O'REILLY®

Broadview

混沌工程

Netflix系统稳定性之道

Casey Rosenthal
Lorin Hochstein
[美] Aaron Blohowiak 著
Nora Jones
Ali Basiri

崔慧 译

Chaos Engineering

中国工信出版集团 电子工业出版社

O'REILLY®

Broadview

混沌工程

Netflix系统稳定性之道

Casey Rosenthal
Lorin Hochstein
[美] Aaron Blohowiak 著
Nora Jones
Ali Basiri

姚秀功 译

Chaos Engineering

中国工信出版集团 电子工业出版社

O'REILLY®

混沌工程

Netflix系统稳定性之道

Casey Rosenthal
Lorin Hochstein
[美] Aaron Blohowiak 著
Nora Jones
Ali Basiri

傑杰○译

Chaos Engineering

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

在一个由很多微服务组成的分布式系统中，我们永远难以全面掌握发生什么事件会导致系统局部不可用，甚至全面崩溃。但我们却可以尽可能地在这些不可用的情况发生之前找出系统中的脆弱点。本书介绍了Netflix的工程师团队是如何根据多年实践经验主动发现系统中脆弱点的一整套方法。这套方法现在已经逐渐演变成计算机科学的一门新兴学科，即“混沌工程”。通过一系列可控的实验和执行实验的原则，混沌工程将揭示出分布式系统中随时发生的各类事件是如何逐步导致系统整体不可用的。

本书既适合研发、测试人员用来了解如何构建健壮的系统，也适合软件架构师用来了解设计创建高可用微服务体系的前沿方法，同时更适合在大型互联网或技术组织中专门负责系统稳定性的工程团队阅读。

©2017 by Casey Rosenthal,Lorin Hochstein, Aaron Blohowiak, Nora Jones, Ali Basiri,Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2019. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

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

版权贸易合同登记号　图字：01-2019-4197

图书在版编目（CIP）数据

混沌工程：Netflix系统稳定性之道 /（美）凯西·罗森塔尔（Casey Rosenthal）等著；侯杰译. —北京：电子工业出版社，2019.8

书名原文：Chaos Engineering

ISBN 978-7-121-36351-1

I.①混… II.①凯… ②侯… III.①分布式计算机系统－系统工程 IV.①TP338.8

中国版本图书馆CIP数据核字（2019）第071011号

责任编辑：刘恩惠

印　　刷：

装　　订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱　邮编100036

开　　本：787×1092　1/32　印张：3.75　字数：72千字

版　　次：2019年8月第1版

印　　次：2019年8月第1次印刷

定　　价：45.00元

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

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

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

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

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

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

序

要设计良好的系统需要考虑很多因素，比如可靠性、安全性、可扩展性、可定制化、可伸缩性、可维护性、用户体验等。为了更高效地支撑业务发展，越来越多的企业选择基于云服务或云原生理念来构建平台。采用新思路和新技术必然会带来系统架构和组织结构的变革，引入风险因素。如何通过实验证明生产环境下的分布式系统在面对失控条件的时候依然具备较强的“可观测性”和故障恢复能力呢？这就是混沌工程要解决的问题。

为了提升研发效率、支撑未来业务发展，2008年淘宝完成了服务化拆分和改造。伴随着应用数量的激增，多起因为不合理依赖导致的可用性故障发生了。作为保障高可用的技术团队，2011年我们开始尝试使用故障注入的方法来验证和治理系统的依赖问题，并在故障模拟实现、自动化验证、环境隔离、流量制造等方面进行了多次方案升级，沉淀了一套自动化的依赖治理方案。2016年，为了更真实地验证系统的容错设计和组织响应问题的能力，我们开始尝试在生产环境中进行故障场景演练。通过几年的发展，线上故障演练已经覆盖了大部分核心业务，提前发现了很多系统、工具、流程方面的问题。

也许很多人听说过Netflix的Chaos Monkey，但是大多数人对于混沌工程的概念还是比较模糊的。随着2017年本书的英文版面世，这种状况得到了改观。书中凝聚了第一批混沌工程师的智慧和经验，从建立稳定状态的假设、用多样的现实世界事件做验证、在生产环境中进行实验、自动化实验以持续运行、最小化爆炸半径五个角度对混沌工程进行抽象和概括，详细地阐述了混沌工程的演进和实践原则。书中的每条原则都既传递了一种思想，也代表着一套工具产品的设计思路。结合阿里巴巴在依赖治理、故障演练方面的实践积累，更能体会到每条原则的精妙。

近两年，混沌工程发展得很快，2017年被收录到ThoughtWorks的技术雷达中，2018年成为CNCF（Cloud Native Computing Foundation）

的一个技术领域，越来越多的企业开始计划引入混沌工程。作为混沌工程的早期实践者和推动者，有三点经验与大家分享：

第一，引入混沌工程，需要建立进行面向失败设计（可以使系统暴露出已有问题的设计）和拥抱失败的技术文化。

在思想上要认识到混沌工程的核心是通过引入一些风险变量去暴露已有问题，而不是创造问题。在恰当的时间和可控的爆炸半径下进行实验，有助于问题的发现和处理，降低潜在故障带来的影响。

传统的基础设施对稳定性和健壮性有非常高的要求，虽然降低变更频率可以减少故障，但这不是解决问题的根本方法。随着各项服务被迁移到云上，基础设施的管理被转移给云厂商负责，上层业务更需要做好面向失败设计才可以应对可能存在的极端情况。

第二，实施混沌工程，需要定义一个清晰可衡量的目标。

混沌工程的业务价值并不适合用过程指标（比如模拟了多少种实验场景、发起多少次实验等）来衡量，需要配合其他稳定性手段一起来衡量。比如，前期可以选择对历史故障进行复现，确保故障改进的有效性；中期可以选择监控发现率，验证故障发现能力和监控的完备程度；随着实施混沌工程的经验越来越丰富，后期可以考虑引入一些复杂的MTTR（Mean Time To Restoration）度量指标，比如故障的“发现-定位-恢复”时长这种综合性指标。

第三，推广混沌工程，要在控制风险的前提下不断提升效率。

越贴近生产环境的实验，结果越真实，同时风险也越大。大家可以先从一些简单的场景开始尝试，逐渐增加对系统和组织的信心。实验准备比实验执行更重要，混沌工程不是故障注入测试，要明确定义系统稳定状态和终止条件。

混沌工程是一种实践思想，本身不绑定任何技术或工具。不过在进行规模化推广和实施时，从真实性、开发成本、运维效率的角度考虑，还是建议复用一些成熟的开源组件或商业工具。

减少问题的最好方法就是让问题经常性地发生，通过不断重复失败过程并找出解决方案，来持续提升系统的容错能力和弹性。混沌工程作为一门新兴学科，还处于一个定义和被定义的过程。如果你对混

沌工程感兴趣，愿意去了解和实践混沌工程，非常推荐你从阅读本书开始行动。

周洋（花名：中亭）

阿里巴巴高可用架构团队高级技术专家

开源项目ChaosBlade发起人

2019.4

译者序

软件服务于人类的历史，历经了从单机软件在本地运行，到复杂系统通过网络提供服务的发展历程。软件的功能和质量每向前发展一步，都伴随着更多新的组成部分的加入。为了更好地服务于更多用户，进行规模更大、更复杂的运算，软件系统的能力需要不断进化升级，小型软件开始一步一步演化发展为大型分布式复杂系统。在软件系统变强的同时，越来越多的组成部分被加入系统，复杂性也在随之逐步增加。

每个软件从业者从写下第一行代码开始，就一刻不停地在和软件中的错误做斗争。开发和维护（修复缺陷、确保资源充足等保障软件运行的活动）是一对伴随软件运行而产生的双生子。热爱从零到一开发软件是开发者的天性，看着自己编写的软件完美运行，为其他人提供服务，一直是驱动开发者前进的动力。为了更快更好地开发软件，我们不断改进开发方法和软件架构，但是开发者在使用新的方法和更复杂的架构时，往往会低估潜在的风险。

近年来，随着系统架构逐渐向微服务架构演化，开发效率以及系统扩展性大幅提高。但同时，系统的复杂性也随之逐渐增长到了一个拐点，传统的测试方法已经不能全面理解和覆盖系统所有可能的行为，测试的有效性被大打折扣。我们通过各种测试、SRE、DevOps、金丝雀发布、蓝绿部署、预案、故障演练等方法，希望能够防患于未然。但服务规模不断增长，服务之间的依赖性所带来的不确定性也呈指数级增长。在这样的服务调用网中，任何一环出现的正常或异常的变化，都有可能对其他服务造成类似蝴蝶效应一般的影响。

软件系统自身复杂度的激增、开发者在引入复杂性的同时对风险的低估和忽视，是系统可用性面临的两大挑战。

为了应对这两大挑战，Netflix选择了一条不同寻常的路。从混乱猴子开始，Netflix为应对不确定性的领域带来了一种全新的思维方式——主动出击。这种主动出击的思维方式衍生出的一套实践方法，被称为混沌工程（Chaos Engineering），它旨在从根本上改变开发者应

对软件缺陷和故障的思维方式。在此之前，我们期望通过一系列的测试验证手段，尽最大的可能确保在线上运行的系统没有缺陷和故障。而混沌工程的理念认为这既不现实，也不符合系统自然发展的规律。混沌工程提倡我们首先要正面接受系统一定会存在缺陷，并且一定会时不时地发生故障的事实；然后，要求我们通过一系列实验找出可能发生问题的风险点，进而在不断加固系统的同时，促使开发者在开发软件时必须选择将防御性内建在系统中。

混沌工程的理论，建构于塔勒布在《反脆弱》一书中所阐述的思想之上，即系统如何在不确定性中获益。在接受“系统越复杂，越脆弱”的事实之后，让系统在每一次失败中获益，然后不断进化，这是混沌工程的核心思想。在实践中，混沌工程提倡用一系列实验来真实地验证系统在各类故障场景下的表现，通过频繁地进行大量实验，既使得系统本身的反脆弱性持续增强，也让开发者对系统越来越有信心。这个信心同时也是系统高速迭代，占尽市场先机的一个前提因素。

各个行业都涌现出了很多基于混沌工程应对上述两大挑战的实践案例。译者所从事的汽车金融行业是一个长链条，重流程，涉及获客、风控、审批、资金流转、贷后管理等多个环节的复杂业务体系。任何差错都有可能造成故障及问题数据的蔓延，轻则导致各种程度的业务不可用，重则可能会造成重大资损（资产损失）。传统的各类方法已经无法保障这样一个大规模系统的可用性和正确性，因此我们在2018年下半年开始采用混沌工程的思想，实践了若干方法，目前看来已经初见一些成效。在混沌工程原则的指导下，我们为实施线上实验开发了实验组的控制功能，从而可以将线上实验的影响控制在最小的范围，即最小化爆炸半径。这为我们在线上放心进行实验提供了基础保障。举一个线上实验的例子：我们通过切断对第三方存储服务的依赖，验证了备用存储服务是否能够无缝接管。第一次执行实验的时候，我们发现虽然在移动端上传的图片可以成功存储在备用存储空间里，但是在业务链条的末端有个别读取服务却没有能够正确地从备用存储空间中获取到该图片。实验只影响了一小部分流量，快速修复后，实验可以进一步在满流量的情况下运行。接下来，通过每天自动运行这样的实验，我们可以确保任何后续对系统的变更都不会引入新的系统行为盲点。因为我们会每天自动运行大量实验，开发者不得不

在编码时思考“我的代码如何在这些混沌实验场景下存活下来”，并逐步提高质量，形成正向循环。

混沌工程目前还是一个新兴的学科，它为软件工程行业带来了全新的思维方式。相信越来越多的实践和工具，会一步步释放这个学科的能量，吸引更多的实践者。团队或组织可以在任何时间点引入混沌工程的理念，这是一门实践性的学科，所以现在就和我们一起，开启你的混沌之旅吧！

译者介绍

侯杰，美利金融集团技术副总裁，TGO鲲鹏会会员，毕业于南京大学；曾就职于IBM中国、IBM澳大利亚和iClick（爱点击）；在多个行业的大型组织机构中负责过研发和管理工作，拥有十多年大规模分布式信息系统的设计、研发和实施经验。

技术审校者

周洋，花名中亭，阿里巴巴高可用架构团队高级技术专家，混沌工程布道师，开源项目ChaosBlade发起人。具有多年高可用保障、产品研发和系统架构经验，曾担任2015年双11稳定性负责人。目前负责高可用技术云化输出，并担任应用高可用服务（AHAS）及集团突袭演练负责人。

第一部分　混沌工程介绍

> 混沌工程是一门新兴的技术学科，它的初衷是通过实验性的方法，让人们建立复杂分布式系统能够在生产中抵御突发事件能力的信心。
>
> ——*Principles of Chaos Engineering*

只要你有过在生产环境中实际运行一个分布式系统的经历，你就应该清楚，各种不可预期的突发事件是一定会发生的。分布式系统天生包含大量的交互、依赖点，可能出错的地方数不胜数。硬盘故障，网络不通，流量激增压垮某些组件……我们可以不停地列举下去。这都是每天要面临的常事，任何一次处理不好就有可能导致业务停滞、性能低下，或者其他各种无法预料的异常行为。

在一个复杂的分布式系统中，我们单靠人力并不能够完全阻止这些故障的发生，而应该致力于在这些异常行为被触发之前，尽可能多地识别出会导致这些异常的、在系统中脆弱的、易出故障的环节。当我们识别出这些风险时，就可以有针对性地对系统进行加固、防范，从而避免故障发生时所带来的严重后果。我们能够在不断打造更具弹性[1]系统的同时，建立对运行高可用分布式系统的信心。

混沌工程正是这样一套通过在系统基础设施上进行实验，主动找出系统中的脆弱环节的方法学。这种通过实验验证的方法显然可以为我们打造更具弹性的系统，同时让我们更透彻地掌握系统运行时的各种行为规律。

混沌工程的应用可以简单到在STG环境[2]的某个实例上运行kill -9[3]来模拟一个服务节点的突然宕机，也可以复杂到在线上（生产环境中）挑选一小部分（但足够有代表性的）流量，基于这部分流量按照一定的规则或频率自动运行一系列实验。

混沌工程在Netflix的发展历程

2008年Netflix开始将服务从数据中心迁移到云上，之后就开始尝试在生产环境中开展一些系统弹性的测试。过了一段时间，这个实践过程才被称为混沌工程。最早被大家熟知的是“混乱猴子”（Chaos Monkey），因为其在生产环境中随机关闭服务节点而“恶名远扬”。进化成为“混乱金刚”（Chaos Kong）之后，这些之前获得的小规模益处被扩大到非常大。规模的扩大得益于一个叫作“故障注入测试”（Fault Injection Test，FIT）的工具。我们随后确立了混沌工程的若干原则，用于将这个实践进行规范并学科化，同时推出了混沌工程自动化平台，使得我们能够在微服务体系架构上7×24小时不间断地自动运行混沌工程实验。

在开发这些工具和进行实践的过程中，我们逐渐意识到，混沌工程并非是简单地制造服务中断等故障。当然，尝试破坏系统和服务很简单，但并不是全都可以有建设性地、高效地帮我们发现问题。混沌工程存在的意义在于，能让复杂系统中根深蒂固的混乱和不稳定性浮出水面，让我们可以更全面地理解这些系统中的固有现象，从而有根据地在分布式系统中实现更好的工程设计，来不断提高系统弹性。

本书介绍混沌工程的主要概念，以及如何在组织中实践这些概念和经验。也许我们开发的相关工具只适用于Netflix自身的业务和系统环境，但我们相信工具背后的原则可以被更广泛地应用于其他领域。

---

[1]译者注：系统应对故障、从故障中恢复的能力。

[2]译者注：STG环境是系统发布流程中的多个环境之一。完整的部署环境一般包括开发环境、测试环境、STG模拟或登台环境（Staging Environment）及生产环境。STG环境一般被认为是最接近生产环境的部署环境，通常STG环境可能会使用一些生产环境的服务或数据库。

[3]译者注：kill -9是一个Linux命令，kill命令用来终止一个进程，加上-9参数表示强制终止进程。

第1章　为什么需要混沌工程

混沌工程是一种通过实验探究的方式来让我们理解系统行为的方法。就像科学家通过实验来研究物理和社会现象一样，混沌工程通过实验来了解特定的系统。

如何使用混沌工程提高系统弹性呢？混沌工程通过设计和执行一系列实验，帮助我们发现系统中潜在的、可以导致灾难的或让我们的用户受损的脆弱环节，推动我们主动解决这些环节存在的问题。和现在各大公司主流的被动式故障响应流程相比，混沌工程向前迈进了一大步。

混沌工程和测试的区别

混沌工程、故障注入和故障测试在侧重点和工具集的使用上有一些重叠。举个例子，Netflix的很多混沌工程实验的研究对象都是基于故障注入来引入的。混沌工程和其他测试方法的主要区别在于，混沌工程是发现新信息的实践过程，而故障注入则是基于一个特定的条件、变量的验证方法。[1]

例如，当你希望探究复杂系统会如何应对异常时，会对系统中的服务注入通信故障，如超时、错误等，这是一个典型的故障注入场景。但有时我们希望探究更多其他非故障类的场景，如流量激增、资源竞争条件、拜占庭故障（例如性能差或有异常的节点发出错误的响应、异常的行为、对调用者随机返回不同的响应等）、非计划中的或消息内容非正常组合的处理等。如果一个面向公众用户的网站突然出现流量激增的情况，从而产生了更多的收入，那么我们很难将这种情况称为故障，但我们仍然需要探究清楚系统在这种情况下会如何变现。和故障注入类似，故障测试是通过对预先设想到的可以破坏系统的点进行测试，但是并不能去探究上述这类更广阔领域里的、不可预知的、但很可能发生的事情。

我们可以描述一下测试和实验最重要的区别。在测试中，我们要进行断言：给定一个特定的条件，系统会输出一个特定的结果。一般来说，测试只会产生二元的结果，即验证一个结果是真还是假，从而判定测试是否通过。严格意义上来说，这个实践过程并不能让我们发掘出系统未知的或尚不明确的认知，它仅仅是对已知的系统属性可能的取值进行测验。而实验可以产生新的认知，而且通常还能开辟出一个更广袤的对复杂系统的认知空间。整本书都在探讨这个主题——混沌工程是一种帮助我们获得更多的关于系统的新认知的实验方法。它和已有的功能测试、集成测试等测试已知属性的方法有本质上的区别。

一些混沌工程实验的输入样例：

·□模拟整个云服务区域或整个数据中心的故障。

·□跨多实例删除部分Kafka主题（Topic）[2]来重现生产环境中发生过的问题。

□·挑选一个时间段，针对一部分流量，对其涉及的服务之间的调用注入一些特定的延时。

□·方法级别的混乱（运行时注入）：让方法随机抛出各种异常。

□·代码插入：在目标程序中插入一些指令，使得故障注入在这些指令之前先运行。[3]

□·强迫系统节点间的时间彼此不同步。[4]

□·在驱动程序中执行模拟I/O错误的程序。

□·让一个Elasticsearch集群的CPU超负荷。

混沌工程实验的可能性是无限的，根据不同的分布式系统架构和不同的核心业务价值，实验可以千变万化。

混沌工程绝不是Netflix的专属

我们在和其他公司或组织的专业人士讨论混沌工程时，经常收到的一个反馈是：“哇哦，听起来非常有意思，但是我们的系统功能和业务与Netflix完全不同，所以这东西应该不适合我们吧。”

虽然我们提供的特定的案例都来自于我们在Netflix的经验，但是书中所描述的基本原则并不针对任何特定的组织，所介绍的实验设计指南也没有基于任何特定的架构或者工具集。在第9章里，我们会讨论混沌工程的成熟度模型，读者可以采用这个模型，评估自己将要或者已经开始实施的混沌工程实践的初衷是否合理、切入点是否正确，以及下一步应该朝哪个方向继续开展实践。

最近的一次混沌工程社区日聚集了来自不同组织的混沌工程实践者，参会者来自Google、Amazon、Microsoft、Dropbox、Yahoo!、Uber、cars.com、Gremlin Inc.、加州大学圣克鲁兹分校、SendGrid、北卡罗来纳州立大学、Sendence、VISA、New Relic、Jet.com、Pivotal、ScyllaDb、GitHub、DevJam、HERE、Cake Solutions、Sandia National Labs、Cognitect、Thoughtworks和O’Reilly出版社。在本书中，你会找到来自各行各业（金融、电子商务、航空航天等）的关于混沌工程实践的案例和工具。

混沌工程也同样适用于传统行业，如大型金融机构、制造业和医疗机构。对于金融交易所依赖的复杂系统，有大型银行正在使用混沌工程来验证其交易系统是否有足够的冗余。[5]对于生死攸关的医疗系统，在美国，混沌工程在许多方面被当作模型应用在了临床试验系统中，从而形成了美国医疗验证的黄金标准。横跨金融、医疗、保险、火箭制造、农业机械、工具制造领域，无论是数字巨头企业还是创业公司，混沌工程作为一门复杂系统改进学科，正在寻找它在各行各业中的立足点。

飞机启动失败？

在伊利诺伊大学香槟分校，Naira Hovakimyan和她的研究团队把混沌工程用在了喷气式战斗机上[6]。团队由两名B-52飞行员、一名F-16飞行员、两名飞行测试工程师和两名安全飞行员组成。在试飞过程中，飞机被注入了几种不同的故障配置。这些配置甚至包括飞机重心的突然变化和空气动力学参数的变化！能否在故障发生时重新构建机体爬升动力学参数以及其他会导致故障的配置对于团队来说是极大的挑战。在制定并亲身实践了一系列故障情景之后，团队最终能够自信地认定该系统对于低空飞行是安全的。

实施混沌工程的前提条件

在判断你的团队是否已经准备好实施混沌工程之前，你需要回答这样一个问题：你的系统是否已经具备一定的弹性来应对真实环境中的一些异常事件，像某个服务异常、网络闪断或瞬间延迟提高这样的事件。

如果你的答案是明确的“No”，那么在实施本书中讨论的各项原则之前，你需要先做一些准备工作。混沌工程非常适合用于暴露生产系统中未知的脆弱环节，但如果你很确定一个混沌工程实验会导致系统出现严重的故障，那么进行这样的实验是没有任何意义的。你需要先解决这个问题，然后再回到混沌工程，在进行混沌工程实验之后，你要么能继续发现更多未知的脆弱点，要么能对系统真实的弹性水平更有信心。

实施混沌工程的另一个前提条件是配套监控系统，你需要用它来判断系统当前的各项状态。如果无法对系统行为进行观察，你就无法从实验中得出有效的结论。由于每个系统都是独一无二的，因此当混沌工程暴露出系统脆弱环节时，如何更好地进行根源分析，我们留给读者作为练习。

混乱猴子（Chaos Monkey）

2010年年底，Netflix向全世界推出了“混乱猴子”。这家流媒体服务提供商在几年前就开始陆续将服务迁移到了云上。之前数据中心中的垂直扩展导致了很多单点故障，其中一些故障甚至导致了当时的DVD业务大规模中断。云服务不光带来了水平扩展的机会，同时把无差别的重度基础设施运维工作转移给了可靠的第三方。

数据中心本来就会时不时地发生一些小故障，然而在云服务的水平扩展架构中，提供同一个服务的节点数大幅增加，发生这类故障的概率也大幅增加。在数以千计的服务节点里，几乎可以确定的是，时不时地就会有节点出现异常或者掉线。需要有一种全新的方法，既可以保留水平扩展带来的好处，同时又有足够的弹性来应对节点时不时发生的故障。

在Netflix，并没有制度要求工程师一定要按照某种规定来构建所有东西。相反的是，高效的领导者会在工程师之间建立一种强有力的一致规约或原则，让工程师在自己的领域里找到解决问题的最好办法。在这个节点时不时就会发生故障的案例里，我们要建立的强有力

的规约和原则就是，我们开发的服务要具备在单一节点突然掉线的情况下还能持续提供端到端服务的能力。

在业务正常进行期间，混乱猴子会伪随机地关闭生产环境中正在运行的节点，而且关闭的频率比正常节点发生故障的频率还要高很多。通过高频率地触发这些不常见且潜在的灾难性事件，我们使工程师有了强大的动力在开发服务时必须考虑到如何轻松应对这类事件。工程师必须尽可能早地频繁处理这类故障。除此以外，自动化、冗余、回滚策略，以及其他弹性设计的最佳实践，都使工程师在这些故障发生时能很快让自己的服务恢复正常。

经过几年的发展，混乱猴子逐渐变得更加强大，现在它可以指定终止一组节点，并且通过与Spinnaker（我们的持续发布平台）集成来自动进行线上实验。但从根本上来说，它提供的还是与2010年以来一样的功能。

混乱猴子最大的成就在于使工程师之间形成了构建具备足够弹性的服务的规约和原则。现在，它已经是Netflix工程师文化中不可或缺的一部分了。在过去五年左右的时间里，只有一次节点掉线影响了我们的服务。当时正是混乱猴子终止了一个由于部署失误而没有冗余的服务节点造成了问题。幸运的是，这个故障发生在白天工作时间，而且是在刚刚部署了这个服务不久后，所以对用户造成的影响非常小。可想而知，如果这个服务一直在线上运行了几个月，而在某个周末的晚上，当负责该服务的工程师没有值班（on-call）的时候，混乱猴子终止了它的节点，那将会造成多大的灾难。

混乱猴子的美妙之处就在于此，它能尽可能地将服务节点失效带来的痛苦提前，同时让所有工程师在构建一个具有足够弹性应对失败的系统上，达成一个一致的目标。

---

[1]译者注：混沌工程和故障注入本质上是思维方式上的不同。故障注入首先要知道会发生什么故障，然后一个一个地注入，然而在复杂分布式系统中，想要穷举所有可能的故障，本身就是奢望。混沌工程的思维方式是主动去找故障，是探索性的，你不知道摘掉一个节点、关掉一个服务会发生什么故障，虽然按计划做好了降级预案，但是关闭节点时却引发了上游服务异常，进而引发雪崩，这不是靠故障注入或预先计划能发现的。

[2]译者注：Kafka主题（Topic）是Kafka对消息的归类方式，相当于传统消息中间件的队列（Queue）。

[3]译者注：插入的指令可以干扰程序执行，使得执行程序后提前返回不同的结果或者抛出异常等。

[4]译者注：很多程序的执行或者逻辑会依赖系统时间，比如定时任务，程序获取系统时间后根据时间执行一些逻辑，如果分布式系统各节点的时间不一

致，就有可能引发不同节点的逻辑不一致。

[5]译者注：系统需要足够的冗余来应对峰值及突发状况。

[6]Julia Cation.Flight control breakthrough could lead to safer air travel. Engineering at Illinois,2015.

第2章　管理复杂性

复杂性对工程师来说既是挑战也是机遇。你需要一支技术纯熟同时有足够应变能力的团队，来成功管理和运行一个包含许多组件和交互的分布式系统。在这样的复杂系统中充满了创新和优化的机会。

软件工程师通常会对这三个方面进行优化：性能、可用性、容错能力。

性能

这里的性能特指使延迟或资源成本最小化。

可用性

可用性，即系统正常响应和避免宕机的能力。

容错能力

容错能力是指系统从非正常状态中恢复的能力。

通常，一个有经验的团队会同时针对这三个方面进行优化。

在Netflix，工程师们还会考虑第四个方面：

新功能开发的速度

新功能开发的速度指工程师可以把新功能、创新功能提供给用户的速度。

Netflix在软件工程的决策过程中，非常明确地鼓励端到端的功能开发速度，而不仅仅是快速部署本地功能。在以上四者中找到平衡的过程，可以为架构选型提供必要的信息。

在充分考虑了这些方面之后，Netflix选择采用微服务架构。但我们要记住康威定律：

> 任何组织所做的系统设计（广义的系统）的结构，将不可避免地复制这个组织信息沟通的组织结构。
>
> ——Melvin Conway，1967

在微服务架构中，各团队彼此独立地开发和运营他们的服务。这就允许每个团队可以自行决定何时将代码布置到生产环境中。这个架构策略以沟通协调为代价来提高新功能的开发速度。将一个工程组织视为许多个这样的小型团队通常更直观。我们希望这些小团队的特点是松散耦合（即没有太多的组织架构限制，而是强调小团队之间的协作）和高度协调（即每个人都能看得更全面，从而明确他们的工作是如何有助于和其他团队一起实现更大的目标的）的。小团队之间的有效沟通是成功实施微服务架构的关键。混沌工程通过适时地验证系统弹性，来支持快速的功能开发、实验，以及增强团队对系统的信心。[1]

理解复杂系统

想象一个向消费者提供产品信息的分布式系统，其微服务架构如图2-1所示。

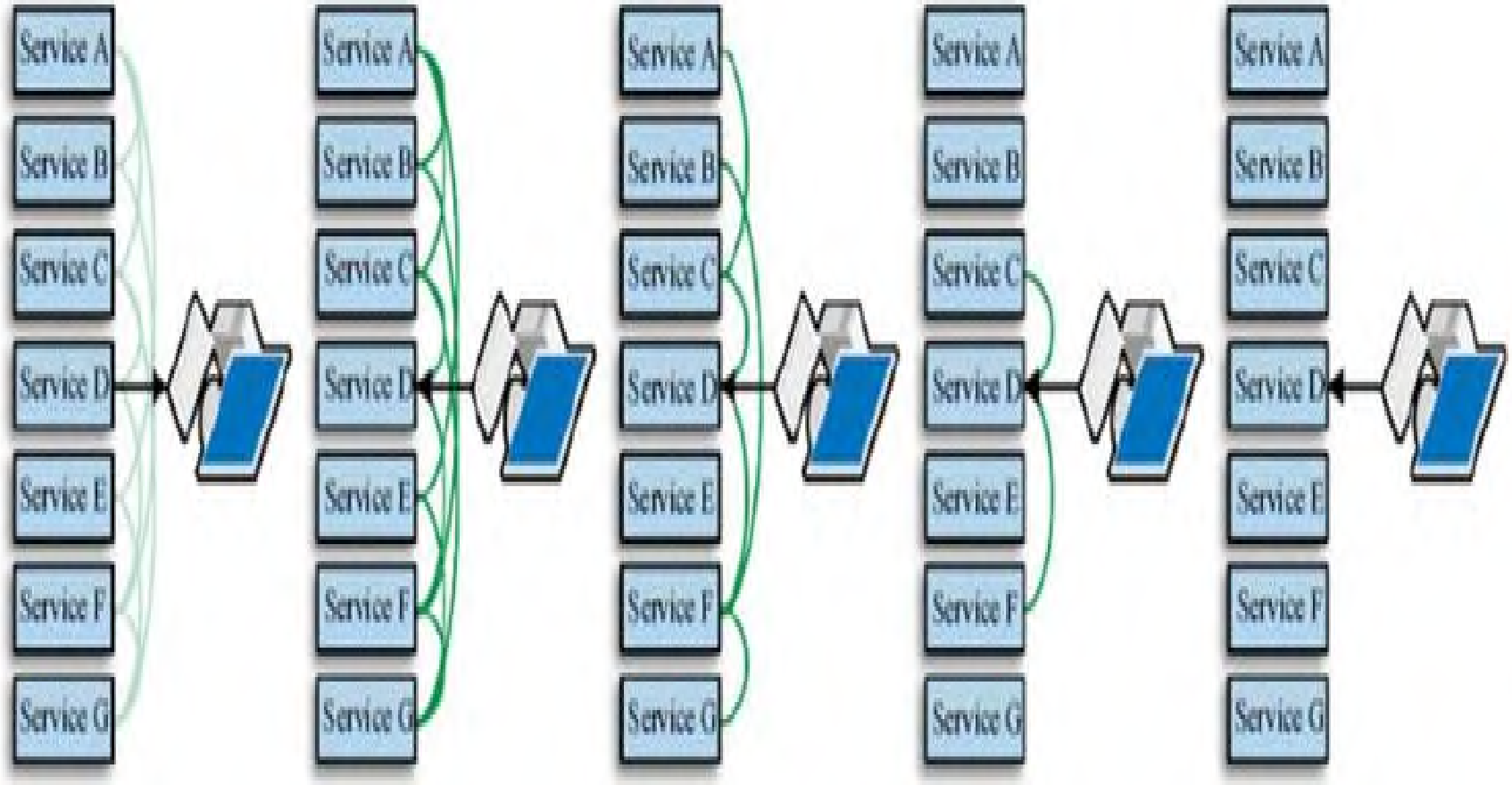
Service A
Service B
Service C
Service D
Service E
Service F
Service G

图2-1　微服务架构

这个微服务架构由7个微服务组成，从微服务A到微服务G。微服务A存储用户的个人信息。微服务B存储用户的登录账户信息，如用户上一次登录的时间和所访问的信息。微服务C存储的是关于产品的信息。微服务D作为API网关处理所有来自外部的接口访问。

让我们来看一个关于请求的例子。一个用户通过手机App访问了一些信息：

·□请求首先进入微服务D，即API服务。

□·微服务D本身并没有请求所需要的所有信息，所以它进一步请求微服务C和微服务F以获取必要的数据。

·□微服务C和微服务F也同时需要更多的信息来满足请求，于是微服务C请求微服务A，微服务F请求了微服务B和微服务G。

□·微服务A也需要访问微服务B，微服务B需要访问微服务E，同时微服务G也需要访问微服务E。对微服务D的一个请求，扩散到了整个微服务架构。而且在所有依赖的服务没有返回响应或者超时之前，API不会向手机App返回响应信息。

这样的请求模式非常常见，而且在有一定规模的系统中这类交互的数量要大得多。有趣的是，对于有一定规模的系统来说，擅长搭建紧密耦合的单体系统的传统架构师所能发挥的作用被显著削弱了。传统架构师的角色更多的是负责理解系统中各个组成部分是如何组成整个系统的，以及它们之间是如何有效交互的，然而在一个大型分布式系统中人类难以胜任这个角色。太多的组件，频繁的改动和革新，无数非计划中的组件交互，人类是不可能把这些内容全都放在大脑中的。微服务架构给我们带来了开发速度和灵活性的提升，代价却是牺牲了我们的掌控性和可理解性。这个缺失恰好为混沌工程创造了机会。

其实在任何一个复杂系统中都是这样的，即使是一个单体系统，在它变得越来越大，依赖越来越多的时候，也不会有一个架构师可以理解添加一个新的功能对整个系统意味着什么。也许有个非常有趣的例外就是有一类系统的设计原则里本来就不会考虑可理解性，例如深度学习、神经网络、遗传进化算法和其他机器智能算法。即使人类揭

开这些算法的盖子来看它们的内部构造，也很难理解一系列的权重和非无效解产生的浮点数。只有系统整体发出的响应才能被人类所理解。[2]整个系统应该具有意义，而系统的任何子部分都不需要有意义。

在一个请求/响应的过程中，意大利面条式的[3]调用图代表了典型的、需要混沌工程关注的系统固有的混乱。传统测试，如单元测试、功能测试、集成测试，在这里是不够用的。传统测试只能告诉我们正在测试的系统中的某个属性的断言是真还是假。但现在我们需要更进一步发现会影响系统行为的更多未知属性。也许一个基于真实事件的例子有助于说明这个不足。

系统复杂性的例子

微服务E包含用户个性化体验的信息，例如预测用户下一个动作，用以在手机App上展示相应的选项。一个用于展示这些选项的请求也许会先从微服务A获取用户的信息，然后从微服务E获取用于个性化的信息。

现在我们先对这些微服务是如何被设计和运行的做一些合理的假设。由于请求数量很大，因此我们采用一个固定的散列函数把用户请求均衡分散开，这样一个固定的用户只会由一个特定的节点来服务，而不是所有微服务A的节点响应整个用户群的请求。例如，在微服务A背后的数百个节点中，所有来自用户“CLR”[4]的请求只会被路由到节点A42上。如果节点A42出现问题，那么足够智能的路由逻辑会将节点A42的职责路由到集群中的其他节点上。

当下游所依赖的服务出现异常时，微服务A有合理的预案。如果微服务A无法和持久层通信，它就会从本地缓存中返回结果。

在运行时，每一个微服务都会平衡考量监控、报警和资源，以合理地兼顾性能和对内部的洞察，而不会对资源的利用率不管不顾。扩展策略会基于CPU负载和I/O性能来决定在资源稀缺时加入更多节点，以及在资源闲置时去掉多余的节点。

现在我们的环境就绪了，让我们来看看请求的模式。用户“CLR”启动了手机App应用，然后发送了一个请求来获取内容丰富的App首页。不巧的是，他的手机目前并不在服务区。用户并不知道自己不在服务区，于是他发出了多次对首页的请求，这些请求都被手机操作系统缓存在了本地队列，以等待网络连接恢复后发出。App本身也有重试机制，位于操作系统的本地队列之外，也将这些请求缓存在了App自身的队列中。[5]

突然，网络连接恢复了。手机操作系统同时把数百个请求一次性地发送出去。因为用户“CLR”发起的是对首页的请求，所以微服务E被同时请求了数百次以获取和用户个性化体验相关的信息。每一个对微服务E的请求都会先请求微服务A。于是微服务A在被其他微服务（如微服务E）请求了数百次的同时，还要响应打开和加载页面的请求。由于微服务A的架构设计，所有来自用户“CLR”的请求都被路由到了节点A42上。节点A42在这么大的流量下无法使所有请求都从持久层获取数据，所以它切换到从本地缓存中获取数据。

从缓存中响应请求大大减少了为每个请求提供服务所需的处理时间和I/O开销。事实上，节点A42的CPU负载和I/O开销会突然降到很低的水平，以至于其负载平均值低于集群扩展策略的回收阈值。于是策略考虑到资源的有效利用，对集群进行了收缩，对节点A42进行了回收，同时将流量转发到了集群的其他节点。其他节点这时就需要额外地处理本属于节点A42需要做的工作。比如节点A11接管了来自用户“CLR”的请求。

在本属于节点A42需要处理的请求转移到节点A11的过程中，微服务E中对微服务A的请求超时了。于是微服务E为了自身能应对上游的请求，启动了它的预案，返回了一些不含个性化的内容。

用户“CLR”最终收到了响应，注意到内容不像平时那样个性化，于是多刷新了几次首页。节点A11比平时需要处理的工作更多了，所以它也逐渐从缓存中返回一些稍稍过时的信息。CPU负载和I/O开销相应地降低了，系统根据扩展策略再一次提示集群可以收缩了。

其他一些用户也逐渐注意到他们的App展示给他们的内容不像往常那样个性化了。他们也开始不断刷新内容，这又触发了更多对微服务A的请求。额外的流量压力致使微服务A中更多的节点选择从缓存中

返回信息，CPU负载和I/O开销进一步相应地降低，集群进一步加速收缩。更多的用户注意到了问题，触发了用户引起的重试风暴。最终，整个集群都开始从缓存中返回信息，重试风暴压垮了其余节点，微服务A掉线了。微服务B对于微服务A掉线没有预案，于是进一步拖垮了微服务D，于是整个服务基本上都中断了。

从例子中学到了什么

上面的场景在系统理论中被称为“牛鞭效应”。输入中的一点扰动便会触发一个自我强化的循环，最终导致输出结果的剧烈波动。在上面的例子中，输出的波动拖垮了整个应用。

上述例子中最重要的一个特征是每一个单一的微服务的行为都是合理的，只有在特定场景下这些行为组合起来才会导致系统预料之外的行为。这一类交互的复杂性不是人力可以完全预料到的。即使每一个微服务都可以被全面地覆盖进行测试，我们仍然不能在测试场景或集成测试环境中看到这类预料之外的行为。

期待架构师能理解这些组件和组件的交互模式，从而能充分预测这些预料之外的系统效应，是不合理的。而混沌工程提供了可以让这些效应浮出水面的工具，从而让我们建立对复杂分布式系统的信心。有了这个信心，我们就可以为这些既庞大又充满迷雾，无法被个人全部理解的系统设计有效的架构，同时兼顾功能开发的速度。

混乱金刚（Chaos Kong）

在混乱猴子（ChaosMonkey）成功的基础之上，我们决定继续深入。混乱猴子的功能是关闭节点，而混乱金刚可以关闭整个AWS区域。

Netflix视频的每一个字节都来自于我们的CDN。在峰值的时候，我们贡献了大约北美互联网三分之一的流量。这是世界上最大的CDN，它同时也包含着许许多多“令人着迷”的工程问题，我们先把这里很多关于混沌工程的问题放在一旁。现在，先来聚焦在Netflix其他的一些服务[6]上，我们称之为Netflix服务的“控制平面”（ControlPlane）。

除了视频流媒体来自CDN，所有其他和我们服务的交互都是由AWS云服务的三个区域提供的。在数千种我们支持的设备上，从2007

年的蓝光播放器一直到最新款的智能手机，我们的云上应用可以处理全流程的服务，包括启动、注册、浏览、选择视频、播放以及播放时的心跳监测等。

2012年的圣诞节期间，AWS的一个单一区域发生了严重的中断故障，这个事故让我们必须尽快采取多区域的策略。如果你对AWS区域不太了解，那么你可以将它们理解成多个数据中心。通过多区域的故障恢复策略，我们可以把所有用户从发生故障的区域转移到另一个区域，最大限度地限制单个区域中断的范围和时长，避免发生与2012年类似的中断事故。

这项工作需要多个微服务架构团队之间进行大量的沟通协调。我们在2013年年底开发完成混乱金刚，打算用它来关闭整个AWS区域。这个强制要求让我们的工程师在开发出的服务务必要能够在AWS不同区域间平滑迁移这一目标上达成了一致。这里，我们只是模拟整个区域中断的故障，毕竟我们没有权限真正把整个AWS覆盖的区域都中断掉。

当我们认为已经为跨区域故障恢复做好了准备时，我们就开始了每个月一次的混乱金刚演习。第一年，我们经常会发现故障恢复中存在各种各样的问题，这些问题给了我们大量的改进空间。到第二年，我们已经可以非常平滑地进行演习。现在，我们已经可以非常规律地进行演习了，无论这类故障是基础设施导致的还是我们自己的软件问题导致的，都可以确保服务时刻具备应对整个区域中断故障的弹性。

---

[1]译者注：如果系统不具备足够的弹性，那么每次开发新功能上线都会是一场灾难，团队会对上线后系统是否能正常运行毫无信心，士气低落。如果对系统的弹性、稳定性具备足够的信心，一方面说明大家在开发新功能时，已经具备了足够的防御性思维；另一方面说明系统本身的设计已经具备足够的弹性来应对线上的异常，这样显然会缩短开发时间，因为不会有太多异常或需要返工的地方，新的实验也可以在原有实验的基础上被快速添加。

[2]译者注：在这样的系统中，每一个单独的步骤给出来的结果是没有什么可理解的意义的，只是一些评分值，只有模型最终产出的结果才能被人所理解。比如，判断一张图片上所展示的是不是狗，其过程中的成千上万个中间过程结果不是给人去理解的，只有最后的结果，是狗还是不是狗，才对人有意义。

[3]译者注：意大利面条式的设计是一个很经典的计算机软件设计术语，形容软件开发中的一种现象，系统中的逻辑纠缠在一起，没有清晰的模块和层次，各种功能交织在一起，其中的各种关系就像一盘意大利面条。

[4]译者注：用户代号，比如用户Bill、Adam。

[5]译者注：App自身会通过代码实现缓存队列的功能，这些缓存队列在App层，不属于操作系统层。

[6]译者注：这类服务类似于路由器中执行路由功能的部分。

第二部分　混沌工程原则

> 优化一个复杂系统的性能通常需要在混乱的边缘进行，即在系统行为即将开始变得混乱、无迹可寻之前。
>
> ——SydneyDekker，*Drift Into Failure*

“混乱”一词让我们想起随机性和无序性。然而，这并不意味着混沌工程的实施也是随机和随意的，也不意味着混沌工程师的工作就是引发混乱。相反的是，我们把混沌工程视为一门原则性很强的学科，特别是一门实验性的学科。

在上面的引用中，Dekker观测了分布式系统的整体行为，他也主张从整体上了解复杂系统是如何失效的。我们不应该仅仅着眼于发生故障的组件，而是应该尝试去理解，像组件交互中一些偶发的意外行为，最终是如何导致系统整体滑向一个不安全、不稳定的状态的。

你可以将混沌工程视为一种解决“我们的系统离混乱边缘有多少距离”的经验方法。从另一个角度去思考，“如果我们把混乱注入系统，它会怎么样？”

在这一部分，我们会介绍混沌工程实验的基本设计方法，之后会讨论一些更高级的原则。这些原则建立在真正实施混沌工程的大规模系统之上。在实施混沌工程的过程中，并不是所有高级原则都必须用到。但我们发现，运用的原则越多，你对系统弹性的信心就越充足。

实验

在大学里，电气工程专业的学生必须学习一门“信号和系统”的课程，在这门课程中他们学习如何使用数学模型来推理电气系统的行为。其中一门需要掌握的技术被称为拉普拉斯变换。你可以用拉普拉斯变换，将整个电路的行为用一个数学函数来表达，我们称之为传递函数。传递函数描述的是系统在受到脉冲冲击时如何响应，输入信号

包含所有可能的输入频率的总和。一旦你有了一个电路的传递函数，就可以预测它在收到所有可能的输入信号时会如何响应。

软件系统里并没有类似的传递函数。像很多复杂系统一样，我们无法为软件系统表现出的各种行为建立一个预测模型。如果我们有这样一个模型，可以推导出一次网络延迟骤升会给系统带来什么影响，那就太完美了。但不幸的是，迄今为止我们并没有发现这样一个模型。

因为我们缺乏这样一个理论的预测模型，因此就不得不通过经验方法来了解在不同的情况下我们的系统会如何表现。我们通过在系统上运行各种各样的实验来了解系统的表现。我们尝试给系统制造各种麻烦，看它会发生什么状况。

但是，我们肯定不会给系统不同的随机输入。我们在系统分析之后，期望能够最大化每个实验可以获得的信息。正如科学家通过实验来研究自然现象一样，我们也通过实验来揭示系统的行为。

故障注入测试（FIT）

有关分布式系统的经验告诉我们，各种系统问题基本都是由预料外的事件或不良的延迟导致的。2014年年初，Netflix开发了一个名为FIT的工具，意思是故障注入测试（Failure Injection Testing）。这个工具允许工程师在访问服务的一类请求的请求头中注入一些失败场景。这些被注入失败场景的请求在系统中流转的过程中，微服务中被注入的故障锚点会根据不同的失败场景触发相应的逻辑。

例如，我们要测试系统在某一个保存用户数据的微服务中断时的弹性。我们预计系统中的某些服务不会如预期运行，但一些基础功能——例如重放（Playback）——对已登录的用户应该可以正常使用。通过使用FIT，我们指定5%的用户进入这个用户数据失败的场景。5%的请求会在请求头中包含这个失败场景。当这些请求在系统中传播时，只要是发到客户数据微服务的请求，就会自动收到故障的响应。

高级原则

在开发混沌工程实验时，请牢记以下原则，它们将有助于实验的设计。在接下来的章节中，我们将会深入研究每一个原则：

□·建立稳定状态的假设。

□·用多样的现实世界事件做验证。

□·在生产环境中进行实验。

□·自动化实验以持续运行。

□·最小化爆炸半径。

预测和预防故障

在2017年的美洲SRECon大会上，Preetha Appan介绍了她和她的团队在 indeed.com 上开发的一个引入网络故障的工具。在演讲中，她介绍了对预防故障能力的切实需要，而不是仅仅在故障发生时做出响应。作为一个守护进程，他们的工具Sloth运行在其基础设施的每一个节点上，包括他们的数据库和索引服务器。

参见：https://www.usenix.org/conference/srecon17 americas/program/presentation/appan。

第3章　建立稳定状态的假设

任何复杂系统都会有许多可变动的部件、许多信号，以及许多形式的输出。我们需要用一个通用的方式来区分系统行为是在预料之内的，还是在预料之外的。我们可以将系统正常运行时的状态定义为系统的“稳定状态”。

如果你在开发或运行一个软件服务，那么，你如何清楚地了解它是否在正常工作？你如何定义这个服务的稳定状态[1]？你应该从哪里着眼来回答上面的问题？

稳定状态

在系统思维领域中使用“稳定状态”这个术语，来指代一个系统倾向于维持在一定范围或模式内的属性，就像人体系统将体温维持在一定范围内一样。我们期望通过一个模型，基于所期望的业务指标来描述系统的稳定状态。这是我们在识别稳定状态方面的一个目标。要牢记，稳定状态一定要和客户接受程度一致。在定义稳定状态时，要把客户和服务之间的服务水平协议（SLA）纳入考量的范围。

如果你的服务是一个新服务，那么想知道它是否正常工作的唯一途径可能就是自己去运行一下。例如，你的服务可以通过一个网站来访问，那么你可能需要打开网页并尝试触发一个任务或事务来检查这个服务。

这种快速检查系统健康状况的方法显然并不理想：它是劳动密集型的，也就是说我们基本不会或不常会做这件事。我们可以自动化运行这样的测试，但这还不够。如果这些测试并不能揭示我们需要发现的问题怎么办？

更好的方法是先搜集和系统健康有关的数据。如果你在阅读本书，那么，我们相信你已经在使用某种指标收集系统来监控自己的系统了。有大量的开源工具和商业工具可以帮助我们采集系统方方面面的数据：CPU负载、内存使用情况、网络I/O，以及各类时序信息，例如需要多长时间来响应一个Web请求，或者各类数据库的查询耗时。

系统指标有助于帮助我们诊断性能问题，有时也能帮助我们发现功能缺陷。业务指标与系统指标形成对比，业务指标通常回答这样的问题：

□·我们正在流失用户吗？

□·用户目前可以操作网站的关键功能吗？例如，在电子商务网站上为订单付款，或者将商品添加到购物车等。

□·目前存在较高的延迟致使用户不能正常使用我们的服务吗？

某些组织有非常明确的和收入直接相关的实时指标。例如，Amazon和eBay会跟踪销量，Google和Facebook会跟踪广告曝光次数。

由于Netflix使用的是按月订阅模式，因此我们没有这类指标。我们的确会测量注册率，这是一个重要的指标，但是单独看注册率并不能反映整体系统的健康状况。

我们真正想要的是一个可以反映当前活跃用户满意状况的指标，因为满意的用户才更有可能会连续订阅。可以这么说，如果当前和系统正在做交互的用户是满意的，那么我们就有较大的信心——系统目前是健康的。

遗憾的是，我们目前还没找到一个可以直接地、实时地反映用户满意度的指标。我们会监控客服电话的呼叫量，这或许是一个可以间接反映客户满意度的指标，但是从运营角度出发，我们需要更快、更细粒度的反馈。Netflix有一个还不错的、可以间接反映用户满意度的指标——播放按钮的点击率。我们将这个指标叫作视频每秒开始播放数，简称为SPS（Starts Per Second）。

SPS很容易测量，而且因为用户付费订阅服务的直接目的就是看视频，所以SPS应该和用户满意度密切相关。比如，这个指标在美国东海岸的下午6点会明显高于早上6点。我们可以据此来定义系统的稳定状态。

相比于某个服务的CPU负载来说，Netflix的网站可靠性工程师（SRE）会更关注SPS的下降：当SPS下降时系统会立刻向他们发送警报。CPU负载的尖刺有时重要，有时不重要，而像SPS这样的业务指

标才是对系统是否处于稳定状态的精确描述。这样的指标（而不是内部的一些像CPU负载这样的指标）才是我们要关注并验证的。

很多现有的数据采集框架已经默认采集大量的系统级别指标，所以通常来说，让你的系统有能力抓取业务级别的指标比抓取系统级别的指标更难。然而花精力来采集业务级别的指标是值得的，因为它们才能真实地反映系统的健康状况。

这些指标获取的延迟越低越好：那些在月底算出来的业务指标和系统今天的健康状况毫无关系。

在选择指标时，你需要平衡以下几点：

□·指标和底层架构的关系。

□·收集相关数据需要的工作量。

·□指标和系统接下来的行为之间的时间延迟。

如果你还不能直接获得和业务直接相关的指标，则也可以先暂时利用一些系统指标，比如系统吞吐率、错误率、99%以上的延迟等。你选择的指标和自己的业务关系越强，得到的可以采取可执行策略的信号就越强。你可以把这些指标想象成系统的生命特征指标，如脉搏、血压、体温等。同样重要的是，在客户端验证一个服务产生的警报可以提高整体效率，并可以作为对服务器端指标的补充，以构成某一时刻用户体验的完整画面。

如何描述稳定状态

与人体的生命体征一样，你需要清楚“健康”的数值范围。例如，我们知道37摄氏度左右是人体的健康温度。

牢记我们的目标：期望通过一个模型，基于所期望的业务指标来描述系统的稳定状态。

很多业务指标并不像我们的体温那样稳定，它们也许会经常剧烈波动。我们再举一个医学中的例子，心电图展示了心脏附近人体表面的电压差，这个信号是用来观测心脏行为的。但心电图采集的信号会

随着心脏跳动而变化，因此医生不能将这个信号与单个阈值进行比较来判断患者是否健康。医生需要比较的是信号波动的模式和患者健康时的模式是否一致。

在Netflix，SPS也不是一个和人体体温一样的稳定指标。它也随着时间波动。图3-1描绘的就是SPS和时间的波动关系，但可以看出，它有一个稳定的模式。这是因为人们更倾向于在晚餐时间看电视节目。比如，因为SPS随时间的变化可以预期，所以我们就可以用一周前的SPS波动图作为稳定状态的模型。Netflix的可靠性工程师们总是将过去一周的波动图放在当前的波动图之上，以发现两者的差异性。就像图3-1中当前的图线是红色的，上一周的图线是黑色的。

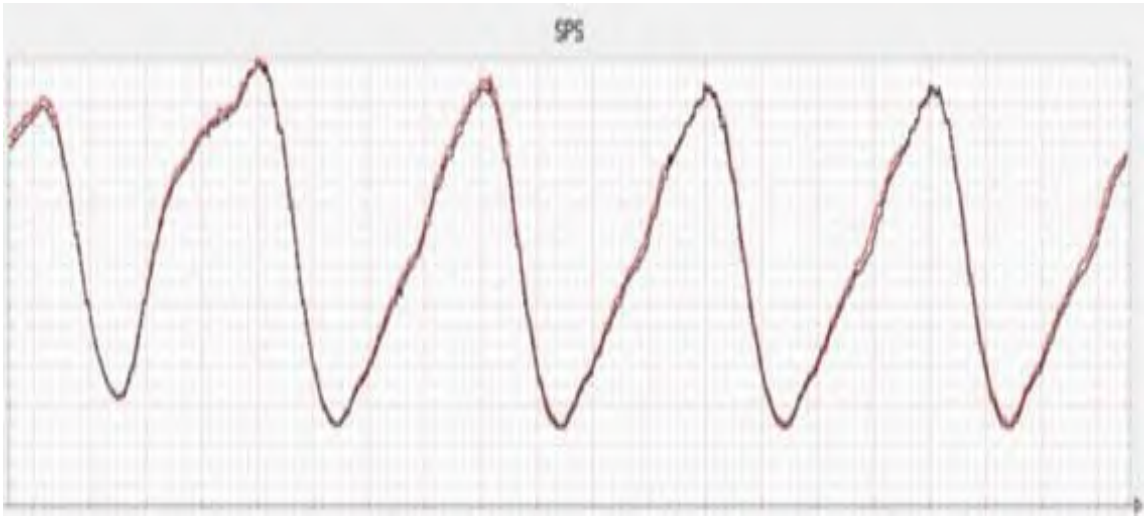

图3-1　SPS随时间规律地变化

扫码看彩图

你所处的行业决定了你的指标是否以一种可以预测的方式随时间波动。例如，如果你负责一个新闻网站，流量的尖刺可能来源于一个大众关注度高的新闻事件。某些事件的尖刺可能可以预测，比如选举、重大赛事，但是其他类型的事件不太可能被预测到。在这一类场景中，准确描述系统的稳定状态将会非常复杂。无论哪种情况，描述稳定状态都是建立一个有意义的假设所必需的前提条件。

建立假设

当你进行混沌工程实验的时候，应该首先在心里对实验结果有一个假设。将你的系统直接置于各种事件中看看系统会怎么样的想法可能比较诱人。然而，没有一个预先的假设，你就不清楚应该从数据里找什么，最终也难以得出有效的结论。

定义好指标并理解其稳定状态的行为之后，你就可以使用它们来建立实验的假设了。思考一下，当你向系统中注入不同类型的事件时，稳定状态的行为会发生什么变化。例如，你向中间层的服务增加请求数量，稳定状态会被破坏还是保持不变？如果被破坏了，你期待系统如何表现？

在Netflix，我们使用混沌工程来提高系统弹性，因此我们实验的假设一般是这样的形式：我们向系统注入的事件不会导致系统稳定状态发生明显的变化。

例如，我们在弹性实验里故意让一个非关键服务中断，以验证系统是否可以优雅降级。我们可能会中断基于用户浏览历史来展示个性化影片列表的服务，这时系统应该返回一个默认的影片列表。

每当我们进行非关键服务中断的实验时，我们的假设都是注入的故障不会对SPS产生影响。换句话说，我们的假设是，实验措施不会使系统行为偏离稳定状态。

我们还会定期执行演习，例如将一个AWS区域的流量全部转移到另外两个区域。目的是验证我们在进行这类故障恢复的时候，SPS不会偏离稳定状态。这可以让我们对出现一个区域中断性故障时执行故障恢复充满信心。

最后，让我们思考一下，如何衡量稳定状态行为的变化。即便你已经建立了稳定状态行为的模型，你也需要定义清楚，当偏离稳定状态行为发生时如何测量这个偏差。如果你曾调节过报警系统的阈值设置，就应该清楚，定义偏离稳定状态的偏差是否在合理的范围内是具有挑战性的。如果你将“正常”的偏差范围定义清楚了，就可以获得一套验证假设的完善的测试集。

金丝雀分析

在Netflix，我们使用金丝雀发布[2]：我们首先把新代码发布到一个只接受一小部分生产流量的小型集群中，然后进行验证以确保新发布版本的健康性，最后再进行全面发布。

我们使用一个名为自动金丝雀分析（Automated Canary Analysis，ACA）的内部工具，通过稳定状态的指标来验证金丝雀集群是否健康。ACA将一个与金丝雀集群大小相同并运行旧代码的基准集群，和一系列系统指标进行比较，金丝雀集群的得分必须足够高才能通过金丝雀发布阶段。服务拥有者还可以向ACA中添加应用的自定义指标。

通过ACA，工程师可以清晰地观察描述稳定状态的重要参数，同时可以基于稳定状态建立的假设，对不同集群间的表现进行比较验证。其他的一些混沌工程工具也会使用ACA提供的服务来测试稳定状态下有关变化的各类假设。

---

[1]译者注：以Netflix的播放服务为例，当某个时间点的每秒点击播放次数大于一个阈值时，该服务就处于稳定状态；再比如一个打车服务，当司机接单率大于80%的时候，该服务就处于稳定状态。所以这里所说的如何定义稳定状态，就是如何找到这个指标。

[2]译者注：金丝雀发布是一种发布软件版本的技术，通过让新版本只对少量用户开放，以验证新功能是否存在不可接受的缺陷，从而降低向所有用户推出新版本的风险。金丝雀的命名起源于17世纪英国煤矿工人使用金丝雀检测煤矿中有毒气体的危险程度。金丝雀对瓦斯十分敏感。空气中即使有极其微量的瓦斯，金丝雀也会停止鸣唱。当瓦斯含量超过一定限度时，在人类还毫无察觉时，金丝雀却已毒发身亡。

第4章　用多样的现实世界事件做验证

每个系统，从简单到复杂，只要运行时间足够长，都会受到不可预测的事件和条件的影响。例如，负载的增加、硬件故障、软件缺陷，以及非法数据（有时称为脏数据）的引入。我们无法穷举所有可能的事件或条件，但常见的有以下几类：

□·硬件故障。

·□功能缺陷。

·□状态转换异常（例如发送方和接收方的状态不一致）。

·□网络延迟和分区。

□·上行或下行输入的大幅波动以及重试风暴。

□·资源耗尽。

□·服务之间不正常的或者预料之外的组合调用。

□·拜占庭故障（例如性能差或有异常的节点发出有错误的响应、异常的行为、对调用者随机地返回不同的响应等）。

□·资源竞争条件。

□·下游依赖故障。

也许最复杂的情况是上述事件的各类组合导致系统发生异常行为。

要彻底阻止对可用性的各种威胁是不可能的，但是我们可以尽可能地减轻这些威胁。在决定引入哪些事件的时候，我们应当估算这些事件发生的频率和影响范围，权衡引入它们的成本和复杂度。在Netflix，我们选择关闭节点的一方面原因是，节点中断在现实中发生的频率很高，而引入关闭节点事件的成本和难度很低。对于区域故障来说，即使引入一些事件的成本高昂且引入流程复杂，我们还是必须

要做，因为区域性故障对用户的影响是巨大的，除非我们有足够的弹性应对它。

文化因素也是一种成本。例如在传统数据中心的文化中，基础设施和系统的健壮性、稳定性高于一切，所以传统数据中心通常会对变更进行严格的流程控制。这种流程控制，和频繁关闭节点的操作是一对天然的矛盾体。随机关闭节点的实验对传统数据中心的文化是一种挑战。随着服务从数据中心迁移到云上，管理基础设施的职责被转移给了云服务提供商，硬件的各类故障都由云服务平台管理，工程部门对硬件故障就越来越习以为常。这种认知实际上在鼓励一种将故障当作可预料的态度，这种态度可以进一步推动混沌工程的引入和实施。虽然硬件故障并不是导致线上事故最常见的原因，但是注入硬件故障是在组织中引入混沌工程并获益的一个较简单的途径。

和硬件故障一样，一些现实世界的事件也可以被直接注入，例如每台机器的负载增加、通信延迟、网络分区、证书失效、时间偏差、数据膨胀等。除此之外的一些事件的注入可能会具有技术或文化上的障碍，所以我们需要寻找其他方法来看一看它们会如何影响生产环境。[1]例如，发布有缺陷的代码。金丝雀发布可以阻止许多显而易见的简单软件缺陷被大规模发布到生产环境中，但其并不能阻止全部的缺陷被发布出去。故意发布有缺陷的代码风险太大，可能会给用户带来严重的影响（参见第7章）。要模拟这类发布所带来的缺陷问题，一种办法是对相应的服务调用注入异常。

Blockade

戴尔云管理团队开发了一个开源的、基于Docker的、用来测试分布式应用的网络故障和分区的混沌工程工具，名叫Blockade。它通过在Docker的宿主机网络管理中创建各种异常场景，影响在Docker容器中运行的应用。这个工具的一些功能包括在容器间创建任意分区、容器数据丢包、容器网络延迟注入，以及在故障注入时便捷的系统监控能力。

我们了解了可以通过对相应服务调用注入异常来模拟一次失败的部署，这是因为错误代码带来的影响已经被隔离、限制在运行它们的寥寥无几的数台服务器中。通常来说，故障隔离既可以是物理隔离，也可以是逻辑隔离，隔离是容错的必要但不充分条件。要获得一个可以接受的结果，还需要某种形式的冗余或者优雅降级。如果一个子部

件发生故障就能引发整个系统不可用，那么说明这个故障没有被隔离。一个故障的影响范围和隔离范围被称为这个故障的故障域。

提供产品的组织设定产品的可用性预期，并负责制定SLA，比如哪些东西一定不能失败或者失败时是否具备应对的预案。发现和验证故障域以确保满足产品的可用性预期是工程师团队的责任。

实施混沌工程时采用故障域的概念具有一定的乘数效应[2]。回到刚才的例子，如果对服务调用失败的模拟是成功的，那么这不仅可以验证该服务对部署缺陷代码的弹性，同时也可以验证它在高负载、错误配置、其他异常终止等场景引发失败时的弹性。此外，在故障域中你可以向系统注入各类故障来观察故障的症状。例如，你在真实环境中看到了一些症状，便可以逆向找出症状的根源故障及其发生的概率。在故障域的级别进行实验还有个好处，那就是可以让你对导致故障的无法预料的因素做好准备。

我们应该对系统注入引发故障的根因事件（root-cause event）。每一个资源都会形成一个故障域，这个故障域包括所有对它有强依赖的部件（当该资源不可用时，所有依赖也都不再可用）。向系统注入故障的根因事件会暴露出这些因为资源共享形成的故障域。团队也经常会为这类资源共享的情况而感到惊讶。

我们不需要穷举所有可能对系统造成改变的事件，只需要注入那些频繁发生且影响较大的事件，同时要足够了解会被影响的故障域。工程师在设计系统架构的时候也许已经考虑了故障域。例如在微服务架构中，服务组[3]是最重要的故障域之一。有时团队认为他们的服务不是关键服务，但最后却在出现故障时因为没有做合理的隔离而导致整个系统宕机。所以，在系统中用实验验证这些预先定好的边界非常关键。

再强调一次，注入的事件一定是你认为系统能处理的。同时，注入的事件应该是所有可能的真实世界的事件，而不仅仅是故障或延迟。我们上面举的例子更多关注软件部分，但在真实世界里人的因素对系统弹性和可用性也具有至关重要的作用。例如，对故障处理中人工控制的流程和工具进行实验或演习也会提高可用性。

Netflix的混沌工程的规范化

在Netflix，混乱猴子和混乱金刚是由一个集中的团队进行开发、发布、维护、制定规则和执行的。而FIT是一个自助服务工具。这也是第一次我们的工具需要微服务工程师们的时间和接受才能运行。当然，他们不得不对工具所发现的对弹性的威胁做出回应。即便许多团队在开发阶段确实尝试到了FIT的好处，但还是难以广泛和高频率地使用。我们有FIT这么强大的工具来提高弹性，但是却遇到了如何普及的问题。

Netflix的混沌工程团队认为，现在我们分别有了小规模（如关闭节点）和大规模（关闭整个区域）的实践，但还缺少中间环节的最佳实践：持续提高系统对微服务各类故障的弹性。FIT为这方面的探索提供了一个基础，但是执行这样一个实验需要开发者自行配置，这使得它并没能像混乱猴子和混乱金刚那样，在工程师团队中被广泛使用。

我们回过头来仔细思考了如何将混沌工程落到实践中。我们走访了一些工程师，询问混沌工程对他们来说意味着什么。多数人的回复都是，混沌工程就是在生产环境中搞破坏。这听起来很有意思，很多人都会在生产环境中搞破坏，但这些破坏却毫无价值。我们要想办法让混沌工程规范化。

2015年年中，我们发布了混沌工程原则（*Principles of Chaos Engineering*，https://principlesofchaos.org/），将混沌工程定义为计算机科学中的一门新学科。

通过制定这些原则，将混沌工程规范化，我们在Netflix进一步推动了混沌工程的实践。我们为混沌工程的架构绘制了蓝图：我们清楚目标是什么，而且清楚如何评估我们做得够不够好。混沌工程原则为我们将混沌工程提升到新的高度奠定了基础。

---

[1]译者注：主动引发线上生产环境故障的做法，在很多保守的组织和行业中实施时会遭遇较大的阻力。要注意，这是一种完全不同的思维方式，会对一些根深蒂固的思维和文化产生冲击。在这种环境中，译者建议首先在一个尽可能接近生产环境的环境（如之前提到的STG环境）中，进行非常小规模的尝试，并将实际结果展示给组织中的相关部门。通过很多小步骤的尝试，逐渐使团队获得信心。

[2]译者注：乘数效应（Multiplier Effect）是一种宏观的经济效应，也是一种宏观经济控制手段，是指经济活动中某一变量的增减所引起的经济总量变化的连锁反应程度。这里泛指在实施混沌工程时采用故障域的概念，可以将混沌工程带来的好处扩大好几倍。

[3]译者注：服务组是指一组特定领域内的服务，或者是为了实现一个更大的单一目标的服务群。

第5章　在生产环境中进行实验

在我们这个行业里，在生产环境中进行软件验证的想法通常都会被嘲笑。“我们要在生产环境中验证”这句话更像是黑色幽默，它可以被翻译成“我们在发布之前不打算完整地验证这些代码”。

经典测试的一般信条是，寻找软件缺陷要离生产环境越远越好。例如，在单元测试中发现缺陷比在集成测试中发现更好。这里的逻辑是，离生产环境的整个部署越远，就越容易找到缺陷的根本原因并将其彻底修复。如果你曾经分别在单元测试、集成测试和生产环境中调试过问题，上述逻辑的好处就不言而喻了。[1]

但是在混沌工程领域，整个策略却要反过来。在离生产环境越近的地方进行实验越好。理想的实践就是直接在生产环境中进行实验。

在传统的软件测试中，我们是在验证代码逻辑的正确性，是在对函数和方法的行为有良好理解的情况下，写测试来验证它们对不对。换句话说，是在验证代码写得对不对。

而当进行混沌工程的实验时，我们所感兴趣的是整个系统作为一个整体的行为。代码只是整个系统中比较重要的一部分，而除了代码，整个系统还包含很多其他方面，特别是状态、输入，以及第三方系统导致的难以预见的系统行为。

下面来深入了解一下为什么在生产环境中进行实验对混沌工程来说是至关重要的。我们要在生产环境中建立对系统的信心，所以当然需要在生产环境中进行实验。否则，我们就仅仅是在其他并不太关心的环境中建立对系统的信心，这会大大削弱这些实践的价值。

状态和服务

我们之前简要讨论过系统的“状态”。在这一节中，我们详细讨论一下有状态服务。如果我们不需要在系统中维护任何状态，那么软件

工程将会简单很多。然而状态却在我们建造的这类系统中无处不在。

在微服务架构中提到状态的时候，我们通常说的是有状态服务，例如数据库服务。仅仅用数据库保存一些像测试设置开关这样的系统，和用数据库保存所有生产数据的系统在行为上是不同的。其他一些有状态服务包括缓存服务、对象存储服务，以及可持久化的消息服务。

配置数据是另一种影响系统行为的状态。无论你是使用静态配置文件、动态配置服务（像etcd），还是两者的组合（像我们在Netflix一样），这些配置信息本身也都是一种状态，而且它们可能会严重影响系统行为。

即使在无状态的服务中，状态仍然在内存中以数据结构的形式存在于请求之间，并因此会影响后续的请求。

还有无数的角落里隐藏着许多状态。在云服务中，一个自动伸缩组（Auto Scaling Group，ASG）[2]中的虚拟机或者容器的个数也是一种系统的状态，这种状态会随时间、外部需求或集群变化而不断改变。网络硬件，如交换机和路由器，也有状态。

总有一些意想不到的状态会伤害到你。如果你是我们的目标读者，你应该已经有一些“伤疤”了。这些对系统弹性的威胁正是混沌工程所感兴趣的，要想解决掉它们，你需要使生产环境中存在的各式各样的状态问题在混沌工程实验中暴露出来。

生产环境中的输入

对于软件工程师来说，最难的一课莫过于，用户永远不会如你预期的那样与系统进行交互。这个问题在设计系统的UI（User Interface，用户界面）时尤为典型，在我们设计混沌工程实验的时候也需要牢记。

设想一下，你的系统提供了一个从用户接收不同类型请求的服务。你可以为用户输入设计一个组合数据模型，但因为用户永远不会

如你预期般地行动，因此生产环境的系统总是会收到测试所覆盖不到的组合数据。

真正对系统建立信心的唯一方法就是，在生产环境中对真实的输入数据进行验证并实验。

第三方系统

> 分布式系统就是，其中有一台你根本不知道的机器发生故障了，有可能会让你自己机器上的服务也发生故障。
>
> —— Leslie Lamport

我们虽然可以预见所有在控制范围内的系统的状态，但也总是会依赖于外部系统，这些外部系统的行为我们不可能全都知道。2012年的平安夜，在Netflix全体成员的记忆上烙上了深深的印记。AWS一个区域的 ELB（Elastic Load Balancing）服务发生故障，导致Netflix的全部服务严重中断。

如果你的系统部署在像AWS或Azure这样的云服务中，那么大量的你所依赖的、又不完全了解的外部服务的存在就是显而易见的。但即使你的系统全都运行在自己的数据中心上，你也会发现在生产环境中系统还是会依赖其他的外部服务，例如DNS、SMTP、NTP等。就算这些服务都是自己部署的，它们也经常会需要和不受你控制的外部服务进行交互。

如果服务提供了一个Web UI，那么用户所使用的浏览器就变成了系统的一部分，但它不受你控制。即使你对客户端拥有全面的控制，例如IoT设备，也仍然逃不开用户所连接的网络环境和设备的影响。

第三方系统在自己的生产环境中的行为总是和在它与其他环境集成的大环境中的行为有所不同。这进一步强调了在生产环境中进行实验的必要性，生产环境才是你的系统和第三方系统进行真实交互的唯一场所。

面向云服务的混沌工程工具

只要系统部署在云上，那么混乱就是不可避免的。所幸我们的社区已经注意到并且开发了一些优秀的、基于云的混沌工程工具，并且

可以和不同的云提供商整合使用。除了混乱猴子，值得一提的工具有Chaos Lambda，它可以让我们在生产时间随机关闭AWS的自动伸缩组（ASG）实例。还有Microsoft Azure的Fault Analysis Service，它是专为构建在Microsoft Azure Service Fabric上的测试服务而设计的工具。

生产环境变更

在Netflix，我们的系统一直在不断更新。工程师和自动脚本每天都在通过不同的方式更新着系统，例如，发布新代码，更改动态配置，添加持久化的数据，等等。

如果将系统的概念扩展，使其包括这些生产环境中的变更，那么很明显在测试环境中想要模拟这些系统行为是非常困难的。

外部有效性

当心理学家或教育研究人员等社会科学家进行实验时，他们的主要关注点之一就是“外部有效性”：这个实验的结果是否能概括我们真正感兴趣的现象？或者我们测量结果的产品运行环境是否是专门为测试而准备的？

如果你不直接在生产环境中运行混沌工程实验，那么我们在本章所讨论的问题（状态、输入、第三方系统、生产环境变更）就都是混沌工程实验的外部有效性的潜在威胁。[3]

不愿意实践混沌工程的说辞

我们认识到，在有些环境中，直接在生产环境中进行实验可能非常困难甚至不可能。我们并不期待工程师将干扰注入行驶中的自动驾驶汽车的传感器里。但是，多数用户操作的应该都不是这类安全攸关的系统。

我很确定它会宕机

如果你不愿意在生产环境中进行混沌工程实验的原因是，对系统在注入事件时的反应缺乏信心，那么这可能是你的系统还不够成熟以应对混沌工程实验的信号。你应该在对系统的弹性具备一定信心的时候再进行混沌工程实验。混沌工程的一个主要目的是识别系统中的薄弱环节。如果你已经看到了明显的薄弱环节，那么你应该首先专注于提高系统在这一点上的弹性。当你确信系统有足够的弹性时，就可以开始进行混沌工程实验了。

如果真的宕机了，我们就会有大麻烦

即使你对系统具有弹性有很大的信心，你也许还会犹豫不决——要不要进行混沌工程实验。因为你担心在实验揭示出系统薄弱环节时会造成过多的破坏。

这是一个非常合理的顾虑，也是我们正在致力解决的问题。我们采用的方法是通过下面两个策略来将潜在的影响范围最小化：

□·允许快速终止实验。

·□将实验造成的爆炸半径最小化。

在进行任何混沌工程实验之前，都应该先有一个用来立即终止实验的“大红色按钮”（我们真的有一个大红色按钮，虽然是虚拟的）。更好的方法是将这个功能自动化，当监测到对稳定状态有潜在危害时立即自动终止实验。

第二个策略要求，在设计实验时，要考虑到如何既能从实验中获得有意义的结论，又能兼顾最小化实验可能造成的潜在危害。这一点将在第7章中讨论。

离生产环境越近越好

即便你不能在生产环境中进行实验，也要尽可能地在离生产环境最近的环境中进行。越接近生产环境，对实验外部有效性的威胁就越少，对实验结果的信心就越足。

不能在生产环境中做实验？

2015年，在纽约的Velocity Conference上，来自Fidelity Investment的Kyle Parrish和David Halsey做了题为《太大而无法测试：如何破坏一个在线交易平台而不造成金融灾难》的演讲。他们在一个拥有大型机的金融交易系统上进行了混沌工程实验。用他们的话来说：“我们意识到，用我们的灾备环境，配合生产环境的前端，就可以用现有的一些部件再构造出一套生产环境。我们越深入探究，这个想法就越可行。我们发现我们的大型机灾备系统非常理想，因为它与生产环境保持实时同步，包含所有生产环境的代码、数据、处理能力和存储能力，支持团队也完全了解它是如何运行的。我们也清楚地了解到，我们可以在这个系统上执行2倍或3倍于实际生产峰值时的流量。我们可以再搭建一套生产环境！”这是一个有创造力的、绕过制度规范来模拟生产环境的实例。

记住：为了保障系统在未来不会遭受大规模中断，冒一点可控的风险是值得的。

---

[1]译者注：在越接近生产环境的环境中调试问题，涉及的服务、系统就越多、越复杂，也就越难找到问题的原因。

[2]译者注：自动伸缩组一般是指具有相同应用场景的一组云服务器实例，这些服务器实例对外提供同一组服务。一个伸缩组内的云服务器实例有数量限制，并需要做其他伸缩配置，运行时伸缩组会自动进行实例伸缩，调整实例数量。

[3]译者注：如果不在生产环境中进行实验，而在如测试环境中进行实验得到实验结果，那么并不能认为在生产环境中也会得出同样的结果，所以所做的实验就是无效的。

第6章　自动化实验以持续运行

自动化是最长的杠杆[1]。在混沌工程的实践中，我们自动执行实验，自动分析实验结果，并希望可以自动创建新的实验。

自动执行实验

手动执行一次性的实验是非常好的第一步。当我们想出寻找故障空间的新方法时，经常从手动的方法开始，小心谨慎地处理每一件事以期建立对实验和对系统的信心。所有当事人都聚集在一起，并向CORE（Critical Operations Response Engineering，Netflix的SRE团队的名称）发出一个警示信息，说明一个新的实验即将开始。

这种谨小慎微的态度有利于：a）正确运行实验，b）确保实验有最小的爆炸半径。在成功执行实验后，下一步就是将这个实验自动化，让其持续运行。

如果一个实验不是自动化的，那么就可以将这个实验废弃。

当今系统的复杂性意味着，我们无法先验地知道，生产环境的哪些变动会改变混沌工程实验的结果。基于这个原因，我们必须假设所有变动都会改变实验结果。在共享的状态、缓存、动态配置管理、持续交付、自动伸缩和对时间敏感的代码等因素的作用下，生产环境实际上处在一个无时不变的状态。这导致的结果就是，对实验结果的信心是随着时间而衰减的。

在理想情况下，实验应该随着每次的变化而执行，这有点儿像混沌金丝雀。当发现新的风险时，操作人员如果相当确定风险的根源是即将发布的新代码，那么他就可以选择是否阻止发布新版本并优先修复缺陷。这种方法可以帮助我们更深入地了解生产环境中可用性风险[2]的发生和持续时间。在非理想情况下，在每年一次的演习中进行问

题调查就很难，需要完全从零开始检查，而且很难确定这个潜在的问题在生产环境中存在多久了。

如果实验不是自动化的，那么它就不会被执行。

在Netflix，服务的可用性由该服务的开发和维护团队全权负责。我们的混沌工程团队通过培训、工具、鼓励和压力来帮助服务所有者提高服务的可用性。我们不能也不应该让工程师牺牲开发速度，专门花时间来手动定期执行混沌工程实验。相反地，我们要自己投入精力来开发混沌工程的工具和平台，以期不断降低创建新实验的门槛，并使这些实验能够完全自动运行。

混沌工程自动化平台（Chaos Automation Platform，ChAP）

我们的混沌工程团队在2015年的大部分时间里都以咨询的形式[3]，在关键微服务上运行混沌工程实验。这对于真正了解FIT的能力和局限非常必要，但是我们也知道这样手把手的咨询方式没有办法大规模进行混沌工程实验。我们需要一个可以在整个组织中将这个实践规模化的机制。

2016年年初，我们有了一个计划，将混沌工程的原则引入微服务层。我们注意到FIT有一些问题妨碍了自动化和广泛应用。其中一部分问题可以通过FIT自身解决，而解决另一部分问题则需要较大的工程量，比请求头修改和进程间通信故障点注入要复杂得多。

2016年年底，我们发布了混沌工程自动化平台，简称ChAP，旨在解决这些不足之处。

FIT 中的大多数问题都是由于缺乏自动化导致的。过多的人工参与，例如设置故障场景、观测运行时关键指标的变化等，被证明是大规模应用的障碍。我们倾向于依靠现有的金丝雀分析（参见第3章中对金丝雀分析的介绍）来自动判断实验是否在可接受的范围内执行。

随后我们使用一个自动化模板开始进行真正的实验。在上面讨论过的FIT的例子里，我们影响了5%的流量，观察这些流量对SPS的影响。如果没有观察到任何影响，我们会将受影响的流量提高到25%。任何影响都有可能被其他与SPS相关的噪声所掩盖。像这样用大量流量做实验是有风险的，会比较容易出现各种各样的故障，这会消减我们对系统的信心。所以，我们可以隔离一些小的影响，这样可以防止多个故障同时出现。

为了最小化爆炸半径，ChAP要为每个需要进行实验的微服务创建一个控制节点和一个实验集群。如上述实例中所说的，当测试一个用户信息的微服务时，ChAP会询问我们的持续集成工具Spinnaker关于这个集群的信息。ChAP用这个信息创建这个服务的两个节点，一个作

为控制节点，另一个作为实验节点。然后它会分流出一小部分流量，并平均分配在控制节点和实验节点上。只在实验节点里注入故障场景。当流量在系统中流转时，我们可以实时比对控制节点和实验节点上的成功率和操作问题。

有了自动化的实验，我们就拥有了较高的信心，我们可以通过一一比对控制节点和实验节点监测到即使很小的影响。我们只影响流量中很小的一部分，并且已经将实验进行了隔离，这样就可以并行运行大量的实验了。

2016年年底，我们将ChAP与持续交付工具Spinnaker集成在一起，这样微服务就可以在每次发布新版本时运行混沌工程实验了。这个新功能有些类似于金丝雀，但是在进行自动混沌工程实验时我们需要它不间断地运行，不会立即自动优雅降级，因为这里的目的是要尽可能发现所有潜在的系统问题。我们给予服务所有者，在发现微服务中的薄弱环节时，选择不触发降级的机会。。

自动创建实验

如果你已经可以配置定期自动运行的实验，那么你就处在一个非常好的状态了。然而，我们认为还可以追求一个更好的自动化水平：自动设计实验。

设计混沌工程实验的挑战并非来自定位导致生产环境崩溃的原因，这些信息在故障跟踪中就有。我们真正想要做的是，找到那些本不应该让系统崩溃的事件的原因，包括那些还从未发生过的事件，然后持续不断地设计实验进行验证，保证这些事件永远不会导致系统崩溃。

然而这是非常困难的。导致系统波动的原因非常多，我们不可能有足够的时间和资源穷举所有可能导致问题的事件及其组合。

正确路径驱动的故障注入（LDFI）

一个值得关注的关于自动创建实验的研究是一项叫作正确路径驱动的故障注入（Lineage-Driven Fault Injection，LDFI）的技术，由加州大学圣克鲁兹分校的PeterAlvaro教授开发。LDFI可以识别出可能导致分布式系统故障的错误事件组合，其工作原理是通过推断系统正常情况下的行为来判断需要注入的候选错误事件。

2015年，Peter Alvaro与Netflix的工程师合作，来研究是否可以把LDFI应用到我们的系统上。他们成功地在Netflix FIT框架的基础上开

发了一个LDFI的版本，并且识别出了可能导致严重故障的一些错误事件组合。

有关如何在Netflix开展这项工作的更多信息，请参阅第七届ACM云计算研讨会论文集（SoCC'16）上发表的论文《互联网规模的自动化故障测试研究》（*Automating Failure Testing Research at Internet Scale*，http://dx.doi.org/10.1145/2987550.2987555）。

---

[1]译者注：实现自动化只需要做一次，但是能收获无数次随时随地执行自动化实验所带来的好处，具有杠杆效应。

[2]译者注：由于新代码带来的风险在上线的过程中是很容易发现的，并可以很好地判断风险的发生和持续时间，但是将这类带有风险的代码部署到生产环境中后便很难再发现并做出判断。

[3]译者注：以项目的形式为服务的开发者提供如何开始和执行混沌工程实验的咨询意见，并帮助他们落地执行。

第7章　最小化爆炸半径

1986年4月26日，人类历史上最严重的核事故之一发生在乌克兰的切尔诺贝利核电站。具有讽刺意味的是，灾难是由于一次弹性演习导致的：一次验证冷却剂泵冗余电源的演习。虽然我们大多数人并不从事像核电站冷却系统这样高危项目的工作，但每一次的混沌工程实验的确具备导致生产环境崩溃的风险。混沌工程师的一项专业职责就是要理解和降低生产风险，可以为实验设计良好的系统，以阻止大规模的生产事故，将受影响的用户数量降到最少。

不幸的是，我们经常运行本来只会影响一小部分用户的测试，却由于级联故障无意中影响到了更多的用户。在这些情况下，我们不得不立即中断实验。虽然我们绝不想发生这种情况，但随时遏制和停止实验的能力是必备的，这可以避免造成更大的危机。我们的实验通过很多方法来探寻故障会造成的未知的和不可预见的影响，所以关键在于如何让这些薄弱环节曝光出来而不会因意外造成更大规模的故障。我们称之为“最小化爆炸半径”。

能带来最大信心的实验也是风险最大的，是对所有生产流量都有影响的实验。而混沌工程实验应该只承受可以衡量的风险，并采用递进的方式，进行的每一步实验都在前一步的基础之上。这种递进的方式不断增加我们对系统的信心，而不会对用户造成过多不必要的影响。

最小风险的实验只作用于很少的用户。为此在我们验证客户端功能时只向一小部分终端注入故障。这些实验仅限于影响一小部分用户或一小部分流程。它们不能代表全部生产流量，但却是很好的早期指标。例如，如果一个网站无法通过早期实验，那么就没有必要影响其余的大量真实用户。

在自动化实验成功之后（或者在少量的设备验证没有涵盖要测试的功能时），下一步就是运行小规模的扩散实验。这种实验会影响一小部分用户，因为我们允许这些流量遵循正常的路由规则，所以它们最终会在生产服务器上均匀分布。对于此类实验，你需要用定义好的

成功指标[1]来过滤所有被影响的用户，以防实验的影响被生产环境的噪声掩盖。[2]小规模扩散实验的优势在于，它不会触及生产环境的阈值，例如断路器的阈值，这样你便可以验证每一个单一请求的超时和预案。这可以验证系统对瞬时异常的弹性。

接下来是进行小规模的集中实验，通过修改路由策略将所有实验覆盖的用户流量导向特定的节点。在这些节点上会做高度集中的故障、延迟等测试。在这里，我们会允许断路器打开，同时将隐藏的资源限制暴露出来。如果我们发现有无效的预案或者奇怪的锁竞争等情况导致服务中断，那么只有实验覆盖的用户会受到影响。这个实验模拟生产环境中的大规模故障，同时可以把负面影响控制到最小，结果却能使我们对系统建立高度的信心。

风险最大但最准确的实验是无自定义路由的大规模实验。在这个实验级别，实验结果应该在主控制台上显示，同时因为断路器和共享资源的限制，实验可能会影响不在实验覆盖范围内的用户。当然，没有什么比让所有生产环境中的用户都参与实验，能给你更多关于系统可以抵御特定故障场景的确定性了。

除了不断扩大实验范围，在实验造成过多危害时及时终止实验也是必不可少的。有些系统设计会使用降级模式来给用户带来较小的影响，这还好，但是在系统完全中断服务的时候，就应该立即终止实验。这可以由之前讨论过的“大红色按钮”来处理。

我们强烈建议实施自动终止实验，尤其是在定期自动执行实验的情况下。关于弄清楚如何构建一个可以实时监控我们感兴趣的指标，并可以随时实施混沌工程实验的系统，这完全依赖于你手上的独特的系统构造，我们将此留给读者当作课后练习。

为了尽可能高效地应对实验发生不可预料的情况，我们要避免在高风险的时间段运行实验。例如，我们只在所有人都在办公室工作的时间段运行实验。

如果实验的工具和仪器本身就会对定义好的指标产生影响，那么整个混沌工程的目的就被破坏了。[3]我们要的是建立对系统弹性的信心，记住每次只检验一个可控的故障。

[1]译者注：衡量实验是否成功的指标。

[2]译者注：如果没有定义区分度足够高的成功指标，就不容易区分出在所有被影响到的用户里，哪些是真的通过了验证的，哪些是碰巧没触及要测试的点也通过了验证的。比如，将“到达流程最后一步”定义为成功指标，结果所有被影响的用户都到达了最后一步，但是因为所走路径不同，所以难以区分哪些是正常到达最后一步的，哪些是异常到达的。

[3]译者注：如果工具和仪器的使用会造成结果偏移，那么实验就没有意义了。比如，实验本身会造成系统负载大增，延迟大涨，那么得出的结论和后续要做的应对措施就都没有意义了。

第三部分　混沌工程实践

在第一部分和第二部分中，我们讨论了混沌工程的初衷和背后的理论。然而，将理论付诸实践是具有挑战性的。任何系统都没有现成的用于构建、调度、自动化实验的工具，而且即使是最好的混沌工程框架，也需要与实际项目适配好才能发挥作用。

混沌工程起源于Netflix，但它的影响力已经从技术领域扩展到了其他行业。例如，在一些会议上，我们经常听到来自金融行业的工程师们的一些保留意见。他们不愿意实施混沌工程实验的原因是，害怕实验会给客户带来资金或行业合规性的影响。然而我们坚持认为，该发生的故障，不会受个人意志的影响而不发生。即使实验暴露风险点的同时会导致一些小的负面影响，但是提前了解和控制影响范围，也比最终措手不及地应对大规模事故要好得多。现在我们已经看到很多银行和金融公司在实践混沌工程，这为那些犹豫不决的行业提供了很多正面案例。

医疗行业的工程师们也表达了类似的疑虑。诚然，一个娱乐服务的中断只会让人觉得不方便，金融交易的失败则会令人感到郁闷并可能导致较大的资金损失，但医疗技术领域的任何失败都可能会危及生命。然而我们还是想指出，使混沌工程规范化的许多科学原则正是起源于医学的。医学研究的最高标准就是临床试验。我们绝不会忽视将混沌工程引入医疗领域会带来的潜在影响，我们只是想说明，实际在医疗领域的临床试验中已经存在很多和混沌工程有同样风险的先例，而临床试验中所存在的风险已经被人们广泛接受了。

对于混沌工程这样一个新生的学科来说，付诸实践是最有意义且最迫切的。了解如何实施，实施的复杂性，以及实施时需要关注的问题，既可以帮助你明确自己目前在混沌工程之路上所处的位置，也可以为你指明你所在的组织要想进行成功的混沌工程实践该在何处发力。

第8章　设计实验

在讨论完理论原则之后，我们来看一看设计混沌工程实验的细节。下面是一个大致的流程：

1. 选定假设。
2. 设定实验的范围。
3. 识别出要监控的指标。
4. 在组织内沟通到位。
5. 执行实验。
6. 分析实验结果。
7. 扩大实验范围。
8. 自动化实验。

选定假设

你需要做的第一件事就是选定要验证的假设，我们在第4章中讨论过。比如，你最近发现在访问Redis时有超时报错的情况，于是你想确保系统不会被缓存访问超时所影响。再比如，你希望验证主从数据库配置，在主数据库故障时可以无缝切换到从数据库。

不要忘记，系统的一个重要组成部分是维护它的人。人工的行为对于降低事故率至关重要。比如，有的公司采用如Slack或HipChat之类的即时通信工具进行故障时的沟通，同时对于即时通信工具不可用时如何处理故障，也有一套应急流程。那么，值班的工程师如何很好地掌握这个应急流程呢？执行一个混沌工程实验就是很好的办法。

设定实验的范围

选定了要验证的假设后，下一步需要做的就是设定好实验的范围。这里有两个原则：“在生产环境中运行实验”和“最小化爆炸半径”。实验离生产环境越近，我们能从实验中获得的收益就越大。虽然这么说，但还是要仔细关注可能对系统和用户造成影响的风险。

我们需要尽可能地将实验对用户造成的影响降到最低，所以我们应该从一个最小范围的测试开始，然后一步步扩大范围，直到我们认为系统可以应对我们预期中最大的影响。

因此，如第7章中描述过的，我们提倡将第一个实验的范围控制得越小越好。而且在生产环境中执行实验之前，应该先在测试环境中试一试。一旦进入生产环境，就一定要从最少的用户流量开始尝试。例如，如果你要验证系统在缓存超时时会怎么做，你可以先用一个测试客户端来调用生产环境的服务，并且只针对这个客户端引入缓存超时。

识别出要监控的指标

明确了假设和范围之后，我们需要选定用来评估实验结果的指标，我们在第3章讨论过。要尽可能基于指标来验证假设。例如，你的假设是“即使主数据库出现故障，所有服务也都能正常运行”，那么在运行实验之前，你需要清晰地定义什么是“正常”。再举个例子，如果你有一个明确的业务指标，比如“每秒订单数”，或者更低级别的指标，比如“响应时间”和“响应错误率”，那么在执行实验之前要明确定义清楚这些指标的合理数值范围。

一旦实验产生的影响比预料中的严重，你就应该做好立即终止实验的准备。可以预先设定好明确的阈值，例如失败请求占比不超过5%。这可以帮助你快速决定，在实验进行时要不要按下那个“大红色按钮”。

在组织内沟通到位

当你第一次在生产环境中执行混沌工程实验时，你需要通知所在组织中的其他成员：你将要做什么，为什么要这么做，以及什么时间要做。

第一次执行的时候，你可能需要协调好每个需要参与的、对结果感兴趣的，以及对产生的影响有顾虑的团队。随着执行实验的次数越来越多，你和你的组织会获得越来越充足的信心，这时就没有必要每次都为了实验事先向大家发送通知了。

关于混乱金刚的通知

> 我们第一次执行混乱金刚的实验来验证AWS区域故障恢复的实践效果时，整个过程中与公司各团队之间进行了大量的沟通，目的是让每一个人都清楚区域失效的时间点。这个过程中不可避免地会有很多例如功能发布等的要求被提出，我们只能将实验推迟。
>
> 随着我们一次又一次越来越频繁地进行实验，混乱金刚逐渐被大家看作一个“常规”事件。结果就是事先沟通的次数变得越来越少。现在，我们每三周执行一次，但已经不需要事先发出通知了。我们内部有一个可以订阅的日历会标明混乱金刚实验在哪一天执行，但我们不会指明当天具体的执行时间。

执行实验

截至目前，准备工作都已经完成，是时候执行实验了！要盯住那些指标，因为你可能随时需要终止实验。随时终止实验的能力异常关键，因为现在是直接在生产环境中运行实验，随时都可能对系统造成过度危害，进而影响外部用户的使用。例如对于一个电子商务网站，你一定要密切关注用户是否能够正常结算。要确保有足够的报警机制，能实时获悉这些关键指标是不是掉到了阈值以下。

分析实验结果

实验结束后，用“选定假设”一步中制定的指标数据来验证之前的假设是否成立。系统对于注入的真实事件是否具备足够的弹性？有没有任何预料之外的事情发生？

多数混沌工程实验暴露出来的问题都会涉及多服务之间的交互，所以要确保把实验结果反馈给所有相关的团队，一同从整体的角度来消除隐患。

扩大实验范围

如第7章中所描述的，当你从小范围实验中获得了信心之后，就可以逐步扩大实验范围了。扩大实验范围的目的是进一步暴露小范围实验无法发现的一些问题。例如，微服务架构中存在少量的超时或许不会有什么问题，但超过一定比例就可能会导致系统整体瘫痪。

自动化实验

如第6章中所描述的，当你有信心手动执行混沌工程实验之后，就可以开始周期性地自动化运行实验，持续从中获得更大的价值了。

第9章　混沌工程成熟度模型

我们将混沌工程的定义标准化的一个目的是，在执行混沌工程项目时，可以有标准来判断这个项目做得是好是坏，以及如何可以做得更好。混沌工程成熟度模型（CMM）给我们提供了一个评估当前混沌工程项目成熟度状态的途径。把当前项目的状态放在成熟度模型图上，你就可以据此设定想要达到的目标，也可以和其他项目的状态进行对比。如果你想要提升这个项目的状态，CMM的坐标轴会给出明确的方向和建议——应该朝哪里努力。

CMM的两个坐标轴分别是“熟练度”和“应用度”。缺乏熟练度的时候，实验会比较危险、不可靠，且有可能是无效的。缺乏应用度的时候，所做的实验就不会有什么意义和影响。要在适当的时候变换在两个不同维度上的投入，因为在任何一个时期，要发挥混沌工程项目的最大作用都需要在这两个维度上保持一定的平衡。

熟练度

熟练度可以反映出，组织中混沌工程项目的有效性和安全性。项目各自的特性会反映出不同程度的熟练度，有些完全不具备熟练度，而有些可能具备很高的熟练度。熟练度的级别也会因为混沌工程实验的投入程度而有所差异。我们一般用入门、简单、高级和熟练四个级别对熟练度进行描述。

入门

□·未在生产环境中运行实验。

□·全人工流程。

□·实验结果反映系统指标，而不是业务指标。

□·对实验对象注入一些简单事件，如“关闭节点”。

简单

□·用复制的生产环境中的流量来运行实验。

□·自助式创建实验，自动运行实验，手动监控和停止实验。

□·实验结果反映聚合的业务指标。

·□对实验对象注入较高级的事件，如网络延迟。

□·实验结果是手动整理的。

□·实验的定义是静态的。

□·具有可以支持对历史实验组和控制组进行比较的工具。

高级

□·在生产环境中运行实验。

□·自动分析结果，自动终止实验。

□·实验框架和持续发布工具集成。

□·在实验组和控制组之间比较业务指标差异。

□·对实验组引入一些事件，如服务级别的影响和组合式的故障。

□·持续收集实验结果。

·□具有可以交互式地比对实验组和控制组的工具。

熟练

·在开发流程中的每个环节及所有环境中运行实验。

□·设计、执行和终止实验完全自动化。

□·将实验框架和A/B测试以及其他测试工具集成，以减少噪声干扰。

□·可以注入如对系统的不同使用模式、返回结果和状态的更改等类型的事件。

□·实验具有动态可调整的范围以找寻系统拐点[1]。

□·实验结果可以用来预测收入损失。

·□对实验结果的分析可以用来做容量规划。

·□实验结果可以用来区分服务实际的关键程度。

应用度

应用度用来衡量混沌工程实验覆盖的广度和深度。应用度越高，暴露的脆弱点就越多，我们对系统的信心也就越充足。类似熟练度，我们也对应用度的不同级别进行了定义，分别是“暗中进行”、适当投入度、正式采用和成为文化。

“暗中进行”

□·对重要项目不采用。

□·只覆盖少量系统。

□·组织内部基本没有感知。

·□早期使用者偶尔进行混沌工程实验。

适当投入度

□·实验获得正式批准。

□·工程师兼职进行混沌工程实验。

□·多个团队有兴趣并参与进来。

□·一些重要服务会不定期地进行混沌工程实验。

正式采用

□·有专门的混沌工程团队。

□·所有故障的复盘都会进入混沌工程框架来创建回归实验。

□·定期对大多数关键服务进行混沌工程实验。

□·偶尔进行实验性的故障复盘验证，例如通过“比赛日”的形式来做。

成为文化

□·对所有关键服务进行高频率的混沌工程实验。

□·对多数非关键服务进行高频率的混沌工程实验。

□·混沌工程实验是工程师入职流程的一部分。

□·所有系统组件都默认要参与混沌工程实验，不参与的话需要进行特殊说明。

绘制成熟度模型图

绘制成熟度模型图时，以熟练度作为纵轴，应用度作为横轴。四个象限如图9-1所示。

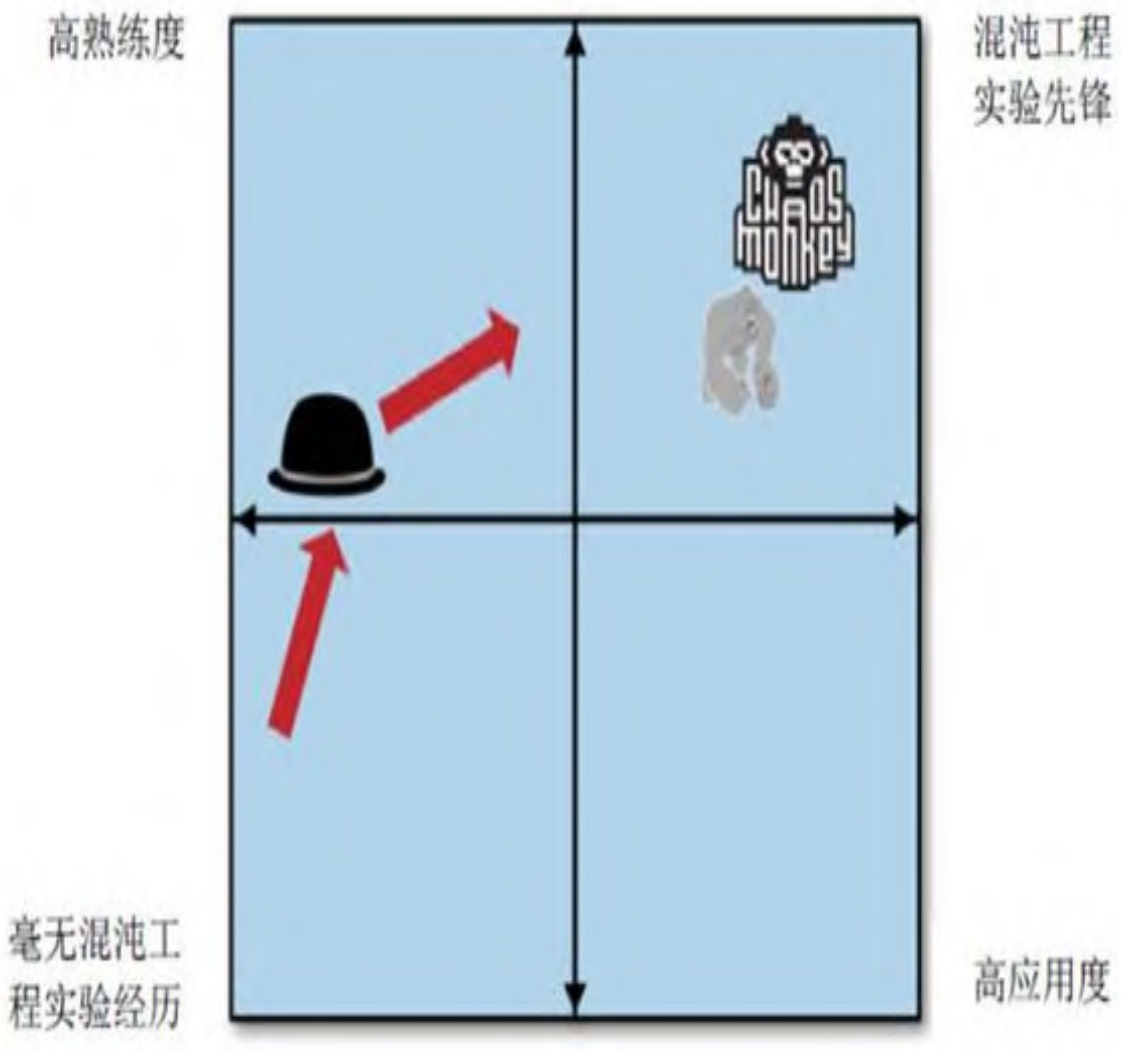

图9-1　成熟度模型图

在图9-1中，我们用混乱猴子、混乱金刚和混沌工程自动化平台ChAP（图中的帽子）作为示例。在进行本书的写作时，我们的ChAP处在熟练度较高的位置，在之前的进展中，我们努力的方向如图9-1中的箭头所示。现在，根据图9-1就知道，我们需要专注在它的应用度上，以发挥出ChAP的最大潜力。

CMM可以帮助我们理解混沌工程项目处在什么状态，建议我们应该聚焦在哪方面来进行持续提升，同时给我们指出改进的方向。

---

[1]译者注：通过不断扩大爆炸半径，即扩大流量和影响范围，找到例如雪崩这样的问题的发生点，这类问题的发生通常都有一个临界点，用一两个单独

的条件测试时一切都正常，但给予其一定的流量压力，或使用某些调用的组合，就会引发问题，这个临界点就是拐点。

第10章　结论

我们相信，在任何一个开发和运行复杂分布式系统的组织机构里，如果既想拥有高开发效率，又想保障系统具有足够的弹性，那么混沌工程一定是必备的方法。

混沌工程目前还是一个非常年轻的领域，相关的技术和工具也都在逐步发展中。我们热切地期望各位读者可以加入社区，和我们一起不断实践、拓展混沌工程。

一些资源

我们创建了社区网站（http://chaos.community/）和Google Group（https://groups.google.com/forum/#!forum/chaos-community）。期待你加入我们。

你也可以在Netflix的官方博客（https://medium.com/ netflix-techblog）上找到更多关于混沌工程的信息。当然，还有很多其他组织在实践混沌工程，大家可以参考以下文章：

·□*Fault Injection in Production: Making the Case for Resiliency Testing*

·□*Inside Azure Search: Chaos Engineering*

□·*Organized Chaos With F#*

□·*Chaos Engineering 101*

□·*Meet Kripa Krishnan, Google's Queen of Chaos*

□·*Facebook Turned Off Entire Data Center to Test Resiliency*

·□*On Designing And Deploying Internet-Scale Services*

另外，还有很多为不同场景开发的开源工具：

Simoorg

Simoorg是LinkedIn开发的故障注入工具。它非常易于扩展，并且其中的很多关键组件都是可插拔的。

Pumba

Pumba是基于Docker的混沌工程测试工具以及网络模拟工具。

Chaos Lemur

Chaos Lemur是用来测试高可用性系统弹性的工具，可以在本地进行部署，允许随机关闭BOSH虚拟机。

Chaos Lambda

Chaos Lambda是在办公期间随机关闭AWS ASG节点的工具。

Blockade

Blockade是基于Docker的、可以测试网络故障和网络分区的工具。

Chaos-http-proxy

Chaos-http-proxy可以向HTTP请求注入故障的代理服务器。

Monkey-Ops

Monkey-Ops是用Go语言实现的，可以在OpenShift V3.X上部署并且可以在其中生成混沌工程实验。Monkey-Ops可以随机停止OpenShift组件，如Pod或者DeploymentConfig。

Chaos Dingo

Chaos Dingo目前支持在Azure相关服务上进行实验。

Tugbot

Tugbot是在使用Docker的生产环境中进行测试的框架。

也有一些书籍讲到关于混沌工程的一些主题：

***Drift Into Failure*，Sidney Dekker（2011）**

Dekker的理论是，在一个组织内部，事故的发生都是因为各系统随着时间慢慢滑向一个不安全的状态，而不是因为某个组件突发问题或操作人员操作失误。你可以把混沌工程想象成专门用来对抗这种滑动过程的方法。

To Engineer Is Human: The Role of Failure in Successful Design，Henry Petroski（1992）

Petroski描述了他们那里的工程师是如何从过去的失败中汲取教训并进步的，而不是从过去的成功中找经验。混沌工程正是一种既能发现系统中的问题点，又能避免对系统造成大规模影响，将爆炸半径尽可能控制到最小的方法论。

***Searching for Safety*，Aaron Wildavsky（1988）**

Wildavsky的主张是，为了提高整体的安全性，必须要管理好所有风险。尤其是采用不断试错的方法，长期来说比尝试完全避免事故发生获得的结果更好。混沌工程同样也是通过在生产环境中进行实验来拥抱风险，以期获得系统更大的弹性的方法。

O'REILLY®

混沌工程

Netflix系统稳定性之道

本书凝聚了第一批混沌工程师的智慧和经验，详细地阐述了混沌工程的演进和实践原则。如果你对混沌工程感兴趣，愿意去了解和实践混沌工程，非常建议你从阅读本书开始行动。

周洋（花名：中亭）
阿里巴巴高可用架构团队高级技术专家
开源项目ChaosBlade发起人

上架建议：计算机>大型网站
责任编辑：刘恩惠
封面设计：李　玲

Broadview®
WWW.BROADVIEW.COM.CN

ISBN 978-7-121-36351-1
9 787121 363511 >
定价：45.00元

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)