

O'REILLY®

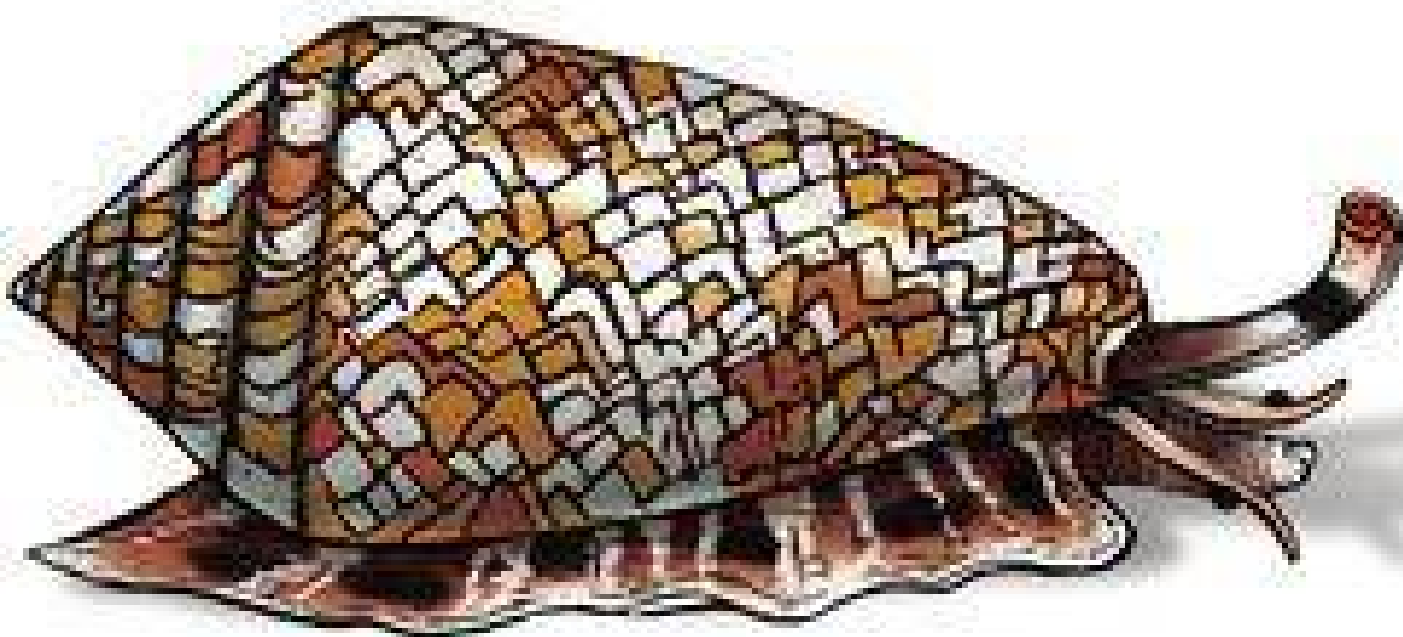
Broadview®

第2版

可伸缩架构

Architecting for Scale, 2nd Edition

云环境下的高可用与风险管理



[美] Lee Atchison 著

张若飞 译



中国工信出版集团



电子工业出版社

O'REILLY®

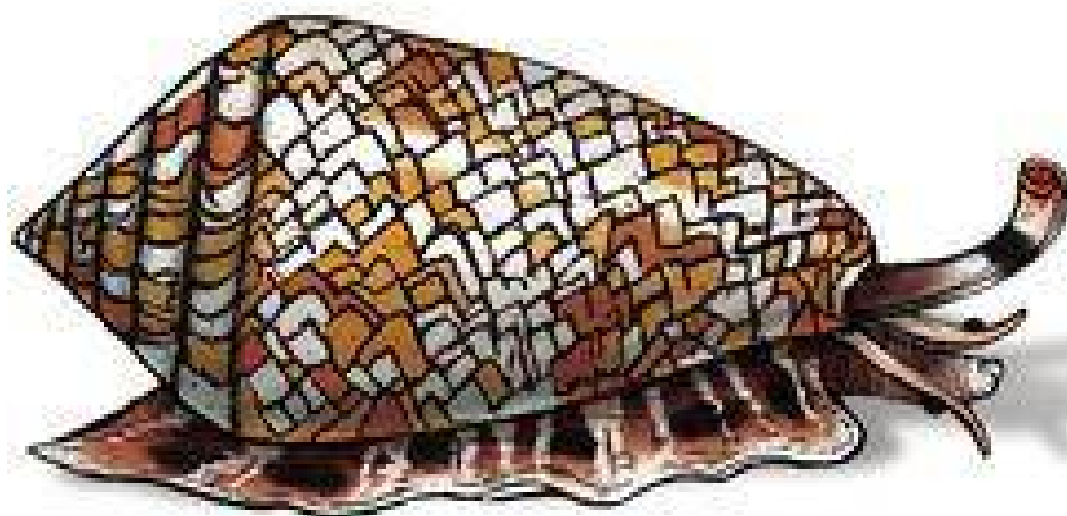
Broadview®

第2版

可伸缩架构

Architecting for Scale, 2nd Edition

云环境下的高可用与风险管理



[美] Lee Atchison 著

张若飞 译



中国水利水电出版社



中国水利水电出版社

版权信息

C O P Y R I G H T

书名：可伸缩架构：第2版. 云环境下的高可用与风险管理

作者：（美）李·艾奇逊（Lee Atchison）著

译者：张若飞

出版社：电子工业出版社

出版时间：2020年9月

ISBN：9787121393433

字数：307千字

版权方：电子工业出版社有限公司

版权所有·侵权必究

内容简介

本书是一本关于现代化软件架构的书。书中介绍了如何构建和更新你的关键应用程序来满足日益苛刻的数字化客户的需求。书中还介绍了如何实现高可用性，如何使用现代化的开发和运维技术来架构应用程序，如何组织开发团队让应用程序和业务获得成功，如何将系统扩展到最大规模，以及如何利用云计算的可用资源来迎接上述挑战。

本书的目标读者包括构建和管理大规模应用程序和系统的软件工程师、架构师、技术经理及总监。如果你管理着软件开发人员、系统可靠性工程师、DevOps工程师，或者经营着一个拥有大规模应用程序和系统的机构，本书中所提供的建议和指导都能够帮助你，让你的系统运行得更加平稳和可靠。

O'Reilly Media, Inc. 介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主
题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

推荐语

不要拿你的生意做赌注。规模化的发展是一个不可避免的趋势。本书会告诉你如何切实可行地做到这一点。

——Colin Bodell, Shopify Plus工程副总裁, Amazon网站应用平台前
副总裁

本书是为想要知道如何实现可伸缩系统的主管、经理和架构师准备的全面指南。

——Ken Gavranovic, New Relic的EVP兼GM, Interland公司（现在是
web.com）的CEO兼创始人

时刻考虑可能出现的故障情况，是构建大规模应用程序的一个关键因素。本书将帮助你学习如何做到这一点，以及如何在用户增长和公司发展过程中，依然保持应用程序正常运行。

——Patrick Franklin, American Express的EVP兼CTO, Google前工程
副总裁

本书会告诉你，如何在应用程序（以及公司）不断扩张以满足用户日益增长的需求的同时，保证一切正常运转。

——Lew Cirne, New Relic公司CEO

致Beth
我的爱人，我的生命，我的一切

译者序

互联网从一穷二白发展到今天，“可伸缩性”“水平伸缩”这些以前只有大型互联网公司才面临的技术挑战，现在几乎是任何一家互联网公司都需要想办法去解决的难题。在我看来，写好一段代码不难，写好一个单机软件也不难，但是写好一个伸缩性良好的大规模分布式系统却很难。这其中最难的地方首先在于思想观念的转变。程序员一开始接触到的就是确定的概念和世界，当程序逻辑一定时，什么样的输入就应该有什么样的输出。

但是在面对大规模系统时，我们首先要承认的是不确定性，即有很多我们想不到、料不到的事情会发生，一个边角的小错误都可能导致整个系统全面“雪崩”。因此，我们不能再把系统当作一个稳定的、确定的程序看待，而需要在设计时充分考虑到那些可能不确定的情况。其次，就是要改变管理整个系统的有效手段。随着系统规模的不断增长，之前的人工操作方式已经明显无法跟上机器的增长速度，规模增长所带来的可靠性、可用性问题时刻都在挑战着整个团队的极限。因此，如何有效地评估、预测、管理、监控一个大规模的系统，这件事本身已经变成一个复杂的系统性工程，需要有效的方法论、原则、工具及最佳实践经验的相应支撑。

当一个系统的规模开始扩张时，可用性往往是系统首先要面临的问题。在激增的流量面前，每一次系统处理缓慢、崩溃都会给公司的业务带来损失，因此理解什么是高可用性对系统保障来说至关重要。为了维持或提高系统的可用性，我们需要一套标准和手段来测量系统的可用性，从而在第一时间发现系统可用性是否发生了变化，并制定相应的解决方案。

随着系统规模不断扩大，各种不确定的因素会被放大，这些不确定性会给系统带来更多的不确定性。在构建大规模可伸缩系统的时候，需要先梳理清楚系统中存在哪些不确定性，把它们定义成风险因素并建立相应的风险模型，定期回顾并确保更新风险因素。在这之后，我们要针对每个风险因素制定相应的应对方案和措施，并持续对风险管理计划、缓和计划和容灾计划进行测试和评估，这样才能避免在风险发生时手忙脚乱。

当系统规模发展到一定程度时，系统本身的架构瓶颈就会突显出来，之前的单体系统已经难以再开发和维护。这时候，不管是从技术角度出发，还是从团队人员角度出发，服务化都是一个必然的趋势。我们需要将原有的系统功能和逻辑拆分成多个独立的服务来进行管理。但是，随着服务越来越多，如何管理各个服务、服务间如何通信，以及团队如何划分都是新的问题。随着业界在这些方面的经验越来越多，经过Netflix、Amazon等公司的实践和推广，微服务的概念出现，完善的解决方案和工具也开始出现。

在解决这些问题之后，我们会发现，系统规模再继续发展就会遇到IDC（互联网数据中心）的限制，单个数据中心甚至没有空间来存放服务器，也支持不了所需的网络带宽。幸运的是，云计算及各种云服务的出现为我们创造了无限的可能。我们无须再考虑硬件和场地的的问题，无须再处理烦琐的运维事项，可以按照使用量来付费，甚至很多服务也无须自己搭建，可以直接使用云服务提供商提供的各种高性能、高可用服务。云计算给我们带来的是一种颠覆性的变化，不仅影响到我们的开发、运维、技术栈，而且更深远地影响到我们对技术和架构的思考方式，进而为创造更多商业价值提供可能。

本书作者Lee Atchison是New Relic的首席云架构师和布道师，负责领导公司基础设施产品的搭建，并且帮助公司设计了一个稳定的、基于服务的系统架构。他曾在Amazon担任了7 年高级技术经理，深刻了解如何设计基于云的、可伸缩的系统，也主导创建了AWS Elastic Beanstalk等产品。在本书中，Lee根据自身多年的经验，为我们分享了在大规模可伸缩系统的设计和实现中应该注意的几个方面，这其中的方法和技巧都是经过时间考验的无价经验。相信各位读者看完此书后，不会再惧怕那些未知的困难，能够在保持系统规模增长的同时，保证系统的高可用性和可伸缩性。

愿以后的每个系统都是高可用、可伸缩的系统。

序

第2版序言

本书是一本内容全面的技术图书，面向那些已经认识到，所有公司都已经从简单地称自己为“数字型企业”，转变为如果不真的这么做就会面临破产的企业管理人员。银行、保险和其他曾经拥有巨大“护城河”的行业，正在被一些新贵企业所冲击。这些企业能够提供令人惊叹的用户体验，是因为它们的运营方式真的像一家数字型企业一样，而不仅仅是纸上谈兵。

本书对于那些希望在实践中拥有一个高可靠性的可操作路线图、实现现代化运维理念（例如，DevOps、网站可靠性工程等），以及想应用最新技术概念和服务（例如，微服务、云计算、边缘计算）的主管、经理和架构师们来说，绝对是一本全面指南。

我很高兴能够在New Relic公司与Lee共事，New Relic是一家能够让企业在全全球范围内监控它们数字业务的公司。在New Relic的时候，Lee周游了世界，帮助许多公司实现了数字化转型，加速了创新的实现，并且交付了百分之百可用的服务。

在许多30分钟的会议上，我一次又一次地看到Lee推动了许多公司的转型。请各位尽情享受这本书！它将对你的公司和职业生涯产生深远的影响！

Ken Gavranovic, New Relic的EVP兼GM,
Interland公司（现在是web.com）的CEO兼创始人

第1版序言

我们生活在一个有趣的时代，可以称它为软件的“寒武纪大爆炸”时代。在发展过程中，构建新系统的成本呈数量级下降，同时系统之间的关联程度呈同等数量级增长。借助于Amazon的AWS、微软的Azure和Google的GCP等资源，我们可以将系统在物理上扩展到一个几年前还只能想象的规模。

这些资源及其似乎无限的能力，正在以各种前所未见的方式，将新的思想、产品和服务极其快速地传播出去。但是，只有当我们构建的系统可以保持扩展的同时，所有这些探索才能成为可能。与以前相比，虽然构建小型系统变得容易很多，但是构建一个可以快速、可靠扩展的系统，并不像增加更多的硬件和存储空间那么容易，实践证明，这要难得多。

每个软件系统都会经历一个可预见的生命周期，从一个人能够完全理解的、小型的、设计精妙的解决方案，迅速增长为一个积累了大量技术债务的庞大系统，随后又逐渐分裂成由一些不完善的服务随机组成的组合，并最终演化成在广度（更多用户）和深度（更多功能）方面均可稳定伸缩的、设计良好的分布式系统。对于这样的系统来说，我们很容易从外部了解要做哪些事情（让它变得更加可靠！），但又很难了解它的内部细节。幸运的是，本书是一本关于这方面不可或缺指南，从可用性到服务层，从比赛日到风险模型，Lee一步步地介绍了影响大规模系统的各个关键因素和实践方式。

Lee加入我们时，是New Relic第一次从仅拥有一个产品正在向多个产品转型的时期，当时我们正沉浸在用户极速增长和公司成功的喜悦中。Lee的到来，为我们带来了他在Amazon的丰富经验，不管是零售业务还是AWS业务都曾经历过巨大的增长。Lee曾是这些团队的领头人，他曾经积极参与过与可伸缩性有关的所有事情，也遇到过很多失败。对我们来说，幸运的是，他已经经历过这些挫折与困苦，其中的教训可以让我们避免再犯同样的错误。

在Lee加入New Relic之前，多年以来，我们一直处于系统服务不可用的尴尬处境。我们原有的庞大系统也逐渐无法支持业务的发展，不管是可用性、可靠性还是性能都不是很好。但是，通过充分运用Lee在本书中所写的各项技巧，我们逐渐克服了这些困难，并构建了如今稳定可靠的企业级服务。其中我们使用的一个工具，建立了可用性工

程的四个级别：青铜、白银、黄金和白金。要达到青铜级，团队必须拥有风险模型及预定义的SLA标准。要达到白银级，团队必须能够监控风险模型中标识出来的问题，并使用比赛日的方式来解决。黄金级意味着风险已经被缓解。白金级如同CMM 5级一样，不仅系统可以自愈，而且我们关注于持续性的改进。我们首先集中精力对第一级的服务进行改进，然后上升到第二级的服务，逐步推进，最终使得所有团队都至少达到白银级，并且大多数团队通过了黄金级，甚至有几个团队达到了白金级。

后来，我加入了InVision App这个更年轻的公司。我又一次经历了从早期成功转向高速增长的过程，一直推荐大家去使用Lee之前带给我的技术和工具。在这个新系统、新产品、新公司的爆炸年代，我强烈建议大家跟我做一样的事：向Lee学习如何构建可伸缩的系统。

——Bjorn Freeman-Benson博士

前言

本书是一本关于现代化软件架构的书。书中介绍了如何构建和更新你的关键应用程序，来满足日益苛刻的数字化客户的需求。书中还介绍了如何实现高可用性，如何使用现代的开发和运维技术来架构应用程序，如何组织开发团队让应用程序和业务获得成功，如何将系统扩展到最大规模，以及如何利用云计算的可用资源来迎接上述挑战。

设计可伸缩架构的整个过程，远不只是知道如何处理大流量。

本书的读者对象

本书的目标读者包括构建和管理大规模应用程序和系统的软件工程师、架构师、技术经理及总监。如果你管理着软件开发人员、系统可靠性工程师、DevOps工程师，或者经营着一个拥有大规模应用程序和系统的机构，本书中所提供的建议和指导都能够帮助你，让你的系统运行得更加平稳和可靠。

如果你的应用程序已经从很小的规模变得很大（并且正在经历着增长所带来的各种问题），那么你可能正在为系统的低可靠性和低可用性烦恼。如果你正在头疼如何管理技术债务及相关的系统故障，本书恰好提供了这些方面的指导，能够帮助你通过降低技术债务，让应用程序更轻松地扩展到更大规模。

编写本书的原因

在Amazon零售和AWS业务团队从事了7年构建高可伸缩应用程序的工作之后，我加入了New Relic这个正在迅速成长的公司。当时，New Relic已经感受到了因为缺少管理高可伸缩应用程序的系统、流程所带来的痛苦，但是尚未完整形成能够扩大其应用程序规模的流程和规范。

在New Relic，我目睹了一个公司在规模扩张过程中所经历的痛苦与挣扎，同时也意识到，还有很多其他公司每天都在经历着同样的痛苦。

现在，我在世界各地旅行，与许多客户和像你一样的人谈论云计算、可伸缩性、可用性，以及如何构建现代化应用程序的关键过程。我会举办一些演讲、小组讨论、课程及研讨会。我会与工程部门的领导和管理人员进行面对面的交流，帮助他们实现目标，并从他们身上学习什么是可行的，什么是不可行的。我会撰写一些文章，接受一些采访，并参加一些播客节目。

编写本书的初衷，是帮助那些正在面对其应用程序高速增长的人们，使其了解到一些有用的流程和最佳实践，避免他们掉入规模扩张过程的各种陷阱之中。

无论你的应用程序每年增长十倍还是百分之十，也无论增长的是用户数量、交易数量、数据存储量还是代码复杂性，本书都可以在构建和维护应用程序方面提供帮助，以帮助你在保持高可用性的前提下实现增长。

如今我们所说的规模

随着应用程序的增长，它们开始变得非常复杂，并且要处理非常巨大的流量。

复杂性的增加意味着脆弱性也在增加。更多的流量意味着管理流量的机制要更加新颖和复杂。

应用程序开发人员很少从一开始构建的就是一个可伸缩的应用程序。我们经常认为自己已经实现了可伸缩性，并且相信已经能够让应用程序扩展到我们可以想象的最高级别。但是更常见的情况是，我们在逻辑上和应用程序中发现了许多错误。这些错误只有在我们开始面

临扩展的问题时才会出现，从而使得应用程序更加难以扩展到可支持更大的流量和更大的数据集。

这就导致了更高的复杂性和更强的脆弱性。

最终，这种规模-脆弱性-规模-复杂性的循环会成为应用程序的死亡螺旋，因为它会遇到断电、宕机和其他各种服务质量及可用性的问题。

但是这些其实都是你的问题，你的客户并不关心这些问题。他们只是想用你的应用程序来完成他们所期望的工作。如果你的应用程序持续宕机、运行缓慢或者不稳定，客户就会放弃你，转而寻找能够处理他们业务的竞争对手。

如何能提高应用程序的可伸缩性，即使是才开始意识到这些因为规模增长导致的限制？显然，我们在应用程序的生命周期中越早考虑可伸缩性，扩展就越容易。但是，我们不希望过度地设计应用程序来获得超出需要的可伸缩性。在生命周期中的任何时刻，你都可以使用各种技术来提高应用程序的可伸缩性。

但是在考虑如何使用扩展应用程序的技术之前，必须先确定什么是应用程序的可用性。在迈出这一步之前，没有什么比这更重要了。如果你现在不考虑这一点，那么随着应用程序的不断扩展，迟早你会搞不清楚它的工作方式，并且开始出现随机的、意想不到的问题。这些问题会导致宕机和数据丢失，并将极大地影响你构建和改进应用程序的能力。此外，随着流量和数据的增加，这些问题只会变得更加糟糕。在做任何其他事情之前，请你先梳理清楚系统的可用性和风险管理。

第2版中的新内容

虽然这本书中讨论的许多概念大多与时间无关，但是许多（例如，无服务器计算）技术必须保持与时俱进，以便能够反映过去四年的行业变化。

另外，在过去的几年里，我一直在世界各地讨论这些话题。从与客户和其他专家的各种交流中，我学到了很多知识，会将学到的这些知识融入这个版本。

本书还对如何使用云计算服务进行了大量的更新。

最后，我重写了第1版中的大量内容，对章节进行了重新组织，使得读者更容易获取相关信息。

使用云计算服务

基于云计算的服务正在以极高的速度增长和占领市场。软件即服务（SaaS）正在成为应用程序开发的规范，这主要缘于对这些基于云计算的服务需求。SaaS应用程序由于其多租户的特点，尤其关注可伸缩性的问题。

随着世界的变化，我们越来越关注SaaS服务、基于云计算的服务和海量数据应用程序，可伸缩性变得越来越重要。

对于我们的云应用程序的规模和复杂性，似乎还没有看到增长结束的迹象。

如今管理可伸缩性的最先进的技术，在未来都将只是基本的功能，而将来的可伸缩性问题的解决方案会让今天的解决方案看起来过于简单和初级。我们的行业将需要越来越复杂的系统和架构来处理未来要面对的规模。

自然地，随着时间的推移，这本书中的一些资料会逐渐过时。我的目的是提供尽可能多的内容，让它们能够经得起时间的考验。

服务和微服务

对于服务和微服务这两个术语的使用，业界存在很多争议。我个人不喜欢“微服务”这个术语，因为它意味着服务的特定规模，这未必是一个健康的假设。许多服务都是小型的，有些确实是“微型的”，但也有许多服务是很大的。大小是否合适是取决于上下文环境的，并且受到许多条件和标准的影响^[1]。在我看来，术语“微服务”的使用会影响这种讨论。然而，我也意识到，“微服务”这个术语在业界非常流行。

还有一些人将“服务”这个术语定义为SOA的一部分，并进一步将这些术语与十多年前流行的某种架构联系起来。我认为这些比较是不

准确和令人困惑的。

我个人倾向使用“服务”这个术语，但我知道很多人更愿意使用“微服务”这个术语。因此，在与其他公司讨论时，这两个术语我都会使用，具体选择哪个取决于上下文环境。在我看来，这两个术语的意思完全相同。

不过，“服务”这个词还有一个值得讨论的用法。例如，当我们指称某个外部服务的时候，比如“Amazon提供了Amazon S3服务”，“服务”这个词的用法看起来是截然不同的，似乎在采用这个词的另一种用法，但实际上两者是一回事。“服务”是一个软件模块，它提供了特定的功能和支持该功能的数据。无论服务由你的开发人员编写还是由Amazon的工程师编写，都是无关紧要的。然而，我确实意识到，有时区分这两种类型的“服务”是很重要的。

这就是我在书中使用这些术语的方式。根据上下文环境，这两个术语可以互换使用。在这本书中，你一定会看到我对“服务”这个词的偏爱。你应该假设这两个术语的意思是完全相同的。当我指称另一家公司提供的特定类型的服务，比如某个云服务时，也会这样表示。因此，在本书中你将会看到“AWS服务”“云服务”“SaaS服务”等术语。

现代化的数字客户体验

在现代数字世界中，软件应用程序已经成为我们的品牌和公司的标志。客户与我们互动是通过我们的软件来实现的。我们的应用程序不仅仅是客户体验的一部分，在许多情况下，它们就是整个客户体验。软件对我们的成功来说至关重要，现代化的客户希望我们的应用程序也要现代化，他们如何看待我们的品牌和公司，在很大程度上取决于他们如何看待我们的软件。

一个不够现代化的应用程序

考虑一下这个例子：在我儿子的智能手机上有一个应用程序，我的儿子必须使用这个程序来获得一些医疗福利。它是一个政府提供的应用程序，由美国政府开发和负责运行。

这个应用程序不是一直都能工作的。当你在一天中某个“不合适”的时间启动应用程序时，你将收到一条错误消息。错误消息的意思是：“该应用程序仅在东部时间星期一到星期五的上午9点到下午5点之间可以使用。”

没错，实际情况就是这样。这就是我儿子智能手机上的一个移动软件应用程序，除了在东海岸的营业时间之内，软件是被禁止使用的。

你的企业可以使用这样的应用程序吗？它在使用时也有这样的限制吗？商业化的企业如果像这样对客户设定限制，还能够继续经营下去吗？

不，我敢打赌，没有一个商业化的企业能够以这种方式生存并且对待它的客户。我们必须为客户提供令人难忘的用户体验。应用程序必须在客户希望使用它们的任何时候都能工作，必须在所有的时间里都可以工作，一天24小时，一周7天。如果不是这样，我们就会让客户感到失望，失望的客户就会离开。

本书导读

管理可伸缩性并不只是管理流量，还包括管理风险和可用性。通常来说，所有这些描述相同问题的不同方式，并且它们息息相关。因此，为了能够合理地讨论可伸缩性问题，我们还必须考虑到可用性、风险管理、团队及组织流程，以及像微服务和云计算这样的现代架构模型。

因此，本书被组织成5个主要部分，每个部分都代表了面向可伸缩架构的主要原则。让我们分别来看一下。

原则1. 可用性：维护现代化应用程序的可用性

现代化软件必须保持高可用性。客户不会容忍服务中断。如果你的应用程序在客户需要时不能工作，那么他们将不会成为你的长期客户。

第 I 部分讨论了应用程序可用性对客户的重要性，以及它如何受到应用程序伸缩的影响。理解、测量和提高可用性是这部分的重点。

第I部分的章节包括：

- 第1章，理解、测量和提高可用性
- 第2章，两次失误的高度——预留从错误中恢复的空间

原则2. 现代化应用程序架构：使用服务

现代化的软件需要使用现代化的应用程序架构。现代化的应用程序架构要求远离单体应用程序，而采用基于服务的架构。

无论是从伸缩流量的角度，还是从伸缩组织处理应用程序能力的角度来看，单体应用程序都很难进行伸缩。单体程度越大，更改应用程序的速度就越慢，能够处理和有效管理它的人员就越少，流量变化和流量增长对可用性产生负面影响的可能性就越大。

面向服务的架构通过在流量伸缩方面提供更大的灵活性来解决这些问题。此外，它们还提供了一个可伸缩的框架，允许大型开发团队能够处理应用程序，使得应用程序本身可以变得更大、更复杂。

第 II 部分的章节包括：

- 第3章，使用服务
- 第4章，服务和数据
- 第5章，处理服务故障

原则3. 组织：为现代化应用程序建立可伸缩性的组织

除非你的开发团队使用现代化的过程管理手段，否则你无法构建现代化的软件。这包括服务所有权责任和开发流程。

应用程序如何伸缩并不是最重要的，但如果你的开发团队的组织架构不支持可伸缩性，或者你的组织没有正确的文化来驱动更高的可用性和更大的可伸缩性，那么你就无法伸缩你的应用程序。

组织好你的团队，使其可以更好地支持你的可伸缩性需求，这会创造出一种文化，从而支持应用程序的可伸缩需求。

第III部分的章节包括：

- 第6章，服务所有权——STOSA
- 第7章，服务分级
- 第8章，服务等级协议

原则4. 风险：现代化应用程序的风险管理

你不能消除一个系统中的所有风险，这是不可能的。所有复杂的系统都有其固有存在的风险。我们必须学会如何管理风险，并将风险作为评估技术债务和对应用程序改进做出决策的一个工具。

理解风险、测量风险和根据测量的风险结果来确定行为的优先级，是构建一个具有高伸缩性、高可用性的应用程序的重要工具。

第IV部分的章节包括：

- 第9章，如何在设计可伸缩架构时使用风险管理
- 第10章，比赛日
- 第11章，构建低风险系统

原则5. 云计算：利用云计算

现代化应用程序中的高可用性要求能够灵活伸缩。我们负担不起为了满足应用程序的峰值需求，而采购过剩的基础设施。我们必须根据当前的需要，按需动态地分配和消费基础设施资源。

动态的基础设施，以及能够支持和优化动态基础设施的应用程序，是构建高伸缩性、高可用性的应用程序的一个关键架构组件。

动态的基础设施是公有云的基础优势。利用好公有云对于保证应用程序的高可用性至关重要。

第V部分的章节包括：

- 第12章，使用云计算来设计可伸缩架构
- 第13章，云计算改变的5个行业趋势
- 第14章，SaaS和租赁类型
- 第15章，在AWS云上分发你的应用程序
- 第16章，托管的基础设施
- 第17章，云资源分配
- 第18章，无服务器计算和函数即服务
- 第19章，边缘计算
- 第20章，地理位置对云计算的影响

这些是构建满足客户现代化需求的应用程序的5个关键原则。这些原则构成了设计可伸缩架构的基础。

在线资源

本书的网站（网址参见链接1）[\[2\]](#)上提供了一些额外的信息，包括补充材料的链接地址。你可以在我的网站（网址参见链接2）上找到更多关于我的信息，也可以关注我的博客（网址参见链接3）。

本书使用的约定

本书使用以下一些排版约定：



该图标表示提示或者建议。

该图标表示一般说明。



O'Reilly在线学习 (O'Reilly Online Learning)

O'REILLY® 40多年来, O'Reilly Media为许多企业提供技术咨询、商业培训、知识和洞察力, 帮助企业获得成功。

我们独特的专家和创新者网络会通过书籍、文章、会议和在线学习平台分享他们的知识和专长。O'Reilly的在线学习平台可以让你按需访问在线培训课程、深度学习路径、交互式编程环境, 以及来自O'Reilly和200多个其他出版商的大量文章和视频。更多信息, 请访问O'Reilly公司的网站。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版社。

美国:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国:

北京市西城区西直门南大街2号成铭大厦C座807室 (100035)

奥莱利技术咨询 (北京) 有限公司

O'Reilly的每一本书都有专属网站, 你可以在那里找到本书的相关信息, 包括勘误列表、示例代码以及其他信息。本书的网站地址是:

<https://oreil.ly/architecting-for-scale-2e>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅网站
http://www.oreilly.com。

在Facebook上找到我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

致谢

虽然我无法一一列举出所有为本书出版做出贡献的人，但我还是想特别感谢以下帮助过我的人。

- Ken Gavranovic，“朋友”这个词已经不足以来形容我俩的关系。永远相信猴子的力量。
- Bjorn Freeman-Benson，在开始编写本书时，你就给了我很多支持，并且提供了加入New Relic的工作机会，让我能够去思考如何编写本书。我很高兴我们的友谊能够在一起共事之后持续至今。
- Kevin McGuire，我们既是朋友也是知己。我们从在New Relic开始就在一起工作，你的远见和想象力为我的职业生涯指明了焦点和方向，而我直到今天也从未偏离。
- Abner Germanow、Darren Cunningham、Jay Fry、Bharath Gowda和Robson Grieve，你们为我争取到了在New Relic工作的机会。与你们在一起的这段日子，是我感到最开心、最有收获，以及对我个人来说最充实的时光。我非常怀念那些时光。尤其是Abner，如果没有你，我就不会有今天的事业。你引导我进入这个新的角色，并帮助我从一个工程师和架构师成长为一个战略家、专家和思想领袖。谢谢你相信我，并在这条路上指导我。
- Jim Gochee，是你向我介绍了New Relic这个产品，最终它也成就了我的工作。

- Lew Cirne, 是你的远见带给了我们New Relic, 也带给了我一份工作和一个家庭。每次跟你进行一对一的谈话后, 我都会被你的幽默和热情所感染和打动。怪不得New Relic会如此成功。
- Kevin Downs, 我的朋友和云计算伙伴。替我向老鼠问好。顺便说一下, 容器才是王道。
- Brandon Sangiovanni, 我的朋友。从美国职棒大联盟到漫威和米奇, 你已经搞定了我在本书中讨论的许多挑战, 而你仍然毫发未损、笑对人生! 感谢你的支持、你的知识, 以及最重要的, 你的友谊。
- Abbas Haider Ali, 我非常敬重你。我们都扮演着行业思想领袖的角色, 有一个可以交流想法和提出建议的人简直太好了。你对这本书早期草稿的建议使它变得更好。谢谢你!
- Kurt Kufeld, 你曾是我的导师, 帮助我适应了这个古怪、混乱、漫长、乏味, 以及最终让我收获颇丰的世界, 它叫Amazon。
- Greg Hart、Scott Green、Patrick Franklin、Suresh Kumar、Colin Bodell和Andy Jassy, 你们给了我之前从未想过的在Amazon和AWS工作的机会。
- Brian Anderson, 我的编辑, 以及现在O'Reilly的编辑Kathleen Carr, 是你们让本书和O'Reilly的很多项目得以进行。Brian负责出版了本书的第1版。而Kathleen鼓励我在第2版中增加了更多内容, 还包括相关的课程、培训和知识讲座。

O'Reilly的Amelia Blevins, 你对本书的版式、布局和内容提出了实质性的编辑建议。这些建议对本书的质量和可读性产生了巨大的提升。

向所有在读完第1版后联系我的人致敬, 你们给了我赞扬、鼓励和建议, 感谢你们的帮助, 是你们帮助我保持写作的动力, 并给了我关于改进第2版的想法。

感谢我的家人, 尤其是我的妻子Beth, 你永远是指引我生活方向的灯塔。与你在一起, 我的生活变得越来越光明, 我前进的道路也越

来越清晰。

所有帮助过我的人，以及所有我没有提到的人，我也衷心地感谢你们。

最后，我还必须介绍一下那些毛茸茸的小家伙：Issie，爱打鼾的西班牙猎犬；Abbey，快乐的柯基犬；以及Budha，一只好动的小猫咪，它们给我带来的可不仅仅是本书中的错别字。

读者服务

微信扫码回复：39343



- 获取博文视点学院20元付费内容抵扣券
- 获取本书参考资料中的配套链接
- 获取更多技术专家分享的视频与学习资源
- 加入读者交流群，与更多读者互动

[1] 我在第3章的“深入了解服务”一节中更详细地讨论了如何看待服务的大小。

[2] 请访问<http://www.broadview.com.cn/39343>下载本书提供的附加参考资料。正文中提及参见“链接1”“链接2”等时，可在下载的“参考资料.pdf”文件中查询。

第 I 部分

原则1. 可用性：维护现代化应用程序的可用性

没有高可用性，就没有可伸缩性。

现代化的软件必须具有高可用性。客户不会容忍服务中断。如果你的应用程序在客户需要时不能工作，那么他们将不会成为你的长期客户。

应用程序的可用性对于我们和我们的客户非常重要，并且影响着我们如何看待应用程序的可伸缩性。理解、测量和提高可用性是本部分各章节的重点。

第1章 理解、测量和提高可用性

重大赛事

今天是周日—观看重大赛事的日子。你邀请了最好的20位朋友到家里来，打算在新买的300英寸Ultra Max TV上观看这次比赛。所有人都来了，你的房子里放满了零食和啤酒，每个人都在大笑。比赛就要开始了，然后……

灯灭了……

电视不亮了……

对于你和你的朋友来说，比赛也结束了。

失望至极。你拨打当地电力公司的电话。服务人员冷冷地说：“很抱歉，我们只能保证当地电网95%的可用性”。

为什么可用性很重要？因为你的用户期待你能在任何时间都提供服务。任何小于100%可用性的系统都会对你的业务带来灾难性的后果。

如果你的系统有很多不错的功能但是没法使用，那么没人会关心你的系统本身如何。

在设计面向可伸缩的系统架构时，最重要的一个话题就是可用性。虽然有些公司和服务认为一定的宕机时间是合理且允许的，但是对于大多数业务来说，宕机不可能不影响用户的满意度，甚至最终会影响到公司的发展。

以下是所有公司在确定系统可用性对公司和用户的重要程度时，必须问自己的一些基本问题。这些问题本身，以及其显而易见的答案，就是为什么可用性对可伸缩性是如此重要的核心：

为什么要从你这里买？

如果用户在需要使用你的服务的时候却不能用，他们为什么要买你的服务？

你的客户会怎么想？

如果用户在需要使用你的服务的时候却不能用，他们会有什么样的想法和感受？

你的客户高兴吗？

如果你的服务不可用，你如何能让用户感到高兴、给公司赚钱、达成你的商业承诺和目标呢？

只有系统是可用的，才有可能让用户感到高兴并愿意使用你的系统。因此，系统可用性和用户满意度之间存在着直接且重要的联系。

高可用性对于构建高可伸缩系统是一个极其重要的因素，因此我们在本书中会花费很大篇幅来讨论这个话题。当面临大量的业务需求时，你如何构建一个高可用的系统（服务、应用程序或者环境）呢？

可用性与可靠性

可用性与可靠性是两个很相似但完全不同的概念。理解这两者之间的区别，对于理解本书后续内容至关重要。

在我们的语境中，可靠性一般指一个系统的质量。通常来说，它意味着一个系统能够按照技术标准持续运行的能力。如果一个软件通过了所有的测试套件，并且基本完成了它应该做到的事情，那么我们可以说它是可靠的。

“我的查询返回的响应是正确的吗？”

而我们所提到的可用性，一般指系统在其能力范围内执行任务的能力。系统是否运转正常？是否可以操作？是否可以响应？如果答案是“是”，那么系统就是可用的。

“我收到回复了吗？”

“回复及时吗？”

如你所见，可用性和可靠性非常相似。如果系统不可靠，它也很难是可用的，如果它不可用，也很难是可靠的。

以下是我们对这些术语的正式定义。

可靠性

系统是否具备无差错地执行预期操作的能力。

可用性

为了执行这些操作，系统当前可运行的能力。

一个计算 $2+3$ 却得到结果6的系统，其可靠性很低。一个计算 $2+3$ 却永远不返回结果的系统，其可用性很低。可靠性问题通常可以通过测试来修复，而可用性问题通常较难解决。

你会因为在系统中引入了一个bug，导致计算 $2+3$ 得到了6。这通过测试套件可以很容易发现并修复。

但是，假设你的应用程序可以很可靠地计算 $2+3=5$ 。现在假设它运行在一个网络连接不好的计算机上，结果如何？有时它可能返回5，有时它可能什么都不返回。这时候，应用程序是可靠的，但它不是可用的。

在本书中，我们主要关注于如何设计高可用的系统架构。我们会假设你的系统是可靠的，假设你知道如何构建和运行测试套件。我们只会在可靠性对系统架构或者可用性造成直接影响时才讨论它。

什么导致了低可用性

究竟是什么原因导致之前运行正常的系统，却慢慢变得不可用了呢？通常，这会有许多种原因。

资源耗尽

用户数量的增加会导致系统使用的数据量增加，从而可能导致系统资源耗尽，应用程序运行越来越慢并最终无法响应。

预期之外的压力变化

随着应用程序被越来越多的人使用，可能需要对代码和应用程序进行修改以支撑不断增加的压力。这些改动通常都是在最后一刻被草草实现，缺乏足够的考虑或计划，因此也增加了问题出现的可能性。

流动行为的增加

当应用程序高速发展时，通常需要更多的开发人员、设计师、测试人员和其他人来开发并维护它。大量的个体共同协作会产生大量的流动行为，包括新功能、变更功能，或者只是一般的维护工作。开发和维护应用程序的人员越多，就会产生越多的流动行为，从而增加了相互之间产生负面作用的可能。

外部依赖

应用程序依赖的外部资源，例如，SaaS服务、基础设施或者云服务越多，由于这些资源导致的可用性就会越多。

技术债务

随着应用程序复杂性的增加，通常会导致技术债务（例如，通常随着应用程序逐渐发展和成熟，对软件未实现的修改和未修复的bug也会逐渐积累）。技术债务会增加可用性问题的可能。

所有快速发展的应用程序都有一个、多个或者所有这些问题。因此，之前运行正常的程序，很可能会逐渐出现可用性的问题。通常这些问题会慢慢浮现，有时也会突然发生。

但是大多数不断发展的应用程序最终都会面临可用性的问题。

可用性问题会增加你的经营成本，增加你的用户的成本，降低用户对你的信任度和忠诚度。如果系统一直存在可用性问题，你的公司不可能长期存活。

构建可伸缩的应用程序，意味着构建高可用的系统。

测量可用性

测量可用性对保证系统高可用非常重要。只有通过测量可用性，你才能了解应用程序当前的运行状况，以及检查可用性随时间发生的变化。

测量Web应用程序的可用性时，最常用的办法是计算用户可访问的时间百分比。我们可以通过以下公式来计算一段时间内的可用性：

网站可用性百分比 = (该期间的总秒数 - 系统宕机的秒数) / 该期间的总秒数

我们举一个示例来说明。假设在整个4月，你的网站宕机了两次，第一次宕机37分钟，第二次宕机15分钟。那么你的网站的可用性是多少呢？

从这个示例中可以看出，只要很短一段故障时间，就会对可用性百分比造成影响：

系统宕机的总秒数 = (37 + 15) × 60 = 3,120秒

该月份的总秒数 = 30天 × 86,400秒/天 = 2,592,000秒

网站可用性百分比 = (该期间的总秒数 - 系统宕机的秒数) / 该期间的总秒数

= (2,592,000秒 - 3,120秒) / 2,592,000秒

网站可用性百分比 = 99.8795

因此，你的网站的可用性为99.8795%。

N个9

通常我们会用“N个9”来形容可用性。这是表示高可用性百分比的一个简化方式。表1-1列举了各个9的含义。具有“2个9”可用性的应用程序必须在99%的时间内可用。这意味着在一个月里，应用程序宕机432分钟仍然可以达到99%的可用性目标。相比之下，一个“4个9”的应用程序需要在99.99%的情况下都是可用的，这意味着它在一个月里只能宕机4分钟。

表1-1：N个9

N个9	百分比	每个月的故障时间
2个9	99%	432 分
3个9	99.9%	43 分
4个9	99.99%	4 分
5个9	99.999%	26 秒
6个9	99.9999%	2.6 秒

从上面的示例中，我们可以看到前述网站的可用性百分比小于3个9（99.8795%，近似于99.9%）。对于一个要维持5个9的网站来说，每个月只能宕机26秒。

如果你希望系统是高可用的，那么什么样的可用性数值是合理的呢？对于这个问题，没有唯一的答案，因为这取决于你的网站、用户期望、业务需要以及业务期望。你需要自己来决定什么样的数字能够满足你的业务。

通常，对于简单的Web应用程序来说，3个9的高可用性是可接受的。通过表1-1可以看到，这表示每个月的故障时间不能超过43分钟。

计划中的故障也依然是故障

不要想当然地以为自己的网站是高可用的，计划中和日常维护工作所导致的系统不可用时间也要计算在可用性百分比之中。

我经常听到这样的评论：“我们的应用程序从来也没有出现过故障，因为我们会经常性地系统维护。通过每周安排两个小时进行维护，我们保证了系统的高可用性。”

这真的能保证应用程序的可用性吗？让我们来计算一下。

网站可用性百分比 = (该期间的总秒数 - 系统宕机的秒数) / 该期间的总秒数

每周的小时数 = 7天 × 24小时 = 168小时

每周不可用的小时数 = 2小时

网站可用性（没有故障） = (168小时 - 2小时) / 168小时 = 98.8%

网站可用性（没有故障） = 98.8%

即使应用程序没有出现任何故障，至多也只能达到98.8%的可用性。这甚至低于2个9的标准（98.8%，与99%类似）。

计划中的维护和未预期的故障其实效果差不多。你的用户希望应用程序是可用的，一旦不可用，你的用户就会有负面的体验。他们不会关心你这是计划中的维护还是意外的故障。

通过数字来体现可用性

测量可用性对于保证系统高可用十分重要，不管是现在还是将来。本节讨论了测量可用性的常用方法，并提供了一些如何选择合适的可用性的建议。

提高下降的可用性

你的应用程序在线上运转正常，系统一如往常，团队高效运转。所有事情看上去都是那么好。你的流量持续增长，销售部门非常高兴。所有一切都很好。

然后出现了一些小意外。你的系统发生了一次预料之外的故障。但是这样还好，你的可用性到目前为止还是非常好的。一次故障并不是什么大问题。你的流量还在持续增加。每个人都对此不以为然——这只是“一件小事”罢了。

然后另一次故障又发生了。好吧，总体上来说，我们还是干得不错的。不用慌张，这只不过是另外“一件小事”罢了。

然后又出现了一次故障……

现在你的CEO开始有一点儿担心了。用户开始询问出了什么事情。你的销售团队也开始担心了。

然后又出现了一次故障……

突然，你曾经认为稳定且正常的系统正在变得越来越不稳定，你的故障正在被越来越多的人关注。

现在你遇到真的麻烦了。

发生了什么事？保持系统高可用是一件很难的事情。可用性开始下降时，你应该做什么？你的应用程序可用性已经下降或者正在开始下降，而你需要提高它来保证让用户满意，你应该做什么？

当可用性下降时，知道如何处理会帮助你避免陷入恶性循环。你需要如何来避免可用性下降？以下是一些关键的步骤：

- 测试并跟踪当前的可用性
- 将手动流程自动化
- 自动化部署过程
- 维护和跟踪管理系统中的所有配置
- 允许快速更改并进行实验，并且保证如果出现了问题可以轻松回滚
- 以不断改进应用程序和系统为目标
- 随着应用程序的变化和增长，将可用性作为一个核心问题来解决

下面几节将进一步详细介绍这些关键步骤。

测试并跟踪当前的可用性

要理解可用性发生的变化，你必须首先测量出当前的可用性是多少。通过跟踪系统可用及不可用的时间，计算出可用性百分比，可以帮助你了解一段时间内的可用性程度。通过这个值，你可以判断出可用性是在提高还是降低。

你应当不断地监控可用性百分比，并定期收集结果。此外，你需要覆盖系统中所有的关键事件点，例如，更改系统配置或者升级的时候，这样可以了解到系统事件和可用性问题的关系，也能够帮你确定系统的可用性风险。

接下来，你必须从可用性的角度出发来理解系统应当如何运行。服务分级是一个可以帮助你管理系统可用性的工具。服务分级指的是为各个服务打上一些简单的标签，表示该服务对于系统的关键程度。这使得你和团队能够区分出哪些是至关重要的服务，哪些是很重要但非必需的服务。我们将在第7章深入讨论服务分级。

最终，你需要创建并维护一个风险模型。通过这个工具，你可以了解当前的技术债务以及相关的风险。我们会在第9章来更加完整地介绍风险模型。

既然你已经知道了如何跟踪可用性，以及如何确定并管理你的风险，你需要定期审查你的风险管理计划。

除此之外，你应该建立并实施风险缓和计划来降低系统的风险。它可以为你和开发团队提供一系列具体的实施任务，解决掉系统中最危险的部分。我们会在第9章来详细讨论这部分内容。

将手动流程自动化

为了维护高可用性，你需要移除未知以及不可控的因素，而执行手动操作就是一个常见的为系统带来不可控或未知结果的原因。

你应该永远不要在生产环境中执行手动操作。

当你对系统进行了某项更改后，它对系统造成的影响可能有益，也可能有害。因此，仅允许使用可重复性的任务会给你带来如下好处：

- 在实施更改前进行测试的能力。在更改之前进行测试，对于避免因失误造成的故障来说至关重要。
- 完全控制任务执行的能力。这使你可以在进行更改之前，对更改的内容和步骤进行完善。
- 由第三方审查更改任务内容的能力。这降低了更改任务产生未预期结果的可能性。
- 通过版本控制来管理任务的能力。版本控制系统可以让你清楚了解任务被更改的时间、人员以及原因。
- 对相关资源进行更改的能力。通过更改来提高一台服务器的能力是很不错，但是如果能够将相同的更改，完全一样地应用到所有受影响的服务器上，能够为我们带来更大的价值。
- 让所有相关资源保持一致的能力。如果你总是对资源（例如，服务器）进行临时的手动更改，那么它们之间会逐渐产生不同的行为方式。由于无法找到可比较的基准行为，这会增加诊断问题的难度。
- 实施重复性任务的能力。重复性任务都是可审计的任务。对于可审计的任务，可以事后从整个系统层面分析其影响是积极的还是消极的。

事实上，很多系统的生产环境没有任何人有权访问。唯一能够访问的方式就是通过自动化的程序。正是出于我们以上提到的几点原因，这些系统的管理员才将系统隔离了起来。

总而言之，如果你不能重复执行一个任务，那么这个任务就没什么用。事实证明，自动化执行更改可以保持系统和应用程序的稳定性。这其中包括服务器配置更改、性能调优、重启服务器、重启任务、改变路由规则，以及升级和部署软件包。现在我们来看一些你应该使用的可重复任务的例子。

自动化部署

通过自动化部署，可以确保整个系统中的更改都是一致的，并且下次再进行相似更改时，你可以知道它所带来的影响。除此之外，自动化部署系统可以帮助你更可靠地将系统回滚到已知的良好状态。

配置管理

对于服务器的内容，不要“临时手动地更改某个配置变量”，而要使用自动化的方式进行更改。

至少，你可以编写一段进行更改的脚本，然后通过你的软件更改管理系统来检查该脚本。这样，你可以对系统中的所有服务器进行统一更改。此外，当需要向系统中添加新的服务器或者替换旧服务器时，使用已有的配置可以帮助你更安全地将新服务器添加到系统中，可将影响降到最低。

但是一种更好的、与现代化的最先进的配置管理最佳实践相一致的方法是，引入一个称为基础设施即代码（Infrastructure as Code）的概念。基础设施即代码包括，以一种标准的、机器可读的规范来描述你的基础设施，然后通过一个基础设施工具来传递该规范，该工具可以创建或者更新你的基础设施和配置以使其符合规范。Puppet和Chef等工具可以帮助你简化这个过程的管理。

然后你可以将该规范提交到版本控制系统中，这样就可以跟踪规范的变更，就像跟踪代码变更一样。只要有人对规范进行了更改，就可以通过基础设施工具来运行规范，从而更新你当前的基础设施，使其与规范相匹配。

如果有人需要对基础设施或者其配置进行更改，那么他们必须首先对规范进行更改，将更改提交到版本控制系统，然后通过基础设施工具来“部署”这次更改，从而更新当前的基础设施，使其与规范相匹配。通过这种方式，你可以：

1. 确保基础设施的所有组件都具有一个一致的、已知的以及稳定的配置。

2. 跟踪对基础设施的所有更改，以便在需要时回滚这些更改，或者用来帮助解决与系统事件和宕机相关的问题。

3. 允许结对评审过程——一个类似代码评审的过程，以确保对基础结构的更改是正确和适当的。

4. 允许创建重复的环境，以便使测试、准生产和开发环境保持与生产环境相同。

同样的流程适用于所有的基础设施组件。这其中不仅包括服务器及其操作系统配置，还包括其他的云组件、VPC、负载均衡器、交换机、路由器、网络组件，以及监控应用程序和系统。

为了让基础设施即代码的管理手段发挥作用，必须始终对所有的系统更改使用这个流程。无论何种情况，任何绕过基础设施管理系统进行更改的行为都是不可接受的。永远不能这样做。

你可能都想象不到，我接收到多少封像这样的运维更新邮件：“我们的服务器昨天晚上遇到了一个问题。我们遇到了操作系统最大打开文件数量的限制，于是我手动修改了系统变量，并增加了可允许打开的最大文件数量，服务器已经恢复正常。”

这意味着，一旦有人不小心覆盖了之前的更改，服务又会出现问题，因为没有任何文档记录了这次更改。或者，其他运行相同程序的服务器也会遇到这个问题，因为它们没有进行这个更改。

或者有人进行了另一个改动，这样就会破坏应用程序，因为它与你刚才所做的未记录的更改是不一致的。

一致性、可重复性以及细节的专注，是成功执行配置管理流程的关键因素。我们这里所描述的标准、可重复的配置管理流程，对于让你的大规模系统保持高可用性至关重要。

更改实验和高频次更改

拥有一个高可重复的、高自动化的更改和升级流程，带来的另一个好处是允许你对更改进行实验。假设你需要对服务器进行一次更

改，并且你相信这会提高应用程序的性能。通过自动化的配置管理流程，你可以做到以下几点：

- 用文档记录你想要进行的更改。
- 让经验丰富的人来审查这次变更，他们可以提出建议和改进措施。
- 在一个测试或者预发布环境中进行测试。
- 快速、简单地部署更改内容。
- 快速检查结果。如果这次更改没有达到想要的结果，可以快速回滚到上一次的正常状态。

实现这个流程的关键是，拥有一个具有回滚能力的自动变更流程，以及对系统进行频繁、简易、小范围更改的能力。^[1]前者可以让你对多台服务器进行统一更改，后者可以让你对更改内容进行实验，并且在失败的时候进行回滚，让用户几乎感觉不到影响。

自动化的变更完备性测试

当拥有自动化的变更和部署流程时，^[2]你可以对所有更改实现一个自动化的完备性测试。你可以使用浏览器测试程序来测试Web应用程序，或者使用一个合成测试程序来模拟用户的交互。

当准备好将变更内容部署到生产环境时，你可以让部署系统先将它们自动部署到某个测试或者预发布环境上。然后，就可以运行这些自动化测试，并检验变更内容的正确性了。

如果通过所有测试，你可以将变更内容按照统一的方式部署到生产环境中。根据测试的不同情况，应该定期对生产环境进行测试，并验证变更的正确性。

通过让整个流程自动化，你会对每次更改更加有信心，知道它们不会对生产系统造成负面的影响。

改进你的系统

现在，你已经拥有了一个可以监控可用性的系统，一个可以跟踪风险和缓和风险的方法，以及一个简单、安全、统一进行变更的流程，可以将精力集中到如何提高应用程序的可用性这一点上了。

你需要经常检查你的风险模型以及恢复计划，将它们作为问题来复盘流程中的一个部分。执行那些能够降低风险模型中风险的计划。将这些变更通过自动化、安全的方式进行部署，并进行完备性测试。之后，你应当检查这些变更对可用性的提升效果。持续这个流程直到可用性达到你希望并且应该达到的程度。

时刻关注不断变化和发展中的应用程序的可用性

随着系统的发展，你需要处理越来越大的流量和越来越多的数据。本书中的大量内容可以帮助你解决不断变化和发展中的应用程序的可用性和伸缩性问题，尤其是在第2章讨论的如何管理失误和错误。还可以通过阅读第7章来确定影响可用性的关键服务的服务分级制度。此外，第8章会讨论的服务等级协议（SLA）的管理。

通常来说，你的应用程序会不断发生变化。因此，你的风险管理、风险缓和、应急方案以及恢复计划也需要不断进行相应的改变。

当可用性开始下降时知道该如何处理，可以帮助你避免进入恶性循环。

提高应用程序可用性的5个要点

构建一个高可用、可伸缩的应用程序不是一件容易的事，也不会有天上掉下来的馅饼。问题总会以你从未预期的方式出现，让你精心设计的功能对所有用户都停止工作。

这些可用性问题通常会在你想不到的地方出现，甚至一些最严重的问题会来自最不可能出现的地方。

一次简单的图标故障

这是我亲身经历的一个因为忽视依赖故障的典型示例。我们的一个应用程序向用户提供了一个服务，为每个页面顶部提供一个自定义的图标，来表示当前登录的用户。这个图标由一个第三方系统负责生成。

有一天，这个生成图标的第三方系统发生了故障。我们的应用程序以为该系统总是会正常运行，因此并不知道如何处理这种情况。结果，我们的应用程序也跟着发生故障。仅仅因为生成图标这样一个非常小的“功能”出现故障，整个系统无法提供任何服务。

如何才能避免这样的问题呢？如果我们能够预料到第三方系统可能发生故障，就可以在设计过程中考虑到这个故障发生的场景，从而发现我们的应用程序也会随之发生故障。这样，我们就能添加一些逻辑来检查第三方服务，在问题发生时删除图标，或者在问题发生时捕获错误，避免它传递下去并影响页面的其他部分。

一次小小的检查和一些错误恢复机制，就可以帮助应用程序保持正常运行。否则，我们的应用程序就会经历严重的服务中断。

所有这一切都是因为缺少了一个图标。

没人能够预料到问题会在何处发生，也不可能依靠测试来发现所有这些问题。许多问题都是系统性的问题，而不仅仅是代码的问题。

为了发现这些可用性的问题，我们需要后退一步，系统地去了解应用程序的运行机制。以下是5个你可以关注并且应当关注的要点，它们能够帮助你的系统在规模增长的同时保证高可用性：

- 时刻考虑应对故障
- 时刻考虑如何伸缩
- 缓和风险
- 监控可用性
- 以可预期及明确的方式来处理可用性问题

让我们来详细讲解其中的每一个要点。

要点1：时刻考虑应对故障

正如Amazon的CTO Werner Vogels所说，“所有事情每时每刻都会失败”。你应当提前为应用程序和服务发生故障而做出计划。问题迟早会产生。不过现在，我们要讲的是如何解决它。

假设你的应用程序发生了故障，那么它是如何发生的？当你构建系统的时候，应当在设计和实现的方方面面都考虑可用性。

设计

你考虑过任何设计模式吗？你使用它们来帮助你提升软件的可用性了吗？

通过使用一些设计模式，例如捕获底层异常、重试逻辑和断路器，可以帮助你捕获错误并尽可能避免影响其他功能。这样，你就能够限制问题的影响范围，即使应用程序的某些部分出现问题，依然能够提供其他一些有用的功能。

依赖

如果你依赖的组件出现了故障，你会怎样做？如何进行重试？如果问题是一个无法恢复的（硬件）故障，你会怎样做？如果是一个可恢复的（软件）故障呢？

断路器模式在处理依赖故障时非常有用，因为它可以降低依赖故障对你的系统的影响。如果没有断路器，可能会因为依赖故障而降低系统的性能（例如，需要一个很长的超时机制来检测故障）。而使用了断路器，你可以“放弃”并停止使用某个依赖，直到你确认它已经恢复了正常工作。

用户

如果出现问题的原因是系统的某个用户，你会怎样做？你能够处理海量的请求吗？你能够限制海量的流量吗？你能够处理传入的垃圾数据吗？如果数据量非常大，你会怎样做？

有些时候，拒绝式服务可能来自“友方”。例如，可能会因为应用程序的用户看到一个临时活动，而导致增加大量请求。或者，用户程序中的一个bug，可能导致他们向你的应用程序拼命地发送请求。如果这样的事情发生了，你会怎样做？流量突增会让你的应用程序宕机吗？你能否检测出这种问题，并通过限制请求的速度来降低或者消除其影响吗？

要点2：时刻考虑如何伸缩

你的系统现在运行正常，并不意味着它明天还能够继续运行正常。大多数Web应用程序的流量都是在不断增加的。一个今天产生一定流量的网站，明天可能会产生远比你想象大得多的流量。当你构建系统时，不要只考虑当前的流量，还要考虑未来的流量。

具体一点，这可能意味着：

- 需要设计出能够增加数据库数量和容量的架构。
- 考虑限制数据伸缩的原因。当数据库达到容量极限的时候会发什么？你需要确认这些限制因素并在到达极限之前解决它们。
- 应当能够很容易地添加额外的应用程序服务器。这通常需要仔细考虑在何处和如何来维护状态，以及流量是如何路由的。
- 将静态流量导向离线提供方。这样系统只需要处理必要的动态流量。使用外部的内容分发网络（CDN）不仅可以降低网络需要处理的流量，还能够利用CDN的伸缩效率将静态内容更快地分发给用户。
- 考虑是否可以静态生成一些动态资源。通常来说，看上去动态显示的内容实际上大多数是静态的，并且生成静态内容可以提高应用程序的可伸缩性。这种“应该静态的动态资源”有些时候隐藏在你想象不到的地方。

内容究竟应该是静态的还是动态的

通常，看上去是动态的内容实际上大多数是静态的。设想网站上常见的顶部导航栏，绝大多数时候，其中的内容都是静态的，但是偶尔也会出现一些动态的内容。例如，如果你没有登录，页面的顶部可能会显示“请登录”，如果你已经登录了则显示“你好，Lee”（当然前提是你的名称是Lee）。

这是否意味着整个页面都必须动态生成呢？显然不是。除了页面的登录/问候部分，其他部分都是静态的，通过CDN可以轻松地进行分发并节省你的计算资源。

当导航栏中大多数内容都是静态内容时，你可以在用户的浏览器中动态地将变更内容添加到页面上（例如，根据具体情况添加“请登录”或者“你好，Lee”的内容）。通过这些动态数据进行分组，并与静态内容加以区分，可以提高Web页面的性能，降低应用程序需要处理的动态数据量。这样可以提高可伸缩性，并最终提高可用性。

要点3：缓和风险

保持系统高可用需要消除系统中的风险。当系统发生故障时，通常我们已经在之前将故障原因确定为了风险。因此，确定风险是提高可用性的一个重要方法。所有的系统中都存在以下这些风险：

- 系统崩溃的风险
- 数据库崩溃的风险
- 返回结果不正确的风险
- 网络连接失败的风险
- 新部署的软件功能出现故障的风险

保持系统高可用需要消除风险。但是当系统变得越来越复杂时，消除所有风险也变得越来越不现实。要保持一个大型系统的高可用，更多的是管理系统的风险，知道这些风险是什么，哪些风险是可接受的，以及你能够做什么来缓和风险。

我们将之称为风险管理。我们会在本书的第9章着重讨论风险管理，它是构建高可用系统的核心内容。

风险管理中的一个部分是风险缓和。风险缓和指的是当问题发生时，我们知道如何去尽可能降低问题所带来的影响。缓和意味着即使当服务和资源不可用时，依然尽可能确保你的系统以最好的、最完整的状态工作。风险缓和需要考虑哪些事情可能会出错，并且立即制订相应的计划，以便当问题发生时能够提供相应的解决方案。

风险缓和——一个没有搜索功能的网上商店

假设有一个售卖T恤的网上商店。它是一个很常见的在线商店，你可以在它的首页上浏览T恤，跳转到其他页面查看不同的T恤分类，并且可以搜索指定风格和类型的T恤。

为了实现搜索的功能，这类网上商店通常需要调用一个独立的搜索引擎，这可能是一个单独的服务，或者由第三方的搜索服务提供。

但是，因为搜索功能是一个独立的功能，所以你的系统就存在搜索服务不可用的风险。你的风险管理计划需要确定出该问题，并且将“搜索引擎失败”列为风险之一。

如果没有风险缓和计划，当搜索服务失败时，可能会产生一个错误页面，或者返回不正确或无效的结果——不管怎样，它都会带来很差的用户体验。

这个示例中的风险缓和计划可能是这样的：

我们知道最受欢迎的T恤是红色条纹T恤，60%访问网站的用户最终都停留在（并很可能最后会购买）这个商品上。因此，如果搜索服务停止了，我们可以显示一个“很抱歉”的页面，下方显示最受欢迎的T恤列表，其中就包括红色条纹T恤。这会鼓励遇到这个错误页面的用户，继续浏览别人曾经感兴趣的T恤。

此外，我们还可以显示一个“下一次购买享受10%折扣”的优惠券，这样就可以鼓励之前没有进行购买的用户，在搜索服务恢复正常后，继续回到我们的网站上进行购买。

上面的示例演示了什么是风险缓和，而确认风险、确定该如何处理风险，以及如何实现这些缓和措施的过程则被称为风险管理。

风险管理经常会暴露应用程序中未知的、需要立即修复的问题，还可以用来处理已知的故障问题，减少故障恢复时间或者降低严重性。

可用性和风险管理息息相关。构建一个高可用的系统，主要就是要考虑如何管理风险。

要点4：监控可用性

除非你看到问题发生，否则你不会知道应用程序中存在着问题。你应当确保对应用程序进行了适当的监控，以便可以从外部和内部两个视角来观察应用程序的运行状况。

监控的程度取决于应用程序的特点和要求，但是通常必须具备以下这些监控。

服务器监控

监控服务器的健康状况，并且确保它们始终在有效运行。

配置变化监控

监控系统配置的变化，以便确定它们对应用程序的影响。

应用程序性能监控

深入了解你的应用程序和服务，确保它们按照预期运行。

人为测试

从用户的角度来实时检测应用程序的运行情况，以便在用户真正发现问题之前发现它们。

报警

当问题发生时通知相关人员，以便使问题可以得到快速有效的解决，将对用户的影响降到最低。

如今市面上有许多非常优秀的监控系统，包括免费的和付费的服务。我个人推荐New Relic，它提供了之前提到的所有监控和报警能力。作为一款软件即服务（SaaS）的软件，它能够支持任何规模的应用系统的监控需求。

当你对应用程序和服务进行监控之后，就可以开始寻找它们的运行趋势。当你明确了一定的趋势之后，可以开始寻找一些异常值，将它们作为可能存在的可用性问题。你可以利用这些异常值，在系统发生故障之前通过监控工具来发送警报。除此之外，还可以在系统增长过程中时刻进行跟踪，确保可伸缩性计划的实施。

你应当为服务间通信建立内部的、私有的运行目标，并持续对它们进行监控。通过这种方式，当出现任何与性能或者可用性相关的问题时，你都可以快速诊断出哪个服务或者系统出现了问题，并定位问题的原因。此外，你可以发现一些“热点”——性能超出预期的地方，以及针对这些问题制订相应的开发计划。

要点5：以可预期及明确的方式来处理可用性问题

如果你对监控中所发生的问题置之不理，那么监控系统就毫无用处。这意味着当问题发生时必须发出报警，这样你才能有所行动。除此之外，你应当建立整个团队都遵循的流程，帮助诊断问题，并轻松修复常见的故障。

例如，如果某个系统无法响应，你可能会有一系列措施来解决。这其中可能包括运行一个测试来诊断问题原因，重启一个已知会导致系统无法响应的守护进程，或者当其他手段都失败时重启整个服务器。为常见的故障问题提供标准化流程可以减少系统不可用的时间。

此外，它们还可以提供更多有用的诊断信息，帮助工程师团队找到常见问题的根本原因。

当触发某个服务的报警时，该服务的负责人必须是第一个被通知到的。毕竟，他们负责修复自己服务中的所有故障。但是，其他与之紧密相关或依赖的团队也应当收到报警信息。例如，如果某个团队使用了一个特殊服务，他们希望知道该服务什么时候出现故障，从而在问题发生时能够更加主动地保证自己的系统不受影响。

这些标准的流程和办法应当被写进支持手册中，团队中每个负责人人手一份。这本支持手册还应该包含相关系统和服务的联系人列表，以及当无法用简单手段解决问题时，需要上报问题的联系人列表。现在有一些SaaS应用程序，可以自动化支持文档的管理和版本控制的工作，使得我们在事件进行中也可以使用它们。

所有这些流程、办法以及支持手册都应该提前准备好，以便当服务出现问题时，值班人员能准确知道如何在不同的情况下进行操作，以便快速恢复服务。这些流程和办法之所以非常有效，是因为故障通常都发生在一些不太恰当的时间点，例如午夜或者周末这些效率比较低的时间。这些建议可以帮助你的团队更聪明、更安全地将系统恢复到可运行状态。

做好准备

没人能够预测到可用性问题在什么地方、什么时间发生。但是你可以假设它们会发生，尤其是当你的系统面临越来越多的用户需求、变得越来越复杂的时候。提前做好处理可用性问题的准备，是降低问题出现概率和严重性的最佳方法。本章讨论的5个技巧，为保持应用程序的高可用性提供了可靠有效的手段。

[1] 根据Amazon CTO Werner Vogels所述，Amazon在2014年对个人主机进行了5000万次部署，平均一秒一次。

[2] 这个流程可以是但不一定必须是流行的持续集成和持续部署（CI/CD）流程。

第2章

两次失误的高度——预留从错误中恢复的空间

我想先和你分享一个我无意中听到的故事：

我们想知道——修改MySQL数据库上的一个设置会如何对性能产生影响，但是担心这个改动会导致生产数据库发生故障。因为我们不想弄垮生产数据库，所以决定将这个改动先应用到备份数据库上。毕竟，备份数据库并不是有人一直在使用。

听上去有点道理，是吧？不知道你之前有没有听到过这样的事情？

好吧，这里的问题在于数据库正在被人使用着。它目前的用途是为生产数据库提供备份，除此之外，它不应当被用于其他任何用途。

如你所见，出于实验性质的目的，备份数据库现在被用于测试不同的设置。这样带来的结果就是，随着这些设置生效，备份数据库与主生产数据库的差异开始越来越大。

直到，有一天，不可避免的事情发生了。

生产数据库出现了故障。

备份数据库一开始的确做了应该做的事——它接管了主数据库的工作，只是事实上它做不到。由于备份数据库上的设置已经与主数据库上的设置相差太多，导致它无法再正常处理原来主数据库上的数据。

于是备份数据库慢慢发生了故障，然后整个网站都无法工作了。

这是一个真实的故事。你有一个备份、复制的数据库，原本准备是在主数据库发生故障时，作为主数据库来使用的。但是，这个备份数据库没有受到像主数据库一样的尊重对待，并且也失去了它的主要能力——成为备份数据库的能力。

两次失误不会让事情变得正确，两次失误也无法抵消彼此，两次失误也无法自己修复。一次主数据库故障再加上一个管理不善的备份服务器，会让这一天变得糟糕。我们如何能避免这类可靠性问题呢？无线电控制飞机会告诉我们答案。

两次失误的高度

如果你曾经操作过无线电控制（R/C）飞机，可能听说过那句话“让你的飞机保持两次失误的高度”。当你学习如何操纵R/C飞机，尤其是开始学习如何进行特技飞行时，你会学得很快。失误其实就相当于高度，出现一个失误，你就失去一定的高度。当你失去太多高度时，坏事就发生了。因此，让你的飞机保持“两次失误的高度”意味着让你的飞机飞得足够高，从而有足够的高度从两个不相关的失误中恢复飞行。

你可以想象一下：在恢复飞行的过程中，你通常压力很大，而且可能处于一种惊恐的情绪中，很可能会做出一些反常的事情——正是这种情形让你可能产生另一个失误。因此如果你飞得不够高，就会坠机。

换一种角度说，如果你能够在两次失误的高度飞行，即使发生了一次失误，也总有一次能够从失误中恢复的机会。

同样的思想，对于我们创建高可用、大规模的应用程序是非常重要的。

我们如何在应用程序中“保持两次失误的高度”？对于初学者而言，当确定出系统要面对的失败场景后，我们遍历所有可能存在的分支场景，并制订相应的恢复计划。需要确定恢复计划本身没有错误或者缺陷——简而言之，需要检查恢复计划是否可以正常工作。如果发现它不能正常工作，那么就应当重新制订恢复计划。

这只是一个可能会用到“两次失误的高度”的场景，当然还有很多其他的场景，我们会通过一些示例来说明它们对应用程序的影响。

场景1：丢失一个节点

我们来看一个跟Web服务流量有关的示例场景。

假设你正在开发一个预计每秒处理1000个请求（请求/秒）的服务，而且我们假设服务中的单个节点每秒只能处理300个请求。

问题：你需要多少个节点才能支撑该流量？

通过一些简单的数学计算我们可以得到结果：

所需的节点数=请求总数/每个节点能处理的请求数量

其中：

所需的节点数

表示处理指定数量请求所需的节点数量。

请求总数

该服务预期处理的请求总数。

每个节点能处理的请求数量

服务中每个节点能处理的请求数量的平均值。

代入我们的数字之后：

所需节点数量=1000/300≈3.3

所需节点数量=4个节点

你需要4个节点来处理每秒1000个请求的服务压力。转换一下，当使用4个节点时，每个节点需要处理：

每个节点处理的请求数量=请求总数/节点数量

每个节点处理的请求数量=1000/4=250

每个节点需要每秒处理250个请求，这低于每秒300个请求的单个节点限制，如图2-1所示。



图2-1：4个节点，每个节点每秒处理250个请求

现在你有了4个节点。你不仅可以处理预计的流量，并且因为有了4个节点，可以允许丢失掉1个节点。你已经做好丢失1个节点的准备了，是不是？真的吗？

当然，实际上你并没有。如果你在流量峰值时丢失1个节点，你的服务仍然会失败。为什么？因为如果丢失1个节点，你的流量会分布在剩余3个节点上。此时：

每个节点处理的请求数量=请求总数/节点数量

每个节点处理的请求数量=1000/3=333

每个节点需要每秒处理333个请求，这超过了每秒300个请求的节点限制，如图2-2所示。

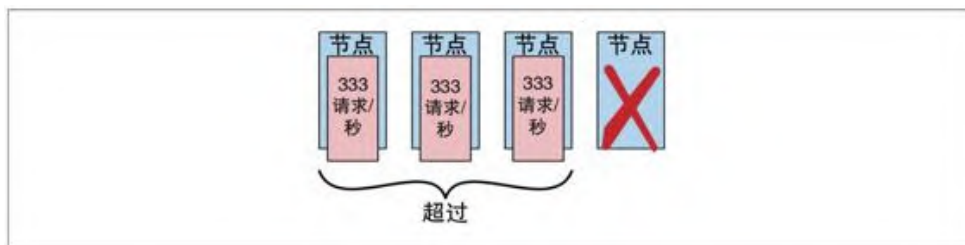


图2-2: 4个节点，一旦某个节点出现故障就会影响到其余3个节点

因为每个节点每秒只能处理300个请求，所以服务器现在已经过载。在当前情况下，或者给所有用户提供较差的性能，或者会丢掉某些请求，或者以其他形式无法提供服务。总之，你的可用性开始下降了。

如你在图2-2中所见，如果系统失去了其中的1个节点，就无法继续提供完整的能力。因此，即使你以为能够从1个节点故障中恢复，但实际上却不能。你的系统现在是脆弱的。

为了能够处理节点故障，你需要4个以上的节点。如果你希望能够处理1个节点故障，那么就需要5个节点。这样，即使5个节点中的其中1个出现故障，还剩下4个节点可以处理流量：

每个节点处理的请求数量=请求总数/节点数量

每个节点处理的请求数量=1000/4=250

如图2-3所示，因为这个值低于每个节点每秒300个请求的限制，所以它们有足够的能力来继续处理流量，即使是1个节点出现了故障，也不会对系统造成任何影响。

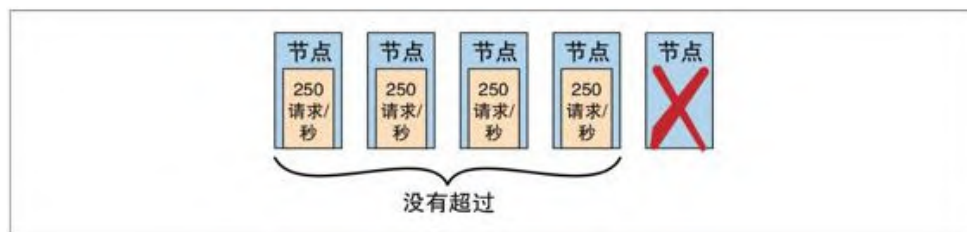


图2-3: 5个节点，1个节点出现故障不会造成其他影响

场景2：升级过程中出现的问题

另一个“两次失误的高度”的例子是升级中的应用程序。升级和日常维护可能会导致没有预期到的可用性问题的。

假设你有一个平均流量为每秒1000个请求的服务。此外，我们假定服务中单个节点的处理上限是每秒300个请求。如之前的示例所述，最少需要4个节点来运行服务。为了能够处理预期流量并支持单节点故障，你为服务配置了5个节点。

现在，假设你希望对正在运行的服务进行一次软件升级。为了在升级过程中保证服务正常运行，你决定使用滚动部署的方式。

简而言之，滚动部署就是每次只升级1个节点（临时将它设置为离线状态来进行升级）。在成功升级第一个节点并重新处理流量之后，继续升级第二个节点（临时将它设置为离线）。持续这个过程直到5个节点都完成升级。

因为在每次升级时，只有1个节点是离线状态，所以总是有4个节点在处理流量。因为4个节点已经足够处理所有的流量，所以升级过程中服务不会受到影响。

这是一个很不错的计划。你已经构建了一个不仅能处理单节点失败，还可以通过滚动部署实现不停机升级的系统。

如果在升级过程中某个节点发生了故障呢？这个时候，你有1个节点因为升级不可用，还有1个节点出现故障，这样只剩下3个节点，不足以处理所有的流量。这时，你就会遇到服务降级或者系统整体不可用的情况。

在升级时遇到某个节点故障的可能性有多大呢？

你遇到过多少次升级失败呢？事实上，升级过程中一个参数带来的节点故障概率可能比其他时候大得多。升级和节点故障并不是完全孤立的。



我们得到的教训是：即使你认为有冗余节点来处理各种故障情况，但是如果两个或多个问题同时发生（因为问题都是有关联性的），就可能会根本没有任何冗余。这样的系统就容易出现可用性的问题。

因此，要想使用每秒300个请求的节点来处理每秒1000个请求的流量，我们需要：

4个节点

可以处理流量，但是不能处理单个节点故障。

5个节点

可以处理单个节点故障，或者在维护或升级时允许单个节点不可用。

6个节点

可以处理多个节点故障，或者在维护或升级时，允许同时存在1个节点升级失败和1个节点不可用。

场景3：数据中心恢复

我们将这个问题扩大一点，来看一看数据中心的冗余和恢复。

假设你的服务正在处理每秒10,000个请求的流量。因为单个节点每秒只能处理300个请求，所以这意味着你需要34个节点，这还不考虑故障冗余和升级的情况。

为了让系统增加一些额外的处理能力，我们使用了总共40个节点（每个节点每秒处理250个请求）。现在即使失去多达6个节点也可以处理所有的流量。

让我们来做得更好一点：现在我们把这40个节点平均分布到4个数据中心，这样可以有更好的冗余性。

现在，我们可以像恢复节点故障一样来恢复数据中心故障了，如图2-4所示。但是现在就算是恢复了吗？

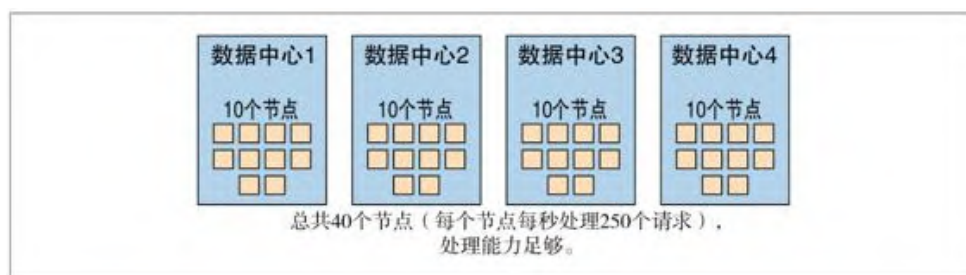


图2-4：4个数据中心，40个节点，有足够的处理能力来处理请求

答案是否定的。显然，我们可以处理单个节点故障，因为我们已经提供了额外的6个（40-34）节点。但是如果某个数据中心出现故障了怎么办？

如果某个数据中心出现故障，我们就丢失了四分之一的服务器。在这个例子中，我们就从40个节点减少到了30个节点。此时每个节点不再每秒只需处理250个请求了，而是需要每秒处理333个请求，如图2-5所示。因为这超过了单个节点的处理能力，所以我们又遇到了可用性的问题。

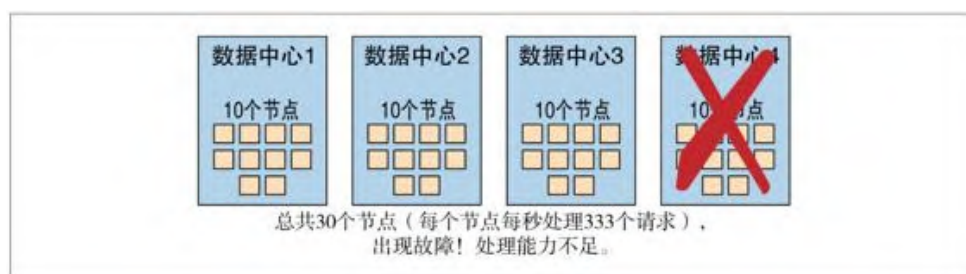


图2-5：4个数据中心，1个出现故障，只剩30个节点，处理请求的能力不足

即使我们使用多个数据中心，但是一旦某个数据中心出现故障，就会让我们无法再处理增长的流量。我们认为自己可以从某个数据中心的故障中恢复，但实际上不能。

因此，你究竟需要多少台服务器

究竟需要多少台服务器才能处理数据中心出现故障的情况呢？我们一起来解答这个问题。

依然基于相同的假设，即我们需要最少34台服务器才能处理所有流量。如果我们使用4个数据中心，究竟需要多少台服务器才能真正满足数据中心冗余呢？

显然，即使4个数据中心其中之一出现故障，我们需要确保任何时候都拥有34台正常工作的服务器。这意味着我们需要有34台服务器遍布在其他3个数据中心中：

每个数据中心的节点数=服务器最小数量/（数据中心数量-1）

每个数据中心的节点数=34/（4-1）

每个数据中心的节点数 $\approx 11.333 \approx 12$ 台服务器/数据中心

因为我们需要在每个数据中心有12台服务器，并且即使4个数据中心之一出现故障，我们在每个数据中心仍然有12台服务器，所以：

总节点数=每个数据中心的节点数 $\times 4=48$

因此，我们需要48个节点来保证，即使有1个数据中心出现了故障，依然有34个节点能够工作。

如果数据中心的数量发生了变化，会对我们的计算造成什么影响呢？如果我们只有2个数据中心呢？如之前一样：

每个数据中心的节点数=服务器最小数量/（数据中心数量-1）

每个数据中心的节点数=34/（2-1）

每个数据中心的节点数=34

总节点数=每个数据中心的节点数 $\times 2=68$

如果有2个数据中心，那么需要68个节点。如果拥有4个数据中心，那么需要48个节点来保证数据中心冗余。如果拥有6个数据中心，那么需要42个节点来保证数据中心冗余。

注意，随着数据中心数量的增加，所需节点的数量也在减少。这表明了一个看似奇怪的结论：

为了提供整个数据中心故障恢复的能力，当你拥有的数据中心越多时，每个数据中心需要的节点越少。

这似乎是一种倒退，与我们的直觉相差甚远。从这个例子中，我们得到的教训是：虽然其中的内容可能无法直接应用到实际情况中，但是理论依然适用。当你在设计恢复计划时一定要小心，你的直觉可能会与实际情况相背离，如果你的直觉是错误的，那么就会带来可用性的问题。如果你在前面的示例中仅凭直觉，那么会认为在任何情况下都不会有足够的节点来处理数据中心故障，或者最终节点的数量会超出所需的数量，从而无法获得所需的弹性级别。

场景4：隐蔽的共享故障类型

有些时候，很多问题看上去都是独立的，不太可能同时发生，但事实上它们却是互相关联的。这意味着在某些场景下，多个故障可能会一起出现。

假设你的服务运行在4个节点上。你为了做好充分准备，使用了一共6个节点——足够同时处理单节点故障和升级失败。

你现在放心地认为系统是安全的。

随后发生了一件事：在数据中心的机房中，某个机架上的供电系统出现了问题，导致整个机架无法工作。

通常在这个时候，你会意识到所有6个服务器都放置在同一个机架上。你是如何发现的呢？因为所有6台服务器都下线了，同时你的服务也完全无法提供了。

所以也别提什么冗余了……

当你认为自己的系统是安全的时候，可能实际上并不是这样的。我们知道不是所有问题都是完全独立的。在你的所有服务器之间，普遍存在着可能看不到或者至少是没发现的情况，那就是它们都共享着相同的机架和供电系统。

请确保仔细检查了隐藏的共享故障因素，它们可能会导致你精心准备的计划付之东流，为系统带来可用性风险。

场景5：故障循环

故障循环指的是，当某个特定问题导致系统故障时，为了修复它，你需要或者不得不制造另一个更严重的问题。

解释故障循环最好的方式是用下面这个跟服务器无关的例子。假设你生活在一个非常棒的公寓中，有一个封闭的车库来存放东西。哇！这真不错。但是这里经常停电，于是你决定买一台发电机，以备不时之需。你买了发电机、汽油，然后把它们放在车库中。生活看上去很美好。

然后，当停电时，你准备去车库找你的发电机。这时候你第一次意识到，唯一进入车库的方式就是通过车库的电子门—但是因为停电它无法打开。

哎呀！

这正是因为虽然你制订了备选计划，但并不意味着在需要时能执行它。

同样的问题也会出现在我们的服务上。一次服务故障会不会因为导致其他看起来无关的问题，从而让我们难以修复？例如，你的服务出现故障，部署一个新的版本有多简单？如果你用于部署的服务失败会怎样？如果你用来监控其他服务性能的服务失败了会怎样？

请确保你的恢复计划在问题发生时能够实现。如果不能考虑问题间的依赖关系以及相应的解决方案，你就会面临可用性的问题。

管理你的应用程序

“两次失误的高度”意味着不仅要看到表面的问题，还要往深层次了解。你需要确保不存在互相依赖的问题，并且准备的恢复机制能够在问题发生时真的帮你恢复系统。

此外，请不要忽视问题。问题不会自己消失，并且它们会根据你的可用性计划发生变化。因为即使出现故障的是备份数据库，也并不意味着可以不管它。对待你的备份和冗余系统，应该像对待主系统一样认真，毕竟它们的重要性是一样的。

我经常跟朋友们说，“如果它跟生产环境有接触，那它就是生产环境”。不要认为生产环境中的任何事都是稳定可靠的。

做到这一点很难。我们很难知道什么时候会出现不同级别或相互依赖的故障。你应当多花一些时间来观察系统的情况并解决它们。

航天飞机

让我们用一个独立、冗余、能够恢复多级错误的系统示例来结束本章。事实上，它也是第一个将冗余和故障管理思想发挥到极致的大规模软件系统。它让宇航员可以将生命托付给它。

没错，我指的就是美国航天飞机计划。

航天飞机计划有一些重大且严重的机械问题，但是我们不会在这里来讨论它们。我们要说的是航天飞机中的软件系统，以及它如何将冗余和独立的错误恢复发挥到极致。

航天飞机的主计算机系统由5台计算机组成。其中4台计算机中运行的软件和硬件都一样，但是第5台则不同。我们稍后会来讨论这一点。

在执行任务的关键部分（例如，发射和着陆）时，这4台主计算机都会运行一样的程序。这4台计算机中的数据和软件也是一模一样的，因此它们计算出的结果也应该是一样的。所有这4台计算机执行一样的计算，并不断地与其他计算机比较结果。如果，在某一时刻，任意一台计算机计算出了一个不同的结果，这4台计算机投票选举哪一个结果是正确的。在使用获得选举的结果后，产生错误的计算机会在整个飞行过程中被关闭。航天飞机可以在只有3台计算机的情况下安全飞行，并且可以在只有两台计算机的情况下安全着陆。

这就相当于民主制度的最终形态。胜者统治，败者终结。

但是如果4台计算机无法达成一致呢？这可能出现在多次失败以及多台计算机关机的情况下。或者，由于某个严重的软件问题同一时间影响了4台计算机（毕竟这4台计算机运行着完全一样的软件）。

这时就需要第5台计算机来发挥作用了。通常它都处于空闲状态，但是如果有需要，它可以执行与其他4台计算机一样的计算。关键在于它上面运行的软件。第5台计算机上运行的软件，是由一个完全独立的开发小组开发的简化版本。在理论上，它不会与主软件存在一样的软件错误。

因此，如果4台计算机上的主软件之间不能就结果达成一致，它们会将决定最终结果的权利交给第5台完全独立的计算机。

这就是一个从上层角度隔离可能问题的高冗余、高可用的系统。

在航天飞机运行的30年中，计划从来没有在执行过程中由于软件或者计算机的故障而遇到危及生命的严重问题——即使它是当时空间计划使用的最复杂的软件系统。

第 II 部分

原则2. 现代化应用程序架构：使用服务

现代化的软件需要使用现代化的应用程序架构。现代化的应用程序架构要求远离单体应用程序，而采用基于服务的架构。

无论是从伸缩流量的角度，还是从伸缩组织处理应用程序能力的角度来看，单体应用程序都很难进行伸缩。单体程度越大，更改应用程序的速度就越慢，能够处理和有效管理它的人员就越少，流量变化和流量增长对可用性产生负面影响的可能性就越大。

面向服务的架构通过在流量伸缩方面提供更大的灵活性来解决这些问题。此外，它们还提供了一个可伸缩的框架，允许大型开发团队能够处理应用程序，使得应用程序本身可以变得更大、更复杂。

第3章 使用服务

现代化的软件需要使用现代化的应用程序架构，但是现代化的软件架构都包含什么呢？设计可伸缩和高可用应用程序的关键之一，是采用基于服务或微服务的架构。传统的单体应用程序开发流程不具有让应用程序保持可伸缩性和高可用性的能力。

通常来说，应用程序都是一个独立、大型、明确的单体程序。这个独立的单体程序包含了一个应用程序的所有业务逻辑。为了改进某一项业务功能，一个开发人员必须在这个应用程序的内部进行修改，同时，其他所有开发人员也必须在其中进行修改。这样，开发人员之间很容易造成干扰，并且互相冲突的修改会导致更多的问题和故障。

在一个面向服务的架构中，单个服务只包含所有业务逻辑的某个子集。这些独立的服务互相连接，为应用程序提供完整的业务逻辑。

让我们来比较一下单体架构和面向服务的架构，看看为什么面向服务的架构能够提供更好的组织可伸缩性和应用程序可伸缩性。

单体应用程序与面向服务的应用程序

传统的大型单体应用程序在一个组件中包含了所有逻辑和功能，各个代码片段之间相互交叉并依赖。它是一个编译后的源代码片段，创建了一个包含应用程序大部分或者所有功能的可执行文件。图3-1表示了一个典型的单体应用程序。

这也是大多数发展成单体应用程序的系统看上去的样子。在图3-1中，你可以看到5个独立的开发团队在相互重叠的部分同时工作。我们无法知道在某一个时间点，究竟是谁在哪一块上工作，也不难想象代

码变更所导致的冲突和问题。代码质量，以及因此产生的系统质量和可用性也会受到影响。此外，系统变得越来越复杂，单个开发团队想要更改代码也变得越来越困难，不得不面对与其他团队的协作、互相冲突的改动，以及组织内部纠缠不清等问题。

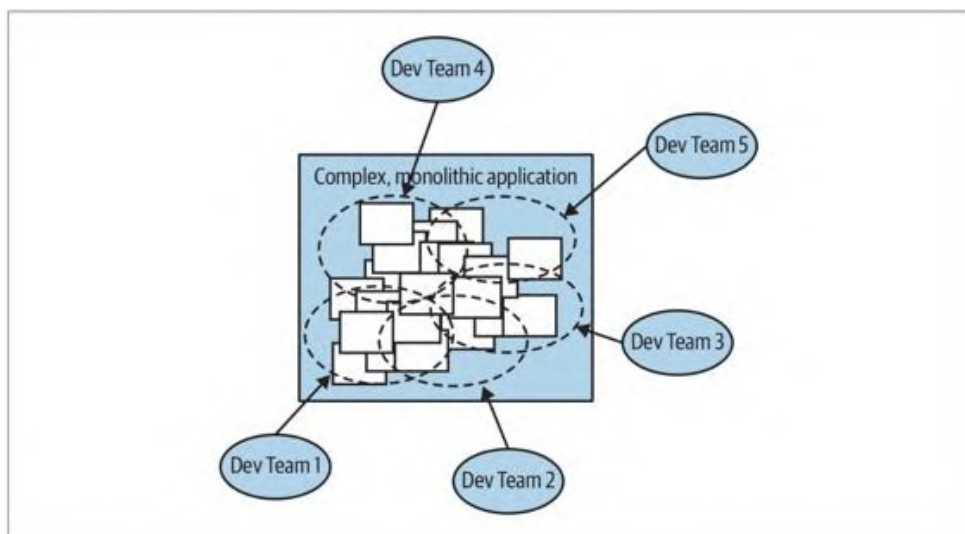


图3-1：一个大型的、复杂的单体应用程序

图3-2展示了由一系列服务所组成的同样的应用程序。每个服务都有一个清晰的负责人，并且每个团队都有一个清晰的、无重合的职责范围。

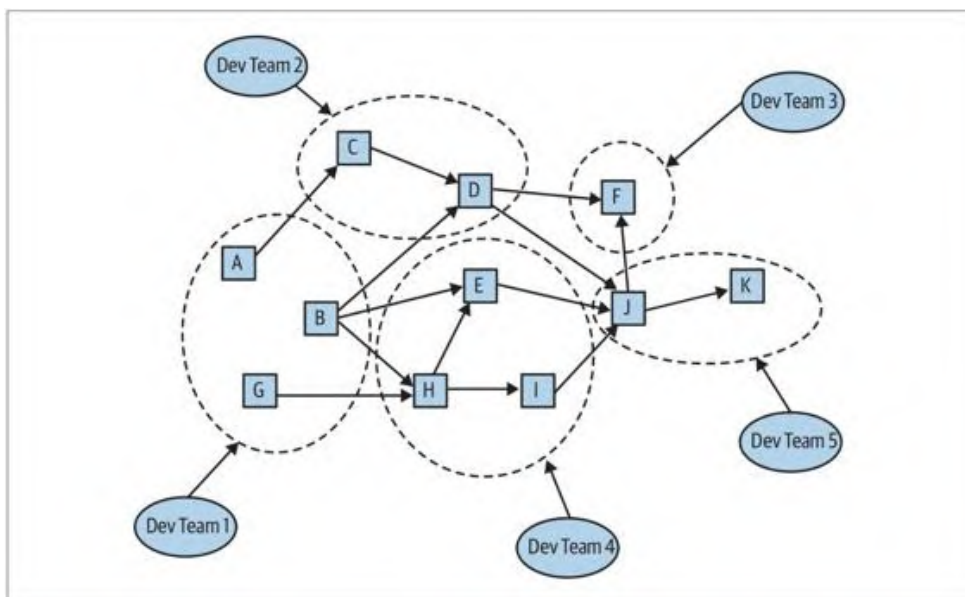


图3-2：一个大型的、复杂的但基于服务的应用程序

面向服务的架构能够将一个应用程序拆分成多个不同的领域，每个领域由单独的团队来负责管理。这种职责隔离对于构建大规模应用程序至关重要，可使各个服务之间独立地完成工作，避免影响到系统中其他组中开发人员的工作。

当构建高可伸缩的应用程序时，基于服务的应用程序提供了以下几点好处。

伸缩性决定

这使你能够从更细粒度来考虑伸缩性方面的决定，从而进行更有效的系统和组织优化。

团队分配和关注

面向服务的架构使你能够将任务分配给不同的团队，让它们关注于系统中不同的可伸缩和高可用需求，并让它们有信心知道，自己的决定会对整个系统产生合理的影响。

复杂的本地化

使用基于服务的架构，你可以将各个服务看作一个个黑盒，只有服务的所有者需要了解该服务内部的复杂逻辑，其他开发人员只需要知道服务所能提供的能力，而不需要知道它内部的工作原理。这种认知和复杂性上的隔离，能帮助你创建更大型的应用程序，并且能更有效地管理它们。

测试

基于服务的架构比单体应用程序更容易测试，从而可以增加系统的可靠性。

但是，如果服务之间的边界没有设计好，面向服务的架构会增加系统整体的复杂性。这种复杂性会导致更低的可伸缩性，并降低系统的可用性。因此，选择合适的服务和服务边界至关重要。

所有权收益

让我们来看一对服务。

在图3-3中，我们可以看到由两个独立团队所管理的两个服务。“左服务”正在使用由“右服务”提供的能力。

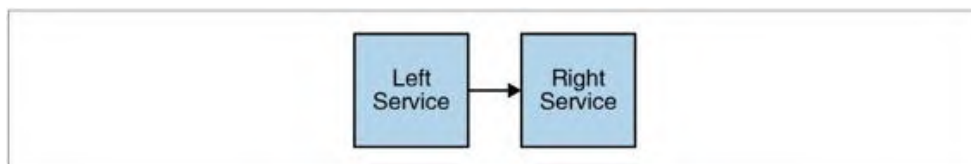


图3-3：一对服务

我们先从左服务所有者的角度来看这张图。显然，左服务的团队需要了解左服务的整体结构、复杂性、正确性、交互操作、代码以及其他内容。但是它们需要了解右服务的什么呢？首先，左服务团队需要了解右服务的以下几件事情：

- 该服务提供的能力。
- 如何调用那些能力（API语法）。
- 调用那些能力的意义和结果（API语义）。

这些是左服务需要了解的基本信息。那它们不需要了解右服务的哪些信息呢？这包含很多内容，例如：

- 左服务不需要了解右服务是一个单独的服务，还是多个子服务的组合。
- 左服务不需要了解右服务需要依赖其他什么服务。
- 左服务不需要了解右服务是用什么语言编写的。
- 左服务不需要了解右服务所使用的硬件或系统架构。
- 左服务甚至不需要了解谁来提供右服务（但是，它们需要知道遇到问题时联系谁）。

如图3-4所示，右服务可以很复杂，也可以很简单。但是对于左服务的团队成员来说，右服务就像一个黑盒，不知道其内部的结构是什么样的，如图3-5所示。只要左服务知道这个黑盒的接口是什么（API），就可以使用这个黑盒所提供的能力。

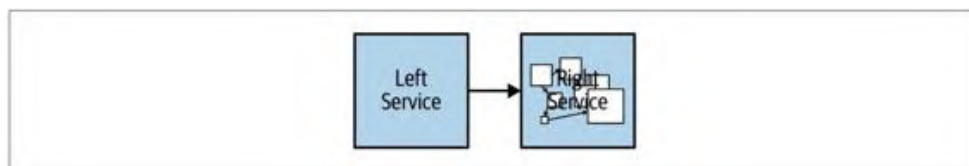


图3-4：右服务的内部情况

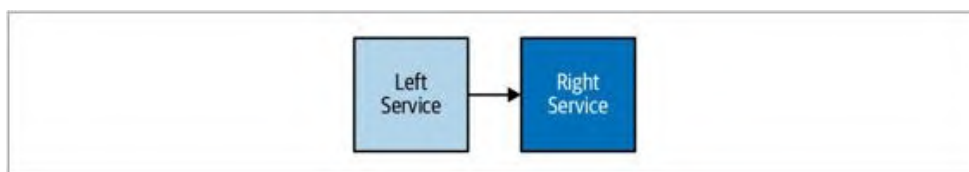


图3-5：右服务的复杂性被隐藏起来

出于管理的目的，左服务必须能够依赖右服务所提供的约定。这个约定描述了左服务如何使用右服务的所有事项。

约定包含以下两部分。

服务的能力（API）

服务的用途。

如何调用服务以及每个调用的意义。

服务的响应性

API使用的频率

什么时候可以使用API

API响应的速度

API是否存在依赖

所有这些信息组成了右服务提供给左服务的约定。只要右服务遵守这个约定，左服务就不需要知道或者关心右服务是如何来实现这些约定的。

该约定的响应性部分被称为服务等级协议，或者SLA。它是让左服务能够放心使用右服务，而不用关心右服务如何实现的关键指标。我们将在第8章重点讨论SLA。

通过让每个服务都有清晰的所有权，团队可以只关注于它们所负责的模块，以及依赖服务所提供的API约定。这种职责隔离使得组织能够更容易容纳更多的团队，因为团队之间的耦合程度更加松散，所以团队之间的距离（不管是组织上的层级还是物理上的距离）也就无关紧要。只要双方保持着一致的约定，你就可以扩展组织的规模，创建出更大型、更复杂的应用程序。

规模收益

应用程序的不同部分在伸缩性上有着不同的需求。生成首页的组件比生成用户设置页面的组件会更频繁地被使用到。

通过在服务之间使用清晰的API和API约定，你可以独立决定和实现每个服务的伸缩性需求。这意味着如果首页是最经常被访问的页面，相对于管理用户设置页面的服务来说，你可以提供更多的硬件资源给生成首页的服务。

通过独立管理每个服务的伸缩性需求，可以实现以下事情：

- 通过与相关功能负责团队的密切合作，提供更加准确的可伸缩性。
- 将系统资源留给真正需要伸缩性的组件。
- 将伸缩性的决定权交回团队，因为它们才是最了解服务需求的人（服务所有者）。

基于服务的架构使得组织和应用程序变得更容易伸缩，并且能够伸缩到更大的规模。在下一章中，我们将更详细地介绍服务。

拆分服务

一个服务提供了应用程序中其他服务所需要的某些能力。例如，账单服务（提供了用户开具账单功能的组件）、账户创建服务（管理创建账户的组件），以及通知服务（包含了通知用户事件和使用情况的功能）。

每个服务都是一个独立的组件。这里的“独立”很重要，独立的服务必须满足以下几个条件。

维护自己的代码库

服务拥有自己的代码库，且区别于其他服务的代码库。

管理自己的数据

服务需要管理其自身数据的状态。服务之间唯一能访问到数据的方式就是通过定义的API，其他服务不能直接接触当前服务的数据或者状态信息。

向其他服务提供能力

服务有一个定义良好的能力集合，并将这些能力提供给系统中的其他服务。换句话说，它提供了一个API。

消费其他服务的能力

服务按照标准的、约定的方式，使用其他服务提供的能力。换句话说，它使用其他服务的API。

单一所有者

每个服务只能由一个开发团队负责和维护。虽然一个团队可以负责和维护多个服务，但是一个服务只能由一个团队负责和维护。

如何定义服务

你如何来决定应该将系统中的哪些部分独立出来形成自己的服务呢？

这是一个好问题，但没有唯一的答案。某些公司喜欢“服务化”，因此将系统拆分成很多（成百上千）个非常小的微服务。另一些公司则将系统拆分成一些较大的服务。这个问题并没有谁对谁错。但是，整个行业在向着更小的微服务的趋势发展。像Docker和Kubernetes等技术通过提供管理大量小型服务的基础设施，使得这些大量的微服务成为一种可能的系统拓扑。



我们在本书中将交替使用服务和微服务这两个名词。

深入了解服务

那么如何来决定服务的边界呢？公司组织、文化以及应用程序的类型都会在很大程度上影响服务边界的确定。

以下是一组你可以用来确定服务边界的指导原则。不过，这些只是原则，并不是规则，也很可能随着工业进程的发展而发生改变，但是它们可以帮助我们思考什么是服务以及如何划分服务。

以下是较高层面上的指导原则（按照先后顺序）。

特定的业务需求

是否有特定的业务需求（例如，会计、安全或者监管）会影响服务的边界？

清晰和独立的团队所有权

负责该功能的团队是否清晰和独立（例如，在另一个城市，或者在另一个楼层，或者甚至只是一个不同的经理），是否会帮助确定服务边界？

天然隔离的数据

服务所管理的数据是否天然地与系统其他数据相独立？将数据放到一个单独的数据存储器中是否会对系统造成过大压力？

共享的能力/数据

服务是否提供了一些被其他服务使用的共享能力，这些共享能力是否需要共享数据？

我们现在分别来解释以上每一项原则。

指导原则1：特定的业务需求

在某些情况下，某些特定的业务需求会决定一个服务的边界。这些需求可能是监管、法律、安全需要或者一些关键业务的需求。

假设你的系统接受来自用户的在线信用卡支付。但是你应该如何收集、处理并且存储这些信用卡，以及它们所代表的支付呢？一个好的商业策略是将处理信用卡的过程交给另外一个服务，与系统中的其他服务区分开。

将关键业务逻辑放在独立的服务中，可以实现有效的分隔。例如，对于信用卡业务来说，有以下几点原因：

法律/监管需求

法律和监管需求会要求你在保存信用卡的方式上，选择与其他业务逻辑或其他业务数据不同的方式。将信用卡处理业务拆分成独立的服务，使我们能更容易地处理这些数据。

安全

出于安全的考虑，你可能需要额外的防火墙来保护这些服务器。

校验

你可能需要对这些能力进行更加严格的产品测试，以保证它们的安全性。

限制性访问

你通常会希望限制对这些服务器的访问，这样只有必要人员才能访问高度敏感的支付信息，例如，信用卡。你通常不希望或者不需要将这些访问权限提供给所有的工程人员。

理解关键业务逻辑的需求，是决定服务边界应该位于何处的一个重要考虑因素。

指导原则2：清晰和独立的团队所有权

应用程序正在变得越来越复杂，需要越来越多的开发人员进行开发，通常也提供更专业的功能。随着开发人员、团队和开发地点的增加，团队之间的协作变得越来越难。

服务能够将更小的、清晰的、独立的模块所有权赋予不同的团队。



一个单独的服务应该由一个3~8人组成的开发团队来负责和运行。这个团队应该负责该服务的所有方面。

这样就减少了团队之间的依赖，使得独立的团队更容易运行系统和实现创新。

如之前所述，单个服务应该只由单个团队负责和运行。关键是要确定这个团队可以负责该服务的所有方面。这意味着这个团队要负责服务所有的开发、测试、部署、性能以及可用性等各个方面。

根据服务的复杂性和功能不同，一个团队也可以管理多个服务。除此之外，如果多个服务本质上是类似的，那么让一个团队来管理所有这些服务可能会更容易一些。

出于安全原因隔离团队

有些时候，你希望限制能够访问某个服务代码和数据的人员数量和范围。这对于需要审计或者存在法律约束的服务来说尤其重要，例

如，之前所讨论的信用卡支付处理。限制对含有敏感数据服务的访问，可以降低数据暴露所导致的问题。像这种情况，你可以从物理上限制只有关键人员才能访问服务的相关代码、数据和系统。

除此之外，将敏感数据分成两个或多个服务，每个服务由不同的团队管理，也可以有效降低多个服务同时泄露数据的风险。

出于安全原因隔离数据

当你在处理信用卡支付时，信用卡号码本身可以被存储在一个服务中。而使用信用卡所必需的信息（例如，账单地址和CCV代码）可以存储在第二个服务中。通过将该信息分开保存在两个服务中，每个部分都由单独的团队负责，你可以避免任何员工无意或有意地将信用卡信息泄露出去，造成不当影响。

你甚至可以选择不由自己来保存信用卡号码，而是将它们存储在一个第三方信用卡服务公司的服务中。这可以保证即使你的其中一个服务被入侵，信用卡本身的数据也不会被泄露。

指导原则3：天然隔离的数据

服务的一个要求是，其托管状态和数据需要与其他数据相隔离。出于多种原因，让多个独立的代码库对同一组数据进行操作是有问题的。只有在你隔离数据之后，隔离代码和所有权才能起作用。

图3-6展示了一个服务（服务A）试图访问存储在另一个服务（服务B）中的数据。它说明了服务A访问服务B中数据的正确方式，即服务A通过API调用来访问服务B，然后让服务B访问自己数据库中的数据。

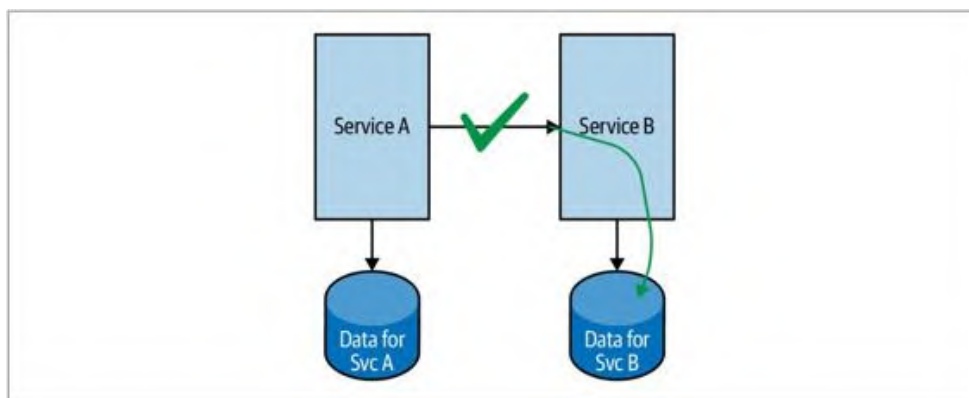


图3-6：共享数据的正确方式

如图3-7所示，如果服务A试图直接访问服务B中的数据，而不是通过服务B的API，那么所有问题都会出现。这种数据集成会导致服务A和服务B过度耦合，并且维护数据和迁移模式时会出现问题。一般来说，如果服务A越过服务B的业务逻辑，直接访问服务B的数据，会导致严重的数据版本不一致和数据损坏问题。这种情况应该被严格禁止。

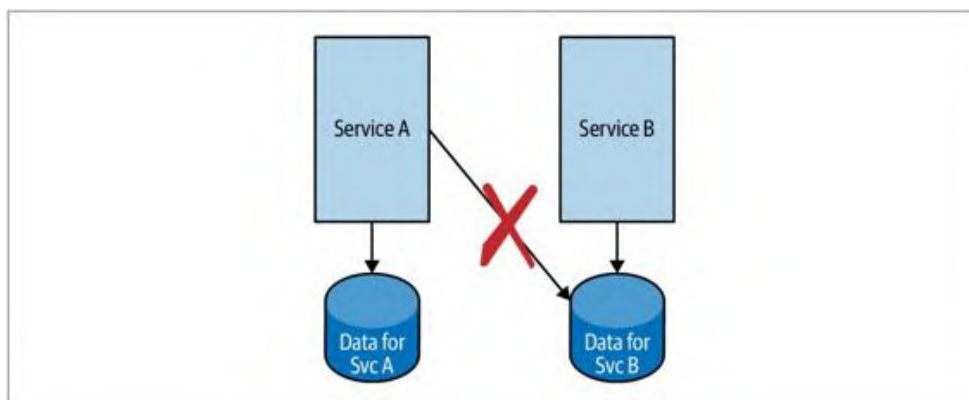


图3-7：共享数据的错误方式

如你所见，确定数据划分界线是确定服务划分界线的关键环节。对于一个服务来说，它是否能够负责自己的数据，并通过外部服务接口来提供数据访问呢？如果答案是“是”，说明这是一个很好的服务边界。如果答案是“否”，说明这不是一个很好的服务边界。

一个服务如果需要操作另一个服务的数据，必须通过数据所在服务的公开接口来访问。

指导原则4：共享的能力/数据

有些时候我们可以很容易地创建一个服务，因为它负责提供一系列的能力和/或数据。这些能力和数据可能需要在很多其他服务之间共享。

说明该原则最好的一个示例就是用户身份服务，它只是简单地提供系统中指定用户的信息，如图3-8所示。

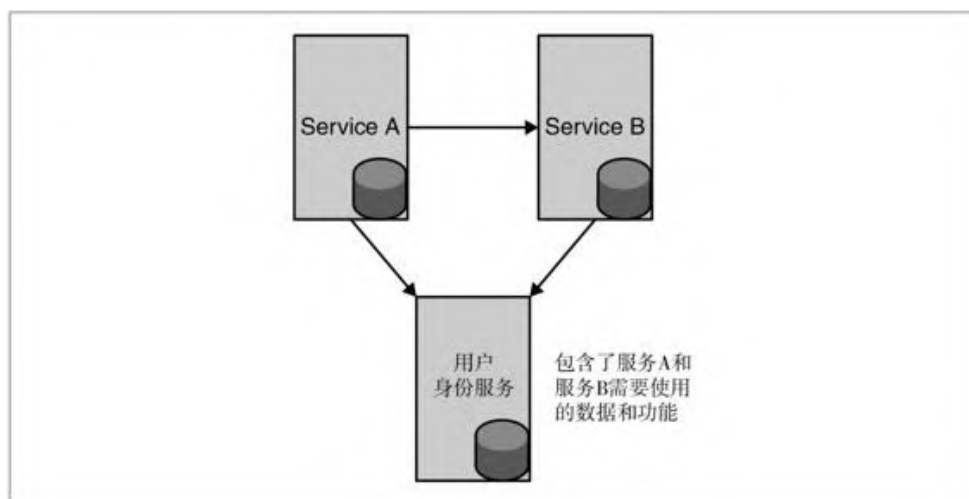


图3-8：通过服务与其他服务共享通用数据

这个数据服务的业务逻辑也许不复杂，但是它最终负责所有用户的常用信息。这些信息通常被大量其他的服务所使用。

通过一个中心化的服务来提供和管理这类信息，在实际中非常实用。

多种原因

之前的指导原则大概描述了决定服务边界的基本标准。但是，通常你需要综合考虑多种原因才能最终做出决定。

例如，从数据所有权和共享能力的角度来看，只有一个用户身份服务是合理的，但是从团队所有权角度来看，可能就不一定了。保存

用户身份的数据存储在某个数据库中可能是合理的，但是可能存储在一个或多个单独的服务中会更好。

举例说明，一个用户的个人资料中通常会有搜索选项，但是很少被搜索以外的功能使用。因此，这个数据可能更适合存储在一个搜索身份服务中，而不是用户身份服务中。这也可能是出于数据复杂性甚至是性能原因的考虑。

最终，你必须同时运用你的判断力和之前的标准。当然，你还必须考虑公司的业务逻辑和需要，以及特定的业务需求。

过犹不及

但是，通常你会将应用程序过度拆分成太多服务。之前的标准可能会被滥用于创建服务，从而导致创建出过多的服务。

例如，除了提供一个简单的用户身份服务之外，你还可能将它再拆分成多个更小的服务，例如：

- 用户名称服务
- 用户地址管理服务
- 用户邮件地址管理服务
- 用户祖籍管理服务

这样做很可能有点拆分过度了。

将服务拆分成过细粒度会带来一些问题，包括整体的系统性能影响。但是从最基本的角度来说，每当要将一个功能拆分成多个服务时，你都会做以下几件事：

- 降低单个服务的复杂性（通常来说）。
- 增加系统整体的复杂性。

体积小的服务通常会降低单个服务的复杂性，但是，你拥有的服务越多，需要交互的独立服务也越多，从而导致整体系统架构变得越来越复杂。

如果一个系统中存在过多的服务，可能会带来以下这些问题。

影响大局观

想要记住整个系统架构变得越来越难，因为系统变得越来越复杂。

更大故障概率

更多独立组件之间需要相互协作，导致故障发生的概率更大。

难以修改服务

每个独立服务的消费者都越来越多，增加了修改服务可能对使用者造成影响的可能。

更多依赖

每个独立服务都越来越依赖于其他的服务。更多的依赖意味着发生更多问题的可能。

这些问题都可以通过在服务间定义清晰的接口边界来缓解，但这不是一个完美的解决方案。重要的一点是，我们需要在服务数量和服务体积之间找到一个适当的平衡点。

找到适当的平衡

最终，决定服务的数量和每个服务的体积，是一个很复杂的问题。它需要在创建更多服务所带来的好处，以及创建整体更复杂的系统所带来的坏处之间，仔细进行平衡。

创建过少的服务会带来同单体系统类似的问题，过多开发人员会同时在一个服务上工作，并且单个服务本身会变得异常复杂。

创建过多的服务会使单个服务变得异常简单，但是由于服务之间的交互，整体系统会变得异常复杂。我真的听说过，有些系统利用微服务来定义只是简单返回一个布尔值“是”或“否”的服务——这显然过于偏激。为服务准确定义合适的大小没有错，但是这取决于你的系统和公司的文化。最好的建议是，当你定义服务和架构时，时刻记得在复杂性上保持平衡。

在系统、组织和公司文化之间找到适当的平衡，对于充分利用微服务的好处非常重要。

确定服务大小的适当平衡，对于创建一个优化运维和管理的应用程序架构，以及保持应用程序的高可用性和可伸缩性来说，都非常重要。

第4章

服务和数据

当你在构建应用程序，或者将应用程序迁移到基于服务的架构时，一定要注意在系统中存储数据和状态的方法。

无状态服务—没有数据的服务

无状态服务指的是自己不管理任何数据和状态的服务。所有服务用到的状态和数据都来自请求传递过来的值。

无状态服务提供了优秀的伸缩能力。因为它们都是无状态的，所以无论是垂直扩展还是水平扩展，通过添加额外服务器来提升处理能力都是一件很简单的事情。如果你的服务不需要维护状态，那么就能够在如何伸缩和何时伸缩服务规模方面获得最大的灵活性。

此外，如果不需要考虑服务状态，你还可以在服务的前端使用某些缓存技术，利用更少的资源来处理更大的流量压力。

显然，并非所有的服务都可以是无状态的，但是对于那些可以是无状态的服务来说，可伸缩性的优势是巨大的。

有状态服务—有数据的服务

如我们之前所述，当你希望存储数据的时候，显然应当将数据存储在尽可能少的几个服务和系统中，同时避免直接在服务之间暴露数据，减少服务之间对内部数据的了解。

但是，真相却往往并不是如此。

我们应当尽可能地将数据保存在本地。服务必须使用的数据应当存储在服务本地，而其他数据应当存储到其他服务器或者数据库中，离需要这些数据的服务越近越好。

将数据保存在本地能带来以下几点好处。

减小单个数据集合的体量

由于你的数据分散在多个数据集合中，每个数据集合的体量就会较小。越小的数据集合的体量意味着数据交互越少，这使得伸缩数据库变得更加容易。这种方式也被称为功能性分区，即将数据根据不同的功能进行拆分，而不是基于数据集合的体量。

本地访问

通常，当你访问数据库或者存储器中的数据时，都是在访问一条记录或者一段记录。很多时候，其实你并不一定需要其中的很多数据。通过将数据分散到多个数据集合中，减少了不必要的数据查询量。

优化访问方式

通过将数据分散到不同数据集合中，你可以对每个数据集合进行适当优化。例如，某个数据集合是否应当保存到关系数据库中，还是应当保存到键/值数据库中？

将你的数据与使用数据的服务相关联，可以创建一个更加具有可伸缩性的解决方案，以及一个更易于管理的架构，随着应用程序的扩展，你可以更轻松地扩展数据的需求。

数据分区

数据分区的含义有很多种。在本节中，它表示将数据根据其中的某些键划分到不同的数据段中，通常为了利用多个数据库来存储更大规模的数据集合，或者获得比单个数据库更高的访问速度。

数据分区还有其他的类型（例如，之前提到的功能性分区），但是，在本节中，我们只关注基于键的数据分区策略。

举一个简单的数据分区示例，即将某个应用程序的所有数据按照账号进行分区。这样所有账户名称由A~D开头的的数据被划分到一个数据库中，所有账户名称由E~K开头的的数据被划分到另一个数据库中，以此类推（如图4-1所示）。^[1]这是一个非常简单的例子，但是应用程序的开发人员通常会采用数据分区技术，来大幅提升可以访问应用程序的用户并发数量，同时支持更大规模的数据量。

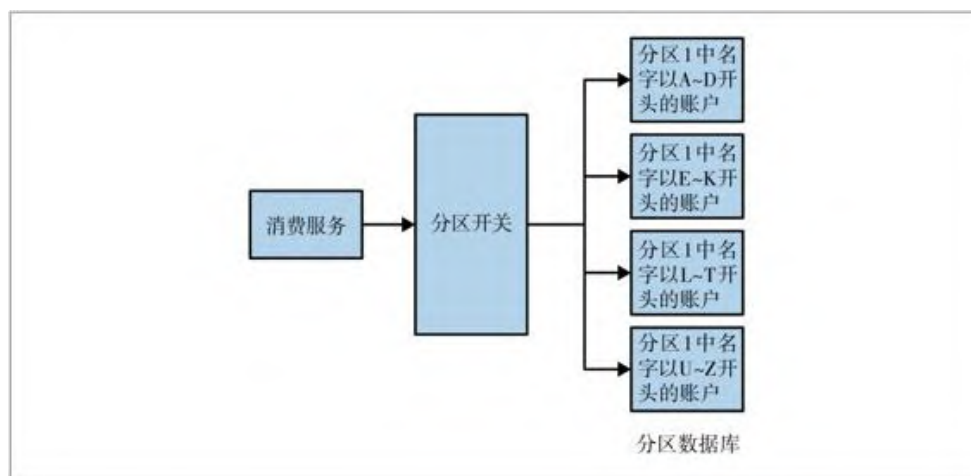


图4-1：根据账户名称进行数据分区的示例

一般来说，你应当尽可能避免使用数据分区。为什么？因为一旦像这样对数据进行分区，就可能遇到以下这些问题。

应用程序复杂性

增加了应用程序的复杂性。因为从现在开始，你在实际获取数据之前，需要先去计算数据存放在哪里。

跨分区查询

无法很容易地获取多个分区中的数据。这一点在做业务数据分析查询时非常有用。

分区使用倾斜

需要很谨慎地选择建立分区所依赖的分区键。如果你选择了错误的键，那么就可能导致数据库分区使用不均衡，某些分区使用频繁而其他分区很少被使用，这样不仅会降低分区的有效性，同时也会增加数据库管理和维护的复杂性，如图4-2所示。

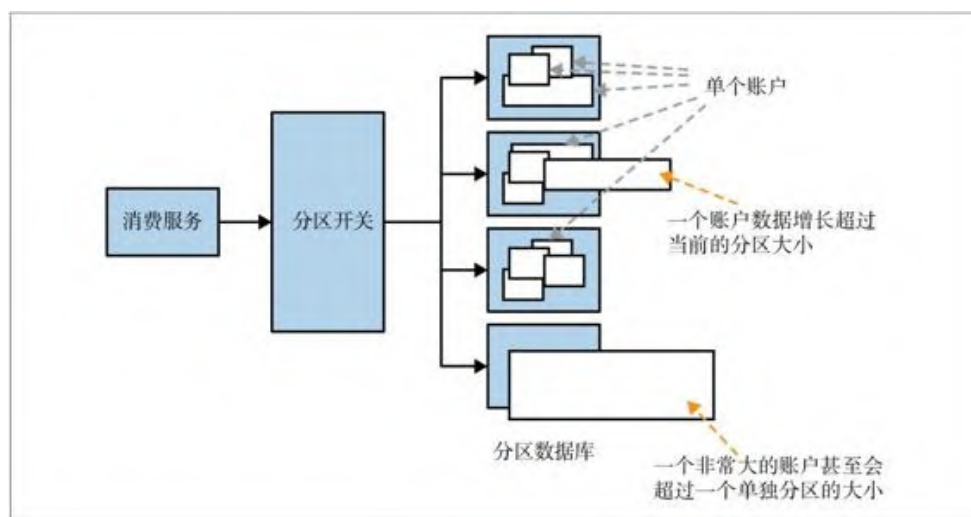


图4-2：账户超出数据分区能力的示例

重新分区

需要偶尔进行重分区来平衡分区之间的数据。根据选择的分区键和数据集合的类型、体量不同，重分区可能变得非常困难、非常危险（数据迁移），甚至在某些时候，几乎无法实现。

一般来说，账户名和账户ID都不是好的分区键（虽然它们经常被作为分区键使用）。这是因为一个账户的数据体量可能会发生变化。它开始时可能很小，适合分配在一个存放大量小账号的分区中。但是，如果随着时间推移，它变得越来越大，很快会因为单个分区无法承载压力，所以不得不重新进行分区，来更好地平衡账户数据。如果一个账号变得太大，甚至超过单个分区的容量，就会导致整个分区方案失败，即使是重分区也无法解决这个问题。

更好的分区键应该是大小尽可能保持固定的键。分区数量增长应当尽可能保持独立和一致，如图4-3所示。如果需要重分区，应该是所有的分区都一致地增长，因为分区数据量太大以至于无法处理所以才重新分配分区。

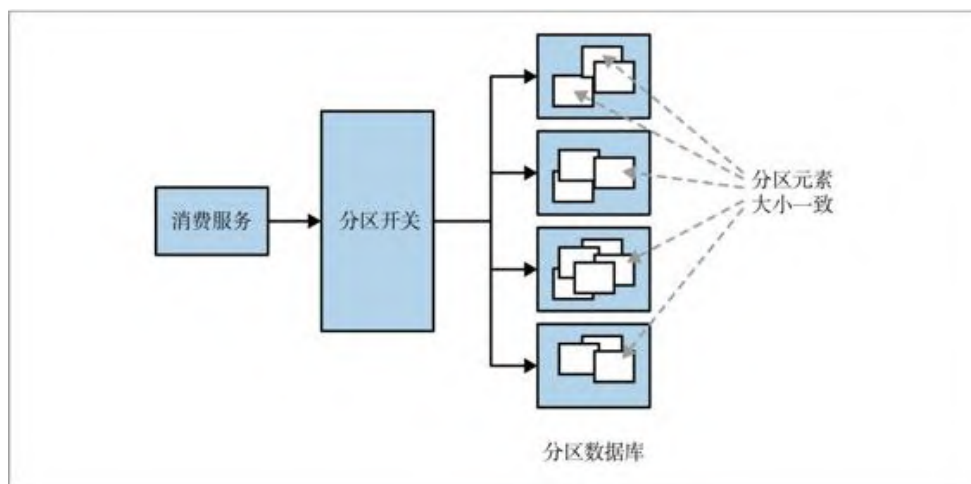


图4-3：数据保持一致增长的分区示例

实际中可能会用到的分区方案是，选择一个会产生大量小元素的分区键，然后将这些小分区映射到一个更大的分区数据库中。以后如果需要重分区，只需更新映射关系，将个别小元素移动到新的分区中即可，这样就避免了对整个系统的大范围重分区。如何选择和使用适当的分区键本身就是一门艺术。

及时处理增长的烦恼

如今大多数应用程序都经历着流量、体量、复杂性，以及开发维护人员数量的增长的过程。通常，我们会忽视这些增长所带来的问题，直到遇到瓶颈不得不开始解决它们。但是，到那个时候通常已经晚了。各种问题已经积压到很严重的程度，很多早期能够解决问题的简单技巧，到那个时候已经不再适用了。

如果在设计应用程序架构的时候，没有提前考虑到应用程序可能伸缩的情况，那么我们所做出的架构决策，可能就会阻碍我们根据业务需求进行伸缩的能力。

可以在设计和架构新的应用程序或者已有应用程序的时候，考虑对未来可能的规模变化的影响。比如，你为今后的伸缩预留了多少空间？你将遇到的第一个可伸缩性瓶颈是什么？当遇到这个瓶颈时会发生什么？如何在不对应用程序进行大量架构重构的情况下，对这个瓶颈做出响应并解决掉它？

如果能够在到达这种程度之前及时思考如何保持系统的长期增长，就可以预防很多问题的出现且可以不断改进应用程序，使它们安全平稳地度过增长期。

[1] 实际中经常使用的基于账户的分区机制，是使用账户标识来代替账户名称。本例中使用账户名称是为了方便读者理解。

第5章

处理服务故障

在构建大型基于微服务的系统时，如何处理服务故障是一个必须解决的问题。你拥有的服务越多，服务出现故障的可能性越大，依赖于故障服务的其他服务数量也会越多。如何在不增加应用程序不稳定性的情况下，处理这些服务故障呢？在本章中，我们将讨论一些处理服务故障的技术。

级联式的服务故障

假设你有一个服务，它依赖于多个服务，并且有一些服务也依赖于它。图5-1展示了这个“*Our Service*”与它的多个依赖（*Service A*、*Service B*和*Service C*），以及依赖于它的多个服务（*Consumer 1*和*Consumer 2*）。该服务依赖于3个服务，并且被另外2个服务依赖。

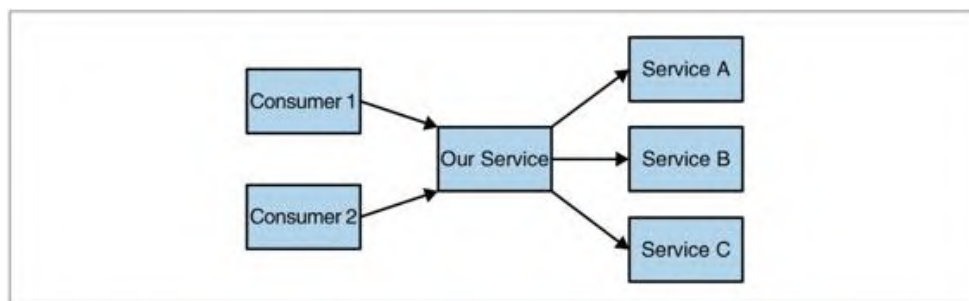


图5-1：某服务以及它的依赖服务和消费者们

如果其中一个依赖服务出现了故障会怎样呢？图5-2展示了服务A出现故障时的情形。

除非你非常小心，否则服务A的故障会导致“*Our Service*”也出现故障，因为它依赖于服务A。

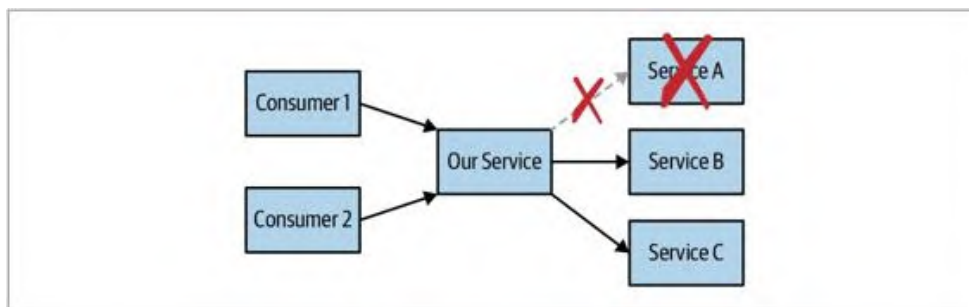


图5-2：某服务和一个发生故障的依赖服务

如果“*Our Service*”发生故障，会导致*Consumer 1*和*Consumer 2*出现故障。这个错误是可以级联发生的，从而导致更多的服务故障，如图5-3所示。

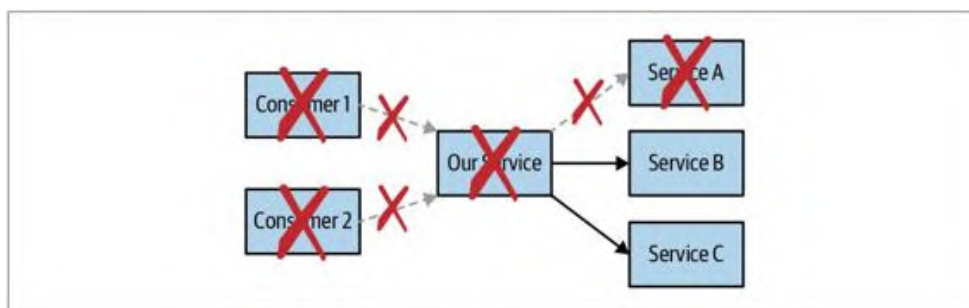


图5-3：级联故障

如果没有经过仔细检查，系统中的一个服务可能会导致整个系统发生严重的问题。

如何防止级联故障的发生呢？有些时候你什么也做不了一某个依赖服务中的一个错误就会导致你的服务（以及其他依赖服务）出现故障，因为这是从依赖的角度所决定的。有些时候，如果依赖服务发生故障，你的服务就无法完成应有的工作。但也不是全部如此。事实上，当某个依赖服务出现故障时，你还是有很多措施能够挽救自己的服务的。

如何响应服务故障

当你的某个依赖服务出现故障时，你应当如何响应？作为一个服务的开发人员，你对依赖服务故障的响应必须是：

- 可预测的
- 可理解的
- 对当前情形是合理的

我们来分别讨论一下其中的每一点。

可预测的响应

拥有可预测的响应，是当前服务能够依赖其他服务的一个重要方面。你必须对一系列特定的环境和请求提供可预测的响应，才能够避免之前提到的级联故障。否则，如果你掉以轻心，那么即使一次很小的故障也可能导致级联性的大面积故障，并最终引发大问题。

因此，即使其中一个下游的依赖服务出现了故障，你仍然需要为其生成一个可预测的响应，例如，可能是一个错误提示消息。只要在你的API中包含了适当的错误处理机制，生成这种错误响应是完全可以接受的。



错误响应与不可预测的响应并不一样。一个不可预测的响应指的是服务预料之外的响应，而错误响应是一个有效的响应，表示你无法处理特定的请求。二者是不同的。

如果你的服务正要执行操作“3+5”，那么它应该返回一个数字，准确地说应该是数字“8”，这是一个可预测的响应。如果你的服务准备执行操作“5/0”，那么它应该返回“非数字”或者“错误，无效请求”。这些都属于可预测的响应。不可预测的响应指的是例如某一次返回“50000000000”，但是下一次又返回“38393384384337”的情况（有时候又被称为“无用的输入输出”）。

一个属于“无用的输入输出”的响应不是一个可预测的响应，相对而言，一个可预测的响应应该是“无效的请求”。

你的上游依赖服务会希望你提供一个可预测的响应。当你遇到无用输入时，不要也返回一个无用输出。如果你将一个不可预测的响应

提供给了下游服务，那么会在整个服务调用链上传递这种不可预测性。这种不可预测的行为迟早会反馈到你的用户那里，从而影响业务的发展。可能更糟的是，这种不可预测的响应会将无效的数据插入你的业务流程，导致业务流程数据不一致和数据无效。这会影响你的业务分析，也会给用户造成不好的体验。

即使你的依赖服务出现故障或者发生了不可预测的行为，你也应尽可能不要把这种不可预测性传递给依赖于你的服务。



可预测的响应实际上意味着它是一个计划中的响应。不要有“好吧，如果依赖服务出现故障，我也做不了任何事情，只好任由服务出现故障”这样的想法。如果所有事情都出现故障了，你需要主动发现在当前情况下，一个合理的响应应当是什么样的，然后再检测当前情况是否满足条件，并返回预期的响应。

可理解的响应

可理解，意味着你和上游服务之间对响应的格式和结构都表示同意，从而在你和上游服务之间形成了一个约定。即使你的依赖服务出现了故障，你的响应也必须控制在约定的边界之内。永远不应该仅仅因为依赖服务违反了它的API约定，你就违反你自己的API约定。应该确保约定的接口能够覆盖所有意料之外的情况，包括依赖服务出故障的情况。

合理的响应

你的响应应该说明当前服务实际发生了什么事情。当问到“3+5等于几”的时候，即使依赖服务出现故障，也不应该返回内容为“红色”的响应。你的服务可以返回“对不起，我无法计算结果”或者“请稍后尝试”的响应，但绝对不应该返回“红色”作为答案。

虽然这听上去没什么，但是你会对实际中不合理的响应所带来的问题数量感到惊讶。例如，假设一个服务希望获取一个已过期、需要删除的账户列表，如图5-4所示。你可能会调用一个“过期账户”服务

（返回一个需要删除的账户列表），然后继续删除列表中的所有账户。

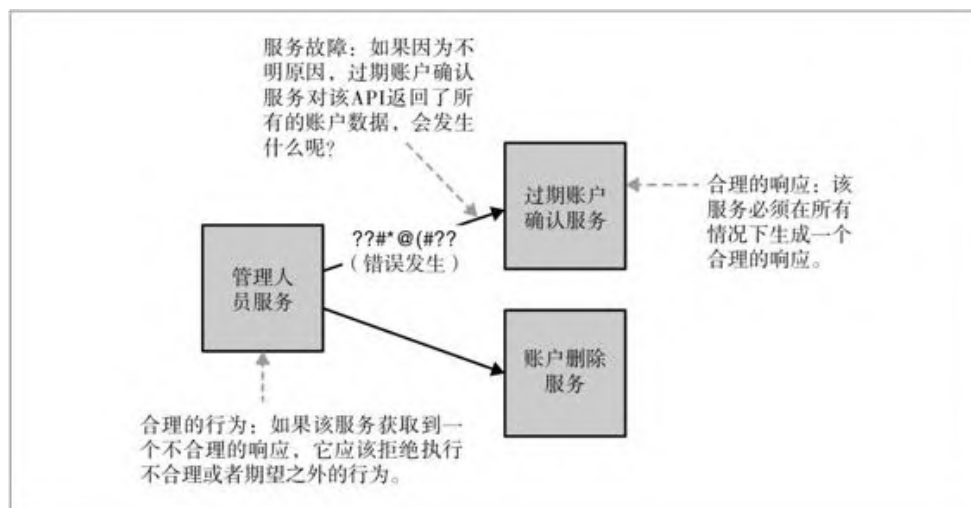


图5-4: 不合理的API响应

如果“过期账户”服务出现问题, 无法返回有效的响应, 它应当返回“无”, 或者“对不起, 现在无法返回结果”这样的响应。想象一下, 如果它没有返回一个合理的响应, 而是返回了系统中所有账户的列表呢? 在这种情况下, “管理服务”会继续尝试并删除系统中的所有账户, 这肯定是错误的做法, 如果突然删除应用程序中的所有账户, 结果将是灾难性的。

如何确定故障

既然知道了如何响应失败, 接下来让我们讨论如何确定依赖的服务是否出现故障。如何确定一个依赖服务是否出现了故障呢? 这取决于故障的模式。以下是几种需要重点了解的故障模式。

乱码响应

这种响应是无法理解的。它是一种以无法理解的形式存在的“垃圾”数据。这可能代表响应的格式错误, 或者格式中存在语法错误。

表示致命错误发生的响应

这种响应是可以理解的。它表示在处理请求时发生了严重的错误。这通常可能不是因为网络传输的原因，而是服务本身出现了错误。它还可能是由于发送给服务的请求无法被理解而引起的。

结果可以理解，但是所需的结果不匹配

这种响应是可以理解的。它表示操作执行成功，没有严重错误，但是返回的数据无法与预期的结果相匹配。

结果超出预期范围

这种响应是可以理解的。它表示操作执行成功，没有严重错误。返回的数据是合理的，并且格式正确，但是数据本身不在预期之内。例如，假设一个计算当天距离当年第一天日期的请求，如果它返回一个“843”的结果会怎样呢？这个结果是可理解的、可解析的，没有失败，但是显然没有在预期的结果范围之内。

没有接收到响应

请求已经发出，但是没有接收到服务发回的响应。这可能是由于网络通信故障、服务出现问题，或者服务故障导致的。

接收响应很慢

请求已经发出，并且响应也接收到了。响应的内容正确，也在预期范围之内。但是，响应比预期到达时间晚很长时间才接收到。这通常表明服务或者网络压力过大，或者存在其他资源分配不合理的问题。

当接收到乱码时，你立即知道响应不可用，并且可以采取适当行动。一个可理解但是不匹配所需结果的响应可能更难被发现，也更加难以决定如何行动，但是至少我们还可以得到这些响应。

一个永远无法到达的响应，使你很难对结果采取适当的行动。如果你只想给用户返回一个错误响应，那么在依赖服务处加上一个简单的超时处理，可能就能够捕获到丢失的响应。

捕获永远无法到达的响应的更好方式

但是，超时不一定总是行得通。例如，如果一个服务通常需要50毫秒响应请求，但是有些时候可能只需要10毫秒，但有些时候又需要500毫秒，你该怎么办呢？你应该将超时时间设置成多少呢？一个显然的答案是大于500毫秒。但是，如果你对用户承诺的响应时间小于150毫秒呢？显然，仅仅将时间设置成500毫秒并不合理，因为这就相当于将依赖服务的错误直接传递给了用户。这违反了可预测和可理解的测试原则。

如何解决这个问题呢？一个可能的答案是使用断路器模式。这种代码模式让你的服务可以追踪对依赖服务的所有调用，了解有多少次调用成功，多少次调用失败（或者超时）。如果失败次数达到某个阈值，断路器会“切断”与依赖服务的连接，让你的服务以为依赖服务已经宕机，并停止向该服务发送请求或者接收响应。这使得你的服务可以立即检测到依赖服务发生故障并采取适当的行动，从而保证上游服务的SLA。

随后，你可以通过向故障服务定期发送请求，来检查它是否恢复正常。如果它又可以成功处理请求（超过一个预定的阈值）了，那么断路器会“重置”，此时你的服务就可以恢复使用该依赖服务。

一个很慢的响应（相对于永远无法接收的响应）可能是最难处理的。这个问题变成了到底多慢算是慢？这是一个很难回答的问题，仅仅通过简单的超时（不管使用或者不使用断路器）很难很好地处理这种情况，因为慢响应可能在“某些时候”又很快，从而导致出现一些很古怪的结果。记住，响应的可预测性是服务的一个重要特征，如果你所依赖的服务总是不可预测地出现故障（因为慢响应和不稳定的超时），这也会影响你对依赖服务提供可预测的响应。

一种检测慢响应的更优雅的办法

除了断路器和相似的方法，还有一种更优雅的超时机制可以处理慢响应情况。例如，你可以创建多个“桶”来捕获最近调用依赖服务的性能。每次调用依赖服务时，你将返回响应的时间存储到一个桶中。你只在桶中保存一段时间内的响应时间。然后，可以通过这些桶来计算出一个触发断路器的规则。例如，你可以创建以下这些规则：

- 如果“1分钟之内接收到500个请求超过150毫秒”，则触发断路器。
- 如果“1分钟之内接收到50个请求超过500毫秒”，则触发断路器。
- 如果“1分钟之内接收到5个请求超过1000毫秒”，则触发断路器。

这类分级技巧可以帮助你更早地发现更严重的慢响应问题，又不会忽略轻微的慢响应问题。

适当的行为

当错误发生时你应该做什么呢？这取决于错误的类型。以下是一些针对不同错误类型你可以采取的方案。

优雅降级

如果一个依赖服务出现了故障，你的服务没有响应还能够工作吗？它是否能够在缺少故障服务的响应下，继续执行所需的任务？如果在这种情况下它还能执行有限的功能，那么就是一个优雅降级的例子。

优雅降级指的是在当前服务缺少某个故障服务的结果时，可以通过降低工作量来尽可能地完成工作。

提供有限功能

假设你有一个Web应用程序，用来创建一个售卖T恤的电子商务网站。我们还假设有一个“图片服务”为网站上显示的图片提供URL。如果应用程序调用了该图片服务，但是图片服务出现了故障，那么应用程序应该怎么办呢？一种选择是，应用程序继续给用户显示所需产品，但是没有该产品的图片（或者显示一条“图片不可用”的消息）。这个Web应用程序可以继续作为电子商务网站运行，只是缺少了显示产品图片的能力。

这种做法，比起只是因为图片不可用，整个电子商务网站就只好停运并返回错误的做法已经好太多了。

这就是一个提供有限功能的例子。对于一个服务（或者应用程序）来说，即使因为依赖服务由于故障不能再提供所需数据，也应当尽可能地提供服务价值，这一点是非常重要的。

优雅补偿

有观点认为，如果请求没有返回足够有用的结果，那么也应该算作失败。除了生成一条错误消息，你还能做些什么为用户提供价值呢？

即使你无法完全满足用户的需要，但是按照为用户提供价值的方向去改进，这就是一个优雅补偿的例子。

优雅补偿示例

我们继续“提供有限功能”例子中所描述的请求，假设提供指定产品详情内容的服务失败了。这意味着网站无法为所需产品提供任何可显示的信息。显然只显示一个空白页是不合理的，因为这样对用户来说没有任何用处；显示一条错误消息（“对不起，发生了一个错误”）也不是一个好主意。

但是，你可以显示一个向用户道歉但同时提供网站上当前最流行产品链接的页面。虽然这并不是用户真正想要的，但是它给用户提供了一些价值，避免了仅仅显示一个突兀的错误页面的情况。

尽早失败

如果你的服务无法收到故障服务的响应就无法工作呢？如果无法选择提供有限的功能或者优雅补偿呢？当你无法接收来自故障服务的响应时，就无计可施了。在这种情况下，你能做的只有让请求失败了。

如果你已经确定无法挽救请求，那么应当尽快让请求失败。当你知道请求一定会失败后，就不要再执行请求中其他的操作或任务了。

这样做的结果就是你会尽可能地对请求进行完备的检查，尽可能地提前确认请求的正确性，当你继续发送请求时，该请求很可能成功。

除以零

假设有一个计算两个整数相除的服务。我们都知道除数是不能为0的。如果你接到一个像“3/0”这样的请求，你可能会试着计算结果。在计算过程中，最终会发现无法得到结果，于是抛出一个错误。

既然你知道所有除以0的计算都会失败，那么只需对请求的数据进行一下检查。如果除数是0，则立刻返回一个错误。没有任何理由再去尝试进行计算。

为什么应该尽早失败？有以下几个原因。

资源节约

如果请求注定失败，那么在它失败之前进行的所有工作都是徒劳的。如果这些工作包括需要多次调用依赖的服务，那么浪费这么多宝贵的资源却只能得到一个错误。

响应性

你越快确定一个请求会失败，就能更快地把这个结果返回给发起请求的一方。这使得请求方可以更快地做出其他决定。

错误复杂性

有些时候，如果你继续处理注定会失败的请求，很可能会造成更难诊断的问题。我们还以“3/0”这个计算为例，你可以立即确定这次计算会失败，并返回错误消息。如果你选择继续计算，就会产生错误，但是可能会以更加复杂的方式出现—例如，根据你计算除法的算法不同，这可能会导致一个无限循环，最终只有等到超时后才能退出。

因此，除了得到一个提示，例如，“除以0”的错误，你还可能会等待很长时间并得到一个“操作超时”的错误。相比而言，你应该很清楚哪个错误信息在诊断问题时会更加有用。

用户导致的问题

当你的服务可以接收来自用户的无效输入时，尽早失败变得更为重要。如果你知道服务已经规定了哪些合理的限制条件，请尽早检查它们。

一个浪费资源的真实案例

在我之前工作过的一家公司中，有一个账户服务存在性能问题。该服务逐渐变得越来越慢，直到它几乎不可用。

在深入研究问题之后，我们发现有人向账户服务发送了一个错误请求。有人请求账户服务去获取100,000个用户的账户列表，包括所有账户的详细内容。

现如今，没有任何正常的业务会有这样的需要（在这个环境中），因此这个请求本身显然是无效的。100,000这个值超出了这个请求的合理范围。

但是，账户服务很负责任地尝试去处理这个请求，于是处理、处理、不断地处理……

这个服务最终由于没有足够资源来处理如此巨大的请求，所以失败了。当处理了几千个账户后它停止了服务，并返回了一个简单的错误消息。

而发起无效请求的调用方服务，由于接收到了失败消息，于是决定重试这个请求，于是重试、重试、不断地重试。

账户服务不断地处理这几千个账户，将这些结果扔到了一个失败信息中。但是它不断在重复做这件事。

不断失败的请求消耗了大量的可用资源。随着过多资源被消耗，导致其他合法请求也开始排队，并最终导致整个服务出现故障。

如果能在账户服务处理请求前进行一个简单的检查（例如，检查请求的账户数量是否合理），就可以避免对资源的大量浪费。此外，如果返回的错误消息能够表明这个错误是永久性的，并且是由一个无效的参数所导致的，那么调用方服务能够看到这个“永久错误”的消息，也就不会再去重试一定会失败的请求了。

提供服务约束

这个案例的结果告诉我们一定要提供服务约束。例如，如果你知道服务不能同时处理超过5000个账户的数据，一定要在服务约定中声明这个限制条件，进行测试，让所有超过该限制数量的请求都直接失败。

小结

“无用输入、无用输出”不是一种处理错误的正确方法，因为它将识别错误结果的责任传递给可能无法做出有效决策的其他服务。错误的数据应该被尽早发现，并得到适当的处理。此外，即使在出故障的情况下，服务也应该始终以可靠和可理解的方式运行。而且它们不应该产生垃圾的或者无法理解的结果。

第III部分

原则3. 组织：为现代化应用程序建立可伸缩性的组织

除非你的开发团队使用现代化的过程管理手段，否则你无法构建现代化的软件。现代化的应用程序需要现代化的组织。

应用程序如何伸缩并不重要，但如果你的开发团队的组织架构不支持可伸缩性，或者你的组织没有正确的文化来驱动更高的可用性和更大的可伸缩性，那么你就无法伸缩你的应用程序。

组织好你的团队，使其可以更好地支持你的可伸缩性需求，这会创造出一种文化，从而支持应用程序的可伸缩需求。

第6章

服务所有权——STOSA

在第3章中，我们讨论了什么是服务，以及如何利用服务来帮助处理应用程序的复杂性，以及将其划分到许多不同的开发团队中，让每个团队都使用自己的代码库并支持自己的服务。我们讨论了如何划定服务的大小以及服务之间应该如何交互。

但是我们并没有深入探讨团队对服务的“所有权”的具体含义，以及为什么这种所有权是重要的。在本章中，我们会介绍服务所有权的意义，以及如何来实践由独立团队负责的服务架构。

由独立团队负责的服务架构

什么是“由独立团队负责的服务架构（STOSA）”？STOSA对于由多个开发团队来负责并管理一个或多个系统服务的大型组织来说，是一个重要的指导原则。拥有一个STOSA的系统和组织意味着什么？要达到STOSA，必须满足以下几个条件：

- 你必须有一个基于服务或者微服务架构建立的应用程序。
- 由多个开发团队负责构建和维护这个应用程序。
- 应用程序中的每个服务必须分配给某个开发团队，由它们所有。哪个团队所有哪个服务应该有良好的文档记录，并且组织中的每个人都可以很方便地知道。
- 不允许有某个服务被分配给多个开发团队。
- 一个开发团队可能会负责多个服务。

- 开发团队负责管理服务的所有方面，从服务架构到设计再到开发、测试、部署、监控和故障处理。
- 服务之间有清晰的边界，包括文档编写良好的API接口。
- 服务拥有自己的数据。数据是服务的一部分。如果一个服务需要访问存储在另一个服务中的数据，那么它必须使用一个文档编写良好的API来访问这些数据。
- 各服务负责维护服务之间的服务等级协议，开发团队负责提供服务进行监控和报警。

一个基于STOSA的应用程序，就是一个所有服务都满足以上条件的应用程序。一个基于STOSA的组织，就是所有服务团队都满足以上条件，并且所有应用程序都是STOSA应用程序的组织。

在一个基于STOSA的组织中，每个团队的大小是确定的（通常在3~8人之间）。如果一个团队太小，它无法有效管理某个服务。如果团队太大，管理团队的工作会太繁重。

图6-1展示了一个基于STOSA的组织，以及它是如何来管理STOSA应用程序的。

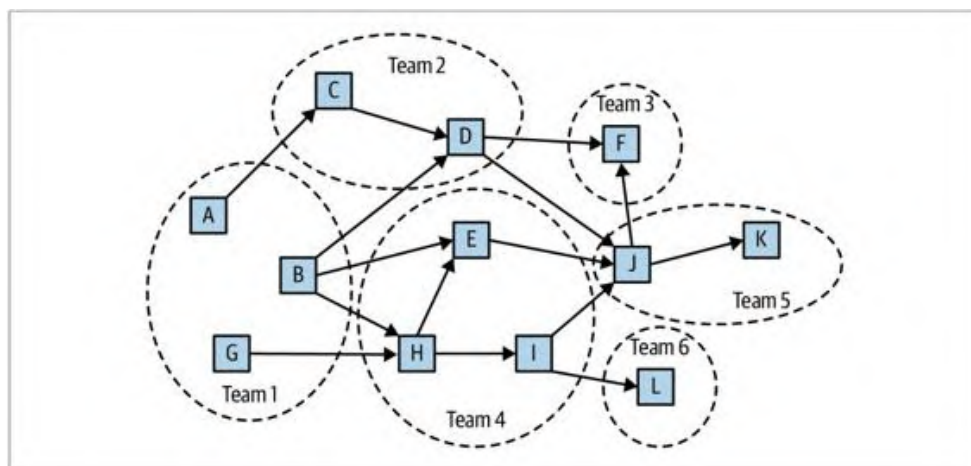


图6-1：管理基于STOSA应用程序的STOSA组织

在图6-1中，标识从A到L的方块表示应用程序中每个独立的服务。椭圆表示完全负责这个服务的开发团队。

这个应用系统包含了由6个团队管理的12个服务。你会注意到，每个服务由一个团队来管理，但是一个团队可能会管理多个服务。每个服务都有一个所有人，并且一个服务只能有一个所有人。

应用程序的每个方面都有清晰的所有权，对于系统的任何一个部分，你都可以很容易地知道谁是负责人，以及有疑问、问题或者改动应该联系谁。

图6-2展示了一个非基于STOSA的应用程序和组织。

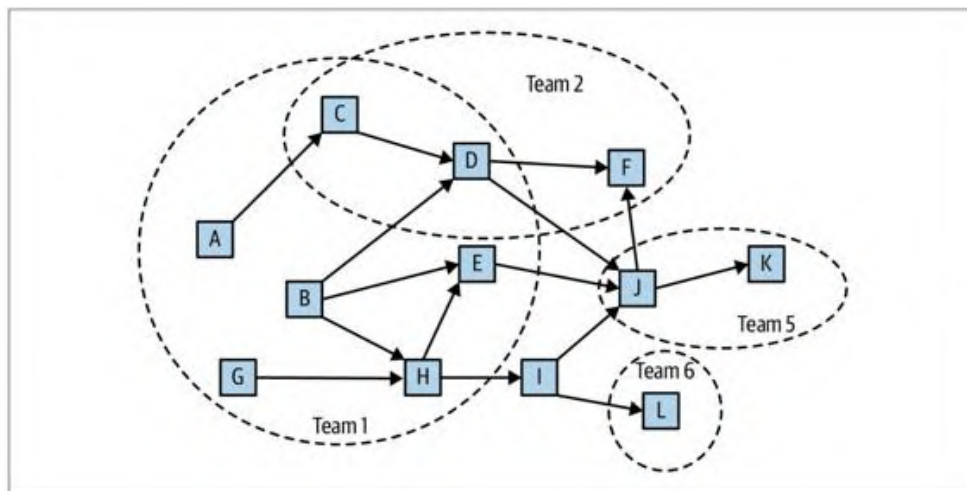


图6-2：非基于STOSA的组织

你会注意到几件事情。首先，服务I没有所有者，并且服务C和服务D归属于多个团队。

在图6-2中，服务之间没有清晰的所有权。如果你需要在服务C或者服务D中完成某些事情，不清楚谁应该为此负责。如果其中一个服务出现问题，应该由谁来响应？如果你需要让服务I来完成某些事情，你应当联系谁？缺少清晰的所有权和责任会让一个复杂系统变得更加难以管理。

STOSA应用程序和组织的好处

随着程序体积的增加，复杂性也在增加。一个基于STOSA的应用程序，可以比一个非基于STOSA的应用程序发展得更大，可以由一个更大

的开发团队来管理。因此，它可以在向更大规模发展的同时，依然保持稳定的、文档化的以及可支持的接口。

一个基于STOSA的组织比一个非基于STOSA的组织，可以管理更大、更复杂的应用程序。这是因为STOSA可以在多个开发团队之间有效分散系统的复杂性，同时保持清晰的所有权和职责范围。

“拥有”一个服务意味着什么

在一个STOSA组织中，服务所属的团队需要百分之百地负责服务的所有方面。这个团队可能需要其他团队的支持（例如，硬件运维团队），但是最终该服务是由这个团队来完全负责的。

其中特别包括以下这些方面。

API设计

负责所有内部、外部API的设计、实现、测试和版本管理工作。

服务开发

负责服务所有业务逻辑的设计、开发和测试工作。

数据

负责管理服务所产生和维护的所有数据，包括数据的展现、模型、访问方式以及生命周期。

部署

负责决定服务是否及何时需要升级，以及相关的部署工作，其中包括对所有服务节点的验证和回滚工作，以及保证部署过程中服务的可用性。

部署窗口

决定什么时间可以安全地进行部署。这涉及公司或产品范围的停服以及各个服务的时间窗口。

产品基础设施变更

服务所需的所有产品基础设施变更（例如，负载均衡器的设置和系统调优）。

环境

管理服务的生产环境，以及所有的开发、预发布以及预生产部署环境。

服务SLA

讨论、确定并监控SLA，以及保障服务满足SLA的相关工作职责。

监控

确保为服务的各个方面建立了监控，包括监控服务的SLA。还包括定期检查监控的情况。

突发事件响应

确保系统在出现故障时能够生成报警事件。安排轮流值班制度以及报警通知管理，确保团队中有人能够处理突发事件。确保对突发事件的响应速度能够满足之前定义的SLA。

报告

向其他团队（消费服务或者依赖服务）发送内部报告，以及管理服务的运行健康报告。

通常，这些方面中的一些不是由服务所属的团队直接处理，而是由公共的基础设施、工具、运维或平台工程团队负责的。然而，即使在由其他核心团队处理的情况下，确保这些行为被控制在满足其SLA和客户期望所需的水平，最终也是服务所属团队的责任。

通常，以下几项不是由所属团队负责，而是有一个公共的团队来处理。

服务器/硬件

所有生产环境和其他环境下需要的硬件和基础设施。这通常由一个运维团队、云服务提供商或者二者同时提供。

工具集

集中提供和管理团队需要的各种工具。这其中包括部署工具、编译和代码管理工具、监控工具、值班和突发事件响应工具，以及报告工具。

数据库

通常由一个公共团队来管理负责存储服务数据的硬件和数据库。但是，数据本身、数据模式以及对数据的使用，通常由服务所属团队来管理。

图6-3展示了一个基于STOSA的组织机构。特别要注意的是，从组织机构角度来看，所有服务所属的开发团队都是平级的。它们统一由一些支持团队，包括运维、工具、数据库以及其他团队来提供支持。所有这些团队可能都基于其他核心团队之上，为整个组织提供支持，而不是为某个单独的服务提供支持。这可能包括像架构设计或者项目管理这样的团队。



图6-3：基于STOSA的组织架构

在STOSA组织中，服务所属团队会负责该服务的所有方面。它们可能会依赖于核心团队或者支持团队，但是最终是由它们来处理该服务的所有问题，保证服务的正常运行。

例如，我们假设因为核心团队的部署工具问题导致服务部署失败，那么也应该由服务所属的团队来负责。虽然它们可能需要与开发工具的团队来沟通解决，但是最终对此负责的一定是这个团队。它们不能推托说“这是工具团队的错误”这样的话。因此，即使最后发现事实是这样，但是因为服务出现了故障，所以由负责这个服务的团队来承担责任。

伴随着对结果完全负责，服务团队拥有强大的决策权力。通常，服务所属的团队会被分配一系列需要实现的需求，但是如何实现由它们来决定。这个团队可能需要遵守很多系统方面的要求（例如，架构方面的设计原则或者规则，必须使用的工具，只限使用某种编程语言和硬件，或者行业相关的监管要求等），但是这些最终都只是团队需要满足的需求而已。

除了这些需求之外，所有的设计和实现细节都由服务团队来决定。

最终，服务所属团队会承诺达到预期的结果，并维护合理的SLA。

使用核心团队和服务

通常在一个强大的STOSA组织中，服务团队可能会选择不使用一个标准的共享核心和支持功能。例如，它们可能支持自己的数据库，而不是使用由一个统一的数据库团队提供和支持的数据库。或者它们可以决定使用自己的云服务提供商，而不选择由运营团队支持的云服务提供商。

只要服务团队满足其特定的需求，就不必强制它们使用这些公共基础设施组件。当然，对于服务团队来说，使用标准的、共享的能力是有好处的。如果团队选择不使用这些由其他团队支持的共享能力，实际上可能会为自己带来额外的支持问题。然而，关键是这个决定是服务团队自己做出的，并且必须考虑到它所带来的影响。

这种方式的一个优点是能带给核心团队动力和责任，让它们把服务团队当作真正的客户。如果不能为客户提供他们需要的功能，客户就会做出其他的选择。这可以为集中式团队提供强大的动力，从而为服务团队提供更高质量的产品。

你的组织不需要一定这样做来支持STOSA，事实上，你的服务团队很可能会提出需求，要求使用核心的基础设施组件。但总的来说，服务团队的灵活性越大，其独立创造性就越大，最终产生的服务也就越好。

随着组织的发展和规模的扩大，各个服务团队自然会倾向于接受这些标准化的核心平台团队。事实上，在一个大型的、高度分散的组织中，服务团队“被迫”使用公共平台和“选择”使用公共平台可能没有什么区别，因为这是它们满足指定需求的唯一方法。你越能让服务团队做出这样的选择，无论是真实的还是想象的，你就越能在组织中获得更多的支持。

小结

STOSA是一个重要的模型，用来确定各个开发团队应该如何拥有和管理服务。它描述了一种组织文化模型，使得构建和维护可伸缩的服务成为一种可能。在第III部分的其余章节中，我们将继续讨论服务所有权，并将重点放在服务交互和服务之间的接口上。

第7章 服务分级

开发包含许多服务的大型、复杂的应用程序会存在可用性的问题。某个服务的故障可能会导致依赖它的所有服务都出现故障。这样所带来的级联影响会让你的整个系统都不可用，尤其是当一个非关键服务的故障导致了关键服务出现故障时，会带来非常恶劣的影响。

服务分级是与某个服务相关的标签，其表明了该服务对你的业务运营的关键程度。服务分级允许你管理应用程序的复杂性，并以一种分布式和有组织的方式来理解单个应用程序服务的重要性。

应用程序的复杂性

如图7-1所示，有些时候，最不起眼的服务反而可能出现故障。

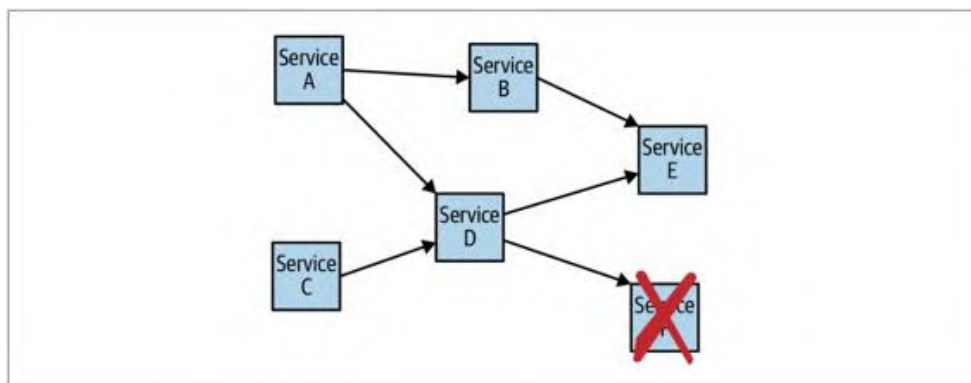


图7-1：某个服务失败……

这可能导致你的整个应用程序都无法提供服务，如图7-2所示。

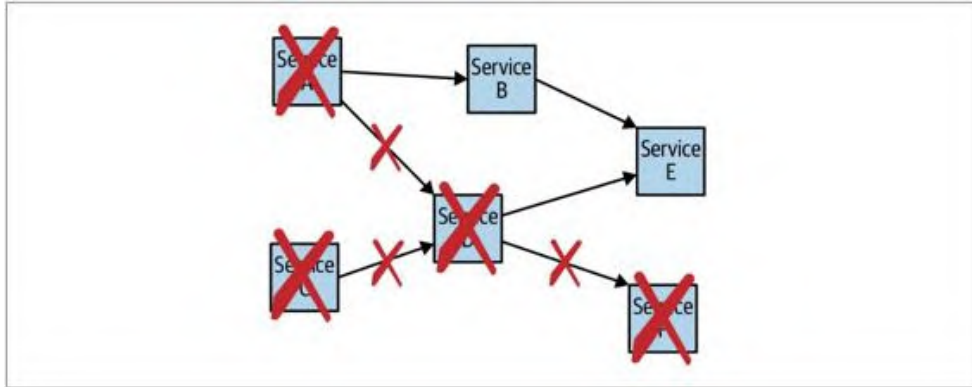


图7-2：可以导致级联性故障

我们已经在第5章讨论了很多可以预防依赖服务发生故障的方式。但是，服务间的弹性也会增加复杂性和成本，并且有些时候并不需要。以图7-3为例，如果服务D对于服务A的运行来说不是必要的，会发生什么呢？为什么仅仅因为服务D出现故障，服务A也要发生故障呢？

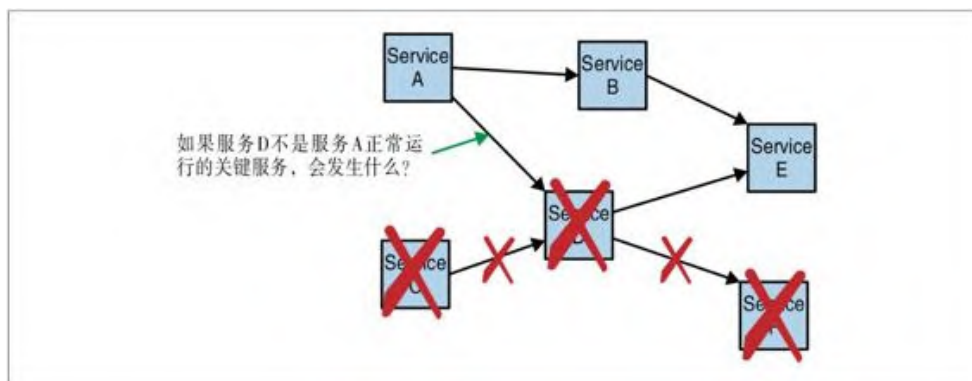


图7-3：如果服务D并不是关键服务，会发生什么呢

你如何知道什么时候服务的依赖链是关键的，什么时候又不是关键的呢？服务分级正是一种可以帮助你管理服务依赖的方法。

什么是服务分级

服务分级其实只是与某个服务关联的标签，表示该服务对于业务运行的关键程度。服务分级让你能够区分出哪些服务是关键性任务，哪些服务有用但不是至关重要的。

通过比较各个依赖服务的服务级别，你可以确定哪些服务依赖于业务最为敏感，哪些服务略微次之。

为服务分配服务级别标签

系统中的所有服务，不管它多大或多小，都应当被分配一个服务级别。以下内容会大致告诉你如何来为一个服务分配级别（你可以根据自己的业务需要来调整建议）。

1级服务

1级服务是系统中最关键的服务。如果某个服务出现故障会导致用户或者公司业务产生重大损失，那么应当考虑将这个服务定为1级。

以下是一些1级服务的示例。

登录服务

如果用户不能登录到你的系统，那么也就无法使用你的系统。

信用卡处理服务

如果用户不能使用他们的信用卡，他们就无法完成订单，你也收不到钱。

权限服务

不是所有用户对同一个功能都有相同的访问权限。如果权限系统出现故障，用户就无法访问他们相应的功能。

订单处理服务

如果用户无法支付并完成订单，那么你也无法收到钱，用户也无法收到商品。

1级服务的故障对于公司来说有重要的影响。

2级服务

2级服务对于业务非常重要，但是在关键性上不如1级服务。如果2级服务发生故障，会导致用户体验显著受到影响，但是不会让他们完全无法使用你的系统。

2级服务也会显著影响你的后台服务流程，但是可能不会直接对用户造成影响。以下是一些2级服务的示例。

搜索服务

用户希望能够在你的站点上搜索产品和信息。如果没有搜索功能，他们仍然可以使用你的网站，但影响某些功能的使用。

订单结算服务

在仓库中处理订单并将订单发送给用户非常重要，但是用户不会注意到在完成订单时出现了短暂的停机。

2级服务的失败会对用户产生负面影响，但是不会导致系统完全失败。

3级服务

3级服务会对用户造成较小的、不容易注意到的，或者很难发现的影响，或者对你的业务和系统造成有限的影响。

以下是一些3级服务的示例。

用户头像服务

在网站页面上显示客户图标或者头像的服务。如果它无法工作，大多数人可能不会注意到。如果他们注意到了，也不是什么大问题。

推荐服务

推荐服务是在你的网站上交叉销售产品的好方法，但是如果它无法工作，客户仍然可以购买商品，你仍然可以完成那些订单。

每日提醒服务

通常，我们希望在所有客户首次访问我们的站点时，在页面顶部显示一条消息。如果不能提供这一功能，客户可能会错过一笔订单，但他们甚至可能不知道错过了什么。

用户甚至可能注意不到3级服务出现了失败的情况。

4级服务

4级服务即使失败，也不会对用户体验造成任何严重的影响，更不会对业务或者资金方面造成任何严重损失。

以下是一些4级服务的示例。

销售报告生成服务

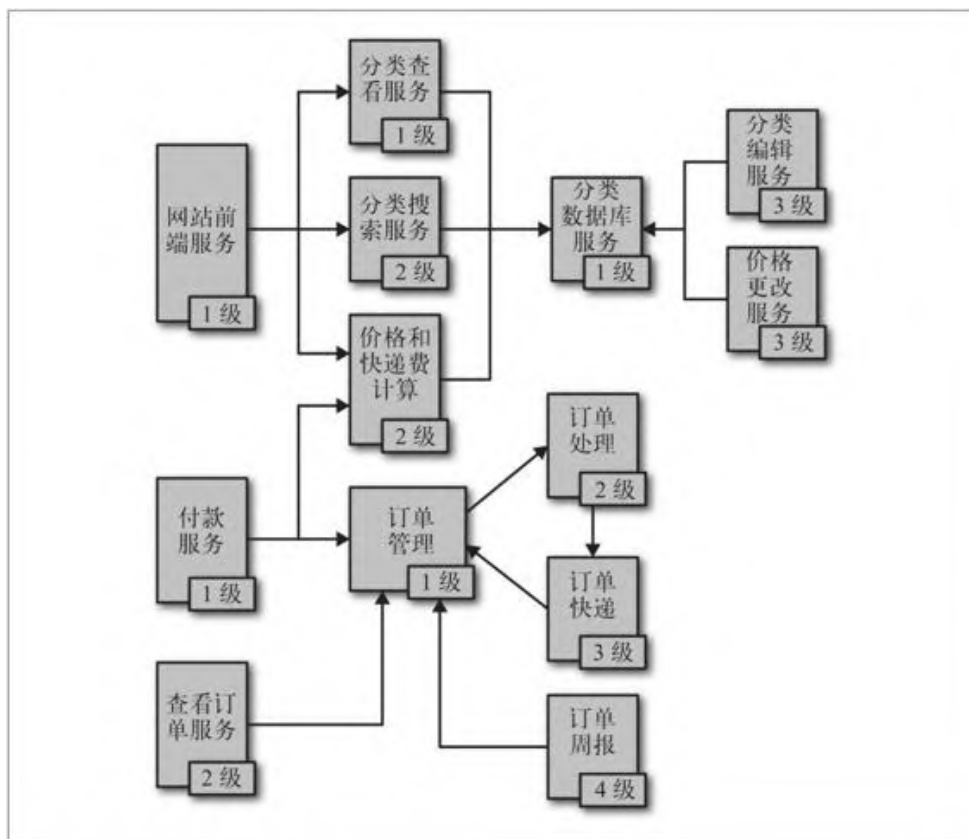
生成每周销售报告的服务。虽然销售报告很重要，但是生成服务的短暂故障不会造成任何严重影响。

电子邮件发送服务

定期向用户发送电子邮件的服务。如果服务宕机一段时间，邮件发送可能会被推迟，但是通常不会对用户造成任何严重的影响。

示例：在线商店

图7-4所示的是一个由许多服务组成的在线商店示例程序。每个服务都有一个标签来表示被分配的服务级别。



看着图7-4，从描述中想象一下每个服务的职责，并想象一下当服务出现故障时用户可能或应该面对的体验。服务级别应该与用户体验保持一致。

下面提供了一些该应用中的示例服务。

网站前端服务 (1级)

这是生成和展示网站的服务。它负责生成HTML页面，与用户在浏览器中进行最直接的交互。

这是一个1级服务，因为如果没有它，用户就无法访问你的整个在线商店。如果该服务不可用，那么会对用户造成严重的影响，所以它通过了1级的测试。

分类查看服务 (1级)

该服务会读取分类数据库，并将合适的分类数据发送给前端服务。它用来生成展示单个商品详细信息的页面。

这也是一个1级服务，因为没有它用户就无法在线查看任何商品。如果该服务不可用，那么会对用户造成严重的影响，所以它通过了1级的测试。

分类搜索服务（2级）

这个服务处理来自用户的搜索请求，并返回匹配搜索项的商品列表。

这是一个2级服务，因为虽然搜索对网站来说是很重要的功能，但是没有它，用户依然能够在网站上浏览商品，直到搜索服务恢复。用户体验会被显著削弱，但是网站依然可用。

分类数据库服务（1级）

这是存储分类数据本身的数据库服务。

这是一个1级服务，因为如果没有分类数据库，网站就无法显示任何商品。

分类编辑服务（3级）

这是一个你的雇员用来添加新的分类商品，以及编辑已有分类商品的服务。

该服务是一个3级服务，因为它对于用户成功完成购买不会起到关键影响。虽然不能将新的商品添加到数据库中会影响业务，但是它不会立即或者直接影响到你的用户，并且这一点缺陷是可以接受的。

付款服务（1级）

这个服务用来为用户显示付款流程。如果没有该服务，你的用户就无法购买商品。

这是一个1级服务，因为它对你的用户（他们不能购买商品）和业务（用户购买不了商品你就无法挣钱）都会造成非常严重的影响。

订单快递（3级）

这是打包并快递用户订单（一个很简单的例子）的服务。没有该服务，你的用户就无法收到他们购买的订单。

这看上去可能应该是一个1级服务，因为快递订单对于业务来说很关键。但是我们可以这样想一想：如果你一个小时无法快递订单，对用户会造成什么样的影响呢？对你的业务会造成什么影响呢？大多数情况下，它对用户造成的影响非常有限——一个小时的延迟并不会影响用户接收到他们的包裹的时间。它可能对你的业务有一些影响，因为负责打包订单的雇员可能会在一段时间内无法工作。因为它对业务有一些不严重的影响，也不会对用户造成严重的影响，所以3级的标签是合适的。

订单周报（4级）

这是负责收集订单数据并为财务和管理人员生成每周业务报告的服务。

这是一个4级服务，因为它对用户体验不会造成任何影响。报告延迟一小段时间生成可能会对业务产生一些影响，但是不会很严重。

这个例子应该让你对如何为服务分配级别有了一定的认识。

现在你已经了解了各个服务分级，应该能够对应用程序中的所有服务使用适当的服务分级标签了。既然我们的服务已经标记了标签，那么如何使用这些标签，以及它们会带来什么价值呢？

使用服务分级

当为所有服务分配服务级别之后，如何在服务运维时使用它们呢？下面是几种常见的方式。

期望

服务期望的运行时间是什么？它的可靠性如何？它存在多少问题？它允许多长时间出现一次故障？

响应性

应当以多快的速度来响应问题？可以通过哪些途径来解决问题？

依赖

依赖你的服务，以及你所依赖的服务，它们的服务级别分别是多少？它们对你的服务有什么样的影响？

我们分别来看其中的每一点。

期望

服务的期望对于用户来说是很重要的一个部分。服务等级协议（SLA）是管理这些期望的一种方式。由于这部分非常重要，所以我们会在第8章用一整章的篇幅来深入讲解这个话题。

响应性

当系统中发生某个问题时，你对问题的响应性取决于以下两个因素：

- 问题的严重性
- 出现问题的服务级别

1级服务的高严重性问题应该比3级服务的高严重性问题更加重视对待。这很显而易见，但是如果1级服务发生了一个中严重性的问题，可能需要比3级服务的高严重性问题更快地响应。图7-5说明了这一点。

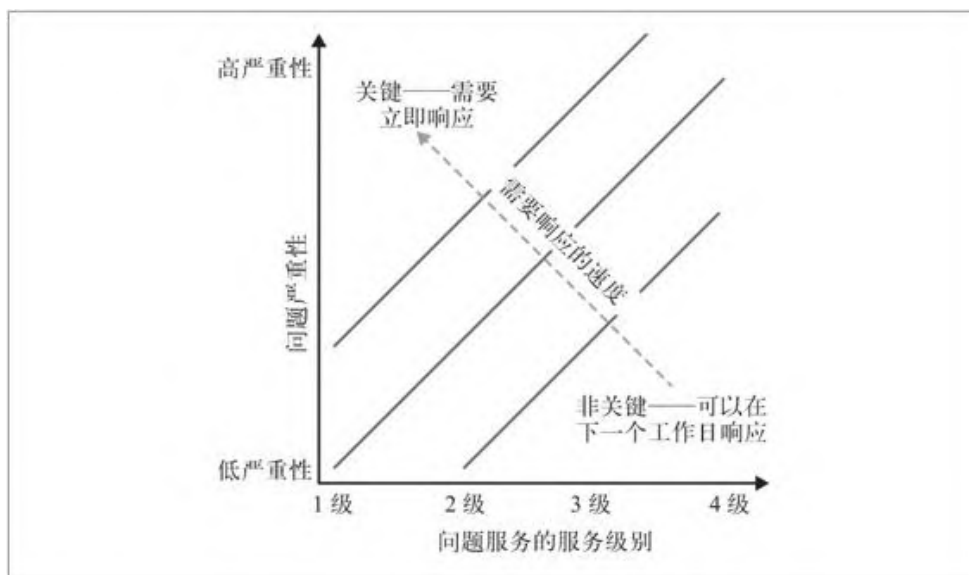


图7-5：不同服务级别和问题严重性需要的响应性

问题越严重，或者服务越重要（服务级别数字越低），那么就需要越及时、越紧急地响应。图7-5中的几条平行线表示相似的响应重要性。1级服务的中低严重性问题，可能需要与3级服务最严重的问题具有相似的响应程度。4级问题几乎永远也不需要紧急响应。

另外，2级服务的低严重性问题的响应程度，可能与4级服务的高严重性问题相似。

你可以借助图中的信息来调整响应性的各个方面。例如，你可以使用响应性级别来决定以下几件事：

- 哪些服务的哪类问题需要立即发送报警通知。
- 期望的SLA。
- 对于低优先级问题的上报路径。
- 提供响应的时间安排（24×7或者仅办公时间）。
- 是否提供紧急部署或者产品更改。
- 根据服务的可用性和响应性所制定的SLA。

依赖

在构建服务的时候，该服务被分配的级别与其他依赖服务的级别之间的关系非常重要。图7-6展示了不同服务级别之间的关系。

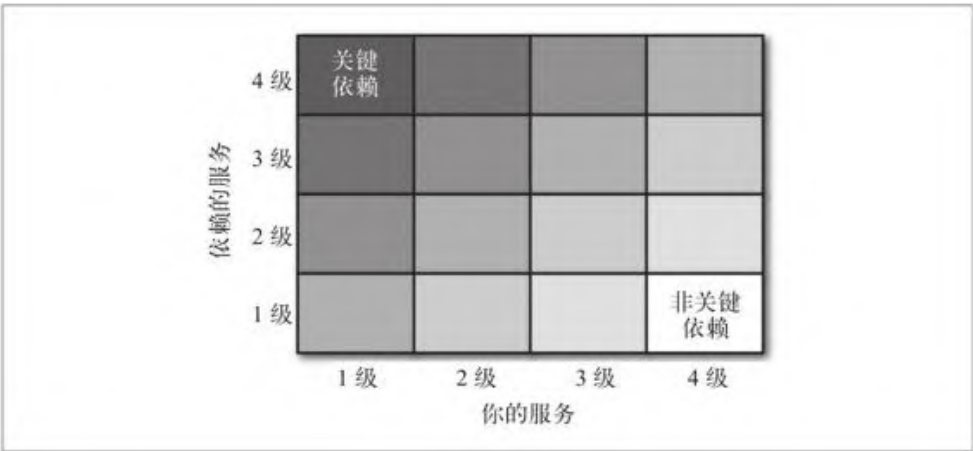


图7-6：服务依赖关系图

如果你的服务的级别高于（数字小于）依赖服务的级别，那么这个依赖就是一个关键依赖。如果你的服务的级别低于（数字大于）依赖服务的级别，那么这个依赖就是一个非关键依赖。

关键依赖

如果你确定你的依赖是关键依赖，那么作为一个服务的开发人员，你需要小心处理依赖发生故障的情况，避免对服务造成严重的影响。

如果关键依赖出现故障，你的服务应当仍然能够尽力提供功能。因为依赖服务的级别比较低（数字较高），意味着它无法拥有同当前服务一样的可用性和可靠性。

请你查看一下图7-4所示的系统，关注一下1级的网站前端服务。当这个服务试图向用户展示某个产品的详细页面时，它需要确定该产品的当前价格。为了做到这一点，它需要调用价格和快递费计算（PSCC）服务来获取商品价格。

如果PSCC服务（2级服务）不能提供服务呢？网站前端服务（1级服务）仍然必须尽可能地提供功能。那么，这个时候它应该怎么办呢？

它需要优雅地处理来自PSCC服务的失败消息（或者响应丢失）。一旦它确定PSCC服务发生故障，它需要立刻决定如何来显示产品详情页面。以下是一些备选项：

- 它可以在页面上使用之前缓存的价格（如果之前缓存过的话）。
- 它可以显示产品详情页面，但是不显示当前价格。它可以显示例如“无货”或者“当前价格未定”，或者甚至“添加商品到购物车中以查看当前价格”的消息。

用户仍然可以看到产品的图片、其他用户的评论，以及其他产品的细节内容。虽然用户体验下降了，但是他们仍然能够在你的网站上完成一些非常重要的操作。

我们称这种办法为优雅降级（已经在第5章中详细地介绍了如何处理服务故障）。

非关键依赖

如果你确定你的依赖是非关键依赖，那么你可以几乎忽略这个依赖的服务失败。

这是因为你的依赖服务拥有更高的服务级别（数字较低），因此也会拥有比你的服务更高的可用性和响应性。

我们再次以图7-4所示的在线商店系统为例，但是这次关注于订单周报这个4级服务。因为它需要获得生成报表的信息，它需要调用订单管理这个1级服务。

如果订单管理服务出现失败会发生什么呢？订单周报服务应该如何应对呢？其实订单周报服务也可以直接失败。因为订单管理服务是一个1级服务，它的任何问题都需要非常快速地被处理，因此响应性和紧急性都非常高一远高于对订单周报服务失败的处理。

因此，当订单管理服务出现故障时，根本不需要对订单周报服务做任何特殊处理，因为如果订单管理服务不可用，周报服务不可用也是正常的。

小结

服务级别为服务所有者、依赖服务和用户提供了一种展现服务关键性的便捷方式，从而让它们可以轻松地理解和沟通服务之间的各种期望。这种简易性降低了出错的概率，并且服务级别建立的简单模型，让人们和服务之间的沟通变得更加容易。

第8章

服务等级协议

服务等级协议（SLA）的核心内容就是期望管理。如我们在第7章所讨论的，每个人对服务的期望都是不同的。许多期望都与服务的级别相关，但是如果我们更深入了解的话，这些期望会更加具体。

本书中讨论的服务等级协议，并不是公司和用户之间的法律或合同协议，这是团队和服务所有者之间的协议，它们提供了一个沟通服务间期望的机制。

SLA与SLO

近年来，SLO（服务水平目标）这个术语已经被广泛使用。SLA和SLO之间的区别在于，SLA用来描述对外部客户的法律承诺，而SLO用来描述团队之间服务度量的目标。通过这些定义可以知道，从一个服务到另一个服务的协议（例如本章中所讨论的）更符合SLO的定义。从技术上讲，这是一种有效区分最新术语的方法。

然而，我不同意这点区别。因为从我的立场来看，这种区别通过一个似乎缺少承诺的术语（SLO）淡化了服务对服务承诺的重要性，SLO似乎比SLA描述的承诺更弱。这才是问题的核心。在我看来，一个团队的某个服务对另一个团队的另外一个服务做出的性能承诺，应该与对客户的法律承诺同等重要。因此，我将SLA一词用于客户协议和内部服务之间的承诺。

由于这些原因，在本书中，特别是在本章中，你可以放心地认为SLO和SLA这两个术语大部分时间是可以互换的。

在本章中，我们将讨论SLA，以及它在外部客户和内部客户中的使用。我们将讨论如何将SLA作为一种在各个服务团队之间获得信任的方法，以及如何使用SLA来解决团队之间的问题。

什么是服务等级协议

服务等级协议（SLA）是一个提供某种级别可靠性和性能的承诺，用来在服务所有者和用户之间创建一个牢固的合约关系。

例如，一个隔天快递服务的SLA可能是在第二天上午9点之前发送某个包裹。而一个航线的SLA可能是在航班到达后的30分钟内发放行李。一个电力公司的SLA可能是在多短时间内修复风暴之后的电力故障。

用户的期望

我们回想一下之前的章节，以图7-4中所示的在线商店系统为例。用户希望即使在不使用网站时，网站依然是运行着的——即他们希望网站是高可用的。他们还希望网站加载速度非常快，这样在使用时就不会有延迟。此外，他们还希望在网站上能买到想要的商品，希望库存充足且可以随时邮寄。最后，他们希望在支付订单后，能够在合理的时间内收到他们订购的商品。

通过“用户的期望”这个例子，每个期望都可以表述成一个SLA。例如：

可用性

用户希望网站随时可以访问。你可以将这一点描述成网站可运行的最小时间百分比。例如，一个可用性SLA可能会这样描述，“我们的商店至少99.4%的时间是可用的”。

加载时间

用户希望页面快速加载—即希望网站看上去是立刻响应的。你有很多方式来描述这一点，但是最简单的方式是用页面需要加载的最长时间—例如，“99%的时间页面会在4秒内加载”（请参考本章后面的“排名SLA”小节）。

产品

用户希望在商店中买到他们想买的商品，也期待这些商品有库存并且能够立即发货。你可以将这一点描述成一个百分比，例如，“至少分类中80%的商品都有现货”。

快递

用户期望他们购买的产品能够尽快到货。你可以用从下订单到发货的时间，或者以商品到达用户手中的时间来描述这一点。例如，“我们在24小时之内送达商品”。

以上所有这些都是SLA的例子。虽然它们之间存在很大区别，但是目的都相同，就是描述用户对系统的期待是什么样子的。

你可以在系统运行和与用户交互时测量这些SLA的实际情况，可以生成一些图表来展示不同时间的测量结果。但是SLA指标是服务正常运行的底线。图8-1展示了商品库存的情况，对指定时间拥有库存商品的百分比进行了测量。

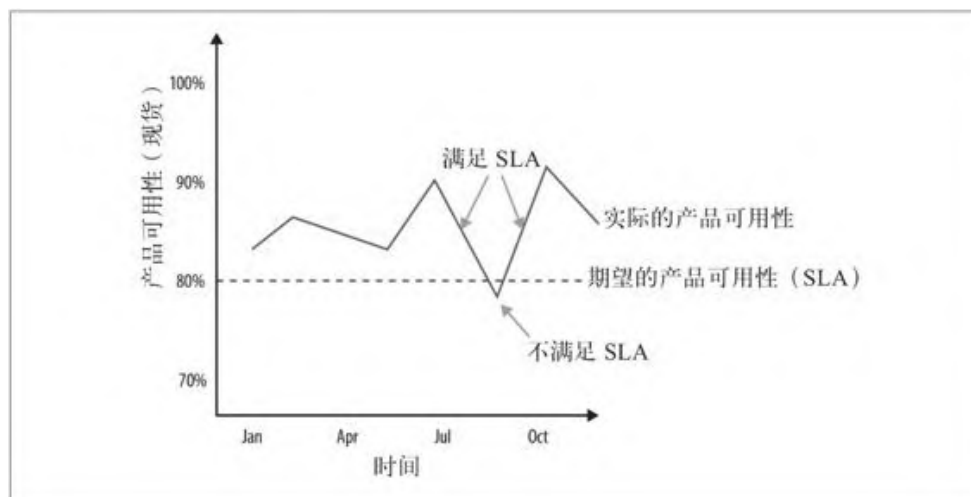


图8-1：SLA的比较情况示例

从图8-1中可以看到，现货商品的百分比会随着时间发生变化。还可以看一下SLA线，它表示你期望的情况是高于80%。

大多数时候，现货商品的百分比都是高于SLA的（我们称为满足SLA）。但是，夏末的某段时间内，它下降到了80%的SLA以下（我们称为没有满足SLA）。

外部的SLA和用户承诺

有时用户会与公司签订协议，要求满足一定的SLA要求，如果没有满足，可能会有一些赔偿或者其他规则。

例如，Amazon的AWS就与用户签订了SLA，如果它没能满足这些SLA，会提供一定的经济补偿。

以Amazon EC2为例，如果AWS的月运行时间低于99.95%，它会给受影响的用户提供10%的费用减免。如果低于99.0%，它会给予30%的费用减免。^[1]

通过SLA来监控系统的情况，对于内部业务使用来说非常有帮助（确保系统能按照用户预期的标准来执行）。或者，像AWS一样，SLA可以为用户提供资金方面的承诺。不管怎样，两种情况下的SLA和测量SLA的方法都是一样的。

外部SLA与内部SLA的对比

在“用户期望”和“外部的SLA和用户承诺”这两个案例中，主要介绍了外部的SLA。为了能够描述用户使用系统的情况，我们需要对这些SLA制定标准和进行监控。

但是SLA也可以用于并应该用于不同服务之间的相互沟通。这样，你就可以将SLA作为一个沟通机制，帮助服务的所有者和调用方之间沟通各自的期望和需求。

内部SLA很重要

对于复杂的多服务系统来说，内部SLA对系统的健康和可运维性至关重要。^[2]为什么这么说呢？显然，如果服务所依赖的服务没能达到SLA要求，你也就无法保证自己服务的SLA。^[3]

如果所依赖的服务只能够在90ms后返回响应，那你如何将响应速度提升到50ms呢？

如果所依赖的服务只能提供90%的可用性，那你如何让自己的服务提供99%的可用性呢？

SLA可以作为一种信任的手段

SLA的意义在于如何用高度分布和可伸缩的方式来建立服务之间的信任。当你相信某个依赖服务可以达到它的期望值时，你就有信心来设置自己服务的期望值。

建立信任

以图7-4所示的在线商店系统为例。假设你和团队负责价格和快递成本的计算服务。你的内部用户是网站的前端服务和结算服务。它们主要依赖于你的查找指定商品价格的功能。由于这些服务都使用该值来生成展现给最终用户的网页，所以它们需要迅速查找到相应的价格。你的团队同意保证每个查询价格的请求在20ms内返回。

现在，当你要满足这个承诺时，你意识到服务还需要快速访问分类存储服务，其中包含你需要计算价格的数据。但是，考虑到你已经给了20ms的承诺，你会担心分类存储服务是否能足够快速地提供所需数据，而分类存储服务是由另一个团队负责的。如何能确定那个团队有能力满足你的性能需求呢？你有两个选择。

第一个选择是联系分类存储服务团队，深入了解它们服务的工作原理，找到性能问题。然后，分析这个团队是否能够满足你需要的性能。当然，这种方式是高度侵入式的，成本昂贵并且在大型组织中不太可能实现。

另一个选择是与分类存储团队进行协商，在服务的性能SLA上达成一致。假设你通过与团队的沟通，它们同意每个响应小于10ms。你知道如果它们能够如此迅速地响应，你就可以达到对用户承诺的20ms。

只要它们能够保证自己的SLA，你也可以保证你的SLA。

你可以通过不断监控其他团队的性能是否达到SLA要求，来了解它们的运行情况。如果它们能够持续满足承诺的SLA，你就会对依赖的服务更有信心，从而可以将更多精力关注在自己的服务上，只需要保证继续满足对用户20ms的承诺。

SLA可以用于问题诊断

SLA还提供了一种检测复杂系统中是否存在问题的方式。如果某个服务正在经历各种问题，首先就要检查它的依赖服务是否满足了SLA。如果某个依赖服务没有满足自己应该达到的SLA，你就可以先从这点开始，逐步诊断服务发生问题的原因。

查找问题

以图7-4所示的在线商店系统为例。假设你和团队负责价格和快速成本的计算服务，如上一页的“建立信任”中所述。

现在，假设你某天半夜接到一个电话。你的服务在提供价格查询功能时变得缓慢，并且影响到了公司的用户。你开始检查服务性能是否能够达到20ms以内，但是发现平均每个查询请求要500ms才能返回。这肯定会影响到前台页面的响应速度，从而令用户感到不满。

但是，是什么原因导致的问题？是服务中出现了什么问题吗？还是其中一个依赖服务存在问题？

可能是你的服务出现了某些问题——可能是硬件或者其他方面。但是，在你打算花费大量时间试图去寻找服务的问题之前，最好先检查一下依赖服务的性能。

由于我们知道服务依赖于分类存储服务，并且与它们之间达成的SLA约定是10ms，所以我们检查该服务性能后发现它也存在着性能问题。调用该服务的请求不仅不能在10ms内返回，甚至超过了400ms。显然，该团队正在遭遇某个性能问题。你找到并拨打了这个服务的团队的联系电话，发现它们正在处理这个问题。

意识到这很可能是导致你的服务性能问题的原因后，你开始跟踪其他团队解决问题的进展。这比花费大量宝贵时间寻找自己服务的问题更有意义。

通过与依赖服务建立定义良好的SLA，当你自己的服务或者依赖服务出现问题时，你可以非常容易地定位问题发生的原因。

SLA的性能检测方法

有许多可用来检测服务性能的方法，但具体使用哪种方法，要根据服务的用户和所有人的需求来决定。以下是一些常见的性能检测方法。

调用延迟

这种方法用来测量服务需要多长时间来处理一个请求并返回响应。测量结果通常以微秒或者毫秒计算。服务的消费者必须知道一个请求的处理时间，因为这部分时间会计算在处理请求的总时间内。这就是之前章节中使用的SLA类型。

流量

这种方法用来测量服务在一段时间内能够处理多少请求。通常按照每秒有多少请求数量来测量，服务的所有者必须知道来自消费方服务的流量，才能知道自己的服务是否能够满足对方的期望。

运行时长

这种方法用来测量服务正常、无故障运行的时间。通常以百分比计算，它可以测量服务在一段时间内（通常以天、月或者年计）的可

用性。

错误率

它用来测量服务在一段时间内出现过多少次失败。通常以百分比计算，通过一段时间内失败的请求数除以请求总数得出结果。

限定SLA

限定SLA通常指期望满足的一个限定值。如果实际的性能好于限定值，则说明满足了SLA。如果实际的性能差于限定值，说明没有满足SLA。限定值本身就是SLA的值。

例如，“调用速率必须小于1000请求/秒”表示对服务预期流量的一个限定SLA。如果预期的流量小于指定的限制，则该服务已满足其SLA。

另一个例子是，“至少99.5%的时间服务是可正常运行的”表示对该服务的可用性要求。如果服务的可用性大于指定的百分比，则该服务已满足其SLA。

你可以在大多数性能检测中使用限定SLA。

排名SLA

当你可以测量某个值，并且可以确保该值始终好于限定值时，适合使用限定SLA。这种SLA更适合于描述可用性、运行时间和错误率。

另一种SLA测量方法是排名SLA。当某个事件的实际性能变化较大时，你可以用排名SLA来测量这个事件的性能。

排名SLA适合测量调用延迟等值。通常，服务处理每个请求的时间差异可能会较大，大多数情况下我们并不关心每个请求的处理时间，只要绝大多数请求可以在限定时间内处理就好。

排名SLA用一个高于或低于某个指定值的百分比数值来表示，通常的形式如下：

TP<百分比>低于<值>

另一个例子：

TP90低于20毫秒

我们可以将其读作“90%的请求在20毫秒以内处理”。

通常，我们需要计算某个事件的性能一例如，服务的调用延迟，并用排名的方式表示出来。

假设有一个可以响应外部请求的服务。在经过一段时间的观察后，我们记录下了这些调用的延迟时间，如图8-2所示。

Service Call Latency - Actual	
Req Time	Latency
T + 1 sec	5 msec
T + 2 sec	10 msec
T + 3 sec	20 msec
T + 4 sec	30 msec
T + 5 sec	15 msec
T + 6 sec	8 msec
T + 7 sec	12 msec
T + 8 sec	45 msec
T + 9 sec	12 msec
T + 10 sec	22 msec
T + 11 sec	4 msec
T + 12 sec	8 msec
T + 13 sec	12 msec
T + 14 sec	15 msec
T + 15 sec	14 msec
T + 16 sec	28 msec
T + 17 sec	21 msec
T + 18 sec	32 msec
T + 19 sec	15 msec
T + 20 sec	22 msec

图8-2：服务调用延时表

可以根据这些值绘制一幅如图8-3所示的图表。

通过这些数据，可以计算出该服务的一些最高延迟。

TP90

这表示90%的延迟都低于该值。在本例中，90%的数据就是18个数据点。除去最高的两个数据点（45毫秒和32毫秒），只剩下18个数据点，其中最高值是30毫秒。因此我们说TP90是30毫秒。

TP80

这表示80%的延迟都低于该值。在本例中，这意味着去掉20%的最高值（4个数据点：45、32、30和28）。在剩下的16个数据点中，最高的值是22毫秒。因此我们说TP80是22毫秒。

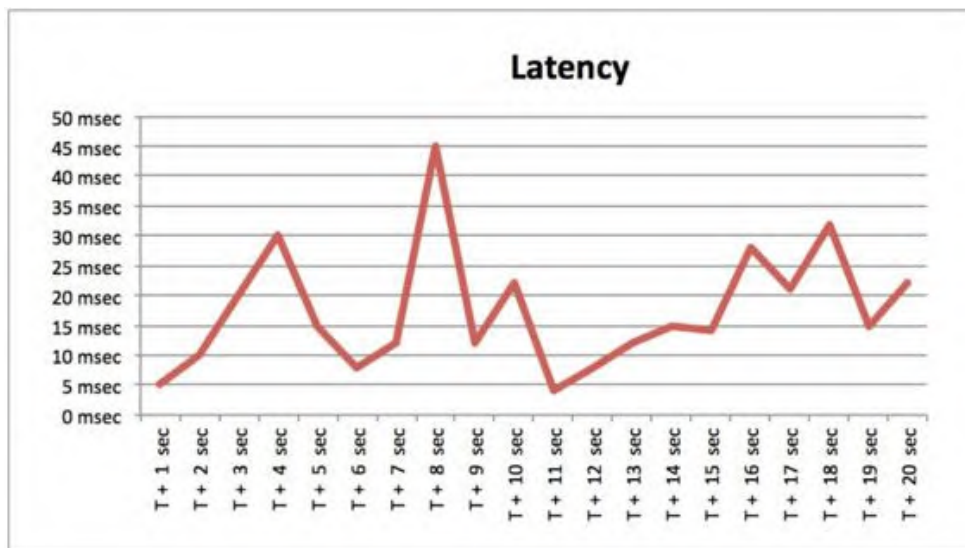


图8-3：服务调用延时图

以此类推，以下是一些可以从数据中计算出的其他TP值：

TP95=32毫秒

TP90=30毫秒

TP80=22毫秒

TP50=14毫秒

以下是其他一些可能会用到的值：

TP_{max}=45毫秒（最大值）

TP_{min}=4毫秒（最小值）

TP_{avg}=18毫秒（平均值）

这个排名显然会随着时间发生变化。当你计算出这些值之后，可以使用限定SLA来定义期望值。例如，在这个示例中，你的服务的SLA可能如下：

TP90<35毫秒

如果做到了这一点，那么服务就满足了自己的SLA。但是，如果它之前承诺的是如下SLA：

TP80<20毫秒

那么服务就无法满足自己的SLA（当前TP80是22毫秒）。

SLA的条件

SLA有时可能以另一种度量方式作为条件来表示。例如，某个服务可以保证一定的延迟，但是必须是在合理时间的前提下。因此，一个SLA可能有如下表示：

当流量<250,000请求/秒时，调用延迟TP90<25毫秒

在这里，为了满足我们的SLA，当流量低于250,000请求/秒时，调用延迟的TP90必须小于25毫秒。如果流量大于这个数字，那么调用延迟可以是任何值。

究竟应当定义多少内部SLA，以及定义哪些内部SLA

在创建服务时你可能想问，究竟应该为服务定义多少个内部SLA？

首先，应当保证尽可能少的SLA数量。当SLA的数量增加时，SLA的意义和相互影响会变得非常复杂。

你应当确保SLA覆盖了服务的所有关键部分，为所有的主要功能都定义合适的SLA，尤其是在对业务至关重要的地方。

你应当与服务的消费方一起来协商SLA，因为不能满足消费方需求的SLA就是一个无用的SLA。但是，尽量对所有的消费方都使用一样的SLA。服务应当尽可能用一套SLA来满足所有消费方的需求，因为为每个消费方都建立一组SLA，不仅会极大增加复杂性，而且不会带来任何实际的价值。

你应当只定义那些实际中可以实现监控和报警的SLA。如果你无法有效地验证SLA，定义它也没有任何意义。此外，应当关心服务是否违反了SLA，因为这应该是问题发生的最先表现，因此你需要确保当服务不满足SLA时可以第一时间接收到相关报警。

你可以对超出SLA的值进行监控和报警，并将在它们之上的值作为内部的SLA。这些数据在查找和管理服务问题的时候非常有用，同时又不会实际承诺给消费方。

你应当建立一个包含所有SLA和监控的仪表盘界面，这样一眼就可以发现正在发生的问题。并且你应当将这个界面共享给所有的依赖方，这样它们也可以看到你的服务情况。

除此之外，你需要确保可以访问到所有依赖服务的仪表盘界面，这样你可以监控它们是否正在发生问题，以确定问题是否会影响到你的服务。

为什么内部SLA很重要

监控和使用SLA会很快得到广泛应用，并且你可以很容易快速了解到SLA的各项监控细节。

在理想情况下，我们的目标不只是建立对所有SLA的监控，而是发掘一个可以用来对比的数值。任何数字都好过没有数字。内部SLA的目的不是为了增加数字，而是为当前服务和其他依赖服务提供指导，帮助在团队之间设置合理的期望。

内部SLA是扩展应用程序规模的一个关键组件，这样你在开发和管理应用程序时才可以利用更多的开发团队。它改进了复杂性的伸缩能力和整个应用程序的可用性。

SLA可以并且应该成为你与其他团队进行沟通的语言的一部分。

[1] 你可以在链接4所指示的网址了解到AWS如何计算SLA和费用的详细信息。

[2] 或者SLO。这就是本章前面描述的SLA和SLO之间的现代化区别。

[3] 请参考第6章有关团队服务所有权的信息。

第IV部分

原则4. 风险：现代化应用程序的风险管理

如果你无法识别系统中的风险，你就不可能管理风险。

……但是，同样存在“未知的未知”——即我们不知道自己不知道。如果纵观我们国家和其他自由国家的历史，往往困难的是后者。

——唐纳德·拉姆斯菲尔德（美国前国防部长）

所有的复杂系统都存在风险，这是所有系统不可避免的一个部分。我们不可能消除一个复杂系统（例如，一个Web应用程序）中的所有风险。但是，检查风险并确定对风险的接受程度，对于保持系统健康非常重要。

我们在本部分会整体介绍什么是风险，以及如何来识别风险。然后，会介绍风险管理流程，其可帮助降低风险所带来的影响。

现在让我们回顾一下第1章中的重大赛事例子。这里是一个简短的介绍：

周日有一个重大的赛事。

你邀请朋友们来到家里，一起在新买的电视上观看比赛。

比赛马上就要开始了，但是突然停电了，灯和电视都灭了。这场比赛对于你和朋友们来说也结束了。

你给电力公司致电，服务人员说：“非常抱歉，我们只能保证电网95%的可用性”。

示例中的电力公司就存在着风险，它们承受着重大赛事时停电的风险。

它们甚至还将风险进行了量化（只保证大约95%的电力）。

电力公司知道哪些事件会导致停电，例如，电线损坏。因此，为了保证电线的安全，它们通常会这么做：

- 将电线埋到地下（为了避免被风吹走）。
- 加固电线（为了降低被风暴吹断的概率）。
- 增加备用电力系统（即使一个系统出故障，另一个系统还可以继续工作）。

但是这些方案都是有成本的。是否值得投入资金加固电线？是否值得将电线埋入地下？相比风险所带来的损失，这些降低风险的投入是否值得？这些问题都属于风险管理问题，也是我们要讨论的问题。

我们将从描述风险管理的基础开始，包括两个非常重要的概念，可能性和严重性。

接下来，我们将介绍一个跟踪风险的基本工具，那就是风险模型。然后，我们将讨论降低风险的方法——比赛日，并且以一些可以用来构建风险较低的应用程序的想法作为结束。

第9章

如何在设计可伸缩架构时使用风险管理

风险管理包括确定系统中风险的位置，确定哪些风险必须消除、哪些风险可以暂时存在，以及如何降低这些留存风险发生的可能性和严重性。

当某个风险被触发（或者发生）时，你或你的系统会遭受损失。这些损失可能包括公司或者用户的数据丢失、无法给用户提供服务，或者是返回无效或错误的结果。不管怎样，这些损失都会让用户对你管理他们数据和业务的能力丧失信心，最终承受损失的只能是你自己。

但是，你必须从辩证的角度来评估这样的损失：投入的成本是否可以避免风险再次发生？

最终，风险管理就是在消除风险的成本与风险发生的成本之间保持平衡。

识别风险

风险管理的第一步就是列出所有已知的风险，以及它们的危害性和发生的可能性。

我们称这个列表为风险模型，稍后图9-1会展示一个风险模型的例子。

最初的风险模型的创建需要团队的头脑风暴。你可以从以下多个来源收集风险模型：

- 开发人员的头脑风暴
- 已知的重点售后支持问题
- 已知的安全风险和漏洞
- 已知的系统不完善或者缺失的能力
- 已知的性能瓶颈点
- 已知的流量峰值和变化
- 来自业务负责人、支持人员或者用户的特殊考虑
- 已知的系统技术债务

在这个列表里有一些显而易见的事项，也有一些你可能从未想到的事项。这是一件好事，因为你应该尽可能地去发现更多的风险威胁，如果你认为自己都考虑到了，有可能说明你还没有考虑得足够细致。

建立风险模型还包括为风险出现的可能性和造成的影响分配优先级。

消除最严重的风险

在你完成这个列表之后，应该从中确定出最严重的风险事项。如何知道哪些风险是最严重的呢？可以先找出经常会发生的风险，或者还没有发生但一旦发生就会造成严重问题的风险。当然，最严重的风险就是既经常发生又对系统有严重影响的风险。我们会在下一页的“可能性与严重性”一节中讨论严重性与可能性的区别，以及如何利用这一点来帮助找到最严重的风险，从而更好地管理风险。

在图9-1中，“如果用户身份服务停止服务会导致前端无法展现”可能就是最严重的风险了。

一旦你识别出了高危风险，请将它们加到风险计划中，并定期检查它们是否依然存在。

风险缓和

对于每个风险，不管它们是不是最危险的，我们都应当尽可能地去思考有没有降低风险发生概率或者风险影响程度的办法。这些办法被称为风险缓和措施。

风险缓和措施的价值很高，你需要重点发现那些既能够降低风险（不管是严重性还是发生概率）又易于实现或成本合理的措施。

我们以图9-1中的“如果用户身份服务停止服务会导致前端无法展现”的风险为例，一个可能的风险缓和措施是缓存某些用户身份信息，这样即使用户身份服务不可用，前台也可以将这些信息显示出来。

你可以重点关注那些最严重的风险，寻找降低这些风险的方式。但是，也要注意那些不太容易被快速修复的风险，找到降低这些风险严重性和发生概率的办法，几乎就等同于修复它们。

定期检查

如果你不定期检查风险模型，它就会慢慢失去作用。你的团队应当至少每个季度都对风险模型进行检查，对于非常活跃及极其关键的系统的风险模型可能需要每月进行检查。此外，在每次出现事故之后也应该重新检查风险模型，看是否存在已知风险能够正确覆盖所发生的事故？

当检查风险模型时，首先要检查最近新增或者确定的风险，并将它们添加为新的事项，同时移除已经不再存在风险的事项。

接下来，需要检查风险的严重性或者可能性的变化。通常，风险缓和措施会帮助降低风险的严重性和可能性，或者你了解到更多可能会增加风险发生概率或者严重性的信息。我们经常会遇到的情况是，在进行完上一次检查之后，风险却发生了，这时候你可能会将之前低优先级的风险标记为高优先级。此时，你需要考虑将它们加入风险模型中看是否可以消除（修复）这些风险？

最后，检查是否有新的或者更新后的风险事项需要被加入模型。

对风险管理的总结

如何管理系统中的风险？以下是一些你可以用来参考的基本步骤。

识别风险

首先，创建一个系统中已知风险的列表，即风险模型。对列表中各个风险事项排列优先级。

消除最严重的风险

找出列表中最大的风险，制订相应的解决计划。

风险缓和

对于无法暂时解决的主要风险，制订一个缓和计划来降低风险发生的可能性和严重性。

定期检查

定期检查你的风险模型。

可能性与严重性

理解严重性和可能性之间的关系非常重要。管理风险，你需要知道何时该考虑严重性而不是可能性，或者相反。理解两者的区别对分析系统中风险的重要性至关重要。我们认为所有风险都由以下两方面组成。

严重性

如果发生风险，所需付出的代价（例如，如果停电，用户会受到什么影响）。

可能性

风险发生的概率（例如，巨型风暴发生的概率）。

管理风险就是管理这两者，你可以降低风险的严重性或者可能性。对于所有已知的风险，你不需要双管齐下，但是同时考虑这两者对提高管理风险的能力非常重要。

风险的重要程度就是风险发生的严重性与可能性两者之和。如果要成功地管理风险，你必须同时考虑两者以及它们之间的关系。为了降低风险，你需要至少降低其中之一，或者严重性，或者可能性。

理解二者区别的最佳办法就是通过观察各种各样的风险，了解它们各自在可能性和严重性上的不同。我们将通过如下示例来了解二者之间的区别。

在线T恤商店

假设我们正在管理一个在线的T恤商店。这个商店是一个标准的在线零售商店，提供当前在售的T恤列表、每个T恤的详细情况页面（包括展示图片），以及一个用户用来购买、支付和快递的订单处理系统。

现在我们来了解一下这个在线商店有哪些常见风险。

十佳列表：低可能性，低严重性

在使用这个例子之前，我们先假设在线商店有一个展示售卖量排名前十的T恤的列表。访问网站的用户可以看到这些销售量最好的T恤，可快速、方便地查看商品详情并进行购买。

现在，如果这个十佳列表由于一些原因（也许是服务故障）无法展示，会发生什么呢？如果它无法展示，我们可以将它替换成一个静态的T恤列表，这些T恤不需要必须是卖得最好的前十个款式。这个服务很少出现故障，因为十佳列表很好生成，不应该出现什么问题。

那么关于这个十佳列表的显示，它的风险是什么呢？

我们来看一看这个风险：

- 风险的可能性很低，因为展示列表的服务非常稳定（我已经说过，这个列表很容易生成）。
- 但是如果列表没有显示，问题的严重性有多大呢？我已经提到，如果十佳列表没有显示，那么会显示一个代替的列表。虽然不够理想，但是它对用户的影响很小，并且对业务的影响也不会太大。因此，这个风险的严重性也是较低的。
- 所以，这是一个低/低的风险，意味着风险的可能性和严重性都很低。

像这样的风险很容易被忽视，通常也不会受到进一步关注，因为它很少发生，即使发生影响范围也很小。

订单数据库：低可能性，高严重性

在这个例子中，我们假设订单数据存储传统的关系数据库中。当用户生成某个订单时，会在数据库中增加一条记录；而当你处理、收集以及快递订单时，会更新数据库中的数据。随后，你会用这些数据来生成财务报表，为商业计划或者税务计算等商业用途提供帮助。

因为数据库很重要，所以你采用了带有备份功能（例如，RAID磁盘阵列）的高质量硬件环境。你还经常对数据进行备份。

但是，数据库仍然是一个故障单点。它包含了大量关键的商业数据，如果数据库不可用，则网站也无法提供服务（无法提交任何订单）。数据库故障会给企业带来极大的损失。

那么关于这个订单处理系统的数据库，它的风险是什么呢？

我们来看一看这个风险：

- 失败的可能性很低，因为我们使用了高质量的、带有备份的硬件。数据库本身非常可靠。

- 但是，数据库故障的严重性可能很高。因为如果数据库真的发生故障，整个订单处理流水线就无法运作，你会面临丢失关键商业数据的风险。
- 因此，这是一个低/高的风险，意味着它的可能性很低但是严重性很高。
- 像这样的风险很容易被忽视，因为它们很少发生（可能性低）。但是，忽视它们可能会带来严重的后果（严重性高）。

考虑到高严重性，你也许会希望采取措施来降低这个风险的严重性。例如，你希望使用一个从数据库，作为主数据库的实时副本，这样当主数据库发生故障时，可以快速让从数据库作为主数据库使用，避免大量时间和数据上的恢复工作。另外，你还可以采用分布式数据库，将数据分散到多台服务器上，这样即使其中一个数据库服务器发生故障，整个集群还可以提供服务。

不管使用上面提到的哪种方法，都可以将这个风险从低/高风险降低到低/中风险（低可能性，中严重性），甚至是低/低风险（低可能性，低严重性）。

我们会在本章后面的“风险缓和”一节来进一步讨论像这样能够显著降低风险严重性的风险缓和措施。

自定义字体：高可能性，低严重性

在这个T恤在线商店的示例中，假设你需要对所有文字和描述使用自定义字体，以便让网站看上去更加漂亮。你找到了理想的字体，不过它由第三方的字体服务方提供。为了使用这个字体，你会让用户的Web浏览器直接从第三方下载字体文件。如果自定义字体无法使用，浏览器会使用之前标准的系统字体。

但是，你注意到字体服务方有时会发生问题，且过于频繁。当字体服务方出现问题时，你的用户就无法使用好看的自定义字体了。

不幸的是，这经常发生。那么关于好看的自定义字体，它的风险是什么呢？

让我们来看一看这个风险：

- 字体不能显示的可能性很高，因为字体服务方经常会发生问题。
- 但是，当问题发生时，你的网站可以继续提供服务—只是看上去没有你想要的效果。因此，这个问题的严重性较低。
- 你的网站可能会看上去不那么美观，但是没有影响任何功能。
- 因此，这是一个高/低风险，意味着它发生的可能性高但是严重性低。

缓和这个风险的方法主要是降低问题发生的可能性。

你可以与第三方一起来改进服务的可用性，降低问题的发生概率。或者，你可以挑选出多个可以提供同样或类似字体的备份提供方，如果第一个提供方出现故障，则切换到其他提供方。这些都是你可以用来降低风险发生概率的手段。

由于该风险的严重性已经很低，没有必要再降低其严重性。

T恤图片：高可能性，高严重性

在这个T恤在线商店的示例中，我们来看一看网站上展现的T恤图片。因为用户如果无法看见T恤是什么样子的，是不会进行购买的，所以这些图片对于商店来说非常重要。如果网站无法显示出T恤图片，用户会选择离开，你也会损失订单。

但是，你用来存储图片的服务器却非常脆弱，它时常出现故障，并且似乎在磁盘读取图片方面存在问题。这台服务器很旧，需要被替换成新的服务器。它经常死机，需要定期重新启动，也经常由于需要更换部件而停机。但是不管怎样，你还是使用这台服务器来存储图片。

那么关于图片无法显示导致网站无法使用的故障，它的风险是什么呢？

让我们来看一看这个风险：

- 图片无法显示的可能性很高，因为服务器不稳定并且经常出现故障。
- 风险的严重性也很高，因为如果图片无法显示，用户会离开网站，也不会购买商品。
- 因此，这是一个高/高风险，意味着它的可能性很高（硬件经常故障）并且严重性也很高（用户不会进行购买）。

这种类型的风险最为严重，因为它发生的概率很高，并且会对业务产生严重的影响。

你最应该关注这一类风险。

这个示例可能看上去显而易见，但是应用程序中会存在很多类似的高/高风险。不过，通常这类风险都隐藏得很深，你只有仔细审视系统才会发现它们，这也是为什么风险管理如此重要的原因。

风险模型

管理风险的第一步是理解系统中已有的风险。识别、标记并对已知的风险排列优先级，这就是风险模型所要做的事情。

风险模型是管理系统中风险的一个关键方面。它是一张包含了系统所有已知风险的当前状态的表格。

图9-1列举了一个风险模型的示例。

Risk ID	System	Owner	Description	Date	Severity	Mitigation Plan	Status	Comments
1	Inventory	Team	Requires user identity service to function. Fails and fails if service is down.	10/13/11	Low	high	Open	5/26/16, fix
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								

图9-1：风险模型示例（请参考以下详细列表）

模型中的每一行表示系统中一个独立的、可量化的风险。表格中的各列包含了具体风险项的详细描述。

风险模型包含了以下风险项。

风险ID (Risk ID)

这是每个风险的唯一标识。它可以是任何类型的，但通常选择唯一的整数标识，这是最简单的方式，也能够满足需要。[\[1\]](#)

系统 (System)

这是包含风险的系统、子系统或者模块的名称。这个值取决于你对应用程序的定义和划分，通常会类似“前端”“主数据库”“服务A”的值。

所有人 (Owner)

负责该风险的个人（或者团队）的名称，所有人同时也负责制订该风险的缓和计划和解决计划。

风险描述 (Description)

该风险的概要描述。它应该尽可能简短，便于查看，但又不能太短，以便唯一、准确地标识该风险。

日期标识 (Date Identified)

识别该风险并将其添加到模型中的日期。

可能性 (Likelihood)

表示该风险发生的可能性（分低、中、高三级）。我们在本章前面的“可能性与严重性”一节中已经详细讨论了这个值，你可以使用这个值对风险模型进行排序，确定哪些是最需要关注和立即解决的风险。[\[2\]](#)

严重性 (Severity)

表示风险发生的严重性或者影响（分低、中、高三级）。我们在本章前面的“可能性与严重性”一节中已经详细讨论了这个值，你可以使用这个值对风险模型进行排序，确定哪些是最需要关注和立即解决的风险。

风险缓和计划（Mitigation Plan）

这一列描述了可以使用的或者正在使用的用来降低该风险严重性或者可能性的风险缓和措施。

状态（Status）

该列表示该风险的当前状态。这个值通常是“活跃”“已缓和”“正在修复”或者“已解决”等内容。

ETA

表示该风险距离计划解决日期（如果有的话）的估计时间。

监控（Monitoring）

该列表示你是否对该风险的发生进行了监控，如果是，则包含你实现监控的步骤。如果没有监控该风险，应该标明原因，以及计划对其进行监控的估计时间。

触发计划（Triggered Plan）

该列表示如果该风险真的发生，你计划如何来处理它。触发计划通常是一个管理层面的计划，而不是一个事件-响应计划。[\[3\]](#)

备注（Comments）

使用该列来记录任何不适合记录在风险描述中，或者不属于风险描述的信息。

除此之外，你还可以根据公司情况，在模型中添加你认为重要的列，例如：

跟踪ID (Tracking ID)

如果通过一个缺陷跟踪系统或里程碑跟踪系统来管理风险，可以用该列来记录该风险在系统中的跟踪ID。

历史 (History)

该风险在过去是否发生过？什么时间？发生频率？等等。

风险模型的作用域

现在，你可能在想“我应该为整个公司建立一个风险模型，还是应该分别为每个团队或者服务建立一个风险模型？”

这是一个好问题。对于小型公司来说，整个公司可以使用一个风险模型，但是它很快会变得相当笨重。每个服务一个风险模型可以在服务层面提供良好的可见性，但是降低了公司层面的可见性。像“哪个服务的风险对于公司最重要？”这样的问题就变得难以回答。

我推荐为每个团队提供一个风险模型。因为通常由团队来决定要完成的功能或问题，以及它们的优先级，所以也应当从团队层面来管理风险模型，由团队来决定其中各项风险的优先级。你可以在第6章了解更多从团队层面来管理风险的内容。

最后，你应当根据组织架构来设定风险模型的作用域。例如，每个团队、小组或者组织各自使用一个风险模型，来管理自己的工作范围和优先级。它们可能从上级管理部门接受任务和指导，但是会自己来决定大多数工作的优先级和执行内容。

创建风险模型

首先，你需要借助于一个风险模型模板。我已经为你创建了许多流行的表格工具的模板，你可以在我们的网站（参见链接5）上下载它们。

虽然你可以随意修改它们，但是对于第一个风险模型来说，最好还是尽量不要修改。当你在使用模型和管理风险方面有了一些经验之

后，可以再将模板修改成你想要的样子。

为了向你说明如何使用它，这个模板中提供了一条风险的示例，如图9-1所示。你可以在开始之前删掉这条风险。

通过头脑风暴建立风险列表

当你准备好模板后，第一步是通过头脑风暴得到一份风险列表。你应当试着把所有你能想到的风险都列出来，而不仅仅是那些你关心的风险。在这个过程中不要去分析它们，只要把它们尽可能都列出来就可以了。

以下是一些可以进行头脑风暴的很好来源。

开发团队

与你的开发团队召开一次会议。团队成员会对他们的服务有大量的担忧，仔细聆听他们的担忧，并将这些担忧作为风险项添加到列表中。

售后支持

看一下你的售后支持情况。是否存在高于正常支持压力的问题？你的售后支持人员都在说什么？你有没有可以查看的售后支持论坛？需要大量支持的方面通常都会产生系统风险。

安全威胁

思考一下已知的安全威胁和漏洞。不管它们有多么严重或重要，它们都是服务所面临的风险。

待完成的功能列表

浏览一下你待完成的功能列表，是否存在系统健康所缺失的关键能力，尤其是仔细检查与监控和维护有关的功能。

性能

思考一下系统的性能情况。有没有性能较差的地方？

业务负责人

与业务负责人进行一下交谈。他们有什么顾虑？

相关团队

与你的服务的相关团队人员进行一下交谈，包括内部用户、附属团队、Q/A团队等。他们都有什么想法？

系统和流程

你有没有为系统和流程编写应有的文档？是不是遗漏了一些与系统功能相关的必要文档，或者只是存在于少数几个人的脑袋中？

技术债务

你的团队中有没有已知的、明确的技术债务？这些技术债务包括难以理解的、过于复杂的，或者过于冗余的代码。已知的技术债务通常都是风险项。

你很可能发现列表中有些显而易见的风险项，但也会有些令你感到惊讶的项目。这很好，你会希望发掘尽可能多的风险漏洞，如果没有任何令你惊讶的风险项出现在列表中，有可能说明你考虑得还不够周到。

填写可能性和严重性字段

现在，遍历整个列表，填写每一项风险的可能性和严重性字段。对于这两个字段，可以使用低/中/高（或者类似）的值。

确保你能够准确区分可能性和严重性的概念。如果有任何不清楚的地方，可以重新阅读本章前面的“可能性与严重性”一节。通常，在这个步骤中很容易将可能性和严重性搞混。

先填写可能性，再填写严重性，这样可能会更容易一些。记住，风险项一旦发生会非常严重、但是几乎不可能发生的情况非常常见（同样，也可能非常容易发生，但是发生后却没有什么严重影响）。最终，你会得到许多状态为低、中、高的风险项，这是正常的。

不管你如何来完成它，如果混淆了可能性和严重性，那么这份列表也就失去了它应有的意义。

与你的开发团队进行另一次头脑风暴有助于完成这个任务。这次头脑风暴应该与之前提到的分别进行，不要在识别风险的头脑风暴中同时分析可能性和严重性。

风险项详情

现在，你可以填写风险模型中其他的基本信息，包括系统、所有人、日期和状态列。确认你为每个风险项都分配了一个风险ID（一个从1开始的数字即可）。

你有没有监控某个风险？“监控”列中的值表示你是否愿意在该风险被触发时接收通知。

缓和计划

你需要从严重性最高的风险项开始，制订它们的风险缓和计划。然后再从可能性最高的风险项开始。

缓和计划，指的是你现在或者即将准备为降低风险严重性或者可能性所采取的措施。缓和计划并不是要消除风险，它只是降低风险的严重性和可能性。

当你执行完缓和计划中指定的方案后，风险的严重性或者可能性应该发生下降，这时可以将该缓和计划删除。如果需要，可以增加新的缓和计划。

你不需要为风险模型中的每个风险项都制订缓和计划。有些风险项是明显必须被解决的，不能只是降低可能性或者严重性。此外，有些低可能性、低严重性的风险也不需要再为其制订缓和计划。

触发计划

触发计划指的是当风险真正发生时你需要采取的措施。这可能会像“修复缺陷”这样简单的描述，也可能是更详细的描述。例如，如果发生了某个风险，是否有一些可以立即执行并降低影响的措施？如果有，它们也应该被加入触发计划。

从严重性最高的风险项开始，为每个风险项制订相应的触发计划。



注意，触发计划不能用来代替事件响应文档，例如应急手册。风险模型不是在事件响应中必须使用的一个参考工具，它（包括触发计划）应该是用来决定风险发生后所采取行为的一个管理工具。

使用风险模型来制订计划

当你创建了风险模型后，所有的计划会议都应该参考它。这不仅包括产品管理的长期计划会议，也包括与工程师一起进行的敏捷开发计划会议。

在每个计划会议中，应该检查最重要的一些风险。[\[4\]](#)会上应当询问的问题列表如下所示：

- 与上次检查相比，这个风险现在是不是更严重了？
- 是否应该在这次计划排期中，安排人员来解决某些风险？
- 是否应该在这次计划排期中，安排人员来执行缓和风险的措施，降低它们的可能性或者严重性。

每个计划会议都应该包括对风险模型的检查，并且风险模型中的每一项（不管是解决风险还是缓和风险）都应该与其他工作一起排列优先级。

如果你的团队在计划会议中使用Jira或者Pivotal Tracker这样的工具，你可能希望将最关键的风险添加到这些跟踪工具中。如果是这

样做的话，你应该在跟踪工具中添加风险ID这一项，同时在风险模型中添加一列跟踪ID，记录这些风险在跟踪工具中的ID。

维护风险模型

风险模型的最大挑战是模型本身很容易过时。我们的天性会让我们建立完模型后，将它扔到某个抽屉中并遗忘它。

如果你不花时间来维护风险模型，那么它很快会变得过时和无用。

为了始终保持风险模型的内容及时准确，你应该与相关权益人（包括开发团队和合伙人）定期对它进行检查。这个时间可以是每月，但不应该超过一个季度。确切的重复周期应该根据你的业务流程来制定。如果你马上要进入一个计划周期，那么在周期开始前进行一次定期的模型检查是最理想的情况。

检查风险模型的参与人

注意，你应该定期变换检查风险模型的人员。

通过要求不同人员来检查并评价风险模型，你可以获得崭新的视角，也可以避免让定期检查会议变得毫无意义。

在检查中，你需要：

发现新风险

系统中有没有增加新的或者近期识别出来的风险？确保它们都被涵盖在风险模型中。[\[5\]](#)

删除旧的风险

模型中的某些风险是否已经不存在——因为它们无法再现，或者引发的原因已经被修复？如果是，从模型中删除这些风险项。

更新可能性和严重性

发现可能性和严重性发生变化的风险项。通常，近期实施的风险缓和计划有利于降低风险的可能性或者严重性，或者收集到其他影响可能性或者严重性的信息。如果有，请更新这些风险项。

检查优先级高的风险

检查所有高可能性或者高严重性的风险。分别对它们进行讨论，确保所有信息都准确无误。是否可以制订新的缓和计划，或者有任何缓和计划需要更新？关于触发计划呢？你是否对风险进行了监控？如果没有，为什么不监控？你还能做哪些事情来改善风险情况？

检查优先级低的风险

如果时间允许，继续检查可能性和严重性低的风险。你不用每次都检查每项风险，但是需要确保经常查看高优先级的风险。除此之外，你还可以单独安排一个会议来详细检查所有低优先级的风险，确保它们没有被人遗忘，以及确保不会隐藏或错过任何应该提升它们优先级的机会和原因。

与管理团队共享你的风险模型

你应当与产品管理和上级管理团队分享你的风险模型。这对于那些无法每天直接与之交流的团队来说，是一个有效的沟通工具，能够帮助大家在某些特定问题上达成一致的认识。

最近我发现在开管理大会之前，有一个很不错的做法。指定某人收集全公司的所有风险模型，将它们整合到一张大表中。然后，只保留高可能性或者高严重性的风险，其余的都删除。在整个管理大会上，根据这个核心的“高/高”风险列表来讨论公司所有产品的风险，统一不同团队对风险模型的理解，并积累最佳实践经验。

风险缓和

风险模型中的风险缓和计划列用来说明可以进行或者正在进行哪些风险缓和措施，来降低当前风险的严重性、可能性，或者二者皆有。我们要做的就是把一个高/高风险降低到一个高/中风险或者中/高风险。^[6]我们不是要完全消除风险，只是降低风险发生的可能性和严重性。

你可以按照一个基本流程来缓和风险。缓和计划详细描述了你要（立即或者即将）为降低风险可能性或严重性所做的事情。

风险缓和是为了知道当问题发生时应该做什么事情，能够尽可能降低问题所带来的影响。缓和措施是为了确保即使在服务或者资源发生故障的时候，系统也能够尽可能完好地工作。

我们先以一个缓和计划为例进行说明。假设有一个系统中使用的数据库，我们再进一步假设，这个数据库运行在一个高质量的硬件上，并提供了RAID磁盘阵列或者服务器级别冗余硬件的备份功能。我们相信这个数据库是高度可靠并且高度可用的。因此，在我们的风险模型上，将数据库失败标记为低可能性。

但是，数据库仍然存在单点故障的可能。如果这个数据库服务出现问题（虽然可能性比较小），你的整个系统都会受到影响。因此，在我们的风险模型上，它的严重性是高。

于是，这个风险是一个低/高风险，非常类似之前所描述的场景。

如何能缓和这个风险呢？首先，一个办法是添加多个读副本数据库，以图9-2所示的热备份的形式运行。如果主数据库出现故障，热备份数据库可以极大降低系统停止服务的总体时间。该办法可以降低风险的严重性，甚至可能将它降为一个低/中风险。

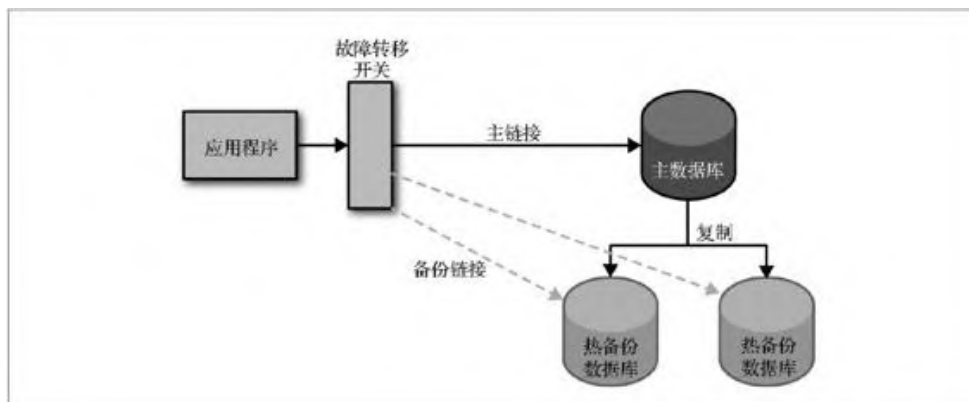


图9-2：用于缓和风险的热备份数据库示例

这就是一个缓和计划。

那么风险缓和和风险管理之间的区别是什么呢？它们有着类似却不同的概念。

风险缓和

风险缓和指的是通过降低风险发生的可能性，或者降低风险发生时的严重性，来降低风险的影响。

风险管理

风险管理指的是在解决风险和缓和风险之间做出选择。我们需要考虑是否能够经济、及时地解决风险，还是仅仅只需降低风险的影响即可。

恢复计划

如果已知的风险真的发生了，你必须处理它的后果。你可以通过一个恢复计划建立一系列操作，处理和修复风险所造成的影响和问题。

恢复计划通常不会影响可能性，只会影响风险的严重性。

恢复计划是风险缓和的一种特殊类型，专门用来降低风险发生的严重性。恢复计划描述了一个已知风险发生后你应该采取的行为，包

括以下几种：

- 尽可能迅速地停止导致问题的行为。
- 降低问题影响的临时行为。
- 通知用户问题情况，以及他们可以采取的降低影响的行为。
- 上报问题的流程，以及需要通知的公司内部人员（这能够让公司统一了解问题情况，一起处理问题和任何可能带来的波及影响）。

一个好的恢复计划，一定是作为风险缓和计划的一部分提前制订的，因此当问题真的发生（触发）时，每个人都知道应该做什么来恢复问题。

恢复计划应该包括以下内容：

- 触发执行恢复计划的必要前提。
- 执行恢复计划需要涉及的人员列表。
- 执行恢复计划的详细步骤，以及由谁来执行哪些步骤。
- 需要通知的管理人员、上级人员。
- 问题解决后必要的跟进措施。

恢复计划应该保存在团队都知晓的地方，即当危机发生时每个人都知道在哪里能找到它。这个地方可以是支持手册或者内部支持网络上。当执行某个恢复计划之后，应该对故障进行事后分析，同时也应该对恢复计划进行分析，以决定是否存在任何改进的措施，或者确保所有改动都经过批准。

如果针对某个具体风险的恢复计划是有效的，那么就说明这个风险缓和计划也是有效的，可以用来降低指定风险的严重性。

恢复计划

对于一个灾难性的数据库故障来说，图9-2所示的复制过程就是一个恢复计划的开始。但是，要形成一个完整的恢复计划，还需要引入一个实施故障转移的过程，确定何时进行故障转移的标准，执行故障转移的确认流程，以及在故障转移后的清理工作。

容灾计划

容灾计划是一种恢复计划，主要用来描述当某种灾难发生时公司应该采取哪些措施。这类灾难通常严重性很高，但是可能性很低。

一个符合容灾计划的灾难例子就是，一个或多个数据中心不可用（不管是由技术问题、自然灾难还是严重安全漏洞导致的）。

你可以像建立恢复计划那样建立和管理容灾计划。二者之间唯一的区别在于要缓和的风险的严重性、细节程度以及执行计划的投入。

一般来说，在公司和管理团队内部都会更加重视容灾恢复计划。针对这些灾难类型，可能需要提前安排具体业务的恢复时间。但是，这些都不是与恢复计划显著区分的地方。

改进我们的风险状况

风险缓和通过降低系统中风险的影响，已经成为提高系统可靠性和可伸缩性的一个重要途径。它也说明，现实中的风险不一定都能够解决，但是降低它的影响或者严重性却很有可能，而且通常这也足够达到我们期望的可用性程度。在与风险模型一起使用时，风险缓和计划提供了一个有用的工具来改进系统的健康状况。我们会在下一章进行介绍。

[1] 但是这个ID不应该是表格中的行号。因为模型中的行可能会进行排序、新增或者删除，这些都会改变某些风险在模型中的行号。风险ID应该是在跟踪风险的整个过程中不会发生变化的唯一标识。

[2] 为了能够方便地对可能性和严重性的值进行排序，你可以使用数字1、2、3来代表由低到高的程度。通常，我们会按照“1-低”“2-中”“3-高”的规则，并利用表格编程的能力来限制该值只允许有这三个选择。

[3] 事件-响应计划应该是已经制订好的，值班人员在事件应对手册或者其他工具中可以找到它们。

[4] 最关键的风险指的是严重性最高、可能性最高，或者两者都很高的风险。

[5] 但是，建议你一旦确信发现了新的风险，就立刻将它添加到风险模型中，而不要等到下一次的~~风险~~回顾会议。你可以在回顾会议中更新风险的所有数据，但是一旦发现就应当立即将它标注出来。

[6] 或者更低的其他组合，例如从中/高变成中/中、中/低或者低/低。

第10章 比赛日

我们经常犯一个危险的习惯性错误，就是建立恢复计划和容灾计划后，将它们搁置在抽屉里，只有在需要的时候才会想起它们来。

如果你真的这样做，我几乎可以保证，当你需要恢复/容灾计划的时候，它们已经过时了。除此之外，如果你不保持更新它们，就可能引入大量其他的问题，导致计划无法执行，或者在实际中无法成功执行。

因此，你应该定期对恢复/容灾计划进行测试。应该把定期测试这些计划和其他风险缓和措施，作为你的公司文化的一部分。

我们用来测试这些计划的方法被称为举行比赛日。“比赛日”指的是通过测试来触发系统中某个失败模型，然后观察你的操作人员和工程师如何进行响应，包括他们如何执行恢复计划和容灾计划。在比赛日过后，团队通过事后复盘来发现计划中的问题并进行后续改进。这些改进会不断更新恢复/容灾计划，以便当问题真正发生时可以派上用场。

预发布环境和生产环境

你可能想知道，是否应当在预发布环境或者生产环境中来测试恢复计划。这个问题很难回答，也没有标准答案。我们先分析一下以下这些选项。

预发布/测试环境

在预发布/测试环境中测试恢复计划是最安全的。预发布或者测试环境允许你执行一些无法在生产环境中执行的破坏性测试。另外，你不用害怕这些测试可能造成的失误。如果你决定通过预发布/测试环境来测试恢复计划，请记住以下内容。

- 确保预发布/测试环境与生产环境是完全隔离的。这个环境不应该依赖任何生产资源，生产资源也不应该依赖任何测试环境的资源。请参考图10-1。
- 确保预发布环境和测试环境与生产环境尽可能保持一致。虽然使用预发布/测试环境来测试恢复计划是可行的，并且你也可以使用这类环境来测试各种破坏性的故障场景，但是，它们并不能保证在生产环境中一定得到相同的结果。

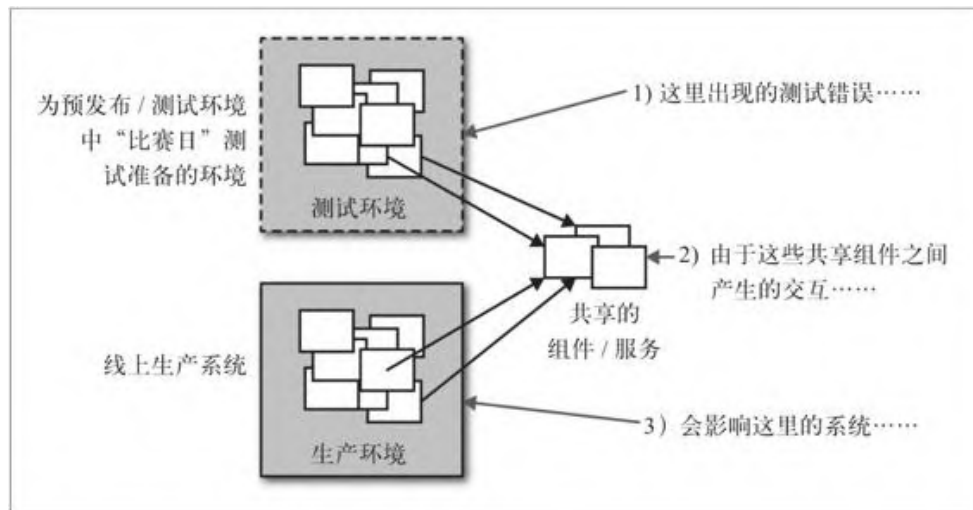


图10-1：不隔离测试环境的危险性

这是因为生产环境的服务器规模更大，承载着更大的数据集，并且管理着远超其他环境的实时流量。

这些差异使得某些测试在非生产环境下变得无效。

在理想情况下，测试环境应该达到跟生产环境一样的规模，并且使用与生产环境一样的数据，但是，这通常在成本和维护方面无法实现。

如果你认为某个测试必须与生产环境规模一样的情况下进行测试，那么也许应该考虑在生产环境中进行测试。

生产环境

在生产环境下测试风险和恢复计划看上去没有道理。为什么要强制让生产环境出现一次故障，只是为了确定它不会发生故障？答案很简单：你在团队就绪并且准备充分的情况下（换句话说，不是在午夜），同时在一个对用户影响最小的时间点上，在仔细考虑测试的每个步骤后，可以安全地在生产环境下进行实际测试，从而了解恢复真正故障所要付出的努力。

如果你决定在生产环境中测试你的恢复计划，请记住以下内容：

- 认真考虑对当前正在使用系统的用户造成的影响。
- 从业务角度考虑测试。是选择给用户增加风险，还是选择根据这些测试来降低长期风险，这需要权衡。
- 选择你的团队成员状态最好的时刻（通常是工作日白天大家都在办公的时候）来进行测试，但是也要选择对用户影响最小的时刻（例如，选择流量较低的时候，而不是在月底或者季度末销售高峰等关键时刻）。
- 确保你能够快速、简便地部署所需更新，或者回滚失败的更新。

在生产环境中举行比赛日的担心

你应当对在生产环境中进行比赛日很小心地进行计划和监控。如果计划恰当，生产环境中的比赛日可以非常好地暴露生产环境中的问题。以下是一些你可以在生产环境中举行的比赛日示例。

服务器故障

当系统中某台服务器发生故障时会发生什么呢？试着让一台服务器无法工作。如果你的系统拥有充足的备份机器，这可能对生产系统

不会造成任何影响。你可以使用排除法来发现这类问题，并利用恢复计划来替换故障服务器。

网络分区

如果发生网络故障或者网络分区会产生什么影响呢？如果经过仔细计划，你可以在避免对生产环境系统造成重大影响的情况下，模拟网络分区的情况。它们可以用来测试系统的通知和跟进行为，以及你的团队对事件的响应能力。

数据中心故障

如果整个数据中心出现故障会发生什么呢？如果经过仔细计划，你的系统应该能够处理这类事件。你会如何响应这类故障呢？

随机故障

如果系统中存在更小的随机错误会发生什么呢？你的应用程序是否能从这些错误中适当地进行恢复？

从很多方面来看，最后一项都是最令人害怕的。为什么？因为你可以想象出当服务器或者数据中心发生故障时会发生什么。你可能已经制订了处理这些情况的计划（如果你还没有，应该立即动手制订）。但是对于一个“随机”问题来说，即使范围很小，也像是一些你无法掌控的东西。说实话，它的确是。但是，正是这些随机事件，成为你在构建高可用、低风险系统时最大的绊脚石。

捣乱的猴子

Netflix将随机故障问题上升到了一个新的高度。它们开发了一个称为“捣乱的猴子”的系统放到整个应用程序中。该系统可以随机或定期地在生产环境中以及用户使用过程中造成随机故障。对于管理应用程序的工程师和运维人员来说，他们并不知道“捣乱的猴子”做了什么。它假设工程师们已经准备了恰当的恢复和缓和流程，因此它所造成的这些故障，应该在对用户没有造成影响的前提下被解决或者临时处理掉。

“捣乱的猴子”只在工程师在岗的时间发生，这样工程师可以响应任何无法自愈的问题。“捣乱的猴子”背后的哲学理念是，为了鼓励并且实际要求去构建高可用的、可自愈的服务和应用程序，可以在无须任何人工干预的情况下自我恢复。它选择在白天工程师在岗的时候进行测试，是为了避免问题发生在系统更加繁忙（用户更多）的晚上，临时找不到负责的工程师。这个新的尝试在Netflix已经取得了成功。

“捣乱的猴子”是比赛日测试的一个最佳实践范例，并且Netflix也对其比赛日的架构完成了一些非常令人称赞的事情。

但是，它需要巨大的付出、巨大的资源，以及在Netflix能够满足安全、有效地在生产环境中运行“捣乱的猴子”之前，巨大的代码修改工作。

因此，“捣乱的猴子”不应该是你进行生产环境比赛日测试的第一步。但是，如果你的公司有足够的能力和付出，可以将它作为一个发展的目标。

小结

比赛日测试，是一个能够帮助生产环境在系统层面保证完全运行的重要手段。它允许你通过一种安全的方式，来验证自己的支持计划和流程在问题发生时，是否能够真正正确无误地运行。

如果一切都完成得很好，比赛日测试可以极大改善系统的可用性，并降低生产环境中存在严重问题的风险。

第11章

构建低风险系统

在第9章中，我们了解到如何缓和系统中已有的风险。但是，还有一些技巧可以帮助你主动构建低风险的系统。本章会来介绍这些技巧中的一部分。

技巧1：介绍冗余

冗余可能会让你避免出现停机的问题，但是这可能会以系统复杂性为代价。

技巧2：理解独立性

知道组件为什么要具有独立性，以及理解服务、资源和系统组件之间的（有时是隐含的）依赖关系，是非常重要和有用的。

技巧3：管理安全性

坏人已经成为可用性问题的常见原因，并给现代应用程序带来了巨大的风险。[\[1\]](#)

技巧4：鼓励简单性

复杂性是稳定性的大敌。应用程序越复杂，就越容易出现問題。

技巧5：自我修复

即使出现了问题，你的修复过程越自动化，问题对客户造成的影响就越小。

技巧6：标准化运维流程

业务方式的变化可能会引入风险，并最终导致可用性問題。标准化、文档化和可重复的流程减少了手动错误导致停机的可能性。

这并不是一个完整的列表，但是至少可以在你构建和伸缩应用程序时考虑如何降低风险。

技巧1：介绍冗余

冗余是一个提高应用程序可用性和可靠性的显著方式，同时也能降低系统的风险状况。但是，冗余会给应用程序增加复杂性，这又增加了应用程序的风险。因此，重要的是控制增加冗余的复杂度，以及实际中是否能对风险状况带来可测量的改进效果。

以下是一些“安全”的冗余改进示例：

- 将应用程序设计为可同时安全运行在多个独立硬件上（例如，并行服务器或者冗余数据中心）。
- 将应用程序设计为可以独立运行任务。这可以在不过多增加复杂性的前提下，帮助恢复故障资源。
- 将应用程序设计为可以异步运行任务。这可以在避免影响主程序处理的前提下，通过队列来延迟运行任务。
- 将本地状态存储到一个指定区域。这可以降低应用程序其余部分对状态管理的需要，提高使用冗余组件的能力。
- 尽可能使用幂等接口。幂等接口指的是可以重复调用的接口，无须担心一个动作被执行多次所带来的影响。

幂等接口通过简单的重试机制可以实现错误恢复。

幂等接口

幂等接口是可以多次调用的接口，并且只有第一次调用才有效果。连续或者重复地调用没有效果。而每次调用非幂等接口时都会产生影响。

理解这一点的最好方法是举例说明。

下面的内容描述了一个幂等接口。你可以调用命令“设置当前车速为35英里/小时”任何次数。每次你调用它时，汽车的速度都会被设置为每小时35英里。无论你调用多少次接口，汽车仍然以每小时35英里的速度运行。

使用幂等接口设置汽车的速度

假设你有一辆智能汽车，它支持一个API，允许你更改汽车的速度。该API提供了一个接口，允许你发出以下命令：

将当前的车速设置为35英里/小时。

发出这个命令会将汽车的速度设置为每小时行驶35英里。

下面的内容描述了一个非幂等的接口。每次调用接口时，你都会用指定的数字来改变汽车的速度。如果使用正确的值调用正确次数的接口，那么也可以将汽车的行驶速度设置为35英里/小时。

使用非幂等接口设置汽车的速度

假设你有另一辆智能汽车，它还支持另一个API，可以让你改变车的速度。但是，这辆车有一个不同的API接口，允许你发出以下命令：

将当前的车速增加5英里/小时。

调用API 7次之后，你的汽车速度会从0改为35英里/小时。

但是，每次调用该接口时，汽车都会按照指定的数量来改变速度。如果你一直用“将当前车速提高5英里/小时”的命令来调用，那么每次调用后汽车的速度会越来越快。在本例中，重要的是调用接口的次数，因此这是一个非幂等的接口。

对于一个幂等接口，汽车“驾驶员”只需要告诉汽车需要的行驶速度即可。如果出于某种原因，行驶速度为35英里/小时的请求没有被汽车执行，那么驾驶员只需简单地（并且安全地）重复发送请求，直到它确定汽车接收到了请求，然后驾驶员就可以确定汽车实际上的行驶速度是35英里/小时了。

但是对于非幂等接口来说，如果汽车“驾驶员”希望按照35英里/小时的速度行驶，它需要发送一系列命令让汽车加速，直到它按照35英里/小时的速度行驶。如果其中一个命令失败，驾驶员需要通过某种机制知道汽车的当前速度，并决定是否重新发送“加速”命令。不能只是简单地重发加速命令——必须首先弄清楚是否需要发送这个命令。这显然是一个更加复杂，也更加容易出错的操作。

使用幂等接口相比使用非幂等接口来说，会让驾驶员执行更简单的操作，降低出错的概率。

增加了复杂性的冗余改进

哪些例子属于增加了复杂性的冗余改进呢？实际上，至少对大多数应用程序而言，很多冗余改进看上去似乎有用，但是却增加了复杂性，实际效果弊大于利。

假设我们要创建一个系统的并行版本。在这个版本中，如果其中一个系统出现故障，另一个系统会替代它实现所需的功能。虽然这对于那种要求极度高可用的系统（例如，航天飞机）非常重要，但是它通常过于复杂，导致复杂性急剧增加，而复杂性增加的同时意味着风险增加。

另一个例子是明显分开的活动。微服务是一个能够显著提高系统质量、降低风险的模型。第3章包含了关于服务和微服务的更多内容。但是，如果极端来说，将系统划分成过于细粒度的微服务可能会导致增加系统的整体复杂性，同时增加了风险。

技巧2：理解独立性

共享某个组件的一些组件可能会声称它们之间是独立的，但是实际上，它们都依赖于那个通用组件，如图11-1所示。

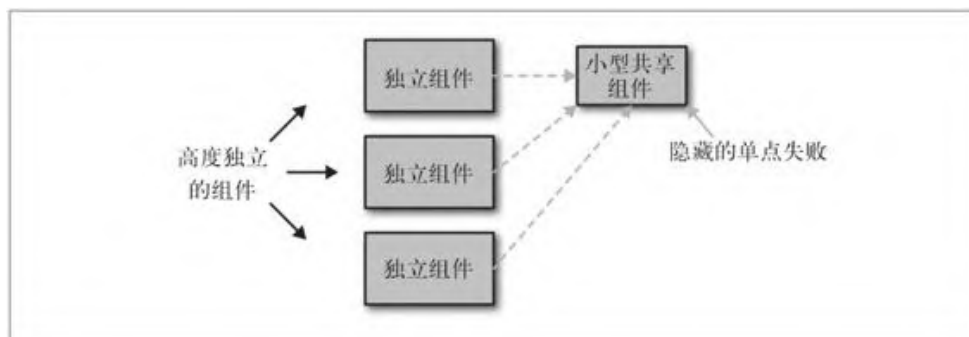


图11-1：依赖于共享组件会降低组件之间的独立性

如果这些共享组件很小、很不引人注目，或者根本没人知道它们的存在，那它们就可能成为系统中的故障单点。

假设有一个应用程序运行在5台独立的服务器上。

你可以通过这5台服务器来增加系统整体的可用性，降低单台服务器故障导致的系统不可用。图11-2展示了这个应用程序。

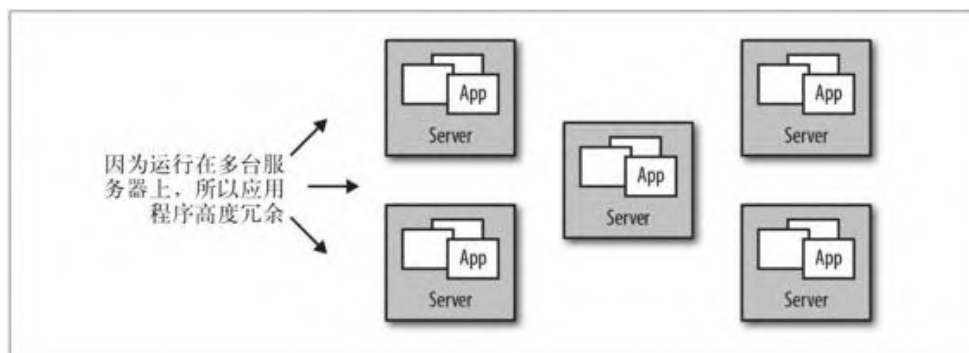


图11-2：独立服务器……

但是如果这5台服务器实际上是运行在同一台硬件服务器上的5台虚拟机呢？或者这些服务器都运行在同一个机柜上，如果这个机柜的电源发生故障会怎样呢？如果共享的硬件服务器出现故障又会怎样呢？示意图如图11-3所示。

如图11-3所示，你的“独立服务器”可能并不像你想象中那么独立。

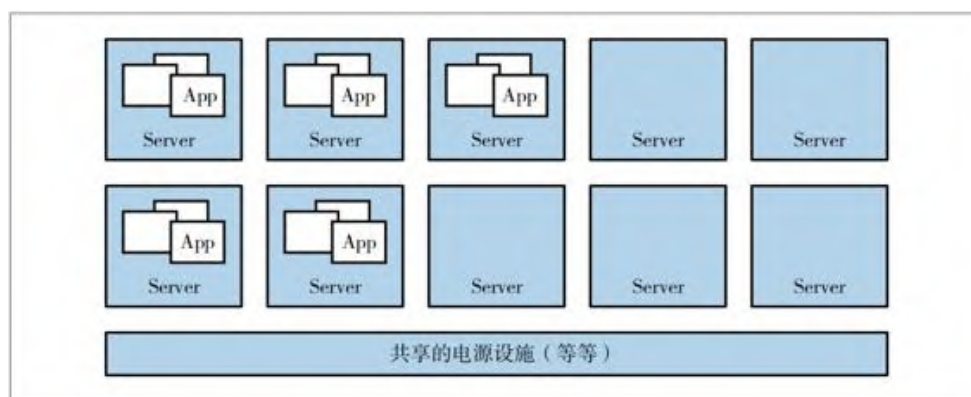


图11-3：……不像你想象中那么独立

技巧3：管理安全性

在软件系统中，坏人永远是个问题。即使在出现大规模Web应用程序之前，安全和安全监控也永远是构建系统过程中需要考虑的一部分。

但是相比之前，Web应用程序已经变得更大、更复杂，存储着更多的数据，处理着更大的流量。随着这些系统中的数据越来越有价值，试图危害系统的坏人也急剧增加。他们或者试图获取高度敏感的数据，或者让大型系统无法使用。有些人出于经济利益，有些人只是一时兴起。不管动机如何，结果如何，坏人都正在成为一个越来越大的问题。

Web应用程序的安全问题超出了本书的讨论范围。但是，无论是对于高可用性，还是对于降低大规模系统的风险来说，高质量的安全措施都是必不可少的。这里的关键点在于，你应该在风险分析、风险缓和以及开发过程中均考虑到安全因素。但是，其中应该包含的内容超出了本书的讨论范围。

技巧4：鼓励简单性

复杂是稳定的敌人。系统越复杂，就越不稳定。越不稳定，风险性就越大，从而降低可用性。虽然我们的应用程序逐渐变得越来越大，并且明显愈加复杂，但是如何在设计架构和实现时始终保持简单性，对于保证应用程序的可维护、安全和低风险至关重要。

从现代软件架构理论方面来看，基于微服务的架构通常会带来额外的复杂性。基于微服务的架构大幅度降低了单个组件的复杂性，使得我们可以通过更简单的技术和设计来创建单个服务。但是，虽然这降低了单个服务的复杂性，但是却增加了构建大规模系统所需独立模块（即微服务）的数量。为了让大量的独立模块一起工作，你不得不增加各个模块之间的互相依赖，从而增加了系统的整体复杂性。

在构建基于微服务的系统时，很重要的一点是，你需要在更简单的单个服务和更复杂的整体设计之间做出权衡。

技巧5：自我修复

在系统中设计自我纠正和自我修复的流程，可以降低可用性故障的风险。

正如我们在第1章所讨论的，如果你希望达到5个9的可用性，每个月的宕机时间不能超过26秒。即使你只达到3个9的可用性，每个月也只能容忍43分钟的宕机时间。如果某个服务出现故障后，需要某人在半夜起来发现、诊断并修复问题，那么这43分钟显然很难够用。一次事故就可能让你无法达到每月3个9的目标，更别说要达到4个9或者5个9的目标了，因此必须做到能够在没有任何人工干预的情况下自动修复问题。

这就是为什么要建立自我修复系统的原因。自我修复系统听上去好像是一种高级、复杂的系统，但实际上并不一定如此。一个自我修复的系统可能只是部署在多个服务器之前的负载均衡器，当某个服务器无法处理请求时，可以将请求快速路由给其他服务器。这就是一个自我修复系统。

自我修复系统有很多级别，从简单到复杂。下面举例进行说明。

- 一个热备份数据库，与主生产数据库保持同步。如果主生产数据库出于任何原因出现故障或者下线，热备份数据库会自动承担起主数据库的角色，并开始处理请求。
- 一个请求重试失败的服务，这样即使原有请求临时遇到问题，新的请求也能够成功。
- 一个保留待处理工作的队列系统，如果某个请求失败，它可以稍后被重新调度到一个新的工作进程上，从而增加了完成工作的可能性，避免了丢失工作记录的可能性。
- 一个时刻运行并试图导致系统故障的后台进程（例如，像Netflix中“捣乱的猴子”一样的系统），可用来检查并确保系统可以自己恢复到正常状态。
- 一个向多个独立开发和管理的服务发送请求，但进行相同计算的服务。如果所有服务计算的结果相同，那么会采用该结果。如果其中一个（或者多个）独立服务返回了与大多数其他服务不同的结果，那么该结果会被弃用，该服务会被认为发生故障，停机等待维护。

这些都只是一些例子。注意上面列出的越靠后的例子，给系统增加的复杂度越高。你需要小心这一点，尽可能在能够显著降低风险、成本又最低的情况下使用自我修复系统，避免在本来已经很复杂的系统和架构中，使用更加复杂的自我修复功能，以及避免自我修复系统本身存在的故障风险。

技巧6：标准化运维流程

软件系统会涉及人，是人就会犯错。通过可靠的运维流程，你可以将系统中人的影响降到最低，并且减少人参与操作的过程，降低错误发生的可能性。

文档化、可重复的流程可以降低人工操作中的一个重要的问题——健忘：忘记执行步骤、执行了错误的顺序，或者执行某个步骤时出现失误。

但是文档化、可重复的流程也只能降低这个问题的发生概率。人还存在其他的问题。人会犯错，他们会按错键盘、他们以为自己知道在干什么但实际却不知道。他们的每一次操作都不太一样，每一次操作也无法审查，甚至可能在情绪不好的时候执行错误的操作。

如果你能够将系统中需要人执行的操作自动化，那么就能降低失误发生的概率，提高任务完成的可能性。

重启服务器

假设你出于某个特殊目的（我们不去管这不是一个好主意），需要定期重启某台服务器（或者一系列服务器）。

你可能会让用户登录服务器，成为超级用户，然后执行“reboot”命令。但是，这会导致几个问题：

- 现在只要有人需要执行该命令，你都需要授予他们登录生产服务器的权限。此外，他们还必须有超级用户的权限才能执行重启命令。
- 当某人以超级用户身份登录服务器后，他可能不小心执行了其他命令，导致服务器出现故障。
- 当某人以超级用户身份登录服务器后，他们可能会恶意执行一些危害服务器的操作，例如，在Linux上运行rm -rf/命令。
- 你可能没有任何关于操作发生的记录，也没有谁执行重启、为何重启的记录。

相对于通过人工操作来重启服务器，你可以实现一个自动执行重启的流程。除了能够执行重启之外，它还可以提供以下好处：

- 减少对生产服务器身份授权的发放，消除失误和恶意行为出现的可能性。
- 可以记录所有执行重启的操作。

- 可以记录谁发起了重启操作。
- 可以验证发起重启操作的人是否有相应的权限（细粒度权限——你可以将重启服务器的权限授予一个用户组，而不给他们任何其他访问权限）。
- 可以确保在服务器重启前执行其他任何必要的操作。例如，临时将服务器从负载均衡器上删除，或者优雅地停止运行中的应用程序等。

你可以看到，通过自动化整个流程，可以避免人为失误，并能够更好地控制由谁来如何执行操作。

小结

自动化过程是可重复的过程。可重复的过程是经过测试的过程。经过测试的过程比临时过程错误更少。就是这么简单。

降低正在构建的系统中的风险，需要实现以降低风险为目的的标准技术。这些技术虽然简单，但是可以有效地降低风险，从而增加应用程序的可用性。

[1] “坏人”是那些为了不正当的目的而试图损坏或危害一个系统的人。

第 V 部分

原则5. 云计算：利用云计算

高可伸缩性、高可用性的应用程序需要高度动态的基础设施。

当我们构建和架构具有高可用、高可伸缩性的Web应用程序时，还必须处理应用程序上高度变化的负载。从基础设施管理的角度来看，这意味着需要提供足够的基础设施资源。如果你的应用程序需要20到200个服务器（取决于当前使用你的应用程序的用户数量），那么你最好确保始终有200个可用的服务器。实际上，你可能应该有250个可用的服务器，以防你估计错误。如果做不到这一点，就意味着你可能会遭遇因为规模导致的电力限制或者停电事故，那么你的可用性也会受到影响，客户会感到沮丧。想一想，有多少次你试图使用一个非常流行的网站，但是却发现这个网站的速度慢得让人无法接受？这是因为规模导致的电力限制或停电的结果，也是对规模做出了错误资源规划的结果。

随着互联网的发展和我们对它的使用越来越成熟，使用频率和对它的期望也越来越高。预测这些应用程序的伸缩需求变得更加困难。此外，很少有公司能够负担得起在经济低迷时期大量闲置的过剩资源。

结果是什么呢？我们需要高度动态的基础设施，即可以根据我们的可伸缩性和可用性需求自动调整大小。

这是公有云最重要的特性之一，可以帮助我们构建高度可伸缩的应用程序。你可以非常容易地动态创建和回收基础设施，来满足你当前的需求。当你的应用程序处于低需求时期时，你不需要闲置多余的基础设施资源。当你的应用程序超出使用预期时，你可以轻松地添加额外的资源来满足当前的需求。

一个为伸缩性而设计、运行在适当配置的动态云基础设施上并且构建良好的应用程序，可以有效地处理应用程序伸缩到任何级别的需求，并且能够消除高峰使用期间的电力限制和停电现象。

我们的目标和愿望是有效地处理任何伸缩需求，正因为如此，公有云对于构建高可伸缩性、高可用性和现代化的应用程序至关重要。

本部分的各章节将讨论如何在现代化的、可伸缩的应用程序中使用云计算。

第12章

使用云计算来设计可伸缩架构

在过去的几年里，人们对云计算的认识和知识的了解有了显著的增长。就在不久之前，“使用云计算”还只是一些较为激进的组织考虑做的事情，或者是那些希望降低基础设施成本的初创公司。

但是，大量企业很快就认识到了云计算的价值。在过去的几年里，除了较保守的企业外，其他企业都接受了云计算，这使得使用云计算成为主流。或者至少现在采用云计算的愿望已经成为主流。

然而，对于许多企业来说，在云计算领域取得成功仍然是一个艰巨的挑战。组织常常对迁移到云计算的好处期望过高，而低估了迁移本身所需的工作量和迁移对公司文化的影响。一个不幸的结果可能是一种恶性循环：指责、推诿、争抢任何可以被认为是胜利的东西。

当组织发现这些问题时，可能会认为它们已经受够了，并且在它们能够充分利用云计算之前停止迁移的过程。但是，如果推迟真正的改革，它们就无法认识到真正驱使迁移到云计算的成本和创新收益。随着迁移成本的激增和对承诺的特性、功能和应用程序的失望，公司最终可能会认为云计算只不过是一种昂贵的资源浪费。

为什么会这样？通常，最大的错误来自对云计算的错误理解。企业管理层倾向于认为向云计算迁移是一种简单的“直接迁移”操作，即它们认为只需要很少的修改，就可以简单地把在自己的数据中心中运行的现有应用程序，直接迁移到云计算上。

然而，真正的大规模云计算的成功，需要的不仅仅是“直接迁移”式的操作。它需要企业成功地驾驭动态的云计算世界。动态的云计算不仅促进了应用程序的伸缩能力，使这个过程变得更快、更容易，它还能帮助开发团队更快地响应变更，并且更快地实现这些变

更。这不是一种奢侈，而是确保现代化应用程序可用性的一个必要条件，因为这些应用程序表现出了极端的伸缩需求和极端的峰值性能。当你不知道客户何时会使用你的应用程序时，就很难预测静态基础设施需求。你需要动态的基础设施来满足这些现代化应用程序的需求，同时又不会浪费大量的资源。

然而，使用动态的云计算比简单的“直接迁移”式操作，能更有效地使用云计算资源。这是因为在迁移之后，应用程序和基础设施的可见性发生了变化。许多资源都变成了动态的，因此出于什么目的跟踪哪些资源也变成了动态的。此外，应用程序现在在一个团队直接控制之外的基础设施上运行，这个概念对于许多大型企业来说是陌生的。

幸运的是，大量使用动态云计算并不一定是可怕的或危险的，而是安全的和有效的，但是这是一个持续的学习过程。组织必须愿意去学习和接受云计算产品，以便使它们的需求和期望与云计算所能提供的实际功能相匹配。云计算的成熟度有一条曲线，从简单的“起重机式操作”到基于云计算功能对架构完全重写。本章将讨论这个云计算成熟度曲线。

云计算成熟度的6个级别

至关重要的是，你不能指望一下子就能达到目标。在迁移到云计算的过程中，组织需要经历如下6个基本的成熟度级别。

- 级别1—体验阶段：什么是云计算？
- 级别2—确认阶段：我们可以信任云计算吗？
- 级别3—使用服务器和SaaS：直接迁移，确认云计算运行良好。
- 级别4—使用增值服务：动态云计算成为一种实践。
- 级别5—使用特有的服务：动态云计算在企业文化中根深蒂固。
- 级别6—强制使用云计算：为什么我们要有自己的数据中心？

要想成功地迁移到云计算上，组织必须认识到这种云计算成熟度曲线的存在，并理解它的操作和流程的含义。从一个成熟度级别进入下一个成熟度级别并不总是很容易，也不总是很快的，每个组织的具体细节都有所不同。此外，组织有时会满足于适合其文化的云成熟度级别，但没有达到最终的阶段。如果这符合它们的期望和业务需求，那也是可以的。

级别1：体验云计算

迈向云计算的第一步依赖于一些安全的技术，即可以对一些简单的应用程序或应用程序的一些非关键部分应用的技术。

级别1涉及对一个应用程序或者应用程序的某一部分使用云计算，来测试云服务是否工作正常。通常，首先使用的服务是存储解决方案，例如，Amazon Simple Storage Service (S3)，因为我们通常将一些内容存储在云中，避免处理云计算所需的复杂流程和系统，例如，云服务器和无服务器计算。

这一级别通常从某个一次性的实验开始，由一个或多个团队执行独立的迁移。此时没有创建任何云计算策略，这一切都是为了弄清楚云计算到底是什么。

级别2：确认云计算

这是组织的云文化的一个关键演化点，因为它开始涉及整个公司的规章制度，例如法律、财务、安全等。在这一点上，信任会成为一个核心问题。我们能否依靠云计算来获得商业上的成功？我们能信任地把数据放到云上吗？我们是否知道如何以及在何处信任云计算是合适的，以及如何确保云计算足够安全以满足我们的需求？

这是关于如何在公司内部使用云计算的策略开始形成的时候。这些指导方针的实际内容，从正式的政策到特殊的“企业文化”理解，其实都没有那么重要。重要的是整个公司都参与进来，所有的利益相关者都参与进来。

级别3：使用云计算中的服务器和应用程序

云计算成熟度的第三个级别是组织开始替换内部的服务器和其他后端资源。这些仍然是简单的直接迁移式的应用程序，其基本思想是“让我们将应用程序迁移到云上，看看会发生什么”。

在这个级别，目标是理解云计算如何为整个应用程序工作。这时候组织开始享受使用云计算的实际优势，比如降低成本和增加灵活性。

然而，企业在这方面需要小心。级别3可能是一个危险点。如果企业试图在这个阶段确定使用云计算的价值，它们可能会发现承担了云计算的成本，却没有享受到相应的好处。这可能会导致公司放弃迁移，并认为它们对整个云计算的努力是失败的。解决方案是不把这一级别作为终点，而是作为一个过渡点。当你完成了一次简单的迁移后，要避免停下来的诱惑，更不能说：“够了，我们现在‘上云’了。”继续下一步很重要，我们要充分利用云计算所提供的功能。

级别4：使用增值的托管服务

这是云计算的一些内在价值开始显现的地方。在这个级别，组织开始关注云计算的托管服务，例如，托管数据库。托管数据库的服务，例如，Amazon Relational Database Service (RDS)、Amazon Aurora和Microsoft Azure SQL，可以为应用程序提供数据库功能，同时只需要很少的管理工作，而是交给云服务提供商来管理数据库。组织也可以考虑像Amazon Elasticsearch、Amazon Elastic Beanstalk和Amazon Elastic Container Service (ECS) 这样的服务来提供托管计算。

随着动态云计算在这里开始发挥作用，也带来了云计算的最大好处。这也是公司承诺至少为一些战略型的应用程序和服务使用云计算的时候。

级别5：使用云特有的服务

一旦一家公司成为一个基于云计算的组织，它就可以利用高价值的、云专属的服务。这些服务只能在云计算中使用，是专门为动态云计算设计的。这个级别使用的一些服务示例包括无服务器计算（例如，AWS Lambda或Microsoft Azure函数）、高度可伸缩的数据库（例如，Amazon DynamoDB）和其他通用服务，如Amazon Simple Queue Service (SQS) 和Amazon Simple Notification Service (SNS)。

在级别5，动态云计算的概念被嵌入组织的应用程序开发和管理过程中。这些服务的使用也开始将企业与特定的云服务提供商联系起来。虽然许多云服务提供商提供了无服务器计算能力，但每个服务提供商的方式略有不同。当组织开始使用这些高价值、特有的云服务时，它们不仅与云计算绑定到一起，还与特定的云服务提供商绑定到了一起。

级别6：全面拥抱云计算

这是采用云计算的最终成熟度级别。在这个最高级别，组织开始要求对所有新的应用程序都使用云计算，并且开始要求将现有的所有应用程序都迁移到云上。这一级别客户的最终目标通常是完全摆脱它们公司的数据中心，并依靠云计算来满足所有的基础设施需求。

这一级别在云原生公司中尤其常见，因为遗留系统的需求会使完全上云变得复杂。对于云原生公司来说，强制使用完全基于云计算的应用程序要容易得多。更多的老牌企业可能会选择保留它们的数据中心。然而，越来越多的传统企业正在尝试云计算，放弃管理自己数据中心的业务。

组织与应用程序的成熟度级别对比

这6个云计算成熟度级别适用于单个应用程序、组织或整个企业。但是，某个应用程序的云计算成熟度级别，可能会高于或低于整个组织的云计算成熟度级别。

例如，早期可迁移到云上的系统包括内部的应用程序，因为它们对客户和业务本身的负面影响风险更小。实际上，内部应用程序可能直接跳到级别6。

更大、更复杂的遗留企业系统迁移到云上的速度可能会慢得多。

与此同时，整个企业本身可能仍然停留在级别1、2或者3，并且永远无法达到级别6。

这完全是正常的，也是意料之中的，它说明了大型企业组织采用云计算的复杂性。

使用云计算时可能犯的错误

当你采用云计算时，很容易陷入几个非常具体的陷阱，从而导致使用策略中的重大问题。这些错误通常是迁移失败的直接原因，或者至少被认为是导致迁移失败的原因。它们还可能导致成本效益比偏离云计算的真正价值。当你在执行迁移到云计算的工作时，请注意不要落入以下任何一个陷阱。

陷阱1：不相信云安全

对于新接触云计算的公司来说，最大的误解之一是对云计算的信任问题。这表现在许多不同的方面，但是安全性是主要的方面。

安全性对所有公司都非常重要。迁移到公有云，意味着将原本安全位于公司防火墙之后的应用程序放到公有云上。你第一次考虑这么做的时候会觉得很可怕。你能信任云来保证你的数据安全吗？你的应用程序在这样的公共环境中安全吗？

简单的回答是，是的。

对于绝大多数公司来说，你的公司在公有云服务提供商手里可能比你自己的防火墙后面更安全。为什么会这样呢？因为云服务提供商以信任为生。如果它们不能保证客户数据的安全，它们就无法开展自己的业务。

云服务提供商在构建高质量的安全团队上投入了大量资金，这些团队将时间花在提升安全协议和流程的技术水平上。将你的数据交给信誉良好的公有云服务提供商，你可以获得由安全领域的领导者带来

的经验教训和最佳实践的优势。除非你的公司有和云服务提供商一样的资源来投资安全，否则你可以从这些经验中得到很多好处。

通过使用公有云服务提供商以及它提供的所有安全产品，你可以将应用程序和数据保存在公有云中，而不必放在自己的防火墙后。

陷阱2：直接迁移到云计算

在采用云计算的早期阶段，许多公司考虑将应用程序直接迁移到云计算中，认为只需将应用程序从自己数据中心的服务器上取下来，然后将它们移动到在云服务提供商创建的服务器上。

这种类型的迁移方式称为“直接迁移”，我们在本章前面已经讨论过，它是采用云计算成熟度中的一个基础级别。

虽然“直接迁移”是将应用程序从数据中心迁移到基于云的数据中心的一种快速、有效的方法，但是它并不能使应用程序变得对云友好。它并没有利用云的原生价值和原生能力。是的，你可以从直接迁移中获得一些好处，包括只需在另一个区域启动服务器就可以扩展到其他数据中心，但是也就是这些好处了。事实上，如果在云上托管这种基础的应用程序，实际上可能比放在你自己的数据中心更糟糕。为什么？成本。

云计算能够而且确实为能够动态分配云资源的用户提供了显著的成本优势，但是它的成本通常无法与数据中心提供的静态基础设施相比。当使用云的动态功能时，你可以节省资金。如果只是简单地迁移一些基础的应用程序，通常不会为你省钱，反而会让你花更多的钱。

直接迁移会花费你的金钱和时间，而且不会给你带来任何你想要迁移到云上的好处。

陷阱3：无服务器的诱惑—太过于依赖炒作

人们很容易陷入对云计算的大肆宣传中，最新的、最好的云服务产品似乎是所有问题的解决方案。然而，与任何新技术一样，了解如何以及在何处应用该技术，对于成功地使用它是至关重要的。这当然

也适用于云服务提供商最新提供的函数即服务（FaaS）产品，例如，AWS Lambda和Microsoft Azure功能。

这些产品承诺为你的软件提供一个可执行环境，而不需要你管理它们所运行的服务器。这种“无服务器计算”服务对那些希望使用云计算来降低基础设施管理成本的公司来说，是非常有吸引力的。但是，与所有新技术一样，诸如Lambda之类的FaaS产品可能适用于某些类型的问题，而不适用于其他类型的问题。

然而我经常听到一些个人的声明，比如，“Lambda将解决我的计算基础设施问题”和“我们将把所有的软件都迁移到Lambda上”。

对于那些认为Lambda这样的FaaS产品可以解决所有问题的人，我建议他们要小心。AWS Lambda和其他云服务提供商提供的类似产品，为某类计算环境提供了巨大的优势，但是它们可能被过度使用。

如果它们被强制用来解决不适合解决的问题，那么实际上会为你和基础设施管理带来更多的问题。

你可以将它们作为应用程序架构的重要部分使用，但不要依赖它们来解决所有的计算问题。只在需要的时候使用它们。

何时以及如何使用多个云计算平台

在决定将应用程序迁移到云上时，你需要在选择云服务提供商之前考虑许多因素。你需要什么功能？哪个云平台更快？哪一个更便宜？哪一个更可靠？

但是，还有一个越来越常见的问题：我应该使用多少个云服务提供商？

看似显而易见的答案是一个服务提供商，但是使用云计算的客户列举了许多使用多个服务提供商的理由。首先，你想使用的一些功能可能只能由某个云服务提供商提供，而其他功能可能只能由另一个云服务提供商提供。另一个原因是，在签合同时，选择多个服务提供商而不是单一服务提供商可能会提供更好的谈判空间。另一个经常提到的原因是可靠性，即当某个云服务提供商宕机时，另一个云服务提供

商仍然可用。或者原因也许是随机的，组织的一部分人喜欢某个服务提供商，而另一部分人喜欢另一个服务提供商。

但是你的答案可能适合你的组织，也可能不适合。使用多个云服务提供商可能会给你带来好处，也可能会伤害到你，你可能会因为上面提到的任何原因使用它。

让我们来看看如何根据你的具体情况做出最好的决定。

如何定义我们所说的多个云平台

在讨论需要多少个云平台之前，我们需要一些定义。多个云平台的实际优点和缺点，在很大程度上取决于你正在考虑的多云环境类型。因此，让我们看看三种不同类型的云配置：联合云应用程序、选择性云应用程序和单云应用程序。

联合云应用程序

联合云应用程序是指一个应用程序使用两个或多个云服务提供商来提供并行计算功能。指定的应用程序或服务，可以在任何一个或者所有支持的云服务平台上运行，如图12-1所示。

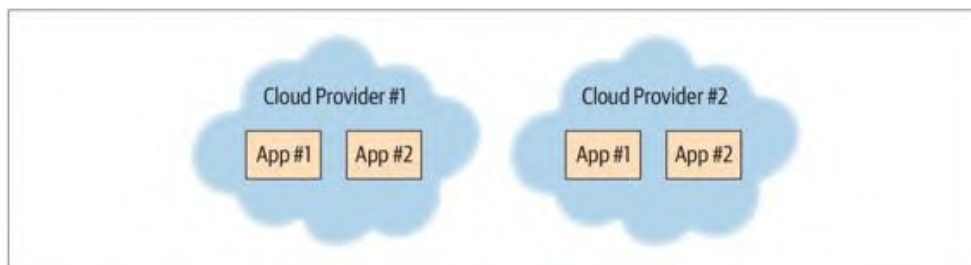


图12-1：联合云应用程序运行在多个云平台上

App# 1和App# 2可以在任何一个云服务提供商上运行。如果需要，你还可以同时跨两个云服务提供商来平衡每个应用程序的负载。如果一个云服务提供商变得不可用，另一个云服务提供商可以接管运行应用程序的责任。

每个应用程序必须设计为可以在任意一个云服务提供商上运行，并且应用程序可以使用任意一个可用的云服务提供商来处理指定的请求。如果一个云服务提供商不可用，另一个云服务提供商可以接管处理应用程序请求的责任。

这种方法的一个主要优点是应用程序的弹性。如果一个云服务提供商遇到问题，应用程序负载可以轻松且相对快速地重新路由到另一个云服务提供商。这使得即使其中一个服务提供商出现了严重的故障，应用程序仍可以继续运行。

这种架构经常被作为绑定单一云服务提供商的解决方案，因为你可以在多个云服务提供商之间切换负载。但是，这种架构也有明显的缺点。例如，支持应用程序的每个开发和运维团队必须了解多个云服务提供商的工作方式，而这种知识和理解不是免费的。类似地，你必须在多个云服务提供商上测试和维护每个应用程序。

此外，当应用程序被编写为支持多个云服务提供商时，它们无法利用某个指定云服务提供商提供的更加深入的特性。你必须将应用程序编写为，可以使用所有云服务提供商都可以支持的常见的功能。

在大多数情况下，这种方法的缺点超过了对弹性的提升。

选择性云应用程序

这是指你的公司维护着与多个云服务提供商的关系，但是指定的应用程序完全运行在某个云服务提供商上，如图12-2所示。

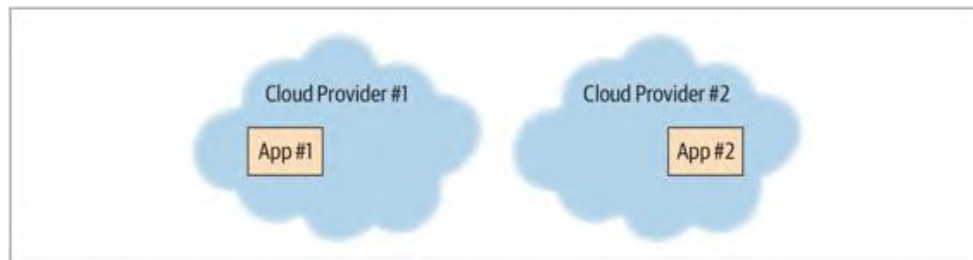


图12-2：选择性云应用程序，每个应用程序都运行在一个指定的云服务提供商上

你可以看到，每个应用程序仅被托管在一个云服务提供商上，但是不同的应用程序可能托管在不同的云服务提供商上。

在这种场景中，一个指定的组件被设计为只能在某个云服务提供商上运行。

这种方法的一个明显缺点是弹性。如果云服务提供商变得不可用，那么在该云服务提供商上运行的应用程序或服务将停止运行。你无法简单地将流量转移到另一个云服务提供商。这通常是一个智力问题，而不是实际问题。整个云服务提供商变得不可用是很少见的。通常，只会有一个或多个可用区或者区域变得不可用。一个恰当编写的应用程序可以利用多个可用区和区域来提高应用程序的弹性，而不必求助于使用多个云服务提供商。我们将在第15章更详细地讨论AWS的区域和可用区。

然而，这种架构的一个真正的潜在优势是，单个应用程序或者服务可以根据所属团队的标准，独立地选择它们想要使用的云服务提供商。

这种架构要求支持指定应用程序的开发和运维团队，只需要学习和理解单个云服务提供商的运维方式。此外，每个团队都可以利用其云服务提供商提供的特有的、更深入的特性。应用程序也可以按指定云服务提供商的最佳实践来设计、构建和优化应用程序本身。

但是，在这种架构中，公司必须与每个支持的云服务提供商维护相应的关系和协议。这更多的是一个管理问题，而不是技术问题，但它可能是你的组织的一个问题。

在大多数情况下，这种方法为你提供了所需的多云灵活性，以及与云服务提供商进行深度集成的能力，而不需要像联合云应用程序那样的复杂性，也不会显著牺牲应用程序的弹性。

组织经常会回到这个特定的架构中。一个团队或者组织为其应用程序选择一个云服务提供商，而另一个团队或组织为其应用程序选择另一个云服务提供商。企业作为一个整体是多云的，但是单个应用程序使用单独的云服务提供商。

单云应用程序

这是最简单的设计，在这种设计中，由单个云服务提供商满足公司内部的所有云计算需求，如图12-3所示。

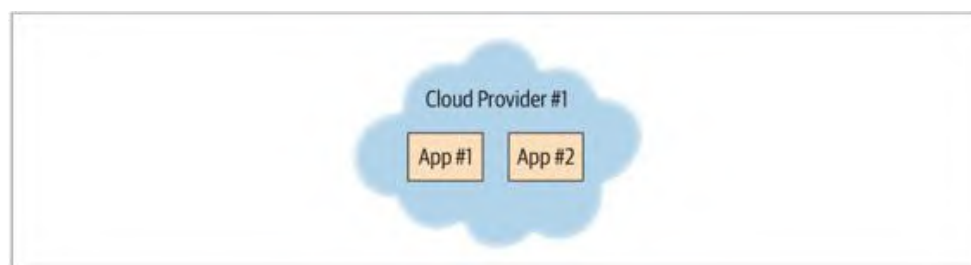


图12-3：单云应用程序—所有应用程序都运行在单个云服务提供商上

在这个架构中，公司对单个云服务提供商进行标准化，这允许所有的开发和运维团队都关注一个云服务提供商的能力。知识可以很容易地在团队之间共享，多个团队可以形成一组最佳实践。所有的应用程序都可以利用云服务提供商提供的全部特性。

从管理的角度来看，只拥有一个云服务提供商可以简化对提供商的管理。此外，由于所有的流量都要经过一个云服务提供商，所以它的使用量更高，你可以与服务提供商协商更好的价格和其他条款。

但是，这个解决方案显然需要让单一的云服务提供商做出强有力的承诺，这对一些公司来说可能是有问题的。当问题发生时，并且如果你被绑定在一个服务提供商时，协商解决方案就会困难得多。

这个解决方案可能是所有管理和控制选项中最简单的方法，但是它限制了开发团队的灵活性。

选择哪个模型？哪种云？

那么应该使用哪种云计算模型呢？哪个对你的公司是有意义的？最终的答案取决于你的公司和应用程序的需要。

从应用程序的角度来看，在多个云服务提供商上运行应用程序的优势通常会被维护多云应用程序的成本和复杂性所抵消。因此，几乎在所有情况下，图12-1所示的联合云应用程序模型都没有意义。如果你不知道在将应用程序与单个服务提供商绑定时，如何维护应用程序的高可用性，请考虑使用该服务提供商提供的高可用解决方案。例

如，为在AWS上运行的应用程序简单地使用多个可用区和多个区域，可以极大地提高应用程序的弹性，还不会导致在多个独立的云服务提供商上运行应用程序的成本增加和能力降低。

在选择特定的云服务提供商时，你应该查看以下内容：

历史上的可靠性

这项服务在历史上有多可靠？如何快速处理故障中断？中断会影响整个服务提供商，还是在任何时候只影响特定的区域（允许你使用多个区域来提高弹性）？

可用性技术的能力

云服务提供商是否为你提供了多个可用区和多个独立的地理区域，来允许故障隔离和故障转移？可用区和区域的独立性有多高？

可用的服务

云服务提供商是否提供了你所需的服务类型和功能？

迁移到云计算的原因

你为什么要将应用程序迁移到云上？是为了加速创新还是为了帮助你扩大规模？不管是什么原因，确保它与云服务提供商提供的功能相匹配。

如果你选择使用多个云服务提供商，那么请确保这样做是因为你需要它们提供的能力。不要想使用多个云服务提供商来提高弹性。现实情况是，成本和劣势超过了这种方式带来的收益。

云计算小结

本章主要讨论如何使用云计算来构建可伸缩的应用程序。我们讨论了组织在使用和信任云计算能力上的成熟度级别，介绍了组织在采用云计算时所犯的常见错误、这些错误可能导致的陷阱，以及我们如何以及何时在应用程序中使用多个云服务提供商。

第13章

云计算改变的5个行业趋势

云计算已经改变了我们构建和运行应用程序的一些理念。但是，当我们还在考虑如何使用云计算来改变应用程序的时候，云计算已经发生了变化，并且同时改变了我们对云计算的理解。

云计算有哪些变化

云计算这十年来逐渐变得成熟。云计算提供商也增加了很多类型的产品，它们不再只是简单提供诸如文件存储、计算能力等资源。像AWS就提供了超过160种不同的服务，来满足各式各样不同的计算需求。[\[1\]](#)Azure和Google也提供了上百种服务。

因此，对于我们和应用程序来说，云计算带给我们最大的变化是什么？下面会介绍云服务驱动、改变和鼓励的5个行业趋势。

变化1：对基于微服务架构的认可

正如我们在本书中讨论的，基于服务或者微服务的架构在最近几年已经变得越来越普及，为了减少技术债务、让应用程序更易维护，将应用迁移到基于服务的架构正在成为一种标准技术。

随着各个公司将它们的应用迁移到云计算上，云计算已经成为它们整体产品升级策略的一部分。这些策略还包含向最先进的应用程序架构迁移，例如这几年趋之若鹜的微服务架构和其他服务化架构。也正是因为像Docker这些技术的出现，使得应用程序开发使用微服务架构成为可能。

此外，函数即服务（Function as a Service, FaaS）产品，例如，AWS Lambda，为创建无须服务器的简单微服务提供了可信手段。即便暂时不考虑这种架构风格的优缺点，在云平台中创建FaaS服务也促进了基于微服务架构的发展。

意识到这一点，云计算提供商已经开始提供一些价值更高的产品，例如，用来管理微服务容器的EC2容器服务以及运行基于FaaS微服务的AWS Lambda。

变化2：更小、更专业的服务

随着将应用程序逐渐迁移到云上，我们开始思考如何将云计算用到自己的服务中。一些过去由应用程序自身提供的功能，目前都被云计算所代替。

如今一些主流的云计算提供商，已经可以提供诸如数据库、缓存服务、队列服务、日志服务、CDN以及音视频转码等多项服务。

变化3：更专注于应用程序

云计算已经让我们不必再过多关注于基础设施的创建和管理，而是让我们能够将更多精力用在应用程序和相关环境上。此外，云计算也很大程度地降低了应用程序的管理负担，使得我们可以更专注于解决程序如何运行的问题。

变化4：微型初创公司

云计算使得一些非常小的初创公司也可使用相关服务。这些微型初创公司经常是自筹资金、个人运营，但通过云计算，它们也能够使用一些廉价的、可伸缩的计算服务或其他服务。

对于一个有想法的人来说，将其想法实现并从中盈利并不容易。如果不需要昂贵的基础设施投入就可以快速建立一个计算生态系统，无疑能够帮助他们将一些新奇的想法快速投向市场。尤其是一些在线游戏的移动应用，它们可从这种能力中收获颇丰。

这些初创公司通过最少的投入让产品快速上线，去市场中检验成功与否。对于那些蓬勃发展的公司，云计算为其应用提供了廉价且易于伸缩的能力，使得公司可以将基础设施的投入，维持在与业务发展合理的比例上。从财务的角度讲，这让初创公司更加容易运营和管理。

变化5：安全和合规已经成熟

在云计算出现的早期，安全问题是很多公司拒绝使用云计算的一个主要因素。

意识到需提升安全的需求后，云计算提供商为云应用提供了更好的安全能力，也以PCI、SOC和HIPPA等安全法规的形式加强了安全保障。

通过不断增加高等级的安全措施，安全因素已经不再是公司将产品迁移到云计算的障碍。

变化还在继续

变化是不可避免的，云计算已经改变了我们构建和运行应用程序的观念。我们已经开始构建一些更小的、更专业的服务，并且已经学会了如何处理日益增长的海量数据，也将更多的精力从基础设施建设转移到了应用程序本身。小公司的模式变得越来越可行，同时带来了更多新的想法和创意。安全也已经成为我们做每件事的标准。

云计算已经成熟，我们也逐渐越来越熟练地使用云计算。这在未来会继续发展，我们必须不断适应并紧跟未来的变革。只有这样，我们的应用程序才能持续保持成长和发展。

第V部分的其余章节将更详细地介绍云计算的各个方面，以及它们如何影响大型应用程序的开发和架构。

[1] 截至2019年，AWS拥有165项服务。

第14章

SaaS和租赁类型

“软件即服务”（SaaS）是一个目前常用的术语。在最基本的层面上，SaaS是指由第三方在其计算机上运行和操作的软件，而不是由你在自己的计算机上运行和操作的软件。

但SaaS可能是一个非常误导人的术语。一些过去开发过定制化（on-prem）软件的公司已经决定“进入云计算时代”并提供SaaS服务。通常，它们会把卖给客户的软件安装到自己的数据中心的硬件上。它们将之称为“云”产品，称自己为“软件即服务”公司。

尽管这可以被称为SaaS产品，但是它跟基本的托管主机没太大区别。该公司为你托管该软件，但是你仍然拥有自己的软件实例，并且你的公司（或者第三方供应商）需要管理该软件实例。这些产品具有与定制化软件解决方案相关的所有问题，包括版本的问题和痛苦的升级等。这将导致新的功能开发更慢、修复bug的速度更慢、成本更高和停机时间更长。

而且这些产品都有与定制化解决方案相同的伸缩问题。

“软件即服务”公司可能将此称为SaaS，但是你并没有真正获得SaaS的好处。

让我们仔细研究一下不同类型的SaaS服务，并且了解一下如何利用它们来帮助伸缩应用程序。

比较托管主机和不同类型的SaaS

在业界，托管主机和SaaS之间存在很多混淆。另外，不同类型的SaaS之间也存在概念混淆。为了帮助澄清这些概念，我给出以下定义。

托管主机

托管主机是指提供商提供硬件，并为客户提供帮助，在一个托管平台上运行某个软件栈的指定实例。提供商提供的服务可以简化软件的安装和设置，降低管理软件的复杂性。但是如何管理软件最终是客户的责任，容量规划和伸缩也都是客户的责任。

多租户SaaS

多租户SaaS是我们通常认为的“真正的”SaaS。提供商提供的软件平台可以让许多客户使用软件平台的同一个运行实例。客户通过软件本身的访问控制彼此隔离。升级通常由提供商完成，对客户是透明的。

单租户SaaS

单租户SaaS是提供商为其客户提供一个软件平台，但软件的一个指定实例（在指定的计算机上运行）只能给一个客户使用。提供商会运行多个实例，通常每个客户一个。客户通过对整个软件实例的访问彼此隔离。升级通常由提供商完成，对客户是透明的。

这三种类型各有利弊。

托管主机

托管主机是我在这里列出的三个选项中最“简单”的一个。它指的是主机提供商为你提供多台服务器（虚拟的或者物理的），让你在它们的硬件上运行软件。为了增加其服务的有用性，提供商通常会在其服务器上自动安装一些通用的软件包，以便让你能够快速、轻松地启动和运行一个通用软件包，而不必管理软件本身的安装。一个很常见的例子就是WordPress博客内容管理系统。托管服务提供商会自动在你租用的服务器上安装WordPress软件，允许你开始建立博客或者网站。

然而，这些提供商通常只做你（客户）要求它们做的事情。如果出现新版本，它们通常不会自动为你升级软件。有些服务提供商可能会提供升级帮助，但通常是在你的请求和帮助下进行操作的。通常，你需要担心正在运行的软件版本是否存在bug或安全漏洞，决定何时以及如何升级是你的责任。你可能不需要亲自进行安装，但是在管理软件方面，你有许多与使用传统定制化软件相同的担忧。

托管主机因此通常需要与定制化软件类似的软件管理级别。

多租户SaaS

从概念上讲，多租户SaaS非常简单。多租户SaaS提供商有许多客户，可能有数千个，都运行在应用程序软件的单个实例上。客户共享相同的运行时软件。所有客户的数据通常位于相同的数据库中，但是每个客户的数据通过业务、软件、安全规则和需求，与其他客户的数据在逻辑上是分离的。

这样的架构有很多优点。对于提供商来说：

- 升级可以自动同时应用于所有客户。可以快速、持续地推出新功能。
- “内部”很容易重现客户的问题，因为重现问题所需的环境与客户使用的环境完全相同。
- 系统资源，例如，CPU、内存和存储，根据需要在客户之间共享和分配。这意味着单个客户“尖峰”的使用量，可以分摊到当前未使用其资源的用户的可用资源中。
- 提供商可以在软件中使用集中式的信任机制。这包括一些关键的操作功能，例如安全性、可用性和可伸缩性。通过集中这些功能，可以将更多的专业知识应用于更多的客户，从而创建更好的整体解决方案。

随着客户数量的增加，同时管理所有客户的规模经济也得到了改善。以集中的方式管理客户实例，比管理单个客户实例和系统的复杂性容易得多。

这样的架构对客户有以下好处：

- 不需要担心软件升级、应用安全补丁、硬件扩容、存储和网络等，提供商会为你管理这些。
- 更容易获得新功能，且新功能的推出速度通常比传统软件更快。
- 关键bug通常可以更快地得到修复，并且可以更快地将修复发布到生产系统中。
- 更多的资源（包括计算机资源和信任）可以按照客户的需求使用，因为资源可以更容易地分配和应用于更多的人。

但是，如果没有强大可靠的提供商提供SaaS服务，这种架构可能会有一些缺点。软件可能会突然意外地发生改变。如果在错误的时间为某个客户（例如，在一个财政季度末，或者在一个繁忙的购物季节）做了这件事，结果可能是令人不安和有问题的。

但是，总的来说，由高质量和负责任的提供商运行的多租户SaaS平台，提供的优点远远多于缺点，而且它正在迅速成为许多企业中消费者和企业客户的首选。



多租户SaaS并不意味着所有客户都在一个实例上。多租户SaaS提供商可能提供多个实例，每个实例上都有许多客户。这通常是出于地理原因（例如，欧盟客户与美国客户处于不同的实例上），但是也可以出于负载均衡和水平伸缩的原因这样做。

单租户SaaS

单租户SaaS是一种不太为人所知和使用较少的SaaS模型，但是它仍然在许多重要的领域中被使用。单租户SaaS本质上是一个软件提供商，它将软件栈的整个实例提供给单个客户。为了管理多个客户，提供商通常运行其软件栈的多个实例。

单租户SaaS有以下优点：

- 客户之间的数据通常在物理上更加隔离（请注意，这是一个合理的假设，但并不总是如此）。
- 由于独立的客户使用独立的资源集，因此客户之间不会“窃取”别人的资源。
- 每个客户必须为自己的需求分配足够的资源，而资源不能在客户之间共享。
- 单个客户可以运行不同于其他客户的软件版本（其优点和缺点可能需要单独用一篇文章来介绍）。
- 就像多租户SaaS一样，客户不需要担心升级软件等事情，提供商会为你管理这些。

混合不同类型的SaaS

多租户SaaS和单租户SaaS并不相互排斥。单个提供商可能有一个应用程序，其中一些客户运行在多租户实例上，而其他客户（可能是提供商最大或者最关键的客户）运行在多个单租户实例上。

通常，单租户SaaS和多租户SaaS之间的区别对客户来说是透明的。虽然客户可能会意识到，并且可能由于合同原因要求提供商提供单租户，但是客户的日常体验通常不会受到此决定的影响。事实上，多租户应用程序的一些后端系统实际上可能是单租户的，反之亦然。

常见的SaaS的特点

关键要了解的是，以下这些通用特征适用于任何真正的SaaS产品，无论是单租户的还是多租户的：

- 它们是由提供商提供和管理的软件。
- 客户不需要担心软件的运行和管理。
- 软件版本的问题由提供商负责处理，客户不需要担心这些问题。

- 客户可以专注于使用软件，而不是运行软件。

这些特征通常不适用于托管主机的模式。

SaaS与托管主机

SaaS和托管主机的最大区别是什么？最大的区别在于所有权和决策的制定。在SaaS环境（单租户或多租户）中，关于升级和bug修复安装的决策由服务提供者负责。在托管环境中，关于升级和bug修复安装的决策由客户负责。在真正的SaaS环境中，客户不用担心软件管理方面的问题。在托管环境中，他们需要担心这些。

考虑一下托管一个WordPress博客。你的WordPress博客是放在wordpress.com上，还是放在GoDaddy公司的服务器上，还是放在自己的服务器上运行（例如，一个EC2实例）？所有这些模型都存在，并且在某种程度上都可以被认为是“云”。但是这些都是假的，没有一个是SaaS。wordpress.com是多租户SaaS的一个例子。GoDaddy的例子可以是单租户SaaS，也可以是托管主机，这取决于它们提供的功能。EC2实例是一个真正的托管主机的例子。

小结

正如我在本章中所述的那样，每种模型都有其优点。它们是否适合某种实现，取决于每种情况下的需求和优先级。对于希望对软件如何实现和运行高度控制，并且有足够技术能力来有效地管理它们的企业来说，托管主机可能是一个合适的选择。

然而，当人们购买云计算或者SaaS解决方案时，他们通常希望提供商提供这些功能，并处理性能调优、升级和安全性等问题。在这种情况下，一定要注意类似SaaS的解决方案，并确保了解自己的需求以及服务提供商提供的功能。

第15章

在AWS云上分发你的应用程序

在第2章中，我们讨论了跨多个数据中心分发应用程序的价值，这是一种在高可伸缩的环境中提高可用性的方法。

同样的原理也适用于云计算。当将部分或者完整的应用程序放到云上时，我们需要观察它们在云中所处的位置。应用程序在云中的分布情况与在普通数据中心一样重要，特别是在应用程序可伸缩的情况下。我们还讨论了共享基础设施组件（例如，机架电源）中未知的公共故障点的危险。云计算也有一些在应用程序部署设计过程中应该注意的常见故障点。

由于云的运维方式，云服务让你更难了解应用程序如何分布，也让你更难以主动地将应用部署得更加分散。一些云服务提供商甚至没有提供足够的信息，让你知道应用程序运行的地理位置。这使得设计可恢复的基础设施架构变得更加困难。

幸运的是，AWS会帮助你分发应用程序。像AWS这种大型云计算提供商，虽然不会告诉你程序运行的具体位置，但是会提供足够的信息让你来决定程序在哪里运行。然而，为了做出正确的决定，你必须理解这些信息的微妙之处。如果你想理解这些信息并加以利用的话，需要先理解AWS的架构设计。

在本章中，我们将讨论AWS的架构，以及如何设计和部署你的应用程序来避免常见故障。

AWS的架构

首先让我们讨论一些AWS中的术语。

AWS区域

云资源相连形成的一大片地区称为AWS区域，表示一个特定的地理区域。一般来说，一个区域指的就是一个大洲或者国家的一部分（比如西欧、亚太东北地区或者美国西部）。它们描述并记录了云资源的地理多样性，并且由多个可用区（AZ）组成。[\[1\]](#)

AWS区域通常由一个字符串来表示它所在的地理位置，表15-1列出了目前的AWS区域，以及它们的名称和所在的地理位置。

表15-1：AWS区域列表

区域名称 ^a	覆盖的地理区域
us-east-1	US East (N. Virginia)
us-east-2	US East (Ohio)
us-west-1	US West (N. California)
us-west-2	US West (Oregon)
ca-central-1	Canada (Central)
eu-west-1	EU (Ireland)
eu-west-2	EU (London)
eu-west-3	EU (Paris)
eu-central-1	EU (Frankfurt)
eu-north-1	EU (Stockholm)
ap-northeast-1	Asia Pacific (Tokyo)
ap-northeast-2	Asia Pacific (Seoul)
ap-northeast-3	Asia Pacific (Osaka)
ap-southeast-1	Asia Pacific (Singapore)
ap-southeast-2	Asia Pacific (Sydney)
ap-east-1	Asia Pacific (Hong Kong)
ap-south-1	Asia Pacific (Mumbai)
sa-east-1	South America (São Paulo)

^a 截至2019年7月的AWS区域和可用区。

AWS可用区

AWS可用区是AWS区域的子集，表示一个区域指定部分的云资源，但是它们在网络拓扑结构上是彼此隔离的。AWS可用区描述和记录了云

资源在网络拓扑上的多样性。如果两个云资源属于不同的可用区，即使它们属于同一个AWS区域，你仍然可以认为它们属于两个不同的数据中心。如果两个云资源属于同一个可用区，它们可能会位于同一个数据中心、同一楼层、同一机架，甚至同一台物理服务器上。

AWS可用区用一个字符串来命名，以该可用区所在的AWS区域名称开始，随后是一个字母（a~z）。表15-2列举了一些可用区及其所在的AWS区域。

表15-2：AWS可用区名称（示例）

区域名称	可用区名称
us-east-1	us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e
us-west-1	us-west-1a, us-west-1b, us-west-1c
us-west-2	us-west-2a, us-west-2b, us-west-2c
ap-northeast-1	ap-northeast-1a, ap-northeast-1b, ap-northeast-1c
...	...

数据中心

这个术语并不在AWS的术语表中，但是为了能够将传统的非云术语与AWS术语对应起来，我们在这里仍然使用它。

数据中心是一个用来放置系统资源（例如，服务器）的指定楼层、建筑物或者建筑群。

我们的目标是将应用程序分布到多个数据中心。因此，了解数据中心如何对应实际的云计算实现非常重要。

总体架构概述

图15-1从整体上展示了AWS云服务的架构。AWS由多个AWS区域组成，为了给全世界绝大多数地方提供高质量的访问，这些AWS区域在地理上分布于全球的各个地方。每个AWS区域不仅接入了互联网，并且区

域之间可以彼此互通，但是同其他互联网一样，它们也需要使用远程网络连接。

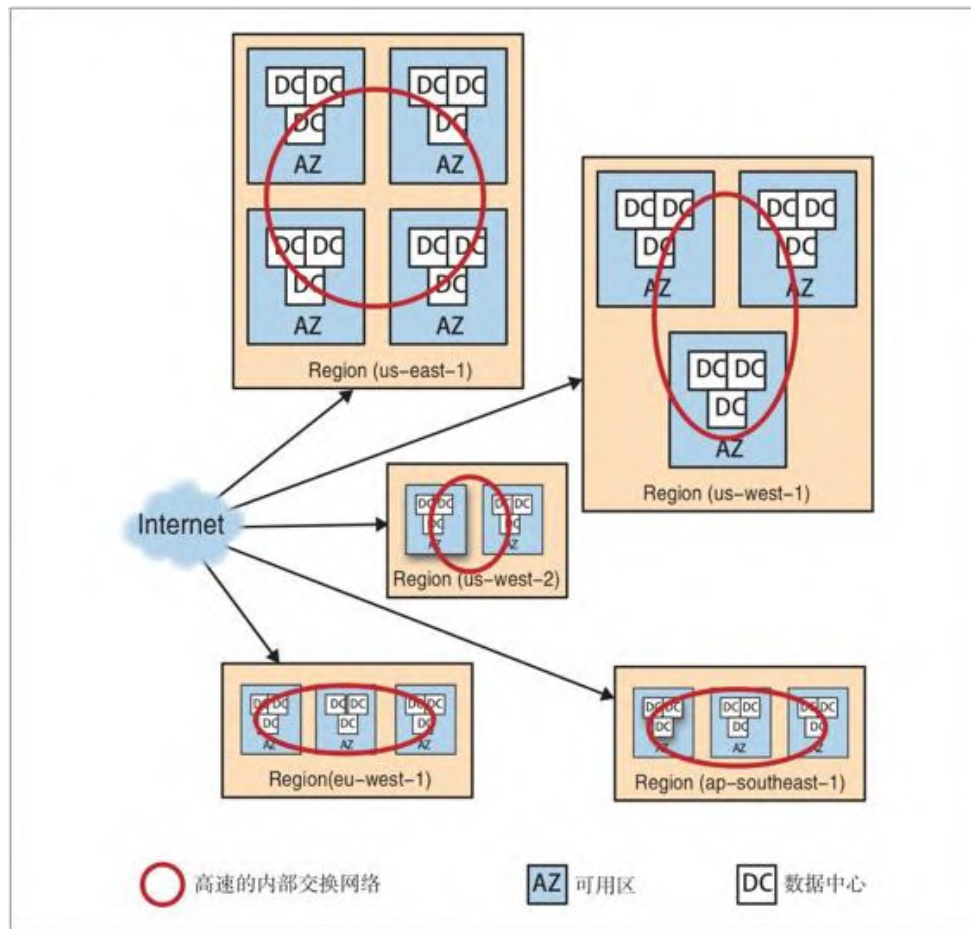


图15-1: AWS数据中心架构

一个AWS区域由一个或多个AWS可用区组成。如图15-2所示，同一个AWS区域内的可用区之间，通过高速的交换机网络互相连通，这样访问同一AWS区域内任意两台服务器的速度几乎一样，无须担心它们所处的可用区的位置。

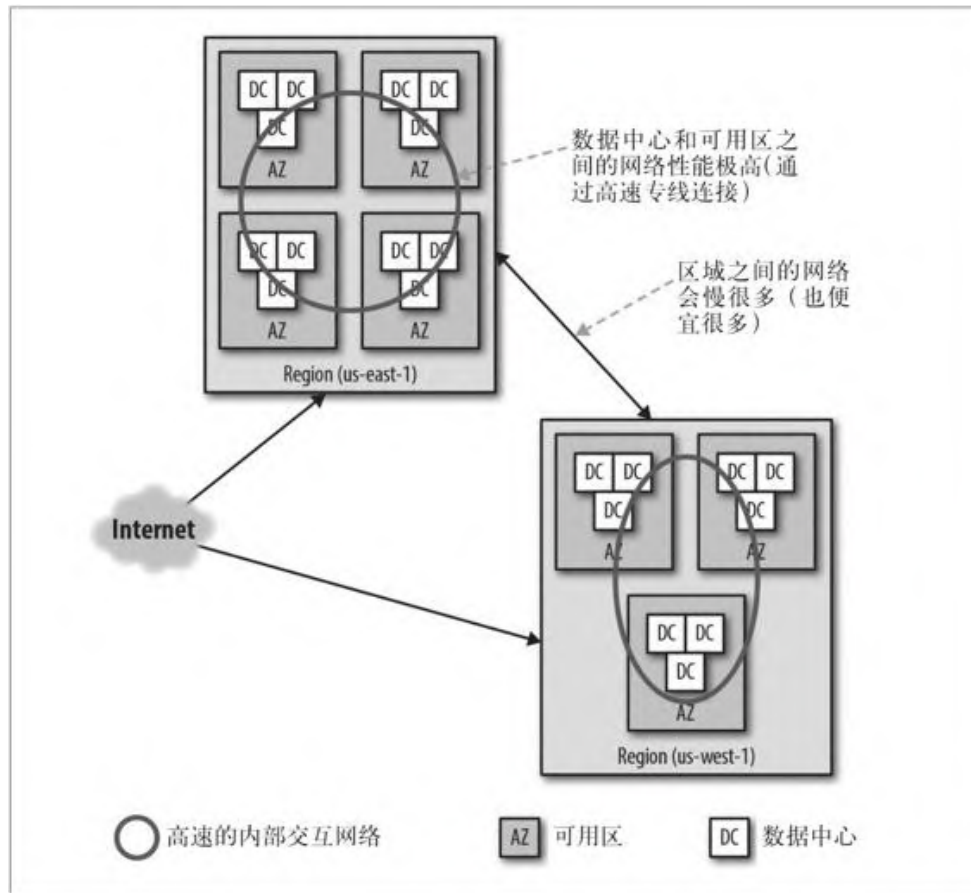


图15-2: AWS区域和可用区的网络性能

一个可用区是由一个还是多个数据中心组成，取决于这个可用区的大小。

如你所见，这种网络拓扑设计使得同一区域内的应用程序可以很容易地部署在多个不同的可用区中。这种分布式设计是为了在单个数据中心出现故障时能够切换到备份系统，同时保留独立组件之间高速、透明的通信能力，而不用关心它们处于哪个可用区内。



虽然不能选择服务器所在的数据中心，但是将两台服务器放在两个可用区可以保证服务器位于两个数据中心。

不过，由于AWS区域的设计，整个应用程序必须位于某个区域中，无法做到与其他区域的高速通信。如果你想在多个区域中运行某个应用程序，那么每个区域都需要能够独立运行一套该程序。这样，每个

地理区域都可以在本地访问到应用程序的实例，避免了远程网络通信的代价，如图15-3所示。这也是由AWS网络流量成本模型所决定的，它规定同一区域内的两个可用区之间的通信是免费的，但是跨区域通信或者从区域内访问互联网是需要收取一定费用的。

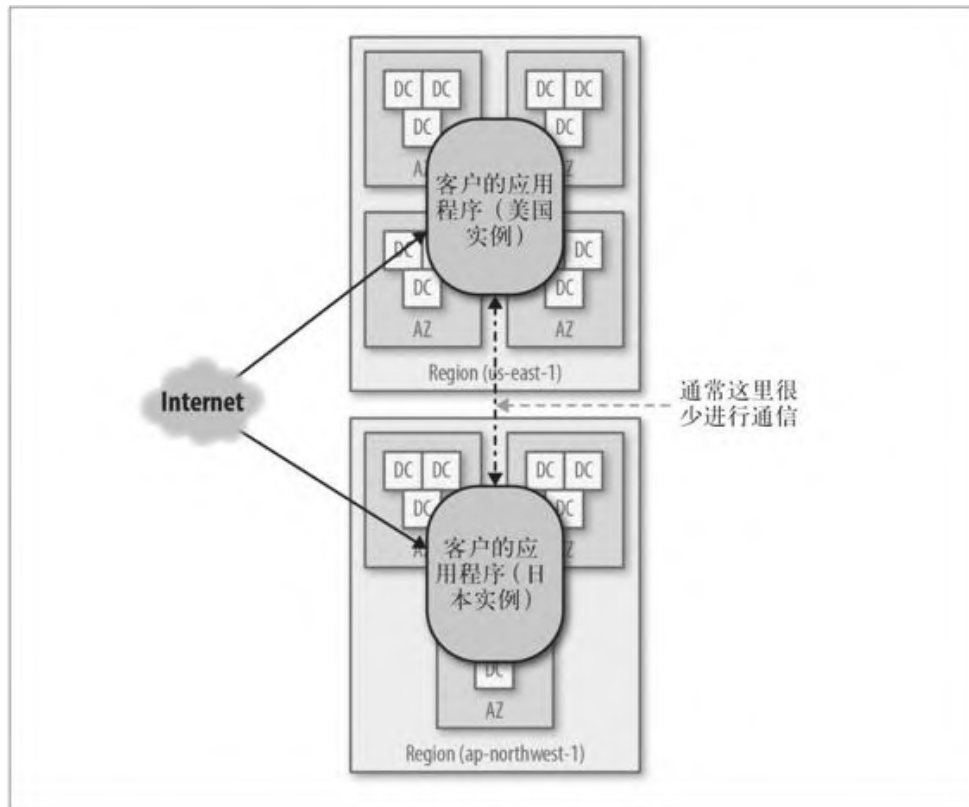


图15-3：客户的架构

这种架构模式不管从成本的角度，还是从时间延迟（区域之间的网络延迟要高于可用区之间的网络延迟）的角度来说，都非常重要。此外，这种架构可以让应用程序支持各国不同的法规，例如欧盟-美国隐私保护^[2]和GDPR^[3]。

可用区不是数据中心

在一个AWS账户中，你基本可以认为两个不同可用区（例如，us-east-1a和us-east-1b）中的EC2计算实例，分别位于两个不同的数据中心中。

但是，当你使用多个AWS账户时就不一定了。如果你使用账户1在可用区us-east-1a中创建了一个EC2实例，同时使用账户2在可用区us-east-1b中创建了另一个EC2实例，实际上，这两个实例可能就位于同一个数据中心，甚至可能位于同一台物理服务器上。

为什么会这样？这是因为可用区名称并不是静态映射到指定数据中心的，两个不同的账户，相同的可用区名（例如，us-east-1a）可能代表了两个不同的数据中心。

当你创建一个AWS账户时，它会随机地创建一个可用区到指定数据中心的映射，这意味着虽然都是“us-east-1a”可用区，但是它们的物理位置可能相隔甚远。表15-3说明了这一点。这里我们展示了一个区域中的多个数据中心（从1到8），然后演示了在4个示例账户中，可用区与数据中心可能的映射关系。

表15-3：不确定的可用区映射关系

数据中心	AWS 账户 1	AWS 账户 2	AWS 账户 3	AWS 账户 4	...
DC #1	us-east-1a	us-east-1d		us-east-1e	...
DC #2	us-east-1a	us-east-1c	us-east-1a	us-east-1a	...
DC #3	us-east-1b	us-east-1a	us-east-1d	us-east-1d	...
DC #4	us-east-1c		us-east-1a	us-east-1b	...
DC #5	us-east-1d	us-east-1b	us-east-1c	us-east-1c	...
DC #6	us-east-1e		us-east-1b		...
DC #7			us-east-1e		...
DC #8		us-east-1e			...

从这张表中你应当能注意到一些事情。首先，一个账户的一个可用区，实际上可能位于多个数据中心中。这意味着，你在同一个账户、同一个可用区内创建的两个EC2实例，有可能位于同一台物理服务器上，也有可能位于完全不同的数据中心中。其次，在不同账户中创建的两个EC2实例，即使它们的可用区不同，也不能确定它们是否位于一个数据中心。

例如，在表15-3中，如果账户1在us-east-1b中创建了一个实例，账户3在us-east-1d中创建了另外一个实例，这两个实例有可能都位于数据中心3内。

你需要牢记这一点，原因很简单，即使你在两个不同账户的不同可用区中创建了两个EC2实例，并不代表它们是互相独立的，也不能因此作为高可用的依据。

正如在本书的第2章中所讨论的，保持冗余组件的独立性，对系统的可用性和风险管理至关重要。但是，当我们使用多个AWS账户时，AWS的可用区模型并没有强制要求这一点，只有在一个AWS账号下的可用区模型才能做到这一点。

为什么你要使用多个AWS账户？实际上这种现象很普遍，很多公司会为不同的部门或者群组创建多个AWS账户，而AWS这么做可能是出于计费、权限管理或者其他原因的考虑。有时安全策略要求使用多个AWS账户。



当AWS宣布出故障时，它会在其服务状态网站上公布这次故障。但是，当它讨论故障发生在何处时，它会说中断影响了指定区域的“某些可用区”，但不会说具体哪些可用性区受到了影响。

这样做的原因是由于AWS的可用区的映射关系：如果我们假设4号数据中心有问题，对你来说可能是“us-east-1a”，而对其他人来说可能就是“us-east-1c”。它们无法给出具体的可用区名，因为每个账户下的可用区名都是不一样的。

为什么AWS使用这种怪异的映射方法呢？主要原因在于负载均衡。当用户要启动多个EC2实例的时候，如果这些实例平均分布在多个可用区中，那么排在后面的可用区可能就没人去关心了。实际上，相比起“us-east-1e”，人们更愿意使用“us-east-1a”作为常用的可用区名。这主要还是由人所决定的。如果AWS不采用这种人工映射的方式，按照字母表顺序，字母靠前的可用区就会使用过多，而字母靠后的可用区就会使用较少。通过这种人工映射的方式，AWS可以更有效地来平衡系统压力。

如何通过地理多样性真正做到高可用

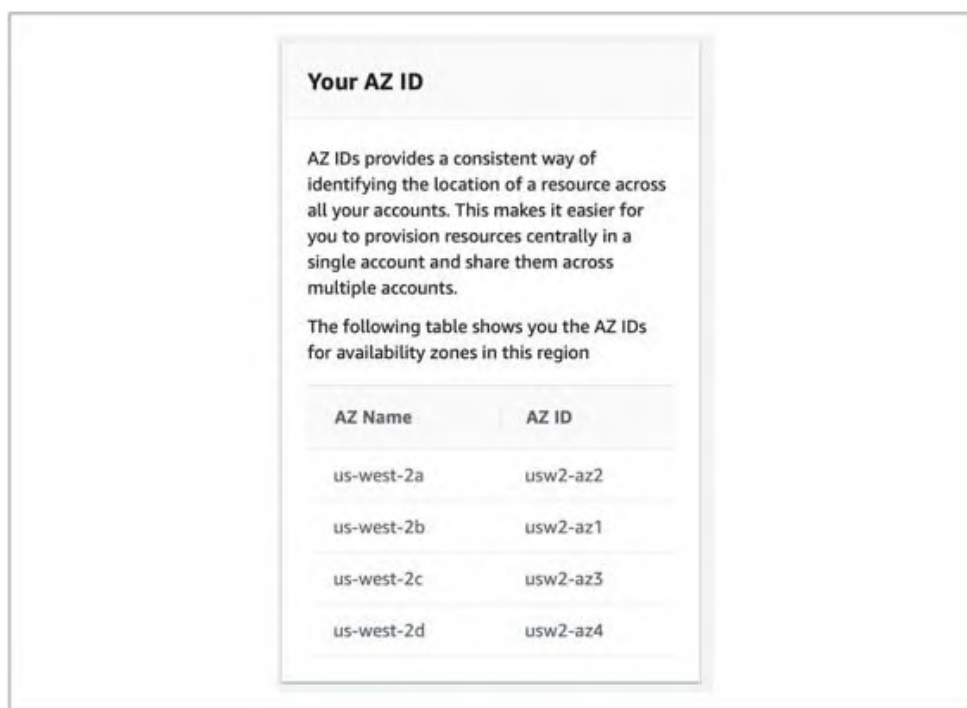
如何能确保使用的AWS资源一定有备份，并且该备份一定位于其他的数据中心，从而真正降低故障所带来的风险呢？

有一些方法可以做到。首先，确保你在同一个账户的不同可用区中维护了备份。如果备份在不同的账户中，确保你在每个账户内的多个可用区中都保留了备份。不要跨账户来比较可用区。

映射多个账户中的可用区

在AWS中，我们可以确定两个账户中的可用区是否可能在同一个数据中心。这需要使用AWS资源访问管理器和AWS控制台。

请登录到AWS控制台并单击“Resource Access Manager”服务。[\[4\]](#)在该页面中，请查找标题为“Your AZ ID”的区域（编写本书时，它位于右边栏大概中间偏下的位置，如图15-4所示）。



Your AZ ID	
AZ IDs provides a consistent way of identifying the location of a resource across all your accounts. This makes it easier for you to provision resources centrally in a single account and share them across multiple accounts.	
The following table shows you the AZ IDs for availability zones in this region	
AZ Name	AZ ID
us-west-2a	usw2-az2
us-west-2b	usw2-az1
us-west-2c	usw2-az3
us-west-2d	usw2-az4

图15-4：AWS某个指定账户的指定区域，AZ到数据中心ID的映射表

在这里你会看到AWS可用区名称（例如，us-east-1a和us-east-1b）和AZ ID之间的映射。你可以将AZ ID看作某个物理数据中心的名称。

AZ ID数字对于所有的AWS账户都是一样的。虽然你不能比较多个账户之间的AZ名称（例如，us-west-2a），但是可以比较多个账户之间的AZ ID（例如，usw2-az2）。通过AZ ID，你可以确定两个不同账户中的两个可用区是否在同一个数据中心中。

该表提供了指定AWS区域中某个指定账户的可用区名称到AZ ID的映射。为了查看不同账户的AZ ID，请使用该账户登录到控制台。为了查看不同区域的AZ ID映射，请从控制台右上角的区域下拉菜单中选择所需要的区域。

分发应用程序

当你决定应用程序将使用哪些可用区时，应当基于避免常见故障点的原则来选择可用区。使用AWS可用区是确保避免常见故障点的良好开端，但是你必须小心并理解AZ映射的实际工作方式，特别是对于跨多个AWS账户的应用程序。如果你不了解这些，就可能会隐藏降低应用程序真实可用性的常见故障点。

[1] 一个区域也可能只有一个可用区。

[2] 作为欧盟安全港的继承者，欧盟-美国隐私保护计划是一套隐私保护原则，用来管理欧盟公民对欧盟以外国家的数据传输行为。它通常关心的是如何存储数据才能符合当地的法律，而AWS区域的概念正是为此提供了支持。

[3] GDPR，或者称为一般数据保护条例（General Data Protection Regulation），规定了欧盟公民的数据必须如何处理。

[4] 或者你可以直接访问资源访问管理器控制台（网址参见链接6）。

第16章

托管的基础设施

当提到云服务时你会想到什么？像大多数人一样，你可能会想到以下几点：

- 文件存储（例如，Amazon S3、Azure Cloud Storage或者Google Cloud Storage）
- 服务器（例如，Amazon的EC2、Azure Servers或者Google Compute Engine）

事实上，你最常用到的也就是这两种资源。

然而，云服务公司提供了其他各种托管服务来帮助你减轻管理上的压力，提高可用性，甚至提升可伸缩性。了解如何组织、管理这些云服务，有助于你决定在应用程序中使用哪些功能。

虽然这里讨论的概念适用于所有云服务提供商，但在本章中，我们将重点以AWS作为示例并加以说明。

基于云的服务架构

基于云的服务有以下三种基本类型：

- 原生资源
- 基于服务器的托管资源
- 无服务器的托管资源

图16-1说明了这三种类型。原生资源只提供基本的服务器虚拟化支持。在虚拟硬件上运行程序的应用程序和操作系统都是由客户拥有和管理的。只有虚拟化层由云服务提供商管理。基于服务器的托管资源遵循相同的基础系统架构，它们仍然在虚拟服务器上运行软件。不同之处在于，在服务器上运行的软件也由云服务提供商管理。在无服务器的托管资源中，运行资源的软件由云服务提供商管理，但是运行软件的基础设施对客户是不可见的，客户不受基础设施管理方式的影响。

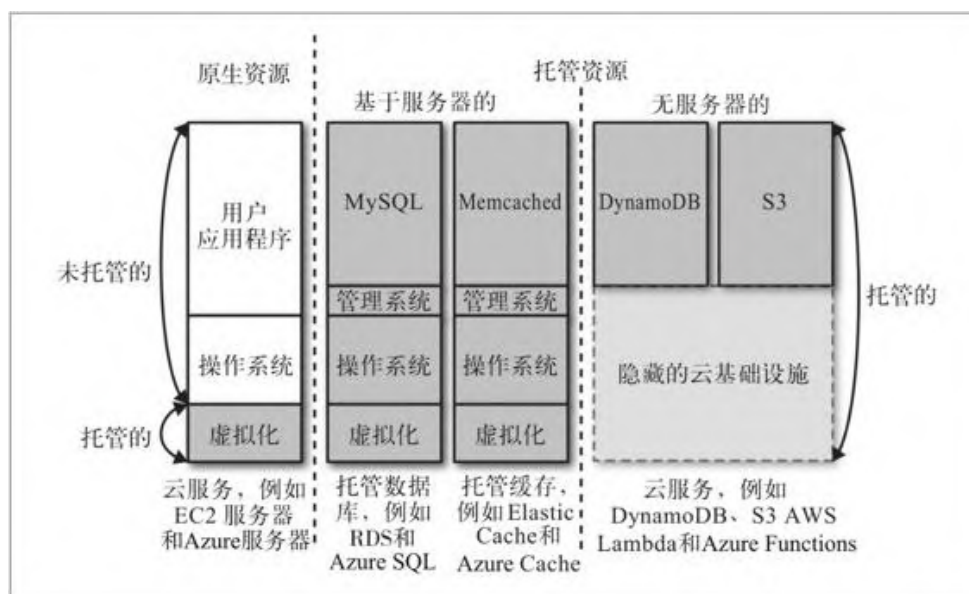


图16-1：基于云的服务类型

让我们更详细地看看这三种类型的云服务。

原生资源

原生的云资源为用户提供了一些基本能力，以及基础的管理功能。原生云资源的例子就是Amazon EC2或Azure虚拟服务器，它们以一种托管的方式提供了原生服务器的能力。

云服务提供了服务器虚拟化的基本管理功能，以及创建实例和初始文件系统的功能。但是，在实例启动并运行后，云服务提供商就无法了解到实例内部的操作了，如图16-2所示。

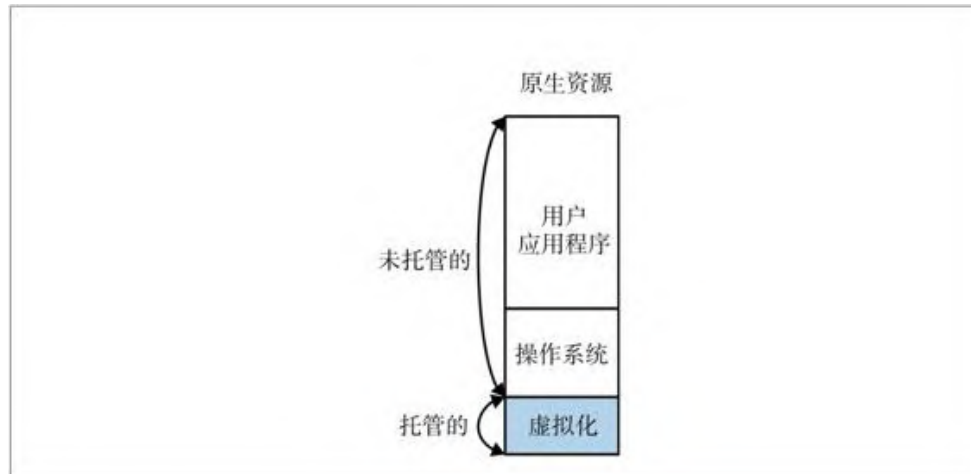


图16-2: 原生资源管理职责

云服务提供商管理着实例输入/输出的数据流，以及CPU和CPU使用率。但是它们并不知道服务器上运行着什么，也不会监控任何内部的运行情况。

像AWS这么做是有目的的。服务器上运行什么是用户的事情，AWS并不想对这一点负责。AWS只负责到虚拟机这个范围，虚拟机内部的事情一概不管。AWS甚至对此有一个名称：AWS共享责任模型。对于AWS的每个服务，该模型描述了AWS的职责是什么，以及客户的职责是什么。以EC2为例，共享责任模型描述了AWS在虚拟化层的责任结束，以及客户在操作系统层的责任开始。

使用原生资源的影响

你可以从不同角度来观察这样做的结果。[\[1\]](#)对于EC2实例来说，AWS会收集各种指标，并通过CloudWatch展现给你，所有这些都是网络级别的指标，其中包括：

- 流入/流出实例的网络流量
- 磁盘数据的读取/写入量
- CPU的使用率

除了这些指标以外，还有一些显然很有价值的指标AWS却并没有提供：

- 可用内存和磁盘空闲数量
- 活动进程数量
- 交换区或者换页情况
- 哪个进程消耗了最多的资源

这些指标不存在，是因为它们依赖于在实例上使用的操作系统，而该操作系统不是由云服务提供商管理的。

你还可以在实例的访问控制中看到原生资源的影响。AWS可以通过ACL管理对实例的网络访问，但是你需要对实例（操作系统）的用户登录功能负责。

基于服务器的托管资源

基于服务器的托管资源，为指定的云服务提供了一个全栈的托管方案。一个很好的例子就是Amazon的关系数据库服务（RDS）或者Microsoft Azure SQL数据库。这些服务在托管服务器基础设施之上提供了一个托管的数据库应用程序。图16-3显示了如何管理基于服务器的托管资源。

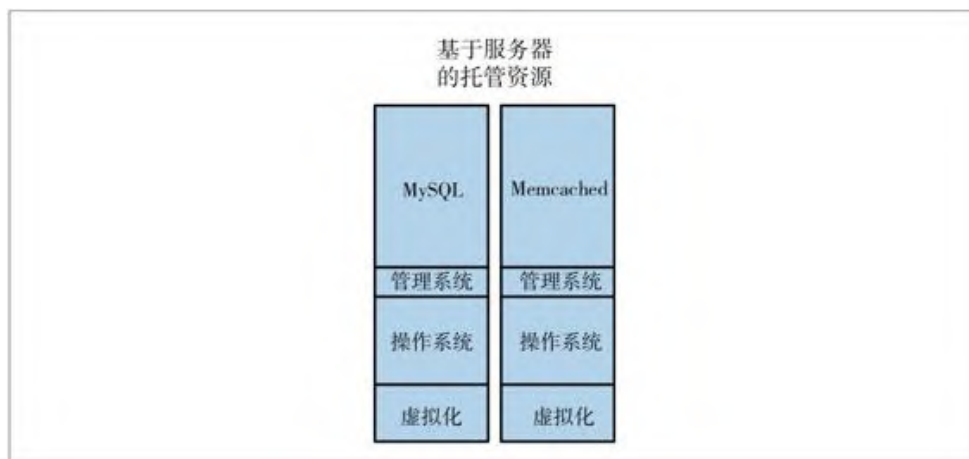


图16-3：基于服务器的托管资源管理职责

托管数据库解决方案（例如，Amazon的关系数据库服务或Microsoft的Azure SQL）在现有的托管服务器上运行数据库和特殊管理软件，使得云服务提供商能够管理服务器上的整个堆栈、服务器和软件。数据库软件是行业标准数据库（如MySQL、PostgreSQL或SQL Server），在标准托管服务器上以完全托管的方式运行。这些服务从头到尾提供了完整的托管数据库解决方案。

我们再回顾一下图16-1（图16-3显示了该托管堆栈的详细视图），你可以看到Amazon的关系数据库服务（RDS）的架构。基本上，当你启动一个RDS实例时，就启动了一个运行着特定操作系统、特定管理软件以及数据库本身的EC2实例。Amazon不但负责管理EC2服务器，还包括整个软件堆栈、操作系统和数据库软件。

使用基于服务器的托管资源的影响

你可以通过查看由RDS实例提供的CloudWatch指标，来了解RDS的运行情况。除了基本的EC2实例信息之外，你还可以了解到数据库本身的一些监控信息，例如：

- 数据库的连接数
- 数据库使用的文件系统空间
- 数据库当前的查询数量
- 数据库复制的延迟时间

这些都是只能从操作系统或者数据库软件获取到的指标。

另外一个了解服务的方式就是学习你可以操作的各项配置。除了可以配置服务器的基本信息（例如，网络连接数和挂载磁盘），还可以配置一些数据库本身的信息，例如，最大连接数、缓存信息，以及其他配置和性能调优参数。

无服务器托管资源

无服务器托管资源是提供某些特定功能的资源，但不公开该功能所运行的服务器基础架构。在AWS中有几个这样的例子，包括Amazon S3、DynamoDB和AWS Lambda。Azure Functions就是这样一个无服务器的托管服务的例子。图16-4显示了如何设计和管理无服务器托管资源。

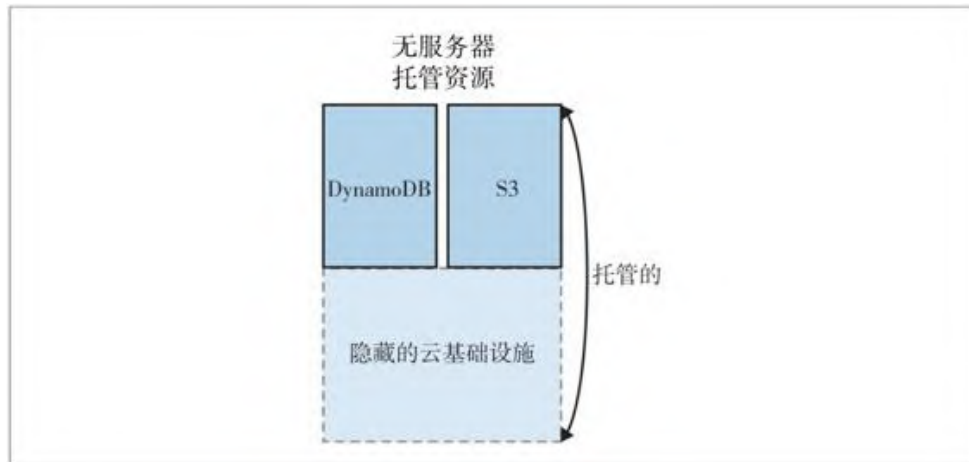


图16-4：无服务器托管资源的管理职责

对于无服务器的托管资源，一个很好的例子就是Amazon S3服务。这个服务提供了基于云的文件存储和传输服务。当在S3上保存一个文件时，你会直接和S3服务进行通信，Amazon不会为你分配任何服务器。实际上，这时可能会有一台或多台服务器在背后为你完成操作。

整个执行过程是可以托管的，但是你能只能查看文件，以及调用暴露出来的服务接口（例如，在S3上进行上传、下载、删除文件等操作）。你无法了解或控制任何在服务底层运行的操作系统或者服务器。这些服务器由所有使用服务的用户共享使用，由Amazon统一进行管理和控制，无须用户干预。

另一个无服务器服务的例子是AWS Lambda。该服务提供基于云的函数执行。与Amazon S3中的情况一样，它没有为你分配执行操作的服务器。在后台运行一台或多台服务器来执行请求，这件事情对你来说是不可见的。

为什么使用无服务器

无服务器服务的最大优点之一，是它们不需要你采取额外的操作来分配资源就可以进行伸缩。如果应用程序的流量突然增加，Amazon 将自动应用适当的资源来处理Amazon S3或AWS Lambda增加的需求。你不需要为此分配额外的资源，全部由AWS管理。

这与基于服务器的托管服务不同，后者要求你根据预期的流量负载采取操作。在创建Amazon RDS数据库实例时，必须调整它的大小以满足预期的流量需求。如果你的流量低于这个水平，就是在浪费资源。如果你的流量需求提高到这个水平以上，你将面临数据库容量耗尽和请求不足的风险。你必须管理应用程序的资源分配。

使用托管资源和非托管资源的影响

使用某个托管服务或者其中的一部分功能，会给你带来很多好处，主要包括以下几点：

- 你不需要安装或者升级托管系统上的软件。
- 你不需要调整或者优化系统（但是可以通过云服务提供商进行调优）。
- 你不需要监控或者保证软件运行的性能。
- 你不需要安装任何软件，云服务提供商就可以提供一些所需的监控数据。
- 云服务提供商可以为服务提供备份和复制的功能。
- 云提供提供商可以为你的服务提供更高级别的安全性。

但是托管服务也有不足之处：

- 不能改变软件的运行机制。
- 无法控制何时、如何去升级软件，以及当前运行的软件版本。 [\[2\]](#)
- 只能使用由云服务提供商提供的监控数据和配置项。

如果你不使用托管服务，也有一些好处，主要包括以下几点：

- 你可以控制运行什么软件，运行什么软件版本，以及如何进行安装。
- 你可以控制软件升级的时间和方式，以及决定是否需要升级。
- 你可以按照需要的内容和方式来监控和控制软件。

当然，使用非托管资源也有以下弊端：

- 没有免费的午餐，你需要自己负责系统的所有管理和维护工作。
- 你需要自己实现数据备份和复制机制。
- 你需要自己监控软件，以保证它们正常运行，否则无人会知道何时发生了故障。
- 如果软件运行出现问题，也只有你自己能够去修复它。云服务提供商无法提供任何帮助。
- 你必须管理服务的安全性，你的云服务商几乎无法提供同样多的帮助。

小结

了解一个服务是否是托管的，以及它是基于服务提供器的还是无服务器的，可以在一个高度可伸缩的应用程序中，帮助你充分使用服务以及决定如何使用服务。

[1] 关于使用CloudWatch监控AWS的更多信息，请访问链接7所指向的网址。

[2] 不过，云服务提供商也会提供某些功能。例如，RDS提供了支持的数据库版本列表，但并不支持所有版本。但是对于像S3这样的服务，你根本无法控制软件的升级过程。

第17章

云资源分配

在构建高可伸缩的应用程序时，部署应用程序的一个重要步骤是分配操作应用程序所需的资源。资源可以是任何东西，从计算机实例到数据存储。如何分配这些资源，以及如何确定应该分配哪些资源，这对你的应用程序很重要。如果为应用程序分配的资源太少，就会导致应用程序无法运行，从而产生可用性问题。如果分配的资源太多，可能会有太多闲置的资源，以造成大量浪费。

这是与所有高可伸缩应用程序的斗争，对于有流量高峰的应用程序来说更是一个问题。如果你的应用程序在一个相对较短的时间内使用率极高，而在其他时间使用率明显较低，那么如何有效地分配资源可能是一个问题。

这是云计算的主要优势之一。通过云计算，你可以根据需要动态地分配资源，以便有效地处理流量高峰时的请求，并且不会在非繁忙时段留下大量未使用的资源。

但是管理云资源并不是一项简单的任务，需要仔细考虑。要想成功地管理云资源分配需求，而不造成浪费或者资源不足，需要你了解云中的资源分配是如何工作的。你需要理解它们是如何分配、使用和计费（最重要）的。

云资源通常可以被划分为以下两类：

- 基于使用量的资源
- 固定额度的资源

所有云资源都属于这两类中的某一类，类别不同，对应管理这些资源的流程也有很大差异。

让我们分别讨论这两类资源的使用方式。

基于使用量的资源分配

基于使用量的资源不是被分配的，而是按照应用程序的使用率来计算的。它们无须分配，只会按照实际使用量计费。

基于使用量的云资源有以下特征：

- 没有资源分配的步骤，也无须进行容量规划。
- 如果应用程序需要较少的资源，你就可以使用较少的资源，收费也较低。
- 如果应用需要较多的资源，你就可以使用较多的资源，收费也较高。
- 在合理范围内，你不需要做任何事就可以将使用量从一个极小值扩大到一个非常大的值。
- “合理范围”由云服务提供商根据其能力来统一定义。
- 通常情况下你不需要知道这些资源是如何分配、伸缩的，这些对你都是不可见的。

Amazon S3就是一个经典的基于使用量的云资源。S3服务是按照你存储和传输的数据量来计费的。你根本不用预先估计需要多大的存储空间，或者多大的数据传输量。无论你需要多大的量（在系统限制范围内），S3都能支持，你只需要为实际的使用量付费。

以下是基于使用量的资源的附加示例：

- Azure云存储
- AWS Lambda
- Azure Functions

- Amazon简单邮件服务

因为不需要提前规划容量，所以这种服务易于管理和伸缩。这种基于使用量的资源才是云服务带来的真正好处之一。这也得益于云服务多租户的特点。

在Amazon S3这样的服务背后，是海量的磁盘存储系统和海量的服务器，按照每个用户的每个请求按需分配。如果你的应用程序对资源有需求高峰，就可以从一个共享的可用资源池中分配。

这些可用资源池被所有的用户共享，所以这个底层资源池可能非常巨大。当你的资源使用高峰逐渐消退时，另外一个用户的使用高峰可能刚刚开始，这样本来服务于你的资源就会被分配给另一个应用程序使用。这些资源转移再分配的过程对你来说都是透明的。

只要可用资源池足够大，就足以处理所有请求，以满足所有用户的使用高峰，不会出现资源不足的情形。服务的规模越大（使用服务的用户越多），对云服务提供商来说，平衡资源使用高峰和计划足够容量的能力也就越强。

用量大户

只要没有单个用户占用大量可用资源的情况出现，这种模型就是有效的。如果某个大户占用了资源池中的大部分资源，那么它在高峰期间就会遇到资源不足的情况，并且也会影响其他用户可以使用的容量。

但是像Amazon S3这样的服务，它们的规模如此巨大，以至于没有用户可以用掉大部分资源，所以它的资源分配机制依然成谜。[\[1\]](#)

然而，即使是Amazon S3也有自己的限制。如果你的应用程序需要存储或者传输海量的数据，那么就会遇到一些S3的限制，目的是防止造成其他用户的使用资源不足。因此，大量使用S3资源的服务会遇到这些人为限制，并且自己也会遇到资源不足的情况。这种情况一般只会发生在PB级数据的存储和传输场景下。

即使你真的需要使用如此多的S3资源，也有一些办法能够突破这种限制。除此之外，你还可以联系Amazon申请提高限制的阈值。它们会根据你的需求，将这个值上升到合理的范围，并将其加入后续的容量规划中，以此来保障有足够可用的容量来满足你或者其他用户的需求。

固定额度的资源分配

固定额度的资源指的是已经预分配在分散单元中的云资源。你提出需要多少某种资源，然后系统帮你分配。资源总量是按照你的要求进行预分配，而不是按照你的实际使用情况来实时分配。

固定额度的资源有以下一些特点：

- 分散在不同的单元中。
- 你需要指定需要多少资源，系统为你进行分配。
- 如果应用程序使用的资源少于所分配的资源，那么就会剩下一部分未使用的资源。
- 如果应用程序需要更多资源，那么预分配的资源可能会不够用。
- 需要合理计划资源的使用情况，避免出现过度分配或者分配不足的情况。

固定额度资源分配的典型例子就是服务器，例如，Amazon EC2实例。你可以指定需要的实例数量以及服务器大小，由云服务将它们分配给你使用。此外，一些托管的基础服务，例如云数据库也使用这种分配方式。对于这些服务，你只需指定服务的单元数量以及单元大小，云服务会帮你分配好资源。

以下是一些固定额度资源分配的其他例子：

- Amazon RDS
- Amazon Aurora

- Azure SQL
- Amazon ElastiCache
- Amazon Elasticsearch Service
- Azure Cache

不过也有一些固定额度资源在操作上会略有不同，例如，Amazon的DynamoDB服务。在使用DynamoDB时，你需要指定DynamoDB表的容量，这样就不是按照服务器的大小来计算，而是按照吞吐量来计算容量了。^[2]你指定要为表分配多少额度，它们就可以使用多少。如果你没有使用这么多额度，剩下的就浪费了。如果系统的使用超出了分配的额度，就会出现资源不足，直到被分配更多的额度。因此，虽然看上去不同，但其实它们也是按照与服务器相似的方式进行分配的。表17-1显示了AWS几个主要的固定额度资源服务和每个服务使用的分配单元。

表17-1：固定额度资源服务的分配单元

AWS 服务	容量分配单元	分配属性
Amazon EC2	实例小时数	实例大小，运行小时数
Amazon RDS	实例小时数	数据库大小，运行小时数
Amazon Aurora	实例小时数	数据库大小，运行小时数
Amazon ElastiCache	实例小时数	缓存大小，运行小时数
Amazon DynamoDB (固定额度)	吞吐容量单元	分配的写入数量 分配的读取数量
Amazon DynamoDB (按需)	请求单元	利用的写入数量 利用的读取数量
Amazon DynamoDB (数据存储) ^a	存储的 GB 数量	消耗的按需存储

^a DynamoDB的数据存储是基于使用量的资源。此表中包含的内容说明服务可以同时使用多种类型的分配机制。

调整分配

通常情况下，额度都是按照一定梯度进行分配的（例如，服务器按照每小时计算，DynamoDB的容量也按照每小时计算）。你可以调整分配给应用程序的服务器数量，或者分配给DynamoDB表的容量，但是只能按照一定的梯度进行调整（例如，服务器的大小，或者容量单元的大小）。虽然有不同的梯度可以选择（例如，大小不同的服务器），但你每次必须分配整个服务器或者整数容量。

你有责任确保手头拥有充足的资源。这类似传统的数据中心对服务器的容量规划。你需要先按照当前需求去分配容量，等到确认容量需求有变化后再重新分配。这是传统的、不基于云分配服务器的方式，但也可以用于基于云的服务器分配。但是，还有其他更自动化的方法来更改容量分配。

资源容量的自动分配

云上的分配比传统数据中心的容量分配更加简单。因此，我们可以使用一些算法来自动改进分配策略。例如：

按需

可以使用一个静态分配，然后等待，直到你消耗了大部分已分配的容量。此时，你可以根据需要再增加容量分配。

固定周期

你可以根据与使用模式匹配的固定计划，自动更改你的分配。例如，可以在大量使用的白天时间增加可用的服务器数量，在使用较少的夜间时间减少服务器数量。

自动化（自动伸缩）

你可以监视资源的特定指标，来确定资源何时被大量使用，何时被轻度使用。然后，根据这些数据，你可以动态和自动地分配额外的资源，或者回收过多的资源。你可以在应用程序中构建这种自动伸缩机制，或者使用云服务提供的自动伸缩机制，例如，Amazon EC2 Auto Scaling，它会根据配置的指标和条件自动分配和释放EC2实例。

无论选择哪种方式调整资源分配，要记住一点，已分配的容量就是你的可用容量，有可能部分容量被浪费掉，但更糟的是出现资源不足的情况。

自动分配的问题

即使你使用了自动分配的方案（例如，Amazon EC2 Auto Scaling 为应用程序提供额外的容量），也不意味着自动伸缩算法在应用程序的资源变得匮乏之前，就能足够快地注意到这种需求。当你的资源需求本来就非常缺乏时，这尤其是一个问题。这种现象称为容量分配倾斜，即使使用自动伸缩的分配方法，也会导致资源饥饿或者资源闲置。

以Amazon的弹性负载均衡（ELB）为例，它会根据当前的流量自动伸缩，为应用提供负载均衡的功能。如果请求的流量很小，ELB就会减少用于负载均衡的服务器数量，如果请求流量变大，ELB会自动增加更多的服务器来处理请求。这些对用户来说都是自动并且透明的。这就是ELB如何能够收取较低的入门费用，但是又可以在流量激增时自动扩容（增加相应的价钱）的原因。这样你可以在流量低时节省费用，在流量高时自动扩容。

但是，在某些场景下，这种自动分配机制可能会带来一些副作用。如果你的应用因为一些社交媒体活动突然走红，流量突然达到一个高峰，负载均衡器可能来不及迅速进行扩容，那结果会怎样呢？在请求量升高一段时间之后，负载均衡器的资源可能逐渐不足，导致页面请求响应变慢，或者无法响应，从而破坏用户体验。虽然当ELB发现流量增加之后会自动增加处理负载均衡的机器，自动改善资源不足的情况，但是这种扩容通常需要几分钟才能完成。在这个过程中，用户的体验会下降，系统可用性也会受到影响。

为了避免这种影响，你可以联系Amazon的客服代表，提前预警可能到来的流量变化，这样他们可以提前对负载均衡器进行预热^[3]，可以在流量高峰到来之前，提前对负载均衡器进行扩展（使用更好和更多的服务器）。不过这只对一些能预料流量增加的情况有效，对毫无征兆的突发情况不起作用。

动态分配，动态成本

你通常可以随时调整容量分配，[\[4\]](#)根据需求去增加或减少资源。

这就是云服务的优势之一，如果当前每小时需要500台服务器，而下一个小时只需要200台，那么你只需要支付一小时500台和一小时200台的费用。这种方式简单明了。

但是，由于分配资源容量所赋予的灵活性，你会为此多花一些钱。

如果你的需求很稳定怎么办？如果你经常需要至少200台服务器怎么办？当你对服务器需求很稳定的时候，为什么要为按小时计费的灵活性付费呢？

预留容量

于是预留容量出现了。预留容量是指，你预先向云服务提供商承诺在一段时间内（比如1~3年）使用一定量的资源。作为交换，你可以享受到一定的优惠。

预留容量并不会限制分配资源的灵活性，它只是向云服务提供商保证能够使用一定量的资源。

假设你的应用程序平时需要200台服务器来支撑，在流量高峰时需要扩容到500台。你可以使用自动扩容功能动态调整服务器的数量，因此服务器的使用数量会在200到500之间动态变化。

因为你一直都需要使用至少200台服务器，所以你可以购买200台服务器的预留容量。比如你购买了1年200台服务器的使用量，这200台服务器的价钱相对较低，虽然你需要一直付这么多钱，但是还好，因为你一直都需要使用这么多服务器。

至于另外的300台服务器（500-200），你可以按以小时计费（价格更高）的方式付费，不过只是使用的时候才付费。

通过承诺预先分配资源，你可以较低的价格获得这些资源。[\[5\]](#)

资源分配技术的利与弊

如表17-2所示，基于使用量的资源分配和固定额度的资源分配都有相应的利弊。

表17-2：云资源分配技术之间的比较

	固定额度分配	基于使用量分配
服务示例 (Amazon AWS)	EC2、ELB、RDS、DynamoDB、Azure SQL、Azure Servers	S3、Lambda、SES、SQS、SNS、Azure Functions
需要额度计划	是	否
收费依据	分配的额度	花费的额度
未使用的情况	额度闲置	N/A
超出使用的情况	应用程序资源匮乏	N/A
是否可以预约额度以节省开支	是	否
如何伸缩额度	手动或者通过脚本来变更额度，可以延迟变更	自动并及时变更
如何处理用量峰值	允许可能的用量匮乏，或者提高额度	无感知处理
如何处理超出的额度	已分配，并且只能供你使用	放到全局资源池中，可供其他客户使用

固定额度的分配方法需要预先进行容量规划，而基于使用量的分配方法则不需要。使用固定额度的资源分配，你将根据请求的额度而不是实际消耗的额度来收费。这意味着可能会浪费额度，或者产生应用程序资源匮乏的情况。

[1] 根据Amazon最新发布的相关数据，2013年S3系统存储量已达2万亿个存储对象，如果把这些存储网络比喻成银河系，每个存储单元是一颗星的话，那么每颗星可以容纳5个存储对象。详情请参考链接8所指向的文章。

[2] DynamoDB也支持按需使用的定价模型，它更类似一种基于使用量的资源。

[3] 更多信息请参考文章《ELB最佳实践》，网址参见链接9。

[4] 有些服务有限制，例如，DynamoDB，对调整容量的频率有一定的限制。

[5] 使用预留资源还可以保证这些资源分配在你想要的可用区中。如果不使用预留资源，当你希望在指定可用区中创建某个实例时，AWS可能无法满足这一点。

第18章

无服务器计算和函数即服务

函数即服务（Function as a Service, FaaS）产品，例如，AWS Lambda和Azure Functions，都是相对较新的软件执行环境，为在不需要服务器的情况下创建简单的微服务提供了可靠性。因此，业界创造了术语无服务器（serverless）来指代这些类型的执行环境。[\[1\]](#)

FaaS产品提供了事件驱动的计算功能，不需要购买、搭建、配置或者维护服务器。FaaS产品，例如，AWS Lambda和Azure Functions，提供了几乎无限的可伸缩性，可以在次秒级的级别上进行支付。

AWS Lambda之类的服务可以伸缩到几乎任何合理的规模，而不需要采取任何操作。这就是FaaS的真正力量。

以下是一些常见的使用场景：

- 新上传图片的格式转换。
- 实时的指标数据处理。
- 流式数据的验证、过滤和转换。

它适合于以下任意一种形式的数据处理：

- 应用程序或环境中某个事件发生后需要进行一些操作。
- 数据流需要进行过滤或者转换。
- 对传入数据进行必要的边界值校验。

目前有很多关于FaaS的炒作。FaaS服务并不是万能的，它们的实际功能只在特定类型的架构中非常有用。以下是一些可以有效利用FaaS的特定类型的应用程序。这些示例使用了AWS云和AWS Lambda FaaS服务。

示例1：事件处理

以一个图片管理程序为例。用户可以将图片上传到云服务并存储在S3这样的存储服务中，应用程序会显示图片的缩略图，并且让用户更新图片的相关属性，例如名称、地点、图片中的人名等。

这种简单的应用程序就可以使用AWS Lambda对用户上传到S3后的图片进行处理。当用户上传一张图片后，会自动触发一个Lambda函数，为该图片生成一个缩略图并上传到S3。同时，另一个Lambda函数会获取图片的多个特征（例如，尺寸、分辨率等）并将这些元数据存储在数据库中。随后，图片管理程序可以提供操作数据库中元数据的功能。

这个架构如图18-1所示。

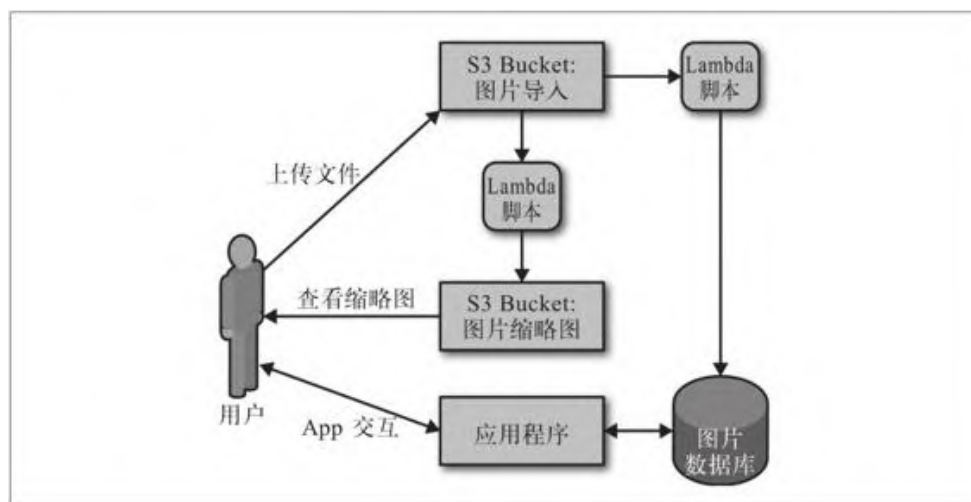


图18-1：文件上传Lambda的用法

图片管理程序无须参与图片的上传过程，它可以依赖于标准的S3上传功能和这两个Lambda函数，由它们完成文件的所有上传过程。这

样，图片管理程序只需要考虑它所擅长的事情：操作已有图片在数据库中保存的元数据。

示例2：手机应用后端

假设有一个手机游戏，在云服务中存储了用户的游戏进度、奖品以及高分榜，这些数据用于共享社区以及个人用户的移动设备上。

这个应用程序需要在后端创建一系列API，来完成将数据存储到云服务中、从云服务中获取用户信息，以及进行社区交互的功能。云计算后端运行在AWS上。

这些API都是通过一个关联了许多Lambda函数的API网关来创建的。^[2]这些Lambda脚本会执行一些必要的数据库操作，来处理游戏的云服务后端工作。

该架构如图18-2所示。

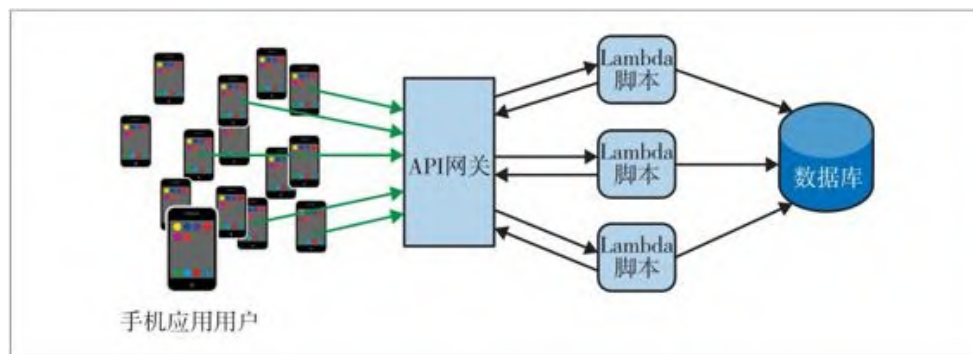


图18-2：手机后端Lambda用法

在这个例子中，后端不需要任何服务器，并且所有扩展都是自动处理的。

示例3：物联网数据采集

假设有一个应用程序，可以获取分布在全世界各地的大量传感器数据。由于这些传感器会定期向应用程序发送数据，所以应用程序的

服务器端会定期收到大量的数据，需要将它们保存在某种数据存储库中。这些数据由某些后端应用程序所使用，我们不在这个例子里详细讨论它们。我们只关心数据采集过程。

数据采集模块需要对数据进行验证，可能会执行一些简单的数据处理，然后将结果保存到存储器中。

这个程序只是简单地执行一些基本的数据验证操作，然后将数据保存下来供以后使用。虽然应用程序比较简单，但是必须以巨大的规模运行才能够支持每分钟百万甚至十亿次的数据采集，而实际的规模取决于传感器的数量以及传感器生成的数据数量。

该架构使用了数据采集流水线^[3]，将数据发送给一个AWS Lambda函数，由该函数在数据存储前进行必要的过滤和处理。

该架构如图18-3所示。

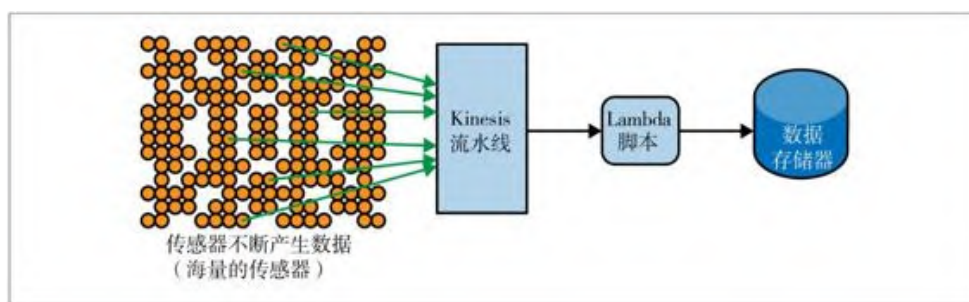


图18-3：物联网传感器采集示例

Lambda非常适合处理那些需要定期高速抽取的海量数据。

FaaS的优缺点

FaaS有一个主要的优点：可伸缩性。例如，AWS Lambda非常善于处理大流量负载，同时无须扩展应用程序所在的底层基础设施。

为了达到这一点，它需要代码本身运行起来极其简单，这样才能方便地在多台服务器上快速、高效地并行执行，按需伸缩。

这就是FaaS的核心：以超大规模运行小型代码片段。这个核心使得AWS Lambda对于上面所有三个示例架构来说都是一个有效的解决方案。

那么在什么场景下你应当使用FaaS呢？为了回答这个问题，我们先了解一下FaaS的缺点：

- 隐含的编程需求（简单、事件驱动、快速处理）。
- 复杂的配置和安装。
- 通常没有原生的内置预发布或测试环境。
- 通常没有原生的部署/回滚功能。
- 通常没有能够开发、测试FaaS函数的开发环境，或者环境受限。

简而言之，FaaS在小型脚本的伸缩性上做得很好，但是不适合用来部署大规模的应用程序。它不适合执行复杂的计算和复杂的交互，不适合复杂的代码和执行环境。代码越复杂，在FaaS中使用就越不理想。有一些专注于无服务器计算的公司正在努力解决其中的一些问题，比如改善开发和部署环境。但这些还都是新的、未经测试的工具和功能。

如果用得好，FaaS可以极大地帮助你满足规模上的极度伸缩。但是，你需要将它用在适合的场景下，对于那些超出FaaS能力范围的需求，请使用其他的部署和执行方式。

无服务器炒作和FaaS的未来

有些人认为AWS Lambda和其他FaaS产品正在“接管世界”，并将最终导致传统的基于服务器的计算和容器技术（例如，Docker和Kubernetes）过时。我坚决反对这种论断。FaaS计算服务，例如，AWS Lambda和Azure Functions，为特定类型的计算需求提供了一个强大的、高可伸缩的计算能力。但是，它们不会“接管世界”，也不会取代传统的基于服务器和基于容器计算的需求和价值。

请小心强制使用无服务器函数来解决你的所有计算需求。有些人试图让他们的应用程序“百分之百地在Lambda上运行”。FaaS计算并不是解决所有计算需求的完美方案，它不应该被强迫扮演这个角色。在我看来，基于容器和服务器的软件总是比无服务器/FaaS在计算版图中占据更大的一块。

为了正确的目的使用正确的工具，不要强迫FaaS用在它不适合的地方。

[1] 术语“无服务器”也适用于云服务产品，比如，DynamoDB、S3和Azure Cloud Storage。这是一种对“函数即服务”（FaaS）的不同解释。

[2] Amazon的API网关服务是一个与AWS Lambda配合创建API的服务。

[3] Amazon Kinesis是一个实时的流式数据采集流水线，用来处理大量的数据流请求，经常与AWS Lambda一起使用。

第19章

边缘计算

到底什么是边缘？边缘是监测一个农场的天气和干旱情况，以确保收获最佳的作物产量。边缘是一架自动无人机，可以单独飞行、拍照、收集环境或地理数据。边缘是一个半拖车，将它的位置、货物量和运行状态等信息传送给一个中央运输系统。边缘是一款智能家电，当你的东西快用完时，它会自动知道，并帮助你订购更多的东西。边缘是一款智能家居设备，它可以监控并保护我们的安全，比如在发现火灾时关闭炉子，或者在你不在家时打开警报。

所有这些都是边缘计算的例子。每一种都是一种新颖的用法。当我们想到边缘计算的时候，这些例子会浮现在我们的脑海中。但是究竟什么是边缘计算呢？边缘计算是应用程序的一部分，它使应用程序更接近操作所在的位置。

我们所说的“操作”是什么意思？这里的“操作”指的是你想要处理的数据来源。这可能是应用程序的终端用户，或者一个被控制的系统。或者它可能放在你的应用程序的最后，代表程序存在的原因，即它被设计和建造的目的。

边缘计算就是把计算放在靠近需要的地方。边缘计算的定义很简单，就是把计算放到它应该在的地方。

因此，当我们监测一个农场的干旱情况时，正在从难以到达的地方收集大量的数据。当我们谈到一架自动无人驾驶飞机时，我们指的是它保持在空中，并且不受风和天气的影响，没有人类的参与。当我们谈到半拖车时，它正在收集有用的信息，比如卡车的位置，它是否能够以安全的速度移动，它使用了多少燃料，其上的货物状况如何。对于一个自动化的家庭设备来说，理解什么时候正在发生危险，并采取行动防止危险蔓延，这就是一种智能。

当涉及面向可伸缩的架构时，边缘计算是不同的。关于如何构建边缘计算服务的规则、它们伸缩的方式，以及它们利用基础设施的方式都是不同的。让我们仔细研究一下边缘计算，看看它与云计算中的传统计算有什么不同，以及我们需要如何从可伸缩的角度用不同的方式处理它。

如今的边缘计算

这些都是边缘计算的应用示例，但是这些例子中的大多数都是超出我们日常经验的。我们还没有看到自动无人驾驶飞机在头顶飞过，也没有看到微小的天气变化对农业的影响报告。

但是，边缘计算最重要的应用如今确实存在，而且每天都有更多的应用变得实用起来。你不需要等待太长时间就能够看到边缘计算的广泛应用：

- 请你去当地的杂货店。扫描仪正在为销售点的机器收集数据，这可以在将结果发送到云端之前知道你欠了多少钱。
- 请你去看看当地的联邦快递代理。它们使用扫描仪来跟踪你的包裹，这样你就可以随时知道包裹的位置和到达时间。
- 或者请你看看自己。当你单击最喜欢的咖啡店应用程序中的按钮之后，就可以等待咖啡的到达。

在所有这些情况下，你都在使用边缘应用程序，并且处于边缘计算之中。

甚至在家里，你会在浏览器的智能网页客户端中阅读电子邮件吗？是的，这也是边缘计算。电子邮件应用程序同时具有服务器组件和边缘组件。边缘组件正在浏览器中运行。它离用户很近，为了给你带来更好的用户体验。

没错，这也是边缘计算。

所有这些都是边缘计算的应用，它们都在利用边缘设备上运行的边缘软件。虽然细节不同，但是应用程序的基本结构是相同的。所有

这些都是边缘计算的例子。

为什么我们要关心边缘计算

为什么我们要关心什么是边缘计算，什么不是？因为构建、运维和伸缩边缘应用程序的模式，与构建、运维和伸缩云端应用程序和服务端应用程序的模式不同。伸缩边缘应用程序或者服务的需求，与伸缩云应用程序或服务的需求非常不同。保证一个边缘应用程序或服务运行的要求，也与保证云端应用程序或服务端程序运行的要求不同。

伸缩性和可用性都会受所使用的计算类型的影响，无论计算发生在边缘中还是在云中。

与云计算相比，边缘计算应该是什么

如果伸缩一个应用程序或服务，以及保证其高可用性，会受到服务是否处于边缘的影响，那么我们应该如何决定将计算的哪一部分放在边缘中或云中呢？

换句话说，是什么让这个边缘成了边缘？

为了回答这个问题，让我们回到边缘计算的目的上来。边缘计算的目的是让时间敏感的操作更接近需要它们的地方。这意味着：

- 如何保证无人机在所有条件和环境下都能够安全飞行。
- 如何保持浏览器电子邮件应用程序的响应性，即当你单击一个按钮时，应用程序应该立即响应。
- 如何保证家庭安全系统正常工作，即使与互联网（即云计算）之间的连接是不可靠的。
- 如何让你的移动应用程序及时响应你的交互。

这与一般云计算中典型的集中式计算不同。集中式的计算是进行数据收集和分析的地方，但是这里是订单发生的地方，是与他人和系统发生通信的地方。

大型计算应用程序架构就是把计算放在能够保持计算高效运行的地方。而边缘计算成功的关键是，将计算放在它可以高效运行的地方，而不一定放在对开发人员和运维人员最方便的地方。



边缘计算应该是什么？

边缘计算就是把计算放到它应该可以高效运行的地方，而不是放在对开发和运维方便的地方。

为什么会这样？因为把计算放到边缘要比把它们都放在云端更困难，风险也更大。构建和维护边缘软件比构建和维护云端的服务器软件更加困难。

所以当我们把计算放在边缘的时候，我们有充分的理由这样做。

如何决定把什么放在边缘？以无人驾驶汽车为例

那么如何决定将一些计算放到云端还是放在边缘呢？为了进行演示，我们来看一个示例，其中云计算和边缘计算都是应用程序成功的必需条件。这也是一个如今备受关注的例子，它需要大量的边缘计算和云计算：无人驾驶汽车。

无人驾驶汽车是一个独特的“野兽”。生产和操作一辆可以独立于驾驶员操作的汽车，需要大量的软件和大量的计算。这是人工智能和数据处理相结合的最新技术。

让我们来看看一辆无人驾驶汽车由哪些部分组成。

无人驾驶汽车中有很多传感器和控制装置。它通过传感器来探测障碍物的位置和道路的位置，通过摄像头来检测你前面的东西是（a）你跟随的车，（b）正在过马路的行人，还是（c）“道路封闭”的障碍物。或者它只是一个滚过街道的球，可能正在被一个小孩子追逐？

检测、决定、理解和应对每一种可能性，对各方的安全是至关重要的。

无人驾驶汽车也有操纵汽车的控制装置。当然，它有转向控制、刹车控制和动力控制。但是它也会通过控制和传感器来监测汽车本身的健康状况。马达运转正常吗？油箱里有足够的汽油吗？油压可以接受吗？乘客座位够凉吗？我们现在应该打开安全气囊吗？

所有这些都需要计算。有些计算必须在汽车中进行，但有些可以在云端进行，那究竟什么计算应该发生在哪里？有些事情很自然应该发生在车里，而有些事情应强制发生在车里。下面几个计算的例子都必须发生在车中：

图像识别

- 在我附近的是一个人还是另一辆车？

威胁检测

- 那个人正在我前面跑，还是我前面的那辆车在刹车？

公路管理

- 道路的边缘在哪里？我前面是一个停车标志吗？

碰撞控制

- 我需要快速刹车并且右转以避免撞车吗？

所有这些都是必须立即发生的、对时间敏感的计算。它们不能因为互联网连接情况不良而无法工作。它们不能因为云服务器正忙于处理其他请求而延迟计算。所有这一切都必须自动地立即发生，而且每次都必须准时发生。它必须总是可用的。

这些计算必须在车里进行，这就是无人驾驶汽车的边缘计算。

但是无人驾驶汽车还需要进行其他计算，它们可以而且应该在云端进行。可以在云端计算的例子有：

行驶方向

我怎么从A点行驶到B点？哪一条是最佳路线？

道路状况

前方有道路施工吗？是否需要绕道或改变路线？

交通情况

这条路的交通状况是否会让人们选择走另一条路？

汽车效率

我们可以调整汽车的设置，使其更有效地运行吗，例如更低的油耗或者更安全的排放？

汽车保养

我们的汽油快用完了吗？最近的加油站在哪里？需要保养设备吗？最近的维修点在哪里？

车队、汽车共享和使用管理

谁在使用哪辆车？我们是否有效地使用了车队的资源？我们能否优化车辆的使用，为客户提供更好的服务？想想用无人驾驶汽车来代替优步吧。

所有这些计算对于操纵汽车来说都很重要，但是不像前面的计算那样对时间敏感。这些计算可以（实际上也应该）在云中进行，它们通常需要访问集中式的数据，例如，地图和交通信息。在云端更容易访问这些信息。车队管理需要与其他车辆和其他集中式的系统进行协调。这些服务需要使用云计算中的功能，而这些功能在汽车本身中是不容易获得的。

但是更重要的是，这些计算不像边缘计算那样对时间敏感。确定如何避开交通拥堵，不如确定前面的人是否需要你转弯躲避来得及。因为对性能的需求是不同的，所以这些计算可以在云中进行。

云端的软件更容易开发、管理和操作，它们可以很容易地与其他
的云端软件协调，并且可以更好地利用集中式的数据和系统。

边缘计算中的软件可以更快、更可靠地响应紧急情况，它对特定
情况的反应和适应能力更强。此外，边缘计算软件可以提供更高级别
的数据安全性，因为它能够增加数据的本地化，减少对集中式数据的
依赖。

边缘计算的可伸缩与云计算的可伸缩不同

响应性和易于管理并不是边缘计算和云计算之间的唯一区别。在
边缘计算中构建可伸缩的软件，与在云中是不同的。边缘软件可以水
平伸缩，因为添加新用户意味着添加新实例。而云计算软件是垂直伸
缩的，因为添加新用户意味着现有软件必须处理更多的请求，并且必
须伸缩得更多。

边缘计算的伸缩是关于如何管理实例的。我如何升级这么多的软
件实例？如何监视所有这些实例的执行情况？云计算伸缩就是关于如
何管理资源的。我是否拥有所需的资源来满足软件运行所需的资源？

对于运行在边缘计算中的软件：

- 边缘计算软件通常会运行成千上万个实例。
- 边缘计算软件可以在大量地理位置不同的地区运行，通常在数百
万个地点中每个地点都有一个实例。
- 边缘计算软件的每个实例通常一次只做一件事、管理一个设备或
者执行一个操作。

对于运行在云端的软件：

- 云端软件运行的实例通常比边缘软件少得多。云端软件可以运行
在数十个、数百个或数千个服务器和实例上，但是实例的数量远
远低于边缘软件的实例数量。

- 云端软件通常在单个地点或者少数几个地点运行，并且在服务器集群中运行。
- 每个云端软件实例通常会管理数千个不同的任务，有时是同时管理，以便处理许多不同的用户需求。

这些差异导致了非常不同的伸缩需求。随着时间的推移，应用程序的总数会不断增加，每个系统处理伸缩需求的方式也不尽相同。

对于云端软件，必须随着并发用户数量的增加而扩展。软件使用得越多，必须运行的实例就越多。你必须将软件设计和构建为可以让更多的实例快速上线，以便满足更高的扩展需求，并且对资源的需求和分配也必须能够同时扩展，以免软件在高负载下崩溃。对于云端软件来说，负载是随着使用量的增加而增加的。

但是，对于边缘软件来说，每个边缘设备通常都只处理一个用户和一组请求。随着对软件需求的增加，需要使用更多的软件实例，但是每个实例都是独立自治的，并不知道其他实例的存在。因此，即使应用程序的用户数量不断增加，边缘软件的负载仍然是平稳的。边缘软件是意识不到需要扩展的。

然而，满足所有用户需求的实例数量，是根据边缘计算软件的用户数量线性增长的。如果你增加100万辆新的自动驾驶汽车，就增加了100万个软件的新实例。软件的每个实例仍然只管理一辆车，但是实例的数量增长很快。对于云端软件，所需的实例数量虽然也是根据用户的数量增长的，但增长的速度并不相同。如果云端应用程序增加100万个新用户，可能只需要添加数十台或数百台服务器，但肯定不是数百万台。

这一切意味着什么？对于云端软件来说，资源管理是关注的焦点。确保有足够的资源来满足软件运行的规模，这是你在扩展时最应关心的问题。对于边缘软件，实例管理是关注的焦点。如何管理、操作、升级和监控大量的软件实例，是扩展过程中最需要关注的问题。

使用边缘计算和云计算的标准

决定一个服务是在边缘运行还是在云端运行很重要，那么应该使用什么标准来决定某个服务或者应用程序应该运行在哪里呢？

以下是一套具体的建议标准，方便你用来做出决定。

标准	边缘还是云
计算是及时的，或者对延迟高度敏感	边缘
你需要高可响应的软件	边缘
需要大量的计算资源	云
计算的使用是难以预料的	云
对网络连接问题高度敏感	边缘
需要访问全局数据和较少的个性化数据（例如，交通情况）	云
所有其他情况	云

为什么在其他情况下，都要使用云计算而不是使用边缘计算呢？原因有很多，其中一些之前已经有所提示。但是具体来说，以下是云服务比边缘服务更受欢迎的几个可能原因：

- 边缘服务更加难以管理，也更加难以升级。
- 边缘服务有各种特殊的管理问题。你可能必须使用多个版本、不同功能的边缘硬件。
- 边缘服务存在软件版本的管理问题。很容易在不同的边缘设备上运行不同版本的服务软件。
- 边缘服务由于其高度分布式的特性，较难监控和管理。

边缘计算成功的8个关键点

我们已经看到，管理边缘软件可能更具有挑战性，尤其是在一个高度可伸缩的应用程序中。我们还看到了为什么使用边缘软件对应用程序至关重要。基于这一点，我们如何才能成功地在大规模应用程序中，有效地使用边缘计算呢？成功地在应用程序中构建边缘计算有8个关键点。它们很简单，但是对成功使用边缘计算都是非常有价值的建议。

#1：知道什么时候应该使用边缘计算

这是本章前面所讲内容的延伸。你必须主动决定是使用边缘计算还是云计算来满足你的计算和存储需求。

记住边缘计算有什么好处，云计算有什么好处。并且记住边缘计算相比云计算有什么缺点。当你不确定的时候，选择云计算。只有在对边缘最优的时候才使用边缘计算。

#2：不要忽略边缘计算中的DevOps原则

在考虑边缘计算时很容易忽略DevOps原则。你将经常听到诸如“边缘计算是高度定制化的计算”和“边缘计算需要新的处理过程和流程”之类的评论。这些都是常见的理解。

但是请你记住什么是DevOps。DevOps是关于：

- 所有权和责任
- 分布式决策
- 人员、流程和工具（最重要）

边缘计算中使用的流程可能会改变，你使用的工具也可能不同。但是仍然会有流程，仍然会有工具。涉及的人都是一样的。

DevOps也可以很好地应用于边缘计算中。

#3：制定一个高度分布式的部署策略

在构建应用程序的时候，我们通常不会充分考虑如何使用高度自动化和高度可靠的过程将应用程序部署到生产环境中。我们经常会说：“可以以后再解决这个问题”。尽管自动化和可重复的部署对所有应用程序都至关重要，但是它们对边缘应用程序的重要性更大。这是因为边缘应用程序的远程特点以及会涉及大量的节点。

如果没有一个可靠的、高度自动化的部署过程，你的边缘应用程序将会遇到故障和失败。

#4：尽可能减少版本控制

边缘部署是困难的，因此要减少边缘应用程序所需的部署数量。

尽量少部署。

这违背了DevOps和敏捷开发过程的传统要求。但是为什么在边缘计算中可以呢？

DevOps和敏捷开发过程使用了CI/CD（持续集成和持续部署）的原则。这些原则鼓励广泛使用更多和更小的部署。这个建议对于基于云和服务器的软件非常有用，但是对于边缘软件来说，版本控制就成了一个问题。自动化的升级过程是关键，并且会涉及大量的节点。边缘应用程序对部署流程的要求比云端应用程序大得多。因此，通过减少部署数量来减少版本控制的数量是有价值的。

然而，这里有一个微妙的界限。你可能会过多地减少部署次数，从而使每次升级的规模和复杂性大大增加，进而增加部署失败的风险。所以请谨慎使用这条建议。尽可能地减少版本控制，而不是过度减少。持续部署仍然是一种有用的策略，快速部署仍然有其价值。你只需在边缘计算环境中平衡该策略的有效性和成本的增加。

#5：减少每个节点的配置项

考虑到在大型边缘部署中会涉及大量的节点，你很难管理这些边缘设备软件，除非它们都运行相同的硬件和硬件版本。你也很难管理这些边缘设备的软件，除非它们都使用相同的软件配置项。节点集群中的硬件/软件配置越多样化，你就越难有效地管理所有这些节点。管理、监控和升级将变得更加困难。当有更多的配置项可以使用时，扩展边缘应用程序的所有方面都会变得更加复杂。

如果每个远程温度探测器都在相同的硬件上运行，则更容易构建和管理软件。如果你有20个不同版本的温度探测器硬件，或者不同厂

家生产的不同版本，管理所有这些差异就会变得极其复杂，不仅增加了边缘软件的运维难度，而且增加了遇到问题的可能性。

当然，减少配置和配置项并不总是可能的。一个最好的例子是移动端应用。移动端应用是必须在大量不同的硬件/软件配置上运行的边缘应用程序。这不是你的选择，而是客户的选择。拥有大量不同的软件和硬件配置对所有移动应用开发人员来说都是一个挑战。这个问题实际上证明了我的观点。你需要减少配置的数量来让软件管理变得更加容易。有时这是不可能的，但应当尽可能地去这么做。

#6：伸缩也是边缘计算要面对的一个问题，不仅仅是云计算

后端云服务的伸缩，是关于每个节点能够处理多少负载，以及处理这些负载所需的资源。边缘计算伸缩是指你可以处理多少个节点。

它们都是伸缩性方面的问题。

边缘节点的管理十分困难，理解并认识到边缘软件存在可伸缩性问题，以及如何管理它们，对于构建高可伸缩性、高可用性的应用程序非常重要。

#7：重视监测和分析

更多的节点和分布式节点，意味着了解每个节点在任何时间的执行情况是很重要的。但是如果没有良好的分析，这是很难做到的。边缘系统管理需要对高可伸缩系统中每个节点的健康状况进行持续监测。

此外，包含边缘节点健康状况分析的高级报告往往是发给组织的更高级领导的。在云计算中，单个服务器或数据中心的性能对一般公司的高层管理人员来说并不重要，但是在大多数企业中，了解有多少自动无人机工作良好，又有多少工作不佳，被认为具有更高的重要性。

#8：边缘计算不是魔法

边缘计算并不新鲜，也并不“特别”。我们做边缘计算已经很多年了，只是换了个名字称呼它。可能将它称为“浏览器应用程序”、“移动端应用程序”或者“销售点”设备。但是它们其实都是边缘计算。

边缘计算并不是一种新的计算方式。但是，边缘计算是一种对现有计算进行分类和标记的新方法。

这种新的分类和标记方法，对于边缘计算的未来是有益的和令人鼓舞的。这意味着将来会有更好的面向边缘计算的工具，将会有为边缘计算量身定制的服务。我们已经看到一些云服务提供商在做这样的事情，比如AWS已经提供面向边缘计算和面向IOT的服务。

但是现有的工具（不是专门用于边缘计算的工具）仍然适用于构建和管理边缘服务和应用程序。边缘计算不是魔法。

边缘计算小结

上面所述是成功构建边缘计算应用程序的8个关键点。总之，它们是一些简单但非常有价值的策略。

了解哪些类型的应用程序最好构建在边缘计算上，哪些应用程序最好构建在云计算上，这一点很重要。理解应用程序在边缘环境中和在云环境中对可伸缩性的影响是很重要的。

这一切都是关于如何理解和管理现代化应用程序及其组件的，无论它们是云计算还是边缘计算的组件。

第20章

地理位置对云计算的影响

这一章与本书的其他章节不同。这是我个人在全球旅行的经历中，对于不同文化如何影响不同地区接受和采用云计算的总结。

在我目前的工作中，我有幸在世界各地与许多公司交流云计算。在这些旅行中，我注意到云计算是如何被使用的，以及它如何影响公司的文化，而这些都跟所在的地理位置有关。这在很大程度上是由于国家和地区的文化差异影响了企业文化。如果你在一家跨国公司工作，并致力于推动云计算，那么你可能会发现这些差异非常有趣。

我特别注意到，在欧洲、美国、东南亚、澳大利亚和新西兰之间有着不同的发展趋势。根据我的旅行经历，我确定了地理多样性影响采用云计算的5个方面。

云无处不在，只不过影响的层次不同

对于任何行业和整个世界来说，都想利用各种方式来获得云计算所带来的好处。然而，采用云计算的级别和成熟度却因地域而异。我在第12章的“云计算成熟度的6个级别”一节中谈到了云计算的成熟度，但是重点在于企业文化。我在旅行中发现，企业在如何采用云计算方面存在地理位置上的差异。

例如，澳大利亚和新西兰的公司往往比世界其他地方的同行会更快地采用云计算技术。它们渴望获得信息，了解如何利用云计算来改善它们的业务。它们积极地研究早期的技术，希望了解到云计算将如何影响和帮助业务取得增长。[\[1\]](#)

相反，在德国和DACH注1的其他地区，情况恰恰相反。这一地区的公司更加注重安全，它们希望在进一步采用一项新的技术之前，充分了解它可能带来的影响。当面对一项新技术时，它们的反应通常是，“这会比我已经拥有的技术更好吗？它会导致什么问题？”。这是一种对待技术更为保守的方式。

这两种方法没有对错，都是使用云计算的独特方式。

替换心态影响你如何接受云计算

我注意到，世界上不同地区的技术架构，在修复和解决问题方面存在着显著差异。

为了理解这一点，我们需要回顾一些历史，从工业革命初期开始。历史上，在欧洲和北美洲，工业化国家需要的大多数产品都是在当地生产的，或者至少是在非常强大的贸易路线上很容易获得的。因此，人们很容易得到设备，当设备坏了，很容易得到设备的备件。这就产生了一种“替换心态”，在这种心态下，坏掉的东西很容易就能得到修复和替换，再也不会去想它们了。东西坏了，但是产生的问题很容易解决，而且是一个一劳永逸的解决办法。

在新西兰等较偏远的地区，情况并非如此。新西兰非常孤立，贸易路线又长又窄。从历史上看，产品运送到新西兰需要6个月的时间，因此很难找到替换部件。如果农场里的一台拖拉机坏了，等上6个月更换零件意味着你错过了整个种植季节。因此，公司和个人必须根据手头上有的东西，非常巧妙地找出修复损坏设备的办法。快速和临时的修复是常见的，设备被组装在一起刚好可以完成工作就行。

在新西兰，人们甚至有一个短语来表达这种心理。这就是“8号线心态”，它来自新西兰偏远地区常见的电线尺寸，通常用于安装临时的机械装置。“8号线心态”这个短语今天仍然被用来描述新西兰人利用手头任何的废旧材料来解决问题的方法。

这种与其他国家联系紧密的国家和联系不那么紧密的国家之间的操作方法差异，甚至影响了这些国家中企业的心态。在欧洲，人们倾向于使用可靠的、定义良好的流程和过程，并且使用经过良好测试和“批准”的方法来解决问题。这包括它们如何对待软件和云计算。在

美国，这种趋势更多的是一种替代心态。如果有东西坏了，就用好用的东西代替它，然后继续前进。不会闲坐着担心这件事。

然而，在新西兰和澳大利亚，情况就不同了。在这些国家中有一种倾向，就是用手头上的东西来解决问题，而且只要达到恰好能解决眼前问题的程度就行。他们更多地关心使用更新、更快、更巧妙的、可以优先完成工作的技术，而不是固有或者传统的流程。

这种心态上的差异影响了云服务提供商应该如何与这些不同地区的客户进行沟通。新西兰和澳大利亚急切地采用云技术和更新的云计算功能，因为这些新工具可以帮助它们解决问题。在美国，云计算是一种全新的、经过改进的业务处理方式，也被广泛采用。

然而，在欧洲，云计算被视为一种新的做事方式，它可能比当前的做事方式更好，也可能不会更好。它们通常采取的是观望态度，只有在看到云计算可以解决特定的、可测量的问题时才使用它。

哪个云服务提供商最重要

在美国，Amazon Web Services (AWS) 显然是最受欢迎的云服务提供商，但是近年来，微软Azure的使用也开始升温。在澳大利亚和新西兰也是如此，AWS在这两个国家似乎更加根深蒂固，而微软Azure还没有在市场思维模式方面取得重大进展。

然而，在欧洲和英国，虽然AWS也很受欢迎，但是微软Azure几乎存在于所有的对话中。这是与我交谈过的大多数人所偏爱的云计算技术。

在所有地区都有一些特定行业的公司存在“抵触AWS”的现象。这包括许多将亚马逊视为竞争对手的零售或者电子商务公司。这种敏感阻碍了这些公司使用AWS，它们转而关注Microsoft Azure和谷歌云平台（GCP）。这种影响在美国和欧洲最为强烈，在亚洲似乎不那么普遍。

GCP很少出现在企业客户讨论中，但是当它出现时，大多数出现在那些“抵触AWS”的公司中。IBM云计算也出现在欧洲，特别是在DACH地区，但是在其他地方很少见。

重要的技术区别

在每个地区都有一些不同的云计算技术，它们具有不同的重要性。这其中的一些差异包括：

私有云

虽然私有云在云计算的早期历史中是一个流行的时髦词语，但是它在世界上大多数地方已经变得不那么重要，也不那么普遍了，除了德国和DACH地区的其他国家。在这些国家，人们非常关注私有云。这主要是由于政府、企业和消费者对于安全和隐私的担忧，该地区的公司今天仍然认为公有云存在问题。这些问题限制了该地区的公司将业务迁移到公有云，因此将私有云功能放在自己的数据中心将成为下一个最佳选择。

容器化

虽然容器化在世界各地都很流行，但是在英国和整个欧洲大陆，人们对它的兴趣特别浓厚。在英国和欧洲部分地区，它被视为一种更容易利用微软和Linux技术的方式。

安全

云安全对每个人都很重要。然而，在我访问的大多数地方，这通常被认为是一个已知的以及可以解决的问题。然而，在德国，它被认为是阻碍采用公有云的主要问题。

DevOps和CI/CD

在美国、澳大利亚、新西兰、英国和荷兰，人们热衷于讨论这些话题。在DACH地区，对这些话题的讨论没有那么普遍，考虑到他们对云计算的看法，这并不奇怪。

数据主权是普遍的

云计算有一个非常重要的方面—数据主权。数据主权是希望将客户的数据保存在创建和使用数据的国家。

同时对以下方面有一种强烈的需要：

出于性能原因，将数据保存在本地

对于主动访问的数据，延迟是一个主要问题，特别是在亚太地区，该地区到其他国家的延迟通常更高。让数据在地理位置上更加接近，可以显著改善客户体验。这使得亚太地区的一个焦点是将数据存储到亚太地区。

在本地控制它们的数据

保持对所有数据的控制权也值得关注。新颁布的法律和即将出台的法律变更，要求企业关注数据的存储方式和存储地点。越来越多的人认为，在地区之外存储数据是安全和隐私问题。

我的看法

重要的是要理解使用云计算的文化，不仅是一家公司与另一家公司之间的差异，而且是一个国家与另一个国家之间的差异。在世界的不同地区，人们看待和利用云计算的方式存在着真实而显著的差异，了解这些差异对于在这些地理区域上工作的人来说非常重要。

[1] 达赫地区由德国、奥地利和瑞士组成。

第VI部分 总结

可伸缩的架构不只是用来处理大量的用户。

第21章 综述

在本书中，我们对许多主题进行了大量介绍，这些内容合在一起可以帮助你伸缩应用程序。我们关注以下5个原则：

- 原则1—可用性：维护现代化应用程序的可用性。
- 原则2—现代化应用程序架构：使用服务。
- 原则3—组织：为现代化应用程序建立具有可伸缩性的组织。
- 原则4—风险：现代化应用程序的风险管理。
- 原则5—云计算：利用云计算。

原则#1—可用性

可用性是应用程序执行其能够执行的任务的能力。这与可靠性不同，可靠性是指应用程序不出现错误的能力。一个计算“2+3”返回“6”的系统的可靠性很差，但是计算“2+3”从不返回结果的系统的可用性很差。可用性差是由许多原因造成的，包括以下几点：

- 资源枯竭
- 基于负载的意外变更
- 增加了组件的数量
- 外部的依赖

- 技术债务

当应用程序试图扩展到超出其能力范围时，它的可用性通常是第一个受害者。我们已经了解了可用性是什么，如何测试它，以及如何在高可伸缩的应用程序中使用工具来提高可用性，甚至在不断增加伸缩需求的情况下也可以做到。

原则#2—架构

服务是一个独特的封闭系统，它提供了业务功能来支持构建一个或多个大型产品。服务提供了一种应用程序架构模式，以一种提升系统和开发团队可伸缩性的方式来促进系统的构建。

在构建高可伸缩的应用程序时，服务需要做出更多的伸缩决策、适应更多的团队关注和控制、减少本地级别的复杂性，以及改进测试和部署功能的能力。

我们提供了一些工具和建议，帮助你构建服务级别的高可用性，并减少服务故障对应用程序及其用户的影响。

原则#3—组织

伸缩会影响你的组织，而不仅仅是你的应用程序。我们了解了由独立团队负责的服务架构模型（STOSA），它可以在伸缩应用程序规模的同时，伸缩开发团队组织，从而使更多的工程师能够有效地处理单个应用程序，而不会牺牲应用程序的可伸缩性或可用性。这涉及定义服务所有者的含义，以及围绕这些原则来组织应用程序。

我们讨论了如何使用工具来管理服务依赖关系，以便保持应用程序的质量，甚至在高速增长期间也是如此，其中包括内部SLA和服务级别。

原则#4—风险

如果你不能识别系统中的风险，就不可能管理系统中的风险。理解你的风险是运维一个高可用、高可伸缩的应用程序的第一步，也是最重要的一步。

在了解了风险之后，你必须管理风险。尽管我们总是想消除风险，但是这样做的成本通常是无法接受的，无论是从实际成本还是从机会成本的角度来看都是如此。你肯定有更重要、更加需要以客户为中心的事情要做，这些事情对你的客户、公司来说都有好处，而不是从应用程序中消除你所知道的每一个风险。

管理风险涉及对每个风险评估两个值：风险的可能性和风险的严重性。一般来说，严重性是风险发生时的成本，而可能性是风险发生的概率。一个不太可能发生但是会对应用程序造成非常严重影响的风险，不一定是你想要消除的风险。同样，一个很可能发生但是对应用程序影响很小的风险，也不一定是你想要优先消除的风险。但是，一个很可能会发生并且会导致严重影响的问题，是你需要优先解决的风险。

我们介绍了一个称为“风险矩阵”的工具，它可以非常有效地帮助你管理应用程序的风险，并且确定需要缓解或消除哪些风险。

我们讨论了降低风险的技术、验证降低风险行动计划的技术，以及一些构建降低风险的应用程序的技术。

原则#5—云计算

最后，我们讨论了云计算，以及如何使用它来构建高可伸缩的应用程序。

我们研究了云计算如何改变了我们对计算和构建应用程序的看法，讨论了如何使用云计算为应用程序构建地理和网络拓扑的多样性，以及如何避免这些方面出现的陷阱，实际上它可能具有内置的依赖性，从而增加了出现问题的风险。

我们讨论了托管基础设施，以及如何在大规模应用程序中使用它。我们讨论了如何分配基于云计算的资源，以及你需要在确保应用程序拥有足够云计算资源方面需要扮演的角色。

然后，我们还讨论了云计算中可以使用的计算方式。我们介绍了 AWS Lambda，以及可伸缩开发所带来的革命性未来。

面向可伸缩架构

为应用程序设计一个面向可伸缩的架构，比构建一个同时处理大量用户请求的应用程序要复杂得多。为了让应用程序具有可伸缩性，会涉及以下很多方面：

- 它必须处理大量且不断增长的客户、大量且不断增长的数据，以及客户想要用应用程序实现的功能也变得越来越复杂。
- 随着公司需求的增加，你需要增加更多的开发人员来开发应用程序，而且不能牺牲开发速度、效率和应用程序的质量。
- 你的应用程序必须时刻保持在线和提供功能，即使在面对所有上述的变化和改进时。

这些都不是容易解决的问题。本书中讨论的技术旨在帮助你解决这些问题，以及更多的应用程序可伸缩性问题。

为了纪念我的继女
Cherise Watts (1981—2019)

关于作者

Lee Atchison是New Relic云架构的高级总监。在过去的8年时间里，他帮助设计和建立了一个坚实的基于服务的产品架构，帮助公司从一个初创公司发展到高流量的大型公司。

Lee拥有33年的行业经验，他曾经在Amazon担任过7年高级经理。在Amazon，他带领团队创建了公司的第一个软件下载商店，创建了AWS Elastic Beanstalk产品，并负责将Amazon零售平台从一个单体架构迁移到基于服务的架构。

Lee曾为一些知名公司提供咨询，主要关于如何实现现代化的应用程序架构和进行大规模的组织转型，包括如何优化云计算平台和基于服务的架构、实践DevOps和高可用性设计。

Lee是一位行业专家，经常会发表文章，并且被诸如*InfoWorld*、*ComputerWorld*、*Diginomica*、*IT Brief*、*ProgrammableWeb*、*The New Stack*、*CIOReview*、*DevOps Digest*和*DZone*等媒体引用。无论从伦敦到悉尼，从东京到巴黎，还是在整个北美，他都是全球活动的重要演讲者。

关于封面

本书封面上的动物是一只纺织锥形蜗牛（也称织锦芋螺）。由于其贝壳上独特的黄棕色和白色特征，所以又被称为“金锥布”，它通常能够长到3到4英寸。纺织锥形蜗牛生活在红海的浅水中、澳大利亚和西非的海岸边，以及印度和太平洋的热带区域。

同其他芋螺类动物一样，纺织锥形蜗牛也是食肉动物，捕食其他蜗牛，通过从“齿舌”（一个类似小针一样的口器）注入毒素将猎物杀死。从其身上提取的“芋螺毒素”毒性非常强，能够导致麻痹或者死亡。

纺织锥形蜗牛一次可产下几百枚蛋，自由成长为成年蜗牛。虽然它们的壳有时被作为装饰物贩卖，但是它们的数量巨大，不属于濒危物种。O'Reilly书籍封面上的许多动物都是濒危动物，它们对于世界来说非常重要。

封面插图由Karen Montgomery绘制，基于*Wood's Illustrated Natural History*一书中的黑白版画。

O'REILLY®

可伸缩架构：云环境下的高可用与风险管理（第2版）

每一天，企业都面对着如何让关键应用程序可伸缩的问题。当流量和数据需求不断增加时，这些应用程序会变得更加复杂和脆弱，暴露出大量的风险并降低了可用性。随着“软件即服务”概念的流行，可伸缩性变得十分重要。

本书更新了重要的现代化架构范式，例如，微服务和云计算，通过实际案例的指导，让你了解如何在降低客户期望的条件下，构建可以处理海量请求、数据和需求的系统。工程和运维领域的架构师、经理和总监将学习到如何构建可以更加平稳和可靠地伸缩应用程序的知识，从而满足客户的不同需求。

- 你会了解可伸缩性如何影响服务的可用性，为什么它很重要，以及如何改进它。
- 你会深入了解如何保证一个现代化的基于服务的应用程序架构的高可用性，以及降低服务故障的影响。
- 你会了解什么是“由独立团队负责的服务架构”，它是一种可以让你的开发团队随应用程序一起伸缩的模型。
- 你会学习到如何理解、测量和降低系统的风险。
- 你会学习到如何使用云计算来构建高可伸缩的应用程序。

“不要拿你的生意做赌注，规模化的发展是一个不可避免的趋势。本书会告诉你如何切实可行地做到这一点。”

——Colin Bodell

Shopify Plus工程副总裁，
Amazon.com网站应用平台
前副总裁

“本书是为想要知道如何实现可伸缩系统的主管、经理和架构师准备的全面指南。”

——Ken Gavranovic

New Relic的EVP兼GM，
Interland公司（现在是web.com）
的CEO兼创始人

Lee Atchison是New Relic云架构的高级总监。在过去的8年时间里，他帮助设计和建立了一个坚实的基于服务的产品架构，帮助公司从一个初创公司发展到高流量的大型公司。Lee拥有33年的行业经验，包括曾经在Amazon担任过7年高级经理。

SYSTEM ADMINISTRATION

图书分类：系统管理

责任编辑：张春雨

封面设计：Karen Montgomery 张健



Broadview®
www.broadview.com.cn

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)



定价：79.00元