迁移到云原生应用架构

Migrating to Cloud Native Application Architectures

目录

第 1 章

云原生的崛起

> 软件正在吞噬世界。
>
> —— Mark Andreessen

近些年来，在一些长期由领导者支配的行业中，这些领导者的领先地位已经岌岌可危，这都是由以这些行业为核心业务的软件公司造成的。像 Square、Uber、Netflix、Airbnb 和特斯拉这样的公司能够持续快速增长，并且拥有傲人的市场估值，成为它们所在行业的新领导者。这些创新公司有什么共同点？

- 快速创新
- 持续可用的服务
- 弹性可扩展的 Web
- 以移动为核心的用户体验

将软件迁移到云上是一种自演化，使用了云原生应用架构是这些公司能够如此具有破坏性的核心原因。对于云，我们指的是一个任何能够按需、自助弹性提供和释放计算、网络和存储资源的计算环境。云的定义包括公有云（例如 Amazon Web Services、Google Cloud 和 Microsoft Azure）和私有云（例如 VMware vSphere 和 OpenStack）。本章中我们将探讨云原生应用架构的创新性，然后验证云原生应用架构的主要特性。

1.1 为何使用云原生应用架构

首先我们来阐述下将应用迁移到云原生架构的动机。

1.1.1 速度

天下武功，唯快不破，市场竞争亦是如此。想象一下，能够快速创新、实验并交付软件的企业，与使用传统软件交付模式的企业，谁将在市场竞争中胜出呢？

在传统企业中，为应用提供环境和部署新版本花费的时间通常以天、周或月来计算。这种速度严重限制了每个发行版可以承担的风险，因为修复这些错误往往跟发行一个新版本有差不多的耗时。

互联网公司经常提到它们每天几百次发布的实践。为什么频繁发布如此重要？如果你可以每天实现几百次发布，你们就可以几乎立即从错误的版本恢复过来。如果你可以立即从错误中恢复过来，你就能够承受更多的风险。如果你可以承受更多的风险，你就可以做更疯狂的试验 —— 这些试验结果可能会成为你接下来的竞争优势。

基于云基础设置的弹性和自服务的特性天生就适应于这种工作方式。通过调用云服务 API 来提供新的应用程序环境比基于表单的手动过程要快几个数量级。然后通过另一个 API 调用将代码部署到新的环境中。将自服务和 hook 添加到团队的 CI/CD 服务器环境中进一步加快了速度。现在，我们可以回答精益大师 Mary Poppendick 提出的问题了 —— “如果只是改变了应用的一行代码，您的组织需要多长时间才能把应用部署到线上？“答案是几分钟或几秒钟。

你可以大胆想象 一下，如果你也可以达到这样的速度，你的团队、你的业务可以做哪些事情呢?

1.1.2 安全

光是速度快还是不够的。如果你开车是一开始就把油门踩到底，你将因此发生事故而付出惨痛的代价（有时甚至是致命的）！不同的交通方式如飞机和特快列车都会兼顾速度和安全性。云原生应用架构在快速变动的需求、稳定性、可用性和耐久性之间寻求平衡。这是可能的而且非常有必要同时实现的。

我们前面已经提到过，云原生应用架构可以让我们迅速地从错误中恢复。我们没有谈论如何预防错误，而在企业里往往在这一点上花费了大量的时间。在追寻速度的路上，大而全的前端升级，详尽的文档，架构复核委员会和漫长的回归测试周期在一次次成为我们的绊脚石。当然，之所以这样做都是出于好意。不幸的是，所有这些做法都不能提供一致可衡量的生产缺陷改善度量。

那么我们如何才能做到即安全又快速呢?

可视化

我们的架构必须为我们提供必要的工具，以便可以在发生故障时看到它。我们需要观测一切的能力，建立一个“哪些是正常”的概况，检测与标准情况的偏差（包括绝对值和变化率），并确定哪些组件导致了这些偏差。功能丰富的指标、监控、警报、数据可视化框架和工具是所有云原生应用架构的核心。

故障隔离

为了限制与故障带来的风险，我们需要限制可能受到故障影响的组件或功能的范围。如果每次亚马逊的推荐引擎挂掉后人们就不能再在亚马逊上买产品，那将是灾难性的。单体架构通常就是这种类型的故障模式。云原生应用架构通常使用微服务。通过将系统拆解为微服务，我们可以将任何一个微服务的故障范围限制在这个微服务上，但还需要结

合容错才能实现这一点。

容错

仅仅将系统拆解为可以独立部署的微服务还是不够的；还需要防止出现错误的组件将错误传递它所依赖的组件上而造成级联故障。Mike Nygard 在他的《Release It! - Pragmatic Programmers》一书中描述了一些容错模型，最受欢迎的是**断路器**。软件断路器的工作原理就类似于电子断路器（保险丝）：断开它所保护的组件与故障系统之间的回路以防止级联故障。它还可以提供一个优雅的回退行为，比如回路断开的时候提供一组默认的产品推荐。我们将在“容错”一节详细讨论该模型。

自动恢复

凭借可视化、故障隔离和容错能力，我们拥有确定故障所需的工具，从故障中恢复，并在进行错误检测和故障恢复的过程中为客户提供合理的服务水平。一些故障很容易识别：它们在每次发生时呈现出相同的易于检测的模式。以服务健康检查为例，结果只有两个：健康或不健康，up 或 down。很多时候，每次遇到这样的故障时，我们都会采取相同的行动。在健康检查失败的情况下，我们通常只需重新启动或重新部署相关服务。云原生应用架构不要当应用在这些情况下无需手动干预。相反，他们会自动检测和恢复。换句话说，他们给电脑装上了寻呼机而不是人。

1.1.3 弹性扩展

随着需求的增加，我们必须扩大服务能力。过去我们通过垂直扩展来处理更多的需求：购买了更强悍的服务器。我们最终实现了自己的目标，但是步伐太慢，并且产生了更多的花费。这导致了基于高峰使用预测的容量规划。我们会问”这项服务需要多大的计算能力?”然后购买足够的硬件来满足这个要求。很多时候我们依然会判断错误，会在如黑色星期五这类事件中打破我们的可用容量规划。但是，更多的时候，我们将会遇到数以百计的服务器，它们的 CPU 都是空闲的，这会让资源使用率指标很难看。

创新型的公司通过以下两个开创性的举措来解决这个问题：

- 它们不再继续购买更大型的服务器，取而代之的是用大量的更便宜机器来水平扩展应用实例。这些机器更容易获得，并且能够快速部署。
- 通过将大型服务器虚拟化成几个较小的服务器，并向其部署多个隔离的工作负载来改善现有大型服务器的资源利用率。

随着像亚马逊 AWS 这样的公有云基础设施的出现，这两个举措融合了起来。虚拟化工作被委托给云提供商，消费者只需要关注在大量的云服务器实例横向扩展它们的应用程序实例。最近，作为应用程序部署的单元，发生了另一个转变，从虚拟机转移到了容器。

由于公司不再需要大量启动资金来部署软件，所以向云的转变打开了更多创新之门。正在进行的维护还需要较少的资本投入，并且通过 API 进行配置不仅可以提高初始部署的

速度，还可以最大限度地提高我们应对需求变化的速度。

不幸的是，所有这些好处都带有成本。相较于垂直扩展的应用，支持水平扩展的应用程序的架构必须不同。云的弹性要求应用程序的状态短暂性。我们不仅可以快速创建新的应用实例；我们也必须能够快速、安全地处置它们。这种需求是状态管理的问题：一次性与持久性如何互相影响？在大多数垂直架构中采用的诸如聚类会话和共享文件系统的传统方法并不能很好地支持水平扩展。

云原生应用架构的另一个标志是将状态外部化到内存数据网格、缓存和持久对象存储，同时保持应用程序实例本身基本上是无状态的。无状态应用程序可以快速创建和销毁，以及附加到外部状态管理器和脱离外部状态管理器，增强我们响应需求变化的能力。当然这也需要外部状态管理器自己来扩展。大多数云基础设施提供商已经认识到这一必要性，并提供了这类服务的健康管理。

1.1.4 移动应用和客户端多样性

2014 年 1 月，美国移动设备占互联网使用量的 55 %。专门针对桌面用户而开发的应用程序的时代已经过去。不过，我们必须假设用户装在口袋里到处散步的是超级计算机。这对我们的应用架构有很大的影响，因为指数级用户可以随时随地与我们的系统进行交互。

以查看银行账户余额为例。这项任务过去是通过拨打银行的呼叫中心，前往 ATM，或者在银行的一个分支机构的向柜员请求完成的。这些客户互动模式在任何时间内，都会对银行为底层软件系统提出新需求产生极大的限制。

迁移到网上银行导致访问量的上升，但并没有从根本上改变交互模式。您仍然必须在计算机终端上与系统进行交互，这仍然显著限制了需求。正如我的同事 Andrew Clay Shafer 经常说的那样，“我们口袋里正带着超级计算机到处游走”，我们开始对这些系统带来很大负载。现在，成千上万的客户可以随时随地与银行系统进行互动。一位银行行政人员表示，在发薪日，客户会每隔几分钟检查一次余额。遗留的银行系统架构根本无法满足这种需求，而云原生的应用程序体系结构却可以。

移动平台的巨大差异也对应用架构提出了要求。客户随时都可能与多个不同供应商生产的设备，运行多个不同的操作系统平台，运行多个版本的相同操作平台以及不同类别的设备（例如手机与平板电脑）进行交互。这不仅对移动应用程序开发人员，还对后端服务的开发人员造成了各种限制。

移动应用程序通常必须与多个传统系统以及云原生应用架构中的多个微服务进行交互。这些服务无法设计成支持客户使用的各种各样移动平台的独特需求。强迫实现这些不同的服务，为移动应用程序开发人员上带来了负担，增加了应用访问延迟和网络访问频率，导致应用响应慢、耗电量高，最终导致用户删除您的应用程序。云原生应用架构还

通过诸如 API 网关之类的设计模式来支持移动优先开发的概念，API 网关将服务聚合负担转移回服务器端。

1.2 云原生架构的定义

现在我们将探索云原生应用架构的几个主要特征，和这些特征是如何解决我们前面提到的使用云原生应用架构的动机。

1.2.1 12 因素应用

12 因素应用是一系列云原生应用架构的模式集合，最初由 Heroku 提出。这些模式可以用来说明什么样的应用才是云原生应用。它们关注速度、安全、通过声明式配置扩展、可横向扩展的无状态 / 无共享进程以及部署环境的整体松耦合。如 Cloud Foundry、Heroku 和 Amazon ElasticBeanstalk 都对部署 12 因素应用进行了专门的优化。

在 12 因素的背景下，应用（或者叫 app）指的是独立可部署单元。组织中经常把一些互相协作的可部署单元称作一个应用。

12 因素应用遵循以下模式：

代码库

每个可部署 app 在版本控制系统中都有一个独立的代码库，可以在不同的环境中部署多个实例。

依赖

App 应该使用适当的工具（如 Maven、Bundler、NPM）来对依赖进行显式的声明，而不该在部署环境中隐式的实现依赖。

配置

配置或其他随发布环境（如部署、staging、生产）而变更的部分应当作为操作系统级的环境变量注入。

后端服务

后端服务，例如数据库、消息代理应视为附加资源，并在所有环境中同等看待。

编译、发布、运行

构建一个可部署的 app 组件并将它与配置绑定，根据这个组件 / 配置的组合来启动一个或者多个进程，这两个阶段是严格分离的。

进程

该 app 执行一个或者多个无状态进程（例如 master/work），它们之间不需要共享任何东西。任何需要的状态都置于后端服务（例如 cache、对象存储等）。

端口绑定

该应用程序是独立的，并通过端口绑定（包括 HTTP）导出任何 / 所有服务。

并发

并发通常通过水平扩展应用程序进程来实现（尽管如果需要的话进程也可以通过内部管理的线程多路复用来实现）。

可任意处置性

通过快速迅速启动和优雅的终止进程，可以最大程度上的实现鲁棒性。这些方面允许快速弹性缩放、部署更改和从崩溃中恢复。

开发 / 生产平等

通过保持开发、staging 和生产环境尽可能的相同来实现持续交付和部署。

日志

不管理日志文件，将日志视为事件流，允许执行环境通过集中式服务收集、聚合、索引和分析事件。

管理进程

行政或管理类任务（如数据库迁移），应该在与 app 长期运行的相同的环境中一次性完成。

这些特性很适合快速部署应用程序，因为它们不需要对将要部署的环境做任何假定。不对环境假设能够允许底层云平台使用简单而一致的机制，轻松实现自动化，快速配置新环境，并部署应用。以这种方式，十二因素应用模式能够帮我们优化应用的部署速度。

这些特性也很好地适用于突发需求，或者低成本地“丢弃”应用程序。应用程序环境本身是 100 %一次性的，因为任何应用程序状态，无论是内存还是持久性，都被提取到后端服务。这允许应用程序以易于自动化的非常简单和弹性的方式进行伸缩。在大多数情况下，底层平台只需将现有环境复制到所需的数目并启动进程。缩容是通过暂停正在运行的进程和删除环境来完成，无需设法地实现备份或以其他方式保存这些环境的状态。就这样，12 因素应用模式帮助我们实现规模优化。

最后，应用程序的可处理性使得底层平台能够非常快速地从故障事件中恢复。

此外，将日志作为事件流处理能够极大程度上的增强应用程序运行时底层行为的可视性。

强制环境之间的等同、配置机制的一致性和后端服务管理使云平台能够为应用程序运行时架构的各个方面提供丰富的可视性。以这种方式，十二因素应用模式能够优化安全性。

1.2.2 微服务

微服务将单体业务系统分解为多个“仅做好一件事”的可独立部署的服务。这件事通常代表某项业务能力，或者最小可提供业务价值的“原子“服务单元。

微服务架构通过以下几种方式为速度、安全、可扩展性赋能：

- 当我们将业务领域分解为可独立部署的有限能力的环境的同时，也将相关的变更周期解耦。只要变更限于单一有限的环境，并且服务继续履行其现有合约，那么这些更改可以独立于与其他业务来进行开展和部署。结果是实现了更频繁和快速的部署，从而实现了持续的价值流动。
- 通过扩展部署组织本身可以加快部署。由于沟通和协调的开销，添加更多的人，往往会使软件构建变得更加苦难。弗雷德·布鲁克斯（Fred Brooks，人月神话作者）很多年前就教导我们，在软件项目的晚期增加更多的人力将会时软件项目更加延期。然而，我们可以通过在有限的环境中构建更多的沙箱，而不是将所有的开发者都放在同一个沙箱中。
- 由于学习业务领域和现有代码的认知负担减少，并建立了与较小团队的关系，因此我们添加到每个沙箱的新开发人员可以更快速地提高并变得更高效。
- 可以加快采用新技术的步伐。大型单体应用架构通常与对技术堆栈的长期保证有关。这些保证的存在是为了减轻采用新技术的风险。采用了错误的技术在单体架构中的代价会更高，因为这些错误可能会影响整个企业架构。如果我们可以在单个整体的范围内采用新技术，将隔离并最大限度地降低风险，就像隔离和最小运行时故障的风险一样。
- 微服务提供独立、高效的服务扩展。单体架构也可以扩展，但要求我们扩展所有组件，而不仅仅是那些负载较重的组件。当且仅当相关联的负载需要它时，微服务才会被缩放。

1.2.3 自服务敏捷架构

使用云原生应用架构的团队通常负责其应用的部署和持续运营。云原生应用的成功采纳者已经为团队提供了自服务平台。

正如我们创建业务能力团队为每个有界的环境构建微服务一样，我们还创建了一个能力小组，负责提供一个部署和运行这些微服务的平台。

这些平台中最大好处是为消费者提供主要的抽象层。通过基础架构即服务（IAAS），我们要求 API 创建虚拟服务器实例、网络和存储，然后应用各种形式的配置管理和自动化，以使我们的应用程序和支持服务能够运行。现在这种允许我们自定义应用和支持服务的平台正在不断涌现。

应用程序代码简单地以预构建的工件（可能是作为持续交付管道的一部分生成的）或

Git 远程的原始源代码的形式“推送”。然后，平台构建应用程序工件，构建应用程序环境，部署应用程序，并启动必要的进程。团队不必考虑他们的代码在哪里运行或如何到达那里，这些对用户都是透明得，因为平台会关注这些。

这样的模型同样适合于后端服务。需要数据库？消息队列或邮件服务器？只需要求平台来配合您的需求。平台现在支持各种 SQL/NoSQL 数据存储、消息队列、搜索引擎、缓存和其他重要的后端服务。这些服务实例然后可以“绑定”到您的应用程序，必要的凭据会自动注入到应用程序的环境中以供其使用。从而消除了大量凌乱而易出错的定制自动化。

这些平台还经常提供广泛的额外操作能力：

- 应用程序实例的自动化和按需扩展
- 应用健康管理
- 请求到或跨应用程序实例间的动态路由和负载均衡
- 日志和指标的聚合

这种工具的组合确保了能力团队能够根据敏捷原则开发和运行服务，从而实现速度，安全性和规模化。

1.2.4 基于 API 的协作

在云原生应用架构中，服务之间的唯一互动模式是通过已发布和版本化的 API。这些 API 通常是具有 JSON 序列化的 HTTP REST 风格，但也可以是其他协议和序列化格式。

只要有需要，在不会破坏任何现有的 API 协议的前提下，团队就可以部署新的功能，而不需要与其他团队进行同步。自助服务基础设施平台的主要交互模式也是通过 API，就像其他业务服务一样。供给、缩放和维护应用程序基础设施的方式不是通过提交单据，而是将这些请求提交给提供该服务的 API。

通过消费者驱动的协议，可以在服务间交互的双方验证协议的合规性。服务消费者不能访问其依赖关系的私有实现细节，或者直接访问其依赖关系的数据存储。实际上，只允许有一个服务能够直接访问任何数据存储。这种强制解耦直接支持云原生的速度目标。

1.2.5 抗脆弱性

Nassim Taleb 在他的 Antifragile（Random House）一书中介绍了抗脆弱性的概念。如果脆弱性是受到压力源的弱化或破坏的质量系统，那么与之相反呢？许多人会以稳健性或弹性作出回应 —— 在遭受压力时不会被破坏或变弱。然而，Taleb 引入了与脆弱性相反的抗脆弱性概念，或者在受到压力源时变得更强的质量系统。什么系统会这样工作？联想下人体免疫系统，当接触病原体时，其免疫力变强，隔离时较弱。我们可以像这样建立架构吗？云原生架构的采用者们已经设法构建它们了。Netflix Simian Army

项目就是个例子，其中著名的子模块“混沌猴”，它将随机故障注入到生产组件中，目的是识别和消除架构中的缺陷。通过明确地寻求应用架构中的弱点，注入故障并强制进行修复，架构自然会随着时间的推移而更大程度地收敛。

1.3 本章小结

本章中，我们讨论了希望通过软件赋予我们业务的能力并迁移到云原生应用架构的动机：

速度

比我们的竞争对手更快速得创新、试验并传递价值。

安全

在保持稳定性、可用性和持久性的同时，具有快速行动的能力。

扩展

根据需求变化弹性扩展。

移动性

客户可以随时随地通过任何设备无缝的跟我们交互。

我们还研究了云原生应用架构的独特特征，以及如何赋予我们这些能力：

12 因素应用

一套优化应用设计速度，安全性和规模的模式。

微服务

一种架构模式，可帮助我们将部署单位与业务能力保持一致，使每个业务能够独立和自主地移动，这样一来也更快更安全。

自服务敏捷基础设施

云平台使开发团队能够在应用和服务抽象层面上运行，提供基础架构级速度，安全性和扩展性。

基于 API 的协作

将服务间交互定义为可自动验证协议的架构模式，通过简化的集成工作实现速度和安全性。

抗脆弱性

随着速度和规模扩大，系统的压力随之增加，系统的响应能力和安全性也随之提高。

在下一章中，我们将探讨大多数企业为采用云原生应用架构而需要做出哪些改变。

第 2 章

在变革中前行

> 从客户给我们下达订单开始，一直到我们收到现金为止，我们一直都关注时间线。而且我们正在通过删除非附加值的废物来减少这个时间表。
>
> —— Taichi Ohno

Taichi Ohno 被公认为精益制造之父。虽然精益制造的实践无法完全适用于软件开发领域，但它们的原则是一致的。这些原则可以指导我们很好地寻求典型的企业 IT 组织采用云原生应用架构所需的变革，并且接受作为这一转变所带来的部分的文化和组织转型。

2.1 文化变革

企业 IT 采用云原生架构所需的变革根本不是技术性的，而是企业文化和组织的变革，围绕消除造成浪费的结构、流程和活动。在本节中，我们将研究必要的文化转变。

2.1.1 从信息孤岛到 DevOps

企业 IT 通常被组织成以下许多孤岛：

- 软件开发
- 质量保证
- 数据库管理
- 系统管理
- IT 运营
- 发布管理
- 项目管理

创建这些孤岛是为了让那些了解特定领域的人员来管理和指导那些执行该专业领域工作的人员。这些孤岛通常具有不同的管理层次，工具集、沟通风格、词汇表和激励结构。这些差异启发了企业 IT 目标的不同范式，以及如何实现这一目标。

但这里面存在很多矛盾，例如开发和运维分别对软件变更持有的观念就是个经常被提起的例子。开发的任务通常被视为通过开发软件功能为组织提供额外的价值。这些功能本

身就是向 IT 生态系统引入变更。所以开发的使命可以被描述为“交付变化”，而且经常根据有多少次变更来进行激励。

相反，IT 运营的使命可以被描述为“防止变更”。IT 运营通常负责维护 IT 系统所需的可用性、弹性、性能和耐用性。因此，他们经常以维持关键绩效指标（KPI）来进行激励，例如平均故障间隔时间（MTBF）和平均恢复时间（MTTR）。与这些措施相关的主要风险因素之一是在系统中引入任何类型的变更。那么，不是设法将开发期望的变更安全地引入 IT 生态系统，而是通过将流程放在一起，使变更变得痛苦，从而降低了变化率。

这些不同的范式显然导致了许多额外的工作。项目工作中的协作、沟通和简单的交接变得乏味和痛苦，最糟糕的是导致绝对混乱（甚至是危险的）。企业 IT 通常通过创建基于单据的系统和委员会会议驱动的复杂流程来尝试“修复”这种情况。企业 IT 价值流在所有非增值浪费下步履瞒珊。

像这样的环境与云原生的速度思想背道而驰。专业的信息孤岛和流程往往是由创造安全环境的愿望所驱动。然而，他们通常提供很少的附加安全性，在某些情况下，会使事情变得更糟！

在其核心上，DevOps 代表着这样一种思想，即将这些信息孤岛构建成共享的工具集、词汇表和沟通结构，以服务于专注于单一目标的文化：快速、安全得交付价值。然后创建激励结构，强制和奖励领导组织朝着这一目标迈进的行为。官僚主义和流程被信任和责任所取代。

在这个新的世界中，开发和 IT 运营部门向共同的直接领导者汇报，并进行合作，寻找能够持续提供价值并获得期望的可用性、弹性、性能和耐久性水平的实践。今天，这些对背景敏感的做法越来越多地包括采用云原生应用架构，提供完成组织的新的共同目标所需的技术支持。

2.1.2 从间断均衡到持续交付

企业经常采用敏捷流程，如 Scrum，但是只能作为开发团队内部的本地优化。

在这个行业中，我们实际上已经成功地将个别开发团队转变为更灵活的工作方式。我们可以这样开始项目，撰写用户故事，并执行敏捷开发的所有例程，如迭代计划会议，日常站会，回顾和客户展示 demo。我们中的冒险者甚至可能会冒险进行工程实践，如结对编程和测试驱动开发。持续集成，这在以前是一个相当激进的概念，现在已经成为企业软件词典的标准组成部分。事实上，我已经是几个企业软件团队中的一部分，并建立了高度优化的“故事到演示”周期，每个开发迭代的结果在客户演示期间被热烈接受。

但是，这些团队会遇到可怕的问题：我们什么时候可以在生产环境中看到这些功能？

这个问题我们很难回答，因为它迫使我们考虑自己无法控制的力量：

- 我们需要多长时间才能浏览独立的质量保证流程？

- 我们什么时候可以加入生产发布的行列中?
- 我们可以让 IT 运营及时为我们提供生产环境吗?

在这一点上，我们意识到自己已经陷入了戴维・韦斯特哈斯（Dave Westhas）所说的 scrum 瀑布中了。我们的团队已经开始接受敏捷原则，但我们的组织却没有。所以，不是每次迭代产生一次生产部署（这是敏捷宣言的原始出发点），代码实际上是批量参与一个更传统的下游发布周期。

这种操作风格产生直接的后果。我们不是每次迭代都将价值交付给客户，并将有价值的反馈回到开发团队，我们继续保持“间断均衡”的交付方式。间断均衡实际上丧失了敏捷交付的两个主要优点：

- 客户可能需要几周的时间才能看到软件带来的新价值。他们认为，这种新的敏捷工作方式只是“像往常一样”，不会增强对开发团队的信任。因为他们没有看到可靠的交付节奏，他们回到了以前的套路将尽可能多的要求尽可能多地堆放到发布版上。为什么？因为他们对软件能够很快发布没有信心，他们希望尽可能多的价值被包括在最终交付时。
- 开发团队可能会好几周都没有得到真正的反馈。虽然演示很棒，但任何经验丰富的开发人员都知道，只有真实用户参与到软件之中才能获得最佳反馈。这些反馈能够帮助软件修正，使团队去做正确的事情。反馈推迟后，错误的可能性只能增加，并带来昂贵的返工。

获得云原生应用架构的好处需要我们转变为持续交付。我们拥抱端到端拥抱价值的原则，而不是 Water Scrum Fall 组织驱动的间断平衡。设想这样一个生命周期的模型是由 Mary 和 Tom Poppendieck 在《实施精益软件开发（Addison-Wesley)》一书中描述的“概念到现金”的想法中提出来的。这种方法考虑了所有必要的活动，将业务想法从概念传递到创造利润的角度，并构建可以使人们和过程达到最佳目标的价值流。

我们技术上支持这种使用连续交付的工程实践的方法，每次迭代（实际上是次每个源代码提交!）都被证明可以以自动化的方式部署。我们构建部署流水线，可自动执行每次测试，如果该测试失败，将会阻止生产部署。唯一剩下的决定是商业决策：现在部署可用的新功能有很好的业务意义吗？我们已经知道它已经如广告中的方式工作，但是我们要现在就把它们交给客户吗？因为部署管道是完全自动化的，所以企业能够通过点击按钮来决定是否采取行动。

2.1.3 从集中治理到分散自治

Waterscrumfall 文化中的一部分已经被特别提及，因为它已经被视为云原生架构采纳的一个关键。

企业通常采用围绕应用架构和数据管理的集中治理结构，负责维护指导方针和标准的委

员会，以及批准个人设计和变更。集中治理旨在帮助解决以下几个问题：

- 可以防止技术栈的大范围不一致，降低组织的整体维护负担。
- 可以防止架构选型中的大范围不一致，从而形成组织的应用程序开发的共同观点。
- 整个组织可以一致地处理跨部门关切，例如合规性。
- 数据所有权可由具有全局视野的人来决定。

之所以创造这些结构，是因为我们相信它们将有助于提高质量、降低成本或两者兼而有之。然而，这些结构很少能够帮助我们提高质量节约成本，并且进一步妨碍了云原生应用架构寻求的交付速度。正如单体应用架构导致了限制技术创新速度的瓶颈一样，单一的治理结构同样如此。架构委员会经常只会定期召集，并且经常需要很长的等待时才能发挥工作。即使是很小的数据模型的变化 —— 可能在几分钟或几个小时内完成的更改，即将被委员会批准的变更 —— 将会把时间浪费在一个不断增长的待办事项中。

采用云原生应用架构时通常都会与分散式治理结合起来。建立云原生应用的团队拥有他们负责交付的能力的所有方面。他们拥有和管理数据、技术栈、应用架构、每个组件设计和 API 协议并将它们交付给组织的其余部分。如果需要对某事作出决策，则由团队自主制定和执行。

团队个体的分散自治和自主性是通过最小化、轻量级的结构进行平衡的，这些结构在可独立开发和部署的服务之间使用集成模式（例如，他们更喜欢 HTTP REST JSON API 而不是不同风格的 RPC）来实现。这些结构通常会在底层解决交叉问题，如容错。激励团队自己设法解决这些问题，然后自发组织与其他团队一起建立共同的模式和框架。随着整个组织中的最优解决方案出现，该解决方案的所有权通常被转移到云框架 / 工具团队，这可能嵌入到平台运营团队中也可能不会。当组织正在围绕对架构共识进行改革时，云框架 / 工具团队通常也将开创解决方案。

2.2 组织变革

在本节中，我们将探讨采用云原生应用架构的组织在创建团队时需要进行的变革。这个重组背后的理论是著名的康威定律。我们的解决方案是在长周期的产品开发中，创建一个包含了各方面专业员工的团队，而不是将他们分离在单一的团队中，例如测试人员。

2.2.1 业务能力团队

> 设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构。
>
> —— Melvyn Conway

我们已经讨论了“从孤岛到 DevOps”将 IT 组织成专门的信息孤岛的做法。我们很自然

地创造了这些孤岛，并把个体放到与这些孤岛一致的团队中。但是当我们需要构建一个新的软件的时候会怎么样?

一个很常见的做法是成立一个项目团队。该团队向项目经理汇报，然后项目经理与各种孤岛合作，为项目所需的各个专业领域寻求“资源”。正如康威定律所言，这些团队将很自然地在系统中构筑起各种孤岛，我们最终得到的是联合各种孤岛相对应的孤立模块的架构：

- 数据访问层
- 服务层
- Web MVC 层
- 消息层
- 等等

这些层次中的每一层都跨越了多个业务能力领域，使得在其之上的创新和部署独立于其他业务能力的新功能变的非常困难。

寻求迁移到将业务能力分离的微服务等云原生架构的公司经常采用 Thoughtworks 称之为的“逆康威定律”。他们没有建立一个与其组织结构图相匹配的架构，而是决定了他们想要的架构，并重组组织以匹配该架构。如果你这样做的话，根据康威定律，您所期望的架构终将出现。

因此，作为转向 DevOps 文化的一部分，我们组织了跨职能、业务能力的团队，开发的是产品而不再是项目。开发产品需要长期的付出，直到它们不再为企业提供价值为止。（直到你的代码不再运行在生产上为止!）构建、测试、交付和运营提供业务能力的服务所需的所有角色都存在于一个团队中，该团队不会向组织的其他部分交接代码。这些团队通常被组织为“双比萨队”，意思是如果不能用两个比萨饼喂饱，那就意味着团队规模太大了。

那么剩下的就是确定要创建的团队。如果我们遵循逆康威定律，我们将从组织的领域模型开始，并寻求可以封装在有限环境中的业务能力。一旦我们确定了这些能力，我们就可以创建为这些业务能力的整个生命周期负责的团队。业务能力团队掌握其应用程序从开发到运营的整个生命周期。

2.2.2 平台运营团队

业务能力团队需要依赖于我们前面提到的”自助敏捷基础架构“。

事实上，我们可以这样来描述一种特殊的业务能力 —— 开发、部署和运营业务的能力。这种能力应该是平台运营团队所具有的。

平台运营团队运营自助敏捷基础架构平台，并交付给业务能力团队使用。该团队通常包括传统的系统、网络和存储管理员角色。如果公司正在运营云平台，该团队也将拥有管

理数据中心的团队或与他们紧密合作，并了解提供基础架构平台所需的硬件能力。

IT 运营传统上通过各种基于单据的系统与客户进行互动。由于基于平台操作流来运行自助服务平台，因此必须提供不同形式的交互方式。正如业务能力团队之间通过定义好的 API 协议相互协作一样，平台运营团队也为该平台提供了 API 协议。业务能力团队不再需要排队申请应用环境和数据服务，而是采用更精简的方式构建按需申请环境和服务的自动化发布管道。

2.3 技术变革

现在我们将一些问题转移到了云中的 DevOps 平台。

2.3.1 分解单体应用

传统的 n 层单体式应用部署到云中后很难维护，因为它们经常对云基础设施提供的部署环境做出不可靠的假设，这些假设云很难提供。例如以下要求：

- 可访问已挂载的共享文件系统
- P2P 应用服务器集群
- 共享库
- 配置文件位于常用的配置文件目录

大多数这些假设都出于这样的事实：单体应用通常都部署在长期运行的基础设施中，并与其紧密结合。不幸的是，单体应用并不太适合弹性和短暂（非长期支持）生命周期的基础设施。

但是即使我们可以构建一个不需要这些假设的单体应用，我们依然有一些问题需要解决：

- 单体式应用的变更周期耦合，使独立业务能力无法按需部署，阻碍创新速度。
- 嵌入到单体应用中的服务不能独立于其他服务进行扩展，因此负载更难于优化。
- 新加入组织的开发人员必须适应新的团队，经常学习新的业务领域，并且一次就熟悉一个非常大的代码库。这样会增加 3-6 个月的适应时间，才能实现真正的生产力。
- 尝试通过堆积开发人员来扩大开发组织，增加了昂贵的沟通和协调成本。
- 技术栈需要长期承诺。引进新技术太过冒险，可能会对整体产生不利影响。

细心的读者会注意到，该列表正好与“微服务”的列表相反。将组织分解为业务能力团队还要求我们将应用程序分解成微服务。只有这样，我们才能从云计算基础架构中获得最大的收益。

2.3.2 分解数据

仅仅将单体应用分解为微服务还是远远不够的。数据模型必须要解耦。如果业务能力团队被认为是自主的，却被迫通过单一的数据存储进行协作，那么单体应用对创新的阻碍将依然存在。

事实上，产品架构必须从数据开始的说法是有争议的。由 Eric Evans（Addison-Wesley）在领域驱动设计（DDD）中提出的原理认为，我们的成功在很大程度上取决于领域模型的质量（以及支持它的普遍存在的语言）。要使领域模型有效，还必须在内部一致 —— 我们不应该在同一模型内的一致定义中找到重复定义的术语或概念。

创建不具有这种不一致的联合领域模型是非常困难和昂贵的（可以说是不可能的）。Evans 将业务的整体领域模型的内部一致性子集称为有界上下文。

最近与航空公司客户合作时，我们讨论了他们业务的核心概念，自然是“航空公司预订”的话题。该集团可以在其预定业务中划分十七种不同的逻辑定义，几乎不能将它们调和为一个。相反，每个定义的所有细微差别都被仔细地描绘成一个个单一的概念，这将成为组织的巨大瓶颈。

有界上下文允许你在整个组织中保持单一概念的不一致定义，只要它们在有界上下文中一致地定义。

因此，我们首先需要确定可以在内部保持一致的领域模型的细分。我们在这些细分上画出固定的边界，划分出有界上下文。然后，我们可以将业务能力团队与这些环境相匹配，这些团队将构建提供这些功能的微服务。

微服务提供了一个有用的定义，用于定义 12 因素应用程序应该是什么。12 因素主要是技术规范，而微服务主要是业务规范。通过定义有界上下文，为它们分配一组业务能力，委托业务能力团队对这些业务能力负责，并建立 12 因素应用程序。在这些应用程序可以独立部署的情况下，为业务能力团队的运维提供了一组有用的技术工具。

我们将有界上下文与每个服务模式的数据库结合，每个微服务封装、管理和保护自己的领域模型和持久存储。在每个服务模式的数据库中，只允许一个应用程序服务访问逻辑数据存储，逻辑数据存储可能是以多租户集群中的单个 schema 或专用物理数据库中存在。对这些概念的任何外部访问都是通过一个明确定义的业务协议来实现的，该协议的实现方式为 API（通常是 REST，但可能是任何协议）。

这种分解允许应用拥有多语言支持的持久性，或者基于数据形态和读写访问模式选择不同的数据存储。然而，数据必须经常通过事件驱动技术重新组合，以便请求交叉上下文。诸如命令查询责任隔离（CQRS）和事件溯源（Event Sourcing）之类的技术通常在跨上下文同步类似概念时很有帮助，这超出了本文的范围。

2.3.3 容器化

容器镜像（例如通过 LXC、Docker 或 Rocket 项目准备的镜像）正在迅速成为云原生应用架构的部署单元。然后通过诸如 Kubernetes、Marathon 或 Lattice 等各种调度解决方案实例化这样的容器镜像。亚马逊和 Google 等公有云供应商也提供一流的解决方案，用于容器化调度和部署。容器利用现代的 Linux 内核原语，如控制组（cgroups）和命名空间来提供类似的资源分配和隔离功能，这些功能与虚拟机提供的功能相比，具有更少的开销和更强的可移植性。应用程序开发人员将需要将应用程序包装成容器镜像，以充分利用现代云基础架构的功能。

2.3.4 从管弦乐编排到舞蹈编舞

不仅仅服务交付、数据建模和治理必须分散化，服务集成也是如此。企业服务集成传统上是通过企业服务总线（ESB）实现的。ESB 成为管理服务之间交互的所有路由、转换、策略、安全性和其他决策的所有者。我们将其称之为编排，类似于导演，它决定了乐团演出期间演奏音乐的流程。ESB 和编排可以产生非常简单和令人愉快的架构图，但它们的简单性仅仅是表面性的。在 ESB 中隐藏的是复杂的网络。管理这种复杂性成为全职工作，这成为应用开发团队的持续瓶颈。正如我们在联合数据模型所看到的，像 ESB 这样的联合集成解决方案成为阻碍幅度的巨大难题。

诸如微服务这样的云原生架构更倾向于舞蹈，它们类似于芭蕾舞中的舞者。它们将心智放置在端点上，类似于 Unix 架构中的虚拟管道和智能过滤器，而不是放在集成机制中。当舞台上的情况与原来的计划有所不同时，没有导演告诉舞者该怎么做。相反，他们会自适应。同样，服务通过客户端负载均衡和断路器等模式，适应环境中不断变化的各种情况。

虽然架构图看起来像一个庞杂的网络，但它们的复杂性并不比传统的 SOA 大。编排简单地承认并暴露了系统原有的复杂性。再次，这种转变是为了支持从云原生架构中寻求速度所需的自治。团队能够适应不断变化的环境，而无需承担与其他团队协调的开销，并避免了在集中管理的 ESB 中协调变更所造成的开销。

2.4 本章小结

本章中，我们探讨了大多数企业采用云原生应用架构所需要做出的变革。从宏观总体上看是权力下放和自治：

DevOps

技能集中化转变为跨职能团队。

持续交付

发行时间表和流程的权力下放。

自治

决策权力下放。

我们将这种权力下放编成两个主要的团队结构：

业务能力团队

自主决定设计、流程和发布时间表的跨职能团队。

平台运营团队

为跨职能团队提供他们所需要运行平台。

而在技术上，我们也分散自治：

单体应用到微服务

将个人业务能力的控制分配给单个自主服务。

有界上下文

将业务领域模型的内部一致子集的控制分配到微服务。

容器化

将对应用包装的控制分配给业务能力团队。

编排

将服务集成控制分配给服务端点。

所有这些变化造就了无数的自治单元，辅助我们以期望的创新速度安全前行。

在最后一章中，我们将通过一组操作手册，深入研究迁移到云原生应用架构的技术细节。

第 3 章

迁移指南

现在我们已经定义了云原生应用架构，并简要介绍了企业在采用它们时必须考虑做出的变化，现在是深入研究技术细节的时候了。对每个技术细节的深入讲解已经超出了本报告的范围。本章中仅是对采用云原生应用架构后，需要做的特定工作和采用的模式的一系列简短的介绍，文中还给出了一些进一步深入了解这些方法的链接。

3.1 分解架构

在和客户讨论分解数据、服务和团队后，客户经常向我提出这样的问题，“太棒了！但是我们要怎样实现呢?”这是个好问题。如何拆分已有的单体应用并把他们迁移上云呢?

事实证明，我已经看到了很多成功的例子，使用增量迁移这种相当可复制的模式，我现在向我所有的客户推荐这种模式。SoundCloud 和 Karma 就是公开的例子。

本节中，我们将讲解如何一步步地将单体服务分解并将它们迁移到云上。

3.1.1 新功能使用微服务形式

您可能感到很惊奇，第一步不是分解单体应用。我们假设您依然要在单体应用中构建服务。事实上，如果您没有任何新的功能来构建，那么您甚至不应该考虑这个分解。（鉴于我们的主要动机是速度，您如何维持原状还能获取速度呢?）

> 团队决定，处理架构变化的最佳方法不是立即分解 Mothership 架构，而是不添加任何新的东西。我们所有的新功能以微服务形式构建...
>
> —— Phil Calcado, SoundCloud

所以不要继续再向单体应用中增加代码，将所有的新功能以微服务的形式构建。这是第一步就要考虑好的，因为从头开始构架一个服务比分解一个单体应用并提出服务出来容易和快速的多。

然而有一点不可避免，就是新构建的微服务需要与已有的单体应用通信才能完成工作，这个问题怎么解决?

3.1.2 隔离层

> 因为我们大部分的业务逻辑都是基于 Rails 的单体应用，所以我们的微服务基本也要跟它们通信。
>
> —— Phil Calcado, SoundCloud

Eric Evans（Addison-Wesley）的领域驱动设计（DDD）讨论了隔离层的思想。其目的是允许两个系统的集成，而不允许一个系统的领域模型破坏另一个系统的领域模型。当您将新功能集成到微服务中时，不希望这些新服务与整体的紧密结合，让他们深入了解整体的内部结构。隔离层是创建 API 协议的一种方式，使得整体架构看起来像其他微服务。

Evans 将隔离层的实施划分为三个子模块，前两个代表着经典设计模式。

（来自 Gamma 等人，Design Patterns：Elements of Reusable Object-Oriented Soware [Addison Wesley]）：

表现层

表现层的目的是为了简化与单体应用接口集成的过程。单体应用设计之初很可能没有考虑这个集成，因此我们引入了表现层来解决这个问题。它没有改变单体应用的模型，这很重要，注意不要将转换和集成问题耦合到一起。

适配器

我们用适配器来定义 service，用来提供我们需要的新功能。它知道如何获取系统请求并使用协议将请求发送给单体应用的表层。

转换器

转换器的职责是在单体应用与新的微服务之间进行请求和响应的领域模型转换。

这三个松耦合的组件解决了以下三个问题：

1. 系统集成
2. 协议转换
3. 模型转换

剩下的是通信链路的位置。在 DDD 中，Evans 讨论了两种选择。当您无法访问或更改遗留系统时，第一，将系统的表现层设置为主要功能。我们的重点在于我们控制的整体，所以我们将倾向于 Evans 的第二个建议，适配器到表现层。使用这种替代方法，我们将表现层构筑到单体中，允许在适配器和表现层之间进行通信，因为在为此专门构建的组件之间创建连接更容易。

最后，要注意隔离层可以促进双向通信。正如我们新的微服务可能需要与整体进行通信以完成工作一样，反之亦然，特别是当我们进入下一阶段时。

> 在架构调整后，我们的团队可以在更加灵活的环境中自由构建新功能和增强功能。然而，一个重要的问题仍然存在：我们如何从名为 Mothership 的单体 Rails 应用程序中提取功能？
>
> —— Pilil Calcado，SoundCloud

我从 Martin Fowler 的题为“扼杀应用”的文章中借用了“扼杀巨石”的想法。在这篇文章中，Fowler 解释了逐渐创造“围绕旧系统边缘的新系统，让它几年来慢慢增长，直到旧系统被扼杀”的想法。这种情况同样适用于我们。通过提取的微服务和其他隔离层的组合，我们将围绕现有单体的边缘构建一个新的云原生系统。

两个标准帮助我们选择要提取哪些组件：

1. SoundCloud 指出了第一个标准：识别单体中的有界上下文。如果您回想起我们之前讨论有限上下文，它需要一个内部一致的领域模型。我们的单体领域模型极有可能不是内部一致的。现在是开始识别子模型的时候了，里面有我们要提取的候选者。
2. 第二个标准是优先考虑的：在众多的候选者中我们应该首先提取哪一个呢？我们可以回顾一下迁移到云原生架构的第一个原因：创新速度。什么候选微服务将最受益于创新速度？我们显然希望选择那些正在改变我们当前业务需求的服务。看看单体应用的积压。确定需要更改的代码的区域，以便提交更改的要求，然后在进行所需更改之前提取适当的有界上下文。

3.1.4 潜在的结束状态

我们怎么知道何时结束？下面有两个基本的结束状态：

1. 单体架构已经被完全扼杀。所有的有界上下文都被提取为微服务。最后一步是确定消除不再需要隔离层的机会。
2. 单体架构被扼杀到了这样一个点：额外服务提取的成本超过必要开发努力的回报。单体的一些部分可能相当稳定 —— 它们几年来都没有改变，还是一直都在运行得好好的。迁移这些部分可能没有太大的价值，维持必要的隔离层与其集成的成本足够低，我们可以长期负担。

3.2 使用分布式系统

当我们开始构建由微服务组成的分布式系统时，我们还会遇到在开发单体应用时通常不会遇到的非功能性要求。有时，使用物理定律就可以解决这些问题，例如一致性、延迟和网络分区问题。然而，脆弱性和易控性的问题通常可以使用相当通用的模式来解决。在本节中，我们将介绍帮助我们解决这些问题的方法。

这些方法来自于 Spring Cloud 项目和 Netflix OSS 系列项目的组合。

3.2.1 版本化和分布式配置

在“12 因素应用“中我们讨论过通过操作系统级环境变量为应用注入对应的配置，强调了这种配置管理方式的重要性。这种方式特别适合简单的系统，但是，当系统扩大后，有时我们还需要附加的配置能力：

- 为调试一个生产上的问题而变更运行的应用程序日志级别
- 更改 message broker 中接收消息的线程数
- 报告所有对生产系统配置所做的更改以支持审计监管
- 运行中的应用切换功能开关
- 保护配置中的机密信息（如密码）

为了支持这些特性，我们需要配置具有以下特性的配置管理方法：

- 版本控制
- 可审计
- 加密
- 在线刷新

Spring Cloud 项目中包含的一个可提供这些功能的配置服务器。此配置服务器通过 Git 本地仓库支持的 REST API 呈现了应用程序及应用程序配置文件（例如，可用开 / 关切换的一组配置作为一组，如“deployment”和“staging”配置）（图 3 -1）。

例 3-1 是示例配置服务器的默认配置文件：

1. 该配置中指定了后端 Git 仓库中的 `application.yml` 文件。
2. greeting 当前被设置为 ohai。

例 3-1 中的配置是自动生成的，无需手动编码。我们可以看到，通过检查它的 /env 端点（例 3-2），greeting 的值被分发到 Spring 应用中。

1. 该应用接收到来自配置服务器的 greeting 的值：ohai。

现在我们就可以无需重启客户端应用就可以更新 greeting 的值。该功能由 Spring Cloud 项目中的一个名为 Spring Cloud Bus 的组件提供。该项目将分布式系统的节点与轻量级消息代理进行链接，然后可以用于广播状态更改，如我们所需的配置更改（图

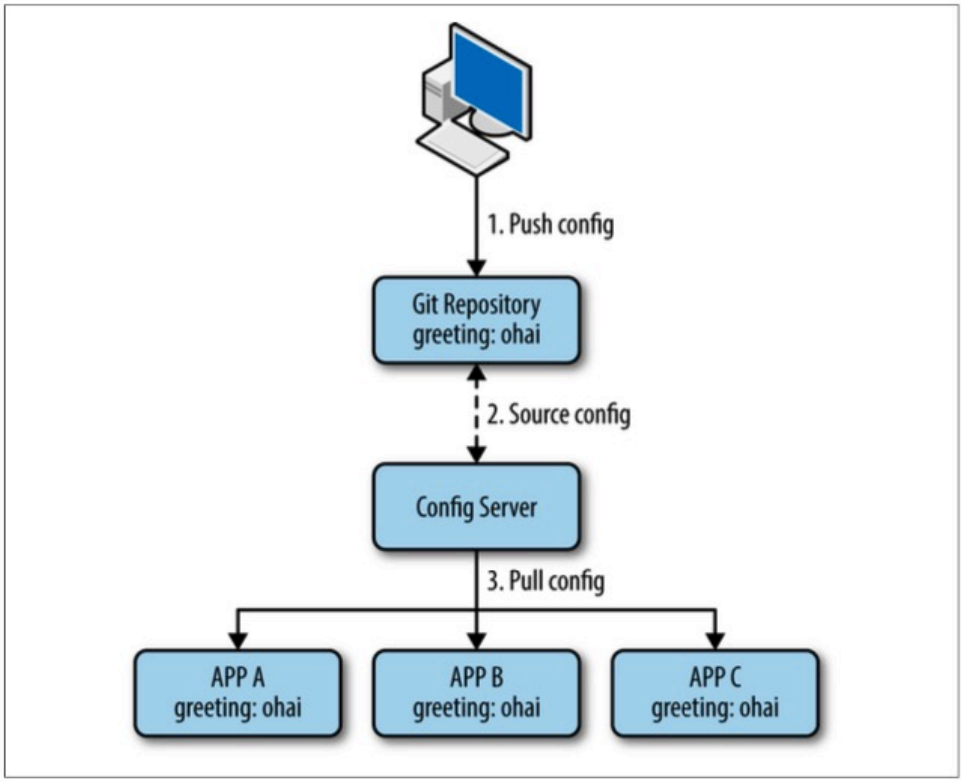

Figure 3-1. The Spring Cloud Config Server

图 3-1: Spring Cloud Config Server

Example 3-1. Default application profile configuration for a sample Config Server

```
{
    "label": "",
    "name": "default",
    "propertySources": [
      {
          "name": "https://github.com/mstine/config-repo.git/applica
tion.yml", ❶
        "source": {
          "greeting": "ohai" ❷
      }
    }
  ]
}
```

图 3-2: 例 3-1

Example 3-2. Environment for a Config Server client

```
"configService:https://github.com/mstine/config-repo.git/applica
tion.yml": {
  "greeting": "ohai" ❶
},
```

图 3-3: 例 3-2

3-2)。该项目将分布式系统的节点与轻量级消息代理进行链接，然后可以用于广播状态更改，如我们所需的配置更改（图 3-2)。

只需通过对参与总线的任何应用程序的 /bus/refresh 端点执行 HTTP POST（这显然应该进行适当的安全性保护)，指示总线上的所有应用程序使用配置服务器中的最新的可用值刷新其配置。

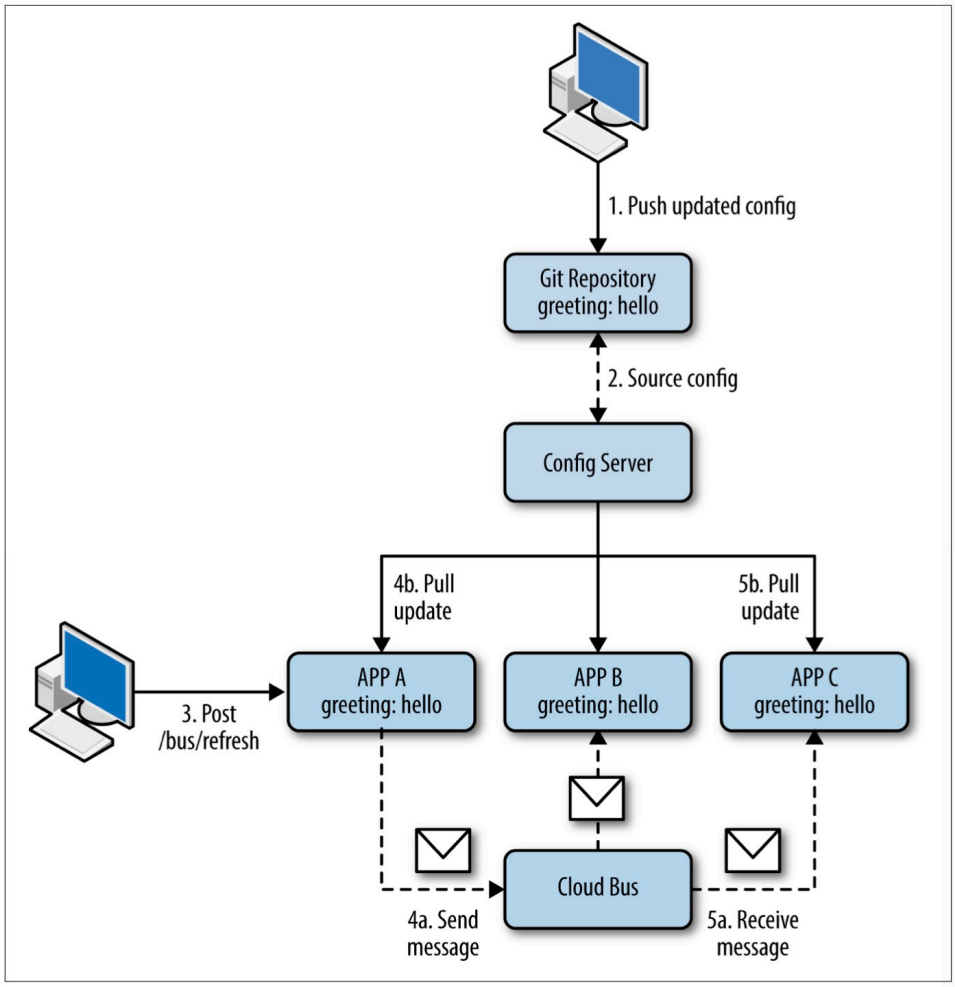

Figure 3-2. The Spring Cloud Bus

图 3-4

3.2.2 服务注册发现

当我们创建分布式系统时，代码的依赖不再是一个方法调用。相反，消费它们必须通过网络调用。我们该如何布线，才能使组合系统中的所有微服务彼此通信？

云中的（图 3-3）的同样架构模式是有一个前端（应用程序）和后端（业务）服务。后端服务往往不能直接从互联网访问，而是通过前端服务访问。服务注册提供的所有服务的列表，使它们可以通过一个客户端库到达前端服务（路由和负载均衡），客户端库执行负载均衡和路由到后端服务。

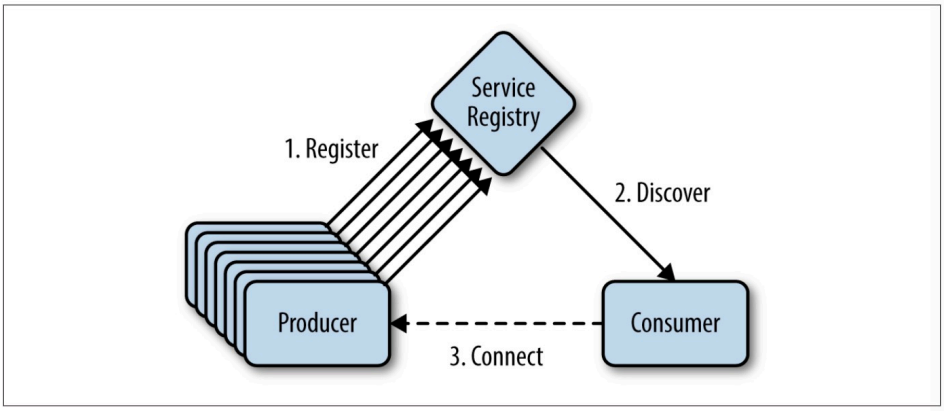

Figure 3-3. Service registration and discovery

图 3-5

在使用服务定位器和依赖注入模式的各种形式之前，我们已经解决了这个问题，面向服务的架构长期以来一直使用各种形式的服务注册表。我们将采用类似的解决方案，利用 Eureka，这是一个 Netflix OSS 项目，可用于定位服务，以实现中间层服务的负载平衡和故障转移。为了使用 Netflix OSS 服务，Spring Cloud Netflix 项目提供了基于注释的配置模型，这大大简化了开发人员在开发 Spring 应用程序时对 Eureka 的心力耗费。

在例 3-3 中，只需简单得在代码中添加 `@EnableDiscoveryClient` 注释，应用程序就可以进行服务注册和发现。

1. `@EnableDiscoveryClient` 开启应用程序的服务注册发现。

该应用程序就能够通过利用 DiscoveryClient 与它的依赖组件通信。例 3-4 是应用程序查找名为 PRODUCER 的注册服务的一个实例，获得其 URL，然后利用 Spring 的 RestTemplate 与之通信。

1. 开启的 DiscoveryClient 通过 Spring 注入。
2. `getNextServerFromEureka` 方法使用 round-robin 算法提供服务实例的位置。

Example 3-3. A Spring Boot application with service registration/discovery enabled

```
@SpringBootApplication
@EnableDiscoveryClient ❶
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }

}
```

图 3-6: 例 3-3

Example 3-4. Using the `DiscoveryClient` to locate a producer service

```
@Autowired
DiscoveryClient discoveryClient; ❶

@RequestMapping("/")
public String consume() {
    InstanceInfo instance = discoveryClient.getNextServerFromEur
eka("PRODUCER", false); ❷

  RestTemplate restTemplate = new RestTemplate();
       ProducerResponse response = restTemplate.getForOb
ject(instance.getHomePageUrl(), ProducerResponse.class);

  return "{\"value\": \"" + response.getValue() + "\"}";
}
```

图 3-7: 例 3-4

3.2.3 路由和负载均衡

基本的 round-robin 负载平衡在许多情况下是有效的，但云环境中的分布式系统通常需要更高级的路由和负载均衡行为。这些通常由各种外部集中式负载均衡解决方案提供。然而，这种解决方案通常不具有足够的信息或上下文，以便在给定的应用程序尝试与其依赖进行通信时做出最佳选择。此外，如果这种外部解决方案故障，这些故障可以跨越整个架构。

云原生的解决方案通常将路由和负载均衡的职责放在客户端。Ribbon Netflix OSS 项目就是其中的一种。（图 3-4）

Ribbon 提供一组丰富的功能集：

- 多种内建的负载均衡规则：

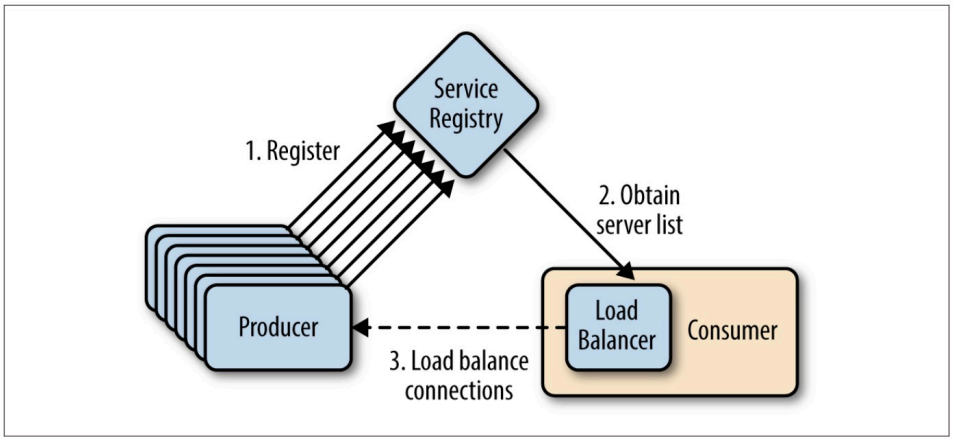

Figure 3-4. Ribbon client-side load balancer

图 3-8

  - Round-robin 轮询负载均衡
  - 平均加权响应时间负载均衡
  - 随机负载均衡
  - 可用性过滤负载均衡（避免跳闸线路和高并发链接数）
  - 自定义负载均衡插件系统
- 与服务发现解决方案的可拔插集成（包括 Eureka）
- 云原生智能，例如可用区亲和性和不健康区规避
- 内建的故障恢复能力

跟 Eureka 一样，Spring Cloud Netflix 项目也大大简化了 Spring 应用程序开发人员使用 Ribbon 的心力耗费。开发人员可以注入一个 LoadBalancerClient 的实例，然后使用它来解析应用程序依赖关系的一个实例（例 3-5），而不是注入 DiscoveryClient 的实例（用于直接从 Eureka 中消费）。

Example 3-5. Using the `LoadBalancerClient` to locate a producer service

```
@Autowired
LoadBalancerClient loadBalancer; ❶
```

图 3-9: 例 3-5-1

1. 由 Spring 注入的 LoadBalancerClient。
2. choose 方法使用当前负载均衡算法提供了服务的一个示例地址。

Spring Cloud Netflix 通过创建可以注入到 Bean 中的 Ribbon-enabled 的 RestTemplate bean 来进一步简化 Ribbon 的配置。RestTemplate 的这个实例被配置为使用 Ribbon（示例 3-6）自动将实例的逻辑服务名称解析为 instanceURI。

```
@RequestMapping("/")
public String consume() {
  ServiceInstance instance = loadBalancer.choose("producer"); ❷
      URI    producerUri    =    URI.create("http://${instance.host}:$
{instance.port}");

  RestTemplate restTemplate = new RestTemplate();
   ProducerResponse  response  =  restTemplate.getForObject(producer
Uri, ProducerResponse.class);

  return "{\"value\": \"" + response.getValue() + "\"}";
}
```

图 3-10: 例 3-5-2

Example 3-6. Using the Ribbon-enabled `RestTemplate`

```
@Autowired
RestTemplate restTemplate; ❶

@RequestMapping("/")
public String consume() {
    ProducerResponse  response  =  restTemplate.getForObject("http://
producer", ProducerResponse.class); ❷
  return "{\"value\": \"" + response.getValue() + "\"}";
}
```

图 3-11: 例 3-6

1. 注入的是 RestTemplate 而不是 LoadBalancerClient。
2. 注入的 RestTemplate 自动将 `http://producer` 解析为实际的服务实例的 URI。

3.2.4 容错

分布式系统比起单体架构来说有更多潜在的故障模式。由于传入系统中的每一个请求都可能触及几十甚至上百个不同的微服务，因此这些依赖中的某些故障实质上是不可避免的。

> 如果不进行容错，30 个依赖，每个都是 99.99% 的正常运行时间，每个月将导致 2 个小时的停机时间（99.99%^30=99.7% 的正常运行时间 = 2 小时以上的停机时间）。
>
> —— Ben Christensen，Netflix 工程师

如何避免这类故障导致级联故障，给我们的系统可用性数据带来负面影响？Mike

Nygard 在他的 Pragmatic Programmers 中提出了几个可以觉得该问题的几个模式，包括：

熔断器

当服务的依赖被确定为不健康时，使用熔断器来阻绝该服务与其依赖的远程调用，就像电路熔断器可以防止电力使用过度，防止房子被烧毁一样。熔断器实现为状态机（图 3-5）。当其处于关闭状态时，服务调用将直接传递给依赖关系。如果任何一个调用失败，则计入这次失败。当故障计数在指定时间内达到指定的阈值时，熔断器进入打开状态。在熔断器为打开状态时，所有调用都会失败。在预定时间段之后，线路转变为“半开”状态。在这种状态下，调用再次尝试远程依赖组件。成功的调用将熔断器转换回关闭状态，而失败的调用将熔断器返回到打开状态。

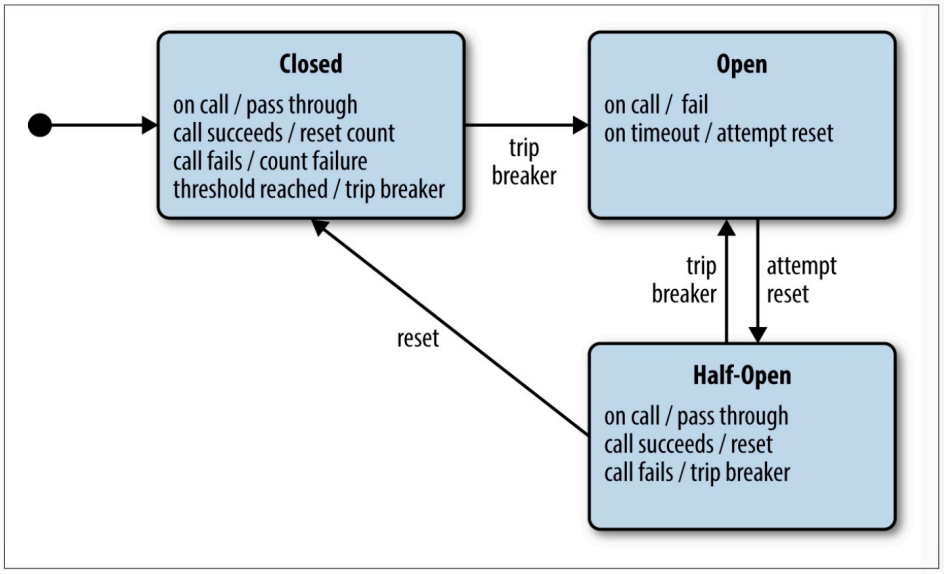

Figure 3-5. A circuit breaker state machine

图 3-12: Figure 3-5

隔板

隔板将服务分区，以便限制错误影响的区域，并防止整个服务由于某个区域中的故障而失败。这些分区就像将船舶划分成多个水密舱室一样，使用隔板将不同的舱室分区。这可以防止当船只受损时造成整艘船沉没（例如，当被鱼雷击中时）。软件系统中可以用许多方式利用隔板。简单地将系统分为微服务是我们的第一道防线。将应用程序进程分区为 Linux 容器，以便使用单个进程无法接管整个计算机。另一个例子是将并行工作划分为不同的线程池。

Netflix 的 Hystrix 应用了这些和更多的模式，并提供了强大的容错功能。为了包含熔断器的代码，Hystrix 允许代码被包含到 HystrixCommand 对象中。

Example 3-7. Using a `HystrixCommand` object

```
public class CommandHelloWorld extends HystrixCommand<String> {

    private final String name;

    public CommandHelloWorld(String name) {
                super(HystrixCommandGroupKey.Factory.asKey("Exam
pleGroup"));
        this.name = name;
    }

    @Override
    protected String run() { ❶
        return "Hello " + name + "!";
    }
}
```

图 3-13: 例 3-7

1. run 方法中封装了熔断器

Spring Cloud Netflix 通过在 Spring Boot 应用程序中添加 `@EnableCircuitBreaker` 注解来启用 Hystrix 运行时组件。然后通过另一组注解，使得基于 Spring 和 Hystrix 的编程与我们先前描述的集成一样简单（例 3-8）。

Example 3-8. Using `@HystrixCommand`

```
@Autowired
RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "getProducerFallback") ❶
public ProducerResponse getProducerResponse() {
   return  restTemplate.getForObject("http://producer",  ProducerRes
ponse.class);
}

public ProducerResponse getProducerFallback() { ❷
  return new ProducerResponse(42);
}
```

图 3-14: 例 3-8

1. 使用 `@HystrixCommand` 注解的方法封装了一个熔断器。
2. 当线路处于打开或者半开状态时，注解中引用的 `getProducerFallback` 方法，提供了一个优雅的回调操作。

Hystrix 相较于其他熔断器来说是独一无二的，因为它还通过在其自己的线程池中操作

每个熔断器来提供隔板。它还收集了许多关于熔断器状态的有用指标，其中包括：

- 流量
- 请求率
- 错误百分比
- 主机报告
- 延迟百分比
- 成功、失败和拒绝

这些 metric 会被发送到事件流中，然后被 Netflix OSS 项目中的另一个叫做 Turbine 的组聚合。每个单独的和聚合后的 metric 流都可以在强大的 Hystrix Dashboard（图 3-6）中以可视化的方式呈现，该页面提供了很好的分布式系统总体健康状态的可视化效果。

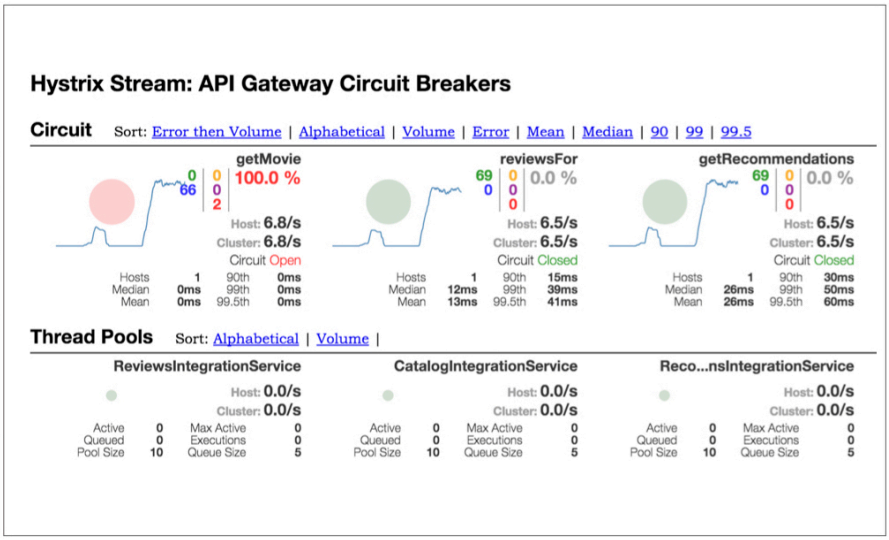

Figure 3-6. Hystrix Dashboard showing three sets of circuit breaker metrics

图 3-15

3.2.5 API 网关 / 边缘服务

在“移动应用和客户端多样性”中我们探讨过服务器端聚合与微服务生态系统。为什么有这个必要？

延迟

移动设备通常运行在比我们家用设备更低速的网络上。即使是在家用或企业网络上，为了满足单个应用屏幕的需求，需要连接数十（或者上百）个微服务，这样的延迟也将变得不可接受。很明显，应用程序需要使用并发的方式来访问这些服务。在服务端一次性捕获和实行这些并发模式，会比在每一个设备平台上做相同的事情，来得更廉价、更不容易出错。

延迟的另一个来源是响应数据的大小。在 Web 服务开发领域，近年来一直趋向于“返回一切可能有用的数据”的做法，这将导致响应的返回数据越来愈大，远远超出了单一的移动设备屏幕的需求。移动设备开发者更倾向于通过仅检索必要的信息而忽略其他不重要的信息，来减少等待时间。

往返通信

即使网速不成问题，与大量的微服务通信依然会给移动应用开发者造成困扰。移动设备的电池消耗主要是因为网络开销造成的。移动应用开发者尽可能通过最少的服务端调用来减少网络的开销，并提供预期的用户体验。

设备多样性

移动设备生态系统中设备多样性是十分巨大的。企业必须应对不断增长的客户群体差异，包括如下这些：

- 制造商
- 设备类型
- 形式因素
- 设备尺寸
- 编程语言
- 操作系统
- 运行时环境
- 并发模型
- 支持的网络协议

这种多样性甚至扩大到超出了移动设备生态系统，开发者目前可能还会关注家用消费电子设备不断增长的生态系统，包括智能电视和机顶盒。

API 网关模式（图 3-7）旨在将客户端的这些需求负担从设备开发者转移到服务器端。API 网关仅仅是一类特殊的满足单个客户端应用程序的微服务（如特定的 iPhone App），并为其提供一个到后端的入口。每个请求同时访问数十（或数百）个微服务，汇总响应并转化，以满足客户应用的需求。在必要时，它们还进行协议转换（例如，HTTP 到 AMQP）。

API 网关可以使用任何支持 web 编程和并发模式的语言、运行时、框架，和能够目标微服务进行通信的协议来实现。热门的选择包括 Node.js（由于其反应式编程模型）和 Go 编程语言（由于其简单的并发模型）。

在这次讨论中，我们将坚持使用 Java，并给出一个 RxJava 例子，一个 Netflix 开发的 Reactive Extensions 的 JVM 实现例子。如果仅使用 Java 语言所提供的原生方法来组成多个工作或数据流是一个很大的挑战，而 RxJava 是一种致力于缓解这种复杂性的技术（还包括 Reactor）技术之一。

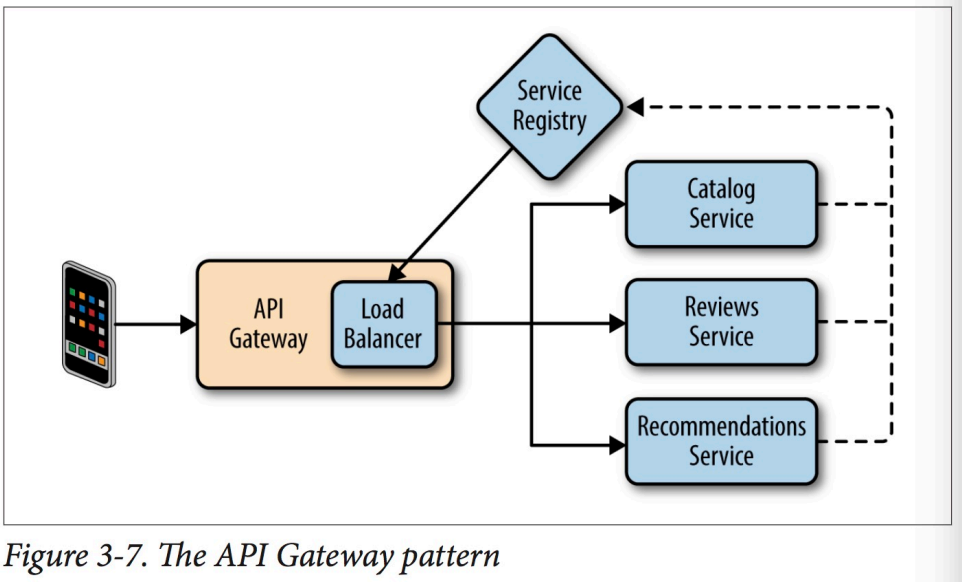

Figure 3-7. The API Gateway pattern

图 3-16

在这个例子中我们将创建一个类似 Netfilx 的网站，它可以在页面上展现一个电影目录，用户可以为这些电影创建评分并进行评论。而且，当用户在浏览某一个标题的页面时，页面上会给予用户相关推荐。为了提供这些能力，需要开发 3 个微服务：

- 目录服务
- 查看服务
- 推荐服务

我们期望的该移动应用的服务的响应如例 3-9 所示：

例 3-10 中的代码利用了 RxJava 的 Observable.zip 方法来并发访问每个服务。在接到三个响应后，代码将它们传递给 Java 8 的 Lambada 表达式处理并生成一个 MovieDetails 实例。该实例可以被序列化并产生如例 3-9 中的响应。

这个例子仅涉及了 RxJava 所有可用功能的一些皮毛，读者可以在 RxJava 的 wiki 上查看进一步信息。

3.3 本章小结

本章中我们讨论了两种帮助我们迁移到云原生应用架构的方法：

分解原架构

我们使用以下方式分解单体应用：

1. 所有新功能都使用微服务形式构建。
2. 通过隔离层将微服务与单体应用集成。
3. 通过定义有界上下文来分解服务，逐步扼杀单体架构。

Example 3-9. The movie details response

```
{
    "mlId": "1",
    "recommendations": [
        {
            "mlId": "2",
            "title": "GoldenEye (1995)"
        }
    ],
    "reviews": [
        {
            "mlId": "1",
            "rating": 5,
            "review": "Great movie!",
            "title": "Toy Story (1995)",
            "userName": "mstine"
        }
    ],
    "title": "Toy Story (1995)"
}
```

图 3-17: 例 3-9

Example 3-10. Concurrently accessing three services and aggregating their responses

```
Observable<MovieDetails> details = Observable.zip(

  catalogIntegrationService.getMovie(mlId),
  reviewsIntegrationService.reviewsFor(mlId),
  recommendationsIntegrationService.getRecommendations(mlId),

  (movie, reviews, recommendations) -> {
    MovieDetails movieDetails = new MovieDetails();
    movieDetails.setMlId(movie.getMlId());
    movieDetails.setTitle(movie.getTitle());
    movieDetails.setReviews(reviews);
    movieDetails.setRecommendations(recommendations);
    return movieDetails;
  }
);
```

图 3-18

使用分布式系统

分布式系统由以下部分组成：

1. 版本化，分布式，通过配置服务器和管理总线刷新配置。
2. 动态发现远端依赖。
3. 去中心化的负载均衡策略
4. 通过熔断器和隔板阻止级联故障
5. 通过 API 网关集成到特定的客户端上

还有很多其他的模式，包括自动化测试、持续构建与发布管道等。欲了解更多信息，请阅读 Toby Clemson 的《Testing Strategies in a Microservice Architecture》，以及 Jez Humbl 和 David Farley（AddisonWesley）的《Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation》。

扫码关注「几米宋」
了解更多

jimmysong.io