

O'REILLY®

# Terraform

多云、混合云环境下实现基础设施即代码  
( 第2版 )

Terraform: Up & Running, 2nd Edition

[美] Yevgeniy Brikman 著  
白宇 译

电子工业出版社  
Publishing House of Electronics Industry  
北京•BEIJING

## 内 容 简 介

这本书介绍了如何通过 Terraform 在多云和混合云的环境下使用基础设施即代码，把软件工程的优秀实践应用于硬件的管理。本书第 2 版涵盖了 Terraform 0.12 版本的重要升级，书中通过大量的代码示例，介绍了 Terraform 的基本功能、企业级模块化部署、自动化测试，以及团队环境下使用基础设施即代码的开发部署流程。本书不仅充分展现了 Terraform 作为一种基础设施即代码工具的魅力，还通过多角度的对比，使读者能够准确把握如何在实战中使用和配置该软件。

系统管理员、DevOps 工程师、开发人员和云服务技术从业者，都能从本书中找到所需要的知识与指导。

© 2019 by Yevgeniy Brikman. All rights reserved.

© 2021 year of first publication of the Translation Publishing House of Electronics Industry Co., Ltd.

Authorized translation of the English edition of Terraform: Up & Running, 2nd Edition © 2019 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

本书简体中文版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2020-3481

## 图书在版编目（CIP）数据

Terraform：多云、混合云环境下实现基础设施即代码：第 2 版 / (美) 叶夫根尼·布里克曼 (Yevgeniy Brikman) 著；白宇译. —北京：电子工业出版社，2021.1

书名原文：Terraform: Up & Running, 2nd Edition

ISBN 978-7-121-40022-3

I. ①T… II. ①叶… ②白… III. ①程序语言 IV. ①TP312

中国版本图书馆 CIP 数据核字（2020）第 234826 号

责任编辑：张春雨

封面设计：Karen Montgomery 张健

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：23.5 字数：400 千字

版 次：2021 年 1 月第 1 版（原书第 2 版）

印 次：2021 年 1 月第 1 次印刷

印 数：2500 册 定价：108.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：010-51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# O'Reilly Media, Inc.介绍

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

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

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

## 业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

---

## 译者序

当我在 2019 年 7 月第一次看到本书第 2 版的预发行版本时，就深深地被它的内容所吸引，并下定决心一定要将这本不可多得的、关于 Terraform 和基础设施即代码（IaC, Infrastructure as Code）的优秀出版物介绍给国内的读者。

作为一名在软件工程领域摸爬滚打近 20 年的从业人员，我在感叹各种技术快速迭代的同时，也深深体会到软件工程领域发展的周期性特点，每 3 到 5 年，就会有新的热点名词出现：Scrum、CI/CD（持续集成）、SRE、中台、AI，等等；每 10 年就会有一个大的转型：Agile（敏捷开发）、DevOps、云计算、机器学习。这些让人眼花缭乱的热点名词，每一个都代表着对一类问题的思考，以及解决这类问题的方法论。它们从出现开始，经历着被质疑、被接受、被追捧，直到被取代的过程。这个过程也标志着软件工程领域对已有难题的解决和继续对未知领域的探索。每一个热点的出现，往往代表着新的机遇、新的挑战，有时甚至是整个行业的洗牌。所以我们在追捧名词和追赶“明星技术”的同时，也要冷静地去挖掘其背后的根本问题，这样才能在五花八门的解决方案中，做出正确的选择。

软件开发自上世纪 60 年代出现后，经历了快速的发展：从结构化编程（以瀑布模型为代表）到敏捷开发；在融合了极限编程、Scrum、DSDM、FDD 等当时流行的软件开发实践的理念后，在 2001 年的雪鸟会议<sup>1</sup>上，著名的《敏捷宣言》被提出。宣言重申了敏捷开发将专注于团队成员间的协作、客户需求的及时响应、软件产品的快速迭代。重点关注在软件开发领域中如何加强产品项目部门和软件开发团队之间的紧密协作，如何将客户的需求快速、准确地转化为软件产品。敏捷开发的出现大大提高了软件开发的效率。

---

<sup>1</sup> 美国犹他州的 Snowbird 滑雪胜地。

在敏捷开发相对成熟后，下一个 10 年是 DevOps 运动的兴起。DevOps 公认起源于 2009 年的 Velocity 大会，可以被看作敏捷开发向 IT 和运维领域的延伸。DevOps 关注如何通过对工具、文化和流程的改进，提升部署自动化和发布的效率。DevOps 兴起到成熟的过程，也是软件开发团队和产品运维团队间直接协作与融合的过程。其间也有不同的侧重点：行业最初主要关注 CI/CD，甚至 DevOps 一时成为持续集成的代名词；后来又变成自动测试——提高代码测试覆盖率、减少手动测试、增加自动测试，自动测试成了 DevOps 工程师所追求的目标；再后来，在很多场合下，DevOps 又被称作 DevSecOps，很显然，安全性成了 DevOps 要解决的问题；再后来又提出了治理与合规（Governance and Compliance）、AIOps、ChatOps，等等。但是一个贯穿始终的关注点就是：运维（Ops）。

DevOps 的重点之所以是运维，一个原因是在开发团队实施敏捷开发后，运维团队成了新的瓶颈；另一方面是运维领域的硬件虚拟化（服务器变成了虚拟机，数据中心变成了云计算平台）所带来的挑战——硬件的搭建和改变更加频繁，使原有的思想和技术很难继续支持业务的发展。虚拟化一方面增加了运维团队的压力，另一方面也为运维团队 DevOps 的改进提供了机遇——那就是将基础设施代码化，使用代码对硬件进行管理，在运维领域借用软件领域的最佳实践，将基础设施的运维纳入软件工程的范畴，最终整体改善软件开发和软件交付的过程。

运维团队要适应的另一个变化是云计算。什么是云计算？如果形象地进行比喻，云计算是企业 IT 服务中的“共享经济”。因为云计算完全符合共享经济的定义：通过互联网进行的、充分利用闲置资源的、按需分配下的重复使用。无论是私有云上组织内部门间的资源共享，还是公有云上不同公司之间的资源（算力、基础设施、安全、冗余、融灾）共享，都把共享经济的特点体现得淋漓尽致。而且随着人工智能、机器学习等领域的快速发展出现的算力匮乏，以及互联网经济活动（例如“双十一”）对运营伸缩能力的刚性需求，还有各种复杂架构系统导致的维护成本的增加，使人们对云服务的需求也从基础设施即服务（IaaS），演进到平台即服务（PaaS），再到最新的软件即服务（SaaS），无论在广度上还是深度上都有了更高的要求。所以说，云计算作为共享经济在企业 IT 服务领域的落地，注定会进一步呈现百花齐放式的发展，多云和混合云是不可避免的趋势。

在多云和混合云的环境下，使用基础设施即代码（以下称 IaC），如何把软件工程的最佳

实践，用于管理云相关的运维活动，就成了一个迫切需要解决的问题。而 Terraform 就是针对这个问题而出现的一个解决方案。作为 HashiCorp 公司开发的一种用于多云和混合云环境的 IaC 工具，Terraform 专注于服务开通功能。虽然 Terraform 还很年轻（处于 pre-1.0 版本），但是近年来有了突飞猛进的发展（请参阅第 1 章表格 1-2），这就导致了相关书籍、培训的短缺，而本书则是一本不可多得的、关于 Terraform 入门和实战内容的精品。

首先，这是一本贴近工程实战的教材，没有照搬官方手册，没有局限于介绍各种参数，而是突出了企业级、工程领域的实战操作。除基本功能外，它还突出了使用 Terraform 时的安全性、重用性、可维护性和可拓展性。例如，本书的第 6 章，从模块的复用角度出发，介绍了生产级 Terraform 代码的编写；第 7 章更是突出了如何在企业工程环境下测试 Terraform 代码；第 8 章介绍了如何在大型团队中使用 Terraform 协同开发基础设施代码。

其次，这是一本需要动手操作的教程。本书精心准备数十个代码示例，从动手编写“Hello,world”开始，逐步深入，到最后实现一个部署整套服务器集群的脚本，以及相关的测试。它使读者能够在短短几章内，超越运行简单介绍性示例的层次，深刻体会 Terraform 实战的特点。

还有就是，本书的第 1 章并没有匆忙地介绍如何安装和使用软件。而是退后一步，从更高的角度介绍了 DevOps、IaC 的最佳实践，以及 IaC 工具的分类：配置管理（configuration management）、编排（orchestration）、服务开通（provisioning）和服务器模板（server templating），并对广泛使用的 IaC 工具进行了多个维度的比较。

本书作者本身是一位软件工程和软件开发领域的专家，所以并没有仅仅将 Terraform 作为一个应用软件来进行介绍，而是在讲解工具的同时，穿插了大量的 DevOps 和 IaC 的最佳实践、开发人员日常使用的小技巧，以及周边的阅读和学习资源（请参考附录 A），极大地拓展了读者的知识面。作为第 2 版，本书经过第 1 版出版后的反馈，以及 Terraform 软件本身的演进，增加了很多读者希望看到的内容，以及对最新发布版本的支持（0.12 版本包含重大语法变化）。具体信息请参考前言：第 2 版新增内容。

云计算、IaC 和 DevOps 这些热门术语，在 2009 年前后才出现，而诸如 Terraform、Docker、Packer 和 Kubernetes 之类的工具在 2018 年前后才稳定发布。所有这些相对较新的工具和

技术都在迅速地变化着，这也意味着它们并不特别成熟，也缺乏经验丰富的从业人员。随着运维团队的主要任务逐渐从硬件管理转移到软件管理上，加上云计算经过了一段时间的积累，将在今后 3~5 年内再一次出现爆炸性增长（企业服务领域），如何在多云和混合云下使用 IaC 管理基础设施，将成为下一个热点。这次你能把握住机会吗？

感谢 Jim（本书作者 Yevgeniy Brikman）给我们带来了这本精彩图书的第 2 版。感谢博文视点的编辑能够把这个优秀的作品引进到国内，并让我有机会参与翻译工作。感谢所有为本书出版做出贡献的人。

白宇

2020 年 1 月

美国硅谷

---

# 前言

很久很久以前，有一个遥远的数据中心。数据中心里有一群被称为系统管理员（sysadmin）的“远古生物”。它们虽然能力强大，但还在用手动方式对基础设施进行部署。它们手动创建和管理整个数据中心的每台服务器、每个数据库、每个负载均衡器，甚至网络配置的每个参数。这是一个黑暗而令人恐惧的年代：人们害怕停机，害怕意外的错误配置，害怕漫长而脆弱的部署工作，担心如果有一天系统管理员突然失踪（休假），该怎么办？一个好消息是，通过 DevOps 的推动，现在，人们的种种担忧有了一个更好的解决方案：*Terraform*。

*Terraform*（见参考资料文前[1]）是一个由 HashiCorp 公司创建的开源工具，它可以让用户使用简单的声明性语言将基础设施定义为代码，并通过一些命令来部署和管理基于各种公有云（例如 Amazon Web Service、Microsoft Azure、Google Cloud Platform、DigitalOcean）、私有云和虚拟化平台（例如 OpenStack、VMWare）的基础设施。例如，与手动点击网页或运行数十个命令相比，下面是在 AWS（Amazon Web Service）上使用 *Terraform* 配置一台服务器所需要的全部代码。

```
provider "aws" {
    region = "us-east-2"
}

resource "aws_instance" "example" {
    ami      = "ami-0c55b159cbfafef0"
    instance_type = "t2.micro"
}
```

要实际部署它，只需要运行以下命令即可。

```
$ terraform init  
$ terraform apply
```

由于其简单、易用且功能强大，Terraform 已成为 DevOps 领域中的一个关键角色。它可以自动化那些烦琐而脆弱的手动工作，使用户拥有一整套坚实的基础设施，并在此基础之上继续构建其他的 DevOps 实践（如自动测试、持续集成、持续交付）和工具（如 Docker、Chef、Puppet）。

对于学习和使用 Terraform 来说，本书是入门和进阶的最快方法。

我们将从头开始部署 Terraform 最基本的“Hello,World”示例（实际上，你已经看到了），直到最后运行一套支持大流量和庞大开发团队的完整技术栈（服务器集群、负载均衡器、数据库），完成这一切只需几章内容即可。本书是一个需要动手的教程，它不仅介绍了 DevOps 和 IaC 的原理，还将带你一起实践几十个在家中就可以完成的代码示例，因此请确保有可以使用的计算机，以便随时进行实践。当完成这些学习和实践内容之后，你将具备 Terraform 的实战能力。

## 谁应该阅读本书

在代码编写完成之后还需要对代码负责的任何人，都适合阅读本书。其中包括系统管理员、运营工程师、发布工程师、站点可靠性工程师、DevOps 工程师、基础设施开发人员、全栈开发人员、工程经理和 CTO。无论头衔是什么，如果你的职责是管理基础设施、部署代码、配置服务器、管理服务器集群的缩放、备份数据、监控应用程序，并需要在凌晨 3 点响应任何系统报警，本书就是写给你的。

总的来说，所有这些任务统统被称为运维。在过去，熟悉如何编写代码但不了解运维的开发人员是很常见的。同样，精通运维但不懂如何编写代码的系统管理员也不在少数。从前我们可以回避这种岗位之间的隔阂问题，但是在“现代世界”中，随着云计算和 DevOps 变得无处不在，几乎每个开发人员都需要了解运维技术，同样每个系统管理员也需要学习编码技能。

本书的读者并不需要是资深程序员或系统管理专家。只需要拥有最基本的编程知识，会使用命令行，了解基于服务器的软件（如网站）如何运行即可。学习所依赖的其他所有内容，在书中都已经涵盖。当完成本书学习时，你将充分了解现代开发和运维的最关键技术之一：将基础设施作为代码进行管理。

通过阅读本书，读者不仅可以学习如何通过 Terraform 将基础设施作为代码进行管理，还会了解如何让它适用于整个 DevOps 领域。在本书结尾处，我希望你将有足够的知识回答以下问题。

- 为什么要使用 IaC（基础设施即代码）
- 配置管理（configuration management）、编排（orchestration）、服务开通（provisioning）和服务器模板（server templating）之间的区别是什么
- 什么时候应该使用 Terraform、Chef、Ansible、Puppet、Salt、CloudFormation、Docker、Packer 或 Kubernetes
- Terraform 是如何工作的，以及如何使用它来管理基础设施
- 如何创建可重复使用的 Terraform 模块
- 如何编写足够可靠的生产级 Terraform 代码
- 如何测试 Terraform 代码
- 如何使 Terraform 成为自动化部署过程的一部分
- 团队中使用 Terraform 的最佳实践是什么

学习本书需要的工具是计算机（Terraform 可以在大多数操作系统上运行）和互联网，当然，求知的欲望也必不可少。

## 我为何撰写本书

Terraform 是一个功能强大的工具，兼容所有主流云提供商。它使用简洁的语言，并且对重用、测试和版本控制提供强大的支持。Terraform 作为一个开源项目，拥有一个友好且活跃的社区。如果一定要让我找出它的不足之处，那就是在成熟度上的欠缺。

Terraform 还是一个比较新的技术。截至 2019 年 5 月，它尚未发布 1.0.0 版本，尽管 Terraform

越来越得到业界的推崇，但仍然很难找到相关书籍、博客文章或专家来帮助你掌握这个工具。Terraform 官方的文档在介绍基本语法和功能方面做得很好，但对惯用模式、最佳实践、测试、可重用性或团队工作流程的相关内容涉及得不够。后者的重要性就好比当你试图流利地讲法语时，不仅应该学习词汇，还应该熟练掌握语法和惯用语一样。

我写本书的目的是帮助开发人员精通 Terraform。在 Terraform 面世后的 5 年中，有 4 年的时间我都在自己的公司 Gruntwork( 见参考资料文前[2] )里使用这个工具。我们将 Terraform 作为核心工具，创建了包含 30 万行代码的可重复使用的模块库，这些基础设施代码经受了数百家公司生产环境的实战检验。多年以来，我们编写并维护的基础设施代码在许多不同的公司里得到应用，这种经历也给我们带来了很多的教训。我的目标是：通过与你分享这些经验和教训，缩短这个漫长的学习过程，使你能够在几天之内熟练掌握 Terraform。

为了能够流利地说法语，只进行阅读训练是完全不够的，你还需要花时间与讲法语的人交谈、观看法语电视节目、听法语音乐。同样，为了能够熟练使用 Terraform，用户需要编写真实的 Terraform 代码，使用它来管理真实的软件，然后将该软件部署在真实的服务器上。因此，请准备好在学习本书过程中阅读、编写和执行大量的代码。

## 本书包含如下内容

下面是本书的大纲。

### 第 1 章 为什么使用 Terraform

DevOps 是如何改变我们运行软件的方式的；IaC 工具的介绍，包括配置管理、服务器模板、编排和服务开通工具；IaC 的好处；Terraform、Chef、Puppet、Ansible、SaltStack、OpenStack Heat 和 CloudFormation 的比较；如何配合使用 Terraform、Packer、Docker、Ansible 和 Kubernetes 等工具。

### 第 2 章 Terraform 入门

Terraform 的安装，Terraform 的语法概述，Terrform 的命令行工具；如何部署单个服务器；如何部署单个 Web 服务器；如何部署 Web 服务器集群；如何部署负载均衡器；如何清理创建的资源。

## 第 3 章 如何管理 Terraform 的状态

什么是 Terraform 的状态；如何存储状态文件，以便于团队成员之间进行共享访问；如何锁定状态文件，防止出现竞争的状况；如何使用 Terraform 管理密码；如何隔离状态文件，减少错误造成的损失；如何使用 Terraform 工作区；在 Terraform 项目内，文件和文件夹布局的最佳实践；如何使用只读状态。

## 第 4 章 使用 Terraform 模块创建可重用基础设施

什么是模块（module）；如何创建一个基本模块；如何使用输入和输出变量构建可配置的模块；局部变量；模块的版本控制；模块使用中的陷阱；使用模块实现可重用的、可配置的基础设施。

## 第 5 章 Terraform 技巧和窍门：循环、if 表达式、部署和陷阱

通过 count 参数、for\_each 表达式、for 表达式和 for 字符串指令进行循环；通过 count 参数、for\_each 表达式、for 表达式和 if 字符串指令实现条件判断；内置功能；零停机部署；常见的 Terraform 陷阱，包括：count 和 for\_each 的限制、零停机部署中的陷阱、为什么通过验证的计划也会失败、重构问题和最终的一致性问题。

## 第 6 章 生产级 Terraform 代码

为什么 DevOps 项目花费的时间比预期的更长；生产级基础设施检查清单；如何构建用于生产环境的 Terraform 模块——小型的模块、可组合的模块、可测试的模块、可发布的模块、Terraform 注册中心、Terraform “逃生舱口” 等。

## 第 7 章 如何测试 Terraform 代码

手动测试 Terraform 代码，沙箱环境和清理，自动测试 Terraform 代码，Terratest，单元测试，集成测试，端到端测试，依赖注入和并行测试，测试阶段，重试，测试金字塔，静态分析，属性检查。

## 第 8 章 如何在团队环境下使用 Terraform

如何在团队环境下部署 Terraform，如何说服你的老板采纳 Terraform，应用程序代码的部署流程，基础设施代码的部署流程，版本控制，Terraform 的黄金法则，代码评审，编码准则，Terraform 的样式，Terraform 的 CI/CD，部署过程。

读者可以从头到尾阅读本书，也可以直接跳到你最感兴趣的章节。请注意，每章的示例都以之前章节的示例作为基础。因此，如果略过之前的章节，请参考使用完整的开放源代码示例（如第 xv 页的“开放源代码示例”中所述的那样）。在本书最后的附录 A 中，包括推荐阅读的文章列表，你可以找到更多有关 Terraform、运维、IaC 和 DevOps 的资源。

## 第 2 版的新增内容

第 1 版发布于 2017 年。2019 年 5 月，我完成了第 2 版的编写。在短短几年内，Terraform 发生了令人惊讶的变化！所以第 2 版几乎是第 1 版篇幅的两倍，包括两个完整的新章节，所有原始章节和代码示例也进行了很大程度的更新。

如果你是第 1 版的读者，仅仅想知道第 2 版有什么新功能，或者只是想看看 Terraform 是如何演化的，那么下面是我总结的一些关注要点。

### Terraform 发布了 4 个主要版本

在第 1 版首次发行时，Terraform 的版本为 0.8。从那时起到本书第 2 版首次发行之间，Terraform 已经发布了 4 个主要版本，现在的版本为 0.12。这些版本引入了许多新功能，并且为用户升级做了大量准备工作<sup>1</sup>！

### 自动测试的改进

Terraform 的自动测试工具和实践已经有了很大的发展。第 7 章是一个全新的关于测试的章节，涵盖了单元测试、集成测试、端到端测试、依赖注入、测试并行性、静态分析等主题。

### 模块的改进

创建 Terraform 模块的工具和实践也有了突飞猛进的发展。在全新的第 6 章中，介绍了如何构建一个可重用的、实战检验过的生产级 Terraform 模块，这种模块是每个公司都希望使用的。

---

<sup>1</sup> 请参考 Terraform 升级指南（见参考资料文前[3]）。

## 工作流程的改进

第 8 章是完全重写的，介绍了团队该如何将 Terraform 集成到工作流程中，其中详细地介绍了如何将应用程序代码和基础设施代码从开发、测试，一路发布到生产环境中的过程。

## HCL2

在 Terraform 0.12 版本中，将基础语言从 HCL 全面升级到 HCL2。升级包括对第一类表达式的支持（这样就不需要将变量包装在 \${...} 中了），丰富的类型限制，惰性计算的条件表达式，对 `null`、`for_each` 和 `for` 表达式、动态内联块等的支持。本书的所有代码示例已经更新为 HCL2，新的语言功能将在第 5 章和第 6 章中重点讲述。

## Terraform state 功能的主要提升

在 Terraform 0.9 版本中，后端成为存储和共享 Terraform 状态文件的主要方式，包括对锁定功能的内部支持。Terraform 0.9 把状态环境作为跨环境部署的管理方式。在 Terraform 0.10 版本中，状态环境又进一步被 Terraform 工作区的概念所替换。我将在第 3 章中介绍所有这些主题。

## Terraform provider 的拆分

在 Terraform 0.10 版本中，Terraform 核心代码与所有提供商的代码（即 AWS、GCP、Azure 等的代码）被拆分并存储于不同的代码库中。这样，每个提供商就可以根据自己的项目节奏，在独立的代码库中开发新版本了。随之而来的变化是，当每次引入一个新的模块时，必须重新运行 `terraform init` 命令来下载提供商的代码，我们将在第 2 章和第 7 章中进行介绍。

## 提供商大规模的增长

自 2016 年以来，Terraform 已从只支持少数几个主要的云服务提供商（例如 AWS、GCP 和 Azure）发展到现在支持 100 多家官方提供商，以及更多的来自社区的提供商<sup>1</sup>。这意味着用户现在不仅可以使用 Terraform 管理更多其他类型的云平台（例如 Alicloud、Oracle Cloud Infrastructure、VMware vSphere 等），还可以通过 Terraform

---

<sup>1</sup> 你将在参考资料文档[5]看到 Terraform 所有提供商列表。

将云平台之外的系统作为代码进行管理：包括版本控制系统（例如 GitHub、GitLab 或 BitBucket）、数据存储系统（例如 MySQL、PostgreSQL 或 InfluxDB）、监视和警报系统（例如 DataDog、New Relic 或 Grafana）、平台工具（例如 Kubernetes、Helm、Heroku、Rundeck 或 Rightscale），等等。另一方面，每个服务提供商的 API 接口覆盖率都有了显著提高，例如，Terraform 的 AWS 服务提供商现在涵盖了大多数重要的 AWS 服务，甚至当新功能出现时，其对 Terraform 的支持早于 AWS 自有的 CloudFormation！

### Terraform Registry

HashiCorp 于 2017 年启动了 Terraform Registry（见参考资料文前[4]）服务，通过网站界面，你可以轻松浏览和使用来自开源社区的可重复使用的 Terraform 模块。2018 年，HashiCorp 进一步开始支持在组织内运行 Private Terraform Registry 的功能。Terraform 从 0.11 版本开始，可以直接引用 Terraform Registry 中发布的模块。我们将在第 6 章的“可发布的模块”一节中进行介绍。

### 更好的错误处理

在 Terraform 0.9 版本中，对状态错误的处理机制进行了更新。如果 Terraform 写入远程后端状态文件时出错，该状态将被保存在本地的 `errored.tfstate` 文件中。在 Terraform 0.12 版本中，又对错误处理进行了彻底的改进，使其能够更早地捕获错误，显示更清晰的错误消息，并在错误消息中包含文件路径、行号和部分代码段。

### 其他细微变化

还有许多其他细微的变化，包括对局部变量的支持（见第 4 章“模块的变量”）；新的“逃生舱口”功能，使 Terraform 能够通过脚本与外界进行互动（见第 6 章“Terraform 模块之外的内容”）；将 `plan` 命令作为 `apply` 命令的一部分来运行（见第 2 章“部署单个服务器”）；修复了 `create_before_destroy` 循环问题；对 `count` 参数进行重大改进，使其能够对数据源和资源进行引用（见第 5 章“循环”）；新增加了数十个内置函数，以及对 `provider` 继承功能的全面改进。

## 在本书中找不到的内容

本书并不打算成为一份详尽的 Terraform 参考手册。我也没有介绍所有的云服务提供商，以及每个提供商支持的全部资源或每个可用的 Terraform 命令。有关这些实质性细节，请参考 Terraform 官方文档<sup>1</sup>。

官方文档包含了许多有用的答案。但是如果熟悉 Terraform、IaC 或运维的概念，你或许都不知道该问些什么问题。因此本书的重点是官方文档未涵盖的内容，使读者能超越运行介绍性示例的层次，在实战中使用 Terraform。通过讨论为什么要使用 Terraform，如何使其搭配你的工作流程，以及哪种最佳实践和应用模式是最有效的，使你能够快速地起步并开始使用 Terraform。

为了能够演示这些模式，我在书中提供了许多代码示例。我已经将代码对第三方系统的依赖降到最低，使你能够轻松地在家中实验这些示例。本书所有示例都只涉及唯一的云提供商：AWS，你只需要注册一个第三方服务就可以完成全部实验（AWS 提供了免费账号，因此运行示例代码不应该产生任何费用）。由于这个原因，本书和示例代码将不包括 HashiCorp 公司的其他付费服务产品——Terraform Pro 和 Terraform Enterprise。我已将所有代码示例进行了开源发布。

## 开源代码示例

你可以在以下网址找到本书所有代码示例：

<https://github.com/brikis98/terraform-up-and-running-code>

在开始之前请将代码复制到本地计算机上，以便在你自己的计算机上练习所有示例。

`git clone https://github.com/brikis98/terraform-up-and-running-code.git`

代码示例分章节存储在 GitHub 库中，大多数示例的内容和书中各章结尾处的代码保持一致。如果想要达到最佳学习效果，请亲自动手从头编写每章的代码。

---

<sup>1</sup> 见参考资料文前[6]。

读者将从第 2 章开始编写代码，学习使用 Terraform，从头开始部署一个 Web 服务器集群。在之后的各章节中，你将对同一个 Web 服务器集群进行开发和改进。

请按照步骤说明亲自动手进行相应的代码修改，把 GitHub 库中的代码示例仅仅作为修改后的验证手段，或者在自己陷入困境时参考使用。



#### 关于版本的注释

本书所有的示例都针对 Terraform 0.12.x 版本（该版本是撰写本书时最新的主要版本）进行了测试。因为 Terraform 是一个相对较新的工具，尚未发布 1.0.0 版本，所以未来的发行版可能不兼容现在的内容，并且某些最佳实践会随着时间的推移而发生更改和演进。我将尝试尽可能多地发布代码更新。由于 Terraform 项目进展非常快，因此你需要通过学习才能跟上它的变化，例如关于 Terraform 的新闻、博客文章和 DevOps Terraform 方面的讲座，请务必查看本书的网站<sup>1</sup>并订阅新闻通讯<sup>2</sup>！

## 使用代码示例

本书将对你的工作有所帮助。一般来说，本书提供的示例代码，你都可以在你的程序和文档中使用。你不必联系我们获得许可，除非你要大量传播代码。例如，从本书中抄录一些代码用于编写程序不需要许可，销售或分销 O'Reilly 随书附带的代码则需要许可；引用本书中的内容和示例用于回答问题不需要许可，将本书中的大量示例代码插入你的产品文档中则需要许可。

我们感谢但不要求注明出处。出处的格式一般包括标题、作者、出版商和 ISBN。例如，“*Terraform: Up and Running, Second Edition* by Yevgeniy Brikman (O'Reilly). Copyright 2017 Yevgeniy Brikman, 978-1-492-04690-5.”。

如果你觉得示例代码的使用不合法或不符合以上的许可权限，请随时联系我们：[permissions@oreilly.com](mailto:permissions@oreilly.com)。

---

<sup>1</sup> 见参考资料文前[7]。

<sup>2</sup> 见参考资料文前[8]。

# 排版约定

本书使用了下列排版约定。

- 中文楷体或英文斜体（*Italic*）  
表示新的术语、URL、邮件地址、文件名称和文件扩展名。
- 等宽字体（`constant width`）  
表示程序片段，也可以用在正文中表示变量名或函数名等程序元素、数据库、环境变量、语句和关键字等。
- 等宽字体加粗（**Constant width bold**）  
表示应该由用户输入的命令或其他文本。



该图标表示一般注释。



该图标表示警告或注意。

## 参考资料说明

本书提供了大量的参考资料以方便读者更好地了解书中提到的相关技术及工具。为了保证参考资料相关链接能够实时更新，特将“参考资料”文档放于博文视点官方网站，读者可在 <http://www.broadview.com.cn/40022> 页面进行下载。

## O'Reilly 在线学习平台 (O'Reilly Online Learning)

**O'REILLY**® 近 40 年来, *O'Reilly Media* 致力于提供技术和商业培训、知识和卓越见解, 来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络, 他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。*O'Reilly 在线学习平台*允许你按需访问现场培训课程、深入的学习路径、交互式的编程环境, 以及 *O'Reilly* 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息, 请访问 <http://oreilly.com>。

## 如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者:

美国:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国:

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)  
奥莱利技术咨询 (北京) 有限公司

*O'Reilly* 的每一本书都有专属网站, 你可以在那里找到关于本书的相关信息, 包括勘误列表、示例代码, 以及其他信息。本书的网站地址: <http://bit.ly/terraform-up-and-running-2e>。

对于本书的评论和技术性的问题, 请发送电子邮件到: [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)。

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

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

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

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

## 致谢

### Josh Padnick

没有你，这本书是不可能完成的。你是第一个向我介绍 Terraform，教给我基础知识并帮助我掌握高级内容的人；感谢你在我共同学习及撰写本书过程中给予我的支持；感谢你作为一位了不起的联合创始人，让我们在经营一家初创公司时还能够享受快乐的生活。非常感激能和你成为好朋友。

### O'Reilly Media

感谢你们出版了我的又一本书。阅读和写作深刻地改变了我的生活，我很荣幸能通过你们与他人分享我的一些著作。特别感谢 Brian Anderson 帮助我以创纪录的时间出版了本书的第 1 版，以及 Virginia Wilson 在发行第 2 版时竟然打破了这一纪录。

### Gruntwork 的员工

我非常感谢你们加入我们这个小型初创公司，一起开发出色的软件，在我编写第 2 版时替我坚守岗位，并且让我们成为了很棒的同事和朋友。

### Gruntwork 的客户

感谢你们的信任，与一家不知名的小公司进行合作，并自愿加入我们的 Terraform 实验项目。Gruntwork 的使命是能够 10 倍地简化软件的开发和部署工作。虽然我们并不能每次都成功地完成这个任务（我在本书中记录了许多错误经历），但我仍然感谢你们的耐心，让我们为改善软件世界进行大胆的尝试。

### HashiCorp

感谢你们构建了许多出色的 DevOps 工具，包括 Terraform、Packer、Consul 和 Vault。你们改变了 DevOps 的世界，并改善了数百万软件开发人员的生活。

Kief Morris、Seth Vargo、Mattias Gees、Ricardo Ferreira、Akash Mahajan 和 Moritz Heiber

谢谢你们阅读本书的早期版本并提供了许多详细的意见和反馈。你们的建议使本书变得更好。

### 第 1 版的读者

读者对本书第 1 版的购买，很大程度上促进了第 2 版的出版。你们的意见、问题、改进请求，以及不断督促使第 2 版在内容上增加了近 160 个页码。希望你们受用于新内容并继续督促我的改进。谢谢你们。

### 妈妈、爸爸、Larisa、Molly

我不小心又写了一本书，这意味着我失去了很多陪伴你们的时间。无论如何，谢谢你们的支持。我爱你们。

## 读者服务

微信扫码回复：40022

- 获取作者提供的各种共享文档等免费资源
- 加入读者交流群，与更多读者互动
- 获取博文视点学院在线课程、电子书 20 元代金券



---

# 目录

第 1 章 为什么使用 Terraform .....	1
DevOps 的崛起 .....	1
什么是基础设施即代码 .....	4
基础设施即代码的好处 .....	16
Terraform 的工作原理 .....	18
Terraform 与其他 IaC 工具的比较 .....	20
小结 .....	34
第 2 章 Terraform 入门 .....	36
设置 AWS 账户 .....	37
安装 Terraform .....	40
部署单个服务器 .....	41
部署单个 Web 服务器 .....	49
部署可配置的 Web 服务器 .....	58
部署 Web 服务器集群 .....	64
部署负载均衡器 .....	68
清理工作 .....	77
小结 .....	78

<b>第 3 章 如何管理 Terraform 的状态</b>	79
什么是 Terraform 的状态	79
共享存储状态文件	82
Terraform 后端的局限性	90
隔离状态文件	92
terraform_remote_state 数据源	103
小结	114
<b>第 4 章 使用 Terraform 模块创建可重用基础设施</b>	115
模块基础知识	118
模块的输入	120
模块的局部变量	125
模块的输出	127
模块中的陷阱	130
模块版本控制	134
小结	140
<b>第 5 章 Terraform 技巧和窍门：循环、if 表达式、部署和陷阱</b>	141
循环	142
有条件的判断	161
零停机部署	176
Terraform 陷阱	186
小结	195
<b>第 6 章 生产级 Terraform 代码</b>	196
为什么构建生产级基础设施需要漫长的过程	198
生产级基础设施检查清单	200

生产级基础设施模块特点	202
小结	234
<b>第 7 章 如何测试 Terraform 代码</b>	<b>235</b>
手动测试	236
自动测试	243
小结	301
<b>第 8 章 如何在团队环境下使用 Terraform</b>	<b>303</b>
在团队中实施 IaC	303
部署应用程序代码的工作流程	310
部署基础设施代码的工作流程	320
将上述各点整合在一起	343
小结	345
<b>附录 A 推荐阅读资料</b>	<b>347</b>
<b>关于作者</b>	<b>350</b>
<b>封面介绍</b>	<b>350</b>

## 第 1 章

# 为什么使用 Terraform

仅仅能够在本地计算机上正常运行软件的代码并不算完工，即使通过了测试和代码评审也不算完成，软件只有最终交付给用户才标志着项目的结束。

软件交付（software delivery）是指通过一系列工作使代码最终对客户“可见、可用”的过程，例如，将代码运行在生产服务器之上，使代码能够应对数据流量的激增和意外停机中断，保护代码免受攻击者破坏。在深入了解 Terraform 的细节之前，让我们退后一步，从宏观上认识 Terraform 是如何融入软件交付的大背景的。

在本章中，我们将深入探讨以下主题。

- DevOps 的崛起
- 什么是基础设施即代码
- 通过代码定义基础设施的好处
- Terraform 的工作原理
- Terraform 与其他基础设施即代码工具的比较

## DevOps 的崛起

在不久之前，当创建一家软件公司时，还需要管理很多硬件。其工作包括：设置机柜和机架、安装服务器、连接电线、安装冷却装置、建立冗余电源系统等。一种很合理的组织划分是，设立一个专门负责编写软件的开发团队（“Dev”），和另一个专门负责管理硬件的运维团队（“Ops”）。

对于两个团队间的协作关系，软件行业内有一个形象的比喻：开发团队在完成一个应用程序的开发之后，将其“扔过墙头”（*toss it over the wall*）给运维团队，由运维团队决定如何部署和运行该应用程序。因为需要完成很多与物理硬件相关的工作（如架设服务器、连接网络电缆），所以运维团队的大多数部署是通过手动完成的。即使是许多与软件相关的工作（如安装应用程序和依赖库），运维人员也是通过手动执行命令的方式在服务器上完成的。

在开始的一段时间内效果还好，但是随着公司的发展壮大，最终会遇到问题。问题通常有这样的：因为软件发布通过手工完成，随着服务器数量的增加，发布的过程变得缓慢、痛苦和不可预知。运维团队有时也会犯错，最终会导致环境中的每个服务器的配置，多多少少与其他服务器有所不同（此问题被称为 *configuration drift*，即配置漂移），而非标准化配置的雪花服务器（*snowflake server*）的出现，导致更多的错误发生。有时开发人员只是耸耸肩说：“同样的代码在我的机器上运行没有问题！”，故障和停机只会变得更加频繁。

运维团队厌倦了每次发布任务后，在凌晨 3 点再次被寻呼机吵醒。因此他们将发布频率降低到每周一次，然后到每月一次，之后每 6 个月一次。在这种一年两次发布的节奏下，团队往往直到发布前的最后几周才开始尝试将所有项目代码合并到一起，这种不经常的合并自然会导致巨大的代码冲突。团队里没有人可以解决全部冲突、稳定发布分支。成员之间开始互相指责，隔阂加重。公司的业务逐渐陷入停顿。

如今，一个正在发生的巨大变化是：很多公司放弃管理自己的数据中心，转而利用诸如 AWS（Amazon Web Services）、微软 Azure、GCP（Google Cloud Platform）等云计算平台。运维团队的主要任务也逐渐从硬件管理转移到软件管理之上（使用如 Chef、Puppet、Terraform 和 Docker 等工具），系统管理员的工作也从架设服务器和插拔网线，转变为编写代码。

这样一来，开发人员和运维人员都将大部分时间花在了软件开发上，两个团队之间的界限

越来越模糊。即使在组织架构上仍然是一个独立的开发团队来负责开发应用程序，另一个运维团队来负责运行和维护应用程序。但是很显然，开发和运维人员需要更加紧密的合作，这就是 *DevOps* 运动的来历。

*DevOps* 不应该只是团队名称、职称或特定技术。相反，它更应该代表过程、思想和技术。每个人对 *DevOps* 的定义略有不同，在本书中，我将遵循以下内容。

*DevOps* 的目标是极大地提高软件交付效率。

团队应该持续地将代码集成到发布分支，并使发布分支始终保持在可部署状态，而不是噩梦一般持续多天地进行代码合并；团队应该每天数十次地部署代码，甚至在每次提交之后都应该进行代码的部署，而不仅仅是每个月才进行一次的代码部署；团队应该构建弹性的、能够自我修复的系统，使用监控和警报机制发现那些无法自动解决的问题，而不是响应频繁的故障和系统停机。

经过 *DevOps* 转型的公司往往都有令人震惊的结果。例如，Nordstrom 在应用 *DevOps* 最佳实践之后，能够将每月交付的功能数量增加 100%，缺陷减少 50%，交货时间（从构思创意到将产品运行在生产环境的总时长）缩短 60%，生产事故的数量减少 60% 至 90%。惠普的 LaserJet 固件部门采用 *DevOps* 后，其开发人员花在开发新功能上的时间从 5% 增加到 40%，总体开发成本减少了 40%。Etsy 通过使用 *DevOps*，从压力大、不频繁地部署（导致大量中断）转变为现在每天进行 25 到 50 次部署，而且停机次数也大大减少<sup>1</sup>。

*DevOps* 有四大核心价值：文化（culture）、自动化（automation）、度量（measurement）和共享（sharing），有时缩写为 CAMS（见参考资料第 1 章[1]）。在本书中我将重点关注其中一个价值：自动化。本书并非是对 *DevOps* 的全面概述，对其他方面感兴趣的读者请参阅附录 A，来获取推荐读物。

---

<sup>1</sup> Gene Kim, Jez Humble, Patrick Debois, John Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. [美]波特兰: IT Revolution Press, 2016.

DevOps 自动化的目标是将软件交付过程自动化。所以落实到管理基础设施方面，也要尽可能多地通过代码来进行，减少点击网页或手动执行 Shell 命令的方式。这种概念通常被称为：基础设施即代码（IaC）。

## 什么是基础设施即代码

基础设施即代码背后的想法是，通过编写和执行代码来定义、部署、更新和销毁基础设施。这代表着一种观念上的重要转变：将运维的各个工作都视为与软件相关，甚至包括那些明显针对硬件的工作（如设置物理服务器）。实际上，DevOps 的一个重要观点是，用户应该将所有事物都在代码中进行管理，包括服务器、数据库、网络、日志文件、应用程序配置、文档、自动测试、部署过程等。

IaC 工具分为 5 大类。

- 专项脚本
- 配置管理工具
- 服务器模板工具
- 编排工具
- 服务开通工具

让我们分别详细介绍一下。

### 专项脚本

自动化最直接的方法是为每一项任务写一个专项脚本。先将任一个手动任务进行步骤分解，使用自己喜欢的脚本语言（如 Bash、Ruby、Python）将每个步骤编写为代码，然后在服务器上运行该专项代码脚本，如图 1-1 所示。

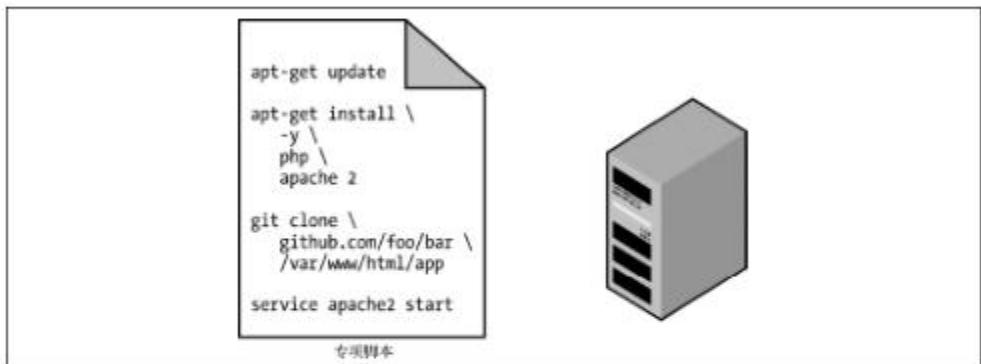


图 1-1：在服务器上运行专项脚本

例如，这是一个用来配置 Web 服务器名的 Bash 脚本 (*setup-webserver.sh*)：它首先安装依赖软件（PHP、Apache），之后从 Git 存储库中复制代码到服务器，最后启动 Apache Web 服务器。

```
# 更新 apt-get 缓存  
sudo apt-get update  
  
# 安装 PHP 和 Apache  
sudo apt-get install -y php apache2  
  
# 从存储库复制代码  
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app  
  
# 启动 Apache  
sudo service apache2 start
```

专项脚本的优点是，用户可以使用任何流行的编程语言，以擅长的方式来编写代码。而专项脚本的缺点也来自编程语言和编程方式的随意性。

尽管针对复杂任务，IaC 专项工具普遍提供了简洁的 API，但是如果使用通用编程语言，还需要为每个任务编写全套的自定义代码。此外，IaC 专项工具通常会要求特定的代码结构，而开发人员在使用通用编程语言时都会有自己的风格和操作。对于安装 Apache 服务器这种 8 行的脚本，以上两点都不是什么大问题，但是如果使用专项脚本来管理服务器、数据库、负载均衡器、网络配置等，那么代码就会变得混乱。

如果你曾经维护过一个大型的 Bash 脚本存储库，就会知道它最终会变成一堆无法维护的代码，就像一锅混乱的意大利面条。总而言之，专项脚本对那些小规模的、一次性的任务很有效，但如果计划把基础设施作为代码进行管理，那么你应该使用专业的 IaC 工具。

## 配置管理工具

Chef、Puppet、Ansible 和 SaltStack 都属于配置管理工具，它们的目的是在现有服务器上安装和管理软件。例如，这是一个名为 *web-server.yml* 的 Ansible 角色文件，它配置了与上一节的 *setup-webserver.sh* 脚本相同的 Apache Web 服务器。

```
- name: Update the apt-get cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2

- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

该代码看起来与 Bash 脚本相似，但是使用诸如 Ansible 之类的工具有以下优势。

### 编码风格

Ansible 严格要求一致的代码风格和可预测的结构，主要体现在文档和文件的布局、清晰的参数命名、密码管理等方面。每个开发人员在使用配置管理工具时，都需要遵循统一的编码规范和格式约定，这会使浏览代码变得更容易。

### 幂等性

编写需要单次运行的专项脚本并不困难，但是编写一个在反复运行后仍能正常工作的

专项脚本就没有那么简单了。每次通过脚本创建一个文件夹，都需要检查该文件夹是否已经存在；每次向文件中添加一行配置，都需要检查该行内容是否已经存在；每次要运行应用程序时，都需要检查该应用程序是否已经在运行。

代码无论运行多少次，仍然可以正常工作的特性称为幂等性。为了使上一节中的 Bash 脚本具有幂等性，需要额外添加许多行代码，包括许多 if 表达式。另一方面，大多数 Ansible 函数在默认情况下已经是幂等的。例如在 `web-server.yml` 中，Ansible 角色只有在服务器尚未安装 Apache 的情况下才进行安装，并且仅在尚未运行 Apache 的服务器上才会尝试启动相应进程。

### 分布性

专项脚本适用于单一、本地计算机的运行。Ansible 和其他配置管理工具则适用于管理大量的远程服务器，可以同时在众多服务器上同步执行代码，如图 1-2 所示。

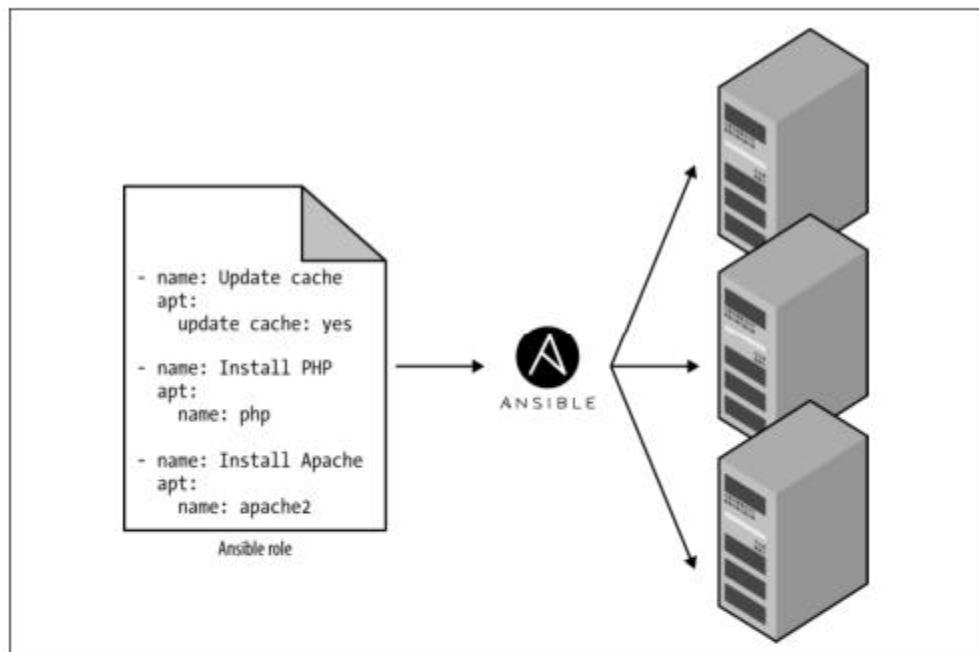


图 1-2：Ansible 这样的配置管理工具可以同时在众多服务器上同步执行代码

例如，为了将 `web-server.yml` 角色文件应用于 5 台服务器，首先你需要创建一个名为 `hosts` 的文件，其中包含这些服务器的 IP 地址，如下所示。

```
[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

接下来，定义以下 Ansible 剧本。

```
- host: webservers
  roles:
    - webserver
```

最后，通过以下方式执行 Ansible 剧本。

```
ansible-playbook playbook.yml
```

这将指示 Ansible 以并行方式同时配置 5 台服务器，或者通过在 Ansible 剧本中设置 `serial` 参数，实现滚动部署 (*rolling deployment*)，以达到分批次更新服务器的效果。例如，当将 `serial` 设置为 2 时，Ansible 将一次更新两台服务器，直到完成全部 5 台服务器的更新。如果使用专项脚本实现这种逻辑，将需要编写几十行，甚至上百行代码。

## 服务器模板工具

服务器模板工具（如 Docker、Packer 和 Vagrant）是最近非常流行的一种工具，有替代配置管理工具的趋势。其背后的设计理念是：与其创建一堆服务器，然后在服务器上运行相同的代码来配置它们，不如使用诸如 Packer 之类的服务器模板工具，将完全独立的服务器“快照”创建为映像（包括操作系统、软件、文件，以及所有其他相关的详细信息）。然后，使用其他 IaC 工具（如 Ansible）在所有服务器上统一安装该映像，如图 1-3 所示。

图 1-4 展示了映像的两种主要类型：左侧的虚拟机和右侧的容器。虚拟机虚拟化硬件，而容器仅虚拟化用户空间。

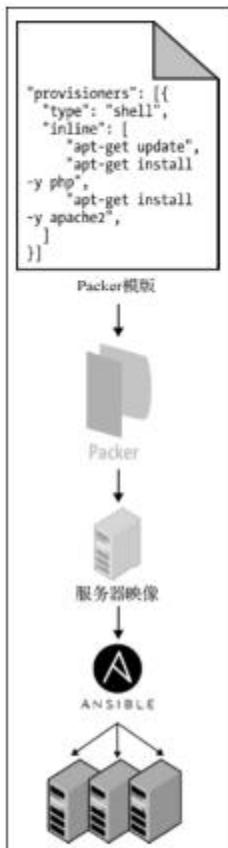


图 1-3：使用服务器模板工具创建服务器的独立映像，然后，使用其他工具在所有服务器上安装该映像

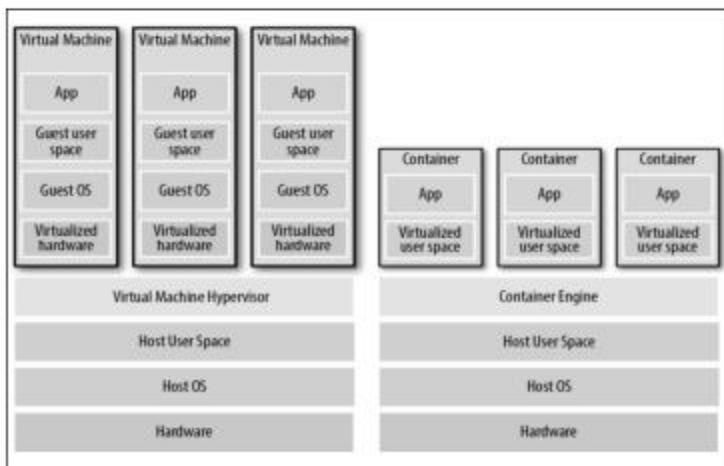


图 1-4：映像的两种主要类型：左侧的虚拟机和右侧的容器

## 虚拟机

利用虚拟机（VM）来模拟包括硬件在内的整个计算机系统。利用下层运行的管理程序（*hypervisor*），如 VMWare、VirtualBox、Parallels 来虚拟（模拟）底层的 CPU、

内存、硬盘和网络。这样做的好处是，在虚拟机管理程序上运行的任何虚拟机映像都只能看到虚拟化的硬件，因此它与主机及任何其他虚拟机映像之间是完全隔离的，并且在所有环境（例如，你的本地计算机、QA 服务器、生产服务器）中都将以完全相同的方式运行。缺点是硬件的虚拟化和每个虚拟机中运行的独立操作系统，都会导致大量的 CPU、内存和启动时间方面的开销。用户可以使用 Packer 和 Vagrant 等工具将虚拟机映像定义为代码。

## 容器

利用容器来模拟操作系统的用户空间<sup>1</sup>。即通过运行诸如 Docker、CoreOS rkt 或 cri-o 之类的容器引擎来创建隔离的进程、内存、文件系统安装点和网络。这样做的好处是，容器引擎上运行的任何容器只能看到自己的用户空间，它从主机层面上和其他容器进行分离。容器将在所有环境（例如，你的本地计算机、QA 服务器、生产服务器）中以完全相同的方式运行。缺点是，在一台服务器上运行的所有容器都共享该服务器操作系统的内核和硬件，因此在隔离度和安全性方面比虚拟机技术差很多<sup>2</sup>。但是，由于内核和硬件是共享的，因此你的容器可以在几毫秒内启动，并且几乎没有 CPU 或内存的额外开销。你可以使用 Docker 和 CoreOS rkt 之类的工具将容器映像定义为代码。

例如，下面是一个名为 *web-server.json* 的 Packer 模板，该模板将创建一个 *Amazon Machine Image (AMI)*，这是一个可以在 AWS 上运行的虚拟机映像。

```
{  
  "builders": [  
    {"ami_name": "packer-example",
```

- 
- 1 在大多数现代操作系统上，代码运行在两个“空间”——内核空间和用户空间之一上。在内核空间运行的代码可以无限地直接访问所有硬件，没有安保限制（即你可以执行任何 CPU 指令，访问硬盘驱动器的任何部分，写入内存中的任何地址）或安全限制（如内核空间崩溃通常会使整个计算机崩溃）。内核空间通常留给操作系统底层的、最受信任的功能（通常这被称为内核）。在用户空间中运行的代码无法直接访问硬件，而必须使用操作系统内核公开的 API。这些 API 可以加强安保限制（如用户权限）和安全限制（如用户空间中的应用程序崩溃通常只影响该应用程序），因此几乎所有应用程序代码都运行在用户空间中。
  - 2 一般来说，容器所提供的隔离足够来运行自有代码，但如果用户需要运行来自第三方的代码（例如，正在构建自己的云服务），可能会导致恶意代码的攻击。那么需要提高虚拟机的隔离性以保证安全。

```

    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0c55b159cbfafe1f0",
    "ssh_username": "ubuntu"
},
"provisioners": [
    "type": "shell",
    "inline": [
        "sudo apt-get update",
        "sudo apt-get install -y php apache2",
        "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
    ],
    "environment_vars": [
        "DEBIAN_FRONTEND = noninteractive"
    ]
]
}
]

```

该 Packer 模板和前面提到的 Bash 代码 `setup-webserver.sh` 具有相同的功能，配置了相同的 Apache Web 服务器<sup>1</sup>。这个 Packer 模板的例子和之前的 Bash 代码之间的唯一区别是，该 Packer 模板无法启动 Apache Web 服务器（例如，通过调用 `sudo service apache2 start` 命令来启动该服务器）。这是因为服务器模板通常用于安装映像中所需要的软件，但是软件只有在映像被启动时（例如，最终将映像部署在服务器上）才会被真正运行。

用户可以通过 `packer build web_server.json` 命令，从 Packer 模板生成一个 AMI。构建完成后，可以在所有 AWS 服务器上安装这个 AMI，并在每个服务器上配置启动时自动运行 Apache 服务（请参阅下一节的示例），它们将以完全相同的方式运行。

请注意，不同的服务器模板工具的用途略有不同。Packer 通常用于创建直接在生产服务器上运行的映像，例如，在 AWS 生产环境账户中运行的 AMI。Vagrant 通常用于创建在开发环境中运行的映像，例如，在 Mac 或 Windows 笔记本电脑上运行的 VirtualBox。Docker 通常用于创建单个应用程序的映像。Docker 可以在生产环境或开发计算机上运行，只要求该计算机已经预安装了 Docker 引擎即可。例如，一种常见的模式是使用 Packer 创建包含 Docker 引擎的 AMI 映像，再将该 AMI 部署在 AWS 账户中的服务器集群上，然后通过

<sup>1</sup> 作为 Bash 的替代者，Packer 还允许用户使用配置管理工具配置映像（如 Ansible 或 Chef）。

Docker 在该集群中部署 Docker 容器来运行应用程序。

“服务器模板化”是向不可变基础设施 (*immutable infrastructure*) 迁移的关键步骤。这个想法是受函数化编程的启发而来的，中心思想是包含不可变的变量：即在为变量设置初始值之后，就无法再次改变。如果需要更新某些内容，则需要创建一个新变量。由于变量永远不会改变，因此对代码进行逻辑推理要容易得多。

不可变基础设施背后的想法很相似：一旦部署了服务器，就永远不会再对其进行改变。如果需要更新某些内容（例如部署代码的新版本），则可以从服务器模板创建新映像，然后将其部署到新服务器上。由于服务器本身不会发生变化，因此可以轻松推断出已部署的内容。

## 编排工具

服务器模板工具非常适合创建虚拟机和容器，但是该如何管理它们呢？在大多数实际案例中需要完成以下管理工作。

- 部署虚拟机及容器，有效利用硬件
- 使用诸如滚动部署、蓝绿部署和灰度部署之类的策略，对现有虚拟机及容器群进行更新
- 监视虚拟机及容器的健康状况，并自动替换不健康的虚拟机和容器（自动修复）
- 根据负载增加或减少虚拟机及容器的数量（自动扩展）
- 在虚拟机、容器之间分配流量（负载均衡）
- 允许虚拟机和容器通过网络相互查找并通信（服务发现）

用于处理这些任务的都可以算作编排工具 (*orchestration tools*)，包括 Kubernetes、Marathon/Mesos、Amazon Elastic Container Service (Amazon ECS)、Docker Swarm 和 Nomad。例如，Kubernetes 允许用户定义如何以代码形式管理 Docker 容器。首先，用户需要部署一个 Kubernetes 集群，该集群包括一组服务器，通过 Kubernetes 管理并运行 Docker 容器。大多数主要的云服务提供商，例如，用于 Kubernetes 的 Amazon Elastic Container Service (Amazon EKS)、Google Kubernetes Engine (GKE) 和 Azure Kubernetes Service (AKS) 等，都对部署托管的 Kubernetes 集群具有原生支持。

拥有可用的集群后，可以在 YAML 文件中定义如何以代码形式运行 Docker 容器。

```
apiVersion: apps / v1
```

```
# 使用 Deployment 部署 Docker 容器的多个副本并以声明的方式发布更新
kind: Deployment

# 有关此部署的元数据，包括其名称
metadata:
  name: example-app

# 配置此部署
spec:
  # 指示部署脚本如何找到你的容器
  selector:
    matchLabels:
      app: example-app

  # 指示部署同时运行 3 个 Docker 容器副本
  replicas: 3

  # 指示如何更新部署，这里配置为滚动更新
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate

  # 这是要部署容器的模板
  template:
    # 这些容器的元数据，包括标签
    metadata:
      label:
        app: example-app

    # 容器的规范
    spec:
      containers:
        # 在 80 端口运行 Apache
        - name: example-app
          image: httpd:2.4.39
          port:
            - containerPort: 80
```

以上文件通过声明方式指示 Kubernetes 创建一个 *Deployment*。

- 一个或多个 Docker 容器一起运行。这个容器组被称为 Pod。前面代码中定义的 Pod 包含一个运行 Apache 的 Docker 容器。

- Pod 中每个 Docker 容器的设置。前面代码中的 Pod 将 Apache 配置为在 80 端口监听。
- 在集群中运行 Pod 的副本数（即 *replica*）。前面的代码配置了 3 个副本。Kubernetes 使用调度算法依据高可用性原则选择最佳服务器，自动决定集群中每个 Pod 的最佳部署位置（例如，尝试在不同的服务器上运行相同 Pod 的副本，这样，一台服务器崩溃不会导致整个应用程序的崩溃）、资源（例如，选择具有容器所需端口、CPU、内存和其他资源可用的服务器）、性能（例如，尝试选择负载最小、已运行容器数量最小的服务器），等等。Kubernetes 还可以持续监视集群以确保始终有 3 个副本同时在运行，并自动替换崩溃或停止响应的 Pod。
- 如何部署更新。在部署新版本的 Docker 容器时，以上代码将首先创建 3 个新副本，等待新副本正常运行后，再取消之前部署的 3 个旧副本。

通过以上示例可以看到，只需要几行 YAML 脚本，便能够具有如此强大的功能！你可以通过运行 `kubectl apply -f example-app.yml` 命令，指示 Kubernetes 部署你的应用程序。也可以通过更改 YAML 文件并运行 `kubectl apply` 命令，再次进行更新。

## 服务开通工具

通过配置管理、服务器模板和编排工具可以定义服务器上运行的代码，但还是需要 Terraform、CloudFormation 和 OpenStack Heat 等工具，负责创建服务器本身。实际上，用户不仅可以使用服务开通工具来创建服务器，还可以创建数据库、缓存、负载均衡器、队列、监控程序、子网配置、防火墙设置、路由规则、安全套接字层（SSL）证书，以及涉及基础设施的所有其他部分，如图 1-5 所示。

例如，下面的 Terraform 代码部署了一台 Web 服务器。

```
resource "aws_instance" "app" {
  instance_type      = "t2.micro"
  availability_zone = "us-east-2a"
  ami                = "ami-0c55b159cbfafef1f0"

  user_data    = <<-EOF
                  #! / bin / bash
                  sudo service apache2 start
                  EOF
}
```

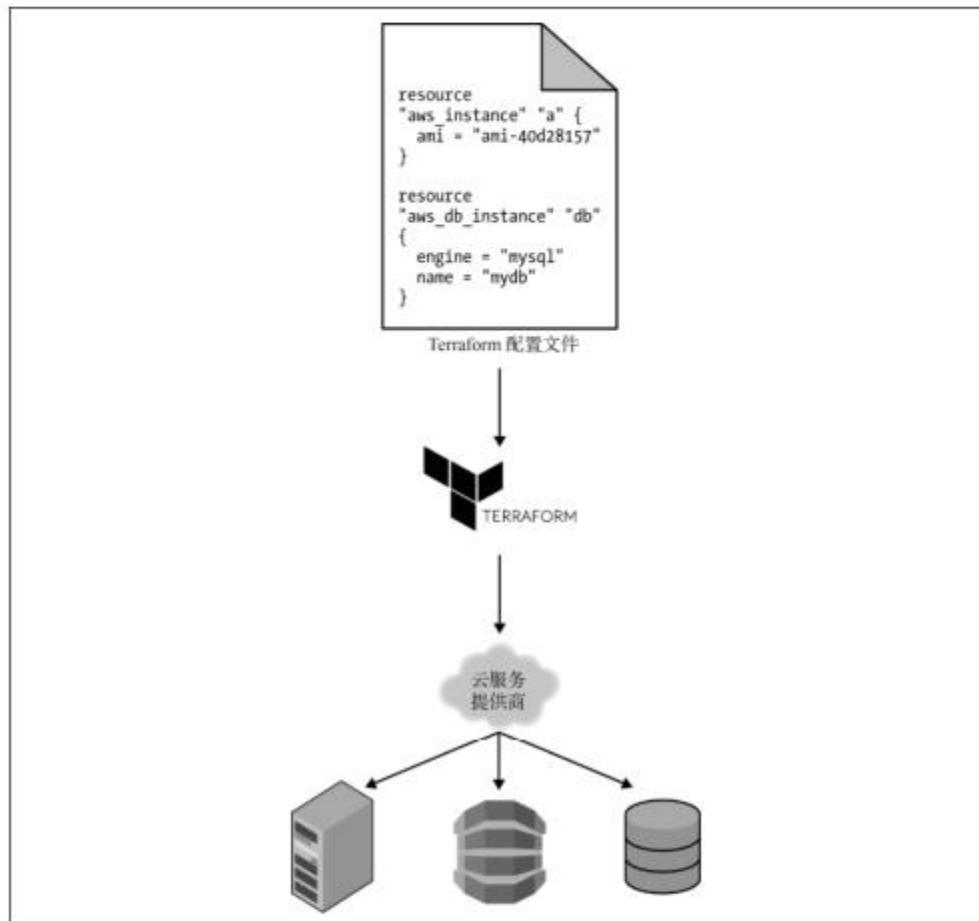


图 1-5：使用服务开通工具，通过云服务提供商提供的接口，来创建服务器、数据库、负载均衡器，以及基础设施的所有其他部分

如果你还不熟悉某些语法，请不要担心。我们先来关注两个参数。

#### ami

这个参数指定了将在服务器上部署的 AMI 的 ID。AMI ID 来自之前章节通过 Packer 模板 *web-server.json* 所创建的映像。其中包括 PHP、Apache 和应用程序源代码。

### `user_data`

这个参数指定了在 Web 服务器启动时，需要执行的 Bash 脚本。上面代码示例中的脚本，用来启动 Apache 服务。

此代码展示了不可变基础设施中常见的一种搭配模式：将服务开通工具和服务器模板工具配合使用。

## 基础设施即代码的好处

现在我们已经学习了不同风格的 IaC 工具。一个非常好的问题是：为什么要学习这么多新的语言和工具，并管理和维护额外的代码呢？

答案是：代码的功能是极其强大的。通过早期投入将手动工作转化为代码，你的软件交付能力将得到显著改善。根据 2016 年 DevOps 状况报告（见参考资料第 1 章[2]），使用 DevOps 实践（例如 IaC）的组织，部署频率提高了 200 倍，从故障中恢复的速度提高了 24 倍，交付周期缩短为原来的 1/2555。

将基础设施定义为代码后，就可以通过软件工程最佳实践来改善软件交付过程了，其中包括如下内容。

### 自助服务

大多数手动部署代码的团队，只有极少数的系统管理员（通常只有一个人）知道部署过程的全部细节，并且是唯一有权限访问生产环境的人。随着公司的发展，这将成为一个主要的瓶颈。如果你的基础设施是通过代码定义的，整个部署过程可以自动化，开发人员可以在必要时自己启动部署过程。

### 速度和安全性

自动化将大大加速部署过程，因为计算机可以比人更快、更安全地执行部署步骤，所以自动化的过程将更加一致，更具可重复性，并且不易出现人为错误。

### 文档

避免将基础设施的细节遗忘在系统管理员的记忆中。将基础设施定义为代码之后，就

可以将其状态保存并显示在任何人都可以读取的源文件中。换句话说，IaC 同时具备文档功能，即使系统管理员休假，也允许组织中的其他人了解部署的运行方式。

#### 版本控制

将 IaC 源文件存储在版本控制系统中，基础设施的历史变更记录将在提交日志中被完整保留。这将成为强大的调试工具，任何时候一旦出现故障，第一步就是检查提交日志并找出基础设施发生了什么变化，第二步是通过简单地还原 IaC 代码到之前已知的正常版本来解决问题。

#### 验证

如果在代码中定义基础设施的状态，则对于每一个更改都可以执行代码评审、运行自动测试。通过静态分析工具扫描代码，可以显著降低缺陷代码出现的概率。

#### 重用

基础设施可以打包成可重用的模块，这样针对每个新产品、新环境，可以避免从头编写部署代码，通过利用已知的、经过实战测试的模块<sup>1</sup>，达到快速开发的目的。

#### 幸福感

使用 IaC 的另一个经常被忽略却非常重要的原因是幸福感。部署代码和管理基础设施经常是重复且乏味的任务。开发人员和系统管理员都厌恶这种没有创意、没有挑战、不被认可的工作。连续几个月的完美的代码部署，却没有引起任何关注。直到有一天部署出了问题，大家才会意识到你的存在。这种情形造成了极大的压力和令人不悦的氛围。IaC 提供了一种更好的选择，让计算机处理擅长的事情（自动化），让开发人员从事擅长的任务（代码开发）。

现在你已经了解了 IaC 的重要性，那么 Terraform 是否是最适合你的 IaC 工具呢？为了回答这个问题，我首先要对 Terraform 的工作原理做一个快速介绍，然后再与其他流行的 IaC 工具（例如 Chef、Puppet 和 Ansible）进行比较。

---

<sup>1</sup> 请参考 Gruntwork Infrastructure as Code Library（见参考资料第 1 章[3]）。

## Terraform 的工作原理

Terraform 使用 Go 语言编写，是由 HashiCorp 公司创建的开源工具。Go 语言代码被编译成一个叫 Terraform 的二进制文件。更准确地说，对于每个支持 Terraform 的操作系统，都存在一个特定的二进制文件。

用户可以从笔记本电脑、构建服务器或任何其他计算机上运行这个二进制文件来部署基础设施。实现这一目标不需要运行任何其他基础设施。因为后台的 `terraform` 可执行文件将代表用户，对一个或多个提供商接口（如 AWS、Azure、Google Cloud、DigitalOcean、OpenStack 等）进行 API 调用。这意味着 Terraform 正在利用提供商已经为 API 服务器部署的基础设施，以及已有的身份验证机制（例如，你已经拥有的 AWS API 密钥）。

Terraform 如何知道要调用哪些 API 呢？答案是：依据用户创建的 *Terraform* 配置文件，这些配置定义了要创建的基础设施。这些配置也就是“基础设施即代码”中的“代码”。下面是一个 Terraform 配置文件的示例。

```
resource "aws_instance" "example" {
    ami          = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
    name        = "demo.google-example.com"
    managed_zone = "example-zone"
    type        = "A"
    ttl         = 300
    rrdatas     = [aws_instance.example.public_ip]
}
```

即使这是你第一次阅读 Terraform 代码，也应该不会有太大困难。此段 Terraform 代码首先调用 AWS 的 API 来部署一台服务器。然后调用 Google Cloud 的 API，创建指向 AWS 服务器 IP 地址的 DNS 条目。Terraform 的语法（将在第 2 章学到）非常简单，可以在多云环境下进行互相关联的部署。

用户可以在 Terraform 配置文件中定义整套基础设施：服务器、数据库、负载均衡器、网络拓扑等，然后将配置文件提交到版本控制系统。接下来，通过运行 Terraform 命令，例

如 `terraform apply` 命令，来部署该基础设施。`terraform` 命令将对代码进行解析，将代码转化为云服务提供商的一系列 API 调用，并在此过程中优化 API 调用。图 1-6 展示了 Terraform（一个二进制文件）工具将用户的配置文件中的内容转换为对云服务提供商的 API 调用。

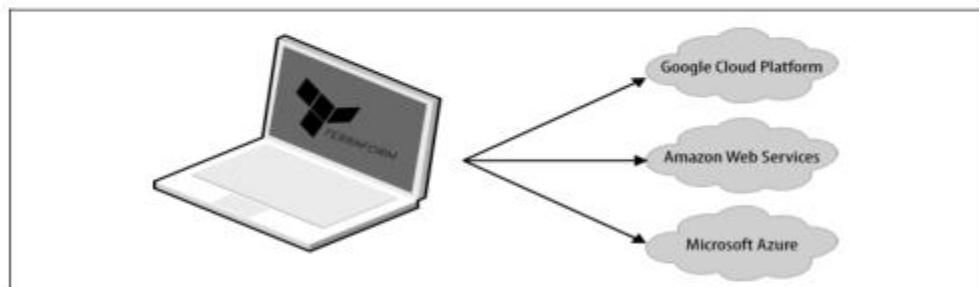


图 1-6：Terraform 工具将用户的配置文件中的内容转换为对云服务提供商的 API 调用

当团队成员需要对基础设施进行更改时，他们不需要在服务器上手动更新基础设施。正确做法是，首先更改 Terraform 配置文件，然后通过自动测试和代码评审来验证变更的正确性，之后将代码更新提交到版本控制系统，最后运行 `terraform apply` 命令，由 Terraform 进行必要的 API 调用来正式部署变更。



#### 云服务提供商之间的透明可移植性

由于 Terraform 支持许多不同的云服务提供商，一个经常被问到的问题是，它是否支持云服务提供商之间的透明移植。例如，如果用户使用 Terraform 在 AWS 中定义了一堆服务器、数据库、负载均衡器和其他基础设施，是否可以指示 Terraform 在其他的云服务提供商（如 Azure 或 Google Cloud）中部署完全相同的基础设施？其实这原本就是个伪命题。实际情况是，不可能在不同的云平台中部署“完全相同的基础设施”，因为云服务提供商本身无法提供相同类型的基础设施！AWS 提供的服务器、负载均衡器和数据库在功能、配置、管理、安全性、可伸缩性、可用性、可观察性等方面与 Azure 和 Google Cloud 所能提供的完全不同。没有简单的方法可以“透明”地解决这些差异，特别是当一个云服务提供商中的功能在其他云服务提供商中根本不存在的情况下。

Terraform 的方法是允许用户编写针对每个云服务提供商的特定代码，从而利用该提供商的独特功能和特性，但对所有提供商使用统一的语言、工具集和 IaC 实践。

## Terraform 与其他 IaC 工具的比较

基础设施即代码（IaC）的概念很出色，但是选择 IaC 工具的过程却困难重重。各种 IaC 工具的功能相互重叠，有的开源，有的提供商业支持。除非用户自己使用过每一个工具，否则很难有一个统一的选择标准。

更难的是，市面上有关工具之间比较的文章，大多数仅仅列出每个工具的大量属性，并让你感觉选择其中任何一个工具都可以取得成功。尽管从技术上讲这是正确的，但在实践中，这对初学者没有任何帮助。这就有点像告诉编程新手，你可以使用 PHP、C、汇编语言或任何语言成功地构建网站。这样的声明，技术上是正确的，但是却没有包含必要的信息供初学者做出正确的决定。

下面我将对最受欢迎的配置管理和服务开通工具——Terraform、Chef、Puppet、Ansible、SaltStack、CloudFormation 和 OpenStack Heat 进行详细的比较。我的目标是：通过解释为什么我的公司 Gruntwork（见参考资料第 1 章[4]）选择 Terraform 作为 IaC 工具，以及从某种意义上说，我为什么编写本书来帮助用户判断 Terraform 这是否对你也是个正确的选择<sup>1</sup>。与所有技术决策一样，这是权衡取舍和优先级排序的问题，即使你的优先级可能与我的有所不同，但我希望通过分享这一思考过程，帮助你做出自己的决定。

以下是要主要的考虑和对比方向。

- 配置管理（configuration management）与服务开通（provisioning）对比
- 可变（mutable）基础设施与不可变（immutable）基础设施对比
- 过程性（procedural）语言与声明性（declarative）语言对比
- 主控模式与无主控模式对比
- 代理方式与无代理方式对比

---

<sup>1</sup> Docker、Packer 和 Kubernetes 不在比较范围之内，因为它们可以与任何配置工具和服务开通工具一起使用。

- 大型社区与小型社区对比
- 成熟技术与前沿技术对比
- 同时使用多个工具

## 配置管理与服务开通对比

正如前面所看到的，Chef、Puppet、Ansible 和 SaltStack 都属于配置管理工具，而 CloudFormation、Terraform 和 OpenStack Heat 都是服务开通工具。实际上区分并不明显，配置管理工具通常可以进行某种程度的服务开通（例如，使用 Ansible 部署服务器），服务开通工具通常也可以进行某种程度的配置管理（例如，使用 Terraform 配置服务器和运行配置脚本）。所以需要根据具体用例，选择最适合的工具。<sup>1</sup>

尤其是，诸如 Docker 或 Packer 之类的服务器模板工具，已经实现了绝大多数配置管理工具的功能。例如从 Dockerfile 或 Packer 模板创建映像后，剩下要做的就是为运行这些映像开通基础设施服务。所以在服务开通阶段，服务开通工具将是最佳选择。

另一方面，如果不使用服务器模板工具，将配置管理工具和服务开通工具一起使用，也是个不错的选择。例如，你可以使用 Terraform 来部署服务器，然后运行 Chef 来配置每个服务器。

## 可变基础设施与不可变基础设施对比

配置管理工具（如 Chef、Puppet、Ansible 和 SaltStack）属于典型的可变基础设施的范例。例如，如果指示 Chef 安装一个新版本的 OpenSSL，它将在现有的服务器上运行软件更新，更改将覆盖现有的内容。随着时间的推进，需要部署越来越多的更新，每台服务器都存在各不相同的更改记录。结果每台服务器的配置都与其他服务器有着细微差别，从而导致错误难以被诊断（这与手动管理服务器时发生的配置漂移问题是类似的，尽管使用了配置管理工具后问题会少很多）。即使使用自动测试工具，也很难发现这种错误。通常，配置更改在测试服务器上工作得很好，但是同样的更改在生产服务器上的行为就会有所不同，这是因为生产服务器累积了长期的更改，这些更改不能 100% 反映在测试环境中。

---

<sup>1</sup> 如今，配置管理和服务开通之间的区别越发不明显了，因为一些主要的配置管理工具已经逐渐改善了它们对服务开通的支持，例如 Chef Provisioning 和 Puppet AWS Module（见参考资料第 1 章[5]）。

如果使用诸如 Terraform 之类的服务开通工具来部署由 Docker 或 Packer 创建的机器映像，大多数“更改”实际上是部署一个全新的服务器。例如，要部署新版本的 OpenSSL，用户可以通过使用 Packer 把新版本的 OpenSSL 打包到新创建的映像里，再将该映像部署到一组新服务器上，然后终止旧服务器。因为每个部署都在新服务器上使用不可变的映像，所以这种方法降低了发生配置漂移错误的可能性，使你更准确地知道每个服务器上正在运行的软件，并允许用户轻松地将部署恢复到之前的任何版本（通过使用以前的映像）。这也使自动测试更加有效，因为同样在测试环境中通过测试的不可变映像，将直接被使用在生产环境中。

当然，在服务开通工具擅长的领域，也可以强制使用配置管理工具进行不可变的部署，但这并不是配置管理工具的常规使用方法。值得一提的是，不可变基础设施也有不足之处。例如，通过服务器模板重建映像并重新部署所有服务器的方式，即使微不足道的更改也会花费很长时间。此外，服务器启动并运行后，基础设施的不变性就会消失，随着进程开始在硬盘上写入数据，服务器的配置也会发生某种程度的漂移（尽管通过提高部署频率可以缓解这种情况）。

### 过程性语言与声明性语言对比

Chef 和 Ansible 鼓励采用过程性编程语言，在这种语言中，用户可以编写代码来指示工具如何逐步实现所要达到的最终状态。Terraform、CloudFormation、SaltStack、Puppet 和 Open Stack Heat 都鼓励使用声明性编程语言，在这种语言中，用户可以编写代码来指示工具所要达到的最终状态，而 IaC 工具将负责决定具体的实现步骤。

为了展示差异，让我们来看一个例子。假设你要部署 10 台服务器（AWS 术语中的 EC2 实例）来运行 ID 为 `ami-0c55b159cbfafef0`（Ubuntu 18.04）的 AMI。下面是使用过程性语言的 Ansible 模板的一个简单例子。

```
- ec2:  
  count: 10  
  image: AMI-0c55b159cbfafef0  
  instance_type: t2.micro
```

接下来是一个通过 Terraform 的声明性语言，实现同样结果的一个简单例子。

```
resource "aws_instance" "example" {
    count      = 10
    ami        = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}
```

从表面上看，这两种方法十分相似，当你最初执行 Ansible 或 Terraform 时，它们会产生相似的结果。但是当你想做出改变时，有意思的事情发生了。

例如你需要将服务器数量增加到 15 个，以应对流量的增加。对于 Ansible，之前编写的进程性代码将不再有用。如果仅将服务器数量直接改为 15 并重新运行代码，它将部署 15 台新服务器，总共为 25 台！因此，你需要知道已经部署了什么，并编写了一个全新的程序脚本，来添加 5 台新服务器。

```
- ec2:
  count: 5
  ami: AMI-0c55b159cbfafe1f0
  instance_type: t2.micro
```

如果利用声明性代码，你要做的就是声明所需的最终状态，然后由 Terraform 自己决定如何达到该最终状态，Terraform 也会记录它过去创建的任何状态。因此，对于部署 5 台服务器，你要做的就是使用相同的 Terraform 配置，并将数量从 10 更新到 15。

```
resource "aws_instance" "example" {
    count      = 15
    ami        = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}
```

如果应用这个配置，Terraform 将意识到它已经创建了 10 台服务器，因此要做的就是再创建 5 台新的服务器。实际上，在应用此配置之前，你可以使用 Terraform 的 `plan` 命令来预览将要发生的变更。

```
$ terraform plan

# aws_instance.example[11] 将被创建
+ resource "aws_instance" "example" [
  + ami      = "ami-0c55b159cbfafe1f0"
  + instance_type = "t2.micro"
  + (...)
```

```
}

# aws_instance.example[12] 将被创建
+ resource "aws_instance" "example" {
  + ami          = "ami-0c55b159cbfafe1f0"
  + instance_type = "t2.micro"
  + ...
}

# aws_instance.example[13] 将被创建
+ resource "aws_instance" "example" {
  + ami          = "ami-0c55b159cbfafe1f0"
  + instance_type = "t2.micro"
  + ...
}

# aws_instance.example[14] 将被创建
+ resource "aws_instance" "example" {
  + ami          = "ami-0c55b159cbfafe1f0"
  + instance_type = "t2.micro"
  + ...
}
```

```
Plan: 5 to add, 0 to change, 0 to destroy.
```

现在，当你要部署不同版本的应用程序（例如 AMI ID `ami-02bcbb802e03574ba`）时，会发生什么呢？对于使用过程性语言的 Ansible，之前的模板都没有用了，你需要编写另一个模板来追踪以前部署的 10 台（或者 15 台）服务器，小心翼翼地更新每一台到新版本。对于使用声明性语言的 Terraform，可以重用相同的配置文件，只需将 `ami` 参数更改为 `ami-02bcbb802e03574ba` 即可。

```
resource "aws_instance" "example" {
  count      = 15
  ami        = "ami-02bcbb802e03574ba"
  instance_type = "t2.micro"
}
```

显然，这些示例已被简化。Ansible 确实允许用户在部署新的 EC2 实例之前使用标签来搜索现有的 EC2 实例（例如使用 `instance_tags` 和 `count_tag` 参数）。在 Ansible 中，根据资源的标签、映像版本、可用区，手动找出每个实例的历史信息，是一项十分繁复的工作。

这凸显了过程性 IaC 工具的两个主要问题。

### 程序代码无法完全捕获基础设施的状态

仅仅通过阅读前面 3 个 Ansible 模板，并不足以了解所部署的内容，你还需要知道这些模板的应用顺序。如果通过不同的顺序执行它们，则最终会拥有不同的基础设施，执行顺序无法被代码记录。换句话说，要推断出 Ansible 或 Chef 代码的运行结果，你需要了解发生过的每项变更的完整历史记录。

### 过程性程序代码限制可重用性

过程性的程序代码的重用性是十分有限的，因为你必须考虑基础设施的当前状态。由于状态不断变化，一周前使用的代码可能现在已经失效，因为设计它时针对的基础设施数状态已经发生改变。过程性代码的存储库会随着时间的推移而变得庞大而复杂。

使用 Terraform 的声明性语言，代码始终代表基础设施的最新状态。通过阅读代码，你可以确定当前环境的部署内容及配置细节，不必担心历史记录或时间顺序。因为不需要手动处理当前状态，这也使创建可重用的代码变得容易。相反，你只需要专注于描述所需要的状态，Terraform 会自动计算出如何从当前状态演进到目标状态。Terraform 代码库往往体积较小且易于理解。

当然，声明性语言也有缺点。如果无法使用完整的编程语言，用户的语法表达能力将十分有限。例如，某些类型的基础设施更改（如零停机部署），很难用纯粹的声明性语法来表达（但并非不可能，正如你将在第 5 章中所看到的）。“逻辑”处理的能力也十分有限（如 if 条件表达式、循环），创建通用的、可重用的代码会变得很困难。幸运的是，Terraform 提供了许多功能强大的原语，如输入变量、输出变量、模块、`create_before_destroy`、`count`、三元语法，以及内置的功能，使用户使用声明性语言也可以创建干净、可配置、模块化的代码。我会在第 4 章和第 5 章介绍这些主题。

### 主控模式与无主控模式对比

默认情况下，Chef、Puppet 和 SaltStack 都要求用户运行主控服务器来存储基础设施状态和发布更新内容。每次更新基础设施中的内容时，客户端（如命令行工具）首先向主控服务器发出命令，之后主控服务器将更新推送到所有其他服务器，或者这些服务器会定期从

主控服务器提取最新更新。

主控服务器模式的优点是：首先，你可以在一个集中的地方查看和管理基础设施的状态。许多配置管理工具甚至为主控服务器提供了 Web 界面（如 Chef Console、Puppet Enterprise Console），以便于更直接地查看正在发生的情况。其次，一些主控服务器可以在后台持续运行，并对服务器配置进行强制约束。如果被管理的服务器上发生了人为的手动更改，那么主控服务器可以恢复该更改，以避免配置漂移。

但是，必须承认运行主控服务器有一些严重的缺点。

#### 额外的基础设施

仅仅为了运行主控服务器，用户需要部署额外的服务器，为了实现高可用性和可伸缩性，甚至需要部署额外的服务器集群。

#### 需要维护

用户需要负责维护、升级、备份、监视和扩展主控服务器集群。

#### 影响安全性

需要为客户端提供与主控服务器进行通信的方法，以及主控服务器与所有其他服务器进行通信的方法。这些通信渠道，意味着需要打开额外的端口，并配置额外的身份验证系统，所有这些都会暴露更多的弱点给潜在的攻击者。

Chef、Puppet 和 SaltStack 都对无主控服务器模式有着不同程度的支持。在这种模式中，你可以只在每个目标服务器上运行代理端软件，通常以周期性间隔运行（例如，每 5min 运行一次 cron job 作业）。代理软件会直接从版本控制系统（而不是主控服务器）中提取最新更新。这种做法降低了不确定性，但是，正如我将在下一节中所讨论的，这仍然留下了许多未解决的问题，尤其是有关如何首次配置服务器和安装代理软件的问题。

默认情况下，Ansible、CloudFormation、Heat 和 Terraform 均为无主控服务器软件。更准确地说，其中一些可能依赖于主控服务器，但是这些服务器已经是正在使用的基础设施的一部分，而不需要你去管理额外的组件。例如，Terraform 使用云服务提供商的 API 与云平台进行通信，从某种意义上讲，API 服务器就扮演着主控服务器的角色，只是它们不需要任何额外的基础设施或额外的身份验证机制（只需要使用已有的 API 密钥）。Ansible

通过 SSH 直接连接到每个服务器来工作，因此，不需要运行任何额外的基础设施或管理额外的身份验证机制（只需要使用已有 SSH 密钥即可）。

## 代理方式与无代理方式对比

Chef、Puppet 和 SaltStack 都要求在被管理的每台服务器上安装代理软件（如 Chef Client、Puppet Agent、Salt Minion）。这些客户端代理软件运行在服务器的后台，负责安装配置管理系统中的更新。

代理方式的缺点如下。

### 关于自举

如何首次进行服务开通并在服务器上安装代理软件？一些配置管理工具只是在回避这个问题，而假设某些外部过程会解决这些问题（例如，首先在 AMI 中安装代理软件，之后通过使用 Terraform 在服务器上部署相关映像）。有些配置管理工具也有特殊自举过程，通过使用云服务提供商的 API 运行一次性服务开通命令，通过 SSH 将代理软件安装在这些服务器上。

### 需要维护

你需要定期细心地更新代理软件，并使其与主控服务器（如果有的话）保持同步。你还需要监控这些代理软件，在它们崩溃时重新启动。

### 影响安全性

如果代理软件从主控服务器（如果不使用主控服务器，则从其他服务器）下拉配置，则需要在每台服务器上打开出站端口。如果主控服务器将配置推送给代理，则需要在每台被管理的服务器上打开入站端口。在这两种情况下，你都必须清楚如何处理服务器与代理之间的身份验证。所有这些都增加了被攻击的风险。

其实 Chef、Puppet 和 SaltStack 也部分支持无代理模式（如 salt-ssh），但是在无代理模式下，配置管理工具通常无法使用全部功能。这就是为什么在一般情况下，Chef、Puppet 和 SaltStack 总是默认为代理和主控服务器模式的原因。

Chef、Puppet 和 SaltStack 的典型架构都有太多的可变因素。例如，Chef 的默认设置（如

图 1-7 所示) 是: 运行在本机的 Chef 客户端与 Chef 主控服务器通信, 主控服务器再将变更部署到所有安装了 Chef 的客户端的服务器上。

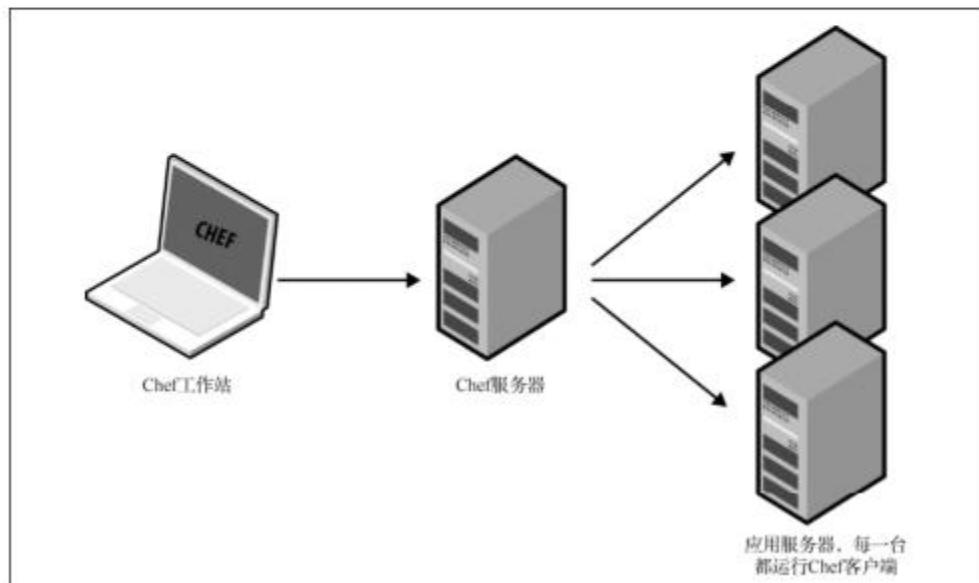


图 1-7: Chef 的默认设置

所有这些额外的系统都会在基础设施中引入额外的风险。每次凌晨 3 点收到错误报告时, 你都需要首先弄清楚出错的是应用程序代码本身、IaC 代码、配置管理客户端、主控服务器、客户端与主控服务器通信的方式, 还是被管理服务器与主控服务器通信的方式, 或者……

Ansible、CloudFormation、Heat 和 Terraform 不需要安装任何代理软件。更确切地说, 其中一些需要代理, 但是这些代理通常已经成了基础设施的一部分。例如, AWS、Azure、Google Cloud 和所有其他云服务提供商都负责在每个物理服务器上安装、管理和认证代理软件。作为 Terraform 的用户, 无须担心任何相关事情, 只需发出命令, 云服务提供商的代理软件将在所有服务器上为你执行命令。当使用 Ansible 时, 目标服务器上需要运行 SSH 进程, 该进程通常是在大多数服务器上默认的运行组件。

Terraform 使用无主控服务器模式和无代理软件的架构（如图 1-8 所示）。只需要运行 Terraform 客户端，就可以通过云服务提供商（例如 AWS）的 API 来完成其余的工作了。

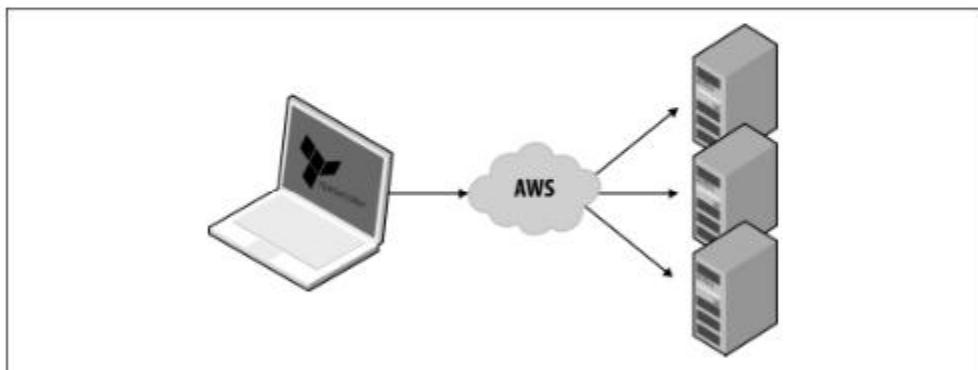


图 1-8：Terraform 使用无主控服务器模式和无代理软件的架构

### 大型社区与小型社区对比

当你选择一项技术时，你也在选择一个社区。在许多情况下，项目所处生态系统的优劣会比技术本身的质量对你的体验产生更大的影响。一个社区将决定：有多少人正在为该项目做出贡献，有多少可使用插件、集成和扩展，有多少在线帮助（例如博客文章、StackOverflow 上的回答）可供参考，以及有多少相关背景的从业人员（如员工、顾问、外部支持公司）可以参与项目。

准确地比较社区是件困难的事情，但是可以通过在线搜索发现一些趋势。表 1-1 为 IaC 社区的比较，它采用我在 2019 年 5 月期间收集的数据，比较了流行的 IaC 工具，比较项目包括开源和闭源、所支持的云服务提供商、GitHub 上的贡献者和得星评级总数、在一个月内（4 月中旬到 5 月中旬）活跃的提交总数、一个月内错误报告总数、有多少开源库可供该工具使用、在 StackOverflow 上针对该工具列出的提问数，以及该工具在 Indeed.com 上的职位数。

表 1-1: IaC 社区的比较

	开源 类型	云平 台	贡献者	星级	提交 (30 天内)	错误报 告 (30 天内)	库	StackOverflow 问题	职位
Chef	Open	All	562	5,794	435	86	3,832 <sup>a</sup>	5,982	4,378 <sup>b</sup>
Puppet	Open	All	515	5,299	94	314 <sup>c</sup>	6,110 <sup>d</sup>	3,585	4,200 <sup>e</sup>
Ansible	Open	All	4,386	37,161	506	523	20,677 <sup>f</sup>	11,746	8,787
SaltStack	Open	All	2,237	9,901	608	441	318 <sup>g</sup>	1,062	1,622
CloudFormation	Closed	AWS	?	?	?	?	377 <sup>h</sup>	3,315	2,318
Heat	Open	All	361	349	12	600 <sup>i</sup>	0 <sup>j</sup>	88	2,201 <sup>k</sup>
Terraform	Open	All	1,261	16,837	173	204	1,4621	2,730	3,641

<sup>a</sup> Chef 市场中 Chef cookbooks 的数目。(见参考资料第 1 章[6])<sup>b</sup> 为了避免对“Chef”一词的误报，这里以关键字“Chef Devops”进行搜索<sup>c</sup> 基于 Puppet Labs JIRA 账户。(见参考资料第 1 章[7])<sup>d</sup> Puppet Forge 中的模块数。(见参考资料第 1 章[8])<sup>e</sup> 为了避免对“Puppet”一词的误报，这里搜索的是“Puppet Devops”。<sup>f</sup> 这是 Ansible Galaxy 中可重复使用的角色的数量。(见参考资料第 1 章[9])<sup>g</sup> 这是 Salt Stack Formulas GitHub 账户中的公式数。(见参考资料第 1 章[10])<sup>h</sup> 这是 awslabs GitHub 账户中模板的数量。(见参考资料第 1 章[11])<sup>i</sup> 基于 OpenStack 的 bug 跟踪系统。(见参考资料第 1 章[12])<sup>j</sup> 无法找到任何 Heat 模板的社区信息。<sup>k</sup> 为了避免“Heat”发生歧义，这里搜索的是“Openstack”。<sup>l</sup> 这是 Terraform 注册中心模块的数量。(见参考资料第 1 章[13])

显然，这并不是一个完美的比较。例如，某些工具拥有多个存储库；而某些工具则使用其他方法来进行错误和问题跟踪；通过“Chef”或“Puppet”等关键词搜索职位数也是有歧义的；Terraform 在 2017 年将提供商代码分成了单独的存储库，因此仅测量核心存储库上的活动就大大低估了 Terraform 的总体活动（至少低估 10 倍）。

但是一些显而易见的趋势是：首先，在这个比较中，大多数是支持多个云服务提供商的开源 IaC 工具，只有 AWS 专用的 CloudFormation 工具是封闭源代码的。其次，Ansible 在人气方面领先于其他工具，Salt 和 Terraform 紧随其后。

另一个值得注意的趋势是，2016 年至 2019 年间这些数字发生的变化。表 1-2 显示了 IaC 社区自 2016 年 9 月至 2019 年 5 月之间的变化，即我所收集的数值每一项的增长率。

表 1-2：IaC 社区自 2016 年 9 月至 2019 年 5 月之间的变化

	开源	云平 台	贡献者	星级	提交(30 天内)	问题报告 (30 天内)	库	StackOverflow 帖子	职位
Chef	Open	All	+18%	+31%	+139%	+48%	+26%	+43%	-22%
Puppet	Open	All	+19%	+27%	+19%	+42%	+38%	+36%	-19%
Ansible	Open	All	+195%	+97%	+49%	+66%	+157%	+223%	+125%
SaltStack	Open	All	+40%	+44%	+79%	+27%	+33%	+73%	+257%
CloudFormation	Closed	AWS	?	?	?	?	+57%	+441%	+249%
Heat	Open	All	+28%	+23%	-85%	+1,566%	0	+69%	+2,957%
Terraform	Open	All	+93%	+194%	-61%	-58%	+3,555%	+1,984%	+8,288%

虽然数据不完美，但是足以看到一个明显的趋势：Terraform 和 Ansible 正在经历爆炸性增长。贡献者、得星评级、开放源代码库、StackOverflow 帖子和职位数量的增加都是史无前例。<sup>1</sup> 这两个工具在当今都有庞大而活跃的社区，从这些趋势来看，它们在未来会更加壮大。

## 成熟技术与前沿技术对比

成熟度是进行任何技术选择时都要考虑的另一个关键因素。

表 1-3 为截至 2019 年 5 月 IaC 成熟度对比，显示了每个 IaC 工具的初始发行日期和当前版本号（截至 2019 年 5 月）。

这不是一个苹果对苹果的比较，因为不同的工具使用不同的版本控制方案，但是一些趋势已经很明显。在这个比较中，Terraform 是迄今为止最年轻的 IaC 工具。它仍然是 pre-1.0.0 版本，因此无法保证 API 的稳定性及向后兼容性，而且 bug 相对较多（尽管大多数都是较小的 bug）。这也是 Terraform 的最大弱点，尽管它在很短的时间内就变得非常流行，但是使用这种新的尖端工具所要付出的代价是：它不如其他一些 IaC 工具显得那么成熟。

<sup>1</sup> Terraform 提交和开放问题的减少完全是由我只测量了核心 Terraform 存储库，而在 2017 年，所有提供商程序代码都分离到了单独的存储库中，因此有 100 多个提供商程序存储库中的大量活动没有计入统计。

表 1-3：截至 2019 年 5 月 IaC 成熟度对比

	首次发布	当前版本
Puppet	2005	6.0.9
Chef	2009	12.19.31
CloudFormation	2011	???
SaltStack	2011	2019.2.0
Ansible	2012	2.5.5
Heat	2012	12.0.0
Terraform	2014	0.12.0

### 同时使用多个工具

在本章中我一直在比较 IaC 工具，每种工具都有各自的优点和缺点，所以在实际工作中，根据应用场景正确地选择工具，并同时组合多个工具来构建基础设施，将成为工作重点。

下面是 3 种在许多公司中都很常见的工具组合。

#### 服务开通工具+配置管理工具

例如，搭配使用 Terraform 和 Ansible，如图 1-9 所示。你可以使用 Terraform 部署所有基础设施，包括网络拓扑（如虚拟私有云 VPC、子网、路由表）、数据存储（如 MySQL、Redis）、负载均衡器和服务器。然后使用 Ansible 将应用程序部署在这些服务器之上。



图 1-9：搭配使用 Terraform 和 Ansible

这是一种简单的搭配方法，因为不需要额外运行的基础设施（Terraform 和 Ansible 都只有客户端程序），并且有很多方法可以使 Ansible 和 Terraform 一起工作（例如，通过 Terraform 将特殊标签添加到服务器，Ansible 使用这些标签来查找服务器并对其进行配置）。主要缺点是，使用 Ansible 通常意味着正在对可变服务器编写大量过程性代码，随着代码库、基础设施和团队的增长，维护会变得更加困难。

#### 服务开通工具+服务器模板工具

例如，搭配使用 Terraform 和 Packer，如图 1-10 所示。使用 Packer 将应用程序打包为虚拟机映像。然后使用 Terraform 部署：运行这些虚拟机映像的服务器，以及其他基础设施，包括网络拓扑（即 VPC、子网、路由表）、数据存储（如 MySQL、Redis）和负载均衡器。



图 1-10：搭配使用 Terraform 和 Packer

这也是一种简单的方法，不需要运行额外的基础设施（Terraform 和 Packer 都只有客户端程序），你会在本书接下来的部分，充分练习通过 Terraform 去部署虚拟机映像。这是一个针对不可变基础设施的部署模型，后期很容易维护。但是，它也有两个主要缺点：首先，虚拟机可能要花费很长时间来构建和部署，这会降低产品迭代的速度。其次，正如你将在后面的章节中所看到的那样，使用 Terraform 能够实现的部署策略很有限（例如，无法在 Terraform 中原生地实现蓝绿部署），因此最终的解决方案是，要么编写很多复杂的部署脚本，要么转向编排工具，如下所述。

## 服务开通工具+服务器模板+编排工具

例如，搭配使用 Terraform、Packer、Docker 和 Kubernetes，如图 1-11 所示。你可以使用 Packer 创建包括 Docker 和 Kubernetes 服务的虚拟机映像。然后通过 Terraform 部署服务器集群，每个服务器都运行此虚拟机映像，以及其余基础设施，包括网络拓扑（即 VPC、子网、路由表）、数据存储（如 MySQL、Redis）和负载均衡器。最后，当服务器集群启动时，它将形成一个 Kubernetes 集群，使用 Kubernetes 来运行和管理 Docker 应用程序。

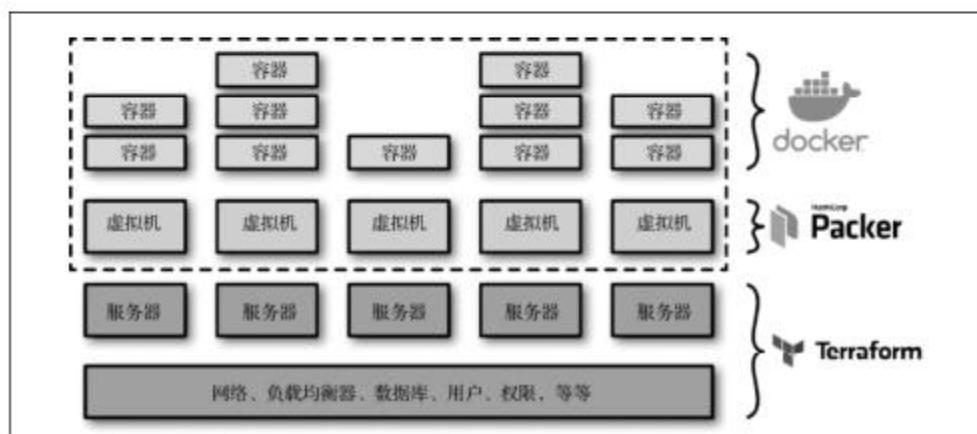


图 1-11：搭配使用 Terraform、Packer、Docker 和 Kubernetes

这种方法的优势在于，Docker 映像的构建速度相当快，你可以在本地计算机上运行和测试它们，并利用 Kubernetes 的所有内置功能，包括各种部署策略、自动修复、自动扩展等。缺点是，这些额外的服务增加了基础设施的复杂性（尽管来自云服务提供商的 Kubernetes 托管服务可以分担部分工作，但 Kubernetes 集群的部署和运维还是比较困难和昂贵的），而且学习、管理和调试这些抽象层（Kubernetes、Docker、Packer）需要花费额外的时间。

## 小结

综上所述，表 1-4 显示了流行的 IaC 工具的使用对比。请注意，该表只列出了使用各种 IaC 工具的默认方式或最常用的方式，如本章前面所述，灵活的 IaC 工具也可以在其他配置方式中使用。例如，可以在没有主控服务器的情况下使用 Chef，也可以使用 Salt 实现不可

变的基础设施。

在 Gruntwork，我们寻找的是一个开源的、支持多云平台的服务开通工具，该工具需要支持不可变的基础设施、声明性语言、无主控服务器模式和无代理软件的体系架构，并拥有庞大的社区和成熟的代码库。如表 1-4 所示，Terraform 尽管不完美，但最符合我们的选择标准。

表 1-4：流行的 IaC 工具的使用对比

	是否 开源	云平 台	类型	基础 设施	语言	代 理	主控服 务器	社区	成熟度
Chef	开源	All	配置管理	可变	过程性 语言	有	有	大	高
Puppet	开源	All	配置管理	可变	声明性 语言	有	有	大	高
Ansible	开源	All	配置管理	可变	过程性 语言	无	无	庞大	中等
SaltStack	开源	All	配置管理	可变	声明性 语言	有	有	大	中等
CloudFormation	私有	AWS	服务开通	不可变	声明性 语言	无	无	小	中等
Heat	开源	All	服务开通	不可变	声明性 语言	无	无	小	低
Terraform	开源	All	服务开通	不可变	声明性 语言	无	无	庞大	低

Terraform 是否也符合你的标准呢？如果是，请继续第 2 章的学习。

## 第 2 章

# Terraform 入门

在本章中，你将开始学习 Terraform 的基础知识。Terraform 是一个容易上手的工具，因此，在大约 40 页的内容里，你将实际操作从运行第一个 Terraform 命令，到使用 Terraform 部署带有负载均衡器的服务器集群，并在服务器之间分配流量等整个流程。该基础设施是运行可扩展的高可用性 Web 服务的良好起点。在随后的章节中，你将进一步开发此示例。

Terraform 可以在多云和混合云环境下工作。例如公共云平台 Amazon Web Services ( AWS )、Azure、Google Cloud 和 DigitalOcean，以及私有云和虚拟化平台（如 OpenStack 和 VMWare）。本章和本书其余部分都将使用 AWS 作为代码示例。AWS 是学习 Terraform 的理想选择，原因如下。

- 迄今为止，AWS 是最受欢迎的云基础设施提供商。它在云基础设施市场中占有 45% 的份额，超过了排名紧随其后的三大竞争对手（微软、谷歌和 IBM）的总和（见参考资料第 2 章[1]）。
- AWS 提供了范围广泛的、可扩展的、可靠的云托管服务，服务包括：可用于部署虚拟服务器的 Amazon Elastic Compute Cloud ( Amazon EC2 )；可以更轻松地管理虚拟服务器集群的 Auto Scaling Groups ( ASGs )；可用于在虚拟服务器集群之间分配流量的 Elastic Load Balancers ( ELB )。<sup>1</sup>
- AWS 提供了慷慨的第一年免费套餐（见参考资料第 2 章[3]），本书中所有示例都可

<sup>1</sup> 如果发现令人困惑的 AWS 术语，请参考简明 AWS（见参考资料第 2 章[2]）。

以在 AWS 免费套餐下运行。即使已经用完了 AWS 免费信用额度，完成书中示例的花费也只需几美元。

如果你以前从未使用过 AWS 或 Terraform，也不用担心，本教程是针对这两种技术的新手而设计的，我将指导用户完成以下步骤。

- 设置 AWS 账户
- 安装 Terraform
- 部署单个服务器
- 部署单个 Web 服务器
- 部署可配置的 Web 服务器
- 部署 Web 服务器集群
- 部署负载均衡器
- 清理工作

## 设置 AWS 账户

如果还没有 AWS 账户，请访问 Amazon 官网并注册。首次注册 AWS 时，你将以 *root* 用户身份登录。该用户具有最高级别访问权限，可以执行账户中的任何操作，因此，从安全角度来看，在日常操作中使用 *root* 用户身份不是一个好主意。事实上，*root* 用户只应该用来创建其他账户和配置账户权限，日常操作应该在受限账户下执行。<sup>1</sup>

要创建受限的账户，需要使用 AWS 身份和访问管理（IAM，*Identity and Access Management*）服务。IAM 是管理账户及用户权限的地方。要创建一个新的 IAM 用户（如图 2-1 所示），请访问 IAM 控制台（见参考资料第 2 章[4]），单击“users”，然后单击“Create User”按钮，输入用户的名称，并确保选择了“Generate an access key for each user”（为每个用户生成访问密钥）（注意，AWS 一直在对其 Web 控制台进行设计上的改动，因此在阅读本书时，IAM 页面的外观可能会略有不同）。

---

<sup>1</sup> 有关 AWS 用户管理最佳实践的更多详细信息，请参阅参考资料第 2 章[5]。



图 2-1：创建一个新的 IAM 用户

单击“Create”按钮，AWS 将显示该用户的安全凭证，如图 2-2 所示，其中包括访问密钥 ID 和访问密钥。请立即保存它们，因为它们今后无法再被显示，本教程后面的部分将会用到该密钥。这个密钥可以访问你的 AWS 账户，因此请将其存放在安全的地方（例如通过密码管理器 1Password、LastPass 或 OS X Keychain 工具来保存），并且永远不要与任何人分享。



图 2-2：AWS 安全凭证（将 AWS 安全凭证存储在安全的地方，不要和任何人分享，不用担心，以上截图中的内容是伪造的）

保存密钥后，单击“Close”按钮，会进入 IAM 用户列表。单击刚创建的用户，然后选择“Permission”选项卡。在默认情况下，新的 IAM 用户没有任何权限，因此无法在 AWS 账户中执行任何操作。

如果要授予 IAM 用户执行某项操作的权限，需要将一个或多个 IAM 策略与该用户的账户相关联。*IAM* 策略是一个 JSON 文档，用于定义用户的权限。你可以创建自己的 IAM 策略，也可以使用一些预定义的 IAM 托管策略。<sup>1</sup>

要运行本书中的示例，请将以下预定义的托管策略添加到你的 IAM 账户，如图 2-3 所示。

- AmazonEC2FullAccess：本章要求
- AmazonS3FullAccess：第 3 章要求
- AmazonDynamoDBFullAccess：第 3 章要求
- AmazonRDSFullAccess：第 3 章要求
- CloudWatchFullAccess：第 5 章要求
- IAMFullAccess：第 5 章要求

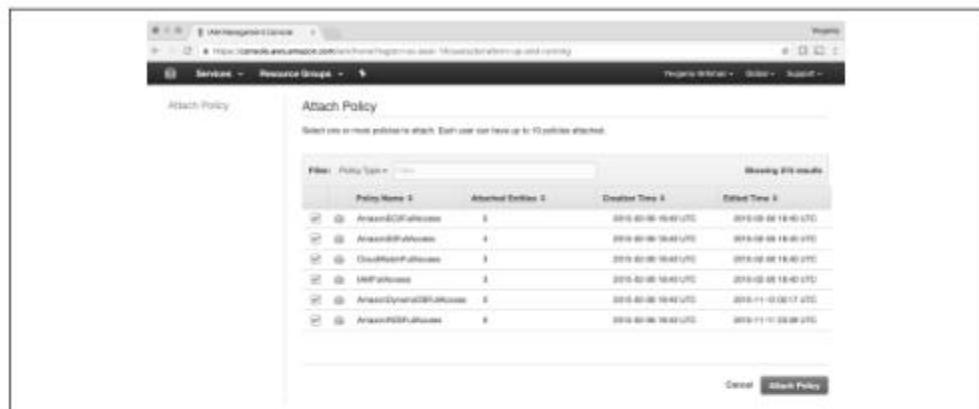


图 2-3：将托管策略添加到你的 IAM 账户

<sup>1</sup> 在 AWS 网站上可以了解有关 IAM 策略的更多信息（见参考资料第 2 章[75]）。



### 关于默认虚拟私有云

如果使用已有的 AWS 账户，请确保该账户中包含默认 VPC (*Virtual Private Cloud*, 虚拟私有云) 设置。VPC 是 AWS 账户中的隔离区域，具有独立的虚拟网络和 IP 地址空间。几乎每个 AWS 资源都需要部署到 VPC 中才能工作，如果未明确指定目标 VPC，则资源将被部署到默认 VPC 中，默认 VPC 是每个新创建的 AWS 账户的一部分。本书中的所有示例都依赖于默认 VPC。因此，如果出于某种原因，之前删除了账户中当前区域的默认 VPC，请尝试切换到其他 AWS 区域（每个区域都有自己独立的默认 VPC）或通过 AWS Web 控制台（见参考资料第 2 章[7]）创建新的默认 VPC。否则你需要针对书中每个示例添加 `vpc_id` 或 `subnet_id` 参数，让示例使用自定义 VPC。

## 安装 Terraform

最直接的方法是，从 Terraform 主页（见参考资料第 2 章[8]）下载 Terraform。单击下载链接，选择操作系统对应的软件安装包，下载 ZIP 文件，然后将其解压缩到要安装 Terraform 的目录中。压缩文件中包括一个名为 `terraform` 的二进制文件，最好将解压后文件的目录添加到 PATH 环境变量中。也可以尝试通过操作系统的程序管理器来安装 Terraform。例如在 OS X 上，可以通过运行 `brew install terraform` 命令安装 Terraform。

要检查一切是否正常，请运行 `terraform` 命令，查看用法说明如下。

```
$ terraform
Usage: terraform [-version] [-help] <command> [ args ]
Common commands:
  apply      Builds or changes infrastructure
  console    Interactive console for Terraform interpolations
  destroy   Destroy Terraform-managed infrastructure
  env       Workspace management
  fmt        Rewrites config files to canonical format
  (...)
```

为了使 Terraform 能够对你的 AWS 账户进行直接操作，需要将环境变量 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY` 设置为之前创建的 IAM 用户的访问 ID 和密钥。例如，下面是在 UNIX / Linux / macOS 终端中进行设置的方法。

```
$ export AWS_ACCESS_KEY_ID = (访问 ID)  
$ export AWS_SECRET_ACCESS_KEY = (密钥)
```

请注意，这些环境变量仅在当前终端会话进程中有效，如果重新启动计算机或打开新的终端窗口，则需要再次运行以上命令。



#### 身份验证选项

除了通过环境变量进行身份验证，Terraform 也支持与所有 AWS 命令行工具和 SDK 工具相同的身份验证机制，使用 `$HOME/.aws/credentials` 文件中的密钥，密钥信息可以通过在 AWS 命令行或 IAM 角色上运行 `configure` 命令自动生成。更多有关信息，请参阅通过命令行对 AWS 进行身份验证的综合指南（见参考资料第 2 章[9]）。

## 部署单个服务器

Terraform 代码是以 *HashiCorp* 配置语言 (*HashiCorp Configuration Language, HCL*) 编写的，扩展名为 `.tf`。<sup>1</sup> HCL 是一种声明性语言，目标是描述所需的基础设施，Terraform 将自动计算生成创建它的方法。Terraform 可以跨各种平台和提供商（包括 AWS、Azure、Google Cloud、DigitalOcean 等）创建基础设施。

用户可以使用任何文本编辑器编写 Terraform 代码。大多数编辑器也提供对 Terraform 语法的支持（需要在网上搜索单词“HCL”而不是“Terraform”来获取相关信息），包括 vim、Emacs、Sublime Text、Atom、Visual Studio Code 和 IntelliJ（IntelliJ 甚至支持重构、查找用法和声明跳转）。

使用 Terraform 的第一步通常是配置要使用的提供商。创建一个空文件夹，并在其中放置一个名为 `main.tf` 的文件，该文件包含以下内容。

```
provider "aws" {  
    region = "us-east-2"  
}
```

---

<sup>1</sup> 你也可以在扩展名为 `.tf.json` 的文件中以纯 JSON 编写 Terraform 代码。可以在 Terraform 网站上了解更多关于 Terraform 的 HCL 和 JSON 语法的信息（见参考资料第 2 章[10]）。

这里告诉 Terraform 将使用 AWS 作为服务提供商，并且要将基础设施部署到 `us-east-2` 区域。AWS 的数据中心遍布世界各地，位于相同地理区域的多个数据中心，被统一命名为一个区域(`region`)。例如 `us-east-2`(俄亥俄州)、`eu-west-1`(爱尔兰)和 `ap-southeast-2`(悉尼)。

每个区域包括多个相互隔离的数据中心，称为可用区 (*Availability Zones, AZ*)，例如 `us-east-2a`、`us-east-2b` 等。<sup>1</sup>

对于每种类型的服务提供商，你可以创建许多不同种类的资源，例如服务器、数据库和负载均衡器。在 Terraform 中创建资源的一般语法如下。

```
resource "<PROVIDER>_<TYPE>" <NAME> {
    [CONFIG ...]
}
```

其中 `PROVIDER` 是提供商的名称(例如 `aws`)。`TYPE` 是在该提供商中创建的资源类型(例如 `instance`)。`NAME` 是一个标识符，你可以在整个 Terraform 代码块范围内通过这个标识符引用该资源(例如 `my_instance`)。`CONFIG` 包括一个或多个特定于该资源的参数或参数组。

例如，如果想在 AWS 中部署单个 `EC2`(虚拟)服务器实例，则需要将 `aws_instance` 资源配置在 `main.tf` 中。

```
resource "aws_instance" "example" {
    ami = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}
```

`aws_instance` 资源支持许多不同的参数，但只有两个参数是必须要设置的。

#### ami

运行在 `EC2` 实例上的 Amazon Machine Image (AMI)。用户可以在 AWS Marketplace (见参考资料第 2 章[12]) 中找到免费和付费的 AMI，也可以使用诸如 Packer 之类的工具创建自己的 AMI(请参阅第 1 章的“服务器模板工具”有关机器映像和服务器模板的讨论)。前面代码示例中的 `ami` 参数指向的是 AMI ID，代表了位于 `us-east-2` 区域的 Ubuntu 18.04 免费映像。

---

<sup>1</sup> 用户可以在 AWS 网站上了解有关 AWS 区域和可用区的更多信息(见参考资料第 2 章[11])。

### `instance_type`

EC2 运行实例的类型。每种类型的 EC2 实例都提供不同数量的 CPU、内存、磁盘空间和网络带宽。EC2 实例类型（见参考资料第 2 章[13]）的页面列出了所有可选配置。之前的示例使用 t2.micro 型号，它具有一个虚拟 CPU、1GB 内存，并且属于 AWS 免费套餐的一部分。



### 使用文档！

Terraform 支持数十个服务提供商，我们很难记住每个提供商所支持的众多资源，以及每个资源的全部参数。在编写 Terraform 代码时，你需要定期参考 Terraform 文档，查找可供使用的资源及其使用方法。例如，这是关于 `aws_instance` 资源的文档（见参考资料第 2 章[14]）。尽管已经使用 Terraform 很多年了，我仍然需要经常查看这些文档！

打开一个终端，进入创建 `main.tf` 的文件夹，然后运行 `terraform init` 命令。

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (terraform-providers/aws) 2.10.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "... constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = ">= 2.10"

Terraform has been successfully initialized!
```

`terraform` 可执行文件包含 Terraform 的基本功能，但它并不包含任何服务提供商（例如 AWS 提供商、Azure 提供商、GCP 提供商等）的代码，所以第一次开始使用 Terraform 时，需要运行 `terraform init` 命令，指示 Terraform 扫描代码，找出用到的提供商，并下载

它们需要使用的代码库。在默认情况下，提供商代码将被下载到`.terraform`文件夹中，该文件夹是 Terraform 的临时目录（用户或许需要将其添加到`.gitignore`，以防止将这个临时目录上传到版本控制系统）。在后面的章节中，你还会看到一些使用`init`命令和`.terraform`文件夹的介绍。每次在使用新的 Terraform 代码时，都需要先运行`init`命令，这个命令可以安全地多次运行（命令是幂等的）。

现在你已经下载了提供商代码，请运行`terraform plan`命令。

```
$ terraform plan

(...)

Terraform will perform the following actions:

# aws_instance.example 将被创建
+ resource "aws_instance" "example" {
    + ami                      = "ami-0c55b159cbfafe1f0"
    + arn                      = (known after apply)
    + associate_public_ip_address = (known after apply)
    + availability_zone        = (known after apply)
    + cpu_core_count           = (known after apply)
    + cpu_threads_per_core     = (known after apply)
    + get_password_data        = false
    + host_id                  = (known after apply)
    + id                       = (known after apply)
    + instance_state           = (known after apply)
    + instance_type             = "t2.micro"
    + ipv6_address_count       = (known after apply)
    + ipv6_addresses            = (known after apply)
    + key_name                 = (known after apply)
    ...
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

`plan`命令可以让你在任何实际更改之前对 Terraform 进行预览，以便代码在发布给外界之前进行最后的检查。`plan`命令的输出，类似于 UNIX、Linux 和 Git 中用过的`diff`命令：加号（+）代表任何新添加的内容，减号（-）代表删除的内容，波浪号（~）代表所有将被修改的内容。在前面的输出中，你可以看到 Terraform 计划按要求创建单个 EC2 的实例。

要创建这个实例，请运行 `terraform apply` 命令。

```
$ terraform apply  
(...)  
  
Terraform will perform the following actions:  
  
# aws_instance.example 将被创建  
+ resource "aws_instance" "example" {  
    + ami                  = "ami-0c55b159cbfafe1f0"  
    + arn                  = (known after apply)  
    + associate_public_ip_address = (known after apply)  
    + availability_zone     = (known after apply)  
    + cpu_core_count        = (known after apply)  
    + cpu_threads_per_core  = (known after apply)  
    + get_password_data     = false  
    + host_id               = (known after apply)  
    + id                   = (known after apply)  
    + instance_state        = (known after apply)  
    + instance_type         = "t2.micro"  
    + ipv6_address_count    = (known after apply)  
    + ipv6_addresses        = (known after apply)  
    + key_name              = (known after apply)  
    (...)  
}  
  
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.
```

```
Enter a value:
```

你会注意到，`apply` 命令显示了和 `plan` 命令一模一样的输出，并要求确认是否要继续执行该计划。因此，虽然 `plan` 是一个单独的命令，但它的主要应用场景是快速完整性检查，以及代码评审（我们将在第 8 章重点介绍），所以在日常使用当中，大部分时间是直接运行 `apply` 命令，并对输出内容进行审查。

输入 `yes`，然后按 Enter 键部署 EC2 实例。

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

```
aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Still creating... [30s elapsed]
aws_instance.example: Creation complete after 38s [id=i-07e2a3e006d785906]
```

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

恭喜你，刚刚成功地使用 Terraform 在 AWS 账户中部署了一个 EC2 实例！要验证这一点，请转到 EC2 控制台（见参考资料第 2 章[23]），将会看到类似于图 2-4 所示的单个 EC2 实例。

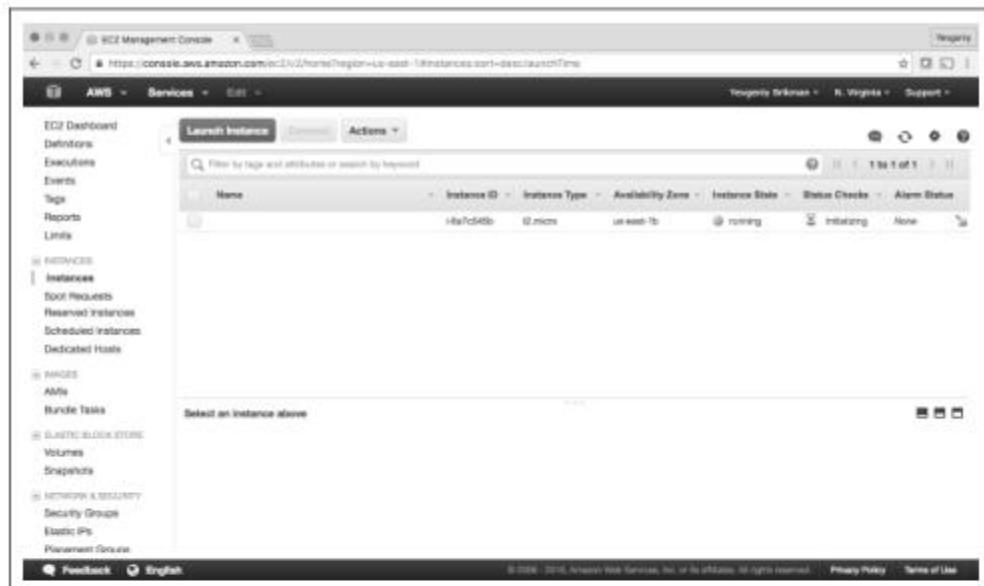


图 2-4：单个 EC2 实例

该实例的确在那里！现在让我们把它变得更有趣一些。首先我们注意到，EC2 实例没有名称。要添加一个名称，可以向 `aws_instance` 资源添加 `tags` 标签。

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = [
    Name = "terraform-example"
  ]
}
```

再次运行 `terraform apply` 命令，将会看到如下输出。

```
$ terraform apply

aws_instance.example: Refreshing state...
(...)

Terraform will perform the following actions:

# aws_instance.example 将被更新
~ resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  availability_zone= "us-east-2b"
  instance_state  = "running"
  ...
  +
  + tags = [
    + "Name" = "terraform-example"
  ]
  ...
}

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value:
```

Terraform 会跟踪已经为这组配置文件创建过的所有资源，所以它知道你的 EC2 实例已经

存在（注意当你运行 `apply` 命令时，Terraform 显示输出 `Refreshing state...`），它同时列出了当前部署的内容与 Terraform 代码中的内容之间的区别（如第 1 章“Terraform 与其他 IaC 工具的比较”中所述，这是与过程性语言相比，声明性语言的优势之一）。前面的比较显示 Terraform 想创建一个名为 `Name` 的标签，输入 `yes` 并按 Enter 键。

刷新 EC2 控制台，你会看到 EC2 实例现在拥有一个名称标签，如图 2-5 所示。

现在，你已经创建了有效的 Terraform 代码，接下来你可能希望将其存储在版本控制系统中。这样就可以与其他团队成员分享此代码，跟踪所有基础设施的变化，并使用提交日志进行错误调试了。下面是如何创建本地 Git 代码库并使用它来存储 Terraform 配置文件的方法。



图 2-5：EC2 实例现在拥有一个名称标签

```
git init  
git add main.tf  
git commit -m "Initial commit"
```

你还需要创建一个名为 `.gitignore` 的文件，它会告诉 Git 忽略某些类型的文件，以免你无意

中将临时文件存入版本控制系统中。

```
.terraform  
* .tfstate  
* .tfstate.backup
```

前面的 `.gitignore` 文件的内容，指示 Git 忽略 Terraform 临时目录 `.terraform` 文件夹，以及 Terraform 用来存储状态的 `*.tfstate` 文件（在第 3 章中，你将了解到为什么不应该提交状态文件）。你还需要提交这个 `.gitignore` 文件。

```
git add .gitignore  
git commit -m "Add .gitignore file"
```

要与团队成员共享此代码，则需要创建一个可以同时访问的共享 Git 存储库。一种方法是使用 GitHub。登录 GitHub 网站，如果你还没有账号，请先创建一个账号，然后创建一个新的存储库。在你的本地 Git 存储库中，将新建 GitHub 存储库配置成名称为 `origin` 的远程节点，如下所示。

```
git remote add origin git@github.com: <YOUR_USERNAME> / <YOUR_REPO_NAME> .git
```

从现在开始，你可以通过 `push` 命令将代码推送到 `origin` 节点，与你的团队成员共享代码了。

```
git push origin master
```

当你希望看到团队成员所进行的更改时，也可以通过 `pull` 命令从 `origin` 获取更新的代码。

```
git pull origin master
```

在本书的其余部分，以及平时使用 Terraform 时，请确保代码更改后定期运行 `git commit` 和 `git push` 命令。这样，你不仅可以与团队成员进行代码协作，还可以将所有基础设施的更改捕获到提交日志中以便于调试。第 8 章将详细介绍如何在团队环境中使用 Terraform。

## 部署单个 Web 服务器

接下来我们将在之前部署的实例上运行一个 Web 服务器。目标是部署一个最简单的网站架构——一个可以响应 HTTP 请求的单个 Web 服务器，如图 2-6 所示。

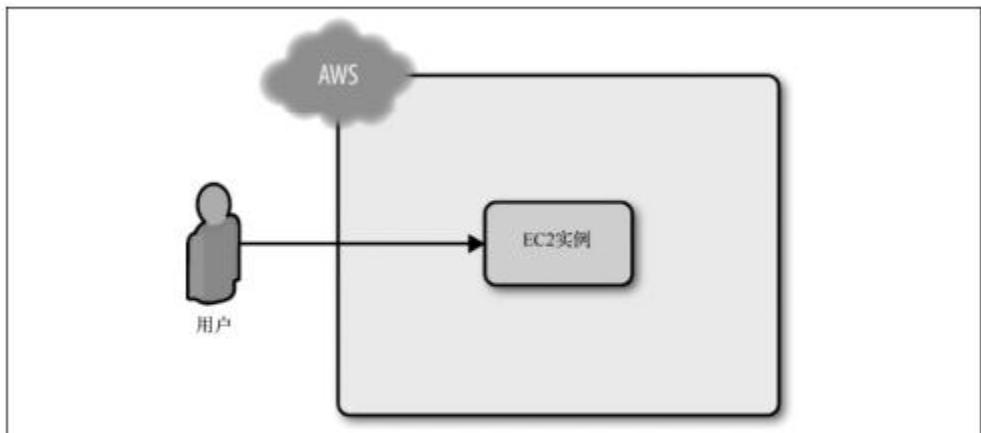


图 2-6：一个可以响应 HTTP 请求的单个 Web 服务器

在真实的用例中，你可能会使用 Ruby on Rails 或 Django 等 Web 框架来构建 Web 服务器，但在书中为了使示例浅显易懂，我们只运行一个非常简单的 Web 服务器，该服务器始终返回文本 “Hello,World”。<sup>1</sup>

```

#!/bin/bash
echo "Hello,World" > index.html
nohup busybox httpd -f -p 8080 &

```

这是一个 Bash 脚本，将文本 “Hello,World” 写入 *index.html*，并运行一个名为 *busybox*（见参考资料第 2 章[16]）的工具，该工具默认已经在 Ubuntu 上安装，并会在端口 8080 上启动 Web 服务器。我在 *busybox* 命令行中使用了 *nohup* 和 &（后台运行符）以便在 Bash 脚本执行退出后，Web 服务器仍会在后台持续运行。



### 端口号

该示例使用 8080 端口，代替默认的 HTTP 80 端口。原因是侦听任何小于 1024 的端口都需要 root 用户特权。这存在安全风险，因为任何设法突破服务器的攻击者也都将获得 root 特权。因此，最佳做法是使用权限受限的非 root 用户来运行 Web 服务器。这意味着你必须使用大于 1024 的较高的端口。在本章后

<sup>1</sup> 如果需要，用户可以在 GitHub 上找到一些极其简单的只有一行代码的 HTTP 服务器示例（见参考资料第 2 章[17]）。

面的部分，你会学到如何配置负载均衡器来侦听 80 端口并将流量路由到服务器上较高的端口，如 8080。

接下来，如何让 EC2 实例来运行此脚本？通常，如第 1 章的“服务器模板工具”中所述，需要使用 Packer 之类的工具来创建自定义 AMI，并在其中安装 Web 服务器工具。但是由于本示例中的 Web 服务器只是通过单行 busybox 命令创建的简化版 Web 服务器，因此你可以直接使用标准的 Ubuntu 18.04 AMI 映像，并将“Hello,World”脚本通过 EC2 实例的用户数据 (*user data*) 参数来执行。EC2 实例启动时，将执行以用户数据方式传递进来的 Shell 脚本或 cloud-init 指令。Terraform 脚本使用 *user\_data* 参数，将 Shell 脚本作为对象，传递给 AMI 用户数据，如下所示。

```
resource "aws_instance"" example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  user_data = <<-EOF
    #! / bin / bash
    echo "Hello, World" >index.html
    nohup busybox httpd -f -p 8080 &
    EOF

  tags      = {
    Name    ="terraform-example"
  }
}

<< - EOF 和 EOF 是 Terraform 的 heredoc 语法，它可以让用户在无须插入换行符的情况下，创建多行字符串。
```

在默认情况下，AWS 阻止 EC2 实例的任何入站或出站流量，为了能够让 Web 服务器开始工作，需要开放 EC2 实例在端口 8080 上的入站流量。首先需要创建一个安全组 (*security group*)。

```
resource "aws_security_group"" instance" {
  name        = "terraform-example-instance"

  ingress {
    from_port     = 8080
```

```
        to_port      = 8080
        protocol    = "TCP"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

该代码创建一个名为 `aws_security_group` 的新资源（注意所有 AWS 服务提供商的资源都包含 `aws_` 前缀），定义该安全组允许来自 CIDR 块 `0.0.0.0/0`，针对端口 `8080` 上的 TCP 入站请求。CIDR 块是一种定义 IP 地址范围的简洁方法。例如，`CIDR` 块 `10.0.0.0/24` 表示介于 `10.0.0.0` 和 `10.0.0.255` 之间的所有 IP 地址，`CIDR` 块 `0.0.0.0/0` 代表所有可能的 IP 地址的范围，因此，此安全组允许端口 `8080` 上的来自任何 IP 的入站请求。<sup>1</sup>

仅仅创建安全组是不够的，你还需要将安全组的 ID 传递到实际使用它的 EC2 实例中。具体方式是使用 `aws_instance` 资源中的 `vpc_security_group_ids` 参数。为此，首先需要了解 Terraform 表达式。

Terraform 中任何具有返回值的对象都被称为表达式。你已经看到了最简单的表达式类型，如字符串（如 "`ami-0c55b159cbfafe1f0`"）和数字（如 `5`）。本书后面还会介绍 Terraform 支持的其他类型的表达式。

引用（*reference*）是一种特别有用的表达式类型，它使用户可以从代码的其他部分访问该值。如果要访问安全组资源的 ID，需要使用资源属性引用（*resource attribute reference*），该引用的语法如下。

```
<PROVIDER>_.<TYPE>.<NAME>.<ATTRIBUTE>
```

其中 `PROVIDER` 是提供商的名称（如 `aws`），`TYPE` 是资源的类型（如 `security_group`），`NAME` 是该资源的名称（如安全组名为 `instance`），以及 `ATTRIBUTE` 是该资源自身的参数（如 `name`）或该资源输出的属性（用户可以在文档中找到每种资源的可用属性列表）。`id` 就是安全组输出的属性之一，引用它的表达式将如下所示。

```
aws_security_group.instance.id
```

---

<sup>1</sup> 要了解更多有关 CIDR 是如何工作的内容，请参考维基百科（见参考资料第 2 章[18]）。若要使用 IP 地址范围和 CIDR 表示法之间转换的便捷计算器，请在浏览器中访问参考资料第 2 章[19]或在终端中安装 `ipcalc` 工具。

你可以在 `aws_instance` 资源的 `vpc_security_group_ids` 参数中使用这个安全组 ID。

```
resource "aws_instance" "example" {
  ami                  = "ami-0c55b159cbfafe1f0"
  instance_type        = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
    #! / bin / bash
    echo "Hello,World" >index.html
    nohup busybox httpd -f -p 8080 &
  EOF

  tags   = {
    Name = "terraform-example"
  }
}
```

当在一个资源内引用另一个资源时，会创建隐式依赖关系。Terraform 可以通过分析这些依赖关系，构建依赖关系图，并使用该关系图自动确定资源的创建顺序。如果你从零部署这个代码，Terraform 知道它需要在创建 EC2 实例之前先创建安全组，因为 EC2 实例引用了安全组的 ID。可以通过运行 `terraform graph` 命令显示依赖关系图。

```
$ terraform graph

digraph {
  compound = " true"
  newrank = " true"
  subgraph "root" {
    "[root] aws_instance.example"
    [label = "aws_instance.example", shape = " box"]
    "[root] aws_security_group.instance"
    [label = "aws_security_group.instance", shape = " box"]
    "[root] provider.aws"
    [label = "provider.aws", shape = " diamond"]
    "[root] aws_instance.example" ->
    "[root] aws_security_group.instance"
    "[root] aws_security_group.instance" ->
    "[root] provider.aws"
```

```

    "[root] meta.count-boundary (EachMode fixup) " ->
        "[root] aws_instance.example"
    "[root] provider.aws (close) " ->
        "[root] aws_instance.example"
    "[root] root" ->
        "[root] meta .count-boundary (EachMode fixup) "
    "[root] root" ->
        "[root] provider.aws (close) "
}

```

以上输出的格式为 DOT 图形描述语言，通过使用桌面应用，例如 Graphviz，或 Web 应用 GraphvizOnline（见参考资料第 2 章[20]）等工具，可以自动生成一个类似图 2-7 所示的 EC2 实例及其安全组的依赖关系图。

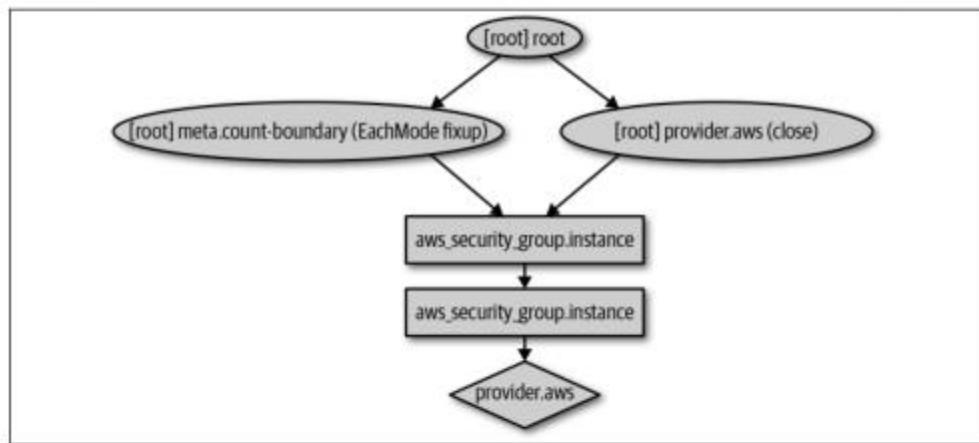


图 2-7：EC2 实例及其安全组的依赖关系图

当 Terraform 遍历你的依赖关系树时，它会尽量使用并行方式创建尽可能多的资源，从而更有效地实现变更。这就是声明性语言的魅力：用户只需指定自己想要的，Terraform 便会以最有效的方式加以实现。

此时运行 `apply` 命令，Terraform 会创建一个安全组并将已存在的 EC2 实例，替换为包括新增用户数据参数的新实例。

```
$ terraform apply

(...)

Terraform will perform the following actions:

# aws_instance.example 必须被替换
-/+resource" aws_instance"" example" {
    ami                      = "ami-0c55b159cbfafe1f0"
    -availability_zone        = "us-east-2c"-> (known after apply)
    -instance_state           = "正在运行"-> (known after apply)
    instance_type             = "t2.micro"
    (...)

    + user_data               = "c765373 ..." # 强制替换
    -volume_tags              = {}-> (known after apply)
    -vpc_security_group_ids   = [
        -"sg-871fa9ec",
    ]-> (known after apply)
    (...)

}

# 将创建 aws_security_group.instance
+ resource" aws_security_group"" instance" {
    + arn                     = (known after apply)
    + description             = "Manage by Terraform"
    + egress                  = (known after apply)
    + id                      = (known after apply)
    + ingress                 = [
        +
            + cidr_blocks      = [
                +" 0.0.0.0/0",
            ]
            + description     = ""
            + from_port       = 8080
            + ipv6_cidr_blocks = []
            + prefix_list_ids = []
            + protocol         = "tcp"
            + security_groups  = []
            + self             = false
            + to_port          = 8080
        ],
    ]
}
```

```
+name          = "terrain-example-instance"
+owner_id      = (known after apply)
+revoke_rules_on_delete = false
+vpc_id        = (known after apply)
}

plan: 2 to add, 0 to change, 1 to destroy.
```

```
Do you want to perform these actions ?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value:
```

计划输出中的`+`符号表示“替换”。通过在计划输出中查找 `forces replacement`（强制替换）字段，可以得到 Terraform 进行强制替换的原因。`aws_instance` 资源中的许多参数如果发生改变，都会强制替换资源，这意味着原来的 EC2 实例将被终止，一个全新的实例将被创建。这一点和在第 1 章的“服务器模板工具”中讨论过的不可变基础设施的概念是一致的。因为更新了 Web 服务器，服务器的所有用户都会经历停机的情况。第 5 章会介绍如何使用 Terraform 进行零停机部署。

在之前的示例中，如果用户输入 `yes`，就会在 AWS 控制台中看到新的 EC2 实例正在被部署，带有 Web 服务器代码的新 EC2 实例将替换旧实例，如图 2-8 所示。

如果单击新创建的实例，可以在屏幕底部的描述面板中找到公共 IP 地址。启动新实例将需要一到两分钟的时间，然后用户就可以使用 Web 浏览器或 curl 之类的工具对该 IP 地址的 8080 端口发送 HTTP 请求了。

```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello,World
```

现在，你已经拥有了一个在 AWS 中可运行的 Web 服务器！

The screenshot shows the AWS EC2 Management Console. On the left, there's a sidebar with navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances (with sub-links for Instances, Spot Requests, Reserved Instances, Scheduled Instances, Dedicated Hosts), S3 Buckets, AMIs, Bundle Tasks, and various storage options like PLAIN BLOCK STORE, Volumes, and Snapshots. The main content area shows a table of instances. The table has columns for Name, Instance ID, Instance Type, Availability Zone, Instance State, Status Checks, and Alarm State. There are two entries: one named 'terraform-example' with Instance ID i-055ebab4, running in us-east-1a, and another named 'terraform-example' with Instance ID i-055ebab5, terminated in us-east-1b. Below the table, a detailed view for the first instance is shown with tabs for Description, Status Checks, Monitoring, and Tags. The description tab displays information such as Instance ID (i-055ebab4), Instance state (running), Instance type (t2.micro), and Private DNS (ip-172-31-81-03.ec2.internal). It also shows Public DNS (ec2-54-237-303-240.compute-1.amazonaws.com), Public IP (54.237.265.240), and Availability zone (us-east-1a).

图 2-8：带有 Web 服务器代码的新 EC2 实例将替换旧实例



## 网络安全

为了使本书中的所有示例都简单易懂，它们不仅将被部署到默认 VPC（如前所述）中，还会被部署到该 VPC 的默认子网中。VPC 分为一个或多个子网，每个子网都有自己的 IP 地址。默认 VPC 中的子网都是公共子网，这意味着它们都具有从公共 Internet 可以访问到的 IP 地址。这也是用户可以从家用计算机试验 EC2 实例的原因。

在公共子网中运行服务器可以快速进行实验，但是在实际使用中存在安全风险。全世界的黑客都在不断地随机扫描 IP 地址以发现任何弱点。如果你的服务器是公开暴露的，那么单个不受保护的端口或运行带有已知漏洞的过时代码，都可以让黑客侵入。

因此，对于生产环境系统，请将所有服务器和数据存储部署在专用子网中，这些子网的 IP 地址只能从 VPC 内部访问，而不能从公共 Internet 访问。公共子网中运行的少数服务器只应该是反向代理服务器和负载均衡器，而且它们也要尽可能地被保护起来（本章稍后将看到有关如何部署负载均衡器的示例）。

## 部署可配置的 Web 服务器

你可能已经注意到，Web 服务器代码中安全组和用户数据配置被同时配置为 8080 端口。这违反了不要重复自己（*Don't Repeat Yourself, DRY*）的原则：每条知识在系统中必须具有唯一、明确、权威的表示形式。<sup>1</sup>如果将端口号配置在两个地方，则很容易在一个地方进行了更新，而忘记在另一个地方进行相同的更改。

为了使代码更 DRY 化和可配置化，Terraform 允许用户定义输入变量。下面是声明变量的语法。

```
variable "NAME" {  
    [CONFIG ...]  
}
```

变量声明的主体可以包含 3 个参数，所有这些参数都是可选参数。

### description

描述参数用来说明如何使用这个变量。团队成员在阅读代码时，或者通过运行 `plan` 或 `apply` 命令都能够看到这个描述信息（稍后将举例说明）。

### default

有多种方法可以为变量赋值，包括通过命令行（使用 `-var` 选项），通过属性文件（使用 `-var-file` 选项）或通过环境变量（Terraform 能够查找并识别前缀为 `TF_VAR_<variable_name>` 的环境变量）。如果未传入任何值，变量将使用默认值。如果没有默认值，Terraform 将以交互方式提示用户输入一个值。

### type

允许对用户输入的变量类型进行强制约束。Terraform 支持许多类型约束，包括 `string`、`number`、`bool`、`list`、`map`、`set`、`object`、`tuple` 和 `any`。如果未指定类型，那么 Terraform 会设置默认约束类型为 `any`。

这是一个输入变量的示例，该输入变量通过类型约束来验证输入的值为数字。

---

<sup>1</sup> Andy Hunt, Dave Thomas. *The Pragmatic Programmer*. [美]新泽西: Addison-Wesley Professional, 1999.

```
variable "number_example" {
  description = "An example of a number variable in Terraform"
  type        = number
  default     = 42
}
```

这是一个变量示例，它会检查输入值是否为列表。

```
variable "list_example" {
  description = "An example of a list in Terraform"
  type        = list
  default     = ["a", "b", "c"]
}
```

类型约束也可以搭配使用。例如，这是一个列表输入变量，要求所有的列表值为数字。

```
variable "list_numeric_example" {
  description = "An example of a numeric list in Terraform"
  type        = list(number)
  default     = [1, 2, 3]
}
```

这是一个要求所有值均为字符串的映射对象。

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type        = map(string)

  default     = {
    key1      = "value1"
    key2      = "value2"
    key3      = "value3"
  }
}
```

你还可以使用类型约束创建更复杂的对象和元组结构类型。

```
variable "object_example" {
  description= "An example of a structural type in Terraform"
  type       = object({
    name      = string
    age       = number
    tags      = list(string)
    enabled   = bool
  })
}
```

```
        })
      default = {
        name    = "value1"
        age     = 42
        tags    = ["a", "b", "c"]
        enabled = true
      }
    }
```

下面的示例创建一个输入变量模型，该变量键值包括：`name`（必须为字符串）、`age`（必须为数字）、`tags`（必须为字符串列表）和`enabled`（必须为布尔值）。如果尝试将此变量设置为与类型约束不匹配的值，Terraform 将报告类型错误。下面的示例尝试将`enabled`变量设置为字符串而不是一个布尔值。

```
variable "object_example_with_error" {
  description = "An example of a structural type in Terraform with an error"
  type        = object({
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })
  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = "invalid"
  }
}
```

会得到以下错误。

```
$ terraform apply
Error: Invalid default value for variable

on variables.tf line 78, in variable "object_example_with_error":
78: default = {
79:   name    = "value1"
80:   age     = 42
81:   tags    = ["a", "b", "c"]
82:   enabled = "invalid"
83: }
```

```
This default value is not compatible with the variable's type constraint: a  
bool is required.
```

在 Web 服务器的示例中，你需要一个存储端口号的变量。

```
variable "server_port" {  
    description = "The port the server will use for HTTP requests"  
    type        = number  
}
```

请注意，`server_port` 输入变量没有默认值，因此，如果运行 `apply` 命令，Terraform 将提示输入 `server_port` 的值，并显示变量中的 `description` 值。

```
$ terraform apply  
  
var.server_port  
  The port the server will use for HTTP requests  
  
Enter a value:
```

如果不想要每次都处理交互式提示，可以通过命令行的 `-var` 参数为输入变量提供初始值。

```
$ terraform plan -var "server_port = 8080"
```

也可以通过环境变量来设置输入变量初始值。命名规范是 `TF_VAR_<name>`，其中 `<name>` 是你要设置的输入变量的名称。

```
$ export TF_VAR_server_port = 8080  
$ terraform plan
```

如果不想在每次运行 `plan` 或 `apply` 时都记住额外的命令行参数，也可以指定一个默认值。

```
variable "server_port" {  
    description = "The port the server will use for HTTP requests"  
    type        = number  
    default     = 8080  
}
```

要在 Terraform 代码中使用输入变量的值，可以使用一种被称为变量引用（*variable reference*）的新型表达式，语法如下。

```
var.<VARIABLE_NAME>
```

下面是如何将安全组资源的 `from_port` 和 `to_port` 参数，设置为变量 `server_port` 的值的示例。

```
resource "aws_security_group" "instance" {
    name          = "terraform-example-instance"

    ingress {
        from_port   = var.server_port
        to_port     = var.server_port
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

在用户数据脚本中设置端口时，最好使用相同的输入变量。要在字符串文字中使用变量引用，需要通过一种被称为插值（*interpolation*）的表达式，其语法如下。

```
"${...}"
```

用户可以在花括号中放置任何有效的变量引用，Terraform 会把它转换为字符串。例如，使用以下方法可以将 `var.server_port` 的取值作为字符串插入到用户数据中。

```
user_data = <<-EOF
#!/bin/bash
echo "Hello,World" > index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

除了输入变量，Terraform 还允许通过使用以下语法来定义输出变量。

```
output "<NAME>" {
    value = <VALUE>
    [CONFIG ...]
}
```

`NAME` 是输出变量的名字，`VALUE` 是任何你希望输出的 Terraform 表达式。`CONFIG` 包含两个可选参数。

#### description

描述参数可以被用来记录输出变量的数据类型。

#### `sensitive`

如果此参数设置为 `true`, Terraform 在运行 `terraform apply` 指令时, 不会在日志中记录输出信息。这是一种非常有用的方式, 可以用来防止记录输出变量中的敏感信息, 例如密码或私钥。

举例来说, 你可以将新创建的服务器的 IP 地址作为变量输出在命令行上, 而不必像之前一样, 手动通过 EC2 控制台进行查找。

```
output "public_ip" {  
    value      = aws_instance.example.public_ip  
    description = "The public IP address of the web server"  
}
```

这段代码再次使用了资源属性引用的功能, 这次引用了 `aws_instance` 资源的 `public_ip` 属性。如果再次运行 `apply` 命令, Terraform 将不会进行任何更改 (因为你尚未更改任何资源), 但是它将在命令运行结果的最后显示新的输出, 如下所示。

```
$ terraform apply  
(...)  
  
aws_security_group.instance: Refreshing state... [id=sg-078ccb4f9533d2c1a]  
aws_instance.example: Refreshing state... [id=i-028cad2d4e6bddec6]  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
public_ip = 54.174.13.5
```

如你所见, 运行 `terraform apply` 命令之后, 输出变量将显示在命令运行结果之后, 这对于使用 Terraform 代码的用户来说, 是非常有用的功能 (知道了在部署 Web 服务器之后需要测试的 IP 地址)。你还可以在无须进行任何更改的情况下, 使用 `terraform output` 命令列出所有的输出变量。

```
$ terraform output  
public_ip = 54.174.13.5
```

运行 `terraform output <OUTPUT_NAME>` 命令来查看名为 `<OUTPUT_NAME>` 的特定输出变量的取值。

```
$ terraform output public_ip  
54.174.13.5
```

这会方便你编写复杂的脚本。例如，你可以创建一个部署脚本，脚本通过运行 `terraform apply` 命令来部署 Web 服务器，之后使用 `terraform output public_ip` 命令来获取公共 IP 地址，最后针对该 IP 地址，运行 `curl` 命令来检验部署是否生效。

输入变量和输出变量，也是创建可配置和可重用的基础设施代码的重要组成部分，在第 4 章中会进一步介绍该主题。

## 部署 Web 服务器集群

运行单个服务器是一个好的开始，但是在现实世界中，单个服务器意味着单点故障。在服务器崩溃或流量过大而导致服务器过载的情况下，用户将无法访问站点。解决方案是运行服务器集群，及时剔除发生故障的服务器，并根据流量调整集群大小。<sup>1</sup>

手动管理这样的集群是个繁重的工作。幸运的是用户可以使用 AWS 的 Auto Scaling Group (ASG) 来实现自动管理，即使用 ASG 运行 Web 服务器集群来代替单个 Web 服务器，如图 2-9 所示，ASG 可以完全自动地处理许多任务，包括启动 EC2 实例集群，监视每个实例的运行状况，替换故障实例，以及根据负载调整集群大小。

创建 ASG 的第一步是创建启动配置(*launch configuration*)，启动配置将定义如何设置 ASG 中的每个 EC2 实例。`aws_launch_configuration` 资源使用了与 `aws_instance` 资源几乎一模一样的两个参数，只是名称略有不同：`ami` 参数在这里被称为 `image_id`，`vpc_security_group_ids` 参数现在是 `security_groups`，因此从内容上看，你可以将后者完全替换为前者。

---

<sup>1</sup> 想要深入了解如何在 AWS 上构建高可用和可扩展的系统请参阅参考资料第 2 章[21]。

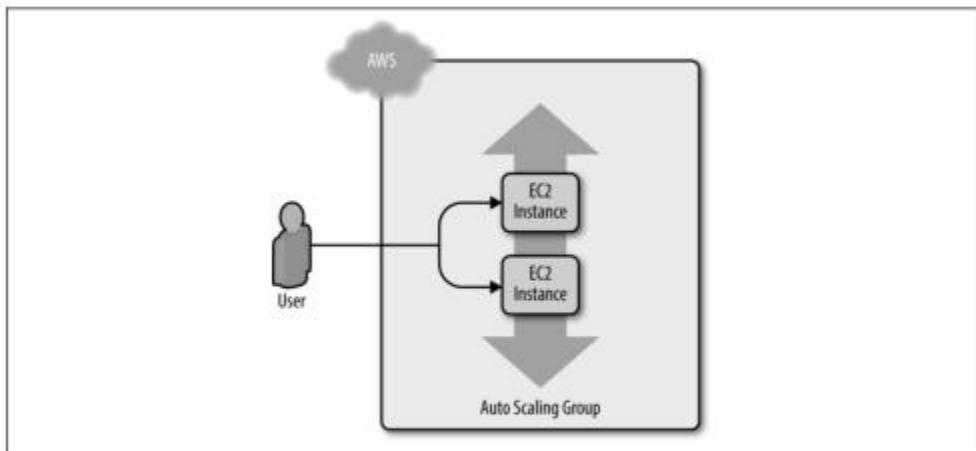


图 2-9：使用 ASG 运行 Web 服务器集群来代替单个 Web 服务器

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Hello, World" > index.html
    nohup busybox httpd -f -p ${var.server_port} &
  EOF
}
```

现在你可以通过 `aws_autoscaling_group` 资源来构建 ASG 了。

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name

  min_size = 2
  max_size = 10

  tag {
    key          = "Name"
    value        = "terraform-asg-example"
```

```
    propagate_at_launch      = true
}
}
```

新创建的 ASG 将包括 2 到 10 个 EC2 实例（初始化时的默认值为 2），每个 EC2 实例都拥有一个名称为 `terraform-asg-example` 的标签。请注意，ASG 的启动配置（`launch_configuration`）参数，被定义为一个资源引用。这会导致一个问题：因为启动配置是不可变的对象，所以如果更改启动配置中的任何参数，Terraform 将尝试替换它。在替换资源时，Terraform 会先删除旧资源，然后创建新资源，但是由于你的 ASG 正在引用旧资源，因此 Terraform 将无法删除启动配置。

可以通过使用生命周期设置来解决这个问题。每个 Terraform 资源都支持生命周期设置，这些生命周期设置用于定义如何创建、更新和删除该资源。一个特别有用的生命周期设置是 `create_before_destroy`。如果将 `create_before_destroy` 设置为 `true`，那么 Terraform 将反转其替换资源的顺序，首先创建替换资源（包括将指向旧资源的所有外部引用，更新为指向替换资源），然后删除旧资源。将 `lifecycle` 代码块添加到 `aws_launch_configuration` 资源定义中，如下所示。

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Hello, World" > index.html
    nohup busybox httpd -f -p ${var.server_port} &
  EOF

  # 当 ASG 中使用启动配置时，必须使用以下生命周期设置
  # https://www.terraform.io/docs/providers/aws/r/launch\_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

为了使 ASG 正常工作，还有另一个需要添加的参数：`subnet_ids`。这个参数定义 ASG 部

署 EC2 实例时的 VPC 子网信息（请参阅上文网络安全内容中有关子网的背景信息）。每个子网都位于相互隔离的 AWS AZ（即隔离的数据中心）中，因此，跨子网部署实例可以确保即使某些数据中心发生故障，服务也可以保持运行。直接使用静态子网列表会使维护或移植更加困难，一个更好的选择是使用数据源（*data source*）获取 AWS 账户中可用的子网列表。

一个数据源代表在每次运行 Terraform 时，从服务提供商（在本例中为 AWS）获取的只读信息。将数据源添加到 Terraform 配置中，并不会创建任何新内容。这只是提供一种手段，通过服务提供商的 API，读取数据并使该数据对于其余 Terraform 代码可见、可用。每个 Terraform 服务提供商都会公开各种数据源。例如，AWS 提供的数据源可以读取 VPC 数据、子网数据、AMI ID、IP 地址范围、当前用户身份等。

使用数据源的语法与使用资源的语法非常相似，如下所示。

```
data "<PROVIDER>_<TYPE>"<NAME> {
  [CONFIG ...]
}
```

在这里，**PROVIDER** 是提供商的名称（例如 `aws`），**TYPE** 是要使用的数据源的类型（例如 `vpc`），**NAME** 是可以在整个 Terraform 代码中引用该数据源的标识符，**CONFIG** 包含一个或多个针对该数据源的参数。例如，以下是如何使用 `aws_vpc` 数据源读取默认 VPC 信息的方式（请参考上文中的“关于默认虚拟私有云”的介绍）。

```
data "aws_vpc" "default" {
  default = true
}
```

对于数据源来说，传入的参数可以被理解为一种搜索过滤器，用于向数据源指示你要查找的信息。使用 `aws_vpc` 数据源时，通过将过滤器设置为 `default=true`，可以指示 Terraform 在 AWS 账户中查找默认的 VPC。

要从数据源中读取数据，请使用属性引用语法。

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

例如，要从 `aws_vpc` 数据源中获取 VPC 的 ID 信息，可以使用以下格式。

```
data.aws_vpc.default.id
```

你可以继续将其与 `aws_subnet_ids` 数据源组合在一起使用，以读取该 VPC 中的子网信息。

```
data "aws_subnet_ids" "default" {
    vpc_id = data.aws_vpc.default.id
}
```

最后，你可以从 `aws_subnet_ids` 数据源中获得子网号，赋值给 ASG 的 `vpc_zone_identifier` 参数。

```
resource "aws_autoscaling_group" "example" {
    launch_configuration = aws_launch_configuration.example.name
    vpc_zone_identifier   = data.aws_subnet_ids.default.ids

    min_size = 2
    max_size = 10

    tag {
        key          = "Name"
        value        = "terraform-asg-example"
        propagate_at_launch = true
    }
}
```

## 部署负载均衡器

学习到这里，你已经可以部署一个完整的 ASG 了。但还有一个小问题：因为现在需要管理多台服务器，每台服务器都有自己的 IP 地址，但通常你只想让最终用户使用一个 IP。解决这个问题的一种方法是使用负载均衡器在服务器之间分配流量，并将负载均衡器的 IP（实际上是 DNS 名称）提供给所有用户。创建一个具有高可用性和可伸缩性的负载均衡器需要做大量工作。不过通过使用 Amazon 的 *Elastic Load Balancer* (ELB) 服务可以让 AWS 为你处理这些工作，图 2-10 所示为使用 Amazon ELB 在 *Auto Scaling Group* 中分配流量。

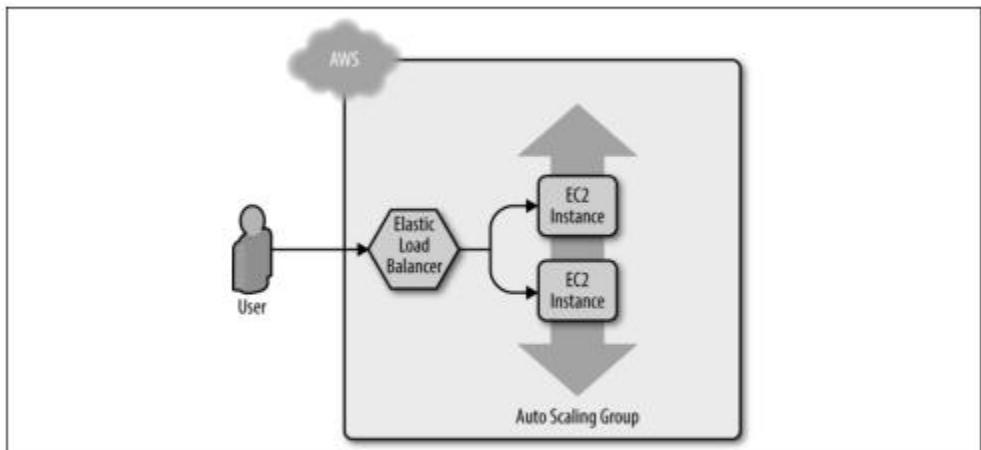


图 2-10：使用 Amazon ELB 在 Auto Scaling Group 中分配流量

AWS 提供了 3 种不同类型的负载均衡器。

#### 应用程序负载均衡器（ALB）

适合 HTTP 和 HTTPS 流量的负载均衡，工作在 OSI 模型的应用层（第 7 层）。

#### 网络负载均衡器（NLB）

适合 TCP、UDP 和 TLS 流量的负载均衡，可以比 ALB 更快地响应向上或向下扩展（NLB 每秒可以处理千万级请求），在 OSI 模型的传输层（第 4 层）上运行。

#### 经典负载均衡器（CLB）

这是早于 ALB 和 NLB 的“传统”负载均衡器，可以处理 HTTP、HTTPS、TCP 和 TLS 流量，但功能与 ALB 或 NLB 相比少了很多。同时在 OSI 模型的应用程序层（第 7 层）和传输层（第 4 层）上运行。

如今，大多数应用程序应该选择 ALB 或 NLB。由于在之前的简单的 Web 服务器示例中，使用的是一个 HTTP 应用程序，没有任何极端性能要求，因此 ALB 是最合适的。

图 2-11 所示为应用程序负载均衡器（ALB）概述，ALB 由以下几个部分组成。

## 侦听器 (listener)

侦听特定端口（如 80）和协议（如 HTTP）。

## 规则器 (listener rule)

接受进入侦听器的请求，并将符合特定路径（如 /foo 和 /bar）或特定主机名（如 foo.example.com 和 bar.example.com）的请求发送到特定目标组。

## 目标组 (target group)

目标组定义了一台或多台服务器，负责响应负载均衡器接收的请求。目标组还对这些服务器执行状况检查，并且只将请求发送到运行良好的节点。

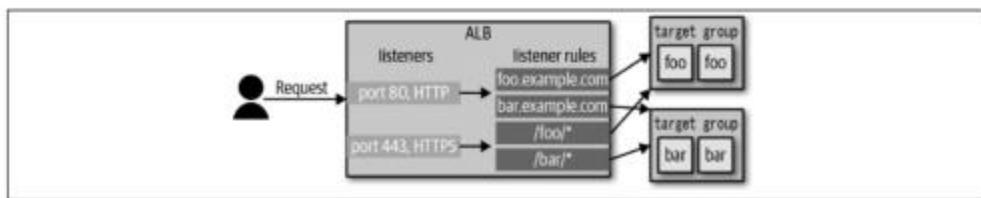


图 2-11：应用程序负载均衡器（ALB）概述

第一步是使用 `aws_lb` 资源创建 ALB 本身。

```
resource "aws_lb" "example" {
  name            = "terraform-asg-example"
  load_balancer_type = "application"
  subnets         = data.aws_subnet_ids.default.ids
}
```

`subnets` 参数通过引用 `aws_subnet_ids` 数据源的结果，将负载均衡器配置为可以使用默认 VPC 中的所有子网。<sup>1</sup> AWS 负载均衡器本身也不是单个服务器，而是由跨子网（跨数据中心）部署的多个服务器组成的。AWS 会根据流量自动缩放负载均衡器服务器的数量，并在单个服务器出现故障时转移流量到其他服务器，因此可以立即获得高扩展性和高可用性。

<sup>1</sup> 为了简化这些示例，我们在同一子网中运行 EC2 实例和 ALB。在生产环境中，你很可能会在不同的子网中运行它们。EC2 实例运行在私有子网中（因此不能从公共 Internet 直接访问它们），而 ALB 运行在公共子网中（因此用户可以直接访问它们）。

下一步是使用 `aws_lb_listener` 资源为这个 ALB 定义一个侦听器。

```
resource "aws_lb_listener" "http" {
    load_balancer_arn = aws_lb.example.arn
    port              = 80
    protocol          = "HTTP"

    # 默认返回一个 404 页面
    default_action {
        type = "fixed-response"

        fixed_response {
            content_type = "text/plain"
            message_body = "404: page not found"
            status_code = 404
        }
    }
}
```

侦听器将 ALB 配置为使用 HTTP 作为协议，侦听默认 80 端口。当请求不符合任何侦听规则时，默认响应将发送简单的 404 页面。

请注意，在默认情况下，所有 AWS 资源（包括 ALB）均不允许任何入站或出站流量，因此需要为 ALB 创建一个安全组，允许通过 80 端口进行入站请求，以便使用 HTTP 访问负载均衡器，并允许在所有端口上进行出站请求，以便负载均衡器执行运行健康状况检查。

```
resource "aws_security_group" "alb" {
    name           = "terraform-example-alb"

    # 允许入站的 HTTP 请求
    ingress {
        from_port   = 80
        to_port     = 80
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }

    # 允许所有出站请求
    egress {
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
    }
}
```

```
    cidr_blocks  = ["0.0.0.0/0"]
}
}
```

你需要通过 `security_groups` 参数配置 `aws_lb` 资源来使用这个安全组。

```
resource "aws_lb" "example" {
  name          = "terraform-asg-example"
  load_balancer_type = "application"
  subnets       = data.aws_subnet_ids.default.ids
  security_groups = [aws_security_group.alb.id]
}
```

接下来需要使用 `aws_lb_target_group` 资源为 ASG 创建目标组。

```
resource "aws_lb_target_group" "asg" {
  name          = "terraform-asg-example"
  port          = var.server_port
  protocol      = "HTTP"
  vpc_id        = data.aws_vpc.default.id

  health_check {
    path      = "/"
    protocol = "HTTP"
    matcher   = "200"
    interval  = 15
    timeout   = 3
    healthy_threshold = 2
    unhealthy_threshold = 2
  }
}
```

目标组将通过定期向每个实例发送 HTTP 请求，进行健康检查，并且仅当实例返回与配置的 `matcher` 值匹配的响应时，才会被认为是“健康”的（例如，你可以配置 `matcher` 来查找响应中的 200 OK 字段）。如果某个实例由于停机或过载而无法返回匹配的响应，那么它将被标记为“运行状况不佳”，目标组将自动停止向其发送流量，以避免对用户产生影响。

目标组如何知道应该向哪个 EC2 实例发送请求呢？你可以通过 `aws_lb_target_group_attachment` 资源，将一组 EC2 实例的静态列表提供给目标组使用。但是由于使用了 ASG，实例可以随时启动或终止，所以静态列表在这种情况下并不适用。相反，你可以利用 ASG 和 ALB 之间的一类集成，将 `aws_autoscaling_group` 资源的 `target_group_arns` 参数指向新创建的目标组。

```

resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier   = data.aws_subnet_ids.default.ids

  target_group_arns     = [aws_lb_target_group.asg.arn]
  health_check_type     = "ELB"

  min_size = 2
  max_size = 10

  tag {
    key          = "Name"
    value        = "terraform-asg-example"
    propagate_at_launch = true
  }
}

```

你还应该将 `health_check_type` 参数从默认的“EC2”更新为“ELB”。因为默认的“EC2”类型只会进行最基本的运行健康状况检查，只有当 AWS 虚拟机管理程序指示虚拟机停机或无法访问时，才认为实例异常。而“ELB”类型的健康检查功能更强大，它指示 ASG 通过目标组的健康检查功能确定一个实例是否正常工作，如果目标组报告虚拟机状态异常，虚拟机也会被自动替换。这样，不仅涵盖了实例停机的情况，而且还能识别由于内存用完或关键进程崩溃导致的响应异常。

最后，通过使用 `aws_lb_listener_rule` 资源，创建侦听器规则，将所有这些部分联系在一起，如下所示。

```

resource "aws_lb_listener_rule" "asg" {
  listener_arn      = aws_lb_listener.http.arn
  priority         = 100

  condition {
    field          = "path-pattern"
    values         = ["*"]
  }

  action {
    type           = "forward"
    target_group_arn= aws_lb_target_group.asg.arn
  }
}

```

上面的代码将根据侦听器规则，将符合路径匹配的请求发送到包含 ASG 的目标组。

在部署负载均衡器之前，需要将之前单个 EC2 实例中的 `public_ip` 输出变量替换为 ALB 的 DNS 名称。

```
output "alb_dns_name" {
    value      = aws_lb.example.dns_name
    description = "The domain name of the load balancer"
}
```

当运行 `terraform apply` 命令时，可以从输出中看到，Terraform 将删除原来的单台 EC2 实例，并创建新的启动配置、ASG、ALB 和安全组。如果计划输出看起来没有问题，则输入 `yes`，然后按 Enter 键。当 `apply` 命令完成后，应该看到 `alb_dns_name` 的输出，如下所示。

```
Output:
alb_dns_name = terraform-asg-example-123.us-east-2.elb.amazonaws.com
```

首先保存这个链接。实例还需要几分钟时间才能完成启动，并在 ALB 中显示为正常状态。与此同时，你可以检查已经部署的内容。打开 EC2 控制台 ASG 部分（见参考资料第 2 章[22]）应该看到已经创建了 Auto Scaling Group（ASG），如图 2-12 所示。

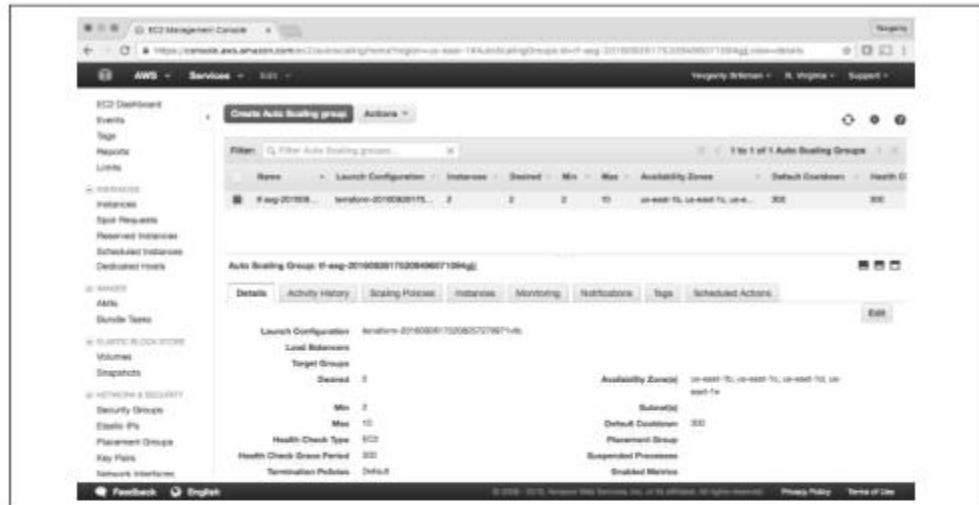


图 2-12：Auto Scaling Group

如果切换到“Instances”选项卡，会看到ASG中的两个EC2实例正在启动，如图2-13所示。

The screenshot shows the AWS EC2 Management Console interface. On the left, there's a navigation sidebar with various services like EC2 Dashboard, Events, Logs, Reports, Limits, Instances, Start Requests, Reserved Instances, Scheduled Instances, Dedicated Hosts, Volumes, Snapshots, Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, and CloudWatch Metrics. The 'Instances' section is currently selected. In the main content area, there's a 'Launch Instances' button and an 'Actions' dropdown. Below that is a search bar and a table titled 'Instances'. The table lists three instances:

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm State
terraform-asg-example	i-04a4c29e	t2.micro	us-east-1a	running	2/2 checks	None
terraform-asg-example	i-0d8f6f6e	t2.micro	us-east-1b	running	2/2 checks	None
terraform-example	i-0d8f6f6e	t2.micro	us-east-1b	running	2/2 checks	None

Below the table, it says 'Instances: [ i-0d8f6f6e (terraform-example), i-04a4c29e (terraform-example) ]'. There are tabs for 'Description', 'Status Checks', 'Monitoring', and 'Logs'. At the bottom of the page, there are links for 'Feedback', 'English', and 'Privacy Policy | Terms of Use'.

图2-13：ASG中的两个EC2实例正在启动

单击“Load Balancers”选项卡，则会看到应用程序负载均衡器，如图2-14所示。

The screenshot shows the AWS EC2 Management Console interface. The left sidebar includes the same navigation items as in Figure 2-13, plus 'Load Balancers' under the 'CloudWatch Metrics' section. The 'Load Balancers' section is currently selected. In the main content area, there's a 'Create Load Balancer' button and an 'Actions' dropdown. Below that is a search bar and a table titled 'Load Balancers'. The table lists one load balancer:

Name	DNS name	Status	VPC ID	Availability Zones	Type
terraform-asg-example	terraform-asg-example-111955677.us-east-1.ambda.amazonaws.com	active	vpc0f600	us-east-1a, us-east-1b	ELB

Below the table, it says 'Load Balancer: terraform-asg-example'. There are tabs for 'Description', 'Instances', 'Health Check', 'Listeners', 'Monitoring', and 'Logs'. Under 'Basic Configuration', there are fields for 'Name' (terraform-asg-example), 'DNS name' (terraform-asg-example-111955677.us-east-1.ambda.amazonaws.com), 'Created time' (September 26, 2018 at 11:08:39 PM UTC+1), 'Hosted zone' (2NSGQCTTQZ74), 'Status' (2 of 2 instances in service), 'Scheme' (internet-facing), and 'VPC' (vpc-0f600). There's also an 'Availability Zones' field with 'subnet-0f780b71 - us-east-1c'. At the bottom of the page, there are links for 'Feedback', 'English', and 'Privacy Policy | Terms of Use'.

图2-14：应用程序负载均衡器

最后，如果单击“Target Group”选项卡，则可以看到目标组，如图 2-15 所示。

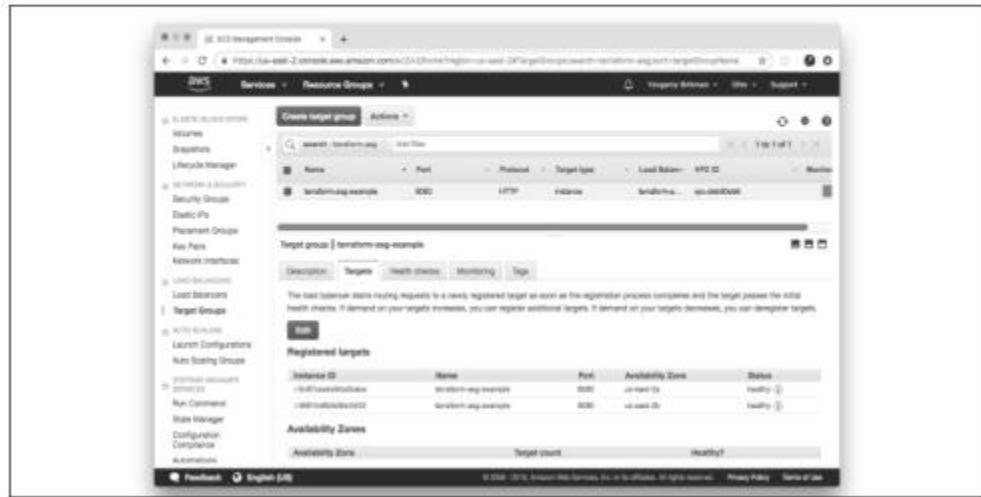


图 2-15：目标组

如果单击目标组，然后在屏幕的下半部分找到“Targets”选项卡，则可以看到实例已经在目标组中注册，正在进行健康状况检查。通常一到两分钟后，状态指示将转为“健康”。现在让我们通过 curl 命令，测试之前保存的 `alb_dns_name` 的输出值，运行如下。

```
$ curl http://<alb_dns_name>
Hello,World
```

ALB 成功地将流量路由到新创建的 EC2 实例。每次访问这个 URL 时，它会选择一个不同的实例来处理请求。现在，一个 Web 服务器集群可以正常运行了！

此时可以通过以下方式观察集群如何响应启动新实例或关闭旧实例的需求。例如，转到“Instances”选项卡并终止一个实例，方法是选中其复选框，单击顶部的“Actions”按钮，然后将实例状态设置为终止。继续测试 ALB 的 URL，即使终止了选中的实例，每个请求还是继续获得 200 OK 的响应，这是因为 ALB 将自动检测到该实例已被关闭，并将路由请求分流到其他实例。更有趣的是，在实例被关闭后不久，ASG 将检测到当前少于两个运行实例，因此会自动启动一个新实例来代替被关闭的实例（自我修复）。你还可以通过

调整 `desired_capacity` 参数大小，来查看 ASG 如何调整后端实例的数目，修改 Terraform 代码中的参数，并重新运行 `apply` 命令。

## 清理工作

当你完成本章或后面章节的 Terraform 实验程序后，最好删除所有已经创建的资源，这将确保 AWS 不会收取额外费用。由于 Terraform 跟踪已经创建的资源，因此清理工作很简单。只需要运行 `destroy` 命令。

```
$ terraform destroy
(...)
Terraform will perform the following actions:

# aws_autoscaling_group.example 将要被销毁
- resource "aws_autoscaling_group" "example" {
  (...)

}

# aws_launch_configuration.example 将要被销毁
- resource "aws_launch_configuration" "example" {
  (...)

}

# aws_lb.example 将要被销毁
- resource "aws_lb" "example" {
  (...)

}

(...)

Plan: 0 to add, 0 to change, 8 to destroy.

Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value:
```

需要注意的是，你应该减少在生产环境中运行 `destroy` 命令的机会！因为销毁命令是无法撤销的操作，在这里 Terraform 提供了最后的机会来检查将要进行操作的内容，显示将要被删除的所有资源的列表，并要求确认删除命令。如果一切正常，请输入 `yes`，并按 Enter 键。接下来，Terraform 将首先构建依赖关系图，以正确的顺序删除所有资源，同时使用尽可能多的并行处理机制。在一两分钟内，你的 AWS 账户将被清理干净。

请注意，在本书的后面，你将继续开发此示例，所以请不要删除 Terraform 代码！但是，可以随时随地 `destroy` 实际部署的资源。毕竟，将基础架构作为代码进行管理的好处是，所有与这些资源有关的信息都被记录在代码中，可以随时使用 `terraform apply` 命令重新创建所有资源。实际上，你最好将最新更改提交到 Git 代码库，以达到跟踪基础设施历史信息的目的。

## 小结

现在，你已经了解了如何使用 Terraform 的基本功能。声明性语言可以轻松准确地描述要创建的基础设施。使用 `plan` 命令，可以在部署更改之前验证更改内容并捕获错误。变量、引用和依赖项可以减少代码中的冗余内容，并使代码具有高度的可配置性。

本章介绍的只是最基本的功能。在第 3 章中，你将学习 Terraform 如何跟踪已经创建的基础设施，以及 Terraform 代码结构对应用的深刻影响。在第 4 章中，你将学习如何使用 Terraform 模块创建可重用的基础设施。

# 如何管理 Terraform 的状态

在第 2 章中，你可能已经注意到，当运行 `terraform plan` 或 `terraform apply` 命令来创建和更新资源时，每次 Terraform 总能找到之前它自己创建的资源，并且按照以前的状态来更新。Terraform 是如何知道应该管理哪些资源呢？你在 AWS 账户中拥有各种基础设施，并使用多种机制（有的手动、有的通过 Terraform 命令、有的通过 CLI 命令行）进行部署，Terraform 应该对哪些基础设施负责呢？

在本章中，你将了解 Terraform 项目如何跟踪基础设施的状态，及其对文件布局、隔离和锁定的影响。下面是将要讨论的关键主题。

- 什么是 Terraform 的状态
- 共享存储状态文件
- Terraform 后端的局限性
- 隔离状态文件
  - 通过工作区进行隔离
  - 通过文件布局进行隔离
- `terraform_remote_state` 数据源

## 什么是 Terraform 的状态

每次运行 Terraform 命令，它都会把创建的基础设施的信息记录在 *Terraform* 状态文件中。默认情况下，当你在 `/foo/bar` 文件夹中运行 Terraform 命令时，Terraform 将创建状态文件

*/foo/bar/terraform.tfstate*。Terraform 状态文件使用自定义 JSON 格式，记录了资源配置文件中的定义与现实世界中的部署的对应关系。例如，假设 Terraform 配置包含以下内容。

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

运行 `terraform apply` 命令之后，以下是生成的 `terraform.tfstate` 文件中的一段内容（截取了部分片段以提高可读性）。

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Terraform 使用这种 JSON 格式，定义了一个类型是 `aws_instance`、名称为 `example` 的资

源，对应的是你的 AWS 账户中 ID 为 `i-00d689a0acc43af0f` 的 EC2 实例。每次运行 Terraform 命令时，它都会从 AWS 上获取这个 EC2 实例的最新状态，并将其与 Terraform 配置文件中的内容进行比较，以确定需要进行哪些变更。换句话说，`plan` 命令的输出是：计算机上存储的代码与现实世界中部署的基础设施之间的区别，这些区别是通过状态文件中的 ID 来发现的。



### 状态文件是专用 API

状态文件格式属于私有 API，每个发行版都会进行更改，仅供 Terraform 内部访问使用。需要注意的是，用户应该避免手动编辑 Terraform 状态文件或通过代码直接读取文件内容。如果由于某种原因需要改写状态文件（这种情况很少见），请使用 `terraform import` 或 `terraform state` 命令（你将在第 5 章中看到关于这两个命令的示例）。

如果仅仅将 Terraform 用于个人项目，使用本地计算机保存 `terraform.tfstate` 文件就足够了。但是，如果要在一个实际产品项目的团队环境下使用 Terraform，这种做法会遇到以下几个问题。

#### 状态文件的共享存储

为了能够使用 Terraform 去更改基础设施，每个团队成员都需要访问相同的 Terraform 状态文件。这意味着你需要将这些文件存储在共享位置。

#### 锁定状态文件

一旦共享了状态文件，又会遇到一个新问题：如何锁定。如果没有锁定机制，那么当两个团队成员同时运行 Terraform 时会发生竞争状况，而若多个 Terraform 进程对状态文件进行并发更新，则会导致写入冲突、数据丢失和状态文件损坏。

#### 隔离状态文件

在对基础设施进行更改时，最好的做法是对不同环境进行隔离。例如，在测试或预发布环境中进行更改时，你要确保不会意外干扰生产环境。但是，如果所有基础设施都定义在同一个 Terraform 状态文件中，那么如何隔离所做的更改呢？

在以下各节中，我将深入探讨这些问题，并向你展示相应的解决方案。

# 共享存储状态文件

允许多个团队成员共享访问的最常规做法是将文件存储在版本控制系统（如 Git）中。虽然将 Terraform 代码存储在版本控制系统中是十分必要的，但是将 Terraform 状态文件存储在版本控制系统中却不是一个好主意，原因如下。

## 手动错误

如果在运行 Terraform 命令之前，忘记从版本控制中读取最新状态文件，或在运行 Terraform 之后，忘记将你的状态文件推送到版本控制系统，那么，团队成员在运行 Terraform 代码时就会使用过期的状态文件，直接的后果是，系统意外地回到之前状态或重复了以前的部署。

## 锁定

无法处理锁定机制，而锁定机制可以避免团队成员对同一个状态文件同时运行 `terraform apply` 命令。

## 机密

Terraform 状态文件属于纯文本文件。与之相关的问题是，Terraform 可能会将与资源相关的敏感数据写入文件。例如，如果使用 `aws_db_instance` 资源创建数据库，Terraform 会把用于数据库访问的用户名和密码以纯文本格式存储在状态文件中。将纯文本格式的机密信息存储在任何地方都不是一个好主意，包括版本控制系统。截至 2019 年 5 月，这仍是 Terraform 一个未解决的问题（见参考资料第 3 章[1]）。Terraform 社区对这个问题，已经有了一些合理的解决方法，我将在后面进行讨论。

比使用版本控制存储状态文件更好的方式是，通过 Terraform 对远程后端的内置支持，管理状态文件的共享存储。Terraform *backend* 定义 Terraform 将如何加载和存储状态文件。在前几章的示例中一直使用的默认后端是将状态文件存储在本地磁盘上的本地后端（*local backend*）。而另一种远程后端（*remote backend*）允许你在远程共享存储系统中保存状态文件。Terraform 支持许多种远程后端，包括 Amazon S3（Simple Storage Service，简单存储服务）、Azure 存储、Google 云存储，以及 HashiCorp 自营的 Terraform Cloud。

Terraform 专业版和 Terraform 企业版。

远程后端成功地解决了刚才列出的 3 个问题，具体如下。

#### 手动错误

配置为远程后端后，在每次运行 `plan` 或 `apply` 命令时，Terraform 都会自动从该后端加载状态文件，并且在每次执行完成后，自动更新状态文件。因此不会出现手动错误。

#### 锁定

大多数远程后端本身都支持锁定功能。当运行 `terraform apply` 命令时，Terraform 将自动获取一个锁；如果其他人已经在运行 `apply` 命令，那么他们拥有该锁，你必须等待其他人完成操作。你可以在运行 `apply` 命令时加上 `-lock-timeout=<TIME>` 参数，指示 Terraform 需要等待多长时间。（例如，`-lock-timeout=10m` 将等待 10min）。

#### 机密

大多数远程后端本身就支持传输加密和状态文件存储加密。此外，这些后端通常会提供接口，用来配置访问权限（例如，将 IAM 策略与 Amazon S3 bucket 配合使用），因此你可以控制谁有权访问你的状态文件，以及可能包含的机密信息。当然如果 Terraform 本身能够支持状态文件加密是最好的方案，但这些远程后端已经大大减少了安全方面的问题，至少状态文件不是存储在磁盘上任意位置的纯文本文件了。

如果你将 Terraform 与 AWS 结合使用，那么 Amazon 的托管文件存储服务 Amazon S3 通常是远程后端的最佳选择，原因如下。

- 这是一项托管服务，因此你不需要部署和管理额外的基础设施。
- 它的设计具有 99.999999999% 的耐用性和 99.99% 的可用性，这意味着你不必担心数据丢失或服务中断。<sup>1</sup>
- 它支持加密，这减少了将敏感数据写入状态文件的顾虑。这虽然不是完美的解决方案，因为团队中有权访问该 S3 bucket 的任何人都可以以未加密的形式查看状态文件；

---

<sup>1</sup> 了解更多关于 S3 担保的信息请参阅参考资料第 3 章[2]。

但是至少数据在静态存储状态中（Amazon S3 使用 AES-256 支持服务器端加密）和传输状态中（Terraform 使用 SSL 加密数据在 Amazon S3 中的读取和写入）处于加密状态。

- 它支持通过 DynamoDB 来锁定（稍后将对此进行详细介绍）。
- 它支持版本控制，因此状态文件的每个修订版本都会被记录，如果出现问题，你可以恢复到旧版本。
- 价格便宜，大多数 Terraform 使用场景很容易通过 AWS 免费套餐实现。<sup>1</sup>

要通过 Amazon S3 远程存储 Terraform 状态文件，第一步是创建 S3 bucket。在新的本地文件夹中创建一个 *main.tf* 文件（应该使用与第 2 章示例不同的文件夹），并在文件顶部将 AWS 定义为服务提供商。

```
provider "aws" {
    region = "us-east-2"
}
```

接下来，使用 `aws_s3_bucket` 资源创建一个 S3 bucket。

```
resource "aws_s3_bucket" "terraform_state"{
    bucket = "terraform-up-and-running-state"

    # 防止意外删除此 S3 bucket
    lifecycle {
        prevent_destroy = true
    }

    # 启用版本控制，以便查看状态文件的所有历史记录
    versioning {
        enabled = true
    }

    # 默认情况下启用服务器端加密
    server_side_encryption_configuration {
        rule {
            apply_server_side_encryption_by_default {
                sse_algorithm = "AES256"
            }
        }
    }
}
```

---

<sup>1</sup> S3 价格信息请参阅参考资料第 3 章[3]。

```
        }
    }
}
}
```

该代码设置了 4 个参数。

#### bucket

这是 S3 bucket 的名称。请注意，S3 bucket 名称在所有 AWS 客户之间必须是全局唯一的。因此，你需要将 `bucket` 参数从 `terraform-up-and-running-state`（我已经创建）更改为你自己的名称。<sup>1</sup>请记住该名称和正在使用的 AWS 区域，因为稍后你将再次需要这两条信息。

#### prevent\_destroy

`prevent_destroy` 是本书第 2 次出现关于生命周期的设定（第 1 次是在第 2 章中的 `crate_before_destroy`）。当你在某个资源上将 `prevent_destroy` 设置为 `true` 时，任何删除该资源的尝试（例如，运行 `terraform destroy` 命令）都会导致 Terraform 退出并出现错误。通过这种方法，可以防止意外删除重要资源（在这个例子中是存储了所有 Terraform 状态文件的 S3 bucket）。当然，如果你确实要删除它，注释掉该行设置即可。

#### versioning

此代码块将启用 S3 bucket 上的版本控制功能，每次更新 S3 bucket 中的文件都会创建该文件的一个新版本。这样，你可以查看文件的旧版本，并随时可以将文件还原为这些旧版本。

#### server\_side\_encryption\_configuration

默认情况下，此代码块设置了 S3 bucket 的服务器端的数据加密。确保状态文件及其中可能包含的所有机密，以加密状态存储在 S3 中的磁盘上。

---

<sup>1</sup> 参考资料第 3 章[4]提供了有关 S3 bucket 名称的更多信息。

接下来，你需要创建一个 DynamoDB 表用于实现锁定功能。DynamoDB 是 Amazon 的分布式键值存储服务。它支持一致性读取和条件写入，这是你构建一个分布式锁定系统所需要的全部功能模块。而且它同样是托管服务，不需要维护额外的基础设施，价格便宜，大多数 Terraform 使用场景很容易通过 AWS 免费套餐实现。<sup>1</sup>

要使用 DynamoDB 实现 Terraform 状态文件的锁定功能，必须创建一个具有主键名为 LockID（确切的拼写和大小写）的 DynamoDB 表。你可以使用 `aws_dynamodb_table` 资源来创建表。

```
resource "aws_dynamodb_table" "terraform_locks" {
    name      = "terraform-up-and-running-locks"
    billing_mode= "PAY_PER_REQUEST"
    hash_key   = "LockID"

    attribute {
        name = "LockID"
        type = "S"
    }
}
```

运行 `terraform init` 命令下载服务提供商代码，然后运行 `terraform apply` 命令进行部署。（注意：要部署此代码，你使用的 IAM 用户需要具有创建 S3 bucket 和 DynamoDB 表的权限，如第 2 章“设置 AWS 账户”中所介绍的。）部署所有内容之后，你将拥有一个 S3 bucket 和 DynamoDB 表。此时你的 Terraform 状态文件仍存储在本地。你需要在 Terraform 代码中添加一个 `backend` 配置，将状态存储在 S3 bucket 中（具有加密和锁定功能）。这是关于 Terraform 自身的配置，因此它位于 `terraform` 代码块中，语法如下所示。

```
terraform {
    backend "<BACKEND_NAME>" {
        [CONFIG...]
    }
}
```

其中 `BACKEND_NAME` 是你要使用的后端类型的名称（如 `s3`），而 `CONFIG` 是一个或多个特定于该后端的参数（例如，所使用的 S3 bucket 名字）。一个完整的 S3 bucket `backend` 配置

<sup>1</sup> 这是 DynamoDB 的价格信息（见参考资料第 3 章[5]）。

如下所示。

```
terraform {
  backend "s3" {
    # 请用你的 bucket 名称替换
    bucket      = "terraform-up-and-running-state"
    key         = "global/s3/terraform.tfstate"
    region      = "us-east-2"

    # 请用你的 DynamoDB 表名称替换
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt       = true
  }
}
```

让我们对设置逐一进行介绍。

#### bucket

S3 bucket 的名称。请将其替换为你先前创建的 S3 bucket 的名称。

#### key

S3 bucket 中 Terraform 状态文件写入的文件路径。你将在稍后的章节中了解将代码目录结构设置为 `global/s3/terraform.tfstate` 的原因。

#### region

S3 bucket 所在的 AWS 区域。要确保将其替换为之前创建的 S3 bucket 所在区域。

#### dynamodb\_table

用于锁定的 DynamoDB 表。请将其替换为之前创建的 DynamoDB 表的名称。

#### encrypt

将其设置为 `true`，Terraform 状态文件将以加密格式存储在 S3 的磁盘上。我们已经在 S3 bucket 中启用了默认加密，因此这是确保数据始终被加密的双保险。

再次运行 `terraform init` 命令，指示 Terraform 将状态文件存储在此 S3 bucket。此命令不仅可以下载提供商的代码，还可以配置 Terraform 后端（稍后你还将看到另一种用法）。

而且，`init` 命令是幂等的，因此可以被多次重复运行。

```
$ terraform init

Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
Pre-existing state was found while migrating the previous "local" backend
to the newly configured "s3" backend. No existing state was found in the
newly configured "s3" backend. Do you want to copy this state to the new
"s3" backend? Enter "yes" to copy and "no" to start with an empty state.

Enter a value:
```

Terraform 自动检测到你在本地已经存在一个状态文件，并提示你是否将其复制到新的 S3 后端。如果输入 `yes`，则应该看到以下内容。

```
Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

运行此命令后，你的 Terraform 状态文件被保存在 S3 bucket 中。你可以通过浏览器访问 S3 管理控制台（见参考资料第 3 章[6]）并单击 bucket 来查看。你应该看到类似于图 3-1 所示的存储在 S3 中的 Terraform 状态文件。



图 3-1：存储在 S3 中的 Terraform 状态文件

启用此后端后，Terraform 将在运行命令之前自动从 S3 bucket 中读取最新状态；并在运行命令后自动将最新状态更新到 S3 bucket 中。要检验实际效果，请添加以下输出变量。

```
output "s3_bucket_arn" {
  value      = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}

output "dynamodb_table_name" {
  value      = aws_dynamodb_table.terraform_locks.name
  description = "The name of the DynamoDB table"
}
```

这些变量将打印出 S3 bucket 的 Amazon 资源名称（ARN）和 DynamoDB 表的名称。运行 `terraform apply` 命令进行查看。

```
$ terraform apply

Acquiring state lock. This may take a few moments...

aws_dynamodb_table.terraform_locks: Refreshing state...
aws_s3_bucket.terraform_state: Refreshing state...

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Releasing state lock. This may take a few moments...

Outputs:

dynamodb_table_name = terraform-up-and-running-locks
s3_bucket_arn = arn:aws:s3:::terraform-up-and-running-state
```

（请注意 Terraform 如何在运行 `apply` 命令之前获取锁，并在运行之后释放锁！）

现在，转到 S3 控制台（见参考资料第 3 章[7]）再次刷新页面，然后单击“version”旁边的灰色显示按钮。现在，你应该在 S3 bucket 中看到 `terraform.tfstate` 文件的多个版本，如图 3-2 所示。



图 3-2: S3 中 Terraform 状态文件的多个版本

这意味着 Terraform 会自动向 S3 推送和读取状态数据文件，并且 S3 保存了状态文件的每个修订版，这对于调试和恢复到旧版本是十分有用的。

Terraform 后端的局限性

现在让我们了解一下，Terraform 的后端的局限性和陷阱。第一个局限性是：如何使用 Terraform 创建一个用于存储 Terraform 状态文件的 S3 bucket？这是一个“先有鸡还是先有蛋”的问题。为此，你必须使用两个步骤。

1. 编写 Terraform 代码以创建 S3 bucket 和 DynamoDB 表，使用本地后端来部署。
  2. 返回 Terraform 代码，向其中添加远程 backend 的配置，使用新创建的 S3 bucket 和 DynamoDB 表，然后运行 `terraform init` 命令将本地状态文件复制到 S3。

当你想删除 S3 bucket 和 DynamoDB 表时，必须执行相反的两个步骤。

1. 转到 Terraform 代码，删除 backend 配置，然后重新运行 `terraform init` 命令将 Terraform 状态文件复制回本地磁盘。
  2. 运行 `terraform destroy` 命令删除 S3 bucket 和 DynamoDB 表。

这样使用两个步骤多少有点烦琐，但好消息是，你的所有 Terraform 代码可以共享同一个 S3 bucket 和 DynamoDB 表，所以你只需要执行一次上述步骤即可（如果你有多个账户，则每个 AWS 账户需要执行一次）。待 S3 bucket 生成后，在其余的 Terraform 代码中，你就可以直接使用 `backend` 开始配置了，无须重复任何其他步骤。

第二个局限性更加痛苦：Terraform 的 `backend` 代码块中不允许使用任何变量或引用。以下代码将不起作用。

```
# 这将不起作用，后端配置中不允许使用变量
terraform {
  backend "s3" {
    bucket      = var.bucket
    region      = var.region
    dynamodb_table = var.dynamodb_table
    key         = "example/terraform.tfstate"
    encrypt     = true
  }
}
```

这意味着你需要手动复制并粘贴 S3 bucket 名称、地区、DynamoDB 表名到你的每一个 Terraform 模块中（你将在第 4 章和第 6 章中学习有关 Terraform 模块的内容，现在你只需要知道，模块是组织和重用 Terraform 代码的重要方式，在现实世界中 Terraform 代码通常是由许多小模块组成的）。更糟糕的是，你必须非常小心，不要复制/粘贴 `key` 值，要确保你部署的每一个 Terraform 模块都拥有唯一的 `key` 值，以免意外覆盖其他模块的状态文件！在跨多个环境部署和管理大量 Terraform 模块的情况下，大量的手动修改和复制/粘贴工作很容易出错。

截至 2019 年 5 月，唯一可用的解决方案是通过部分配置(*partial configuration*)在 Terraform 代码中省略某些 `backend` 参数的静态设置，而且在调用 `terraform init` 命令时，通过 `backend-config` 命令行传递参数。例如，你可以将重复的 `backend` 参数，如 `bucket` 和 `region`，提取到一个单独的文件 `backend.hcl`。

```
# backend.hcl
bucket      = "terraform-up-and-running-state"
region      = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
```

```
encrypt      = true

# 部分配置项。其他设置（例如 bucket、region 等参数）将在运行 terraform init 命令时
# 通过 -backend-config 参数从文件中传入
terraform {
  backend "s3" {
    key      = "example/terraform.tfstate"
  }
}
```

要在 Terraform 代码中，只继续保留了 `key` 参数，因为不同的模块需要设置不同的 `key` 值。

```
$ terraform init -backend-config = backend.hcl
```

Terraform 将合并 `backend.hcl` 中的部分配置与 Terraform 代码中的部分配置，生成可供模块使用的完整配置项集合。

另一个解决方案是使用开源工具 Terragrunt，Terragrunt 试图填补 Terraform 功能上的一些空白，遵循 DRY（不要重复自己）原则，可以将 `backend` 中所有基本后端设置（bucket 名称、区域、DynamoDB 表名）定义在一个文件中，并将 `key` 参数自动设置为模块的相对路径。你将在第 8 章中看到有关如何使用 Terragrunt 的示例。

## 隔离状态文件

通过远程后端和锁定机制，协作不再是问题。但是，另一个问题仍然存在：隔离。首次使用 Terrafrom 时，你可能会想在一个 Terraform 文件或一个文件夹下的一组 Terraform 文件中，定义所有基础设施。这种方法的弊端在于，所有 Terraform 模块的状态都存储在同一个文件中，任何单一的错误都可能破坏所有内容。

例如，尝试在预发布环境中部署应用程序的新版本时，你可能会破坏生产环境中的应用程序。更糟糕的是，由于未使用锁定机制或罕见的 Terraform 自身故障，你可能破坏整个状态文件，直接导致所有环境中的全部基础设施都被破坏。<sup>1</sup>

---

<sup>1</sup> 这是一个关于当你不对 Terraform 状态进行隔离时，会发生什么事情的生动示例（见参考资料第 3 章[8]）。

在开发中使用独立环境的意义在于它们彼此之间的隔离性。当通过同一个 Terraform 配置文件，管理所有环境时，这种隔离性已经被打破。就好比一艘船，为了防止一小部分的漏水殃及整个船舱，需要使用舱与舱之间的隔板充当屏障。你也应该在 Terraform 设计中设置“舱壁隔板”，如图 3-3 所示。

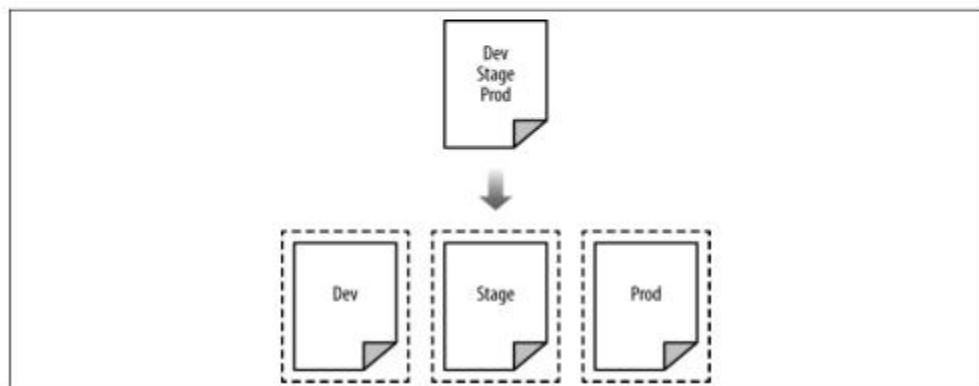


图 3-3：Terraform 使用“舱壁隔板”设计

如图 3-3 所示，与其将所有环境定义在一组 Terraform 配置文件中（顶部），不如针对每一个环境生成一个单独的配置集（底部），这样当一个环境出现问题时，将完全与其他环境隔离开。这里有两种隔离状态文件的方法。

#### 通过工作区（*workspace*）隔离

工作区隔离法适用于对相同的配置进行快速隔离的测试。

#### 通过文件布局隔离

文件布局隔离法适用于需要严格隔离的生产环境用例。

让我们在接下来的部分对两种方法逐一进行介绍。

#### 通过工作区进行隔离

*Terraform* 工作区允许你将 *Terraform* 状态文件存储在多个单独的命名工作区。*Terraform* 从单一名为 `default` 的默认工作区开始，如果你从未明确指定一个工作区，则默认工作区

就是你将一直使用的工作区。使用 `terraform workspace` 命令，可以创建新的工作区或在工作区之间切换。让我们通过部署单个 EC2 实例的代码，来实验 Terraform 工作区的使用。

```
resource "aws_instance" "example" {
  ami = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

使用你在本章前面创建的 S3 bucket 和 DynamoDB 表作为此实例的后端配置，但要将 `key` 设置为 `workspaces-example/terraform.tfstate`。

```
terraform {
  backend "s3" {

    # 替换为你的 bucket 名称
    bucket      = "terraform-up-and-running-state"
    key         = "workspaces-example/terraform.tfstate"
    region      = "us-east-2"

    # 用你的 DynamoDB 表名称替换
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt       = true
  }
}
```

运行 `terraform init` 和 `terraform apply` 命令来部署此代码。

```
$ terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
(...)

Terraform has been successfully initialized!

$ terraform apply
(...)
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

此部署的状态文件存储在默认工作区中。你可以通过运行 `terraform workspace show` 命令来标识当前所使用的工作区。

```
$ terraform workspace show  
default
```

在默认工作区中，状态存储在 `key` 参数指定的确切位置。图 3-4 所示为默认工作区中 S3 bucket 的存储状态，通过查看 S3 bucket，可以看到 `workspaces-example` 文件夹中存在一个 `terraform.tfstate` 文件。

让我们使用 `terraform workspace new` 命令，创建一个名为 `example1` 的新工作区。

```
$ terraform workspace new example1  
Created and switched to workspace "example1"  
  
You're now on a new, empty workspace. Workspaces isolate their state,  
so if you run "terraform plan" Terraform will not see any existing state  
for this configuration.
```

现在请注意，如果运行 `terraform plan` 命令会发生什么情况。

```
$ terraform plan  
  
Terraform will perform the following actions:  
  
# aws_instance.example will be created  
+ resource "aws_instance" "example" {  
    + ami           = "ami-0c55b159cbfafef0"  
    + instance_type = "t2.micro"  
    (...)  
}  
Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform 从头开始创建了一个全新的 EC2 实例！这是因为每个工作区中的状态文件彼此隔离，因为 Terraform 现在处于 `example1` 工作区中，没有使用默认工作区的状态文件，所以看不到已在默认工作区中创建的 EC2 实例。



图 3-4：默认工作区中 S3 bucket 的存储状态

在新工作区中，运行 `terraform apply` 命令，部署第 2 个 EC2 实例。

```
$ terraform apply  
(...)  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

让我们再重复一次这个练习，创建另一个名为 `example2` 的工作区。

```
$ terraform workspace new example2  
Created and switched to workspace "example2"!  
  
You're now on a new, empty workspace. Workspaces isolate their state,  
so if you run "terraform plan" Terraform will not see any existing state  
for this configuration.
```

运行 `terraform apply` 命令再次部署第 3 个 EC2 实例。

```
$ terraform apply  
(...)  
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

现在，你拥有 3 个可用的工作区，可以通过 `terraform workspace list` 命令查看。

```
$ terraform workspace list
```

```
default
example1
* example2
```

并且可以随时使用 `terraform workspace select` 命令，在它们之间进行切换。

```
$ terraform workspace select example1
Switched to workspace "example1".
```

要了解工作区的工作原理，请查看 S3 bucket；你现在应该看到一个名为 `env` 的新文件夹，图 3-5 所示为开始使用自定义工作区后的 S3 bucket 的内容。



图 3-5：开始使用自定义工作区后的 S3 bucket 的内容

在 `env` 文件夹内，每个工作区各自对应一个文件夹，如图 3-6 所示。

在每个工作区中，Terraform 使用你在 `backend` 配置中的 `key` 参数作为路径，可以找到各自的状态文件：`example1/workspaces-example/terraform.tfstate` 和 `example2/workspaces-example/terraform.tfstate`。换句话说，切换到其他工作区等效于更改状态文件的存储路径。

当你已经部署了 Terraform 模块并想对其进行一些实验（如尝试重构代码）时，这很方便，因为你不希望实验影响已经部署的基础设施的状态。Terraform 工作区允许通过运行 `terraform workspace new` 命令，部署完全相同的基础设施的新副本，但将状态存储在单独的文件中。

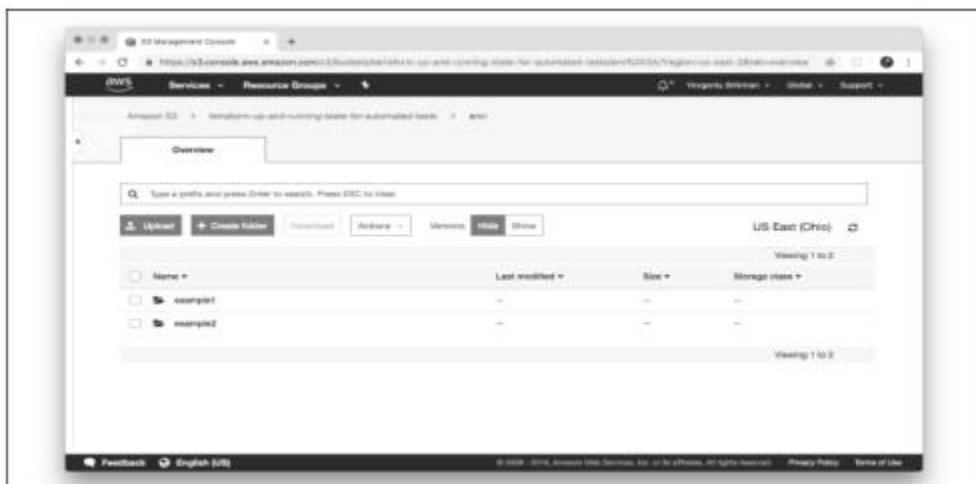


图 3-6：每个工作区各自对应一个文件夹

实际上，甚至可以通过使用表达式读取当前工作区的名称 `terraform.workspace`，再根据所处的工作区来定义模块的不同行为。例如，下面是设置实例在默认工作区中使用 `t2.medium` 类型，但在所有其他工作区中使用 `t2.micro` 类型（如在实验过程中节省开支）的方法。

```
resource "aws_instance" "example" {
    ami = "ami-0c55b159cbfafe1f0"
    instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

上面的代码使用三元语法 (*ternary syntax*)，根据 `terraform.workspace` 的取值，有条件地将 `INSTANCE_TYPE` 设定为 `t2.medium` 或 `t2.micro`。你将在第 5 章看到关于 Terraform 三元语法和条件逻辑的全部详细信息。

Terraform 工作区是快速建立和删除不同版本代码对应的基础设施的好方法，但是它们有一些缺点。

- 所有工作区的状态文件都存储在同一后端（如相同的 S3 bucket）。这意味着你将使用相同的身份验证和访问控制管理所有工作区。这也是工作区隔离并不能彻底隔离环境的一个主要原因（例如，需要完全隔离的预发布环境设施和生产环境设施）。

- 除非运行 `terraform workspace` 命令，否则工作区在代码或终端上将无法被直观地看到。当浏览代码时，模块部署在一个工作区之后和部署在 10 个工作区之后完全相同。这使维护更加困难，因为你无法通过阅读代码来直接了解基础设施。
- 综上所述，工作区是个很容易出错的方法。缺乏可见性使你很容易忘记你所处的工作区，并意外地更改错误的工作区（例如，你的本意是要更改预发布环境，却在生产环境中意外地运行了 `terraform destroy` 命令），由于你必须在所有工作区中使用相同的身份验证机制，因此无法从身份验证层面避免此类错误。

为了获得环境之间的完全隔离，仅仅使用工作区隔离方法是不够的，你应该尝试使用文件布局方法来进行隔离，这将是下一部分的主题。但是在继续之前，请在已创建的 3 个工作区中，分别运行命令 `terraform workspace select <name>` 和 `terraform destroy` 来清理刚刚部署的 3 个 EC2 实例。

## 通过文件布局进行隔离

想要通过文件布局实现环境之间的完全隔离，需要执行以下操作。

- 将每个环境的 Terraform 配置文件放入单独的文件夹中。例如，预发布环境的所有配置都可以存放在名为 `stage` 的文件夹中，而生产环境的所有配置都可以存放在名为 `prod` 的文件夹中。
- 为每个环境配置一个不同的后端，并使用不同的认证和访问控制（例如，每个环境的状态文件可以通过定义，存放在不同的 AWS 账户和不同的 S3 bucket 中）。

使用单独的文件夹，你可以十分清楚地了解到正在部署哪一个环境。使用单独的状态文件和独立的认证机制，将会大大地降低一个环境中的部署过程影响其他环境的可能性。

实际上，你可能希望将隔离概念拓展到环境之外的组件（component）级别，组件通常指一起部署的连续的资源。例如，在为基础设施设置了基本网络拓扑（AWS 术语中特指虚拟私有云及所有与之关联的子网、路由规则、VPN 和网络 ACL）之后，你可能最多每隔几个月更改一次。另一方面，你可能要每天多次部署新版本的 Web 服务器。如果 VPC（虚拟私有云）网络组件和 Web 服务器组件的基础设施在同一组 Terraform 配置文件中进行管理，你会将整个网络拓扑置于随时被损坏的风险中（例如，因为命令要每天多次执行，任何代码中的简单拼写错误或不小心的人为错误命令都会破坏网络基础设施）。

因此，我建议为每个环境（预发布环境、生产环境等）和每个组件（VPC、服务、数据库）使用单独的 Terraform 文件夹（并因此使用单独的状态文件）。让我们来看一下 Terraform 项目建议的文件布局示例。

图 3-7 显示了我使用的 Terraform 项目的典型文件布局。

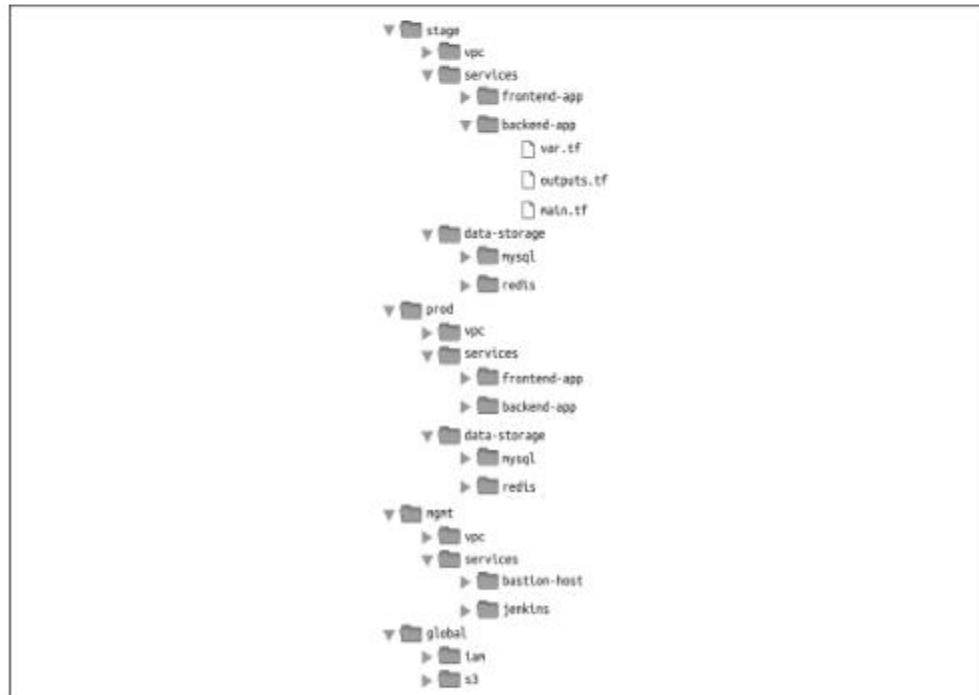


图 3-7：Terraform 项目的典型文件布局

在顶层，每个环境都有单独的文件夹。每个项目的环境定义各有不同，但是典型的命名如下。

#### *stage*

用于运行生产阶段之前预发布活动的环境（如测试）。

#### *prod*

用于运行生产环境的工作负载（如面向用户的应用程序）。

### *mgmt*

用于运行 DevOps 工具的环境（如 bastion host、Jenkins）。

### *global*

用于运行各种环境下都要共享的资源（如 S3、IAM）。

每个环境中，每个组件都有单独的文件夹。每个项目的组件都各不相同，但是典型的命名如下。

### *vpc*

此环境的网络拓扑。

### *services*

在此环境中运行的应用程序或微服务，例如 Ruby on Rails 前端或 Scala 后端。每个应用程序甚至都应该驻留在单独的文件夹中，与其他应用程序隔离。

### *data-storage*

在此环境中运行的数据存储，例如 MySQL 或 Redis。每个数据存储应该驻留在它自己的文件夹中，与其他数据存储隔离。

每一个组件中，都会有相应的 Terraform 的配置文件，其命名规则如下。

### *variables.tf*

输入变量。

### *outputs.tf*

输出变量。

### *main.tf*

资源定义。

当你运行 Terraform 时，它只会在当前的目录中寻找扩展名为 *.tf* 的文件，因此你可以使用任何命名规则。尽管 Terraform 不关心文件名，但你的团队成员会关心。所以请使用一致的、可预测的命名约定，使代码更易于浏览。你将始终知道在哪里可以找到变量、输出或资源。如果单个 Terraform 文件变得越来越大（尤其是 *main.tf*），可以将某些功能分解为单独的文件（如 *iam.tf*、*s3.tf*、*database.tf*），但这也可能表明你应该将代码分成较小的模块，这是我将在第 4 章中探讨的主题。



### 避免复制/粘贴

本节描述的文件布局有很多重复项。例如，相同的 `frontend-app` 和 `backend-app` 同时存在于 `stage` 和 `prod` 文件夹中。不用担心，你无须复制/粘贴所有代码！

在第 4 章，你将看到如何使用 Terraform 模块将所有代码保持为 DRY 的风格。

图 3-8 所示为 Web 服务器集群代码文件布局，让我们使用这个文件夹结构，对第 2 章的 Web 服务器集群代码和本章的 Amazon S3、DynamoDB 代码进行重组。

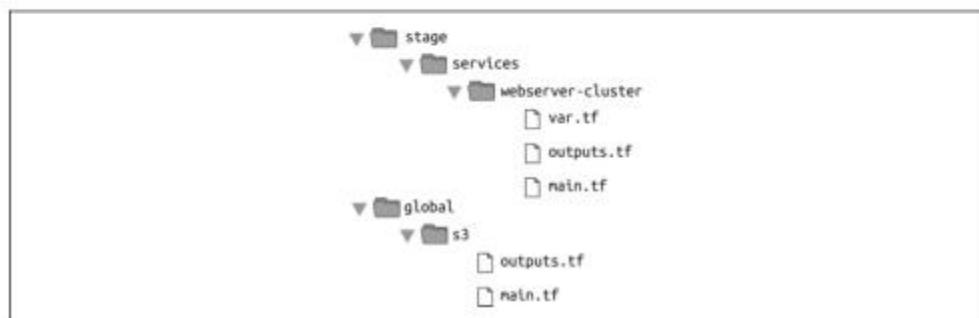


图 3-8：Web 服务器集群代码文件布局

你需要将本章新创建的 S3 bucket 代码移至 `global/s3` 文件夹中，移动输出变量(`s3_bucket_arn` 和 `dynamodb_table_name`)到 `outputs.tf`。移动文件夹时，请确保同时将 `.terraform`(隐藏的) 复制到新位置，这样就无须重新初始化所有内容了。

请将第 2 章创建的 Web 服务器集群移至 `stage/services/webserver-cluster` 目录(首先将其看作 Web 服务器集群的测试或预发布版本；你将在下一章继续添加生产版本)。同样，请确保复制 `.terraform` 文件夹，将输入变量移至 `variables.tf`，并将输出变量移至 `outputs.tf`。

你还应该将 S3 作为 Web 服务器集群的 `backend`。可以从 `global/s3/main.tf` 文件中复制/粘贴 `backend` 代码块，但请将 `key` 更改为与 Web 服务器 Terraform 代码相同的文件夹路径：`stage/services/webserver-cluster/terraform.tfstate`。这样会使版本控制系统中 Terraform 代码的布局与 S3 中 Terraform 状态文件的路径之间形成 1:1 映射，两者之间的关系显而易见。上文中的 S3 模块已经按照命名约定设置了 `key` 参数。

这种文件路径格式的定义便于浏览代码，便于理解每个环境中的已部署的组件。它还可以在环境之间及环境中的组件之间提供最佳的隔离效果，确保出现问题时把损失控制在整个基础设施的局部。

从某种意义上说，该特性同时也是一个缺点：将组件拆分为多个单独的文件夹，可以防止你使用一条命令意外地破坏整个基础设施，但同时也阻止了你通过一条命令来创建整个基础设施。对于单一环境，如果一个 Terraform 配置文件保存所有组件的定义，你可以通过运行一次 `terraform apply` 命令，创建完整环境。但是，如果所有组件都分别保存在单独的文件夹中，那么你需要在每个文件夹中多次运行 `terraform apply`（请注意，使用 Terragrunt，可以通过 `apply-all` 命令来自动执行此过程<sup>1</sup>）。

这种文件布局还有另一个问题：调用资源依赖关系将变得十分困难。如果你的应用代码和数据库代码位于同一个 Terraform 配置文件，那么应用程序代码可以通过属性引用直接访问数据库代码的属性（例如，通过 `aws_db_instance.foo.address` 引用得到数据库地址）。如果使用我推荐的方法将应用程序代码和数据库代码存放在不同的文件夹中，则无法使用属性引用。幸运的是，Terraform 的 `terraform_remote_state` 数据源为这个问题提供了一个解决方案。

## terraform\_remote\_state 数据源

在第 2 章中，你知道了如何使用数据源从 AWS 读取信息，例如 `aws_subnet_ids` 数据源，该数据源返回了 VPC 中的子网列表。Terraform 状态文件也有一个十分有用的数据源：`terraform_remote_state`。你可以使用这个数据源直接读取其他 Terraform 配置写入的 Terraform 状态文件。

让我们来看一个例子。Web 服务器集群需要与 MySQL 数据库通信。运行可伸缩、安全、持久及高度可用的数据库是一项繁重的工作。你可以让 AWS 通过使用 Amazon 的关系数据库服务（*Relational Database Service, RDS*）来替你管理，图 3-9 所示为 Web 服务器集群与部署在 Amazon RDS 之上的 MySQL 通信。RDS 支持各种数据库，包括 MySQL、PostgreSQL、SQL Server 和 Oracle。

---

<sup>1</sup> 更多有关信息，请参见 Terragrunt 的文档见参考资料第 3 章[9]。

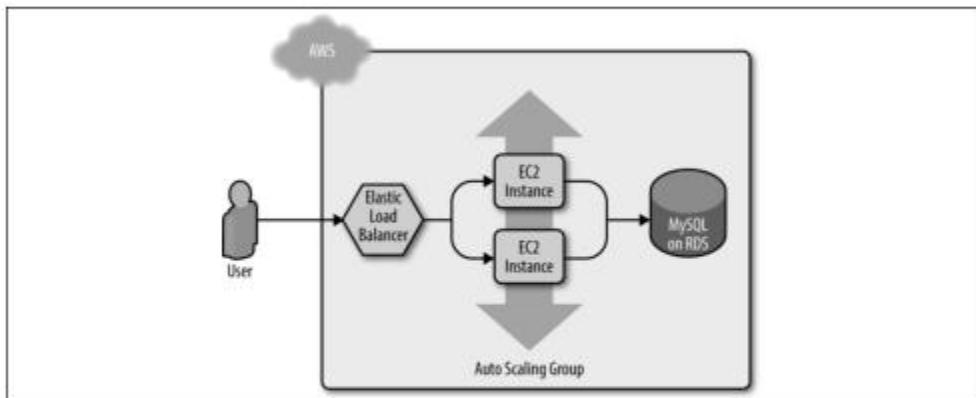


图 3-9：Web 服务器集群与部署在 Amazon RDS 之上的 MySQL 通信

因为对 Web 服务器集群部署更新频率要比数据库的部署频率高得多，如果不想每次部署都存在意外破坏数据库的可能，那么应该将 Web 服务器集群和 MySQL 数据库定义存放在不同的 Terraform 配置文件中。第一步建立一个新文件夹 `stage/data-stores/mysql`，并在文件夹中创建标准的 Terraform 文件 (`main.tf`、`variables.tf`、`outputs.tf`)，图 3-10 所示为在 `stage/data-stores` 文件夹中创建数据库相关的代码。



图 3-10：在 `stage/data-stores` 文件夹中创建数据库相关的代码

接下来，在 `stage/data-stores/mysql/main.tf` 中创建数据库资源。

```
provider "aws" {
    region = "us-east-2"
}

resource "aws_db_instance" "example" {
    identifier_prefix = "terraform-up-and-running"
    engine            = "mysql"
    allocated_storage = 10
    instance_class    = "db.t2.micro"
    name              = "example_database"
    username          = "admin"

    # 我们应该如何设置密码
    password          = "???"
}
```

在文件的开头，你将看到典型的 `provider` 资源的声明，但在其下方是一个名为 `aws_db_instance` 的新资源。这个资源可以在 RDS 中创建一个数据库。在该代码块中，将 RDS 配置为一个拥有 10GB 存储空间、`db.t2.micro` 套餐的 MySQL 实例，该实例具有一个虚拟 CPU 和 1GB 的内存，并且是 AWS 免费的一部分。

请注意，`aws_db_instance` 资源中一个必要参数是用于数据库的密码。因为这是一个机密信息，所以你不应将其直接以纯文本形式输入代码中！这里有两个更好的办法可以将机密信息传递到 Terraform 资源。

处理机密信息的第一种方法是，使用 Terraform 数据源从机密信息存储库中读取机密信息。例如，你可以在 AWS Secrets Manager 中存储机密信息（例如数据库密码），这是 AWS 专为存储敏感数据而提供的托管服务。你可以使用 AWS Secrets Manager UI 来存储密码，然后在 Terraform 代码中通过 `aws_secretsmanager_ager_secret_version` 数据源读取密码。

```
provider "aws" {
    region      = "us-east-2"
}

resource "aws_db_instance" "example" {
    identifier_prefix = "terraform-up-and-running"
    engine           = "mysql"
```

```
allocated_storage = 10
instance_class    = "db.t2.micro"
name              = "example_database"
username          = "admin"

password =
  data.aws_secretsmanager_secret_version.db_password.secret_string
}

data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "mysql-master-password-stage"
}
```

以下是你可能感兴趣的一些机密信息存储服务和数据源的组合。

- AWS Secrets Manager 和 `aws_secretsmanager_secret_version` 数据源 ( 如先前代码所示 )
- AWS Systems Manager Parameter Store 和 `aws_ssm_parameter` 数据源
- AWS Key Management Service ( AWS KMS ) 和 `aws_kms_secrets` 数据源
- Google Cloud KMS 和 `google_kms_secret` 数据源
- Azure Key Vault 和 `azurerm_key_vault_secret` 数据源
- HashiCorp Vault 和 `vault_generic_secret` 数据源

处理机密信息的第二个方法是，使用 Terraform 范围之外的系统（如 1Password、LastPass 或 OS X 钥匙串之类的密码管理器）管理机密信息，然后将机密信息通过环境变量传递到 Terraform 中。为此，请在 `stage/data-stores/mysql/variables.tf` 中，定义一个名为 `db_password` 的输入变量。

```
variable "db_password" {
  description = "The password for the database"
  type        = string
}
```

注意，该变量故意没有设置默认值。因为你不应以纯文本格式存储数据库密码或任何敏感信息。你将使用环境变量来进行赋值。

在 Terraform 配置中，对于每个输入变量 `foo`，都可以使用环境变量 `TF_VAR_foo` 为其赋值。在 Linux、UNIX、OS X 系统中，输入变量 `db_password` 可以通过环境变量 `TF_VAR_db_`

`password` 来设置。

```
$ export TF_VAR_db_password ="(YOUR_DB_PASSWORD)"  
$terraform apply  
  
(...)
```

请注意，`export` 命令前故意留有一个空格，这样做可以避免机密信息存储在 Bash 历史记录中。<sup>1</sup>还有一种更好的方法可以避免意外将机密信息以纯文本形式存储在磁盘上，即使用命令行友好的机密信息存储区，例如 `pass`（见参考资料第 3 章[10]）中，使用子进程安全地将机密信息从 `pass` 读取到环境变量中。

```
$ export TF_VAR_db_password = $(pass database-password)  
$ terraform apply  
  
(...)
```



#### 机密信息始终存储在 Terraform 状态中

从机密信息存储区或环境变量中读取机密信息是确保机密信息不存储在纯文本中的一种好习惯，但是还是需要提醒你：无论你如何读取机密信息，如果将其传递给 Terraform 资源（如参数 `aws_db_instance`），则该机密信息将以纯文本形式存储在 Terraform 状态文件中。

这是 Terraform 的一个已知弱点，现在没有有效的解决方案，因此要特别注意存储状态文件的方式（例如，始终启用加密），以及谁可以访问这些状态文件（例如，使用 IAM 权限来锁定对状态文件所处的 S3 bucket 的访问权限）！

现在，你已经配置了密码，下一步是配置此模块，将其状态存储在先前在 S3 bucket 中创建的路径 `stage/data-stores/mysql/terraform.tfstate` 中。

```
terraform {  
  backend "s3" {  
    # 替换为你的 bucket 名称
```

<sup>1</sup> 在大多数的 Linux/UNIX/OS X 命令行上，每一个执行过的命令都会被存储在某种历史文件中（例如 `/.bash_history`）。如果是以空格开头的命令，大多数命令行程序将跳过将该命令写入历史文件的操作。请注意，如果你的命令行默认情况下未启用此功能，可以将环境变量 `HISTCONTROL` 设置为 `ignoreboth` 来启用此功能。

```
bucket      = "terraform-up-and-running-state"
key         = "stage/data-stores/mysql/terraform.tfstate"
region      = "us-east-2"

# 用你的 DynamoDB 表名称替换它
dynamodb_table = "terraform-up-and-running-locks"
encrypt       = true
}

}
```

然后运行 `terraform init` 和 `terraform apply` 命令来创建数据库。请注意，即使配置一个小型数据库，Amazon RDS 也可能需要大约 10min，因此请耐心等待。

现在你有了数据库，如何将其地址和端口提供给 Web 服务器集群？第一步是将两个输出变量添加到 `stage/data-stores/mysql/outputs.tf` 文件中。

```
output "address" {
  value      = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}

output "port" {
  value      = aws_db_instance.example.port
  description = "The port the database is listening on"
}
```

运行 `terraform apply` 命令，终端中的输出如下所示。

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

address = tf-2016111123.cowu6nts6srx.us-east-2.rds.amazonaws.com
port = 3306
```

现在，数据库模块的输出已经被存储在 Terraform 状态文件中，状态文件位于 S3 bucket —— `stage/data-stores/mysql/terraform.tfstate` 中。接下来，Web 服务器集群代码可以通过使用 `terraform_remote_state` 数据源来读取这个状态文件的数据。`stage/services/webserver-cluster/main.tf` 中数据源定义如下。

```

data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = "YOUR_BUCKET_NAME"
    key    = "stage / data-stores / mysql / terraform.tfstate"
    region = "us-east-2"
  }
}

```

Web 服务器集群代码将使用 `terraform_remote_state` 数据源，读取数据库模块存储在 S3 bucket 文件夹中的状态文件，如图 3-11 所示，数据库将其状态文件写入 S3 bucket( 顶部 )，Web 服务器集群从同一个 bucket 中读取状态。

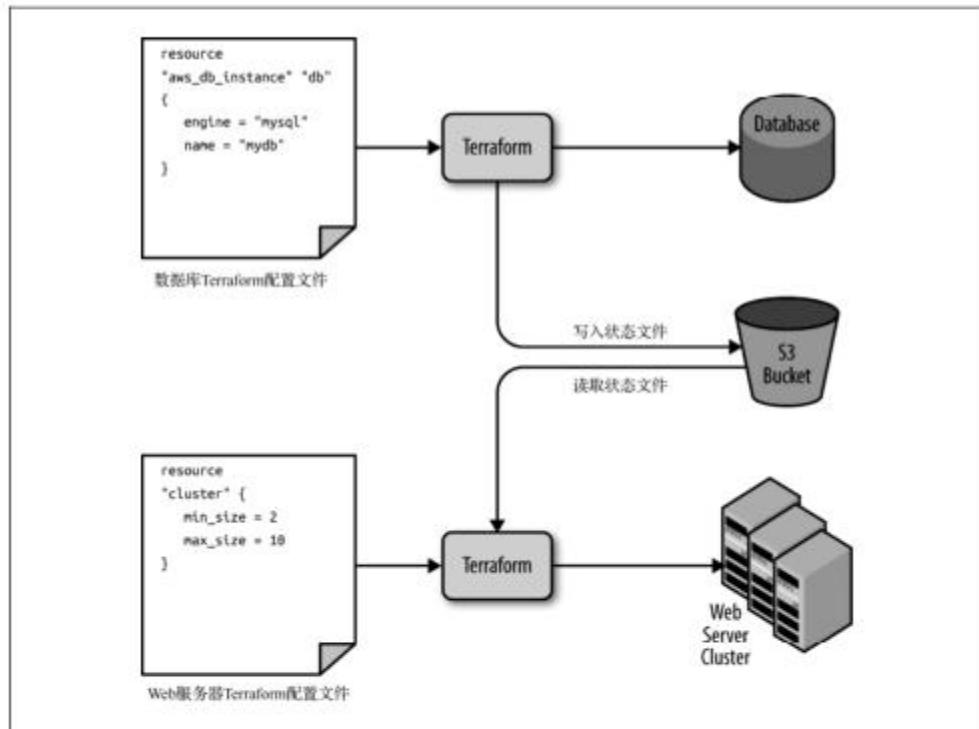


图 3-11：数据库将其状态文件写入 S3 bucket，Web 服务器集群从同一个 bucket 中读取状态

有一点很重要，像所有 Terraform 数据源一样，`terraform_remote_state` 返回的数据对象是只读的。在 Web 服务器集群中，无论执行任何 Terraform 代码操作都不会修改该状态文件，因此读取数据库状态的操作，不会导致数据库本身出现任何问题。

数据库的所有输出变量都存储在状态文件中，通过 `terraform_remote_state` 数据源读取其属性引用的方式如下。

```
data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>
```

例如，下面是 Web 服务器集群实例中的最新版本用户数据，通过 `terraform_remote_state` 数据源读取数据库地址和端口，并在 HTTP 响应中发布该信息。

```
user_data = <<EOF
#!/bin/bash
echo "Hello, World" >> index.html
echo "${data.terraform_remote_state.db.outputs.address}" >> index.html
echo "${data.terraform_remote_state.db.outputs.port}" >> index.html
nohup busybox httpd -f -p ${var.server_port} &
EOF
```

随着 User Data 脚本的增长，这种脚本嵌套变得越来越混乱。通常，将一种编程语言（Bash）嵌入到另一种编程语言（Terraform）会增加维护每种编程语言的难度，因此让我们在这里暂停一下，看看如何能直接使用外部的 Bash 脚本。你可以使用内置函数 `file` 和 `template_file` 数据源实现这一点。让我们一起讨论一下。

Terraform 包含许多内置函数，可以通过以下的表达式执行这些函数。

```
function_name(...)
```

例如，`format` 函数。

```
format (<FMT>, <ARGS>, ...)
```

此函数是根据 `FMT` 中定义的 `sprintf` 字符串语法，格式化 `ARGS` 中的参数的。<sup>1</sup> 运行 `terraform console` 命令打开一个交互式控制台，通过交互式控制台可以很好地实验内置函数的功能。运行 Terraform 语法，查询基础设施的状态，并立即返回结果。

---

<sup>1</sup> 在参考资料第 3 章 [11] 中可以看到 `sprintf` 语法的文档。

```
$ terraform console  
> format ("% .3f", 3.14159265359)  
3.142
```

请注意，Terraform 控制台的只读访问方式不会意外改变基础设施或状态文件。

这里有许多内置函数可以操作字符串、数字、列表和映射。<sup>1</sup> `file` 函数是其中之一。

```
file (<PATH>)
```

此函数读取 PATH 参数中定义的文件，并以字符串形式返回其内容。例如，你可以将你的 User Data 脚本放入 `stage/services/webserver-cluster/user-data.sh` 文件中，并将其内容作为字符串形式加载，如下所示。

```
file (" user-data.sh")
```

难点是，在 Web 服务器集群的用户数据脚本中，需要 Terraform 的一些动态数据，包括服务器端口、数据库地址和数据库端口。之前你可以使用 Terraform 插值，将引用嵌入到 Terraform 代码的用户数据脚本中。但是这不适用于 `file` 函数，你必须通过 `template_file` 数据源一起工作。

`template_file` 数据源有两个参数：`template`，定义将要被处理的字符串 `vars`，是在处理字符串时将要用到的变量集合的映射，它有一个被称为 `rendered` 的输出属性，这是对模板进行处理后的结果。下面实际操作一下，将以下 `template_file` 数据源代码添加到 `stage/services/webserver-cluster/main.tf` 文件中。

```
data "template_file" "user_data" {  
    template = file("user-data.sh")  
  
    vars = {  
        server_port = var.server_port  
        db_address = data.terraform_remote_state.db.outputs.address  
        db_port     = data.terraform_remote_state.db.outputs.port  
    }  
}
```

从上面的代码可以看到，`template` 参数指向 `user-data.sh` 脚本，`vars` 参数包括 3 个 User

---

<sup>1</sup> 这是完整的内置函数列表（见参考资料第 3 章[12]）。

Data 脚本中需要的变量：服务器端口、数据库地址和数据库端口。要使用这些变量，你需要按以下方式更新 *stage/services/webserver-cluster/user-data.sh* 脚本。

```
#!/bin/bash

cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

请注意，此 Bash 脚本与原始脚本相比有一些变化。

- 它使用 Terraform 的标准插值语法识别变量，但是唯一可用的变量是 `template_file` 数据源中的 `vars` 参数定义过的变量映射。请注意，访问这些变量不需要任何前缀修饰，例如，应该使用 `server_port`，而不是 `var.server_port`。
- 该脚本现在包括一些 HTML 语法（如`<h1>`），使输出在 Web 浏览器中更具可读性。



### 关于外部化文件

将 User Data 中的脚本提取到外部文件的好处是，你可以为其编写单元测试。测试代码甚至可以使用环境变量来填充内置插值变量，因为 Bash 语法的环境变量与 Terraform 的内插语法相同。你可以按照以下几行代码为 *user-data.sh* 编写自动测试。

```
export db_address=12.34.56.78
export db_port=5555
export server_port=8888

./user-data.sh

output=$(curl "http://localhost:$server_port")

if [[ $output == *"Hello, World"* ]]; then
    echo "Success! Got expected text from server."
else
    echo "Error. Did not get back expected text 'Hello, World'." 
fi
```

最后一步是更新 `aws_launch_configuration` 资源的 `user_data` 参数，使其指向 `template_file` 数据源的 `rendered` 输出变量。

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data      = data.template_file.user_data.rendered

  # 在使用带有自动伸缩群启动配置时是必需的
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

这样的外部代码比内置的 Bash 脚本清晰了许多！

如果再次使用 `terraform apply` 命令部署此集群，请等待实例在 ALB 中注册，然后在 Web 浏览器中打开 ALB URL，你将看到类似于图 3-12 所示的 Web 服务器集群以编程方式访问数据库地址和端口。

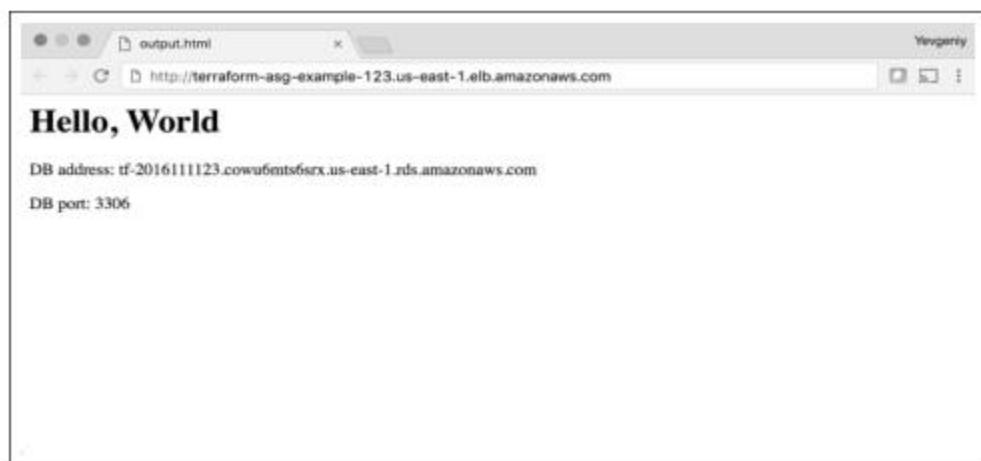


图 3-12：Web 服务器集群以编程方式访问数据库地址和端口

现在你的 Web 服务器集群可以通过 Terraform 以编程方式访问数据库地址和端口了。如果你使用的是真实的 Web 框架（如 Ruby on Rails），则可以将地址和端口设置为环境变量，或将它们写入配置文件，以便你的数据库驱动（如 ActiveRecord）可以使用它们与数据库通信。

## 小结

本章以大量篇幅讨论了为什么需要使用隔离、锁定和状态文件，这是因为基础设施即代码（IaC）与常规编码具有不同的侧重点。当你为典型应用编写代码时，大多数错误不会非常严重，只会破坏单个应用的一小部分。但当你编写用于控制基础设施的代码时，单一的错误也可能会破坏所有应用程序、所有数据存储、整个网络拓扑，以及几乎所有其他内容，因此错误往往会更加严重。所以我建议在 IaC 上使用比典型应用程序代码更多的安全机制。<sup>1</sup>

本章推荐的文件布局的一个缺点是它会导致代码内容的重复。如果在预发布环境和生产环境中同时运行 Web 服务器集群，如何避免在 *stage/services/webserver-cluster* 和 *prod/services/webserver-cluster* 之间复制/粘贴大量代码呢？答案是你需要使用 Terraform 模块，这是第 4 章的主题。

---

<sup>1</sup> 有关软件安全机制的更多信息，请参见参考资料第 3 章[13]。

## 使用 Terraform 模块创建可重用基础设施

在第 3 章结尾，你部署了如图 4-1 所示的负载均衡器、Web 服务器集群和数据库。

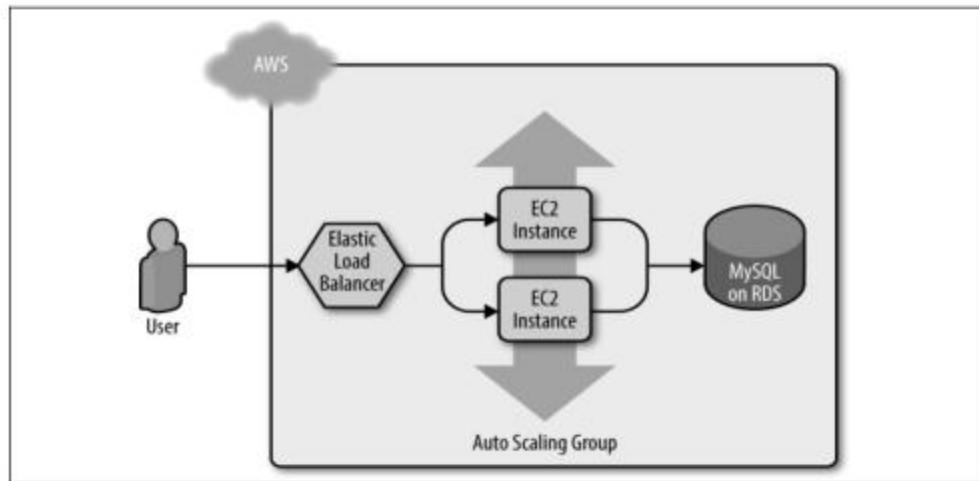


图 4-1：负载均衡器、Web 服务器集群和数据库

作为第一个部署环境，这已经是很不错的成果了，但通常我们需要至少两个独立环境：一个团队内部测试环境（预发布环境）和一个真正的用户可以访问的生产环境，每个环境都有自己的负载均衡器、Web 服务器集群和数据库，如图 4-2 所示。理想情况下，这两种环境应该是完全相同的，但实际工作中为了降低开销，预发布环境中运行的服务器在配置上会更小一些，在数量上会更少一些。

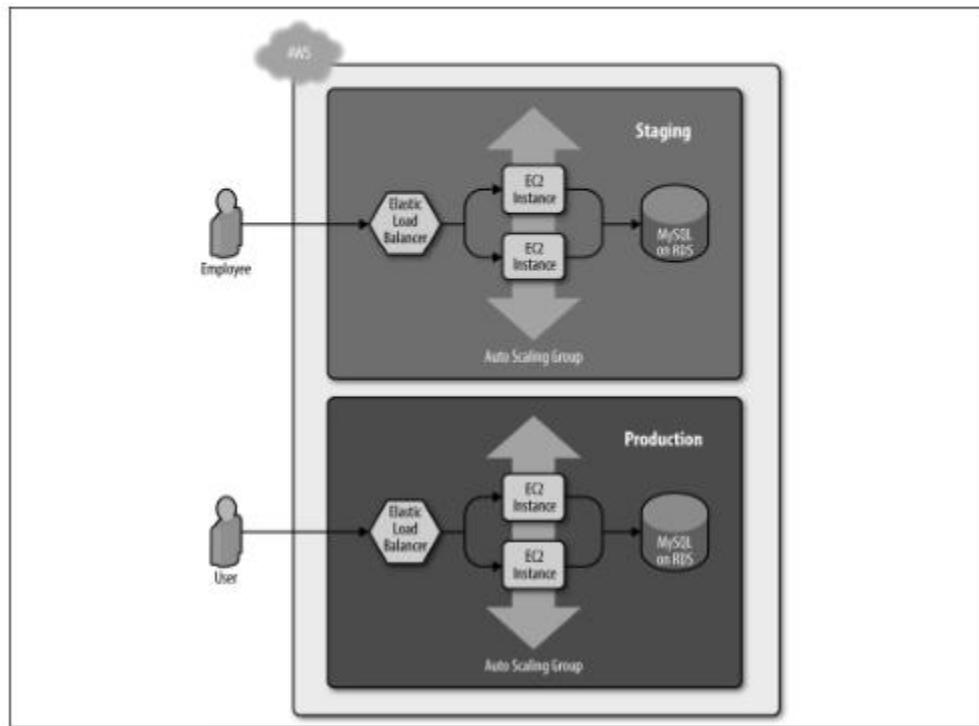


图 4-2：两个独立环境，每个环境都有自己的负载均衡器、Web 服务器集群和数据库

如何快速地添加生产环境，又不必复制/粘贴预发布环境中的所有代码呢？例如，如何避免将 `stage/services/webserver-cluster` 目录中的所有代码复制/粘贴到 `prod/services/webserver-cluster` 目录中？以及避免将 `stage/data-stores/mysql` 中的所有代码复制/粘贴到 `prod/data-stores/mysql` 目录中？

在像 Ruby 之类的通用编程语言中，可以将多个地方重复出现的相同代码转化为函数，并在重复出现的位置调用该函数。

```
def example_function()
    puts "Hello, World"
end
# 放置在代码中的其他位置
example_function()
```

在 Terraform 使用环境下，你可以将代码放在 *Terraform* 模块内，并在代码的其他位置重复调用该模块。通过模块可以避免复制/粘贴相同的代码到预发布环境和生产环境中，而是从两个环境中分别调用同一代码模块。如图 4-3 所示，将代码放入模块中可以在多个环境中重复使用该代码。

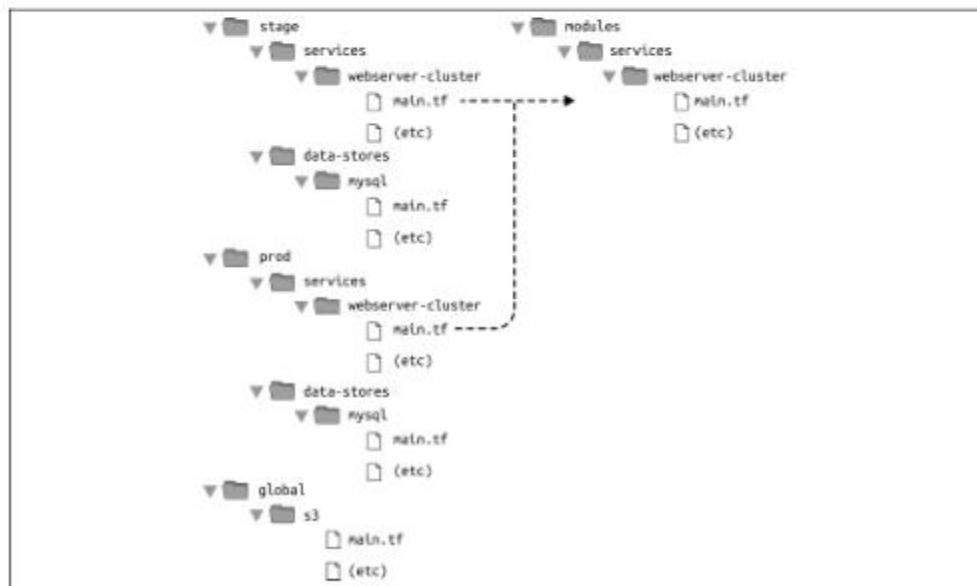


图 4-3：将代码放入模块中可以在多个环境中重复使用该代码

模块化是编写可重用、可维护和可测试的 Terraform 代码的关键要素。一旦开始使用，你一定会喜欢上模块并开始尝试：将所有代码功能模块化，在公司中创建模块共享库，使用网上发现的模块，甚至将整个基础设施看成可重复使用的模块的集合。

在本章中，我将通过以下主题展示如何创建和使用 Terraform 模块。

- 模块基础知识
- 模块的输入
- 模块的局部变量
- 模块的输出

- 模块中的陷阱
- 模块版本控制

## 模块基础知识

Terraform 模块非常简单：位于同一文件夹中的任何 Terraform 配置文件的集合都是一个模块。到目前为止，你编写的所有 Terraform 配置文件，在技术上都可以被称作模块。直接运行、部署这些模块（当前工作目录中的模块被称为根模块）不是特别有意义。真正体现模块功能的方法是，从一个模块中调用另一个模块。

例如，让我们将 `stage/services/webserver-cluster` 中的代码，包括 Auto Scaling Group( ASG )、Application Load Balancer ( ALB )、安全组和许多其他资源，转换为可重用的模块。

第一步，在 `stage/services/webserver-cluster` 中运行 `terraform destroy` 命令，清理之前创建的所有资源。接下来，创建一个新的名为 `modules` 的顶级文件夹，并将所有文件从 `stage/services/webserver-cluster` 文件夹移至 `modules/services/webserver-cluster` 文件夹。最终具有模块和预发布环境的文件夹结构如图 4-4 所示。

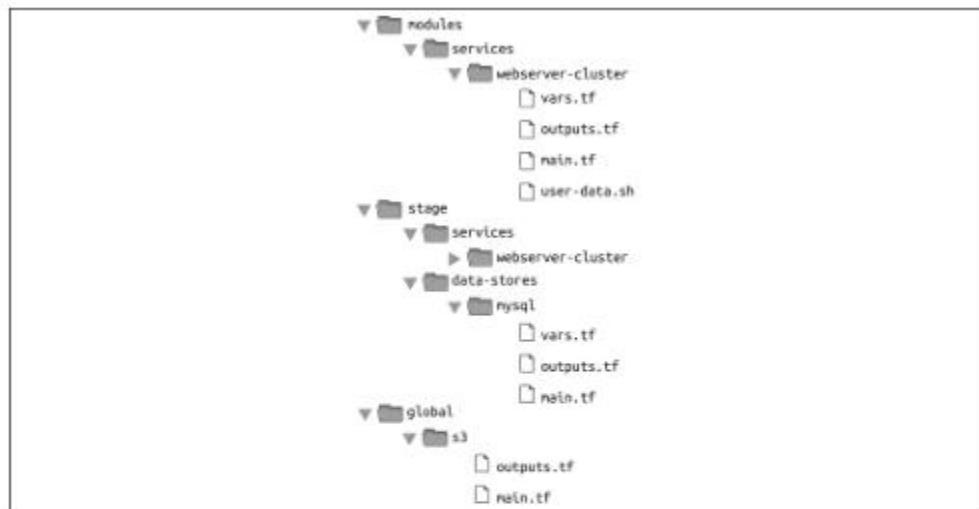


图 4-4：最终具有模块和预发布环境的文件夹结构

打开 `modules/services/webserver-cluster` 目录下的 `main.tf` 文件，删除 `provider` 定义。因为提供商的相关定义应该出现在调用模块的用户代码中，而不是模块本身的配置中。

现在，通过预发布环境使用此模块的语法。

```
module "<NAME>" {
    source = "<SOURCE>

    [CONFIG ...]
}
```

其中，`NAME` 是一个标识符，在整个 Terraform 代码中可以通过使用该标识符来引用此模块（如 `web-service`），`SOURCE` 是模块代码的路径（如 `modules/services/webserver-cluster`），`CONFIG` 包括一个或多个该模块的特有参数。例如，你可以在 `stage/services/webserver-cluster/main.tf` 中创建一个新文件，通过以下方式调用 `webserver-cluster` 模块。

```
provider "aws" {
    region = "us-east-2"
}

module "webserver_cluster" {
    source = "../../modules/services/webserver-cluster"
}
```

然后以同样的方式，在 `prod/services/webserver-cluster/main.tf` 文件中添加如下代码，这样便能在生产环境中重复使用相同的模块了。

```
provider "aws" {
    region = "us-east-2"
}

module "webserver_cluster" {
    source = "../../modules/services/webserver-cluster"
}
```

这样可以实现在多个环境中重复使用代码，从而减少复制/粘贴操作。请注意，无论何时将模块添加到 Terraform 配置中或修改 `source` 模块的参数，都需要在运行 `plan` 或 `apply` 命令之前重新运行 `init` 命令。

```
$ terraform init
Initializing modules...
```

```
* webserver_cluster in ../../modules/services/webserver-cluster  
  
Initializing the backend...  
  
Initializing provider plugins...  
  
Terraform has been successfully initialized!
```

这里展示了关于 `init` 命令的全部技巧：通过一个简单的命令，它可以下载提供商代码和模块并配置后端。

在运行 `apply` 之前，请注意 `webserver-cluster` 模块存在以下问题：文件中所有名称都是静态编码的。包括安全组、ALB 和其他资源的名称都是静态编码的，因此，如果你多次使用此模块，则会出现命名冲突错误。数据库详细信息也是静态编码的，因为你复制到 `modules/services/webserver-cluster` 中的 `main.tf` 文件使用 `terraform_remote_state` 数据源来确定数据库地址和端口，而 `terraform_remote_state` 被静态编码指向预发布环境。

要解决这些问题，需要向 `webserver-cluster` 模块添加可配置的输入，使其在不同的环境中有一样的表现。

## 模块的输入

在 Ruby 之类的通用编程语言中，可以通过添加输入参数来配置函数。

```
def example_function(param1, param2)  
  puts "Hello, #{param1} #{param2}"  
end  
  
# 代码中的其他位置  
example_function("foo","bar")
```

Terraform 的模块也可以具有输入参数。要定义它们，可以使用一种你已经熟悉的机制：输入变量。打开 `modules/services/webserver-cluster/variables.tf` 并添加 3 个新的输入变量。

```
variable "cluster_name" {  
  description = "The name to use for all the cluster resources"  
  type        = string  
}
```

```
variable "db_remote_state_bucket" {
    description = "The name of the S3 bucket for the database's remote state"
    type        = string
}

variable "db_remote_state_key" {
    description = "The path for the database's remote state in S3"
    type        = string
}
```

接下来，在 `modules/services/webserver-cluster/main.tf` 文件中，使用 `var.cluster_name` 替换静态编码名称（如代替 `terraform-asg-example`）。这是对 ALB 安全组进行的修改。

```
resource "aws_security_group" "alb" {
    name = "${var.cluster_name}-alb"

    ingress {
        from_port   = 80
        to_port     = 80
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }

    egress {
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

请注意这里是如何将 `name` 参数设置为 " `${var.cluster_name}-alb`" 的。你需要对另一个 `aws_security_group` 资源进行类似的替换操作（如给它取名为 " `${var.cluster_name}-instance`"），`aws_alb` 资源和 `aws_autoscaling_group` 资源的 `tag` 部分也是如此。

你还应该更新 `terraform_remote_state` 数据源，分别使用 `db_remote_state_bucket` 和 `db_remote_state_key` 作为 `bucket` 和 `key` 的参数，以确保读取正确的环境状态文件。

```
data "terraform_remote_state" "db" {
    backend = "s3"
```

```
config = [
    bucket  = var.db_remote_state_bucket
    key     = var.db_remote_state_key
    region  = "us-east-2"
]
}
```

现在，在预发布环境的 *stage/services/webserver-cluster/main.tf* 文件中，需要相应地设置这些新的输入变量。

```
module "webserver_cluster"{
    source          = "../../modules/services/webserver-cluster"

    cluster_name    = "webservers-stage"
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
    db_remote_state_key   = "stage / data-stores / mysql / terraform.tfstate"
}
```

同样，应该对生产环境中的 *prod/services/webserver-cluster/main.tf* 文件内容进行相应的修改。

```
module "webserver_cluster" {
    source          = "../../modules/services/webserver-cluster"

    cluster_name    = "webservers-prod"
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
    db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"
}
```



生产环境的数据库现在还并不存在。作为练习，请自行添加一个与预发布环境相似的生产环境数据库。

如上所述，设置模块的输入变量的语法与设置资源的输入变量的语法是相同的。输入变量可以看作模块的 API，控制模块在不同环境中的行为。本示例仅针对不同的环境使用不同的名称，你或许还需要配置其他参数。例如，在预发布环境中，可能需要运行一个较小的 Web 服务器集群以节省资金，但在生产环境中，你可能想要运行一个较大的集群以处理大量流量。为此，可以在 *modules/services/webserver-cluster/variables.tf* 中再添加 3 个输入变量。

```

variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g. t2.micro)"
  type        = string
}

variable "min_size" {
  description = "The minimum number of EC2 Instances in the ASG"
  type        = number
}

variable "max_size" {
  description = "The maximum number of EC2 Instances in the ASG"
  type        = number
}

```

接下来，将 *modules/services/webserver-cluster/main.tf* 中的启动配置的 `instance_type` 参数更新为新的输入变量 `var.instance_type`。

```

resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data     = data.template_file.user_data.rendered

  # 在使用带有自动伸缩群启动的配置时必需
  # https://www.terraform.io/docs/providers/aws/r/launch\_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}

```

在同一个文件中，还应该更新 ASG 设置，将 `min_size` 和 `max_size` 参数分别更改为输入变量 `var.min_size` 和 `var.max_size`。

```

resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier   = data.aws_subnet_ids.default.ids
  target_group_arns     = [aws_lb_target_group.asg.arn]
  health_check_type     = "ELB"

  min_size = var.min_size
  max_size = var.max_size
}

```

```
    tag {
        key          = "Name"
        value        = var.cluster_name
        propagate_at_launch = true
    }
}
```

现在，在预发布环境（*stage/services/webserver-cluster/main.tf*）中，可以通过将 `instance_type` 设置为“t2.micro”和将 `min_size` 和 `max_size` 分别设置为 2，保持小集群和低开销。

```
module "webserver_cluster" {
    source          = "../../modules/services/webserver-cluster"

    cluster_name    = "webservers-stage"
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
    db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

    instance_type    = "t2.micro"
    min_size         = 2
    max_size         = 2
}
```

另一方面，在生产环境中，可以使用具有更多 CPU 和内存的 `instance_type`，例如 `m4.large`（请注意，此 Instance 类型不是 AWS 免费套餐的一部分。因此，如果只是进行学习且不想产生开销，请继续设置 `instance_type` 为“t2.micro”），然后可以将 `max_size` 设置为 10，允许集群根据负载情况而收缩或增长（不用担心，集群最初只会启动两个实例）。

```
module "webserver_cluster" {
    source          = "../../modules/services/webserver-cluster"

    cluster_name    = "webservers-prod"
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
    db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

    instance_type    = "m4.large"
    min_size         = 2
    max_size         = 10
}
```

## 模块的局部变量

使用输入变量来定义模块的输入是很好的解决方法，但是如果仅需要在模块内部定义变量来进行中间计算，或者只是保持代码 DRY，不想将该变量公开为可配置输入该怎么办？例如，`webserver-cluster` 模块中的负载均衡器 (`modules/services/webserver-cluster/main.tf`)，其侦听端口为 80（HTTP 的默认端口）。这个端口号被复制/粘贴到多个地方，包括负载均衡器的侦听器。

```
resource "aws_lb_listener" "http" {
    load_balancer_arn = aws_lb.example.arn
    port              = 80
    protocol          = "HTTP"

    # 默认情况下，返回一个简单的 404 页面
    default_action {
        type      = "fixed-response"

        fixed_response {
            content_type = "text/plain"
            message_body = "404: page not found"
            status_code  = 404
        }
    }
}
```

还包括负载均衡器的安全组。

```
resource "aws_security_group" "alb" {
    name = "${var.cluster_name}-alb"

    ingress {
        from_port   = 80
        to_port     = 80
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }

    egress {
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
    }
}
```

```
    cidr_blocks = ["0.0.0.0/0"]
}
}
```

安全组中的值，包括“所有 IP”CIDR 块 0.0.0.0/0，“任何端口”值 0 和“任何协议”值 “-1”也会在整个模块中的多个位置被复制/粘贴。这种方式会使代码难以阅读和维护。你可以将值提取为输入变量，但是那样的话，模块的用户就能够（意外地）改写这些值了，这或许是你不希望看到的。所以除了使用输入变量，你还可以将它们定义为局部变量，放在 `locals` 模块中。

```
locals {
    http_port      = 80
    any_port       = 0
    any_protocol   = "-1"
    tcp_protocol   = "tcp"
    all_ips        = ["0.0.0.0/0"]
}
```

本地变量允许用户为任何 Terraform 表达式分配名称，并在整个模块中引用该名称。这些名称仅在模块中可见，因此它们对其他模块没有影响，并且不能从模块外部改写这些值。要读取本地的值，需要使用本地引用，该引用使用以下语法。

```
local.<NAME>
```

使用以下语法更新你的负载均衡器的侦听器。

```
resource "aws_lb_listener" "http" {
    load_balancer_arn= aws_lb.example.arn
    port            = local.http_port
    protocol        = "HTTP"

    # 默认情况下，返回一个简单的 404 页面
    default_action {
        type = "fixed-response"

        fixed_response {
            content_type= "text/plain"
            message_body= "404: page not found"
            status_code = 404
        }
    }
}
```

还可以更新所有模块中的安全组，包括负载均衡器的安全组。

```
resource "aws_security_group" "alb" {
    name = "${var.cluster_name}-alb"

    ingress {
        from_port   = local.http_port
        to_port     = local.http_port
        protocol    = local.tcp_protocol
        cidr_blocks = local.all_ips
    }

    egress {
        from_port   = local.any_port
        to_port     = local.any_port
        protocol    = local.any_protocol
        cidr_blocks = local.all_ips
    }
}
```

本地变量使你的代码更易于阅读和维护，因此请经常使用它们。

## 模块的输出

ASG 的一项强大功能是，通过配置使其根据运行负载的情况，增加或减少服务器数量。一种方式是使用计划操作 (*scheduled action*)，该操作可以在计划的时间点更改集群的大小。例如，如果预计在正常工作时间内流向集群的流量会很大，则可以使用计划操作在上午 9 点增加服务器数量，在下午 5 点减少服务器数量。

如果你在 `webserver-cluster` 模块中定义了计划操作，它同时既适用于预发布环境，又适用于生产环境。因为你不需要在预发布环境中进行缩放，所以你可以在生产环境的配置文件中定义自动缩放（在第 5 章，你将看到如何在定义资源时使用条件变量，这样做可以将计划操作的定义移至 `webserver-cluster` 模块中）。

要定义计划的操作，请将以下两个关于 `aws_autoscaling_schedule` 的资源添加到 `prod/services/webserver-cluster/main.tf` 文件中。

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
```

```

scheduled_action_name = "scale-out-during-business-hours"
min_size            = 2
max_size            = 10
desired_capacity    = 10
recurrence          = "0 9 * * *"
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
  scheduled_action_name = "scale-in-at-night"
  min_size            = 2
  max_size            = 10
  desired_capacity    = 2
  recurrence          = "0 17 * * *"
}

```

该代码设置了每天早上把第 1 个 `aws_autoscaling_schedule` 资源中的服务器数量增加到 10 个 (`recurrence` 参数使用 cron 语法, 因此 "`0 9 * * *`" 表示每天上午 9 点) 和每天晚上把第 2 个 `aws_autoscaling_schedule` 资源中的服务器数量减少 ("`0 17 * * *`" 表示每天下午 5 点)。然而这两种用法, `aws_autoscaling_schedule` 都缺少一个必要的参数——`autoscaling_group_name`, 也就是特定的 ASG 名称。ASG 本身是在 `webserver-cluster` 模块内定义的, 那么如何访问它的名称呢? 在 Ruby 之类的通用编程语言中, 函数可以通过返回值实现。

```

def example_function(param1, param2)
  return "Hello, #[param1] #[param2]"
end

# 代码中的其他位置
return_value = example_function("foo", "bar")

```

在 Terraform 中, 使用已知的机制——输出变量, 模块也可以返回值。你可以按如下所示, 在 `modules/services/webserver-cluster/outputs.tf` 文件中将 ASG 名称添加为输出变量。

```

output "asg_name" {
  value      = aws_autoscaling_group.example.name
  description = "The name of the Auto Scaling Group"
}

```

你可以使用以下语法访问模块输出变量。

```
module.<MODULE_NAME>.<OUTPUT_NAME>
```

例如：

```
module.frontend.asg_name
```

在 `prod/services/webserver-cluster/main.tf` 文件中，你可以在每个 `aws_autoscaling_schedule` 资源中使用这个语法来设置 `autoscaling_group_name` 参数。

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
    scheduled_action_name = "scale-out-during-business-hours"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 10
    recurrence            = "0 9 * * *"

    autoscaling_group_name = module.webserver_cluster.asg_name
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
    scheduled_action_name = "scale-in-at-night"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 2
    recurrence            = "0 17 * * *"

    autoscaling_group_name = module.webserver_cluster.asg_name
}
```

通过提供另一个 `webserver-cluster` 模块的输出变量——ALB 的 DNS 名称，就可以知道应该使用哪个 URL 来测试集群部署。为此，让我们再添加一个输出变量。

```
output "alb_dns_name" {
    value      = aws_lb.example.dns_name
    description = "The domain name of the load balancer"
}
```

然后，你可以在 `stage/services/webserver-cluster/outputs.tf` 和 `prod/services/webserver-cluster/outputs.tf` 中，通过“直接传递”的方式配置输出变量。

```
output "alb_dns_name" {
    value      = module.webserver_cluster.alb_dns_name
    description = "The domain name of the load balancer"
}
```

到此为止，Web 服务器集群几乎已经可以部署了。在部署之前让我们考虑一些小的陷阱。

## 模块中的陷阱

在创建模块时，请注意以下陷阱。

- 文件路径
- 内联块

### 文件路径

在第 3 章中，我们已经将 Web 服务器集群的 User Data 脚本移动到了外部文件 `user-data.sh` 中，并使用内置函数 `file` 从磁盘中读取该文件。`file` 函数的不足之处在于，使用的文件路径必须是相对路径（因为你可能在许多不同的计算机上运行相同的 Terraform 代码），但是相对路径的基准是什么呢？

默认情况下，Terraform 以当前工作目录的路径来解析相对路径。如果你使用的包含 `file` 函数的 Terraform 配置文件恰恰在你运行 `terraform apply` 命令的同一目录下，脚本是可以正常工作的。也就是说，多数情况需要你将 `file` 函数存放在根模块中。但如果 `file` 函数出现在其他单独文件夹下的模块中，功能将无法正常工作。

要解决此问题，可以使用被称为路径引用（*path reference*）的表达式，其形式为 `path.<TYPE>`。Terraform 支持以下类型的路径引用。

#### `path.module`

返回定义表达式的模块的文件系统路径。

#### `path.root`

返回根模块的文件系统路径。

#### `path.cwd`

返回当前工作目录的文件系统路径。在正常使用 Terraform 时，它与 `path.root` 相同，但是 Terraform 的某些高级用法是从根模块目录以外的目录中运行它的，从而导致这些路径是不同的。

之前实例中的用户数据的脚本，需要使用从模块本身开始的相对路径进行引用，所以在 *modules/services/webservercluster/main.tf* 文件中的 `template_file` 数据源中，应该使用 `path.module` 表达式。

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port      = data.terraform_remote_state.db.outputs.port
  }
}
```

## 内联块

在配置某些 Terraform 资源时，你可以灵活地将其定义为内联块或单独的资源。但在创建模块时，你应该始终使用单独的资源。

例如，`aws_security_group` 资源允许你通过内联块定义入口和出口规则，如在 `webserver-cluster` 模块（*modules/services/webserver-cluster/main.tf*）中所见到的。

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port   = local.any_port
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

你应该使用完全独立的 `aws_security_group_rule` 资源改写模块，来配置相同的入口和出口规则（请确保同时修改模块中的两个安全组）。

```
resource "aws_security_group" "alb" {
    name          = "${var.cluster_name}-alb"
}

resource "aws_security_group_rule" "allow_http_inbound" {
    type          = "ingress"
    security_group_id = aws_security_group.alb.id

    from_port      = local.http_port
    to_port        = local.http_port
    protocol       = local.tcp_protocol
    cidr_blocks   = local.all_ips
}

resource "aws_security_group_rule" "allow_all_outbound" {
    type          = "egress"
    security_group_id = aws_security_group.alb.id

    from_port      = local.any_port
    to_port        = local.any_port
    protocol       = local.any_protocol
    cidr_blocks   = local.all_ips
}
```

如果你尝试混合使用内联块和独立资源，会因为路由规则冲突和互相覆盖而出现错误。因此必须使用二者之一。在创建模块时由于这个限制，应始终尝试使用单独的资源而不是内联块。否则，模块将缺乏灵活性和可配置性。

例如，将 `webserver-cluster` 模块中所有入口和出口规则定义为单独的 `aws_security_group_rule` 资源，可以使模块具有足够的灵活性，允许用户在模块外添加自定义规则。要做到这一点，需要在 `modules/services/webserver-cluster/outputs.tf` 文件中将 `aws_security_group` 的 ID 导出为输出变量。

```
output "alb_security_group_id" {
    value          = aws_security_group.alb.id
    description    = "The ID of the Security Group attached to the load balancer"
}
```

现在，设想在预发布环境中，需要开放一个额外的端口用于测试。用户可以通过将

`aws_security_group_rule` 资源添加到 `stage/services/webserver-cluster/main.tf` 文件中来轻松地实现这一点。

```
resource "aws_security_group_rule" "allow_testing_inbound" {
  type        = "ingress"
  security_group_id = module.webserver_cluster.alb_security_group_id

  from_port      = 12345
  to_port        = 12345
  protocol       = "tcp"
  cidr_blocks    = ["0.0.0.0/0"]
}
```

如果之前定义过任何关于入口或出口规则的内联块，此代码将会出错。请注意，以下 Terraform 资源也存在相同的内联块问题。

- `aws_security_group` 和 `aws_security_group_rule`
- `aws_route_table` 和 `aws_route`
- `aws_network_acl` 和 `aws_network_acl_rule`

至此，你终于可以在预发布和生产环境中部署 Web 服务器集群了。运行 `terraform apply` 命令后，两个单独副本的基础设施可以同时存在。



### 网络隔离

本章中的示例创建的两个环境，不仅在 Terraform 代码层面是完全相互隔离的，而且具有独立的负载均衡器、服务器和数据库，但是它们在网络层并不相互隔离。为了简化本书中的所有示例，所有资源都部署到同一虚拟私有云（VPC）中。这意味着预发布环境中的服务器可以与生产环境中的服务器通信，反之亦然。在实际使用中，在同一个 VPC 中同时运行两个环境会给你带来多个风险。首先，一个环境中的错误可能会影响另一个环境。例如，如果在预发布环境中错误地修改了路由表的配置，那么生产中的所有路由也会受到影响。其次，如果攻击者可以访问到其中一个环境，那么他们也可以访问另一个环境。如果在预发布环境进行的临时修改意外地暴露了一个端口，那么闯入的任何一个黑客不仅可以访问预发布环境中的测试数据，还可以访问生产环境中的数据。

因此，除非是简单的示例和实验，用户都应该在独立的 VPC 中运行每个环境。

实际上，为了实现进一步的安全性，甚至应该在完全独立的 AWS 账户中运行每个环境。

## 模块版本控制

如果预发布环境和生产环境都指向同一文件夹下的模块，则在该文件夹中进行的更改将会在下一个部署中同时影响两个环境。这种耦合性，使得在不影响生产环境的情况下，测试预发布环境变得十分困难。一种更好的方法是使用版本化的模块 (*versioned modules*)，这样可以在预发布环境中使用一个版本（如 v0.0.2），在生产环境中使用另一个版本（如 v0.0.1），如图 4-5 所示，在不同的环境中使用模块的不同版本。



图 4-5：在不同的环境中使用模块的不同版本

到目前为止，在所有示例中，每当使用模块时，都将模块的 `source` 参数设置为本地文件路径。其实除文件路径外，Terraform 还支持其他类型的模块源，例如 Git URL、Mercurial URL 和任意的 HTTP URL。<sup>1</sup> 创建版本化模块的最简单方法是将模块代码放在单独的 Git 存储库中，并设置 `source` 参数为该存储库的 URL。这意味着你的 Terraform 代码将（至少）分散在两个存储库中。

#### *modules*

这个存储库定义可重用模块。将每个模块视为定义基础设施特定部分的“蓝图”。

#### *live*

这个存储库定义在每种环境（预发布、生产、`mgmt` 等）中运行的基础设施。可以将其视为基于 `modules` 存储库中的“蓝图”所建设的房屋。

现在，更新后的 Terraform 代码文件夹结构看起来如图 4-6 所示，这是具有多个存储库的文件布局。

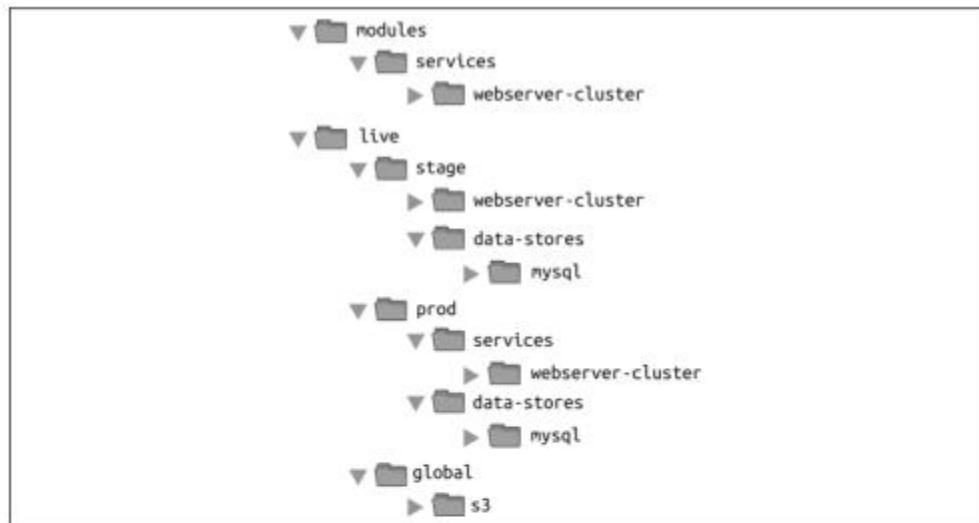


图 4-6：具有多个存储库的文件布局

<sup>1</sup> 有关源 URL 的完整详细信息，请参阅参考资料第 4 章[1]。

要配置此文件夹结构，首先需要将 *stage*、*prod* 和 *global* 文件夹移到一个名为 *live* 的文件夹中。接下来，将 *live* 和 *modules* 文件夹配置为独立的 Git 存储库。以下是将 *modules* 文件夹配置为 Git 存储库的示例。

```
$ cd modules  
$ git init  
$ git add .  
$ git commit -m "Initial commit of modules repo"  
$ git remote add origin "(URL OF REMOTE GIT REPOSITORY)"  
$ git push origin master
```

你可以向 *modules* 存储库添加标签作为版本号。如果你使用的是 GitHub，则可以使用 GitHub UI 创建一个 release（见参考资料第 4 章[2]），它同时将创建标签。如果你不使用 GitHub，则可以使用 Git CLI 创建标签。

```
$ git tag -a "v0.0.1" -m "First release of webserver-cluster module"  
$ git push --follow-tags
```

现在，你可以将预发布环境模块和生产环境模块中的 *source* 参数指向不同的 Git URL，实现模块的版本控制了。如果 *modules* 存储库位于 GitHub 存储库 *github.com/foo/modules* 中，以下是 *live/stage/services/webserver-cluster/main.tf* 文件中 *source* 参数的格式（请注意以下 Git URL 中的双斜杠是必需的）。

```
module "webserver_cluster" {  
  source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"  
  
  cluster_name      = "webservers-stage"  
  db_remote_state_bucket= "(YOUR_BUCKET_NAME)"  
  db_remote_state_key  = "stage/data-stores/mysql/terraform.tfstate"  
  
  instance_type     = "t2.micro"  
  min_size          = 2  
  max_size          = 2  
}
```

如果你只想体验模块版本化，并不想创建 Git 存储库，则可以通过以下代码来使用本书在 GitHub 存储库中的代码示例（见参考资料第 4 章[3]）的模块（我不得不分解 URL 以使其适合本书的排版样式，但所有内容都应在同一行中）。

```
source = "github.com/brikis98/terraform-up-and-running-code//  
code/terraform/04-terraform-module/module-example/modules/  
services/webserver-cluster?ref=v0.1.0"
```

`ref` 参数允许为指向特定的 Git 提交的 `sha1` 哈希值、分支名称或特定的 Git 标签（如本示例）。我通常建议使用 Git 标签作为模块的版本号。因为分支名称不稳定，总是会在分支上获得最新的提交，因此每次你运行 `init` 命令时都有可能得到新的变更。而 Git 提交的 `sha1` 哈希值不是很人性化。Git 标签与提交一样稳定（实际上标签只是指向提交的指针），但是它有更友好的名称。

一种特别有用的标签命名方案是语义版本控制（见参考资料第 4 章[4]），其版本格式为：`MAJOR.MINOR.PATCH`（如 `1.0.4`），下面是如何增加每个部分版本号的特定规则。

- `MAJOR` 版本，当进行了不兼容的 API 更改时
- `MINOR` 版本，当以向后兼容的方式添加新功能时
- `PATCH` 版本，当进行向后兼容的错误修复时

语义版本控制使模块用户能够更准确地理解模块的更改内容，以及模块的升级难度。

将 Terraform 的模块代码更新为使用版本化的 URL 后，需要通过重新运行 `terraform init` 命令，指示 Terraform 下载相关模块代码。

```
$ terraform init  
Initializing modules...  
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v0.1.0  
for webserver_cluster...
```

(...)

这次，你可以看到 Terraform 从 Git 存储库而不是本地文件系统下载模块代码。下载模块代码后，你可以照常运行 `apply` 命令。



### 私有 Git 存储库

如果你将 Terraform 模块置于私有 Git 存储库中，并将这个存储库配置为一个模块的 `source`，则需要设置验证方式，以便 Terraform 访问 Git 存储库。建议使用 SSH 身份验证方式，这样就不需要在代码中存储密钥了。通过 SSH 身份

验证，每个开发人员都可以创建一对 SSH 密钥，将公钥与自己的 Git 用户相关联，将私钥添加到 `ssh-agent` 中，这样在使用 SSH source URL 时，Terraform 将自动使用该密钥进行身份验证。<sup>1</sup>

`source` URL 应该是以下形式的。

```
git@github.com:<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>
```

例如

```
git@github.com:acme/modules.git//example?ref=v0.1.2
```

可以通过终端下的 `git clone` 命令验证你的 URL 的正确性。

```
$ git clone git@github.com:<OWNER>/<REPO>.git
```

如果该命令能成功执行，那么 Terraform 也应该能够使用这个私有 Git 存储库。

现在，你已经在使用版本控制下的模块了。让我们逐步介绍如何进行更改。假设你对 `webserver-cluster` 模块进行了一些更改后，想要在预发布环境中进行测试。首先，提交变化到 `modules` 存储库中。

```
$ cd modules  
$ git add .  
$ git commit -m "Made some changes to webserver-cluster"  
$ git push origin master
```

接下来，你需要在 `modules` 存储库中创建一个新的标签。

```
$ git tag -a "v0.0.2" -m "Second release of webserver-cluster"  
$ git push --follow-tags
```

现在，你可以通过仅仅更新预发布环境中 (`live/stage/services/webserver-cluster/main.tf`) 的 `source` URL 来使用这个新版本。

```
module "webserver_cluster" {  
    source = "git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.2"
```

---

<sup>1</sup> 有关使用 SSH 密钥的指南，请参见参考资料第 4 章 [5]。

```
cluster_name          = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"

instance_type         = "t2.micro"
min_size              = 2
max_size              = 2
}

]
```

在生产环境中 (*live/prod/services/webserver-cluster/main.tf*)，可以继续运行 v0.0.1。

```
module "webserver_cluster" {
  source = "git@github.com:foo/modules.git//webserver-cluster?ref=v0.0.1"

  cluster_name          = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"

  instance_type         = "m4.large"
  min_size              = 2
  max_size              = 10
}
```

当 v0.0.2 版本通过了在预发布环境中的全面测试和验证后，就可以更新生产环境了。但是，如果在 v0.0.2 版本中发现了错误，那也没什么大不了的，因为它对生产环境的实际用户没有影响。修复错误后，提交新版本，然后再次重复整个过程，直到得到一个足够稳定的产品发布版本为止。



### 模块开发

当部署针对共享环境(如预发布环境或生产环境)时，模块版本是非常有用的。但是当你只是在自己的计算机上进行测试时，使用本地文件路径可以更快地进行迭代，因为当在模块文件夹中完成修改后，可以立即在 `live` 文件夹中重新运行 `plan` 或 `apply` 命令来进行部署，这省略了每次提交代码并发布新版本的步骤。由于本书的目的是帮助读者尽快地学习和实验 Terraform，因此后续的代码示例将继续使用模块的本地文件路径。

## 小结

通过将基础设施代码定义为模块，可以将软件工程的最佳实践应用于基础设施代码的开发过程。可以通过代码评审和自动测试来验证模块的每次更改；可以为每个模块创建符合语意版本规范的发布；可以在不同的环境中安全地测试模块的不同版本，如果遇到问题，可以恢复到以前的版本。

所有这些最佳实践，都可以快速而可靠地构建基础设施。因为，这些能够被开发人员重用的基础设施代码是经过充分的验证、测试和记录的。例如，你可以创建一个规范的模块，该模块定义如何部署单个微服务；包括如何运行集群、如何根据负载扩展集群规模，以及如何在集群中分配流量请求。每个团队仅需几行代码就可以使用这个模块来管理自己的微服务。

为了使这样的模块适用于多个团队，该模块中的 Terraform 代码必须足够灵活且可以配置。例如，一个团队可能需要使用模块来部署其无负载均衡器的单个实例的微服务，而另一个团队可能希望自己的微服务运行在十几个实例之上，并使用负载均衡器分配流量。如何在 Terraform 中实现条件判断？有没有办法做 for 循环？有没有一种方法可以在不停机的情况下使用 Terraform 对微服务进行更改？我们将在第 5 章对这些高级的 Terraform 语法进行讨论。

# Terraform 技巧和窍门：循环、 if 表达式、部署和陷阱

Terraform 是一种声明性语言。如第 1 章中所述，使用声明性语言的 IaC 工具，可以提供比过程性语言更为准确的实际部署视图，推理起来更容易，代码库体积也更小。但是使用声明性语言，在某些类型的任务方面会遇到困难。

例如，声明性语言通常没有 `for` 循环，如何在避免使用复制/粘贴的情况下，重复一段逻辑（如创建多个相似的资源）呢？如果声明性语言不支持 `if` 表达式，那又如何有条件地配置资源呢？例如，创建一个 Terraform 模块，该模块可以有条件地为某些用户创建某些资源。最后，又如何使用声明性语言表达一个固有的过程性想法，例如零停机部署？

幸运的是，Terraform 提供了一些原语：`count` 元参数、`for_each` 和 `for` 表达式、一个被称为 `create_before_destroy` 的生命周期块、三元运算符及大量函数。所有这些功能，允许用户执行某些类型的循环、`if` 表达式和零停机部署。以下是本章将要介绍的主题。

- 循环
- 有条件的判断
- 零停机部署
- Terraform 陷阱

# 循环

Terraform 提供了几种不同的循环结构，不同的结构适用于不同的场合。

- `count` 参数，适用于对资源进行循环
- `for_each` 表达式，适用于对循环资源和资源中的内联块进行循环
- `for` 表达式，适用于对列表和映射进行循环
- `for` 字符串指令，适用于对字符串中的列表和映射进行循环

让我们逐一进行学习。

## 使用 `count` 参数进行循环

在第 2 章中，我们已经通过操作 AWS 控制台，创建了一个 AWS Identity and Access Management (IAM) 用户。通过该用户，可以使用 Terraform 创建和管理所有其他的 IAM 用户。考虑以下 Terraform 代码，它应该存放在 `live/global/iam/main.tf` 文件中。

```
provider "aws" {
  region = "us-east-2"
}

resource "aws_iam_user" "example" {
  name = "neo"
}
```

这段代码使用 `aws_iam_user` 资源创建一个新的 IAM 用户。如果需要创建 3 个 IAM 用户该怎么办？在通用编程语言中，会使用 `for` 循环。

```
# 这只是伪代码，它实际上在 Terraform 中无法工作
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo"
  }
}
```

Terraform 没有 `for` 循环或其他内置传统过程逻辑，因此这种语法将不起作用。但是，每个 Terraform 资源都有一个叫作 `count` 的元参数。`count` 是 Terraform 最古老、最简单和最有限的迭代构造器：它所做的只是定义要创建的资源副本个数。下面是使用 `count` 参数创建

3个 IAM 用户的方法。

```
resource "aws_iam_user" "example" {
  count   = 3
  name    = "neo"
}
```

这段代码存在的问题是，创建的 3 个 IAM 用户具有相同的名称。因为 AWS 用户名必须是唯一的，所以这将导致错误。如果使用标准的 for 循环，可以在 for 循环中通过调用索引值 *i*，为每个用户提供唯一的名字。

```
# 这只是伪代码，它实际上在 Terraform 中无法工作
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo.${i}"
  }
}
```

要在 Terraform 中完成类似的操作，可以使用 `count.index` 变量，获取循环中每次迭代的索引值。

```
resource "aws_iam_user" "example" {
  count   = 3
  name    = "neo.${count.index}"
}
```

如果对以上代码运行 `plan` 命令，将看到 Terraform 计划创建 3 个 IAM 用户，每个用户拥有不同的名称（“neo.0”、“neo.1”和“neo.2”）。

```
Terraform will perform the following actions:
# aws_iam_user.example [0] 将被创建
+ resource "aws_iam_user" "example" {
  + arn          = (known after apply)
  + force_destroy = false
  + id           = (known after apply)
  + name         = "neo.0"
  + path          = "/"
  + unique_id     = (known after apply)
}

# aws_iam_user.example[1] 将被创建
+ resource "aws_iam_user" "example" {
```

```

+ arn          = (known after apply)
+ force_destroy = false
+ id           = (known after apply)
+ name          = "neo.1"
+ path           = "/"
+ unique_id     = (known after apply)
}

# aws_iam_user.example[2]将被创建
+ resource "aws_iam_user" "example" {
    + arn          = (known after apply)
    + force_destroy = false
    + id           = (known after apply)
    + name          = "neo.2"
    + path           = "/"
    + unique_id     = (known after apply)
}

```

Plan: 3 to add, 0 to change, 0 to destroy.

当然,诸如"neo.0"之类的用户名并不是特别有用。如果进一步把 `count.index` 与 Terraform 的某些内置函数结合在一起使用,可以进一步自定义循环中的每次迭代内容。

例如,通过输入变量(在 `live/global/iam/variables.tf` 文件中)预先定义所有需要创建的 IAM 用户名。

```

variable "user_names" {
  description      = "Create IAM users with these names"
  type             = list(string)
  default          = ["neo", "trinity", "morpheus"]
}

```

如果使用具有循环和数组功能的通用编程语言,可以通过索引值 `i` 在数组 `var.user_names` 中的取值,为每个 IAM 用户设置不同的名称。

```

# 这只是伪代码,它实际上不会在 Terraform 中工作
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = vars.user_names[i]
  }
}

```

在 Terraform 中,可以通过使用 `count` 和以下函数来完成相同的操作。

## 数组查找语法

在 Terraform 中查找数组成员的语法与其他大多数编程语言相似。

```
ARRAY[<INDEX>]
```

例如，这是在 `var.user_names` 中查找索引值为 1 的元素的方式。

```
var.user_names[1]
```

## *length* 函数

Terraform 有一个名为 `length` 的内置函数，其调用语法如下。

```
length (<ARRAY>)
```

你可能已经猜到，`length` 函数返回 `ARRAY` 列表中的成员数目。它也适用于字符串和映射。

把这些放在一起，得到如下代码。

```
resource "aws_iam_user" "example" {
  count  = length(var.user_names)
  name   = var.user_names[count.index]
}
```

现在，运行 `plan` 命令，将看到 Terraform 计划创建 3 个 IAM 用户，每个用户都拥有唯一的名称。

```
Terraform will perform the following actions:
```

```
# aws_iam_user.example [0] 将被创建
+ resource "aws_iam_user" "example" {
  + arn          = (known after apply)
  + force_destroy = false
  + id           = (known after apply)
  + name         = "neo"
  + path          = "/"
  + unique_id     = (known after apply)
}

# aws_iam_user.example [1] 将被创建
+ resource "aws_iam_user" "example" {
  + arn          = (known after apply)
```

```
+ force_destroy = false
+ id            = (known after apply)
+ name          = "trinity"
+ path          = "/"
+ unique_id    = (known after apply)
}

# aws_iam_user.example [2]将被创建
+ resource "aws_iam_user" "example" {
  + arn      = (known after apply)
  + force_destroy = false
  + id      = (known after apply)
  + name    = "morpheus"
  + path    = "/"
  + unique_id = (known after apply)
}
```

Plan: 3 to add, 0 to change, 0 to destroy.

请注意，对资源使用 count 参数后，它不再是一个单一的资源，而是变成了一个资源的数组。因为 `aws_iam_user.example` 现在是一个 IAM 用户的数组，不能再使用标准语法从该资源读取属性 (`<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>`)，而是要通过使用相应的数组查询语法，在数组中指定索引值。

`<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE`

例如，如果要将其中一个 IAM 用户的 Amazon 资源名称（ARN）作为输出变量，需要执行以下操作。

```
output "neo_arn" {
  value      = aws_iam_user.example[0].arn
  description = "The ARN for user Neo"
}
```

如果要将所有 IAM 用户的 ARN 输出，则需要使用 *splat* 表达式 “\*” 代替索引值。

```
output "all_arns" {
  value      = aws_iam_user.example[*].arn
  description = "The ARNs for all users"
}
```

当运行 `apply` 命令时, `neo_arn` 输出变量仅包含 Neo 的 ARN, 而 `all_arns` 输出变量将包含所有 ARN 的列表。

```
$ terraform apply  
(...)  
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
neo_arn = arn:aws:iam::123456789012:user/neo  
all_arns = [  
    "arn:aws:iam::123456789012:user/neo",  
    "arn:aws:iam::123456789012:user/trinity",  
    "arn:aws:iam::123456789012:user/morpheus",  
]
```

不幸的是, `count` 参数存在两个问题, 这大大降低了它的实用性。首先, 尽管可以使用 `count` 遍历整个资源, 但是不能在资源内部使用 `count` 遍历内联块。内联块是通过以下格式在资源内设置参数的。

```
resource "xxx" "yyy" {  
    <NAME> {  
        [CONFIG...]  
    }  
}
```

其中 `NAME` 是内联块的名称 (如 `tag`), `CONFIG` 包含一个或多个特定于该内联块的参数 (如 `key` 和 `value`)。举例说明, 请回忆 `aws_autoscaling_group` 资源中的标签是如何设置的。

```
resource "aws_autoscaling_group" "example" {  
    launch_configuration = aws_launch_configuration.example.name  
    vpc_zone_identifier = data.aws_subnet_ids.default.ids  
    target_group_arns   = [aws_lb_target_group.asg.arn]  
    health_check_type   = "ELB"  
  
    min_size = var.min_size  
    max_size = var.max_size  
  
    tag {  
        key          = "Name"  
    }  
}
```

```
    value          = var.cluster_name
    propagate_at_launch = true
}
}
```

对于每个 `tag`, 你都需要创建一个新的内联块, 包含 `key`、`value` 和 `propagate_at_launch` 参数。上面的代码对单个标签进行静态编码, 但是你可能希望用户能够使用自定义标签并通过 `count` 参数来遍历这些标签, 动态生成内联 `tag` 代码块, 但不幸的是在内联块中不支持使用 `count` 参数。

其次, `count` 的另一个使用限制是, 当你尝试变更代码时会发生问题。让我们看一下之前创建的 IAM 用户列表。

```
variable "user_names" {
  description      = "Create IAM users with these names"
  type            = list(string)
  default         = ["neo", "trinity", "morpheus"]
}
```

设想从此列表中删除用户 `trinity`, 运行 `terraform plan` 时会发生什么?

```
$ terraform plan

(...)

Terraform will perform the following actions:
# aws_iam_user.example[1] 将被更新
~ resource "aws_iam_user" "example" {
  id           = "trinity"
  ~ name        = "trinity" -> "morpheus"
}

# aws_iam_user.example[2] 将被销毁
- resource "aws_iam_user" "example" {
  - id           = "morpheus" -> null
  - name         = "morpheus" -> null
}
```

```
Plan: 0 to add, 1 to change, 1 to destroy.
```

从输出可以看出, 这里不仅仅删除了名为 `trinity` 的 IAM 用户! 实际上, Terraform 计划将 `trinity` IAM 用户重命名为 `morpheus`, 并删除原始的 `morpheus` 用户。这是怎么回事呢?

在资源上使用 `count` 参数时，该资源将变成资源的列表或数组。不幸的是，Terraform 通过资源在数组中的位置（索引值）标识每个资源。也就是说，第一次运行 `apply` 命令后，生成了 3 个用户，这些 IAM 用户在 Terraform 的内部表示类似如下这样。

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
aws_iam_user.example[2]: morpheus
```

当你从数组的中间删除一个项目时，它后面的所有项目都前移一位，因此在第二次运行 `plan` 时仅有两个名称，Terraform 的内部表示将如下所示。

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus
```

注意用户 `morpheus` 如何从索引 2 移到索引 1 的位置。因为 Terraform 把索引值视为资源的识别标志，所以对于 Terraform 来说，这种更改大致可以翻译为：将索引 1 的存储重命名为 `morpheus`，并删除索引 2。换句话说，每次在使用 `count` 创建资源列表时，如果从列表中间删除一个项目，Terraform 会删除该项目之后的每个资源，然后重新创建这些资源。虽然最终结果可能是你所要求的（即两个名为 `morpheus` 和 `neo` 的 IAM 用户），但是删除和修改资源的过程并非你所期望的。

为了解决这两个限制，Terraform 0.12 引入了 `for_each` 表达式。

### 使用 `for_each` 表达式循环

`for_each` 表达式可以循环遍历列表、集合和映射，用于创建整个资源的多个副本，或资源内的内联块的多个副本。让我们首先了解如何使用 `for_each` 创建资源的多个副本。语法如下。

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  for_each = <COLLECTION>

  [CONFIG ...]
}
```

其中 `PROVIDER` 是提供商的名称（如 `aws`），`TYPE` 是要在该提供商中创建的资源的类型（如 `instance`），`NAME` 是该资源在整个 Terraform 代码范围内的引用标识符（如 `my_instance`），

`COLLECTION` 是一个可供循环的集合或映射(在资源上使用 `for_each` 时不支持列表类型),`CONFIG` 由一个或多个特定于该资源的参数组成。在 `CONFIG` 中, 可以使用 `each.key` 和 `each.value` 对象访问 `COLLECTION` 中当前项目的键和值。

例如, 下面是如何使用 `for_each` 创建 3 个与之前相同的 IAM 用户的。

```
resource "aws_iam_user" "example" {
  for_each      = toset(var.user_names)
  name         = each.value
}
```

注意这里使用 `toset` 函数将 `var.user_names` 从列表转换为集合, 这是因为 `for_each` 在用于资源时仅支持集合和映射。当 `for_each` 循环此集合时, 通过 `each.value` 可以获取每个用户名。用户名也可以通过 `each.key` 取得, 尽管通常只有在使用映射的键值对时, 才会用到 `each.key` 对象。

在资源上使用 `for_each` 后, 资源将变成一个映射, 而不再是一个单一资源(也不是在 `count` 中的资源数组)。要了解其含义, 请删除原始的 `all_arns` 和 `neo_arn` 输出变量, 然后添加一个新的 `all_users` 输出变量。

```
output "all_users" {
  value = aws_iam_user.example
}
```

下面是当运行 `terraform apply` 命令时产生的输出。

```
$ terraform apply
(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

all_users = {
  "morpheus" = {
    "arn"          = "arn:aws:iam::123456789012:user/morpheus"
    "force_destroy" = false
    "id"           = "morpheus"
    "name"         = "morpheus"
```

```

    "path"          = "/"
    "tags"         = {}
}
"neo" = {
    "arn"          = "arn:aws:iam::123456789012:user/neo"
    "force_destroy" = false
    "id"           = "neo"
    "name"         = "neo"
    "path"          = "/"
    "tags"         = {}
}
"trinity" = {
    "arn"          = "arn:aws:iam::123456789012:user/trinity"
    "force_destroy" = false
    "id"           = "trinity"
    "name"         = "trinity"
    "path"          = "/"
    "tags"         = {}
}
}
]

```

你可以看到 Terraform 创建了 3 个 IAM 用户，并且 `all_users` 输出变量包含一个映射，其中映射的键是 `for_each` 中的键值（在本例中为用户名），映射的值是该资源的所有输出。如果要得到和之前 `all_arns` 一样的输出变量，则需要进行一些额外的工作。使用内置函数 `values`（从映射中返回映射的值）和 `splat` 表达式来提取 ARN 值。

```

output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}

```

这是你所期望的输出结果。

```

$ terraform apply
(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

all_arns = [
  "arn:aws:iam::123456789012:user/morpheus",
  "arn:aws:iam::123456789012:user/neo",

```

```
"arn:aws:iam::123456789012:user/trinity",
}]
```

通过 `for_each` 将得到一个资源的映射，与通过 `count` 所得到资源的数组有所不同，这是个非常重要的区别。因为在映射类型中，可以安全地进行删除操作。例如，如果从 `var.user_names` 列表中再次删除 `trinity` 用户，并运行 `terraform plan` 命令，会看到以下内容。

```
$ terraform plan

Terraform will perform the following actions:

# aws_iam_user.example["trinity"] 将要被删除
- resource "aws_iam_user" "example" {
    - arn      = "arn:aws:iam::123456789012:user/trinity" -> null
    - name     = "trinity" -> null
}

Plan: 0 to add, 0 to change, 1 to destroy.
```

这正是我们想得到的！现在只删除了需要被删除的资源，而不会移动其他资源。这就是为什么应该优先使用 `for_each` 而不是 `count` 来创建多个资源副本的原因。

现在让我们将注意力转移到 `for_each` 的另一个优点：它可以在资源中创建多个内联块。例如，在 `webserver-cluster` 模块中可以使用 `for_each` 为 ASG 动态生成 `tag` 的内联块。首先，要允许用户自定义标签，请在 `modules/services/webserver-cluster/variables.tf` 文件中添加一个名为 `custom_tags` 的映射类型输入变量。

```
variable "custom_tags" {
  description = "Custom tags to set on the Instances in the ASG"
  type        = map(string)
  default     = {}
}
```

接下来，在生产环境中的 `live/prod/services/webserver-cluster/main.tf` 文件中，设置一些自定义标签，如下所示。

```
module "webserver_cluster" {
  source = "../../../../../modules/services/webserver-cluster"
```

```

cluster_name          = "webservers-prod"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key    = "prod/data-stores/mysql/terraform.tfstate"

instance_type         = "m4.large"
min_size              = 2
max_size              = 10
enable_autoscaling    = true

custom_tags           =
  Owner                = "team-foo"
  DeployedBy           = "terraform"
}

}

```

前面的代码设置了几组有用的标签：`Owner` 标签定义了哪个团队拥有此 ASG，`DeployedBy` 标签定义了这个基础设施是使用 Terraform 部署的（表示不应手动修改此基础设施，如“Terraform 陷阱”中“有效的计划也可能会失败”所讨论的）。通常一个比较好的做法是，为团队定义一个标签标准，并通过创建 Terraform 模块，使用代码来强化该标准。

既然你已经定义了标签，那么如何在 `aws_autoscaling_group` 资源上进行实际设置呢？你需要在 `var.custom_tags` 变量上进行 `for` 循环，类似于以下伪代码。

```

resource "aws_autoscaling_group" "example" {
  launch_configuration      = aws_launch_configuration.example.name
  vpc_zone_identifier        = data.aws_subnet_ids.default.ids
  target_group_arns          = [aws_lb_target_group.asg.arn]
  health_check_type          = "ELB"

  min_size                  = var.min_size
  max_size                  = var.max_size

  tag {
    key                      = "Name"
    value                     = var.cluster_name
    propagate_at_launch       = true
  }
}

# 这仅仅是伪代码。它实际上不会在 Terraform 中工作
for (tag in var.custom_tags) {
  tag {

```

```
        key          = tag.key
        value        = tag.value
        propagate_at_launch = true
    }
}
}
```

上面的伪代码是无法正常工作的，但是一个 `for_each` 表达式可以实现这个功能。使用 `for_each` 动态生成内联块的语法如下。

```
dynamic "<VAR_NAME>" {
  for_each = <COLLECTION>

  content {
    [CONFIG...]
  }
}
```

其中 `VAR_NAME` 是将用于存储每个迭代（而不是 `each`）的变量的名称，`COLLECTION` 是循环体的列表或映射，`content` 块是每次迭代动态生成的内容。在 `content` 块范围内，可以通过 `<VAR_NAME>.key` 和 `<VAR_NAME>.value` 访问 `COLLECTION` 中当前循环项目的键和值。请注意，当 `for_each` 作用于列表时，`key` 将是索引值，`value` 将是该索引处列表中的项目，而当 `for_each` 作用于映射时，`key` 和 `value` 将和映射中的 `key-value` 对的取值相对应。

综上所述，下面是在 `aws_autoscaling_group` 资源中使用 `for_each` 动态生成 `tag` 代码块的方法。

```
resource "aws_autoscaling_group" "example" {
  launch_configuration      = aws_launch_configuration.example.name
  vpc_zone_identifier       = data.aws_subnet_ids.default.ids
  target_group_arns         = [aws_lb_target_group.asg.arn]
  health_check_type         = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key          = "Name"
    value        = var.cluster_name
    propagate_at_launch = true
  }
}
```

```
)\n\n    dynamic "tag" {\n        for_each           = var.custom_tags\n\n        content {\n            key              = tag.key\n            value             = tag.value\n            propagate_at_launch = true\n        }\n    }\n}
```

如果现在运行 `terraform apply` 命令，应该会看到类似于下面的计划输出。

```
$ terraform apply\n\nTerraform will perform the following actions:\n\n# aws_autoscaling_group.example 将被更新\n~ resource "aws_autoscaling_group" "example" {\n    (...)\n\n    tag {\n        key              = "Name"\n        propagate_at_launch = true\n        value             = "webservers-prod"\n    }\n\n    + tag {\n        + key              = "Owner"\n        + propagate_at_launch = true\n        + value             = "team-foo"\n    }\n\n    + tag {\n        + key              = "DeployedBy"\n        + propagate_at_launch = true\n        + value             = "terraform"\n    }\n}\n\nPlan: 0 to add, 1 to change, 0 to destroy.\n\nDo you want to perform these actions?
```

Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

输入 yes 来确认部署，在 EC2 Web 控制台中可以看到新添加的标签，动态生成的 Auto Scaling Group 标签如图 5-1 所示。

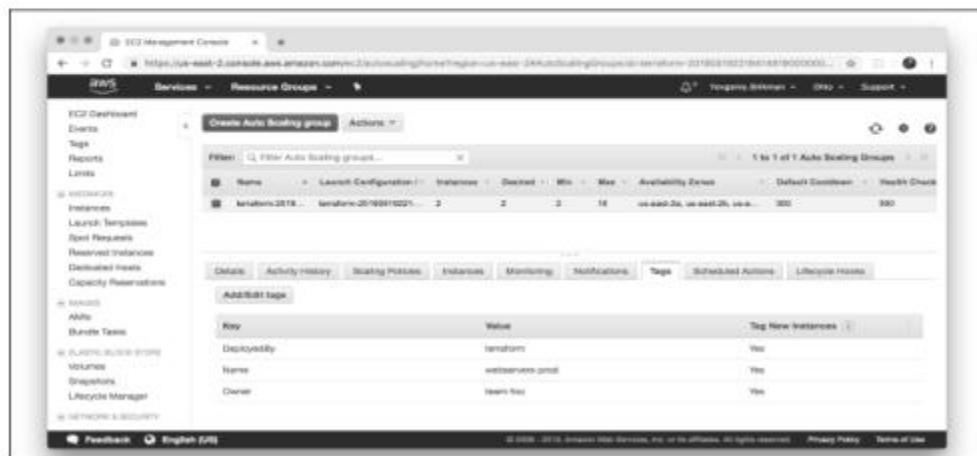


图 5-1：动态生成的 Auto Scaling Group 标签

### 通过 for 表达式进行循环

现在我们已经学习了如何对资源和内联块进行循环，但是如果需要通过循环产生单个值该怎么办？让我们来看一个与 Web 服务器集群不相关的示例。想象一下，有一个包含名称列表的 Terraform 的输入变量。

```
variable "names" {  
    description = "A list of names"  
    type        = list(string)  
    default     = ["neo", "trinity", "morpheus"]  
}
```

该如何将这些名称转换为大写？在通用的编程语言（如 Python）中，你可以编写 for 循环。

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = []
for name in names:
    upper_case_names.append(name.upper())

print upper_case_names

# 打印输出结果为: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python 提供了另一种被称为 *list comprehension* 的语法，用一行代码就可以实现完全相同的操作。

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = [name.upper() for name in names]

print upper_case_names

# 打印输出结果为: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python 还允许你通过指定条件来过滤结果列表。

```
names = ["neo", "trinity", "morpheus"]

short_upper_case_names = [name.upper() for name in names if len(name) < 5]

print short_upper_case_names

# 打印输出结果为: ['NEO']
```

Terraform 通过 *for* 表达式可以实现类似功能(不要和上一节中的 *for\_each* 表达式相混淆)。  
*for* 表达式的基本语法如下。

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

其中 *LIST* 是需要被循环的列表，*ITEM* 是个本地变量名，将被赋予 *LIST* 中每次循环的项目值，而 *OUTPUT* 是一个以某种方式转换 *ITEM* 的表达式。例如，以下 Terraform 代码将 *var.names* 中的名称列表转换为大写。

```
variable "names" {
  description = "A list of names"
```

```
type      = list(string)
default   = ["neo", "trinity", "morpheus"]
}
output "upper_names" {
  value = [for name in var.names : upper(name)]
}
```

如果运行 `terraform apply` 命令，将得到以下输出。

```
$ terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

与 Python 的列表表达式一样，可以通过指定条件来过滤结果列表。

```
variable "names" {
  description = "A list of names"
  type       = list(string)
  default    = ["neo", "trinity", "morpheus"]
}

output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

运行 `terraform apply` 命令会得到如下结果。

```
short_upper_names= [
  "NEO",
]
```

Terraform 的 `for` 表达式还允许使用以下语法对映射对象进行遍历。

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

这里，`MAP` 是要进行循环的映射对象，`KEY` 和 `VALUE` 是局部变量，用来获取 `MAP` 中每次循环的键值对，而 `OUTPUT` 是一个以某种方式转换 `KEY` 和 `VALUE` 的表达式。以下是一个示例。

```
variable "hero_thousand_faces" {
  description  = "map"
  type         = map(string)
  default      = {
    neo        = "hero"
    trinity   = "love interest"
    morpheus  = "mentor"
  }
}
output "bios" {
  value = [for name, role in var.hero_thousand_faces : "${name} is the ${role}"]
}
```

运行 `terraform apply` 命令，会得到如下结果。

```
map_example = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]
```

也可以使用以下语法，通过 `for` 表达式输出一个映射而不是一个列表。

```
#遍历列表
[for <ITEM> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>]

#遍历映射
{for <KEY>, <VALUE> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>}
```

唯一的区别是表达式使用花括号而不是方括号括起来，并且每次迭代不是输出单个值，而是输出一个用箭头隔开的键值对。例如，以下是将映射的键和值转换成大写的方法。

```
variable "hero_thousand_faces" {
  description  = "map"
  type         = map(string)
  default      = {
    neo        = "hero"
    trinity   = "love interest"
    morpheus  = "mentor"
  }
}
output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```

代码运行结果如下。

```
upper_roles = {
    "MORPHEUS" = "MENTOR"
    "NEO"       = "HERO"
    "TRINITY"   = "LOVE INTEREST"
}
```

### 通过 for 字符串指令循环

在本书前面，学习了字符串插值，字符串插值可以在字符串中引用 Terraform 代码。

```
"Hello, ${var.name}"
```

字符串指令(*directive*)允许你使用类似于字符串插值的语法在字符串内使用控制语句(如 for 循环和 if 表达式)，但是字符串指令使用百分号%和花括号{...}代替字符串插值的美元符号\$和花括号{...}。

Terraform 支持两种类型的字符串指令：for 循环和条件指令。在本节中，我们首先将介绍 for 循环，我们将在本章后面介绍条件指令。for 字符串指令使用以下语法。

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

其中 COLLECTION 是一个用于循环的列表或映射，ITEM 是分配给 COLLECTION 中每个元素的局部变量名，BODY 是每次迭代呈现的内容(可以引用 ITEM)，例如如下代码。

```
variable "names" {
    description = "Names to render"
    type        = list(string)
    default     = ["neo", "trinity", "morpheus"]
}

output "for_directive" {
    value = <<EOF
${name}
%{ endfor }
EOF
}
```

运行 terraform apply 命令时，将得到以下输出。

```
$ terraform apply

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

for_directive =
neo

trinity

morphous
```

请注意上面输出中多余的换行符。在字符串指令中可以通过使用标记~来消除所有空格字符（空格和换行符），如果标记出现在字符串指令的开头，就会去除之前的空格，如果标记出现在字符串指令的结尾，就会去除后面的空格。

```
output "for_directive_strip_marker" {
  value = <<EOF
${~ for name in var.names }
  ${name}
${~ endfor }
EOF
}
```

这个新的版本输出如下。

```
for_directive_strip_marker =
neo
trinity
morphous
```

## 有条件的判断

Terraform 不仅提供了几种不同的循环方法，同时也有几种不同的方法来处理条件，每个方法使用的环境略有不同。

**count** 参数

用于资源的条件判断。

## `for_each` 和 `for` 表达式

用于资源和资源中内联块的条件判断。

## `if` 字符串指令

用于字符串中的条件判断。

让我们逐一进行介绍。

## 使用 `count` 参数进行条件判断

之前章节看到的 `count` 参数可以进行基本循环。你可以使用相同的机制来执行基本条件判断。让我们先从 `if` 表达式开始，然后再继续学习 `if-else` 表达式。

### 带有 `count` 参数的 `if` 表达式

在第 4 章中，我们创建了一个 Terraform 模块，该模块可以作为部署 Web 服务器集群的“蓝图”。该模块创建了 Auto Scaling Group (ASG)、Application Load Balancer (ALB)、安全组，以及许多其他资源。

但是，预定操作资源并未在模块中创建。因为只有在生产环境中需要扩展集群的大小，所以可以在生产环境配置 (*live/prod/services/webserver-cluster/main.tf*) 中直接定义 `aws_autoscaling_schedule` 资源。有没有一种方法可以在 `webserver-cluster` 模块中定义 `aws_autoscaling_schedule` 资源，使其有条件地为某些模块的用户创建该资源，而忽略其他用户呢？

让我们试一试。第一步在 `modules/services/webserver-cluster/variables.tf` 文件中添加一个布尔输入变量，通过这个变量指定模块是否启用自动缩放。

```
variable "enable_autoscaling" {
  description      = "If set to true, enable auto scaling"
  type             = bool
}
```

现在，如果是通用的编程语言，可以在一个 `if` 表达式中使用这个输入变量。

```
# 这仅仅是伪代码，它实际上不会在 Terraform 中工作
if var.enable_autoscaling {
  resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
```

```

    scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 10
    recurrence            = "0 9 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
    scheduled_action_name = "${var.cluster_name}-scale-in-at-night"
    min_size              = 2
    max_size              = 10
    desired_capacity      = 2
    recurrence            = "0 17 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
}
}

```

Terraform 不支持 if 表达式，因此这个代码将不起作用。但是，你可以使用 count 参数和利用它的两个属性来完成相同的操作。

- 如果在资源上将 count 设置为 1，可获得该资源的一个副本。如果将 count 设置为 0，则不会创建该资源。
- Terraform 条件表达式的格式为：<CONDITION> ? <TRUE\_VAL> : <FALSE\_VAL>。你可能通过其他编程语言学习了这种三元语法，它将评估 CONDITION 的布尔逻辑，如果结果为 true，则返回 TRUE\_VAL；如果结果为 false，则返回 FALSE\_VAL。

使用这两个特点，可以按以下方式更新 webserver-cluster 模块。

```

resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
    count          = var.enable_autoscaling ? 1 : 0

    scheduled_action_name = "${var.cluster_name}-scale-out-during-business-hours"
    min_size        = 2
    max_size        = 10
    desired_capacity = 10
    recurrence     = "0 9 * * *"
    autoscaling_group_name = aws_autoscaling_group.example.name
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {

```

```
count          = var.enable_autoscaling ? 1 : 0

scheduled_action_name = "${var.cluster_name}-scale-in-at-night"
min_size        = 2
max_size        = 10
desired_capacity = 2
recurrence      = "0 17 * * *"
autoscaling_group_name = aws_autoscaling_group.example.name
}
```

如果 `var.enable_autoscaling` 变量的值为 `true`, 则每个 `aws_autoscaling_schedule` 资源的 `count` 参数都将被设置为 1, 因此每个相应的资源都将被创建。如果 `var.enable_autoscaling` 变量为 `false`, 则每个 `aws_autoscaling_schedule` 资源的 `count` 参数都将被设置为 0, 因此将没有资源会被创建。这正是我们需要的条件逻辑!

现在, 可以在预发布环境的模块调用中 (*live/stage/services/webserver-cluster/main.tf*), 设置 `enable_autoscaling` 为 `false`, 以达到禁用自动缩放功能的目的。

```
module "webserver_cluster" {
  source          = "../../../../../modules/services/webserver-cluster"

  cluster_name    = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"

  instance_type    = "t2.micro"
  min_size         = 2
  max_size         = 2
  enable_autoscaling = false
}
```

同样, 可以在生产环境的模块调用 (*live/prod/services/webserver-cluster/main.tf*) 中, 设置 `enable_autoscaling` 为 `true`, 激活自动缩放功能 (在生产环境中, 请务必同时删除在第 4 章中添加的自定义 `aws_autoscaling_schedule` 资源)。

```
module "webserver_cluster" {
  source          = "../../../../../modules/services/webserver-cluster"

  cluster_name    = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
```

```

db_remote_state_key      = "prod/data-stores/mysql/terraform.tfstate"

instance_type            = "m4.large"
min_size                 = 2
max_size                 = 10
enable_autoscaling       = true

custom_tags               =
  Owner                  = "team-foo"
  DeployedBy              = "terraform"
}

}

```

如果用户将一个明确的布尔值传递到模块中，这种方法效果很好。但是如果布尔值是一个比较复杂的比较结果，如字符串比较，又该怎么做呢？让我们来看一个更复杂的例子。

想象一下，作为 `webserver-cluster` 模块的一部分，需要创建一组 `CloudWatch` 警报。你可以配置一个 `CloudWatch` 警报，当某个指标超过预定阈值时，可以通过不同的机制来发送通知（如电子邮件、短信）。例如，在 `modules/services/webserver-cluster/main.tf` 模块中，使用 `aws_cloudwatch_metric_alarm` 资源来创建警报，如果该集群在 5min 内的平均 CPU 使用率超过 90%，该警报将被触发。

```

resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name          = "${var.cluster_name}-high-cpu-utilization"
  namespace           = "AWS/EC2"
  metric_name         = "CPUUtilization"

  dimensions          =
    AutoScalingGroupName = aws_autoscaling_group.example.name
  }

  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Average"
  threshold           = 90
  unit                = "Percent"
}

```

以上对于 CPU 利用率警报的代码是能够正常工作的。如果要添加另一个在 CPU 积分不足时发出的警报，该怎么处理呢？<sup>1</sup> 以下是一个 CloudWatch 警报，在 Web 服务器集群用光 CPU 积分时就会被触发。

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
    alarm_name          = "${var.cluster_name}-low-cpu-credit-balance"
    namespace           = "AWS/EC2"
    metric_name         = "CPUCreditBalance"

    dimensions = {
        AutoScalingGroupName = aws_autoscaling_group.example.name
    }

    comparison_operator = "LessThanThreshold"
    evaluation_periods = 1
    period              = 300
    statistic            = "Minimum"
    threshold            = 10
    unit                 = "Count"
}
```

美中不足的是，CPU 积分仅适用于 tXXX 实例类型（例如 `t2.micro`、`t2.medium` 等）。较大的实例类型（如 `m4.large`）不使用 CPU 积分，也不报告 `CPUCreditBalance` 指标。因此，如果为这些实例创建 CPU 积分警报，则警报将始终停留在 `INSUFFICIENT_DATA` 状态。有没有一种方法，仅当 `var.instance_type` 变量以字母 t 开头时，才创建警报呢？

可以添加一个新的名为 `var.is_t2_instance` 的布尔输入变量，但是这将和 `var.instance_type` 变量互相重复，当类型发生变化时，你很有可能在更新一个变量时忘记更新另一个。更好的选择是使用条件赋值。

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
    count                = format("% .1s", var.instance_type) == "t" ? 1 : 0

    alarm_name          = "${var.cluster_name}-low-cpu-credit-balance"
    namespace           = "AWS/EC2"
    metric_name         = "CPUCreditBalance"

    dimensions = {
```

<sup>1</sup> 你可以在 EC2 网站上了解 CPU 积分（见参考资料第 5 章[1]）。

```
AutoScalingGroupName = aws_autoscaling_group.example.name
}

comparison_operator      = "LessThanThreshold"
evaluation_periods       = 1
period                  = 300
Conditionals | 155
statistic                = "Minimum"
threshold                = 10
unit                     = "Count"
}
```

报警部分代码与以前相同，只是 count 参数部分稍微复杂了一些。

```
count = format ("%s", var.instance_type) == "t" ? 1 : 0
```

此代码使用 `format` 函数从 `var.instance_type` 变量中提取第 1 个字符，如果该字符为 t（如 `t2.micro`），则设置 `count` 为 1；否则，将 `count` 设置为 0。这样，将只会对真正拥有 `CPUCreditBalance` 指标的实例类型创建警报。

#### 带有 count 参数的 if-else 表达式

现在学习在执行 if 表达式之后，如何使用 if-else 表达式呢？

在本章的前面，我们创建了几个对 EC2 拥有只读访问权限的 IAM 用户。想象一下，如果希望向这些用户之一的 neo 提供 CloudWatch 的访问权限，但允许执行 Terraform 配置的人来确定到底为 neo 分配只读权限，还是读写权限。这是一个人为创造的简化示例，它可以演示一个简单的 if-else 表达式，其中的关键是，代码将执行 if 或 else 分支之一，其余的 Terraform 代码不需要知道具体执行了哪个。

这是一个允许对 CloudWatch 进行只读访问的 IAM 策略。

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name          = "cloudwatch-read-only"
  policy        = data.aws_iam_policy_document.cloudwatch_read_only.json
}
data "aws_iam_policy_document" "cloudwatch_read_only" {
  statement {
    effect      = "Allow"
    actions     = [
      "cloudwatch:Describe*",
      "cloudwatch:Get*",
      "cloudwatch:List*",
      "cloudwatch:Put*",
      "cloudwatch:Update*",
      "cloudwatch:Delete*",
      "cloudwatch:BatchGet*",
      "cloudwatch:BatchPut*",
      "cloudwatch:BatchUpdate*",
      "cloudwatch:BatchDelete*",
      "cloudwatch:PutMetricData"
    ]
  }
}
```

```
    "cloudwatch:Get*",
    "cloudwatch>List*"
]
resources      = ["*"]
}
}
```

这是一个允许对 CloudWatch 进行完全（读取和写入）访问的 IAM 策略。

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name          = "cloudwatch-full-access"
  policy        = data.aws_iam_policy_document.cloudwatch_full_access.json
}

data "aws_iam_policy_document" "cloudwatch_full_access" {
  statement {
    effect      = "Allow"
    actions     = ["cloudwatch:*"]
    resources   = ["*"]
  }
}
```

我们的目标是，基于新的输入变量 `give_neo_cloudwatch_full_access`，将上述 IAM 政策之一分配给用户 `neo`。

```
variable "give_neo_cloudwatch_full_access" {
  description      = "If true, neo gets full access to CloudWatch"
  type            = bool
}
```

如果你使用的是通用编程语言，可以编写如下的 if- else 表达式。

```
# 这只是伪代码，它实际上不会在 Terraform 中工作
if var.give_neo_cloudwatch_full_access {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
    user          = aws_iam_user.example[0].name
    policy_arn    = aws_iam_policy.cloudwatch_full_access.arn
  }
} else {
  resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
    user          = aws_iam_user.example[0].name
    policy_arn    = aws_iam_policy.cloudwatch_read_only.arn
  }
}
```

要让 Terraform 做到这一点，你可以在每个资源的 `count` 参数上使用条件表达式。

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
    count          = var.give_neo_cloudwatch_full_access ? 1 : 0
    user           = aws_iam_user.example[0].name
    policy_arn     = aws_iam_policy.cloudwatch_full_access.arn
}

resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
    count          = var.give_neo_cloudwatch_full_access ? 0 : 1
    user           = aws_iam_user.example[0].name
    policy_arn     = aws_iam_policy.cloudwatch_read_only.arn
}
```

以上代码包含两个 `aws_iam_user_policy_attachment` 资源。第 1 个附加了 CloudWatch 的完全访问权限，具有一个条件表达式，如果 `var.give_neo_cloudwatch_full_access` 变量为 `true`，则表达式结果为 1，否则为 0（这是 `if` 条件子句）。第 2 个附加了 CloudWatch 只读权限，具有完全相反的逻辑，如果 `var.give_neo_cloudwatch_full_access` 变量为 `true`，则表达式结果为 0，否则为 1（这是 `else` 子句）。

如果 Terraform 代码不需要知道实际执行了 `if` 还是 `else` 子句，这个方法效果很好。但是，如果你需要访问 `if` 或 `else` 子句中的资源上的某些输出属性，该怎么办？例如，如果想在 `webserver-cluster` 模块中提供两个不同的 User Data 脚本，并允许用户选择应该执行哪个。目前，`webserver-cluster` 模块通过 `template_file` 数据源，引入 `user-data.sh` 脚本。

```
data "template_file" "user_data" {
    template      = file("${path.module}/user-data.sh")

    vars          = {
        server_port   = var.server_port
        db_address    = data.terraform_remote_state.db.outputs.address
        db_port       = data.terraform_remote_state.db.outputs.port
    }
}
```

目前 `user-data.sh` 脚本的内容是这样的。

```
#!/bin/bash
```

```
cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

假设某些 Web 服务器集群需要使用一个更短的脚本，这被称为 *user-data-new.sh*。

```
#!/bin/bash

echo "Hello, World, v2" > index.html
nohup busybox httpd -f -p ${server_port} &
```

要使用这个新脚本，需要一个新的 `template_file` 数据源。

```
data "template_file" "user_data_new" {
  template = file("${path.module}/user-data-new.sh")

  vars {
    server_port = var.server_port
  }
}
```

如何允许 `webserver-cluster` 模块的用户从这两个 User Data 脚本之中进行选择呢？首先，在 `modules/services/webserver-cluster/variables.tf` 文件中添加一个新的布尔输入变量。

```
variable "enable_new_user_data" {
  description = "If set to true, use the new User Data script"
  type        = bool
}
```

如果使用通用编程语言，则可以在启动配置中添加 `if-else` 表达式，在两个 User Data 的 `template_file` 选项之间进行选择，如下所示。

```
#这只是伪代码，它实际上不会在Terraform中工作
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
  security_groups = [aws_security_group.instance.id]
```

```
    if var.enable_new_user_data {
        user_data      = data.template_file.user_data_new.rendered
    } else {
        user_data      = data.template_file.user_data.rendered
    }
}
```

为了在 Terraform 代码下工作，首先要使用上文中的 if-else 表达式，确保只有一个 template\_file 数据源被实际创建。

```
data "template_file" "user_data" {
    count          = var.enable_new_user_data ? 0 : 1

    template      = file("${path.module}/user-data.sh")

    vars          = {
        server_port = var.server_port
        db_address  = data.terraform_remote_state.db.outputs.address
        db_port     = data.terraform_remote_state.db.outputs.port
    }
}

data "template_file" "user_data_new" {
    count          = var.enable_new_user_data ? 1 : 0

    template      = file("${path.module}/user-data-new.sh")

    vars          = {
        server_port = var.server_port
    }
}
```

如果 var.enable\_new\_user\_data 变量值为 true，data.template\_file.user\_data\_new 将被创建，而 data.template\_file.user\_data 将不会被创建。如果变量值为 false，逻辑就是相反的。接下来需要做的就是将 aws\_launch\_configuration 资源的 user\_data 参数指向实际存在的 template\_file。为此，可以使用另一个条件表达式。

```
resource "aws_launch_configuration" "example" {
    image_id      = "ami-0c55b159cbfafe1f0"
    instance_type = var.instance_type
    security_groups = [aws_security_group.instance.id]
```

```

user_data = (
    length(data.template_file.user_data[*]) > 0
    ? data.template_file.user_data[0].rendered
    : data.template_file.user_data_new[0].rendered
)

# 当 ASG 中使用启动配置时，必须使用以下生命周期设置
# https://www.terraform.io/docs/providers/aws/r/launch\_configuration.html
lifecycle {
    create_before_destroy = true
}
}

```

让我们将上文中 `user_data` 参数逐行进行分析。首先，看一看被评估的布尔条件。

```
length(data.template_file.user_data[*]) > 0
```

注意，两个 `template_file` 数据源都是数组格式的，因为它们都是使用 `count` 参数生成的，因此需要使用数组语法。但是，由于这两个数组之一的长度为 1，另一个数组的长度为 0（数组为空），因此你无法直接访问特定索引（如 `data.template_file.user_data[0]`）。解决方案是使用 `splat` 表达式，该表达式将始终返回一个数组（尽管数组可能为空），并检查该数组的长度。

使用数组的长度，然后我们从下面的表达式中挑选一个。

```
? data.template_file.user_data[0].rendered
: data.template_file.user_data_new[0].rendered
```

Terraform 对条件结果进行懒惰评估，仅当条件为真时，才会对真实表达式（第 1 部分）进行评估，当条件为假时，评估第 2 部分表达式。这样就可以安全地在 `user_data` 和 `user_data_new` 结果上查找索引 0 了，因为我们知道只有非空数组才会被实际评估。

现在，你可以将预发布环境中的 `enable_new_user_data` 参数设置为 `true`，使用新的 User Data 脚本。在 `live/stage/services/webserver-cluster/main.tf` 文件中的代码如下。

```

module "webserver_cluster" {
    source          = "../../../../../modules/services/webserver-cluster"

    cluster_name     = "webservers-stage"
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
    db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"
}
```

```
    instance_type      = "t2.micro"
    min_size          = 2
    max_size          = 2
    enable_autoscaling = false
    enable_new_user_data = true
}
```

在生产环境中，你可以设置 `enable_new_user_data` 参数为 `false`，来继续使用老版本的脚本。*live/prod/services/webserver-cluster/main.tf* 文件中的代码如下。

```
module "webserver_cluster" {
  source           = "../../../../../modules/services/webserver-cluster"

  cluster_name     = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key   = "prod/data-stores/mysql/terraform.tfstate"

  instance_type      = "m4.large"
  min_size          = 2
  max_size          = 10
  enable_autoscaling = true
  enable_new_user_data = false

  custom_tags        =
    Owner            = "team-foo"
    DeployedBy       = "terraform"
}
}
```

使用 `count` 和内置函数来模拟 `if-else` 表达式有点麻烦，但是效果很好，而且从代码中可以看出，它可以隐藏很多复杂的内容，用户可以访问干净、简单的 API。

## 用 `for_each` 和 `for` 表达式实现条件判断

现在我们了解了如何通过 `count` 参数实现资源的条件逻辑，同样也可以通过 `for_each` 表达式实现类似的条件逻辑。如果将一个空集合传递给 `for_each` 表达式，它将产生 0 个资源或 0 个内联块。如果将一个非空集合传入，它将创建一个或多个资源或内联块。唯一的问题是，你该如何判断集合是否为空？

答案是将 `for_each` 表达式与 `for` 表达式结合在一起使用。回想一下 `webserver-cluster` 的模块 *modules/services/webserver-cluster/main.tf* 是如何来设定标签的。

```
dynamic "tag" {
  for_each = var.custom_tags

  content {
    key          = tag.key
    value        = tag.value
    propagate_at_launch = true
  }
}
```

如果变量 `var.custom_tags` 是空的，那么 `for_each` 表达式将没有内容可以进行循环，所以标签被创建。换句话说，你已经在这里处理了一些条件逻辑。可以进一步将 `for_each` 表达式与 `for` 表达式组合如下。

```
dynamic "tag" {
  for_each = {
    for key, value in var.custom_tags:
      key => upper(value)
      if key != "Name"
  }
  content {
    key          = tag.key
    value        = tag.value
    propagate_at_launch = true
  }
}
```

嵌套中的 `for` 表达式用于循环 `var.custom_tags` 变量，将每个 `value` 转换为大写（出于一致性考虑），并且在 `for` 表达式中使用一个条件表达式来过滤排除任何 `key` 值为 `Name` 的结果，这是因为模块已经有了名为 `Name` 的标签。通过 `for` 表达式中对值的过滤，用户可以实现任意条件逻辑。

需要注意的是，在创建多个资源副本时，你可能更倾向于使用 `for_each` 表达式而不是 `count` 参数。但由于在条件逻辑方面，将 `count` 设置为 0 或 1 往往比将 `for_each` 设定为空集合或非空集合更简单。因此，最佳组合是，使用 `count` 有条件地创建资源，使用 `for_each` 表达式处理所有其他类型的循环和条件判断。

使用 if 字符串指令实现条件判断

在本章的前面，我们使用了 `for` 字符串指令在字符串内进行循环。现在让我们看一下另一种类型的字符串指令，其格式如下：

```
%{ if <CONDITION> }<TRUEVAL>%{ endif }%
```

其中 **CONDITION** 是将被进行布尔值评估的任何表达式，**TRUEVAL** 是在 **CONDITION** 为 true 时呈现的表达式。你可以选择包括 **else** 子句，如下所示。

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

其中 FALSEVAL 是当 CONDITION 评估为 false 时呈现的表达式。示例如下。

```
variable "name" {
  description = "A name to render"
  type        = string
}
output "if_else_directive" {
  value = "Hello, ${ var.name != "" ?${var.name} : (unnamed) }${ endif }"
}
```

如果运行 `terraform apply` 命令，同时设置 `name` 变量为 `World`，则会看到以下结果。

```
$ terraform apply -var name="World"

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

  if_else directive = Hello, World
```

如果运行 `terraform apply` 命令，并将 `name` 变量设置为空字符串，则会看到以下结果。

```
$ terraform apply -var name=""  
  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
if else directive = Hello. (unnamed)
```

## 零停机部署

现在模块已经具备了用于部署 Web 服务器集群的简洁的 API，那么下一个重要的问题是，该如何更新集群呢？也就是说，当你更改代码时，如何在整个集群上部署新的 Amazon Machine Image（AMI）？以及如何在不造成用户停机的情况下做到这一点？

首先是将 AMI 作为输入变量公开在 *modules/services/webserver-cluster/variables.tf* 文件中。在实际工作中，这就是你所需要的全部修改，因为实际的 Web 服务器代码将在 AMI 镜像中定义。但是，在本书的简化示例中，所有的 Web 服务器代码实际上都是通过 User Data 脚本来传递的，使用的 AMI 只是标准的 Ubuntu 映像。切换为不同版本的 Ubuntu 并不会为演示带来任何区别。因此，除了新的 AMI 输入变量外，你还需要通过 User Data 脚本，添加一个输入变量来控制单行 HTTP 服务器返回的文本内容。

```
variable "ami" {
  description = "The AMI to run in the cluster"
  default     = "ami-0c55b159cbfafe1f0"
  type        = string
}

variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

本章前面为了练习 if-else 表达式，我们创建了两个 User Data 脚本。下面将其合并为一个，以保持示例简洁。首先，在 *modules/services/webserver-cluster/variables.tf* 文件中，删除输入变量 `enable_new_user_data`。其次，在 *modules/services/webserver-cluster/main.tf* 文件中，删除名为 `user_data_new` 的 `template_file` 资源。然后，在同一文件中，更新另一个名为 `user_data` 的 `template_file` 资源，使其不再使用 `enable_new_user_data` 输入变量，并添加新的 `server_text` 输入变量到 `vars` 代码块中。

```
data "template_file" "user_data" {
  template     = file("${path.module}/user-data.sh")

  vars         = {
```

```
    server_port = var.server_port
    db_address = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
    server_text = var.server_text
}
}
```

现在你需要更新 *modules/services/webserver-cluster/user-data.sh* bash 脚本，在 html 的<h1> 标签中，返回 `server_text` 变量。

```
#!/bin/bash

cat > index.html <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

最后，在 *modules/services/webserver-cluster/main.tf* 文件中找到启动配置，将它的 `user_data` 参数设置为名为 `user_data` 的 `template_file` 资源，并将 `ami` 参数设置为新添加的 `ami` 输入变量。

```
resource "aws_launch_configuration" "example" {
    image_id      = var.ami
    instance_type = var.instance_type
    security_groups = [aws_security_group.instance.id]

    user_data = data.template_file.user_data.rendered

    # 使用带有 ASG 的启动配置时必须使用
    # https://www.terraform.io/docs/providers/aws/r/launch\_configuration.html
    lifecycle {
        create_before_destroy = true
    }
}
```

现在，在预发布环境中 (*live/stage/services/webserver-cluster/main.tf*)，你可以设置新的 `ami` 和 `server_text` 参数，并删除 `enable_new_user_data` 参数。

```
module "webserver_cluster" {
```

```

source          = "../../modules/services/webserver-cluster"

ami            = "ami-0c55b159cbfafe1f0"
server_text    = "New server text"

cluster_name   = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"

instance_type  = "t2.micro"
min_size       = 2
max_size       = 2
enable_autoscaling = false
}

```

该代码使用相同的 Ubuntu AMI，但将 `server_text` 参数更新为一个新值。如果运行 `plan` 命令，会看到类似于以下内容的信息。

```

Terraform will perform the following actions:
# module.webserver_cluster.aws_autoscaling_group.example 将被原地替换
~ resource "aws_autoscaling_group" "example" {
  id           = "webservers-stage-terraform-20190516"
~ launch_configuration = "terraform-20190516" -> (known after apply)
  ...
}

# module.webserver_cluster.aws_launch_configuration.example 必须被替换
+/- resource "aws_launch_configuration" "example" {
  ~ id           = "terraform-20190516" -> (known after apply)
  image_id      = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  ~ name         = "terraform-20190516" -> (known after apply)
  ~ user_data    = "bd7c8a6" -> "4919a13" # 强制替换
  ...
}

Plan: 1 to add, 1 to change, 1 to destroy.

```

如你所见，Terraform 进行了两项更改：首先，将旧的启动配置替换为包含新的 `user_data` 的启动配置。其次，修改 Auto Scaling Group，引用新的启动配置。问题在于，仅仅引用新的启动配置是不起作用的，只有在 ASG 启动新的 EC2 实例时才能产生效果。那么如何

指示 ASG 部署新实例？

一种选择是销毁 ASG（通过运行 `terraform destroy` 命令），然后重新创建它（通过运行 `terraform apply` 命令）。问题是，在删除旧的 ASG 之后，新的 ASG 出现之前，用户将经历停机。如果要实现零停机部署。要先创建新的 ASG，然后再销毁原始的 ASG。事实证明，在第 2 章中看到的生命周期设置 `create_before_destroy`，完全可以做到这一点。

以下是如何利用这个生命周期设置，进行零停机部署的方法。<sup>1</sup>

1. 配置 ASG 的 `name` 参数，使其直接依赖于启动配置的名称。每次当启动配置发生变更时（更新 AMI 或用户数据时，启动配置都会变更），其名称都会更改，因此 ASG 的名称也会更改，这将强制 Terraform 替换 ASG。
2. 将 ASG 的 `create_before_destroy` 参数设置为 `true`，这样每当 Terraform 尝试替换它时，在销毁原始 ASG 之前，会首先创建新的 ASG。
3. 将 ASG 的 `min_elb_capacity` 参数，设置为集群的 `min_size`。这样做的好处是，Terraform 在开始销毁原始 ASG 之前，会等待新的 ASG 中，至少有 `min_size` 台服务器正常运行并通过 ALB 的运行状况检查。

这里是更新后的 `aws_autoscaling_group` 资源在 `modules/services/webserver-cluster/main.tf` 文件中的样子。

```
resource "aws_autoscaling_group" "example" {
    # 特意将启动配置的名称作为 ASG 名称的一部分，当启动配置发生替换时，ASG 也会被替换
    name = "${var.cluster_name}-${aws_launch_configuration.example.name}"

    launch_configuration      = aws_launch_configuration.example.name
    vpc_zone_identifier       = data.aws_subnet_ids.default.ids
    target_group_arns         = [aws_lb_target_group.asg.arn]
    health_check_type         = "ELB"

    min_size                  = var.min_size
    max_size                  = var.max_size
```

---

<sup>1</sup> 这个技巧要归功于 Paul Hinze（见参考资料第 5 章[2]）。

```

# 只有当通过健康检查的服务器达到以下数目时，才能判定 ASG 部署完成
min_elb_capacity      = var.min_size

# 当替换 ASG 时，总是先创建，再删除
lifecycle {
  create_before_destroy= true
}

tag {
  key          = "Name"
  value         = var.cluster_name
  propagate_at_launch = true
}

dynamic "tag" {
  for_each           = {
    for key, value in var.custom_tags:
    key => upper(value)
    if key != "Name" = "Name"
  }
}

content {
  key          = tag.key
  value         = tag.value
  propagate_at_launch= true
}
}
}

```

如果重新运行 `plan` 命令，将看到类似于下面的输出。

```

Terraform will perform the following actions:
# module.webserver_cluster.aws_autoscaling_group.example 必须被替换
+/- resource "aws_autoscaling_group" "example" {
  ~ id    = "example-2019" -> (known after apply)
  ~ name  = "example-2019" -> (known after apply) # 强制替换
  ...
}

# module.webserver_cluster.aws_launch_configuration.example 必须被替换
+/- resource "aws_launch_configuration" "example" {

```

```

~ id          = "terraform-2019" -> (known after apply)
~ image_id    = "ami-0c55b159cbfafe1f0"
~ instance_type = "t2.micro"
~ name        = "terraform-2019" -> (known after apply)
~ user_data   = "bd7c0a" -> "4919a" # 强制替换
(...)

}

(...)
```

Plan: 2 to add, 2 to change, 2 to destroy.

需要注意的关键点是 `aws_autoscaling_group` 资源在其 `name` 参数旁边显示`#forces replacement`（强制替换），这意味着 Terraform 将用包含新 AMI 或用户数据的 ASG 替换它。运行 `apply` 命令来启动部署。当命令运行时，可以思考一下实际运行过程。

过程是从你原来运行的 ASG 开始，让我们假设现在运行的是代码版本 v1，如图 5-2 所示为原始 ASG 运行在代码的 v1 版本。

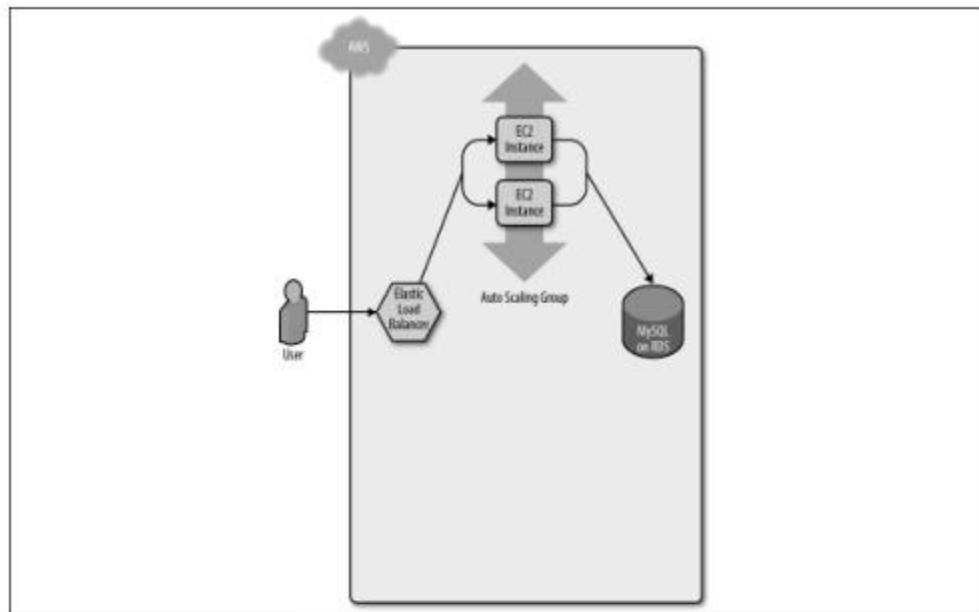


图 5-2：原始 ASG 运行在代码的 v1 版本

对启动配置的某些方面进行更新，例如切换到包含代码 v2 的 AMI，然后运行 `apply` 命令。这迫使 Terraform 开始部署包含代码 v2 的新的 ASG（图 5-3）。

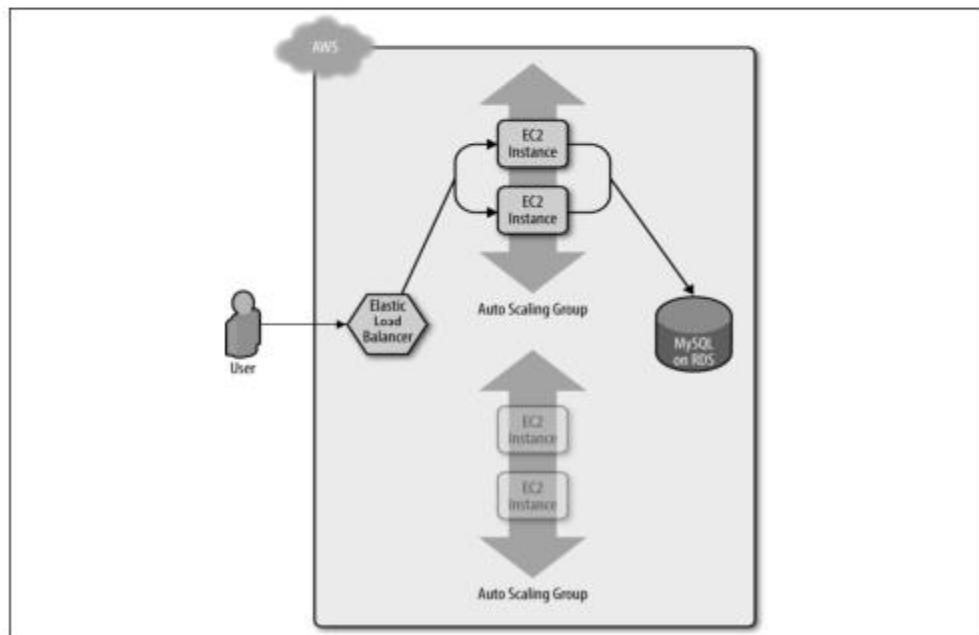


图 5-3：Terraform 开始部署包含 v2 代码版本的新的 ASG

一两分钟后，新 ASG 中的服务器启动，连接到数据库，在 ALB 中注册，并通过运行状况检查开始为流量提供服务，如图 5-4 所示。此时，应用程序的 v1 和 v2 版本将同时运行。用户看到的内容取决于 ALB 如何转发访问请求。

当 v2 版本的 ASG 服务器集群中，至少有 `min_elb_capacity` 数目的服务器在 ALB 完成注册时，Terraform 将开始从 ALB 中注销旧的 ASG 的服务器，然后关闭它们。如图 5-5 所示，旧的 ASG 中的服务器开始被关闭。

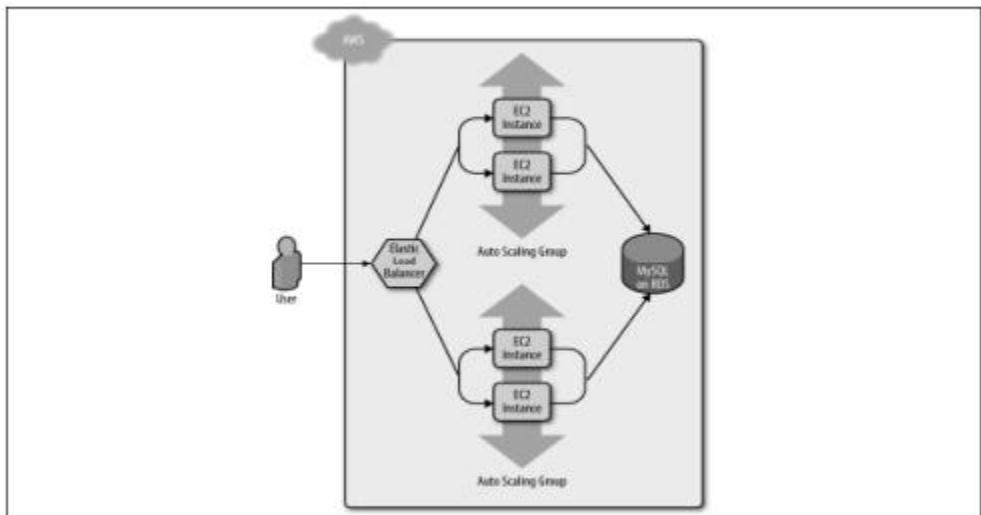


图 5-4：新 ASG 中的服务器启动，连接到数据库，在 ALB 中注册，并通过运行状况检查开始为流量提供服务

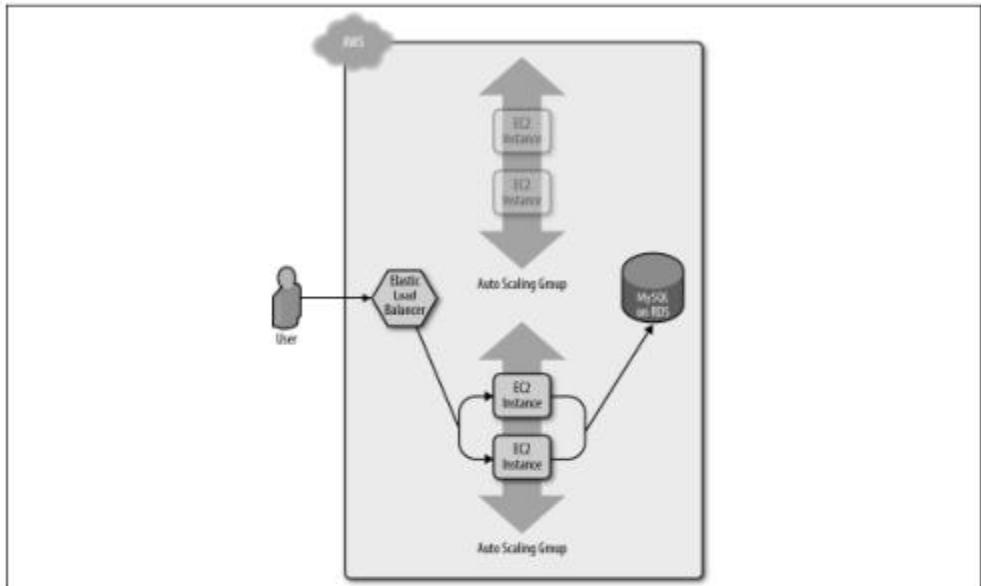


图 5-5：旧的 ASG 中的服务器开始被关闭

一两分钟后，旧的 ASG 将消失，环境中仅包含运行 v2 应用程序的新的 ASG，如图 5-6 所示。

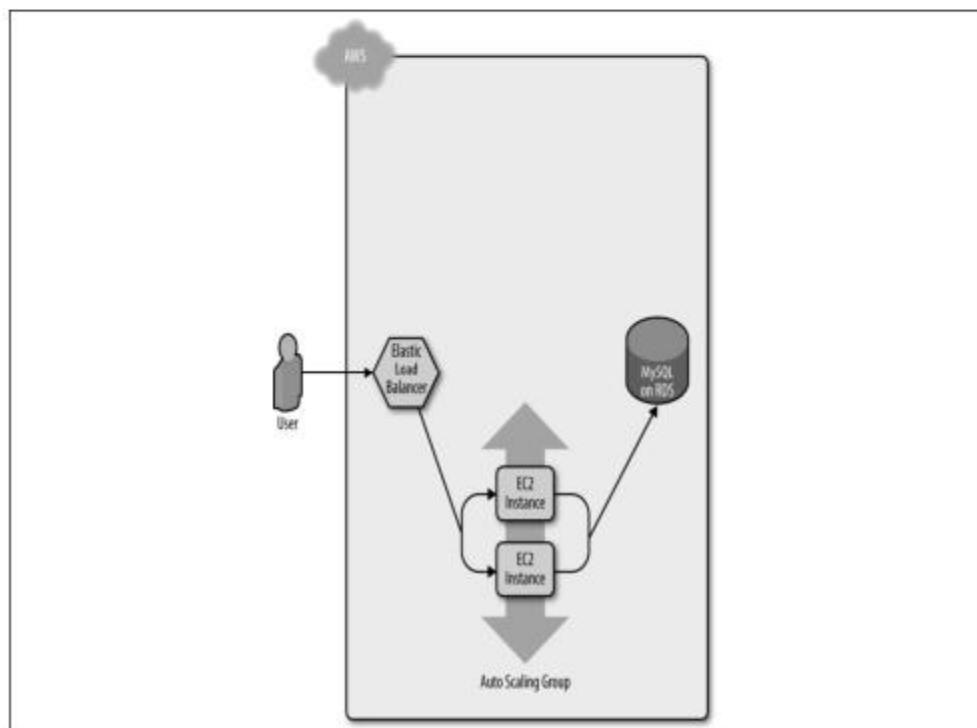


图 5-6：旧的 ASG 将消失，环境中仅包含运行 v2 应用程序的新的 ASG

在整个过程中，始终有服务器在运行并处理来自 ALB 的请求，因此没有停机时间。在浏览器中打开 ALB URL，你应该看到图 5-7 所示的已部署新代码。

成功！新的服务器文本已部署。让我们做一个有趣的实验，对 `server_text` 参数进行新的更改（例如，将其更新为 `foo bar`），然后运行 `apply` 命令。在另一个终端窗口中，如果你使用的是 Linux / UNIX / OS X，则可以使用单行的 Bash 脚本运行 `curl` 命令循环，每秒访问一次 ALB，这让你能近距离查看零停机部署的变化过程。

```
$ while true; do curl http://<load_balancer_url>; sleep 1; done
```

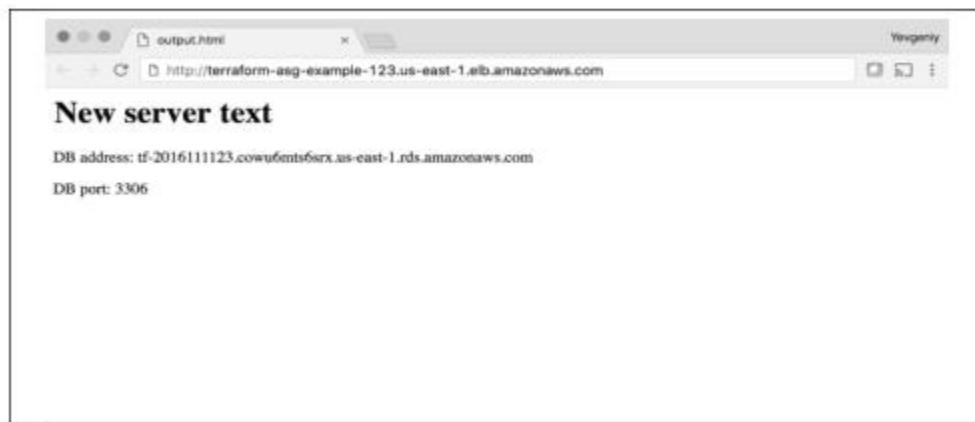


图 5-7：已部署新代码

在约一分钟时，你还是会看到相同的输出：`New server text`。然后，你将开始看到交替显示 `New server text` 和 `foo bar`。这意味着新实例已经在 ALB 中注册并通过了运行状况检查。再过一分钟，`New server text` 消息将消失，你只会看到 `foo bar`，这意味着旧的 ASG 已关闭。输出看起来像如下这样（为显示得更清楚，我只列出了`<h1>`标记内容）。

```
New server text
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

```
foo bar
```

如果在部署过程中出现问题，Terraform 将自动恢复到以前的版本。例如，如果你的应用程序 v2 中存在一个错误，并且无法启动，则新 ASG 中的实例将无法完成在 ALB 中的注册。Terraform 将在 “`wait_for_capacity_timeout`”（默认为 10min）时间内，确认 v2 版本 ASG 中是否有 “`min_elb_capacity`” 数目的服务器已经完成 ALB 中的注册。如果超时导致部署失败，将删除 v2 版本 ASG，并显示错误退出（此时原始 ASG 中 v1 版本的应用仍然正常运行）。

## Terraform 陷阱

在经历了所有这些提示和技巧之后，让我们退后一步，指出一些与循环、if 表达式和部署技术有关的陷阱，以及对 Terraform 整体而言更一般性的问题。

- `count` 和 `for_each` 的局限性
- 零停机部署的局限性
- 有效的计划仍会失败
- 重构需要很小心
- 最终一致性的问题

### `count` 和 `for_each` 的局限性

在本章的示例中，我们在循环和 if 表达式中广泛使用了 `count` 参数和 `for_each` 表达式。这虽然很好用，但是需要注意两个重要限制。

- 你无法在 `count` 或 `for_each` 中引用任何资源输出
- 你不能在 `module` 配置中使用 `count` 或 `for_each`

让我们深入研究一下。

首先，你无法在 `count` 或 `for_each` 中引用任何资源输出。

假设要部署多个 EC2 实例，并且由于某种原因不想使用 ASG，代码可能看起来像下面这样。

```
resource "aws_instance" "example_1" {
```

```
count          = 3
ami           = "ami-0c55b159cbfafe1f0"
instance_type = "t2.micro"
}
```

由于 `count` 设置为固定值，该代码可以正常工作。当你运行 `apply` 命令时，它将创建 3 个 EC2 实例。现在，如果要在当前 AWS 区域中的每个可用区（Availability Zone, AZ）部署一个 EC2 实例，该怎么办？你可以更新代码通过 `aws_availability_zones` 数据源来获取可用区列表，并使用 `count` 参数和数组查找，循环发现每一个可用区，并在其中创建 EC2 实例。

```
resource "aws_instance" "example_2" {
  count          = length(data.aws_availability_zones.all.names)
  availability_zone= data.aws_availability_zones.all.names[count.index]
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
data "aws_availability_zones" "all" {}
```

同样，此代码也可以正常工作，这是因为 `count` 引用可用区数据源。但是，如果你需要创建的实例数取决于某些资源的输出该怎么办？最简单的实验方法是使用 `random_integer` 资源，你可以从名称中猜测它的功能，返回一个随机整数。

```
resource "random_integer" "num_instances" {
  min = 1
  max = 3
}
```

此代码生成一个 1 到 3 之间的随机整数。让我们看看如果在 `aws_instance` 资源的 `count` 参数中使用此资源的 `result` 的输出，会发生什么。

```
resource "aws_instance" "example_3" {
  count          = random_integer.num_instances.result
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

如果运行 `terraform plan`，你会得到以下错误。

```
Error: Invalid count argument
on main.tf line 30, in resource "aws_instance" "example_3":
```

```
38: count      = random_integer.num_instances.result
```

The "count" value depends on resource attributes that cannot be determined until apply, so Terraform cannot predict how many instances will be created. To work around this, use the -target argument to first apply only the resources that the count depends on.

Terraform 要求在 `plan` 阶段计算 `count` 和 `for_each`, 这发生在任何资源创建或修改之前。意味着 `count` 和 `for_each` 可以引用预先设定的固定值、变量、数据源，甚至可以引用资源列表(只要列表的长度在 `plan` 阶段可以被确定), 但不能使用执行阶段计算生成的资源输出。

其次，你不能在模块配置中使用 `count` 或 `for_each`

你可能会想尝试在 `module` 配置中使用 `count` 参数。

```
module "count_example" {
  source = "../../modules/services/webserver-cluster"

  count = 3

  cluster_name      = "terraform-up-and-running-example"
  server_port       = 8080
  instance_type     = "t2.micro"
}
```

这段代码试图在 `module` 中使用 `count` 参数创建 3 个 `webserver-cluster` 资源的副本。或者，你可能会尝试通过将 `module` 里的 `count` 设置为 0，有条件地包含模块。尽管上面的代码看起来完全合理，但是如果运行 `terraform plan` 命令，则会出现以下错误。

```
Error: Reserved argument name in module block

on main.tf line 13, in module "count_example":
13: count = 3

The name "count" is reserved for use in a future version of Terraform.
```

不幸的是，到 Terraform 0.12.6 版本为止，`module` 还不支持 `count` 参数或 `for_each` 表达式。根据 Terraform 0.12 版本的发行说明（见参考资料第 5 章[3]），这是 HashiCorp 计划在将来添加的内容。因此在你阅读本书时，这个功能或许已经可以使用。具体情况请参考 Terraform CHANGELOG（见参考资料第 5 章[4]）。

## 零停机部署的局限性

将 `create_before_destroy` 参数与 ASG 结合使用，是一种很棒的零停机部署技术。但是有一个局限性：它不适用于自动扩展策略。或者更准确地说，每次部署之后，ASG 的大小将被重置为 `min_size`，如果你正在使用自动扩展策略，增加运行的服务器数量，就可能会出现问题。

例如，`webserver-cluster` 模块包含几个 `aws_autoscaling_schedule` 资源，这些资源将在上午 9 点，将集群中的服务器数量从 2 台增加到 10 台。如果在上午 11 点运行部署，新创建的 ASG 只会启动 2 台服务器（`min_size`），而不是 10 台，并将一直保持这种状态，直到第二天上午 9 点为止。

这里有几种可能的解决方法，具体如下。

- 将 `aws_autoscaling_schedule` 上的 `recurrence` 参数从 `0 9 * * *`（表示上午 9 点）更改为 `0-59 9-17 * * *`（即从上午 9 点到下午 5 点，每分钟运行一次）。如果 ASG 已经启动了 10 台服务器，反复运行这个自动缩放策略将不会带来任何效果。但是，如果部署了新的 ASG，运行这个策略可以确保 ASG 在一分钟以内，将服务器实例数增加到 10 台。这种方法有点烦琐，并且从 10 台服务器减少到 2 台服务器，再增加到 10 台服务器的过程，仍然会给用户造成一些问题。
- 使用 AWS API 创建一个自定义脚本，计算 ASG 中正在运行的服务器数目，再通过 `external` 数据源调用此脚本（请参阅第 6 章的“外部数据源”），并设置新的 ASG 的 `desirable_capacity` 参数为此脚本返回值。这样，无论何时启动新的 ASG，它的初始容量始终会与要替换的 ASG 保持相同。唯一的缺点是，使用自定义脚本会降低 Terraform 代码的可移植性，使其更难维护。

理想情况下，Terraform 将为零停机部署提供一流的支持。但是截至 2019 年 5 月，HashiCorp 团队表示没有计划在近期添加这个功能（请参阅参考资料第 5 章[5]了解详情）。

## 有效的计划仍然会失败

有时，当运行 `plan` 命令时，输出看起来是一个非常正确的有效的计划。但是当运行 `apply` 命令时，还是会得到一个错误。例如尝试添加一个与第 2 章中手动创建的 IAM 用户名称

完全相同的 `aws_iam_user` 资源。

```
resource "aws_iam_user" "existing_user" {
    # 你应该将其更改为一个已经存在的 IAM 用户，以便练习使用 terraform import 命令
    name = "yevgeniy.brikman"
}
```

如果现在运行 `plan` 命令，Terraform 将显示一个看起来很合理的输出。

```
Terraform will perform the following actions:
```

```
# aws_iam_user(existing_user) 将被创建
+ resource "aws_iam_user" "existing_user" {
    + arn          = (known after apply)
    + force_destroy = false
    + id           = (known after apply)
    + name         = "yevgeniy.brikman"
    + path          = "/"
    + unique_id     = (known after apply)
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

但如果运行 `apply` 命令，将出现以下错误。

```
Error: Error creating IAM User yevgeniy.brikman: EntityAlreadyExists:
User with name yevgeniy.brikman already exists.
```

```
on main.tf line 10, in resource "aws_iam_user" "existing_user":
10: resource "aws_iam_user" "existing_user" {
```

当然问题在于，这个 IAM 用户已经存在。这个问题不仅会出现在 IAM 用户上，而且几乎会发生在任何资源上。也许有人手动或通过 CLI 命令行创建了该资源，但是无论哪种方式，相同的对象标识符都会导致冲突。这类错误有很多不同的形式，Terraform 的新手通常会被这类错误搞得措手不及。

关键的问题是，`terraform plan` 命令只会检查 Terraform 状态文件中已经存在的资源。如果在 Terraform 范畴之外创建资源（如通过 AWS 控制台的手动操作），这些资源将不会被记录在 Terraform 的状态文件中。因此 Terraform 在运行 `plan` 命令时，不会考虑这些记录之外的资源。结果导致看起来有效的计划仍将失败。

从这一点来看，有两个主要的经验教训。

开始使用 Terraform 后，任何操作都要通过 Terraform 进行。

当基础设施的一部分已经由 Terraform 管理时，切勿手动对其进行更改。否则，不仅将面对各种怪异的 Terraform 错误，而且还会错过许多使用基础设施即代码（IaC）的好处，因为该代码已经不能准确地代表你的基础设施。

对已经存在的基础设施，请使用 `import` 命令。

如果在开始使用 Terraform 之前，已经创建了基础设施，则可以通过 `terraform import` 命令，将基础设施添加到 Terraform 的状态文件中，以便 Terraform 可以管理该基础设施。`import` 命令有两个参数。第 1 个参数是 Terraform 配置文件中资源的“地址”。这里使用与资源引用相同的语法：`<PROVIDER>_<TYPE>.<NAME>`（如 `aws_iam_user.existing_user`）。第 2 个参数是特定于资源的 ID，用于标识要导入的资源。例如，`aws_iam_user` 资源的 ID 和用户名相同（`yevgeniy.brikman`），而 `aws_instance` 资源的 ID 是 EC2 实例的 ID（`i-190e22e5`）。在每个资源文档的页面底部，通常都会描述如何导入它。

例如，你可以使用 `import` 命令，将 Terraform 配置中添加的 `aws_iam_user` 资源与你在第 2 章中创建的 IAM 用户进行同步（显然，需要将命令中的 `yevgeniy.brikman` 替换为你自己的用户名）。

```
$ terraform import aws_iam_user.existing_user yevgeniy.brikman
```

Terraform 将使用 AWS API 查找相应的 IAM 用户，并在状态文件中，将用户与 Terraform 配置中的 `aws_iam_user.existing_user` 资源之间创建关联。从那时起，当你运行 `plan` 命令时，Terraform 将知道这个 IAM 用户已经存在，不再尝试重复创建它。

请注意，如果需要导入大量现有资源到 Terraform，那么从头开始为它们编写 Terraform 代码并一次次地导入是件很麻烦的事情。一些工具可能会提供帮助，例如 Terraforming（见参考资料第 5 章[6]），它可以自动从一个 AWS 账户导入代码和状态。

## 重构需要很小心

重构（refactoring）是一种常见的编程实践。特指在不改变其外部行为的情况下，重新构

建现有代码的内部细节。重构的目的是提高代码的可读性、可维护性和整洁性。重构是用户应该定期进行的基本编码实践。然而，当谈到 Terraform 或任何 IaC 工具，必须十分小心什么是一段代码的“外在行为”，否则会遇到意外的错误。

例如，一种常见的重构实践是，对变量或函数重新命名，使其更加清晰。甚至许多 IDE 都具有对重构的内置支持，可以在整个代码库中自动对变量或函数重新命名。

在通用编程语言中，重命名是无须过多考虑的。但是在 Terraform 中的重命名必须非常小心，否则可能会导致意外故障。

例如，`webserver-cluster` 模块有一个名为 `cluster_name` 的输入变量。

```
variable "cluster_name" {
  description      = "The name to use for all the cluster resources"
  type            = string
}
```

在开始使用此模块来部署微服务时，最初的微服务名称为 `foo`。稍后考虑将服务重命名为 `bar`。这看似微不足道的变化，实际上可能导致故障。

这是因为 `webserver-cluster` 模块在许多资源中都会使用 `cluster_name` 变量，包括两个安全组和 ALB 的 `name` 参数。

```
resource "aws_lb" "example" {
  name          = var.cluster_name
  load_balancer_type = "application"
  subnets       = data.aws_subnet_ids.default.ids
  security_groups = [aws_security_group.alb.id]
}
```

当某些资源的 `name` 参数发生了更改时，Terraform 将删除该资源的旧版本并创建一个新版本来替换它。如果将被替换的资源恰好是 ALB，在新的 ALB 启动之前，将没有任何设施来处理 Web 服务器集群的流量路由。同样，如果要被替换的资源恰好是安全组，则在更替期间，服务器将拒绝所有网络请求，直到新的安全组创建完成。

另一种重构的可能性是更改 Terraform 标识符。例如，`webserver-cluster` 模块中的 `aws_security_group` 资源。

```
resource "aws_security_group" "instance" {
  # ...
}
```

此资源的标识符被称为 `instance`。也许你正在进行重构，认为将名称更改为 `cluster_instance` 会更清晰。

```
resource "aws_security_group" "cluster_instance" {
  # ...
}
```

重构的结果是什么？没错你猜对了：停机。

Terraform 将每个资源的标识符与来自 Cloud 提供商的标识符相关联。例如，`iam_user` 资源与 AWS IAM 用户 ID 相关联，`aws_instance` 资源与 AWS EC2 实例 ID 相关联。如果对资源标识符进行更改。例如，将 `aws_security_group` 的标识符从 `instance` 更改为 `cluster_instance`，从 Terraform 的角度来看，是删除了旧资源并添加了一个全新的资源。如果 `apply` 这些更改，Terraform 将删除旧的安全组并创建一个新的安全组，在这期间服务器将拒绝所有网络流量。

综上所述，下面是 4 个主要的经验。

#### 始终使用 `plan` 命令

运行 `plan` 命令可以捕获所有这些陷阱。仔细阅读输出结果，尤其注意 `terraform plan` 输出提示中的那些将要被删除但是你不想删除的资源。

#### 在销毁前创建

如果确实要替换资源，请仔细考虑是否需要在删除之前先进行创建。如果需要这样，你可以通过 `create_before_destroy` 参数来实现。或者，也可以通过两个手动步骤来实现相同的效果：首先，将新资源添加到配置中，运行 `apply` 命令；接下来，从配置中删除旧资源，再次运行 `apply` 命令。

#### 更改标识符需要更改状态文件

如果要更改与资源关联的标识符（例如，将 `aws_security_group` 从 `instance` 重命名

为 `cluster_instance`），而又不想意外地删除和重建该资源，则需要对 Terraform 状态文件进行相应地更新。永远不要手动更新 Terraform 状态文件，而要使用 `terraform state` 命令来完成更新。在重命名标识符时，需要运行 `terraform state mv` 命令，该命令具有以下语法。

```
terraform state mv <ORIGINAL_REFERENCE> <NEW_REFERENCE>
```

其中 `ORIGINAL_REFERENCE` 是当前对资源的引用表达式，`NEW_REFERENCE` 是要将其移动到的新位置。例如，如果要将 `aws_security_group` 从 `instance` 重命名为 `cluster_instance`，则需要运行以下命令。

```
$ terraform state mv \
  aws_security_group.instance \
  aws_security_group.cluster_instance
```

指示 Terraform 将以前与 `aws_security_group.instance` 关联的状态全部变更为与 `aws_security_group.cluster_instance` 相关联。如果在重命名标识符后运行了这个命令，在今后运行 `terraform plan` 命令时，将显示没有任何更改。

#### 一些参数是不可变的

许多资源的参数都是不能被更改的。如果更改它们，Terraform 将删除旧资源并创建一个新资源来替换它。每个资源的文档通常会说明如果你更改参数会发生什么，因此请养成查阅文档的好习惯。再次强调，请始终使用 `plan` 命令，并考虑是否应使用 `create_before_destroy` 策略。

#### 最终一致性的问题

某些云服务提供商（如 AWS）的 API 是异步的、最终一致的。异步意味着 API 可以立即响应请求，而无须等待请求的操作完成。最终一致性意味着，更改需要花费一些时间才能在整个系统中完成变更，因此在开始一段时间内，你可能会收到不一致的查询结果，具体内容取决于哪个数据存储副本恰好响应 API 调用。

假设你对 AWS 进行了 API 调用，要求其创建 EC2 实例，API 将立即返回成功响应（201 Created），而无须等待 EC2 实例创建完成。如果现在立即尝试连接到该 EC2 实例，很可能失败，因为 AWS 仍在进行配置或该实例尚未启动。如果你再次调用 API 来获取有关

该 EC2 实例的信息，则可能会返回错误（即 404 Not Found）。这是因为有关该 EC2 实例创建的信息可能仍会在整个 AWS 系统中传播，要花几秒钟才能到达系统的各个角落。

总之，如果使用异步和最终一致性的 API，应该等待一段时间，直到该操作已经确认完成并更新整个系统后再重试。不幸的是，AWS 开发工具包没有提供这方面的良好工具，Terraform 过去常常遇到许多类似的错误 # 6813（见参考资料第 5 章[7]）。

```
$ terraform apply  
aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:  
The subnet ID 'subnet-xxxxxx' does not exist
```

这个错误报告描述的是，当创建一个资源（如子网），并尝试获取有关该资源的一些数据（如新创建的子网的 ID）时，Terraform 却找不到它。虽然这些错误（包括 # 6813）中的大多数已得到修复，但仍会不时出现，特别是当 Terraform 添加了对新资源类型的支持时。幸运的是，这些很烦人的错误大多数是无害的。重复运行 `terraform apply` 命令，一切都会恢复正常，因为重新运行命令时，资源变更信息已经传播到整个系统的各个角落了。

## 小结

尽管 Terraform 是一种声明性语言，但是它通过大量的工具，例如变量和模块（第 4 章中能看到），以及在本章中看到的 `count`、`for_each`、`for`、`create_before_destroy` 和内置函数，使语言具有令人惊讶的灵活性和表达能力。本章介绍了大量的 `if` 表达式的使用技巧，请花一些时间浏览功能文档（见参考资料第 5 章[8]），这会使你获得更多的灵感。当然编写代码时也不要太过疯狂，因为仍然需要其他人来阅读和维护你的代码。恰到好处的灵感，可以为模块创建干净而漂亮的 API。

现在让我们继续学习第 6 章，我不仅将介绍如何创建干净而漂亮的模块，而且还是那种你的公司一定会用到的生产级模块。

## 第 6 章

# 生产级 Terraform 代码

构建生产级基础设施是一件很困难的工作，不仅充满压力，而且耗费时间。生产级基础设施 (*production-grade infrastructure*) 包括服务器、数据存储、负载平衡器、安全功能、监视和警报工具、构建管道，以及经营业务所需的所有其他技术。公司将依赖于你构建的基础设施，当流量增加时，基础设施不会崩溃；当发生故障中断时，基础设施不会丢失；当黑客试图闯入时，数据不会被错误地篡改。如果发生了上面描述的任何意外，你的公司都可能会面临倒闭的风险。这就是我在本章中提到的生产级基础设施的挑战所在。

我曾有机会与数百家公司合作，根据所有这些经验，以下是部署一个生产级基础设施项目大概需要花费的时间。

- 如果你的部署完全基于第三方托管服务，例如使用 AWS Relational Database Service (RDS) 运行 MySQL，那么项目将花费 1 到 2 个星期才能将服务部署到生产环境。
- 如果你要运行自维护的、无状态分布式应用程序，例如，Node.js 应用程序集群，运行在 AWS Auto Scaling Group (ASG) 之上，且不需要在本地存储任何数据（将所有数据存储在 RDS 中）。那么准备生产环境部署的时间大约是原来的两倍，即 2 到 4 个星期。
- 如果你要运行自维护的、有状态分布式应用程序，例如，Amazon Elastic Search (Amazon ES) 集群，在 ASG 之上运行并需要将数据存储在本地磁盘上。那么项目时间将增加另一个数量级，大约要 2 到 4 个月时间用来准备生产环境部署。

- 如果你要构建整个体系架构，包括所有应用程序、数据存储、负载平衡器、监视、警报、安全性等，则又会增加一个或两个数量级，大约 6 到 36 个月的工作量。小型公司通常要花 6 个月的时间，而大型公司通常要花几年的时间。

表 6-1 汇总了这些数据，显示了从零开始构建生产级基础设施需要的时间。

表 6-1：从零开始构建生产级基础设施需要的时间

基础设施类型	举例	时间估计
托管服务	Amazon RDS	1 ~ 2 周
自维护分布式系统（无状态）	Node.js 应用集群	2 ~ 4 周
自维护分布式系统（有状态）	Amazon ES	2 ~ 4 个月
全部基础设施	应用程序、数据存储、负载平衡器、监视等	6 ~ 36 个月

如果从未经历过构建生产级基础设施的过程，你可能会对这些数字感到惊讶。我经常听到如下反应。

- “怎么可能要花这么长时间？”
- “我在几分钟内就可以完成在‘云’上部署一台服务器。肯定不需要花费数月的时间来完成剩下的工作！”
- “这都是其他人的数字，而我肯定能够在几天内完成。”很多时候，这来自一位自信满满的工程师。

然而，经历过重大的云迁移，或者曾经从零开始搭建过全新基础设施的人，都知道这些数字是真实的，而且是最好情况下的乐观估计。如果团队在构建生产级基础设施方面缺乏具有深厚专业知识的成员，或者团队忙于各种不同的任务，找不到时间专注于这项工作，那么可能会花费更长时间。

在本章中，我将介绍为什么构建生产级基础设施需要花费这么长的时间，生产级的真正含义是什么，以及哪种模式最适合用来创建可重用的生产级模块。

- 为什么构建生产级基础设施需要漫长的过程
- 生产级基础设施检查清单
- 生产级基础设施模块特点
  - 模块要小型化
  - 可组合的模块
  - 可测试的模块
  - 可发布的模块
  - Terraform 模块之外的内容

## 为什么构建生产级基础设施需要漫长的过程

软件项目的时间估计都是非常不准确的，DevOps 项目的时间估算只会更糟糕。以为只需要 5min 的快速修改，往往会占用一整天的时间；估计需要一天完成的简单功能却持续了两个星期；认为两周后就会投入生产的应用程序，在 6 个月之后尚未完成。基础设施和 DevOps 项目，也许比其他任何类型的软件项目更符合霍夫施塔特定律：<sup>1</sup>

即使考虑到霍夫施塔特定律，其花费的时间也总是比你预期的长。

我认为有 3 个主要原因。第 1 个原因是，DevOps 作为一个行业，仍然处在石器时代。我并不认为这么说是一种侮辱，在某种意义上说该行业仍处于起步阶段。云计算、IaC 和 DevOps 这些术语，仅在 2000 年后第 1 个 10 年中期至后期才出现。而诸如 Terraform、Docker、Packer 和 Kubernetes 之类的工具最初都在 2000 年后第 2 个 10 年的中期至后期才发布。所有这些相对较新的工具和技术都在迅速地变化着。这也意味着它们并不是特别成熟，缺乏经验丰富的从业人员，因此项目花费时间比预期时间更长，也就不足为奇了。

第 2 个原因是 DevOps 似乎特别容易像剪耗牛毛的故事。我向你保证，如果以前从未听说过“剪耗牛毛”，那么这是一个让你很喜欢或非常憎恨的名词。这个词的最佳定义来自塞

---

<sup>1</sup> 霍夫施塔特 (Hofstadter)，道格拉斯·戈德尔 (Douglas R. Gödel)，埃舍尔 (Escher)，巴赫，永恒的金色粒子，20 周年版，纽约：Basic Books，1999。

思·戈丁（Seth Godin）的博客文章：<sup>1</sup>

“我今天想给汽车打蜡。”

“糟糕，水管从冬天起就断裂了。我要去 Home Depot 买一个新的。”

“但是 Home Depot 在 Tappan Zee 桥的另一侧，是收费路段，我没有 EZPass 到那里会很痛苦。”

“等一下！我可以借用我邻居的 EZPass……”

“邻居鲍勃不会借给我他的 EZPass，除非我儿子还给他从他那里借来的 mooshi 枕头。”

“我们之所以还没有归还枕头，因为有些填充料掉了出来，我们需要一些牦牛的毛来重新填充。”

下一件事是，你需要去动物园里剪牦牛毛，以便可以给汽车打蜡。

“剪牦牛毛”这个俗语，泛指所有细微的、看似无关的任务，但是你必须先完成这些任务，才能执行本来想做的事情。如果开发软件，尤其是在 DevOps 行业工作，那么你可能已经看到过上千次这样的情况了。例如，部署一个错字修复程序，其实错字的影响微不足道，只会导致应用程序配置中的一个小错误。但当你部署针对应用程序配置的修复程序后，却突然遇到 TLS 证书类别问题。在 StackOverflow 上钻研了数小时之后，你终于解决了 TLS 问题。尝试再次部署，这次由于部署系统的某些问题而导致失败。又花了数小时来研究部署问题，你发现这次是由于 Linux 版本过时而造成的。接下来，更新整个服务器集群上的操作系统。所有这些工作，仅仅是为了可以“快速”部署针对一个错字的修复程序。

DevOps 似乎特别容易发生此类“剪牦牛毛”的事情。部分原因是 DevOps 技术和现代系统设计不成熟的结果，系统通常与基础设施存在过于紧密的耦合和重复。在 DevOps 世界中所做的每项更改，都有点像试图从一团缠绕的电缆中拉出一根 USB 线，最终使盒子中的其他所有东西都被拉出来。部分原因是因为 DevOps 一词涵盖了令人惊讶的广泛主题：从构建到部署再到安全性等所有内容。

这将引出 DevOps 项目总是很漫长的第 3 个原因。前两个原因：DevOps 在石器时代，“剪牦牛毛”事件，都可以被归类为偶然的复杂性（*accidental complexity*）。偶然的复杂性是

---

<sup>1</sup> 赛思·戈丁，不要给牦牛剃毛，2005。

指选择的特定工具和流程所带来的问题，这与本质的复杂性（*essential complexity*）不同。本质的复杂性是指工作本身所固有的问题。<sup>1</sup>例如，如果使用 C++ 编写股票交易算法，则处理内存分配错误是偶然的复杂性问题。因为如果选择另一种可以自动管理内存的编程语言，就不会遇到这个问题。但是提出一种可以赢得市场的算法则是本质的复杂性问题，因为无论选择何种编程语言，都无法避免地要解决这个问题。

DevOps 项目花费很长时间的第 3 个原因，也是这个问题的本质复杂性原因，在部署生产环境的基础设施之前，有许多的准备工作。如果我们把所有工作列在一个检查清单上，绝大多数开发人员都不了解清单中的各项内容。因此当他们对项目进行评估时，会忽略大量关键且耗时的细节。该清单是下一部分的介绍重点。

## 生产级基础设施检查清单

让我们做一个有趣的实验。在公司里随机问一个问题：“生产环境发布的基本要求是什么？”在大多数公司中，如果向 5 个人问到同样的问题，会得到 5 个不同的答案。有人提到度量指标和警报；有人提到容量规划和高可用性；有人关注自动测试和代码审查；有人关注加密、身份验证和服务器功能补强；如果幸运的话，也许有人会提出数据备份和日志聚合。大多数公司没有明确地定义生产环境发布的要求，这意味着每个基础设施的部署都略有不同，并且可能会忽略一些关键功能。

为了帮助改善这种情况，我想在这里分享一个生产级的基础设施检查清单，表 6-2 为生产级基础设施检查清单。此列表涵盖了将基础结构部署到生产环境中需要考虑的大多数关键指标。

---

<sup>1</sup> Brooks, Frederick P. Jr.: *Mythical Man-Month* 软件工论文集程，周年纪念版，雷德辛格马萨诸塞州：Addison-Wesley Professional, 1995.

表 6-2：生产级基础设施检查清单

任务	描述	工具举例
安装	安装二进制软件包和依赖库	Bash、Chef、Ansible、Puppet
配置	软件运行期间的配置，包括端口、TLS 证书、服务发现、主节点、从节点、复制等	Bash、Chef、Ansible、Puppet
服务开通	基础设施服务开通，包括服务器、负载平衡器、网络配置、防火墙设置、IAM 权限等	Terraform、CloudFormation
部署	在基础设施之上部署服务，无停机更新，包括蓝绿部署、滚动部署和金丝雀部署	Terraform、CloudFormation、Kubernetes、ECS
高可用性	在单个流程、服务器、服务、数据中心或区域发生中断时，服务持续的能力	多数据中心、多区域、复制、自动缩放、负载均衡
扩展性	根据负载进行规模缩放、水平缩放（更多服务器）、自动缩放、复制、分片、缓存、分治垂直缩放（更大的服务器）	
性能	优化 CPU、内存、磁盘、网络和 GPU 的使用。方法包括查询调整、基准测试、负载测试和分析	Dynatrace、valgrind、VisualVM、ab、Jmeter
网络	配置静态和动态 IP、端口、服务发现、防火墙、VPC、防火墙、路由器、DNS、DNS、SSH 访问和 VPN 访问	VPC、防火墙、路由器、DNS、注册、OpenVPN
安全	传输中安全加密（TLS）和磁盘上的安全加密、身份验证、授权、机密管理、服务器强化	ACM、Let's Encrypt、KMS、Cognito、Vault、CIS
度量指标	可用性指标、业务指标、应用指标、服务器指标、事件、可观察性、跟踪和警报	CloudWatch、DataDog、New Relic、Honeycomb
备份和恢复	定期备份数据库、缓存和其他数据。复制到独立的区域/账户	RDS、ElastiCache、replication
成本优化	选择合适的实例类型，使用竞价型和预留实例、使用自动缩放，清除未使用的资源	自动缩放、竞价型实例、预留实例
文档	记录代码、体系结构和实践，创建剧本应对事件	自述文件、Wiki、Slack
测试	为基础设施代码编写自动测试，在每次提交后运行测试、每晚运行测试	Terratest、Inspec、Serverspec、kitchen-terraform

大多数开发人员都知道前几个任务：安装、配置、服务开通和部署。紧随其后的那些条目往往使他们措手不及。例如，是否考虑过服务的弹性？如果服务器出现故障该怎么办？如果负载均衡器出现故障呢？整个数据中心都被黑客攻击，又该怎么办？众所周知，网络任

务也很棘手：设置 VPC、VPN、服务发现和 SSH 访问都是必不可少的任务，可能要花费数月的时间，但实际情况是，在许多项目的计划和时间估计中，这些工作都被完全忽略了。安全任务，例如使用 TLS 进行数据传输加密、处理身份验证，以及存储机密信息，通常也要等到最后一刻才会被提起。

每次在新的基础设施上工作时，请务必根据这个清单进行检查。并非每个基础设施都需要列表中的所有条目，但是你应该有意识地、明确地记录已经实施的条目、决定跳过的条目，以及跳过的原因。

## 生产级基础设施模块特点

既然已经知道了针对每个基础设施需要完成的任务列表，下面我们来讨论如何通过构建可重用模块来实现这些任务的最佳实践。我将介绍以下主题。

- 模块要小型化
- 可组合的模块
- 可测试的模块
- 可发布的模块
- Terraform 模块之外的内容

### 模块要小型化

Terraform 和 IaC 的新手通常会在单个文件或单个模块中定义所有基础设施和所有环境（如 Dev、Stage、Prod 等）。如第 3 章的“隔离状态文件”中所述，这不是一个正确的做法。实际上，我将进一步提出以下主张：超过数百行代码的大型模块或包含多个不相关的部署的基础设施模块，都是有害的。

让我来举例说明大型模块的一些缺点。

#### 大型模块很慢

如果所有基础设施都定义在一个 Terraform 模块中，那么运行任何命令都将花费很长的时间。我曾经看到过很庞大的模块，每次运行 `terraform plan` 命令，都需要 5 到 6min 才能完成！

## 大型模块不安全

如果在一个大型模块中管理所有基础设施，那么任何局部的改动都需要具有访问所有内容的权限。这意味着几乎每个用户都必须是管理员，这就违反了最小特权原则 (*principle of least privilege*)。

## 大型模块有风险

如果将所有鸡蛋都放在一个篮子里，则任何局部的错误都可能破坏所有物品。例如，在预发布环境中，为了对前端应用程序做细微的改动，你不小心拼写错误或运行了错误的命令，这都会导致意外删除生产环境数据库的发生。

## 大模块很难被理解

一个模块中拥有的代码越多，任何个人就会越难理解全部代码。当处理并不完全理解的基础设施时，最终会犯下代价高昂的错误。

## 大型模块很难评审

评审由几十行代码组成的模块很容易。但评审包含数千行代码的模块是几乎不可能的。而且，`terraform plan` 命令不仅运行时间会更长，`plan` 命令的输出结果也会多达几千行，这意味着没有人会去仔细阅读它。例如，没有人会注意到一条小红线表示你的数据库将要被删除。

## 大模块很难测试

基础设施代码很难测试。测试大量的基础设施代码几乎是不可能的。我将在第 7 章具体分析这一点。

简而言之，应该使用小模块来构建代码，每个模块各自完成一个功能。这并不是一个新的或有争议的理论。你可能在不同场合已经听过很多次了，例如 *Clean Code* 中提到的：<sup>1</sup>

函数的第一个规则是它们应该很小；函数的第二个规则是它们应该更小。

假设你正在使用 Java、Python 或 Ruby 之类的通用编程语言，并且遇到了一个 20,000 行

---

<sup>1</sup> 马丁·罗伯特·C. *Clean Code*, 第 1 版, [美]新泽西: Prentice Hall, 2008.

以上的庞大函数。

```
def huge_function(data_set)
    x_pca = PCA(n_components=2).fit_transform(X_train)
    clusters = clf.fit_predict(X_train)
    ax = plt.subplots(1, 2, figsize=(4))
    ay = plt.subplots(0, 2, figsize=(2))
    fig = plt.subplots(3, 4, figsize=(5))
    fig.subplots_adjust(top=0.85)

    predicted = svc_model.predict(X_test)
    images_and_predictions = list(zip(images_test, predicted))

    for x in 0..xlimit
        ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
        ax[0].set_title('Predicted Training Labels')
        ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
        ax[1].set_title('Actual Training Labels')
        ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
    end

    for y in 0..ylim
        ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
        ay[0].set_title('Predicted Training Labels')
        ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
        ay[1].set_title('Actual Training Labels')
        ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
    end

    #
    # 更多函数内容.....
    #
end
```

你会立即知道这是一段有问题的代码，更好的方法是将其重构为多个小而且独立的函数，每个函数只做一件事。

```
def calculate_images_and_predictions(images_test, predicted)
    x_pca = PCA(n_components=2).fit_transform(X_train)
    clusters = clf.fit_predict(X_train)
    ax = plt.subplots(1, 2, figsize=(4))
```

```

fig = plt.subplots(3, 4, figsize=(5))
fig.subplots_adjust(top=0.85)

predicted = svc_model.predict(X_test)
return list(zip(images_test, predicted))
end

def process_x_coords(ax)
  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end

  return ax
end

def process_y_coords(ax)
  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
  end

  return ay
end

#
# 更多小函数.....
#

```

在 Terraform 中应该使用同样的策略。想象一下，如果遇到了如图 6-1 所示的相对复杂的 AWS 架构。

如果此架构是定义在 20,000 行长的单个的大型 Terraform 模块中，则应立即将这视为一种代码异味。更好的方法是将其重构为多个小型且独立的模块，每个模块都只做一件事情，如图 6-2 所示，将相对复杂的 AWS 架构重构为许多小型模块。

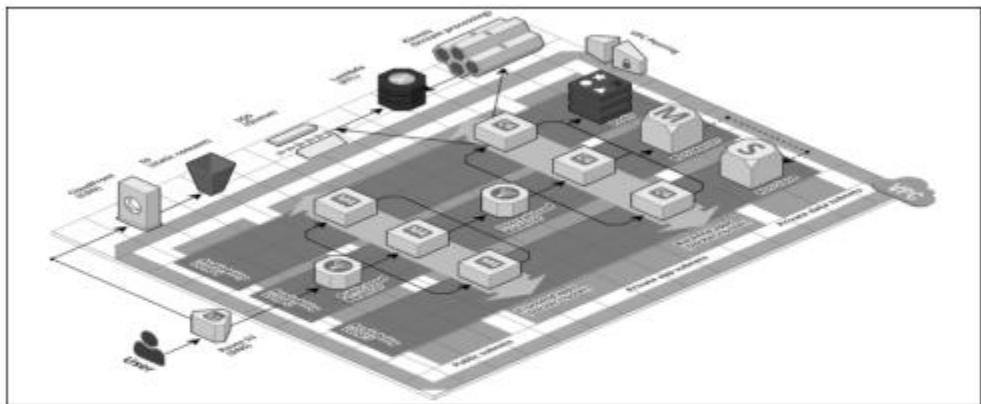


图 6-1：相对复杂的 AWS 架构

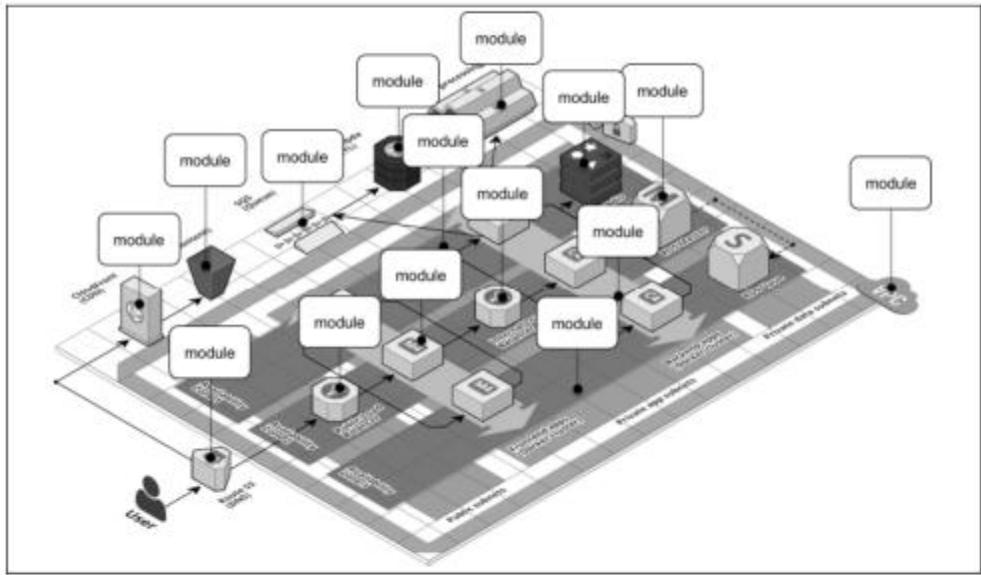


图 6-2：将相对复杂的 AWS 架构重构为许多小型模块

我们一直在构建的 `webserver-cluster` 模块，逐渐显得有点庞大了，它同时操控 3 种不相关的任务。

## 自动缩放组 (ASG)

`webserver-cluster` 模块部署了一个可以执行零停机、滚动部署的 ASG。

## 应用负载均衡器 (ALB)

`webserver-cluster` 模块部署了一个 ALB。

## Hello, World 应用程序

`webserver-cluster` 模块还部署了一个简单的“Hello, World”应用程序。

让我们将上述代码重构为 3 个较小的模块。

### `modules/cluster/asg-rolling-deploy`

一个通用的、可重用的独立模块，用于部署一个可以进行零停机、滚动部署的 ASG。

### `modules/networking/alb`

一个通用的、可重用的独立模块，用于部署一个 ALB。

### `modules/services/hello-world-app`

一个专门用于部署“Hello, World”应用的模块。

在开始之前，请确保运行了 `terraform destroy` 命令，清理上一章中 `webserver-cluster` 的相关部署内容。之后，可以将 `asg-rolling-deploy` 和 `alb` 模块组合在一起。创建一个新文件夹 `modules/cluster/asg-rolling-deploy`，然后将以下资源从 `module/services/webserver-cluster/main.tf` 文件移动（复制/粘贴）到 `modules/cluster/asg-rolling-deploy/main.tf` 文件中。

- `aws_launch_configuration`
- `aws_autoscaling_group`
- `aws_autoscaling_schedule`（两个）
- `aws_security_group`（仅限于管理实例的安全组，而不是管理 ALB 的安全组）
- `aws_security_group_rule`（仅限于实例的安全组规则，而不是为 ALB 创建的安全组规则）
- `aws_cloudwatch_metric_alarm`（全部）

接下来，将以下变量从 *module/services/webserver-cluster/variables.tf* 文件移动到 *modules/cluster/asg-rolling-deploy/variables.tf* 文件中。

- `cluster_name`
- `ami`
- `instance_type`
- `min_size`
- `max_size`
- `enable_autoscaling`
- `custom_tags`
- `server_port`

现在让我们开始准备 ALB 模块。创建一个新的文件夹 *modules/networking/alb*，将下列资源从 *module/services/webserver-cluster/main.tf* 文件移动到 *modules/networking/alb/main.tf* 文件中。

- `aws_lb`
- `aws_lb_listener`
- `aws_security_group`（仅限于管理 ALB 的安全组，而不是管理实例的安全组）
- `aws_security_group_rule`（仅限于两个 ALB 的安全组规则，而不是为实例创建的安全组规则）

创建文件 *modules/networking/alb/variables.tf* 并定义单个变量。

```
variable "alb_name" {
  description      = "The name to use for this ALB"
  type             = string
}
```

将此变量用作 `aws_lb` 资源的 `name` 参数。

```
resource "aws_lb" "example" {
  name           = var.alb_name
  load_balancer_type = "application"
  subnets        = data.aws_subnet_ids.default.ids
```

```
    security_groups      = [aws_security_group.alb.id]
}
```

将此变量 `aws_security_group` 资源的 `name` 参数。

```
resource "aws_security_group" "alb" {
  name           = var.alb_name
}
```

这里有大量的代码移动，你也可以直接使用本章的代码示例，地址见前言“开源代码示例”。

### 可组合的模块

现在，我们有了两个小型模块：`asg-rolling-deploy` 和 `alb`，它们每个都专注于一个功能。接下来如何使它们一起工作呢？该如何构建可重用和可组合的模块呢？这个问题并不是 Terraform 独有的，程序员数十年以来一直在思考同样的问题。引用 UNIX 管道的最初开发者 Doug McIlroy（他同时也是许多其他 UNIX 工具的开发者，包括 `diff`、`sort`、`join` 和 `tr`）的话<sup>1</sup>：

UNIX 的哲学思想是：每个程序只做一件事并做得很完美，再编写程序让它们协同工作。

函数组合（*function composition*）是达到这一点的一种方法。在函数组合中，可以将一个函数的输出，作为输入传递给另一个函数。例如，如果你在 Ruby 中具有如下的简单函数。

```
# 执行加法的简单函数
def add(x, y)
  return x + y
end

# 执行减法的简单函数
def sub(x, y)
  return x - y
end

# 做乘法的简单函数
```

---

<sup>1</sup> Salus, Peter H. *A Quarter-Century of Unix*. [美]纽约 Addison-Wesley Professional, 1994.

```
def multiply(x, y)
    return x * y
end
```

通过使用函数组合将它们放在一起，把 `add` 和 `sub` 函数的输出作为 `multiply` 函数的输入。

```
# 几个简单的函数组成复杂的函数
def do_calculation(x, y)
    return multiply(add(x, y), sub(x, y))
end
```

使函数可组合的主要方法之一是，最大程度地减少副作用，也就是说，在可能的情况下，避免从外界直接读取状态，尽量通过输入参数传递状态。避免将状态直接写入外界，尽量通过输出参数返回计算结果。“副作用最小化”是函数式编程的核心宗旨之一，它使代码更易于推理、测试和重用。重用的特性特别引人注目，因为函数组合允许你通过使用简单的函数来逐步构建更复杂的函数。

尽管在编写基础设施代码时无法完全避免副作用，但是仍然可以在 Terraform 模块中遵循相同的基本原则：通过输入变量传递内容，通过输出变量返回内容，并通过组合简单的模块来构建更复杂的模块。

打开 `modules/cluster/asg-rolling-deploy/variables.tf` 文件，添加 4 个新的输入变量。

```
variable "subnet_ids" {
    description = "The subnet IDs to deploy to"
    type        = list(string)
}

variable "target_group_arns" {
    description = "The ARNs of ELB target groups in which to register Instances"
    type        = list(string)
    default     = []
}

variable "health_check_type" {
    description = "The type of health check to perform. Must be one of: EC2, ELB."
    type        = string
    default     = "EC2"
}

variable "user_data" {
    description = "The User Data script to run in each Instance at boot"
    type        = string
}
```

```
    default      = ""
}
```

第1个变量 `subnet_ids` 定义了 `asg-rolling-deploy` 模块将要部署到的子网。虽然之前我们已经将 `webserver-cluster` 部署到静态编码的默认 VPC 和子网，但通过添加 `subnet_ids` 变量，将允许模块部署到任何 VPC 或子网。接下来的 `target_group_arns` 和 `health_check_type` 变量，配置如何将 ASG 集成到负载均衡器。与之前在 `webserver-cluster` 中内置的 ALB 代码块相比，`asg-rolling-deploy` 现在是一个通用的模块，通过输入变量提供给负载均衡器设置参数，可以在多种环境下使用相同的 ASG 代码。例如，不使用负载均衡器、一个 ALB、多个 NLB 等环境。

将这3个输入变量传递给 `modules/cluster/asg-rolling-deploy/main.tf` 文件中的 `aws_autoscaling_group` 资源，代替之前直接指向资源（如 ALB）和数据源（如 `aws_subnet_ids`）的静态编码设置，并且不需要将那些静态设置复制到新的 `asg-rolling-deploy` 模块中。

```
resource "aws_autoscaling_group" "example" {
  # 这里有意地将 ASG 名称依赖于启动配置的名称，每次启动配置被替换时，ASG 也会被强行替换
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"

  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier = var.subnet_ids

  # 与负载均衡器进行集成
  target_group_arns = var.target_group_arns
  health_check_type = var.health_check_type

  min_size = var.min_size
  max_size = var.max_size

  # 至少要等待这么多数目的实例通过运行状况检查，才能认为 ASG 完成了部署
  min_elb_capacity = var.min_size

  # ...
}
```

第4个变量 `user_data` 用于传递用户数据脚本。之前的 `webserver-cluster` 模块中的静态编码的 User Data 脚本，只能用于部署“Hello,World”应用程序。通过将 User Data 脚本作为输入变量，`asg-rolling-deploy` 模块可以将任何应用部署在 ASG 之上。在此将

`user_data` 变量传递给 `aws_launch_configuration` 资源（替换我们并未复制到 `asg-rolling-deploy` 模块中的 `template_file` 数据源的引用）。

```
resource "aws_launch_configuration" "example" {
    image_id      = var.ami
    instance_type = var.instance_type
    security_groups = [aws_security_group.instance.id]
    user_data      = var.user_data

    # 与自动缩放组一起使用启动配置时需要
    # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
    lifecycle {
        create_before_destroy = true
    }
}
```

需要将一些有用的输出变量添加到 `modules/cluster/asg-rolling-deploy/outputs.tf` 文件中。

```
output "asg_name" {
    value      = aws_autoscaling_group.example.name
    description = "The name of the Auto Scaling Group"
}

output "instance_security_group_id" {
    value      = aws_security_group.instance.id
    description = "The ID of the EC2 Instance Security Group"
}
```

输出这些数据使 `asg-rolling-deploy` 模块更有利于重用，模块的使用者可以通过这些输出变量实现新功能，例如将自定义规则附加到安全组。

出于类似的原因，你应该将另外几个输出变量添加到 `modules/networking/alb/outputs.tf` 文件中。

```
output "alb_dns_name" {
    value      = aws_lb.example.dns_name
    description = "The domain name of the load balancer"
}

output "alb_http_listener_arn" {
    value      = aws_lb_listener.http.arn
    description = "The ARN of the HTTP listener"
```

```
}

output "alb_security_group_id" {
  value      = aws_security_group.alb.id
  description = "The ALB Security Group ID"
}
```

你将很快学习如何使用它们。

最后一步是将 `webserver-cluster` 模块转换为 `hello-world-app` 模块，新模块将调用 `asg-rolling-deploy` 和 `alb` 模块，来完成“Hello,World”应用程序的部署。为此，请将 `module/services/webserver-cluster` 目录重命名为 `module/services/hello-world-app`。在完成上述步骤中的所有更改之后，在 `module/services/hello-world-ap/main.tf` 文件中应该仅存在以下资源和数据源。

- `template_file`（用于用户数据）
- `aws_lb_target_group`
- `aws_lb_listener_rule`
- `terraform_remote_state`（用于数据库）
- `aws_vpc`
- `aws_subnet_ids`

将以下输入变量添加到 `modules/services/hello-world-app/variables.tf` 文件中。

```
variable "environment" {
  description      = "The name of the environment we're deploying to"
  type            = string
}
```

现在，请将之前创建的 `asg-rolling-deploy` 模块添加到 `helloworld-app` 模块中，来部署 ASG。

```
module "asg" {
  source      = "../../cluster/asg-rolling-deploy"

  cluster_name = "hello-world-${var.environment}"
  ami          = var.ami
  user_data    = data.template_file.user_data.rendered
}
```

```
instance_type      = var.instance_type  
  
min_size          = var.min_size  
max_size          = var.max_size  
enable_autoscaling = var.enable_autoscaling  
  
subnet_ids        = data.aws_subnet_ids.default.ids  
target_group_arns = [aws_lb_target_group.asg.arn]  
health_check_type = "ELB"  
  
custom_tags        = var.custom_tags  
}
```

并添加你之前创建的 alb 模块到 hello-world-app 模块中，来部署 ALB。

```
module "alb" {  
  source           = "../../networking/alb"  
  alb_name         = "hello-world-${var.environment}"  
  subnet_ids       = data.aws_subnet_ids.default.ids  
}
```

通过使用输入变量 environment 强化命名规则，所有资源将根据环境来命名（如 `hello-world-stage`、`hello-world-prod`）。代码还为前面新添加的变量设置了合适的输入值，包括 `subnet_ids`、`target_group_arns`、`health_check_type` 和 `user_data`。

接下来，你需要为应用程序配置 ALB 目标组和侦听器规则。更新 `modules/services/hello-world-app/main.tf` 文件中的 `aws_lb_target_group` 资源，将 `environment` 变量作为 `name` 参数的后缀。

```
resource "aws_lb_target_group" "asg" {  
  name           = "hello-world-${var.environment}"  
  port           = var.server_port  
  protocol       = "HTTP"  
  vpc_id         = data.aws_vpc.default.id  
  
  health_check {  
    path          = "/"  
    protocol     = "HTTP"  
    matcher      = "200"  
    interval     = 15  
    timeout      = 3
```

```
    healthy_threshold  = 2
    unhealthy_threshold = 2
}
}
```

现在，更新 `aws_lb_listener_rule` 资源的 `listener_arn` 参数，使用来自 ALB 模块的输出变量 `aws_lb_listener_arn`。

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn      = module.alb.alb_http_listener_arn
  priority          = 100

  condition {
    field      = "path-pattern"
    values     = ["*"]
  }

  action {
    type        = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```

最后，将 `asg-rolling-deploy` 模块和 `alb` 模块中重要的输出变量，再次定义为 `hello-world-app` 模块的输出变量。

```
output "alb_dns_name" {
  value      = module.alb.alb_dns_name
  description = "The domain name of the load balancer"
}

output "asg_name" {
  value      = module.asg.asg_name
  description = "The name of the Auto Scaling Group"
}

output "instance_security_group_id" {
  value      = module.asg.instance_security_group_id
  description = "The ID of the EC2 Instance Security Group"
}
```

这正是函数组合的工作方式：从较简单的部分（ASG 和 ALB 模块）开始，构建更复杂的行为（“Hello,World”应用程序）。Terraform 领域中一个相当普遍的模式是至少会用到两种类型的模块。

#### 通用模块

诸如 `asg-rolling-deploy` 模块和 `alb` 模块，是代码的基本组成部分，在各种不同用例中可以重复使用。你已经看到如何使用它们部署“Hello,World”应用程序了，还可以使用这两个模块部署完全不同的内容。例如，一个用于运行 Kafka 集群的 ASG，或者可以在多应用之间分配负载的完全独立的 ALB（为所有应用程序运行一个 ALB 比为每个应用程序运行各自的 ALB 要节省资源）。

#### 特定用例模块

诸如 `hello-world-app` 之类的模块，将多个通用模块组合在一起，服务一个特定的用例，例如部署“Hello,World”应用程序。

在实际使用中，为了更好地支持组合和重用，可能需要进一步细分模块。例如，为了运行 HashiCorp Consul，可以使用一组来自 `terraform-aws-consul` 代码库（见参考资料第 6 章[1]）的开源的可重用模块。Consul 是一个开放源代码、分布式、键值存储库，有时需要侦听大量不同端口（如服务器 RPC、CLI RPC、Serf WAN、HTTP API、DNS 等）上的网络请求，因此有时会使用多达 20 个安全组规则。在 `terraform-aws-consul` 仓库中，这些安全组规则被定义在一个独立的 `consul-security-group-rules` 模块（见参考资料第 6 章[2]）中。

为了理解这样划分的原因，首先要了解 Consul 是如何部署的。一个常见的用例是将 Consul 部署为 HashiCorp Vault 的数据存储，HashiCorp Vault 是一种开放源代码、分布式机密存储系统，可用于安全地存储密码、API 密钥、TLS 证书等。你可以在 `terraform-aws-vault` 代码库（见参考资料第 6 章[3]）中找到不少开源的、可重用的模块用于运行 Vault，图 6-3 显示了典型生产环境下的 Vault 体系架构。

## Vault Architecture

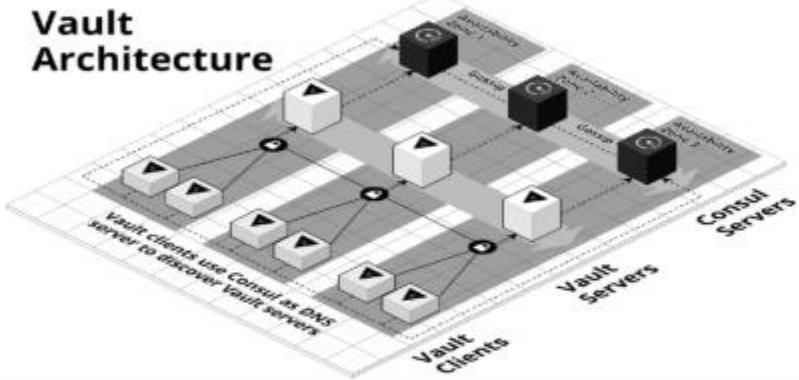


图 6-3：典型生产环境下的 Vault 体系架构

在生产环境中，通常在一组包含 3 到 5 台服务器的 ASG 上运行 Vault（通过部署 `vault-cluster` 模块，见参考资料第 6 章[4]），而 Consul 则运行在另一组包含 3 到 5 台服务器的 ASG 上（通过部署 `consul-cluster` 模块，见参考资料第 6 章[5]），因此这两组独立的 ASG 可以分别进行缩放和安全管理。但是，在预发布环境中，运行这么多服务器是浪费的，为了节省开销可以在一个 ASG（也许大小为 1）上同时运行 Vault 和 Consul，ASG 也是通过 `vault-cluster` 模块部署的。如果所有 Consul 的安全组规则定义在 `consul-cluster` 模块内部，那么通过 `vault-cluster` 模块部署 Consul 时，将无法重用它们（除非手动复制/粘贴 20 多个规则）。所以将规则单独定义在 `consul-security-group-rules` 模块中，这样可以将其直接附加到 `vault-cluster` 或几乎任何其他类型的集群上。

对于一个简单的“Hello,World”应用程序而言，这种功能分解可能有些过度。但对于复杂的现实世界的基础设施而言，将安全组规则、IAM 策略和其他跨领域问题分解为单独的模块，对支持不同的部署模式是十分重要的。使用这种模式的不仅有 Consul 和 Vault，还有 ELK 系统（在 Prod 环境中使用单独集群运行 Elasticsearch、Logstash 和 Kibana，而在 Dev 环境中使用同一集群）、Confluent 平台（在 Prod 环境中使用不同的集群运行 Kafka、ZooKeeper、REST Proxy 和 Schema Registry，但在 Dev 环境中使用相同的集群）、TICK 系统（在 Prod 中使用不同集群运行 Telegraf、InfluxDB、Chronograf 和 Kapacitor，但在 Dev 环境中使用相同集群运行），以此类推。

## 可测试的模块

到此为止，我们已经以 3 个模块的形式编写了很多代码：`asg-rolling-deploy`、`alb` 和 `hello-world-app`。下一步是检查代码是否真正工作。

这里创建的模块不是直接可以部署的根模块。为了测试部署它们，需要编写一些 Terraform 代码，处理插入所需的参数、设置 `provider`、配置 `backend` 等工作。一种清晰的方法是，创建一个 `example` 文件夹。顾名思义，该文件夹将展示如何使用这个模块的示例。让我们尝试一下。

创建具有以下内容的 `examples/asg/main.tf` 文件。

```
provider "aws" {
    region      = "us-east-2"
}

module "asg" {
    source      = "../../modules/cluster/asg-rolling-deploy"

    cluster_name = var.cluster_name
    ami          = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"

    min_size      = 1
    max_size      = 1
    enable_autoscaling = false

    subnet_ids    = data.aws_subnet_ids.default.ids
}

data "aws_vpc" "default" {
    default      = true
}

data "aws_subnet_ids" "default" {
    vpc_id        = data.aws_vpc.default.id
}
```

这段代码使用 `asg-rolling-deploy` 模块，部署了一个大小为 1 的 ASG。通过运行 `terraform init` 和 `terraform apply` 命令，来实际检查它在运行时会不会出现错误，是否真的能够运行起来一个 ASG。现在，添加一个 `README.md` 文件来包含这些指令。这个小小的示例将发挥巨大的作用。在仅有的几个文件和若干行代码中，你实现了如下内容。

## 手动测试工具

当开发 `asg-rolling-deploy` 模块时，基于这段示例代码，可以通过手动方式，反复运行 `terraform apply` 和 `terraform destroy` 命令，检查它是否按预期工作。

## 自动测试工具

正如你将在第 7 章中看到的，示例代码和为模块创建自动测试的方法是一样的。我通常建议将测试放入 `test` 文件夹。

## 可执行文档

如果将此示例（包括 `README.md`）提交到版本控制系统中，则团队的其他成员可以通过它来了解模块的工作原理，并在不编写代码的情况下就可以试用模块。这既是培训团队其他成员的一种方式，又可以通过添加自动测试来确保“教材”始终按预期工作。

你在 `modules` 文件夹中拥有的每个 Terraform 模块，都应在 `examples` 文件夹中有一个相对应的示例，并且 `examples` 文件夹中的每个示例都应在 `test` 文件夹中有一个相对应的测试。实际上，每个模块可能有多个示例（因此，有多个测试）来展示该模块的不同配置和排列组合方式。例如，为 `asg-rolling-deploy` 模块添加其他的示例，展示如何将它与自动缩放策略一起使用、如何将负载均衡器连接到该模块、如何设置自定义标签，等等。

把所有这些组合起来，一个典型的模块文件夹结构将是这个样子的。

```
modules
  examples
    alb
    asg-rolling-deploy
      one-instance
      auto-scaling
      with-load-balancer
      custom-tags
    hello-world-app
    mysql
  modules
    alb
    asg-rolling-deploy
    hello-world-app
    mysql
  test
    alb
    asg-rolling-deploy
    hello-world-app
    mysql
```

下面是一个留给读者的练习，请自己决定如何将之前编写的 RDS 代码转换为 MySQL 模块，并为之添加多个关于 `alb`、`asg-rolling-deploy`、`mysql` 和 `hello-world-app` 的模块示例。

开发新模块时，可以遵循的一种很好的做法是：在编写第 1 行模块代码之前，首先编写示例代码。如果从实现模块代码开始，则很容易迷失在实现细节中，而当你重新审视 API 时，最终会得到一个不直观且难以使用的模块。如果从示例代码开始，你可以充分考虑最优的用户体验，为模块定义一个简洁的 API，然后再进行模块的实现。示例代码无论如何都是测试模块的主要方式，所以这也是测试驱动开发（TDD）的一种形式。我将进一步在第 7 章中介绍有关测试的内容。

一旦开始定期对模块进行测试，你就会发现另一个非常有用的做法：版本固定（*version pinning*）。你应该在所有 Terraform 模块中，通过 `required_version` 参数，调用特定的 Terraform 版本。至少需要设定 Terraform 的主要版本号。

```
terraform {
    # 要求使用任何 0.12.x 版本的 Terraform
    REQUIRED_VERSION = ">= 0.12 <0.13"
}
```

上面的代码将允许在使用该模块时，只调用 Terraform 的 0.12.x 版本，而不能是其他的，例如 0.11.x 和 0.13.x。这是个很关键的问题，因为 Terraform 的每个主要发行版并不向后兼容；例如，从 0.11.x 升级到 0.12.x 需要进行许多代码更改，升级版本需要专门的计划。通过添加 `required_version` 参数，可以在使用其他版本运行 `terraform apply` 命令时，提示错误信息。

```
$ terraform apply

Error: Unsupported Terraform Core version

This configuration does not support Terraform version 0.11.11. To proceed,
either choose another supported Terraform version or update the root module's
version constraint. Version constraints are normally set for good reason, so
updating the constraint may lead to other errors or unexpected behavior.
```

对于生产级代码，我建议执行更严格的版本固定。

```
terraform {
```

```
# 需要 Terraform 0.12.0 版本来运行模块  
required_version = "= 0.12.0"  
}
```

这样做的原因是，即使只是补丁版本号的提升（如 0.12.0→0.12.1）也可能会引起问题。有时是出了错误，有时是向后不兼容的问题（尽管最近很少见）。更大的问题是，一旦用新版本的 Terraform 改写了 Terraform 状态文件，该状态文件与任何旧版本的 Terraform 就不再兼容了。假设现在所有代码都在使用 Terraform 0.12.0 进行部署，有一天一个开发人员无意间安装了 0.12.1 版本，并在一些模块上运行了 `terraform apply` 命令。这些模块的状态文件现在不能再和 0.12.0 版本的 Terraform 一起工作了，因此不得不将所有开发人员的计算机和所有 CI (*Continuous Integration*, 持续集成) 服务器的 Terraform 更新为 0.12.1 版本！

当 Terraform 达到 1.0.0 版本，并开始强制向后兼容时，这种情况可能会有所好转。但是在这之前，我还是建议将代码锁定在确切的 Terraform 版本上。这样，就不会发生意外更新，而是选择计划好的时间进行更新。当进行更新时，同时更新所有代码、所有开发人员的计算机和所有 CI 服务器。

我还建议锁定所有的提供商程序版本。

```
provider "aws" {  
    region = "us-east-2"  
  
    # 允许任意 2.x 版本的 AWS 提供商  
    version = "-> 2.0"  
}
```

此代码将 AWS 提供商程序锁定到任何的 2.x 版本（`->2.0` 语法等效于`>=2.0,<3.0`）。需要再次重申，至少应该锁定主版本号，避免意外引入向后不兼容的更改。是否需要将提供商程序锁定到确切的修订版本，取决于提供商的水平。例如，AWS 提供商程序经常更新，并且在向后兼容性方面保持得很好，因此只需要锁定主要版本，允许自动获取新的修订版本，便于轻松访问新功能即可。每个提供商程序都是不同的，因此请关注他们在保持向后兼容性方面的水平，来决定相应的版本锁定策略。

## 可发布的模块

编写并测试模块之后，下一步就是发布它们。正如在第 4 章的“模块版本控制”中所看到的，可以使用符合语义版本的 Git 标签，如下所示。

```
$ git tag -a "v0.0.5" -m "Create new hello-world-app module"  
$ git push --follow-tags
```

例如，为了部署 `hello-world-app` 模块的 v0.0.5 版本到预发布环境，请把下列代码更新到 `live/stage/services/hello-world-app/main.tf` 文件中。

```
provider "aws" {  
    region = "us-east-2"  
  
    # 允许任意 2.x 版本的 AWS 提供商  
    version = "~> 2.0"  
}  
  
module "hello_world_app" {  
    # TODO: 用你自己的模块 URL 和版本替换它  
    source = "git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"  
  
    server_text          = "New server text"  
    environment         = "stage"  
    db_remote_state_bucket = "(YOUR_BUCKET_NAME)"  
    db_remote_state_key   = "stage/data-stores/mysql/terraform.tfstate"  
  
    instance_type        = "t2.micro"  
    min_size             = 2  
    max_size             = 2  
    enable_autoscaling   = false  
}
```

接下来，将 ALB DNS 名称作为输出变量来传递，更新 `live/stage/services/hello-world-app/outputs.tf` 文件。

```
output "alb_dns_name" {  
    value      = module.hello_world_app.alb_dns_name  
    description = "The domain name of the load balancer"  
}
```

现在，通过运行 `terraform init` 和 `terraform apply` 命令，部署版本控制下的模块。

```
$ terraform apply
...
Apply complete! Resources: 13 added, 0 changed, 0 destroyed.

Outputs:
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

如果运行良好，则可以继续将完全相同的版本（完全相同的代码）部署到其他环境，包括生产环境。如果遇到问题，版本控制可以让你选择重新部署旧版本。

发布模块的另一种方法是，将它们发布到 Terraform 注册中心。公共的 Terraform 注册中心位于参考资料第 6 章[6]，其中包括数百个可重复使用的、社区维护的开源模块，适用于 AWS、Google Cloud、Azure 和许多其他提供商。将模块发布到公共的 Terraform 注册中心有以下要求。<sup>1</sup>

- 该模块必须存放在公共 GitHub 存储库。
- 存储库必须遵循命名规范 `terraform-<PROVIDER>-<NAME>`，其中 `PROVIDER` 指定模块的目标提供商（如 `aws`），而 `NAME` 是模块的名称（如 `vault`）。
- 模块必须遵循特定的文件结构，包括在存储库的根目录中定义 Terraform 代码、提供 `README.md`、使用 `main.tf`、`variables.tf` 和 `outputs.tf` 等约定文件名。
- 代码库必须使用遵循语义版本规则的 Git 标签（`x.y.z`）来进行发布。

如果你的模块满足这些要求，则可以通过使用 GitHub 账户登录到 Terraform 注册中心，使用 Web UI 发布该模块，达到与他人共享的目的。当模块发布到注册中心后，将拥有一个漂亮的界面来显示模块细节，如图 6-4 所示为 Terraform 注册中心中的 HashiCorp Vault 模块。

Terraform 注册中心可以自动解析模块的输入和输出，因此那些输入变量和输出变量也将显示在界面中，包括 `type` 和 `description` 字段，如图 6-5 所示。

---

<sup>1</sup> 你可以在参考资料第 6 章[7]找到有关模块发布的完整详细信息。

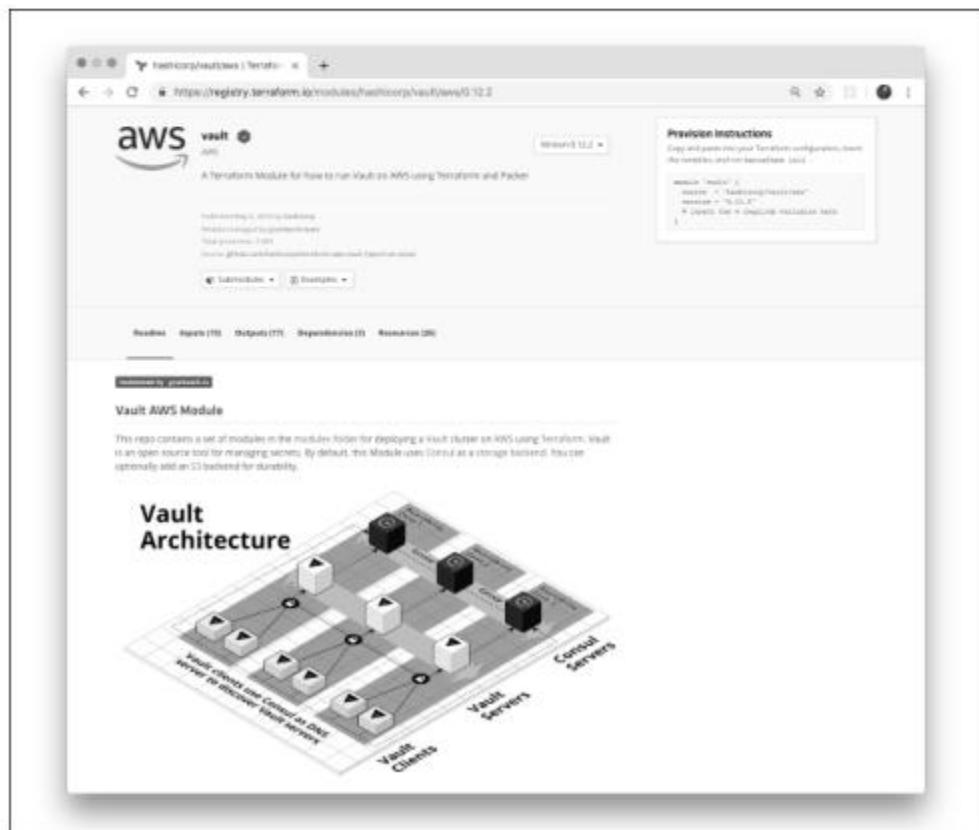


图 6-4：Terraform 注册中心中的 HashiCorp Vault 模块

Terraform 甚至支持使用特殊语法，直接使用 Terraform 注册中心中的模块，可以避免那些带有隐晦的 `ref` 参数的冗长的 Git 地址。通过在 `source` 参数中使用较短的注册中心 URL，在 `version` 参数中指定版本，语法规则如下。

```
module "<NAME>" {
  source  = "<OWNER>/<REPO>/<PROVIDER>"
  version = "<VERSION>"

  # ...
}
```

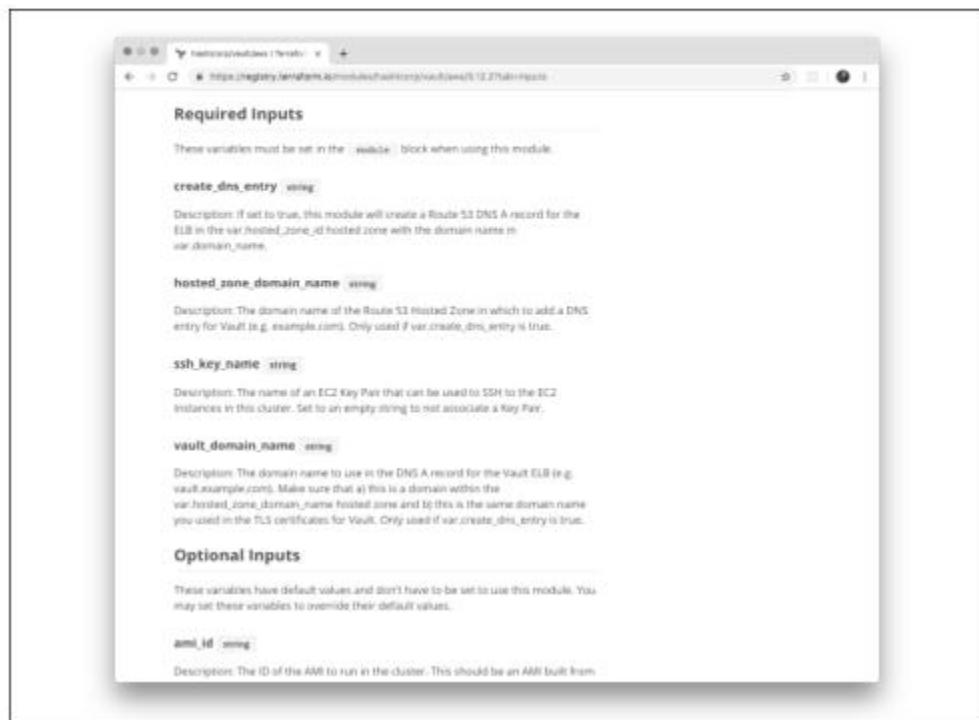


图 6-5：Terraform 注册中心会自动解析并显示模块的输入变量和输出变量

其中 NAME 是在 Terraform 代码中用于引用模块的标识符，OWNER 是 GitHub 存储库的所有者（例如，在 `github.com/foo/bar` 中，所有者是 `foo`），REPO 是 GitHub 存储库的名称（例如，在 `github.com/foo/bar` 中，`repo` 名称是 `bar`），PROVIDER 是代码提供商（例如 `aws`），VERSION 是要使用的模块的版本。这是一个使用来自 Terraform 注册中心的 Vault 模块的示例。

```
module "vault" {
  source  = "hashicorp/vault/aws"
  version = "0.12.2"

  # ...
}
```

如果你是 HashiCorp Terraform 企业版客户，可以在私有 Terraform 注册中心获得相同的体

验。也就是说，注册中心位于你的私有 Git 存储库中，并且只能由你的团队访问。这是在公司内共享模块的好方法。

## Terraform 模块之外的内容

尽管本书主要涉及 Terraform，但是要构建整个生产级基础设施，你还需要使用其他工具，例如 Docker、Packer、Chef、Puppet，当然还有 DevOps 领域的胶带、胶水、主力、值得信赖的 Bash 脚本。这些代码大多数都可以直接和 Terraform 代码一起驻留在 `modules` 文件夹中。例如，之前看到的 HashiCorp Vault 存储库中，`modules` 文件夹不仅包含 Terraform 代码，如前面提到的 `vault-cluster` 模块；还包括 Bash 脚本，如运行的 `run-vault` 脚本（见参考资料第 6 章[8]），它可以在 Linux 服务器引导期间（通过用户数据植入）配置和启动 Vault。

有时可能需要更进一步，直接从 Terraform 模块中，运行一些非 Terraform 代码（如脚本）。有时是为了将 Terraform 与另一个系统进行集成（例如，我们之前通过 Terraform 配置，在 EC2 实例上执行的用户数据脚本）。有时是为了解决 Terraform 的局限性，例如，缺少提供商程序 API 或由于 Terraform 的声明性语法而无法实现的复杂逻辑。如果仔细搜索，你可以在 Terraform 中找到一些可以称为“逃生舱口”的功能来救急，它们包括如下内容。

- 预配器（provisioners）
- `null_resource` 中的预配器
- 外部数据源

下面让我们逐一进行学习。

### 预配器

通过 Terraform 预配器，Terraform 可以执行本地计算机或远程计算机上的脚本，通常用于执行引导、配置管理或清理工作。几种不同的预配器包括 `local-exec`（在本地计算机上执行脚本）、`remote-exec`（在远程资源上执行脚本）、`chef`（在远程资源上运行 Chef Client）和 `file`（复制文件到远程资源）。<sup>1</sup>

---

<sup>1</sup> 你可以在参考资料第 6 章[9]找到预配器的完整列表。

通过将 `provisioner` 代码块添加到资源中来实现预配器。例如，下面是使用 `local-exec` 预配器在本地计算机上执行脚本的方法。

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command      = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

对上面的代码运行 `terraform apply` 命令时，将打印 “Hello,World from”，然后使用 `uname` 命令输出本地操作系统详细信息。

```
$ terraform apply
...
aws_instance.example (local-exec): Hello, World from Darwin x86_64 i386
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

接下来让我们尝试一些复杂的 `remote-exec` 预配器。要在远程资源（如 EC2 实例）上执行代码，Terraform 客户端必须能够执行以下操作。

#### 通过网络与 EC2 实例进行通信

我们已经学习了如何通过配置安全组来实现这个要求。

#### 访问 EC2 实例的身份验证

`remote-exec` 预配器支持 SSH 和 WinRM 连接。由于将启动的 EC2 实例为 Linux (Ubuntu)，因此需要使用 SSH 身份验证，这也意味着需要配置 SSH 密钥。

让我们通过创建一个安全组，允许端口 22 的入站连接（22 端口是 SSH 的默认端口）。

```
resource "aws_security_group" "instance" {
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
```

```
# 为了使这个例子很容易试用，我们允许所有 SSH 连接
# 在实际使用中，应将其锁定为唯一受信任的 IP
cidr_blocks = ["0.0.0.0/0"]
}
}
```

使用 SSH 密钥通常的过程是，在本地计算机上生成 SSH 密钥对，将公钥上传到 AWS，将私钥存储在安全的地方，供 Terraform 代码访问。为了使代码示例更加简单，这里我们将使用 `tls_private_key` 资源自动生成私钥。

```
# 为了使本示例易于尝试，我们在 Terraform 中生成了私钥
# 在实际使用中，你应该在 Terraform 之外管理 SSH 密钥
resource "tls_private_key" "example" {
  algorithm  = "RSA"
  rsa_bits   = 4096
}
```

此私有密钥将被存储在 Terraform 状态文件中，这不适用于生产环境用例，而只是为了本书的练习而进行的简化。接下来，使用 `aws_key_pair` 资源将公钥上传到 AWS。

```
resource "aws_key_pair" "generated_key" {
  public_key = tls_private_key.example.public_key_openssh
}
```

终于，让我们开始编写 EC2 实例代码。

```
resource "aws_instance" "example" {
  ami                      = "ami-0c55b159cbfafe1f0"
  instance_type            = "t2.micro"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name                 = aws_key_pair.generated_key.key_name
}
```

这段代码的前几行应该很熟悉了：它是将 Ubuntu 的亚马逊机器映像（AMI）部署在一台 `t2.micro` 类型的 EC2 上，并让你前面创建的安全组关联这个实例。唯一的新项目是，使用 `key_name` 属性来指示 AWS 将你的公钥与此 EC2 实例相关联。AWS 会自动将指定的公钥添加到服务器实例的 `authorized_keys` 文件中，这将允许你使用相应的私钥 SSH 登录到该服务器。

接下来，让我们向该 EC2 实例添加 `remote-exec` 预配器。

```

resource "aws_instance" "example" {
  ami                  = "ami-0c55b159cbfafe1f0"
  instance_type        = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name             = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }
}

```

`remote-exec` 预配器看起来与 `local-exec` 预配器几乎相同，不同之处是使用 `inline` 参数传递要执行的命令列表，而不是单个 `command` 参数。最后，还需要将 Terraform 配置为在运行 `remote-exec` 预配器时，使用 SSH 方式连接到 EC2 实例。你可以通过 `connection` 代码块来实现。

```

resource "aws_instance" "example" {
  ami                  = "ami-0c55b159cbfafe1f0"
  instance_type        = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name             = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }

  connection {
    type      = "ssh"
    host     = self.public_ip
    user     = "ubuntu"
    private_key = tls_private_key.example.private_key_pem
  }
}

```

`connection` 代码块指示 Terraform 利用 SSH 连接到 EC2 实例的公网 IP 地址。`ubuntu` 作为用户名（这是在 Ubuntu 的 AMI 中默认的 `root` 用户名），`private_key` 为自动生成的私钥。请注意 `host` 参数里的 `self` 关键字设置。`Self` 表达式使用以下语法。

```
self.<ATTRIBUTE>
```

通过在 `connection` 和 `provisioner` 代码块中使用这个特殊语法，引用外层资源输出的

`ATTRIBUTE`。如果尝试使用标准的 `aws_instance.example.<ATTRIBUTE>` 语法，则会遇到循环依赖项错误，因为资源是无法引用其自身的。因此 `self` 表达式是为预配器专门添加解决方案的。

如果在此代码上运行 `terraform apply` 命令，你将看到以下内容。

```
$ terraform apply

(...)

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Provisioning with 'remote-exec'...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...

(... repeats a few more times ...)

aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connected!
aws_instance.example (remote-exec): Hello, World from Linux x86_64 x86_64

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

`remote-exec` 预配器并不知道什么时候 EC2 实例将被启动并准备接受连接。所以它会一直重试 SSH 连接，直到成功或超时（默认的超时时间为 5min，但你可以配置它）。最终连接成功，从服务器获得“Hello,World”的返回结果。

在默认情况下定义一个预配器时，它是一个创建时预配器（*creation-time provisioner*），这意味着它在运行 `terraform apply` 命令时运行，并且仅在资源的初始创建期间运行。该预配器在随后 `terraform apply` 命令重复的运行中，不会被再次调用。因此创建时预配器主要用于运行初始启动绑定代码。如果在预配器上设置了 `when="destroy"` 参数，它将是销毁时预配器，它将在运行 `terraform destroy` 之后，在删除资源之前运行。

同一资源上可以定义多个预配器，Terraform 将从上到下依次运行它们。可以通过 `on_failure` 参数指示 Terraform 如何处理来自预配器的错误：如果设置为“`continue`”，则

Terraform 将忽略该错误并继续进行资源创建或销毁；如果设置为“abort”，Terraform 将中止资源的创建或销毁。

### 预配器与用户数据的对比

通过 Terraform 在服务器上执行脚本的两种不同方法：一种是使用 `remote-exec` 预配器，另一种是使用用户数据脚本。由于以下原因，我发现用户数据脚本是更有用的工具。

- `remote-exec` 预配器需要开通服务器的 SSH 或 WinRM 访问，管理起来更加复杂（如前面介绍的安全组和 SSH 密钥），并且不够安全。相比之下用户数据仅需要 AWS API 访问权限（这也是使用 Terraform 所必须的权限）。
- 用户数据脚本可以与 ASG 一起使用，确保 ASG 中的所有服务器在引导期间都统一执行脚本，包括由于自动扩展或自动恢复事件而启动的新的服务器。而预配器仅在 Terraform 运行时生效，并且根本无法与 ASG 一起使用。
- 用户数据脚本可以通过 EC2 控制台查看（选择一个实例，单击“操作”→“实例设置”→“查看/更改用户数据”），也可以在 EC2 实例中找到执行日志（通常在`/var/log/cloud-init*.log`），这两个特性对调试很有帮助，而预配器都不具备。

预配器仅有的优点如下。

- 用户数据脚本的长度限制为 16 KB，而预配器脚本的长度没有限制。
- Chef、Puppet 和 Salt 预配器分别在服务器上自动安装、配置和运行 Chef、Puppet 和 Salt 客户端。这样可以更轻松地使用配置管理工具，代替临时脚本来配置服务器。

### null\_resource 中的预配器

预配器只能定义在资源内部，但是有时我们会希望执行无须绑定到特定资源的预配器。使用名为 `null_resource` 的资源可以达到这个目的。`null_resource` 类似于普通的 Terraform 资源，但它不会创建任何内容。通过在 `null_resource` 中定义预配器，可以将脚本作为 Terraform 生命周期的一部分来运行，但无须附加到任何真实资源。

```
resource "null_resource" "example" {
```

```
provisioner "local-exec" {
  command = "echo \"Hello, World from $(uname -smp)\""
}

}
```

`null_resource` 资源甚至有一个很方便的参数叫作 `triggers`, 这个参数接受键值对的映射类型。每当数值发生更改时, `null_resource` 将被重新创建, 所有预配器也将被再次执行。例如, 想在每次运行 `terraform apply` 命令时都执行一次 `null_resource` 中的预配器, 可以在 `triggers` 参数中使用 `uuid()` 内置函数, 该函数每次返回一个新的随机生成的 UUID。

```
resource "null_resource" "example" {
  # 使用 UUID 强制此 null_resource 将在每次调用"terraform apply"时重新创建
  triggers      = {
    uuid        = uuid()
  }

  provisioner "local-exec" {
    command      = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

现在, 每次调用 `terraform apply` 命令时, `local-exec` 预配器将都被执行。

```
$ terraform apply
(...)

null_resource.example (local-exec): Hello, World from Darwin x86_64 i386

$ terraform apply

null_resource.example (local-exec): Hello, World from Darwin x86_64 i386
```

## 外部数据源

预配器通常是执行 Terraform 脚本的首选, 但它们并不总是合适的。有时, 你真正想要做的是: 通过执行脚本来获取一些数据, 并使这些数据对 Terraform 代码本身可用。这时应该使用 `external` 数据源, 该数据源允许通过外部命令实现特定协议, 从而充当数据源。

协议如下。

- 通过 `external` 数据源的 `query` 参数，将查询数据从 Terraform 传递到外部程序。外部程序可以从标准输入（`stdin`）中，将这些参数以 JSON 格式读取。
- 外部程序将数据以 JSON 格式写入标准输出（`stdout`），传递回 Terraform。之后，其余 Terraform 代码可以使用 `external` 数据源的 `result` 输出属性，提取 JSON 格式的数据。

例如如下代码。

```
data "external" "echo" {
  program      = ["bash", "-c", "cat /dev/stdin"]
  query       = {
    foo  = "bar"
  }
}
output "echo" {
  value      = data.external.echo.result
}
output "echo_foo" {
  value      = data.external.echo.result.foo
}
```

在这个例子中，使用 `external` 数据源执行 Bash 脚本，脚本只是简单地把在 `stdin` 接收的数据返回到 `stdout`。因此，我们通过 `query` 参数传递的任何数据都应按原样，通过 `result` 输出属性返回。这是运行 `terraform apply` 命令时的结果。

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

echo = {
  "foo" = "bar"
}
echo_foo = bar
```

可以看到 `data.external.<NAME>.result` 中包含外部程序返回的 JSON 结构，并且可

以使用相应语法在该 JSON 中进行查询：`data.external.<NAME>.result.<PATH>`（如 `data.external.echo.result.foo`）。

如果 Terraform 代码需要访问某些数据，但是并没有已知数据源可以用来检索该数据。`external` 数据源是一个不错的急救方案。但是，也请谨慎使用 `external` 数据源和所有其他的 Terraform “急救方案”，因为它们会降低可移植性，使代码变得脆弱。例如，你刚刚看到的 `external` 数据源代码依赖于 Bash，这意味着 Terraform 模块无法通过 Windows 进行部署。

## 小结

现在，你已经了解了创建生产级 Terraform 代码的所有要素，是时候将它们组合在一起了。当你开始开发一个新的模块时，请使用以下过程。

1. 浏览生产级的基础设施检查清单（表 6-2），明确标出想要实现的内容，以及打算跳过的内容。使用这个检查结果再配合表 6-1，可以为你的老板提供一个准确的项目时间估计。
2. 创建一个 `examples` 文件夹并首先编写示例代码，定义模块的最佳用户体验和最简洁的 API。为模块的每个重要使用方法创建一个示例，并包括足够的文档和合理的默认值，使示例尽可能易于部署。
3. 创建一个 `modules` 文件夹，通过编写一组小型化的、可重用的、可组合的模块，开始实施之前定义的 API。通过 Terraform 和其他工具（如 Docker、Packer 和 Bash）的配合，来实现这些模块。确保锁定 Terraform 和提供商程序版本。
4. 创建一个 `test` 文件夹，为每个示例编写自动测试。

现在是时候探索如何为基础设施代码编写自动测试了，你将在第 7 章进行下一步学习。

# 如何测试 Terraform 代码

在 DevOps 的世界中总是充满了忧虑：对停机的忧虑、对数据丢失的忧虑、对安全问题的忧虑。每次进行更新时，总是担心会产生什么负面影响：在每种环境下，更新都会以相同的方式工作吗？这次更新还会发生服务中断吗？如果真的发生故障，又需要熬到深夜来修复吗？随着公司的发展壮大，风险也越来越高，这使得部署过程更加令人害怕、更容易出错。许多公司试图通过减少部署次数来减轻风险，结果反而是每次部署都变得更大，更容易出现错误。

如果用户以代码形式管理基础设施，则有更好的方法来减轻风险：测试。测试的目的是使用户有信心进行更改。这里的关键词是信心：没有任何形式的测试可以保证代码完全没有错误，因此它更像是一种概率游戏。如果可以将全部基础设施和部署过程捕获为代码，那么就可以在预发布环境中对代码进行测试。如果测试通过，完全相同的代码就有很大的可能在生产环境中正常工作。在这样一个充满忧虑和不确定性的世界中，大概率和信心对用户会有很大的帮助。

本章我将介绍测试基础设施代码的过程，包括手动测试和自动测试。本章的大部分内容与自动测试有关。

- 手动测试
  - 手动测试基础知识
  - 清理测试环境

- 自动测试
- 单元测试
- 集成测试
- 端到端测试
- 其他测试方法

## 手动测试

在考虑如何测试 Terraform 代码时，一个很有帮助的方法是，与通用编程语言（如 Ruby）编写的代码进行类比。比方说，你在 *server.rb* 文件中用 Ruby 编写了一个简单的 Web 服务器。

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

这段代码针对/端点的 URL 请求，发送 200 OK 响应并返回正文 “Hello,World”。对于其他所有 URL 发送 404 响应。用户将如何手动测试这段代码呢？典型的方案是：通过一些额外的代码在本地 localhost 上运行 Web 服务器。

```
# 仅当从 CLI 命令行直接调用此脚本时，命令才会运行
# 如果通过其他文件调用则不会被执行
if __FILE__ == $0
  # 运行在本地主机端口 8000 的服务器
  server = WEBrick::HTTPServer.new :Port => 8000
  server.mount '/', WebServer
```

```
# 通过 Ctrl + C 组合键关闭服务器
trap 'INT' do server.shutdown end

# 启动服务器
server.start
end
```

从命令行执行该文件时，将在端口 8000 上启动一个 Web 服务器。

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO WEBrick 1.3.1
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000
```

接下来可以使用 Web 浏览器或 curl 命令来测试服务器。

```
$ curl localhost:8000/
Hello, World

$ curl localhost:8000/invalid-path
Not Found
```

假设你对此代码进行了更改，添加了对 /api 端点的响应，返回 201 状态和一个 JSON 字段。

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo": "bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

要手动测试更新后的代码，请按 Ctrl+C 组合键终止服务器，并再次运行脚本，重新启动

服务器。

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO WEBrick 1.3.1
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000
^C
[2019-05-25 14:15:54] INFO going to shutdown ...
[2019-05-25 14:15:54] INFO WEBrick::HTTPServer#start done.
```

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO WEBrick 1.3.1
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767 port=8000
```

再次通过 curl 命令测试新的版本。

```
$ curl localhost:8000/api
{"foo": "bar"}
```

## 手动测试基础知识

如何将手动测试的方式应用于 Terraform 代码呢？例如，在前面的章节中，已经创建了用于部署 ALB 的 Terraform 代码。这是来自 *modules/networking/alb/main.tf* 文件的摘录。

```
resource "aws_lb" "example" {
  name            = var.alb_name
  load_balancer_type = "application"
  subnets         = var.subnet_ids
  security_groups = [aws_security_group.alb.id]
}

resource "aws_lb_listener" "http" {
  load_balancer_arn    = aws_lb.example.arn
  port                 = local.http_port
  protocol             = "HTTP"

  # 默认情况下，返回一个简单的 404 页面
  default_action {
    type = "fixed-response"
    fixed_response {
      content_type     = "text/plain"
```

```

    message_body      = "404: page not found"
    status_code       = 404
}
}

resource "aws_security_group" "alb" {
  name          = var.alb_name
}

# ...

```

如果将这段代码与 Ruby 代码进行比较，一个明显的区别是：用户无法在自己的计算机上部署 AWS ALB、目标组、侦听器、安全组，以及所有其他基础设施。

这是测试要点 #1：测试 Terraform 代码时，本地无法模拟服务器环境。

不仅是 Terraform，这个问题同样存在于大多数 IaC 工具中。使用 Terraform 进行手动测试的唯一方法是部署到实际环境（即部署到 AWS）。换句话说，在本书中用户一直使用的方式——运行 `terraform apply` 和 `terraform destroy` 命令，就是手动测试 Terraform 的方法。

如第 6 章所述，这就是必须在每个模块的 `examples` 文件夹中包括易于部署的示例的原因之一。手动测试 `alb` 模块的最简单方法是，使用你在 `examples/alb` 下为其创建的示例代码。

```

provider "aws" {
  region      = "us-east-2"
  # 允许任何 2.x 版本的 AWS 提供商程序
  version     = "~> 2.0"
}

module "alb" {
  source      = ".../modules/networking/alb"

  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnet_ids.default.ids
}

```

正如你在整本书中运行过很多次那样，使用 `terraform apply` 命令部署这个示例代码。

```

$ terraform apply

(...)
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

Outputs:

```
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

部署完成后，可以使用 `curl` 等工具来测试，例如，ALB 的默认响应是返回 404。

```
$ curl \  
-s \  
-o /dev/null \  
-w "%{http_code}" \  
hello-world-stage-477699288.us-east-2.elb.amazonaws.com  
404
```



### 验证基础设施

本章中的示例使用 `curl` 命令和 HTTP 请求来确定基础设施是否正常运行。因为我们正在测试的基础设施是响应 HTTP 请求的负载均衡器。对于其他类型的基础设施，需要使用其他形式的验证方式来代替 `curl` 命令和 HTTP 请求。例如，如果基础设施代码部署了 MySQL 数据库，则需要使用 MySQL 客户端进行验证；如果基础设施代码部署了 VPN 服务器，则需要使用 VPN 客户端进行验证；如果基础设施代码部署的服务器根本不响应外部请求，则需要 SSH 到该服务器，并在服务器本地执行一些命令来进行测试，等等。因此，尽管你可以对任何类型的基础设施使用本章中描述的测试架构，但验证步骤会根据不同的测试内容而有所差别。

提醒一下，ALB 之所以返回 404，是因为它没有配置其他监听器规则，并且 `alb` 模块中的默认操作是返回 404。

```
resource "aws_lb_listener" "http" {  
    load_balancer_arn      = aws_lb.example.arn  
    port                   = local.http_port  
    protocol              = "HTTP"  
  
    # 默认情况下，返回一个简单的 404 页面  
    default_action {  
        type = "fixed-response"
```

```
    fixed_response {
        content_type      = "text/plain"
        message_body     = "404: page not found"
        status_code       = 404
    }
}
}
```

现在有了运行和测试代码的方法，就可以开始进行更改了。每次进行更改时（例如，将默认响应更改为返回 401），都需要重新运行 `terraform apply` 命令部署更新。

```
$ terraform apply

(...)

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Outputs:

alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

还可以通过重新运行 `curl` 命令测试新版本。

```
$ curl \
  -s \
  -o /dev/null \
  -w "%{http_code}" \
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com
401
```

完成测试后，运行 `terraform destroy` 命令来进行环境清理。

```
$ terraform destroy
(...)
```

换句话说，在使用 Terraform 时，每个开发人员都需要通过良好的示例代码来进行测试，并需要一个真实的部署环境（如一个 AWS 账户），以等同于 `localhost` 的方式来运行这些测试。在手动测试过程中，会创建和销毁许多基础设施，在此过程中可能会犯很多错误。因此，测试环境最好与其他更稳定的环境（如预发布环境和生产环境）完全隔离。

强烈建议为每个团队设置一个隔离的沙箱环境，允许开发者按需求创建和删除基础设施，

而不必担心影响其他人。实际上，为了减少多个开发人员之间发生冲突的机会（例如，两个开发人员试图创建一个同名的负载均衡器），黄金法则是每个开发人员都拥有完全隔离的沙箱环境。例如，如果开发针对 AWS 的 Terraform 代码，黄金法则是每个开发人员都拥有自己的 AWS 账户，可以用来测试他们想要的任何东西。<sup>1</sup>

## 清理测试环境

拥有大量的沙箱环境对于提高开发人员的生产力至关重要，但是如果不好好管理，最终可能会存在许多持续运行的基础设施，这不仅会造成混乱，还会产生高额费用。

为了避免出现成本急剧上升而无法控制的情况，测试要点 # 2：定期清理沙箱环境。

至少，应该养成一种习惯，开发人员在完成测试部署后，要通过运行 `terraform destroy` 清理所有内容。在某些部署环境下，也可以使用计划运行工具（如 cron job），自动清除未使用的或旧的资源。以下是一些示例。

### *cloud-nuke*（见参考资料第 7 章[1]）

一个开源工具，可以删除云环境中的所有资源。它支持 AWS 中的许多资源（例如，Amazon EC2 实例、ASG、ELB 等），并且将来还会支持其他资源和其他云（Google Cloud、Azure）。主要功能是能够删除所有超过特定期限的资源。例如，一种常见的模式是，每天在沙箱环境中以 cron 作业方式运行一次 `cloud-nuke`，删除超过两天的所有资源。这是基于一个假设：开发人员为手动测试启动的任何基础设施，超过两天就不再需要了。命令如下。

```
$ cloud-nuke aws --older-than 48h
```

### *Janitor Monkey*（见参考资料第 7 章[2]）

一个开源工具，它通过可配置的时间表清除 AWS 资源（默认每周一次）。通过可配置规则，确定是否应清除资源，甚至可以在删除前几天，将通知发送给资源所有者。这是 Netflix Simian Army 项目的一部分，该项目还包括 Chaos Monkey，一个测试应用程

---

<sup>1</sup> AWS 不会对用户拥有多个 AWS 账户收取任何额外费用。如果你使用 AWS Organizations，则可以创建多个子账户，这些账户将通过主账户统一结算费用。

序弹性的工具。请注意，Netflix Simian Army 项目不再被积极维护，但一些新项目正在取代 Netflix Simian Army 项目的各个部分，例如 Janitor Monkey 被 Swabbie（见参考资料第 7 章[3]）所取代。

#### *aws-nuke*（见参考资料第 7 章[4]）

一个开源工具，专门用于删除 AWS 账户中的所有内容。通过 YAML 配置文件指定 *aws-nuke* 需要清理的账号和资源。

```
# 删 除 区 域
regions:
- us-east-2

# 删 除 账 号
accounts:
"111111111111": {}

# 仅 删除 以 下 资 源
resource-types:
targets:
- S3Object
- S3Bucket
- IAMRole
```

然后按以下方式运行它。

```
$ aws-nuke -c config.yml
```

## 自动测试



### 警告：前方大量代码

为基础设施代码编写自动测试需要很大的勇气。自动测试部分可以说是本书中最复杂的部分，不是简单的通过阅读就能掌握的。如果你的目的只是浏览一下，请跳过这部分内容。如果你想认真学习如何编写测试基础设施的代码，请挽起袖子，准备真正写一些代码！这期间不需要运行任何 Ruby 代码（这只用于帮助构建思考模型），但是你将需要编写和运行尽可能多的 Go 代码。

自动测试的目的是：通过编写测试代码，来验证真实代码是否正常工作。正如将在第 8 章中看到的，可以设置 CI 服务器在每次提交后运行这些测试代码，立即还原或修复任何导致测试失败的提交，从而始终将代码保持在工作状态。

从广义上讲，有 3 种类型的自动测试。

### 单元测试

单元测试可以验证单个的、小型代码的功能。单元（*unit*）的定义各不相同，在通用编程语言中，单元通常是指单个函数或类。所有外部依赖项（例如数据库、Web 服务，甚至文件系统）都将被测试替身（*test doubles*）或模拟（*mocks*）代替，从而使用户可以很好地控制依赖项的行为（例如，通过从数据库模拟返回静态编码的响应），用来测试代码在不同情况下的反应。

### 集成测试

集成测试用来验证多个单元是否可以正常协同工作。在通用编程语言中，集成测试可以验证多个功能或类是否正确地协同工作。集成测试通常会使用真实依赖项和模拟的混合；例如，如果要测试应用程序中与数据库通信的部分，你可能想要使用真实数据库进行测试，但是模拟其他依赖项，例如应用程序的身份验证系统。

### 端到端测试

端到端测试涵盖整个运行体系架构（例如，你的应用程序、数据存储、负载平衡器），验证系统是否可以整体工作。这些测试是从最终用户的角度进行的，例如使用 Selenium 通过浏览器自动与产品交互。端到端测试通常运行在生产环境的镜像中（尽量使用更少/更小的服务器以节省成本）。虽然是镜像，但全部使用真实系统，不需要任何模拟。

每种类型的测试都有不同的用途，可以捕获不同类型的错误，因此需要将这 3 种类型的测试同时使用。单元测试的目的是快速运行测试，迅速获得关于更改的反馈，并验证各种不同参数的排列组合，增强对代码基本构建模块（各个单元）能够按预期工作的信心。但是，仅仅验证单个单元在隔离状态下可以正常工作，并不意味着它们组合在一起时仍然可以正常工作，所以需要进行集成测试，确保基本构建模块能够正确地组合在一起。仅仅因为系

统的不同部分可以一起正常工作，并不意味着它们在实际环境中部署后仍然可以正常工作，因此需要进行端到端测试，验证代码在近似生产环境下的行为是否符合预期。

现在让我们看一下，如何为 Terraform 代码编写每种类型的测试。

## 单元测试

要了解如何编写针对 Terraform 代码的单元测试，首先可以参考如何为 Ruby 之类的通用编程语言编写单元测试。再次看一看 Ruby Web 服务器代码。

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '[{"foo": "bar"}]'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

为这段代码编写单元测试有些棘手，因为需要执行以下操作。

1. 实例化 `WebServer` 类。这比听起来要困难得多，因为 `WebServer` 的构造函数扩展了 `AbstractServlet`，所以需要传递完整的 `WEBrick::HTTPServer` 类。你可以为它创建一个模拟，但这需要大量工作。
2. 创建一个 `request` 对象，其类型为 `HTTPRequest`。没有简单的方法来实例化这个类，并且创建它的模拟同样需要大量的工作。
3. 创建一个 `response` 对象，其类型为 `HTTPResponse`。同样，没有简单的方法来实例化这个类，并且创建它的模拟需要大量的工作。

当发现编写单元测试很困难时，通常是由于代码异味造成的，表明需要重构代码。通过重构 Ruby 代码来简化单元测试的一种方法是，将 handlers 功能（即处理 /、/api 和未找到的路径的功能代码）提取到单独的 Handlers 类中。

```
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

新的 Handlers 类有两个关键特点。

#### 简单的输入值

Handlers 类不依赖于 `HTTPServer`、`HTTPRequest` 或 `HTTPResponse`。相反，它的所有输入值都是基本参数，例如 URL 的 `path` 参数，它是字符串类型的。

#### 简单的输出值

Handlers 类无须在一个可变 `HTTPResponse` 对象（副作用）上设置值，它只将 HTTP 响应作为简单值（包含 HTTP 状态代码、内容类型和主体的数组）返回。

通常，以简单值作为输入、输出的代码更易于理解、更改和测试。让我们先来修改 `WebServer` 类，使用新的 `Handlers` 类来响应请求。

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    handlers = Handlers.new
    status_code, content_type, body = handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

这段代码调用 `Handlers` 类的 `handle` 方法，将状态码、内容类型和主体作为 HTTP 响应返回。如你所见，使用 `Handlers` 类很简单。同样的属性也将使测试变得容易。下面是针对/`endpoints` 的单元测试。

```
class TestWebServer < Test::Unit::TestCase
  def initialize(test_method_name)
    super(test_method_name)
    @handlers = Handlers.new
  end

  def test_unit_hello
    status_code, content_type, body = @handlers.handle("/")
    assert_equal(200, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Hello, World', body)
  end
end
```

测试代码调用了同一个 `Handlers` 类的 `handle` 方法，并使用了多个 `assert` 方法，验证从/`endpoints` 返回的响应。运行测试的方法如下。

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000287 seconds.

-----
1 tests, 3 assertions, 0 failures, 0 errors
100% passed
-----
```

看起来测试通过了。现在让我们添加对/`api` 端点和 404 响应的单元测试。

```
def test_unit_api
  status_code, content_type, body = @handlers.handle("/api")
  assert_equal(201, status_code)
  assert_equal('application/json', content_type)
  assert_equal('[{"foo":"bar"}]', body)
end

def test_unit_404
  status_code, content_type, body = @handlers.handle("/invalid-path")
  assert_equal(404, status_code)
  assert_equal('text/plain', content_type)
```

```
    assert_equal('Not Found', body)
end
```

再次运行测试。

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000572 seconds.

=====
3 tests, 9 assertions, 0 failures, 0 errors
100% passed
=====
```

在 0.0005272s 内，就可以确定 Web 服务器代码是否按预期工作了。这就是单元测试的威力：一个快速的正反馈循环，可以帮助用户建立对代码的信心。如果你在代码中犯了任何错误（例如，无意间更改了/api 端点的响应），则几乎会立即被发现。

```
$ ruby web-server-test.rb
Loaded suite web-server-test
=====
Failure: test_unit_api(TestWebServer)
web-server-test.rb:25:in `test_unit_api'
  22: status_code, content_type, body = Handlers.new.handle("/api")
  23: assert_equal(201, status_code)
  24: assert_equal('application/json', content_type)
=> 25: assert_equal('{"foo":"bar"}', body)
  26: end
  27:
  28: def test_unit_404
<"{"foo": "bar"}> expected but was
<"{"foo": "whoops"}>

diff:
? {"foo": "bar "}
? whoops
=====
Finished in 0.007904 seconds.

=====
3 tests, 9 assertions, 1 failures, 0 errors
66.6667% passed
=====
```

## 单元测试基础知识

在 Terraform 代码中与此类单元测试等价的对象是什么呢？首先是确定 Terraform 世界中的“单元”是什么？在 Terraform 中，与单个函数或类最接近的是单个通用模块（使用第 6 章的“可组合模块”中的术语“通用模块”），例如第 6 章中创建的 `alb` 模块。那么又该如何测试该模块呢？

为 Ruby 编写单元测试，需要重构代码，排除复杂依赖项（例如，`HTTPServer`、`HTTPRequest` 或 `HTTPResponse`）。如果考虑一下 Terraform 代码的工作方式：对 AWS 进行 API 调用以创建负载均衡器、侦听器、目标组等，你将意识到该代码中 99% 的工作是与复杂的依赖系统进行通信！没有实际可行的方法将外部依赖项的数量减少到 0，即使可以，那实际情况将变成没有代码再需要测试。<sup>1</sup>

这使我们提出测试要点 #3：无法对 Terraform 代码进行纯粹的单元测试。

但是不要绝望。你仍可以通过编写自动测试来建立信心，以确保 Terraform 代码的行为符合预期，该测试使用你的代码将真实的基础设施部署到真实的环境（例如，真实的 AWS 账户）中。换句话说，用于 Terraform 的单元测试实际上是集成测试。但是，我还是将它们称为单元测试，以强调测试的目标是单个单元（即单个通用模块），以尽快获得反馈。

这意味着编写 Terraform 单元测试的基本策略如下。

1. 创建一个通用的独立模块。
2. 为该模块创建一个易于部署的示例。
3. 运行 `terraform apply` 命令，将示例部署到实际环境中。

---

1 在有限的情况下，重写 Terraform 与提供商通信的端点是可能的。例如，你可以重写 Terraform 与 Amazon S3 进行沟通的端点，并用模拟端点实现 S3 API。对于少数的几个端点，这很好用，但是大多数 Terraform 代码对基础提供商进行了数百次不同的 API 调用，而将它们全部模拟出来是不切实际的。而且，即使完全模拟了它们，这样的单元测试能否使你对代码在正式环境下能够正确运行更有信心呢？例如，如果你为 ASG 和 ALB 创建模拟端点，则你的 `terraform apply` 命令可能会成功，但是这无法等同于代码可以在实际基础设施之上部署一个正常运行的应用程序。

4. 验证刚刚部署的内容是否按预期工作。此步骤特定于进行测试的基础设施类型。

例如对于 ALB，可以通过发送 HTTP 请求，并检查是否收到了预期的响应来进行验证。

5. 在测试结束时，运行 `terraform destroy` 命令进行清理。

换句话说，你执行了与手动测试完全相同的步骤，但是将这些步骤捕获为代码。事实上，这是一个为 Terraform 代码创建自动测试过程的很好的思考模型，问问自己：“我将如何手动验证它可以正常地工作？”然后将测试过程通过代码实现。

可以使用任何编程语言编写测试代码。本书中，所有测试都是使用 Go 语言编写的，这是为了利用名为 Terratest（见参考资料第 7 章[5]）的 Go 开源库的优势，该工具支持各种基础设施即代码（IaC）工具（例如 Terraform、Packer、Docker、Helm）的测试，以及各种各样的环境（例如 AWS、Google Cloud、Kubernetes）。Terratest 有点像瑞士军刀，内置了数百种工具，可以大大简化基础设施代码的测试工作，包括对上面描述的测试策略的一流支持，你通过 `terraform apply` 命令部署一些代码变更，验证其是否工作，然后运行 `terraform destroy` 命令进行清理。

要使用 Terratest，需要执行以下操作。

1. 安装 Go（见参考资料第 7 章[6]）。
2. 配置 `GOPATH` 环境变量（见参考资料第 7 章[7]）。
3. 添加`$GOPATH/bin` 到 `PATH` 环境变量。
4. 安装 Dep，这是 Go 的依赖项管理器（见参考资料第 7 章[8]）。<sup>1</sup>
5. 在 `GOPATH` 的目录下创建一个文件夹存放测试代码：例如，默认的 `GOPATH` 为 `$HOME/go`，因此可以创建`$HOME/go/src/terraform-up-and-running` 目录。
6. 在刚创建的文件夹中运行 `dep init` 命令。这将创建 `Gopkg.toml`、`Gopkg.lock` 文件和一个空的 `vendor` 文件夹。

---

<sup>1</sup> Terratest 的未来版本很可能会切换到使用 `go mod` 进行依赖管理。在撰写本文时，仅初步支持 `go mod`，但是到 Go 1.13 发布时，`go mod` 将在默认情况下启用，因此它很可能成为 Go 语言中依赖项管理的标准工具并消除对于 `GOPATH` 的要求。详细信息请参见参考资料第 7 章[9]。

为了快速检查环境已正确设置，请在新文件夹下创建具有以下内容的 `go_sanity_test.go` 文件。

```
package test

import (
    "fmt"
    "testing"
)

func TestGoIsWorking(t *testing.T) {
    fmt.Println()
    fmt.Println("If you see this text, it's working!")
    fmt.Println()
}
```

使用 `go test` 命令来执行测试，并确保看到下面的输出。

```
$ go test -v

If you see this text, it's working!

PASS
ok terraform-up-and-running 0.004s
```

(`-v` 标志表示输出详细信息，可以确保测试始终显示所有日志输出。)

如果执行成功了，请删除 `go_sanity_test.go` 文件，然后继续为 `alb` 模块编写单元测试。在 `test` 文件夹中创建一个拥有以下基本单元测试结构的文件 `alb_example_test.go`。

```
package test

import (
    "testing"
)

func TestAlbExample(t *testing.T) {
}
```

第一步是通过使用 `terraform.Options` 类型，指示 Terratest 在哪里能找到 Terraform 代码。

```
package test
```

```
import (
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
)

func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }
}
```

注意，测试 alb 模块，实际上是在 *examples* 文件夹中测试模块的示例代码（你应该更新 *TerraformDir* 中的相对路径，指向该示例的文件夹）。这意味着示例代码现在具有 3 个作用：可执行文档、为模块运行手动测试的方法，以及对模块运行自动测试的方法。

请注意，文件顶部现在新添加了一个 Terratest 库的导入。要下载依赖关系到计算机，运行 `dep ensure` 命令。

```
$ dep ensure
```

`dep ensure` 命令将会扫描你的 Go 代码，识别新的导入，自动下载它们及其所有的依赖关系到 *vendor* 文件夹，并将它们添加到 *Gopkg.lock* 文件。如果这有点太不可思议的话，可以替换使用 `dep ensure -add` 命令，明确地添加你想要的依赖关系。

```
$ dep ensure -add github.com/gruntwork-io/terratest/modules/terraform
```

自动测试的下一步，要运行 `terraform init` 和 `terraform apply` 命令来部署代码。Terratest 有方便的辅助工具来实现这些。

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    terraform.Init(t, opts)
    terraform.Apply(t, opts)
}
```

实际上，同时运行 `init` 和 `apply` 命令，是 Terratest 一个很常用的操作，所以还有一个更方便的方法将两个命令一起运行。

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    // Deploy the example
    terraform.InitAndApply(t, opts)
}
```

上面的代码已经是相当有用的单元测试了，因为它将运行 `terraform init` 和 `terraform apply` 命令，并在那些命令未能成功完成的情况下（例如，Terraform 代码出了问题），标注测试失败。如果更进一步，还可以通过向已部署的负载均衡器发出 HTTP 请求，检查其是否返回了期望的数据。为此，你需要获取已部署的负载均衡器的域名。幸运的是，域名可以通过 `alb` 示例的输出变量来获取。

```
output "alb_dns_name" {
    value      = module.alb.alb_dns_name
    description = "The domain name of the load balancer"
}
```

Terratest 具有内置助手，可以从 Terraform 代码读取输出值。

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    // 部署示例
    terraform.InitAndApply(t, opts)

    // 获取 ALB 的 URL 地址
    albDnsName :     = terraform.OutputRequired(t, opts, "alb_dns_name")
    url :          = fmt.Sprintf("http://%s", albDnsName)
}
```

`OutputRequired` 函数能够返回给定模块名称的输出值，如果输出不存在或为空就会标注

测试失败。上面的代码使用 Go 的内置 `fmt.Sprintf` 函数，从输出变量构建一个 URL（不要忘记导入 `fmt` 包）。下一步是对这个 URL 发出一些 HTTP 请求。

```
package test

import (
    "fmt"
    "github.com/gruntwork-io/terratest/modules/http-helper"
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
)

func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    // 部署示例
    terraform.InitAndApply(t, opts)

    // 获取 ALB 的 URL
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%", albDnsName)

    // 通过验证默认返回值是否为 404，判断 ALB 是否正常工作
    expectedStatus := 404
    expectedBody := "404: page not found"

    http_helper.HttpGetWithValidation(t, url, expectedStatus, expectedBody)
}
```

新版本代码导入了新的 Terratest 的程序库 `http_helper`，因此需要再次运行 `dep ensure` 命令进行下载。`http_helper.HttpGetWithValidation` 方法将向传入的 URL 发起一个 HTTP GET 请求，如果回应与预先定义不一致的状态代码和内容，则标记测试失败。

这段代码有一个问题：`terraform apply` 命令运行完成后，到负载均衡器的 DNS 名称正常工作（即已完成传播）之间有一段时间间隔。如果立即运行 `http_helper.HttpGetWithValidation` 可能会失败，即使在 30s 或 1min 后，ALB 是可以正常工作的。如第 5 章中的

“Terraform 陷阱”的“最终的一致性”所述，这种异步且最终一致的行为，在 AWS 中，甚至大多数分布式系统中都是很正常的，解决方案是添加重试。Terratest 有一个辅助方法。

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    // 部署示例
    terraform.InitAndApply(t, opts)

    // 获取 ALB 的 URL
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    // 通过验证默认返回值是否为 404，判断 ALB 是否正常工作
    expectedStatus := 404
    expectedBody := "404: page not found"

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        expectedStatus,
        expectedBody,
        maxRetries,
        timeBetweenRetries,
    )
}
```

`http_helper.HttpGetWithRetry` 方法与 `http_helper.HttpGetWithValidation` 方法几乎完全相同，区别是，如果无法获得预定义的状态代码或内容，它会每间隔一段时间（默认 10s）重新进行连接，直到达到指定的最大重试次数（默认 10 次）。如果最终接收到预定义的响应，则标注测试通过。如果达到最大重试次数后还没有收到预定义的响应，则标注测试失败。

最后一件事是在测试结束时运行 `terraform destroy` 进行清理。或许你已经猜到，有

一个 Terratest 辅助功能：`terraform.Destroy`。但是，如果在测试最终结束时才调用 `terraform.Destroy`，那么之前的任何代码测试失败（例如，因为 ALB 配置错误，`HttpGetWithRetry` 失败），都会导致测试程序在 `terraform.Destroy` 运行之前就已经退出，结果是，为测试部署的基础设施永远不会被正确地销毁。

因此，需要确保即使测试失败，也能够运行 `terraform.Destroy`。在许多编程语言中，会通过 `try/finally` 或 `try/ensure` 结构语句来实现这一点。在 Go 语言中，是通过使用 `defer` 语句实现的，`defer` 声明将在外部函数返回时（无论返回值是什么），确保执行传递给它的代码。

```
func TestAlbExample(t *testing.T) {
    opts := &terraform.Options{
        // 更新以下相对路径，指向你的 alb 示例代码目录
        TerraformDir: "../examples/alb",
    }

    // 在测试结束时清理所有内容
    defer terraform.Destroy(t, opts)

    // 部署示例
    terraform.InitAndApply(t, opts)

    // 获取 ALB 的 URL
    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%", albDnsName)

    // 通过验证默认返回值是否为 404，判断 ALB 是否正常工作
    expectedStatus := 404
    expectedBody := "404: page not found"

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        expectedStatus,
        expectedBody,
```

```
    maxRetries,
    timeBetweenRetries,
)
}
```

注意，`defer` 声明添加在代码靠前的部分，甚至在 `terraform.InitAndApply` 之前，这样可以保证没有任何语句会在 `defer` 声明之前导致测试失败而退出，而且可以避免 `terraform.Destroy` 在调用队列中的等待。

单元测试终于可以运行了。因为此测试是基于 AWS 的基础设施部署的，所以在运行测试之前，需要像往常一样对用户的 AWS 账户进行身份验证（请参阅第 2 章中的“身份验证选项”）。我们在本章的前面已经学习到，手动测试应该在沙箱账户中进行。对于自动测试，这一点更为重要，因此我建议使用完全独立的账户进行身份验证。随着自动测试代码的增加，每个测试套件中都可能包括数百或数千个资源，因此，将它们与其他所有环境隔离开是至关重要的。

我通常建议团队使用完全独立的环境（例如，完全独立的 AWS 账户）进行自动测试，甚至应该与手动测试的沙箱环境区分开。这样就可以基于没有测试会长时间运行的假设，安全地删除测试环境中创建时间超过几个小时的资源了。

通过对正确的 AWS 测试账户进行身份验证后，可以运行测试命令。

```
$ go test -v -timeout 30m

TestAlbExample 2019-05-26T13:29:32+01:00 command.go:53:
Running command terraform with args [init -upgrade=false]
(...)

TestAlbExample 2019-05-26T13:29:33+01:00 command.go:53:
Running command terraform with args [apply -input=false -lock=false]
(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:121:
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:53:
Running command terraform with args [output -no-color alb_dns_name]
```

```
(...)

TestAlbExample 2019-05-26T13:38:32+01:00 http_helper.go:27:
Making an HTTP GET call to URL
http://terraform-up-and-running-1892693519.us-east-2.elb.amazonaws.com
(...)

TestAlbExample 2019-05-26T13:38:32+01:00 command.go:53:
Running command terraform with args
[destroy -auto-approve -input=false -lock=false]
(...)

TestAlbExample 2019-05-26T13:39:16+01:00 command.go:121:
Destroy complete! Resources: 5 destroyed.
(...)

PASS
ok  terraform-up-and-running 229.492s
```

请注意`-timeout 30m`参数与`go test`一起使用的方式。默认情况下，Go语言对测试执行的默认超时为10min，在此之后它会强制终止测试的运行，因为超时运行的测试不仅会导致测试失败，还会阻止清除代码（即`terraform destroy`）的执行。这项ALB的测试大约需要5min，但是每当运行针对真实基础设施环境的Go测试时，设置一个较长的超时时间会比较安全，这可以避免测试意外终止后，各种为测试创建的基础设施仍然继续运行。

测试会产生大量日志输出，如果你仔细阅读它，就能够发现该测试的关键部分。

1. 运行`terraform init`命令。
2. 运行`terraform apply`命令。
3. 使用`terraform output`命令，读取输出变量。
4. 反复向ALB发出HTTP请求。
5. 运行`terraform destroy`命令。

执行速度远没有Ruby单元测试快，但是不到5min就可以自动检验出alb模块是否按预期工作。这是使用AWS基础设施所能获得的最快反馈了，它可以使用户对代码充满信心。

如果代码中存在任何错误（例如，无意中将默认操作返回的状态代码改成了 401），将很快会被发现。

```
$ go test -v -timeout 30m
(...)

Validation failed for URL
http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.
(...)

Sleeping for 10s and will try again.
(...)

Validation failed for URL
http://terraform-up-and-running-h2ezYz-931760451.us-east-2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.
(...)

Sleeping for 10s and will try again.
(...)

--- FAIL: TestAlbExample (310.19s)
    http_helper.go:94:
        HTTP GET to URL
        http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com
        unsuccessful after 10 retries
FAIL    terraform-up-and-running    310.204s
```

## 依赖注入

现在让我们学习如何为稍微复杂的代码添加单元测试。再次回到 Ruby Web 服务器示例，请考虑如果需要新添加一个`/web-service` 端点，端点接受请求后，会对一个外部依赖项进行 HTTP 调用。

```
class Handlers
  def handle(path)
    case path
    when "/"
      plain, 'Hello, World']
    when "/api"
```

```

[201, 'application/json', '{"foo":"bar"}']
when "/web-service"
    # 新的调用端点，会发起另一个对外部网络服务的调用
    uri = URI("http://www.example.org")
    response = Net::HTTP.get_response(uri)
    [response.code.to_i, response['Content-Type'], response.body]
else
    [404, 'text/plain', 'Not Found']
end
end
end

```

更新后的 `Handlers` 类现在可以处理 `/web-service` URL 的请求了：将 HTTP GET 发送到 `example.org`，并处理响应。当你尝试用 `curl` 命令连接这个端点时，将得到以下内容。

```

$ curl localhost:8000/web-service

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <!-- (...) -->
</head>
<body>
<div>
    <h1>Example Domain</h1>
    <p>
        This domain is established to be used for illustrative
        examples in documents. You may use this domain in
        examples without prior coordination or asking for permission.
    </p>
    <!-- (...) -->
</div>
</body>
</html>

```

应该如何为这个新方法添加单元测试呢？如果尝试按同样的逻辑编写测试代码，单元测试将受到外部依赖项（在本例中为 `example.org`）的行为的约束。这有很多缺点。

- 如果该依赖项发生故障，则即使代码没有问题，测试也会失败。
- 如果该依赖项发生了行为变化（例如，返回了一个不同的响应主体），则测试将会

失败，即使代码没有问题，你也需要不断更新测试代码。

- 如果该外部依赖项运行速度很慢，那么你的测试也会很慢，这违反了单元测试的主要好处之一，即快速反馈循环。
- 如果你打算根据该外部依赖项的行为测试代码是否能够处理各种极端情况（例如，代码是否能够处理重定向），那么在缺少对该外部依赖项的实际控制的情况下将无能为力。

使用真实的依赖关系，对于集成测试和端到端测试是有意义的。但是对于单元测试，应该尝试尽可能地减少依赖外部关系项。解决这个问题的典型策略是依赖注入。这种注入可以从代码外部传递（或注入）外部依赖，而不必在代码内对其进行静态编码。

例如，`Handlers` 类不需要处理如何调用 Web 服务的所有细节。相反，可以提取逻辑到一个单独的 `WebService` 类。

```
class WebService
  def initialize(url)
    @uri = URI(url)
  end

  def proxy
    response = Net::HTTP.get_response(@uri)
    [response.code.to_i, response['Content-Type'], response.body]
  end
end
```

这个类将 URL 作为输入，并公开一个用于代理 HTTP GET 响应的 `proxy` 方法。然后，更新 `Handlers` 类将 `WebService` 实例作为输入，并在 `web_service` 方法中使用该实例。

```
class Handlers
  def initialize(web_service)
    @web_service = web_service
  end

  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World!']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
    end
  end
end
```

```
        # New endpoint that calls a web service
        @web_service.proxy
    else
        [404, 'text/plain', 'Not Found']
    end
end
end
```

现在，在代码实现中，你可以注入一个真正的 `WebService` 实例，通过 HTTP 调用 `example.org`。

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    web_service = WebService.new("http://www.example.org")
    handlers = Handlers.new(web_service)

    status_code, content_type, body = handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

在测试代码中，你可以创建模拟版本的 `WebService` 类，模拟返回响应。

```
class MockWebService
  def initialize(response)
    @response = response
  end

  def proxy
    @response
  end
end
```

现在，你可以创建一个 `MockWebService` 类的实例，并在单元测试中，将其注入 `Handlers` 类。

```
def test_unit_web_service
  expected_status = 200
  expected_content_type = 'text/html'
  expected_body = 'mock example.org'
  mock_response = [expected_status, expected_content_type, expected_body]
```

```
mock_web_service = MockWebService.new(mock_response)
handlers = Handlers.new(mock_web_service)

status_code, content_type, body = handlers.handle("/web-service")
assert_equal(expected_status, status_code)
assert_equal(expected_content_type, content_type)
assert_equal(expected_body, body)
end
```

重新运行测试，以确保这一切仍然有效。

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Started
...
Finished in 0.000645 seconds.
-----
4 tests, 12 assertions, 0 failures, 0 errors
100% passed
-----
```

太棒了！使用依赖注入可以最大程度地减少外部依赖，使用户可以编写快速、可靠的测试，并验证各种极端情况。由于之前添加的 3 个测试用例仍然可以通过，因此你可以确信重构没有破坏任何内容。

现在让我们将注意力转向 Terraform，从头开始看看 Terraform 模块的依赖注入是什么样子的。从 `hello-world-app` 模块开始，首先是在 `examples` 文件夹中为它创建一个易于部署的示例。

```
provider "aws" {
  region = "us-east-2"
  # 允许任意 2.x 版本的 AWS 提供商代码
  version = "~> 2.0"
}

module "hello_world_app" {
  source          = "../../modules/services/hello-world-app"
  server_text     = "Hello, World"
```

```

environment = "example"

db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key     = "examples/terraform.tfstate"

instance_type           = "t2.micro"
min_size                = 2
max_size                = 2
enable_autoscaling      = false
}

```

依赖关系的问题立即变得显而易见：`hello-world-app` 假定你已经部署了 `mysql` 模块，并需要 S3 bucket 的详细信息，因为 `mysql` 模块正在通过 `db_remote_state_bucket` 和 `db_remote_state_key` 参数存储状态文件到 S3 bucket。我们的目标是为 `hello-world-app` 模块创建一个单元测试，尽管 Terraform 不可能实现完全不依赖外部的纯单元测试，但我们可以尽可能地最小化外部依赖项的数目。

建立最小化依赖关系的第一步是了解模块现有的依赖关系。一个推荐的文件命名规范是，将与外部依赖项相关的数据源和资源转移到单独的 `dependency.tf` 文件中。例如，这是 `modules/services/hello-world-app/dependencies.tf` 文件的样子。

```

data "terraform_remote_state" "db" {
  backend    = "s3"

  config    = {
    bucket    = var.db_remote_state_bucket
    key       = var.db_remote_state_key
    region   = "us-east-2"
  }
}

data "aws_vpc" "default" {
  default    = true
}

data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}

```

这个命名规范可以使代码的用户立即就能了解，代码对外部世界有哪些依赖内容。对于

`hello-world-app` 模块，我们可以快速了解到，它依赖于数据库、VPC 和子网。那么，如何从模块外部注入这些依赖关系项，以便可以在测试时替换它们呢？你可能已经知道了答案：输入变量。

对于每一个依赖，应该在模块中添加一个新的输入变量，如 `modules/services/hello-world-app/variables.tf` 文件所示。

```
variable "vpc_id" {
  description = "The ID of the VPC to deploy into"
  type = string
  default = null
}

variable "subnet_ids" {
  description = "The IDs of the subnets to deploy into"
  type = list(string)
  default = null
}

variable "mysql_config" {
  description = "The config for the MySQL DB"
  type = object({
    address = string
    port = number
  })
  default = null
}
```

现在有输入变量：VPC ID、子网 ID 和 MySQL 配置。每个变量都指定一个 `default` 值，因此它们是可选变量，用户可以定义变量，或者省略，以便使用默认值。每个变量的默认值是一个之前从未见过的值：`null`。如果将 `default` 值设置为空值，例如空字符串的 `vpc_id` 或空列表的 `subnet_ids`，那么将无法区分设置默认值为空值与模块的用户有意传入空值这两种情况。在这种情况下，`null` 值很方便，因为它专门用于标示未设置变量，意味着用户希望使用默认值。

请注意，`mysql_config` 变量使用 `object` 类型构造函数创建了拥有 `address` 和 `port` 为键值的嵌套类型。这种类型专门为 `mysql` 模块输出类型而设计。

```
output "address" [
  value      = aws_db_instance.example.address
```

```
    description = "Connect to the database at this endpoint"
}

output "port" {
    value      = aws_db_instance.example.port
    description = "The port the database is listening on"
}
```

这样做的优点是，一旦完成重构，便可以将 `hello-world-app` 模块和 `mysql` 模块一起使用，如下所示。

```
module "hello_world_app" {
    source = "../../modules/services/hello-world-app"

    server_text      = "Hello, World"
    environment      = "example"

    # 直接导入 mysql 模块的全部输出
    mysql_config     = module.mysql

    instance_type    = "t2.micro"
    min_size         = 2
    max_size         = 2
    enable_autoscaling = false
}

module "mysql" {
    source = "../../modules/data-stores/mysql"

    db_name          = var.db_name
    db_username       = var.db_username
    db_password       = var.db_password
}
```

由于 `mysql_config` 变量的类型与 `mysql` 模块输出的类型相匹配，所以可以直接在同一行赋值。而且，如果今后类型发生了变化不再匹配的情况，Terraform 将立即报错，提示应立即更新。这不仅是一个函数组合的例子，还是类型安全的函数组合的例子。

在代码工作之前，你需要完成代码重构。由于已经将 MySQL 配置作为参数整体传递，所以 `db_remote_state_bucket` 和 `db_remote_state_key` 变量现在是可选的，因此可以将它

们的 default 设置为 null。

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the DB's Terraform state"
  type       = string
  default    = null
}

variable "db_remote_state_key" {
  description = "The path in the S3 bucket for the DB's Terraform state"
  type       = string
  default    = null
}
```

接下来，使用 count 参数，根据相应的输入变量是否为 null，有条件地创建 3 个数据源。如 *modules/services/hello-world-app/dependencies.tf* 文件所示。

```
data "terraform_remote_state" "db" {
  count      = var.mysql_config == null ? 1 : 0

  backend    = "s3"

  config     = {
    bucket    = var.db_remote_state_bucket
    key       = var.db_remote_state_key
    region    = "us-east-2"
  }
}

data "aws_vpc" "default" {
  count = var.vpc_id == null ? 1 : 0
  default = true
}

data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = data.aws_vpc.default.id
}
```

数据源的引用需要通过条件判断，来决定是使用输入变量还是数据源。让我们使用本地变量来实现。

```
locals {
```

```

mysql_config = (
  var.mysql_config == null
  ? data.terraform_remote_state.db[0].outputs
  : var.mysql_config
)

vpc_id = (
  var.vpc_id == null
  ? data.aws_vpc.default[0].id
  : var.vpc_id
)

subnet_ids = (
  var.subnet_ids == null
  ? data.aws_subnet_ids.default[0].ids
  : var.subnet_ids
)
)

```

请注意，由于数据源使用 `count` 参数，它们现在是数组类型。因此，每次引用它们时，都需要使用数组查找语法（即`[0]`）。检查代码，将每一个引用这些数据源的地方，替换为相对应的本地变量。首先将 `aws_subnet_ids` 数据源更新为本地变量 `local.vpc_id`。

```

data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = local.vpc_id
}

```

然后，将 `asg` 和 `alb` 模块的 `subnet_ids` 参数设置为本地变量 `local.subnet_ids`。

```

module "asg" {
  source          = "../../cluster/asg-rolling-deploy"

  cluster_name    = "hello-world-${var.environment}"
  ami             = var.ami
  user_data       = data.template_file.user_data.rendered
  instance_type   = var.instance_type

  min_size        = var.min_size
  max_size        = var.max_size
  enable_autoscaling = var.enable_autoscaling

```

```

subnet_ids      = local.subnet_ids
target_group_arns = [aws_lb_target_group.asg.arn]
health_check_type = "ELB"

custom_tags      = var.custom_tags
}

module "alb" {
  source          = "../../networking/alb"

  alb_name        = "hello-world-${var.environment}"
  subnet_ids      = local.subnet_ids
}

```

更新 user\_data 资源中的 db\_address 和 db\_port 变量为本地变量 local.mysql\_config。

```

data "template_file" "user_data" {
  template        = file("${path.module}/user-data.sh")

  vars = {
    server_port     = var.server_port
    db_address       = local.mysql_config.address
    db_port          = local.mysql_config.port
    server_text      = var.server_text
  }
}

```

最后，更新 aws\_lb\_target\_group 资源中的 vpc\_id 参数为本地变量 local.vpc\_id。

```

resource "aws_lb_target_group" "asg" {
  name           = "hello-world-${var.environment}"
  port           = var.server_port
  protocol       = "HTTP"
  vpc_id         = local.vpc_id

  health_check {
    path          = "/"
    protocol     = "HTTP"
    matcher       = "200"
    interval     = 15
    timeout      = 3
    healthy_threshold = 2
    unhealthy_threshold = 2
  }
}

```

通过这些更新，现在可以将 VPC ID、子网 ID 和（或）MySQL 配置注入 `hello-world-app` 模块中，或省略其中任何一个参数，模块将使用适当的数据源自行获取这些值。让我们更新“Hello,World”应用程序示例，允许注入 MySQL 配置，但省略 VPC ID 和子网 ID 参数，因为使用默认 VPC 足够进行测试。添加一个新的输入变量到 `examples/hello-world-app/variables.tf` 文件中。

```
variable "mysql_config" {
  description      = "The config for the MySQL DB"

  type = object({
    address        = string
    port           = number
  })

  default         = {
    address        = "mock-mysql-address"
    port           = 12345
  }
}
```

将此变量传递到 `examples/hello-app/main.tf` 文件中的 `world-hello-world-app` 模块。

```
module "hello_world_app" {
  source          = "../../modules/services/hello-world-app"

  server_text     = "Hello, World"
  environment     = "example"

  mysql_config    = var.mysql_config

  instance_type   = "t2.micro"
  min_size        = 2
  max_size        = 2
  enable_autoscaling = false
}
```

现在，在单元测试中可以将 `mysql_config` 变量设置为任何值。在 `test/hello_world_app_example_test.go` 文件中添加一个单元测试。

```
func TestHelloWorldAppExample(t *testing.T) {
  opts := &terraform.Options{
    // 请更新相对路径到 hello-world-app 的示例目录
```

```

        TerraformDir: "../examples/hello-world-app/standalone",
    }

    // 在测试结束后，删除所有内容
    defer terraform.Destroy(t, opts)
    terraform.InitAndApply(t, opts)

    albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    expectedStatus := 200
    expectedBody := "Hello, World"

    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetry(
        t,
        url,
        expectedStatus,
        expectedBody,
        maxRetries,
        timeBetweenRetries,
    )
}

```

上面的代码与 `alb` 示例的单元测试几乎相同。唯一的区别是 `TerraformDir` 参数指向 `hello-world-app` 示例（请确保根据文件系统的需要更新路径），ALB 的预期响应为 200 OK，响应内容为“Hello,World”。唯一需要新增的设置是，向该测试添加 `mysql_config` 变量。

```

opts := &terraform.Options{
    // 请将相对路径指向 hello-world-app 示例路径
    TerraformDir: "../examples/hello-world-app/standalone",

    Vars: map[string]interface{}{
        "mysql_config": map[string]interface{}{
            "address": "mock-value-for-test",
            "port": 3306,
        },
    },
}

```

`terraform.Options` 中的 `Vars` 参数允许在 Terraform 代码中设置变量。该段代码为 `mysql_config` 变量设置了一些模拟数据。用户可以在此设置任何需要的值。例如，可以在测试时启动一个小型驻留内存的数据库，将 `address` 设置为该数据库的 IP。

使用 `go test` 命令，运行这个新的测试用例。通过 `-run` 参数指示仅运行此测试（否则，Go 语言的默认行为是运行当前文件夹中的所有测试，包括你之前创建的关于 ALB 示例的测试）。

```
$ go test -v -timeout 30m -run TestHelloWorldAppExample  
(...)  
PASS  
ok terraform-up-and-running 204.113s
```

如果一切顺利，测试将运行 `terraform apply` 命令，向负载均衡器重复发送 HTTP 请求，等得到了预期的响应后，通过 `terraform destroy` 命令来清理所有内容。总共只需要花费几分钟，现在你已经对“Hello,World”应用程序进行了合理的单元测试。

### 并行运行测试

在上一节中，`go test` 命令使用 `-run` 参数，仅运行了一个测试。如果省略该参数，Go 语言将按顺序运行所有测试。尽管需要花费 4 到 5min 运行单个测试，但对于测试基础设施代码来说还算不错。但是如果有数十个测试，并且每个测试都按顺序运行，那么就可能需要数小时才能完成整个测试套件的运行。为了缩短测试结果的反馈时间，可以通过并行的方式运行测试。

指示 Go 语言并行运行测试，需要做的唯一更改是在每个测试的开头添加 `t.Parallel()`。如 `test/hello_world_app_example_test.go` 文件所示。

```
func TestHelloWorldAppExample(t *testing.T) {  
    t.Parallel()  
  
    opts := &terraform.Options{  
        //请将相对路径指向 hello-world-app 示例路径  
        TerraformDir: "../examples/hello-world-app/standalone",
```

```
    Vars: map[string]interface{}{
        "mysql_config": map[string]interface{}{
            "address": "mock-value-for-test",
            "port": 3306,
        },
    },
}
// ...
}
```

用类似方式修改 `test/alb_example_test.go` 文件。

```
func TestAlbExample(t *testing.T) {
    t.Parallel()

    opts := &terraform.Options{
        // 你应该更新此相对路径以指向你的 alb 示例目录
        TerraformDir: "../examples/alb",
    }
    // ...
}
```

如果现在运行 `go test` 命令，两个测试将并行执行。但是，这里有一个陷阱：这些测试创建的某些资源（例如 ASG、安全组和 ALB）使用相同的名称，由于名称冲突会导致测试失败。测试中即使没有使用 `t.Parallel()`，但由于团队中的多个人正在运行相同的测试，或者你在 CI 环境中运行了多个测试，这些名称冲突也是不可避免的。

这引出了测试要点 # 4：确保资源拥有独立的命名空间。

也就是说，在设计模块和示例时，要确保每个资源的名称都是可配置的。在 `alb` 示例中，需要将 ALB 名称命名改为可配置的。在 `examples/alb/variables.tf` 文件中添加一个具有合理的默认值的新输入变量。

```
variable "alb_name" {
    description      = "The name of the ALB and all its resources"
    type             = string
    default          = "terraform-up-and-running"
}
```

接下来，将这个值传递给 `examples/alb/main.tf` 文件中的 `alb` 模块。

```
module "alb" {
  source          = "../../modules/networking/alb"

  alb_name        = var.alb_name
  subnet_ids      = data.aws_subnet_ids.default.ids
}
```

现在，在 `test/alb_example_test.go` 文件中给这个变量设置一个唯一值。

```
package test

import (
    "fmt"
    "github.com/gruntwork-io/terratest/modules/http-helper"
    "github.com/gruntwork-io/terratest/modules/random"
    "github.com/gruntwork-io/terratest/modules/terraform"
    "testing"
    "time"
)

func TestAlbExample(t *testing.T) {
    t.Parallel()

    opts := &terraform.Options{
        // 你应该更新此相对路径以指向你的 alb 示例目录
        TerraformDir: "../examples/alb",

        Vars: map[string]interface{}{
            "alb_name": fmt.Sprintf("test-%s", random.UniqueId()),
        },
    }
    // ...
}
```

(注意，因为使用了新的 Terratest 辅助方法 `random`，你需要再次运行 `dep ensure` 命令。)

此代码将 `alb_name` 变量设置为 `test-<RANDOM_ID>`，其中 `RANDOM_ID` 是通过 Terratest 的 `random.UniqueId()` 辅助方法返回一个随机数。该助手返回一个随机的、包含 6 个字符的 base-62 字符串。目的是将一个短小的标识符添加到大多数资源名称中，既不会有长度限制问题，又要足够随机，将发生冲突的概率降到最低 ( $62^6 =$  超过 560 亿的不同组合)。这样可以确保在并行运行大量的 ALB 测试时，不会发生名称冲突。

对“Hello,World”应用程序示例进行类似的改造，首先在 `examples/hello-world-app/variables.tf` 文件中添加一个新的输入变量。

```
variable "environment" {
  description      = "The name of the environment we're deploying to"
  type             = string
  default          = "example"
}
```

然后将该变量传递给 `hello-world-app` 模块。

```
module "hello_world_app" {
  source          = "../../modules/services/hello-world-app"

  server_text     = "Hello, World"

  environment     = var.environment

  mysql_config    = var.mysql_config

  instance_type   = "t2.micro"
  min_size        = 2
  max_size        = 2
  enable_autoscaling = false
}
```

最后，在 `test/hello_world_app_example_test.go` 文件中，将 `environment` 变量设置为一个包括 `random.UniqueId()` 的随机值。

```
func TestHelloWorldAppExample(t *testing.T) {
  t.Parallel()

  opts := &terraform.Options{
    // 请将以下相对路径指向 hello-world-app 示例目录
    TerraformDir: "../examples/hello-world-app/standalone",

    Vars: map[string]interface{}{
      "mysql_config": map[string]interface{}{
        "address": "mock-value-for-test",
        "port": 3306,
      },
      "environment": fmt.Sprintf("test-%s", random.UniqueId()),
    }
}
```

```
        },  
    }  
    // (...)  
}
```

当完成这些更改后，就可以并行运行所有测试了。

```
$ go test -v -timeout 30m  
  
TestAlbExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)  
TestAlbExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)  
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)  
  
(...)  
  
PASS  
ok terraform-up-and-running 216.090s
```

能够看到两个测试用例同时运行。整个测试套件所需的时间是最慢的测试用例的执行时间，而不是所有测试用例的时间总和。



### 在同一文件夹中并行运行测试

另外一种需要考虑的并行机制是，对同一 Terraform 文件夹并行运行多个自动测试将会发生什么情况。例如，对 `examples/hello-world-app` 文件夹同时运行几个不同的测试，每个测试在运行 `terraform apply` 命令时，设置了不同的输入变量。这种方法因为它们同时运行 `terraform init` 命令，会覆盖彼此的 `.terraform` 文件夹和 Terraform 状态文件，测试最终会发生冲突。

对同一个文件夹并行运行多个测试，最简单的解决方案是将每个测试复制到一个独立的临时文件夹，在该临时文件夹中运行 Terraform 以避免冲突。Terratest 有一个内置的辅助方法模块可以做到这一点，它甚至可以确保即使那些 Terraform 模块中使用了相对文件路径，在复制后仍旧可以正常工作。请查看 `test_structure.CopyTerraformFolderToTemp` 方法及其文档来了解详细信息。

## 集成测试

现在你已经完成了一些单元测试，下面继续学习集成测试。让我们首先通过 Ruby Web 服务器示例建立一些概念，之后可以将其应用于 Terraform 代码。要对 Ruby Web 服务器代码进行集成测试，需要执行以下操作。

1. 在 localhost 上运行 Web 服务器，以便侦听端口。
2. 将 HTTP 请求发送到 Web 服务器。
3. 确认获得了期望的答复。

让我们在 `web-server-test.rb` 文件中创建一个辅助方法，来实现上述步骤操作。

```
def do_integration_test(path, check_response)
  port = 8000
  server = WEBrick::HTTPServer.new :Port => port
  server.mount '/', WebServer

  begin
    # 以独立线程启动 Web 服务器，避免阻碍测试的运行
    thread = Thread.new do
      server.start
    end

    # 对 Web 服务器的特定路径发出请求
    url = URI("http://localhost:#{{port}}#[path]")
    response = Net::HTTP.get_response(url)

    # 通过 lambda 函数 check_response 验证响应
    check_response.call(response)
  ensure
    # 测试结束后关闭服务器线程
    server.shutdown
    thread.join
  end
end
```

`do_integration_test` 方法将 Web 服务器配置在端口 8000，并以后台线程方式启动（这样 Web 服务器就不会阻碍测试的运行了）。发送 HTTP GET 请求到上面特定的 `path`，将返回的 HTTP 响应传递给 `check_response` 函数进行验证，在测试结束时关闭 Web 服务器。

下面是调用这个方法为/端点编写服务器的集成测试的示例。

```
def test_integration_hello
  do_integration_test('/', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Hello, World', response.body)
  })
end
```

将路径/和lambda函数(本质上是一个内联函数)传递给do\_integration\_test函数。lambda函数用于检查响应是否为200 OK, 响应主体是否为“Hello,World”。其他端点的集成测试也是类似的, 尽管/web-service端点的检查方法略有不同(通过assert\_include而不是assert\_equal), 因为这样可以尽量减少由于example.org端点更改对测试结果造成的影响。

```
def test_integration_api
  do_integration_test('/api', lambda { |response|
    assert_equal(201, response.code.to_i)
    assert_equal('application/json', response['Content-Type'])
    assert_equal('{"foo": "bar"}', response.body)
  })
end

def test_integration_404
  do_integration_test('/invalid-path', lambda { |response|
    assert_equal(404, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Not Found', response.body)
  })
end

def test_integration_web_service
  do_integration_test('/web-service', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_include(response['Content-Type'], 'text/html')
    assert_include(response.body, 'Example Domain')
  })
end
```

让我们运行所有的测试。

```
$ ruby web-server-test.rb
```

```
(...)
Finished in 0.221561 seconds.
-----
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
-----
```

请注意，以前单独执行单元测试，测试套件需要 0.000572s 完成。现在执行集成测试时，却需要 0.221561s 才能完成，用时大约是原来的 387 倍。0.221561s 仍是一个很短的时间，但这是因为我们使用的 Ruby Web 服务器代码是一个没有太多功能的最小化示例。这里要强调的不是绝对时间，而是相对比较趋势：集成测试通常比单元测试慢。稍后我会继续讨论这一点。

让我们将注意力继续保持在 Terraform 代码的集成测试上。如果 Terraform 中的“单元”是单个模块，则验证多个单元协同工作的集成测试首先需要部署多个模块，并确保它们正常工作。在上一节中，我们使用模拟数据而不是真实的 MySQL DB，部署了“Hello,World”应用示例。为了进行集成测试，让我们部署一个真实的 MySQL 模块，并确保“Hello,World”应用与其正确集成。在 *live/stage/data-stores/mysql* 和 *live/stage/services/hello-world-app* 文件夹下可以看到相关代码，这些代码展示了如何对预发布环境（的一部分）进行集成测试。

如本章前面所述，所有自动测试都应在独立的 AWS 账户中运行。因此，测试预发布环境代码时，应该使用一个独立的测试账户，进行身份验证并在其中运行测试。如果模块中存在任何特定于预发布环境的静态代码，请将这些值改为可配置变量，方便之后插入测试参数。特别需要注意，在 *live/stage/data-stores/mysql/variables.tf* 文件中，通过新的输入变量 `db_name`，使数据库名称变得可配置。

```
variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}
```

在代码文件 *live/stage/data-stores/mysql/main.tf* 中，将这个输入变量直接传递给 `mysql` 模块。

```
module "mysql" {
  source      = "../../../../../modules/data-stores/mysql"

  db_name    = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

现在让我们首先在 `test/hello_world_integration_test.go` 文件中创建集成测试框架，稍后填入实现细节。

```
// 请替换为指向你的模块的正确的路径
const dbDirStage  = "../live/stage/data-stores/mysql"
const appDirStage = "../live/stage/services/hello-world-app"

func TestHelloWorldAppStage(t *testing.T) {
  t.Parallel()

  // 部署 MySQL 数据库
  dbOpts := createDbOpts(t, dbDirStage)
  defer terraform.Destroy(t, dbOpts)
  terraform.InitAndApply(t, dbOpts)

  // 部署 hello-world-app
  helloOpts := createHelloOpts(dbOpts, appDirStage)
  defer terraform.Destroy(t, helloOpts)
  terraform.InitAndApply(t, helloOpts)

  // 验证 hello-world-app 的工作
  validateHelloApp(t, helloOpts)
}
```

测试的步骤如下：部署 `mysql`、部署 `hello-world-app`、验证应用程序、销毁 `hello-world-app`（通过 `defer` 在测试最后阶段执行）、销毁 `mysql`（通过 `defer` 在测试最后阶段执行）。其中的 `createDbOpts`、`createHelloOpts` 和 `validateHelloApp` 方法还未被创建，现在让我们逐一实现，从 `createDbOpts` 方法开始。

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
  uniqueId := random.UniqueId()

  return &terraform.Options{
```

```

    TerraformDir: terraformDir,
    Vars: map[string]interface{}{
        "db_name": fmt.Sprintf("test%s", uniqueId),
        "db_password": "password",
    },
}
]

```

到目前为止还没有什么新内容：只是将 `terraform.Options` 设置为传入目录，并设置 `db_name` 和 `db_password` 变量。

下一步是处理该 `mysql` 模块的状态文件存储位置。到目前为止，`backend` 配置始终还是静态值。

```

terraform {
    backend "s3" {
        # 请替换为你自己拥有的 S3 bucket 名称
        bucket      = "terraform-up-and-running-state"
        key         = "stage/data-stores/mysql/terraform.tfstate"
        region      = "us-east-2"

        # 请替换为你自己拥有的 DynamoDB 表的名称
        dynamodb_table = "terraform-up-and-running-locks"
        encrypt       = true
    }
}

```

这些静态值在测试中会导致大问题，如果不改变它们，将会覆盖预发布环境中的实际状态文件。一种选择是使用 Terraform 工作空间（如第 3 章的“通过工作区进行隔离”中所述），但这么做仍然需要访问预发布环境账户中的 S3 bucket，不符合使用完全独立的 AWS 账户进行测试的原则。更好的选择是使用部分配置，如第 3 章的“Terraform 后端的局限性”中所述。将整个 `backend` 配置移到一个外部文件，例如 `backend.hcl`。

```

bucket      = "terraform-up-and-running-state"
key         = "stage/data-stores/mysql/terraform.tfstate"
region      = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt       = true

```

保持 `live/stage/data-stores/mysql/main.tf` 文件中的 `backend` 设置为空。

```
terraform {
  backend "s3" {
  }
}
```

当部署 `mysql` 模块到真正的预发布环境时,可以通过 `-backend-config` 参数指示 Terraform, 使用 `backend.hcl` 中的配置参数。

```
$ terraform init -backend-config=backend.hcl
```

当测试 `mysql` 模块时, `Terratest` 通过使用 `terraform.Options` 的 `BackendConfig` 参数, 传入测试数据库参数。

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
    uniqueId := random.UniqueId()

    bucketForTesting := "YOUR_S3_BUCKET_FOR_TESTING"
    bucketRegionForTesting := "YOUR_S3_BUCKET_REGION_FOR_TESTING"
    dbStateKey := fmt.Sprintf("%s/%s/terraform.tfstate", t.Name(), uniqueId)

    return &terraform.Options{
        TerraformDir: terraformDir,

        Vars: map[string]interface{}{
            "db_name": fmt.Sprintf("test%s", uniqueId),
            "db_password": "password",
        },

        BackendConfig: map[string]interface{}{
            "bucket": bucketForTesting,
            "region": bucketRegionForTesting,
            "key": dbStateKey,
            "encrypt": true,
        },
    }
}
```

请根据你的实际值配置 `bucketForTesting` 和 `bucketRegionForTesting` 变量。还可以在 AWS 测试账户中创建一个 S3 bucket 用作 `backend`, 因为 `key` 的配置 (bucket 中的路径) 包括 `uniqueId`, 可以为每个测试提供一个足够独特的值。

下一步是对预发布环境中的 `hello-world-app` 模块进行一些更新。打开并编辑 `live/stage/services/hello-world-app/variables.tf` 文件。添加以下输入变量：`db_remote_state_bucket`、`db_remote_state_key` 和 `environment`。

```
variable "db_remote_state_bucket" {
  description      = "The name of the S3 bucket for the database's remote state"
  type            = string
}

variable "db_remote_state_key" {
  description      = "The path for the database's remote state in S3"
  type            = string
}

variable "environment" {
  description      = "The name of the environment we're deploying to"
  type            = string
  default          = "stage"
}
```

通过 `live/stage/services/hello-world-app/main.tf` 文件，将这些值传递到 `hello-world/hello-world-app` 模块。

```
module "hello_world_app" {
  source      = "../../../../../modules/services/hello-world-app"

  server_text      = "Hello, World"

  environment      = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key

  instance_type     = "t2.micro"
  min_size          = 2
  max_size          = 2
  enable_autoscaling = false
}
```

现在可以实现 `createHelloOpts` 方法了。

```
func createHelloOpts(
  dbOpts *terraform.Options,
  terraformDir string) *terraform.Options {
```

```

    return &terraform.Options{
        TerraformDir: terraformDir,
        Vars: map[string]interface{}{
            "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
            "db_remote_state_key": dbOpts.BackendConfig["key"],
            "environment": dbOpts.Vars["db_name"],
        },
    }
}

```

将 `db_remote_state_bucket` 和 `db_remote_state_key` 参数设置为与 `mysql` 模块中的 `BackendConfig` 参数相同的值。确保 `hello-world-app` 模块读取的状态文件一定是 `mysql` 模块写入的状态文件。将 `environment` 变量设置为 `db_name`，可以使所有资源都以相同的命名规则来区分。

最后，实现 `validateHelloApp` 方法。

```

func validateHelloApp(t *testing.T, helloOpts *terraform.Options) {
    albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)
    maxRetries := 10
    timeBetweenRetries := 10 * time.Second

    http_helper.HttpGetWithRetryWithCustomValidation(
        t,
        url,
        maxRetries,
        timeBetweenRetries,
        func(status int, body string) bool {
            return status == 200 &&
                strings.Contains(body, "Hello, World")
        },
    )
}

```

此方法使用 `http_helper` 包，与单元测试几乎一样，唯一不同的是，这次使用了 `http_helper.HttpGetWithRetryWithCustomValidation` 方法。这个方法允许用户为 HTTP 响应状态代码和响应内容自定义验证规则。这是十分必要的，因为 `hello-world-app` 的

User Data 脚本返回的是包含其他字符的 HTML 格式，因此需要验证 HTTP 响应内容是否包含（而不是完全等于）“Hello,World”字符串。

好了，现在可以运行集成测试，来验证它是否可以正常工作了。

```
$ go test -v -timeout 30m -run "TestHelloWorldAppStage"  
(...)  
PASS  
ok      terraform-up-and-running    795.63s
```

到此为止，我们有了一个可以用来检查几个模块是否协同工作的集成测试。集成测试比单元测试更为复杂，所花费的时间是单元测试的两倍（需要 10 ~ 15min，而不是 4 ~ 5min）。总的来说，这个时间是很难缩短的，因为瓶颈来自 AWS 部署和销毁 RDS、ASG、ALB 等设施所需的时间。但是在某些情况下，可以通过使用测试阶段（*test stages*），减少测试代码的执行时间。

#### 测试阶段

如果我们来查看集成测试的代码，会注意到它包含 5 个不同的阶段。

1. 在 mysql 模块上运行 `terraform apply` 命令。
2. 在 `hello-world-app` 模块上运行 `terraform apply` 命令。
3. 运行验证代码以确保一切正常。
4. 在 `hello-world-app` 模块上运行 `terraform destroy` 命令。
5. 在 mysql 模块上运行 `terraform destroy` 命令。

当在 CI 环境中运行这些测试时，需要从头到尾运行所有阶段。如果是在本地开发环境中运行，当迭代更改代码时，无须运行所有这些阶段。如果你只对 `hello-world-app` 模块进行频繁更改，但每次更改后却要重新运行整个测试。这意味着对从未改动过的 `mysql` 模块也要进行部署和销毁，导致每次测试都会增加 5 到 10min 的时间消耗。

理想情况下，工作流程应如下所示。

1. 在 mysql 模块上运行 `terraform apply` 命令。
2. 在 hello-world-app 模块上运行 `terraform apply` 命令。
3. 现在，开始迭代开发。
  - a. 更改 hello-world-app 模块。
  - b. 重新在 hello-world-app 模块上运行 `terraform apply` 命令来部署更新。
  - c. 运行验证确保一切正常。
  - d. 如果一切正常，请继续进行下一步。如果不是，请返回到步骤 3 (a)。
4. 在 hello-world-app 模块上运行 `terraform destroy` 命令。
5. 在 mysql 模块上运行 `terraform destroy` 命令。

能够快速执行第 3 步中的循环，是使用 Terraform 进行快速迭代开发的关键。为此，我们需要将测试代码分为几个阶段，选择要执行的阶段，以及可以跳过的阶段。

Terratest 通过 `test_structure` 软件包（请运行 `dep ensure` 命令将其添加）提供了对测试阶段的支持。做法是将测试的每个阶段都包装在带有名称的函数中，然后可以通过设置环境变量来指示 Terratest 跳过其中一些函数。每个测试阶段都将测试数据存储在磁盘上，在后续的测试步骤中可以从磁盘上再次读取它们。让我们在 `test/hello_world_integration_test.go` 文件中尝试一下。先编写测试的框架，后面再逐步完成内容。

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
    t.Parallel()
    // 本书为了更好地展示代码，使用了简短的变量名
    stage := test_structure.RunTestStage

    // 部署 MySQL DB
    defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
    stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

    // 部署 hello-world-app
```

```
    defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
    stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

    // 验证 hello-world-app 是否正常工作
    stage(t, "validate_app", func() { validateApp(t, appDirStage) })
}
```

代码结构与以前相同：部署 mysql、部署 hello-world-app、验证 hello-world-app、销毁 hello-world-app（通过 `defer` 在代码结束时运行）、销毁 mysql（通过 `defer` 在代码结束时运行）。只不过现在每个阶段都包含在 `test_structure.RunTestStage` 方法中。`RunTestStage` 方法包括 3 个参数。

`t`

第 1 个参数值 `t` 是 Go 传递给每一个自动测试的参数。这个参数可以用来管理测试状态。例如，通过调用 `t.Fail()` 来使测试失败。

#### 阶段名称

第 2 个参数用来定义测试阶段的名称。下面的示例中将展示如何使用阶段名称跳过测试阶段。

#### 要执行的代码

第 3 个参数是测试阶段中将要执行的代码。可以是任何函数。

现在让我们来实现每个测试阶段的函数，从 `deployDb` 开始。

```
func deployDb(t *testing.T, dbDir string) {
    dbOpts := createDbOpts(t, dbDir)

    // 将数据保存到磁盘，以便后续在测试阶段可以将数据读回
    test_structure.SaveTerraformOptions(t, dbDir, dbOpts)

    terraform.InitAndApply(t, dbOpts)
```

同以前一样，部署 mysql 的代码调用 `createDbOpts` 和 `terraform.InitAndApply` 函数。唯一的新变化是，在这两个步骤之间，通过调用 `test_structure.SaveTerraformOptions`，将 `dbOpts` 中的数据写入磁盘，以便后续测试阶段可以读取它。例如，这里是对 `teardownDb`

函数的实现。

```
func teardownDb(t *testing.T, dbDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    defer terraform.Destroy(t, dbOpts)
}
```

该函数使用 `test_structure.LoadTerraformOptions` 将之前通过 `deployDb` 函数保存在磁盘中的数据加载到 `dbOpts` 变量。之所以需要通过硬盘而不是内存传递数据，是因为不同测试会执行不同的测试阶段。因此，不同的测试阶段可能属于不同的进程。在本章稍后可以看到，在前几次运行 `go test` 中，你可能会运行 `deployDb` 但跳过 `teardownDb`，在后面的运行中又会反过来，只运行 `teardownDb` 但跳过 `deployDb`。为了确保这些测试运行使用相同的数据，必须将该数据库的信息通过磁盘存储进行传递。

现在让我们来实现 `deployHelloApp` 函数。

```
func deployApp(t *testing.T, dbDir string, helloAppDir string) {
    dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
    helloOpts := createHelloOpts(dbOpts, helloAppDir)

    // 将数据保存到磁盘，以便后续在测试阶段可以将数据读回
    test_structure.SaveTerraformOptions(t, helloAppDir, helloOpts)

    terraform.InitAndApply(t, helloOpts)
}
```

函数重用了之前的 `createHelloOpts` 函数，并对其调用 `terraform.InitAndApply`。同样，唯一的新增加的行为是，利用 `test_structure.LoadTerraformOptions` 从磁盘加载 `dbOpts` 变量，使用 `test_structure.SaveTerraformOptions` 将 `helloOpts` 保存到磁盘。在这一点上，你可能会猜到实现 `teardownApp` 方法如下所示。

```
func teardownApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
    defer terraform.Destroy(t, helloOpts)
}
```

`validateApp` 方法的实现如下所示。

```
func validateApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
```

```
    validateHelloApp(t, helloOpts)
}
```

总体而言，测试代码和原来的集成测试基本一致，只是每个阶段被包裹在 `test_structure.RunTestStage` 中，并且需要一些额外工作，比如通过硬盘保存和加载数据。通过这些简单的更改，可以实现一个重要的功能：通过设置环境变量 `SKIP_foo=true`，指示 Terratest 跳过名为 `foo` 的测试阶段。让我们通过一个典型的开发工作流程，来解释它的工作原理。

首先是运行测试，跳过两个销毁阶段，使 `mysql` 和 `hello-world-app` 模块在测试结束后被保留。因为销毁阶段分别是 `teardown_db` 和 `teardown_app`，所以需要分别设置 `SKIP_teardown_db` 和 `SKIP_teardown_app` 环境变量，以指示 Terratest 跳过这两个阶段。

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)
```

```
The 'SKIP_deploy_db' environment variable is not set,
so executing stage 'deploy_db'.
```

```
(...)
```

```
The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.
```

```
(...)
```

```
The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.
```

```
(...)
```

```
The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.
```

```
(...)
```

```
The 'teardown_db' environment variable is set,
so skipping stage 'deploy_db'.
```

```
(...)  
PASS  
ok terraform-up-and-running 423.650s
```

现在开始对 `hello-world-app` 模块的迭代修改，每次进行更改时，都可以重新运行测试。这一次，不仅要跳过销毁阶段，还要跳过 `mysql` 模块的部署阶段（因为 `mysql` 仍在运行），最终执行的代码是 `deploy app` 和验证 `hello-world-app` 模块。

```
$ SKIP_teardown_db=true \  
  SKIP_teardown_app=true \  
  SKIP_deploy_db=true \  
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'  
(...)  
  
The 'SKIP_deploy_db' environment variable is set,  
so skipping stage 'deploy_db'.  
(...)  
  
The 'deploy_app' environment variable is not set,  
so executing stage 'deploy_db'.  
(...)  
  
The 'validate_app' environment variable is not set,  
so executing stage 'deploy_db'.  
(...)  
  
The 'teardown_app' environment variable is set,  
so skipping stage 'deploy_db'.  
(...)  
  
The 'teardown_db' environment variable is set,  
so skipping stage 'deploy_db'.  
(...)  
  
PASS  
ok terraform-up-and-running 13.824s
```

现在这些测试的运行速度快了很多：每次更改后无须等待 10 到 15min，而是可以在 10 到 60s（取决于更改内容）内，完成更改后的测试运行。在开发过程中这个步骤可能会重复

运行数十次，甚至数百次，因此可以节省大量时间。

一旦对 `hello-world-app` 模块的修改达到了预期效果，就可以清理所有内容了。再次运行测试，这次将跳过部署和验证阶段，仅执行销毁阶段。

```
$ SKIP_deploy_db=true \
  SKIP_deploy_app=true \
  SKIP_validate_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'
(...)

The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.
(...)

The 'SKIP_deploy_app' environment variable is set,
so skipping stage 'deploy_app'.
(...)

The 'SKIP_validate_app' environment variable is set,
so skipping stage 'validate_app'.
(...)

The 'SKIP_teardown_app' environment variable is not set,
so executing stage 'teardown_app'.
(...)

The 'SKIP_teardown_db' environment variable is not set,
so executing stage 'teardown_db'.
(...)

PASS
ok terraform-up-and-running 340.62s
```

通过使用测试阶段，自动测试中可以获得快速反馈，从而大大提高迭代开发的速度和质量。或许这对 CI 环境中的测试时间影响不大，但是对开发环境的影响是巨大的。

## 重试

定期执行针对基础设施代码的自动测试，可能会发现一个问题：不稳定的测试。也就是说，测试偶尔会因为临时的原因而失败，如 EC2 实例偶尔会无法启动，或发生 Terraform 最终

一致性的错误，或与 S3 发生 TLS 握手错误。基础设施世界是一个混乱的地方，因此应该学会如何处理测试中遇到的间歇性故障。

为了使测试更具弹性，可以对已知错误进行重试。例如，在编写本书时，遇到过以下错误类型，尤其是在并行运行许多测试的时候。

```
* error loading the remote state: RequestError: send request failed
Post https://xxx.amazonaws.com/: dial tcp xx.xx.xx.xx:443:
connect: connection refused
```

为了使测试在遇到此类错误时更加可靠，可以通过 `terraform.Options` 的 `MaxRetries`、`TimeBetweenRetries` 和 `RetryableTerraformErrors` 参数，在 `Terratest` 中启用重试机制。

```
func createHelloOpts(
    dbOpts *terraform.Options,
    terraformDir string) *terraform.Options {

    return &terraform.Options{
        TerraformDir: terraformDir,

        Vars: map[string]interface{}{
            "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
            "db_remote_state_key": dbOpts.BackendConfig["key"],
            "environment": dbOpts.Vars["db_name"],
        },
        // 对已知错误，重试 3 次，每次间隔 5s
        MaxRetries: 3,
        TimeBetweenRetries: 5 * time.Second,
        RetryableTerraformErrors: map[string]string{
            "RequestError: send request failed": "Throttling issue?",
        },
    }
}
```

在 `RetryableTerraformErrors` 参数中，可以通过定义一个对已知错误的映射，来授权进行重试：该映射的关键是要在日志中查找错误消息（在此处可以使用正则表达式），该映射的值是 `Terratest` 在匹配到错误并准备开始重试时，将在日志中追加的信息。现在，当测试代码遇到这些已知错误，用户就应该在日志中看到相关信息，然后进程休眠若干秒

(`TimeBetweenRetries` 定义), 命令将被重新运行。

```
$ go test -v -timeout 30m  
(...)  
  
Running command terraform with args [apply -input=false -lock=false -auto-approve]  
(...)  
  
* error loading the remote state: RequestError: send request failed  
Post https://s3.amazonaws.com/: dial tcp 11.22.33.44:443:  
connect: connection refused  
(...)  
  
'terraform [apply]' failed with the error 'exit status code 1'  
but this error was expected and warrants a retry. Further details:  
Intermittent error, possibly due to throttling?  
(...)  
  
Running command terraform with args [apply -input=false -lock=false -auto-approve]
```

## 端到端测试

我们已经完成了单元测试和集成测试的学习,最终的测试类型是端到端测试。在 Ruby Web 服务器示例中,端到端测试可能包括部署 Web 服务器及依赖的任何数据库,然后使用 Selenium 之类的工具从 Web 浏览器进行测试。Terraform 基础设施的端到端测试是十分类似的:将所有内容部署到模拟生产的环境中,从最终用户的角度进行测试。

尽管我们可以使用与集成测试完全相同的策略来编写端到端测试,即创建几十个测试阶段,然后运行 `terraform apply` 命令,进行一些验证,之后运行 `terraform destroy` 命令。但实际上很少这样做,原因与测试金字塔有关,如图 7-1 所示。

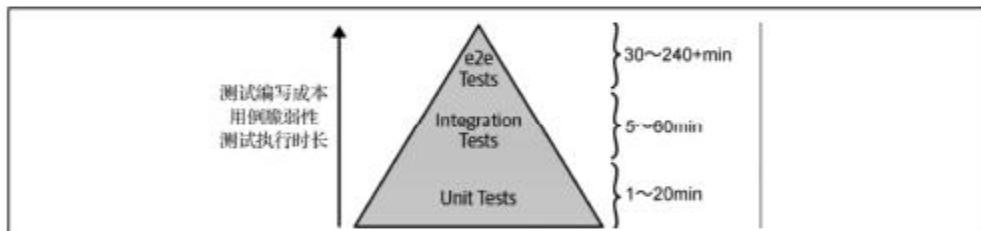


图 7-1：测试金字塔

测试金字塔表达的思想是，应该进行大量单元测试（金字塔的底部）、较少数量的集成测试（金字塔的中间）和更少数量的端到端测试（金字塔的顶部）。因为随着金字塔的上升，编写测试的成本、复杂性、测试的脆弱性，以及测试的运行时间都在增加。

这也是测试要点 #5：较小的模块更易于测试。

在前面的示例部分中，即使测试相对简单的 `hello-world-app` 模块，也需要大量的工作，包括命名空间的隔离、依赖项注入、重试、错误处理和测试阶段。对于更大、更复杂的基础设施，这只会变得更加困难。因此，由于金字塔的底部提供了最快、最可靠的反馈回路，应该针对金字塔底部进行尽可能多的测试。

实际上，当用户到达测试金字塔的顶端时，测试一个从头开始部署的复杂的体系设施，是十分不现实的，主要有如下两个原因。

#### 太慢

从头开始部署整个体系设施，然后再全部销毁，可能会持续非常长的时间——大约几个小时。花费大量时间的测试套件提供的价值相对较小，因为反馈回路太慢了。只能在夜间运行这样长时间的测试套件，意味着你将在早上收到有关测试失败的报告，再经过一段时间调查，提交新的修复，然后等待第二天查看修复是否有效。这样一来，每天只能尝试进行一次错误修复。这种情况下，开发人员开始相互指责，说服管理人员即使在测试失败的情况下也要进行部署，最终会完全忽略测试失败。

#### 过于脆弱

如上一节所述，基础设施世界很混乱。随着部署的基础设施数量的增加，碰到间歇性、不稳定的问题的概率也随之增加。例如，假设单个资源（EC2 实例）有千分之一的机会（0.1%）发生间歇性错误故障（DevOps 世界中的实际故障率可能更高），这意味着部署单个资源的测试时，没有任何间歇性错误的可能性为 99.9%。那么部署两个资源的测试又如何呢？为了使测试成功，让两个资源在部署时，同时不出现间歇性错误，需要相乘来计算概率： $99.9\% \times 99.9\% = 99.8\%$ ；使用 3 个资源，是  $99.9\% \times 99.9\% \times 99.9\% = 99.7\%$ ；使用  $N$  个资源，公式为  $99.9\%^N$ 。

现在让我们考虑不同类型的自动测试。如果对部署了 20 个资源的单个模块进行单元测试，那么成功率是  $99.9\%^{20} = 98.0\%$ 。这意味着 100 个测试中将有 2 个测试失败。在添加一些重试后，通常这些测试会变得相当可靠。假设你对 3 个模块进行了集成测试，这些模块部署了 60 个资源。现在成功的概率是  $99.9\%^{60} = 94.1\%$ 。同样，使用足够的重试逻辑，可以使这些测试足够稳定。那么，如果编写一个端到端的测试来部署整个基础设施（包括 30 个模块或大约 600 个资源）会怎么样？成功的概率是  $99.9\%^{600} = 54.9\%$ 。这意味着运行的测试中将近一半会由于暂时原因而失败！

可以通过重试来处理其中一些错误，但很快就会变成永无休止的砸田鼠游戏：虽然为 TLS 握手超时添加了一个重试，但 Packer 模板中的 APT 仓库又离线了；将重试添加到 Packer 构建中，但是由于 Terraform 最终一致性错误而使构建失败；正当你修复上一个错误时，由于短暂的 GitHub 服务中断，构建本身又失败了。而且由于端到端测试需要花费很长时间，因此每天只能进行一到两次尝试来解决这些问题。

实际上，大部分拥有复杂基础设施的公司，运行端到端测试很少从头部署所有内容。取而代之的是如下的端到端测试策略。

1. 部署一个持久的、类似于生产的测试环境，并保持该环境运行。
2. 每当有人对你的基础设施进行更改时，端到端测试都会执行以下操作。
  - a. 将基础设施的更改部署于测试环境。
  - b. 针对测试环境运行验证（例如，从最终用户的角度使用 Selenium 测试代码），确保一切正常。

通过将端到端测试策略更改为仅部署增量修改，可以将测试时部署的资源数量从几百个减少到几个，使这些测试更快、更稳定。

而且，这种端到端测试的增量部署方法更确切地模仿了生产中部署这些更改的过程。毕竟，我们不会为每个更改都销毁和重建整个生产环境，一般都会增量地部署每个更改。这种端到端测试策略提供了巨大的优势：不仅可以测试基础设施是否正常工作，还可以测试该基础设施的部署过程是否正常工作。

例如，当你需要测试一个关键属性——在不停机的情况下对基础设施进行更新时。之前创建的 `asg-rolling-deploy` 模块声称能够实现零停机、滚动部署。但是如何验证这一点呢？让我们添加一个自动测试。

只需对 `test/hello_world_integration_test.go` 文件中的测试进行一些调整就可以完成这个工作，因为 `hello-world-app` 模块的后台使用了 `asg-rolling-deploy` 模块。首先是在 `live/stage/services/hello-world-app/variables.tf` 文件中公开一个名为 `server_text` 的输入变量。

```
variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

通过 `live/stage/services/helloworld-app/main.tf` 文件，将这个变量传递到的 `hello-worldhello-world-app` 模块。

```
module "hello_world_app" {
  source = "../../../../../modules/services/hello-world-app"

  server_text = var.server_text

  environment          = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key   = var.db_remote_state_key

  instance_type        = "t2.micro"
  min_size              = 2
  max_size              = 2
  enable_autoscaling    = false
}
```

`hello-world-app` 模块在用户数据脚本中使用了 `server_text` 参数，所以每次对参数的更改，将强制开始一个（希望如此）零停机部署。在 `test/hello_world_integration_test.go` 文件中，在 `validate_app` 测试阶段之后，增加一个名为 `redeploy_app` 的测试阶段。

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
  t.Parallel()

  // 本书为了更好地展示代码，使用了简短的变量名
```

```

stage := test_structure.RunTestStage

// 部署 MySQL 数据库
defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

// 部署 hello-world-app
defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

// 验证 hello-world-app
stage(t, "validate_app", func() { validateApp(t, appDirStage) })

// 重新部署 hello-world-app
stage(t, "redeploy_app", func() { redeployApp(t, appDirStage) })
}

```

接下来，实现新的 redeployApp 方法。

```

func redeployApp(t *testing.T, helloAppDir string) {
    helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)

    albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
    url := fmt.Sprintf("http://%s", albDnsName)

    // 每隔 1s，检查一次应用程序是否响应 200 OK
    stopChecking := make(chan bool, 1)
    waitGroup, _ := http_helper.ContinuouslyCheckUrl(
        t,
        url,
        stopChecking,
        1*time.Second,
    )

    // 更新服务器的文本，并重新部署
    newServerText := "Hello, World, v2!"
    helloOpts.Vars["server_text"] = newServerText
    terraform.Apply(t, helloOpts)

    // 确保新版本已部署
    maxRetries := 10
    timeBetweenRetries := 10 * time.Second
    http_helper.HttpGetWithRetryWithCustomValidation(

```

```

        t,
        url,
        maxRetries,
        timeBetweenRetries,
        func(status int, body string) bool {
            return status == 200 &&
                   strings.Contains(body, newServerText)
        },
    )

    // 停止检查
    stopChecking <- true
    waitGroup.Wait()
}

```

此方法执行了以下操作。

1. 使用 `Terratest http_helper.ContinuouslyCheckUrl` 辅助函数在后台执行 goroutine 程序（由 Go 运行时程序管理的轻量级线程），该后台程序每一秒对给定的 ALB URL 执行 HTTP GET 查询，如果任何时候收到的响应不是 200 OK，则标注测试失败。
2. 更新 `server_text` 变量，并运行 `terraform apply` 命令开始滚动部署。
3. 部署完成后，确保服务器使用新的 `server_text` 变量内容进行响应。
4. 停止持续检查 ALB URL 的 goroutine 程序。

这个测试运行的最后阶段，将在日志中看到持续地对 ALB 的 HTTP GET 调用，其中散布着 `terraform apply` 命令进行滚动部署的日志输出。

```

$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

Making an HTTP GET call to URL
http://hello-world-test-thLMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thLMBF-168938547.us-east-2.elb.amazonaws.com

```

```
Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

(...)

Running command terraform with args
[apply -input=false -lock=false -auto-approve]

(...)

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

(...)

PASS
ok terraform-up-and-running 551.04s
```

如果测试通过，意味着 `hello-world-app` 和 `asg-rolling-deploy` 模块能够实现零停机的滚动部署！每次端到端测试将变更的增量部署到测试环境时，都可以使用这个策略来确保部署不会导致停机。

## 其他测试方法

本章的大部分内容都集中在使用 `Terratest` 方法进行自动测试上，但还有其他两个类别的测试方法，可以进行了解。

- 静态分析
- 属性测试

不同类型的自动测试（单元测试、集成测试、端到端测试）捕获不同类型的错误。每种测试方法也将捕获不同类型的错误，因此用户需要同时使用其中几种来进行测试，以获得最佳结果。让我们逐一浏览这些新类别。

## 静态分析

有几种工具可以在不运行 Terraform 代码的情况下对其进行分析，具体如下所示。

### `terraform validate`

这是 Terraform 内置的命令，可用于检查 Terraform 语法和类型（有点像编译器）。

### `tflint`（见参考资料第 7 章[10]）

这是一种专门用于 Terraform 的“lint”工具，它可以根据一组内置规则，扫描 Terraform 代码并捕获常见错误和潜在错误。

### `HashiCorp Sentinel`（见参考资料第 7 章[11]）

一个“策略即代码”框架，可以针对各种 HashiCorp 工具加强规则。例如，创建一个策略，禁止 Terraform 代码中允许入站访问安全组规则 0.0.0.0/0。在撰写本文时，Sentinel 仅适用于 HashiCorp Enterprise 产品，包括 Terraform Enterprise。

## 属性测试

有许多测试工具专注于验证基础设施的特定“属性”。

- `kitchen-terraform`（见参考资料第 7 章[12]）
- `rspec-terraform`（见参考资料第 7 章[13]）
- `Serverspec`（见参考资料第 7 章[14]）
- `Inspec`（见参考资料第 7 章[15]）
- `goss`（见参考资料第 7 章[16]）

大多数工具通过一个简单的领域特定语言（*domain-specific language, DSL*），可以检查已经部署的基础设施是否符合某种规范。例如，如果对部署 EC2 实例的 Terraform 模块进行测试，可以使用以下 `Inspec` 代码来验证实例中特定文件是否具有适当的权限、是否已安装某些依赖项、是否正在侦听特定端口。

```
describe file('/etc/myapp.conf') do
  it { should exist }
  its('mode') { should cmp 0644 }
end
```

```
describe apache_conf do
    its('Listen') { should cmp 8080 }
end

describe port(8080) do
    it { should be_listening }
end
```

这些工具的优点是 DSL 语言趋向于简洁、易于使用，并提供了一个高效的、声明式的方法来验证大量基础设施的属性。这对于强制执行要求清单非常有用，尤其是在合规性方面（例如，PCI 合规、HIPAA 合规等）。这些工具的缺点是：可能所有属性检查都通过了，但基础设施却仍然无法正常工作！作为对比，验证这些相同属性的“Terratest 方式”是向服务器发出 HTTP 请求，查看是否获得了预期的响应。

## 小结

基础设施世界中的一切都在不断变化着：Terraform、Packer、Docker、Kubernetes、AWS、Google Cloud、Azure 等都是移动目标。这意味着基础设施代码会很快地“腐烂”，或者换一种说法：

没有自动测试的基础设施代码其实已经损坏。

这既是格言又是字面意思。每当我花时间编写基础设施代码时，无论花了多少精力来保持代码的清洁、手动测试和进行代码审查，只要开始编写自动测试，又会发现大量的、重大的错误。当花时间进行自动测试后，毫无例外地会发生一些神奇的事情，它消除了本来不会发现的问题（但客户总会发现的问题）。而且，不仅在首次添加自动测试时能够发现错误，如果坚持每次提交后都能运行测试，将会不断发现新的错误，因为 DevOps 世界总是快速地变化着。

我为基础设施代码中编写的自动测试，不仅捕获了我的代码错误，还捕获了使用工具中的重大错误，包括 Terraform、Packer、Elasticsearch、Kafka、AWS 等。如本章所述，编写自动测试并不容易：需要花费大量精力来编写测试代码，花费更多的精力来维护它们，并添加足够的重试逻辑以使其更加可靠，并且还要付出更多的努力来保持测试环境的清洁，以控制成本，但最终这些都是值得的。

当我构建一个模块来部署数据库时，每次代码提交之后，我的测试都会以各种配置启动该数据库的十几个副本，写入数据，读取数据，然后销毁所有内容。每次这些测试通过时，都会增加我对自己代码的信心。除此之外，自动测试会让我睡得更好。我花在重试逻辑和解决最终一致性问题上的时间，换来了我不用在凌晨 3 点起床处理生产环境的停机故障。



### 本书也有测试！

本书中的所有代码示例也都经过了测试。你可以在参考资料第 7 章[17]找到所有代码示例及其相应的测试。

在本章中，用户了解了测试 Terraform 代码的基本过程，包括以下主要内容。

#### 测试 *Terraform* 代码时，没有本地环境

因此，需要通过将实际资源部署到一个或多个隔离的沙箱环境中，进行手动测试。

#### 定期清理沙箱环境

否则，环境将变得难以管理，并且成本将无法控制。

#### 不能对 *Terraform* 代码进行纯粹的单元测试

因此，必须通过编写代码，将实际资源部署到一个或多个隔离的沙箱环境中，来执行所有自动测试。

#### 你必须管理所有资源的命名空间

这可以确保并行的多个测试不会相互冲突。

#### 较小的模块测试起来更简单、更快速

这也是第 6 章的主要内容之一，在本章中有必要重复：较小的模块更易于创建、维护、使用和测试。

在第 8 章中，我们将学习如何将 Terraform 代码和自动测试代码合并到团队工作流程中，包括如何管理环境、如何配置持续集成/持续开发管道等。

# 如何在团队环境下使用 Terraform

虽然在阅读本书和研究代码示例的过程中，你一直在独自工作。但在现实世界中，你很可能是团队中的一员，这会带来许多新的挑战。你或许需要找到一种方法说服团队来使用 Terraform 和其他基础设施即代码（IaC）工具。你可能需要同时面对多个用户，帮助他们理解、使用和修改你编写的 Terraform 代码。你可能需要搞清楚如何将 Terraform 融入其他的技术，并使之成为公司工作流程的一部分。

在本章中，我将深入介绍让 Terraform 和 IaC 工具在团队环境下产生效能的关键步骤。

- 在团队中实施 IaC
- 部署应用程序代码的工作流程
- 部署基础设施代码的工作流程
- 将上述各点整合在一起

让我们逐一学习每一个主题。

## 在团队中实施 IaC

如果你所在的团队已经习惯了手工管理所有基础设施，那么转换为通过代码管理基础设施，不仅需要引入新的工具或技术，还需要团队在文化和流程上的改变。改变文化和流程是一项重大的工程，尤其是在大型公司中，由于每个团队的文化和流程略有不同，因此不能“一刀切”，但是这里有一些技巧在大多数情况下会有所帮助。

- 说服老板
- 逐步开展工作
- 给团队学习的时间

### 说服老板

我在很多公司目睹了相同的故事：一个开发人员发现了 Terraform，在强大功能的启发下，充满热情地完成了一些初步工作，于是激动地向所有人展示 Terraform。可是老板却说：“不”。当然，开发人员感到沮丧和失望：为什么其他人都看不到这样做好处？我们可以使一切工作自动化！我们可以避免很多错误！我们还能偿还所有欠下的技术债务！你们怎么这么盲目？

问题在于，尽管该开发人员看到了采用 Terraform 之类 IaC 工具的好处，但他们却没有看到所有相关的开销。以下是采用 IaC 的部分开销。

### 技能差距

迁移到 IaC 意味着运维团队需要花费大部分时间来编写大量代码：Terraform 模块、Go 测试、Chef recipes 等。尽管有些运维工程师喜欢编码且乐于接受改变，但其他工程师却发现这是一个艰难的任务。许多运维工程师和系统管理员已经习惯了手动进行更改，偶尔会写一些简短的脚本，但接近全职地进行软件开发工作，可能需要学习大量新技能或需要直接雇用新人。

### 新工具

软件开发人员十分依赖他们使用的工具。有些人对此近乎虔诚。每次引入新工具时，一些开发人员会为有机会学习新知识而感到兴奋，而其他开发人员更愿意坚持已经掌握的知识和工具，拒绝花费大量时间和精力来学习新的语言和技术。

### 转变思维方式

如果团队成员已经习惯了手动管理基础设施，他们会更愿意直接实施所有变更。例如，通过 SSH 连接到服务器，执行一些命令。迁移到 IaC 要求转变思维方式，因为现在是通过间接方式进行更改的：首先编辑代码、提交代码，然后让某些自动化过程来部

署更改。新增加的这层“间接方式”是令人沮丧的。对于简单的任务，它比直接实施部署的方式要慢，尤其是在团队成员仍在学习新的 IaC 工具的初期，差距更加明显。

### 机会成本

当你将时间投入到一个项目中时，其实另一方面也就是放弃了其他项目。哪些项目是可以暂停的，以便迁移到 IaC？这些项目重要吗？

团队中的一些开发人员看到这个列表后会十分兴奋。但是更多的人会叹气，包括你的老板。学习新技能、掌握新工具，以及采用新的思维方式或许有好处，也或许没有。但有一点可以肯定：这不是免费的。采用 IaC 是一项重大投资，与任何投资一样，不仅需要考虑潜在的优点，更要考虑缺点。

老板对机会成本尤其敏感。任何经理的主要职责之一就是确保团队工作在最高优先级的项目之上。当你出现并开始兴奋地谈论 Terraform 时，你的老板可能真正听到的是：“天哪，这听起来像是一个重大的承诺，要花多少时间？”这并不是说明老板无视 Terraform 的功能，而是如果将时间花在这上面之后，就没有时间部署搜索团队几个月来一直在询问的新应用，也没有时间准备支付卡行业（PCI）审计，或者分析从上周开始的意外停机问题了。因此，如果你想说服老板和团队应该采用 IaC，那么你的目标并不是要证明它的价值，而是要分析如何能为团队带来比当前其他项目更大的价值。

最没用的一种方法就是，仅仅列出你最青睐的 IaC 工具的功能：例如，Terraform 是声明性的、支持多云环境的、开源的。这是开发人员应该向销售人员学习的领域之一。大多数销售人员都知道，仅专注于产品功能，通常是低效的销售方法。一种更好的销售技巧是关注收益：与其谈论产品可以做什么（“产品 X 可以做 Y！”），不如多谈谈客户使用该产品可以做什么（“您可以使用产品 X 来做 Y！”）。换句话说，向客户展示你的产品可以为他们带来什么样的超级功能。

所以，与其告诉老板 Terraform 是声明性的，不如说明团队将如何使用它来更快地完成项目。与其谈论 Terraform 是支持多云环境的，不如描述一下，如果有一天迁移云环境时，老板不需要担心潜在的工具变更。与其向老板解释 Terraform 是开源的，不如帮助老板了解，从一个活跃的大型开源社区中，可以很容易地为团队雇用开发人员。

关注利益是一个很好的开始，但是最好的销售人员知道一个更有效的策略：关注问题。当你观察一位出色的销售员与客户交谈时，会发现实际上大部分的谈话是由客户进行的。销售人员将主要时间用于倾听和寻找一个答案：客户试图解决的关键问题是什么？最大的痛点是什么？最好的销售人员不会尝试销售某种功能或收益，而是尝试解决其客户的问题。如果该解决方案恰好包含他们所销售的产品，那么情况会更好，但真正的重点在于解决问题，而不是销售。

通过与老板的交谈，可以了解本季度及本年度最重要的问题是什么，你或许会发现 IaC 无法解决这些问题。不过没关系，不是每个团队都需要 IaC！作为一本与 Terraform 有关的书籍的作者，这么说可能有些不同寻常。不过采用 IaC 的成本的确相对较高，尽管在某些场景下会收获长期的回报，但其他场景则不是必需的。例如，一家只有一个运维员工的小型创业公司，或者你正在开发的原型可能在几个月后就会被抛弃，或者只是业余时间在一个自己喜欢的项目，在以上场景中，采用手动方式管理基础设施是正确的选择。有时，即使 IaC 非常适合你的团队，但由于资源有限，优先考虑从事其他项目仍然是正确的选择。

如果确实发现 IaC 可以解决老板关注的关键问题之一，那么你的目标就是向老板展示 IaC 的全部魅力。例如，老板在本季度最关注的问题就是如何提高正常运行时间。在过去的几个月中，已经发生了多次停机，每次数小时，客户抱怨，CEO 每天催促老板加快修复进度。经过深入研究，你发现其中一半以上的停机是由部署过程中的手动错误引起的：例如，在实施过程中，有人大意地跳过了一个重要步骤，或者服务器配置错误，也或者预发布环境的基础设施状态与生产环境不匹配。

当与老板交谈时，不要只是谈论 Terraform 的功能或好处，请用以下方式开始对话：“我有一个方法可以将停机次数减少一半”。我保证这会引起老板的注意。利用这个好机会，向老板描绘一个全自动的、可靠的、可重用的部署过程是如何避免手动错误，将错误率降低一半的，并描绘当部署自动化后，可以继续添加自动测试，进一步减少潜在的停机错误，使整个公司的部署频率提高两倍。让老板梦想成为第一个告诉首席执行官，你们已经成功地将停机事件减少一半，并将部署频率增加一倍的人。这时再提到，根据研究，你相信可以使用 Terraform 来实现这个美好的“未来世界”。

虽然这并不能保证老板会说“同意”，但是采用这种方法，你的胜率会高得多。如果再采

取逐步开展工作的方法，你的胜率会更大。

## 逐步开展工作

我在职业生涯中学到的最重要的教训之一是：大多数大型软件项目最终失败了。大约有四分之三的小型 IT 项目（小于 100 万美元）成功完成，而只有十分之一的大项目（大于 1000 万美元）最终按时并在预算之内完成，而超过三分之一的大项目从来没有完成。<sup>1</sup>

当一个团队选择实施 IaC 时，不仅尝试包括全部部门，涵盖大量基础设施，而且 IaC 项目常常只是更大计划的一部分。看到这种情景，我总是感到担忧。当听到一家大公司的首席执行官或首席技术官发出如下指令：必须在 6 个月内将所有内容都迁移到云中，必须关闭旧的数据中心，每个人都将“执行 DevOps”（无论这里的 DevOps 定义是什么）时，我不禁摇了摇头。在目睹了很多相似的场景后，我可以毫不夸张地说，这种类型的项目，每一个都无一例外地失败了。不可避免的是，直到两到三年后，公司中的每一个人仍在进行迁移，旧的数据中心仍在运行，没人能说出他们是否实现了 DevOps。

如果想成功部署 IaC，或者在任何其他类型的迁移项目上取得成功，唯一可行的方法就是逐步增量地开展工作。增量主义的关键不仅是将工作分割为一系列的小步骤，而是在后面的步骤即便不会发生的情况下，之前的每一步都带来了自己的价值。

为了了解这一点的重要性，请考虑相反的错误增量主义 (*false incrementalism*)。<sup>2</sup>假设你正在进行一个庞大的迁移项目，其被分为若干个小步骤，但是在最后一步完成之前，该项目没有任何实际价值。例如，第一步是重写前端，但是无法启动它，因为它要依靠新的后端。然后，重写了后端，但是也无法启动该后端，因为它只有在将数据迁移到新的数据存储之后才起作用。最后一步是进行数据迁移。只有完成最后一步后，才可以启动所有内容，并开始从中实现价值。这种等到项目结束时才能获得价值的方式，存在着很大的风险。如果项目被取消、搁置或中途发生了重大变化，尽管前期投入了大量资金，但最终也可能颗粒无收。

1 斯坦迪什集团, CHAOS Manifesto 2013: *Think Big, Act Small*. 2013. (见参考资料第 8 章[1]).

2 丹·米尔斯顿, *How To Survive a Ground-up Rewrite without Losing Your Sanity*. OnStartups.com. 2013. (见参考资料第 8 章[2]).

粒无收。

实际上，这正是许多大型迁移项目正在经历的情况。项目从一开始就大张旗鼓，像大多数软件项目一样，花费的时间比预期的时间要长得多。在此期间，市场发生了变化，原来的利益相关者失去了耐心（例如，首席执行官同意花 3 个月的时间清理技术债务，但是 12 个月过去了还没有结束，而现在的重要任务是发布新产品了），项目的命运是在完成之前被取消。如果使用错误的增量主义，将带来最糟糕的结果：付出了巨大的代价，却没有得到任何回报。

因此，增量主义至关重要。项目的每个部分都应该提供一定的价值，这样即使项目没有完成，无论实现到了哪一步，仍然可以输出一定的价值。最好的方法是，每一次集中解决单一的、小规模的、具体的问题。例如，不要试图做“大爆炸”式的云迁移，而是找出一个存在痛点的、小规模的、特定的应用或团队，只是去迁移它们。或者，不要进行“大爆炸”式的“DevOps”迁移，而是尝试识别单一的、小规模的、具体的问题（部署期间的停运），并针对这个特定问题制定解决方案（通过 Terraform，自动化问题最多的部署步骤）。

如果通过快速解决一个真实的具体问题，让团队收获成功，你会开始赢得口碑。该团队可以成为你的啦啦队员，帮助说服其他团队加入迁移。解决特定的部署问题也会使首席执行官感到高兴，并支持你开展更多基于 IaC 的项目。一个又一个的快速胜利接踵而来。如果可以继续重复这个过程，尽早而经常地交付价值，你将更有可能在更大的迁移项目上取得成功。即使更大的迁移项目最终无法进行下去，至少有一个团队已经取得成功，至少有一个部署过程得到了提高，因此投资仍然收到了回报。

### 给团队学习的时间

我希望到目前为止，你已经很清楚采用 IaC 是一项重大投资，不是一夜之间就能发生的事情。也不是因为经理点头同意，就会神奇完成的项目。只有得到所有人的支持，提供了详尽的学习资源（文档、视频教程，当然还有本书！），并为团队成员安排专门学习的时间，才能实现这一目标。

如果团队没有获得所需的时间和资源，那么 IaC 迁移就不太可能成功。无论你的代码多么出色，如果整个团队都不愿意使用它，也很难发挥作用。

1. 团队中的一名开发人员对 IaC 充满热情，花了几个月的时间编写了漂亮的 Terraform 代码并开始使用它部署大量基础设施。
2. 这名开发人员很有成就感且工作效率得到了提高，但不幸的是，团队的其他成员没有时间学习和采用 Terraform。
3. 这时不可避免的情况发生了：故障中断。
4. 现在，当团队中其他成员需要处理故障中断时，有两个选择：(a) 以手动方式进行修改，解决故障，这通常只需要花费几分钟；(b) 使用 Terraform 来解决故障，由于他们并不熟悉怎么使用，可能需要数小时或数天。如果你的团队成员是理性的，都会选择选项 (a)。
5. 由于手动更改，Terraform 代码不再与实际部署的环境相匹配。因此，即使之后团队中的某个人选择了选项 (b) 尝试使用 Terraform，他们也有可能会遇到一致性的错误。逐渐地，大家会失去对 Terraform 代码的信任，并再次退回到选项 (a)，进行更多的手动更改。这使代码与实际情况更加不同步，因此下一个人遇到奇怪的 Terraform 错误的概率会增加，进入这个怪圈后，团队成员会越来越多地进行手动更改。
6. 在非常短的时间内，每个人都回到了手动执行操作的阶段，花费数月编写的 Terraform 代码将不可用，完全是浪费时间。

这种情况并非假设，我在许多公司都遇到过类似的情况。他们有庞大而昂贵的代码库，里面充斥着精致的 Terraform 代码，这些代码被遗忘在那里。为了避免这种情况，你不仅需要说服老板应该使用 Terraform，还需要为团队中的每个人提供学习该工具的时间并让每个人都精通该工具的使用，这样才能在下次发生故障时，使其通过代码修复比通过手工更容易完成。

明确的、清晰定义的工作流程，可以帮助团队更快地采用 IaC。一个小型团队学习和使用 IaC 时，在开发人员的计算机上临时运行它就足够了。但是随着公司和 IaC 使用量的增长，我们需要定义一个系统的、可重复的、自动化的工作流程，用于描述部署的方式。

# 部署应用程序代码的工作流程

在本节中，我将先介绍应用程序代码（如 Ruby on Rails 或 Java /Spring 应用程序）从开发一直到生产环境部署的典型工作流程。在 DevOps 行业中，这个工作流程已被很好地理解，因此读者可能会熟悉其中的某些部分。在本章的后面，我将讨论基础设施代码（例如 Terraform 模块）从开发到生产环境部署的工作流程。这个工作流程在业界几乎不为人所知。因此将该工作流程与应用程序工作流程进行比较，便于将每个应用程序代码部署步骤转换为类似的基础设施代码部署步骤。

应用程序代码工作流程如下所示。

1. 使用版本控制。
2. 在本地运行代码。
3. 进行代码更改。
4. 提交更改以供评审。
5. 运行自动测试。
6. 合并和发布。
7. 部署。

让我们逐个认识每个步骤。

## 使用版本控制

所有代码都应处于版本控制中，没有例外。这是经典的 Joel Test（见参考资料第 8 章[3]）中排名第一的项目，Joel Test 约 20 年前由乔尔·斯波斯基（Joel Spolsky）发布。在此后唯一改变的是使用 GitHub 之类的工具，使版本控制比以往任何时候都容易，并且可以将越来越多的事物表示为代码。这包括文档（用 Markdown 编写的自述文件）、应用程序配置（用 YAML 编写的配置文件）、规范（用 RSpec 编写的测试代码）、测试（用 JUnit 编写的自动测试）、数据库（用 Active Record 编写的迁移概要），当然还有基础设施。

本书的其余部分，我将假设读者使用 Git 作为版本控制工具。例如，以下是从 Git 存储库中检出本书代码示例的方法。

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

默认情况下，这将检出 `master` 分支的内容，但我推荐使用单独的分支完成开发工作。按照以下方法，使用 `git checkout` 命令，可以创建一个名为 `example-feature` 的分支，同时将当前工作环境切换到该分支。

```
$ cd terraform-up-and-running-code  
$ git checkout -b example-feature  
Switched to a new branch 'example-feature'
```

## 在本地运行代码

现在，代码已下载到计算机上，你可以在本地运行它。例如第 7 章中的 Ruby Web 服务器示例，可以按以下方式运行。

```
$ cd code/ruby/08-terraform/team  
$ ruby web-server.rb  
  
[2019-06-15 15:43:17] INFO WEBrick 1.3.1  
[2019-06-15 15:43:17] INFO ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]  
[2019-06-15 15:43:17] INFO WEBrick::HTTPServer#start: pid=28618 port=8000
```

现在，可以使用 `curl` 命令进行手动测试。

```
$ curl http://localhost:8000  
Hello, World
```

或者，执行以下自动测试。

```
$ ruby web-server-test.rb  
(...)  
Finished in 0.633175 seconds.  
-----  
8 tests, 24 assertions, 0 failures, 0 errors  
100% passed  
-----
```

一个值得关注的现象是，针对应用程序代码的手动测试和自动测试都可以完全在本地计算

机上运行。这与本章后面的基础设施更改的工作流程是完全不同的。

## 进行代码更改

现在已经可以执行应用程序代码，也可以开始对其进行更改了。这是一个迭代的过程：进行更改、重新运行手动或自动测试以查看更改是否有效、进行其他更改、重新运行测试等。

例如，将 *web-server.rb* 的输出更改为“Hello, World v2”，重新启动服务器，然后查看结果。

```
$ curl http://localhost:8000
Hello, World v2
```

重新运行测试来验证是否通过。

```
$ ruby web-server-test.rb

A Workflow for Deploying Application Code | 291

(...)

Failure: test_integration_hello(TestWebServer)
web-server-test.rb:53:in `block in test_integration_hello'
  50: do_integration_test('/', lambda { |response|
  51:   assert_equal(200, response.code.to_i)
  52:   assert_equal('text/plain', response['Content-Type'])
=> 53:   assert_equal('Hello, World', response.body)
  54: })
  55: end
  56:
web-server-test.rb:100:in `do_integration_test'
web-server-test.rb:50:in `test_integration_hello'
<"Hello, World"> expected but was
<"Hello, World v2">

(...)

Finished in 0.236401 seconds.
-----
8 tests, 24 assertions, 2 failures, 0 errors
75% passed
-----
```

你会立即收到错误反馈，表明自动测试代码仍然期望使用旧的输出值。接下来可以对测试代码进行修复。工作流程的理念是优化反馈循环，使更改和验证更改是否有效之间的时间最小化。

工作时，应该定期提交代码，并使用清晰的提交日志解释更改内容。

```
$ git commit -m "Updated Hello,World text"
```

### 提交更改以供评审

最终，代码和测试都按照预期的方式执行，现在该提交更改以进行代码评审了。用户可以使用单独的代码评审工具（如 Phabricator 或 ReviewBoard）来执行这个操作。如果是 GitHub 用户，则可以通过创建 *pull request* 来执行。创建 pull request 有几种不同的方法。其中最简单的方法是通过 `git push` 命令将 `example-feature` 分支的代码改动推送给 `origin`（GitHub 本身），同时 GitHub 会自动返回一个包含 pull request 链接的日志输出。

```
$ git push origin example-feature  
(...)  
remote: Resolving deltas: 100% (1/1), completed with 1 local object.  
remote:  
remote: Create a pull request for 'example-feature' on GitHub by visiting:  
remote: https://github.com/<OWNER>/<REPO>/pull/new/example-feature  
remote:
```

在浏览器中打开这个 URL，填写 pull request 标题和说明，然后单击“创建”按钮。现在，你的团队成员将能够进行变更的评审，图 8-1 所示为 GitHub pull request。

### 运行自动测试

通过设置提交拦截脚本（commit hooks），可以对推送到版本控制系统的每个提交运行自动测试。最常见的方法是使用一个持续集成（*Continuous Integration, CI*）服务器，例如 Jenkins、CircleCI 或 TravisCI。大多数流行的 CI 服务器具有专门为 GitHub 内置的集成，不仅能够对每个提交自动运行测试，还能将测试的输出都显示在 pull request 中，图 8-2 所示为 GitHub pull request 中显示的 CircleCI 自动测试结果。

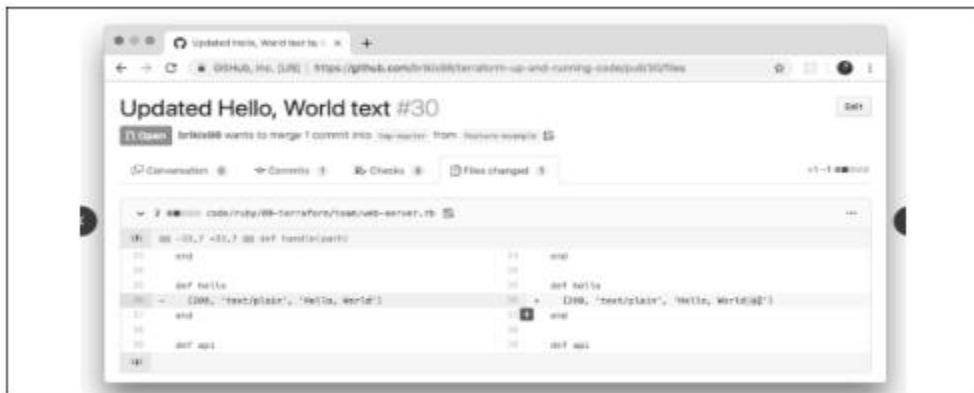


图 8-1：GitHub pull request



图 8-2：GitHub pull request 中显示的 CircleCi 自动测试结果

在图 8-2 中可以看到 CircleCI 已运行并通过了针对代码分支的单元测试、集成测试、端到端测试，以及一些静态分析检查（使用一个叫 snyk 的工具进行安全漏洞扫描）。

## 合并和发布

团队成员对代码的更改进行评审，寻找潜在的错误，加强编码准则（本章的稍后部分将进行详细介绍），检查已有的测试是否通过，并针对任何新行为添加测试。如果一切看起来都正常，代码可以合并到 master 分支中。

下一步是发布代码。如果使用不可变的基础设施实践（如第 1 章的“服务器模板工具”中所述），则发布应用程序代码意味着要将该代码打包到新的不可变的版本化工件。根据打包和部署应用程序的方式，工件可能是新的 Docker 映像、新的虚拟机映像（如新的 AMI）、新的.jar 文件、新的.tar 文件等。无论采用哪种格式，都要确保工件是不可变的（永远都不能对其进行更改），并且具有唯一的版本号（可以将该工件与其他工件区分开）。

例如，使用 Docker 打包应用程序，可以将版本号存储在 Docker 标签中。可以通过将提交的 ID（sha1 哈希）作为标签，使部署后的 Docker 映像能够追溯到其包含的工件的确切原代码。

```
$ commit_id=$(git rev-parse HEAD)
$ docker build -t briki98/ruby-web-server:$commit_id .
```

前面的代码将构建一个名为 briki98/ruby-web-server 的新的 Docker 映像，并使用最新提交的 ID 对其进行标记，该标签类似于 92e3c6380ba6d1e8c9134452ab6e26154e6ad849。稍后，如果需要调试 Docker 映像中的问题，可以通过 Docker 映像标签中的提交 ID，追溯到其中包含的确切代码。

```
$ git checkout 92e3c6380ba6d1e8c9134452ab6e26154e6ad849
HEAD is now at 92e3c63 Updated Hello, World text
```

提交 ID 的缺点是不容易阅读或记忆。另一种方法是创建一个 Git 标签。

```
$ git tag -a "v0.0.4" -m "Update Hello, World text"
$ git push --follow-tags
```

标签是指向特定 Git 提交的指针，但具有更友好的名称。可以在 Docker 映像上使用此 Git 标签。

```
$ git_tag=$(git describe --tags)
$ docker build -t brikis98/ruby-web-server:$git_tag .
```

在调试时，根据特定的标签，检出相应的代码。

```
$ git checkout v0.0.4
Note: checking out 'v0.0.4'.
(...)
HEAD is now at 92e3c63 Updated Hello, World text
```

## 部署

拥有版本控制下的工件后，就可以进行部署了。部署应用程序代码的方式有很多种，考虑因素包括应用类型、如何打包、如何运行、架构、使用的工具等。这里有几个主要的考虑因素。

- 部署工具
- 部署策略
- 部署服务器
- 跨环境推广工件

### 部署工具

部署应用程序可以使用不同的工具。如何选择，取决于如何打包及如何运行它。下面是一些示例。

#### *Terraform*

正如本书中所介绍的，Terraform 可以用于部署某些类型的应用程序。例如，在前面的章节中，创建的名为 `asg-rolling-deploy` 的模块，这个模块可以在整个 ASG 上进行零停机、滚动部署。如果将应用程序打包为 AMI（使用 Packer），则可以使用 `asg-rolling-deploy` 模块，部署新的 AMI 版本。具体做法是，更新 Terraform 代码中的 `ami` 参数，再次运行 `terraform apply` 命令。

#### *Docker* 编排工具

市场上有许多用于部署和管理 Docker 化应用程序的编排工具，包括 Kubernetes（最

受欢迎)、Apache Mesos、HashiCorp Nomad 和 Amazon ECS。如果将应用程序打包为 Docker 映像，可以通过运行 `kubectl apply` 命令，将新的 Docker 映像版本通过 Kubernetes 来部署 (`kubectl` 是 Kubernetes 命令行工具)。`kubectl` 通过读入 YAML 配置文件，确定要部署的 Docker 映像的名称和标签。

## 脚本

Terraform 和大多数编排工具一样仅支持有限的部署策略(将在下一节中讨论)。如果有更复杂的需求，你很可能需要使用通用编程语言编写自定义脚本(如 Python 和 Ruby)、配置管理工具(如 Ansible、Chef)或其他的服务器自动化工具(如 Capistrano)。编写这类脚本最棘手的事情是如何处理失败。例如，如果运行部署脚本的计算机在部署过程中失去 Internet 连接或崩溃，该怎么办？通过编写脚本来实现幂等性、实现故障恢复并正确完成部署不容易。这需要为脚本提供一种记录部署状态的方法(尽管有时可以通过查询基础设施来获取状态)，并构建一个可以处理所有可能的起始和转换状态的有限状态机。

## 部署策略

根据不同的需求，有许多不同的应用程序部署策略可以使用。假设有 5 个运行旧版本的应用程序的副本，你现在想发布新版本应用程序。以下是一些最常见的部署策略。

### 进行替换的滚动部署

删除一个旧版本的应用程序，部署新副本来自替换它，等待新副本启动并通过运行状况检查后，发送实时流量到新副本。重复这个过程，直到所有旧副本都被替换。滚动式替换部署确保同时运行的应用程序副本不超过 5 个，如果系统容量有限(应用程序的每个副本运行在物理服务器上)，或者面对的是一个有状态系统，每个应用程序都有唯一身份(例如，共识系统如 Apache ZooKeeper)，这种方式是十分有用的。需要注意，这个部署策略可以处理更大批次的部署(即一次替换多个应用程序副本，前提是未离线的应用程序副本可以处理负载且不会丢失数据)，另外部署期间，新、旧版本的应用程序将同时运行。

### 不进行替换的滚动部署

直接部署应用程序的一个新副本，等待新副本运行并通过状况检查后，开始发送实时流量到新副本，销毁应用程序的一个旧副本。然后重复这个过程，直到所有旧版本副本被替换。不进行替换的滚动部署的前提是系统拥有灵活的容量（例如，在应用程序所运行的云中，可以随时启动新的虚拟服务器），并且应用程序能接受同时运行超过 5 个以上的副本。这样做的好处是，同时运行的应用程序副本永远不会少于 5 个，因此在部署过程中，运行的容量不会减少。请注意，此部署策略可以处理更大批次的部署（可以同时部署 5 个新副本，这正是之前使用 `asg-rolling-deploy` 模块所做的工作）。在部署期间，新、旧版本的应用程序将同时运行。

### 蓝绿部署

部署应用程序的 5 个新副本，等待它们全部运行并通过状况检查后，将所有流量切换到新副本，然后销毁旧副本。蓝绿部署要求系统具有灵活的容量（例如，在应用程序所运行的云中，可以随时启动新的虚拟服务器），并且应用程序能接受同时运行 5 个以上的副本。优点是在任何时间，用户都只看到一个版本的应用程序。运行的应用程序副本永远不会少于 5 个，因此在部署期间，运行容量不会减少。

### 金丝雀部署

部署该应用程序的一个新副本，等待它启动并通过状况检查，开始向其发送实时流量，然后暂停部署。在暂停期间，将应用程序的新副本（被称为 `canary`）与旧副本之一（被称为 `control`）进行各种维度上的比较：CPU 使用率、内存使用率、延迟、吞吐量、日志中的错误率、HTTP 响应代码等。理想情况下，如果两个服务器行为上没有明显区别，可以确信新代码正常工作。在这种情况下，恢复部署进度，继续使用任意一种滚动部署策略来完成其余部分。另一方面，如果发现任何差异，则是新代码中出现问题的迹象，在问题变得更糟之前取消部署并销毁金丝雀部署实例。

该名称取自“煤矿中的金丝雀”。矿工会带着金丝雀进入隧道，如果隧道中充满危险气体（如一氧化碳），这些气体会在矿工遇到危险之前先杀死金丝雀，从而向矿工发出预警，在进一步造成伤害之前，他们必须立即离开。金丝雀部署提供了类似的好处，可以在生产环

境中，系统地测试新代码，如果出现问题，则较早地发出警告。它只会影响一小部分用户，并且你有足够的时间做出反应来阻止损失的扩大。

金丝雀部署通常与功能开关结合在一起使用。将所有新功能包括在 if 表达式中，if 表达式默认为 false。因此在最初部署代码之后，新功能处于关闭状态。因为所有的新功能是关闭的，所以部署的金丝雀服务器和控件服务器拥有同样的表现，这个时候任何差别都可以自动标记为问题并触发回滚。如果没有问题，你可以通过内部网站管理界面将一小部分用户切换到新功能上。例如，可以最初只为内部员工启用新功能；如果效果良好，则继续为 1% 的外部用户启用新功能；如果仍然有效，则可以将比例提高到 10%；在任何时候出现问题，都可以通过关闭功能开关，降低影响范围。这个过程可以将部署新代码与发布新功能分开处理。

#### 部署服务器

部署任务的运行应该基于 CI 服务器而不是开发人员的计算机，好处如下。

#### 全自动

从 CI 服务器运行部署，将强制自动化所有部署步骤。可以确保将整个部署过程捕获为代码，不会因手动错误而遗漏任何步骤，并且可以确保快速部署且可以重复。

#### 在一致的环境中运行

如果开发人员从自己的计算机运行部署，由于计算机配置的差异，例如，不同的操作系统、不同的依赖版本（不同版本的 Terraform）、不同的配置，以及不同的部署内容（开发人员意外地部署了版本控制系统之外、未提交的版本），你会遇到各种错误。通过从相同的 CI 服务器上运行部署可以消除所有这些问题。

#### 更好的权限管理

不应该将部署权限授予每个开发人员，而应该将其仅授予 CI 服务器（尤其在生产环境下）。相对于数十名或数百名开发人员具有生产访问权限而言，只对一台服务器实施良好的安全性管理要容易得多。

### 跨环境推广工件

在不可变的基础设施的最佳实践中，推出新变更的方法是：将版本完全相同的工件从一个环境部署到另一个环境。例如，如果现在有开发、预发布和生产环境，为了推出 v0.0.4 版本应用程序，则可以执行以下操作。

1. 部署应用程序的 v0.0.4 版本到开发环境。
2. 在开发环境中运行手动和自动测试。
3. 如果 v0.0.4 版本在开发环境中运行良好，重复步骤 1 和 2，将 v0.0.4 版本部署到预发布环境（这被称为工件推广）。
4. 如果 v0.0.4 版本在预发布环境中运行良好，再次重复步骤 1 和 2，将 v0.0.4 版本部署到生产环境。

因为不同环境中运行的是完全相同的工件，所以如果在一个环境中工作正常，工件将有很大概率在另一个环境中也可以正常工作。如果的确遇到问题，也可以通过部署旧版本的工件版本来随时恢复。

## 部署基础设施代码的工作流程

我们已经学习了如何部署应用程序代码，现在让我们深入分析部署基础设施代码的工作流程。在本节中，当我提到“基础设施代码”时，泛指使用任何 IaC 工具（包括 Terraform）编写的代码，用于部署除了单个应用程序之外的任意基础设施更改。例如部署数据库、负载均衡器、网络配置、DNS 设置等。

下面介绍一下部署基础设施代码的工作流程。

1. 使用版本控制。
2. 在本地运行代码。
3. 进行代码更改。

4. 提交更改以供评审。

5. 运行自动测试。

6. 合并和发布。

7. 部署。

从表面上看，它与部署应用程序的工作流程相同，但在本质上却存在很大的差异。部署基础设施的代码更为复杂，并且使用的技术并不十分容易理解，为了让我们更容易学习，首先和应用程序代码部署工作流程进行一下类比。

## 使用版本控制

与应用程序代码一样，所有基础设施代码都应该在版本控制之下。这意味着用户将使用 `git clone` 命令像以前一样检出代码。但是，基础设施代码的版本控制还有一些其他要求。

- 实时代码库和模块代码库
- Terraform 的黄金法则
- 分支的麻烦

### 实时代码库和模块代码库

如第 4 章所述，我们通常需要至少两个单独的版本控制存储库来存储 Terraform 代码：一个用于存储模块代码，一个用于存储实时基础设施代码。模块存储库用来保存已创建的、可重用的、版本控制的模块代码，例如，在本书前面各章中构建的所有模块(`cluster/asg-rolling-deploy`、`data-stores/mysql`、`networking/alb` 和 `services/hello-world-app`)。实时基础设施存储库存储了每种环境（Dev、Stage、Prod 等）中部署的实际基础设施。

一种行之有效的模式是在公司中设立一个基础设施团队，专门研究创建可重用的、健壮的生产级模块。团队通过构建具有组合 API 的模块库（第 6 章提到过），在公司中创造非凡的影响力。也就是说，模块拥有完整的文档记录（包括 `examples` 文件夹中的可执行文档）及整套的自动测试套件，已经版本化，并满足生产级基础设施检查列表中的所有要求（安全性、合规性、可伸缩性、高可用性、监控等）。

如果建立了这样的模块库(或购买了现成的库<sup>1</sup>),公司的所有团队将能够使用这些模块(有点像服务目录)来部署和管理自己的基础设施,而不需要每个团队花费数月的时间,从无到有组装基础设施,或让运维团队成为瓶颈,因为它需要部署和管理每一个团队的基础设施。取而代之的是,运维团队可以将大部分时间花费在编写基础设施模块的代码上,所有其他团队都独立工作,使用这些模块来完成他们的工作。而且,由于每个团队使用具有相同规范的模块,随着公司的发展和需求的变化,运维团队可以向所有团队推出模块的新版本,以确保一致性和可维护性。

或者,只要遵循 Terraform 的黄金法则,它就能够被维护。

### Terraform 的黄金法则

这里有一种检查 Terraform 代码运行状况的快速方法:进入实时存储库,随机选择几个文件夹,然后在每个文件夹中运行 `terraform plan` 命令。如果输出始终为“无变化”,那就太好了,因为这意味着基础设施代码与实际部署的环境相匹配。如果输出显示出很小的差异,并且偶尔会听到团队成员的借口(“哦,对了,我手动调整了一个部署,却忘记更新代码了”),则代码与实际情况不符,并且你可能很快就会遇到麻烦。如果 `terraform plan` 完全失败并出现奇怪的错误,或者每个 `plan` 命令都显示出巨大差异,那么你的 Terraform 代码已经和现实环境完全没有关系,并且可能已经毫无用处。

最佳标准或你真正想要达到的目标,我统称之为 *Terraform* 的黄金法则。

实时存储库的主代码分支应该以 1:1 的形式完全代表生产环境中实际部署的内容。

让我们把这句话分解开来,从结尾开始,然后再往前看。

#### “……实际部署的内容”

确保实时存储库中 Terraform 的代码能够代表最新目标环境的唯一方法是,永远不要进行工具之外的更改。开始使用 Terraform 后,请勿通过 Web UI、手动 API 调用或任何其他机制进行更改。正如第 5 章学习的,工具之外的更改不仅会导致复杂的错误,

---

<sup>1</sup> Gruntwork 基础设施代码库是具有 30 多万行生产级的、商业化支持的、可重复使用的基础设施代码,已经在数百家企业的生产环境中被成功应用和证明见参考资料第 8 章[4]。

而且还会抵消许多使用 IaC 已经带来的优点。

#### “……1:1 形式代表……”

当浏览实时存储库时，通过快速扫描代码，应该可以看出在哪些环境中部署了哪些资源。换句话说，每个资源都应该能找到 1:1 匹配的，签入实时仓库中的代码行。看起来似乎很浅显的道理却很容易出差错。正如我刚才提到的，一种造成错误的方法是进行工具外的更改，这会导致虽然代码存在，但实时基础设施却是不同的。一种更微妙的错误是由于使用 Terraform 工作区来管理环境导致的，虽然部署了实时基础设施，但是代码却没有被保存。也就是说，如果使用 Terraform 工作区部署了 3 个或 30 个环境，但实时代码库中也可能只有一个代码副本。仅通过浏览代码，是无法知道实际部署了什么资源的，这将导致错误并使维护变得更加复杂。因此，如第 3 章的“通过工作区进行隔离”中所述，尽量避免使用工作区来管理环境，而要针对每个环境使用单独的文件和文件夹进行定义，以达到通过浏览实时代码库就可以准确地了解部署环境的目的。在本章的后面，我们将学习如何以少量的复制/粘贴来做到这一点。

#### “主分支……”

你只需要查看一个分支就可以了解生产环境中实际部署的内容。通常这个分支将是 `master`。这意味着，影响生产环境的所有变化应该直接进入 `master` 分支（你可以创建单独分支进行开发，但最终一定要通过 `pull request` 合并到 `master` 分支）。针对生产环境的部署，应该在 `master` 分支上运行 `terraform apply` 命令。在下一节中，我将解释原因。

#### 分支的麻烦

在第 3 章中，我们学习了可以使用 Terraform 后端中内置的锁定机制，确保即使两个团队成员同时对同一组 Terraform 进行配置，运行 `terraform apply` 命令，也不会覆盖彼此的更改。不幸的是，这只能解决部分问题。Terraform 后端锁定机制可以锁定 Terraform 状态文件，但无法锁定 Terraform 代码本身。如果两个团队成员将相同的代码从不同的分支部署到相同的环境中，那么将陷入锁定机制也无法避免的冲突。

例如，假设团队成员之一 Anna，对名为 foo 的应用程序的 Terraform 配置进行了一些更改，该配置文件包括一个 Amazon EC2 实例。

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

因为该应用程序存在很大的访问流量，所以 Anna 决定将 `instance_type` 从 `t2.micro` 改变为 `t2.medium`。

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.medium"
}
```

当 Anna 运行 `terraform plan` 命令时，能够看到以下输出。

```
$ terraform plan

(...)

Terraform will perform the following actions:

# aws_instance.foo 将被更新
- resource "aws_instance" "foo" {
    ami = "ami-0c55b159cbfafe1f0"
    id = "i-096430d595c80cb53"
    instance_state = "running"
    - instance_type = "t2.micro" -> "t2.medium"
    ...
}
Plan: 0 to add, 1 to change, 0 to destroy.
```

因为更改看起来没有错误，因此 Anna 将它部署到预发布环境。

这时 Bill 加入了开发流程，并且开始在不同分支上对同一应用程序的 Terraform 配置进行更改。Bill 想做的就是在应用程序中添加标签。

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
```

```
tags = {
    Name = "foo"
}
}
```

注意，此时 Anna 的更改已在预发布环境中部署，但是由于它们位于不同的分支上，因此 Bill 的代码中 `instance_type` 仍然被设置为 `t2.micro`（旧值）。下面是当他运行 `plan` 命令时所看到的输出（以下日志输出被截断以提高可读性）。

```
$ terraform plan

(...)

Terraform will perform the following actions:

# aws_instance.foo 将被更新
~ resource "aws_instance" "foo" {
    ami = "ami-0c55b159cbfafe1f8"
    id = "i-096430d595c80cb53"
    instance_state = "running"
    ~ instance_type = "t2.medium" -> "t2.micro"
    + tags = [
        + "Name" = "foo"
    ]
    ...
}

Plan: 0 to add, 1 to change, 0 to destroy.
```

糟糕，他将要撤销 Anna 对 `instance_type` 的更改！如果 Anna 仍在预发布环境中进行测试，当服务器突然重新部署后以不一样的状态运行时，她会感到非常困惑。好消息是，如果 Bill 认真阅读 `plan` 命令的输出，他会发现并避免这个可能会影响 Anna 的错误。不管怎样，这个示例还是很好地说明了当从不同的分支向共享环境进行部署时，会出现什么样的问题。

Terraform 后端的锁定机制在这里无能为力，因为这里的冲突与状态文件的并发修改无关。Bill 和 Anna 即使相隔数周部署各自的更改，问题还是相同的。造成这个问题的原因是，分支和 Terraform 不是好的组合。Terraform 的一个隐含属性是，Terraform 代码和真实世

界中部署的基础设施之间存在着 1:1 的映射。因为真实世界只有一个，所以多个分支的 Terraform 代码并没有太大的意义。因此，对于任何共享环境（如 Stage、Prod），请始终从单个分支进行部署。

## 在本地运行代码

现在，代码已经被检出到计算机上，下一步就是运行它。与应用程序代码不同，Terraform 没有“本地主机”的概念。例如，无法将 AWS 的 ASG 部署到笔记本电脑上。如第 7 章的“手动测试基础”中所述，手动测试 Terraform 代码的唯一方法是在沙箱环境中运行它，例如专用于开发的 AWS 账户（或者更好的方法是，每个开发人员拥有一个 AWS 账户）。

拥有沙箱环境后，请运行 `terraform apply` 命令，启动手动测试。

```
$ terraform apply  
(...)  
  
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.  
Outputs:  
alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

然后，使用 `curl` 等工具验证已部署的基础设施是否正常运行。

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com  
Hello, World
```

如果运行用 Go 语言编写的自动测试，需要在测试环境专用的沙箱账户下执行 `go test`。

```
$ go test -v -timeout 30m  
(...)  
  
PASS  
ok terraform-up-and-running 229.492s
```

## 进行代码更改

既然已经可以运行 Terraform 代码，接下来就可以开始像应用程序代码一样进行迭代更改了。每次更改时，可以重新运行 `terraform apply` 命令来部署这些更改，并重新运行 `curl`

命令查看更改是否有效。

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com  
Hello, World v2
```

重新运行 `go test` 命令以确保自动测试用例仍然可以通过。

```
$ go test -v -timeout 30m  
  
(...)  
  
PASS  
ok terraform-up-and-running 229.492s
```

从上面的执行结果可以看出，与应用程序代码的测试的区别是，基础设施代码测试通常需要更长的执行时间。因此需要更多地考虑如何缩短测试周期，以便尽快获得关于更改的反馈。在第 7 章“集成测试”的“测试阶段”中，我们学到通过重新运行测试套件的特定阶段，来大大缩短反馈周期。

例如，一个测试包括以下步骤：部署数据库、部署应用程序、验证两者正常工作、销毁应用程序和销毁数据库。则在最初运行测试时，可以跳过两个销毁部署步骤，保持应用程序和数据库持续运行。

```
$ SKIP_teardown_db=true \  
SKIP_teardown_app=true \  
go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'  
  
(...)  
  
PASS  
ok terraform-up-and-running 423.650s
```

然后，在每次对应用程序进行更改时，都跳过数据库部署和两个销毁步骤。只有应用程序部署和验证步骤被重复执行。

```
$ SKIP_teardown_db=true \  
SKIP_teardown_app=true \  
SKIP_deploy_db=true \  
go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'  
  
(...)
```

```
PASS
ok terraform-up-and-running 13.824s
```

通过将反馈周期从几分钟缩短到几秒钟，极大地提高了开发人员的生产力。

另外进行更改时，请确保定期提交代码。

```
$ git commit -m "Updated Hello,World text"
```

## 提交更改以供评审

代码以期望的方式工作后，可以通过创建 pull request 请求来发起代码评审。就像处理应用程序代码一样。团队将评审代码的更改、查找错误并强制执行编码准则。当以团队形式编写代码时，无论编写的是哪种类型的代码，都应该定义所有人共同遵循的编码准则。什么是“干净代码”？我最推崇的定义出自自我较早的著作 *Hello Startup*（见参考资料第 8 章[5]）中对 Nick Dellamaggiore 的采访。

当我看到一个由 10 位不同的工程师共同编写的文件，已经无法区分哪一部分是由哪个人撰写的。对我来说，这就是干净的代码。

你可以通过代码评审、发布风格指南、编码样式和习惯用语来实现这一点。一旦团队掌握了这些技巧，以相同的方式来编写代码，生产效率就会大大提高。到了那个时候，你的工作重点将变为决定写什么而不是如何写。

—Nick Dellamaggiore，Coursera 基础设施负责人

每个团队都会定义稍微不同的 Terraform 编码指南，在这里，我只列出对大多数团队有用的常见准则。

- 文档
- 自动测试
- 文件布局
- 样式指南

## 文档

从某种意义上来说，Terraform 代码本身就是文档的一种形式。它用一种简单的语言，准确描述了部署哪些基础设施，以及如何配置这些基础设施。但是，没有哪种代码会自己编

写文档。虽然精心编写的代码可以告诉你它做了什么，但据我所知，没有哪种编程语言（包括 Terraform）可以告诉你为什么这样做。

这就是为什么包括 IaC 在内的所有软件，都需要代码本身之外的文档。可以考虑以下几种类型的文档，并且要求团队成员将其作为代码评审的一部分。

#### 书面文档

大多数 Terraform 模块都应具有一个 README 自述文件，用来自述模块的功能、存在原因、使用方式，以及如何修改它。实际上，最好先编写 README 自述文件，再编写实际的 Terraform 代码。因为这样做会让你有限考虑：需要构建的内容，以及为什么要构建它，避免过早地迷失在如何编写代码的细节上。<sup>1</sup>首先花 20min 编写自述文件，可以避免花费数小时的编码时间去解决错误的问题。除基本的自述文件之外，还可以涵盖一些教程、API 文档、Wiki 页面和设计文档，更深入地阐述代码的工作方式，以及为什么需要采用某种特定的代码构建方法。

#### 代码文档

在代码主体内，以注释的方式编写文档。Terraform 将任何以#开头的文本视为注释。不要试图通过注释来解释代码的作用，代码本身应该做到自我解释。仅通过注释提供代码无法表达的信息。例如，代码应如何被使用或代码为何选择了特定的设计。Terraform 还允许为每个输入和输出变量声明一个 `description` 参数，描述应该如何使用变量。

#### 示例代码

如第 6 章中所述，每个 Terraform 模块都应包括示例代码，以说明如何使用该模块。这是展示如何正确使用代码的好方法。为用户提供一种方式：无须编写任何代码就可以试用模块。一个主要的用例是为模块添加自动测试。

#### 自动测试

所有第 7 章的内容都集中在如何测试 Terraform 代码上。因此，在这里我不会重复相关的

---

<sup>1</sup> 首先编写 README 自述文件的开发方式称为自述驱动开发（见参考资料第 8 章[6]）。

任何内容，只想强调，没有测试的基础设施代码是已经损坏的。因此，在任何代码评审中，提出的重要意见之一是：“你打算如何进行测试？”

### 文件布局

团队应该约定 Terraform 代码的存储位置，并定义如何使用文件布局。因为 Terraform 的文件布局会改变 Terraform 状态文件的存储方式，所以需要特别注意文件布局对隔离能力的影响，例如，保证预发布环境中的更改不会意外影响生产环境。在代码评审中，可以考虑强制执行第 3 章“通过文件布局进行隔离”所介绍的文件布局，对不同环境（如 Stage、Prod）与不同组件（如针对整个环境的网络拓扑和环境中的单个应用）提供充分的隔离。

### 样式指南

每个团队都应该制定关于代码样式的约定。包括如何使用空白、换行符、缩进、花括号、变量命名等。尽管程序员喜欢辩论空格与制表符的区别，以及花括号应放在何处，但实际选择并不那么重要。真正重要的是在整个代码库中保持风格一致。大多数文本编辑器和集成开发环境（IDE）都内置了格式化工具。版本控制系统的提交拦截脚本可以强制检查通用的代码布局。

Terraform 甚至有一个内置的 `fmt` 命令，可以自动地重新格式化代码风格。

```
$ terraform fmt
```

你可以将这个命令作为提交拦截脚本的一部分来运行，确保所有进入版本控制系统的代码拥有一致的风格。

### 运行自动测试

与应用程序代码一样，基础设施代码应通过提交拦截脚本，在每次提交后触发 CI 服务器中的自动测试流程，并在 pull request 请求中显示测试的结果。我们已经在第 7 章中学习了如何为 Terraform 代码编写单元测试、集成测试，以及端到端测试。另一种重要的测试是：`terraform plan` 命令。规则很简单。

始终在运行 `apply` 命令之前，运行 `plan` 命令。

Terraform 也会在运行 `apply` 命令时，自动显示 `plan` 命令的输出。因此这个规则的真正含义是，用户应该始终在这里暂停，认真阅读 `plan` 命令的输出！花费 30s 扫描命令输出的“diff”内容，你会很惊讶地发现各种类型的错误。鼓励这种做法的一个好方法是，将 `plan` 命令集成到代码评审流程中。例如，Atlantis（见参考资料第 8 章[7]）是一个开源工具，它会在提交时自动运行 `terraform plan` 命令，并将 `plan` 命令的输出添加到 pull request 的注释中，如图 8-3 所示。



图 8-3：Atlantis 可以自动将 `terraform plan` 的输出添加到 pull request 的注释中

`plan` 命令甚至可以将比较差异输出并存储到一个文件中。

```
$ terraform plan -out=example.plan
```

在之后的部署中，对保存的计划输出文件运行 `apply` 命令，确保部署的内容完全与 `plan` 命令的输出保持一致。

```
$ terraform apply example.plan
```

请注意，与 Terraform 状态文件一样，保存的计划输出文件中可能包含机密信息。例如，如果使用 Terraform 部署数据库，则计划输出文件可能会包含数据库密码。由于计划输出文件未加密，因此，如果要较长时间存储它们，则需要自己进行文件加密。

## 合并和发布

当代码更改和计划输出通过团队成员的评审，并完成所有测试之后。你可以将更改合并到 master 分支中并发布代码。与应用程序代码流程类似，可以使用 Git 标记创建发布的版本。

```
$ git tag -a "v0.0.6" -m "Updated hello-world-example text"  
$ git push --follow-tags
```

在应用程序代码流程中，用户通常需要单独部署一个工件，例如 Docker 映像或虚拟机映像。因为 Terraform 支持从 Git 存储库直接下载代码，所以特定标签的存储库状态就是将要部署的、不可变的、版本控制的工件。

## 部署

你已经拥有了一个不可变的、版本控制下的工件，现在该进行部署了。以下是部署 Terraform 代码的一些关键注意事项。

- 部署工具
- 部署策略
- 部署服务器
- 跨环境推广工件

### 部署工具

部署 Terraform 代码时，Terraform 本身就是需要使用的主要工具。这里也有一些其他有用的工具。

### *Atlantis*

这个开源工具不仅可以将 `plan` 命令的输出添加到 `pull request` 请求中，也可以通过在 `pull request` 请求中添加特殊的评论来触发 `terraform apply` 命令。虽然工具为 Terraform 部署提供了方便的网站界面，但请注意，它不支持版本控制，这个缺点可

能会使对大型项目的维护和调试变得困难。

### *Terraform Enterprise*

HashiCorp 的企业产品提供了一个 Web UI，可用于运行 `terraform plan` 和 `terraform apply` 命令，以及管理变量、机密和访问权限。

### *Terragrunt*

这是一个开源的、基于 Terraform 的外壳工具，它填补了 Terraform 功能上的一些空白。本章稍后将会介绍，如何通过最少的复制/粘贴，在多个环境中部署版本化的 Terraform 代码。

## 脚本

可以通过通用编程语言如 Python、Ruby 或 Bash，来定制使用 Terraform。

## 部署策略

Terraform 本身不提供任何部署策略。没有对滚动部署或蓝绿部署的内置支持，也没有办法对大多数 Terraform 更改设置功能开关（例如，无法通过功能开关控制对数据库的更改）。本质上，局限性来自 `terraform apply` 命令，它的运行严格遵守代码中的具体配置。当然，通过对代码本身的管理，可以实现一些部署策略。例如，前几章中构建的 `asg-rolling-deploy` 模块，实现了零停机、滚动部署。但是由于 Terraform 是一种声明性语言，因此部署中的控制相当有限。

由于这些限制，考虑部署出错时会发生什么状况是至关重要的。在应用程序部署中，部署策略会捕获许多类型的错误。例如，如果应用程序无法通过运行状况检查，负载平衡器将永远不会向其发送实时流量，因此用户不会受到影响。此外，如果出现错误，滚动部署或蓝绿部署策略可以自动恢复到应用程序之前的版本。

另一方面，Terraform 出现错误时，不会自动恢复到之前的状态。部分原因是，使用任意的基础设施代码是不安全的或不可能的。例如，如果应用程序部署失败，恢复到应用程序的较旧版本几乎总是安全的。但是如果正在部署的 Terraform 更改失败，如果更改是删除

数据库或终止服务器，这一类的恢复过程是十分困难的！

而且，正如在第 5 章的“Terraform 陷阱”中看到的，Terraform 在部署中遇到错误是相当普遍的。因此部署策略也应该承认，出现错误是（相对）正常的，要最高优先级地处理这些错误。

### 重试

某些 Terraform 的错误类型是短暂的，如果重新运行 `terraform apply` 命令，错误将会消失。与 Terraform 一起使用的部署工具应该可以检测到这些已知错误，并在暂停后自动重试。Terragrunt 通过内置功能会对已知错误进行自动重试。

### Terraform 状态错误

有时，在运行 `terraform apply` 命令之后，Terraform 将无法保存状态。例如，如果在运行 `apply` 命令期间失去互联网连接，不仅 `apply` 命令注定会失败，Terraform 也无法将更新的状态文件写入远程后端（如 Amazon S3）。在这种情况下，Terraform 会将状态文件保存在磁盘上名为 `errored.tfstate` 的文件中。请确保 CI 服务器不会删除这些文件（例如，在构建后销毁工作空间时）！如果在失败的部署后仍然可以访问此文件，一旦 Internet 连接恢复，可以通过 `state push` 命令，将该文件推送到远程后端（如 Amazon S3），以免出现状态信息丢失。

```
$ terraform state push errored.tfstate
```

### 释放锁时出错

有时，Terraform 将出现无法释放锁的情况。例如，如果 CI 服务器在运行 `terraform apply` 命令期间崩溃，状态文件将永久处于锁定状态。当其他人尝试在相同的模块运行 `apply` 命令时，会得到状态被锁定的错误消息及锁的 ID。如果用户绝对确信这是意外遗留的锁定，可以使用 `force-unlock` 命令强制解除它。请将之前错误消息中的锁的 ID 作为参数传递给命令。

```
$ terraform force-unlock <LOCK_ID>
```

## 部署服务器

与应用程序代码一样，所有基础设施代码的更改都应该通过 CI 服务器，而不是开发人员的计算机来部署。可以使用 Jenkins、CircleCI、Terraform Enterprise、Atlantis 或其他任何安全的自动化平台来运行 `terraform` 命令。这为用户提供了与应用程序代码部署流程相同的好处：强制实现部署过程完全自动化，确保部署始终在一致的环境中进行，并且可以更好地控制生产环境的访问权限。

管理部署基础设施代码的权限，比管理应用程序代码要复杂得多。为了使用应用程序代码，通常可以为 CI 服务器分配最小的固定权限以部署应用程序。例如，要部署到 ASG，CI 服务器通常只需要几个特定的 `ec2` 和 `autoscaling` 的访问权限。但是，为了能够部署任意基础设施代码的更改（例如，Terraform 代码，可能会尝试部署数据库或 VPC 或全新的 AWS 账户），CI 服务器需要管理员权限。

如果 CI 服务器被设计为可以执行任意代码，那么通常很难保护其安全性（加入 Jenkins 安全列表后就能体会到，关于严重攻击漏洞的公告宣布是多么频繁）。因此为 CI 服务器提供永久管理员权限是有风险的。我们可以采取一些措施来最大程度地降低这种风险。

- 不要将 CI 服务器暴露在公共互联网上。也就是说，运行 CI 服务器的专用子网，不应有任何的公网 IP，而只能通过 VPN 访问。这明显会更安全，但是也要付出代价：来自外部系统的 Web hook 将无法工作。例如，GitHub 将无法自动触发你的 CI 服务器中的构建。相反，需要配置 CI 服务器定时查询版本控制系统来获取更新。
- 锁定 CI 服务器。使它只能通过 HTTPS 被访问，要求所有用户都经过身份验证，并遵循服务器强化实践（例如，锁定防火墙、安装 fail2ban、启用审核日志等）。
- 考虑不要在 CI 服务器上分配永久的管理员权限。只给它分配足够的权限，以便可以完成某些类型的基础设施代码更改的自动部署，但是对于更敏感的内容（例如，添加或删除用户或访问机密），要求人工管理员为服务器提供临时管理员密码（例如，1h 后就会过期的凭证）。

## 跨环境推广工件

与应用程序工件一样，用户也需要将不可变的版本化基础设施工件从一个环境推广到另一个环境。例如，将 v0.0.6 版本从开发环境推广到预发布环境再到生产环境。<sup>1</sup>这里的规则也很简单：

始终在生产环境部署之前，通过预发布环境测试 Terraform 的更改。

因为一切部署和测试都通过 Terraform 自动化完成，无须花费太多精力就可以在生产环境部署之前，进行预发布环境中的更改测试，这样做能够捕获大量错误。在预发布环境中进行测试尤为重要，如本章前面所述，因为 Terraform 不会在发生错误的情况下恢复到之前的状态。如果运行 `terraform apply` 命令且出了问题，用户必须自己手动修复错误。所以，在预发布环境中捕获并修复错误，会大大降低生产环境部署中的压力。

跨环境推广 Terraform 代码与跨环境推广应用程序构件的过程极其相似，除了一个额外的步骤：手动执行 `terraform plan` 命令，并检查输出内容。对于大多数应用程序部署来说，这个步骤通常不是必需的，因为大多数应用程序部署都很相似，并且风险相对较低。但是，每个基础设施的部署都可能完全不同，并且错误的代价可能很高（如删除数据库）。因此，请利用部署之前最后的机会，对 `plan` 命令的输出进行评审，这是非常值得做的事情。

例如以下过程，将 Terraform 模块版本的 v0.0.6 版本从开发环境推广到预发布环境，最终到生产环境。

1. 将开发环境代码更新为 v0.0.6 版本，并运行 `terraform plan` 命令。
2. 邀请相关人员评审并批准计划。例如，通过 Slack 自动发送消息。
3. 如果计划获得批准，使用 `terraform apply` 命令部署 v0.0.6 版本到开发环境。
4. 在开发环境中运行手动测试和自动测试。

---

<sup>1</sup> 关于如何在跨环境中推广 Terraform 代码这一部分归功于 Kief Morris 的作品《使用管道与基础设施即代码来管理环境》（见参考资料第 8 章[8]）。

5. 如果 v0.0.6 版本在开发环境中运行良好，重复步骤 1~4，将 v0.0.6 版本推广到预发布环境。
6. 如果 v0.0.6 版本在预发布环境中运行良好，再次重复步骤 1~4，将 v0.0.6 版本推广到生产环境。

一个重要问题是实时代码库中如何处理如何处理实时代码库中各个环境之间的重复代码。例如，如图 8-4 所示的实时代码库文件布局中存在大量针对环境和模块定义的复制/粘贴。

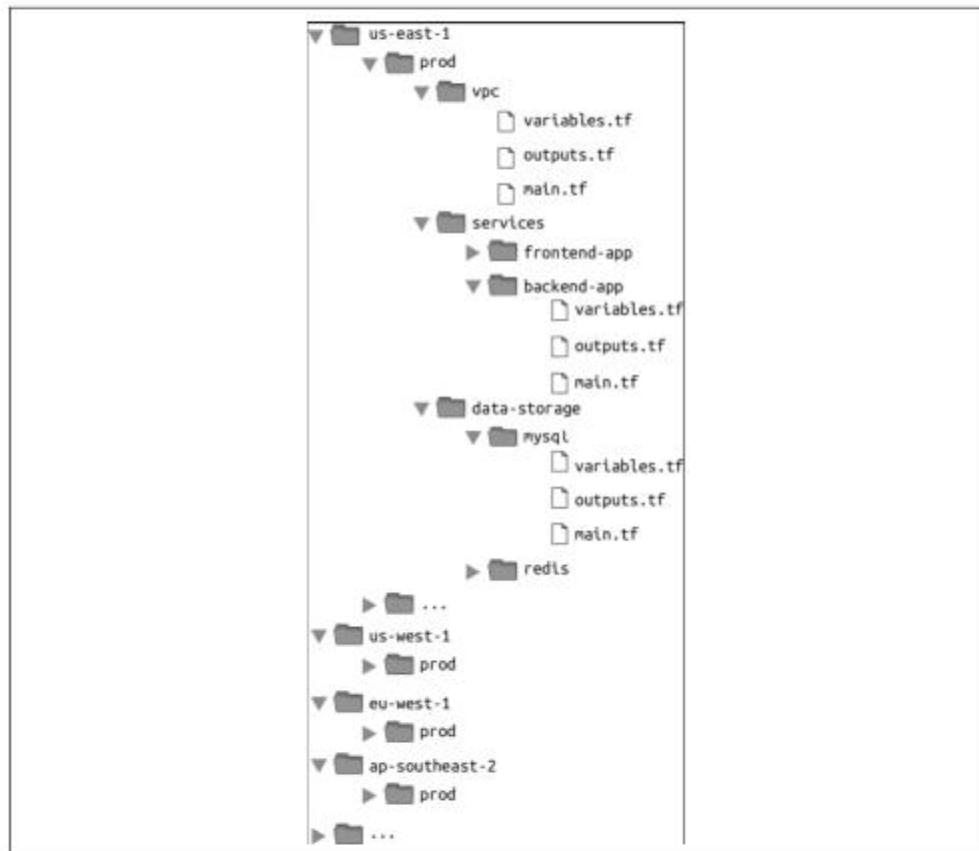


图 8-4：实时代码库文件布局中存在大量针对环境和模块定义的复制/粘贴

此实时代码库包含很多区域，并且在每个区域内都有大量模块，其中大部分都是复制/粘贴相同的内容。每个模块都有一个 *main.tf* 文件，用来引用一个模块代码库中的模块。虽然已经省去了很多复制/粘贴，即使只是实例化一个单独的模块，还是有大量的样板需要在每个环境之间进行复制，包括：

- 提供商配置
- 后端配置
- 设置所有该模块的输入变量
- 从模块输出所有的输出变量

每个模块中，多达数十行或数百行几乎相同的代码，要被复制/粘贴到每个环境。为了使代码更加简洁，并使环境之间推广 Terraform 代码更容易，可以使用前面提到的名为 Terragrunt（见参考资料第 8 章[9]）的开源工具。Terragrunt 是 Terraform 的外壳程序，这意味着安装后（请参阅 Terragrunt README 中的安装说明见参考资料第 8 章[10]），Terragrunt 可以运行所有标准 Terraform 命令。

```
$ terragrunt plan  
$ terragrunt apply  
$ terragrunt output
```

Terragrunt 将使用指定的命令去调用 Terraform，会在基于 *terragrunt.hcl* 文件的配置上，增加一些额外的行为。其基本思想是，*modules* 存储库中定义所有相同的 Terraform 代码，而在实时存储库中，通过 *terragrunt.hcl* 文件，提供一种简洁方式来配置和部署每个环境中的各个模块。如图 8-5 所示为使用 Terragrunt 后的文件布局，这将大量降低实时存储库中的文件和代码行数。

首先，在 *modules/data-stores/mysql/main.tf* 和 *modules/services/hello-world-app/main.tf* 文件中，添加 *provider* 配置。

```
provider "aws" {  
    region = "us-east-2"  
  
    # 允许使用 AWS 提供商的任何 2.x 版本  
    version = "~> 2.0"  
}
```

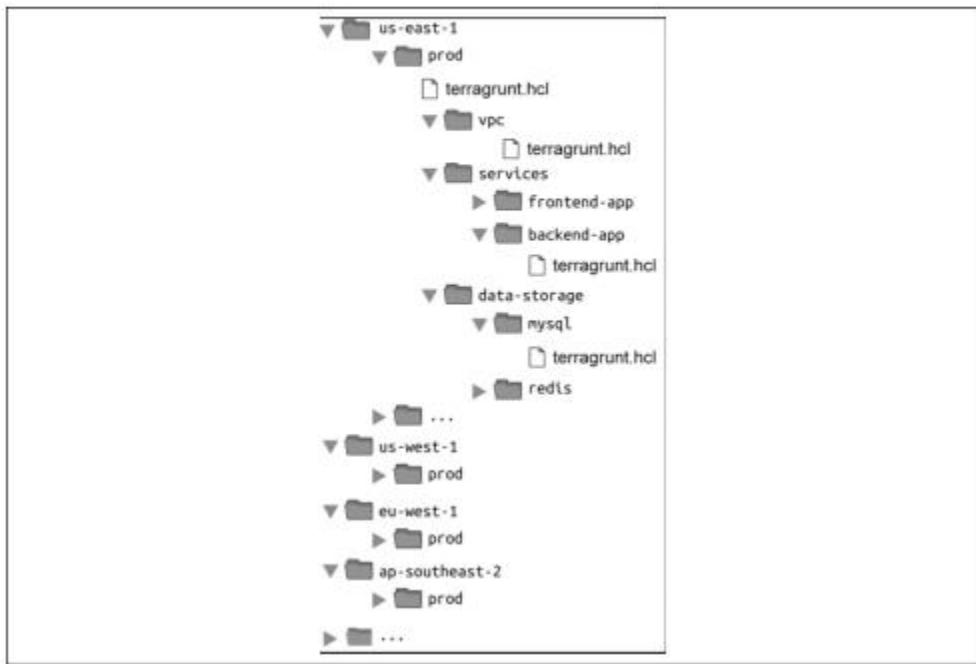


图 8-5：使用 Terragrunt 后的文件布局

接下来，在 `modules/data-stores/mysql/main.tf` 和 `modules/services/hello-world-app/main.tf` 文件中，添加 `backend` 配置，但保持 `config` 块为空（马上会看到如何使用 Terragrunt 填补这个空白块）。

```

terraform {
  # 需要 Terraform 的任意 0.12.x 版本
  required_version = ">= 0.12, < 0.13"

  # 部分配置，其余的将由 Terragrunt 填写
  backend "s3" {}
}
  
```

提交这些更改并发布模块的新版本。

```

$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Update mysql and hello-world-app for Terragrunt"
  
```

```
$ git tag -a "v0.0.7" -m "Update Hello, World text"  
$ git push --follow-tags
```

现在，转到实时存储库，并删除所有以`.tf` 为后缀的文件。用户需要为每个模块，创建一个 `terragrunt.hcl` 文件，代替复制/粘贴 Terraform 代码的工作。例如，这里是 `live/stage/data-stores/mysql/terragrunt.hcl` 文件。

```
terraform {  
    source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"  
}  
  
inputs = {  
    db_name = "example_stage"  
    db_username = "admin"  
  
    # 使用 TF_VAR_db_password 环境变量设置密码  
}
```

正如你所看到的，`terragrunt.hcl` 文件使用和 Terraform 相同的 HashiCorp 配置语言（HCL）语法。当运行 `terragrunt apply` 命令时，代码会找到在 `terragrunt.hcl` 文件中的 `source` 参数，接下来 Terragrunt 将执行以下操作。

1. 检出 `source` 中指定的 URL 的代码到一个临时文件夹。`source` 的参数支持与 Terraform 模块相同的 URL 语法，因此你可以使用本地文件路径、Git URL、版本化的 Git URL（通过 `ref` 参数，如上例所示）等。
2. 在临时文件夹中运行 `terraform apply` 命令，将 `inputs = { ... }` 代码块中定义的输入变量传递给它。

这种方法的好处在于，实时存储库中的代码将被减少到每个模块仅包含一个 `terragrunt.hcl` 文件，该文件包含指向要使用的模块的指针（指向特定的版本），以及为特定环境设置的输入变量。这是所能获得的最简洁的代码。

Terragrunt 还可以保持 `backend` 配置的简洁。不必为每个模块重复定义 `bucket`、`key`、`dynamodb_table` 等参数。而是在每个环境下的 `terragrunt.hcl` 文件中进行定义。例如，`live/stage/terragrunt.hcl` 文件。

```
remote_state {
```

```
backend = "s3"

config = {
  bucket      = "<YOUR BUCKET>"
  key         = "${path_relative_to_include()}/terraform.tfstate"
  region      = "us-east-2"
  encrypt     = true
  dynamodb_table = "<YOUR_TABLE>"
}
}
```

在 `remote_state` 代码块中，使用与往常相同的方式配置 `backend` 参数，但 `key` 值略有不同。`key` 值中使用 Terragrunt 内置函数 `path_relative_to_include()`。这个函数返回此 `terragrunt.hcl` 根文件到包含这个文件的任何子模块之间的相对路径。例如，将这个根文件包含在 `live/stage/data-stores/mysql/terragrunt.hcl` 中，只需添加一个 `include` 代码块。

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  db_name = "example_stage"
  db_username = "admin"

  # 使用 TF_VAR_db_password 环境变量设置密码
}
```

在 `include` 代码块中，通过使用 Terragrunt 内置函数 `find_in_parent_folders()` 找到根目录的 `terragrunt.hcl` 文件。自动从该父文件中继承所有设置，包括 `remote_state` 配置。结果是，`mysql` 模块将使用所有来自根文件的相同的 `backend` 设置，只是 `key` 值将被自动解析为 `data-stores/mysql/terraform.tfstate`。这意味着 Terraform 状态文件将被保存在与实时存储库相同的文件夹结构中，这将很容易识别哪个模块产生了哪个状态文件。

要部署此模块，请运行 `terragrunt apply` 命令。

```
$ terragrunt apply
```

```
[terragrunt] Reading Terragrunt config file at terragrunt.hcl

[terragrunt] Downloading Terraform configurations from
    github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7

[terragrunt] Running command: terraform init -backend-config=(...)

(...)

[terragrunt] Running command: terraform apply

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

你可以在日志输出中看到 Terragrunt 读取了 `terragrunt.hcl` 文件，下载了指定的模块，运行 `terraform init` 命令来配置 `backend`（如果尚不存在，它甚至会自动创建 S3 bucket 和 DynamoDB 表），然后运行 `terraform apply` 命令部署所有内容。

现在，通过添加 `live/stage/services/hello-world-app/terragrunt.hcl` 文件，并在预发布环境中运行 `terragrunt apply` 命令，来部署 `hello-world-app` 模块。

```
terraform {
  source = "github.com/<OWNER>/modules//services/hello-world-app?ref=v0.0.7"
}

include {
  path          = find_in_parent_folders()
}

inputs          = {
  environment   = "stage"

  min_size      = 2
  max_size      = 2

  enable_autoscaling = false
}

db_remote_state_bucket = "<YOUR_BUCKET>"
```

```
    db_remote_state_key      = "<YOUR_KEY>"  
}
```

该模块使用 `include` 代码块从根目录的 `terragrunt.hcl` 文件中继承相同的 `backend` 设置，而 `key` 值正如所期望的那样，将被自动更新为 `services/hello-world-app/terraform.tfstate`。当所有功能在预发布环境中正常工作后，接下来可以在 `live/prod` 目录中创建类似的 `terragrunt.hcl` 文件，通过在每个模块中运行 `terragrunt apply` 命令，将完全相同的 v0.0.7 版本的工作推广到生产环境中。

## 将上述各点整合在一起

现在，我们已经学习了如何将应用程序代码和基础设施代码从开发环境推广到生产环境。表 8-1 为应用程序代码和基础设施代码工作流程比较。

表 8-1：应用程序代码和基础设施代码工作流程比较

	应用程序代码	基础设施代码
使用版本控制	<code>git clone</code> 应用程序存储在单独的代码库中 使用版本分支	<code>git clone</code> 使用 <code>live</code> 和 <code>modules</code> 代码库 不使用版本分支
本地执行代码	运行在本机 <code>ruby web-server.rb</code> <code>ruby web-server-test.rb</code>	运行在沙箱环境 <code>terraform apply</code> <code>go test</code>
更改代码	对代码进行更改 <code>ruby web-server.rb</code> <code>ruby web-server-test.rb</code>	对代码进行更改 <code>terraform apply</code> <code>go test</code> 使用测试阶段
提交代码评审	提交一个 pull request 加强编码标准评审	提交一个 pull request 加强编码标准评审

续表

	应用程序代码	基础设施代码
执行自动测试	通过 CI 服务器运行测试 单元测试 集成测试 端到端测试 静态分析	通过 CI 服务器运行测试 单元测试 集成测试 端到端测试 静态分析 <code>terraform plan</code>
合并和发布	<code>git tag</code> 生成版本化的、不可变的工件	<code>git tag</code> 使用带标签的代码库，作为版本化的、不可变的工件
部署	部署工具：Terraform、编排工具（如 Kubernetes、Mesos）、脚本工具 众多的部署策略：滚动部署、蓝绿部署、金丝雀部署 通过 CI 服务器发布 赋予 CI 服务器有限的特权 推广不可变的、版本化的工件到不同的环境	部署工具：Terraform、Atlantis、Terraform Enterprise、Terragrunt、脚本工具 有限的部署策略，确保处理错误：重试、 <code>errored.tfstate</code> 文件 通过 CI 服务器发布 赋予 CI 服务器管理员特权 推广不可变的、版本化的工件到不同的环境

如果遵循这个过程，你将能够在开发环境中运行应用程序代码和基础设施代码，进行测试、评审，打包为版本化的、不可变的工件，并将这些工件在环境之间进行推广，如图 8-6 所示。

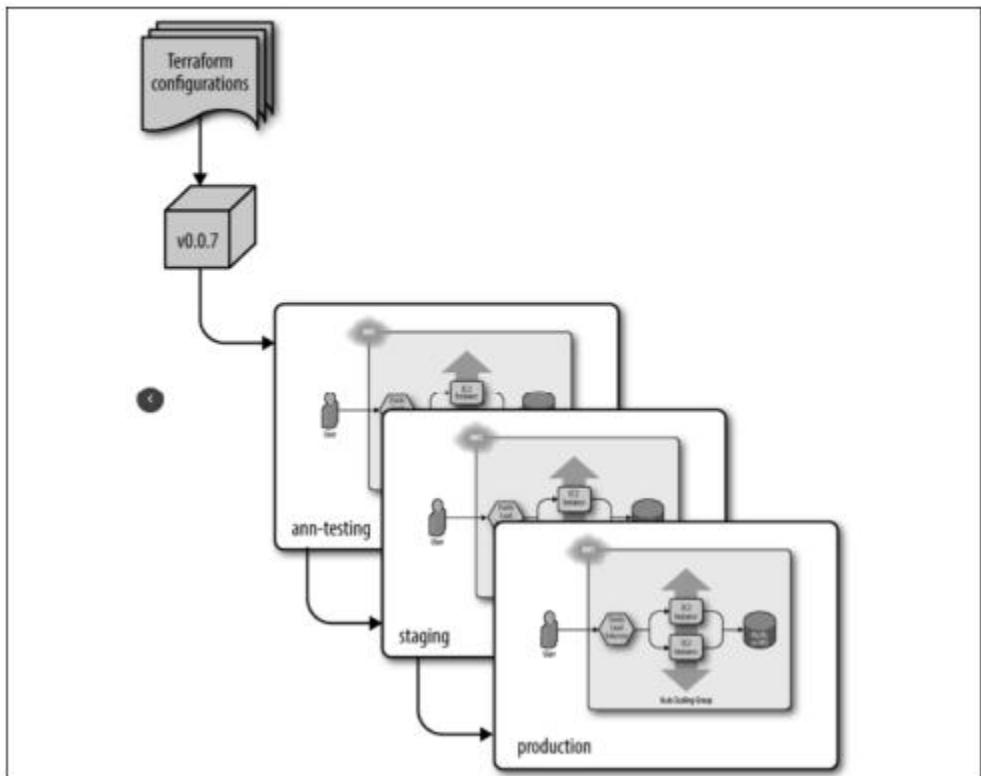


图 8-6：将版本化的、不可变的工件推广到每个环境

## 小结

如果已经学到了本书的最后一章，你应该具备了在现实世界中使用 Terraform 的全部知识，包括如何编写 Terraform 码；如何管理 Terraform 状态；如何使用 Terraform 创建可重用的模块；如何执行循环、if 表达式和部署；如何编写生产级 Terraform 代码；如何测试 Terraform 代码；以及如何以团队形式使用 Terraform。你已经通过示例，完成了部署和管理服务器、服务器集群、负载均衡器、数据库、计划任务、CloudWatch 警报、IAM 用户、可重用模块、零停机部署、自动测试等。最后，当完成所有操作后，请不要忘记在每个模块中运行 `terraform destroy` 命令。

Terraform 及更广泛的 IaC 工具的功效是，使用与管理和应用程序相同的编码原则，来管理应用程序周围的运维问题。这使得软件工程的全部功能，包括模块、代码评审、版本控制和自动测试，都可以用来管理基础设施。

如果正确使用 Terraform，团队将能够更快地部署并更快地响应更改。部署会变得像例行公事一般无聊。在运维世界中，无聊是一件非常好的事情。如果正确地完成了所有的工作，而不是将所有时间花费在手动管理基础设施上，那么团队将有更充足的时间来改进该基础设施，从而使工作更高效。

这是本书的结尾，但仅仅是你使用 Terraform 的旅程的开始。如果需要了解有关 Terraform、IaC 和 DevOps 的更多信息，请参照附录 A，获得推荐阅读的列表。如果你有任何反馈或问题，我很乐意通过邮件保持联系：[jim@ybrikman.com](mailto:jim@ybrikman.com)。感谢阅读本书！

## 附录 A

### 推荐阅读资料

下面是一些我能找到的关于 DevOps、基础设施即代码的书籍、博客、通讯文章以及演讲。

#### 书籍

- *Infrastructure as Code: Managing Servers in the Cloud*, 作者: Kief Morris, O'Reilly 出版
- *Site Reliability Engineering: How Google Runs Production Systems*, 作者: Betsy Beyer、Chris Jones、Jennifer Petoff 和 Niall Richard Murphy, O'Reilly 出版
- *The DevOps Handbook: How To Create World-Class Agility, Reliability, & Security in Technology Organizations*, 作者: Gene Kim, Jez Humble, Patrick Debois 和 John Willis, IT Revolution Press 出版
- *Designing Data Intensive Applications*, 作者: Martin Kleppmann, O'Reilly 出版
- *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 作者: Jez Humble 和 David Farley, Addison-Wesley Professional 出版
- *Release It! Design and Deploy Production-Ready Software*, 作者: Michael T. Nygard, The Pragmatic Bookshelf 出版
- *Kubernetes In Action*, 作者: Marko Luksa, Manning 出版
- *Leading the Transformation: Applying Agile and DevOps Principles at Scale*, 作者: Gary Gruver 和 Tommy Mouser, IT Revolution Press 出版

- *Visible Ops Handbook* by, 作者: Kevin Behr, Gene Kim 和 George Spafford, Information Technology Process Institute 出版
- *Effective DevOps*, 作者: Jennifer Davis 和 Katherine Daniels, O'Reilly 出版
- *Lean Enterprise*, 作者: ez Humble, Joanne Molesky 和 Barry O'Reilly, O'Reilly 出版
- *Hello, Startup: A Programmer's Guide to Building Products, Technologies, and Teams*, 作者: Yevgeniy Brikman, O'Reilly 出版

## 博客

- High Scalability (<http://highscalability.com/>)
- Code as Craft (<https://codeascraft.com/>)
- dev2ops (<http://dev2ops.org/>)
- AWS blog (<https://aws.amazon.com/blogs/aws/>)
- Kitchen Soap (<https://www.kitchensoap.com/>)
- Paul Hammant's blog (<https://paulhammant.com/>)
- Martin Fowler's blog (<https://martinfowler.com/bliki/>)
- Gruntwork blog (<https://blog.gruntwork.io/>)
- Yevgeniy Brikman blog (<https://www.ybrikman.com/writing/>)

## 演讲

- “Reusable, composable, battle-tested Terraform modules” (<http://bit.ly/32b28JD>), Yevgeniy Brikman
- “5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code” (<http://bit.ly/2ZCcEfI>), Yevgeniy Brikman
- “Infrastructure as code: running microservices on AWS using Docker, Terraform, and ECS” (<http://bit.ly/30TYaVu>), Yevgeniy Brikman
- “Agility Requires Safety” (<http://bit.ly/2YJuqJb>), Yevgeniy Brikman
- “Adopting Continuous Delivery” (<https://youtu.be/ZLBhVEo1OG4>), Jez Humble

- “Continuously Deploying Culture” (<https://vimeo.com/51310058>), Michael Rembetsy 和 Patrick McDonnell
- “10+ Deploys Per Day: Dev and Ops Cooperation at Flickr” (<https://youtu.be/LdOc18KhtT4>), John Allspaw 和 Paul Hammond
- “Why Google Stores Billions of Lines of Code in a Single Repository” (<https://youtu.be/W71BTkUbdqE>), Rachel Potvin
- “The Language of the System” ([https://youtu.be/ROor6\\_NGIWU](https://youtu.be/ROor6_NGIWU)), Rich Hickey
- “Don’t Build a Distributed Monolith” (<https://youtu.be/-vzp0Y4Z36Y>), Ben Christensen
- “Real Software Engineering” (<https://youtu.be/NP9AIUT9nos>), Glenn Vanderburg

## 通讯文章

- *DevOps Weekly*
- *DevOpsLinks*
- *Gruntwork Newsletter*
- *Terraform: Up & Running Newsletter*

## 在线论坛

- Terraform Google Group
- DevOps subreddit

## 关于作者

**Yevgeniy ( Jim ) Brikman** 喜欢编程、写作、演讲、旅行和举重。他是 Gruntwork 公司的联合创始人，该公司提供 DevOps 服务。他还是 O'Reilly Media 出版的另一本书 *Hello, Startup* 的作者。作为一名软件工程师，他曾就职于领英（LinkedIn）、TripAdvisor、思科（Cisco）及 Thomson Financial，并在康奈尔大学获得学士和硕士学位。更多信息请访问 [ybrikman.com](http://ybrikman.com)。

## 封面介绍

本书封面上的动物：飞龙蜥蜴（*Draco volans*），是一种小型爬行动物，因使用翼状襟翼滑行的能力而得名，皮肤被称为翼膜。翼膜颜色鲜艳，可以让其滑行长达 8m。飞龙蜥蜴在东南亚许多国家和地区都很常见，包括印度尼西亚、越南、泰国、菲律宾和新加坡。

飞龙蜥蜴以昆虫为食，长度可以长到 20cm 以上；主要生活在森林地区，从一棵树滑行到另一棵树上寻找猎物并躲避捕食者。只有产卵时，雌性飞龙蜥蜴才会从树上下来，将卵产在地面上隐蔽的洞穴里。雄性飞龙蜥蜴具有很强的领地意识，会在树上追逐对手。尽管曾经被认为是有毒的动物，但是飞龙蜥蜴对人类无害，有时会被作为宠物饲养。它们目前没有受到威胁或濒危。

O'Reilly 封面上的许多动物都濒临灭绝；所有这些动物对世界都很重要。要了解如何提供帮助，请访问 [animals.oreilly.com](http://animals.oreilly.com)。

封面图片来自约翰逊的《自然史》。