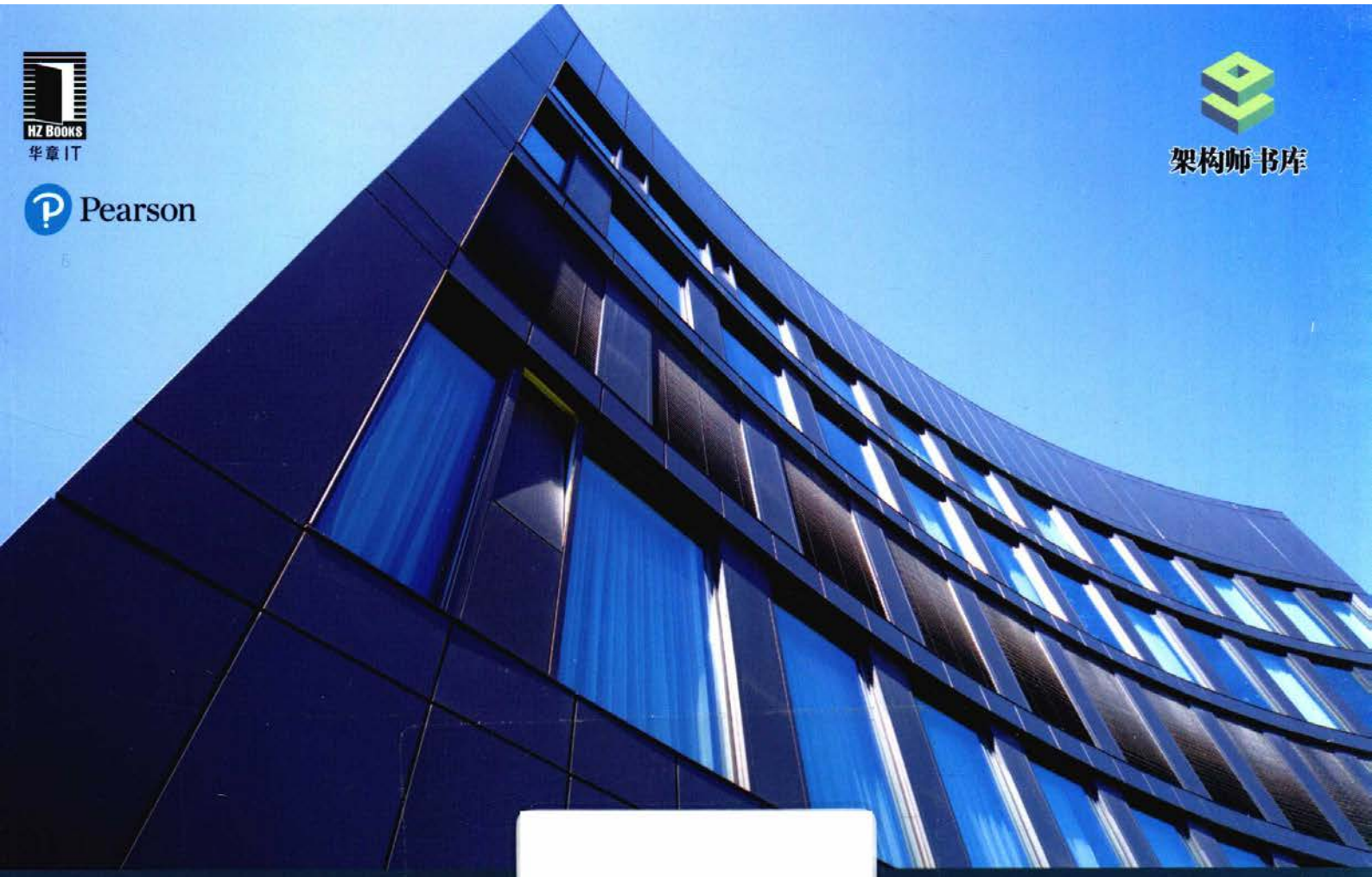


HZ BOOKS
华章IT

Pearson

架构师书库

DevOps

软件架构师行动指南

[澳] 伦恩·拜斯（Len Bass） 英戈·韦伯（Ingo Weber） 朱黎明（Liming Zhu）著
胥峰 任发科 等译

DevOps
A Software Architect's Perspective

为软件架构师所写的第一本DevOps完全指南
通过3个经典案例研究，系统讲解在不同场景下应用DevOps实践的方法
盛大游戏和唯品会的资深DevOps专家联合翻译

机械工业出版社
China Machine Press



架构师书库

DevOps

A Software Architect's Perspective

DevOps

软件架构师行动指南

[澳] 伦恩·拜斯（Len Bass） 英戈·韦伯（Ingo Weber） 朱黎明（Liming Zhu）著

胥峰 任友科 等译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

DevOps：软件架构师行动指南 /（澳）伦恩·拜斯（Len Bass）等著；胥峰等译．—北京：机械工业出版社，2017.3（2017.6 重印）
（架构师书库）
书名原文：DevOps: A Software Architect's Perspective

ISBN 978-7-111-56261-0

I. D… II. ① 伦… ② 胥… III. 软件工程 IV. TP311.5

中国版本图书馆 CIP 数据核字（2017）第 039927 号

本书版权登记号：图字：01-2015-5197

Authorized translation from the English language edition, entitled DevOps: A Software Architect's Perspective, 9780134049847 by Len Bass, Ingo Weber, Liming Zhu, published by Pearson Education, Inc., Copyright © 2015.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2017.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

DevOps：软件架构师行动指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：盛思源　　责任校对：董纪丽
印　　刷：中国电影出版社印刷厂　　版　　次：2017 年 6 月第 1 版第 3 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：16.5
书　　号：ISBN 978-7-111-56261-0　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

·· 译 者 序 ··

为什么要翻译这本书

2016 年 7 月中旬，机械工业出版社华章公司王春华老师问我有没有兴趣翻译一本 DevOps 实践的书，在看过这本书的英文版之后，我欣然答应了。

因为，我知道这是一本有重要价值的书，我愿意花时间把它翻译出来，我想让它被更多人研读学习。

自从在 Agile 2008 大会上讨论 DevOps 一词以来，DevOps 理念得到了 IT 从业者的认可和追捧。数年之后，再次客观地审视 DevOps 运动时，我们发现从整个信息技术领域来看，DevOps 还没有成为开发和运维的事实标准。

原因何在?

我想，其中最重要的一个原因是，很多组织缺少在业务中引入 DevOps 实践的专家知识，也没有很多可以借鉴的成功案例。而这本书正好可以解决这个问题。

本书从软件架构师的视角讲解了引入 DevOps 实践所需要掌握的技术能力，涵盖了运维、部署流水线、监控、安全与审计以及质量关注。第四部分通过 3 个经典案例研究，讲解了在不同场景下应用 DevOps 实践的方法。这对于想应用 DevOps 实践的组织具有切实的指导意义。

初识本书，我有一种相见恨晚的感觉。研读之后，这种感觉尤甚。相信读者也一定会有这样的感觉。

衷心希望本书中文版能给读者带来有价值的信息，为推动 DevOps 进一步发展和应用做出贡献。

读者对象

本书适合以下几类读者阅读：

- 软件架构师
- 中高级运维工程师
- 计算机相关专业的学生
- 希望提高 IT 生产力的人员

勘误支持

虽然译者试图努力保证本书翻译中不出现错误，但鉴于译者的知识和视角，书中难免出现用词错误、技术适用性的问题。在此，译者恳请读者不吝指教，指出错误。请读者发送邮件到 xufengnju@163.com 或者加入 QQ 群 434242482，帮助译者修正错误。本书的勘误将在网站 http://xufeng.info/ 刊载。

译者说明

本书由来自盛大游戏、唯品会等互联网企业的资深 DevOps 专家联合翻译完成。

除胥峰、任发科外，参与翻译的人员还有米全喜、吴昊、黄灵等。全书由胥峰完成统稿和技术审校。

致谢

感谢华章公司引进了本书的中译本版权，这是本中译本得以面市的最核心要素。

感谢华章公司盛思源、关敏和王春华老师，你们专业的编辑能力为本书提供了重要的质量保证。

感谢盛大游戏各位领导对我翻译本书的关心和支持。

感谢我的妻子吕宁和可爱的女儿胥欣，谢谢你们的支持和理解。

胥峰

2016年10月

前　言

多年以来，我们一直在探索研究运维中的问题。自然而然地，我们也一直在追踪DevOps运动。它正在沿着Gartner成熟度曲线向上发展。这种现象有着坚实的业务原因。我们能够找到从信息技术经理视角对DevOps的探讨（例如小说《凤凰项目：一个IT运维的传奇故事》），也能找到从项目经理视角对DevOps的探讨（例如《持续交付：发布可靠软件的系统方法》）。而且，关于文化变革和文化变革对于打破组织单元之间障碍的意义，也有大量的资料。

然而，令我们感到失望的是，鲜有从软件架构师角度对DevOps探讨的资料。把运维人员视为首要干系人并且聆听他们的需求当然是非常重要的。使用工具来支持运维和项目管理也是非常重要的。然而，我们强烈感觉到，DevOps并不仅仅是关于干系人管理和工具使用的。

事实上，正是因为目前缺少这方面的资料，所以我们才决定撰写本书，把这些缺失的内容填充起来。DevOps与设计、过程、工具化和组织结构之间的相互作用令人着迷。我们试图回答两个主要问题：作为软件架构师，我需要做出什么样的技术决策才能够实现DevOps目标？DevOps领域的其他参与者会对我产生什么样的影响？

以上这些问题的答案是，实现DevOps目标可能包含着对你的系统架构和角色以及责任做出重大变更，在把系统部署到生产环境以及在生产环境中支持这些系统时，需要以上角色及责任。

正如软件架构师必须理解他们所设计与构建的系统的业务情境和目标一样，理解DevOps也要理解组织和业务情境，同时也要理解技术和运维情境。我们探索了所有这些内容。

本书的主要读者是那些曾经被问过或者将被问到“这个项目或者组织是否应该采用DevOps实践”这个问题的执业软件架构师。甚至他们不是被追问这个问题，而是他们有可能被告知是否应该采用DevOps实践。与所有书一样，我们期望还有其他类型的读者。那些有兴趣学习更多关于软件架构实践的学生应该可以在本书中找到一些感兴趣的资料。那些想探索研究DevOps课题的研究人员可以在本书中找到重要的背景资料。然而，我们的主要读者还是那些执业软件架构师。

概述

在本书的开始，我们讨论了 DevOps 的背景。在第一部分中，我们研究了 DevOps 的目标以及期望使用 DevOps 来解决的问题。我们会涉及组织和文化问题，也会涉及 DevOps 实践和敏捷方法学的关系。

在第 2 章中，我们研究了云。随着云即平台（Cloud as a Platform）的增长，DevOps 实践也得到了发展。理论上，DevOps 和云即平台是可分开的，但是事实上，虚拟化和云是推动 DevOps 实践的重要力量。

在介绍背景知识的最后一章（第 3 章）中，我们按照信息技术基础设施库（Information Technology Infrastructure Library，ITIL）的理论研究了运维。ITIL 是对运维组的最重要功能进行组织的体系。不是所有的运维工作都包含在 DevOps 实践中，但是理解运维组的职责却会提供重要的上下文信息，特别是当需要理解角色和职责时。

第二部分描述了部署流水线（Deployment Pipeline）。在这一部分中，我们首先探索了微服务架构风格，这出现在第 4 章中。为了应用 DevOps 实践而以微服务风格进行系统架构并不是必需的，但是，却能用微服务架构风格来解决 DevOps 所需要解决的一些问题。

在第 5 章中，我们简略介绍了构建和测试过程以及工具链。理解这些内容很重要，但它们不是我们的焦点。我们介绍了把系统部署到生产环境所关联的各种不同环境以及在这些环境上执行的不同类型的测试。因为在 DevOps 中用到的很多工具也在构建和测试过程中被用到了，所以我们提供了理解这些工具和如何控制它们的上下文信息。

我们以讨论部署结束第二部分。DevOps 的目标之一是加速部署。实现这个目标的一种技术是，使得每个开发团队可以在完成准备后独立地部署他们的代码。独立部署带来了很多一致性的问题。我们将讨论不同的部署模型；管理同时运行在生产环境中不同版本的系统；在发生错误时回滚；以及与实际把系统部署到生产环境中有关的其他主题。

第二部分提供了从功能视角观察部署实践的内容。然而，与其他任何系统一样，质量视角常常是控制系统设计和系统接受度的主要因素。在第三部分中，我们聚焦在横切关注点上。我们首先在第 7 章中讨论了监控和现场测试（live testing）。现代软件测试实践在将系统部署到生产环境中以后并没有结束。首先，系统被广泛地监控以检测问题；其次，在将系统部署到生产环境以后，测试正以各种形式继续进行。

另一个横切关注点是安全。我们在第 8 章中讨论了该内容。我们展示了环境中各种不同类型的安全控制，包括组织范围和特定系统范围的安全控制。我们讨论了与实现安全相关联的不同角色以及在安全审计案例中如何评估这些角色。

安全不是质量关注的全部。在第 9 章中，我们讨论了与 DevOps 实践有关的其他方面的质量。我们讨论的主题包括部署流水线的性能、可靠性和可修改性。

第三部分的最后，我们在第 10 章中讨论了业务关注。像 DevOps 这种涉及面广泛的

实践，如果得不到来自管理层的支持，是无法被采用的。业务计划是寻求这种支持的一种典型方法。因此，我们描述了为引进 DevOps 所需要准备的业务计划的组成元素，然后讨论了如何进行论证、推出和测量业务计划。

在第四部分中，我们描述了 3 个案例研究。这些实施了 DevOps 实践的组织给我们讲解了他们所采用的一些技巧。第 11 章讨论了为了实现业务连续性如何维护两个数据中心；第 12 章展示了一个持续部署流水线的细节；第 13 章描述了一个组织是如何迁移到微服务架构上的。

作为本书的结尾，我们在第五部分展望了 DevOps 的未来。第 14 章描述了我们的研究以及它是如何基于把运维视作一系列过程来进行的。第 15 章预测了未来 3 ～ 5 年 DevOps 将如何发展。

致谢

本书离不开众多人的帮助。在此，感谢 Chris Williams、John Painter、Daniel Hand 和 Sidney Shek，他们为案例研究做出了贡献。同时，感谢 Adnene Guabtni、Kanchana Wickremasinghe、Min Fu 和 Xiwei Xu 在有些章节中提供的帮助。

感谢 Manuel Pais 帮助我们编排了案例研究。感谢 Philippe Kruchten、Eoin Woods、Gregory Hartman、Sidney Shek、Michael Lorant、Wouter Geurts 和 Eltjo Poort 对本书做出了评论或者为本书的某些方面做出了贡献。

下列人士对第 13 章做出了评论，在此表示感谢：Jean-Michel、Lemieux、Greg Warden、Robin Fernandes、Jerome Touffe-Blin、Felipe Cuozzo、Pramod Korathota、Nick Wright、Vitaly Osipov、Brad Baker 和 Jim Watts。

在出版阶段，Addison-Wesley 出版社的编辑用他们一贯的专业和高效的工作为本书顺利出版做出了贡献。他们的专业知识和工作经验提升了本书的质量。

最后，感谢 NICTA 和 NICTA 的管理层。NICTA 是由澳大利亚政府通过通信部（Department of Communications）和由澳大利亚研究委员会（Australian Research Council）通过 ICT 卓越中心（Centre of Excellence）项目组建的。如果没有他们的慷慨支持，本书将难以完成。

图例说明

对于图，我们使用了 4 种不同的图例。我们使用架构标注法来识别我们使用的关键架构概念；使用业务流程建模标注（Business Process Model and Notation，BPMN）来描述一些过程；使用波特价值链标注（Porter's Value Notation）来描述一些其他的过程；使

用 UML 时序图来组织活动的顺序。这里没有给出 UML 时序图标注。我们给出了通过其他方法进行的标注。

架构



图 P-1　人员，包括个体和组



图 P-2　组件（运行时实体）、模块（代码时实体集合）和数据流



图 P-3　特殊化的实体



图 P-4　实体集合

业务流程建模标注

我们使用业务流程建模标注来描述事件和活动 [OMG 11]。



图 P-5　事件标识



图 P-6　活动和活动序列

波特价值链

该标注用来描述过程（包含在业务流程建模标注中建模的活动）。

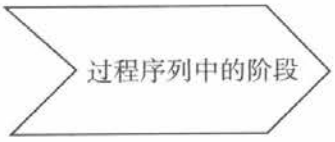


图 P-7　价值链中的条目

•• 目　　录 ••

译者序
前言

第一部分　背　　景

第 1 章　DevOps 是什么 …… 2

1.1　概述 …… 2
1.1.1　定义 DevOps …… 2
1.1.2　DevOps 实践 …… 3
1.1.3　持续部署的例子：IMVU …… 5
1.2　为什么是 DevOps …… 5
1.2.1　发布过程 …… 5
1.2.2　配合不佳的原因 …… 7
1.2.3　运维人员能力有限 …… 7
1.3　DevOps 视角 …… 8
1.3.1　自动化 …… 8
1.3.2　开发团队的职责 …… 9
1.4　DevOps 与敏捷 …… 9
1.5　团队结构 …… 10
1.5.1　团队规模 …… 10
1.5.2　团队角色 …… 10
1.6　协作 …… 13
1.6.1　协作的形式 …… 13
1.6.2　团队协作 …… 14
1.6.3　跨团队协作 …… 14
1.7　障碍 …… 15

1.7.1 文化及组织类型 …… 15
1.7.2 部门类型 …… 16
1.7.3 筒仓思维方式（Silo Mentality）…… 17
1.7.4 工具支持 …… 17
1.7.5 人员问题 …… 17
1.8 小结 …… 18
1.9 更多阅读材料 …… 18

第 2 章 云即平台 …… 20

2.1 概述 …… 20
2.2 云的特性 …… 21
2.2.1 虚拟化 …… 22
2.2.2 IP 和域名系统管理 …… 23
2.2.3 平台即服务 …… 25
2.2.4 分布式环境 …… 25
2.3 独特的云特性对 DevOps 的影响 …… 30
2.3.1 环境 …… 30
2.3.2 轻松创建虚拟机 …… 31
2.3.3 数据考量 …… 31
2.4 小结 …… 32
2.5 更多阅读材料 …… 33

第 3 章 运维 …… 34

3.1 概述 …… 34
3.2 运维服务 …… 34
3.2.1 供给硬件 …… 34
3.2.2 供给软件 …… 35
3.2.3 IT 功能 …… 36
3.2.4 服务级别协议 …… 36
3.2.5 容量规划 …… 36
3.2.6 业务连续性和安全 …… 37
3.2.7 服务策略 …… 38
3.2.8 服务设计 …… 39
3.2.9 服务移交 …… 39

3.2.10 服务运维 …… 40
3.2.11 服务运维概念 …… 40
3.3 服务运维功能 …… 41
3.4 持续服务改进 …… 42
3.5 运维和 DevOps …… 43
3.6 小结 …… 44
3.7 更多阅读材料 …… 44

第二部分 部署流水线

第 4 章 整体架构 …… 48

4.1 DevOps 实践是否需要架构调整 …… 48
4.2 架构结构总览 …… 49
4.2.1 协作模式 …… 50
4.2.2 资源管理 …… 51
4.2.3 架构元素之间的映射 …… 52
4.3 微服务架构的质量 …… 52
4.3.1 可靠性 …… 53
4.3.2 可修改性 …… 54
4.4 团队的亚马逊规则 …… 55
4.5 现有系统的微服务方案 …… 56
4.6 小结 …… 56
4.7 更多阅读材料 …… 57

第 5 章 构建与测试 …… 58

5.1 概述 …… 58
5.2 在部署流水线中移动系统 …… 59
5.2.1 可追溯性 …… 59
5.2.2 环境 …… 60
5.3 横切关注点 …… 61
5.4 开发及提交前测试 …… 63
5.4.1 版本控制与分支 …… 63
5.4.2 功能开关 …… 65
5.4.3 配置参数 …… 66

5.4.4 在开发和提交前测试中的测试 …… 67
5.5 构建与集成测试 …… 67
5.5.1 构建脚本 …… 67
5.5.2 打包 …… 68
5.5.3 持续集成与构建状态 …… 69
5.5.4 集成测试 …… 70
5.6 用户验收测试 / 预发布 / 性能测试 …… 70
5.7 生产环境 …… 71
5.7.1 早期发布测试 …… 71
5.7.2 错误检测 …… 72
5.7.3 现场测试 …… 72
5.8 事件 …… 73
5.9 小结 …… 73
5.10 更多阅读材料 …… 74

第 6 章 部署 …… 75

6.1 概述 …… 75
6.2 部署管理的策略 …… 76
6.2.1 蓝 / 绿部署 …… 76
6.2.2 滚动升级 …… 77
6.3 逻辑一致性 …… 78
6.3.1 相同服务的多个版本同时存在 …… 78
6.3.2 兼容数据库中保存的数据 …… 81
6.4 打包 …… 82
6.5 多环境部署 …… 84
6.6 部分部署 …… 86
6.6.1 金丝雀测试 …… 86
6.6.2 A/B 测试 …… 87
6.7 回滚 …… 87
6.8 工具 …… 89
6.9 小结 …… 90
6.10 更多阅读材料 …… 90

第三部分　横切关注点

第 7 章　监控 …… 94

7.1　概述 …… 94

7.2　监控什么 …… 95

7.2.1　故障检测 …… 96

7.2.2　性能下降检测 …… 96

7.2.3　容量规划 …… 97

7.2.4　用户交互 …… 98

7.2.5　入侵检测 …… 99

7.3　如何监控 …… 99

7.3.1　基于代理的监控和无代理的监控 …… 101

7.3.2　监控运维活动 …… 102

7.3.3　收集和存储 …… 102

7.4　什么时候变更监控配置 …… 103

7.5　解释监控数据 …… 103

7.5.1　日志 …… 104

7.5.2　绘图和展示 …… 105

7.5.3　警报和警告 …… 105

7.5.4　诊断和反应 …… 106

7.5.5　监控 DevOps 过程 …… 106

7.6　挑战 …… 107

7.6.1　挑战 1：持续变更下的监控 …… 107

7.6.2　挑战 2：自下向上与自上向下和在云中的监控 …… 108

7.6.3　挑战 3：监控微服务架构 …… 109

7.6.4　挑战 4：处理大容量的分布式（日志）数据 …… 109

7.7　工具 …… 109

7.8　从监控数据中诊断出异常——Platformer.com 的案例 …… 110

7.8.1　背景 …… 111

7.8.2　数据收集 …… 112

7.8.3　检测异常 …… 112

7.8.4　思考 …… 113

7.9　小结 …… 113

7.10　更多阅读材料 …… 114

第 8 章　安全与安全审计 …… 115

8.1　安全是什么 …… 115
8.2　威胁 …… 117
8.3　需要保护的资源 …… 118
8.4　安全角色和活动 …… 120
8.5　身份管理 …… 122
8.5.1　认证 …… 123
8.5.2　授权 …… 125
8.6　访问控制 …… 126
8.6.1　阻止访问 …… 127
8.6.2　谁负责预防控制 …… 129
8.7　检测、审计和拒绝服务 …… 129
8.8　开发 …… 130
8.9　审计者 …… 130
8.10　应用设计考虑 …… 131
8.11　部署流水线设计考虑 …… 132
8.12　小结 …… 133
8.13　更多阅读材料 …… 134

第 9 章　其他非功能需求 …… 135

9.1　概述 …… 135
9.2　可重复性 …… 136
9.2.1　在恰当的层级上定义和执行过程 …… 136
9.2.2　版本控制所有事物 …… 138
9.3　性能 …… 139
9.3.1　测量重要的事物 …… 139
9.3.2　提高资源使用率 …… 140
9.4　可靠性 …… 141
9.4.1　理解不同服务的可靠性特性 …… 141
9.4.2　早期检测和修复错误 …… 142
9.5　可恢复性 …… 142
9.6　互操作性 …… 143
9.6.1　注意接口的互操作性 …… 143
9.6.2　理解现有的数据模型 …… 143

9.7 可测试性 …… 144
9.8 可修改性 …… 145
9.8.1 一个工具内的修改 …… 145
9.8.2 工具之间交互行为的修改 …… 146
9.9 小结 …… 146
9.10 更多阅读材料 …… 147

第 10 章 业务关注点 …… 148

10.1 概述 …… 148
10.2 业务案例 …… 148
10.2.1 问题和解决问题所带来的好处 …… 149
10.2.2 成本 …… 149
10.2.3 干系人影响 …… 150
10.2.4 风险及其减缓 …… 151
10.2.5 推出计划 …… 153
10.2.6 成功标准 …… 154
10.3 度量和对 DevOps 实践的合规性 …… 155
10.3.1 测量 DevOps 实践的成功度 …… 155
10.3.2 测量对 DevOps 实践的合规性 …… 156
10.3.3 测量干系人的满意度 …… 157
10.4 Dev 和 Ops 之间的交互点 …… 157
10.4.1 许可 …… 157
10.4.2 事故处理 …… 158
10.5 小结 …… 159
10.6 更多阅读材料 …… 159

第四部分 案例研究

第 11 章 支持多数据中心 …… 162

11.1 概述 …… 162
11.2 当前的情况 …… 163
11.3 业务逻辑和 Web 层 …… 163
11.3.1 应用逻辑 …… 163
11.3.2 基础设施 …… 164

11.3.3 增加一个应用 …… 164
11.3.4 发现基础设施 …… 165
11.4 数据库层 …… 167
11.4.1 事务数据 …… 167
11.4.2 基础设施支持 …… 168
11.4.3 会话数据 …… 168
11.5 其他基础设施工具 …… 168
11.5.1 gem 存储库服务器 …… 169
11.5.2 Elasticsearch …… 169
11.5.3 域名系统 …… 169
11.6 数据中心切换 …… 170
11.6.1 受控切换步骤 …… 170
11.6.2 非受控切换 …… 174
11.6.3 定义和自动化切换步骤 …… 175
11.7 测试 …… 177
11.7.1 数据中心切换应用程序 …… 177
11.7.2 基础设施测试 …… 177
11.7.3 持续交付流水线 …… 177
11.8 小结 …… 178
11.9 更多阅读材料 …… 179

第 12 章 实施企业的持续部署流水线 …… 180

12.1 概述 …… 180
12.2 组织背景 …… 180
12.3 持续部署流水线 …… 182
12.3.1 持续部署流水线工具 …… 183
12.3.2 使用 AWS CloudFormation 的环境定义 …… 184
12.3.3 标准化的应用程序生命周期概览及其使用 …… 186
12.3.4 标准化的应用程序生命周期阶段 …… 188
12.3.5 管理复杂的应用程序和流水线状态 …… 194
12.3.6 管理持久化 …… 196
12.4 让安全成为持续部署流水线的基础 …… 196
12.4.1 使用 Amazon CloudFormation 分离职责 …… 196
12.4.2 身份和访问管理 …… 197

12.5 高级概念 …… 198
12.5.1 最小化生产环境和非生产环境之间的偏移 …… 198
12.5.2 解决供应商的限制 …… 198
12.5.3 厂商锁定 …… 199
12.5.4 新的 AWS 内置服务的展望 …… 199
12.6 小结 …… 199
12.7 更多阅读材料 …… 200

第 13 章 迁移到微服务 …… 202

13.1 Atlassian 概述 …… 202
13.2 构建部署微服务的平台 …… 203
13.3 BlobStore：一个微服务例子 …… 206
13.3.1 架构 …… 206
13.3.2 通过纯函数式架构和编程实现安全性和性能 …… 207
13.3.3 解决“非功能需求” …… 210
13.4 开发过程 …… 210
13.4.1 开发人员和支持 …… 211
13.4.2 构建和部署流水线 …… 212
13.4.3 客户应用的生产环境的零停机时间路径 …… 214
13.5 BlobStore 演进 …… 215
13.6 小结 …… 219
13.7 更多阅读材料 …… 219

第五部分 走向未来

第 14 章 作为过程的运维 …… 222

14.1 概述 …… 222
14.2 动机和概览 …… 223
14.3 离线活动 …… 224
14.4 在线活动 …… 227
14.4.1 错误检测 …… 227
14.4.2 错误恢复 …… 229
14.5 错误诊断 …… 229
14.6 监控 …… 231

14.7 小结 …… 231
14.8 更多阅读材料 …… 231

第 15 章 DevOps 的未来 …… 232

15.1 概述 …… 232
15.2 组织问题 …… 233
15.2.1 DevOps 活动中可能涉及的其他组 …… 233
15.2.2 所有关系和重组 …… 234
15.2.3 授权与控制 …… 234
15.3 过程问题 …… 235
15.3.1 厂商锁定和标准 …… 235
15.3.2 计费模型 …… 235
15.3.3 变更的速度 …… 236
15.4 技术问题 …… 237
15.4.1 持续部署流水线概念 …… 237
15.4.2 在持续部署流水线中获得质量 …… 239
15.4.3 实现 …… 239
15.5 错误报告和修复 …… 240
15.6 结束语 …… 240
15.7 更多阅读材料 …… 240

参考文献 …… 241

第一部分

背　景

这一部分为本书的后续章节提供了必要的背景知识。DevOps 是一项运动，它设想在开发组和运维组之间没有冲突。DevOps 的出现与云发展成为大小型组织的基本平台是同时发生的。第一部分有 3 章。

在第 1 章中，我们将定义 DevOps，并且讨论 DevOps 的各种驱动力。DevOps 是一个包罗万象的术语，它可以涵盖多个含义，包括让开发人员和运维人员互相沟通；允许开发团队自动化地部署到生产环境；当在生产环境中发现错误时，让开发团队成为第一个响应者。在这一章中，我们将梳理出各种关注点，并且关于 DevOps 是什么、它的驱动力和目标是什么以及如何实现这些目标，我们也将给出条理清晰的描述。

为了理解某个 DevOps 实践是如何工作的，有必要了解云是如何工作的，我们将在第 2 章中讨论这一内容。特别是，你应该知道虚拟机是如何工作的、IP 地址是如何使用的、域名系统（Domain Name System，DNS）服务器的角色和如何操作它，以及负载均衡器和监控器是如何交互以提供按需伸缩的。

DevOps 包含了对 Dev 和 Ops 实践的修改。在第 3 章中，我们将讨论 Ops 整体。其中讨论了 Ops 提供给组织的服务，也引入了 Ops 的责任——从支持部署的应用到强制组织范围的安全规则。

第 1 章　DevOps 是什么

有人告诉我，我放在书中的每个方程式都会让书的销量减半。
于是我决定一个方程式也不放。

——史蒂芬·霍金

1.1　概述

本书试图回答的问题是："我为什么要关注 DevOps，它对我有何影响？"阅读本书可以找到详尽的答案，不过也可以简而言之：如果你的工作与构建软件系统有关并且你所在的组织有意缩短新功能推向市场的时间，那你就应该关注 DevOps。这是 DevOps 产生的驱动力，DevOps 实践将影响你们如何组织团队、如何构建系统，甚至影响你们所构建系统的架构。如果你是软件工程领域的学生或研究人员，你应该关注采用 DevOps 实践会如何影响你们正在处理的问题。如果你是教育工作者，你也应该关注，因为在课程中融入 DevOps 材料可以帮助学生学习最新的开发实践。

我们首先给出 DevOps 的定义并提供了一个简短的示例，然后介绍 DevOps 运动产生的驱动力、概貌，以及阻碍 DevOps 成功实施的障碍。对 DevOps 的大部分描述都是讨论各种组织和文化方面的问题的。本章先概述这些主题，并围绕这些主题展开本书的剩余章节。

1.1.1　定义 DevOps

2013 年根据应用开发的 Gartner 技术成熟度曲线，将 DevOps 分类为"处于上升期"的类别。这个分类意味着 DevOps 正在成为一个流行术语，正因如此，其本身还没有良好的定义，并且浮夸之词四溢。我们对 DevOps 的定义关注目标，而不关注方式。

DevOps 是一套实践方法，在保证高质量的前提下缩短系统变更从提交到部署至生产环境的时间。

在深入探讨这套实践包含的内容之前，首先看看这个定义隐含的一些内容。

- 在部署对系统的变更时（通常是代码形式），质量很重要。质量意味着适合于包括最终用户、开发人员或系统管理员在内的各种干系人使用。质量还涵盖了可用性、安全性、可靠性及其他属性。保证质量的一种方法是，在把修改后的代码放到生产环境之前必须跑通各种自动化测试用例。另一种方法是，在把变更对外开放之

前，先让一小部分用户对生产环境的变更进行测试。还有一种方法是，对新部署的代码密切监控一段时间。我们在定义中没有说明如何保证质量，但我们要求生产环境代码是高质量的。

- 这个定义要求交付机制也是高质量的。这意味着交付机制应该具备高度的可靠性和可重复性。如果交付机制经常失效，所需的时间就增加了。如果交付变更的方式出现错误，所部署的系统的质量就会受到影响，例如降低可用性或可靠性。
- 我们认为有两个时间周期很重要。一个是开发人员提交新开发的代码的时间。这标志着基本的开发过程结束，部署工作开始。另一个是把代码部署到生产环境的时间。在第 6 章将看到，在将代码部署到生产环境后，有一段时间是现场测试代码并密切监控潜在的问题。在代码通过了现场测试及密切监控后，就可以认为是正式生产环境系统的一部分了。这有别于把代码部署到生产环境进行现场测试和密切监控，并且在通过测试后把新部署的代码提升为等同于以前部署的代码。
- 我们的定义是目标导向的。我们没有说明实践的形式或者是否使用工具来实施。如果一种实践的目的是减少开发人员从提交代码到部署到生产环境之间的时间，那么不论是否包括了敏捷方法、工具和协作形式，这都是一种 DevOps 实践。这与其他几个定义是不同的。例如，维基百科（Wikipedia）强调不同干系人之间的交流、协作与融合，但没有说明这种交流、协作和融合的目的。时间方面的目标是隐含的。其他定义强调 DevOps 与敏捷方法之间的联系。同样，也没有说明使用敏捷方法在开发时间或生产系统质量方面的收益。还有些定义强调使用的工具，却没有提及 DevOps 实践的目标、所需的时间，或者质量。
- 最后，定义中说明的目标并不限于 DevOps 用于测试和部署的实践。为了实现这些目标，很重要的一点是在需求阶段就包含运维人员的视角——这比变更提交阶段要早得多。类似地，这个定义也并不意味着把开发的内容部署到生产环境之后，DevOps 实践就结束了。DevOps 的目标是确保部署的系统在整个生命周期中都是高质量的。这样，有助于实现这个目标的监控活动也包含在了定义中。

1.1.2　DevOps 实践

我们接下来标识 5 种不同类别的 DevOps 实践。这些都是一些有关 DevOps 的著述提及的，并且满足我们的定义。

- 从需求的角度把运维人员视为首要干系人。这些实践符合定义中高质量方面的要求。运维人员有一套与日志及监控相关的需求。例如，日志信息需要能够被运维人员理解和使用。在需求开发阶段引入运维人员可以确保这种类型的需求能够考虑到。
- 让开发人员更多地负责相关事件处理。这些实践的目的是缩短错误从发现到修复的时间。使用这些实践的组织通常是，新部署的内容在开始一段时间由开发人员

负主要责任，之后由运维人员负主要责任。

- 强制推行所有人使用的部署过程，包括开发人员和运维人员。这些实践的目的是确保更高的部署质量。可以避免由于即兴部署和由此产生的配置不当而造成的错误。这个实践也与诊断和修复错误的时间有关。正式的部署过程应该是易于追踪特定部署工件的历史并且易于理解工件中包含的组件。
- 使用持续部署。与持续部署相关的实践是为了缩短从开发人员向代码库提交代码到代码被部署的时间。持续部署同时也强调自动化测试，以提高部署到生产环境的代码的质量。
- 在开发诸如部署脚本这样的基础设施代码时，也应遵守一套与应用程序代码相同的实践。适用于基础设施代码开发的实践，目的是不但确保部署的应用程序是高质量的，而且部署过程能够按照预定计划执行。部署脚本中的错误，例如配置不当，会导致应用程序、环境或部署过程出错。在开发运维脚本和过程时应当采用与正式软件开发中使用的相同的质量控制实践，这将有助于控制这些规格说明的质量。

图 1-1 是 DevOps 过程的概述。最基本的是，DevOps 提倡把运维人员作为首要干系人。准备一个发布是非常严肃、严谨的过程（我们将在 1.2.1 节中描述）。同样对于开发过程中系统会发生的运行时错误的类型，可能需要对运维人员进行培训。他们可能会对日志文件的类型和结构提出建议，可能在需求过程中给出其他类型的意见。极端情况下，DevOps 实践要求开发人员负责监控进度和在部署和执行过程中出现的错误，这样运维人员的声音就可能成为建议的需求。两者之间的实践包含了团队实践、构建过程、测试过程和部署过程的实践。我们将在第 5 章、第 6 章讨论持续部署流水线。我们也会在后面的章节讲述监控、安全与审计。

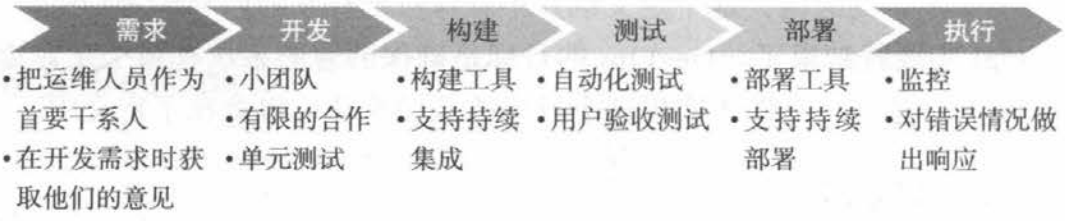


图 1-1 DevOps 生命周期过程 [标注法：波特价值链]

你可能对术语 IT 专业人士（IT Professional）、运维人员（Operator）、运维人员（Operation Personnel）的命名方式有些疑问。另一个相关术语是系统管理员（System Administrator）。IT 专业人士包含了上面提到的角色以及其他人，例如帮助台的支持人员。运维人员和系统管理员这两个术语的区别是历史原因造成的，但是今天已经不是这样了。历史上，运维人员直接接触硬件（安装、配置硬件、管理备份、维修打印机），而系统管理员负责计算机系统的正常运行时间、性能、资源和安全。今天很少有哪个运维人员不需要承担以前分配给系统管理员的职责。我们将使用术语运维人员来指代执行计算机操

作或系统管理的人（也可能两者都做）。

1.1.3　持续部署的例子：IMVU

IMVU 是一家社交娱乐公司，它的产品允许用户以一种 3D 阿凡达式的体验互相连接起来。本节内容改编自一位 IMVU 工程师所写的博客。

> IMVU 采用了持续集成。开发人员尽早提交并经常提交。每次提交都触发测试套件的执行。IMVU 有上千个测试文件，分布在三四十台机器上，测试套件的执行大约需要 9 分钟。在提交的内容通过了所有的测试之后，就自动实施部署。这大约需要 6 分钟。代码移动到集群中的数百台机器上，但是在开始时代码只在一小部分机器上执行（金丝雀）。一个取样程序检查金丝雀的结果，如果回归的数量很多，则改动的版本自动回滚。否则集群中的其他机器就会自动启用。IMVU 平均每天部署 50 次代码。

这个过程的核心是测试套件。每当提交的内容通过执行测试套件并回滚后，都会产生一个捕获出错部署的新测试并将它添加到测试套件中。

注意，对于一个大规模的系统来说，只要 9 分钟就能执行一次完整测试（有信心在生产环境中部署）的情形并不常见。在很多组织中，获得生产环境信心的完整测试套件需要执行数个小时，且常常是在夜间执行的。一个常见的挑战是审慎地缩减测试套件的大小，去掉“不可靠”（flaky）测试。

1.2　为什么是 DevOps

从很多方面来讲，DevOps 是对缓慢发布的问题做出的响应。发布投放到市场的时间越长，从中获得的功能或质量提升的收益就越低。理想情况下，我们希望持续发布。常常用术语**持续交付**或**持续部署**来表示。我们在第 5 章和第 6 章讨论这两个术语的细微差别。在本书中，我们使用术语**持续部署**，或**部署**。我们开始时先描述一个正式发布过程，然后深入探讨缓慢发布的一些原因。

1.2.1　发布过程

向客户发布新系统或现存系统的新版本是软件开发周期中最敏感的步骤。不论系统或版本是对外发布、直接由客户使用还是严格限定于内部使用，都是这种情况。只要系统有多于一个人使用，发布新版本就可能招致不兼容或失败，接下来就是客户的不满。

因此，各个组织在定义发布计划的过程上投入了大量精力。下面的发布计划步骤改编自维基百科。传统上，大部分步骤都是手工完成的。

1）与客户 / 干系人一起定义发布及部署计划并达成一致意见。可以在团队或组织层

面完成。发布或部署计划应该包括那些纳入新发布的功能，并确保运维人员（包括帮助台及支持人员）知晓日程安排，资源需求得到满足，并安排所需的各种附加培训。

2）确保每个发布包包含的一组相关资产与服务组件之间是相互兼容的。随着时间的推移，包括库、平台和依赖服务在内的各种东西都会发生变化。变化可能会引入不兼容。这个步骤是为了防止不兼容性在部署之后才显现出来。第 5 章讨论确保这些兼容性的方法。管理依赖关系是贯穿全书反复出现的一个主题。

3）确保在整个转换活动期间都能保持发布包及组件的完整性并在配置管理系统中准确记录。这个步骤包括两部分：首先，确保组件的旧版本不会无意间被纳入发布中；其次，确保保留此次部署的组件的记录。知道此次部署了哪些要素对于部署后追踪发现的错误是很重要的。我们将在第 6 章讨论部署的细节。

4）确保所有发布包和部署包都能追踪、安装、测试、验证，以及卸载或者在需要时能够回滚。部署的内容在很多情况下可能需要回滚（卸载新版本、重新部署旧版本），例如代码中存在错误、资源不足、许可证或证书过期。

这个清单中列举的活动可以以不同程度的自动化方式实现。如果所有这些活动都主要是通过人工配合完成，那么这些步骤很费力、耗时并且容易出错。任何自动化活动都反映了团队或组织层面对发布过程的一致性理解。因为工具一般都不止一次地使用，所以将对发布过程的一致性理解嵌入工具中会使得这种一致性在多个发布中都是持久维系的。

正确的发布是一件严肃的事情，如果你不想重视它，可以想想这几个媒体报道过的引起了重大财务损失的事件。

- 2012 年 8 月 1 日，骑士资本集团（Knight Capital）升级失败，造成 4.4 亿美元的损失。
- 2013 年 8 月 20 日，高盛集团（Goldman Sachs）升级失败，潜在地可能造成数百万美元的损失。

因为升级失败而导致宕机或故障的例子有很多，这只是其中的两个。在一个组织中，正确的部署升级是大型且重要的活动，并且这种活动应该快速完成，并将出错概率降到最低。有几个组织调查了部署的问题。我们报告其中的两个：

- XebiaLabs 是一个销售部署工具和持续集成工具的组织。在他们 2013 年做的一项调查中，有 130 多位被调查者做出了回应。这些被调查者有 34% 来自 IT 服务公司，健康、财务服务和电信公司的各占 10%。7.5% 的被调查者报告说他们的部署过程“不可信赖”，57.5% 的人报告说他们的部署过程“需要改进”。49% 的人报告说在部署过程中最大的挑战是“在不同环境和应用程序中不一致的地方太多”。32.5% 的人报告说“错误太多”。29.2% 的人报告说他们的部署依赖于定制的脚本，35.8% 的人报告说他们的部署是部分脚本、部分手工的。
- CA 科技公司为客户提供 IT 管理解决方案。他们在 2013 年做了一项调查，有

1300 人做出了回应，这些人所在公司的收入超过 1 亿美元。在那些说是看到了采用 DevOps 带来的收益的人中，53% 的人说他们已经看到软件和服务的部署频率增加，41% 的人说他们预计部署频率会增加。42% 的被调查者说他们看到部署应用程序的质量提升了，49% 的人说他们预计质量会提升。

推广自动化部署是发起这两个调查的组织的利益所在，即使这样，他们也清楚地说明了不同市场的很多公司都在关注部署速度和质量。

1.2.2　配合不佳的原因

考虑在开发团队完成系统的所有编码和测试之后发生的事情。系统需要放入满足以下条件的生产环境中：

- 只有适当的人才能访问它。
- 系统与生产环境中交互的所有其他系统都兼容。
- 有足够的资源满足系统的运行。
- 用于运行的数据是最新的。
- 系统生成的数据可以由生产环境中的其他系统使用。

此外，帮助台支持人员需要就新系统的功能接受培训，运维人员需要就如何排除系统运行中出现的问题而接受培训。发布时机可能也很关键，因为发布时运维人员的关键人员不能缺席，时间上也不能和需要使用现有资源的新的促销活动重合。

这些都不是在不经意间发生的，每一项工作都需要开发人员和运维人员之间的配合。如果开发人员和运维人员不就这些项目中的一个或多个进行沟通，不难想象会出现什么情况。开发人员的一种常见的态度是“做完了开发，程序可以跑起来了”。我们在讨论采用 DevOps 的文化障碍时会探讨这种态度产生的原因。

组织中都有一套过程确保发布过程顺利，其中一个原因是协作配合并不总能以适当的方式进行。这种不满是推动 DevOps 发展的一个驱动力。

1.2.3　运维人员能力有限

运维人员要做很多工作，但是他们能够完成什么或者他们当中谁对什么系统比较熟悉，这方面是存在不足的。考虑一下像维基百科中详细描述的现代运维人员的职责。

- 分析系统日志，识别计算机系统潜在的问题。
- 在现有的数据中心环境中引入并集成新技术。
- 执行系统及软件的例行审计。
- 执行备份。
- 对操作系统进行更新、补丁升级和配置更改。
- 安装并配置新的硬件和软件。
- 增加、删除或更新用户账号信息；重置密码等。

- 回答技术问题并帮助用户。
- 确保安全。
- 用文档记录系统的配置信息。
- 解决所有报告的问题。
- 优化系统性能。
- 确保网络基础设施启动和运行。
- 配置、增加及删除文件系统。
- 学习卷管理工具的知识，如 Veritas（即现在的 Symantec）、Solaris ZFS、LVM 等。

每一项工作都需要深入理解。难怪我们在问一个因特网公司的 IT 总裁他最大的问题是什么时，他回答说“找到并留住合格的人”。

DevOps 运动采用一种不同的方法。它们的方法把以前由运维人员完成的很多任务都自动化了，并让开发人员承担部分剩余的工作，通过这种方法，可以减少对专业运维人员的需求。

1.3 DevOps 视角

一个重大的呼吁是缩短新功能推向市场的时间，减少部署过程中发生的错误，考虑到我们讨论的问题并且这些问题都存在已久，这种呼吁也就不足为奇了。DevOps 有很多形式，是对现有实践不同程度的改编，但是各种不同形式都始终贯穿着两个主题：自动化及开发团队的职责。

1.3.1 自动化

图 1-1 显示了各种生命周期过程。从构建、测试到执行的步骤在某种程度上都可以实现自动化。我们将在适当的章节中讨论这些步骤中的每一步使用的工具，这里首先强调自动化的优点。在 1.7 节中讨论依赖于自动化的一些问题。

工具可以执行这个过程中每一步所需的操作，针对生产环境或者一些外部规格说明检查操作的有效性，把过程中发生的错误通知适当的人，并且保留操作历史，以用于质量控制、报告和审计等目的。

工具和脚本也可以强制推行组织层面的策略。假设组织有一项策略是，每个变更都要有与之相关的合理理由。那么在提交变更前，工具或脚本可以要求做变更的人提供理由。当然这个要求也可能被设法规避，但是让工具要求一个理由，将会增加对这个策略的符合度。

在工具成为一组过程的中心后，就必须对这些工具的使用进行管理。比如说，工具可以从脚本、配置变更或运维的终端调用。如果是复杂的终端命令，即使使用的只是几

个命令，也建议把命令的使用脚本化。工具可以通过规格说明文件进行控制，例如 Chef cookbooks 或 Amazon CloudFormation——今后还会有更多。脚本、配置文件和规格说明文件必须遵从与应用程序代码本身一样的质量控制。脚本和文件也应该进行版本控制，也应该进行错误检查。这个术语常常称为“基础设施即代码”(infrastructure-as-code)。

1.3.2　开发团队的职责

自动化将减少错误的发生，缩短部署时间。为了进一步缩短部署时间，考虑前面详细描述的运维人员的职责。如果开发团队接受 DevOps 的职责，即开发团队交付、支持并维护服务，那么因为所有所需的知识都保留在开发团队，所以向运维团队和支持人员转交知识也就少了。不需要转移知识也就省掉了部署过程中的大量协作步骤。

1.4　DevOps 与敏捷

DevOps 的一个特征是强调 DevOps 实践与敏捷实践的关系。本节将把 DevOps 实践覆盖在 IBM 规范敏捷交付上。我们的重点不是解释规范敏捷交付，而是说明 DevOps 中增加的内容。如果想了解规范敏捷交付，可参阅《Disciplined Agile Delivery: A Practitioner's Approach》一书。如图 1-2 所示，规范敏捷交付包含 3 个阶段——开始、构造和转换。在 DevOps 的语境中，我们把转换解释为部署。

DevOps 实践影响所有这三个阶段。

1）*开始阶段*。在开始阶段中，完成发布计划和初始需求规格说明。

a. 运维人员考虑的内容将为开发人员增加一些需求。本书后面会介绍更多细节，其中的两个需求是维护发布向后兼容性以及软件可切换的特性。运行日志信息的形式和内容影响运维人员排查问题的能力。

b. 发布计划包含了为功能定优先级，它也包含了与运维人员之间的配合，包括发布调度、确定运维人员为了支持新发布需要做哪些培训。发布计划也包含了确保与生产环境中的其他软件包的兼容性以及发布失败时的恢复计划。DevOps 实践使得在发布计划中与协作相关的很多主题不再需要，而其他方面变得高度自动化。

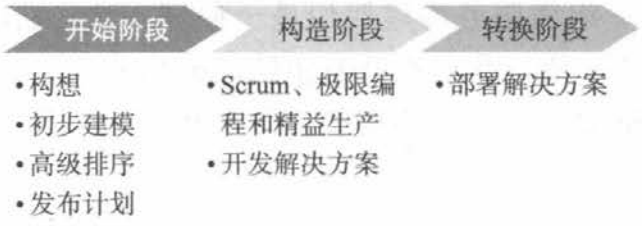


图 1-2　每个发布的规范敏捷交付阶段（改编自 Ambler 和 Lines 编著的《Disciplined Agile Delivery: A Practitioner's Guide》一书）[标注法：波特价值链]

2）*构造阶段*。在构造阶段中，DevOps 实践的关键要素是对代码分支的管理、持续

集成和持续部署的使用、自动化测试的测试用例的合并。这些也是敏捷实践，但却是实现自动化部署流水线的一个重要组成部分。一个新的元素是在构建与转换活动之间，集成和自动连接的。

3）*转换阶段*。在转换阶段中，对解决方案进行部署，并由开发团队负责部署、监控部署的过程，确定是否需要回滚以及何时回滚，并监控部署后的运行。开发团队有一个“可靠性工程师”的角色，他在部署和后续运行过程中负责监控及排查问题。

1.5 团队结构

本节讨论在承担 DevOps 职责的开发团队中，团队一般是多大规模，包含哪些角色。

1.5.1 团队规模

虽然不同的方法论推荐的准确团队规模是不一样的，但是大家都认为团队的规模应该相对较小。亚马逊（Amazon）有一个“两个比萨饼的规则”。即，团队规模应该是两个比萨饼就够吃。虽然这个规则本身有一点模糊（比萨饼有多大，团队成员有多饿），但是这个规则的意图很明确。

小团队的优势在于：

- 可以快速做出决定。在每次会议上参会者都希望表达自己的意见。参会者越少，表达的意见就越少，用来听取不同意见的时间也就越少。这样，与大团队相比，表达意见、统一意见的速度就快。
- 人少比人多更容易组成一个关系密切的群体。关系密切的群体是每个人都相互理解，并遵循为他们制定的同一套目标。
- 人们在小团队面前比在大团队面前更容易表达意见。

小团队的不利之处是较大的任务是少数几个人无法完成的。在这种情况下，这个任务只能拆细，每一部分分给不同的团队，各部分需要高效地融合在一起，这样才能完成一个大任务。为了做到这一点，团队之间需要配合。

团队规模成为整体架构的一个主要驱动因素。一个小团队需要完成一小部分代码。我们将会看到，围绕着一组微服务构建架构是一种好方法，它能够把这些小任务打包在一起并减少显式配合——所以我们把开发团队的输出成果称为“服务”。我们将在第 4 章讨论迁移到小团队驱动的微服务架构的方法及挑战，并在第 13 章中学习一个来自 Atlassian 的案例研究。

1.5.2 团队角色

我们从 Scott Ambler 对敏捷团队中角色的描述中挖掘两个团队角色。

团队领导。这个角色在 Scrum 中称为 Scrum Master，在其他方法中称为团队教练或项目领导。他负责协调团队、获取资源、保护团队不受问题的干扰。这个角色包含了项目管理的软技巧，但是不包含诸如制定计划、安排日程这样的技术能力。这些活动最好留给团队作为一个整体去完成。

团队成员。这个角色有时候称为开发人员或程序员，负责系统的创建和交付。包括建模、编程、测试和发布等其他活动。

团队中执行 DevOps 过程的其他角色还包括服务所有者、可靠性工程师、看门人和 DevOps 工程师。一个人可以承担多个角色，一个角色也可以由多人承担。角色的分配取决于个人能力、个人的工作负荷以及这个角色所需的能力和工作量。我们将在第 12 章的案例研究中讨论一些采用 DevOps 和持续部署的团队角色的例子。

1. 服务所有者

服务所有者在团队中负责外部协作。服务所有者参与系统范围的需求活动，安排团队工作内容优先级，向团队提供来自团队服务的客户信息和提供给团队的关于服务的信息。下一个迭代的需求收集和发布计划活动与当前迭代的概念阶段是可以并行的。这样，虽然这些活动需要协作与时间，但不会延缓交付时间。

服务所有者维护并与其他人交流服务的愿景。每个服务相对都较小，愿景包含了解团队服务所面向的客户以及团队服务所依赖的服务。即，愿景包括了整个系统的架构以及这个架构中每个团队的角色。

对服务所有者的一个关键要求是：既能与其他干系人交流，又能与团队的其他成员交流。

2. 可靠性工程师

可靠性工程师有多个职责。首先，可靠性工程师在部署完成之后立即就要对服务器进行监控。这可能包括使用金丝雀测试集（对最少节点的现场测试）和取自服务的各种不同的指标。本书后面将更详细地讨论这两个概念。其次，可靠性工程师是服务执行期间出现问题后的联系人。这意味着对于高可用性服务，他需要随时待命。在 Google，这个角色称为“现场可靠性工程师”。

在出现问题后，可靠性工程师执行短时间分析以诊断、缓解及修复问题，通常会借助于自动化工具。这可能是在压力非常大的情况下做的（例如，在深夜或是浪漫的晚餐时间）。问题还可能涉及其他团队的可靠性工程师。不论何种情况，可靠性工程师都必须出色地排查故障、诊断问题。可靠性工程师还必须深入理解服务的内部，这样才能修复故障或者采用临时解决方案。

除了短时间分析之外，可靠性工程师还应当发现或者与团队一起发现问题的根本原因。“5 个为什么”（5 Whys）是一个确定根本原因的技巧。不停地问为什么，直到发现原因。例如，部署的服务很慢，直接原因是负载量意外激增。第二个“为什么”是什么因

素造成了意外激增，以此类推。最终，答案是服务的压力测试未包含适当的负载量特性。这个原因可以通过提高压力测试的负载特性而修复。可靠性工程师需要逐步成长为有能力的开发人员，因为他们需要编写高质量的程序并且以自动化的方式实现不断重复发生的诊断、迁移和修复工作。

3. 看门人

Netflix 使用图 1-3 所示的从本地开发到部署阶段的步骤。

图中每个箭头都表示进入下一步的决策。这个决策可以自动实现（Netflix 的做法），也可以手动实现。在部署流水线中，决定将服务进入下一步的手动角色称为看门人角色。看门人决定服务的某个版本或服务的某个部分是否能够通过“门”并进入下一步。看门人可以依赖于详尽的测试结果，使用一个检查表做出决策，也可以与其他人商量，但是代码或服务是否允许进入部署流水线的下一步，这个职责最终落在看门人身上。有时候，在部署到生产环境之前，原来的开发人员成了看门人，根据得到的测试结果做出决定，但是对此负全部责任。在金融行业中，按照监管要求，可能需要有看门人（但不是原来的开发人员）。

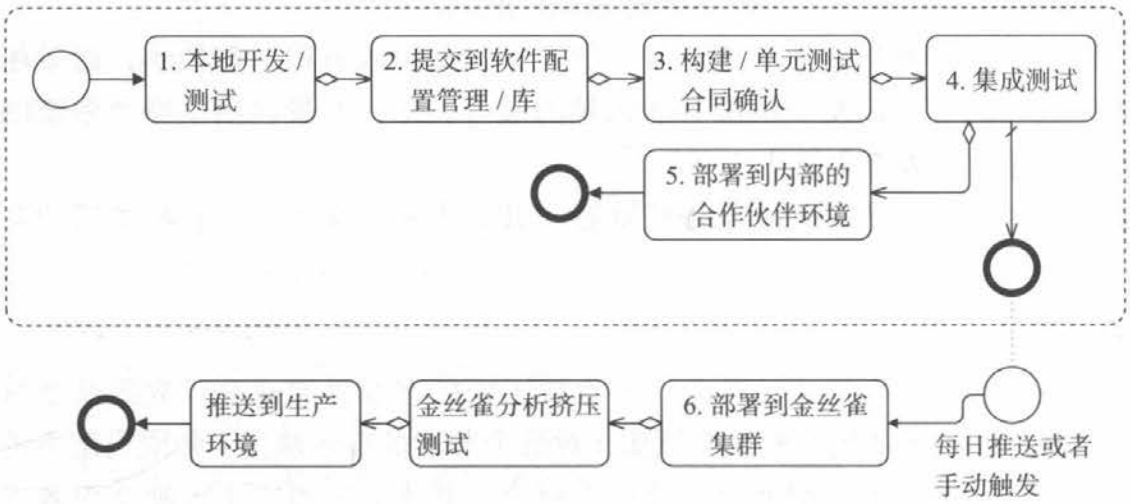


图 1-3 Netflix 部署到生产环境的路径（改编自 http://techblog.netflix.com/2013/11/preparing-netflix-apf-for-deployment.html）[标注法：业务流程建模标注]

Mozilla 有一个叫作*发布协调人*的角色（有时称为*发布管理员*）。这个人要承担协调整个发布的职责。这个发布协调人参加决定发布包含或不包含哪些内容的仲裁会议，了解发布中包含的所有工作的背景和环境，对缺陷严重程度定级过程中存在的争议进行仲裁，可以批准最新添加的内容，可以做出取消的决定。此外，到了最后的发布日，发布协调人是开发人员、质量保证人员（QA）、发布工程师、网站开发人员、公共关系人员和市场人员之间的所有交流的汇聚点。发布协调人就是一个看门人。

4. DevOps 工程师

再次检查图 1-2，看看这个过程中工具的使用。有些工具是代码测试工具、配置管

理工具、持续集成工具、部署工具或部署后的测试工具。

配置管理不仅适用于服务的源代码，而且适用于各种工具的输入信息。这可以让你了解诸如“在上一次部署和这次部署之间有什么变化？”“上一次构建之后增加了什么新的测试？”之类的问题。

工具在进化，工具需要专业知识，工具需要专门的信息输入。DevOps 工程师的角色负责 DevOps 工具链中使用的各种工具的护理和保养。这个角色可以是个人、团队或组织层面。例如，组织可能决定所有人都应该使用某个特定的配置管理工具。团队仍旧需要确定分支策略，每个开发人员可能需要进一步创建分支。需要有关于命名及访问的策略，并且可能是自动强制实施的。选择开发团队使用的配置管理工具的哪个发布是 DevOps 工程师职责的一部分，为开发团队定制工具并监控开发人员是否正确使用也是 DevOps 工程师的职责。以自动化方式实现开发和部署流水线是 DevOps 工程角色的固有职责。这个角色必须存在并有人填补，至于这个角色在组织或团队结构中如何体现，是另一个问题。

1.6　协作

DevOps 的一个目标是最大程度减少协作，以缩短推向市场的时间。需要协作的两个原因是，首先，不同团队开发的各部分能够在一起工作；其次，避免重复工作。《Oxford English Dictionary》（牛津英语词典）对协作的定义是：对复杂个体或活动的不同元素进行组织，以便能够有效地一起工作。本节深入讨论协作的概念与机制。

1.6.1　协作的形式

协作机制有不同的属性。

- 直接的——需要协作的人彼此认识（例如，团队成员）。
- 间接的——协作机制的受众只能根据其特征识别（例如，系统管理员）。
- 持久的——在协作结束后，协作工件仍旧可以得到（例如，文档、电子邮件、公告牌）。
- 短暂的——按照字面意思，这种协作不产生工件（例如，面对面的会议、交谈、电话 / 视频会议），通过手工记录或机器记录仪，短暂协作可以转化为持久协作。
- 同步的——每个人是实时协作的（例如，面对面的）。
- 异步的——每个人不是实时协作的（例如，文档、电子邮件）。

协作机制构建到 DevOps 使用的很多工具中。例如，版本控制系统就是一种自动化协作形式，防止开发人员覆盖其他人的代码。持续集成工具是测试构建的版本是否正确的一种协作形式。

每一种形式的协作都有各自的成本和收益。同步协作需要安排日程，潜在地需要出

差。花在同步协作上的时间对每个人来说都是成本。同步协作的收益包括：参与其中的人马上就能对问题的解决做出贡献。同步协作的其他成本和收益取决于通信带宽、时差、协作的持久性。每一种协作形式都可以根据成本和收益进行分析。

理想的协作机制的特点是，从是否有延迟、需要做的准备工作以及人们的时间这几方面讲，是低成本的，从所有相关干系人的协作的可视性、快速解决问题、有效交流所需信息这几方面讲，是高收益的。

前面提到的维基百科对 DevOps 的定义说明了 DevOps 过程的特征是，“交流、协作与融合”。根据我们对协作的讨论，可以看到有太多的人工交流与协作，特别是同步协作，让 DevOps 的快速推向市场的目标无法实现。

1.6.2 团队协作

团队协作机制有两种类型——人工过程和自动过程。DevOps 人工过程取自敏捷过程，目的是用于持久性不高的高带宽协作。站立会议和信息发射源是人工过程协作机制的两个例子。

自动化团队协作机制的目的是保护团队成员不受自身或其他活动的干扰（版本控制和配置管理系统），用自动化方式完成重复单调的任务（持续集成和部署），加快发现并报告错误的速度（自动化单元测试、集成测试、验收测试和现场生产测试）。目标之一是尽可能快地向开发人员提供反馈。

1.6.3 跨团队协作

再次检查发布过程活动可以清楚地看到跨团队协作是一个最耗时的因素。必须在客户、干系人和其他开发团队以及运维人员之间协作。因此，DevOps 过程试图尽量减少这种协作。从开发团队的角度看，有 3 种类型的跨团队协作：与干系人和客户的上游协作；与运维人员的下游协作；与其他开发团队的交叉流协作。

服务所有者的角色是进行上游协作。下游协作是通过把很多运维职责转移到开发团队来完成的。我们现在重点关注跨团队协作。开发团队之间需要协作的原因有两个——一是确保一个团队开发的代码能够与其他团队开发的代码很好地一起工作；二是避免重复工作。

1）*让程序段能够融合在一起工作*。支持不同的开发团队独立工作并简化这些工作的集成过程的方法之一是有一个软件架构。正在开发中的系统，它的架构将有助于各部分融合在一起工作。其他协作也仍旧是需要的，但是架构发挥了协作机制的作用。架构说明了创建整个系统的一些设计决策。其中的 6 个设计决策为：

a. *职责分配*。在 DevOps 过程中，在架构中说明概要职责，但具体职责是在每个迭代开始时确定的。

b. *协作模型*。协作模型描述了架构的不同组件在运行时是如何协作的。所有元素都

使用一个协作模型，这样在协作模型之间就不需要再协作了。

c. 数据模型。与职责分配一样，数据模型对象及其生命周期是在架构中说明的，但是可以在迭代开始后再细化。

d. 资源管理。将要管理的资源是由架构决定的。对这些资源的限制（例如，缓冲区大小或线程池大小）可以在迭代开始时或者通过在架构中说明的系统范围的策略来决定。

e. 架构元素之间的映射。如果在架构以及分配给团队的工作中说明了这些映射关系，团队之间需要最少的协作。在第 4 章中，在讨论使用 DevOps 过程开发的系统的架构风格时，我们会再次回到这个主题。

f. 绑定时决策。这些是在整体架构中指定的。很多运行时绑定值（running binding value）都将通过配置参数指定，我们将在第 5 章讨论配置参数的管理。

2）避免重复工作。避免重复工作并鼓励复用，是不同开发团队之间协作的另一个理由。DevOps 实践从根本上认为，为了缩短投向市场的时间，重复工作是必需付出的成本。有两个理由。首先，每个团队必须完成的任务都很小，重复工作量也很小。潜在的大面积的重复，例如每个团队都创建自己的数据存储，可以通过架构解决。其次，每个团队都负责自己的服务，在部署后如果自己团队编写的代码出现了问题，那么排除障碍较快，并且避免了把问题升级到其他团队。

1.7　障碍

如果 DevOps 解决了开发中长期存在的问题并有如此明显的好处，为什么不是所有的组织都采用 DevOps 实践呢？本节将探讨采用 DevOps 时遇到的障碍。

1.7.1　文化及组织类型

在讨论 DevOps 时，文化很重要。在跨组织以及同一个组织中的不同群体之间，与 DevOps 相关的文化问题都会影响它的形式与采用。文化不仅取决于你的角色，而且取决于你所在组织的类型。

DevOps 的目标之一是缩短新功能或新产品投向市场的时间。在采用 DevOps 实践时，组织要考虑的一个问题是缩短面市时间所带来的收益与一些事情出现差错之间的权衡。几乎所有的组织都对风险感到担忧。但是，一个特定组织所担忧的风险取决于他们的活动领域。对一些组织来说，出现问题所带来的风险比尽快推向市场更重要。

- 在监管领域运作的组织（金融、医疗保健、公用事业服务）有他们必须遵守的规则，如果违反了运营规则，就会面临处罚，有可能还是很严重的处罚。即使处于监管领域的组织也可能有不受监管的产品。这样，一个金融组织也可以在某些产品上使用 DevOps 过程。对于需要更多监管的产品，这些实践可以修改。例如引入更多的看门人。我们将在第 8 章讨论安全及审计问题。

- 在成熟、行动缓慢的领域中运作的组织（汽车或建筑行业）有很长的前置时间。并且，虽然他们的截止日期是切切实实的，但他们很早就能预见到。
- 某些组织的客户在切换到其他供应商（supplier）时成本很高，例如企业资源计划系统，客户因为担心运行的稳定性而不愿意冒险。某些系统宕机时间的成本远远高于更快速引入一个新功能所能得到的竞争优势。

对于其他组织来说，灵活快速的响应比因为快速移动而偶尔产生的错误重要得多。

- 依赖于业务分析决定产品方向的组织希望在收集数据并根据数据采取行动之间的时间越来越短。因为下一个周期很快到来，出现的错误能够很快纠正。
- 面临激烈竞争压力的组织需要赶在竞争对手的前面把产品和新功能投向市场。

注意这些例子并不取决于组织的规模，而更取决于他们的业务类型。如果监管者监督你们并且能够强行规定你们的运营规范，或者产品特性的前置时间是好几年，或者你们的资本设备的预期使用期限是 40 年，那么你们很难灵活起来。

这里讨论的要点是企业有自己的运营环境，并会继承这个环境中的很多文化。第 10 章将会详细描述。有些 DevOps 实践是破坏性的，例如允许开发人员直接部署产品。另一些 DevOps 实践是增值的，它们不会影响产品整体的工作流或监管。把运维人员作为首要干系人的做法应该落入这个非破坏性的分类中。

一个行动缓慢的组织可能变得更加灵活，一个灵活的组织可能面临监管。如果你考虑采用 DevOps 实践，需要清楚 3 件事。

1）*在你考虑的实践中隐含地包含了其他什么实践？*不做持续集成就无法实施持续部署。独立的实践需要优先采用，先于有依赖关系的实践。

2）*你考虑的特定实践是什么？*它的假定条件、成本和收益是什么？

3）*企业文化是什么，采用这个特定的 DevOps 实践的后果什么？*如果这个实践只影响运营与开发，倒还好说。但如果需要修改整个组织结构和监管办法，那可以说是另一回事了。采用一种实践的困难之处是，它影响组织中的其他部分。但即使是一个开发团队和多个运维人员采用 DevOps，让所有涉及的人都接受 DevOps 文化也是很重要的。采用 DevOps 的一种常见的失败情况是，你雇用了一个 DevOps 工程师，然后就认为万事大吉了。

1.7.2 部门类型

确定一个组织的文化的方法之一是看它的激励产生什么样的结果。依靠销售额提成的销售人员会很努力地推销。根据季度利润获取报酬的 CEO 会关注下一个季度的结果。这是人之本性。对开发人员的激励是生产并发布代码。理想情况下，他们受到的激励是生产没有错误的代码。但是有一部 Dilbert 卡通片显示了这种做法的困难之处。尖脑袋老板对每个发现并修复的缺陷都支付 10 美元，Wally 的回答是："好哇，今天下午我就可以为自己写出一部小客车来。"不管怎么说，开发人员得到的激励应该是让代码进入生产

环境。

另一方面，对运维人员的激励是把宕机时间降至最低。这意味着要去检查并排除宕机的原因。仔细检查是需要花费时间的。而且，避免变更也是减少宕机的一个原因。“没有问题就不要修复”是一个几十年来广为人知的说法。

一般来说，开发人员受到的激励是做出变更（发布新代码），运维人员受到的激励是不要变更。这两种不同的激励机制培育出不同的态度，可能成为文化冲突的原因。

1.7.3　筒仓思维方式（Silo Mentality）

组织中的两个部门有一个共同目标——确保组织取得成功。这句话说起来容易，实际做起来要难得多。一般来说一个人对自己的团队最忠诚，其次才是整个组织。如果开发团队负责制定发布计划，该计划说明将实现哪些功能、以何种优先级实现，那么组织中的其他部门就会感到自己的权力被侵犯了，客户也可能会不满意。如果以前由运维人员完成的工作现在由开发人员完成了，那么运维人员的工作变少了之后会怎么样呢？

这种政治在一个组织中早已司空见惯，但重要性并不因此而降低，也不能因此就视而不见。

1.7.4　工具支持

我们前面描述了自动化过程的优势。这些优势是实实在在的，但也不是没有成本的。

- 每个工具的安装、配置和使用都必须具备专业技能。工具有新发布、新信息和新特征。使用工具的技能经验必须整合到组织中。
- 如果一个组织中的很多不同的开发团队使用同一套过程，那就必须有一个方法定义这些共同使用的过程，并确保所有的开发团队都遵守。使用工具意味着需要遵守隐含在工具中的策略。关于如何定义共同使用的过程，可参见第 12 章中的一个例子。

1.7.5　人员问题

按照 Datamation 2012 IT 薪水指南，一个软件工程师比一个系统管理员的薪水高 50%。把系统管理员（运维）的任务转移给软件工程师（开发），完成任务的人员成本就增加了 50%。这样，为了让完成任务的成本保持一致，用于完成任务的时间就必须削减 1/3。为了真正赢得时间，还需要进一步削减，而自动化是节省这些时间的最常见方法。为了确定采用哪些 DevOps 过程以及如何采用，这是一个组织必须仔细研究的一类成本 / 收益分析。

具备现代技能集的开发人员供不应求，他们的工作量也很重。在他们的工作中增加更多的任务会加剧开发人员的短缺。

1.8 小结

本章的主要知识点是：人们从不同的视角定义 DevOps。例如，运维人员采用敏捷实践，开发人员承担运维责任，以及其他一些视角的定义。但共同目标是缩短一个功能或改进点从业务思路构想到最终部署给用户的时间。

由于文化及技术上的挑战，DevOps 面临着障碍。它可能对团队架构、软件架构、运维的传统方式形成巨大的冲击。我们列出了一些常见的实践，让你对这种冲击有初步的了解。我们将在本书剩余部分详细讲述这些主题。

DevOps 涉及的一些权衡如下：

- 现在需要支持 DevOps 工具了。在对工具进行支持与缩短新功能投向市场的时间之间进行权衡。
- 把职责从 IT 人员转向开发人员。这种权衡是多方面的，下面是需要考虑的一些方面：
  - 两个团队完成同一个任务的成本。
  - 两个团队完成同一个任务的时间。
  - 两个团队中的人员是否可以投入工作中。
  - 在运行期间发现错误时需要的修复时间。如果部署后很快就发现了错误，那么开发人员还有可以迅速诊断问题所需的背景信息，而如果错误一开始是由 IT 人员诊断的，在把错误交给开发人员处理之前可能需要花费一些时间。
- 去掉对新功能和部署的监管。这是在开发团队自治与整体协作之间的权衡。开发团队自治的效率必须超出因为没有整体监管而造成的工作量的重复。

总之，我们相信 DevOps 有潜力带领 IT 走向激动人心的新领域，带来更高频率的创新、更快的循环周期，以此提升用户体验。我们希望你喜欢阅读本书，就像我们喜欢写作本书一样。

1.9 更多阅读材料

通过下列资源可以阅读有关 DevOps 的不同定义：

- Gartner 技术成熟度曲线 [Gartner] 把 DevOps 归为处于上升期：http://www.gartner.com/DisplayDocument?doc_cd=249070。
- AgileAdmins 从敏捷角度解释了 DevOps：http://theagileadmin.com/what-is-devops/。

从下面列出的最近调查及业界报告中，可以找到更多的内容：

- XebiaLabs 对与 DevOps 相关的主题做了广泛调查并有大量的业界报告，可以在如下地址找到：http://xebialabs.com/xl-resources/whitepapers/。

- CA 科技公司的报告让我们能够了解企业对 DevOps 的不同理解，报告可以在如下地址找到：http://www.ca.com/us/collateral/white-papers/na/techinsights-report-what-smart-businessesknow-about-devops.aspx。

虽然有些厂商或社区把持续集成工具向持续部署的方向扩展，但也有很多厂商为持续交付和部署发布了全新的工具。

- 流行的持续集成工具 Jenkins 有很多第三方插件，包括一些扩展到持续集成的工作流。可以在 http://www.slideshare.net/cloudbees 找到一些由 Cloudbees 开发的插件。
- IBM 最近收购了 UrbanCode。它是一个提供持续交付工具套件的新厂商 [InfoQ 13]。
- ThoughtWorks 也发布了名为“Go”的持续部署流水线套件，可以在 http://www.go.cd/ 找到。

本章提到的一些基本概念信息来自下面的维基百科链接：

- 我们引用的一个 DevOps 定义可以在 http://en.wikipedia.org/wiki/System_administrator 找到。
- 发布及部署计划的步骤改编自 http://en.wikipedia.org/wiki/Deployment_Plan。
- 运维人员的职责在 http://en.wikipedia.org/wiki/DevOps 中。
- 5 个为什么起源于丰田汽车公司，在 http://en.wikipedia.org/wiki/5_Whys 上进行了讨论。

关于持续部署是否只是一个梦想也有一些讨论（[BostInno 11]）。Scott Ambler 不仅参与编写了一本关于规范敏捷交付的书（与 Mark Lines 合著）[Ambler 12]，并且还维护着一个博客，我们根据这个博客改编了团队角色的定义 [Ambler 15]。

Netflix 维护着一个技术博客，在上面讨论各种与平台相关的话题。他们的部署步骤在 [Netflix 13] 中进行了讨论。

Mozilla 的发布协调人角色在 [Mozilla] 中进行了讨论。

Len Bass、Paul Clements 和 Rick Kazman 在《Software Architecture in Practice》[Bass 13] 中的第 73 页及后续章节讨论了架构决策。

对 IMVU 的讨论改编自 Timothy Fitz 所写的一篇博客 [Fitz 09]。

第2章 云即平台

我们对云计算的重新定义包含了我们已经做的一切。

……计算机行业是唯一一个比女性时装更容易受时尚驱动的行业了。

……我们将发布云计算公告，因为如果橙色成为新宠，我们就会制作橙色的罩衫。我可不会和这个对着干。

——Larry Ellison

2.1 概述

描述云时，常用电网进行类比。当要使用电时，你将设备插到标准电路上并打开开关。你要为所使用的电付费。大多数情况下，你不用关心各个电力公司是如何发电和传输电力的。但停电时，就无法对此保持无视。此时你会意识到电力的使用之下有着复杂的机制，即便你对导致停电的具体机制仍然一无所知。

美国国家技术与标准研究所（National Institute of Standards and Technology，NIST）给出了一个关于云的描述，其使用了下面这些元素：

- 按需自助服务。用户可以单方面供给计算能力，如服务器时间和网络存储，而无需与服务供应商进行人工交互。
- 广泛的网络接入。能力通过网络提供并借助标准机制进行访问，该机制由异构的瘦或胖客户平台推广使用（例如，移动电话、平板电脑、笔记本和工作站）。
- 资源池化。服务供应商的计算资源被汇聚起来通过多租户模式提供给多个用户，不同的物理资源和虚拟资源按照用户的需要动态地进行分配和重新分配。还有位置独立性，因为客户通常无法控制或者也不知道所提供资源的确切位置，但能够在更高的抽象层次指定位置（如国家、省或者数据中心）。这些资源包括存储器、处理器、内存以及网络带宽。
- 快速弹性。能力能够弹性地提供和释放，在一些情况下是自动的，使规模快速地向外和向内扩展以适应需求。对用户而言，能够供给的能力常常似乎是无限的，并且可以在任何时间分配任意数量。
- 可计量的服务。云系统能够利用计量能力来自动控制和优化资源的使用，该计量能力依据服务的类型进行了某种程度的抽象。资源使用能够被监控、控制，并可以被报告，因此，为服务的供应商和用户提供了透明度。

从运维和 DevOps 的角度，这些特性中最为重要的是按需自助服务和可计量的服务。即使云提供了似乎能够任意取用的无限资源，但你仍旧要为使用付费。如我们将要讨论的，其他特性也很重要，但不像按需自助服务和按使用付费这样占主导地位。

美国国家技术与标准研究所的特性描述隐含了云服务的供应商和用户之间的区别。本书的视角主要是云服务的用户。如果你的组织运营了自己的数据中心，那么这种区分可能会存在一些模糊之处，但即使在这类组织中，通常也不会认为数据中心的管理在 DevOps 的范畴中。

如表 2-1 所示，美国国家技术与标准研究所还对来自于云供应商的不同类型的服务进行了归类。美国国家技术与标准研究所定义了 3 种类型的服务，这些服务都能够应用于 DevOps。

- 软件即服务（Software as a Service，SaaS）。用户能够使用运行在云基础设施上的供应商应用。各种客户端设备通过像 Web 浏览器这样的瘦客户端接口（如，基于网页的电子邮件）或者应用接口来访问应用程序。除了有限的与用户相关的应用配置设置外，用户不用管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储，甚至单独的应用程序功能。

表 2-1 云服务模型

服务模型	示例
SaaS：软件即服务	电子邮件、在线游戏、客户关系管理、虚拟桌面等
PaaS：平台即服务	Web 服务器、数据库、应用服务器、开发工具等
IaaS：基础设施即服务	虚拟机、存储器、负载均衡器、网络等

- 平台即服务（Platform as a Service，PaaS）。用户能够向云基础设施中部署自建的或采购的应用程序，这些应用程序使用了供应商支持的编程语言、库、服务和工具来创建。用户不能管理或控制底层的云基础设施，包括网络、服务器、操作系统或存储器，但能够控制所部署的应用，并可能对应用托管环境进行配置设置。
- 基础设施即服务（Infrastructure as a Service，IaaS）。用户能够配置处理器、存储器、网络以及其他基础计算资源，这里用户能够部署和运行任意软件，可能包括操作系统和应用程序。用户不能管理和控制底层的云基础设施，但能够控制操作系统、存储器，以及部署的应用，并可能对网络模块的选择有一些控制（如主机防火墙）。

我们首先探讨了云涉及的机制，接下来我们探讨这些机制对 DevOps 的影响。

2.2 云的特性

云最根本的推动者是构筑在成千上万通过因特网访问的主机之上的虚拟化技术。我

们首先探讨以 IaaS 为中心的特性，即虚拟化和 IP 管理，接着是 PaaS 提供的一些特性。然后，我们探讨一些普遍的问题，例如数以万计的主机所带来的后果，以及云是如何支持弹性的。

2.2.1 虚拟化

在云计算中，虚拟机（Virtual Machine，VM）是物理机的模拟。一个虚拟机镜像就是一个文件，其中包含了可引导的操作系统和在其上安装的软件。虚拟机镜像提供了启动虚拟机（或者更准确一些，虚拟机实例）所需的信息。本书使用“虚拟机”和“虚拟机实例”表示一个虚拟机实例，并使用“虚拟机镜像”表示启动虚拟机或虚拟机实例的文件。例如，亚马逊机器镜像（Amazon Machine Image，AMI）是虚拟机镜像，它能够用来启动弹性计算云（Elastic Compute Cloud，EC2）的虚拟机实例。

使用 IaaS 时，用户通过云供应商提供的应用程序编程接口（Application Programming Interface，API）从虚拟机镜像获取一个虚拟机实例。应用程序编程接口可能嵌入在命令行解释器、Web 界面，或其他某种工具中。无论如何，它请求带有一组资源的虚拟机——CPU、内存和网络。获得的资源可能被托管在一台计算机上，而该计算机可能还托管其他虚拟机（多租户），但从用户的角度，供应商创建一个相当于独立的计算机。

1. 创建虚拟机

要创建一个虚拟机，需要执行两个不同的活动。

- 用户发出一个命令来创建虚拟机。通常，云供应商有创建虚拟机的实用程序。需要将虚拟机所需的资源、虚拟机计费时对应的账户、要装载的软件（见下文），以及指定虚拟机安全性和外部连接的一组配置参数告知该实用程序。
- 云基础设施确定在哪个物理机上创建虚拟机实例。该物理机的操作系统称为*虚拟机监视器*（hypervisor），它为新虚拟机分配资源并将新机器“联网”，以便它能够发送和接收消息。给新虚拟机分配一个 IP 地址——该地址将用来发送和接收消息。我们已经描述了虚拟机监视器在裸机上的运行情况。有可能还会有其他操作系统类软件的层，但每一层都会引入额外的开销，因此最常见的情况就是我们所描述的这种。

2. 装载虚拟机

每个虚拟机需要装载一组软件才能做有意义的工作。软件可以作为虚拟机的一部分进行装载，也可以在虚拟机启动后由虚拟机装载。可以通过装载和配置带有所需软件和数据的机器来创建虚拟机镜像，然后将机器内存中的内容（通常是以虚拟硬盘的形式）复制到持久性文件中。然后能够从该虚拟机镜像随意地创建新的虚拟机实例。

创建虚拟机镜像的过程称为*烧制镜像*。一个*重度烧制*的镜像包含了运行应用所需的所有软件，而一个*轻度烧制*的镜像则仅仅包含了一部分需要的软件，如操作系统和中间

件容器。我们将在第5章探讨这些选择以及相关的权衡。

虚拟化引入了你应该注意的多种不确定性。

- 由于一个虚拟机与一个物理机上的其他虚拟机共享资源，所以虚拟机之间可能会有性能干扰。这种情况对于云用户来说可能尤其麻烦，因为他们通常看不到位于同一个物理机上的其他用户的虚拟机。
- 当加载虚拟机时，取决于底层物理基础设施和需要动态加载的其他软件，可能有时间和可靠性方面的不确定性。为了搭建不同的环境或者部署新版本的软件，DevOps运维常常需要频繁创建和销毁虚拟机。最重要的是，你要意识到这些不确定性。

2.2.2 IP和域名系统管理

创建虚拟机时，为其分配一个IP地址。IP地址是因特网上的计算机之间路由消息的机制。IP地址、路由以及它们的管理都是复杂的主题。接下来，探讨与虚拟机相关的域名系统（Domain Name System，DNS）和IP地址的持久化。

1. DNS

万维网（World Wide Web，www）之下是一个将URL的一部分转换成IP地址的系统。该功能关注URL的域名部分（例如，ssrg.nicta.com.au），这一部分可以通过DNS解析为一个IP地址。例如，作为浏览器正常启动的一部分，一个DNS服务器的地址会提供给浏览器。如图2-1所示，当你输入一个URL到浏览器中时，浏览器将该URL发送给它知道的DNS服务器，它与一个更大的DNS服务器的网络相连，该DNS服务器将该URL解析为一个IP地址。

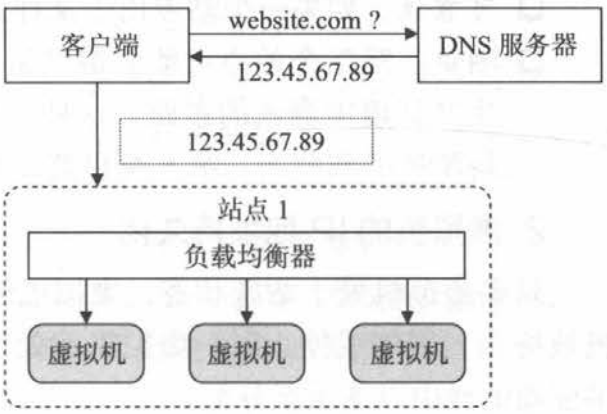


图2-1 DNS返回一个IP地址[标注法：架构]

域名表示了用于解析的路由路径。例如，域名ssrg.nicta.com.au会先去根DNS服务器查找如何解析.au。根服务器会提供澳大利亚DNS服务器的IP地址，该服务器保存澳大利亚的.com名称。.com.au服务器会提供nicta DNS服务器的IP地址，nicta服务器会提供ssrg的IP地址。

该层级结构的重要性在于，较低的层次（.nicta和.ssrg）完全由本地控制。因此，更改.nicta服务器中的ssrg的IP地址相对容易且是本地的。

此外，每个DNS项都有一个名为存活时间（Time To Live，TTL）的属性。存活时间作为DNS项（例如，域名和IP地址的映射）的过期时间。客户端和本地DNS服务器会缓存该DNS项，而该缓存项将在存活时间指定的时间内保持有效。当查询在过期时间

前到达时，客户端 / 本地 DNS 服务器将从其缓存中直接获取 IP 地址。当查询在过期时间之后到达时，IP 地址将由权威的 DNS 服务器来解析。通常，存活时间会设置一个比较大的值；它可能会设置为 24 小时。也可能会将存活时间设置为低至 1 分钟。我们会在第 11 ～ 13 章中的案例研究中看到，DevOps 如何将本地控制和短存活时间结合起来使用。

有一点值得深入说明。在图 2-1 中，针对一个域名，DNS 返回了一个 IP 地址。实际上，DNS 能够返回多个地址。图 2-2 显示 DNS 服务器返回了两个地址。

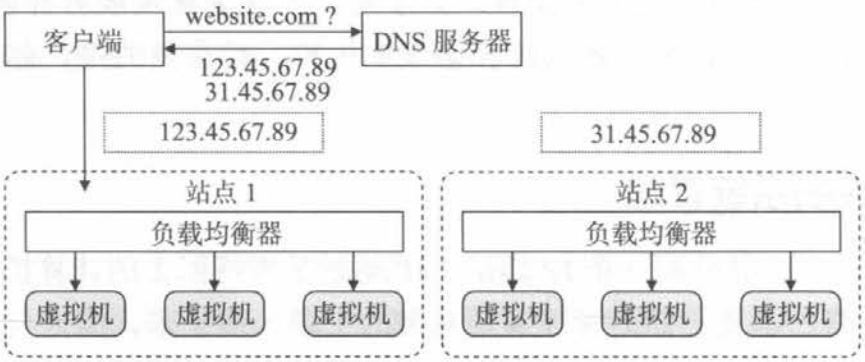


图 2-2 针对一个 URL，DNS 返回两个地址 [标注法：架构]

客户会尝试第一个 IP 地址，如果没有响应，客户将尝试第二个，以此类推。DNS 服务器可能会轮换服务器的顺序以便提供某种程度的负载均衡。

多个站点的存在可能有几个原因：

- 性能。一个站点要服务的用户太多了。从而需要多个站点。
- 可靠性。如果一个站点由于某种原因无法响应，客户可以尝试第二个站点。
- 测试。第二个站点可能会提供某些功能或一个新版本，这些是你想在一个受限的生产环境中测试的东西。这种情况下，到第二个站点的访问被限制在你想让其执行测试的那些人。第 5 章和第 6 章会提供有关该方法的更多细节。

2. 虚拟机的 IP 地址持久化

只要虚拟机处于活跃状态，虚拟机创建时分配给它的 IP 地址就会一直保持。当虚拟机被终止、暂停或停止时，虚拟机就处于不活跃状态。这种情况下，IP 地址会返回给云供应商的池中以便重新分配。

IP 重新分配的后果之一是：如果应用中的一个虚拟机向应用中的另一个虚拟机发送了一个消息，发送方必须验证接收方的 IP 地址尚未过时。考虑下面的序列，这里你的应用至少包含 VM_A 和 VM_B

1）VM_B 收到来自 VM_A 的消息。

2）VM_A 发生故障。

3）云供应商重新分配了 VM_A 的 IP 地址。

4）VM_B 向最初的 IP 地址做出响应。

5）消息传递到一个不属于你应用的虚拟机中。

要避免这个序列，你要么向云供应商请求持久IP地址（通常为高级用户提供），要么你的应用虚拟机就得在发送消息前验证接收方尚处于活动状态并且有相同的IP地址。我们在第4章探讨验证虚拟机的活动状态的机制。

2.2.3 平台即服务

到目前为止，我们探讨的许多方面都是特定于IaaS的。当使用PaaS产品（PaaS offering）时，你可用从很多这样的细节中抽身出来，因为PaaS服务驻留在栈的更高层上并在一定程度上隐藏了底层细节。

正如之前美国国家技术与标准研究所定义中所述，PaaS产品使你能够在预定义的环境中运行应用。例如，你可以将Java的Web应用编译成Web应用程序存档（WAR）文件，并将其部署在托管的Web应用容器中。然后，你可以根据特定的需要来配置服务，比如底层资源的数量（通常已标准化），并将该应用连接到托管的数据库管理系统（SQL或NoSQL）。虽然多数PaaS平台提供了托管的解决方案，或者在自己的基础设施上或者基于IaaS，一些平台也可用于自行管理的安装。

多数PaaS平台提供了一组核心服务（例如，托管Java Web应用、Ruby Gem包、Scala应用等）以及一系列扩展（例如，特定的监控方案、自动伸缩选项、日志流，以及警告等）。某种程度上，PaaS与传统Ops部门提供的一些服务相类似，Ops部门通常接管基础设施层的管理并向开发团队提供一组环境选项来托管他们的系统，开发团队可以在这些环境中进行选择。然而，与传统Ops部门相比，使用具有全球可用性的供应商PaaS通常意味着你会有更多的扩展并能更快地获得更新的选择。

与IaaS类似，如果你对具体的PaaS产品没有经验，你首先要学习怎么使用它。这包括平台特定的工具、结构、配置选项，以及业务逻辑。大多数PaaS平台上手都相对容易，但要掌握命令和配置中错综复杂的细节还要花些时间。

相比于IaaS，PaaS的额外抽象意味着你能够集中精力在系统最重要的部分——应用。你不必处理网络配置、负载均衡器、操作系统、底层的安全补丁等。但这也意味着你要放弃对底层的可见性和控制。如果这是可以接受的，那么使用PaaS就是很值得的。然而，当你之后最终需要额外的控制时，迁移可能非常困难。

2.2.4 分布式环境

本节中，我们要探索在云供应商环境中拥有成千上万的机器所产生的一些影响。这些影响关注各种操作所花费的时间、故障的可能性，以及这两方面在数据一致性上造成的后果。

1. 时间

在独立的计算机系统中，从内存读取数据项的时间与从磁盘读取数据项的时间有很大的差异。由于硬件速度的提升，实际数字会随时间改变，这里只是直观地感受一下这

种差别，从内存顺序访问 1MB（大约 100 万字节）大概需要 12μs（微秒）。从旋转型磁盘访问项目大概需要 4ms（毫秒）将磁盘头移动到正确的位置。然后，读取 1MB 数据大约花费 2ms。

在分布式环境中，消息是应用中不同进程之间的通信方法，消息在相同数据中心的一个来回大概需要 500μs，而在加州和荷兰之间的一个来回大约花费 150ms。

这些数字的后果之一是：确定哪些数据放在内存中或者哪些放在磁盘上是一个性能方面的关键决策。缓存容许在两个地方维护一些数据，但会引入保持数据一致性的问题。第二个后果是，持久性数据的物理存放位置也会对性能产生巨大的影响。结合这两个后果以及故障的可能性引出了使用不同风格的数据库管理系统来保持数据一致性的探讨，接下来将对此进行讨论。

2. 故障

虽然任何特定的云供应商都保证高可用性，但这些保证通常是针对他们整个云的，而不是指其中的组件。单个组件的失败还是会影响你的应用。下面的列表是来自 Google 的一些数据，这些数据是一个数据中心可能遇到的各种故障。亚马逊发布的数据表明，在大约 64 000 台服务器（每个服务器配置两个磁盘）的数据中心，每天平均有 5 台服务器和 17 个磁盘发生故障。

下面是数据中心第一年运营中遇到的问题列表（来自 Jeff Dean 的演讲，Google）：

- 大约 0.5 次过热（大多数机器 <5 分钟内关机，大约 1 ～ 2 天恢复）。
- 大约 1 次 PDU 故障（大约 500 ～ 1000 个机器突然消失，大约 6 小时恢复）。
- 大约 1 次机柜移动（有大量警告，大约 500 ～ 1000 台机器关机，大约 6 小时）。
- 大约 1 次网络重新布线（为期 2 天的时间内梳理了大约 5% 的机器）。
- 大约 20 个机柜故障（40 ～ 80 台机器立刻消失，1 ～ 6 小时找回）。
- 大约 5 次机柜变得不稳定（40 ～ 80 台机器观察到 50% 的丢包）。
- 大约 8 次网络维护（4 次可能引起大约 30 分钟的随机连接丢失）。
- 大约 12 次重启路由器（将 DNS 拿出几分钟）。
- 大约 3 次路由器故障（必须立即拉流量 1 小时）。
- 大约几十个小于 30 秒的 DNS 暂时性问题。
- 大约 1000 个单机故障。
- 大约几千个硬盘故障。
- 磁盘变慢、内存问题、错误配置的机器、运行不正常的机器等。
- 长途连接：野狗（wild dogs）、鲨鱼（sharks）、死马（dead horses）、醉酒的猎人（drunken hunters）等。

从应用或运维的角度，这些故障统计意味着什么？首先，任何特定的虚拟机或网络部分都可能发生故障。这个虚拟机或网络也许正在执行应用或运维功能。其次，由于组件串行使用的故障的可能性与单个组件故障率的积相关，所以一次请求涉及的组件越多，

故障的可能性就越大。我们将分别讨论这两种可能性。

（1）虚拟机故障

分布式系统的架构师要做的主要决策之一是如何在应用的不同部分之间划分状态。如果无状态组件发生故障，可以直接替换它，无需考虑状态。另一方面，状态必须要维护在某个应用可以访问到的地方，而在同一个虚拟机中进行状态获取和计算会带来某种程度的开销。我们区分三种主要情况。

1）*无状态组件*。如果虚拟机是无状态的，则虚拟机故障的恢复可以通过创建同一个虚拟机镜像的另一个实例并确保消息正确地路由到该实例。从故障恢复的角度，这是最理想的情况。

2）*客户端状态*。会话（Session）是两个或多个组件或设备之间的对话。通常，给每个会话分配一个 ID 来提供对话的连续性。例如，你可以通过浏览器和服务器之间的一次交互来登录一个网站。在连续的消息中，会话状态使浏览器能够告知服务器你已成功登录，以及你是你所声称的人。有时，出于安全或应用程序的目的，客户端会添加额外的状态。由于客户端状态必须用消息发送来通知上下文的服务器或一组参数，所以它应该保持最小。

3）*应用程序状态*。应用程序状态包含了应用程序特定的信息或应用程序的特定用户的信息。它可能很大，比如一个知识库或网络爬虫的结果；它也可能很小，比如用户观看视频流时的当前位置。我们确定了 3 类应用程序状态。

a. *少量的持久状态*。持久状态要在多个会话之间或在发生故障的服务器和客户端之间进行维护。少量的持久状态可以保持在平面文件中或文件系统的其他结构中。应用即可以为每个用户保持状态也可以为整个应用保持状态。还可以使用工具将少量状态缓存起来，像 ZooKeeper 或 Memcached 这样的工具可以在多个虚拟机之间维护持久状态。

b. *中等量的持久或半持久状态*。之前看到的时间数据表明，将计算中经常用到的那部分持久状态缓存起来是有利的。在一个虚拟机的不同实例之间维护状态也是有好处的，该虚拟机容许共享该状态。在某种意义上，这相当于硬件层的共享内存，只不过它是通过网络在不同虚拟机之间完成的。像 Memcached 这样的工具意在管理一定量的共享状态，这些状态代表缓存的数据库记录或生成的页面。Memcached 自动向其客户提供了一个一致的数据视图，并通过在服务器之间共享数据，为虚拟机故障时提供了容错性。

c. *大量的持久状态*。大量的持久状态可以保存在数据库管理系统所管理的数据库中或者像 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）这样的分布式文件系统中。Hadoop 分布式文件系统就像一个网络（或者至少是集群）文件系统并且它能自动维护数据项的副本来防止故障。它通过一些机制提供高性能，例如将数据按 64M 的块写入。大的块尺寸造成少量数据写入低效。因此，Hadoop 分布式文件系统应该用于大数据量。既然 Hadoop 分布式文件系统的文件可以在一个集群内访问，那么任何客户故障都不会丢失已提交到 Hadoop 分布式文件系统的数据。

（2）长尾

如图 2-3a 所示，许多自然现象呈现出一种正态分布。值主要分布在平均值附近，越靠近边缘值分布的概率越低。在云上，许多现象，如请求响应时间，表现出一种长尾分布，如图 2-3b 所示。这个结果通常是由于故障的概率随着涉及实体的增多而增加，一个组件的故障会造成响应时间比平常变慢一个数量级（例如，在主网络链路断开并且错误被检测到后，网络数据包要通过不同的链路进行路由）。

在 Map-Reduce 完成时间、搜索查询的响应时间以及亚马逊云的启动实例中都观察到长尾现象。对于后者，满足一个启动实例请求的中位时间是 23 秒，但 4.5% 的请求超过了 36 秒。

尽管尚未证明，但我们的直觉是，分布的偏度（长尾的长度）是为满足一个请求而激活的不同云元素数量的函数。也就是说，诸如计算、读文件或者接收本地消息这样的简单请求都有接近正态的分布。复杂的请求，如大量的 Map-Reduce 任务、大型数据库的搜索或启动虚拟实例，将有像长尾这样的偏态分布。

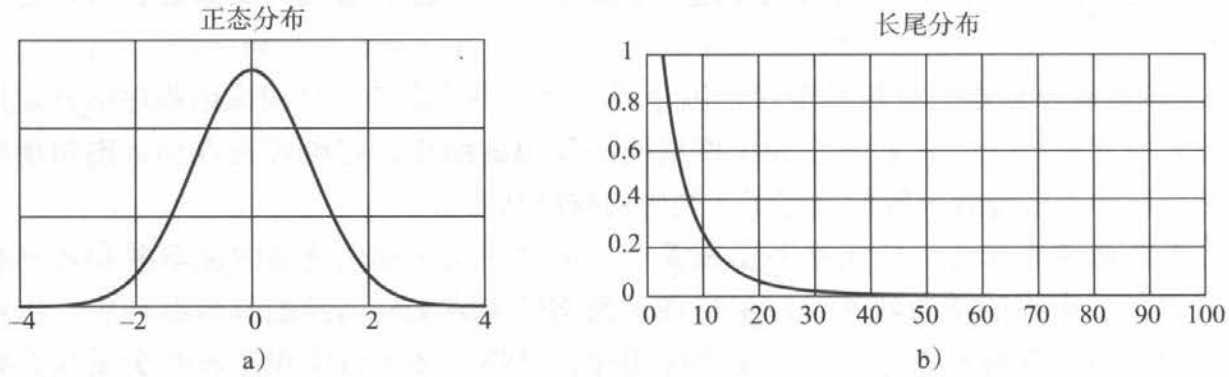


图 2-3 a）正态分布，值集中在均值附近，中值与均值相等；b）长尾分布，有些值离中值非常远

耗费特别长时间响应的请求应该当作故障。然而，这种请求的一个问题是，没有办法知道这个请求已经完全失败，还是最终会完成。防止长尾的一个机制是取消耗时过长的请求，例如，耗时超过了 95% 的历史请求，重新发起该请求。

3. 一致性

考虑到故障的概率，稳妥的做法是复制持久性数据。如果数据项有两个备份，理想状况是，当客户读取数据项时，无论其读取哪个备份，他都应该获取到相同的值。如果数据项的所有备份在特定时刻拥有相同的值，那么就认为它们在那一时刻是*一致的*。记得将数据值写入持久存储是要花时间的。

在分布式系统中，通过引入锁来维护一致性，这些锁控制访问单个数据项的顺序。但锁定数据项会带来访问这些数据项的延迟，因此，产生了各种不同的方案来维护一致性并减少由锁引起的延迟。无论使用什么方案，引入锁带来的延迟将影响数据项的可用性。

另外，云的持久性数据可能按不同区域进行分割，以减少访问时间，特别是在数据量很大时。根据称为 CAP（Consistency：一致性，Availability：可用性，Partition tolerance：分区容错性）原理的理论结果，不可能同时拥有完全可用、一致并分区的数据。最终一致性意味着当改变数据项时，分布、分区和复制的数据即便没有立即一致，在一段时间后将会一致——副本最终将变得一致。

4. NoSQL 数据库

由于种种原因，包括 CAP 原理以及设置关系数据库系统造成的开销，许多数据库系统在 NoSQL 的名称之下被开发出来。起初，这个名字的意思是 No SQL，由于现在有些系统支持 SQL，所以现在它代表 Not Only SQL。

NoSQL 系统使用与关系系统不同的数据模型。关系系统的基础在于将数据呈现为表。NoSQL 系统使用了从键值对到图的数据模型。NoSQL 系统的兴起造成了几个结果。

- NoSQL 系统还没有像关系系统那样成熟，关系系统的许多功能（诸如事务、模式和触发器）还没有被这些 NoSQL 系统支持。如果应用程序需要这些功能，开发人员必须自己实现。
- 应用程序开发人员必须确定哪种数据模型最适合他们使用。不同的应用程序在其持久性数据方面有着不同的需求，这些需求必须在选择数据库系统前得到理解。
- 应用程序可能针对不同需求使用多个数据库系统。键值存储能够高效地处理大量半结构化数据。图数据库能够高效维护数据项之间的连接。使用多个不同数据库系统的好处是，你能够让系统与你的需求更好地匹配。第 11 章的案例研究给出了一个为不同目的使用多个数据库系统的例子。许可证费用和增加的维护费用是使用多个不同数据库系统的缺点。

5. 弹性

快速弹性和供给是美国国家技术与标准研究所确定的云特性之一。弹性意味着服务一个应用的资源（如虚拟机）数量可以根据负载增加和减少。监控现有资源的利用率是度量负载的一种方法。

图 2-4 展示了客户通过负载均衡器访问虚拟机并通过监控确定不同虚拟机（这些虚拟机放在一个伸缩组中）的 CPU 和 I/O 利用率。监控将信息发送给伸缩控制器，这个控制器包含许多规则，这些规则确定何时向伸缩组添加服务器或者何时从伸缩组移除服务器。这些规则可能是被动的（例如，“当利用率达到某个阶段时，添加额外的服务器”）或者是主动的（例如，在早上 7:00 添加额外的服务器，在下午 6:00 移除它们）。当添加新服务器的规则被触发时，伸缩控制器会创建新的虚拟机并确保其装载正确的软件。之后，将新虚拟机注册到负载均衡器中，现在负载均衡器就有额外的虚拟机用来把消息分发给它。还可以通过各种应用程序编程接口来控制伸缩。我们会在第 12 章看到这方面的例子。

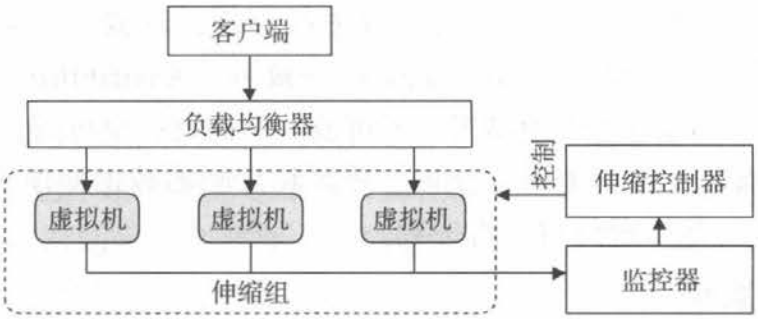


图 2-4 监控作为伸缩的输入 [标注法：架构]

2.3 独特的云特性对 DevOps 的影响

云影响 DevOps 的 3 个独特特性是简单地创建和切换环境的能力；轻松创建虚拟机的能力，以及数据库的管理。我们首先讨论环境。

2.3.1 环境

在我们的上下文中，环境是足够执行软件系统的一组计算资源，包括所有支持软件、数据集、网络通信，以及执行软件系统所需定义的外部实体。

这个定义的关键点是：除了明确定义的外部实体外，环境是自包含的。一个环境通常独立于其他环境。在第 5 章中，我们会看到一些环境，例如开发、集成、用户测试，以及生产环境。在第 12 章的案例研究中，环境的生命周期是其部署流水线的一部分。在开发、测试和部署过程的各阶段有多个环境并不是云独有的特性，但简单地创建和迁移环境的能力是云独有的特性——像克隆新实例一样容易。通过使环境之间没有可修改的共享资源来实现一个环境与另一个环境的隔离。只读资源，例如各种订阅源，可以共享而不会带来问题。因为环境与外部世界的通信只通过已定义的外部实体，所以这些实体能够通过 URL 访问，因此可以单独管理。写或修改这些外部实体的状态应该只在生产环境中进行，并为所有其他环境创建独立的外部实体（例如，仿制品或测试克隆）。

将环境可视化的一种方法是将其看作一个竖井。图 2-5 展示了两个不同环境（测试环境和生产环境）的两个变体。每一个包含了同一个系统稍有差异的版本。两个负载均衡器（负责它们各自的环境）有不同的 IP 地址。如图 2-5a 所示，测试可以通过分流到生产环境的输入流并将副本发送到测试环境来实现。这种情况下，重要的是测试数据库与生产数据库隔离。图 2-5b 给出了一个替代方案。这种情况下，将实际生产消息的某个子集发送到测试环境来执行现场测试。我们在第 6 章讨论金丝雀测试以及其他现场测试方法。环境之间的移动能够用一个脚本来实现，在使用脚本之前要测试其正确性。我们将在第 6 章看看其他在测试和生产环境之间移动的技术。

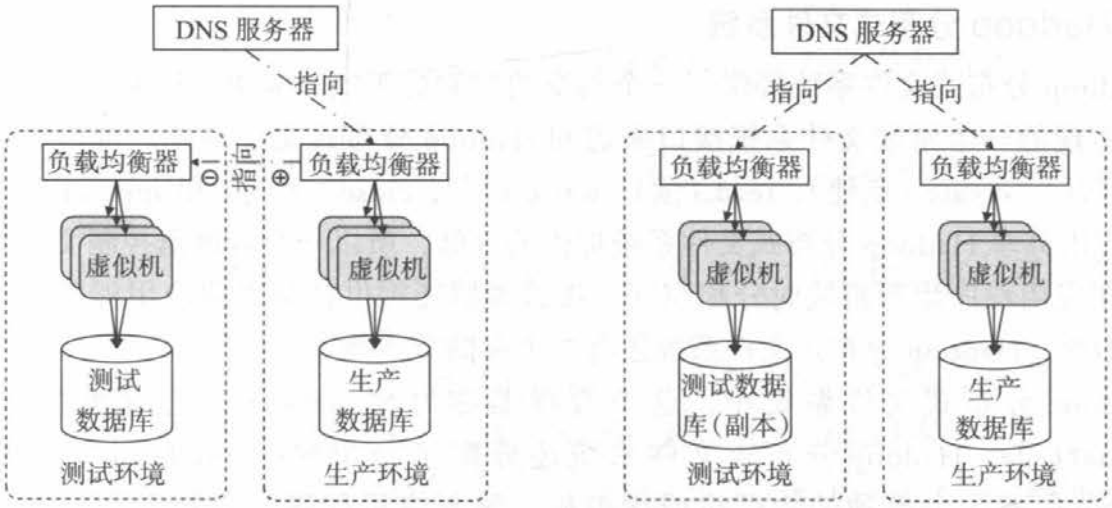


图 2-5 a）使用实际数据测试；b）使用一个用户子集进行现场测试 [标注法：架构]

将生产轻松地从一个环境切换到另一个环境的结果是使业务连续性的实现变得更容易。业务连续性意味着当主数据中心发生灾难时业务能够继续运作。在第 11 章，我们将看到管理多数据中心的案例，但此刻并没有要求将两个环境放在同一个数据中心。如果目标是从一个环境迅速切换到备份环境，就需要同步两个数据库。

2.3.2 轻松创建虚拟机

由于分配新虚拟机过于容易，所以用户管理云方面的一些问题就浮现出来。虚拟机需要打最新的补丁，就像物理机一样，还需要付费。没有打补丁的机器会成为安全风险。而且，在公有云中，用户要付费使用虚拟机。我们听闻过一个发生在主要美国大学的事件，有个学生出去过暑假，但没有清理自己分配的资源，结果回来发现一个 $80 000 的账单。

术语*虚拟机蔓延*用来描述管理过多虚拟机的复杂性。类似地，拥有太多虚拟机镜像的挑战称为*镜像蔓延*。有工具（如 Janitor Monkey）可以扫描账户并确定分配了哪些机器以及它们最近的使用情况。当利用云即平台时，必不可少的活动之一就是为机器的分配和虚拟机镜像归档开发并执行一个策略。

2.3.3 数据考量

云的经济效益伴随 NoSQL 数据库系统一同到来。许多系统利用多个不同的数据库系统，既有关系数据库也有 NoSQL。此外，大量数据为了不同的商务智能或运营目的从各种渠道汇聚在一起。如同云中的计算资源能够通过扩展增加，存储资源也能够增加。我们先探讨 Hadoop 分布式文件系统，它为一个集群内的应用提供存储。Hadoop 分布式文件系统为许多 NoSQL 数据库系统提供文件系统。然后，我们探讨与分布式文件系统相关的运维注意事项。

1. Hadoop 分布式文件系统

Hadoop 分布式文件系统提供了一个共享的存储资源池。应用通过 Java、C 或其他流行语言实现的一个常规文件系统接口来访问 Hadoop 分布式文件系统。可用的命令包括 open（打开）、create（创建）、read（读）、write（写）、close（关闭）和 append（追加）。由于多个应用共享 Hadoop 分布式文件系统提供的存储，所以一个管理器控制文件名的命名空间并在应用程序想写新块时分配空间。该管理器还提供信息以便应用程序能够直接访问特定的块。Hadoop 分布式文件系统还有一个存储节点池。

Hadoop 分布式文件系统中，这个管理器称为 NameNode，存储池的每个元素称为 DataNode。Hadoop 分布式文件系统还分配了一个 NameNode 用于热备份。每个 DataNode 是一个单独的物理机或虚拟机。限制应用程序写固定大小的块——通常是 64MB。当应用程序要向文件写一个新块时，它联系 NameNode 并请求存储该块的 DataNode。每块将复制多次，通常是 3 次。NameNode 用一个 DataNode 列表响应写请求，并得到可写块，之后应用程序把块写入每一个 DataNode 中。

Hadoop 分布式文件系统的许多特性都是用来防范单个 DataNode 的故障并提高 Hadoop 分布式文件系统的性能。就我们的目的而言，关键要素是 Hadoop 分布式文件系统提供了一个在应用程序之间共享的存储池。

2. 运维注意事项

与 Hadoop 分布式文件系统这样的共享文件系统相关的运维注意事项是双重的。

1）谁管理 Hadoop 分布式文件系统的安装？ Hadoop 分布式文件系统既可以是多个应用程序之间共享的一个系统，也可以只是一个应用程序的实例化。在单一应用程序的情况下，其管理职责将落在该应用程序的开发团队身上。在共享的情况下，系统的管理必须指派给组织内的某处。

2）存储在 Hadoop 分布式文件系统内的数据是如何防范灾难的？ Hadoop 分布式文件系统本身会在多个 DataNode 上备份数据，但数据中心的一般故障可能造成 Hadoop 分布式文件系统无法访问或者 Hadoop 分布式文件系统管理的数据被损坏或丢失。对于那些部分业务依赖于 Hadoop 分布式文件系统的持续执行并访问存储在 Hadoop 分布式文件系统中的数据的业务来说，业务连续性成为一个必须解决的问题。

2.4 小结

近些年，云已经成为 IT 界的一个主要趋势。其特性包括计量使用（按使用收费）以及快速弹性，允许应用程序扩展到近乎无限数量的虚拟机。如果架构合理，应用程序确实可以快速伸缩，因此当你的新应用程序像“病毒”一样传播并且每隔几个小时用户数量就翻一倍时，你可以避免让用户失望。另外，当需求减少时，你不会被绑在主要的硬

件投资上，只要将不再需要的资源释放掉就可以了。

云的使用开启了许多有趣的机会，但也意味着你要应对许多分布式计算所关注的问题：

- 云依赖的平台天生就是分布式的，云还利用虚拟化来实现快速扩展和收缩给定用户的资源。
- IP 地址是访问虚拟资源的关键并通过 DNS 条目与 URL 相关联，可以操纵 IP 地址来支撑通过隔离环境进行的各种形式的测试。
- 在大型分布式环境中，单个组件的故障是意料之中的。因此，必须适应故障。这种适应涉及状态的管理以及识别超长时间的请求并从超长请求中恢复。
- 从运维的角度，控制虚拟机的激增，管理不同的数据库管理系统，确保环境满足开发和运维任务的需要是与云相关的新的注意事项。

2.5 更多阅读材料

美国国家技术与标准研究所的云定义是 SP 800-145 这个专门出版物的一部分 [NIST 11]。

针对不同类型的内存以及网络连接的延迟数据源自 http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html。

Jeff Dean 的主题发言列出了新数据中心的问题 [Dean]。

James Hamilton（来自 AWS）在他的报告中给我们展示了大规模发生的故障，报告获取链接如下：http://www.slideshare.net/AmazonWebServices/cpn208-failuresatscale-aws-reinvent-2012。

可以在 http://memcached.org/ 访问 Memcached 系统的网站。

Hadoop 分布式文件系统及其架构的更多信息可以在：

- http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html。
- http://itm-vm.shidler.hawaii.edu/HDFS/ArchDocOverview.html。

找到。长尾分布和它的一些表现在 [Dean 13] 中描述了。

演示文稿 [Kandula] 探讨了 MapReduce 中的异常值。

论文“Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud”为容忍长尾行为提供了方法和架构技巧 [Lu 15]。

Netflix 的 Janitor Monkey 有助于控制虚拟机和镜像蔓延。参见网站：https://github.com/Netflix/SimianArmy/wiki/Janitor-Home。

CAP 原理由 Erick Brewer 首次提出，并由 Gilbert 和 Lynch 证明 [Gilbert 02]。

第3章 运　维

在DevOps社区有一些核心思想家，他们知道IT管理是怎么回事，也意识到在DevOps情境下使用ITIL。也有一些人实际上是很宽松地把握这一点的……

——Rob England，http://www.itskeptic.org/devops-and-itil

3.1 概述

正像DevOps不只是简单包含开发那样，它也不只是简单包含运维。然而，要想理解DevOps，先理解这样一个情境是非常重要的——开发和运维的那些人是从哪里来的。在本章，我们将展示由IT运维组执行的活动。有多少活动是适合于DevOps方法的仍存在争议，而我们将就此争议进行评论。

对运维的一个特征的描述来自于信息技术基础设施库（ITIL）。ITIL对运维人员有着粗糙的描述。ITIL是基于“服务”的概念，它认为运维工作是在全局策略的大背景下支持这些服务的设计、实施、运维和改进。图3-1展示了ITIL如何看待这些活动的相互作用。

我们首先描述传统意义上，运维拥有职责的那些服务上。然后，我们返回到图3-1来讨论服务生命周期。最后，我们讨论如何将DevOps适配到整体架构中。

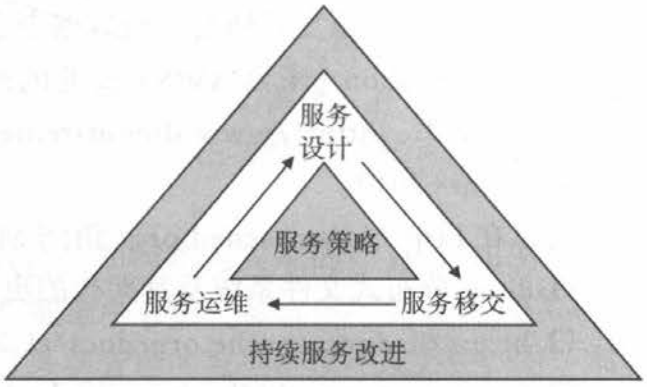


图3-1　服务生命周期（改编自ITIL）

3.2 运维服务

运维服务包括供给硬件、供给软件，或者支持各种不同的IT功能。由运维提供的服务，还包含服务级别协议（Service Level Agreement，SLA）的规格说明和监控、容量规划、业务连续性以及信息安全。

3.2.1 供给硬件

硬件可以是组织拥有的物理硬件，也可以是由第三方或云供应商管理的虚拟硬件，也可以是由个人、项目，或者大型组织中的一部分所使用的硬件。表3-1展示了这些可

能性。

表 3-1 个人、项目和组织使用的硬件类型

使用者	物理硬件	虚拟硬件
个人	笔记本电脑、台式机、平板电脑、智能电话机	用于开发和单元测试的虚拟机
项目	集成服务器、版本控制服务器	用于集成和版本控制的虚拟机
组织	用于打印机、网络基础设施等服务的服务器	用于组织范围的服务的虚拟机

- 物理硬件。
  - 个人硬件。个人硬件包括笔记本电脑、台式机、平板电脑和分配给个人使用的电话机。通常，运维将使用他们提供的标准配置。实际的硬件订购和配置到底是由运维还是由个人自己完成，在不同的组织之间是有所不同的。标准化配置的强制程度在不同的组织之间也有所不同。一种极端情况是，运维提供给每一个员工一组被锁定的设备，而员工只能装载被批准的软件并进行配置。配置之间的兼容性是由标准化来提供的。
  - 项目硬件。项目硬件通常包括集成服务器和版本控制服务器，尽管这些硬件可能在组织范围内进行管理。项目硬件的需求由项目来决定，运维可能会因组织范围内的指派，或项目与运维协商后参与订购、配置和支持项目硬件。
  - 组织范围硬件。这种类型的硬件是由运维负责的。任何数据中心或者常规可用的服务器（如邮件服务器）都是由运维管理和运行的。
- 虚拟硬件。虚拟硬件也可以是个人、项目专属或者组织范围的。它遵循物理硬件的模式。项目通常负责虚拟项目硬件的规格说明和管理，而运维则负责说明和管理组织范围的虚拟硬件。运维通常负责虚拟机的整体使用情况，并且当项目考虑他们的虚拟硬件需求时，运维可以提供预算支持。注意，除了设备、打印机和数据中心的网络基础设施外，很多项目或组织级的物理硬件都是可以在私有云或公有云资源上被虚拟化的。

3.2.2 供给软件

软件可以是内部研发（也可能是通过合同外包），也可以从第三方购买。第三方软件可以是项目特定的，这种情况下，它遵循我们对硬件的处理模式（软件的管理和支持都由项目负责）；或者是组织特定的，这种情况下，软件的管理和支持由运维人员负责。防止内部研发的软件延期是 DevOps 的一个驱动力，我们将在本章的最后回到传统的运维和 DevOps 之间的关系。

表 3-2 不同类型软件的职责

开发者	提供支持者
项目	项目
第三方	运维人员或者项目，取决于使用范围
运维人员	运维人员
DevOps 团队	DevOps 团队

表 3-2 显示了不同类型软件的职责。

3.2.3 IT 功能

运维活动支持不同的功能。这些功能包含：

- **服务台的运维**。服务台的员工负责处理所有的事件和服务请求，并作为所有问题的第一级处理者。
- **技术专家**。运维团队通常有网络、信息安全、存储、数据库、内部服务器、Web 服务器和应用程序以及电话的专家。
- **IT 服务的日常供给**。这些包含周期性和重复性的维护运维、监控、备份以及设施管理。

包含在 DevOps 的运维端中的人通常来自最后两类。日常 IT 服务包含提供新的软件系统或当前系统的新版本，并且 DevOps 的主要目标是改进这个过程。正如我们将在第 12 章中的案例研究看到的那样，信息安全和网络专家也要加入 DevOps，至少参与持续部署流水线的设计，在理想情况下，部署流水线在组织内是共享的以提升标准化和避免随时间而发生漂移（drifting）。

3.2.4 服务级别协议

一个组织和外部服务供应商有大量的服务级别协议。例如，一个云供应商需要保障一定的可用性。传统上，运维的责任是遵守监控和确保服务级别协议。一个组织和它的客户也有大量的服务级别协议。运维按照惯例为确保组织实现其外部服务级别协议负责。与外部服务级别协议一样，运维通常负责实现内部服务级别协议的要求。例如，一个组织拥有自己的网站或电子邮件服务。在 DevOps 运动中，开发和 DevOps 变得要承担更多的应用服务级别协议和外部服务级别协议的责任。

所有这些功能都包含监控和分析来自服务器、网络和应用的不同类型的性能数据。参见第 7 章关于监控技术的大量讨论，以及第 10 章关于从业务视角来讨论监控什么内容。

3.2.5 容量规划

运维负责确保组织可以获得充足的计算资源。对于物理硬件，这包含采购和配置机器。包括采购和配置这些硬件的前置时间需要考虑在计划之内。

更重要的是，运维负责提供充足的资源，以便一个组织产品的消费者可以浏览产品、下订单和检查订单的状态等。这包含预测工作量和工作量的特征。有些预测可以基于历史数据，但有新产品上线或推广活动时，运维也需要与业务协作。DevOps 强调运维与开发之间的协作，但还有其他干系人参与协作活动。例如，在容量规划中，其他干系人就是业务和营销的人。利用云的弹性、为使用付费的模型以及提供新的虚拟硬件的便捷性，容量规划越来越变为运行时监控和自动伸缩的工作，而不是规划硬件采购。

3.2.6 业务连续性和安全

在灾难发生时，一个组织需要保持关键服务可用，这样内部和外部客户都可以维持他们的业务。组织可以使用 2 个关键参数对能够维持业务连续性的多个可选方案进行成本 / 收益分析：

- 恢复点目标（Recovery Point Objective，RPO）。当灾难发生时，对数据丢失的最长容忍时间是多少？如果每小时备份数据一次，则恢复点目标为 1 小时，因为丢失的数据可能是自上一次备份后的累积量。
- 恢复时间目标（Recovery Time Objective，RTO）。当灾难发生时，不能提供服务的最长容忍时间是多少？例如，如果一种方案需要 10 分钟来获取在另一个数据中心的备份，再需要 5 分钟来实例化新的服务器以便使用这些备份数据，则恢复时间目标为 15 分钟。

这两个值是独立的，因为有些数据的丢失也许是可以容忍的，但不能提供服务则是不能容忍的。也可能能够忍受没有服务，但不允许丢失数据。

图 3-2 显示了 3 种可选的备份方案，它们有不同的恢复点目标。在第 11 章的案例研究中，我们描述了一个组织所采用的方案。另一个方案则会在第 13 章中讨论到，这种方案使用了云供应商的服务来进行复制。

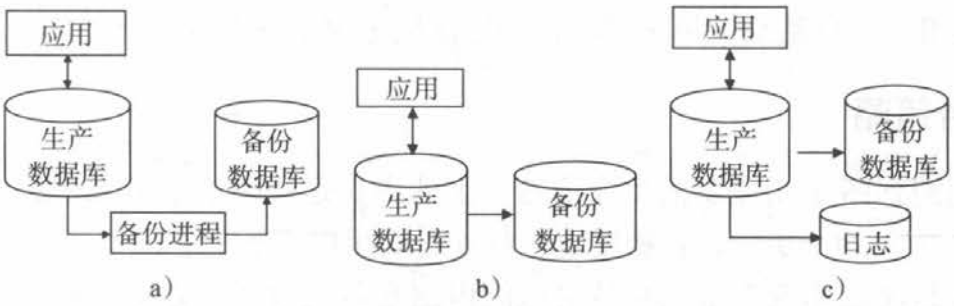


图 3-2 数据库备份策略：a）由一个独立的代理实施备份；b）由数据库管理系统实施备份；c）数据库管理系统实施备份并将所有的事务记录到日志中 [标注法：架构]

1）图 3-2a 显示了一个外部代理（备份进程），它定期复制数据库。不需要应用程序支持，但备份进程应该复制数据库的一致版本。也就是说，当时没有更新正在应用。如果备份进程对数据库管理系统是外部的，那么事务可以继续进行，因此备份应该谨慎地启动。这种情况下，恢复点目标为两次数据备份之间的时间间隔。也就是说，如果一个灾难正好先于启动备份进程，则从上一次备份后的期间内的所有变更都会丢失掉。

2）图 3-2b 显示了一个没有外部代理的可选方案。这种情况下，数据库管理系统定期创建一个复制版本。图 3-2a 和图 3-2b 的区别在于，在图 3-2b 中，确保一致性是由数据库管理系统完成的。而在图 3-2a 中，一致性则是由一些管理启动备份进程的机制来保证的。如图 3-2a 所示，恢复点目标为两次复制之间的时间周期。如果这个数据库是一个关系数据库管理系统（RDBMS），它提供一些复制（也就是说，一个事务只完成一次提

交，当复制数据库也已执行了该事务时），则在灾难发生期间丢失的事务就是那些还没有提交给复制数据库的事务。然而，每一次事务的开销都会增加。

3）通过数据库管理系统对每一次的写入都进行日志记录，将图 3-2b 修改为图 3-2c。这样，数据就可以通过首先备份数据库，然后重放日志中的条目进行再创建。在恢复时如果可以得到日志和备份数据库，则恢复点目标为 0，因为所有的数据或者在数据库中或者在日志中。提交事务到生产数据库的协议是，只有当各自的日志条目已经写入数据库中后才能提交事务。虽然这个方案中可能有些事务还没有完成，但是已经完成事务的数据不会丢失。这个方案被高可靠性的关系数据库管理系统使用，也被分布式文件系统使用，例如 Hadoop 分布式文件系统。当考虑恢复时间目标（即，在发生停电或灾难后让应用程序重新运行所需要的时间）时，备选方案包括：使用多数据中心（在第 11 章案例研究中讨论）；使用由云供应商提供的独立的可用区或域；使用多个云供应商。

通过考虑恢复时间目标和恢复点目标，业务可以对各种灾难恢复技术进行成本 / 收益分析。有些技术将包含应用系统架构，如在不同的副本上复制和维护状态一致性。其他技术，如周期性备份技术，可以和任何应用架构一起实施。使用应用层上的无状态服务器和云供应商提供的不同域，可以产生短的恢复时间目标，但没有解决恢复点目标问题。

传统上，运维负责计算机系统的整体安全。保护网络、检测入侵者、为操作系统打补丁都是由运维执行的活动。第 8 章将在一定深度上讨论安全及其维护。

3.2.7 服务策略

现在我们回到图 3-1 显示的生命周期。图的中央是我们枚举的每个服务的策略：硬件供给、软件供给、IT 功能、容量规划、业务连续性和信息安全。

制定策略是决定你希望组织在特定时间内达到特定水平，它首先判断组织当前的位置，决定从当前位置到一个期望状态的路径。期望状态同时受到内部和外部事件的影响。内部事件包含人才流失、硬件故障、新软件发布、市场活动和商务活动等，它们都会影响期望状态。外部事件（例如收购、政府政策或者消费者反馈等）也会影响期望状态。可能发生的事件都有发生的概率，因此，策略规划和算命有一些共通的要素。

理解未来在资源和能力方面的需求将有助于加强当前还不能达到要求的部分。例如，如果你希望明年把一些组织的应用搬到云上，则了解组织内部是否有拥有这些技术的人是很重要的。如果没有，你需要决定招一些有潜力的新人，还是培养现有的员工让他们拥有这些技术，或者两者都有。未来需求将带来持续服务改进活动。

策略规划需要花费时间，并且需要协调不同的干系人。策略规划与其说是一个实际的计划，不如说它是从组织内的多个视角和限制做出的考虑。因此，不要太频繁地定义服务策略，而是应该将结果以易于在短期内实现的方法作为宽松的指导方针。我们将在第 13 章讨论微服务迁移的策略计划，及其案例研究的实施。

3.2.8 服务设计

在一个新的或修改过的服务实施之前，需要对它进行设计。与任何设计一样，你不仅需要考虑所实现服务的功能，而且还需要考虑其他类型的质量。在设计一个服务时需要考虑以下因素：

- 这个服务将涉及哪种自动化并将其作为服务的一部分？任何自动化都应该按照软件设计原则进行设计。总的来说，这些包括 Thomas Erl 对服务设计提出来的 8 个原则：
  - 标准化合同
  - 松耦合
  - 抽象
  - 可复用性
  - 自治
  - 无状态
  - 可发现性
  - 可组装性
- 服务的治理和管理结构是什么？服务需要管理和变革。人们为服务的绩效负责，而服务的修改应该从标题（如果不是从名称）加以区分。
- 服务的服务级别协议是什么？服务是如何度量的，需要哪种监控结构来支持该度量？
- 服务的人才需要是什么？这些服务可以由现有的员工提供还是需要招聘有特殊技能的新员工或者合同工？或者，这些服务可以全部采用外包人员吗？
- 服务的合规性影响是什么？服务满足了哪些合规需求，引入了哪些？
- 服务对容量的影响是什么？是否需要获取额外的资源，获取的时间是什么？
- 服务的业务连续性影响是什么？发生灾难时是否要求必须保持业务连续性，如何实现？
- 服务的信息安全影响是什么？哪些数据是敏感的，并且必须进行保护，以及谁对这些数据负责任？

  在 ITIL 中，服务设计的章节讨论了这些问题的细节。

3.2.9 服务移交

服务移交包含服务设计和运维之间的所有活动，换句话说，将一个新服务或者修改后的服务成功运营需要的所有活动。因此，本书的很大一部分内容是关于服务移交的，因为它将影响软件新版本的引入。

移交和计划支持包含以下方面：资源、能力和变更计划；移交的范围和目标；文档

需求；可应用规则和监管的考虑；财务计划和里程碑。本质上，服务移交包含服务的实施和交付两个阶段。DevOps 和持续部署要求服务移交的交付部分高度自动化，这样它才能够处理高频率的移交，并提供更好的质量控制。第 5 章和第 6 章将有很多这方面的讨论。

除了实现服务设计部分中枚举的考虑外，服务移交还包括将新服务或者修订过的服务的相关知识扩展给用户或者运营中的那些中间支持者。

例如，设想一个新版本的部署工具将实施。需要回答下面这 3 个问题：

- 新版本能支持旧版本的所有功能吗？如果不能，支持旧版本的用户的移交计划是什么？
- 引入了哪些新功能？部署工具的脚本应该如何修改，以及谁将负责修改？
- 新版本是否要求或者支持不同的服务器配置，包含测试、预发布（staging）和生产服务器？

与其他软件一样，部署流水线中的工具也会发生变化。“基础设施即代码”的一个含义是那些变更与在为用户开发的软件的变更一样需要管理。这种管理在有些方面可能隐含在部署流水线中，而其他方面则需要注意。ITIL 区分了 3 种模式：

- 标准变更（例如，经常发生和低风险的变更）
- 常规变更
- 紧急变更

每一种变更都应该使用不同的管理方式，也要求不同的管理重视程度和监督。在 ITIL 的服务移交章节有更多的细节讨论。

3.2.10 服务运维

虽然软件开发工程师和架构师更关注开发，但是只有在运营中，用户才能从好的设计、实施和移交中获得收益——或者不能获得收益。支持在这里扮演了一个重要的角色，尤其是在事故和故障管理中。监控和改造是另一个关注点。我们将在第 9 章进行更多的探讨。

3.2.11 服务运维概念

在运维过程中，ITIL 把事件定义为“任何可检测或者可识别的事情，它对 IT 基础设施管理或者 IT 服务交付以及对造成服务偏移的影响评估具有重要意义”。事件是通过配置项、IT 服务或监控工具创建的。更具体地，监控工具可以主动从配置项或服务中拉取事件信息，或者它们可以（被动地）收到这些信息。运维过程中关于事件的信息包括：

- 来自系统和基础设施的状态信息。
- 环境条件，如冒烟探测器。
- 软件许可证的使用。
- 安全信息（例如，来自入侵检测）。

- 正常活动，例如来自服务器和应用程序的性能指标。

事故，按照ITIL的定义，是“任何中断或可能中断服务的事件”。它们可能是由用户（例如，通过电话或邮件）、技术人员、支持台员工或者监控工具报告的。事故管理是DevOps将影响的一个领域。

事故管理的核心活动有：

- 将事故记录到日志中。
- 分类和排优先级。
- 初步诊断。
- 如果有需要，上报给有相应技术或授权的人。
- 调研和诊断，包括任何关于事故影响程度和范围的分析。
- 解决和恢复，不管是通过获得支持人员引导的用户，还是直接通过支持人员，或者通过内部和外部的专家。
- 事故结束，包含如果需要，则进行重新分类、用户满意度调查、存档以及判断这个事故是否还可能发生。

事故管理是DevOps改变传统运维活动的一个领域。与一个特定软件系统的运维相关的事件转交给开发团队。不管谁接到问题的报告，都必须记录事故并追踪解决过程。在第五部分中，我们将讨论如何在DevOps和过程视角的帮助下将故障检测、诊断和恢复自动化。

3.3 服务运维功能

监控是运维过程中最重要的核心，因为它收集事件、检测事故和度量以判断是否符合服务级别协议。它提供了服务改善的基础。服务级别协议也可以定义和监控运维活动，例如，发生事故后的响应时间。

监控可以和其他控制结合在一起，例如，对云资源的自动伸缩，即在一个Web服务器池中，当平均CPU负载达到70%时就触发一个规则来启动新的Web服务器。控制可以是开环或者闭环。开环控制（即不考虑监控反馈）可以用于在预定的时间进行常规备份。在闭环控制中，在决定采取行动时考虑监控信息，例如在自动伸缩的例子中，闭环反馈周期可以嵌入在更复杂的控制环中，其中低级别的控制对具体的度量指标进行反应，而高级别的控制考虑更广范围的信息和长时间跨度的发展趋势。在最高级别，控制环可以连接不同的生命周期活动。根据所需度量的测量偏差，持续服务改进可以产生服务策略、设计和移交的改变——所有这些最终也都会反过来改变服务运维。

监控的结果由开发或运维团队来进行分析并采取行动。当制定DevOps过程时必须做一个决定：应该由哪个团队负责处理事故？参见第10章关于事故处理的讨论。有一种

DevOps 实践是开发团队分析他们自己开发的单系统监控的结果，而包括基础设施的多系统监控则由运维团队负责，同时运维团队还负责需要一个或多个开发团队合作处理的事故上报。

3.4 持续服务改进

每一个被组织采用的流程都应该从这个视角来考虑：这个流程到底多有效？还可以怎样改进？这个流程如何适应公司整体流程框架？

我们讨论的所有运维服务（硬件和软件的供给、IT 支持功能、服务级别协议的规格说明和监控、容量规划、业务连续性和信息安全）都是组织层面的流程。需要从我们提出来的几个问题的视角来进行监控和评估。

从组织层面来说，每一个服务都应该有一个负责人，并且这个服务的负责人需要负责检查监控、评估和改进。

持续服务改进的主要焦点是要在 IT 服务和业务需求之间达成一致——不管这些需求是否已发生变化还是维持不变。如果这些需求已经发生变化，则对 IT 服务的期望变化可以关注范围、功能或者服务级别协议。如果业务需求相同，则可以扩展 IT 服务来更好地支持它们，但对它们的改进也可以专注在提升效率上。DevOps 关注更快、更可靠地把这些变化引入实践中。

图 3-3 描绘了 ITIL 建议的 7 步骤改进流程。这个数据驱动的流程开始于一个明确的愿景、策略和目标，这些能够驱动当前的改进周期。基于此，第 1 步定义应该度量什么，以便理解应该改进什么，并且当改进完成后，是否达到了期待的目标。指标可以粗略地分成 3 类：技术、流程和服务。

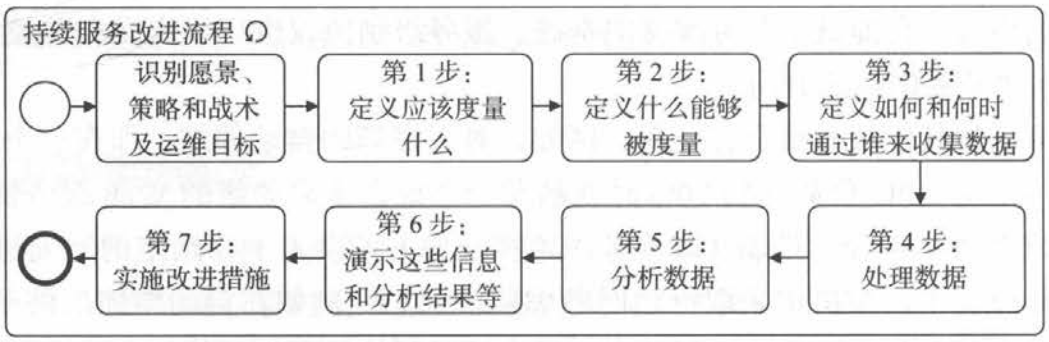


图 3-3 服务持续改进流程（改编自 ITIL）

实际的数据收集是在第 3 步发生的。这对于建立基线是非常重要的（如果它们已经不存在了）用于以后的比较。而且，数据的收集（谁收集和如何收集、什么时间、什么频率收集）需要明确定义。第 4 步，处理数据（例如，将不同数据源的数据进行聚合或按照指定的时间间隔进行）。分析数据是在第 5 步进行。在第 6 步，演示分析中得出的信息，并

决定改进措施。而第7步将已制定好的改进措施付诸实施。这些行动将影响服务的整个生命周期的所有阶段——策略、设计、移交或运维。

3.5 运维和 DevOps

讨论完ITIL的核心概念和阶段后，现在我们强调将来传统IT运维和DevOps之间将形成什么样的交互。我们想要传达的信息是，如果认为ITIL是过于“重量级”而不适合DevOps过程，那么这个观点是短视的，并且这个观点将导致要再走过那些ITIL框架中所试图解决的“坑”。

运维的职责是供给硬件和软件、拥有特殊技能的人员、服务级别协议的规格说明和监控、容量规划、业务连续性和信息安全。这些职责的大部分包含在DevOps过程中和过程外。任何关于运维的哪些工作应该纳入DevOps实践的讨论，都必须考虑运维现在正在开展的所有活动，包括功能性活动、个体技能和可用性。影响运维的所有活动是：

- 硬件供给。虚拟化的硬件可以由开发团队分配或者由更自动化的应用来分配。
- 软件供给。内部开发的软件将由开发团队进行部署，其他软件则由运维提供。
- IT功能供给。就开发团队将负责事故管理和部署工具来说，必须有拥有专业知识的人来执行这些任务。
- 服务级别协议的规格说明和监控。对于那些特定于某个应用的服务级别协议来说，开发团队将负责监控、评估和响应事故。
- 容量规划。开发团队负责单个应用的容量规划，而运维团队则负责整体的容量规划。
- 业务连续性。开发团队负责与应用架构有关的业务连续性的那些方面，而运维团队则负责其他的部分。运维可以为业务连续性提供服务和策略，反过来它们又提供给开发团队使用。
- 信息安全。开发团队负责涉及某个特定应用的信息安全的那些方面，运维团队负责其余部分。

涉及DevOps工作的人数取决于组织采用了哪些过程。一个组织估算运维团队的20%和开发团队的20%参与DevOps过程。影响不同团队参与度的因素有：

- 在事故发生时，开发团队作为第一事故响应责任人的范围。
- 是否有专门的DevOps团队对持续部署流水线的工具负责。
- 两个团队成员的技能和可用性。

ITIL服务移交和DevOps方法之间的一个差别在于ITIL假设非常大的发布包（需要小心计划、变更管理等）是可行的——与典型的DevOps场景中遇到的高频率的小的发布包相比。Rob Spencer在他发表的一篇博客中建议，将DevOps发布视为“更小规模交付物的并发流”，并给出了表3-3中的示例。

表 3-3 发布包示例（改编自 R.Spencer 的博客）

流	频率	ITIL 角色 / 流程
代码对象的检查、测试和部署	每日	研究和开发管理（R&DM）、服务资产和配置管理（SACM）
针对新的功能需求，知识库更新的创建和测试	每隔 1 天	SACM、服务验证和测试（SV&T）、知识管理
正式可操作的验收测试	1 星期 2 次	SV&T、服务等级管理（SLM）、业务关系经理（BRM）、应用 / 技术功能经理
硬件交付物	按需的	R&DM、技术管理
早期生命支持和持续服务改进	每日	持续服务改进（CSI）、SLM、BRM、服务所有者

表 3-3 中的大部分行现在包含在开发流程的生命周期中都可以看作不变量的标准。在 DevOps 中，这些不变量通常频率远高于 ITIL。然而，右边列中的流程和角色来自 ITIL，因此将使用那些已证明的方法和流程。尤其是，最后一行包含“早期生命支持和持续服务改进”的即时连接流。假设发布是每天进行的，则早期生命支持阶段实际上是永无止境的。

虽然很多初创公司会认为这种方法太复杂，而更大规模和更成熟的组织则会发现定义 DevOps 和 ITIL 之间的关系是有用的，并且这种方法可以提高 DevOps 接受度和对 DevOps 的支持。

3.6 小结

ITIL 为运维活动提供通用的指导。这些活动包含供给硬件和软件；提供诸如服务台运维和特殊技能专家的功能；提供日常的 IT 服务。与这么多流程规格说明标准一样，ITIL 对这些活动如何实施提供通用指导而不是具体指导。例如，ITIL 不会说“用 X 的目标度量 A”，而是说“为了目标 X，选择一个可以让你达到 X 的度量方法”。

组织的活动应该满足组织的某些战略目标，并且需要设计、实施、监控和改进。DevOps 实践有降低从开发人员提交到生产环境并且快速修复已发现问题的目标，这影响 DevOps 提供的服务类型，并将提供监控和改进这些服务的机制。DevOps 对运维的具体影响取决于组织类型及该组织采取的 DevOps 实践的特殊性。

一种审视 DevOps 和 ITIL 之间的关系的方法是，DevOps 对各种 ITIL 服务提供持续交付，而不要求把这些服务打包进主要发布版本（Major Release）。

3.7 更多阅读材料

ITIL 是一个在 20 世纪 80 年代由英联邦政府开始的一种标准。它经历过了很多次的修订、巩固、改进等。ITIL 的最后版本是在 2011 年发布的。它一共出版了 5 卷 [Cannon

11；Hunnebeck11；Lloyd11；Rance11；Steinberg11]。

Thomas Erl 以面向服务的架构思想写了大量的关于服务的设计问题的文章和书籍，这些需求不仅对软件有用，而且具有更广泛的可应用性。我们把它应用到由运维提供的服务中。请参考他的书《Service Oriented Architecture: Principles of Service Design》[Erl 07]。

讨论 ITIL 和 DevOps 之间的关系的博客有：

- “DevOps and ITIL: Continuous Delivery Doesn't Stop at Software”（DevOps 和 ITIL：软件开发的持续交付不会停）[Spencer 14]。
- “What is IT Service ?”（什么是 IT 服务？）[Agrasala11]。
- FireScope 是一家涉及企业监控的公司：参见博客“What is IT Service ?（什么是 IT 服务？）[FireScope13]。

维基百科中的一篇文章定义恢复点目标（RPO）并将它与恢复时间目标（RTO）进行对比。网址为 http://en.wikipedia.org/wiki/Recovery_point_objective。

第二部分

部署流水线

在这一部分中，我们将关注把代码尽快部署到生产环境的方法同时保证高质量。这些方法视为一个流水线。在这个流水线上，在将代码部署到生产环境前，代码需要通过一个个质量闸门。在部署流水线中，DevOps 的架构方面和过程方面相互交叉。通过本部分的 3 章，我们将涉及以下目标：最小化不同开发团队之间的协作需求；最小化集成不同开发分支所需要的时间；完成高质量的测试集并快速且高质量地把代码部署到生产环境。

在第 4 章中，我们将解释微服务架构，并论证为什么微服务架构满足了很多协作的需求，因此去除了在部署前对显式协作的要求。

对持续部署流水线的需求要求测试是高效的，且仅仅需要完成有限量的合并。我们将在第 5 章中讨论这些问题。

一旦代码是“生产环境就绪”的，那么仍然需要大量的选项才能实际部署该代码。存在一些不同的“要么全部要么一点儿也没有”的部署策略，也存在着一些不同的部分部署策略。一种常见的部署策略会导致一个服务的多个版本同时活跃，这种情况会引起对一致性的质疑。另外，允许任何团队在任何时间都能够进行部署也会导致在客户和他们正在使用的服务之间出现潜在一致性问题。我们将在第 6 章中讨论这些问题以及其他问题。

第4章 整体架构

分布式系统是这样一个系统，你可能不知道但确实存在一台计算机的故障会导致你的计算机无法使用。

——Leslie Lamport

本章开始讲述 DevOps 实践的结构含义。这些实践活动对系统的整体结构以及应用到系统元素中的技术有重大意义。DevOps 目标的一部分是通过使用隐性和不经常的协作代替直接的协作来实现的。我们将讨论正在开发的系统架构是如何作为隐性的协作机制的。我们首先讨论 DevOps 实践是否一定意味着架构变化。

4.1 DevOps 实践是否需要架构调整

你可能已经为当前系统和架构做了大量投资，如果为了采用 DevOps 必须重构系统，我们会问“这样做值得吗?”在本节中我们将看到一些 DevOps 实践是独立于架构的，而为了获得其他的全部好处，可能需要架构重构。

回顾第 1 章中 DevOps 实践的 5 个分类。

1）从需求的角度来看，将运维视为首要干系人。将运维的需求纳入系统可能需要某些架构方面的修改。尤其是，运维需求可能是在日志、监控和支撑事故处理的信息领域上。这些需求和其他修改系统的需求一样，一般情况下可能需要对架构进行小的修改，不是大的修改。

2）确保开发更多地负责相关事故处理。这种改变本身只是过程改变，应该不要求架构修改。但是，和第 1 个分类类似，一旦开发意识到事故处理的需求，就可能对架构做一些修改。

3）要求所有人执行部署过程，包括开发和运维人员。通常情况下，当过程变成强制执行时，某些人可能被要求改变他们正常操作过程，他们工作的系统结构也可能会随之变化。部署过程能被强制执行通常是在每个系统的初始阶段。每个系统，当它被初始化时，它确认系统的承继关系。也就是说，部署的执行是通过一系列步骤来实现的，检查每个步骤都已经发生。更进一步，部署所依赖的系统（例如，操作系统或者中间件）也有可验证的承继关系。

4）使用持续部署。持续部署是这样一个实践，它带来了最深远的架构修改。一方面，组织可以在架构没有大修改的基础上采用持续部署实践。这部分可以参考第 12 章

的案例研究。另一方面，采用了持续部署实践的组织经常开始转向基于微服务的架构。这部分可以参考第13章的案例研究。我们会在本章的剩余部分讨论采用微服务架构的原因。

5）像开发应用程序代码一样开发基础设施代码。这些实践不会对应用程序代码造成影响，但可能影响基础设施代码的架构。

4.2 架构结构总览

在开始探索总体结构的细节前，让我们来明确怎么使用某些术语。术语*模块*和*组件*定义是被频繁重载的，并且在不同的写作中以不同的方式被使用。对我们而言，模块是一个具有一致功能的代码单元。而组件则是一个可执行单元。编译器或解释器将模块转换为二进制文件，构建器将二进制文件转换为组件。因此开发团队直接开发模块。组件是开发团队开发模块的结果，所以也可以说开发团队开发组件，但我们应该清楚组件的开发是开发团队的一个间接活动。

如第1章所述，使用DevOps过程的开发团队通常比较小，并且只有有限的团队间协作。小团队意味着他们开发的组件只有有限的范围。当团队部署一个组件时，除非该组件和与它交互的其他组件都兼容，否则该组件不能直接进入生产。这种兼容性通过多个团队的协作得到保障，或者通过架构的定义可以隐式地得到保障。

组织可以在架构没有大的修改的基础上采用持续部署。例如，第12章中的案例研究就是从根本上不可知的架构。但是，大幅度减少组件进入生产的时间，就需要架构支持：

- 部署而不需要与其他团队的显式协作减少了把组件进入生产的时间。
- 让不同版本的同一个服务同时运行在生产中，可以让不同团队成员无需和该团队的其他成员协作就可以部署。
- 错误发生时执行的回滚能够支持多种形式的现场测试。

*微服务架构*是满足以上需求的架构风格。在实践中，这种模式的架构已经被很多组织采用，或者促进了很多DevOps实践。尽管项目需求可能背离这种模式，但采用DevOps实践的项目仍然保留了这一良好的通用基础。

微服务架构将一组提供少量功能的服务集合整合到一起，这些系统的整体功能来自多个服务。第6章还讨论微服务架构，基于此架构做出的改变能让团队具有不依赖其他团队独立部署服务的能力，能在生产环境中同时部署服务的多个版本，并且能够非常容易地回滚到之前的版本。

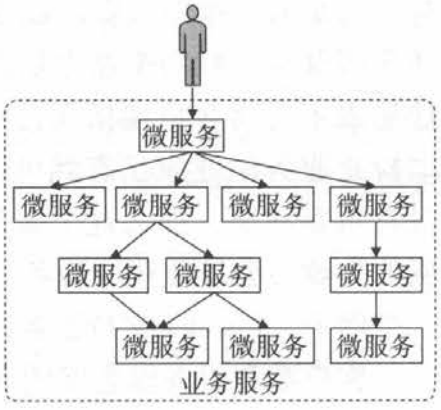


图4-1 通过一个服务依次调用多个服务的用户交互关系

图4-1描述了使用微服务架构的系统框架。用户和一个面向用户的服务交互。相应地，该服务利用了其他服务

的集合。我们使用服务这个术语定义提供服务的组件，使用客户端定义请求服务的组件。一个组件可以是某个交互中的一个客户端，也可以另一个交互中的服务。例如 ,LinkedIn 系统，对于一个用户请求，服务深度可能达到 70 个之多。

由小服务组成的架构是应对小型团队的一种方案。接下来我们将讨论，为了最小化团队间协作的需求，应该全局地指定架构的哪些方面。我们将讨论 3 种设计决策，它们作为基础设施设计的一部分，可以进行全局设计，这样根据这些决策就去除了团队间协作的需求。这三种分类是：协作模式、资源管理和架构元素映射。

4.2.1　协作模式

如果两个服务交互，则负责这些服务的两个开发团队必须通过某种形式进行协作。包含在整体架构中的协作模式的 2 个细节是：客户端如何发现它希望使用的服务；各个服务之间如何通信。

图 4-2 描绘了服务与其客户端之间的交互。服务用注册表进行注册。注册信息包括服务名以及如何调用它的信息，例如，作为 URL 和 IP 地址的终端地址。客户端可以从注册表检索关于服务的信息并使用这些信息调用该服务。如果注册表提供 IP 地址，那么注册表就是本地 DNS 服务器——本地，因为通常情况下，注册表不对因特网开放，只在应用程序的环境内部使用。Netflix Eureka 是一个可以承担 DNS 服务器的云服务注册表。注册表作为一组可用服务的目录，也可以进一步用作执行版本化、所有者关系、服务级别协议（Service Level Agreement，SLA）等。第 6 章将详细讨论注册表的扩展功能。

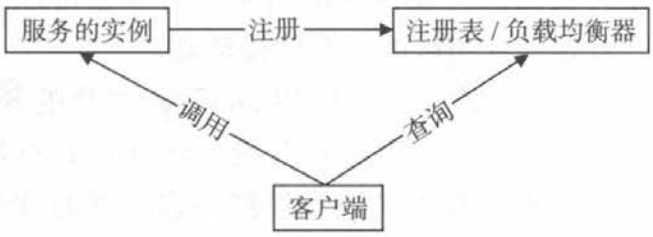


图 4-2　一个服务的实例用注册表给自己注册，客户端向注册表查询该服务的地址并调用该服务

一般情况下，一个服务有多个实例，既是为了支持一个实例无法承载的大的工作负荷，也是为了预防失败。注册表可以轮询已注册的多个实例以实现负载均衡。也就是说，注册表既是负载均衡器也是注册表。最后，考虑服务的实例失败的可能性。这种情况下，注册表不会将客户端指向已经发生故障的实例。通过要求服务周期性地重新注册或者主动检查服务的健康状态就可以防止失败的实例继续提供服务。如果在指定的周期内服务没有重新注册，那么这个服务将从注册表中移除。如果服务存在多个实例，那么一个实例的失败不会导致移除服务。上文提到的 Netflix Eureka 是提供负载均衡功能的注册表的一个例子。Eureka 支持服务周期性地重新注册的需求。

客户端和服务器之间的通信协议可以是任何远程通信协议，如 HTTP、RPC、SOAP 等。这些服务可以提供 RESTful 接口，也可以不提供。远程通信协议应该是不同服务间通信的唯一方法。服务提供的接口细节仍然需要跨团队的协作。稍后，当我们讨论亚

马逊的案例时，我们可以看到提供这种协作的一种方法。我们也可以看到一个明确的需求——限制服务之间的通信仅能使用远程通信协议。

4.2.2 资源管理

两类资源管理决策可以是全局的，并集成到架构中——创建 / 销毁虚拟机以及管理需求的变更。

1. 创建和销毁虚拟机

为了应对客户端需求或者故障，可以创建新的虚拟机。当需求消失时，应该销毁对应的实例。如果实例是无状态的（即，它们在请求之间不保留任何信息），那么一旦新的实例创建了，就可以将它投入服务中。类似地，如果实例中不保留状态，那么销毁它就没什么代价：经过一段冷却期后，实例不再接收新的请求，并响应当前的请求，这个实例就可以被销毁了。因此，冷却期应该足够长，以便实例能响应它收到的所有请求（即积压队列）。如果你因为请求的减少而销毁一个实例，那么积压队列应该是很小的——在其他情形下，这个操作需要非常小心。无状态服务的另一个额外优势是，消息可以发送给服务的任何实例，这样将有助于在实例之间进行负载均衡。

这会导致一个全局决定——在服务实例外维护状态。正如在第 2 章中讨论的那样，大量的应用状态可以维护在持久存储器上，少量的应用状态可以由 ZooKeeper 这样的工具来维护，并且客户端状态至少不应该由服务提供者维护。

确定哪个组件控制创建和销毁服务的实例是另一个重要方面。控制组件有 3 种可能的存在方式。

1）服务本身负责创建或销毁实例。服务了解自己的队列长度和响应请求的性能。它把这些指标和阈值进行对比，如果超过阀值则创建和销毁虚拟机。假设请求的分布是公平的，在某种意义上，在服务的所有实例中，一个特定实例（例如最先的实例）可以决定何时创建和销毁实例。因此，允许服务扩展和收缩能力以满足需求。

2）客户端或客户端链中的一个组件可以负责创建或销毁服务的实例。例如，基于需求，客户端可能知道它将很快向服务提交超过其预先设定阈值的请求，因此客户端可以主动创建服务的新实例。

3）一个外部组件监控服务实例的性能（例如，CPU 负载），在负载达到设定的阀值时它能创建或销毁实例。和 CloudWatch 监控系统协作，亚马逊的自动伸缩组可以提供这样的能力。

2. 管理需求

一个服务的实例数反映了来自客户端请求的数量。我们刚才讨论了创建和销毁实例的几种不同方法，以及这些方法怎么管理需求的不同假设情况。

- 一种管理需求的方法是监控性能。另一种策略是怎么实现监控（例如，在每个服

务实例内部运行监控代理，或者在外部使用特殊组件进行监控）。当需求增长到需要采取措施的地步时，临时创建一个新实例。需要花费时间创建新实例，所以重要的一点是，指示器应该是及时的，甚至应该是预测性的，并为虚拟机的创建留下足够的时间。第 7 章会讨论监控的细节。

- 另一个可能的技术是使用服务级别协议控制实例的数量。服务的每个实例通过它的服务级别协议保证它能够在指定的延迟下处理一定数量的请求。然后这个服务的客户端能知道在指定的延迟条件下，它们能发送多少请求且能获得多少响应。这个技术有许多约束。首先，客户端强加到服务的请求会依赖于强加到客户端的需求，所以会出现一个需求链瀑布效果。这个瀑布可能导致服务级别协议的规格说明和实现性变得不确定。其次，服务的每个实例知道它能处理多少个请求，但是客户端可以有服务的多个可使用的实例。因此，准备组件必须要知道你的服务当前存在多少个实例。

4.2.3　架构元素之间的映射

架构中可以指定的最后一种协作决策是架构元素之间的映射。我们讨论两种不同类型的映射——工作指派和分配。这两种映射都是全局的。

- 工作指派。一个团队可以在多个模块上工作，但让多个开发团队工作在相同模块上需要这些开发团队之间的大量协作工作。因为协作花费时间，所以更简单的一种结构是将单个团队的工作打包成模块并在开发模块之间的接口，以允许不同团队开发的模块相互操作。实际上，David Parnas 在 20 世纪 70 年代对模块的原始定义就是对一个团队的工作指派。虽然不需要，但每个组件（即微服务）一个开发团队负责是合理的。也就是说，当链接时，构成一个组件的一组模块是一个开发团队的输出。这没有阻碍一个团队负责多个组件，但是它意味着涉及一个组件的任何协作都限制在一个团队中，涉及多个开发团队的任何协作是跨组件的。考虑到我们讨论的架构约束集，跨团队协作的需求是有限的。
- 分配。每个组件（即微服务）以独立的可部署单元存在。这允许将每个组件分配给一个（虚拟）机器或容器，或者允许将多个组件分配给一个（虚拟）机器。一个微服务的重新部署和升级对其他微服务没有影响。第 6 章将展开研究这部分内容。

4.3　微服务架构的质量

我们已经讨论了架构模式（微服务架构），通过全局架构选择，它减少了团队间的协作需求。这种模式对可靠性（无状态服务）和可修改性（小型服务）的质量提供一定的支持，但是团队还可以采用其他的实践来提高他们服务的可靠性和可修改性。

4.3.1 可靠性

可靠性问题的3个根源：数量不多的团队间协作、环境正确性，以及服务实例失败可能性。

1. 数量不多的团队间协作

数量有限的团队间的协作可能导致开发客户端的团队与开发服务的团队在接口语义方面产生歧义。特别是，对服务的一个意外输入或者从服务产生一个意外输出都可能发生。有很多可供选择的改进方案。首先，团队应该实践一些防御性编程，而不是假设服务的输入或者调用服务的结果都是正确的。检查值的合理性有助于尽早检测错误。提供一个丰富的异常集将能快速确定错误的原因。其次，应该审慎地对所有或者大部分微服务进行集成和端到端测试。由于涉及大量微服务和真实的外部资源，频繁执行这些测试是非常昂贵的。使用一种叫作消费者驱动契约（Consumer Driven Contract，CDC）的测试实践可以缓解这个问题。也就是说，测试微服务的测试用例是由该微服务的所有消费者决定且共同拥有的。对CDC测试用例的任何修改需要得到消费者和微服务的开发者的一致同意。作为集成测试的一种形式，执行CDC测试用例比执行端到端测试用例代价要小一些。如果恰当地实践了CDC，那么可以非常信任这个微服务而不需要执行很多端到端测试用例。

CDC是一种协作的方法，隐含着应该如何随时构建和进化微服务的用户故事。消费者和微服务开发者共同构建和拥有用户故事。CDC定义成为将功能分配给微服务的函数，它由作为协作一部分的服务所有者来管理，它定义了下一个迭代，因此，它不会推迟当前迭代的过程。

2. 环境的正确性

在从单元测试到后期生产的过程中，服务将在多个不同环境中运行。每个环境都是通过代码和配置参数集合预先提供和维护的。代码和配置参数的错误是非常常见的。不一致的配置参数也是可能的。因为基于云的基础设施中的不确定性程度，所以即使执行正确的代码和配置也可能导致错误的环境。因此，服务的初始化部分应该测试它的当前环境以判断它是否符合要求。同样，也应该测试配置参数，尽可能检测不同环境中的意外的不一致性。如果服务的行为依赖于它的环境（例如，某些动作在单元测试阶段执行，而不在生产阶段执行），那么初始化阶段应该定义环境，并提供设置来开启或者关闭这个行为。DevOps中的一个重要趋势是像管理应用程序代码一样引入版本控制和测试来管理设置环境的代码和配置参数。这是在第1章中定义的“基础设施即代码”的例子，在第5章中还会详细讨论。基础设施代码的测试是一个相当有挑战的问题。我们会在第7章和第9章中讨论这些问题。

3. 实例故障

实例都有可能出现故障。直接部署或通过虚拟技术将实例部署到物理机器上。在大

型数据中心，物理机器的故障是很常见的。客户端检测服务实例故障的标准方法是通过请求是否超时来判断。一旦发生了超时，客户端可以重新发起一个请求，取决于使用的路由机制，假设请求将被路由到不同的服务实例上。如果收到多次超时，则认为这个服务出现故障，可能尝试替代的操作来实现期望的目标。

图 4-3 展示了客户端尝试访问一个有故障服务的时间轴。客户端向服务发起一个请求，请求超时。客户端继续发起请求，请求又超时。此时，知道故障发生了 2 次超时间隔。短暂的超时间隔（快速失败）可以启动客户端请求服务的客户端的快速响应机制。然而，短暂的超时间隔可能引发服务实例响应变慢的假象。结果可能是这些请求都实际交付了服务，但可能并不是及时交付的。另一个结果是，执行了替代动作。设计时服务需要考虑同一个服务的多次调用不会引入错误。幂等（Idempotent）是服务的一种机制，允许服务被相同的输入重复调用，同时产生相同的输出——换言之，没有产生错误。

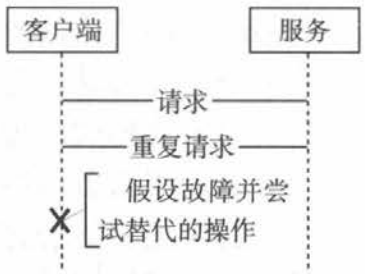


图 4-3　识别依赖服务的故障的时间轴 [标注法：UML 序列图]

图 4-3 中服务的替代操作被高亮标注了。也就是说，如果服务失败，客户端可以执行替代操作。图 4-3 没有展示如果没有提供替代操作会发生什么情况。这种情况下，服务向客户端报告失败和上下文信息——也就是，从特定的底层服务没有获得响应。在第 7 章将深入探索报告错误的话题。

4.3.2　可修改性

使服务具有可修改性意味着让可能的变化简单化，并减少变化带来的连锁反应。在两种情况下，使服务具有更灵活的可修改性的方法有：1）封装可能变化的影响部分；2）封装可能引起变化的连锁反应的交互部分。

1. 识别可能的变化

来自开发过程而不是提供的服务的某些可能的变化包括：

- 服务执行所在的环境。模块在一个环境中通过单元测试，在另一个环境中进行集成测试，在第三个环境中进行验收测试，在第四个环境中投入生产。
- 与你的服务交互的其他服务的状态。如果其他服务正处于开发阶段，那么这些服务的接口和语义可能变化很快。因为你可能不知道外部服务的状态，所以安全的实践是，尽可能把与外部服务之间的所有通信都看作可能发生变化。
- 你的服务所使用的第三方软件和库的版本。第三方软件和库可能随意变化，有时会中断你的服务。我们听过一个案例，外部系统在部署过程中删除了一个核心接口。在不同的环境中使用相同的虚拟机镜像可以防止虚拟机内部的改变，但没法

对抗外部系统的改变。

2. 减少连锁反应

一旦发现可能的变化，你应该防止这些类型的变化在你的服务中产生连锁反应。典型的做法是引入某些单一目的模块，这些模块的作用是本地化和隔离对环境、对其他服务、对第三方软件或库的改变。你服务的其他部分通过新引入的带有稳定接口的模块与这些可变的实体进行交互。

与其他服务的任何交互，例如，都是通过特定模块作为中介的。其他服务的改变反映在这个中介模块上，并可以缓冲连锁影响到你服务的其他部分。其他服务的语义变化，事实上，可能产生连锁反应，但中介模块可以吸收一些影响，因此减少这种连锁反应带来的后果。

4.4 团队的亚马逊规则

第 1 章提到，亚马逊有一条规则：没有团队应该大到需要分食两个比萨。在 21 世纪早期他们采用一个内部微服务架构。与采用的微服务相关联的是，如何使用这些服务所遵循的规则列表：

- 所有团队通过服务接口开放他们的数据和功能。
- 团队必须通过接口与其他团队通信。
- 不允许其他形式的服务间 / 团队间的通信：没有直接链接、不能直接访问其他团队的数据存储、没有共享内存模式、没有后门等。允许的唯一通信是通过网络上的服务接口调用。
- 不用关心他们（其他服务）使用了何种技术。
- 所有服务接口，无一例外，都要在一开始就设计成可以外部化的。也就是说，团队必须计划和设计能够将接口对外部开发者开放。

每个团队产生一定数量的服务。每个服务被完全封装起来，除了公共接口。如果另一个团队想使用服务，它必须先找到接口。接口的文档必须包含足够的语义信息，以使服务的使用者可以准确地确定条目（如“客户”或者“地址”）的正确定义。在一个组织的不同部门中这些概念有时有不同的意思。关于接口的语义信息可以保存在前面描述过的注册表或负载均衡器中，假设语义信息是机器可读的。

通过让每个服务都具有潜在的外部可用性，是否提供一个全局可用的服务还是让它本地可用，是业务决策，而不是技术决策。通过库，外部服务可以隐藏在应用程序编程接口（API）的后面，因此这个需求不是用于接口技术的预判断。

使用这套规则后，亚马逊拥有了大量的服务集合。他们销售业务的一个网页使用了超过 150 个服务。可伸缩性由每个服务单独管理，包含在它的服务级别协议中（针对特定

的工作负荷设定可保证的响应时间）。合同包括服务保证什么样的需求等级。服务级别协议绑定了客户端和服务器端。如果客户端的需求超越了服务级别协议中承诺的工作负荷，则响应速度慢就变成客户端的问题，而不是服务端的问题。

4.5　现有系统的微服务方案

尽管微服务反映了 DevOps 的小的而独立的团队原理，但是大多数组织有不是这么架构的大型关键任务系统。这些组织需要决定是否将他们的架构迁移到微服务构架，哪些进行迁移。第 10 章将讨论迁移。以下内容是一个架构师在采用微服务架构时需要确认的：

- 在需求规格说明时考虑运维关注的部分。
- 被开发的系统整体结构应该是一个小的、独立的服务组成的集合。
- 对于客户端和其他需要的服务，每个服务都是不可信的。
- 需要定义和理解团队角色。
- 服务需要注册到本地的注册表 / 负载均衡器。
- 服务必须周期性地刷新它们的注册信息。
- 服务需要给客户端提供服务级别协议。
- 服务要设计成无状态且短暂的。
- 如果服务必须维护状态，则它应该在外部持久存储上维护状态数据。
- 如果服务所依赖的其他服务发生了故障，则需要备用方案。
- 服务有防御检查，以便拦截来自客户端的错误输入或其他服务的输出。
- 外部服务、环境信息、第三方软件和库的使用都应该本地化（例如，它们要求通过特定于那个外部服务、环境信息、外部软件或库的模块）。

然而，采用微服务架构会引入新的挑战。当一个应用由许多网络连接的微服务组成时，可能会有延迟和其他性能问题。需要很仔细地设计服务之间的认证和授权，以确保它们不会增加不可忍受的开销。监控、调试和分布式追踪工具需要调整以适配微服务。如前所述，端到端测试是昂贵的。你很难不依赖过去的组件和现有的数据而从头开始重新构建应用。

没有数据丢失和中断，立即从当前架构迁移到微服务架构是一个挑战。迁移过程中可能需要创建一个临时解决方案。第 13 章将讨论这些挑战并学习 Atlassian 案例研究，其中 Atlanssian 描述了他们在转向微服务架构中的一些初始步骤。架构师在执行迁移过程中应该有一个考虑的检查清单。

4.6　小结

实现减少在不同团队中协作的 DevOps 的目标可以使用微服务架构模式，其中协作

机制、资源管理决策、架构元素的映射都是通过架构来指定的，因此可以最小化团队间协作。

可以将开发实践活动加入微服务架构模式，以实现可靠性和可修改性，例如识别和隔离可能变化的区域。

采用微服务架构模式会在监控、调试、性能管理和测试等方面引入额外的挑战。从现有架构向微服务架构模式迁移需要非常认真地规划和努力。

4.7 更多阅读材料

关于软件架构的更多信息，我们推荐以下参考书目：

- 《Documenting Software Architectures, 2nd Edition》[Clements 10]。
- 《Software Architecture in Practice, 3rd Edition》[Bass 13]。

服务描述、分类和管理在《Handbook of Service Description》[Barros 12] 中有详细描述。这本书中描述的服务是外部可见的，不是微服务，但很多讨论都和微服务相关。

微服务架构模式在《Building Microservices: Designing Fine-Grained Systems》[Newman 15] 中有描述。

很多组织已经实践了微服务架构开发第一个版本和 DevOps 的某个版本，也分享了他们有价值的经验。

- 你可以在以下网址阅读到亚马逊的更多案例：
  - http://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/
  - http://www.zdnet.com/blog/storage/soa-done-right-the-amazon-strategy/152
- 在 Netflix 也分享了很多他们使用微服务架构的挑战 [Tonse 14]。

Eureka 的 Netflix 实现（这是他们的开源内部负载均衡器 / 注册表，可以在 https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance 上找到具体内容。

消费者驱动契约（Consumer Driven Contract，CDC）可以参见 Martin Fowler 的博客“Consumer-Driven Contracts: A Service Evolution Pattern”（消费者驱动契约：服务演进模式）[Fowler 06]。

第 5 章　构建与测试

测试带来故障，故障带来理解。

——Burt Rutan

5.1　概述

虽然架构师喜欢把重点放在设计与实现上，但是由于一些原因，用于支持开发与部署过程的基础设施也很重要。这个基础设施应该支持下面的需求：

- 团队成员可以并行地在系统的不同版本上工作。
- 一个团队成员开发的代码不会在无意中覆盖另一个团队成员开发的代码。
- 如果一个团队成员突然离开团队，工作不会丢失。
- 团队成员的代码易于测试。
- 团队成员的代码可以容易地与同一个团队的其他成员编写的代码集成在一起。
- 一个团队开发的代码可以容易地与其他团队开发的代码集成在一起。
- 系统的集成版本可以容易地部署到不同环境中（例如，测试环境、预发布环境（staging）、生产环境）。
- 系统的集成版本可以在不影响系统生产的版本的情况下容易且完整地测试。
- 可以对系统最近部署的新版本进行密切监控。
- 一旦代码放入生产环境，如果出现问题，能够找到旧版本的代码。
- 万一出现问题代码可以回滚。

实践中的架构师为什么应该关注开发和部署的基础设施，最重要的原因是，他们或项目经理需要确保开发的基础设施能够满足上述需求。对结果负责，没有什么比这更值得关注了。

虽然用于支持这些任务的工具这些年来已经进化且更加先进了，但上述需求没有一个是新的。我们使用*部署流水线*的概念来组织本章内容。如图 5-1 所示，在确保高质量的前提下，部署流水线包含了开发人员提交代码与代码实际投入正式生产环境之间的步骤。

当开发人员把代码提交到联合版本控制系统时，部署流水线开始。在提交之前，开发人员将不得不在他们的本地环境中执行一系列提交前（pre-commit）测试。无法通过提交前测试当然也就意味着无法提交。然后提交操作触发正在开发的服务的集成构建。这

个构建由集成测试进行测试。如果这些测试成功了，那么构建就提升到准生产环境（预发布环境），在这个环境中再测试一次。然后提升到生产环境并进行密切监控。在经过一段时间的密切监控后，再提升到正式生产环境中。在不同的组织中具体任务可能稍有不同。例如，小公司可能没有预发布环境，或者对最近部署的版本没有特别的监控。大公司针对不同的目的可能有多个不同的生产环境。我们在第 6 章描述这些不同的生产环境。

定义*持续集成*的一种方法是在一个阶段与下一阶段之间有一个自动触发器，直到集成测试。即，如果构建成功，则触发集成测试。如果不成功，则通知对失败负有责任的开发人员。*持续交付*定义为，使用自动触发器，直到预发布系统。这是图 5-1 中标记为用户验收测试 / 预发布环境 / 性能测试的方框。我们用术语*预发布*（staging）描述这些不同的功能。*持续部署*意味着到最后一步（即部署到生产系统）也是自动化的。在将服务部署到生产环境之后会密切监控一段时间，然后将它升级到正式生产系统。在这最后一步，监控和测试仍旧存在，但是这个服务在这方面与其他服务并无区别。本章中，我们关注这个流水线的构建与测试方面的内容。第 6 章描述部署实践，第 7 章讨论监控方法。

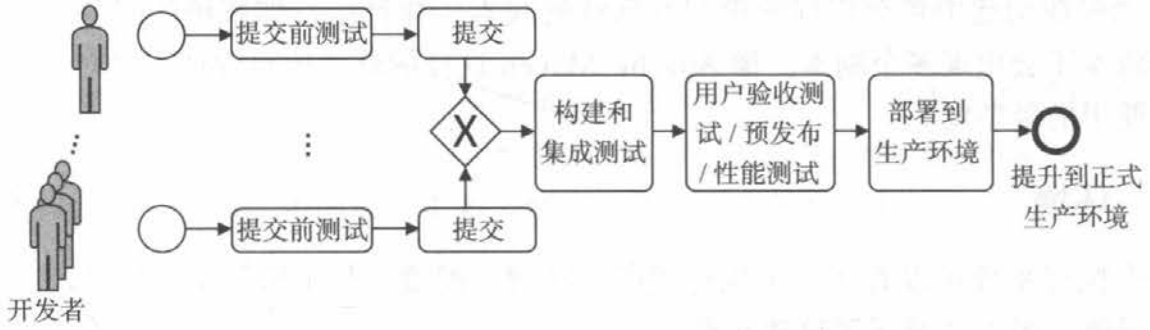


图 5-1　部署流水线 [标注法：业务流程建模标注]

我们使用部署流水线组织本章的主题。然后我们讨论不同步骤的横切关注点，然后讨论提交前阶段、构建与集成测试、用户验收测试 / 预发布 / 性能测试、生产环境及生产后期。不过，在讨论之前，首先讨论系统在流水线中的移动。

5.2　在部署流水线中移动系统

提交的代码按照图 5-1 所示的方式移动，但是代码不是自主移动的。而是通过工具移动的。这些工具是由它们的程序（这里称为*脚本*）或开发人员 / 运维人员的命令控制的。本节关注这个移动的两方面：

- 可追溯性。
- 与流水线中每个步骤相关的环境。

5.2.1　可追溯性

可追溯性意味着，对于生产中的任何系统，可以准确地确定它如何进入生产环境。

这意味着不仅要跟踪源代码，而且还要跟踪作用于系统元素的所有工具的所有命令。单独的命令很难跟踪。为此，通过脚本的控制工具比通过命令的控制工具好很多。脚本及相关的配置参数应该像应用程序代码一样进行版本控制。一种叫作*基础设施即代码*（Infrastructure-as-code）的移动采用的就是这个原理。测试也是通过版本控制系统维护的。配置参数可以像文件一样保存在版本控制系统中，或者通过专用配置管理系统保存。

把基础设施作为代码看待意味着这种代码也应该进行像应用程序源代码一样的质量控制。即，代码应该进行测试，对代码的变更应该以某种方式进行管理，不同的部分应该有各自的负责人。

把所有的东西都放入版本控制或配置管理系统让你能够在任何一个地方重新创建准确的环境，从本地环境到生产环境。这不但对追踪问题很有帮助，而且能够在新环境中快速且灵活地部署应用程序。

要求把所有东西都纳入版本控制的一个复杂之处是如何处理像 Java 库这样的第三方软件。这些库可能很庞大，并消耗很多存储空间。库也会发生变化，所以必须找到一种机制，确保在构建中包含正确的第三方软件的版本，并且运行你系统的服务器上的库的同一个版本不会出现多个副本。像 Apache Maven 这样的软件项目管理工具对于将有助于管理库使用的复杂度。

5.2.2　环境

一个执行系统可以看作一组执行代码、环境、配置、与主系统交互的环境之外的系统、数据等。图 5-2 显示了这些元素。

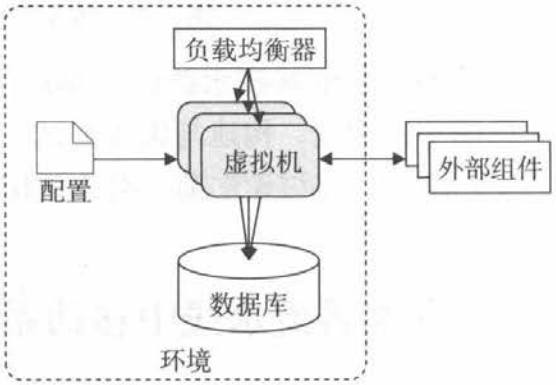


图 5-2　环境的一个例子 [标注法：架构]

当系统通过部署流水线移动时，这些内容一起工作产生所需的行为或信息。

- *提交前*。代码是开发人员正在开发的系统的模块。把这个代码构建到可以测试的内容，需要访问由其他开发人员创建的版本控制度中的合适部分。在第 1 章中，我们讨论讨论了减少团队间的协作。提交前需要团队内部的协作。这个环境一般是笔记本或台式机，外部系统使用存根或模拟程序，并且测试时只使用有限的数据。只读的外部系统，例如 RSS Feed，可以在提交前阶段中进行访问。配置参数应该反映环境，并控制调试级别。
- *构建与集成测试*。环境通常是一个持续集成服务器。对代码进行编译，构建组件并烧制为虚拟机镜像。镜像可以是重度烧制，也可以是轻度烧制（关于打包，请参见后面的章节）。这个 VM 镜像在流水线后续工作中不会改变。在集成测试中，一组测试数据形成了测试数据库。这个数据库不是生产数据库，相反，它包含了

足够多的数据以便执行与集成相关的自动化测试。配置参数把构建的系统与集成测试环境相连接。

- 用户验收测试 / 预发布 / 性能测试。这个环境尽可能地接近于生产环境。运行自动化验收测试，通过人工生成的工作负荷进行压力测试。数据库应该包含真实生产数据的一个子集。对于非常大的数据集，它可能无法包含真实数据的所有副本，但这个子集应该足够大，以便测试能够在真实环境中运行。配置参数把需要测试的系统与更大的测试环境相连接。不允许从预发布环境访问生产数据库。
- 生产。生产环境应该访问实时数据库，并有足够的资源以便适当地处理其工作负荷。配置参数将系统与生产环境相连接。

这些环境的每一种配置都是不同的。例如，在开发环境中的日志通常比生产环境中的日志记录更详细的内容。这样做有助于开发人员发现缺陷，产生的性能开销也没有那么严重。另一个例子关注凭证：访问生产资源的凭证，如实时客户数据库，是不应该让开发人员得到的。虽然不可避免地对配置做一些变更，但重要的是让这些变更最小化，以防止对系统行为的影响。因此，从生产系统中使用大量不同的配置进行测试是没有帮助的。

维基百科提供的环境列表比这里提供的长很多，因为它枚举了更多不同的测试环境。对于讲述本章内容来说，我们列举的环境已经足够了，但如前所述，根据组织的规模、监管环境以及其他因素，可能需要更多的环境。维基百科的列表是：

- 本地：开发人员的笔记本 / 台式机 / 工作站。
- 开发：开发服务器，也称为沙盒。
- 集成：持续集成（Continuous Integration，CI）构建目标，或者用于开发人员测试副作用。
- 测试 / 质量保证（Quality Assurance，QA）：用于功能、性能测试、质量保证等。
- UAT：用户验收测试。
- 预发布 / 生产前：生产环境的镜像。
- 生产环境 / 实时：服务于终端用户 / 客户。

在转到部署流水线的实际步骤之前，先讨论测试的横切关注点。

5.3　横切关注点

本节讨论部署流水线各个横切关注点：测试框架、负面测试、回归测试、错误的可追溯性、组件的大小以及环境的拆解。

- 测试框架。测试框架是一组软件及配置的测试数据，通过在各种条件下运行它，并对它的行为及输出进行监控，以便测试一个程序单元。测试框架是必需的以便自动测试。测试框架的一个关键特征是，它产生报表。特别是，它应该至少识别

出哪个测试失败了。本章讨论的大部分类型测试都应该能够由测试框架自动实现和驱动。

- **负面测试**。大多数测试都遵循“快乐路径”，在关于环境的所有假设都满足且用户使用正确的输入按照正确的顺序执行操作时，检查系统的行为是否符合预期。当这些假设条件不满足时，测试系统行为是否符合预定也是很重要的。为这个目的所做的测试统称为负面测试。这样的例子（模拟或实际的）包括用户以错误的顺序执行操作（如单击按钮、调用命令、在意外时间终止用户界面（User Interface，UI）/ 浏览器等）或者模拟的连通性问题，如外部服务变得不可用、在意外时间连接意外终止等。常见的期望是应用程序能够**优雅地降级或失败**（例如，根据实际问题的需要只降低功能），并且，如果故障是不可避免的，提供有意义的错误信息，并以可控的方式退出。
- **回归测试**是在通过最初的测试后维护并重新运行测试的根本原因。根据维基百科，回归测试“……在对系统现有的功能及非功能领域进行改进、打补丁或配置更改等变更后力求发现新的软件缺陷，或者叫作**回归**。”回归测试的另一个用途是确保已修复的缺陷以后不会再次引入。在修复测试驱动开发环境中的缺陷时，一种好的做法是增补测试套件，该测试套件把重新引入缺陷的测试增加进去。在当前版本中，这个测试应该是通不过的。在修复缺陷之后，该测试应该通过。有可能自动创建回归测试：在部署流水线中的后期检测到的失败（例如，在预发布测试中）可以自动记录，并作为新的测试添加到单元或集成测试中。
- **错误的可追溯性**（也称为谱系或种源形式）。如果生产环境中出现缺陷，需要能够尽快找出运行的是哪个版本的源代码，这样才能检查并重现缺陷。我们归纳了处理这种情况的两个选项，这两个选项都假定，有一套改变系统的机制，它可能是通过Chef、重度烧制的镜像等的配置管理系统。无论是哪一种变更机制，都假设每一个有效的变更都使用这个机制。理论上，与那种实践产生偏差（deviation）是有可能的，也许是不小心，也许是有意为之，需要事先有一套机制来检测或预防它们。
  - 允许可追溯性的第一个选择是把起标识作用的标签与打包的应用程序关联起来，例如用于说明起源的各部分软件和脚本的提交ID。在Java和.NET环境中，包可以通过元数据来丰富，其中可以深加提交ID。容器或虚拟机镜像可以以类似的方式丰富。可以将信息添加到日志行中，并且这些信息可以（有选择地）交付到中央日志库中。通过这种方式，一个有故障的虚拟机不会破坏用于分析该故障的信息。
  - 另一个选择是把生产环境中每个机器的起源放到一个外部配置管理系统中。这种系统的一个例子是Chef，它可以跟踪它在用于一台机器所做的所有变更。这种方法的优点是，任何一个需要访问起源信息列表的应用程序都知道这些信息的位置。缺点是，当不使用提供这种功能的一个现有系统时，保持集中列表最

新可能是一个复杂工作。

- 小组件。我们在第 1 章提到过，小团队意味着小组件。在第 4 章中，我们把微服务作为小组件的一种表现加以讨论。小组件也容易分别进行测试。小组件的路径少，接口和参数一般也比较少。“小”的结果就意味着小组件易于测试，需要的测试用例也少。但是，如第 1 章所述，“小”也会额外增加集成方面的难度，并且因为包含了较多的组件，所以也需要进行端到端测试。
- 环境拆解。当一个环境不再用于某个特定目的时，如预发布测试，就应该将它拆解。拆解环境的一个原因是释放与此环境相关的资源，另一个原因避免无意间与这些资源进行交互。第 12 章的案例研究明确把拆解作为过程的一部分。在资源的目的已经实现后，很容易失去对资源的跟踪。出于安全的目的，每个虚拟机都必须打补丁，不使用、失去跟踪的资源为恶意用户提供了一个攻击点。我们将在第 8 章讨论一个不使用的虚拟机被人利用的例子。

我们将在本章的后面更详细地讨论测试。现在，你应该理解不同的环境可以用于不同类型的测试；成功完成的测试越多，你对版本质量的信心就越高。

5.4　开发及提交前测试

提交步骤前的所有任务都是由每个开发人员在他们的本地机器上执行的。这里不讨论代码开发及语言选择。我们讨论版本与分支、功能开关、配置参数以及提交前测试的一般主题。

5.4.1　版本控制与分支

当今，即使一个小的开发项目也要放入版本控制系统中——也叫作修订控制或源代码控制。这样的系统可以追溯到 20 世纪 50 年代的手工系统。并行版本控制（Concurrent Version System，CVS）可以追溯到 20 世纪 80 年代，SVN（Subversion）可以追溯到 2000 年。Git（2005 年发布）是当前一个流行的版本控制系统。版本控制的核心功能是：能够识别源代码不同的版本；不同开发人员之间共享代码修订；记录从一个版本到下一个版本之间谁做了修改；记录修改范围。

CVS 和 SVN 是集中式解决方案，其中每个开发人员都从中央服务器上检查代码，然后将变更提交回这台服务器。Git 是一个分布式版本控制系统：每个开发人员都有包含所有内容的 Git 存储库的一个本地克隆（或副本）。提交是在本地存储库完成的。一系列变更可以通过中央服务器进行同步，其中服务器的变更与本地存储库同步（使用 pull 命令），本地变更可以提交给服务器上使用（使用 push 命令）。只有当本地存储库是最新的时才能执行 push 命令，所以在执行 push 命令前一般先执行 pull 命令。在执行 pull 命令过程中，对同一个文件（例如，同一个 Java 类）所做的变更会自动合并。不过，这个合并可能失

败，此时开发人员必须在本地解决所有的冲突。从（自动或半手工）合并得到的结果变更将在本地提交，然后 push 到服务器上。

几乎所有的版本控制系统都支持创建新分支。分支本质上就是存储库（或一部分）的一个副本，允许两个或多个工作流的独立演化。例如，如果一部分开发团队正在开发一组新功能，而生产环境中运行着以前的版本并在生产系统中发现了一个致命错误，那么生产环境中的当前版本必须修复。实现方式可以是以发布到生产环境的代码的版本为基础，创建一个用于修复的分支。在错误得到修复且修复的版本发布到生产环境后，这个用于修复缺陷的分支一般会合并回主分支（也称为主干、主线或主分支）。

如果需要强调我们前面所讨论的可跟踪性，这个例子就有用了。为了修复错误，需要确定执行哪些代码（代码的可追溯性）。这个错误可能是配置的问题（配置的可追溯性）或者是用于部署到生产环境的工具套件的问题（基础设施的可追溯性）。

虽然分支结构有用并且很重要，但使用分支时存在两个问题：

1）分支可能太多，你不知道特定任务应该在哪一个分支上完成。图 5-3 显示了有多个分支的分支结构。在这种结构中，很难确定某个特定的变更应该在哪一个分支上完成。为此，短暂的任务不应该创建新的分支。

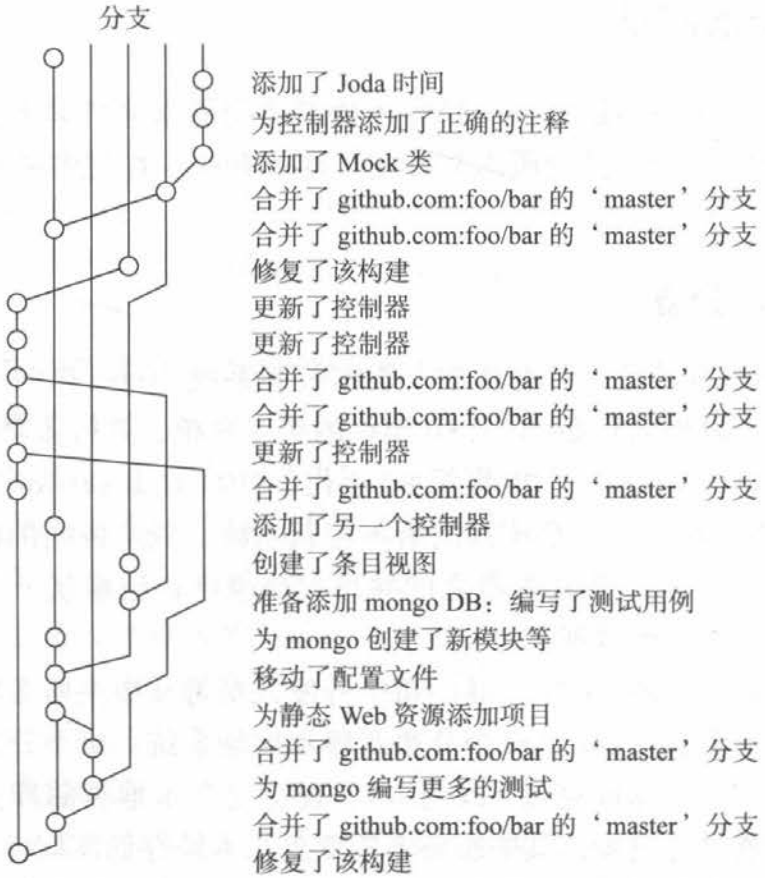


图 5-3　一个有 20 名开发人员的短暂项目的 Git 历史，显示了很多合并操作（改编自 http://blog.xebia.com/2010/09/20/git-workflow/）[直线代表不同的分支，斜线代表分叉或合并]

2）合并两个分支可能是困难的。不同的分支同时演化，并且，开发人员经常修改了代码的许多不同的部分。例如，为了修复缺陷、保护版本不受最近发现的漏洞的影响或者支持迫切需要的变更，几个开发人员可能同时对生产环境中的现有版本进行修改。同时，多组开发人员可能正在开发一个新发布，每个小组都在一个不同的功能分支上工作。到了开发周期将要结束时，需要把所有功能分支合并在一起，并包括维护前一个发布所做的改动。

分支结构的一个替代方案是让所有开发人员都在主干上直接工作。在每次提交时开发人员处理集成问题，而不需要重新集成一个大的分支，这个解决方案要简单一些，但是与使用分支相比，它需要更频繁的操作。Paul Hammant 讨论了 Google 如何使用这一技术。Google 的开发是全面基于主干的：15 000 名开发人员向主干提交，每天平均有 5500 次提交，每天运行 7500 万个测试用例。

所有开发工作都在一个主干上的问题是，开发人员可能同时在同一模块中的多个不同任务上开发。在一个任务结束时，该模块还不能提交，直到另一个任务也完成才能提交。这种做法会在部署流水线中的新功能中引入不完整或未测试的代码。为了解决这个问题，提出了功能开关。

5.4.2　功能开关

功能开关（也称为*功能标志*或*功能切换*）是围绕着不成熟代码的一个“if”语句。程序清单 5-1 显示了一个例子。在源代码中，还不能用于测试或生产的新功能将被禁用，例如，设置一个全局布尔变量。当这个功能就绪后，开关可以打开，各自的代码就可以使用了。常见的实践是把功能的开关放到配置文件中，这是下一节要讲述的主题。功能开关允许你持续地交付新发布，这些版本可以包含未完成的新功能——但这些不会影响应用程序，因为这些新功能还处于关闭状态。只有当这些功能可以发布并且成功地通过了所有所需的测试后，生产环境中的开关才会打开（即打开这个功能）。

程序清单 5-1　使用功能开关的伪代码

```
If (Feature_Toggle) then
      new code
   else
      old code
   end;
```

我们将在第 6 章中讨论功能开关的另一个用途。

但是，功能开关有一定危险。在第 1 章中讨论了骑士资本集团的案例。那个在 45 分钟损失超过 4.4 亿美元的事件中，涉及对功能开关的错误使用。多年前使用过的一个开关的名字在最新的版本中被重复使用，但在旧版本中，它代表其他的含义。因为在开关打开时，有一台正在运行旧版本的生产服务器，所以 4.4 亿美元就这样没有了。教训 1：不

要重复使用开关名称。教训 2：尽可能快地集成功能并去掉开关测试。

当有许多功能开关时，管理它们就变得复杂了。一个有帮助的做法是建立一个专用工具或库，它能够掌握系统中所有的功能开关，知道它们的当前状态，能够改变它们的状态，并能最终从代码基中去除功能开关。

5.4.3　配置参数

配置参数是一个能够更改系统行为的外部设置变量。配置设置可以是：希望展示给用户的语言、数据文件的位置、线程池大小、屏幕上背景的颜色或功能开关设置。如你所见，潜在的配置参数清单是无穷无尽的。

为了本书的目的，我们关心的配置设置或者控制系统与其环境的关系，或者控制与系统目前运行的部署流水线中的阶段相关的行为。

配置参数的数量应该保持在一个可管理的水平。较多的配置参数通常会导致它们之间的复杂联系，多个参数之间的兼容设置集也只有几个软件配置专家才知道。虽然灵活性是我们期望的目标，但太复杂的配置意味着你基本上是在创建一个专属的编程语言。例如，SAP 商业套件一度有成千上万个配置参数。虽然这种灵活性让很多公司可以在他们的环境中使用软件，但这也意味着只有专家团队才能做出正确的设置。

当今大部分的编程语言都有很好的库，提供相对健壮的配置处理。这些库的操作包括：检查值是否已经指定了（或者否可以使用默认值）并且格式和范围是否正确；确保 URL 是有效的；甚至可以检查配置是否与多个配置选项兼容。

可以按照使用时间把配置参数分组，例如，在构建时、部署、启动或运行时是否考虑它们。任何一个重要的选项都应该在使用前检查。URL 及其他对外部服务的引用应该在启动阶段都再次检查，以确保它们从当前环境是可达的。

对配置参数要做的一个决策是，那些值在部署流水线的不同步骤中是否应该是相同的。如果生产系统的值是不一样的，你还必须确定它们是否必须保密。这些决策分为 3 类。

1）多个环境中的值是相同的。功能开关和性能相关的值（例如，数据库连接池大小）在性能测试 / 用户验收测试 / 预发布和生产环境中都应该是相同的，但是在本地开发人员机器上可能是不同的。

2）取决于环境，值是不同的。在生产环境中运行的虚拟机的数量可能比测试环境中的多。

3）值必须是保密的。访问生产数据库或更改生产基础设施的凭证必须保密并且只能在需要访问它们的人之间共享——没有哪个具有一定规模的组织敢冒险让一个开发实习生把客户数据带走。

对配置参数的值保密，为部署流水线带来一定的复杂性。总体目标是让这些值是生产环境中现在使用的值，并对它们进行保密。一个技术是给部署流水线赋予元数据的权

利并对流水线的访问做出限制。例如，当一个新的虚拟机部署到生产环境中时，部署流水线可以给这台虚拟机授权来通过在生产环境中操作所需的凭证访问密钥存储。另一个部署流水线的技术是在机器的虚拟环境中设置网络配置，这样如果机器是生产环境的一部分，它就可以访问生产数据库服务器、生产配置服务器等。在这种情况下，只有部署流水线应该有权利在网络的生产环境部分创建机器。

5.4.4　在开发和提交前测试中的测试

在开发过程中有两种类型的测试过程。第一种是一种设计哲学（测试驱动开发），第二种是单元测试。

- **测试驱动开发**。在遵循这种哲学时，在编写一个功能的实际代码之前，先为它开发一个自动化测试。然后开发功能，目标是满足测试的要求。在测试通过之后，可以重构代码以满足更高的质量标准。这种做法的一个优点是为所有的代码创建快乐或晴天路径测试。
- **单元测试**。单元测试是代码级的测试，每个单元测试都是测试单独类或方法。单元测试套件应该穷举覆盖并很快地运行。典型的单元测试检查只依赖于一个类的代码的功能，并且不应该涉及与文件系统或数据库的交互。常见的实践是，以这种方式来编写代码：复杂而必需的产物（例如数据库连接）组成类的输入——单元测试可以提供这些工件的模拟版本，它们需要的开销小，运行也快。

虽然开发人员可以在任何时刻运行这些测试，但现在的实践是强制实施**提交前测试**。在执行提交前，自动运行这些测试。典型地，它们包含一组相关的单元测试，还有几个冒烟测试。冒烟测试是特定的测试，它以快速（但不完整）的方式检查服务的整体功能是否仍然能够执行。目标是在集成测试前可以发现通过单元测试但未破坏整体系统的缺陷。一旦提交前测试成功，就可以执行提交操作了。

5.5　构建与集成测试

构建是从源代码和配置等的输入创建一个可执行工件的过程。因此，它主要包括**编译**源代码（如果使用的是编译型语言）和打包执行所需的所有文件（例如，从代码生成的可执行文件、如 HTML、JavaScript 等解释型文件）。在构建完成后，就执行一组自动化测试，测试与系统其他的部件的集成是否发现了错误。这里可以重复执行单元测试，以生成比单个开发人员更广的历史信息。

5.5.1　构建脚本

构建与集成测试是通过持续集成（Continuous Integration，CI）服务器执行的。这个

服务器的输入应该是可以由单个命令调用的脚本。换句话说，运维人员或持续集成服务器执行构建的唯一输入是命令“build”，持续集成服务器的其他操作都由脚本控制。这个实践确保构建是可重复的、可追溯的。因为脚本可以再次执行，所以可以实现可重复性；因为可以对脚本进行检查以确定集成在一起的各个部分的来源，所以可以实现可追溯性。

5.5.2　打包

构建的目的是创建适合于部署的东西。有多种把系统元素打包并用于部署的标准方法。打包的合适方法将取决于生产环境。一些打包选择是：

- **运行时特定的包**，例如 Java 归档、Web 应用程序归档以及 Java 或 .NET 程序集中的联邦政府采购法规（Federal Acquisition Regulation，FAR）。
- **操作系统包**。如果将应用程序打包到目标 OS 的软件包中（例如 Debian 或 Red Hat 打包系统），很多已得到很好证明的工具可以用于部署。
- **虚拟机镜像**可以从模版镜像创建，以包含最新版本的变更。另外，一个新的构建可以分发到现有虚拟机上。这些选项接下来会讨论。不管怎么说，虚拟机镜像可以根据需要为各种环境实例化。使用它们的一个负面影响是，它们需要一个兼容的虚拟机监视器：VMWare 镜像需要 VMWare 虚拟机监视器、亚马逊网络服务（Amazom Web Services，AWS）只能运行亚马逊云机器镜像等。这隐含着测试环境必须使用相同的云服务。否则，部署就需要相应地进行修改，这意味着在测试环境中的部署不一定需要测试用于生产环境的部署脚本。
- **轻量级容器**是一种新现象。像虚拟机镜像一样，轻量级容器可以包含运行应用程序所需的所有库和其他软件段，同时还保留过程、权限、文件等的独立性。与虚拟机镜像不同，轻量级容器不需要主机上的虚拟机监视器，它们也不包含整个操作系统，这样减少了开销、负载和大小。轻量级容器可以运行在本地开发人员机器上、在组织拥有的测试服务器上以及公有云资源上——但是它们需要一个兼容的操作系统。理想情况下，应该使用相同操作系统的相同版本，因为否则，如前所述，测试环境将无法全面反映生产环境。

在使用虚拟机镜像或轻量级容器时，在应用程序中实施变更有两种主流策略：**重度烧制的镜像**和**轻度烧制的镜像**，还有一些介于重度烧制和轻度烧制之间的策略。这里的烧制指的是镜像的创建。重度烧制的镜像在运行时不能更改。这个概念也可以用术语**不可改变的服务器**来表达：虚拟机启动之后，（除配置值之外）不能对其进行更改。如果在构建阶段烧制自动发生，那么在接下来的测试阶段和生产阶段就使用这个相同的服务器镜像。通过了所有测试的镜像给出了一个很强的保证：除了一些配置值外，从这个镜像创建的服务器在生产环境中将面对与测试环境中一样的状况。重度烧制的镜像不仅封装了对应用程序的变更，而且还封装了对已安装的包变更。这增加了对镜像的信任并移除了在新虚拟机开始运行 / 运行时的不确定性和延迟，因为没有软件以其中任何一种方式

更新。第 6 章将讨论如何基于重度烧制的镜像来推出新版本。

除了在运行时允许对实例进行某些更改外，轻度烧制的镜像与重度烧制的镜像非常类似。例如，每当基于 PHP 的应用程序变更时，烧制新镜像、根据镜像启动新虚拟机、把所有已有的虚拟机下线等工作可能就小题大做了。这种情况下，停止 Web 应用服务器、根据版本控制系统检查新的 PHP 代码，并重新启动 Web 应用服务器就足够了。虽然这样做不如重度烧制的镜像那样让人信心十足，但在时间与金钱方面，它更有效。

从构建产生的工件（例如二进制可执行文件），经过测试的版本（并发现它的质量是可接受的）才应该部署到生产环境中。换句话说，如果可执行代码是用 Java、C 这种需要编译的语言编写的，那么在构建阶段之后不应该重新编译。我们看到过取决于使用的编译器版本而产生的缺陷。这个缺陷存在于编译器的一个版本中并在下一个版本中得到了修复。由于编译器的缺陷，在通过部署流水线时重新编译引入了更改应用程序行为的可能性。

不论使用什么打包机制，部署流水线中的构建步骤都应该包含编译、打包或绕制镜像，并把该构建的内容归档到构建存储库中。

5.5.3　持续集成与构建状态

在将构建设置为通过单个命令可调用的脚本之后，持续集成可按如下方式完成：

- 持续集成服务器得到新提交的通知或者定期检查是否有新提交。
- 在检测到新的提交时，持续集成服务器可以检索它。
- 持续集成服务器运行构建脚本。
- 如果构建成功，则持续集成服务器运行自动化测试——如前面及下一节所述。
- 持续集成服务器把它的活动的结果提供给开发团队（例如，通过内部网页或电子邮件）。

持续集成中的一个重要概念叫作*使构建失败*。如果编译 / 构建过程失败，或者如果提交触发的自动化测试违反了来些度量指标可接受的值，则说这次提交使构建失败。例如，忘记在提交中添加一个新文件，但更改了假设新文件存在的其他文件，这样就会让构建失败。测试可以大致分为关键测试（单个测试的失败会造成构建失败）和次关键测试（只有当失败测试的百分比超过某个设定的阈值时才会造成构建失败）。

所有度量指标都可以概括为二元结果：构建是否（足够）好？（即，未造成失败，或称为绿色构建，）或者构建是否（足够）不好？（即，中断，或称为红色构建。）

使构建失败意味着同一分支上的其他团队成员也无法构建。因此，持续集成测试实际上对团队大部分人都停用了。修复构建变成了一个高优先级的事情。有些团队要求团队成员在构建修复之前必须戴上一顶“我使构建失败了”的帽子，以此强调不要使构建失败的重要性。

测试状态可以以多种方式显示。有些团队使用电子小部件（例如熔岩灯），或者在一

个大的、可见的监视器上显示每个组件的红 / 绿灯。其他团队使用桌面通知，特别是当他们在客户的办公地点时，其中如果显示一个大红灯，则客户可能会感到紧张。

最后，如果你们的项目分成了多个组件，那么这些组件是可以分别构建的。在版本控制中，它们可以作为一个源代码项目，也可以作为多个。不论哪种情况，组件都可以分别构建到不同的可执行文件中（例如，Java 中的独立 JAR）。如果是这种情况，就应该有一个专用的构建步骤，它把所有组件都合并到一个包中。这增加了部署流水线中的复杂度（例如，增加了如何分发组件的难度）。它也允许非集中式构建：持续集成服务器可以把构建工作分发到多台机器上，例如闲置的开发人员机器上。不过，分别构建组件的一个难点是，确保只能部署组件的兼容版本。这些内容及相关考虑事项将在第 6 章中讨论。

5.5.4　集成测试

集成测试是测试已构建的可执行工件的步骤。环境包括与外部服务的联系，例如代理数据库。把其他服务包含进来就要求一种区分生产请求和测试请求的机制，这样运行测试不会触发实际事务，例如生产、装运或支付。实现这种区分的方式是：提供模拟服务；使用服务的主人提供的测试版本；或者（如果处理测试感知的组件）可以标记测试信息，这样可以利用构建到与该服务通信所使用的协议中的机制。如果使用服务的模拟版本，一种好的实践是把测试网络与真实服务区分开（例如，通过防火墙规则），以绝对确保在运行测试时不会发送真实请求。在测试时比使构建失败更糟糕的是影响生产数据库。在 5-8 节讨论事件时我们会再次回到这个主题。

5.6　用户验收测试 / 预发布 / 性能测试

预发布是把系统部署到生产环境前的部署流水线的最后一步。预发布环境尽可能镜像生产环境。在这个步骤中的测试类型如下所示：

- 用户验收测试（User Acceptance Test，UAT）是预期用户使用系统当前版本的用户界面所做的测试，可以根据测试脚本测试，也可以使用探索性测试方式。这是在用户验收测试环境中执行的，用户验收测试环境是接近生产环境的镜像，但仍然使用外部服务的测试或模拟版本。此外，用户验收测试环境中，其中测试用户或用户验收测试运维人员没有足够的权限，有些保密数据也会删除或替换。对于很难或无法自动化的方面，例如一致的外观和感受、可使用性或探索性测试，用户验收测试是有价值的。
- 自动化验收测试是可重复的用户验收测试的自动化版本。这种测试通过用户界面控制应用程序，试图密切镜像一个人类用户应该做的操作。自动化在确保每次都能以完全相同的方式执行操作的同时，它能够减轻用户验收测试的一些工作量。

这样，与成本相对昂贵的人类测试员相比，自动化验收测试在白天或夜间的少量时间能够完成更多可重复性的工作。由于把验收测试自动化需要较多的工作，所以通常只对最重要的、需要重复执行的且不大可能需要做很多维护的检查点实现自动化。一般情况下，这些测试是在专业测试套件中说明并执行的，它不应该包含用户界面中的小变更，例如把一个按钮向右移动几个像素。自动化验收测试的执行相对较慢并需要正确的设置。

- 前面提到的冒烟测试是自动化的验收测试的一个子集，用于快速分析提交的代码是否破坏了应用程序的某些核心功能。一般认为这个术语起源于流水线工程：一个密闭的流水线系统充满了烟雾，如果发生了泄漏，可以容易地检测到烟雾。一个基本原则是对每一个用户故事都遵循快乐路径做一次冒烟测试。冒烟测试应该相对较快地执行，它们甚至就可以作为提交前测试的一部分。
- 非功能测试是对性能、安全、容量及可用性等方面的测试。合适的性能测试应该有适当的设置，使用与生产环境相当的资源并且每次测试时资源都非常相似。这样可以确保度量的是应用程序的变化而不是背景噪声的影响。对于其他环境的设置，虚拟化和云技术让事情变得更容易。不过，特别是对于公有云资源，在这方面需要加以小心，因为公有云常常表现出性能上的变化。

5.7　生产环境

把系统部署到生产环境并不意味着对系统行为的观察或测试结束了。我们在这里讨论早期发布测试、错误检测以及现场测试。

5.7.1　早期发布测试

有多种形式的早期发布测试。第 6 章讨论如何发布应用程序以实现早期发布测试。现在我们重点介绍测试方法。

- 最传统的方法是 beta 发布。选择几个用户，他们常常订阅 beta 程序，具有访问应用程序的某个预发行（beta）版本的权限。beta 测试主要用于软件的内部部署（on-premise use）。
- 金丝雀测试方法是把新版本首先部署到几台服务器上，观察它们如何执行。它是 beta 测试的云测试。类似于在地下采煤时使用金丝雀方法，这里金丝雀的痛苦信号表明存在有毒气体，这些首先被部署的少量服务器会被密切监控以检测是否因升级而产生了非期望的结果。一个（或几个）应用程序服务器从当前版本升级到该应用程序的一个稳定的且经过充分测试的发布候选版本（release candidate）。负载均衡器把一小部分用户请求引导到候选版本上并持续监控。如果候选服务器在某些度量指标（例如性能、可伸缩性、错误数等）上是可以接受的，候选版本就

部署到所有服务器上。

- A/B 测试与金丝雀测试相似，差别仅在于 A/B 测试的目的是，对于特定的业务的关键绩效指标（Key Performance Indicator）来说，确定哪个版本执行得更好。例如，一个推荐产品的新算法可以增加收入，或者用户界面更改可以带来更多的点击量。

5.7.2　错误检测

系统即使通过了所有测试，也仍然可能有错误。这些错误可以是功能的，也可能是非功能的。用于确定非功能错误的技术包括监控系统指示的不良行为。这可以包括监控对用户请求的响应时间、队列长度等。Netflix 报告它们监控了 95 个不同的指标并与历史数据进行对比。与历史数据有偏差时会触发警告，通知给运维人员、开发人员，或两者同时通知。

警告升级后，跟踪并寻找源头可能会相当困难。系统产生的日志对于是否能够跟踪源头来说很重要。第 7 章将讲述该主题，不过为了理解本章内容，还是需要说明，造成警告的软件起源以及触发警告的用户请求都必须很容易得到，这一点很重要。需要能够对错误做出诊断，这是我们强调使用自动化工具维护软件活动历史的一个原因。

不论是哪一种情况，在错误诊断并修复之后，错误的原因都可以成为未来发布的一个回归测试。

5.7.3　现场测试

监控是一种被动的测试形式。也就是说，系统以正常方式运行并收集关于它们行为及性能的数据。在将系统放入生产环境后，测试的另一种形式是干扰正在运行的系统。这种形式称为*现场测试*（Live testing）。Netflix 有一套叫作 Simian Army 的测试工具。Simian Army 的元素同时是被动的和主动的。被动元素检查运行的实例以确定未使用的资源、过期的证书、实例的健康检查以及符合最佳实践。

Simian Army 的主动元素向生产系统注入特定类型的错误。例如，Chaos Monkey 随机杀死活跃的虚拟机。我们在第 2 章中讨论过，云上的故障是常见的。如果物理服务器发生故障，那么该机器上的所有虚拟机都会突然终止。因此，应用程序对于这种故障应该有弹性。Chaos Monkey 模拟这种故障类型。杀死一个实例之后，包括响应时间这样的所有度量指标都应该监控，以确保系统不会受到故障的影响。当然，一次杀死的实例不能太多。

Simian Army 的另一个主动元素是 Latency Monkey。Latency Monkey 向消息中注入延迟。网络变得繁忙并出乎意料地慢。Latency Monkey 通过手工延迟一个服务发到另一个服务的消息来模拟变慢的网络。与使用 Chaos Monkey 一样，这种测试要小心，不要影响客户。

5.8　事件

无论你测试得多么好或者部署工作组织得多么好，在系统进入生产环境之后总会存在错误。理解部署后的错误的潜在原因，有助于更快地诊断问题。我们没有部署后错误的分类或不同类型的相对频率。不过我们有几个从 IT 专业人士那里听到的轶事。

- 一个开发人员把测试代码连接到了生产数据库。这样的例子我们听到多次了。一次是一位经验欠缺的开发人员，另一次是一位开发人员通过通道打开了一个通向生产环境的 SSH。
- 不同组件之间存在版本依赖关系。当组件之间存在这种依赖关系时，部署的次序就变得重要，如果次序不正确，就有可能产生错误。第 6 章将讨论使用功能开关来避免这个问题。
- 有依赖关系的系统中的一个变更与部署一致。例如，从一个有依赖关系的系统中删除应用所依赖的服务，而这个删除发生在所有预发布环境都通过测试之后。本章讨论的镜像烧制与这个问题有关。如果有依赖关系的系统已经生成镜像，那么接下来发生的变更就不会被合并。如果有依赖关系的系统在镜像之外，那么构建一个可执行镜像的特征就不会影响这个错误的出现。
- 有依赖关系的系统的参数设置是错误的。即，在有依赖关系的系统中出现队列溢出和资源耗尽。受影响组织所采取的修正措施应该是调整受影响系统的配置并增加监控规则。

5.9　小结

有一个适当部署流水线对于快速创建和部署系统是非常重要的。流水线至少包括 5 个主要步骤——提交前、构建与集成测试、用户验收测试 / 预发布 / 性能测试、生产环境及提升到正式生产环境。

每个步骤都要在不同的环境中执行，并使用一组不同的配置参数值——虽然这组参数的数量应该尽可能地限制。系统在流水线中移动时，你可以对其正确性获得越来越多的信心。但即使系统发布到正式生产环境，也会发现错误，从性能和可靠性的角度也有可提升之处。即使在将系统放到生产环境或提升到正式生产环境中后，也可以通过现场测试机制继续进行测试。

功能开关可以让生产环境中的代码无法访问。它们允许在提交的模块中包含不完整的代码。当代码不再需要时应当删除它们，因为否则，它们会让代码基错乱，并且更改用途的功能开关可能导致错误。

测试应该是自动化的，通过测试框架运行，结果应该报告给开发团队以及其他感兴趣的团体。将系统放入生产环境后很多事件都是由开发人员或配置错误引起的。

参与 DevOps 项目的架构师应该确保：

- 应该建立各种工具及环境以便使它们的活动是可追溯的且可重复的。
- 应该根据配置参数在不同环境中是否变化以及它们的保密要求对它们进行组织。
- 部署流水线中的每个步骤都有一组自动化测试以及相应的测试框架。
- 当需要切换的代码放入生产环境，并且经过判断认为已经成功部署后，应当删除功能开关。

5.10　更多阅读材料

有关本章包含的很多主题的更详细的讨论，参阅《 Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation 》[Humble 10]。

Carl Caum 在他的博客中讨论了持续交付和持续部署的差别 [Puppet Labs 13]。

本章中的很多基本概念信息来自维基百科。

- http://en.wikipedia.org/wiki/Revision_control 对修订控制系统进行了概要讨论。特定系统，如 Git，有它们自己的条目。
- http://en.wikipedia.org/wiki/Test_harness 讨论了测试框架。
- http://en.wikipedia.org/wiki/Regression_testing 讨论了回归测试。
- http://en.wikipedia.org/wiki/Development_environment 列示了不同类型的环境类型（或服务器层）。

Paul Hammant 在 [DZone 13] 中讨论了分支与基于主干方法的对比。

在 [Gillard-Moss 13] 可以找到对重度烧制的镜像与轻度烧制的镜像之间讨论的一个例子。

公有云中的性能变化的主题在多种出版物中进行了调查，例如“ Runtime Measurements in the Cloud:Observing, Analyzing, and Reducing Variance ”[Schad 10]。

Simian Army 是在 [Netflix 15] 中定义和讨论的。

第6章 部　署

错误代码725：它运行在我机器上。

——HTTP状态码7XX的RFC：开发人员错误

6.1 概述

部署是将服务的某个版本投入生产环境的过程。服务的初始部署可以视为服务从没有版本进入初始版本。由于初始部署对大多数系统只发生一次并且新的版本会频繁出现，所以本章将探讨服务的升级。如果是初始版本，那么我们讨论的某些问题（比如当前已部署版本的停机时间）就不相关了。部署的总体目标是：对系统用户产生最小影响的情况下，将服务的升级版本投入生产环境中，这些影响可能是失败或者停机时间。

服务变更有3个原因——修复错误、提高服务的质量，或者增加新功能。为了简化最初的讨论，我们假定部署是一个要么全有要么全无的过程——部署结束时，要么运行服务的所有虚拟机都已部署了升级版本，要么没有任何机器部署。本章稍后，我们会看到部分部署有其存在的必要，但此时我们先推迟讨论。

图6-1所示是我们关心的情况。这是图4-1的改进，其中微服务3正在升级（灰色显示）。微服务3依赖于微服务4和5，而微服务1和2（微服务3的客户端）依赖于微服务3。目前，我们假定一个虚拟机只运行一个服务。这会使我们专注在服务上——它们的设计及它们的关系，并将服务的部署与虚拟机的部署等同看待。我们将在本章的后面讨论其他情况。

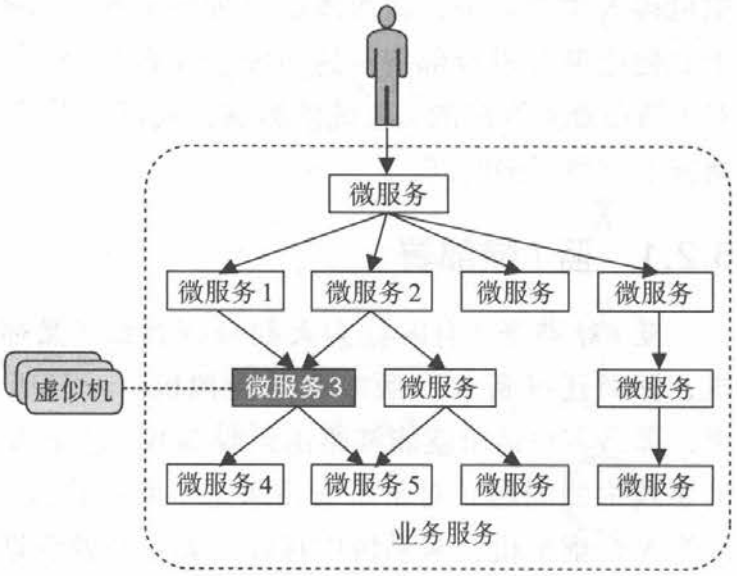


图6-1 微服务3正在升级（改编自图4-1）[标注法：架构]

图6-1还展示了服务运行在多个虚拟机上。具体服务的虚拟机数量取决于该服务所承担的工作负荷，对于要为众多客户提供的服务，其可能发展到数百台甚至数千台虚拟机。每个活跃的虚拟机都部署了一个版本的服

务，但不是所有的虚拟机都运行相同的版本。

部署的目标是从当前状态（*N* 台虚拟机正在运行服务的老版本 A）迁移到新状态（*N* 台虚拟机运行该服务的新版本 B）。

6.2　部署管理的策略

部署管理有两个常用策略——蓝 / 绿部署和滚动升级。它们在成本及复杂性上有所不同。成本可能包括虚拟机的费用以及虚拟机中运行的软件的许可费用。在我们更详细地讨论这些策略前，我们需要先做以下两个假设：

1）当部署新版本时，应该维护向客户提供的服务。对许多因特网电子商务而言，无需停机地维护提供给客户的服务是至关重要的。它们的客户遍布全球，期望能够全天候地处理业务。当然，一天中的某些时段比其他时段更繁忙，但服务必须随时可用。对于客户主要位于一个地理区域的组织，它能够承担计划的停机时间——但如果停机是可以避免的，为什么要停机呢？将停机时间安排到非工作时间要求系统管理员和运维人员要在非工作时间工作。这是避免停机的另一个原因。

2）开发团队应该能够在无需其他团队协同的情况下随时部署他们服务的新版本。这必然会影响其他团队开发的客户服务。我们之前已探讨了开发团队的协调同步与发布新功能时间之间的关系。让开发团队或个人开发者发布他们服务的新版本而无需协调开发客户服务的团队，可以去除同步协调的一个理由。然而，这也会引起逻辑问题，我们会在 6.3 节中讨论这些问题。

此外，将一个版本的新虚拟机投入生产环境需要花费时间。为了将服务升级后的虚拟机投入生产环境，新版本必须加载到虚拟机中并初始化和集成到环境中，有时要依赖于其他服务先进行部署。这可能会花费几分钟。因此，取决于操作可能的并行程度及其对（仍在服务客户的）系统的影响，成百上千虚拟机的升级可能花费数小时或者，在极端情况下，数天的时间。

6.2.1　蓝 / 绿部署

蓝 / 绿部署（有时称为大翻转或者红 / 黑部署）包括了 *N* 台提供版本 A 服务的虚拟机，同时还准备 *N* 台版本 B 的虚拟机。一旦版本 B 的 *N* 台虚拟机配置好并准备服务请求，那么客户请求就能够路由到版本 B。这是指示域名服务器（DNS）或负载均衡器更改消息路由的问题。对于所有请求这个路由切换可以一步完成。在监管期后，配置了版本 A 的 *N* 台虚拟机会从系统中移除。如果在监管期内有任何问题发生，则路由可以切回来，请求会再次转到运行版本 A 的虚拟机上。这种策略在概念上很简单，但在虚拟机和软件许可的花费上却非常昂贵。运行时间很长的请求和有状态的数据在切换和回滚期间需要

特别关注。

在终止所有的版本 A 的虚拟机之前准备包含版本 B 的 *N* 台虚拟机是成本的来源。首先，新虚拟机必须全部准备好。虽然准备可以并行完成，但准备数以百计的虚拟机仍然颇为耗时。在整个过程中要分配超出为客户提供服务所必需的额外 *N* 台虚拟机，这包括初始配置版本 B 的时间以及完全切换到版本 B 后的监管时间。因此，为了这些时间，基于虚拟机的成本会翻倍。

该模式的一种变体是逐步进行流量切换。先将小部分请求先被路由到版本 B，可以有效地进行金丝雀测试。我们在第 5 章提及金丝雀测试并在 6.6.1 节进行详细讨论。如果一段时间内一切顺利，就会准备更多版本 B 的虚拟机并将更多请求路由到该虚拟机池中，直到所有请求都路由到版本 B。这提高了你对部署的信心，但也引入了一些一致性的问题。我们在 6.3 节探讨这些问题。

6.2.2 滚动升级

滚动升级包括将少量版本 B 的虚拟机一次直接部署到当前生产环境中，同时关闭相同数量的运行版本 A 的虚拟机。假定我们一次部署一台版本 B 的虚拟机。一旦一台额外的版本 B 的虚拟机被部署并接收请求，那么一台版本 A 的虚拟机就从系统中移除。重复该过程 *N* 次就得到版本 B 的完整部署。该策略花费不多但却更复杂。它在部署期间会消耗少量额外的虚拟机，但会引入一些一致性问题并有更大的风险干扰当前的生产环境。

图 6-2 表示了 Amazon 云中的额滚动升级。每个虚拟机（目前仅包含一个服务）会退役（从弹性负载均衡器（ELB）上移除并注销，以及终止），而后一个新的虚拟机会启动并注册到 ELB。该过程将持续到所有包含版本 A 的虚拟机替换为包含版本 B 的虚拟机为止。如果进行滚动升级时你并未充分利用你的虚拟机，并且你一次终止一台或少量虚拟机时还能维持期望的服务级别，那么滚动升级的额外花费就可能很低。如果你在开始滚动升级前添加了少量虚拟机，这会花费一些成本来减少性能的影响以及滚动升级的风险。

滚动升级期间，一部分虚拟机提供版本 A 的服务，而其余的虚拟机提供版本 B 的服务。这会造成由于版本混合而带来失败的可能。我们在下一节讨论这种类型的失败。

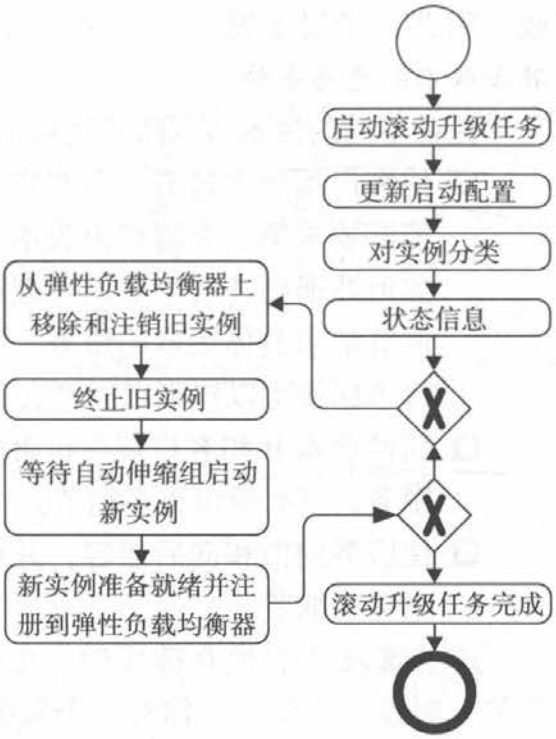


图 6-2 滚动升级的一种表示 [标注法：业务流程建模标注]

6.3　逻辑一致性

假设使用滚动升级完成部署，引入了一种逻辑不一致性——相同服务的多个版本会同时存在。那些在部署完成前启用新版本的蓝 / 绿部署变种也会发生这样的问题。

重温图 6-1 并假定部署服务时无需与其客户或依赖服务进行同步协调，我们可以看到逻辑不一致性的第二个可能的来源——服务与其客户之间在功能上的不一致性。

逻辑不一致性的第三个来源是服务与保存在数据库中的数据之间的不一致性。

我们现在讨论这三种不一致性。

6.3.1　相同服务的多个版本同时存在

图 6-3 显示了由于相同服务有两个活跃版本而造成的不一致性。图中展示了两个组件——客户及服务的两个版本（版本 A 和版本 B）。客户发送了一条消息，这个消息被路由到版本 B。版本 B 完成相关操作并返回一些状态给客户。然后客户将该状态包含在接下来对服务的请求中。第二个请求被路由到版本 A，但该版本不知道如何理解这个状态，因为这个消息应该由版本 B 接收。因此，错误就发生了。这个问题称作*混合版本的竞态条件*。

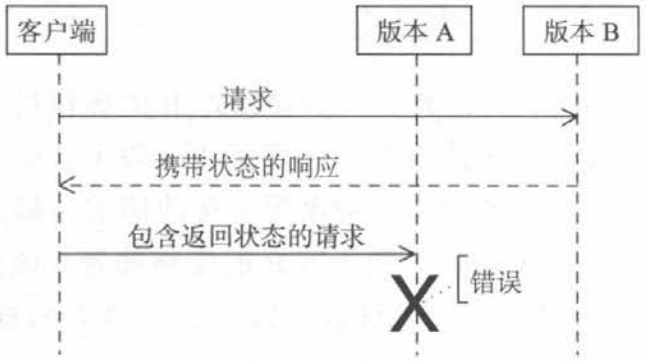


图 6-3　混合版本的竞态条件，导致一个错误 [标注法：UML 时序图]

几种不同的技术可以防止这种情况。

- 使客户端版本知道从而使它了解最初的请求是由版本 B 的虚拟机服务的。然后它能够要求第二个请求由版本 B 的虚拟机提供服务。在第 4 章中，我们描述了服务如何注册到注册中心或负载均衡器。这样的注册可以包含版本号。然后客户端就能够请求具体版本的服务。服务的响应消息应该包含一个标签，以便客户端通过这个标签可以知晓它刚刚交互的服务的版本。
- 切换版本 B 和客户端中包含的新功能，以便在任何给定的时间仅有一个版本提供服务。下面给出更多细节。
- 使服务向前和向后兼容，并在特定请求没有得到满足时使客户端意识到。下面给出更多细节。

这些选项并非相互排斥的。也就是说，你可以在一个向后兼容的设置中使用功能开关。假设一个例子，你对一个服务进行了重大的重组并为其添加了新功能。在滚动升级中，你会在一些虚拟机上安装重组的新版本，但并不激活新功能。这要求新版本向后兼容。

我们首先讨论功能开关。第 5 章介绍的功能开关是将其作为一种不影响测试过

程的情况下部署部分已完成代码的方法。这里我们将相同的机制用于在升级中激活新功能。

1. 功能开关

如果服务仅由一个小团队开发，则服务的功能是有限的。这意味着功能可能会跨多个服务。相应地，这意味着你必须在两个方向上协调功能的激活。首先，你刚刚部署的服务的所有虚拟机必须激活该功能的服务部分。其次，功能实现所涉及的所有服务必须将其对应的功能部分激活。

功能开关，如第5章所述，能够用来控制是否激活一个功能。重申一下，功能开关是if语句内的一段代码，其中if条件基于一个可以从外部设置的功能变量。使用这个技术意味着与激活一个功能相关的问题是：(a) 确定参与实现一个功能的所有服务已完全升级；(b) 在这些服务的所有虚拟机上同时激活该功能。

这两个问题是分布式系统的元素之间进行同步的例子。进行这样同步的现代主流方法都基于Paxos或ZAB算法。正确实现这些算法是很困难的。然而，像ZooKeeper这样的系统中有标准实现，并且还很容易使用。

让我们从服务的角度看看这是如何起作用的。为了描述简单，我们假定要部署的服务只实现了一个功能的一部分，*功能X*。当服务的虚拟机部署后，它将自己注册为对FeatureXActivationFlag感兴趣。如果该标志为false，那么该功能就被关闭；如果标志为true，那么该功能就被打开。如果FeatureXActivationFlag的状态发生变化，通知虚拟机并做出相应的反应。

一个与要升级系统内的所有服务都无关的代理负责设置FeatureXActivationFlag。这个代理可能是一个人，也可能是自动完成的。该标志由ZooKeeper维护，因此在涉及的虚拟机之间保持一致。只要将开关信息同步地通知给所有虚拟机，那么该功能就会被同步激活，也就不会产生导致失败的版本不一致问题。同步信息广播是由ZooKeeper执行的。这种将ZooKeeper用于功能开关的特定用法常常会在其他工具中实现。比如，Netflix的Archaius工具提供了分布式系统的配置管理。被管理的配置可以是功能开关或者其他属性。

代理知道实现*功能X*的各种服务并且直到所有这些服务都完成升级才激活该功能。因此，没有要求这些涉及的服务要按照某种特定的顺序升级，甚至或者彼此之间暂时聚拢在一起。可能要花费数天甚至数周才能让涉及的所有服务实现*功能X*。

一个难题是决定这些虚拟机何时已“完全升级”。虚拟机可能出现故障或者变得不可用。等待这些虚拟机升级后才激活功能是不可取的。按第4章描述的方式使用注册表或负载均衡器能够激活代理以避免这些问题。我们知道，每个虚拟机必须定期更新其注册信息以表明自己尚处于活动状态。激活的代理检查已注册的相关虚拟机来确定所有相关服务的虚拟机何时已升级到适当的版本。

2. 向后和向前兼容

使用功能开关来协调新功能涉及的各种服务是防范多版本造成失败的一项有效措施。另一项措施是确保服务的向前和向后兼容。

- 如果服务的新版本与旧版本在行为上是一致的，那么该服务就是向后兼容的。对于旧版本服务所已知的请求，新版本提供相同的行为。也就是说，服务的版本 B 提供的外部接口是该服务的版本 A 提供的外部接口的超集。
- 向前兼容意味着客户端优雅地处理不正确方法调用的错误响应。假设客户端希望使用一个方法，而该方法只在服务的版本 B 中才有，但版本 A 中没有提供。如果该服务返回的错误编码表明它没有识别出该方法调用，那么客户端能够推断出它访问了该服务的版本 A。

要求向后兼容乍看起来似乎会阻碍很多对服务的变更。如果你不能更改接口，你如何能够添加新功能，或者，例如，重构服务呢？实际上，可以使用图 6-4 中描绘的模式实现保持向后兼容。

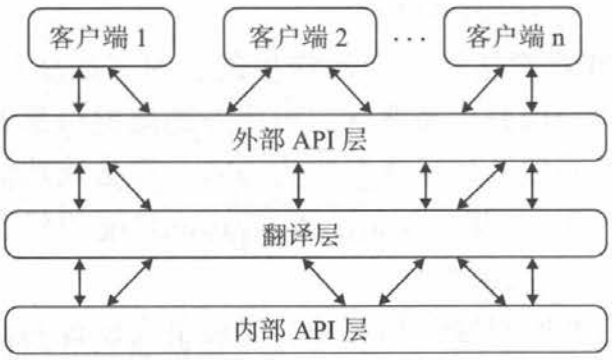


图 6-4　维护服务接口的向后兼容 [标注法：架构]

升级的服务要区分内部接口与外部接口。外部接口包括之前版本已存在的所有接口以及当前版本可能添加的新接口。内部接口可以在每个版本中进行调整。在外部接口和内部接口的中间是一个翻译层，它将旧接口映射为新接口。对客户端而言，旧接口对新版本仍然是可用的。如果客户端想要使用新功能，那么新接口对该功能也是可用的。

使用该模式的一个结果是，没有任何客户端在使用了过时的接口后对这些接口进行维护。通过监控和记录所有的服务调用，可以确定哪个客户端使用哪个接口。一旦接口在足够长的时间内没有被使用，就可以将其废弃。废弃一个接口可能导致额外的维护工作，因此不要轻率地做这件事情。

向前与向后兼容允许你控制的服务独立升级。但不是所有的服务都受你控制。尤其是，第三方服务、库以及遗留服务可能无法向后兼容。这种情况下，有几种技术可以使用，但没有一个是万无一失的。

- 发现。第 4 章描述了如何进行服务注册，以便客户端能够发现它们。注册应该包

含服务的版本号。客户端可以请求将它们连接到服务的特定版本或满足某些限制条件的版本。如果没有满足限制条件的服务，那么客户端要么执行补救操作要么报告失败。这要求客户端了解其请求服务的版本和通过注册版本号服务要遵循的架构。标准社区正在讨论版本号是否应该成为服务接口的一部分。

- 探索。发现假定服务用注册表进行注册。库和许多第三方软件系统不进行这样的注册。这种情况下，在库和第三方系统上使用内省能够使客户端确定版本号。内省要求库或第三方软件要使其版本号在运行时可以访问，或者通过接口或者通过其他机制（比如记录在文件中的版本号）。内省也假定客户端知道其所需服务的版本。
- 可移植层。图 6-5 展示了可移植层的概念。可移植层提供了单一的接口，该接口可以被转换为各种类似系统的接口。这种技术已用于将应用程序移植到不同的操作系统上，允许多种不同的设备从应用程序的角度看是一样的，或者允许不同数据库系统的替代。在第 4 章中，我们确定了需求：一个组件的外部系统之间的交互要局限到一个模块中。这个模块充当了可移植层。一个需求是为可移植层定义的接口适合于管理外部系统的所有版本。根据外部系统的两个版本是否需要共存，这个模式有两个变体。如果两个版本需要共存，则可移植层必须在运行时确定使用哪个版本的外部系统，服务必须提供一些基础以便可移植层能够进行选择。管理不同协议的设备属于这一类。如果两个版本无需共存，那么可以在构建时确定使用哪个版本，可移植层的正确版本可以整合到该服务中。图 6-5 展示了两个版本共存的情况。

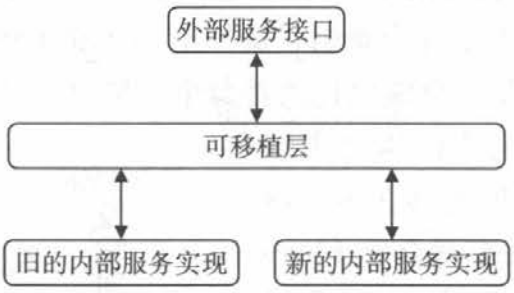


图 6-5 外部系统的两个版本共存的可移植层 [标注法：架构]

6.3.2 兼容数据库中保存的数据

除了维护各种服务之间的兼容性外，一些服务必须还能够以一致的方式读 / 写数据库。假设数据模式发生了变化：在模式的旧版本中，客户地址是一个字段；新版本中，地址分成了，街道、城市、邮编和省。这种情况下，不一致意味着一个服务想要将地址写到一个字段中，使用的模式已经将地址分成了不同的部分。不一致是由数据库模式的改变引发的。注意，模式可能是像关系数据库管理系统（RDBMS）中那样显式的，或者

像各种 NoSQL 数据库管理系统中那样隐含的。

这种模式改变的最基本解决方法是不修改现有字段，但只增加新的字段或表，这样做就能够不影响现有的代码。新字段或表的使用可以增量地集成到应用程序中。实现这点的方法之一是将新字段和表当作一次发布中的新功能。也就是说，要么新字段和表的使用由功能开关控制，要么服务对数据库字段和表向前和向后兼容。

然而，如果模式变更是绝对需要的，那么你有两个选择：

1）将持久性数据从旧模式转换为新模式。

2）在读 / 写期间将数据转换为适当的形式。这可以通过服务也可以通过数据库管理系统实现。

这些办法并非相互排斥。你可以在后台执行这种转换，在该转换正在执行时转换实时数据。当满足需求时现代 RDBMS 能够在线上将数据从一种模式重组为另一种模式——但要以存储和性能为代价。参考 6.10 节引用的 Sockut 和 Iyer 针对此问题的讨论以及所使用的技术。NoSQL 数据库系统通常不提供这样的功能，因此，如果你要使用它们，你不得不为你自己的具体情况设计一个解决方案。

6.4　打包

现在，根据获得的最新版本我们从运行时期间的服务一致性转向构建过程的一致性。如第 4 章所述，决定用组件打包服务和每个服务打包为一个组件不是终止你的打包决策。你必须决定驻留在相同虚拟机上的不同组件之间的绑定时机以及将服务放入虚拟机的策略。将组件打包到虚拟机镜像上称为*烧制*，选择范围从*轻度烧制*到*重度烧制*。我们在第 5 章讨论了这些选择。我们这里要继续讨论的是每个虚拟机要装载的进程数。

虚拟机是运行在虚拟机监视器之上的一个镜像，这个虚拟机监视器能够在多个租户（tenant）或虚拟机之间共享裸金属（bare metal）处理器、内存和网络。虚拟机的镜像按计划加载到虚拟机监视器中。

虚拟机镜像可以包含多个独立的进程——每一个进程是一个服务。那么问题是：多个服务应该放入一个虚拟机镜像中吗？图 6-6 给出了两种选择。在上面的选择中，开发人员提交一个部署服务，这个服务嵌入在一个虚拟机镜像中。例如，Netflix 声称他们将每个虚拟机打包为一个服务。在下面的选

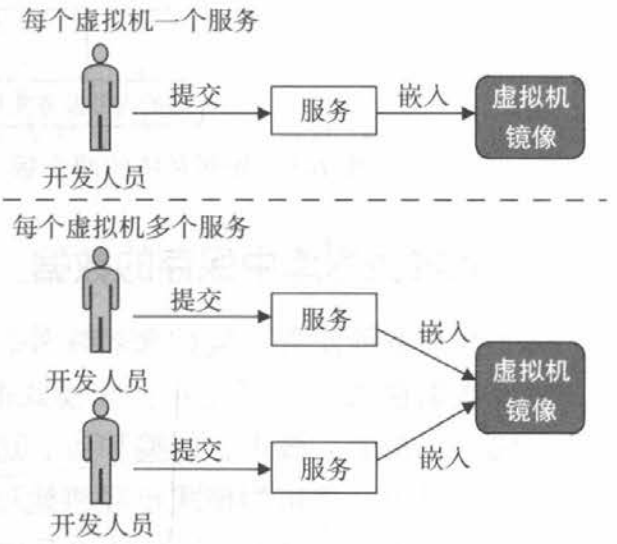


图 6-6　打包服务的不同选择 [标注法：架构]

择中，不同的开发人员提交不同的服务到一个虚拟机镜像中。轻量级容器的出现经常假定每个容器一个服务，但有可能每个虚拟机有多个容器。

这两种选择的一个差异是，虚拟机必须烧制的次数。如果每个虚拟机一个服务，那么当提交服务中的变更时就创建一个虚拟机镜像。如果每个虚拟机两个服务，那么无论提交第一个或第二个服务的变更都必须重新烧制虚拟机镜像。这个差别是很小的。

当服务 1 向服务 2 发送消息时，一个更重要的差别出现了。如果两个服务在相同的虚拟机中，那么该消息无需离开虚拟机进行传送。如果它们在不同的虚拟机内，就要涉及更多的处理和潜在的网络通信。因此，当每个服务打包到一个虚拟机中时，消息的延迟是很大的。

另一方面，将多个服务打包到相同虚拟机中有可能造成部署竞态条件。竞态条件出现，因为不同的开发团队没有协调它们的部署安排。这就意味着他们可能在（大致）相同的时间部署其升级。下面的例子假设升级的服务包含在虚拟机（重度烧制）的部署部分中，而不是之后由部署的软件加载。

我们在图 6-7 中看到了一种可能。开发团队 1 创建了一个具有服务 1（S1）的新版本（V_{m+1}）和服务 2（S2）的旧版本的新镜像。开发团队 2 创建的具有服务 1 的旧版本和服务 2 的新版本（V_{n+1}）的新镜像。两个团队的供给流程有重合，这造成了部署竞态条件。我们在图 6-8 中看到了相同问题的另一个版本。在这个例子中，在开发团队 2 已经提交了他们的变更后开发团队 1 构建他们的镜像。结果是相似的，因为部署的最终版本没有服务 1 和服务 2 的最新版本。

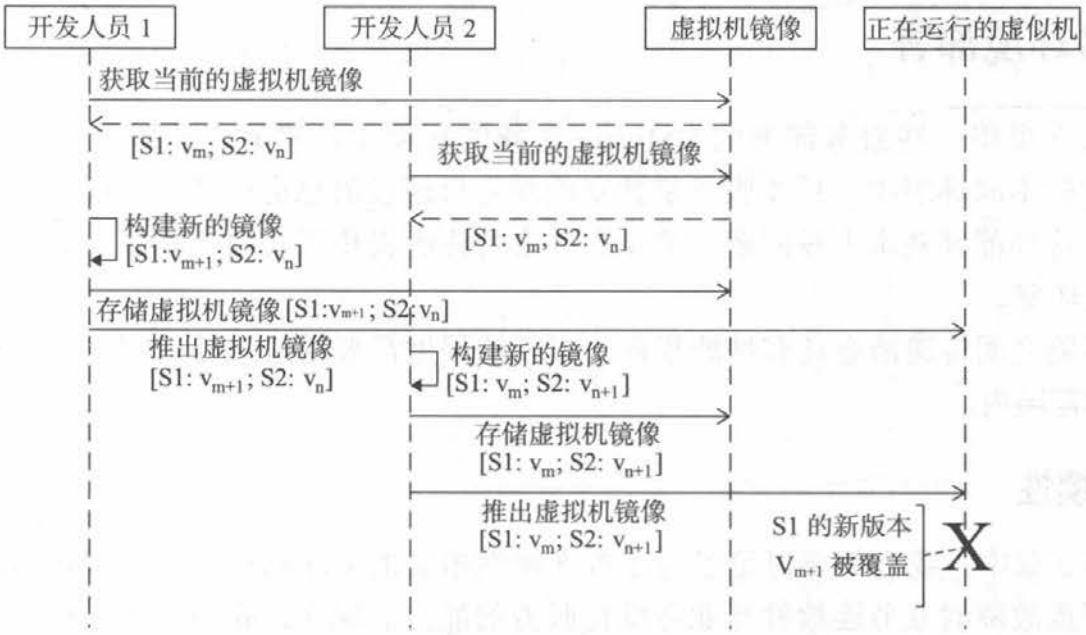


图 6-7 当两个开发团队独立部署时的一种竞态条件 [标注法：UML 时序图]

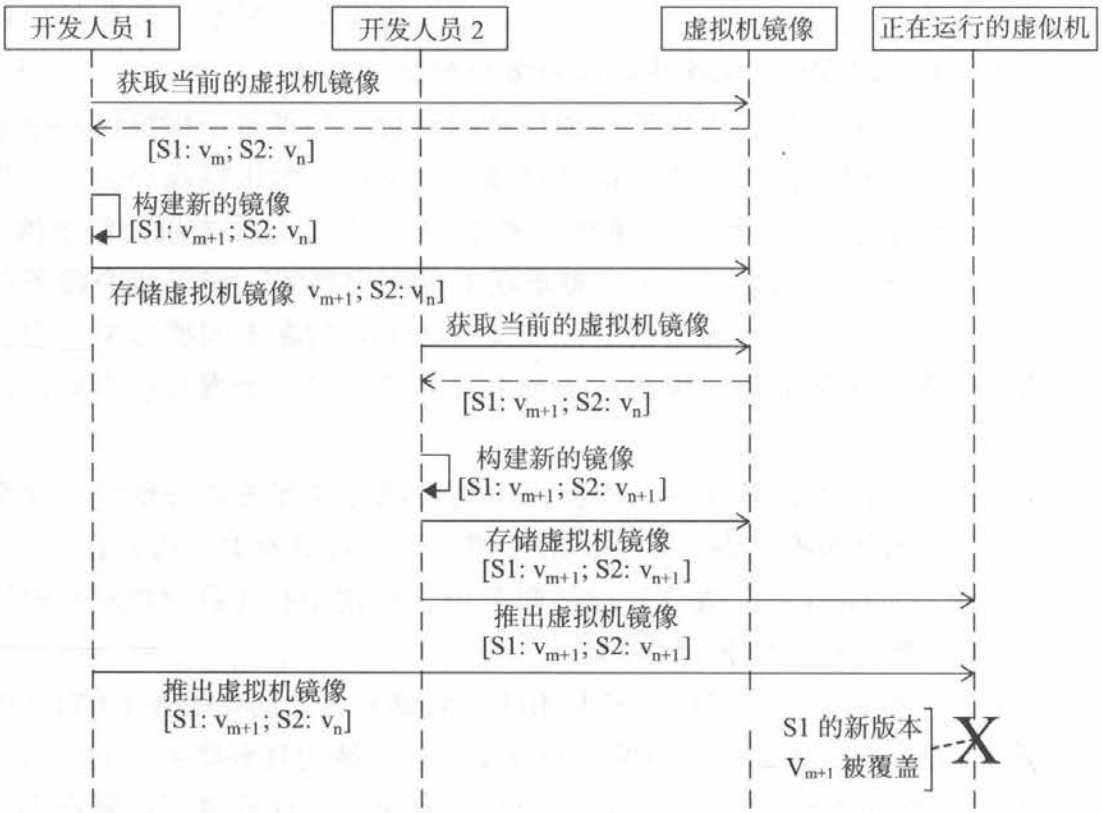


图 6-8　当两个开发团队独立部署时的另一种竞态条件 [标注法：UML 时序图]

在同一个虚拟机中包含多个服务是在降低延迟与可能的部署竞态条件之间进行权衡。

6.5　多环境部署

你也许想将一些服务部署到 VMware 这样的环境中，将另一些服务部署到亚马逊 EC2 这样的不同环境中。只要服务是独立的并且只通过消息进行通信，那么使用我们给出的设计这种部署基本上没问题。第 4 章讨论的注册表和负载均衡器需要能够将消息导向不同的环境。

在环境之间传递消息还有性能损耗。损耗的程度需要通过实验确定以便总损耗在可以接受的范围内。

业务连续性

在第 2 章中，我们简要讨论了与业务连续性相关的需求和概念。我们知道，当面对灾难或严重故障时业务连续性是业务维持服务的能力。现在，我们开始了解如何实现业务连续性。从根本上来说，通过部署物理上和逻辑上相互分隔的站点来实现业务连续性。我们对公有云的部署和私有云的部署进行了区分，尽管其基本元素（状态的管理）是相同的。我们在第 10 章进一步讨论灾难恢复，在第 11 章讨论案例研究。

1. 公有云

总的来说，公有云是极其可靠的。它们由成千上万个物理服务器组成并提供了大量的复制和故障转移服务。然而，故障一定会发生。这些故障可能发生在你系统的某个虚拟机上或者发生在其他的云服务中。

- 虚拟机的故障并不罕见。云供应商通过购买标准硬件来实现规模效益。硬件的任何部分都可能发生故障——内存、磁盘、主板、网络和CPU。可能会完全失效也可能是部分失效。底层硬件部分失效会使虚拟机变慢，尽管它还在运行。不论哪种情况，你必须创建你的系统以检测虚拟机故障并对它们做出反应。这超出了本章的范围。
- 云基础设施的故障很少见，但并不是不可能发生。快速搜索“public cloud outages”（公有云停机）能够告诉你最近发生的那些备受瞩目的服务中断事件。其他停机虽不太出名但仍然发生了。通过选择部署虚拟机的方法能够让你幸免于许多停机事故。

亚马逊EC2分散在全球的多个地区（写作本书时有9个）。每个地区有多个可用性区域。每个可用性区域安放在与其他可用性区域物理上相互不同的地方，并且有自己的电力供应、物理安全等。如果你将系统的虚拟机部署到同一地区的不同可用性区域，则你对云停机有一定程度的防护。如果你将系统的虚拟机部署到不同地区，那么你对停机将有更多防护，因为某些服务，如弹性负载平衡，是每地区独立的。当你部署到不同可用性区域或地区时，两个需要牢记的注意事项是状态管理和延迟。

1）*状态管理*。如第4章所述，使服务没有状态有几点好处。如果服务没有状态，那么可以在任何时间创建额外的虚拟机来处理增加的工作负荷。额外的虚拟机也可以在虚拟机故障发生时创建。当具有合适的基础设施时，创建或删除无状态服务的虚拟机对客户端是透明的。无状态服务的缺点是状态必须维护在系统的某个地方；当服务需要获取或修改这个状态时，可能会增加延迟。延迟增加的一个后果是服务可能会在本地缓存状态。这就意味着在某些环境中你需要清理缓存。少量状态可以用Memcached（正如其名称所表示的，它就是专为缓存而设计的）这样的服务进行维护。大量的状态应该维护在持久性存储库中。部署到不同的可用性区域或地区要求持久性存储库保持一致。MRDMS可以配置为自动提供该服务。有些NoSQL数据库系统也提供了在存储库的多个虚拟机之间进行复制。公有云供应商通常为此提供特定服务，尽管对于亚马逊RDS副本之间的复制只能跨可用性区域。

与使服务无状态有关的一个问题是：由第三方软件该第三方软件维护状态，提供的服务超出你的控制范围，并且没有提供复制服务。当你制订业务连续性计划时，从这样的软件到不同供应商的迁移是你必须考虑的权衡之一。

2）*延迟*。从一个可用性区域向另一个发送消息会增加一点儿延迟。消息从一个地区发送到另一个地区会给系统增加更多的延迟。一组测量显示给欧洲地区内的可用性区域

带来了 1.35ms 的额外延迟，给欧洲和美国东部地区之间带来了 231ms 的额外延迟。对于业务连续性，额外延迟是你必须考虑的另一个权衡。

2. 私有云

许多组织被要求或决定维护私有数据中心，而不使用公有云。这些数据中心坐落于不同的地点，比如，相距大约 100 英里，彼此之间有高速链路。我们看到许多组织有两个数据中心但没有一个组织有 3 个数据中心。拥有 3 个数据中心时，维护业务连续性的成本会增加 50%，而双重失败的概率通常认为非常不可能。这是一个具体组织必须做的风险管理决策。

从软件架构的角度看，使用两个数据中心与使用公有云中的两个可用性区域之间的唯一区别是数据中心内的硬件选择。在公有云中，你可以指定你想要分配何种类型的（虚拟）硬件。在私有云中，解决不同硬件问题的方法是使数据中心的硬件完全一致。然后，服务可以部署到任何一个数据中心而无需使服务或部署工具知道数据中心。虚拟化提供了硬件独立性的一些措施，但像物理核的数量或刀片的使用这样的硬件功能会影响操作系统和性能。如果两个数据中心有影响操作系统的硬件功能并且这些功能是虚拟机可见的，那么虚拟机就无法直接从一个环境移动到另一个环境。如果硬件差异对性能有影响，那么部署工具需要进行弥合，比如，由于第二个数据中心的机器慢 50%，所以通过为第二个数据中心的每个服务多配置 50% 的虚拟机来解决。

拥有两个完全相同的数据中心的一个更大的优点是：当一个数据中心上的预期负载比较低时，可以在这段时间内使用该数据中心进行生产前测试。

6.6　部分部署

至此，我们的讨论一直关注要么全部部署要么不部署。现在，我们讨论两种类型的部分部署——金丝雀测试和 A/B 测试。第 5 章简单介绍了这些测试。这里我们详细描述如何实现这些类型的部分部署。

6.6.1　金丝雀测试

一个新版本将在预发布环境（其尽可能接近生产环境）中测试后部署到生产环境。然而，新版本中仍然有可能存在错误。这些错误可能是功能方面的，也可能对质量有影响。在真实生产环境中执行进一步的测试是金丝雀测试的目的。金丝雀测试在概念上类似于在收缩包装的软件世界中的 beta 测试。

一个问题是，向谁暴露金丝雀服务器。这可能是随机采样的用户。另一种是依据用户所属的组织来决定，比如，开发组织的雇员，或特定的用户。这个问题也可以基于地理位置来回答，比如，那些被路由到特定数据中心的请求由金丝雀版本来服务。

进行金丝雀测试的机制取决于功能是否使用功能开关进行激活或者服务是否假定向前或向后兼容。不论哪种情况，一个新功能只有在交付该功能涉及的所有服务都已部分部署后才能被完全测试。

通过使注册表或负载均衡器知道金丝雀并让其将消息从指定的测试者路由到金丝雀版本，就可以将消息路由到金丝雀了。

如果新功能受功能开关的控制，那么在金丝雀版本上打开该功能的开关激活这些功能并使测试得以进行。

如果服务使用了向前和向后兼容，那么一旦新功能涉及的所有服务都已升级到新版本，测试就完成了。任何一种情况，你都应该仔细监控金丝雀，并且一旦发现错误就应该将它们回滚。

6.6.2 A/B 测试

第 5 章介绍了 A/B 测试。这是通过部分部署在生产环境中进行的另一种测试形式。“A”和“B”指的是服务的两个不同版本，要么呈现了不同的用户界面，要么呈现了不同的行为。这里，当将这两种不同版本呈现给用户时，测试的是用户的行为。

如果 A 或 B 在一些业务度量（比如已下订单）中表现出更好的行为，那么这个版本变成生产版本，而另一个版本则报废。

实现 A/B 测试与实现金丝雀很相似。注册表或负载均衡器必须知道 A/B 测试并确保一个用户要么由 A 行为的虚拟机服务，要么由 B 行为的服务器服务，但不能两个都服务。选择呈现版本 B 的用户可能是随机的，也可能是精心设计的。如果是精心设计的，那么可以考虑地理位置、年龄组（对于注册用户）或客户级别（比如，“金卡”飞行常客）等因素。

6.7 回滚

部署后的一段时间是新版本服务的试用期。它虽然经历了各种形式的测试，但它仍然不能被完全信任。第 1 章讨论的发布计划囊括了识别新版本的潜在不值得信赖之处，发布计划的规定之一就是测试回滚计划。回滚意味着恢复到之前的发布。也有可能前滚——也就是说，修正错误并生成包含错误修正的新版本。前滚本质上只是升级的一个实例，因此我们不在前滚上浪费笔墨。

由于回滚的敏感性以及前滚的可能性，所以自动触发回滚很少。应该有一个人在决策圈中，他决定错误是否严重到应该停止当前的部署。这个人随后决定回滚还是前滚。

如果带有版本 A 的虚拟机仍然可用，类似于在关闭所有版本 A 的虚拟机之前在蓝 / 绿部署模型中，可以仅仅通过将流量重新定向回这些虚拟机来完成回滚。处理持久状态问题的一个办法是，在试用期内，让版本 A 的虚拟机一直接收版本 B 所收到请求的副本。

然而，如果你正在使用滚动升级模型或者你无法简单地用版本 A 全部替换版本 B，那么你就要用更复杂的方法将版本 B 的虚拟机替换为版本 A 的虚拟机。新版本 B 在其生命期内可能处于 4 种状态中的一种：未安装、部分安装、完全安装但尚处试用，或已应用到生产环境。

其中的两种状态不可能回滚。如果版本 B 尚未安装，那它当然不能被回滚。一旦它已经提交，它也不能被回滚——尽管旧版本可以当作一次新部署而重新部署。如第 5 章所述，如果版本 B 已经提交，然后要移除版本 B 中已经激活的所有功能开关，这应该放到开发团队的活动清单中执行。

剩下的两种状态（即，版本 B 部分安装或完全安装但尚处试用）能够回滚。回滚策略取决于是否使用了功能开关以及开关是否已激活。它适用于剩下的两种状态。

- *未使用功能开关*。这种情况下，回滚虚拟机就是禁用那些虚拟机并重新安装运行版本 A 服务的虚拟机。
- *使用功能开关*。如果功能开关还没有激活，那么我们有之前的版本。禁用运行版本 B 的虚拟机并重新安装版本 A。如果功能开关已激活，先停用。如果这阻止了后续的错误，那么就不需要进一步的行动。如果不能，那么情况就像没有功能开关一样。

其余的情况处理持久性数据，而这是最复杂的。假设所有的版本 B 的虚拟机已经安装并且版本 B 的功能已经激活，但需要回滚。回滚到版本 B 已安装但功能未激活的状态只要关闭新功能就行，这是一个简单操作。当考虑持久性数据时，复杂性就随之而来了。

当发现错误时，关注点是错误数据已经写入数据库。处理错误的数据库值是精细操作，并且对业务有巨大的影响。这里我们给出了一个通用方法，但要小心使用。你肯定不想让情况更糟。

我们的通用方法是，回滚那些已激活新功能的服务所处理的请求，并使用旧的、可工作的服务来重放它们。我们先讨论完成回滚需要做什么，而后讨论可能发生的潜在问题。

为了回滚可能有问题的交易，需要先识别它们。可以通过为每个数据项维护一个谱系来做到这一点。这个谱系包括写数据项服务的版本号和触发写数据请求的标识符。它还涉及将足够多的请求数据记录到日志中，以便可以从中恢复一条从初始请求到写入数据的因果链。

当回滚被触发后，识别实现功能所涉及的服务的版本。这使你能够识别出由那些版本写入的数据项，接着，识别出要重放的请求。移除已识别的数据项并恢复所有被重写的数据项，清洗了数据库中的潜在错误地直接写入的数据。恢复被重写的数据项需要保持数据字段及其值的历史。有可能一个错误数据会级联触发依赖的行为。更糟的是，错误数据值可能对外部有影响。例如，客户可能已经看到了被大幅削减的票价并完成了购票。

通过维护数据项的谱系可以实现追踪级联行为并确定哪些数据值有可能的错误以及哪些已经躲过了系统。如果一个数据项的谱系包含了其依赖的数据项，那么就能准确定位到那些保存在数据库中的被依赖数据项并将其移除。对于那些外部可见的依赖行为，用日志记录任何外部可见数据项的来源，这使你能够确定错误功能的影响，但修正将变成一项业务工作。有些错误地被外部看到的数据可能并没有严重的影响，但有些确实有。确定外部可见错误数据所造成的影响需要特殊处理，并且要与业务决策者一起去实现。

一旦违规数据被清除掉，指定的请求就可以用旧版本的服务进行重放。这会重新生成那些已移除的数据，但是以没有错误的形式。这个策略的一个问题是，请求可能依赖已移除的功能。这种情况下，重放应该触发其中一个服务给出错误提示。重放机制必须知道如何处理这些错误。

如你已经感受到的，识别和修正数据库中的错误数据是一个复杂而微妙的操作，需要收集大量的元数据。

6.8 工具

存在大量用于管理部署的工具。对这些工具分类的一种方法是确定它们是否直接影响被部署实体的内部构件。如第5章所述，如果虚拟机镜像包含了需要的所有新版本的软件，那么你可以用新版本的虚拟机全部替换旧版本的虚拟机。这称为使用重度烧制的部署方法。或者，你可以使用工具改变虚拟机的内部构件，以便可以在不终止虚拟机的情况下用新版本替换旧版本。即便你终止带有旧版本的虚拟机，你可以启动一个新的轻度烧制的虚拟机，并在部署过程的后期从内部访问该机器来部署新版本。

比如，Netflix Asgard是一个开源的、基于Web的工具，它用来管理基于云的应用和基础设施。Asgard对虚拟机的内容不感兴趣。它使用包含新版本的虚拟机镜像并为这些镜像创建虚拟机。Asgard的一个功能是，它了解滚动升级这样的部署过程。它允许使用规格说明指定一个周期内升级的虚拟机数量。基础设施即服务（IaaS）供应商也提供了专用工具来调节虚拟机配置，这用作部署的一部分。例如，亚马逊允许用户将CloudFormation脚本当作一种参数化、声明式的虚拟机部署方法。CloudFormation脚本理解依赖关系和回滚。

Chef和Puppet是两个管理虚拟机内部项目的工具。它们可以替换虚拟机内部的一个软件的版本并确保配置设置遵循规格说明。

一个新兴的趋势是在部署中使用轻量级的容器工具，比如Docker。轻量级容器是操作系统级的虚拟化技术，用于在单个主机上（虚拟机或物理机）运行多个独立的操作系统。它们类似于虚拟机，但它们更小且启动得更快。

像Vagrant和Test Kitchen这样的镜像管理和测试工具有助于控制虚拟机和虚拟机内

部的项目。开发人员可以启动类生产环境来进行提交前测试和集成测试，以便发现那些仅在生产环境出现中的问题。

6.9　小结

将服务部署到多个虚拟机的策略包括蓝 / 绿部署和滚动升级。蓝 / 绿部署不会引入任何逻辑问题，但要求分配服务所需的两倍数量的虚拟机。滚动升级在资源使用方面更有效，但却引入了一些逻辑一致性问题。

- 单个服务的多个不同版本可能同时处于活动状态。这些版本可能提供了不一致的服务。
- 客户端可能假设依赖服务的一个版本，但实际上是由另一个版本服务的。
- 由于选择打包多个和依赖的服务以及多个团队执行同步部署，所以竞态条件可能存在。将选择的服务数量打包到一个虚拟机中经常要在资源利用、性能和部署复杂性之间进行权衡。

逻辑一致性问题的解决方法涉及使用某些功能开关的组合、向前和向后兼容，以及版本意识。

部署必须偶尔回滚。功能开关支持回滚功能，但当回滚一个部署时，对持久性数据的处理格外敏感。

部署还在实现业务连续性上发挥了重要作用。部署到不同的站点提供了度量连续性的一种方法。拥有一个包含复制机制的架构允许在意外电力中断发生时在更短的时间内进行修复和恢复处理。

有各种部署管理工具。轻量级容器和镜像管理工具的出现有助于开发人员更容易地部署到小规模的类生产环境中进行测试。

6.10　更多阅读材料

要了解更多关于执行升级的危险，你可以在 [Dumitras 09] 中发现关于该主题的经验研究。

Paxos 算法很难理解和实现。这就是为什么我们建议使用已经实现它并提供了更高级功能的库或工具。但如果你想要更好地理解这个算法，你可以看看来自图灵奖获得者作家 Leslie Lamport 的最新的（所谓简单的）解释。

ZooKeeper 基于 ZAB 算法，而且可以说使用得比 Paxos 更广泛。你可以在 http://zookeeper.apache.org 发现 ZooKeeper 的更多信息以及到一些更高级工具的链接。对于 Paxos 与 ZooKeeper 的 ZAB 之间的对比，参考 [Confluence 12]。

是否对模式进行更改还是回滚错误升级，你都可以在 [Sockut 09] 中找到重组实时数据库的更多资料。

要了解更多虚拟机镜像重度烧制和轻度烧制方法的优缺点，参考 [InformationWeek 13]。

在下面的链接中，你可以发现更多有关多个地区 / 虚拟机的服务之间的延迟信息。

- http://www.smart421.com/cloud-computing/amazon-web-services-inter-az-latency-measurements/
- http://www.smart421.com/cloud-computing/which-amazon-web-services-region-should-you-use-for-your-service/

至于工具，这里可以找到我们提及的各种工具的更多信息：

- Netflix Asgard：https://github.com/Netflix/asgard Amazon
- CloudFormation：http://aws.amazon.com/cloudformation/
- Chef：http://docs.opscode.com/chef_overview.html
- Puppet：http://puppetlabs.com/puppet/what-is-puppet
- Docker：https://www.docker.com/whatisdocker/
- Vagrant：https://www.vagrantup.com/

第三部分

横切关注点

第二部分描述了部署流水线的方面。它是从功能视角做出的描述，它关注部署流水线的组成部分。在本部分中，我们关注那些与流水线交叉相关的主题。第三部分包括 4 章。

在第 7 章中，我们讨论系统执行期间数据收集、数据处理和数据解释。对很多目的来说，这些数据都是至关重要的，比如错误检测和恢复、预测以及性能问题的识别等。

在第 8 章中，我们从多个不同的视角来讨论安全。一个视角是审计者必须评估你的应用或者环境的安全符合其需求的程度。我们也讨论保护部署流水线。在任何情况下，我们的讨论都将包括那些恶意尝试打破安全的行为和你组织内的人士做出的非恶意的、偶然的打破安全的行为。

除了安全以外，还有许多其他的质量属性对 DevOps 来说也是重要的。我们将在第 9 章中讨论这些属性。我们将说明，对于部署流水线的成功执行，质量是如何发挥重要作用的，这些质量属性包括可追踪性、性能、可靠性和可重复性。

最后，在第 10 章中，我们关注业务。如果没有来自业务其他部分（包括管理层）的支持，一个组织无法采用很多 DevOps 实践。这一章将讨论如何开发一个 DevOps 的业务计划，包括应该采取的测量类型以及如何实现对 DevOps 实践的增量采用。

第7章 监　　控

先获得真实情况，然后在合适的时候打破它。

——Mark Twain

7.1 概述

在软件开发和运维领域，监控有非常悠久的历史。最早的监控器是硬件设备，类似于示波器，在监控生态系统中还存在着这样的监控器。本章忽略历史而关注软件监控。软件监控涵盖非常多的监控类型以及与它们相关的关注内容。收集多层的指标（资源 / 操作系统 / 中间件 / 应用层）、绘图和分析指标、记录日志、生成关于系统健康状态的警告以及度量用户交互等活动都属于监控的内容。

Richard Hamming 说："计算的目的是洞察世界，而不是计数"。这种来自监控的洞察分为 5 种类别。

1）识别故障以及相关的缺陷，既包括执行时的，也包括故障发生后在反思会上发现的。

2）识别单一系统和一组交互系统的性能问题。

3）为长期和短期容量规划和付账目的而描述工作负荷。

4）度量用户对接口和业务产品的各种反应。在第 5 章和第 6 章中，我们讨论了 A/B 测试技术。

5）检测企图入侵系统的入侵者。

我们将术语*监控*定义为，观察并记录系统状态变化和数据流的过程。状态变化可以通过状态的直接度量或通过记录更新状态的影响部分的日志来表示。数据流可以通过记录内部组件与外部系统之间的请求和响应来获取。支持这些功能的软件系统叫作*监控系统*。

当提到监控工作负荷时，我们引入与运维活动相关的工具和基础设施。环境中的所有活动构成了数据中心的工作负荷时，这里包含以运维为中心的工具和监控工具。

在本章中，我们关注两方面：随着 DevOps 运动而出现的监控的新方面和挑战。DevOps 的持续交付、持续部署实践和强烈依赖自动化意味着更频繁的系统变更。使用微服务架构也使得数据流的监控更具挑战。我们将在 7.6 节中具体讨论这些挑战。以下列举了新挑战的一些例子：

- 持续变更下的监控是一件非常困难的事情。传统的监控高度依赖于异常检测。你知道正常运营时系统的状况。你设置指标的阈值并监控以检测异常行为。如果系统改变了，你可以重新调整指标。如果你的系统因为持续部署实践和云弹性而持续变化，那么上述方案就没有效果了。基于正常运营情况设置阈值将会在部署过程中产生大量误报。部署时候关闭警报，当系统处于一个不稳定状态时，可能错过某些严重的错误。第6章讨论了多个部署可能同时发生并且这些部署可以使阈值的设置复杂化。
- 云环境引入了不同的层级，从应用程序编程接口（Application Programming Interface，API）调用到虚拟机资源使用等。为不同的场景选择自上而下方法还是自下而上方法，以及平衡这些方法都不是一件简单的事情。
- 当采用了第4章讨论的微服务架构时，监控要求把更多的重心放到移动部分。它还要求将更多服务间的通信记录在日志中，以保证遍历的区域服务的用户请求仍然满足你的服务级别协议。如果出错，你需要通过对海量（分布式）数据的分析确定原因。
- 在大型分布式系统中管理日志成为一个挑战。当你有成百上千个节点时，集中收集所有的日志信息将变得非常困难，或者耗费成本也很大。因为日志记录的数量巨大、噪声和来自于多个独立源的日志的不一致性，所以对巨大日志集合的分析也是一个很有挑战的事情。

监控解决方案必须像基础设施的其他部分一样进行测试和验证。在各种环境中测试监控解决方案是测试的一部分，但非生产环境的规模可能无法达到生产环境的规模，这意味着监控环境在被部署到生产环境中之前只能完成部分测试。我们曾听说过，涉及监控的功能开关导致一个重要的因特网服务中止了45分钟。这个事件不仅强调测试监控软件的重要性，而且也强调了维护功能开关控制的重要性。

我们将按照监控什么、怎样监控、什么时候监控和怎么解释监控数据来组织本章的内容。我们提供了指向工具的指针，深入讨论了刚才描述的挑战，并提供了怎样解释监控数据的例子。

7.2 监控什么

大部分被监控的数据来自栈的不同级别。表7-1列出了可以从监控数据和收集到的数据的栈的部分获得的洞察。请注意，为了实现监控的大部分目的，需要把整个栈都包括在监控中。我们强调支持运维的工具是增加工作负荷、有故障，并应该被监控的应用程序。在第6章中，我们指出了在部署过程中来自竞态条件的故障。监控配置和资源规格说明文件的变更可以检测到这些错误。

被监控的基本项包含输入、资源和输出。资源可以是硬件资源，例如 CPU、内存、硬盘和网络，即使它们是虚拟化的。资源也可以是软件资源，例如队列、线程池和配置规格说明。输出包括交易和面向业务的活动。

我们将通过表 7-1 来讨论监控的目的。

表 7-1　不同级别的栈的监控目的

监控目标	数据来源
发现检测	应用和基础设施
性能降低检测	应用和基础设施
容量规划	应用和基础设施
针对业务的用户反应	应用
入侵检测	应用和基础设施

7.2.1　故障检测

物理基础设施的任何元素都可能发生故障。故障原因各不相同，从过热到老鼠咬了电缆等。整体故障相对容易发现：应该有数据流的地方没有数据了。部分故障却难以检测出来，例如，电缆没有固定住并且导致性能下降；在机器因为过热完全出故障前，它会经历间歇故障，等等。

检测物理基础设施的故障是数据中心提供者的责任。为操作系统或者其虚拟操作系统提供仪器将为数据中心提供数据。

软件故障可能是部分的也可能是整体故障的。整体故障相对容易发现。部分软件故障有很多原因，与部分硬件故障的情况相同。底层的硬件可能有部分故障；下游服务可能已发生了故障；软件或它的支持软件可能已被错误配置了，等等。

检测软件故障可以归纳为以下 3 种：

1）监控软件从外部点对系统进行健康检查。

2）系统内的特殊代理执行监控。

3）系统自己检测到问题并发出报告。

有些故障可能也显示为性能问题，接下来的章节我们开始对此讨论。

7.2.2　性能下降检测

性能下降检测毫无疑问是监控数据的最常见用途。通过对当前数据和历史数据客户和最终用户抱怨的研究很容易发现性能下降。理想状态下，监控系统能在用户被明显影响前就捕获性能下降。

性能度量包括延迟、吞吐量和使用率。

1. 延迟

延迟是指活动从开始到结束的时间。它可以按照不同的粒度等级进行度量。在粗粒度下，延迟是指用户发起一个请求到请求完成的间隔时间。在细粒度下，延迟是指在网络上发出一个消息到接收到这个消息的间隔时间。

延迟也可以在基础设施或应用层进行度量。在一台单独的物理机器内部度量延迟可以通过在开始活动前读取时钟，活动结束后再读取时钟，然后计算时间间隔的方式来获得。跨多台物理机器度量延迟有更多的问题，因为很难同步时钟。后面，我们会详细讨

论这个问题。

在报告延迟时，重要的是把这些延迟与它们度量的活动相关联。更进一步，响应用户请求的延迟是一个累加值，它是把请求完成前的所有活动的延迟累计，由于并发执行这种延迟会有所调整。在诊断延迟问题的原因时，了解为完成原始请求而执行的各个子活动的延迟是很有帮助的。

2. 吞吐量

吞吐量是单位时间内某个特定类型操作的数量。虽然吞吐量可以用于基础设施活动（例如，每分钟读取硬盘的数量），但它更多用于应用级别。例如，每秒事务数（Transaction per Second）是一个常见的报告指标。

吞吐量提供了一种系统范围的度量，它涉及所有的用户；而延迟只关注单个用户或客户端。高吞吐量可能与低延迟有关，也可能没关系。它们的相关性依赖于用户的数量和用户的使用方式。

吞吐量降低本身不是问题，它可能是用户数量下降导致的。这个问题可以通过比较用户数量和吞吐量来发现。

3. 使用率

使用率是资源使用的相对量，通常是通过在感兴趣的资源中插入探针来度量。例如，CPU 的使用率可能是 80%。高使用率可以用于提前预警延迟或吞吐量问题，也可以作为诊断工具用于发现延迟和吞吐量出现问题的原因。

资源可以在基础设施级别也可以在应用级别。硬件资源，例如 CPU、内存、硬盘和网络，最好通过基础设施来度量。软件资源，例如队列或线程池，通过应用或基础设施来度量，这取决于资源所在的位置。

理解使用率通常需要把使用与活动或者应用关联起来。例如，应用 1 使用了 20% 的 CPU，磁盘压缩占用了 30% 等。因此，把应用或活动与这些度量连接起来是数据收集的重要部分。

7.2.3 容量规划

我们区别对待长期和短期容量规划。长期容量规划是由人工完成的，从时间上来看，包括按天、按周、按月和按年。短期容量规划是自动执行的，它的频度是分钟级的。

1. 长期容量规划

长期容量规划要考虑工作负荷需求，要匹配真实的或者虚拟化的硬件需求。在物理数据中心，这涉及订购的硬件。在虚拟化的公有数据中心，它涉及决定分配的虚拟资源的数量和特征。在上述两种情况下，容量规划过程的输入包括：1）从监控数据收集的当前工作负荷特征；2）基于业务考虑和当前工作负荷所做出的未来工作负荷的预测。基于未来工作负荷、未来的吞吐量和延迟要求以及多种准备方案的成本，组织将挑选一个方

案并为它做预算。

2. 短期容量规划

在虚拟化环境中，例如云环境，短期容量规划是为应用创建一个新的虚拟机（VM）或删除一个存在的虚拟机。常用的方法是基于基础设施收集的监控信息进行判断和执行。在第 4 章中，我们讨论了多种基于当前工作负荷的虚拟机实例的分配方案。监控当前虚拟机实例的使用情况是每种选择的重要部分。

监控数据也用于公有云的计费。在第 2 章中我们描述了美国国家科学与技术协会定义的云的最核心特性，就是为使用付费。为了实现为使用付费，使用量必须能够判断出来，这是通过云供应商的监控来完成的。

7.2.4　用户交互

用户满意度是业务的一个重要元素。除了应用自身的实用性和品质以外，用户满意度依赖于 4 个可以被监控的元素。

1）*用户请求的延迟*。用户期望响应时间快。根据应用，响应方面的微小变化可能带来巨大的影响。Google 的报告显示，如果一个搜索页面从 100 毫秒延迟到 400 毫秒，那么就会明显地影响用户执行的搜索数量。亚马逊也有过类似报告。

2）*与用户交互的系统的可靠性*。前面，我们讨论过故障和故障检测。

3）*特定业务产品或用户界面修改带来的影响*。在第 5 和 6 章中，我们讨论了 A/B 测试。从 A/B 测试中获取的度量数据必须是对测试目的有意义的，数据必须与系统的变体 A 或 B 相关联。

4）*组织的特定指标集*。每个组织都有一套指标集，用于判断它们的产品和支持服务的效果。如果你运营一个相册网站，你会比较在意照片的上传率、照片的大小、照片的处理时间、照片的受欢迎程度、广告的点击率和用户的活跃度。其他组织将有不同的指标集，但用户满意度或组织提供的基于计算机的服务的有效性都是重要的关注点。

通常有两种类型的用户交互监控。

1）真实用户监控（Real User Monitoring，RUM）。真实用户监控本质上记录与一个应用的所有用户交互。真实用户监控数据用来评估真实服务水平的用户体验以及服务器端变化是否正确传播到用户端。真实用户监控通常是被动的，对应用的开销没有影响、不会增加工作负荷、不会改变服务器端的应用。

2）*综合监控*。综合监控类似于开发人员对应用进行压力测试。使用某些模拟系统或者使用实际的客户端软件（例如浏览器）为期望的用户操作写脚本。但是，测试的目的不是用高负载进行压力测试，而是为了监控用户体验。综合监控可以让你以系统化的、可重复的方式监控用户体验，而不依赖于用户当前怎么使用系统。综合监控是第 5 章讨论的自动化用户验收测试的一部分。

7.2.5 入侵检测

入侵者可以通过破坏一个应用程序入侵系统，例如，通过错误的认证或中间人攻击。应用程序可以监控用户和他们的操作，以判断这些操作是否匹配用户在组织中的角色且是否符合以往的操作习惯。例如，用户 John 有一个安装了应用程序的手机，该手机目前在澳大利亚，那么来自尼日利亚的登录尝试就是可疑的。

入侵检测器是一个软件应用程序，它通过查看异常来监控网络流量。这些异常可能是由未授权用户尝试破坏系统引起的，也有可能是由于违反了组织安全策略引起的。

入侵检测器使用多种不同的技术来识别攻击。通常它们使用来自组织的网络的历史数据来了解正常的操作是什么。它们也使用一些库，这些库包含了在各种攻击中观察到的网络流量模式。将网络上的当前流量与预期的（来自组织的历史记录）和异常的（来自攻击历史）流量进行比较，以判断当前网络是否存在攻击。

入侵检测器也能监控网络流量，以判断组织的安全策略是否受到来自非恶意行为的违反。例如，一位员工因为实验目的试图打开一个端口与外部进行通信，但是该组织的安全策略规定不允许特殊的端口与外部进行通信。入侵检测器可以发现这类违反策略的情况。

在 7.5 节中，我们将讨论入侵检测器生成警告和警报。与它们在所有的监控系统中存在误报和漏报一样，入侵检测器也存在这样的问题。

判断一个具体的数据异常是否反映一个入侵，不是一个容易的任务。7.8 节将用一个例子来具体讨论它。

7.3 如何监控

监控系统通常与被监控的元素进行交互，如图 7-1 所示。被监控的系统（图 7-1 中的系统 1、2……）可以是独立的应用程序或服务的集合，也可以是一个应用程序。如果系统主动地提供了被监控的数据（标注为“无代理”的箭头），那么监控是入侵式的且影响系统设计。如果系统不主动提供被监控的数据（标注为“基于代理”的箭头），那么监控是非入侵式的且不会影响系统设计。第 3 种数据源在图 7-1 上用“健康检查”箭头标注。外部系统可以通过健康检查、性能相关请求或事务监控来监控系统或者应用级别的状态。

通过代理或者非代理收集的数据最后都发送到中心存储库中（图 7-1 中的监控数据存储）。一般来说这个中心存储库是分布式的——所以它是逻辑上的中心而不是物理上的中心。从初始收集到中心存储库的每一步都可以进行过滤和聚合。判断过滤和聚合量的条件是：生成数据的大小、本地节点的潜在故障和必要通信的粒度。因为本地节点可能发生故障且数据变得不可用，所以从本地节点获取数据是重要的。将所有数据直接发送

到中心存储库可能导致网络阻塞。因此，在设置监控框架时，选择从本地节点到中心存储库之间的中间步骤以及在每一步过滤和聚合数据是重要的架构决策。

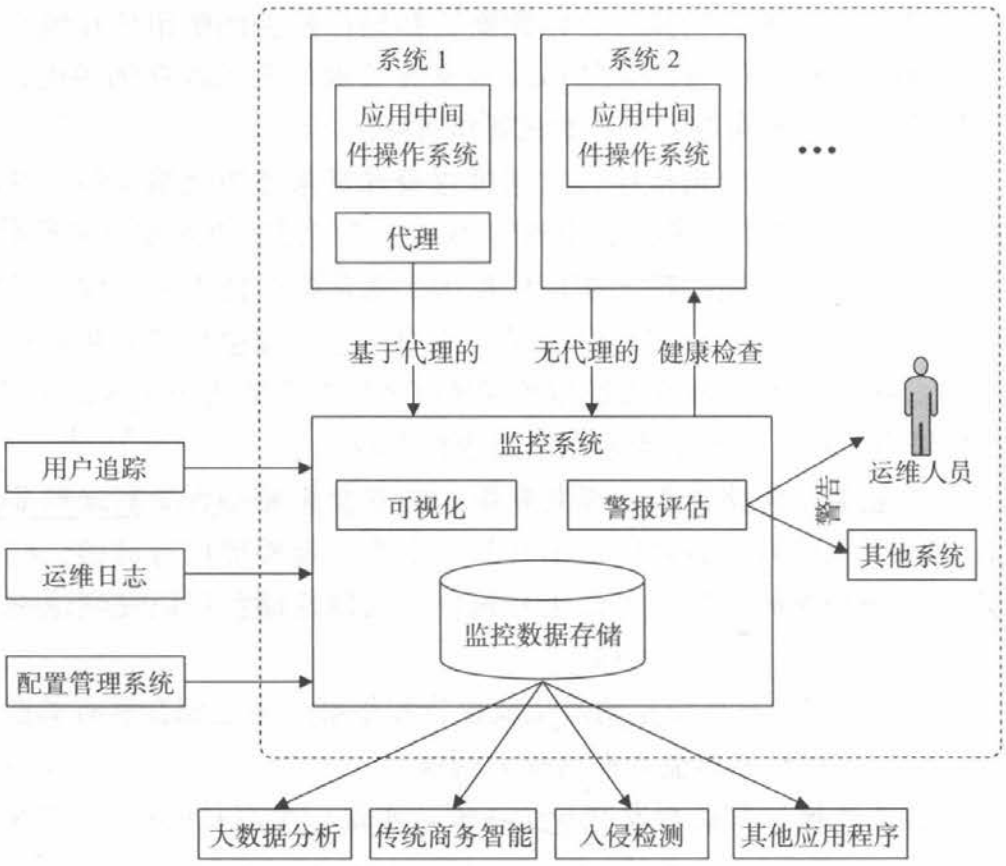


图 7-1　监控系统与被监控的元素交互 [标注法：架构]

做出过滤和合并决定的一个策略是考虑数据丢失的影响。有些数据表现为被瞬间读取，它可以被另一个瞬间读取集合短暂地替代。丢失这类数据对整体监控没有影响，也不会影响触发报警。

一旦监控数据被收集起来，你就可以做很多事情。可以配置警报来触发警告以通知运维人员或其他系统主要的状态发生了变化。图形化和仪表板的使用，可以将系统状态的变化可视化地展现给运维人员。监控系统也允许运维人员深入到详细的监控数据和日志中，这对错误诊断、根本原因分析和决定解决问题的最佳方案具有重要的作用。

目前我们已经描述了监控系统的传统视图，但监控系统和其他系统之间的新的交互作用正逐渐挑战这个视图。图 7-1 中，我们用虚线区域之外的部分表示这些内容。

可以对监控数据流和历史数据执行流式处理和（大）数据分析。不仅可以使用系统级的监控数据获得对系统特性的了解，还可以使用应用级和用户级的监控数据分析出用户行为和意图。

由于监控数据的不同使用需求不断增长，所以很多公司开始为监控系统和整体应用系统采用统一的日志和以指标为中心的发布 - 订阅架构。越来越多的数据类型，包括非

传统日志和指标数据，都放入统一的存储库中，其中各种其他系统（不管是否与监控相关）都可以订阅其感兴趣的数据。统一视图的含义是：

- 它明显减少了任何两个系统的耦合度。系统以发布 - 订阅方式与统一的日志交互，这可以使发布者忽略订阅者的特定身份，反之亦然。
- 它简化了多种数据源的集成工作。对监控数据的很多分析都涉及对多个数据源的关联。我们已经提到过，把业务指标和性能指标相关联。这些度量的来源不相同。使用中心日志存储库允许数据根据属性进行关联，例如使用时间戳而不是它们的来源。

当以对待系统级监控数据的方式对待应用程序和用户监控数据时，监控系统和被监控系统之间的界限就变得很模糊了——来自各个地方和各个层级的数据都对了解系统和用户情况有帮助。因此，当你考虑所有其他系统都将信息放入中心存储库并从中心存储库获取信息时，这里展示的架构就不再只是监控系统架构。

我们现在更详细地讨论该架构的多个方面，也就是说，获取监控数据的方法、监控操作、数据收集和存储。

7.3.1 基于代理的监控和无代理的监控

在某些情况下，被监控的系统已经有内部监控能力，它能够通过已定义的协议提供访问。例如，简单网络管理协议（Simple Network Management Protocol，SNMP）就是从服务器和网络设备收集指标的常用机制。这个协议在网络设备中非常有用，因为网络设备通常是作为一个封闭系统出现的，并且你不能在这些网络设备上安装监控代理。Windows 管理规范（Windows Management Instrumentation，WMI）提供了 Windows 系统访问管理数据的方法。你可以使用 SSH 等协议远程访问系统，获取可用的数据。当无法安装代理时，无代理监控就非常有用，它可以简化监控系统的部署。在 7.2 节中，我们讨论了那些给监控系统提供信息的应用程序。

应用程序响应度量（Application Response Measurement，ARM）是一个工业标准，它为应用程序触发动作提供了方法，例如请求一个外部的支持 ARM 的系统来开始或结束追踪一个事物并把一个事物在不同系统上所花费的时间关联起来。

基于代理的方案和无代理的方案都有各自的优势和弱点。无代理方案在部署和维护方面占优势。但是，如果存储库在你网络的外面，那么它是不安全的，因为，系统需要开放很多端口且防火墙规则会放宽以允许系统的不同层与外部世界进行数据通信。相反，主机上的代理可以与操作系统和应用程序进行本地通信并从一个通道发送所有收集到的信息。同时，基于代理的方法能够优化网络传输和处理开销。

除了从系统内部收集监控数据外，还可以从外部视角收集信息。可以设置健康检查对系统进行周期性扫描，或者从外部用户的视角执行性能监控。

如前所述，多种类型的信息被认为是监控信息或者至少有助于监控数据分析。在设

计系统时，需要考虑如下问题：信息来源是什么？信息与应用程序和监控架构怎样匹配？质量影响如何？

7.3.2　监控运维活动

有些运维工具（例如 Chef）能够监控资源（例如配置设置），以判断它们是否与预定义的设置一致。我们也提到过监控资源规格说明文件来识别变更。这两种类型的监控最好都通过代理执行，代理可以周期性地采样实际值和包含这些值的文件。

视基础设施为代码意味着基础设施采用与其他应用程序一样的方式给监控提供数据。它的实现方式包括：基于代理的、无代理的或外部的。

在第 14 章中，我们将讨论如何执行运维工具和脚本行为的细粒度监控。这种监控包括对监控数据的断言。例如，在滚动升级过程中，用运行应用程序的新版本的虚拟机替代服务采用的一组虚拟机。然后，你能预料到，剩余机器的 CPU 平均使用率会有所提升。

7.3.3　收集和存储

监控的核心是记录和分析时间序列数据，换言之，时间戳数据点的序列。这些数据点通常以连续的时间间隔及时获取，它们显示出状态和状态变化的某些方面。此外，被监控的系统将生成不同安全等级的时间戳事件通知。这些通知通常输出为日志。监控系统可以直接度量或收集存在的数据、统计、日志，然后将它们转换为指标，这些指标的属性包括时间和空间。然后使用预定义的协议把这些数据传输到存储库。这些输入数据流经常需要进一步处理为时间序列，并存储在一个时间序列数据库中。3 个关键的挑战是：按时间关联监控项、按上下文关联监控项以及处理监控数据的容量。

- **按时间关联监控项**。在分布式系统中的时间戳可能是不一致的。一个集群中的不同节点的时钟可能有数毫秒的差别。不同集群的不同节点的时钟的差别会更大。因此，通过时间戳判断两个项是时间相关的或者甚至是否是顺序发生的，会带来问题。使用时间间隔来判断关系而不使用精确测量是一种方法，虽然它可能导致遗漏一些关联关系，如果两次相关测量的时间差间隔大于定义的用于判断关联关系的时间窗口。
- **按上下文关联监控项**。消息的上下文通常与消息本身一样重要。假设你正在执行滚动升级并在每个升级中替换两个实例。不同的节点可能生成关于这个实例升级的日志消息。如果无法判断这两个消息对应于同一个实例，那么想重建时间序列并诊断问题将是非常困难的。在监控数据流时也会遇到相同的问题。来自系统的实例的特定消息是该实例的输入的直接响应，并且是对用户请求或外部事件的间接响应。对于特定消息，识别其直接和间接触发器，对分析性能问题或故障是非常重要的。

- 监控数据的容量。你可能需要一个保留策略来妥善处理收集的数据容量。设置一个一刀切的监控数据保留时间可能不是最优的：你可能注重最近刚过去的监控数据，对它采取细粒度的存储，逐渐地对久远一点的数据采取粗粒度的数据聚合。不同的策略依赖于当前可用的存储容量和指标的关键程度。为了快速的查询处理或显示，你可能选择将基本数据处理为带有索引的专用视图。

一种流行的时间序列数据库叫作轮转数据库（Round-Robin Database，RRD），它采用良好的保留策略配置能力来存储和显示时间序列数据。随着我们跨入大数据时代，大数据的存储和处理方案将逐渐应用到监控数据中。你可以把监控数据当作输入到流式系统用于实时处理的数据流，并与大量历史数据相关联。你可以把所有数据导入大数据存储系统，例如 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS），或者把它存档在相对不太昂贵的在线存储系统中（例如亚马逊的 Glacier）。

7.4 什么时候变更监控配置

监控是基于时间或基于事件的。基于时间的监控按照时间间隔进行报告，但是时间间隔不需要对所有应用程序和应用程序的执行都统一到一个不变的时间间隔上。时间频率和事件的生成都应该是可配置的，并且随着数据中心内发生的事件而更改。以下列举了一些事件的案例，它们可能改变监控配置：

- 警告。我们在下一节讨论警报和警告。警告的一个结果是可能增加采样的频率。如果警告没有转变成警报，那么就可以降低频率。
- 部署。我们在第 6 章中讨论的部署场景可以触发监控变更。这包括：
  - 金丝雀部署。因为金丝雀部署的目的是测试新版本，所以需要密切监控新版本。
  - 滚动升级。我们讨论了一些可能的竞态条件（race condition），这取决于你打包到虚拟机中的服务。密切监控可以更容易且更快速地发现竞态条件。
  - 功能激活或者失活。激活或者失活功能将改变服务的行为。这些变化应该触发监控配置的变化。
- 对任何基础设施软件（包括 DevOps 工具）的变更。与对应用程序本身做出变更一样，对基础设施软件的变更也能够影响应用程序的行为或者性能。
- 对配置参数的变更。在现代分布式系统中，错误的主要来源之一就是错误的参数。在参数变更以后进行更细节的监控可以帮助更快地检测问题。

7.5 解释监控数据

现在假设监控数据（基于时间的和基于事件的）都已经收集到中心存储库中。其他系

统和工程师会不断地添加数据和不断地检查数据。我们从描述一些关于日志消息内容的通用原则开始。

7.5.1　日志

日志是事件的时间序列，因为它是按时间排列的记录序列。记录一般追加在日志的最后。与直接记录状态不同，日志通常记录可能导致系统状态变化执行的动作。变化的值本身可能没有记录到日志中。

日志在监控中起着非常重要的作用，尤其是在 DevOps 设置中。开发活动中，程序员非常熟悉应用日志，他们在应用日志中输出系统状态和动作以帮助他们的开发、测试和调试活动。然后生产环境部署后大部分日志将关闭或移除，这样只有警告和严重信息将记录在日志中并显示出来。开发人员写的日志通常是为了开发人员自己使用，而不是为了运维人员使用。DevOps 运动的动机之一就是把运维人员当作首要干系人，这意味着日志应该能被运维人员使用。这些日志的数据源不仅是应用。Web 服务器、数据库系统和 DevOps 流水线都产生日志。另一种重要类型的日志是由运维工具输出的日志行组成的。当使用升级工具升级系统、使用迁移工具迁移系统、使用配置管理工具重新配置系统时，将关于这些运维操作或者变更历史记录到日志中——这些对于错误检测、诊断任何操作（包括由 DevOps 流水线触发的）都是非常重要的。

在运维中使用的日志可以发现和诊断问题。在调试中使用的日志可以发现错误。在问题后的取证阶段使用的日志可以用来理解导致特定问题的事件序列。下面列举了一些生成日志的一般规则：

- 日志应该有一致的格式。因为一些日志是由你控制范围以外的第三方系统生成的，所以这一条并不总是能够满足的。然而，在你控制范围内的日志，必须要以一致的格式生成。
- 日志应该包含解释为什么要生成这类日志消息。例如，“检测到错误条件”或“追踪代码”等标签可以应用到日志中。
- 日志条目应该包含上下文信息。上下文信息不仅仅是日期和时间，还应该包括支持追踪日志条目的信息：
  - 代码中的日志条目的来源。
  - 生成日志条目时进程执行的进程号。
  - 导致进程执行这个日志生成器的请求的请求 ID。
  - 生成日志消息的虚拟机的 VM ID。
- 日志应该包含筛选信息。将日志消息收集到一个存储库中，该存储库可以通过查询访问。严重级别是一个筛选信息的例子，警告级别是另一个筛选信息的例子。

7.5.2 绘图和展示

一旦你拥有了所有相关数据，以不同方式将其可视化将是很有用的。大部分监控数据是时间序列数据，它适合绘制出来。一个灵活的系统应该允许你对绘制内容以及怎样绘制拥有完全的控制。一些监控系统内置了很强的可视化能力。还有一些专门用于可视化和查询的系统，例如 Graphite，它支持对大数据进行实时的绘图。

你可以设置一个仪表盘，它在聚合层级上展示系统及其组件的最重要的实时指标。在检测问题时，你也可以交互式的深入细节或者通过历史进行导航。有经验的运维人员将通过图的可视模式来识别问题。图可能有尖峰、突发、周期性变化、稳定的上升和下降趋势或者零散的事件，在被监控的状态和环境的特征语境下，所有这些都应该被理解。在一个运行在共享物理资源上的虚拟化环境中，或者在一个持续部署设置中，将有大量合理的变更在进行，例如资源伸缩、资源迁移以及滚动升级。因此，看得见的异常可能并不一定意味着问题。对运维人员来说，查看图并且在复杂的设定中指出哪些交互导致了该感知图将变得越来越具有挑战。同样，这也对警告系统和警报配置带来了挑战。

7.5.3 警报和警告

监控系统向运维人员提供重要事件的信息。这些信息以警报或警告的形式提供。从技术上说，警告是为了通知的目的，早于警报（例如，数据中心的温度上升），相反，警报需要运维人员或其他系统的响应，需要他们采取行动（例如，数据中心着火了）。警告和警报都是运维人员配置以后生成的。警告和警报可以被事件触发（例如，某个特定的物理服务器失去了响应），也可以由于某个数值超过了阈值（例如，对一个特定的硬盘来说，响应时间超过了一个可接受的数值）而触发，也可以由数值和趋势的复杂组合而触发。

理想地，监控系统生成的每一个警报都代表一个真实的问题，需要引起重视。当警报出现时，警告将提供更多的信息以便做进一步的分析和诊断，并提供补救的指导信息。遗憾的是，我们无法生活在理想世界中。所以，典型问题是：

- 怎样配置监控系统以减少误报（不需要响应的警报）和漏报（需要采取行动但警报没有提示）？
- 怎样配置监控系统使得警告能提供诊断警报的所需信息？

在监控系统中，将包含覆盖系统很多方面的大量指标，生成警告或者警报有时需要折中处理。运维人员会收到误报，或者从不同通道上收到对同一事件的一堆警告。在这样的情景下，运维人员很快就会变得“警告疲劳”而选择忽略警告或者粗暴地关闭警告。另一方面，如果你尝试减少误报，你可能会错失重要事件而导致漏报。如果警报条件设置得很具体，那么你会在微小错误发生的早期就收到很多信息——但是当系统随时间而变化或者当系统表明暂时受到一些合理但未知的运维活动干扰时，你将面临警报效能减少的风险。持续部署和云弹性会加剧这个问题。如你所见，为监控系统确定一个正确的配置不

是一件简单的事情，它将依赖于环境和从未接触问题的严重程度。

一些用于改善警告和警报有效性的通用规则是：

- 为警报引入背景信息。这可能包括在特定的时间和行动中禁用某些警告。例如，当更换一台物理机器时，就应该禁用报告这台计算机健康状态的警报。其他复杂的背景与外部事件或干扰运维有关。
- 如果发生某些事件则不仅要触发警报，而且，如果期望发生的事件未发生，则也要设置触发警报。这有助于我们训练和测试警报机制，因为当期望发生的事件没有发生时我们应该设置警报。
- 将那些可能指向相同事件的不同警告汇聚在一起。
- 设置清晰的严重等级和紧迫性等级，这样接收到警告的人或系统可以采取相应的措施。

7.5.4　诊断和反应

运维人员经常使用监控系统来诊断问题，并观察减缓和恢复的过程。然而，监控系统不是交互的、自动化的诊断系统。因此，运维人员只能以特设的方式尝试关联事件、探索细节、执行请求以及检查日志。同时，他们手动触发更多的诊断测试和恢复动作（如重启进程或隔离有问题的组件）以及从监控系统中观察它们的效果。

在前面的章节中我们讨论了可靠性工程师。可靠性工程师的技能核心是不确定性存在问题上的诊断能力。一旦问题被确定，通常反应就非常明确了，有时，可能的反应有不同的业务后果。如果反应的业务后果引起问题，则组织的升级程序就应该指出谁做决定。

7.5.5　监控 DevOps 过程

DevOps 过程应该被监控起来，这样可以优化过程并且发现问题。在第 3 章中，我们讨论了过程的改进。这样的改进依赖于获取信息。

Damon Edwards 列举了 5 个重要的监控事项：

1）业务指标。

2）周期时间。

3）发现错误的平均时间。

4）报告错误的平均时间。

5）丢弃（重复工作）的数量。

可以观察到这 5 个值的原始数据来自多个数据源和多个系统。如前所述，能够把来自多个数据源的数据关联起来对于解释监控数据是非常重要的。

7.6 挑战

本节深入讨论 7.1 节中提到的 4 个挑战。这些挑战是由 DevOps 实践和现代计算机环境引发的。

7.6.1 挑战 1：持续变更下的监控

在运维背景中的问题是一个偏离了正常行为的信号。正常行为假设系统已经稳定运行了很长时间。然而，在一个大型复杂环境中，*变更是常态*。我们不是讨论变化的工作负荷或者应用的动态方面，尽管那些都是人们都非常期望了解的。新的挑战来自于：1）云弹性，它使得基础设施资源变得更灵活；2）自动化的 DevOps 运维，它能触发很多零散的运维操作（例如升级、重配置、备份）。零散的运维、持续部署和部署实践使得软件变更更加频繁。我们已经看到了一天多次将新版本部署生产环境中将成为惯例。每次部署都是对系统的一次变更，都可能影响监控。此外，这些变更可能同时发生在应用或基础设施的不同部分。

你使用系统过去的监控数据来完成新系统的性能管理、容量规划、异常检测和错误诊断能达到什么程度？在实践中，运维人员可能关闭监控以减少有计划维护和升级过程中的误报。当变化成为常态时，这就会导致没有监控——就像飞舞中的盲人。

一种技术是仔细地识别数据的未改变部分。例如，使用没有计量单位的数据（例如，比率）。可以发现，尽管单个变量频繁变化，但两个变量的比率是相对稳定的。另一种技术是主要监控那些已经变化的事情。

通过监控新系统的一小部分推出时在生产环境上产生的问题的方式，金丝雀测试具备了很多优点。我们在第 6 章中讨论了这些优点。一种技术是将金丝雀的性能与历史性能进行对比。如果新功能改变带来的变化无法被合理解释的话，那么这种变化可能意味着出了问题。

与持续变更中进行监控相关的另一个挑战是监控参数的规格说明。当系统不是特别复杂且相对稳定时，说明需要监控的内容、设置阈值以及定义警告逻辑可以使用人工的方式来完成。过去，大型的监控重新配置通常发生在主要基础设施变更和迁移到新的基础设施的时候，例如，迁移到虚拟化环境或云环境中。新软件发布每隔几个月才进行，这将有宽裕的时间来调整监控。即使在这样的环境中，在偶尔变化中设置和维护监控系统所带来的复杂度仍然经常被监控专家当作最大的挑战。

系统基础设施和系统本身的持续变更使得监控参数的设置变得复杂。在基础设施这边，第 5 章提到，即使向相同的虚拟机提交请求，仍然存在巨大的性能差别。这些差别来自于你无法控制的因素，如你得到的 CPU 的类型。你的监控可能需要调整以适应这样的变化，或者你可以配置缩放控制器以便用新的虚拟机来替换性能下降的虚拟机（期望更幸运一些可以获得更高性能的虚拟机）。

因此，尽可能地自动化警报、警告和阈值的配置是有意义的。监控配置过程是另一个 DevOps 过程，它可以也应该被自动化。当你提供一台新服务器时，部分工作是在监控系统中自动地注册这台服务器。当服务器停止使用时，应该自动触发注销流程。

我们讨论了作为变更结果的变更配置，但讨论中假设是变更配置的规则是手动设置的。有些阈值可以通过底层变更自动地推算出来。其他阈值可以通过时间积累而自动地学习。例如，在对一小组服务器所做的金丝雀测试中，监控结果可以作为整个系统的新基准，并且被自动地聚集起来。

7.6.2　挑战 2：自下向上与自上向下和在云中的监控

监控的主要目标是尽可能快地发现缺陷、错误或小规模的故障，这样你能够在早期对它们做出反应。为了实现这个目标，自然而然地采用自下向上的方式进行监控：在理想状态下，根据聚合值，低层中的错误和单个模块中的错误，可以在它们传播和影响到上层应用服务器或者应用本身前，就能够被发现。这里有两个挑战。

第一，将有更多的单个模块级别和其他低级别的内容被监控。你有一个由多个组件组成的应用，它部署在上百台服务器上，这个部署依赖网络和存储组件的支持。一个根源可能触发组件之间的严重问题。在现实环境中，将这些现象相关联并找到根源是非常困难的。

第二，该挑战与云弹性和自动化导致的持续变更有关。在云中，低层基础设施和服务器之间有合法理由（例如，防止服务器漂移的终止操作、伸缩以及滚动升级）和非法理由（例如，实例失败或资源共享不确定性）。从合法理由中识别非法理由是巨大的挑战。

采取自上而下方法来监控基于云的和高复杂度的系统是解决这些问题的一种尝试。你监控上层或者聚合数据，如果你注意到顶层上的问题你再以智能的方式深入低层数据。低层数据仍然必须被收集但不会系统化地监控错误。低层数据的收集仅仅做到性能、存储和运输开销允许的程度。这不是“银弹”，是有原因的。首先，你可能会失去尽早发现问题的机会，当你在上层注意到有错误时阻止影响的扩大可能已经来不及了。其次，怎样深入到低层数据是一个问题。你检测到上层问题的时刻距离低层根本原因出现的时刻可能已经过去相当长的一段时间了。现代分布式系统有内置的容错机制来掩盖故障和错误，防止它们在系统层面出现并影响终端用户体验。最重要的一点是，当最初故障发生直到它扩散到整个系统变得明显，可能经过很长一段时间。你不能简单地依赖上层错误检测的时间戳。而且，你也不能假设与原始问题相关的指标和日志还仍然在那里：它们可能随着网络的僵死节点或区域一起消失了。在大型系统中尝试将所有相关数据运输到一个安全地方是一个非常大的挑战，因为每秒都会有百万级的内部和外部请求。

没有简单的解决方案。自下而上和自上而下监控都是很重要的，在实践中应该结合在一起。背景信息通常比时间戳更有用。如前所述，把关于变更的运维知识合并到监控数据中，是一种关联事件的更好方法。

7.6.3 挑战3：监控微服务架构

在前面的章节中，我们讨论了DevOps对架构带来的一个结果是采用微服务架构，它使得可以为每个微服务组建一个独立团队。然而，这样会让你的系统变成扇形系统或深层次系统。每个外部请求都可能要穿越大量内部服务才能得到响应。如果任何一个服务响应得慢，那么整个响应时间就会拉长。在大型系统中，在任何给定的时间内，一部分或另一个部分都可能会经历一定程度的减慢，这可能会导致全部响应中出现一部分不可被接受的情况。第2章讨论了长尾响应。微分区和选择复制能让迁移更简单，它适用于把服务从有问题的网络部分中迁移出来。对相同服务的多个请求进行监控并确定只有一个响应是必需的是一个挑战。

另一个挑战是微服务的兴起怎么识别并修复响应慢的节点。我们曾提到尽早确定偶尔发生的性能问题的困难。在具有许多节点的微服务架构中，在这些仍然工作的节点中找出响应慢的节点是非常困难的。问题是，什么是“慢”？怎么选择合适的阈值？我们会在第13章的案例研究中讨论一些解决方案。

7.6.4 挑战4：处理大容量的分布式（日志）数据

在大型系统中，监控每件事情会引入性能、传输和存储方面的巨大开销。一个大型系统很容易生成数百万个事件、指标数据，并且每秒都产生大量的日志条目。关于数据量的考虑有如下点：

1）在很小时间间隔内收集指标的性能开销是巨大的。取决于系统当前的状态，运维应该使用变化的和可调节的时间间隔，而不是一些固定的时间间隔。如果有异常的初始迹象或者当出现了一个偶尔发生的操作时，你可能想用细粒度监控，并且当情况解决或操作结束时你可以返回到大的时间间隔。

2）应该使用现代分布式日志或消息系统来进行数据收集，而不是自己构建一个。分布式日志系统（如Logstash）能收集所有种类的日志，并在数据运输前进行大量的本地处理。这种类型的系统允许你减少性能开销、消除噪声，甚至在本地识别错误。LinkedIn开发了Kafka，一个高性能的分布式消息系统，它主要用于日志聚合和监控数据收集。它采用面向事件的架构，并将输入的数据流与处理过程解耦。

3）随着大数据分析的出现，研究人员开始使用高级机器学习算法来处理噪声、不一致性和大容量数据。这是一个值得注意的领域。

7.7 工具

现在有很多可供使用的监控系统和工具，都来自开源社区和商业公司。因为术语监控的概念太复杂，所以通常很难比较那些监控系统和工具。下面列举了一些代表性的

产品：

- Nagios：因为 Nagios 其丰富的插件资源，所以它大概是最广泛使用的监控工具。插件就是基本代理，它收集你感兴趣的指标。有一个庞大而活跃的社区维护这些插件。但是，Nagios 核心是一个大型的警告系统且只有很少的功能。在处理云环境时 Nagios 也存在一些力不从心的地方。在云环境中，服务器经常会上线和下线。
- Sensu 和 Icinga：许多系统试图改善 Nagios。Sensu 就是一个高度可扩展的和可伸缩的系统，它能在云环境中有效发挥作用。Icinga 是 Nagios 的一个分支。它关注更具可扩展性的分布式监控架构并易于扩展。Icinga 拥有比 Nagios 更强大的内部报告系统。以上这 2 个系统都能够重用 Nagios 的庞大插件库。
- Ganglia：Ganglia 最初被设计成收集集群指标。它把节点层的指标复制给附近节点，以防止数据丢失和与中心存储库的过度通信。很多 IaaS 供应商支持 Ganglia。
- Graylog2、Logstash 和 Splunk：这 3 个是分布式日志管理系统，是为大量基于文本的日志系统量身定做的。它们有前端提供对日志的综合研究和强大的搜索功能。
- CloudWatch 和类似工具：如果你使用公有云，云供应商通常会提供一些监控解决方案。例如，AWS 提供了 CloudWatch，它允许以固定时间间隔收集数百种指标。
- Kafka：如前所述，由于实时收集大量日志和指标面临巨大的挑战，被其他系统多次使用，设计了一些专用系统实现收集和分发的功能。Kafka 就是一个发布 - 订阅模式的消息系统，它既能用于监控又能用于其他目的。
- 流处理工具（Storm、Flume、S4）：如果你需要持续地收集大量的日志和指标，那么你正在有效地创建监控数据流。因此，可以使用流处理系统来处理监控数据，即使以实时方式。

应用性能索引（Application Performance Index，Apdex）是由一个企业联盟制订的公开标准。它定义了计算中报告和比较软件应用程序性能的标准方法。它的目的是将测量结果转换为对用户满意度的洞察力，通过定义统一的方法来分析和报告所测量性能满足用户期望的程度。

7.8　从监控数据中诊断出异常——Platformer.com 的案例

我们认为监控的两个原因是识别性能问题并发现系统中的入侵者。在本节中，我们描述来自 Platformer.com 公司的一个小的案例研究，它展示出了 3 个方面的内容：

- 在性能数据中识别异常的两个原因并不总是简单的。
- 确定监控应用程序的性能是开发人员的职责还是运维人员的职责并不总是简单的。
- 不同组织实体之间缺少协作将带来开销。

我们先展示数据收集的背景，以此开始我们的讨论。接着我们讨论怎样观察数据和分析数据。最后，我们对这个事故对 DevOps 和职责带来的思考进行总结。

7.8.1　背景

Platformer.com 是澳大利亚一家平台即服务（Platform as a Service，PaaS）供应商。它提供应用程序的市场，例如内容管理解决方案、客户关系管理等，以及数据库和其他底层系统。通过它们的接口，客户可以指定何时、如何伸缩他们的系统、如何实现灾难恢复等。价值来自比 IaaS 提供更高级的抽象服务，这样客户不用处理所有理解和管理 IaaS 服务的复杂性，就可以获得类似的好处。客户可以避免被特定的云供应商锁定，因为它提供了足够的相同接口来支持多个云供应商。

Platformer.com 的客户在基础设施服务上有 3 个选择：

1）*访问第三方云供应商*。Platformer.com 支持 AWS、微软 Azure、Rackspace 和 OrionVM 这些第三方供应商。客户通过 Platformer.com 的门户网站访问第三方云供应商。

2）*在客户的私有云上访问*。这个选项将 Platformer.com 软件安装到客户的私有云上。

3）*在客户的私有数据中心上访问*。这个选项与使用客户的私有云类似，但不需要客户采用云方案。

Platformer.com 使用分层架构提供它们的服务，如图 7-2 所示。该架构优势是应用程序编程接口给客户提供了 Platformer.com 常用服务的视图，客户不用关心交付机制和用于显示返回给客户监控信息的仪表盘。

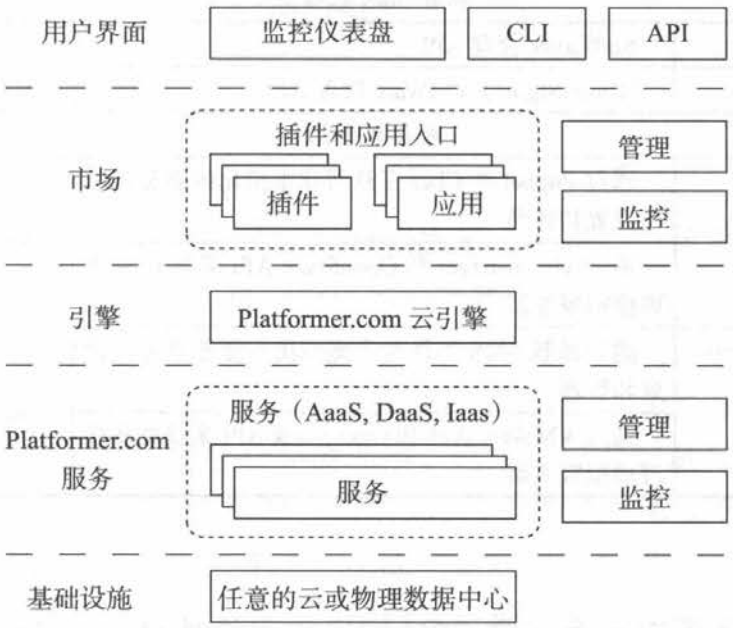


图 7-2　Platform.com 架构 [标注法：架构]

本讨论使用 Platformer.com 的客户叫作 PhotoPNP。他们是一个非营利组织，提供了展览、教育和图像出版的在线服务。他们依赖 Platformer.com 的服务来提供基于 Web 的内容管理解决方案 Joomla，也集成了电子商务应用。

7.8.2　数据收集

当提供在基于完全不同的平台上执行一个应用程序编程接口时，像 Platformer.com 这样的 PaaS 供应商必须或者提供他们支持的最小化的通用平台接口（所有平台都支持的服务），或者在一些平台上模拟其他平台上提供的服务。

在监控方面，Platformer.com 提供 CPU、硬盘、内存和网络性能的度量。这些度量为 Platformer.com 负载均衡活动需求、在需要的情况下提供虚拟机或者销毁现有虚拟机提供了数据支持。度量通过仪表盘展示给 Platformer.com 的客户。

因为 Platformer.com 支持多种平台，所以它们使用了多种度量工具，这取决于底层的平台。表 7-2 展示了底层平台和在每个平台上使用的监控解决方案。

表 7-2　与平台相关联的监控解决方案

基础设施供应商	管理解决方案	监控解决方案
公有 IaaS 供应商		
AWS	AWS 管理 API	AWS CloudWatch
Rackspace	Rackspace 管理 API	Rackspace 云监控和警告
OrionVM	OrionVM API	Nagios 或 AlienVault
微软 Azure	Azure 管理 API	Azure 监控和警告
IBM/SoftLayer（公有的）	SoftLayer 管理 API	SoftLayer 综合监控和警告
DigitalOcean	DigitalOcean 管理 API	Nagios 或 AlienVault
私有 IaaS 供应商		
IBM/SoftLayer（私有的）	SoftLayer 管理 API	SoftLayer 综合监控和警告
Telkomsigma	（CloudSigma）VMWare 管理 API	Nagios 或 AlienVault
内部部署 IT		
物理服务器（Linux）	通过 Puppet 和 Chef 工具对非虚拟化的服务器进行设置和管理	Nagios 或 AlienVault
	通过 virt-manager 和 OpenStack API 来构建和管理虚拟服务器	
物理服务器（Windows）	通过微软 SMS 工具对非虚拟化的服务器进行设置和管理	Nagios 或 AlienVault
	通过 VMWare API 和 OpenStack API 来设置和管理虚拟服务器	

7.8.3　检测异常

PhotoPNP 的服务器有一个正常 CPU 使用率大约是 5% 的 CPU，但是在 9 月 17 日和 18 日，这个值跳到了 17%。这个尖峰与收集的其他资源指标的变化相关联。CPU 负

载中的这个尖峰是系统中有入侵者的征兆。——这时，在一台 PhotoPNP 服务器上。Platformer.com 开始关注，调查为什么会发生这个尖峰。最后发现 PhotoPNP 在晚上给潜在用户开放了系统并导致了这个情况。PhotoPNP 使用 Google 分析（Google Analytics）检查了这段出问题用户层的指标数据，最后证实了这个调查结果。

7.8.4 思考

这个案例引发了一系列不同的争论。

- 在 DevOps 社区有一个讨论，谁应该是观测数据中异常事件的第一响应人。在这个案例中，我们看到平台供应商没有足够的观察力把异常看作正常应用级需求。因此，如果开发团队是第一响应人，那么混乱就不会发生。另一方面，如果 CPU 使用中的尖峰是由入侵者导致的，则使应用程序的开发者成为第一响应人会导致异常处理的响应滞后。此外，要求应用程序开发者能够发现入侵就等于要求他们具备远超出他们能力的专家级技能。
- 入侵者可能已经入侵了 PhotoPNP 服务器的迹象来自 CPU 使用率的检查，但对负载的归因分析需要引入应用级别的指标数据，而不仅仅是系统级别指标数据。

  前面我们讨论了使用日志管理系统将来自不同源的日志和指标相关联。在 Platformer.com 的案例中，应用级指标是不可用的，因为它们的可视度仅仅局限在基本的系统级指标。
- 如果 PhotoPNP 已经通知 Platformer.com 他们正在计划一个会导致额外负载的事件，那么就不会发生有潜在入侵者的猜测。在 Platformer.com 的案例中，业务级实体在另一个组织中——同一个组织中这类沟通问题也很容易发生，所以这是一个在业务和 IT 部门之间缺乏协作的例子。

7.9 小结

至少有 5 个目的证明需要进行监控：检测故障、诊断性能问题、规划容量、获取数据以洞悉用户互动以及检测入侵。每一个目的都需要不同的数据和不同的解析数据方法。

已经有很多监控系统，常见的结构允许人们利用基于时间和事件的信息收集。常见的结构也会迎合那些监控感知的应用（Monitoring-Aware）和监控不感知的应用。典型的监控流水线把监控的数据存放在中心存储库中，这里可以对它进行查询并且用于生成警报、警告和可视化内容。在执行分析时，将来自多个数据源的数据相关联是非常重要的。

持续部署实践提高了变更的频率——从应用到底层基础设施。面对这种变更，留给监控方案的观察和调整时间就非常少了。这要求自动化监控配置过程本身，包括自动的智能、动态调整警报阈值。云环境使得系统的某些部分更不透明，并带来了基础设施层

的持续变化。监控工具需要适应这样的环境。

系统的复杂度、分布程度和规模都在增加。指标和日志的巨大容量要求新一代基础设施能支持对监控数据的收集、传输和存储。一旦你收集了海量的监控数据，为了从监控数据中获得洞察力，你可能要依赖于大数据分析提供的能力。这种洞察力将不再仅仅是关于系统健康度和性能的分析，而是关于业务和客户分析。

7.10　更多阅读材料

《Effective Monitoring and Alerting》[Ligus 13] 深入研究了本章讨论过的很多话题的细节。

Ganglia 是我们提到的一种监控工具，是《Monitoring with Ganglia》[Massie 12] 的主题。

《Building Microservices: Designing Fine-Grained Systems》[Newman 15] 描述了微服务架构风格。

作为统一机制的日志的思想来自 LinkedIn 页面 [Kreps 13]。

对 Google 搜索延迟响应影响的研究来自 Google 研究的博客 [Brutlag 09]。

Damon Edwards 提供了 DevOps 监控过程的类型 [Edwards 14]。

一如既往，我们依赖维基百科上的描述：

- 真实用户监控：http://en.wikipedia.org/wiki/Real_user_monitoring
- 综合监控：http://en.wikipedia.org/wiki/Synthetic_monitoring
- Apdex：http://en.wikipedia.org/wiki/Apdex

我们提到的工具可以在下面的链接中找到：

- RRDtool：http://oss.oetiker.ch/rrdtool/
- 应用响应度量：https://collaboration.opengroup.org/tech/management/arm/
- Logstash：http://logstash.net/
- Nagios：http://www.nagios.org/
- Sensu：http://sensuapp.org/
- Icinga：https://www.icinga.org/
- Graylog：http://graylog2.org/
- Splunk：http://www.splunk.com/
- CloudWatch：http://aws.amazon.com/cloudwatch/
- Kafka：http://kafka.apache.org
- Storm：http://storm.incubator.apache.org/
- Flume：http://flume.apache.org/
- S4：http://incubator.apache.org/s4/

第 8 章　安全与安全审计

是人就会犯错；
当你有了 root 密码后，你就真的可以把它搞糟了。

——匿名

在 DevOps 环境中讨论安全的最初反应是，假设安全实践是非敏捷的，并且安全实践事实上可以阻碍改进从代码提交和验收到正常生产环境中的时间。我们认为这种反应是完全落后的。讨论采用 DevOps 实践而不考虑安全就会让安全团队成为 DevOps 实践的批评者，并注定导致在很多企业中引入 DevOps 实践的失败。在第 12 章的案例研究中，我们将看到这样一个方法——它提倡把安全团队集成到采用 DevOps 实践的过程中。在讨论安全时，需要考虑的其他一些 DevOps 活动是：

- 安全审计。当安全审计迫在眉睫时，Dev 和 Ops 之间的协作就变得非常重要了。
- 保证部署流水线的安全。部署流水线本身对恶意攻击者来说就是一个有吸引力的目标。
- 微服务架构。采用微服务架构引入了新的安全挑战。

对于金融组织、处理健康记录或者其他私人信息的组织或者对金融交易有内部控制的组织（几乎所有组织），安全审计是无可争辩的事实。安全审计检查组织的全部方面，从它的策略到它的实现。DevOps 中的一个流行语是“基础设施即代码”，也就是说把脚本和 DevOps 过程规格说明视为代码，使用与代码相同的质量控制实践。安全策略、监管规则和配置可以自然地内置在基础设施代码和自动化中以便易于审计。自动化也可以帮助生成审计输出和检测非合规性。在讨论安全的特定方面时，我们将回到这个主题。

安全审计验证组织已经满足需求。这些需求为组织的策略和实践提供保护，在 IT 以及非 IT 方面。这意味着，无论开发组遵循什么实践，这些实践都必须遵守这些需求。

本章首先讨论安全的含义，然后讨论对安全的威胁。那时，我们展示组织的角色以判断谁应该为解决那些威胁做出什么工作。我们讨论了用以解决威胁的技术，并且解释了在安全审计中会发生的情况。随后，我们从应用开发的视角分析了安全问题。最后，我们以讨论部署流水线的安全来结束本章。

8.1　安全是什么

很容易通过简写 CIA 来记住安全。CIA 代表了机密性（Confidentiality）、完整性

（Integrity）和可用性（Availability）。机密性意味着没有未经授权的人可以访问信息；完整性意味着没有未经授权的人可以修改信息；可用性意味着经过授权的人员可以访问信息。

授权是机密性、完整性和可用性定义中的一个核心部分。最安全的系统没有使用者，但也没有任何用处。让经过授权的人获得访问同时拒绝未经授权的人访问是一件困难的事情。授权包括回答下面问题的 2 个元素：谁正在试图访问或者修改信息？这些人是否拥有他们请求操作的执行权限？有很多技术支持以上的这些元素。在 DevOps 环境下，上面提到的信息和操作指的是应用以及同等重要的部署流水线（例如源代码、构建服务器和特定的流水线操作）。而且，这些技术已经纳入了广泛可用的软件包。来自安全专家的强烈建议之一是“不要从烟草开始卷制香烟”（此处指的是不要自造安全方案，而应该采用一些成熟的技术，比如加密算法等——译者注）。不知不觉引入的错误可能是难以察觉的。你可能检测不到这些隐蔽的错误，但它们却给攻击者提供了一个途径来破坏你的系统。

安全专家做出的一个假设是，任何一个保护方法都是可以被绕过的。因此，安全专家提倡“深度防御”，它意味着攻击者必须绕过大量不同的防御手段才能够破坏你的系统。考虑在一个典型的间谍故事中，报纸上的敏感信息是如何保护的：在一个隔离的庄园周围建设了围墙，在围墙内有警犬，房子上有安全系统，房间内有被锁上的房门，最后房间内的保险箱里放着敏感的报纸。这个模拟说明，你需要的安全等级是成本 / 收效分析的结果。你的组织希望花费在安全上的费用依赖于如果你的系统被破坏了将造成多大的损失。同样，攻击者希望花费在破坏你系统上所付出的成本依赖于他能够从这次破坏中获得的效益的多少。

深度防御也提出了这样一个观点：你的系统可能被破坏。在这样的情况下，你需要一套机制来检测发生了什么以及让你拥有恢复的能力。因此，与 CIA 相关联的也是不可否认性的特性：个体不能否认他在你系统内的数据上执行过的操作。这一点对审计来说是重要的。

另一个概念是攻击的生命周期。攻击可能被阻止，在攻击时它们被检测到，或者在攻击成功后它们被检测到。在安全领域中，最小化安全风险的措施称为“安全控制”。控制可以是预防性的、检测性的或者纠正性的，这依赖于它们应用在攻击的生命周期内。

最后，通过分析谁对控制的实施负责，我们也可以从另一方面对控制进行分类。技术控制（例如加密）是在应用或者基础设施内实施的。组织级的控制，例如在厂商发布了安全补丁的 24 小时内要将这些补丁应用到系统上，是通过由组织开发的策略和程序来实施的。这两种类型的控制可以是互补的，一个动作可能同时需要这两种类型的控制。例如，不仅应该将安全补丁迅速地安装（组织级的），而且系统也应该可以响应对于补丁级别的查询（技术的）。

8.2　威胁

在设计系统或者子系统时，你可以采取这样一个视角，那就是站在攻击者的角度来看。微软引入了简写为 STRIDE 的威胁模型。请参考见图 8-1。

STRIDE 代表：

- 身份欺骗（spoofing identity）。身份欺骗的一个例子是非法访问，然后使用另一个用户的认证信息，例如用户名和密码。
- 篡改数据（tampering with data）。篡改数据是指对数据的恶意修改。
- 否认性（repudiation）。否认性威胁是指，用户否认执行了一个动作，而且没有其他的当事人能够提供相反的证明。
- 信息泄露（information disclosure）。信息泄露威胁是指将信息暴露给不应该访问它的个体。
- 拒绝服务（denial of service）。拒绝服务（Denial of Service，DoS）攻击的目标是对有效用户的服务可用性。例如，通过使 Web 服务器暂时不可用或者无法使用。
- 权限提升（elevation of privilege）。在这种类型的威胁中，非特权用户获得了特权访问，因此他有足够的权限来破坏或者摧毁整个系统。

身份欺骗
篡改数据
否认性
信息泄露
拒绝服务
权限提升

图 8-1　STRIDE 模型

请注意这些威胁与 CIA 定义有关。身份欺骗绕过了认证；篡改数据破坏了数据的完整性；否认性是对在发生了入侵事件或者试图违背规则以后发生了什么的明确表述；信息泄露是机密性的反面；拒绝服务违背了可用性；权限提升是允许违反任何 CIA 特性的技术。

在审计过程中，你应该做好准备来证明系统内的控制如何与其他组织和平台的控制联合起来以减少所有这些威胁成功可能性。

这里，我们也应该提到内部人——内部人造成了威胁的一类来源。软件工程研究所（Software Engineering Institute，SEI）把内部人定义为“当前或者前雇员、承包商或者业务伙伴，他们拥有或者曾经拥有对组织的网络、系统或者数据的被授权的访问权限”。来自内部人的攻击是指，内部人滥用了访问权限来有意地违背一个 CIA 特性。

Verizon 报告，大约 15% 的数据入侵来自内部人。因此，内部攻击是有重要影响的。在进行组织的安全分析时，应该考虑这些内部攻击。

Verizon 和软件工程研究所把攻击的动因宽泛地描述为：

- 经济利益的。受经济利益驱动的攻击包括盗窃金钱或者可以售出的物品。有些市场专门用于交易通过攻击盗取的物品，例如信用卡数据。追踪这些市场可以让你了解一些已发生的入侵情况。
- 知识产权。许多攻击尝试获得知识产权，例如工商企业的商业机密或者政府组织

的机密情报。

- 破坏。这类攻击包括拒绝服务攻击和修改面向用户的信息（例如网站），也包括由心怀不满的雇员做出的对敏感数据的彻底破坏等。

最后，重要的一点是，很多问题是由与安全相关的错误导致的，而不是有意的攻击，正如在本章所引用的谚语“是人就会犯错；当你有了 root 密码后，你就真的可以把它搞糟了”。

8.3　需要保护的资源

在安全的元素中，C 和 I 与“信息”有关。信息是需要保护的关键资源之一。信息可以是静止的、在使用中的或者在传输中的。这包括与 DevOps 活动相关的信息，例如源代码、测试数据、日志、更新以及部署版本到生产环境的人。

- 静止的信息保存在持久存储上。这些信息的访问方式包括：通过由操作员控制的软件系统来访问；通过实际占有这种持久存储来访问。前者的一个例子是，一个合法用户登录并且接收允许他访问特定数据的凭证。被访问的软件理解这些凭证，并且知道如何获取、显示和修改数据。后者的一个例子是，在你的笔记本电脑上存储了敏感数据的副本，而你的电脑却被人从汽车的后备箱里盗走了。在 DevOps 环境下，除了保护与应用相关的持久性数据以外，你也应该考虑在你的纯文本日志行中记录的信息是否是敏感的；你的测试数据（它可能是敏感的生产数据库的一个早期快照）是否被有效地保护了；对源代码执行的安全措施是否是足够的。我们需要考虑通过加密与接下来的解密来保护所有数据以及因加密和解密导致的性能成本之间的权衡。正如我们将要看到的，微服务具有的小规模特性使得它更容易执行特定服务的安全策略。
- 使用中的信息是指正在被信息系统使用的数据。它可以显示给用户，它可能因性能或可靠性原因存储在缓存中，它也可能因性能或可靠性原因存储在虚拟机中。这些信息当前存在于信息系统中，如果用户拥有该信息系统一部分的访问权限，那么他就可以访问这些数据。在 DevOps 环境中，很多人提倡缩短服务器的使用周期（作为使用不断重生的服务器或者短暂的服务器的一个结果，在这些服务器上的变更意味着把一组旧的虚拟机替换为一组新的虚拟机），这样做的目的是提高可靠性并防止服务器漂移。这种实践也增加了安全性，它体现在销毁了随时间在服务器上累积的敏感信息。使用中的信息也可以内部加密，并且仅仅在显示时解密。这可能导致缓存代理的有效性更低，并且使用起来更加麻烦。
- 传输中的信息正在从一个地方迁移到另一个地方。如果这种迁移是通过网络进行的，那么通过访问网络就能获取到这些信息。网络访问可以发生在其中一个传输端点（Endpoint）上，或者发生在一个中间点（Intermediate Point）上。例如，监

控网络流量或者使用防火墙来过滤特定信息既是合乎情理的，也依赖于能在传输信息的中间点上进行网络访问。如果数据是从同一台物理主机的一个虚拟机移动到另一个虚拟机上，或者数据是通过套接字（socket）从一个进程移动到另一个进程，那么这些数据要么是被端点访问要么是被传输机制访问。有很多技术可以用于在传输中加密数据并认证两个传输端点。但它在证书和密钥管理方面增加了复杂性，同时也因为认证 / 授权和加密 / 解密带来了额外的性能损失。关于这部分内容，我们将在后面介绍微服务架构时讨论。

计算资源也是需要保护的。它是 CIA 中的 A（Availability）。经过授权的用户应该能够访问他们需要的资源。同样，这也包括了 DevOps 资源，例如构建服务器。许多原因会导致资源对已授权用户是不可用的：

- 最简单的理由是，你忘记了你的密码或者因一时乱放而找不到你的密钥。我们都会使用多个系统。这些系统需要不同的密钥。并且它们对密码的长度和组合有不同的要求。而且，我们也被告诫说，不要重用密码，对不同的目的使用特定的密钥。在这样的情况下，忘记一个特定的密码或者错误地管理了密钥就不是一件稀奇的事情。系统应该提供一种方式来恢复或者使一个密码或者密钥无效。
- 你的密码、密钥或者证书也可能被恶意地重置或者窃取。如果攻击者成功窃取了你的证书，那么这个攻击者就可以冒充你，然后做一些恶意的行为。在发生变更的情况下，系统应该提供一个方法来验证和告警用户，也应该快速地提供修正方法。管理、监控和替换被窃取的证书是复杂的过程，这些过程耗时巨大，也常常发生在停机时间内。
- 你尝试访问的系统可能正在遭受拒绝服务攻击。通过发送精心编排的请求序列，拒绝服务攻击使系统为了响应这些请求而消耗了资源。对于授权用户来说，这些资源就变得不可用了。安装基于 IP 地址的网关过滤器是成功阻止拒绝服务攻击的一种方法。应用程序编程接口（Application Programming Interface，API）密钥是另一种用来限制访问频次和用户恶意使用的流行方法，不管这些问题是有意还是无意造成的。

不是每一个入侵者都破坏 CIA 特性之一。假设一个非授权的学生正在使用你的系统做作业。这个用户对系统的需求是比较低的，因此他的使用不会导致破坏可用性。这个学生也没有访问或者修改信息，因此他的使用也没有破坏机密性和完整性。然而，这种使用仍然是非法的，也意味着你的计算资源没有得到较好的保护。

关于资源保护的最后关注点来自于云的特殊性质，特别是在 DevOps 环境下。开发人员能够轻易地创建虚拟机镜像（例如为了使用重度烧制的部署方法）和以这些镜像运行虚拟机实例。很容易失去对镜像和实例的追踪，特别是在没有为使用付费的情况下，例如在私有云中。我们使用*虚拟机镜像泛滥*一词来描述虚拟机镜像激增到管理系统失去对它们追踪。这些镜像占有了大量的存储空间，对哪个镜像用于哪些实例失去追踪也带来

了额外的安全问题。例如，在正常的打补丁过程中遗漏了镜像或者实例，那么与打过补丁的系统相比，这些镜像和实例在攻击中更加脆弱。近期，由于虚拟机泛滥导致成功入侵例子来自于 BrowserStack：一个未被使用的虚拟机没有打补丁，因此它被 Shellshock 攻破。然后一些包含在这个未被使用的虚拟机中的凭证被用来攻破活跃的虚拟机。

失去追踪的实例也可能导致额外的成本。我们应该建立机制来追踪虚拟机镜像和实例。例如，Netflix 的 Simian Army 工具套件包含了 Janitor Monkey 工具，该工具查找未被使用的资源，然后删除它们。Security/Conformity Monkey 工具也查找安全违背或者安全漏洞。一些操作系统厂商提供了特定的工具来帮助你检查虚拟机或者物理机上是否应用了最新的补丁，例如 Red Hat 的 Spacewalk。如前所述，对虚拟机强制一个较短的生命周期，意味着在设定的时间后终止这个虚拟机并且替换它，即使这个虚拟机是正常工作的。对虚拟机强制一个较短的生命周期是一个用来防止虚拟机泛滥的技术。它不仅仅防止了配置漂移和鼓励了更好的容错机制，而且它也提高了安全性（减少了攻击模式并有助于追踪以及删除敏感信息踪迹）。

8.4　安全角色和活动

我们识别与安全相关的 4 个不同的角色。当我们讨论为了实现一个安全环境所需要的各种活动时，我们将引用这些角色。执行这些角色的人可能属于同一个组织或者属于不同的组织。

1）*安全架构师*。安全架构师负责设计组织的网络来实现网络的安全。安全架构师也负责监督网络的实施。

2）*解决方案架构师*。解决方案架构师负责设计系统以支持组织的业务功能。开发人员实施这些设计。

3）*IT 人员*。组织的 IT 人员负责监控和追踪与可能的安全攻击有关的任何事件。IT 人员也负责实施由安全架构师设计的架构。

4）*平台提供者*。平台提供者负责保障组织使用的云计算平台边界的安全、保证平台上的客户之间的隔离以及保证客户能够在他们需要时获得足够的资源。平台提供者也提供了安全架构师使用的服务。

你可以看到这些角色之间的一些依赖关系。平台提供者提供了一个根基。对于很多组织来说，这个根基可能对很多团队和业务单元是无法使用的。安全架构师为整个组织使用由平台提供的服务而设计安全。IT 人员实施和监控由安全架构师提供的安全设计。解决方案架构师在安全架构和平台上设计系统。

在 DevOps 环境下，这些角色执行的活动可以内置在工具中。一如往常，这些工具的使用需要记录在日志中并保留以便由审计人员进行进一步的检查。在通过基础设施即代码来实施与网络和基础设施相关的安全方面，对于开发人员的角色，也存在着一个争

论：开发人员把一些 IT 人员的职责纳入其中。随着安全厂商开始提供应用程序编程接口和支持更多的自动化，也随着软件定义网络（Software-Defined Network，SDN）的出现，这成为一种现实。最重要的问题可能不是 Dev 或者 Ops 在实施一个特定的安全设计层是否扮演了角色，而是这些安全设计层是否被实施、是否被追踪的、是否是自动化的以及是否是可审计的。

安全社区一直活跃在识别和发布组织应该采用的控制类型上，这些控制类型用于管理信息安全和隐私的风险。国家标准与技术研究所（National Institute of Standards and Technology，NIST 800-53）和国际标准化组织 / 国际电工委员会（International Organization for Standardization/International Electrotechnical Commission，ISO/IEC 27001）发布了两个不同的且被广泛采用的列表。这两个组织互相协作，它们的列表交叉引用了对方，因此这两个列表是非常相似的。我们基于 NIST 800-53 开始我们的讨论，因为在因特网上它可以被自由使用。

按照功能相关的类别来进行控制是一种方法。NIST 800-53 的类别是：访问控制、意识和培训、审计和责任制、安全评估和授权、配置管理、应急计划、身份识别和认证、事故响应、维护、介质保护、物理和环境保护、规划、人员安全、风险评估、系统和服务获取、系统和通信保护、系统和信息完整性以及项目管理。

正如你所看到的，这些类别包含了广泛的支持活动，这些活动由组织内的很多参与者完成。枚举所有这些控制将花费超过 200 页的文档。这些控制不同于由组织执行的活动和由信息系统执行的活动。由组织执行的活动的例子包括：组织建立和管理加密密钥，这些密钥对于在信息系统内部署加密算法来说是必需的；由信息系统执行的活动的例子包括：按照联邦法律、行政命令、指示、政策、监管和标准的要求，信息系统实现组织定义的加密使用和每次使用所需要的加密算法类型。

这些控制主要是以结果来指定的，而不是以方法。例如，一种控制描述为“在建立 [选择（1 个或多个）：本地；远程；网络] 连接前，信息系统唯一地识别和认证 [指派：组织定义的特定的和种类的设备]”。没有讨论关于系统是如何识别和认证的，而只是说要存在这样的过程。你应该实施、测试和监控与这些控制的一致性。对于实施来说，这仅仅是通过手动或者自动方法来执行控制。你将依赖平台上现有的安全机制、服务或库来实现安全功能，例如授权、认证和加密。其中一些机制可以通过代码调用安全产品的应用程序编程接口来自动地实现。那么，这些代码就成为审计中的重要证据。对测试来说，这意味着将与安全相关的测试集成到持续部署流水线，这将导致在提交前在集成开发环境（IDE）中进行静态分析或者在构建和测试服务器上执行与安全相关的测试用例。安全测试的这种持续的和集成特性成为审计的另一种证据。最后，对于监控来说，可以通过实施对生产环境的安全监控和一致性检查来检测和修正任何违反安全的问题。因此，实施了控制的 DevOps 过程和工具在使用中不但没有破坏组织通过审计的能力，而且在很多案例中还达到了提高组织能力的效果。

从审计人员的视角来看，评估组织的安全控制的方法是从组织采用的策略开始。审计人员最初试图回答问题：组织的策略是否适合于满足系统或被评估系统的安全需求？由 NIST 800-53 枚举的组织级控制或指令提供了评估的起点。不同领域已经使用 NIST 800-53 作为起点，然后在其上增加了它们自己领域特定的需求。相对于 NIST 800-53 来说，这些领域特定的变种更为人们所熟悉。对于各自行业的从业者来说，以下名称都是熟悉的，例如健康信息的 HIPAA、信用卡信息的 PCI 以及电网的安全概要。然后，审计人员开始执行已选择的控制以实现特定策略并尝试回答下面的问题：选择的控制是否适合于实现该策略？它们实现了这些控制吗？这些控制是否按照预期发挥作用？同样，由 NIST 800-53 枚举的实施控制或它的指令提供了一个判断这些控制是否适合的起点。审计人员基于组织提供的证据来判断是否正确地实施了控制以及是否如预期运作。

我们关注技术控制——软件架构师至少对这些控制中的一部分有一些输入或者控制。技术控制有 3 种类别：

- **通道内的控制**。这些控制允许合法用户访问网络、认证用户以及授权用户访问信息或资源。这些控制支持并且也应该应用于涉及应用本身的活动中、涉及部署流水线以及其他操作中。例如，修改脚本应该包括认证和授权，并且也应该在版本控制系统中进行追踪。
- **通道外的控制**。这些控制的目的是阻止通过非批准的通道进行访问。例如，旁路攻击（side channel attack）利用计时、功耗方面的漏洞以及声波和电磁泄漏来获得有用的信息。信息可以是静止的、在使用中的或者在传输中的，并且在任何情况下，信息都应该是被保护的。不应该在正常的通道之外使用资源，并且应该评估旁路（side channel）。
- **审计**。系统内的各种活动都应该以记录的形式保留下来，例如资源的使用、数据的访问和数据的修改。审计控制的目的是保证创建和维护这样的记录。同样，在 DevOps 环境下，这意味着大量不同的事物：1）使用自动化的工具和基础设施即代码来记录安全测试结果；2）在 DevOps 流水线中集成安全测试；3）DevOps 流水线和其他操作本身的安全性。在安全审计中，所有这些都提供了有用的证据。

8.5　身份管理

除非应用不加限制地对所有用户都是可用的，否则识别用户就是判断一次特定使用是否合法的先决条件。身份管理指的是创建、管理和删除身份的任务所需的所有任务。在用户账户的生命周期内，身份管理包括添加或者删除对特定系统的访问、重置丢失的密码 / 密钥以及执行周期性的密码 / 密钥修改。为了审计的目的，身份管理任务中的所有活动都应该记录在日志中——不仅是人触发的活动，而且也要包括通过工具或者脚本执行的活动。服务或者微服务之间的调用也需要认证。

身份管理与平台提供者和安全架构师的角色有关。平台提供者提供了管理平台的所有用户身份的方法；安全架构师供了管理一个组织的系统的所有用户身份的方法。考虑到组织的系统是在平台提供者提供的平台上运行的，那么组织的用户同时也是平台的用户。访问应用、开发活动和部署流水线可以使用相同的身份管理。当我们讨论授权时，我们再讨论这个概念。

有大量商业的身份管理工具存在。将身份管理控制分类在 NIST 800-53 的身份和认证类别中。身份管理工具以及认证工具都是由安全架构师控制并由 IT 人员来执行。

8.5.1　认证

认证控制的目的是验证你确实是你所说的那个你。也就是说，认证控制是防止欺骗攻击。这里的“你”也包括了调用其他服务的服务。这里，我们关注认证一个个体，然后再讨论服务之间的认证。

认证一个个体会因为大量的原因而变得复杂。

- “你”可能不是意味着真的你，它可能是指代表你的系统。
- “你”可能不能够被系统唯一地识别出来，相反，“你”可能是一个角色。
- 你的认证机制（例如密码或者证书）可能已经被泄露了。
- 你可能不再是一个雇员或者系统的授权用户了。

在安全领域中，你作为个体，有 3 种方法来认证你。这些是你知道的某些事物（例如，密码）、你拥有的事物（例如，智能卡）或者表征着是你的事物（例如，指纹）。有些系统要求你使用以上 3 种类别中的 2 种来认证。例如，你的自动取款机（ATM）卡有一个磁条（你拥有的事物），同时也要求你输入个人识别码（Personal Identification Number，PIN）（你知道的事物）。其他系统需要你知道多个事物。例如，你需要知道密码和对一个安全问题的答案。在系统要求你提交你所知道的事物前，它向你认证它自己。例如，在你输入密码前，系统通过展示一张你预先选择的图片来向你表明你确实是在正确的系统上操作。这种技术试图避免使你的密码泄露。基于证书的方法在某些方面更加安全，但是它需要更复杂的基础设施设置。

在本讨论的剩余部分，我们将详细说明认证控制的不同类型。

1. 与代表你的系统有关的控制

对硬件和软件的考虑是不同的，因此我们相应地把讨论进行了拆分。

（1）硬件

保证只有合法设备才能够连接到你的系统的强有力方法是，要求对设备进行预注册。这能够阻止中间人（man-in-the-middle）攻击。一种弱形式是，让你的系统在外部系统上留下状态（例如，cookie），这个状态用来标识以前曾经访问过你系统的系统。如果外部系统上没有这样的状态，那么将可能被询问额外的问题。

维护就是一种场景，其中，强有力方法变得重要。也就是说，你的系统有一个物理

组件（例如，自动取款机），通过使用一个专用计算机来执行维护。对这个专用计算机进行注册可以防止使用一个欺骗性的维护计算机来破坏你的系统。

（2）软件

我们曾经提到，你的系统可能通过一个平台来访问资源。你的系统有用户，同时你的系统是这个平台的一个用户。要求用户先登录到你的系统，然后再登录到这个平台，是不可取的。原因是：

- 用户拒绝对同一个系统进行多次登录。尽管你知道，事实上，有多个系统，但从用户的角度，这些多个系统应该作为一个系统出现。
- 被访问的平台资源可能是被多个用户共享的，把相同的密码暴露给多个用户违反了认证控制原则。要求每个用户在一个平台以及系统上有一个账户是一件麻烦的事情。例如，删除一个账户，要求你的系统与平台之间进行协作，而这一点是难以管理的。

存在两种基本的技术支持一个系统使用用户凭证来访问另一个系统。一种技术是单点登录（Single Sign-On，SSO），另一种技术是使用独立的、由系统管理的凭证。

- 单点登录依赖于提供服务的不同凭证。曾经开发了这种能力的多个不同版本，但是最有名的可能是 Kerberos。初次登录产生一个 Kerberos 票据授权票据。这个票据用于登录到其他接受 Kerberos 票据的系统。
- 系统管理的凭证意味着，你的系统维护了一组凭证，这些凭证用于访问平台或者其他系统。你的系统使用这些代表你用户的凭证来认证对平台的使用。你必须考虑的一个问题是，如何保护这些凭证免于非授权的访问。这个主题在 NIST 800-53 中称为“系统和通信保护”的一组独立的控制集中。证书也有一个截止日期，无法与外部系统通信的常见原因是使用了过期的证书。Netflix 的 Simian Army 工具中一个成员是安全猴子（Security Monkey），它们的职责是检查过期的证书。

2. 基于角色的认证

基于角色的认证（Role-Based Authentication，RBA）是一种基于角色而不是身份来分配辨别信息的技术。例如，你可能使用 root 密码以超级用户登录系统。超级用户是你承担的角色，它不是你的身份。基于角色的认证简化了一些问题，因为一旦识别了一个角色，那么与这个角色相关的访问权限就可以自动地分配。基于角色的认证的问题是缺少可追踪性。一个问题可以追踪到“超级用户”，但不能追踪到个体。在下一节讲到授权时，我们将讨论如何纠正这个问题。

3. 防止密码泄露的控制

密码被泄露的方式有如下几种：

- 通过各种形式的蛮力攻击，攻击者破解了一个个体的密码。控制指定最小的密码长度、密码生命周期以及对密码重用的限制。

- 通过社会工程（social engineering）方法，攻击者判断出了用户的密码。有关于对用户进行安全教育的控制，但是使用社会工程来判断密码的最臭名昭著的案例可能来自 Stuxnet 蠕虫。这个蠕虫利用了这样一个事实——有些系统密码是在被攻击软件中硬编码的（hard coded）。另外，也可以在因特网上寻找这些密码。一种防止这种类型攻击的控制是，要求在系统进入生产环境前，修改默认密码。
- 已授权的用户改变了角色或者离开了组织。我们把角色改变的讨论放到下一节讲述授权时。但是当雇员离开组织时，控制指定他在很短的时间内删除他们的账户权限。
- 你的系统被入侵了，因此导致密码泄露。控制指定密码必须使用已批准的加密软件以加密形式存储的。如果数据是以足够强的方式加密的，那么这些数据是难以被解密的。而且，提供加密和解密功能的软件也必须是被批准的，也就是说，该软件是经过授权的测试组织测试过的。

8.5.2　授权

一旦用户被识别出来，那么就有可能基于对该用户授权的权限来控制其对资源的访问。在 NIST 800-53 中最相关的控制是“AC-3，Access Enforcement（访问强制）”：

> 控制：按照适当的访问控制策略，信息系统强制要求必须要有已批准的授权才能对信息和系统资源进行逻辑访问。

和认证一样，可以将授权记录到日志中，不管资源是人工访问的，还是通过脚本或者工具访问的，同时还要记录谁应该对这个授权负责。

1. 控制对资源访问的技术

有两种基本技术用于控制对资源的访问：访问控制列表（Access Control List，ACL）和能力（capability）。

- 访问控制列表：访问控制列表是附加在一个资源上的列表，这个列表包括了用户或者角色以及对其允许的操作，而这种资源的一些例子包括文件系统或者数据库字段。当用户请求访问资源来执行一个特定操作时，系统检查这个列表以判断该用户或者角色是否有权限对该资源执行他所请求的操作。
- 能力：能力是令牌，该令牌授予对资源的特定权限。一个形象的比喻是钥匙和锁。能力是一把钥匙；资源维护锁。当请求访问资源时，该资源验证访问请求所携带的令牌包含了对令牌提供者来说足够的权限以至于能够被授权访问。

不管使用哪种技术来控制访问，都应该采用最小化权限（Least Possible Privilege）对用户或者角色授权以使他们能够执行其所需要的任务。

2. 基于角色的访问控制

我们在本章中曾经讨论了基于角色的认证，并且指出把权限分配给角色简化了管理

大量用户带来的挑战。现在我们来讨论如何基于角色来进行访问控制。就系统而言，每一个用户都应该拥有一个唯一的身份。但是，用户可能会改变角色，因此他就具有了不同的访问权限。例如，考虑 root 密码。假设一个运维被晋升到了一个不需要 root 访问的职位上。在这种情况下，有 3 种选择：1）让 root 密码维持原样，这将导致非授权的个体（已晋升的运维）继续知道 root 密码；2）改变 root 密码，这将导致所有其他运维都必须了解这个新密码；3）使用基于角色的访问控制（Role-Based Access Control，RBAC）。

基于角色的访问控制建立在个体与角色之间的映射关系上。一个角色被允许一些特定的访问权限，并且身份管理系统维护了用户与角色之间的映射关系。身份管理系统也维护了角色与权限之间的映射关系。那么，当用户角色发生变化时，用户与角色之间的映射关系也会被相应地改变，同时将对应于新角色的信息提供给授权系统。因此，我们那个假想的被晋升的运维人员将从运维角色中移除，并给他赋予新的角色。身份管理系统将提供对应的新权限给该个体，同时把 root 访问权限从该个体中删除。为了审计的目的，这种转变也要记录到日志中。

在大型组织中，基于角色的访问控制变得很复杂，因为它假设在整个组织中都是一种一致的角色定义。很多大型组织在组织的不同部分都有类似但不同的角色。以一种一致的方式来定义角色可能包含在组织内的某些地方把一个角色上的责任转移到另一个角色上。例如，假设组织的一个部分已经采用了持续部署实践，而另一个部分还没有采用持续部署实践。那么，你将把“谁可以授权部署到生产环境？”分配给哪个角色？

让我们通过一个部署流水线的例子来说明这个问题。在 Jenkins 这样一个流行的持续集成工具中，在对部署流水线提供授权方面，有一些可供选择的方法。你通常希望对流水线的不同部分实施不同的授权。例如，开发者可能不被授权触发某些特定类型的质量保证（Quality Assurance，QA）作业（job）或者部署到生产环境。你可以使用 Jenkins 角色策略插件（role strategy plug-in）来定义不同的角色，这些角色对流水线的不同部分拥有授权。然后，你就可以把作业和角色关联起来。此时，它是通过对作业名进行正则表达式（Regular Expression）匹配来完成的，这可能是非常复杂的以至于难以管理的，特别是当你拥有很多作业时。另一种替代方案是使用 Jenkins 矩阵授权插件（Matrix Authorization Plug-in）。在该插件中，你把所有作业组织到不同的文件夹中，然后你就可以在文件夹级别定义授权了，它是通过把文件夹映射到用户或者角色来实现的。

8.6　访问控制

身份管理控制的目的是防止欺骗、篡改、信息泄露以及权限提升（对于那些已经成功通过认证和授权通道的用户）。篡改和信息泄露仍然一种威胁，这些威胁来自于那些未通过认证和授权通道的人。在本节中，我们将讨论这个问题。在下一节中，我们将讨论 STRIDE 模型的剩余元素——不可否认性和拒绝服务。

我们首先讨论用于防止篡改和信息泄露的控制。在本章开头讲到的那个间谍的比喻，在这里也是相关的。阻止访问，然后，如果阻止访问失败，那么就设法让入侵者找到的信息是不可用的。

8.6.1　阻止访问

按照从外到内的工作方式，必须定义系统或者组织的软件系统的边界。也就是说，需要被保护的资源必须被清晰地识别出来。资源可能有不同的保护等级。例如，资源对未认证的用户来说是可读的（比如，一个对因特网开放的网站），或者资源对未认证的用户是不可读的（一个内部网站）。

1. 定义边界

组织的网络可以划分成子网（subnet），每个子网有自己的边界。每个子网代表了一组资源的集合，这些资源拥有相同的保护级别。在确定边界时，使用微服务架构提供了更高的灵活性。一旦边界定义下来，那么就可以对从边界外部到边界内部的通信以及从边界内部到边界外部的通信进行控制。来自因特网的访问与来自移动接入（mobile access）的访问是区别对待的，相应地，与来自内部的访问也是区别对待的。有防火墙、网关、路由器、防护装置、恶意代码分析和虚拟化系统或者加密的通道来保护每个网段。这个整体结构处在安全架构师的范围内。

子网外存在的工具（例如，部署工具）必须要获得授予的许可，这样它们才可以部署到子网内。这些许可可以继承自工具的调用者。一个称为隔离区（Demilitarized Zone，DMZ）的子网是对因特网访问开放的，但限制访问内部网络。面向外部的网站一般部署在这个子网内。使用这种类型的边界保护，外部的访问必须穿过防火墙或者网关，这些防火墙或者网关可以限制端口使用、维护列入黑名单的 IP 地址以及执行其他检查。如果攻击者想对数据或者资源获得非授权的访问，那么他必须首先通过在边界上的边界检查（perimeter checking）。

2. 隔离

隔离是与边界检查相关的技术。隔离是指逻辑上不同的功能是分离的，或者物理地，或者逻辑地。物理分离一直以来都有被使用——不要把你想保护的资源连接到因特网上，并且限制对这些资源的物理访问。在现代世界中，物理分离仅仅在极少的情况下是合适的，特别是在过程控制的环境下，但事实上，当访问系统的主要方式是通过因特网时，物理分离是不可行的。

隔离，在现代情境下，可以解释为分离。例如，你可以基于安全敏感度来分离计算功能。然后就可以建立一个边界，在数据或者进程要跨越边界时，要求出具凭证。敏感的个人数据可以从其他数据中分离出来。然后根据拥有的不同的凭证集合来授权访问个人信息的不同属性集合。

当资源共享时，例如两台虚拟机共享一台物理服务器，那么隔离是由在物理服务器上运行的系统软件来执行的。内存、硬盘和网络都可以在云环境中共享。内存隔离是通过使用虚拟内存技术来实现的；硬盘隔离是通过使用硬盘分区来执行的；网络使用隔离是通过使用的网络协议来完成的。

3. 加密

为了防止攻击者获取对数据的访问，不管数据是静止的还是在传输中的，都可以使用加密来保护数据。NIST 800-53 中的很多控制都描述了使用加密算法和软件。为了通过审计过程，算法和软件都必须证实是正确而强有力的。

在使用中的数据仅能通过破坏进程的隔离来接触到。为了性能和人为的原因，在使用中的数据通常是不加密的。加密数据和解密数据是耗时的——这是性能的原因。人难以阅读加密后的数据——这是人为的原因。

观察不同的技术是如何相互补充的。隔离识别边界，边界控制阻止非授权的访问，加密意味着一旦攻击者得到了数据，那么它也是无法解释的。

4. 其他要考虑的内容

在阻止访问方面要考虑的 3 个其他的相关方面是：退役数据（decommissioning data）、打补丁以及变更管理。

1）那些不再有用的数据可能仍然保留在系统上。但这些数据可能被攻击者利用。有些控制处理如何使数据无用。这可能包括从该数据存储的所有位置上移除，但是它也可能包括为了审计目的而保留该数据的副本。

2）系统有漏洞。厂商通过补丁来修复这些漏洞。必须将这些补丁应用到系统。控制指出补丁必须马上应用到系统，且系统也必须能够在被请求时上报它们的补丁级别。

3）重要的一点是，追踪系统上所有软件的版本和补丁级别。这不仅仅应该发生在进行根本原因分析（root cause analysis）时，而且也应该从安全的视角去考虑。知道系统中哪些潜在的漏洞已经打补丁了是一件重要的事情。为了能够证明系统可以抵御特定类型的攻击，不但软件的版本而且配置规格说明和部署规格说明的版本也是需要被追踪的重要内容。存在一组配置管理控制的集合对于 DevOps 来说也是特别重要的。不经过正式过程就直接改变配置不但是一个糟糕的实践，而且可能违反安全控制并且在安全审计时导致问题。

让我们用一个服务认证的例子来说明以上的全部内容。不同服务之间或者服务和浏览器之间的通信可能需要是被认证的而且是被加密的以防止被窃听或者中间人攻击。一种方式是使用 HTTPS 来加密流量。使用了 HTTPS，服务器端服务确实是如它所声明的身份，客户端服务也得到了强有力的保证。HTTPS 带来的一个问题是，组织需要管理 HTTPS 证书颁发和撤销过程，但这种管理工作的自动化却不是一件简单的事情。如果你有很多微服务和服务器，那么这将成为团队和部署流水线的严重负担。因为在微服务之

间的多次认证和加密导致反向代理（例如 Squid）失效，所以也带来了性能损失。你可能希望在边界上拥有较强的安全，这样你就可以选择在你的安全网络内不使用加密，而仅仅在你跨越边界通信时进行加密。

8.6.2　谁负责预防控制

我们已经识别了与预防非授权访问相关的 3 个角色：安全架构师、解决方案架构师和平台提供者。边境保护（border protection）是在边境内的系统所有者的职责。也就是说，平台提供者负责保护对平台资源的访问；安全架构师负责保护对组织资源的访问；解决方案架构师负责保护对特定系统的访问。防御性编程（defensive programming）和对进入消息（incoming message）缺乏信任是表征安全开发原则的两个设计实践。

相同的所有权的概念确定了其他类型的预防控制的责任。如果它关注你控制系统的一部分，那么你就负责保护该数据、保证可审计性以及保持补丁及时更新。

8.7　检测、审计和拒绝服务

我们已经讨论了一些预防措施，但是检测正在发生的攻击却涉及控制的不同集合。所有这些控制都包含监控。可以对资源进行监控以发现异常使用模式。可以对消息进行监控以发现大量不同的特征，范围从为了寻找一个开放端口而进行的端口扫描，到重复的登录尝试，再到页面获取请求的速度。所有这些控制都是由可用的工具来提供的，并且也是平台提供者、IT 人员和安全架构师的职责。解决方案架构师通常不会直接包含在这些控制中。

STRIDE 模型中的 R 代表否认性（Repudiation）。审计活动是重要的，既有业务原因（例如，“我没有订购那个东西”），也有取证原因（例如，“攻击者导致了什么样的破坏”）。一些需要记录的项包括：账户创建、账户的修改、访问控制机制的重写、特权函数的使用、安全属性的创建或删除、来自内部和外部源的连接，以及对软件或者配置做出的变更等。

一旦已经创建了审计追踪，那么必须对它进行保护。记录信息可能毫无帮助，因为攻击者可能修改该信息来掩盖踪迹。审计记录必须加密，而且要在独立于被审计的系统进行存储，同时保护对这些记录的访问。

不要把审计追踪和日志混为一谈。审计追踪会持续存在数月或者数年且具有法律地位，主要用于安全目的。日志持续存在的时间是以日（甚至更短）来计算的，它主要用于支持运维和开发的需求。

审计记录是我们提到的全体干系人的职责。干系人识别他们控制范围内可能发生的重要事件，然后他们负责决定这些事件以安全的方式添加到审计追踪中。

迄今我们还未讨论的 STRIDE 模型中的一个元素是 D——拒绝服务（denial of

service)。拒绝服务保护是平台提供者和安全架构师的责任。存在许多技术和工具用于限制拒绝服务攻击的影响。例如，边界控制设备可以过滤特定类型的包并限制被访问的端口以保护内部系统。限速（Rate-limiting）或者流量整形（Traffic-shaping）交换机也用于抵抗拒绝服务攻击。

8.8　开发

NIST 800-53 的一些控制说明了开发过程的各个方面。为了再一次提到基础设施即代码，必须开发脚本和 DevOps 工具的其他输入，而且它们也应该像应用代码开发一样受到相同的审查。另一方面，安全测试必须与部署流水线集成。开发人员必须证明他们已经明确满足了安全需求，并且已经执行了威胁建模和导出质量指标等过程。

安全的 5 个设计原则是：

1）为客户提供完成他们任务所需的最小权限。如果需要临时访问，那么在使用后要立即撤销它。

2）机制应该尽可能地小和简单。正如我们在第 5 章中提到的，带有狭窄接口（narrow interface）的小模块能够更快地测试。这个模块将更快的执行每个测试，因为它较小，并且因为接口是狭窄的，所以要测试的接口参数的数量也是比较少的。

3）对每个对象的每次访问都必须检查，不仅仅在正常使用中，而且还要在初始化、关闭和重启中。

4）把由多个用户共享而且被所有用户所依赖的机制的数量减少到最少。每个共享的机制都是一个潜在的信息路径。

5）利用故障安全（fail-safe）默认项。争论为什么一个特定的进程或者客户需要获得访问，而不是为什么那个进程或者客户不应该获得访问。

这些设计原则适用于应用设计和部署流水线本身。安全并不仅仅是通过良好设计就能达成的；它也需要良好的编码实践。有多个关于安全编码实践的列表，并且这些列表已经内置到了静态分析工具中。在部署流水线中，系统必须经过的一个安全闸门（Security Gate）是对编码实践的测试。另一个是对各种运行时攻击方法进行测试，例如跨站脚本(cross-site scripting)。

8.9　审计者

在这种背景下，审计者要寻找的东西是什么？答案是“以上所有内容”。审计者应该考虑所有事情，从代码和脚本的开发实践到采用哪种控制来防止哪种攻击等。

举个具体的例子，关于身份管理，考虑审计者应该提问什么问题——但注意，他们

会按照我们已经讨论的所有安全元素的相似顺序进行提问。首先，他们应该考虑组织在提供账户和撤销账户方面的策略。组织内的角色是否被清晰地识别出来？与正常账户或者特殊角色相关联的权限是什么？组织与它们的平台提供者如何交互？谁应该负责身份管理系统？

这些问题关系到安全架构师和平台提供者。关注点与策略有关，目标是保证在组织级别有合适的策略，而且组织与平台提供者之间的接口是良好定义的。

平台提供者可以获得独立的认证以证明他们符合一个或者多个特定领域的标准。如果他们已经获得了这种认证，那么平台提供者就不再需要参与这个审计过程了。

下一步，审计者涉及解决方案架构师，并且询问一些关于特定系统的相同问题。同样，它的目标是从策略的视角检查组织内的系统。审计者也会询问些关于开发过程的问题。开发人员是否有安全意识？在部署流水线中是否有安全测试？是否实施了评审？列举的设计考虑是否在早期就被使用而且是经过验证的？在开发脚本和使用 DevOps 工具方面是否也实施了相同的实践？以及更多类似的问题。

然后，审计者关心这些策略是如何被实现的？身份管理是如何实现的？密码是如何保存的？如何确认密码符合强度要求？平台的凭证是如何被组织管理的？如何对系统进行安全测试？以及更多类似的问题。让安全测试用例放在代码中并且与流水线集成，或者在良好测试的脚本中自动执行安全策略对于审计者来说都是很好的证据。

最后，审计者要求提供一些样本的证据。为我创建一个新账户、显示我获得的权限、给我显示一些记录以说明一旦一个员工离开组织需要花费多少时间来禁用这个账户，并且给我展示这是如何链接回你的变更管理系统。

在许多案例中，存在很多控制能够解决相同的问题。被审计的组织必须证明，他们对组织和技术控制的特殊组合满足要求。没有一个全能型的响应。如果一种控制在实现上是有缺陷的或者没有提供证据，则另一个控制可能满足被评估的需求。

8.10　应用设计考虑

对安全来说，云的使用和微服务架构带来了一些特殊的设计考虑。

- 对应用主机（也就是云上的虚拟机）来说，需要采取一些额外的安全考虑。我们使用 AWS 云作为一个例子。
  - 在初始注册和设置后，任何云范围的 AWS 管理账户（与 root 账户一样）都不应该继续使用。通过使用 AWS 身份和访问管理（Identity and Access Management，IAM），我们应该为不同目的设置不同的身份（用户或者角色），这些身份具有（对资源的）最小权限。
  - 不同用户之间不应该共享任何 EC2 密钥对。
  - 使用服务器端的加密来保证存储对象的安全，例如 AWS S3。

- 任何虚拟机都不应该访问因特网，除非这种访问是通过一个网关并且限制它只能访问所需要的端口。应该使用具有合适子网的虚拟专用网络。
- 使用 AWS CloudTrail 日志来监控和审计访问历史。
- 把日志从 EC2 实例上转移到外部处理和存储组件上。

- 应该能够将组件隔离并独立地部署而不影响其他组件。这是考虑到本书讨论过的安全和其他原因。
- 应该将组件编码为防御性的并不信任它们的调用者。这不但是考虑到安全的原因，也是为了可靠性的原因。
- 应该给组件提供适合于其所在执行环境的配置，这些配置有些情况下是通过动态查询一个外部服务获得的。应该将这些组件编码为在初始化时测试所有的配置，然后当调用其他组件或者资源时使用这些配置。
- 配置应该保存在版本受控的持久存储中，这样：
  - 为了审计的目的，可以追踪设置和使用这些配置，并且万一组件发生故障了，配置的值也是可用的。
- 服务之间的调用应该是被认证的，应该考虑认证带来的性能损耗。
- 与外部世界的通信应该是加密的，而且内部服务之间的通信也应该考虑加密。需要考虑的内容包括：数据敏感性、边界安全和性能开销。
- 为了减小攻击模式（attack profile），应当使用一个良好补丁的基础镜像来为不同的微服务创建其他定制化的镜像。可以考虑设置一个独立的团队来负责创建这个安全的基础镜像，并且仅仅允许每个开发团队对他们的服务进行有限的定制化。

8.11　部署流水线设计考虑

部署流水线本身也可以托管在云上，特别是测试环境，它可以从云弹性、可重复的纯净设置和不同环境之间的较好一致性上获得好处。云托管的安全考虑与我们在前面部分讨论过的完全一样。其他的特殊安全考虑可能包括：

- 在大部分时间内锁定你的流水线环境，并且追踪对流水线的所有变更。
- 集成贯穿在流水线中的持续安全测试，这包括集成开发环境 / 提交前分析、构建和集成服务器以及端到端的测试环境。
- 在生产环境中集成安全监控。我们曾经提到过的一个例子是 Netflix 的一致性和安全猴子（Conformity and Security Monkey）。
- 在每次各自的测试完成以后，或者至少是周期性的，把测试环境拆解。这不仅仅能够减少长时间运行的实例的安全风险，而且也提供了一个在再次启动前更新安全补丁的机会。
- 通过基础设施即代码尽可能地自动化流水线，促进代码复用，特别是为了在各种

测试和生产环境之间提高环境的一致性。这包括通过安全厂商的应用程序编程接口来自动化安全运维。

- 考虑加密敏感日志和测试数据，不管是它们是静止的还是在传输中的。
- 严禁未通过流水线（以及它的变更追踪）对任何环境进行直接变更。尽可能尝试使用监控数据、转移的日志和复制的环境进行诊断，而不是直接访问和修改环境。
- 测试基础设施代码（不仅仅是应用代码）的安全漏洞。
- 能够通过自动化生成周期性的一致性和审计输出。

8.12　小结

对于任何希望认证为合理处理了敏感数据的组织来说，重要的一点是证明你的系统是安全的。这些证明通常提交给一个审计组织，这个组织是被信任来执行全面审计的。

特定领域的安全需求已经被汇编整理成了系统需要实现的一些控制的列表。这些控制的目的是应对各种威胁。其中一种威胁模型是 STRIDE——欺骗（Spoofing）、篡改（Tampering）、否认性（Repudiation）、信息泄露（Information disclosure）、拒绝服务（Denial of service）和权限提升（Elevation of privilege）。一种常用的控制的列表是由 NIST 发布的。这个列表涵盖了组织控制和技术控制。技术控制可以分类为：处理身份管理的、处理访问控制的、处理攻击检测的、处理维护审计追踪的、处理开发过程的以及处理拒绝服务的。

这些类别中的每一类都生成大量的安全需求。当组织被审计时，将这些需求聚合起来。组织必须知道他们的安全需求是什么，必须提供证明来说明实现了满足需求的控制，以及必须提供证明来说明这种实现是正确的。

传统的 Devs 和 Ops 也分别形成了安全关注点的两个不同领域。开发者关注应用安全设计，而运维关注基础设施和运维环境安全。应用安全也依赖于运维安全。通过基础设施即代码、开发人员驱动的自动化和 DevOps 工具，DevOps 正在把一些运维安全责任转移给开发人员和工具。这要求开发人员和应用设计人员要更加清楚运维环境的安全关注点。

与其他任何横切关注点一样，从一开始就应该考虑安全验证和确认，并且通过 DevOps 流水线自动化地在不同阶段执行。在单元、集成和系统级别上执行合适的和半自动化的安全分析和测试不是一件简单的事情。安全分析经常是一件专家驱动的人力密集型的活动。现在，安全分析扩展到了部署时间，也要求在持续交付和部署流水线上实现完全的自动化。

DevOps 中的安全不仅是关于应用和运维安全的，而且也是关于流水线本身的安全的，例如构建 / 测试服务器安全、微服务组件安全、环境安全和动态配置期间的安全。视基础设施为代码提供了保证 DevOps 过程安全的理念。

8.13　更多阅读材料

为了获得对架构级安全关注点的一些理解，请参阅 [Bass 13] 的第 9 章。

关于云特定的安全问题，你可以在下面这个链接上找到许多模式的目录：http://www.opensecurityarchitecture.org/cms/library/patternlandscape/251-pattern-cloud-computing。

你可以在下面这个链接上找到关于 STRIDE 威胁模型的更多信息：http://msdn.microsoft.com/en/library/ee823878(v=cs.20).aspx。

关于减缓内部人攻击的方法，软件工程研究所（Software Engineering Institute，SEI）有一个技术报告，请参见 [SEI 12]。

NIST 800-53 是美国联邦信息系统的安全控制目录。你可以在 [NIST 13] 中找到它以及一些相关的出版物。

关于对当前某些安全攻击的有用分析，你可以参考维基百科的条目（http://en.wikipedia.org/wiki/Stuxnet）和 Verizon 的报告（http://www.verizonenterprise.com/DBIR/2013/download.xml）。

关于微服务特定的安全考虑，请参见 [Newman 15] 中关于安全的章节。

通过搜索“安全和 DevOps”，你可以找到讨论 DevOps 与安全之间关系的大量博客条目。

维基百科讨论了安全控制和它们的类型，地址是：http://en.wikipedia.org/wiki/Security_controls。

关于 BrowserStack 攻击的讨论，可以在 [ITSecurity 14] 中找到。

安全猴子在 http://techblog.netflix.com/2014/06/announcing-security-monkey-aws-security.html 中有描述。

第 9 章　其他非功能需求

我小时候，家里的菜单就两个选择：吃或者不吃。

——Buddy Hackett

9.1　概述

在本书的第二部分中，我们讨论了持续部署流水线的主要功能，例如构建、测试和部署。还有一些其他的 DevOps 操作，例如错误检测、诊断和恢复，它们引用了类似过程的流水线。在本章中，我们使用“DevOps 流水线”来表示 DevOps 的全部方面。

如果你是软件架构师，你可能知道“非功能需求”这个词，它用来描述软件的质量关注点，而不是关注软件的基本功能和正确性。对 DevOps 来说，非功能需求对应于如下类似的问题：在流水线中这些功能执行得怎么样？在需要时你可以精确地重复你的 DevOps 操作吗？从业务概念提出到将它最终发布耗时多少？在流水线中，不同的工具是如何互操作的？在第三部分中，我们一开始讨论了一些主要的关注点，例如监控和安全。在本章中，我们将涉及更多的关注点。如表 9-1 所示，在该表中，你可以找到我们将要讨论的非功能需求的列表和它们的主要质量关注点。

表 9-1　DevOps 流水线的非功能需求和质量关注点

非功能需求	质量关注点
可重复性	重复相同操作的可能性的程度
性能	执行 DevOps 操作所需要的时间和资源
可靠性	在一定时间周期内，DevOps 流水线及其内部各个软件保持服务状态的程度
可恢复性	失败的 DevOps 操作可以恢复到期望状态的程度，而仅对它所操作的应用造成极小的影响
互操作性	在特定环境下，不同 DevOps 工具通过接口有效地交换信息的程度
可测试性	通过测试，DevOps 运维软件能够很容易地展示其错误
可修改性	修改 DevOps 软件、过程或者应用运维环境所需要的工作量

我们主要讨论 DevOps 流水线本身的非功能需求，而不是流水线产生和操作的应用。在流水线与应用之间存在着某些较强的关联。例如，一次升级操作的性能和可恢复性可能对被升级应用的性能和可恢复性产生极大的影响——但本章不会对这种关联进行研究。我们将从两个不同方面考虑 DevOps 流水线的非功能需求问题：产品和过程。

首先，DevOps 流水线本身就是一件软件产品，这个产品的最终用户是开发和运维。与其他任何软件一样，DevOps 流水线的设计可以由良好的软件架构实践来控制，并且带

有早期的和明确的对来自干系人质量的关注。正如我们在前面所倡导的那样，我们认为把运维作为首要干系人是重要的，因为这样才能够揭示出 DevOps 流水线的更多功能和非功能需求。

其次，DevOps 流水线具有过程的特性。相对于产品质量和性能，这里提到的一些非功能需求更多与过程质量和性能相关联。我们可以在两个不同的层面来改进这种面向过程的系统。在一个层面上，DevOps 过程可以看起来像一个人力密集型的过程，这一点与软件开发过程类似。我们可以把在改进软件开发过程中学习到的经验应用到 DevOps 过程中，包括敏捷、生命周期模型、质量控制和成熟度模型。事实上，DevOps 运动无疑是源于把 Dev 工具和实践应用在 Ops 领域中的敏捷尝试。在另一个层面上，DevOps 过程可能适合于工作流系统或者业务过程管理系统，在这个系统中，预定义的 DevOps 工作流运行在一个工作流引擎中。工作流的质量关注点可以通过以下几种方式来解决：断言不同任务的前置和后置条件、更好的资源分配、异常处理以及管理长时间运行的事务。在第 14 章中，我们将更细节地讨论把运维当作过程的实践。

接下来，我们将讨论表 9-1 中列出的每个非功能需求。与其他软件一样，DevOps 过程的设计也涉及相关质量关注点之间的权衡。

9.2　可重复性

可重复性是指过程可以使用在不同应用或者分支上重复的程度。可重复性可以通过计数过程失败和成功的次数来测量。如果一个以前成功执行的过程现在失败了，那么这个失败就意味着在不同的环境下该过程是不可重复的。

有些过程是由个别团队定义的，例如部署微服务到生产环境的过程，而其他过程则是跨团队完成的，例如在微服务间分配功能。还有其他的过程可能是跨组织执行的，例如使用某些特定的工具。推出 DevOps 过程的一部分关注确定哪些过程是团队内的、哪些过程是团队之间的、哪些过程是涉及整个组织的。请参考第 10 章以获得该主题的更多讨论内容。实现可重复性的两个关键活动是：定义和执行过程；对所有工件维护版本控制。接下来，我们将讨论这些活动。

测量可重复性取决于能够识别两次运维的执行是执行同一个过程。完成这种识别活动的一种方法是，检查过程踪迹以保证它们以相同的顺序执行了相同的步骤。换句话说，我们把可重复性和可追踪性等同看待。如果不能识别过程的步骤，那么知道一个步骤的结果是否是该步骤的前一个执行的重复是不可能的。

9.2.1　在恰当的层级上定义和执行过程

软件开发和 IT 运维一直都是以人为中心的，包括了一些创新的解决问题的活动。即

使在 DevOps 领域正朝着完全自动化发展的趋势下，仍然有一些人力密集型的活动，例如发布计划和控制、提取复杂的监控规则和问题诊断等。这意味着要在以下两个方面做出权衡：定义和执行重复的动作以提高质量，提供自由空间以激发期望的创新活动。

过程提供了指导，很可能，过程的开发是一个理性过程的结果，在这个理性过程中，考虑了各种权衡。然而，过程的严格执行减少 Dev 和 Ops 的灵活性。过程执行既是自动化的问题，也是社会过程的问题。社会过程，例如戴一顶写着“我破坏了构建”的帽子，也教育和鼓励团队成员遵从一些特定的过程。

我们分别从开发和运维角度看看这个问题。

现代开发人员使用一组功能丰富的不同工具来完成任务，这些工具包括代码编辑器、编译器、调试工具、静态分析工具、测试工具和源代码版本控制系统。集成开发环境（Integrated Development Environment，IDE）可能集成了一部分这样的工具。对于定义和执行过程或者提供自由空间，每个工具都有自己的决定。然而，很少对使用这些工具的开发人员活动之间的过程流进行监管。例如，开发人员可能选择在提交大块代码到存储库（Repository）前仅仅做最小测试。这些大块代码不但极有可能导致构建失败、集成测试失败、暂时中断整个团队的持续交付流水线，而且调试和合并工作也可能会受到巨大影响。

为了获得可重复性，过程必须执行选定的实践。选择执行哪种实践是权衡的一部分，这种权衡深入 DevOps 过程的设计中。例如，团队可能强制定义提交前 / 推送钩子或者测试，以使在将代码集成到重要的分支前可以使用这种钩子或者测试来检查代码。提交前测试可能包含从代码风格检查到执行一组测试用例等各种功能。然而，如果这些检查所花费的时间过长且开发人员必须等待它们完成，那么开发人员的生产力可能会因此降低。我们必须在合适的执行测试级别上做出权衡——增加的可靠性（通过大量的本地测试获得）和风险（构建失败和影响流水线的风险）。

我们的目标是，在合适的级别上围绕部署工作流建立定义的和可重复的最佳实践。这些最佳实践不但关注独立的活动，而且还关注团队中不同活动之间的流的数量和质量。过去，这大部分是通过教育和管理实践来推动的，这些实践受到了生命周期模型、敏捷方法和能力成熟度模型（Capability Maturity Model，CMM）的影响。在 DevOps 中，通过可重复的自动化，这些最佳实践正在被日益强化，虽然在执行和严格之间也存在着权衡。

对运维人员来说，“把你自己从作业中自动化出来”已经成为了这个社区的口号。IT 运维拥有复杂的自动化工具，以至于他们几乎能够重复他们工作的每一个方面。早期，配置管理工具，例如 CFEngine，在管理大型服务器集群的配置更新方面起到了帮助作用。甚至在虚拟化时代之前，诸如 LoudCloud 和 Opsware 等先驱企业就已经针对自动化的服务器和网络供应提供了解决方案。虚拟化技术更加简化了可重复的运维。VMWare 和 Amazon 云等厂商已经提供了应用程序编程接口和工具来帮助实现可重复性。运维人员

常常有自己喜欢的脚本和定时任务（Cron）来解决具体问题和流水线的不同工具和平台。

这些方法一直都是有用的，但它们并不一定是在合适的级别上执行了可重复的实践。首先，可能很少监管不同自动化任务之间的流，而且它们可能具有较低的可重复性。运维人员可能决定执行任务的特定序列，即使当每个任务都包含在脚本中且是可重复的。当我们考虑全流水线和它的可重复性时，这一点就特别重要。其次，运维人员也可能需要实时解决问题和故障以减少停机时间。在这个情况下，将产生巨大的压力。这些压力要求运维人员快速地做些事情，而不是首先定义可重复脚本要做的事情、测试脚本、对这些脚本进行版本控制，然后再运行它。在平衡可重复性与实时问题解决方面的一个必要步骤是，清晰地定义什么时候允许进行特定的不可重复的运维以及如何在事件后补救这个情况。最后，通过脚本和自动化工具定义运维本质上不实现可重复性，因为脚本是变化的且脚本的不同版本在不同的情境下可能运行不同的次数。在需要时，保证运维确实是可重复的不是一个可以简单实现的目标。这就导致了我们在下一节对版本控制的讨论。

9.2.2　版本控制所有事物

正如我们所描述的那样，DevOps 过程至少有两个层级。一些步骤是通过使用一个工具来执行的，一些步骤涉及了多个工具。为了保证可重复性，这两个层级都应该置于版本控制系统的管理之下。首先，我们讨论为什么一个工具（脚本）管理的步骤应该被版本系统控制。这些步骤是运维人员使用的脚本。

1）随着时间的推移，脚本会发生变化。造成这种变化的可能原因包括，对脚本进行了改进或者为了完成一个稍不同的作业而引入了新的变体。这样就很容易导致对何时在哪些系统上运行了哪个脚本的哪个特定版本失去追踪。任何改变基础设施或者环境的脚本或者代码都应该置于版本控制之下，与应用代码被版本系统控制一样。这正是基础设施即代码（Infrastructure-as-Code）的一个例子。通过这种方式，不但脚本 / 代码的所有历史版本都是可用的，而且关于这些脚本 / 代码变更的信息也保留下来，例如变更的理由、谁做了变更和变更的时间。

2）为了运行脚本通常需要输入一些参数，这些参数执行在一个特定环境上并对那个特定环境做出变更。追踪和重复一次执行或者理解一次执行不仅要求有被执行的正确版本的脚本，而且还需要把这些参数的值作为追踪执行的信息加以保留。

可以通过不同的方式实现追踪和重复一次运行。第一种方法是将一次特定运行的所有步骤记录在日志中，包括正在执行的步骤到了什么状态。这种方法高度依赖于你的脚本和其他人的工具 / 脚本（你无法完全控制其他人的工具和脚本）所产生的日志的质量和粒度。有时，你需要根据日志中不完备的信息来尝试重建一个已经完成的执行。第二种方法是捕获一次特定执行的状态变更而不管每个步骤如何和是否记录在日志中。一些状态（例如，基于文件的配置和工件）适合于版本控制，因为你可以简单地把这些文件放在

一个版本控制系统中，然后可以追踪对它们的所有变更（通过脚本或者通过人工）。其他的一些非基于文件的状态需要额外的工作以进行周期性的状态捕获。例如，应该周期性地捕获在云中运行的虚拟实例的状态和它们的关系并存储下来以便以后参考，特别是如果 DevOps 操作已经改变了它们。还有一些现成的工具，例如 Netflix Edda，它们用于捕获亚马逊云资源的状态变更以及查询这些变更，包括与最近操作的相关性。

这些追踪技术简化为检查点和日志的标准的可靠性技术。检查点环境状态，然后将检查点创建后出现的任何变更请求记录在日志中。如果日志变得太长，那么就可以创建一个新的检查点并重新启动日志。环境的状态和使用的参数可以通过检查日志和检查点来确定。

创建检查点和变更日志的最后一个任务点把所有步骤结合为一个可重复的高层级过程。完成这项工作的一些可能的方法和工具是：

- **部署工具**。可以通过部署工具来实现维护高层级过程的可追踪性，因为它是将实例放到生产环境中的最后阶段。
- **配置管理数据库**（Configuration Management Database，CMDB）。正如我们在第 2 章中所提倡的以及在第 12 章的案例研究中所学习到的，配置参数存储在数据库中。该数据库也可以记录访问，这样后来我们也可以知道访问这些配置的工具和脚本。
- **对数据项打标签**。每个脚本都处理一些实体。用处理它的每个脚本的标识信息对实体打标签，将导致最终部署的版本具有可追踪的信息。

总之，重要的是把特定日志（或者部分日志）与一个版本化的脚本的特定执行的状态快照相链接，这样你的操作以后就完全可重复和可理解了。

9.3　性能

DevOps 流水线类似于一件软件产品，其中性能体现在它所完成的有用工作的数量以及为了完成这些工作所花费的时间和资源。类似于传统软件的性能，DevOps 流水线的性能也可以通过以下的指标来测量：对给定的任务（例如，构建任务或者部署任务）的响应时间，这些任务在流水线的不同阶段的吞吐量；以及这些任务对底层资源（例如，计算资源和人力资源）的使用率。

9.3.1　测量重要的事物

在你可以提高流水线的性能之前，你应该首先测量它。在第 10 章中，我们将讨论一些 DevOps 流水线的高级业务性能指标。为了提高流水线的性能，我们需要更详细的度量。

从高级别来看，我们感兴趣的性能是从业务概念到它的成功部署之间的时间。在实

现和部署业务概念时，它历经了流水线的子过程，有时还会在这些子过程之间迭代。重要的是，测量它花在每个子过程中的时间。这些时间包括任务在队列中等待的时间和任务实际执行的时间。例如，集成测试资源可能被另一个不同分支上的提交所阻塞，你的提交被放置在一个队列中等待，直到这些资源被重新释放。

还要测量发生的不同类型的错误以及导致这些错误的原因。例如，构建错误是导致交付流水线变慢的一个关键来源。Google 最近的一项的实证研究表明，在所有的构建错误中按照数量来看排在最前面的构建错误与依赖性问题有关，其中 C++ 中为 52.68%，Java 中为 64.71%。理解这些错误可以指导改进工作。

第 3 种类型的度量关注合规性。正如前面章节所提到的，由于各种原因，主动或者机械执行最佳实践不一定是最优的行为。相反，对合规性监控暴露了一些问题的端倪。并不是所有的偏差都是问题，有一些可能是合理的。另一方面，多个合规性偏差可能意味着需要改进过程。对最佳实践来说，这是可选的；对监管规则来说，这可能是强制的。例如，通常要求金融业保证在将任何代码发布到生产环境前进行代码评审，这种代码评审是由未参与实际功能或者补丁开发的开发人员来完成的。虽然这个规则应该由正在使用中的工具来执行，但是在发布时进行合规性监控也是一个好的实践，以确保没有通过这种或者那种方式来绕过执行机制。

另一种性能度量可以关注当问题发生时与当问题被检测到和最后被修复时所消耗的时间。在问题检测和修复上出现的延迟可能指向了一些改进机会。

总之，我们的建议是：测量那些可能减慢流水线速度的事物的性能，然后根据每个问题的相对成本和收益把这些改进按照优先级排序。

9.3.2　提高资源使用率

为开发团队准备计算资源是一件代价昂贵和麻烦的事情。开发人员需要访问高性能的开发环境。准备一个内部的、物理的开发环境不但费时也费钱。也可能需要准备大量的构建和测试服务器集合。测试和预发布环境需要能够尽可能地模拟生产环境，这意味着需要昂贵的复制。在这种场景下，浪费现象可能是随处可见的，因为开发人员机器、构建 / 测试服务器，以及测试 / 预发布环境常常是为尖峰时段准备的，由此导致这些资源在非尖峰时段是未被充分利用的。

通过以下几个不同的策略，可以减少这种浪费：

- 把以上的所有环境都迁移到云上，从而关闭那些当前未被充分利用的服务器，并仅为你所使用的资源付费。这不仅提高了资源使用率，而且对于可重复性也起到了帮助作用。在它未被使用时拆除环境以及在需要时重新启动它可以自动执行要定义的一个可重复过程。这就带来了一致的和纯净的环境和良好测试的脚本。它进一步缓解了由不一致的环境所导致的问题——“它在我的计算机上工作”问题。利用云的灵活的可伸缩性和流水线设置，可以通过并行方式分配和使用更多资源。

这些额外的资源可以用于消除瓶颈或者通常提高各种流水线任务的速度。

- 使用容器而不使用虚拟机。相对于虚拟机，容器可以被更快地部署。利用容器来支持部署流水线目前还只处在初期阶段，但这类应用一定会增长。

把流水线完全迁移到云上的一个弊端是，如果环境是托管在第三方云上，那么你就失去了对你环境的完全控制和可视化。同时，也存在着小概率但影响巨大的风险，例如整个开发和测试环境的服务中断等。

9.4　可靠性

软件会失败。可靠性是指整个 DevOps 流水线和它的各个组件维持服务的能力。DevOps 流水线必须处理大量不同类型的工具。一些工具是开发环境本地的，例如代码编辑器和集成开发环境（Integrated Development Environment，IDE）。一些服务是由专用服务器来提供的，例如持续构建 / 集成。部署过程必须通过指定的操作系统 / 中间件机制或者在云环境中通过基础设施应用程序编程接口和虚拟机来处理基础设施服务。有些这样的服务必须通过网络进行远程访问。DevOps 流水线可以看作一个处理各种分布式服务的分布式超级系统。这些服务和它们的可靠性通常不在你的直接控制下。为了提高流水线的可靠性，你可以应用一些解决方案。

9.4.1　理解不同服务的可靠性特性

凭经验来理解流水线中各种服务和软件的可靠性是改善整体流水线可靠性的最关键步骤。软件发布和部署的频率增加意味着许多服务和软件每天都被访问数十次甚至数百次。你应该对这些服务进行压力测试，并且为这些服务建立可靠性和计时描述（timing profile）。重要的是注意这些服务的可靠性不一定是可以改善的，既因为该服务的控制权和所有权，也因为该服务的复杂度。

一旦你理解了各个服务的可靠性，你就可以使用多种技术来提高它们的可靠性而无须修改或者控制原始服务。

- 使用封装原始服务来提高可靠性。这种封装可以利用标准的容错机制。例如，可以利用计时描述来实现快速失败机制，这样就不需要等待慢于 95% 个百分位的服务响应。通过发送比需要（预想某些失败或慢请求）稍微多一些的请求来进行防范。虽然这可能导致额外的成本，但是在关键点上它可以极大地提高流水线的可靠性。你也可以使用这些封装来拦截某些请求，这样就可以将这些请求重定向到检查点关键状态或者重新设定这些请求到检查点关键状态的优先级以便在发生错误时实现更高效地撤销。
- 使用远程服务的本地镜像。这种技术经常发生在处理远程代码 / 软件存储库和依赖性解析的实践中。DevOps 流水线通常需要访问用于提供新实例的第三方库和

软件包。当偶然下载和使用一个新版本时，通过因特网下载所有的包有时是不可靠的，并可能导致版本冲突。本地镜像既可以提高存储库服务的可靠性，也可以强制实施版本期望（versioning expectation）。

9.4.2　早期检测和修复错误

与常规的软件开发和调试一样，在 DevOps 流水线上消耗的大量时间并不是在成功的构建、测试和部署上所花费的。相反，当出现问题时，你需要花费大量时间用于分析和修复问题。例如，花费在判断构建错误或者部署错误上的时间以及花费在回滚上的时间对流水线的性能产生了巨大的消极响应。

一种解决方案是在流水线的早期阶段执行更多的测试。但是测试会耗费大量时间，特别是当涉及多个目标平台和大量集成 / 系统测试套件时。同时，一些不易察觉的错误仅仅会随着时间的推移而显现出来，并且仅仅在大规模生产环境中才能暴露出来。

另一种解决方案是对一些高频率发生的错误通过更好的工具来进行更早期的明智而审慎的检查。例如，作为前面提到的 Google 实证研究的一项结果，正在进行大量研究以便让更好的依赖性解析服务器来完成向后和向前的依赖性解析从而更早地检测出与依赖性相关的问题（在构建错误被触发之前）。

很多早期错误是不易察觉的，但是它们也会在很多地方留下蛛丝马迹，例如在日志中或者监控中。你可以建立机制来更早地检查日志和断言期望的状态，而不是响应失败警报。其他一些错误可能只能通过对比随时间变化的历史趋势来检测，或者通过对比过去成功的执行来检测。第 14 章将介绍一些最新的研究结果，使用日志和断言比其他方式更早期地检测错误。一旦能够更早地检测到这些错误，就可以通过从这些错误中恢复来提高可靠性，这就引出了下一节对可恢复性的讨论。

9.5　可恢复性

与许多质量属性类似，DevOps 操作的可恢复性也应该是从一开始就是内置的而不是后来添加的。可恢复性的目标是实现故障后容易恢复，不管是因为内部或者外部系统导致的，还是人为操作导致的。你可以通过很多方式实现这个目标：

1）在操作逻辑中包含大量异常处理（exception handling）。这包括更多的防御性编程技术用于检测各种特定步骤的前置条件（pre-condition）和后置条件（post-condition），使用异常处理来修复或者优雅地退出到一个期望状态，例如该步骤执行前的一致性状态。

2）内置对外部监控或者恢复系统的支持。因为很多操作是长时间运行的，所以不总是很容易地同步检查操作步骤的输出，即使在初始成功后也不总是确定可以保持期望的状态。设想你的任务是启动一个应用实例。一次典型的成功启动需要花费数分钟，实例

会历经多个中间状态，这些状态表明成功提供了虚拟机、中间件栈、应用和它的正确配置。你只能够在数分钟后通过周期性检查来获知一些中间结果。同时，外部监控和健康检查服务可能已经检查了一些条件，并且，如果需要，已经从某些特定错误（例如失败的虚拟机）中恢复了。外部监控和恢复服务应该与指定操作过程的脚本联合工作。

3）在设计软件时，把运维人员视为首要的干系人。很多复杂的恢复任务必须是在运维人员诊断了问题的根源后通过手动完成的。重要的是，运维软件或者通过日志或者状态捕获工具来产生相关的信息，这些信息让运维人员更容易做出明智的恢复决策。

4）使长时间运行的操作的每个独立步骤都能够自我恢复，这样只有在很少的情况下才需要回滚整个操作。

9.6　互操作性

互操作性是指，在特定的环境下，不同工具可以通过接口有效地交换信息的程度。DevOps 流水线通常包含了很多不同的工具，这些工具来自不同的商业厂商和开源项目。对于这样的流水线来说，这是预期的标准，其中使用高度专用的软件来实现各个任务是最好的选择，并且不同工具的互操作是以“最大努力”（Best Effort）的方式来实现的，而不是通过一种自上而下（top-down）的规划来实现的。我们可以通过一些方法在不同的软件之间实现更好的互操作性。

9.6.1　注意接口的互操作性

虽然团队可以为完成他们的任务而构建一个内部版本的工具或者定制化一个现有的工具，但是在大部分情况下，他们必须实现现有工具之间的互操作，而他们对于这些工具仅仅有有限的控制。重要的是，选择那些具有稳定应用程序编程接口、灵活脚本能力和活跃插件生态系统的工具，其中活跃插件生态系统包含与其他工具互操作的很多插件。

例如，流行的源代码控制系统 Git 可以支持与静态分析、测试、构建以及通知 / 消息系统的互操作性，这种互操作性是通过基于事件的钩子来实现的：使用恰当的 Git 输出其他系统的调用可以在 Git 事件之前或者之后执行，例如提交或者推送就是这样的 Git 事件的例子。

9.6.2　理解现有的数据模型

良好的互操作性依赖于隐含在流水线中的每个工具中的数据模型。重要的是决定在互操作的工具之间需要交换的主要数据模型的语法和语义。但这一点并不总是容易实现的。

例如，持续集成（Continuous Integration，CI）实践是相对成熟的，并且有大量支

持持续集成的工具。这些工具中的数据模型都有一个有关构建的明确概念。然而，当我们把流水线扩展到持续部署（Continuous Deployment，CD）时，我们需要追踪将哪个构建部署在哪里以及何时部署。而且，不是所有的构建都部署，或者一个特定的部署可能需要连接回来自多个构建的信息。持续部署中隐含的数据模型更多是关于工件以及到基础设施和环境的映射。在把持续集成工具应用于持续部署目的时，这两个数据模型间的差距正在导致一些问题。为了诊断调试监控目的当需要从部署到构建的反馈时，这些问题就显得特别重要。如果互操作只是关于从持续集成到持续部署的触发器，那么一些关键信息将会丢失。有些工具，例如 Go 持续交付软件，正在通过以下方式来解决这个问题——提供更丰富的数据模型来捕获持续集成 / 持续部署的需求。如果你正在使用现有的工具，那么你可能需要在你的协调脚本中包含额外的数据模型以追踪这些映射。

9.7　可测试性

可测试性是指通过测试来证明软件中错误所需要的付出。对于应用软件来说，开发人员实践了单元测试、集成测试等。然而，DevOps 流水线提出了额外的挑战。这些挑战来自测试基础设施结果的难度。我们记得，基础设施即代码是关于使用代码（而不是手动命令）来设置（虚拟）服务器和网络、安装软件包以及为感兴趣的应用配置环境。然而，基础设施相关代码的可测试性是困难的，因为代码的真实执行包含了长时间运行的任务，例如运转虚拟机、下载和安装软件以及在大量节点上可靠地执行所有任务。仅仅知道你的命令已经收到和开始执行是不够的——你还需要知道在数分钟甚至数小时后你所期望的结果是否实现了。

以 Chef 为例。Chef 是一个流行的工具，为了准备基础设施和部署应用，人们使用它配置大量的系统。你可以通过写 Chef Cookbook 来表示你希望 Chef 在节点上（包括物理机、虚拟机、LXC 容器等）做的工作，例如安装 / 升级软件包和应用配置变更。Chef 系统将尝试在所有节点上应用 Cookbook。如何测试 Chef Cookbook 代码以保证它执行了你期望的工作？如何保证代码质量是良好的？

与其他任何类型的代码类似，你可以对 Cookbook 执行单元测试。ChefSpec 正是对 Chef Cookbook 执行单元测试的工具。基础设施代码的单元测试（例如 Chef Cookbook）的含义与传统的单元测试稍有不同。请记住，通过收敛，代码将在真实节点上安装 / 更新软件包并进行配置。在执行单元测试时，如果你确实启动了大量测试实例并等待以便安装和更新这些实例，那么这将是漫长的和代价昂贵的。这违背了单元测试的主要目标——快速和本地执行以便让开发人员得到快速的反馈。因此，ChefSpec 实际上测试 Chef 的输入是否是你所期望的，特别是当逻辑复杂时。Chef 运行在开发机器的内存中，它并不实际执行或者完成收敛。通过这种类型的单元测试仍然可以在早期梳理出一些问题。

一旦你通过了所有的单元测试，那么你就可以运行集成测试了。集成测试真实地测试运行在测试环境中的 Chef 的输出结果。实现这一目的的工具包括 Test Kitchen 和 Serverspec。Test Kitchen 负责管理集成测试，例如实际启动测试机器（例如 EC2 实例或者 LXC 容器）和收敛运行在这些机器上的给定的 Chef。然后，Test Kitchen 执行实际的测试以保证这些机器处在你所期望的状态上。这些测试本身使用 RSpec 语言写在 Serverspec 中。RSpec 语言使用了人工可读的形式，它连接回业务场景，也简化了指定集成测试用例的工作。

如果你遵循了测试驱动的开发实践并首先编写测试套件，那么你就可能产生改善了可测试性的代码。

单元和集成测试使你对代码质量和预期的行为产生了信心。然而，执行模拟真实生产环境的大规模系统测试仍然是一个难以实现的任务。

当进行到集成和系统测试时，你可能注意到你编写的测试用例常常非常类似于你设置的监控规则。这一点儿也不奇怪，因为监控是关于运行时断言和对一次成功准备的监视的。基于重用和“一次且仅一次”的精神，你可以考虑在监控配置中重用测试用例，或者把这些测试用例作为监控规则。相应地，如果有一些监控规则是与部署的期望结果相关的，那么使用这些监控规则作为测试用例。

正如前几章提到的，生产环境中的测试通常用于测试复杂环境中运行的大规模复杂应用，其中某些问题只能在生产环境中发现。金丝雀测试（在生产环境中的一个小的服务器子集上运行新版本或者配置）可以最小化这种测试的风险。这种相同的思想也可以用于基础设施即代码测试，在这种测试中，影响基础设施的代码在生产环境的一个子集上运行，它仅影响全体应用的一小部分，同时需要严密监控它的行为。

9.8　可修改性

可修改性是指在现有软件上进行修改的能力的度量。当可修改性应用在部署流水线上时，它意味着以下两种变更能力：变更流水线中一个阶段的交互行为或者变更从一个阶段进行到下一个阶段的条件。从某种意义上，可修改性意味着预见可能需要的变更类型。我们将可修改性讨论分为两类：处理流水线中的一个工具的修改，处理流水线中的工具之间的交互的修改。

9.8.1　一个工具内的修改

实现软件内可修改性的一个基本技术是，将相关活动封装成模块，并使各个模块之间尽可能地松耦合。把相关活动保持在一个模块内的理由是，该模块内一个活动的变更将需要模块内的其他活动也发生变更，但是希望这些变更不要扩展到一个模块之外。

这个通用建议也适用于为流水线中的工具所编写的脚本。然而，封装意味着什么以

及使两个模块松耦合意味着什么，可能有不同的解读。如 Chef 或者 Puppet 这样的工具都是声明式的，而不是过程式的。从根本上来说，Cookbook 描述了对一组实体的预期编排，这种编排是执行该 Cookbook 的结果而不是解释如何实现那样的编排。

当应用于 Cookbook 时，对封装相关活动的建议是：保持较小的 Cookbook，并让 Cookbook 关注一个任务。对那个任务的变更可以通过修改一个 Cookbook 来实现，在这个 Cookbook 中，声明语句之间的交互是清晰的；在一个更大的 Cookbook 中，可能出现意想不到的交互。

Cookbook 之间仍然可能有交互，而且这些交互可能难以控制。一种技术是利用由 Chef 或者 Puppet 提供的排序机制来控制 Cookbook 的执行顺序。利用这种排序机制可以对 Cookbook 之间的交互提供更多了解和控制，而不是让 Chef 或者 Puppet 来决定执行的顺序。

实现软件内可修改性的第四个基本技术是，把变量参数化而不是把这些变量构建到代码中。这就为流水线工具创建了一个配置参数。在第 5 章中，我们讨论了对配置参数的管理。配置参数提供了一个可以为活动指定数值的地方，这些值可能因为使用者或者情境不同而发生变化。但是这些参数也是必须被管理的。把这些配置参数放在配置管理数据库中提供了一个可以对配置参数进行控制、修改和访问的中央位置。

9.8.2　工具之间交互行为的修改

在第 12 章的案例研究中，我们看到了这样一个事实的例子：不同开发团队之间的部署流可能是不同的。流水线的组成随着时间的变化也可能会改变。相对于单个工具来说，流水线上可能发生的变更类型包括：

- 替换同一个工具的不同版本。工具之间可能存在版本依赖，同一个软件应用程序中同样存在版本依赖。预测新版本中将发生哪些变更是一项困难的活动。在第 6 章中，我们讨论了如何隔离系统的一个部分使之不受其他部分变更的影响。
- 替换不同的工具。通常，替换一个工具将需要重写连接脚本。自动地从使用一个工具迁移到使用另一个工具是一件非常困难的事情。有时，工具厂商会提供迁移工具以辅助这种变更，但是这个情况[⊖]在 DevOps 领域内是十分罕见的。
- 修改必须从一个工具发送到另一个工具的参数。使用配置管理数据库减少了该问题的范围。一个工具向数据库写信息，另一个工具从数据库中读该信息。

9.9　小结

正如存在对任何应用软件的质量关注点一样，也存在着类似的 DevOps 流水线的质

⊖ 指工具厂商提供辅助迁移工具。——译者注

量关注点。DevOps 流水线需要处理不同类型的软件，从桌面集成开发环境（Integrated Development Environment，IDE）到代码存储库、测试 / 构建服务器、云基础设施应用程序编程接口和（测试、预发布和生产的潜在污染环境）。随着软件交付和部署频率的增加，为了获得本章讨论的非功能需求，出现了一些新的挑战。关键的结论是，非功能需求和质量关注点应该是在初早期考虑并且应该是内置的，而不是事后将它们追加进去。当你处理超级系统中的现有工具时，这一点并不容易实现。

表 9-2 总结了我们讨论过的技术，这些技术用来实现特定的质量。但是，天下没有免费的午餐。本章提到的用来获得一项非功能需求的预期结果的每种技术，都包含了对其他非功能需求的权衡。理解那些权衡对于实现满足你需求的流水线是非常重要的。

表 9-2　实现这种质量的技术总结

非功能需求	实现这种质量的技术
可重复性	维护活动的踪迹。版本控制所有事物。使用配置管理数据库来维护参数。在需要的地方执行
性能	测量以判断过程中的瓶颈。在它未使用时拆解环境。在云上执行尽可能多的操作，因为云上的资源在未使用时可以被释放
可靠性	识别不同服务的故障率。对高故障率的服务建立镜像。通过工具尽快地检测故障，这些工具的任务是监控组件以发现其执行时间处在正常范围以外
可恢复性	在脚本中内置异常处理。为监控服务提供信息。保证生成合适的诊断信息以支持更快地调试
互操作性	选择那些具有稳定接口和灵活脚本能力的工具。保证流水线的不同阶段的数据模型是一致的
可测试性	为专用工具使用单元和集成测试。协调测试用例与监控规则
可修改性	基于对工具的预期变更来模块化脚本。将运维动作封装到小模块中，这些小模块之间是松耦合的

9.10　更多阅读材料

有关软件架构中的非功能需求，可以阅读参考文献 [Bass 13]，它包含了专门的章节来讲解一些非功能需求方面的内容。

通过阅读参考文献 [Seo 14]（发布的 Google 研究）可以了解常见的构建错误。

如果想了解更多关于基础设施代码可测试性和测试驱动开发，可以在参考文献 [Nelson-Smith 13] 中找到一些不错的资料。

第 10 章　业务关注点

如果想让上帝大笑，给他看看你的商业计划书。

——Barry Gibbons

10.1　概述

本章将从管理的角度讨论 DevOps。因此，它是面向有定义过程的组织。如果你在一个创业企业中，那么管理结构将是稀疏的，官僚主义也比较少。随着组织规模的增长，除了极少的案例以外，组织结构几乎都会有所变化。

在已建立的企业中引入一项重要的新技术通常是一个自下向上和自上向下的过程。只有最简单的技术变更，那些只影响一个团队的技术才可以引入而不需要管理层的认同，并且没有得到来自技术专家（在这个案例中，专家是指开发人员和运维人员）的认同，不能引入任何技术变更。DevOps 的支持者要求在组织结构上和组织与外部干系人的交互方式上做出重大的变更。因此，为了获得管理层的支持，需要让管理层确信这些变更所带来的收益大于成本。因此，我们首先从 DevOps 的业务案例开始讨论，然后再讨论 DevOps 实践的量化指标和合规性。在本章的结尾，我们将触及组织内开发人员和运维人员之间相互交互的其他方面。

10.2　业务案例

业务案例必须在以下 4 个方面说服管理层：1）存在一个被攻击的真实问题；2）解决问题的方法是合理的；3）解决问题所带来的好处大于成本；4）干系人不会过于失望。管理层也希望看到对风险和风险缓解的枚举、初始推出计划和项目的成功标准。表 10-1 显示了一个用于引入 DevOps 的典型业务案例的组成部分。本节的后续内容将讨论这些点。

表 10-1　引入 DevOps 的业务案例的组成部分

组成部分标题	内容
问题	为什么对组织来说引入 DevOps 实践是有好处的
成本	引入 DevOps 实践的预期成本是什么
干系人影响	对内部和外部干系人的影响是什么

（续）

组成部分标题	内容
风险和缓解	与引入 DevOps 实践相关联的组织和技术风险是什么如何减缓这些风险
推出计划	推出 DevOps 实践的计划是什么
成功标准	我们如何知道 DevOps 实践的引入是成功的

10.2.1　问题和解决问题所带来的好处

在所有案例中使用敏捷的目的都是为了减少从业务概念到将它部署给用户之间的时间。为了让这个论点对 DevOps（可以说 DevOps 正是由敏捷所启发出来的）更具体，请参考第 1 章。DevOps 的目的是减少从向系统提交变更到这个变更被部署在正式生产环境所需要的时间，同时保证高质量。保证高质量意味着，在将最终的高质量系统提升到正式生产环境之前，可能进行多次问题检测和问题修复的迭代。减少问题检测和问题修复之间的时间也是重要的。

因此，两个重要的度量是，从提交到初始生产环境的时间和问题检测到修复的时间。对于这两项，组织的当前状态应当作为基准。当前从业务概念到代码提交然后到部署的时间（时间分布、中值时间和平均时间）是多少？问题检测与修复之间的当前时间是多少？业务案例也应该对因引入 DevOps 实践所实现的价值设定目标。

设定目标是一项困难的活动。首先，关于 DevOps 实践的有效性有有限的定量报告。其次，正如我们在第 1 章中识别出来的，有 5 种不同类别的 DevOps 实践，每种实践对实现目标价值都有一定的影响。在关于推出计划的章节，我们将回到 DevOps 实践的类别。最后，每个组织都是不同的——因此，虽然有工业基准可能有所帮助，但是这些基准仍然需要适应本地情况。

当前价值和目标价值之间的差值表示 DevOps 带来的好处。这些价值不必是财务的。例如，减少业务概念与它部署之间的时间所带来的好处，可以以时间作为度量。减少这些时间所带来的财务价值比评估时间上的减少更加具有不确定性。我们要做的案例就是，具有重要影响力的干系人会从这些时间的减少中获得好处。

组织的变更需要拥护者。在理想情况下，在技术层面和管理层面都有这样的拥护者。另外，在理想情况下，拥护者应该包括来自两个主要受影响的组的代表——Dev 和 Ops。这些拥护者应该是对准备业务案例负责的个体。

10.2.2　成本

与 DevOps 相关联的成本中有一部分是持续成本，有一部分是一次性成本。持续成本与工具和人有关。引入 DevOps 需要拥有一组工具集。一旦获得了这些工具，不管是开源的还是商业的，都需要对它们进行管理。不管是在团队层面还是在组织层面，某个人需要对获取和构建新的发布以及对使这些工具保持可用负责任——请参考在第 12 章中的

讨论。这种责任可以指派给现有的团队，但这种做法并没有有效地减少成本。必须确定这些成本。而且，也需要相应的培训来学习这些工具的使用并了解它们的特性。新职员必须学习如何使用这些工具以及与这些工具相关联的过程。在新工具替换现有工具的情况下，也应该考虑授权费用和维护成本。一般来说，持续成本可以与现有的各个成本进行比较，以判断这些成本是减少了还是增加了。

一次性成本是引入 DevOps 实践的直接花费。在这些实践的初始执行时，往往比后期的使用低效。工具需要引入，人员需要培训。这些任务可以通过内部方式或者外部方式来完成。你的组织可以雇用咨询顾问来指导你们完成 DevOps 实践的引入，或者你的组织可以决定全部依靠内部人员来引入 DevOps。第 12 章的案例研究给了我们一个例子，在那个例子中咨询顾问指导 DevOps 的引入。而第 13 章的案例研究给我们讲述了一个依靠内部人员来完成 DevOps 引入的例子。业务案例应该给出推荐，要么依靠外部咨询顾问，要么依靠内部人员，同时要提供做这种选择的依据。

另一种一次性成本是修改现有系统以支持 DevOps 实践的花费。这些现有系统可能包括软件工具、现有过程或者现有产品。业务计划应该探究两个方面：需要对现有系统进行怎样广泛的修改？将资源转移到修改现有系统对未来的开发计划有什么样的影响？

10.2.3　干系人影响

迁移到 DevOps 可能对干系人造成影响。如果没有造成影响，那么为什么做这个事情呢？干系人可以分为内部的和外部的。

1. 内部干系人

很明显引入 DevOps 会影响两类内部干系人，即 Dev 和 Ops。另外，将创建一类新的干系人（执行 DevOps 角色的个体）。一般来说，Dev 组将获得额外的职责和控制，而 Ops 组将失去职责和控制，DevOps 将成为新的角色。多少职责和控制被转移取决于采用的 DevOps 过程。在第 1 章中，我们识别了 5 类 DevOps 过程。这里，我们将根据这 5 类 DevOps 过程来讨论额外的职责和转移职责。

1）把 Ops 视为首要干系人。在这种情况下，Dev 获得了额外的职责——从 Ops 获取需求，同时 Ops 也获得了额外的职责——提供需求。另外，Dev 和 Ops 都获得了额外职责，那就是保障需求是满足的。

2）涉及 Dev 更直接地参与事故处理。在这种情况下，Dev 承担了与事故处理更紧密关联的职责，这些事故包括一个系统。Ops 和 Dev 承担了定义一个过程来识别哪些事故首先通知 Dev 和哪些事件首先通知 Ops 的职责。

3）执行将把软件变更部署到生产环境中的一致性过程。在这种情况下，Dev 和 Ops 都参与定义这个过程的。最大的可能是，DevOps 角色负责保证执行了这个过程。

4）采用与开发应用代码相同的实践集合来开发基础设施。因为基础设施代码主要是

由 Ops 或者 DevOps 角色来开发的，定义和执行这些实践的职责应该由他们来承担。

5）实现持续部署流水线。在 Dev 开发的这些项目的范围内 Dev 组负责分配资源、发布计划和制定部署决策。当 Ops 组与一个系统或者部署相关联时，Ops 组将失去那些职责。DevOps 角色对 DevOps 工具负有完全的责任。这不仅包括安装和维护工具的适当版本，还包括培训其他团队来使用这些工具。

灾难恢复是大家共同承担的职责。Dev 组对何种程度的复制被构建在系统架构中负责。DevOps 角色负责通过部署决策来管理的部分灾难恢复的工作，如第 11 章的案例研究。收集和报告整体可用性指标是 DevOps 角色和 Ops 团队共同承担的职责。如果灾难发生，则 Ops 团队负责触发迁移到备用站点的工作。

应对 DevOps 工具中的变更可能让 Dev 团队中产生烈火灼心的感觉。如果 DevOps 角色持续地完善使用这些工具的过程，那么这些变更可能让 Dev 团队应接不暇。完善过程是 DevOps 角色章程中的一部分，但是 DevOps 团队应当对其他团队吸收持续变更的限制保持敏感。

2. 外部干系人

如第 1 章所述，使用正式发布过程为业务干系人和管理层干系人提供了对发布特定功能过程的高度可视性。当功能需要多个开发团队全部发布他们自己的部分而每个团队又自主决定何时发布他们自己的部分时，决定发布一个功能的过程将会变得困难。

然而，DevOps 实践的主要目标是减少从向系统提交变更到将这个变更部署到正式生产环境（同时保证高质量）的时间。业务干系人和管理层干系人需要理解这样一种情况——在采用 DevOps 部署实践时，他们做出了一种权衡：为了获得更快的生命周期时间，他们放弃了正式发布过程所需要承担的可视性。虽然每个团队的质量和速度可能都很高，但是他们的独立性是通过牺牲发布过程的协调性换来的。

10.2.4　风险及其减缓

与引入 DevOps 实践相关联的风险既有组织层面的也有技术层面的。

1. 组织风险

在博客和其他论坛上对 DevOps 的许多讨论是关于打破 Dev 和 Ops 之间的壁垒。壁垒是存在的，因为这两个组织单元拥有不同的使命、不同的文化和不同的激励。让这两个不同的组织单元联合起来意味着至少需要对对方单元的使命、文化和激励具有较高层的理解。而且，不同的报告链也是一个壁垒，物理距离是另一个壁垒。社会科学研究已证实，超过 30 米的物理距离会造成人们之间协作的壁垒。

DevOps 工程师的新角色的创建也会在组织内部造成压力。这个角色必须配置职员，必须为这个角色制订报告线。为新角色配备职员和宣告新角色将在那些未参与这个新角色的人中造成紧张气氛。

另外，把脚本和配置置于版本控制系统中并控制如何部署系统的新版本可能表示当前实践的变更。因此，那些受影响的人可能会拒绝这样的变更。

减缓这些风险包括处理我们前面提到的那些方面——不同的使命、文化、激励和报告链，以及受影响人之间的物理距离和对现有实践的变更。

管理这些变更既是学术研究的主题，也是多个咨询顾问的专业知识的主题。在DevOps业务计划中，你应该在你的组织内识别出为引入DevOps而做出的尝试以及如何管理这些尝试并且取得了哪种程度的成功。减缓这些风险的一个建议方案是调整各组的关键业绩指标（Key Performance Indicator，KPI）以反映部署中的全局的而不是个体的成功。也就是说，当前Dev关键业绩指标强调了新功能的代码编写，而当前Ops关键业绩指标强调了系统的稳定性。相反，在将新功能成功地部署到生产环境而没有影响系统稳定性时，你应该同时奖励这两个小组。

2. 技术风险

从管理的视角来看，业务计划应该回答两个基础的技术问题：需要对应用的现有生产架构做什么变更？如何维护生产数据库的完整性？如果生产架构要修改以支持持续部署，那么受影响的架构应该识别为风险。同样，组织的生产数据库代表了组织最重要的资产之一。对生产数据库完整性的威胁应该识别为风险。

（1）现有生产应用需要的变更

为了支持持续部署，可能需要对现有生产环境应用架构做两种类型的变更——更多详细内容请参见第二部分。

- **状态管理**。如果可能的话，组件应该是无状态的。在发生故障时，无状态的组件更容易恢复，因为在组件无状态的情况下替换一个故障组件不是一件困难的事情。持续部署实践也是基于替换组件和服务构建的，这些替换行为是频繁发生的。包含状态的组件需要识别出来，然后进行修改把这些组件上对状态的维护去掉。从一个组件去除状态的方法有多种：把状态存储在数据库中；让客户端维护状态；使用状态协作服务（例如ZooKeeper）。
- **功能开关**。如果使用了蓝/绿部署模型，那么不需要功能开关。如果使用了滚动升级部署模型，那么就需要使用功能开关来实现对新功能的控制，同时应该引入功能开关经理来控制功能。

（2）维护生产数据库的完整性

生产数据库的完整性可能被以下两种情况破坏：

- 测试数据被错误地包含在生产数据库中。保证预发布环境是与生产环境相独立的，因此生产数据库，可以通过使用自动化的脚本、凭证和防火墙规则来实现。
- 对生产环境的部署破坏了该数据库。从数据库中的错误数据恢复无异于没有使用DevOps实践。业务计划应该包括对当前回滚、前滚计划的讨论，以实现对错误数据的修正。

10.2.5　推出计划

任何推出计划都依赖于你组织当前的位置和你的最终目标是什么。与 DevOps 工具一样，你可以实现“创世大爆炸”（Big Bang）交付，这种方式下，所有事情是一次性完成的；或者也可以实现增量交付，这种方式下，引入 DevOps 实践，允许有一段时间来寻找缺陷并让人们理解和适应这些新的实践。创世大爆炸方式会更快速地交付结果，但是它更容易引起更多的抵触和遭遇更多的错误。考虑到对现有软件产品带来的变更（例如，采用微服务架构来促进 DevOps 实践），采用增量交付通常是更可取的方式。在第 13 章的案例研究中，我们将讨论如何实现这样的增量变更。第 12 章的案例研究建议设置一个入职培训团队，这个团队的唯一工作就是帮助其他 Dev 团队把他们的应用迁移到新的 DevOps 工具集上。因此，这就强制执行了一种增量路径——在任何时间点上，入职培训团队仅仅可以处理一定数量的项目。

成熟度模型作为判断组织是否处于一个特定规模和组织需要实现哪些工作以便在那个规模上变得更“成熟”的方法已变得相当流行。在 10.6 节中引用的 DevOps 成熟度模型定义了 5 个不同的类别来表示在实现持续交付时需要考虑的关键方面。这些类别是：文化和组织、设计和架构、构建和部署、测试和验证、信息和报表。

为了撰写业务计划中的推出计划部分，你需要使用这些类别来识别在一定时间周期（例如 2 周、1 个月、2 个月和 6 个月）以后你希望你的组织达到的状态。识别在每个活动中你需要采取的步骤以及你所选定的每个时间周期的特定目标。

例如，在设计和架构类别中，你可以识别在每个时间周期内哪些应用要实现哪个比例的组件无状态化。在构建和部署类别中，你可以识别哪些工具在生产环境中使用以及在部署流水线的哪些阶段使用，并且识别谁首先使用这些工具以及在每个时间周期结束后由谁来使用这些工具。

在第 1 章中，我们识别了 DevOps 的 5 个不同方面。这些不同方面可以作为推出一系列 DevOps 实践的指导。

- 在开发系统的需求时，让 Dev 把 Ops 作为首要干系人。一方面，这是最容易实施的实践。每个组织都有一个过程用于开发系统的需求，这些需求是否指定为用户故事，还是以某种更加正式的形式来指定。在开发需求的过程中，应该以某种形式将干系人包含进来，所以类似于其他干系人实施这种实践涉及 Ops。另一方面，对某些组织来说，把 Ops 视作首要干系人代表着文化的变换。文化的变换是一个组织最难做出的变更之一。一种技术是让一位在 Dev 社区有影响力的成员作为领头人。组织有独立于管理层结构的影响力层次结构。在推出计划和从 Ops 收集需求方面，让 Dev 社区中有影响力的成员参与会对 Dev 社区中的其他成员产生连锁反应。
- 让 Dev 更直接地参与事故处理中。这是一个比把 Ops 视为首要干系人更困难的主

题。它不仅包含了文化的变化，而且还包含了职责和过程方面的变更。对于在这个过程中的试点应用来说，第一步是定义由开发组处理的事故。这些事故应该是对这些试点应用特定的。直到你对描述事故有了一定的经验后，你才可以跨应用来统一描述事件故的特征。在一组事故已经定义后，必须定义一个把那些事故输入给 Dev 的过程。这个过程依赖于事故的类型并知道这个事故如何到达组织。如果它是一个来自外部的缺陷报告，那么应该将它定向传送给 Ops 和 Dev，Dev 的联系人可以决定下一步如何进行。如果它是一个内部的报告，例如基于监控工具的警告，那么监控系统可以依据规则来指派联系人。参见第 11 章中的例子。每个事故的来源都需要有它自己的向 Dev 或者 Ops 通知事故的方法。最后，需要在 Dev 组内寻找事件联系人。在这种情况下，个性比影响力更加重要。作为联系人角色的人员应该能快速解决问题，因为他们的首要任务是补救引起事故报告的问题。作为联系人角色的人也应该是坚定自信的（如果不是使人厌烦的），因为他们的下一个任务是对 Dev 过程提出变更的建议以防止这一类事件再次发生。

- 为了将软件变更部署到生产环境，强制执行一致性过程。在这种情况下，从结束处开始。也就是说，为将系统部署到生产环境中，定义一组一致性的闸门（Gate）。这些闸门包括对架构进行检查以确认部署新版本不会导致当前正在运行版本的操作中断。接下来，你可以定义一组闸门以便在部署中用于修改系统。这些闸门应该应用于对系统的任何变更，不管是在现有虚拟机上打补丁（如果你选择允许这样做）还是烧制一个新的镜像。在将软件部署到生产环境时回滚是一个重要的考虑因素，所以回滚过程应该是一致性过程的一个部分。在定义了这些闸门且正常工作后，有可能考虑导致对系统进行修改以及附加的自动化的步骤了。然而，第一步骤还是从定义闸门开始。
- 开发基础设施代码时采用与开发应用程序代码相同的实践。这种情况下，我们的建议是，在软件开发周期的早期阶段开始这样的实践，而不是在将变更部署到生产环境的最后阶段。配置管理和版本控制是应用程序代码开发中行之有效的实践。确保在开发基础设施代码的过程中遵守这些实践。基础设施库代码的自动化测试可以通过使用独立的环境来实现，也可以通过特定代码类型的专用测试框架来实现。
- 实施持续交付流水线。第 12 章的案例研究描述了一个由 Sourced Group 实施的持续交付流水线。在实施持续交付流水线时需要做出的最基础的决策是，不同的开发团队需要何种程度的协作以生成一个发布。要求的协作越多，每个团队的独立性越少。我们在第 4 章和第 6 章中对此进行了描述。

10.2.6　成功标准

成功标准基于推出计划和采用 DevOps 的理由。推出计划为已识别出的每个时间周

期内的各个类别提供指标。即使这些指标都达到了，也需应该进一步实现采用 DevOps 的总体目标：减少从向系统提交变更到将这个变更部署到正式生产环境中（同时保证高质量）的时间。这个目标对更高层面的目标（让终端用户获取业务概念）产生影响，同时这个目标也依赖于减少从错误检测和到错误修复之间时间的更具体的方法。

从业务案例的角度来看，一个重要的元素是要有可度量的成功标准。将你的努力关注在你可以收集的指标上，这些指标包括在引入 DevOps 前和引入 DevOps 后的各种指标。你不应该处于这样的位置：你辩护说这个项目是成功的，但没有足够的数据来支持你的论点。

10.3　度量和对 DevOps 实践的合规性

在第 7 章中我们阐述了在设计度量时，应该有特定的目标。在本节中，我们讨论业务感兴趣的各种度量，这些度量涉及 DevOps 和它的采用。有 3 类要测量的指标：DevOps 实践达到了何种程度的成功；不合规 DevOps 实践的案例有哪些；以及干系人对 DevOps 实践的满意度级别是什么。

10.3.1　测量 DevOps 实践的成功度

DevOps 实践的目标和流水线的阶段决定了应该采取的度量类型。特别是，DevOps 成功度的 2 个主要度量是从提交到生产的时间和从错误发现到错误修复的时间。

减少提交和部署之间的时间。图 10-1 重复了第 5 章中的部署流水线。从提交到它成功部署之间的时间是以下时间的和：在部署流水线的每个阶段上的队列中的等待时间；在流水线中的每个服务器上花费在处理上的时间。在持续集成服务器上，记录以下度量：随时间的推移活跃的分支数；从分支创建到将它合并主干的时间；以及它执行测试所花费的时间。因为最佳实践不鼓励使用分支，或者至少要求尽快将分支合并到主干，所以测量分支数和它们的生命期能够让你评估遵循最佳实践的程度。大量的测试时间是延迟的另一个来源，尽管它是一个错误数量之间的权衡，从测试套件中溜过的错误数量之间的权衡。

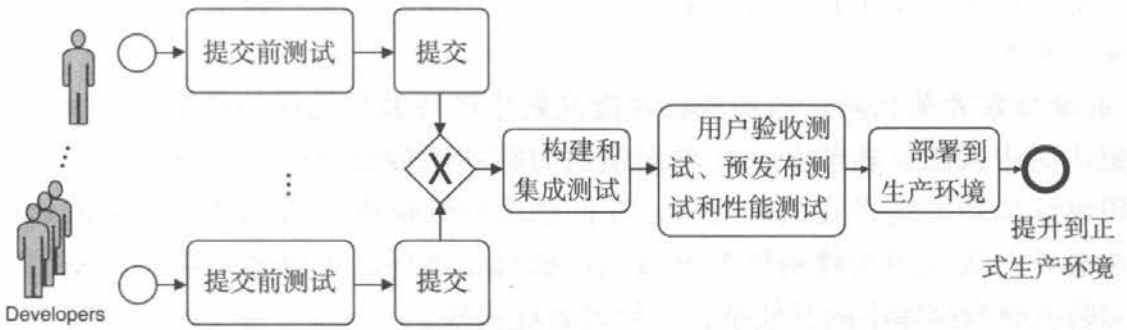


图 10-1　部署流水线（从图 5-1 复制而来）[标注法：业务流程建模标注]

在预发布环境中执行的度量记录了这个阶段花费的时间。执行测试花费的时间是多少？镜像在预发布服务器上的停留时间是多少？如果需要人工审核，那么需要多少时间才能获得部署该变更的批准？

不是所有的提交都能够带来成功的部署。有些提交没有通过测试，导致返工和重新进入流水线。Damon Edwards 提倡独立地测量返工。因为中断构建会导致每个团队成员无法成功地推进代码在流水线中前进，所以监控中断的构建和它们的修复也是同样重要的。特别地，这样的修复应该是最高优先级，度量可以判断是否遵循了指导方针。

最后，你应该能够判断把一个镜像从预发布环境部署到服务中时发生的错误数量和类型以及实际花费的错误修复时间。也就是说，测量在部署过程中发生的错误。这些错误与在被部署的系统中发现的错误是不同的，这个内容将在下一节中讨论。这些度量可以通过部署工具和生产环境获取。

聚合这些度量能够让你了解到部署流水线的性能。你想要了解流水线中的最长延迟在哪里，并集中精力减少这些延迟。

减少从错误发现到错误修复的时间。在这个情境中的，错误是指在一个服务的生产环境版本中发现的错误。关于这个目标，事实上有两个不同的方面。首先，流水线的各个阶段的自动化是增加了还是减少了最终漏网到生产环境中的错误数量和严重程度？其次，从错误发现到错误修复的时间是否因为引入了 DevOps 实践而有所变化？工单系统（ticket system）记录了生产错误及其影响，可以挖掘这些记录以确定这两个问题的答案。

10.3.2　测量对 DevOps 实践的合规性

如果认为人们总是遵守预先设定的实践，那么就大错特错了。人们可能由于缺乏知识或者由于顽固而不遵循实践，因为实践引入了额外的工作，或者仅仅因为他们不愿意做出改变。我们识别出两类在合规性方面可能出现问题的实践。

1）*启动虚拟机*。如果每个人都遵循部署实践，那么使用的工具能够维护历史信息，并可以发现实例的出处。一个未遵从实践的例子是，在某个事故中当运维从控制台启动一个虚拟机时。为了发现那些未通过文档化的过程就把变更部署到生产环境中的虚拟机，有必要扫描运行中的虚拟机。每个运行中的虚拟机都应该是从经过了所有闸门的完整测试的镜像创建的。

2）*去除功能开关代码*。当功能已经提交到生产环境且已保持稳定后，功能开关代码应该从源代码中去除。功能开关经理知道何时将功能提交到生产环境中。经过一段周时间后，假设这项功能是稳定的。此时，在问题追踪数据库中可以创建一条记录，这条记录标识功能开关代码的去除将作为一项动作执行。然后，可以追踪功能开关代码的去除并与在问题追踪数据库中的其他条目一样设置优先权。

10.3.3　测量干系人的满意度

测量干系人满意度的一种方法是通过让干系人填写简短的调查表。另一种方法是识别破坏性的事件，这是基于破坏性导致不满意的假设。

- 简短的调查表。可以要求内部干系人把他们的满意度按照 1 ～ 5 来打分。不同类型的干系人有不同的关注点，因此可以针对干系人的特定类型定制调查问题。应该让干系人知道他们的打分是从他们的视角进行的。例如，Dev 和 Ops 人员可以对监控信息的有效性进行评论，Ops 人员可以对为变更脚本而制订的配置管理策略发表评论，业务干系人可以对从提出设想到部署生产环境这个整体周期时间发表评论。
- 危险。任何过程改进努力的目标之一是就实际执行这个过程的人员来说，要去除对英雄主义工作的必要性。在 DevOps 上下文中，英雄主义的工作是由故障或者由不合适的事件周期引起的。
- 故障。当故障发生时，通常没有时间对原因进行深入的分析。最初关注在减缓故障的影响上。然而，当系统已经恢复正常开始生产后，应该进行事后反思。如果这些故障的根本原因是 Dev 与 Ops 之间的交互问题、DevOps 实践的缺失或者未遵循 DevOps 实践，那么这些故障应该成为测量用户对推出 DevOps 的满意度的一种方法。理想情况下，以上列举的故障数量会随着使用 DevOps 实践的增加而减少。
- 不充足的事件周期。以下事件可能导致服务中断：安全审计、对现有系统推出变更或者安装补丁。与故障一样，这些事件引起的中断也应该成为测量用户对推出 DevOps 的满意度的一种方法。

10.4　Dev 和 Ops 之间的交互点

在 Dev 和 Ops 之间有许多交互点，这一点我们还没有进行细节的讨论。本节中的两点是许可[⊖]和事故处理。

10.4.1　许可

软件许可是对软件使用或者软件再分发进行管理的法律协议。我们关注是你的组织支付了费用或者应该已经支付了费用的许可，我们也关注这些许可对 DevOps 实践带来的影响。我们首先描述许可是如何工作的。

许可可以视为一个令牌，这个令牌提供了对被许可的软件包的访问。许可可以是企业范围的或者用于指定数量的应用程序或者用户的。许可是为被许可软件包的特定版本

⊖ 这里指软件许可。——译者注

颁发的。这些许可可以有具体的过期时间，也可以没有。

许可是由被许可的软件包动态验证的。在被许可的软件包初始化的某个点上，它检查是否已经给它提供了一个有效的许可。许可可以放在已知的位置、由应用程序提供，或者更常见的一种情况，由许可服务器提供。许可服务器位于网络上，它存储了许可许少并且这些许可提供对外访问。

我们识别 3 个场景，其中，Dev 和 Ops 都涉及与许可有关的问题。

1）过期。如果一个应用程序的许可已经过期了，那么这个应用程序就不能再运行了。通常，更新许可是 Ops 的责任，应该有一套过程来检测即将过期的许可并确定应该更新哪些许可。然而，在这个过程中的任何懈怠都会引起应用程序的故障并将该故障报告给 Dev。应用程序应该有一个具体的故障消息来识别哪个软件系统的哪个版本不能获得许可，这样你就不需要花费额外的时间来判断是什么原因导致了应用程序发生故障。

2）许可不可用。有些许可是“不固定的许可”。也就是说，应用程序实例的最大数可以同时使用许可的软件包。如果创建一个使用了超过最大数的授权的软件包的新虚拟机，那么它将失败。同样，它应该在失败时输出一条明确的错误消息。此时，问题就转移到了 Ops 上。Ops 应该能够判断哪些运行中的虚拟机正在使用了哪些许可，这些虚拟机是否仍然是需要的。

3）软件审计。在极少的情况下，软件厂商可能要求对组织进行审计以确保他们的软件副本都是适当许可的。有些组织执行他们自己的软件审计来确保他们遵从了他们的所有许可要求。Ops 可以执行这种审计，但是他们需要能够在任何时间点上枚举所有正在执行中的虚拟机，而且能够判断在那些虚拟机包含了哪些被许可的软件包以及那些软件包在当时是否经过了正确的许可。如果 Dev 维护了运行中软件的可追踪性（因此可以知道运行中的虚拟机所包含的组件以及这些组件的历史信息），那么这些需求是可以满足的。许可服务器通常保存了使用它们许可的虚拟机的启动和停止的历史信息，因此检测到失败的虚拟机的一个结果应该是更新对应的许可历史信息。

10.4.2　事故处理

Dev“把发布包隔墙递出”给 Ops，Ops 负责把应用部署到生产环境。这种实践带来的好处之一是，事故的角色和职责是明确的：Ops 处理出现的任何涉及那个应用程序的事故；如果他们无法处理这个问题，那么通过升级过程，将 Dev 包含进来。一旦 Dev 控制了部署过程，那么这种情况就变得更加复杂了。

一旦事故发生了，有 3 个可能的情况：

1）事故显然与应用有关。这种情况下，Dev 组是管理该事故的初始联系人。在第 1 章中，我们讨论了可用性工程师的角色。该角色应该对事故引起的任何活动负责。

2）事故与硬件或者基础设施故障有关。这种情况下，Ops 负责诊断和修复故障。这种事故可能由 Dev 引起，然后 Dev 把该事故移交给 Ops，但是这是否是一个选择取决于

你的组织的决定。

3）事故的原因是不明确的。举个例子，假设网络明显地慢了。这可能是由硬件和基础设施引起的，也可能是因为一个应用错误地洪泛了网络。其他的例子可能更加微妙，例如我们在第 2 章中讨论的有关长尾效应的例子。第一个挑战是诊断问题的原因。一旦诊断完成，修正这个错误的职责又把我们带回到了刚刚前面的两个案例。从根本上，你的组织需要有一个明确的升级策略，这个策略回答以下问题：谁负责问题的初始检查？多长时间后这个问题应该升级？问题应该升级给谁？

10.5　小结

实施 DevOps 实践需要管理层认同，相应地，需要一个领军人物来说服管理层 DevOps 实践是有益于组织的。说服管理层采用一项新技术实践的通常方法是通过创建业务案例。DevOps 实践的业务案例包括成本、收益、风险和风险减缓、推出规划和成功标准。

一旦 DevOps 实践开始了，那么重要的测量是采用这项实践的成功度、与相关实践的合规性以及干系人对这些环境变化的满意度。

另外，Dev 和 Ops 必须在组织内进行交互，以解决软件许可和事故处理的问题。

10.6　更多阅读材料

关于业务考虑，读者可以在如下的网站获得更多信息：

- 博客“DevOps Considerations”：http://techopsexec.com/2013/09/10/devops-considerations/。
- 《Communications Networks in R&D Laboratories》[Allen 70]。
- 关于变更管理的维基百科的条目：http://en.wikipedia.org/wiki/Change_management#Managing_the_change_process。

与软件开发成熟度模型一样，你可以在参考文献 [InfoQ 13] 中找到关于 DevOps 的成熟度模型的更多信息。

Damon Edward 的文章 [InfoQ 14] 对于更好地理解网络测量很有帮助。

第四部分

案例研究

在这一部分中，我们将描述3个案例研究，目的是强化在前面章节中提出的观点。全部的案例研究都来自积极参与了实施DevOps实践的组织，而且每一个案例都被选中以用例说明DevOps的一个特定方面。

为了实现业务连续性，对很多组织来说，维护多个数据中心是一个可选择的方案。这样的选择要求数据中心不仅与它们维护的数据同步，而且还与每个数据中心内安装的软件和硬件同步。在第11章中，来自Rafter的Chris Williams将为我们分享如何实现两个数据中心的同步。

很多企业想采用DevOps实践，但苦于没有这方面的专门知识而无法直接实施。在第12章中，John Painter和Daniel Hand给我们讲解了咨询公司Sourced Group是如何指导企业来实现DevOps的。

在第4章中，我们倡导使用微服务作为一种提高部署速度的方法。大多数组织中存在着一些旧系统。为了把这些旧系统转换到微服务架构上，我们必须对它们进行重新架构。在第13章中，来自Atlassian的Sidney Shek将给我们说明如何实施一个这样的微服务。

第 11 章　支持多数据中心

Chris Williams

Rafer 正在使得学习资料（和高等教育一起作为一个整体）对于学生来说更加能够承担得起。

——http://www.rafter.com/about-rafter/

11.1　概述

很多年以来，学生一直为高昂的新教材费用而苦恼，但这些新教材在使用后却变得一文不值。Rafter（原来为 BookRenter.com）把这个现象作为一个商业机会，并且在 2008 年开创了一项向学生租赁教材的业务。这项业务的前提很简单：学生确定好新学期所需要的教材，然后从 BookRenter 网站上订购自己需要的教材。Rafter 将被选中的书籍运送到学生手中。在学期末，租赁了书籍的学生把书籍归还给 Rafter，因此这些书籍在下一个学期又可以被其他学生使用了。

从上面的描述中，你可能已经推导出来了：这个租赁业务是季节性的。如果在学期之初 BookRenter.com 发生了宕机，那么客户就流失了。对 Rafter 来说，在每年的高活跃时间段内确保业务连续性是十分重要的。基于此，Rafter 实施了多个举措，其中最重要的一个举措就是为了冗余性而并行运行了两个数据中心。

因为有两个数据中心，所以 Rafter 不但具有了在主数据中心发生故障时进行服务迁移的能力，而且维护着一个完全复制了生产环境的测试站点。在两个数据中心之间保持同步提出了以下的挑战：不但要保持数据同步，而且要保持环境是复制的，并且同时还要确保应用是被合理架构的。在本章中，我们将探索 Rafter 是如何完成这些不同形式的同步的。

有两个基本用例存在于迁移服务请求从一个数据中心到另一个数据中心的过程中：受控的和非受控的。受控的迁移是指，主数据中心依然可用，在切换到辅助数据中心前有时间进行各种准备措施。这种类型的迁移用来测试切换过程中涉及的各种措施，也用于为主数据中心的维护提供可能。非受控的迁移作为一个灾难（不管是自然的还是人为的）的结果而发生。我们在描述了 Rafter 实践中的解决方案后将回到本节提到的这些用例。

11.2　当前的情况

Rafter 当前运行了两个具有完全相同的硬件的数据中心。它们运行它们自己的数据中心，因为在做出多个数据中心决定时，Rafter 不能从公有云上获得所需要的输入 / 输出性能。这个决定当前正在被重新评估。虽然 Rafter 可以选择切换到公有云上，但是由于监管原因，要求很多组织运行它们自己的数据中心。这两个数据中心分别位于北美大陆的两端，但是用户却察觉不到响应时间上的差别。每个数据中心拥有大概 300 台虚拟机（它们使用 VMWare）。一种典型的虚拟机配置是 16GB 内存和 4 个虚拟 CPU 内核。前端层吞吐量平均大约为每秒 30 000 ～ 50 000 个请求。工作负载中读和写的比例大致相当，其中 80% 的请求来自应用程序编程接口（API），剩下的 20% 的请求来自浏览器。另外大约有 150 台虚拟机用于分析和预发布 / 测试目的，这些虚拟机的部分运行在本地私有云（Eucalyptus）上，一部分运行在公有云（Amazon Web Services）上。因为这些服务器上的服务级别协议（Service Level Agreement，SLA）比 Rafter 数据中心内的服务器上的宽松，所以实时灾难恢复不是必需的。

Rafter 有大约 35 个 Ruby on Rails 应用，大约 50 个后端应用是用 Ruby 编写的，两个应用是用其他编程语言（Clojure 和 R 等）编写的。Rafter 使用了标准的 3 层架构——Web 层、业务逻辑层和数据库层。我们首先将介绍业务逻辑层，然后介绍支持的数据库层。

11.3　业务逻辑和 Web 层

我们讨论与业务逻辑和 Web 层相关联的两个方面。第一个方面是应用的逻辑，第二个方面是 Rafter 用来支持这些应用的基础设施。

11.3.1　应用逻辑

如果一个应用需要存储在多次请求之间保持持久的状态，那么它就必须使用以下两种数据存储方案中的一种：可以切换的数据存储（如 Rafter 的 SQL 数据库）或者对这两个数据中心都是外部可用的资源（如 AWS S3）。这种限制甚至可以应用于负载均衡环境中的一个数据中心。如第 4 章所述，在本地服务器上存储应用状态不是一个好的实践，因为需要这些数据的后续请求可能发送到没有包含这些数据的另一台服务器上。Rafter 遵循了在数据库层（外部的或者复制的）保持应用状态的实践。

另外，应用配置更改对于支持数据中心切换不是必要的，因为所有的外部资源都是通过域名系统（DNS）主机名来访问的，这些主机名在切换过程中并不发生变化。

每次 Rafter 部署一个应用程序的新版本时，它将同时部署到两个数据中心。这种部

署方法确保了这两个数据中心运行应用程序的相同版本。部署系统使用基础设施库来计算将应用程序部署到哪些服务器上，因此在部署系统上不需要维护服务器的独立列表。Rafter 通过这种方法实现了我们在第 5 章中讨论的可追踪性。

因此，在将新版本部署到辅助数据中心时的一个问题是，物理服务器以及应用虚拟机可能是离线并且对部署不可用的。解决这个问题的方法是，在 Chef 服务器上存储每个应用的最新版本的信息。一旦运维团队完成了他们的维护工作并再次启动虚拟机，Chef 将检测到服务器上的应用是过时的版本，并将在这些服务器上部署应用的最新版本。这种方案确保了应用代码在不同虚拟机之间保持同步，即使在每次代码部署时虚拟机不总是处于可用状态的情况下。

11.3.2 基础设施

基础设施以库的形式提供支持。这个库作为 gem 打包在 RubyGems 中。RubyGems 是 Ruby 语言的包管理器，它提供了一种分发 Ruby 程序和库的标准方法。这个库用于很多基础设施有关的应用中，为增加新应用和发现有关基础设施的信息提供了一个框架。我们将讨论这些方面以及保持同步。

11.3.3 增加一个应用

在 Rafter 平台上的每一个应用都有一个小的 JavaScript 对象表示法（JavaScript Object Notation，JSON）文件，这个文件作为蓝图包含了如何在基础设施上准确地安装这个应用的指令。这些 JSON 格式文件中的一些属性的例子如下所示：

- 应用的名称
- 类型
- Git 存储库
- 主机名
- 定时任务
- 守护程序
- 日志轮转
- 防火墙规则
- 数据库授权
- 安全套接字层（Secure Sockets Layer，SSL）证书
- 负载均衡器虚拟 IP

基础设施库可以在 Chef 中运行，读取这些 JSON 文件并决定如何设置应用程序。这个库包含了平台的许多默认配置（如果需要可以重写），所以这些 JSON 文件不至于过大。

举个例子，设想有一个名字为“cat”的简单的 Ruby on Rails 应用程序，它的代码位于 GitHub 上的存储库中。那么这个应用程序的 JSON 文件看起来可能是这样的：

```
{
  "id" : "cat",
  "repo_url" : "git@github.com:org/cat.git",
  "type" : "rails"
}
```

在 Chef 服务器上，这个“cat”应用程序通过角色属性或者节点属性分配给一台虚拟机（或者一组虚拟机）。然后，Chef 在这台虚拟机上运行并使用库来查询这个“cat”应用程序的最新 JSON 文件，应用适当的设置步骤，如下所示：

1）检查来自 GitHub 的“cat”应用程序。

2）将“cat”应用程序部署到本地虚拟机。

3）用正确的虚拟主机为 cat.rafter.com 上的应用程序设置 nginx 和 unicorn（Ruby on Rails 应用程序服务器），并设置正确的 SSL 证书。

4）为应用程序设置默认的日志轮转。

这仅仅是一个简单的例子，但是库可以处理更复杂的应用程序设置，如安装定时任务（cronjob）和守护程序（daemon）、创建数据库账户、管理数据库授权、控制哪些开发人员对虚拟机上的应用程序有访问权限、处理开发和预发布环境等。库也可以为不同虚拟机集合上的应用程序创建不相同的层。例如，一个应用程序可以设置为在前端虚拟机的一个集合上只提供 Web 流量，在虚拟机的另一个集合上只运行定时任务和后端守护程序，这样这个应用程序的两个实例之间就不会竞争资源。

11.3.4　发现基础设施

Chef 将有关整个基础设施的信息存储在 Chef 服务器上。Chef 服务器对外提供一组应用程序编程接口，用于查询这些数据和存储有关基础设施的附加信息（通过称为数据包（data bay）的一种基于 JSON 的文档）。Rafter 库使用 Chef 服务器作为有关基础设施信息的数据库。

这里有几个例子来说明基础设施库的使用。注意，其中一些变量在它们定义后在例子中重复使用。

- 通过以下调用来列出整个基础设施上的所有应用程序：

```
DevOps::Application.all
```

- 获得关于一个特定应用程序的详细信息（从它的 GitHub 存储库到它所使用的数据库等全部信息）：

```
myapp = DevOps::Application.load("myapp")
myapp.repo_url
      >git@github.com:org/repo.git
myapp.application_database
      >myapp_production
```

- 寻找出应用程序部署的第一个虚拟机：

```
node = myapp.nodes.first
node.name
>web01
```

- 获得关于虚拟机的详细信息，如托管这个虚拟机的数据中心：

```
node.datacenter.name
>dc1
```

- 获得数据中心的状态：

```
node.datacenter.active?
>true
```

不同的应用程序使用这个库完成不同的任务：

- 当 Chef 正在虚拟机上运行以便应用配置时，广泛使用这个库。需要应用程序设置哪些定时任务、设置哪些守护程序、添加哪些防火墙规则等，都由这个库来决定。例如，下面是 Chef cookbook 中的一个片段，它根据虚拟机所在的数据中心的状态为虚拟机添加一条防火墙规则：

```
if node.datacenter.active?
     iptables_rule "block_api" do
               enable false
     end
else
    iptables_rule "block_api"
end
```

- 部署系统使用这个库发现应该将特定的应用程序部署在哪些虚拟机上。如果虚拟机位于一个特定的数据中心内，那么可能需要做特殊的考虑。例如，两个这样的部署限制是：
  - 在部署时，不要从非活跃的数据中心执行数据库迁移，因为这个过程会非常慢。
  - 跳过部署到离线数据中心的任何虚拟机。
- 测试和预发布环境使用基础设施库获得关于可以部署到测试和预发布虚拟机上的所有应用程序的信息，并判断应用程序之间的依赖关系。

保持基础设施同步

辅助数据中心采用与主数据中心完全相同的方式设置。它包含了与主数据中心完全相同数量的虚拟机，预计所有的应用程序虚拟机都是完全相同的配置。所以，在主数据中心创建一台新虚拟机时，同时在辅助数据中心也创建了一台相同配置的虚拟机。Chef 保证在两个数据中心的所有虚拟机都以相同的方式配置。当然，在两个数据中心之间有

一些微小的配置差异，这些差异是命名导致的（不同的 IP 空间和域），但在其他方面，它们是完全相同的。每当将一个配置变更推送到 Chef 时，将该变更应用于两个数据中心的虚拟机上。

11.4　数据库层

虽然复制位于上层的无状态虚拟机相对简单（完全复制是目标），但是对于数据库层来说，这个就有点儿复杂了。在该层上，使用了 3 种不同的数据库，每一种数据库用于一个特殊的目的。

11.4.1　事务数据

Rafter 的大部分数据存储在一个事务型、符合 ACID 的、SQL 支撑的数据库中，该数据库叫作 Clustrix。Clustrix 是一个集群化的数据库应用，是 MySQL 的一个简单替代。每个数据中心有 3 个 Clustrix 节点，虽然这 3 个节点的行为作为一个数据库。大约 0.5TB 的数据存储在这个数据库中，通常每天的数据变化量为数 GB。

每个数据中心的 Clustrix 数据库采用主主复制模式进行配置。然而，在辅助数据中心的数据库有一个对它们进行设置的 READONLY（只读）标志，因此没有应用程序可以意外地将数据写入这些数据库——如图 11-1 所示。Rafter 的目标是让这两个数据中心之间的复制延时尽可能地小。大部分时间在 1 秒以内，但这是一个需要严密监控的指标，因为应用程序变更可能带来增加延迟的新的查询或者数据约束。

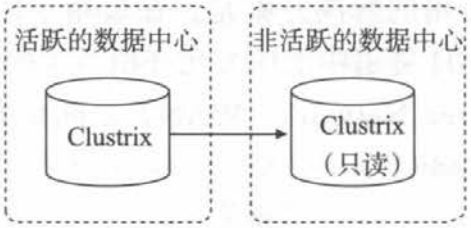


图 11-1　采用主主复制模式的 Clustrix 数据库管理系统（DBMS）[标注法：架构]

这种形式的复制与标准 MySQL 复制有相同的痛点。这种复制是单线程的和异步的，所以传入的写请求的复杂性和速率都可能导致辅助数据库落后于活动数据库。幸运的是，有许多工具可以用来处理这些问题，并且数据库厂商每年都会提出一些新的解决方案（例如，混合复制、多线程从库、从库预取等）。在解决复杂的请求方面，Rafter 使用的一种方法是重写请求这样它可以更快地运行，或者将它分解为更小的数据块（例如，LIMIT 10 000）。另一种方法是切换到基于行的复制（Row-Based Replication，RBR），这样从库不需要执行复杂的请求且可以简单地更新任何更改的行。处理大量写比较难办。一个特定的应用程序负责总写入量的一半。这个数据不是实时应用程序数据所需要的。因此 Rafter 将这个应用程序改为把数据写入平面文件（flat file）而不是写入 Clustrix 数据库中。一个专用应用程序以小时为单位收集这些被写入的文件，然后将它们导入数据仓库（数据仓库的服务级别协议与生产环境基础设施不同）。虽然现在还没有使用，但是 Clustrix 还

支持对不同的数据库设置多个复制流，这样它就成为处理高写入量的另一个可选工具。

11.4.2 基础设施支持

Redis 是一个键值缓存和存储。Rafter 在它们的基础设施中使用 Redis 分别作为作业排队系统（Resque 和 Sidekiq）、缓存、快速的键值存储数据库。每个数据中心有两个独立的 Redis 集群——设置如图 11-2 所示。其中一个 Redis 节点作为“主”Redis，其中所有应用程序都向其发送它们的读和写请求。在发生故障时，主 Redis 可以切换到任意一个从 Redis 上。随着 Redis 使用的增加，因为发送的写请求的速率增加，所以从 Redis 跟不上主 Redis。Rafter 开始注意到这个延迟是因为主服务器上的内存在逐渐增加，而 Redis 中存储的数据却没有增加。Rafter 经过分析发现，Redis 在内存中缓冲了从服务器的所有挂起的命令，但是因为从服务器跟不上复制的速度，所以挂起的命令不断增长。其他组织也分析了这个问题。根本问题是，Rafter 在两个数据中心之间生成了比可用带宽更多的 Redis 流量。可用的解决方案是，在启用了压缩功能的 SSH（安全 Shell）隧道中运行 Redis 复制。在 SSH 隧道中的压缩允许相当少的带宽（以大约 20% 的 CPU 利用率增加）在广域网（Wide Area Network，WAN）之间传输数据。这个改进使得到从 Redis 的复制能够跟得上主 Redis。

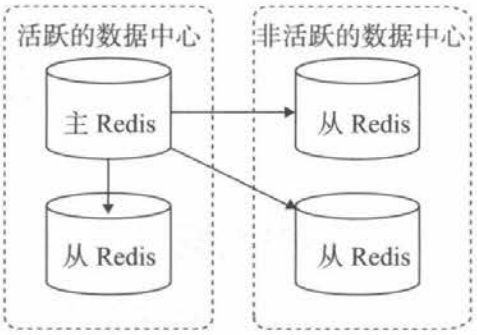


图 11-2 Redis DBMS[标注法：架构]

11.4.3 会话数据

Rafter 使用 Memcached 的 Couchbase 实现来存储会话数据并在整个平台上进行缓存。这是平台上的一个数据库的例子，其中数据在数据中心之间是不同步的。图 11-3 说明了数据中心内的 Couchbase 的表示。两个 Couchbase 集群是完全独立的，因此当辅助数据中心被激活时，缓存将过期。这不会导致问题，因为这些数据要么是临时数据（会话数据），要么是在缓存未命中时应用程序自动从其他数据源重新填充的缓存数据。

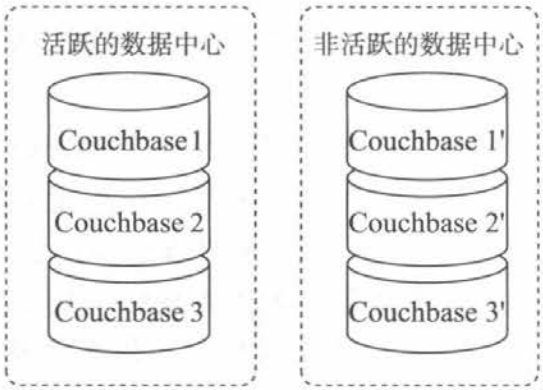


图 11-3 Couchbase（Memcached）[标注法：架构]

11.5 其他基础设施工具

Rafter 基础设施中使用了许多其他的工具，包括 gem 存储库服务器和 Elasticsearch。

在从一个数据中心故障切换到另一个数据中心时，DNS 的管理也是一个重要的元素。可以参考第 2 章中有关 DNS 管理和存活时间（Time To Live，TTL）的内容。我们将在下面讨论 Rafter 对 DNS 服务器的使用。

11.5.1　gem 存储库服务器

Ruby 应用程序使用“gems”作为一种打包外部库的方式。典型地，Rafter 把它们的应用程序之间的共享代码打包成 gems。特定的 gem 和它们的版本绑定为应用程序内部的依赖。Rafter 在两个数据中心分别构建了一个 gem 存储库集群作为发布私有 gem 的地方。图 11-4 说明了 gem 存储库服务器的表示。Rafter 从一个名为 Geminabox 的开源 gem 服务器开始，它为开发者提供了一个简单的用户界面（UI），以便将 gem 的新版本上传给服务器。然而，它没有提供对高可用性的支持，并且简单地使用本地文件系统作为存储 gem 的存储库。为了构建高可用性系统，Rafter 首先在每个数据中心创建多个 gem 服务器，并把这些服务器放在负载均衡器的后面。但是，这不能自动同步所有 gem 服务器之间的数据。以后 Rafter 创建了在每个 gem 服务器上运行的脚本。这个脚本同步底层 gem 存储库与它的两个邻居（一个在本数据中心，一个在另一个数据中心）。在底层，他们使用了 Unison，一个类似于 rsync 的开源双向文件同步工具，它每分钟同步 gem 存储库文件。他们还创建了一个监控脚本来查询每一个 gem 服务器并在发现任何不一致时发出警告。

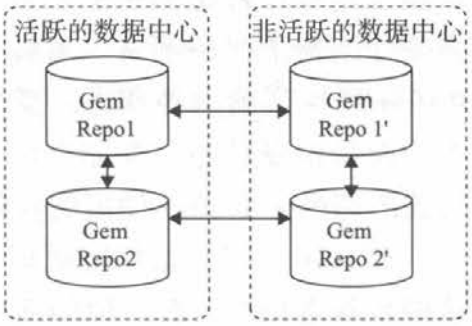


图 11-4　Gem 存储库服务器 [标注法：架构]

11.5.2　Elasticsearch

Rafter 有一个 6 节点 Elasticsearch（一个搜索和分析引擎）集群，这个集群横跨两个数据中心。Elasticsearch 的分片（对数据进行水平划分）分配功能可以保证所有的分片都可以完整地复制到辅助数据中心。也可以将 Elasticsearch 训练为数据中心感知的，所以对于请求，它更倾向于相同数据中心内的节点。当 Rafter 第一次部署 Elasticsearch 时，遇到了数据中心之间的连接性问题，因为 Elasticsearch 集群功能需要所有集群节点之间是稳定的、低延迟的链路。Rafter 解决这个问题的方法是对一些 TCP 内核参数和 Elasticsearch 自己的超时设置进行调优，这样可以避免在其他数据中心的节点因为超时而不断离开和加入集群的问题。在广域网链路上运行 Elasticsearch 集群的用户目前还不是很多，因此从某种意义上这是一个新的领域——但是除了以上提到的易于解决的问题点以外，Rafter 迄今还没有遇到过任何重大的问题。

11.5.3　域名系统

在每个数据中心内，Rafter 维护了两个 DNS 服务器来提供本地域名服务。一个 DNS

服务器作为其他 3 个服务器的主服务器。所有 DNS 服务器都是运行 Webmin 管理界面的标准 BIND DNS 服务器。Webmin 提供了一个更新和删除 DNS 记录的简单的基于 Web 的用户界面，并提供了一些更高级设置项，例如提升一个 DNS 从服务器为主服务器或者修改一个从服务器的指派的主服务器。Rafter 创建了一个与这个用户界面交互的 Ruby 库，这样实现 DNS 变更的自动化。Rafter 为大部分的 DNS 记录设置了 60 秒的存活时间，为指向数据库虚拟 IP 的 DNS 记录设置了 1 秒的存活时间。这些时间的选择是一种对变更的速度与由在短时间间隔为查询 DNS 引起的因特网流量之间的权衡。

11.6 数据中心切换

数据中心可以处于以下 3 种状态之一：活跃的、非活跃的、离线的。活跃的意思是，在这个数据中心内的服务器上没有任何限制。非活跃的意思是，定时任务、守护程序和后端应用程序都是停止的和被禁用的。离线的意思是，数据中心是非活跃的，并且将不接受任何新的代码部署。离线模式通常应用于数据中心的维护。在进行数据中心切换时 Rafter 利用了前两种状态。受控切换用于维护或测试目的。在受控切换过程中，主数据中心保持活跃状态。在非受控切换中，主数据中心是非活跃的。我们下面将对在这两种形式的切换中 Rafter 使用的步骤进行说明。

在 Rafter 刚开始执行数据中心切换时，其中的一些步骤是手动执行的，没有过程把这些步骤连接在一起。他们简单地在 Word 文档中写下所有步骤，然后一步一步地执行。当它们在这个过程中感到自信时就开始自动化这些操作。Rafter 从创建一个 Ruby 库开始，这个库使用了领域特定语言（Domain Specific Language，DSL）来定义切换过程中的步骤。在讨论了受控和非受控的切换步骤后，我们将阐释自动化方案。

11.6.1 受控切换步骤

一个受控切换过程包含 18 步，从最初检查临时禁用监控警告到切换所有层到另一个数据中心。为了说明这些步骤的目的，假设以下的缩略语表：

- DC1：从这个数据中心切换出来。
- DC2：切换到这个数据中心。

1）确认 Clustrix 数据库有低复制延迟。首先检查复制的延迟在 60 秒内。这样做是有原因的。曾经有一次在数据中心的切换过程中，Rafter 把所有的站点都关闭了以后，才发现他们的数据库复制由于一个大的查询而存在 30 分钟的延迟。在继续那次切换前，他们不得不等数据库复制赶上之后才进行，这样就导致了更长的停机时间。

2）在监控系统（Scout）上禁用警告。Scout 是一个类似于 Nagios 的监控系统，它被 Rafter 用来监控服务器（CPU、磁盘、网络、URL 健康检查等）。Rafter 禁用警告的目的是防止在切换过程中获取传呼和邮件。

3）展示“网站即将关闭”的横幅。Rafter 在他们的网站上放了一个小的横幅来告诉用户网站将在 10 分钟内关闭维护，提醒他们尽快完成购买流程。部署系统通过在每个应用服务器上的某个目录中放置一个特殊文件的方式来实现自动处理。

4）在 Chef 服务器上将 DC1 标记为“非活跃的”。因为 Chef 服务器用作一个部署数据库，所以将一个新的 JSON 文档推送到 Chef 服务器，标记 DC1 数据中心为“非活跃的”。现在使用了基础设施库的客户端将能够知道这个变化。这时在切换过程中，DC1 和 DC2 都处在非活跃状态。

5）启动 DC1 上的所有虚拟机上的 Chef 客户端。Chef 提供了一个名为“knife ssh”的命令，它能够在匹配一个特定搜索的一组虚拟机上并行执行命令。命令：knife ssh “roles:dc1”“sudo chef-client”启动 DC1 上的所有虚拟机上的 Chef 客户端。

因为 Chef 使用了基础设施库，所以它将检测它运行所在的虚拟机是非活跃的数据中心——因此，Chef 客户端执行适当的步骤来重新配置虚拟机。这样做的结果是去除任何应用有关的定时任务并停止 / 禁用任何后端守护程序。

6）发送“TERM”信号到 DC1 上正在运行的任何定时任务或者后端脚本。大量应用程序通过 cron 在后端运行，通过发送 TERM 信号来终止它们。这一步是通过类似如下的命令来实现的：

```
knife ssh "roles:dc1" "ps hww -C ruby -o pid,user,cmd
| grep app_user | awk '{print \\$1}' | xargs -i kill
-TERM {}"
```

7）向在 DC1 上拒绝停止运行的任何顽固脚本发送“-9”信号。有时，TERM 信号不足以使该脚本停止。例如，一个应用程序可能会错误地捕获所有异常（包括 TERM 信号）。迄今为止，已经修正了 Rafter 应用程序中的大部分有问题的脚本，但这一步骤保护了一些新的写得不好的应用。

8）在有外部用户的网站上建立一个维护页面。Rafter 会建立一个带有类似如下内容的维护页面：“我们正在执行列入计划的维护，网站将在 15 分钟后恢复可用”。部署系统通过将一个特殊文件放入每个应用程序服务器上的一个目录中自动完成这个工作。在两个数据中心都执行这个步骤。

9）在 DC2 中提升一个新的主 Redis 服务器，默认情况下，Redis 没有任何自动切换技术，所以 Rafter 不得不建立他们自己的系统来实现这个功能。操作过程如下所示（见图 11-5）：

a. 配置所有的应用程序以便连接到主机名为“redis-master01”的服务器。这个主机名解析为数据中心内的一个本地 IP 别名。

b. 从当前主 Redis 服务器上释放这个 IP 别名，这将切断当前任何已经建立的 Redis 连接。然后将当前时间作为一个心跳键值写入主 Redis 服务器，在要提升为 DC2 内的从 Redis 服务器上检查这个键值是否有相同的值。这能够保证从 Redis 服务器的复制已经跟

上了主 Redis 服务器。

c. 将选择的 Redis 从服务器提升为主服务器，将原来的 Redis 主服务器降级，重新配置所有的 Redis 从服务器使用新的主服务器。

d. 将 IP 别名绑定到新的主 Redis 服务器。发送一个免费地址解析协议（Address Resolution Protocol，ARP）消息来保证刷新所有的 ARP 缓存。

e. 更新 redis-master01 的 DNS 记录来解析新的 IP 别名。在这个方案中依赖 DNS 是关注这个解决方案的一个方面，因为客户端可以缓存这个信息。然而，Rafter 没有这样的问题。推断是因为在释放 IP 别名时客户端到旧主服务器的连接被中断了，所以这就强制 Redis 客户端库重新连接——只有那时 Redis 客户端才连接到新的主服务器。在这个 DNS 记录上设置 1 秒的存活时间。

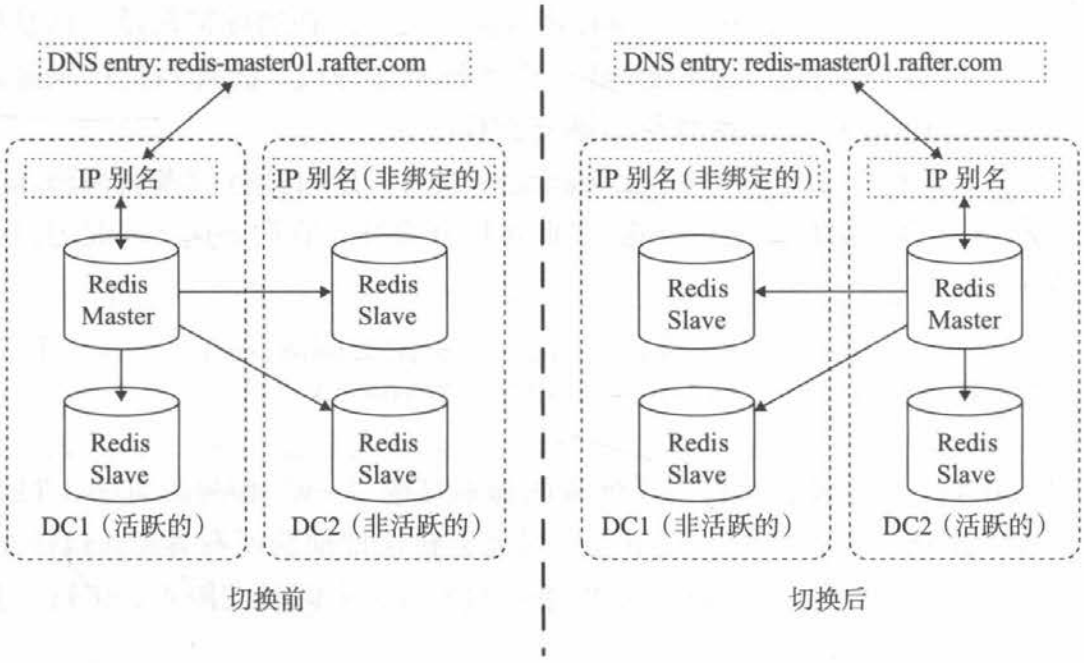


图 11-5　Redis 提升过程 [标注法：架构]

10）在 DC2 中提升一个新的 Clustrix 主服务器。因为在每一个数据中心的 Clustrix 数据库都配置为主主模式，所以在切换时不需要重新配置复制。Redis 也通过 DNS 来实现指向到当前活跃的 Clustrix 数据库。下面是详细的步骤：

a. 给在 DC1 内的 Clustrix 数据库添加一个 READONLY 标志。

b. 通过检查每个数据库上的二进制日志位置，确保两个数据库是完全同步的。检查 DC2 已经接收了所有的 DC1 的二进制日志，反过来也是一样的。

c. 从 DC2 Clustrix 数据库中移除 READONLY 标志。

d. 更新 Clustrix 主 DNS 记录来指向 DC2 中的 Clustrix 的虚拟 IP 地址。简单更新 DNS 记录是不够的，因为来自应用程序的连接将继续保持对 DC1 数据库开放。为了关闭这些连接，使用“show processlist”遍历 DC1Clustrix 上的所有连接，使用“kill”命令杀死所有打开的会话。这就强制应用程序打开一个到数据库的新连接，现在结束到 DC2

中的数据库的连接。

11）在 Chef 服务器上将 DC2 标志为“活跃的”。这遵循与步骤 4 相同的过程。此时，DC1 是非活跃的，DC2 是活跃的。

12）启动 DC2 中所有虚拟机上的 Chef 客户端。与步骤 5 类似，使用“knife ssh”命令启动 DC2 中的所有服务器上的 Chef 客户端。既然 DC2 处于活跃状态，那么 Chef 将执行所有必要的部署变更以便将虚拟机转换为活跃状态。从本质上来说，这意味着为所有后端应用程序安装定时任务、安装和启动应用程序守护程序、重新配置应用程序专用的防火墙规则。

13）在所有面向用户的 Web 应用程序中去除维护页面和横幅。这是由部署系统完成的。

14）更新 Akamai 内容分发网络（Content Delivery Network，CDN）。Akamai 作为一个提供了 CDN 服务的代理，它位于 Rafter 的主购物车应用程序的前端。使用 Akamai 提供的 SOAP 应用程序编程接口，更新 Akamai 以便将流量发送到 DC2。大约经过 5 分钟的时间，Akamai 将开始把流量发送到新的数据中心。

15）更新所有应用程序的公有 DNS。对于未使用 Akamai 的应用程序，使用 AWS Route 53 来更新公有 DNS 记录以指向 DC2 中的 IP 地址。因为对公有 DNS 有 1 分钟的存活时间，所以在这些变更后的几分钟内，大部分用户开始访问 DC2 中的应用程序。RubyGem 基础设施库在这个过程中起到辅助作用。这个伪代码看起来如下所示：

```
DevOps::Applications.all do |app|
  update_public_dns(app.fqdn, app.external_ip)
end
```

16）更新所有应用程序的本地 DNS。更新本地 DNS 服务器，这样内部服务的 URL 指向 DC2 中的 IP 地址。此外，基础架构库辅助：

```
DevOps::Applications.all do |app|
  update_local_dns(app.fqdn, app.internal_ip)
end
```

17）更新 Rafter 的监控系统（Scout）中的升级优先级。在监控系统中，将 DC1 中的服务器产生的警告级改为“正常的”，这意味着运维人员继续通过电子邮件接收来自这个数据中心内的服务器（但不是网页）的警告。将 DC2 中的服务器的升级优先级更新为“紧急的”。这意味着 DC2 中的服务器的重要警告可以打开 Pagerduty[⊖] 上的警告，结果就是发送出呼叫。Rafter 已经发现监控非活跃数据中心内的服务器依然是重要的，因为没有人希望在切换时才发现潜在的服务器问题。然而，使用基于电子邮件的警告而不是呼叫就已经足够了。该步骤是通过 Ruby 代码来实现的（该 Ruby 代码通过“Ruby

⊖ PagerDuty 是一家新兴的互联网创业公司，它开发出一款能够在服务器出问题时发送提醒的产品，包括屏幕显示、电话呼叫、短信通知、邮件通知等。——译者注

mechanize”库与 Scout 网站交互）。

18）重新启用 Rafter 监控系统（Scout）上的警告。在更新了优先级后，仍然需要重新启用警告。

11.6.2 非受控切换

在当前活跃数据中心完全不可用时（例如，由于数据中心内停电事故等）就会产生对非受控切换的需求。这种类型的切换有较少的步骤，因为彻底关闭活跃的数据中心是不可能的。在这种情况下，有很大可能发生数据不一致。很可能，在原始的活跃的数据中心恢复在线后，需要做一些数据整理的工作。即使只有 1 秒的数据库复制延迟，也有可能在发生故障时使非活跃的数据中心丢失一些数据。

与前面讲的受控切换一样，我们假设以下的缩略语表：

- DC1：从这个数据中心切换出来。
- DC2：切换到这个数据中心。

实施非受控切换需要执行的 9 个步骤如下所示：

1）在 Chef 服务器上将 DC2 标记为“活跃的”、将 DC1 标记为“非活跃的”。发布一个新的 JSON 文档到 Chef 服务器以生效。现在使用了基础设施库的所有应用程序都能发现数据中心的正确状态是什么。

2）如果需要，将 DC2 中的一个本地 DNS 服务器提升为主服务器并重新配置从服务器。如果 DC1 包含了当前的主服务器，那么没有本地 DNS 变更可以处理。因此，DC2 中的一个 DNS 从服务器必须提升为主服务器，其他服务器应该成为这个新主服务器的从服务器。这可以通过 Ruby“ mechanize”脚本（该脚本使用了 Webmin 基于 Web 的用户界面）来实现。

3）提升 DC2 中的一个新的 Clustrix 主服务器。这里受控切换需要执行的子步骤如下所示：

a. 移除 DC2 中的数据库上的 READONLY 标志。

b. 更新 Clustrix 主 DNS 记录指向 DC2。

4）提升 DC2 中的一个新的 Redis 主服务器。这里受控切换需要执行的子步骤如下所示：

a. 将 DC2 中的一个 Redis 从服务器提升为主，并重新配置 DC2 中的其他 Redis 从服务器为新的主服务器的从服务器。

b. 绑定 IP 别名到新的主服务器。

c. 更新 redis-master01 的 DNS 记录。

5）更新所有应用程序的本地 DNS。这个步骤与受控切换中的步骤相同。

6）更新所有应用的公有 DNS，这个步骤与受控切换中的步骤相同。

7）更新 Akamai CDN。这个步骤与受控切换中的步骤相同。

8）启动 DC2 中的所有服务器上的 Chef 客户端。这个步骤与受控切换中的步骤相同。

9）更新 Rafter 监控系统中的升级优先级。这个步骤与受控切换中的步骤相同。

11.6.3　定义和自动化切换步骤

切换步骤分为两个用例：它们启动一个外部 shell 命令（如“knife ssh”命令）或者 Ruby 代码（如检查数据库延迟）。下面是受控切换中的步骤 1 的示例：

```
step "Check: Ensure replication delay is low on Clustrix" do
  prereqs :clustrix_databases, :datacenters
  ruby_block do
    result = nil
    puts "Checking Slave Delay in each datacenter"
    @datacenters.each do |datacenter|
      client =  Mysql2::Client.new(:host =>
@clustrix_databases[:vips][datacenter], :username =>
"root", :password => @clustrix_databases[:password])
      row = client.query("show slave status").first
      puts "Slave delay in #{datacenter}:
#{row["Seconds_Behind_Master"]}"
      if row["Seconds_Behind_Master"] > 60
        result = failed("Slave delay must be less than
60.")
      end
      client.close
    end
    result ? result : success
  end
end
```

在以上的步骤中，在数据中心切换过程中，在基础设施库自动运行 ruby_block（从第 3 行起）中的 Ruby 代码。使用提供的成功 / 失败帮助方法，这段代码返回失败或者成功。在数据中心过程中的每一步都有一个在 DSL 中定义的相应步骤。

DSL 也包含“prereqs”的概念，因为大多数步骤都需要一组通用的数据输入（例如，需要切换的数据中心、数据库信息和收集的凭证）。每一步都定义了它需要的先决条件数据而不是一遍一遍地收集相同的数据。这些先决条件数据定义在另一个 DSL 中。在数据中心切换脚本的开始，在任何步骤执行前，将代码插入以便查询实现先决条件所需要的信息。例如，clustrix_databases 先决条件看起来如下所示：

```
prereq "clustrix_databases" do
  if ENV['CLX_PASS']
     db_info = {:password => ENV['CLX_PASS']}
  else
     puts "Please enter the root password for the database:"
     db_info = {:password => STDIN.noecho(&:gets).chomp!}
```

```
    end
    db_info[:vips] = DevOps::Clusters.load("clustrix")[:vips]
    db_info
  end
```

接下来，为了匹配这两种类型的数据中心切换，扩展 DSL 以便将这些步骤逻辑组成列表。例如，下面是一个受控切换的步骤列表：

```
list "Controlled Switch" do
  step "Check: Ensure replication delay is low"
  step "Disable Scout Notifications"
  step "Put up 'Site Going Down Soon' message"
  step "Set current datacenter to inactive"
  step "Run chef in DC to swap from"
  step "Run killer with -TERM"
  step "Run killer with -9"
  step "Put up 'Maintenance' pages"
  step "Swap redis"
  step "Swap DB"
  step "Set new DC to active"
  step "Run chef in DC to swap to"
  step "Remove 'Maintenance' pages"
  step "Swap Akamai"
  step "Swap Route 53 DNS"
  step "Swap Datacenter DNS"
  step "Update Scout Notification Groups"
  step "Enable Scout Notifications"
end
```

对于列表中使用的每一步，库都寻找具有相应名称的步骤实现。

基于这种框架来定义步骤并以特定顺序为这些步骤分组，Rafter 创建了一个基于控制台的应用程序来交互式地运行这些列表。在启动时，它显示按照名字定义的所有列表。例如：

```
Pick a set of Steps to perform: <enter defaults to #1>
1. Controlled Switch
2. Uncontrolled Switch
Choose: 1
```

在控制台中，运维人员选择一个列表来运行。然后，用户界面显示这个列表中的所有步骤的名字并要求确认。它还支持跳过执行某些步骤或者如果需要从不同的步骤开始执行。

```
The following steps will be run:
1. Check: Ensure replication delay is low
2. Disable Scout Notifications
[...]
Press Enter if satisfied with order, or type in alternative
order of steps, comma delimited, ranges allowed:<enter>
```

接下来，控制台应用程序运行选中步骤的任何先决条件代码，然后开始运行每个步骤并在控制台展示这些步骤的输出。使用哪个“自动驾驶”（auto pilot）模式的程度取决于运维人员。运维人员可以运行每一步都要求“回车”以处理下一步，或者在前一步骤完成后通过进入下一步列表可以自动运行。在某个步骤失败的情况下，运维人员可以选择跳到下一步或者重试这个失败的步骤。

控制台应用程序既可以从开发人员的机器上运行，也可以在一台公用服务器（这个服务器让网络访问两个数据中心的服务器并通过因特网访问 Chef 服务器）上运行。所有命令都是使用运维人员自己提供的凭证（如 sudo 密码、SSH 密钥、Chef 密钥等）来运行的，因此即使未授权用户获得了对控制台应用程序的访问权，他仍然没有执行这些步骤所需要的访问级别。

11.7　测试

与其他过程和软件一样，必须对基础设施和切换控制进行测试。

11.7.1　数据中心切换应用程序

单元测试用 RSpec 编写，它涵盖了前面部分讲到的每个步骤、底层的 DSL 库和控制台应用程序。对代码和逻辑中明显的缺陷来说，这提供了一个很好的完整性检查。然而，不是每个方面都能完整地测试，因为到资源的连接（比如数据库和外部的命令等）在测试中是被限制的。为了帮助发现单元测试不可能捕获的缺陷，整个控制台应用程序默认以排练（dry-run）/ 调试模式运行。这个模式下，它执行过程中的每一个步骤但是跳过执行步骤中一些潜在的非安全的任务（例如改变状态等）。一个帮助（helper）方法“ debug?”用于在步骤中检查该步骤是否正在以调试模式运行。这使得可以在整个数据中心切换过程前以排练模式（dry-run mode）安全地执行。

11.7.2　基础设施测试

在数据中心切换过程中进行的大量繁重的工作实际上是在整个数据中心切换应用程序的外部并在重新配置服务器时由基础设施库和 Chef 来执行的。基础设施库和 Chef cookbook 不断变化，因此它们需要更严格的测试。为了确认一项变更没有中断现有的功能，运行使用 RSpec 的单元测试。除了执行单元测试外，Rafter 还建立了一套集成测试。这些测试首先在预发布环境服务器平台上启动一个全新的服务器，然后使用 Serverspec 来确认 Chef 和基础设施库是否正确地配置了该服务器。

11.7.3　持续交付流水线

所有对 Rafter 基础设施和数据中心切换应用程序的变更都会经过持续交付流水线

(使用 TeamCity)。任何新的变更提交给一个名为“测试”的分支，然后 TeamCity 在这个测试分支上运行语法检查、lint（代码风格检查器）和单元测试。如果全部通过了检查，那么 TeamCity 启动一个虚拟机，并在这个虚拟机上执行一套集成测试。最后，如果没有测试失败，那么 TeamCity 自动将变更合并到预发布分支，然后自动发布所有的基础设施变更到预发布环境。当每个变更在预发布环境工作正常后，将这些变更手动合并到生产环境分支，在生产环境分支上重复这些测试，如果没有测试失败，那么将新的版本自动发布到生产环境。

11.8 小结

有两个数据中心并且能够从一个数据中心平滑地故障迁移到另一个数据中心涉及应用程序、数据库系统和基础设施管理系统。在本章中，我们描述了 Rafter 是如何实现这种能力的。

在应用层，应用程序必须按照以下原则进行设计：

1）应用程序应该只在被认可的数据存储上存储状态。

2）在设计阶段，就必须考虑应用程序的数据存储需求。应用程序变更和新功能可能导致复制的巨大延迟，相应地会损害快速切换的能力。通常，可以做出一些让步。例如，如果数据对业务来说不是关键的，那么它可能就不需要复制到两个数据中心。另外，应用程序可以按照不同方式存储数据，这样可以减少数据量（例如存储聚合数据而不是每条记录）。

3）必须设计后端应用程序以便优雅地停止。大多数的 Rafter 后端应用程序能够在任何时间被安全地终止运行，但有一些应用程序如果在错误的时间被终止运行，那么它们可能导致数据不一致。例如，设想一个对用户收费然后把这个收费情况记录到数据库中的应用程序。如果应用程序是在收费完成但还没有将收费记录写入数据库中时被终止运行，那么用户可能在以后被再次收费。为了避免这个问题，创建对终止执行敏感的应用程序以拦截 TERM 信号（通过 Ruby 的 trap（陷阱）回调），只完成当前循环，然后优雅地退出。

在基础设施层，Ruby on Rails 和 Chef 是重要的工具，但是它们必须设计成数据中心感知的，通过建立管理基础设施和将各种应用程序部署到适当的数据中心的基础设施库来实现这一点。

在数据库层，一个核心组成部分是理解分布并且能够配置成支持分布式数据的数据库系统。Rafter 不得不做出一些特殊准备以便能够让这些系统正确地运行。

虽然在每一层都采用了恰当的架构，但是管理从一个数据中心切换到另一个数据中心包含了一系列虽然可以自动化但是必须以某个特定顺序执行的步骤。

11.9　更多阅读材料

在本章中，我们涉及了很多技术。你可以在下面这些链接上找到更多细节内容：

- Chef：http://docs.opscode.com/chef_overview.html
- Ruby on Rails: http://rubyonrails.org/
- Ruby Gems: https://rubygems.org/
- GitHub：https://github.com/
- Clustrix：http://www.clustrix.com/
- Redis：http://redis.io/
- Couchbase：http://www.couchbase.com/
- Memcached：http://memcached.org/
- Elasticsearch：http://www.elasticsearch.org/
- Unison：http://www.cis.upenn.edu/~bcpierce/unison/
- Scout：https://scoutapp.com/
- RSpec：http://rspec.info/
- Serverspec：http://serverspec.org/
- TeamCity：http://www.jetbrains.com/teamcity/
- Resque：https://github.com/resque/resque
- Sidekiq：http://sidekiq.org/
- Akamai：http://www.akamai.com/

虚拟 IP 地址（VIP）的讨论在维基百科网站：http://en.wikipedia.org/wiki/Virtual_IP_address。

Redis 复制问题的讨论在附录的参考文献 [3Scale 12] 中。

第 12 章　实施企业的持续部署流水线

John Painter 和 Daniel Hand

Sourced Group 是一个企业咨询组织，它致力于为企业带去基于云的解决方案架构和自动化带来的好处。

——http://www. sourcedgroup.com.au/

12.1　概述

在过去的几年内，企业已经越来越多地利用供应商（如 Amazon Web Services(AWS)）提供的云计算服务。通过迁移到云计算，企业通常期望实现两个主要成果：成本效率的提高；他们产品或者商业成果的敏捷性和（发布）速度的提高。云计算平台的一个核心特征是编程接口和自动化框架的广泛可用性。这些接口最初用于管理基础设施（如服务器和存储器），但很快演变为包括应用本身的部署和管理，现在它用于主要的持续部署系统，例如持续部署流水线（Continuous Deployment Pipeline，CDP）。如第二部分所述，持续部署流水线监控应用程序的源代码控制状态。当源代码的变更已经提交后，这些变更被持续部署流水线获取，构建和打包应用，然后在不同的环境和以不同的目标执行各种测试。一旦应用程序达到“生产环境就绪”状态，持续部署流水线调用特定的编程接口来部署和更新这个应用程序的副本。对基础设施和应用部署的自动化控制让团队能够关注应用程序代码而不是应用程序部署，导致实现了企业中的开发项目的敏捷性和速度目标。

Sourced Group 在与来自金融服务、媒体、电信业和航空业等领域的处于领先地位的澳大利亚企业的合作过程中创造和提炼了持续部署流水线参考架构。本章的案例研究将介绍这一架构。Sourced Group 是一个由具有金融服务背景的人士于 2010 年创立的企业咨询组织。当前，Sourced Group 的团队企业工程师可以分为两类核心技能领域：数据管理（数据库和数据仓库）；解决方案架构和自动化。

在本章中，我们讨论在企业内实施持续部署流水线的不同方面：组织背景、持续部署流水线本身以及安全是如何管理的。然后，我们将介绍 AWS 提供的高级概念和新服务。最后，我们将对本章进行总结。

12.2　组织背景

对于持续部署流水线项目来说，Sourced Group 通常通过中期战略咨询契约与企业合

作来设计和交付 CDP 框架给组织。任何这样的实施都必须适合客户的组织和它的文化。因为 Sourced Group 的契约是在有限时间内的，所以一个活动是识别和培训在契约结束后对持续部署流水线负责的人。图 12-1 展示了 Sourced Group 通常采用的组织结构。应该形成两个组：一个*持续部署适职小组*，它负责与开发人员的接口以便使他们的应用程序接入持续部署流水线中；一个*持续部署工程团队*，它负责设计和管理流水线及其组件。持续部署工程团队及其维护的工具一起组成了 CDP 卓越中心（Center Of Excellence，COE）。在一个小型组织中，持续部署*适职小组*和持续部署工程团队可能是一个团队。一种常见的情况是，开发人员本身就是持续部署*适职小组*的成员。在 Sourced Group 契约完成后，这些组继续对持续部署流水线负责任。

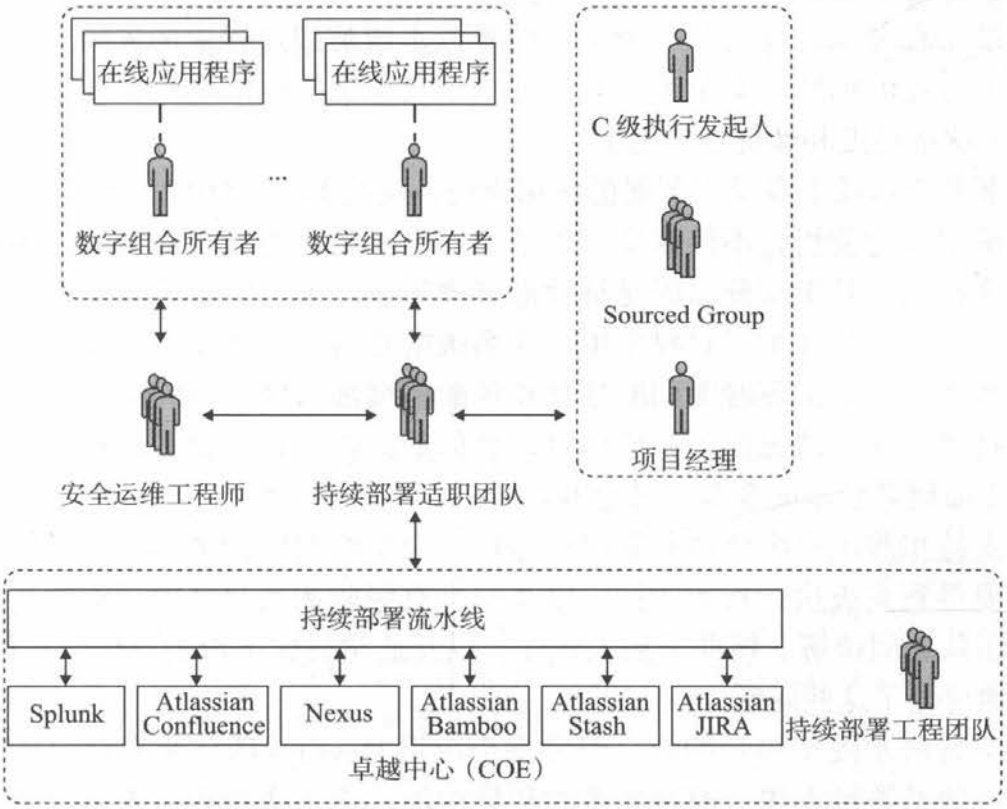


图 12-1　项目团队架构 [标注法：架构]

教育和知识转移是采用任何新技术取得成功的关键。为此，在契约的早期，Sourced Group 组织*适职小组*以包含虚拟的或持续部署工程团队的借调人员。*适职小组*帮助 DevOps 团队开发或将他们现有的应用程序迁移到持续部署流水线。除了平滑部署过程的迁移外，适职小组还向持续部署工程团队提供实时的反馈，这些反馈来自数字组合经理和他们的应用团队。反馈内容通常包括平台需求和消费者反馈。

一个典型的 Sourced Group 的客户有多个开发团队，每一个团队都有自己的项目、技能集合和他们应用程序的云计算的使用（如果有的话）。这个组织的多样性使引入持续部署流水线的任务复杂化。对于企业组织来说常见的情况是，他们成功地在云计算平台上实施了一个或者多个小型应用程序*试点*项目，这些项目在部署技术上由不同程度的人员

干预和自动化。这些试点项目是引入持续部署流水线的一个好起点。可以参考第 10 章中关于推出 DevOps 实践的一些讨论。

设计和实施一个新的持续部署流水线并对组织进行培训所需要做出的努力取决于许多因素，包括一个或者多个现有平台、人员的可用性、技能和实践经验以及来自更广泛的业务的支持与承诺。另一个重要因素是客户环境被监管和限定的级别，这些监管或者限定来自一个或者多个行政体，比如澳大利亚审慎监管局（Australian Prudential Regulatory Authority，APRA）或者支付卡行业（Payment Card Industry，PCI）。

除了适职团队以外，观察图 12-1 所示的安全运维人员（SecOps）的角色。因为安全运维人员已经与开发团队有了交互，并且因为对任何企业来说安全都是一个关键的考虑因素，所以安全运维工程师团队就成为引入持续部署流水线过程中的一个自然附属。因此，安全运维工程师总是成为每一个持续部署流水线契约过程中的关键干系人。正如我们在第 8 章中讨论的那样，安全运维人员有一个具有挑战性的角色需要完成，并且需要平衡对商业风险的速度和敏捷性的追求。

对持续部署流水线本身及它部署的应用来说，安全都是 CDP 的一个核心关注点。通用持续部署流水线的使用，不仅保留了当前的安全机制，而且在很多案例中还极大地增强了这些安全机制，其中部分原因是持续部署流水线能够对它部署的全部组件执行策略和过程。另一个增强是减少了有权限用户和系统的数量。虽然安全运维人员通常从质疑持续部署流水线开始，但随着时间的推移和经验的增加，他们倾向于成为持续部署流水线的热情支持者。这种情况的部分原因是，他们的关注点自动合并到构建中，正如我们在应用程序生命周期阶段的章节所讨论的。

执行官支持和跨项目组合项目管理也是每一个契约期间的关键。任何新技术的企业级部署都需要得到高级执行官的支持，以及一个有经验的项目经理来解决内部挑战、组织资源、在团队之间调解、辅助干系人管理，以及通常确保平滑的交付。在第 10 章中我们也更广泛地讨论了这些问题。

在持续部署流水线契约开始时，通常每个应用 DevOps 团队都有自己的持续部署流水线。这些持续部署流水线一般是离散的和异构的。为了实现集中安全、合规性和规模经济，将通用工具和技术的所有权分配给持续部署工程团队。这个团队对涉及的工具提供企业范围的支持。由 Sourced 引入的一些典型工具包括 Splunk、Atlassian Confluence、Sonatype Nexus、Atlassian Bamboo、Atlassian Stash 和 Atlassian JIRA。工具团队是持续部署流水线卓越中心的一部分，因此是它持续部署工程团队，它是现有 DevOps 团队的成员。理想情况下，这些成员希望专攻各自的技术，或者已经发展或者想要发展支持企业范围平台所需要的技能。

12.3 持续部署流水线

持续部署流水线为企业提供了一个标准的方法以便管理应用程序的生命周期。规模

较大的企业，特别是金融领域的企业，有强大的风险管理框架，通常情愿牺牲少量的敏捷性来换取保证性和降低风险。通过隔离变更到功能分支、测试该变更并提供到生产环境的受限的路径，应用程序生命周期的标准化降低了风险。

本节（迄今为止本章最详细的部分）将讨论持续部署流水线的以下 3 个部分：工具、标准化的应用程序生命周期、状态与持久性的管理。

12.3.1　持续部署流水线工具

图 12-2 给出了形成持续部署流水线的工具以及它们之间交互的概览。将来自 Atlassian JIRA（票据系统）的数据同时提供给 Atlassian Stash（源代码修改系统）和 Atlassian Bamboo（持续部署系统）。Stash 为 Bamboo 提供了应用程序和配置源代码以构建镜像，然后 Bamboo 将这些镜像部署到 AWS。由于 Bamboo 不直接和 AWS 交互，所以它利用商业插件“AWS 的任务”。

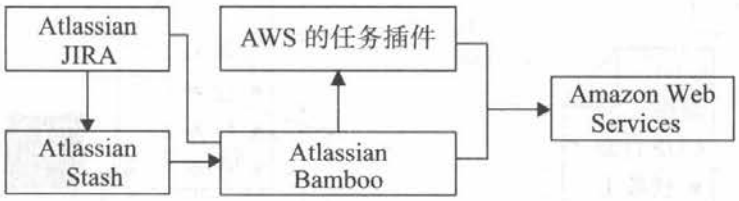


图 12-2　完整的持续部署流水线工具集 [标注法：架构]

任何持续部署流水线的两个基本元素是源代码存储库和持续集成 / 持续部署（CI/CD）工具。市场上有很多开源的、托管的和商业的源代码存储库，但是企业通常有监管的关注或者知识产权（Intellectual Property，IP）保护需求，它们控制使用防火墙解决方案（例如 Atlassian Stash 或者 GitHub 企业版）。Sourced Group 使用 Atlassian Bamboo 作为选择的持续部署工具，它形成持续部署流水线的支柱。Atlassian Bamboo 提供了一些关键功能，这些功能对于企业持续部署流水线是重要的，例如与票据和审核追踪系统的紧密集成，但它绝无仅有的分支管理是实现持续部署流水线目标的关键。这一点，我们将在后面进行讨论。

持续部署流水线的核心功能是执行标准化的应用程序生命周期，同时带来保证性收益。生命周期定义为（软件）计划，与源代码控制系统一起执行。为了提供保证性，重要的是，将相同的生命周期（或计划）应用于功能或者测试分支和主线或集成分支。Atlassian Bamboo 通过支持计划中的分支感知来实现这一点，它允许在任意数量的源代码分支上执行该计划的多个副本。自动检测和清理分支，进一步减少人的努力。图 12-3 展示了如何使用分支的虚拟计划。

相反，许多持续部署系统以不同的计划管理每一个分支。如图 12-4 所示，作为不同计划管理每一个分支将创建一个管理点，并不可避免地导致计划之间的漂移，这相应地引起标准化和保证性的损失。

虽然对持续部署流水线的成功来说并不是必需的，但通常使用票据管理解决方案（例如 Atlassian JIRA）。这极大地改进了该业务记录到日志中的针对问题或功能请求的代码变更的交叉引用。这种集成可以进一步做到基于在最近应用构建过程中被关闭的票据（Ticket）来自动化地生成发布说明（Release Note）。

可以将持续部署流水线参考架构应用于多个公有云和私有云环境中。为了本章的目的，我们将 AWS 作为目标平台。这个案例研究的后续部分假设对大量 AWS 服务和产品有基本的理解。应该参考第 2 章来学习云的概念，并通过 AWS 详细文档来了解特定信息。

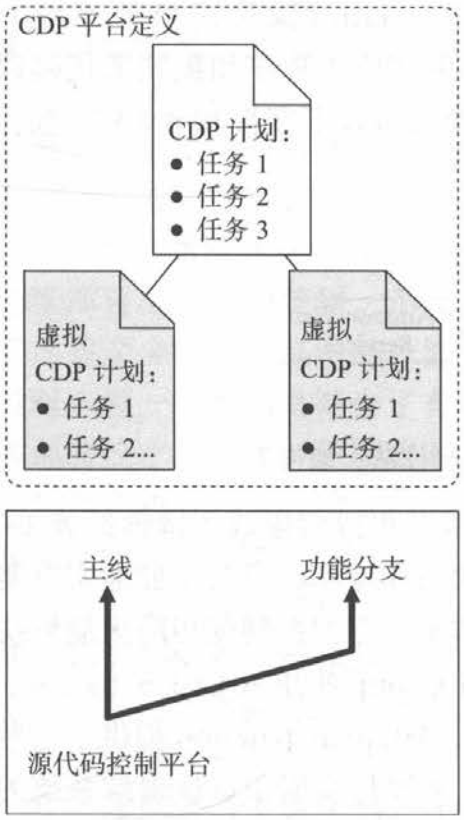


图 12-3　每个分支的虚拟持续部署流水线（CDP）计划 [标注法：架构]

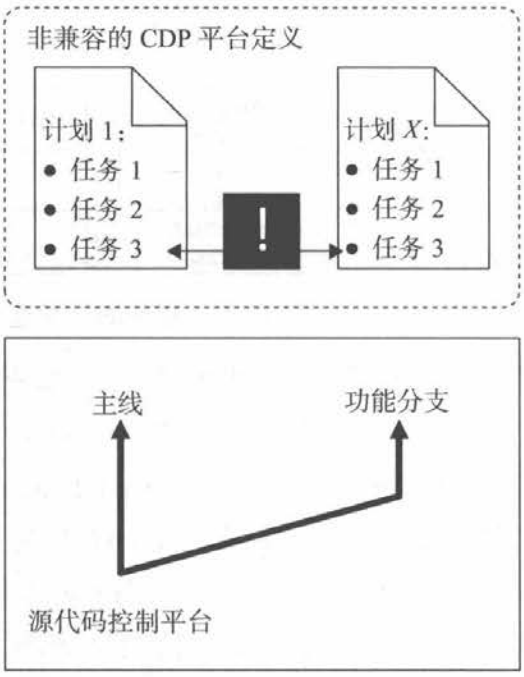


图 12-4　许多 CD 工具需要为每个分支创建离散的计划，导致漂移 [标注法：架构]

12.3.2　使用 AWS CloudFormation 的环境定义

持续部署流水线非常依赖于亚马逊的 CloudFormation（CF）。CF 提供了定义完整虚拟环境的能力，包括资源和安全组件，作为用 JSON 编写的声明的配置文件。可以对环境进行一致性地复制并获得与应用程序代码一样级别的单元测试。如果一个特定资源请求失败了，可能由于错误的配置，那么整个栈将被简单地拆掉。这样可以让客户节省维持一个有错误的栈而付出的费用。

为少量的应用程序开发和维护 Amazon CF 模板是相对简单的。然而，随着应用程序数量的增加，为每个应用程序维护一个独立的模板就变得越来越低效了。此外，持续部

署流水线为其大部分步骤和应用程序中的每一层都使用了一个独立的 CF 脚本。

AWS 经常发布用户希望利用的更新和最佳实践。然而，对模板引入变更是耗时且复杂的，因为必须管理不同的代码基。Sourced Group 解决这个挑战的方法是，管理一组常用的通用运维模板，这些模板在“烧制”或者部署时动态地与应用特定的模板合并，如图 12-5 所示。这确保变更和更新在持续部署流水线的控制下高效地渗透到所有的环境中。提供集中式管理模板集的另一个好处是，在将新的应用程序添加到持续部署流水线时，它可以极大地减少团队需要的时间和工作。在团队有极少的或者完全没有部署应用到 AWS 的经验时，这就显得特别重要，因为它去除了大量繁重的工作。

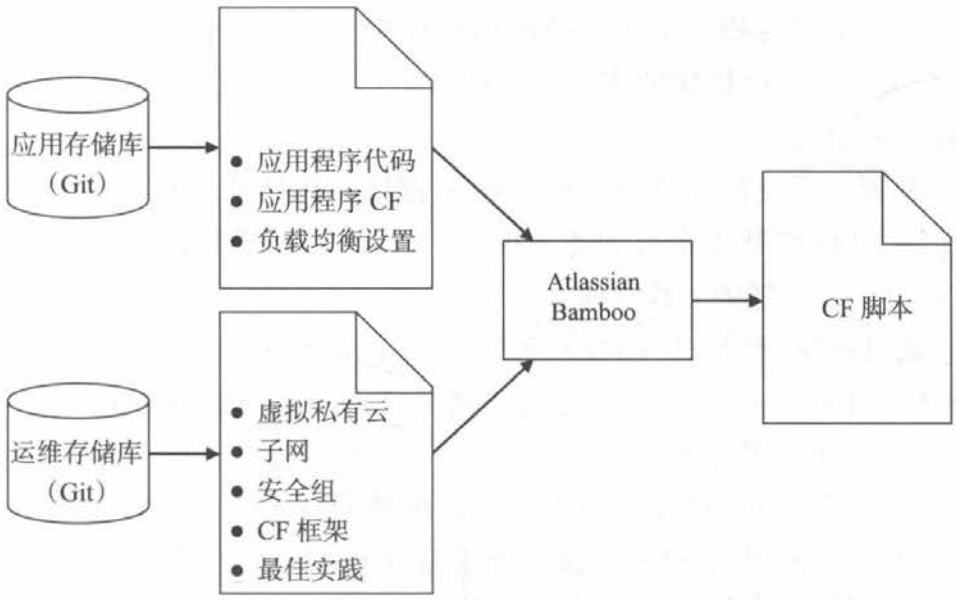


图 12-5　将运维模板和应用配置合并为一个 CF 脚本 [标注法：架构]

图 12-5 展示了 CF 模板合并过程。手头上的应用模板与独立的运维模板合并以创建一个在“烧制”过程中使用的单一模板。运维模板包括边界安全组（负责阻止外部访问组织的虚拟私有云（Virtual Private Cloud，VPC））和网络安全组（负责在虚拟私有云边界内的网络的安全）。这些模板与其他模板按照优先级方式合并，这样应用模板就不会重写其中一个运维模板中指定的配置。

运维模板存储在一个 Git 存储库中，应用模板存储在另一个 Git 存储库中。这样就提供了更高级别的控制和隔离。Atlassian Bamboo 将各种模板合并为一个 CF 模板。

单独的 CF 模板的合并也可以实现企业安全策略中描述的那些控制的实现。特别是，可以说明基于安全策略的关于不同组件的需求，并且可以集中实现这些需求，这就所谓的“消费单位”。例如，如果企业的要求是，所有简单存储服务（Simple Storage Service，S3）桶都记录到日志中并启用了版本控制，并且如果应用程序的 CF 模板使用 S3 桶，那么在合并的 CF 模板中标准 S3 桶将替换为在标准桶之上的但实现了那些需求的版本。这个扩展的版本是由持续部署工程团队与 SecOps 合作构建的。因此，通过消费这个 S3“单元”，扩展后的版本的 S3 桶（根据定义）是与 S3 兼容的。

12.3.3　标准化的应用程序生命周期概览及其使用

持续部署流水线的主要技术结果是应用程序生命周期的标准化。应用程序生命周期可以分解为 5 个主要阶段，如图 12-6 所示。



图 12-6　持续部署流水线中的应用程序经过的阶段 [标注法：架构]

1）*构建和测试*。执行功能代码测试并生成应用程序工件。这两个主题不是本章关注的要点——关于这两个主题请参阅第 5 章。

2）*烧制*。在临时目标操作系统上启动应用程序工件和配置，然后通过对快照做镜像来烧制，从这个镜像可以创建新的虚拟机。在 AWS 中，这称为亚马逊机器镜像（Amazon Machine Image，AMI）。

3）*部署*。部署一个新的、独立的栈的应用程序，包括通过自动伸缩组（Autoscaling Group，ASG）以 AMI 的副本新启动的虚拟机，以及支持基础设施和配置，如负载均衡、伸缩、监控、组网和（可能的）数据库。

4）*发布*。通过将指向现有栈的域名系统（Domain Name System，DNS）条目修改为现在指向新栈来发布该新栈。在发布之前，需要修改新栈来匹配当前栈的容量和规模以便能够在推出新栈时保证服务的连续性。

5）*拆解*。一旦将所有流量迁移到新栈时，拆解旧栈，因为不再需要它了。慎重的做法是，在拆解旧栈前执行安全检查以保证所有流量已经从当前环境迁移了。如果拆解了一个正在服务生产环境请求的环境，那么这将是一个令人尴尬的事情。

在下一节中，我们将详细讨论每一个阶段——这里我们只描述整体工作和它们的使用。标准化的应用程序生命周期在持续部署系统（Atlassian Bamboo）中作为一个计划实施。Bamboo 监控源代码控制系统，并在任何分支上的每一次提交后都执行这个计划。标准化的应用程序生命周期和源代码控制系统对应用程序开发人员和业务人员都提供了高的保证性。如果生命周期在一个低位的分支（如功能分支或测试分支）上成功运行，那么你一定可以保证这个变更将在更高位的分支（如生产）上成功执行。

当面对进行变更的任务时，如添加一个新的功能，应用程序开发人员从源代码创建分支、进行变更，然后将变更提交给这个分支。通过持续部署流水线这可以触发标准的应用程序生命周期目标。持续部署流水线将为这个功能分支构建离散的和独立的环境，允许进行单独的测试和试验，而不会中断其他功能开发。在这个环境中，验证功能工件——最好是通过高度自动化的测试。非阻塞的、独立的环境方法是平台的一个基础。大的和试验的功能可以与小的变更一起并行运行，虽然它们仍然满足同一个标准。对于开发者来说，一个完整的可发布的功能是由什么构成的是很清晰的，而业务获得了对一次迭代或者不同功能进展的实时视图。

当一个功能是可发布的，或者生产环境就绪时，开发人员通过 Git “ pull request” 将不同的功能分支合并到高位的分支中。然后首席开发人员负责评审该功能分支的构建结果，访问验证环境，接受它进入迭代分支，如图 12-7 所示。一旦冲迭代支完成，团队负责人将所有变更块提交给测试团队。小规模的团队，或者具有较少监管和风险管理需求的团队，可以选择简化这个模型，这样可以将功能分支直接发布到用户验收测试（User Acceptance Test，UAT）或者预发布环境中。

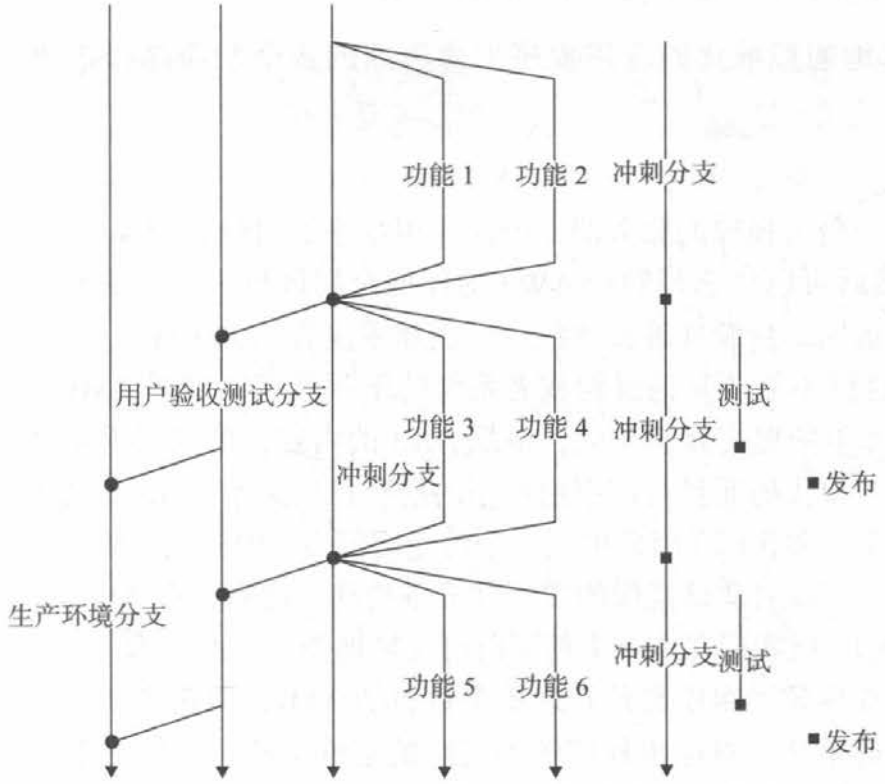


图 12-7　独立地开发、测试和集成功能 [标注法：左边（版本管理分支）；右边（开发冲刺、独立测试、发布阶段的时间轴）]

你可能注意到，第 5 章引入的持续部署流水线与本章讲述的持续部署流水线在形式上相当不同。特别是，第 5 章讲到的持续部署流水线与提交前测试、提交、构建 / 打包 / 单元测试、集成测试、用户验收测试 / 预发布，以及生产环境相关。大部分与这里讨论的分支集合是一致的。这里的持续部署流水线是为每一个分支制定的：功能分支检查本地测试；冲刺分支测试不同分支之间、与其他系统和第三方服务之间的集成；由用户验收分支创建的栈用于验收和性能测试；生产环境分支实际上对应于现场应用。

除了生产环境以外，高位分支的使用是特定于每个组织的，但 DevOps 通常提倡更少的分支而不是更多的分支——参见第 5 章中的讨论。企业经常有大的、可能在地理上分散的团队，这些团队的大部分在软件开发上采用“迭代”方法学。由于监管和风险需求，经常需要在发布前，让所有变更都通过一个独立测试团队的评估。通过为每个分支提供离散的非阻塞环境，持续部署流水线迎合了这种需求。这使得在一个独立团队测

试时开发能够继续进行。使用这种方法，开发不能长时间阻塞（等待测试和签收（sign off））——虽然功能提交与发布之间的时间可能仍然很长，但项目的整体速度没有受到影响。

图 12-7 展示了进入用户验收测试和生产环境的功能流。可以独立地开发、测试和部署独立的分支。

12.3.4 标准化的应用程序生命周期阶段

这里我们讨论的标准化的应用程序生命周期的各个阶段是：烧制、部署、发布和拆解。

1. 烧制

烧制阶段在一台可释放的服务器上创建应用程序的自包含镜像（AMI）。将服务器烧制到 AMI 中，然后可以将它复制到 AWS 的任何全局区域中。在第 5 章中，我们讨论了可以实施的烧制级别。持续部署流水线可以创建重度烧制的镜像，并为每个 AMI 创建一个 AWS ASG，这样不需要其他过程或者系统的干预就可以部署 AMI。AMI 是不可改变的，与 Git 存储库中的提交直接关联，根据 AMI 的内容，可以获得高度的可视性和置信度。因此，不但对源代码而且对同应用程序的所有已知副本，Git 形成单点真理，因为它们都是从给定提交点源代码（因此也是一个给定时间点）的源代码烧制的。这个镜像不需要进一步启动——可以启动该镜像的实例并准备启动。注意，在 AWS 中虚拟机称为实例（即，AMI 的实例），但我们在本章中将同时可交换地使用这两个术语。

虽然这个过程导致了组织必须管理更多数量的 AMI，但它带来了许多好处。这些好处包括：部署的简单化、对应用程序的不间断的运维支持、在伸缩事件时减少虚拟机启动时间、启动过程中减少失败和增加的一致性。

图 12-8 展示了烧制过程中的每一个步骤。应用的每一层都需要一个独立的镜像。如果一个业务应用包含了通常的 3 层（Web 层、应用层、数据库层），那么持续部署流水线就烧制 3 个不同的 AMI，每一层一个。简要地，烧制的步骤如下所示：

1）检查 Amazon CF 模板和从 Atlassian Stash 构建脚本。

2）合并应用程序和运维 Amazon CF 模板，如图 12-5 的讨论。产生的 CF 模板为烧制准备环境。

3）将 Amazon CF 模板上传到 Amazon S3。

4）运行预处理，它从 DynamoDB 中收集所有可用的构建工件（参考关于管理流水线状态的章节）。

5）从合并后的 CF 模板构建烧制的环境。这将为每一层创建一个构建器实例，它们分别形成烧制的每个镜像的基础。这一步也把需要的软件和配置加载到构建器实例中，通过使用类似 CF 的 cfn-init 的启动系统。该启动系统将在下一节中详细介绍。

6）将每个构建器实例的副本烧制为 AMI。

7）运行后处理，它收集新的 AMI ID，把它们放入工件仓库（参考关于管理流水线状态的章节），用合适的说明信息标志它们（如提交 ID）。

8）拆解和清理构建器实例。

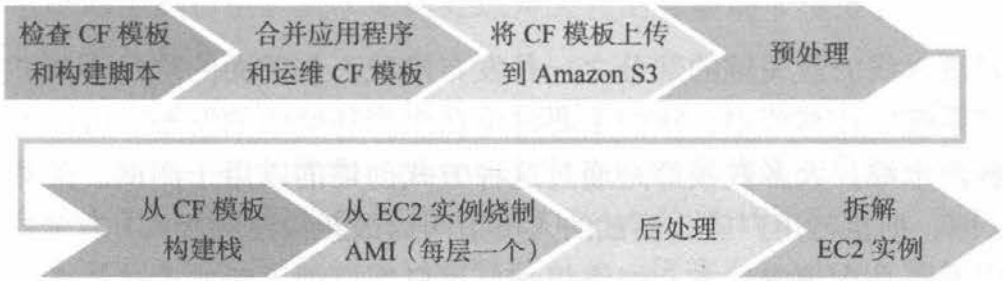


图 12-8　持续部署流水线烧制过程 [标注法：波特价值链]

烧制过程使用 CF 来启动一个标准的（Vanilla）实例或者预定义的企业标准操作环境 AMI，如 Amazon Linux、Redhat Enterprise Linux 或者 Microsoft Windows Server。该实例使用 AWS CF 启动系统 cfn-init 独立地启动应用程序。包的安装、文件和服务管理都是通过 cfn-init 代理处理的。cfn-init 是声明式并实现原子行为，所以它要么表示所有声明项都成功完成，要么表示失败和终止。这提供了高度的保证性：配置是正确的，并且应用程序正确地安装。如果在指定的超时后没有收到执行成功的信号，那么就销毁实例、将构建标记为中断——回想在第 5 章中中断构建过程的讨论。如果 cfn-init 执行成功了，那么通过 Bamboo “ AWS 的任务” 插件调用 Amazon EC2 应用程序编程接口来从当前运行的实例烧制 AMI。烧制完成后，该实例就不要需要了，因此将被终止运行。

2. 部署

通过在前一个阶段的重度烧制的镜像，生命周期的部署阶段就变得非常简单。对每一层而言，通过的烧制的镜像启动配置（Launch Configuration，LC）将它交给新创建的 AWS 自动伸缩组。然后，持续部署流水线指定这个自动伸缩组中的最小实例数，相应地它马上启动该数量的 AMI 实例。为每一个自动伸缩组配置弹性负载均衡（Elastic Load Balancing，ELB），在可用的虚拟机上分配传入的请求。节点注册是不需要的，因为自动伸缩组和负载均衡器维护这些信息。实例也不需要进行额外配置，因为这些配置在烧制阶段就启动了。

运维存储库通过它的 CF 模板为自动伸缩组定义基础标准。因此，所有的自动伸缩组都是按照最佳实践 CloudWatch 警报、触发器和自动伸缩策略等来设置的。应用团队即使不是自动伸缩性方面的专家也能使用这些服务，并且很多实现错误也可以通过这样的标准化来避免。

给自动伸缩组实例提供了一个身份和访问管理（Identity and Access Management，IAM）角色来保障对其他 AWS 服务（如 S3）的安全访问。因为角色是独立栈的一部分且定义在应用程序存储库中，所以通过源代码控制和部署系统对角色的任何变更进行追踪

和审计。这创建了安全的 IAM 凭证，避免了需要以明文字符串或者配置文件的方式向实例传递凭证。IAM 描述也是与栈直接关联的，因此如果发生了不合法的访问，那么可以更新栈来刷新 IAM 凭证。

3. 发布

持续部署流水线的最关键的部分之一是发布阶段。与前面的阶段不同，发布阶段影响当前的生产系统。在本节中，我们主要讨论在生产环境中的发布。可以为每一个分支执行持续部署流水线，大多数栈都是通过这种方式创建的以用于测试。虽然发布仍然可能是与测试（即，把执行用户验收测试的人员指向到最新构建的版本）有关的步骤，但是在生产环境中发布是有趣的地方：如何将来自用户的持续不断的请求流重定向到应用程序的新版本？

虽然发布过程自身是自动的，但是持续部署流水线可以在这个步骤中暂停，等待信息把关人手动批准流水线中的这个步骤。这时手动触发器的使用也让在发布前进一步测试和验证获得无限的时间，这里，新创建的栈已经是可用的。另外，如果对自动测试集合有高度的置信度，那么只要满足所有的先决条件持续部署流水线可以执行该步骤。

图 12-9 展示了如何调整 DNS 条目来实现把旧栈的流量重定向到新栈。我们这里可以使用名为 Route 53 的 AWS DNS 服务。

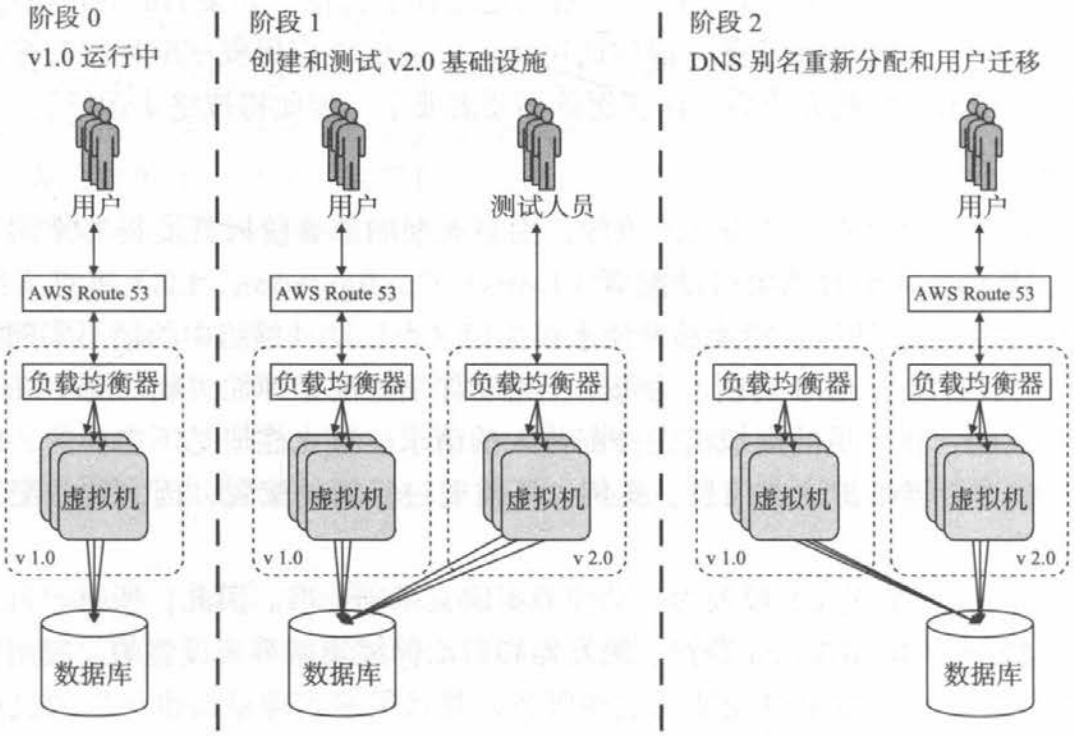


图 12-9 应用程序发布阶段 [标注法：架构]

在发布的阶段 0，只有旧版本（v1.0）正在运行。阶段 1 是这个过程中的一点，其中我们有两个应用程序栈：当前生产栈，称为“红栈”或者活跃栈；新构建的栈，称为

“黑栈”或者非活跃栈。发布是在阶段 2 完成的，这里实时流量从红栈重定向到黑栈。如图 12-9 所示，栈的无状态部分在两个不同版本之间是完全独立的，但持久性数据库将从一个构建保留到下一个构建。因此，在阶段 1 中显示的测试必须保持无损的，因为它是在现场生产数据库上执行的。关于如何处理持久性的更多细节将推后到本章中关于管理持久性的部分再予以阐述。

图 12-10 更详细地说明了 DNS 结构。使用 Route 53，为每个栈创建了一个类型为 ALIAS 的 DNS 条目。也就是说，ALIAS 指向到栈中最顶层的负载均衡器。通过这个 ALIAS，将测试人员引导到指定的构建（如图 12-10 阶段 1 所示）上。系统的用户一般是通过用户友好的主机名（CNAME）作为进入点。将这个 CNAME 解析为一个“浮动的 DNS 记录”，总是指向活跃栈的另一个 CNAME。通过将这个浮动的 DNS 记录从旧的 ALIAS 切换到新的 ALIAS 来完成发布。我们在第 2 章中概要地讨论了 DNS，在第 6 章中讨论了 DNS 在部署中的角色。这里特殊之处是，3 个 DNS 记录是分层的，具有对应的次序指向关系。需要注意的第 2 点是，这里使用了 ALIAS 和 CNAME（从一个主机名到另一个主机名），而不是使用简单的“ A”记录（从一个主机名到一个或者多个 IP 地址）。对于 CNAME，很容易理解它为什么是必需的。然而，使用 ALIAS，因为 AWS 弹性负载均衡提供了一个自动生成的（非用户友好的）DNS 主机名。这样做的目的是 AWS 可以为负载均衡透明地伸缩资源。

更新浮动记录是使用为发布准备的一个独立的 CF 脚本来实现的。与前面的阶段不同，在那些阶段中，创建一个离散的 CF 栈；而在发布阶段中，更新一个现有的 CF 栈。这利用了 CF 内置的更新功能，启用这个功能需要在 AWS Bamboo 插件中设置“如果已存在就更新栈”的标志。当启用这个功能时，Bamboo 将首先尝试寻找一个具有期望名字的栈，然后更新它；如果没有找到这样的栈，Bamboo 就创建这个栈。通过 CF 来控制 DNS 变更，发布或者回滚就成为可以审计的 CF 栈更新。程序清单 12-1 展示了一个支持这种更新的 CF DNS 记录的样本。持续部署流水线通过“ BuildNumber”参数提供当前构建的版本号。

程序清单 12-1　发布 CloudFormation 的示例

```
"Resources" : {
  "Route53DNSRecord" : {
   "Type" : "AWS::Route53::RecordSet",
   "Properties" : {
    "HostedZoneName" : mydomain.com.,
    "Comment" : "Application DNS Record",
    "Name" : "myapplication.",
    "Type" : "CNAME",
    "TTL" : "10",
    "ResourceRecords" : "myapplication-",{"Ref:
```

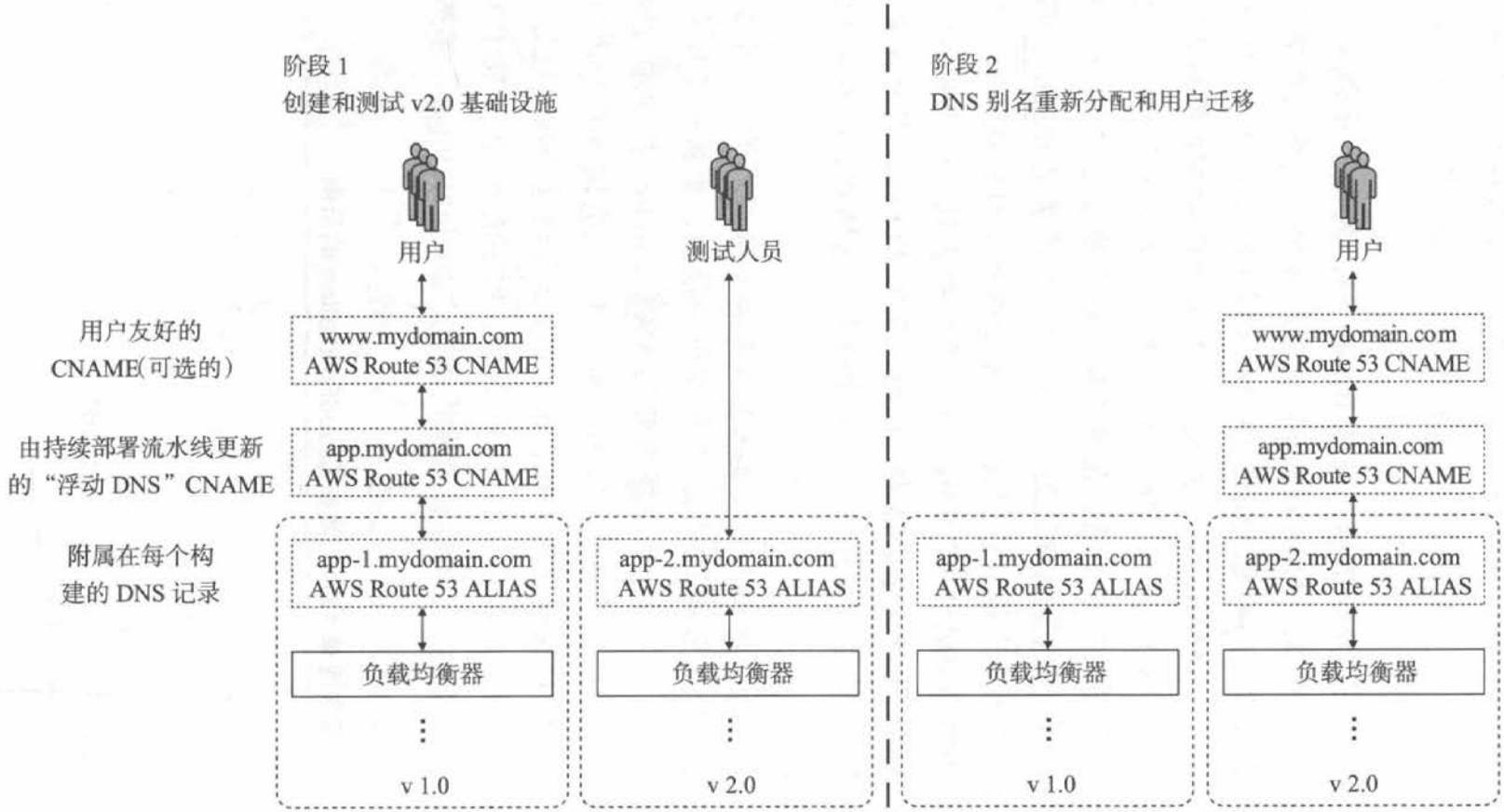


图 12-10　发布过程中的 DNS 结构和变更（图 12-9 中的阶段）[标注法：架构]

```
        "BuildNumber"},".mydomain.com"
    }
  }
}
```

就其本质而言，发布是持续部署流水线的最危险的阶段，因为它对生产应用有直接的影响。我们使用了大量的技术来减小这种风险，允许平滑的自治部署（其中最关键的部分是流量匹配和回滚）。

（1）流量匹配

一旦确认应用程序栈可以进行生产发布，我们仍然需要采取一些步骤来保证栈之间的平滑迁移。当把流量从当前生产栈迁移到新栈时，我们正在把流量从一个繁忙的、“热的”环境迁移到一个“冷的”、未伸缩的环境。如果这个过程没有得到正确的管理，那么它将对性能造成破坏性的影响，并可能导致应用中断。因此，重要的是，在将在线流量滚动到新环境前，我们预热（增加）这个新环境。通过利用 AWS 应用程序编程接口，我们以编程方式检查与当前生产负载均衡器相关联的当前健康实例的数量。获得这个信息让我们伸缩当前环境以匹配生产环境所要求的相同数量的实例。这是通过调整相关的自动伸缩组的期望实例数来实现的。同样重要的是，在滚动流量前，保证新添加的实例在负载均衡器上是“ InService”状态。取决于自动伸缩组设置，新的环境也可能由于低流量而缩减（实例数）。因此，要么这个迁移流量的任务在实际切换流量前被（人工）接管，要么就必须把最低（Minimum）实例数量设置成和期望（Desired）实例数量相同，然后在流量滚动完成后再改回去。

如果部署的应用使用了基于实例的文件或者内存缓存系统，那么在流量滚动前预热缓存是有利的。这将保证应用的首批用户[⊖]不因为缓存创建或重创建过程而影响访问质量。

（2）回滚

如果由于某种原因发布失败了，那么它可能需要回滚，也就是说，切换回应用程序的旧版本。使用发布的 Route 53 和 CF 简化了回滚过程。在发布新栈到生产环境后，我们保持旧栈正常运行直到所有发布后测试完成。在这段时间中，如果发现了新发布版本中的问题，回滚到旧环境是一个简单且自动化的过程，基本上是发布的逆向操作。回滚可以通过 Bamboo 手动启动，然后 Bamboo 通过 CF 更新浮动 DNS 记录来重新指向旧栈。在这个过程中，依然关键的是持续部署流水线使流量 / 负载匹配——前面的生产栈很可能在这个时间内缩减，当它没有主动接收负载时。在关键回滚时流量匹配步骤可以跳过，其中低可用性和性能可能更倾向于做出必要回滚的条件。

4. 拆解

一旦将生产流量重定向到新栈并且也没有回滚的可能性了（通过风险评估或者其他业

⊖ 指在时间上访问该应用的前面几个用户）。——译者注

务过程确定的），那么旧栈就不再需要了。但是在拆解旧栈前，仍然需要采取一些步骤。

首先，为了避免拆解了一个正在生产环境中的栈，将持续部署流水线配置为在拆解前确认栈没有正在接收任何流量。有许多检查，包括轮询流量标志的弹性负载均衡和测量每个关联的虚拟机的 CPU 使用率。如果其中任何一个指标返回了非预期的值，那么持续部署流水线就会停止拆解活动并对运维发出警告。在运维确认了该栈未被使用后，他可以重新运行这个任务。这个步骤对于保护因疏忽造成的前期拆解是至关重要的，例如如果即使在流量已经从该栈滚动走了后旧栈仍然在处理一些批处理或基于队列的任务。

其次，如果应用程序栈包含了 S3 桶，那么在拆解前持续部署流水线必须首先从这些存储桶中删除所有对象。CF 将只允许拆解空的 S3 桶以保证数据不是被不经意销毁的。

最后，因为每一个栈都是完全离散的，所以持续部署流水线可以删除旧栈，这就意味着停止所有关联的资源。

持续部署流水线的一个主要好处是简化了为不同测试环境创建新开发应用栈的过程。这种简便性经常导致 AWS 资源的快速消费，随着为许多分支和环境创建新的开发栈。为了确保 AWS 成本保持在可接受的水平，当不再需要开发栈时将它们拆解是有必要的。在大多数企业中，这是通过团队责任和一组自动强制规则的组合来实现的。开发团队有责任保证他们的环境在使用后对它们进行清理和拆解。在开发分支上，拆解是持续部署流水线的最后阶段，它是由手动触发的。还没有拆解的任何栈将仍然显示为“在进行中”，这在 Bamboo 中作为活跃的环境而可见。一组自动化的任务能够有助于保证将未使用的栈移除：

- 在非工作时间内关闭所有非生产环境。
- 在周末和假期强制拆解所有非生产栈。

通过对所有的 CF 资源打标签，我们可以保证这些过程只应用于非生产栈。

12.3.5 管理复杂的应用程序和流水线状态

复杂的应用程序可能包含有数十甚至数百个组件。即使遵循了 DevOps 最佳实践并避免了大量组件之间的紧密依赖，很多应用栈可能仍然包含几个组件。例如，一个应用包含 Web 层、业务逻辑层以及数据存储层上的 MySQL 和 S3。那么这通常需要在创建上层前创建数据存储，这样可以配置上层来使用这些数据存储。因此，持续部署流水线代理（应用的每个组件都有自己的代理）可以调度为不同的阶段（Phase）的一个部分。如图 12-11 所示，数据存储的持续部署流水线代理在阶段 1 中调度，Web 层和业务逻辑层的持续部署流水线代理在阶段 2 中调度。这种调度必须以应用为单位进行定义，它是通过为每一个组件分配阶段数来实现的。虽然这些代理通常独立地处理，但是有一个例外：发布是在相同阶段中的所有代理上同步发生的。这是必须的，以便可以一致性地重定向实时流量，即，在任何时刻将来自一个构建的栈用于处理实时流量。

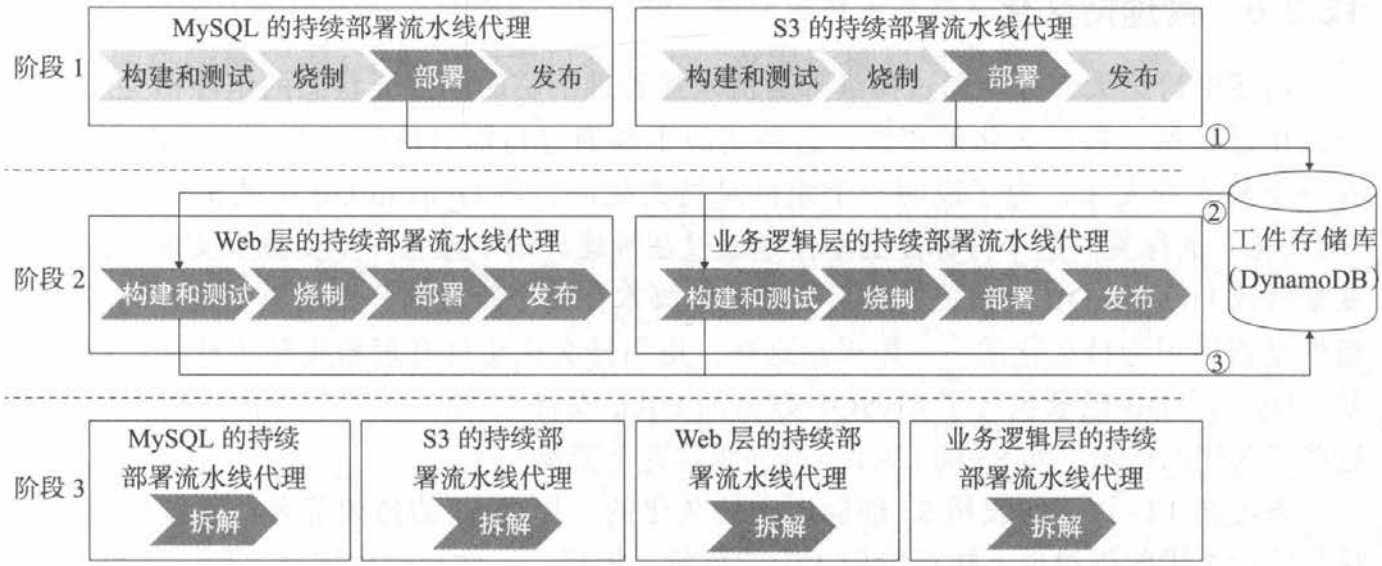


图 12-11　流水线状态 [标注法：波特价值链和架构]

另一方面是，不是每个组件都实际上需要执行持续部署流水线中的每个阶段。在这个例子中使用的数据存储是 AWS 产品，因此它可以被直接部署而不需要任何其他阶段了。在图 12-11 中，这是通过把阶段 1 中的两个数据存储代理中的其他阶段设置为浅灰色来表示的。阶段 3 包含了所有组件的拆解阶段。这是在定义中隐含表示：拆解是所有组件同步完成的。同时需要注意，为给定应用程序定制持续部署流水线并不影响持续部署流水线实现或者最佳实践——所有这些都是在应用特定的 CF 模板上完成的，这些模板与企业范围的运维 CF 模板合并。因此，可以相对容易地实现和测试持续部署流水线或者运维 CF 模板的变更，然后将它们同时应用在数百个应用上。

持续部署流水线的最后一个组件是流水线状态存储库。它为存储 CF 的输出和其他工件（这些输出或者工件可以被后续的流水线阶段或者应用本身消费）提供了一个高可用的、一致的和可靠的存储服务。持续部署流水线使用 Amazon 的 DynamoDB 作为工件存储库来管理流水线状态，如图 12-11 的右侧所示。DynamoDB 提供了一个完全管理的 NoSQL 数据存储，该数据存储是高可用的、分布式的、安全的并提供一致的低延迟性能。

在图 12-11 中与工件存储库相邻的圆圈中的数字表示两个组件的构建和测试阶段示例的数据创建和消费的顺序。当某个组件的持续部署流水线代理到达阶段的结束时，它的后处理系统收集所有的相关联输出并把这些输出存储在 DynamoDB 中。持续部署流水线代理进入下一个阶段并从 DynamoDB 中获取该构建的所有已知工件，使它访问来自所有先前构建过程、阶段和代理的组合信息。通过解耦数据处理（DynamoDB）和控制（CDP），持续部署流水线变得更可伸缩：不同阶段、组件和过程的数量与复杂度变得不相关。一个过程消费来自 DynamoDB 的数据，执行它的任务，将新形成的信息存储回 DynamoDB 中，为下一个阶段的消费做准备。

12.3.6 管理持久化

前述的讨论大部分与通过持续部署流水线管理的临时的或无状态的组件相关。大部分应用还包括一些持久化的资源，这些资源不能通过与临时栈相同的生命周期来管理。在一个特定分支上，为了说明一个组件是持久化的，将 DynamoDB 中的相关记录打上“持久化”的标签。这个打标签的动作是通过在构建计划中设置一个变量来实现的，这个变量修改 DynamoDB 中相关的条目。预处理脚本在每个阶段的开始收集变量并判断一个组件是否标识为持久化的——如果是这样，重用持久化组件并屏蔽重新创建和拆解。例如，DynamoDB 记录包含了 MySQL 数据的 URL 或者 S3 桶。所有后续的构建都将获取这些持久化的记录，如 S3 桶 URL，直到删除这个变量。

参考图 12-11。假设将 S3 桶标识为持久化的，那么 S3 的持续部署流水线代理既不执行阶段 1 中的创建也不执行阶段 3 中的拆解。相反，工件存储库保留了指向现有 S3 桶的 URL，并将所有其他组件配置为使用这个 URL。

因为持久化的标志是每个分支的设定，所以生产分支以外的大多数组件不会使用到它。参考图 12-7 中的分支。通过在生产分支上设置持久化标志，保护现场生产数据库，它不会被替换或者拆解。然而，低位分支，如用户验收测试，没有设置这个标志。因此，自动测试套件可以依赖于一个清洁的、一致性的数据库来开始——即使中断最后的构建并而导致数据库处于不一致的形状。

如何升级持久化的数据存储（例如模式或者数据库引擎）是另一个复杂的问题——这个问题不在本章的范畴之内。

12.4 让安全成为持续部署流水线的基础

安全是在持续部署流水线中的基础层面加以解决，这体现在两个方面：对流水线本身的运维；与它管理的应用相关联的资源。

第 8 章在一定细节上讨论了安全。在本节，我们主要讨论使用 Amazon CF 来分隔离职责以及使用 AWS IAM 来进行身份认证和授权。

12.4.1 使用 Amazon CloudFormation 分离职责

如前面讨论 CF 时所述的那样，在网络传输组、网络安全组、运维人员组和应用组之间执行职责分离是通过 Amazon CF 模板的组合和基于优先级的破坏性（例如重写）合并过程来实现的。

网络传输组负责数据中心间的连通性，例如，确保企业数据中心到 AWS 的传输连通性。这包括配置 AWS Direct Connect、边界网关协议（Border Gateway Protocol，BGP）和 IPsec/VPN，也包括具有自动故障转移的冗余链路设置。网络安全组负责边界网络和

安全。这两个组维护定义环境的 CF 模板，持续部署流水线在这个环境中管理资源。例如，网络传输组的 CF 模板与虚拟私有云、对等连接和路由表有关。相比之下，网络安全组的模板与虚拟私有云子网、安全组和访问控制列表（Access Control List，ACL）有关。

运维人员维护实现了以下方面最佳实践的模板：主机和资源日志记录、高效利用 Amazon 可用区、资源标记等。类似地，每个应用组开发和维护应用程序特定的 CF 模板，这些模板主要关注应用程序的健康检查、自动伸缩规则、触发器和阈值。

在部署一个特定应用程序时，持续部署流水线从应用程序模板开始，首先叠加运维模板，这将有效地屏蔽或者覆盖由应用程序开发人员指定的任何无特权的配置设置。这种合并是偏向于运维的破坏性的，因此可以强制执行最佳实践和标准。例如，假设应用的所有者指定所有应用组件都应该存在于一个可用区内以最小化延迟，但是企业策略要求高的系统可用性——那么，运维 CF 模板就会覆盖该配置并把资源放在多个可用区中。

12.4.2　身份和访问管理

在持续部署流水线中有很多需要考虑身份认证和访问管理的领域。第一个领域是在主机系统管理。当在持续部署流水线的外部对主机进行管理时，常见和必要的是管理员可以远程登录或者发出远程命令。例如，如果一个应用服务器停止提供服务，那么管理员可以登录到主机上，调查问题，可能重新启动失败的服务。当有效和可靠地部署时，登录到主机变得不再必要——甚至实际上导致增加问题处理的时间。去除管理员访问减少了整体攻击面并去除了管理 SSH 公钥和私钥的需求。

Amazon IAM 角色、模式和策略可以广泛地用于限制对 EC2 实例、用户和 AWS 资源（如 AWS S3）等的访问及权限。如果需要获得对这些服务的详细解释，请参考相关的产品文档（见 12.7 节）。在 AWS 白皮书中可以找到运维和安全清单的部分内容。

在可能的情况下，需要权限访问 AWS 资源（例如，S3 桶中的对象和 Amazon 简单队列服务 SQS）的 EC2 实例，有一个与它们相关联的 IAM 角色。这允许集中管理访问权限并在需要时立即撤销访问权限。IAM 角色是在任何都可能是限制访问敏感 AWS 资源的首选方法。给通过持续部署流水线部署的所有兼容的资源默认发一个 IAM 角色或策略。

对于不能提供 IAM 角色支持但需要 IAM 凭证（例如，访问密钥和保密密钥）的应用，持续部署流水线可以把 IAM 凭证内置到需要它们的每一个虚拟机中。作为一个附加的安全措施，在每次部署栈时，这些凭证都会被轮换。也就是说，在烧制阶段，创建一组新的凭证，并烧制到需要它的 AMI 中；在拆解阶段，这些凭证是无效的。这种方式限制了安全漏洞对栈的影响，防止相同的凭证来访问属于其他应用或环境的资源。另外，这些凭证至少每 3 个月轮换一次。

12.5 高级概念

在本章中，我们已经简述了 Sourced Group 持续部署流水线框架的设计。随着企业的成熟，持续部署流水线方面出现了一些可以探索的高级领域。

12.5.1 最小化生产环境和非生产环境之间的偏移

企业中的一个常见问题是，长时间运行的非生产环境不断与生产环境发生偏移。Sourced Group 解决这个问题的方法是，针对每一次构建的最新的生产环境快照刷新非生产环境的所有方面，如图 12-12 所示。

非生产环境使用来自生产环境的持久化数据存储的夜间快照。从这些快照按需创建资源。这允许对只落后生产环境几个小时的持久化数据存储进行开发、测试、集成。

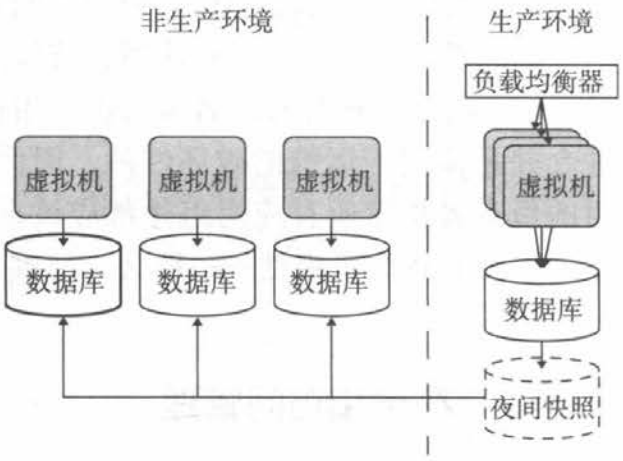


图 12-12 使用非生产环境构建的生产数据库快照 [标注法：架构]

12.5.2 解决供应商的限制

一些常见的云平台对它们服务的某些元素施加了硬限制。通常情况下，这些限制反映了服务正处于产品开发过程的某个时间点上。不管这些限制是临时的还是更永久的，持续部署流水线都需要考虑这些因素。这些限制中的一个是，对每个虚拟私有云的安全组数的 AWS 的当前限制——参考 12.7 节提到的相关文档。默认限制是 100 个安全组，对大多数企业部署有 200 个模糊的硬限制。这个限制影响了一个虚拟私有云中的资源部署并明显影响了持续部署流水线。为了解决这些挑战，为每一个客户实现一个“自动伸缩的虚拟私有云模型”。在这个模型中，针对安全组的可用性和最终耗尽，动态创建或者删除虚拟私有云。结果是按照每个安全组需求扩大应用交付虚拟私有云的“自动伸缩的”数量。高级过程流如下所示：

1）接收来自用户的栈构建命令。

2）计算该栈需要多少安全组。

3）检查所需的安全组数是否可以符合现有的应用交付虚拟私有云。

4）如果是，提供应用。

5）如果否，调用虚拟私有云构建作业来构建一个新的应用交付虚拟私有云。

6）从 DynamoDB 中的预填充配置表获取所需的 IP/ 子网细节。

7）部署应用栈。在 DynamoDB 中为该栈注册必要的虚拟私有云 ID。

一个看门人进程持续地查询 DynamoDB 数据来获得所有正在运行的虚拟私有云和活跃应用的信息。如果没有正在运行的应用使用给定的虚拟私有云，那么看门人进程将终

止虚拟私有云容器和相关的组件。

这个高级过程依赖于动态提供 IP 地址空间的能力和某些功能（如虚拟私有云对等）。在互连的企业环境中，关键的是回程数据中心连接完全通过应用程序编程接口来管理，解决方案如 AWS Direct Connect 的，这种解决方案可以创建虚拟接口。有些交叉连接元素的预填充可能是需要的（例如，虚拟局域网（Virtual Local Area Network，VLAN））。这些数据插入 DynamoDB，并执行一个容量管理进程来避免资源耗尽。

12.5.3　厂商锁定

Sourced Group 的持续部署流水线是与 AWS 高度绑定的，特别是 CF，因此带来了迁移到其他平台（例如 VMWare、OpenStack 和 CloudFoundry）的挑战。目前，导致这种问题的原因是在基础设施即服务（Infrastructure as a Service，IaaS）层缺少跨平台标准。未来，虽然供应商之间的平台即服务（Platform as a Service，PaaS）兼容性的发展将增加可移动性，但有理由假设，如果你需要控制 IaaS 层，那么总是需要一定量的初始繁杂工作来支持多个公有云供应商。

12.5.4　新的 AWS 内置服务的展望

在 2014 年 11 月的 AWS 全球大会上，发布了许多新产品。这些新产品聚焦在配置、部署和应用程序生命周期的管理——这明确地表示持续部署技术和过程已经达到了一定的成熟度和标准化，它们可以作为通用的软件即服务（Software as a Service，SaaS）来提供服务。

在最近发布的 AWS 服务中，AWS CodeDeploy 是其中一个有可能用于简化部分持续部署流水线的服务。CodeDeploy 是一个自动化部署代码到 Amazon EC2 实例的服务。CodeDeploy 替代了很多在烧制阶段过程中包含的启动活动，它实际上提供了一个更高级版本的 cfn-init。合并 CF 模板的 Sourced Group 技术允许相当简单地与 AWS CodeDeploy 集成，因此可以减少持续部署流水线的复杂度。

12.6　小结

多年以来，Sourced Group 一直在帮助企业实施持续部署流水线。构建 Sourced Group 的流水线大约经过 5 阶段生命周期：构建和测试、烧制、部署、发布以及拆解。每一个阶段都依赖于一组工具来实现该过程的自动化。然后将这个标准化的生命周期应用于应用程序源代码控制系统中的所有分支，当开发人员和业务人员将功能合并到生产环境中时，给他们提供高度的保证性。使用离散的应用栈和自动化的发布管理极大地减少了风险和发布软件所需要的时间。

安全是企业中的一个重要关注点，SecOps 是支持任何平台成功的关键。持续部署流水线提供了进入 AWS 环境的一个单一的入口点，其中，来自运维的可执行的 CF 模板提供了 SecOps 需要的监管和合规能力。

教育和文化是广泛采用持续部署流水线的核心因素。虽然持续部署流水线工程团队支持流水线本身，但持续部署流水线适职小组提供对新开发团队的支持。这些支持包括适职和传递知识，这些知识既包括通用的持续部署技术也包括特定持续部署流水线实施。

12.7 更多阅读材料

如需进一步阅读或者获取关于此处列出工具的扩展信息，请参考下面的这些链接：

- Atlassian：https://www.atlassian.com/
- Bamboo：https://www.atlassian.com/software/bamboo
- Bamboo 分支管理：https://confluence.atlassian.com/display/BAMBOO/Using+plan+branches
- Bamboo 的 AWS 任务插件：https://marketplace.atlassian.com/plugins/net.utoolity.atlassian.bamboo.tasks-for-aws
- Stash：https://www.atlassian.com/software/stash
- Stash 分支权限：https://confluence.atlassian.com/display/STASH/Using+branch+permissions
- Stash “pull”请求：https://confluence.atlassian.com/display/STASH/Using+pull+ requests+in+Stash
- JIRA：https://www.atlassian.com/software/jira
- JIRA 和 Bamboo 集成：https://confluence.atlassian.com/display/JIRA/Viewing+the+Bamboo+Builds+related+to+an+Issue
- Amazon Web Services：http://aws.amazon.com
- 持续交付 / 持续集成最佳实践：http://www.slideshare.net/AmazonWebServices/continuous-integration-and-deployment-best-practices-on-aws-adrian-white-aws-summit-sydney-2014
- CloudFormation 初始化：http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-init.html
- 包括 cfn-init 的 CloudFormation helper 脚本：http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-helper-scripts-reference.html
- 身份和访问管理（IAM）：http://aws.amazon.com/iam/
- DynamoDB：http://aws.amazon.com/dynamodb/

- 安全文档：http://aws.amazon.com/security/
- 白皮书，包括运维和安全检查清单：http://aws.amazon.com/whitepapers/
- VPC 限制，例如安全组的数量：http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Appendix_Limits.html
- CodeDeploy：http://aws.amazon.com/codedeploy/
- Splunk：http://www.splunk.com/
- Sonatype Nexus：http://www.sonatype.com/nexus

第 13 章　迁移到微服务

Sidney Shek

我们的产品帮助所有规模的团队追踪和共享事务，更聪明地工作并共同创造更好的软件。

——http://www. atlassian.com/company

13.1　Atlassian 概述

Atlassian 开发两类工具：一类是团队生产力工具，例如 JIRA（问题追踪和软件开发）、Confluence Wiki[⊖]、HipChat 消息和 JIRA 服务台；另一类是开发工具，例如 Bamboo 持续集成服务器和 Bitbucket 托管的存储库。前一章的案例研究描述了 Sourced Group 流水线如何使用其中一些工具。其中很多工具既可用于内部部署的服务器安装又可用于通过 Atlassian 托管的云产品。

Atlassian 云当前服务于来自 130 个国家的大约 20 000 个客户。一个“客户”代表一个注册了一个或者多个应用的团队或者组织，一个客户可能有十几到数千个终端用户。当前大约有 60 000 个应用实例，它们每天处理大约 1TB 的网络流量。为了支持这样的负载，Atlassian 在美国有两个生产环境数据中心（大约有 60 个机架服务客户请求，有 8200 个 CPU 内核和 5300 块物理硬盘）。第 11 章中的案例研究介绍了一个如何保持多个数据中心同步的例子。客户应用的实例当前托管在 OpenVZ 的容器中。请参考第 5 章中关于轻度烧制与重度烧制[⊜]和容器使用的讨论。使用当前最先进的容器技术已经取得成功，这有助于 Atlassian 按数量级降低客户的成本。今后，Atlassian 正在从一体化架构应用迁移到无租户的基于微服务的架构，在这个架构中前端服务器（通常与用户在同一个地理区域内）服务终端用户的请求，而通用业务逻辑和数据层在应用之间和客户之间共享。在这个迁移中，Atlassian 的目标是：

- 通过将数据和服务设置在高终端用户更近的位置来为客户提供更好的性能。
- 以更低的成本提高 Atlassian 云的可伸缩性以处理不断增加的客户数量。目标是以 Atlassian 数据中心内的现有基础设施来支持 2 倍于当前数量的客户。
- 为了更好的性能和更低的成本支持在适当情况下使用公有云供应商。

⊖ Confluence 是一个专业的企业知识管理与协同软件，也可以用于构建企业 Wiki。——译者

⊜ 此处烧制指对虚拟机镜像的管理方式。——译者注

- 在数据中心中断事件中支持更简单的数据灾难恢复。
- 提高部署的速度特性以服务客户。

Atlassian 计划在公有云供应商（例如 AWS）上部署一些这样的微服务，同时配置虚拟专用网络（Virtual Private Network，VPN）基础设施（例如 Amazon Direct Connect）来设置允许与 Atlassian 数据中心进行双向通信。应用实例在迁移过程中依然保留在 Atlassian 数据中心内。

对 Atlassian 的一个主要挑战是，变更必须不能导致中断或对终端用户造成更差的性能，特别是在迁移到微服务的过程中。因此，对新的微服务来说，许多微服务的关注点是需要的主要内容，例如：

- 确保数据迁移到正确的格式和位置，没有数据丢失的风险。
- 在推出微服务时，确保按需修改应用以支持新的微服务，没有潜在的功能缺失。
- 可以推出确保微服务中的新功能而无停机时间，并且在任何意外失败的情况下回滚。
- 为支持团队提供替代的支持工具。例如，当前的支持人员可以登录到客户容器来调试和访问所需的数据和日志。在共享服务的环境中，必须提供新工具来方便相同支持用例。
- 确保充分的性能监控和告警到位。对微服务来说，差的性能和中断可以影响许多客户，所以问题必须被尽早确定以减少解决时间。

13.2　构建部署微服务的平台

许多基础设施组件对所有微服务（例如部署平台、到 Atlassian 数据中心的网络连接、日志和监控等）来说都是通用的。这些组件合并为一个高可用性的微服务的平台即服务（Platform-as-a-Service，PaaS）以防止在技术和配置选项上的不必要重复和不一致性。当前，PaaS 运行在 AWS 基础设施上，构建 AWS 工具（例如 CloudFormation），当在需要的地方提供额外的功能时。就大部分而言，将底层云服务供应商从微服务开发人员中抽象出来。图 13-1 中的架构图展示了 AWS 中的 Atlassian PaaS 的主要组件。这些包括：DNS 服务的 Route 53、平衡部署在多个可用区（Availability Zone，AZ）上的 EC2 实例的多个服务之间的传入请求的弹性负载均衡（Elastic Load Balancing，ELB）、指标和报警的 CloudWatch。服务可以访问各种 AWS 资源，例如关系数据库（Relational Database Service，RDS）、DynamoDB、S3 和简单队列服务（Simple Queue Service，SQS）。通过在 Squid 代理部署的 ELB，将服务到因特网和虚拟私有云外面的 AWS 资源的出站请求定向到高可用性的多个可用区。而且，将日志消息发送到部署在 EB 前端的多个可用区中的 EC2 实例的 Elasticsearch/Kibana 集群。

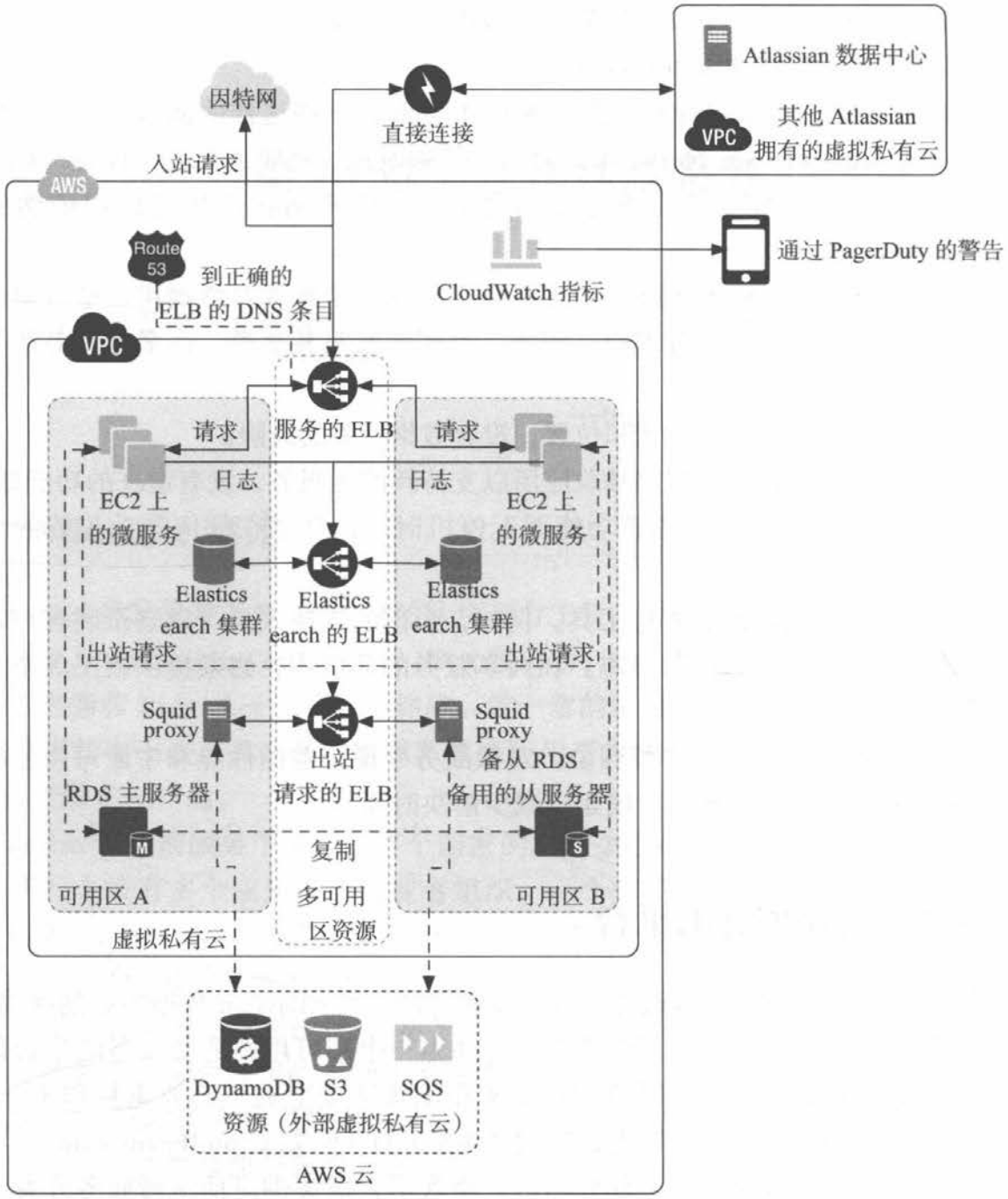


图 13-1　Atlassian 微服务 PaaS 中使用的组件 [标注法：AWS 简单图标]

Atlassian 的微服务 PaaS 提供了以下的功能：

- ***运行微服务的一致性容器***。微服务实例运行在具有烧制的 AMI 的一个 AWS EC2 实例上，AMI 由 PaaS 团队控制，而不是由微服务开发人员控制。AMI 包含了必需的运行时和 PaaS 基础设施。实例的规模可以由微服务开发人员来控制（例如，可以为 CPU 密集型微服务指定计算优化的实例）。为了将微服务部署到 PaaS 平台上，开发人员仅仅需要提供一个服务描述符，这个服务说明包含了服务配置和元数据（例如，需要的资源、环境变量）以及需要运行的工件（例如，JVM 服务的

一个二进制 JAR 文件或者一个 Docker 镜像）。

- 资源设置和管理，例如创建和管理 S3 桶、DynamoDB 表和简单通知服务（Simple Notification Service，SNS）主题。目的是从微服务开发人员抽取资源实现（例如，AWS 与谷歌），并支持对特定资源的更好管理。例如，AWS SNS 主题当前不能用 CloudFormation 模板来更新，必须重新创建。
- 微服务实例之间的自动伸缩和负载均衡。微服务开发人员仅仅需要在服务描述符和自动伸缩标准中指定所需实例的最少数量。Atlassian PaaS 将为每一次部署创建必需的 AWS 负载均衡器和自动伸缩配置。当前，使用带有自动伸缩组的 AWS 弹性负载均衡。
- 日志合并和搜索。微服务仅仅需要将日志条目输出到控制台（标准输出或者标准错误），最好是 JSON 格式。fluentd 自动收集和解析日志条目，并传递到 Elasticsearch 集群。开发人员和支持团队通过 Kibana 访问合并的日志，这允许搜索日志和对统计数据绘图（例如随时间变化的错误的直方图）。
- 指标收集、合并、报告和报警。标准的基础设施级指标（例如 CPU 负载、弹性负载均衡延迟和错误率），通过 AWS CloudWatch 和 Stackdriver 得到支持。当阈值超过重要指标时，通过 PagerDuty 触发警告。其他微服务特定的指标也是通过日志基础设施收集的，以便通过 Kibana 绘图。
- AWS 中的微服务之间的安全网络基础设施和 Atlassian 数据中心内的现有应用。这是在 Atlassian 数据中心内的应用与部署在 AWS 上的微服务（包括类似 Atlassian 内部 DNS 这样的服务）之间的双向高速连接。
- 零停机时间部署和支持快速回滚。微服务栈是无缝升级的，这包括创建一个新栈（例如，新的 AWS ELB、自动伸缩组和 EC2 实例），然后一旦微服务实例处于可服务状态就切换 DNS 条目（当前在 Route 53 中）到新栈。如果认为该升级是成功的，则删除旧栈；否则通过再次切换 DNS 条目回滚到旧栈。这个过程在第 2 章和第 12 章中的案例研究中有详细的描述。将 DNS 存活时间设置为较低的值（60 秒）以方便快速切换。而且，这些栈仅仅代表微服务本身。资源（例如，RDS 表或者 S3）是通过前面描述的资源设置机制来独立管理的。
- 多个环境以支持不同级别的部署和测试。PaaS 提供了 4 个独立的环境。这些环境的目的是保证每个应用或者微服务都历经每个阶段。因此它们正在对软件进行测试，这些软件将继续在下一个环境中运行。这 4 个独立的环境是：
  - 域开发。这个环境在开发过程中由微服务团队用于测试。
  - 应用开发。这个环境支持外部依赖性的测试（例如，JIRA 和 Confluence 等应用与其他微服务）。这个环境对 Atlassian 中的所有开发人员都是可用的。在这个环境中的微服务通常连接到所有 Atlassian 基于云的应用开发实例以支持“喂

狗粮”（dogfooding）[⊖]。

- 预发布。这个环境和生产环境高度相似。例如，在多个 AWS 区域都有部署。将 Atlassian 喂狗粮实例（即，Atlassian 内部使用的生产环境级实例）挂接在这些环境上。该环境主要用于测试生产环境配置。
- 生产环境。该环境支持客户 Atlassian 云实例。在两个独立 AWS 区域部署了一个 PaaS，在区域故障时它提供了高可用性。

13.3 BlobStore：一个微服务例子

选择开发哪些微服务是由为终端用户带来的价值（例如，优化的用户体验）和为 Atlassian 本身带来的价值（例如，对灾难恢复提供支持）驱动的。迄今，Atlassian 设计的微服务包括：单点登录（Single Sign-On，SSO）、合并账单、用户体验研究、文档转换、二进制对象存储和应用实例管理，在流水线中还有更多的微服务。作为一个规范的例子，我们将讨论 BlobStore，这是一个存储来自应用的二进制数据（blob）的微服务，二进制数据的例子包括 JIRA 问题的附件、Confluence 页面，甚至软件二进制文件。BlobStore 是一个早期微服务，是 Atlassian 的第 4 个微服务，是在 Atlassian PaaS 生产环境中运行的第 1 个微服务。

BlobStore 的主要业务驱动是能够实现更简化的客户数据灾难恢复。当前，客户的附件数据保存在运行客户实例的计算机节点旁边的存储节点上。在写作本书时，在两个数据中心之间大约有 40TB 的附件数据，因此支持灾难恢复包含时间密集型和资源密集型的操作——在数据中心之间传输该数据。另外，还存在一个挑战性的“打包”问题，因为复制的数据必须装入目标存储节点上的可用空间内。使用 BlobStore，应用程序二进制数据将存储在 AWS S3 中，使用 AWS 数据中心之间的 S3 复制，因此不再需要定制的数据中心之间的传输了，将客户实例故障切换到第二个数据中心就变得相当简单了。把客户数据保存在类似 S3 的存储上也使得在不同数据中心之间移动客户实例是可行的，以便为终端用户和他们的数据提供托管，并提供数据中心之间的更好的负载均衡。

从技术的观点来看，BlobStore 也代表了一个显而易见的和相对简单的组件，这个组件可以从所有 Atlassian 应用中抽象出来，也是向应用之间共享微服务的方向上迈出的逻辑上的早期一步。

13.3.1 架构

微服务带来的主要好处之一是，因为微服务规模小，所以可以采用对当前问题最适合的技术栈来开发它们，在实现方法上也可以非常多样。实际上，这也正是在 Atlassian

⊖ 指邀请公司员工参与测试，提供反馈和建议。——编辑注

中的情况，这里微服务当前是由 Java、Scala 和 Node.js 混合而成的。BlobStore 是一个小型的基于 Scala 的微服务，大约有 2500 行代码，由一个小团队开发和管理。BlobStore 使用了 Twitter 创建的轻量级 Finagle RPC 框架来为二进制数据暴露一个简单的 HTTP 应用程序编程接口（Application Programming Interface，API）。BlobStore 的核心组件如图 13-2 所示。将一组 BlobStore 服务器部署到 AWS 中的 Atlassian PaaS 上，在 S3 上存储二进制数据，在 DynamoDB 中存储键映射。为了从应用程序代码中抽取对 BlobStore 的访问（例如，在请求上增加必要的 HTTP 头部），并且为了保证在客户端上二进制数据是以流式传输的（而不是在内存中缓冲），在应用实例上安装了 BlobStore 客户端插件。

BlobStore 的核心是“内容寻址”存储。客户（例如 JIRA 或 Confluence）发送二进制数据和一个“逻辑键”到 BlobStore，这些二进制数据存储在 AWS S3 上，AWS S3 是由二进制数据的内容的 SHA-1 散列（称为内容散列）键控的。BlobStore 维护了客户的逻辑键和内容散列之间的映射关系。这种映射关系目前存储在 AWS DynamoDB 中，DynamoDB 是一个低维护、高可用和可伸缩的键值存储。客户可以通过 RESTful 资源基于逻辑键或内容散列来访问二进制数据。复制二进制数据和映射关系使它们在不同的 AWS 区域中是可用的。在两个 AWS 区域之间 BlobStore 服务器以主动 - 被动配置运行，因为这提供了来自 Atlassian 数据中心的最小延时。在 Atlassian 数据中心的 DNS 服务器上的 BlobStore DNS CNAME 条目将流量定向到正确的 AWS 区域中的 BlobStore 服务。

13.3.2　通过纯函数式架构和编程实现安全性和性能

BlobStore 是纯函数式的架构。其核心原则是，*数据是永远不可改变的*。这意味着，没有数据（二进制数据或表示键映射变化的记录）可以被销毁或者修改；相反，为了改变数据的当前表示，只是将“事实”追加到系统中。数据实质上存储在一个单一祖先的版本控制系统中。底层的不可变数据存储也使得它能够暴露不可变的应用程序编程接口——对于给定的请求，响应总是相同的。不可变的应用程序编程接口支持缓存、为较简单的业务逻辑提供请求的可组合性，以及较简单的测试，因为其中无需底层服务的知识就可以简单地模拟响应。使用函数式架构导致了 2 个核心设计决策，这些决策带来了很多好处。

首先，二进制数据是根据 S3 中的内容散列存储的。这意味着，对于给定的内容，S3 上的位置将总是相同的。一段内容可以上传到 BlobStore 多次，但是使用内容散列作为键，该内容在 S3 上将只需要存储一次。使用 SHA-1 算法保证发生冲突的概率是无限小的。这种方式不但支持重复数据删除，而且还支持对二进制数据的简单数据缓存：使用者既可以通过逻辑键也可以通过内容散列来请求二进制数据。通过逻辑键的请求重定向到这个内容散列的持久化 URL，当它对给定内容保持不变时该内容散列可以被缓存起来。这种缓存对于最小化 AWS 的链路之间网络延时影响是重要的。虽然从每个数据中心到 AWS 都是高速链路，但是在访问二进制数据时，其网络延时大于访问磁盘上文件的延时是不可避免的。BlobStore 能够使用 HTTP 客户端的本地缓存或缓存服务器（例如

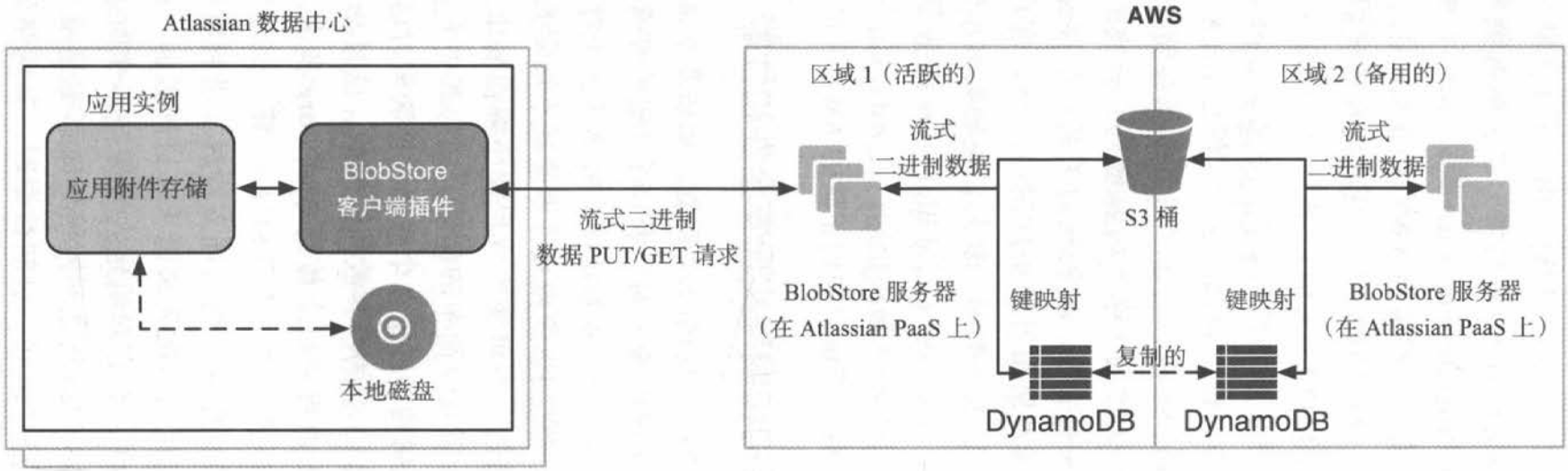


图 13-2　BlobStore 架构 [标注法：AWS 简单图标]

Atlassian 数据中心的 Varnish）来减小这种延时。删除二进制数据是快速和非破坏性的操作，因为它只需要删除对应的键映射而不是二进制数据本身。这意味着，在意外删除时，数据本身是可恢复的；在某个时间点后，没有映射的旧数据可以通过垃圾回收（Garbage Collection，GC）来删除。

其次，使用事件溯源模型来存储键映射。不是存储映射本身，存储映射上的“事件”（如插入或者删除事件）。为了恢复给定时间点上的一个指定的键映射，我们必须收集截至那个时间点的所有事件并按照顺序重放这些事件。在这个模型中，没有数据被删除，所以可以恢复因疏忽而删除的映射。我们仅仅查询在指定时间点上的映射的键映射存储。同样，对映射更改的审核追踪也无需额外的工作。另外，即使使用了如 DynamoDB 这样的简单且高可伸缩的键 - 值存储，事件溯源也提供基本的事务能力。具体地它不需要传统的关系数据库管理系统，关系数据库管理系统在跨区域复制的环境中是难以伸缩的。另一个好处是模式演变相对容易实施。使用 NoSQL 数据存储意味着，不再需要“停止全世界”[⊖]活动来执行模式更新。同样，新事件或者属性的变更可以通过数据版本感知的对象序列化控制程序（object marshaller）来处理（对象序列化控制程序总是以最新模式版本来表示数据），不需要更新或者重写现有数据。

除了函数式架构外，BlobStore 采用了函数式编程的概念。服务器用函数式风格的 Scala 语言编写，基本的概念是只要有可能就要保持变量是不可变的；新类型，包括表示逻辑函数或者运算的类型，根据需要创建，而不使用容易混淆的基本类型（primitive type）。这使得 Atlassian 能够通过编译时检查来测试代码而不是依赖单元测试或运行时检查，并带来了以下的好处：

- 减少了对显式测试的依赖，因为通过编译时类型检查，编译器极大地帮助确保代码的正确性。
- 代码是简洁的。
- 不可变性和并发性概念（例如 future）使得你能够更容易解决并发性问题。
- 纯函数式代码更易于单元测试，因为对于给定的输入，输出是确定性的。“模拟”（Mocking）是很少需要的，我们能够使用基于属性的测试（通过 ScalaCheck 和 Speces）来自动生成测试数据以帮助覆盖边界情况。

将函数概念应用于微服务开发的最大的挑战之一是已经为开发人员开发了适当的技能集合。虽然函数编程和函数概念已经出现了一段时间，但它们对于大部分开发人员来说通常还是不熟悉的（虽然这个情况随着 Scala、Clojure 和 Haskell 逐渐流行而有所改善）。BlobStore 团队是由具有函数式编程经验的开发人员组建的，因此他们能够让其他团队成员快速地学习。另外，在 Atlassian 中共享知识的文化有助于在公司范围内构建函数式编程和架构经验。

⊖ 这里指停止对外部的所有服务。——译者注

13.3.3 解决“非功能需求”

我们已经暗示了 BlobStore 如何设计和实施策略来解决伸缩性和可用性的要求。表 13-1 讨论了一些重要的期望的架构特征是如何实现的。

表 13-1 解决 BlobStore 的非功能需求

可伸缩性	BlobStore 服务器和客户端大量使用了流式传输。数据缓冲限制为大约 10kb，因此大到数 GB 大小的二进制数据可以安全和并发地传输而不会有内存溢出的风险。这使得 BlobStore 服务器能够支持大量并发的连接，服务器也能够部署在低内存实例上，或者与更多的内存密集型微服务共享实例 BlobStore 服务器也是无状态的，在请求之间不保留使用者的状态。当负载需要时可以启动新的服务器实例。服务器性能当前是受限于 CPU 的，所以当一个指定的 CPU 水平维持 10 分钟时，自动伸缩组启动新的实例。在低负载期间自动伸缩组也缩小该组
可用性	Atlassian PaaS 在不同的 AWS 区域中有独立的生产环境，在每一个生产环境中有独立的 BlobStore 部署。应用实例通过知名 DNS 主机名连接到 BlobStore 服务器部署，通过适当的 DNS 条目定向到活跃的部署。当一个 AWS 区域发生故障时，通过修改 DNS 条目将应用实例重定向到第二个 BlobStore 部署 键映射和二进制数据都在 AWS 区域之间进行复制（AWS 数据流水线用于键映射，AWS 内置的跨数据中心复制用于 S3），所以如果将应用实例移动到其他 Atlassian 数据中心（为了灾难恢复或者负载均衡），这些实例将访问它们的数据 BlobStore 使用高可用的数据存储（DynamoDB 和 S3），这些数据存储支持跨可用区的复制
安全	每一个请求都是经过身份认证和授权的以确保只有指定客户的应用实例可以访问客户的数据 另外，根据 BlobStore 有每个客户的二进制数据加密以保证数据的区分。当某个客户的实例受到安全破坏时，只有那个用户的数据是可以访问的。当整个 S3 桶的 AWS 访问密钥被泄露时，没有数据可以轻易地获取
可扩展性	BlobStore 服务器架构支持底层数据存储和键映射存储的方便替换
可维护性	BlobStore 全程采用了函数式编程技术，这使得它支持更简单的测试，并且类型的大量使用保证它更易于确信未来的代码变更是正确的，这些变更是由现有的或新的团队成员实施的

13.4 开发过程

Atlassian 中的大部分团队都是小规模的，最多大概 5 个开发人员，使用由团队自己决定的某种形式的敏捷或者精益开发方法。作为一个例子，BlobStore 开发团队由 5 个开发人员构成，使用一周冲刺的 Scrum。

团队通常也包括来自质量保证（Quality Assurance，QA）测试人员。与传统测试人员不同，QA 团队成员的角色是提供对测试方法和任务的测试和指导的监督。例如，QA 可能参与建立“开发人员在测试”过程（这里第二个开发人员确认测试代码，并且执行必要的手动测试）和评审测试用例的完整性，特别是在软件是由团队外人员开发的情况下。QA 也协助进行探索测试、设置测试事件（例如闪电测试）以便在部署大的功能变更前进行确认，并设置 QA 演示会话（演示整个解决方案并在更大范围的一组人员中进行端到端

的测试）。

Atlassian 特别是对微服务开发团队的一个主要挑战是，在团队之间维护可管理级别的交互。微服务对如何开发、部署和支持核心应用有重要影响。在团队之间还有部署依赖关系，例如在 PaaS 和 BlobStore 团队之间某些 PaaS 功能可能阻碍部署新 BlobStore 版本。传统上，这些交互的持续确认一直有点儿非正式，途径包括：由架构师和经验丰富的员工构建的知识、通过博客发表物和在内部 Confluence 实例上的页面及全员演示文稿在企业范围的知识共享。这种协作方式对共享和开发解决方案效果很好，但是最近，引入了通过项目经理的更加正式的途径来保证团队之间的依赖关系被更一致地确认和追踪。BlobStore 团队交互的一些例子包括：

- 来自产品团队的架构师的早期参与保证可以捕获相关的需求。可以正式捕获这些需求作为 JIRA 问题并定期评审它们。通常在产品和微服务团队之间有受控的交互用来减少噪声。架构师和团队负责人代表了信息流的主要导管。
- 随着产品任务列表的演进，产品和基础设施团队对 BlobStore 也提出了新的需求。在作为 JIRA 中的功能被追踪前，这些需求已作为非正式的讨论开始。
- PaaS 和 BlobStore 团队之间的密切交互作为新需求和问题被确认和解决，当 PaaS 和 BlobStore 并行推出时。这些团队使用了非正式聊天室交互和在 JIRA 中正式的问题追踪的组合。团队协作也是通过项目经理进行更正式的管理。
- 相关产品代码变更由产品和微服务团队共同评审。
- 其他团队可以自由查看微服务代码变更——这种方式提供了一种在团队之间共享知识和技术以及确认可能问题（如潜在的性能和安全问题等）的很好方式。

13.4.1　开发人员和支持

Atlassian 中的开发团队正在变得越来越需要负责将他们的软件部署到生产环境中。在 BlobStore 的案例中，开发团队负责将 BlobStore 发布到生产环境并按阶段将相关功能发布到客户应用实例，包括确认每个阶段的适合实例、开发必要的脚本和回滚过程、在客户实例上“翻转开关”，以及在错误条件下通过 PagerDuty 发出警告。这种更高级别的责任鼓励彻底测试、审核生产代码，已有访问底层数据的带内和带外的相关工具。

虽然 Atlassian 确实包含了一些 DevOps 概念，但是事实上完整的 DevOps 过程模型需要一个专门的服务运维小组来负责应用实例和底层平台的可用性，而 Atlassian 在过去几年已远离了这种实践。多个组共同承担运行时间的责任在快速解决事件时是不高效的，因此 Atlassian 做出了这种改变。直到应用或者微服务移交给服务运维小组时才认为它完全处于生产环境，在开发团队、基础设施团队和运维小组之间有明确的责任定义。运维小组是在发生事故时第一个被呼叫的团队，为应用程序和服务的底层平台提供 24/7 的支持。运维小组遵循开发团队提供的运行手册来进行初始事故解决，也可给服务所有者升级（一般包括开发团队中的成员）以便进一步调查。他们也负责服务指标报告（例如，服

务的正常运行时间、事故统计信息、是否达到了服务级别协议等）和支持基础设施（例如服务目录、和 JIRA 的警告集成以及报告系统等）。在移交前、移交过程中和移交后服务运维小组和开发团队之间都有协作。在移交前，开发团队必须提供运维用例所需的文档和工具。例如，使用 BlobStore，将附件数据的备份 / 恢复过程改变为涉及 AWS 资产，并且需要开发必要的文档和脚本。在移交过程中，服务运维人员开始度量服务指标，并作为第一呼叫人。在移交后，除了在需要时一起工作解决事故外，开发团队也参与事件后评审核。服务运维人员能够为开发人员提供指标，这些指标可以为新用户故事确认和任务列表优先级（例如可以减轻运维压力的新功能，或者缺陷修复的优先级）提供帮助。

还有专门的支持团队（客户提倡者）来负责面对客户的请求。虽然很多微服务可能是“在封面下的”，但不可避免的是，它们对终端用户有影响，因此需要在微服务开发人员和支持团队之间建立交互。例如，在特定情况下，终端用户的数据导入或导出可能需要是带外数（例如，如果他们的应用实例由于某种原因是不可用的）。从本地基于文件的附件存储迁移到 BlobStore 微服务意味着将不再易于通过标准“文件系统”机制来访问到应用附件数据，相反，该数据模型支持对附件的带外数据访问，并且已经开发了附带的工具。

另外，很多团队在开发生命周期中集成了“支持角色的开发者”（Developer on Support）（在第 1 章中称为*可靠性工程师*），其中团队成员也轮转为支持角色（例如，每次 2 星期）。在角色转换的过程中，开发人员与支持团队紧密合作来调查与某个特定软件相关的问题。“支持角色的开发者”这种方式在短期内确实减少了功能开发所需要的可用资源，但它的目的是，从更长期来看，一方面，对当班的“支持角色的开发者”参与到故障排除的需求变得更少，另一方面，通过更好的设计和工具化，开发软件够更容易由运维人员支持。

13.4.2 构建和部署流水线

BlobStore 代码和配置保存在 Git 存储库中（Atlassian Stash）。该团队使用了功能分支的策略。为每一个新任务或者故事创建一个功能分支，这个功能分支是短期的，一般存在 2 ～ 3 天。在任何时间点，通常都有一两个开发人员在一个分支上工作。为了合并到主干，增加一个功能分支的 pull 请求，但它必须经过没有在这个分支上工作的两个开发人员批准。也需要成功的分支构建。

已经创建了一系列的 Bamboo 构建和部署计划来实现对主干上变更的自动构建和测试并自动将这些变更部署到服务器和客户端代码的环境中。图 13-3 展示了在持续交付流水线中的关键步骤。注意，客户端插件的部署路径与 BlobStore 服务器的部署路径是不同的。

部署流水线的一些关键功能是：

- 流水线已经简化的，所以从合并到主干的时间开始在大约 1 小时内将代码部署到

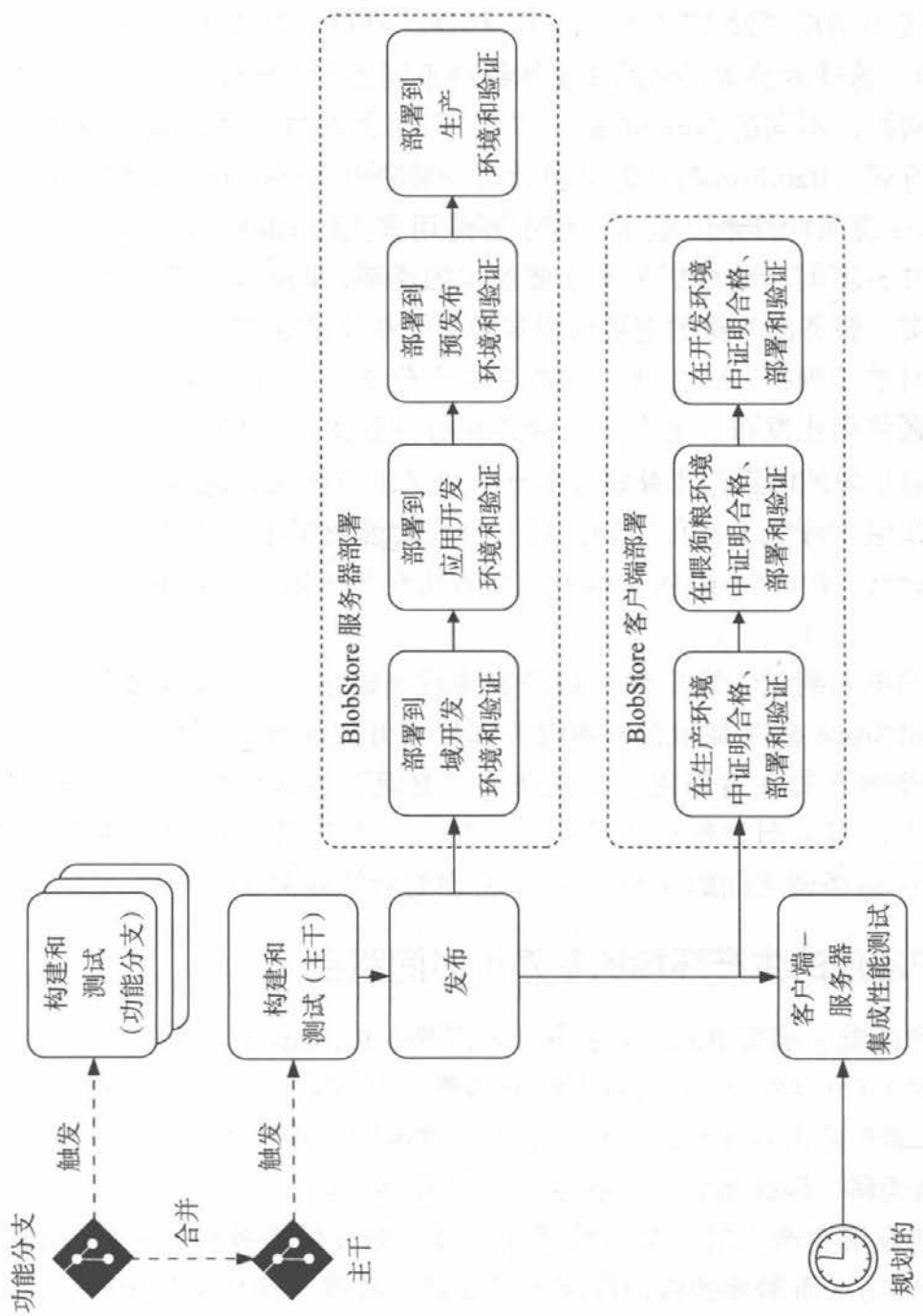


图 13-3　BlobStore 部署流水线 [标注法：业务流程建模标注]

生产环境中。在部署的早期阶段，通常一天 2 ～ 3 次将服务器代码部署到生产环境中。在大约 3 小时后 BlobStore 客户端流水线对 Atlassian 开发人员可用，因为它需要对应用进行更大量的测试。如果需要，发布到生产环境作为每周部署的一个部分。

- 在流水线的早期“发布”代码（即，构建、打标签和部署到集中的 Nexus 工件存储库中）。这样做是为了消除在流水线的不同点上的构建基础设施上发生变更的可能性（例如，不同的 Java SDK 版本）。不利之处是，这不能很好地适应 Bamboo 操作的方式。Bamboo 与作为部署计划一部分而执行的分支和发布很好地工作。
- BlobStore 遵循标准语义版本。这主要是用来支持 BlobStore 客户端并可能改变服务器 API；不间断的变更带来次要版本的递增，间断的应用 API 变更导致主要版本的递增。服务器本身的变更仅仅导致补丁版本的递增。
- 我们的目的是维护一个主线，也就是主干分支。当前的构建流水线说明主干分支上的问题将阻止发布。这个可以通过在合并到主干前彻底测试和保证主干构建的流水线总是绿的的规范来减缓。另一个可选的方法是，建立一个独立的“稳定的”分支，在这个分支上发布代码，只有当成功通过了主干构建后，主干才可以合并到稳定分支。BlobStore 团队认为这个方法对于小规模团队来说涉及了太多的额外工作。
- 自动化的单元测试、客户端 - 服务器集成测试以及与 AWS 资源的集成测试都必须在 BlobStore 客户端和服务器组件部署到开发环境前通过测试，这样以便目前团队以外的开发人员使用。在成功的“冒烟”测试后自动将服务器部署到后续的环境中，包括端到端的验收测试（测试从客户的应用到部署在恰当环境中的 BlobStore 服务器之间的完整路径）。定期进行性能测试。

13.4.3 客户应用的生产环境的零停机时间路径

除了在生产环境中部署 BlobStore 服务器以外，BlobStore 团队也负责推出客户应用中的必要变更到生产环境，但不能带来额外的停机时间。这作为两个独立的步骤并行执行：数据迁移和客户应用程序代码迁移 / 部署。这说明甚至可以在应用程序代码变更正在进行时执行数据迁移，因此缩短了达到生产状态所需要的时间。

通过部署到应用实例上的一个新数据迁移插件触发的带外操作来完成数据迁移。该插件独立于 BlobStore 所需要的应用程序代码变更。数据迁移是幂等操作，这意味着它可以在任何时间重新执行而不会重写任何已迁移的数据。因为二进制数据的内容散列是作为键映射的一个部分存储的，所以判断一个二进制数据是否已经迁移是很简单的是不需要比较二进制数据。在发布应用代码变更时，连续运行数周迁移，如果受控的和递增的数据迁移不会引起过量的网络和其他资源的消耗。

应用程序代码变更包括 3 种操作模式：仅本地的（即，无 BlobStore）、主要本地的

（即，异步操作 BlobStore）和主要远程的（即同步操作 BlobStore，回退到本地读缓存）。使用功能标志来启用这些模式，可以在运行时修改功能标志而不需要重启。临时的“主要本地”模式用于保证完整代码和到 BlobStore 服务器的网络路径是经过可靠性测试的，并收集性能指标来确定真实场景的操作延时和对终端用户体验的潜在影响。这种分阶段的方法使得我们在达到完全生产环境状态前对整个解决方案充满了信心。

发布的两个阶段，即数据迁移和切换功能标志，来启用代码变更，从“金丝雀”实例（附件大小和数量随机分布的大约 10 个实例）逐步开始。当成功后，将发布迁移到 100 个实例的组，然后是一个机架（大约 1000 个客户），随后迁移到两个数据中心的机架上，最后是在所有客户实例上切换。这个过程花费数周以保证 BlobStore 开发团队能够对问题做出响应而不会对客户造成严重影响。

13.5　BlobStore 演进

在这个项目进行过程中，BlobStore 微服务已经演进了，包括后端实现的重大变更和新功能的添加，例如每个客户的加密密钥、复制 / 移动二进制数据和字节范围的请求。到目前为止，这些变更都是平滑的，对客户应用代码的影响即使有也会很小。

一个好的例子是增加每个客户的加密密钥。最初，使用 S3 桶上的 AWS 服务器端加密来实现加密。在开发过程中，产生了支持每个客户的加密密钥的需求，这就导致了在 BlobStore 服务器内部执行加密的变化。最后，这需要大的服务器端变更（S3 键结构的变更）和对 HTTP API 相对小的变更，但是从客户应用的角度来看，不需要明显的变更。这些变更是通过以下方面来实现的：

- 在此期间，客户加密密钥是由 BlobStore 在一个实例上管理的，目的是在不久的将来将它授权给一个独立的微服务。这意味着客户应用不会感知到加密过程。
- 为了保持数据不可变性，在 S3 中的二进制数据同时基于纯文本的内容散列和加密密钥的安全散列来创建键。这使得保证数据仅仅能够使用正确的密钥来解密变得简单。如果提供了错误的密钥，那么在 S3 中的二进制数据是不存在的。这也使得支持相同内容同时使用多个加密密钥成为了可能，当客户的加密密钥随时间发生变化时这是非常重要的，因为它允许在重新加密过程中客户端使用旧的密钥访问数据，所以这个过程可以在线运行而没有停机时间。

Atlassian PaaS 和 BlobStore 服务器当前正在生产环境中服务客户。对所有的 Atlassian 云实例启用一个主要应用（JIRA）中的 BlobStore 功能。直到目前，还没有发现重大的缺陷或者生产环境事故。BlobStore 团队认为这个成果归因于 BlobStore 采用的“安全第一”的方法，具体地：

1）使用函数式架构和函数式编程来保证高的代码质量，避免了对数据的非故意损坏。

2）在开发数据和代码迁移发布计划时应小心。

3）关注解决方案的可支持性，包括开发必要的工具和指标。

4）自动化大部分测试和部署，包括性能测试以保证系统可以处理预计的负载。

总之，即使在当前微服务迁移的早期，我们已经观察到了大量的好处：

- 相对于在已有的一体化应用上进行变更，*微服务极大地缩短了部署新功能或者修复缺陷所需要的时间*。这是因为：
  - 可以快速地修改更小的、更集中的代码基。例如，在几天内就实现了BlobStore服务器端的增加或者删除功能的变更并将它们部署到生产环境，其中的大部分时间是等待审核。小的代码基意味着架构是简洁的，且代码有完整的测试覆盖率。
  - 使用微服务的良好设计接口，构建流水线仅仅包含了少量对微服务提供的具体功能的集成测试。这极大地减少了构建流水线的周期时间。
  - 对微服务的进行变更和部署将这些变更可以独立于客户应用。这意味着微服务是从应用部署周期中解耦的，而应用部署周期由于在整个一体化应用上运行综合集成测试将会变得更长。
- *资源的高度合并*。例如，我们能够用少量的BlobStore服务器实例来服务所有的现有客户，而不是占用每个客户容器内的资源（CPU、内存和磁盘等），后一种方式很难在客户实例之间共享。类似的场景是用Atlassian ID单点登录（Single Sign-On，SSO）。
- *能够使用更适合任务的非JVM的包*。通过将应用捆绑到Docker镜像并部署到PaaS，我们不再局限于一个JVM可以实现的功能（例如，在需要时，我们可以与内置工具集成以获得更好的性能）。
- *能够使用对特定微服务更合适的语言*。在实施时，我们不再局限于使用Java开发语言甚至使用Java虚拟机（Java Virtual Machine，JVM），这导致可以使用各种语言（例如，Node.js、Python和Clojure）新的微服务快速开发。

与所有新技术一样，虽然已经出现并解决了很多挑战，但是随着我们开始构建更多互相交互的微服务，更多的挑战正在逐渐出现。这些挑战包括：

- *微服务部署比具有更多移动部件的一体化应用更“复杂”*。虽然微服务代码基更小，但是需要一些额外的考虑和基础设施，例如新的部署流水线、网络、日志、指标收集等。这给早期微服务的开发和部署增加了很多时间。Atlassian PaaS大大地简化了这些问题，利用了早期微服务的大量工作，这极大地缩短了后续服务的开发时间。
- *新的架构关注点起到了作用，这与传统一体化应用相当不同*。很多关注点并不难实现。挑战在于保证所有工程师，包括在客户应用上工作的工程师，需要知道这些关注点。这些关注点包括：

- 需要无状态的业务逻辑来支持水平可伸缩性。服务在请求之间是无会话状态，并且需要谨慎处理，如果有的话，缓存数据。
- 需要服务支持多个租户。例如，服务需要能够为一个请求查找特定租户的数据、在租户之间划分数据并保证租户只能访问它们自己拥有的数据。
- 到微服务的额外跳导致网络延时增加。从短期的早期的微服务的短期内，使用不可变的 API 缓存是相对容易实现的。然而，随着我们开发服务的网络（其中请求遍历多个网络跳），这可能成为问题。需要采取更谨慎的措施来保证数据是尽可能地可缓存的。
- 微服务之间的低延迟身份认证和授权。微服务需要彼此之间的身份认证和授权请求，也可能需要模拟终端用户作为这些请求的一部分。以一种高性能的方式处理这个问题，特别是在请求遍历多个微服务（每个微服务都可能需要执行身份认证和授权检查）时，将是一个当前正在解决的挑战。然而，现有机制，例如 Kerberos，已经说明这是可能的。
- 在新环境中独立地部署和回滚微服务而无需大量长时间运行的一套集成测试。应该互相独立地部署微服务，然而变更需要与相同环境中的其他服务兼容。当前正在使用集成测试来验证这种情况，但随着微服务数的增加，这可能成为瓶颈。正在进行的工作是调查概念，如客户驱动的契约、关于测试运行的元数据维护和特定微服务版本组合的可靠性，但目前它仍然是一个未解决的问题。
- 确保微服务保持“小的”和功能聚焦的并且提供了良好定义的应用程序编程接口。在一体化应用中，在相同的部署单元内增加功能成为一个习惯，很容易将这个行为带到微服务开发中。为了解决这个问题，Atlassian SaaS 和应用架构师已经开发了一个定义微服务边界的路线图。同样，这在 Atlassian 内可能不是严重的问题，因为我们的一体化应用已经架构为高度模块化的，这种模块化是通过基于开放服务网关协议（Open Service Gateway initiative，OSGi）（当前已经是 HTTP 的了）的插件系统实现的。最后，因为新服务在线，所以微服务可能需要随着时间推移而重构以进一步拆分功能。具有良好定义接口的小的代码基使得这个过程变得相对容易。
- 对计划构建的微服务的依赖。因为范围小，所以微服务通常需要与其他微服务合作来实现全部功能。然而，在早期阶段，不是所有依赖的微服务都是可用的，这就导致了临时解决方案和变通方法。例如，客户数据加密和密钥管理严格上不在 BlobStore 微服务的范围内，但是在这样的服务变得可用前，需要开发临时解决方案。另外，当开发了一个适当的微服务时[⊖]，BlobStore 需要重构。这要求微服务重构的重构理念体系和时间范围。即便如此，因为它们的规模较

⊖ 指实现客户数据加密和密钥管理功能的微服务。——译者注

小，所以重构应该是相对容易的。

- 依赖其他项目（应用或者服务）。这个挑战存在于任何架构中，但是变得更加明显，因为微服务是“跨应用的”，所以有更多的触点。良好定义的应用程序编程接口和项目范围有助于清晰地确认这些依赖关系而因此可以对它们进行管理。而且，早期微服务工作（例如，Atlassian PaaS 和 BlobStore）的经验已经带来了过程优化，特别是围绕团队间的沟通和依赖追踪以及对敏捷方法论的输出的更好利用以确认延迟并适当地调整。
- 支持微服务与支持一体化应用是不同的。如前所述，处理微服务的工具和过程与支持一体化应用的工具和过程是相当不同的。这些工具和过程的开发需要被当作相关因素考虑微服务工作中。微服务开发团队、服务运维和支持团队之间的密切交互对于保障平滑的迁移给支持团队是极其重要的。正在开发运维团队适职过程以标准化和简化这个过程。除了一般支持外，调试微服务的分布式网络是完全不同的且比一体化应用更困难。需要进一步调查分布式追踪解决方案，例如 Twitter Zipkin，它通过为请求入口生成唯一的 ID 来追踪相关点上的请求。

截至目前，微服务带来的好处远远大于它的弊端，Atlassian 期望随着他们获得经验这些挑战将变得越来越少，构建模板和基础设施来加速部署，并减少一体化应用的大小。有很多因素允许 Atlassian 利用早期微服务的动力来实现成功的到完全无租户架构的迁移：

- 有许多紧迫的业务问题，包括如何减少每个客户的成本、提高应用的可伸缩性以便在能够快速交付新功能和应用的同时服务新客户。正如我们在大型的基于 Web 的业务（例如 Amazon、Netflix 和谷 Google）中看到的那样，如果可以正确地实施（即，保证问题在解决的同时不会因此累积大量的技术债务），那么共享的微服务代表了应对这些挑战的一个良好解决方案。
- 微服务开发、敏捷方法和 DevOps 概念产生了相似的结果：小型的高质量的功能块的每次快速交付。正如在这个案例研究中讨论的，敏捷和 DevOps 很好地嵌入 Atlassian 内的团队中，因此微服务，究其本质是可以快速交付的小的功能片段，是支持更快团队速度的自然演进。
- PaaS 的开发是在 Atlassian 内采用微服务的转折点。PaaS 为开发人员引入了一个方便的平台以便在其上进行研究和创新，与公有 PaaS 产品（例如 Heroku 和 Elastic Beanstalk）类似，虽然也提供了标准化的通用基础设施来帮助生产环境支持。因此，更广泛内部发布的几周内，开发基于不同技术栈的大量微服务。

在写这个案例研究时，正在将更多的微服务部署到 Atlassian PaaS，正在将 BlobStore 功能与其他 Atlassian 云应用和微服务（例如，新的文档转换微服务）进行集成。另外，Atlassian PaaS 将合并其他有用的基础设施组件，包括更好的集中式指标收集、密钥存储和 AWS 资产管理。还有流水线中的许多其他的核心微服务，例如租户管理、身份认证 /

授权以及外部的任务调度。采用真正的敏捷风格，Atlassian 首先采摘低悬的水果[⊖]，然后加快速度和在新挑战出现时采用这样的计划。

13.6　小结

可以递增地迁移到微服务架构。这需要确认服务之间的共性来进行初始迁移，并采用应用架构来利用这些通用服务。对 Atlassian 来说，它也需要开发微服务所依赖的 PaaS。这两个不同的开发流引入了协作的必要性，但是一旦已经部署了许多微服务后 PaaS 就变得相对稳定了。

BlobStore 是一个影响持久性数据的微服务，因此它的实现可能比其他类型的服务更加敏感。它是通过使其存储的二进制数据不可变的方式来实现的。这种不可变性支持回滚，也支持错误追踪，因为所有二进制数据的副本在垃圾回收前都是被保留的。

13.7　更多阅读材料

本章中提到的技术细节可以在下面的链接上获得：

- Amazon Direct Connect：http://aws.amazon.com/directconnect/
- CloudWatch：http://aws.amazon.com/cloudwatch/
- DynamoDB：http://aws.amazon.com/dynamodb/
- ELB：http://aws.amazon.com/elasticloadbalancing/
- Elasticsearch：http://www.elasticsearch.org/
- Finagle RPC：https://twitter.github.io/finagle/
- Fluentd：http://www.fluentd.org/
- Kibana：http://www.elasticsearch.org/overview/kibana/
- OpenVZ：http://openvz.org/Main_Page
- Route 53：http://aws.amazon.com/route53/
- RDS：http://aws.amazon.com/rds/
- S3：http://aws.amazon.com/s3/
- Simple Queue System：http://aws.amazon.com/sqs/
- Squid：http://www.squid-cache.org/
- SNS：http://aws.amazon.com/sns/
- Stackdriver：http://www.stackdriver.com/

⊖ 指通过很少努力就收获到的成果。——译者注

第五部分

走向未来

在这部分中，我们讨论在未来的几年 DevOps 可能如何发展。首先，我们已经对 DevOps 进行了数年的研究，我们将描述研究结果和方向。其次，我们将更广泛地推测 DevOps 的发展。

在第 1 章中，我们定义 DevOps 为一个过程集，它的目标是减少从变更提交与代码部署在正式生产环境之间的时间。业务过程管理社区数年来一直在进行过程挖掘和建模。我们的研究涉及从日志为单个 DevOps 过程创建过程模型，并使用这个模型来驱动过程的错误检测、诊断和恢复。我们将在第 14 章中讨论该内容。

DevOps 一直在快速发展中，并将继续发展。这种发展将触及 DevOps 活动的 3 个方面：组织的结构、过程定义和技术。在第 15 章中，我们将推测未来 3 到 5 年这些活动将如何变化。

第 14 章　作为过程的运维

Xiwei Xu 和 Min Fu

如果你不能将你所做的事情描述为一个过程，那么你不知道你在做什么。

——W. Edwards Deming

14.1　概述

如第 9 章所述，持续部署流水线不仅仅是另一个具有超系统特性的软件产品，它也具有强烈的过程特性。对于很多其他的操作，例如诊断、备份和恢复、升级，以及维护，也具有这样的特性。即使使用你喜欢的定时任务和脚本可能流水线化一组小工具——这是管理员世界中熟悉的概念。我们可以把运维界视作在应用和系统上运作的大量的面向过程的系统。这些过程不仅仅是顺序的，而且在过程和任务级都有大量的同时性和并行性。

本章的目的是讨论把运维当作过程带来的影响。“当作”指的是，你能够从现有的运维软件 / 脚本以及它们的日志中发现一个过程模型。你可以：

- 分析发现的过程模型以寻找改进的机会。
- 使用过程模型来监控各种运维的进展、检测错误并从错误中尽快恢复。
- 在反映活跃的运维过程的层级上设置监控阈值。你想在步骤级而不是整个过程级实现它，因为步骤级的监控能够更早地发现错误并恢复。
- 利用过程模型帮助其他活动，例如根本原因诊断。

执行这些活动是困难的，因为经常缺少正在发生的情况的上下文信息。过程模型和进展监控提供了所需要的上下文信息。过程模型也可以是关联不同事件和监控指标的集中位置，以便完善对运行时系统的理解。机会是丰富的。发现的过程模型可以被用来编排原始过程中的变体，也可以成为执行未来过程的应用的机制。

面向过程的系统可以视为工作流系统或者业务过程管理系统。这些领域的相关结果包括从日志和事件踪迹中挖掘过程模型、过程分析、运行时监控和预测、过程质量改进和人力密集型过程。你可以在第 6 章关于滚动升级的讨论中看到这个视角的例子。在本章中，我们主要讨论从工作流或者（业务）过程的视角来讨论运维过程。

你需要考虑的最后一个引导性问题是过程模型的抽象程度。过程模型是一组活动的规格说明，当这些活动执行时，导致完成所需的结果。这组活动可以在细粒度级（在执行过程的每一个步骤上）建模或者在粗粒度级（在过程中执行主要活动）建模。建模级取

决于发现过程模型的来源的丰富度以及你期望从该模型中获得的结果。与软件系统一样，过程模型可以从以下两个方面理解：运行时属性（性能、可靠性和安全性是 3 个重要的质量指标）；开发时间属性（互操作性和可修改性是两个重要的指标）。在本章中，我们主要讨论对运维过程（这些运维过程具有正确的过程模型规格说明）的可靠性所做的研究。其他视角包括保证过程（及其模型）的正确性、改进过程执行的性能，或者在期望的粒度级上高效地构建模型。

14.2　动机和概览

如第 9 章所述，可靠性是指整体部署流水线全局和它的单个部件在定义的时间周期内维护服务提供的能力。典型的流水线必须处理来自不同类型系统的不同类型的错误响应——从云应用程序编程接口（API）调用的错误码到配置变更的潜在不报告错误。在第 2 章讨论的云内的不确定性也引入了一些随机错误，这样以前是成功执行的脚本可能产生无效的输出。这意味着，防御式编程策略虽然重要，但是在这种情况下是不充分的。相反，我们提倡获得对过程的期望状态的理解，把期望状态与实际状态进行比较。本质上，这是本章讨论的方法的基础。运维过程的一些特点使得这个方法比其他通用业务过程更加易于处理。特别是：

- 运维过程仅仅操作少量类型的实体。在本章使用的滚动升级的例子中，操作的实体包括弹性负载均衡器（Elastic Load Balancer，ELB）、自动伸缩组（Autoscaling Group，ASG）、启动配置（Launch Configuration，LC）和虚拟机（Virtual Machine，VM）。在通用业务过程中，可能要操作大量不同种类的实体。
- 运维过程的时间期限（time frame）是以几十分钟来测量的，如果不是以几小时。这意味着，在几分钟内收集日志、检测错误和从错误中恢复是有用的。在通用业务过程中，时间期限可能更短。
- 运维工具通常生成高质量的日志，这些日志用于创建过程模型而没有许多噪声。

为了知道一个过程应该达到的期望状态，我们首先需要发现一个合适的过程模型。一旦我们完成了这个工作，我们就可以准备进行错误检测、诊断和恢复。我们通过分析过程成功执行的日志来发现过程模型。在我们已经实现了成功执行并生成相关日志后，离线进行这种发现。

在过程的执行阶段，我们比较过程的期望状态和过程的当前状态。这两个状态之间的任何差别都表示可靠性问题，并提供了恢复策略的线索。这些活动与过程（并发地）在线发生。

我们使用滚动升级的过程作为贯穿本章的运行例子。滚动升级把应用的新版本部署到服务中，一次部署一台或者多台服务器。它从服务中删除服务器（可能删除服务器），在那个服务器或者一台替换的服务器上加载应用的新版本，并启动新加载的服务器。在

第 6 章中，我们更详细地讨论了滚动升级。图 14-1 是从第 6 章中复制的，显示了在 AWS 上 Asgard 使用的滚动升级过程[⊖]。

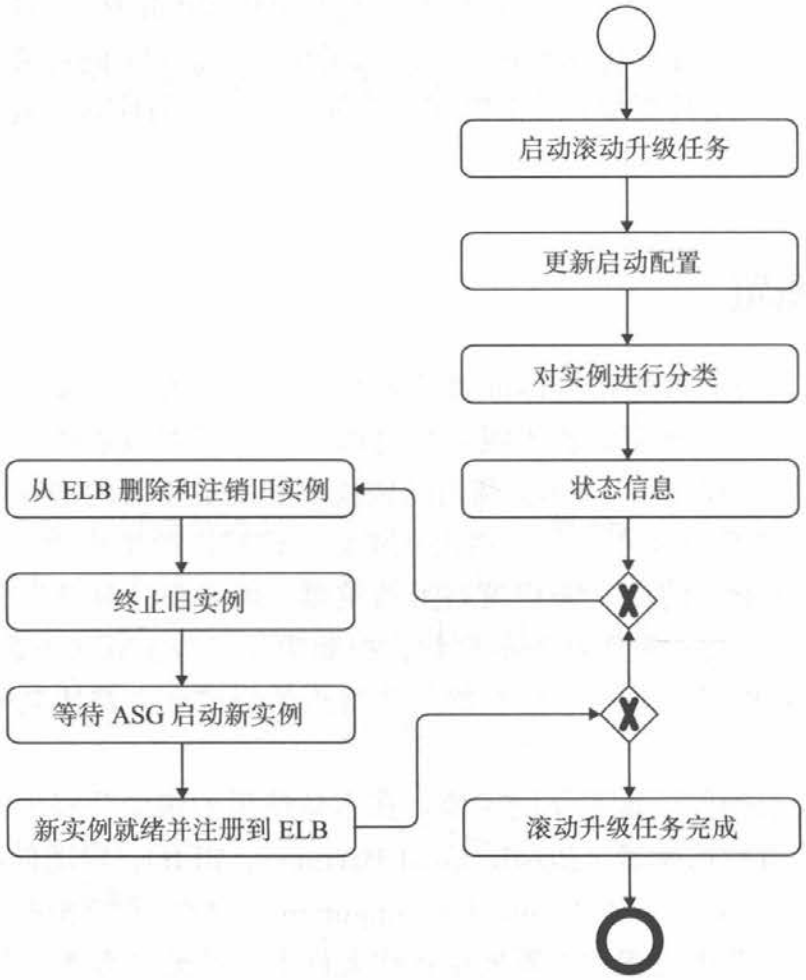


图 14-1 从 AWS 上的 Asgard 执行滚动升级过程（从图 6-2 复制）[标注法：业务流程建模标注]

14.3 离线活动

正如我们所说的，过程模型是离线创建的。它可以是被手动创建的，这种创建基于你对运维和代码 / 脚本的理解。另外，过程挖掘技术可以用于发现过程，特别是从日志中发现过程。在本节中，我们讨论过程挖掘活动，这些活动是离线执行的，它们基于过程的成功执行。这些活动提供了在线错误检测和恢复的基础。

有大量的理由支持我们更喜欢使用过程挖掘技术而不是手动过程模型创建。首先，自动化是技术采用的关键，它减少了创建过程模型所需要的技术水平。你将获得大量持续进化的运维。手动模型创建和后期维护导致高成本。其次，我们经常无法访问运维软

⊖ Asgard 是 Amazon Web Services（AWS）应用部署和云管理的 Web 接口，是基于 Web 的应用部署和云管理工具。——译者注

件的源代码 / 脚本，因此对这些软件的理解就必须通过外部观察到的蛛丝马迹（例如日志）来进行。再次，运行时日志可以用于触发测试和诊断，因为运维过程可以在不改变原始运维软件的情况下继续进行。

我们知道，在滚动升级中，将当前运行旧版本的 k 个实例（k 为一个较小的数字）一次从服务队列中删除，并用 k 个运行新版本的实例替换。通常需要几分钟的时间才能完成每一轮的替换。执行成百或者上千个实例的滚动升级使用一个较小值的 k 将花费很长的时间。

如程序清单 14-1 所示，Asgard 工具执行滚动升级并生成日志。

程序清单 14-1　Asgard 滚动升级产生的日志（缩减版）

```
"2014-05-26_13:17:36 Started on thread Task:Pushing
ami-4583197f into group testworkload-r01 for app
testworkload."
"2014-05-26_13:17:38 The group testworkload-r01 has 8
instances. 8 will be replaced, 2 at a time."
"2014-05-26_13:17:38 Remove instances [i-226fa51c] from
Load Balancer ELB-01"
"2014-05-26_13:17:39 Deregistered instances [i-226fa51c]
from load balancer ELB-01"
"2014-05-26_13:17:42 Terminating instance i-226fa51c"
...
"2014-05-26_13:17:43 Waiting up to 1h 10m for new
instance of testworkload-r01 to become Pending."
```

如果观察日志行，可以在不查阅源代码的情况下了解这个过程正在进行的操作。例如，这个软件正在推送 Amazon 虚拟机镜像（Amazon virtual Machine Image，AMI）到一个有 8 个实例的实例组中。AMI 包含该软件的新版本。计划是每次升级两个实例，直到所有实例升级完成。将旧实例从 ELB 中剔除 / 注销，并在系统在等待包含新版本的实例启动时终止执行。随后，将这个新实例添加并注册到 ELB 中（在程序清单 14-1 中未显示）。你期望循环执行替换步骤，直到将所有实例都升级到新版本。

过程挖掘技术支持从这些日志中发现图 14-1 所示的过程模型而不需要访问源代码。从日志中创建过程模型有两个基本步骤：1）根据日志代表的活动把它们分组并用活动名给它们打标签；2）利用类似 ProM 的工具使用已打标签的日志来创建过程模型。图 14-2 显示了存储在 Logstash（一个日志管理工具）中的 Asgard 的日志，然后这些日志用于生成过程模型。

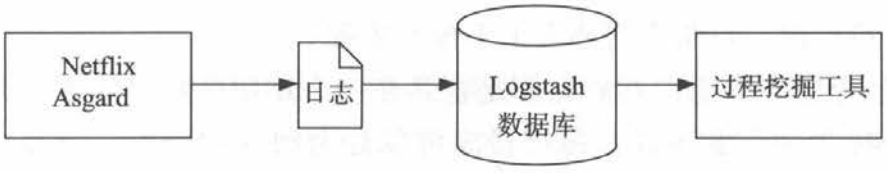


图 14-2　利用 Asgard 日志来产生过程模型 [标注法：架构]

Asgard日志并不是这种操作的日志信息的唯一来源。在AWS中，一项名为CloudTrail的功能将所有的Cloud API调用记录在日志中。Asgard滚动升级操作调用Cloud API来完成某些特定的步骤，例如注销、终止和启动实例。这些Asgard操作步骤在CloudTrail日志中留下了踪迹，但是这些信息是以较低级别的抽象存在的——API调用级别。一些其他步骤，例如“Status info”，不会包含任何Cloud API调用，因此它们不会在CloudTrail中留下任何踪迹。对相同运维过程的多个信息源进行组合或关联是可能的。这种关联可能不仅提供了更有效的过程模型，而且关联本身也可以用于把原因与影响相关联，也可以使用该信息进行断言、诊断，甚至恢复。程序清单14-2显示了一条来自CloudTrail的样本日志条目。注意，从这个日志条目可以确认被操作的AWS资源（虚拟机实例），以及与这个请求相关联的标识信息和参数。

程序清单14-2 样本CloudTrail日志

```
{ "awsRegion": "us-west-2",
  "eventName": "TerminateInstances",
  "eventSource": "ec2.amazonaws.com<http://ec2.amazonaws.
com>",
  "eventTime": "2014-01-24T01:59:58Z",
  "eventVersion": "1.0",
  "requestParameters": {"instancesSet": {"items":
[{"instanceId": "i-5424a45c"}]}},
    "responseElements": {"instancesSet": {"items":
[{"currentState": {"code": 32,"name": "shutting-
down"},"instanceId": "i-5424a45c","previousState":
{"code": 32,"name": "shutting-down"}}]}},
"sourceIPAddress":"autoscaling.amazonaws.com<http://
autoscaling.amazonaws.com>",
    "userAgent": "autoscaling.amazonaws.com<http://
autoscaling.amazonaws.com>",
    "userIdentity": {"accountId": "066611989206","arn":
"arn:aws:iam::066611989206:root","invokedBy":
"autoscaling.amazonaws.com<http://autoscaling.amazonaws.
com>","principalId": "066611989206","type": "Root"}}
```

图14-3显示了过程活动与CloudTrail日志如何基于时间戳相关联。这种关联使你能够确定在过程模型的哪个活动中对哪些AWS资源进行操作。而且，了解活动初期这些资源的状态并了解应该发生的操作类型使你能够确定在每个活动结束时AWS资源的期望状态。在14.4节，我们将对这个思想进行详细讲解。

在开发过程模型时，有两个需要人工干预的步骤：

1）人必须检查活动的组以判断它们是否是在一个期望的粒度级上，并且为这些组分配名字。从两个极端的方式来说，每行日志可以分为独立的一组，或者有一个包含所有日志行的唯一一个分组。选择正确的粒度级需要一些判断。

2）人必须检查生成的过程模型。在过程模型中存在虚假活动或者迁移是可能的。必须有人判断模型确实表示了被建模的过程。

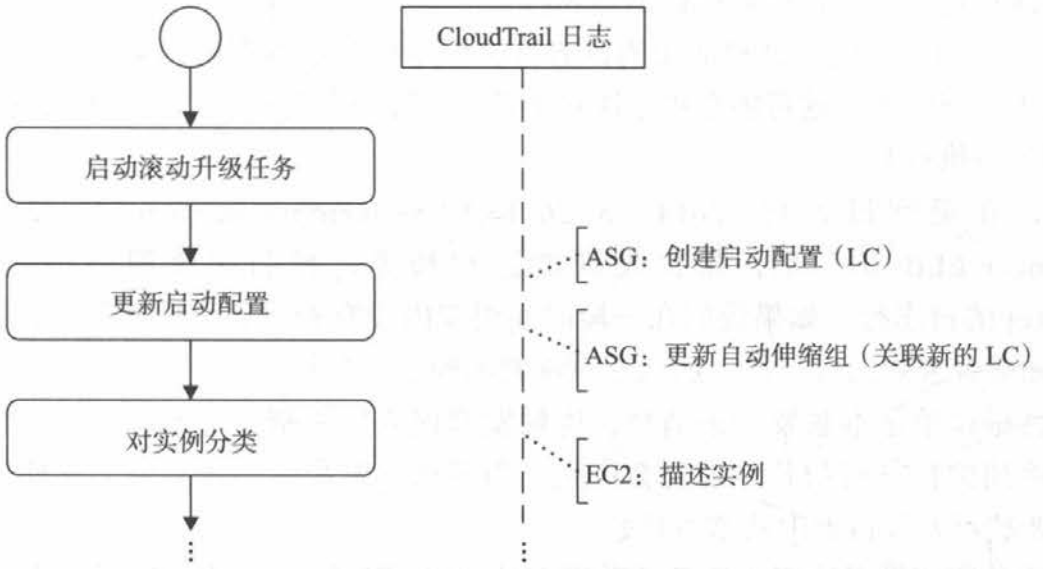


图 14-3　把 CloudTrail 日志与过程模型相关联以判断过程模型的每个活动所操作的 AWS 资源 [标注法（左）：业务流程建模标注]；[标注法（右）：UML 时序图]

对于一个熟练的分析师来说，创建过程模型的活动应该在一天内完成。

14.4　在线活动

我们知道，我们当前的主要焦点是滚动升级过程的可靠性。这意味着我们想要对滚动升级执行过程中出现的错误进行检测、诊断和从错误中恢复。错误检测和恢复可以在执行滚动升级的执行中在线完成。诊断是发生在检测错误之后的活动，它可以是在线的或者离线的。快速在线诊断可能带来更明确的恢复，但是对底层问题的详细分析最好是离线完成的，恢复后把系统恢复到稳定状态。

此时有些计时信息是非常有用的。可以创建和快速处理 Asgard 日志。另一方面，在 API 调用后 15 分钟内，当前 CloudTrail 日志是不可用的。这意味着错误检测和恢复只使用 Asgard 日志就可以继续进行。CloudTrail 日志对理解在每个活动结束时的 AWS 资源的期望状态是有用的，但是由于延迟，这些日志不能直接用于错误检测和恢复中被直接利用到。

14.4.1　错误检测

从 Asgard 产生的日志行中，我们可以检测每个活动步骤的开始和结束。根据过程模型，我们知道期望的步骤序列。一种错误检测模式是寻找期望序列以外的步骤。这样的序列以外的步骤叫作“一致性错误”。

一致性检查可以检测到以下类型的错误：

- 未知的：完全未知的日志行。
- 错误：对应于一种已知错误的日志行。
- 不适合的：对应于已知活动的日志行，但是根据过程实例当前执行状态，这个活动不应该发生。这可能是由于跳过的活动（过程向前执行）或者未执行的活动（过程向后执行）。

例如，在见到日志行“2014-05-26_13:17:38 Remove instances [i-226fa51c] from Load Balancer ELB-01”后，根据发现的过程模型，我们应该期望很快终止实例[i-226fa51c]的日志行。如果我们在一段时间周期内没有看到那个日志行或者看到了一个不同的已知或者未知的日志行，那么这意味着出现了某种错误。

一致性错误给运维触发一条消息，也触发错误恢复机制。这条发送给运维的消息在出现错误序列的日志行的几秒内就会产生。当手动诊断和错误恢复时，这使得运维知道在数千或者数百万行日志中从哪里开始。

第二种类型的错误检测依赖于活动操作的AWS资源。我们知道，通过把CloudTrail日志与过程模型相关联，我们可以确定每个活动所操作的AWS资源。例如，活动“从ELB中删除和注销旧实例”应该导致在该活动完成后在ELB上注册的实例数比活动开始时少一个。

在该过程运行时执行期间被删除的具体实例与在离线分析时被删除的实例是不同的，但是我们知道实例数都应该减少了一个。通过记录活动开始时的ELB的状态，并将它与活动结束时的ELB的状态进行比较，我们可以确定：1）事实上，一个特定的实例已经从ELB删除；2）该实例ID是已知的，因此在下一个活动“终止执行旧实例”中，可以精确地知道应该终止哪个实例。

滚动升级过程仅仅操作4类AWS资源：ELB、ASG、启动配置和虚拟机实例。这意味着，在活动开始时保存这些资源的状态是能够迅速完成的。而且，在活动结束时，我们可以通过查询AWS来确定这些资源的当前状态。这些查询请求的响应时间取决于AWS，但是我们的经验是响应时间大概是数秒。这两个时间意味着，把在活动结束时这些资源的状态与期望的状态进行比较可以在数秒内完成。而且，可以通过Asgard的独立操作过程来完成状态的保存和比较，因此对于正常的滚动升级过程并不会造成负面影响，除非检测到了错误。

通过这些方法可以检测的错误类型包括云故障导致的错误（例如长尾效应），也包括同时部署不同实例的两个团队之间的相互干扰导致的错误。我们已经检测到的错误类型如下所示：

1）在升级过程中改变的AMI。

2）密钥对管理错误。

3）安全组配置错误。

4）在升级过程中改变的实例类型。
5）在升级过程中 AMI 不可用。
6）在升级过程中密钥对不可用。
7）在升级过程中安全组不可用。
8）在升级过程中 ELB 不可用。

14.4.2　错误恢复

假设已经检测到了一个错误。我们有相关的 3 个 AWS 资源的状态集合：
1）上一个活动开始时的状态。
2）当前的错误状态。
3）期望的状态。

我们知道当前的错误状态是由于某些原因导致的。至少有两种从错误自动恢复的方法：回滚到上一个活动开始时的状态；或者前滚到期望的状态。

执行以上两种活动的困难随着环境的不同而有所变化。例如，假设没有 ELB 中注销旧实例。那么恢复可能涉及重试注销操作。另一方面，随着过程的复杂度提高，不可能返回到上一个活动时的开始状态。当暂停或者删除一个虚拟机时，它的 IP 地址就丢失了。在这个情况下，用原始 IP 地址恢复虚拟机是不可能的。

14.5　错误诊断

修复错误不可能修复错误的根本原因。例如，在前面关于错误检测的章节提到的一些错误是由竞态条件和发布冲突导致的，这些竞态条件和发布冲突是因为两个不同团队同时部署不同版本的系统引起的。如果没有解决这种冲突，那么系统可能还会再次发生相同的错误。因此，我们现在讨论诊断错误。

我们不关注软件本身的缺陷，我们关注云操作中的典型原因导致的错误诊断。诊断软件中的缺陷当然是重要的和有用的，但是这个主题不在当前我们的研究范畴内。为了诊断运维错误，我们使用*故障树*作为参考模型。在这样的故障树中，每个节点代表了一个故障或者错误，这个故障或者错误又相应地可能是由子节点代表的错误所引起的。图 14-4 显示了我们用来检测滚动升级错误的故障树的一部分。虽然构建这个故障树需要花费一定的气力，但这种努力是一次性的，而且该故障树也可以在许多不同的云运维中被重用。

对于过程进展的了解在诊断中发挥了帮助作用。知道错误发生在哪一步就限定了可能的原因，这些原因与在那个步骤中关联的 AWS 资源有关系。因此，我们可以裁剪故障树，只保留影响那些资源的因素而把其他因素排除在外。而且，曾经发生的错误类型的历史数据也使我们能够把故障树的每个分支与错误的可能性相关联，并使用那些可能性来指导诊断过程。

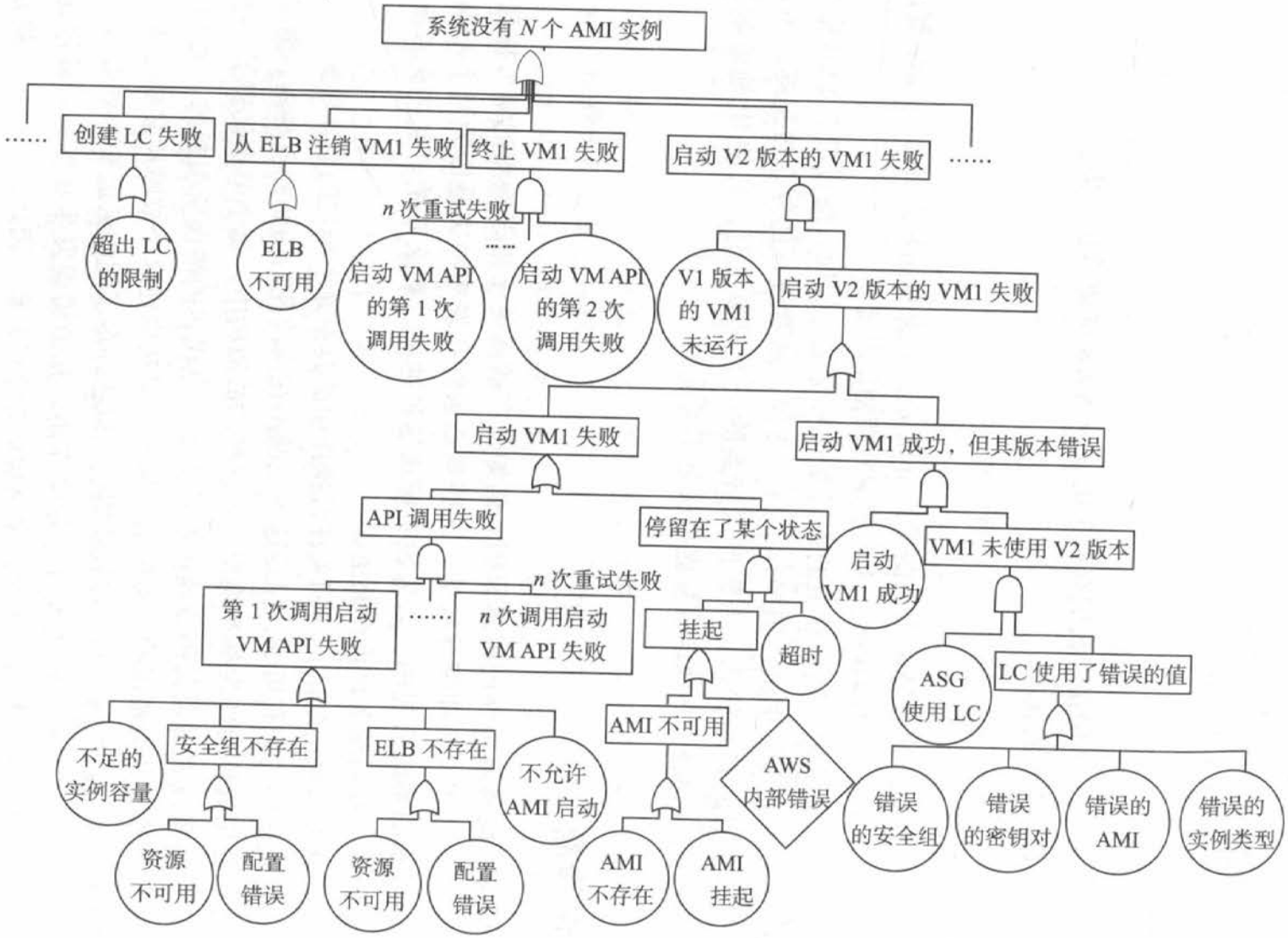


图 14-4 自动化的错误诊断的故障树的一部分 [标注法：故障树]

14.6　监控

如第 7 章所述，使用阈值进行警告和警报的问题之一是，如果阈值设置得过低，那么就会产生大量的误报。如果阈值设置得过高，那么就会导致漏报的概率增加——也就是说，遗漏实际的错误。正常的实践是调整阈值以实现一个可以接受的误报数。

在运维过程的执行期间警告或警报数增加，因为在这些过程中虚拟机正在从服务中删除或者添加到服务中。在这些运维进行时，有些组织关闭警告和警报，这样他们就不会被大量的警告和警报所淹没。

知道过程正在进行和知道过程的当前活动允许我们动态调整监控阈值。例如，如果你正在执行一个滚动升级，那么你就知道一个实例什么时候将从服务中删除。假定在这期间工作负荷相对稳定，从服务中删除一个实例将暂时增加其他服务器的工作负荷。例如，当一个服务器从服务队列删除后，我们可以相应地临时提高 CPU 阈值；而当新的服务器处于活跃状态并且分担了负载后，可以再降低这个阈值。

14.7　小结

在本章中，我们总结了过去两年我们所做的一些研究。把运维视作一个过程使我们可以从日志行创建一个过程模型，并使用这个模型来检测和（在某些情况下）修复由运维导致的错误。我们研究中的一个关键元素是，使用过程情境来提供信息以使我们能够确定由模型化的过程所操作的 AWS 资源的期望状态。知道期望的状态对错误检测甚至从这些错误中恢复提供了支持。而且，知道过程情境也使得我们能够动态调整监控阈值以减少在运维过程进行中产生的误报。

14.8　更多阅读材料

你可以在本书参考文献 [van der Aalst 11] 中找到更多有关过程挖掘的内容。

你可以通过阅读由 Xu 撰写的论文（详见参考文献 [Xu 14]）和 Weberl 撰写的论文（详见参考文献 [Weber 15]）来了解我们在错误检测、诊断和恢复领域所做的一系列研究。这些内容的更新在网站 http://ssrg.nicta.com.au/projects/cloud/ 上。

你可以在如下的链接上找到本章所提到的一些技术信息：

- AWS：“Error Codes—Amazon Elastic Compute Cloud,” http://docs.aws.amazon.com/AWSEC2/latest/APIReference/api-error-codes.html
- Logstash：http://logstash.net
- Asgard：https://github.com/Netflix/asgard

第 15 章　DevOps 的未来

关于未来的最好事情是它一天一天地到来。

——亚伯拉罕·林肯

15.1　概述

在本章中，我们猜想未来——这永远是一个危险的命题。首先，我们做出这样的假设——DevOps 会进入“跨越鸿沟”（crossing the chasm）的过程。“跨越鸿沟”一词是由 Geoffrey Moore 创造的，它描述了技术扩散过程。在这个模型中，创新者首先采用了一项技术。然后从早期采用者开始被主流采用。为了被主流采用，需要大量的资料来说服主流，使他们认为这项技术是值得投资的，这些资料描述了如何使用这项技术、成功案例和业务考虑。对主流采用所需要的资料要求称为“鸿沟”（chasm）。如果没有提供这些资料，那么这项技术将难以发展。从早期采用者开始，这项技术进入主流，并且被大量采用，最后是后期采用者和落后者。

对 DevOps 来说，创新者是 Google、亚马逊或者 Netflix。被创建的资料包括谈话、小型聚会和书籍（包括本书）、新工具、LinkedIn 组和博客。

现在，大量其他组织正在通过这种或者那种方式调研和采用 DevOps 实践。这些组织包括：

- 有实力的互联网公司，但他们不需要创新者的敏捷性。Atlassian（请参考第 13 章）就属于这类公司。
- 企业。大型和小型企业都在调研和采用持续交付和持续部署以及 Dev 和 Ops 的更好的集成。在第 12 章中的案例研究中能够找到与 Sourced Group 合作的这种类型的组织。
- 创业公司。创业公司的首要目标是获得客户基础。他们不是特别关心他们所使用的过程。他们的架构在一开始往往是一体化的，而且运维过程通常是由开发人员执行的。一旦创业公司规模变大，他们就寻找一个更加灵活的架构和更有组织的结构。在这个阶段中，他们开始采用 DevOps。

正如我们贯穿全书所讨论的，DevOps 触及组织的、过程的和技术的问题。在本章中关于未来的猜想正是基于这些问题来构架的。

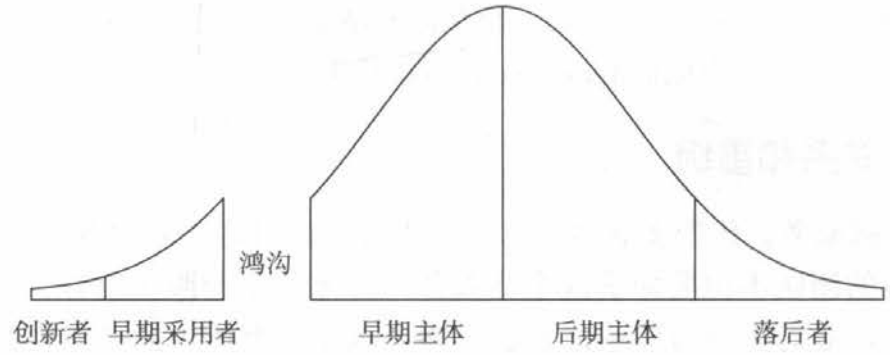


图 15-1　跨越鸿沟

15.2　组织问题

我们将讨论 3 个不同的组织问题：在 DevOps 活动中可能涉及的其他组、所有关系和重组，以及授权和控制。

15.2.1　DevOps 活动中可能涉及的其他组

DevOps 开始是作为一项打破 Dev 和 Ops 之间壁垒的运动而展开的，它的目标是减少把系统部署到生产环境中组织的惰性。有一些其他组会对 Dev 和 Ops 造成潜在的问题。

- 业务计划。业务计划影响一个组织的大部分领域。新产品、折扣、测试市场和供应链的变化都会对 Dev 和 Ops 产生影响。大部分成熟的组织都有现成的过程用于计划和管理这些类型的事件。如果减少了从开发到生产的时间，那么协调对业务计划负责的团队和对 DevOps 负责的团队将占据更大比例的全部时间。调解业务团队与 Dev 和 Ops 之间的协作将造成压力。这种“ BizOps”可以通过以下方式实现：将相关信息迁移到内容管理系统并使用该系统进行协作。
- 数据科学家。大数据用于获得对业务的了解，它可以驱动战略思考和实时客户获取。操作复杂的大数据分析集群、维护数据摄取流水线（data ingestion pipeline）和集成数据分析（机器学习模型和预测结果）到产品中将为 DevOps 的快速响应和一种新型的业务分析（大数据科学家）提出额外的要求。
- 安全。第 12 章中的案例研究包含了在建立持续交付流水线时安全组角色的讨论。第 8 章还讨论了安全审计。让安全过程和安全审计更加敏捷不仅是组织问题，而且也是监管机构要考虑的事情。在使合规自动化和合规性测试方面，有一些正在进行的工作。这些工作不仅仅可以应用在安全问题上，而且可以应用在与会计法规（accounting regulation）的合规性方面，例如萨班斯 - 奥克斯利法案（Sarbanes-Oxley）。
- 战略规划。持续部署流水线可能带来其他的业务机会。例如，Netflix 有机会开发

一套运维工具但选择不这样做，因为它不是它们的核心业务。其他组织可能开发其他的业务机会作为他们 DevOps 活动的结果。

15.2.2 所有关系和重组

小团队拥有微服务，但很多其他团队的应用可以调用这些微服务。当升级微服务时，拥有这个微服务的团队不但要保证这个微服务与全部的下游服务兼容，而且，更重要的是，还要保证它不会对上游服务造成负面影响。这是一个传统的依赖性问题，但是微服务的复杂性、规模和对实时性能的要求加重了这个问题。一种新的解决方案是，让上游用户的开发团队共同拥有下游服务的测试套件的所有权，让变更协商发生在测试套件中。另一方面，微服务环境中的端到端测试的代价是巨大的。许多团队的端到端测试的集体所有制是不现实的，因为人们不受限制地给这个测试套件添加新的测试，导致每次执行这些测试都会花费巨大的代价（因为需要的时间和经常出问题的测试）。

组织经常重组它们的结构。当组织经过一段周期已经开发微服务时，他们将拥有大量微服务，有些正处于开发中，有些仅仅处于维护阶段。当重组发生时，某个团队可能最终拥有一个他们不熟悉的微服务。这会引起常见的、长期的软件工程问题，这些问题涉及文档和知识转移。

而且，微服务可能是相当陈旧的，以至于它不能利用新兴的工具。例如，监控部分依赖于提供适当信息的应用。由新团队继承的旧的微服务可能不符合该团队生成监控数据的标准，也可能不具有符合质量标准的自动化测试。

15.2.3 授权与控制

授权与控制是另一个组织问题。一方面，有些关键决策（例如，发布和 A/B 测试）是授权给小团队和个人的，而不是通过层次化的、人力密集的审批流程来执行的。这种授权提高了速度，但是也给生产系统带来了重大的风险。解决这个问题的一种方法是，在流水线过程本身中增加自动化的质量控制。例如，在通过自动化测试和其他闸门之前，锁定生产环境。可以预见的两个潜在问题是：

1）*过程倾向于包罗万象*。面对问题的一个反应是给过程增加新的元素（例如流、闸门和结构）。随着时间的推移，过程失去了它们原来的动机，没有人回想起来为什么某个元素会出现在过程中。另外，元素中的功能存在的原因可能是已经发生了变化且可能不再是相关的，但是该元素还存在。

2）*过程的所有权*。开发团队可以拥有部署过程，或者组织可以拥有过程。在第 12 章的案例研究中，有一个组织范围的基础过程，每个团队可以基于这个过程进行裁剪。亚马逊有一个小组专门负责创建构建工具，这些工具提供了基本过程，其中可以对参数进行裁剪。而且，一旦发生重组，过程的所有权可以交给一个不熟悉这个过程或者它的基本原理的团队。

15.3　过程问题

未来将要出现的一些过程问题是旧的问题，例如关于厂商锁定（vendor lock-in）的顾虑，有些是因为使用了云作为平台而出现的（如计费模型问题），有些是因为部署的变更速度加快导致的问题。

15.3.1　厂商锁定和标准

持续部署流水线使用了许多不同的工具，并部署到一个平台上。所有这些工具以及平台都带来了厂商锁定的可能性。这不是一个新问题，而是一个在计算机行业中存在了至少 50 年的问题。但人们对这个问题的顾虑并没有因此而减少。

厂商锁定问题的一种解决方案是使用标准。虽然使用标准的语言和接口不能保证可移植性，但是它简化了问题。迄今，对于在持续部署流水线中使用的工具，仍然没有一个广泛采用的标准，尽管我们的预测之一就是随着 DevOps 跨越鸿沟，对标准的呼声也会越来越强烈。

在缺少标准的情况下，抵御厂商锁定的解决方案包括：

- 防御性编程（defensive programming）。在第 9 章中，我们讨论了可以用于支持可修改性的技术。在部署流水线中从一个工具转换到另一个工具就是一个可修改性的场景。
- 迁移程序（migration program）。通过构建一个专用的迁移程序，把程序从一个工具移动到另一个工具是有可能的。当目标概念是源概念的超集时，这类程序工作得最好。当没有直接映射时，可以使用在目标环境中枚举源概念的技术。这种枚举通常包含对目标上的特定参数选择做出的假设，它并不是一种最理想的解决方案，虽然这种枚举在人力与机器效率方面来看经常是划算的。

避免厂商锁定的另一种技术是，提供开放的应用程序编程接口（API）并为工具之间的互操作性推动建立一个活跃的插件生态系统。例如，Jenkins 就是一个流行的持续集成产品，它通过外部开发的插件来实现与许多其他工具的互操作。这使得人们能够选择他们最喜欢的版本控制、测试和依赖管理系统。

15.3.2　计费模型

在云平台上用于对资源计费的模型分为 4 类：

1）基于消费的。基于预先声明的计划，支付你所使用资源的费用。

2）基于套餐的。在一个特定的时间周期内（例如，一个月），一次支付可以无限使用或者在一定上限内使用。

3）基于广告的。允许在你的网页或者显示器上投放广告以换来费用的减少。

4）基于市场的。在特定时间，基于供求关系来决定支付多少。使用拍卖来为客户分

配资源。

这些计费模型的组合使用也是存在的。例如，AWS 基于分配的虚拟机的特征（基于消费的）按每小时进行计费（基于套餐的）。

在面对基于套餐的和基于消费的模型组合时，一种策略是使用虚拟机池，并通过使用容器来改变虚拟机的内容。如果使用了这种策略，那么自动伸缩规则将有两个级别。一个级别是在虚拟机内分配和回收容器，另一个级别是分配和回收虚拟机。容器模型也符合微服务架构，这里最好将微服务部署在容器内。

15.3.3 变更的速度

成功的持续部署流水线应该增加部署的频率，但这将带来一些后果：

- **与部署有关的质量将发生变化。**当每月只部署一次时，重要的是保证新版本是正确的并且部署过程是平滑进行的。部署所花费的实际时间不会引起大的关注，只要部署是在某个比较长的窗口进行的。当你每天部署 10 次，那么这些关注点将发生变化。对于部署来说，主要的质量关注点是避免中断。例如，及时发现的且没有对用户的服务造成影响的失败的部署，不是一个主要的关注点，因为下一个部署将在很短的时间内发生。我们的关注点是在重复失败的部署，而不是在某个特定部署的偶然失败。对无状态的服务来说，在发生失败时进行回滚是明智的策略，因为在下一次部署时变更又可以被应用。
- **自动化的错误检测 / 恢复是重要的。**当你每月只部署一次时，你有时间仔细地检测每次部署的正确性并在发生错误时手动排查错误。当你每天部署 10 次时，手动修复错误将变得不可行，因为这个错误可能也会影响其他部署。因此，监控和发现错误的工具以及自动化回滚的工具将变得重要。诊断工具也是重要的。理想情况下，诊断工具应该能够指出应用系统中或部署流水线中的错误来源。
- **工作负荷和应用行为变化。**通过持续交付，组织可以快速地在他们的产品中引进新的服务和功能。这将改变用户行为，继而改变工作负荷和流量模式。即使这种改变不是那么显著，但是当软件在升级到性能更优异或者更稳定的版本时，它对内部服务的消耗也会变得不同。通过使用历史数据和对系统的特定版本进行基准测试，可以得出一些经过仔细推敲的监控阈值。现有的监控系统正是依赖于这些阈值进行报警的。如果一个软件版本通常运行数周甚至数月，那么这种方式可能还是正常工作的。但是如果每天部署 10 次新版本，那么对新版本的阈值进行调整将是一个主要挑战。如果没有恰当的调整，那么可能导致很多误报产生。
- **环境变化。**当前的监控工具也假定运行应用的环境在本质上是保持不变的。频繁的部署可能破坏这种假设并导致另一种类型的误报。给定更高频率的部署，这意味着在部署过程中需要关闭监控工具，这个工作是非常频繁发生的。监控工具需要成为部署感知的并感知其他过程的干扰。

换言之，高的变更速度最终变成持续的变更，这将从根本上改变测试和监控的方法。组织可能更依赖于生产环境的金丝雀测试[⊖]（Canary test）和智能化监控，而不再依赖于耗时的端到端的测试。所有这些正在模糊化端到端的测试与监控之间的边界，所以它也要求全新的解决方案。另一方面，智能化监控可以进行预测分析以预先伸缩基础设施和应用，而自适应监控能够动态地改变被监控的实体以及使用各种情境信息的阈值。

15.4　技术问题

我们对技术方面的预测是，持续交付流水线将逐渐被视为一个实体而不是一组独立工具的集合。当前，持续交付流水线通常总是一个单独的工具链，每个工具拥有自己的脚本，这些工具通过一些集成脚本连接起来。这些集成脚本将发展，持续部署流水线的环境也将发展，这两者一起将形成一个集合，这个集合将视作一个具有多个部件的实体，而不是松散地连接在一起的部件的集合。当简单地通过脚本将现有工具集成起来时，你将面临的一些主要问题是：

- 贯穿整个流水线的可追踪性是缺失的。例如，要想知道将哪个特定的构建部署到哪个环境是困难的。有些主要的中断是由于把测试构建连接到生产环境组件所导致的。
- 缺少可追踪性导致错误诊断是非常困难的。生产环境的错误日志通常不含有关联到上游构建、测试或提交活动及工件的信息。建立这样的追踪不但花费大量的时间，而且手动操作也可能使得更智能的自动化恢复动作不再实用。
- 安全凭证经常需要在工具之间传递从而导致安全风险。云提供的安全功能增加了额外的复杂度。
- 很多现有的流行工具是在持续交付和持续部署时代之前创建出来的。使用这些工具自然地表示新概念和实践是困难的。

在发展过程中我们预见到的一些事情是：持续部署流水线概念的引入和各种流水线上的质量实现。

15.4.1　持续部署流水线概念

贯穿本书，我们已经讨论了特定于持续部署流水线运维的各种概念。在持续部署流水线规格说明中，这些概念应该置于首要状态。也就是说，集成脚本应该有支持这些概念的规格说明和执行的机制。下面，我们列举这些概念的最相关部分：

- 环境。通过流水线的移动提交代码主要是把该代码部署到不同的环境。如第 2 章所定义的，环境通常包含一个负载均衡器、在该环境中运行的系统以及必要的支

⊖ 金丝雀测试指面向一小组终端用户的小型测试，用以测试和验证变更。——译者注

撑实体（如数据库和配置参数）。另外，环境可能还包含其他服务或者模拟版本等。流水线工具应该能够描述环境、创建环境、把代码或从代码构建的可部署工件迁移到或者迁移出环境，以及在维护映射关系、踪迹和活动关联时拆解环境。对于生产环境，流量匹配和预热等的概念也应该是内置的，或者是规格说明所允许的。

- **部署**。在部署中有各种不同的关注点，这些关注点在流水线工具中应该是可以规范的，同时它们的具体实现细节是隐藏的并由不同的平台或特定的系统提供者提供。这些部署是：
  - **蓝/绿部署（或红/黑部署）**。指定一个蓝/绿部署应该只是指定一个需要部署的镜像和被部署的目标（虚拟机的数量、自动伸缩规则、虚拟机的位置）。支持基础设施应该关注：a）保证正确地安装新版本；b）在合适的时候把流量迁移到新版本上。旧版本的删除应该包含在这个概念中，或者作为一个独立的概念包含在拆解中。
  - **滚动升级**。指定滚动升级应该是指定需要部署的镜像、需要替换的实例和滚动升级的粒度。
  - **回滚**。回滚应该能够通过给定部署的ID来指定。通过知道前面的部署和它是如何指定的，所有其他参数都应该是可以获取的。
  - **金丝雀部署（Canary deployment）**。这两种部署风格都应该能够实现金丝雀部署。指定金丝雀的数量和它们替换的条件（随机的、基于客户的或者基于地理位置的）。
  - **A/B测试**。虽然金丝雀部署并行地运行两个版本来保证新版本是正确工作的，但A/B部署用于测试新版本（这里指的B版本）是否在某些指标上执行得更好，例如用户对特价的接受。
  - **功能开关**。功能开关的激活或者失活规格说明应该包含功能ID，不管它是被激活或者失活，以及变更的范围，用来指定应该切换所有的还是指定的服务器。
  - **拆解**。拆解环境的步骤是：删除它的虚拟机和其他资源，并从任何DNS条目中删除它。这个过程可以指定为独立于部署的活动，或者它可以包含在蓝/绿部署中。
- **监控**。在部署期间的监控包括监控流水线的性能，以及监控部署在预发布和生产环境中的应用程序的行为。流水线应该明确地通知各种监控工具有关应用程序和环境的变化。这些信息对于监控工具调整阈值和压制合理变更中产生的误报是必要的。
- **复制数据或版本**。虽然维护多个数据中心的组织数可能会缩减——它们更多地通过重构来充分利用云供应商的复制服务，但是仍然有一些组织会维护自己的数据中心或者混合私有云/公有云。应该能够在流水线工具中指定在数据中心之间进行同步。而且，也应该能够指定数据复制的类型和复制的频率。

- 流水线的服务级别协议（Service Level Agreement，SLA）。组织需要能够预测完成部署的每个阶段所需要的时间。例如，金丝雀部署可能影响 5% 的用户。推出金丝雀部署需要多少时间？推出剩余的部分需要多少时间？滚动升级可能节省 40 美金，但是比蓝 / 绿部署多花费 30 分钟。部署的声明式方法（通常用于配置管理工具，例如 Chef 和 Puppet）隐藏了部署的过程复杂性，但也引入了不确定性，这些不确定性表现在时间上（不可预测的收敛时间）和错误恢复上（只能使用最傻的重试机制）。把应用服务级别协议和流水线服务级别协议传递给下层系统对于声明式的部署方法来说可以获得更好的可预测性。另一种方法是引入过程视图，这个视图是在更高层的抽象。第 14 章对这种实践进行了讨论。所有这些都能够帮助运维和开发来计划和调度部署。
- 应用的服务级别协议。在部署应用的新版本时，可以保持和修改当前版本的服务级别协议。应该具有对应用服务级别协议指定这些调整的能力。
- 配置管理数据库（Configuration Management Database，CMDB）。配置管理数据库是集成的持续部署流水线的一个关键部分。因此，配置管理数据库对于复制、访问控制和使用提出了特殊的需求。当前，在典型的配置管理工具中的数据模型有一个有限的范围，不能满足于跨流水线的配置需求。让配置管理数据库成为首要概念并且让它具有更广的范围可以支持这些特殊需求。

15.4.2　在持续部署流水线中获得质量

使用集成的持续部署流水线，可以通过 3 个方面获得质量。

1）不要做任何愚蠢的事。你应该能够指定一系列约束来防止某些配置出现。例如，禁止除了正式环境以外的任何其他环境访问生产数据库。另一个例子，保证对某个特定区的部署有指定的配置设定。第 3 个例子，保证在一个环境中的任何变更都经过了特定的质量闸门。

2）自动化的检测错误、诊断错误和从错误中恢复。在第 14 章中，我们讨论了用于检测错误、诊断错误和从错误中恢复的技术，这些错误都发生在运维过程中。这些技术可以被自动化，同时可以成为集成的持续部署流水线的常规部分。另外，这些技术中的很多也应该与更智能的监控系统集成。运维过程情境在压制误报、检测小错误和诊断根本原因等方面起重要作用。

3）预测运维结束时间。我们将服务级别协议作为一个重要的概念。和这些服务级别协议相关联的是运维性能的监控和预测。当需要时，这将对部署计划提供支持。除了了解当前正在进行的或者已经完成的步骤之外，它还提供了对运维过程的可视化。

15.4.3　实现

实现存在各种可能性，如持续部署流水线。这种可能性涵盖了从定义一个新的工具独立的领域特定语言到创建一个 Ruby Gem。这个 Ruby Gem 集成了现有 Ruby Gem 中

已经存在的或者在其他工具集基础上创建上层结构的部署可能性。

不管实现策略如何，我们定义的概念应该是实施中的首要实体，并且实现流水线质量应该构建到流水线基础设施中。

15.5 错误报告和修复

在本书的开始，我们把 DevOps 定义为一组过程，这种过程的目的是减少从提交代码到将代码部署到正式生产环境的时间。

实现正式生产环境的最优和最高效方法是减少部署过程中漏报的错误数量。因此，在保证高质量的部署代码和检测部署后的错误之间的分隔线是不清晰的。

在第 5 章中，我们讨论了现场测试。现场测试是由 Simian Army 这样的工具完成的，该技术检测和修复错误而不把错误暴露给用户。当 Simian Army 的一个成员检测到错误时，错误的原因是已知的，因为本质上，这是在生产环境中进行的测试用例。

在形式化方法最终获得了足够的成熟度后，我们可以把它与微服务结合使用。形式化方法的目前水平能够证明小于 10 000 行代码的正确性。例如，第 13 章描述的微服务，它们的代码一般都少于 5000 行。因此，形式化的验证微服务看起来是可能的。而且，微服务之间仅仅通过消息传递来通信。这种简化的通信机制看起来也能够为支持形式化的微服务验证提供帮助。对于整个栈来说，形式化验证也许是不可能的，因为一些低层服务或协议具有内在的复杂性。

即使没有对微服务进行形式化验证，使用静态分析与执行过程中可用的钩子（hook）（例如，调试器）的组合看起来也能够快速地修复应用程序的错误。

15.6 结束语

对广泛采用 DevOps 来说，它现在正处于“鸿沟”中间位置，但 DevOps 的动量和增加的资料数量，将把它带到被主流采用的阶段，并将成为标准实践。采用 DevOps 所获得的有效性和效力将使得 Dev 和 Ops 能够以更快的速度向世界提供软件创新。目前，我们很高兴能够为推动 DevOps 跨越鸿沟而做出一些贡献。

15.7 更多阅读材料

维基百科 http://en.wikipedia.org/wiki/Crossing_-the_Chasm 对“跨越鸿沟”有所描述。

Ethann Castell 在 IBM 博客上撰写了云定价模型，请阅读参考文献 [Castell 13]。

参考文献

[3Scale 12] J. M. Pujol. "Having Fun with Redis Replication Between Amazon and Rackspace," July 25, 2012, http://tech.3scale.net/2012/07/25/fun-with-redis-replication/

[Agrasala 11] V. Agrasala. "What is IT Service?" December 6, 2011, http://vagrasala.wordpress.com/2011/12/06/what-is-it-service/

[Allen 70] T. J. Allen. "Communication Networks in R&D Laboratories," *R&D Management*, 1(1), 1970.

[Ambler 12] S. W. Ambler and M. Lines. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012.

[Ambler 15] S. Ambler. "Large Agile Teams," January 9, 2015, https://www.ibm.com/developerworks/community/blogs/ambler/?lang=en

[Barros 12] A. Barros and D. Oberle (Eds.). *Handbook of Service Description: USDL and Its Methods*. Springer, 2012.

[Bass 13] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 3rd Edition*. Addison-Wesley, 2013.

[BostInno 11] J. Evanish. "Continuous Deployment: Possibility or Pipe Dream?" November 21, 2011, http://bostinno.streetwise.co/2011/11/21/continuous-deployment-possibility-or-pipe-dream/

[Brutlag 09] J. Brutlag. "Speed Matters," Google Research, June 23, 2009, http://googleresearch.blogspot.com.au/2009/06/speed-matters.html

[Cannon 11] D. Cannon. *ITIL Service Strategy*. The Stationery Office, 2011.

[Castell 13] E. Castell. "The Present and Future of Cloud Pricing Models," IBM Cloud Products and Services, June 12, 2013, http://thoughtsoncloud.com/2013/06/present-future-cloud-pricing-models/

[Clements 10] P. Clements, F. Bachmann, L. Bass, et al. *Documenting Software Architectures, 2nd Edition*. Addison-Wesley Professional, 2010.

[Confluence 12] M. Serafini. "Zab vs. Paxos," March 28, 2012, https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab+vs.+Paxos

[Dean] J. Dean. "Designs, Lessons and Advice from Building Large Distributed Systems," http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf

[Dean 13] J. Dean and L. André Barroso. "The Tail at Scale," *Communications of the ACM*, *56*(2), pp. 74–80, 2013.

[Dumitras 09] T. Dumitraş and P. Narasimhan. "Why Do Upgrades Fail and What Can We Do About It?" Middleware '09 Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Springer, 2009, http://dl.acm.org/citation.cfm?id=1657005

[DZone 13] P. Hammant. "Google's Scaled Trunk Based Development," May 9, 2013, http://architects.dzone.com/articles/googles-scaled-trunk-based

[Edwards 14] D. Edwards. "DevOps is an Enterprise Concern," InfoQ QCon interview with Damon Edwards by Manuel Pais, May 31, 2014, http://www.infoq.com/interviews/interview-damon-edwards-qcon-2014

[Erl 07] T. Erl. *Service-Oriented Architecture: Principles of Service Design*. Prentice Hall, 2007.

[FireScope 13] "What is an IT Service?" November 12, 2013, http://www.firescope.com/blog/index.php/service/

[Fitz 09] T. Fitz. "Continuous Deployment at IMVU: Doing the Impossible Fifty Times a Day," February 10, 2009, http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/

[Fowler 06] I. Robinson. "Consumer-Driven Contracts: A Service Evolution Pattern," June 12, 2006, http://martinfowler.com/articles/consumerDrivenContracts.html

[Gilbert 02] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *ACM SIGACT News*, *33*(2), pp. 51–59, 2002.

[Gillard-Moss 13] P. Gillard-Moss. "Machine Images as Build Artefacts," December 20, 2013, http://peter.gillardmoss.me.uk/blog/2013/12/20/machine-images-as-build-artefacts/

[Hamilton 12] J. Hamilton. "Failures at Scale & How to Ride Through Them," November 30, 2012, Amazon Web Services.

[Humble 10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.

[Hunnebeck 11] L. Hunnebeck. *ITIL Service Design*. The Stationery Office, 2011.

[InfoQ 13] A. Rehn, T. Palmborg, and P. Boström. "The Continuous Delivery Maturity Model," February 6, 2013, http://www.infoq.com/articles/Continuous-Delivery-Maturity-Model

[InfoQ 13] E. Minick. "A Continuous Delivery Maturity Model," July 17, 2013, http://www.infoq.com/presentations/continuous-delivery-model

[InfoQ 14] D. Edwards. "Introducing DevOps to the Traditional Enterprise," June 18, 2014 http://www.infoq.com/minibooks/emag-devops

[InformationWeek 13] J. Masters Emison. "Cloud Deployment Debate: Bake Or Bootstrap?" October 30, 2013, http://www.informationweek.com/cloud/infrastructure-as-a-service/cloud-deployment-debate-bake-or-bootstrap/d/d-id/1112121

[ITSecurity 14] D. Raywood. "Shellshock hit our old unpatched server, admit BrowserStack," November 13, 2014, http://itsecurityguru.org/shellshock-hit-old-unpatched-server-admit-browserstack/#.VGVvZskhMuI

[Kandula] S. Kandula, G. Ananthanarayanan, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris. "Combating Outliers in Map-Reduce," Microsoft Research, http://research.microsoft.com/en-us/um/people/srikanth/data/combating%20outliers%20in%20map-reduce.web.pptx

[Kreps 13] J. Kreps. "The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction," December 16, 2013,

http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[Lamport 14] L. Lamport. "Paxos Made Simple," ACM SIGACT News 32, 4, December 2001, http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple

[Ligus 13] S. Ligus. *Effective Monitoring and Alerting*. O'Reilly Media, 2013.

[Lloyd 11] V. Lloyd. *ITIL Continual Service Improvement*. The Stationery Office, 2011.

[Lu 15] Q. Lu, L. Zhu, X. Xu, L. Bass, S. Li, W. Zhang, and N. Wang. "Mechanisms and Architectures for Tail-Tolerant System Operations in Cloud," *IEEE Software*, Jan-Feb 2015, pp. 76–82.

[Massie 12] M. Massie, B. Li et al. *Monitoring with Ganglia*, O'Reilly Media, 2012, http://ganglia.sourceforge.net/

[Mozilla] C. AtLee, L. Blakk, J. O'Duinn, and A. Zambrano Gasparian. "Firefox Release Engineering," http://www.aosabook.org/en/ffreleng.html

[Nelson-Smith 13] S. Nelson-Smith. *Test-Driven Infrastructure with Chef, 2nd Edition*. O'Reilly Media, 2013.

[Netflix 13] "Preparing the Netflix API for Deployment," November 18, 2013, http://techblog.netflix.com/2013/11/preparing-netflix-api-for-deployment.html

[Netflix 15] J. Kojo, V. Asokan, G. Campbell, and A. Tull. "Nicobar: Dynamic Scripting Library for Java," February 10, 2015, http://techblog.netflix.com

[Newman 15] S. Newman. *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, 2015.

[NIST 11] P. Mell and T. Grance. "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, NIST Special Publication 800-145, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[NIST 13] "Security and Privacy Controls for Federal Information Systems and Organizations," NIST 800-53, Rev. 4, April, 2013 http://csrc.nist.gov/publications/PubsDrafts.html

[OMG 11] "Business Process Model and Notation," Version 2.0, OMG, January 2011, http://www.bpmn.org]

[Puppet Labs 13] C. Caum. "Continuous Delivery Vs. Continuous Deployment: What's the Diff?" August 30, 2013, http://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff

[Rance 11] S. Rance. *ITIL Service Transition*. The Stationery Office, 2011

[Schad 10] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," Proceedings of the VLDB Endowment, 3(*1*), 2010.

[SEI 12] G. Silowash, D. Cappelli, A. P. Moore, R. F. Trzeciak, T. J. Shimeall, and L. Flynn. "Common Sense Guide to Mitigating Insider Threats, 4th Edition," December, 2012, http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=34017

[Seo 14] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. "Programmers' Build Errors: A Case Study (at Google)," *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014).

[Sockut 09] G. H. Sockut and B. R. Iyer. "Online Reorganization of Databases," *ACM Computing Surveys*, 41(*3*), Article 14, July 2009.

[Spencer 14] R. Spencer. "DevOps and ITIL: Continuous Delivery Doesn't Stop at Software," Change & Release Management

blog, April 5, 2014, http://changeandrelease.com/2014/04/05/devops-and-itil-continuous-delivery-doesnt-stop-at-software/

[Steinberg 11] R. A. Steinberg. *ITIL Service Operation*. The Stationery Office, 2011.

[Tonse 14] S. Tonse. "MicroServices at Netflix," August 8, 2014, http://www.slideshare.net/stonse/microservices-at-netflix

[van der Aalst 11] W. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[Weber 15] I. Weber, C. Li, L. Bass, S. Xu and L. Zhu. "Discovering and visualizing operations processes with POD-Discovery and POD-Viz," International Conference on Dependable Systems and Networks (DSN), Rio de Janeiro, Brazil, June, 2015.

[Xu 14] S. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun. "POD-Diagnosis: Error diagnosis of sporadic operations on cloud applications," International Conference on Dependable Systems and Networks (DSN), Atlanta, GA, USA, June, 2014. http://ssrg.nicta.com.au/projects/cloud/

DevOps

软件架构师行动指南

DevOps

A Software Architect's Perspective

DevOps为加速新软件功能的发布和改善对生产环境系统的监控带来了希望，但是对软件架构师和软件架构来说，DevOps的关键意义却常常被忽视。

Len Bass、Ingo Weber和Liming Zhu这三位卓越的架构师全面解决了这些问题。他们不仅研究了软件架构师为实现DevOps目标必须要做出的决策，并且说明DevOps的其他参与者有可能以哪种方式来影响架构师的工作，还详细介绍高效部署DevOps所需要的组织、技术和运营环境，以及DevOps对每个开发阶段的影响。作者解决了把多个功能关联起来的横切关注点问题，提供了对合规性、性能、可靠性、可重复性和安全方面的切合实际的洞察。

本书通过3个真实的案例讲解在不同场景下应用DevOps实践的方法：数据中心复制以保障业务连续性的实践、管理一个持续部署流水线的例子、迁移到微服务架构的案例。

本书全面覆盖了以下内容：

- 为什么DevOps要求在系统架构和IT角色这两个方面都做出重大的改变。
- 虚拟化和云计算如何使DevOps实践成为现实。
- 将运维和它的服务生命周期集成到DevOps中。
- 设计能够和DevOps实践完美配合的新系统。
- 将DevOps与敏捷方法和测试驱动开发集成在一起。
- 处理故障检测、升级规划和其他关键问题。
- 管理从DevOps的独立部署模型中衍生出的一致性问题。
- 将安全控制、角色和审计集成到DevOps中。
- 为DevOps的应用、推出和度量而准备业务计划。

上架指导：计算机\运维设计

ISBN 978-7-111-56261-0

9 787111 562610 >

定价：69.00元

Pearson

www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn