Netflix 混沌工程手册 Part 1：混沌工程简介

作者：Netflix工程师团队

* 侯杰

阅读数：9809 2019 年 2 月 15 日 08:00

*本文翻译自 Netflix 工程师合著的*[*Chaos Engineering*](https://learning.oreilly.com/library/view/chaos-engineering/9781491988459/)*一书。这本书介绍了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更广泛地应用于其他领域。*

*InfoQ 将就这一专题持续出稿，感兴趣的同学可以持续关注。*

**译者序**

近年来，随着服务化、微服务和持续集成的逐渐普及，从开发到线上的便捷性大幅提高。我们在使用这些便利性所带来的好处的同时，对其负面性的问题也要有所认知。尤其是在一个复杂的分布式服务体系中，故障发生的随机性和不可预测性都大大增加了。快速迭代的门槛越来越低，但是对复杂系统稳定性的考验却在成倍增长。在这条路上，各家都有自己的经验实践。

Chaos Engineering 混沌工程从出现到标准化成为一门学科，是伴随着 Netflix 过去三年多时间里同稳定性持续战斗的历程一起成长起来的，这是每一次故障带来的深度思考，抽象而成的理论和实践的结合。混沌工程是一门相对高级的系统稳定性治理方法论，它提倡采用探索式的研究实验，发现生产环境中的各种风险。为什么说相对高级，因为成功实施混沌工程，要求对现有系统的弹性有一定信心。

**前言**

*混沌工程是一门新兴的技术学科，他的初衷是通过实验性的方法，让人们建立对于复杂分布式系统在生产中抵御突发事件能力的信心。*

*——混沌工程原则*

只要你有过在生产环境中实际运行过分布式系统的经历，你就应该清楚，各种不可预期的突发事件一定会发生。分布式系统天生包含大量的交互、依赖点，可以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，我们可以一直列举下去。这都是每天要面临的常事儿，处理不好就会导致业务停滞，性能低下，或者是其他各种无法预期的异常行为。

在复杂的分布式系统中，人力并不能够阻止这些故障的发生，我们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，我们就可以有针对性地进行加固，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（弹性：系统应对故障、从故障中恢复的能力）系统的同时，树立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。这种通过实证的验证方法显然可以为我们打造更具弹性的系统，同时让我们更透彻的掌握系统运行时的各种行为规律。

实践混沌工程可以简单如在 STG 环境的某个实例上运行 kill -9 来模拟一个服务节点的突然宕机，也可以复杂到在线上挑选一小部分（但足够代表性）的流量，按一定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 开始从数据中心迁移到云上，之后就开始尝试在生产环境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之为混沌工程。最早被大家熟知的是“混乱猴子”（Chaos Monkey），以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，这些之前获得的小收益被无限扩大。规模的扩大得益于一个叫做“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，用以将这个实践规范的学科化 ，同时我们推出了混沌工程自动化平台，能够在微服务体系架构上，24\*7 不间断地自动运行混沌工程实验。*

*在开发这些工具和实践的过程中，我们逐渐意识到，混沌工程并非是简单的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、高效地发现问题。混沌工程的意义在于，能让复杂系统中根深蒂固的混乱和不稳定性浮出表面，让我们可以更全面地理解这些系统性固有现象，从而在分布式系统中实现更好的工程设计，不断提高系统弹性。*

**1. 为什么需要混沌工程**

混沌工程是一种通过实证探究的方式来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。

实践混沌工程是如何提高系统弹性的呢？它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、或让用户受损的脆弱环节，推动我们主动解决这些环节。相比现在各大公司主流的被动式故障响应流程，混沌工程向前迈进了一大步。

1.1 混沌工程和测试的区别

混沌工程，故障注入 FIT 和故障测试在侧重点和工具集上有一些重叠。举个例子，在 Netflix 的很多混沌工程实验研究的对象都是基于故障注入来引入的。混沌工程和这些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，而故障注入则是对一个特定的条件、变量的验证方法。

例如当你希望探究复杂系统如何应对异常时，对系统中的服务注入通信故障，如超时，错误等，这是一个故障注入的典型场景。但有时我们希望探究更多其他的非故障类的场景，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有错误的响应、异常的行为、对调用者随机性的返回不同的响应，等等）、非计划中的或非正常组合的消息处理，等等。因为如果一个面向公众用户的网站突然收到激增的流量，从而产生更多的收入，我们很难称之为故障，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，故障测试方法通过对预先设想到的可以破坏系统的点进行测试，但是并没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：即给定一个特定的条件，系统会输出一个特定的结果。测试一般来说只会产生二元的结果，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，它仅仅是对我们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。这整本书我们都是在探讨这个主题——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、集成测试等以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故障；
* 跨多实例删除部分 Kafka topic 来重现生产环境中发生过的问题；
* 挑选一个时间段，和针对一部分流量，对其涉及的服务间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；
* 在代码中插入一些指令可以允许在这些指令之前运行故障注入；
* 强制系统节点间的时间不同步；
* 在驱动程序中执行模拟 I/O 错误的程序；
* 让某个 Elasticsearch 集群 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。

1.2 混沌工程绝不仅是 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。”

虽然我们提供的案例都来自于 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，所介绍的设计实验的指导也没有基于任何特定的架构或者工具集。后面，我们会讨论混沌工程成熟度模型，希望评估一下自身为什么，在什么时间点，以什么方式进行混沌工程实践的读者可以采用。

像最近的一次混沌工程社区日（一个来自不同组织的混沌工程实践者的聚会），参会者来自 Google，Amazon，Microsoft，Dropbox，Yahoo!，Uber，[cars.com](http://cars.com/" \t "_blank)，Gremlin Inc.，加州大学圣克鲁兹分校，SendGrid，北卡罗莱纳州立大学，Sendence，Visa，New Relic，[Jet.com](http://jet.com/" \t "_blank)，Pivotal，ScyllaDb，GitHub，DevJam，HERE，Cake Solutions，Sandia National Labs，Cognitect，Thoughtworks，and O’Reilly 出版社。在本书里，你会看到来自各行各业（金融，电商，航空航天，等等）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。交易依赖复杂系统吗？有大型银行正在使用混沌工程来验证交易系统是否有足够的冗余。是否有人命悬一线？在美国，混沌工程在许多方面被当做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、再到数字巨头和创业公司，混沌工程正在成为复杂系统改进学科的立足点。

***飞机启动失败？***

*在伊利诺伊大学的香槟分校，Naira Hovakimyan 和她的研究团队把混沌工程用在了喷气式战斗机上。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突然变化和空气动力学参数变化！这给团队能否在故障发生时重新构建机体爬升动力学参数，以及应对其他会导致故障的配置都带来了极大的挑战。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。*

1.3 混沌工程的前提条件

在判断你的组织是否已经准备好实施混沌工程之前，需要回答这样一个问题：你的系统是否已经具备一些弹性来应对真实环境中的一些异常事件，像某个服务异常、或网络闪断、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。混沌工程非常适合于揭示生产系统中未知的脆弱环节，但如果你很确定混沌工程实验会导致系统出现严重的故障，那运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，然后你不仅能继续发现更多不知道的脆弱点，还能提高对系统真实弹性水平的信心。

混沌工程的另一个前提条件是监控系统，你需要用它来判断系统当前的各项状态。如果没有对系统行为的可见能力，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，对于如何针对混沌工程揭示出的脆弱环节进行根本原因分析，我们留给读者作为练习。

***混乱猴子（Chaos Monkey）***

*2010 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之前的几年开始迁移到云上。之前的数据中心垂直扩展给 Netflix 带来过很多单点故障，其中一些故障甚至大规模中断了当时的 DVD 业务。云服务不光带来了水平扩展的机会，同时可以把重度的基础设施运维工作转移到可靠的第三方。*

*数据中心本就会时不时的发生一些小故障，然而到了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，发生这类故障的几率也大幅增加。数以千计的服务节点里，时不时就会有节点出现异常或者掉线。所以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随时应对节点故障。*

*在 Netflix，并没有制度要求工程师一定要按照某种规定来构建任何东西。相反，高效的领导者在工程师之间建立强有力的一致规约或原则，然后让工程师在自己的领域里找到解决问题的最好办法。在节点随时会发生故障的案例里，我们要建立的强力规约和原则就是，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。*

*混乱猴子在业务正常进行的时间段内，伪随机地关闭生产环境中正在运行中的节点，而且关闭的频率比正常节点故障频率还要高很多。通过高频率的触发这些不常见的且具备灾难性的事件，我们给了工程师强大的动力在开发他们的服务时必须考虑到如何轻松应对这类事件。工程师必须尽可能提早并快速地处理这类故障。再加上自动化、冗余、回滚策略，以及其他弹性设计的最佳实践，工程师很快就可以让自己的服务在这些故障发生时也能保持正常运行。*

*经过几年的发展，混乱猴子逐渐变得更强大，现在它可以指定终止一组节点，并且通过与 Spinnaker（持续发布平台）集成进行自动的线上实验。但从根本上它还是提供 2010 年以来一样的功能。*

*混乱猴子最大的成就在于让我们的工程师之间形成了构建具备足够弹性服务的规约和原则。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，仅有一次节点掉线影响了我们的服务。当时正巧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，因而造成了问题。幸运的是，这个故障发生在白天工作时间，这个故障的服务刚刚部署不久，对用户的影响也非常小。可想而知，如果这个服务一直在线上运行了几个月，混乱猴子在某个周末的晚上终止了它的节点，且负责该服务的工程师没有在 on-call 的情况下，将造成多大的灾难。*

*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的痛苦提到最前，同时让所有工程师在构建具有足够弹性应对失败的系统上，达成一致的目标。*

**2. 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套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**

​ 在这里特指对延迟或资源成本的最小化。

**可用性**

​ 系统正常响应和避免停机的能力。

**容错能力**

​ 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**

指工程师可以把新功能，创新功能提供给用户的速度。

Netflix 在软件工程中的决策过程中，非常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，可以为架构选型的决策提供必要的信息。

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：

任何组织在设计一套系统（广义概念）时，所交付的设计方案在结构上都与该组织的通信结构保持一致。

*Melvin Conway, 1967*

在微服务架构中，各团队彼此独立开发和运营自己的服务。每个团队可以自行决定何时将代码送入生产环境。这个架构策略以沟通协调为代价来提高新功能开发速度，通常工程组织被划分为许多个这样的小型团队。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（即每个人都能看到更全面更大的全景，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）。小团队之间的有效沟通是成功实施微服务架构的关键。混沌工程适时的提供了系统弹性的验证能力，来支持快速功能开发，实验，以及团队对系统的信心。

2.1 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。如下图，这个服务由 7 个微服务组成，从 A 到 G。A 服务存储了用户的个人信息。B 服务存储用户的登录账户信息，如用户上一次登录的时间和访问了什么信息。C 服务是关于产品信息的。D 服务作为 API 网关处理所有来自外部的接口访问。

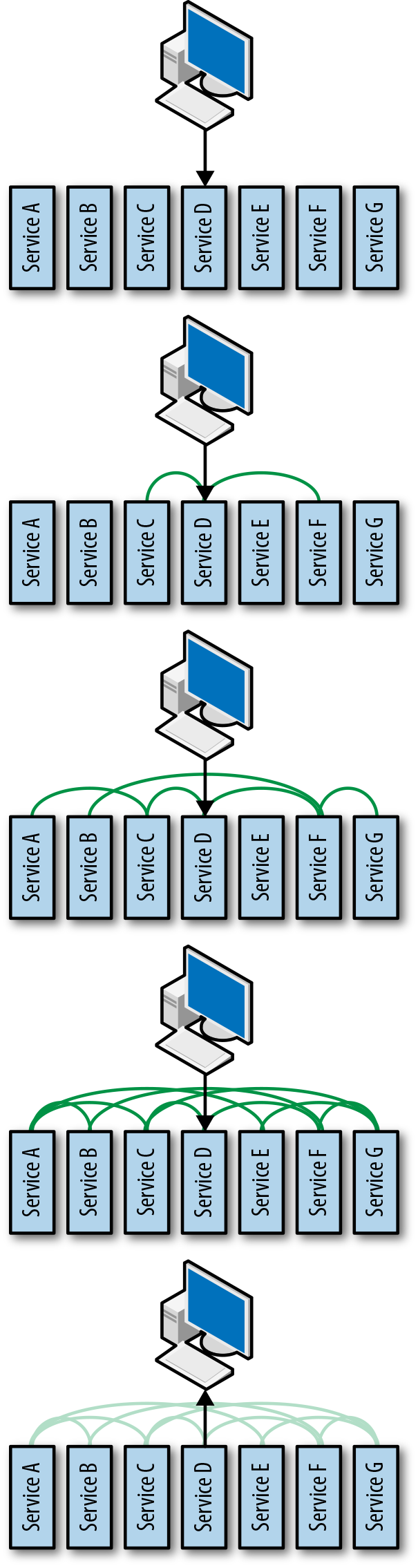


图 1 微服务架构

来看一个请求的例子，用户通过手机 App 访问了一些信息：

* 请求首先进入 D 服务，即 API 服务；
* D 服务本身并没有所有需要的信息，所以它进一步请求 C 服务和 F 服务以获取必要的数据；
* C 服务和 F 服务也同时都需要更多的信息来满足请求，于是 C 服务请求 A 服务，F 服务请求了 B 服务和 G 服务；
* A 服务也需要访问 B 服务，B 服务需要访问 E 服务，同时 G 服务也需要访问 E 服务；
* 对 D 服务的一个请求，扩散到了整个微服务架构里。而且当所有依赖的服务没有返回或者超时之前，API 不会向手机 App 返回响应。

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。相比紧密耦合的单体应用来说，有趣的是传统架构师的角色被显著削弱了。传统架构师的角色更多是负责理解系统中各个组成部分是如何组成整个系统的，以及他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脑中。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，即使是一个单块系统，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。如果人类揭开这些算法的盖子来看看他们的内部构造时，一系列的权重和非无效解产生的浮点数对个人理解来说就太困难了。只有系统整体发出的响应才能被人类所理解。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，意大利面条式的调用图代表了典型的，需要混沌工程关注的系统固有混乱。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。

2.2 系统复杂性示例

E 服务包含提供用户定制化体验的信息，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去 A 服务获取用户的信息，然后去服务 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设计和运行的做一些合理的假设。由于请求数量很大，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服务，而不是所有 A 服务的节点都面对整个用户群。例如在 A 服务背后的数百个节点中，所有来自用户“CLR”的请求只会被路由到节点 A42。如果 A42 出现任何问题，足够智能的路由逻辑会将 A42 的职责路由到集群中的其他节点。

如果下游所依赖的服务出现异常，A 服务有合理的预案。如果 A 服务无法和持久层通信，它就从本地缓存中返回结果。

在运行时，每个微服务会平衡监控、报警和资源的考量，以合理的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。扩展规则会基于 CPU 负载和 I/O 性能来决定是否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，它在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。

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的发送出去。因为用户“CLR”发起的是对首页的请求，所以 E 服务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 E 服务的请求都会先请求 A 服务。于是 A 服务同时被打开首页和其他服务（如 E 服务）请求了数百次。由于 A 服务的架构设计，所有“CLR”的请求都被路由到了节点 A42。A42 在这么大流量下无法对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。

从缓冲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，A42 的 CPU 和 I/O 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。于是策略考虑资源的有效利用，对集群进行了收缩，对 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的处理本属于 A42 需要做的工作。比如 A11 接管了来自“CLR”的请求。

在 A42 转移到 A11 的过程中，E 服务中对 A 服务的请求超时了。于是 E 服务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。A11 比平时需要处理的工作更多了，所以它也逐渐切换到从缓存中返回一些稍稍过时的信息。CPU 和 I/O 相应的降低了，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，这又触发了更多的对 A 服务的请求。额外的流量压力致使 A 服务中更多的节点选择从缓存中返回信息，CPU 和 I/O 进一步相应降低，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，A 服务掉线了。B 服务对于 A 服务掉线没有预案，于是进一步拖垮了 D 服务，于是整个服务基本上都中断了。

2.3 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导致了系统预期之外的行为。这一类交互的复杂性不是人力可以完全预期到的。每一个微服务都可以被全面的测试覆盖，但我们任然不能在任何测试场景或集成测试环境中看到这类行为。

期待任何人类的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期之外的系统效应，都是不合理的。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，我们就可以为这些又庞大又充满迷雾，无法被某一个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（Chaos Monkey）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，而混乱金刚是用来关闭整个 AWS 区域。*

*Netflix 视频的每一个字节都来自于 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，我们先把混沌工程问题放在一旁，现在先聚焦在 Netflix 其他的一些服务上，我们称之为 Netflix 服务的“控制平面”（Control Plane）。（这类服务类似于路由器中执行 Routing 功能的部分。）*

*除了视频流媒体来自 CDN 之外，所有其他与服务的交互都是由 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，云上应用处理全流程的服务，包括启动、注册、浏览、选择视频、播放，播放时的心跳检测，等等。*

*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，这个事故迫使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，可以把它们想象为多个数据中心。通过多区域的故障恢复策略，我们可以把所有用户从故障的区域转移到另一个，最大限度地控制单个区域中断造成影响的范围和时长。*

*这项工作需要负责微服务架构的多个团队之间进行沟通协调。我们在 2013 年底完成了混乱金刚，我们打算用它来关闭整个 AWS 区域。这个硬性的强制促进了工程师们建造可以在 AWS 区域间平滑过渡服务的一致目标。这里我们只是模拟整个区域中断的故障，毕竟我们没有权限把整个 AWS 区域真正中断掉。*

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

以上是本书的第一部分，接下来的第二部分将会介绍**混沌工程原则**，有兴趣的同学可以持续关注。