

# Monolith to Microservices

wangwei17

# 目录

# 前言

翻译说明

---

# 第1章. 关于微服务，此章足矣

1.1 什么是微服务

1.2 单体应用

1.3 关于耦合和内聚

1.4 领域驱动设计

# 第2章. 规划迁移

- 2.1 了解迁移的目标
- 2.2 为什么选择微服务
- 2.3 什么时候微服务不是一个好的选择
- 2.4 权衡折衷
- 2.5 带动其他人员
- 2.6 组织变革
- 2.7 增量迁移的重要性
- 2.8 变革的成本
- 2.9 我们从哪里启程
- 2.10 领域驱动设计
- 2.11 融合模型
- 2.12 重组团队
- 2.13 如何知道迁移过程是否有效

# 第3章. 拆解单体应用

3.1 改变单体应用，还是不改变?

3.2 迁移模式

3.3 绞杀者模式

3.4 在功能迁移过程中对其修改

3.5 UI组合模式

3.6 抽象分支模式

3.7 新老并行模式

3.8 装饰协作模式

3.9 CDC模式

# 第4章. 拆分数据库

4.1 共享数据库

4.2 然而，现在无法拆分数据库

4.3 数据库视图

4.4 把数据库包装成服务

4.5 把数据库视为服务接口

4.6 改变数据的所有权

4.7 数据同步

4.8 在应用中同步数据

4.9 跟踪器写入模式

4.10 拆分数据库

4.11 先拆分数据库还是先拆分代码

4.12 数据分离的例子

4.13 拆分表

4.14 把表的外键关系移至代码

4.15 数据库事务

4.16 Sagas

# 第5章. 成长的烦恼

5.1 服务越多，烦恼越多

5.2 团队不断扩张时的微服务所有制问题

5.3 破坏性的服务变更

5.4 报表问题

5.5 监控和问题定位

5.6 本地开发体验的问题

5.7 微服务部署和状态管理的问题

5.8 端到端的测试

5.9 局部优化 VS. 全局优化

5.10 系统的鲁棒性和弹性

5.11 孤儿服务的问题

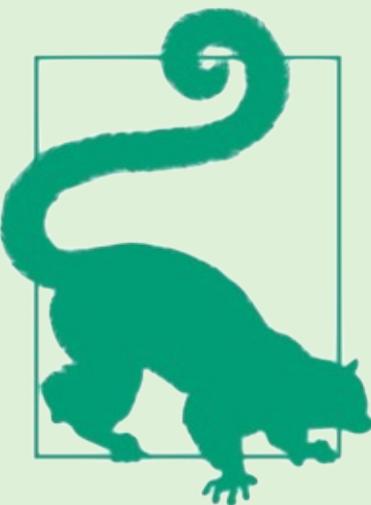
# 第6章. 结束语

## 6. 结束语

---

# 从单体应用到微服务架构

**Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith**中文版。



- 已经完成书籍核心内容的翻译工作，书籍内容也会在这里进行更新。
- 欢迎大家提出宝贵意见，不管是翻译的内容问题，还是字词错误，都欢迎大家提交 issue(网页下方的评论区域)，我会及时跟进。
- 译者信息：王伟 ([wangwei1237](#))

## 注意

- 这种区域的内容并非是原书的内容，而是根据译者的经历补充的额外资料。
- 为了增加对专有名词的理解，在翻译过程中增加了对专有名词的额外解释。例如：沉没成本，货物崇拜等。

## 版权声明

本翻译稿是在阅读原版英文的过程中记录总结而成，属于兴趣爱好而为，作为和大家学习交流使用。如需纸质中文版，希望大家购买正规出版社出版的中文翻译版。

本翻译稿采用“保持署名—非商用”创意共享4.0许可证。只要保持署名和非商用，您可以自由地阅读、分享本翻译稿。

您可以：

- 下载、保存以及打印本翻译稿
- 网络链接、转载本翻译稿的部分或者全部内容，但是必须在明显处提供读者访问本翻译稿发布网站的链接

您不可以：

- 以任何形式出售本翻译稿的电子版或者打印版
- 擅自印刷、出版本翻译稿
- 以纸媒出版为目的，改写、改编以及摘抄本翻译稿的内容

Well, that escalated quickly, really got out of hand fast!

——王牌播音员

在深入探讨如何使用微服务之前，对什么是微服务架构达成共识非常重要。我会讨论一些我经常看到的、关于微服务架构的误解，同时也会讨论那些经常被忽略的、微服务的细节之处。需要具备坚实的基础知识，才能充分利用本书其余部分的内容。因此，本章会：

- 解释什么是微服务架构
- 简要回顾微服务是怎样发展起来的（当然，这意味着还要回顾单体架构）
- 探讨微服务架构所带来的优势和挑战

### 达成共识的重要性

在讨论某个概念之前，首先对要讨论的事情达成共识，确保我们讨论的是一个事情，这一点非常重要。

参加过很多技术讨论会议，总会发现大家在会上讨论的热火朝天，激情四射，但是到了最后却没达成什么结论。后来，仔细想了下，发现大家会上讨论的某个事情压根就不是一个事情。

一百个人眼中有一百个哈姆雷特，虽然都是在讨论哈姆雷特，但是不幸的是，我们说的不是一个哈姆雷特。

随着接触到的技术概念越来越多，我发现，不同的团队眼中的CI的概念是不同的，敏捷的概念是不同的，CI/CD的概念也是不同的……

现在我终于发现，不同团队所说的微服务的概念也是不同的……

能在一个不同的概念上讨论问题，还能有一些看起来似乎合理的跟进事项，这看起来确实是一件非常神奇的事情。

所以，这里，我极力的推荐大家认真的阅读本书的第一章，这是领会本书精髓的第一步。

# 什么是微服务

微服务是围绕业务领域建模的、可独立部署的服务。微服务利用网络相互通信，作为一种可选的架构，微服务为解决可能遇到的问题提供了许多选择。由此可见，微服务架构基于多个微服务间的协作。

微服务是一种面向服务的架构（*SOA: service-oriented architecture*）。微服务强调如何划分服务边界，并且其关键是服务的独立可部署。同时，微服务还具有与技术无关的优势。

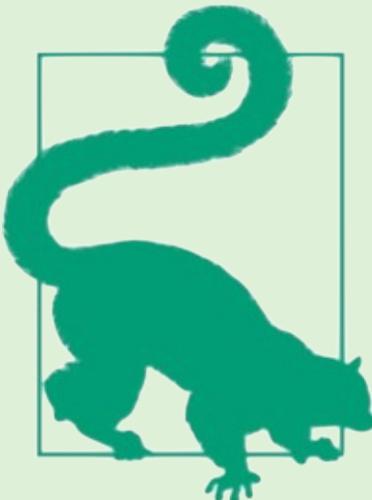
关于微服务和SOA之间的区别，我们将在[第三章的绞杀者模式](#)中具体介绍。

从技术角度来看，微服务通过一个或多个网络端点（*network endpoints*）开放其封装的业务功能。微服务利用这些网络相互通信，从而使其成为一种分布式系统。微服务封装了数据的存储和检索，并通过定义清晰的接口开放其封装的数据。因此，对于微服务而言，数据库会隐藏在服务边界内部。

如上的所有内容都需要展开来细说，因此让我们更深入地研究一下微服务的这些概念。

# 独立部署

独立部署的概念即：无需利用任何其他的服务就可以变更微服务并将其部署到生产环境。更重要的是，独立部署并不仅仅如此。实际上，独立部署是我们在系统中管理部署的方式。大部分的软件发布都应遵循独立部署的准则。独立部署的想法虽然简单，但是执行起来却很复杂。



如果只从本书学到一件事情，那就是：确保自己拥抱微服务独立部署的概念。养成独立部署的习惯：无需部署其他任何东西就可以把单个微服务的变更发布到生产环境。一旦养成这个习惯，许多美好的事情将随之而来。

为了保证独立部署，需要确保服务是低耦合的。换句话说，我们能够在无需更改其他任何服务的情况下而变更一个服务。这意味着我们需要在服务之间建立明确的、定义清晰的、稳定的契约。但是，某些具体的实现却导

致独立部署变得很困难——例如，共享数据库的方案就特别的麻烦。对具有稳定接口的、低耦合的服务的需求指引着我们思考：如何首先找到服务边界。

# 针对业务领域建模

跨程序边界变更的代价很高。如果发布某个功能需要修改两个服务并编排这两个服务的部署，那么这比在一个服务（或者一个单体应用）内进行相同的变更要做更多的工作。因此，我们接下来想找到方法——以确保尽可能少的跨服务变更。

在无法使用真实的案例来解释某些概念时，本书会沿用我在[Building Microservices](#)一书中的方法：使用一个虚拟的领域和公司来解释这些概念。本书讨论的公司是Music Corp。Music Corp是一家大型跨国组织，尽管主要专注于销售CD，但却仍以某种方式维持经营。

我们已经决定让Music Corp在21世纪也能[爆发激情](#)，并且作为该步骤的一部分，我们正在评估现有的系统架构。在[图1-1](#)中，我们看到了一个简单的三层架构（*three-tiered architecture*）。我们有一个基于Web的用户界面，一个业务逻辑层后端（该层为单体应用），以及传统数据库中的数据存储。通常，不同的团队拥有[图1-1](#)中的不同的层。

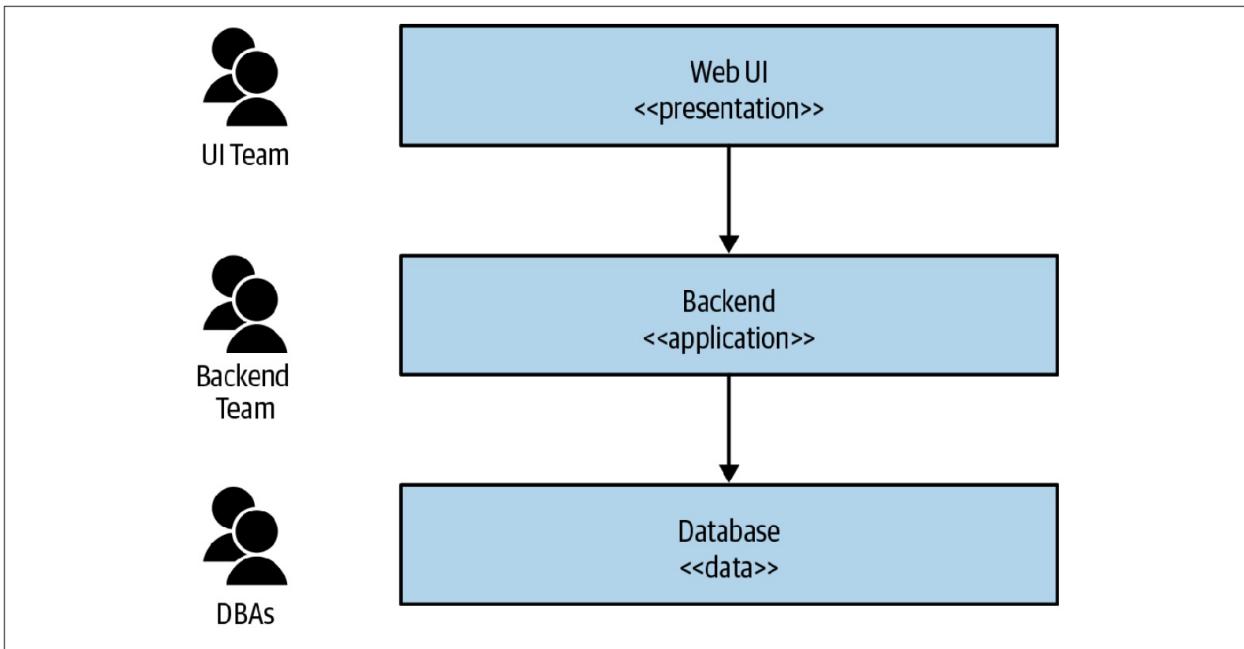


图1-1. Music Crop的传统的3层架构的系统

我们希望简单的修改功能以允许用户可以指定他们喜欢的音乐流派。这要求我们：

- 更改UI，以显示音乐流派选择的UI
- 更改后端服务，为UI显示并修改音乐流派提供数据
- 更改数据库，以保存用户选择的音乐流派

如[图1-2](#)所示，这些不同层级的变更将由不同的团队来管理，并且这些变更需要按照正确的顺序来部署。

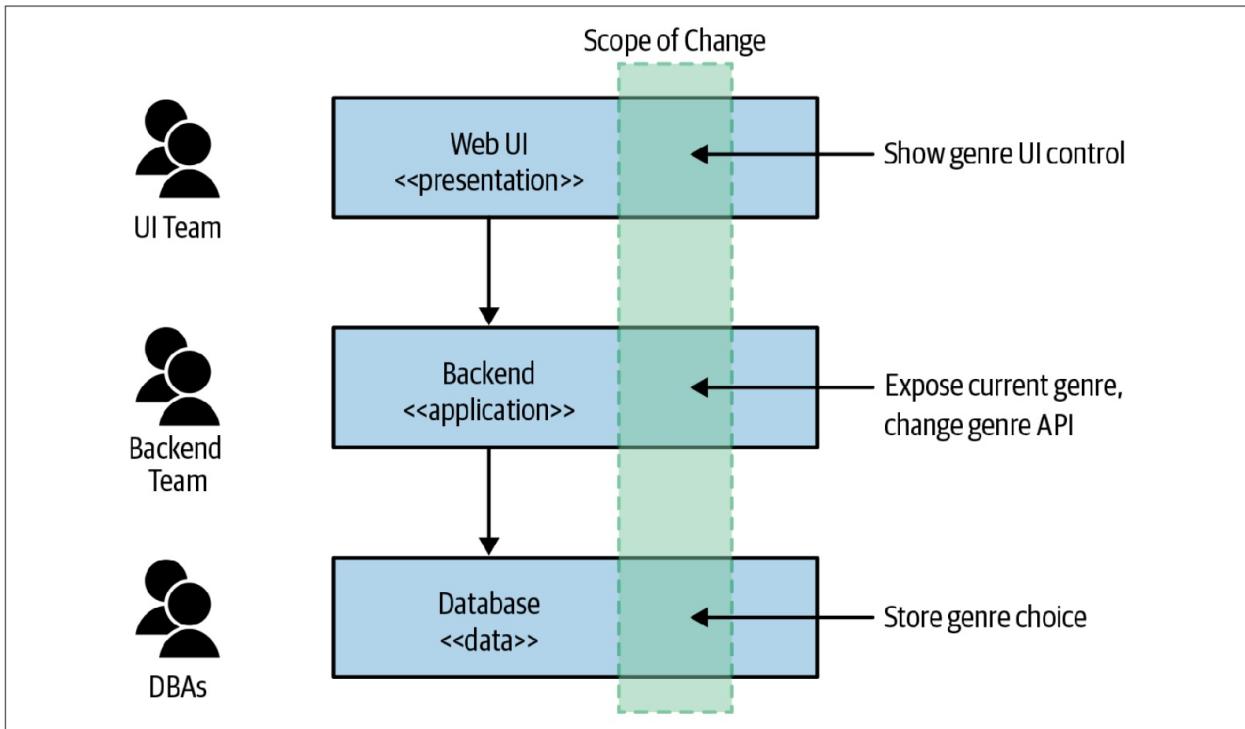


图1-2. 跨层级修改涉及到更多的工作

目前这种架构还不错。所有架构最终都围绕某些目标进行优化。三层架构之所以如此普遍，部分原因在于它是通用架构——每个人都听说过。因此，选择我们已经在其他地方看到过的通用架构，通常是我们不断看到三层架构的原因之一。但是我认为我们一次又一次的看到这种架构的最大原因是：三层架构基于我们的团队的组织方式。

正如著名的康威定律所陈述的那样：

设计系统的组织，其产生的设计等同于组织之内、组织之间的沟通结构。

Melvin Conway, *How Do Committees Invent?*

三层架构是康威定律在实践中的一个很好的例子。过去，IT组织主要根据人员的核心能力来分组：数据库管理员与其他数据库管理员组成一个团队，Java开发人员与其他Java开发人员处于一个团队，前端开发人员则处

于另外的团队。我们根据人员的核心能力对其分组，因此我们创建了可以与这些团队保持一致的IT资产。

康威定律解释了为什么这种三层架构会如此普遍。三层架构并不算太差，它只是围绕一种软件开发的力量而优化——我们历来按照人们对技术的熟悉程度来分组。但是我们面对的问题已经发生了改变。我们对软件的期望已经发生了变化。现在，我们将人员组织在一个多技术的团队，以减少团队间的交接和团队间的隔阂。我们希望比以往更快地发布软件。这促使我们在组织团队的方式以及如何拆分系统方面做出不同的选择。

改变功能主要是业务功能的变化。但是在图1-1中，业务功能实际上分布在三层架构中的每一层，这增加了跨层修改功能的可能性。对于三层架构而言，我们具有技术上的高内聚，但是业务功能却是低内聚的。如果我们想让功能变更变得更容易，我们需要改变组织代码的方式。在组织代码时，可以选择用业务功能的内聚来替换技术的内聚。最终，每个服务可能包含（也可能不包含）所有的三层，但是，是否包含所有的三层就是服务实现需要关心的问题了。

### 技术的高内聚和业务的高内聚

高内聚、低耦合是Larry Constantine在1974年发表的**Structured Design**中提出的概念，目前已经成为判断软件设计好坏的标准之一。

由于历史原因，我们仅从软件工程的领域来考虑内聚和耦合，就像本节讲到的三层架构一样。但是，我们必须意识到，软件是实现业务功能的手段，在实现业务功能的时候，不可避免的会从技术的角度对业务进行拆解，这也导致业务的拆分也会依赖软件的具体实现。

然而，这种方式是合理的吗？是目标依赖执行还是执行要依赖目标？

这让我想起了最近几年一直提倡的人人都是产品经理的概念，这实际上就是在要求工程师在实现业务功能的时候，除了考虑技术上的内聚外，还要考虑业务上的内聚，然后综合二者来设计所采用的架构。然而，不幸的是，大多数团队却在让研发人员为产品提各种idea。

微服务首先是业务逻辑上的高内聚、低耦合，然后才是业务逻辑内的工程实现上的高内聚、低耦合。

让我们将图1-2与图1-3中所示的替代架构进行比较。我们有一个专门的Customer服务，该服务开放了一个UI，以允许用户更新其信息，并且用户的状态也存储在该服务中。最喜欢音乐类型的选择是与给定的用户相关的，因此图1-3的功能变更会更加本地化。在图1-3中，我们还显示了从Catalog服务中获取可用类型的列表。我们还看到一个新的Recommendation服务会从Customer服务访问用户所喜欢的音乐流派信息。我们可以在后续的发布版中轻松实现Recommendation服务。

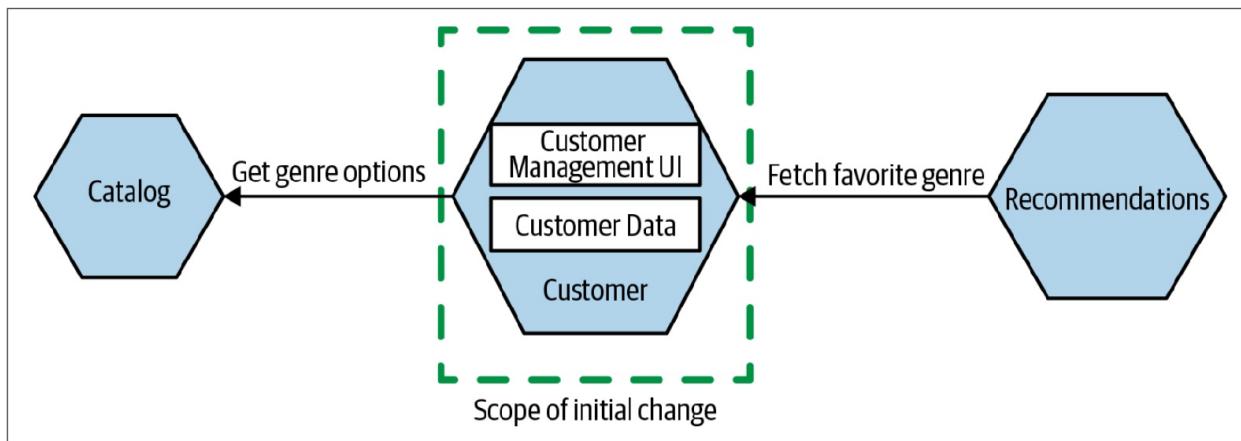


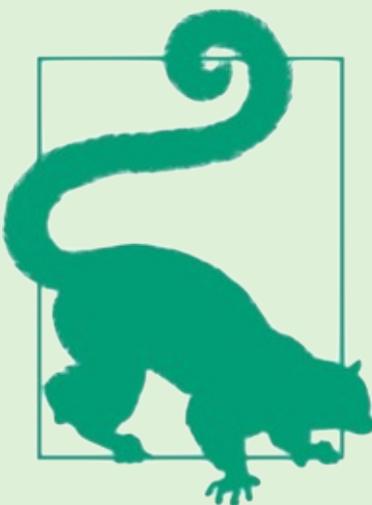
图1-3. 专门的Customer服务可以使记录客户喜欢的音乐类型变得更加容易

在图1-3的情况下，Customer服务将三层架构中的每一层都封装成很轻薄的一层——简单的UI，简单的应用逻辑和简单的数据存储。然而，所有这些层都封装在Customer服务内部。

业务领域已成为推动系统架构发展的主要力量。希望业务领域可以让功能变更更容易，同时让我们更轻松地围绕业务领域组织团队。业务领域非常重要，因此我们会在本章即将结束的时候重新讨论围绕领域建模的概念。我会分享一些关于领域驱动设计的想法，这些想法将影响我们对微服务架构的看法。

# 不要共享数据库

关于微服务，我所能见到的、人们最难应对的事情之一是：微服务不应该共享数据库。如果一个服务要访问另一服务所拥有的数据，则该服务应该通过访问另一个服务以获取所需的数据。这种方式使服务可以确定：共享哪些内容，隐藏哪些内容。服务的内部实现细节可能会出于各种原因而修改。禁止共享数据库还允许服务将其内部的实现细节映射到更稳定的公共契约，从而确保稳定的服务接口。如果需要独立部署的特性，则服务之间必须具备稳定的接口。服务公开的接口不断变化将产生连锁反应，这会导致其他的服务也需要进行变更。



除非必须，否则不要共享数据库。即便如此，也要尽一切可能避免共享数据库。在我看来，如果要实现独立部署，共享数据库是最糟糕的事情之一。

正如我们在[针对业务领域建模](#)一节中讨论的那样，我们希望将服务视为业务功能的端到端切片（*end-to-end slices*）：在适当的地方封装UI，应用程序逻辑和数据存储。这是因为我们要减少为修改业务相关功能所付出的精力。以这种方式封装数据和行为，可以使我们的业务功能具备高度的内聚。隐藏服务背后的数据库，还可以确保降低服务耦合。我们稍后会继续讨论耦合和内聚。

很难做到不共享数据库，特别是当我们当前的单体系统拥有一个庞大数据库时。幸运的是，我们会用整个第4章来讨论如何拆分单体数据库。

共享数据不仅局限于业务单元之间对同一数据库的共享，还存在于交付团队的不同角色之间。例如：研发和测试之间。

2014年的时候，我负责一款App的测试工作。为了提升测试的效率，对于server端的测试而言，采用接口测试的方案来测试。那时候，研发是通过一个本地的lib库来进行MySQL数据的读写，而测试如果要访问数据就必须将研发代码中的lib库迁移到测试代码中，然后根据MySQL的部署情况来修改lib库的配置，从而使不同的人员可以访问自己的数据库。虽然每次运行这些case的时候会提示：确认数据库配置是否正确。但是糟糕的是，这种提示很少有人关注，并且更糟糕的是，case的运行经常因为这种原因而失败。

后来，我提出了一个一切皆接口的方案：在server端提供一个接口来封装本地的lib库。所有的测试case通过封装的数据接口来获取数据库中的数据。事实证明，这种方式实在是一种完美的方案。

# 微服务能够带来什么优势

微服务拥有多种多样的优势。独立部署为改善系统的规模和鲁棒性开辟了新的模型，并允许我们混搭不同的技术。由于微服务之间可以并行工作，因此我们可以投入更多开发人员以解决一个问题。同时，微服务还能保证这些开发人员的工作不会互相干扰。开发人员可以将精力关注在系统的一部分上，因此，开发人员可以更轻松地了解他们所关注的那部分系统。程序隔离还可以扩展我们可以选择的技术——混合使用不同的编程语言、编程样式、部署平台或数据库，以便找到合适的组合。

综上所述，微服务架构可以为我们提供灵活性。微服务还为我们将来如何解决问题提供了更多选择。

不过，务必注意，如上的这些优势都不是免费的。可以采用多种方法拆解系统，并且从根本上讲，我们要实现的目标会以不同的方向推动系统拆解。因此，了解我们试图从微服务架构中获得什么非常重要。

# 微服务会产生什么问题

计算机的价格不断降低，因此我们可以拥有更多的计算机，这成为面向服务的架构（*SOA: service-oriented architecture*）成为现实的部分原因。与其将系统部署在单台大型机上，还不如将系统部署在多台较为便宜的机器。*SOA*是解决如何才能最好地构建跨多台计算机的应用程序的一种尝试。其中，最主要的挑战之一就是网络问题——计算机之间相互通信的方式。

网络之中的计算机之间的通信并不是即时通信。这意味着我们必须考虑网络通信的延迟（*latency*），尤其是当网络延迟远远超过本地进程内操作的延迟时，我们必须更加关注网络延迟。当网络延迟并不是固定数值时，情况会变得更糟。不固定的网络延迟导致系统行为变得不可预测。同时，我们还必须解决偶尔发生的网络通信失败的现实场景，例如：丢包，网络电缆中断。

网络的挑战使得单进程的单体应用的行为会相对简单。另外，类似事务这样的挑战也会让微服务系统更加难以处理。实际上，是如此的困难，以至于随着系统复杂性的增加，我们可能不得不放弃事务及其带来的安全性，以采用其他的技术（不幸的是，这些技术和事务之间的取舍非常不同）。

处理任何网络调用都可能会失败这样的事实是一件令人头疼的事。同时，我们与之通信的服务可能由于某种原因而无法访问或出现异常的事实同样令人头疼。除此之外，我们还需要开始尝试研究：如何在多台计算机上保证数据的一致性。

当然，如果使用的好，微服务这种新技术可以带来巨大的收益。但是，如果使用不当，新技术就会帮助我们更快的以更有趣、更昂贵的方式犯错。老实说，除了微服务可以带来的好处外，微服务似乎是一个可怕的东西。

值得注意的是，几乎所有的、我们认为是单体应用的系统都是分布式系统。从数据库中读取数据并将数据显示在Web浏览器的单进程应用程序可能会运行在不同的机器。此时，至少有三台通过网络进行通信的计算机。分布式单体系统和微服务架构的区别在于其分布式的程度。当拥有更多计算机，并通过更多网络进行通信时，我们更有可能遇到与分布式系统相关的棘手问题。此处简要讨论的这些问题最初可能不会出现。但是随着时间的流逝，随着系统的发展，即使我们不会遇到所有的问题，也会遇到其中的大多数问题。

正如我的老同事、老朋友、微服务专家James Lewis所说：“Microservices buy you options”。James的话是经过深思熟虑的——they buy you options。微服务是有成本的，因此对于我们的选择而言，我们必须确定该成本是否是值得的。关于微服务成本的话题，我们将在第2章中更详细地探讨。

曾几何时，我也简单的认为之前采用的分布式的系统就是微服务的架构。直到读了本书，了解到微服务的**独立部署**特性之后，才发现了微服务和之前的分布式单体系统的区别。

# 用户界面的微服务

在拥抱微服务架构时，我经常看到，人们将工作重点完全放在服务器端，从而使得UI还保留在一个单一的单体层中。如果我们想要一种能够更轻松地、更快地部署新功能的架构，那么将UI保留为单体可能是一个很大的错误。我们可以，也应该考虑拆分UI。我们将在第3章中探讨拆解UI的技术。

# 微服务的相关技术

掌握一整套新技术来配合全新的微服务架构实在是太诱人了，但是我奉劝大家不要陷入这种诱惑之中。采用任何新技术都会有成本，新技术还将带来一些动荡。但愿，这些成本还是值得的（当然，如果选择了正确的技术）。但是初次采用微服务架构时，我们掌握的技术已经足够了。

微服务架构的正确发展和管理涉及到解决与分布式系统相关的众多挑战，这些挑战可能是我们以前从未遇到过的挑战。我认为，在遇到这些问题时，如下的做法会更加有用：

1. 利用我们熟悉的技术栈及时解决这些问题
2. 然后考虑改变现有的技术是否可以帮助解决我们发现的问题

正如我们已经谈到的那样，究其根本而言，微服务与技术无关。只要服务可以通过网络相互通信，其他所有的技术都可以缓缓图之。微服务具有巨大的优势——他允许我们可以根据需要来混搭技术堆栈。

不必使用Kubernetes，Docker，容器或公有云。更无需使用Go，Rust或其他任何新的编程语言来编码。实际上，就微服务架构而言，我们所选择的编程语言并不重要。对于微服务架构而言，不同编程语言的差异仅在于某些语言可能具有更丰富的生态系统——支持库和框架。如果最了解PHP，就用PHP开始构建服务！<sup>1</sup>莫让方案的选择成为问题的一部分！选择最适合我们的方案，并在遇到问题时做出变更以解决问题。

# 微服务的规模

我最经常遇到的问题是：一个微服务的规模应该控制在多大范围？因为从术语“微服务（*micro-services*）”来看，“micro”就包含在名称中，因此讨论微服务的规模也就不足为奇。但是，一旦了解了是什么使得微服务可以作为一种架构而工作，微服务规模的概念实际上就会成为最不应该关心的问题之一。

如何来度量微服务的规模呢？使用代码行数？代码行数的度量方式有时并没有意义。25行的Java代码实现的功能对于Clojure而言，可能仅需要10行就可以实现。但是，这并不意味着Clojure好于/差于Java，只能说明相比于其他语言，有些语言更富于表达性。

我能想到的、“规模”一词对微服务而言有任何意义的、最接近的描述是微服务专家Chris Richardson曾经说过的——微服务的目标是“具有尽可能小的接口”。此时规模的含义更接近信息隐藏的概念（我们稍后将会讨论），但也确实代表了在事实之后寻找意义的尝试——当我们第一次谈论这些东西时，我们的主要关注点，至少是最初的关注点，是这些服务真的很容易替换。

最终，“规模”的概念与上下文高度相关。对于在某个系统上工作了15年的人而言，他们会觉得他们的100K行代码的系统非常容易理解。然而，对于该项目的新手而言，他们会觉得100K行代码的系统太大了。同样的，如果询问一家刚刚开始微服务转型的公司（该公司可能拥有少于十个微服务），与多年以来一直采用微服务的相同规模的公司相比（该公司现在已经拥有数百个服务），我们可能会得到不同的答案。

我奉劝人们不要担心规模。刚开始时，专注于两个关键问题尤为重要：

- 首先，我们可以处理多少个微服务？随着我们拥有的服务不断增加，系统的复杂性将随之增加。同时我们还必须学习新技能（也许采用新技术）来应对系统不断增加的复杂性。因此，我坚决主张逐步迁移到微服务架构。
- 其次，如何定义微服务边界以充分利用业务的拆解，而又不会因为业务拆解让所有事情变得一团糟？

我们会在本章的其余部分介绍如上的主题。

### “微服务”一词的历史

早在2011年，当我在ThoughtWorks咨询公司工作时，我的朋友、当时的同事**James Lewis**就对他所谓的“micro-apps”非常感兴趣。他发现有些使用SOA的公司已经开始使用这种模式，并且这些公司正在优化这种“micro-app”架构以使服务易于替换。上述被提及的公司都对快速部署特定功能感兴趣，但是他们认为，如果需要扩展所有功能，可以用其他技术栈将其重写。

当时突出的是这些服务的范围有多小。在几天内就可能实现（或重写）其中的某些服务。James继续说：“services should be no bigger than my head.”，其思想就是：功能范围应该易于理解，因此才会易于改变。

后来，在2012年，James在一次架构峰会上分享了这些想法。我们当中有些人也出席了那次架构峰会。在那次会议上，我们讨论了以下事实：实际上这些所谓的“micro-apps”并不是独立的应用程序，因此“micro-apps”的说法并不完全正确。相反，“microservices”这种称呼似乎更合适。<sup>2</sup>

# 微服务所有制

利用围绕业务领域建模的微服务，我们可以看到：IT产物（可独立部署的微服务）与业务领域之间是一致的。当我们考虑转变技术公司从而打破“业务”和“IT”之间的鸿沟时，微服务这一思想会引发我们的共鸣。如图1-4，在传统的IT组织中，软件开发的行为通常和业务部门（业务部门是连接客户，并定义需求的部门）完全分离。这类组织的功能失调是多种多样的，因此此处无需再展开讨论。

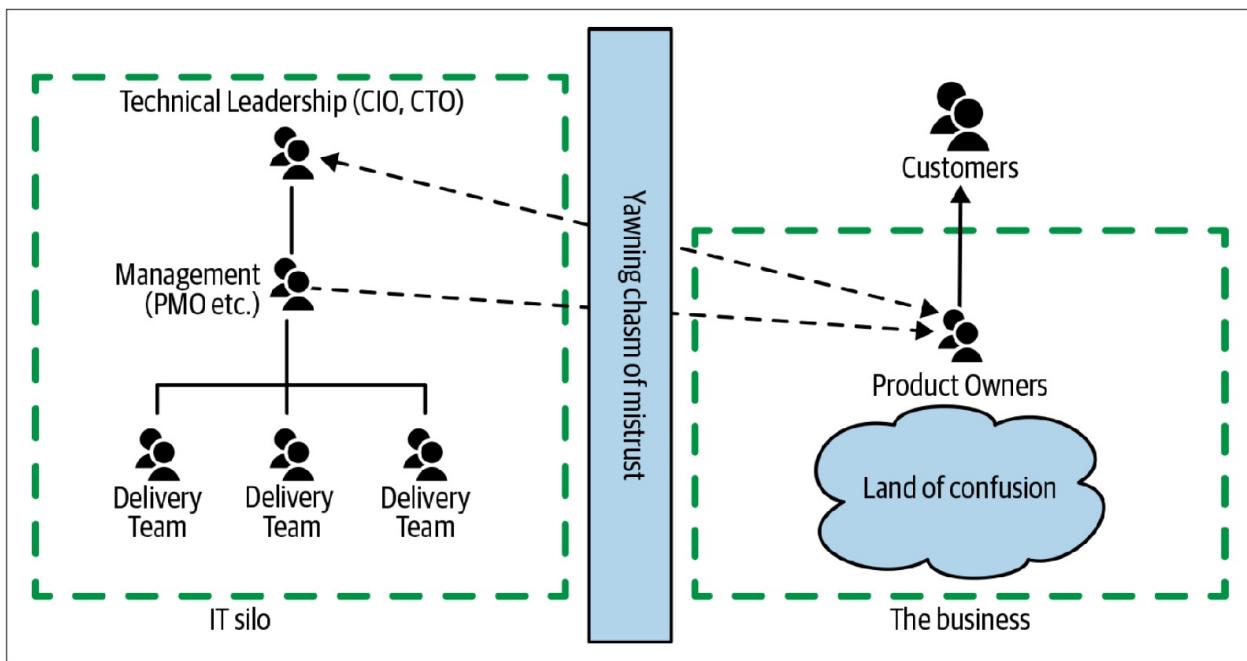


图1-4. 传统的IT/Business的划分

相反，如图1-5所示，我们看到真正的技术组织和先前的、分散的组织孤岛完全整合在一起。现在，产品负责人直接作为交付团队的一部分而工作，这些团队围绕面向客户的产品线而对齐，而不再是随心所欲的技术分组。对于图1-5而言，集中式IT部门已不是常态，任何的集中式IT部门的存在都是为了支持以客户为中心的交付团队而存在。

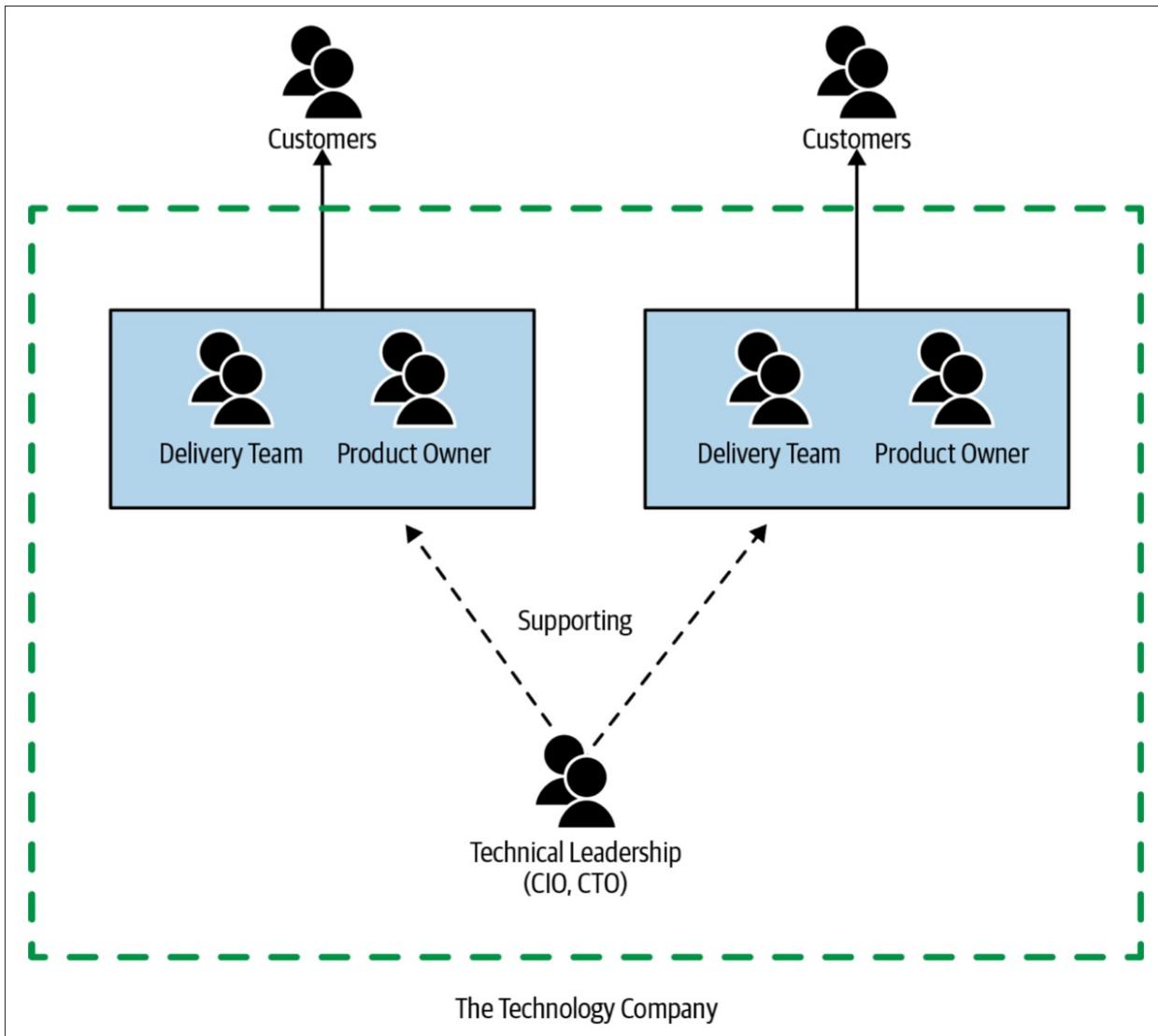


图1-5. 技术公司如何集成软件交付的例子

虽然并非所有组织都做出了这一转变，但微服务架构使转变更加容易。如果希望交付团队围绕产品系列进行调整，服务围绕业务领域进行调整，那么将所有权明确分配给这些面向产品的交付团队会变得更加容易。减少跨团队共享服务是最小化交付冲突的关键——面向业务领域的微服务架构让组织结构的转变变得更加容易。

<sup>1</sup>. For more on this topic, I recommend PHP Web Services by Lorna Jane Mitchell (O'Reilly). There is far too much technical snobbery out there toward some technology stacks that can unfortunately border on contempt for people who work with particular tools. After reading Aurynn Shaw's "[Contempt Culture](#)" blog post, I recognized that in the past I have been guilty of showing some degree of contempt toward different technologies, and by extension the communities around them. ↩

<sup>2</sup>. I can't recall the first time we actually wrote down the term, but I vividly recall my insistence, in the face of all logic around grammar, that the term should not be hyphenated. In hindsight, it was a hard-to-justify position, which I nonetheless stuck to. I stand by my unreasonable, but ultimately victorious choice. ↩

Copyrights © wangwei all right reserved

# 单体应用

我们已经介绍过微服务的相关内容，但是本书讨论的是：从单体迁移到微服务。因此，我们还需要确定“单体”的含义。

当我在本书中谈到“单体”时，我主要指的是一个部署单元。当系统中的所有功能必须一起部署时，将其视为一个单体。符合这一要求的单体系统至少有三种类型：单进程（*single-process*）系统<sup>译注1</sup>，分布式单体（*distributed monolith*）系统和第三方黑盒（*third-party black-box*）系统。

# 单进程的单体应用

讨论单体时，能想到的、最常见的例子是这样一个系统：所有的代码都作为程序而部署，如[图1-6](#)所示。我们可能会拥有该程序的多个实例以保证程序的鲁棒或扩展。但是，从根本上讲，所有的代码都打包在一个程序中。这些单进程系统几乎总是从数据库中读取数据或将数据存储到数据库。因此，实际上，他们本身就可以构成简单的分布式系统。

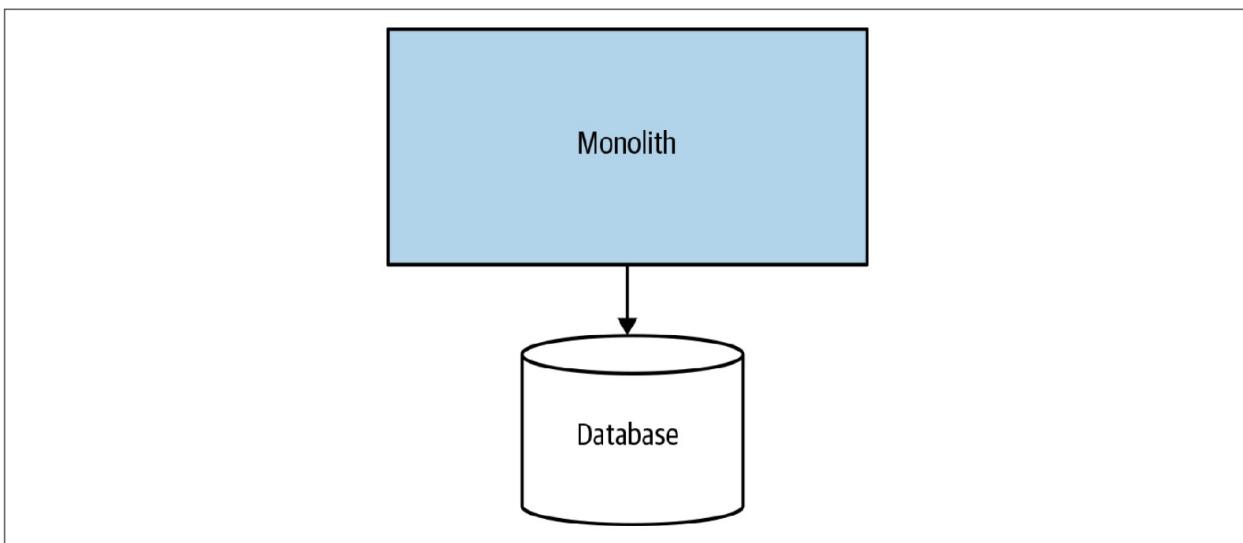


图1-6. 单进程单体应用：所有的代码都被打包进一个程序

我所能看到的、人们与之斗争的单体系统，大多数都是属于这种单进程的单体系统。因此，我们会花费大部分时间来重点关注单进程的单体系统。从现在开始，除非另有说明，“单体”的含义均为这种单进程的单体系统。

## 模块化单体

模块化单体是单进程单体的一种变体：单个程序由独立的模块组成，每个模块都可以独立工作，但仍需要组合到一起才能部署，如图1-7所示。把软件拆分为模块的概念并不新鲜，我们会在本章后面再回顾模块化的一些历史。

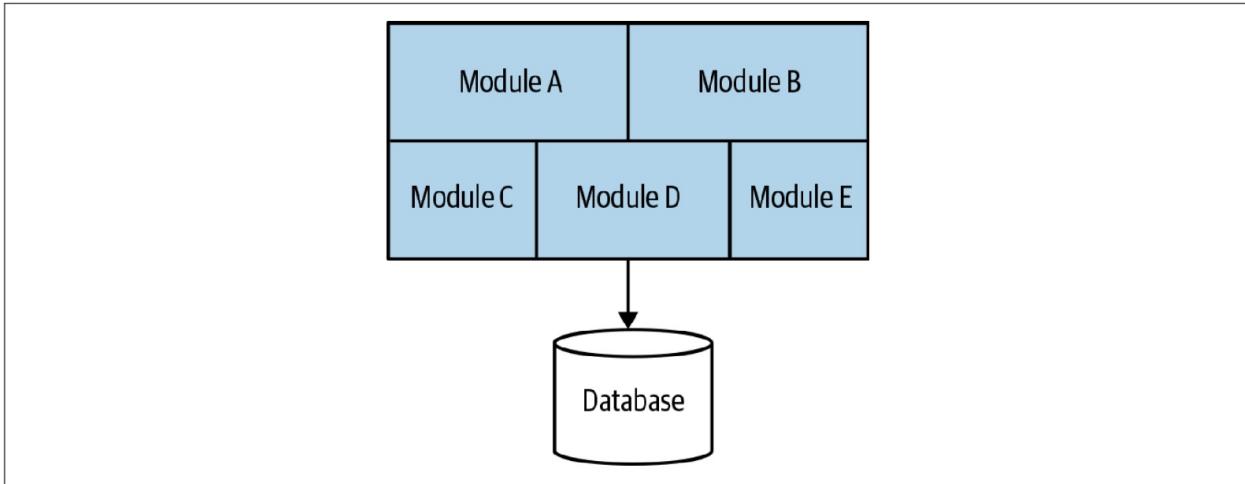


图1-7. 模块化的单体结构：程序中的代码被拆分为不同的模块

对于很多组织而言，模块化单体是一个不错的选择。如果模块的边界定义良好，则可以进行高度并行的工作。不过，模块化单体在避免更加分布式的微服务架构所带来的挑战时，也同时失去了微服务部署更简单的优势。[Shopify](#)是使用模块化单体技术来代替微服务架构的一个很好的例子，并且对于该公司而言，模块化的单体架构看起来运作良好。<sup>3</sup>

模块化单体的挑战之一是：数据库往往很少在代码层面拆分。这会导致，如果将来想要取消单体架构时，我们可能会面临重大挑战。我看到过有些团队试图通过把数据库拆分到和模块对齐的方式，来进一步推动模块化单体，如图1-8所示。从根本上讲，即使我们可以采用图1-8的方式让代码独立，但是对现有的单体进行图1-8所示的改造仍然具有很大的挑战性。如果想尝试做类似的事情，在第4章中探讨的许多模式可以为我们提供帮助。

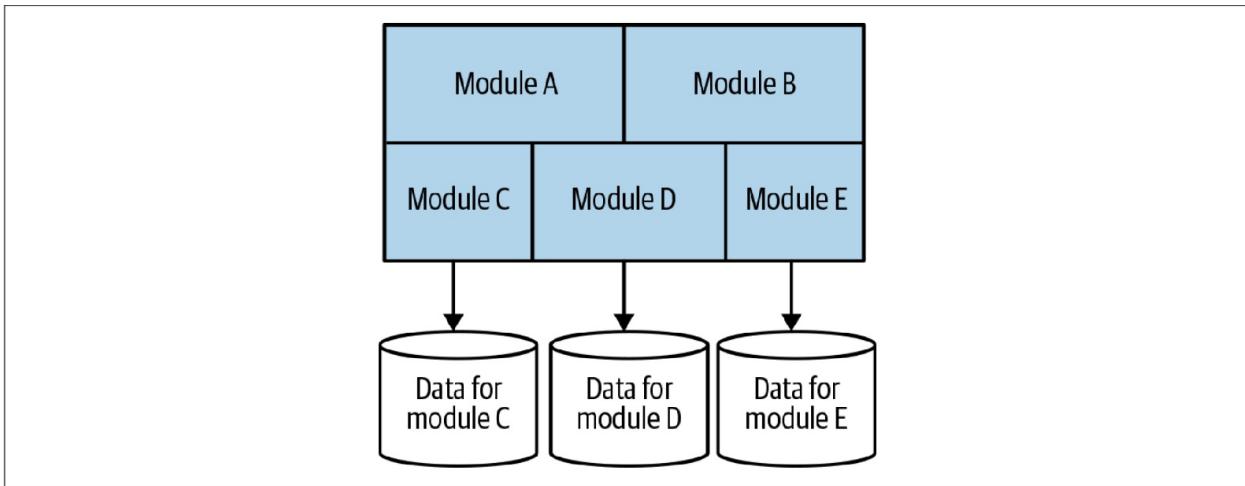


图1-8. 具有数据库拆分的模块化单体

# 分布式的单体应用

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. (不知道哪个角落的某台计算机发生了故障，导致了我们自己的电脑无法使用，这就是分布系统。)<sup>4</sup>, [译注2](#)

——Leslie Lamport

分布式单体是由多个服务构成的系统，但是无论出于何种原因，整体系统都必须一起部署。分布式单体很好地满足了SOA的定义，但是通常都无法兑现SOA的承诺。以我的经验而言，分布式单体结构具有分布式系统的所有缺点，同时还具有单进程单体的缺点，但是其既没有足够的分布式系统的优势也没有足够的单进程单体的优势。我在工作中遭遇到的分布式单体很大程度上影响了我对微服务架构的兴趣。

分布式单体通常出现在这样的环境：没有将足够的精力放在诸如信息隐藏和业务功能内聚之类的概念上，于是导致了架构的高度耦合。在这种架构中，某个服务的变化经常会如“以石击水，不起浪花，也泛涟漪”一样影响到其他的服务，看似毫无问题的、服务的本地范围内的变化，却会破坏系统的其他部分。

# 第三方的黑盒系统

我们还可以把某些第三方软件视为单体，我们可能希望在迁移工作中对其进行“分解”。这种第三方软件可能包括：工资管理系统，CRM系统和HR系统。这些软件的共性是：软件由其他人开发，我们无法对其代码进行更改。第三方软件可能是在我们自己的基础架构上部署的现成（*off-the-shelf*）软件，也可能是我们正在使用的某种SaaS产品。我们在本书中探讨的许多分解技术也可以应用于那些无法更改其底层代码的系统。

# 单体应用的挑战

无论是单进程单体还是分布式单体，单体通常更容易遭受耦合的风险，特别是实现（*implementation*）耦合和部署（*deployment*）耦合。我们稍后就会讨论耦合的相关主题。

随着越来越多的人在同一个地方工作，他们之间的工作就会出现相互阻碍的情况。不同的开发人员想要更改同一段代码，不同的团队想要在不同时间发布功能（或延迟部署）。谁拥有什么、谁要做出决定的事情就会纠缠不清。大量的研究已经表明了权责混乱带来的挑战。<sup>5</sup> 我把这个问题称之为交付冲突。

拥有单体并不意味着我们一定会面临交付冲突的挑战，但是微服务架构却意味着我们永远不会面临交付冲突的问题。微服务架构确实为我们提供了可以划定系统不同部分的权责的具体界限，从而为减少交付冲突提供更大的灵活性。

# 单体应用的优势

但是，单进程单体也有很多优点：

- 更简单的部署拓扑可以使其避免分布式系统下的许多陷阱
- 简化开发人员的工作流程
- 简化监控、故障排除和诸如端到端测试之类的行为

单体还可以简化单体内部的代码复用。如果要在分布式系统中复用代码，我们则必须决定采用哪种复用方式：是复制代码，还是抽取为库，亦或把公共的功能置于某个服务。对于单体而言，选择就简单得多了。很多人都喜欢单体的这种简单性——所有的代码都在这里，所以就使用它吧！

不幸的是，因为单体架构所固有的问题，人们开始把单体视为一种应该避免其发生的事情。我遇到过各种各样的人，他们都把“单体”称为“遗留代码”的代名词。这种想法是个问题。单体架构是一种选择，而且是一种有效的选择。单体并非在所有情况下都是正确的选择，微服务也并非适用于所有的情况，但是单体仍然是一种选择。如果我们陷入如下的陷阱：通过系统性抹黑单体来作出交付软件的可行选择，那么我们会面临没有正确的做事的风险，也会让我们的用户面临没法正确使用我们的软件的风险。我们将在第3章中进一步探讨单体和微服务的权衡，并讨论一些工具，用以帮助我们更好地评估在我们的场景下哪种架构更适合我们。

---

<sup>3</sup>. For an overview of Shopify’s thinking behind the use of a modular monolith rather than microservices, Kirs- ten Westeinde’s [talk on YouTube](#) has some useful insights. ↪

<sup>4</sup>. see <https://www.microsoft.com/en-us/research/publication/distribution/> for more. ↪

<sup>5</sup>. Microsoft Research has carried out studies in this space, and I recommend all of them. As a starting point, I suggest “[Don’t Touch My Code! Examining the Effects of Ownership on Software Quality](#)” by Christian Bird et al. ↪

译注<sup>1</sup>. 单进程系统并不意味该系统只有一个进程在工作。 ↪

译注<sup>2</sup>. Leslie Lamport是在计算机领域非常多才多艺的一个科学家。他在1978年发表的关于分布系统时序的一篇论文是被引用最多的文章之一，他在1982年描述的拜占庭将军问题成为阐述分布系统协商机制的经典案例，他在1985年推出的LaTeX语言成为学术界最流行的排版语言，他在1990年设计的行为时序逻辑语言（TLA）为并发事件推演提供了数学工具。分布式系统领域著名的Paxos算法也是由Leslie Lamport发明的。 ↪

Copyrights © wangwei all right reserved

# 关于耦合和内聚

在定义微服务边界时，了解耦合和内聚之间的平衡非常重要。耦合说明改变一件事情时需要改变另一件事情。内聚说明了我们如何组织相关代码。耦合和内聚这两个概念直接关联。Constantine的法则很好地阐明了耦合和内聚的关系：

A structure is stable if cohesion is high, and coupling is low. (如果一个结构具备高内聚、低耦合的特性，则该结构是稳定的。)

——Larry Constantine

高内聚、低耦合看似是一个明智、有用的观点。如果我们有两段紧密相关的代码，因为相关功能分散在这两者之间，因此是低内聚的。同时，因为当更改相关代码时，这两段代码都需要更改，因此是高耦合的。

因为跨分布式系统的服务边界进行更改的成本非常高，因此，如果代码系统的结构发生变化，则会非常麻烦。此时，必须在一个或多个可独立部署的服务上进行更改，或许还要处理修改服务契约带来的影响，这些都可能是变更的巨大阻力。

单体结构的问题在于，单体对内聚和耦合而言是相反的（低内聚，高耦合）。单体架构不是倾向于内聚，而是倾向于把变化的代码组织在一起，获取各种不相关的代码并将其拼贴在一起。同样，对于单体而言，低耦合实际上并不存在。如果想改一行代码，可能很容易地就能够修改，但是却不能在不影响单体的其他部分的情况下部署变更，必须重新部署整个系统才能部署那一行代码的变更。

我们的目标是尽可能地拥抱独立部署的概念，也就是说，我们希望无需改动其他的任何东西就可以更改我们的服务并将其部署到生产环境。因此，我们对系统的稳定性有一定要求。为此，我们要求我们所依赖的服务具备稳定性，并且要为依赖我们的服务提供提供稳定的服务契约。

关于耦合和内聚已经有大量的信息，所以在这里过多地重复这些内容并不明智，但是我认为应该对其进行总结，特别是把这些想法置于微服务架构的上下文中。内聚和耦合的概念会极大地影响我们对微服务架构的看法。这不足为奇：内聚和耦合是模块化软件所关注的。那么，除了通过网络通信并可以独立部署的模块以外，微服务架构还是什么呢？

## 耦合和内聚简史

在计算领域，内聚和耦合的概念已经存在很长时间了。内聚和耦合的概念最初由Larry Constantine在1968年提出。这对概念构成了如何编写计算机程序的许多思想的基础。Larry Constantine和Edward Yourdon的**Structured Design(Prentice Hall, 1979)**等书籍随后影响了几代程序员（我本人在攻读大学学位期间，是要求必需阅读该书的，该书距其首次出版已有近20年的时间）。

Larry在1968年（对计算机领域而言，这是特别幸运的一年）的全国模块化编程研讨会（*National Symposium on Modular Programming*）上首次提出了内聚和耦合的概念。在该会议上还首次提出了康威定律。1968年，还举办了两次北约软件工程大会（*infamous NATO-sponsored conferences*）[译注1](#)，并且在会议上提出了软件工程的概念（软件工程是Margaret H. Hamilton在之前创造的一个术语）。

# 内聚

我听说过的描述内聚最简洁的定义之一是：“代码一起变化，并保持在一起。”就我们的目的而言，这是一个很好的定义。正如我们已经讨论过的那样，我们正在围绕简化业务功能变更的过程来优化微服务架构，因此我们希望将功能分组，以便可以在尽可能少的地方进行更改。

如果我想改变发票审批的管理方式，则不必在多个服务之间寻找要进行更改的功能，然后再协调发布这些新更改的服务以推出我们的新功能。相反，我想确保更改仅涉及修改尽可能少的服务，以保持较低的变更成本。

# 耦合

像节食一样，信息隐藏是一件说起来容易做起来难的事情。

——David Parnas, *The Secret History Of Information Hiding*

我们喜欢我们喜欢的内聚，但是我们对耦合却很谨慎。“耦合”的事情越多，必须一起变更的事情就越多。然而，耦合有很多不同的类型，并且每种类型可能需要不同的解决方案。

有很多现有技术可以用于耦合类型的分类，特别是Myers, Youdan和Constantine所做的工作<sup>译注2</sup>, <sup>译注3</sup>。对于耦合类型而言，我会在本书提出我自己的观点，但这并不是说该领域以前所做的工作是错误的。之所以提出我的观点，仅仅是因为我发现这种分类更有助于人们理解分布式系统耦合的相关内容。因此，我在本书提出的耦合类型的目标并不在于对不同形式的耦合进行详尽的分类。

## 信息隐藏

在有关耦合的讨论中，会不断的出现一个概念：信息隐藏技术。信息隐藏技术最初由David Parnas在1971年提出。信息隐藏技术也是David Parnas在研究如何定义模块边界工作中的成果。<sup>6</sup>

信息隐藏的核心思想是将经常更改的代码与不会发生更改的代码区分开来。我们希望模块边界是稳定的，其应该隐藏那些希望经常变化的模块的实现部分。也就是说，只要保持模块的兼容性，就可以安全地修改模块内部。

就我个人而言，我采用的方式是：尽可能少的公开模块（或微服务）边界。一旦某些东西成为模块接口的一部分，再将其从模块接口中移除就非常困难了。但是，如果现在将其隐藏，则日后就可以随时决定将其共享。

作为面向对象（OO）软件的一个概念，封装（*Encapsulation*）会与信息隐藏联系在一起。但是根据我们所接受的封装的定义，封装和信息隐藏可能不是一回事。OOP中的封装意味着将一个或多个事物捆绑到一个容器中——想想类的字段和作用于这些字段的方法。然后，可以在类定义中使用可见性标记来隐藏类的部分实现。

我推荐阅读Parnas的“The Secret History of Information Hiding”，来更深入地了解信息隐藏的历史。<sup>7</sup>

## 实现耦合

实现耦合(*implementation coupling*)通常是我所看到的、最有害的耦合形式。幸运的是，实现耦合通常是最容易解决的耦合形式。对于实现耦合而言，A与B耦合于B的实现——当B的实现发生变化时，A也会变化。

这里的问题是，实现细节通常是开发人员的任意选择。解决问题有很多方法，我们会从中选择一个方法，但我们可能会改变主意。但是，当我们决定改变主意时，我们不希望影响依赖该模块的服务（还记得独立部署吗？）。

实现耦合的经典又常见的例子是以共享数据库的形式出现的。在图1-9中，Order服务包含了系统中所有订单的记录。Recommendation服务根据顾客的购买历史向顾客推荐其可能要购买的唱片。当前，Recommendation服务会直接从数据库访问订单数据。

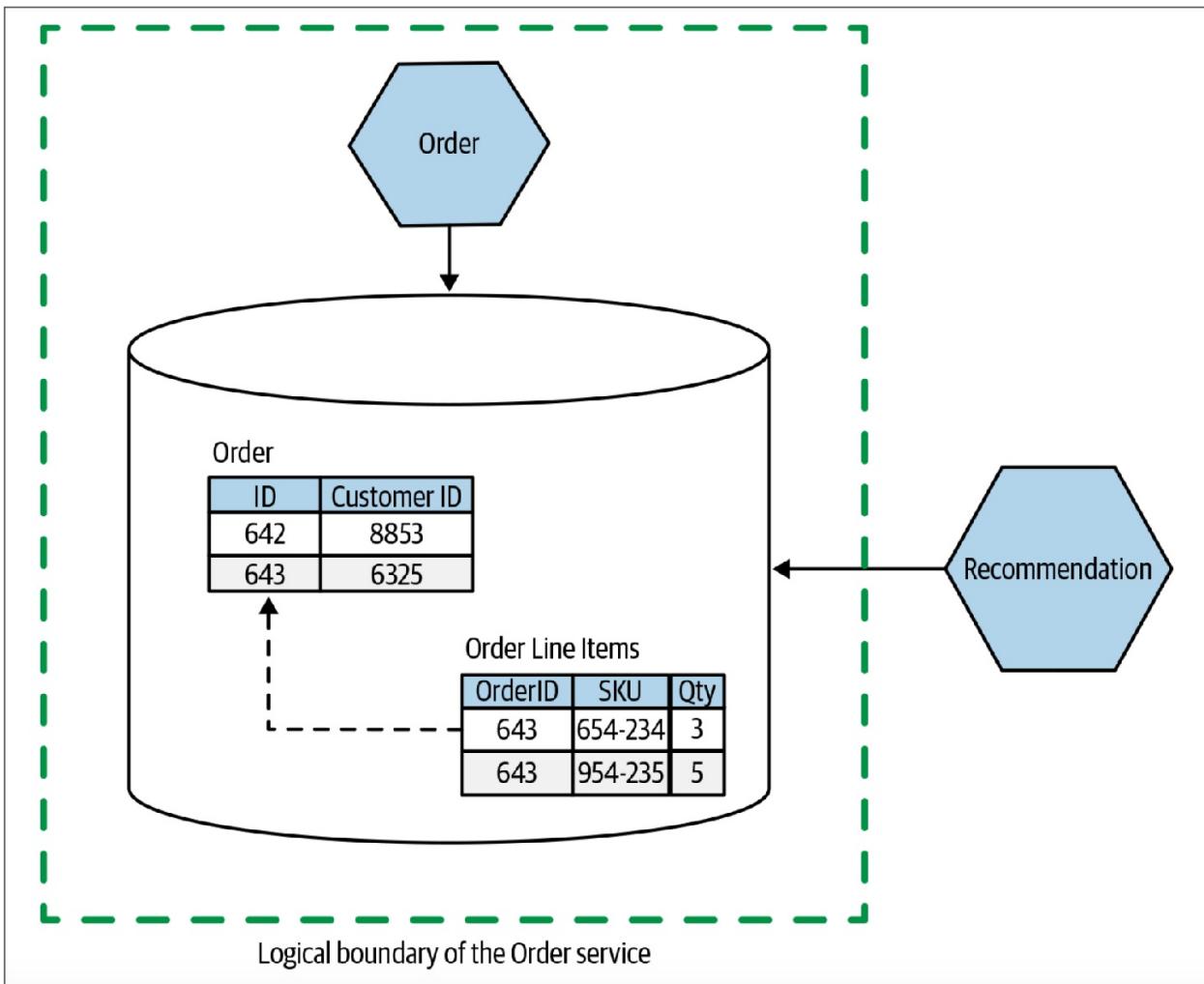


图1-9. Recommendation服务直接访问Order服务存储的数据

Recommendation服务需要有关已下订单的信息。我们[稍后会讨论](#)，在某种程度上，这是不可避免的领域耦合。但是在图1-9这种特殊的情况下，我们会耦合到特定的数据库模式结构（*schema structure*），SQL语言，甚至记录的内容。如果Order服务修改了列的名称，拆分了“Customer Order”表或执行了其他任何操作，那么从概念上讲，该服务仍包含订单信息。但是，此时，我们却破坏了Recommendation服务获取订单信息的方式。更好的选择是隐藏获取订单信息的实现细节，如图1-10所示——此时，Recommendation服务通过API调用访问其所需的信息。

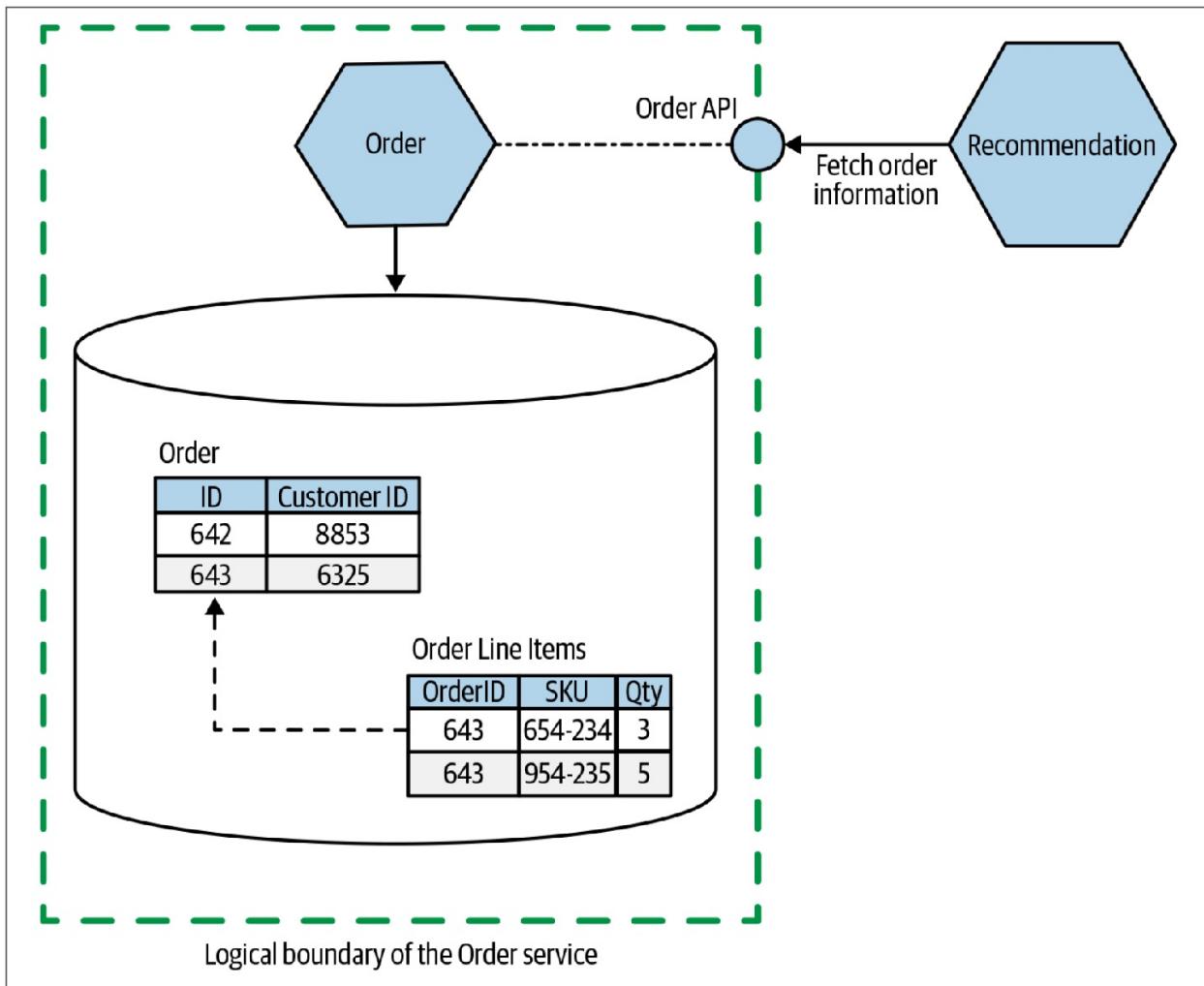


图1-10. 隐藏内部的实现细节，Recommendation服务通过API访问订单信息

如图1-11所示，我们还可以让Order服务以数据库的形式发布数据集，该数据集的目的在于供其他服务批量访问。该数据集维护Order和其他服务的公共契约，Order服务只需要发布相应数据，而其服务内部所做的任何更改对于其他服务而言都是不可见的。这种方式也为改善面向消费者的数据模型提供了机会，同时为适应消费者的需求提供了机会。我们将在第3章和第4章中更详细地探讨这种模式。

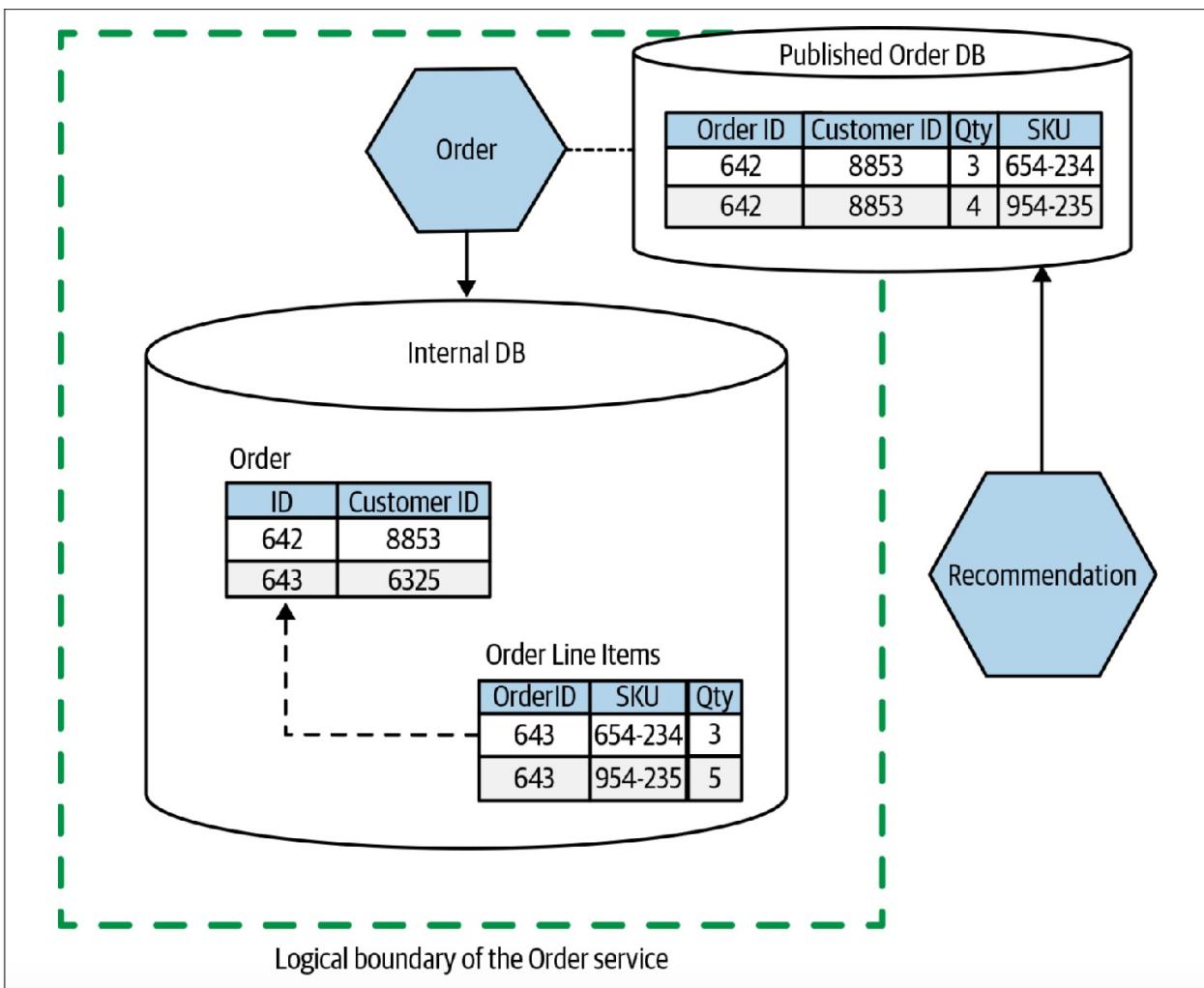
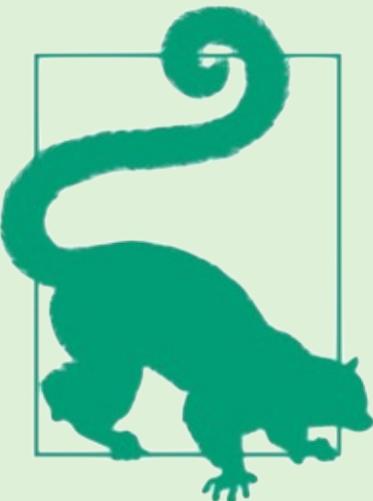


图1-11. Recommendation服务通过公开的数据库访问订单信息，该数据库的结构与Order服务的内部数据库不同

实际上，在上述两个例子中，我们都在使用信息隐藏。将数据库隐藏在定义良好的服务接口后面的行为使服务可以限制公开内容的范围，并可以使我们修改此数据的表示方式。

另一个有用的技巧是在定义服务接口时使用“由外而内”的思想——首先从服务使用者的角度考虑事情，从而驱动服务接口的设计，然后再找出如何实现该服务契约的方法。另一种方法的做法恰恰相反，不幸的是，我已经观察到该方法被普遍使用。团队采用数据模型或其他的内部实现细节开发服务，然后考虑将其公开给外界。

在“由外而内”的思想下，我们首先要问：“服务的消费者需要什么？”我并不是说我们要问自己：我们的客户需求是什么。我的意思是：我们要咨询那些调用我们服务的人。



把微服务公开的服务接口看作用户界面（*user interface*）。使用“由外而内”的思维与将要调用我们的服务的人员合作以设计服务接口。

把服务与外界的契约视为用户界面。在设计用户界面时，询问用户他们想要什么，并与我们的用户一起迭代设计。在制定服务契约时，也应该采用相同的方式。这意味着我们最终将得到更易于消费者使用的服务，除此之外，这还有助于使外部契约与内部实现之间保持一定的隔离。

## 时间耦合

时间耦合（*temporal coupling*）主要是运行时的问题，通常是分布式环境中同步调用的主要挑战之一。当发送一条消息，处理该消息的方法是以时间连接起来时，我们称之为时间耦合。这听起来有些奇怪，所以让我们看一下图1-12的一个例子。

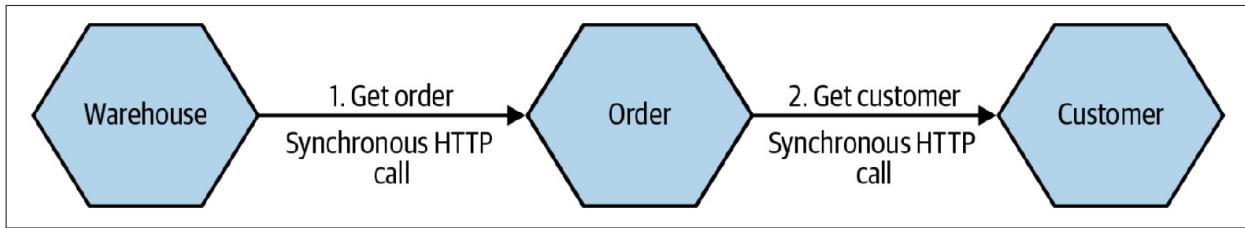


图1-12. 三个服务之间利用同步调用来执行操作的方式称为时间耦合

在这里，我们看到Warehouse服务对下游Order服务发起了同步HTTP调用，以获取有关订单的所需信息。为了实现该请求，Order服务必须再次利用同步HTTP调用从Customer服务获取信息。为了完成整个操作，Warehouse、Order和Customer这三个服务需要一起工作并保持链接。因此，这三个服务是时间耦合的。

我们可以利用各种方式来减少时间耦合的问题。可以考虑使用缓存——如果Order服务缓存了它所需要的、来自Customer服务的信息，那么在某些情况下，Order服务就能够避免与下游服务之间的时间耦合。我们也可以考虑使用异步请求，例如使用消息代理（*message broker*）之类的机制。消息代理允许把消息发送到下游服务，并在下游服务可用之后处理该消息。

对于分布式系统而言，时间耦合的另一个问题是：当同步调用的下游依赖链特别长时，则服务发生故障的可能性就会增加。因为每增加一层调用，同时增加的还有该次调用失败的可能性。

关于service-to-service的通信方式的完整探讨不在本书的讨论范围之内，但是在[Building Microservices](#)一书的第4章中有对此的更详细的介绍。

## 部署耦合

考虑一个由多个静态链接库而构成的单个程序。如果我们修改了其中某个模块的一行代码，并且想要部署该变更。为此，我们必须部署整个单体程序——甚至还包括那些未作修改的模块。一切都必须一起部署，因此，我们具有部署耦合。

就像静态链接程序的例子一样，我们可能不得不使用部署耦合，但是部署耦合也是一个选择的问题，可以选择来自实践的[火车发版模式（release train）](#)。使用火车发版模式，可以预先制定发布计划。通常，该计划可以重复使用。当发布时间点到来时，会部署自上次发布以来的所有变更。对于某些人来说，火车发版模式可能是一种有用的技术。但是，我更愿意将其视为向适当的按需发布技术发展的过渡技术，而不是将其视为终极目标。我就曾经在这种组织中工作过，作为这种火车发版过程的一部分，我们会一次部署系统中的所有服务，而不去思考这些服务是否需要变更。

部署会带来风险。有很多方法可以降低部署的风险，其中之一就是仅变更需要变更的内容。如果可以减少部署耦合，就可以通过缩减部署范围来降低每次部署的风险。或许可以通过把较大的程序拆分为可独立部署的微服务来实现这一目标。

越小的发布会越来越低的风险，同时也让我们更少的犯错。如果确实出现问题，因为我们所做的变更比较少，因此更容易找出问题所在及其解决方法。寻找减小发布量的方法是持续交付（*continuous delivery, CD*）的核心。<sup>8</sup> 持续交付也赞成快速反馈和按需发布的方法的重要性。发布范围越小，推出新功能就越容易、越安全，同时我们还将获得更快的反馈。我一直在寻找让持续交付更容易采用的架构，对微服务的兴趣也源自先前对持续交付的关注。

当然，减少部署耦合并不需要微服务。像Erlang这样的运行时允许把模块的新版本热更新到正在运行的程序。最终，我们中的更多的人可能会使用我们日常使用的技术堆栈所支持的类似能力来减少部署耦合<sup>9</sup>。

我也曾经参与过一个大型的项目，该项目涉及到数10个团队的协作。而这个项目每次新功能的上线都是一场噩梦：数十个团队的几十个研发人员在一个会议室，按照既定的依赖顺序缓慢的部署。一旦在某个地方发生了部署异常，我们所有人就会意识到，今天晚上可能没法下班了……

## 领域耦合

从根本上说，在一个由多个独立服务组成的系统中，服务之间不得不存在某些交互。在微服务架构中，服务之间的交互是对实际领域中的交互的建模，因此，领域耦合是结果。如果要下订单，则需要知道客户购物车中有哪些物品。如果要运送产品，则需要知道运送的目的地。在我们的微服务架构中，这些信息可能包含在不同的服务中。

以MusicCorp作为具体的例子。我们有一个存储货物的仓库。当用户下单购买CD时，仓库的工作人员需要了解：需要对哪些CD拣货打包，还需要了解要把包裹发送到何处。因此，订单信息需要与仓库的工作人员共享。

图1-13展示了这样的一个例子：

- Order Processing服务把订单的所有详细信息发送到Warehouse服务
- 然后Warehouse服务触发该订单商品的打包

在此期间，Warehouse服务使用客户ID从单独的Customer服务获取用户的相关信息，以便我们了解如何在订单发出时通知用户。

在这种情况下，Order Processing服务与Warehouse服务共享整个订单，这可能没有什么意义——因为Warehouse服务仅需要拣货的商品信息和收获地点的信息。Warehouse服务不需要知道商品的价格（如果需要在包裹中包含发

票，则可以提供PDF格式的发票并传递到Warehouse服务）。在必须控制信息以使其无法广泛共享时，我们也会遇到问题——如果我们共享整个订单信息，最终可能会使信用卡的详细信息暴露给不需要该信息的服务。

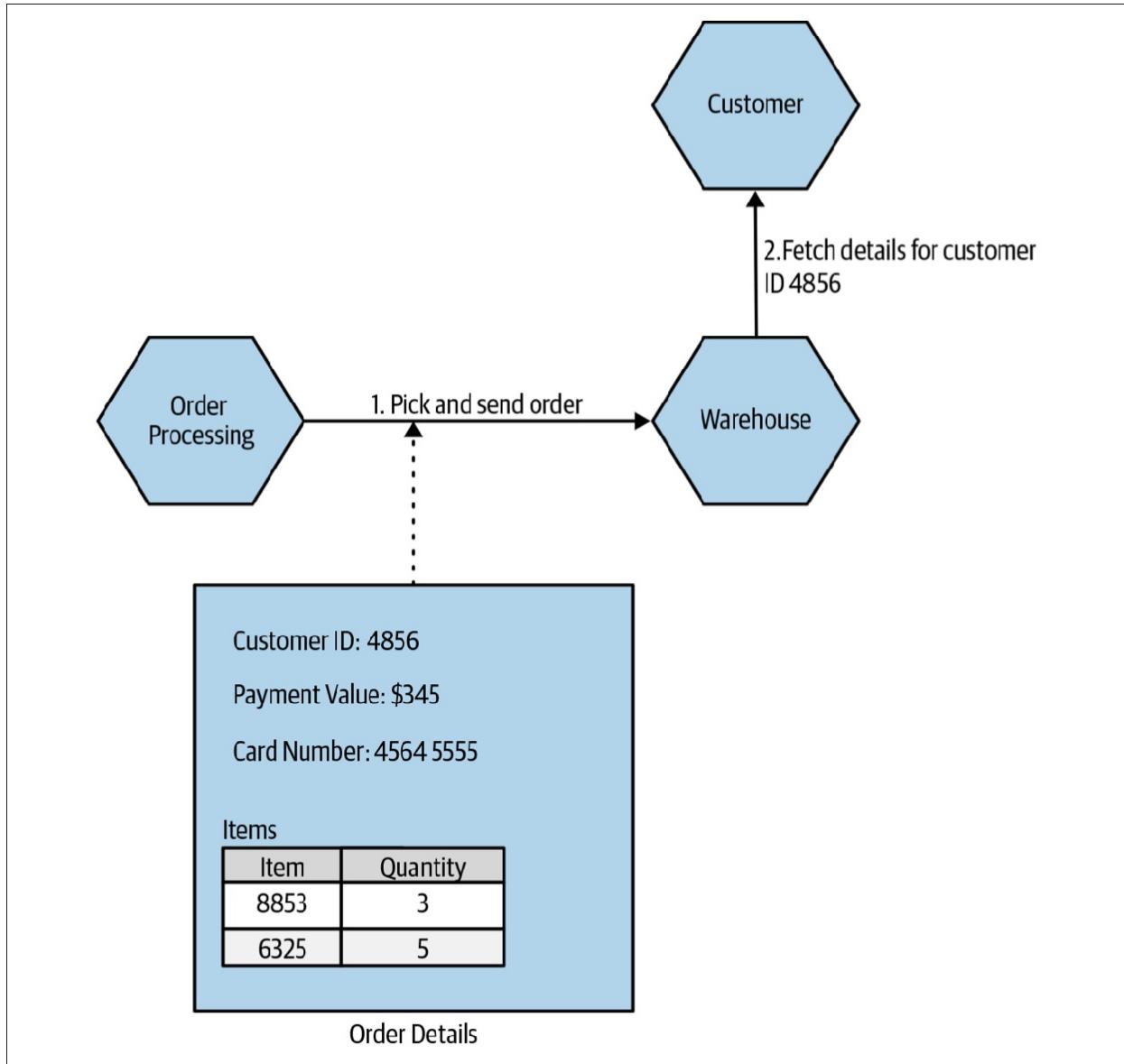


图1-13. 把订单发送到仓库以允许打包

因此，我们可能会提出一个新的领域概念——Pick Instruction，其仅包含Warehouse服务所需的信息，如图1-14所示。这是信息隐藏的另一个例子。

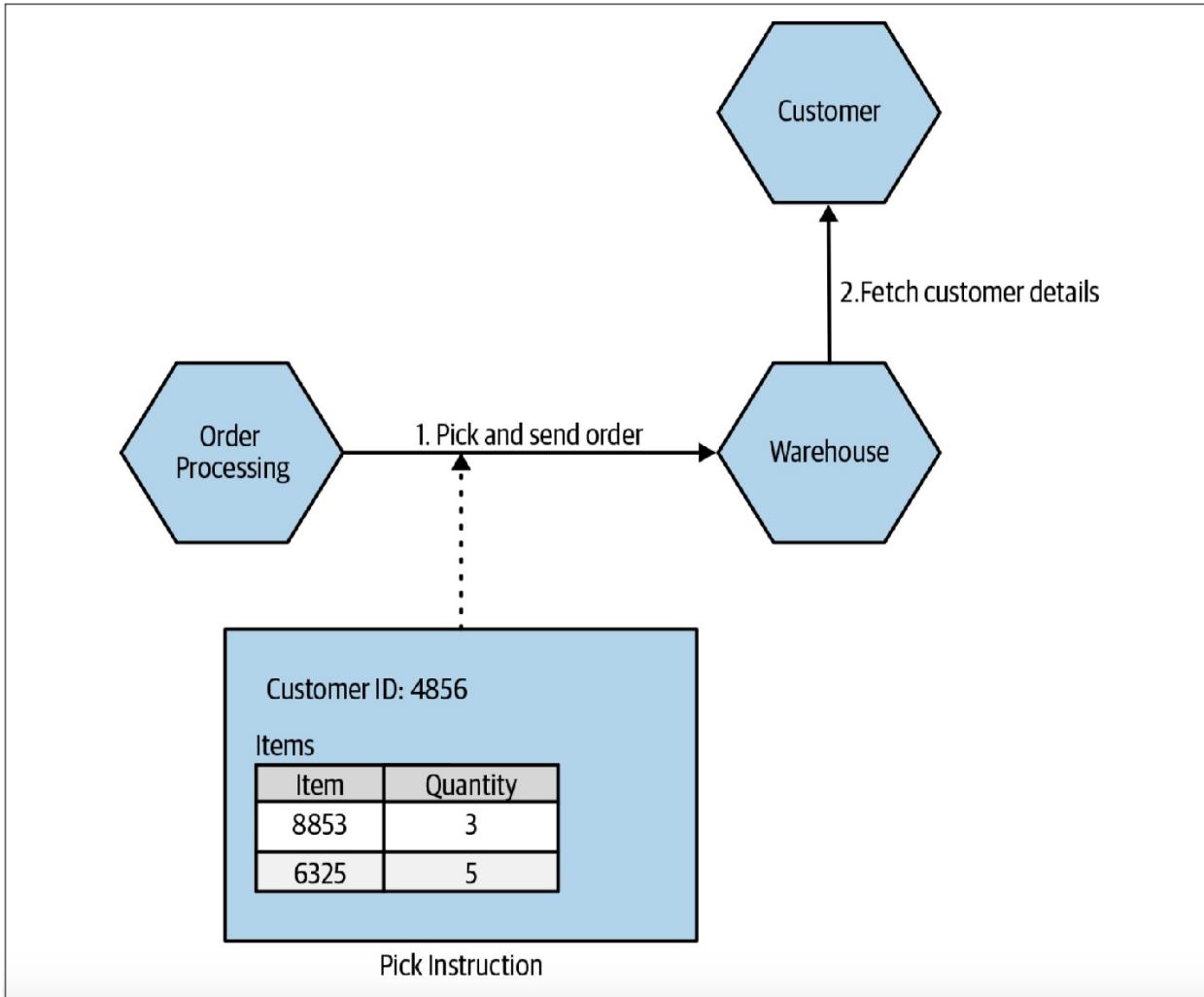


图1-14. 使用Pick Instruction减少发送到Warehouse服务的信息量

我们可以通过消除Warehouse服务所需的信息来进一步减少耦合。甚至，当Warehouse服务需要用户信息时，可以由Pick Instruction提供所有的适当详细信息。如[图1-15](#)所示。

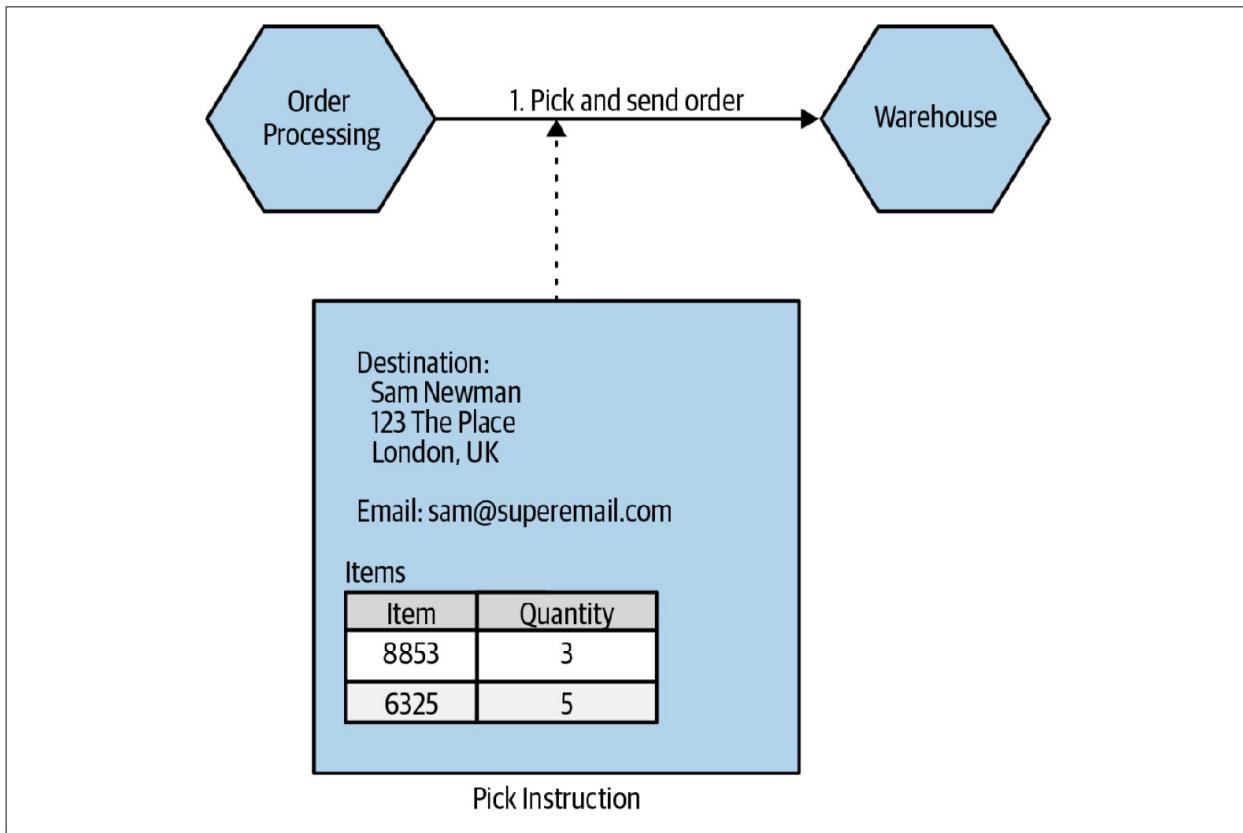


图1-15. 把更多信息放入Pick Instruction可以避免Warehouse服务调用Customer服务

图1-15所示的这种方法可能意味着Order Processing服务必须在某个时刻访问Customer服务才能先生成Pick Instruction。但是，无论如何，Order Processing可能会出于其他原因而访问客户信息，因此图1-15的方法不是什么大问题。“发送”一个Pick Instruction的过程意味着Order Processing服务对Warehouse服务发起了API调用。

除了API调用外，另一种方法是让“Order Processing”发出某种类型的事件，并由Warehouse服务消费该事件，如图1-16所示。通过发送由Warehouse服务消费的事件，我们有效地翻转了依赖关系。之前，Order Processing服务需要依赖Warehouse服务以确保订单的发货。而现在，Warehouse服务只需要监听来在Order Processing服务的事件即可。这两种方法各有千秋。我会

选择哪种方法取决于对Order Processing的逻辑与封装于Warehouse服务中的功能之间的交互的广泛理解。某些领域建模也可以帮助解决如上的方法的选择，我们接下来就会对其探讨。

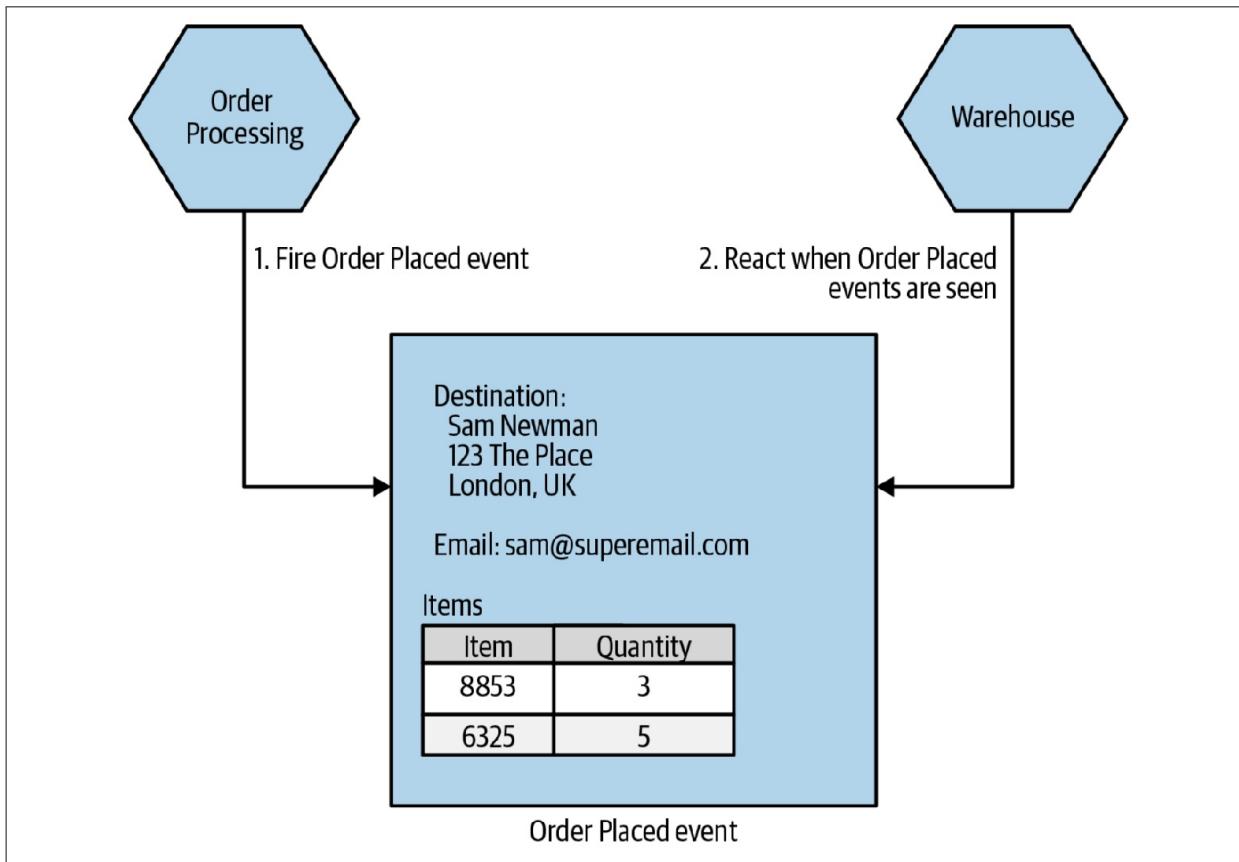


图1-16. 触发Warehouse服务可以接收的事件，该事件仅包含打包和发货的信息

从根本上讲，需要一些有关订单的信息，以使Warehouse服务能够执行某些工作。我们无法避免这种层级的领域耦合。但是，通过仔细思考我们要共享什么信息以及如何共享这些信息，我们仍然可以降低耦合的层级。

---

<sup>6</sup>. Although Parnas's well known 1972 paper “On the Criteria to be Used in Decomposing Systems into Modules” is often cited as the source, he first

shared this concept in “Information Distributions Aspects of Design Methodology”, Proceedings of IFIP Congress ‘71, 1971. ↵

<sup>7</sup>. See Parnas, David, “The Secret History of Information Hiding.” Published in Software Pioneers, eds. M. Broy and E. Denert (Berlin Heidelberg: Springer, 2002). ↵

<sup>8</sup>. See Jez Humble and David Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Upper Saddle River: Addison Wesley, 2010) for more details. ↵

<sup>9</sup>. Greenspun’s 10th rule states, “Any sufficiently complicated C or Fortran program contains an ad hoc, infor- mally specified, bug-ridden, slow implementation of half of Common Lisp.” This has morphed into a newer joke: “Every microservice architecture contains a half-broken reimplementation of Erlang.” I think there is a lot of truth to this. ↵

译注<sup>1</sup>. 原文为two now infamous NATO-sponsored conferences，不确定为什么要用now infamous来形容这次大会。 ↵

译注<sup>2</sup>. W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured design," in IBM Systems Journal, vol. 13, no. 2, pp. 115-139, 1974, doi: 10.1147/sj.132.0115. ↵

译注<sup>3</sup>. Yourdon Edward, Constantine Larry, "Structured design. Fundamentals of a discipline of computer program and systems design." ↵

# 领域驱动设计

正如我们已经讨论的那样，围绕业务领域对服务建模对于微服务架构而言具有明显的优势。问题是如何给出该模型——领域驱动设计（DDD）应运而生。

希望程序更好地代表程序所运行其中的现实世界并不是一个新的想法。可以利用诸如Simula之类的面向对象编程语言对真实业务领域建模。但是，要真正实现这个想法，不仅需要程序语言能力，还需要更多其他的能力。

Eric Evans的领域驱动设计<sup>10</sup>提出了一系列重要思想，这些思想可以帮助我们更好地在程序中表示问题领域。对面向领域驱动设计的全面探索不在本书的讨论范围之内，但是我会简要概述其中的、微服务架构会涉及到的重要思想。

# 聚合

在DDD中，聚合有多种不同的定义，因此聚合是一个有点令人困惑的概念。聚合只是对象的任意集合吗？聚合是从数据库中提取的最小单元吗？对我而言，一直有效的模型是：首先将聚合视为真实领域概念的表示，例如Order, Invoice, Stock Item等。聚合通常具有一个围绕其产生的生命周期，在聚合生命周期中，聚合像状态机一样执行。我们希望将聚合视为独立单元，我们要确保把处理聚合状态转换的代码与状态本身放在一起。

在考虑聚合和微服务时，单个微服务将处理一种或多种不同类型的聚合的生命周期和数据存储。如果另一个服务中的功能想要更改这些聚合中的某一个，则或者直接发送一个对该聚合进行更改的请求，或者让聚合本身对系统中的其他事情做出反应以启动自己的状态转换，如[图1-17](#)所示。

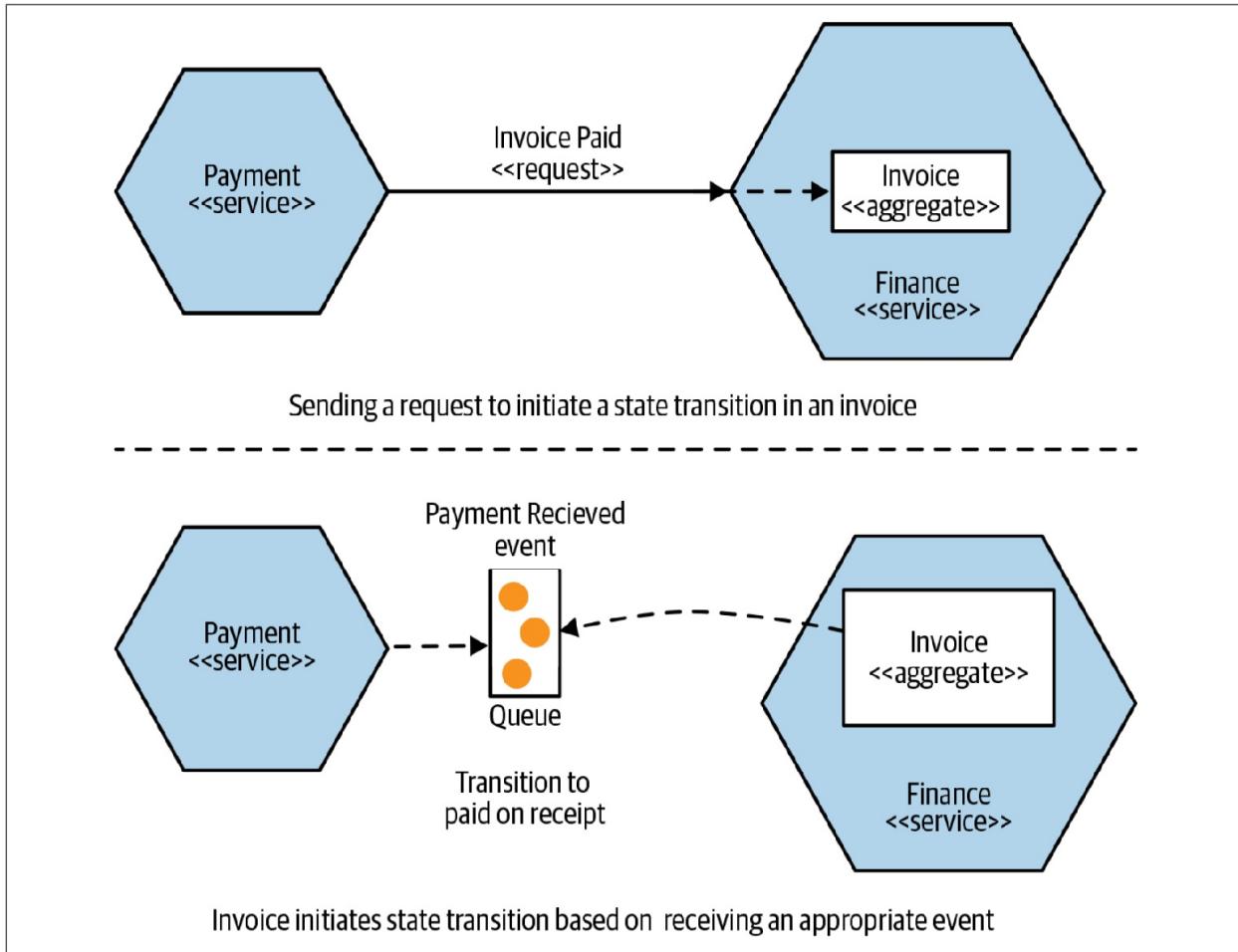


图1-17. Payment服务可能会以不同的方式触发Invoice聚合中的Paid状态转变

这里要了解的关键一点是，如果外部服务请求聚合中的状态转换，聚合可以拒绝该请求。理想情况下，我们希望聚合不能执行非法的状态转换。

聚合可以与其他聚合存在关系。在[图1-18](#)中，我们有一个Customer聚合，它与一个或多个Order聚合相关联。我们决定把Customer和Order建模为单独的聚合，并且可以让不同的服务来处理该聚合。

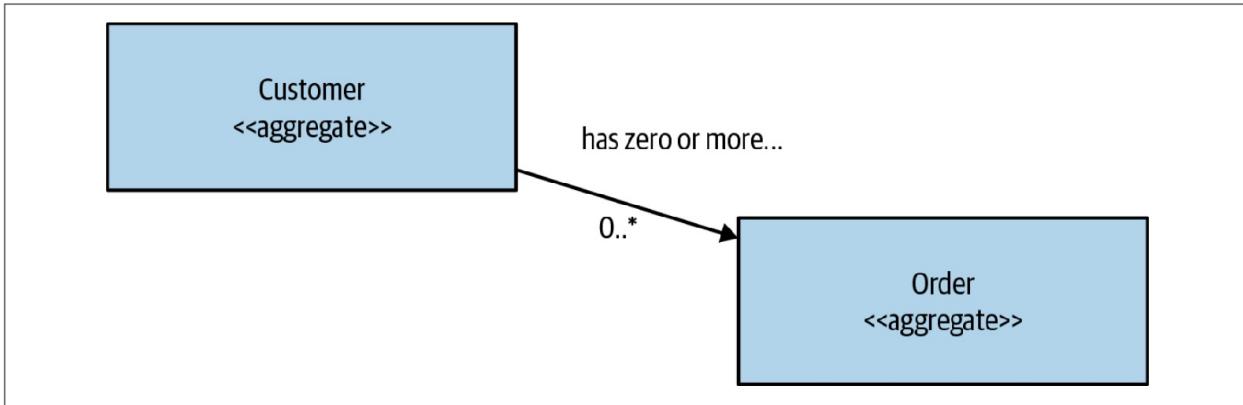


图1-18. 一个Customer聚合与一个或多个Order聚合相关联

把系统分解为聚合的方法有很多，并且有些选择是非常主观的。我们可能出于性能原因或便于实现的原因而决定随着时间的推移来重塑聚合。不过，首先，我认为实现的问题是次要的。首先让系统用户的心理模型来指导初始设计，直到其他的因素发挥作用。在[第2章](#)中，我将介绍一种称为Event Storming的协作方式，以在非开发人员的帮助下塑造领域模型。

# 界定的上下文

界定的上下文通常代表组织内部的更大的组织边界。在界定的上下文范围内，需要执行明确的职责。如上的说法有点模糊，所以让我们来看另一个具体示例。

在Music Corp，我们的仓库看起来非常忙碌：管理发货的订单（和零星的退货），接收新库存，繁忙的叉车等等。然而，财务部门可能不那么忙碌，但在我们组织内部仍然发挥着重要的作用，例如处理薪资，支付货款等。

界定的上下文隐藏了实现细节。有很多只需要内部关心的问题，例如，除了仓库人员以外，几乎没有人会对使用的叉车类型感兴趣。应该对外界隐藏这些内部问题——外界不需要知道、也不必关心这些内部问题。

从实现的角度来看，界定的上下文包含一个或多个聚合。有些聚合可能暴露在界定的上下文之外，而其他的聚合则可能隐藏于界定的上下文内部。与聚合一样，界定的上下文可能与其他界定的上下文存在关系——当映射到服务时，这些依赖关系将成为服务间的依赖关系。

# 将聚合和界定的上下文映射到微服务

聚合和界定的上下文为我们提供了内聚的服务单元，这些服务单元在更广泛的系统中具有明确定义的接口。聚合是一个独立的状态机，它专注于系统中的单个领域的概念。界定的上下文表示相关联的聚合的集合。在界定的上下文的帮助下，聚合可以通过显示接口与更广泛的世界交互。

因此，聚合和界定的上下文都可以很好地作为服务边界。正如我已经提到的，刚开始时，我想大家会希望减少使用的服务数量。因此，我认为应该把目标锁定为包含整个界定的上下文的服务。一旦适应该方法，在决定将服务拆分为较小的服务时，考虑围绕聚合边界来拆分服务。

这里的一个技巧是，即使以后决定将对整个界定的上下文进行建模的服务拆分为较小的服务，仍然可以对外界隐藏此想法，甚至可以通过向下游消费者提供更粗粒度的API来实现。把服务拆分成更小的部分是由实现决定的，因此，如果可以的话，我们最好将其隐藏起来！

# 进一步阅读

全面研究领域驱动设计是一项值得的活动，但这不在本书的范围之内。如果想进一步了解领域驱动设计，建议阅读Eric Evans的*Domain Driven Design*<sup>10</sup> 或Vaughn Vernon的*Domain-Driven Design Distilled*<sup>11</sup>。

---

<sup>10</sup>. Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley, 2004). ↪

<sup>11</sup>. See Vaughn Vernon, Domain-Driven Design Distilled (Boston: Addison-Wesley, 2014). ↪

直接讨论拆解单体应用的技术细节实在是太简单了，本书的剩余部分会重点关注拆解单体应用的技术细节。但是在此之前，我们确实需要先探讨一些非技术的问题：

- 从系统的哪个部分开始迁移？
- 如何管理变更？
- 如何在迁移过程中带动其他人？
- 还有一个需要在早期就思考的重要问题——是否应该首先使用微服务？

# 了解迁移的目标

微服务不是我们的目标。我们也不会因为选择了微服务架构而“获胜”。选择微服务架构应该是一个清醒的决定，该决定是基于理性决策的结果。我们选择迁移到微服务架构的目的是实现当前的系统架构无法实现的功能。

如果不了解要实现的目标，我们又如何为我们要选择哪种方案的决策过程提供信息？我们对微服务架构的期望会影响我们关注的焦点以及工作的优先级。

了解我们的目标会帮助我们避免成为分析瘫痪（在面临大量选择时，做出过度分析，最终导致无法作出任何决择）的受害者。不了解目标还会让我们面临陷入货物崇拜<sup>译注1</sup>的风险，此时我们就会假设：“如果微服务对Netflix有好处，那么微服务对我们也有好处！”

## 微服务的普遍问题

多年以前，我在一次会议上组织了一个微服务研讨会。像往常一样，我希望了解与会者为什么会参与该研讨会，以及他们希望通过这次研讨会获得什么。在这次研讨会中，有几个与会者来自同一家公司。我也非常好奇：为什么他们的公司会派遣他们参与这次研讨会。

我问了他们其中的一员：“你为什么要参与这次研讨会？你为什么会对使用微服务感兴趣？”

然而他们的答案却让我吃惊：“我们也不知道原因呀，我们的老板通知我们来参与这次研讨会！”

他们的回答引起了我的兴趣，我也想对此一探究竟。

我继续追问：“那么，你知道老板为什么让你参与这次研讨会吗？”

与会者回答：“好吧，你可以直接问一下我们的老板，他就坐在我们的后面。”

于是，我问了他们的老板同样的问题：“你们为什么要使用微服务？”

他们老板的回答同样令我无言以对：“我们的CTO说我们正在做微服务，所以我认为我们应该弄清楚微服务到底是什么！”

这个真实的故事虽然很有趣，但却是关于微服务的一个普遍的问题。我遇到过很多团队，他们在决定采用微服务架构之前并没有真正了解他们为什么要采用微服务架构或者他们希望采用微服务架构来实现什么目的。

如果对为什么要使用微服务架构没有一个清晰的认识，则会带来各种各样的问题。微服务可能需要大量的资源投入，这些投入或者是直接的人力或者资金的增加，或者是在增加服务新特性的同时需要安排好微服务改造的优先级。事实上，因为可能需要经过一段时间的积累才能看到微服务改造带来的好处，所以这会使情况变的更加复杂。有时候，这会导致团队陷入如下的场景：在经历了一年或更长时间的微服务改造之后，团队成员却已经忘记了当初为什么要对系统进行微服务改造。这不仅仅是一个简单的沉没成本谬误 (**sunk cost fallacy**) [译注2](#) 的问题，实际上整个团队根本就不知道为什么要迁移到微服务架构。

### 沉默成本谬误的例子

你花钱买了一张电影票，看了半个小时以后发现这是一部浪费时间的烂片。你是拔腿就走，还是坚持看完？

你在站台等了二十分钟，公交车都迟迟不来。你是打个出租车走呢，还是继续在站台等？

依据亚当·斯密的“理性人”假设，人会通过谨慎思考，做出最理性的决策，以谋求利益的最大化。因此，在发现成本已经沉没时，理性的做法，当然是只考虑未来的成本和收益。但是人类并没有超级计算机那样理性冷静的头脑，在做出决策时，往往过于重视已付出的成本，从而掉入沉没成本谬误的陷阱。

陶渊明说：悟已往之不谏，知来者之可追。就是这个意思。

同时，我还会收到分享关于迁移微服务架构的ROI的需求。有些团队希望通过确凿的事实和数据来支撑他们为什么应该考虑采用微服务架构的决策。然而实际情况又是另一番光景：由于没有对如上提到的微服务的相关问题进行详细的分析研究（即使团队也确实做过一些调研），调研中分析到的优势可能会因为团队之间所处的背景不同而很难移植。

那么，我们又将何去何从？靠猜来工作？答案当然是否定的。我坚信我们能够并且应该可以对我们的开发、技术以及架构选择的有效性进行更好的研究。为此，很多工作已经完成并且以类似 [The State of DevOps Report](#) 的形式发布出来，然而类似的工作对于架构的分析却谈之甚少。在决策中，我们至少应该努力采用更具批判性的思维，并采用实验的心态来替换那些迁移前的严格的调研工作。

对于希望实现的目标，我们需要有一个清晰的了解。如果无法正确的评估我们所寻求的回报，那么也就无法计算ROI。我们要专注于希望实现的结果，而不是一味地坚持单一的方法。我们需要明智地思考到达目标的最佳方法，即使这意味着要放弃大量工作或者重新回到老式的枯燥乏味的但却是有效的方法，我们也要执行。

# 三个关键问题

在与公司合作以帮助他们了解是否应该考虑采用微服务架构时，我倾向于提出相同的问题：

**希望实现什么？**

这应该是与业务目标一致的一组结果，并且可以用系统所能提供的用户价值来描述清楚。

**是否考虑过微服务的替代方案？**

正如我们稍后将探讨的那样，通常还会有很多其他的方案也可以实现微服务所带来的类似的好处。

**是否了解过这些方案？**

如果没有，为什么不去了解一下？

有时，使用更简单的技术就可以达到目标。

**如何知道迁移过程是否有效？**

如果决定着手微服务的迁移，如何知道迁移过程是否朝着正确的方向开展？

我们将在本章的结尾处再讨论该主题。

我已经不止一次地发现，提出如上的这些问题足以让公司重新考虑是否要继续采用微服务架构。

---

译注<sup>1</sup>. 货物崇拜：是一种宗教形式，尤其出现于一些与世隔绝的落后土著之中。当货物崇拜者看见外来的先进科技物品，便会将之当作神祇般崇拜。 ↪

译注<sup>2</sup>. 沉默成本：沉没成本由诺贝尔经济学奖的获得者，芝加哥大学教授理查德·泰勒提出，是已发生或承诺，无法收回的成本支出。从决策的角度看，以往发生的费用只是造成当前状态的某个因素，当前决策所要考慮的是未来可能发生的费用及所带来的收益，而不考慮以往发生的费用。人们在决定是否去做一件事情的时候，不仅是看这件事对自己有没有好处，而且也看过去是不是已经在这件事情上有过投入。我们把这些已经发生不可收回的支出，如时间、金钱、精力等称为“沉没成本”。 ↪

Copyrights © wangwei all right reserved

# 为什么选择微服务

我无法确定你们公司的目标。你们自己比我更了解你们公司的目标和你们目前正在面临的挑战。我能给出的是：为什么全世界的公司都采用微服务架构的原因。本着诚实的精神，我还会给出可以实现相同的结果的、微服务架构之外的其他方法。

# 提升团队自治

无论从事什么行业，都要认识到员工的重要性，并吸引他们正确地做事，为他们提供自信、动力、自由和渴望，以挖掘出他们真正的潜力。

——John Timpson（英国Timpson公司老板）

很多公司都已经获得创建自治团队的好处。保持小规模的团队，可以避免过多的官僚主义，从而使得团队成员之间建立密切的联系，并能一起高效工作。小团队也帮助很多公司在同行之中获得更有效的发展和扩张。在戈尔公司（*W. L. Gore & Associates*），所有的业务部门都不会超过150名员工，从而确保部门内的每个人都彼此认识。利用这种小团队的组织方式，戈尔公司取得了巨大的成功。为了使得这些较小规模的业务部门可以正常工作，必须赋予小团队权力和责任，以使其能够独立运作。

Timpsons——一家非常成功的英国零售商，已经通过增强全体员工能力、减少对公司总部的需求、允许本地商店自行做出决定、赋予员工可以给予不满意的顾客多少折扣的权利等方式实现了公司的大规模经营。公司的现任董事长John Timpson也以取消内部规则并将其替换为仅有的两条如下规则而闻名：

- Look the part and put the money in the till.
- You can do anything else to best serve customers.

自治也需要在较小的团队规模下才能成为可能。我合作过的大多数现代化公司都希望在其内部创建更多的自治团队，他们经常尝试复制其他公司的团队自治模式，例如：亚马逊的两个披萨（*two-pizza*）模式或Spotify模

式<sup>1</sup>。

如果一切安好，团队自治就能够给予员工授权，并帮助员工成长以更快地完成工作。当团队拥有微服务并完全控制这些服务时，这些团队就提升了他们可以在公司中的自治程度。

## 还可以怎么做

可以通过多种方式实现团队自治（职责分配）。将更多职责下放到团队的方法并不需要改变架构。从本质上讲，这是一个确定哪些职责可以下放到团队的过程，并且可以通过多种方式来实现职责的分配。一种方案就是将代码库的部分权限授予不同的团队（该方案同样可以让模块化的单体应用从中受益）：也可以将代码库按照功能划分为不同部分，并为不同部分确定有决策权限的人员以实现职责的分配（例如，Ryan最了解广告展示，因此由Ryan来负责广告展示的部分；Jane最了解查询的性能调优，因此涉及查询性能的任何工作都必须首先经过Jane）。

提高自治度会让事情变的简单，我们无需等待其他人的响应就可以完成工作。自助式服务避免了那些日复一日的、需要运维团队到场才能完成的工作。因此，采用自助服务的方式来配置机器或环境的方案是一种巨大的推动力。

# 缩短产品上市的时间

具备如下的能力时，我们就可以更快地向我们的客户发布新的功能：

- 对单个微服务进行变更并对变更进行部署
- 在部署单个微服务的变更时无需依赖其他服务的发布

投入更多的人来解决问题也是一种方案——这种方案我们稍后就会讨论。

## 还可以怎么做

在考虑如何更快地发布软件时，会有很多变量会影响软件的发布速度。那么，我们从哪个因素开始执行呢？我总是建议执行某种生产路径模型 (*path-to-production modeling*) 的演练，这种演练可能有助于发现那些意想不到的最严重的发布瓶颈。

我想起了多年以前，一个大型投资银行的一个项目。我们受邀来帮助他们提升软件的交付速度。我们收到如下的信息：“开发人员花费太长时间才能将产品部署到生产环境！”Kief Morris——我的一位同事——花了一些时间梳理出软件交付所涉及到的所有阶段，然后研究从产品负责人 (PO: *product owner*) [译注1](#) 提出某个想法到该想法真正在产品中实现的整个过程。

Kief Morris很快就发现：从开发人员开始工作到将其部署到生产环境，平均需要大概6周的时间。由于整个过程涉及到手工操作，因此我们认为可以利用适当的自动化手段将整个过程减少几周。但是，Kief在生产路径模型

中发现了一个更大的问题——从产品负责人<sup>译注1</sup>提出想法到开发人员开始实现该想法，通常会耗时40周。通过专注于后者的流程改进，我们帮助客户更大地缩短了新功能的上市时间。

因此，在解决如何快速发软件时，需要考虑软件发布所涉及到的所有阶段。关注整个过程需要多长时间，每个阶段的持续时间（包括该阶段的总时间以及真正用于该阶段的时间<sup>译注2</sup>），并标注出整个过程的痛点。毕竟，微服务可能是解决软件快速发布的部分方案，可以并行尝试很多其他的方案。

# 有效的扩容

把我们的服务拆分为相互独立的微服务，就可以单独扩容拆分后的微服务。这意味着我们也能以此进行有效的扩容——只需要扩容那些当前会限制系统负载能力的微服务即可。同样的，对于那些负载较小的微服务，我们可以对其缩容，甚至在不需要时将其关闭。这就是为什么这么多构建 SaaS 产品的公司会采用微服务架构的原因——微服务让这些公司能够更好地控制运营成本。

## 还可以怎么做

对于扩容，有大量的替代方案可供我们选择，在决定使用微服务架构之前，可以更容易的对其中的大多数方案进行试验。可以从获取更强性能的机器 (*bigger box*) 开始，如果使用公有云或其他类型的虚拟化平台，则可以通过简单地配置来让程序运行在具备更高性能的机器上。显然，这种“垂直扩展”[译注3](#) 有其局限性，但是对于负载的短期快速优化而言，该方案也可以处于考虑范围之内。

对于当前的单体应用而言，传统的水平扩展[译注4](#)——运行多个单体应用的副本——的有效性已经得到证实。通过某种负载分配机制（例如负载均衡或者队列），运行单体应用的多个副本可以轻松处理更多负载。然而，如果系统的负载瓶颈是数据库，则水平扩展则可能无济于事，当然，这也取决于采用的数据库是否支持水平扩展。可以快速评估出水平扩展的有效性。同时，水平扩展的缺点比成熟的微服务架构少得多。再者，水平扩展也比较简单。因此，在采用微服务之前，真的应该放手一试。

还可以采用能够更好地解决负载技术的其他替代方案。但是，这通常不是一件容易的事——考虑一下把当前的程序迁移到新型的数据库或编程语言的工作。对于微服务架构，当需要改变技术时，只需在那些使用该技术的微服务内部改变技术即可，而其他的微服务则保持不变，从而可以减少技术更改的影响。因此，实际上，微服务架构可以让改变技术变得更加容易。

# 提高系统鲁棒性

从单租户 (*single-tenant*) [译注5](#) 软件到多租户 (*multi-tenant*) SaaS [译注6](#) 应用程序的转变意味着系统宕机机会更普遍。客户对软件可用性的期望以及软件在客户生活中的重要性都在增加。通过把应用程序分解为单独的、可独立部署的程序，我们提供了许多机制来提高应用程序的健壮性。

利用微服务架构，系统的功能得到了拆解，因此，我们可以实现更健壮的架构。也就是说，系统中部分功能的异常不会导致整个系统宕机。我们还能将时间和精力集中在系统中对鲁棒性要求最高的服务，以确保系统的关键部分保持万无一失。

## 弹性与鲁邦性 (Resilience Versus Robustness)

我们经常在如下的场景时讨论弹性：

- 提高系统的能力从而避免系统宕机
- 在发生故障时优雅地处理故障
- 在问题发生时迅速恢复服务

如今，在称为[弹性工程学](#)的领域中已经做了很多工作。弹性工程不仅涉及计算机领域，而是将其应用到的所有领域作为一个整体来研究。David Woods——弹性模型的开拓者——从更广泛的角度来看待弹性的概念，并指出弹性并非像我们之初想象的那样简单的事实，其中，弹性要求我们对已知故障源和未知故障源的处理能力进行划分。<sup>2</sup>

John Allspaw——David Woods的同事——帮助我们来区分鲁棒性和弹性。鲁棒性是系统对预期变化做出反应的能力。弹性是指组织应对那些未曾预料到的事情的能力，这也包括：通过类似混沌工程的方法而

建立的一种实验文化。例如，我们知道有一台特定的机器可能会死机，因此我们通过对实例进行负载均衡为系统带来冗余。这就是提升鲁棒性的一个例子。组织不可能预见到所有的潜在问题，弹性即是组织为这些无法预见到的问题而做好准备的过程。

特别需要注意的是：微服务不一定会免费提升系统的鲁棒性。相反，微服务为设计如下的系统提供机会：具备更好地网络分区容错能力，服务宕机容错能力……

仅仅把功能分散到多个独立的程序和独立的机器上不仅不能保证提升鲁棒性——恰恰相反——可能会扩大系统故障的范围。

## 还可以怎么做

通过某种负载均衡器或其他的类似队列<sup>3</sup>的负载分配机制运行单体应用的多个副本，从而增加系统冗余。我们可以通过在多个故障平面上部署单体应用的实例来进一步提高应用的鲁棒性（例如，不要把所有机器都放在同一机架或同一数据中心）。

对更可靠的硬件和软件的投资同样可以产生收益，因此，可以对系统宕机的现有原因进行彻底的排查。我见过很多的线上事故，这些事故因为过度依赖手工操作或人们“不遵守规范”而引起，这通常意味着个人的无辜错误可能会产生重大影响。2017年，5月27日，英国航空公司发生重大的“IT系统故障”，导致进出伦敦希思罗机场和盖特威克机场的所有航班都被取消。此次故障是由单个数据中心的电源意外关闭而无意触发。如果应用程序的鲁棒性依赖于人类永远不会犯错，那么我们将一路艰难。

# 扩大开发人员数量

我们可能都已经意识到了这个问题：投入大量开发人员到项目中以试图加速项目进展的做法常常适得其反。但是有些问题确实需要更多的人才能完成。正如Frederick Brooks在其极具影响力的著作《人月神话》（*The Mythical Man Month*）<sup>4</sup> 中所概述的那样：只有能够把工作划分为独立的不同部分，并且划分之后的工作之间的交互非常有限，增加更多的人才会继续提高交付速度。Frederick Brooks以田间收割庄稼为例：对于收割庄稼而言，让多人并行工作是一件简单的事情，此时每个人的工作都不需要与其他人进行互动。软件开发却很少像收割庄稼般工作。在软件开发时，每个人完成的工作并不完全相同，一个人的工作输出经常会是其他人的输入。

有了清晰定义的边界，以及主要关注在确保限制微服务之间的耦合的架构，我们可以抽取出可独立开发的代码段。因此，我们希望通过减少交付冲突扩大开发人员的数量。

为了成功扩大解决问题的开发人员的数量，团队之间必须具备高度的自治。仅仅拥有微服务是不够的，还必须思考：团队和服务权限之间如何保持一致，团队之间需要协调什么。我们还需要把工作进行拆解，从而使得变更不需要经过太多微服务之间的协调。

## 还可以怎么做

由于微服务本身是没有耦合的、可以独立处理的服务，因此微服务对于较大的团队而言有很好的效果。模块化的单体应用是另一种替代方案：不同的团队拥有各自的模块，只要模块之间交互的接口保持稳定，团队之间就

可以继续独立执行变更。

但是，模块化的单体应用的方案有些局限。不同的团队之间仍然存在某种形式的冲突。对于该方案而言，软件仍然需要把全部模块打包在一起，因此部署工作仍然需要相应各方之间的协调。

# 拥抱新的技术

单体应用通常会限制我们的技术选择。通常，我们在后端使用一种编程语言。我们固化到一种部署平台，一种操作系统和一种数据库。通过微服务架构，我们有机会为每个服务选择不同的技术。

通过将技术变更隔离在一个服务边界内，我们可以在隔离的服务中了解新技术的优势，并在新技术出现问题时限制其影响。

以我的经验，虽然成熟的微服务组织通常会限制他们支持的技术栈数量，但微服务之间所使用的技术很少是同质的。能够以安全的方式尝试新技术的灵活性可以为组织提供竞争优势：既可以为客户提供更好的结果，也可以帮助开发人员因为掌握新技能而保持快乐。

## 还可以怎么做

如果我们仍然继续将软件作为一个单独的程序来交付，那么我们可以引入的技术就会受到限制。当然，我们可以在同一运行时（*runtime*）安全地采用新语言——作为一个例子，JVM可以在相同的进程中托管多种语言编写的代码。新的数据库类型会出现更多问题，因为这意味着需要对先前的整个数据模型进行某种拆解以允许其增量迁移。除非我们打算立即完全地升级到新的数据库技术，否则更新数据库技术是一种复杂且有风险的行为。

如果当前的技术堆栈被认为是“burning platform”，那么别无选择，只能用更新更好的技术栈来替换当前的技术栈<sup>5</sup>。当然，没有什么可以阻止我们用新的应用逐步替换现有应用。第3章介绍的绞杀者模式可以很好地实现这一目标。

## 复用?

复用是微服务迁移中最常提及的目标之一，而在我看来，复用却不是首先应该考虑的目标。就其本质而言，复用并不是人们想要的直接结果。人们希望复用可以带来其他好处。我们希望通过复用更快地发布新功能，或者降低成本。但是，如果我们的目标是这些事情，那请跟踪这些目标，否则最终会做了错误的优化。

为了解释我的意思，让我们更深入地研究选择复用作为目标的常见原因之一。我们希望通过复用更快地发布功能。我们认为，通过复用现有的代码来优化我们的开发流程，此时不必编写过多的代码——只需要更少的工作，就可以更快地发布软件。

是这样吗？

让我们来看一个简单的例子。Music Corp的客服团队需要格式化PDF文件才能给顾客提供发票。该系统的其他部分已经在处理PDF文件的生成：我们在仓库中生成用于打印的PDF，为运送给客户的订单生成装箱单，并给供应商发送订单请求。

按照复用的目标，团队可能会直接使用现有的PDF生成功能。但是目前，该功能由组织中的不同部门的不同团队管理。因此，我们现在必须与他们协作，并让他们进行必要的更改以支持我们的需求。这意味着我们不得不要求他们为我们而工作，或者我们不得不自己修改代码并提交pull request（假设公司是这样的工作模式）。无论哪种方式，我们都必须与另外的团队协作才能进行更改。

我们会花时间与他人协作并完成更改，然后就可以实现功能更改。但是，我们发现，与花费时间修改现有代码相比，实际上可以更快地编写自己的实现以更快的将功能交付给客户。如果实际目标是缩短产品

上市时间，那么这可能是正确的选择。但是，如果针对复用进行了优化，并希望可以更快地将产品推向市场，那么我们最终做的是导致我们放慢脚步的事情。

对于复杂系统而言，度量复用性非常困难。正如我所描述的那样，通常我们会为实现其他目标而做某些事情。我们需要花时间专注于实际目标，并认识到复用可能并不总是正确的方案。

---

<sup>1</sup>. Which famously even Spotify doesn't use anymore. ↪

<sup>2</sup>. See David Woods, “Four Concepts for Resilience and the Implications for the Future of Resilience Engineering.” *Reliability Engineering & System Safety* 141 (2015) 5–9. ↪

<sup>3</sup>. See the competing consumer pattern for one such example, in *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf, page 502. ↪

<sup>4</sup>. See Frederick P. Brooks, *The Mythical Man-Month*, 20th Anniversary Edition (Boston: Addison-Wesley, 1995). ↪

<sup>5</sup>. The term “burning platform” is typically used to denote a technology that is considered end-of-life. It may be too hard or expensive to get support for the technology, too difficult to hire people with the relevant experience. A common example of a technology most organizations consider a burning platform is a COBOL mainframe application. ↪

译注<sup>1</sup>. product owner，在Scrum敏捷开发中有三种主要的角色：Product Owner（产品负责人，简称"PO"），Scrum Master（敏捷教练），Team（团队）。产品负责人是有授权的产品领导力核心，担任的是产品经理的角色。 ↪

译注2. elapsed time and busy time. ↵

译注3. 垂直扩展：通过优化系统中的现有的模块的处理能力来提高系统负载。 ↵

译注4. 水平扩展：不是通过优化系统中模块的负载，而是简单的通过增加更多的模块来提高系统负载。 ↵

译注5. 单租户：指的是为每个客户单独创建各自的软件应用和支撑环境。每个客户都有一份分别放在独立的服务器上的数据库和操作系统，或者使用强的安全措施进行隔离的虚拟网络环境中。 ↵

译注6. 多租户：简称SaaS，是一种软件架构技术，是实现如何在多用户环境下（此处的多用户一般是面向企业用户）共用相同的系统或程序组件，并且可确保各用户间数据的隔离性。 ↵

Copyrights © wangwei all right reserved

# 什么时候微服务不是一个好的选择

我们花了很长时间来探索微服务架构的潜在优势。但是在某些场景下，我一点都不建议使用微服务。现在让我们来看一下不建议使用微服务的几种场景。

# 不清楚业务领域时

错误地定义服务边界的代价很高。错误的服务边界可能导致大量的跨服务更改，过于耦合的组件，并且通常而言情况会比单体系统还要糟糕。

在“[Building Microservices](#)”一书中，我分享了ThoughtWorks公司的SnapCI<sup>译注1</sup>团队的经验。尽管他们对CI（持续集成）领域非常了解，但他们最初为托管CI解决方案而尝试的服务边界并不正确。这带来了很大的变更成本和管理成本。经过几个月的鏖战，该团队决定将服务合并回一个大应用程序。后来，当应用程序的功能集有所稳定并且团队对CI领域有了更深入的了解时，找到那些稳定的边界就变得容易了。

SnapCI是一个CI/CD的托管工具。该团队以前曾开发过另一个类似的工具Go-CD。Go-CD是一个开源的，可以在本地部署而不是云端托管的CD（持续交付）工具。尽管在SnapCI项目的早期，SnapCI和Go-CD项目之间有一些代码复用，但最终SnapCI变成了一个全新的代码库。尽管如此，该团队在CD工具领域的经验使他们有信心地、更快地确定服务边界，并将系统构建为一组微服务。

然而，几个月后，事情变的明朗起来：SnapCI的使用场景（*use cases*）和之前想的有所不同，以至于最初对服务边界的划分并不完全正确。这导致了大量的跨服务更改，并带来较高的更改成本。最终，该团队将服务合并回一个单体系统，使他们可以有时间更好地了解边界应该在何处。一年后，该团队便有能力把单体系统拆分为微服务。事实证明，这次拆分的微服务的边界更加稳定。SnapCI的例子还远不是我所看到的过早拆分的唯一例子。过早地将系统拆分为微服务可能会付出高昂的代价，尤其是当我们还是该领域的新手时。在许多方面，拥有要拆分为微服务的现有代码库比尝试从一开始就使用微服务要容易得多。

如果我们认为，我们对自己的领域尚未做到了如指掌，那么在进行系统拆分之前先深入了解业务可能是个好主意。（这是进行领域建模的另一个原因，我们稍后将会讨论。）

# 初创企业

很多因使用微服务而闻名的公司都被认为是初创企业<sup>译注2</sup>，因此初创企业不适合使用微服务这个观点看起来是有争议的。但实际上，这些初创企业——包括Netflix，Airbnb等——都只是在其发展的后期才转向微服务架构。微服务对于“规模企业”<sup>译注3</sup>而言可能是一个不错的选择，因为初创企业至少已经建立了产品/市场适应性的基础，现在正在扩大规模以提高（或可能只是实现）盈利能力。

有别于规模企业，初创企业经常尝试各种想法，以期找到用户需求。随着市场空间的探索，产品的原始愿景可能会发生巨大变化，从而导致其产品领域发生巨大变化。

一家真正的初创企业可能是一家资金有限的小企业，需要集中所有精力为其产品找到正确的需求。微服务主要解决初创企业在发现满足用户需求方案之后的问题。换句话说，微服务是解决初创企业成功后会遇到的各种问题的好方法。因此，首先要专注于成功！如果最初的想法不好，那么是否使用微服务构建也就无所谓了。

拆解现有的**brownfield**<sup>译注4</sup>系统要比初创企业创建新的**greenfield**<sup>译注5</sup>系统要容易得多。对于“brownfield”系统而言<sup>译注4</sup>，有更多的资源可以提供给我们。我们有可以检查的代码，我们可以与使用和维护该系统的人员交谈。对运行中的系统进行更改，我们也能知道怎样的更改看起来是好的，也使我们能更容易的知晓我们是否在决策过程中犯错或者过于激进。

当然，我并不是说初创企业永远不要选择微服务，而是说这些因素意味着我们应该谨慎。仅围绕一开始就很清晰的边界进行拆分，其余部分则保留在单体之中。这也将使我们有时间从运营的角度来评估我们的成熟度——

如果我们管理两个服务都很费劲，那么管理十个服务就将很困难。

# 需要客户安装并管理的软件

如果我们的软件需要打包并交付给客户，然后由客户自己运维，那么微服务可能是一种糟糕的选择。当迁移到微服务架构时，会给运维领域带来很多复杂度。那些用于监控单体应用并排查单体应用故障的先前的技术很可能不适用于新的分布式系统。现在，迁移到微服务的团队通过采用新技能——或采用新技术——来克服这些挑战。然而，这些通常不是我们的最终客户所期望的。

通常，对于需要客户安装的软件，我们会指定特定的平台。例如，我们可能会说“需要Windows 2016 Server”或“需要macOS 10.12或更高版本”。这些是明确定义的目标部署环境，我们很有可能会使用这些系统的管理人员所熟知的机制来打包单体软件（例如，发布Windows服务，并捆绑在Windows Installer程序包中）。我们的客户可能熟悉以这种方式购买并运行软件。

想像一下，当我们给客户一个程序来运行和管理时所遇到的麻烦，我们还要再给客户10个或20个程序？甚至更激进地，难道我们还期望客户在Kubernetes集群或类似集群上运行我们的软件？

现实情况是，我们不能期望客户拥有管理微服务架构的技能或平台。即使客户这样做了，他们也可能没有我们所需的相同的技能或平台。例如，不同的Kubernetes安装方式之间就有很大的差异。

早在2019年底的KubeConNA中，Google API基础设施的架构师Louis Ryan就透露了Istio控制平面架构将要进行调整的消息。[从即将发布的1.5版本开始](#)，原本多个独立的组件将会整合在一起，成为一个单体结

构。复杂是万恶之源，停止焦虑，学会爱上单体。

# 暂无充分理由时

最后，如果对要实现的目标没有清晰的认识，我们就有最充分的理由不采用微服务。正如我们将要探讨的那样，采用微服务的目标决定我们从何处开始迁移，同时也决定我们如何拆解当前的系统。如果对目标没有清晰的愿景，我们就将在黑暗中摸索。迁移微服务仅仅是因为其他人都在做微服务，这是一个糟糕透顶的主意。

## 为什么像王者荣耀这样的游戏Server不愿意使用微服务？

无独有偶，在浏览知乎的时候，发现了这篇文章，具体可以[点击跳转到文章正文](#)。

所以游戏的核心在于小规模群体之间的高速网络通信。就是对方说的realtime。多了一个10ms的延迟玩家就要骂娘了。

因此，从实时的方面讲，如果我们提供的服务需要强实时性，那么微服务可能确实不是一个好的选择。多增加一层网络交互，都会不可避免的带来延迟的增加。

正如文中所说：微服务不是什么银弹。

---

译注<sup>1</sup>. Snap目前已经下线了。 ↵

译注<sup>2</sup>. 初创企业（startup）：创新精神的证明，用于证明产品是否符合市场需求。 ↵

译注3. 规模企业（scale-up）：介于初创企业（Start-up）和公司（Corporate）之间，是“下一个层次的初创企业”。初创企业是创新精神的证明，而规模企业意味着影响力和激烈的竞争。经合组织（OECD）给规模企业的定义是：三年内员工(或营业额)年平均增长率超过20%，并且在观察期开始时员工人数超过10人的企业。 ↵

译注4. Brownfield：指的是在遗留系统之上开发和部署新的软件系统，或者需要与已经在使用的其他软件共存。 ↵

译注5. Greenfield：指的是在全新环境中从头开发的软件项目。 ↵

Copyrights © wangwei all right reserved

# 权衡折衷

到目前为止，我已经概述了人们可能想单独使用微服务的原因，并简要给出了考虑其他选择的理由。但是，在现实世界中，人们通常会尝试一次改变许多事情，而不是尝试一次改变一件事情。一次改变太多事情会导致事情的优先级比较复杂，从而导致所需的更改量快速增加，并且无法在短期内看到任何收益。

一切的开始都比较简单。我们需要重新设计应用程序，以应对流量的大幅增长。我们也确定微服务是未来的方向。

有人冒出来问：“那么，如果我们采用微服务的话，我们可以同时使我们的团队更加自治！”

另一个人附和道：“这给了我们一个很好的机会来尝试Kotlin！”

在不知不觉中，就有了一个庞大的变革计划，该计划正在：

- 试图发挥团队自治能力
- 扩容应用程序并同时引入新技术
- 为加强工作计划而采取其他措施

微服务就在这种情况下被锁定为解决流量上涨的方案。如果只关注扩容，在迁移过程中可能就会发现，还不如对现有的单体应用进行水平扩展。然而，水平扩展的方案却不能帮助我们达到次要目标：提高团队自治或引入Kotlin作为编程语言。

因此，把变革背后的核心驱动力与希望获得的任何的间接收益区分开来非常重要。在上述例子中，提高应用程序的容量是最重要的事情。为了其他的次要目标的进展而进行的工作（例如提高团队自治）可能会很有用，但是如果它们阻碍或损害了关键目标，就应该把这些次要目标往后放。

重要的是要认识到某些事情比其他事情更重要。否则，就无法正确确定优先级。此时，我喜欢把每个期望的结果视为一个滑动条。每个滑动条都以中间位置为起点。当一件事情变得更加重要时，必须放弃另一件事情的优先级——如图2-1所示。图2-1清晰地表明，虽然我们想让多语言编程变得更容易，但却没有确保应用程序具有更高的弹性重要。当要弄清楚如何前进时，清晰地阐明这些结果并对其进行排序可以使决策更加容易。

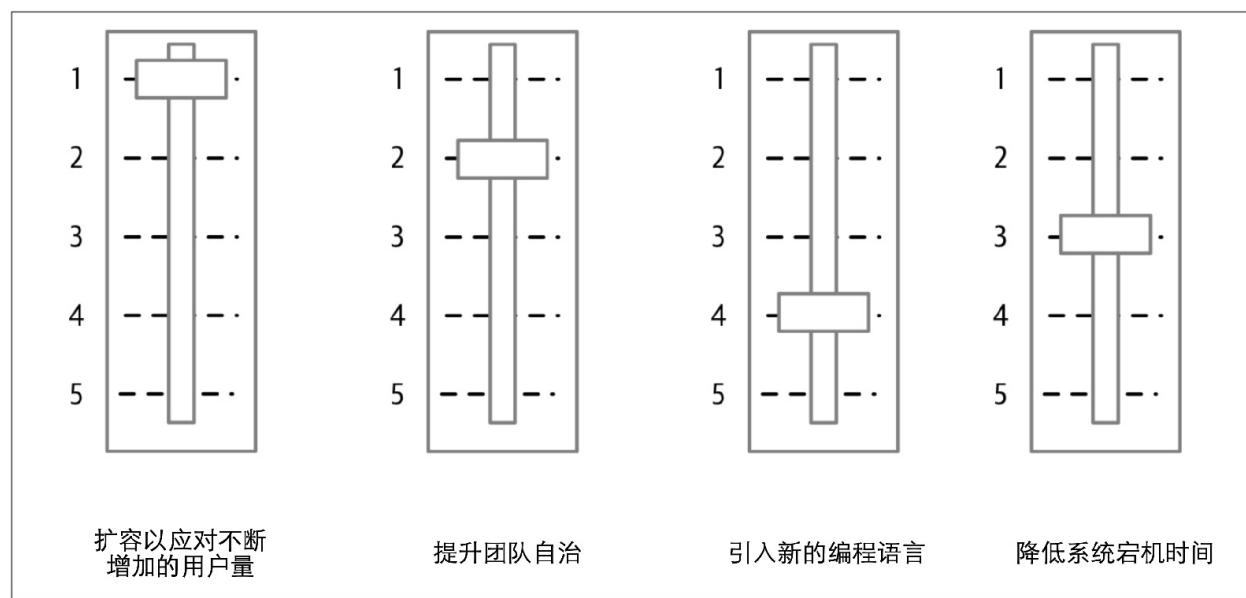


图2-1. 利用滑动条来平衡优先级

可以修改不同目标的相对优先级（并且随着我们了解的更多，也应该调整他们的优先级）。图2-1所示的方法可以帮助并指导我们做出决策。如果想划分职责，将更多的权力下放到新成立的自治团队，则类似图2-1这样的简单模型可以为自治团队的本地决策提供信息，并帮助自治团队做出更好的选择，从而使他们与整个公司层面要实现的目标保持一致。

Copyrights © wangwei all right reserved

# 带动其他人员

经常有人问我：“怎么才能让老板相信微服务呢？”这个问题通常来自于这些开发人员：他们已经看到了微服务架构的潜力，并确信微服务就是未来的方向。

通常，当人们不同意某个方案时，是因为他们可能对我们要实现的目标持有不同的观点。重要的是，对于我们要达到的目标而言，我们要和需要一起同行的其他人有共同的理解。如果我们对于目标的立场一致，那么我们至少知道我们仅在如何达到目标方面存在分歧。因此，问题又回到了目标——如果组织中的其他人也有一致的目标，那么他们就更有可能加入微服务的迁移。

实际上，值得更深入的探讨：怎么才能从那些帮助组织变革的知名模型中寻找灵感，以实现既可以帮助我们推销微服务，又可以使微服务落地实施。接下来我们将对其探讨。

Copyrights © wangwei all right reserved

# 组织变革

Dr. John Kotter提出的组织变革的8个步骤是全球管理者变革的基础。John Kotter把变革所需行动提炼为环环相扣的、可理解的步骤，这也是该模型能够具备如此影响力的部分原因。虽然John Kotter的变革模型并非变革方面的唯一模型，但我发现它是最有用的模型。

图2-2丰富的概述了关于领导变革的过程，此处，将不再对其进行过多的介绍<sup>6</sup>。但是，有必要简要概述这些步骤并思考这些步骤将如何在我们考虑采用微服务时给我们提供帮助。

在概述该变革模型之前，我应该注意到：John Kotter的变革模型通常用于建立大规模的组织行为转变。因此，如果只想将微服务带到一个10人的团队，那就大材小用了。不过我发现，John Kotter的模型——尤其是前面提到的8个步骤——即使在这些较小规模的范围内，也很有用。

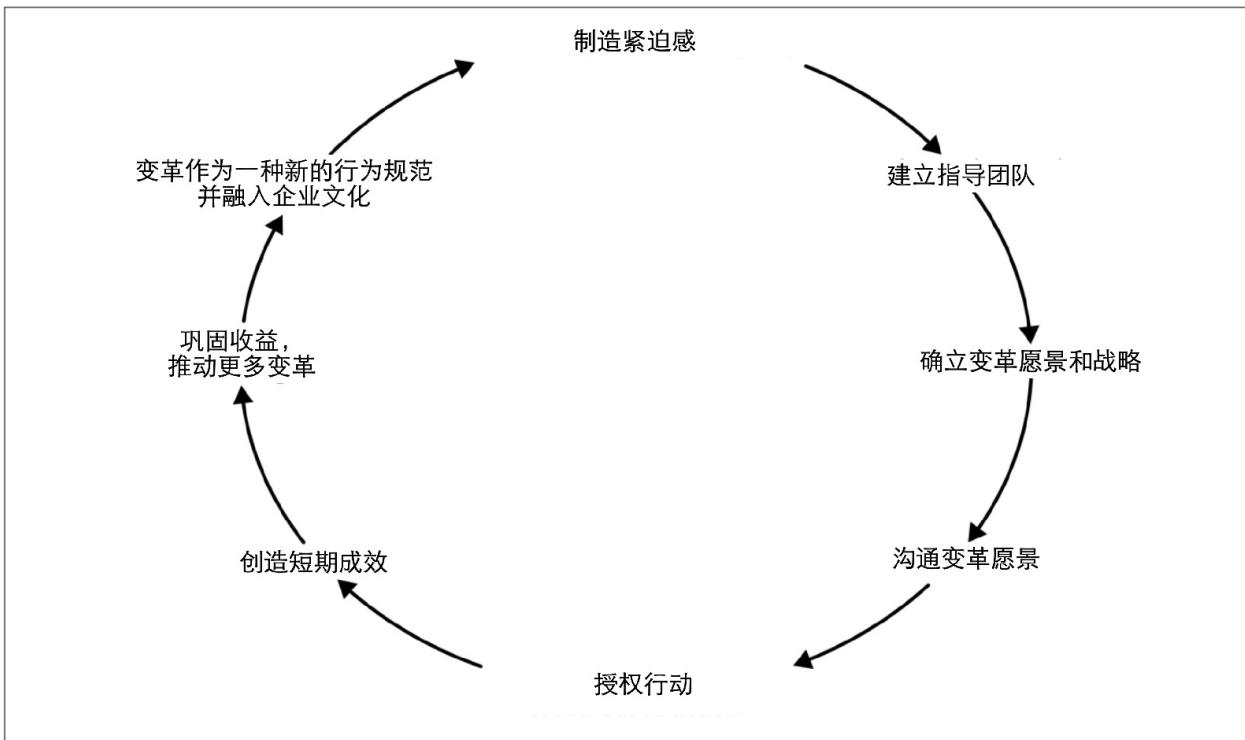


图2-2. Kotter提出的组织变革的8个步骤

# 制造紧迫感

人们可能会认为我们转向微服务的想法是一个好主意。问题是：我们的想法可能只是组织中涌动着的很多好想法之一。让大家都认为确实要采用微服务的诀窍是帮助团队认识到：现在就是进行这一特殊变革的时刻。

寻找“教化”的时机可以帮助我们达到目的。有时，修补羊圈的正确时机是在狼通过窟窿将羊叼走之后，因为人们突然意识到羊被叼走是需要考虑的事情。现在他们意识到羊圈上有一个窟窿：“哦，看那，修补这个窟窿就可以啦！”在危机解决之后的那一刻，我们的建议会在人们的意识中短暂停留一会儿，此时就是推动变革的时机。一旦时间拖的太长，人们就会慢慢忘记痛点以及痛点的原因。

谨记，要做的事情不是对人们说：“我们现在应该做微服务！”我们应该分享待实现的目标的紧迫感——正如我之前所述，微服务不是目标！

# 建立指导团队

推动变革不需要所有人员参与，但需要足够的资源来推动变革的发生。需要确定组织内部哪些人员可以帮助我们推动变革。如果是开发人员，那么可以首先选择我们团队中的同事，或者更高等级的同事——技术主管，架构师或交付经理。投入多少人员来推动变革需要根据变革的范围来定。如果只是要改变自己团队的工作方式，那么确保提供足够的空中掩护<sup>译注1</sup>就足够了。如果想改变整个公司的研发模式，则可能需要公司执行级别的高管来支持（CIO或CTO）。

别人参与进来并帮助我们实现变革可能不容易。无论我们认为这个主意有多好，如果我们从未被人所知，或者从未与别人合作过，别人为什么会支持我们的主意呢？因此，首先要赢得信任。如果有人曾经与我们在小的事情上合作并取得成功，则他们更有可能会支持我们的大创意。

让软件交付之外的人员参与到变革中是一件重要的事情。如果我们所在的企业已经解决了“IT”和“业务”之间的屏障，那么不引入交付之外的人员也是可以的。另一方面，如果这些屏障仍然存在，则可能需要跨越鸿沟在业务部门寻找支持者。当然，如果我们采用微服务架构的重点是解决业务所面临的问题，这将更容易实现微服务的落地。

需要IT部门之外的人员参与到变革的原因在于：我们所做的许多变革都可能对软件的工作方式和行为产生重大影响。例如，我们需要围绕如下的类似场景而给出不同的权衡：

- 系统在故障模式下的行为方式
- 如何解决延迟

例如，在分布式系统中，缓存是一种用来避免服务调用，进而降低系统中关键请求的延迟的好方法。但是，却需要在缓存以及用户可能会看到脏数据之间做出权衡和折衷。这种缓存方案是正确的做法吗？我们可能必须与用户讨论这个问题——并且如果连公司内部的用户体验团队<sup>译注2</sup>都不了解变革的逻辑，这将是一场艰难的讨论。

# 确立变革愿景和战略

到了这一步时，我们已经为变革召集齐了人马，并且已经就希望带来的变化（愿景）以及如何实现目标（战略）达成一致。愿景是一件棘手的事情。愿景既要务实，又要务虚，并且关键是要找到二者之间的平衡点。越大范围的愿景，就需要做更多的工作以让人们参与进来。但是，愿景没有必要必须是高大上的事情，并且愿景也可以在小规模的团队发挥作用——我们必须降低我们的Bug数！。

愿景主要解决我们的目标是什么的问题。战略则解决我们将如何实现目标。微服务则是用来实现我们期望的目标——因此，微服务是战略的一部分。谨记，战略可能会改变。致力于愿景很重要，但是在面对相反的证据下还过分执着于特定战略是危险的，并且这种执着可能导致我们陷入值得关注的[沉没成本谬误](#)。

# 沟通变革愿景

有一个宏大的愿景当然很棒，但也不要宏大的让人们不相信愿景有实现的可能。我最近见到一个大公司的CEO发表声明说：

在接下来的12个月中，我们将通过迁移到微服务并采用云原生技术来降低成本并提高交付速度。

——未透漏姓名的CEO

这家公司中，我与之交谈过的员工，没有人相信CEO声明的愿景有任何实现的可能性。上述声明的部分问题是存在潜在的相互矛盾的目标：全面变革软件交付方式可能会帮助实现更快地交付软件，但是在12个月内完成这样的变革并不会降低成本。在变革的过程，我们可能必须引进新技能，并且在采用新技能之前，变革可能会对生产效率产生负面影响。上述声明的另一个问题是时间范围的描述。对于这个特殊的公司而言，其变革速度足够慢以至于12个月的目标被认为是荒唐可笑的。因此，无论我们分享什么愿景，都必须令人信服。

在分享愿景时，我们可以从小处着手。几年前，我在谷歌参与了一个名为“Test Mercenaries”的计划，其旨在帮助推广测试自动化的实践。该计划开始启动的原因来自于谷歌的某个小组（grouplet）先前在帮助分享自动化测试的重要性的事情上所作出的努力。现在，我们称这种小组为实践社区（*community of practice*）。该计划在分享测试相关信息的早期努力之一是一项名为“Testing on the Toilet”的行动。TotT由固定在厕所门背面的、简短的一页文章组成，以便人们可以“在闲暇时”阅读这些测试的相关信息！我并不建议其他公司也采用这种技术，但在谷歌，TotT的效果确实很好。<sup>7</sup> TotT确实是一种分享那些小而可行的建议的有效方法。

最后一点，目前，人们更倾向于使用Slack之类的系统来取代面对面交流。当要分享类似愿景这种重要信息时，面对面的交流会大大提高效率。面对面分享使我们更容易了解到人们听到这些想法的反应，有助于校准我们的信息并避免误解。即使需要其他形式的沟通以在较大的组织中传播我们的愿景，也要先进行尽可能多的面对面分享。面对面分享将帮助我们更有效地优化我们的信息。

# 授权行动

授权是管理咨询公司<sup>译注3</sup>在帮助他们自己完成工作时的代名词。通常，这意味着一些非常简单的事情——消除障碍。

我们已经分享了自己的愿景并鼓舞了士气，然后会发生什么？变革还是没有发生。最常见的问题之一是：人们忙于现在所做的事情以至于对变革分身乏术。这就是为什么公司经常会为组织引入新人（例如通过招聘或顾问）以便为团队变革提供额外的力量和专业知识的原因。

举一个具体的例子，在采用微服务时，围绕基础设施供应的现有流程可能是一个真正的问题。如果我们对于部署新的产品服务的解决方案涉及到提前三个月订购硬件，那么采用允许按需定制虚拟化执行环境（例如虚拟机或容器）的技术可能会是一大福利。转而采用公有云供应商也可以带来好处。

不过，我确实想回应上一章的建议。不要为了新技术而引入新技术。引入新技术的目的在于解决我们看到的具体问题。一旦确定障碍，就采用新技术来解决这些问题。不要花一年的时间去定义“完美的微服务平台”，而到头来却发现它实际上并不能解决我们遇到的问题。

作为Google的“Test Mercenaries”计划的一部分：

- 我们最终创造了很多框架，以简化测试套件的创建和管理
- 我们促进了可视化的测试作为CR(code review)系统的一部分
- 我们甚至最终推动并创造了公司范围的新CI工具，以使测试执行更加容易。

不过，我们并不是一次完成了所有操作。我们与几个团队合作，了解了痛点，从中吸取了教训，然后花时间投入到新的工具。我们也是从小事做起——解决更容易的创建测试套件是一件非常简单的事情，但是改变公司范围的CR系统则是一个更大的问题。在取得成功之前，我们一直没有尝试过解决更大的问题。

# 创造短期成效

如果需要花很长时间才能看到变革的成果，人们就将对愿景失去信心。因此，需要获得一些短期的胜利。最初，将重心放在那些小的、容易的、触手可及的事情上将有助于提供变革的动力。对于微服务拆分而言，可以轻松地从单体应用中抽取出来的功能应该具有较高的优先级。但是，正如我们已经确定的那样，微服务本身并不是目标。因此，我们需要对功能抽取的难易程度和其带来的收益进行权衡。我们将在本章的稍后部分再次讨论这种权衡。

当然，如果我们选择了容易的事情，并且最终遇到了很大的问题，此时可能需要审视我们的战略，并且可能让我们重新考虑自己在做什么。这完全没有任何问题！关键是，如果我们首先专注于简单的事情，那么我们很可能会早日洞悉这些问题。犯错是很正常的——对于犯错，我们所能做的就是理清问题以确保我们尽快从那些错误中学习。

# 巩固收益，推动更多变革

一旦取得成功，最重要的就是不要不思进取。如果不继续努力，唯一的成果可能就是短期成果。在成功（失败）后停下来反省一下很重要，这样我们就可以思考如何继续推动变革。当我们位于公司的不同部门时，我们可能需要不同的方法。

随着微服务迁移工作的深入，我们可能会发现更难以前进。起初，我们可能会推迟数据库的拆分，但也不能永远拖延。正如我们将在第4章中探讨的那样，我们可以使用许多技术，但是需要仔细考虑哪种方法是正确的方法。谨记，单体应用中某个区域的拆分方法可能对其他部分的拆解无能为力，我们需要不断尝试新的方法以取得新的进展。

# 变革作为一种新的行为规范并融入企业文化

通过持续迭代、推动变革、并分享成功（和失败）的故事，新的工作方式将开始一切如常。我们工作的很大一部分是与同事，其他团队和其他人分享变革的故事。通常，一旦我们解决了一个难题，就会进入下一个难题的解决。为了实现规模化变革并坚持下去，持续寻找在组织内部分享信息的方法至关重要。

随着时间流逝，新的做事方式成为工作的方式。如果我们认为公司在采用微服务架构上还有很长的路要走，微服务是否是正确的方案就不再是一个问题。这就是目前的工作方式，并且公司了解怎样才能做好。

继而会产生一个新问题：一旦新的方式成为已确立的工作方式，我们如何确保未来更好的方案有发展的空间，甚至取代现有的工作方式？

---

<sup>6</sup>. Kotter's change model is laid out in detail in his book *Leading Change* (Harvard Business Review Press, 1996). ↪

<sup>7</sup>. For a more detailed history of the change initiative to roll out testing at Google, Mike Bland has an [excellent write-up](#) that is well worth a read. Mike wrote up a [detailed history](#) of Testing on The Toilet as well. ↪

译注<sup>1</sup>. Macmillan Dictionary将空中掩护定义为“一项军事行动，在这种行动中，飞机会定期飞过某个区域，以阻止敌人的进攻”。为团队提供空中掩护意味着为团队创造一些安全空间以便让他们实验和学习，并

且当实验失败时，需要保护他们免受他人的质疑。 ↵

译注<sup>2</sup>. 原文为：The people who champion the cause of your users inside your organization. 公司内部负责倾听用户声音，为用户代言的团队或部门，是综合了产品团队，用研团队，客服团队等职能的人群。 ↵

译注<sup>3</sup>. 管理咨询公司是指从事软科学研究开发、并出售“智慧”的公司，又称“顾问公司”。管理咨询公司属于商业性公司，接收委托者的意向和要求，运用专门的知识和经验，用脑力劳动提供具体服务。 ↵

Copyrights © wangwei all right reserved

# 增量迁移的重要性

If you do a big-bang rewrite, the only thing you're guaranteed of is a big bang.

——Martin Fowler（《重构：改善既有代码设计》的作者）

如果已经确定拆分现有的单体系统是正确的选择，我强烈建议每次抽取一点点功能，逐步拆解这些单体应用。这种增量迁移的方法将帮助我们在实践中了解微服务，同时还可以限制所犯错误的影响面（我们肯定会在迁移过程中犯错）。可以把我们的单体应用视为一整块大理石。我们可以一下就把整块石头炸碎，然而这样的方式很少有完美的结局。采用蚕食的方法则更为明智。

问题在于，将可拆解的单体系统（*nontrivial monolithic system*）[译注1](#)迁移  
到微服务架构的尝试成本可能很大，而且如果我们一次完成所有的操作，  
我们很难从迁移中得到良好的反馈——哪里是正常的，哪里是异常的。将  
迁移过程拆分为很多更小的步骤则会让迁移更容易，我们可以分析每一个  
迁移步骤并从中学习。正因如此，在敏捷出现之前，我就一直是迭代交付  
的忠实拥护者——因为我承认我会犯错误，因此我需要一种方法来降低错  
误的级别。

任何迁移微服务架构的操作都应谨记如下的原则：

- 将宏大的工程拆分成很多小的步骤
- 拆分之后的每个步骤都可以执行并从中学习
- 如果某一步被证明是错误的，则这也仅仅是一小步而已
- 对于每一步而言，无论对错，我们都可以从中学习，然后为我们的下

## 一步骤提供信息

如前所述，将事情分解成较小的部分还可以让我们识别出快速成果并从中学习。这会帮助我们让下一步变得更容易，同时也帮助我们提升工作动力。通过一次拆分一个微服务，我们无需等待大规模的部署，同时还可以逐步释放微服务带来的价值。

对于正在考虑微服务的用户提供的建议而言，增量迁移已经成了最优先的建议。如果我们认为这是个好主意，那么就从小处着手。选择一两个功能领域，用微服务实现它们，将其部署到生产环境，并思考其是否有效。在本章的后面部分，将提供一个[模型](#)，该模型可以确定应该从哪些微服务开始迁移。

# 至关重要的是部署到生产环境

需要特别注意的是，只有把抽取的微服务部署到生产环境并使其发挥作用时，整个抽取工作才是完整的。增量抽取的部分目标是使我们有机会学习和理解拆分本身的影响。只有在服务投入生产环境之后，我们才能学习到大多数的重要经验。

微服务拆分可能导致很多问题，例如：定位的问题，链路跟踪的问题，延迟的问题，参照完整性<sup>译注2</sup>的问题，级联故障的问题……。其中的绝大多数问题只有在微服务投入生产环境之后才会出现。接下来的章节，我们将研究部署到生产环境时限制问题发生的影响的技术。如果我们进行较小的更改，发现（并修复）迁移产生的问题就会更加容易。

---

译注<sup>1</sup>. 在线性代数中，非平凡解是齐次方程或齐次方程组的非零解。对应的，平凡解就是显而易见的解、没有讨论的必要但是为了结果的完整性仍需要考虑的结果，例如0解。因此，非平凡的单体系统意味着该系统可以拆解。 ↵

译注<sup>2</sup>. 参照完整性是关系型数据库的完整约束之一，属于数据完整性的一种，其余的完整性约束还有：实体完整性、用户自定义完整性。

↵

# 变革的成本

在本书中，我提到了从小处着手、增量变更的很多原因，但是最主要的原因之一是：我们要了解每一次变更的影响，并在必要的时候修改变更的方案。增量变更可以更好地降低犯错的代价，但是却并不能完全避免犯错。在变更过程中犯错不可避免，我们应该拥抱变更过程中所犯的错误。我们还应该了解如何才能更好的降低错误的代价。

# 可逆决策和不可逆决策

亚马逊首席执行官Jeff Bezos在2015年度的致股东信中提出了对亚马逊工作方式的有趣见解。信中包含如下的精华部分：

有些决策像一扇单向门，是至关重要的、不可逆的或几乎不可逆的。因此，这些决策必须经过深思熟虑与磋商，然后有条不紊地、谨慎地、缓慢地推进。如果我们穿过这扇门，但是却不喜欢在另一边看到的东西，我们也没有办法再回到从前。我们称其为**1型决策**。然而大多数决策却并非如此：它们像一扇双向门，是可变的、可逆的。如果我们做了一个次优的**2型决策**，就没必要过度看重结果的好坏。我们可以重新打开这扇门，然后再回到从前。拥有决策权的高管或者小组织有权而且理所应当地快速做出**2型决策**。

——Jeff Bezos, 致亚马逊股东的信（2015）

Bezos继续说，那些不经常做决定的人可能会陷入一个陷阱：像对待**1型决策**一样对待**2型决策**。这导致一切都变成生与死的问题，所有事情都变成了一项重大任务。问题在于：采用微服务架构会带来很多有关如何做的选择，这意味着需要比以前做更多的决策。如果不习惯这一点，我们可能会发现自己陷入了这个陷阱，同时进度也将陷入停顿。

这些术语并非惟妙惟俏，并且也很难记住**1型**或**2型**的实际含义，因此我更喜欢用不可逆和可逆来代替**1型**和**2型**。<sup>8</sup>

尽管我喜欢这个概念，但我认为决策并非总是这二者中的其中之一，我认为需要更细粒度的决策类型。我宁愿将不可逆决策和可逆决策视为位于决策频谱的两端，如图2-3所示。

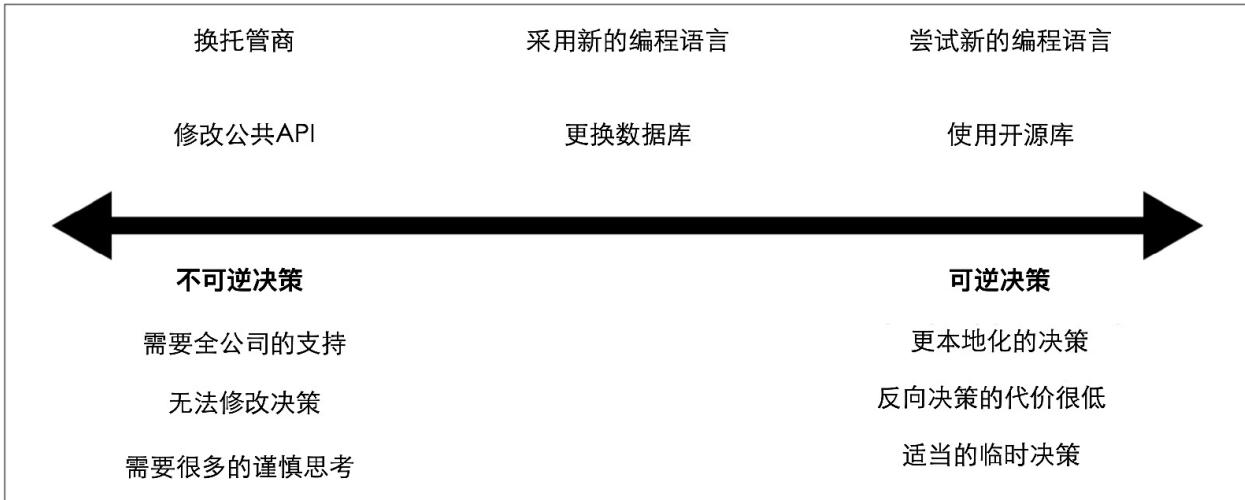


图2-3. 可逆决策和不可逆决策的差异及其例子

起初，评估我们在决策频谱上的位置可能具有挑战性，但是从根本上讲，如果我们后续决定改变主意的话，那么一切都将回到对影响面的理解。决策越接近不可逆决策，后续改变决策方向产生的影响就越大。

实际情况是，作为微服务迁移的一部分，我们做出的大量决策都将朝着可逆决策的方向发展。软件可能会经常发生回滚或撤消，我们可以回滚软件的变更或回滚软件的部署。我们需要做的是：考虑后续改变主意的成本。

不可逆的决策需要更多的投入、更谨慎的思考，我们应该（确实需要）花更多的时间去决策。我们越靠近决策频谱的右端——朝着可逆决策的方向——就越可以依赖接近问题的同事做出正确的决策。我们知道，即便他们做出了错误的决策，后续修复该错误也很容易。

# 从更简单的地方开始试验

在代码库中移动代码所涉及的成本非常小。很多工具可以支持我们这么做，并且如果在移动过程中产生错误，也可以很快解决这些错误。但是，拆分数据库的工作量却很大，同时，回滚数据库变更也同样复杂。类似的，拆解过度耦合的服务，或必须完全重写多个消费者使用的API也是一项艰巨的任务。巨大的变更成本意味着这些操作会增加风险。我们如何管理这种风险？我的方法是：在影响最小的地方试错。

我倾向于在变更成本和犯错成本都尽可能低的地方做很多思考：可以采用白板：

- 在白板上勾勒出我们的设计
- 观察在跨服务边界时运行用例会发生什么，以我们的唱片店为例：
  - 想象当客户搜索唱片、在网站上注册或购买专辑时会发生什么？
  - 调用了什么？
  - 是否看到了奇怪的循环引用？
  - 是否看到两个服务之间的调用过于频繁（可能表明应将它们合二为一）？

---

<sup>8</sup>. Hat tip to Martin Fowler for the names here! ↪

# 我们从哪里启程

OK，我们已经谈到了清楚的阐明目标和理解可能存在的、潜在的权衡折衷的重要性。

接下来呢？

接下来，我们需要了解那些想要抽取到服务中的功能，以便我们可以开始理性地思考我们接下来要创建什么样的微服务。在拆解现有的单体系统时，我们需要某种形式的逻辑分解，领域驱动设计（*DDD: domain-driven design*）就可以派上用场了。

Copyrights © wangwei all right reserved

# 领域驱动设计

在第一章中，我介绍了领域驱动设计（DDD）。DDD是帮助定义服务边界的重要概念。开发领域模型有助于确定如何安排拆分工作的优先级。在图2-4中，我们为Music Corp提供了一个示例性的高层次（high-level）领域模型<sup>译注1</sup>。从图2-4可以看到：作为领域建模结果的一组界定的上下文。我们可以清楚地看到这些界定的上下文之间的关系，可以想象它们将代表组织内部的交互。

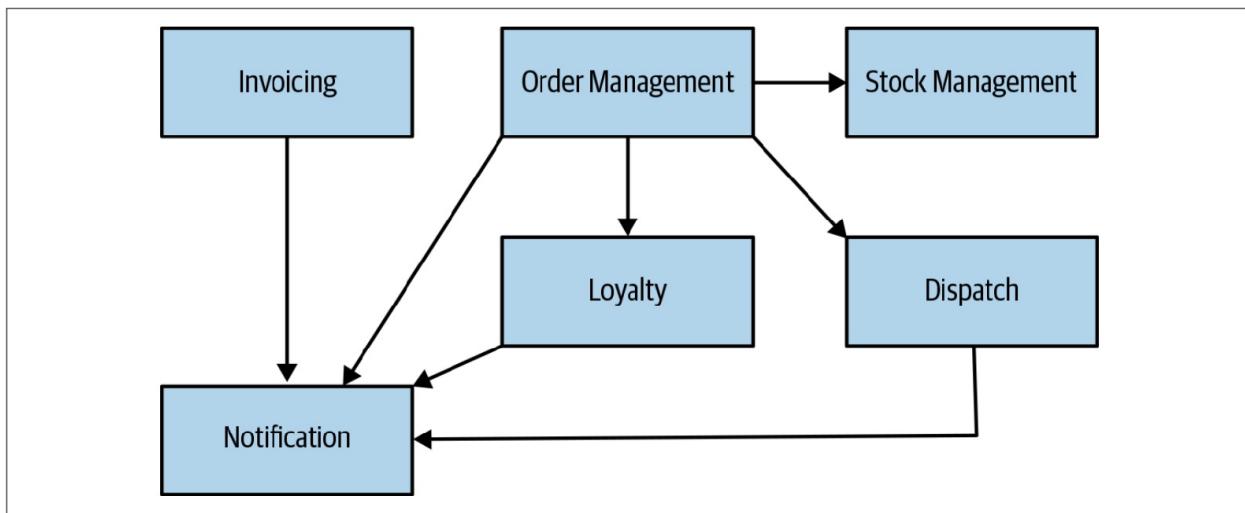


图2-4. Music Corp的界定的上下文及其之间的关系

每一个界定的上下文都代表一个拆分之后的潜在的服务单元。如前所述，界定的上下文是定义微服务边界的一个很好的起点。既然如此，我们就有了待确定优先级的事项清单。我们也可以从界定的上下文之间的关系形式获得有用的信息，这些信息可以帮助我们在抽取不同的功能时评估其相对难度。我们很快会再讨论这个想法。

我会考虑提出一个领域模型，这是构建微服务迁移的必不可少的步骤。通常令人生畏的是：许多人没有构建这种模型的直接经验。他们还非常担心这会涉及多少工作。实际上，虽然经验可以极大地帮助构建这样的逻辑模型，但是即使付出少量的努力也可以获得一些真正有用的收益。

# 还要走多远

拆解现有系统往往令人望而却步。许多人可能会构建并继续构建系统，——十有八九——可能有更多的人在日常工作中使用这种方式。在确定范围的情况下，尝试提出整个系统的详细领域模型可能会很艰巨。

在确定从何处开始拆解时，我们需要从领域模型中获取到足够的信息来做出合理的决策。认识到如上的这一点非常重要。我们对系统中最需要关注的部分可能已经有所了解。因此，为单体系统提出一个高层次功能分组的通用模型就足够了。然后，再从中选择那些我们想要深入研究的部分。如果仅关注系统的一部分，可能会忽略需要解决的较大的系统性问题——这种风险始终存在。但是，我也不会为此而困惑——我们无需一开始就不犯错误，我们只需要利用足够的信息来做出明智的后续操作即可。我们可以（而且应该）在学习中不断完善自己的领域模型，并在推进过程中不断更新以反映新的功能。

# 事件风暴

由Alberto Brandolini创建的Event Storming是一项由技术和非技术的利益相关者（stakeholder）[译注2](#)共同定义领域模型的一项团队协作活动。事件风暴以自底向上的方式进行：

- 参与者首先定义领域事件（*Domain Events*），即系统中发生的事情
- 然后，将这些事件分组并聚合
- 然后将聚合后的事件分组为界定的上下文

Event Storming并不意味着必须先构建一个事件驱动（*event-driven*）的系统，了解这一点非常重要。相反，Event Storming重点在于了解系统中发生了哪些（逻辑）事件——像系统的利益相关者[译注2](#)那样明确事件的事实。虽然领域事件可以映射为事件驱动系统的触发事件（触发事件是事件驱动系统的一部分），但是可以用不同的方式来表示领域事件。

Alberto用Event Storming技术真正关注的是：集体定义模型的想法。该活动的输出不仅仅是模型本身，还有大家对模型的共同理解。为了使该过程可以正常进行，我们需要找到合适的利益相关者[译注2](#)来参与讨论，而这通常是最大的挑战。

更详细地探讨Event Storming已经超出了本书的讨论范围，但是Event Storming技术是我用过且非常喜欢的一种技术。如果想探索更多Event Storming的相关内容，可以阅读Alberto的[Event Storming简介](#)（该书目前还在撰写中）。



# 使用领域模型确定优先级

可以从类似图2-4的图中获得一些有用的见解。根据上游或下游依赖的数量，可以推断出抽取不同功能的难易程度。例如，如果我们考虑抽取**Notification**，那么我们可以清楚地看到许多入站依赖（*inbound dependencies*）[译注3](#)。如图2-5所示：系统的许多部分都需要使用Notification。因此，如果要抽取新的Notification服务，则需要处理大量现有代码——将从本地调用现有的Notification改为其微服务调用。我们将在第3章中探讨有关此类变更的多种技术。

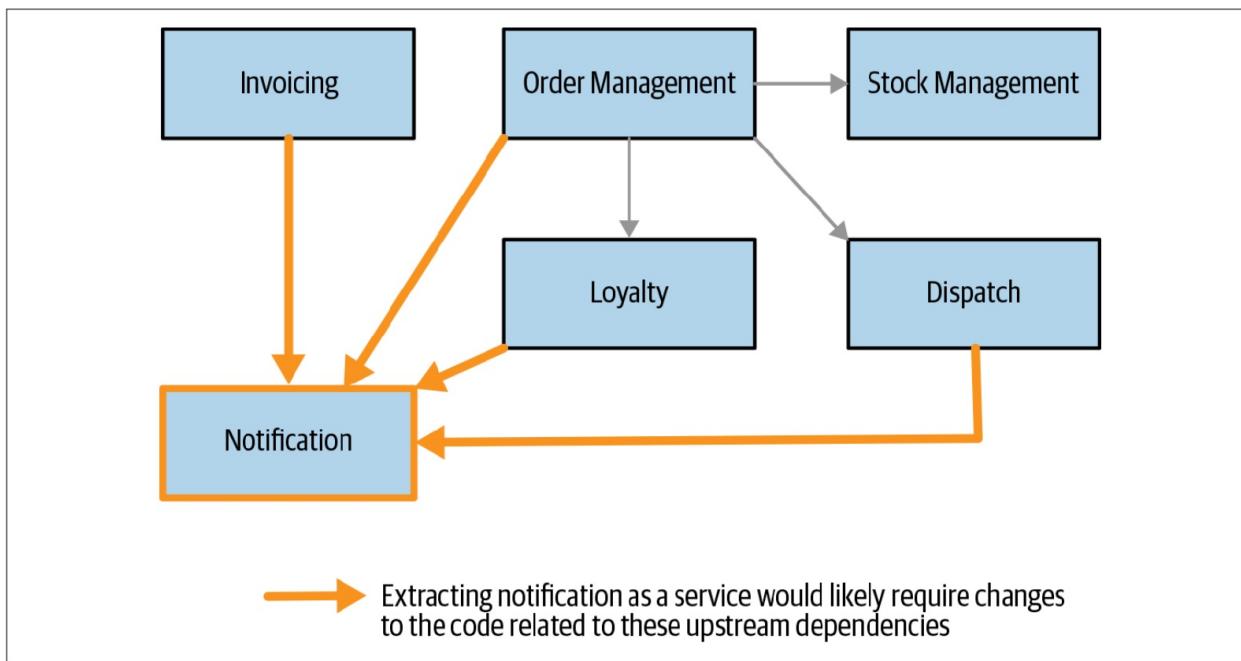


图2-5. 从领域模型看，Notification功能存在逻辑耦合，因此抽取难度较大  
因此，Notification可能不是一个好的开始。另一方面，如图2-6所示，  
Invoicing功能很可能是更容易抽取的系统行为。Invoicing没有入站依赖，  
这对单体应用所必须进行的变更量将会减少。这种情况下，类似[绞杀者](#)的

模式可能是有效的，因为我们可以轻松地在这些入站调用到达单体之前对其进行代理。我们将在下一章探讨绞杀者模式及其他很多模式。

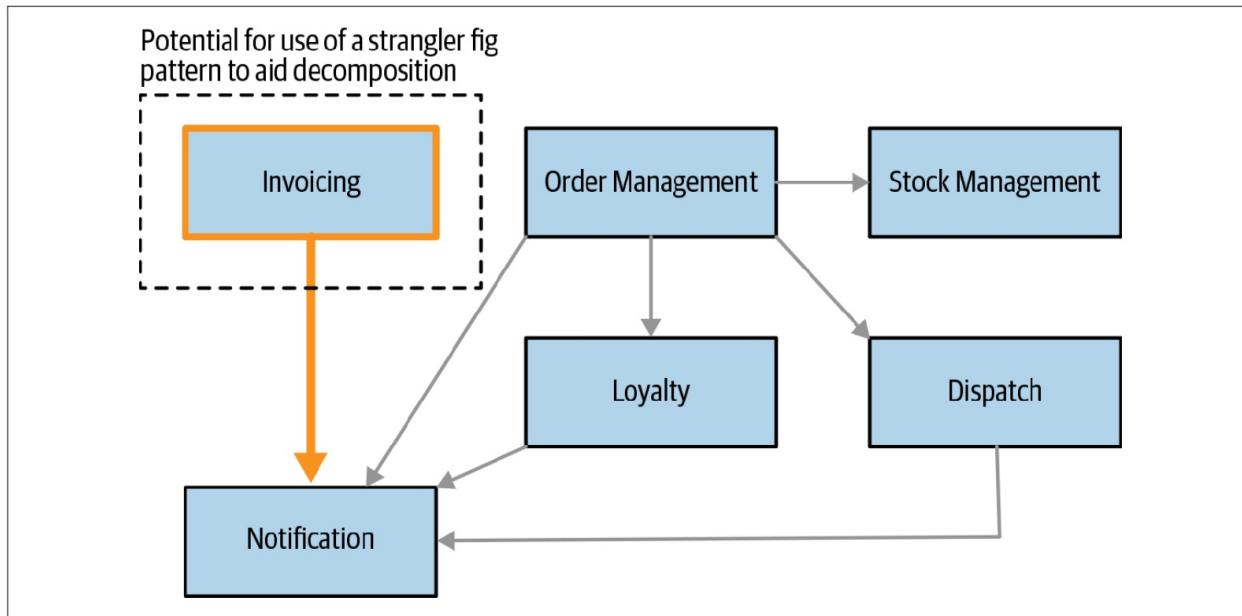


图2-6. 发票系统看起来更容易抽取

在评估抽取难度时，如上的关联关系是一种开始的好方法。但是我们必须理解：领域模型代表了现有系统的逻辑视图。不能保证系统的底层代码结构也是以领域模型的方式来组织。这意味着逻辑模型可以指导我们了解功能耦合的多少，但是我们可能仍然需要查看代码本身，以更好地评估功能之间的耦合。类似本节的领域模型不会显示哪些界定的上下文会把数据存储在数据库中。我们可能会发现，Invoicing功能管理着大量的数据，这意味着我们需要考虑数据库拆分对其带来的影响。正如我们将在第4章中讨论的那样，我们可以并且应该将单体系统的数据存储区分开。但是我们不想在我们的前几个微服务中一道讨论数据库的拆分。

因此，我们能够通过领域模型这面镜子来观察哪些是容易的事情以及哪些是比较难的事情。区分事情的难易程度是有价值的行为——毕竟我们希望获得快速的胜利！但是，我们必须记住，我们只是把微服务视为实现特定

目标的一种方法。实际上，我们可能会发现发票功能确实是一个简单的开始，但是如果我们的目标是帮助缩短上市时间，并且发票功能又几乎没有改变，那么首先抽取发票功能的方式并没有有效利用时间。

因此，我们需要将事情的难易程度与微服务拆分带来的好处相结合。

---

译注<sup>1</sup>. 高层次设计（high-level design）是整个系统的设计——包括系统架构和数据库设计。它描述了该系统的各个模块和功能之间的关系。低级别设计（low-level design）定义了用于该系统的每个部件的实际逻辑。 ↵

译注<sup>2</sup>. stakeholder：包括这样的个人和组织，他们或者积极参与项目，或者其利益在项目执行中或者成功受到积极或消极影响。Stake的直接翻译是“筹码”或“赌注”，所以“Stakeholder”可以直接翻译成为“拿着筹码的人”。大部分文章会翻译为“项目干系人”，但是这个翻译会更加让人迷惑。为了更加直观和阅读通顺，这里统一翻译为“利益相关者”。

↵

译注<sup>3</sup>. There are two main varieties of dependencies: outbound and inbound. Outbound means that the project relies on one activity being carried out by a particular team so as to proceed to another task or deliver results. Conversely, inbound means that other parties are relying on a particular team to finish a task before they deliver results. An example of an outbound dependency is when one has to wait for the legal department to review and approve a particular document before being allowed to make any adjustments to the system. ↵

# 融合模型

我们希望获得一些快速的成功，以实现如下的目标：

- 尽早取得进展
- 营造一种有动力的感觉
- 早日获得方法有效性的反馈

这将迫使我们选择更容易抽取的功能。但是，我们还需要从拆解中获得收益——我们如何将其纳入我们的思考范围？

从根本上讲，这两种形式的优先级都是有意义的，但是我们需要一种机制来将两者统一起来并做适当的权衡折衷。我喜欢使用[图2-7](#)所示的简单结构来解决这个问题。沿着图中所示的两个坐标轴来放置每个要抽取的候选服务。x轴表示拆解服务将带来的价值，y轴表示拆解服务的难易程度。

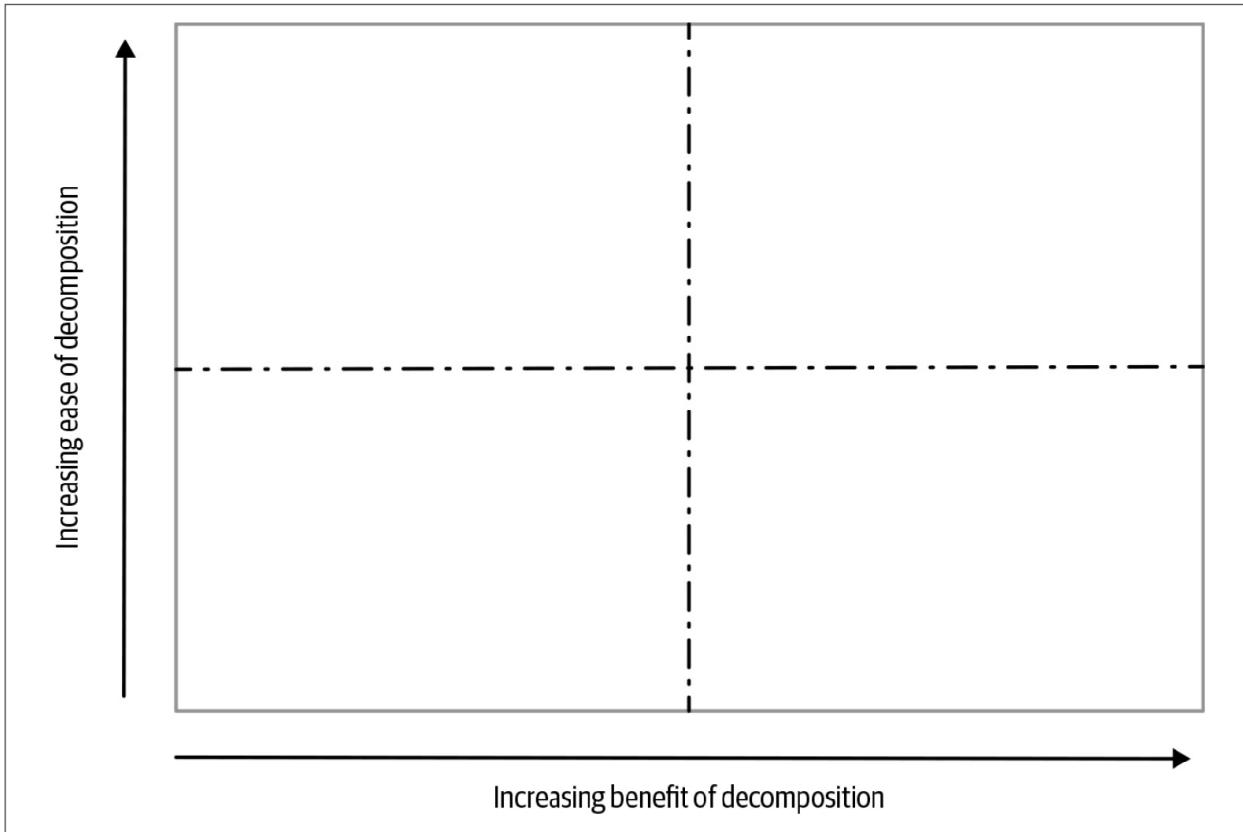


图2-7. 确定服务拆分优先级的简单模型

像一个团队一样协作完成图2-7所示的优先级确定的过程，我们可以确定最佳抽取对象。如图2-8所示，和所有的、好的四象限模型一样，最佳抽取对象位于右上角象限。右上角象限中的功能（包括Invoicing）代表了我们认为应该易于提取、并且还将带来收益的功能。因此，从该象限的功能组中择一个（或两个）服务作为要抽取的第一个服务。

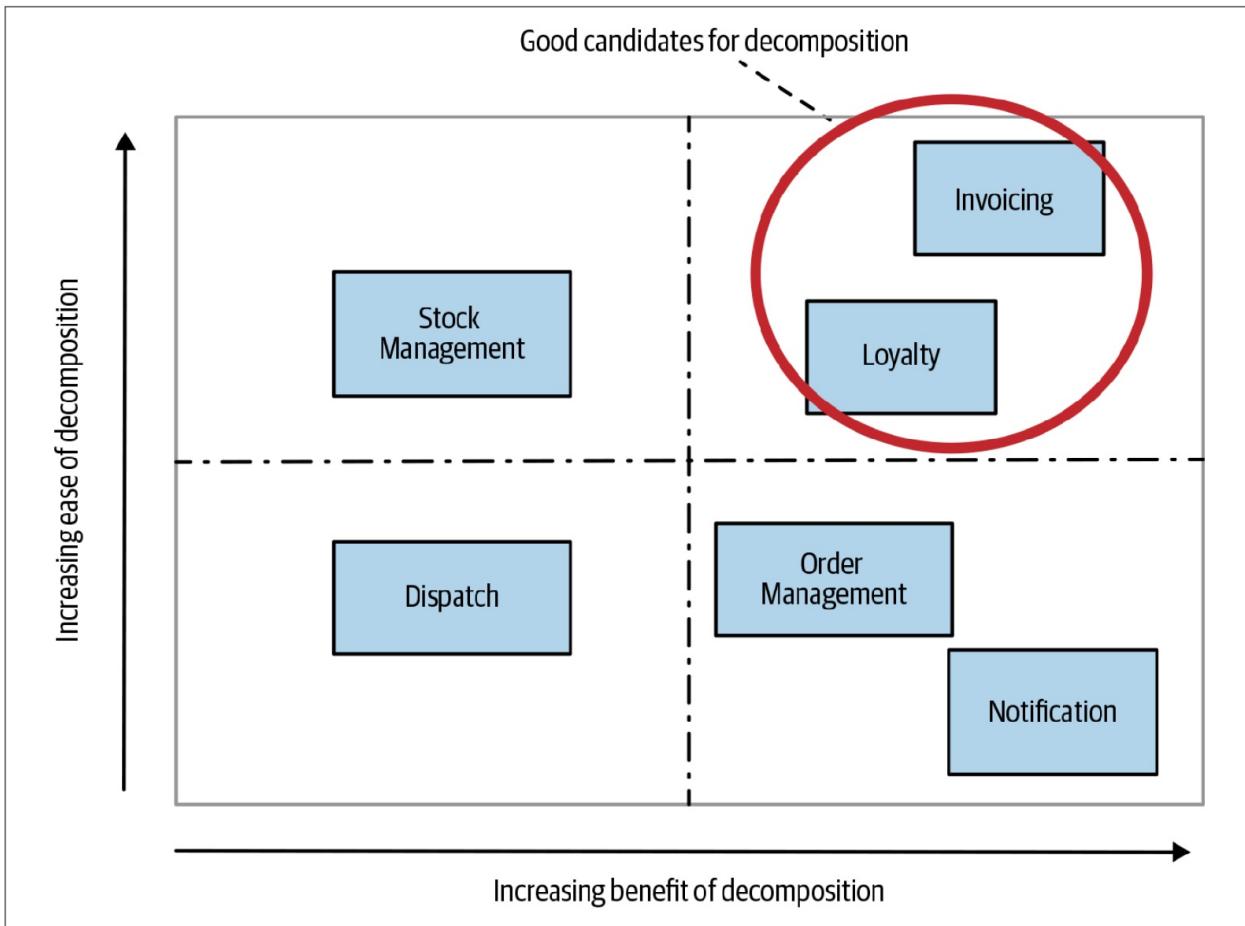


图2-8. 使用优先级四象限确定优先级的例子

一旦开始开始进行变更，我们会学到更多。我们认为容易的事情会变得很难。我们认为困难的事情会变得容易。这种情况非常自然！但这确实意味着如下的事情非常重要：重新了解如上的确定优先级的工作，并在我们了解了更多信息后重新规划优先级。也许随着我们的不断学习，我们会意识到抽取Notifications功能可能比想象中容易。

Copyrights © wangwei all right reserved

# 重组团队

本章之后，我们将主要着眼于需要对架构和代码进行的变更，以成功实现向微服务的过渡。但是，正如我们已经探讨的那样，使架构和组织保持一致对于充分利用微服务架构至关重要。

但是，我们可能处于需要组织变革以利用这些新技术的情况。尽管对组织变革的深入研究不在本书的讨论范围之内，但在我们深入探讨技术之前，我会对组织变革方面提供些许意见。

# 转变组织结构

过去，我们围绕核心技能构建IT组织。Java开发人员与其他Java开发人员处于一个团队中。测试人员与其他测试人员处于一个团队。DBA自己一个团队。在开发软件时，需要从不同的团队指派人员来为这些周期很短的计划而工作。

因此，软件开发需要团队之间的多次交接。商业分析师会与客户沟通并发现客户需求。然后，分析师写下需求并将其交给开发团队。开发人员会完成一些工作并将其交给测试团队。如果测试团队发现问题，则将其打回。如果测试通过，则可以继续交给运维团队来部署。

这种组织的孤岛化似乎很熟悉。考虑我们在上一章中讨论的分层架构。进行简单更改时，分层架构可能需要更改多个服务。组织孤岛也是如此：开发或更改软件所需的团队越多，花费的时间就越长。

需要打破这些组织孤岛。对于许多组织而言，专职的测试团队现在已成为历史。相反，测试专家正在成为交付团队不可或缺的一部分，从而开发和测试可以更加紧密地合作。DevOps运动也部分导致许多组织从集中式运维团队转移到其他方面，而将更多的运维责任推到交付团队。

在将这些专职团队的角色推到交付团队的情况下，这些集中式团队的角色已经转移：从自己完成工作，变成帮助交付团队完成工作。这可能需要：不同的专家深入到交付团队，创建自助服务工具，提供培训以及其他一系列活动。这些集中式团队的责任已经从做事转移到了赋能。

因此，我们看到越来越多的独立自主团队能够比以往更多地负责端到端的交付周期。他们关注的重点是产品的不同区域，而不是特定的技术或活动——就像我们从面向技术的服务转向围绕业务垂直功能的服务建模一样。现在，需要了解的重要一点是：尽管这种转变是多年来一直很明显的明确趋势，但却并没有普及开来，也不是一种可以快速完成的转变。

# 没有放之四海而皆准的模式

在本章的开头，我们讨论了是否使用微服务应取决于面临的挑战以及希望带来的变化。变更组织结构也同样重要。了解我们的组织是否需要变革以及需要如何变革，依赖于我们的背景、工作文化和团队人员。这就是为什么仅复制别人的组织设计会特别危险的原因。

之前，我们非常简短地谈到了Spotify模型。Henrik Kniberg和Anders Ivarsson于2012年发表了著名的论文“[Scaling Agile @ Spotify](#)”。文章中提及的Spotify如何组织自己得到了越来越多的关注。该文章普及了分队（*Squads*），分会（*Chapters*）和协会（*Guilds*）的概念，这些术语现在软件开发行业中很普遍了（尽管被误解了）。最终，人们称这种模型为“Spotify模型”，尽管Spotify从未使用过“Spotify模型”这样的术语。

随后，大量公司采用了Spotify这样的结构。但是，与微服务一样，许多公司还没有充分思考就倾向于使用Spotify模型，此处的思考包括：Spotify的运营环境，商业模型，面临的挑战或公司的文化。事实证明，对于一家瑞典的音乐流媒体公司可以运作良好的组织结构可能并不适用于投资银行。此外，“Scaling Agile @ Spotify”还简要介绍了Spotify在2012年的运作方式，并且此后情况发生了变化。事实证明，甚至连Spotify也不使用Spotify模型。

同样的道理也适用于我们微服务的变革工作。绝对可以从其他组织的工作中汲取灵感，但不要以为对其他人有用的东西会在我们的背景下起作用。关于Spotify模型，就像Jessica Kerr所说：“[复制问题，而不是答案](#)”。Spotify的组织结构反映了其为解决其问题而进行的变革。在我们做事、尝

试新事物时，学习那种灵活的、质疑的态度，但要确保我们应用的变革是植根于：对公司的理解，对公司需求的理解，对公司员工的理解，和对公司文化的理解。

举一个具体的例子，我看到很多公司对他们的交付团队说：“现在你们都需要部署软件并提供24/7的全天候支持。”这可能是及其混乱的并且毫无帮助。有时，做出大胆的声明可能是使事情前进的好方法，但要为其可能带来的混乱做好准备。如果我们在这样的环境中工作：开发人员习惯于朝九晚五（9-5）的工作，而不是随时待命，从未在支持或运维环境中工作，并且对SSH也并未了如指掌。那么如上的声明是一种疏远员工并使他们离职的一种很好的方式。但是，如果我们认为24/7的全天候支持是适合我们的组织的正确举动，那就太好了！将其作为一种愿望、作为我们要实现的目标来谈论，并说明原因。然后与员工一起努力实现该目标。

如果真的想转向更加完全拥有软件整个生命周期的团队，需要认识到团队的技能需要改变。可以向团队提供帮助和培训，也可以向团队增加新成员（把当前运维团队的成员嵌入到交付团队）。就像我们的软件一样，无论要带来什么变化，我们都可以以增量变化的方式实现这一目标。

### DevOps并不意味着没有运维

围绕DevOps，有很多普遍的误区。有人认为DevOps意味着开发人员执行所有操作，而无需运维人员。事实并非如此。从根本上说，DevOps是一种文化运动，其基础是打破开发和运维之间的障碍。我们可能仍然希望由专家担任类似运维的这些角色，或者可能不需要。但是无论想做什么，都希望促进与交付软件有关的人员达成共识，无论他们的具体职责是什么。

有关此方面的更多信息，我推荐参考《团队拓扑》<sup>9</sup>，该书探讨了DevOps的组织结构。有关该主题的另一个很棒的资源是《Devops手册》<sup>10</sup>，但是该手册的涉及面更广。

# 做出改变

因此，如果不是单纯的复制别人的组织结构，那应该从何处开始呢？当与正在改变交付团队角色的组织合作时，我打算从明确列出该公司交付软件所涉及的所有行为和职责开始。然后，将这些行为映射到现有的组织结构。

如果已经对生产路径进行了建模（我非常支持），则可以在现有视图上覆盖这些行为和职责的所有权边界。另外，如图2-9所示的一些简单的事情，也可以很好地工作。仅从所有涉及到的角色中选择**利益相关者**，然后组成一个小组来集体“头脑风暴”公司软件发布的所有行为活动。

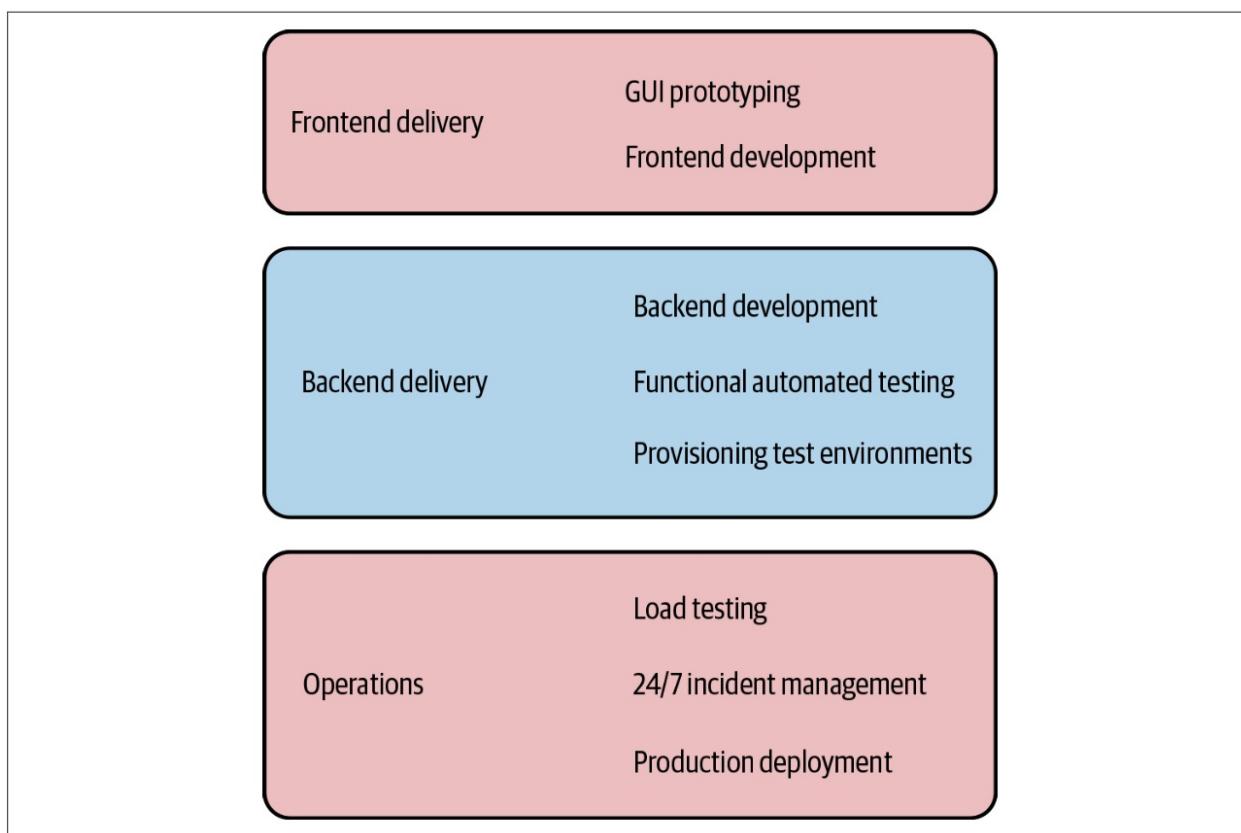


图2-9. 一个交付相关职责的子集以及如何将其映射到现有的团队

对当前“现状”拥有图2-9所示的这种理解非常重要，因为这可以帮助每个人对所涉及的所有工作达成共识。孤岛性组织的本质是，当处于不同的部门时，可能很难理解另一个部门的工作方式。我发现图2-9所示的方式确实有助于组织真实的认识变化的速度可以有多快。我们可能会发现，并非所有团队都是平等的——从测试到部署的过程，有些团队可能已经为自己做了很多事情，有些团队可能完全依赖其他团队。

如果交付团队已经在为自测和用户测试而自行部署软件，那么部署到生产环境的步骤可能不会那么大。另一方面，我们仍然必须考虑如下事情带来的影响：tier 1级支持（随身携带传呼机）[译注1](#)，定位线上问题等。这些技能依赖人们多年工作的积累，期望开发人员在一夜之间就具备这些技能是完全不现实的。

一旦有了现状图，就可以在某个合理的时间范围内重新规划未来的情况。我发现需要六个月到一年的时间来尽可能深入的详细讨论：

- 哪些职责正在交接？
- 如何实现这种过渡？
- 要实现这一转变需要什么？
- 团队需要什么新技能？
- 要进行的各种更改的优先级是什么？

以图2-9所示的现状为例，在图2-10中，我们决定合并前端团队和后端团队的职责。我们还希望团队能够自己提供测试环境。但是要做到这一点，运维团队需要提供一个自助服务平台供交付团队使用。我们希望交付团队最终能够处理自己软件的所有问题，因此我们希望使团队更加满意所涉及到的工作。让交付团队拥有自己的测试部署是一个很好开端。我们还决定，交付团队将在工作日内处理所有的软件问题。因为在工作日，运维团队可以为交付团队提供指导，从而可以有机会为交互团队提供一个安全的环境以加速组织融合的进度。

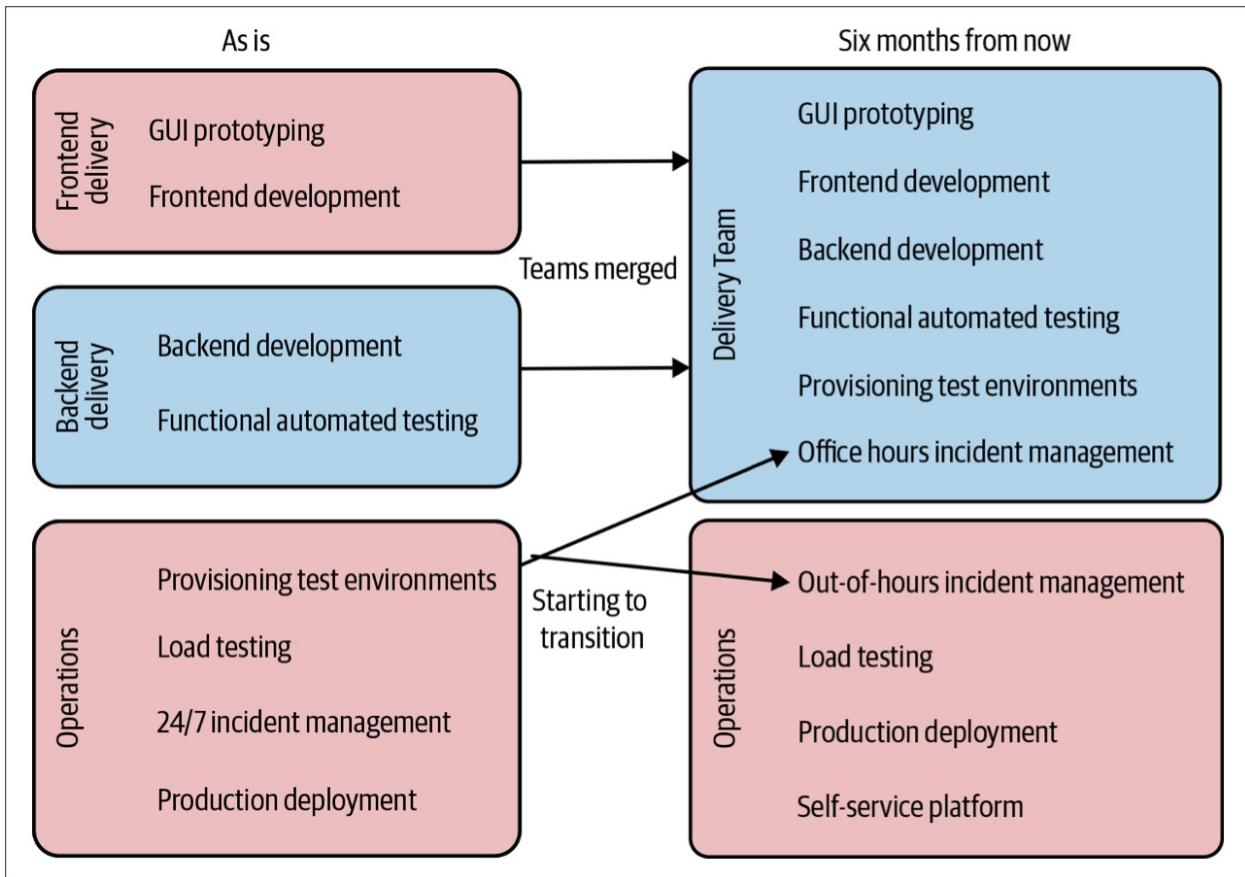


图2-10. 我们想要如何在组织内部重新分配职责的一个例子

在开始进行的变革时，全局视图确实可以提供帮助，但是还需要花时间与团队成员一起确定这些变革是否可行，如果可行，如何进行变革。通过将变革划分为特定职责，我们还可以采用增量方式来实现变革。对于我们而言，首先要集中精力消除由运维团队提供测试环境的需求，这是正确的第一步。

# 改变技能

在评估团队成员需要的技能并帮助他们缩小差距时，我非常喜欢让人们进行自我评估，并以此对团队可能需要什么支持来进行变革有更广泛的了解。

一个具体的例子是：我在ThoughtWorks工作期间参与的一个项目。我们受雇来帮助《英国卫报》重建其在线版的风格（[下一章将会介绍](#)）。为此，《英国卫报》的在线版团队需要掌握新的编程语言及相关技术。

在项目开始时，我们的联合团队提出了一系列核心技能，这些技能对于《英国卫报》的开发人员的工作至关重要。然后，每个开发人员都根据这些标准进行自评，并以1-5分制给自己打分：1意味着一无所知，5意味着可以为此写一本书。每个开发人员的分数都是私密的，且仅与指导他们的人共享。我们的目标不是要求每个开发人员都应该将每个技能提高到5分。我们的目标更多的是：开发人员自己为将要达到的分数设定目标。

作为教练，我的职责是确保：如果我所指导的开发人员想要提高Oracle技能，我将确保他们有机会能够提高其Orcale技能。这可能涉及：

- 确保他们可以在使用该技术的需求开发中工作
- 为他们推荐学习视频
- 考虑让他们参加培训课程或会议
- .....

可以使用自评来建立一个可视化的技能雷达图用以显示：人们可能希望将自己的时间和精力专注在哪一技能区域。如[图2-11](#)所示，该例说明：我确实想将自己的时间和精力集中在提高Kubernetes技术水平和Lambda技术水

平，这也许表明我现在必须自己管理软件发布的事实。在图中突出我们目前对自己的哪方面的技能感到满意同样重要。[图2-11](#)的自评显示，Go编码不是现在需要关注的技能。

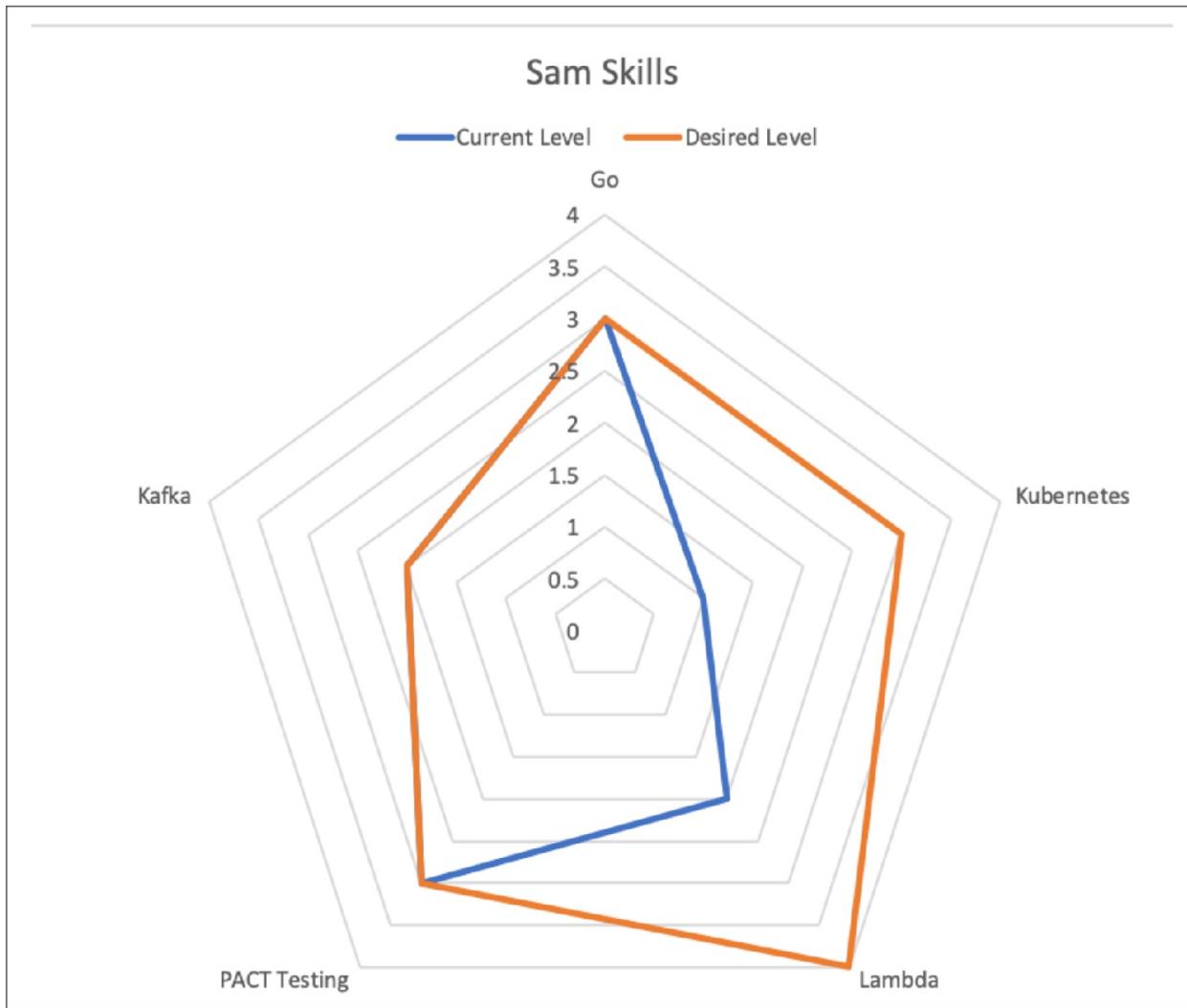


图2-11. 一个技能雷达图的例子，用以显示我希望提升哪些技能

保持这种自评结果的隐私性非常重要。关键不是要别人对自己进行评分，关键是为了帮助员工指导自己的发展。公开这种内容将大大改变自评的结果。例如，突然间，人们会担心自己的分数太低以至于可能会影响到绩效考核。

尽管每个人的打分是私密的，但是仍然可以用其来建立整个团队的自评图。获得匿名的自我评估等级，并为整个团队制定技能图。团队的技能图可以帮助突出显示那些需要在系统层面上解决的技能提升需求。如图2-12所示，虽然我可能对自己的PACT<sup>译注2</sup>技能比较满意，但团队整体却希望在该领域进一步提高，同时Kafka和Kubernetes是另外的需要重点关注的领域。团队技能图会突出表明对小组学习的需求，或表明可能需要进行更大的投资，例如举办内部培训课程。与团队共享总体情况还可以帮助个人了解他们为帮助整个团队找到所需平衡而做出的贡献。

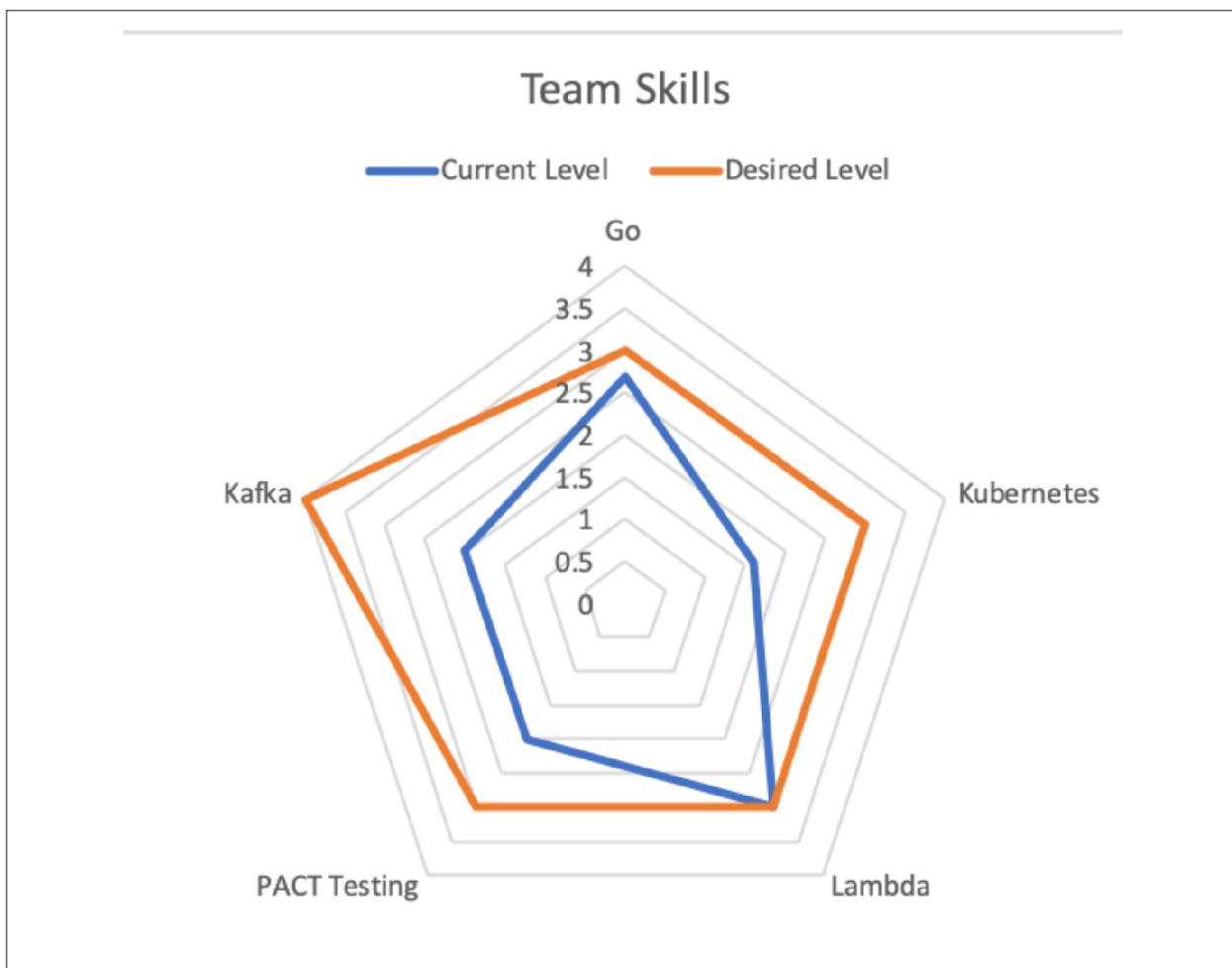


图2-12. 从团队整体情况看，需要提升Kafka, Kubernetes以及PACT Testing  
译注2 技能

当然，改变现有团队成员的技能并不是唯一的方法。我们通常针对的是可以承担更多职责的一支交付团队。因此，这不一定意味着每个人都在做更多的事情。正确的方法可能是让拥有所需技能的新员工加入团队。我们可以聘请Kafka专家加入团队，而不是帮助开发人员学习更多有关Kafka的知识。引入新成员的方式可以解决团队的短期问题，然后团队将拥有一个团队内专家，该专家还可以帮助其他人在该领域学习到更多技能。

关于改变技能的主题，可以有更多内容去探讨，但我希望我已经分享了足够的内容以帮助大家入门。首先要了解我们自己的员工和文化，以及用户的需求。一定要从其他公司的案例研究中得到启发，但是如果大胆复制他人的方案对我们不起作用时，也不要感到惊讶。

---

<sup>9</sup>. Manuel Pais and Matthew Skelton, Team Topologies (IT Revolution Press, 2019). ↪

<sup>10</sup>. Gene Kim, Jez Humble, and Patrick Debois, The DevOps Handbook (IT Revolution Press, 2016). ↪

译注<sup>1</sup>. tier 1: 是技术支持的等级，具体可以参考support levels。tier 1 is the basic help desk resolution and service desk delivery. It's support for basic customer issues such as solving usage problems and fulfilling service desk requests that need IT involvement. If no solution is available, tier 1 personnel escalate incidents to a higher tier. 因为需要及时响应用户的问题，因此，要求技术支持人员要随身携带传呼机或移动设备，以备在非工作时间也能及时响应用户的问题。 ↪

译注<sup>2</sup>. **PACT**: 一个开源契约测试框架，最早是由澳洲最大的房地产信息提供商REA Group的开发者及咨询师们共同创造。REA Group的开发团队很早便在项目中使用了微服务架构，并在团队中对于敏捷和测试

的重要性早已形成共识，因此设计了Pact框架。Pact工具于2013年开始开源，发展到今天已然形成了一个小的生态圈，包括各种语言（Ruby/Java/.NET/JavaScript/Go/Scala/Groovy...）下的Pact实现，契约文件共享工具Pact Broker等。Pact的用户已经遍及包括RedHat、IBM、Accenture等在内的若干知名公司，Pact已经是事实上的契约测试方面的业界标准。 ↵

Copyrights © wangwei all right reserved

# 如何知道迁移过程是否有效

我们都会犯错。即使我们全心全意地开始微服务之旅，也必须接受如下的事情：

- 我们无法了解微服务所涉及到的所有内容
- 在某个时候，我们可能还会意识到，事情可能会失败

问题是：我们是否知道微服务是否有效？我们是否知道我们是否犯了错误？

根据我们希望实现的目标，我们应该尝试定义一些指标，并利用这些指标来跟踪并帮助我们回答如上的这些问题。我们将会简单探讨一些示例性的指标。但我想借此机会强调一个事实：此处，我们不仅仅只是在谈论定量指标。我们还需要考虑来自一线员工的定性反馈。

定量指标和定性指标可以为正在进行的上述评估工作提供信息。我们需要建立检查点，以便团队有时间思考自己是否朝着正确的方向前进。在这些检查点期间，不应该只是问自己“这有用吗？”，还应该问问自己“我们是否应该尝试其他方法呢？”

让我们来看一下如何组织这些检查点活动，以及一些我们可以跟踪的指标。

# 定期检查

无论是什么迁移，都必须在交付过程中留有一些时间以停下来反思，从而方便分析可用信息并确定是否需要更改方案。对于小型团队来说，这种反思可能是非正式的，或者可能是定期进行回顾。对于较大的工作计划，需要把类似的反思设定为定期的、明确的活动，例如：可以把交付活动的相关领导组织在一起，每月进行一次会议，以回顾事情的进展。

无论以怎样的频率停下来反思，也无论采用的形式有多正式（或非正式），我都建议反思要确保涵盖以下内容：

1. 重新陈述我们期望迁移到微服务的目标。如果公司业务方向变了，那么我们的前进方向将不再有意义，那就停下来吧！
2. 查看所有的现有量化指标，以确认我们是否取得进展。
3. 寻求定性反馈——人们是否认为迁移仍然是有进展的。
4. 决定今后要进行哪些更改。

# 定量指标

所选择的、用于跟踪进度的指标取决于要实现的目标。例如，如果目标是缩短产品的上市时间，则度量cycle time<sup>译注1</sup>、部署次数和故障率则很有意义。如果目标是尝试扩展应用程序以处理更多负载，那么报告最新的性能测试将是明智的。

老话说的好：考核什么，就能得到什么。因此，值得注意的是，指标可能很危险。度量标准可能会被无意或有意地进行计算。我想起了妻子告诉我的一家公司：她在这家公司上班，这家公司根据外部供应商关闭的服务单数量来度量供应商的服务，并根据这些结果给供应商付款。这会是什么呢？即使问题没有解决，供应商也将关闭问题，取而代之的是让人们重新提交一个新的问题单。

有的指标可能很难在短时间内有量的变化。如果在微服务迁移的前几个月就看到cycle time<sup>译注1</sup>的大幅改善，我会感到惊讶。实际上，我更希望看到cycle time<sup>译注1</sup>会在迁移的初期变得更糟。因为团队需要不断适应新的工作方式，因此短期内对工作方式的变革通常会对生产效率产生负面影响。这就是为什么采取较小的增量变革如此重要的另一个原因：变化越小，潜在的负面影响就越小，并且一旦发生负面影响，可以更快地解决。

# 定性指标

... Software is made of feelings (软件是由情感构成的)

—Astrid Atkinson (@shinynew\_oz)

无论定量指标的数据结果如何，软件还都是由人来构建的。把人的反馈引入到成功的度量中非常中要。

- 他们喜欢这个过程吗?
- 他们感到有授权了吗?
- 亦或是他们感到不知所措?
- 他们是否获得了承担新职责或掌握新技能所需的支持?

在向上级汇报类似的迁移的任何类型的定量指标打分时<sup>11</sup>，应该包括对团队氛围的检查。如果团队成员喜欢我们正在进行的变革，那就太好了！如果不是，那么我们可能需要做一些事情。忽略员工提供给我们的内容，而完全依靠定量指标会使我们陷入麻烦。

# 避免沉默成本谬误

需要关注**沉没成本谬误**。增加检查过程的目的是确保我们对变化保持客观的了解，同时希望该检查过程可以帮助我们避免陷入沉没成本的谬误。人们投入很多精力来用以前的方法做某事，即使有证据表明该方法已经行不通，人们仍然会保持沉默。此时，就产生了沉默成本谬误。有的时我们会为自己辩护：“随时会发生改变！”有的时候，我们可能会在组织内部投入大量政治资本以进行变革，以至于我们现在无法退缩。无论哪种方式，都是在论证——沉没成本谬误都与情感投资有关——我们沉迷于一种古老思维方式，以至于我们不能放弃它。

以我的经验而言，赌注越大，赢面就越大，而出问题时也越难退出。沉没成本谬误也被称为协和式谬误<sup>译注2</sup>。协和式谬误的名称取自一个失败的项目，该项目是英国和法国政府斥巨资建造的超音速客机（Concorde：协和飞机）。尽管所有证据表明该项目将永远无法带来财务收益，但越来越多的资金被注入到该项目。不论协和飞机在工程上取得了什么成功，它却从未成为可用的商业客运飞机。

如果每一步都很小，那么避免沉没成本谬误的陷阱就会变得容易。此时，改变方向更加容易。使用本节讨论的检查点机制来反映正在发生的事情。我们无需在出现问题的第一迹象时就退出或改变路线。但是无视我们正在收集的、关于引入的变革是否成功的（或其他方面的）证据，比一开始就不收集任何证据更愚蠢。

# 接受新的方法

拆解一个单体应用会涉及到很多变量，并且需要采取多种不同的途径和方法。讨论至此，我希望大家已经不会感到惊讶。可以肯定的是，并非所有事情都会顺利进行。对恢复所做的更改而言，我们需要持有开放的心态。可以尝试新的方法；或者有时仅让变更持续一段时间，以便我们可以了解变更所产生的影响。

如果我们尝试拥抱不断进取的文化，总是想尝试新的东西，那么在需要时改变方向就变得更加自然。如果您将变更或流程改进的概念弱化为离散的工作，而不是将其纳入我们所做的所有工作中，那么我们就有可能将其视为一次性的交易活动。完成这次工作就可以了！如若如此，这也意味着没有给我们带来更多的变化！这种思维会致使我们过几年就会发现自己已落后于所有竞争对手，同时在追赶的道路上还面临一座大山。

---

<sup>11</sup>. Yes, this has happened. It's not all fun and games and Kubernetes.... ↵

译注<sup>1</sup>. 精益生产中的Cycle Time。生产线中的工序处理完成一个订单或上游制工序成品所需要的时间，就是Cycle Time。生产线中的每道工序都有自己独立的Cycle Time，Cycle Time的长短取决于对应工序的生产能力。 ↵

译注<sup>2</sup>. 协和式谬误（concorde fallacy）是指在一种计划上花费了时间和能量，就要继续付出努力，直至完成。其中，付出的代价甚至超过得到的利益。 ↵

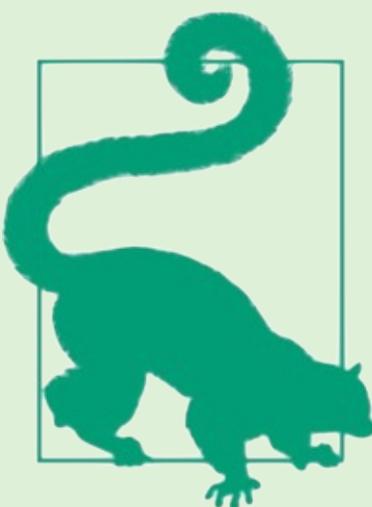
在第2章中，我们探讨了如何思考迁移微服务架构这件事。具体而言，我们探讨了微服务是否是个好主意，还探讨了如果微服务是一个好主意的话，我们应该如落地新架构并确保我们朝着正确的方向前进。

我们讨论了一个良好的服务看起来是什么样子的，还讨论了为什么较小的服务可能对我们更好。但是，可能已经存在大量不遵循这些模式的应用了，我们如何处理这一事实呢？我们如何在不需要进行大规模重写的情况下拆解这些单体应用呢？

在本章的其余部分，我们将探讨各种迁移模式和技巧，以便为采用微服务架构而提供帮助。我们将研究那些适用于如下场景的模式：

- 供应商提供的黑盒软件
- 遗留系统
- 计划继续维护并发展的单体应用

但是，要使增量部署能够正常工作，我们必须确保我们可以继续开发并使用现有的单体软件。



记住，我们希望增量迁移。我们希望逐步迁移到微服务架构，从而使我们能够从迁移过程中得到学习，并在需要时改变主意。



# 改变单体应用，还是不改变？

在迁移过程中，首先要考虑的是：我们是否计划（或能够）修改现有的单体。

如果有能力修改现有的系统，就可以为使用各种模式来修改系统提供最大的灵活性。但是，在某些情况下，这会是一个严格的约束条件，从而导致我们没有这种机会来修改现有的系统。现有系统可能是没有源代码的供应商产品，也可能是用我们不再具备对应能力的技术来实现的。

同时，也许还有一些较软的驱动因素让我们不愿修改现有的系统。当前的单体架构可能处于非常糟糕的状态，以至于变更成本太高了。因此，我们想减少损失并重新开始（尽管我在第二章详述了迁移需要考虑的事情，我仍担心人们会轻易得出这个结论）。不愿修改当前单体的另一种可能是：还有很多其他的人也会开发该单体系统，我们担心对单体的修改会影响他们的工作。某些模式——例如稍后将探讨的抽象分支模式 (*Branch by Abstraction pattern*) ——可以缓解这些问题，但我们仍然会给出如下结论：对单体的修改会给其他人带来太大的影响。

源代码丢失也是一种令人难忘的场景。我曾经与一些同事一起来帮忙扩展一个计算量繁重的系统。该系统的底层计算由我们提供的一个 `c lib库` 执行。我们的工作是集合各种输入，将它们传递到该库，从该库取回计算结果并存储结果。这个库本身千疮百孔。内存泄漏和极其低效的API设计只是导致问题的两个主要原因。我们索要该库的源代码数月之久，以便我们可以修复这些问题。但是，我们被拒绝了。

多年以后，我遇见了那个项目的发起人（sponsor）。于是我问他：为什么不让我们修改底层库。直到那时，他才终于承认是他们把源代码弄丢了。这太难堪了，因此他不想告诉我们！不要让这种情况发生在我们身上。

因此，希望我们处于如下的场景：可以使用并修改现有单体系统的代码库。但是，如果我们不具备这个条件，这是否意味着我们陷入困境之中了？完全相反，此时，很多模式可以帮助我们。我们很快就会简要介绍其中的部分模式。

# 剪切，复制，还是重写

一张取自网络的斗图



即使可以访问单体的现有代码，但是当我们开始把功能迁移到新的微服务时，如何处理现有的代码也并不总是一目了然。我们应该按原样移动代码，还是重新实现功能？

如果现有的单体代码已经被充分拆分了，则可以通过移动代码来节省大量时间。这里的关键是要了解我们想从单体中复制代码，并且至少在现阶段，我们不想从单体中删除此功能。为什么要这么做呢？因为让该功能在

单体中保留一段时间会为我们提供更多选择。这可以为我们提供一个回滚点，或者为我们提供并行运行两种实现版本的机会。接下来，一旦我们对迁移感到满意，我们就可以从单体中删除该功能。

# 重构单体

我观察到，通常，应用单体的现有代码到新的微服务的最大的障碍是：现有的代码库没有围绕业务领域概念来组织。技术分类更为突出（例如，想一下我们所看到的所有Model, View, Controller包名）。当我们尝试移动业务领域的功能时，这可能会很困难：现有的代码库与该业务分类不匹配，因此，即使找到要移动的代码都可能会是问题！

如果确实沿着业务领域边界重新组织了现有的单体架构，那么我强烈推荐Michael Feathers的《修改代码的艺术》一书。在该书中，Michael定义了接缝（seam）的概念：接缝是指程序中的特殊的点，在这些点上，无需做任何修改就可以改变程序的行为。本质而言，可以围绕要修改的代码段定义接缝，对该接缝进行新的实现，并在变更完成后用新的实现替换旧有的实现。Michael把使用接缝来安全的工作的技术作为帮助清理代码库的一种方式。

虽然Michael的接缝的概念通常可以应用于许多范围，但接缝的概念确实与**界定的上下文**非常吻合。因此，尽管《修改代码的艺术》没有直接引用领域驱动设计的概念，但是我们也可以使用该书中的技术，并按照书中的这些原则组织我们的代码。

## 模块化的单体？

一旦我们开始了解现有的代码，显然，接下来值得考虑的事情是：采用新识别出的接缝，并开始把接缝抽取为单独的模块，从而使单体成为模块化的单体。我们仍然只有一个部署单元，但是该部署单元由多个静态链接的

模块组成。这些模块的确切形式取决于我们的底层技术栈——对于Java而言，模块化单体将由多个JAR文件组成；对于Ruby应用而言，模块化的单体则可能是Ruby gems的集合。

正如我们在本书开始时简要提到的那样，将单体拆分为可以独立开发的模块可以带来很多好处，并且可以避免微服务架构的许多挑战。模块化的单体还是许多组织的最佳选择。我已经与多个团队进行了交谈，他们已经开始将其单体拆分为模块化的单体，以最终转向微服务架构，然而却发现模块化的单体已经解决了大多数的问题！

## 增量重写

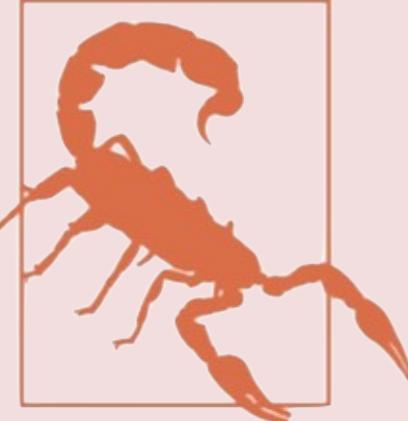
我总是倾向于先尝试挽救现有的代码，然后再诉诸于重新实现功能。我在上一本书[Building Microservices](#)中给出的建议也遵循了这个思路。有时，团队会发现他们从这项工作中获得了足够的利益，并认识到他们最开始时就不需要微服务！

然而，我必须接受的是，实际上，我发现很少有团队采用重构单体的方法来迁移到微服务。取而代之、似乎更常见的是：一旦团队确定了新创建的微服务的职责，他们便会对该功能进行全新的实现。

但是，如果我们开始重新实现功能，我们是否会面临重蹈大规模重写（*big bang rewrites*）覆辙的危险？迁移的关键是确保我们一次只重写一小部分功能，并定期将重新实现的功能交付给客户。如果重新实现服务行为的工作需要几天或几周，那就没问题了。如果时间表开始看起来有几个月了，那么需要重新检查自己的方法。

# 迁移模式

我已经看到在微服务迁移中使用的许多技术。在本章的其余部分中，我们将探讨这些模式，并研究它们可以应用于哪些方面以及如何实现这些模式。记住，就所有模式而言，他们并非适用于所有的场景。对于每一种模式，我会尝试提供足够的信息，以帮助我们了解它们是否对可以解决我们的问题。



务必确保我们了解每种模式的利弊。这些模式并不是万能方法。

我们将从与单体迁移和整合有关的技术开始研究，并且将主要处理应用程序代码所在的位置。不过，首先，我们将介绍一种最有用、且最常用的技  
术：绞杀者模式。

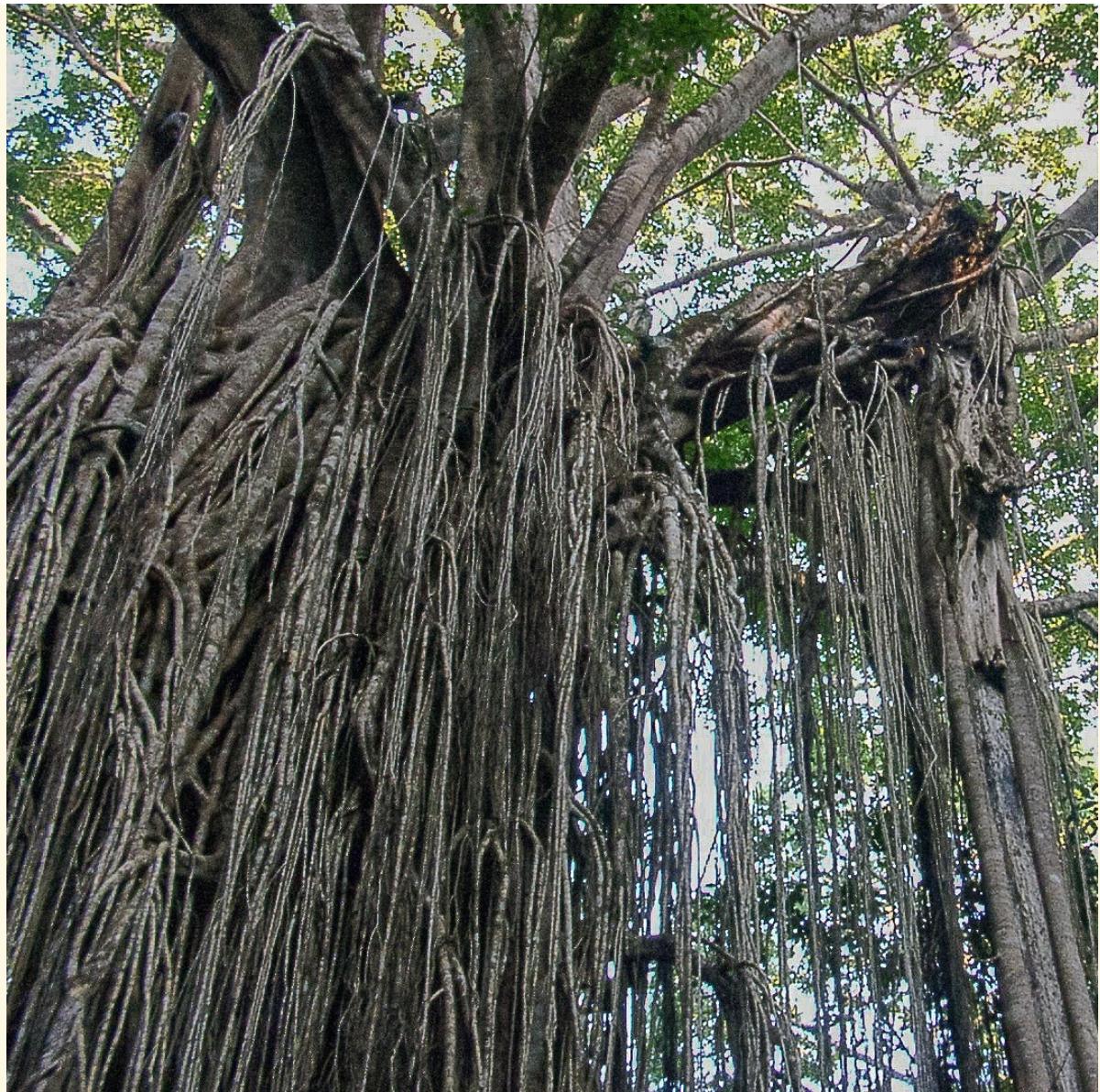
Copyrights © wangwei all right reserved

# 绞杀者模式

绞杀者模式是一种系统重写的常见技术。[Martin Fowler](#)从某种藤类植物汲取灵感，并首次提出了[绞杀者模式](#)。这种藤生长于树上部的树枝，然后朝着地面的方向生长并落地生根，最后逐渐包围原来的树木。这种藤所包围的树为新的藤提供最初的结构支撑，并且如果到最后阶段，我们可能会看到原来的树死了、并且腐烂了，只剩下了新的、现在自支撑的藤。

Martin Flower博客中提到的绞杀藤图片





与此类似，在软件开发中，我们的新系统最初应由现有系统支持和包装。这个主意就是：新、旧系统可以共存，给予新系统成长的时间，并完全替代旧系统。我们将很快看到，这种模式的主要好处是它支持我们增量迁移 to新系统的目标。而且，这种模式使我们在可以利用已经发布的新系统的的同时，具备暂停甚至完全停止迁移的能力。

很快就会看到，当我们在软件中实现绞杀者模式时，我们不仅要努力朝着新的应用程序架构迈进，而且还要确保每一步都易于逆转，从而降低每一步的风险。

# 绞杀者模式是如何工作的

虽然绞杀者模式通常用于从一个单体系统迁移到另一个单体系统，但我们希望利用绞杀者模式将单体系统迁移到一系列微服务。这可能涉及从单体中复制代码（如果可能），或者重新实现所讨论的功能。另外，如果所讨论的功能需要状态的持久化，则需要考虑如何将该状态迁移到新服务，并需要考虑如何再将该状态迁回单体。我们将在第4章中探讨与数据相关的迁移模式。

如图3-1所示，实现绞杀者模式需要三个步骤：

1. 首先，确定现有系统中要迁移的部分。需要使用我们在[第2章](#)中讨论的那种权衡方式，来判断要首先解决系统的哪些部分。
2. 然后，需要在新的微服务中实现此功能。
3. 在准备好新的实现之后，需要能够将请求从单体重新路由到新的微服务。

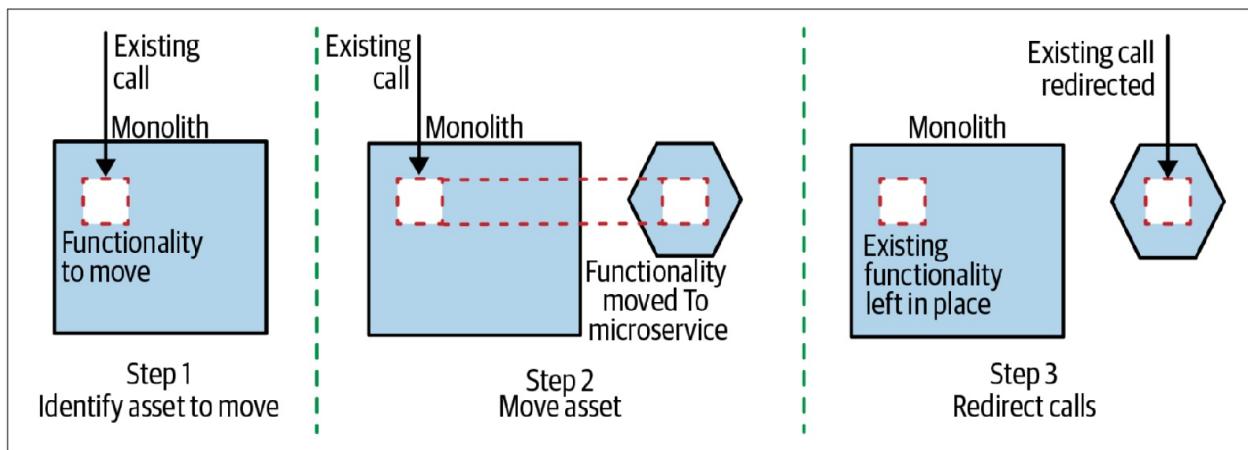
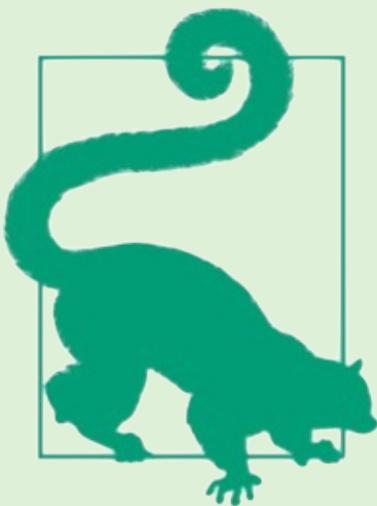


图3-1. 绞杀者模式概览

值得注意的是，在把请求重定向到迁移之后的服务之前，即使已将新功能部署到生产环境中，但从技术角度而言，该新功能仍然没有启用。这意味着，我们可能会花一些时间来获得正确的功能，并在一段时间内实现该功能。我们可以把这些修改推送到生产环境中，并在知道尚未使用新服务的情况下，让我们对修改感到安全，也让我们对新服务的部署和管理感到满意。一旦新服务实现了与单体一样的等效功能，就可以考虑使用类似并行运行（*parallel run*）的模式（稍后将探讨并行运行模式），让我们确信新功能可以按照预期运行。



区分部署（*deployment*）和发布（*release*）的概念非常重要。仅仅把软件部署到给定的环境并不意味着用户实际上正在使用它。将这两件事视为独立的概念，可以在服务使用前在最终的生产环境中验证软件的能力，从而可以降低新软件的风险。类似绞杀者模式、并行运行（*parallel run*）和金丝雀发布（*canary release*）之类的模式就利用了部署和发布是独立活动这一事实。

绞杀者方法的关键点不仅在于我们可以逐步将新功能迁移到新系统，而且在于，如果需要，我们还可以非常轻松地回滚此变更。记住，我们都会犯错，因此我们希望拥有一种技术，使我们不仅可以尽可能低成本地犯错误

（因此会有很多小步骤），而且还可以快速修正错误。

如果单体内部的其他功能也会用到正在抽取的功能，则我们还需要修改这些调用的方式。我们将在本章的后面介绍一些技术用于解决这种场景。

# 何处使用绞杀者模式

绞杀者模式可让我们在无需触碰或修改现有系统的情况下将功能迁移至新的服务架构。当现有的单体还由其他人一起开发时，绞杀者模式可以减少不同人员之间的冲突。当单体程序实际上是一个黑盒系统（例如，第三方软件或SaaS服务）时，绞杀者模式也非常有用。

有时，我们可以一次提取一个完整的、端到端的、功能切片，如图3-2所示。除了对数据的担忧之外，这大大简化了功能抽取的过程。我们将在本书的后面部分介绍数据相关的内容。

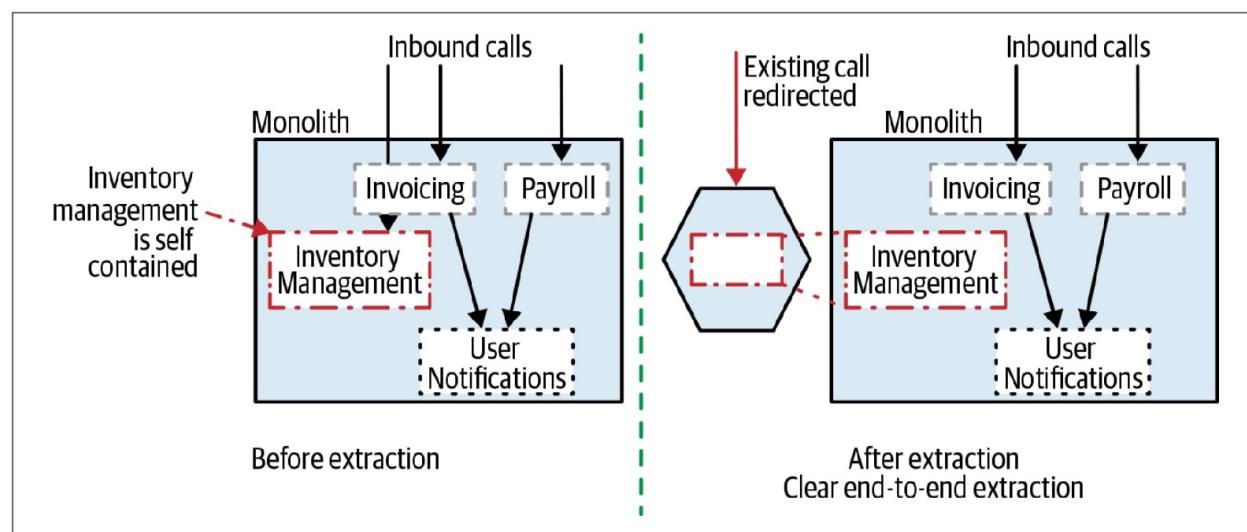


图3-2. 库存管理功能的简单的端对端的抽象

为了执行这种整洁的（*clean*）端到端抽取，我们可能倾向于提取更大的功能组以简化该过程。这可能会导致一个比较棘手的平衡——抽取更大的功能切片，将会承担更多的工作，但会简化某些集成的挑战。

如果我们确实想“蚕食”，我们可能需要考虑更多的“浅”抽取，如图3-3所示。此时，我们会抽取Payroll功能，但是实际上，Payroll功能会依赖仍旧位于单体中的其他功能，例如此例中的发送用户通知的功能。

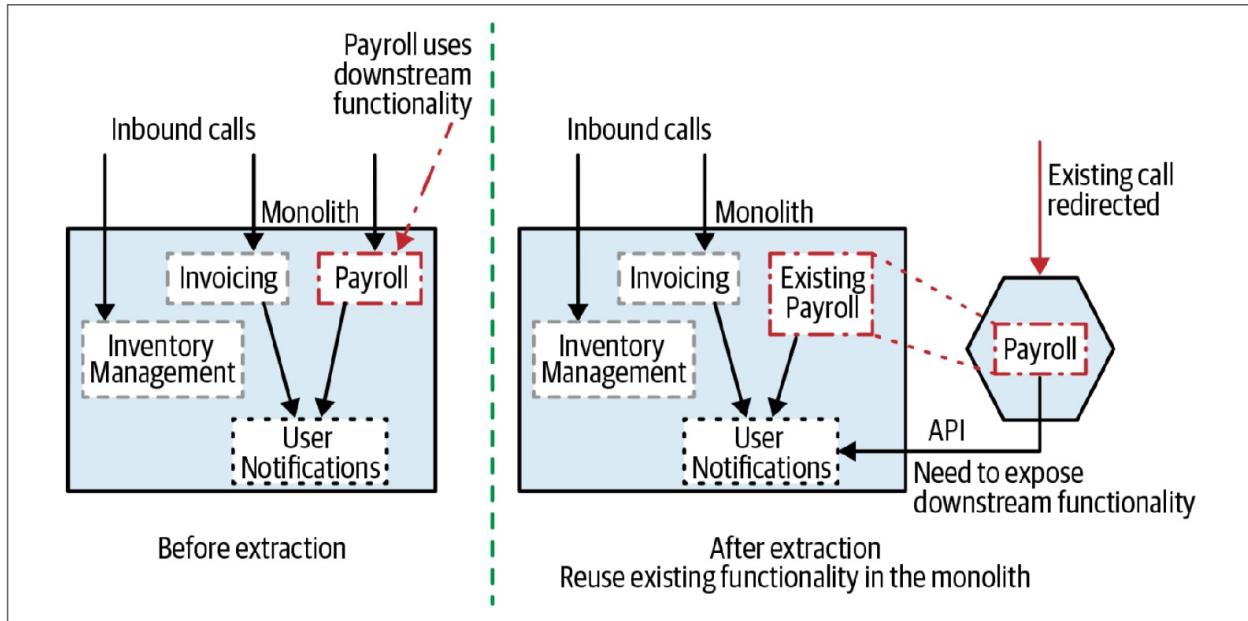


图3-3. 抽取的功能仍然依赖单体

我们没有重新实现User Notifications功能，而是通过在单体中公开该功能，从而为我们的新微服务提供了User Notifications功能——显然，这需要修改单体本身。

但是，为了使绞杀者模式可以发挥作用，我们需要理清我们关注的功能到我们希望迁移的部分的入站调用图。例如，在图3-4中，我们理想的情况是迁移给用户发送User Notifications的功能到一项新服务中。但是，由于该通知由现有单体的多个入站调用而触发，因此，我们无法清晰地重定向来自系统外部的调用。取而代之的是，我们需要研究一种类似于第104页<sup>译注1</sup>的“抽象分支模式”中所述的技术。

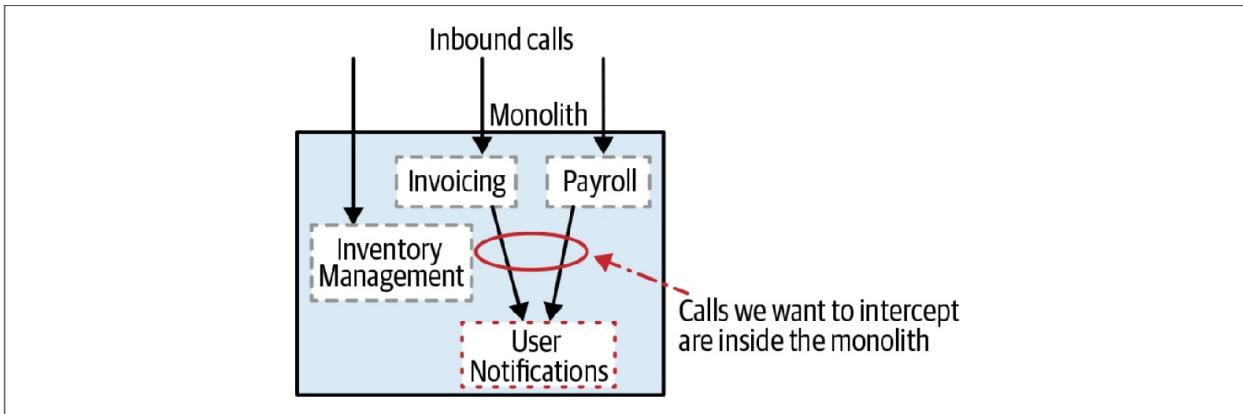


图3-4. 当要迁移的功能位于现有系统的较深的层次时，则绞杀者模式无法发挥作用

我们还需要考虑请求现有系统的请求性质。正如我们稍后将探讨的，类似HTTP之类的协议非常适合重定向。

HTTP本身具有内置的透明重定向的概念，代理可以使用该概念来清晰的了解入站请求的性质并做出对应的转发。其他类型的协议（例如某些RPC）则可能不太适合重定向。我们需要在代理层所做的，用来理解并转发入站调用的功能越多，则此协议的可行性越小。

尽管有这些限制，但绞杀者模式一次又一次地证明了自己是一种非常有用的数据迁移技术。作为一种轻巧易用、且易于处理增量变更的方法，在探索如何迁移系统时，绞杀者模式通常是我的首选方法。

# HTTP反向代理

HTTP具有某些很有趣的能力。HTTP可以轻而易举的以一种对调用者透明的方式拦截并重定向请求。这意味着可以使用绞杀者模式迁移现有的、HTTP接口的单体系统。

在图3-5中，我们看到了对外公开HTTP接口的现有单体系统。该应用程序可能是无头应用（*headless*）[译注2](#)，或者实际上可能是由上游UI调用的HTTP接口。无论哪种方式，我们的目标都是相同的：在上游调用和下游单体之间插入一个HTTP反向代理。

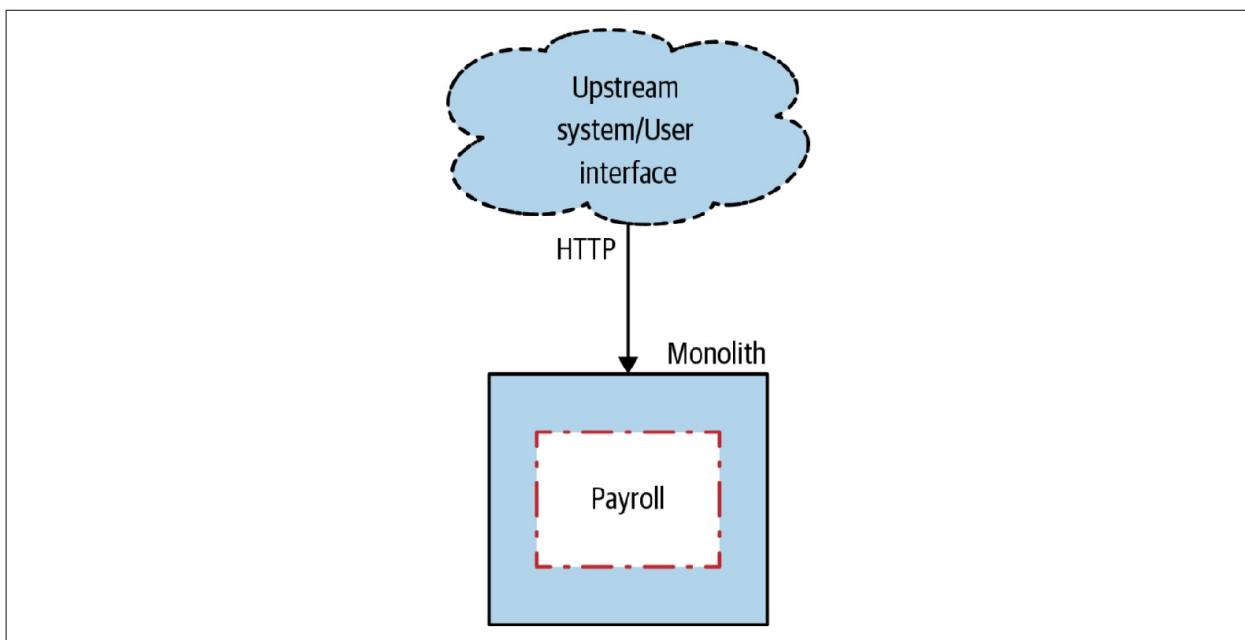


图3-5. 应用绞杀者模式前的以HTTP驱动的单体系统

## 正向代理和反向代理

代理有正向代理和反向代理之分，因此，在继续探讨之前，需要对不同的代理有一个概念上的区分。

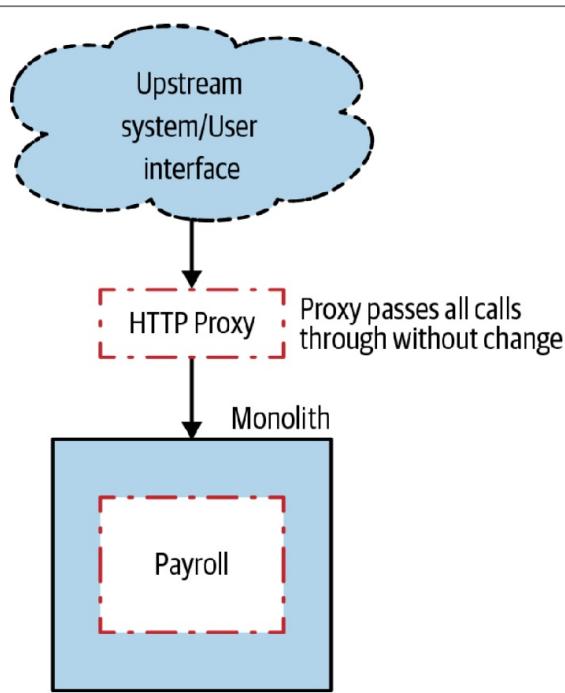
如果从代理的对象而言，我们可以较为容易的对二者进行区分。

如果代理是用来代表用户访问原来用户无法直接访问到的资源，这种情况就属于正向代理。在使用正向代理时，用户一般需要手动进行配置，例如 `export http_proxy=http://127.0.0.1:1087`，用户也会非常明确，此时是使用代理来访问目标资源。

如果，代理对于用户是透明的，代理是用来代理所访问的资源时，这个时候的代理就属于反向代理。例如nginx在做负载均衡时所起到的作用。

## 步骤1：插入反向代理

除非已经有了合适的、可重复使用的HTTP反向代理，否则建议首先让HTTP反向代理就位，如图3-6所示。在第一步中，反向代理无需对请求做任何处理，只是仅仅透传所有的请求。



### 图3-6. 步骤1：在单体和上游系统之间插入反向代理

通过此步骤，我们可以评估在上游调用和下游单体之间增加额外的网络节点的影响。我们需要对新增加的代理组件设置任何必须的监控，并且基本上需要对其观察一段时间。从延迟的角度来看，我们将在所有请求的处理路径中增加一个网络节点和处理进程。我们期望使用良好的代理和网络来使得对延迟的影响最小（也许在几毫秒左右）。但是，如果事实并非如此，则我们有机会停下步伐并在继续前进之前将延迟太大的问题调查清楚。

如果单体前面的代理已经就绪，则可以跳过此步骤。但是，务必确保我们了解如何重新配置此代理以重定向请求。我建议至少要有重定向的经验，以便在假设代理稍后可以工作之前确保代理可以按照预期而工作。如果在打算给新服务发送请求之前，就发现代理无法按照预期而工作，那这真是令人讨厌的惊喜！

### 步骤2：迁移功能

如图3-7所示，一旦我们的HTTP反向代理就绪，接下来，我们就要开始抽取我们的新微服务。

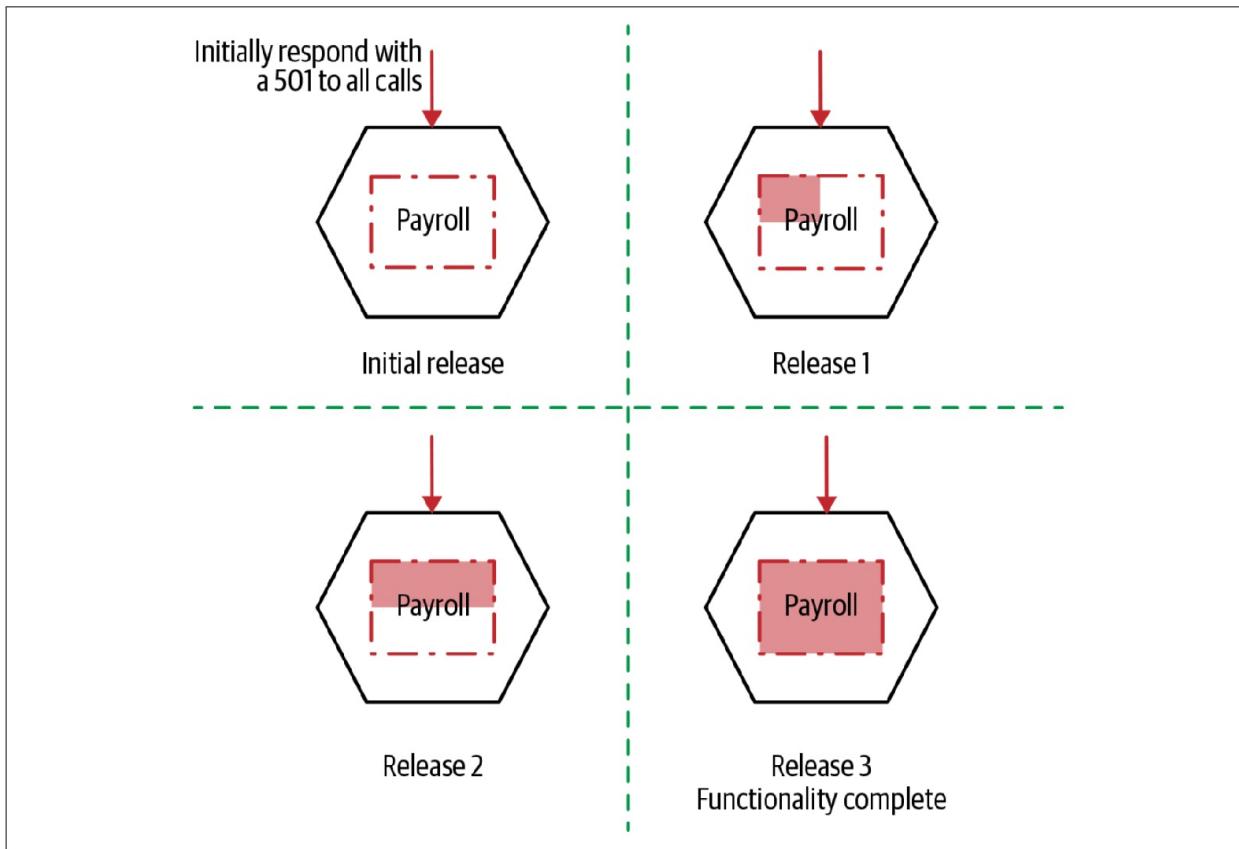


图3-7. 增量实现要迁移的功能

此步骤本身可以分为多个阶段。首先，在没有实现任何功能的情况下，启动并运行基本服务。服务需要能够接收对应功能的调用，但是在此阶段，可以只返回**501 Not Implemented**。即使在这一步，我们也会将此服务部署到生产环境中。这使我们可以熟悉生产部署过程，并可以测试就绪的服务。此时，我们的新服务尚未发布，因为我们尚未重定向现有的上游调用。实际上，我们将软件部署从软件发布中区分开来，这是一种常见的发布技术，稍后我们将再次介绍。

### 步骤3：重定向请求

如图3-8所示，一旦我们完成了需要迁移的所有功能，我们重新配置代理即可实现请求的重定向。如果由于某种原因而失败，那么我们可以将重定向切换回去——对于大多数代理来说，这是一个非常快速和容易的过程，从而可以实现快速回滚。

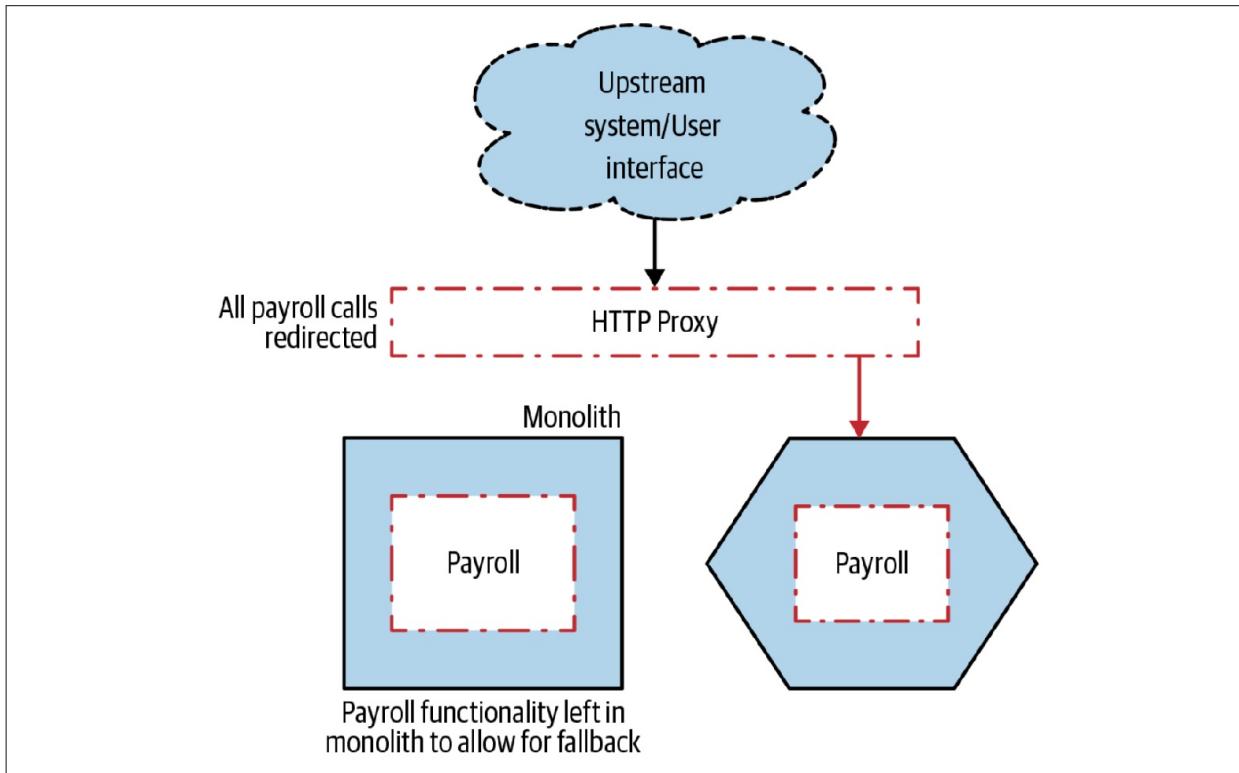


图3-8. 将请求重定向到Payroll功能，然后完成迁移

我们可能决定使用某种类似功能切换之类的方法来实现重定向，这种方式可以使所需的配置状态更加明显。在使用金丝雀发布时，可以考虑使用代理重定向请求的方式来实现增量发布新功能。当然，也可以应用并行运行模式，或者其他本章将会讨论的模式。

# 如何解决数据的迁移?

到目前为止，我们还没有谈论如何迁移数据。在[图3-8](#)中，如果我们新迁移的Payroll服务需要访问单体中的数据库时，我们该怎么办？关于数据的迁移，我们将在第4章中更全面地探讨。

# 代理的选择

代理的实现方式部分取决于单体使用的协议。如果现有的单体使用HTTP，那么会是一个不错的开始。对HTTP如此广泛的支持，以至于对管理重定向而言，我们有太多的选择。我可能会选择像NGINX这样的专用于代理的软件。NGINX是在考虑了这类场景的情况下而创建的，并且可以支持多种重定向的机制。我们可以尝试并测试NGINX提供的不同的重定向机制，并且这些重定向机制看起来表现非常好。

相较于某些重定向，有的重定向更简单。考虑使用**URI path**来重定向，也许就像REST<sup>译注3</sup>资源所展示的那样。在图3-9中，我们将整个Invoice资源迁移到了我们的新服务上，这很容易通过解析**URI path**来实现重定向。

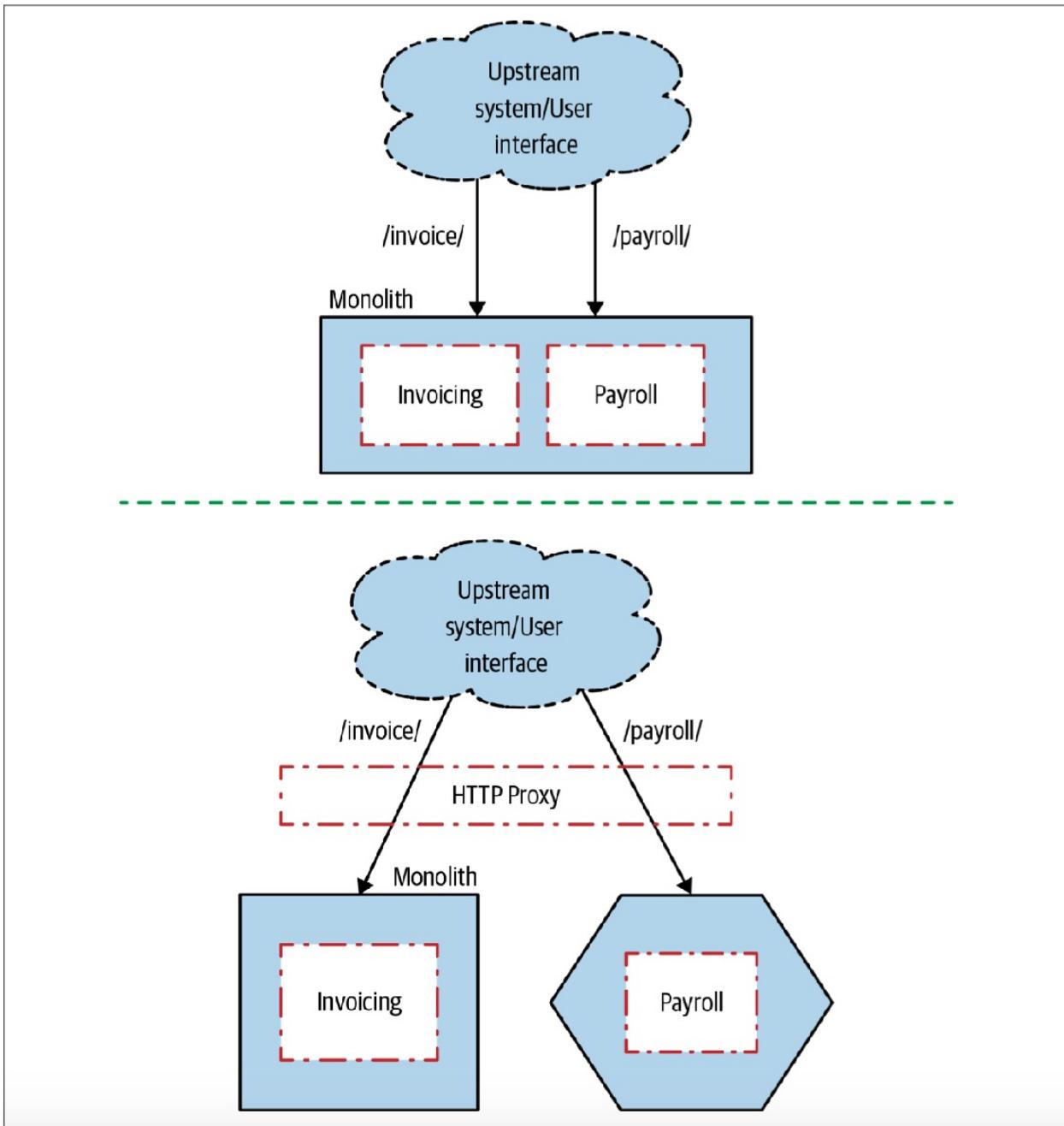


图3-9. 根据资源重定向

但是，如果当前系统把所请求的功能特性的相关信息隐藏于请求体 (*request body*) 中（可能是表单参数），我们需要利用POST参数来控制重定向规则，这种方式有时是可以的，但是会更复杂。如果我们发现我们

遇到了这种情况，此时有必要检查我们可以选择的代理，以确保它们能够处理这种场景。

除非必须，永远都不要用POST参数来路由请求。

如果截获并重定向请求的过程太过复杂，或者单体所使用的协议的支持力度并不广泛，此时，我们可能会动心去自己编写一些代码。但是，对于自己编写代码的这种方法而言，我们要非常谨慎。我自己之前写了几个网络代理（一个用Java实现，另一个用Python实现），但是，这可能只能用来说明我的编码能力。并且在各种场景下，我自己编写的代理的效率都非常低，这大大增加了系统的延迟。如今，如果我需要更多的自定义功能，我可能会考虑将自定义功能添加到一个专用代理中。例如，NGINX允许我们使用Lua来为NGINX增加自定义功能。

## 增量推新

如图3-10所示，反向代理技术允许通过一系列小步骤改变架构，并且，每一个小步骤都可以与系统的其他工作一起完成。

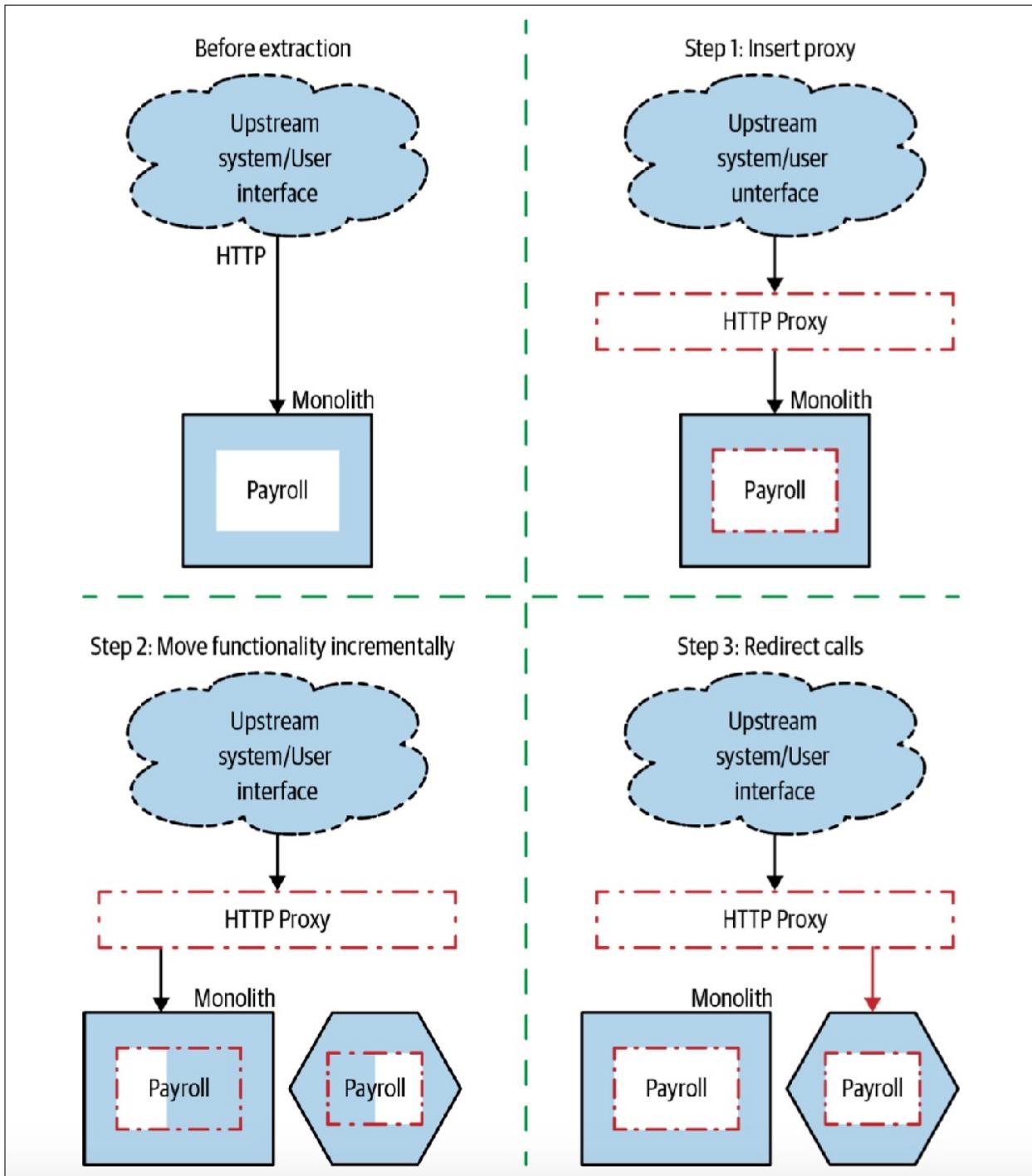


图3-10. 基于HTTP的绞杀者模式的实现

我们可能会认为，完全切换到Payroll新实现的工作量仍然太大。在这种情况下，可以迁移较少的功能。例如，如图3-11所示，可以考虑仅迁移部分Payroll功能并适当的转换请求方式——部分功能在单体中实现，部分功能在微服务中实现。如果单体和微服务中的功能都需要查看同一组数据，则可能会导致问题。此时可能需要共享数据库，并且会带来因共享数据库而导致的所有问题。

没有巨大的变更，在架构升级时，就无需停止升级的进程<sup>译注4</sup>。代理技术可以更容易的将迁移工作拆解为多个阶段，并且还可以与其他交付工作一起交付。与其将工作（*backlog*）拆分为“feature” story 和“technical” story，不如将所有这些工作放在一起。要擅长增量修改技术架构，同时又能提供新功能！

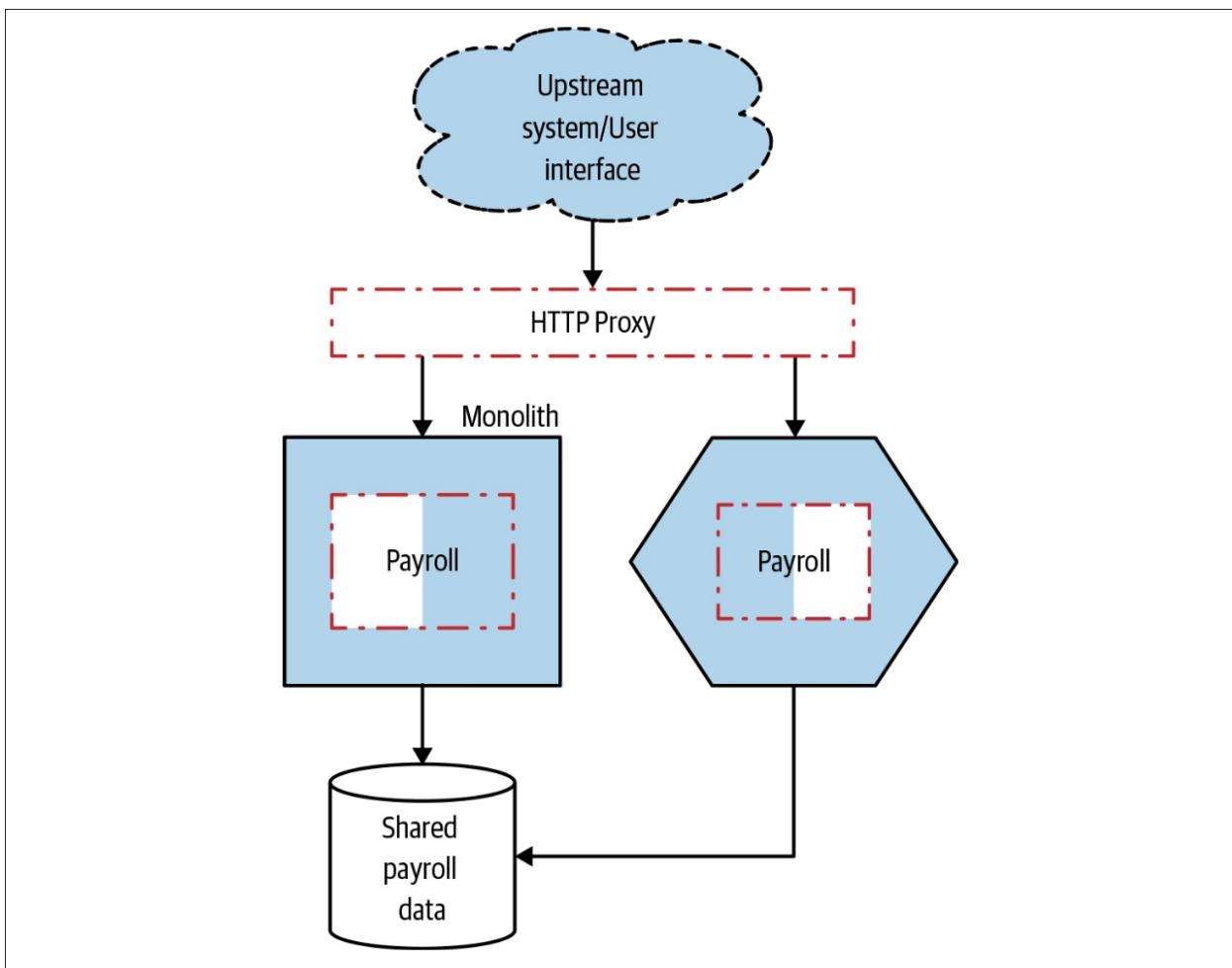


图3-11. 基于HTTP的绞杀者模式的实现

# 改变协议

也可以使用代理来转换请求的协议。例如，当前对外开放的可能是基于HTTP的SOAP接口，但是新的微服务将改为gRPC接口。然后，我们可以配置代理以对请求和响应做相应的转换，如[图3-12](#)所示。

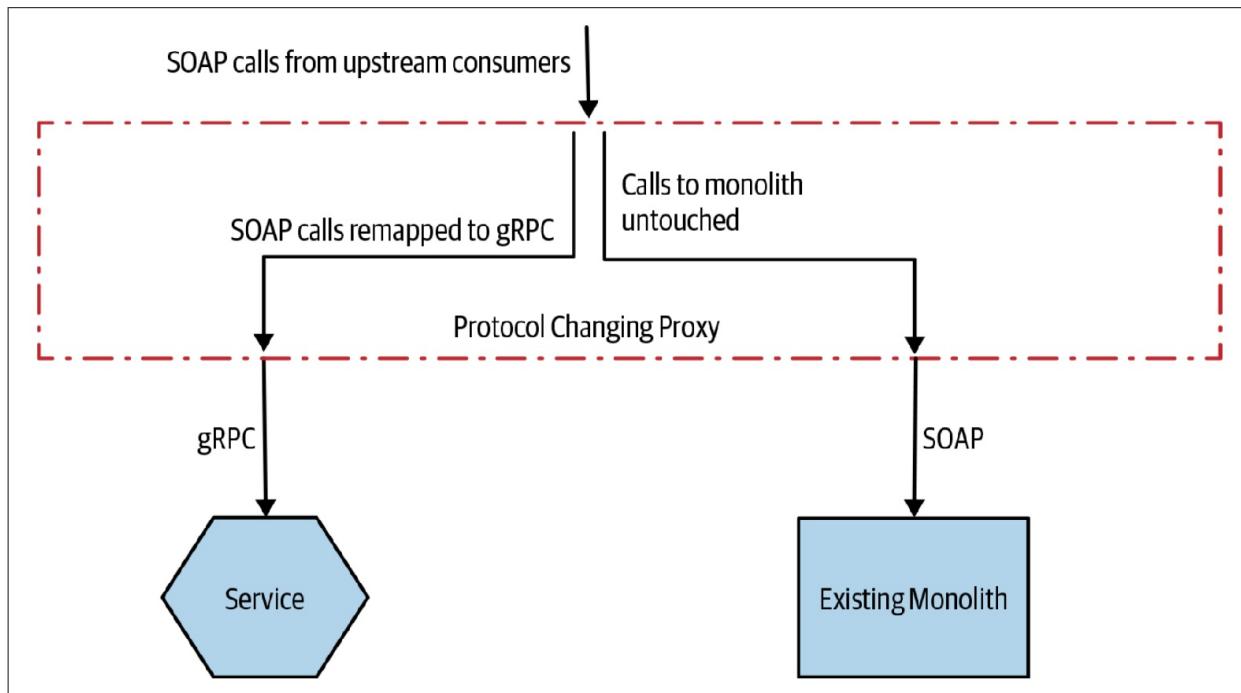


图3-12. 作为绞杀者模式迁移的一部分，可以使用代理来改变通信协议

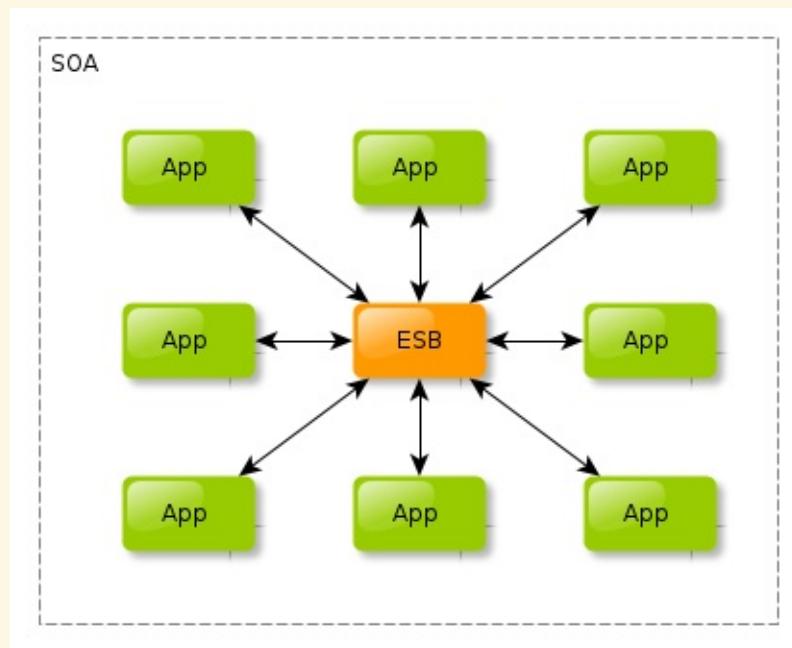
对于这种方法而言，由于代理服务器本身的复杂度和执行逻辑开始不断积累，因此，我确实对此方法有所担忧。对于单个服务，这看起来还不错。但是，如果开始转换多个服务的协议，则在代理中完成的工作会越来越多。我们通常会优化服务的独立部署，但是，如果多个团队都需要编辑一个共享的代理层，则会降低变更速度和部署速度。需要注意的是，我们增加的不只是一个新的冲突来源。当我们讨论微服务架构时，经常会有这样

的口头禅：“**Keep the pipes dumb, the endpoints smart**”<sup>译注5</sup>。我们希望减少推送到共享中间件层的功能，因为将更多的功能推送到共享的中间件层的方式确实会降低功能开发的速度。

### smart endpoints and dumb pipes

[smart endpoints and dumb pipes](#)出自Martin Fowler的介绍微服务的一篇文章，其含义就是：保持通信层的简单性，而让服务更智能。

这是区分微服务和SOA的本质特征。对于SOA而言，需要一个ESB层来整合系统中的服务，从而降低系统的复杂性。而很多的功能和逻辑就要放在ESB层，此时可能就是一种：智能通信，而服务简单的方式了。具体如下图所示：



也就是说，SOA是：smart pipes and dumb endpoints。

Martin Fowler在文章中也没有否认SOA和Microservice的关系。

This common manifestation of SOA has led some microservice advocates to reject the SOA label entirely, although others consider microservices to be one form of SOA, perhaps service orientation done right. Either way, the fact that SOA means such different things means it's valuable to have a term that more crisply defines this architectural style.

——Martin Fowler

其实，Microservice是SOA的传承，但最本质的一个区别就在于微服务的“Smart endpoints and dumb pipes”。“Smart endpoints and dumb pipes”让微服务成为真正的分布式的、去中心化的架构，其本质就是去ESB，把所有的“智能”逻辑——包括路由、消息解析等——都隐藏在服务内部，从而去掉SOA中的那个大一统的ESB。

但是，去掉ESB之后，意味着服务之间的通信会变得更多，因为这意味着服务的拓扑结构由星型网络重新变成了总线型网络。当然，这也是微服务架构区别于传统的分布式系统的一个重要特征：网络交互的规模。但是优点就是能够做到：独立部署。

如果想迁移当前所使用的协议，我宁愿将协议的映射放在服务本身——也就是让服务同时支持新、旧通信协议。如图3-13所示，在服务内部，对旧协议的调用可以在服务内部重新映射为新的通信协议。这避免了对其他服务要使用的代理层的变更管理，并使该服务完全控制此功能如何随时间而变化。我们可以将微服务视为网络端点上的功能集合。我们可能会以不同的方式向不同的消费者开放相同的功能。在服务内部支持不同的消息或请求格式，基本上只是在满足上游消费者的不同需求而已。

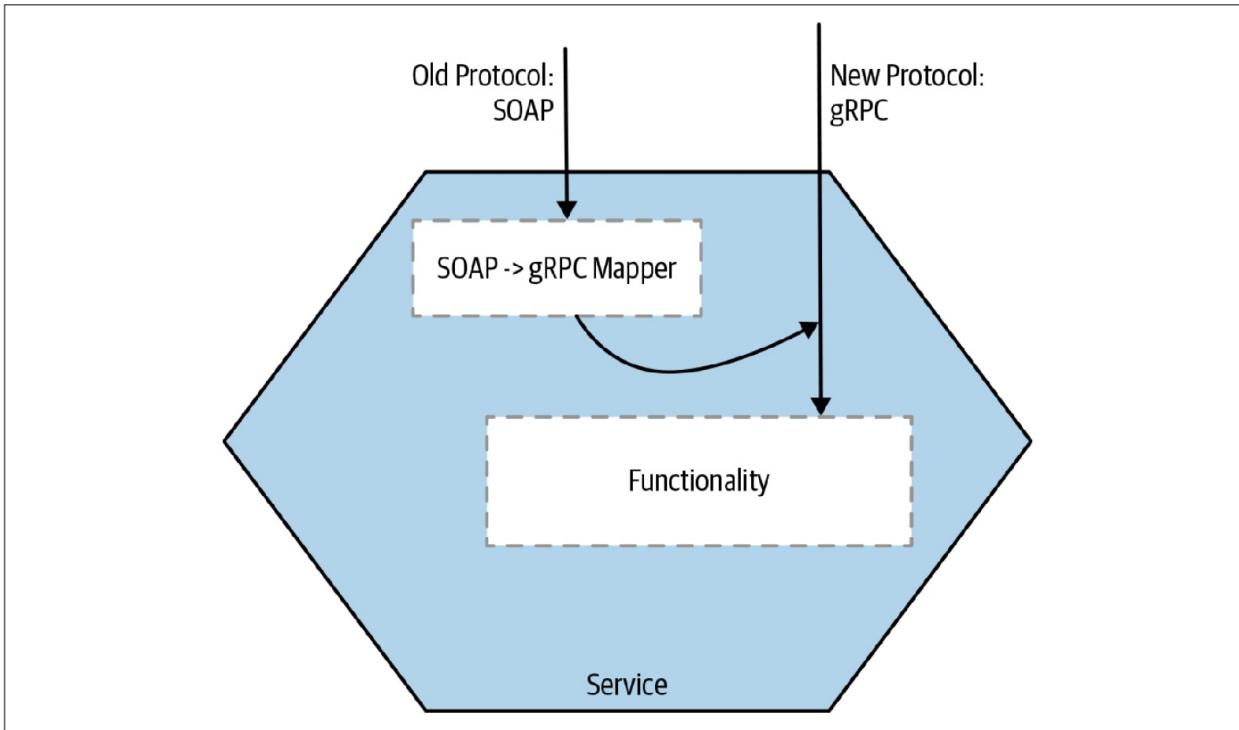


图3-13. 如果要改变协议类型，请考虑让服务通过多种协议开放其功能

通过在服务的内部对指定服务的请求和响应进行映射，可以让代理层保持简单，同时也让代理层更加通用。此外，通过提供同时支持两种通信协议的服务，我们也给自己留有时间以在淘汰旧API之前迁移上游服务。

## service mesh

在[Square公司](#)，人们采用一种混合方法来解决为不同的上游服务提供不同协议的问题<sup>1</sup>。他们决定从自研的RPC迁移到gRPC以实现服务到服务(*service-to-service*)的通信。gRPC是一个有着丰富的生态系统、并有广泛支持的开源RPC框架。为了尽可能降低迁移的痛苦，他们希望减少每个服务的变更量。为此，他们使用了服务网格(*service mesh*)。

如图3-14所示，使用服务网格，每个服务实例都可以通过自己的专用本地代理与其他服务实例通信[译注6](#)。可以为每个代理实例和与其配对的服务实例进行特殊配置。还可以使用控制平面(*control plane*)集中控制并监控这

些代理。由于没有中央代理层（central proxy layer），因此，有效的避免了共享的“smart” pipe的隐患。此时，如果需要，每个服务可以拥有自己的“服务到服务”的通道。值得注意的是，由于Square架构的发展方式，该公司最终不得不使用开源代理Envoy创建自己的服务网格来满足其需求，而无法使用像Linkerd或Istio这样的现有的解决方案。

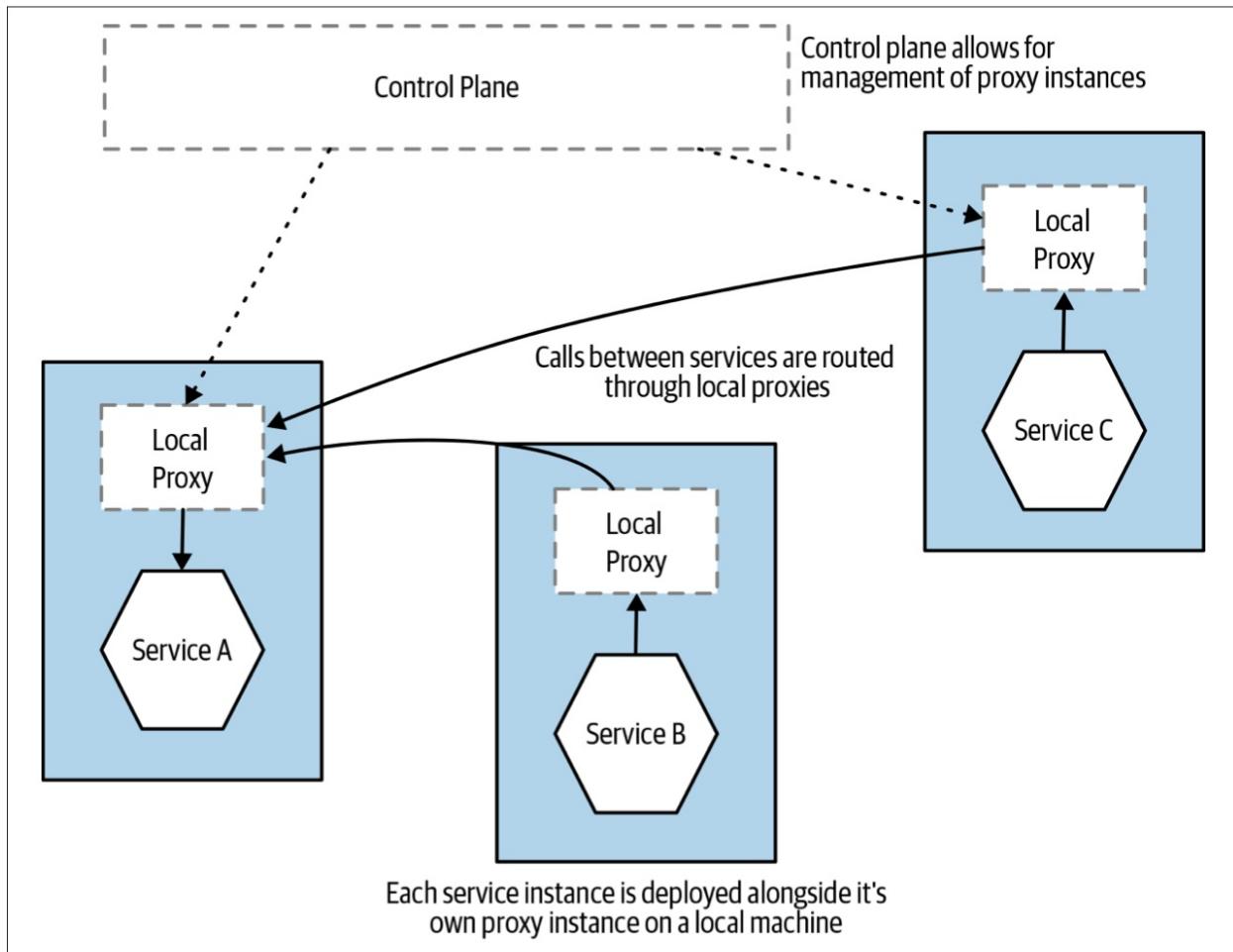


图3-14. service mesh概览

服务网格越来越受欢迎，从概念上讲，我认为服务网格的方法非常正确。服务网格是处理常见的service-to-service通信问题的好方法。我担心的是，尽管一些非常聪明的人做了很多工作，但要花一些时间才能使得服务网格的工具可以稳定的使用。毫无疑问，Istio看起来是服务网格领域的领导

者，但Istio却远非是这个领域的唯一选择，并且似乎每周都会有新的工具出现。我一般建议：在做出选择之前，给服务网格尽可能多的时间来使其稳定。

## FTP协议的例子

尽管我详细的论述了绞杀者模式在基于HTTP的系统中的使用，但没有什么可以阻止我们拦截并重定向其他形式的通信协议。瑞士房地产公司Homegate使用一种绞杀者模式的变体来改变用户上传新房产清单的方式。

Homegate公司的用户通过FTP上传清单，现有的单体系统对上传的文件进行处理。该公司热衷于向微服务架构转型，并且还希望开始支持一种新的上传机制。新的上传机制将不再采用FTP的批量上传方式，而是采用符合即将批准的REST API标准的REST API。

Homegate公司不想让用户感知到变化，而是希望执行无缝修改。这意味着，至少目前，FTP仍是用户与系统交互的机制。最后，如图3-15所示，Homegate公司拦截了FTP上传（通过检测FTP服务器中的日志变化），并将新上传的文件定向到一个适配器，而该适配器则将上传的文件转换为新的REST API请求。

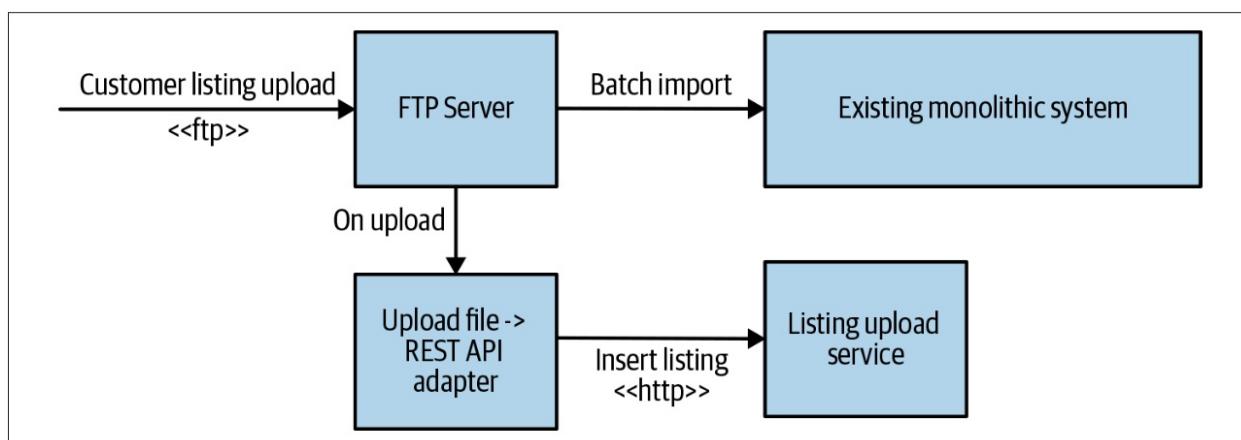


图3-15. 拦截FTP上传，并将其转到Homegate的新的listings服务

从用户的角度来看，上传过程本身并没有改变。其好处来自这样一个事实：处理上传的新服务能够更快地发布新数据，从而帮助用户更快地发布广告。稍后，会计划直接向客户开放新的REST API。有趣的是，在迁移到新服务期间，启用了两种清单上传机制。这使团队可以确保两种上传机制均能正常运行。这是“并行运行模式”的一个很好的例子，我们稍后将在第113页[译注7](#)的“并行运行模式”中探讨。

## 消息拦截的例子

到目前为止，我们已经研究了拦截同步调用，但是如果单体系统是由某种其他形式的协议驱动的呢？单体也许是通过消息代理（*message broker*）接收消息。此时，基本模式是一致的——我们需要一种方法来拦截请求，并将其重定向到我们的新微服务。主要的区别在于协议本身的性质。

## 基于内容的路由

在图3-16中，我们的单体系统会接收大量的消息，我们需要拦截其中的部分消息。

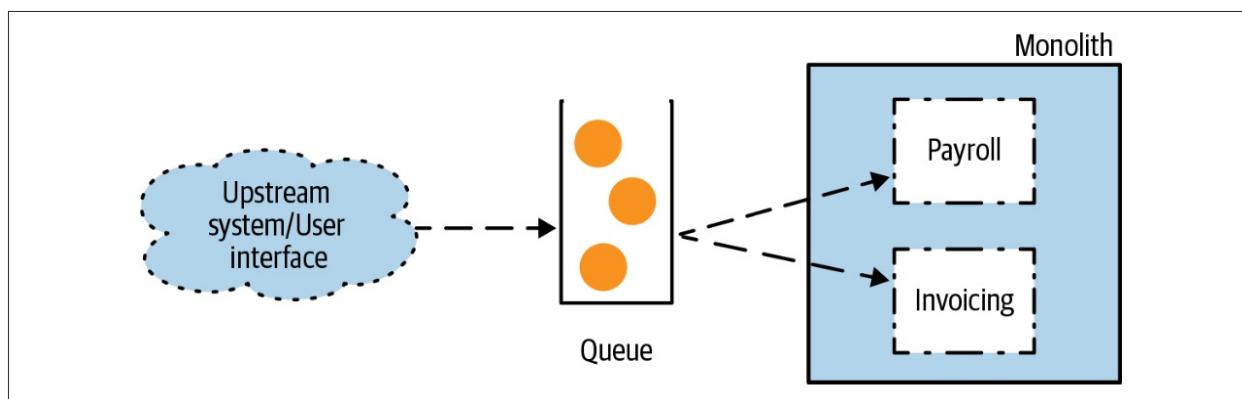


图3-16. 单体通过一个消息队列来接收请求

一种简单的方法是拦截所有将要发送给下游单体的消息，过滤消息以便可以将消息发送到合适的位置，如图3-17所示。这基本上就是基于内容的路由器模式的实现。Enterprise Integration Patterns<sup>2</sup>中描述了基于内容的路由器模式。

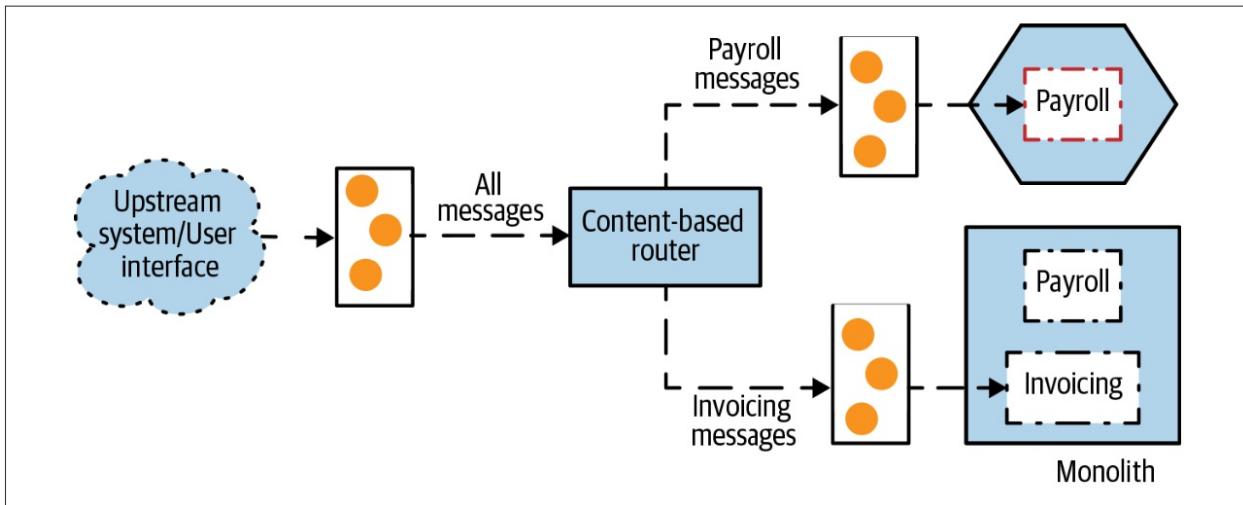


图3-17. 使用基于内容的路由来拦截请求

这种技术使我们能够保持单体不变。但是，我们在请求路径上放置了另一个队列，这可能会增加额外的延迟，这也是我们需要管理的另一件事情。另一个问题是：我们把多少“smarts”置于消息传输层？在*Building Microservices*一书的第4章，我谈到了在服务之间的网络中利用过多的“smarts”所带来的挑战，因为这会让系统更难于理解和更改。

相反，我强烈建议接受“smart endpoints, dumb pipes”的口头禅。“smart endpoints, dumb pipes”是我仍然坚持要做的事情。可以认为，基于内容的路由是我们实现了“smart pipe”，从而增加了如何路由系统之间的请求的复杂性。在某些情况下，基于内容的路由是一种非常有用的技术，但需要找到一个平衡。

## 选择性消费

另一种选择是修改单体，让其忽略那些应该由我们的新服务接收的消息，如图3-18所示。此时，我们的新服务和单体共享同一个队列，并且在服务内部使用某种模式匹配程序来监听他们关心的消息。这种过滤技术是基于消息的系统中的普遍的需求，可以使用类似JMS中的Message Selector之类的技术或其他平台上的等效的技术来实现。

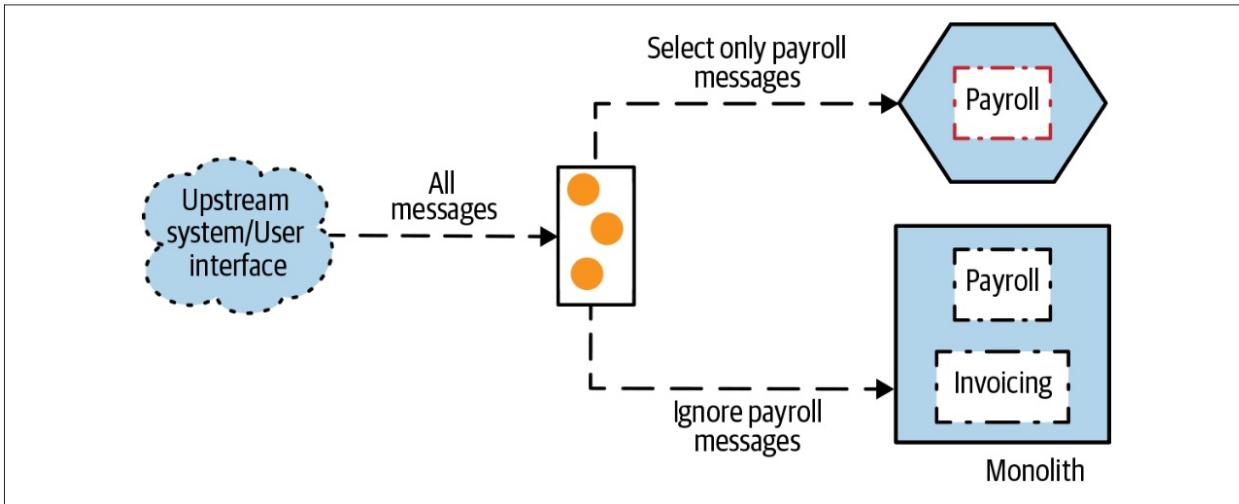


图3-18 使用基于内容的路由来拦截请求

图3-18的过滤方法不需要创建额外的队列，但是会面临很多挑战。

- 首先，底层的消息技术可能支持也可能不支持共享单个队列的消息订阅（共享单个队列的订阅是一个通用的功能，所以如果是这种情况，我会感到惊讶）。
- 其次，当我们要重定向请求时，需要对两边的修改做出非常好地协调才可以。我们需要让单体停止读取新服务需要处理的消息，然后让新服务来处理消息。同样，还原请求拦截也需要执行两次修改以回滚。

同一队列的消费者类型越多，并且过滤规则越复杂，问题就会变得越多。可以想象这样一种情况：规则重叠时，两个消费者会接受到相同的消息；反之，则两个消费者均忽略了某些消息。因此，我可能会考虑仅对少量消

费者或一组简单的过滤规则使用选择性消费的方法。尽管要注意前面提到的潜在缺点，尤其是陷入“smart pipes”的问题，但随着消费者类型的增加，基于内容的路由方法可能更有意义。

选择性消费方案或基于内容的路由方案所增加的复杂度是：如果我们采用异步request-response的通信方式，则需要确保可以将请求再路由回客户端，并希望客户端不会意识到事情有所变化。消息驱动系统中的请求路由还有其他的选择，其中许多选择可以帮助我们实现绞杀者模式的迁移。我在这里推荐一个非常好的资源：Enterprise Integration Patterns<sup>2</sup>。

# 其他的协议

希望可以从这些例子中了解：即使使用不同类型的协议，也有很多方法可以拦截对现有单体的调用。如果我们的单体是由批处理文件上传驱动的，此时该怎么办？拦截批处理文件，提取要拦截的请求，然后将其从文件中删除，最后再转发。确实，有些机制会使的此过程变得更加复杂，并且使用HTTP之类的协议会容易得多。但是，经过一些创造性的思考，绞杀者模式可以用于很多情况。

# 绞杀者模式的其他例子

在希望增量升级现有系统时，无论处于什么情况，绞杀者模式都非常有用。并且，绞杀者模式不仅限于实施微服务架构的团队。Martin Fowler在2004年提出绞杀者模式之前，该模式就已经使用了很长时间。在我的前东家——ThoughtWorks——时，我们经常使用绞杀者模式来帮助重建单体应用程序。Paul Hammant在博客上整理了一份我们使用了绞杀者模式的、不完全的项目清单。<sup>1</sup> 这些项目包括：trading company's blotter，机票预订应用程序，铁路售票系统和分类广告门户网站。

<sup>1</sup>. For a more thorough explanation, see “[The Road to an Envoy Service Mesh](#)” by Snow Pettersen at Square’s developer blog. ↩

<sup>2</sup>. Bobby Woolf and Gregor Hohpe, *Enterprise Integration Patterns* (Addison-Wesley, 2003). ↩

译注<sup>1</sup>. 此处对应的是英文原书的第104页，而不是翻译之后的页数。 ↩

译注<sup>2</sup>. headless为没有UI驱动的应用程序，例如谷歌浏览器的headless模式。 ↩

译注<sup>3</sup>. REST: REpresentational State Transfer，也就是“表述性状态转换”。REST来自于Roy Fielding的[博士论文](#)。基于这篇论文里的理论，衍生出了RESTful API的接口设计风格。 ↩

译注<sup>4</sup>. 原文为：No big bang, stop-the-line re-platforming required. ↩

译注<sup>5</sup>. [smart endpoints and dumb pipes](#)。此处不对该术语进行翻译，因

为保持原汁原味会更好。这个术语出自Martin Fowler的[介绍微服务的一篇文章](#)。其含义就是：保持通信层的简单性，而让服务更智能。这是区分微服务和SOA的最本质的特征。对于SOA而言，需要一个ESB层来整合系统中的服务，从而降低系统的复杂性，而很多的功能和逻辑就要放在ESB层，此时可能就是一种：智能通信，而服务简单的方式了。 ↵

译注<sup>6</sup>. 这种模式也成为垮斗模式。 ↵

译注<sup>7</sup>. 此处为原书的第113页，而不是翻译之后的页码。 ↵

Copyrights © wangwei all right reserved

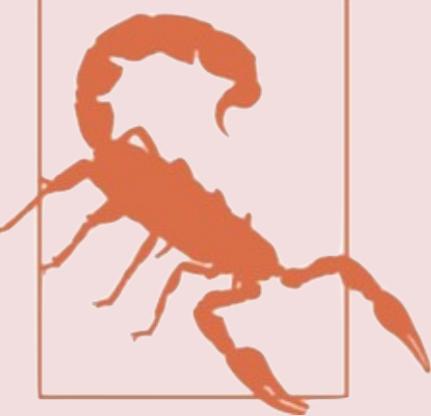
# 在功能迁移过程中对其进行修改

贯穿全书，我将重点介绍我专门选择的模式，因为这些模式可用于把现有系统增量迁移到微服务架构。增量迁移的主要原因之一是，他使我们可以把迁移工作与正在进行的功能交付整合在一起。但是，当我们想要在迁移系统时修改或丰富系统行为时，仍然会存在问题。

例如，想象一下，我们将使用绞杀者模式把现有的Payroll功能从单体中迁移出来。绞杀者模式允许我们将迁移过程划分为多个步骤，并且理论上，我们可以在任何一个阶段回滚迁移操作。如果我们向用户推出了新的Payroll服务，并发现该新服务存在问题，则可以把对Payroll功能的调用迁移回老的系统。如果单体和微服务的Payroll功能是一致的，这种方式是非常好的。但是，如果我们在Payroll的迁移过程中修改了Payroll的功能呢？

如果在新的Payroll微服务上修复了少量的bug，但是没有在单体中实施同样的工作，那么回滚也将导致这些错误再次出现在系统中。如果Payroll的微服务增加了新功能，则会带来更大的问题——回滚意味着用户将无法使用该新增的功能。

没有简单的方法可以解决这种问题。如果想要在迁移完成之前修改功能，那么就必须接受回滚会更加困难这一事实。如果不允许在迁移完成之前进行任何更改，则事情会变得更容易。迁移花费的时间越长，对系统的这一部分实施“功能冻结”（*feature freeze*）就越难——只要有一个需求需要修改系统的一部分，则这种需求就不可能消失。完成迁移所需的时间越长，对于“仅在迁移到该功能时将该功能实现就好”[译注1](#)的观点所承受的压力就越大。每次迁移的规模越小，在迁移完成之前修改待迁移功能的压力就越小。



迁移功能时，尽量消除对待迁移功能的任何更改——尽可能把新功能或 Bug 修复推迟到迁移完成之后。否则，可能会降低系统的可回滚能力。

| 译注<sup>1</sup>. 原文为：just slip this feature in while you're at it. ↪

Copyrights © wangwei all right reserved

# UI组合模式

到目前为止，我们所考虑的技术主要是用于服务端的增量迁移工作，但是前端UI（*user interface*）为我们提供了一些有用的机会可以把来自于现有单体或新的微服务架构中的部分功能整合起来。

多年以前，我参与了一个项目，该项目用来帮助《英国卫报》的在线版将其现有的内容管理系统迁移到新的、基于Java的定制平台。迁移过程将与发布新风格的在线报纸并行进行，新风格的在线版本是配合印刷版的重新发行而开展。我们希望采用增量迁移的方法，我们针对不同的垂类（旅行，新闻，文化）把现有CMS到全新网站的过渡拆分为多个阶段。即使在这些垂类内部，我们也寻找机会将迁移拆分成较小的部分。

最终，我们使用了两种有用的组合技术（*compositional techniques*）。通过与其他公司的交流，我越来越发现，过去几年来，这些技术的变化是许多组织采用微服务架构的重要组成部分。

# 页面组合的例子

在《英国卫报》的迁移项目中，尽管我们推出了单独的widget（稍后将讨论widget），但迁移计划始终还是基于页面的迁移，以便使全新的网站风格得以实现。基于页面的迁移是按照垂类来进行的，Travel垂类是我们迁移的第一个页面。在此过渡时间内，用户在访问网站的新部分时会看到和老版本不同风格的页面。在该过程中，我们还费尽心思来确保所有旧页面的链接都重定向到了新位置（新页面的URL已经发生变化）。

几年后，当《英国卫报》改变其技术以摆脱目前的Java单体架构时，他们再次使用了类似的、一次迁移一个垂类的技术。这次迁移，他们使用更快的CDN来实施新的路由规则，从而可以像使用内部代理一样有效地使用CDN。<sup>3</sup>

作为提供房产在线交易的澳大利亚公司，REA公司拥有不同的团队来分别负责商业地产和住宅地产，并且每个团队都拥有其完整的渠道。在这种情况下，因为团队可以拥有整个端到端的体验，因此基于页面的组合方法是有效的。实际上，REA公司为不同的渠道采用了完全不同的品牌，这意味着基于页面的拆分更加有意义，此时可以为不同的客户群体提供完全不同的体验。

# widget组合的例子

在《英国卫报》的升级中，Travel垂类是第一个要迁移到新平台的页面。之所以这样做，部分原因是因为Travel垂类在分类方面遇到了一些有趣的挑战，同时Travel垂类也并不是该网站中最引人注目的部分。我们希望上线某些迁移，并从迁移中汲取经验，但同时也要确保如果出现问题，不会影响网站的主要部分。

《英国卫报》的旅行部分涵盖了世界各地的、迷人的目的地的深入报到。我们没有采用整体迁移旅行部分的方案，反之，我们希望能发布更多的低调的版本来测试系统。我们部署了一个单独的widget来显示新系统定义的旅行目的地的TOP 10榜单。如图3-19所示，将该widget拼接到旧的旅行页面。在该案例中，我们采用Apache服务器，并使用了一种称为Edge-Side Includes (ESI) [译注1](#)的技术。使用ESI技术，可以在网页中定义模板，然后Web服务器会对此处的内容进行拼接。

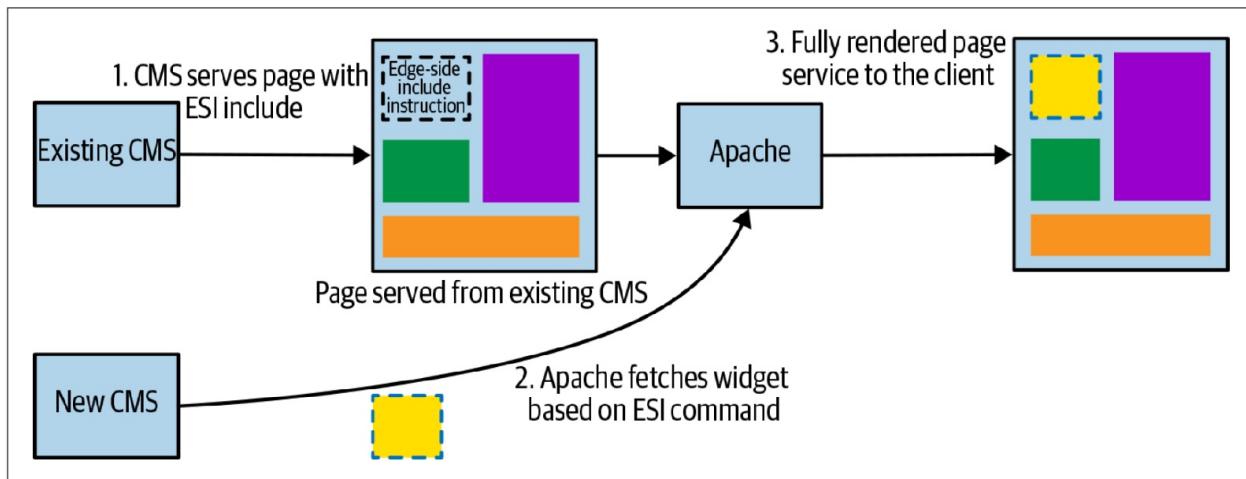
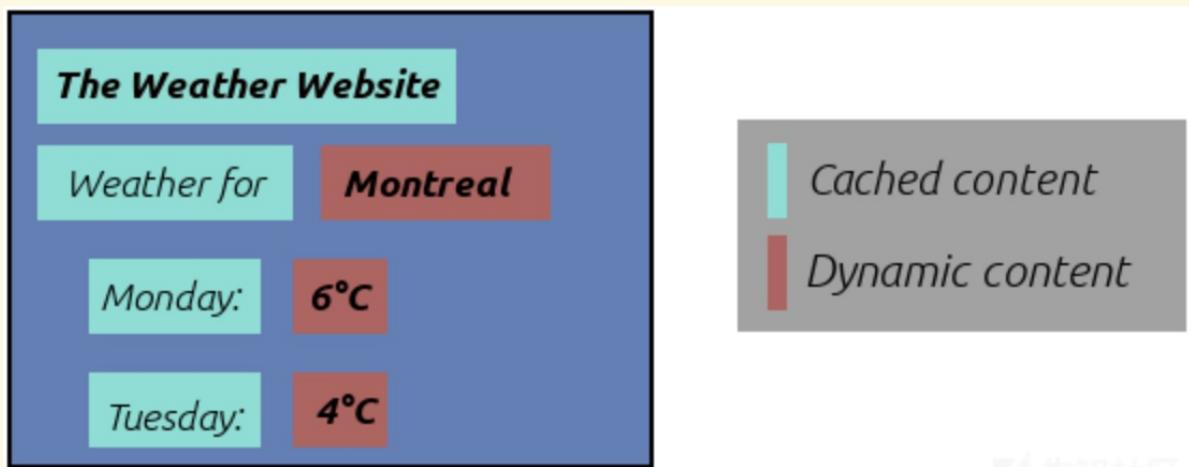


图3-19. 使用ESI技术来拼接来自新的CMS系统的内容

ESI技术

ESI语言是基于XML标签的标记语言，用于改善HTTP中间件加载大量Web内容缓存时造成的性能下降。开发者可以把页面的动态内容替换为ESI标签，从而实现缓存的多样化。当用户请求该页面时，代理服务器会解析处理ESI标签，获取内容，从而有效减轻后端服务器的压力。

举个例子，有一个查询天气的网站，网站会定时缓存城市天气页面的内容，如下图，可以用ESI标签代替获取动态数据的参数。



上面这个例子的HTML可以是如下的样子：

```
<body>
  <b>The Weather Website</b>
  Weather for <esi:include src="/weather/name?
  id=$(QUERY_STRING{city_id})" />
  Monday: <esi:include src="/weather/week/monday?
  id=$(QUERY_STRING{city_id})" />
  Tuesday: <esi:include src="/weather/week/tuesday?
  id=$(QUERY_STRING{city_id})" />
  <!--
  ...
  -->
```

但是，ESI也有自己的问题。例如，HTTP中间件服务器不能正确识别ESI标签是来自上游服务器还是恶意用户，换句话说，攻击者可以注入恶意ESI标签，HTTP中间件会相信标签来自上游服务器，并且盲目解析并且转发。

因此，在使用ESI技术时，在获得ESI带来的便利的同时，也要注意ESI的负面影响。

如今，仅在服务器端拼接widget似乎非常少见了。这主要是因为基于浏览器的技术已经变得更加复杂，从而使得浏览器（或native app）自己就可以完成更多的页面组合。这意味着，对于基于widget的Web UI而言，浏览器经常会使用多种技术来多次调用并加载各种widget。这样做的另一个好处是，如果一个widget无法加载——可能是由于后端服务不可用——但是，其他widget仍然可以展现。此时，仅导致部分服务降级，而非全部服务降级。

对于《英国卫报》而言，尽管最后我们主要使用基于页面的组成，但许多其他的公司却大量使用基于widget的组合技术。例如，Orbitz（现在已被Expedia收购）就创建了专用服务来为单个widget提供服务<sup>4</sup>。在转向微服务之前，Orbitz网站就已经拆分为独立的UI“modules”（Orbitz是这样对其命名的）。这些模块可以表示：搜索表单，预订表单，地图等。如图3-20所示，这些UI模块最初直接由Content Orchestration服务来提供。

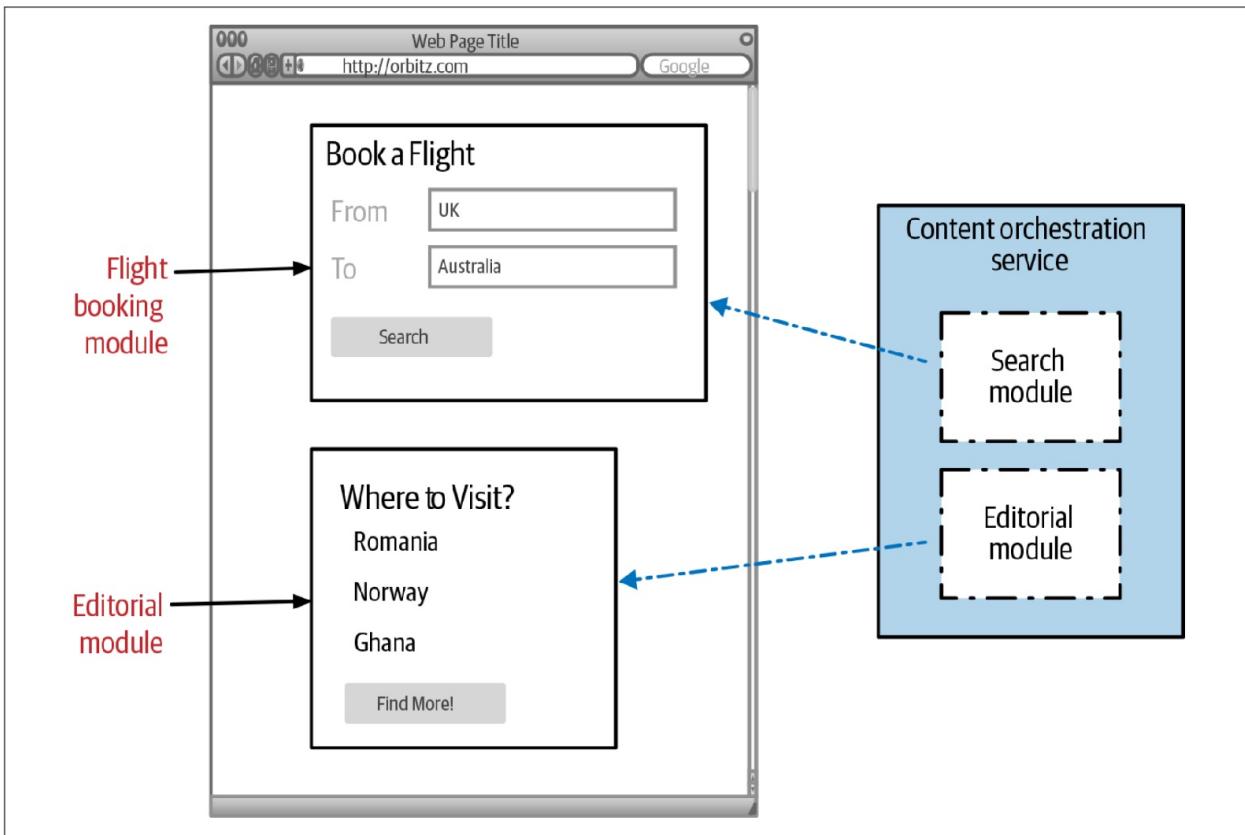


图3-20. 在迁移到微服务之前，Orbitz的Content Orchestration服务为所有的模块提供服务

Content Orchestration服务实际上是一个庞大的单体。所有拥有这些模块的团队都必须协调在单体内部所作的变更，从而导致发布变更存在大量延期。这是我在[第1章](#)中强调的交付冲突问题的经典案例：每当团队必须协调其他团队才能推出变更时，变更成本就会增加。为了实现更快的发布周期，当Orbitz决定尝试微服务时，他们聚焦于根据这些模块来进行服务拆分，并且从editorial模块开始开启迁移之旅。如[图3-21](#)所示，Content Orchestration服务的功能已经变成已迁移到微服务的模块访问其下游服务的代理。

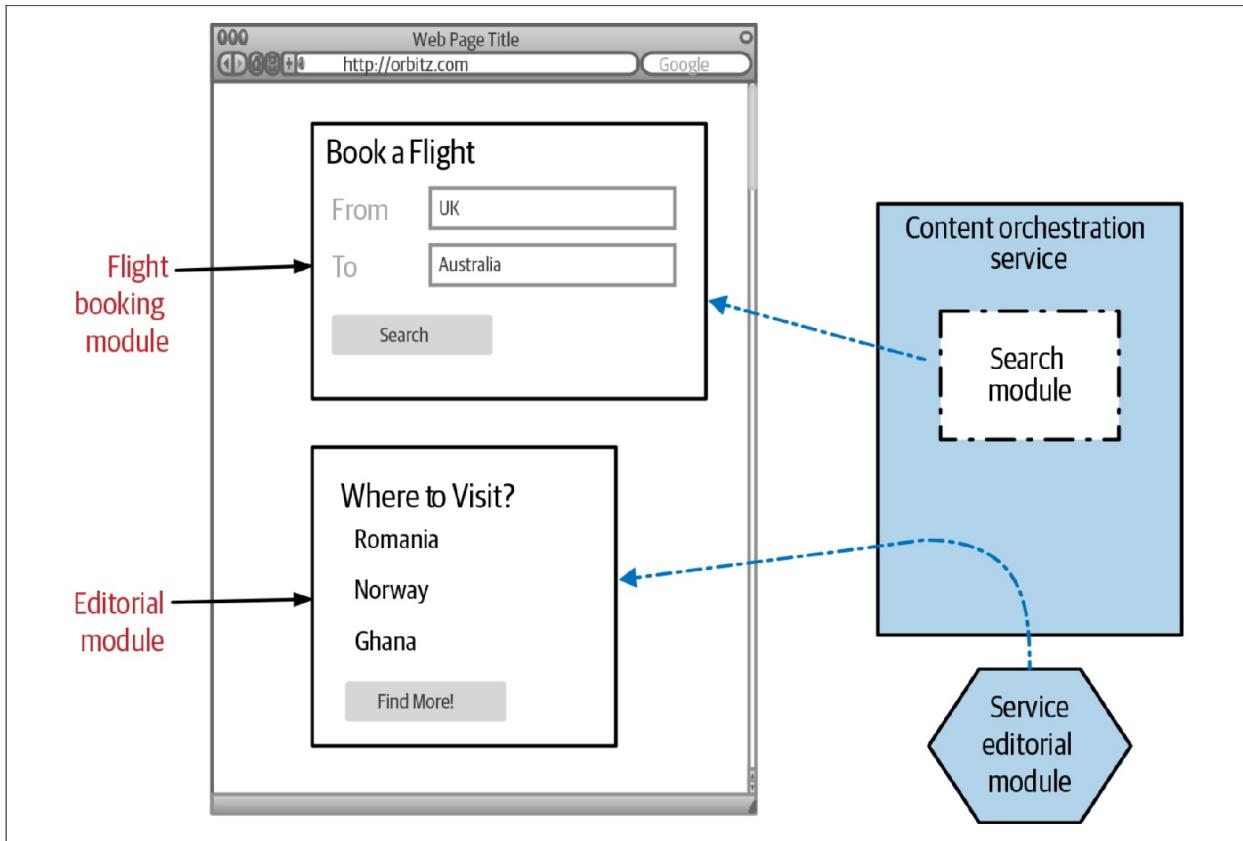


图3-21. 每次迁移一个模块，Content Orchestration服务成为新服务的代理

前端UI已经按照这些模块进行了视觉拆分，这一事实让Orbitz的微服务迁移的工作更容易以增量方式来开展。由于各个独立模块之间已经拥有清晰的owner，因此这进一步帮助了迁移工作，从而使得可以在不干扰其他团队的情况下更轻松地执行迁移。

值得注意的是，并非所有的UI都适合拆分为新的widget，但是在可以的情况下，这会使得增量迁移到微服务的工作变得更加容易。

## 移动应用

尽管我们主要讨论基于Web的UI，但是其中的一些技术也可以很好地适用于手机客户端。例如，Android和iOS都提供了UI组件的功能，从而更容易隔离这些UI组件，或以不同方式重新组合这些UI组件。

部署原生移动应用程序的变更的挑战之一是：Apple App Store和Google Play商店都要求提交应用的新版本并对其审核，只有审核通过之后，用户才能更新到应用的新版本。尽管在过去的几年，应用商店签发应用程序的时间已大大减少，但这仍然增加了部署应用新版本的时间。

以此观之，APP本身也是一个单体：如果要更改APP的某一个单独的部分，则仍然需要部署整个APP。并且还必须考虑以下事实：用户必须下载APP的新版本才能看到新功能——使用基于Web的应用程序时，我们通常无需处理这些事情，因为更改可以无缝地交付给用户的浏览器。

为了解决如上提到的问题，对现有的APP而言，许多公司都允许无需重新部署APP的新版本就能实现动态变更。通过在服务器端部署更改，客户端可以立即看到新功能，而不必部署APP的新版本。尽管有些公司会使用更复杂的技术，但可以使用web view之类的方法轻松实现这一目标。

### APP审核是好还是不好？

对于国内而言，Android应用商店的审核还是非常快的。并且，有些Android应用的驱动是开发者可以自己控制的，这使得Android应用的发布会更快速。

但是iOS应用就没有这么幸运了。对于非越狱的iOS设备，苹果要求，所有的应用程序都必须来源于苹果的App Store。然而，每次提交应用到App Store的时候，整个的审核周期有时候又是非常漫长的。这会导致如果App出现一个只能发版才能解决的问题，那么这个问题的收敛可能就会非常慢。因此，为了应对这种问题，出现了很多热更新技术或插件技术，比如之前的JSPatch技术。不过，后来，苹果也针对这种行为做过严厉的打击。

在我看来，虽然对于开发者而言，审核会增加应用更新的时间，对于开发者是不友好的。但是对于用户而言，审核是整个应用生态的必须手段。因为审核的存在，利用公司的力量来打平了用户的能力，从而实现为用户赋能。并非所有的用户都具备很高的安全意识或者掌握很高深的计算机技术，如果没有审核的存在，对于用户而言，无疑是一场噩梦。2020年，央视的315晚会上就曝光了部分SDK会窃取用户隐私的现象。但是，对于iOS设备而言，这种窃取的现象还是比较少见的。

多年之前，也流行过一句笑话：如果想买苹果的产品，不需要做太多的比较，你就说你想买什么价位的就行。

因此，我向来非常支持苹果的严格的审核制度，并且这种审查制度的规则的公开透明的。另外，从一定程度上，这种审查会促使开发者更加重视应用的质量，因为一旦出现BUG，那么修复起来的周期可能会比较长。

Spotify在所有平台上的UI都是高度面向组件（*component-oriented*）的，包括其iOS和Android应用程序。我们看到的几乎所有内容都是一个单独的组件：从简单的文本标题，到专辑封面甚至是播放列表<sup>5</sup>。这些模块中的某些模块又由一个或多个微服务来支持。在服务器端，以声明的方式来定义这些UI组件的配置和布局。Spotify工程师无需向应用商店提交APP的新版本就能够改变用户所看到的界面并快速回滚这些变更。这使他们可以更快地进行实验并尝试新功能。

# 微前端的例子

随着网络带宽和Web浏览器能力的提升，运行于浏览器中的代码的复杂性也不断提高。现在，许多基于Web的UI都使用某种形式的单页应用程序（SPA: *single page application*）框架，从而消除了由不同网页组成应用程序的概念。此时，我们拥有一个功能更强大的UI，其中所有内容都运行于一个窗口——以前这种浏览器内（*in-browser*）的用户体验只有利用类似Java Swing这样的较重的UI SDK才能实现。

利用单个页面提供整个界面，我们显然不能再考虑基于页面组合的技术了，因此我们必须考虑某种形式的基于widget的组合技术。已经做了很多尝试来为web编写通用widget。最近，[Web Components规范](#)正在尝试定义支持跨浏览器的标准组件模型。但是，经过很长时间之后，Web Components标准才流行起来，其中的一个绊脚石是浏览器的支持。

对于Web Components标准中的shadow dom技术而言，各浏览器的支持情况如下图所示：



不同浏览器对其他技术的支持情况，可以参考站点[Can I Use](#)。

人们并没有坐等Web Components来解决问题，而是使用诸如Vue，Angular或React之类的单页面应用程序框架来解决问题。另外，许多人试图解决如何把SDK构建的UI模块化的问题，其中这些SDK最初设计为拥有整个浏览器窗口。这导致人们朝着某些人所谓的*微前端（Micro Frontends）*的方向发展。

乍一看，Micro Frontends实际上只是将UI拆分为相互独立的不同组件。然而，在这一方面上，Micro Frontends并不是什么新鲜的技术——在我出生之前，面向组件的软件就已经出现好几年了！更有意思的是，人们正在研究如何让Web浏览器，SPA SDK和组件化可以协同工作。如何精确地利用Vue和React创建单个UI，而又在其依赖不发生冲突时，仍然允许它们可以共享信息？

深入讨论微前端超出了本书的范围，部分原因是根据使用的SPA框架的不同，完成此工作的方式也会有所不同。但是，如果发现自己要拆解单页应用程序时，那么我们并不孤单，已经有很多人为此提供了共享技术和库来完成这项工作。

# 何处使用

作为允许系统可以迁移到新技术平台<sup>译注2</sup>的一种技术，UI组合非常有效，因为它允许迁移功能的整个垂直类部分。但是，要使其正常工作，需要具有改变现有UI的能力，以允许我们可以安全的插入新功能。本书稍后将介绍组合技术，但值得注意的是，可以使用哪种技术通常取决于实现UI所采用的技术的特性。一个好的、老式的网站可以让UI组合变得容易，而SPA技术的确增加了一些复杂性，并且还常常存在一系列令人困惑的实现方法！

<sup>3</sup>. It was nice to hear from Graham Tackley at The Guardian that the “new” system I initially helped implement lasted almost 10 years before being entirely replaced with the current architecture. As a reader of the website, I reflected that I never really noticed anything changing during this period! ↪

<sup>4</sup>. See Steve Hoffman and Rick Fast, “Enabling Microservices at Orbitz”, YouTube, August 11, 2016. ↪

<sup>5</sup>. See John Sundell, “Building Component-Driven UIs at Spotify”, published August 25, 2016. ↪

译注1. ESI Language Specification 1.0: <https://www.w3.org/TR/esi-lang/>.  
↪

译注2. re-platforming，译为采用新的技术栈来实现现有技术栈实现的系统。 ↪

# 抽象分支模式

为了让绞杀者模式可以发挥作用，我们需要能够拦截对单体发起的请求调用。但是，如果要抽取的功能位于现有系统中的底层时，我们该怎么办？再次回到前面、如图3-4的例子，想一下提取Notification功能的需求。

为了抽取系统底层的功能，我们需要修改现有系统。这些修改可能是非常大的修改，并且会破坏同时开发代码库的其他开发人员的工作。此时，我们将陷入相互竞争的紧张局势。一方面，我们希望以增量方式进行修改。另一方面，对工作在代码库的其他区域的其他人员而言，我们希望减少对其带来的干扰。这自然会促使我们朝着希望快速完成工作的方向迈进。

通常，当修改现有代码库的某些部分内容时，人们会在一个独立的源代码分支上修改。如此，可以在修改代码的同时而不影响其他开发人员的工作。但是，这种方式面临的挑战是：一旦在分支上完成修改，这些变更就必须合并回之前的分支。合并分支通常会带来严峻的挑战。分支存在的时间越长，合并分支带来的问题就越大。此刻，我不会详细介绍与长期存在的源代码分支相关的问题，只是说它们与CI的原则背道而驰。我会引用来自“[The 2017 State of DevOps Report](#)”中收集的数据来表明：采用基于主干的开发模式（避免使用分支，直接在主干上修改）以及使用短暂的分支会有助于提高IT团队的性能。我不是long-lived分支的粉丝，而且并非我一人如此。

因此，我们希望能够以增量的方式修改代码库，同时还希望把开发代码库其他部分的开发人员的干扰降到最低。此时，我们可以使用另一种模式。这种模式允许我们在不依赖源代码分支的情况下逐步修改单体。与源代码

分支方法不同，抽象分支模式修改现有的代码库，以实现在同一代码版本中安全地共存不同的修改，同时还不会造成过多的干扰。

# 抽象分支模式如何工作

抽象分支模式由5个步骤构成：

1. 为要替换的功能创建一个抽象。
2. 修改使用现有功能的客户端，以使用该功能的新的抽象。
3. 重新实现该抽象，在我们的例子中，新的实现将会调用我们的新的微服务。
4. 在抽象中进行切换，以使用我们的新实现。
5. 清理抽象，并删除旧的实现。

让我们看一下，如何利用如上的步骤迁移如图3-4中所示的Notification功能。

## 第一步：创建抽象

第一个任务是创建一个抽象用来表示需要修改的代码与其调用者之间的交互，如图3-22所示。如果已经对现有的Notification功能做了合理拆分，那么创建一个抽象就像我们在IDE中应用Extract Interface重构一样简单。如果没有，则可能需要抽取一个前面介绍过的[接缝](#)。创建抽象可能会让我们在代码库中搜索那些对发送电子邮件、SMS、或可能存在的其他通知机制的API的调用。找到这些代码，并创建用于其上游模块使用的抽象是必需的步骤。

IDEA中的Extract Interface重构

The screenshot shows the IntelliJ IDEA interface with the project 'TestJava' open. In the left sidebar, the 'Project' view shows a package structure with 'com.wangwei' containing 'TestClass'. The 'TestClass.java' file is open in the main editor. A context menu is displayed over the code, specifically over the declaration of 'method1'. The menu is titled 'Show Context Actions' and includes options like 'Copy Reference', 'Paste', and 'Refactor'. The 'Refactor' submenu is expanded, showing 'Extract Interface...', which is highlighted with a yellow box. Other options in the 'Refactor' submenu include 'Move Class...', 'Copy Class...', 'Type Parameter...', 'Extract Delegate...', 'Extract Superclass...', 'Reveal in Finder', 'Open in Terminal', 'Local History', 'Compare with Clipboard', and 'Create Gist...'. The code in 'TestClass.java' is as follows:

```
package com.wangwei;  
public class TestClass {  
    public static final  
    public void method1()  
    }  
    public void method2()  
}
```

The screenshot shows the IntelliJ IDEA interface with the project 'TestJava' open. In the left sidebar, the 'Project' view shows a package structure with 'com.wangwei' containing 'TestClass' and 'TestInterface'. The 'TestInterface.java' file is open in the main editor. The code is as follows:

```
package com.wangwei;  
public interface TestInterface {  
    void method1();  
    void method2();  
}
```

```
package com.wangwei;
/**
 * @author wangwei
 */
public class TestClass implements TestInterface {
    public static final double PI = 3.14;

    @Override
    public void method1() {
    }

    @Override
    public void method2() {
    }
}
```

## 第二步：使用抽象

创建抽象之后，我们现在需要重构使用Notification功能的现有客户端，以使用新的抽象，如图3-23所示。IDE中的Extract Interface重构有可能为我们自动完成了，但是以我的经验，这通常是一个递增的过程，并涉及到手动跟踪对相关功能的入站调用。令人高兴的是，这些变更很小并且是逐步进行的，很容易一步一步来完成，而又不会对现有的代码产生太大影响。此时，系统行为不应有功能上的变化。

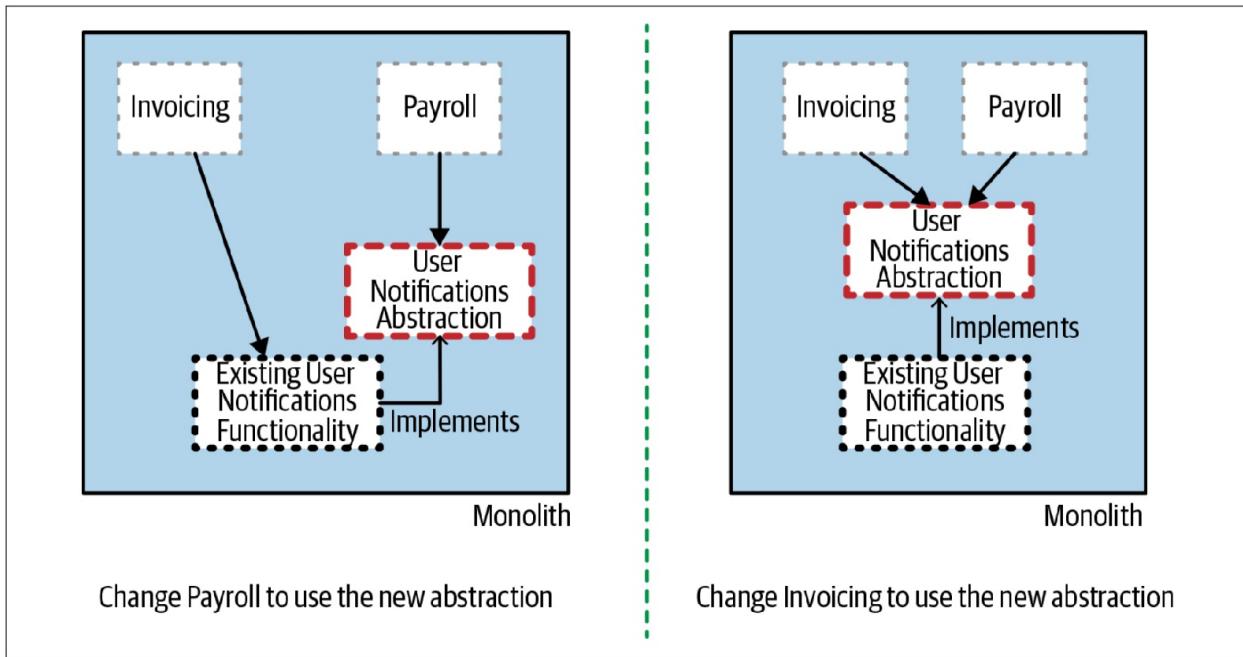


图3-23. 修改现有的客户端以使用新的抽象

### 第三步：创建抽象的新实现

新的抽象已经发挥作用，我们现在就可以开始实现新服务了。如[图3-24](#)所示，在单体内部所实现的Notification的功能只是用来调用外部服务的客户端而已，Notification的大部分功能都位于外部的服务。

此时，关键是要认识到，对于创建的抽象而言，尽管我们在代码库中同时有两个实现，但是当前系统中只有一个实现处于活动状态。直到我们认为我们的新服务可以提供服务时，我们的新服务实际上一直处于休眠状态。当我们努力在新服务中实现所有的等效功能时，抽象的新实现可能会返回Not Implemented错误。当然，这并不会阻止我们为已经编写的功能编写测试用例，并且这是尽早集成这项工作的好处之一。

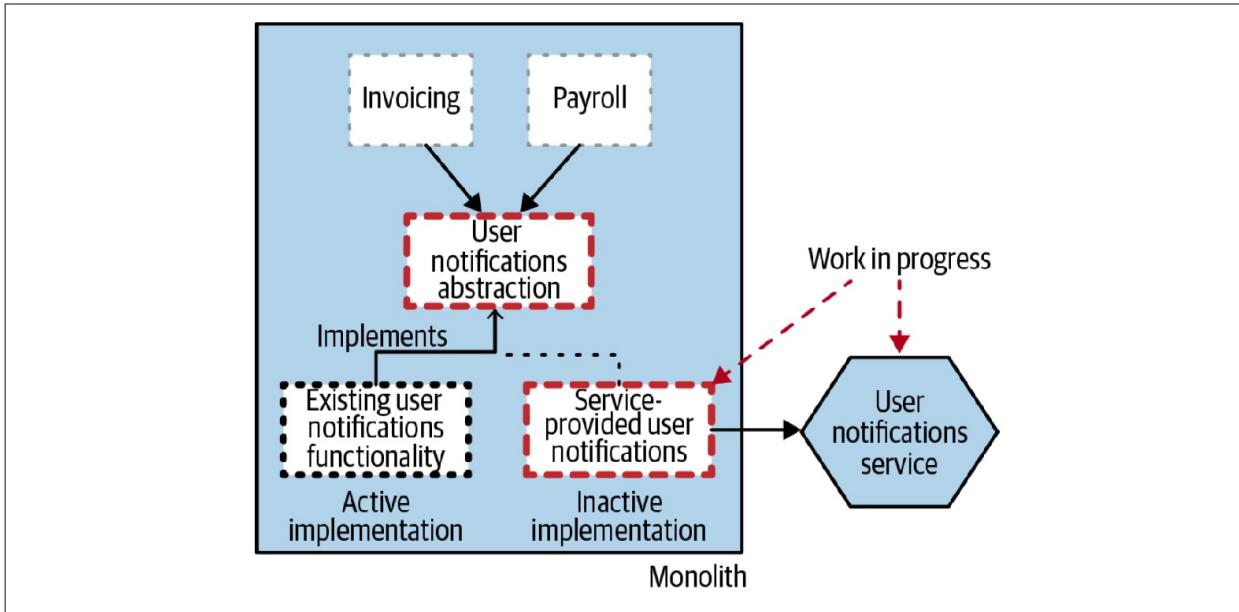


图3-24. 创建抽象的新实现

在此过程中，我们也可以像使用绞杀者模式那样，把开发中的Notification服务部署到生产环境。新服务还没有完成的事实是极好的，此时，由于我们对Notifications抽象的新实现尚不存在，因此实际上并不会调用新的Notification服务。但是，我们可以部署新的服务，对其进行测试，并验证我们已经实现的功能是否可以正常运行。

此阶段可能会持续很久。[Jez Humble](#)详细介绍了如何利用抽象分支模式来[迁移GoCD中的数据库持久层](#)，当时GoCD被称为Cruise Control。从iBatis到Hibernate的切换持续了几个月的时间，在此期间，GoCD仍然保持着每周两次的交付频率。

#### 第四步：

一旦我们认为新的实现能够正常工作，此时就可以切换抽象点，以便启用我们的新实现，同时旧的功能将不再使用，如[图3-25](#)所示。

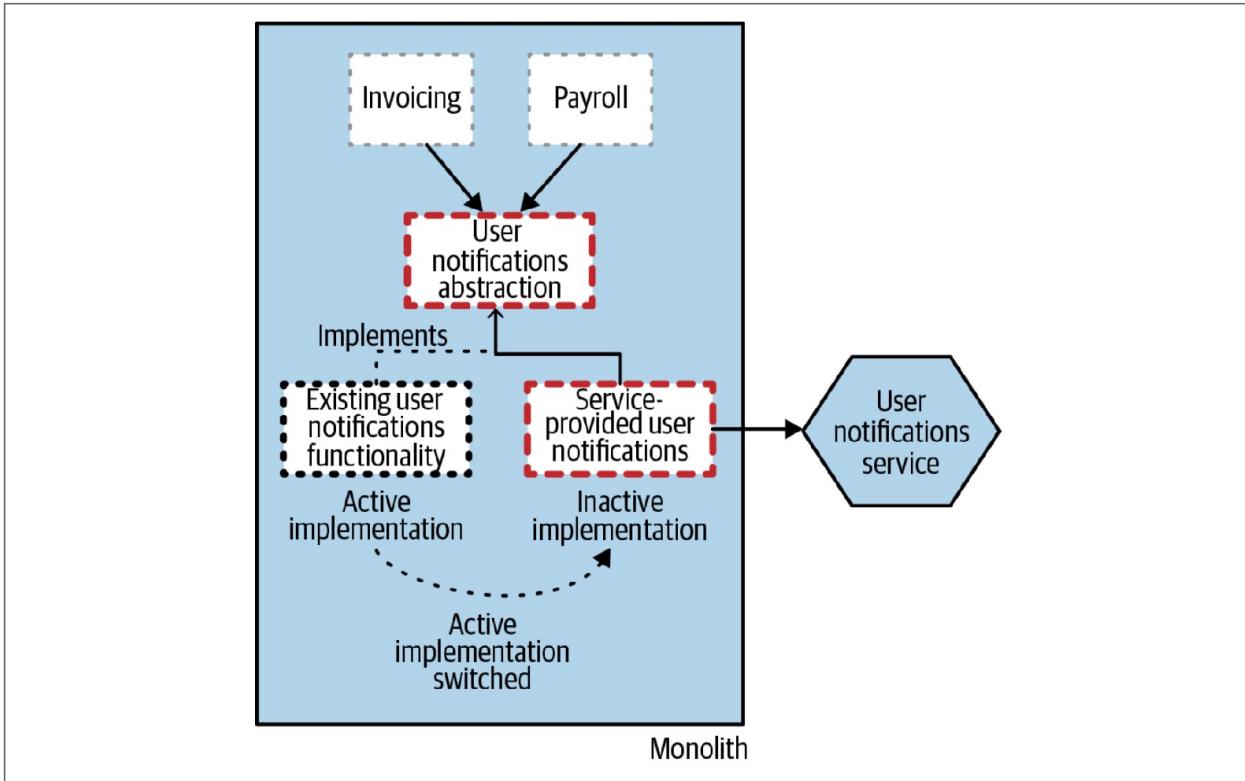


图3-25. 切换到抽象的新实现以启用我们的新的微服务

理想情况下，像绞杀者模式一样，我们希望使用某种机制以便可以轻松实现切换。如果发现问题，这种机制使我们能够快速切换回旧功能。常见的解决方案是使用 [功能开关（feature toggles）](#)。在图3-26中，我们使用配置文件实现切换，这使我们可以无需修改代码就可以改变正在使用的实现。如果想了解有关功能切换以及如何实现功能开关的更多信息，那么[Pete Hodgson的文章](#)就很好。

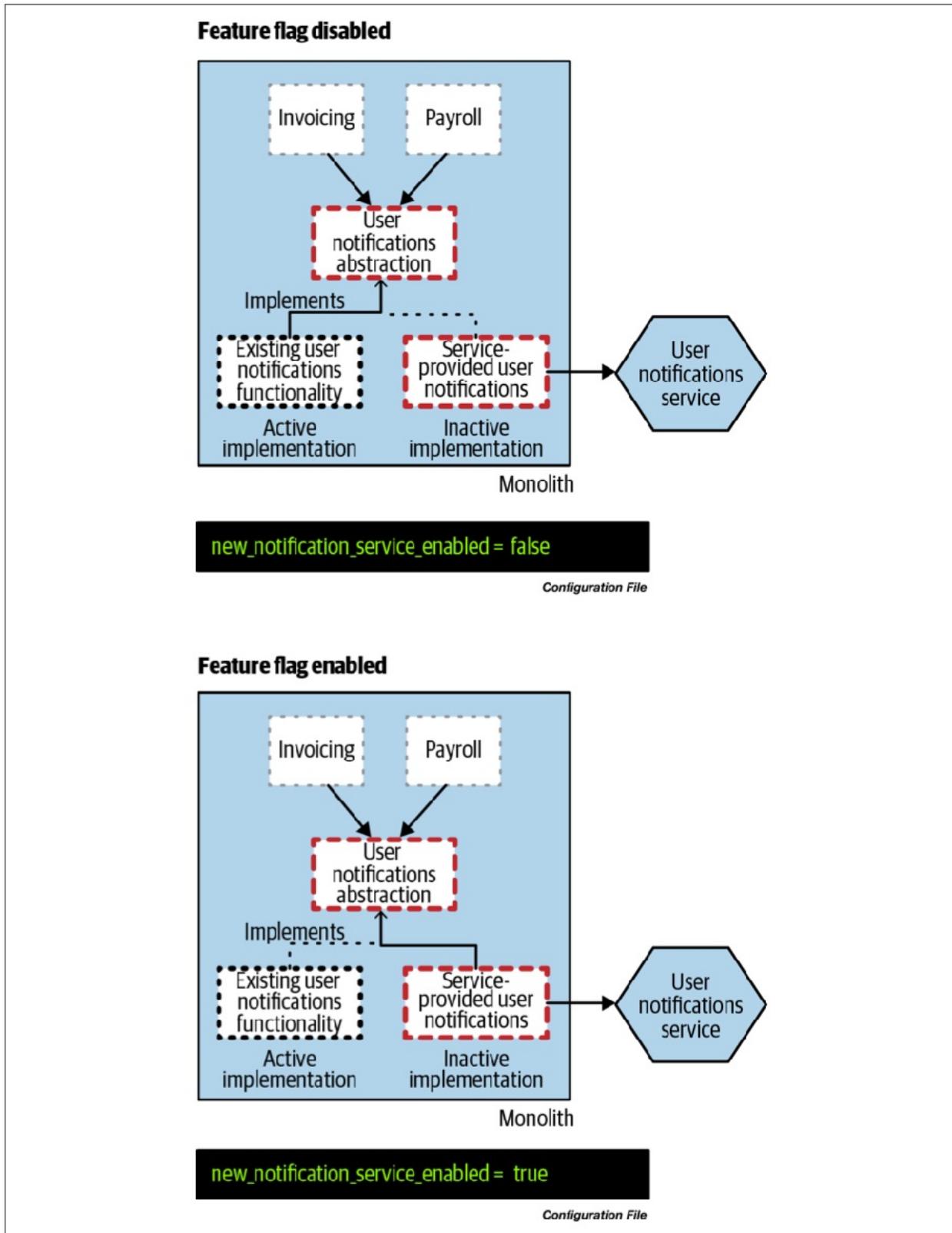


图3-26. 使用功能开关切换不同的实现

在这个阶段，我们有同一个抽象的两个实现，我们希望它们应该在功能上等效。我们可以测试其等效性，但是我们还可以选择在生产环境中同时使用这两种实现来提供额外的验证。[113页的“并行运行模式”](#)会进一步探讨此想法。

## 第五步：清理

现在，随着我们新的微服务为用户提供所有的通知，我们可以将注意力转移到背后的清理工作。此时，以前的User Notification功能已不再使用，因此显而易见的步骤就是将其删除，如[图3-27](#)所示。我们开始为单体瘦身！

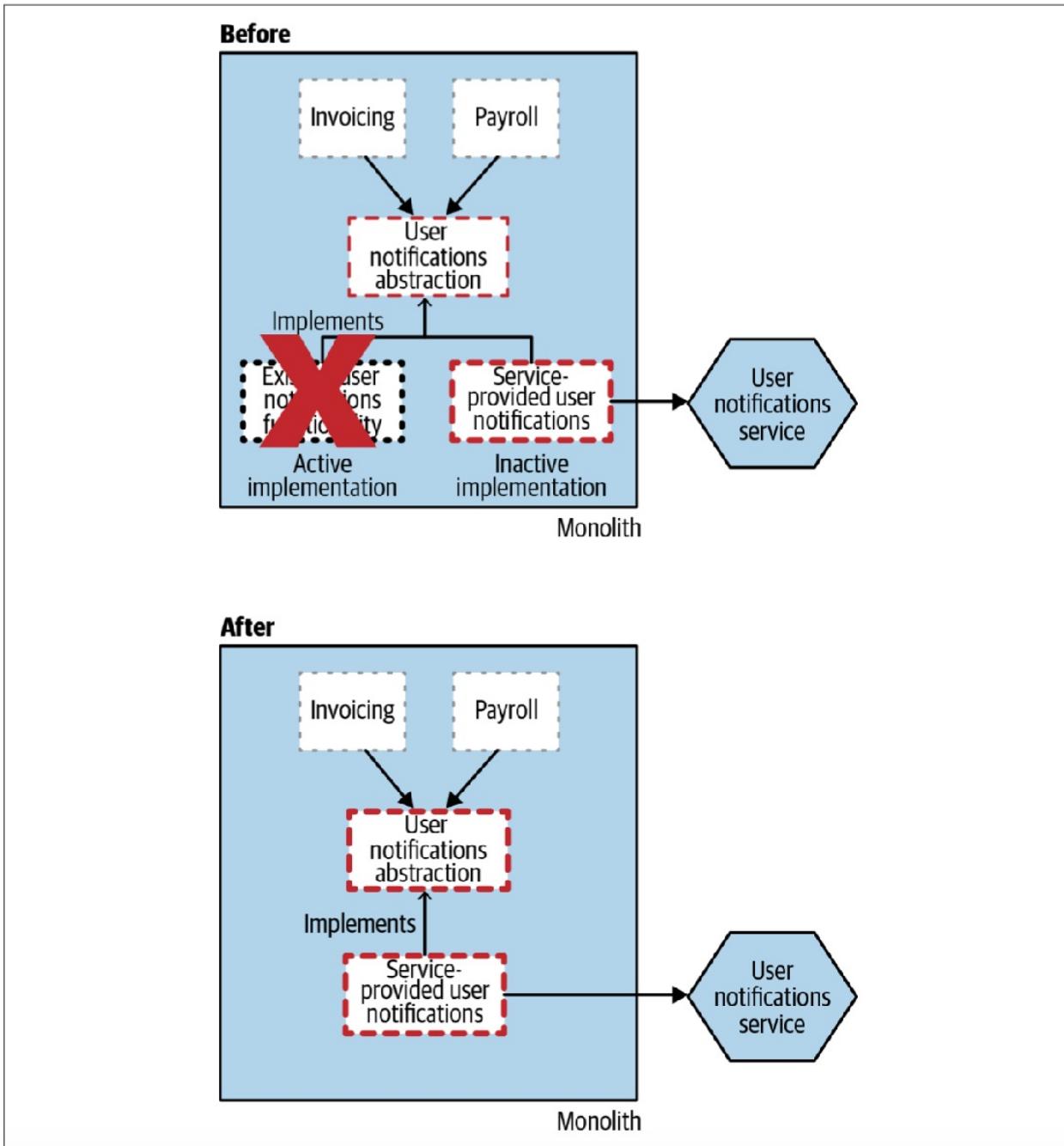


图3-27. 移除旧的实现

当删除旧的实现时，删除我们可能已经实现的任何功能开关也很有意义。与使用功能开关相关的真正问题之一是：让旧开关无处不在——不要这样做！删除不再需要的开关，以便保持简洁。

最后，随着旧实现的消失，我们可以选择删除之前创建的抽象，如图3-28所示。但是，创建抽象可能会将代码库改进到需要保持其存在的程度。如果抽象像接口一样简单，则保留抽象对现有代码库的影响最小。

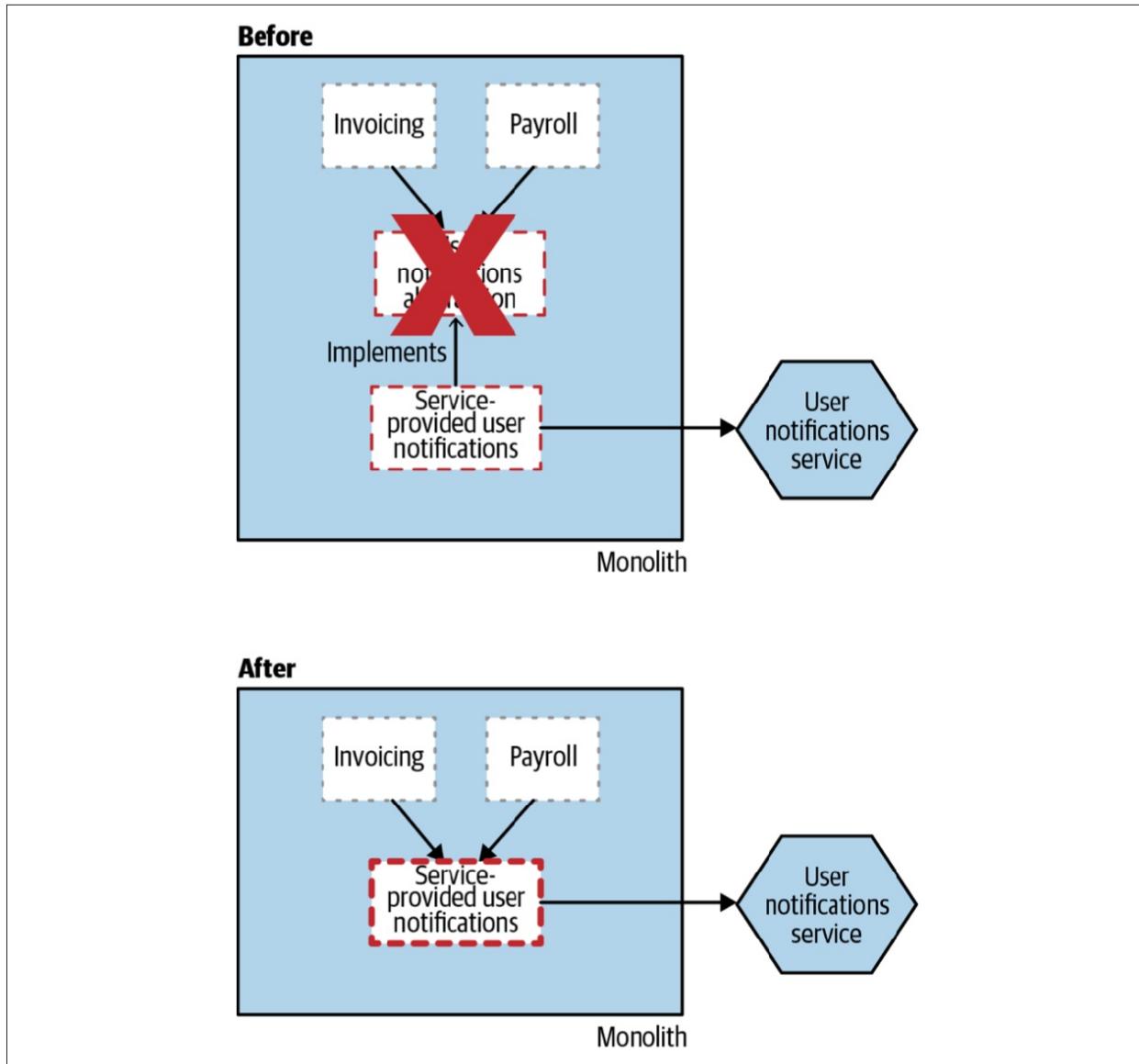


图3-28. 移除抽象（该步骤为可选操作）

# 后备方案

如果我们的新服务无法正常运行，我们可以切换回旧的实现，这种能力非常有用。但是，有没有办法可以实现其自动切换呢？Steve Smith详细介绍抽象分支模式的一种变体——[校验抽象分支模式](#)。如图3-29所示，校验抽象分支模式现了一个实时的校验步骤，其想法是，如果对新实现的调用失败，则可以改用旧实现。

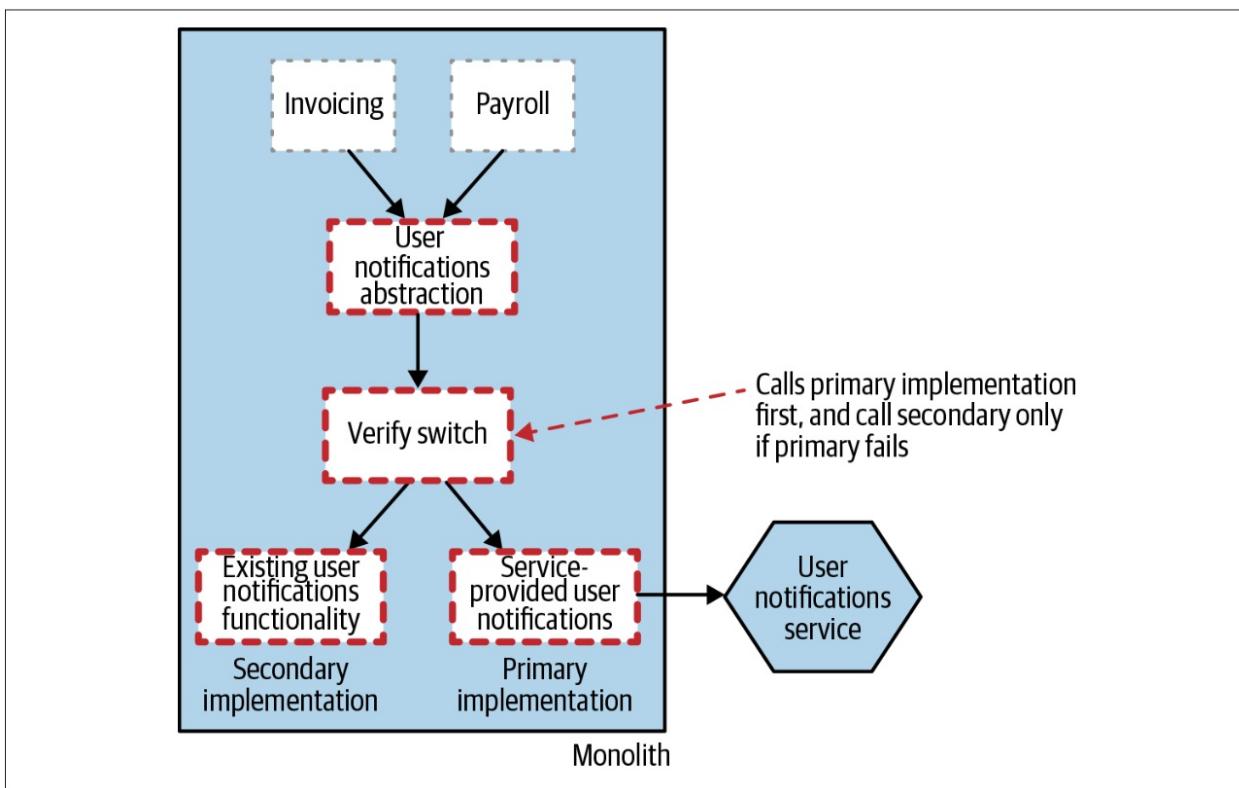


图3-29. 校验抽象分支模式

这显然增加了一些复杂度，不仅增加了代码的复杂度，而且还增加了系统推理的复杂度。实际上，抽象的两种实现都可能在任何给定的时间点处于活动状态，从而更加难以理解系统的行为。如果这两个实现无论如何都是有状态的，那么我们还需要考虑数据一致性问题。尽管对于任何情况下的

实现切换而言，数据一致性都是一个挑战，但是“校验抽象分支模式”允许我们在请求维度上实现来回切换不同的实现。这意味着需要一个这两种实现都可以访问的共享数据集。

稍后，我们将在介绍更通用的并行运行模式时，更详细地探讨如上的想法。

# 何处使用

抽象分支模式是一种通用模式。当对现有代码库的修改会持续很长时间，但我们又想避免在修改过程中对其他开发人员造成干扰时，对于这种修改的任何情况而言，抽象分支模式都可以发挥作用。我认为，与在几乎所有情况下都使用跨度较长的代码分支相比，抽象分支是一个更好的选择。对于向微服务架构的迁移，我几乎总是希望首先使用绞杀者模式，因为它在许多方面都比较简单。但是，在某些情况下，例如此处的Notification，则不能使用绞杀者模式。

抽象分支模式假定我们可以修改现有系统的代码。如果出于某种原因，我们无法修改系统代码，则可能需要看一下其他的方案，我们将在本章的其余部分探讨其中的一些方案。

Copyrights © wangwei all right reserved

# 新老并行模式

在部署新实现之前，所能做的事情也就只有执行尽可能多的测试了。在正常测试过程中，需要尽最大努力来尽可能的模拟生产环境，并完成对新微服务的预发布的验证。但是我们都知道，考虑生产环境中可能发生的每个场景并非总是可以实现。但是，我们还有其他可用的技术。

绞杀者模式和抽象分支模式都允许在生产环境中同时存在同一功能的新旧实现。通常，这两种技术都允许我们或者执行单体中的旧实现，或者执行基于微服务的新解决方案。为了降低切换到新的基于服务的实现的风险，这些技术使我们能够快速切换回以前的实现。

当使用并行运行时，我们不是调用新旧实现的其中之一，而是同时调用二者，以允许我们比较其结果以确保它们是等效的。尽管调用了两种实现，但在任何给定的时间内，只有一个实现的结果是正确的。一般而言，在不断校验并相信我们的新实现之前，我们认为旧实现的结果是正确的。

尽管并行运行模式通常用于并行运行两个系统，但该模式已经以不同的形式被使用了数十年。当比较相同功能的两种实现时，并行运行模式在单个系统中同样有用。

并行运行技术不仅可以用于验证：新实现是否可以提供与现有实现一致的响应。并行运行技术还可用于非功能性参数的验收。例如，新服务的响应是否够快？我们发现太多超时了吗？

# 比较信贷衍生品定价的例子

许多年前，我参与了一个项目，该项目的目的在于修改一种称为信贷衍生品 (*credit derivatives*) 的金融产品的计算平台。银行需要确保他们所提供的各种衍生品对他们而言是明智的。银行能在这笔信贷衍生品的交易中赚钱吗？该交易风险大吗？该信贷衍生品一旦发行，市场环境也将发生变化。因此，银行还需要评估当前交易的价值，以确保它们不会因市场环境的变化而遭受巨额亏损。<sup>6</sup>

我们几乎完全重写了执行这些重要计算的现有系统。由于涉及的资金量很大，而且很多人的奖金也基于所进行的交易的价值，因此人们对这种变化非常关注。我们决定同时用新老系统执行两组计算，并每天对比新老系统的计算结果。定价事件是通过事件触发的，所以，复制定价事件很简单，因此两个系统都可以计算定价，如图3-30所示。

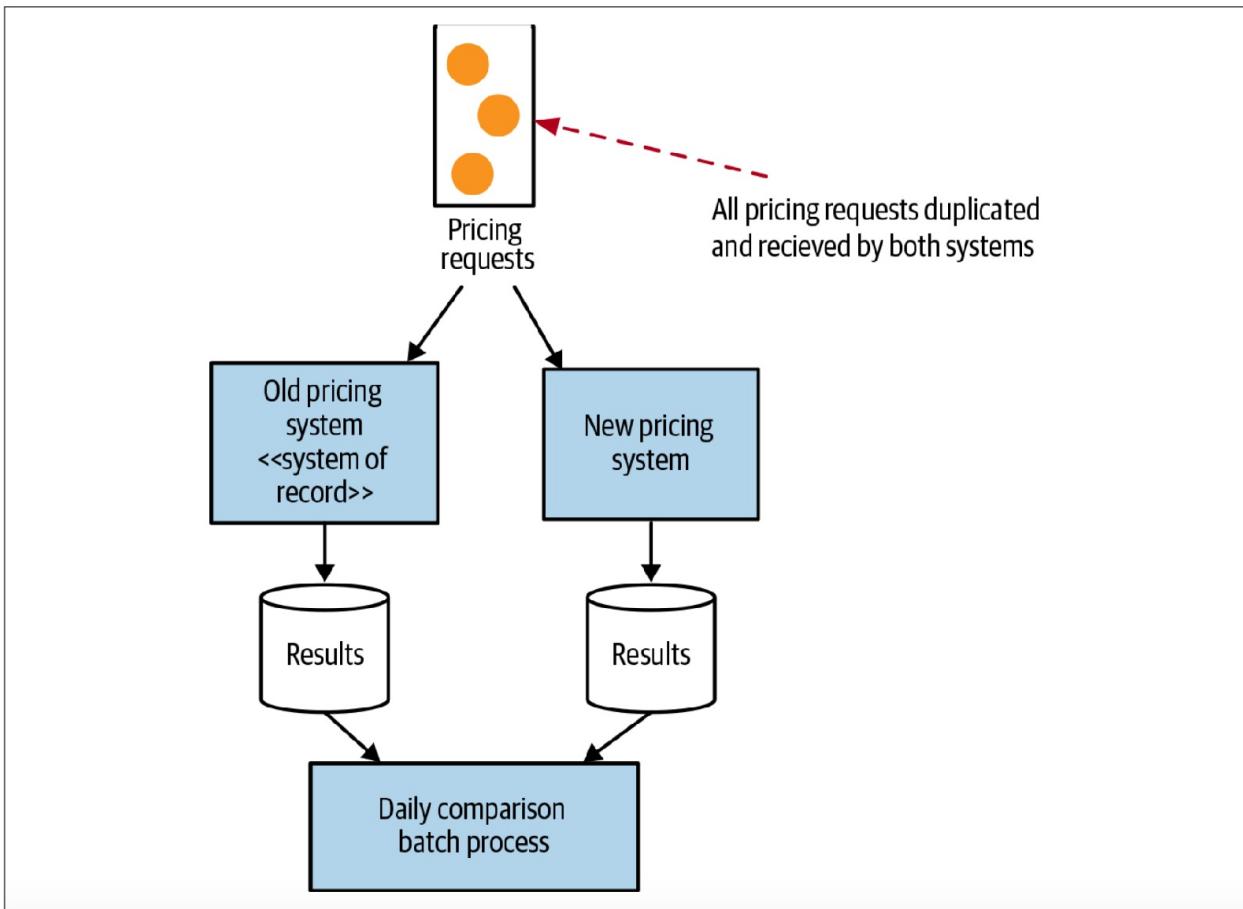


图3-30. 并行运行的例子——调用两个定价系统，并离线比较结果

每天早晨，我们会对结果进行对账，并考虑结果中的任何差异。实际上，我们写了一个程序来对账。我们将对账结果写入Excel中，以便可以轻松地与银行专家讨论变化。

事实证明，我们确实有一些问题需要修复，但是我们还发现了由现有系统的bug而引起的大量差异。这意味着某些结果的差异实际上是正确的，但是我们不得不对其进行解释（由于可以在Excel中显示结果，因此该工作变得更加容易）。我记得，我不得不与分析家坐下来，并解释了为什么要以老系统的计算结果为准。

最终，一个月后，我们将计算系统切换到我们的新平台。又过了一段时间，我们淘汰了旧系统（因为我们要对老系统上完成的计算执行审计，因此我们才让老系统又运行了几个月）。

# Homegate公司的房产清单的例子

正如我们先前在第93页“[FTP的例子](#)”中所讨论的那样，Homegate并行运行两个列表导入系统，并比较新微服务与现有单体的运行结果。客户上传一次FTP会触发两个系统的运行。一旦确认新的微服务以同样的方式运行，则会在旧的单体中禁用FTP导入。

## N-Version编程

可以说，在某些安全性至关重要的控制系统（例如飞机的电传飞行控制系统（*fly by wire*））中会存在某种并行运行的变体。飞机不再依赖机械控制，而是越来越依赖数字控制系统（*digital control system*）。当飞行员使用fly-by-wire而不是拉线来控制飞机的方向舵（*rudder*）时，会将控制输入发送到控制系统，以决定飞机的偏航角度。这些控制系统必须解释它们正在发送的信号，并采取适当的行动。

显然，这些控制系统中的bug相当危险。为了消除系统缺陷的影响，在某些情况下，会同时使用相同功能的多个实现。信号会发送到同一子系统的所有实现中，然后不同的实现对该信号做出响应。然后，对不同实现的响应结果进行比较并选出正确的响应，一般而言，该选择通过投票来表决。这种技术被称之为N-version编程技术<sup>7</sup>。

N-version编程的最终目标是不替换任何实现，这与我们在本章中介绍的其他模式不同。反之，可以相互替换的所有实现将继续彼此共存，并且这些可以相互替换的实现有望减少任何给定子系统中的bug所带来的影响。

# 校验技术

我们可以用并行运行模式来比较两个实现的功能等效性。如果我们以前面介绍的信贷衍生品定价为例，则可以将两个版本视作函数——给定相同的输入，我们期望得到相同的输出。但是，我们也可以（并且应该）对非功能性方面进行验证。跨网络边界的调用可能会引入大量的延迟，并且可能会因为超时，分区（*partitions*）等原因而导致请求丢失。因此，我们的验证过程还应该扩展到：确保在可接受的故障率范围内以及响应时间内完成对新微服务的调用。

# 使用间谍技术

对我们之前的通知功能的例子而言，我们不想向客户发送两次电子邮件。在这种情况下，间谍技术可能会派上用场。间谍模式来源于单元测试，间谍程序可以代表某项功能，并允许我们在某些事情完成后进行验证。间谍程序以桩的形式表示并替换某些功能。

因此，对于Notification功能，我们可以使用间谍程序替换实际发送电子邮件的代码，如图3-31所示。然后，新的通知服务将在并行运行阶段使用此间谍程序，以便我们可以验证：当服务接收到sendNotification调用时是否会触发发送电子邮件的行为<sup>译注1</sup>。

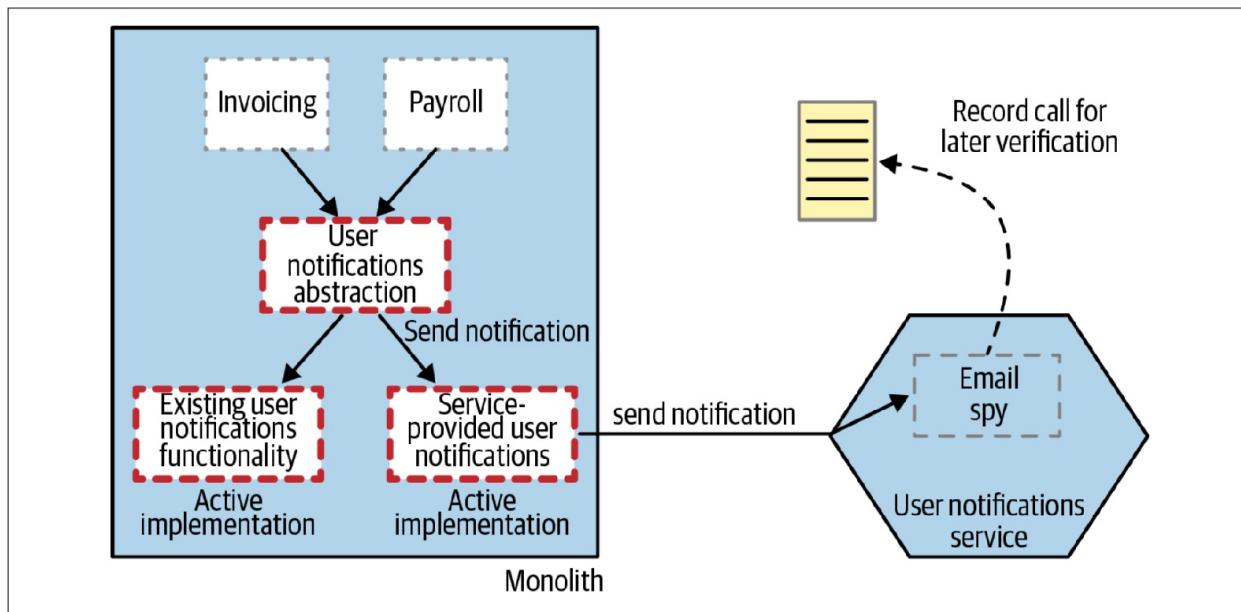


图3-31. 在并行运行阶段，用间谍程序来校验发送邮件的功能

请注意，我们可能已经决定在单体内部使用间谍程序来避免对Notification服务发起远程调用。但是，这可能并非我们希望的事情，因为实际上，我们确实希望考虑远程调用的影响，以了解新Notification服务的超时、故障

或一般延迟是否会导致问题。

事实上，间谍程序运行于单独的进程中，这会增加验证过程的复杂度。如果我们希望在原始请求的范围内进行实时验证，则可能需要在Notifications服务上开放某些方法，以允许在调用发送到我们的Notifications服务之后可以进行验证。这种做法的工作量很大，而且在许多情况下，我们不需要实时验证。交互录制模型可以用于校验进程外的间谍程序，以允许执行带外(*out of band*)校验<sup>译注2</sup>(也许是天级别的校验)。显然，使用间谍程序替换对Notifications服务的调用，可以简化校验，但同时，我们可以做的测试却更少了！用间谍程序替代的功能越多，实际测试的功能就越少。

# **Github Scientist**

GitHub的Scientist库是一个著名的库，可帮助我们在代码级实施并行运行模式。Scientist是一个Ruby库，可以并行运行新、旧实现并捕获有关新实现的信息，以帮助我们了解新的实现是否正常运行。我自己并没有使用过Scientist库，但是我可以看到使用这样的库是如何真正帮助大家基于现有系统来验证新的微服务系统。如今，该库已经有多个语言版本的实现：Java，.NET，Python，Node JS，.....

# Dark Launching and Canary Releasing

值得一提的是，并行运行不同于传统意义上的金丝雀发布。金丝雀发布会让部分用户体验到新功能，而其他的大多数用户看到的还是旧的实现。这种方法的目的是：如果新系统出现问题，则仅会影响一小部分的请求。

另一个相关的技术被称为**dark launching**<sup>译注3</sup>。利用**dark launching**，我们可以部署并测试新功能，但是新功能对用户是不可见的。因此，并行运行是实现**dark launching**的一种方法：实际上，在我们切换系统之前，用户都无法感知到“新”功能。

**dark launching**，并行运行，和金丝雀发布是用来验证新功能是否正常工作的技术。在新功能不能按预期的工作时，该技术还可以用来降低影响。所有的这些技术都被称为“渐进式交付（*progressive delivery*）”。渐进式交付是James Governor创造的一个术语集，该术语集描述了用来帮助控制如何以更细微的方式向用户推出软件的系列方法，从而允许我们在验证功效时更快地发布软件。

# 何处使用并行运行模式

实施并行运行是一件很琐碎的事情，通常只对高风险的变更实施该模式。我们将在第4章中审查用于医疗记录的并行模式的例子。对于何时使用并行运行模式，我肯定会经过相当仔细的选择——实施这种模式需要与可以获取的利益进行权衡。我只使用过一两次并行运行模式，但是在使用时，它极其有用。

---

<sup>6</sup>. Turned out we were terrible at this as an industry. I recommend Martin Lewis's *The Big Short* (W. W. Norton & Company, 2010) as an excellent overview of the part that credit derivatives played in the global financial crisis of 2007–2008. I often look back at the small part I played in this industry with a great deal of regret. It turns out not knowing what you're doing and doing it anyway can have some pretty disastrous implications. ↪

<sup>7</sup>. See Liming Chen and Algirdas Avizienis, “N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,” published in the Twenty-Fifth International Symposium on Fault-Tolerant Computing (1995). ↪

译注<sup>1</sup>. side effect是针对函数而言的，如果一个函数修改了自己范围之外的资源，则称之为有副作用，否则，就没有副作用。此处，新的服务会触发给用户发送邮件，因此，称之为副作用。 ↪

译注<sup>2</sup>. 在计算机领域，带外管理（Out-of-band management）指的是使用独立管理通道来进行设备维护。因此，此处的带外校验也就是利

用独立的其他进程来校验结果。 ↵

译注<sup>3</sup>. 此处不再对该术语进行翻译，避免因此而带来的语义差异。 ↵

Copyrights © wangwei all right reserved

# 装饰协作模式

如果要基于单体内部发生的事情来触发某些行为，但又无法修改单体时，会发生什么？此时，装饰协作模式可以给予我们很大的帮助。广为人知的装饰器模式可以允许我们在不改变原有对象的情况下给对象增加新功能。我们将使用装饰器来让我们的单体看起来像是直接调用了我们的服务，即使我们实际上并未修改单体。

我们不是在请求达到单体之前截获调用，而是允许这些请求照常执行。然后，基于请求的结果，我们可以利用任意的协作机制来调用外部微服务。让我们通过Music Corp的例子来详细探讨这个想法。

# 会员计划的例子

Music Corp公司整体上在围绕用户而工作！我们希望增加根据用户下单情况而给用户增加积分的功能，但是我们当前的下单功能非常复杂，我们不希望立即对其修改。因此，下单功能将保留在现有的单体中，但我们将使用一个代理来拦截这些调用，并根据单体的响应结果来确定要给用户增加多少积分，如[图3-32](#)所示。

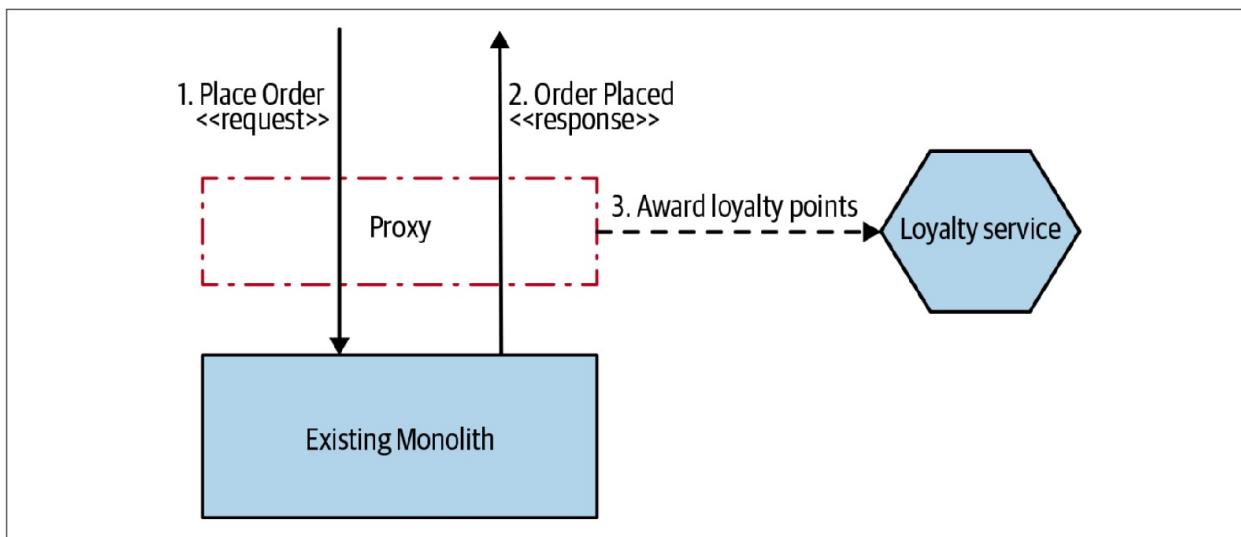


图3-32. 成功下单后，代理会调用Loyalty服务为用户增加积分

绞杀者模式让代理非常简单。现在，我们的代理正具备更多的“智能”。它需要请求新的微服务，并将响应回传给用户。和以前一样，请时刻关注代理的复杂度。在代理中增加的代码越多，最终，它就越有可能成为具有微服务功能的微服务。此时，代理作为一种技术的服务，也会具有我们之前讨论的关于微服务的所有挑战。

另一个潜在的挑战是，我们需要从入站请求中获取足够的信息才能够调用微服务。例如，如果我们想基于订单的价格来奖励积分，但是在下单的请求或响应中又没有订单价格，那么我们可能需要查找额外的信息——也许会回调单体来获取所需的信息，如图3-33所示。

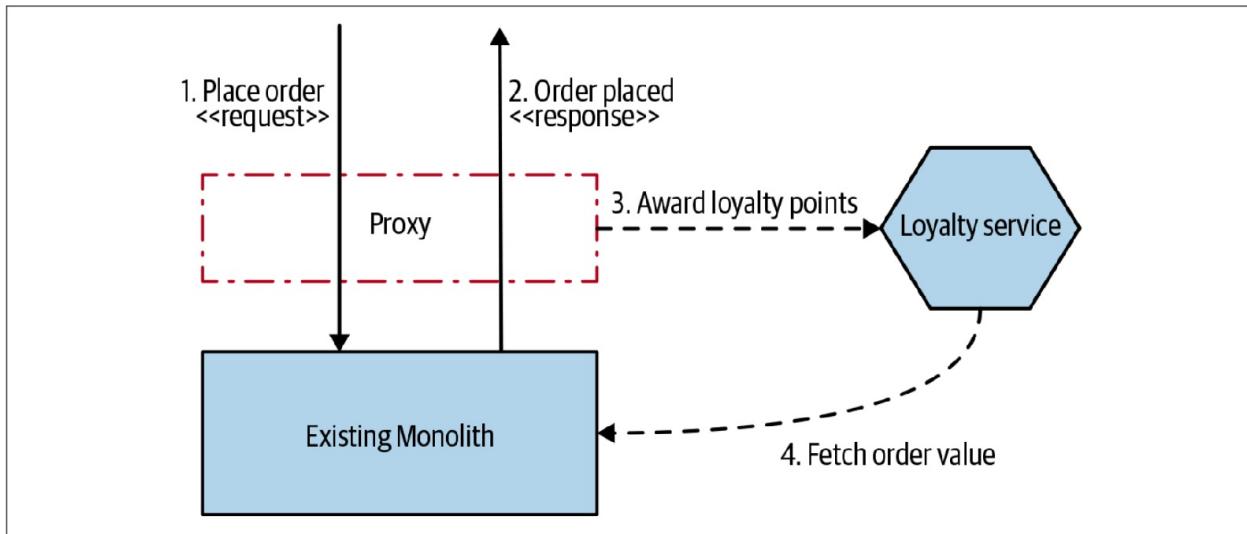


图3-33. Loyalty服务需要加载额外的订单的详细信息来确定奖励用户多少积分

该调用可能会产生额外的负载，并且会引入循环依赖，因此最好修改单体以便在下单完成后可以提供所需的信息。但是，这可能需要修改单体的代码，或者可能需要使用更具侵入性的技术，例如变更数据捕获技术（CDC: *change data capture*）。

# 何处使用

在保持简单时，与CDC相比，装饰协作模式是一种更优雅、耦合性更低的方法。当可以从入站请求或者单体响应中提取所需信息时，装饰协作模式最有效。在需要更多信息才能调用新服务时，装饰协作模式最终会变得更加复杂。我的直觉是，如果单体的请求和响应不包含所需的信息，在使用此模式之前请仔细思考。

Copyrights © wangwei all right reserved

# CDC模式

不用拦截并处理对单体发起的请求，我们可以使用CDC技术对数据库中的变更做出反应。为了使CDC正常工作，必须将底层的捕获系统耦合到单体的数据库中。这一点确实是CDC模式中的无法避免的挑战。

# 发行会员卡的例子

我们希望集成一些功能以便在用户注册时为其打印会员卡。目前，在注册时会创建一个会员帐户。如图3-34所示，当从单体返回注册请求的响应时，我们仅知道用户已注册成功。为了打印会员卡，我们需要和用户相关的更多详细信息。这使得在单体上游插入该行为——可能使用装饰协作模式——更加困难，我们需要在响应返回的地方增加对单体的请求以提取所需的其他信息，而这些信息是否通过开放的API而提供是未知的。

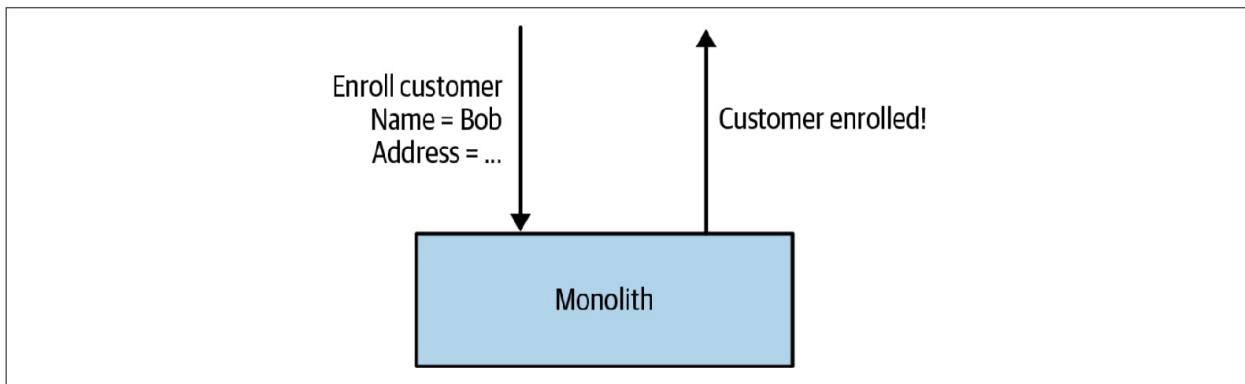


图3-34. 当用户完成注册后，单体无法提供更多的信息

我们决定使用CDC模式。我们会检测LoyaltyAccount表的所有插入操作，数据插入后，我们将调用新的Loyalty Card Printing服务，如图3-35所示。在这种特殊场景下，我们决定触发一个“Loyalty Account Create”事件。我们的打印进程最好以批处理方式运行，因此，我们可以在消息代理（*message broker*）中建立要执行的打印列表。

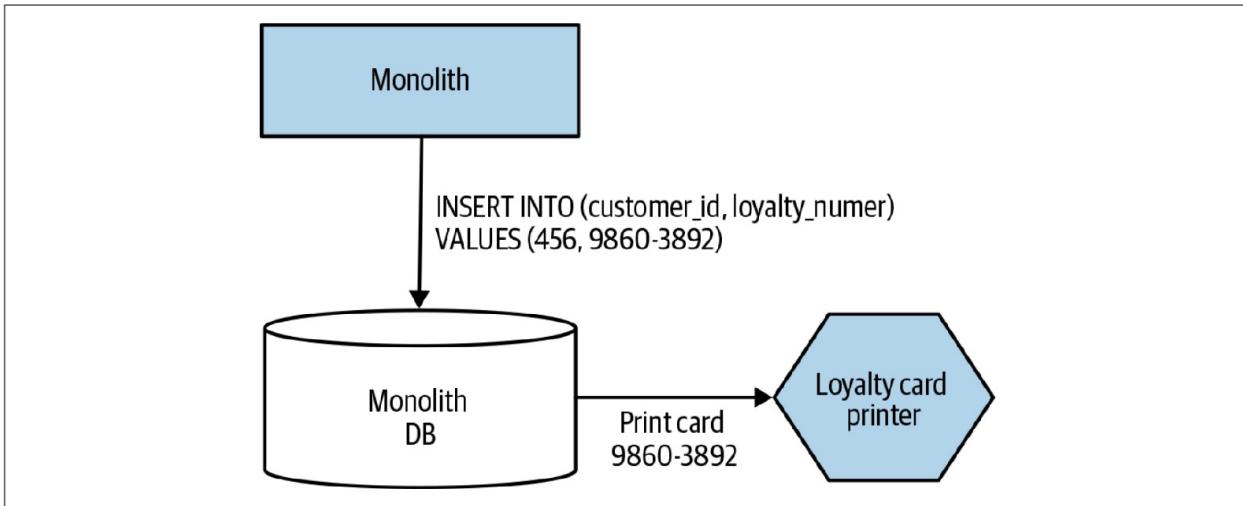


图3-35. CDC如何调用新的打印服务

# CDC的实现

我们可以使用各种技术来实现CDC，所有这些技术在复杂性、可靠性和及时性方面都有不同的权衡。让我们来了解下其中的几种技术。

## 数据库触发器

大多数关系型数据库都允许在修改数据时触发自定义的行为。这些触发器的确切定义以及它们可以触发的事情各不相同，但是所有的现代关系型数据库都以某种方式支持这些触发器。在图3-36的例子中，只要 LoyaltyAccount表存在INSERT操作，就会调用我们的服务。

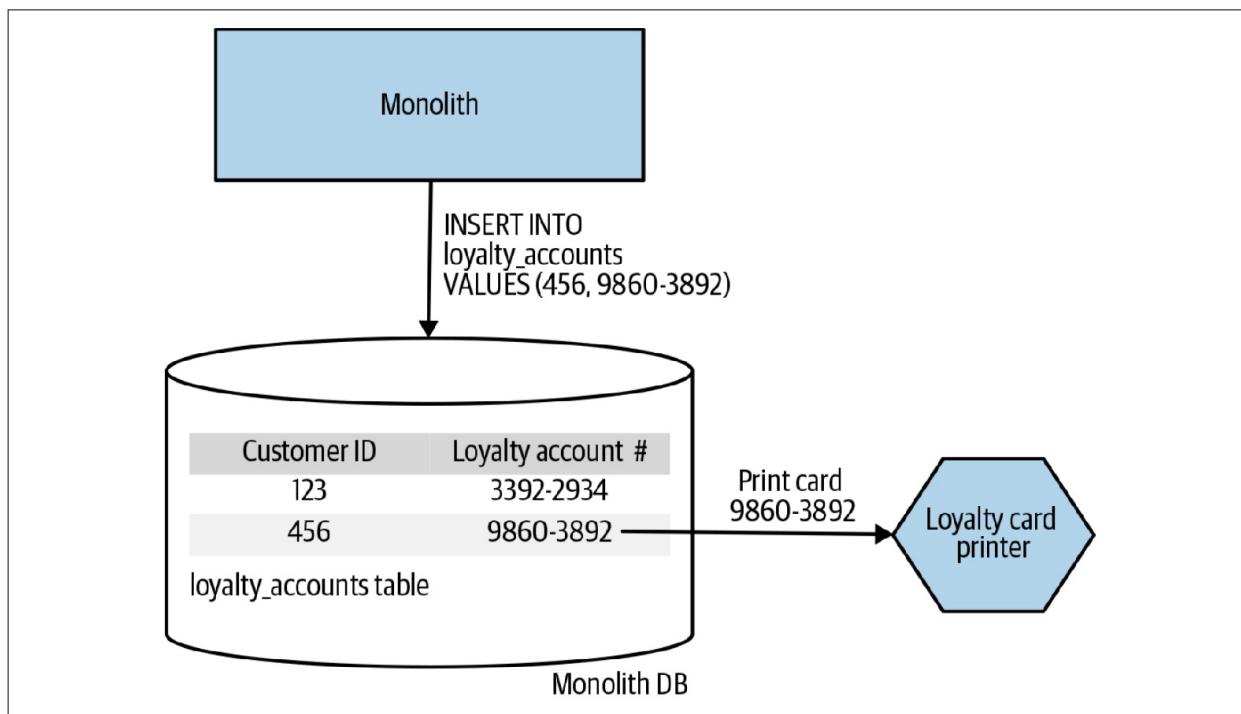


图3-36. 当插入数据时，使用数据库触发器来调用微服务

像其他任何存储过程一样，触发器也需要安装到数据库中。这些触发器的功能可能也有局限性，尽管至少对于Oracle而言，可以用触发器来调用Web服务或自定义Java代码。

乍一看，触发器似乎很简单。无需运行任何其他软件，无需引入任何新技术。但是，像存储过程一样，数据库触发器可能会让事情越来越糟。

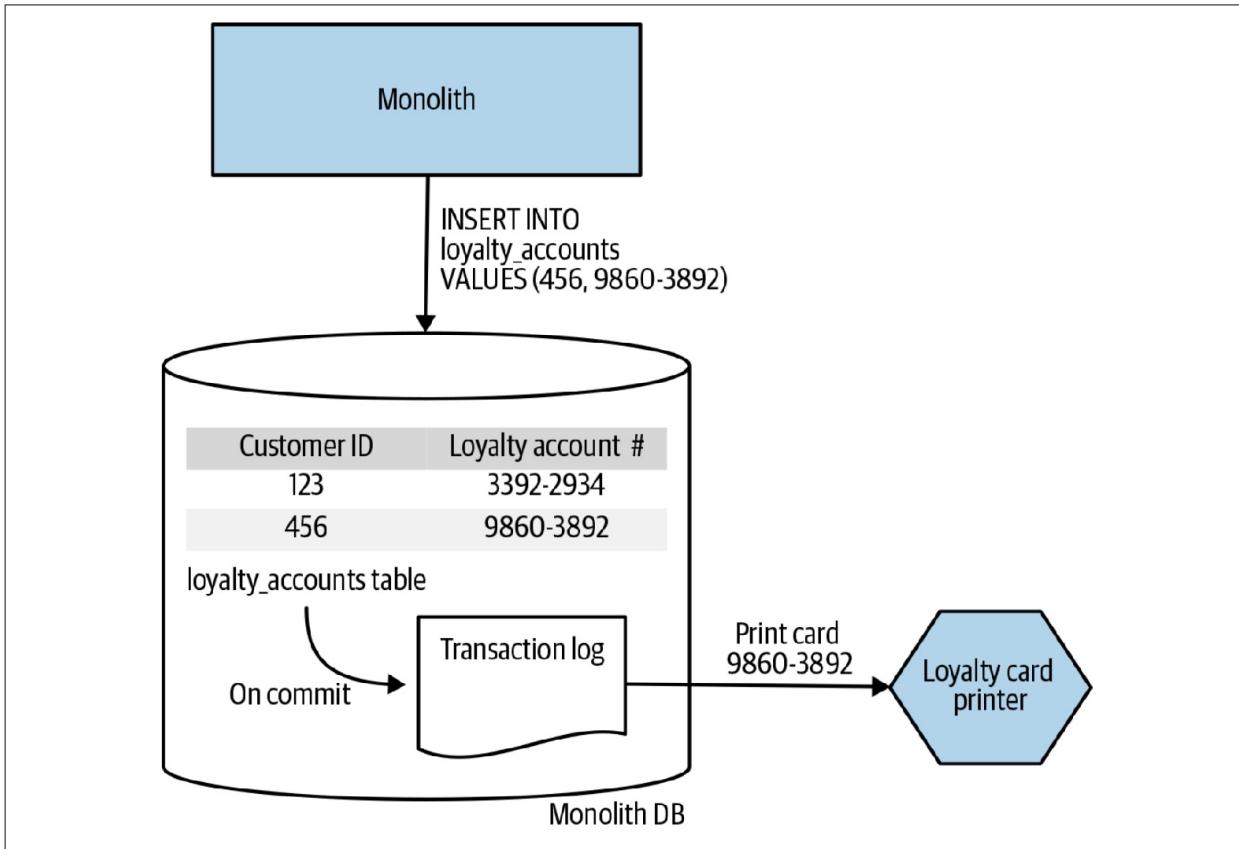
我的一个朋友Randy Shoup曾经说过：“一两个数据库触发器并不可怕，但用触发器来构建整个系统是一个可怕的想法。”这通常是与数据库触发器相关的问题。触发器越多，就越难理解系统的实际工作方式。问题经常出在数据库触发器的工具和变更管理——触发器使用的过多，应用程序就可能会成为某种巴洛克风格的建筑[译注1](#)。

因此，如果要使用触发器，请非常谨慎地使用它们。

## 轮训事务日志

在大多数数据库（当然是所有的主流事务数据库）中，都存在一个事务日志。事务日志通常是一个文件，其中记录了所有已完成的数据库的修改。对于CDC而言，最精密的工具都倾向于使用此事务日志。

基于事务日志的CDC系统以单独的进程而运行，并且系统与现有数据库的唯一交互是通过事务日志进行的，如[图3-37](#)所示。此处，值得注意的是，只有已提交的事务才会记录在事务日志中（这一点非常重要）。



基于事务日志的CDC工具需要了解底层的事务日志格式，并且不同类型的数据库的事务日志格式有所不同。因此，此处可以使用的工具取决于我们所使用的数据库。尽管有大量的工具用于解析事务日志，但大多数工具都是用来支持数据复制。还有许多解决方案旨在把事务日志的变更映射为消息代理（*message broker*）中的消息。如果微服务本质上是异步的，那么这中映射可能非常有用。

基于事务日志的CDC工具需要了解底层的事务日志格式，并且不同类型的数据库的事务日志格式有所不同。因此，此处可以使用的工具取决于我们所使用的数据库。尽管有大量的工具用于解析事务日志，但大多数工具都是用来支持数据复制。还有许多解决方案旨在把事务日志的变更映射为消息代理（*message broker*）中的消息。如果微服务本质上是异步的，那么这中映射可能非常有用。

除了一些限制外，在很多方面，基于事务日志的CDC是最简洁的解决方案。事务日志本身仅显示底层数据的变化，因此不必担心需要查出到底发生了什么更改。工具在数据库之外运行，并且可以基于事务日志的副本运行，因此也无需过多的担心耦合或冲突。

## 批处理增量复制

可能最简单的方法是编写一个程序，该程序定期扫描数据库并检查哪些数据发生了变化，然后将变化的数据复制到目标位置。这些任务通常使用 cron 之类的工具或类似的批处理调度工具来执行。

主要的问题是要清楚自上次复制以来哪些数据发生了变化。数据库的模式设计可能让数据修改显而易见，也可能无法察觉数据的变化。有些数据库允许查看表的元数据，以查看数据库的某些部分何时发生了修改，但这种方法并不是通用的方法。当希望获取行级修改的信息时，可能只能获取表级修改的时间戳。我们可以自己添加这些时间戳信息，但这可能会增加大量的工作，并且 CDC 系统可以更优雅地处理数据变化的问题。

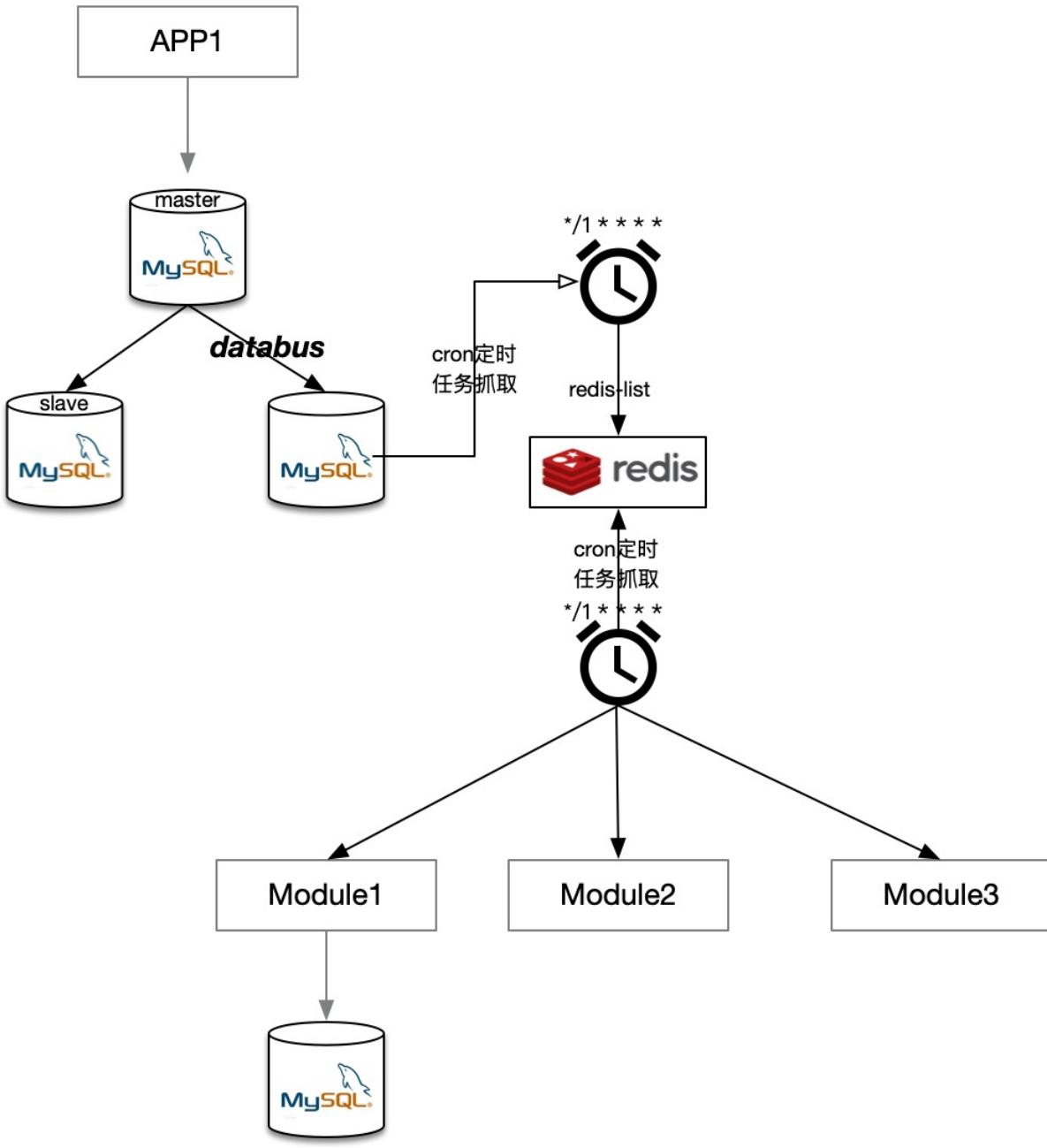
# 何处使用

CDC是一种有用的通用模式，尤其是在需要复制数据的情况下（我们将在第4章中做更多探讨）。在微服务迁移的情况下，最有效的地方是需要对单体数据的变化做出反应，但是此时又无法使用绞杀者模式或装饰器模式在系统范围内拦截该数据变化，同时又无法修改底层的代码库。

通常，由于CDC模式的一些实现会遇到挑战，因此我会尝试尽量少使用这种模式。数据库触发器有其缺点，而功能完善的CDC工具可以解决事务日志的处理，但是却会给我们的解决方案增加极大的复杂度。但是，如果您理解了这些潜在挑战，那么这将是一个有用的工具。

## 尽量少用CDC

2年前，我帮助一个在线业务来梳理质量风险，在梳理的过程中就发现该系统的核心重点功能采用一种基于事务日志的CDC系统来完成，整个系统的架构如下图所示。



业务会将数据存储在Mysql中，为了使得其他业务也能够感知数据变化，采用到了databus工具来同步数据变化到另外的一个数据。然后会用一个cron定时任务从转存的数据中拉取新数据并存入list列表作为消息队列使用，接下来会用另外的一个cron定时任务从redis中获取消息队列的数据，并将数据发送给其它业务来出来该数据。

对于离线业务或者迁移的过渡阶段而言，这种构架无可厚非，但是对于在线业务而言，这种架构的问题就太多了，尤其是在整个架构中的不同部分的监控都不完善的情况下。

忽然，有一天，负责同步数据的databus所在的机器宕机了，这导致了整体依赖该CDC的在线业务的数据全部出现了延迟。

使用CDC确实会引入更多的复杂性，如果可以，我真心觉得还不如使用通用的消息队列（例如ActiveMQ，Kafka……）来实现这种数据变化的感知。

---

译注<sup>1</sup>. 巴洛克建筑是17~18世纪在意大利文艺复兴建筑基础上发展起来的一种建筑和装饰风格。放荡不羁，极尽奢华是巴洛克建筑的主要特征，在巴洛克建筑里装饰满了壁画雕塑。这种建筑的特点是重于内部的装饰，其全体多取曲线，常常穿插曲面与椭圆空间。此处借用巴洛克风格的建筑来形容使用太多触发器而构建的系统，因为触发器也是位于数据内部的，相当于建筑的内部装饰。 ↪

Copyrights © wangwei all right reserved

正如我们已经探索的那样，有很多方法可以将功能提取到微服务中。但是，我们需要解决一个显而易见但又没有人愿意讨论的问题：即，我们如何处理数据？微服务对于信息隐藏而言是最有效的，这也促使我们朝着在微服务中完全封装其自身的数据存储和检索机制的方向发展。这导致我们得出这样的结论：在向微服务架构迁移时，如果我们希望从迁移中获得最大收益，则需要拆分单体数据库。

但是，拆分数据库并不是一件容易的事。我们需要考虑拆分期间的数据同步，逻辑模式与物理模式分解，事务完整性，joins，延迟等问题。在本章中，我们将着眼于这些问题，并探索可以帮助我们解决这些问题的模型。

但是，在开始拆分之前，我们应该研究管理单个共享数据库所面临的挑战和应对模式。

# 共享数据库

正如我们在[第1章](#)中讨论的那样，我们可以从领域耦合，时间耦合或实现耦合的角度来思考耦合。因为人们普遍在多个schema之间共享数据库。因此，在三种耦合中，对于数据库而言，我们最常用的就是实现耦合，如[图4-1](#)所示。

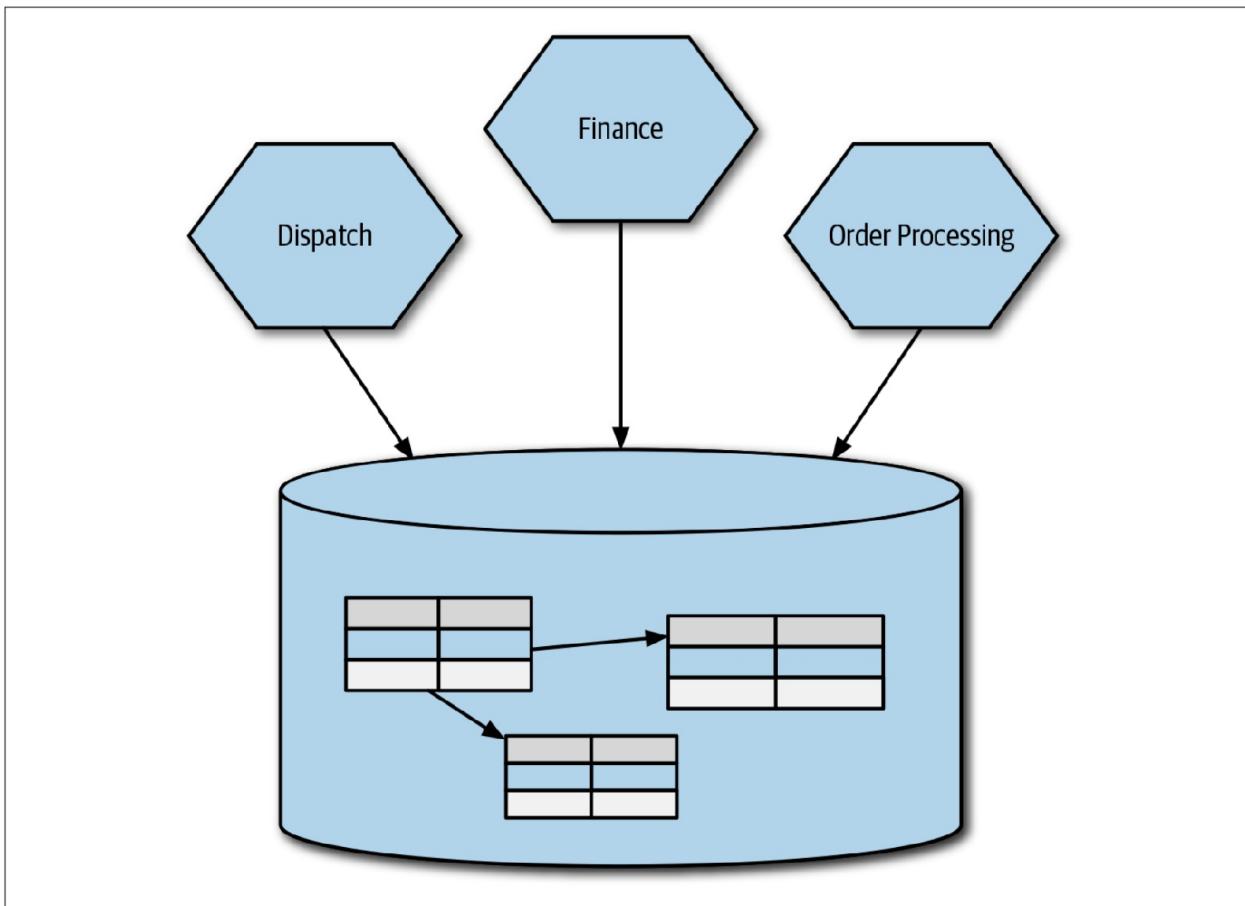
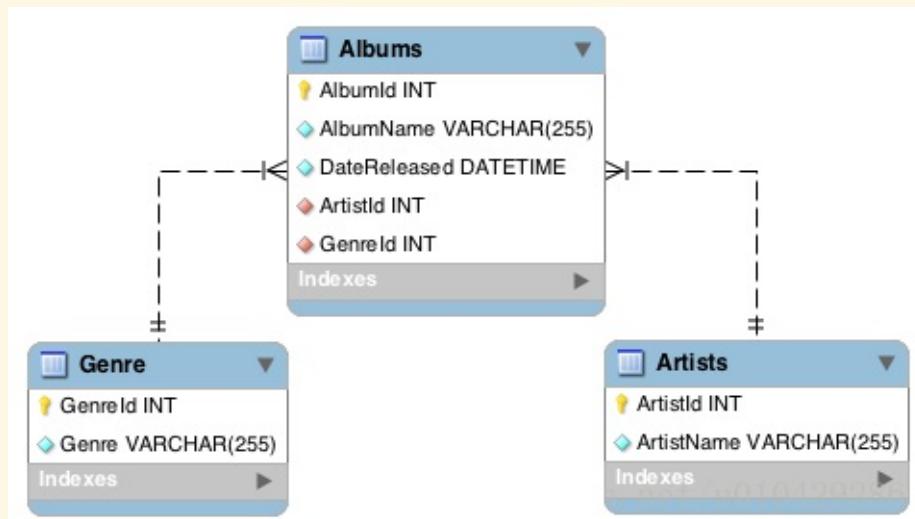


图4-1. 多个服务都直接访问同一个数据库

从表面上看，有很多问题会涉及到在多个服务之间共享单个数据库。但是，主要的问题是，我们自己放弃了决定共享什么和隐藏什么的机会，而这与我们对信息隐藏的渴望背道而驰。这意味着很难了解可以安全修改 schema 的哪些部分。知道外部可以访问数据库是一回事，但是不知道他们使用模式的哪一部分是另一回事。可以使用视图来缓解这种情况，我们将在稍后对此进行讨论，但这并不是一个完整的解决方案。

## schema

在数据库中，schema 是数据库的组织和结构，schema 中包含了 schema 对象，可以是表(table)、列(column)、数据类型(data type)、视图(view)、存储过程(stored procedures)、关系(relationships)、主键(primary key)、外键(foreign key)等。数据库 schema 可以用一个可视化的图来表示，它显示了数据库对象及其相互之间的关系，如下图所示：



以上是模式图的一个简单例子，显示了三个表及其数据类型、表之间的关系以及主键和外键。

schema 和 database 是否等同呢？是否一致，取决于数据库供应商：

1. MySQL 的文档中指出，在物理上，模式与数据库是同义的，所

以，模式和数据库是一回事。

2. 但是，Oracle的文档却指出，某些对象可以存储在数据库中，但不能存储在schema中。因此，模式和数据库不是一回事。
3. 而对于SQL Server，根据[SQLServer technical article](#)，schema是数据库SQL Server内部的一个独立的实体。所以，他们也不是一回事。

schema这个词可以用在很多不同的环境中，当切换到一个新的数据库管理系统时，一定要查看该系统是如何定义schema的。

另一个问题是，谁“控制”数据变的越来越不清晰。处理数据的业务逻辑在哪里？现在是否存在跨服务处理数据的情况？这写问题意味着业务逻辑缺乏内聚。[正如我们之前讨论的那样](#)，在将微服务视为行为和状态的组合时，就要封装一个或多个状态机。现在，如果修改状态的行为散布在系统中，那么确保可以正确实现该状态机是一个棘手的问题。

如图4-1，如果三个服务都可以直接修改订单信息，那么在服务之间的行为不一致时会发生什么？当确实需要修改修订订单的行为时会发生什么——必须将这些修改应用于所有的这三个服务吗？如前所述，我们的目标是提高业务功能的高内聚，而共享数据库往往意味着缘木求鱼。

# 解决共享数据库的方法

把数据库拆分开来以允许每个微服务拥有自己的数据，这尽管可能是一项艰巨的任务，但却几乎总是首选方案。如果无法做到这一点，则使用数据库视图（参阅128页的[数据库视图](#)）或采用数据库包装服务（请参阅132页的“[把数据库包装成服务](#)”）会有所帮助。

# 可以在何处使用共享数据库

我认为，仅在两种场景下，直接共享数据库可以适用于微服务架构。

- 第一种是当数据是只读的静态参考数据时。稍后，我们会更详细地探讨该主题。现在，考虑一个schema，该schema包含了国家货币代码信息、邮政编码或邮政编码查询表等信息。此时，数据结构是高度稳定的，并且该数据的变更控制通常作为管理任务来处理。
- 另一种场景是，当一个服务将数据库作为明确的端点（*end-point*）而直接公开时，该端点是为处理多个使用者而设计和管理。当我们讨论数据库即服务接口时，我们将进一步阐明该想法（参见135页的[数据库即服务接口](#)）。

Copyrights © wangwei all right reserved

# 然而，现在无法拆分数据库

因此，理想情况下，我们希望我们的新服务拥有独立的schema。但是，当从现有的单体系统上开启行程时，我们无法让新的服务拥有独立的schema。难道这意味着我们应该始终将这些schema进行拆分？我仍然坚信，在大多数情况下，拆分schema是合适的，但一开始就对其进行拆分并非总是可行的。

正如我们稍后将要探讨的那样，有些时候，拆分schema所涉及的工作会花费很长时间，或者会涉及对系统中特别敏感的部分的修改。在这种情况下，使用各种应对模型可能会非常有用。这些模型至少可以避免让事情变得更糟，并且在最好的情况下，这些模型会成为迈向更好的未来的不错的垫脚石。

## Schemas and Databases

过去，我经常混用“database”和“schema”这两个术语，对此我感到很内疚。因为“database”和“schema”存在一些不同，因此混用二者有时会引起混乱。从技术上讲，我们可以将模式视为保存数据的表的集合，其中这些表在逻辑上是独立的，如[图4-2](#)所示。然后，多个模式可以托管在一个数据库引擎上。根据我们所处的背景，当人们说“database”时，他们可能是指“schema”或数据库引擎。例如，当人们说“The database is down!”的时候，他们指的是数据库引擎。

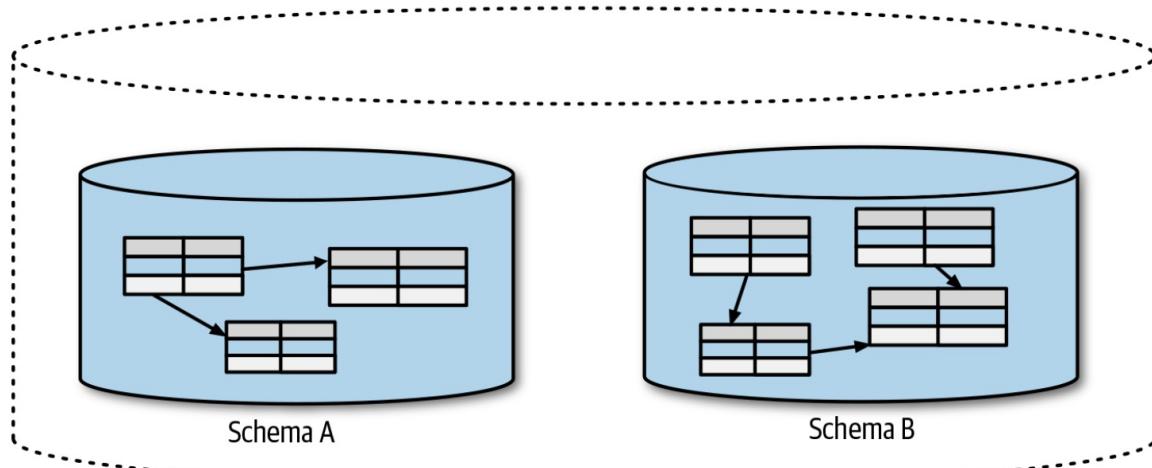
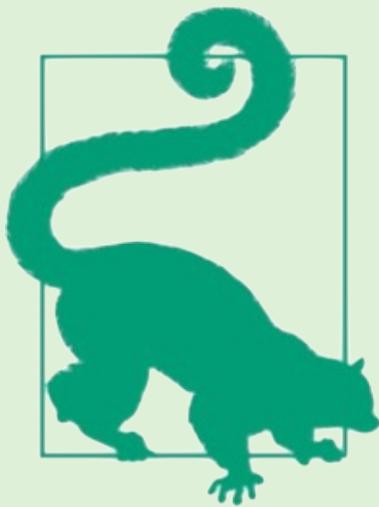


图4-2. 一个数据库引擎的实例可以托管多个schema，schema之间是逻辑隔离的

由于本章主要关注逻辑数据库的概念，在逻辑数据库的背景下，通常使用术语“database”来有效的关联一个逻辑上隔离的schema。因此，在本章中，我仍将使用“database”来关联逻辑上隔离的“schema”。因此，在我说“database”的地方，要想到“逻辑上隔离的schema”。为了简洁起见，除非另有明确说明，否则我将在图表中忽略数据库引擎的概念。

值得注意的是，各种NoSQL数据库可能有、也可能没有类似的逻辑划分，尤其是对于云供应商提供的数据库。例如，在AWS上，DynamoDB仅具有表的概念，其使用基于角色的访问控制来限制谁可以查看或修改数据。如何在这种情况下考虑逻辑划分，会是一件具有挑战的事情。



对于当前系统而言，我们将会遇到看起来无法立即解决的问题。与团队的其他成员一起来解决该问题，以便每个人都可以达成共识：这是一个你想解决的问题，即使你现在还不知道怎么解决。然后确保，至少从现在开始，做正确的事。随着时间的流逝，一旦掌握了一些新的技能和经验，最初看似无法解决的问题将变得更容易处理。

Copyrights © wangwei all right reserved

# 数据库视图

有时候，我们希望让多个服务使用一个数据源。在这种情况下，可以使用视图来降低耦合。利用视图，可以为服务提供一个schema，该schema是对它底层的schema的有限投影（*limited projection*）。这种投影可能会限制服务对数据的可见性，从而隐藏服务不应该访问的信息。

## 数据库视图

视图是从一个或几个基本表（或视图）中导出的虚拟的表。在系统的数据字典中仅存放了视图的定义，不存放视图对应的数据。视图是原始数据库数据的一种变换，是查看表中数据的另外一种方式。

可以将视图看成是一个移动的窗口，通过它可以看到感兴趣的数据。视图是从一个或多个实际表中获得的，这些表的数据存放在数据库中。那些用于产生视图的表叫做该视图的基表。一个视图也可以从另一个视图中产生。

视图的定义存在数据库中，与此定义相关的数据并没有再存一份于数据库中。通过视图看到的数据存放在基表中。

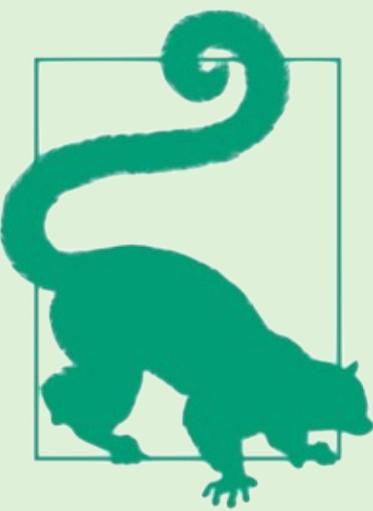
视图看上去非常像数据库的物理表，对它的操作同任何其它的表一样。当通过视图修改数据时，实际上是在改变基表中的数据。相反地，基表数据的改变也会自动反映在由基表产生的视图中。

由于逻辑上的原因，有些视图可以修改对应的基表，而有些则仅用于查询使用。

# 作为公共契约的数据库

在[第3章](#)中，我讨论了帮助一家现在已经倒闭的投资银行对其现有的信用衍生品系统更换平台的经历。我们解决了数据库严重耦合的问题：我们需要提高系统的吞吐量，以便为使用该系统的交易者提供更快的反馈。经过一番分析，我们发现系统的瓶颈是对数据库的写操作。经过短暂的调整后，我们意识到，如果我们重新组织schema，我们可以大大提高系统的写入性能。

正是在这一点上，我们发现有多个不在我们控制范围之内的应用会对数据库进行读请求，有时也会是读/写请求。不幸的是，我们发现所有这些外部系统都使用相同的用户名和密码，因此无法了解这些用户是谁，他们正在访问什么。我们估计，这涉及到“超过20个”应用，但是，这是根据对入站调用的一些基本分析得出的结论<sup>1</sup>。



如果每个参与者（例如，一个人或一个外部系统）拥有一组不同的凭据，则限制特定角色的访问会更加容易，减少撤销凭据和轮换凭据的影响，并更好地了解每个参与者在做什么。管理不同的凭据集可能很麻

烦，尤其是在微服务系统中，可以需要多个凭据集来管理每一个服务。我喜欢使用专门的secret stores来解决此问题。HashiCorp的Vault是该领域的出色工具。Vault可以为诸如数据库之类的短暂且范围有限的访问生成每个角色的凭证。

我们不知道这些用户是谁，但我们必须与他们联系。最终，有人想到禁用他们正在使用的共享帐户，然后等待他们与我们联系以进行投诉。显然，这不是解决我们最初不应该遇到的问题的好方法，但是这个方法是有效的。然而，我们很快就意识到，这些应用程序中的大多数都不再进行主动维护（*active maintenance*），这意味着它们不会为新的schema设计而升级<sup>2</sup>。实际上，我们的数据库schema已成为不可修改的、面向公众的契约：我们必须继续维持该schema结构。

# 视图的呈现

我们的解决方案是首先解决外部系统对schema执行写操作的场景。幸运的是，在我们的case中，这种情况很容易解决。对于所有的读操作的客户端，我们创建了一个专门的视图schema，该schema看起来和旧schema一样，然后让客户端访问该新的schema，如图4-3所示。只要可以保持视图不变，我们就可以修改自己的schema。让我们仅讨论那些涉及很多存储过程的场景。

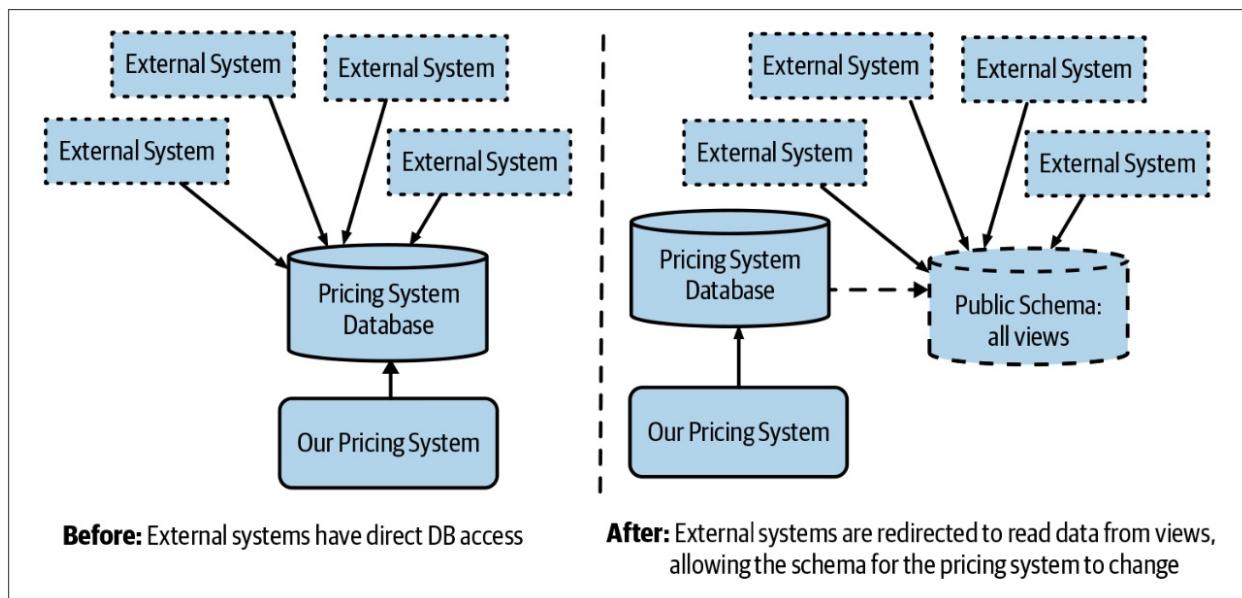


图4-3. 使用试图以允许修改其底层的schema

在投资银行的例子中，最终，视图和其底层的schema的差异非常大。当然，我们可以更简单地使用视图，也可以隐藏我们不希望外界看到的信息。举一个简单的例子，在图4-4中，我们的loyalty服务仅提供系统中的会员卡列表。目前，会员卡信息以customer表的列的形式存储在customer表。

因此，我们定义了一个视图，该视图仅开放表中的customer ID和loyalty ID的映射，而不开放customer表中的其他任何信息。同样，单体数据库中的其他的任何表对loyalty服务是完全隐藏的。

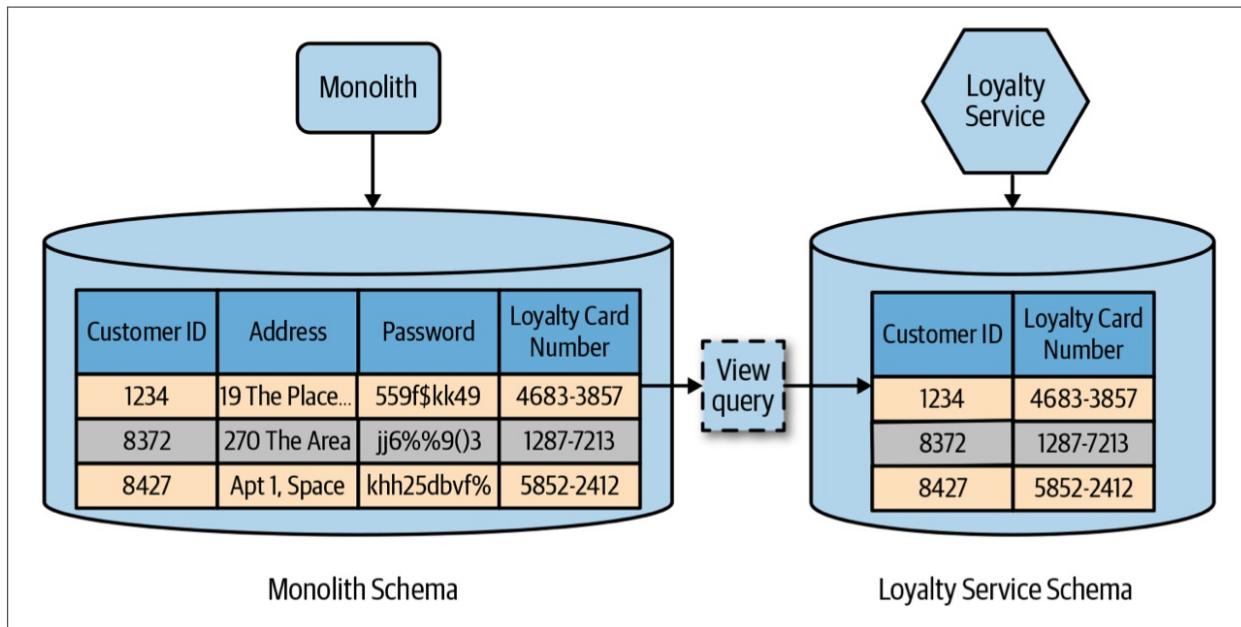


图4-4. 数据库试图投影其底层schema的子集

视图能够从底层的源schema中投影有限信息的能力使我们能够实现某种形式的信息隐藏。视图让我们可以控制信息的共享和隐藏。视图不是一个完美的解决方案，该方法存在一些限制。

根据数据库的特性，可以选择创建一个materialized view（物化视图）。对于materialized view，通常使用缓存来预先计算视图。这意味着从视图中读取数据时无需读取其底层schema，从而可以提高性能。然后，需要围绕如何更新这个预计算的视图来进行权衡和折衷，materialized view很可能意味着我们可能正在从视图中读取“脏”数据。

# 视图的限制

视图的实现方式可能有所不同，但是视图通常是查询的结果。这意味着视图本身是只读的。这直接限制了视图的用途。另外，尽管视图是关系数据库的常见功能，而且许多更成熟的NoSQL数据库都支持视图（例如，Cassandra和Mongo都支持视图），但并非所有的数据库都支持视图。即使我们的数据库引擎确实支持视图，也可能会有其他限制，例如，源schema和视图都必须位于同一数据库引擎中。这种限制可能会增加物理部署的耦合，从而导致潜在的单点故障。

# 视图的所有制

值得注意的是，底层的源schema的修改可能需要视图的更新，因此应该仔细考虑谁“拥有”该视图。我建议将任何已发布的数据库视图与其他任何服务接口等同看待，因此，团队应该让视图随源schema而保持最新状态。

# 何处可以使用视图

当我认为无法拆分现有的单体schema时，我通常会使用数据库视图。理想情况下，如果最终目标是通过服务接口公开信息，则应尽量避免使用视图。相反，最好进行适当的schema分解。值得关注视图技术的局限性。但是，如果认为完全拆分schema的工作量太大，那么视图可能是朝着正确方向迈出的一步。

---

<sup>1</sup>. When you're relying on network analysis to determine who is using your database, you're in trouble. ↪

<sup>2</sup>. It was rumored that one of the systems using our database was a Python-based neural net that no one understood but “just worked.” ↪

Copyrights © wangwei all right reserved

# 把数据库包装成服务

有时候，当某些事情太难处理时，隐藏这些乱糟糟的事情是有意义的。使用把数据库包装成服务的模式（*database wrapping service pattern*），我们可以做到：将数据库隐藏在某个服务之后，该服务作为数据库的一层薄薄的包装，从而将数据库依赖转变为服务依赖，如图4-5所示。

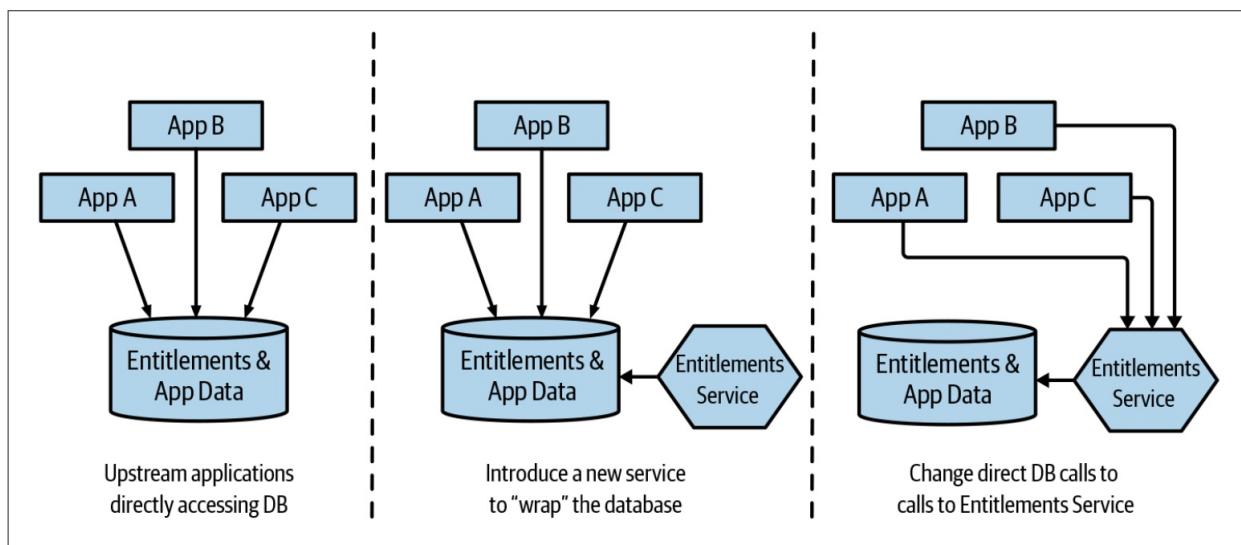


图4-5. 使用一个服务来包装数据库

几年前，我曾在澳大利亚一家大型银行短暂的工作，以帮助其中的一个部门实施产品的的改进路径。第一天，我们与关键人物进行了一些访谈，以了解他们面临的挑战，并建立了当前流程的概况。会议期间，有人跑上前来并介绍说他们是公司中该领域的DBA主管。他说：“请阻止他们把东西放到数据库中！”

我们喝了一杯咖啡，然后DBA阐述了问题。在大约30年的时间里，作为该组织的“皇冠上的明珠”，一个商业银行系统已初具规模。该系统最重要的部分之一是管理他们所谓的“权利(*entitlements*)”。在商业银行中，管理哪些

个人可以访问哪些帐户并确定这些帐户可以进行的操作非常复杂。为了让大家可以大致了解这些“权利”，我们来考虑一名可以查看公司A、B、C的账户的会计员。对于公司B而言，帐户之间的转账限额为\$500；对于公司C，账户之间的转账没有限额，但最多可以提取\$250。这些“权利”的维护和应用几乎完全在数据库的存储过程中进行管理。所有的数据访问都通过此“权利”逻辑来控制。

随着银行规模的扩大，以及其逻辑和状态数量的增长，数据库开始无力承担负载。“我们已经将所有可能的钱都给了Oracle，但还远远不够。”令人担忧的是，考虑到预期的增长，即便算上硬件性能的提高，该组织的需求最终也会超出数据库的能力。

在进一步探讨该问题时，我们讨论了拆分数据库以减少负载的想法。问题是，这些错综复杂的核心在于这个权利系统。试图理清这团逻辑将是一场噩梦，并且在这一领域犯错会带来巨大的风险：搞错一步，就可能会阻止某人访问其帐户，或者更糟的是，某人可能会获得权限来访问他们不应该有权限的财产。

我们提出了一个计划来尝试解决这种情况。我们相信，短期内我们无法修改权利系统，因此，当务之急是至少要使问题不能变得更糟糕。因此，我们需要阻止人们把更多数据和行为放入entitlements数据库中。一旦一切就绪，我们就可以考虑移除entitlements数据库中那些比较容易提取的部分，以希望把数据库的负载降低到可以减轻对系统长期生存能力的担忧。这将为下一步的思考提供一些喘息的空间。

我们讨论了引入新的Entitlements服务，该服务使我们能够“隐藏”有问题的schema。最初，Entitlements服务的行为很少，因为当前数据库已经以存储过程的形式实现了很多行为。但是我们的目标是鼓励编写其他应用的团队将entitlements数据库视为其他人的应用，并鼓励他们在本地存储自己的数据，如图4-6所示。

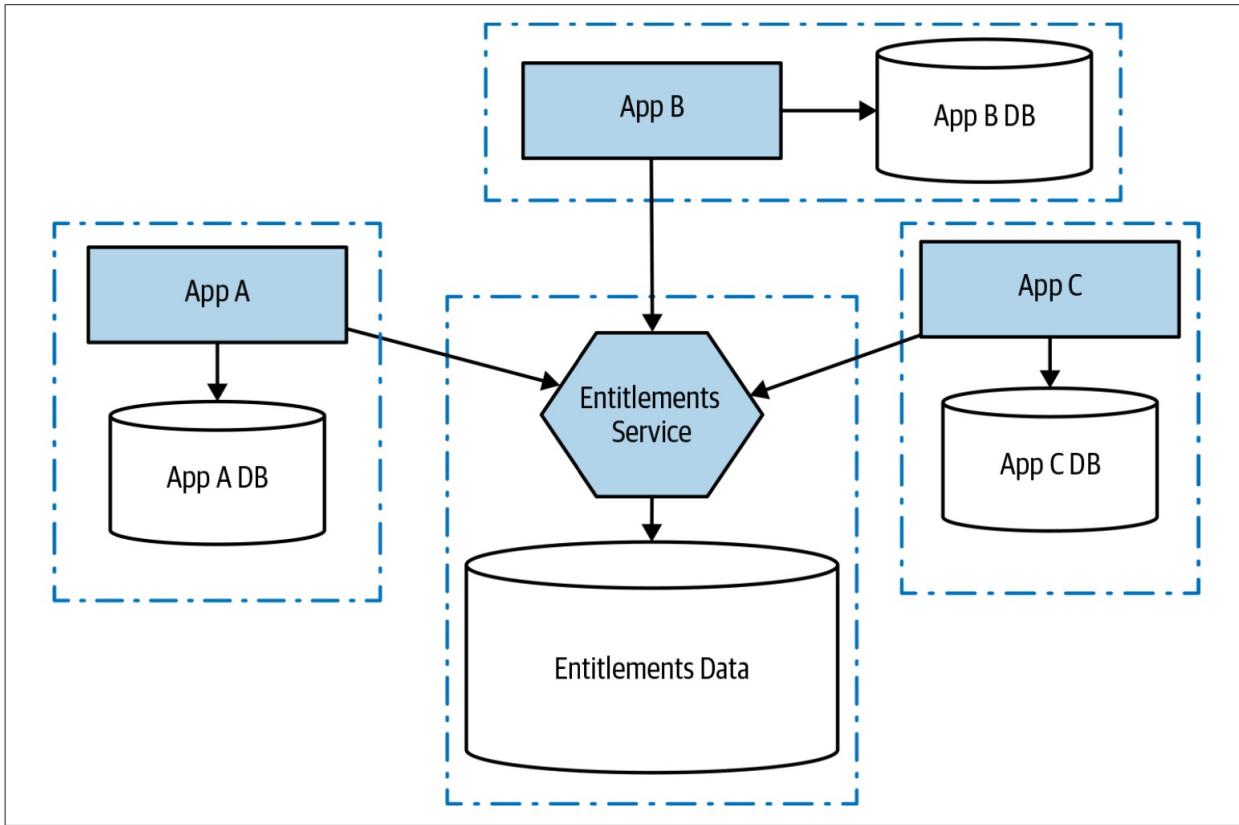


图4-6. 使用把数据库包装成服务的模式来减少对中央数据库的依赖

就像使用数据库视图一样，把数据库包装成服务允许我们可以控制要共享的内容以及要隐藏的内容。这种方法为其下游的消费者提供了一个固定的接口，并且变化发生在接口幕后的数据库上，以此来改善数据依赖的场景。

# 何处使用该模式

该模式在底层数据schema很难拆分的情况下非常有效。通过在schema周围放置一个显式包装器，并明确只能通过该schema来访问数据，这至少可以阻止数据库的进一步增长。数据库的包装服务清楚地描述了什么是“我们的”以及什么是“其他人的”。我认为，当我们把底层的数据schema和服务层的所有权调整到同一团队时，这种方法也最有效。服务API需要适当的包含在托管接口中，并且对该API层的修改方式进行适当的监督。这种方法对上游应用程序也有好处，因为它们可以更轻松地了解它们如何使用下游的schema。这使得那些针对测试而进行的打桩之类的行为更加易于管理。

与使用简单的数据库视图相比，该模式更具优势。首先，我们不必呈现可以映射到现有表结构的视图；我们可以在包装服务中编写代码，以对基础数据进行更复杂的投影。包装服务还可以通过API调用执行写操作。当然，采用这种模式确实需要上游消费者做出改变，上游消费者必须从直接的数据库访问转移到API调用。

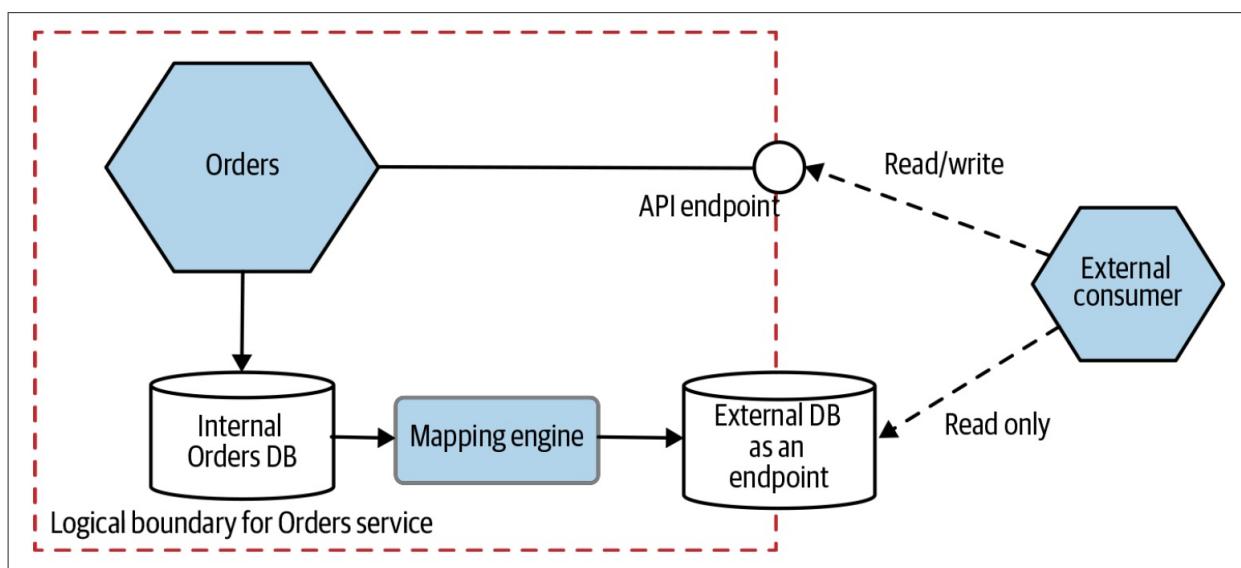
理想情况下，使用此模式是进行更根本的修改的垫脚石，使我们有时间拆分API层之下的schema。可以说，我们只是在解决问题，而不是解决根本问题。尽管如此，本着不断改进的精神，我认为这种模式与此密切相关。

Copyrights © wangwei all right reserved

# 把数据库视为服务接口

有时，客户端（clients）只需要一个用于查询的数据库。可能是因为他们需要查询来获取大量数据，或者可能是由于外部各方已经在使用需要SQL的工具链（例如Tableau之类的工具，这些工具通常用于对业务指标的深入分析）。在这些情况下，允许客户端查看服务在数据库中管理的数据是很有意义的，但是我们应注意将公开的数据库与服务边界内使用的数据库分开。

我见过的一种有效的方法是创建一个专用数据库，该数据库设计为只读的端点而公开，并在底层数据库中的数据变更时修改此专用数据库。实际上，服务可以将事件流作为一个端点而公开，并且将同步API作为另一个端点，利用这种方式也可以将数据库开放给外部使用者。在图4-7中，我们看到了Orders服务的一个例子：通过API开放了一个读/写端点，并以一个只读接口的形式开放了一个数据库。映射引擎在获取内部数据库的修改，并确定需要在外部数据库中进行什么变更。



## 图4-7. 开放专用数据库为端点，以使内部数据库保持隐藏状态

### 数据库报表模式 (Reporting Database Pattern)

Martin Fowler已经在[Reporting Database](#)中对本节的方法进行了描述，那么为什么我在这里还要使用另一个名称？当我与更多人交谈时，我意识到，尽管报表是这种模式的常见应用，但这并不是人们使用该技术的唯一原因。与传统的面向批处理的工作流相比，允许客户端定义即席查询 (*ad hoc queries*) 的功能具有更广泛的使用。因此，尽管该模式可能广泛应用于支持这些报表的case，但我想用一个不同的名称来反映这一事实，即该技术可能具有更广泛的适用性。

映射引擎可以完全忽略内部数据库的变更，也可以将内部数据库的变更直接开放出来，也可以介于二者之间。关键是：映射引擎充当内部数据库和外部数据库之间的抽象层。当内部数据库的结构修改时，映射引擎需要修改以确保对外的数据库与内部数据库保持一致。在几乎所有的情况下，映射引擎都会落后于内部数据库的写操作。通常，映射引擎的具体实现决定了才延迟的大小。从公开数据库中读取数据的客户端需要认识到他们可能会读到脏数据。因此，我们可能会发现，以编程方式公开外部数据库的最新更新时间是合适的。

# 实现一个映射引擎

此处的细节在于确定如何更新，即如何实现映射引擎。

- 我们已经研究了CDC系统，CDC是一个非常好的选择。事实上，CDC可能是最可靠的解决方案，同时还提供了最为及时的数据更新。
- 另一个选择是使用批处理来复制数据，但是这种方案可能会存在问题：因为该方案可能会增加内部数据库和外部数据库之间的数据延迟；并且对于某些数据schema而言，很难确定需要复制哪些数据。
- 第三种选择是从相关服务来监听触发事件，并使用监听事件来更新外部数据库。

之前，我会使用批处理来处理此问题。但是，如今，我可能会使用专门的CDC系统，例如Debezium。批处理程序无法运行或者无法长时间运行已经让我多次受到伤害。随着世界不再使用批处理作业，并且希望更快地获取数据，批处理的方案正在被实时解决方案而取代。使用CDC系统来处理数据变更很有意义，尤其是如果我们正在考虑使用CDC来开放服务边界之外的事件时，尤其如此。

# 和数据库视图的对比

把数据库视为服务接口的模式比简单的数据库视图更为复杂。数据库视图通常与特定的技术栈绑定：如果我想提供Oracle数据库的视图，则视图的底层数据库和托管视图的架构都要运行在Oracle上。通过本节的方法，我们公开的数据库可以是完全不同的技术栈。我们可以在服务内部使用Cassandra，但可以用传统的SQL数据库作为开放的端点。

把数据库视为服务接口的模式比数据库视图具有更强大的灵活性，但是需要更多的成本。如果下游消费者的需求可以通过简单的数据库视图来满足，那么一开始实现的工作量可能会减少。请注意，数据库视图可能会限制接口的演变方式。我们可以从使用数据库视图开始，然后再考虑转移到专用的报表数据库。

# 何处使用该模式

显然，因为作为端点而开放的数据库是只读的，因此，该方案仅对需要只读访问权限的客户端有用。该方法非常适合于报表的case，此时，我们的客户可能需要对给定服务所拥有的大量数据进行合并。可以扩展该方法，然后将数据库的数据导入到更大的数据仓库中，从而可以查询来自多个服务的数据。我在《Building Microservices》的第5章对此进行了更详细的讨论。

不要低估确保此外部数据库投影的时效性所需的工作。根据我们当前服务的实现方式，这可能是一项复杂的工作。

Copyrights © wangwei all right reserved

# 改变数据的所有权

到目前为止，我们还没有真正解决根本问题。我们只是用各种不同的绷带  
来包扎一个大型的、共享的数据库。在开始考虑从巨型的单体数据库中抽  
取数据这一棘手的任务之前，我们需要考虑我们所讨论的数据实际上应该  
存放在何处。当我们把服务从单体中拆分出来时，某些数据应该随之一起  
提供，而有些数据则应该保留在原处。

- 如果我们接受这样的想法——[微服务封装一个或多个聚合相关联的逻辑](#)，那么我们还需要把聚合的状态管理以及与之关联的数据移到微服  
务自己的数据schema中。
- 另一方面，如果我们的新的微服务需要与仍在单体中的聚合交互，则  
需要通过定义明确的接口来开放此功能。

现在，让我们看看如上的两种选择。

# 在单体中开放聚合

在图4-8中，我们的新服务——Invocing需要访问与管理发票没有直接关系的各种信息。至少，Invocing服务需要当前的Employees聚合的相关信息来管理审批工作。而目前，Employees的相关数据全部位于单体数据库中。在单体上，通过服务端点（可能是API或事件流）开放Employees聚合的相关信息，我们明确了Invoice服务所需要的信息。

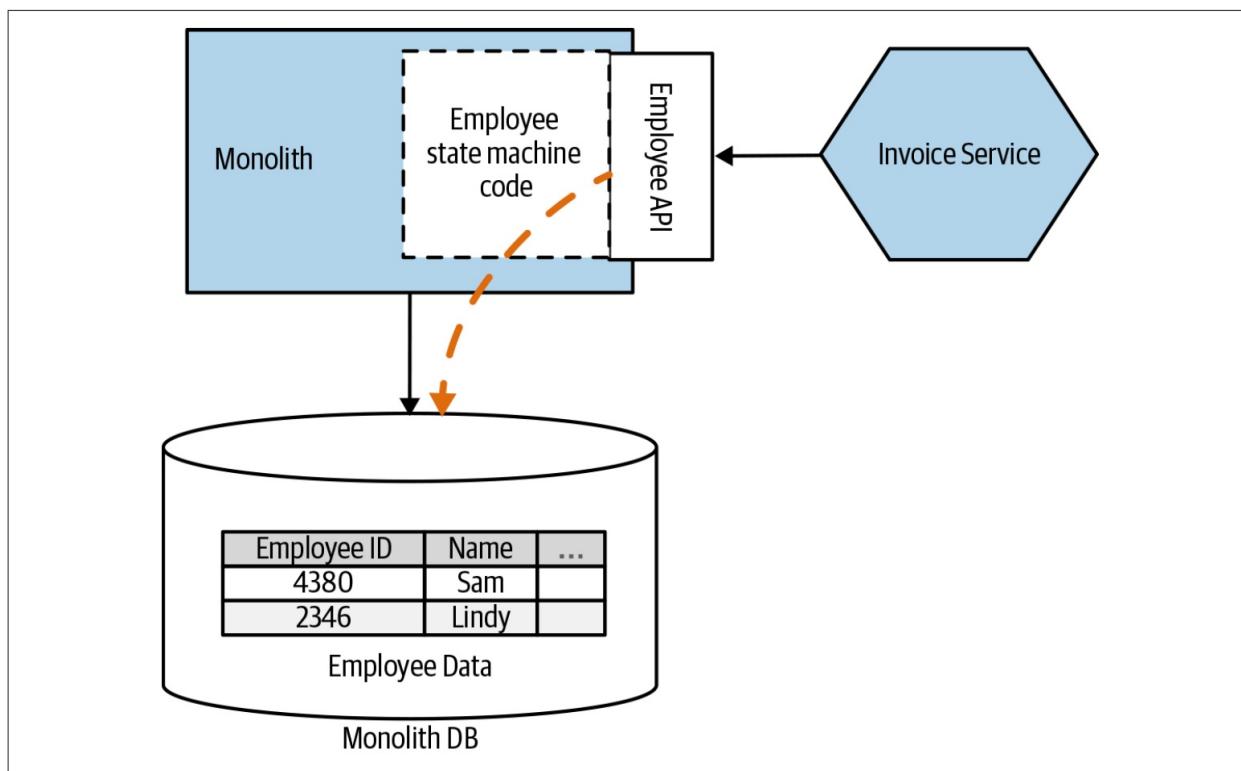


图4-8. 通过适当的服务接口开放单体的信息，从而使我们的新微服务可以访问这些信息

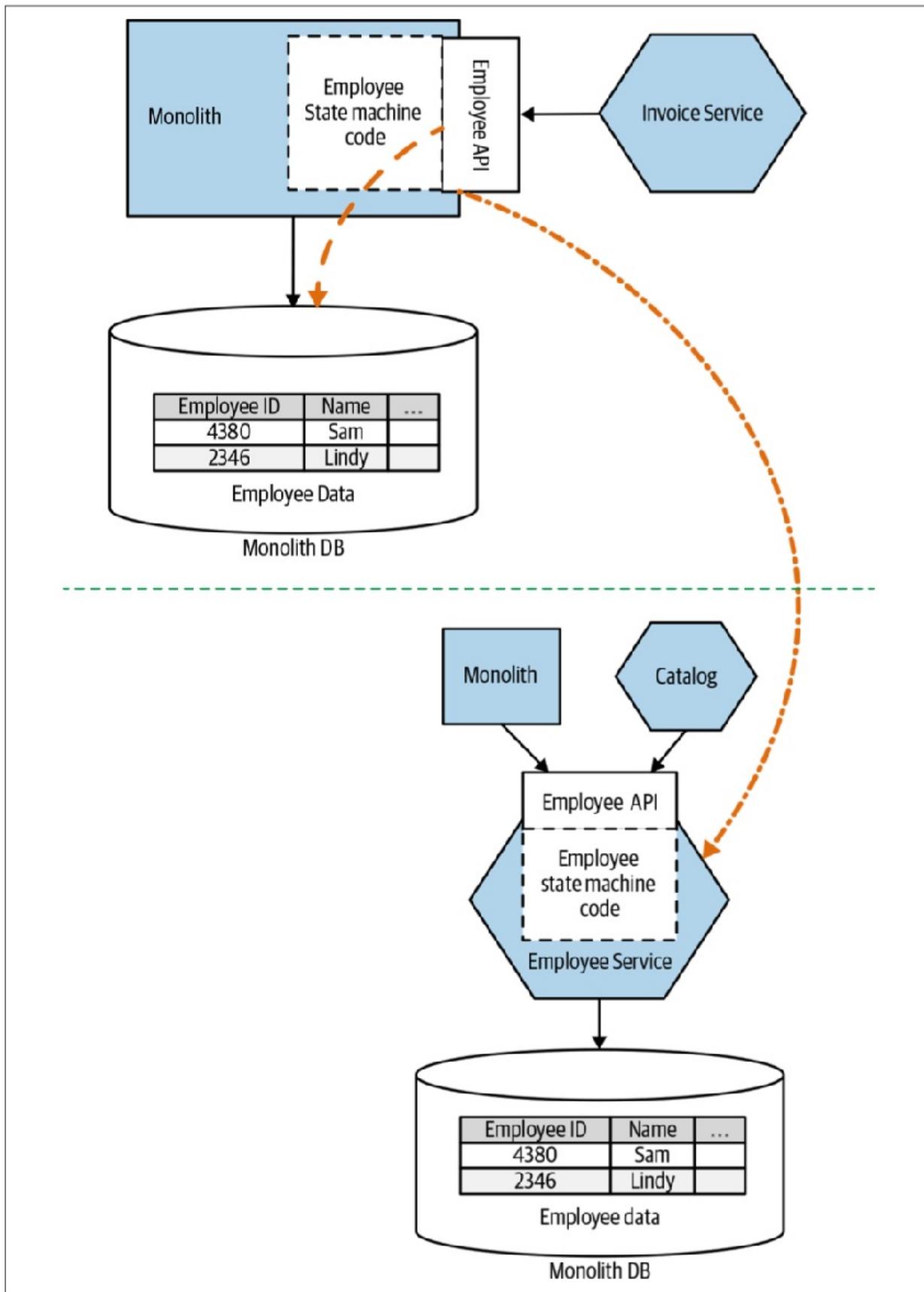
我们想要把微服务视为行为（*behaviro*）和状态（*state*）的组合。我已经讨论过这样的想法：把微服务视为包含一个或多个用来管理领域聚合（*domain aggregates*）的状态机。当从单体中暴露聚合时，我们希望以相

同的方式来思考。单体仍然“拥有”如下的概念：什么样的状态修改是允许的，什么样的状态修改是不允许的。我们不想仅仅将其视为数据库的包装器。

除了公开数据之外，我们还将公开如下的操作：允许外部各方查询聚合的当前状态，允许外部各方发起新状态转换的请求。我们仍然可以决定从服务边界公开聚合的哪些状态，并限制可以从外部请求什么状态转换操作。

## 抽取更多服务的途径

通过定义Invoice服务的需求，并在定义明确的接口中显式公开所需的信息，我们将有可能发现未来的服务边界。在[图4-9](#)中，接下来，显而易见的步骤可能是提取Employee服务。通过一个API来开放与员工相关的数据，我们已经摸索了很久来了解消费者对新的Employee服务的需求。



## 图4-9. 使用现有的端点来驱动抽取新的Employee服务

当然，如果我们确实从单体中提取了那些employees聚合，而单体需要那些employee数据，则可能需要修改单体才能使用Employee新服务！

## 何处使用该模式

当仍然由数据库“拥有”所要访问的数据时，此模式可以很好地使新服务可以获得所需的访问权限。抽取服务时，新服务回调单体来访问其所需的数据可能比直接访问单体的数据库的方式要多做些工作，但从长远来看，这是一个更好的主意。我仅在无法修改所涉及到的单体以公开这些新端点的情况下才会考虑使用数据库视图。在无法修改单体的情况下，可以使用单体数据库的数据库视图，也可以使用前面讨论的CDC模式（具体参见第120页的[CDC模式](#)），或创建一个专用的数据库包装服务（参阅第132页的[把数据库包装成服务](#)），以开放我们所需的Employee信息。

# 修改数据的所有权

我们已经研究了：当Invoice服务需要访问其他的功能所拥有的数据（如前所述，我们需要访问Employee数据）时，会发生什么。但是，当我们考虑当前位于单体中的数据应该由我们新抽取的服务来控制时，会发生什么？

在图4-10中，我们概述了需要进行的修改。因为应该由Invoice来管理数据的生命周期，因此我们需要把与Invoice相关的数据从单体中移出，并移至新的Invoice服务中。然后，我们需要修改单体，以将Invoice服务视为与发票相关的真实来源；并修改单体，以使其调用Invoice服务端点以读取数据或请求数据修改。

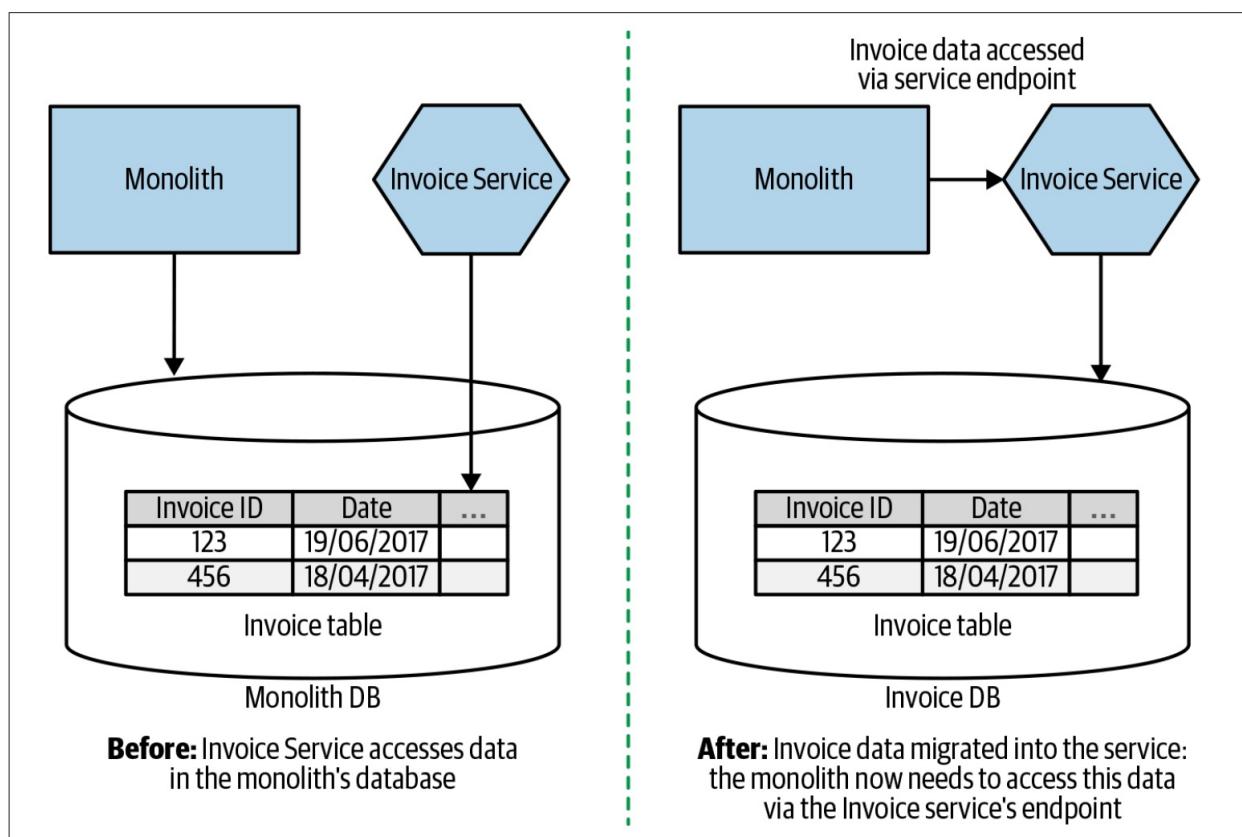


图4-10. 新的Invoice服务拥有相关数据的所有权

但是，从现有的单体数据库中剥离发票数据可能是一个复杂的问题。我们可能必须考虑破坏外键约束、破坏事务边界等等的影响，所有的这些主题，我们都将在本章的后面来介绍。如果可以修改单体，以便单体仅需要读取与发票相关的数据，则可以考虑从发票服务的数据库中投影视图，如图4-11所示。但是，此时，数据库视图的所有限制也都将同时生效。因此，强烈建议修改单体，以直接调用新的Invoice服务。

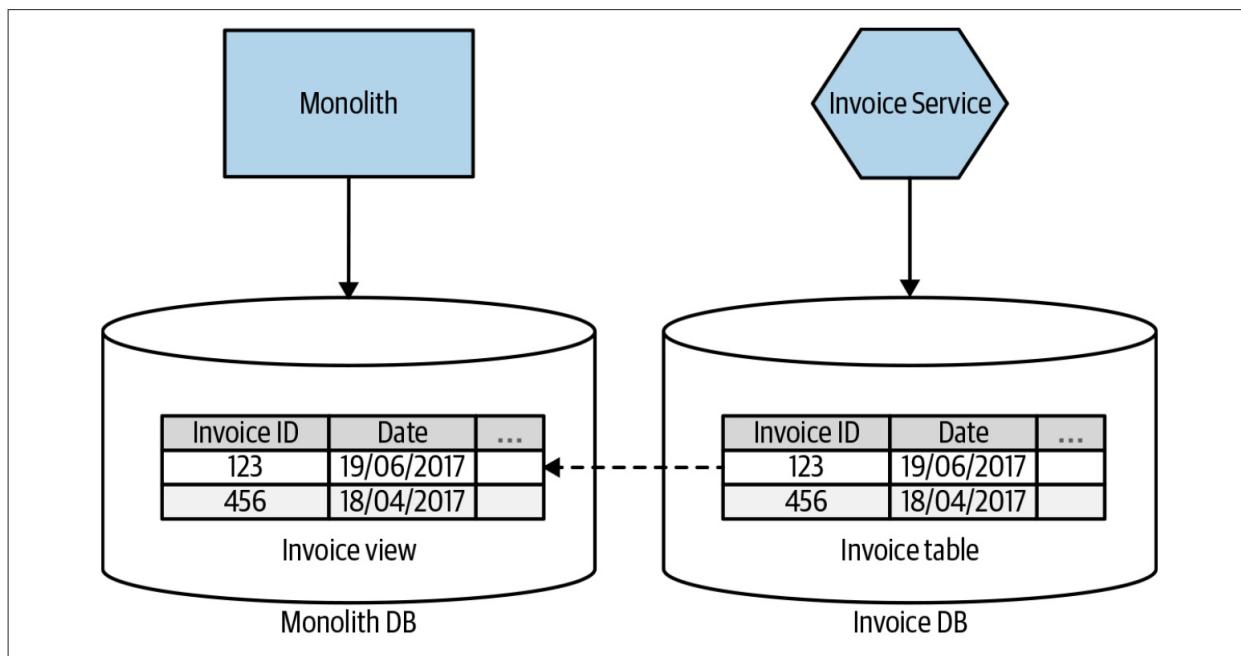


图4-11. 把Invoice数据作为视图投影回单体

## 何处使用该模式

何处使用该模式更加明确。如果新抽取的服务封装了修改某些数据的业务逻辑，则这些数据应在新服务的控制之下。数据应从从来的地方移到新服务中。当然，将数据移出现有数据库的过程并非是一个简单的过程。实际上，将数据移除现有数据库将是本章其余部分的重点。

# 数据同步

正如我们在[第3章](#)中讨论的那样，像如**绞杀者模式**之类的方法的好处之一是：当我们切换到新服务时，如果有问题，我们可以再切换回旧的服务。当所讨论的服务所管理的数据需要在单体和新服务之间保持同步时，就会出现问题。

在[图4-12](#)中，我们会看到这样的一个例子。我们正在切换到新的Invoice服务。但是，新服务以及现有的、单体中的等效代码同样在管理此数据。为了维护在两种实现之间进行切换的能力，我们需要确保两组代码都可以看到相同的数据，并且可以用一致的方式来维护这些数据。

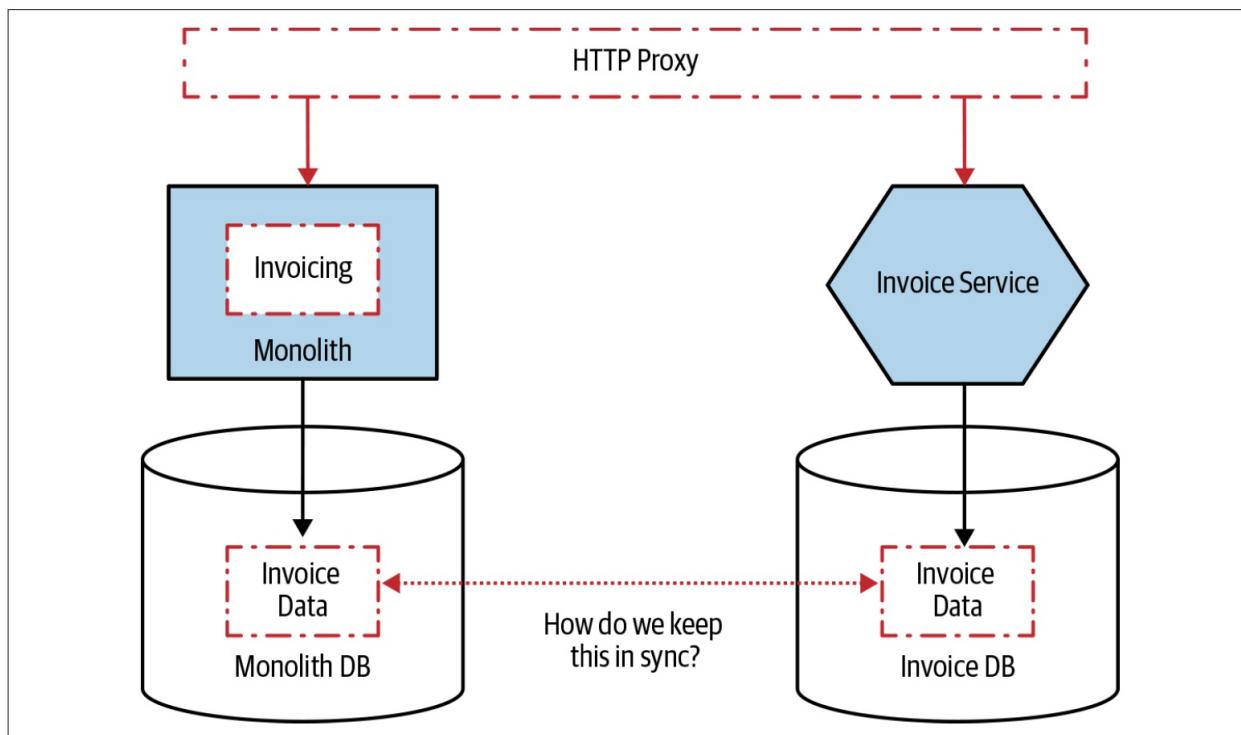


图4-12. 我们想使用绞杀者模式迁移到新的Invoice服务，但是该服务会管理状态

那么，此时我们有什么方案呢？首先，我们需要考虑数据之间所需要的数据一致性的程度。如果任何一组代码都需要始终看到完全一致的发票数据，那么最直接的方法之一就是：确保把数据保存在一个地方。把数据保存在一个地方可能会使我们倾向于让新的Invoice服务短期内直接从单体中读取数据；或者使用视图，正如我们在第128页的[数据库视图](#)的部分中所探讨的那样。一旦我们认为服务切换是成功的，此时我们就可以迁移数据，正如我们先前在第141页的[改变数据的所有权](#)一节中所讨论的那样。但是，不要夸大使用共享数据库的担忧：应该仅将其作为一种短期措施，作为更完整的抽取服务的一部分；长时间的使用共享数据库会导致长期的痛苦。

如果我们要进行一次大规模（big-bang）的服务切换（我会尽量避免这种情况），同时迁移应用程序代码和数据，则可以在切换到新的微服务之前使用批处理程序来复制数据。把与发票相关的数据复制到我们的新微服务之后，新服务就可以开始提供服务。但是，如果我们需要回滚到使用现有的单体系统中的功能时，会发生什么？在微服务架构中修改的数据不会反应在单体数据库的状态上，因此我们最终可能会丢失状态。

另一种方法是考虑利用代码使两个数据库保持同步。因此，我们会让单体或新的Invoice服务同时对这两个数据库进行写操作。这会涉及一些更仔细的思考。

虽然可以在代码层面来执行数据库的双写，但是，我觉得，除非万不得已，不要再代码层面进行数据库的双写。

# 在应用中同步数据

即使在最简单的情况下，将数据从一个位置切换到另一位置也可能是一项复杂的工作。然而，数据越有价值，则迁移的复杂度则越高。当我们开始思考迁移病历数据时，仔细考虑如何迁移数据就显得尤为重要。

几年前，咨询公司Trifork参与了一个项目，以帮助存储丹麦公民的病历数据<sup>3</sup>。该系统的初始版本已将数据存储在MySQL数据库中，但随着时间的推移，人们逐渐意识到，对于系统将会面临的挑战而言，MySQL数据库将不再合适。人们决定使用另一种数据库：Riak。人们希望Riak数据库可以使系统得到更好地扩展以处理预期的负载，同时还可以提供改善的弹性（resiliency）特性。

现有系统将数据存储在一个数据库中，但是系统可以脱机的时间是有限制的，并且不丢失数据至关重要。因此，需要一种解决方案，该解决方案允许公司把数据迁移至新的数据库，同时还需要具备验证迁移的机制，并在此过程中具有快速回滚的机制。

人们决定让应用程序本身来执行两个数据源之间的数据同步。这个想法是：最初，现有的MySQL数据库仍然是数据源，但是在一段时间内，应用程序将确保MySQL和Riak中的数据保持同步。一段时间之后，Riak将在MySQL下线之前成为该应用程序的数据源。让我们更详细地看一下这个过程。

# Step 1. 批量同步数据

第一步是在新数据库中拥有数据的副本。对于病历项目而言，这涉及将数据从旧系统批量迁移到新的Riak数据库。在进行批量导入时，现有系统保持运行状态，因此，导入的数据源是从现有MySQL系统获取的数据快照，如图4-13。这会带来一种挑战，因为当批量导入完成时，源系统中的数据很可能已改变。然而，对于这个案例而言，让源系统脱机是不切实际的。

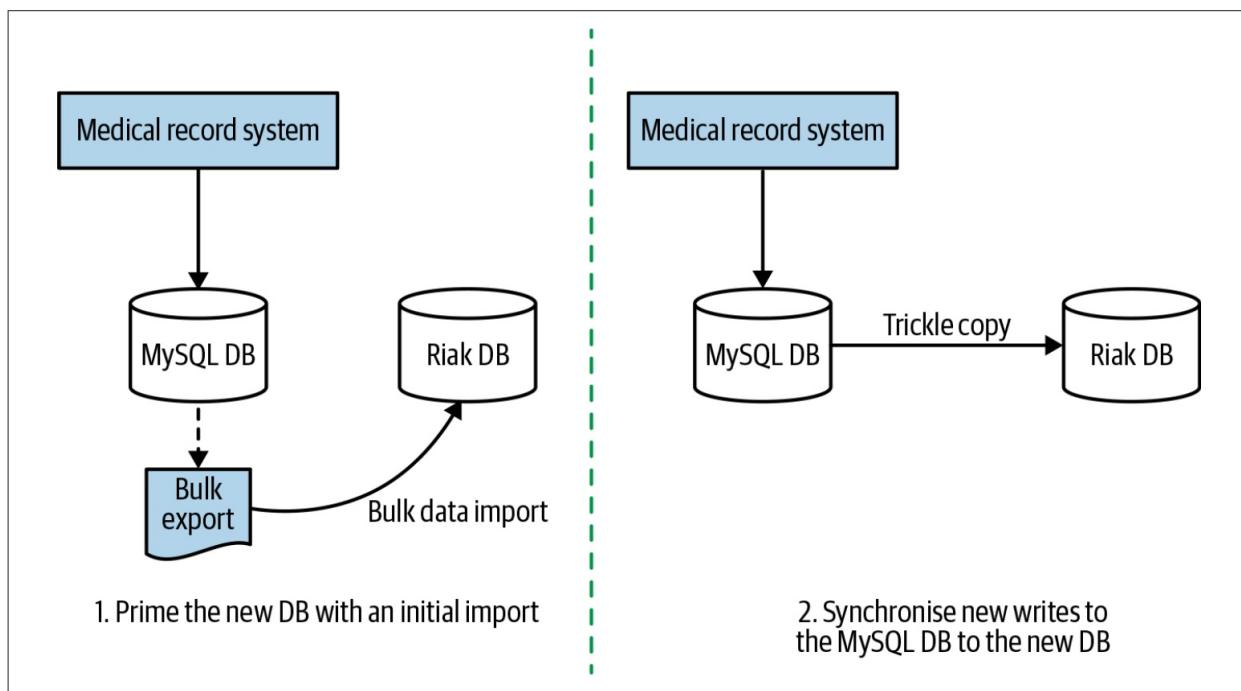


图4-13. 准备新的数据存储以进行基于应用程序的同步

批量导入完成后，执行CDC，从而把数据导入以后的数据修改应用到新数据库。这样可以使Riak与MySQL同步。一旦做到这一点，就该部署新版本的应用程序了。

## Step 2. 同步写操作，从老数据库读取数据

现在两个数据库都处于同步状态，因此部署该应用程序的新版本，该版本会将所有数据写入两个数据库，如图4-14所示。在此阶段，目标是确保应用程序正确地把数据写入两个数据库，并确保Riak的行为在可接受的公差（*acceptable tolerances*）范围之内。此时，仍然从MySQL读取所有数据，这确保了即使Riak出现问题，仍然可以从现有的MySQL数据库中检索数据。

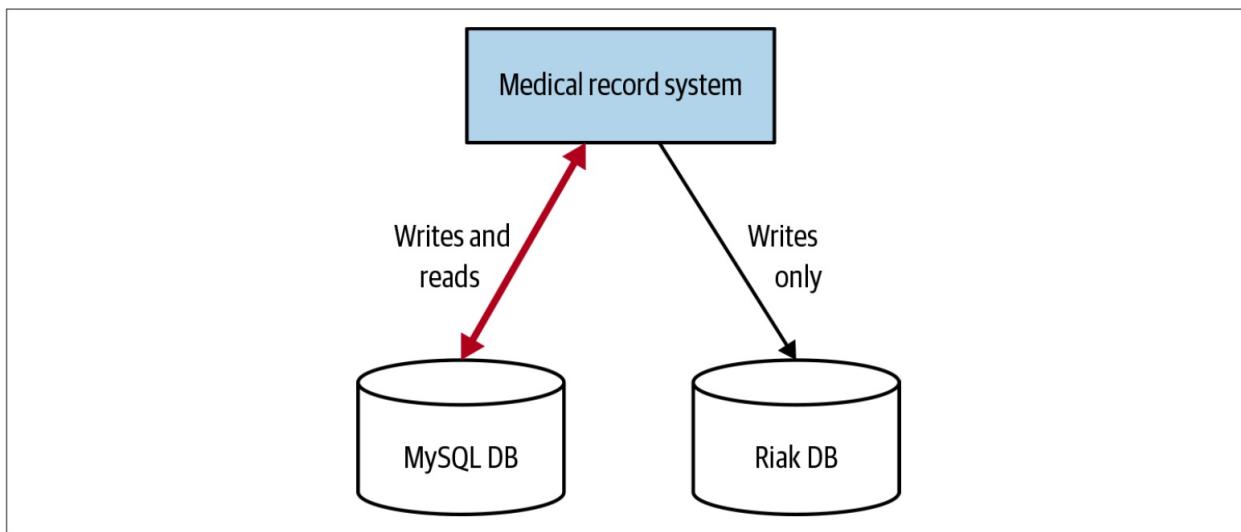


图4-14. 应用程序保持两个数据库的同步，但是只使用一个数据作为数据源  
只有对新的Riak系统建立起足够的信心之后，才能进行下一步。

## Step 3. 同步写操作，从新数据库读取数据

在此阶段，已验证对Riak的写操作是正常的。最后一步是确保从Riak读取数据也起作用。现在，对应用程序的简单修改使Riak成为数据源，如图4-15所示。请注意，我们仍然会同时写入两个数据库，因此，如果出现问题，我们会有一个回滚的选择。

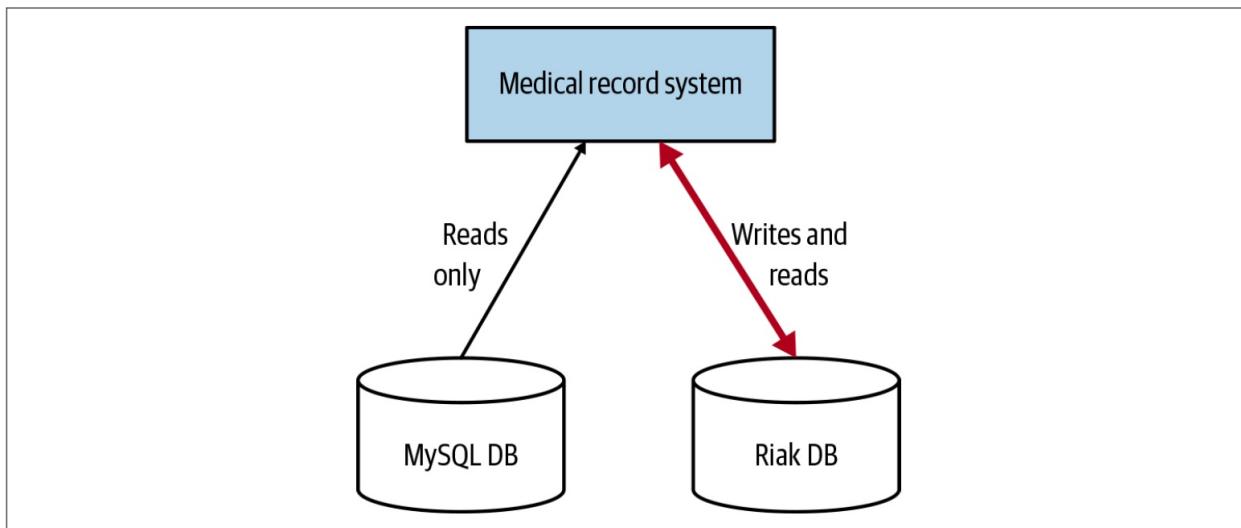


图4-15. 现在，新数据库是数据源，但是旧数据库仍保持同步

一旦新系统已经足够成熟，就可以安全地移除旧数据库。

# 何处使用该模式

在丹麦的病历系统中，我们只需要处理一个应用程序。但是，我们一直在谈论的是拆分微服务的场景。那么，这种模式真的有帮助吗？首先要考虑的是，如果想在拆分应用程序代码之前先拆分数据库，则此模式可能很有意义。在[图4-16](#)中，我们恰好看到了这种情况，我们首先复制了与发票相关的数据。

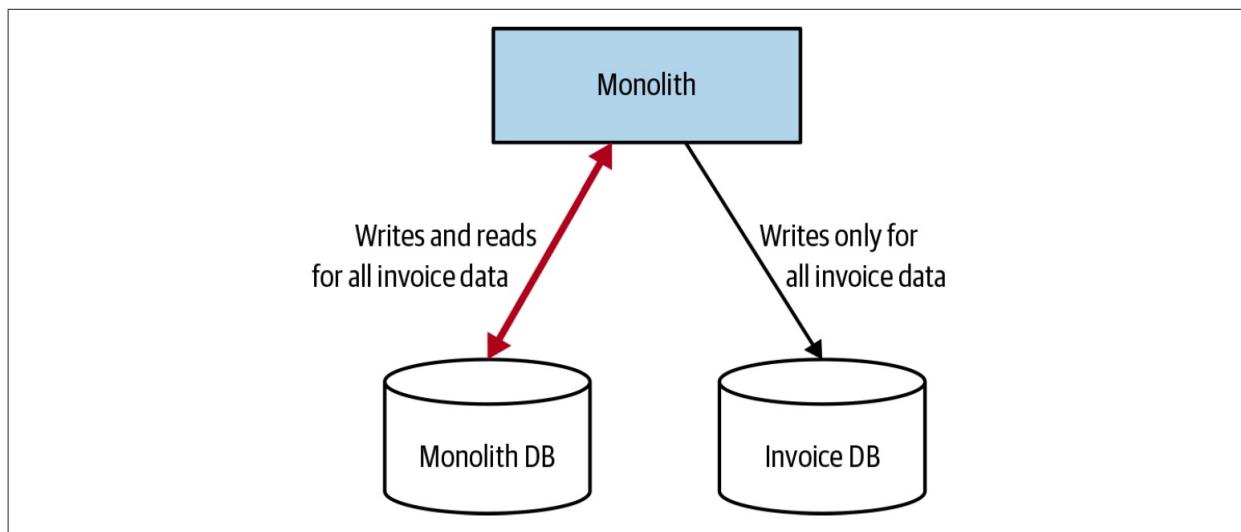


图4-16. 单体保持两个数据库同步的例子

如果实施正确的话，两个数据源应始终保持同步，从而在需要快速切换数据源以进行回滚等情况下为我们提供显着的好处。因为无法在任何时间让应用程序脱机，因此，在丹麦病历系统的例子中，使用此模式似乎很明智。

现在，可以考虑使用这种模式。在这种模式下，单体服务和微服务都可以访问数据，但这会让系统变得极其复杂。在[图4-17](#)中，我们就遇到了这种情况。单体和微服务都必须确保跨数据库的正确同步才能使这种模式起作用。

用。如果任何一方出了问题，我们可能会遇到麻烦。如果可以确保，在任何时间点，只有Invoice服务和单体中的Invoice功能的二者之一正在写数据库，则可以使用像如我们之前讨论过的绞杀者模式这种简单的切换技术来极大地减轻这种复杂性。但是，如果写操作可能由单体中的Invoice功能触发，也可能由新的Invoice服务触发（成为金丝雀发布的一部分），由此产生的同步将非常棘手，因此，我们可能不希望使用此模式。

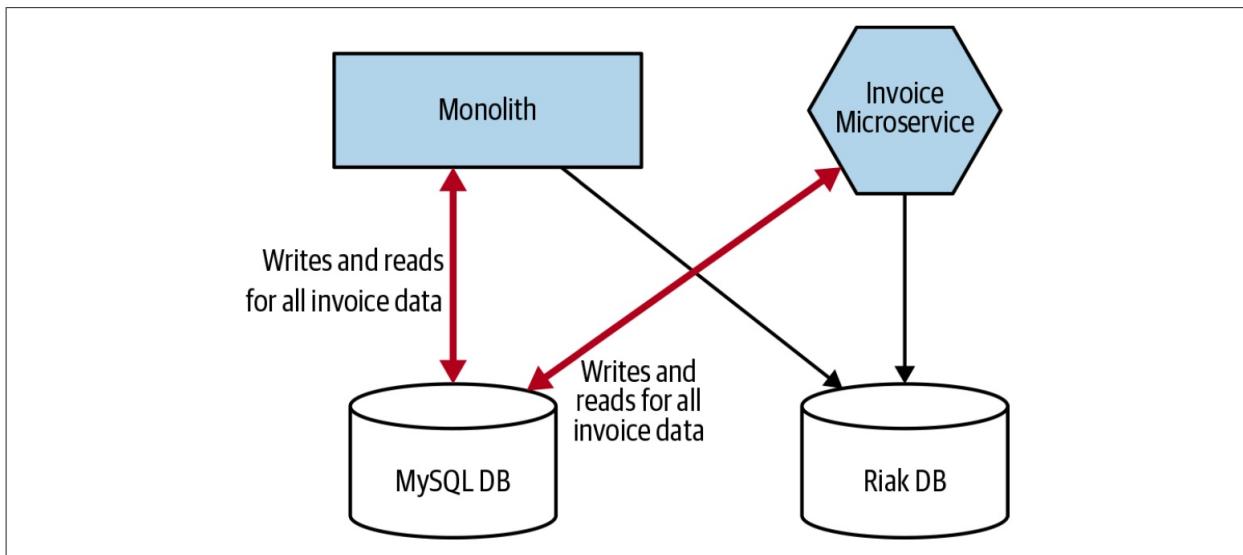


图4-17. 单体和微服务均试图使两个数据库保持同步的例子

<sup>3</sup>. For a detailed presentation on this topic, you can view a recording of Kresten Krab Thorup's talk “[Riak on Drugs \(and the Other Way Around\)](#)”.



# 跟踪器写入模式

图4-18中的tracer write（跟踪器写入）模式可以说是应用程序中同步数据模式（参见第145页的[在应用程序中同步数据部分](#)）的一种变体。使用tracer write，我们就可以采用增量的方式来移动数据源，并容许在迁移期间存在两个数据源。我们将确定一个新的服务来托管已重定向的数据。当前系统仍在本地维护数据记录，但是在修改数据时，也可以确保这些修改通过其服务接口写入新的服务。可以修改现有代码，以开始访问新服务，并且一旦所有功能都使用新服务作为数据源，就可以淘汰旧的数据源。需要仔细考虑如何在两个数据源之间同步数据。

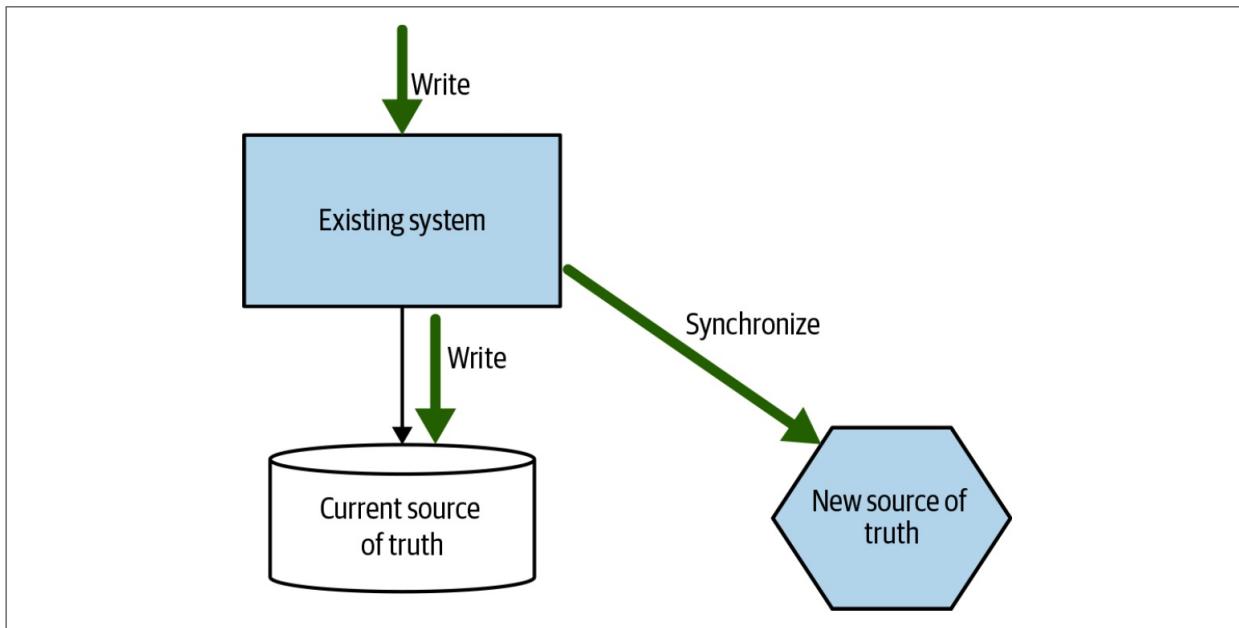


图4-18. tracer write通过在迁移过程中容纳两个数据源，从而允许从一个系统到另一个系统的增量数据迁移

单一的数据源是一种太过理性的愿望。单一的数据源让我们能够确保数据的一致性，控制对数据的访问，并可以降低维护成本。问题是，如果我们坚持只为数据提供一个数据源的话，那么我们将被迫陷入一种情况——改变这些数据所在的位置将成为一个巨大的转换。在发布之前，单体是数据源。发布后，我们的新的微服务才是数据源。问题是，在这种转换过程中，很多事情都会出错。诸如tracer write之类的模式允许进行阶段性切换，从而减少每次发布的影响，作为交换条件，我们需要更加容忍拥有多个数据源。

这种模式称为tracer write的原因是，我们可以从一小部分数据开始同步，并随着时间的流逝而增加同步的数据，同时还可以增加新数据源的使用者的数量。如果以[图4-12](#)中概述的例子为例，与发票相关的数据已从单体中移至新的Invoice微服务，则我们可以首先同步基本的发票数据，然后迁移发票的联系信息（*contact information*），最后同步付款记录，如[图4-19](#)所示。

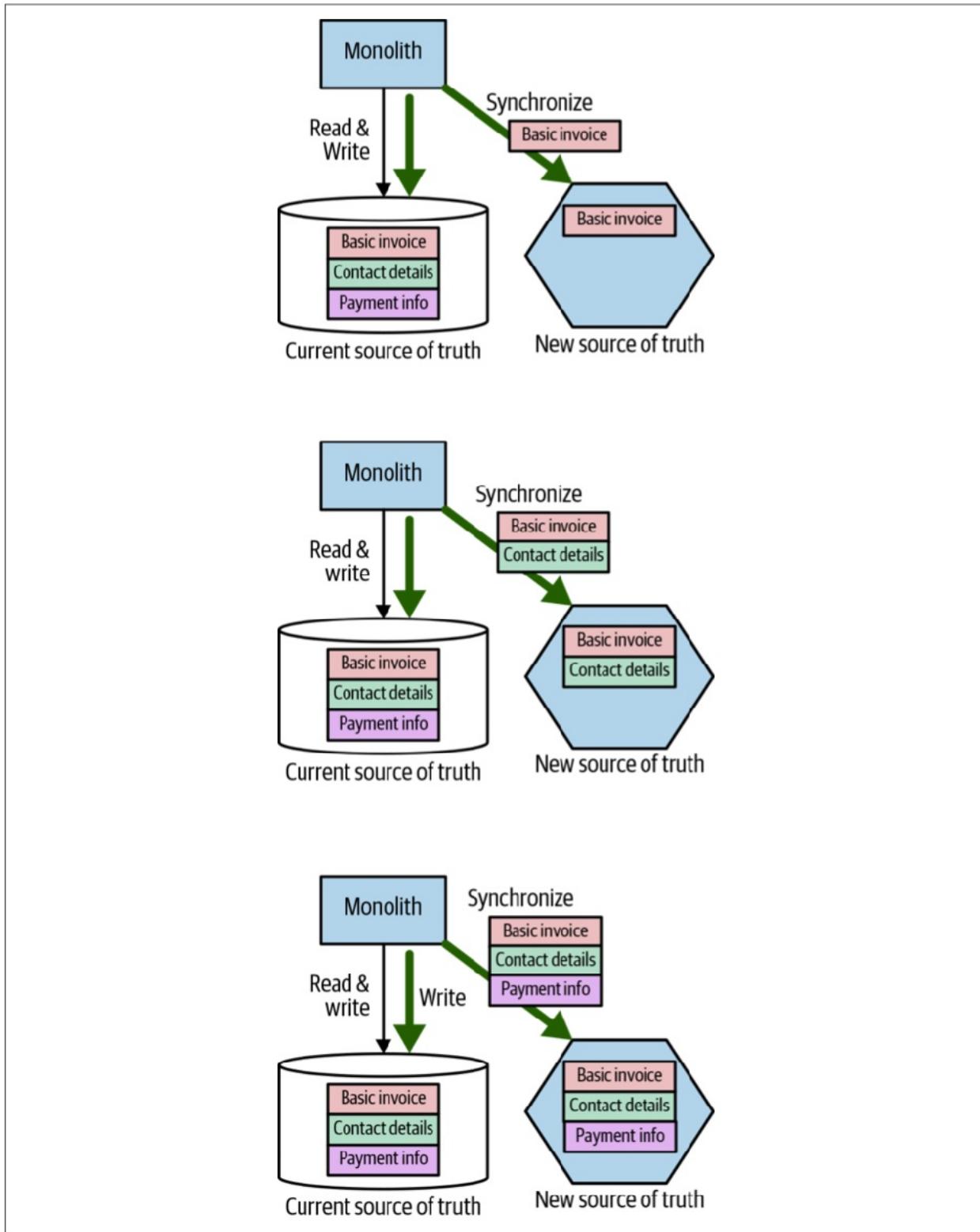


图4-19. 逐步将发票相关信息从单体转移到我们的Invoice服务

其他需要发票相关信息的服务可以选择从单体来获取信息，也可以选择从新服务来获取信息，具体取决于它们所需要的信息。如果他们需要的信息仍然仅在单体中可用，则他们将不得不等待该数据以及支持该数据的功能的迁移。一旦新的微服务中的数据和功能可用，消费者就可以切换到新的数据源。

我们的例子中的目标是迁移所有的消费者以使用Invoice服务，这些消费者包括单体本身。在图4-20中，我们看到了迁移过程中的几个阶段。最初，我们仅将基本的发票信息写入两个数据源。一旦我们确定此信息已正确同步，单体就可以开始从新服务中读取其数据。随着更多数据的同步，单体可以将新服务用作越来越多数据的数据源。一旦所有数据都同步了，并且旧数据源的最后一个消费者也已经切换到新服务，我们就可以停止同步数据。

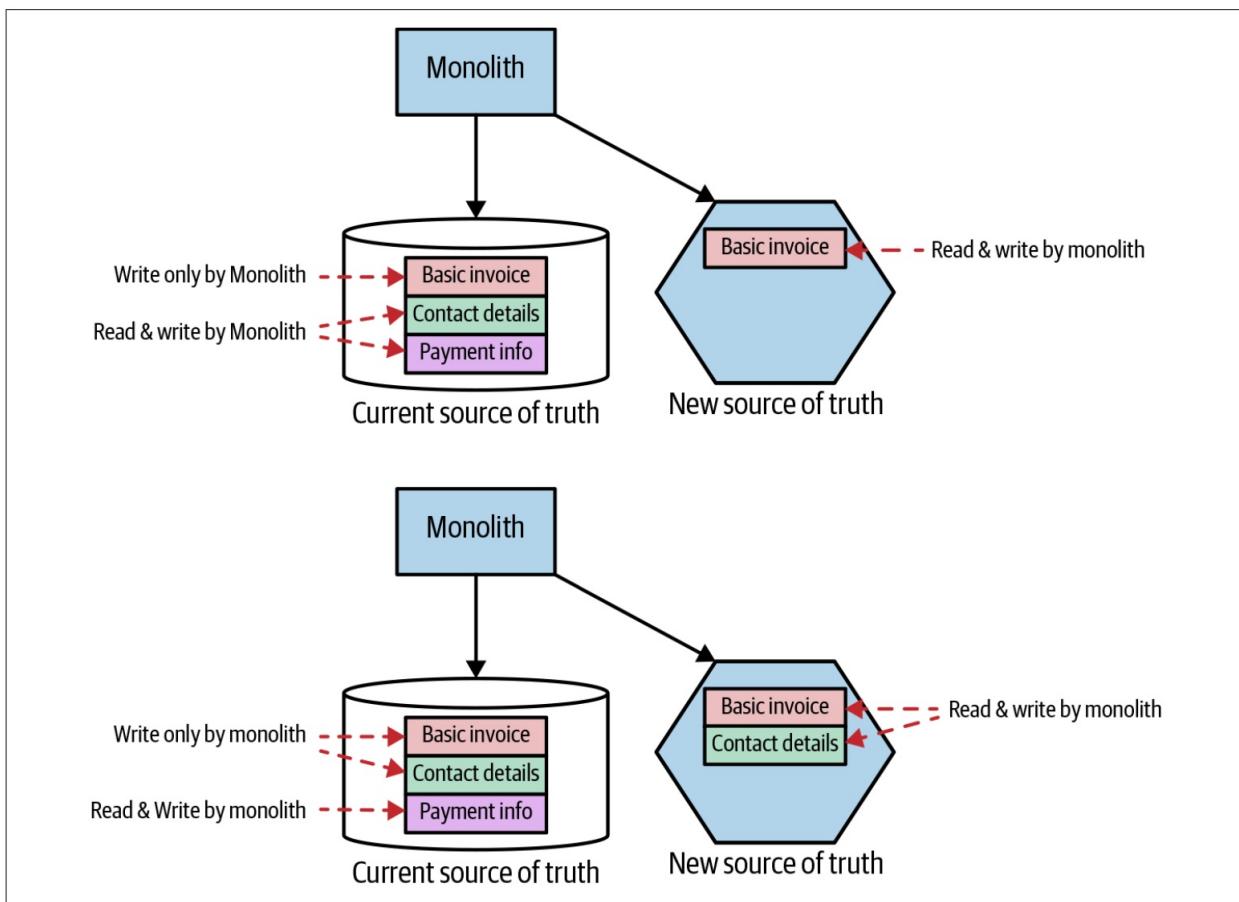


图4-20. 作为tracer write的一部分来下线旧的数据源

# 数据同步

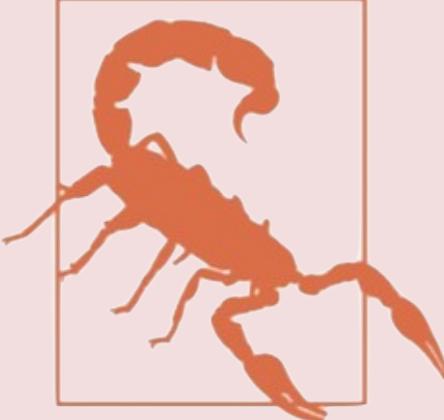
tracer write模式需要解决的最大的问题是：副本数据不一致带来的任何问题。要解决此问题，我们有几种选择：

- **Write to one source**: 所有的写操作都发送到其中的一个数据源。数据写完之后，再同步到其他的数据源。
- **Send writes to both sources**: 上游客户端发出的所有写请求都发送到两个数据源。这种方法通过确保客户端自己调用每个数据源或通过依赖中间件将请求广播到每个下游服务来生效。
- **Seed writes to either source**: 可以将写请求发送到任一数据源，并且在后台，数据在系统之间以双向方式同步。

把写操作发送到两个数据源，或发送到一个数据源并依赖某种形式的幕后同步，这是两种看起来似乎可行的解决方案，我们稍后将探讨使用这两种技术的例子。但是，尽管写入任何一个数据源的方案从技术上讲也是一种选择，但应避免使用这种方法，因为这种方法需要双向同步（双向同步可能很难实现）。

在所有的方案下，两个数据源中的数据一致性都会存在延迟。数据不一致的持续时间取决于很多因素。例如，如果采用每天夜间用批处理的方式把数据更新从一个源复制到另一个源，则第二个数据源可能包含有延迟24小时的数据。如果使用CDC系统将更新从一个系统流式传输到另一个系统，则数据不一致的时间窗口可以是几秒钟或更短的时间。

无论这种不一致的时间窗口是多少，这种同步给我们提供了所谓的最终一致性——最终，两个数据源将具有相同的数据。必须了解哪种数据不一致的周期适用于我们的场景，并使用该时间来驱动我们如何实现数据同步。



重要的是，在维护两个这样的数据源时，必须进行某种对帐流程，以确保能够按照预期执行数据同步。对账流程可能就像对每个数据库执行几条SQL查询一样简单。但是，如果不检查是否按预期进行数据同步，可能会导致两个系统之间出现不一致，并且在意识到不一致的存在时，为时已晚。在还没有消费者的情况下，让新的数据源运行一段时间，直到我们对新数据源的运行方式感到满意为止，是非常明智的。正如我们将在下一部分中探讨的那样，[Square公司](#)就是这样做的。

# Square公司的订单的例子

此模式最初是由Square的开发人员Derek Hammer分享给我的，此后，我发现了使用该模式的其他例子<sup>4</sup>。Derek Hammer详细介绍了该模式如何帮助理清Square的与外卖相关订单的业务部分。

在最初的系统中，一个单独的Order概念用于管理多个工作流程：

- 一个工作流用于客户订购食物
- 另一个工作流用于餐厅准备食物
- 第三个工作流用于管理配送员取食物并配送给客户的状态

尽管如上的三个工作流都使用相同的Order，但是其利益相关者的需求是不同的。对于客户来说，Order是他们选择要配送的东西，也是他们需要付费的东西。对于餐馆来说，Order是需要烹调并运送的东西。对于配送员来说，Order是需要及时将其从餐厅送到顾客身边的东西。尽管有这些不同的需求，但订单的代码和关联数据都被绑定在一起。

将所有这些工作流程捆绑到一个Order概念中，在很大程度上就是我之前所说的“交付冲突”的源头——不同的开发人员尝试针对不同的用例进行修改会互相干扰，因为所有这些都需要对代码库的同一部分进行修改。Square希望拆解Order，以便可以独立地修改每个工作流，并且还可以实现不同的缩放需求和健壮性需求。

## 创建新的服务

第一步是创建一个新的Fulfillments服务，如图4-21所示，该服务管理与餐厅和配送员相关的订单数据。Fulfillments服务将成为Order数据子集的新的数据源。最初，该服务仅公开了允许创建与Fulfillments相关的实体的功能。新服务启用后，将有一个后台工作线程把和Fulfillments相关的数据从现有系统复制到新的Fulfillments服务。该后台工作线程只是利用了Fulfillments服务提供的API，而不是直接将数据插入到数据库中，从而避免了直接访问数据库。

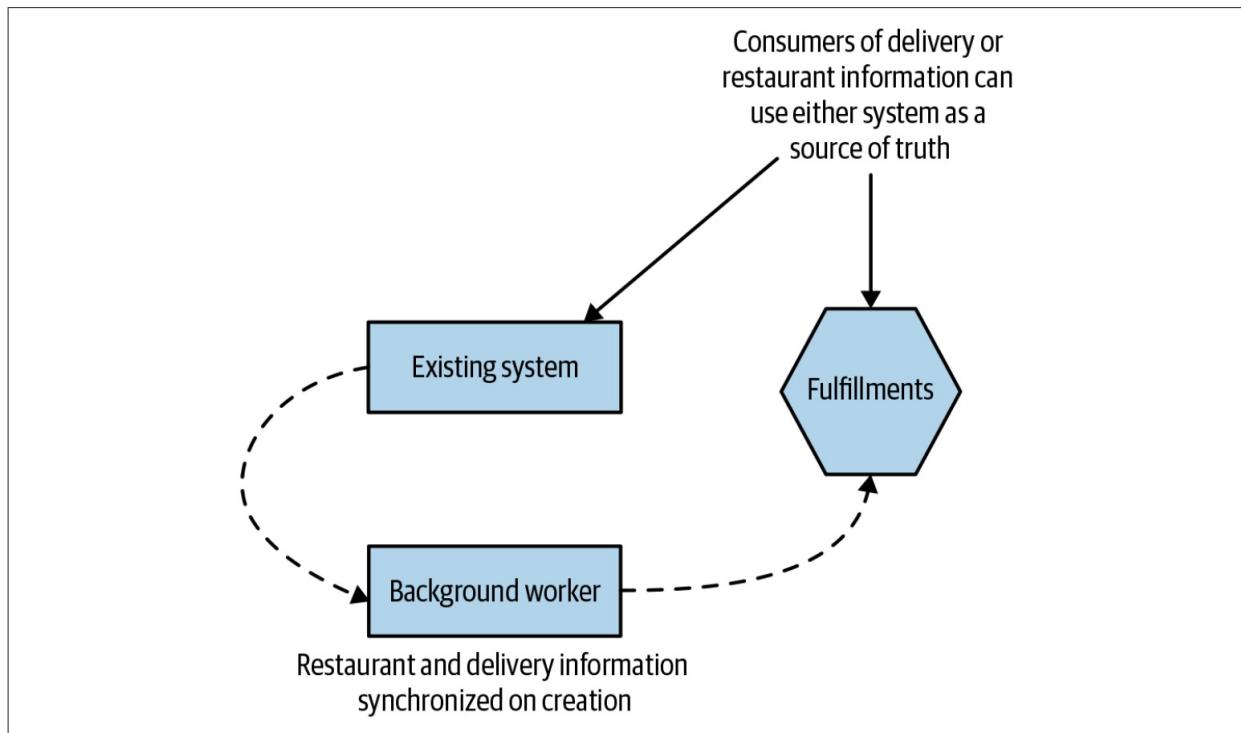


图4-21. 新的Fulfillments服务用于从现有系统复制与fulfillments相关的数据

通过功能标记（*feature flag*）来控制后台工作线程。该功能标记可以启用或禁用以停止数据复制。从而可以确保，如果后台工作线程在生产环境中引起任何问题，很容易停止该程序。该系统在生产环境中运行了足够长的时间，以确保数据同步是正常的。一旦人们觉得该后台工作线程是按照预期工作的，便删除了功能标记。

## 同步数据

对现有系统进行的修改导致与fulfillment相关的数据通过新的Fulfillments服务的API写入到新服务中。Square通过确保两个系统都进行了所有更新来解决此问题，如图4-22所示。但是，并非所有的更新都需要应用于两个系统。正如Derek所解释的那样，现在Fulfillments服务仅代表Order概念的一个子集，仅需复制配送或饭店客户所关心的订单的更新。

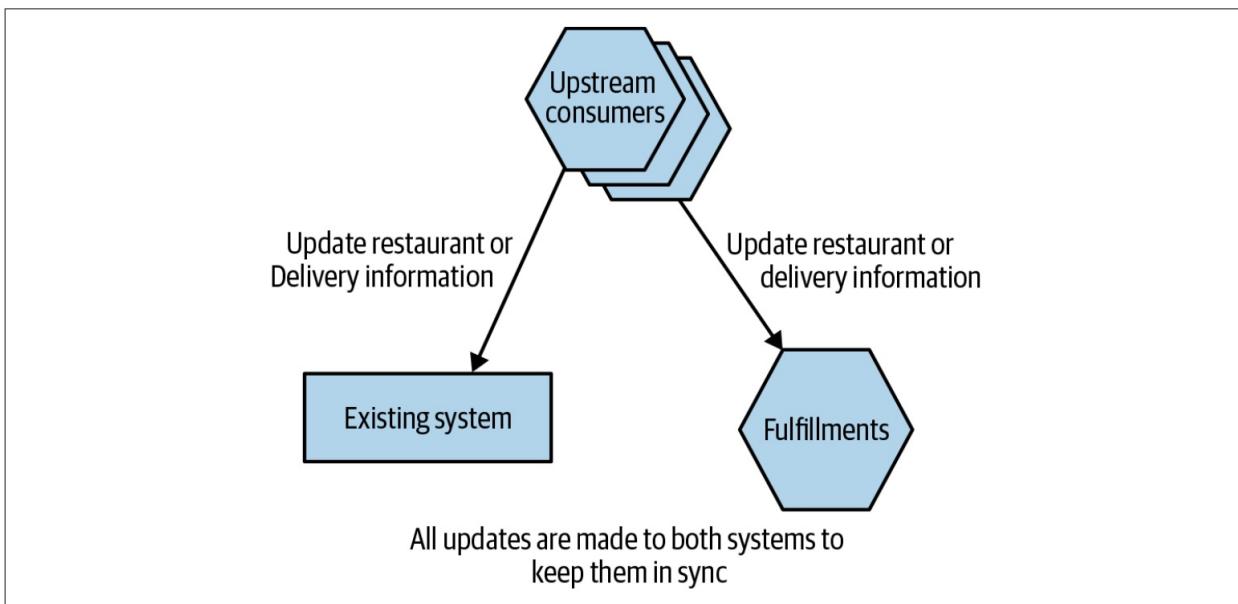


图4-22. 通过确保所有使用者对这两种服务进行适当的API调用来同步后续更新

任何修改对面向餐厅或面向配送信息的代码都需要进行修改，以执行两组API的调用：一组调用现有系统，另一组调用微服务。如果一组调用是成功的，而另一组调用失败了，则上游的客户端还需要处理任何的错误情况。对两个下游系统（现有的订单系统和新的配送服务）的数据修改并非以原子方式完成。这意味着可能会有一个短暂的窗口，在该窗口中，一个系统中可以看到数据修改，而在另一个系统则看不到数据修改。在两个系统都完成数据修改之前，可以看到，两个系统之间存在不一致。这是最终一致性的一种形式，我们在前面已经讨论过。

就Order信息的最终一致性而言，对于此特定的用例而言，这不是问题。数据在两个系统之间的同步速度足够快，不会影响系统用户。

如果Square一直使用事件驱动的系统来管理Order更新，而不是使用API，则他们可以考虑使用另一种实现方式。在图4-23中，我们有一条消息流，可以触发修改Order状态。现有系统和新的Fulfillments服务都会收到相同的消息。上游客户端不需要知道这些消息有多个消费者，可以使用发布-订阅方式的消息代理来解决多个消费者的场景。

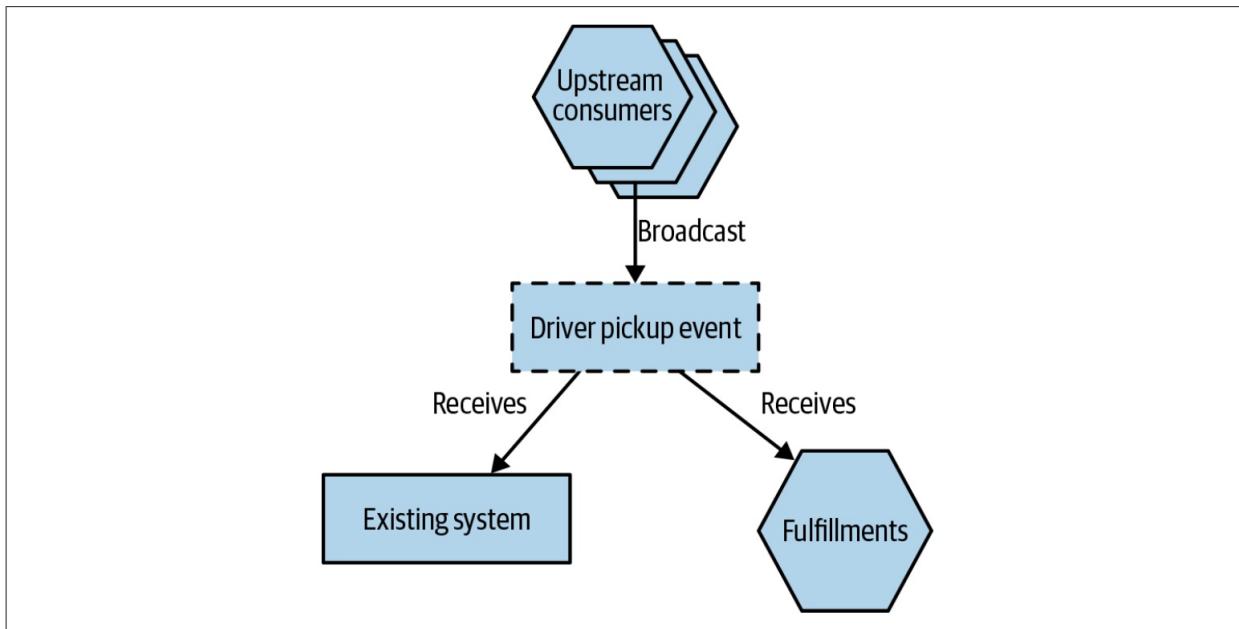


图4-23. 另一种同步方法是让两个数据源都订阅相同的事件

将Square的架构改造为基于事件的架构以满足该用例将需要大量工作。但是，如果已经在使用基于事件的系统，则可以更轻松地管理同步过程。还要值得注意的是，因为不能保证现有的系统和Fulfillments服务同时处理同一事件，基于时间系统的架构仍将表现为最终一致性。

## 迁移消费者

现在，随着新的Fulfillments服务拥有饭店和配送员工作流所需的所有信息，管理这些工作流程的代码可以开始切换到新服务。在迁移期间，可以增加更多功能来满足消费者的需求。最初，Fulfillments服务仅需要实现一个API，该API可以为后台工作线程创建新的记录。随着新的消费者迁移到新服务，可以评估消费者的需求并将新功能增加到服务中，以支持他们的需求。

对于Square的情况，事实证明，增量迁移数据以及增量改变消费者以使其采用新的数据源的方式都非常有效。Derek说，让所有消费者都切换到新的服务，这几乎是一件不可能的事情。增量迁移只是例行发布中所做的又一个小的变更（这也是我在本书中一直强烈提倡增量迁移模式的另一个原因！）。

从域驱动的设计的角度来看，当然可以辩称：和配送员，客户和餐厅相关的功能都代表了不同的有界上下文。从这种观点出发，Derek建议最好进一步考虑将该配送服务拆分为两个服务：一个用于餐厅，另一个用于配送员。尽管如此，虽然仍有进一步拆分的空间，但目前的这种迁移似乎已经非常成功。

在Square公司的例子中，Square决定保留重复的数据。把与餐厅和配送相关的订单信息保留在现有系统中，可以在无法使用Fulfillments服务的情况下，仍能提供信息的可见性。当然，这需要保持数据同步。我想知道，随着时间的推移，是否会重新审视保留重复数据的这一做法。一旦对Fulfillments服务的可用性有足够的信心，删除后台工作进程以及让消费者仅执行一组更新调用可以很好地帮助简化架构。

# 何处使用该模式

实现数据同步可能是大多数工作所在。如果可以避免双向同步，而是使用此处概述的一些较简单的方法，则可能会发现该模式更容易实现。如果已经在使用事件驱动的系统，或者有可用的CDC管道，那么我们可能已经有很多可用的模块用于同步数据。

需要仔细考虑可以容忍两个系统在多长时间内是不一致的。有些case可能不在乎系统之间的数据不一致性，其他的case可能希望复制几乎是即时的。可接受的数据不一致的时间窗口越短，实施此模式将越困难。

---

<sup>4</sup>. Sangeeta Handa shared how Netflix used this pattern as part of its data migrations at the QCon SF conference, and Daniel Bryant subsequently did a [nice write-up](#) of this. ↪

# 拆分数据库

我们已经详细讨论了使用数据库作为集成多种服务所面临的挑战。现在应该很清楚了，我不是这种方法的粉丝！这意味着，在数据库中，我们也需要找到[接缝](#)，以便可以对其进行清晰的拆分。但是，数据库拆分是一件很难处理的事情。在介绍某些拆分方法的示例之前，我们应该简要讨论数据库的逻辑分离和物理部署之间的关系。

接缝是Michale在《修改代码的艺术》一书中提出的概念。接缝是指程序中的特殊的点，在这些点上，无需做任何修改就可以改变程序的行为。本质而言，可以围绕要更改的代码段定义接缝，对该接缝进行新的实现，并在变更完成后用新的实现替换旧有的实现。Michael把使用接缝来安全的工作的技术作为帮助清理代码库的一种方式。

# 数据库的物理分离 VS 逻辑分离

当我们谈论拆分数据库时，我们主要是在试图实现数据库的逻辑分离。如[图4-24](#)所示，单个数据库引擎完全能够承载多个逻辑上分离的schema。

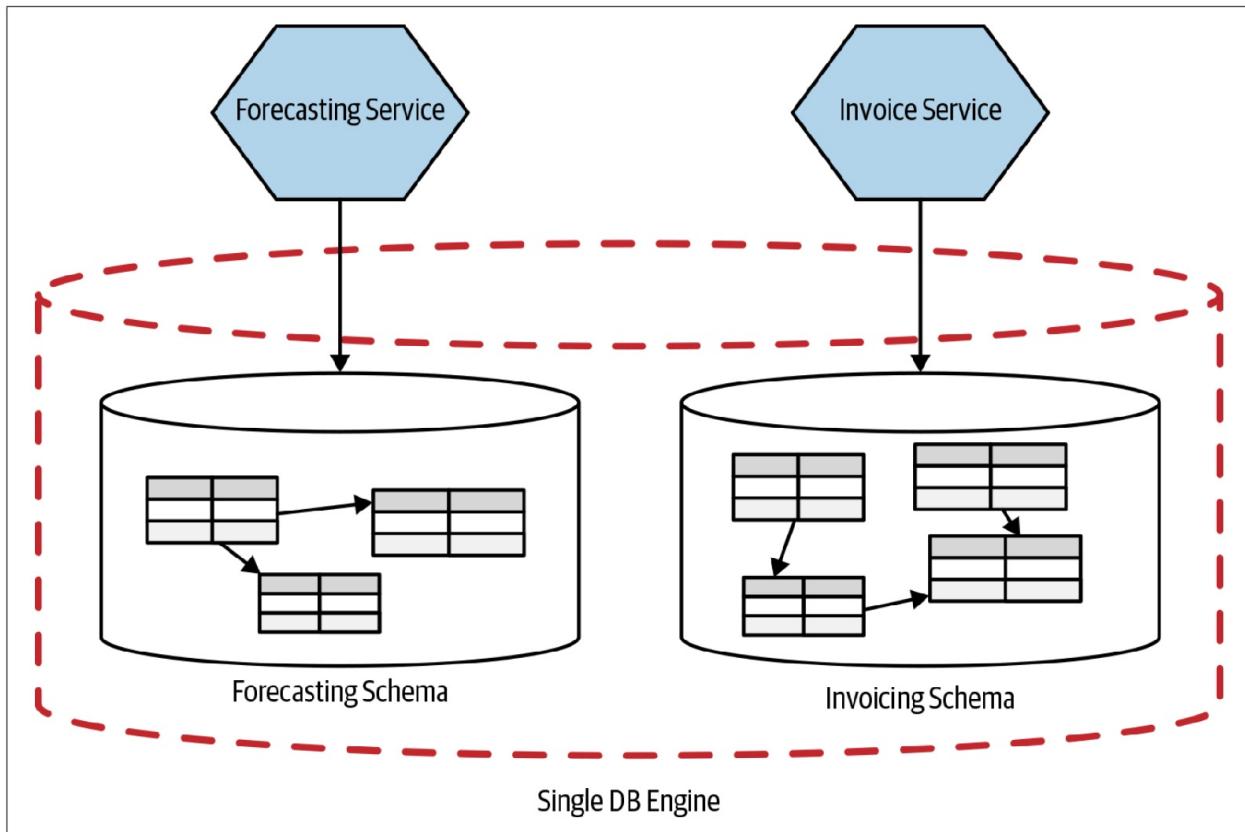


图4-24. 两个服务使用运行在同一个物理引擎之上的相互分离的逻辑schema

更进一步，我们可以把每个逻辑schema置于单独的数据库引擎，这也给我们带来了数据库的物理分离，如[图4-25](#)所示。

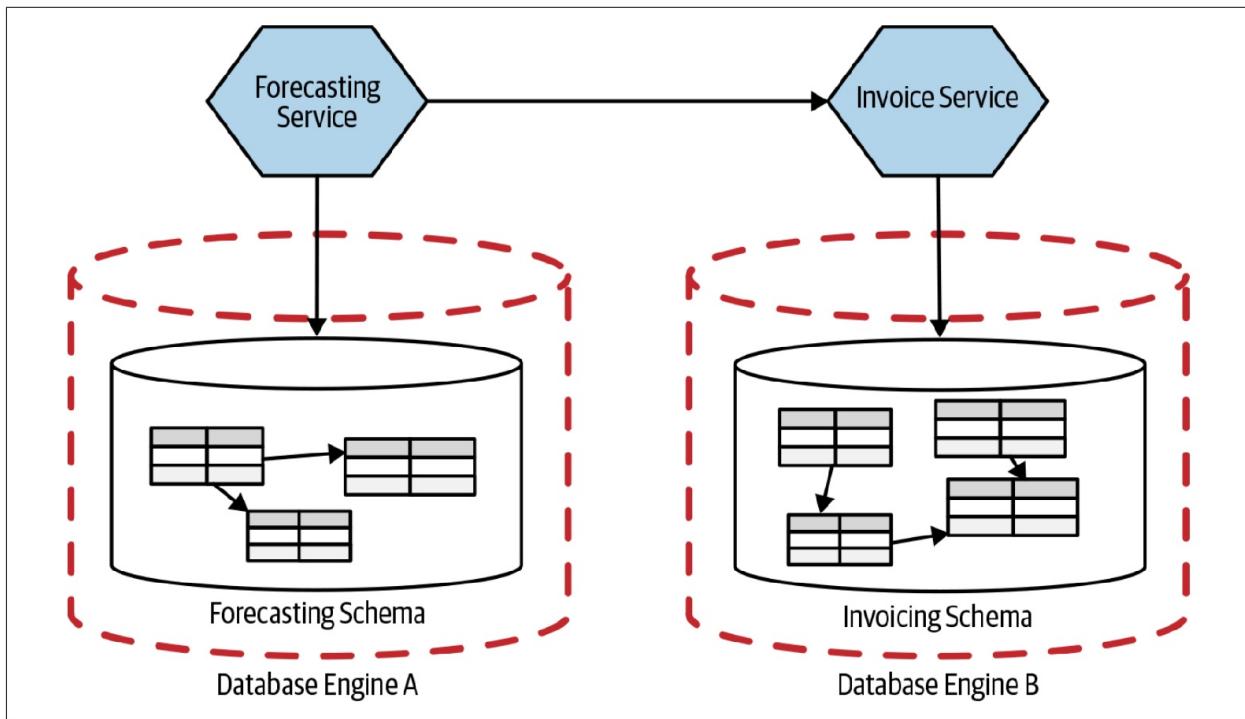


图4-25. 两个服务使用运行在分离的物理引擎之上的分离的逻辑schema

为什么要在逻辑上拆分schema，但却仍然将这些schema置于单个数据库引擎上呢？好吧，从根本上讲，逻辑拆分和物理拆分可以实现不同的目标。逻辑拆分可以实现更为简单的独立变更和信息隐藏；而物理拆分则可以提高系统的鲁棒性，同时还可以帮助消除资源冲突，从而提高吞吐量或降低延迟。

如图4-24所示，当我们在逻辑上拆分数据库schema，但让他们继续位于相同的物理引擎时，我们就存在单点故障的风险。如果数据库引擎出现故障，则两个服务都会受到影响。但是，没那么简单。许多数据库引擎都具有避免单点故障的机制，例如：多主数据库模式，温备故障转移机制

(*warm failover mechanisms*) 等。实际上，在组织中，已经付出了巨大的努力来创建高弹性数据库集群。同时，由于多集群会涉及到时间、精力和成本（那些令人讨厌的许可证费用也会叠加！），因此很难证明拥有多个集群是合理的。

## 不同类型的故障转移机制

- 冷备服务器（cold server）是在主服务器故障的情况下才使用的备服务器。
- 温备服务器（warm server）一般都是周期性开机，根据主服务器内容进行更新，然后关机。经常用温备服务器来进行复制和镜像操作。
- 热备服务器（hot server）时刻处于开机状态，同主机保持同步。当主机故障时，可以随时启用热备服务器。

另一个考虑因素是，如果要开放数据库视图，可能需要多个schemas之间共享同一个数据库引擎。源数据库和托管视图的schemas都要位于同一个数据库引擎。

当然，即使我们可以选择在不同的物理引擎上运行单独的服务，我们也需要先在逻辑上拆分这些服务的数据schemas。

Copyrights © wangwei all right reserved

# 先拆分数据库还是先拆分代码

到目前为止，我们已经讨论了对共享数据库有用的很多模式，并希望转向较少耦合的模型。稍后，我们会详细研究数据库的拆分模式。但是，在此之前，我们需要讨论拆分数据库和拆分代码的顺序。直到应用程序代码运行在其自己的服务中，并将服务控制的数据抽取到服务自己的逻辑隔离数据库，微服务的抽取才算完成。但是，本书主要是关于实现增量变更的，因此，我们必须探索：如何对微服务的抽取过程进行排序的方法。我们有几种选择：

- 先拆分数据库，然后再拆分代码
- 先拆分代码，然后再拆分数据库
- 同时拆分数据库和代码

不同的选择各有利弊。现在，我们来看一下这些选择，并根据所采用的方法，来看一些可能有用的模式。

# 先拆分数据库

使用分离的schema，执行单个操作时，可能会增加数据库的调用次数。之前，我们可以用单个SELECT语句获得所需的所有数据，而现在，我们需要从两个schema中获取到数据，并在内存中对其进行合并。当我们转而使用两个schema时，最终，我们也破坏了事务的完整性，这可能会对我们的应用程序产生重大影响。本章稍后会讨论这些挑战，本章稍后的内容涵盖了分布式事务和sagas之类的话题，以及它们如何帮助我们解决这些挑战。如图4-26所示，我们拆分了schemas，但并未拆分应用程序的代码。如果我们意识到变更有异常，我们就可以还原变更或继续调整，从而不会影响服务的任何使用者。一旦确信数据库的拆分是合理的，我们就可以考虑将应用程序代码拆分成两个服务。

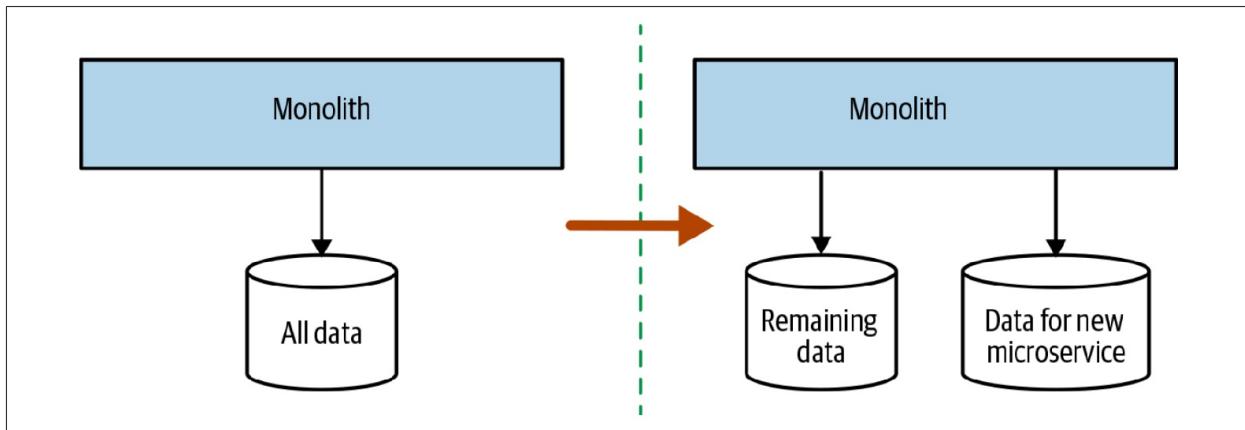


图4-26. 首先拆分schema可以使我们更早的发现性能和事务完整性方面的问题

另一方面，先拆分数据库的缺点是，该方法很难产生太多的短期利益。使用该方法时，我们仍然是部署单体代码。可以说，共享数据库的缺点是随着时间的流逝而逐渐显现出来，因此，此刻，我们正在花费时间和精力来

获得长期回报，而不是获得足够的短期利益。有鉴于此，只有在特别关心性能或数据一致性问题时，我才会选择先拆分数据库的方法。还需要考虑的是，如果单体本身是一个黑盒系统，例如一款商业软件，那么我们将无法使用先拆分数据库的方法。

## 工具说明

因为很多原因，修改数据库是一件困难的事，原因之一就是只有有限的工具可以让我们轻松的修改数据库。修改代码时，我们有IDE的内置重构工具；并且，还存在额外的好处，那就是：从根本上讲，我们要修改的系统是无状态的。对于数据库而言，我们要修改的内容是有状态的，同时，我们还缺少良好的重构类工具。

许多年前，这种工具上的空白促使我和两个同事，Nick Ashley和Graham Tackley，开发了一个名为DBDeploy的开源工具。这个工具现在已经不存在了（开发一个开源工具与维护一个开源工具完全不同！），DBDeploy允许捕获修改并将其存入SQL脚本，这些脚本可以以确定性的方式在schema上运行。每个schema都有一个特殊的表，该表用于跟踪已应用了哪些schema脚本。

DBDeploy的目标是允许我们对schema进行增量修改，以控制每个修改的版本，并允许在不同时间修改多个schema（例如开发schema，测试schema和生产schema）。

如今，我推荐人们使用[FlywayDB](#)或提供类似功能的产品，但是无论选择哪种工具，我都强烈建议确保这些工具允许我们捕获版本控制的增量脚本中的每个修改。

## 每个界定的上下文一个数据持久层

## repository的概念

在《企业架构模式》中，译者将repository翻译为资源库，并给出如下说明：通过用来访问领域对象的一个类似集合的接口，在领域与数据映射层之间进行协调。

在《领域驱动设计：软件核心复杂性应对之道》中，译者将repository翻译为仓储，并给出如下说明：一种用来封装存储，读取和查找行为的机制，它模拟了一个对象集合。

而在此处，将repository翻译为资源库或者仓储都不太合适，这会让这个概念越翻译越难以理解。就像把“back pressure”翻译成背压一样。

微软的官方文档：[Design the infrastructure persistence layer](#)中，是这样定义repository的：

Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer. If you use an Object-Relational Mapper (ORM) like Entity Framework, the code that must be implemented is simplified, thanks to LINQ and strong typing. This lets you focus on the data persistence logic rather than on data access plumbing.

The Repository pattern is a well-documented way of working with a data source. In the book [Patterns of Enterprise Application Architecture](#), Martin Fowler describes a repository as follows:

A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping.

因此，简单而言，此处，repository的含义就是：数据持久层。

一种常见的做法是增加一个数据持久层（*repository layer*）来把代码和数据库绑定在一起，从而实现把数据库映射成对象或数据结构。数据持久层可以利用类似Hibernate的框架来支持。没有必要为所有的数据访问都创建一个数据持久层，如图4-27所示，可以按照界定的上下文来拆解数据持久层，这很有价值。

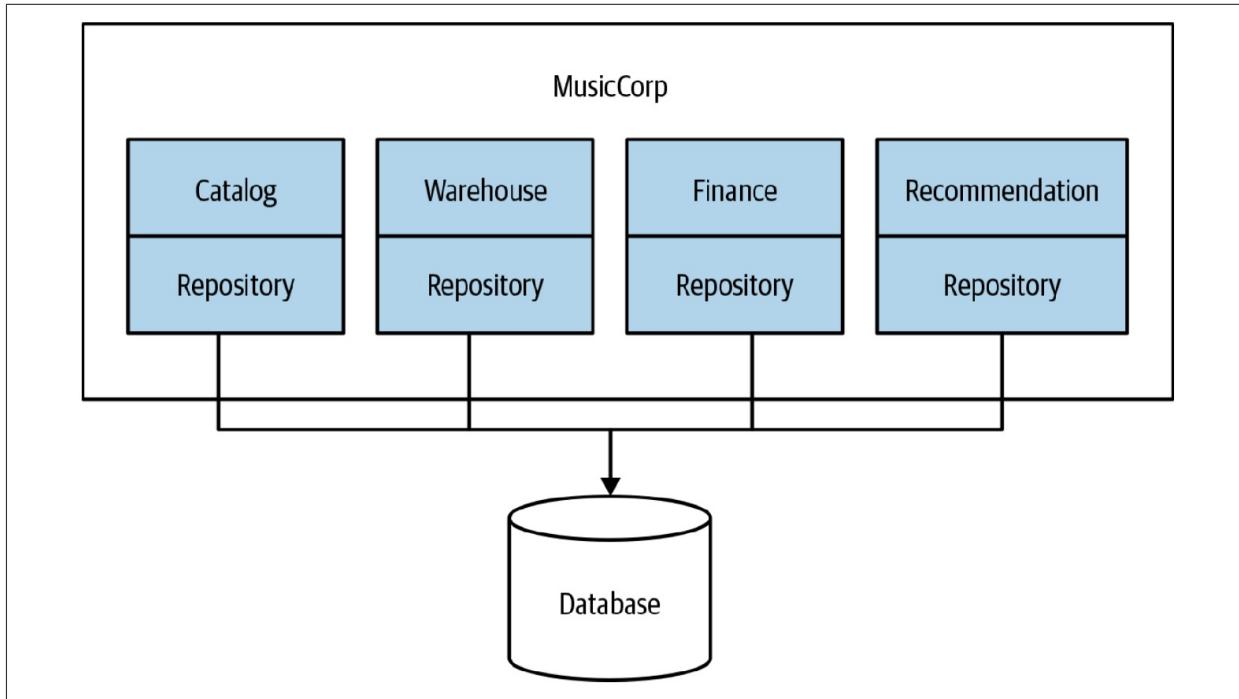


图4-27. 拆分数据持久层

给定一个上下文，将数据库的映射代码和该上下文的代码放置在一起，可以帮助我们了解哪些代码位使用了数据库的哪些部分。例如，如果在每个界定的上下文中使用诸如映射文件 (*mapping file*) 之类的技术，那么 Hibernate可以让我们非常清楚的了解到：哪些代码位使用了数据库的哪些部分。我们可以由此而看到：哪些界定的上下文访问了schema中的哪些表。如上的做法可以帮助我们极大地了解：将来进行任何拆分时需要移动哪些表。

但是，采用数据映射的方法并不能给我们提供全部的线索。例如，我们也许可以判断出 `finance` 的代码使用了 `ledger`表，`catalog` 的代码使用了 `item`表；但是，我们可能不清楚数：数据库是否存强制实施了从 `ledger` 表 到 `item`表 的 外键。数据库级别的约束可能是迁移工作的绊脚石，发现这些数据库级别的约束，我们需要使用另一种工具来可视化数据。非常好的起点是：使用免费的[SchemaSpy](#)之类的工具，该工具可以可视化表之间的关系。

如上的方法可以帮助我们了解表之间的耦合，这些耦合可能会跨越未来的服务边界。但是，如何斩断这些耦合？在多个界定的上下文中使用相同的数据表会是什么情况？我们将在本章的稍后部分详细探讨该主题。

## 何处使用该模式

在我们希望重新修改单体以更好地了解如何拆分单体的任何情况下，为每一个界定的上下文定义一个数据持久层的模式都非常有效。按照领域概念拆分数据持久层会帮助我们了解到：微服务的[接缝](#)不仅存在于数据库，而且还存在于代码本身。

## 每个界定的上下文一个数据库

一旦从应用程序的角度明确隔离数据访问，就可以把这种方法继续应用于数据schema。微服务独立可部署的思想的核心是：微服务应该拥有自己的数据。在分离出应用程序代码之前，我们可以通过围绕识别出的界定的上下文来明确地分离数据库，并拆分数据库。

在ThoughtWorks的时候，我们当时正在实施一些新的机制来计算和预测公司的收入。在此过程中，我们确定了需要编写的三个主要的功能区域。我与项目领导——Peter Gillard-Moss——讨论了这个问题。Peter解释说，该功能感觉很独立，但是他担心如果将这些功能放在单独的微服务中会带来额外的工作。目前，他的团队很小——只有3个人，并且认为团队还无法证明拆分这些新服务是合理的。最后，他们选择了一个模型，该模型把新的收入功能有效地部署为单个服务，该单个服务包含三个隔离的界定的上下文（每个上下文最终都成为单独的JAR文件），如图4-28所示。

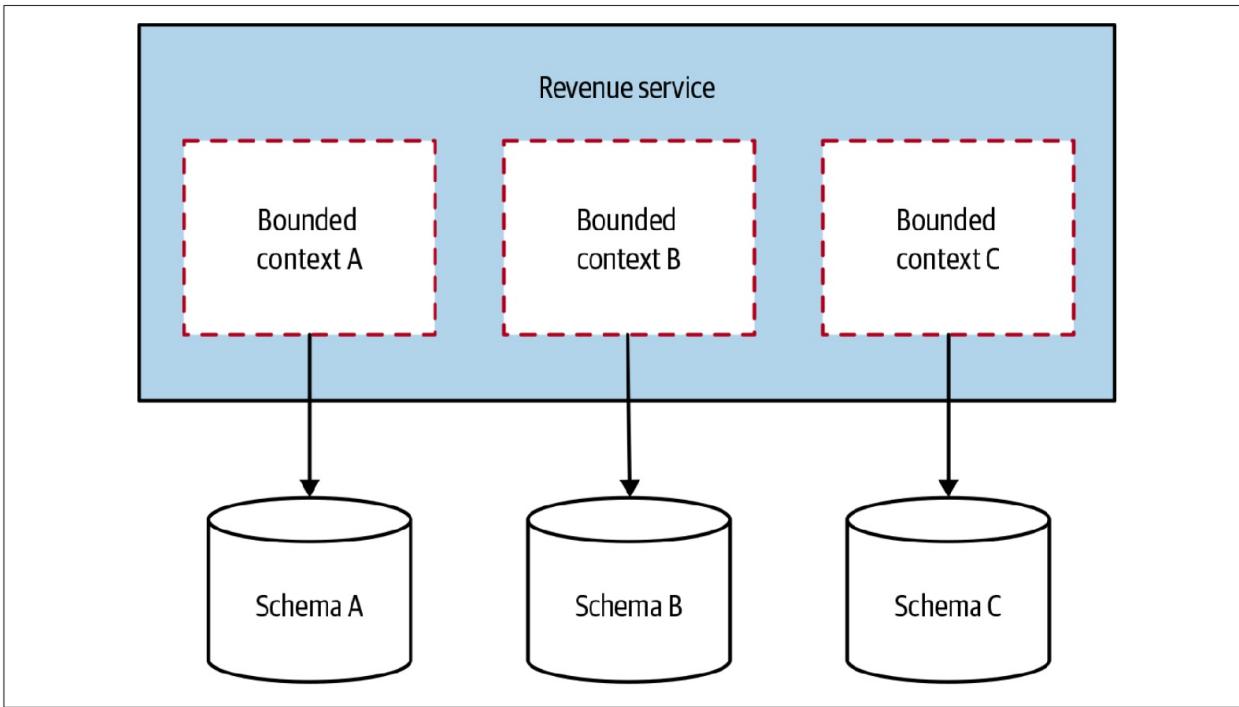


图4-28. Revenue服务中的每个界定的上下文都有其独立数据库schema，后续可以对其进行分离

每个界定的上下文都有其自己的、完全独立的数据库。该想法是：如果以后需要把这些上下文分离为微服务，这会容易得多。然而，事实证明，根本不需要这么做。几年后，revenue服务仍然保持不变，还是处于一个单体和多个关联数据库的形式，一个模块化单体的绝好例子。

## 何处使用该模式

乍一看，对于单体而言，维护分离的数据库的额外工作没有多大意义。我把该模式视为一种“[对冲](#)”模式。与单个数据库相比，分离为多个数据库需要做更多的工作，但是可以在以后的微服务迁移时保留选择的余地。即使从未使用过微服务，清晰的分离数据schema也确实可以帮助我们，尤其是当我们有很多人在单体上工作时。

我总是建议人们使用该模式来构建全新的系统（而不是重新实现现有系统）。我不建议新产品或初创企业采用微服务。对于新产品或初创企业而言，人们对域的了解可能还不够成熟，还无法确定稳定的域边界。特别是对于初创公司而言，其产品的性质可能会发生巨大变化。但是，每个界定的上下文一个数据的模式可能是一个不错的选择。分离数据schema，让其保持在可以方便日后分离服务的状态。这样，我们将获得数据库分离的好处，同时还降低了系统的复杂性。

# 先拆分代码

总的来说，我发现，大多数团队会先拆分代码，然后拆分数据库，如图4-29所示。团队希望从新服务中获取短期收益，以使他们有信心利用数据库分离来完成服务拆分。

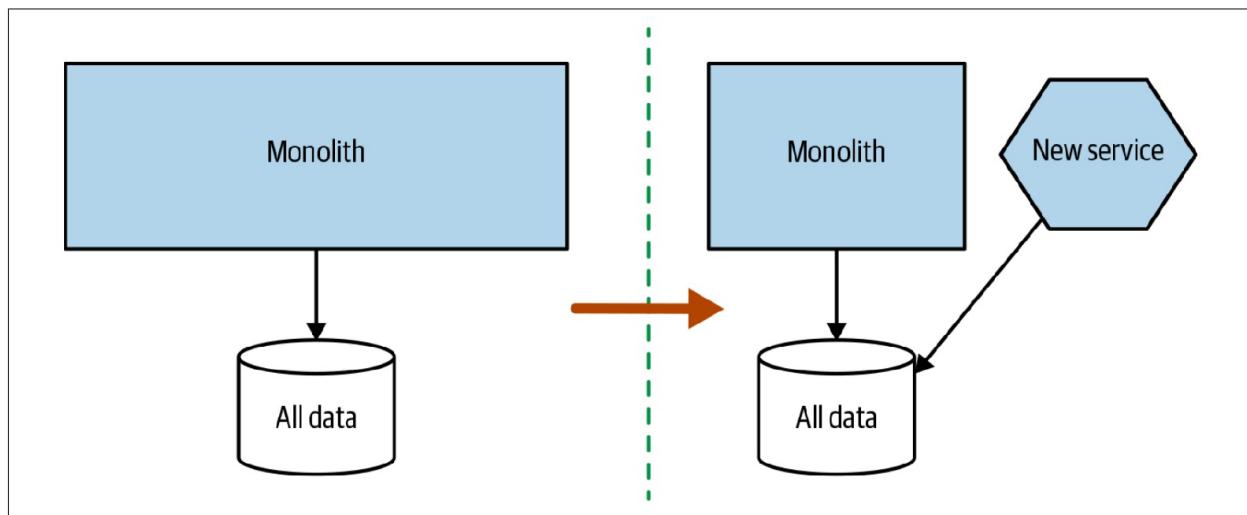


图4-29. 先拆分应用程序层让我们拥有一个共享的数据schema

通过拆分应用程序层，可以更轻松地了解新服务所需的数据。我们还能获得较早的拥有可独立部署的代码的好处。我一直对先拆分代码的方法感到担心的是：团队在应用程序的拆分上可能会走得很远然后停下来，从而继续保持共享数据库的状态。如果这就是我们的方向，那么必须了解：没有完成数据层的分离会给未来带来麻烦。我见过陷入这种陷阱的团队，但也可以很高兴地向在“此处”做正确事情的组织报告这些谈资。JustSocial就是一个做正确事情的组织，JustSocial使用先拆分代码的方法作为自己的微服务迁移的一部分。此处，另外的风险是我们可能会延迟发现那些把 `join` 操作 放在应用程序层而引起的令人讨厌的意外。

如果这是我们的方向，请对自己忠诚：我们是否可以确保将微服务所拥有的所有数据拆分出来，作为下一步的一部分？

## 把单体作为数据访问层

与其直接从单体中访问数据，不如在单体中创建一个访问数据的API。在图4-30中，Invoice服务需要Customer服务中有关雇员的信息，因此我们创建了一个Employee API，以允许Invoice服务访问这些信息。JustSocial的Susanne Kaiser与我分享了这种模式，JustSocial公司已经成功地将其用于微服务迁移。这种模式有很多用途，令我感到惊讶的是，该模式似乎没有获得应有的知名度。

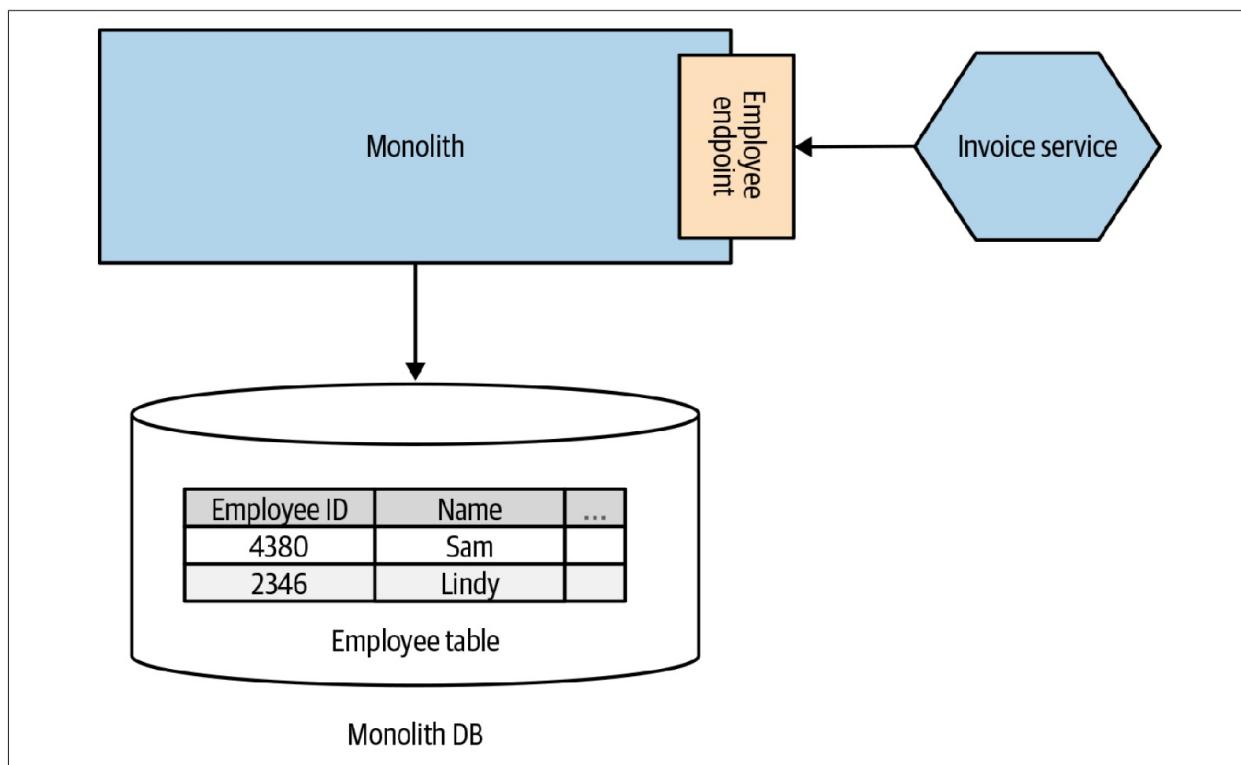
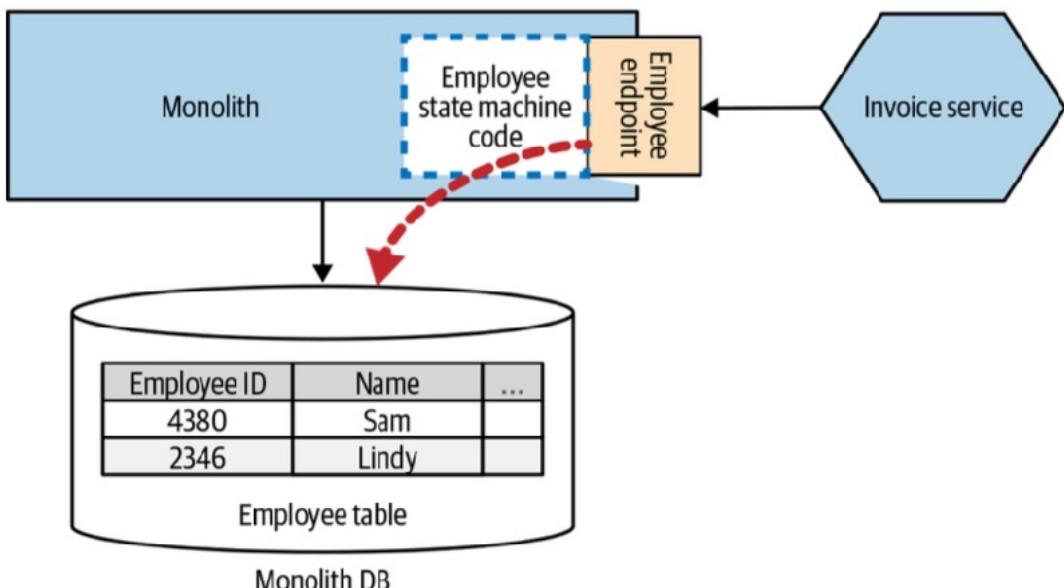


图4-30. 在单体上开放API可使服务避免直接与单体数据绑定

把单体作为数据访问层的方法没有得到广泛使用的部分原因可能是因为人们在某种程度上认为“单体已死且无用”。人们想摆脱单体。人们不会考虑让其更有用！但是，该方法有明显的好处：我们不必解决数据拆分（或者尚未开始解决数据拆分），但却可以隐藏信息，从而使我们的新服务与单体的隔离更加容易。如果我认为单体中的数据将保留在那里，我将更倾向于采用此模型。该方式仍可以很好地工作，特别是如果我们认为新服务实际上是没有状态的。

把这种模式视为识别其余待抽取的服务的一种方式并不难。扩展“把单体作为数据访问层”的想法，我们可以将Employee API从单体中分离出来，并让其成为一个微服务，如[图4-31](#)所示。

**Before:** Employee data and functionality hosted in the monolith, accessed via an API



**After:** Employee data and functionality extracted from the monolith into a new microservice

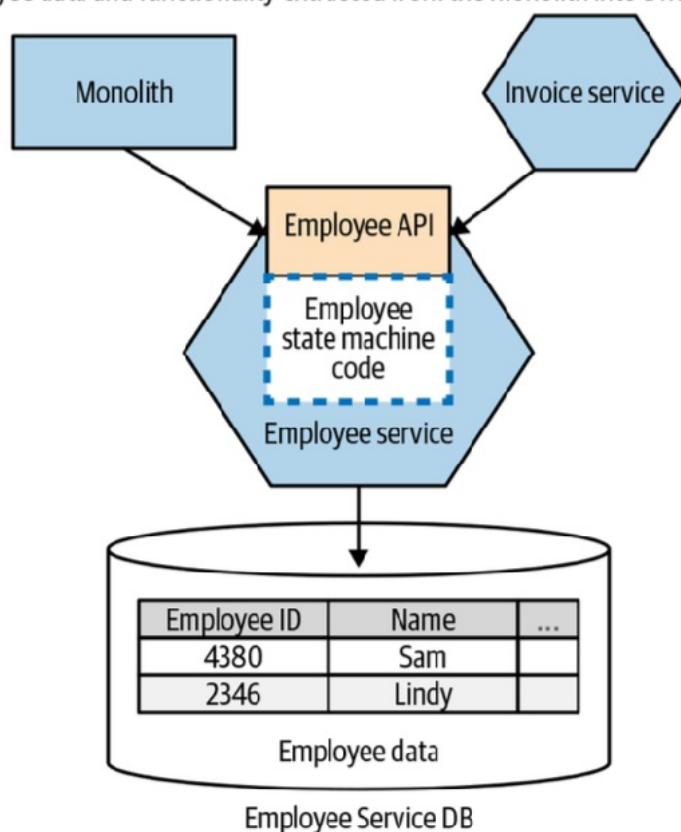


图4-31. 使用Employee API识别要从单体中拆分的Employee服务的服务边界

## 何处使用该模式

当管理数据的代码仍在单体中时，把单体作为数据访问层的方法最有效。正如我们之前所讨论的，在涉及数据时，一种看待微服务的方法是：把微服务看成是封装状态和管理状态转换的代码。因此，如果仍在单体中提供此数据的状态转换，则意味着微服务需要经过单体中的状态转换来访问（或修改）数据状态。

如果在单体数据库中尝试访问的数据确实应由微服务“拥有”，那么我更倾向于建议跳过此模式，而是选择把数据拆分出去。

## 多schema存储

正如我们已经讨论过的，“不要让不好的情况变得更糟糕”是一个好主意。如果我们仍在直接使用数据库中的数据，这并不意味着微服务存储的新数据也应放在该数据库中。[图4-32](#)展示了一个Invoice服务的例子。发票的核心数据仍然存在单体中，我们当前从单体中访问发票的核心数据。我们向Invoices服务增加了审核的功能；审核功能相当于不在单体中的、全新的功能。为此，我们需要将审核者存储在一个表中，并把员工对应到到Invoice ID。如果把这张新表放到单体里，我们会扩张现有的数据库！相反，我们把这些新数据存储到微服务自己的schema中。

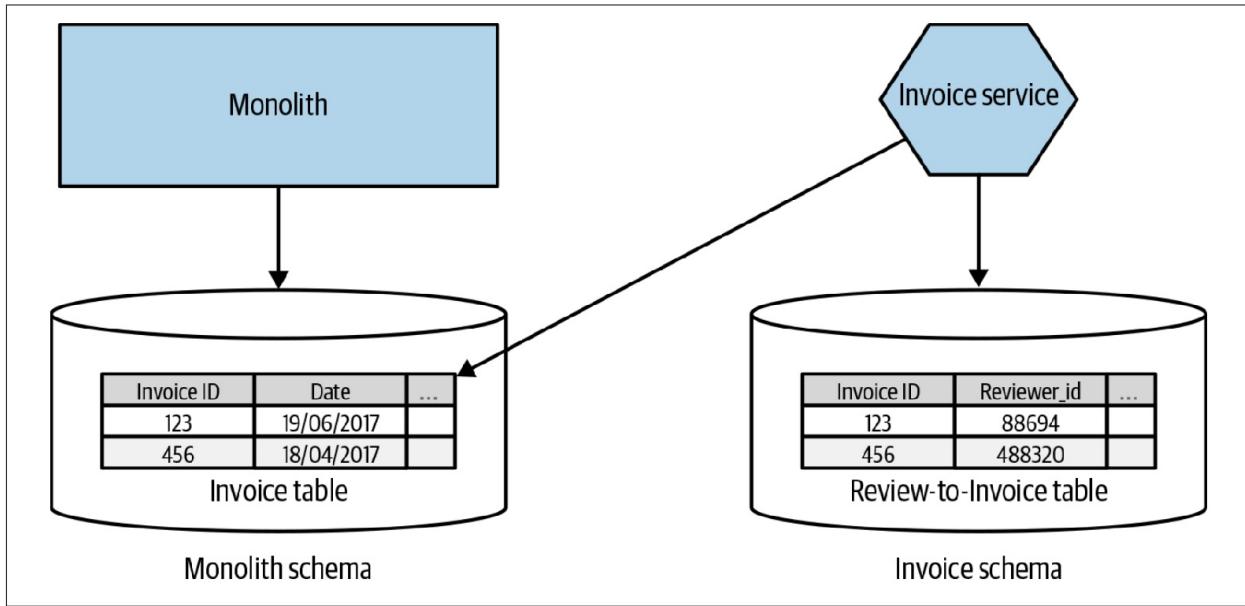


图4-32. Invoice服务把新数据存于自己的schema中，但是仍然会直接访问单体中的老数据

在图4-32的例子中，我们必须考虑：当外键实际上跨越schema边界时，会发生什么。在本章的后面，我们将更深入地探讨这个问题。

从单体数据库中提取数据将花费一些时间，而且这个过程也不是一步就能完成。因此，微服务既访问单体数据库中的数据，同时又管理其自己的本地存储，对于这种情况，我们应该感到高兴。当我们设法从单体中清除其余数据时，可以一次将其迁移到新schema的一个表中。

## 何处使用该模式

当向微服务添加需要存储新数据的全新功能时，此模式很好用。显然，新增加的数据并不是单体所需的数据（新的功能不存在于单体中），因此，开始时就让新数据从单体数据库中分离出来。当把数据从单体移到微服务自己的schema中时，此模式也很有意义——这个过程可能需要一些时间。

如果我们所访问的数据位于单体schema中，并且我们从未计划将其转移到微服务自己的schema中，我强烈建议将该模式与把单体作为数据访问层（参见166页的[把单体作为数据访问层](#)）的方法结合使用。

# 同时拆分数据库和代码

当然，从分阶段的角度来看，我们可以选择在一个较大的步骤中完成拆分，如图4-33所示。此时，我们同时拆分代码和数据。

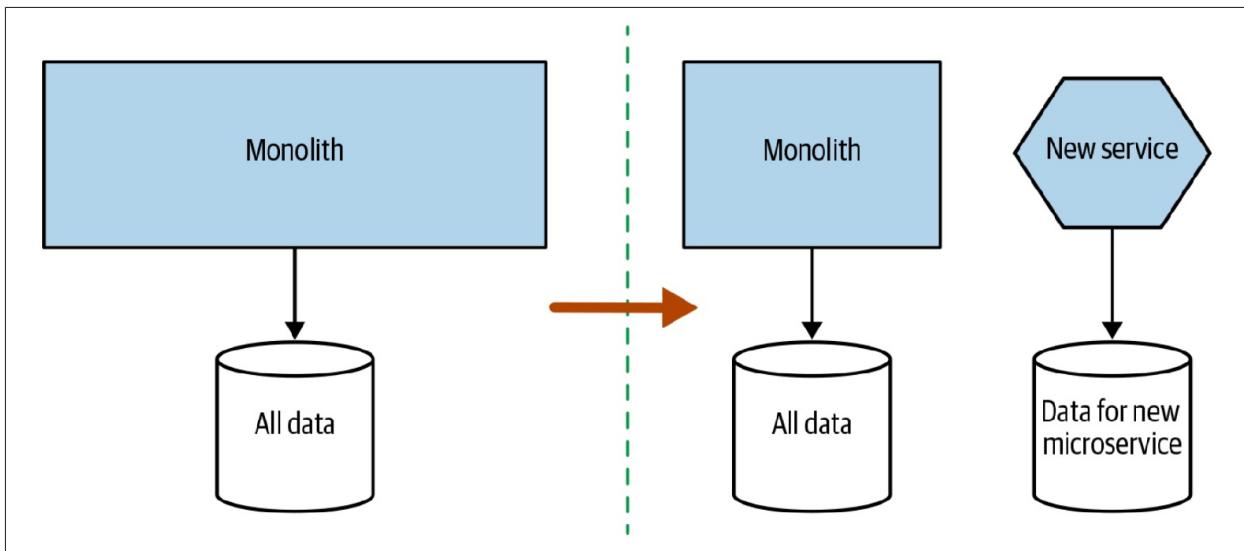


图4-33. 在一个步骤中即拆分代码又拆分数据

我所担心的是，这种拆分需要迈出更大的一步，并且需要更长的时间才能评估结果。我强烈建议避免使用这种方法，而应先拆分数据层或应用程序层。

# 那么，我应该先拆分什么呢

我明白：你可能已经厌倦了所有这些“取决于”的东西，对吗？我不能怪你。问题是，每个人的情况都不一样，所以我想为你提供足够的上下文信息，并讨论各种利弊，以帮助你下定决心。但是，我知道，有时人们并不想在这些事情上过多思考，而只是想要一个方法而已，所以就有了这一节的内容。

## a hot take

正如在维基百科上的解释：在新闻业中，a hot take是“针对新闻报道的”蓄意挑衅性评论，这种评论“通常是在紧迫的期限内撰写的，很少进行研究或报道，甚至更少地思考”。

如果可以修改单体，并且担心对性能或数据一致性的潜在影响，我会先拆分数据。否则，我将先拆分代码，并使用代码的拆分来帮助我理解代码对数据所有制的影响。但重要的是，我们也要为自己思考，并考虑在特定情况下会影响决策过程的所有因素。

Copyrights © wangwei all right reserved

# 数据分离的例子

到目前为止，我们已经在较高的层次上研究了数据分离，但是与数据库拆分相关的挑战非常复杂，还有一些棘手的问题需要探索。现在，我们将研究一些更底层的数据拆分模式，并探讨其可能产生的影响。

## 关系型数据库 VS 非关系型数据库

本章详细介绍的许多重构例子都在探讨使用关系数据库时遇到的挑战。关系型数据库的性质给拆分数据带来了额外的挑战。许多人可能正在使用非关系型数据库。即使使用非关系型数据库，以下的许多模式仍然适用。对于如何进行变更而言，我们的约束可能会更少，但是我仍然希望这些建议是有用的。

Copyrights © wangwei all right reserved

# 拆分表

有时，会在一个表中发现需要跨越两个或多个服务边界的数据，并且这可能会变得很有趣。在图4-34中，有一个共享表——Item表，Item表不仅存储所售商品的信息，还存储库存相关的信息。

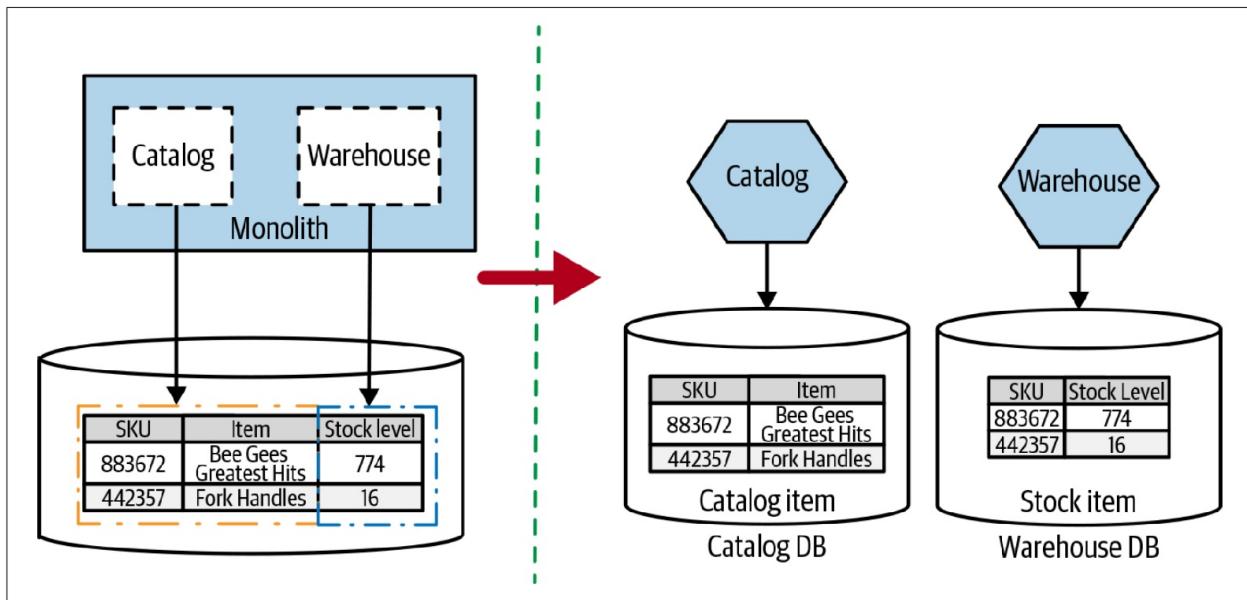


图4-34. 桥接两个界定的上下文的单个表

在该例子中，我们希望把Catalog和Warehouse拆分为新服务，但这两个服务的数据却混合存储在一个表中。因此，我们需要把数据拆分为两个表，如图4-34所示。本着增量迁移的精神，在拆分schema之前，应先在现有的schema中拆分表。如果这些表位于单个schema中，则有必要声明 `Stock.SKU` 列到 `Catalog.SKU` 的外键。但是，由于我们计划将这些表最终迁移到单独的数据库中，此时，我们将不会用一个数据库来确保数据的一致性（我们将在短期内更详细地探讨这一想法），因此我们可能不会从创建外键中获得太多收益。

这个例子很简单，可以轻松地实现按照列来分离数据。但是，当很多代码都会更新同一列时，会发生什么？在图4-35中，我们有一个Customer表，其中包含了Status列。

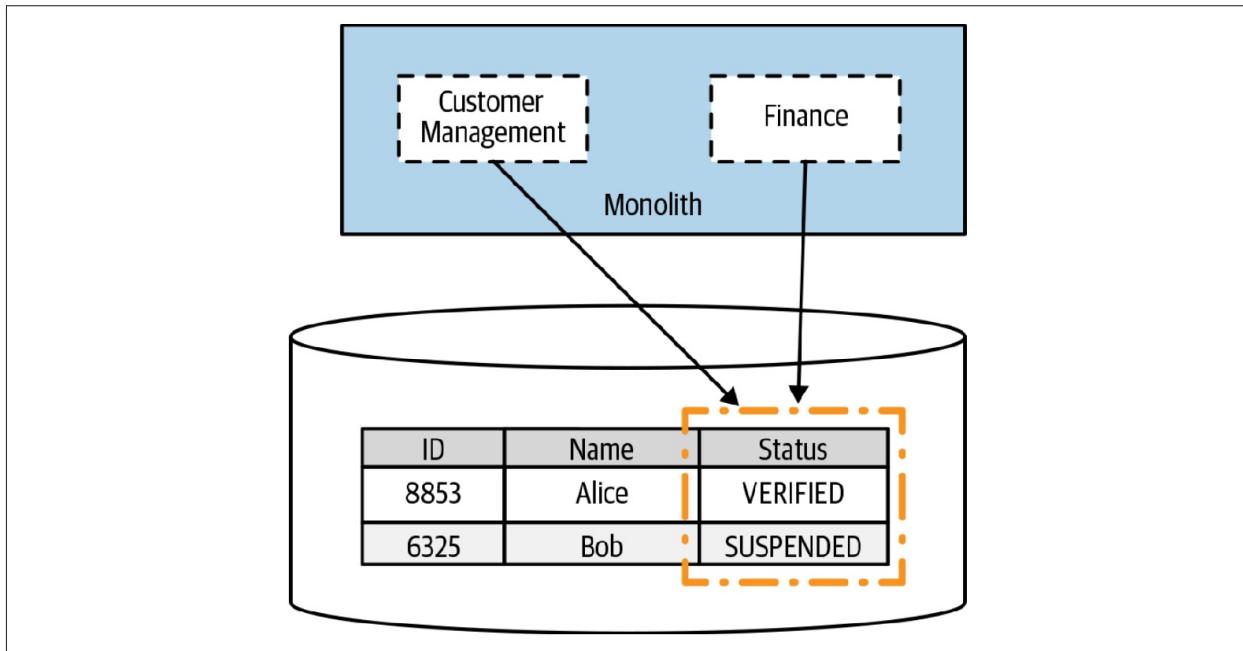


图4-35. Customer Management和Finance的代码都会修改Customer表中的Status

客户在注册过程中会更新 `status`，以标记客户是否已经验证了其email。此时，`status` 的值从 `NOT_VERIFIED` 变为 `VERIFIED`。验证后，客户便可以购物。如果客户未付款，财务模块会暂停客户的业务并把客户的 `status` 更新为 `SUSPENDED`。在这种情况下，客户的 `status` 看起来仍然是客户域模型的一部分，因此，`status` 还应该由将来的Customer服务来管理。记住，在可能的情况下，我们希望把域实体（domain entity）的状态机保留在单个服务边界内。对于客户而言，更新状态看起来肯定是其状态机的一部分！这意味着服务拆分之后，我们的新的Finance服务需要调用Customer服务以更新此状态，如图4-36所示。

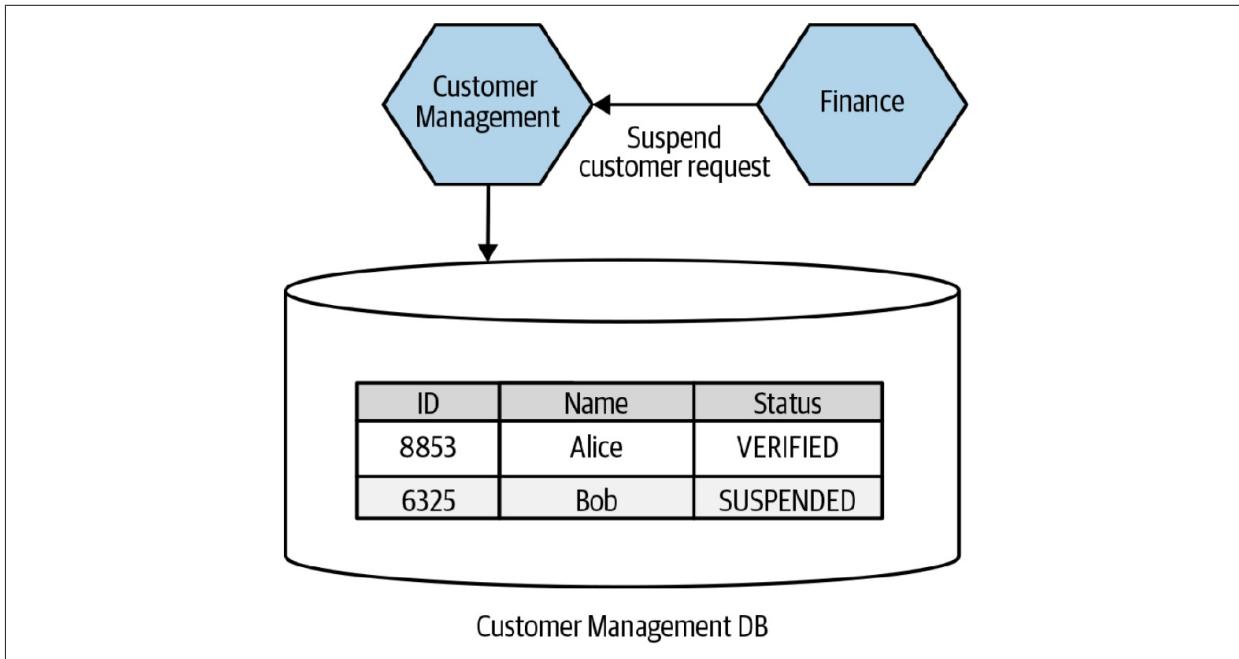


图4-36. 新的Finance服务需要进行服务调用以暂停客户的业务

像这样来拆分表的一个大问题是，我们失去了数据库事务赋予我们的安全性。本章的末尾，在[187页的事务](#)和第[193页的Sagas](#)中，我们将更深入地探讨该主题。

# 何处使用该模式

从表面上看，拆分表似乎很简单。当表由当前单体中的两个或更多界定的上下文所拥有时，需要对齐这些界定的上下文来拆分表。如果发现代码库的多个部分会更新表中的特定列，则需要判断：谁应该“拥有”该数据。还要明确“拥有”该数据的代码是现有的领域概念吗？这些判断将有助于确定该数据的位置。

Copyrights © wangwei all right reserved

# 把表的外键关系移至代码

我们已决定抽取Catalog服务，该服务可以管理并开放有关艺术家，曲目和专辑的相关信息。当前，单体中与catalog相关的代码会使用 `Albums` 表来存储有关我们可以发售的CD的信息。最终，`Ledger` 表会引用这些专辑并跟踪专辑的销售信息，如图4-37所示。`Ledger` 表中的每一行仅记录了每张CD的销售额以及所售商品的 `ID`。在该例中，我们把 `ID` 称为 `SKU` (*a stock keeping unit*)，这是零售系统中的常见做法。

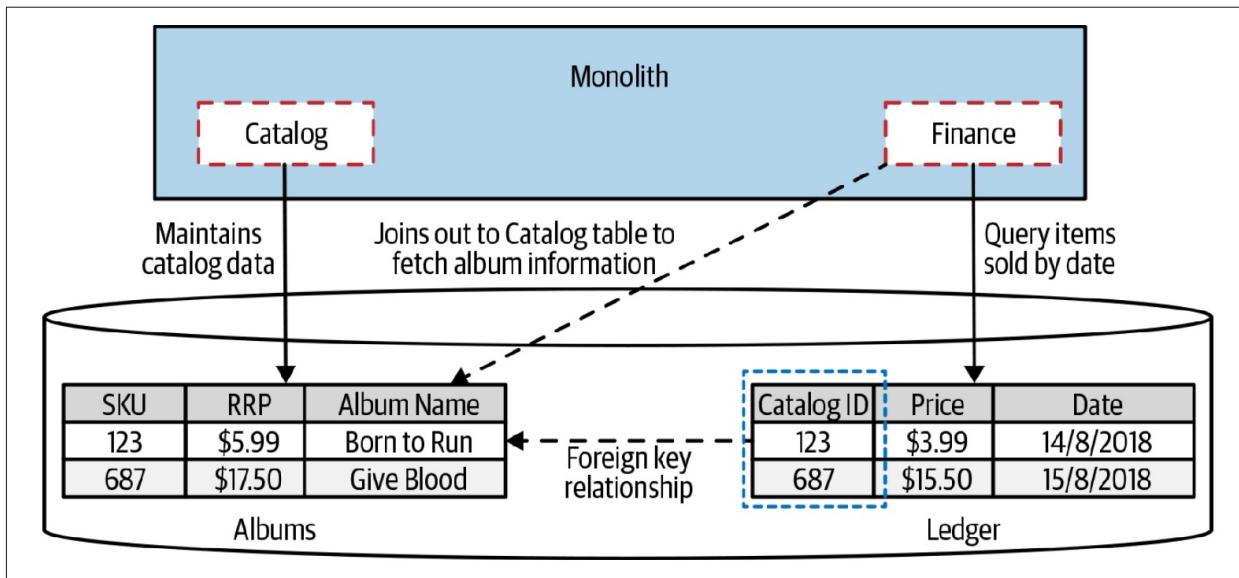


图4-37. 外键关系

每个月底，我们需要产出一份报告来概述我们最畅销的CD。`Ledger` 表可以帮助我们来了解哪个 `SKU` 的销售量最大，但是有关该 `SKU` 的信息却在 `Albums` 表。我们希望使报告易于阅读，因此，与其说“我们卖出了400张 `SKU 123` 并获得\$1,596的销售额”，不如说“我们卖出了400张Bruce

Springsteen的Born to Run，并获得\$1,596的销售额”。为此，财务相关代码所触发的数据库查询需要连接 `Ledger` 表和 `Albums` 表的信息，如图4-37所示。

我们在schema中定义了外键，因此我们标识 `Ledger` 表中的行和 `Albums` 表中的行存在关系。通过定义这样的关系，底层数据库引擎能够确保数据的一致性，即：如果 `Ledger` 表中的行引用了 `Albums` 表中的行，我们知道 `Albums` 表肯定存在对应数据。在我们的场景中，这意味着，我们始终都可以获得所售专辑的相关信息。这些外键关系还使数据库引擎可以进行性能优化，以确保join操作尽可能快。

我们希望将Catalog和Finance代码拆分到各自的相应服务，这意味着数据也必须随服务一起拆分。最终，`Albums` 表和 `Ledger` 表将处于不同的schema中，那么，我们的外键将会如何？好吧，我们要考虑两个关键问题。

- 首先，当我们的新的Finance服务将来要产出此报告时，如果不能再通过数据库join来获取与Catalog相关的信息，它将如何检索这些信息？
- 另一个问题，对于新的架构而言，可能存在数据不一致的事实，我们该怎么办？

# 消除join操作

让我们先替换数据库的join操作。在单体系统中，为了连接 `Album` 表中的行与 `Ledger` 表中的销售信息，我们需要数据库执行连接操作。我们将执行一个SELECT查询，并在该语句中join `Albums` 表。这需要调用单个数据库来执行查询并获取我们所需的所有数据。

在我们基于微服务的新架构中，由Finance服务来产出畅销书报告，但该服务却没有本地的专辑数据。因此，Finance服务需要从新的Catalog服务获取专辑数据，如图4-38所示。生成报告时，Finance服务首先查询 `Ledger` 表，提取上个月最畅销的SKU列表。至此，我们所拥有的只是一个SKU列表，以及每个SKU的销售量。这是我们在本地拥有的唯一信息。

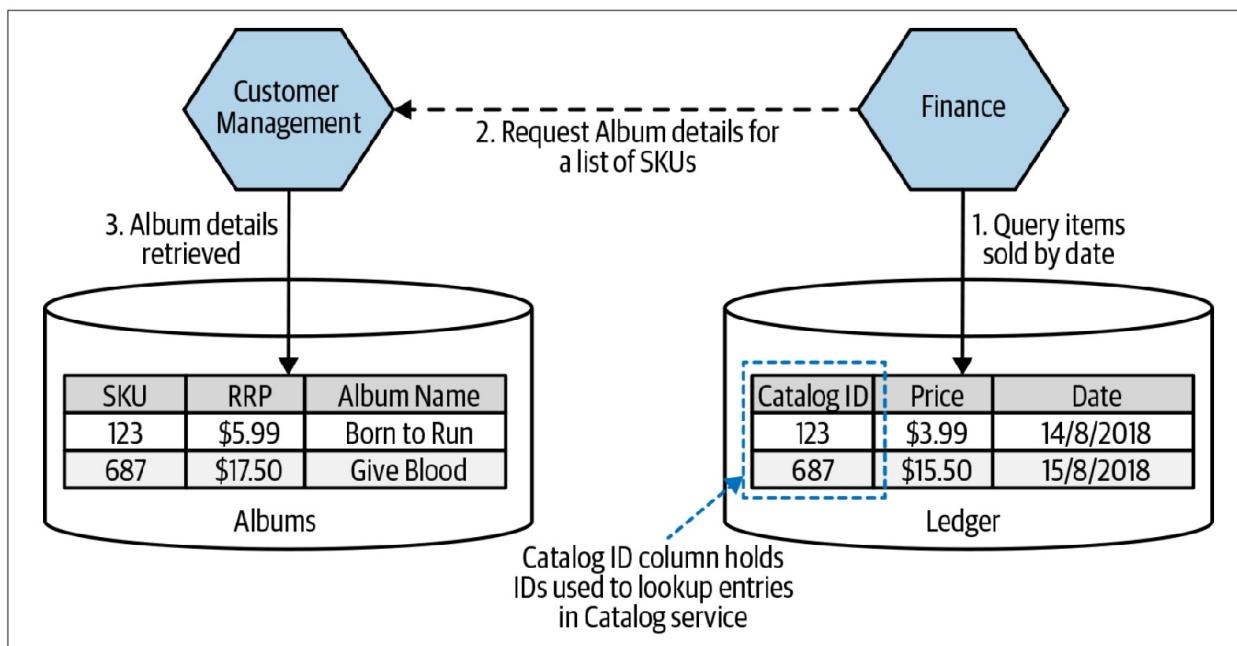


图4-38. 使用服务调用来替换数据库的join操作

接下来，我们需要调用Catalog服务，以请求每个SKU的信息。此请求将导致Catalog服务在其自己的数据库上执行本地SELECT。

从逻辑上讲，仍然存在join操作，但现在，join操作位于Finance服务内部，而不是数据库。不幸的是，几乎没有提高任何效率。我们已经从一个世界进入到另一个全新的世界。从只有一条SELECT语句的世界，到需要两条SELECT语句的世界：先对 `Ledger` 表执行SELECT查询，接着调用Catalog服务，而对Catalog服务的服务调用又触发了对 `Albums` 表执行SELECT语句，如[图4-38](#)所示。

在这种情况下，如果此操作的整体延迟没有增加，我会感到非常惊讶。对于该例而言，该报告每月产生一次，并可能会缓存该报告，因此延迟可能不是什么大问题。但是，如果这是一个频繁的操作，那可能会有更多问题。通过在Catalog服务中批量查找SKU，或者甚至在Finance服务本地缓存所需的专辑信息，我们可以降低延迟增加带来的影响。

最终，延迟的增加是否是一个问题，只能由我们自己决定。我们需要了解关键操作可接受的延迟，并能够衡量当前的延迟。此时，像Jaeger这样的分布式系统可以给我们提供帮助，其提供了精确获得跨服务的操作时间的能力。即使会降低服务的操作的速度，但其速度仍然足够快，则使其速度变慢是可以接受的，尤其是在为获取其他好处而权衡取舍的情况下。

# 数据一致性

更棘手的问题是，由于Catalog服务和Finance服务是相互分离的服务，并且各自的schema也相互分离，因此，最终可能会导致数据不一致。使用单个schema时，如果 `Ledger` 表中存在对 `Albums` 表中行的引用，则将无法删除 `Albums` 表中的对应行。数据schema会强制保持数据一致性。但是，在新的架构中，不存在这样的强制手段。此时，我们有什么选择？

## 在删除数据之前进行检查

第一个方案是，从 `Albums` 表中删除记录时，确保与Finance服务进行核对，以保证该记录尚未被引用。问题是，我们很难保证可以正确执行此操作。假设我们要删除 `SKU 683`。我们会调用Finance服务并要求该服务确认：是否在使用 `SKU 683`。Finance服务给出未使用此记录的响应。然后，我们在 `Albums` 表中删除该记录，但是在执行时删除操作时，Finance服务创建了对 `SKU 683` 的引用。为了阻止这种情况的发生，我们需要暂停在 `SKU 683` 上创建新的引用，直到完成数据删除为止。此时，可能需要加锁，也可能会面临分布式系统中所有的挑战。

检查数据是否处于使用中的另一个问题是，创建了Catalog服务的反向依赖。现在，需要检查任何使用了 `Albums` 数据的其他服务。即使只有一个服务使用了 `Albums` 的信息，这也已经糟糕透顶了；更何况，随着该数据的消费者越来越多，情况会变得更加糟糕。

我奉劝各位不要考虑该方案，因为很难确保正确的执行此操作，并且该方案还会带来的高度的服务耦合。

## 优雅的处理删除

更好的方案是让Finance服务处理以下事实：Catalog服务可能不会以优雅的方式在 `Album` 表中存储信息。如果我们查不到给定的 `SKU`，则可以简单地显示“专辑信息不可用”。

在这种情况下，当我们请求一个曾经存在的 `SKU` 时，Catalog服务需要告诉我们该 `SKU` 曾经存在，但是现在不再可用。例如，如果使用HTTP，则可以使用 `410 GONE` 作为响应。`410` 和常用的 `404` 并不一致。`404` 表示找不到所请求的资源，而 `410` 则表示请求的资源曾经可用，但现在不再可用。这种区别很重要，尤其是在追踪数据不一致问题时！即使不使用基于HTTP的协议，也要考虑是否可以从支持类似响应的方案中受益。

如果我们想真正进步，在删除Catalog数据时，我们要确保通知到Finance服务，也许可以采用订阅事件的方案。当收到Catalog的删除事件时，我们可以把现在删除的Album信息复制到Finance的本地数据库中。在这种特定情况下，这听起来像是矫枉过正；但在其他的情况下，该方法可能会有用，尤其是在我们要实现一个分布式状态机，以执行诸如跨服务边界的级联删除之类的操作时。

## 不允许删除

确保我们不会在系统中引入太多的数据不一致的另一种方法是：简单地不允许删除Catalog服务中的数据。如果在现有系统中删除某件商品只是类似于确保该商品无法出售，我们可以实施软删除。为此，我们可以使用status把该数据标记为不可用，甚至可以将其移动到专用的“graveyard”表<sup>5</sup>。在这种情况下，Finance服务仍可以请求专辑数据。

## 那么，该如何处理删除

总的来说，我们创建了一个在单体系统中不存在的故障模式。不同的解决方案可能以不同的方式影响我们的用户，因此在寻找解决方案时，我们必须考虑用户的需求。因此，选择正确的解决方案需要了解我们的特定背景。

就我个人而言，在这种特定的情况下，我可能会通过两种方式解决此问题：

- 不允许删除Catalog中的专辑信息
- 确保Finance服务可以处理已经删除的数据

人们可能会争辩说，如果Catalog服务不能删除数据，那么Finance服务的查询将永远不会失败。但是，作为故障的结果，Catalog服务可能会恢复到较早的状态，这意味着我们要查找的数据不再存在。我不希望Finance服务在这种情况下陷入困境。当然，似乎不太可能出现这种情况，但我一直在寻求构建弹性的系统，并且需要考虑——如果调用失败了，该怎么办？在Finance服务中优雅地处理此问题似乎很容易。

# 何处使用该模式

当开始考虑有效解除数据库的外键时，需要确保的第一件事是：不能拆分那些实际上是一体的东西。如果我们担心我们正在拆分一个聚合，请暂停拆分并重新考虑。在此处的 `Ledger` 和 `Albums` 的例子中，显然，我们有两个独立的聚合，并且它们之间具有关系。但是，请考虑另一种情况：`Order` 表，在 `Order Line` 表中关联的多行数据包含了我们已订购商品的详细信息。如果我们把订单拆分为单独的服务，则会遇到数据完整性的问题。实际上，订单项（*order lines*）是订单本身的一部分。因此，我们应该将它们视为一个整体，如果我们想把 `Order` 表迁移出单体，则应该同时迁移 `order` 表和 `orderLine` 表。

有时，从单体schema中多抽取一些内容，我们可以同时移动存在外键关系的双方，从而使我们的工作变得更加轻松！

# 共享静态数据的例子

静态引用数据（这些数据不经常修改，但通常很关键）会带来一些有趣的挑战，并且我已经看到了多种方法来管理静态引用数据。通常，会把静态引用数据存储在数据库。在为Java项目编写了自己的 `StringUtil` 类之后，我可能会将该类用到的国家代码（*country codes*）存储在数据库中，如图4-39所示）。

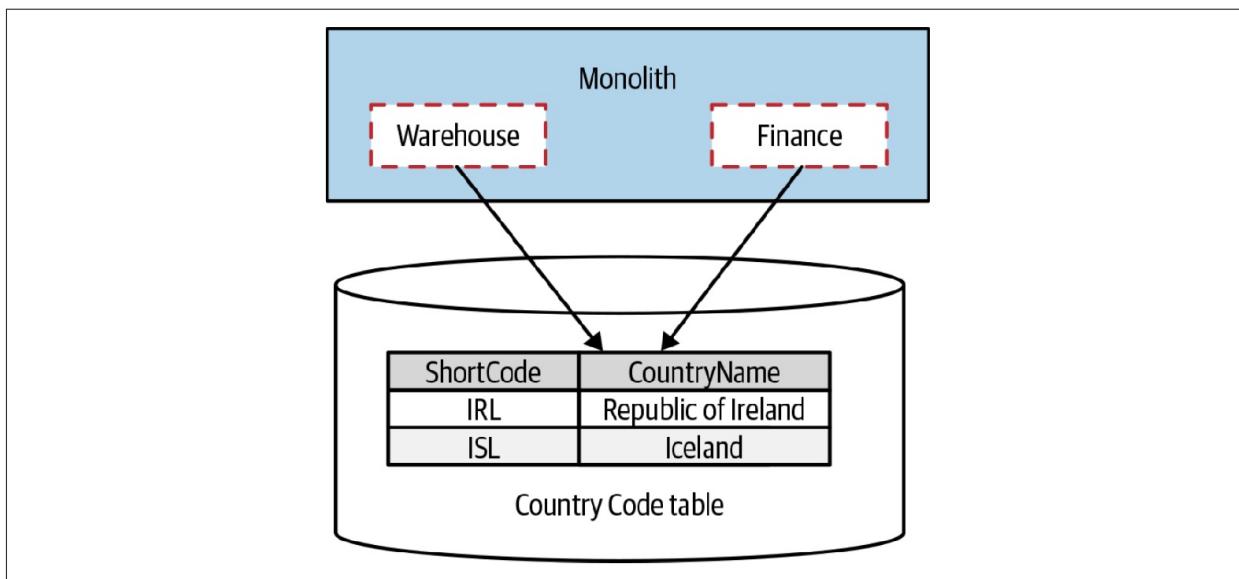


图4-39. 数据库中的国家代码

我一直在问：为什么像国家代码这样的、不经常变化的小量数据需要放在数据库，但是无论是出于什么底层原因，这些存储在数据库中的共享静态数据的例子还是很多。那么，当我们的音乐商店的多个部分的代码都需要访问相同的静态引用数据时，我们该怎么办呢？好吧，事实证明，此时，我们有很多选择。

## 静态引用数据的副本

为什么不像图4-40一样，让每个服务都拥有自己的数据副本？这个观点可能会让许多人大吃一惊。数据副本？你疯了吗？听我说完！没有想象的那么疯狂。

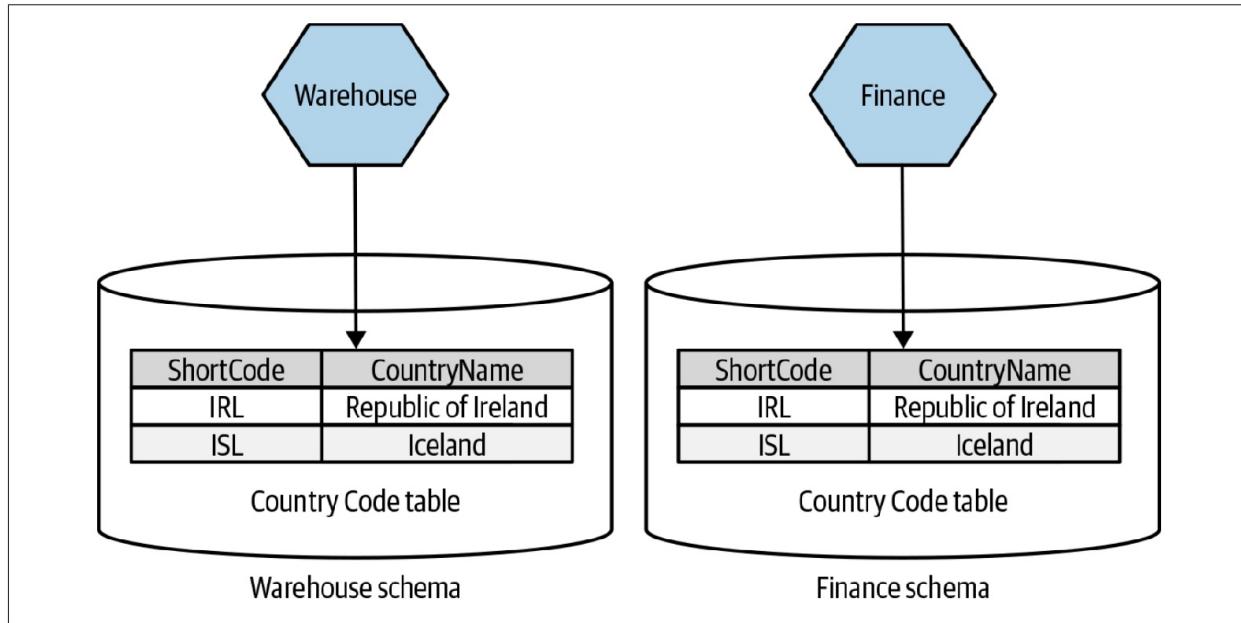


图4-40. 每个服务都拥有其自己的Country Code表

对数据副本的担忧可以归结为两件事：

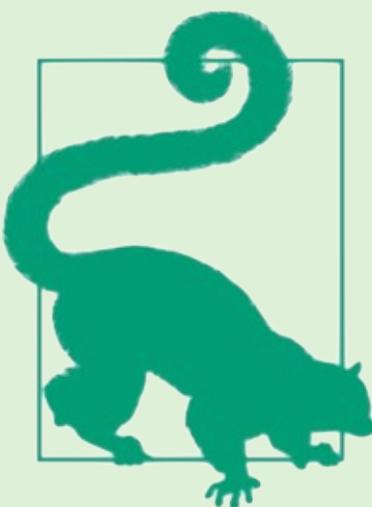
- 首先，每次需要修改数据时，必须在多个地方进行修改。但是在这种情况下，数据的修改频率是多少呢？随着南苏丹(简称SSD)的成立，最后一次国家成立并得到官方认可的时间是2011年。所以我认为修改频率不是什么大问题，对吧？
- 更大的担忧是，如果数据不一致会发生什么？例如，Finance服务知道南苏丹是一个国家，但令人费解的是，Warehouse服务的数据没有更新，其对南苏丹一无所知。

数据不一致是否是问题，取决于如何使用数据。在我们的例子中，考虑Warehouse服务使用国家代码数据来记录cd的生产地。结果，我们没有储存任何南苏丹制造的cd，所以，我们丢失这些数据的事实并不是问题。另一

方面，Finance服务需要国家代码信息来记录销售信息，我们有南苏丹的用户，所以，我们需要更新信息。

当数据仅在每个服务内使用时，数据不一致不是问题。回想一下我们对界定的上下文的定义：界定的上下文是关于隐藏在边界内的信息。另一方面，如果数据是这些服务之间通信的一部分，那么，我们就会有不同的担忧。如果Warehouse服务和Finance服务需要相同的国家信息数据，那么，我肯定会担心数据副本的这种特性。

当然，我们也可以考虑使用某种后台进程来保持这些副本的同步。在这种情况下，我们不太可能保证所有的副本都是一致的。但是，如果我们的后台进程运行的足够频繁(和快速)，那么我们就可以减少潜在的数据不一致窗口，这就足够了。



作为开发人员，当我们看到数据副本的时候，我们通常会做出糟糕的反应。我们担心管理信息副本的带来的额外成本，更担心数据是否会出现不一致性。但有时候，两害相权取其轻。如果我们可以避免引入耦合，接受数据的副本就是一个明智的取舍。

## 何处使用数据副本模式

该模式应该很少使用，大家应该更喜欢我们稍后提及的一些选择。对于大量数据而言，当并非所有服务都必须查看完全相同的数据集时，该模式有时是有用的。在英国，像邮政编码文件这样的文件可能是一个不错的选择，可以定期更新邮政编码到地址的映射。邮政编码文件是一个相当大的数据集，以代码的形式管理该数据可能会很痛苦。如果想直接使用邮政编码数据，这可能是选择数据副本模式的另一个原因。但是，坦诚而言，我不记得自己曾经用过这种模式！

## 专门的引用数据

如果确实想要国家代码的唯一数据源，可以将该数据重定向到一个专用的 schema，或许为所有静态引用数据而预留，如图4-41所示。

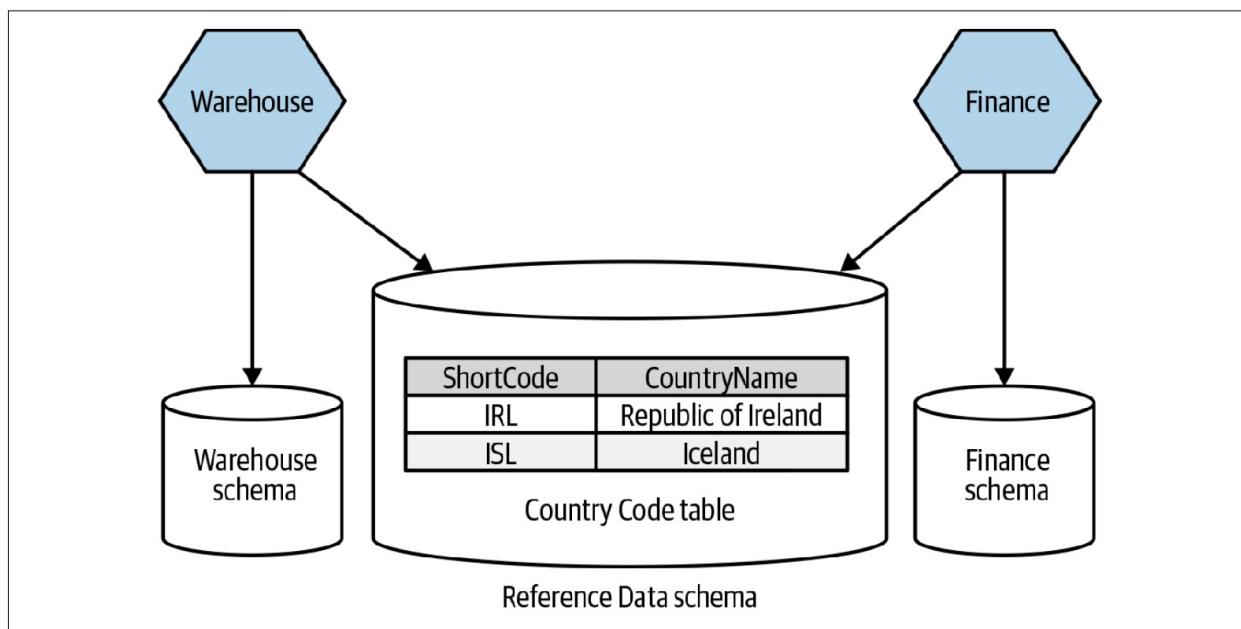


图4-41. 使用专门的共享schema作为引用数据

我们必须考虑共享数据库的所有挑战。在某种程度上，数据的性质会或多或少的抵消对数据耦合和修改的担忧。这些数据很少修改，而且结构简单，因此，我们可以更容易地将这个引用数据看作是一个已定义的接口。在这种情况下，我将把引用数据作为它自己的版本化实体来管理，并确保人们理解：schema的结构代表了对消费者的一个服务接口。对这个schema的破坏性修改可能是痛苦的。

在一个schema中保存这些共享数据，确实为服务提供了如下的机会：服务仍然可以将该数据用于本地数据的join查询。但是，要做到这一点，可能需要确保schema位于相同的底层数据库引擎。这除了潜在的单点故障之外，还增加了从逻辑存储映射到物理存储的复杂性。

## 何处使用专门的引用数据

该方法有很多优点。我们避免了使用数副本的担忧，而且数据的格式极有可能不会改变，因此我们对数据耦合的担忧也得到了缓解。对于大量数据，或者，当需要跨schema执行join操作时，该模式是一种有效的方法。记住，对schema格式的任何修改都可能对多个服务造成重大影响。

## 静态引用数据的library

当仔细检查时，国家代码数据并没有多少记录。假设使用ISO标准国家代码，我们也只看到了249个国家<sup>6</sup>。这非常适合在代码层存储这些数据，或许可以作为一个简单的静态枚举类型。事实上，我曾经多次以代码的形式存储少量静态引用数据，我也见过在微服务架构中应用该方法的案例。

当然，如果没有必要的话，我们宁愿不复制这些数据，因此，这致使我们考虑将这些数据放在一个库中，任何需要这些数据的服务都可以静态链接这个库。美国时装零售商Stitch Fix经常使用这样的共享库来存储静态引用

数据。

Randy Shoup博士——Stitch Fix公司的技术VP说过，这种技术对于那些体量较小并且又不经常改变甚至根本就不会改变(如果存在数据修改，会有很多针对该数据变化的预先警告)的数据最有效。考虑经典的衣服尺码——其一般尺码为XS、S、M、L、XL，或者用于裤子尺寸的裤腿内缝（裆到裤脚的长度）。

在我们的例子中，我们在枚举类型 `Country` 中定义了国家代码映射，并将其打包到一个库中，以便在我们的服务中使用，如图4-42所示。

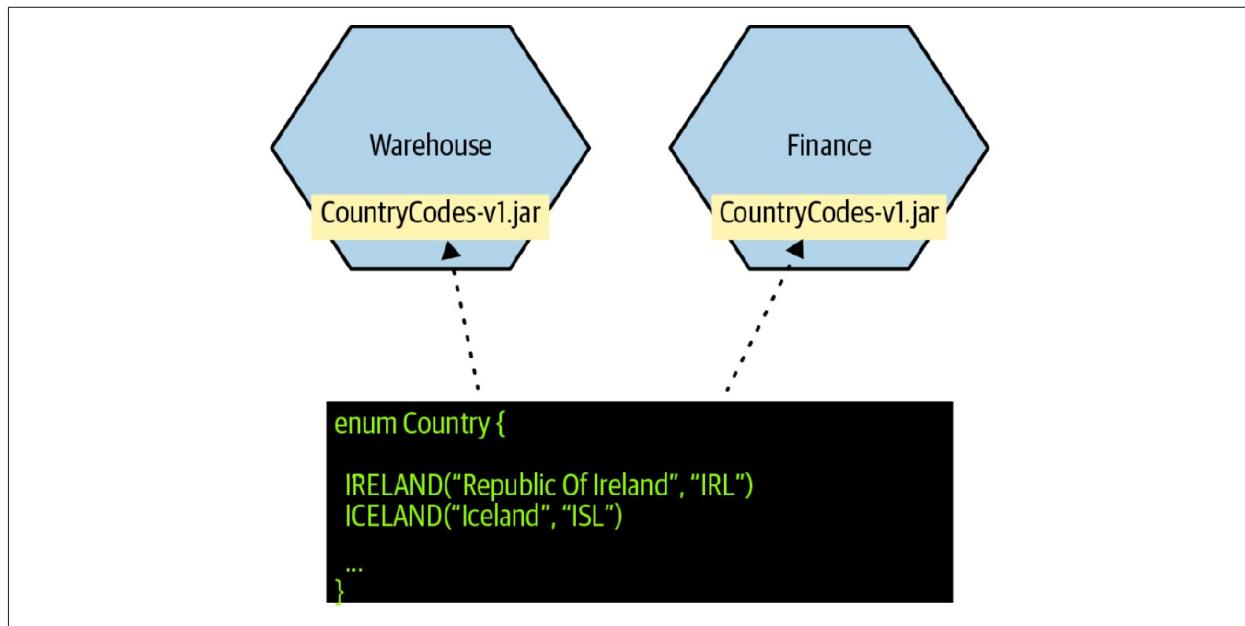
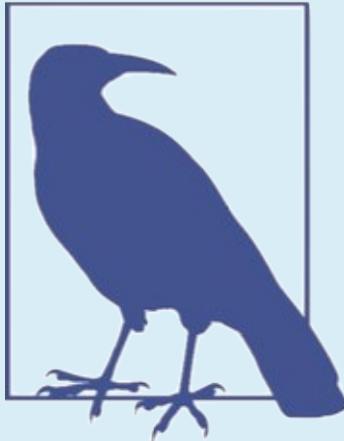


图4-42. 在多个服务共享的库中存储引用数据

这是一种简洁的解决方案，但也不是没有缺点。显然，如果混合使用多种技术栈，我们可能无法共享一个共享库。还记得“微服务”的黄金法则吗？我们需要确保微服务是可独立部署的。如果我们需要更新国家代码library，并让所有服务立即获取新数据，我们需要在新library可用时重新部署所有服务。这是一个经典的lock-step发布，而这正是我们在微服务架构中试图避免的事情。

在实践中，如果我们需要在任何地方都能获得相同的数据，那么充分的关注数据变化可能会有所帮助。Randy给出的一个例子是：需要给Stitch Fix的一个数据库增加一个新的颜色。需要将这个变化发布到使用此数据类型的所有服务，但需要大量时间来确保所有团队都获得了最新的数据版本。如果考虑国家代码的例子，如果需要添加新的国家，我们会有很多的提前通知。例如，2011年7月，南苏丹在公投之后的六个月，成为独立国家，这种通知给了我们很多时间来推行我们的变化。很少有新的国家是一时兴起就建立起来的！



如果微服务会使用共享库，请记住，我们必须接受：我们可能会在生产环境中部署不同版本的库共享库！

这意味着，如果我们需要更新国家代码库，我们需要接受这样一个事实：我们不能保证所有的微服务都使用相同版本的library，如图4-43所示。如果我们必须要求所有的微服务都使用相同版本的library，也许接下来的另一种方案可能会有所帮助。

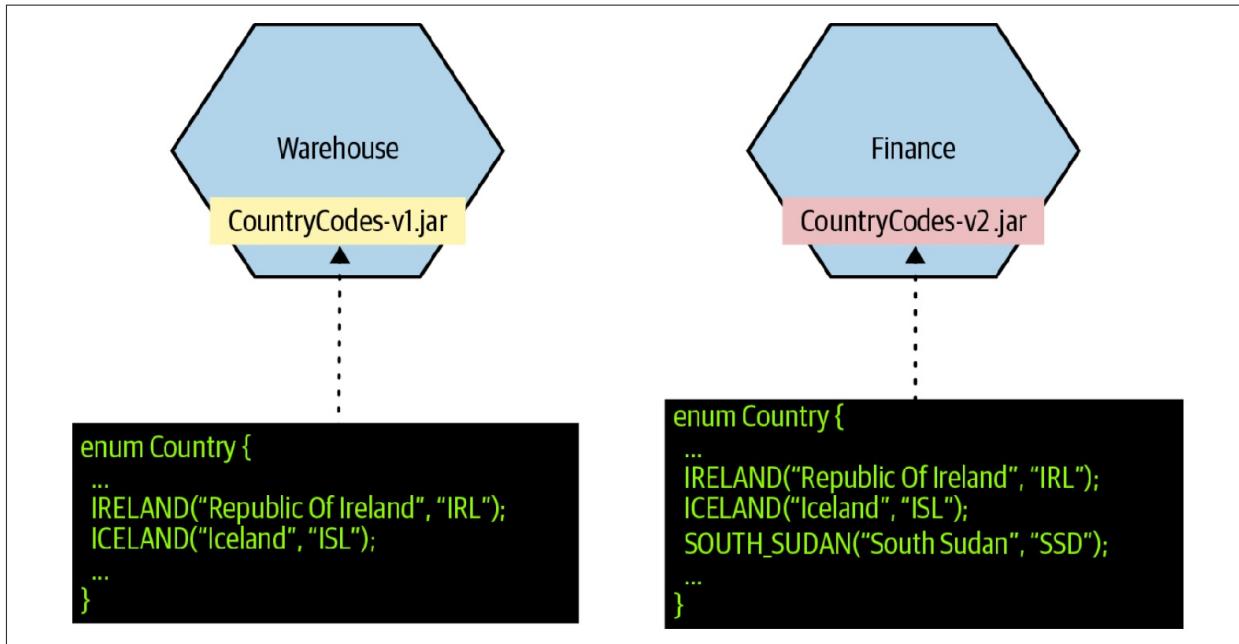


图4-43. 引用数据的共享库之间的差异可能会导致问题

在该模式的一个简单的变种中，我们把相关的数据保存在一个配置文件中，可能是一个标准的配置文件，或者，如果需要，也可以保存在更结构化的JSON中。

## 何处使用静态引用数据的library

对于体量较小的数据，我们可以轻松地让不同的服务看到该数据的不同版本，我们经常忽略这个不错的方案。关于哪个服务拥有什么版本的数据特别有用。

## 静态引用数据服务

我觉得大家能预料到这最后的一个方案。这是一本关于创建微服务的书，所以，为什么不考虑为国家代码创建一个专门的服务，如[图4-44](#)？

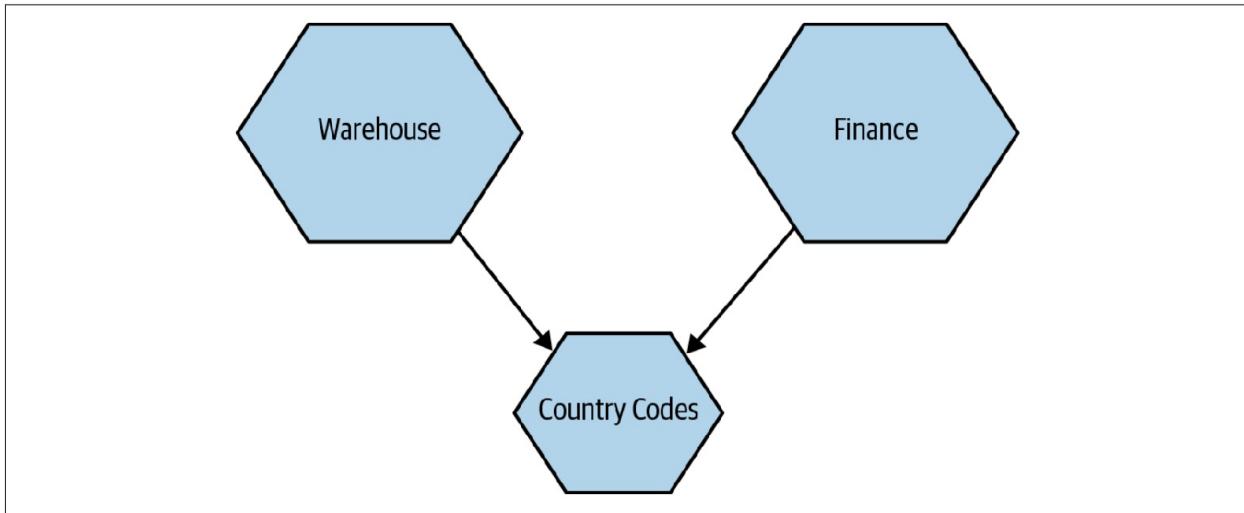


图4-44. 从一个专门的服务中提供国家代码

我曾经在世界各地的团队中经历过这种场景，对从一个专门的服务中提供国家代码的方式而言，人们的看法截然不同。有些人会立刻想：“这可能行得通！”通常，大部分人会开始摇头，并说一些诸如：“这看起来太疯狂了！”的话。当我们深入挖掘时，我们就能触及他们所担忧的核心问题；该方法看起来工作量很大，并且还可能增加了复杂性，但却没有带来多少好处。“过犹不及”（overkill）这个词经常出现！

让我们进一步探讨这个问题。当我和人们聊天并试图理解——为什么有些人能接受这个想法，而另外有些人却不能接受该想法时，事情通常归结为两点。

- 在创建和管理微服务的成本较低的环境中工作的人，更有可能考虑此方案。
- 如果创建一个新服务，即使是像国家代码这样简单的服务，都需要几天甚至几周的工作，那么，人们就会自然而然的抵制创建这样的服务。

我的前同事，同时也是O'Reilly的会员作者——Kief Morris<sup>7</sup> 向我讲述了他在英国一家大型国际银行的一个项目中的经历，这个项目花了将近一年的时间才让某些软件的第一次发布得到许可。在启动任何项目之前——从设计的签署到部署机器的准备，必须先与银行内部的10多个团队进行磋商。不幸的是，这样的经历在大型组织中并不少见。

在一些组织中，部署新软件需要大量的手工操作和批准，甚至可能需要购买并配置新的硬件。因此，创建服务的固有成本是巨大的。在这样的环境下，创建新服务时需要高度的选择性；必须提供大量的价值来证明额外的工作是合理的。此时，创建类似国家代码服务这样的事情可能会变得不合理。另一方面，如果可以在一天或更短的时间内启动一个服务模板并将其推到生产环境中，然后为我们完成创建服务的所有工作，那么我更有可能认为这是一个可行的方案。

更好的是，国家代码服务非常适合像Azure Cloud Functions或AWS Lambda这样的“功能即服务”（Function-as-a-Service）的平台。较低操作成本的功能很吸引人，而且非常适合简单的服务，比如Country Code服务。

该方法所提到的另外的担忧是：通过为国家代码增加一个服务，我们将增加另一个可能影响网络延迟的依赖。我认为，使用单独的服务的方法并不比使用专用数据库存储这些信息的方法更糟，而且可能更快。为什么呢？正如我们已经确定的，这个数据集中只有249条数据。我们的Country Code服务可以很容易地将之存储在内存中，并直接提供。我们的Country Code服务可能只是将这些数据存储在代码中，而不需要额外的数据存储。

当然，客户端也可以主动缓存这些数据。毕竟，我们通常不会向该数据中增加新数据！我们还可以考虑使用事件让消费者知道数据何时发生了变化，如图4-45所示。当数据发生变化时，相关的消费者可以通过事件获得通知，并使用该事件来更新他们的本地缓存。因为数据修改的频率较低，

因此我推测，在该场景下，传统的、基于TTL的客户端缓存已经足够好了。但是，多年前，我曾对一个通用的Reference Data服务使用了类似的方法，其效果很好。

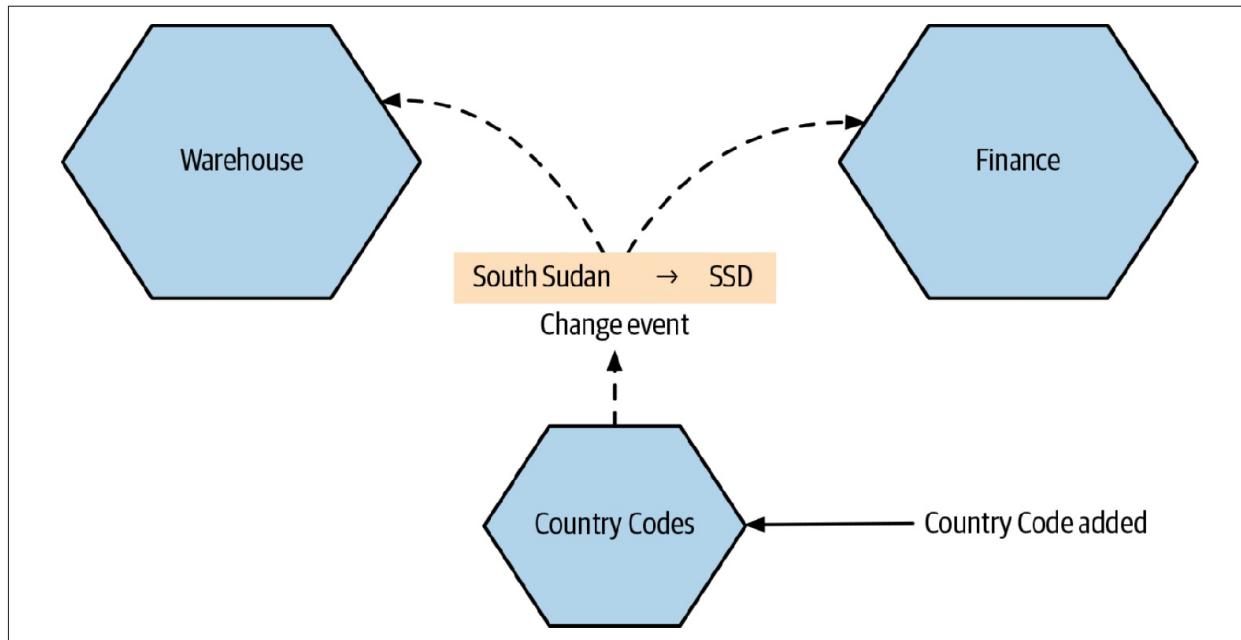


图4-45. 发送数据更新事件以允许消费者可以更新其本地缓存

## 何处使用静态引用数据服务

如果在代码中管理数据的生命周期，我就会使用这个方法。例如，如果我要开放一个API来更新这些数据，我需要在某个地方来存放这些代码，而将其放在专用的微服务中就很有意义。此时，我们有一个包含此状态的状态机的微服务。如果我们想在数据发生变化时发送事件，或者只是想为测试提供一个比桩更方便的方法时，使用静态引用数据服务也很明智。

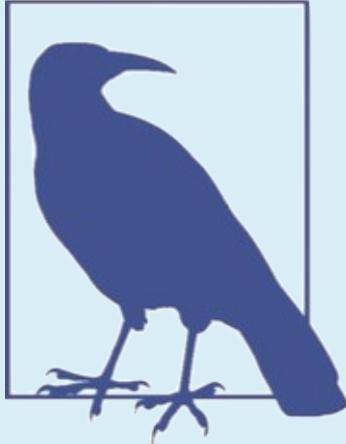
此处的主要问题似乎总是归结为创建另一个微服务的成本。创建微服务的工作是否足以证明这项工作的合理性，或者是否有更好的方法？

我该怎么做呢？

OK，我给了大家很多选择。那么，我该怎么做呢？我想，我不能永远保持中立，因此，我会给出一些建议。假设我们不需要确保所有服务的国家代码在任何时候都是一致的，那么我可能会将这些信息保存在一个共享库中。对于这类数据，把数据保存在共享库中似乎比在本地服务的schema中复制这些数据更有意义；这些数据本质上很简单，而且量级较小（国家代码、服装尺码等等）。对于更复杂的引用数据或体量更大的引用数据，可能会促使我将其放入每个服务的本地数据库中。

如果数据需要在服务之间保持一致，我会考虑创建一个专用的服务（或者可能将这些数据作为更大范围的静态引用服务的一部分而提供）。

只有当创建一项新服务的工作很难证明其是否合理时，我才可能选择这类数据的专用schema的方法。



在前面的例子中，我们介绍了一些可以帮助我们分离数据schema的数据  
库重构方法。关于这个主题的更详细的讨论，可以参考Scott J. Ambler  
和Pramod J. Sadalage所写的**Refactoring Databases** (Addison-Wesley)  
一书。

<sup>5</sup>. Maintaining historical data in a relational database like this can get complicated, especially if you need to programmatically reconstitute old versions of your entities. If you have heavy requirements in this space, exploring event sourcing as an alternative way of maintaining state would be worthwhile. ↪

<sup>6</sup>. That's ISO 3166-1 for all you ISO fans out there! ↪

<sup>7</sup>. Kief wrote Infrastructure as Code: Managing Servers in the Cloud (Sebastopol: O'Reilly, 2016). ↪

Copyrights © wangwei all right reserved

# 数据库事务

在拆分数据库时，我们已经谈到了可能会导致的一些问题。维护数据库的引用完整性会成为问题，延迟可能会增加，同时生成报表等行为会变得更加复杂。我们已经研究了应对这些挑战的各种方法，但还有一个很大的挑战：事务如何解决？

在事务中修改数据库的能力可以使我们的系统更易于理解，因此也更易于开发和维护。我们依靠数据库来确保数据的安全性和一致性，从而让我们可以有精力考虑其他事情。但是，当我们跨数据库拆分数据时，我们将失去使用数据库事务以原子方式更新数据的好处。

在探讨如何解决事务问题之前，让我们简要看一下普通的数据库事务可以为我们提供什么。

# 事务的ACID特性

当我们谈论数据库事务时，通常，我们所谈论的是具备ACID特性的事务。ACID是概述数据库事务的关键特性的首字母缩写，我们依赖ACID来确保系统数据存储的持久性和一致性。ACID代表原子性（*atomicity*），一致性（*consistency*），隔离性（*isolation*）和持久性（*durability*），这些属性给我们提供如下的能力：

- 原子性：确保事务中完成的所有操作要么全部完成要么全部失败。对我们试图做出的任何修改，如果由于某些原因而导致修改失败，则整个操作都将中止，就好像从未进行过任何修改一样。
- 一致性：修改数据库时，确保数据库处于有效、一致的状态。
- 隔离性：允许多个事务同时运行而不会产生相互干扰。确保一个事务期间进行的任何临时状态的修改对其他事务不可见，可以实现此目的。
- 持久性：交易一旦完成，需要确保我们相信，即使系统出现故障，数据也不会丢失。

值得注意的是，并非所有数据库都提供事务。我曾经使用过的所有的关系数据库都提供了事务，许多新的NoSQL数据库（例如Neo4j）也提供了事务。多年来，MongoDB仅在单个文档上支持ACID事务，对MongoDB中的多个文档进行原子更新，可能会引起问题<sup>8</sup>。

该书不是一本详细介绍事务的书。因此，为了简洁起见，我对事务的很多描述进行了简化。对于那些想进一步探索这些概念的人，我推荐《Designing Data-Intensive Applications》<sup>9</sup>这本书。在接下来的内容中，我

们将主要关注事务的原子性。这并不是说其他属性不重要，只是因为原子性往往是我们划分事务边界时遇到的第一个问题。

# 缺乏原子性的ACID

我想澄清的是，当我们拆分数据库时，仍然可以使用ACID式的事务，但是，此时，事务的范围缩小了，其用途也缩小了。参见图4-46。我们一直在跟踪新客户加入到我们系统所涉及到的过程。我们已经到达该过程的最后阶段，该阶段会将客户的状态从“PENDING”修改为“VERIFIED”。在用户完成注册后，我们还想从PendingEnrollments表中删除匹配的记录。对于单个数据库而言，可以在单个ACID数据库事务的范围内完成——要么都写入新行，要么都不写入新行。

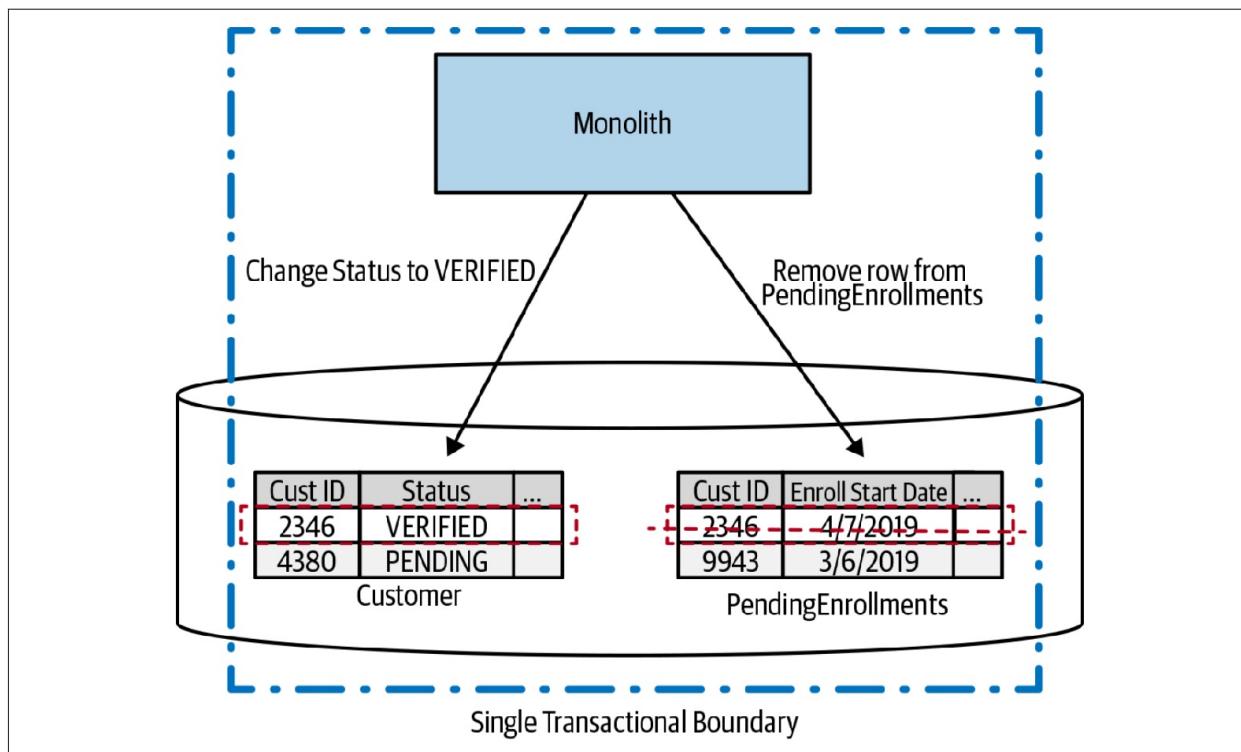


图4-46. 在一个事务的范围内更新两个表

对比图4-46和图4-47。我们在图4-47中执行完全相同的修改，但是却在不同的数据库中进行修改。这意味着要考虑两个事务，每个事务都可以独立于另一个事务而工作。

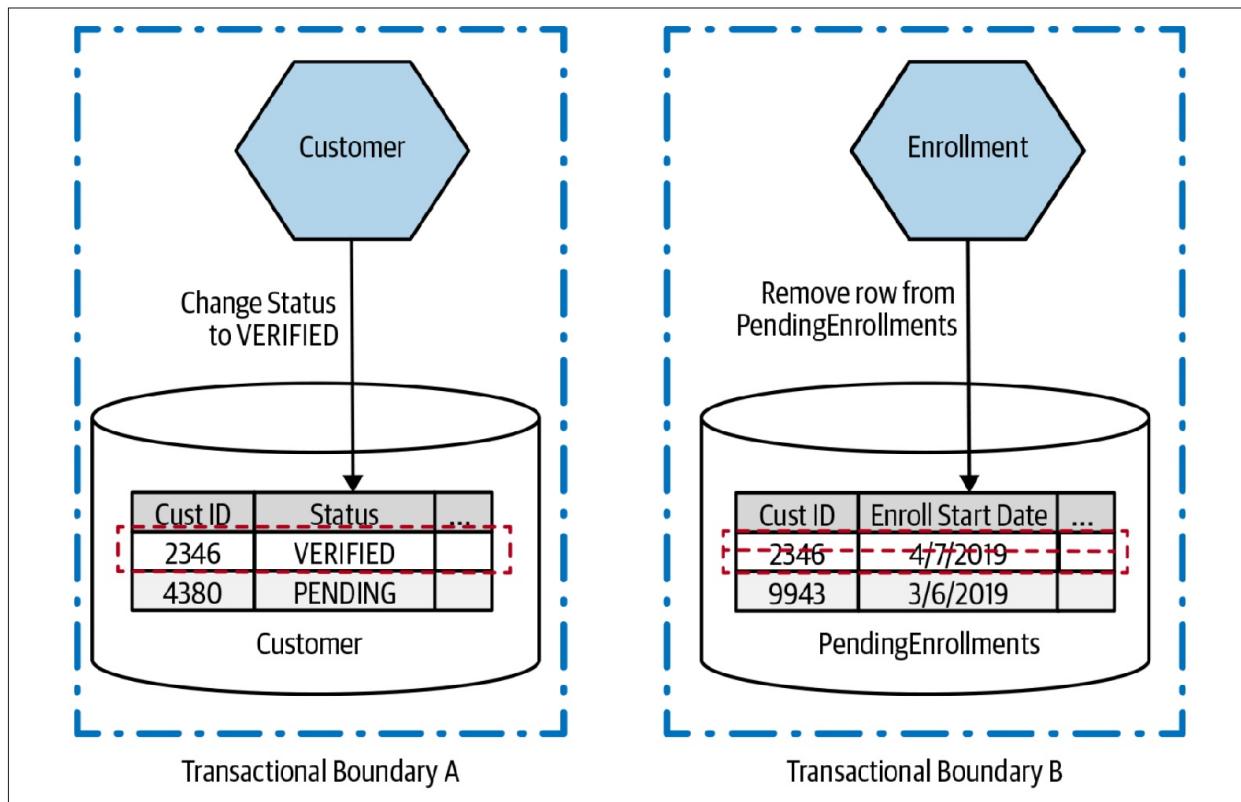


图4-47. 在两个不同事务的范围内完成数据的更新

当然，我们可以对这两个事务进行排序，只有在我们能修改Customer表中的数据时，才从PendingEnrollments表中删除对应的数据。但是，我们仍然必须考虑：如果从PendingEnrollments表中删除失败该怎么办？此时，所有的处理逻辑都需要我们自己实现。可以重新执行这些步骤是处理该场景的一个非常有用的想法（我们在探索sagas时会再次提到这一点）。但是，从根本上讲，把一个事务操作拆分为两个独立的数据库事务，我们必须接受：我们失去了整个操作的原子性保证。

缺乏原子性可能会引发严重的问题，尤其是，如果我们要迁移的系统依赖于数据的原子性。在这一点上，人们开始寻找其他解决方案，以能够一次修改多个服务。通常，人们开始考虑的第一个选择是分布式事务。让我们来看一下实现分布式事务的最常见算法之一——二阶段提交，作为探索与分布式事务整体相关的挑战的一种方法。

## 二阶段提交

二阶段提交算法(有时缩写为2PC)经常用于试图提供在分布式系统中进行事务性修改的能力。该分布式系统中，作为整个操作的一部分，多个独立的进程都可能需要更新。我想让大家先了解到2PCs有其局限性，我们会讲到这些局限性，了解这些局限性是值得的。分布式事务，更具体地说是二阶段提交，经常由迁移到微服务架构的团队提出，来作为解决他们所面临的挑战的一种方法。但是，正如我们将看到的，2PCs可能无法解决我们的问题，并且可能会给我们的系统带来更多的混乱。

2PC算法分为两个阶段（因此得名二阶段提交）：投票阶段（*voting phase*）和提交阶段（*commit phase*）。在投票阶段，一个中央协调者（*central coordinator*）与所有将参与事务的workers协商，并要求确认是否可以修改状态。在图4-48中，我们可以看到两个请求：一个请求用于把客户状态更新为VERIFIED，另一个请求用于从PendingEnrollments表中删除数据。如果所有的workers都同意执行要求其执行的状态修改，那么算法则进入下一阶段。如果任何一个worker回复无法执行对应的操作，可能是因为所要求的状态修改违反了服务本地的某些条件，则整个操作中止。

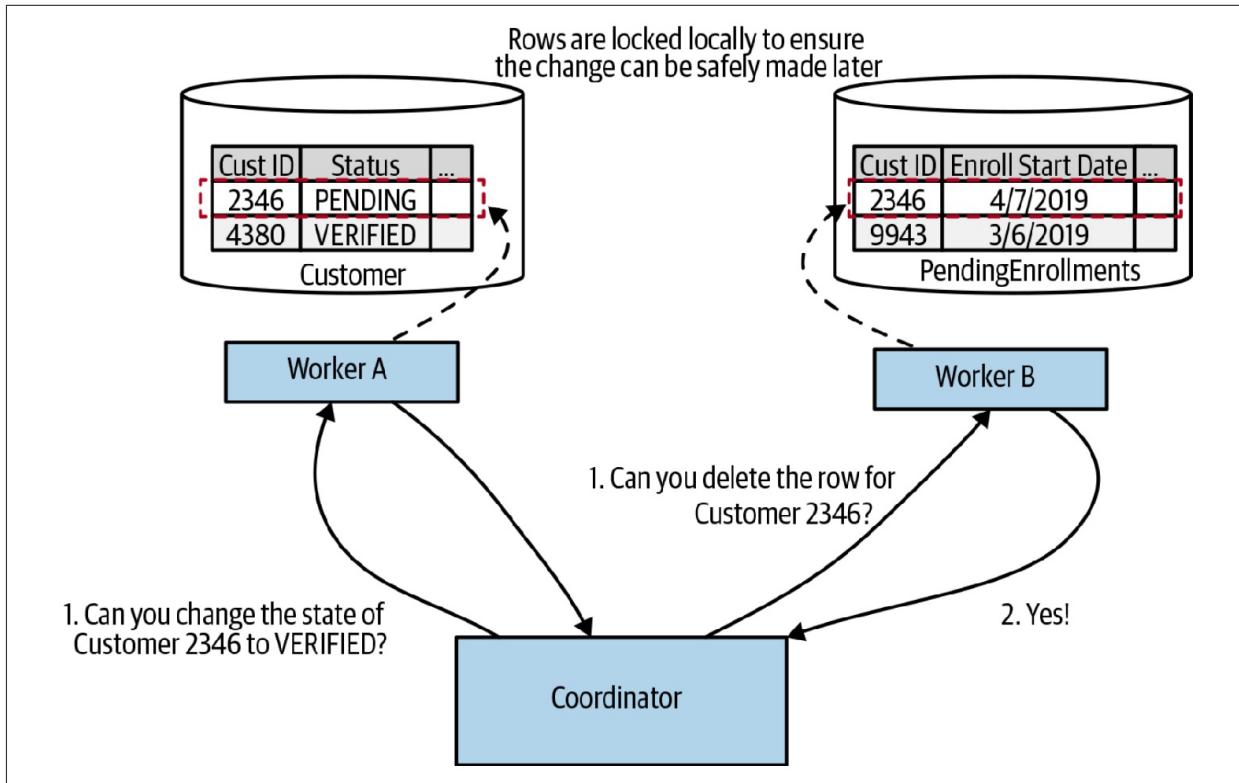


图4-48. 在二阶段提交的第一阶段，所有的workers投票来决定他们是否可以实施一些本地的状态修改

需要强调的是，在workers表示可以进行更新后，更新并不会立即生效。相反，workers保证在未来的某个时刻能够做出更新。Workers如何做出这样的保证呢？例如，在图4-48中，Worker A说它能够更新Customer表中的行状态，以更新特定客户的状态到VERIFIED。如果在后续的某个点进行了不同的操作，删除了该行，或者执行了另一个较小的更新，但却意味着后续更新到VERIFIED的操作被认为是非法的，该怎么办？为了保证后续可以执行该更新，Worker A可能必须锁定记录，以确保如上提到的变更不会发生。

如果有任何的workers没有投票，则中央协调者需要向所有参与方发送回滚消息，以确保它们可以在本地清理，以释放它们可能持有的任何锁。如果所有的workers都同意进行修改，我们就进入提交阶段，如图4-49所示。在

该阶段，会进行实际的修改，并释放相关的锁。

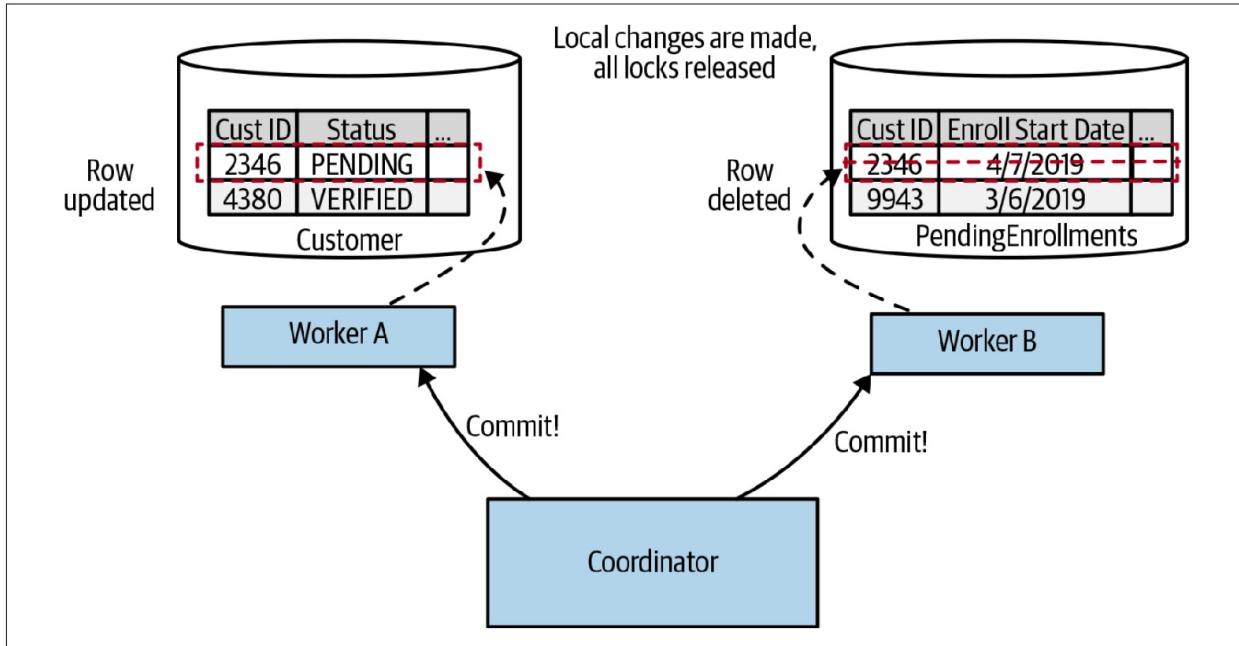


图4-49. 在二阶段提交的提交阶段，更新才被实际应用

需要注意的是，在这样的系统中，我们无法保证这些提交会完全同时发生。协调者需要向所有参与者发送commit请求，不同参与者可能会在不同的时间接收到该消息并处理。这意味着，如果我们允许在事务协调者(*transaction coordinator*)之外查看这些workers的状态，我们有可能看到Worker A所做的修改，但却尚未看到Worker B所做的修改。协调者和workers之间的延迟越大，workers处理响应的速度越慢，这种不一致的时间窗口就越大。回到我们对ACID的定义，隔离性确保我们在事务过程中不会看到中间状态。但在二阶段提交中，我们失去了ACID的隔离性。

当**2PC**工作时，通常，其核心只是协调分布式锁。Workers需要锁定其本地资源，以确保可以在第二阶段进行提交。在单进程的系统中管理锁并避免死锁并不是一件好事。现在，想象一下在多个参与者之间协调锁的挑战。此时并非处于有利地形。

有很多与二阶段提交相关的失败模式，我们没有时间去探索这些失败的模式。考虑如下的问题：一个worker投票以表示可以继续执行事务，但是，在协调者发出commit请求时，该worker却没有响应。那么，此时我们该怎么办？这些故障模式中的某些故障可以自动处理，但有些故障却让系统处于需要手动取消的状态。

参与者越多，系统的延迟就越大，二阶段提交的问题就越多。分布式锁可能是向系统中注入大量延迟的一种快速方法，尤其是在如下的情况下：锁定的范围较大，事务的持续时间较长。因此，二阶段提交通常仅用于非常短暂的操作。操作时间越长，锁定资源的时间就越长！

关于二阶段提交的详细信息，可以参考：<https://www.cs.princeton.edu/courses/archive/fall16/cos418/docs/L6-2pc.pdf>。

# 对分布式事务直接说No

鉴于如上所概述的所有的原因，我强烈建议避免使用分布式事务（例如二阶段提交）来协调跨不同微服务的状态变化。那我们还能做什么呢？

Well，第一个选择是不要一开始就将数据拆分开来。如果要以一种真正的原子性和一致性的方式来管理某些状态，同时我们又没办法在没有ACID式的事務的情况下合理的解决该问题，那么将在单个数据库中存储该状态，并将管理该状态的功能置于单个服务（或者我们的单体）中。如果我们正在研究在哪里拆分单体，并确定了哪些拆分是容易的（或很难的），那么，我们就可以很好的决定：现在就对当前在一个事务中管理的数据进行拆分是一件很难处理的事情。先在系统的其他区域上展开工作，然后后续再回过头来解决该问题。

但是，如果我们确实需要拆分在一个事务中管理的数据，却又希望避免管理分布式事务的所有麻烦，此时会发生什么？我们如何在多个服务中执行操作但又避免资源锁定？如果操作需要花几分钟，几天甚至几个月的时间，该怎么办？在这种情况下，我们可以考虑使用另一种方法：sagas。

---

<sup>8</sup>. This has now changed with support for multidocument ACID transactions, which was released as part of Mongo 4.0. I haven't used this feature of Mongo myself; I just know it exists! ↩

<sup>9</sup>. See Martin Kleppmann, *Designing Data-Intensive Applications* (Sebastopol, O'Reilly Media, Inc., 2017). ↩

# Sagas

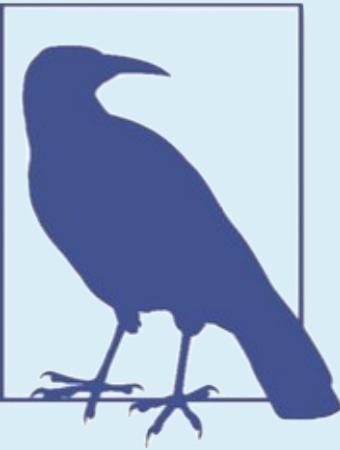
与二阶段提交不同，saga通过设计一个算法来协调状态的多种修改，但在状态修改时不需要长时间锁定资源。为此，我们将涉及到的步骤建模为可以独立执行的离散活动。saga具有强制我们对业务流程进行显示建模的附加好处，这种方式会带来重大好处。

Hector Garcia-Molina和Kenneth Salem首先提出了saga的核心思想<sup>10</sup>来思考如何最好地处理他们所谓的长活事务（LLT: *long lived transactions*）所面临的挑战。LLT可能会花费很长时间（几分钟，几小时，甚至几天），并且事务的过程会对数据库进行修改。

如果直接将LLT映射到普通数据库事务，则单个数据库事务将跨越LLT的整个生命周期。这会导致在LLT期间需要长时间锁定多行甚至整表，如果其他的进程试图读取或修改这些锁定的资源，则会导致严重的问题。

相反，Hector Garcia-Molina和Kenneth Salem建议我们将这些LLT拆分为一系列的事务，每个事务都可以独立处理。Saga的想法是：每个子事务的持续时间将更短，并且只会修改受整个LLT影响的部分数据。其结果就是，随着锁范围的降低和锁时间的减少，底层数据库中的冲突将大大减少。

尽管sagas最初的设想是一种解决单个数据库中的长活事务的机制，但sagas也可以很好地协跨多个服务的修改。我们可以将一个业务流程分解为一个调用集合，这些调用将作为单个saga的一部分来调用所协调的服务。



在继续探讨sagas之前，我们需要明白，saga并不能给我们提供常规数据库的ACID事务中涉及到的原子性。当我们把LLT分解为单独的事务后，我们在saga自身的层面上就没有了原子性。对LLT内部的每个子事务而言，我们确实都有原子性，如果需要，每个子事务都可以关联到一个ACID事务的修改。Saga给我们提供了足够的信息来判断其状态；并由我们来处理该状态产生的影响。

让我们看一个简单的、如图4-50所示的订单处理流程，我们可以用这个流程来进一步探索微服务架构下的sagas。

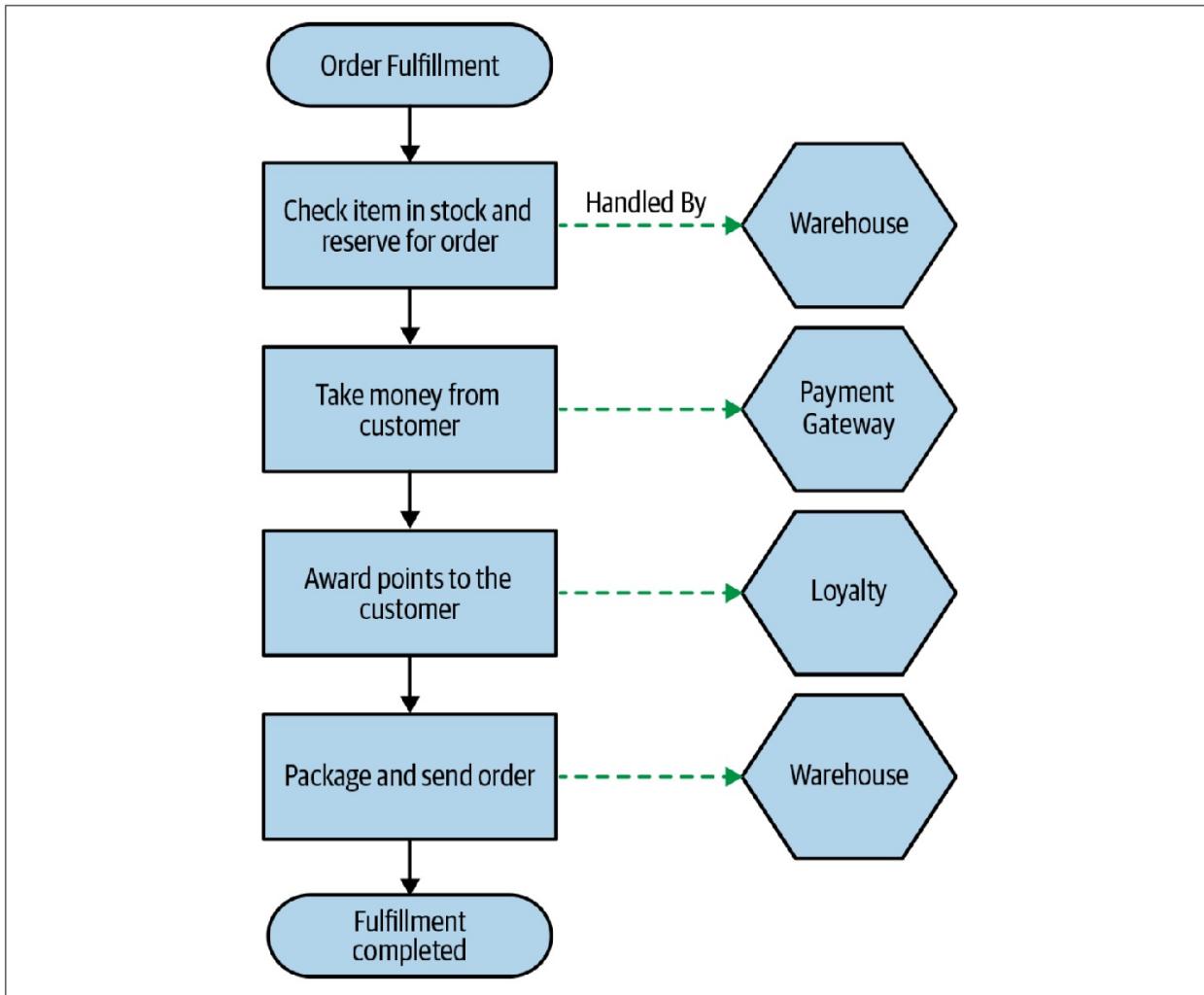


图4-50. 订单处理流程的例子，其中包含负责执行操作的服务

在该例子中，把订单处理过程表示为一个saga，此流程中的每个步骤代表可以由不同服务执行的操作。每个服务都可以在其本地的ACID事务中处理任何的状态修改。例如，当我们使用Warehouse服务来检查和预订库存时，Warehouse服务可能会在其本地的Reservation表中创建一跳预定记录，这种数据的修改会在常规的事务中处理。

# saga的故障模式

随着将一个saga分解为很多相互独立的事务，我们需要考虑如何处理故障——更具体地说，就是在故障发生时如何恢复系统状态。saga的原始论文中描述了两种类型的恢复：后向恢复（*backward recovery*）和前向恢复（*forward recovery*）。

- 后向恢复涉及到故障的还原，然后清理（回滚）所有已经完成的事务。为此，我们需要定义补偿操作，以允许我们撤消先前提交的事务。
- 前向恢复使我们可以从故障发生的地方进行恢复，并继续整个流程的处理。为此，我们需要能够重试事务，这意味着，我们的系统要保留足够的信息以允许重试。

根据要建模的业务流程的性质，我们需要考虑：任何故障模式都会触发后向恢复或前向恢复，或二者都触发。

## saga回滚

对于ACID事务而言，其回滚在事务提交之前发生。事务回滚后，对于数据库而言就好像什么都没有发生一样：我们试图进行的修改并没有发生。但是，对于saga而言，会涉及到多个事务，在我们决定回滚整个操作之前，其中的某些事务可能已经提交。那么，如何在事务提交之后对其回滚呢？

让我们再来看一下如图4-50所示处理订单的例子。考虑如下的故障模式：我们的流程已经到了尝试打包商品的阶段，但是却发现，我们在仓库中找不到该商品，如图4-51所示。系统认为该商品是存在的，只是货架上没有该商品！

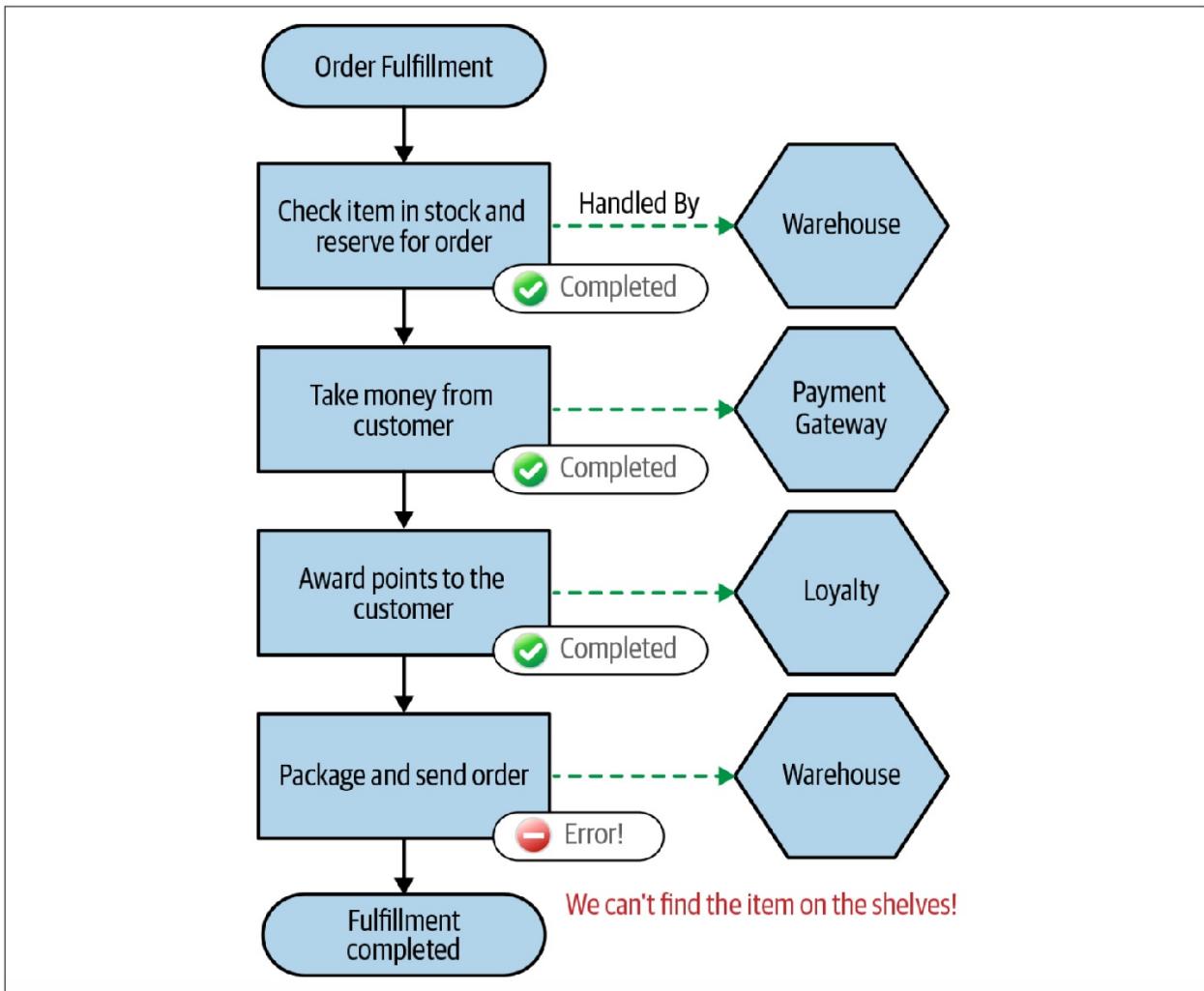


图4-51. 我们尝试打包商品，但是我们在仓库中找不到该商品

现在，假设我们决定回滚整个订单，而不是给予客户延期交货该商品的选择。问题在于，我们已经完成该订单的付款并为之发放了积分。

如果所有这些步骤都在单个数据库事务中完成，则简单的事务回滚将清除所有这些数据。但是，订单执行流程中的每个步骤都是由不同的服务调用处理，每个服务调用都运行于不同的事务范围。对于整个订单操作而言，没有简单的“回滚”。

相反，如果要实现回滚，则需要实现一个补偿事务（compensating transaction）。补偿事务是撤销先前已经提交的事务的操作。为了回滚订单处理过程，对于saga中已经提交的步骤，我们将触发其补偿事务，如图4-52所示。

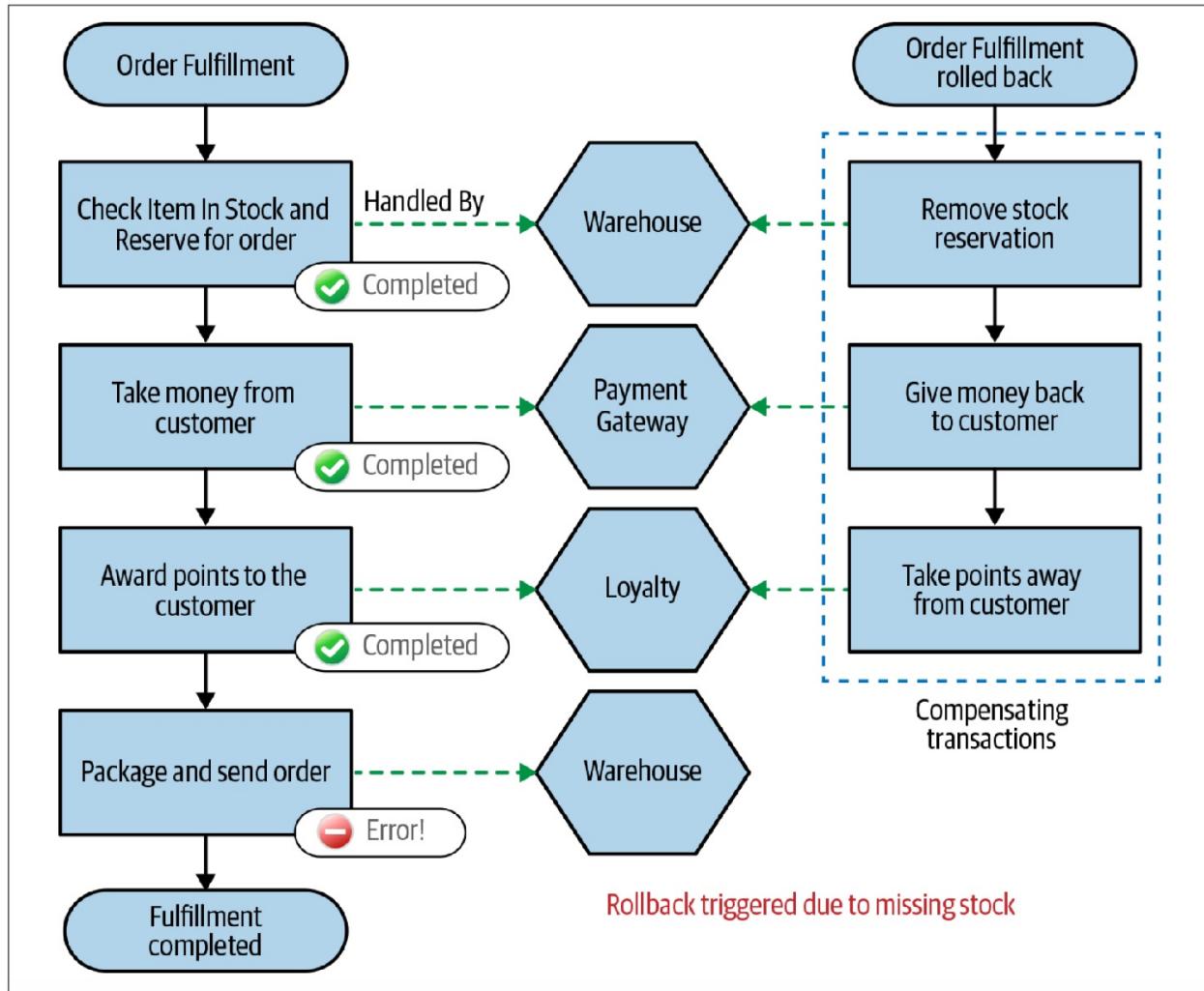


图4-52. 触发整个saga的回滚

值得一提的是，补偿事务的行为与普通数据库回滚的行为并不完全相同。数据库回滚发生在提交之前；在回滚之后，事务就好像从未发生过一样。在补偿事务的情况下，事务当然会发生。我们正在创建一个新的事务来还

原原始事务所做的修改，但是，我们无法回滚时间，从而也无法做到让原始事务就像没有发生过一样。

因为我们无法总是干净地还原事务，所以我们说这些补偿事务是语义回滚。有时，我们不能清理所有的修改，但是，我们为saga的[上下文](#)做了足够的努力。例如，我们有一个步骤可能涉及向客户发送email，以告知其订单已经在路上了。如果我们决定回滚发送电子邮件的操作，但是我们无法撤回已经发送的email<sup>11</sup>！相反，补偿事务可能会向客户发送第二封电子邮件，通知客户订单有问题，并且已经取消该订单。

当然，有些email客户端允许我们撤回已经发送的邮件（例如Microsoft Outlook就有这个功能），但是也仅限于在收件人没有查看该邮件的情况下才可以撤回。一旦用户阅读了该邮件，是无法撤回该邮件的。

在系统中，持久化与回滚saga相关的信息是完全合适的。实际上，这些信息可能是非常重要的信息。出于多种原因，我们可能希望在Order服务中保存已中止订单的记录，以及所发生事情的相关信息。

## 调整操作步骤来减少回滚

在图4-52中，可以通过调整执行步骤来让回滚变得更简单。一个简单的修改就是：仅在订单实际发送时才奖励积分，如图4-53所示。此时，如果我们在尝试打包并发送订单时遇到问题，就不必担心需要回滚积分的操作。有时，我们仅通过调整执行流程就可以简化回滚操作。通过提前执行那些最有可能失败的步骤，并让流程更早地失败，我们可以避免在没有首先执行这些步骤时而导致的需要稍后触发其补偿事务的情况。

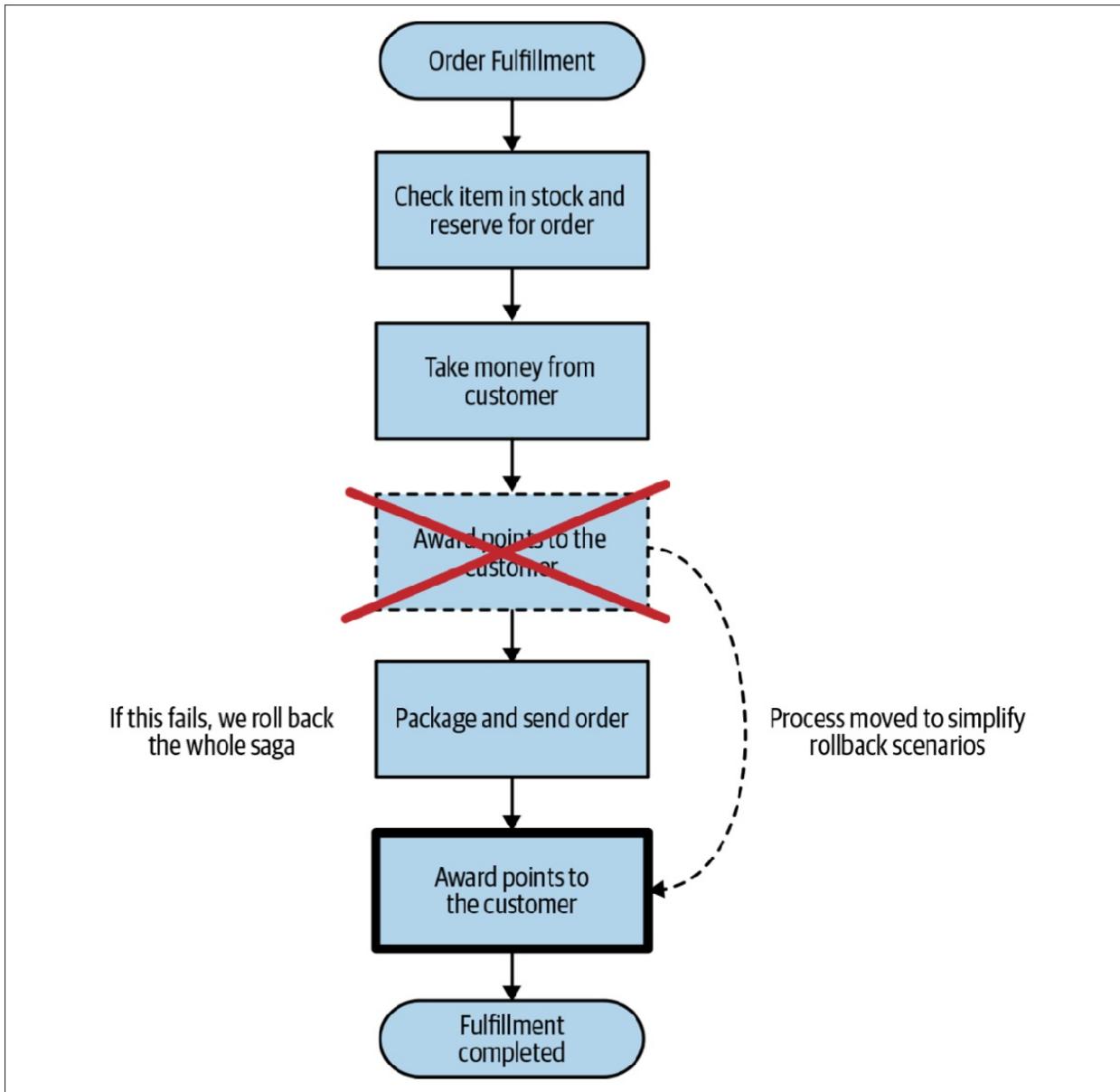


图4-53. 把操作步骤移动到saga的末端可以减少发生故障时所必须的回滚

如果可以做如上的调整，则这种修改可以让我们的工作更加轻松，我们甚至无需为某些步骤创建补偿事务。如果执行补偿事务非常困难，则这中调整尤其重要。我们可以将难以执行补偿事务的步骤移动到流程的末端以达到一个不再需要回滚其操作的阶段。

## 混合使用后向恢复和前向恢复

混合使用不同的故障恢复模式是完全合适的。有的故障可能需要回滚；其他的故障则可以从故障发生的地方进行恢复，并继续整个流程的处理。例如，对于订单处理而言，一旦客户已经付款，并且商品已经打包，则剩下的唯一步骤就是发送包裹。如果出于某种原因，我们无法发送包裹（也许今天，我们的快递公司因为其货车上没有足够的空间而无法接受该订单），则回滚整个订单似乎很奇怪。相反，我们可能只是重新发送包裹，并且如果重试失败，则需要人工干预以解决这种情况。

# 实现Sagas

到目前为止，我们已经研究了sagas运行的逻辑模型，但是我们需要更深入地研究saga的实现方法。我们来看一下saga的两种实现方式。协调sagas (*orchestrated sagas*) 更接近原始的解决空间 (*solution space*)，并且主要依靠中央协调器集来跟踪事件。可以将之与编排sagas (*choreographed sagas*) 进行对比。编排sagas无需中央协调器，而是使用更低耦合的模型，但是这种方式使的对saga流程的跟踪变得更加复杂。

## 协调sagas

协调sagas使用一个中央协调者（从现在开始我们称之为协调器）来定义操作的执行顺序并触发任何必需的补偿行为。可以将协调sagas视为一种命令-控制 (*command-and-control*) 的方法：中央协调器控制在什么时间发生什么事情，因此可以很好地了解任何给定saga所发生的情况。

以图4-50所示的订单处理流程为例，让我们看看该中央协调程序是如何作为一组协作服务而工作的，如图4-54所示。

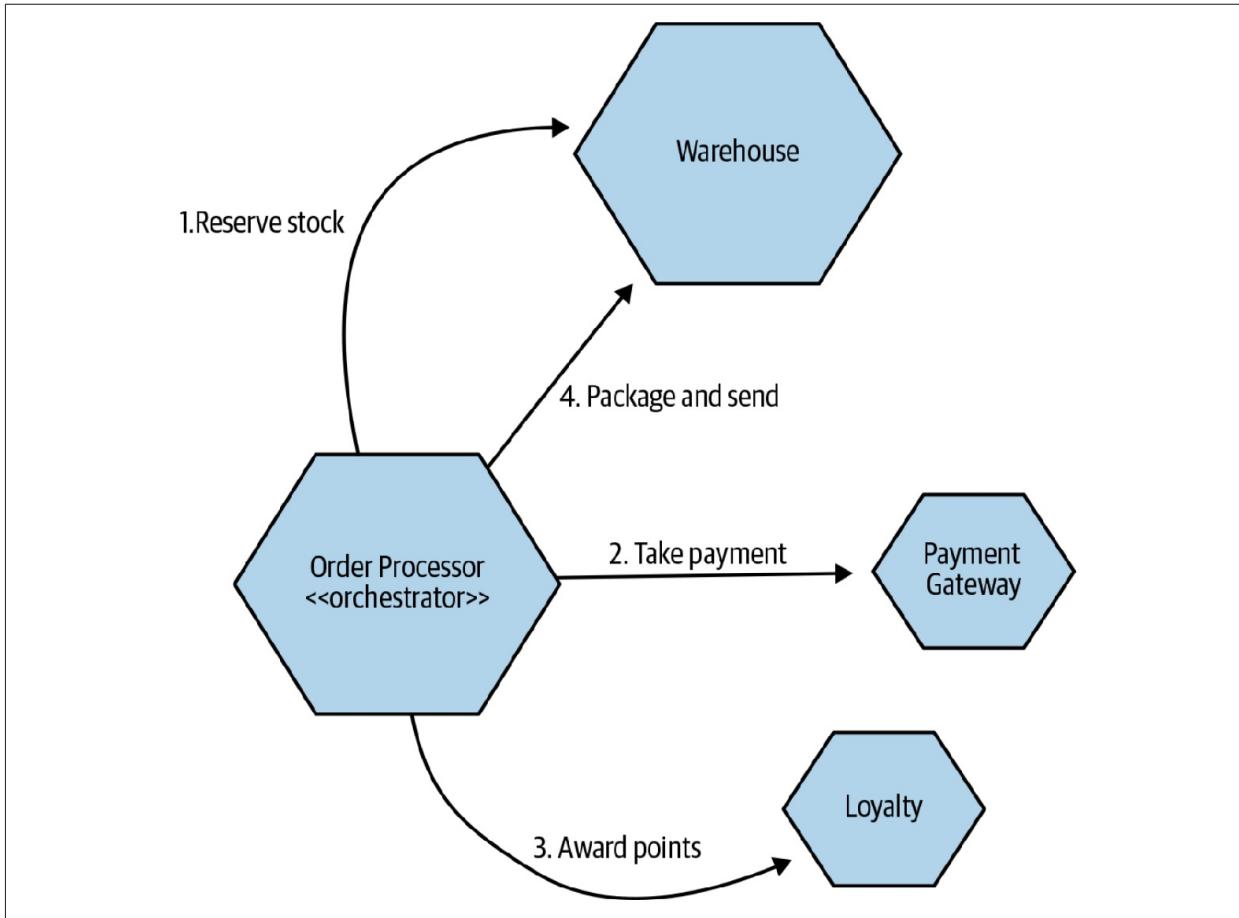


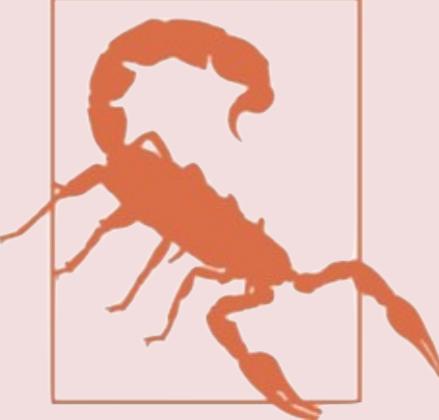
图4-54. 如何用协调saga来执行订单处理流程的例子

[图4-54](#)中，**Order Processor**扮演中央协调器的角色，来协调我们的处理过程。**Order Processor**知道执行该操作需要哪些服务，并决定何时调用这些服务。如果调用失败，**Order Processor**可以决定要怎么做。这些协调处理器 (*orchestrated processors*) 倾向于大量使用服务之间的请求/响应调用：**Order Processor**将请求发送到某个服务（例如**Payment Getway**），并期望得到一个响应以使其知道请求是否成功同时以及期望在该响应中提供请求的处理结果。

在**Order Processor**内部对我们的业务流程进行显示建模是非常有益的。这样，我们就可以查看系统中的某个部分，并了解该过程应该如何工作。如此，可以让新手更加轻松的入门，并有助于更好地理解系统的核心部分。

不过，协调saga有一些缺点需要考虑。

- 首先，从本质上讲，这是一种耦合的方法。**Order Processor**需要了解所有的相关服务，从而使我们在[第1章](#)中讨论的领域耦合的耦合程度更高。尽管这并不是不好，但我们仍然希望将领域耦合保持在可能的最低限度。此时，**Order Processor**需要了解并控制很多事情，以至于这种形式的耦合很难避免。
- 另一个更微妙的问题是，本应该在服务本身处理的逻辑可能开始被合并到协调器。如果这种情况开始发生，我们可能会发现自己的服务变得几乎没有自己的行为，仅仅是从诸如**Order Processor**之类的协调器那里接受订单而已。重要的是，此时，我们还仍然把构成这些协调流程的服务视为具有各自本地状态和行为的实体。这些服务控制其本地的状态机。



如果将逻辑集中到某个地方，则会变成一个中心化的架构。

还记得我们在[第3章第3节](#)中介绍的微服务和SOA的区别以及微服务的*smart endpoints and dumb pipes*吗？

避免协调流带来的过多的中心化的一种方法是，针对不同的流程，确保我们有不同的服务来扮演协调器的角色。我们可能有一个**Order Processor**服务来处理下单，一个**Returns**服务来处理退货和退款流程，一个**Goods Receiving**服务来处理新库存的上架，等等。那些协调器可能会使用类似Warehouse服务之类的服务；这样的模型使我们可以更轻松地将功能保留在Warehouse服务中，从而允许我们在所有的这些流程中复用功能。

## BPM 工具？

业务流程建模（*BPM:business process modeling*）工具已经出现了好多年了。总的来说，BPM的设计目的是让非开发人员使用可视化的拖放工具来定义业务流程。BPM的想法是，开发人员创建业务流程的组件，然后非开发人员将这些组件连接在一起，形成更大的流程。使用这些工具似乎是实现协调sagas的一种很好的方式，实际上，流程协调几乎是BPM工具主要的使用场景（或者，反过来说，使用BPM工具导致我们不得不采用协调）。

根据我的经验，我非常不喜欢BPM工具。主要原因是，根据我的经验，非开发人员定义业务流程的中心思想几乎从来都不是真的。针对非开发人员的工具最终会被开发人员所使用，并且非开发人员可能会有很多问题。非开发人员通常需要使用GUI来修改流程，他们创建的流程可能难以（或不可能）进行版本控制，可能流程本身在设计时就没有考虑测试的问题，等等。

如果开发人员要实现我们的业务流程，让开发人员使用他们知道和了解以及适合其工作流的工具。通常，这意味着仅让他们使用代码来实现这些功能！如果确实需要了解业务流程的实现方式或运作方式，那么从代码投影一个可视化的工作流要比使用可视化的工作流来描述代码的工作方式要容易得多。

正创建对开发人员更友好的BPM工具上花费了很多精力。开发人员对这些工具的反馈似乎褒贬不一，但对于某些工具来说，它们的效果很好，看到人们尝试改进这些框架，我也非常高兴。如果需要进一步探索这些工具，可以看一下[Camunda](#)和[Zeebe](#)，这两个工具都是针对微服务开发人员的、开源的协调框架。

## 编排sagas

编排sagas旨在在多个协作的服务之间分配一个saga中不同操作的职责。如果协调saga是一种命令-控制（command-and-control）的架构，那么编排sagas则代表了一种要信任-但也要查证的架构。正如我们在图4-55的例子中看到的那样，编排sagas通常会大量的使用事件来实现服务之间的协作。

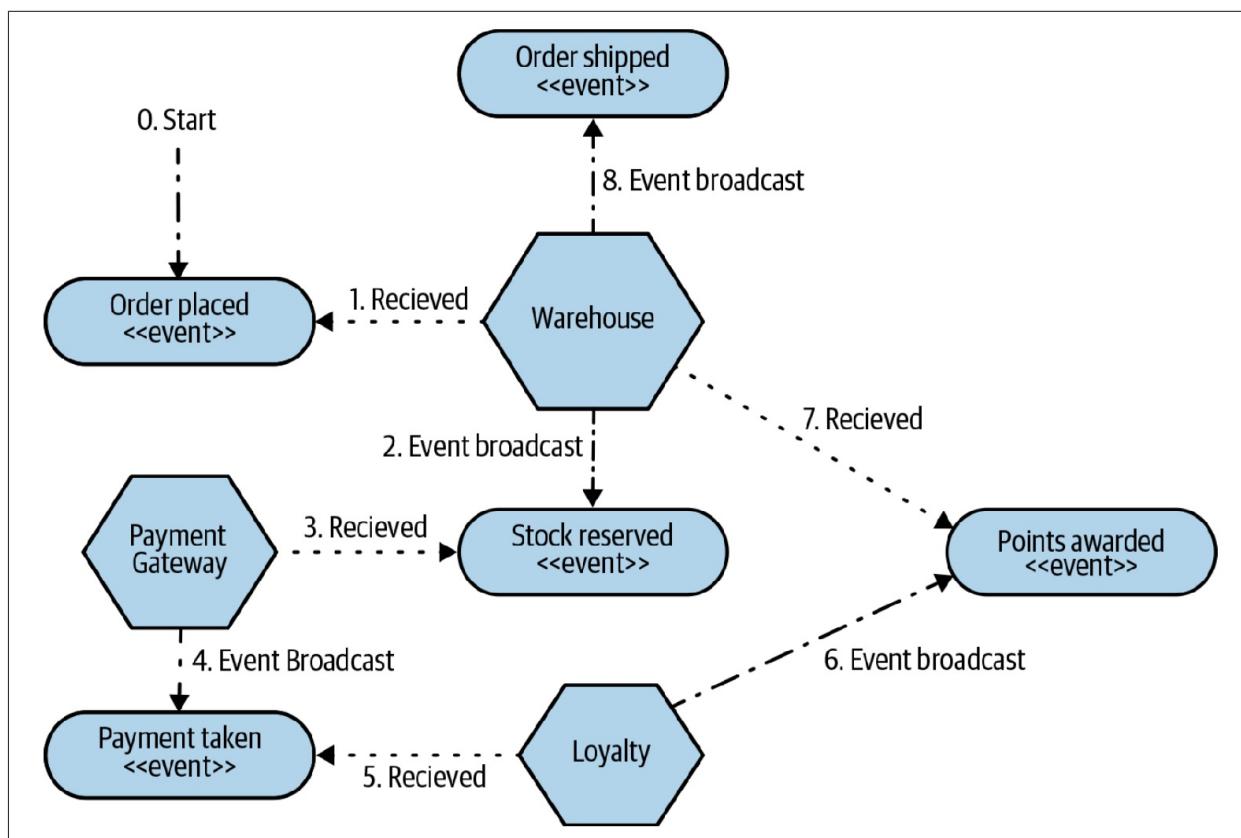


图4-55. 编排saga是如何执行我们的订单处理流程的例子

1987年，在美苏中程核武器协议谈判时，里根旁敲侧击，对戈尔巴乔夫说，他最喜欢一句俄国谚语：“trust, but verify”。意思是，他相信苏联领导人的承诺，但还必须有个有效的验证机制。具体可以参见维基百科的词条[Trust, but verify](#)。

这里面有很多事情，因此，值得更详细地对此探讨。

首先，这些服务需要对接收到的事件做出反应。从概念上讲，事件会在系统中进行广播，对其感兴趣的任何一方都可以接收该事件。我们不会把事件发送给某个服务，我们只需要发送事件即可，对事件感兴趣的服务可以接收这些事件并采取相应的反应。在[图4-55](#)的例子中，当Warehouse服务接收到第一个Order Placed事件时，它就知道要保留适当的库存并在其完成后触发一个事件。如果Warehouse服务无法处理库存，则需要抛出一个适当的事件（也许是Insufficient Stock事件）来中止该订单流程。

通常，我们会使用某种消息代理（*message broker*）来管理事件的可靠广播和分发。多个服务可能会对同一事件做出反应，此时，可以使用topic类型的消息。对某种事件类型感兴趣的服务可以订阅特定的topic，而不必担心这些事件的来源。由消息代理来确保该持久化该topic，并将与之相关的事件成功分发给订阅者。例如，我们可能有一个Recommendation服务，该服务还会监听Order Placed事件，并用之来构建用户可能喜欢的音乐的数据

库。

在编排sagas的架构中，任何一个服务都没有必要知道任何的其它服务的存在。他们只需要知道，在收到某个事件时该服务该怎么做。从本质上讲，这使得架构的耦合更小。由于在实现时，订单处理流程被拆分到了此处的4

个服务中，同时流程的实现也是分布式的，因此我们也避免了对中心化逻辑的担忧（如果没有一个可以集中逻辑的地方，那么逻辑就不会被集中！）。

编排saga的缺点是很难确定正在发生的事情。使用协调saga时，我们在协调器中显示的建模我们的流程。现在，对于本节所呈现的编排saga的架构，我们如何建立一个流程应该是什么样的心智模型？我们必须查看相互隔离的每个服务的行为，并在自己的头脑中重新构建该流程——即使是像[图4-55](#)这样的简单业务流程，在脑中重新构建该业务流程也远非是一个简单的过程。

缺乏对业务流程的显示表示已经够糟糕了，但是我们还缺乏一种了解saga状态的方式，这也让我们无法在需要时采取补偿行为。我们可以把某些职责交给独立的服务以执行补偿操作，但是从根本上讲，我们需要一种方法来了解在某些恢复情况时saga的状态是什么。缺乏查询saga状态的中控是一个大问题。我们通过协调器来实现saga状态的查询，那么，此时，我们如何解决这个问题呢？

最简单的方法之一是：通过消费已经发出的事件来从现有系统中投影有关saga状态的视图。如果我们为saga生成一个唯一的ID，则可以把这个ID放在该saga包含的需要发送的所有事件中，这就是所谓的关联ID。然后，我们利用一个服务来快速消费所有的这些事件，并呈现一个视图以展示每个订单处于什么状态。如果其他服务本身无法解决问题，则可能会以编程方式采取行动来解决问题，并将其作为处理过程的一部分。

## 混合模式

尽管协调sagas和编排sagas在如何实施sagas方面似乎截然相反，但我们可以轻松地对其进行混合，也可以轻松的进行模型匹配。我们系统中的一些业务流程可能天然适合某种模型或另一种模型。我们可能还会有一个包含两

一种saga模型的单个saga。例如，在订单处理的例子中，在Warehouse服务的边界内，当管理包裹的打包和发货时，即使原始请求是较大的编排saga的一部分，我们也可以使用协调saga<sup>12</sup>。

如果决定使用混合模型，那么，我们仍然要有一种清晰的方式来理解作saga所发生的事情，这一点非常重要。没有这种清晰的方式，理解故障模式就会变得复杂，同时，从故障中恢复将非常困难。

## 应该使用编排saga呢还是使用协调saga

实现编排sagas时可能会随之给我们以及团队带来我们所不熟悉的想法。编排sagas通常假设要大量使用事件驱动的协作，而这种协作尚未得到团队的广泛理解。但是，以我的经验来看，跟踪saga中的流程所带来的额外的复杂性几乎总是会超过拥有耦合度更低的架构所带来的好处。

但是，对于协调sagas和编排sagas，除了我自己的个人喜好之外，我的普遍建议是：当一个团队拥有整个saga的实现时，我会使用协调sagas。在这种情况下，在团队边界内，内在的更耦合的架构更易于管理。如果整个saga由多个团队参与，因为更容易将实现saga的职责分配给多个团队，并且低耦合的架构可让这些团队更独立地工作，所以，我非常喜欢拆分度更高的编排sagas。

# Sagas VS 分布式事务

我希望，正如我现在已经分解的那样，分布式事务面临一些重大挑战，除非某些非常特殊的情况，我倾向于避免这些挑战。在我们如今构建的各种应用程序上实施分布式事务而言，Pat Helland——分布式的先驱，为我们提炼了其基本的挑战<sup>13</sup>：

在大多数分布式事务系统中，单节点的故障会导致事务提交停止。反过来，这会使应用程序陷入困境。在分布式事务系统中，分布式事务越大，系统崩溃的可能性就越大。当驾驶一架需要所有引擎都运转的飞机时，增加引擎会降低飞机的可用性。

——Pat Helland, *Life Beyond Distributed Transactions*

根据我的经验，把业务流程显示地建模为一个saga，避免了分布式事务的许多挑战，同时还可以使本来可能被隐式建模的流程对开发人员更加显式和明显。让系统的核心业务流程成为一等概念将带来很多好处。

关于实现协调saga和编排saga的更详细的讨论以及各种实现细节，不在本书的讨论范围之内。在《Building Microservices》的第4章中对此进行了介绍，但我也推荐《Enterprise Integration Patterns》这本书来深入了解该主题的许多方面<sup>14</sup>。

---

<sup>10</sup>. See Hector Garcia-Molina and Kenneth Salem, “Sagas,” in ACM Sigmod Record 16, no. 3 (1987): 249–259. ↪

<sup>11</sup>. You really can’t. I’ve tried! ↪

<sup>12</sup>. It's outside the scope of this book, but Hector Garcia-Molina and Kenneth Salem went on to explore how multiple sagas could be "nested" to implement more complex processes. To read more on this topic, see Hector Garcia-Molina et al, "Modeling Long-Running Activities as Nested Sagas," *Data Engineering* 14, no. 1 (March 1991: 14–18). ↪

<sup>13</sup>. See Pat Helland, "Life Beyond Distributed Transactions," *acmqueue* 14, no. 5. ↪

<sup>14</sup>. Sagas are not mentioned explicitly in either book, but orchestration and choreography are both covered. While I can't speak to the experience of the authors of *Enterprise Integration Patterns*, I personally was unaware of sagas when I wrote *Building Microservices*. ↪

Copyrights © wangwei all right reserved

一旦采用微服务架构，沿途将会遇到各种挑战。我们已经了解了其中的某些问题，但我想进一步探讨这些挑战，以帮助在遇到这些问题之前对其有所了解。

对于可能会遇到的各种问题，我希望本章除能提供足够的信息。我无法在本书中全部解决这些问题，此处概述的许多问题在“Building Microservices”一书中已有更详细的处理方案，在编写该书时就充分考虑了这些挑战。

我还想提供一些提示来帮助大家确定：何时需要解决这些问题。同时，我还会给出在迁移微服务的旅程中，哪里最有可能出现这些问题。

# 服务越多，烦恼越多

采用微服务架构时，何时会出现哪种问题与很多因素有关。服务交互的复杂度，组织的规模，服务数量，技术的选择，延迟和所要求的正常运行时间等因素仅仅是可能导致问题的部分原因。这意味着很难说出何时会，或实际上是否会，遇到这些问题。

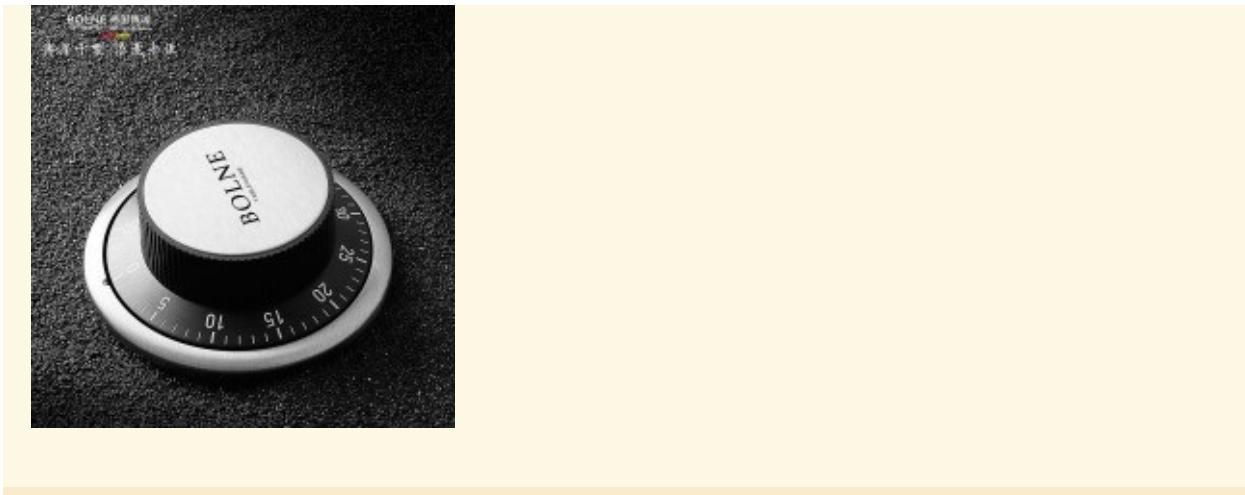
不过，总的来说，我已经意识到，提供几十个服务的公司与提供数百个服务的公司所遇到的问题大不相同。对于何时会出现哪种问题而言，服务数量的指标和其他的任何指标一样好。此处，我需要标注一下，当我谈论服务数量时，除非另有说明，否则我所指的是不同逻辑的服务。把这些服务部署到生产环境之后，每个服务都可以有多个服务实例。

不要把采用微服务视为按一下开关，将其视为转动某个拨盘。当转动拨盘并获得更多服务时，希望可以有更多的机会从微服务中获取好处。但是，当转动拨盘时，会遇到不同的痛点。此时，需要找到解决这些问题的方法，而解决方案可能需要新的思维方式、新的技能、不同的技术、甚至是新的技术。

## 拨盘计时器

按一下开关是一件最简单不过的事情了，整个过程没有任何的中间阶段。但是微服务的迁移却并非是这样的。

就像拨盘计时器一样，从0到指定的时间之间会有很多的间隔和步骤，而对于不同的时间所需要的发条的动力也不相同。微服务的迁移就像拨盘上的不同的刻度一样，随着刻度值的变化，会遇到不同的问题。



根据这些问题最有可能出现的阶段，图5-1大致绘制了本章其余部分将介绍的痛点。这种划分是非常不科学的，并且很大程度上基于经验，但我仍然认为将其作为一种概述很有用。

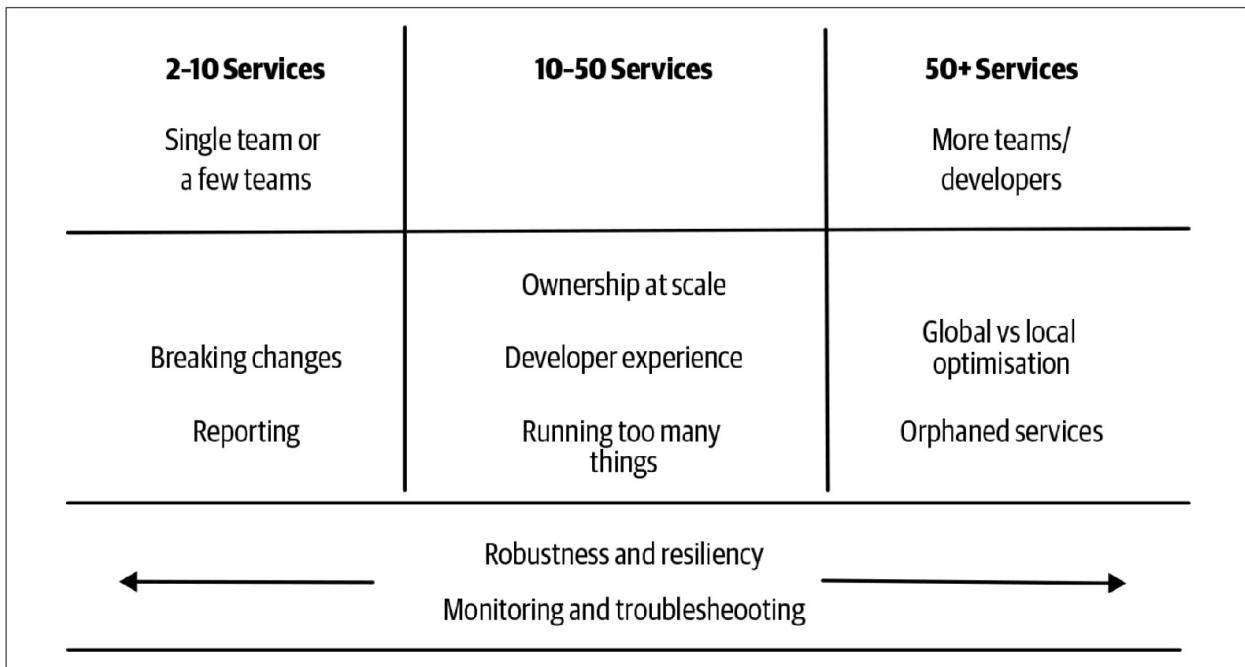


图5-1. 在高层级上显示出某些痛点经常出现的位置

我并不是说肯定会在这些时间点遇到这些问题。某些涉及到的变量会让图5-1这样简单的图表无法真正实现。当这些问题发生时，可能会变化的一个特别的因素是：架构最终是如何耦合在一起的。架构的耦合越大，围绕鲁

棒性、测试、跟踪等问题就会越早的表现出来。我所做的所有的一切就是希望照亮那些可能存在的潜在陷阱。

但是，请记住，应该将图5-1用作一个通用指标。需要确保自己正在建立反馈机制，以寻找我在此处概述的一些潜在指标。

现在，我已经阐述了图5-1的注意事项，让我们更详细地研究一下每个问题。我将提供一些提示以指出哪些因素会让这些问题显现出来，了解这些问题如何影响我们，并提供一些解决这些挑战的思路。

Copyrights © wangwei all right reserved

# 团队不断扩张时的微服务所有制问题

随着越来越多的开发人员加入微服务架构，我们将会处于可能想要重新考虑如何处理服务所有制的地步。

Martin Fowler已经从一般的代码所有制的角度区分了[不同类型的代码所有制](#)。我发现，从广义上讲，Martin Fowler对代码所有制的分类在微服务所有制的上下文中也起作用。此处，主要是从修改代码的角度，而不是从部署、一线支持等角度来考虑服务的所有制。在讨论各种问题之前，首先让我们看一下Martin的代码所有制的分类，并将它们放在微服务架构的上下文中：

- 强代码所有制：所有的服务都有负责人。如果其他人想要修改非自己负责的服务的代码，则他们必须将自己的修改提交给服务的负责人，并由服务的负责人来确定是否允许此次修改。其他人可以使用 `pull request` 的方式来修改非自己负责的服务。
- 弱代码所有制：大多数服务都是有负责人的，但是其他人无需利用 `pull request` 之类方法就可以直接修改代码。实际上，源代码的控制虽然设置为允许任何人修改任何内容，但是如果我们会修改其他人的服务，仍然期望我们事先与这些服务的负责人沟通。
- 集体代码所有制：任何个人都不拥有任何事情，任何人都可以任意修改他想修改的事情。

# 该问题如何表现出来

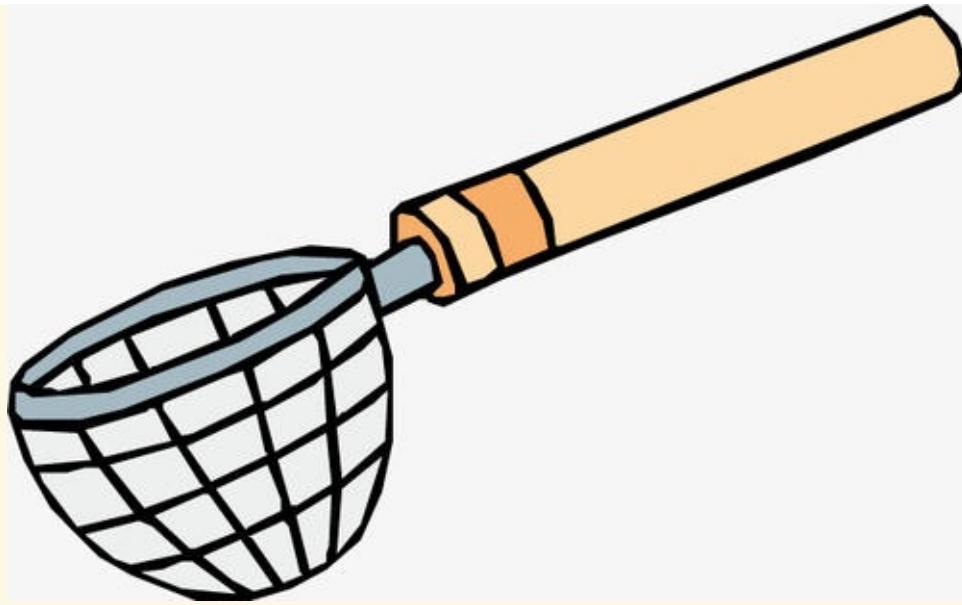
随着服务数量和开发人员的增加，我们可能会开始遇到集体所有制的问题。为了使集体所有制可以发挥作用，集体必须有足够的联系，以便对什么是好的变化达成共识，同时可以从技术的角度对所提供的特定服务的方向达成共识。

不管怎样，我已经发现集体代码所有制对于微服务架构而言是一种灾难。我曾与一家金融科技公司分享过一个小型团队快速发展的故事。该团队从30–40个开发人员发展到100多人，但是在这个过程中，除了“people know what is right”外，他们没有对系统的不同部分或任何概念的所有制进行任何的职责划分。

随着系统架构的发展，整个系统不再像之前一样清晰，而是成为一个糟糕的“分布式单体”。因为整个系统充斥着洞（*hole*），因此团队的某位开发人员称他们的架构为“漏勺架构”——当他们想要开放数据或执行大量点对点调用时，他们就会“打一个新的洞”<sup>1</sup>。实际上，单体系统更容易解决这些挑战，而分布式系统下解决这些问题则要困难得多——改变一个分布式单体的成本更高。

## 漏勺式系统

日常生活中，漏勺就像下面的图一样：



还有一种系统，我称之为“筛子式系统”，平时看起来挺健壮的，但是实际上充满了各种漏洞。一旦倒上水，才发现漏洞百出呀，并且水量越大，危险系数越大。



# 该问题何时会出现

对于许多刚起步的团队来说，集体代码所有制模型是有意义的。当开发人员的数量较少时（大约20名），我对这种模型的效果感到很满意。随着开发人员的增加或者这些开发人员并非集中于一个地方，要使每个人都对诸如如下的事情有一致的想法越来越难：

- 什么是一次好的提交
- 各个服务应如何发展

对于正处于快速增长阶段的团队而言，集体所有制模型是有问题的。此处的问题在于，要发挥集体所有制的作用，就需要时间和空间来达成共识，并随着学习到的新知识来更新这些共识。一般而言，人越多，越困难。并且，如果招聘新人的速度很快（或让新人快速投入到项目中）时，这是相当困难的一件事情。

# 该问题的解决方案

以我的经验，强代码所有制几乎是组织、实施大规模微服务架构的组织所采用的模型。对于该大规模架构而言，会有多个团队、100多名开发人员参与其中。每个团队都可以轻松地决定良好变更的规则。每个团队可以视为在本地采用集体代码所有制。强代码所有制模型还可用于面向产品的团队，如果团队拥有一些围绕业务领域的服务，那么团队将更加专注于业务领域的某个领域。这使得保持以客户为中心的、建立领域专业知识的团队变得更加容易，这些团队通常由指派到该团队的产品负责人来指导团队的工作。

---

<sup>1</sup>. A colander is a bowl with lots of holes, used for straining pasta, for example. ↵

Copyrights © wangwei all right reserved

# 破坏性的服务变更

微服务作为更大的系统的一部分而存在。每个微服务要么使用其他微服务提供的功能，要么向其他微服务提供自己的功能，要么二者兼而有之。对于微服务架构而言，我们正在努力实现其独立可部署性。但是为了实现其独立部署，我们需要确保对微服务的修改不会干扰其下游消费者。

我们可以用契约的形式把服务的功能开放给其它的微服务。契约并不仅仅用来描述：“微服务将返回的数据”。契约还涉及到定义服务的预期行为。无论是否与下游服务明确的建立该契约，该契约都会存在。当修改服务时，需要确保没有违反该契约；否则，就会出现严重的线上问题。

我们迟早都要应对破坏性变化所带来的挑战——要么是因为有意做出的非向后兼容变更的决定，要么是因为无意的修改。对于无意而为之的修改，我们以为他只会影响本地服务，却发现它以难以想象的方式破坏了其他的服务。

# 该问题如何表现出来

此问题最严重的情况是，因破坏契约兼容性的、新的微服务生效而导致的系统中断。这表明，我们没有尽早捕捉意外的破坏契约的行为。如果没有快速的回滚机制，那么这些问题将是灾难性的。解决这种类型的故障的唯一的优势就是，除非修改的是很少使用的那部分服务，否则变更发布之后，很快就会表现出这类故障。

另一种迹象是，我们开始发现人们试图协调多个服务的同时部署（有时称之为同步发布（*lock-step release*））。在试图管理客户端和服务器之间的契约变化时，也可能会出现该现象。在团队中，偶尔的同步发布并不会太糟，但是如果同步发布很常见，则需要进行一些调查。

# 该问题何时会出现

我发现，破坏性修改的问题是团队很早就能遇到的发展烦恼，尤其是当开发跨越多个团队时。在一个团队中时，人们在做出破坏性修改时往往会有更清楚的认识，部分原因是开发人员很有可能会同时修改服务和该服务的消费者。当一个团队正在修改其他团队会使用的服务时，可能会更频繁的出现破坏性修改的问题。

随着时间的流逝，随着团队变得越来越成熟，团队会更加努力地做出改变来避免出现故障，并且还建立机制来尽早发现问题。

# 该问题的解决方案

我有一套管理违反契约的规则，它们非常简单：

1. 不要违反契约。
2. 参考第一条。

好的，我稍微开个玩笑。对开放出来的契约进行破坏性的修改并不好，并且也很难管理。如果可以的话，我们真的要最小化这种破坏性的修改。也就是说，如下的规则是更现实的规则：

1. 消灭意外的破坏性修改。
2. 在做破坏性修改之前要三思而后行——可以避免破坏性修改吗？
3. 如果需要进行破坏性修改，请给下游消费者一些时间来迁移到新的契约。

让我们更详细地了解这些步骤。

## 消灭意外的破坏性修改

微服务的显示模式 (*explicit schema*) 可以快速检查出契约中的破坏性修改。如果开放了一个方法，该方法的参数为两个整型参数，而现在改方法的参数为1个整型参数，则这显然是一项破坏性变更，这种破坏性修改在新的模式中显而易见。向开发人员提供显示模式可以帮助开发人员及早发现破坏性修改。如果开发人员必须手动修改模式，那么这将成为一个显示的步骤，该步骤有望让开发人员停下来并对修改进行思考。如果使用的是正

规的模式格式，那么也可以选择以编程的方式来处理模式修改的问题，尽管该方式并不能按照我所想要的那样工作。[protolock](#)就是该类工具的一个例子，该工具实际上将禁止不兼容的模式修改。

许多人会默认选择使用无模式的交换格式 (*schema-less interchange formats*)，最常见的例子就是JSON。尽管理论上而言，可以为JSON定义显式模式，但实际上却并未应用于实践。开发人员一开始都讨厌正规模式的约束。但是，在他们不得不处理跨服务的破坏性修改之后，他们就会改变主意。还值得注意的是，使用模式的某些序列化格式的数据可以提升服务的性能，这一点也值得考虑。

但是，破坏协议结构只是破坏性修改的一部分。还需要考虑语义上的破坏。如果我们的计算方法仍然有两个整型参数，但是最新版本的微服务是将这两个整数相乘，而以前只是将它们相加，这也是契约的破坏性修改。实际上，测试是检测此问题的最佳方法之一。我们将在稍后讨论。

无论做什么，当开发人员修改对外开放的契约时，最好的方式就是尽可能地让修改对开发人员显而易见。这可能意味着要避免采用魔术般地序列化数据或者利用代码生成模式的技术，而更愿意采用手动的方式来生成这些内容。相信我，让服务契约难以修改总比不断破坏下游消费者更好。

## JSON-Schema

对于Json格式的接口定义，已经不单纯停留在理论阶段了，目前已经有了一些可用于实践的方案：

- [JSON-Schema](#)目前可以用来定义Json格式的接口。
- 由Swagger发起的，目前由[Linux Foundation](#)维护的[OAS:OpenAPI Specification](#)也可以用来解决这个问题。
- 当然，最近的[YAPI平台](#)也可以用于解决json接口定义的问题。

前两年，在我负责质量的横向TOPIC的时候，我亲身经历了几十起破坏性修改隐示的json契约而导致的严重的线上问题。因为研发人员拒绝使用显示的、手工编写的JSON-Schema，当然这也确实会花一些时间。因此，研发人员根本就不会意识到他们对服务的修改实际上已经破坏了契约。因此而带来的问题，各种各样，但每一个问题都非常严重，并且一上线就开始表现出来，例如：

- 客户端崩溃
- 下游服务异常
- .....

尤其是整个系统服务数量较多，而不同服务采用的技术栈又不同时，这种情况尤为严重。例如上游的 PHP 服务，将 string 类型的数据修改为 integer 类型，而下游的 c++ 利用 rapid-json 或 boost 来处理该数据时，就会引起服务异常。这时，最好的情况是下游服务立刻发生崩溃而让研发人员感知到问题，否则就会带来严重的逻辑错误。然而，即便是崩溃这种及时反馈机制而言，对于服务的影响也是非常严重的。

崩溃还是逻辑异常，这是一个非常难的问题。

既然如此，那还不如使用 JSON-Schema，当然这也意味着要花一点点时间来写 JSON-Schema。没有不劳而获的好处，就是这样子的。

## 在破坏性修改之前，三思而后行

如果可能的话，尽可能对契约进行扩展性修改。增加新的方法，资源，topics或在不删除旧方法的情况下支持新功能。尝试找到方法以便在支持旧版本的同时仍然可以支持新版本。这可能意味着最终不得不支持旧代码，

但这仍然比解决破坏性修改的工作更少。请记住，如果我们决定破坏契约，必须由我们来解决因此而带来的问题。

## 给下游消费者时间来迁移

从一开始我就很清楚，微服务是为可独立部署而设计。当修改微服务时，需要能够在无需部署任何其他东西的情况下将该微服务部署到生产环境。为了达到这个目的，需要以不影响现有消费者的方式修改服务契约。因此，即使有新的契约可用，也要允许下游消费者仍然可以使用旧契约。然后，需要给所有的消费者留有时间去修改他们的服务，以迁移到较新的服务版本。

有两种方法可以实现如上的目的。第一种方法是运行两个版本的微服务，如[图5-2](#)所示：可以同时使用Notifications服务的两个构建版本，每个构建都开放了供消费者选择的相互不兼容的功能。这种方法的主要挑战是，必须具有更多的基础设施来运行额外的服务，可能必须维护服务版本之间的数据兼容性，并且可能需要对所有正在运行的版本进行bug修复，这不可避免地需要拉取源代码分枝。如果仅在短时间内共存两个版本，则可以在某种程度上缓解版本共存的问题，只有在这种情况下，我才会考虑采用这种方法。

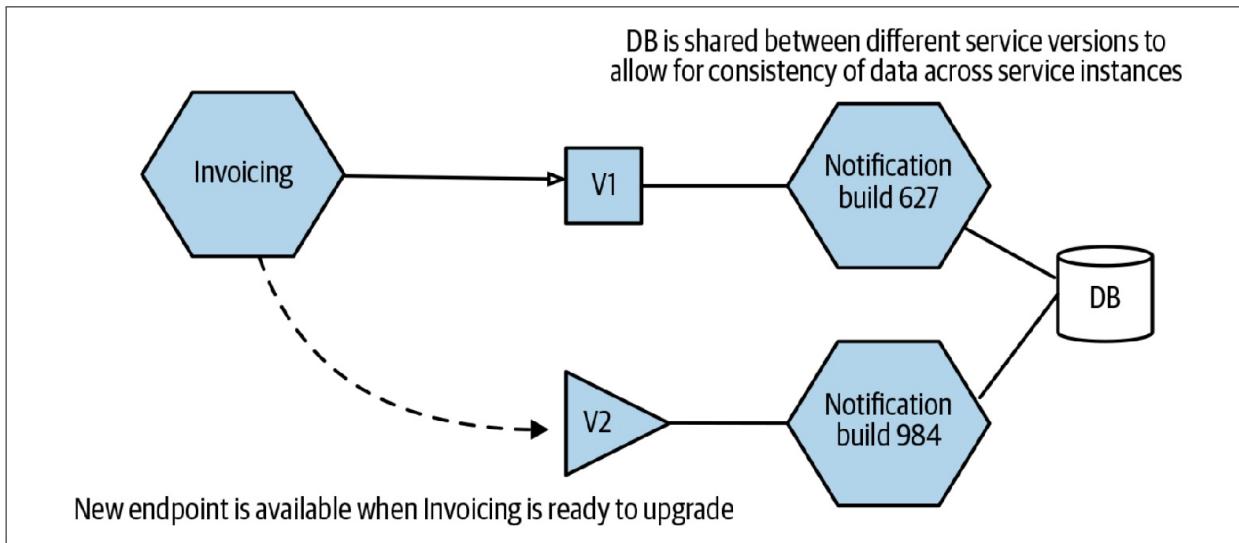


图5-2. 共存同一微服务的两个版本以支持向后不兼容的升级

我更喜欢的方法是：让正在运行的微服务版本同时支持新、老两种契约，如图5-3所示。这可能会涉及在不同端口上开放两个API。这将复杂性推到了微服务的实现，但避免第一种方法的挑战。我与很多团队做过沟通，由于外部的服务无法修改，数年之后，他们会在同一个服务中支持三种或更多的旧的契约。这中境地并不好，但是如果我们将无法改变的服务正在使用我们提供的服务时，我仍然认为该方法是最好的选择。

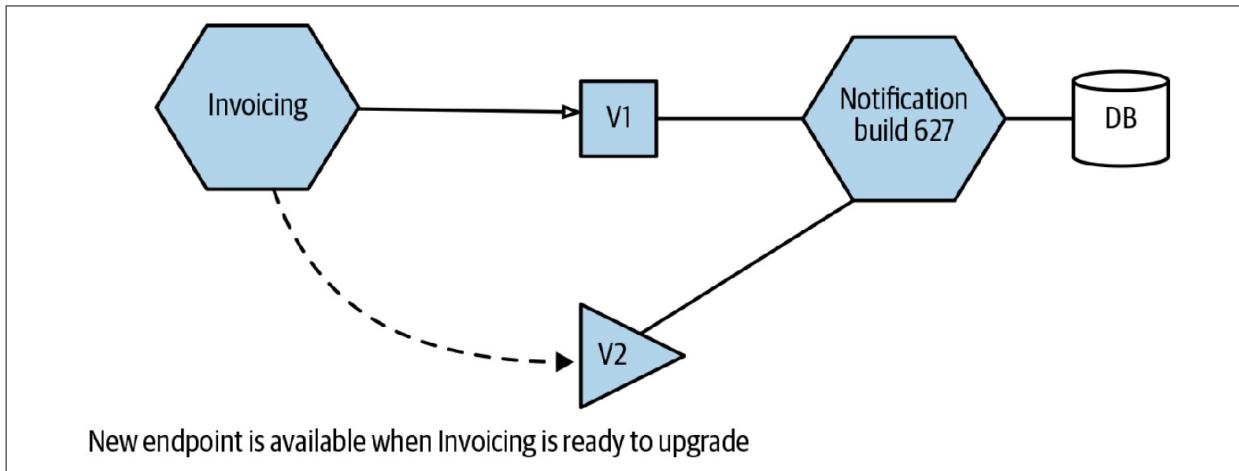


图5-3. 一个服务开放了两个版本的契约

当然，如果同一个团队同时处理消费者和生产者，则可以同步发布并部署消费者和生产者的新版本。这不是我想要经常做的事情，但是至少在一个团队中，更容易管理发布顺序——只是不要养成这种习惯！

团队内部的变更更容易管理，因为我们可以控制修改的服务及其下游消费者。随着要修改的微服务的使用范围更广，管理变更的成本也会随之增加。结果就是，我们可以更轻松地进行团队内部的破坏性修改，但是破坏开放给第三方的API可能会很痛苦。

不管我们怎么做，都需要与使用我们的服务的管理者进行良好的沟通。我们可能会给他们带来不便，因此与他们保持良好的关系是一个好主意。像对待客户一样对待服务的消费者——我们应该好好对待客户！

### 解决破坏性修改的问题——要快！

随着组织的微服务越来越多，他们最终会找到方法来最大程度的消除意外的破坏性变化，并提出一种可管理的机制来处理有目的的变化。如果不这样做，那么破坏性变化的影响会大的让微服务架构站不住脚。换句话说，我高度怀疑没有解决破坏性修改的小型微服务组织无法持续到成为大型的微服务组织。

Copyrights © wangwei all right reserved

# 报表的问题

单体系统通常会有一个单体数据库。这意味着，那些想要分析全部数据（通常涉及跨数据的较大的join操作）的分析者，可以使用现成的数据模式来执行他们的报表。可以直接在单体数据库上执行这些报表，也可以如图5-4一样，在其只读副本上执行这些报表。在微服务架构下，我们打破了这种单体模式。这并不意味着不再需要跨越全部数据的报表。但是，现在，我们的数据分散在多个逻辑上相互隔离的数据模式中，因此获取全部数据的报表已经变得更加困难。

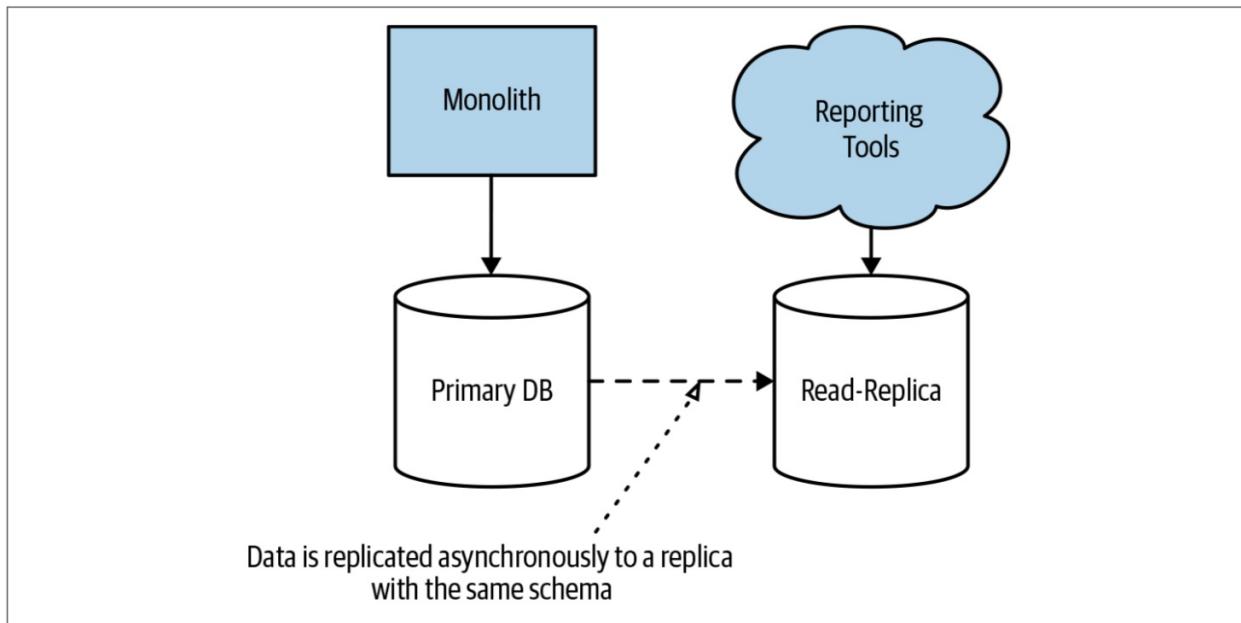


图5-4. 直接利用单体数据库产出报表

# 该问题何时会发生

报表的问题往往会出现的较早，该问题通常发生在我们开始考虑拆分单体的阶段。希望能在问题出现之前就发现该问题，但是我已经不止一次的发现，架构迁移到了一半时，团队才意识到架构的调整会给报表的使用者带来痛苦。很多时候，并没有尽早的考虑下游报表的需求，因为报表的需求超出了正常软件开发和系统维护的领域——报表的需求成为了一个盲区。

## out of sight, out of mind

在这里，我将"out of sight, out of mind"译作盲区。盲区意味着着某些事情处于视野范围之外的区域，因此我们无法感知这些事情的存在，但是这并不是说这些事情就不存在。

你看见或者看不见，事情就在那里，不曾消失。

不单纯是报表的问题，盲区会导致其他的风险的存在。我亲眼见过多次线上事故就是因为参与者的盲区而引起的，此时他们一般会说：

- 我不知道这个模块还调用了其他模块，所以没考虑到影响
- 我对这块功能也不是很了解，所以没想到这里可能会异常
- RD说这里没有改动，所以就没有关注

如果报表已经使用专用数据源来计算，例如[数据仓库或数据湖](#)，则可以避开此问题。接下来，我们只需要确保微服务能够将适当的数据复制到现有的数据源即可。

# 该问题的解决方案

在许多情况下，那些关心在一个地方访问所有数据的利益相关者可能会在工具链和流程上进行投资，以希望可以使用SQL来直接访问数据库。因此，他们的报表可能与单体数据库的模式设计有关。这意味着，除非要改变报表的工作方式，否则我们仍需要提供一个单独的数据库来产出报表，并且该数据库要尽可能的与旧的模式设计相匹配，以限制任何修改带来的影响。

解决此问题的最直接方法是，首先将用于报表的单个数据库与微服务用于存储和检索数据的数据库分开，如图5-5所示。这样一来，报表数据库的内容和设计就可以与各微服务使用的数据库的设计和演变解耦。此时，还允许在考虑报表用户的特定需求的情况下，修改新报表的数据库。我们要做的就是：搞清楚微服务如何将数据“推送”到新的数据库。

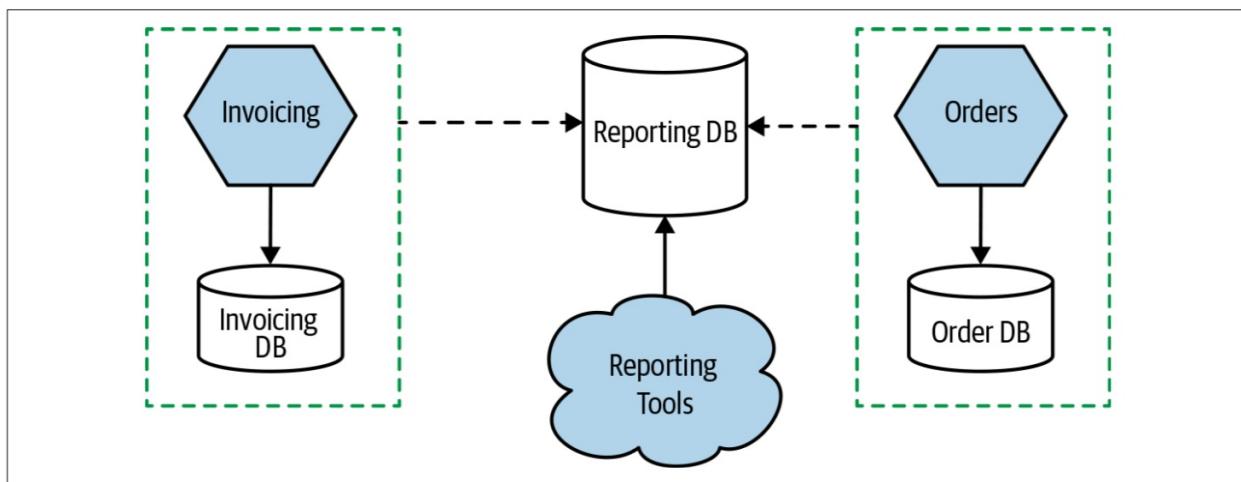


图5-5. 将不同微服务的数据推送到专用的报表数据库

我们已经在第4章中研究了解决此问题的解决方案。CDC系统显然是解决此问题的潜在的解决方案，但是诸如数据视图之类的技术也可能是一种方案，对多个微服务公开的数据视图执行投影操作而产生单个报表。还可以考虑使用其他技术，例如将数据复制的逻辑作为微服务代码的一部分，或者利用中间件把对上游服务的监听事件传递给报表数据库。

在《Building Microservices》一书的第5章，我更详细地探讨了有关该主题的挑战和解决方案。

Copyrights © wangwei all right reserved

# 监控和问题定位

We replaced our monolith with microservices so that every outage could be more like a murder mystery.

我们用微服务代替了我们的单体，这使得每次服务中断都更像是一个迷案。

——[Honest Status Page\(@honest\\_update\)](#)

对于标准的单体应用而言，我们可以采用一种相当简单的方法来监控。我们需要监控的机器数量很少，并且在某种程度而言，应用程序的故障模式是二元的——应用程序通常要么全部可用，要么全部不可用。而对于微服务架构，可能仅有一个服务实例出现故障，或者可能仅需要考虑某种类型的实例——我们可以作出正确的选择吗？

对于单体系统，如果CPU的利用率长时间处于100%的状态，我们知道这是一个大问题。但是，在具有数十或数百个进程的微服务架构中，我们可以作出同样的论断吗？当只有一个进程的CPU利用率达到100%时，我们是否需要在凌晨3点把某人喊醒？

随着微服务的数量越来越多，如下的事情变的更加复杂：

- 查清楚问题出在哪里
- 了解我们所看到的那些问题是否是实际上需要担心的事情

随着微服务架构的发展，监控和问题定位的方法需要随之改变。监控和问题定位是一个需要持续关注和投入的领域。

# 该问题何时会发生

准确预测何时会出现监控和定位的问题比较困难。简单的答案可能是：“线上第一次出现问题”，但要尝试找出问题所在时（该问题是上线之前，开发和测试人员可能必须处理的问题）。当有几个服务时，可能会遇到这些问题，或者直到达到20个或更多的服务时，才可能遇到这些问题。

因为很难准确预测现有的监控方法何时会开始失效，所以我只能建议大家优先实施一些基本的改进。

# 该问题是怎样发生的

从某种意义上说，很容易发现该问题。我们会发现我们无法解释或理解线上问题，尽管系统看起来很健康，但仍会触发报警，并且很难回答一个简单的问题“系统的一切都还好吗？”

# 该问题的解决方案

有很多机制——其中一些易于实施，而另一些则更为复杂——可以帮助我们改变微服务架构下的监控以及问题定位的方式。接下来是对要考虑的一些关键事项的详尽介绍。

## 日志聚合

对于少量的机器，尤其是长时间运行的机器，当我们需要检查日志时，通常会登录该机器并获取信息。微服务架构的问题在于我们有更多的进程，这些进程通常运行在更多的机器上，而这些机器可能是短暂运行的机器（例如虚拟机或容器）。

日志聚合系统允许我们捕获所有日志，并将日志转存到一个中央位置，并可以在该中央位置搜索日志。在某些情况下，日志聚合系统甚至可以用来产生报警。存在许多日志聚合系统可供选择，从开源的[ELK](#)（[Elastic search](#), [Logstash](#)/[Fluent D](#), [Kibana](#)）到我个人最喜欢的[Humio](#)，这些系统都非常有用。



在实现微服务架构之前，请首先考虑实现日志聚合。日志聚合非常有用，并且可以很好地测试组织在运维空间中实施变更的能力。

日志聚合是最简单的实现机制之一，我们应该尽早构建日志聚合系统。实际上，我建议日志聚合是实现微服务架构时应该做的第一件事。之所以如此，部分原因是因为日志聚合从一开始就非常有用。此外，如果团队难以实施合适日志聚合系统，则可能需要重新考虑是否对微服务做好了准备。实施日志聚合系统所需的工作非常简单，作为一个组织，如果还没有准备好日志聚合系统，微服务可能就太激进了。

## 请求追踪

只分析相互隔离的、单个服务的信息，则很难了解：

- 在微服务之间的一系列调用中，哪里失败了？
- 哪个服务导致了延迟峰值？

能够整理出连续的请求调用链并将其视为一个整体非常有用。

首先，为进入系统的所有请求生成关联ID，如图5-6所示。Invoice服务收到请求时，会为其提供一个关联ID。当Invoice请求Notification服务时，Invoice服务会将该关联ID传递到Notification服务。ID的传递可以通过HTTP头、或者消息有效负载（*message payload*）中的字段、或某种其他的机制来完成。通常，我希望使用API网关或service mesh来生成最初的关联ID。

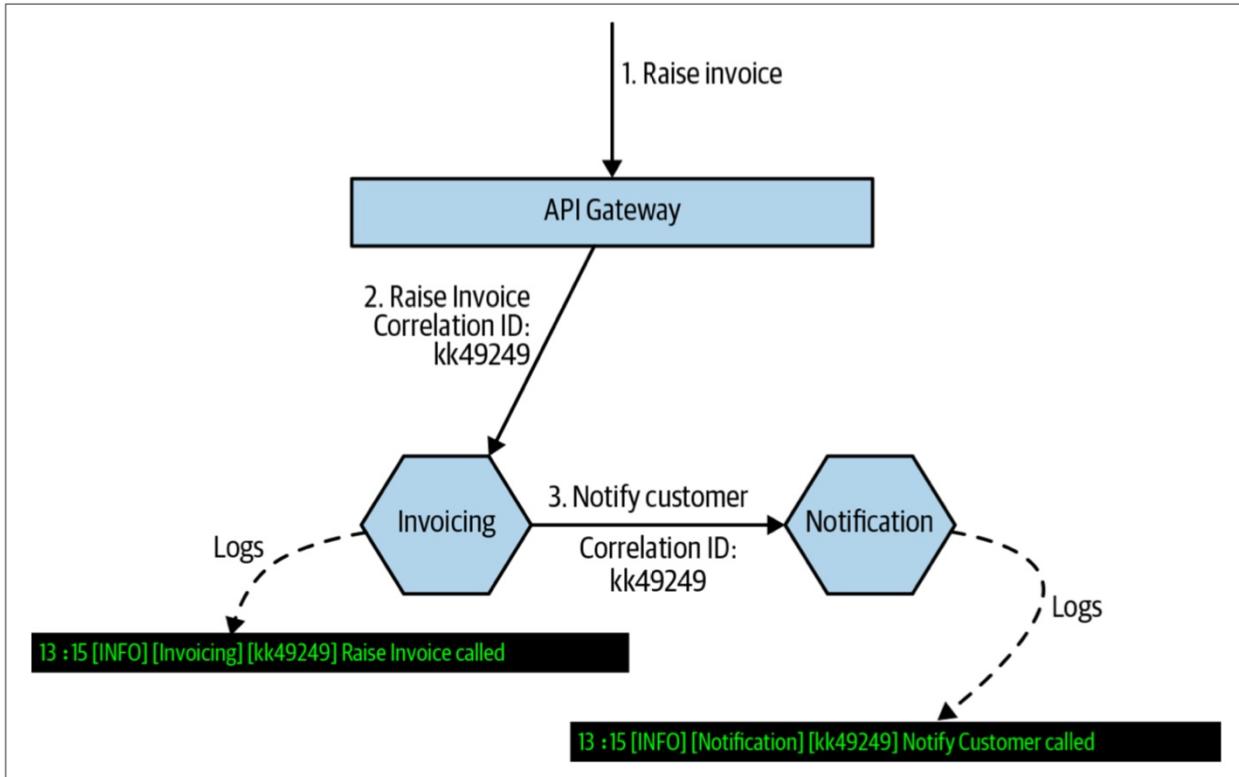


图5-6. 使用关联ID来确保某个特殊的调用链信息可以关联起来

Notification服务在处理请求时，可以使用相同的关联ID来记录该服务所做的事情，从而允许我们可以使用日志聚合系统来查询与给定关联ID相关联的所有日志（假设将关联ID置于日志格式中的标准位置）。当然，还可以使用关联ID执行其他操作，例如管理分布式事务sagas（如第4章所述）。

更进一步，我们可以使用工具来追踪请求的时间。由于日志聚合系统的工作方式——定期对日志进行批处理并转发到中央代理，因此日志聚合系统无法获取准确的信息以使我们可以精确的确定在调用链中的不同请求所花费的时间。分布式跟踪系统，例如图5-7所示的、开源的Jaeger，可以给予我们帮助。

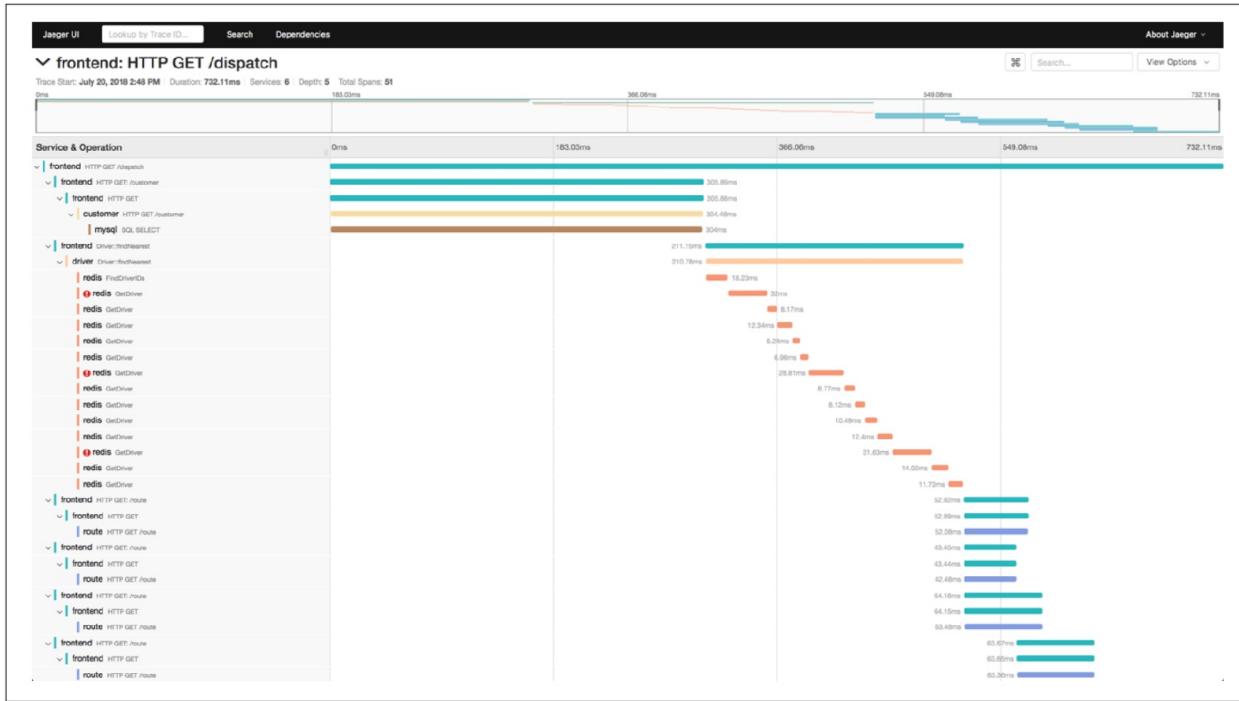


图5-7. Jaeger是一个用于捕获分布式跟踪信息并分析单个调用性能的开源工具

应用程序对延迟的敏感度越高，越希望尽快实现像Jaeger这样的分布式跟踪工具。值得注意的是，如果已经使用了关联ID，并已经将其用于现有的微服务架构（在需要分布式跟踪工具之前，我通常会提倡这一点），那么我们现有的服务栈已经具备这种能力，对其进行简单修改就可以将数据推送到合适的工具。也可以使用service mesh，虽然service mesh无法很好地处理单个微服务内部的调用跟踪，但是service mesh至少可以为我们处理入站和出站跟踪。

## 线上测试

功能自动化测试通常用于在部署之前给我们提供反馈以确定我们的软件是否具备部署的质量。但是，一旦上线，我们仍然希望获得同样的反馈！即使给定的功能在线上曾经是可用的，新的服务部署或环境修改也可能会在以后破坏该功能。

通过把假的用户行为注入到我们的系统，我们可以用人造流量的方式来定义我们所期望的行为，并在预期行为没有发生时给出相应的报警。在我以前工作过的Atomist公司，对于新用户，我们有某些复杂的新用户引导流程，该引导流程需要用户使用其GitHub和Slack帐户对我们的软件授权。而该引导过程的早期，会有足够多的服务出现问题，例如GitHub API的限速。我的同事，Sylvain Hellegouarch，编写了注册假用户的脚本。我们会定期让其中的某个假用户触发注册流程，而整个端到端的过程会编写成脚本。如果注册失败，通常表明我们的系统出了问题。此时，“假”用户比正常注册的用户可以更好地发现问题！

线上测试的一个很好的起点是：采用现有的、端到端的测试用例，并对其进行修改以使其可以用于生产环境。重要的是需要确保这些“测试”不会对线上造成意外影响。在Atomist，我们创建了GitHub和Slack帐户，我们可以控制这些账号来合成流量，因此没有真正的用户会参与或受到影响，并且对于我们的脚本而言，之后可以轻而易举的清除这些帐户。另一方面，我确实听说过关于一家公司的报道，因为他们没有考虑到测试订单也会发货这一事实，因此该公司最终意外订购了发往他们总部的200台洗衣机。所以，线上测试时一定要小心！

## 向着可观察性发展

在传统的监控和报警过程中，我们要考虑可能出了什么问题，然后采集信息以告诉我们什么时候会出现这种情况，并以此来触发报警。因此，我们主要是在处理那些已知原因的问题——磁盘空间不足，服务实例无响应，或出现延迟峰值。

随着我们的系统变得越来越复杂，越来越难以预测系统可能无法正常工作的所有原因。此时，重要的是，允许我们在出现这些问题时询问系统开放式问题，以帮助我们首先止损并确保系统可以继续运行，同时也允许我们

收集足够的信息来进一步解决问题。

因此，我们需要能够采集有关系统运行状况的大量信息，使我们能够在事后对我们所不知道的数据提出问题。请求跟踪和日志可以构成重要的数据源，我们可以从中提出问题并使用真实的信息，而不用靠猜测来确定问题所在。秘诀在于使这些信息易于在上下文中查询并查看。

不要以为我们会预先知道答案。然而，如果我们以为我们会预知答案，我们将会感到惊讶。因此擅长于向系统提出问题，并确保可以使用工具链来执行即席查询（*ad hoc querying*）。如果想更详细地探讨这个概念，我建议将Distributed Systems Observability一书作为一个很好的起点<sup>2</sup>。

### ad hoc

ad hoc 一般都说是查询，那么到底什么是即席查询呢？

在wikipedia上的解释为：ad hoc允许终端用户自己去建立特定的、自定义的查询请求。通常是通过一个用户友好的图形界面来进行数据查询，而无需用户对SQL 或者数据库架构有深入的了解。

ad hoc查询通常是临时的，有特殊目的的，一般其查询计划也无法重复利用。

<sup>2</sup>. See Cindy Sridharan, Distributed Systems Observability (Sebastopol: O'Reilly Media, Inc., 2018). ↵

# 本地开发体验的问题

随着服务越来越多，开发人员的体验可能会开始变差。像JVM这样的资源密集型的runtimes会限制开发人员的单台机器上可以运行的微服务的数量。我可以在笔记本电脑上以单独的进程运行四个或五个基于JVM的微服务，但是我可以运行十个或二十个吗？可能不行。即使使用占用资源较少的runtimes，可以在本地运行的服务数量也有限制。当我们无法在一台机器上运行整个系统时，我们会不可避免地开始讨论该怎么办。

# 该问题如何表现出来

由于必须维护更多的服务，因此需要更长的时间来执行本地构建并运行服务，每天的开发进度可能会开始放慢速度。开发人员将开始需要更大性能的机器来出来解决他们必须处理的服务数量。对于短期方案而言，这种方式是可以的。但是，如果我们的服务数量在持续的增加时，采用更大性能的机器仅仅可以为我们争取一点时间而已。

# 该问题何时会发生

这个问题何时能确切的表现出来取决于开发人员希望在本地运行的服务数量以及这些服务的资源占用量。使用Go, Node或Python的团队很可能会发现他们可以在遇到资源约束之前在本地运行更多服务——但是使用JVM的团队可能会更早的遇到此问题。

我还认为，对多个服务实行集体所有制的团队更容易受到这个问题的影响。在开发过程中，这些团队更需要在不同服务之间切换的能力。对少量的服务实行强代码所有制的团队通常只会专注于自己的服务，并且更有可能开发出机制来打桩其无法控制的外部服务。

# 该问题如何解决

如果我想在本地开发，但是想减少在本地必须运行的服务数量，那么一种常见的技术就是为那些不想运行在本地的服务“打桩”，或者使用运行在其他地方的服务实例。纯粹的远程开发可以让我们对托管在功能更强大的基础设施上的许多服务进行开发。但是，随之而来的挑战是：

- 远程开发的联通性，这对于远程工作人员或经常出差的人来说可能是一个问题
- 从远程部署软件到发现服务可以正常运行的反馈周期较慢
- 开发人员所需的开发资源或与其相关联的资源成本的爆炸性增长

[Telepresence](#)是一个旨在让Kubernetes用户可以更轻松地进行本地/远程混合开发的工具。我们可以在本地开发服务，Telepresence可以将对其他服务的调用代理到远程群集，从而使得我们可以两全其美。Azure的云功能也可以在本地运行，但可以连接到远程的云资源，从而使我们可以利用快速的本地开发流程创造出的功能来创建服务，同时仍然可以在云环境中运行这些服务。

## telepresence

kubernetes集群内部的微服务的特点是，各微服务之间在集群内可以互相访问，集群外部没有很好的方法来获取集群内部的功能，比如获取数据库中的数据。当然，实际上，借助于service的nodePort可以实现。

此外，当前的CICD流程步骤为：提交代码 -> jenkins 拉取源码 -> maven 编译 -> docker编译 -> 部署 多个步骤，虽然该过程由jenkins自动完成，但是每次调试均需要经历该步骤，降低效率。

telepresence是一款为kubernetes微服务框架提供快速本地化开发功能的开源软件。通过telepresence，可以实现在本机运行本地代码，本地代码能够获取远端k8s集群的各项资源。说白了，telepresence就是给本机提供了k8s集群的代理，使本机直连到远程k8s集群，从而访问集群内部资源。

认识到开发体验会随服务数量的增加而发生的变化非常重要，因此我们需要适当的反馈机制。我们需要持续的投入，以确保随着开发人员所使用的服务数量的增加，他们仍然能保持尽可能高的生产力。

Copyrights © wangwei all right reserved

# 微服务部署和状态管理的问题

随着拥有更多的服务以及服务实例，我们将需要部署、配置和管理更多的进程。当服务的数量不断增加时，用于处理单体应用程序的部署/配置的现有技术可能无法很好地扩展。

尤其是服务的期望状态管理（DS: *desired state*）会越来越重要。DS管理使我们能够指定所需的服务实例的数量和位置，并确保可以随着时间的推移而保持这种状态。当前，我们可以手动管理单体的DS，但是当拥有数十个或数百个微服务时，尤其是在每个微服务具有不同的DS时，手动管理DS的方法无法很好地扩展。

# 该问题如何表现出来

我们将开始发现：越来越多的时间会花在管理部署和解决部署过程中出现的问题上。如果该过程依赖于手工操作，则总是会犯错误——而且意外的错误对分布式系统的影响很难预测。

随着不断增加的服务和服务实例，我们会发现自己需要更多的人来管理与部署&维护这些服务相关的行为。这可能导致需要更多人来支持我们的运维团队，或者可能会发现交付团队花在部署问题上的时间占比会更高。

# 该问题何时会发生

一切都与服务规模有关。拥有的微服务越多，并且这些微服务的实例越多，手动处理或更多传统的自动化的配置管理工具（例如Chef和Puppet）将不再适用。

# 该问题的解决方案

需要一个可以实现高度自动化的工具，该工具可以让开发人员最好地进行自助服务式的部署，并实现自动化的DS管理。

对于微服务而言，Kubernetes已成为该领域的首选工具。K8s要求我们对服务进行容器化，但是一旦完成，我们就可以使用K8s在多台机器上管理服务实例的部署，并确保服务可以扩展以提高其鲁棒性和解决负载问题（假设有足够的硬件资源）。

我认为原生K8s对开发人员不友好。许多人都在从事更高层次的、对开发人员更友好的K8s抽象工作，我希望这项工作能继续下去。未来，我希望许多开发人员甚至都不会意识到他们的软件是运行在K8s上，因为K8s将仅仅是实现细节。我倾向于看到较大的组织采用K8s的发行版本，例如RedHat的OpenShift。K8s的发行版本将K8s与工具捆绑在一起，从而更容易应用于企业内部环境——还可以解决企业认证和访问管理控制。其中一些发行版本还为开发人员提供了简单化的抽象。

## Vanilla Kubernetes

Vanilla Kubernetes指纯净、原生的Kubernetes，一般还有Vanilla JavaScript/Vanilla Linux等用法，指原生JavaScript或Linux，而不是它们的方言版或发行版本。

原生Kubernetes指的是Kubernetes的原生未修改版本，提供源代码下载。

之所以称为原生版，是因为在软件界有一个长达几十年的传统，即打上“Vanilla”原生标签的软件被部署到任何应用程序或平台上时，表示这是没有修改过的官方版本。类似于，我们还会听到“原生Linux”，这是指使用纯粹的、官方的 Linux 内核源代码构建 Linux 内核，而不像在 Linux 发行版本中，会修改 Linux 内核程序。

与原生 Kubernetes 相对的是 Kubernetes 发行版，例如 Rancher，Red Hat OpenShift，或基于云的 Kubernetes 服务，例如 Amazon EKS。这些发行版采用了开源 Kubernetes 代码，并将其集成到更广泛的平台中，而这些平台通常包含不属于 Kubernetes 本身的管理、监视和安全工具。这些平台中的很多平台还提供安装程序，简化 Kubernetes 安装程序。

当然，最近也有人提出了不适用发行版本的 Kubernetes 的 5 个理由，具体可以参考：[5 Reasons Not to Use Kubernetes Distributions](#)

如果有幸使用公共云，则可以使用公有云提供的很多不同的选择来处理微服务架构的部署，包括托管的 K8s 产品。例如，AWS 和 Azure 在 K8s 托管领域都提供了多个选择。我非常喜欢“功能即服务”（FaaS），FaaS 是所谓的“serverless”的子集。使用合适的平台，开发人员只需要关心代码，大部分的运维工作则由底层的平台来处理。尽管目前的 FaaS 产品确实有其局限性，但它们仍然为运维工作的急剧减少提供了希望。

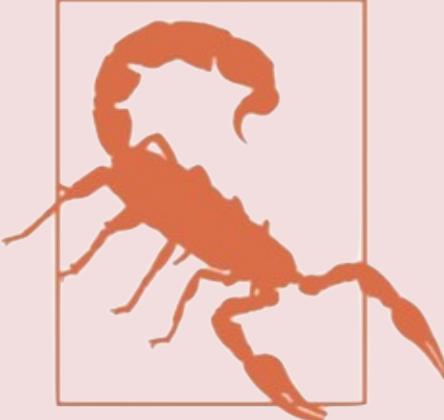
与那些我与之合作的、已经使用共有云的团队，我并不倾向于从 K8s 或类似的基于容器的平台开始。我会优先采用 serverless 的方法——因为可以减少运维的工作量，因此会尝试将诸如 FaaS 之类的 serverless 技术作为默认选择。如果我们的问题不满足我们所使用的 serverless 产品的限制，那么请寻

找其他的选择。显然，并非所有的问题空间都是相同的。但是我认为，如果我们已经在公共云上，我们可能并不总是需要像K8s这样的基于容器的平台的复杂性。

### Problem space & Solution space

关于problem space和solution space的区别可以参考：[PROBLEM SPACE vs SOLUTION SPACE](#)。

problem space，简单理解就是当前环境下，业务所面临的一系列问题和其背后的需求。solution space，则是针对问题空间的解决方案，思考的是如何设计并实现软件系统以解决这些问题，属于工程设计实施阶段，通常是技术专家主导的解决方案设计和实现。



我确实看到人们在采用微服务的过程中会过早的接触K8s，人们通常认为K8s是微服务的前提条件。然而，并非如此，像K8s这样的平台擅长帮助我们管理多个服务，但是我们应该等到拥有足够的服务以至于当前的方法和技术开始捉襟见肘时再考虑K8s这样的平台。我们可能会发现我们只需要五个微服务，并且可以使用现有解决方案轻松地解决问题——在这种情况下，那就太棒了！不要仅仅因为看到其他所有人都在使用基于K8s的平台而使用这些平台，对于微服务而言，也是如此！

Copyrights © wangwei all right reserved

# 端到端测试

任何类型的自动化功能测试，都需要一个完美的平衡点。测试执行的功能越多，测试的范围越广，我们对应用程序的信心就越高。另一方面，测试范围越大，测试运行的时间就越长，同时在测试失败时找出失败的原因就会越困难。

就其覆盖的功能而言，任何类型的系统的端到端测试的覆盖范围都是最大的。并且我们通常认为，相比与覆盖范围更小的单元测试而言，端到端测试更难以编写和维护。但是，端到端测试通常是值得的，因为我们希望通过执行像用于一样使用我们的系统的端到端测试来获得信心。

但是，使用微服务架构时，我们的端到端测试的“范围”变得非常大。现在，我们必须跨多个服务运行测试，所有这些服务都需要为测试场景而部署并做适当的配置。我们还必须为环境问题（例如服务实例停止运行、部署失败的网络超时）导致的测试失败进而产生的错误判断做好准备。我认为与标准的单体架构相比，在针对微服务架构进行端到端测试时，我们更容易受到无法控制的问题的影响。

随着测试范围的扩大，我们将花费更多的时间来应对出现的问题，以至于尝试创建和维护端到端的测试变得非常耗时。

# 该问题如何表现出来

一个迹象就是：端到端的test suite不断增加，并且需要越来越长的时间才能完成测试。这是由于多个团队无法确定覆盖的测试场景，而为了“以防万一”而增加新的test suite。我们会在端到端的test suite中发现更多的测试失败，这些失败并没有突出标记出代码问题——所以，开发人员常常只是再次运行测试以查看是否通过。

花费在端到端测试上的时间越来越长，以至于我们开始看到更多测试人员甚至是单独的测试团队所面临的压力。

# 该问题何时发生

这个问题往往会悄悄地出现，但是我发现，在由多个团队处理用户的不同操作时，该问题表现的最为强烈。每个团队的工作越独立，他们就越容易在本地管理自己的测试。越是需要跨团队的测试流程，大范围的端到端测试就越是问题。

# 该问题的解决方案

在《Building Microservices》一书中，我概述了很多方法来帮助我们改变测试的方法。实际上，在书中，有一整章专门对其进行了介绍。但是，此处有一个帮助大家入门的简短的摘要。

## 限制功能自动化测试的范围

如果要编写覆盖多个服务的test case，请尝试确保将这些测试限制在管理这些服务的团队内部。换句话说，要避免跨团队边界的、更大范围的测试。将测试的所有制控制在一个团队内部，可以：

- 更容易地了解正确覆盖了哪些场景
- 确保开发人员可以运行和调试测试
- 更清楚地阐明了谁应该为确保测试可以运行并通过而负责

## 使用消费者驱动的契约

可能会考虑使用消费者驱动的契约（CDC: *consumer-driven contracts*）来代替跨服务的test case。使用CDC，可以让微服务的使用者根据可执行的规范来定义他们所期望的我们的服务的行为，该可执行的规范就是一个测试。我们在修改服务时，要确保这些测试仍然是通过的。

由于这些测试是从消费者的角度定义的，因此我们可以很好地弥补意外的契约破坏。我们还可以从消费者的角度了解他们的需求。重要的是可以了解：不同的消费者可能希望从我们这里获得不同的东西。

可以使用简单的开发流程来实现CDC，但是可以使用支持该技术的工具来简化CDC。最好的工具可能是[Pack](#)。

值得注意的是，我已经发现很多团队利用CDC取得了巨大成功，但是对于其他的团队而言，采用CDC却非常困难。CDC是合理的，我知道，它可以很好地工作。但是，我还没有完全理解某些人在采用这种技术时所面临的挑战。对于解决一个非常困难的问题，CDC仍然是一种未被充分利用的实践。

## 使用自动发布修复和渐进式交付

利用自动化测试，我们通常试图在问题影响产品之前就发现问题。但是，随着系统变得越来越复杂，做到这一点也变得越来越困难。因此，如果确实发生了线上问题，则值得花费精力来减少其影响。

正如我们在[第3章](#)中谈到的那样，渐进式交付是用于控制如何以增量方式向客户推出软件的新版本的总称。渐进式交付的想法是，我们可以用一组少量的客户来评估新版本的影响，并确定是否以及何时可以继续发布新版本或者回滚新版本。渐进式交付的例子是金丝雀发布。

通过为“服务应该如何表现”定义可接受的度量，就可以用自动化的方式控制渐进式交付。举一个简单的例子，可以把95分位的延迟和错误率定义为可接受的阈值，并且仅在满足这些指标时才继续发布新版本。否则，可能会自动回滚到最近的版本，从而让我们有时间去分析所发生的事情。

许多组织使用了这种自动发布修复技术。特别的，Netflix公司就对使用该技术做了详尽的介绍。Netflix开发了Spinnaker作为部署管理工具，该工具可以帮助控制服务的渐进式交付，但是可以通过许多其他的方式将这些想法付诸实践。

我并不是说应该考虑自动发布修复，而不是进行测试。我只是说，要考虑可以从哪里获得最大的收益。如果问题确实发生了，将工作置于问题的捕获而不是仅仅着眼于阻止问题的发生，可能最终会得到一个更强大的系统。

需要注意的是，尽管这些技术可以很好地协同工作。即使我们认为自动修复不是我们需要马上做的事情，但实施某种形式的渐进式交付仍然具有巨大的价值。即使只是手动控制渐进式交付，也可以在仅将新软件发布给所有人上迈出了一大步。

## 持续提升质量反馈周期

了解如何测试以及在何处测试是一个持续的挑战。需要有背景 (*context*) 的人在整个开发过程中全面了解，以适应应用程序的测试方式和位置。这意味着我们需要这样的人员：

- 当他们发现产品在某个区域的缺陷不断增加时，他们能够识别出需要增加新的测试来覆盖该区域的需求。
- 在努力提升质量反馈周期时，当他们发现有的测试已经被覆盖到时，他们也需要删除该测试。

简而言之，这是关于平衡快速反馈与安全性的事情。需要像添加新测试一样去识别、删除、替换错误的测试。

Copyrights © wangwei all right reserved

# 局部优化 VS. 全局优化

假设团队对自己的本地决策负有更多的责任，或许团队会拥有他们所管理的微服务的整个生命周期，那么我们将需要开始在本地决策与更多的全局指标之间做出平衡。

作为该问题的一个例子，我们来考虑下管理Invoicing, Notifications和Fulfillment服务的三个团队。因为Invocing团队特别了解Oracle，因此，该团队决定采用Oracle作为数据库。因为MongoDB特别适合Notifications团队的编程模型，因此该团队希望使用MongoDB。然而，因为Fulfillment团队已经在使用PostgreSQL了，因此，他们希望使用PostgreSQL。当我们依次查看每个决策时，这很有意义，同时我们可以了解该团队是如何做出这种决策的。

但是，如果退一步并放眼大局，我们必须问问自己：作为一个组织，我们是否要为某些相似的功能引入3种数据库，为其构建相应的技能并支付许可费用。仅采用一个数据库是否会更好呢？对于所有人而言，仅采用一个数据库并不完美，但是对于大多数人来说这足够好了。如果没有能力查看本地发生的事情，并且又无法将本地发生的事情放于全局空间，那么我们将如何做出此类决策？

# 该问题如何表现出来

我发现这个问题的最常见的方法是：当有人突然意识到多个团队以不同的方式解决了相同的问题，却从未意识到他们都在试图解决同一问题。随着时间流逝，这可能会变得非常低效。

我记得和澳大利亚房地产公司REA的人交谈过。经过多年的微服务构建，他们意识到自己的团队可以通过多种方式部署服务。当人们从一个团队转到另一个团队时，这会引起问题，因为他们必须学习新的做事方式。但是，也很难证明每个团队正在做重复的工作。结果，他们决定进行某些工作以提出一种通用的方法来解决此问题。

通常，可能是在午餐时无意偷听了一段评论之后，偶然发现了这些事情。如果拥有某种跨团队的技术小组，例如“[实践社区](#)”，则可以更早地发现这些问题。

# 该问题何时发生

随着时间的流逝，在多团队的组织中往往会出现此问题，尤其是为团队提供更多的自由来决定其工作方式的组织。不要期望在微服务旅程的早期就发现此问题。一开始，我们对如何做事可能会有一个清晰的共同理解。随着时间的推移，每个团队将越来越专注于自己的本地问题，并将基于本地问题来优化他们解决问题的方式，以至于“这就是我们的工作方式”的核心共识开始发生变化。

在组织经历了一段时间的扩张之后，我经常发现人们正提出并讨论该问题。在短时间内涌入大量开发人员使之前临时的信息共享难以扩展。这可能导致需要桥接更多的信息孤岛。

因为集体服务所有制在解决问题上需要一定程度的一致性。因此，如果我们采用集体服务所有制，那么可能会帮助避免或至少限制这些问题。换句话说，如果我们想采用集体所有制，则必须解决此问题。否则，我们的集体所有制将无法扩展。

# 该问题的解决方案

我们已经讨论了一些可以在此方面有所帮助的想法。在[第2章](#)中，我们探讨了不可逆决策和可逆决策的概念，如[图5-8](#)所示。变革成本越高，影响越大，我们越希望在决策背后达成更广泛的共识。影响越小，回滚就越容易，本地团队就可以做更多的决策。

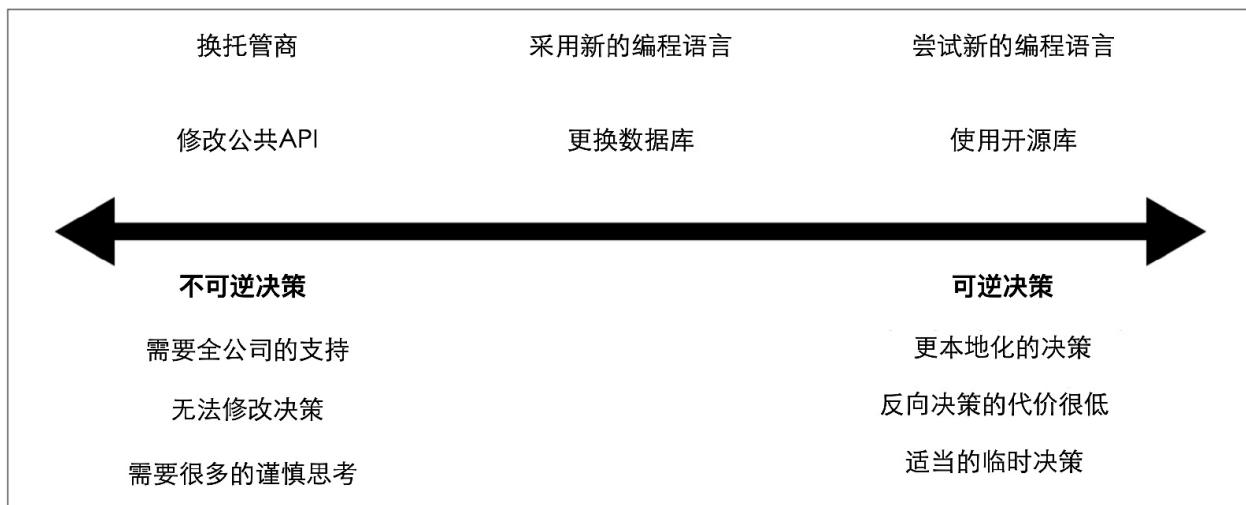


图5-8. 可逆决策和不可逆决策的差异及其在决策频谱图中的例子

诀窍是帮助团队中的人们认识到他们的决策是趋向于不可逆决策呢还是趋向于可逆决策。决策越是趋向于不可逆决策，让团队边界之外的其他人参与决策就变得越重要。为了使该方法可以正常运行，需要团队至少对大局问题有基本的了解，以了解他们可能和大局问题重叠的地方。同时，团队还需要一个网络，在该网络中，他们可以揭示这些问题并从其他团队的同事的参与中获得帮助。

作为一种简单的机制，明智的做法是：每个团队至少有一位技术负责人成为跨部门技术小组的成员，而该跨部门的技术小组用于解决本章节涉及到的问题。该小组可以由CTO，CA（首席架构师，chief architect）或负责公

司总体技术构想的其他人担任主席。

这个跨部门小组可以双向工作。除了提供一个让团队可以在更大的论坛上讨论他们想讨论的本地问题的地方之外，人们还可以在这里找到跨领域的问题。如果团队之间没有任何沟通，我们又如何认识到我们正在以不同的方式在本地解决问题，或许，在全局范围内解决问题可能更明智？

根据组织的性质，我们可以依赖临时的非正式流程。例如，在Monzo，人们可以提交在内部称为“提案”的自由格式文档。这些提案会发布到共享空间，并通过Slack提醒整个公司有新提议可用。对提案感兴趣的各方可以讨论该提案并进行完善。期望这些建议并不是完美的，实际上必须是可以改变的。对于Monzo来说，这似乎行之有效，部分原因是该公司围绕沟通和责任分担的文化。

从根本上讲，每个组织都需要在全局决策和本地决策之间找到适当的平衡。愿意让团队承担多少责任？想让高层控制多少责任？让团队承担的责任越多，获得更大自治权的好处就越多。但是，此时需要权衡的是：在解决问题的方式上可能缺乏一致性。越从高层推动事情，就越需要达成共识，这很可能会降低事情的推动速度。我无法告知大家如何以适合自己的方式在这两种力量之间取得平衡，大家需要自己解决该平衡。只需要知道这种平衡的存在，同时需要确保收集正确的信息，以确保可以随着时间而调整这种平衡。

Copyrights © wangwei all right reserved

# 系统的鲁棒性和弹性

如果更习惯于单体系统，则分布式系统可能会出现很多我们不熟悉的故障模式。网络可能会丢包，网络调用可能会超时，机器可能会死机或停止响应。在简单的分布式系统（例如传统的单体应用程序）中，这些情况可能很少见。但是，随着服务数量的增加，这些罕见的情况会变得越来越普遍。

# 该问题如何表现出来

不幸的是，这些问题最有可能出现在生产环境。在传统的开发和测试周期中，我们仅在短时间内重新创建了类似线上的环境。这些罕见的事件不太可能在线下出现，并且他们通常在发生之后就会消失。

# 该问题何时发生

诚实讲——如果我能提前告诉大家，你们的系统何时会受到不稳定的困扰，我就不会写这本书了。因为，我可能会将写书的时间用于在海滩的某个地方喝鸡尾酒。我只能说，随着服务数量的增加和服务调用的数量的增加，会越来越容易受到弹性问题的困扰。服务之间的交互越多，遭受到连锁故障和“back pressure”之类的事情的可能性就越大。

## back pressure

back pressure是一种现象：在数据流从上游生产者向下游消费者传输的过程中，上游生产速度大于下游消费速度时，导致下游的Buffer溢出，这种现象就叫做back pressure。

back pressure的概念源自工程概念中的back pressure：在管道运输中，气流或液流由于管道突然变细、急弯等原因导致由某处出现了下游向上游的逆向压力，这种情况称作 back pressure。这是一个很直观的词：向后的、往回的压力。可是，国内的热力工程界对这个词的正式翻译是 背压，把 back 翻译成了 背，着实有点让人迷惑。

因此，这里就不翻译back pressure一词了。

back pressure指的是在Buffer有上限的系统中，Buffer溢出的现象，它的应对措施只有一个：丢弃新事件。

1. 生产速度大于消费速度，所以需要Buffer；
2. 外部条件有限制，所以Buffer需要有上限；
3. Buffer达到上限这个现象，有一个简化的等价词叫做back

pressure;

4. back pressure的出现其实是一种危险边界，唯一的选择是丢弃新事件；

如上，就是back pressure的本质，back pressure的出现其实是一种危险边界，唯一的选择是丢弃新事件。

# 该问题的解决方案

一个好的出发点是：问自己一些关于我们所有的服务调用的相关问题。

1. 我知道调用失败的方式吗？
2. 如果调用确实失败了，我知道该怎么办吗？

回答完这些问题后，就可以开始寻找各种解决方案了。服务彼此之间的隔离会有所帮助，或许包括引入异步通信以避免时间耦合（我们在[第1章](#)中谈到了这一主题）。使用合理的超时可以避免较慢的下游服务的资源冲突。同时，结合断路器之类的模式，可以启动快速失败，以避免back pressure问题。

运行多个服务副本可以帮助解决实例宕机的问题，可以使用平台实现DS管理，以确保服务在crash时可以重新启动。

重申第二章中的观点，弹性不仅仅是实现某些模式。弹性一套完整的工作方式——建立一个组织，该组织不仅要准备处理不可避免的问题，而且还要根据需要改进工作方法。可以将这种想法付诸实践的一种具体方法是：在出现线上问题时记录下来，并记录所学到的知识。我常常发现，组织一旦解决了最初的问题，就快速的前进了——仅仅是几个月之后，同样的问题就又出现了。

之前，我发现某个团队在ES6的兼容性问题上连续出现了多次线上问题。

老实说，在这里，我只是做了一点肤浅的研究。对于这些想法的更详细的研究，我建议阅读《Building Microservices》一书的第11章；或者看看 Michael Nygard的《Release It！》，该书由Pragmatic Bookshelf在2018年发行。

Copyrights © wangwei all right reserved

# 孤儿服务的问题

我们拥有令人惊奇的技术，同时我们正在构建极其复杂和可大规模扩展的系统，但仍然会出现一些最平淡无奇的问题，这看起来很奇怪。举个例子，我发现，许多组织都为如下的事情而努力：准确地知道自己拥有什么，他们在哪里，谁拥有他们。

随着微服务越来越专注于它们的用途，我们会发现，越来越多的服务已经愉快地运行了数周、数月甚至数年，而无需对其进行任何修改。一方面，这正是我们想要的。独立部署是一个很吸引人的概念，部分原因是它使系统的其余部分保持稳定，并且保持系统中不需要修改的部分的稳定性是一个好主意。

我将这些经年未动的服务称为孤儿服务，因为从根本上说，公司中没有人对这些服务负责。

# 该问题如何表现出来

我记得曾经听到（也许是伪造的）一个关于在旧的办公室里发现旧服务器的故事。没有人能记得起那里还有服务器，但是这些服务器仍然快乐地运行着，做着他们之前做的事情。没有人能确切地记得这些新发现的服务器的功能，人们不敢将其关闭。微服务也会表现出某些类似的特征；我们不知道这些服务在哪里，并且（我们假设）他们正在工作中。但是，我们有一个同样的问题：我们可能不知道该如何处理它们，并且这些担心会阻止我们修改这些服务。

根本问题是，如果该服务确实停止工作了或需要进行修改，那么人们将无所适从。我已经与不止一个团队交流过，这些团队分享了他们不知道所涉及服务的源代码在哪里的故事，这是一个很大的问题。

# 该问题何时发生

对于长期使用微服务的组织来说，通常会出现此问题——时间太长了，以至于对该服务的用途的集体记忆都被时间冲散了。与此微服务有关的人员，或者忘记了对该服务做了什么，或者也许离开了公司。

# 该问题的解决方案

我有一个（未经尝试的）假设，即实践服务的集体所有制的组织可能不容易出现孤儿服务的问题，主要是因为他们已经实现了允许开发人员在不同服务之间流转并对其修改的机制。这类组织可能已经限制了语言和技术选择，以减少在服务之间切换上下文的成本。他们可能有通用的工具来对服务进行修改、测试和部署。如果自上次修改服务以来这些常规做法已发生变化，那么这当然可能无济于事。

我曾与许多遇到孤儿服务问题的公司进行过交谈，最终，这些公司创建了简单的内部注册表，以帮助整理服务的元数据。有些注册表只是简单地抓取源代码仓库，寻找元数据文件来构建服务列表。可以将这些信息与来自服务发现系统（如consul或etcd）的真实数据合并，以更全面地了解正在运行的服务以及可以与谁谈论该服务。

英国《金融时报》创建了Biz Ops，以帮助解决孤儿服务的问题。该公司拥有由全球不同的团队开发的数百个服务。Biz Ops工具（[图5-9](#)）为他们提供了一个单一的场所，除了有关网络和文件服务器等其他的IT基础设施服务的信息之外，还可以在该工具中找到有关其微服务的许多有用的信息。这些信息建立在图数据库之上，图数据库在收集哪些数据以及如何对信息进行建模方面具有很大的灵活性。

**FT Biz Ops Admin**

ABOUT SEARCH MY STUFF ADD SOMETHING REPORT (BETA)

System: FT.com Article Page

General information

Ownership & knowledge

Technical overview

Data governance

Related resources

Failover

Data recovery

Release

Key Management

Monitoring

Troubleshooting

More Information

Service Operability Review

Miscellaneous

Show the definition of "System" ▾

[View runbook](#) [View Heimdall dashboard](#)

## System: FT.com Article Page

### General information

**Code** ⓘ next-article

**Name** ⓘ FT.com Article Page

**Description** ⓘ A service that provides the user facing layer of articles for [FT.com](#)

**Primary URL** ⓘ <https://www.ft.com/content/00b2e3c8-e2b0-11e8-a6e5-792428919cee>

**Service tier** ⓘ Platinum

**Lifecycle stage** ⓘ Production

### Ownership & knowledge

**Delivered by team** ⓘ [Next](#)

**Supported by team** ⓘ [Next](#)

**Stakeholders** ⓘ

**Known about by** ⓘ

### Technical overview

**Host platform** ⓘ Heroku

**Architecture** ⓘ

next-article serves the following routes: ^/content/([a-f0-9]+)-([a-f0-9]+)-([a-f0-9]+)-([a-f0-9]+)-([a-f0-9]+)-([a-f0-9]+)-([a-f0-9]+)

[Delete](#) [Edit](#)

图5-9. 《金融时报》用于整理有关其微服务信息的Biz Ops工具

但是，Biz Ops工具比我所见过的大多数工具都更先进。Biz Ops工具计算所谓的系统可操作性得分，如图5-10所示。其想法是，服务及其团队应做某些事情，以确保可以轻松地运维服务。这些事情从确保团队在注册表中提供正确的信息到确保服务进行正确的运行状况检查。通过计算这些分数，对是否有需要修复的地方，团队可以做到一目了然。

This project is currently in BETA, please provide feedback, or volunteer to take part in UX testing, in the #ops-and-rel slack channel.

**FT System Operability Score** powered by *Biz Ops*

ALL GROUPS TEAM LEAGUE TABLE SYSTEM LEAGUE TABLE ABOUT

ALL GROUPS > ENTERPRISE SERVICES > CLOUD ENABLEMENT > FTPLATFORM2

**System: ftplatform2** Silver Production

Score: 92%

Manual review status: Ready for ops review 🎉  
Score updated: 42 minutes ago

Bear in mind that the scoring rules are a work in progress. In some cases (particularly with regard to third party systems) they won't always make perfect sense, so use your discretion. Not getting 100% is normal for now. Fixing all the things you know how to fix is the main thing to aim for.

Runbook aspect	Status	Failings	Remedial actions
dependents	Error	Some dependents have a serviceTier above Silver	-
healthchecks	Warning	healthchecks has Healthcheck without URL	-
moreInformation	Info	moreInformation contains HTML - prefer to use markdown moreInformation contains links moreInformation contains references to non-platinum documentation (Confluence, google sites)	-
monitoring	Info	monitoring contains HTML - prefer to use markdown monitoring contains links	-
firstLineTroubleshooting	Info	firstLineTroubleshooting contains links	-
secondLineTroubleshooting	Info	secondLineTroubleshooting contains links	-

图5-10. 《金融时报》微服务的服务可操作性得分的例子

服务注册表之类的工具会有所帮助，但是如果孤儿服务比注册表出现的还要早，会发生什么呢？关键是要使这些新发现的“孤儿”服务与其他服务的管理方式保持一致，这要求我们要么将孤儿服务的所有权分配给现有团队（如果实行强服务所有制），要么提出该服务需要改进的工作项目（如果采用集体服务所有制）。

Copyrights © wangwei all right reserved

# 结束语

就这样，我们来到了本书的结尾。在整个过程中，我希望能传达两个关键信息。

- 首先，给自己足够的空间并收集正确的信息以做出合理的决定。不要只是复制别人的方案。反之，请考虑我们自己的问题和上下文，评估各种选择并继续前进，同时如果以后需要的话，我们也可以做出改变。
- 其次，请记住，逐步采用微服务、许多相关技术以及实践是关键。没有两个相同的微服务架构——尽管可以从其他人工作中吸取教训，但是我们需要花一些时间来找到一种适合我们的情况的方法。把整个迁移过程拆分为可管理的步骤，我们可以随时随地的调整我们的方案，因此，可以为我们提供最大的成功机会。

微服务绝对不是对所有人都适合。但是，希望阅读完这本书后，不仅可以更好地了解微服务是否适合我们，而且还可以获得有关如何开启微服务之旅的一些想法。

Copyrights © wangwei all right reserved