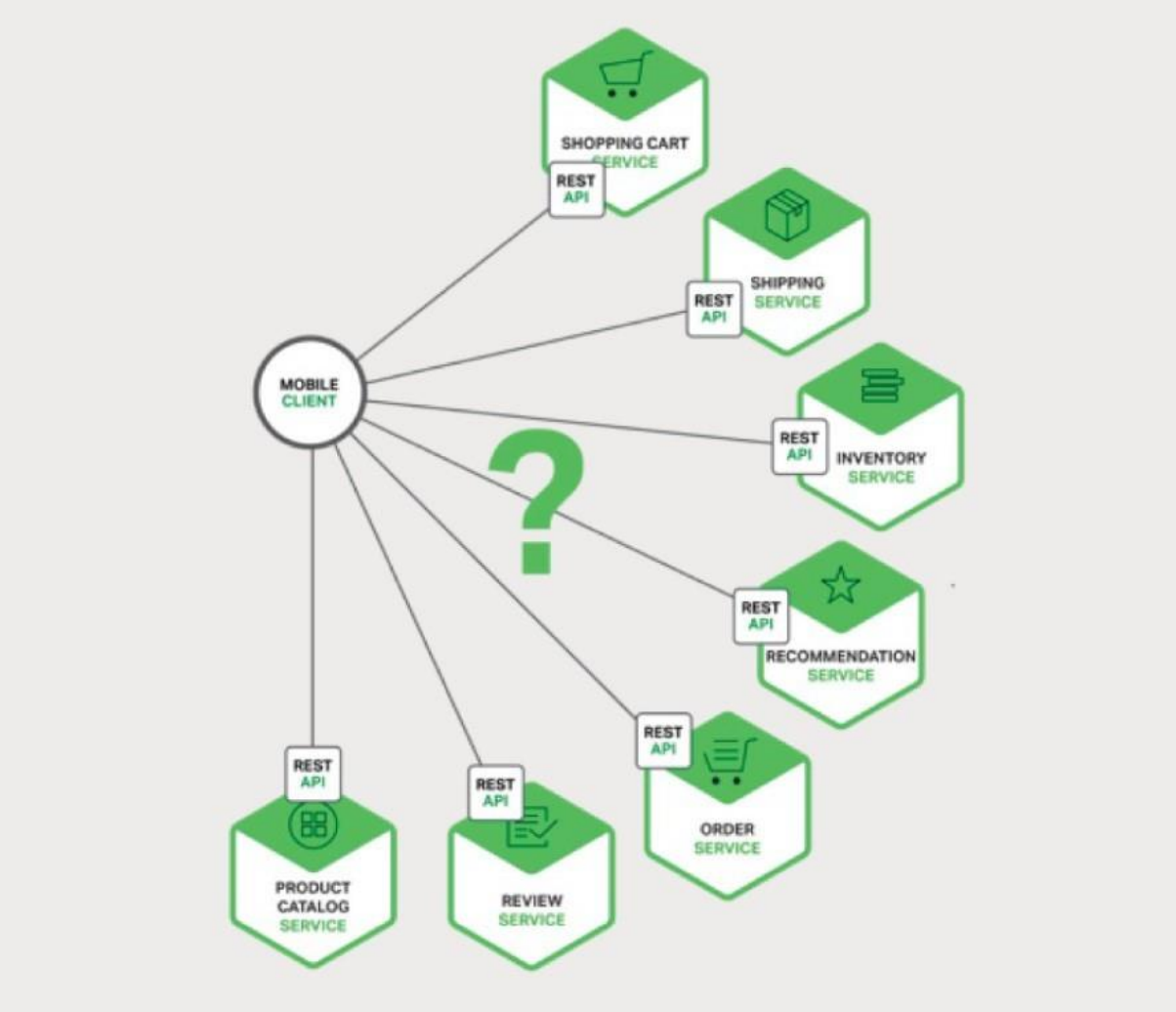


图 2-2、将移动客户端的需求映射到相关微服务

我们需要决定移动客户端如何访问这些服务。让我们来看看有哪些方法。

2.2 客户端与微服务直接通信

理论上，客户端可以直接向每个微服务发送请求。每个微服务都有一个公开的端点：

`https://serviceName.api.company.name`

该 URL 将映射到用于跨可用实例分发请求的微服务负载均衡器。为了检索特定的产品页面信息，移动客户端将向上述的每个微服务发送请求。

不幸的是，这种方式存在着挑战与限制。第一个问题是客户端的需求与每个微服务暴露的细粒度的 API 不匹配。在此示例中，客户端需要进行七次单独请求。如果在更加复杂的应用中，它可能需要做更多的工作。例如，Amazon 展示了在产品页面渲染中如何牵涉到数百个微服务。虽然客户端可以通过 LAN 发送许多请求，但在公共互联网下效率低下，在移动网络必然是不切实际。

客户端直接调用微服务存在的另一个问题是有些可能使用了非 web 友好协议。一个服务可能使用了 Thrift 二进制 RPC，而另一个则可能使用 AMQP 消息协议。这两个协议无论是对浏览器还是防火墙而言都是不友好的，最好是在内部使用。应用程序在防火墙之外应该使用 HTTP 或者 WebSocket 之类的协议。

这种方法的另一个缺点是它难以重构微服务。随着时间推移，我们可能会想改变系统划分服务。例如，我们可能会合并两个服务或者将服务拆分为两个或者多个。然而，如果客户端直接与服务进行通信，实施这类的重构将变得非常困难。

由于存在这些问题，很少有客户端直接与微服务进行通信。

2.3 使用 API 网关

通常更好的方法是使用 API 网关。API 网关是一个服务器，是系统的单入口点。它类似于面向对象设计模式中的门面（Facade）模式。API 网关封装了内部系统架构，并针对每个客户端提供一个定制 API。它还可用于认证、监控、负载均衡、缓存和静态响应处理。

图 2-3 展示了 API 通常如何整合架构

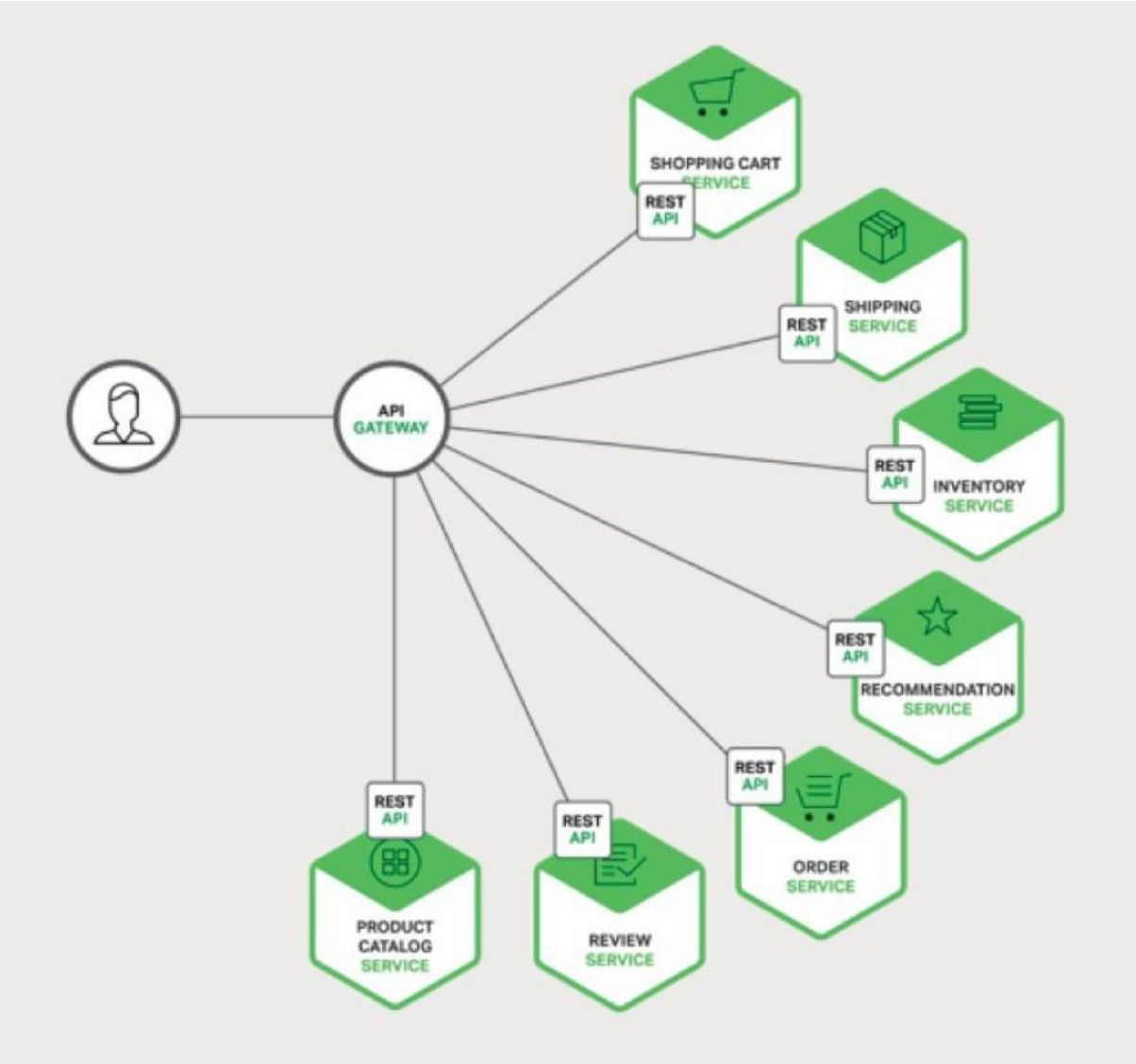


图 2-3、使用 API 网关的微服务

API 网关负责请求路由、组合和协议转换。所有的客户端请求首先要通过 API 网关，之后请求被路由到适当的服务。API 网关通常会通过调用多个微服务和聚合结果来处理一个请求。它可以在 Web 协议（如 HTTP 和 WebSocket）和用于内部的非 Web 友好协议之间进行转换。

API 还可以为每个客户端提供一个定制 API。它通常会为移动客户端暴露一个粗粒度的 API。例如，考虑一下产品详细信息场景。API 网关可以提供一个端点 **/productdetails?productid=xxx**，如图 2-3 所示，一个使用了 API 网关的微服务。允许移动客户端通过一个单独的请求来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、推荐、评价等）并组合结果。

一个很好的 API 网关案例是 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、机顶盒、智能手机、游戏机和平板电脑等。起初，Netflix 尝试为他们的流媒体服务提供一个通用的 API。后来，他们发现由于设备种

类繁多，并且他们各自有着不同需求，所以并不是能很好地运作。如今，他们使用了 API 网关，通过运行特定设备适配代码来为每个设备提供一个定制 API。

2.4 API 网关的优点与缺点

正如您所料，使用 API 网关同样存在好处与坏处。使用 API 网关的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信，而不必调用特定的服务。API 网关为每种类型的客户端提供了特定的 API，减少了客户端与应用程序之间的往返次数。它还简化了客户端代码。

API 网关也存在一些缺点，它是另一个高度可用的组件，需要开发、部署和管理。还另外，还有一个风险是 API 网关可能会成为开发瓶颈。开发人员必须更新 API 网关以暴露每个微服务的端点。

重要的是更新 API 网关的过程应尽可能地放缓一些。否则，开发人员将被迫排队等待网关更新。尽管 API 网关存在这些缺点，但对于大多数的真实应用来说，使用 API 是合理的。

2.5 实施 API 网关

我们已经了解了使用 API 网关的动机与权衡。接下来让我们看看您需要考虑的各种设计问题。

2.5.1 性能与可扩展性

只有少数公司能达到 Netflix 的运营规模，每天需要处理数十亿的请求。然而，对于大多数应用来说，API 网关的性能和可扩展性是相当重要的。因此，在一个支持异步、非阻塞 I/O 平台上构建 API 网关是很有必要的。可以使用不同的技术来实现一个可扩展的 API 网关。在 JVM 上，您可以使用基于 NIO 的框架，如 Netty、Vertx、Spring Reactor 或者 JBoss Undertow。一个流行的非 JVM 选择是使用 Node.js，它是一个建立在 Chrome 的 JavaScript 引擎之上的平台。另一个选择是使用 NGINX Plus。

NGINX Plus 提供了一个成熟、可扩展和高性能的 Web 服务器和反向代理，它易于部署、配置和编程。NGINX Plus 可以管理身份验证、访问控制、负载均衡请求、缓存响应，并且提供了应用程序健康检查和监控功能。

2.5.2 使用响应式编程模型

API 网关通过简单地把他们（请求）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，如产品详细信息请求，对后端服务请求而言是彼此独立的。为了缩短响应时间到最小，API 网关应该并发执行独立请求。

然而，有时候，请求是相互依赖的。首先，API 网关可能需要在将请求路由到后端服务之前，通过调用验证服务来验证请求。同样，为了从客户的愿望清单中获取产品的信息，API 网关首先必须检索包含该信息的客户资料，然后检索每个产品的信息。另一个有趣的 API 组合案例是 Netflix 视频网格。

使用传统的异步回调方式来编写 API 组合代码会很快使您陷入回调地狱。代码将会变得杂乱、难以理解并且容易出错。一个更好的方式是使用响应式方法以声明式编写 API 网关代码。响应式抽象的例子包括 Scala 的 Future、Java 8 中的 CompletableFuture 和 JavaScript 中的 Promise。还有 Reactive Extensions（也称为 Rx 或 ReactiveX），最初由 Microsoft 为 .NET 平台开发。Netflix 为 JVM 创建了 RxJava，专门应用于其 API 网关。还有用于 JavaScript 的 RxJS，它可以在浏览器和 Node.js 中运行。使用响应式方式可让您能够编写出简单而高效的 API 网关代码。

2.5.3 服务调用

一个基于微服务的应用程序是一个分布式系统，必须使用一个进程间（inter-process）通信机制。有两种进程间通信方案。一是使用基于消息的异步机制。某些实现采用了消息代理，如 JMS 和 AMQP。其他采用无代理的方式直接与服务通信，如 Zeromq。

另一种类型的进程间通信采用了同步机制，如 HTTP 和 Thrift。系统通常会同时使用异步和同步方式。甚至可以为每种方式应用多个实现。因此，API 网关需要支持各种通信机制。

2.5.4 服务发现

API 网关需要知道与其通信的每个微服务的位置（IP 地址和端口）。在传统应用程序中，您可以将这些位置硬编码，但在现代基于云的微服务应用程序中，找到所需的位置不是一件简单的事情。

基础设施服务（比如消息代理）通常都有一个可以通过系统环境变量来指定的静态位置。但是，要确定应用程序服务的位置并不是那么容易。

应用服务可以动态分配位置。此外，由于自动扩缩和升级，一个服务的整组实例可以动态变更。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：服务端发现或客户端发现。第四章中更详细地描述了服务发现。现在需要注意的是，如果系统使用客户端发现，API 网关必须能够查询服务注册中心，该注册中心是所有微服务实例及其位置的数据库。

2.5.5 处理局部故障

实施 API 网关时必须解决的另一个问题是局部故障问题。当一个服务调用另一个响应缓慢或者不可用的服务时，所有分布式系统都会出现此问题。API 网关不应该无期限地等待下游服务。但是，如何处理故障问题取决于特定的方案和哪些服务发生故障。例如，如果推荐服务在获取产品详细信息时没有响应，API 网关应将其余的产品详细信息返回给客户端，因为它们对用户仍然有用。建议可以是空的，也可以用其他代替，例如硬编码的十强名单。然而，如果产品信息服务没有响应，那么 API 网关应该向客户端返回错误。

如果可以，API 网关还可以返回缓存数据。例如，由于产品价格变化不大，当价格服务不可用时，API 网关可以返回被缓存的价格数据。数据可以由 API 网关缓存或存储在外部缓存中，如 Redis 或者 Memcached。API 网关通过返回默认数据或缓存数据，确保系统发生故障时最小程度上影响到用户体验。

Netflix Hystrix 是一个非常有用的库，用于编写调用远程服务代码。Hystrix 可以使超出指定阈值的调用超时。它实现了断路器模式，防止客户端不必要地等待无响应的服务。如果服务的错误率超过指定阈值，Hystrix 将会跳闸，所有请求将在指定的时间内立即失败。Hystrix 允许您在请求失败时定义回退操作，例如从缓存读取或返回默认值。如果您正在使用 JVM，那么您一定要考虑使用 Hystrix。如果您是在非 JVM 环境中运行，则应使用同等作用的库。

2.6 总结

对于大多数基于微服务的应用程序来说，实现一个 API 网关是很有意义的，API 网关充当着系统的单入口点，并且负责请求路由，组合和协议转换。它为每个应用程序客户端提供了一个自定义 API。API 网关还可以通过返回缓存或默认数据来掩盖后端服务故障。在下一章中，我们将介绍服务间的通信。

微服务实战：NGINX Plus 作为 API 网关

by Floyd Smith

本章讨论了如何将 API 网关作为系统的单入口点。它可以处理诸如负载均衡、缓存、监控和协议转换等其他功能 — 当 NGINX 充当反向代理服务器时，其可以作为系统的

单入口点，并且支持所有提到的一个 API 网关具有的附加功能。因此使用 NGINX 作为 API 网关的主机可以很好地工作。

将 NGINX 作为 API 网关并不是本书最开始的想法。NGINX Plus 是一个用于管理和保护基于 HTTP 的 API 流量的领先平台。您可以实现自己的 API 网关或使用现有的 API 管理平台，其中许多使用了 NGINX。

使用 NGINX Plus 作为 API 网关的理由括：

- **访问管理**
  上至典型的 Web 应用级别，下至每个个体微服务级别，您都可以使用各种访问控制列表（ACL）方法，并且可以轻松实现 SSL/TLS。

- **可管理性与弹性**
  您可以使用 NGINX 的动态重新配置 API、Lua 模块、Perl 来更新基于 NGINX Plus 的 API 服务器，也可以通过 Chef、Puppet、ZooKeeper 或 DNS 来改变。

- **与第三方工具集成**
  NGINX Plus 已经可以与某些先进的工具集成在一起，如 3scale，Kong 和 MuleSoft 集成平台（仅列举在 NGINX 网站上提及的工具）。

NGINX Plus 被广泛用作 NGINX 微服务参考架构中的 API 网关。您可以利用在这里收集的文章以及 MRA（微服务参考架构）来了解如何在您自己的应用程序中实现这一点。

3 进程间通信

本书主要介绍如何使用微服务架构构建应用程序，这是本书的第三章。第一章介绍了微服务架构模式，将其与单体架构模式进行对比，并讨论了使用微服务的优点与缺点。第二章描述了应用程序客户端通过扮演中间人角色的 API 网关与微服务进行通信。在本章中，我们来了解一下系统中的服务是如何相互通信的。第四章将详细探讨服务发现方面的内容。

3.1 简介

在单体应用程序中，组件可通过语言级方法或者函数相互调用。相比之下，基于微服务的应用程序是一个运行在多台机器上的分布式系统。通常，每个服务实例都是一个进程。

因此，如图 3-1 所示，服务必须使用进程间通信（IPC）机制进行交互。

稍后我们将了解到多种 IPC 技术，但在此之前，我们先来探讨一下涉及到的各种设计问题。

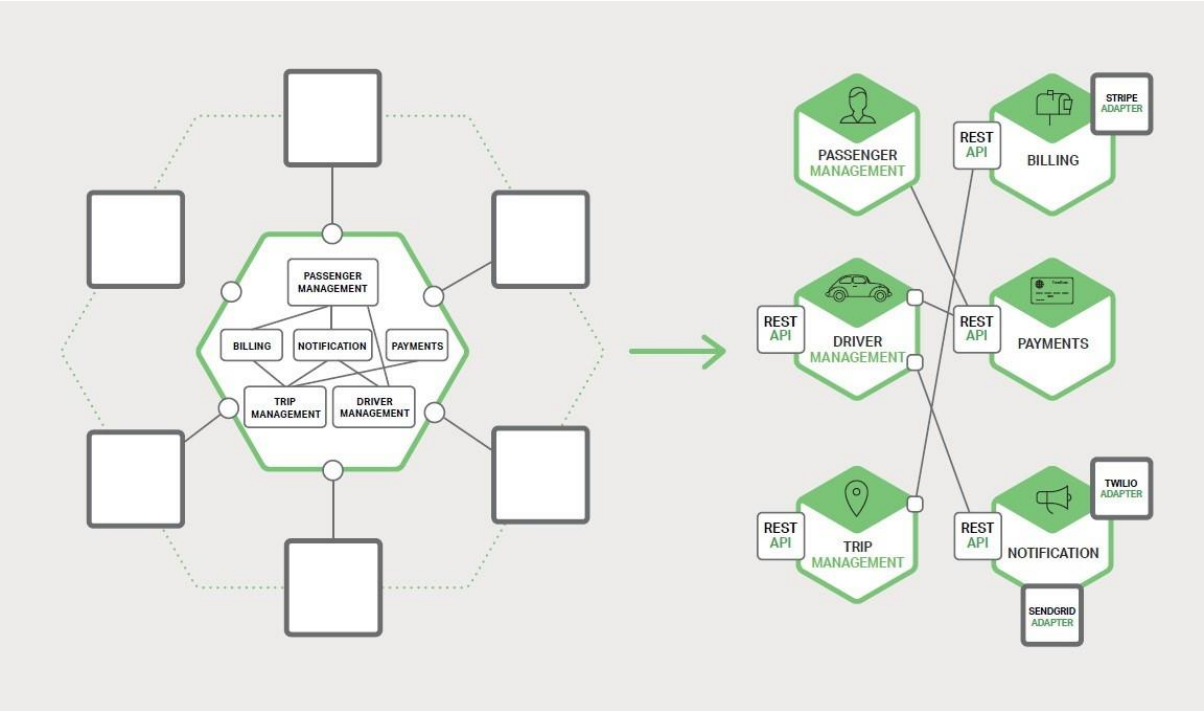


图 3-1、使用进程间通信交互的微服务

3.2 交互方式

当为服务选择一种 IPC 机制时，首先需要考虑服务如何交互。有许多种客户端 — 服务交互方式。它们可以分为两个类。第一类是一对一交互与一对多交互：

- **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。

第二类是同步交互与异步交互：

- **同步** 客户端要求服务及时响应，在等待过程中可能会发生阻塞。
- **异步** 客户端在等待响应时不会发生阻塞，但响应（如果有）不一定立即返回。

下表展示了各种交互方式。

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

表 3-1、进程间通信方式

一对一交互分为以下列举的类型，包括同步（请求/响应）与异步（通知与请求/异步响应）：

- **请求/响应**
  客户端向服务发出请求并等待响应。客户端要求响应及时到达。在基于线程的应用程序中，发出请求的线程可能在等待时发生阻塞。
- **通知（又称为单向请求）**
  客户端向服务发送请求，但不要求响应。
- **请求/异步响应**
  客户端向服务发送请求，服务异步响应。客户端在等待时不发生阻止，适用于假设响应可能不会立即到达的场景。

一对多交互可分为以下列举的类型，它们都是异步的：

- **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消费。
- **发布/异步响应**
  客户端发布请求消息，之后等待一定时间来接收消费者的响应。

通常，每个服务都组合着使用这些交互方式。对于一些服务而言，单一的 IPC 机制就足够了，但其他服务可能需要组合多个 IPC 机制。

图 3-2 显示了当用户请求打车时，打车应用中的服务可能会发生交互。

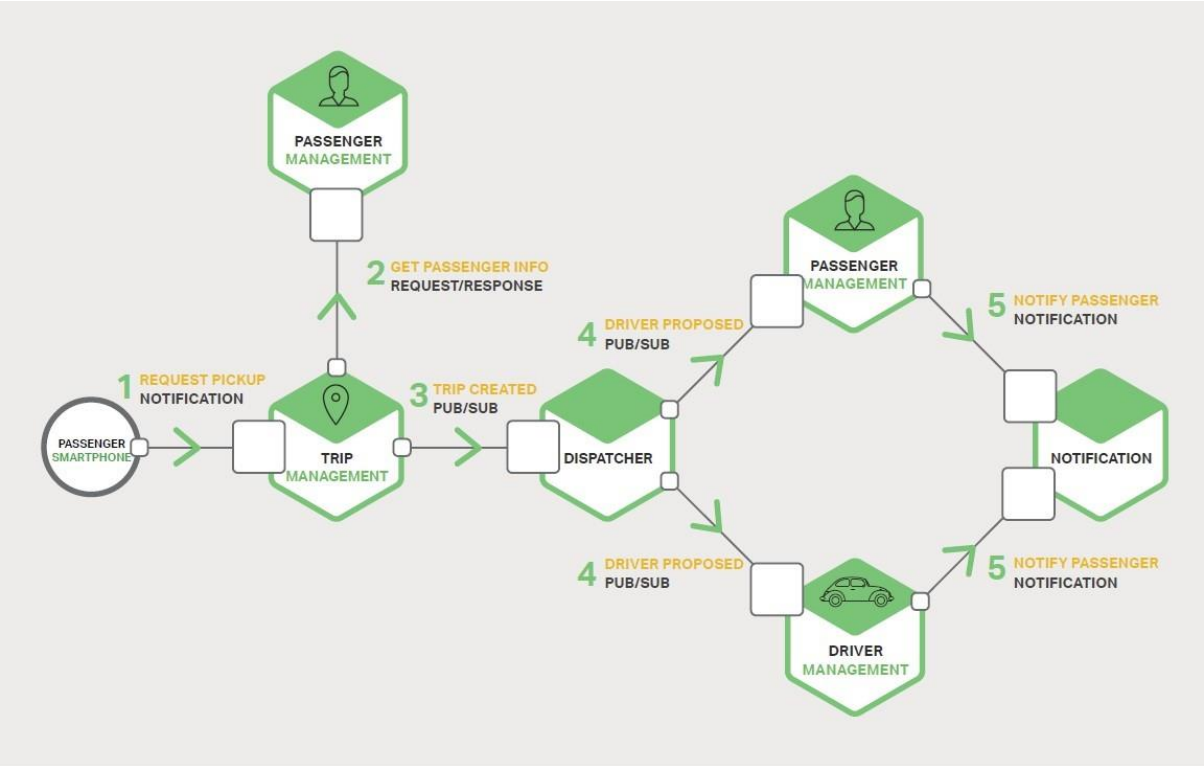


图 3-2、使用了多种 IPC 机制的服务交互

服务使用了通知、请求/响应和发布/订阅组合。例如，乘客的智能手机向 Trip Management 微服务发送一条通知以请求一辆车。Trip Management 服务通过使用请求/响应来调用 Passenger Management 服务以验证乘客的帐户是否可用。之后，Trip Management 服务创建路线，并使用发布/订阅通知其他服务，包括用于定位可用司机的 Dispatcher。

现在我们来看一下交互方式，我们先来看看如何定义 API。

3.3 定义 API

服务 API 是服务与客户端之间的契约。无论您选择何种 IPC 机制，使用接口定义语言（interface definition language，IDL）来严格定义服务 API 都是非常有必要的。有论据证明使用 API 优先法定义服务更加合理。在对需要实现的服务 API 定义进行迭代之后，您可以通过编写接口定义并与客户端开发人员进行审阅来开始开发服务。这样设计可以增加您的成功机率，以构建出符合客户端需求的服务。

正如您将会在后面看到，定义 API 的方式取决于您使用何种 IPC 机制。如果您正在使用消息传递，那么 API 是由消息通道和消息类型组成。如果您使用的是 HTTP，那么 API 是由 URL、请求和响应格式组成。稍后我们将详细地介绍关于 IDL 方面的内容。

3.4 演化 API

服务 API 总是随着时间而变化。在单体应用程序中，更改 API 和更新所有调用者通常是一件直截了当的事。但在基于微服务的应用程序中，即使 API 的所有消费者都是同一应用程序中的其他服务，要想完成这些工作也是非常困难的。通常，您无法强制所有客户端与服务升级的节奏一致。此外，您可能需要逐步部署服务的新版本，以便新旧版本的服务同时运行。因此，制定这些问题的处理策略还是很重要的。

处理 API 变更的方式取决于变更的程度。某些更改是次要或需要向后兼容以前的版本。例如，您可能会向请求或响应添加属性。此时设计客户端与服务遵守鲁棒性原则就显得很有意义了。使用较旧 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，并且客户端忽略所有多余的响应属性。使用 IPC 机制和消息格式非常重要，他们可以让您轻松地演化 API。

但有时候，您必须对 API 作出大量不兼容的更改。由于您无法强制客户端立即升级，服务也必须支持较旧版本的 API 一段时间。如果您使用了基于 HTTP 的机制（如 REST），则一种方法是将版本号嵌入 URL 中。每个服务实例可能同时处理多个版本。或者，您可以部署多个不同的实例，每个实例用于处理特定版本。

3.5 处理局部故障

正如第二章中关于 API 网关所述，在分布式系统中存在局部故障风险。由于客户端进程与服务进程是分开的，服务可能无法及时响应客户端的请求。由于故障或者维护，服务可能需要关闭。也有可能因服务过载，造成响应速度变得极慢。

例如，请回想第二章中的产品详细信息场景。我们假设 Recommendation Service 没有响应。客户端天真般的实现可能会无限期地阻塞以等待响应。这不仅会导致用户体验糟糕，而且在许多应用程序中，它将消耗如线程之类等宝贵资源。以致最终，在运行时将线程用完，造成无法响应，如图 3-3 所示。

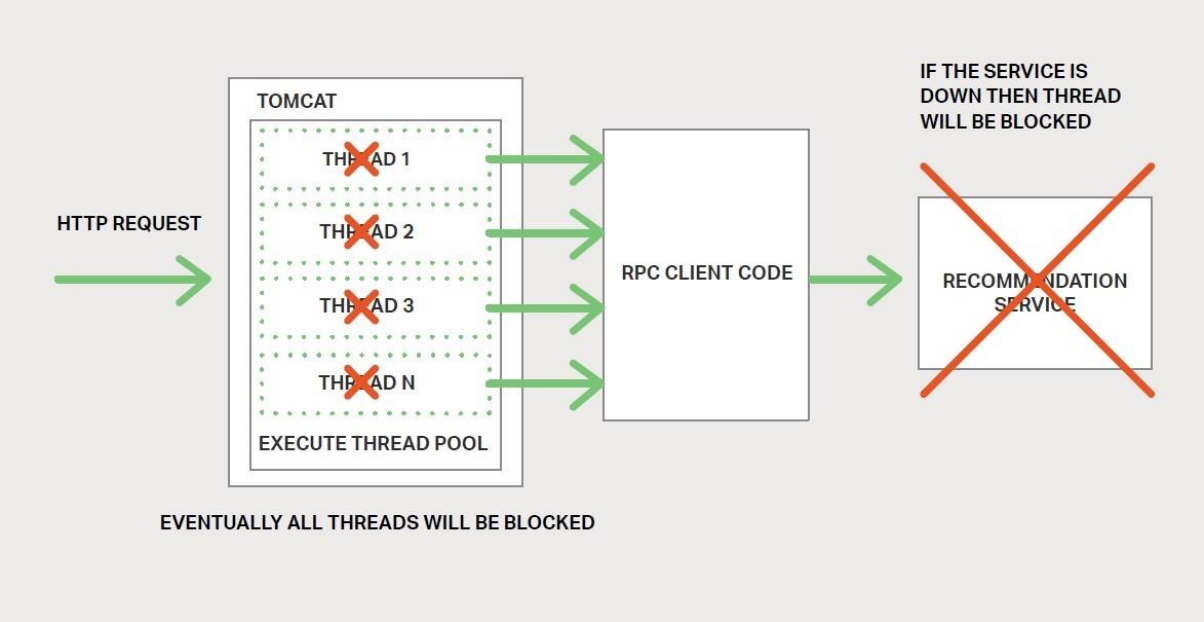


图 3-3、因无响应服务引起的线程阻塞

为了防止出现此类问题，您必须设计您的服务以处理局部故障。以下是一个由 Netflix 给出的好办法。处理局部故障的策略包括：

- **网络超时**
  在等待响应时，不要无限期地阻塞，始终使用超时方案。使用超时方案确保资源不被无限地消耗。

- **限制未完成的请求数量**
  对客户端拥有特定服务的未完成请求的数量设置上限。如果达到了上限，发出的额外请求可能是毫无意义的，因此这些尝试需要立即失败。

- **断路器模式**
  追踪成功和失败请求的数量。如果错误率超过配置阈值，则断开断路器，以便后续的尝试能立即失败。如果出现大量请求失败，则表明服务不可用，发送请求将是无意义的。发生超时后，客户端应重新尝试，如果成功，则关闭断路器。

- **提供回退**
  请求失败时执行回退逻辑。例如，返回缓存数据或者默认值，如一组空白的推荐数据。

Netflix Hystrix 是一个实现上述和其他模式的开源库。如果您正在使用 JVM，那么您一定要考虑使用 Hystrix。如果您在非 JVM 环境中运行，则应使用相等作用的库。

3.6 IPC 技术

有多种 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，比如基于 HTTP 的 REST 或 Thrift。或者，可以使用异步、基于消息的通信机制，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、基于文本的格式，如 JSON 或 XML。或者，可以使用如 Avro 或 Protocol Buffers 等二进制格式（更加高效）。稍后我们将讨论同步 IPC 机制，但在此之前让我们先来讨论一下异步 IPC 机制。

3.7 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。客户端通过发送消息向服务发出请求。如果服务需要回复，则通过向客户端发送一条单独的消息来实现。由于通信是异步的，因此客户端不会阻塞等待回复。相反，客户端被假定不会立即收到回复。

一条消息由头部（如发件人之类的元数据）和消息体组成。消息通过通道进行交换。任何数量的生产者都可以向通道发送消息。类似地，任何数量的消费者都可以从通道接收消息。有两种通道类型，分别是点对点（point-to-point）与发布订阅（publish-subscribe）：

- **点对点通道**发送一条消息给一个切确的、正在从通道读取消息的消费者。服务使用点对点通道，就是上述的一对一交互方式。
- **发布订阅通道**将每条消息传递给所有已订阅的消费者。服务使用发布订阅通道，就是上述的一对多交互方式。

图 3-4 展示了打车应用程序如何使用发布订阅通道。

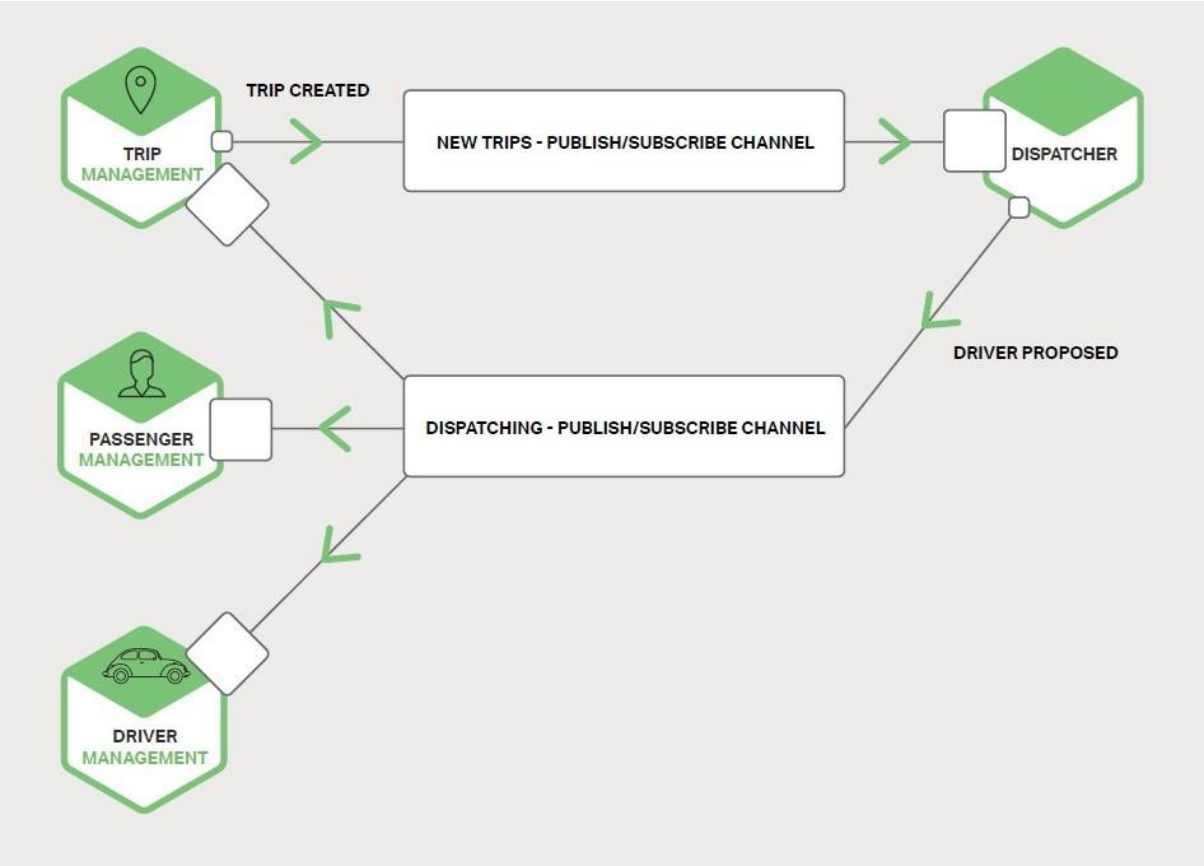


图 3-4、使用了发布/订阅通道的打车应用

Trip Management 服务通过向发布订阅通道写入 Trip Created 消息来通知已订阅的服务，如 Dispatcher。Dispatcher 找到可用的司机并通过向发布订阅通道写入 Driver Proposed 消息来通知其他服务。有许多消息系统可供选择，您应该选择一个支持多种编程语言的。一些消息系统支持标准协议，如 AMQP 和 STOMP。其他消息系统有专有的文档化协议。

有大量的开源消息系统可供选择，包括 RabbitMQ、Apache Kafka、Apache ActiveMQ 和 NSQ。从高层而言，他们都支持某种形式的消息和通道。他们都力求做到可靠、高性能和可扩展。然而，每个代理的消息传递模型细节上都存在着很大差异。

使用消息传递有很多优点：

- **将客户端与服务分离**
  客户端通过向相应的通道发送一条消息来简单地发出一个请求。服务实例对客户端而言是透明的。客户端不需要使用发现机制来确定服务实例的位置。

- **消息缓冲**
  使用如 HTTP 的同步请求/响应协议，客户端和服务在交换期间必须可用。相比之下，消息代理会将消息写入通道入队，直到消费者处理它们。这意味着，例如，即使订单执行系统出现缓慢或不可用的情况，在线商店还是可以接受客户的订单。订单消息只需要简单地排队。

- **灵活的客户端 — 服务交互**
  消息传递支持前面提到的所有交互方式。
- **毫无隐瞒的进程间通信**
  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，由于物理因素和局部故障的可能性，他们实际上是完全不同的。消息传递使这些差异变得非常明显，所以开发人员不会被这些虚假的安全感所欺骗。

然而，消息传递也存在一些缺点：

- **额外的复杂操作**
  消息传递系统是一个需要安装、配置和操作的系统组件。消息代理程序必须高度可用，否则系统的可靠性将受到影响。
- **实施基于请求/响应式交互的复杂性**
  请求/响应式交互需要做些工作来实现。每个请求消息必须包含应答通道标识符和相关标识符。该服务将包含相关 ID 的响应消息写入应答信道。客户端使用相关 ID 将响应与请求相匹配。通常使用直接支持请求/响应的 IPC 机制更加容易。

现在我们已经了解了使用基于消息的 IPC，让我们来看看请求/响应的 IPC。

3.8 同步的请求/响应 IPC

当使用基于同步、基于请求/响应的 IPC 机制时，客户端向服务器发送请求。该服务处理该请求并返回响应。

在许多客户端中，请求的线程在等待响应时被阻塞。其他客户端可能会使用异步、事件驱动的客户端代码，这些代码可能是由 Futures 或 Rx Observables 封装的。然而，与使用消息传递不同，客户端假定响应能及时到达。

有许多协议可供选择。有两种流行协议分别是 REST 和 Thrift。我们先来看一下 REST。

3.8.1 REST

如今，开发 RESTful 风格的 API 是很流行的。REST 是一种使用了 HTTP （几乎总是）的 IPC 机制。

资源是 REST 中的一个关键概念，它通常表示业务对象，如客户、产品或这些业务对象的集合。REST 使用 HTTP 动词（谓词）来操纵资源，这些资源通过 URL 引用。例如，GET 请求返回一个资源的表述形式，可能是 XML 文档或 JSON 对象形式。POST 请求创建一个新资源，PUT 请求更新一个资源。

引用 REST 创建者 Roy Fielding：

> “REST 提供了一套架构约束，当应用作为整体时，其强调组件交互的可扩展性、接口的通用性、组件的独立部署以及中间组件，以减少交互延迟、实施安全性和封装传统系统。” — Roy Fielding，**《架构风格与基于网络的软件架构设计》**

图 3-5 展示了打车应用程序可能使用 REST 的方式之一。

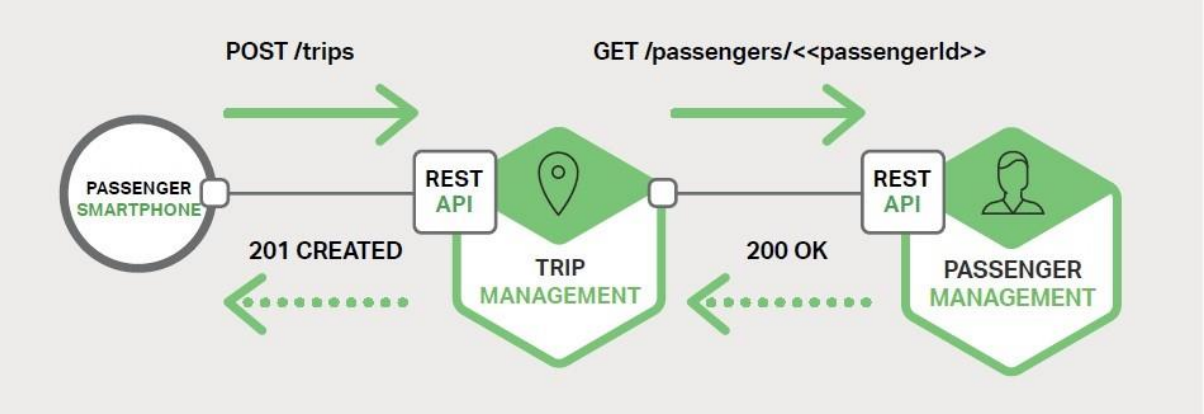


图 3-5、使用了 RESTful 交互的打车应用

乘客的智能手机通过向 Trip Management 服务的 **/trips** 资源发出一个 POST 请求来请求旅程。该服务通过向 Passenger Management 服务发送一个获取乘客信息的 GET 请求来处理该请求。在验证乘客被授权创建旅程后，Trip Management 服务将创建旅程，并向智能手机返回 201 响应。

许多开发人员声称其基于 HTTP 的 API 就是 RESTful。然而，正如 Fielding 在这篇博文中所描述的那样，并不是都是这样。

Leonard Richardson 定义了一个非常有用的 REST 成熟度模型，包括以下层次：

- **级别 0** 的 API 的客户端通过向其唯一的 URL 端点发送 HTTP POST 请求来调用该服务。每个请求被指定要执行的操作、操作的目标（如业务对象）以及参数。
- **级别 1** 的 API 支持资源概念。要对资源执行操作，客户端会创建一个 POST 请求，指定要执行的操作和参数。
- **级别 2** 的 API 使用 HTTP 动词（谓词）执行操作：使用 GET 检索、使用 POST 创建和使用 PUT 进行更新。请求查询参数和请求体（如果有）指定操作的参数。这使服务能够利用到 Web 的基础特性，如缓存 GET 请求。
- **级别 3** 的 API 基于非常规命名原则设计，HATEOAS（Hypermedia as the engine of application state，超媒体即应用程序状态引擎）。基本思想是 GET 请求返回的资源的表述，包含用于执行该资源上允许的操作的链接。例如，客户端可以

使用发送 GET 请求检索订单返回的订单响应中的链接来取消订单。HATEOAS 的一个好处是不再需要将 URL 硬编码在客户端代码中。另一个好处是，由于资源的表示包含可允许操作的链接，所以客户端不必猜测可以对当前状态的资源执行什么操作。

使用基于 HTTP 的协议有很多好处：

- HTTP 简单易懂。
- 您可以使用浏览器扩展（如 Postman）来测试 HTTP API，或者使用 curl 命令行测试 HTTPAPI（假设使用了 JSON 或其他一些文本格式）。
- 它直接支持请求/响应式通信。
- HTTP 属于防火墙友好。
- 它不需要中间代理，简化了系统架构。

使用 HTTP 也存在一些缺点：

- HTTP 仅直接支持请求/响应的交互方式。您可以使用 HTTP 进行通知，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（没有一个中间者来缓冲消息），所以它们必须在交换期间都运行着。
- 客户端必须知道每个服务实例的位置（即 URL）。如第二章关于 API 网关所述，这是现代应用程序中的一个复杂问题。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 接口定义语言的价值。有几个可以选择，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，需要您使用一个单独的规范，如 JSON 模式。除了用于描述 API 之外，IDL 通常还具有可从接口定义生成客户端 stub 和服务器 skeleton 的工具。

3.8.2 Thrift

Apache Thrift 是 REST 的一个有趣的替代方案。它是一个用于编写跨语言 RPC 客户端和服务器 skeleton。Thrift 提供了一个 C 风格的 IDL 来定义您的 API。您可以使用 Thrift 编译器生成客户端 stub 和服务器端 skeleton。编译器可以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。服务定义类似于一个 Java 接口。它是强类型方法的集合。Thrift 方法可以返回一个（可能为 void）值，或者如果它们被定义为单向，则不会返回值。返回值方法实现了请求/响应的交互方式，客户端等待响应，并可能会抛出异常。单向方式对应通知互动方式，服务器不发送响应。

Thrift 支持多种消息格式：JSON，二进制和压缩二进制。二进制比 JSON 更有效率，因为其解码速度更快。而且，顾名思义，压缩二进制是一种节省空间的格式。当然，JSON 是人性化和浏览器友好的。Thrift 还为您提供了包括原始 TCP 和 HTTP 在内的传输协议选择。原始 TCP 可能比 HTTP 更有效率。然而，HTTP 是防火墙友好的、浏览器友好的和人性化的。

3.9 消息格式

我们已经了解了 HTTP 和 Thrift，现在让我们来看看消息格式的问题。如果您使用的是消息系统或 REST，则可以选择自己的消息格式。其他 IPC 机制如 Thrift 可能只支持少量的消息格式，甚至只支持一种。在任一种情况下，使用跨语言消息格式就显得非常重要了。即使您现在是以单一语言编写您的微服务，您将来也可能会使用到其他语言。

有两种主要的消息格式：文本和二进制。基于文本格式的例子有 JSON 和 XML。这些格式的优点在于，它们不仅是人类可读的，而且是自描述的。在 JSON 中，对象的属性由键值对集合表示。类似地，在 XML 中，属性由命名元素和值表示。这使得消息消费者能够挑选其感兴趣的值并忽略其余的值。因此，可以轻松地向后兼容作出微小更改的消息格式。

XML 文档的结构由 XML 模式（schema）指定。随着时间的推移，开发人员社区已经意识到 JSON 也需要一个类似的机制。一个选择是使用 JSON Schema，无论独立或作为 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，特别是 XML。因为消息是自描述的，每个消息除了它们的值之外还包含属性的名称。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。如果您使用的是 Thrift RPC，您可以使用二进制 Thrift。如果您选择的消息格式，包括了流行的 Protocol Buffers 和 Apache Avro。这两种格式都提供了一种用于定义消息结构的类型 IDL。然而，一个区别是 Protocol Buffers 使用标记字段，而 Avro 消费者需要知道模式才能解释消息。因此，Protocol Buffers 的 API 演化比 Avro 更容易使用。这里有篇博文对 Thrift、Protocol Buffers 和 Avro 作出了极好的比较。

3.10 总结

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、如何演变 API 以及如何处理局部故障。微服务可以使用两种 IPC 机制：异步消息传递和同步请求/响应。为了进行通信，一个服务必须能够找到另一个服务。在第四章中，我们将介绍微服务架构中服务发现问题。

微服务实战：NGINX 与应用程序架构

by Floyd Smith

NGINX 使您能够实现各种伸缩和镜像操作，使您的应用程序更加灵敏和高度可用。您为伸缩和镜像所作的选择会影响到您如何进行进程间通信，这是本章的主题。

我们在 NGINX 方面建议您在实现基于微服务的应用程序时考虑使用四层架构。Forrester 在这方面有详细的报告，您可以从 NGINX 上免费下载。这些层代表客户端（包括台式机或笔记本电脑、移动、可穿戴或 IoT 客户端）、交付、聚合（包括数据存储）和服务，其中包括应用功能和特定服务，而不是共享数据存储。

四层架构比以前的三层架构更加灵活，具有可扩展、响应灵敏、移动友好，并且内在支持基于微服务的应用程序开发和交付等优点。像 Netflix 和 Uber 这样的行业引领者能够通过使用这种架构来实现用户所需的性能水平。

NGINX 本质上非常适合四层架构，从客户端层的媒体流，到交付层的负载均衡与缓存、聚合层的高性能和安全的基于 API 的通信的工具，以及服务层中支持灵活管理的短暂服务实例。

同样的灵活性使得 NGINX 可以实现强大的伸缩和镜像模式，以处理流量变化，防止安全攻击，此外还提供可用的故障配置切换，从而实现高可用。

在更为复杂的架构中，包括服务实例实例化和需求不断的服务发现，解耦的进程间通信往往更受青睐。异步和一对多通信方式可能比高耦合的通信方式更加灵活，它们最终提供更高的性能和可靠性。

4 服务发现

本书主要介绍如何使用微服务来构建应用程序，现在是第四章。第一章已经介绍了微服务架构模式，并讨论了使用微服务的优点与缺点。第二章和第三章介绍了微服务间的通信，并对不同的通信机制作出对比。在本章中，我们将探讨服务发现（service discovery）相关的内容。

4.1 为何使用服务发现

我们假设您正在编写某些代码，这些代码调用了有 REST API 或 Thrift API 的服务。为了发送一个请求，您的代码需要知道服务实例的网络位置（IP 地址与端口）。在运行于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，在现代基于云的微服务应用中，这是一个更难解决的问题，如图 4-1 所示。

服务实例具有动态分配的网络位置。此外，由于自动扩缩、故障与升级，整组服务实例会动态变更。因此，您的客户端代码需要使用更精确的服务发现机制。

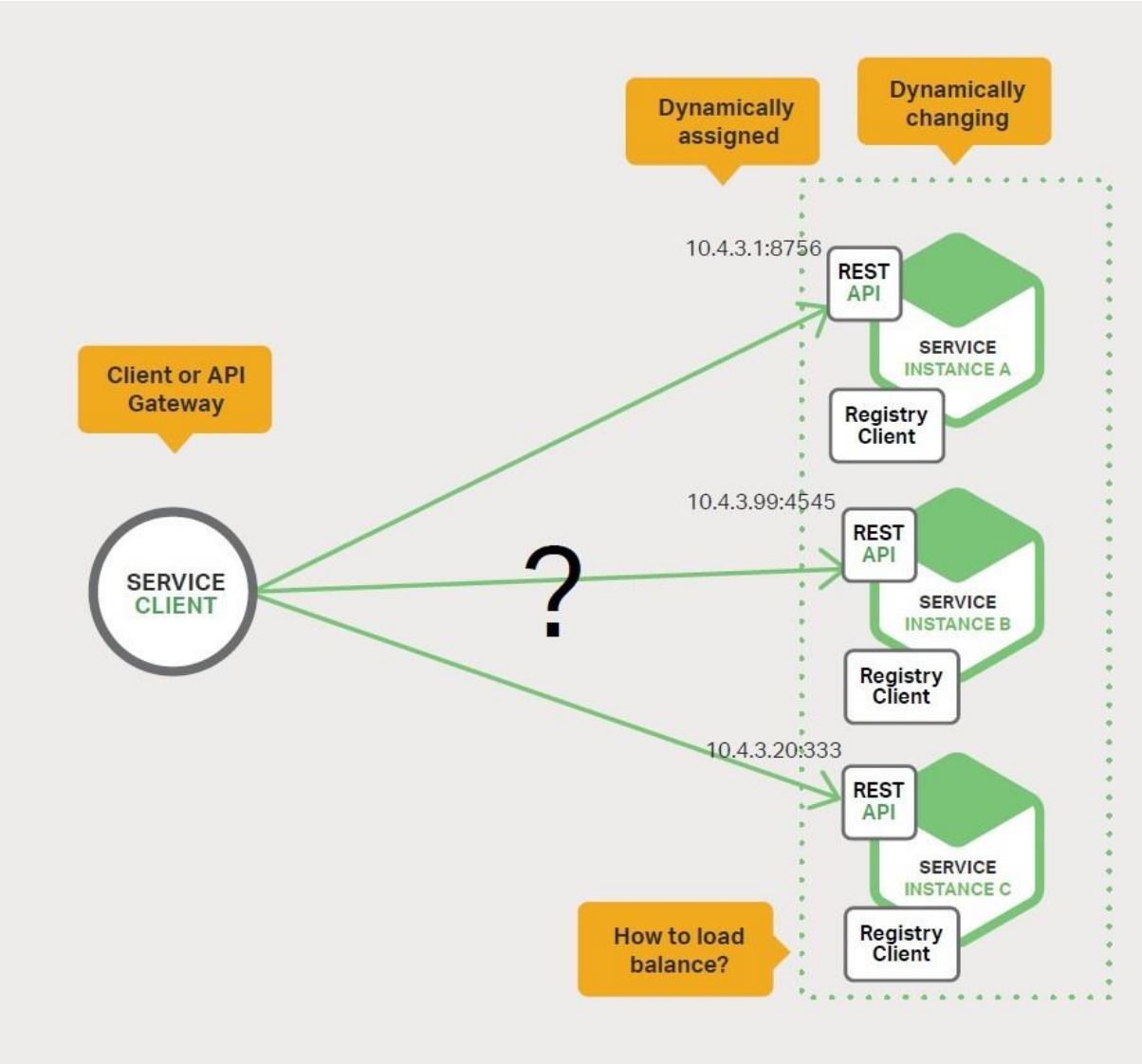


图 4-1、需要服务寻找帮助的客户端或 API 网关

有两种主要的服务发现模式：客户端发现（client-side discovery）与服务端发现（server-side discovery）。让我们先来看看客户端发现。

4.2 客户端发现模式

当使用客户端发现模式时，客户端负责确定可用服务实例的网络位置和请求负载均衡。客户端查询服务注册中心（service registry），它是可用服务实例的数据库。之后，客户端利用负载均衡算法选择一个可用的服务实例并发出请求。

图 4-2 展示了该模式的结构

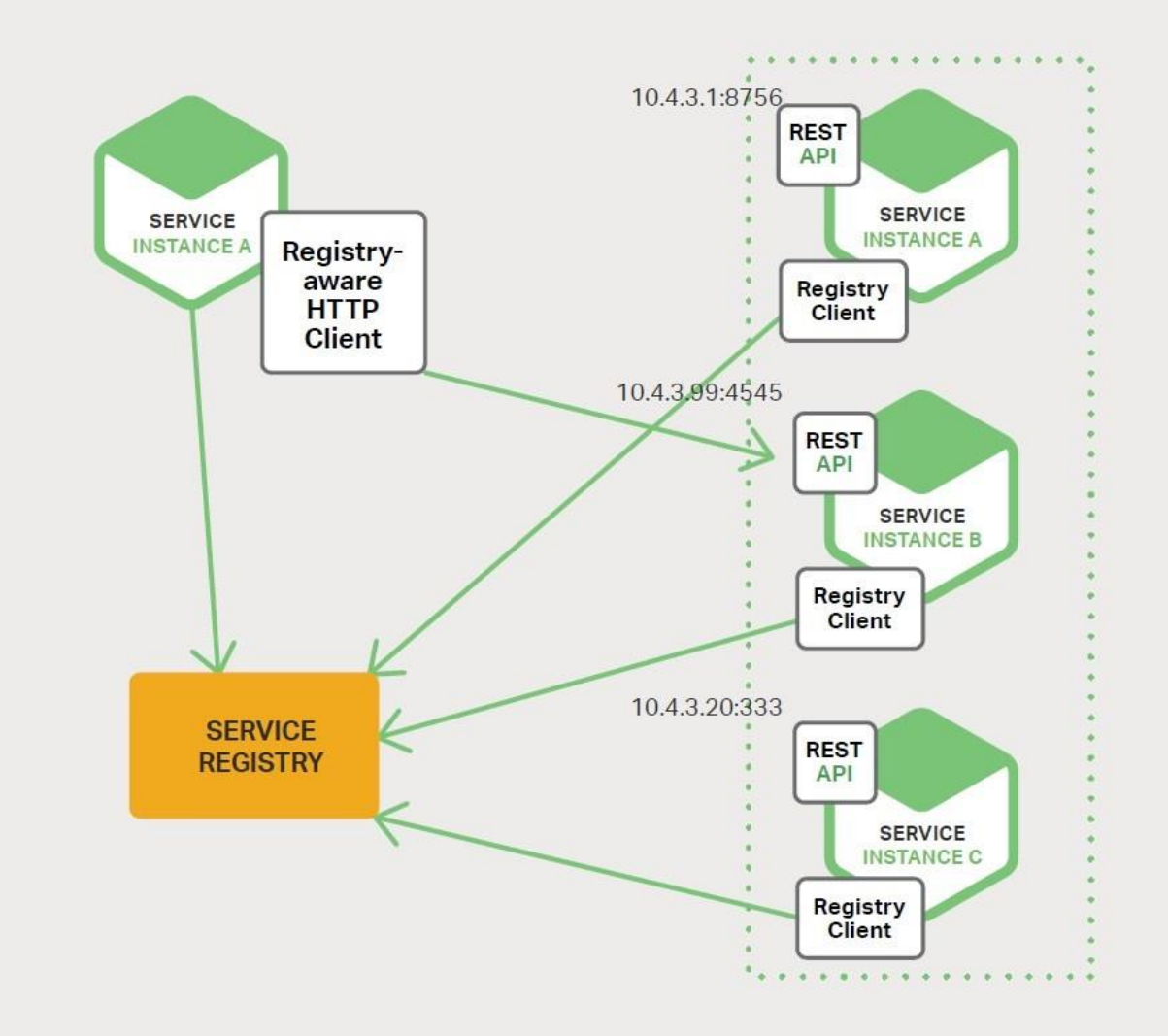


图 4-2、客户端可以承担发现服务任务

服务实例的网络位置在服务注册中心启动时被注册。当实例终止时，它将从服务注册中心中移除。通常使用心跳机制周期性地刷新服务实例的注册信息。

Netflix OSS 提供了一个很好的客户端发现模式示例。Netflix Eureka 是一个服务注册中心，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 Ribbon 是一个 IPC 客户端，可与 Eureka 一起使用，用于在可用服务实例之间使请求负载均衡。本章稍后将讨论 Eureka。

客户端发现模式存在各种优点与缺点。该模式相对比较简单，除了服务注册中心，没有其他移动部件。此外，由于客户端能发现可用的服务实例，因此可以实现智能的，特定于应用程序的负载均衡决策，比如使用一致性哈希。该模式的一个重要缺点是它将客户端与服务注册中心耦合在一起。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。

现在我们已经了解了客户端发现，接下来让我们看看服务端发现。

4.3 服务端发现模式

服务发现的另一种方式是服务端发现模式。图 4-3 展示了该模式的结构：

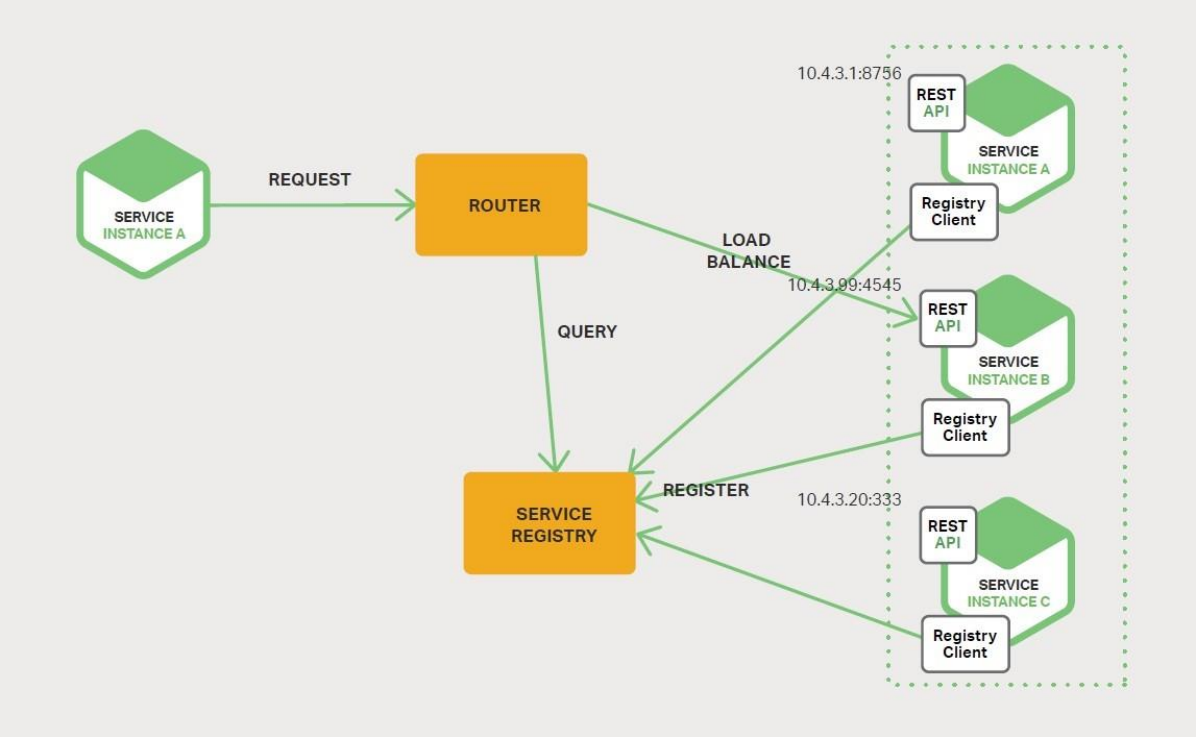


图 4-3、服务器间也可以处理服务发现

客户端通过负载均衡器向服务发出请求。负载均衡器查询服务注册中心并将每个请求路由到可用的服务实例。与客户端发现一样，服务实例由服务注册中心注册与销毁。

AWS Elastic Load Balancer（ELB）是一个服务端发现路由示例。ELB 通常用于负载均衡来自互联网的外部流量。然而，您还可以使用 ELB 来负载均衡虚拟私有云（VPC）内部的流量。客户端通过 ELB 使用其 DNS 名称来发送请求（HTTP 或 TCP）。ELB 负载均衡一组已注册的 Elastic Compute Cloud（EC2）实例或 EC2 Container Service（ECS）容器之间的流量。这里没有单独可见的服务注册中心。相反，EC2 实例与 ECS 容器由 ELB 本身注册。

HTTP 服务器和负载均衡器（如 NGINX Plus 和 NGINX）也可以作为服务端发现负载均衡器。例如，此博文描述了使用 Consul Template 动态重新配置 NGINX 反向代理。Consul Template 是一个工具，可以从存储在 Consul 服务注册中心中的配置数据中定期重新生成任意配置文件。每当文件被更改时，它都会运行任意的 shell 命令。在列举的博文描述的示例中，Consul Template 会生成一个 nginx.conf 文件，该文件配置了反向代理，然后通过运行一个命令告知 NGINX 重新加载配置的命令。更复杂的实现可以使用其 HTTP API 或 DNS 动态重新配置 NGINX Plus。

某些部署环境（如 Kubernetes 和 Marathon）在群集中的每个主机上运行着一个代理。这些代理扮演着服务端发现负载均衡器角色。为了向服务发出请求，客户端通过代理

使用主机的 IP 地址和服务的分配端口来路由请求。之后，代理将请求透明地转发到在集群中某处运行的可用服务实例。

服务端发现模式有几个优点与缺点。该模式的一大的优点是把发现的细节从客户端抽象出来。客户端只需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都实现发现逻辑的必要性。另外，如上所述，一些部署环境免费提供此功能。然而，这种模式存在一些缺点。除非负载均衡器由部署环境提供，否则您需要引入这个高可用系统组件，并进行设置和管理。

4.4 服务注册中心

服务注册中心（service registry）是服务发现的一个关键部分。它是一个包含了服务实例网络位置的数据

库。服务注册中心必须是高可用和最新的。虽然客户端可以缓存从服务注册中心获得的网络位置，但该信息最终会过期，客户端将无法发现服务实例。因此，服务注册中心由使用了复制协议（replication protocol）来维护一致性的服务器集群组成。

如之前所述，Netflix Eureka 是一个很好的服务注册中心范例。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 请求注册其网络位置。它必须每隔 30 秒使用 PUT 请求来刷新其注册信息。通过使用 HTTP DELETE 请求或实例注册超时来移除注册信息。正如您所料，客户端可以使用 HTTP GET 请求来检索已注册的服务实例。

Netflix 通过在每个 Amazon EC2 可用性区域（Availability Zone）中运行一个或多个 Eureka 服务器来实现高可用。每个 Eureka 服务器都运行在具有一个 Elastic IP 地址的 EC2 实例上。DNS TEXT 记录用于存储 Eureka 集群配置，这是一个从可用性区域到 Eureka 服务器的网络位置列表的映射。当 Eureka 服务器启动时，它将会查询 DNS 以检索 Eureka 群集配置，查找其对等体，并为其分配一个未使用的 Elastic IP 地址。

经过 Eureka 客户端 — 服务与服务客户端 — 查询 DNS 以发现 Eureka 服务器的网络位置。客户端优先使用相同可用性区域中的 Eureka 服务器，如果没有可用的，则使用另一个可用性区域的 Eureka 服务器。

以下列举了其他服务册中心注：

- etcd
  一个用于共享配置和服务发现的高可用、分布式和一致的键值存储。使用了 etcd 的两个著名项目分别为 Kubernetes 和 Cloud Foundry。

- Consul
  一个发现与配置服务工具。它提供了一个 API，可用于客户端注册与发现服务。Consul 可对服务进行健康检查，以确定服务的可用性。

- Apache ZooKeeper
  一个被广泛应用于分布式应用程序的高性能协调服务。Apache ZooKeeper 最初是一个 Hadoop 子项目，但现在已经成为一个独立的顶级项目。

另外，如之前所述，部分系统，如 Kubernetes、Marathon 和 AWS，没有明确的服务注册中心。相反，服务注册中心只是基础设施的一个内置部分。

现在我们已经了解服务注册中心的概念，接下来让我们看看服务实例是如何被注册到服务注册中心。

4.5 服务注册方式

如之前所述，服务实例必须在服务注册中心中注册与注销。有几种不同的方式来处理注册和注销。一是服务实例自我注册，即自注册模式。另一个是使用其他系统组件来管理服务实例的注册，即第三方注册模式。我们先来了解自注册模式。

4.6 自注册模式

当使用自注册模式时，服务实例负责在服务注册中心注册和注销自己。此外，如果有必要，服务实例将通过发送心跳请求来防止其注册信息过期。

图 4-4 展示了该模式的结构。

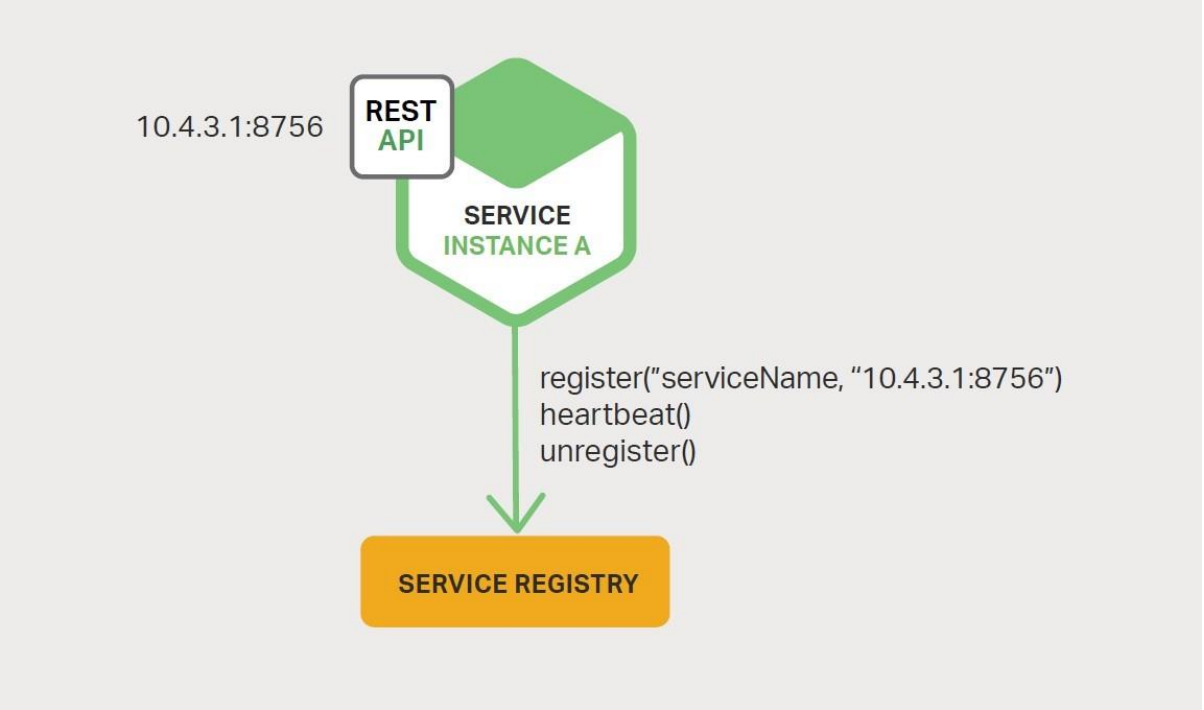


图 4-4、服务可以自我处理注册

该方式的一个很好的范例就是 Netflix OSS Eureka 客户端。Eureka 客户端负责处理服务实例注册与注销

的所有方面。实现了包括服务发现在内的多种模式的 Spring Cloud 项目可以轻松地使用 Eureka 自动注册服务实例。您只需在 Java Configuration 类上应用 **`@EnableEurekaClient`** 注解即可。

自注册模式有好有坏。一个好处是它相对简单，不需要任何其他系统组件。然而，主要缺点是它将服务实例与服务注册中心耦合。您必须为服务使用的每种编程语言和框架都实现注册代码。

将服务与服务注册中心分离的替代方法是第三方注册模式。

4.7 第三方注册模式

当使用第三方注册模式时，服务实例不再负责向服务注册中心注册自己。相反，该工作将由被称为服务注册器（service registrar）的另一系统组件负责。服务注册器通过轮询部署环境或订阅事件来跟踪运行实例集的变更情况。当它检测到一个新的可用服务实例时，它会将该实例注册到服务注册中心。此外，服务注册器可以注销终止的服务实例。

图 4-5 展示了该模式的结构：

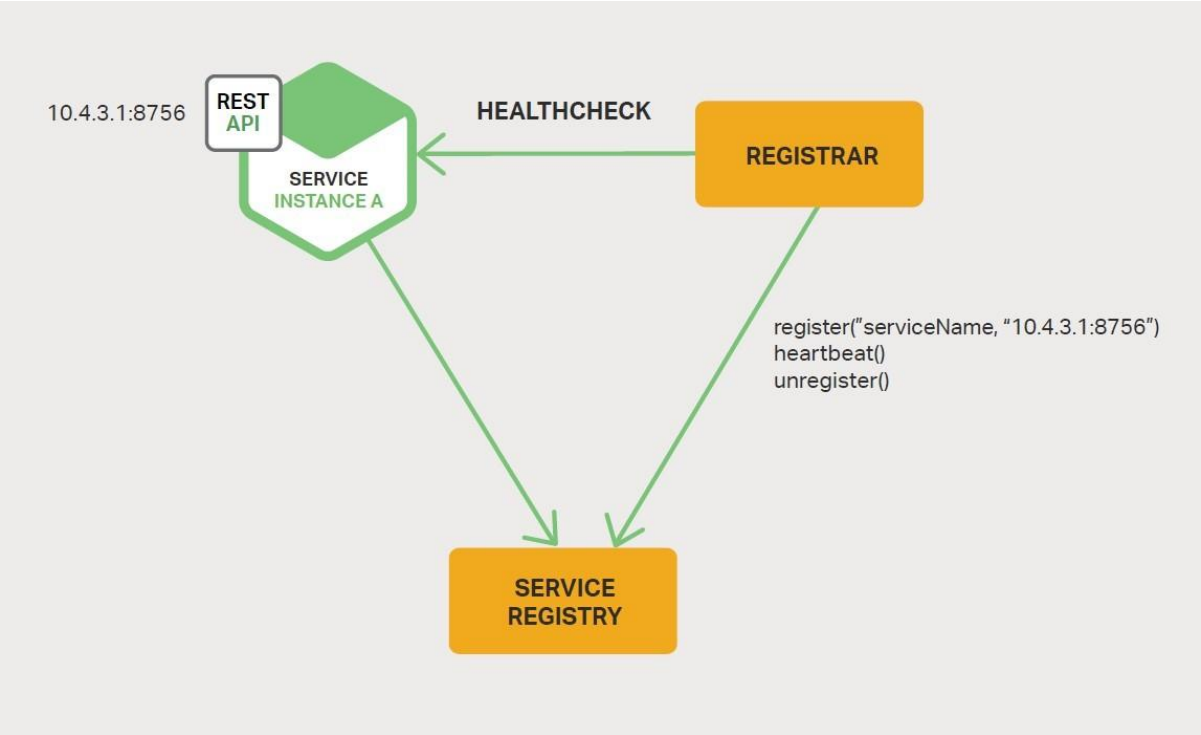


图 4-5、一个单独的服务注册器可负责注册其他服务

开源的 Registrator 项目是一个很好的服务注册器示例。它可以自动注册和注销作为 Docker 容器部署的服务实例。注册器支持多个服务注册中心，包括 etcd 和 Consul。

另一个服务注册器例子是 NetflixOSS Prana。其主要用于非 JVM 语言编写的服务，它是一个与服务实例并行运行的侧中应用。Prana 使用了 Netflix Eureka 来注册和注销服务实例。

服务注册器在部分部署环境中是一个内置组件。Autoscaling Group 创建的 EC2 实例可以自动注册到 ELB。Kubernetes 服务将自动注册并提供发现。

第三方注册模式同样有好有坏。一个主要的好处是服务与服务注册中心之间解耦。您不需要为开发人员使用的每种编程语言和框架都实现服务注册逻辑。相反，仅需要在专用服务中以集中的方式处理服务实例注册。

该模式的一个缺点是，除非部署环境内置，否则您同样需要引入这样一个高可用的系统组件，并进行设置和管理。

4.8 总结

在微服务应用程序中，运行的服务实例集会动态变更。实例具有动态分配的网络位置。因此，为了让客户端向服务发出请求，它必须使用服务发现机制。

服务发现的一个关键部分是服务注册中心。服务注册中心是一个可用服务实例的数据库。服务注册中心提供了管理 API 和查询 API 的功能。服务实例通过使用管理 API 从服务注册中心注册或者注销。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：客户端发现与服务端发现。在使用了客户端服务发现的系统中，客户端查询服务注册中心，选择一个可用实例并发出请求。在使用了服务端发现的系统中，客户端通过路由进行请求，路由将查询服务注册中心，并将请求转发到可用实例。

服务实例在服务注册中心中注册与注销有两种主要方式。一个是服务实例向服务注中心自我注册，即自注册模式。另一个是使用其他系统组件代表服务完成注册与注销，即第三方注册模式。

在某些部署环境中，您需要使用如 Netflix Eureka 或 Apache ZooKeeper 等服务注册中心来设置您自己的服务发现基础设施。在其他部署环境中，服务发现是内置的，例如，Kubernetes 和 Marathon，可以处理服务实例的注册与注销。他们还在每一个扮演服务端发现路由角色的集群主机上运行一个代理。

一个 HTTP 反向代理和负载均衡器（如 NGINX）也可以用作服务端发现负载均衡器。服务注册中心可以将路由信息推送给 NGINX，并调用一个正常的配置更新；例如，您可以使用 Consul Template。NGINX Plus 支持额外的动态重新配置机制 — 它可以使用 DNS 从注册中心中提取有关服务实例的信息，并为远程重新配置提供一个 API。

微服务实战：NGINX 的灵活性

by Floyd Smith

在微服务环境中，由于自动扩缩、故障和升级，您的后端基础设施可能会不断变化，这些变化包括了服务的创建，部署和扩展。如本章所述，在动态重新分配服务位置的环境中需要服务发现机制。

将 NGINX 应用于微服务的一部分好处是，您可以轻松地配置其自动响应后端基础设施作出的变更。

NGINX 配置不仅简单灵活，而且兼容 Amazon Web Services 使用的模板，可以更轻松地管理特定的服务变更与受负载均衡的变更服务组。

NGINX Plus 具有即时重新配置 API，无需重新启动 NGINX Plus 或手动重新加载配置就能感知受负载均衡服务组的变更。在 NGINX Plus Release 8 及更高版本中，您可以将对 API 所做的变更配置为在重新启动和配置重新加载时保持不变。（重新加载不需要重新启动，不要断开连接）NGINX Plus Release 9 及更高版本支持使用 DNS SRV 记录进行服务发现，可与现有服务器发现平台（如 Consul 和 etcd）进行更紧密地集成。

我们在 NGINX 创建了一个用于管理服务发现的模型：

1. 为几个应用程序的每个应用运行单独的 Docker 容器，包括如 etcd 的服务发现应用程序、服务注册工具、一个或多个后端服务器以及用于负载均衡其他容器的 NGINX Plus 本身。
2. 注册工具监控 Docker 的新容器，并使用服务发现工具注册新服务，此外，还可以删除消失的容器。
3. 容器及其运行的服务将自动添加到负载均衡上游服务器中或从中删除。

此 Demo 应用程序可用于多个服务发现应用程序：Consul API、来自 Consul 的 DNS SRV 记录、etcd 以及 ZooKeeper 等。

5 事件驱动数据管理

本书主要介绍如何使用微服务构建应用程序，这是本书的第五章。第一章介绍了微服务架构模式，讨论了使用微服务的优点与缺点。第二和第三章描述了微服务架构内通信方式的对比。第四章探讨了与服务发现相关的内容。在本章中，我们稍微做了点调整，研究微服务架构中出现的分布式数据管理问题。

5.1 微服务和分布式数据管理问题

单体应用程序通常具有一个单一的关系型数据库。使用关系型数据库的一个主要优点是您的应用程序可以使用 ACID 事务，这些事务提供了以下重要保障：

- **原子性（Atomicity）** 所作出的改变是原子操作，不可分割
- **一致性（Consistency）** 数据库的状态始终保持一致
- **隔离性（Isolation）** 即使事务并发执行，但他们看起来更像是串行执行
- **永久性（Durable）** 一旦事务提交，它将不可撤销

因此，您的应用程序可以很容易地开始事务、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大好处是它提供了 SQL，这是一种丰富、声明式和标准化的查询语言。您可以轻松地编写一个查询来组合来自多个表的数据，之后，RDBMS 查询计划程序将确定执行查询的最佳方式。您不必担心如何访问数据库等底层细节。因为您所有的应用程序数据都存放在同个数据库中，因此很容易查询。

很不幸的是，当我们转向微服务架构时，数据访问将变得非常复杂。这是因为每个微服务所拥有的数据对当前微服务来说是私有的，只能通过其提供的 API 进行访问。封装数据可确保微服务松耦合，独立演进。如果多个服务访问相同的数据，模式（schema）更新需要对所有服务进行耗时、协调的更新。

更糟糕的是，不同的微服务经常使用不同类型的数据库。现代应用程序存储和处理着各种数据，而关系型数据库并不总是最佳选择。在某些场景，特定的 NoSQL 数据库可能具有更方便的数据模型，提供了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（如 Elasticsearch）是合理的。类似地，存储社交图数据的

服务应该可以使用图数据库，例如 Neo4j。因此，基于微服务的应用程序通常混合使用 SQL 和 NoSQL 数据库，即所谓的混合持久化（polyglot persistence）方式。

一个分区的数据存储混合持久化架构具有许多优点，包括了松耦合的服务以及更好的性能与可扩展性。然而，它也引入了一些分布式数据管理方面的挑战。

第一个挑战是如何实现维护多个服务之间的业务事务一致性。要了解此问题，让我们先来看一个在线 B2B 商店的示例。Customer Service （顾客服务）维护客户相关的信息，包括信用额度。Order Service （订单）负责管理订单，并且必须验证新订单，不得超过客户的信用额度。在此应用程序的单体版本中，OrderService 可以简单地使用 ACID 交易来检查可用信用额度并创建订单。

相比之下，在微服务架构中，ORDER （订单）和 CUSTOMER （顾客）表对其各自的服务都是私有的，如图 5-1 所示：

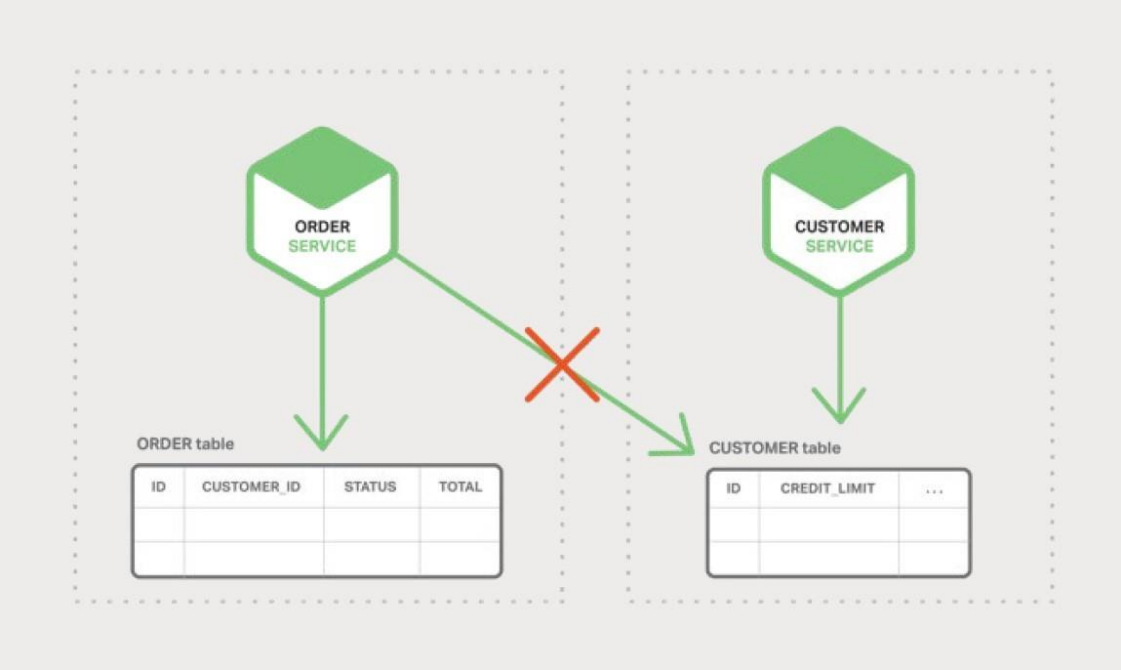


图 5-1、每个微服务都有各自的数据

Order Service 无法直接访问 CUSTOMER 表。它只能使用客户服务提供的 API。订单服务可能使用了分布式事务，也称为两阶段提交（2PC）。然而，2PC 在现代应用中通常是不可行的。CAP 定理要求您在可用性与 ACID 式一致性之间做出选择，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，都不支持 2PC。维护服务和数据库之间的数据一致性至关重要，因此我们需要另一套解决方案。

第二个挑战是如何实现从多个服务中检索数据。例如，我们假设应用程序需要显示一个顾客和他最近的订单。如果 Order Service 提供了用于检索客户订单的 API，那么您可以使用应用程序端连接以检索数据。应用程序从 Customer Service 中检索客户，并从 Order Service 中检索客户的订单。但是，假设 Order Service 仅支持通过主键查找订单（也许它使用了仅支持基于主键检索的 NoSQL 数据库）。在这种情况下，没有有效的方法来检索所需的数据。

5.2 事件驱动架构

许多应用使用了事件驱动架构作为解决方案。在此架构中，微服务在发生某些重要事件时发布一个事件，例如更新业务实体时。其他微服务订阅了这些事件，当微服务接收到一个事件时，它可以更新自己的业务实体，这可能导致更多的事件被发布。

您可以使用事件实现跨多服务的业务事务。一个事务由一系列的步骤组成。每个步骤包括了微服务更新业务实体和发布事件所触发的下一步骤。下图依次展示了如何在创建订单时使用事件驱动方法来检查可用信用额度。

微服务通过 Message Broker （消息代理）进行交换事件：

- Order Service（订单服务）创建一个状态为 NEW 的订单，并发布一个 Order Created （订单创建）事件。

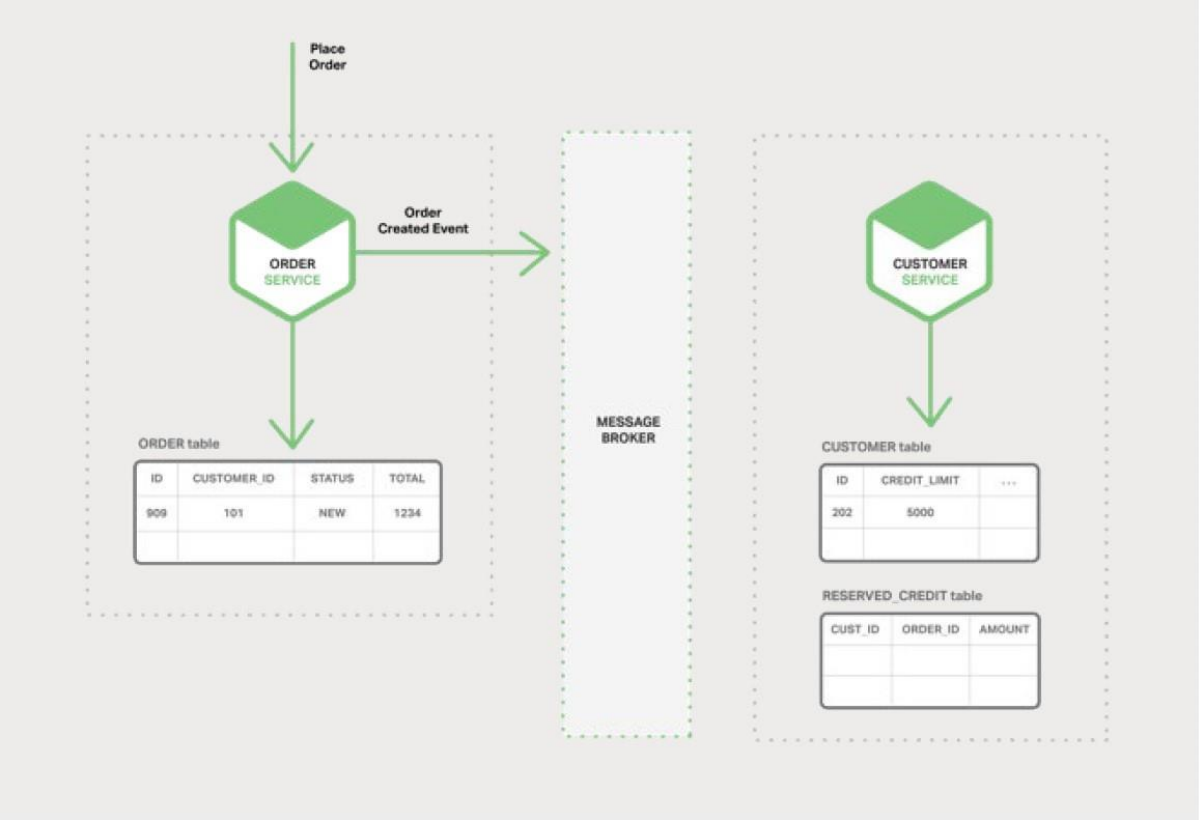


图 5-2、Order Service 发布一个事件

- Customer Service （客户服务）消费了 Order Created 事件，为订单预留信用额度，并发布 Credit Reserved 事件。

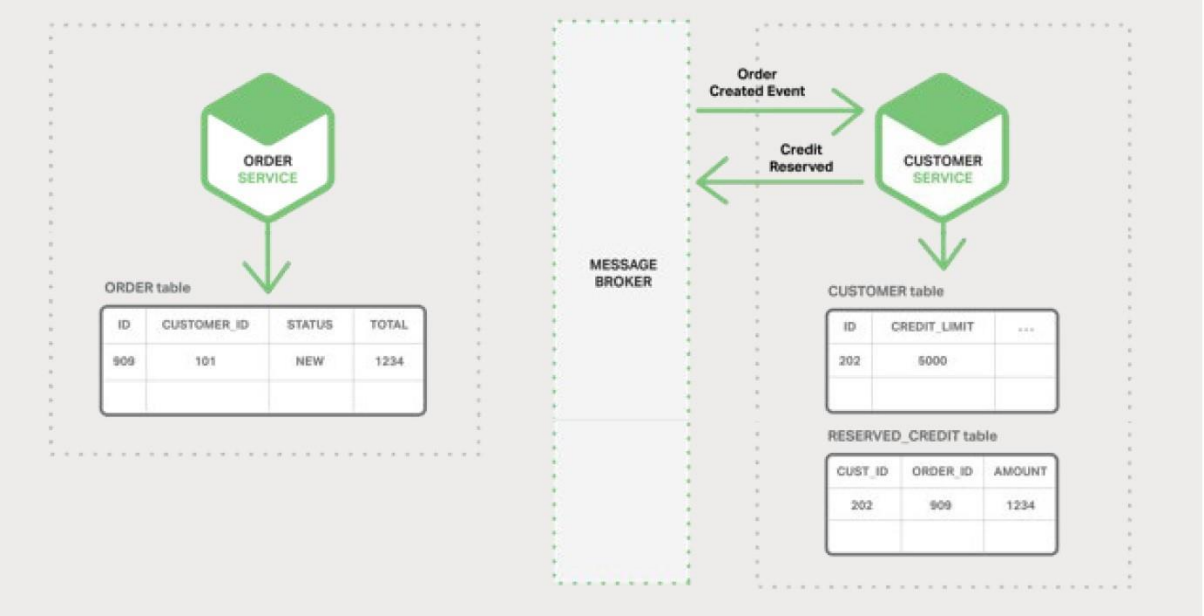


图 5.3、Customer Service 响应

- Order Service 消费了 Credit Reserved （信用预留）事件并将订单的状态更改为 OPEN。

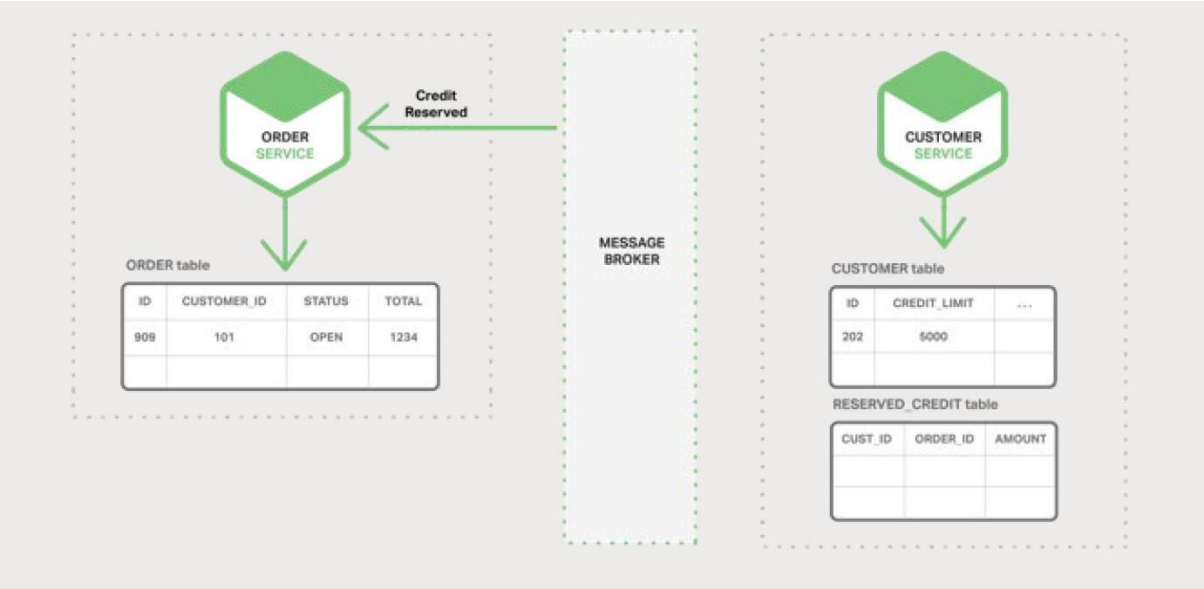


图 5-4、Order Service 作用于响应

更复杂的场景可能会涉及额外的步骤，例如在检查客户信用的同时保留库存。

假设（a）每个服务原子地更新数据库并发布事件，稍后再更新，（b）Message Broker 保证事件至少被传送一次，您可以实现跨多服务的业务事务。需要注意的是，这些并不是 ACID 事务。它们只提供了更弱的保证，如最终一致性。该事务模型称为_BASE 模型。

您还可以使用事件来维护多个微服务预先加入所拥有的数据的物化视图（materialized view）。维护视图

的服务订阅了相关事件并更新视图。图 5-5 展示了 Customer Order View Updater Service （客户订单视图更新服务）根据 Customer Service 和 Order Service 发布的事件更新 Customer Order View （客户订单服务）。

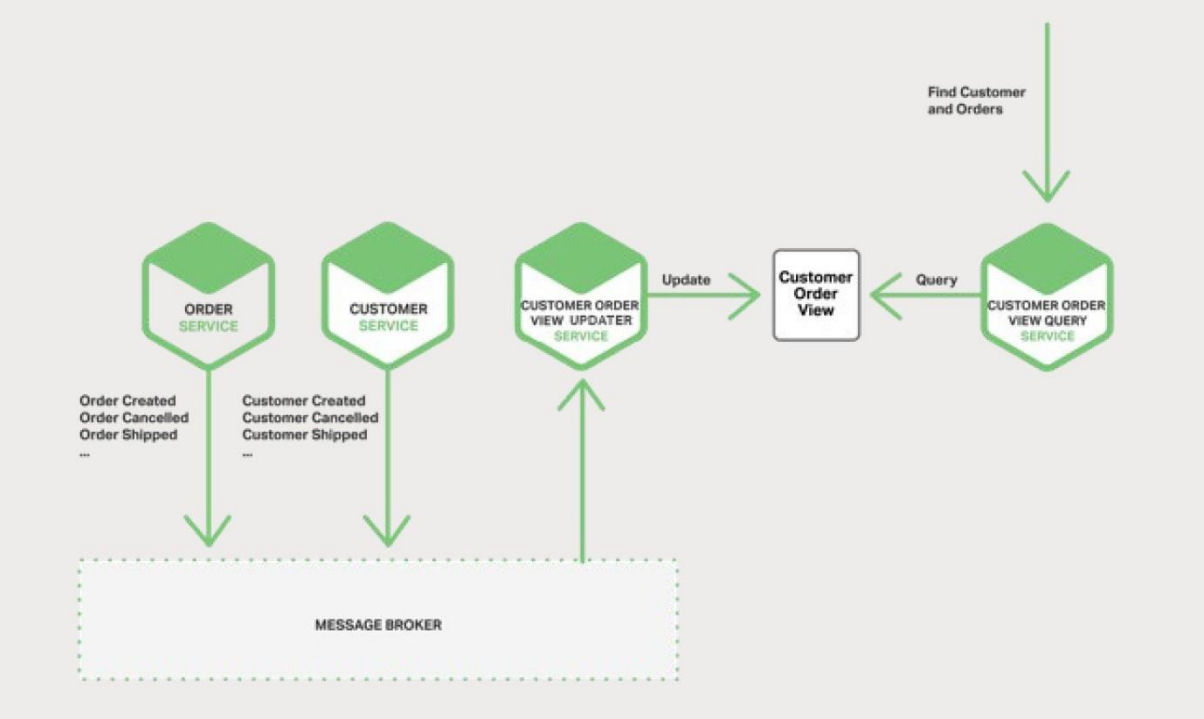


图 5-5、Customer Order View 被两个服务访问

当 Customer Order View Updater Service 接收到 Customer 或 Order 事件时，它会更新 Customer Order View 数据存储。您可以使用如 MongoDB 之类的文档数据库实现 Customer Order View，并为每个 Customer 存储一个文档。Customer Order View Query Service （客户订单视图查询服务）通过查询 Customer Order View 数据存储来处理获取一位客户和最近的订单的请求。

事件驱动的架构有几个优点与缺点。它能够实现跨越多服务并提供最终一致性事务。另一个好处是它还使得应用程序能够维护物化视图。

一个缺点是其编程模型比使用 ACID 事务更加复杂。通常，您必须实现补偿事务以从应用程序级别的故障中恢复。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。因为未提交的事务所做的更改是可见的。如果从未更新的物化视图中读取，应用程序依然可以看到不一致性。另一个缺点是订阅者必须要检测和忽略重复的事件。

5.3 实现原子性

在事件驱动架构中，同样存在着原子更新数据库和发布事件相关问题。例如，Order Service 必须在 ORDER 表中插入一行数据，并发布 Order Created 事件。这两个操作必须原子完成。如果在更新数据库后但在发布事件之前发生服务崩溃，系统将出现

不一致性。确保原子性的标准方法是使用涉及到数据库和 Message Broker 的分布式事务。然而，由于上述原因，如 CAP 定理，这并不是我们想做的。

5.4 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。诀窍在于存储业务实体状态的数据库中有一个用作消息队列的 EVENT 表。应用程序开启一个（本地）数据库事务，更新业务实体状态，将事件插入到 EVENT 表中，之后提交事务。一个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。设计如图 5-6 所示。

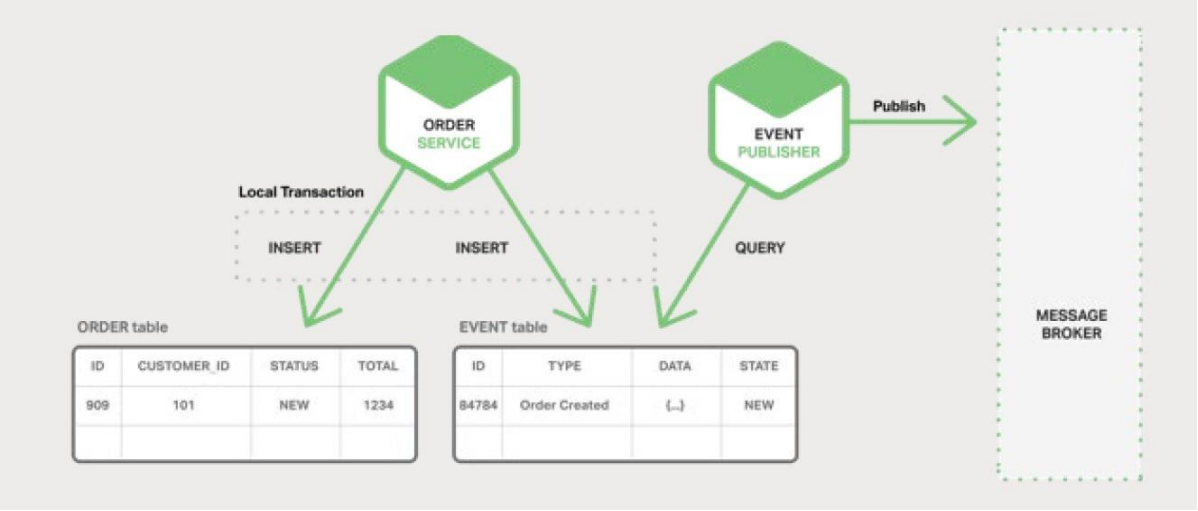


图 5-6、本地事务实现原子性

Order Service 将一行记录插入到 ORDER 表中，并将一个 Order Created 事件插入到 EVENT 表中。

Event Publisher（事件发布者）线程或进程从 EVENT 表中查询未发布的事件，之后发布这些事件，最后更新 EVENT 表以将事件标记为已发布。

这种方法有好有坏。好处是它保证了被发布的事件每次更新都不依赖于 2PC。此外，应用程序发布业务级事件，这些事件可以消除推断的需要。这种方法的缺点是它很容易出错，因为开发人员必须要记得发布事件。这种方法的局限性在于，由于其有限的事务和查询功能，在使用某些 NoSQL 数据库时，实现起来将是一大挑战。

该方法通过让应用程序使用本地事务更新状态和发布事件来消除对 2PC 的依赖。现在我们来看一下通过应用程序简单地更新状态来实现原子性的方法。

5.5 挖掘数据库事务日志

不依靠 2PC 来实现原子性的另一种方式是使用线程或进程发布事件，该线程或进程对数据库的事务或者提交日志进行挖掘。当应用程序更新数据库时，更改信息被记录到数据库的事务日志中。Transaction Log Miner 线程或进程读取事务日志并向 Message Broker 发布事件。设计如图 5-7 所示。

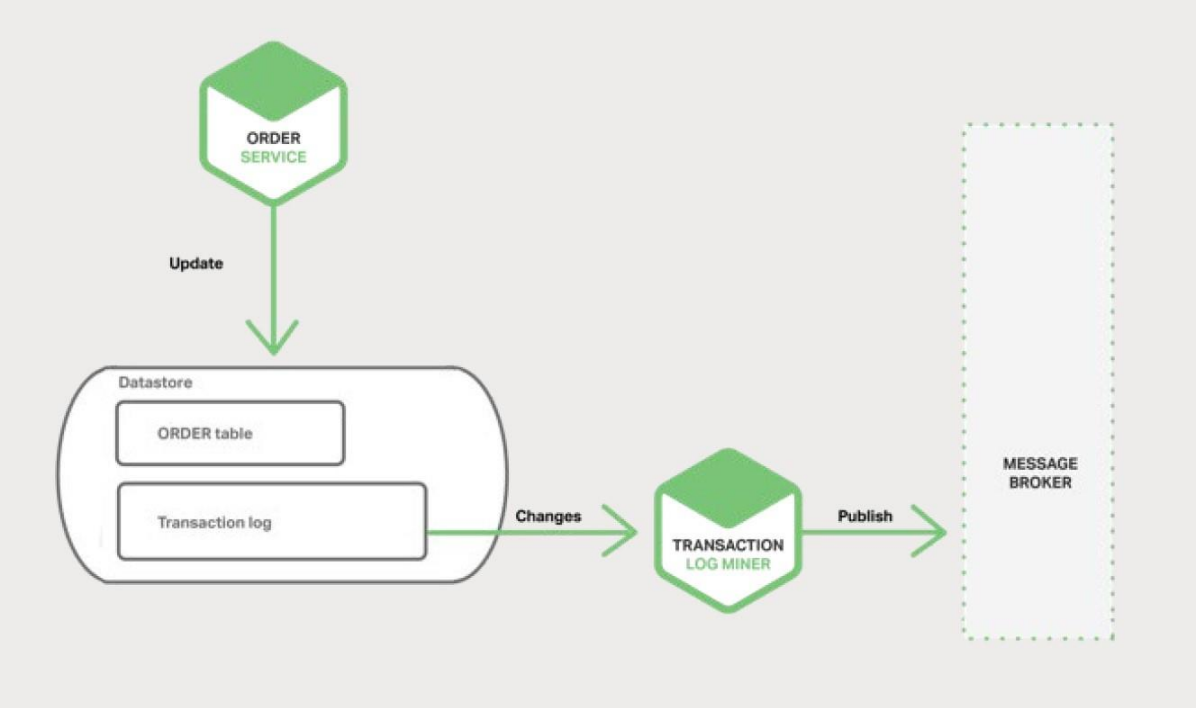


图 5-7、Message Broker 可以公断数据事务

使用这种方法的一个示例是 LinkedIn Databus 开源项目。Databus 挖掘 Oracle 事务日志并发布与更改相对应的事件。LinkedIn 使用 Databus 保持与记录系统一致的各种派生数据存储。

另一个例子是 AWS DynamoDB 中的流机制，它是一个托管的 NoSQL 数据库。DynamoDB 流包含了在过去 24 小时内对 DynamoDB 表中的项进行的更改（创建、更新和删除操作），其按时间顺序排列。应用程序可以从流中读取这些更改，比如，将其作为事件发布。

事务日志挖掘有各种好处与坏处。一个好处是它能保证被发布的事件每次更新都不依赖于 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。一个主要的缺点是事务日志的格式对于每个数据库来说都是专有的，甚至在数据库版本之间格式就发生了改变。而且，记录于事务日志中的低级别更新可能难以对高级业务事件进行逆向工程。

事务日志挖掘消除了应用程序在做一件事时对 2PC 的依赖：更新数据库。现在我们来看看另一种可以消除更新并仅依赖于事件的不同方式。

5.6 使用事件溯源

事件溯源通过使用完全不同的、不间断的方式来持久化业务实体，实现无 2PC 原子性。应用程序不存储实体的当前状态，而是存储一系列状态改变事件。该应用程序通过回放事件来重建实体的当前状态。无论业务实体的状态何时发生变化，其都会将新事件追加到事件列表中。由于保存事件是一个单一操作，因此具有原子性。

要了解事件溯源的工作原理，以 Order（订单）实体为例。在传统方式中，每个订单都与 ORDER 表中的某行记录相映射，也可以映射到例如 ORDER_LINE_ITEM 表中的记录。

但当使用事件溯源时，Order Service 将以状态更改事件的形式存储 Order：Created（创建）、Approved（批准）、Shipped（发货）、Cancelled（取消）。每个事件包含足够的数据来重建 Order 的状态。

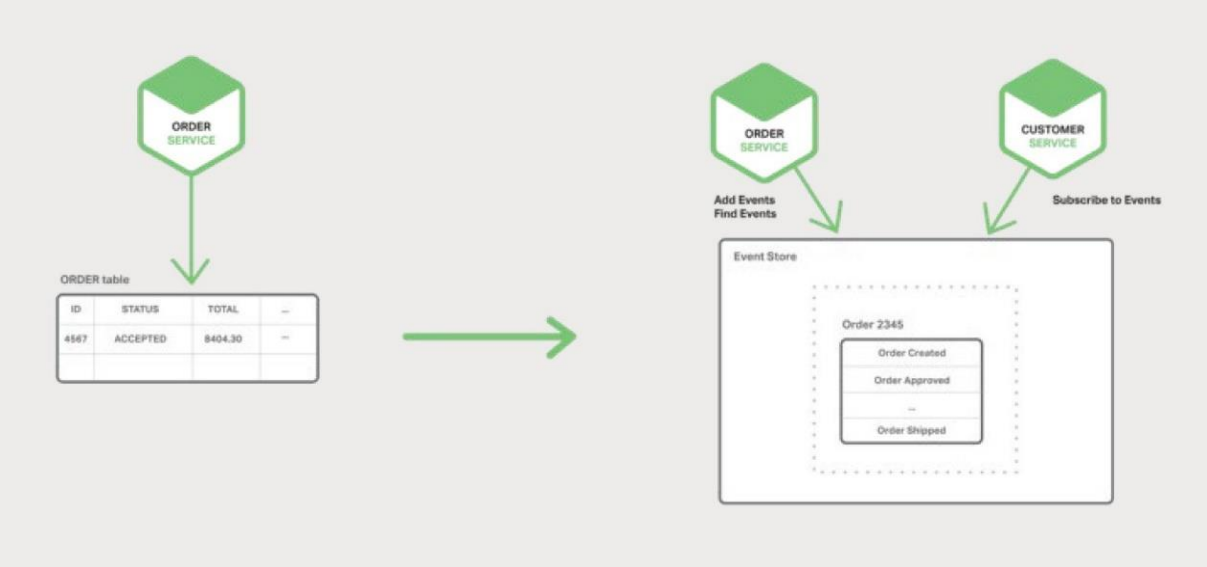


图 5-8、事件有完整的数据

事件被持久化在事件存储中，事件存储是一个事件数据库。该存储有一个用于添加和检索实体事件的 API。

事件存储还与我们之前描述的架构中的 Message Broker 类似。它提供了一个 API，使得服务能够订阅事件。事件存储向所有感兴趣的订阅者派发所有事件。可以说事件存储是事件驱动微服务架构的支柱。

事件溯源有几个好处。它解决了实现事件驱动架构的关键问题之一，可以在状态发生变化时可靠地发布事件。因此，它解决了微服务架构中的数据一致性问题。此外，由于它持久化的是事件，而不是领域对象，所以它主要避免了对象关系阻抗失配问题。事件溯源还提供了对业务实体所做更改的 100% 可靠的审计日志，可以实现在任何时间点对实体进行时间查询以确定状态。事件溯源的另一个主要好处是您的业务逻辑包括松耦合的交换事件业务实体，这使得从单体应用程序迁移到微服务架构将变得更加容易。

事件溯源同样有缺点。这是一种不同而陌生的编程风格，因此存在学习曲线。事件存储仅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。

5.7 总结

在微服务架构中，每个微服务都有自己私有的数据存储。不同的微服务可能会使用不同的 SQL 或者 NoSQL 数据库。虽然这种数据库架构具有明显的优势，但它创造了一些分布式数据管理挑战。第一个挑战是如何实现维护多个服务间的业务事务一致性。第二个挑战是如何实现从多个服务中检索数据。

大部分应用使用的解决方案是事件驱动架构。实现事件驱动架构的一个挑战是如何以原子的方式更新状态以及如何发布事件。有几种方法可以实现这点，包括了将数据库作为消息队列、事务日志挖掘和事件溯源。

微服务实战：NGINX 与存储优化

by Floyd Smith

基于微服务的存储方法涉及大数量和各种数据存储，访问和更新数据将变得更加复杂，DevOps 在维护数据一致性方面面临着更大的挑战。NGINX 为这种数据管理提供了重要支持，主要有三个方面：

1. **数据缓存与微缓存（microcaching）**
   使用 NGINX 缓存静态文件和微缓存应用程序生成的内容可减轻应用程序的负载、提高性能并减少问题的发生。

2. **数据存储的灵活性与可扩展性**
   一旦将 NGINX 作为反向代理服务器，您的应用程序在创建、调整大小、运行和调整数据存储服务器的大小时可获得很大的灵活性，以满足不断变化的需求 — 每个服务都拥有自己的数据存储是很重要的。

3. **服务监控与管理，包括数据服务**
   随着数据服务器数量的增加，支持复杂操作和具有监控和管理工具显得非常重要。NGINX Plus 内置了这些工具和应用程序性能管理合作伙伴的接口，如 Data Dog、Dynatrace 和 New Relic。

微服务相关的数据管理示例可在 NGINX 微服务参考架构的三大模型中找到，其为您设计决策和实施提供了起点。

6 选择部署策略

本书内容主要介绍如何使用微服务构建应用程序，这是本书的第六章。第一章介绍了微服务架构模式，讨论了使用微服务的优点与缺点。之后的章节讨论了微服务架构的方方面面：使用 API 网关 、进程间通信、服务发现和事件驱动数据管理。在本章中，我们将介绍部署微服务的策略。

6.1 动机

部署单体应用程序意味着运行一个或多个相同副本的单个较大的应用程序。您通常会在每台服务器上配置 N 个服务器（物理或虚拟）并运行 M 个应用程序实例。单体应用程序的部署并不总是非常简单，但它比部署微服务应用程序要简单得多。

微服务应用程序由数十甚至上百个服务组成。服务以不同的语言和框架编写。每个都是一个迷你的应用程序，具有自己特定的部署、资源、扩展和监视要求。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相应的 CPU、内存和 I/O 资源。更具挑战性的是尽管如此复杂，部署服务也必须快速、可靠和具有成本效益。

有几种不同的微服务部署模式。我们首先看看单主机多服务实例模式。

6.2 单主机多服务实例模式

部署微服务的一种方式是使用单主机多服务实例（Multiple Service Instances per Host）模式。当使用此模式时，您可以提供一个或多个物理主机或虚拟主机，并在每个上运行多个服务实例。从多方面来讲，这是应用程序部署的传统方式。每个服务实例在一个或多个主机的标准端口上运行。主机通常被当作宠物对待。

图 6-1 展示了该模式的结构：

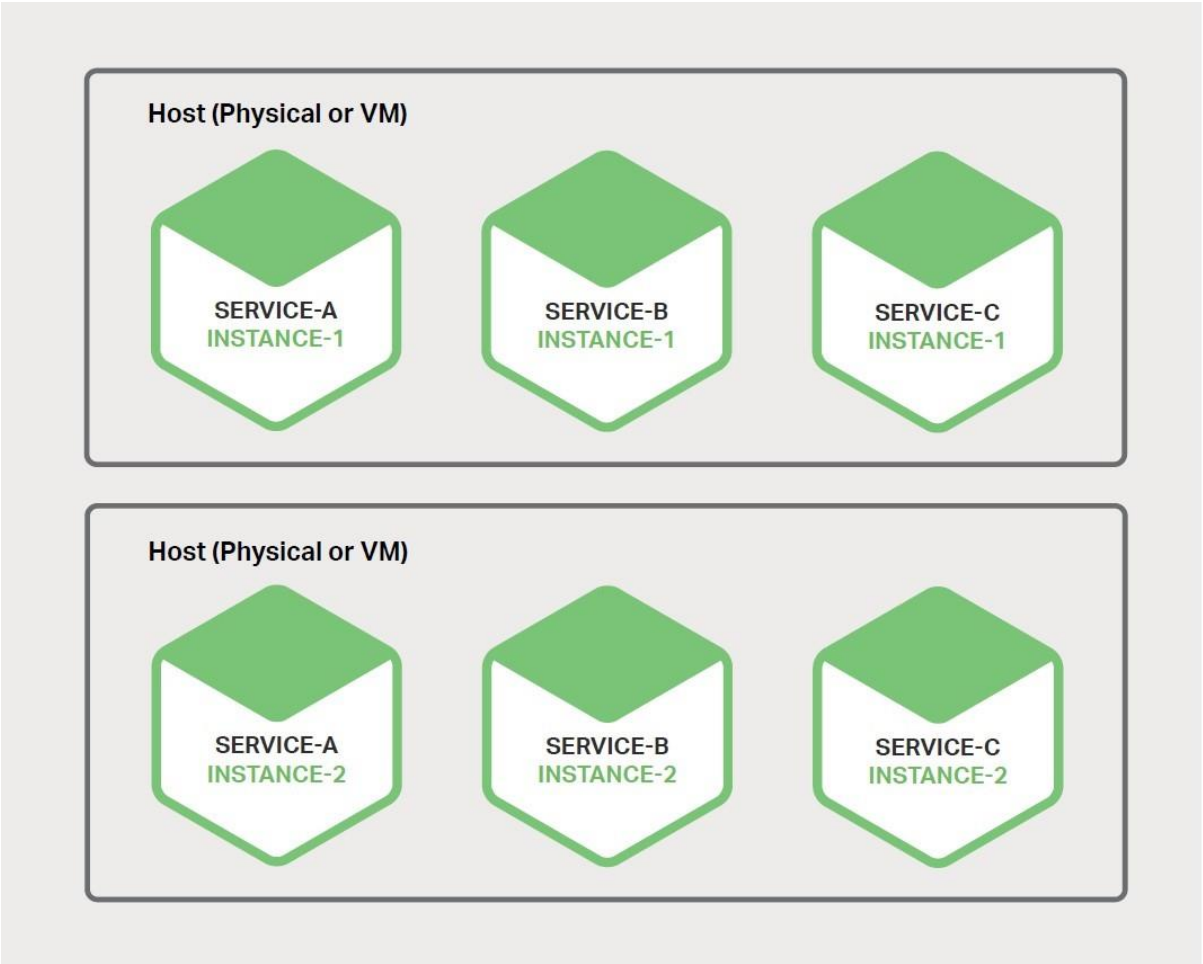


图 6-1、主机可支持多个服务实例

这种模式有几个变体。一个变体是每个服务实例都是一个进程或进程组。例如，您可以在 Apache Tomcat 服务器上将 Java 服务实例部署为 Web 应用程序。一个 Node.js 服务实例可能包含一个父进程和一个或多个子进程。

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，您可以在同一个 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 软件包。

单主机多服务实例模式有优点也有缺点。主要优点是其资源使用率相对较高。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了多个服务实例（例如，共享相同的 Apache Tomcat 服务器和 JVM 的多个 Web 应用程序），则效率更高。

这种模式的另一个优点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。如果服务是使用 Java 编写的，则可以复制 JAR 或 WAR 文件。对于其他语言，例如 Node.js 或 Ruby，您可以直接复制源代码。在任一情况下，通过网络复制的字节数都是相对较小的。

另外，由于缺乏开销，通常启动一个服务是非常快的。如果该服务是自己的进程，您只需要启动它即可。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者重新启动容器。尽管这很有吸引力，但单主机多服务

实例模式有一些明显的缺点。一个主要的缺点是服务实例很少或者没有隔离，除非每个服务实例是一个单独的进程。虽然您可以准确地监视每个服务实例的资源利用率，但是您不能限制每个实例使用的资源。一个行为不当的服务实例可能会占用掉主机的所有内存或 CPU。

如果多个服务实例在同一进程中运行，那么将毫无隔离可言。例如，所有实例可能共享相同的 JVM 堆。行为不当的服务实例可能会轻易地破坏在同一进程中运行的其他服务。此外，您无法监控每个服务实例使用的资源。

这种方式的另一个重要问题是部署服务的运维团队必须了解执行此操作的具体细节。服务可以用多种语言和框架编写，因此开发团队必须与运维交代许多细节。这种复杂性无疑加大了部署过程中的错误风险。

正如您所见，尽管这种方式简单，但单主机多服务实例模式确实存在一些明显的缺点。现在让我们来看看可以绕过这些问题部署微服务的其他方式。

6.3 每个主机一个服务实例模式

部署微服务的另一种方式是使用每个主机一个服务实例（Service Instance per Host）模式。当使用此模式时，您可以在主机上单独运行每个服务实例。这种模式有两种不同形式：每个虚拟机一个服务实例模式和每个容器一个服务实例模式。

6.3.1 每个虚拟机一个服务实例模式

当您使用每个虚拟机一个服务实例模式时，将每个服务打包为一个虚拟机（VM）镜像（如 Amazon EC2 AMI）。每个服务实例都是一个使用该 VM 镜像启动的 VM（例如，一个 EC2 实例）。

图 6-2 展示了该模式的结构：

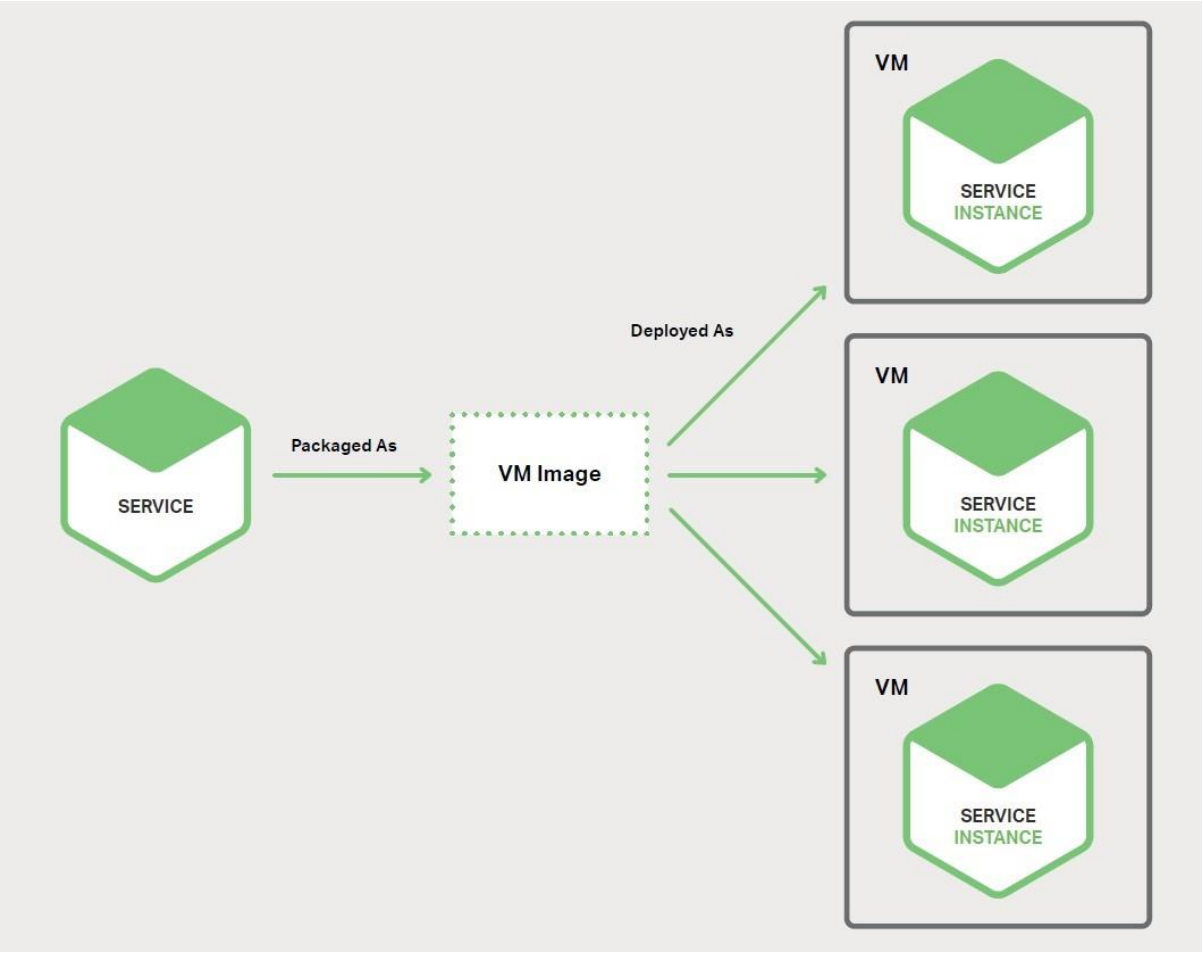


图 6-2、服务可以各自运行在自己的虚拟机中

这是 Netflix 部署其视频流服务的主要方式。Netflix 使用 Aminator 将每个服务打包为 EC2 AMI。每个运行的服务实例都是一个 EC2 实例。

您可以使用多种工具来构建自己的虚拟机。您可以配置您的持续集成（CI）服务器（比如 Jenkins）来调用 Aminator 将服务打包为一个 EC2 AMI。Packer 是自动化虚拟机镜像创建的另一个选择。与 Aminator 不同，它支持各种虚拟化技术，包括 EC2、DigitalOcean、VirtualBox 和 VMware。

Boxfuse 公司有一种非常棒的方式用来构建虚拟机镜像，其克服了我将在下面描述的虚拟机的缺点。Boxfuse 将您的 Java 应用程序打包成一个最小化的 VM 镜像。这些镜像可被快速构建、快速启动且更加安全，因为它们暴露了一个有限的攻击面。

CloudNative 公司拥有 Bakery，这是一种用于创建 EC2 AMI 的 SaaS 产品。您可以配置您的 CI 服务器，以在微服务通过测试后调用 Bakery。之后 Bakery 将您的服务打包成一个 AMI。使用一个如 Bakery 的 SaaS 产品意味着您不必浪费宝贵的时间来设置 AMI 创建基础架构。

每个虚拟机一个服务实例模式有许多优点。VM 的主要优点是每个服务实例运行是完全隔离的。它有固定数量的 CPU 和内存，且不能从其他服务窃取资源。

将微服务部署为虚拟机的另一个优点是可以利用成熟的云基础架构。如 AWS 之类的云提供了有用的功能，例如负载平衡和自动扩缩。将服务部署为虚拟机的另一个好处是它封装了服务的实现技术。一旦服务被打包成一个虚拟机，它就成为一个黑匣子。VM 的管理 API 成为部署服务的 API。部署变得更加简单、可靠。

然而，每个虚拟机一个服务实例模式也有一些缺点。一个缺点是资源利用率较低。每个服务实例都有一整个 VM 开销，包括操作系统。此外，在一个典型的公共 IaaS 中，VM 具有固定大小，并且 VM 可能未被充分利用。

此外，公共 IaaS 中的 VM 通常是收费的，无论它们是处于繁忙还是空闲。如 AWS 之类的 IaaS 虽然提供了自动扩缩功能，但很难快速响应需求变化。因此，您经常需要过度配置 VM，从而增加部署成本。

这种方法的另一缺点是部署新版本的服务时通常很慢。由于大小原因，通常 VM 镜像构建很慢。此外，VM 实例化也很慢，同样是因为它们的大小。而且，操作系统也需要一些时间来启动。但请注意，这并不普遍，因为已经存在由 Boxfuse 构建的轻量级 VM。

每个虚拟机一个服务实例模式的另一个缺点是通常您（或组织中的其他人）要对很多未划分的重担负责。除非您使用 Boxfuse 这样的工具来处理构建和管理虚拟机的开销，否则这将是您的责任。这个必要而又耗时的活动会分散您的核心业务。

接下来让我们看看另一种部署更轻量级微服务的替代方式，它也有许多与虚拟机一样的优势。

6.3.2 每个容器一个服务实例模式

当您使用每个容器一个服务实例模式（Service Instance per Container）模式时，每个服务实例都在其自己的容器中运行。容器是一个操作系统级虚拟化机制。一个容器是由一个或多个运行在沙箱中的进程组成。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。容器技术的相关例子有 Docker 和 Solaris Zones。

图 6-3 展示了该模式的结构：

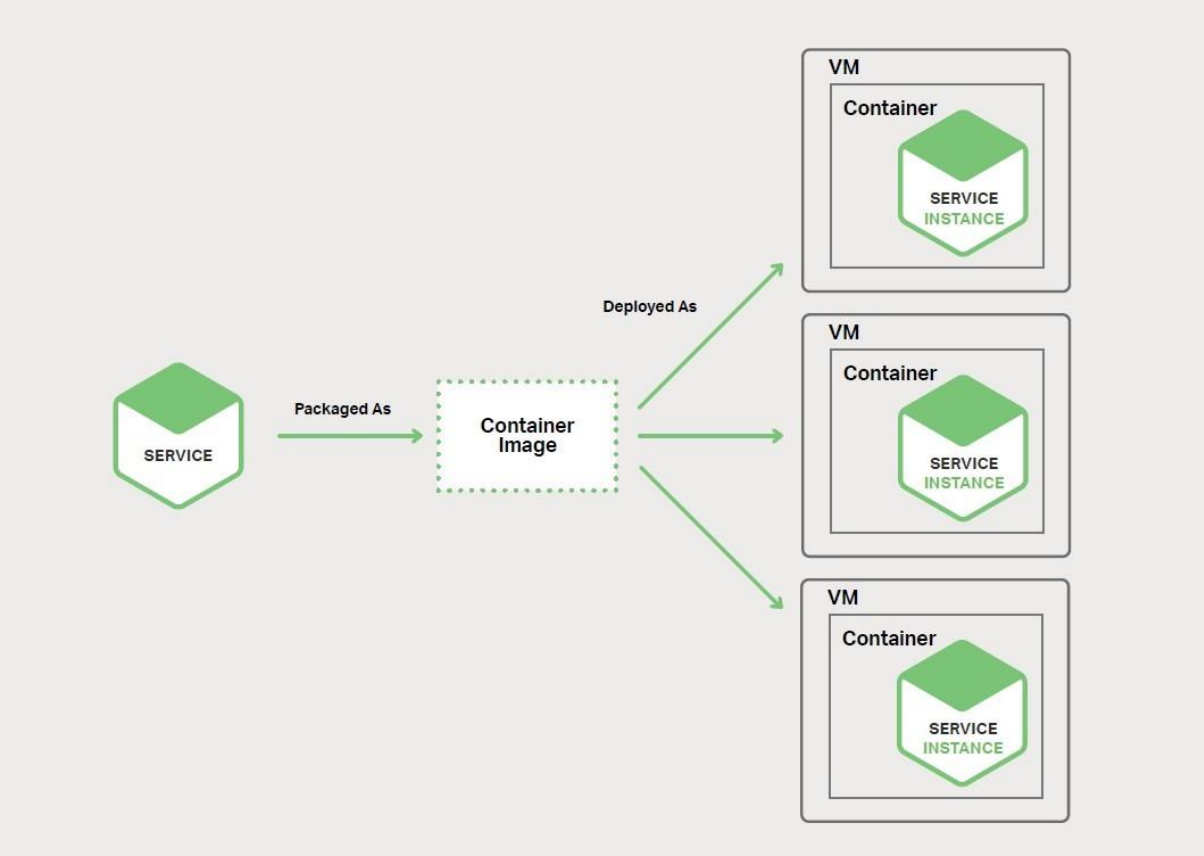


图 6-3、服务可以各自运行在自己的容器中

要使用此模式，请将您的服务打包成一个容器镜像。容器镜像是由运行服务所需的应用程序和库组成的文件系统镜像。一些容器镜像由完整的 Linux 根文件系统组成。此外它更加轻便。例如，要部署一个 Java 服务，您可以构建一个包含了 Java 运行时的容器镜像，可能是一个 Apache Tomcat 服务器和编译好的 Java 应用程序。

将服务打包成一个容器镜像后，您将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工具（如 Kubernetes 或 Marathon）来管理容器。集群管理工具将主机视为一个资源池。它根据容器所需的资源和每个主机上可用的资源来决定每个容器放置的位置。

每个容器一个服务实例模式模式有优点和缺点。容器的优点与虚拟机的类似。它们将服务实例彼此隔离。您可以轻松地监控每个容器所消耗的资源。此外，与 VM 一样，容器封装了服务实现技术。容器管理 API 作为管理您的服务的 API。

然而，与虚拟机不同，容器是轻量级技术。容器镜像通常可以非常快速地构建。例如，在我的笔记本电脑上，将一个 Spring Boot 应用程序打包成一个 Docker 容器只需要 5 秒钟的时间。容器也可以很快地启动，因为没有繁琐的操作系统引导机制。当一个容器启动时，它所运行的就是服务。

使用容器有一些缺点。虽然容器基础架构正在快速发展走向成熟，但它并不像虚拟机的基础架构那么成熟。此外，容器不像 VM 那样安全，因为容器彼此共享了主机的 OS 内核。

容器的另一个缺点是您需要负责未划分的容器镜像管理重担。此外，除非您使用了托管容器解决方案 [如 Google Container Engine 或 Amazon EC2 Container Service（ECS）]，否则您必须自己管理容器基础设施以及可能运行的 VM 基础架构。

此外，容器通常部署在一个按单个 VM 收费的基础设施上。因此，如之前所述，可能会产生超额配置 VM 的额外成本，以处理负载峰值。

有趣的是，容器和 VM 之间的区别可能会有些模糊。如之前所述，Boxfuse VM 可以很快地构建和启动。Clear Containers 项目旨在创建轻量级虚拟机。Unikernels 也正在蓬勃发展。Docker 公司于 2016 年初收购了 Unikernel 系统。

还有一个日益流行的 server-less（无服务器）部署概念，这是一种避免了“在容器中还是在虚拟机中部署服务”问题的方法。接下来我们来看看。

6.4 Serverless 部署

AWS Lambda 就是一个 serverless 部署技术示例。它支持 Java、Node.js 和 Python 服务。要部署微服务，请将其打包成 ZIP 文件并将上传到 AWS Lambda。您还要提供元数据，其中包括了被调用来处理请求（又称为事件）的函数的名称。AWS Lambda 自动运行足够的微服务服务实例来处理请求。您只需根据每个请求所用时间和内存消耗来付费。当然，问题往往出现在细节上，您很快注意到了 AWS Lambda 的局限性。但是，作为开发人员的您或组织中的任何人都无需担心服务器、虚拟机或容器的任何方面 ，这非常有吸引力，足以令人难以置信。

Lambda 函数是无状态服务。它通常通过调用 AWS 服务来处理请求。例如，当图片上传到 S3 存储桶时 Lambda 函数将被调用，可插入一条记录到 DynamoDB 图片表中，并将消息发布到 Kinesis 流以触发图片处理。 Lambda 函数还可以调用第三方 Web 服务。

有四种方法调用 Lambda 函数：

- 直接使用 Web 服务请求
- 自动响应一个 AWS 服务（如 S3、DynamoDB、Kinesis 或 Simple Email Service）生成的事件
- 通过 AWS API 网关自动处理来自应用程序客户端的 HTTP 请求
- 按照一个类似 cron 的时间表，定期执行

正如您所见，AWS Lambda 是一个便捷的微服务部署方式。基于请求的定价意味着您只需为服务实际执行的工作支付。另外，由于您不需要对 IT 基础架构负任何责任，因此可以专注于开发应用程序。

然而，其也存在一些明显的局限性。Lambda 函数不适用于部署长时间运行的服务，例如消耗第三方消息代理消息的服务。请求必须在 300 秒内完成。服务必须是无状态的，

因为理论上，AWS Lambda 可能为每个请求运行一个单独的实例。他们必须使用受支持的语言之一来编写。服务也必须快速启动，否则，他们可能会因超时而终止。

6.5 总结

部署微服务应用程序充满着挑战。您可能有数个甚至数百个使用了各种语言和框架编写的服务。每个应用程序都是一个迷你应用程序，有自己特定的部署、资源、扩展和监视需求。有几个微服务部署模式，包括每个虚拟机一个服务实例和每个容器一个服务实例模式。部署微服务的另一个有趣的选择是 AWS Lambda，一种 serverless 方式。在本书的下一章也是最后一章中，我们将介绍如何将单体应用程序迁移到微服务架构。

微服务实战：使用 NGINX 在不同主机上部署微服务

by Floyd Smith

NGINX 对于各种类型的部署具有许多优势 – 无论是单体应用程序、微服务应用程序还是混合应用程序（将在下一章介绍）。使用 NGINX，您可以智能抽取不同的部署环境出来并整合入 NGINX。如果您使用针对不同部署环境的工具，则有许多应用程序功能以不同的方式工作，但如果使用 NGINX，那么在所有环 境中都可以使用相同的方式进行工作。

这一特性也为 NGINX 和 NGINX Plus 带来了第二个优势：通过在多个部署环境中同时运行应用程序来扩展应用程序的能力。假设您拥有和管理着的本地服务器，但是您的应用程序使用情况正在增长，并且预计将超出这些服务器可以处理的峰值。如果您已经使用了 NGINX，您就有了一个强大的选择：扩展到云端 – 例如，扩展到 AWS 上，而不是购买、配置和保持额外的服务器来为了以防万一。也就是说，当您的本地服务器上的流量达到容量限制时，可根据需要在云中启动其他微服务实例来处理。

这只是因使用 NGINX 变得更加灵活的一个例子。维护单独的测试和部署环境、切换环境的基础设施、以及管理各种环境中的应用程序组合都变得更加现实和可实现。

NGINX 微服务参考架构被明确设计为支持这种灵活部署，其假设在开发和部署期间使用容器技术。如果您还没尝试，可以考虑转移到容器、NGINX 或 NGINX Plus，以轻松地转移到微型服务，以及使您的应用程序、开发和部署灵活性以及人员更具前瞻性。

7 重构单体为微服务

本书主要介绍如何使用微服务构建应用程序，这是本书的第七章，也是最后一章。第一章介绍了微服务架构模式，讨论了使用微服务的优点与缺点。随后的章节讨论了微服务架构的方方面面：使用 API 网关 、进程间通信、服务发现、事件驱动数据管理和部署微服务。在本章中，我们将介绍单体应用迁移到微服务的策略。

我希望这本电子书能够让您对微服务架构、其优点和缺点以及何时使用它有很好的了解。微服务架构也许很适合您的组织。

您正工作于大型复杂的单体应用程序上，这是相当不错的机会。然而，您开发和部署应用程序的日常经历是缓慢而痛苦的。微服务似乎是一个遥不可及的天堂。幸运的是，有一些战略可以用来逃离单体地狱。在本文中，我将描述如何将单体应用程序逐渐重构为一组微服务。

7.1 微服务重构概述

单体应用程序转换为微服务的过程是应用程序现代化的一种形式。这是几十年来开发人员一直在做的事情。因此，在将应用程序重构为微服务时，有一些想法是可以重用的。

一个不要使用的策略是“大爆炸”重写。就是您将所有的开发工作都集中在从头开始构建新的基于微服务的应用程序。虽然这听起来很吸引人，但非常危险，有可能会失败。据 AsMartin Fowler 讲到：**“大爆炸重写的唯一保证就是大爆炸！”**（"the only thing a Big Bang rewrite guarantees is a Big Bang!"）。

您应该逐步重构单体应用程序，而不是通过大爆炸重写。您可以逐渐添加新功能，并以微服务的形式创建现有功能的扩展 — 以互补的形式修改单体应用，并且一同运行微服务和修改后的单体。随着时间推移，单体应用程序实现的功能量会缩小，直到它完全消失或变成另一个微服务。这种策略类似于在 70 公里/小时的高速公路上驾驶一辆汽车，很具挑战性，但比尝试大爆炸改写的风险要小得多。

葡萄树

Martin Fowler 将这种应用现代化策略称为杀手应用（Strangler Application）。这个名字来自发现于热带雨林中的葡萄树（也称为绞杀榕）。一棵葡萄树生长在一棵树上，以获取森林冠层之上的阳光。有时，树死了，留下一个树形的腾。应用现代化也遵循相同的模式。我们将构建一个新的应用程序，包括了围绕遗留应用的微服务（它将会慢慢缩小或者最终消亡）。

让我们来看看能做到这点的不同策略。

7.2 策略一：停止挖掘

洞穴定律说到，每当您身处在一个洞穴中，您应该停止挖掘。当您的单体应用变得难以管理时，这是一个很好的建议。换句话说，您应该停止扩张，避免使单体变得更大。这意味着当您要实现新功能时，您不应该向单体添加更多的代码。相反，这一策略的主要思想是将新代码放在独立的微服务中。

应用此方法后，系统架构如图 7-1 所示。

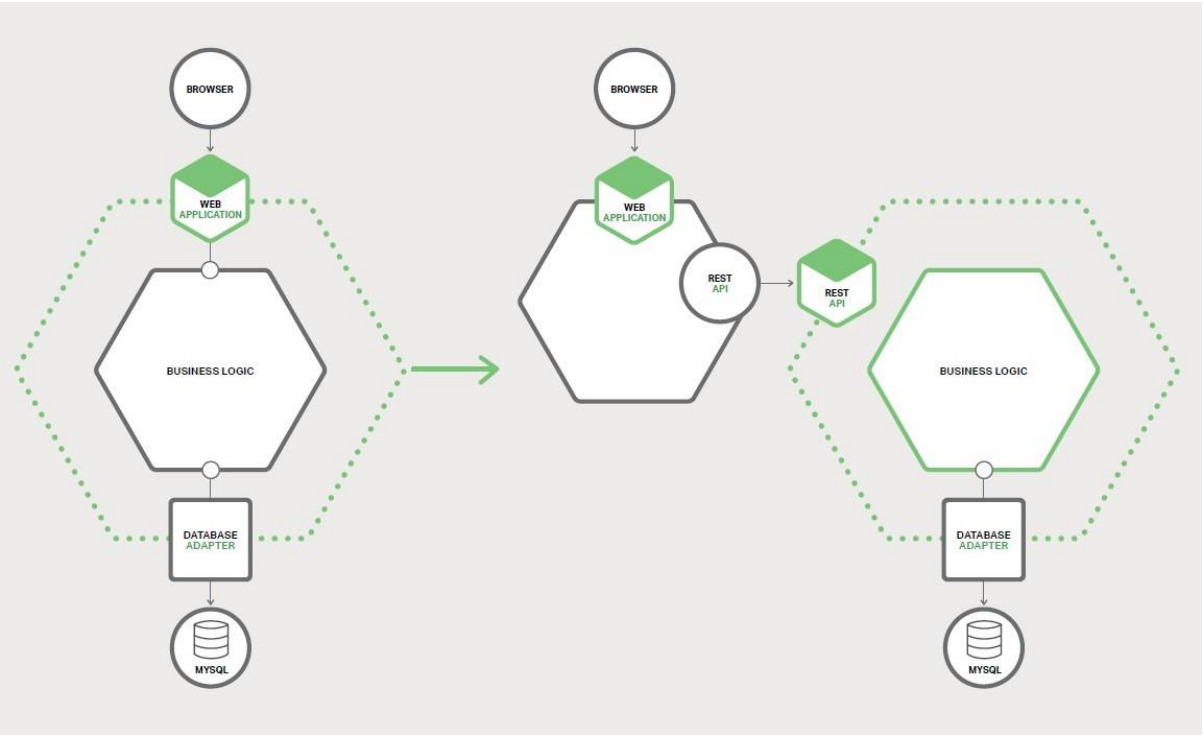


图 7-1、将新功能实现为单独的服务，而不是将模块添加到单体

除了新服务和传统的单体，还有另外两个组件。第一个是请求路由，它处理传入的（HTTP）请求，类似于第二章中描述的 API 网关。路由向新服务发送与新功能相对应的请求。它将遗留的请求路由到单体。

另一个组件是粘合代码，它将服务与单体集成。一个服务很少孤立存在，通常需要访问单体的数据。位于单体、服务或两者中的粘合代码负责数据集成。该服务使用粘合代码来读取和写入单体数据。

服务可以使用三种策略来访问单体数据：

- 调用由单体提供的远程 API
- 直接访问单体数据库
- 维护自己的数据副本，与单体数据库同步

粘合代码有时被称为防护层（anti-corruption layer）。这是因为粘合代码阻止了服务被遗留的单体领域模型的概念所污染，这些服务具有自己的原始领域模型。粘合代码在两种不同的模型之间转换。防护层一词首先出现于埃里克·埃文斯（Eric Evans）所著的必读图书《领域驱动设计》（Domain Driven Design）中，并在白皮书中进行了改进。开发一个防护层并不是一件简单的事情。但是，如果您想要从单体地狱中走出来，这是必不可少的步骤。

使用轻量级服务来实现新功能有几个好处。它防止单体变得更加难以管理。该服务可以独立于单体开发、部署和扩展。可让您创建的每个新服务体验到微服务架构的优势。

然而，这种方法没有解决单体问题。要解决这些问题，您需要分解单体。让我们来看看这样做的策略。

7.3 策略二：前后端分离

缩小单体应用的一个策略是从业务逻辑层和数据访问层拆分出表现层。一个典型的企业应用由至少三种不同类型的组件组成：

- **表现层（Presentation Layer，PL）**
  处理 HTTP 请求并实现（REST）API 或基于 HTML 的 Web UI 组件。在具有复杂用户界面的应用中，表现层通常存在大量代码。

- **业务逻辑层（Business Logic Layer，BLL）**
  作为应用程序核心，实现业务规则的组件。

- **数据访问层（Data Access Layer，DAL）**
  访问基础架构组件的组件，如数据库和消息代理。

一方的表现逻辑和另一方的业务和数据访问逻辑之间通常有一个完全的隔离。业务层具有由一个或多个门面组成的粗粒度 API，其封装了业务逻辑组件。这个 API 是一个天然的边界，您可以沿着该边界将单体拆分成两个较小的应用程序。一个应用程序包含表现层。另一个应用程序包含业务和数据访问逻辑。分割后，表现逻辑应用程序对业务逻辑应用程序进行远程调用。

重构之前和之后的架构如图 7-2 所示。

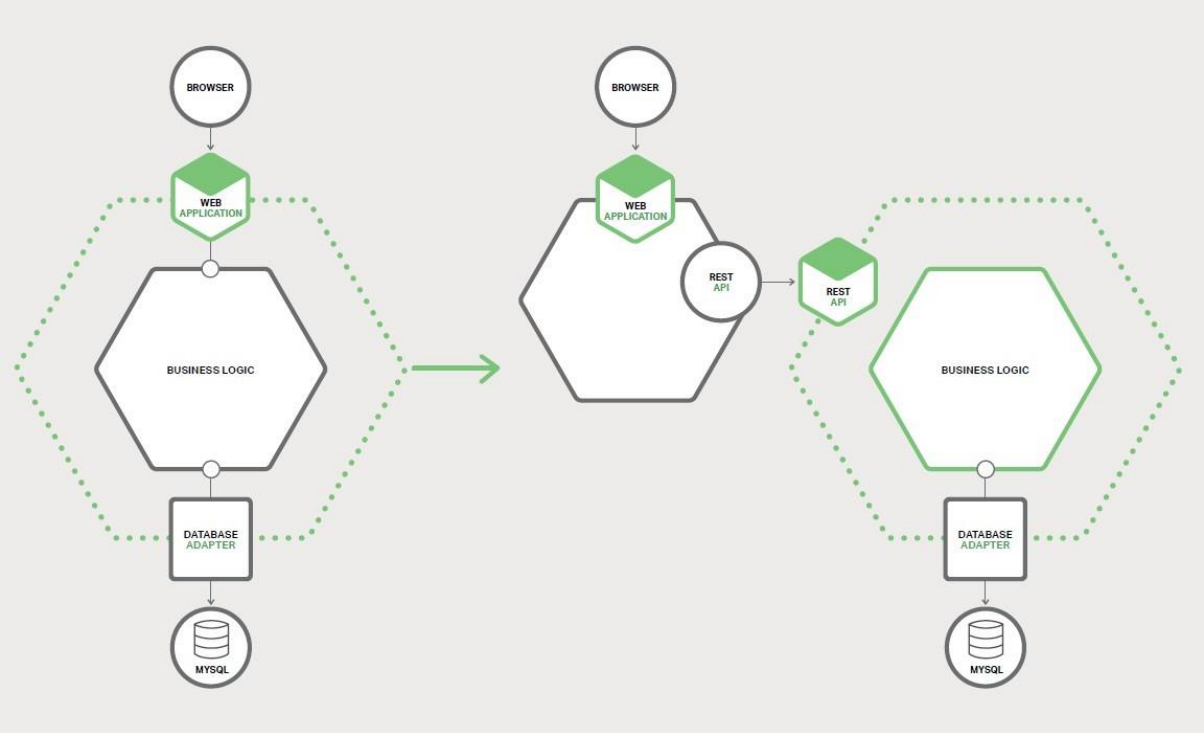


图 7-2、重构现有的应用程序

以这种方式拆分单体有两个主要优点。它使您能够独立于彼此开发、部署和扩展这两个应用。特别是它允许表现层开发人员在用户界面上快速迭代，并且可以轻松执行 A/B 测试。这种方法的另一个优点是它暴露了可以被您开发的微服务调用的远程 API。

然而，这一策略只是一个局部解决方案。两个应用程序中的一个或两个很可能是一个无法管理的单体。您需要使用第三种策略来消除剩余的整体或单体。

7.4 策略三：提取服务

第三个重构策略是将庞大的现有模块转变为独立的微服务。每次提取一个模块并将其转换成服务时，单体就会缩小。一旦您转换了足够的模块，单体将不再是一个问题。或者它完全消失，或者变得足够小，它就可以被当做一个服务看待。

7.4.1 优先将哪些模块转换为微服务

一个庞大而复杂的单体应用由几十个或几百个模块组成，所有模块都是提取的候选项。弄清楚要先转换哪些模块往往存在一定的挑战。一个好的方法是从容易提取的几个模块开始。您将得到微服务的相关经验，特别是在提取过程方面。之后，您应该提取那些能给您最大利益的模块。

将模块转换为服务通常是耗时的。您想按照您将获得的利益对模块进行排列。提取频繁更改的模块通常是有益的。一旦将模块转换为服务，您就可以独立于单体开发和部署，这将加快开发工作。

提取这些与单体的其他模块有显著不同的模块也是有益的。例如，将有一个有内存数据库的模块转换为服务是很有用的，这样可以部署在具有大量内存的主机上，无论是裸机服务器、虚拟机还是云实例。同样，提取实现了计算昂贵算法的模块也是值得的，因为该服务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，您可以使应用程序更加容易、廉价地扩展。

当找到要提取的模块时，寻找现有的粗粒度边界（又称为接缝）是有用的。它们使模块转成服务变得更容易和更连廉价。有关这种边界的一个例子是一个仅通过异步消息与应用程序的其他部分进行通信的模块。将该模块转变为微服务相对比较廉价和简单。

7.4.2 如何提取模块

提取模块的第一步是在模块和单体之间定义一个粗粒度的接口。因为单体需要服务拥有的数据，它很可能是一个双向 API，反之亦然。由于模块和应用程序的其余之间存在着复杂的依赖关系和细粒度的交互模式，因此实现这样的 API 通常存在挑战。由于领域模型类之间的众多关联，使用领域模型模式来实现的业务逻辑尤其具有挑战性。您通常需要进行重大的代码更改才能打破这些依赖。图 7-3 展示了重构。

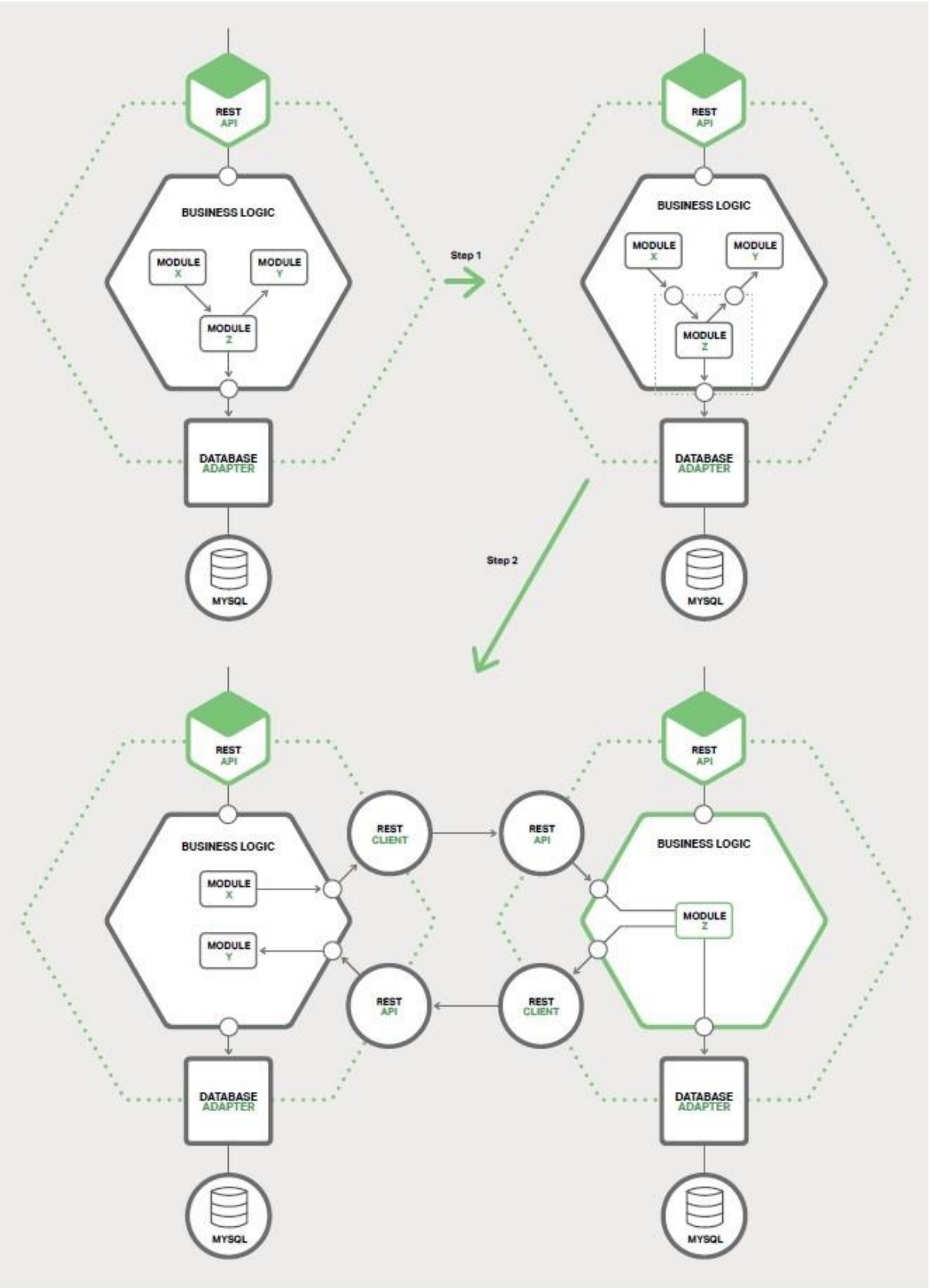


图 7-3、单体模块可转换为微服务

一旦实现了粗粒度的接口，您就可以将模块变成独立的服务。要做到这点，您必须编写代码以使单体和服务通过使用进程间通信（IPC）机制的 API 进行通信。图 7-3 显示了重构前、重构中和重构后的架构。

在此例中，模块 Z 是要提取的候选模块。其组件由模块 X 使用，并且它使用了模块 Y。第一个重构步骤是定义一对粗粒度的 API。第一个接口是一个使用模块 X 来调用模块 Z 的入站接口。第二个接口是一个使用模块 Z 调用模块 Y 的出站接口。

第二个重构步骤是将模块转换为一个独立服务。入站和出站接口使用 IPC 机制的代码来实现。您将很可能需要通过将 Module Z 与 Microservice Chassis 框架相结合来构建服务，该框架负责处理诸如服务发现之类的横切点。

一旦您提取了一个模块，您就可以独立于单体和任何其他服务开发、部署和扩展其他服务。您甚至可以从头开始重写服务。在这种情况下，整合服务与单体的 API 代码成为在两个领域模型之间转换的防护层。每次提取服务时，您都会朝微服务方向迈近一步。随着时间的推移，单体将缩小，您将拥有越来越多的微服务。

7.5 总结

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。您不应该从头开始重写您的应用来迁移到微服务。相反，您应该将应用程序逐渐重构为一组微服务。可以使用这三种策略：将新功能实现为微服务；从业务组件和数据访问组件中分离出表现组件；将单体中的现有模块转换为服务。随着时间推移，微服务的数量将会增长，您的开发团队的灵活性和速度也同样会增加。

微服务实战：用 NGINX 征服单体

by Floyd Smith

如本章所述，将单体转换为微服务可能是一个缓慢而具有挑战性的过程，但这同样具有许多好处。使用 NGINX，您可以在实际开始转换过程之前获得微服务器的一些优势。

您可以通过将 NGINX 放在您现有的单体应用之前，以节省迁移微服务所花费的大量时间。以下简要说明与微服务有关的好处：

- **更好地支持微服务**
  如第五章尾栏所述，NGINX 和 NGINX Plus 具有利于开发基于微服务的应用的功能。当您开始重新设计单体应用时，由于 NGINX 的功能，您的微服务将执行得更好、更易于管理。

- **跨环境的功能抽象**

从您管理的服务器甚至是各种公共云、私有云和混合云上将功能迁移到 NGINX 作为反向代理服务器可以减少部署在新的环境中的设施数量变化。这补充扩展了微服务所固有的灵活性。

- **NGINX 微服务参考架构可用性**
  当您迁移到 NGINX 时，您可以借鉴 NGINX 微服务参考架构（MRA，Microservices Reference Architecture），以便在迁移到微服务之后定义应用程序的最终结构，并根据需要使用的 MRA 部分应用于您创建的每个新的微服务。

总而言之，实使用现 NGINX 作为您转型的第一步，将压倒您的单片应用程序，使其更容易获得微服务的所有优势，并为您提供用于进行转换的模型。您可以了解有关 MRA 的更多信息，并获得 NGINX Plus 的免费试用版。