云原生基础设施

书栈(BookStack.CN)

目　录

致谢

关于本书

前言

介绍

第1章 什么是云原生基础设施？

第2章 何时采用云原生

第3章 云原生部署的演变

第4章 设计基础架构应用程序

第5章 开发基础架构应用程序

第6章 测试云原生基础设施

第7章 管理云原生应用程序

第8章 保护应用程序

第9章 实现云原生基础设施

附录A 网络弹性模式

附录B 锁定

附录C Box：案例研究

致谢

当前文档 《云原生基础设施》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-07-22。

云原生基础设施

《云原生基础设施》，作者*Justin Garrison*和Heptio资深布道师*Kris Nova*，本书是关于创建和管理基础设施，以适用于云原生应用全生命周期管理的模式和实践。

阅读完这本书后，您将会有如下收获：

- 理解为什么说云原生基础设施是高效运行云原生应用所必须的
- 根据准则来决定您的业务何时以及是否应该采用云原生
- 了解部署和管理基础架构和应用程序的模式
- 设计测试以证明您的基础架构可以按预期工作，即使在各种边缘情况下也是如此
- 了解如何以策略即代码的方式保护基础架构

前言

我还记得有一次告警呼叫让我半夜醒来，发现是生产环境离线了。我们的系统瘫痪了，我们不可避免地要赔钱，这是我的错。

从那一刻起，我就一直痴迷于构建坚如磐石的基础设施和基础设施管理系统，这样我就不会再重蹈覆辙了。在我的职业生涯中，我为Terraform、Kubernetes，一些编程语言和Kops做出过贡献，并创建了Kubicorn。我不仅见证了系统基础设施的发展，而且我也帮助完善了它。随着基础设施行业的发展，我们发现企业基础设施现在正以新的、令人兴奋的方式通过栈的应用层进行管理。到目前为止，Kubernetes是这种管理基础设施的新范例的最成熟的例子。

我与人合著了这本书，部分地介绍了将基础设施作为云原生软件的新范例。此外，我希望鼓励基础设施工程师开始编写云原生应用程序。在这本书中，我们探讨了管理基础设施的丰富历史，并为云原生技术的未来定义了管理基础设施的模式。我们解释了基础设施由软件化API驱动的重要性。我们还探索了创建复杂系统的第一个基础设施组件的引导问题，并教授了扩展和测试基础设施的重要性。

我于2017年加入Heptio，担任资深布道师，并且很高兴能与行业中最聪明的系统工程师密切合作。构建纯粹的开源技术对我来说一直都很重要，Heptio也拥有这种激情。我很荣幸能在这样一个环境中工作，让我更热爱这个行业。我希望你喜欢这本书，就像Justin和我喜欢写这本书一样。

——Kris Nova

介绍

基础设施技术的历史向来引人入胜。由于要求以巨大的规模经营，它已经经历了一次快速的颠覆性变革。除了计算机和互联网的早期，基础设施创新的步伐是无与伦比的。这些创新使基础架构更快，更可靠，更有价值。

有些公司的人将基础设施推到了极限，他们已经找到了自动化和抽象的方法，提取基础设施更多商业价值。通过提供灵活的可用资源，他们将曾经是昂贵的成本中心转变为所需的商业公用事业。

然而，公用事业公司很少为企业提供财务价值，这意味着基础设施往往被忽略并被视为不必要的成本。这使得投入创新或改进的时间和金钱很少。

这样一个如此简单且令人着迷的业务堆栈如何被轻易忽略？当基础设施出现故障时，业务显然会受到重视，那么为什么基础设施很难改善呢？

基础设施已经达到了使消费者都感到无聊的成熟度。然而，它的潜力和新挑战又激发了实施者和工程师们新的激情。

扩展基础设施并使用新的业务方式让来自不同行业的工程师都能找到解决方案。开源软件（OSS）和社区间的互相协作，这股力量又促使了创新的激增。

如果管理得当，今天基础设施和应用方面的挑战将会不一样。这使得基础设施建设者和维护人员可以取得进展并开展新的有意义的工作。

有些公司克服了诸如可扩展性、可靠性和灵活性等挑战。他们创建并封装了一些项目，这项项目中封装了可供他人遵循的模式。这些模式有时很容易被实现者发现，但在其他情况下，它们不太明显。

在本书中，我们将分享来自云原生技术前沿的公司的经验教训，使您能够有效解决可靠运行可伸缩应用程序的问题。现代商业发展非常迅速。本书中的模式将使您的基础设施能够跟上业务的速度和敏捷性需求。更重要的是，我们会让您自行决定何时采用这些模式。

这些模式中有很多都已经在开源项目中得到了体现。其中一些项目由云原生计算基金会（CNCF）维护。这些项目和基金会并不是模式的唯一体现，但忽视它们会让你失去理智。以它们为例，但要自己进行尽职调查，以审核您所采用的每个解决方案。

我们将向您展示云原生基础设施的益处以及可扩展系统和应用程序的基本模式。我们将向您展示如何测试您的基础设施，以及如何创建一个可以适应您需求的灵活的基础设施。您将了解哪些方面是重要的以及接下来会发生什么事情。

这本书可以激励你继续前进并自由分享你在社区中学到的知识。

谁应该读这本书

如果您是开发基础设施或基础设施管理工具的工程师，那么本书就是为您准备的。本书将帮助您了解创建旨在在云环境中运行的基础设施的模式、流程和实践。通过了解应该怎么做，您可以更好地了解应用程序的作用，以及应该何时构建基础设施或使用云服务。

应用程序工程师从本书中还可以发现哪些服务应该是其应用程序的一部分，哪些服务应该由基础设施提供。通过本书，他们还会发现他们应该与编写应用程序管理基础设施的工程师共同承担的责任。

希望提升技能并系统地在设计基础设施和维护云网关基础设施方面发挥更大作用的系统管理员也可以从本书中学到很

多。

你是否在公有云中运行所有的基础设施？本书将帮助您了解何时使用云服务以及何时构建自己的抽象或服务。

运行在数据中心还是本地云？我们将概述现代应用对基础设施的期望，并将帮助您了解利用当前投资的必要服务。

这本书不是一本教程，除了给出实现示例之外，我们没有指出特定的产品。对于经理、董事和高管来说，这可能太过技术性，但可能会有所帮助，具体取决于该角色的参与和技术专长。

最重要的是，如果您想了解基础设施如何影响业务，请阅读本书，以及如何创建经证实可为具备全球互联网规模运营的企业工作的基础设施。即使您的应用程序不需要扩展到这种规模，如果您的基础设施是使用此处描述的模式构建的，并且考虑到灵活性和可操作性，这本书仍然值得一读。

为什么我们写了这本书

我们希望通过专注于模式和实践而不是特定产品和供应商来帮助您了解。当前存在太多的解决方案而不了解它们本身到底要解决什么问题。

我们相信通过云原生应用程序管理云原生基础设施的好处，并且我们希望所有人都具有这种意识。

我们希望回馈社区，推动行业向前发展。我们发现这样做的最好方式是解释业务和基础设施之间的关系，阐明问题并解释发现它们的工程师和组织所做的解决方案。

以不涉及产品的方式解释模式并不总是很容易，但了解产品存在的原因很重要。我们经常使用产品作为模式的例子，但只有当它们会帮助您提供解决方案的实施示例时才会提到。

如果没有无数人数万小时的自愿编写代码，帮助他人以及投资社区，我们就不会走到这里。我们非常感谢那些帮助我们了解这些模式的人们，我们希望能够回馈并帮助下一代工程师。这本书就是我们表达谢意的方式。

浏览本书

本书的组织结构如下：

- 第1章介绍云原生基础设施是什么以及我们如何走到当前这一步。
- 第2章可以帮助您决定是否以及何时采用后面章节中预先描述的模式。
- 第3章和第4章展示了应该如何部署基础设施以及如何编写应用程序来管理它。
- 第5章将教你如何从测试开始就设计可靠的基础设施。
- 第6章和第7章展示了如何管理基础设施和应用程序。
- 第8章总结并提供了一些有关未来发展的见解。

如果你像我们一样，你不会从前到后完整看完本书。以下是关于本书主题的一些建议：

- 如果您是一位专注于创建和维护基础设施的工程师，您应该至少阅读第3章至第6章。
- 应用程序开发人员可以专注于第4、5和7章关于将基础架构工具开发为云原生应用程序。
- 所有不构建云原生基础设施的人都将从第1、2、8章中受益匪浅。

在线资源

您应该通过访问CNCF网站熟悉云原生计算基金会（CNCF）及其托管项目。本书中的许多项目都被用作示例。

您还可以通过查看CNCF景观项目（参见图P-1），了解项目这些项目的全局视图。

云原生应用程序是从Heroku的12因素的定义开始的。我们会解释它们之间的相似之处，但你应该熟悉下12因素是什么（参见http://12factor.net）。

还有许多关于DevOps的书籍、文章和演讲。尽管本书不关注DevOps实践，但是在实现云原生基础设施时，如果没有DevOps规定的工具、实践和文化，将很难实现。

图P-1. CNCF景观

致谢

Justin Garrison

感谢Beth、Logan、我的朋友、家人以及在此过程中支持我们的同事。感谢那些帮助我们的社区和社区领袖以及给予宝贵反馈的评论者。感谢Kris让这本书变得更好，感谢读者花点时间阅读本书并提高你的技能。

Kris Nova

感谢Allison、Bryan、Charlie、Justin、Kjersti、Meghann和Patrick为我写这本书所作出的帮助。我爱你们，永远感激你们为我所做的一切。

第1章 什么是云原生基础设施？

基础设施是指支持应用程序的所有软件和硬件。这包括数据中心、操作系统、部署流水线、配置管理以及支持应用程序生命周期所需的任何系统或软件。

已经有无数的时间和金钱花在了基础设施上。通过多年来不断的技术演化和实践提炼，一些公司已经能够运行大规模的基础设施和应用程序，并且拥有卓越的敏捷性。高效运行的基础设施可以使得迭代更快，缩短投向市场的时间，从而加速业务发展。

云原生基础设施是有效运行云原生应用程序的要求。如果没有正确的设计和实践来管理基础设施，即使是最好的云原生应用程序也会浪费。巨大的规模并不是遵循本书列出的实践的先决条件，但如果你想获得云计算的回报，你应该听从那些开创了这些模式的人的经验。

在我们探索如何构建用于在云中运行应用程序的基础设施之前，我们需要了解我们是如何走到这一步。首先，我们将讨论采用云原生实践的益处。接下来，我们将看一下基础设施的简要历史，然后讨论下一阶段的功能，称为“云原生”，以及它与你的应用程序，它运行的平台及业务之间的关系。

一旦您明白了这个问题，我们将向您展示解决方案以及如何实现它。

云原生优势

采用这本书中的模式的好处有很多。它们仿照谷歌、Netflix和亚马逊这些成功的公司—不是单靠模式保证它们的成功，而是它们提供了这些公司成功所需的可扩展性和敏捷性。

通过选择在公有云中运行基础设施，您可以更快地创造价值并专注于您的业务目标。只需构建您创建产品所需的内容，并从其他提供商那里获得服务，就可以缩短交付时间，提高灵活性。有些人可能因为“供应商锁定”而犹豫不决，但最糟糕的锁定是你自己建立的锁定。有关不同类型的锁定，以及您应该如何处理的更多信息，请参阅附录B。

消费服务还可让您使用您所需的服务构建定制平台（有时称为平台服务[SaaP]）。当您使用云托管的服务时，您无需精专于操作运行应用程序所需要的每项服务。这极大地影响了您改变业务并为您的业务增值的能力。

当您无法使用服务时，您应该构建应用程序来管理基础设施。当你这样做时，规模瓶颈不再取决于每个运维工程师可以管理多少个服务器。相反，您可以像扩展应用程序一样来扩展您的基础设施。换句话说，如果您能够运行可扩展的应用程序，则可以使用应用程序扩展您的基础设施。

同样的好处适用于构建灵活且易于调试的基础设施。您可以使用与管理业务应用程序相同的工具来洞察您的基础设施。

云原生实践还可以缩小传统工程角色之间的差距（DevOps的共同目标）。系统工程师将能够从应用程序中学习最佳实践，并且应用开发工程师可以拥有应用程序运行所在的基础设施的所有权。

云原生基础设施的解决方案不一定适用于所有问题，您有责任了解它是否适合您的环境（参见第2章）。然而，它的成功在创造这些实践的公司以及许多采用推广这些模式的工具的公司中显而易见。请参见附录C的一个例子。

在我们深入了解解决方案之前，我们需要了解这些模式如何从创建它们的问题演变而来。

服务器

在互联网初始阶段，Web基础设施开始于物理服务器。服务器庞大，吵闹且昂贵，而且他们需要大量的电力和人员来保持它们的运行。他们受到广泛照顾，并尽可能保持长时间运行。 与云基础设施相比，购买这些设备并让应用程序运行在上面会更困难。

一旦你买了一个，它就是你的了，无论好坏，都要维护。使用物理服务器适合已确定成本的业务。持有物理服务器并运行的时间越长，您花费的钱越多。做适当的产能规划并确保您获得最佳的投资回报总是很重要的。

物理服务器非常棒，因为它们功能强大，可以根据需要进行配置。它们的故障率相对较低，并且使用冗余电源供应，风扇和RAID控制器来避免出现故障。他们也可以持续运行很长时间。企业可以通过延长保修和更换零部件，从购买的硬件中挤出额外的价值。

但是，物理服务器会导致浪费。服务器不仅没有被充分利用，而且还带来了很多开销。在同一台服务器上运行多个应用程序是很困难的。当服务器最大限度地用于多个应用程序时，软件冲突，网络路由和用户访问都变得更加复杂。

硬件虚拟化承诺可以解决其中的一些问题。

虚拟化

虚拟化使用软件来模拟物理服务器的硬件。虚拟服务器可以按需创建，完全可以通过软件编程，只要您可以模拟硬件，永远不会磨损。

使用hypervisor可以增加这些优势，因为您可以在物理服务器上运行多个虚拟机（VM）。它还允许应用程序可移植，因为您可以将虚拟机从一台物理服务器移动到另一台物理服务器。

然而，运行自己的虚拟化平台的一个问题是虚拟机仍然需要硬件来运行。公司仍然需要拥有运行物理服务器所需的所有人员和流程，但是现在容量规划变得更加困难，因为他们也必须考虑到虚拟机开销。至少，公有云出现之前就是如此。

基础设施即服务

基础设施即服务（IaaS）是云提供商的众多产品之一。它提供了原始的网络、存储和计算能力，客户可以根据需要使用它们。它还包括一些支持服务，如身份和访问管理（IAM）、供应和库存系统。

IaaS允许公司摆脱他们的所有硬件，并从别人那里租用虚拟机或物理服务器。这释放了大量人力资源，摆脱了购买、维护以及在某些情况下容量规划所需的流程。

IaaS从根本上改变了基础设施与业务的关系。不是随着时间的推移受益的资本支出，而是运营业务的运营支出。企业可以像支付电力和人们的时间一样支付基础设施。通过基于消费的计费，您越早摆脱基础设施，运营成本就越低。

托管的基础设施还为客户提供了可消费的HTTP应用编程接口（API），以便按需创建和管理基础设施。工程师不需要购买订单并等待物品出货，就可以进行API调用，并创建服务器。服务器可以轻松删除和丢弃。

在云中运行基础设施不会使您的基础架构成为云原生。IaaS仍然需要基础设施管理。在购买和管理物理资源之外，您可以（也有许多公司）认为IaaS与他们过去购买并在自己的数据中心架设的传统基础设施一模一样。

即使没有“货架和堆叠”，仍然有大量的操作系统、监控软件和支持工具。自动化工具帮助减少了运行应用程序所需的时间，但通常根深蒂固的流程会影响IaaS的全部优势。

平台即服务

就像IaaS对VM消费者隐藏了物理服务器一样，平台即服务（PaaS）也对应用程序隐藏了操作系统。开发人员编写应用程序代码并定义应用程序的依赖关系，平台负责创建运行，管理和暴露它所必要的基础设施。与需要基础设施管理的IaaS不同，PaaS中的基础设施由平台提供商管理。

事实证明，PaaS限制要求开发人员以不同的方式编写应用程序，以便平台可以有效管理。应用程序必须包含允许它们由平台管理而不访问底层操作系统的功能。工程师不能再依赖SSH登入到服务器来读取磁盘上的日志文件。现在应用程序的生命周期和管理由PaaS控制，工程师和应用程序需要进行适应。

这些限制带来了很大的好处。应用程序开发周期变短了，因为工程师不需要花时间管理基础设施。在平台上运行的应用程序是我们现在称为“云原生应用程序”的开始。他们利用代码中的平台限制，并且在许多情况下已经改变了今天编写应用程序的方式。

> 12因素应用程序
>
> Heroku是提供公共消费型PaaS的早期先驱之一。通过自己平台的多年扩展，该公司能够确定帮助应用程序在其环境中更好运行的模式。Heroku定义了开发人员应该尝试实现的12个主要因素。
>
> 这12个因素是通过将代码逻辑与数据分离来使开发人员高效，尽可能自动化，独立的构建、传输和运行阶段过程；并声明所有的应用程序的依赖关系。

如果您通过PaaS提供商使用所有基础设施，恭喜，您已拥有云原生基础设施的诸多优势。这包括Google App Engine、AWS Lambda和Azure Cloud Services等平台。任何成功的云原生基础设施都将向应用工程师展示自助服务平台，以部署和管理他们的代码。

但是，许多PaaS平台不足以满足业务需求。他们通常会限制语言运行平台、库和功能以实现从应用程序中抽离基础架构的承诺。公有PaaS提供商还将限制哪些服务可以与应用程序集成以及这些应用程序可以在哪里运行。

公有平台交易应用程序的灵活性，使基础架构成为别人的问题。图1-1是如果您运行自己的数据中心，在IaaS中创建基础设施，在PaaS上运行应用程序或通过软件即服务（SaaS）运行应用程序时需要管理的组件的直观表示。

您需要运行的基础设施组件越少越好；但是在公有PaaS提供商中运行所有应用程序可能不是一种选择。

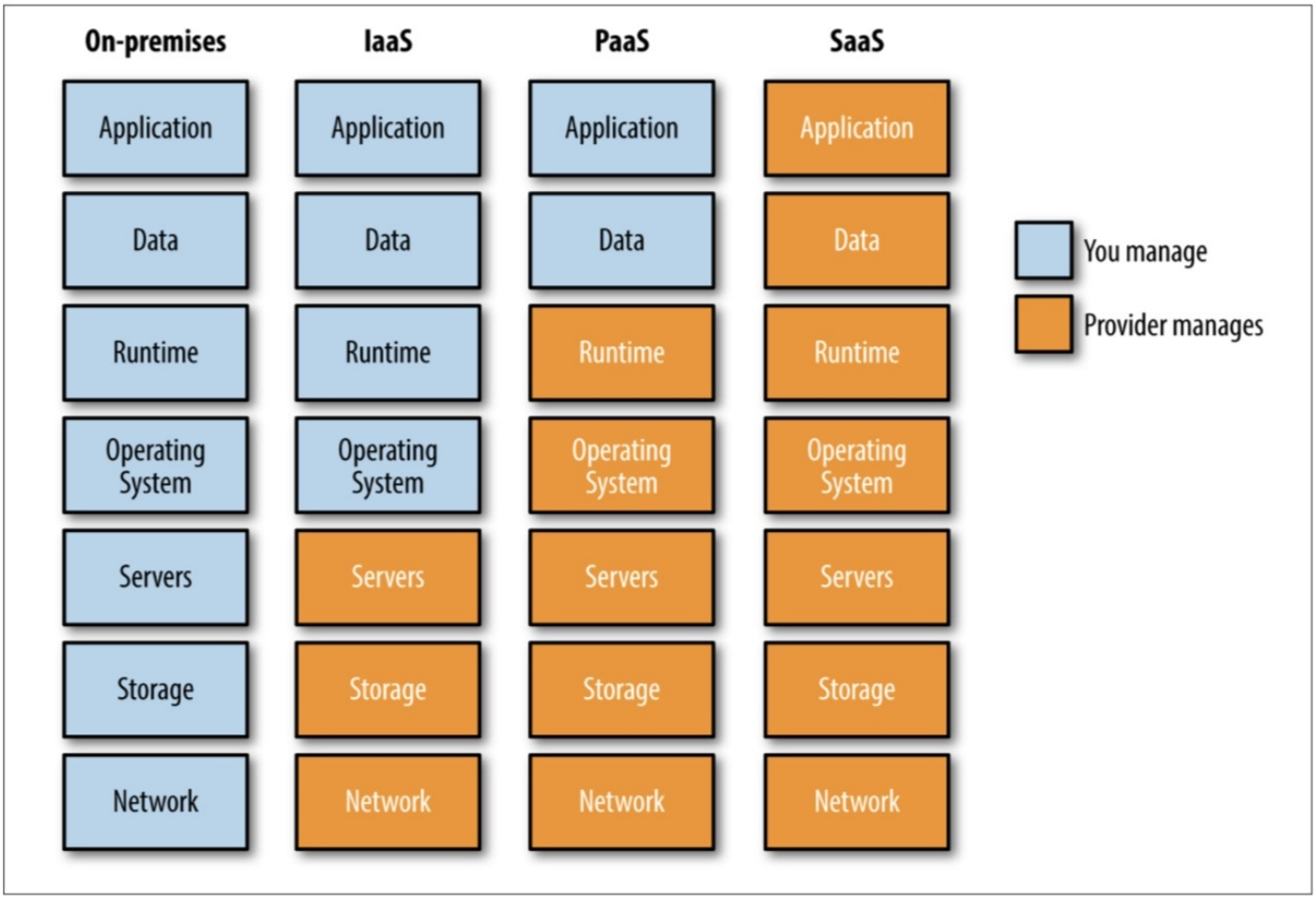

图1-1. 基础设施层

云原生基础设施

“云原生”是一个过度使用的术语。尽管它已被市场所劫持，但是对工程和管理仍然有意义。对我们来说，它意味着公有云提供商存在的世界中技术的变革。

云原生基础设施是隐藏在有用的抽象背后的基础设施，由API控制，由软件管理并具有运行应用程序的目的。利用这些特征运行基础设施，能以可扩展高效的方式管理该基础设施。

当它们成功地向消费者隐藏复杂性时，抽象是有用的。它们可以实现技术的更复杂的使用，但是它们也限制了技术的使用方式。它们适用于低级技术，例如TCP如何提取IP或更高级别的技术，如虚拟机如何抽象物理服务器。抽象应该总是允许消费者“向上移动堆栈”而不是重新实现底层。

云原生基础设施需要抽象基础IaaS产品以提供自己的抽象。新层负责控制它下面的IaaS，并将自己的API暴露给消费者控制。

由软件管理的基础设施是云中的一个关键区别点。软件控制的基础设施使基础设施能够扩展，并且在弹性，供应和可维护性方面也发挥着重要作用。软件需要了解基础设施的抽象概念，并知道如何获取抽象资源并相应地在可消费的IaaS组件中实现它。

这些模式不仅影响基础设施的运行方式，而且在云原生基础设施上运行的应用程序类型、在其上工作的人员类型与传统基础架构中是不同的。

如果云原生基础设施看起来很像PaaS产品，那么我们如何才能知道在构建自己的产品时需要注意什么？让我们快速描述一些可能看起来像解决方案的领域，但不提供云原生基础设施的所有方面。

什么不是云原生基础设施？

云原生基础设施不仅是在公有云上运行基础设施。仅仅因为你从其他人那里租用服务器时间并不会使您的基础设施云原生化。管理IaaS的流程通常与运行物理数据中心没有什么不同，许多将现有基础架构迁移到云的公司都未能获得回报。

云原生不是关于在容器中运行应用程序。当Netflix率先推出云原生基础设施时，几乎所有应用程序都部署了虚拟机映像，而不是容器。打包应用程序的方式并不意味着您将拥有自治系统的可扩展性和优势。即使您的应用程序是通过持续集成和持续交付渠道自动构建和部署的，这并不意味着您可以从可以补充API驱动部署的基础设施中受益。

这也并不意味着你只能运行容器编排器（例如Kubernetes和Mesos）。容器编排器提供了云原生基础设施所需的许多平台功能，但并未按预期方式使用这些功能，这意味着您的应用程序会被动态调度以在一组服务器上运行。这是一个非常好的起步，但仍有工作要做。

> 调度器与编排器
>
> 术语“调度器”和“编排器”通常可以互换使用。
>
> 在大多数情况下，编排器负责集群中的所有资源利用（例如：存储，网络和CPU）。该术语典型地用于描述执行许多任务的产品，如健康检查和云自动化。

调度器是编排平台的一个子集，仅负责选择运行在每台服务器上的进程和服务。

云原生不是微服务或基础设施即代码。微服务意味着更快的开发周期和更小的独特功能，但是单片应用程序可以具有相同的功能，使其能够通过软件有效管理，并且还可以从云原生基础设施中受益。

基础设施即代码以机器可解析语言或领域特定语言（DSL）定义、自动化您的基础设施。将代码应用于基础架构的传统工具包括配置管理工具（例如Chef和Puppet）。这些工具在自动执行任务和提供一致性方面有很大帮助，但是它们在提供必要的抽象来描述超出单个服务器的基础设施方面存在缺陷。

配置管理工具一次自动化一台服务器，并依靠人员将服务器提供的功能绑定在一起。这将人类定位为基础设施规模的潜在瓶颈。这些工具也不会使构建完整系统所需的云基础设施（例如存储和网络）的额外部分自动化。

尽管配置管理工具为操作系统的资源（例如软件包管理器）提供了一些抽象，但它们并没有抽象出足够的底层操作系统来轻松管理它。如果一位工程师想要管理系统中的每个软件包和文件，这将是一个非常艰苦的过程，并且对于每个配置变体都是独一无二的。同样，定义不存在或不正确的资源的配置管理仅消耗系统资源并且不能提供任何价值。

虽然配置管理工具可以帮助自动化部分基础设施，但它们无法更好地管理应用程序。我们将在后面的章节中通过查看部署，管理，测试和操作基础架构的流程，探讨云原生基础设施的不同之处，但首先，我们将了解哪些应用程序是成功的以及应该何时与原生基础设施一起使用。

云原生应用程序

就像云改变了业务和基础设施之间的关系一样，云原生应用程序也改变了应用程序和基础设施之间的关系。我们需要了解与传统应用程序相比，云本身有什么不同，因此我们需要了解它们与基础设施的新关系。

为了写好本书，也为了有一个共享词汇表，我们需要定义“云原生应用程序”是什么意思。云原生与12因素应用程序不

同，即使它们可能共享一些类似的特征。如果你想了解更多细节，请阅读Kevin Hoffman撰写的“超越12因素应用程序”（O’Reilly，2012）。

云原生应用程序被设计为在平台上运行，并设计用于弹性，敏捷性，可操作性和可观察性。弹性包含失败而不是试图阻止它们;它利用了在平台上运行的动态特性。敏捷性允许快速部署和快速迭代。可操作性从应用程序内部控制应用程序生命周期，而不是依赖外部进程和监视器。可观察性提供信息来回答有关应用程序状态的问题。

> 云原生定义
>
> 云原生应用程序的定义仍在发展中。还有像CNCF这样的组织可以提供其他的定义。

云原生应用程序通过各种方法获取这些特征。它通常取决于应用程序的运行位置以及企业流程和文化。以下是实现云原生应用程序所需特性的常用方法：

- 微服务
- 健康报告
- 遥测数据
- 弹性
- 声明式的，而不是反应式的

微服务

作为单个实体进行管理和部署的应用程序通常称为单体应用。最初开发应用程序时，单体有很多好处。它们更易于理解，并允许您在不影响其他服务的情况下更改主要功能。

随着应用程序复杂性的增长，单体应用的益处逐渐减少。它们变得更难理解，而且失去了敏捷性，因为工程师很难推断和修改代码。

对付复杂性的最好方法之一是将明确定义的功能分成更小的服务，并让每个服务独立迭代。这增加了应用程序的灵活性，允许根据需要更轻松地更改部分应用程序。每个微服务可以由单独的团队进行管理，使用适当的语言编写，并根据需要进行独立扩缩容。

只要每项服务都遵守强有力的合约，应用程序就可以快速改进和改变。当然，转向微服务架构还有许多其他的考虑因素。其中最不重要的是弹性通信，我们在附录A中有讨论。

我们无法考虑转向微服务的所有考虑因素。拥有微服务并不意味着您拥有云原生基础设施。如果您想阅读更多，我们推荐Sam Newman的Building Microservices（O’Reilly，2015）。虽然微服务是实现您的应用程序灵活性的一种方式，但正如我们之前所说的，它们不是云原生应用程序的必需条件。

> 健康报告
>
> 停止逆向工程应用程序并开始从内部进行监控。 - Kelsey Hightower, Monitorama PDX 2016: healthz

没有人比开发人员更了解应用程序需要什么才能以健康的状态运行。很长一段时间，基础设施管理员都试图从他们负责运行的应用程序中找出“健康”该怎么定义。如果不实际了解应用程序的健康状况，他们尝试在应用程序不健康时进行监控并发出警报，这往往是脆弱和不完整的。

为了提高云原生应用程序的可操作性，应用程序应该暴露健康检查。开发人员可以将其实施为命令或过程信号，以便应用程序在执行自我检查之后响应，或者更常见的是：通过应用程序提供Web服务，返回HTTP状态码来检查健康状态。

> **Google Borg示例**
>
> Google的Borg报告中列出了一个健康报告的例子：
>
> 几乎每个在Borg下运行的任务都包含一个内置的HTTP服务器，该服务器发布有关任务运行状况和数千个性能指标（如RPC延迟）的信息。Borg会监控运行状况检查URL并重新启动不及时响应或返回HTTP错误代码的任务。其他数据由监控工具跟踪，用于仪表板和服务级别目标（SLO）违规警报。

将健康责任转移到应用程序中使应用程序更容易管理和自动化。应用程序应该知道它是否正常运行以及它依赖于什么（例如，访问数据库）来提供业务价值。这意味着开发人员需要与产品经理合作来定义应用服务的业务功能并相应地编写测试。

提供健康检查的应用程序示例包括Zookeeper的ruok命令和etcd的HTTP / 健康端点。

应用程序不仅仅有健康或不健康的状态。它们将经历一个启动和关闭过程，在这个过程中它们应该通过健康检查，报告它们的状态。如果应用程序可以让平台准确了解它所处的状态，平台将更容易知道如何操作它。

一个很好的例子就是当平台需要知道应用程序何时可以接收流量。在应用程序启动时，如果它不能正确处理流量，它就应该表现为未准备好。此额外状态将防止应用程序过早终止，因为如果运行状况检查失败，平台可能会认为应用程序不健康，并且会反复停止或重新启动它。

应用程序健康只是能够自动化应用程序生命周期的一部分。除了知道应用程序是否健康之外，您还需要知道应用程序是否正在进行哪些工作。这些信息来自遥测数据。

遥测数据

遥测数据是进行决策所需的信息。确实，遥测数据可能与健康报告重叠，但它们有不同的用途。健康报告通知我们应用程序生命周期状态，而遥测数据通知我们应用程序业务目标。

您测量的指标有时称为服务级指标（SLI）或关键性能指标（KPI）。这些是特定于应用程序的数据，可以确保应用程序的性能处于服务级别目标（SLO）内。如果您需要更多关于这些术语的信息以及它们与您的应用程序、业务需求的关系，我们推荐你阅读来自Site Reliability Engineering（O’Reilly）的第4章。

遥测和度量标准用于解决以下问题：

- 应用程序每分钟收到多少请求？
- 有没有错误？
- 什么是应用程序延迟？
- 订购需要多长时间？

通常会将数据刮取或推送到时间序列数据库（例如Prometheus或InfluxDB）进行聚合。遥测数据的唯一要求是它将被收集数据的系统格式化。

至少，可能最好实施度量标准的RED方法，该方法收集应用程序的速率，错误和执行时间。

请求率

收到了多少个请求

错误

应用程序有多少错误

时间

多久才能收到回复

遥测数据应该用于提醒而非健康监测。在动态的、自我修复的环境中，我们更少关注单个应用程序实例的生命周期，更多关注关于整体应用程序SLO的内容。健康报告对于自动应用程序管理仍然很重要，但不应该用于页面工程师。

如果1个实例或50个应用程序不健康，只要满足应用程序的业务需求，我们可能不会收到警报。度量标准可让您知道您是否符合您的SLO，应用程序的使用方式以及对于您的应用程序来说什么是“正常”。警报有助于您将系统恢复到已知的良好状态。

> 如果它移动，我们跟踪它。有时候我们会画出一些尚未移动的图形，以防万一它决定为它运行。
>
> —Ian Malpass，衡量所有，衡量一切

警报也不应该与日志记录混淆。记录用于调试，开发和观察模式。它暴露了应用程序的内部功能。度量有时可以从日志（例如错误率）计算，但需要额外的聚合服务（例如ElasticSearch）和处理。

弹性

一旦你有遥测和监测数据，你需要确保你的应用程序对故障有适应能力。弹性是基础设施的责任，但云原生应用程序也需要承担部分工作。

基础设施被设计为抵制失败。硬件用于需要多个硬盘驱动器，电源以及全天候监控和部件更换以保持应用程序可用。使用云原生应用程序，应用程序有责任接受失败而不是避免失败。

> 在任何平台上，尤其是在云中，最重要的特性是其可靠性。
>
> —David Rensin，e ARCHITECT Show：来自Google的关于云计算的速成课程

设计具有弹性的应用程序可能是整本书本身。我们将在云原生应用程序中考虑弹性的两个主要方面：为失败设计和优雅降级。

为失败设计

唯一永远不会失败的系统是那些让你活着的系统（例如心脏植入物和刹车系统）。如果您的服务永远不会停止运行，您需要花费太多时间设计它们来抵制故障，并且没有足够的时间增加业务价值。您的SLO确定服务需要多长时间。您花费在工程设计上超出SLO的正常运行时间的任何资源都将被浪费掉。

您应该为每项服务测量两个值，即平均无故障时间（MTBF）和平均恢复时间（MTTR）。监控和指标可以让您检测您是否符合您的SLO，但运行应用程序的平台是保持高MTBF和低MTTR的关键。

在任何复杂的系统中，都会有失败。您可以管理硬件中的某些故障（例如，RAID和冗余电源），以及某些基础设施中的故障（例如负载平衡器）。但是因为应用程序知道他们什么时候健康，所以他们也应该尽可能地管理自己的失败。

设计一个以失败期望为目标的应用程序将比假定可用性的应用程序更具防御性。当故障不可避免时，将会有额外的检查，故障模式和日志内置到应用程序中。

知道应用程序可能失败的每种方式是不可能的。假设任何事情都可能并且可能会失败，这是一种云原生应用程序的模式。

您的应用程序的最佳状态是健康状态。第二好的状态是失败状态。其他一切都是非二进制的，难以监控和排除故障。Honeycomb首席执行官CharityMajors在她的文章“Ops：现在每个人都在工作”中指出：“分布式系统永远不会起作用;它们处于部分退化服务的持续状态。接受失败，设计弹性，保护和缩小关键路径。“

无论发生什么故障，云原生应用程序都应该是可适应的。他们期望失败，所以他们在检测到时进行调整。

有些故障不能也不应该被设计到应用程序中（例如，网络分区和可用区故障）。该平台应自主处理未集成到应用程序中的故障域。

优雅降级

云原生应用程序需要有一种方法来处理过载，无论它是应用程序还是负载下的相关服务。处理负载的一种方式是优雅降级。 “站点可靠性工程”一书中描述了应用程序的优雅降级，因为它提供的响应在负载过重的情况下“不如正常响应准确或含有较少数据的响应，但计算更容易”。

减少应用程序负载的某些方面由基础设施处理。智能负载平衡和动态扩展可以提供帮助，但是在某些时候，您的应用程序可能承受的负载比它可以处理的负载更多。云原生应用程序需要知道这种必然性并作出相应的反应。

优雅降级的重点是允许应用程序始终返回请求的答案。如果应用程序没有足够的本地计算资源，并且依赖服务没有及时返回信息，则这是正确的。依赖于一个或多个其他服务的服务应该可用于应答请求，即使依赖于服务不是。当服务退化时，返回部分答案或使用本地缓存中的旧信息进行答案是可能的解决方案。

尽管优雅的降级和失败处理都应该在应用程序中实现，但平台的多个层面应该提供帮助。如果采用微服务，则网络基础设施成为需要在提供应用弹性方面发挥积极作用的关键组件。有关构建弹性网络层的更多信息，请参阅附录A.

可用性数学

云原生应用程序需要在基础设施之上建立一个平台，以使基础设施更具弹性。如果您希望将现有应用程序“提升并转移”到云中，则应检查云提供商的服务级别协议（SLA），并考虑在使用多个服务时会发生什么情况。

让我们拿运行我们的应用程序的云来进行假设。

计算基础设施的典型可用性是每月99.95%的正常运行时间。这意味着您的实例每天可能会缩短到43.2秒，并且仍在您的云服务提供商的SLA中。

另外，实例的本地存储（例如EBS卷）也具有99.95%的可用性正常运行时间。如果幸运的话，他们都会同时出现故障，但最糟糕的情况是他们可能会在不同的时间停机，让您的实例只有99.9%的可用性。

您的应用程序可能还需要一个数据库，而不是自己安装一个计算可能的停机时间为1分26秒（99.9%可用性）的情况下，选择可靠性为99.95%的更可靠的托管数据库。这使您的应用程序的可靠性达到99.85%，或者每天可能发生2分钟和9秒的宕机时间。

将可用性乘到一起可以快速了解为什么应以不同方式处理云。真正不好的部分是，如果云提供商不符合其SLA，它将退还其账单中一定比例的退款。

虽然您不必为停机支付费用，但我们并不知道世界上存在云计算信用的单一业务。如果您的应用程序的可用性不足以超过您收到的信用额度，那么您应该真正考虑是否应该运行这个应用程序。

声明式，非反应式

由于云原生应用程序设计为在云环境中运行，因此它们与基础设施和支持应用程序的交互方式与传统应用程序不同。在云原生应用程序中，与任何事物进行通信的方式都是通过网络进行的。很多时候，网络通信都是通过RESTful HTTP调用完成的，但它也可以通过其他接口（如远程过程调用（RPC））来实现。

传统的应用程序会通过消息队列，写在共享存储上的文件或触发shell命令的本地脚本来自动执行任务。通信方法对发生的事件作出反应（例如，如果用户单击提交，运行提交脚本）并且通常需要存在于同一物理或虚拟服务器上的信息。

> 无服务器
>
> 无服务器平台是云原生化的，并通过设计对事件做出响应。他们在云中工作得很好的原因是他们通过HTTP API进行通信，是单用途功能，并且在他们所称的功能中声明。该平台还可以通过在云中进行扩展和访问来提供帮助。

传统应用程序中的反应性通信常常是尝试增强弹性。如果应用程序在磁盘或消息队列中写入文件，然后应用程序死亡，则消息或文件的结果仍可能完成。

这并不是说不应该使用像消息队列这样的技术，而是说它们不能被依赖于动态和不断发生故障的系统中的唯一弹性层。从根本上讲，应用程序之间的通信应该在云原生环境中改变 - 不仅因为还有其他方法来构建通信弹性（请参阅附录A），还因为在云中复制传统通信方法往往需要更多工作。

当应用程序可以信任通信的弹性时，他们应该停止反应并开始声明。声明式沟通相信网络将传递消息。它也相信应用程序将返回成功或错误。这并不是说应用程序监视变化并不重要。 Kubernetes的控制器正是这样做到API服务器。但是，一旦发现变更，他们就会声明一个新的状态，并相信API服务器和kubelets会做必要的事情。

声明式通信模型由于多种原因而变得更加健壮。最重要的是，它规范了通信模型，并且它将功能实现从应用程序转移到远程API或服务端点，从而实现某种状态到达期望状态。这有助于简化应用程序，并使它们彼此的行为更具可预测性。

云原生应用程序如何影响基础设施？

希望你可以知道云原生应用程序与传统应用程序不同。云原生应用程序不能直接在PaaS上运行或与服务器的操作系统紧密耦合。它们期望在一个拥有大多数自治系统的动态环境中运行。

云原生基础设施在提供自主应用管理的IaaS之上创建了一个平台。该平台建立在动态创建的基础设施之上，以抽象出单个服务器并促进动态资源分配调度。

自动化与自治不一样。自动化使人类对他们所采取的行动产生更大的影响。

云原生是关于不需要人类做出决定的自治系统。它仍然使用自动化，但只有在决定了所需的操作之后。只有在系统不能自动确定正确的事情时才应该通知人。

具有这些特征的应用程序需要一个能够实际监控，收集度量标准并在发生故障时做出反应的平台。云原生应用程序不依赖于人员设置ping检查或创建Syslog规则。他们需要从选择基本操作系统或软件包管理器的过程中提取自助服务资源，并依靠服务发现和强大的网络通信来提供丰富的功能体验。

结论

运行云原生应用程序所需的基础设施与传统应用程序不同。基础设施用于处理的许多责任已经转移到应用程序中。

云原生应用程序通过分解为更小的服务来简化其代码复杂性。这些服务提供直接构建到应用程序中的监控，指标和弹

性。需要新的工具来自动管理服务激增和生命周期管理。

现在基础设施负责整体资源管理、动态协调、服务发现等等。它需要提供一个平台，服务不依赖于单个组件，而是依赖于API和自治系统。第2章将更详细地讨论云原生基础设施功能。

第2章 何时采用云原生

云原生基础设施并不适合所有人。任何架构设计都是一系列的权衡。只有熟悉自己需求的人才能决定哪些权衡是有益的，哪些是有害的。

不要在不了解其影响和限制的情况下采用工具或设计。我们相信云原生基础设施有很多好处，但需要意识到它不应该被盲目的采用。我们不愿意引导别人通过错误的方式来满足他们需求。

你怎么知道是否应该使用云原生基础设施进行架构设计？以下是一些你需要了解的问题，以确定云原生基础设施是否适合你：

- 你有云原生应用程序吗？(有关可从云原生基础设施中受益的应用程序功能，请参阅第1章)
- 你的工程团队是否愿意并且能够编写体现其作业功能的生产质量代码作为软件？
- 你在本地或公有云是否拥有基于API驱动的基础设施(IaaS)？
- 你的业务是否需要更快的开发周期或非线性人员/系统缩放比例？

如果你对所有这些问题都回答“yes”，那么你可能会从本书其余部分介绍的基础设施中受益。如果你对这些问题中的某个问题回答是“no”，这并不意味着你无法从某些云原生实践中受益，但是你可能需要做更多工作，然后才能从此类基础设施中充分受益。

在你的业务准备好之前尝试采用云原生基础设施同样糟糕，因为它将强制使用一个不正确的解决方案。试图在调查不充分的情况下获得收益，你可能会失败，并将其架构视为有缺陷或无益处。由于过去的偏见，一个经过尝试的失败的解决方案以后可能很难采用，无论它是否是正确的解决方案。

在你准备将组织和技术成为云原生时，我们将讨论一些需要关注的领域。有很多事情需要考虑，但一些关键领域是你的应用程序，组织中的人员，基础设施系统和你的业务。

应用程序

应用程序是准备工作最简单的部分。设计模式已经很完善，自公共云出现以来，工具性能得到了显着提升。如果你无法构建云原生应用程序并通过经过验证和测试的管道自动部署它们，则不应继续采用基础设施来支持它们。

构建云原生应用程序并不意味着你必须先拥有微服务。这并不意味着你必须用最新的趋势语言开发你的所有软件。这意味着你必须编写可以由软件管理的软件。

在开发过程中，人类应该与云原生应用程序进行交互。其他一切都应该由基础设施或其他应用程序来管理。

了解应用程序的另一种方法是在它们需要动态地扩展多个实例时。扩展通常意味着负载均衡器后面的同一个应用程序有多个副本。它假定应用程序将状态存储在存储服务（即数据库）中，并且不需要运行实例之间的复杂协调。

动态应用程序管理意味着不需要人参与这项工作。应用程序度量触发了扩展，基础设施做了正确的操作来扩展应用程序。这是大多数云环境的基本特征。运行动态伸缩的资源组并不意味着你拥有云原生基础设施;但如果需要auto-scaling，它可能表明你的应用程序已准备就绪了。

为了使应用程序受益，编写应用程序和配置基础设施的人员需要支持这种工作方法。如果没有人愿意放弃对软件的控制，你将永远无法实现它的好处。

人

人是云原生基础设施中最难的部分。

如果你想建立一个能够用软件取代人们职能和决策的体系结构，你需要确保他们知道你有最大的利益。他们不仅需要接受变化，还需要自己去寻求并改变。

开发应用程序很困难;运营基础设施很难。应用程序开发人员经常相信他们可以用工具和自动化取代基础架构操作员基础架构操作员希望应用程序开发人员能够编写更可靠的代码，并提供自动调试和恢复。这些紧张关系是DevOps的基础，DevOps有许多其他书籍，包括由Jennifer Davis和Katherine Daniels撰写的EffectiveDevOps(O’Reilly，2016)。

人们不会扩大规模，也不擅长重复，平凡的工作。

应用程序和系统工程师的目标应该是消除世俗和重复的任务，以便他们可以专注于更有趣的问题。他们需要具备开发可以包含业务逻辑和决策的软件的技能。需要有足够的工程师来编写所需的软件，更重要的是，维护它。

最关键的方面是他们需要一起工作。如果没有其他方面的支持，工程的一方无法迁移到运行和管理应用程序的新方式。团队组织和沟通结构非常重要。

我们将尽快解决团队准备情况，但首先，我们必须确定基础架构系统何时准备好用于云原生基础设施。

系统

云原生应用程序需要系统抽象。应用程序不应该关注单个硬编码主机名。如果你的应用程序无法在个别主机上运行，那么你的系统尚未准备好使用于云原生基础设施。

使用单个服务器(虚拟或物理)运行操作系统，并将其转换为访问资源的方法，这就是我们所说的“抽象”。单个系统不应该是应用程序部署的目标。资源(CPU、RAM和磁盘)应该集中在所有可用的机器上，然后由平台根据应用程序的请求进行分配。

在云原生基础设施中，你必须隐藏底层系统以提高可靠性。云计算基础设施(如应用程序)会预期的发生基础组件故障，并且旨在优雅地处理此类故障。这是必要的，因为基础设施工程师不再控制堆栈中的所有内容。

> **Kubernetes**云原生基础设施
>
> Kubernetes是一个框架，它使管理应用程序变得更容易，并且以一种云的方式促进了这样做。但是，你也可以用一种非云原生的方式使用Kubernetes。
>
> Kubernetes公开了基于其核心功能的扩展，但这不是你的基础设施的最终目标。其他项目(例如，OpenShift)建立在它之上，将Kubernetes从开发人员和应用程序中抽象出来。
>
> 平台是你的应用程序应该运行的地方。云原生基础设施支持它们，但也鼓励了运行基础设施的方式。
>
> 如果你的应用程序是动态的，但你的基础架构是静态的，那么你很快就会陷入单靠Kubernetes无法解决的僵局。

当它不再是一个挑战时，基础设施已经准备好成为云原生的了。一旦基础设施变得简单，自动化、自助服务和动态，它就有可能被忽略。当系统可以被忽略，并且技术变得单调时，是时候向上移动堆栈了。

如果你的系统管理依赖于硬件定制或在“混合云”中运行，则你的系统可能还没有准备好。可能需要管理一个数据中心，并且私有化。你需要保持警惕，将建立数据中心的责任与管理基础设施的责任分开。

谷歌、Facebook、亚马逊和微软都发现通过开放的计算项目从头开始创建硬件是有好处的。需要创建自己的硬件有性能和成本的限制。因为硬件设计和基础架构构建者之间存在明确的责任分离，这些公司能够在创建定制硬件的同时运行云原生基础设施。它们不会受到“内部部署”的阻碍。相反，他们可以共同优化其硬件和软件，以获得更高的效率和性能。

管理自己的数据中心需要大量时间和金钱的投入。创建一个私有云也是如此。两者都需要建立和管理数据中心团队、创建和维护API的团队以及在IaaS API之上创建抽象的团队。

所有这些都可以完成，并且决定管理整个堆栈是否有价值取决于你的业务。

现在，我们可以看看其他业务领域需要准备什么来转移到云原生实践。

商业

> 如果系统的体系结构和组织的体系结构不一致，则组织的体系结构会胜出。
>
> —鲁斯马兰，“康威定律”

企业变革速度非常缓慢。当通过扩展人员来管理扩展系统不再有效时，以及产品开发需要更多灵活性时，他们可能已经准备好采用云原生实践了。

人无法无限扩展。对于增加管理更多服务器或开发更多代码的每个人来说，支持他们的人力基础设施(例如办公室空间)都有一定的压力。因为需要更多的沟通和协调，其他人也会有额外的开销。

正如我们在第1章中讨论的那样，通过使用公有云，你可以通过租用服务器来减少一些流程和人员开销。即使使用公有云，你仍然会需要管理基础设施详细信息的人员(例如服务器，服务和用户帐户)。

当通信结构反映业务需要创建的基础设施和应用程序时，业务已准备好采用云原生实践。这包括反映像微服务这样的体系架构的通信结构。他们可能是小型的独立团队，无需通过层层管理与其他团队交流或合作。

> **DevOps和Cloud Native**
>
> DevOps可以补充团队合作的方式，并影响使用的工具类型。它对采用它的公司有很多好处，包括快速原型化和提高部署速度。它也非常注重组织的文化。
>
> 云原生需要高性能组织，但更注重于设计，架构和健康度，而不是团队工作流程和文化。但是，如果你认为可以成功的实现云原生模式，而无需解决应用程序开发人员、基础设施运营商以及技术部门中任何人员之间的交互问题，那么你可能会感到意外。

迫使业务变化的另一个限制因素是需要更多的应用程序敏捷性。企业不仅需要快速部署，还需要彻底改变部署的内容。

部署的原始数量无关紧要。重要的是尽可能快地提供客户价值。相信部署的软件将第一次，甚至是第100次，满足所有客户的需求，这是一个谬论。

当业务意识到需要频繁迭代和更改时，它可能已经准备好采用云原生应用程序了。只要它在人员效率和旧流程限制方面遇到限制，并且可以随时更改，就可以为云原生基础设施做好准备。

所有那些表明何时采用云原生的因素都不能说明全部情况。任何设计都需要权衡折衷。因此，在某些情况下，云原生基础设施不是正确的选择。

什么时候不需要云原生基础设施

只有知道限制时，理解系统的好处才是重要的。也就是说，知道哪些限制可以满足您的需求通常可能更多是决定性因素而非利益。

记住需求随时间变化也很重要。现在关键的功能可能不会在未来。同样，如果以下情况现在不会使这种设施变得理想，那么您可以控制许多这些情况，并且可以改变采用它们。

技术限制

就像应用程序一样，在基础架构中，最简单的项目是技术性的。如果您知道什么时候应该采用基于技术优势的云原生基础设施，那么您可以改变这些特征，以找出何时不应该采用云原生基础设施。

第一个限制是没有云原生应用程序。正如在讨论中

第1章，如果您的应用程序需要人工交互，无论是调度，重新启动还是搜索日志，云原生基础设施都没有多大好处。

即使有一个可以动态调度的应用程序，也不会使其成为本地云。如果您的应用程序在Kubernetes上运行，但仍需要人员设置监控，日志收集或负载平衡器，则它不是云原生。只因为你有Kubernetes并不意味着你已经完成。

如果你有一个管弦乐队，重要的是看看它是如何运行的。您是否需要下订单，创建票据或发送电子邮件以获取服务器？

这些是您没有自助服务基础架构的指标，这是云计算的一项要求。

在云中，您只需提供帐单信息并调用API。即使您在内部运行服务器，您也应该有一个可以构建IaaS的团队，然后将云原生基础设施分层布局。

如果您要在自己的数据中心中构建云环境，图2-1显示了您的基础架构组件适合的示例。所有原始组件（例如，计算机，存储，网络）都应该可以从自助式IaaS API中获得。

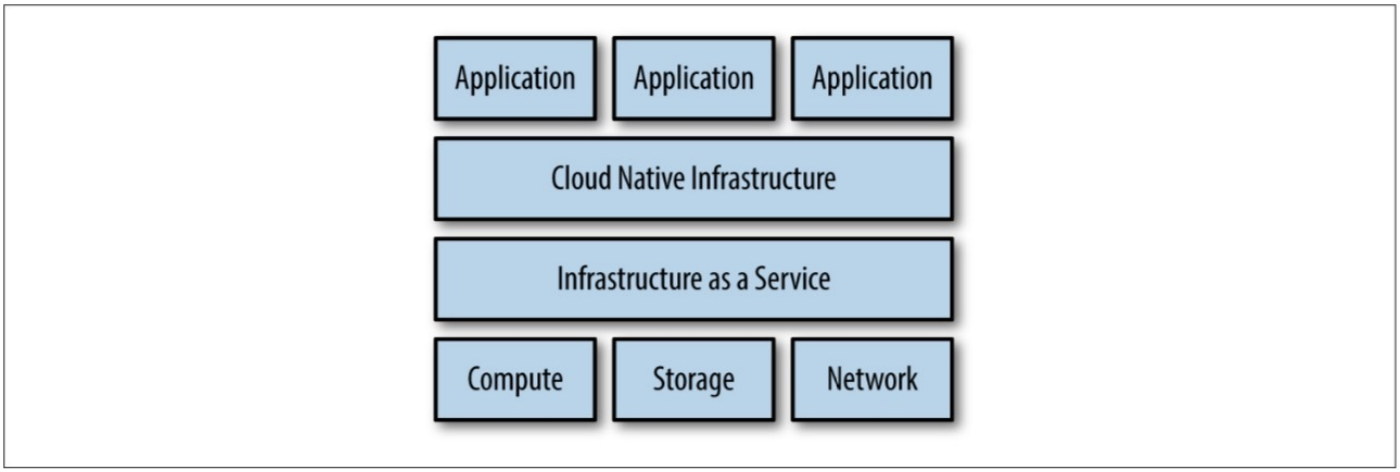

图2-1. 云原生基础设施的示例图层

在公共云中，您拥有IaaS和托管服务，但这并不意味着您的业务已准备好迎接公有云的启用。

当您构建运行应用程序的平台时，了解您正在进行的操作非常重要。最初的开发只是构建和维护平台所需花费的一小部分，特别是对业务至关重要的平台。

维护通常会消耗大约40%到80%（平均60%）的软件成本。因此，这可能是最重要的生命周期阶段

发现业务需求和建立开发所需的技能可能对于一个小团队来说太过分了。一旦你掌握了开发所需平台的技能，你仍然需要投入时间来改进和维护系统。这需要比初始开发更长的时间。

为企业提供绝对最佳的环境是公共云提供商的产品。如果你不能或者不愿意让你的平台成为你的业务的一个区别，那么你不应该自己创建一个平台。

请记住，你不必自己构建一切。您可以使用可以组装到所需平台的服务或产品。

可靠性仍然是您基础架构的关键特性。如果您还没有准备好放弃对基础设施堆栈的更低级别的控制，并且仍然通过接受故障来制造可靠的产品，那么云原生基础设施并不是正确的选择。

也有可能超出你的控制的限制。非技术限制同样重要，难以解决。

业务限制

如果现有流程不支持更改基础架构，则需要首先克服该障碍。幸运的是，你不必一个人做。

本书希望有助于向需要说服力的人清楚地解释好处和流程。还有许多案例研究和公司分享他们采用这些做法的经验。本书附录C中将提供一个案例研究，但是您可以为您的企业找到相关示例并与同行和管理层分享。

如果企业还没有实验的途径和支持尝试新事物的文化（以及伴随失败而来的后果），那么改变流程可能是不可能的。在这种情况下，您的选择是达到必须改变事物的临界点，或者说服管理层认为改变是必要的。

从外部看，企业是否准备采用云原生设施是不可能的。一些明确指出公司未准备好包括的过程：

- 需要人工干预的资源请求
- 定期安排需要人工操作的维护窗口
- 手动库存跟踪和资源分配
- 电子表格清单

如果除了负责服务的团队以外的其他人员参与进程以调度，部署，升级或监视服务，则可能需要在迁移到云原生基础设施之前或迁移期间解决这些流程。

有时也有业务无法控制的过程，例如行业法规。可悲的是，这些变化甚至比内部流程更难和更慢。

如果行业法规限制了发展的速度或敏捷性，我们就没有任何建议，除非你能做到。如果法规不允许在公共云中运行，请尽量使用技术来运行内部部署。管理层将需要为任何管理机构制定的法规制定一个案例。

云原生基础设施还有另一个非技术障碍。在一些公司中，有一种不使用第三方服务的文化

如果您的公司不愿意或通过无法使用的流程来使用第三方托管服务，则可能不适合采用云原生基础设施。我们将在附录B中更详细地讨论何时使用托管服务。

结论

> 要成功，单靠计划是不够的。一个人也必须即兴创作
>
> —艾萨克·阿西莫夫

在本章中，我们讨论了何时采用云原生基础设施的注意事项。有许多领域要记住，每种情况都是独一无二的。希望这

些指导原则中的一部分可以帮助您发现进行变更的适当时机。

如果您的公司已经采用了一些云原生实践，这些问题可以帮助确定可以采用这种架构的其他领域。当你应该采用这些权衡和利益是正确的解决方案，以及如何开始时，了解这一点非常重要。

如果您尚未将云原生实践应用于您的工作，则没有捷径。企业和员工将需要共同决定这是正确的解决方案，并共同取得进展。没有人独自成功。

第3章 云原生部署的演变

我们在前一章中讨论了在采用云原生基础设施之前需要什么。在部署之前，需要有API驱动的基础设施供应（IaaS）。

在本章中，我们将探讨云原生基础设施拓扑的概念，并在云中实现它们。我们将学习可以帮助运维人员控制其基础设施的常用工具和模式。

部署基础设施的第一步应该是能够将其表述出来。传统上，可以在白板上处理，或者如果幸运的话，可以在公司wiki上存储的文档中处理。今天，一切都变得更加程序化，基础设施表述通常以便于应用程序解释的方式记录。无论如何表述，全面的表述基础设施的需求是不变的。

正如人们所期望的那样，精巧的云基础设施可以从简单的设计到非常复杂的设计。无论复杂性如何，必须对基础设施的表现给予高度的重视，以确保设计的可重复性。能够清晰地传递想法更为重要。因此，明确、准确和易于理解的基础设施级资源表述势在必行。

我们也将从制作精良的表述中获得很多好处：

- 随着时间的推移，基础设施设计可以共享和版本化。
- 基础设施设计可以被fork和修改以适应特殊情况。
- 表述隐含的是文档。

随着本章向前推进，我们将看到基础设施表述是如何成为基础设施部署的第一步。我们将以不同的方式探索表述基础设施的能力和潜在缺陷。

表述基础设施

首先，我们需要理解表述基础设施的两个角色：作者和观众。

作者将定义基础设施，通常是人类运维人员或管理员。观众将负责解释基础设施表述。有时候，这是一个运维人员执行手动步骤，但希望它是一个可以自动分析和创建基础设施的部署工具。作者在准确表达基础设施方面表现得越好，我们就可以在听众解释表达的能力中获得更多的信心。

创作基础设施表述时主要关心的是要让观众了解它。如果目标受众是人，则表述可能以技术图或抽象代码的形式出现。如果目标受众是一个程序，那么表示可能需要更详细的信息和具体的实施步骤。

尽管有观众，作者应该让观众更容易使用。随着复杂性的增加以及人与程序共同使用基础设施，这将变得非常困难。

表示法需要易于理解，以便能够对其进行准确分析。易于阅读但分析不准确的表述否定了整个工作。观众应该总是努力去解释他们的表述，而不是做出假设。

为了使表达成功，解释需要可预测。如果作者忽略了一个重要的细节，那么最好的观众就会很快失败。具有可预测性将在应用变更时减少错误的发生，并有助于在作者和受众之间建立信任。

基础设施即图

我们用到了白板，开始绘制一张基础设施图。通常情况下，这个过程始于在角落上代表互联网的云形状，以及一些指

向方框的箭头。每个框代表系统中的一个组件，箭头表示它们之间的交互。图3-1是基础设施图的一个例子。

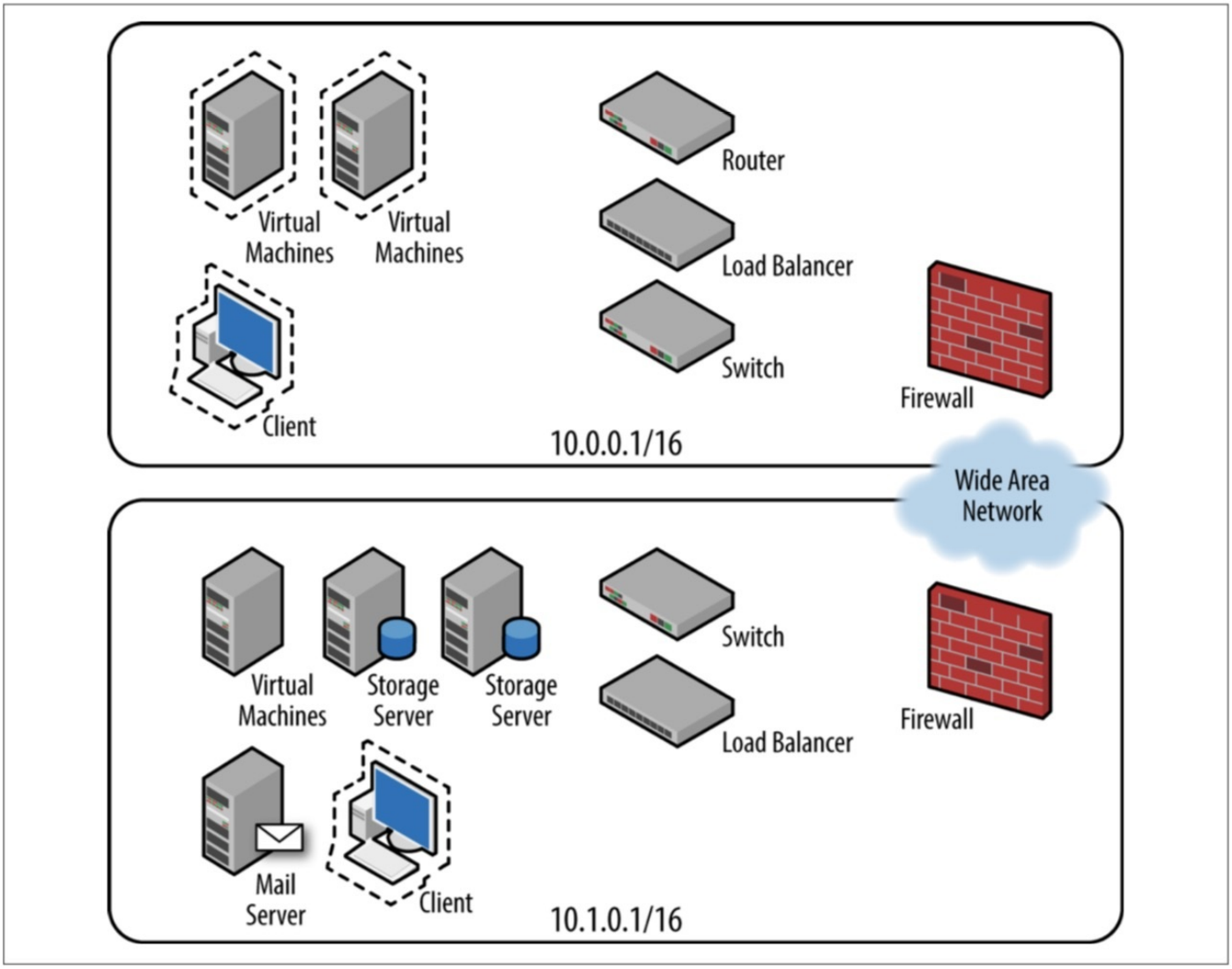

图3-1. 简单的基础设施图

这是一个非常有效的头脑风暴和将想法传达给其他人的方法。它允许对复杂的基础设施设计进行快速而强大的表示。

图片适用于人类，大量人群和CEO。这些图也适用，因为它们使用常用语来表示关系。例如，此框可能会将数据发送到那个框，但不会将数据发送到其他框。

不幸的是，图表对于计算机来说几乎是不可能理解的。在计算机视觉迎头赶上之前，基础设施图仍然是一个代表，可以用眼球来解释，而不是代码。

> 从图中进行部署
>
> 在例3-1中，我们看一个来自bash_history文件的熟悉的代码片段。它代表一个基础设施运营商，作为描述基础服务器与网络、存储和转租服务运行的图表的受众。
>
> 运维人员已经手动部署了一台新的虚拟机，并通过SSH连接到了该机器并开始配置它。在这种情况下，人类充当图解释者，然后在基础设施环境中采取行动。
>
> 大多数基础设施工程师在他们的职业生涯中都这样做了，而且这些步骤对于某些系统管理员来说应该是非常熟悉的。

例3-1. bash_history

```
1. sudo emacs /etc/networking/interfaces
```

```
sudo ifdown eth0
sudo ifup eth0
sudo fdisk -l
sudo emacs /etc/fstab
sudo mount -a
sudo systemctl enable kubelet
```

基础设施即脚本

如果您是一个系统管理员，您工作的一部分是在复杂系统中进行更改；确保这些更改是正确的也是您的责任。需要将这些变化传播到广阔的系统中是非常现实的。不幸的是，人为错误也是如此。管理员为这项工作编写便利脚本并不奇怪。

脚本可以帮助减少重复任务中人为错误的数量，但自动化是一把双刃剑。这并不意味着准确性或成功。

> 对于SRE，自动化可以让你力量倍增，单它不是万能药。当然，加倍的力量并不会自然地改变应用力的准确性：不经意地进行自动化可能会产生很多的问题。
>
> ——Niall Murphy、John Looney和Kacirek，自动化在谷歌的演变

编写脚本是自动执行步骤以产生所需结果的好方法。该脚本可以执行各种任务，例如安装HTTP服务器，配置并运行它。但是，脚本中的步骤在调用时很少考虑到它们的结果或系统的状态。

在这种情况下，脚本是编码数据，表示创建所需基础设施应该发生的情况。另一位运维人员或管理员可以评估您的脚本，并希望了解脚本正在做什么。换句话说，他们会解释你的基础设施表示。了解所需的基础设施需要了解步骤如何影响系统。

脚本的运行时会按照它们定义的顺序执行这些步骤，但运行时不知道它正在生成什么。脚本是代码，脚本的执行结果希望是所需的基础设施。

这适用于普遍的场景，但这种方法存在一些缺陷。最明显的缺陷是运行相同的脚本可能获得两个不同的结果。

如果脚本第一次运行的环境与第二次运行的环境大不相同？从科学的角度来说，这将类似于程序中的缺陷，并会使实验数据无效。

使用脚本来表示基础设施的另一个缺陷是缺少声明状态。脚本的运行时不理解结束状态，因为它只提供了执行步骤。人类需要从步骤中解释理想的结果，以了解如何进行改变。

我们看到过很多人类难以理解的代码。随着配置脚本复杂性的增长，我们解释脚本的能力就会减弱。此外，您的基础设施页需要随时间而变化，脚本将不可避免地需要更改。

如果不将步骤抽象为声明性状态，为了给每个可能的初始状态创建过程，脚本将不断增长。这包括抽象出操作系统（例如apt和DNF）之间的步骤和差异，以及验证可以安全地跳过哪些步骤。

基础设施即代码带来了一些工具，这些工具提供了一些抽象，以帮助减轻使用脚本管理基础设施的负担。

> 从脚本部署
>
> 创建基础设施的下一个发展是开始采用先前手动管理基础设施的流程，并通过将工作封装在脚本中来简化它。想象一下，我们有一个名为createVm.sh的bash脚本，它将在我们的本地工作站中创建一台虚拟机。
>
> 该脚本需要两个参数。第一个是分配给虚拟机上的网络接口的静态IP地址。第二个是以千兆字节为单位的大小，用于创建卷并将其挂载到虚拟机。

> 示例3-2将基础设施的基本表示形式显示为脚本。该脚本将提供新的基础设施，并在新创建的基础设施上运行任意配置脚本。该脚本可能演变为高度可定制的，并且可能是（危险地）自动化的，只需点击一下按钮即可运行。

例3-2. 基础设施即脚本

```
#!/bin/bash

# Create a VM with a NIC on 10.0.0.17 and a 100gb volume

createVm.sh 10.0.0.17 100

# Transfer the bootstrapping script

scp ~/vm_provision.sh user@10.0.0.17:vm_provision.sh -v

# Run the bootstrap script

ssh user@10.0.0.17 sh ~/vm_provision.sh
```

基础设施即代码

配置管理曾经是代表基础设施的主要角色。我们可以将配置管理视为抽象脚本，自动考虑初始状态以执行正确的过程。最重要的是，配置管理允许作者声明节点的期望状态，而不是实现它所需的每一步。

配置管理是基础设施即代码的第一步，但相关工具很少超出单个服务器的范围。配置管理工具在定义特定资源和他们的状态方面做得非常出色，但由于基础设施需要资源之间的协调，所以出现了复杂性。

例如，服务的DNS条目在提供服务之前不可用。在主机可用之前不应该提供该服务。如果不能在独立节点之间协调多个资源，则配置管理提供的抽象化是不足的。有些工具增加了协调资源之间配置的能力，但协调通常是程序性的，责任落到了人们的协调资源和理解所需状态上。

您的基础设施不包含没有通信的独立实体。代表基础设施的工具需要考虑到这一点。因此，需要另一种表示来管理低级别抽象（例如操作系统）以及供应和协调。

2014年7月，有个开源工具在代码发布的时候采用了更高级别的基础设施抽象概念。这个名为Terraform的工具非常成功。它在配置管理完善并且公有云的采用呈上升趋势的时间节点发布。用户看到了新环境中工具的局限性，Terraform很好的满足了他们的需求。

> 在2011年时，我们最初将基础设施视代码。我们注意到我们正在编写工具来解决许多项目的基础设施问题，并希望将流程标准化。
>
> ——Hashicorp首席执行官兼Terraform创始人Mitchell Hashimoto

Terraform使用专门的领域特定语言（DSL）表示基础设施，它在人类可理解的图像和机器可分析的代码之间做了良好的折衷。Terraform最成功的部分是抽象的基础设施视图，资源协调以及应用时利用现有工具的能力。Terraform与云API进行通信以配置基础设施，并可在必要时使用配置管理来配置节点。

这是该行业的根本性转变，因为我们看到一次性配置脚本正在消失。越来越多的运营商开始在新的DSL中开发基础设施表示。过去在基础设施上手动操作的工程师现在正在开发代码。

新的DSL解决了将基础设施表示为脚本的问题，并成为表示基础设施的标准。工程师发现他们正在开发更好的基础设施代码，并允许Terraform对其进行解释。与配置管理代码一样，工程师们开始将他们的基础设施表述存储在版本控制

系统中，并将基础设施与软件等同看待。

通过表述基础设施的标准化方式，我们摆脱了学习各种专有云API的痛苦。尽管并非所有云资源都可以用单一表示抽象出来，但大多数用户可以接受其代码中的云锁定。拥有人类可读并且机器可解析的基础设施表示，而不仅仅是独立的资源声明，这一点永远得改变了行业。

> 从代码部署
>
> 在面临将基础设施部署为脚本的挑战之后，我们已经创建了一个程序来解析输入并针对我们的基础设施采取行动。
>
> 例3-3显示了从Terraform开源库中获取的Terraform配置。注意代码中有变量，需要在运行时解析。
>
> 基础设施的声明性表示很重要，因为它没有定义创建基础设施的各个步骤。这使我们能够分离需要调配的部分和调配的部分。这就是使这种基础设施代表成为新范例的原因；这也是向软件基础设施演进的第一步。
>
> 以这种方式来表示基础设施对于工程师来说是一种常见的强大做法。用户可以使用Terraform来应用基础设施。

例3-3. example.tf

```
# Create our DNSimple record
resource "dnsimple_record" "web" {
domain = "${var.dnsimple_domain}"
name = "terraform"
value = "${hostname}"
type = "CNAME"
ttl = 3600
}
```

基础设施即软件

基础设施即代码是朝着正确方向发展的强大举措。但是代码是基础设施的静态表示，并且有其局限性。您可以自动执行部署代码更改的过程，但除非部署工具持续运行，否则仍会出现配置漂移。传统上，部署工具只能在一个方向上工作：它只能创建新对象，并且不能轻易删除或修改现有对象。

为了掌握基础设施，我们的部署工具需要根据基础设施的初始表示进行工作，并对数据进行变更以创建更灵活的系统。当我们开始将基础设施表示视为一个可持续执行所需状态的可版本化数据体时，下一步就是将基础设施视为软件。

Terraform从配置管理中吸取教训并改进了这一概念，以更好地配置基础设施和协调资源。应用程序需要一个抽象层来更有效地利用资源。正如我们在第1章中所解释的那样，应用程序不能直接在IaaS上运行，而需要在可以管理资源和运行应用程序的平台上运行。

IaaS将原始组件作为临时API端点呈现，平台呈现更容易被应用程序使用的资源的API。其中一些资源可能提供IaaS组件（例如，负载均衡器或磁盘卷），但其中许多资源将由平台管理（例如，计算资源）。

平台揭示了一个新的基础设施层，并不断强化所需的状态。平台的组件也是应用程序本身，可以使用相同的期望状态声明进行管理。

API机制允许用户获得将基础设施标准化为代码的好处，并增加了随着时间的推移版本化和更改表示的能力。API允许通过标准实践（如API版本控制）消费资源的新方式。API的使用者可以将其应用程序构建到特定的版本，并相信在使用新的API版本之前，它们的使用不会中断。其中有些做法是以前基础设施即代码工具所缺少的重要功能。

通过持续强化表示的软件，我们现在可以保证我们系统的当前状态。通过提供正确的抽象，平台层变得更加易于使用。

您可能正在绘制基础设施演变与软件演进之间的相似之处。堆栈中的这两层以非常相似的方式进化。

> 软件正在吞噬世界。
>
> —Marc Andreessen

封装基础设施并将其视为版本化的API将会非常强大。这极大地提高了负责解释表示的软件项目的速度。由平台提供的抽象是跟上快速增长的云所必需的。这种新模式是当今的模式，并且已经被证明可以扩展到难以估量的基础设施和应用程序。

> 从软件部署
>
> 基础设施即代码和基础设施与软件之间的根本区别在于，软件能够改变数据存储，从而改变基础设施的表示。这是由软件来管理基础设施，代表是运营商和软件之间的交换。
>
> 在例3-4中，我们看看使用YAML表示的基础设施。我们可以信任该软件来解释这种表示，并为呈现YAML的结果。
>
> 就像与我们开发基础设施代码时一样，我们从基础设施的表示开始。但在这个例子中，软件会持续运行，并确保表示会随时间的推移。从某种意义上说，这仍然是只读的，但是软件可以扩展这个定义来添加自己的元信息，比如标记和资源创建时间。

例3-4. infrastructure.yaml

```
location: "New York 1"
name: example
dns:
    fqdn: infra.example.com
network:
  cidr: 172.0.0.0/12
serverPools:
  - bootstrapScript: /home/user/bootstrap.sh
    diskSize: 40gb
    firewalls:
      - rules:
          - ingressFromPort: 443
            ingressProtocol: tcp
            ingressSource: 0.0.0.0/0
            ingressToPort: 443
    maxCount: 1
    minCount: 1
    image: centos-amd64-7
    subnets:
      - cidr: 172.0.100.0/24
```

部署工具

我们现在了解部署基础设施的两个角色：

作者

定义基础设施的组件

观众

部署工具解释表示并采取行动

我们可以通过很多途径来表述基础设施，采取行动的组成部分是对最初表示的逻辑反映。准确地表示适当的基础设施层并尽可能消除该层的复杂性非常重要。通过简单、有针对性的发布，我们将能够更加准确地应用所需的更改。

《站点可靠性工程》（O’Reilly，2016）总结说：“简单版本通常比复杂版本更好。衡量和理解单一变更的影响，而不是同时发布的一批变更要容易得多。“

随着我们对基础设施的表示随着时间的推移而变化，以便从底层组件中抽象出来，我们的部署工具已经发生变化，以匹配新的抽象目标。

我们正在将基础设施视为软件边界，并且可以注意到基础设施部署工具新时代的早期迹象。互联网上的开源项目正在出现，声称能够随着时间的推移管理基础设施。工程师的工作是了解项目管理的基础设施层以及它如何影响其现有工具和其他基础设施层。

云原生基础设施方向的第一步是采用配置脚本并安排它们持续运行。有些工程师会故意设计这些脚本，以便随着时间的推移安排好。我们开始看到精心设计的全局锁定机制、高级调度策略和分布式调度方法。

这基本上是配置管理承诺的，尽管在更具资源特定的抽象中。感谢云计算，管理基础设施的自动化脚本的日子已经过去了。

> 自动化已死。
>
> —Honeycomb首席执行官Charity Majors

我们正在想象一个我们开始以完全不同的方式看待基础设施工具的世界。如果您的基础设施旨在运行在云上，那么IaaS不是您应该解决的问题。使用云提供的API，并构建可直接由应用程序使用的新基础设施层。

我们在基础设施发展方面处于特殊地位，我们从第一天开始就将基础设施部署工具设计为优雅的应用程序。

良好的部署工具是可以快速从基础设施的人性化表示到可工作基础设施的工具。更好的部署工具是撤销任何与初始表示不一致的变更的工具。最好的部署工具可以完成所有这些工作，而无需人工参与。

在我们构建这些应用程序时，我们不能忘记从处理复杂系统至关重要的专业工具和软件实践中学到的重要经验。

我们将看到的部署工具的一些关键方面是幂等性和处理失败。

幂等性

软件应该是幂等的，这意味着持续输入相同的输入，必须并始终获得相同的输出。

在技术上，这个想法被超文本传输协议（HTTP）通过像PUT和DELETE这样的幂等方法而著名。这个想法飞非常强大，并且在软件中宣传幂等性的保证可以塑造出更佳复杂的软件应用程序。

我们从早期的配置管理工具中学到的经验之一就是幂等性。我们需要记住这个功能为基础设施工程师提供的价值，并且继续将这种模式构建到我们的工具中。

能够自动创建、更新或删除基础设施，保证无论您运行任务的频率如何，始终都会输出相同的结果，这非常令人兴

奋。它允许运维人员开始自动化任务和杂事。过去对于运维人员来说，过去相当大量的工作现在可以像在网页中点击按钮一样简单。

幂等保证也有助于运营商在其基础设施上执行质量科学。运营商可以在许多物理位置开始复制基础设施，并知道别人重复他们的程序会得到同样的结果。

我们开始注意到围绕这种自动执行任意任务以实现可重复性的思想构建的整个框架和工具链。

就像软件一样，基础设施也是如此。运营商开始使用这些表示和部署工具自动管理整个管理基础设施的流水线。现在，运维人员的工作变成了开发自动执行这些任务的工具，而不再是自己执行任务。

处理失败

任何软件工程师都可以告诉你在代码中处理故障和边缘案例的重要性。作为基础设施管理员我们自然而然就要考虑这些问题。

如果部署作业在执行过程中失败，更重要的是在这种情况下会发生什么，会发生什么情况？

在考虑失败的情况下设计我们的部署工具是朝着正确方向迈出的又一步。失败时发送消息或在监控系统中注册警报。我们保存了自动化任务的详细日志。在失败的情况下，我们甚至将逻辑连接在一起。

我们沉迷于失败。我们在失败的情况下开始采取行动，并在事件发生时采取行动。

但围绕单个组件可能出现故障的想法来构建系统与构建组件以使其更容易出故障完全不同。根据故障重试组件或调整其方法是将系统的弹性进一步深入到软件中。这允许更稳定的系统并减少系统本身所需的整体支持。

面向故障而设计组件，而不是系统。

最终一致性

以设计失败的组件为名，我们需要学习一个描述处理失败的常用方法的术语。

最终的一致性意味着企图随着时间的推移调和一个系统。较大的系统和较小的组件都可以遵循这种随时间推移重试失败过程的理念。

最终一致的系统的好处之一是运维人员可以确信它最终会达到预期的状态。这些系统的一个担忧是，有时他们可能花费不恰当的时间来达到所需的状态。

知道什么时候选择一个稳定但缓慢的系统与一个不可靠但快速的系统是管理员必须做出的技术决策。在这个决定中要注意的重要关系是系统交换速度的可靠性。这并不容易，但如果有疑问，请始终选择可靠的系统。

原子性

与最终一致的系统相反的是原子系统，这是一项保证交易，决定了整个工作的成功。如果作业无法完成，则会恢复所做的更改并完全失败。

想象一下需要创建10个虚拟机的工作。工作到达第七台虚拟机，出现问题。根据最终的一致性方法，我们只会反复尝试这项工作，希望最终获得10个虚拟机。

了解我们只能创建7个虚拟机的原因非常重要。想象一下，云计算允许我们创建多少个虚拟机是有限制的。最终一致性模型将继续尝试创建另外三台机器，并且不可避免地会失败每次。

如果这项工作是设计成原子的，那么它将在第七台机器上达到极限，并意识到这是一场灾难性的失败。这项工作将负责删除部分系统。

因此，运维人员可以放心，他们或者完全按照预期建立系统，或者根本不会创建任何东西。这是一个很有意义的想法，因为为了能让系统正常工作，基础设施中的许多组件都依赖于系统中的其他部分。

我们可以引入信心来换取不便。也就是说，管理员会相信他们的系统状态永远不会改变，除非可以应用完美的改变。为了交换这个完美的系统，运维人员可能会面临很大的不便，因为系统可能需要很多工作才能保持平稳运行。

选择一个原子系统是安全的，但可能不是我们想要的。工程师需要知道他们想要什么系统，以及何时选择原子性与最终一致性。

结论

部署基础设施的模式很简单，并且在云可用之前一直保持不变。我们代表基础设施，然后使用一些设备，将基础设施变为现实。

基础设施层与软件应用层具有惊人的类似历史。云原生基础设施也不例外。我们开始发现自己在重复历史，并以新的方式学习古老的教训。

如果我们已经知道其软件对手的未来，那么对于预测基础设施行业未来的能力还有什么要说的？

云原生基础设施是基础设施演变的一种自然而可能预期的结果。能够以可靠和可重复的方式部署、表示和管理它是必要的。随着时间的推移，我们能够部署我们的部署工具，并转移我们的工作方式，这对于将我们的基础设施保持在一个能够支持其应用层的空间中至关重要。

第4章 设计基础设施应用程序

在前一章中，我们了解了代表基础设施以及围绕它的部署工具的各种方法。在本章中，我们将看看如何设计部署和管理基础设施的应用程序。在上一章中我们重点关注基础设施即软件的开放世界，有时称为基础设施即应用。

在云原生环境中，传统的基础设施运维人员需要成为基础设施软件工程师。这仍然是一种新兴的做法，与过去的其他运营角色不同。我们迫切需要开始探索模式和制定标准。

基础设施即软件与基础设施即代码之间的根本区别在于，软件会持续运行，并会根据调解器模式创建或改变基础设施，我们将在本章后面对其进行解释。此外，基础设施即软件的新范例是，软件现在与数据存储具有更传统的关系，并公开用于定义所需状态的API。例如，该软件可能会根据数据存储中的需要改变基础设施的表示形式，并且可以很好地管理数据存储本身！希望进行协调的状态更改通过API发送到软件，而不是通过运行静态代码库中的程序。

迈向基础设施即软件的第一步是让基础设施的运维人员意识到自己是软件工程师。我们热烈欢迎您来到这个领域！先前的工具（例如配置管理）也有类似的目标来改变基础设施运维人员的工作职能，但是运维人员通常只会在狭窄的应用范围内编写有限的DSL（即单一节点抽象）。

作为一名基础设施工程师，您的任务不仅是掌握设计、管理和运维基础设施的基本原则，还需要具有将您的专业知识封装成坚如磐石的应用程序的能力。这些应用程序代表了我们将要管理和改变的基础设施。

构建管理基础设施软件工程不是一件容易的事情。我们有管理传统应用的所有问题和担忧，而且我们正处于一个尴尬的境地。基础设施软件工程看上去似乎很荒谬，构建软件来部署基础设施，这样就可以在新创建的基础设施之上运行相同的软件，这很尴尬。

首先，我们需要了解这个新领域中工程软件的细微差别。我们将研究在云原生社区中得到验证的模式，以了解在应用程序中编写干净和逻辑代码的重要性。但首先，基础设施从哪里来？

自举问题

1987年3月22日，周日，Richard M. Stallman发送了一封电子邮件到GCC邮件列表，报告成功使用C编译器完成了自行编译：

该编译器在*68020*上编译正确，最近又在*vax*上进行了编译。最近在*68020*上正确编译了*Emacs*，并且还编译了*texin-C*和*Kyoto Common Lisp*。但是，可能仍然有许多错误，希望你能帮我找到。

我将离开一个月，所以现在报告的错误将得不到处理。—*Richard M. Stallman*

这是软件历史上的一个重要转折点，因为工程软件首次完成了自举。Stallman开创了一个可以自行编译的编译器。即使在哲学上接受这个表述可能也是困难的。

今天我们正在解决与基础设施相同的问题。工程师必须想办法解决几乎不可能的系统自举问题，并在运行时生效。

一种方法是手动创建云计算和基础设施应用程序中的第一个基础设施。尽管这种方法确实有效，但它通常伴随着警告，即运维人员应该在部署更合适的基础设施后销毁初始引导基础设施。这种方法乏味、难以重复且容易出现人为错误。

解决这个问题的更优雅和云原生方法是做出（通常是正确的）假设，试图引导基础设施软件的任何人都有本地机器，

我们可以利用这个本地机器。现有机器（您的计算机）可作为第一个部署工具，自动在云中创建基础设施。基础设施就位后，您的本地部署工具可以将其自身部署到新创建的基础设施并持续运行。良好的部署工具可以让你在完成后轻松清理。

在初始基础设施引导问题解决后，我们可以使用基础设施应用程序来引导新的基础设施。现在本地计算机已经被排除在外，现在我们完全运行在云端。

API

在前面的章节中，我们讨论了表示基础设施的各种方法。在本章中，我们将探讨为基础设施提供API的概念。

当用软件实现API时，很可能会通过数据结构来完成。因此，根据您使用的编程语言，将API视为类、字典、数组、对象或结构是安全的。

API将是数据值的任意定义，可能是字符串、整数或布尔值。API将通过JSON或YAML格式进行编码和解码甚至可能存储在数据库中。

对于大多数软件工程师来说，为程序提供可版本化的API是很常见的做法。这允许程序随着时间移动、改变和增长。工程师可以声称支持较旧的API版本并提供向后兼容性保证。在基础设施即软件中，由于这些原因，使用API是优选的。

寻找一个API作为基础设施的接口是用户使用基础设施即软件的许多线索之一。传统上，基础设施即代码是用户将要管理的基础设施的直接表示，而API是管理的确切底层资源之上的抽象。

最终，API只是代表基础设施的数据结构。

状态

在基础设施即软件工具的环境中，我们要管理的对象是基础设施。因此，对象状态只是我们的程序对软件的审计表示。

对象的状态最终将回到基础设施表示的内存中。这些内存中的表示应映射到用于声明基础设施的原始API。审计的API或对象状态通常需要保存。

存储介质（有时称为状态存储）可用于存储新审计的API。介质可以是任何传统存储系统，例如本地文件系统、云对象存储或数据库。如果数据存储在类似文件系统的存储中，那么该工具将很可能以逻辑方式对数据进行编码，以便可以在运行时轻松对数据进行编码和解码。常见的编码包括JSON、YAML和TOML。

当设计程序时您可能会想要将用于存储其他数据的特权信息存储起来。这究竟是不是最佳实践具体取决于您的安全性要求以及您计划存储数据的位置。

记住存储秘密可能是一个漏洞，这一点很重要。在设计软件来控制堆栈最基本的部分时，安全性至关重要。所以通常值得额外的努力来确保秘密是安全的。

除了存储有关程序和云提供商凭证的元信息之外，工程师还需要存储有关基础设施的信息。重要的是要记住，基础设施将以某种方式呈现，理想情况下，该程序易于解码。记住对系统进行更改不会立即发生，而随着时间的推移也很重要。

存储这些数据并能够轻松访问是设计基础设施管理应用程序的重要部分。仅基础设施定义很可能就已经是系统中最具智慧价值的部分。我们来看一个基本的例子，看看这些数据和程序如何一起工作。

重新审视例4-1至4-4，因为它们被用作本章进一步演示的具体例子。

一个文件系统状态存储示例

想象一下，数据存储在一个名为state的目录中。在该目录中，有三个文件：

- meta_information.yaml
- secrets.yaml
- infrastructure.yaml

这个简单的数据存储可以准确地封装需要保留的信息，以便有效管理基础设施。

`secrets.yaml` 和 `infrastructure.yaml` 文件存储基础设施的表示形式， `meta_information.yaml` 文件（示例4-1）存储其他重要信息，例如基础设施上次调配时间，调配时间和日志信息。

例4-1. state/meta_information.yaml

```
lastExecution:
  exitCode: 0
  timestamp: 2017-08-01 15:32:11 +00:00
  user: kris
  logFile: /var/log/infra.log
```

第二个文件 `secrets.yaml` 保存私人信息，用于在程序执行过程中以任意方式验证（例4-2）。

重申一下，以这种方式存储秘密可能是不安全的。我们仅以 `secrets.yaml` 为例。

例4-2. state/secrets.yaml

```
apiAccessToken: a8233fc28d09a9c27b2e2f
apiSecret: 8a2976744f239eaa9287f83b23309023d
privateKeyPath: ~/.ssh/id_rsa
```

第三个文件 `infrastructure.yaml` 将包含API的编码表示形式，包括使用的API版本（示例4-3）。我们可以在这里找到基础设施表示，例如网络和DNS信息，防火墙规则和虚拟机定义。

例4-3. state/infrastructure.yaml

```
location: "San Francisco 2"
name: infra1
dns:
    fqdn: infra.example.com
network:
  cidr: 10.0.0.0/12
  serverPools:
  - bootstrapScript: /opt/infra/bootstrap.sh
    diskSize: large
    workload: medium
    memory: medium
    subnetHostsCount: 256
    firewalls:
```

```
      - rules:
          - ingressFromPort: 22
            ingressProtocol: tcp
            ingressSource: 0.0.0.0/0
            ingressToPort: 22
    image: ubuntu-16-04-x64
```

起初 `infrastructure.yaml` 文件可能看起来只不过是基础设施代码的一个例子。但是，如果仔细观察，您会发现许多定义的指令都是具体基础设施之上的抽象。例如， `subnetHostsCount` 指令是一个整数值并定义了子网中主机的预定数量。该程序将设法为运维人员划分网络中定义的更大的无类别域间路由（CIDR）值。运维人员不会声明子网，只需要声明有多少主机。软件会帮运维人员完成剩下的操作。

程序运行时可能会更新API并将新的表示写入数据存储区（本案例中仅是一个文件）。继续我们的 `subnetHostsCount` 示例，假设程序确实为我们挑选了一个子网CIDR。新的数据结构可能如例4-4所示。

```
location: "San Francisco 2"
name: infra1
dns:
    fqdn: infra.example.com
network:
  cidr: 10.0.0.0/12
serverPools:
  - bootstrapScript: /opt/infra/bootstrap.sh
    diskSize: large
    workload: medium
    memory: medium
    subnetHostsCount: 256
    assignedSubnetCIDR: 10.0.100.0/24
    firewalls:
      - rules:
          - ingressFromPort: 22
            ingressProtocol: tcp
            ingressSource: 0.0.0.0/0
            ingressToPort: 22
    image: ubuntu-16-04-x64
```

请注意程序如何编写assignedSubnetCIDR指令，而不是由运维人员操作。另外，请记住更新API的程序是用户如何以软件方式与基础设施进行交互的标志。

现在，请记住，这只是一个例子，并不一定主张使用抽象计算子网CIDR。不同的用例可能需要在应用程序中进行不同的抽象和实现。关于构建基础设施应用程序的一个好处是，用户可以以任何他们认为可以解决自己问题的方式设计软件。

数据存储（ `infrastructure.yaml` 文件）现在可以被认为是软件工程领域的传统数据存储。也就是说，该程序可以对文件进行完全的写入控制。

我们会发现，这会带来风险，但对工程师来说也是一个巨大的胜利。基础设施表示不必存储在文件系统的文件中。相反，它可以存储在任何数据存储中，如传统数据库或键/值存储系统。

为了理解软件如何处理这种新的基础设施表示的复杂性，我们必须理解系统中的两种状态—API形式的预期状态，可

在 `infrastructure.yaml` 文件中找到，另一种可以在现实（或审计）中观察到的实际状态。

在这个例子中，软件还没有做任何事情或者采取任何行动，而我们正处于管理时间线的开始。因此，实际状态将是什么都没有，而预期状态将是封装在 `infrastructure.yaml` 文件中的任何状态。

调解器模式

调解器模式（reconciler pattern）是一种软件模式，可用于管理云原生基础设施。该模式强化了基础设施的两种表现形式—第一种是基础设施的实际状态，第二种是基础设施的预期状态。

调解器模式将迫使工程师以两个独立的途径忘记这些表示，以及实现一个解决方案，以协调实际状态达到预期状态。

协调模式可以被认为是一套四种方法和四种哲学规则：

1. 所有的输入和输出都使用数据结构。
2. 确保数据结构是不可变的。
3. 保持资源映射简单。
4. 使实际状态符合预期状态。

这些模式的消费者可以依靠这些强大的保证。此外，他们将消费者从实施细节中解放出来。

规则1：为所有输入和输出使用数据结构

实现调解器模式的方法只能接受和返回数据结构。结构必须在调解器实现的上下文之外定义，但实现必须知道它。

通过仅接受用于输入的数据结构并将其作为输出返回，消费者可以协调其数据存储中定义的任何结构，而不必担心该协调如何发生。这也允许在运行时或者程序的不同版本中改变、修改或切换实现。

尽管我们希望尽可能经常遵守第一条规则，但是永远不要将数据结构和代码库紧密结合也非常重要。始终遵守最佳的抽象和分离实践，绝不使用API的子集来传递函数或类。

规则2：确保数据结构不可变

考虑像合同或担保这样的数据结构。在调解器模式的上下文中，实际和期望的结构在运行时设置在内存中。这保证了在调解之前结构是准确的。在协调基础设施的过程中，如果结构发生变化，则必须创建一个具有相同保证的新结构。明智的基础设施应用程序将强制数据结构的不变性，即使工程师试图改变数据结构，它也不会工作，或者程序会出错（甚至可能会编译不过）。

基础设施应用程序的核心组件是将表示映射到一组资源的能力。资源是需要运行以满足基础设施要求的单个任务。这些任务中的每一个都将负责以某种方式更改基础设施。

基本示例可能是部署新虚拟机，设置新网络或配置现有虚拟机。这些工作单元中的每一个都将被称为资源。每个数据结构都应映射到一定数量的资源。应用程序负责推理结构并创建资源集。图4-1中显示了API映射到单个资源的示例。

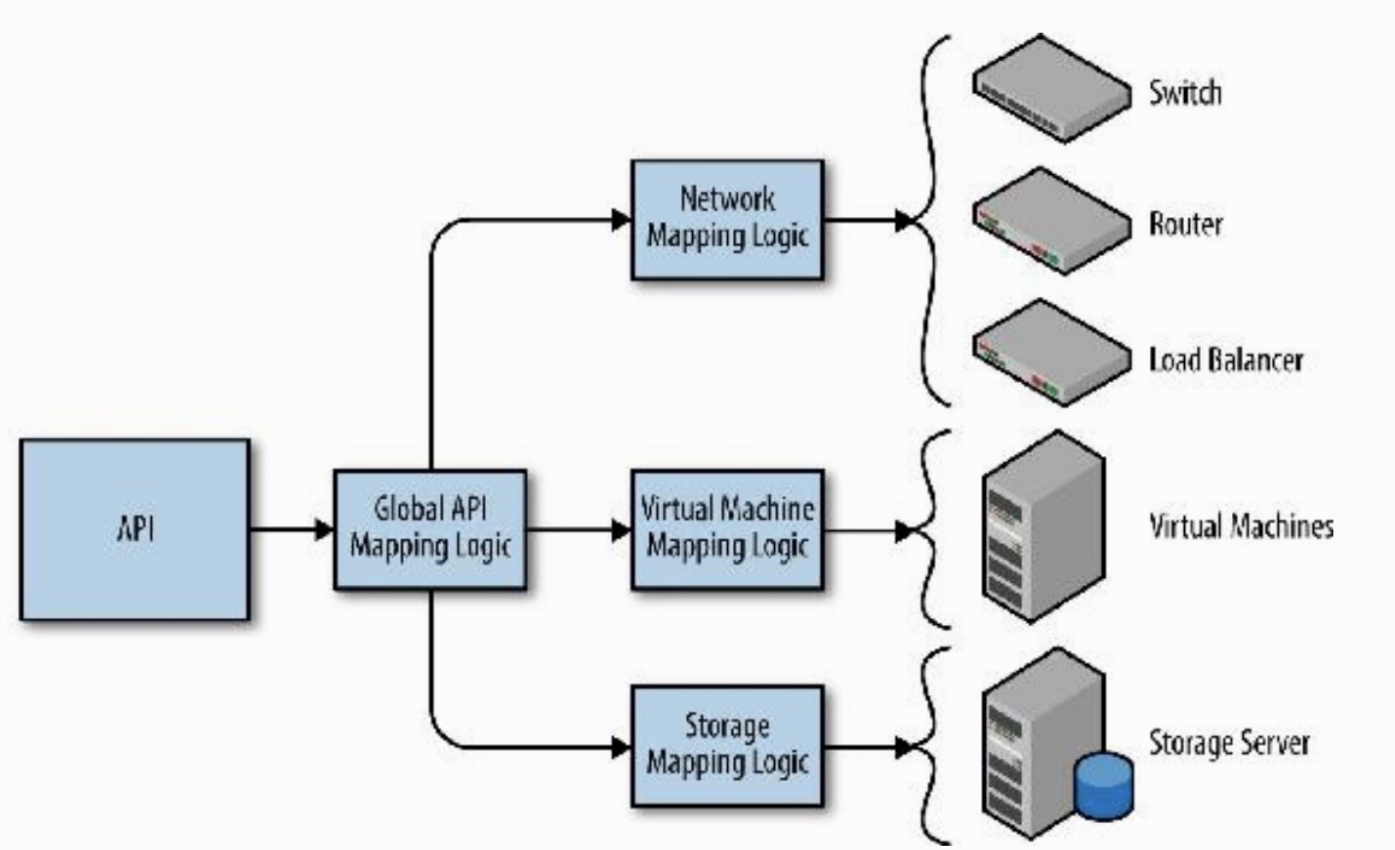

图4-1. 将结构映射到资源的图表

调解器模式演示了一种处理数据结构的稳定方法，因为它会改变资源。由于调解器模式需要比较资源状态，所以数据结构必须是不可变的。这意味着无论何时需要更新数据结构，都必须创建新的数据结构。

注意基础设施的变化。每次发生突变时，实际的数据结构都是陈旧的。一个聪明的基础设施应用程序将意识到这个问题并相应地处理它。

一种简单的解决方案是在发生突变时更新内存中的数据结构。如果每次突变都更新实际状态，则可以观察调解过程，因为实际状态会随时间经历一系列更改，直到最终匹配预期状态并且调解完成。

规则3：保持资源映射简单

在调解者的幕后，这个模式就是一个实现。一个实现只是一组代码，具有创建，修改和删除基础结构的方法。一个程序可能有很多实现。

每个实现最终都需要将数据结构映射到一组资源。这组资源需要按逻辑方式组合在一起，以便程序可以推断每个资源。

除了创建资源的基本模型之外，您必须非常注意每个资源的依赖关系。许多资源依赖于其他资源，这意味着许多基础设施都依赖于其他部分。例如，在将虚拟机放入网络之前，网络需要存在。

调解器模式规定应该使用用于分组资源的最简单的数据结构。

解决资源映射问题是一个工程决策，每个实现都可能会发生变化。仔细挑选数据结构非常重要，因为从工程角度看，调解器需要稳定且易于理解。

映射数据的两种常见结构是集合和图形。

一组是可以迭代的资源的平面列表。在许多编程语言中，这些被称为列表、集合、数组等。

图形是通过指针链接在一起的顶点（vertex）的集合。根据编程语言，图的顶点通常是结构或类。顶点通过在顶点某处定义的指针有一个到另一个顶点的链接。图形实现可以通过指针从一个跳到另一个来访问每个顶点。

例4-5是Go编程语言中一个基本顶点的例子。

例4-5. 示例顶点

```
    // Vertex is a data structure that represents a single point on a graph. // A single Vertex can have N number
1.  of children vertices or none at all. type Vertex struct {
2.  Name string
3.      Children []*Vertex
4.  }
```

遍历图的例子可能像迭代遍历每个子元素一样简单。这种遍历有时被称为“walking the graph”。

例4-6是通过Go中写入的深度优先遍历递归访问图中每个顶点的示例。

例4-6. 深度优先遍历

```
    // recursiveWalk will recursively dig into all children, // and their children accordingly and echo the name
1.  of // the vertex currently being visited to STDOUT.
2.  func recursiveWalk(v *Vertex){
3.  fmt.Printf("Currently visiting vertex: %s\n", v.Name) for _, child := range v.Children {
4.          recursiveWalk(child)
5.      }
6.  }
```

首先，图的简单实现似乎是解决资源图的合理选择，因为可以通过逻辑方式构建图来处理依赖关系。虽然图会起作用，但它也会带来风险和复杂性。实施图来绘制资源的最大风险是在图中有周期。一个循环是当一个图的一个顶点通过一条以上的路径指向另一个顶点时，这意味着遍历该图是一个无止境的操作。

必要时可以使用图，但在大多数情况下，调解器模式应该映射一组资源，而不是图。通过使用一个集合，调解器可以程序化地遍历资源，并提供线性方法来解决映射问题。此外，撤销或删除基础设施的过程与通过反向遍历集合一样简单。

规则4：使实际状态符合预期状态

在调解器模式中提供的保证是用户获得准确的或者错误的内容。这是一个保证使用调解器的工程师可以信赖的。这一点很重要，因为消费者不必关心验证调解器突变是否是幂等性的并按预期结束。实施最终是为了解决这个问题。有了这种保证，在更复杂的操作中使用调解器模式，如控制器或operator，现在变得更加简单。

在返回调用代码之前，实现应检查新调解的实际数据结构是否与最初预期的数据结构匹配。如果没有，它应该是错误的。消费者不应该关心验证API，并且应该相信如果出现问题调解器会报错。

由于数据结构是不可变的，并且如果调解器模式不成功，API将会出错，所以我们可以高度信任API。对于复杂的系统，重要的是您能够相信您的软件可以工作或以可预测的方式失败。

调解器模式的方法

根据我们刚刚解释的调解器模式的信息和规则，让我们看看这些规则是如何实现的。我们将通过查看实现调解器模式的应用程序所需的方法来执行此操作。

调解器模式的第一种方法是 `GetActual()` 。这种方法有时称为审计，用于查询基础设施的实际状态。该方法通过生成资源映射，然后程序地调用每个资源以查看存在什么（如果有的话）。该方法将根据查询更新数据结构，并返回表示实际正在运行的以填充数据结构。

一个更简单的方法 `GetExpected()` 将从数据存储中读取对象的预期状态。在 `infrastructure.yaml` 示例（例4-4）中， `GetExpected()` 将简单地解组这个YAML并将其以内存中的数据结构的形式返回。在这一步没有进行资源审计。

最令人兴奋的方法是 `Reconcile()` 方法，其中调解器实现将获得对象的实际状态和预期状态。

这是调解器模式的意图驱动行为的核心。底层调解器实现将使用在 `GetActual()` 中使用的相同资源映射逻辑来定义一组资源。然后协调执行将对这些资源进行操作，独立协调每一个资源。

了解每个资源调解步骤的复杂性非常重要。调解器实现必须以两种方式工作。

首先，从所需和实际状态获取资源属性。接下来，将更改应用到最小的一组属性，以使实际状态与所需的状态匹配。

只要这两个基础设施的表示有冲突，调解器执行必须采取行动并改变基础设施。协调步骤完成后，调解器实施必须创建一个新的表示，然后转到下一个资源。在所有资源调和后，调解器实现将新的数据结构返回给接口的调用者。现在这个新的数据结构准确地代表了对象的实际状态，并应该保证它与原始的实际数据结构相匹配。

调解器模式的最后一个方法是 `Destroy()` 方法。 `wordDestroy()` 是故意选择在 `Delete()` 上的，因为我们希望工程师意识到该方法应该销毁基础设施，并且从不禁用它。 `Destroy()` 方法的实现很简单。它使用与前面实现方法中定义的资源映射相同的资源映射，但仅对资源进行反向操作。

Go中的模式示例

例4-7是Go编程语言中定义的调解器模式的四种方法。

如果你不知道Go，别担心。该模式可以很容易地用任何语言实现。我们只使用Go，因为它清楚地定义了每种方法的输入和输出类型。请阅读每种方法的注释，因为它定义了每种方法需要做什么以及何时应该使用。

例4-7. 调解器模式接口

```
// The reconciler interface below is an example of the reconciler pattern.
// It should be used whenever a user intends on mutating infrastructure based on a // state that might have
changed over time.
type Reconciler interface {
    // GetActual takes no arguments for input and returns a populated data
    // structure as well as a possible error. The data structure should
    // contain a complete representation of the infrastructure.
    // This is sometimes called an audit. This method
    // should be used to get a real-time representation of what infrastructure is
    // in existence.
GetActual() (*Api, error)
    // GetExpected takes no arguments for input and returns a populated data
```

```
    // structure that represents what infrastructure an operator has declared to
    // exist, as well as a possible error. This is sometimes called expected or
    // intended state. This method should be used to get a real-time representation
    // of what infrastructure an operator intends to be in existence.
GetExpected() (*Api, error)
    // Reconcile takes two arguments.
    // actualApi is a populated data structure that is returned from the GetActual
    // method. expectedApi is a populated data structure that is returned from the
    // GetExpected method. Reconcile will return a populated data structure that is
    // a representation of the new "actual" state, as well as a possible error.
    // By definition, the data structure returned here should match
    // the data structure returned from the GetExpected method. This method is
    // responsible for making changes to infrastructure.
    Reconcile(actualApi, expectedApi *Api) (*Api, error)
    // Destroy takes one argument.
    // actualApi is a populated data structure that is returned from the GetActual
    // method. Destroy will return a populated data structure that is a
    // representation of the new "actual" state, as well as a possible error. By
    // definition, the data structure returned here should match
    // the data structure returned from the GetExpected method.
    Destroy(actualApi *Api) (*Api, error)
}
```

审计关系

随着时间的推移，我们基础设施的最后一次审计变得陈旧，增加了我们对基础设施的表示不准确的风险。因此，折衷的办法是运维人员可以调整审计频率以确定基础设施表示的准确性。

调解是隐式的审计。如果没有任何变化，调解器就什么也不用做，这一步就成为了审计，验证我们对基础设施的表示是否准确。

此外，如果在我们的基础设施中碰巧发生了一些变化，调解器将检测到这一变化并尝试纠正它。在完成调解后，基础设施的状态将保证准确。因此，隐含地，我们再次审计了基础设施。

配置管理中的审计和调解器模式

基础设施工程师可能熟悉来自配置管理工具的调解器模式，这些工具使用类似的方法来改变操作系统。配置管理工具通过一组资源来管理工程师定义的一组清单或配方。

该工具将对系统采取行动以确保实际状态和所需状态匹配。如果没有更改，则执行简单审计以确保状态匹配。

配置管理与云原生基础设施应用程序不同的原因是，配置管理传统上是抽象的单节点，并且不会创建或管理基础设施资源。

一些配置管理工具正在将其在这个领域的使用扩展到一定程度的成功，但它们仍然属于基础设施类的代码范畴，而不是软件提供的基础设施的双向关系。

轻量级和稳定的调解器实施可以产生强大的结果，并快速协调，从而为运维人员提供准确的基础设施表示的信心。

在控制器中使用调解器模式

编排管理工具（如Kubernetes）为我们提供了一个可以方便地运行应用程序的平台。控制器是为预期状态提供控制回路。Kubernetes建立在这个基础之上。调解器模式可以很容易地审计和协调由Kubernetes控制的对象。

想象一下在以下步骤中循环将无休止地流经调解器模式：

1. 调用 `GetExpected()` 并从数据存储中读取基础结构的预期状态。
2. 调用 `GetActual()` 并从环境中读取以获取基础结构的实际状态。
3. 调用 `Reconcile()` 并调和状态。

以这种方式实施调解器模式的程序将用作控制器。由于很容易看出控制器本身的程序必须有多小巧，因此该方案的优雅显而易见。

此外，改变基础设施就像改变状态存储一样简单。控制器将在下次调用 `GetExpected()` 时读取更改并触发协调。负责基础设施的运维人员可以放心，稳定可靠的循环在后台安静地运行，在基础设施环境中执行它的意愿。现在，运维人员通过管理应用来管理基础设施。

> 控制回路的目标搜寻行为非常稳定。Kubernetes已经证明了这一点，我们曾经发现过一些没有被注意到的错误，因为控制回路基本上是稳定的，并且会随着时间的推移而自行修正。
>
> 如果您被边缘（edge）触发，则会冒着损害您的状态的风险，并且永远无法重新创建状态。如果你是水平（level）触发的模式是非常宽容的，并允许组件在一定空间内纠正。这使得Kubernetes工作得很好。
>
> —Joe Beda，Heptio公司首席技术官

销毁基础设施现在就像通知控制器我们希望销毁基础设施一样简单。这可以通过多种方式完成。一种方法是让控制器尊重禁用的状态文件。这可以通过从开启一个比特位反转来表示。

另一种方式可能是删除状态的内容。无论运维人员如何选择发送 `Destroy()` 信号，控制器都准备好调用 `convenienceDestroy()` 方法。

本章小结

基础设施工程师也是软件工程师，负责构建先进的高度分布式系统，在后台开发。他们必须编写管理他们负责的基础设施的软件。

虽然这两个学科之间有许多相似之处，但基础设施管理应用程序的工程需要终身学习。诸如引导基础设施之类的难题不断发展，需要工程师不断学习新事物。还需要维护和优化基础设施，这一定会让工程师长期受雇。

本章为用户提供了强大的模式和基础知识，将不明确的API结构映射为粒度资源。这些资源可以应用到您的本地数据中心、私有云或公有云中。

了解这些模式的工作原理对于构建可靠的基础设施管理应用程序至关重要。本章阐述的模式旨在为工程师提供构建声明式基础设施管理应用程序的起点和灵感。

在构建基础设施管理应用程序时，没有正确或错误的答案，只要应用程序遵循Unix哲学：“做一件事并把它做得很好。“

第5章 开发基础设施应用程序

在构建应用程序以管理基础设施时，我们要将需要公开的API与要创建的应用程序等量看待。这些API将代表您的基础设施的抽象，而应用程序将使用API消费这些这些基础设施。

务必牢牢掌握两者的重要性，以及如何利用它们来创建可扩展的弹性基础设施。

在本章中，我们将举一个云原生应用程序和API的虚构示例，这些应用程序和API会经历正常的应用程序周期。如果您想了解更多有关管理云原生应用程序的信息，请参阅第7章。

设计API

> 这里的术语API是指处理数据结构中的基础设施表示，而不关心如何暴露或消费这些API。通常使用HTTP RESTful端点来传递数据结构，但如何实现对本章并不重要。

随着基础设施的不断发展，运行在基础设施之上的应用程序也要随之演变。为这些应用程序设置的功能将随着时间而改变，因此基础设施是隐性地演变。随着基础设施的不断发展，管理它的应用程序也必须发展。

基础设施的功能、需求和新进展将永无止境。如果我们幸运的话，云提供商API将会保持稳定并且不会频繁更改。作为基础设施工程师，我们需要做好准备，以适应这些需求。我们需要准备好发展我们的基础设施和支持它的应用程序。

我们必须创建可缩放的应用程序，并准备对其进行扩展。为了做到这一点，我们需要了解在不破坏应用程序现有流程的情况下对应用程序进行大量更改的细微差别。

管理基础设施的工程应用程序的美妙之处在于它将运维人员从其他人的意见中解放出来。

应用程序中使用的抽象现在由工程师来完成。如果一个API需要更多的文字，它可以有；或者如果它需要被自以为是，并且被大量抽象出来，那可以有。字面和抽象定义的强大组合可以为运维人员准确地提供他们想要和需要管理基础设施的内容。

添加功能

根据功能的性质，向基础设施应用程序添加功能可能非常简单也可能非常复杂。添加功能的目标是我们应该能够添加新功能而不会危害现有功能。我们绝不希望引入会以给系统其他组件带来负面影响的功能。此外，我们一直希望确保系统输入在合理的时间内保持有效。

例5-1是本书前面介绍的基础设施API演化的具体示例。我们称之为API v1的第一个版本。

例5-1. v1.json

```
{
    "virtualMachines": [{
        "name": "my-vm",
        "size": "large",
        "localIp": "10.0.0.111",
        "subnet": "my-subnet"
    }],
```

```
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

想象一下，我们希望实现一项功能，允许基础设施运维人员为虚拟机定义DNS记录。新的API看起来略有不同。在例5-2中，我们将定义一个名为version的顶级指令，这会让我们的应用程序知道这是API的v2版本。我们还将添加一个新的块，用于在虚拟机块的上下文中定义DNS记录。这是v1中不支持的新指令。

例5-2. v2.json

```
{
    "version": "2",
    "virtualMachines": [{
        "name": "my-vm",
        "size": "large",
        "localIp": "10.0.0.111",
        "subnet": "my-subnet",
        "dnsRecords": [{
            "type": "A",
            "ttl": 60,
            "value": "my-vm.example.com"
        }]
    }],
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

这两个对象都是有效的，应用程序应该继续支持它们。应用程序应检测到v2对象是否打算使用内置于应用程序中的新DNS功能。该应用程序应该足够聪明，以适当地导航新功能。将资源应用于云时，新的v2对象的资源集将与第一个v1对象相同，但添加了单个DNS资源。

这引入了一个有趣的问题：应用程序应该如何处理老的API对象？应用程序仍应在云中创建资源，但可以支持无DNS的虚拟机。

随着时间的推移，运维人员可以修改现有虚拟机对象以使用新的DNS功能。应用程序自然会检测到增量并为新功能创建DNS记录。

弃用功能

让我们快速转到下一个API版本v3。在这种情况下，我们的API已经发展，当前表示IP地址的方式陷入了僵局。

在API v1的第一个版本中，我们能够通过本地IP指令方便地为网络接口声明一个本地IP地址。我们的任务是为虚拟机提供多种网络接口。需要注意的是，这将与最初的v1 API相冲突。

让我们来看一下示例5-3中新的v3版本的API。

例5-3. v3.json

```
{
    "version": "2",
    "virtualMachines": [{
        "name": "my-vm",
        "size": "large",
        "networkInterfaces": [{
            "type": "local",
            "ip": "10.0.0.11"
        }],
        "subnet": "my-subnet",
        "dnsRecords": [{
            "type": "A",
            "ttl": 60,
            "value": "my-vm.example.com"
        }]
    }],
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

使用定义多个网络接口所需的新数据结构，我们已弃用本地IP指令。但是我们并没有删除定义IP地址的概念，我们只是简单地重组了它。这意味着我们可以开始在两个阶段废弃该指令。首先警告，然后是拒绝。

在警告阶段，我们的应用程序可能会输出一个关于不再支持本地IP指令的警告。应用程序可以接受在对象中定义的指令，并将旧的API版本v2转换为用户的新API版本v3。

转换将采用为本地IP定义的值，并在新网络接口指令中创建与初始值相匹配的单个块。应用程序可以继续处理API对象，就好像用户发送了v3对象而不是v2对象一样。预计用户会注意到该指令已被弃用，并及时更新其表示。

在拒绝阶段，我们的应用程序将彻底拒绝v2 API。用户将被迫更新他们的API到更新的版本，或者甘愿在他们的基础设施中冒此风险。

> 弃用是非常危险的
>
> 这是一个极其危险的过程，成功导航可能会非常困难。拒绝输入必须出于很好的理由。
>
> 如果输入信息会在应用程序中破坏保证，则应拒绝该信息。否则，通常的最佳实践是警告并继续。
>
> 破坏用户的输入很容易运维人员感到不安和沮丧。

对API进行版本控制的基础设施工程师必须对在何时弃用哪些功能做出最佳判断。此外，工程师需要花时间提出巧妙的解决方案，可以是警告或转换。在某些情况下，能够做到悄无声息的转换对不断发展的云原生基础设施来说是一个巨大的胜利。

突变基础设施

基础设施需要随着时间的推移而变化。这是云原生环境的本质。不仅应用程序频繁部署，而且运行基础设施的云提供

商也在不断变化。

基础设施的变化可以有多种形式，比如扩大或缩小基础设施，复制整个环境或消耗新资源。

当运维人员承担变更基础设施的任务时，我们可以看到API的真实价值。假设我们想要扩展环境中的虚拟机数量。不需要更改API版本，但对基础设施的表示做一些小的调整将很快反映出变化。就这么简单。

然而，重要的是要记住，在这种情况下，运维可能是一个人，或者很可能是另一个软件。

请记住，我们故意将我们的API构造成易于被计算机解码。我们可以在API的两侧使用该软件！

> 使用**Operator**消费和生产**API**
>
> Operter—构建云原生产品和平台的公司CoreOS创造了这个术语，即Kubernetes控制器，取代了人类参与管理特定应用的需求。他们通过协调预期的状态，以及设定预期的状态来做到这一点。
>
> CoreOS在他们的博客文章中这样描述Operator:
>
> Operator是特定应用程序的控制器，它代表Kubernetes用户扩Kubernetes API以创建、配置和管理复杂有状态应用程序的实例。它建立在基本的Kubernetes资源和控制器概念的基础上，但包含一个域或特定于应用程序的知识以实现常见任务的自动化。
>
> 该模式规定Operator可以通过给定声明性指令集来更改环境。Operator是工程师应该创建的用于管理其基础设施的云原生应用程序类型的完美示例。

设想一个简单的情景—自动调节器（autoscaler）。假设我们有一个非常简单的软件，可以检查环境中虚拟机上的平均负载。我们可以定义一个规则，只要平均负载平均值高于0.7，我们就需要创建更多的虚拟机来均匀地分配我们的负载。

Operator的规则会随着负载平均值的增加而跳闸，最终Operator需要用另一台虚拟机更新基础设施API。这样可以扩大我们的基础设施，但同样容易，我们可以定义另一个规则以在负载平均降至0.2以下。请注意，Operator这个术语在这里应该是一个应用程序，而不是一个人。

这是自动缩放的一个非常原始的例子，但是模式清楚地表明软件现在可以开始扮演人类运维人员的角色。

有许多工具可以帮助扩展如Kubernetes、Nomad和Mesos等基础设施上的应用程序负载。这假定应用程序层正在运行一个编排调度器上，它将为我们管理这个。

为了进一步把基础设施API的价值最大化，想象一下，如果多个基础设施管理应用程序使用相同的API。这是一个非常强大的基础设施演进模式。

我们来看看相同的API—记住它只有几千字节的数据，并且在两个独立的基础设施管理应用程序运行。图5-1显示了两个基础设施应用程序如何从相同的API获取数据但将基础设施部署到两个独立的云环境的示例。

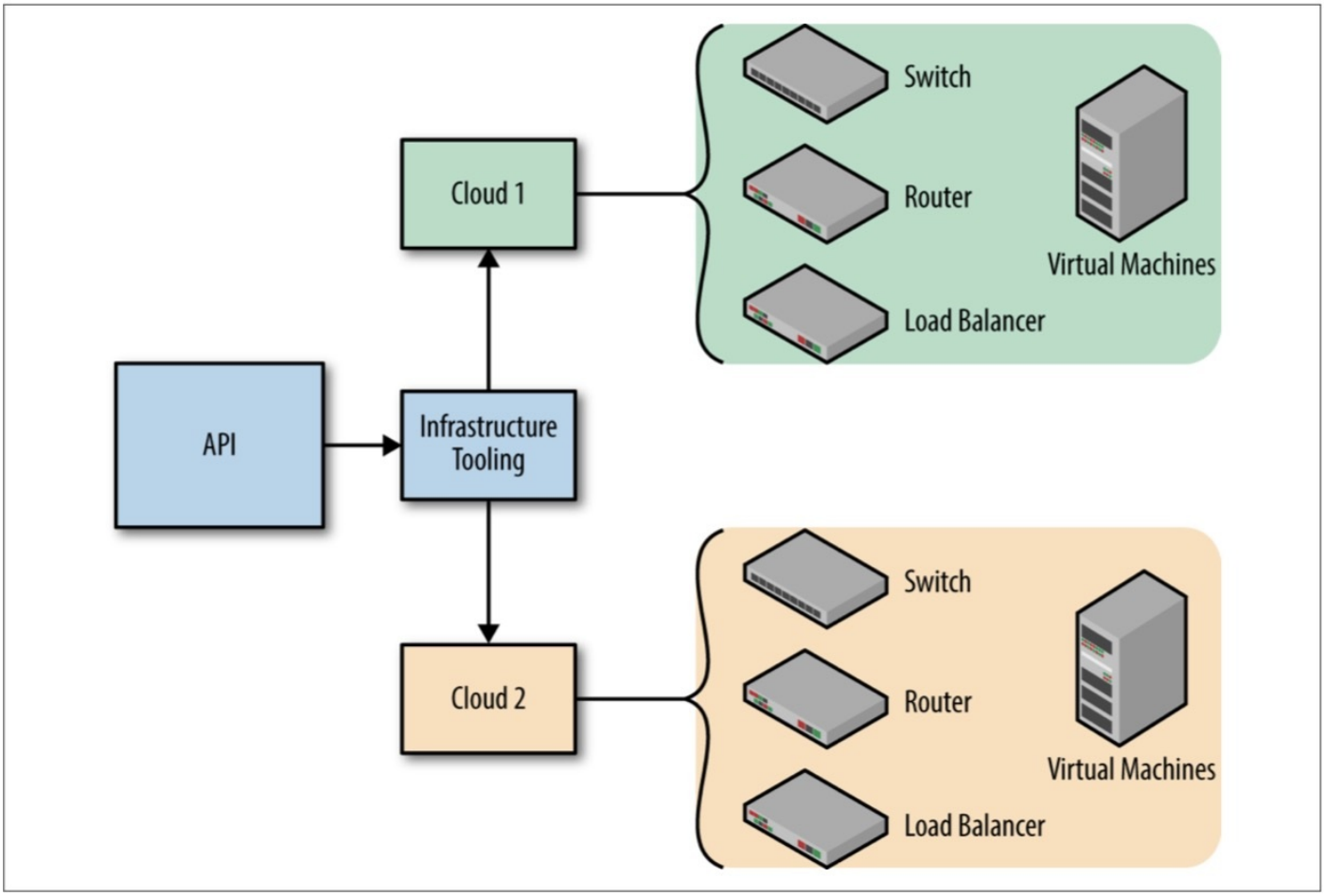

图5-1. 一个API被部署在两个云中

该模型为基础设施工程师提供了能够为多个云提供商提供通用抽象的强大功能。现在我们可以看到确保API的应用程序如何在多个地方代表基础设施。如果基础设施API负责提供自己的抽象和资源调配，则基础设施不必与单个云提供商的抽象相关联。用户可以在他们选择的云中创建独特的基础设施排列。

维护云提供商兼容性

虽然保持API与云提供商的兼容性将会有很多工作要做，但在对于部署工作流程和供应流程时，却很少需要改变。请记住，人类比技术更难改变。如果您可以为人类保持一致的环境，它将抵消所需的技术开销。

您还应该权衡多云兼容性的好处。如果它不是您的基础设施的需求，您可以节省大量的工程工作。考虑云厂商锁定时请参阅附录B.

我们也可以推测在同一个云中运行不同的基础设施管理应用程序。这些应用程序中可能会对API进行不同的解释，这会导致对于运维人员的意图的定义略有不同。将运维人员对基础设施的意图与应用程序之间的转换可能正是我们所需要的。图5-2显示了两个应用程序正在读取相同的API源，但根据环境和需要不同地实施数据。

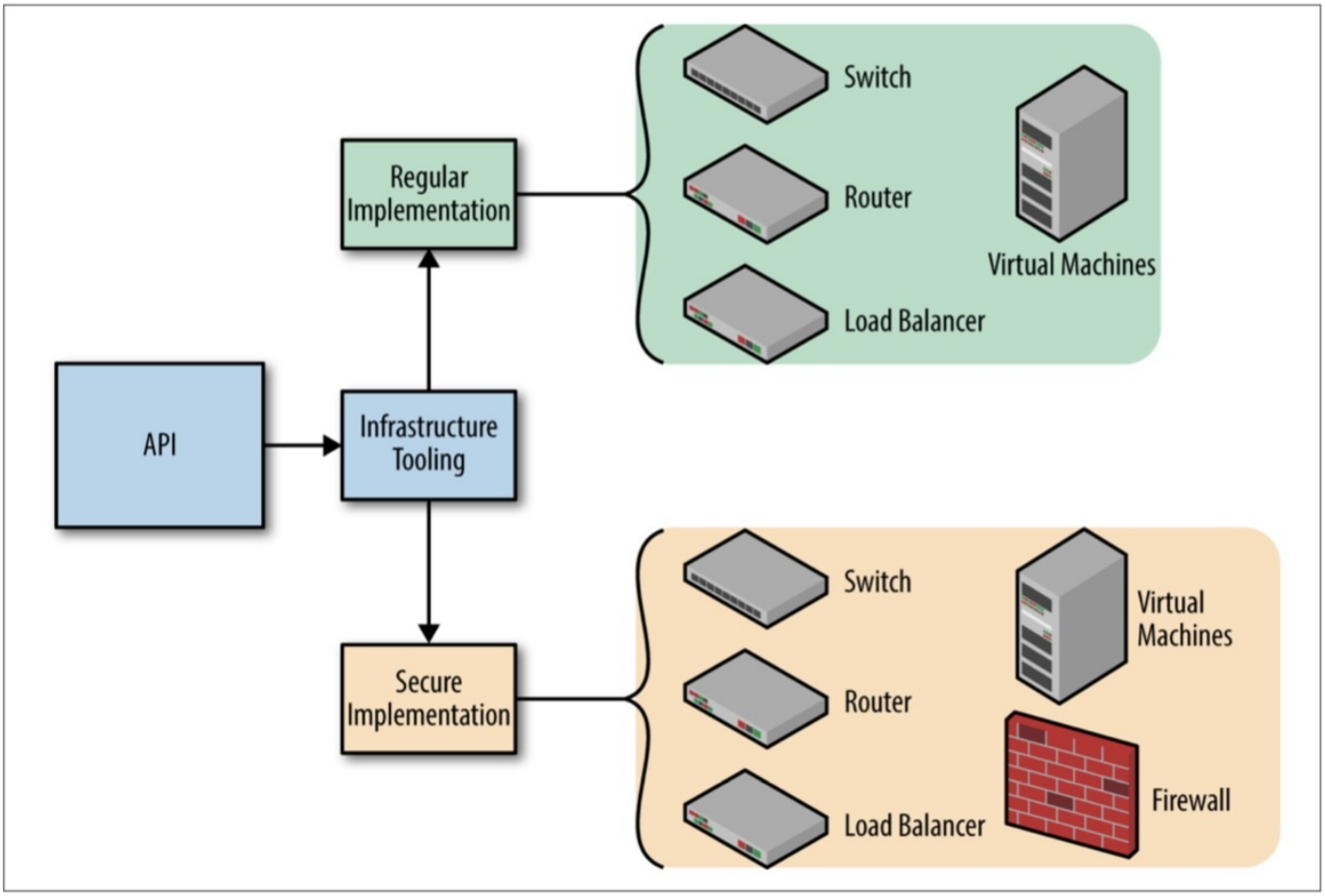

图5-2. 一个API以不同的方式部署在同一个云中

结论

与基础设施API相比，基础设施应用的排列组合是无止境的。这为基础设施工程师提供了一个非常灵活和可扩展的解决方案，希望能够以不同的环境和方式掌握基础设施。

我们为了满足基础设施要求而可能建立的各种应用现在已经成为基础设施本身的代表。这是第3章定义的软件基础设施的缩影。

请务必记住，我们构建的应用程序本身就是云原生应用程序。这是一个有趣的故事，因为我们正在构建云原生应用程序来管理云原生基础设施。

第6章 测试云原生基础设施

基础设施是用来支撑应用程序的。可信任的软件对于工程成功至关重要。如果每次在终端输入ls命令，都会发生随机动作，那么你将永远也不会相信ls，而是去找另一种方式来列出目录中的文件。

我们需要能够信任我们的基础设施。本章旨在开放信任和验证基础设施的意识形态。我们将描述的实践旨在增加对应用程序和基础设施工程的信心。

软件测试在当今的软件工程领域非常普遍。然而，如何测试基础设施的还没很明确的最佳实践。

这意味着在本书中的所有章节中，这一节应该是最令人兴奋的！像您这样的工程师有空间该领域发挥出色的影响力。

软件测试是一种证明软件可以正常工作的有效做法，软件不会失败，并且在各种特殊情况下仍然有效。因此，如果我们将相同的范例应用于基础设施测试，测试目标如下：

1. 证明基础设施按预期运行。
2. 证明基础设施不会失败。
3. 证明这两种情况在各种边缘情况下都是正确的。

衡量基础设施是否有效需要我们先定义什么叫有效。现在，您应该对使用基础设施和工程代表应用程序的想法感到满意。

定义基础设施API的人应该花时间构思一个可以创建有效基础设施的理智的API。例如，如果创建了一个定义虚拟机的API但没有使之可以运行的网络信息，这种做法就很愚蠢。您应该在API中创建有用的抽象，然后使用第3章和第4章中提出的想法来确保API创建正确的基础设施组件。

我们已经开始开发一个心智模型，通过定义API来对我们基础设施的健全性进行检查。这意味着我们可以翻转逻辑并想象出相反的情况，这将是除了原始心智模型中的东西之外的所有东西。

为基础设施定义基本的完整性测试的做法很值得努力。因此测试基础设施的第一步是证明您的基础设施是按照预期存在的，并且没有任何东西存在与原意相反。

在本章中，我们将探索基础设施测试，并为新的测试工具顺序奠定基础。

我们该测试什么？

在开始编写代码之前，我们必须首先确定需要测试哪些方面。

测试驱动开发是测试优先的常见做法。测试是为了证明测试的关注的方面而编写的，并且从一开始就会隐含失败。经过开发周期旨在使测试通过；也就是说，该软件是为满足每个测试中定义的要求而开发的。这是一个强大的实践，可以帮助软件保持专注，并帮助工程师对他们的软件以及测试结果产生信心。

这是一个可以以多种方式回答的哲学问题。对我们需要证明真实而非虚假的想法有一个好的想法对于建立值得信赖的基础设施是必不可少的。例如，如果存在依赖基础设施的业务问题，则应该对其进行测试。更重要的是，许多业务不仅依赖基础设施，还会在出现问题时自行修复。

确定您的基础设施需要填充的问题空间代表了需要编写的第一轮测试。

远景规划是基础设施测试的另一个重要方面，但应该谨慎。在足够的前瞻性和过度工程化之间有一条看不见的界限。如果有疑问，坚持最少量的测试逻辑。

在我们完全了解需要的测试之后，就可以考虑实施测试套件。

编写可测试代码

协调者模式的规则不仅旨在创建一个干净的基础设施应用程序，而且还旨在鼓励可测试的基础设施代码。

这意味着，在应用程序的每一个重要步骤中，我们总是会重新创建一个相同类型的新对象，也就是说基础系统的每个主要组件都会使用相同的输入和输出。使用相同的输入和输出可以更轻松地以编程方式测试软件的小型组件。测试将确保您的组件按预期工作。

然而，在编写基础设施测试代码时，还有许多其他有价值的经验值得借鉴。我们将在假设情景下看看测试基础设施的具体示例。在我们浏览场景时，您将学到测试基础设施代码的经验教训。

我们还会提出一些工程师在开始编写可测试基础设施代码时可以遵守的规则。

验证

采用非常基础的基础设施定义，如示例6-1。

例6-1. infrastructure.json

```
{
    "virtualMachines": [{
        "name": "my-vm",
        "size": "large",
        "localIp": "192.168.1.111",
        "subnet": "my-subnet"
    }],
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

这些数据的目的应该是显而易见的：确保一个名为 `my-vm` 的虚拟机的大小为 `large` ，IP地址为 `192.168.1.111` 。这些数据也暗示确保一个名为 `my-subnet` 的子网将容纳虚拟机 `my-vm` 。

希望你注意到了这个数据有什么问题。虚拟机的IP地址超出了子网的可用CIDR范围。

应用程序运行此数据应该会导致失败，因为虚拟机的网络设置无效。如果我们的应用程序的构建是为了盲目地允许部署任何数据，我们将创建可联网的基础设施。尽管我们应该编写测试以确保新的虚拟机能够在网络上进行路由，但我们还可以做更多事情来帮助强化我们的应用程序并使测试更加轻松。

在应用程序处理输入之前，我们可以首先尝试验证输入。这在软件工程中是很常见的做法。

想象一下，如果不是盲目部署这个基础设施，我们首先试会验证输入。在运行时，我们的应用程序可以容易的检测到

虚拟机的IP地址在虚拟机所连接的子网中不起作用。这将阻止输入到达我们的基础设施环境。由于知道应用程序将故意拒绝无效的基础设施表示，我们可以编写happy和sad测试来确保实现此行为。

Happy测试可以对条件进行正面处理。换句话说，它是一种向应用程序发送有效API对象并确保应用程序接受有效输入的测试。Sad测试，可以对相反的情况或负面情况进行分析。例6-1是一个sad测试的例子，它将一个无效的API对象发送给应用程序，并确保应用程序拒绝无效输入。

这种新模式使测试基础设施非常快速，而且通常不用费什么力气。一个工程师就可以开发大量的happy和sad测试，即使是最奇怪的应用程序输入也是如此。此外，测试集合可以随着时间的推移而增长；在欺骗API对象流入环境场景中时，工程师可以快速添加测试以防止再次发生。

输入验证是测试最基本的事情之一。通过在我们的应用程序中编写简单的验证来检查理智的值，我们可以开始过滤应用程序的输入。这也给了我们一个很容易定义有意义的错误并快速返回错误的途径。

验证提供信心，而不会让您等待基础设施发生变异。这为面向API开发的工程师创建了更快的反馈循环。

输入您的代码库

编写易于测试的代码非常重要。容易出错的问题可能会导致成本上升，因此需要围绕专有输入设计应用程序。专有输入是仅与程序中的一个点相关的输入，获得所需输入的唯一方法是线性执行程序。以这种方式线性编写代码对于人类大脑来说是有意义的，但这也是有效测试的最难的模式之一，特别是当涉及到测试基础设施时。

专有输入陷入困境的例子如下：

1. 函数 `DoSomething()` 的调用返回 `Something {}` 。
2. `Something {}` 传递给函数 `NextStep (Something)` 并返回 `SomethingElse {}` 。
3. `SomethingElse {}` 被传递给函数 `FinalStep (something else)` ，返回true或false。

这里的问题是，为了测试 `FinalStep()` 函数，我们首先需要遍历步骤1和2。在测试的情况下，这会引入复杂性和更多的失败点；它甚至可能不会在测试执行的环境中工作。

更优雅的解决方案是以这样一种方式构造代码，即可以在程序的其余部分使用相同的数据结构上调用最后一的 `step()`:

1. 代码初始化 `GreatSomething {}` ，它实现了方法 `great.DoSomething()` 。
2. `GreatSomething {}` 实现 `NextStep()` 方法。
3. `GreatSomething {}` 实现了 `something.FinalStep()` 方法。

从测试的角度来看，我们可以为我们希望测试的任何步骤填充 `GreatSomething {}` ，并相应地调用这些方法。这个例子中的方法现在负责处理它们扩展的对象中定义的内存。这与最后一种方法不同，在这种方法中，特殊的内存中的结构被传递到每个函数中。

这是一个更加优雅的设计，因为测试工程师可以轻松地为任何步骤合成存储器，并且只需要关注学习同一数据的一个表示。这是更加模块化的，如果它们很快被发现，我们可以回到任何故障。

当您开始编写构成您的应用程序的软件时，请记住，在传统运行时间线期间，您需要在许多点上跳转到代码库。构建你的代码以便于在任何时候轻松地输入代码库，因此在内部测试系统至关重要。在这样的情况下，你可以成为自己最好的朋友或最大的敌人。

自我意识

> 你如何衡量我，就会如何按照你说的要求去做。
>
> ——Eliyahu M. Goldratt

在编写代码和测试时注意自己的置信度。自我意识是软件工程中最重要的部分之一，也是最容易被忽视的部分之一。

测试的最终目标是增加对应用程序的信心。就基础设施领域来说，我们的目标就是增强对基础设施的信心。

据说，测试基础设施的方法没有对错之分。在应用程序中可以通过代码覆盖率和单元测试来建立信心，但是对于基础设施来说这样做可能会存在误导。

代码覆盖率是以程序化方式衡量代码可以达到预期的行为。这个度量标准可以用作原始数据点，但我们要知道即使是覆盖率达到100%的代码库仍然可能会出现极端中断，这一点至关重要。

如果你以代码覆盖率来衡量测试结果，那么工程师就会编写更容易被测试覆盖的代码，而不是编写更适应该任务的代码。Dan Ariely在他刊登于哈弗商业评论的文章“衡量标准决定一切”：

人们的行为会根据衡量指标作出相应的调整。衡量指标会促使某个人在该指标上做出优化。你想要得到什么，你要去衡量什么。

我们应该衡量的唯一指标是信心，即我们的基础设施可以按预期工作，并且我们可以证明这一点。

衡量信心几乎是不可能的。但是有些方法可以从工程师的心理和情绪中抽取有意义的数据集。

问自己以下几个问题，记录下答案：

- 我担心这行不通吗？
- 我可以肯定，这将做我认为会做的事吗？
- 如果有人更改此文件，会发生什么情况？

一个从问题中提取数据的最强大技术是比较以前的经验水平。例如，工程师可以做出如下陈述，团队的其他成员很快就会明白他想要传达的内容：

比起上个季度，这次代码发布更令人担忧。

现在，根据团队以前的经验，我们可以开始为我们的信心水平制定一套标准，从0开始表示完全没有信心，随着时间的流逝然后增加到非常自信。当我们了解了我们担心应用程序的哪些问题之后，再为了增加信心而制定测试内容就很简单了。

测试类型

了解测试的类型以及测试方式将有助于工程师增加其对基础设施应用程序的信心。这些测试不需要编写，而且没有正确或错误之分。唯一的问题是我们相信应用程序会做我们想要它做的事情。

基础设施断言

在软件工程中，有一个重要的概念是断言，这是一种强制的方式——完全确定条件是否成立。目前已经有许多成功的框

架使用断言来测试软件。断言是一个微小的函数，它将测试条件是否为真。这些功能可以在各种测试场景中使用，以证明概念正在发挥作用和增加我们的信心。

在本章的其余部分中，我们将提到基础设施断言。您需要对这些断言的内容以及他们希望完成的内容有基本的了解。您还需要对Go语言有基本的了解，才能充分认识这些断言正在做什么。

在基础设施领域需要声明我们的基础设施有效。构建这些断言功能的库对于您的项目来说是一个值得的练习。开源社区也可以从这个工具包测试基础设施中受益。

例6-2显示了Go语言中的断言模式。假设我们想测试虚拟机是否可以解析公共主机名，然后路由到它们。

例6-2. assertNetwork.go

```
type VirtualMachine struct { localIp string
}
func (v *VirtualMachine) AssertResolvesHostname( hostname string,
expectedIp string,
message string) error { // Logic to query DNS for a hostname,
    // and compare to the expectedIp
return nil
}
func (v *VirtualMachine) AssertRouteable( hostname string,r
port int,
message string) error {
// Logic to verify the virtualMachine can route
    // to a hostname on a specific port
return nil
}
func (v *VirtualMachine) Connect() error {
// Logic to connect to the virtual machine to run the assertions return nil
}
func (v *VirtualMachine) Close() error {
// Logic to close the connection to the virtual machine return nil
}
```

在这个例子中，我们将两个断言作为VirtualMachine {}结构体上的方法来存储。方法签名是我们将在此演示中关注的内容。

第一种方法AssertResolvesHostname()演示了一种将用于检查给定主机名是否解析为预期IP地址的方法。第二种方法AssertRouteable()演示了一种用于检查给定主机名是否可在特定端口上路由的方法。

注意VirtualMachine {}结构体是如何定义成员本地IP的。另请注意，VirtualMachine {}结构体具有Connect()函数以及Close()函数。这是因为断言框架可以在虚拟机的上下文中运行这个断言。测试可以在基础设施环境之外的系统上运行，然后连接到环境中的虚拟机以运行基础设施断言。

在例6-3中，我们演示了工程师该如何在本地系统上编写Go测试。

例6-3. network_test.go

```
func TestVm(t *testing.T) {vm := VirtualMachine{
```

```
    localIp: "10.0.0.17",

if err := vm.Connect(); err != nil {
    t.Fatalf("Unable to connect to VM: %v", err)
}

defer vm.Close()

if err := vm.AssertResolvesHostname("google.com", "*",

"google.com should resolve to any IP"); err != nil {t.Fatalf("Unable to resolve hostname: %v", err)

}

if err := vm.AssertRouteable("google.com", 443,

"google.com should be routable on port 443"); err != nil {t.Fatalf("Unable to route to hostname: %v", err)

}}
```

该示例使用Go语言中的内置测试标准，这意味着该函数将作为应用程序中Go测试的正常运行测试的一部分执行。测试框架将测试名称以 `_test.go` 结尾的所有文件，并使用以 `TestXxx` 开头的签名名称测试所有函数。该框架还将 `*test.T` 指针传递给以这种方式定义的每个函数。

这个简单的测试将使用我们之前定义的断言库来完成以下几个步骤：

1. 尝试连接到应在10.0.0.17上可访问的虚拟机。
2. 在虚拟机上尝试断言虚拟机可以解析google.com并且可返回一些IP地址。
3. 在虚拟机上尝试声明虚拟机可以通过端口443路由到google.com。
4. 关闭与虚拟机的连接。

这是一个非常强大的程序。它为我们的基础设施按预期工作建立了信心。它还引入了一个优雅的脚手架，供工程师定义测试，而不必担心它们将如何运行。

开源社区迫切需要这样的基础设施测试框架。基础设施测试的标准化和可靠方法将成为开发者工具箱中的有益补充。

集成（integration）测试

集成测试也被称为端到端（e2e）测试。这些是长期运行的测试。按照预生产的方式来运行系统，这些证明可靠性和增加信心的最有价值的测试。

编写集成测试套件可能很有趣，也很有意义。在集成测试基础设施管理应用程序的情况下，测试将执行基础设施生命周期的大扫除。

线性集成测试套件的一个简单例子如下：

1. 定义一个常用的基础设施API。
2. 将数据保存到应用程序的数据存储区。
3. 运行该应用程序并创建基础设施。
4. 针对基础设施运行一系列断言。

5. 从应用程序的数据存储中删除API数据。
6. 确保基础设施已成功销毁。

在此过程中的任何一步，测试都可能失败，并且测试套件应该清理发生变异的基础设施。这是测试可以按照预期销毁基础设施重要的原因之一。

测试使我们相信，该应用程序将创建并销毁预期的基础设施，并按预期工作。随着时间的推移，我们可以增加步骤4中运行的断言的数量，并继续强化套件。

集成测试工具可能是我们测试基础设施最强大的环境。没有集成测试工具，运行像单元测试这样的小测试有多大价值。

单元（unit）测试

单元测试是测试系统并单独运行其组件的基本部分。单元测试的责任是小而谨慎。单元测试是软件工程中的常见做法，因此将成为基础设施工程的一部分。

在编写基础设施测试的情况下，测试系统的一个组件是困难的。基础设施的大多数组件都建立在彼此的基础之上。相应地测试软件通常需要改变基础设施来测试并查看其是否工作。这个过程通常涉及大部分系统。

但这并不意味着为基础设施管理系统编写单元测试是不可能的。事实上，前面例子中定义的大部分断言在技术上将都是单元测试！单元测试只测试一个小组件，但在大型集成测试系统环境中使用时，它们可能非常有用。

在测试基础设施时鼓励进行单元测试，但请记住，它们运行的上下文通常需要相当大的开销。这种开销通常以集成测试的形式出现。将单元测试的小而谨慎的检查与更大的整体测试模式相结合，使基础设施工程师对其基础设施按照预期工作具有高度的信心。

模拟（Mock）测试

在软件工程中，综合系统的常见做法是模拟测试。在模拟测试中，工程师编写或使用旨在欺骗或伪造系统的软件。

一个简单的例子就是使用一个旨在与API通信并以“mock”模式运行的SDK。SDK不会将任何数据发送到API，而是合成SDK认为API在各种情况下应该执行的操作。

确保模拟软件准确地反映它正在合成的系统的责任在于开发模拟软件的工程师手中。在某些情况下，模拟软件也是由开发其正在模拟的系统的工程师开发的。

尽管可能有一些模拟工具保持最新并且比其他工具更稳定，但使用模拟系统合成您计划测试的基础设施时存在一个普遍的道理：虚假系统只会给您带来虚假信心。

现在，这条规则可能看起来很苛刻。但它的目的是鼓励工程师不要轻易走出去，并通过构建真正的集成套件来运行测试的实践。虽然模拟系统功能强大，但将它作为基础设施测试的核心（因此也是您的信心）是非常危险的。

大多数公有云提供商对其资源实施配额限制。想象一下与一个对资源有严格限制的系统进行交互的测试。模拟系统可能会尽最大努力限制资源，但是如果不在运行时审核实际系统，模拟系统将无法确定您的基础设施是否实际部署。在这种情况下，您的模拟测试会成功。但是，当代码在真实环境中运行时，它会中断。

这只是许多实例中的一个例子，这些实例证明了为什么变异实际基础设施和发送实际网络数据包比使用模拟系统更可靠。请记住，测试的目标是增强您的基础设施在真实环境中按照预期工作的信心。

这并不是说所有的模拟测试都不好。了解模拟正在测试的基础设施与为了方便而模拟另一部分系统之间的差异非常重要。

工程师需要决定什么时候适合使用模拟系统。我们只是告诫工程师不要对这些系统有太大的信心。

混沌（chaos）测试

混沌测试可能是我们将在本书中介绍的测试基础设施中最令人兴奋的方法。它正在进行测试，以证明在基础设施中发生不可预知的事件，而不会影响基础设施的稳定性。我们通过故意破坏基础设施并衡量系统如何应对灾难来做此演示。与我们所有的测试一样，我们将以基础设施工程师的身份来应对这个问题。

我们将编写旨在以意想不到的方式打破生产系统的软件。建立对系统的信心的一部分是理解他们如何以及为什么会破坏。

> **Google如何建立信心**
>
> 学习如何破解系统的一个例子可以在谷歌的DiRT（灾难恢复培训）计划中看到。该计划旨在帮助Google的SRE熟悉他们所支持的系统。在站点可靠性工程中，他们解释说，DiRT计划的目的是因为“长时间与生产脱节可能会导致信心问题，不管是过于自信还是不自信，该计划仅仅是为了发现当事件发生时的知识差距”。

不幸的是，如果没有系统来衡量影响并从灾难中恢复过来，就不会让工程团队感觉释然。再次，我们将要求运行本章前面定义的基础结构断言。微小的单一责任功能为测量系统随时间的稳定性提供了绝佳的数据点。

测量混乱

我们再来看一下例6-3中的 `AssertRouteable()` 函数。想象一下，我们有一个服务，将连接到虚拟机，并尝试保持连接打开。服务每秒都会调用 `AssertRouteable()` 函数并记录结果。来自此服务的数据是虚拟机在其网络上路由的能力的准确表示。只要虚拟机可以路由，数据就会在图形上产生一条直线，如图6-1所示。

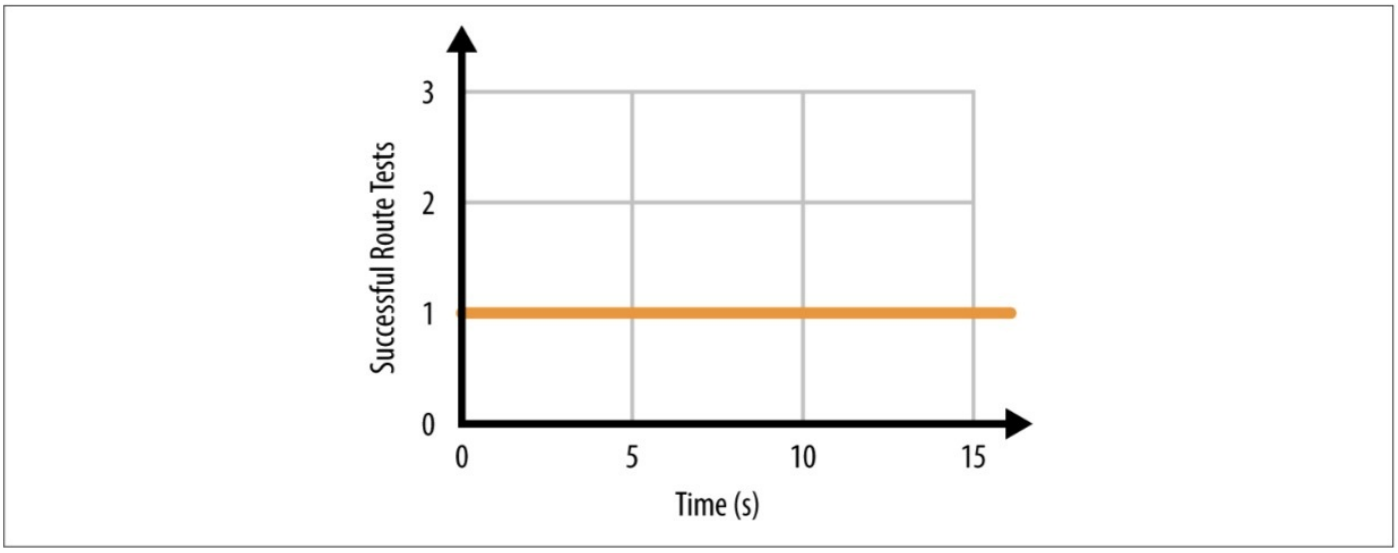

图6-1. 随着时间推移的AssertRoutable测试图

如果在任何时候连接断开或者虚拟机不再能够路由，那么图形数据会发生变化，并且我们会看到图形上的线条发生变化。随着基础设施自行修复，线路开启该图将再次稳定下来，如图6-2所示。

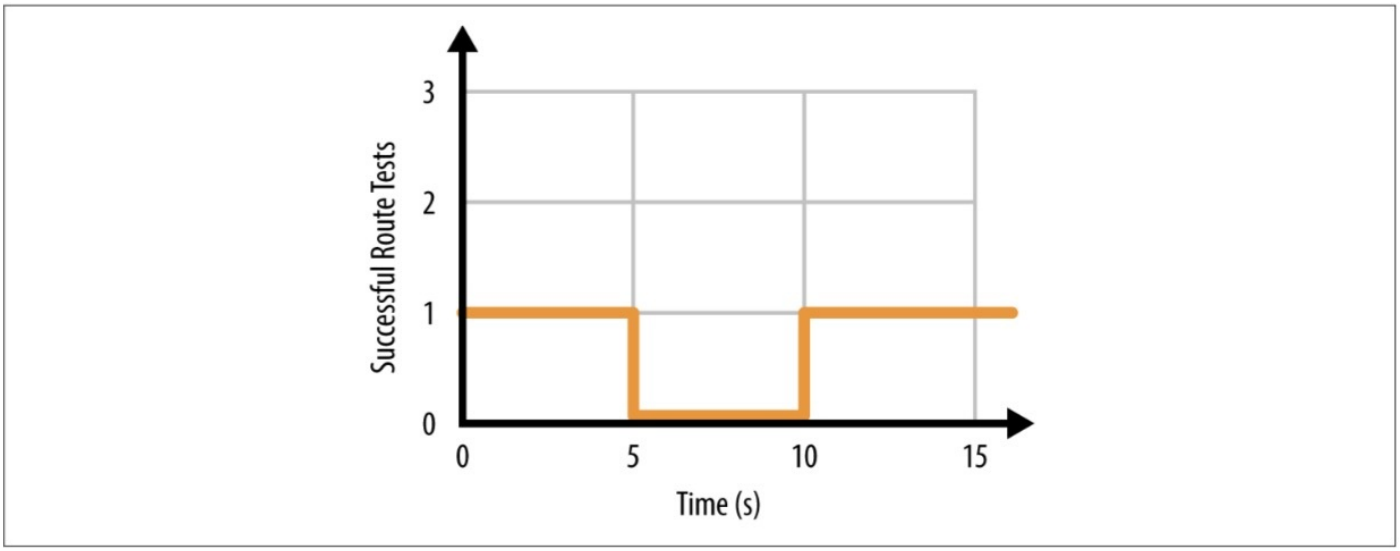

图6-2. 失败并且随着时间的推移修复AssertRoutable测试

这里考虑的重要方面是时间。随着时间的推移，测量混乱将伴随着混沌的测量。

我们可以快速扩展测量。想象一下，名为 `AssertRouteable()` 的服务现在正在虚拟机上调用一组100个基础结构断言。另外，假设我们有100台虚拟机正在测量。

这将对我们的基础设施产生大约每秒1.0×104个断言。来自我们的基础设施断言的大量数据使我们能够创建强大的图形表示基础设施。以可查询的格式记录数据也可以进行高级混沌调查。

随着混沌的测量，拥有可靠的测量工具和服务非常重要。以有意义的方式存储来自服务的数据也很重要，以便稍后可以引用它们。强烈建议将数据存储在日志聚合器或其他容易索引的数据存储中。

系统的混乱与系统的可靠性成反比。因此，它直接反映了我们正在评估的基础设施的稳定性。这意味着，当事情发生中断或引入变化时，将信息随时间绘制成分析是非常有价值的，以了解是否降低了稳定性。

引入混沌

将混沌引入系统的另一种说法：“故意破坏系统”。我们希望总结出我们可能在野外看到的意想不到的基础设施问题组合。如果我们不会故意注入混沌，那么云提供商、互联网或某个系统会为我们做这件事。

> **Netflix的猿人军队**
>
> Netflix推出了它称之为猿猴军队（Simian Army）的系统，导致其混乱。猴子、猿猴以及猿猴家族的其他动物都以不同的方式造成混乱。Netflix解释了这些工具之一Chaos Monkey的工作原理：
>
> 构建Chaos Monkey是我们的哲学，它是随机禁用我们的生产实例以确保我们能够在没有任何客户影响的情况下经受这种常见故障的工具。这个名字来自于在数据中心（或云区域）用武器释放野猴以随机击落实例并通过电缆咀嚼的想法—这一切都是在我们不间断的继续为客户提供服务的同时进行。通过在工作日中间运行Chaos Monkey，在受到严密监控的环境中，工程师站在一边解决任何问题，我们仍然可以学习有关系统弱点的教训，并构建自动恢复机制来处理这些问题。所以下一次星期天上午3点有一个实例失败的时候，我们甚至不会注意到。
>
> 就云原生基础设施而言，Monkey是基础设施作为软件和利用协调模式的很好例子。主要区别在于它们旨在以意想不到的方式摧毁基础设施，而不是可预测地创建和管理基础设施。

此时，您应该有一个准备好使用的基础设施管理应用程序，或者至少有一个。用于部署，管理和稳定基础设施的基础设施管理应用程序也可用于引入混乱。

想象一下两个非常相似的部署。

第一个示例6-4代表有效（或happy）基础架构。

例6-4. infrastructure_happy.json

```
{
    "virtualMachines": [{
        "name": "my-vm",
        "size": "large",
        "localIp": "10.0.0.17",
        "subnet": "my-subnet"
    }],
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

我们可以使在环境中设置的方式来部署此基础设施。这个基础设施应该部署且运行稳定。就像以前一样，随着时间的推移记录您的基础设施测试非常重要；图6-3就是一个例子。理想情况下，您运行的测试数量应该随着时间的推移而增加。

我们决定引入混乱。因此，我们创建了原始基础设施管理应用程序的副本，但这次我们采取了更加险恶的方式部署基础设施。我们利用我们的部署工具的能力来审计基础设施，并对已经存在的基础设施进行更改。

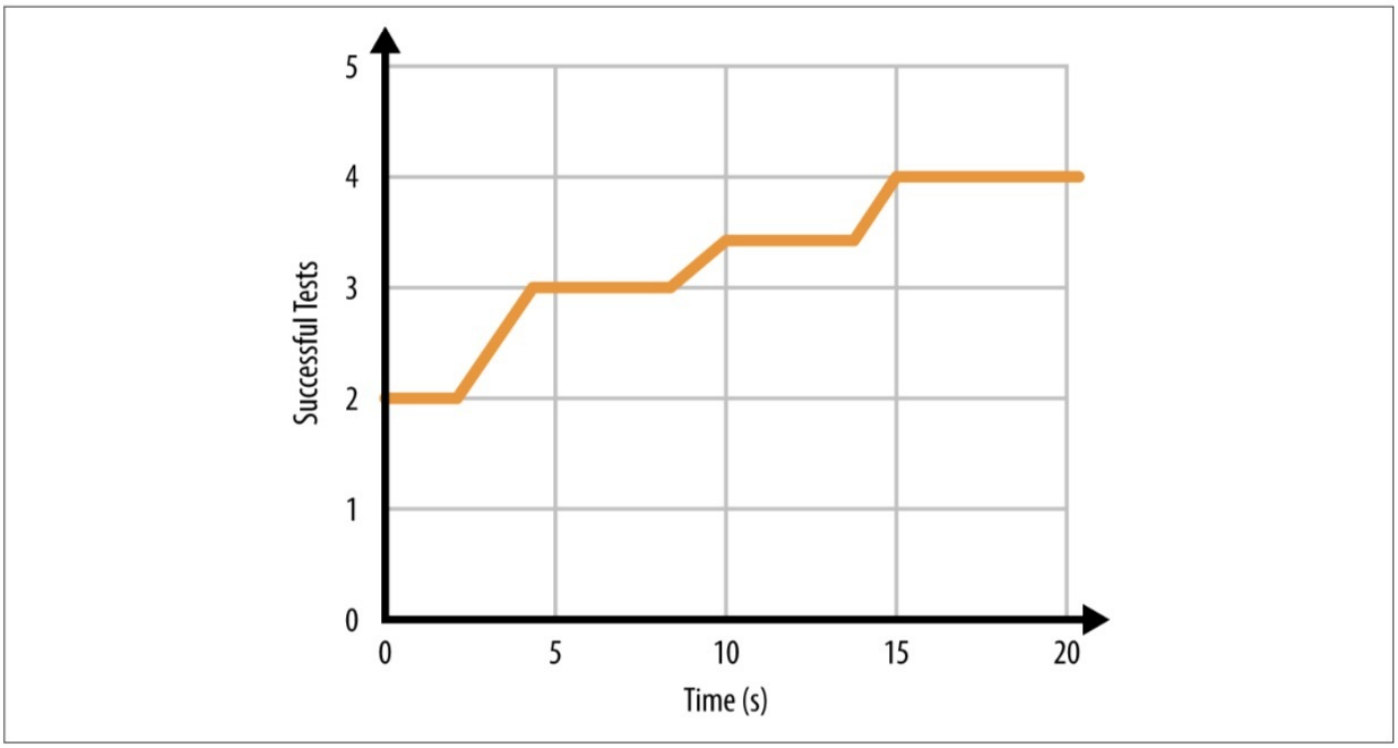

图6-3. 成功的测试

第二次部署将代表有意故障的基础设施，并仍使用与原始基础设施相同的标识符（名称）。基础设施管理工具将检测现有基础设施并进行更改。在第二个示例（示例6-5）中，我们将虚拟机大小更改为较小，并且意图将虚拟机的静态IP地址192.168.1.111分配到10.0.100.0/24范围之外。

我们知道虚拟机上的工作负载不会在小型虚拟机上运行，并且我们知道虚拟机将无法在网络上路由。这是我们将要介

绍的混乱情况。

例6-5. infrastructure_sad.json

```
{
    "virtualMachines": [{
        "name": "my-vm",
        "size": "small",
        "localIp": "192.168.1.111",
        "subnet": "my-subnet"
    }],
    "subnets": [{
        "name": "my-subnet",
        "cidr": "10.0.100.0/24"
    }]
}
```

由于第二个基础设施管理应用程序默默地对基础设施进行了更改，因此我们可以预料会看到事态发展。我们图中的数据将开始波动，如图6-4所示。

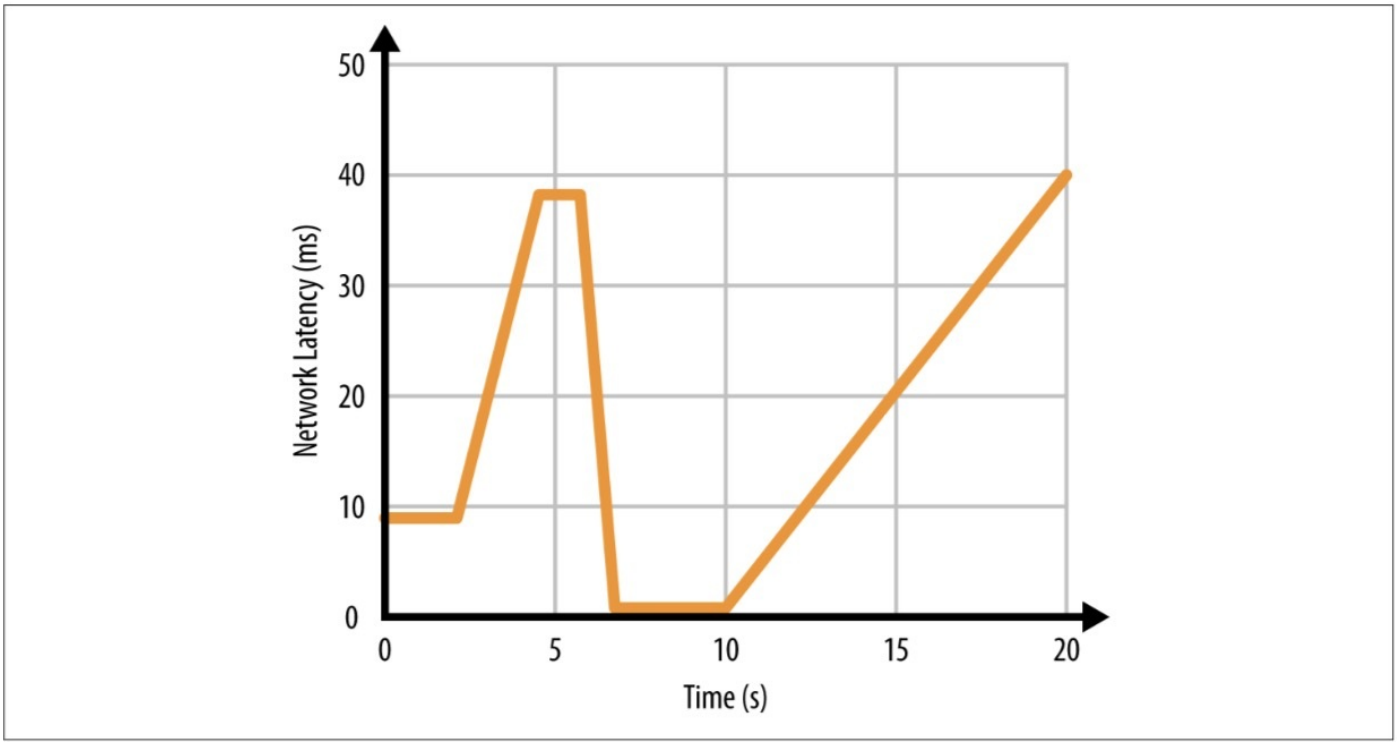

图6-4. 包含网络故障的图形

如果虚拟机上的任何应用程序未完全中断，则应该缓慢地失败。虚拟机的内存和CPU现在已经过载。该shell无法fork新进程。负载平均值远高于20。系统正在接近死锁，我们甚至无法访问虚拟机来查看错误，因为没有任何方式可以路由到冒名顶替者的IP地址。

正如预期的那样，初始系统将检测到底层基础设施中的某些内容发生了变化，并会相应地进行调整。冒名顶替者系统脱机是非常重要的，否则两个系统之间可能会有永无休止的和解，而这两者将会按照指示的方式相互竞争以纠正基础设施。

这种引入混沌的方法之美在于，我们不需要开发任何额外的工具或花费任何工程时间编写混沌框架。我们以巧妙的方

式滥用了原有的基础设施管理工具，引发了一场灾难。

当然，这可能并不总是一个完美的解决方案。与您的生产基础设施应用程序不同，您的混沌应用程序应该有一定的限制，以确保它们有益。一些常见的限制是能够根据标签或元数据排除某些系统，不能在非工作时间运行混沌测试，并将混沌限制在特定的百分比或系统类型。

现在引入随机混沌的主要负担在于基础设施工程师随着时间推移而随机化探索的工作流程的能力。当然，基础设施工程师还需要确保从实验中收集的数据以可消化的格式提供。

监控基础设施

除了测试基础设施外，我们不能忘记监控正在运行的系统。测试和熟悉的失败模式可以让您对基础设施充满信心，但要测试系统可能出现的所有故障是不可能的。

监测可以检测到在测试期间未识别的异常并执行正确的操作是非常重要的。通过积极监控站点基础设施还可以增强我们的信心，即当发生的事情没有被认为是“正常”时，我们会收到警报。明确什么时候以及如何提醒人类这些异常是一个很有争议的话题。

在云原生环境中监控基础设施的实践中涌现出许多优秀的资源。我们不会在这里讨论这些主题，但您应该先阅读Rob Ewaschuk的“监控分布式系统：Google的SRE团队的案例研究”（O’Reilly），并观看MonitoramaConference上的视频。两者都可以在线免费观看。

无论您实施哪种监控解决方案，都要记住用云原生方法来创建您的监控规则。规则应声明并存储为代码。监控规则应与您的应用程序代码放在一起，并以自助服务的方式提供。当测试和遥测可能满足您的大部分需求时，不要过度补偿监控。

结论

测试可以将强我们对基础设施的信心，我们所支持的应用程序也获得了信心和信任。如果一个测试套件不能提供信心，它的价值应该是有问题的。记住，本章中提出的工具和模式是出发点，旨在激发和吸引在这个领域中工作的工程师。无论测试类型或运行它们的框架如何，最重要的一点是工程师可以开始相信他们的系统。作为工程师，我们通常会通过观察实践证明事情按预期工作的动手演示获得信心。

而且，在生产中进行实验不仅是可以的，而且是值得鼓励的。您需要确保环境是为了进行这种实验而建立的，并且实施了适当的跟踪，以便不会浪费测试！

现实测量是基础设施开发和测试的重要组成部分。能够从工程角度和运营角度来封装现实是运用基础设施的重要组成部分，因此可以确信它能够按预期运行。

第7章 管理云原生应用程序

云原生应用程序依赖基础设施才能运行。正如我们在前面的章节中所展示的，云原生基础设施又要通过应用程序来维护。

使用传统基础设施，大部分工作时间，维护和升级应用程序都由人完成。这可以包括在单个主机上手动运行服务或使用自动化工具定义基础结构和应用程序的快照。

但是，如果基础设施可以由应用程序管理并同时管理应用程序，那么基础设施工具就会成为另一种应用程序。工程师在基础设施的责任可以用调解器模式表示，并内置到在该基础设施上运行的应用程序中。

我们刚刚花了前三章来解释我们如何构建可以管理基础设施的应用程序。本章将介绍如何在基础设施上运行这些或其它任何应用程序。

正如之前讨论的，保持基础设施和应用程序的简单是非常重要的。治理应用程序复杂性的常用方法是将它们分解成小的，易于理解的组件。我们通常通过创建单一职责的服务来实现这一点，或者将代码分解为一系列事件触发的函数。

随着小型、可部署单元的扩张即使是最自动化的基础设施也可能被压垮。管理大量应用程序的唯一方法是让它们承担第1章中所述的功能性操作。应用程序需要在可以按规模管理之前变成原生云。

本章不会帮助你构建下一个伟大的应用程序，但它将会为你提供一些基础，让你的应用程序在云原生基础设施上运行时运行良好。

应用程序设计

有很多书讨论如何构建应用程序，本书不打算去讨论。但是，了解应用程序体系结构如何影响了有效的基础设施设计仍然很重要。

正如我们在第1章中所讨论的那样，我们将假设应用程序被设计为云原生，因为它们从云原生基础设施中获得最大收益。从根本上说，云原生意味着应用程序旨在由软件而不是人类管理。

应用程序的设计与打包方式是分开考虑的。应用程序可以是云原生，并且可以打包为RPM或DEB文件，也可以部署到虚拟机而不是容器。它们可以是单体应用或微服务，可以用Java或Go编写。

这些实现细节不影响应用程序被设计成在云上运行。

作为一个例子，假设我们有一个用Go编写的应用程序，被打包在一个容器中。我们甚至可以假设容器运行在Kubernetes上，并且无论你选择怎么定义，这都将被视为微服务。

这个假设的应用就是“云原生”？

如果应用程序将所有活动日志记录到文件并硬编码数据库IP地址？也许它不接受运行时配置并将状态存储在本地磁盘上。如果它不以可预见的方式存在或挂起并等待人工进行调试呢？

这个应用程序可能会从所选择的语言和包装方式中呈现出云原生，但它绝不是云原生应用。像Kubernetes这样的框架可以通过各种功能来帮助管理这个应用程序，但即使你能够使其运行，该应用程序也被明确设计为由人类维护和运行。

第1章详细介绍了使应用程序在云原生基础设施上运行得更好的一些特点。如果我们具有第1章中规定的特点，应用程序还有另一个考虑因素：我们如何有效地管理它们？

实施云原生模式

诸如弹性伸缩，服务发现，配置，日志，健康检查和相关监控指标等功能都可以以不同方式在应用程序中实现。实现这些功能的常见做法是通过导入实现相关功能的标准语言库。 Netflix OSS和Twitter的Finagle是在Java语言库中实现这些功能的很好的例子。

当你使用库时，应用程序可以导入这个库，并且它会自动获得许多相关的功能，无需额外的代码。当一个组织内支持的语言很少时，这种模式很有意义。这种模式很容易去实现最佳实践。

当组织开始实施微服务时，它们往往倾向于使用多语言服务。这样可以自由地为不同的服务选择正确的语言，但是这很难为每种语言维护库。

另一种获得这些特征的方法是通过所谓的”Sidecar”模式。此模式将实施各种管理功能的应用程序捆绑在一起。它通常作为单独的容器来实现，但你也可以通过在虚拟机上运行另一个守护进程来实现它。

SideCar的例子包括以下内容：

Envoy代理

为服务增加弹性伸缩和监控指标

注册

通过外部服务发现注册服务

动态配置

订阅配置更改并通知服务进程重新加载

健康检查

提供用于检查应用程序运行状况的HTTP端点(EndPoints)

Sidecar容器甚至可以用来适配Polyglot容器，通过暴露特定于语言的端点与使用库的应用程序进行交互。来自Netflix的Prana正是为那些不使用标准Java库的应用程序做的。

当集中的团队管理特定的边车进程时，Sidecar模式很有意义。如果工程师想要在它们的服务中暴露监控指标，它们可以将其构建到应用程序中,或者一个单独的团队也可以提供处理日志记录输出并公开计算出的监控指标的边车应用。

在这两种情况下，服务都可以添加功能，而不必重写应用程序。一旦能够使用软件管理应用程序，我们来看看应该如何管理应用程序的生命周期。

应用程序生命周期

除了云原生应用程序的生命周期应该由软件管理，云原生与传统应用程序的生命周期没有什么不同。

本章不打算解释管理应用程序时涉及的所有模式和选项。我们将简要讨论几个阶段，在云原生基础设施之上运行云原

生应用，这几个阶段受益程度最高：部署，运行和下线。

这些主题并不都包含所有选项，但还有很多其他书籍和文章可供参考，这取决于应用程序的架构，语言和所选库。

部署

部署是应用程序最依赖基础设施的一个领域。虽然没有什么东西会阻止应用程序自行部署，但基础设施管理还可以管理更多的方面。

我们不会涉及你如何进行整合和交付，但是在这个领域的一些做法很明确。应用程序部署不仅仅是获取代码并运行它。

云原生应用程序旨在由软件管理各个阶段。这包括的周期性的健康检查以及初始化部署。应尽可能地消除技术，流程和策略中的人为造成的瓶颈。

应用程序的部署首先应该是自动、自助的，并且如果处于积极的开发中，则应该是频繁触发的。它们也应该被测试，验证能够稳定运行。

一次替换应用程序的所有实例很少会成为新版本和新功能的发布方案。新功能在配置标志成"gated"，可以在不重启应用的情况下选择性地动态启用。版本升级部分展开，通过测试进行验证，并在所有测试通过时以受控方式展开。

当启用新功能或部署新版本时，应该存在控制流向或隔离应用流量的机制（请参阅附录A）。这可以限制中断的影响，并允许缓慢的部署和更快的应用性能反馈循环进行新功能的使用。

基础设施应负责部署软件的所有细节。工程师可以定义应用程序版本，基础设施要求和依赖关系，并且基础设施将朝着该状态发展，直至满足所有要求或需求更改。

运行

运行应用程序应该是应用程序生命周期中最平稳最稳定的阶段。运行软件最重要的两个的方面在第1章中讨论：了解应用程序在做什么以及可操作性即可以根据需要更改应用程序。

我们已经在第1章中详细介绍了关于应用报告健康和遥测数据的可观察性，但是当事情不按预期工作时，你会做什么？如果应用程序的遥测数据显示它不符合SLO，那么如何解决和调试应用程序？

对于云原生应用程序，你不应该通过SSH连接到服务器的形式查看日志。如果你需要SSH，更应该考虑使用日志或其它的服务替代。

你仍然需要访问应用程序（API），日志数据（云日志记录）以及获取某个位置的堆栈的服务，但这值得通过演练来查看是否需要传统工具。当事件中断时，你需要一个调试应用程序和基础设施组件的方法。

在调试一个坏了的系统时，你应该首先查看你的基础设施测试，如第5章所述。测试应公开所有未正确配置或未提供预期性能的基础设施组件。

只因为你不管理底层基础设施并不意味着基础设施不能成为你问题的原因。通过测试来验证期望值将确保你的基础设施能够以你期望的方式运行。

在排除基础设施后，你应该查看应用程序以获取更多信息。转向应用程序调试的最佳位置是应用性能管理（APM）以及可能通过OpenTracing等标准进行的分布式应用程序跟踪。

OpenTracing示例，实现和APM不在本书的范围之内。总而言之，OpenTracing允许你在整个应用程序中跟踪调用，以更轻松地识别网络和应用程序通信问题。 OpenTracing的示例可视化可以在图7-1中看到。 APM为你的应用程序添加了用于向收集服务报告指标和故障的工具。

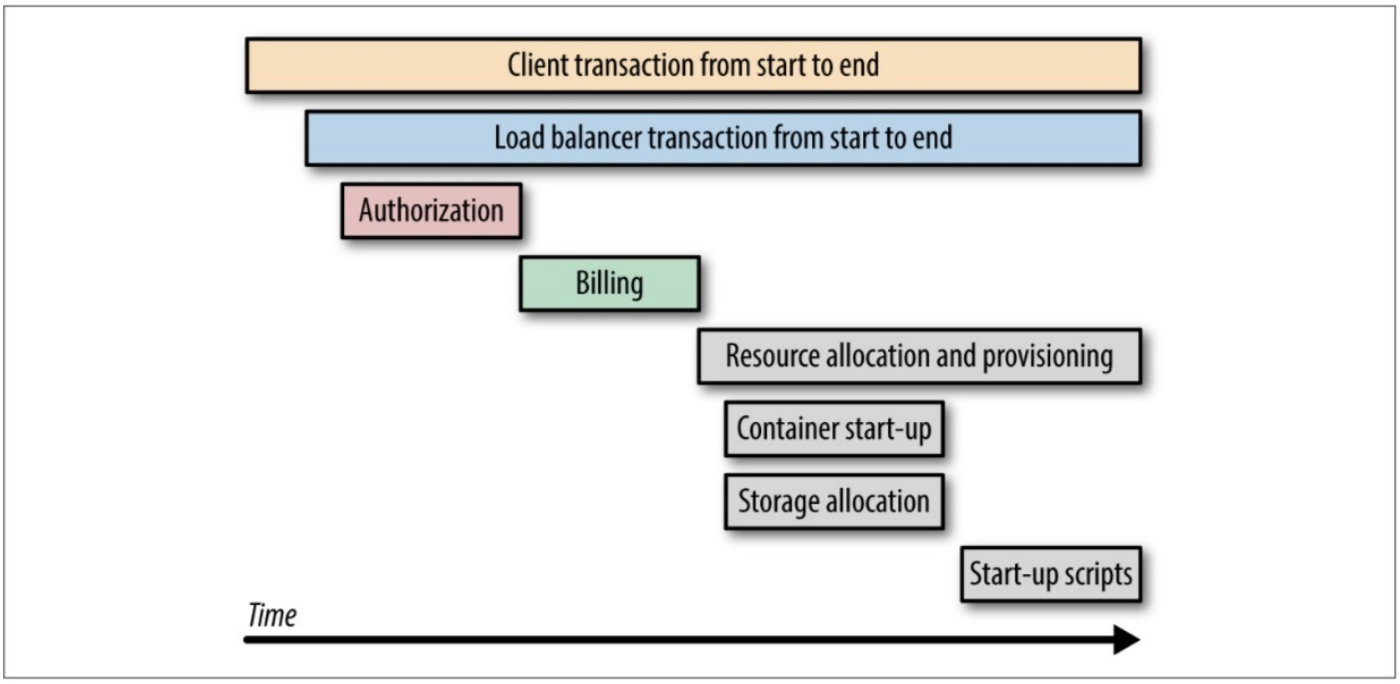

图7-1. OpenTracing可视化

当测试和跟踪仍然没有暴露出问题时，有时你只需要在应用程序上启用更详细的日志记录。但是，如何在不破坏问题的情况下启用调试？

运行时配置对于应用程序很重要，但在云原生环境中，无须重启应用程序，配置就应该是动态的。配置选项仍然通过应用程序中的库实现，但标志值应该能够通过集中协调器，应用程序API调用，HTTP协议Header或多种方式进行动态更改。

Netflix的Archaius和Facebook的Gate-Keeper是动态配置的两个例子。前Facebook工程师经理贾斯汀米切尔(Justin Mitchell)在Quora的帖子中分享到：

> [Gatekeeper]从代码部署中解耦出来的功能。我们可以在几天或几周内发布新功能，因为我们观看了用户指标，性能并确保服务可以随时扩展。

允许对应用程序配置进行动态控制，可以实现更多对曝光过度的新功能的控制并更好地测试已部署代码的覆盖范围。只因为推新代码很容易并不意味着它是适合所有情况的正确解决方案。

基础设施可以帮助解决此问题，并通过协调何时启用新功能和基于高级网络策略的路由控制来启用更灵活的应用程序。这种模式还允许更细粒度的控制和更好的协调发布或回滚场景。

在动态的自助服务环境中，部署的应用程序数量将快速增长。你需要确保你有一个简单的方法来动态调试应用类似自助服务模型中的部署应用。

与工程师喜欢推新应用程序一样，反过来很难让它们下线旧应用程序。即使如此，它仍然是应用程序生命周期中的关键阶段。

下线

部署新的应用程序和服务在快速迭代的环境中很常见。下线应用程序应该像创建它们一样自动化。

如果新的服务和资源被自动化部署与监控，则它们应该按照相同标准下线。尽快部署新服务而不删除未使用的服务是应对技术债务的最简单方法。

识别应该下线的服务和资源是特定的业务。你可以使用应用程序遥测的经验数据来了解某个应用程序是否正在被使用，但是下线应用程序的决定应由该业务决定。

基础设施组件（例如，VM实例和负载均衡器端点）应在不需要时被自动清理。自动化组件清理的一个例子是Netflix的Janitor Monkey。该公司在一篇博文中解释道：

> Janitor Monkey通过应用一组规则来决定资源是否应该成为候选的清理内容。如果任何规则确定该资源是被清理的候选内容，则Janitor Monkey标记该资源并安排清理时间。

所有这些应用阶段的目标是让基础设施和软件来管理原本由人类管理的方面。我们采用协调模式与组件元数据相结合的方式来不断运行，并根据当前上下文对需要采取的高层次操作做出决策。以此来取代由人类编写的临时自动化脚本。

应用程序的生命周期不是唯一一个需要依赖于基础设施的阶段。还有一些每个阶段都要依赖于基础设施服务的程序。我们将在下一节讨论一些提供给这类应用的支持服务和基础设施API。

基础设施上的应用要求

云原生应用程序对基础设施的期望不仅只是执行二进制文件，它们还需要抽象，隔离与保证它们将如何运行和被管理。作为回报，它们被要求提供钩子和API以允许基础设施管理它们。为了成功运行，两者就需要有一种共生关系。

我们在第1章中定义了云原生应用程序，并刚刚讨论了一些生命周期的要求。现在让我们看看云原生应用程序对从运行它们的基础设施建设的更多期望：

- 运行与隔离
- 资源分配和调度
- 环境隔离
- 服务发现
- 状态管理
- 监控和记录
- 监控指标聚合
- 调试和跟踪

所有这些期望都应该是服务的默认选项，或者是由自助API提供。我们将更详细地解释每个要求，以确保这些期望被明确的定义。

应用程序运行和隔离

除了有时候需要的解释器，传统应用程序只需要一个内核就可以运行。 云原生应用仍然需要它们，但云原生应用运行时同样也需要与操作系统和其他应用程序隔离。隔离使多个应用能够在同一台服务器上运行并控制它们的依赖和资源。

应用隔离有时被称为多租户。该术语可用于在同一服务器上运行的多个应用程序以及在共享集群中运行应用程序的多个用户。用户可以运行经过验证的可信代码，也可以运行你不能控制且不信任的代码。

云原生不意味着需要使用容器。 Netflix率先推出了许多云原生模式，当公司从原来的方式过渡到在公共云上运行时，它使用虚拟机作为它们的部署工具，而不是容器。 FaaS服务（例如AWS Lambda）是用于打包和部署代码的另一种流行的云原生技术。在大多数情况下，它们使用容器进行应用程序隔离，但容器包装对用户是不可见的。

> 什么是容器？
>
> 容器有很多不同的实现。 Docker推广了术语“容器”来描述一种在隔离的环境中打包和运行应用程序的方式。基本上，容器使用内核原语或硬件功能来隔离单个操作系统上的进程。
>
> 容器隔离级别可能会有所不同，但通常这意味着应用程序使用独立的根文件系统、命名空间以及来自同一服务器上其他进程的资源分配（例如，CPU和RAM）运行。容器格式已被许多项目采用，并创建了开放容器计划（OCI），该计划定义了如何打包和运行应用程序容器的标准。

容器隔离还会给编写应用程序的工程师造成负担。它们现在负责声明所有的软件依赖关系。如果它们没做到这点，应用程序将无法运行，因为必要的库将不可用。

容器经常被选中来用于云原生应用程序，因为已经出现了更好的用于管理它们流程和编排的工具。虽然容器是实现运行时和资源隔离的最简单方式，但这并不总是（并且也可能不会）如此。

资源分配和调度

从历史上看，应用程序可以提供最低系统要求的粗略估计，人类有责任确定应用程序满足什么需求下可以运行。人工调度可能需要很长时间才能准备好应用程序运行的操作系统和依赖项。

部署可以通过配置管理实现自动化，但仍然需要人员验证资源并标记服务器以运行应用程序。云原生基础设施基于于依赖隔离，允许应用程序在任何资源可用的地方运行。

通过隔离，只要系统有可用的进程，存储和可访问的依赖，应用程序就可以在任何地方被调度。动态调度通过将决策留给机器更好地消除了人为瓶颈。集群调度程序从所有系统收集资源信息并计算出应用程序的最佳位置。

> 人为控制应用程序不能很好的伸缩。人类生病，休假（或至少它们应该），通常是瓶颈。随着规模和复杂性的增加，人们也不可能清楚地记住应用程序在哪里运行。
>
> 许多公司试图通过招聘更多人来扩大规模。这加剧了系统的复杂性，因为调度需要在多个人之间进行协调。最终，人为调度将采用电子表格（或类似的解决方案）来保存每个应用程序的运行位置。

动态调度并不意味着操作员无法控制。基于调度器可能没有的知识，操作员仍然可以覆盖或强制进行调度决策。覆盖和手动资源调度应通过API提供，而不是会议请求。

解决这些问题是Google编写名为Borg的内部集群调度程序的主要原因之一。在Borg的研究报告中，谷歌指出：

Borg提供了三大好处：（1）它隐藏了资源管理和失败处理的细节，因此用户可以专注于应用程序开发；（2）以非常高的可靠性和可用性运行，并支持相同的应用程序；（3）让我们可以有效地在数以万计的机器上运行工作负载。

调度程序在任何云原生环境中的角色都非常相似。从根本上说，它需要抽象出许多机器并允许用户而不是服务器请求资源。

环境隔离

当应用程序由许多服务组成时，基础设施就需要提供一种方法来定义所有依赖的隔离。传统的方法是通过将复杂的服务器，网络或群集隔离成开发或测试环境来管理依赖关系。基础设施应能够通过应用程序环境在逻辑上分离依赖关系，而不会发生集群完全重复。

逻辑分割环境允许更好地利用硬件，减少重复的自动化，并且更容易测试应用程序。在某些情况下，需要单独的测试环境（例如，需要进行低级别更改时）。但是，应用程序测试并应该在基础设施完全的重复的情况下进行。

环境可以是传统的永久性开发，测试，预发和生产，也可以是动态分支或基于提交（commit）。它们甚至可以是生产环境的一部分，通过动态配置和实例的选择性路由启用功能。

环境应由应用程序所需的所有数据，服务和网络资源组成。这包括诸如数据库，文件共享和任何外部服务之类的东西。云原生基础设施可以创建低开销的环境。

基础设施应该能够提供环境，然而它被使用。应用程序应遵循最佳实践，允许灵活配置以支持环境，并通过服务发现发现支持服务的端点。

服务发现

应用程序几乎可以肯定依靠一项或多项服务来提供商业利益。基础设施的责任是提供一种服务在每个环境基础上找到彼此的方式。

某些服务发现需要应用程序进行API调用，而其他服务则通过DNS或网络代理公开透明地进行。使用什么工具并不重要，但服务使用服务发现很重要。

尽管服务发现是最古老的网络服务之一（即ARP和DNS），但它经常被忽视并且不被利用。在每个实例文本文件或代码中静态定义服务端点是不可扩展的，且不适合云原生环境。端点(Endpoint)注册应该在创建服务时自动发生，并且端点可用或消失。

云原生应用程序与基础设施一起工作以发现其相关服务。这些包括但不限于DNS，云元数据服务或独立服务发现工具（即etcd和consul）。

状态管理

如果有状态管理的话基础设施将能知道应用程序实例需要做什么。这与应用程序生命周期截然不同，因为生命周期适用于整个开发过程中的应用程序。状态适用于启动和停止的实例。

应用程序有责任提供API或钩子，以便检查其当前状态。基础设施的责任是监控实例的当前状态并采取相应的行动。

以下是一些应用程序状态：

- 已提交
- 预定
- 准备好了
- 健康
- 不健康
- 终止

这些状态和相应行动的简要概述如下：

1. 一个应用申请提交运行。
2. 基础设施检查请求的资源并安排应用程序。当应用程序启动时，它将提供一个准备好/未准备好的状态。
3. 基础设施将等待就绪状态，然后允许使用应用程序资源（例如，将实例添加到负载均衡器）。如果应用程序在指定的时间前未准备就绪，基础结构将终止它并安排一个新的应用程序实例。

4. 一旦应用程序准备就绪，基础设施将监控活动状态并等待不健康状态，或者直到应用程序设置为不再运行。

还有比上述更多的状态。如果要对状态进行正确的检查和采取行动，则状态需要得到基础设施的支持。 Kubernetes通过事件，探测和挂钩实现应用程序状态管理，但是每个编排平台都应该具有类似的应用程序管理功能。

当应用程序被提交，计划或缩容时，会触发Kubernetes事件。探测器用于检查应用程序何时准备好提供流量（准备就绪）并确保应用程序健康（存活）。挂钩用于在进程启动之前或之后需要发生的事件。

应用程序实例的状态与应用程序生命周期管理同样重要。基础设施在确保实例可用并据此采取行动方面起着关键作用。

监控和记录

应用程序不应该要求被监控或被记录;它们是基础设施运行的基本条件。更重要的是，如果需要被监控和记录，其配置应该以应用程序资源请求相同的方式声明为代码。如果你拥有部署应用程序的所有自动化功能，但无法动态监控服务，云原生基础设施仍有待完成。

状态管理（即进程健康检查）和日志记录处理应用程序的各个实例。日志系统应该能够根据应用程序，环境，标签或任何其他有用的元数据整合日志。

应用程序应该尽可能没有单点故障，并且应该运行多个实例。如果一个应用程序有100个实例正在运行，就算单个实例变得不健康，监控系统也不应触发警报。

监控从整体上看应用程序，并用于调试和验证所需的状态监控与警报不同，因为应根据应用程序的度量和SLO触发警报。

指标聚合

要知道应用程序处于健康状态时的行为方式，收集指标。它们还可以提供有关不健康时可能被破坏的信息的见解，并且就像监控一样，收集的指标应作为代码与应用程序定义的一部分被请求。

基础设施可以自动收集有关资源利用率的指标，但应用程序有责任呈现服务级别指标的指标。

虽然监测和日志记录是应用程序运行时状况检查，但指标可提供所需的遥测数据。没有指标，就无法知道应用程序是否满足服务级别目标以提供商业价值。

> 从日志中提取遥测和健康检查数据可能很诱人，但要小心，因为日志记录需要后处理，并且比应用特定监控指标来说开销更重。
>
> 在收集指标时，你希望尽可能接近实时数据。这需要一个可扩展且简单高效的解决方案。
>
> 应该使用日志进行调试，并且应该预计数据处理的延迟。

与日志记录类似，指标通常在实例级被收集，然后汇总在一起以提供完整的服务视图，而不是单个实例的显示。

一旦应用程序提供收集指标的方法，基础设施的工作就是搜刮，整合和存储指标用于分析。收集指标的端点应该可以根据每个应用程序进行配置，但数据格式应该标准化，以便可以在单个系统中查看所有指标。

调试和跟踪

应用程序在开发过程中很容易调试。集成开发环境（IDE），代码断点以及在调试模式下运行都是工程师在编写代码时

可以使用的所有工具。

对于部署的应用程序来说，自检要困难得多。当应用程序由数十或数百个微服务或独立部署的功能组成时，此问题更为严重。当用多种语言和不同的团队编写服务时，也可能无法将工具内置到应用程序中。

基础设施需要提供调试整个应用程序的方法，不仅仅是单个服务。调试有时可以通过日志记录系统完成，但是复现错误需要较短的反馈循环。

如前所述，调试对于动态配置来说是很好用的。当发现问题时，应用程序可以切换到详细日志记录，而无需重新启动，并且流量可以通过应用程序代理有选择地路由到实例。

如果问题无法通过日志输出解决，那么分布式跟踪提供了一个不同的界面来可视化发生的事情。分布式跟踪系统（如 `OpenTracing`）可以补充日志以帮助人类调试问题。

跟踪为调试分布式系统提供了更短的反馈循环。如果它不能构建到应用程序中，则可以通过代理或流量分析由基础结构透明地完成。当你大规模地运行任何协调的应用程序时，基础结构提供了一种调试应用程序的方法。

尽管在分布式系统中设置跟踪有很多好处和实现细节，但我们不会在此讨论。应用程序跟踪一直非常重要，并且在分布式系统中越来越困难。云原生基础设施需要提供可以以透明方式跨越多个服务的跟踪服务。

结论

应用程序需求已经改变：带有操作系统和软件包管理器的服务器已经不够用了。应用程序现在需要协调服务和更高级别的抽象。抽象允许资源与服务器分离并根据需求以编程的方式使用。

本章中提出的要求并不是基础设施可以提供的所有服务，但它们是云原生应用程序所期望的基础。如果基础设施不提供这些服务，那么应用程序将不得不实施它们，否则它们将无法达到现代业务所需的规模和速度。

基础设施不会自行发展;人们需要改变它们的行为，并从根本上想到以不同的方式运行应用程序需要做的事情。幸运的是，有些项目已经在借鉴开创了这些解决方案的公司的经验了。

应用程序依赖基础设施的功能和服务来支持敏捷开发。基础设施要求应用程序公开端点和集成以自主管理的方式。工程师应尽可能使用现有的工具，并设计出有弹性的简单解决方案。

第8章 保护应用程序

我们已经讨论过只能从云应用程序提供基础架构。我们知道基础设施也负责运行这些相同的应用程序。

运行由应用程序配置和控制的基础架构可以让我们轻松扩展。我们通过学习如何扩展应用来扩展基础设施。我们还通过学习如何保护应用程序来保护我们的基础设施

在动态环境中，我们已经表明人类无法扩展来管理复杂性。我们为什么会认为他们可以扩大规模来处理政策和安全问题？

这意味着，就像我们必须创建通过协调器模式强制执行基础架构状态的应用程序一样，我们需要创建实施安全策略的应用程序。在我们创建应用程序以执行策略之前，我们需要以机器可分析的格式编写我们的策略。

作为代码的政策

由于政策没有明确定义的技术实施，所以政策难以纳入代码。它更多地关注业务如何完成，而不是业务运营。

经常变化的方式和方式都是如此，但如何更加自以为是，并且不容易被抽象化。它也是组织特定的，可能需要了解创建基础架构人员的通信结构的具体细节。

策略需要应用于应用程序生命周期的多个阶段。正如我们在第7章中所讨论的，应用程序通常有三个阶段;部署，运行和退役。

部署阶段将在应用程序和基础架构更改发布之前应用策略。这将包括部署规则和一致性测试。运行阶段将包括持续的遵守和执行访问控制和隔离。退休阶段很重要，以确保没有服务落后于未安排或未维护的州。

在这些阶段中，您需要将政策分解为明确的，可操作的实施。模糊的政策无法执行。您需要将实现放在代码中，然后创建应用程序或使用现有的应用程序来执行策略规则。

一旦您将策略作为代码使用，您应该将其视为代码。策略更改应视为应用程序更改并在版本控制中进行跟踪。

控制应用程序部署的相同策略也应该适用于您的新策略部署。您可以使用与部署应用程序相同的工具跟踪和部署的基础架构组件越多，就越容易了解正在运行的内容以及更改如何影响系统。

作为代码的政策带来的巨大好处是您可以轻松地添加或删除策略并对其进行跟踪，因此记录了谁执行了策略，何时执行了策略以及提交和提交请求的评论。由于该政策以代码形式存在，因此您现在可以为自己的政策编写测试！如果你想验证一个策略能够做正确的事情，你可以使用第5章中的测试实践。

让我们更仔细地看看如何将策略应用到应用程序生命周期。

部署Gateway

部署gateway确保应用程序的部署符合业务规则。这意味着您将需要构建部署管道，并且不允许从用户机器进行直接生产部署。

在实施集中化策略之前，您需要集中控制，但是应该从小规模开始，并在实施之前证明解决方案可行。部署管道的好处远不止于策略执行，而且应该是任何拥有少数开发人员的组织中的标准。

以下是一些政策示例：

- 部署只有在所有测试都通过后才能进行。
- 新应用程序要求高级开发人员检查更改并对提取请求发表评论。
- 生产工件推送只能从部署管道发生。

gateway不应该强制运行状态或应用程序的API请求。应用程序应该知道如何配置基础架构组件，并通过合规性和审计将策略应用于这些组件，而不是在应用程序部署期间应用。

部署gateway策略的一个例子是，如果您的组织在过去的3点之前不允许部署代码，在没有经理批准的星期五。

这个很容易放入代码中。图8-1是代表政策的非常简化的图。

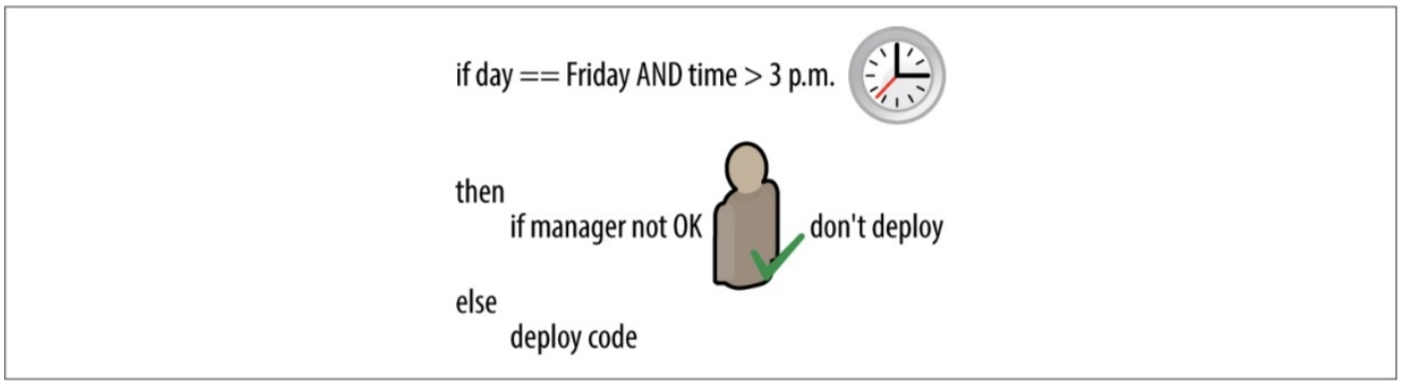

图8-1. 部署策略

您可以看到该策略简单地检查了允许部署部署的一周中的时间和日期。如果是星期五和下午3点以后，那么政策会检查管理员确定。

该策略可以通过经理发送的经过验证的电子邮件，经过验证的API调用或各种其他方式来获得OK通知.2决定首选通信方法的内容以及等待批准的时间长度取决于策略。

这个简单的例子可以用很多不同的选项进行扩展，但确保该策略不符合人类解析和执行是很重要的。人的解释是不同的，而不明确的政策通常不会得到执行。

通过确保策略可以阻止新的部署，我们可以为解决生产环境的状态节省很多工作。有一系列可以通过软件验证的事件可以帮助您理解您的系统。版本控制和持续部署管道可以验证代码;使用策略和流程作为代码可以验证软件的部署方式和时间。

除了确保通过公司政策部署正确的事情之外，我们还应该轻松地使用模板部署受支持的事物，并通过一致性测试强制执行它们。

符合性测试

在任何基础架构中，您都需要提供建议的方式来创建特定类型的应用程序。这些建议成为用户根据需要消费和拼凑在一起的基石。

这些建议是可堆肥的，但不能太小是很重要的。他们需要在其预期的功能和自助服务方面可以理解。我们已经建议将云原生应用程序打包为容器并通过协调器使用;由您决定什么最适合您的用户以及您想要提供哪些组件

> 您可以通过多种方式为用户提供模板化基础架构。

> 一些例子是提供一个模板化的代码，比如Jsonnet，或者完全模板化的应用程序，例如Helm的图表。
>
> 您还可以通过您的部署管道提供模板。这些可以是Terraform模块或特定于部署的工具，例如Spinnaker模板。
>
> 创建部署模板允许用户成为模板的消费者，随着最佳实践的发展，用户将自动受益。

基础架构模板的最关键的方面是使做正确的事情变得容易，并且很难做错事情。如果您满足客户的需求，那么获得适配器将会容易得多。

但是，我们需要模板化基础架构的全部原因是我们可以执行符合性测试。符合性测试的存在是为了确保应用程序和基础架构组件符合组织的标准。

对不符合标准的基础设施进行测试的一些示例如下：

- 不在自动缩放组中的服务或端点
- 不在负载均衡器后面的应用程序
- 前端层直接与数据库交谈

这些标准是关于基础设施的信息，您可以通过拨打云提供商的API来找到这些信息。合规性测试应持续运行，并强制执行贵公司采用的架构标准。

如果发现基础架构组件或应用程序架构违反了所提供的标准，则应尽早终止它们。您越早可以对模板进行编码，越早可以检查不符合标准的应用程序。在应用程序的生命早期解决不受支持的体系结构非常重要，因此可以最大限度地减少对体系结构决策的依赖。

合规性处理如何构建应用程序和维护可操作性。合规性测试确保应用程序和基础架构保持安全。

一致性测试

合规性测试不会测试架构设计，而是集中于组件的实施，以确保它们遵守定义的策略。放在代码中的最简单的策略是那些有助于安全的策略。围绕组织需求（例如HIPAA）的策略也应定义为代码并在合规性测试期间进行测试。

合规政策的一些例子是：

- 对象存储的用户访问受限，并且不能被公共因特网读取或写入
- API端点全部使用HTTPS并具有有效证书
- 虚拟机实例（如果有的话）没有过分宽松的防火墙规则

虽然这些策略不会使应用程序免受所有漏洞或错误的影响，但是如果应用程序确实被利用，合规性策略应将影响范围降至最低。

Netflix在其博客文章“The Netflix Simian Army”中通过其Security Monkey解释了其执行合规性测试的目的：

> Security Monkey是Conformity Monkey的延伸。它会查找安全违规或漏洞（如未正确配置的AWS安全组），并终止违规实例。它还确保我们所有的SSL和DRM证书都是有效的，并且不会延期。

将您的策略放入代码并通过观察云提供商API继续运行它，可以让您保持更高的安全性，立即捕获不安全的设置，并随时跟踪版本控制系统的策略。根据这些策略不断测试您的基础架构的模型也非常适合调节器模式。

如果您认为策略是需要应用和实施的配置类型，那么实施它可能会更简单。请务必记住，随着您的基础架构和业务需求的变化，您的合规政策也应该如此。

部署测试在将应用程序部署到基础架构之前进行监视，合规性和合规性测试都处理正在运行的应用程序。确保您拥有策略的最后一个应用程序生命周期阶段是了解何时以及如何废止应用程序和生命周期组件。

活动测试

合规性和合规性测试应该删除那些未通过定义策略的应用和基础设施。还应该有一个应用程序清理旧的和未使用的基础架构组件。高级使用模式应基于应用程序遥测数据，但仍有其他基础架构组件很容易被遗忘并需要退役。

在云环境中，您可以根据需要消耗资源，但忘记您的要求很容易。如果没有自动清理旧的或未使用的资源，您最终会对您的使用费用感到惊讶，或者需要耗费大量人力时间进行手动审计和清理。

您应该测试并自动清理的资源的一些示例包括：

- 旧磁盘快照
- 测试环境
- 以前的应用程序版本

负责清理的应用程序需要根据默认策略做正确的事情，并为工程师指定的异常提供灵活性。

正如第7章所提到的，Netflix已经实现了它所称的“Janitor Monkey”，它的实现完美地描述了这种需要的模式：

> 看门人猴子在“标记，通知和删除”过程中工作。当Janitor Monkey将资源标记为清理候选人时，它会安排删除资源的时间。删除时间在标记资源的规则中指定。
>
> 每个资源都与一个所有者电子邮件相关联，该电子邮件可以在资源上指定为标签，或者您可以快速扩展Janitor Monkey以从您的内部系统获取信息。最简单的方法是使用默认的电子邮件地址，例如您的团队的电子邮件列表中的所有资源。您可以配置若干天，以指定何时让Janitor Monkey在计划终止之前向资源所有者发送通知。默认情况下，数字为3，表示业主将在终止日期前3个工作日收到通知。
>
> 在3天期间，资源所有者可以决定资源是否可以删除。如果资源需要保留更长时间，则所有者可以使用简单的REST接口将资源标记为未被Janitor Monkey清除。所有者总是可以使用另一个REST接口来删除该标志，然后Janitor Monkey将能够再次管理该资源。
>
> 当Janitor Monkey看到标记为清理候选者的资源并且预定的终止时间已经过去时，它将删除资源。如果资源所有者想要提前释放资源以节省成本，则还可以手动删除该资源。当资源状态改变而使资源不是清理候选者时，例如一个分离的EBS卷被附加到一个实例，Janitor Monkey将取消该资源的标记并且不会终止。

拥有自动清理基础架构的应用程序可以降低您的复杂性和成本。该测试将协调模式应用到应用程序的最后生命周期阶段。

还有其他一些基础架构的实践很重要，需要考虑。其中一些实践适用于传统基础架构，但在云原生环境中需要进行不同的处理。

不断测试基础架构的各个方面有助于您了解自己遵守的政策。当基础设施进入并频繁发生时，很难审计哪些变化可能会导致停电或使用历史数据来预测未来趋势。

如果您希望从帐单声明中获取该信息或通过推断基础架构的当前快照，您将很快发现所提供的信息是该工作的错误工具。为了跟踪变化并预测未来，我们需要有审计工具，可以快速向我们提供我们需要的信息。

审计基础架构

在这个意义上审计云原生基础设施与审核调解器模式中的组件不同，它与第6章中讨论的测试框架也不同。相反，当我们谈论审计时，我们指的是对变更的高级概述和基础设施内的组件关系。

跟踪基础设施中存在的内容以及它与其他组件的关系，为我们了解当前状态提供了重要的背景。当某些事情中断时，第一个问题几乎总是“什么改变了？”审计为我们回答了这个问题，并且可以用来告诉我们如果我们应用变更会受到什么影响。

在传统的基础设施中，配置管理数据库（CMDB）通常是基础设施当前状态的真相来源。但是，CMDB不会跟踪基础架构或资产关系的历史版本。

云提供商可以通过库存API为您提供CMDB替换服务，但它们可能不会激励您显示历史趋势或让您查询您需要进行故障排除的特定详细信息，例如主机名。

一个好的云审计工具可以让你显示当前基础设施与昨天或上周相比的差异（“差异”）。它应该能够将云提供商数据与其他来源（例如容器编排器）相结合，以便您可以查询云提供商可能没有的数据的基础架构，理想情况下，它可以自动构建组件关系的地形表示。

如果您的应用程序完全在单个平台上运行（例如Kubernetes），则收集资源依赖关系的拓扑信息要容易得多。自动可视化关系的另一种方法是统一关系发生的层次。

对于在云中运行的服务，可以在服务之间的网络通信中识别关系。有很多方法可以识别服务之间的网络流量，但审计的重要考虑是对信息进行历史跟踪。您需要能够轻松识别关系何时发生变化，就像您识别组件来来往往时一样容易。

> 自动识别服务关系的方法
>
> 跟踪服务之间的网络流量意味着您需要了解用于通信的网络协议的工具。您不能简单地依赖流入或流出服务端点的原始数据包流量。你需要一种方法来利用信息流来建立关系模型。
>
> 检查网络流量和构建依赖关系图的一种流行方式是通过网络代理。
>
> 网络代理的一些实现示例是linkerd和envoy。服务通过这些代理来传输所有流量，这些代理知道正在使用的协议和其他相关服务。如附录B所述，代理还允许其他网络弹性模式。

跟踪时间拓扑结构是一个强大的审计工具。结合基础设施的历史，它将确保“改变了什么？”问题更容易回答。

还有一个审计方面涉及在当前基础设施状态下建立信任。应用程序通过所描述的测试工具获得信任，但基础架构的某些方面无法应用相同的测试。

用可验证的可重现组件构建基础架构可以提供很大的信任。这种做法被称为不可变基础设施。

不可变基础设施

不可变的基础架构是通过替换而不是修改来创建变更的做法。换句话说，不是运行配置管理来对所有服务器应用更改，而是建立新服务器并丢弃旧服务器。

云环境很大程度上受益于创建变更的方法，因为部署新VM的成本通常非常低，比管理可强制配置并保持实例运行的另一个系统要容易。因为服务器是虚拟的（即用软件定义），所以您可以像构建服务器映像一样应用与构建应用程序相同的实践。

> 物理服务器的传统基础架构与系统创建时的云具有完全相反的优化。提供物理服务器需要很长时间，并且在修改现有操作系统方面有很大的好处，而不是替换旧的。

不可变基础设施的问题之一是建立信任链，为部署创造黄金形象。图像在构建时应该是可验证的（例如，签名密钥或图像哈希），并且一旦部署就不应改变。

图像创建也需要自动化。历史上使用金色图像的最大问题之一是它们通常依靠人类在耗时的创作过程中贯穿检查表。

存在工具（例如，Hashicorp的Packer）来自动化构建过程，并且没有理由存在旧图像。自动化还允许旧的清单成为可以审计和版本控制的脚本。了解配置工具何时发生变化以及由谁建立信任的不可变基础架构的另一方面。

发生更改时，应该经常更改，您需要一种方法来跟踪更改内容和原因。审计将有助于确定发生了什么变化，而不可变的基础架构可以帮助您追踪到Git提交或拉取请求。

追踪历史也大大有助于从失败中恢复过来。如果您有一个已知可用的虚拟机映像，另一个虚拟机映像不可用，则可以像部署新的破损版本一样快速地部署以前的工作版本。

不可变的基础架构不是云原生基础设施的要求，但环境从这种基础架构管理风格中受益匪浅。

结论

通过本章介绍的实践，您将能够更轻松地控制在基础架构中运行的内容，并跟踪其到达的方式。将您的政策放在代码中可让您跟踪更改，而且您不会依赖人类正确解读政策。

审计和不可变的基础架构为您提供了更好的信息，以确保您的系统安全并帮助您更快地从故障中恢复。

除了本章前面讨论的符合性测试要求之外，我们不会在本书中讨论安全性，但您应该及时了解您使用的技术和云提供商的安全最佳实践。安全性最重要的方面之一是在整个堆栈中应用“分层安全”。

换句话说，仅仅在主机操作系统上运行防病毒守护进程不足以保护您的应用程序。您需要查看您控制的所有基础架构层，并应用适合您需求的安全监控。

本章中介绍的所有测试都应以第4章中介绍的相同协调模式运行。收集信息以了解当前状态，根据一组规则查找更改，然后使这些更改都适合云原生模式。

如果您可以将您的政策作为代码实施，那么您将超越云的技术优势，并为云原生模式实现商业利益。

历史和地形审计可能看起来并不明显，但随着基础设施的增长以及变化率和应用敏捷性的增加，这些审计将非常重要。将传统方法应用于管理云基础架构不是本地云，本章向您展示了您应该利用的一些好处以及您将面临的一些挑战。

第9章 实施云原生基础设施

如果您认为云原生基础设施是您可以购买的产品或者您可以运行服务器的云提供商，我们很抱歉让您失望。如果不采用这些做法并改变您建设和维护基础设施的方式，您就不会受益。

它不仅仅影响服务器，网络和存储。它关乎的是工程师如何管理应用程序，就像接受故障一样。

围绕云原生实践建立的文化与传统技术和工程组织有很大不同。我们并不是解决组织文化或结构问题的专家，但如果您希望改变组织结构，我们建议您从高绩效组织中实施DevOps实践的角度来看待价值观和经验教训。

一些需要探索的地方是Netflix的文化套餐，它促进了自由和责任感，还有亚马逊的双比萨团队，这些团队以低开销推广自治团体。云原生应用程序需要与构建它们的团队具有相同的解耦特征。康威法律最好地描述了这一点：“设计系统的组织被限制为产生这些组织的通信结构副本的设计。”

在我们结束本书时，我们希望关注哪些领域是您采用云原生实践时最重要的。我们还将讨论一些预测变化的基础设施模式，以便您知道将来要寻找什么。

关注改变的地方

如果您拥有现有的基础架构或传统数据中心，则过渡到云原生不会在一夜之间发生。如果您有足够大的基础架构足迹或两个以上的人员管理基础架构，试图强制实施基础架构管理的新方法可能会失败。

采用这些模式的主要场所令人惊讶地与配置新服务器或购买新软件无关。要开始采用云原生基础设施，首先关注这些领域非常重要：

- 人
- 架构
- 混乱
- 应用程序

直到您准备好这些区域以使用本书中描述的实践，才能开始更改基础架构。

人

> 比竞争对手更快学习的能力可能是唯一的可持续竞争优势。
>
> —Arie de Geus

正如我们在第2章中所讨论的，人们是实施任何变革中最难的部分。他们的抵制有很多原因，而那些要求进行变更以帮助其影响的人有责任。

当需要改变的驱动激励时，变更更容易。主动提高潜力是一个很好的激励因素，但如果没有紧迫感，就很难改变行为。

人们抗拒改变的许多原因来自恐惧。人们喜欢习惯，因为他们感到控制并避免意外。

为了使任何技术取得成功的重大转变，您需要与人们合作以最大限度地减少他们的恐惧。给他们一种主人翁感，并解

释变化的明确目标。确保突出新旧技术之间的相似之处，特别是在改变后他们将扮演的角色。

人们了解这种变化并不是因为他们在旧系统或现有系统中的失败，这一点也很重要。他们需要明白，要求已经改变，环境不同，并且希望他们成为变革的一部分，因为你尊重他们所做的并且对他们能做的事有信心。

他们需要学习新事物，为此，失败是必要的，预期的，并且是进步的标志。

鼓励学习和实验，并奖励适应数据洞察的人员和系统。让工程师通过诸如“百分之二十的时间”之类的自由来探索新的可能性，可以做到这一点。

如果敏捷系统没有改变，它就没有好处。一个不适应和改进的系统将无法满足正在改变和学习的企业的需求。

一旦你能够激发人们寻求改变，你应该用信任和自由来赋予他们力量。不断引导他们将自己的目标与业务需求结合起来，并赋予他们应聘的管理职责。

如果人们已经准备好采用本书中的做法，那么实现它就没有什么限制。关于创建组织变革的更深入的指导，我们推荐阅读John P. Kotter（哈佛商业评论出版社）的“领导变革”。

改变环境文化需要组织的大量努力和支持，以及更改应用程序运行的基础架构。您选择的架构可能会对采用云原生模式的能力产生重大影响。

架构

> 弹性，安全性，可伸缩性，可部署性，可测试性是架构问题。
>
> ——Jez Humble

将应用程序迁移到云原生基础设施时，您需要考虑如何管理和设计应用程序。例如，作为云原生应用程序前身的12因子应用程序受益于在平台上运行。它们被设计为最小化手动管理，频繁更改和弹性。许多传统应用程序的架构都是为了抵制自动化，不经常升级和失败。在移动它之前，您应该考虑应用程序的体系结构。

个人应用程序体系结构是一个问题，但您还需要考虑应用程序如何与基础架构内的其他服务进行通信。应用程序应与云环境中支持的协议以及通过明确界定的接口进行通信通过采用微服务保持应用程序范围很小可以帮助定义应用程序间接口并提高应用程序部署速度。但是，采用微服务会暴露出新的问题，如应用程序通信速度较慢以及分布式跟踪和策略控制网络的需求。不要采用微服务和他们带来的问题 - 而不能在您的基础架构中提供好处。

虽然您几乎可以适应任何应用程序在容器中运行并使用容器编排器进行部署，但如果首选迁移所有关键业务数据库服务器，您将很快得到努力。

首先确定接近具有第1章概述的特性的应用程序并获得在云原生环境中运行它们的经验。一旦您围绕简单的应用程序集体获得经验和良好实践，那么您就可以决定接下来要做什么。

您的服务器和服务也不例外。在将基础架构切换为不可变的之前，您应确保它解决了当前的问题，并意识到新问题。

可靠系统中最重要的架构考虑是争取简单的解决方案。软件和系统自然会变得复杂，这会造成不稳定。在云中，您可以释放对许多区域的控制，因此在仍然可以控制的区域保持简单性很重要。

无论您将内部部署基础架构迁移到云中还是创建新的解决方案，都要确保您在第1章中进行可用性数学计算，并为混乱情况做好准备。

混沌管理

> 拥抱失败并期待混乱。
>
> ——Netflix的Andrew Spyker

当您构建云原生应用程序和基础架构时，其目标是创建最小可行产品（MVP）和迭代.5尝试指定您的客户需要什么或他们将如何使用您的产品可能会有用一段时间，但成功应用程序将适应而不是预测。

客户需求改变;应用程序需要随它们改变。您无法计划和构建完整的解决方案，因为在产品准备就绪时，需求已发生变化。

保持敏捷并在现有技术基础上发展很重要。就像您为应用程序导入库一样，您应该为基础架构使用IaaS和SaaS。你越努力建立自己，你就越能提供价值。

无论何时释放对某物的控制，都会冒着意想不到的风险。如果因为导入的库已更新而导致应用程序中断，您将会知道这是什么感觉。

您的应用程序依赖于库所提供的功能，该功能已更改。你是否应该删除库并编写自己的功能以便控制它？答案几乎总是不。相反，您更新应用程序以使用新库，或者将正在运行的较旧版本的库与应用程序捆绑在一起，以暂时避免损坏。

运行在公共云上的基础架构也是如此。您不再控制硬连线网络，使用什么RAID控制器，或者虚拟机的哪个版本的虚拟机管理程序运行。您所拥有的只是可以在底层技术之上提供抽象的API。

您无法控制底层技术何时发生变化。如果云提供商弃用所有大型内存实例类型，则您别无选择，只能遵守。您要么适应新的尺寸，要么支付更换提供商的成本（时间和金钱）（请参阅附录B关于锁定）。

最重要的是，您构建的基础架构不再可以从单个关键服务器获得。如果您采用云原生基础设施，无论您是否喜欢，您正在构建一个分布式系统。

通过简单地避免失败来保持服务可用的旧做法不起作用。目标不再是您可以设计的最大数目的九个 - 它是可以逃脱的最小数量的九个。

现场可靠性工程以这种方式解释它:

坚持SLO将百分之百满足是不现实的和不可取的：这样做可能会降低创新和部署的速度，需要昂贵的，过于保守的解决方案，或两者兼而有之。相反，最好允许一个错误预算 - 一个可能错过SLO的速率 - 并且每天或每周对其进行跟踪。

工程的目标不可用;它创造了商业价值。你应该制造弹性系统，但不要以过度工程解决方案为代价来避免混乱。

测试更改以防止停机的旧方法也不起作用。为大型分布式系统创建测试环境并不重要。当服务经常更新并且部署是自助服务时尤其如此。

当Netflix每天有4,000次部署或10,000个同时运行的Facebook版本时，对环境进行快照是不可能的。测试环境需要动态分离生产部分。基础架构需要支持这种测试方法，并支持经常测试生产中的新代码所带来的失败。

你可以测试一些混乱（参见第5章），但混沌根据定义是不可预测的。准备您的基础架构和应用程序，以便可预测地对混乱做出反应，而不要试图避免它。

应用程序

基础设施的目的是运行应用程序。如果您的业务完全基于向其他公司提供基础架构，那么您的成功仍取决于其运行应用程序的能力。

如果你建立它，他们不能保证来。您创建的任何抽象需要提高运行应用程序的能力。

> 避免“泄漏抽象”
>
> 抽象并不完全隐藏他们抽象的实现细节。泄漏抽象定律指出：“所有非平凡的抽象在一定程度上都是泄漏的。”
>
> 这意味着你进一步抽象某事，隐藏事物的细节就越困难。
>
> 例如，应用程序资源请求通常通过请求CPU核心的百分比，内存量和磁盘存储量来抽象化。这些资源的物理分配不直接由应用程序管理（API由他们规定），但对资源的请求很明显地表明运行应用程序的系统具有可用的这些类型的资源。
>
> 相反，如果抽象是服务器或数据中心（例如，50台服务器，0.2个数据中心）的百分比，那么抽象并不具有相同的含义，因为不存在单一大小的服务器或数据中心单元。确保创建的抽象对于将使用它们的应用程序有意义。

云原生实践的好处是，随着您提高运行应用程序的能力，您还可以提高运行基础架构的能力。如果您遵循第3-6章的模式，您将很好地适应使用应用程序不断改进和调整您的基础设施以满足应用程序的需求。

重点关注构建范围小，易于适应，易于操作和故障发生时具有弹性的应用程序。确保这些应用程序负责所有基础架构管理和更改。如果你这样做，你将创建云原生基础设施。

预测未来

如果您已经采用了本书中的模式和实践，那么您现在处于未知领域。运行全球最大基础设施的公司已采用这些做法。无论他们运行基础架构的新模式尚未公开披露或仍在发现之中。

好消息是您的基础设施现在被设计为敏捷和变化。您可以更轻松地适应您遇到的任何新挑战。

为了展望未来的道路，我们可以不考虑现有的基础架构模式，而是考虑基础架构在哪里获得灵感 - 软件。几乎所有基础设施采用的模式都来自软件开发模式。

例如，查看分布式应用程序。很多年前，当软件暴露于互联网时，在单一代码库下的单个服务器上运行应用程序不会扩展。这包括管理应用程序的性能和流程限制。

该应用程序需要复制到其他服务器上，然后进行负载均衡以满足需求。当达到限制时，它被分解成更小的组件，创建API以通过HTTP远程调用功能，而不是通过应用程序库。

基础设施采取了类似的方法来扩大规模;它在大多数领域只适应比软件更慢的速度。像Kubernetes这样的现代化基础设施平台将基础设施管理分解成更小的组件。这些组件可以根据其性能瓶颈（CPU，内存和I / O）独立扩展，并且可以快速迭代。

有些应用程序没有相同的扩展需求，也不需要适应新的体系结构。工程师的角色之一是知道何时何时采用新技术。只有那些了解其堆栈的局限性和瓶颈的人才能决定什么是正确的方向。

密切关注新解决方案应用程序在发现新的局限时所开发的内容。如果您能够理解限制条件和变更原因，您将始终走在基础设施的前沿。

结论

本书的目标是帮助您更好地理解云原生基础设施是什么以及您为什么要采用它。虽然我们相信它比传统基础设施有很多优势，但我们不希望您在不理解它的情况下盲目使用任何技术。

不要期望在不采用其伴随的文化和流程的情况下获得所有好处。只是在公共云中运行基础架构或运行容器编排器不会改变该基础架构如何适应的过程。

在亚马逊网络服务中手动创建少量虚拟机，通过SSH连接到每台虚拟机，创建Kubernetes群集比改变人们的工作方式更容易。前者不是云原生的。

请记住，配置基础架构的应用程序不是及时的静态快照;他们应该不断运行并将基础设施推向期望的状态。管理基础设施不是关于维护主机。您需要为资源创建抽象，用API来表示这些抽象，以及使用它们的应用程序。

目标是满足您的应用程序的需求，并编写与您的业务流程和工作职能相当的软件应用程序。随着流程和功能的频繁变化，软件需要轻松适应。

掌握它时，您将不断发现在创建新抽象，保持服务弹性以及推动可伸缩性极限方面的挑战。如果您能够找到新的限制和有用的抽象，请回馈给您所属的社区。

通过研究和社区开放源代码和回馈是这些模式如何从开创它们的环境中涌现出来。共享允许更多的创新和见解传播，并使这些实践的寿命和适应性更容易为每个人。

创新不仅仅来自寻找解决方案或构建下一个伟大的产品。它采取看似不重要的形式提出问题，发言并失败。

请继续做所有这些事情，特别是失败和分享。

附录A 网络弹性模式

在云环境中运行时，应用程序需要具有弹性。特别容易出现故障的一个重要领域是网络通信。添加网络弹性的一种常见模式是创建一个导入到应用程序中的库，该库提供本附录中描述的网络弹性模式。但是，导入的库很难维护以多种语言编写的服务，并且当新版本的网络库发布时，会给应用程序增加测试和重新部署的负担。

代替使应用程序处理网络弹性逻辑，可以将代理置于适当的位置，作为应用程序的保护和增强层。代理的优势在于避免应用程序需要额外的复杂代码，并尽量减少开发人员的工作量最初和持续的发展。

可以在连接层（物理或SDN），应用程序或透明代理中处理网络弹性逻辑。虽然代理不是传统网络堆栈的一部分，但它们可用于透明地管理应用程序的网络弹性。

透明代理可以在基础架构中的任何位置运行，但与应用程序的距离越近越有利。他们还需要在协议中尽可能全面，以及他们可以代理的开放系统互连模型（OSI模型）层。

通过实施以下模式，代理在基础架构的弹性中扮演着积极的角色：

- 负载均衡
- 卸载
- 服务发现
- 重试和截止日期
- 断路

代理也可以用来为应用程序添加功能。一些功能包括：

- 安全和认证
- 路由（入口和出口）
- 洞察力和监测

负载均衡

负载平衡应用程序的方法有很多种，以及为什么您应始终将负载平衡器放在云原生应用程序之前的诸多原因：

DigitalOcean在“5个DigitalOcean负载平衡器使用案例”中解释了一些很好的理由：

- 水平缩放
- 高可用性
- 应用程序部署
- 动态流量路由

协调透明代理（例如envoy和linkerd）是负载平衡应用程序的一种方式。具有透明代理句柄负载平衡的一些好处是：

- 对所有端点的请求视图允许更好的负载平衡决策。
- 基于软件的负载平衡器可灵活选择正确的方式来平衡负载。

透明代理不必盲目地将流量传递给下一个路由器。正如Bunyan在其博客文章“超越循环：负载均衡延迟”中指出的，他

们可以集中协调以对基础设施有更广泛的了解。这使得负载均衡能够全局优化流量路由，而不是仅为本地优化快速分组切换。

随着对端点的更多了解以及哪些服务正在发送和接收请求，代理可以更合理地向哪里发送流量。

加载脱落

“站点可靠性工程”手册解释了卸载与负载平衡不同。尽管负载平衡试图找到正确的后端来发送流量，但是如果应用程序无法接受请求，负载剔除会有意地丢弃流量。

通过删除负载来保护应用程序实例，可以确保应用程序不会重新启动或被迫进入不利条件。删除请求比等待超时并要求重新启动应用程序要快得多。

当某些事情中断或流量过多时，减载可以帮助保护应用程序实例。当事情正常运行时，应用程序应该通过服务发现发现其他相关服务。

服务发现

服务发现通常由运行服务的编排系统处理。透明代理可以绑定到相同的数据并提供附加功能。

代理可以通过将多个源绑定在一起（例如，DNS和键值数据库）并将它们呈现在统一接口中来增强标准服务发现。这允许实现者在不重写所有应用程序代码的情况下改变其后端。如果在应用程序之外处理服务发现，则可以更改服务发现工具而不必重写任何应用程序代码。

由于代理可以对基础架构中的请求提供更全面的视图，因此可以决定端点何时健康与否。这与其他功能配合使用，例如负载均衡和重试，以将流量路由到最佳端点。

当允许服务彼此发现时，代理服务器还可以考虑额外的元数据。他们可以实现逻辑，如节点延迟或“距离”，以确保为请求发现正确的服务。

重试和截止日期

通常，应用程序会使用内置逻辑来知道如何处理对外部服务失败的请求。这也可以由代理无需额外的应用程序代码来处理。

代理拦截应用程序的所有入口和出口流量并路由请求。如果传出请求失败，代理可以自动重试，而无需涉及应用程序。如果请求因任何其他原因返回，则代理可以根据其配置中的规则进行适当处理。

这很好，只要应用程序对延迟有弹性。否则，代理应根据申请截止日期返回失败通知。

截止日期允许应用程序指定允许请求的时间长度。由于代理可以“追踪”到目的地和返回的请求，因此它可以在使用代理的所有应用程序中强制执行最终期限策略。

当超过最后期限时，失败将返回给应用程序，并且可以决定适当的操作。选项可能会降级服务，但应用程序也可能选择将错误发回给用户。

断路

该模式以相同的断路器室内布线命名。当一切正常工作时，电路默认为“关闭”状态，并允许流量流过断路器。当检测到故障时，电路“打开”并打破流量。

> 重试模式使应用程序能够重试操作，以期它会成功。断路器模式阻止应用程序执行可能失败的操作。
>
> —Alex Homer，云设计模式：云应用程序指令性架构指南

断开的电路可以是单个端点或整个服务。打开后，不会发送任何流量，并且所有发送流量的尝试都将立即返回失败。

与家庭电路不同，即使处于开放状态，代理也会测试失败的端点。当检测到故障后再次可用时，可将损坏的端点置于“半开”状态。此状态将发送少量流量，直到端点被标记为失败或健康。

这种模式可以使得应用程序快速失败，并且只能发送到健康端点，从而使应用程序更快。通过不断检查端点，网络可以自行修复并智能地路由流量。

除了这些弹性功能外，代理还可以通过以下方式增强应用程序。

TLS和身份验证

代理可以终止传输层安全性（TLS）或代理支持的任何其他安全性。这使得安全逻辑能够集中管理，而不是在每个应用程序中重新实现。然后可以在整个基础架构中更新新的安全协议或证书，而无需重新部署应用程序。

身份验证也是如此。但是，授权仍应由应用程序管理，因为它通常是更细粒度的应用程序特定功能。在应用程序监督它们之前，用户会话cookie可以由代理验证。这意味着只有通过认证的流量才会被应用程序看到。

这不仅可以节省应用程序的时间，还可以防止某些类型的滥用的停机时间。

路由（入口和出口）

当代理在所有应用程序之前运行时，它们控制流入和流出应用程序的流量。他们还可以管理流入和流出集群的流量。

正如反向代理可以用于在N层体系结构中路由到后端一样，服务代理也可能暴露于集群外部的流量，并用于路由到达的请求。这是代理可以用来知道流量位置的另一个数据点来自何处以及它正在发送的位置。

通过对所有服务到服务通信的深入了解，反向代理意味着可以比传统的反向代理更好地了解路由选择。

Insight和监控

利用所有关于基础设施内流量流的知识，代理系统可以公开关于单个端点和整个集群范围内的流量视图的指标。这些数据点传统上已经在专有网络系统或难以自动化的协议中暴露出来（例如， SNMP）。

由于代理立即知道端点无法访问的时间，所以他们首先知道端点何时不健康。编排系统还可以检查应用程序运行状况，但应用程序可能不知道向编排工具报告正确的状态是不健康的。有了服务于同一服务的所有终端的知识，代理也可以成为监控服务运行状况的最佳位置。

附录B 锁定

关于使用云提供商和避免供应商锁定存在很多争议。这场辩论往往比实际更具意识形态。

锁定通常是工程师和管理层关心的问题。应该以与选择编程语言或框架相同的方式将其作为应用程序的风险来权衡。编程语言和云提供商的选择是锁定的形式，工程师有责任了解风险并了解风险何时可以接受。

当您选择供应商或技术时，请记住以下几点：

- 锁定是不可避免的。
- 锁定是一种风险，但并不总是很高。
- 不要外包思维。

锁定是不可避免的

在技术上有两种类型的锁定：

技术锁定

在开发堆栈中处理较低的决策

供应商锁定

大多数情况下，处理作为项目一部分使用的服务和软件（供应商锁定还可能包括硬件和操作系统，但我们只关注服务）

技术锁定

开发人员将选择他们熟悉的技术或为正在开发的应用程序提供最大利益的技术。这些技术可以从供应商提供的技术（例如.NET和Oracle数据库）到开源软件（例如Python和PostgreSQL）。

在此级别提供的锁定通常要求符合API或规范，这将影响应用程序的开发。有时候可以选择所选择的技术，但是它们通常具有很高的转换成本，因为技术对应用程序的设计有很大影响。

供应商锁定

供应商，如云提供商，是一种不同形式的锁定。在这种情况下，您正在耗用供应商的资源。这可以是基础设施资源（例如，计算和存储），或者它可以是托管软件（例如，Gmail）。

消耗资源的堆栈越高，应该从消耗的资源（例如Heroku）获得的价值就越高。高层次资源是从底层资源中抽离出来最多的，并使产品的生产速度更快。

锁定是一种风险

技术锁定通常是一次性决定或与供应商达成使用该技术的协议。如果您不再与供应商达成支持协议，则您的软件不会立即中断;它只是变得自我支持。

开源软件可以减少来自技术的锁定量，但并不能消除它。使用开放标准可以进一步减少锁定，但了解开放标准与开放源代码之间的差异很重要。

仅仅因为别人编写代码并不能使其成为标准。同样，专有系统可以形成非官方标准，允许从它们迁移出去（如AWS S3）。

供应商锁定的原因通常不仅仅是技术锁定，而是因为供应商锁定的风险高于技术。如果您不支付供应商，您的申请将停止运行;你不再能够访问你所支付的资源。

如前所述，供应商服务提供更多价值，因为它们允许产品开发不需要所有较低级别的实现。不要避免托管服务来消除风险;你应该像对待其他任何事情一样权衡服务的风险和回报。

如果服务提供标准接口，则风险非常低。界面越是自定义，或者产品越独特，切换的风险就越高。

不要外包思维

本书的目标之一就是帮助你自己做出决定。不要盲目听取其他人的意见或报告，不知道咨询的内容以及它如何适用于您的情况。

如果您可以通过使用托管云服务更快地交付产品，则应该选择一个供应商并开始使用它们。虽然衡量风险是好的，但对具有类似解决方案的多家供应商进行无尽的争论，并且自己构建服务并不能很好地利用时间。

如果多个供应商提供类似的服务，请选择最容易采用的服务。开始使用该服务后，限制将很快显现。选择供应商时最重要的因素是选择一个与您具有相同创新步伐的供应商。

如果供应商的创新速度比您快，那么您将无法利用其最新技术，并且可能不得不花大量时间从旧技术中迁移。如果供应商的创新过于缓慢，那么您将不得不根据供应商提供的内容构建自己的抽象，而且您不会专注于您的业务目标。

为了保持竞争力，您可能需要消耗尚未拥有标准或替代品的资源（例如，新的和实验性的服务）。不要因为他们会让你陷入这项服务而感到害怕。重视保持竞争力或失去市场份额的风险，您的竞争对手可能会更快地进行创新。

了解您无法避免的风险以及您的业务有多大风险。做出最大化回报并将风险降至最低的决策

附录C Box：案例研究

> 以下内容最初由CNCF发布在Kubernetes.io上，并且在此获得许可。

在2014年夏天，Box感到十年的硬件和软件基础设施的痛苦，这与公司的需求无法保持一致。

一个平台允许其超过5000万用户（包括政府和大型企业如通用电气公司）管理和共享云中的内容，Box最初是一个庞大的数百万行代码的庞大的PHP代码，内置裸机数据中心。它已经开始在巨石上慢慢消失，分解成微服务。 “随着我们扩展到全球各地，公共云战争正在升温，我们开始专注于如何在许多不同的环境和许多不同的云基础设施提供商之间运行我们的工作负载，”Box联合创始人和服务架构师Sam Ghods。 “迄今为止，这是一个巨大的挑战，因为所有这些不同的提供商，特别是裸机，都有非常不同的界面和与他们合作的方式。”

当Ghods参加DockerCon时，Box的云本土之旅加速了。该公司已经认识到，它不能再仅仅使用裸机来运行其应用程序，并正在研究与Docker的容器化，使用OpenStack进行虚拟化以及支持公共云。

在那次会议上，Google宣布发布Kubernetes容器管理系统，Ghods赢得了胜利。 “我们研究了许多不同的选择，但Kubernetes确实很出色，特别是因为博格老兵的团队非常强大，并且具有完全基础架构不可知的方式来运行云软件，”他说，内部集装箱协调人Borg。 “事实上，第一天它的设计与裸机一样运行，就像我们可以在我们的数据中心内实际迁移到它一样，然后使用相同的工具和概念在公共云提供商上运行，好。”

另外：Ghods喜欢Kubernetes拥有一套通用的API对象，如pod，服务，副本集和部署，这些对象创建了一个一致的表面来构建工具。 “甚至像OpenShift或Deis这样的构建在Kubernetes之上的PaaS层仍然将这些对象视为一流的原则，”他说。 “我们很高兴能够在整个生态系统中共享这些抽象概念，这会产生比我们在其他潜在解决方案中更多的动力。”

Box六个月后在一个生产数据中心的集群中部署了Kubernetes。 Kubernetes在0.11版本之前仍然是预测版。他们从小开始：Ghods的团队在Kubernetes上运行的第一件事就是Box API监视器，它确认了Boxis。 “这只是一个让整个管道运作正常的测试服务，”他说。接下来是一些处理作业的守护进程，它们“很好而且安全，因为如果他们遇到任何中断，我们不会失败来自客户的同步传入请求。”

几个月后，该团队可以发送并要求提供信息的第一个现场服务启动。那时，Ghods说：“我们对Kubernetes集群的稳定性感到满意。我们开始移植一些服务，然后我们将增加集群的大小和端口数量，最后每个数据中心的服务器数量大约为100台，这些服务器纯粹专用于Kubernetes。而且在未来的12个月里，这个数字将会增长很多，达到数百甚至数千。“

在观察开始使用Kubernetes进行微服务的团队时，“我们立即看到正在发布的微服务数量有所增加，”Ghodsnotes说。 “显然，通过微服务构建软件的更好方式已被压抑，并且灵活性的提高帮助我们的开发人员提高了生产力，并为更好的架构选择做好了准备。”

Ghods反映，作为早期采用者，Box与公司现在的经历有不同的旅程。他说：“我们肯定是在等待某些事情稳定或获得释放功能的锁定步骤，”他说。 “在早期，我们对Kubectl应用等组件做了很多贡献，并等待Kubernetes释放它们中的每一个，然后我们会升级，贡献更多，并来回多次。整个项目从我们第一次在Kubernetes上进行实际部署到整体可用性需要大约18个月的时间。如果我们今天完成同样的事情，它可能会少于六个。“

无论如何，Box无需为Kubernetes对公司工作做过多修改。 Ghods说：“我们团队在Box中实施Kubernetes所做的绝大多数工作一直致力于在我们现有的（往往是遗留下来的）基础架构内工作，例如将我们的基础操作系统从RHEL6升级到RHEL7或整合它将纳入我们的监控基础架构Nagios。但总体而言，Kubernetes非常灵活，能够适应我们的许

多限制因素，并且我们在裸机基础架构上非常成功地运行它。“

对于Box来说，更大的挑战也许是文化上的挑战。 Ghods说：“Kubernetes和一般的云本身代表了一个非常大的范式转换，并且它不是非常渐进的，”Ghods说。 “我们基本上是这样说的，Kubernetes将会解决所有问题，因为它能够以正确的方式做事，而一切都会突然变得更好。但是要记住，它不像其他许多解决方案那样可靠。你不能说这家或那家公司花了多少时间做这件事，因为还没有那么多。我们的团队必须真正为资源而战，因为我们的项目有一点点的恐惧。“

从经验中学习，Ghods为经历类似挑战的公司提供了以下两条建议：

1. 提前和经常交付。对于Box来说，服务发现是一个巨大的问题，团队必须决定是建立一个临时解决方案还是等待Kubernetesto本身满足Box的独特要求。经过多次辩论之后，“我们刚开始专注于提供可行的解决方案，然后处理可能在稍后迁移到更原始的解决方案，”Ghods说。 “无论多么微不足道，团队的上述目标应始终是为基础架构上的实际生产用例服务。这有助于保持团队本身和组织对项目的看法。“
2. 保持开放的态度，了解公司必须从开发人员那里抽象出什么，以及什么没有。早期，团队在Dockerfiles之上构建了一个抽象，以帮助确保所有容器映像具有正确的安全更新。事实证明这是多余的工作，因为容器映像是不可变的，您可以在构建后扫描它们以确保它们可以不包含漏洞。因为通过集装箱化来管理基础设施是一个不连续的飞跃，所以最好先直接使用本地工具并学习其独特的优势和注意事项。抽象只能在实际需要出现之后才能建立。

最后，影响力非常强大。 “在Kubernetes之前，”Ghods说，“我们的基础设施非常陈旧，需要6个多月才能部署一个新的微服务。现在，一个新的微服务部署时间不到五天。我们正在努力让它达到不到一天。诚然，这六个月的大部分时间都是由于我们的系统有多么糟糕，但裸机本质上是一个难以支持的平台，除非您有像Kubernetes这样的系统来帮助管理它。“

按Ghods的估计，Box距离他成为90％ Kubernetes店的目标还有几年的时间。 “到目前为止，我们已经完成了一项稳定的，关键任务的Kubernetes部署，它提供了很多价值，”他说。 “现在我们所有电脑的10％左右都运行在Kubernetes上，我认为明年我们可能会超过一半。我们正在努力实现所有无状态服务使用案例，并计划在此之后将我们的重点转移到有状态服务。“

事实上，这就是他在整个行业中的设想：Ghods预测Kubernetes有机会成为新的云平台。 Kubernetes提供了一个涵盖不同云平台的API，包括裸机，以及“当我们可以针对单一界面进行编程时，我不认为人们已经看到了可能的全部潜力”，他说。 “与AWS改变基础架构一样，您不必再考虑服务器或机柜或网络设备，Kubernetes使您能够专注于您正在运行的软件，这非常令人兴奋。这是愿景。“

Ghods指出了已经在开发或最近发布的作为云平台的项目：集群联合，仪表板UI和CoreOS’setcd运营商。 “我真的相信这是我在云基础架构中看到的最激动人心的事情，”他说，“因为它是一个前所未有的自动化和智能环境，其基础设施对每个基础设施平台都是可移植和不可知的。”

由于早期决定使用裸机，Box不得已开始了Kubernetes之旅。但是Ghods表示，即使公司现在不必对云提供商不可知，Kubernetes也可能很快成为行业标准，因为越来越多的工具和扩展是围绕API构建的。

“同样的方式，偏离Linux是没有意义的，因为它是如此的标准，”Ghods说，“我认为Kubernetes正在走相同的道路。现在还处于早期阶段 - 文档仍然需要工作，用于编写和发布规格到Kubernetes集群的用户体验仍然很艰难。当你处于尖端时，你可能会出血一点。但底线是，这是行业发展的方向。从现在开始的三到五年，如果以其他方式运行基础设施，真的会非常震惊。“