微服务

架构实战

基于 Spring Boot、Spring Cloud、Docker

郑天民 / 著

结合作者自身在互联网行业实施服务化的经历讲解微服务技术体系与实践

Spring Boot、Spring Cloud和Docker等主流技术构建微服务架构

结合业务场景，通过全面案例给出工具在构建微服务架构中的工程实践

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

微服务架构实战

基于 Spring Boot、Spring Cloud、Docker

郑天民／著

人民邮电出版社
北京

图书在版编目（CIP）数据

微服务架构实战 : 基于Spring Boot、Spring Cloud、Docker / 郑天民著. -- 北京 : 人民邮电出版社, 2018.12
ISBN 978-7-115-49090-2

Ⅰ. ①微… Ⅱ. ①郑… Ⅲ. ①互联网络－网络服务器 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字(2018)第185411号

内 容 提 要

本书主要介绍微服务架构实现过程中所应具备的技术体系。全书围绕实现微服务架构的基础组件和关键要素，讨论了使用 Spring Boot、Spring Cloud、Docker 等技术体系构建服务治理、负载均衡、服务容错、服务网关、配置中心、事件驱动、服务安全、服务监控、服务测试和服务部署等核心主题，并基于这些核心主题给出了具体的案例分析。

本书面向立志于成为微服务架构师的后端开发人员，读者不需要有很深的技术水平，也不限于特定的开发语言，但是熟悉 Java EE 常见技术并掌握系统设计基本概念，将有助于更好地理解书中的内容。同时，本书也可供具备不同技术体系的架构师同行参考。

◆ 著　　郑天民
责任编辑　刘　博
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
固安县铭成印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：18.25　　2018 年 12 月第 1 版
字数：399 千字　　2018 年 12 月河北第 1 次印刷

定价：59.80 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前言

当下互联网行业飞速发展，快速的业务更新和产品迭代给系统开发带来了新的挑战。围绕来自系统开发团队外部及内部的变化与挑战，如何更为合理地划分系统和团队边界，如何更加有效地组织系统开发过程，如何通过技术手段识别和消除开发过程中存在的浪费，成为广大系统开发人员和管理人员亟待解决的问题。在这种背景下，微服务架构应运而生了。要想实现微服务架构，不但要了解微服务的设计原理，还要掌握相应的技术体系。关于前者，笔者已出版了《微服务设计原理和架构》一书。本书主要面向的是后者，即围绕微服务架构的实现技术给出相应的工具框架和工程实践，旨在为广大开发人员提供对主流微服务架构实现技术的完整、全面和实用的介绍。

全书围绕实现微服务架构的基础组件和关键要素，引入 Spring Boot、Spring Cloud、Docker 等技术体系构建微服务体系。本书在组织结构上分为如下所示的 12 章，并结合这些章节内容要点给出一个完整的实现案例。

第 1 章　微服务架构设计，简要介绍微服务的建模方法、服务拆分和集成、基础组件和关键要素以及实现技术。

第 2 章　使用 Spring Boot 构建服务，引入 Spring Boot 作为单个微服务的实现框架。

第 3 章　Spring Cloud Netflix Eureka 与服务治理，引入 Eureka 实现服务注册和发现。

第 4 章　Spring Cloud Netflix Ribbon 与负载均衡，引入 Ribbon 实现客户端负载均衡。

第 5 章　Spring Cloud Netflix Hystrix 与服务容错，引入 Hystrix 实现面向消费者的服务容错。

第 6 章　Spring Cloud Netflix Zuul 与 API 网关，引入 Zuul 实现 API 网关。

第 7 章　Spring Cloud Config 与配置中心，引入 Spring Cloud Config 实现分布式配置中心。

第 8 章　Spring Cloud Stream 与事件驱动，引入 Spring Cloud Stream 实现服务之间的消息传递。

第 9 章　Spring Cloud Security 与服务安全，引入 Spring Cloud Security 实现服务的安全访问。

第 10 章　Spring Cloud Sleuth 与服务监控，引入 Spring Cloud Sleuth 实现服务的跟踪和监控。

第 11 章　Spring Test 与服务测试，引入 Spring Test 实现多层次的服务测试。

第 12 章　Docker 与服务部署，引入 Docker 实现服务的高效部署。

通过对全书内容的系统学习，读者将对微服务架构的技术体系和实现机制有全面而深入的了解，为后续的工作和学习铺平道路。

在本书的撰写过程中，我的家人特别是我的妻子给予了我极大的支持和理解。感谢以往以及现在公司的同事们，身处业界领先的公司和团队，让我得到很多学习及成长的机会，没有平时大家的帮助，就不可能有本书的诞生！

本书所涉及的源代码，读者可到人邮教育社区（www.ryjiaoyu.com）下载。

由于时间仓促，编者水平有限，书中难免存在不足及疏漏之处，恳请广大读者批评指正。有任何建议和意见，可关注微信公众号“程序员向架构师转型”，与作者交流。

郑天民

2018 年 5 月

目录

第 1 章

微服务架构设计

近年来，微服务架构（Microservices Architecture）已经成为一种主流的软件开发方法论，它把一种特定的软件应用设计方法描述为能够独立部署的服务套件。所谓微服务（Microservices），就是一些具有足够小的粒度、能够相互协作且自治的服务体系。每个微服务都比较简单，仅关注于完成一个功能并能很好地完成该功能，而这里的功能代表的是一种业务能力。构建微服务体系需要一套完整的方法论和工程实践，而微服务架构的提出代表的就是实现微服务体系的架构模式，即为我们提供了这些方法论和工程实践。从这个角度讲，微服务架构需要我们理解、学习并应用到日常开发过程中去。

微服务架构基于分布式系统，同时借助了面向服务架构和企业服务总线的设计理念并做了改进和优化，从而形成一种新的架构体系。微服务架构一方面具备技术、业务和组织上的优势，另一方面也在技术架构和研发过程中存在巨大挑战。微服务架构的实施需要具备一定的前提，而构建微服务架构是一项系统工程，涉及服务建模、实现技术、基础设施和研发过程等各个维度。在实施过程中，也需要根据现有系统的具体情况采用合适的实施模式。

本章作为全书的开篇，对微服务设计原理与架构做了全面介绍。本书的关注点是微服务架构的实现技术，本章也会梳理目前市面上主流的微服务技术体系并完成技术选型。在本章的最后，我们还会给出全书的组织架构。

1.1 直面微服务架构

顾名思义，微服务区别与其他服务体系的关键在于它的“微”特性。“微”是小的同义词，所以容易让人联想到微服务都是小型的服务，这是微服务的第一个特性。微服务之间只有通过相互的协作和交互才能构成完整的服务体系，而这种协作和交互机制也是微服务区别其他服务体系的另一个主要方面。

1.1.1 分布式系统与微服务架构

所谓分布式系统（Distributed System）是指硬件或软件组件分布在不同的网络计算机上，

彼此之间仅仅通过消息传递进行通信和协调的系统。从这个定义中可以看出，分布式系统包含两个区别于单块系统（Monolith System）的本质性特征：一个是网络，分布式系统的所有组件都位于网络之中，对于互联网应用而言，则位于更为复杂的互联网环境中；另一个是通信和协调，与单块系统不同，位于分布式系统中的各个组件只有通过约定、高效且可靠的通信机制进行相关协作，才能完成某一项业务功能。这些特征是我们在设计和实现分布式系统时首先需要考虑的。

分布式系统相较于单块系统在具备一定优势的同时，也存在一些我们不得不考虑的特性，包括但不限于网络传输的三态性、系统之间的异构性、数据一致性、服务的可用性等。以上问题是分布式系统的基本特性，我们无法避免，只能想办法进行利用和管理，这就给我们设计和实现分布式系统提出了挑战。微服务架构本质上也是一种分布式系统，但在遵循通用分布式特性的基础上，微服务架构还表现出一定的特殊特性。下面将围绕微服务架构的这些特殊特性展开讨论。

Martin Fowler 指出，微服务架构具有以下特点。

（1）服务组件化

所谓组件（Component）是一种可独立替换和升级的软件单元。在我们日常开发过程中，可能会设计和使用很多组件，这些组件可能服务于系统内部，也可能存在于系统所运行的进程之外。而服务就是一种进程外组件，服务之间利用诸如 RPC（Remote Procedure Call，远程过程调用）等通信机制完成交互。服务组件化的主要目的是服务可以独立部署。如果某个应用程序是由一个运行在独立进程中的很多组件组成，那么对任何一个组件的改变都将导致整个应用程序必须重新部署。但是如果把应用程序拆分成很多服务，通常情况下，只需要重新部署那个改变的服务即可。在微服务架构中，每个服务运行在其独立的进程中，服务与服务之间采用轻量级通信机制互相沟通。

（2）按业务能力组织服务

在寻找把一个大的应用程序进行拆分的方法时，研发过程一般都会围绕产品团队、UED 团队、APP 前端团队和服务器端团队而展开，这些团队也就是通常所说的职能团队（Function Team）。当使用这种模式对团队进行划分时，任何一个需求变更，无论大小，都将导致跨团队协作，从而增加沟通和协作成本。而微服务架构下的划分方法则有所不同，它倾向于围绕业务功能的组织来分割服务。这些服务面向的是具体业务结构，而不是面向某项技术能力。因此，团队是跨职能的（Cross-Functional）特征团队（Feature Team），包含用户体验、项目管理和技术研发等开发过程所要求的所有技能。每个服务都围绕着业务进行构建，并且能够被独立地部署到生产或类生产环境。

（3）去中心化

服务集中治理的一种好处是在单一平台上进行标准化，而采用微服务的团队更喜欢不同的标准。把集中式系统中的组件拆分成不同的服务，我们在构建这些服务时就会有更多的选择。

对具体的某一个服务而言，应该根据业务上下文，选择合适的语言和工具进行构建。

此外，微服务架构也崇尚于对数据进行分散管理。当集中式的应用使用单一逻辑数据库进行数据持久化时，通常选择在应用的范围内使用一个数据库。然后，微服务让每个服务管理自己的数据库，无论是相同数据库的不同实例，还是不同的数据库系统。

（4）基础设施自动化

许多使用微服务架构的产品或者系统，它们的团队拥有丰富的持续集成（Continuous Integration）和持续交付（Continuous Delivery）经验。团队使用微服务架构构建软件需要更广泛的依赖基础设施自动化技术。

在微服务中同样需要考虑服务容错性设计等分布式系统所需要考虑的问题，我们对以上特点进行总结和提炼，认为微服务具备业务独立、进程隔离、团队自主和交付独立性等“微”特性。

1.1.2 微服务架构的优势与挑战

微服务并不是一个纯技术的事物，而是涉及业务边界、基础设施、组织架构等的综合体系。微服务架构存在明显的优势，但也面临着各种挑战。

1. 微服务架构的优势

微服务架构能够为我们带来许多优势，包括技术上的优势、业务结构上的优势及组织上的优势。

（1）技术优势

微服务架构提供的是一种高内聚、低耦合的组件化方案。组件所能带来的独立性与健壮性微服务都可以具备，但是微服务的组件化特征更多的表现在对业务的提炼和对边界的思考。使用微服务架构迫使我们使用诸如领域驱动设计（Domain Driven Design，DDD）的思想去开展策略设计和技术设计，从而为更好地划分业务功能、提取界限上下文和开展系统集成工作提供依据。而这些方法和依据的背后恰恰是我们在架构设计过程中经常会碰到的问题。

当系统架构转变成一系列微服务之间的通信和集成时，我们就明白具体实现技术已经不是系统设计和开发的主要约束条件。因为在微服务架构中，各个微服务之间使用的是轻量级的通信机制。所谓轻量级是指这些通信机制与具体实现技术无关，不受限于某一个特定协议或交互媒介。每个微服务高度独立，可以采用适合自身开发团队和技术体系的工具及框架来实现某个微服务，从而为我们提供了宝贵的技术自由度。

在微服务架构中，通过服务拆分与集成，单个服务在保持通信方式不变的前提下，对其内部功能和技术的改变不会对外部依赖它的服务产生任何影响。结合可扩展性的概念，微服务架构无疑具备这方面的优势。高扩展性往往能够带来高可伸缩性，因为可以伸缩的前提是对系统进行合理地拆分。当我们明确系统的运行瓶颈，并把引起这些瓶颈的业务功能构建成独立的微

服务，就可以采用服务集群等手段有效加强服务的运行环境和状态。

微服务是可用于改造遗留系统（Legacy System）的强有力武器。面对遗留系统，一方面该系统的技术体系可能存在设计上的重大缺陷，另一方面则是因为代码量巨大且不容易修改。在代码层次与遗留系统进行直接集成是痛苦且具有挑战性的工作，但是遗留系统以提供接口的方式暴露某些功能入口仍然是一个相对容易实现的过程。一旦获取这些接口，微服务架构就能与之进行通信并完成功能整合。

持续交付通过简单、可重复的流程来确保软件发布过程的可靠性，微服务作为独立的可部署单元，非常适合使用持续交付，因为每一项服务都可以在不依赖于其他服务的条件下完成发布和部署。基于微服务架构，持续交付管道可以运行得更快，从而加速问题反馈，这是持续交付的主要目标之一。而持续交付的另一个目标是降低系统风险，微服务小而独立，一旦出现问题很容易进行修复和回滚操作。

（2）业务和组织优势

微服务架构的技术特征决定了开发各自微服务的团队之间只需要进行较少的协调工作，这为降低研发过程浪费提供了基础。从组织管理的角度出发，自治式团队较之集中式管理模式下的团队在团队建立和发展上能够得到更好的可扩展性。集中式团队普遍受限于技术上的约束和决策，诸如可用性、性能等非功能性需求都不是由某个业务功能和系统的独立团队所负责，而是需要集中式的、系统级别的实现和管理。微服务架构可以把大型团队打散成小型团队，小型团队比大型团队具有更低的失败可能性。从这点上讲，微服务架构还能够降低团队组织级别的风险。

微服务架构从技术角度给出了缩短产品开发周期的方法，主要表现在并行的开发模式上。将业务拆分成多个微服务能够让不同的业务功能处于一种并行开发的状态，因为每个团队所负责的业务需求只影响到团队自身的微服务，所以各个团队能够独立开发，整个系统也很容易分布到各个团队中。如果涉及简单业务需求的变更或者是发布部署的要求，独立的微服务之间也不需要太多的统一协调工作。大规模的统一协调工作通常只发生在业务结构产生重大变更的场景，而这种场景对于软件开发而言显然是应该尽量避免并进行提前规划。

2. 微服务架构面临的挑战

将一个系统分散到多个微服务中使得系统整体结构变得更加复杂，这在技术架构和研发过程中同样给我们带来了一定挑战。

（1）技术架构挑战

微服务架构所带来的独立性可能导致每个服务采用不同的技术体系。去中心化的设计思想意味着微服务之间不需要共享技术，然而缺少通用的技术体系同样也会加剧系统开发的复杂度。去中心化是微服务架构的基本特征之一，但当每个微服务团队都采用自身团队擅长的技术进行微服务构建时，导致的问题就是没有一个人能够明白系统中所采用的所有技术。尽管多数情况

下，我们并不需要深入到其他团队的微服务中，但从统一发布和运维等角度去看待整个系统时，这种技术复杂性就可能会是一个问题。我们需要把控中心化和去中心化之间的平衡性。

每个微服务都应该具备与其他服务保持相互独立的版本控制机制，我们提倡为每个微服务建立版本并根据业务迭代更新这个版本。如果每个服务都能够完全按照自身服务的演进过程进行独立发布，那么事情还比较简单。但如果某一个服务与另一个服务具有版本关联性，即这两个服务要不都不发布，要不一起发布，那么事情就变得复杂了。如果这些具有版本关联性的服务很多且版本的更新频率很高，如何正确管理服务版本就是一项具有挑战性的工作。

（2）研发过程挑战

对研发过程而言，分散的服务在一定程度上等同于分散的业务需求，来源于微服务具有的业务独立性和明确的服务边界。但对于整个系统而言，需求的边界并不那么容易划分。当我们面对微服务架构，如何确定业务功能的粒度，如何把非功能性需求分解到各个微服务中，如何从系统整体上把握需求的优先级等都是我们不得不面对的问题。

对于大多数尚未采用微服务架构的团队而言，使用微服务架构就是一个引入变化的过程。关于这一点，很多人都存在一定的误解，最大的误解就是认为新想法一旦引入就意味着已经成功了。当我们尝试采用微服务架构时，团队管理人员想了很多办法、做了很多工作找到了问题的切入点，跟各个利益方讨论之后终于引入了大家都赞同的变化，然后就期望这个变化能按照自己想的那样发挥效果，这是不对的。当微服务架构被引入后，我们还要做很多事情，前面所提到的各种技术、架构和过程的挑战都需要我们进行跟踪和协调。

1.1.3 实施微服务架构

实施微服务架构存在固定的模式，最典型的实施模式是从一个单块系统逐步转变为多个维度的微服务架构。通常，不同的业务功能根据优先级、工作量等可以转化为一个一个的微服务。促使这种转变的主要目的可能来自于业务的快速变化和迭代要求，或者来自于将系统的部署与运维变得独立而简单，而更为重要的原因则可能来自于对架构可扩展性和可伸缩性的考虑。当然，从单块系统转变为微服务架构在方式上不同团队、不同场景可能也存在差异性，对于那些对服务的可用性要求很高的场景而言，可以先通过使用一些服务隔离手段设法提高服务的整体可靠性之后再拆分微服务；而有些场景下，对业务的分离可能是系统的最大痛点，那么快速、合理的拆分业务就是实施微服务架构的第一步。

当然，现实中也存在从无到有实施微服务架构的机会，尽管这种机会非常少见。即使有这种机会，一般的做法也是先构建一个单块系统，然后再设法加以改进，因为从无到有的过程就是一个从小到大的过程，而从微服务实施的前提上讲，小系统没有必要直接采用微服务架构。但在系统架构的初始设计阶段，采用领域驱动的设计思想建立粗粒度的领域模型是一种最佳实践，有助于后续向微服务架构转型的实施过程。领域驱动设计同样也驱使技术团队在设计阶段

就考虑到合适的团队分工和各个团队之间的独立性。

混合式是尝试微服务架构的第三种方式，因为微服务架构很容易与现有系统并存，这些微服务能够很好的对系统起到补充作用。如果需要，我们能够很快实现很多微服务，而一旦出现问题，移除这些微服务的成本也较低。微服务与遗留系统之间的易整合性是我们采用微服务的一个主要原因，关于这种整合关系及其实施方法可参考笔者所著的《微服务设计原理与架构》一书中的内容。

1.2 服务建模方法

如同系统架构设计首先需要进行业务建模一样，微服务架构的设计同样从服务建模开始。一般而言，软件行业建模工作的目的在于将业务需求转变为一种能够直接用于软件实现的载体，以 UML（Unified Modeling Language，统一建模语言）为代表的建模语言是这种载体的具体表现形式。在微服务架构中，服务建模的目的同样是为了便于开发人员开展后续的服务设计和实现工作。本节讨论微服务建模的切入点，包括服务的模型、服务的边界和服务的数据。

1.2.1 服务的模型

服务模型提供服务的概念模型，并给出服务的统一表现形式。服务的概念模型来自两个维度：一个是服务标准，即什么样的服务才是一个好的服务；另一个是服务级别，即不同的服务应该具备不同的重要性。

关于服务标准，SOA 体系中的很多设计理念同样适合于微服务架构，包括服务无状态（Service Statelessness）、服务可重用（Service Reusability）、服务可发现（Service Discoverability）、服务自治（Service Autonomy）和服务松耦合（Service Loose Coupling）等在内的设计原则都是服务设计上的核心标准。

关于服务级别，可以从发生具体事故时服务对用户体验的影响、所造成的经济损失等角度对服务进行具体分级。在服务分级上，一种常见的分级方法是将系统服务分成 3 个不同的等级：一级服务具备完善的容错降级机制及对低级别服务的熔断措施、定期压测、配置高级别的监控报警流程等；二级服务多采用异步方式进行系统交互，容忍暂时数据不一致性；三级服务则可随时降级整个服务。

服务需要具备统一的表现形式，也就是需要具备契约化的约束条件，而这种契约化的约束条件一般可以通过文档的方式进行展现。对于每一个服务而言都应该提供一份契约文档，并发布到统一的管理平台以方便服务相关人员进行查看和更新。服务契约（Service Contract）化要求至少对服务的几个基本方面作出说明，包括具体接口的 API 接入技术说明；服务能力的描述

（如服务所属的业务模块、所能提供的业务功能以及具体的应用场景）；提供这些能力所约定的一些限制条件说明（这些限制条件多数与业务场景有关，现实中以下这种情况并不少见，即同一个接口定义以及其所代表的业务能力在不同的场景中会出现不同的表现形式）；支持的最新版本和历史版本的说明（这点对于微服务而言是必备的要素）。显然，将服务契约化有助于在开发人员之间形成统一语言，减少多余且可能反复的口头沟通，降低协作成本。

1.2.2 服务的边界

在明确了服务分类和服务模型之后，接下去的工作就是识别服务，识别服务的切入点在于识别服务与服务之间的边界（Boundary）。明确服务边界是下一节中将要介绍的服务拆分与集成的前提。

在微服务架构中，识别服务边界的方法可以参考领域驱动设计思想。在领域驱动设计中，有两个主要的设计维度，即设计的策略维度和设计的技术维度。其中，设计的策略维度更关注如何设计领域模型以及对领域模型的划分，其目的在于清楚划分不同的系统与业务关注点。策略维度是一个面向业务、具备较高层次的设计维度，更偏重于业务架构的梳理以及考虑如何把业务架构和技术架构相结合的问题。策略维度的通用语言（Ubiquitous Language）、子域（Sub-Domain）、界限上下文（Boundary Context）等概念为识别服务边界提供了一整套方法论。设计的技术维度包含聚合（Aggregate）、领域事件（Domain Event）等组件，有助于组织服务内部以及服务与服务之间进行交互的方式。关于领域驱动设计的更多内容可以参考相关资料。

在通过使用领域和界限上下文进行服务边界划分的过程中，也存在一些服务边界划分的原则。常见的边界划分原则包括服务关联度原则、业务能力职责单一原则、读写分离原则以及组织关系原则。

1.2.3 服务的数据

通过确定服务边界，服务在逻辑上就变成了一个个独立的个体，但是我们还要小心一个并不容易进行独立处理的因素，那就是数据。绝大多数服务都会依赖数据，而很多数据可能也会被一批服务所依赖。

规范化数据模型是传统关系型数据库设计的核心，即通过三大范式实现数据的有效存储，并为开发人员提供一整套对数据的操作方式。规范化数据模型有利有弊，它虽然为如何管理关系型数据提供了最佳设计理念，但同时也限制了数据查询的灵活性和高效性，特别是当查询涉及很多关联（Join）场景时，会导致查询性能严重低下。规范化数据模型的另一个问题在于中心化思想，即把数据统一存储在一个中央数据库中。当大量数据存在于同一数据库时容易造成

数据库访问瓶颈，从而影响数据访问性能，并为系统可用性埋下隐患。

针对规范化数据模型存在的以上问题，微服务架构中数据管理的基本思路就是数据去中心化。数据去中心化场景包括跨表查询场景、跨库查询场景以及技术耦合场景。在对数据去中心化进行不断尝试的过程中，采用“代码分离→重复数据库模式→迁移数据读写操作→抽取服务化接口”等步骤构成了完整的数据去中心化流程。关于各种数据去中心化场景、应对方式以及去中心化工作流程的详细描述，可以参考笔者所著的《微服务设计原理与架构》一书。

1.3 服务拆分与集成

本节将在服务建模的基础上，简要分析服务拆分的策略和手段，同时给出了对拆分之后的服务进行集成的各种实现方法和技术体系。

1.3.1 服务拆分

在微服务架构中，我们认为服务是业务能力的代表，需要围绕业务进行组织。服务拆分的关键在于正确理解业务，识别单体内部的业务领域及其边界，并按边界进行拆分。所以，微服务的拆分模式本质上是基于不同的业务进行拆分。

服务拆分存在两大维度，即业务与数据。业务体现在各种功能代码中，通过确定业务的边界，并使用领域与界限上下文、领域事件等技术手段可以实现拆分。而数据的拆分则体现在如何将集中式的中心化数据转变为各个微服务所拥有的独立数据，这部分工作同样十分具有挑战性。

关于业务和数据谁应该先拆分的问题，可以是先数据库后业务代码，也可以是先业务代码后数据库。在拆分中遇到的最大挑战可能是数据层的拆分，因为在数据库中，可能存在各种跨表连接查询、跨库连接查询，以及不同业务模块的代码与数据耦合得非常紧密的场景，这会导致服务的拆分变得困难。因此，在拆分步骤上推荐数据库先行。数据模型能否彻底分开，很大程度上决定了微服务的边界功能是否彻底划清。

根据系统自身的特点和运行状态，服务拆分的方法通常分为绞杀者与修缮者两种模式。绞杀者模式（Strangler Pattern）指的是在现有系统外围将新功能用新的方式构建为新的服务的策略，通过将新功能做成微服务方式，而不是直接修改原有系统，逐步地实现对老系统的替换。采用这种策略，随着时间的推移，新的服务就会逐渐“绞杀”老的系统。对于那些规模很大而又难以对现有架构进行修改的遗留系统，推荐采用绞杀者模式。而修缮者模式就像修房或修路一样，将老旧待修缮的部分进行隔离，并用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同功能。从这种思路出发，修缮者模式更多地表现为一种重构技术，具体实现可以参考 Martine Fowler 的 Branch By Abstraction 重构方法。