



DevOps 的概念与实践 (2014年)



本迷你书由  亚马逊 AWS 赞助



InfoQ ueue

写在前面的话

本书总结了运维领域在 2014 年对 DevOps 的实践与新的理解。

DevOps 并非是一个新的概念，为何每年还需要回顾它？

这就好像敏捷开发一样。我们一开始只是听说有这样一个好的实践，有些人开始用了，有些人还没开始用；有些人用得已经比较深了，有些人则才刚刚开始掌握这个实践的指导思路。

在掌握新概念、践行新实践的过程中，每个人都会都有自己的一套理解。随着这个过程的深入，我们对“什么是 DevOps，什么不是 DevOps”这个问题会有更多自己的解读。

但是，重要的并非是我们所做的事情是否贴合概念本身，而在于我们实践的成果是否带来了这个概念所阐述的好处。这是为什么概念的回顾必须跟实践经验一同检验的原因。

祝大家能通过本书内容，更好的检验自己团队的 DevOps 进程！

InfoQ 中文站

目 录

第一部分 概念回顾与澄清

什么是（不是）DevOps，我们如何实现 DevOps

关于 DevOps 你必须知道的 11 件事

持续部署就意味着用户满意吗

DevOps ≠ Chef/Puppet

DevOps 与信息安全

数据库变更部署自动化秘诀

第二部分 实践经验

一些好的规则

携程首席架构师谈 DevOps：找到合适的人最重要

基于反馈控制实现可靠的自动扩展服务

梁定安：解密腾讯 SNG 云运维平台“织云”

Etsy 是如何做到每天 50 次以上部署的

跟 Monty Taylor 和 Jim Blair 聊 OpenStack 的持续集成与自动化测试

Amazon DynamoDB 在游戏开发中的应用

第一部分

概念回顾与澄清

什么是（不是）DevOps，如何实现 DevOps

作者 Abel Avram，译者 孙镜涛 发布于 2014 年 3 月 11 日

在本文中我们将会讨论一些人们对 DevOps 的误解，同时会介绍一个能够带来 DevOps 文化转变的流程。

在一篇题为“[不，你并不是一个 DevOps 工程师](#)”的博文中，Cloud Technology Partners 公司的副总裁兼首席架构师 Mike Kavis 谈论了一些与 DevOps 相关的错误想法。例如，他提到一些团队是如何误用术语 DevOps 的：

企业正在为 DevOps 苦恼。他们都想得到 DevOps，即使很多企业并不知道它到底是什么。在很多情况下，我会看到那些自称 DevOps 的基础设施团队在领导一个基层倡议。当我问他们开发团队在哪里的时候，他们通常会说“我们并没有邀请他们”，甚至更糟“我们并没有同他们交流”。

一些工程师将自己宣传为 DevOps，但是他们并不是，因为根据 Kavis 所说“DevOps 并不是一个人，一个角色或者一个头衔。即使你可以声称自己是一个 DevOps 工程师，但是这仅是你自己的看法，实际上你并不是。”如果 DevOps 不是一个角色，一个资格，一个头衔，那么它到底是什么呢？Kavis 的定义是：

DevOps 是一种文化转变，或者说是一个鼓励更好地交流和协作（即团队合作）以便于更快地构建可靠性更高、质量更好的软件的运动。

然后他详细描述说：

DevOps 是软件开发生命周期（SDLC）从瀑布式到敏捷再到精益的发展。DevOps 超越了敏捷，它的关注点是从 SDLC 中移除浪费。通常情况下，发现浪费或者瓶颈的形式包括：不一致的环境，人工的构建和部署流程，差的质量和测试实践，IT 部门之间缺少沟通和理解，频繁的中断和失败的协定以及那些需要珍贵的资源、花费重要的时间和金钱才能保持系统运行的全套问题。

我看到的另一个重复模式是：一个“DevOps”团队的第一步通常是决定他们是否应该使用 Chef 或者 Puppet（或者是 Salt、Ansible 等其他任何热门的东西）。他们甚至还没有定义自己打算解决的问题，即使他们手头的工具可以解决它们。这些团队通常会紧张地构建数千行脚本，但是这就产生

了一个问题：“我们的职责是编写 Chef 脚本还是通过质量更好、更稳定的产品更快地进入市场？”在大多数情况下，这些团队会将自己逼入绝境，大量的专有脚本实际上是增加了系统的浪费，而隐藏在 DevOps 运动之后的驱动力是从系统中移除浪费，这些团队并没有做到这一点。

如果说 DevOps 是一种打算让开发某个产品的多个团队之间能够更好的交流和协作的文化变革，那么下一个问题是该如何实现 DevOps，我们如何将这种文化引入到自己的公司中？

[DTO Solutions](#) 的共同创建者 Damon Edwards 在 2013 年的 DevOps Days Mountain View 上做了题为“[如何发起一个 DevOps 变革](#)”的主题演讲，他推荐通过一个三步走的过程将 DevOps 文化引入到某个组织中：

1. 弄清楚“为什么”

根据 Edwards 所说，首先非常清楚组织成员为什么会聚到一起，知道他们试图实现什么，清楚他们的目的是什么是非常重要的。为了找到这些问题的答案我们应该直接与组织中的这些人交流，询问他们为什么会出现在这里。组织的主要目标是我们实现 DevOps 文化的唯一原因，除此之外没有其他原因。

Edwards 认为 DevOps 仅仅是达到目标的一种手段，但是它本身并没有结束：“DevOps 并不是你的为什么，不是你合作伙伴的为什么，当然也不是你业务的为什么”。他甚至建议忘记 DevOps 团队，而是使用服务交付作为替代，因为“我们的职责是创造服务”。

2. 实现组织合作

按照 Edwards 介绍的过程，接下来需要做的是使整个组织合作，让所有人基于一组共享的条件和规则向一个共同的目标努力。当你能够把同一个目标指定给多人的时候，一个组织就实现了正确的合作，他们会选择同样的方式去实现各自的目标；他们对于同一个问题有同样的答案。这可能是“组织合作的终极梦想”。

为了完成这种合作，组织内部必须要有人描绘一个 **DevOps 景象**。这并不能通过教学过程实现，因为人们只会尝试着机械性地遵循这些步骤。我们需要的是教会大家一种思维方式。根据 Edwards 所说，这可以通过遵循下面的几个步骤实现：

1. 教导基本的概念，例如“单件流、批量工作、限制在制品的数量、拉式 vs 推式、持续交付”以及可以使用的工具等组织将会共享的一些通用词汇的概念。

2. 让所有人目标一致，通过：
 - a. 价值流程图——一个精益概念，它详细描述了一个组织内部发生的信息流和制品流，引导价值创造。
 - b. 时间线分析——试图发现时间花费在哪里，瓶颈在哪里。
 - c. 浪费分析——确定一个组织所产生的各种各样的浪费以便于尽可能地消除浪费。
3. **发展度量链**，它的意思是对价值交付链中的各个活动进行度量，发现各个活动相互之间的影响。
4. **针对基线识别项目/实验**。识别哪些项目或者活动偏离了基线，并且采取纠正措施。
5. **重复第 2 至 4 步**。这一步构成了持续改进流程。

为了实现这些想法，Edwards 建议了一个 3 天的计划：

- 第 1 天**——教导原则，提出一个方案进行研究，模式和反模式。
- 第 2 天**——分析组织当前的状态，提供问题识别技术和改进指标。
- 第 3 天**——讨论解决方案和工具链自动化原则，构建一个路线图。

3. 持续改进循环

这些循环的目的是通过制定计划、实现计划、测量输出和决定如何持续地改进流程。

在最近举行的 QCon London 2014 上 Edwards 做了题为 “[Dev ‘Programming’ Ops For DevOps Success](#)” 的分享，其中就讲到了这些原则，这个分享会发布到 InfoQ 上。

查看英文原文：[What Is \(Not\) DevOps, and How Do We Get There?](#)

查看原文：[什么是（不是）DevOps，我们如何实现 DevOps？](#)

关于 DevOps 你必须知道的 11 件事

作者 Gene Kim，译者 戚一品 发布于 2014 年 2 月 21 日

编辑注：本文来自技术出版商 [IT Revolution](#)，最初以白皮书的形式发布为 [*The Top 11 Things You Need to Know about DevOps*](#)。经本文作者 Gene Kim 和本文译者戚一品的授权与推荐，现将完整译文分享给 InfoQ 中文章的读者们参阅。

关于作者

Gene Kim 在多个角色上屡获殊荣：CTO、研究者和作家。他曾是 Tripwire 的创始人并担任了 13 年的 CTO。他写过两本书，其中包括 *The Visible Ops Handbook*，目前他正在编写 *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* 和 *DevOps Cookbook*。Gene 是 IT 运维的超级粉丝，痴迷于改进运维流程——在不影响当前 IT 生产环境的情况下，使得开发人员可以向生产交付更多可运行的功能，而非只是完成代码。他与多个顶级互联网公司合作过，致力于改进他们的发布流程，提高 IT 运维流程的完整性。2007 年，Computer World 将 Gene 列入了“40 岁以下的 40 个创新 IT 人士”的名单中，普度大学还给他颁发了杰出校友奖以表彰他在专业领域的成就和领导力。

目录

1. 什么是 DevOps
2. DevOps 与敏捷有什么不同
3. DevOps 与 ITIL 以及 ITSM 有什么不同
4. DevOps 与可视运维
5. DevOps 的基本原则
6. DevOps 模式的应用领域
7. DevOps 的价值
8. 信息安全和 QA 如何融入 DevOps 的工作流
9. 我最喜欢的 DevOps 模式一
10. 我最喜欢的 DevOps 模式二
11. 我最喜欢的 DevOps 模式三

1. 什么是 DevOps

术语“DevOps”通常指的是新兴的专业化运动，这种运动提倡开发和 IT 运维之间的高度协同，从而在完成高频率部署的同时，提高生产环境的可靠性、稳定性、弹性和安全性。

为什么是开发和 IT 运维？因为典型的价值流就是在业务（定义需求）和客户（交付价值）之间。

DevOps 运动的起源通常被放在 2009 年前后，伴随着许多运动的相辅相成和相互促进——效率研讨会运动，特别是由 John Allspaw 和 Paul Hammond 展示的开创性的“一天 10 次部署”，基础设施即代码“运动”（Mark Burgess 和 Luke Kanies），“敏捷基础设施运动”（Andrew Shafer），“敏捷系统管理”运动（Patrick DeBois），“精益创业”运动（Eric Ries），Jez Humble 的持续集成和发布运动，以及 Amazon 的“平台即服务运动”等这些运动的相辅相成和相互促进而发展起来的。

DevOps 的合著者 John Willis 写了一个非常好的帖子，参见：<http://itrevolution.com/the-convergence-of-devops/>。

2. DevOps 与敏捷有哪些不同

相对于瀑布开发模式，敏捷开发过程的一个基本原则就是以更快的频率交付最小化可用的软件。在敏捷的目标里，最明显的是在每个 Sprint 的迭代周期末尾，都具备可以交付的功能。

部署的高频率经常会导致部署堆积在 IT 运维的面前。StreamStep 公司的创始人，Clyde Logue 总结过一句话：“敏捷对于开发重新获得商业的信任是大有益处的，但是它无意于将 IT 运维拒之门外，DevOps 使得 IT 组织作为一个整体重新获得商业的信任。”

DevOps 和敏捷软件开发是相辅相成的，因为它拓展和完善了持续集成和发布流程，因此可以确保代码是生产上可用，并且确实能给客户带来价值。

DevOps 不仅仅创建了一个面向 IT 运维的工作流，当代码已经开发完成但是却无法被部署到生产上时，这些部署就会堆积在 IT 运维的面前，客户也将因而无法享受到任何价值，更糟糕的是，部署经常导致 IT 环境的中断和服务不可用。

DevOps 具有与生俱来的文化变革的基因组成，因为它革新了开发和 IT 运维之间的
工作流和传统的衡量标准。John Willis 和 Damon Edwards（两位都是 *DevOps Cookbook*
合著者）就这个话题写过很多文章：<http://itrevolution.com/devops-culture-part-1/>。

3. DevOps 与 ITIL 和 ITSM 有什么不同

尽管很多人视 DevOps 为 ITIL 和 ITSM 的颠覆，而我则有着不同的看法，在支撑 IT 运
维的业务流程方面，ITIL 和 ITSM 流程无疑还是最好的。实际上，他们描述了需要被
IT 运维支持的功能集合，这些功能集合足以支撑 DevOps 式的工作流。

敏捷和持续集成以及持续发布是开发的输出，这些输出同时作为 IT 运维的输入，为了
适用跟 DevOps 相关的快速部署的节奏，ITIL 流程的很多方面，特别是围绕着变更、
配置和发布流程方面，需要自动化。

DevOps 的目标不仅只是增加变更的频率，而且也支持在不中断和破坏当前服务的基础
上，确保功能部署成功，同时也可快速检测和修复缺陷。这引入了服务设计，事故
和问题管理方面的 ITIL 新准则。

4. DevOps 和可视运维如何搭配

2004 年，我与 Kevin Behr 以及 George Spafford 合著了 *The Visible Ops Handbook*，可视
运维是一个说明性的指南，该指南使得高性能 IT 运维能顺利实现“从优秀到卓越”的转
变，关键点之一是如何控制和减少计划外的工作。

在开发和 IT 运维之间，DevOps 不仅聚焦在创建快速和稳定的计划工作流，而且
DevOps 也有一个更全面的方法来系统的消除计划外工作，定义开发弹性准则，并负责
管理和减少技术债务。

5. DevOps 的基本原则

在 *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win* 和
DevOps Cookbook 的书里，我们描述了 DevOps 的支撑原则——“DevOps 三个基本
点”，所有的 DevOps 模式都可以源自这 3 个基本点。

第一个基本点强调整个系统的性能，而非将性能局限于特定的工作领域里，这个工作领域可以大到一个部门（例如开发和 IT 运维）或者小到一个个人贡献者（例如开发者，系统管理员等）。

重点是由 IT 推动的业务价值流，换句话说，它始于需求定义（比如被业务或 IT 部门定义），进行开发构建，又交给 IT 运维，最后价值以一种服务的形式交付给客户。

实践第一个基本点的结果——决不传递一个已知缺陷至下游，决不因小失大，总是致力于改进流程，执着于深刻理解系统需求（根据戴明的理论）。

第二个基本点是关于创建从右至左的反馈回路，几乎所有的流程改进计划的目标都是缩短和放大反馈回路，以便可以持续进行必要的修正。

应用第二个基本点的结果——包括理解和回应所有内部和外部客户，缩短和放大所有的反馈回路，必要时，非常容易的嵌入客户需要的知识。

第三个基本点是打造一种文化用来促进两件事情——持续不断的探索精神，勇担风险的精神以及从成功和失败中来学习的能力，同时也得谨记：重复和实践是融会贯通的前提。

这两件事情对我们来说同等重要，探索精神和勇担风险的精神可以确保我们持续改进，它甚至意味着我们可能到达了之前曾未到过的危险区域，因此这也迫使我们去学习，掌握并融会贯通那些技能，因而使得我们能够顺利离开危险区。

第三个基本点的结果——分配时间去改进每天的例行工作，培养一种奖励冒险精神的风气，同时主动引入故障到系统中，从而提高弹性。

6. DevOps 模式的应用领域

在 *DevOps Cookbook* 里，我们将 DevOps 模式分成 4 个领域。

领域一，将开发延伸至生产中——包括拓展持续集成和发布功能至生产，集成 QA 和信息安全至整个工作流，确保代码和环境可在生产中直接部署。

领域二，向开发中加入生产反馈——包括建立开发和 IT 运营事件的完整时间表用于帮助事件的解决，使得开发融入无指责的生产反思，尽可能使得开发可以自助服务，同时创建信息指示器用来表明本地的决策如何影响全局的目标。

领域三，开发嵌入到 IT 运维中——包括开发投入到整个生产问题处理链，分配开发资源用于生产问题管理，并协助退回技术债务，而且开发为 IT 运维提供交叉培训，增加 IT 运维处理问题的能力，从而降低升级问题的数量。

领域四，将 IT 运维嵌入至开发——包括嵌入和联络 IT 运维资源至开发，帮助开发创建为 IT 运维(部署，生产代码的管理等)使用的可重用的用户故事，定义一些可以被所有项目共用的非功能性需求。

7. DevOps 的价值

我相信企业在应用了 DevOps 之后可以得到 3 个业务优势：产品快速推向市场（比如，缩短开发周期时间和更高的部署频率），提高质量（比如，提高可用性，提高变更成功率，减少故障，等等）并提高组织的有效性（比如，将时间花在价值增加活动中，减少浪费，同时交付更多的价值至客户手中）。

产品快速推向市场

2007 年，在 IT 流程协会，在评测了超过 1500 个 IT 组织结构之后，我们得出结论：相比较于一般的组织，高效的 IT 组织的效率要高出 5 到 7 倍。变更要多出 14 倍，变更故障率只有前者的 $1/2$ ，第一次修复率要高出 4 倍，而且一级事故时间要短 10 倍。重复审计发现要少 4 倍，通过内部控制来检测漏洞方面要高出 5 倍左右，并且 8 倍于前者的项目到期日表现！

在我们的研究中，观察到的最高部署频率大约是每周 1000 次生产变更，变更成功率为 99.5%，我们认为这真得很快……

其中一个高绩效的特点是，最好正在继续变得更好。这绝对是发生在部署频率的领域内。在应用了 DevOps 实践的组织正表现出更快的快速部署和实施，而且相比于一般组织要快几个数量级。

埃森哲最近做了一个研究：互联网公司都在做什么？通过亚马逊的记录显示，他们在保持目前每周部署 1000 次的情况下，同时还能保证 99.999% 的成功率！

<http://www.youtube.com/watch?v=dxk8b9rSKOo>

[http://www.slideshare.net/slideshow/embed_code/9466635?startSlide=33\)](http://www.slideshare.net/slideshow/embed_code/9466635?startSlide=33)

维持高部署率（即，快速的迭代次数）的能力转化为业务价值的两种主要方式：

组织如何快速的把一个想法变成价值交付到客户手中（比如，Damon Edwards 和 John Willis 说得“概念到落地”）组织同时可以做多个尝试。

对我来说，如果我在一个每 9 个月才能部署一次的机构里，而我的竞争对手可以每天部署 10 次，那么无疑我将有着明显的竞争劣势。

高频率部署也实现了快速和持续不断的部署。Intuit 公司的创始人，Scott Cook 一直在组织的各个层面，不停的倡导“犀利创新文化”，我最喜欢的例子之一就记录在 <http://network.intuit.com/2011/04/20/leadership-in-the-agile-age/>。

“每一位员工应该能够做到快速，高速的交付……Dan Maurer 负责我们的消费者部门，包括 TurboTax 网站。当他接手的时候，我们一年只做几次部署，但是通过营造一个犀利的创新文化，在报税季节的 3 个月里，他们现在能做 165 次部署。商业价值？网站转化率高达 50%。员工价值？这帮家伙们真得喜爱它，因为可以将他们的想法很快交付到市场中。”

对我来说，Scott Cook 的故事最令人震惊的是，他们在繁忙的报税季节做所有这些部署！在旺季，大多数组织都会冻结任何变更（例如，从十月到一月，零售商可能经常有变更冻结）。但如果在旺季，若你能提高转换率，而你的竞争对手不能，那么这就是一个真正的竞争优势。

达到这个效果的前提就是，在不影响客户的基础上，可以快速的完成并部署很多小的变更。

减少 IT 浪费总量

Mike Orzen 和我很早就谈到了 IT 价值流中的巨大浪费，这些浪费是缘起于交付期限延长，不良的交接，计划外工作和返工。基于 Michael Krigsman 的一篇文章，在应用了 DevOps 的原则之后，可以挽回很多价值而非浪费。

我们计算过，如果能够减少一半的 IT 浪费量，然后把这些省下来的钱重新投资，若能得到 5 倍的投资回报，那么每年可以产生 30 万亿美元的价值。

这就是溜过我们指尖的巨大的价值和机会。占了全球 GDP 的 4.7%，甚至超越整个德国的经济产出。

我觉得这真的很重要，尤其是当我想到我的三个孩子将继承的这个世界，这些浪费对生产率，生活水平和经济繁荣的潜在影响正在不断增加，让我觉得不得不改变。

然而，还有一个更大的成本，在大部分的 IT 组织结构里，工作是吃力不讨好和令人沮丧的，人们觉得他们自己就像被困在一部不断重复的恐怖电影里，无法改变最终的结局。管理人员本应将 IT 管理的很好，但是他们放弃了这样的职责，直接导致开发，IT 运维与信息安全之间无休止的部门冲突，而审计师的出现只会让事情变得更糟。

长期来说，必然的结果就是进步迟缓。IT 专业人士的生活往往令人泄气和沮丧的，通常导致渗透在生活方方面面的无力感和高压状态。IT 专业人员面临着压力相关的健康问题、社交问题、宅在家里等问题，这样使得他们不但不健康，同时生活也很可能难以维继。

作为人，我们注定就是去贡献，感觉就好像我们正在积极发挥作用，与众不同。但是，往往当 IT 专业人员向他们的组织寻求帮助的时候，他们会得到回答：“你不明白”，更糟的是，他们甚至会得到“你不重要”这样的待遇。

8. 信息安全和 QA 如何融入 DevOps 工作流

DevOps 的高部署频率通常会给 QA 和信息安全带来很大的压力，考虑这样一种情形，开发每天部署 10 次，而信息安全通常需要 4 个月的时间来评估应用的安全。很显然，在代码开发和代码安全审计方面的速率一点都不匹配。

2011 年 Dropbox 故障就是一个著名的例子，其体现了未经充分测试的开发代码带来的风险有多大。因为这次事故，认证功能被关闭了 4 小时，从而导致未授权的用户可以访问所有存储的数据。

当然对 QA 和信息安全来说，也不全是坏消息，开发会通过持续集成和好的发布惯例（持续测试的文化）来维持高频率部署。换句话说，一旦代码被提交，自动测试便开始运行，而且一旦发现问题，必须马上解决，就像开发人员在检查还没编译的代码。

通过集成功能测试，集成测试和信息安全测试到开发的每天例行工作中，问题将会被更快发现，同时也会被更快解决。

同样，也有着越来越多的信息安全工具，比如 Gauntlet 和 Security Monkey，可以帮助我们在开发和上线的过程中测试安全对象，达到信息安全目标。

但是也有一个很重要的问题需要考虑，静态代码分析工具通常需要花费很长时间才能运行完，以数小时或天记。在这种情况下，信息安全就必须注明特定的有安全隐患的模块，比如加密，认证模块。只要这些模块变化，一轮全面的信息安全测试就运行，否则部署就可以继续，而不需要全覆盖信息安全测试。

需要特别提到的一点是，我们观察到，相较于标准的功能单元测试，DevOps 工作流依赖于检测和恢复更多一点。换句话说，当然开发以软件套件的方式交付的时候，那么部署变更和补丁就比较困难，同时 QA 也严重依赖代码测试来验证功能的正确性。另一方面，当软件以服务的形式交付，缺陷就可以被很快修复。而且 QA 也可以减少测试依赖，取而代之的更多依赖缺陷的生产监控，只要缺陷能被快速的修复。

代码故障恢复可借助于“功能标签”等手段，通过以配置的形式来启用或禁用某些代码功能，从而达到避免推出一个全功能部署，而只部署通过测试的功能至生产。

当功能不可用或性能出现下降等较坏的情况发生的时候，依赖于检测和恢复进行 QA 将会一种更好的选择。但是当面对损失保密性或数据和系统一致性的时候，我们就不能够依赖检测和恢复这种方法。取而代之的是，在部署之前，必须进行充分的测试，否则可能导致重大的安全事故。

9. 我最喜欢的 DevOps 模式一

通常，在软件开发项目中，开发都会用完所有计划中的时间用于开发功能。这样会导致无法充分解决 IT 运维的问题，于是他们就在定义，创建和测试数据库、操作系统、网络、虚拟化等代码依赖的方面直接抄捷径，以此节省时间。

所以这就是开发和 IT 运维以及次优结果之间的永恒的紧张关系的主要原因。后果很严重，比如不适当的定义和指定环境、无法重部署、代码和环境的不兼容，等等。

在这种模式下，我们会再开发过程的早期提出环境要求，并强制代码和环境必须被一起测试的策略，一旦使用敏捷开发方法，我们可以做到非常简洁和优雅。

按敏捷的要求，在每个迭代结束后，我们就会发布能运行且可被部署的代码，通常时间为两周。我们将修改敏捷迭代周期策略，不仅仅只交付能运行且可被部署的代码，同时在每个迭代周期的早期，还必须准备好环境用于部署这些代码。

由此，我们不再让 IT 运维负责创建生产环境的规格要求，取而代之的，建立一个自动化的环境创建流程，这种机制不仅仅只创建生产环境，同时也包括开发和 QA 环境。

通过使得环境早期即可用，甚至可能早于软件项目开始之前，开发和 QA 可以在统一和稳定的环境中运行和测试他们的代码，从而控制不同环境之间的差异。

此外，通过保持不同阶段（例如，开发、QA、集成测试、生产）尽可能小的差异，在生产部署之前，我们就能发现并修复代码和环境之间的互操作性问题。

理想情况下，我们建立的部署机制是完全自动化的。可以使用像 Shell 脚本、Puppet、Chef、Soaris Jumpstart、Redhat Kickstart、Debian Preseed 等很多工具来完成。

10. 我最喜欢的 DevOps 模式二

BrowserMob 前 CEO，Patrick Lightbody 曾经说过：“当我们在凌晨 2 点叫醒开发工程师来解决问题时，缺陷被修复的比以前更快了，这真是一个惊人的反馈回路。”

这是我最喜欢的引用之一。

它强调了问题的关键点，开发一般会在周五的 5 点提交代码，然后高高兴兴的回家，而 IT 运维则要花费一整个周末来收拾残局。更糟的是，缺陷和已知错误在生产上不断递归，迫使 IT 运维不停的救火，而根本原因从不被修复，造成这种现象的原因就是开发总是关注开发新功能。

第二种模式的一个重要要素就是缩短和放大反馈回路，使得开发更贴近客户体验（包括 IT 运维和最终用户）。

注意这里的对称性，模式一讨论的尽早让环境统一并可用即是将 IT 运维嵌入到开发，而模式二则为将开发嵌入到 IT 运维中。

我们将开发嵌入进 IT 运维的问题升级链中，可以将他们放在三级支持中，甚至使开发对整个代码的部署成功负责。要么回滚，要么修复缺陷，直到服务恢复。

我们的目标不是让开发取代 IT 运维，相反，就是想让开发看到他们工作和变更的下游变化，激励他们以 IT 运维的视角来更快的解决问题，从而达到一个全局的目标。

11. 我最喜欢的 DevOps 模式三

在开发和 IT 运维之间 DevOps 价值流中，另一个经常发生的问题就是不够规范。这方面的例子是，每个部署都带有其特殊性，因此也使得每次部署后的环境带有特殊性，一旦这样的事情发生，那么这个组织里就没有针对流程配置的控制。

在这种模式下，我们定义可重用且可跨多个项目的部署流程，敏捷方法里有个很简单的解决方案。就是将部署的活动变成一个用户故事。例如，我们为 IT 运维构建一个可重用的用户故事，叫做“部署到高可用环境”，这个用户故事定义了明确的构建环境的步骤、需要多长时间、需要哪些资源等。

那么这些信息可以被项目经理用来集成部署内容到项目计划中去。例如，如果我们知道在过去的 3 年时间里，“部署到高可用环境”用户故事被部署了 15 次，每次的平均部署时间为 3 天，加或减一天，那么我们对自己的部署计划将会非常有信心。

此外，我们还可以因部署活动被合适的集成到软件项目中而获得信心。

我们也得认识到有些特定的软件项目要求特别的环境，且其不被 IT 运维官方支持，我们可以允许这些被生产允许的环境中的例外，但是被 IT 运维部门外面的人提供支持的。

通过这种方法，我们即获得了环境标准化的好处（比如，减少生产差异，环境更一致，IT 运维的支持和维护能力增加），又能再允许的特殊情况下，提供业务需要的灵活性。

感谢[丁雪丰](#)对本文的审校。

查看原文：[关于 DevOps 你必须知道的 11 件事](#)

持续部署就意味着用户满意吗

作者 Savita Pahuja，译者 曹知渊 发布于 2014 年 9 月 29 日

持续部署（continuous deployment）使企业能通过自动化的构建、测试和部署循环来快速交付高质量的软件。它使投资更容易得到回报，产品团队更早地得到用户反馈，也简化了部署流程。但从商业的角度看，持续部署也那么好吗？

Steve Blank 是[斯坦福大学](#)的咨询副教授，他在他最近的[博文中](#)提到，从消费者的角度来讲，持续部署可能意味着不满意。

虽然从工程角度来说，持续部署确实是一种更好的开发流程，但它对一家公司的商业模式和客户的期望都有意义深远的影响。

他对比了发布周期较长的瀑布交付流程（waterfall delivery process）和更频繁的持续交付流程。以前，公司如果使用瀑布模型，产品上市需要几个版本周期。公司计算年收入额的时间点，都是围绕瀑布模型的软件发布周期来设置的。跟瀑布式开发相反，敏捷开发模式在一个不断更新的基础版本上，增量和持续地进行交付，这也会影响公司年收入额的计算模式。公司可以通过云来交付产品的改进版本，这样用户就能不断地用上更好的产品。

Steve 举了 Adobe 的例子，他们现在把整个产品线迁移到了云端，称之为 Adobe Creative Cloud。用户不再需要为新产品付钱，而是支付每年的订阅费用。这种做法使公司的年收入更趋向于稳定，但是从消费者的角度来讲，却是不好的。

他描述了 Adobe 在运用持续部署过程中遇到的问题。

虽然持续部署让 Adobe 从高端用户身上不断获得稳定的年收入，但是他们也制造了两个问题。首先，不是所有客户都相信 Adobe 新的订阅式商业模式能给自己带来好处。如果客户停止支付每月的订阅费用，那他们不但会失去工作所依赖的 Adobe Creative Suite 软件（Photoshop、Illustrator 等），同时还可能无法访问已经完成的作品。

其次，Adobe 这种定位过高的策略无意中伤害了要求比较低的学生、小公司和轻度个人用户，把他们送到了优秀竞争对手的怀抱，比如用 Pixelmator、Acorn、GIMP 替代 PhotoShop，用 ArtBoard 替代 Illustrator。

抛弃低价值客户、增加年收入和短期利润的结果就是，Adobe 培养了未来的竞争对手。

Steve 也提到了[特斯拉](#)的例子。特斯拉在不断地推出更好的车型。

在未经消费者许可的情况下，特斯拉单方面删除了用户已经付费的功能，对于有云端功能的商品来说，麻烦就此开始。其次，特斯拉取消了新车的年度发布机制（model years），他们对持续开发软件和硬件的激进推广，使现在的用户对他们的期望出奇地高。有些用户觉得他们理应获得所有刚刚投产的硬件新功能，即便这些新功能（比如更快地充电、全新的泊车传感器）在他们买车的时候还不存在——即便他们的车是后向不兼容的。

新车的年度发布机制，使用户的期盼有一个明确的时间边界。缺少了这种边界，就会使一些客户失望。

查看英文原文：[Does Continuous Deployment Depict Customer Disatisfaction](#)

查看原文：[持续部署就意味着用户满意吗？](#)

DevOps ≠ Chef/Puppet

作者 阮志敏 发布于 2014 年 2 月 17 日

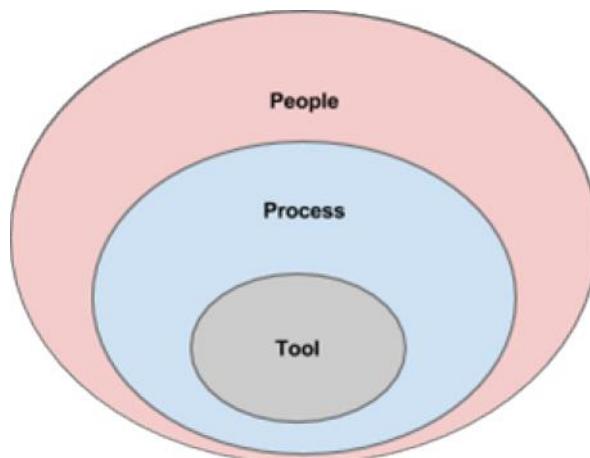
DevOps 是一个热词，但是毫无疑问，也是未来的趋势（注 1）。高效率的 IT 组织常常贴着 DevOps 的标签，是人们讨论的焦点和学习的对象。2009 年时，人人都在讨论如何像 Flickr 一样一天内完成十几次的部署（注 2）。今天，人人都谈论如何像 Netflix/Pinterest 一样基于云基础设施构建大规模、高可用、可伸缩的服务（注 3）。

如何才能实现 DevOps 呢？很多人会不假思索地告诉你，使用 Chef/Puppet，你就能实现 DevOps。于是，DevOps 变成了一个简单的问题，选择 Chef 还是 Puppet。这并不奇怪，在开源软件盛行的今天，各种软件声称自己是 DevOps 工具，而人们通常不会去思考一项新技术或者新思路背后的缘由，人们需要的是“银弹”。

如同 Agile，把 DevOps 等同于工具层面的 Chef/Puppet，是错误的，会严重束缚人们的思维。在国内 Cloud Native 应用开发时代即将开启的今天，充分认识 DevOps 是很有必要的（注 4）。

(一) DEVOPS 不仅仅是工具

DevOps 是 Agile 的延伸，Ailge 依靠 Dev & Biz 部门紧密协作，通过增量交付的方式来解决需求多变的难题。DevOps 则进一步延伸到 IT 运维，通过 Dev & Ops 的紧密协作提高软件交付的质量和频率。人的重要性大于流程，流程的重要性大于工具。这个结论适应于 Agile，也同样适用于 DevOps。工具带来的影响是短期的和片面的，流程和人所产生的影响是长期的和全面的。



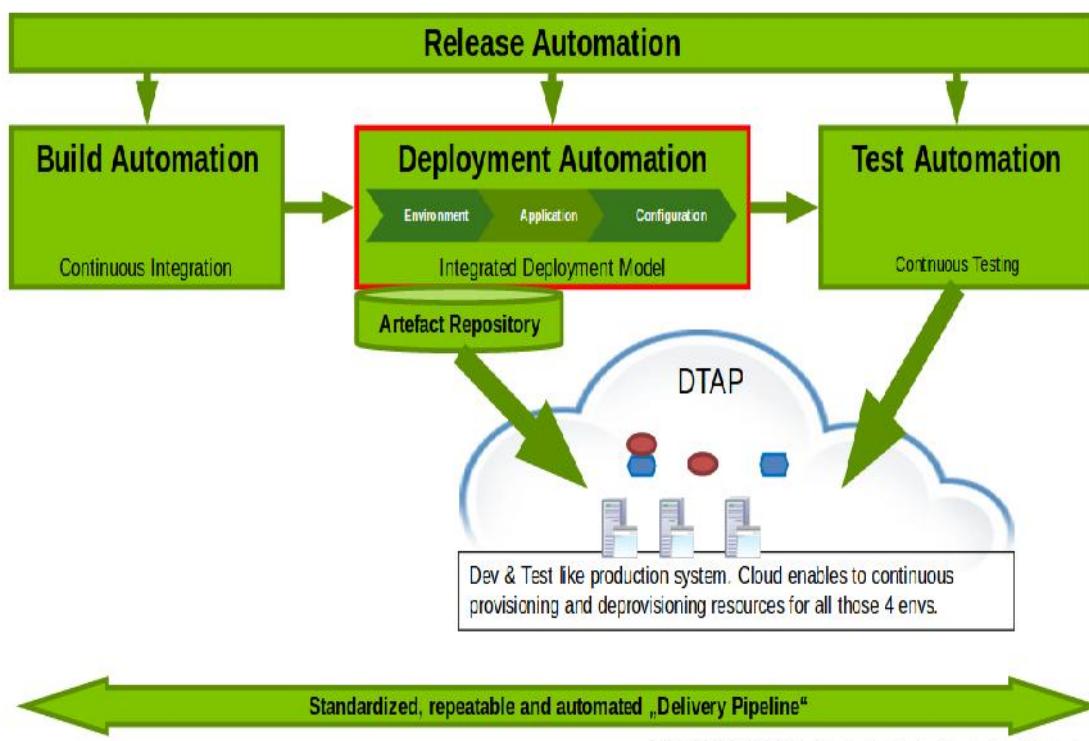
事实上，大部分人都知道这个道理，只是在潜意识中仍然把 DevOps 当成 Chef/Puppet 等工具。建设 DevOps 的正确步骤应该是充分理解 DevOps 的原则，认真分析自身需求和条件，选择正确的方法和流程，最后才是选择或构建适当的工具。Learn By Example 仍然是学习和建设 DevOps 的重要途径。在今后的各种会议上，分享 DevOps 经验会越来越多。我们应该不仅仅关注讲演中提到的工具，更要多关注流程、组织结构和文化方面的分享。

DevOps 不仅仅是工具，即便是工具，其也是建设 DevOps 所需工具链中的可选工具。

(二) Chef/Puppet 只是 DevOps 工具链中的可选工具

DevOps 目的是打造标准化的、可重复的、完全自动化的 Delivery Pipeline，其范围涵盖需求，设计，开发，构建，部署，测试和发布。除了需求、设计和开发外，其他的四个步骤都是可以自动化的。自动化是提高可测试性，一致性，稳定性和交付频率的核心。

下图来自 IBM [Agile, ITIL, Cloud How DevOps brings it all together](#)。该图非常清晰地解释 DevOps 如何实现交付的自动化（注 5）。



原图来自 IBM Agile, ITIL, Cloud How DevOps brings it all together (from slideshare)

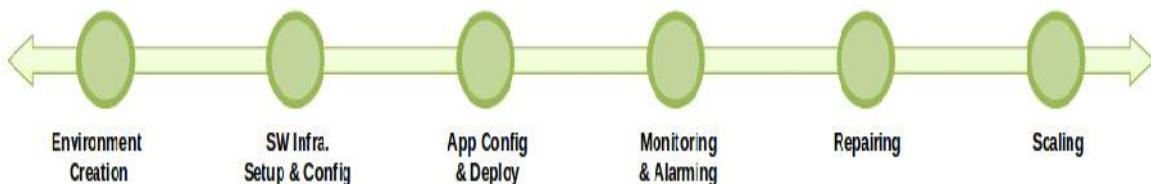
图中 DevOps 流程所需要的工具和环境有：

1. 源代码版本控制工具：比如 SVN, Git 等。
2. 持续集成工具：比如 Jenkins, Bambo 等。
3. Artifact 存储仓库：持续集成构建后的 artifact 都统一放在一个仓库中，比如 Nexus/Artifactory, 当然也可以是 FTP, S3 等。
4. 配置和部署工具：Chef/Puppet/CFEngine, Fabric/ControlTier，也包括新兴的 Docker 等。
5. Cloud Provision 工具：在云环境下，由于任何 IT Infra 资源都以编程接口提供，意味着 Full-Stack Automation (from “bare-metal to running business services”) 成为了可能。Cloud provision 工具可以自己通过 API 构建(如 Netflix Asgard)，也可以直接使用 IaaS 服务商提供的扩展服务如 AWS CloudFormation & Opsworks，也可以使用第三方工具如 Ringscale/Scalr 等。相当一部分 Cloud Provision 本身也集成了 Chef/Puppet 来实现后续的部署和配置。
6. 测试工具：除了传统的测试工具外，还需要模拟 Infra 灾难、验证系统健壮性的工具，如 Netflix 的 Chao Monkey。
7. 发布工具：一般情况下，人们需要拥有 DTAP 四个环境，即开发环境、测试环境、Staging 环境和生产环境。每种环境的作用，部署方式和代码版本等是不一样。比如开发环境是持续部署的，测试环境是定期如每天晚上自动部署，Staging 和生产环境是手动触发的。
8. 云基础设施：包括 AWS/Azure 等公有云，Cloudstack/Openstack 等私有云。

因此，我们看到，Chef/Puppet 只是实现 DevOps 工具链的可选工具，可以用来实现配置和部署自动化。但是仅靠 Chef/Puppet 本身无法实现 Full-Stack 部署自动化。

（三）仅靠 Chef/Puppet 本身无法实现 Full-Stack 部署自动化

如果要实现 Full-stack Automation，那么就必须实现环境创建自动化，软件安装和配置自动化，应用部署和配置自动化，监控和告警自动化，故障检测和恢复自动化，扩展自动化，如下图所示（注 6）。



1. 环境创建：创建 VMs、网络、存储、负载均衡，协调不同角色 VMs 的创建过程和配置。
2. 软件安装和配置：操作系统配置，比如创建用户、组，设置 ulimit 参数等；基础软件安装和配置，比如 mysql/nginx。这些软件的特点是变动不频繁。对于 Chef/Puppet 来说，这个步骤的自动化是其最擅长的。它们都提供大量现成的 Recipes，并抽象各种异构系统之间的差异。
3. 应用部署和配置：部署应用代码，比如 war 包、db 脚本、php/rails 代码等。这部分的变动是频繁的。对于 Chef/Puppet 来说，其是可以做这个工作的，但是很多人认为用 Fabric/Glu 等更为合适。另外，对于复杂的系统来说，如果保证升级期间系统的可用性，系统的各个应用组件需尽量是无状态和松耦合的。如果系统的组件之间是有依赖的，那么升级期间必须动态地协调（Orchestration）、控制好相关组件的行为。
4. 监控和告警：包括 OS 级别和应用级别的可用性和性能监控。如果发现异常，需要能够自动发出告警信息。
5. 健康检测和恢复：为了应付云基础设施故障，系统需要 Design By Failure. 在异常发生时，系统可以发现并进行自动处理。
6. 自动伸缩：一般情况下，业务存在高峰期和低估期。为了节省成本，系统应该是可以自动伸缩的。

对于上述的每一个步骤，看似都存在现成的工具可以用来实现自动化。但是，实现 Full-Stack 部署自动化从来就不是一件容易的事情，绝非简单通过选择、拼凑相关工具即可实现。Autodesk 中国研发中心最近在 InfoQ 上分享了他们[基于 AWS 的自动化部署实践](#)。文章详细阐述了业务目标，设计目标和限制，和最终的实现方案，是一个非常好的案例。在这里，我们再来分析一下两种差异较大的实现方式。

（1）基于 PaaS 的实现方式（以 Cloud Foundry 为例）

环境创建	用户不需要创建物理资源环境，Cloud Foundry 会自动创建并分配资源给各个租户，用户无法控制底层 OS 等
软件安装和配置	用户不需要软件安装。Cloud Foundry 已经安装好相关软件，只是支持的类型和版本有限
应用部署和配置	Cloud Foundry 提供接口，用户调用接口进行应用部署和配置。应用类型必须是 Cloud Foundry 支持的，只能进行有限的配置，比如 Tomcat 的配置参数等
监控和告警	Cloud Foundry 提供监控服务，但是 Metric 类型有限，无法支持自定义 Metric
健康检测和恢复	Cloud Foundry 提供 Container 级别的健康检测和恢复
自动伸缩	基于 Cloud Foundry 提供的 monitoring 接口和应用控制接口，可以实现 instance 级别的自动伸缩

貌似这种方式是最完美的方案，Cloud Foundry 基本替你实现了上述的所有自动化步骤，应用开发人员只需专注于应用逻辑本身的开发。然而，Cloud Foundry 也限制了用户的选择权和控制权，特别是大型的、复杂的、分布式系统，开发人员需要的是 Full-Stack 可控制性。

(2) Netflix 的实现方式

环境创建	通过自己研发的 Asgard 管理和部署工具实现
软件安装和配置	基础软件和配置都打包成 AMI，基于 Netflix 自己的打包工具 Aminator
应用部署和配置	同上，应用代码和配置也打包进 AMI（注 7）
监控和告警	Leverage AWS Cloudwatch, 同时也将通过自己开发的 Servo Lib 将 custom metric 发送至 Cloudwatch.
健康检测和恢复	Leverage Atoscaling group, 健壮性测试有 Netflix 自己开发的 Chaos Monkey 工具
自动伸缩	Leverage AWS AutoScaling Group

Netflix 除了 Leverage AWS 的 CloudWatch/Auto Scaling Group/ELB 等服务外，自己也开发了一系列工具完成了 Full-Stack 部署自动化。这些工具通过 [Asgard](#) 这个可视化云管理和部署控制台集成起来。另外，Deployable image 这种部署模式虽可简化部署并确保一致性，但是一部分复杂性转移到了应用层面（注 7）。系统的各个组件是无状态和松耦合，同时还需要 [Eureka](#) 这种服务来实现中间层的负载和 failover。

在上述两个案例中，我们都看不到 Chef/Puppet 的影子。即便是 Cloud Foundry 自身的部署工具 Bosh，但也不是基于 Chef/Puppet。所以，尽管 Chef/Puppet 非常流行，并且有很多成功案例，但是我们决不能把 DevOps = Chef/Puppet。它们只是建设 DevOps 所需工具链中的可选工具，仅此而已。

参考文档

注 1: [What's DevOps?](#)

注 2: [10+ Deploys Per Day: Dev and Ops Cooperation at Flickr](#)

注 3: [Ops, DevOps and PaaS \(NoOps\) at Netflix](#)

注 4: [对国内云计算三个现象的思考](#)

注 5: [Agile, ITIL, Cloud How DevOps brings it all together](#)

注 6: [“It's the App, Stupid!” on Orchestration, DevOps Automation and What's in Between](#)

注 7: [Netflix deployable image](#)

关于作者

阮志敏，AWS 认证架构师，目前在三星中国研究院从事 PaaS 相关工作。

感谢[丁雪丰](#)对本文的审校。

查看原文: [DevOps_Chef/Puppet](#)

DevOps 与信息安全

作者 崔康 发布于 2014 年 5 月 3 日

DevOps 在国内社区逐渐推广，其关注于改善开发与运维团队之间的沟通与协作。最近，来自于 Yellow Spider 公司的 COO Leslie Sachs 和顾问 Bob Aiello [撰文](#) 分享了如何使用 DevOps 最佳实践来使您的信息安全更加健壮和有效。

首先，作者回答了为何信息安全是 DevOps 的关键组成部分之一：

DevOps 在确保开发人员与运维人员能一起工作并且更有效率方面非常成功。通过 DevOps，运维团队可以获得他们所需要用于了解如何建立有效和可靠应用程序构建、打包和部署过程方面的信息。而信息安全组也同样有许多与运维团队一样的需求。此外，InfoSec 还需要获得他们所需要用于确保整个系统是安全和可靠方面的信息。正如 DevOps 帮助开发和运维团队能更有效地一起工作一样，DevOps 也可以帮助开发和信息安全团队更有效地一起工作。在 DevOps 中，持续部署已经成为 DevOps 的一个关键实践，并且关注于通过自动化的构建、打包和部署来自动化部署流水线。通过提供一个平台可以在开发生命周期里尽早访问和定位安全问题，信息安全团队也同样能够从部署流水线上得到显著的收益。只要一旦有风险评估被介入，有效的安全保障就应当永远与之同步进行。

需要强调的是，DevOps 可以帮助定位安全风险。作者指出，作为软件或系统开发工作的一部分，风险需要被理解和定位。安全保障不能仅仅在开发过程的末尾才加入进来。系统需要在开发生命周期的最开始，就将安全保障与设计和开发一道同时得到关注。

- DevOps 提供了必需的构造来帮助定位众多安全风险，这是创建任何复杂技术系统的内在要求。
- 安全漏洞往往是各种事件的直接后果。例如，在 C/C++ 系统中进行不恰当的编码实践就可能导致缓冲溢出条件有机会被恶意攻击者用于实施越级程序权限。缓冲溢出攻击经常被攻击者所利用，来获得系统的控制权，甚至可以很有效地获得 root 权限。

- 运行时事件的发生也可能导致不恰当的安全控制，例如发生在不同组件之间的身份验证与授权。一个常见的安全问题根源就是来自于由于某次部署所应用的不正确的访问权限。
- 另外一个安全问题领域是确保所部署的是正确的代码。在部署过程中所带来的错误有可能会暴露给恶意攻击者。
- 在不同接口之间的配置问题经常会暴露给攻击者，以被其用于尝试侵入系统。一旦系统缺乏防范措施，不恰当的安全控制问题就有可能导致非授权的变更变得非常难以识别，而且难于执行鉴定证明来查清到底有哪些变更由于错误或是由于恶意目的所造成的。

作者指出，通过有效的源代码管理来构建安全的系统，软件质量需要从最开始构建。

DevOps 及部署流水线帮助有效地创建和提供测试环境，以用于评估和测试在组件间接口上的安全漏洞。通过以及时到位的方式提供一个健壮的测试环境可以增强安全性，通过提供一个自动化的测试平台可以用于识别需要被定位的安全问题。一旦代码中的问题被找到，它们就能被定位作为缺陷或变更请求，从而在它们的整个生命周期中被追踪到，以确保已被识别的风险可以被定位。

在“开发、运维和信息安全团队之间的协作”方面，作者强调，信息安全团队通常非常缺乏足够能理解复杂系统常见的内在自有安全漏洞方面的技术专家。

正如 DevOps 能改善开发与运维之间的沟通，DevOps 同样能增强信息安全团队的能力，它允许信息安全团队完全理解整个应用以及它是如何构建、打包和部署的。这一知识可以帮助 InfoSec 维护一个相关的和有效的关注。通过理解系统的基础架构，InfoSec 同样可以帮助理解何时安全会被突破以及应该采取什么样的措施应对该安全漏洞，尤其在基础设施自身缺乏安全免疫力的前提下。在许多案例中，这对使用自动化的过程重建服务器来说非常关键。

在“自动化应用程序的构建、打包和部署”方面，作者指出，可以以多种方式来创建自动化过程来构建、打包和部署代码来支持敏捷迭代开发。

自动化构建过程是实现持续集成（continuous integration）和持续交付（continuous delivery）的一个前提条件。构建过程应当自动化地嵌入不可改变的版本 ID 到每一个由构建过程所使用或所创建的配置项

(configuration item, CI) 中。这也是加密哈希值 (hash) 应当被创建的地方。冲刺里程碑发布版本 (Sprint milestone release) 可以被用于测试和验证在项目启动阶段 (inception) 尚未被完全理解的需求。更加重要的是，关于如何构建每一个组件的技术细节可以被隐含性地文档记录，并可以被包括了信息安全方面的感兴趣的利害相关者所审阅。在代码中创建变体可以建立应用程序的测试，包括机制化代码来建立与安全相关的测试。

当然，要想实现信息安全，执行入侵测试是必不可少的步骤。

部署流水线提供了一个有效的框架来创建所必须的测试环境来执行入侵测试。通过 DevOps，入侵测试可以在整个软件和系统生命周期中贯彻进行。通过改善开发和信息安全团队之间的沟通，DevOps 及 InfoSec 都可以在通常专门为入侵测试所留出的短小时时间片断里设计和执行更加有效的入侵测试。部署流水线同样可以使信息安全人员可以获得对于有效评估安全漏洞来说非常关键的技术信息。

最后，作者指出，必须要提供安全可信的应用程序基础。

为了确保可信的应用程序基础，应当使用可以安全识别准确二进制代码以及所有其他配置项（包括 XML 及属性配置文件）的过程来构建应用程序代码，以便您可以毫无疑问地证明正确的代码确实被部署和确认没有非授权的变更发生。这些过程通常会使用在应用程序自动化构建和打包过程中就首先创建的加密哈希值。一旦代码被部署，哈希值就可以被重新计算来验证所有的代码是否被正确部署。

查看原文：[DevOps 与信息安全](#)

AWS Summit

AWS 技术峰会 · 北京 2014

12月12日 北京国际饭店会议中心

沃纳 · 威格尔，亚马逊公司CTO



沃纳 · 威格尔博士是亚马逊公司全球副总裁兼CTO，负责推动亚马逊公司的技术愿景，以及全球范围内基于亚马逊的用户需求的持续创新。

加入亚马逊之前，沃纳 · 威格尔是康奈尔大学的科学家，是多个研究项目的首席研究员，专注于企业关键任务计算系统的可扩展性及稳健性研究。他还曾担任多个企业的副总裁及CTO，实现科研成果的产业化。

沃纳 · 威格尔拥有阿姆斯特丹自由大学的博士学位，曾在多份期刊及会议上发表超过80篇主要针对企业计算系统的分布式技术处理的文章。

[了解更多内容>>](#)



数据库变更部署自动化秘诀

作者 Yaniv Yehuda，译者 马德奎 发布于 2014 年 3 月 28 日

瞬息万变的世界：敏捷 & DevOps

由于业务需求是变更最主要的驱动者，少做一些，但做得更好，交付更快，这是领先的企业和成功的企业与其他企业的不同之处。

当竞争对手交付了相关功能，速度比你快，质量比你好，那么你最终会丧失市场份额。用投资于销售和市场营销活动的方式弥补产品的不足，其代价会很高，而且可能不可靠，而且你可能会发现，客户转向了质量卓越的产品。

这正是“敏捷开发”产生的原因：需要更快地采取行动，应对不断变化的需求（因为我们的目标市场和竞争对手永远不会静止不动），可信赖的最佳品质，经常资源不足。敏捷就是源于科技公司和 IT 部门的期望。

自然地，下一步是找到一个将敏捷应用于生产的方法：连接开发和运维。这就产生了 DevOps。

运维的主要目标是保证应用程序的稳定和健康，而开发的主要目标是不断地创新，并提供满足业务和客户需求的应用程序。理解这两点是 DevOps 发展的关键。既然变更 是稳定最大的敌人这点没有疑问，那么理解和调和这种冲突应该是 DevOps 的主要目标。

为了有效地掌握敏捷冲刺部署以及实施 DevOps，人们需要能够实现部署自动化。否则，部署和发布就需要手动操作步骤和过程，而这些操作并不总是能够准确地重复，容易出现人为错误，并且无法进行高频率地处理。

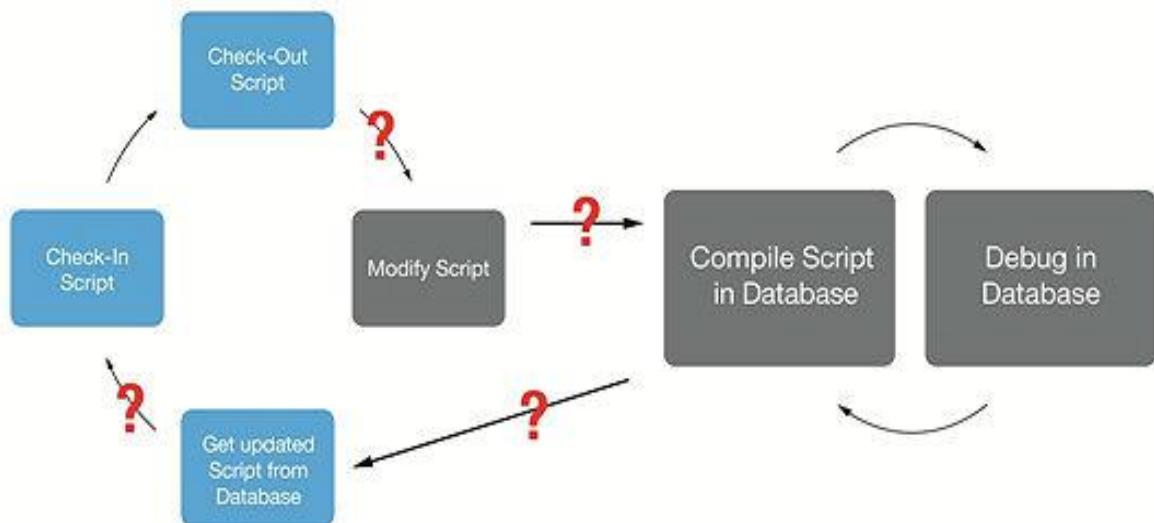
处理数据库部署并不简单；不像其它软件组件和代码或者已编译的代码，数据库不是一个文件集合。那不是可以从开发环境复制到测试环境然后复制到生产环境的东西，因为数据库是个容器，里面装了我们最有价值且必须保存的资产——业务数据。它保存了所有应有程序内容、客户事务处理等。为了促成数据库变更，需要开发迁移代码——用于处理数据库模式结构（表结构）的脚本、数据库代码（过程、函数等）和应用程序使用的内容（元数据表、查找内容表或者参数表）。

数据库变更部署流程的挑战

应对数据库挑战的一个方法是，迫使数据库变更遵循一般过程：创建数据库对象的脚本，然后存储在传统的版本控制系统中。

那造成了其它的挑战，包括：

1. 由于是两个单独的系统，版本控制系统中的脚本与它们所代表的数据库对象之间没有关联。数据库代码的编写和测试都是在数据库端完成，脱离了任何最佳编码实践（检入、检出、标记等），容易出现“旧时代”的所有问题，比如：
 - 代码覆盖在数据库中很常见，因为没有什么能够防止它发生。
 - 在数据库上运行代码之前，要先从版本控制系统中取得脚本，还要防止工作在错误的版本上；但是没有强制措施可以保证这一点。
 - 脚本在版本控制系统中的路径可能会出错，因为开发人员靠记忆做这件事。
 - 流程之外的更新会被忽视等。
2. 脚本是手动编写的，容易出现认为错误、语法错误等。
3. 为了拥有以后可能需要的一切，开发人员竟然不得不为每个对象保存两到三个脚本：对象的实际代码、升级脚本和回滚脚本。
4. 脚本很难进行整体测试。一个人单独更新了一个对象，而另一个人单独更新了另一个对象，如果脚本需要以特定的顺序运行，那么以任意顺序运行通常就会由于错误的依赖关系而产生错误。
5. 如果一个脚本被开发成代表整个更新而不是单个变更的单独脚本，那么它可以处理依赖关系，但处理项目范围的变更就要困难得多了。那是一个很长的命令列表。
6. 除非非常有经验，否则这些脚本中会缺少从编写到运行这段时间内发生在目标环境里的变更；可能会覆盖生产环境中的热补丁，或者与另一个团队并行操作。
7. 内容变更管理非常困难。实际上，版本控制系统不适合元数据或者查找内容。在大部分情况下，根本就没有对它们进行管理。

Version Control ProcessDB Development Process

最近十年，出现了另一种理念，就是使用工具处理环境间迁移代码的生成。这种操作方式被贴上了“**比较&同步**”的标签，是说用一种机械的比较方法检查源环境中的数据库对象，并将它与目标环境进行比较，如果发现差异，就会自动生成一个仿照源对象更改目标对象的脚本。在一段时间内，这似乎是个好方法，直到其缺陷变得越来越明显。

数据库的比较常常是在选定的检查点上执行的，通常是在开发周期结束之后，部署之前：

1. 比较工具并不清楚发生在它运行之前的变更，或者任何发生在目标环境中的变更。没有版本控制，我们就没有变更信息，只知道特定时间点的差异。
2. 对象脚本保存在传统的版本控制解决方案中，而部署使用的是比较&同步工具，你可以放开了想象，这两者之间是如何的无法协同。一个系统对另一个系统一无所知。
3. 手动检查和关于每个变更的详细知识必须是部署流程的一部分。否则，下面这样的不幸就会发生，用过时的代码或者来自完全从事另外一项工作的团队的结构覆盖了生产环境中恰当的、最新的更新（比如由一个开发团队提供的热补丁）。
4. 团队之间的代码合并完全无法实现。如果需要合并，就需要手动编写代码。

手动流程使用“比较&同步”工具还是可以的，但需要熟练和耐心。对数据库而言，试图基于这些工具自动化部署流程包含了相当大的风险。

DBA 深知数据库部署的陷阱，同时作为最不合时宜的故障的受害人，往往回避基于上述流程的自动化，因为他们对自动脚本生成器的准确性没有信心，或者对保证预先准备好的、手工生成的脚本在开发完成后任何时候都依然正确的能力没有信心。为了避免冲突，他们常常把事情掌握在自己手中。小心翼翼地检查变更，手动创建尽可能贴合部署活动的变更脚本，相比之下，这种做法似乎不那么令人沮丧。

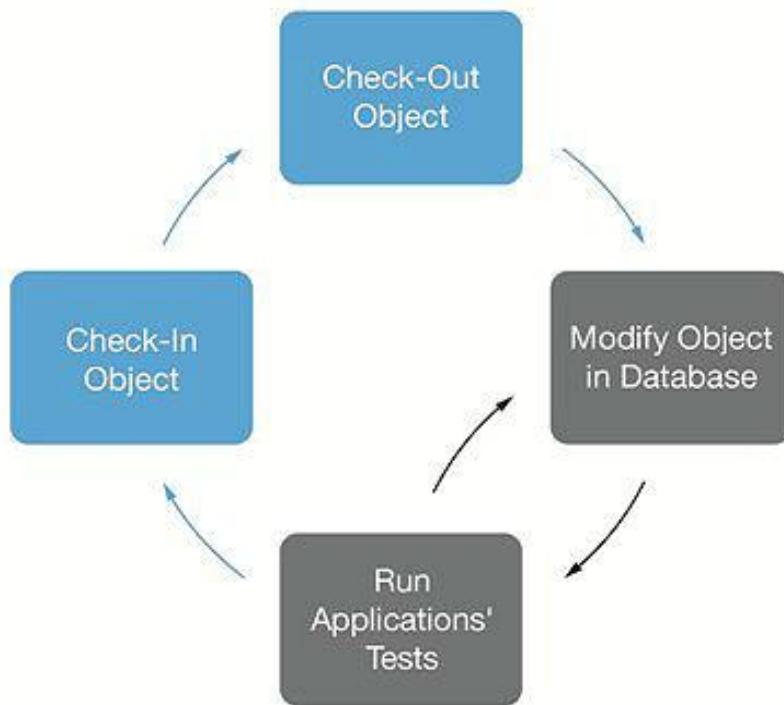
安全的数据库部署自动化

通过将数据库对象变更脚本写进传统的版本控制系统中实现自动化的做法有局限性、不灵活、与数据库本身脱节，而且可能不合标准，并容易因为脚本冲突丢失目标环境的更新。使用“比较&同步”工具实现自动化则是一件有风险的事。这两种理念没有结合在一起，一个不知道另一个，必须找出一种更好的解决方案。

为了将数据库恰当地自动化，必须考虑下列因素：

1. 在执行一个工作流程时，有恰当的**数据库版本控制系统**，应对数据库独有的挑战。这可以防止任何流程外的变更、代码覆盖、或者不完整的更新。
2. 利用已经证明了的版本控制**最佳实践**，获得关于谁在什么时间因为什么做了什么的完整信息。确保变更的完美记录是以后部署的基础。

Development & Version Control Process



3. 与**基于任务的开发**相协调，使每个版本控制下的变更与一个变更请求或者一个问题单相关联。这使得基于任务的部署、部分部署和最后时刻的范围变更可以在代码和数据库之间协调。
4. 确保配置管理&一致性，这样，每个开发环境、分支、主干、沙箱以及测试或生产环境都遵循相同的结构、一致的状态；或者对任何偏差和差异做详细说明。
5. 处理部署流程自动化的脚本化接口能够在每次执行时提供可重复的结果。如果不得不使用用户界面一次又一次地做同样的工作，那么即使是最先进的解决方案也会变得繁琐。
6. 提供可靠的部署脚本，能够解决冲突、合并代码、以及与其他团队交叉更新；同时还能忽略错误的代码覆盖，以及完全集成到版本控制库。

Simple Compare & Sync

Source vs. Target		Action
=		No Action
	≠	?

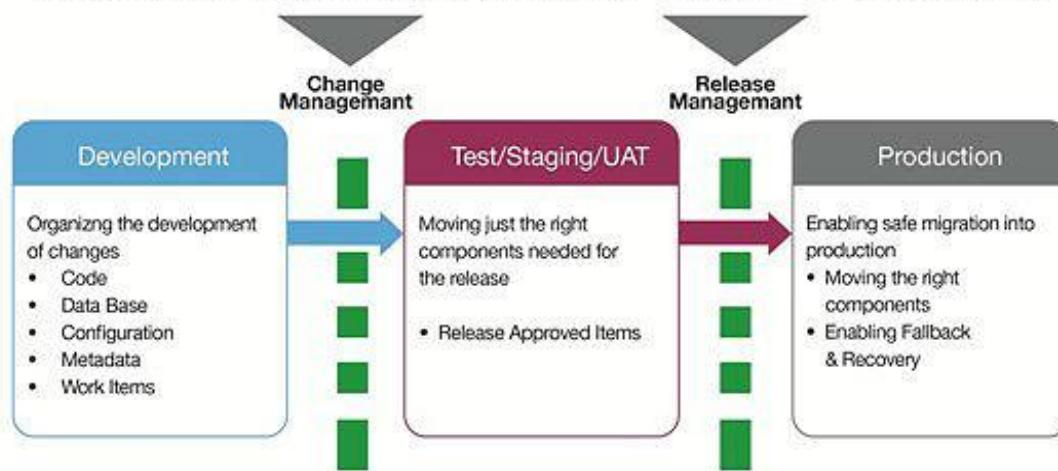
Baseline Aware Deployment

Source vs. Baseline	Target vs. Baseline	Action
=	=	No Action
≠	=	Override
=	≠	Ignore
≠	≠	Merge

7. 动态提供自动生成的开发脚本，处理部署项目范围内的任意组合，从多模式的大型更新，到基于单任务的变更及其所依赖的对象。
8. 在变更部署前后，利用“标签”（标记数据库结构快照和相关内容）作为安全网，这样，随时都可以简单快速地回滚。
9. 可以完全集成到其它系统（ALM、变更管理/问题单、构建服务器以及发布管理器）。

实现一种能够应对这些挑战的解决方案，将使企业能够实行恰当的数据库自动化。数据库自动化很容易与变更和发布流程的其余部分集成，进而实现完整的端到端的自动化。

Challenges of Development & Release to Operation



总结

数据库对自动化提出了一个真正的挑战。将数据库对象变更脚本写进传统的版本控制系统或者使用“比较&同步”工具，对于自动化而言，这两种理念要么效率不高，要么是件纯冒险的事，因为它们彼此之间互不知晓。要以数据库 DevOps 的形式实现一种更好的解决方案。

数据库 DevOps 应该遵循已经证明了的变更管理最佳实践，在数据库上强制实行单一的变更流程，能够解决部署冲突，降低代码覆盖、交叉更新和代码合并的风险，同时能够插入到发布流程的其余部分。

关于作者

Yaniv Yehuda 是 [DBmaestro](#) 的联合创始人和 CTO，这是一家专注于数据库开发和部署技术的企业级软件开发公司。Yaniv 还是 Extreme Technology 的联合创始人和开发主管，这是一家面向以色列市场的 IT 服务提供商。Yaniv 曾经是以色列国防军计算机中心 Mamram 的一名上尉，他在那里担任软件工程经理。

查看英文原文：[The Secrets of Database Change Deployment Automation](#)

查看原文：[数据库变更部署自动化秘诀](#)

第二部分

实践经验

一些好的规则

作者 Peter Neumark , 译者 潘瑾瑜 发布于 2014 年 8 月 21 日

什么是明智的标准化

想象一下第一次和特别的人约会。当你到达最喜欢的餐馆时，所有的灯都熄灭了，你身处黑暗之中。奇怪的是，从厨房传来的声音又表明这里像往常一样正在营业中。你听到一位女服务员走来，等待着引导你到没有灯光照射到的座位上。你的同伴不知所措，并且有一点害怕。你是打算留下，还是找个正常点的地方吃饭？

Web 应用就像餐馆一样，人们通过其所提供的体验对其进行评价。即使是短暂的中断也会影响服务提供商的口碑或服务水平。政策和指导方针在防止代价高昂的服务中断中扮演着重要的角色。不幸的是，它们也能导致不理智决策的产生，从而造成更大的损害。比如公司内“DevOps 团队”的建立。这将导致所有的运维知识都被隔离在一个单独的团队中。尽管这样一个管理层指令可能预示着 DevOps 的到来，但它什么都不是。

工程师鄙视无逻辑的、官僚主义的规则。这些规则是前进的障碍物。然而，每家公司至少都会有一些这样的规则。在过去，可能有好的理由在一些问题上制定这样的规则。渐渐地，这些规则过时了。但是，规则制定者不能（或不敢）取消它们。当使用 C++ 代码库时，由于历史原因，被告知不能使用 STL；参与的 Java 项目被坚定地拒绝从 1.4 迁移到新版本。任何有过这样经历的人都明白有些措施可能会对生产力产生消极的影响。

我们应该忘记这些规则吗

面对这些像障碍物一样的规则，我们都很想将它们废除。不幸的是，“无为”的公司通常都没有成功地废除它们。好的规则是一种重要的交流形式。这种形式关乎到长期策略、从过去吸取到惨痛教训、以及来自用户需求中的发现。理想情况下，一个组织制定的与时俱进的规则，可以帮助个人增强做出正确决定的信心。在实践中，这样的情況真的发生过吗？

一家公司是否拥有真正有效的指导方针，Netflix 就是一个很好的例子。至少通过阅读他们的[博客](#)和[开源的代码](#)会给人留下这样的印象。比如，即使没有和 Netflix 的任何员

工聊过，我也能确定，“[构建它，运行它](#)”是一个他们如何把开发和运维结合起来的不错的想法。另一个明确的原则是：写代码是为了构建一个可靠的、可扩展的服务，而不是为了其他目的。他们开源了所写的大部分后台软件。这个事实比任何事都更具有说服力。

Netflix 已经构建了 Netflix 内部 Web 服务框架（NIWS）。这是一个自定义的软件栈，用于创建可靠地运行在云上的内部 Web 服务。NIWS 采用了一些不太流行的技术和不太常用的方法。使用这种与最佳实践背道而驰的方法需要有相当强的自信。毫无疑问，部分可以归因于落实的政策。这些政策让工程师可以不受限制地考虑问题。

Netflix 的负载均衡

在 Netflix 如何挑战常规的例子中，我最喜欢的是他们是如何在 NIWS 中实现负载均衡的。面向客户的流量仍使用传统的负载均衡处理器（一个标准的 Amazon EC2 ELB），但对于 Netflix 服务器之间的流量，他们选择了一个完全不同的方案，称作客户端负载均衡（client-side loadbalancing）。基本思想很简单：取消专门用于负载均衡的节点。这些节点用于在 Netflix 服务器间转发流量。客户端本身维持着一张列表，记录了可用的后台节点。当客户端发送请求时，直接与所选的后台实例交互。而这样就没有必要使用专门的负载均衡器。

客户端负载均衡并不是 Netflix 发明的。但是，它是有名的公司里第一个在基础架构中完全使用这种技术的（公平地说，同一时期内，Twitter 和 Yahoo 也在做基于相同概念的实验）。在多个后台服务器上做均衡的标准方法是：通过一个负载均衡器，如 Amazon EC2 ELB，或者在服务器上运行类似 HAProxy 的软件。对于这么关键的组件，使用保守的方法和一种大多数工程师都熟悉的技术是很有意义的。但是，几乎没有公司在 Netflix 之前试验客户端负载均衡的方法。其真正原因是，他们甚至都没有考虑到这种方法。

对于从事大规模应用程序开发的软件工程师，每天都要和各种库和组件打交道。这有点像鱼和水的关系。在能使用一种特定方法成功地构建系统这么多年后（也许几十年），对已经经过考验的方法或者系统的构建模块提出疑问，这看起来是在浪费时间。在许多公司里，这些决定已经被写进政策中。这些政策基本上是不可变的。但是，Netflix 采用了客户端负载均衡的方法，并因此取得了显著的成功。首先，他们从系统中移除了一个单点故障点（对于频繁地在没有警告下就停止服务的 EC2 实例，这是一个重大的胜利）。其次，通过将负载均衡的逻辑集成进客户端，负载均衡的策略可以参考客户端提供的信息。比如，考虑以下的负载均衡规则：

向客户端的 EC2 可访问区域（EC2 Availability Zone）中的可用节点发送请求。如果这样的实例不存在，则在当前区域中找一个运行已超过一天的实例替代。

传统的负载均衡器并没有被设计成用来执行这种自定义逻辑。它们也无法获取太多关于客户端的信息（比如一个客户端所属的 EC2 可访问区域）。自定义负载均衡逻辑变成了应用的一部分，使用相同的语言编写。这意味着编写代码的单元测试用例变得容易。而在传统上，这被认为是“基础设施的东西”。因此，这不仅让制定更复杂、更智能的决策成为可能；也使得人们对工作能如期完成更有信心。从某方面看，NIWS 将 DevOps 带入了下一个层次：开发人员和运维工程师不仅坐在一起，在同一个团队中工作；而且他们使用相同的开发语言，向相同的代码库提交更新。

Prezi 加入客户端负载均衡俱乐部

用一个内部的客户端负载均衡实现替代标准的负载均衡器，这种让 Netflix 受益的技术只适用于 Netflix 吗？不一定。在 prezi.com，我们对内部流量也采用了这种技术。我们的一些应用服务器运行着若干个服务。当这些服务通信时，我们希望它们优先选择本地的服务实例，而不是向网络中发送请求。然而，如果需要访问的服务没有运行在同一台服务器上，那么就可以访问任何一个该服务的实例。对于 Prezi，获得的好处是，尽可能地避免了网络流量、减少了在 AWS 上的支出和响应时间。目前运行于 prezi.com 产品上的负载均衡逻辑由以下的这段 Scala 代码实现：

```
override def choose(key: scala.Any): Server = Option(getLoadBalancer).map(lb =>

    lb.getServerList(true).filter{server =>

        server.getHost == config.getHostname && serverIsAvailable(lb, server)

    }.getOrElse(Seq()) match {

        case Seq() => super.choose(key)

        case matchedServers => matchedServers(0)

    }

}
```

Netflix 的工程师可以设计出 NIWS，而不用担心质疑当前技术所带来的后果。因为公司的规则授权他们这么做。即使任何人都可以获得 NIWS 的技术，只有那些有着类似思维的公司才能够使用这种技术去搭建产品。具体来讲，期望工程师基于技术价值做出决定的公司和完美主义的公司是无法利用这样的技术的。

Netflix 验证（Netflix test）

期望所有的工程师在做决定时不受办公室政治、流行技术和害怕改变的制约，这是不可能的。然而，减少这些方面的影响，对确保开发不会误入歧途有很大的帮助。一堆武断的限制规定会让工程师的设计缺乏创新和效用。相比之下，一些好的规则限制了问题空间、明确了约束、改善了产品的质量。

基于 NIWS 栈的源代码，Netflix 在决定如何实现一个组件时会考虑两件事：

1. 这个组件发生故障的可能性及后果是什么？
2. 当这个组件的设想场景发生改变时，是否容易修改这个组件的行为？

我将这些问题称为 Netflix 验证。这两个问题是紧密关联的。甚至可以说，第二个问题包含了第一个问题。这两个问题之所以意义重大，是在于他们如实地镜射出了 Netflix 的商业目标。这个目标就是提供可靠的、可扩展的服务。其他也有相同目标的公司也能从这个验证中受益。但是，这个验证的真正力量在于它没有提到的东西。它没有提到任何具体的技术或者[供应商](#)。

不适用于完美主义者

真正让我惊讶的是，Netflix 的代码只专注于足够好即可，而无过之。别误解，目前我所看到的代码容易阅读，并且有很高的单元测试覆盖率。即便如此，我也没有预料到 Netflix 能专注在足够好这个级别。比如，在代码的许多地方，当后台线程启动之后，就再没有任何操作停止它们。这看起来有很大的问题，直到你意识到 Netflix 不在节点上进行软件升级。他们通过启动一个新的 EC2 实例集群来部署新版本的应用。当通过监控验证新版本应用运行正常后，就将老集群关闭。如果有人使用了这些部署工具（也是开源的），那么就没有僵尸线程的问题。然而，如果有人在一个像 Glassfish 的应用服务器上使用 Netflix 的库，那么每次重新部署都将会触发内存泄漏。

代码中包含大量单例模式的类，也是我未预料到的。当我们以一种 Netflix 未预见的方式使用一个 NIWS 库时，我们很快会发现自己在不断挣扎地使用错综复杂的技术来处理问题。包括使用多个类加载器。

最后，尽管 wiki 页面上关于代码的文档有很大的帮助，但是这样的文档太少了，很多细节都没有描述。通常，代码就是文档。有几次，我在 github issue tracker 上找到了一些解决 NIWS 相关问题的最佳建议。

我的许多同事，在第一次接触 Netflix 生态圈时都有点不知所措。他们的第一反应是谴责那些写代码的工程师未经训练或者太懒惰。“应该有一些规则关闭这些没用的线程”，我听他们这样说着。然而，对于 Netflix，我们所列出的 NIWS 的缺点，都不算是一个真正的问题。用于处理线程关闭的时间被用在了其他更需要的地方。如果有人想要以不支持的方式重用代码，那么单例模式的类只是其中一个会遇到的问题。最后，尽管写文档是一件好事。但是，可读性高的代码和大量内部专业知识让文档成为了一个可选项。Netflix 建立了关于线程管理、恼人的设计模式和最小化文档量的规则。通过建立这些规则，让工程师可以专注于其他主要问题。

事实上，我已经意识到 Netflix 的软件栈之所以成功，是因为它有着旺盛的精力。这不仅让 Netflix“砍掉了软件栈的一些边角”的事实可以被接受，而且实际上也催生了一个更好的产品。编写了大量描述代码的文档还得保证文档不会过期，因为代码总是在不断的演进。编写不会用到的特性会使开发者失去动力、且难以为团队证明自己，对社区也没有什么好处，因为这部分代码不会在产品中被验证到。在 Prezi，我们有一些一直想开源的项目。但由于缺乏时间加入一些我们希望的改进，目前还不能将它们开源。Netflix 成功地开源了大量的代码却没有破产。因为它们一直专注于代码的可读性和单元测试，而不是加入过多的亮点，以及保证其不会过时。Netflix 实施的这些合理规则，使得它的设计开发可以应对不断快速增长的用户；甚至是不断开源所写的代码。

因此所有的特定规则都不好吗

如果用 Netflix 验证再形成一些指导方针，那么这些方针是相当通用的。例如，通过努力获得成功的名言，像“花 10% 的时间用于偿还技术债”，或者技术信息，如“0.6.1 版的 NodeJS 使我们的 Web 应用变得不稳定，别使用它”。如果把从过去失败中总结获取的教训忘了，这难道不是一种浪费吗？

这样的建议，和最佳实践、知名的组件一样，是非常有价值的。在加速开发和简化系统的运维方面，通过多年的验证，这些建议已经获得了工程师的信任。比如在 Prezi，大多数后台系统都是用 Python 写的，并使用了 gunicorn web 服务器、Django web 框架和 MySQL 数据库。在公司的初期，这个软件栈使得开发者能够专注于新产品的特性上。多年来，“使用 Django 和 MySQL 开发服务”就如同“不要在周五下午 3 点后部署”一样明确。这些都不是 Prezi 成文的规则，但却早已在实行中。

随着注册用户数从 0 攀升到 4000 万，许多当初采用这个平台的实际情况都已时过境迁。比如，当所有的网站流量都由一个应用处理时，将所有用户数据存入一个 MySQL

数据库中是有意义的。如今，Prezi 拥有许多独立的服务。这些服务对响应延时、可靠性和一致性上都有不同的需求。许多服务运行在 EC2 上，将数据库当做键一值存储的容器，通过主键访问数据。第一年制定的技术指导方针，尽管在那时有用，但没有一条能帮助我们应对目前工程上的挑战。

只要标准的技术和特定的规则没有过时，就能够激发工程师的产出。问题在于，当这些特定的规则不再适用时，仍然被强制实施。

固定的接口集

对于已经过时的规范而言，一个问题（而且很常见）是软件接口的过时。我最喜欢的例子是 Java Servlet API。即使它并没有真的过时！实际上，它是一个优秀的接口：直观、稳定、有完整的文档、很多不同应用服务器都是使用它实现的。

当 Prezi 决定探索 JVM，将其作为我们可靠的 Django 栈的一个可选平台时，我们选择了一个轻量级的代理应用作为我们的试用项目。我强烈地表明应该使用 Jetty 和 Servlet API，而不是团队考虑的另一个可选方案。这个方案使用一个不知名的 Scala Web 服务器。6 个月之后，我们关闭了原有的代理程序，而用一个基于 [Spray](#)（这个技术我是投反对票的）写的程序取而代之。部分原因是：对于我们的用例，使用它可以获得更多效率。因为在我们的用例中，响应时间主要受发出的 HTTP 请求的响应延时的影响。我开始从代码层面思考：我们想要什么样的目标，想使用什么样的接口。我们如何写单元测试。开发者社区有多大。这正是 Servlet API 在抽象层面解决的问题。我本应该考虑（或谈论）关于它是如何利用硬件资源的。具体来说，瓶颈在于：处理请求时是否需要大量的 CPU 或者 IO 资源。由于在我们的用例中，大部分时间都花费在等待发出的 HTTP 请求的响应上，所以没有这样的资源要求。这就是代理程序的本质。鉴于我们的用例，使用 Servlet 的方法对每一个请求都创建一个专门的线程，不仅毫无必要地限制了处理请求的并行数，而且也无法高效地利用内存。

Servlet API 不适用于这个问题的事实，并不能说明那些常用的接口或 Java 编程语言不好。数以千计的公司使用 Servlet 构建了令人惊叹的产品。其他编程语言也具有相似语义的 Web 服务器接口。这个故事想表明的是，我在使用特定的指导方针时脱离了实际的场景。接口是用来解决某一个特定问题的。当问题不再是尝试解决的问题时，使用给定的接口不是一个好的选择（无论这个接口有多流行或者多新颖）。

DevOps 的规则

DevOps 能量来自于合作中的人有着完全不同的技能。相比于成员技能单一的团队；一个拥有各种不同技能的团队，包括长满胡子的系统管理员、函数式编程的狂热粉丝，更有可能构建出可靠和可扩展的服务。

成员技术背景的不同使得团队更加需要明确的规则。开发者不需要知道为什么使用的自定义 Linux 内核有着一大串的编译参数。类似地，不是所有人都需要担心代码中有多少单例模式对象的存在。“写 shell 脚本时必须添加 shebang 行”，或者“解析用户数据的代码要有单元测试”。像这样的标准适用于团队中的每一个人，并且会帮助到那些在特定领域内没有足够经验做好事情的人。特定规则只有被适当的使用，才会对团队产生积极的作用。

更通用的规则，像这些 Netflix 验证只适用于制定高层级决策，但是能够应用地更久。管理团队既需要通用的规则也需要高层级的规则。诀窍是要及时发现我们制定的规则是否已不再发挥期望的作用。

如果我们回到文章开头的那个餐馆，打开冰箱门，不同盒子上有着不同的保质期时间。有的可能几个月，比如番茄酱；有的可能几个小时，比如鱼。做饭要用到不同的原料，而每种原料有自己的保质期。保持原料的新鲜，使得最终做出的食物可口，这是一个厨师的责任。同样，不仅在我们决定要将什么进行标准化这件事上需要智慧，在及时发现我们的标准是否已失去意义这件事上也需要真正的智慧。

关于作者

Peter Neumark 是 Prezi 的一名 devops 人员。他和妻子 Anna，以及两个儿子住在匈牙利的布达佩斯。当不用调试 python 代码或换尿布时，Peter 喜欢骑自行车。

查看英文原文：[A Few Good Rules](#)

感谢[赵震一](#)对本文的审校。

查看原文：[一些好的规则](#)

携程首席架构师谈 DevOps：找到合适的人最重要

受访者 Eric Ye 作者 杨赛 发布于 2014 年 8 月 25 日

个人简介 叶亚明（Eric Ye），携程首席架构师，负责移动、Web、呼叫中心等部门的研发工作，领导开发的业务和领域包括酒店、机票、商务旅游、开放 API、全球站、用户体验研究。他从过去十年的电子商务变革中，总结出六种有效的编程模型，目前被广泛应用于携程内部的产品研发过程中。此外，他还致力于升级携程网架构并创建新一代框架，以提高可扩展性和可用性。

全球架构师峰会（International Architect Summit，下简称 ArchSummit）是由 InfoQ 中文站主办的一次全球性架构师峰会。ArchSummit 专门针对架构师人群，讲述与架构和架构师相关的各方面趋势、技术和案例。这也是继 QCon 之后，InfoQ 中文站主办的又一次高端技术盛会。

InfoQ：大家好，我是 InfoQ 的主持人，现在在 ArchSummit 大会现场。今天十分高兴的邀请到携程的技术负责人高级技术副总裁 Eric Ye（叶亚明）来接受我们的采访。今天的话题是运维工程师在时代的职业发展，首先您是如何对运维的工作如何划分的？包括在携程这一块你们对运维工程师是怎样分工的？

Eric Ye：运维工程师在携程，或者是在大部分的中国互联网公司，一般是按照他们的功能划分的，比如说 DBA、SA、应用运维、Security、还有 Storage、network，加上 Tooling，还有一部分是负责上架下架的 siteops。

InfoQ：为大规模系统做运维，大概是大型互联网公司开始就是比较兴起的，然后您在这里也是有了十几年的经验了，然后您觉得这个大规模系统运维的思路和理念在过去几年有什么比较大的变化？

Eric Ye：这个问题，我结合我在海内外的一些经历，稍微展开来讲一下，大致分为 3 个阶段。

第一阶段：1995 年到 2005 年，互联网刚刚兴起的这个十年，互联网公司随着用户量、流量增加，运维变得越来越复杂，大家还是凭着运维工程师的经验来做事

情，按照前面的功能划分，那些工程师对某个方面经验会越来越深入，但是从本质上，还是依赖于人手工来做运维的。

第二阶段，过去凭经验的手工方式，一方面反应慢，另一方面有经验的人毕竟很少，怎么能提高运维的效率，缩短排障时间，他们开发了好多工具。但是有些工具要求基础设施必须要标准化，简单化；如果基础设施很复杂，五花八门的设备，加上架构也不一样，自然影响自动化程度，运维工程师的压力会越来越大。

第三阶段，最近几年，像 Facebook, LinkedIn 这些公司已经走在前列，负责运维的人很少。像 Facebook 这么大的公司，运维人员就 20 个左右，这么大的流量，它是怎么做到的？它们有 DevOps 这样一个很超前的理念，在开发人员完成功能开发之后到生产环境发布，基本上全自动起来。

除了 Facebook/LinkedIn 还有一类是云计算的供应商，像亚马逊，Google，他们不光是把 DevOps 贯彻在公司里头，还要将 DevOps 展现给第三方开发者。

InfoQ：那么既然讲到 DevOps，其实 DevOps 像您刚才也提到，它涉及一个全链路的，包括从测试，到最后的部署，整个环节。那么如果一个公司的运维团队，现在还没有开始这个阶段，那么他们从哪里开始介入会比较容易一些？

Eric Ye：你这个问题问的还蛮深的，开发人员完成了一个功能之后到生产发布的所有环节，DevOps 负责将其全自动化。

DevOps 不光是个概念，它是这样一套系统，开发人员有了它，开发人员就是 Ops 人员，与传统概念“开发人员不管运维”是相反的。这套系统有哪些组成部分？第一是，你要测试，作为开发人员写完代码以后，要做 Unit test、功能 Test、CI test，这些对开发人员来说是蛮自然的。之后要到 staging，通过整个功能集成测试，再将功能代码发布到生产环境。但生产环境，Dev 的人员是不能直接接触的，通过 DevOps 把生产环境抽象起来展现给开发人员，也就是说开发人员也不需要知道后面的生产环境是什么样子，通过 DevOps 平台，可以方便快捷的发布到标准的基础设施之上。听起来很简单，但实际上发布需要非常严谨，有阶梯式的发布（灰度发布），尤其像携程这个规模，每个集群规模都不小，那这些发布过程中一旦碰到问题怎么办？如果一个代码在测试的环境没问题，但生产环境上又有问题怎么办？这背后需要有一套自动检测的工具，监控整个发布过程，当代码出错率高到一定程度，可以自动刹车，自动回滚，如果代码质量符合生产环境的要求，就往前推进，一直到整个生产环境完成发布，没有问题以后，开发人员才能离开这个生产环境。

这一整个过程都由开发去掌控，这样全自动的效率会非常之高；刚才讲了 DevOps 整个流程的细节，这里头门槛还是蛮高的，如果是传统运维的人，会觉得这是蛮不可思议的事情。从开发人员来说，这怎么弄？我以前只知道写代码，这个整个过程是怎么玩儿的。

对于成功的那些公司像 LinkedIn、Google、Facebook，是怎么做这件事情的呢？它们有个组织架构，叫做 Infrastructure Tooling，既做基础设施，又负责自动化工具研发，他们是蛮资深的工程师，还不是一般的产品开发的工程师，而是对系统、对基础设施的建设非常有兴趣的一些人。

现在来回答你的问题，作为一个公司，他如果没有 DevOps 怎么进入，怎么去开发这套系统？我觉得门槛是蛮高的，第一你要找到合适的人，这些人是对基础设施、对工具很有兴趣、很有研究的一些人；第二他们需要了解当前开源社区、业界的技术趋势，有很多与 DevOps 相关的开源系统已经非常流行，比如说代码管理工具 Git，代码 Code review 工具 Review Board/Gerrit，还有像 CI 方面有 Jenkins。需要借助于现有开源系统来构建 DevOps；第三基础设施的标准化非常重要，没有标准就无法实现最大程度自动化。有了基础设施标准，这套系统应该是每个开发工程师的开发、测试、发布用到的工具是一样的。如果每个团队、部门都不一致，那个 DevOps 又变成一个空的东西。

总之 DevOps 涉及到的细节蛮多的，如何打造，简而言之，就是要有合适的、有经验的人，起点要站的高一点，不要闭门造车，在标准的基础设施上构建各种涵盖各个环节的统一工具。

InfoQ：第一步找到合适的人最重要的，而且这个人是不是我从企业内部，他本身了解这套系统会比较好，因为如果从外头引入的话，他可能不了解这套系统？

Eric Ye：两方面的力量都要结合。内部工程师对现有系统比较了解，是有帮助的，因为我们做 DevOps 这样的系统，是要解决他们眼前碰到的问题。不过这还是不够，你需要引进知道 DevOps 的概念，知道 DevOps 怎么搭建的人才，这两个力量会产生合力，更快开发出 DevOps 系统。自己学的话还是比较慢，而且容易走弯路。

InfoQ：刚才您也提到了云计算的那个运维跟之前的运维也不太一样，云计算是将很多这种边界的工作，就是以前开发、运维的边界，他移到了运维这个端。然后如果是传统的运维

工程师他到了云计算会无法掌控，因为它要复杂很多，比如说很多运维吐槽 OpenStack，说我搞不定，他就没有办法去运维那套系统，您觉得这是因为云计算的系统做的还不够友好，还是说这些人的能力跟不上？

Eric Ye: 我觉得两个原因都有，第一云计算不算很成熟，也很复杂，涉及到的东西太多，刚才讲到有 Storage、Network、Compute，也涉及到跟开发有关的发布，应用的运营，还有 Scale UP, Scale down，有弹性计算能力，在世界上比较成功的云计算公司也是少数几家，都是以技术为导向的，驾驭这个产品对技术要求非常高，不是只靠一个运营团队就能搞定的，所以难度比较大。

第二对于传统运维工程师，这些理念对他是蛮挑战的，因为他以前知道系统是怎么运维的，DB 如何管理，但这些技能在云计算面前，它发不出力，有的时候都找不到门。那让他们去驾驭云计算平台，需要一个重新学习的过程，好比他以前是开车的，一下子要他开飞机，这个难度还是不小的。所以说他必须要学习了。

你刚才讲到的 OpenStack 最近是比较火，但 OpenStack 的历史来看相对短，没有多少年时间，并且涉及到的模块、技术非常广，质量还在提升过程中。如果你有一个团队去用 OpenStack，三个月到六个月，只能知道一些皮毛，里面的水还是蛮深的。也可以这么说一些运维团队，要真正去学会 OpenStack，驾驭 OpenStack，难度非常之大，会觉得有点力不从心，容易觉得这个系统太庞大、不好用，或者是 Bug 比较多。

携程算是比较早就用 OpenStack 了，大概在一年半以前已经进入，我们现在很多的系统已经基于 OpenStack 来做，我前面提到基础设施标准化，就用 OpenStack 的方式去实现，而不是用一个文档规范来标准化。另外携程有一个比较独特的 OpenStack 应用场景，就是呼叫中心虚拟桌面云。所有的呼叫中心不再需要台式机，呼叫中心员工办公只需要一个云客户端加显示器即可，真正的桌面都运行在后端的云里面。虚拟桌面云整个平台，包括后端对桌面、云终端的运维管理、资源分配调度、动态伸缩等等功能，都是基于 OpenStack 来打造的。在整个过程中，我们也碰到了很多坑，但我们还是跃过去了，给 OpenStack 修复了很多 bug。一旦研发到这个深度，OpenStack 会对携程这样的企业，或者其他的企业很有价值。如果光是去做一个 POC，十几个人去用用它，用了三个月以后觉得太复杂放弃，那么很难发现它的真正价值。

总体来说，造成不少对 OpenStack 吐槽的现状，不仅仅是运维的能力不够，也因为 OpenStack 还不够成熟，这两方面都有。

InfoQ: 最后一个问题，就是长远来看，有个预测，就是未来可能整个 IT 架构会逐渐往云的方向倾斜，可能像小的企业他也不太会倾向于自己架台机器跑跑，那么所以就有可能运维工程师都跑到云平台去做了，没有进云平台的就只好改行了。您对这个运维工程师这个工种未来是怎么看？

Eric Ye: 随着云技术越来越深入，越来越成熟以后，云技术这个趋势是不可避免的。一些小的创业公司，小的企业，把系统搬到云上，难度不是太大。举例来说，原本需要一百人的团队来开发、运维云系统，现在十个人就可以搞定。小公司对运维工程师的需求，会随着云技术的推进会减少。所以说这种职业需求在这些公司会缩小。

对于中型的公司，又面临这么一个选择，已经有了一定的规模的 IT 设施，把这些 IT 设施搬迁到云上去，从技术上说是可行的，但是需要很多改动，需要逐步落地，这个改动落地是一个成本，可能有这样的疑问，已经有成熟的团队，可以把现有基础设施维护好，那为什么还要搬迁到云上去？搬迁过程还有额外成本。但是因为一旦搬迁落地后，它对运维的需求就大大降低了，可以说是一个长痛还是短痛的选择问题。

云技术的好处除了整个运营成本会降低，还能支持弹性计算，当网站流量突然增大的时候，能快速扩容；所以很多中小企业非常愿意采用云技术。当然不是说有了云技术就不需要运维了，而是会直接减少运维的人员数量。

大企业接纳云技术，需要一个漫长的过程，一方面现在提供云技术的公司，还不能直接快速的把大企业的应用搬迁上来就能直接运行；背后还需要大量的工作，并且出于数据安全的考虑，支撑业务的核心应用会很长一段时间在由内部运维人员进行管理，但边缘的一些应用或者新开发的应用、比较符合标准的应用，搬到云技术上还是可行的。我认为这些大企业会分批的，逐步尝试云技术，这个过程应该会很漫长。

对于你的问题，那些运维的人员在云的前面会不会成为消失的工种？这个工种会长时间存在，但是长远来看，随着云计算的成熟，对于传统运维的技能的需求会逐渐的降低，所以如果一个运维工程师要去规划自己的职业，他应该去考虑这个问题，适应这个趋势，重新学习，才是长远之计。

InfoQ: 十分感谢 Eric 接受我们的采访。

查看原文：[携程首席架构师谈 DevOps：找到合适的人最重要](#)

基于反馈控制实现可靠的自动扩展服务

作者 Philipp K. Janert，译者 梅雪松 发布于 2014 年 3 月 13 日

介绍

当我们部署服务应用到生产环境时，需要决定在线的服务器数量。这是个困难的决定，因为对于一个指定的流量负载，通常我们不知道需要多少服务器。因此，人们为了“安全起见”，不得不使用更多（也可能是过多）的服务器。但是服务器是有成本的，所以这样做会让事情变得过于昂贵。

事实上，事情比这还要糟糕。流量不会一整天都保持不变。如果我们按高峰期流量部署服务器，那么基本上大部分的服务器大多数时间都处于使用率不足的状态。特别是在一个基于云的部署场景中，服务器实例能够在任何时刻自由分配，我们应该能够意识到，如果不不论何时，我们都只激活处理该时刻负载所需的服务器实例，那么就能显著地节省成本。

对于这个问题，一种可能的办法是使用一个固定的时间表，对一天中的每个小时都指定（以某种方式）所需的服务器实例数量。这种方法的难点是固定的时间表无法处理随机的变化：如果由于某种原因，今天的流量比昨天大 10%，那么这个时间表将无法提供额外的服务器来处理这个意料之外的负载。同样地，如果流量峰值提前了半个小时，一个基于固定时间表的系统也将无法应对。

与其使用一个固定的（基于时间的）计划，我们也可以考虑一种基于规则的解决方案：对任何给定的流量负载，我们有一种规则来指定使用多少服务器实例。这个方案比基于时间表的方案更具有弹性，但它需要我们（提前）知道处理每种流量负载需要多少服务器。而且如果流量的性质发生变化，这是有可能发生的，例如，如果需要长时间运行的查询的比例增加了，会发生什么呢？基于规则的方案将无法正确应对。

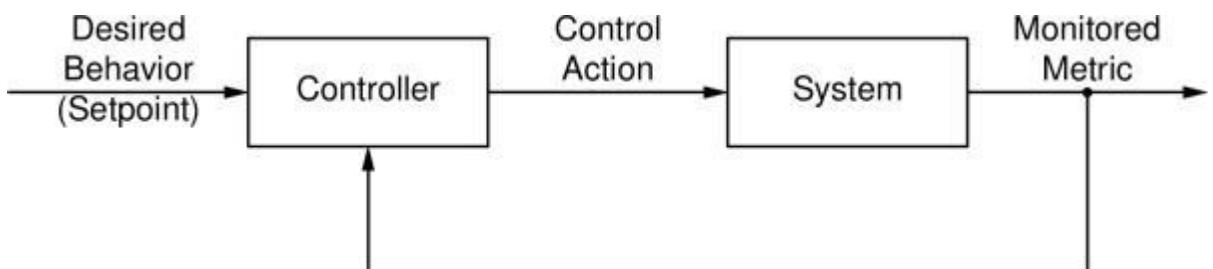
反馈控制是一种设计模式，它完全能够处理所有这些挑战。通过持续监视一些服务指标（例如响应时间），如果指标值偏离期望值，则做出适当的调整（例如增加或减少服务器）。因为反馈来源于被控制系统的实际行为，因此它能够处理甚至不可预见的事件（例如流量超出了所有的预期）。另外，对比之前基于规则的方案，反馈控制只需要很少的被控制系统的相关信息。其原因是反馈是真正的自我修正：因为持续对服务质量指标进行着监测，一旦它偏离期望值，将被立即观察到并立即进行修正。而且

在必要时，这个过程是可以重复的。简单地说：如果响应时间恶化，反馈控制系统只需激活额外的服务器实例，如果仍然没有改善，它就再增加更多的服务器实例。这就是反馈控制的全部过程。

反馈控制一直是机械和电气工程的标准方法，但在软件架构中，它很少作为一种设计理念。它特别适用于信息不完整，并且会随机变化的情况，它与计算机科学确定性的算法解决方案完全不同。最后，尽管反馈控制的概念很简单，但要使其生效，在一个生产环境中部署一个真实的控制器仍然需要了解和明白一些实战“技巧”。本文将介绍一些概念和常见的困难。

反馈回路的本质

下图显示的是一个基本的反馈回路。在右侧，我们看到控制系统。它的“输出”是相关的服务质量指标。指标的值不停地提供给控制器，与左侧输入的期望值进行比较（系统的输出指标的期望值被称为“设定值”）。基于这两个输入（服务质量指标的实际值与期望值），控制器计算出控制系统合适的控制动作。例如，如果响应时间的实际值比期望值慢，控制动作将包括激活额外的服务器实例。



该图显示了所有反馈回路的通用结构。它的基本组件是控制器和被控制的系统。信息从系统的输出流经返回路径到达控制器，与“设定值”进行比较。通过这两个输入，控制器决定相应的控制动作。

那么，控制器实际上做了哪些工作？它如何决定应该采取什么动作？

要回答这个问题，我们要记住使用反馈控制的主要目的是减少系统实际输出与期望值的偏差。这个偏差可以表示为“跟踪误差”：

$$\text{误差} = \text{实际值} - \text{期望值}$$

为了减少这种误差，控制器可以做任何它认为合适的事情。我们当然有绝对的自由来设计这个算法，但是我们要知道被控制系统的一些知识。

让我们重新考虑数据中心这个场景。我们知道增加服务器数量可以减少平均响应时间。所以，我们可以选择这样一种控制策略，当实际响应时间比期望值差时，就增加在线服务器的数量（如果情况相反，则减少服务器数量）。但其实我们还能做得更好，因为这个算法只有标志，没有将误差的大小考虑在内。如果跟踪误差很大，我们就应该做更大的调整。事实上，通常的做法是将控制动作正比于跟踪误差。

$$\text{动作} = k * \text{误差}$$

其中 k 是一个数值。选择这种算法，大的偏差将导致大的修正动作，而小的偏差将导致相应较小的修正。这两个方面都很重要：大动作是为了快速减少大的偏差。但同样重要的是当误差较小时，控制动作应该变小。我们只有做到这一点，控制循环才会趋向于一个稳定的状态。否则，其行为将总是围绕着期望值振荡，通常我们希望避免这种结果。

正如我们之前所说的：可以自由地为反馈控制器选择特定算法，但通常保持简单是一种好的设计思想。反馈控制的“神奇”取决于信息流的回路结构，而不是那些特别复杂的控制器。为了允许更简单的控制器，反馈控制需要更复杂的系统架构。

然而有一件事是必须确保的：控制动作必须在正常的方向上。为了保证这一点，我们应该对被控制系统的一些行为有所了解。通常这不是问题，正如我们知道更多的服务器意味着更快的响应时间，等等。但是这是一个我们必须知道的关键信息。

具体实现面临的问题

目前为止，我们关于反馈控制的描述大部分是概念。然而，将这些高级别的想法转换成具体实现时，需要处理一些实现细节。最重要的是对于跟踪误差的大小应该产生多大的控制动作。（如果我们使用之前的公式，就是要为常量 k 选择一个值。）

在控制实现中，为这个常量选择特定值的过程，被称为控制器“调试”。控制器调试是工程上权衡的一种表达方式：如果我们选择相对较小的控制动作，控制器将缓慢地响应，跟踪误差将保持较长时间。另一方面，如果我们选择较大的控制动作，控制器将快速响应，但是也存在“过度修正”的风险，导致相反方向的误差。如果我们让控制器做过大的动作，可能让控制回路变得不稳定：当发生这种情况时，控制器尝试用不断增长的序列动作来补偿每个偏差，所有时间都在增加动作幅度，从一个极端摆到另一

个极端。这种形式的不稳定比缓慢操作更糟糕，因此必须避免。控制器调试的挑战是在不造成回路不稳定的前提下，找到控制动作的最大值。

在选择控制动作的大小时，通常的作法是逆向工作：给定一个某种规模的跟踪误差，多大的修正动作能够完全消除这个误差？请记住，我们不需要精确地知道这个值，反馈控制的自修正特性确保在选择调试参数值时可以有一定的公差。但我们至少要保证量级的顺序是正确的。（换句话说：要将平均查询响应时间提高 0.1 秒，我们大约要增加一台、十台还是一百台服务器？）

有些系统对控制动作的响应比较慢。例如，一台新的（虚拟的）服务实例，可能要花几分钟的时间，才能开始接收传入的请求。在这种情况下，我们要把这种滞后或延迟考虑在内：在增加的实例生效前，跟踪误差将持续存在，我们必须防止控制器增加更多的实例，否则，我们最终将有太多的服务器在线。不能快速响应的系统是一种特殊的挑战，需要加倍小心。然而，已经有现成的方法来“调试”这类系统。（基本上，首先需要了解滞后或延迟的时间，然后使用专门的插件公式获得调试参数值。）

需要特别考虑的因素

我们要始终牢记反馈控制是一种反应式控制策略：事情会先变得糟糕（至少有一点糟糕），然后才会有修正动作。如果这是不能接受的，那么反馈控制可能不适合。在实践中，这通常不是问题：一个调试良好的反馈控制器将检测并响应甚至非常小的偏差，通常能保证系统比基于规则的策略或人工操作更接近于它所需的行为状态。

一个更加需要重点关注的情况是，没有一种反应式控制策略能够处理比控制动作生效更快的干扰。例如，如果增加在线服务器实例需要几分钟时间，那么我们无法响应几秒甚至更短时间内的流量峰值。（与此同时，我们完全能够处理几分钟或几小时内的流量变化。）如果我们需要处理波形非常尖的负载，那么我们必须想办法提高控制动作的速度（例如提供热备用服务器），或者采用非反应式的机制（例如使用消息缓冲）。

另一个值得考虑的问题是如何选择服务质量的度量方式。反馈控制器最终做的唯一事情是保持输出值与期望值匹配，因此我们要确保我们选择的度量方式能够很好地表达这种行为。同时，在任何时间，这种度量方式都必须是有效且快速的。（例如，我们无法在一个有明显延迟的度量方式上构建一个有效的控制策略。）最后要考虑的是这种度量方式不能有太多的噪音，因为噪音可能“干扰”控制器。如果这种度量方式本身会有噪音，那么通常需要先进行平滑处理之后才能用于控制信号。（例如，与最近一

个请求的响应时间相比，最近数个请求的平均响应时间是一个更好的信号：取平均值能够平滑随机的变化。）

总结

虽然我们介绍了数据中心自动扩展的反馈控制，但它其实有更广泛的应用领域：不论我们是需要维护一些期望的行为，还是面对不确定和变化，都应该考虑将反馈控制作为一个选项。它比确定性的方法更可靠，也比基于规则的解决方案更简单。但它需要新的思考方式，并且明白一些特定的技术才能生效。

延伸阅读

本文只介绍了反馈控制的一些基本概念。更多信息请访问[我的博客](#)或者阅读[我的书](#)（*Feedback Control for Computer Systems*; O'Reilly, 2013）。

关于作者

Philipp K. Janert 提供数据分析和算法建模的咨询服务，他之前的职业是物理学家和软件工程师。他是最畅销书《基于开源工具的数据分析》和 *Gnuplot in Action: Understanding Data with Graphs* 的作者。在他最近的著作 *Feedback Control for Computer Systems* 中，他演示了汽车巡航控制的原理也同样适用于数据中心的管理和其他企业系统。他也为 O'Reilly Network、IBM developerWorks 和 IEEE Software 撰写文章。他拥有华盛顿大学理论物理学的博士学位。你可以访问他公司的[网站](#)。

查看英文原文：[Reliable Auto-Scaling using Feedback Control](#)

查看原文：[基于反馈控制实现可靠的自动扩展服务](#)

梁定安：解密腾讯 SNG 云运维平台“织云”

作者 刘宇 发布于 2014 年 9 月 24 日

SNG 是腾讯体量最大、产品线最丰富的一个事业群，其覆盖了 QQ、手机 QQ、腾讯开放平台、腾讯云平台、广点通、移动分发平台应用宝在内的多条业务线。可见 SNG 的运维体系的庞大，早在 2013 年 QCon 北京大会上，腾讯业务运维 T4 专家、总监赵建春就在 QCon 分享过 [《海量 SNS 社区网站高效运维探索》](#)，当时引起了运维界的广泛关注；而整个 SNG 的运维又是如何运作的呢？

梁定安，2009 年加入腾讯运营部，先后从事系统运维、业务运维、运维规划和运营开发的工作，目前是社交平台业务运维组 Leader，可以说是整个 SNG 云平台的缔造者，也是今年 [QCon 上海 2014](#) 大会自动化运维的讲师，届时将分享《腾讯 SNG 织云自动化运维体系》的话题。

为什么会有织云？织云重点解决什么样的问题？面对错综复杂的业务，织云又是如何自寻突变的呢？梁定安会全面介绍这个平台的特性、底层技术组成、以及给 SNG 所带来的价值。

InfoQ：梁定安你好，织云是什么时候开始做的？

梁定安：2012 年底 CTO Tony 将云化战略推广到公司内部各个 BG，织云正是在公司云战略的背景下，为 SNG 量身定做的内部云管理平台，定位是为 SNG 自研业务提供虚拟化管理和自动化运维的平台，几乎所有 SNG 的业务（Qzone、QQ 秀、QQ 相册、QQ 音乐、QQ 等）的运维操作都基于织云平台完成。

InfoQ：织云定位为内部的自动化运维平台，那么它具备那些特征呢？

梁定安：织云平台定义了 SNG 业务的标准化运营规范，在平台中运维人员抽象出上层的管理节点，减少与统一运维对象，降低海量运维的复杂度，得益于运营环境的标准化建设，有更多通用的自动化工具被设计开发，配合流程引擎的驱动，使我们逐步迈入自动化的运维阶段。平台最大的特色是“一键上云”帮助 SNG 自研业务快速实现织云最初的上云目标；而“自动调度”则实现标准化服务的容量自动维护；还有我们基于内核 inotify 的一致性监控，保证了配置资源与现网资源的一致性。此外，织云的核心模块

还有：资源管理，包管理，配置管理，自动流程，中心文件源，权限中心，都是自动化运维必不可缺的重要功能模块之一。

InfoQ：“一键上云”实现没有那么容易吧？非标准化业务应该是很难接入，在实现这一目标时，你们又遇到那些难题？

梁定安：困难是必然存在的，我们第一个遇到的难题是虚拟化选型和适配改造，在 XEN 和 LXC 之间，我们选择更轻量成本更低的 LXC 作为织云平台的虚拟机，在运营过程中，由于虚拟机与实体机管理模式的差异，我们没少踩坑。如同母机下子机对 CPU、网卡流量抢占造成相互影响，子机多 crontab 集中调度，常用系统命令只显示母机状态，LXC 内网 NAT 管理改变原运维习惯等等问题，都被一一啃下，LXC 顺利本土化成功。

第二个难题是运维标准化的普及，像 QQ 秀、会员这类腾讯比较早期的业务，在当时没有标准化规范的背景下，现网的运维管理有诸多阻碍顺利上云的问题点，如程序混布、hardcode IP 地址、依赖非标准库等等，在接入织云前都要经过运维和开发的适配改造。为了顺利推动让业务开发配合运维做上云的改造，我们设计了织云的“自动调度”、“一致性监控”、“权限中心”等核心功能，让开发改造的工作量有了充足的利好回报，让旧业务标准化的改造如期完成。我们乐观的看到织云平台使 SNG 运维从 D/O 分离转型为 DevOps 模式。在织云平台中，没有运维和开发的严格区分，平台的用户会在既定的标准管理框架中，对统一管理对象——模块，进行录入或变更资源配置（包、配置文件、权限、目录、脚本），流程引擎则会按照既定的次序和调用不同的自动工具，完成用户的一键上云或其他变更需求，而这一切都会在织云的标准化体系保障中自动化的实现。

一键上云的成功，得益于 SNG 自研业务的标准化运维管理的良好基础，我们所有程序的包管理（pkg 包系统）、配置文件 svn 管理（CC 系统）、目录管理（中心文件源）、权限管理（标准 api）、名字服务（L5）的高覆盖率，都可以经过轻量的改造即可成为织云平台的功能之一。再辅以流程引擎，将上云的步骤串成自动化的流程。用户只需在织云平台上管理好模块与依赖资源的关系，织云便可以一键式的完成整个迁云的过程。

InfoQ：织云的自动调度是实现业务动态扩缩容？你们又是如何控制“雪崩”的？面对业务的大量突发，是全自动，还是人工干预？

梁定安：是的，我们认为当一个模块可以灵活的实现扩容自动化时，它便具备了跨 IDC/城市迁移的调度能力。同理，我们对运维的业务按核心功能的不同分别抽象成不同类型的 SET/服务视图（包含多个模块），当整个 SET 的标准化程度且 SET 内的模块都具备自动调度的基础能力时，我们便可以对整个 SET 进行调度操，目前 Qzone、说说、广点通等重点业务的 SET 已经具备快速调度能力。

雪崩的预防是负载均衡管理中必须解决的一个问题，在 SNG 我们拥有一款十分出色的负载均衡+容错组件 L5，L5 利用名字服务将一个集群标识成一个 L5ID，主调方可以通过嵌入 L5api 并调用 get_route 来获取被调 L5ID 中的每个 IP: port，然后当请求结束后 update_route 来更新该 L5ID 的每个 IP:port 的成功与延时信息，L5 组件便可以通过全局数据的整合，在下个调用时间片动态的为 L5ID 下的每个 IP 分配合适的请求量，利用这个原理，L5 根据实际每个被调 IP:port 的请求量、成功率和延时的波动数据，可以计算出每个 IP: port 最大可支持的请求量，当遇到业务请求量陡增的场景，L5 会启动过载保护，在保证被调方饱和的请求量前提下，对新到的请求全部拒绝服务，以防止雪崩的发生。这些都是由 L5 组件全自动实现的。

织云的自动调度原理是对服务/模块的容量指标（CPU/流量）进行监控，当触发扩容阀值时，织云后台便自动触发部署、测试、灰度、上线这一系列的全自动流程，保证线上服务容量的可用，除非耗尽可用设备 buff，否则织云的自动调度功能辅以 L5 的优秀容错能力，可保持业务容量处于高可用的水平。

InfoQ：一致性应该是多个方面的，包括上面有提到一致性监控，还有织云的另一大特色服务一致性管理，这里的关系与具体包含的内容都有那些？

梁定安：先介绍下一致性本身，这是一套 C/S 的监控程序，C 利用内核 inotify 订阅监控的对象，并通过动态上报的架构，在一个集群内选取一个 leader 汇总数据后，传输到 S 进行数据落地，实测性能可达监控 1000 个文件秒级感知，是我们实现织云前台实时监控现网环境的核心手段。

回到一致性管理，视乎场景的不同，具体可以分为两大类：资源的一致性和服务的一致性。资源的一致性，比较容易理解，织云自动调度中依赖的资源，包括：包（进程、版本、运行状态）、配置文件（md5）、目录

(md5)、权限、后置脚本 (md5)。通过一致性的管理，使织云能够在无人职守的前提下，自动的变更运营环境。服务的一致性则与 SNG 业务的形态耦合比较重，如 Qzone 的多地容灾分布，每地的服务彼此一致，主要也是利用一致性的监控管理，服务一致性管理的会比资源一致性管理纬度要粗，往往只包含多个模块的包、通用目录、权限这 3 类。

InfoQ：织云的核心模块有：资源管理，包管理，配置管理，自动流程，一致性监控，中心文件源。对于开发团队而言，在织云做这些操作，和不使用织云有什么区别？大体的操作流程是什么样子的？

梁定安：织云提供给开发和运维团队的是工作效率的提升，让我逐一为大家介绍这些核心模块的功能。

资源管理，我们对现网操作的最小管理对象——模块，部署上线所依赖的所有资源，和操作这些资源的先后次序，都将录入到织云的资源配置，并存入模块的 CMDB 配置管理数据库，成为该模块必不可少的属性之一，结合自动流程可实现多种自动化的操作。原则上模块的资源配置只能由少数模块负责人修改，并且当资源发生变更时，织云提供锁表的能力，防止交叉篡改。没有织云的资源管理，运维/开发只能依赖 wiki、文档等古老的方式记录，并且无法自动化。包管理，是 SNG 一个最基础的系统，运维要求每个上线运营的程序，都必须按照 SNG 包管理规范打包，包管理本身提供了一套完整的框架，包括：进程名字与端口管理，启停方式统一管理，标准化的目录结构，完善的进程监控体系，灵活升级回滚的 svn 管理等等。包管理是一切标准化的基础，离开它，运营环境将一片狼藉。

配置管理，指的是配置文件管理，包括线上编辑、版本迭代、diff、前后置脚本、批量下发/回滚等等功能，是单文件管理的利器。没有它的话，就只能回到脚本发布时代了。

自动流程，指的是织云的流程引擎，主要功能是将不同的标准化工具串联起来，前者的输出可作为后续工具输入用途，支持流程分支汇总的能力，可通过串联不同的工具，拼装出不同的自动化流程，实现无人职守的各种操作。这就是人肉和自动化的差别。

一致性监控，主要就是秒级感知现网变化的能力。有了它，开发再也不会提让运维批量查一批 IP 的配置是否一致的需求了。

中心文件源，主要是为目录管理诞生的一套 svn 管理的系统，可根据策略不同，自动与现网同步或被同步，且支持大批量的分发能力。没有它，大伙还是只能靠脚本来进行目录类的发布管理，效率低下。

InfoQ：在织云上的业务部署监控、日志收集是如何实现的？而运维“织云”这个平台运维工程师又是如何保证平台的稳定性？

梁定安：托管于织云上的业务使用的是 SNG 内部统一的上报通道，将监控数据和日志上报到监控平台的 storm 集群中，监控数据处理集群利用 mongoDB、rabbitMQ 和 Infobright 对大量的流数据进行处理，最终产生监控告警或报表。织云负责提升运维效率，统一监控平台则负责提供监控告警管理。

织云平台本身的可用性，也是严格按照 SNG 可运营规范要求设计的。值得一提的是，织云为了保证具备自动调度能力的模块在关键时刻的可用性，我们特别设计了演习功能，每天都随机抽取演习，随时检验平台核心能力的可靠。

InfoQ：大部人都认为运维不如研发，作为一名运维工程师，你是如何看待这两者之间的关系的？

梁定安：干一行爱一行，既然选择了运维岗位，我认为就应该热爱运维工作并努力在这个岗位上做到卓越，否则就是对个人对工作的不负责。

谈谈运维和研发的区别，任何一款互联网产品，当具备一定的运营规模时，必然会有开发和运维的明确分工。开发专注于产品功能的实现，运维则会从质量、监控、容灾、成本、安全等纬度去支持产品的发展。

认为运维不如研发的看法，往往都是看到了处于初级阶段的运维。我认为优秀的运维团队，和开发团队之间是互惠互利、缺一不可的。以 SNG 社交平台业务运维团队举例，我们会根据业务的规模，提出很多优化建议，如优化 Qzone 的分布架构，制定跨地域、机房的容灾策略，对核心服务抽象成 SET 化管理，建设 SET 调度的智能决策和执行系统；降低业务的运营成本，引入更廉价的运营方案，根据不同的场景，提出用 SSD 硬盘存储替换内存存储，用虚拟化解决长尾服务的成本压力，或者利用 buff 设备提供离线计算能力；我们还建立了大数据分析系统，为移动业务提供运维 SDK，

利用机器学习能力建立外网用户反馈智能分析告警的平台；为了让运维和研发能够更及时的查阅业务运营状态，我们还开发了手机侧的运维工具 MSNG，提供了关键服务的核心指标展示和通用的告警处理工具；为了解决告警量大的问题，我们研发了告警预处理系统，告警根源分析系统等，这些都是运维团队为产品和研发团队输送的超越运维范畴的价值。

也许运维这个岗位不常会在镁光灯下，成为耀眼的明星团队，但是目前运维团队的价值远未被发掘完，我们仍在积极探索未来可以触及的新方向。

查看原文：[梁定安：解密腾讯 SNG 云运维平台“织云”](#)

Etsy 是如何做到每天 50 次以上部署的

作者 João Miranda，译者 马凯 发布于 2014 年 4 月 1 日

[Daniel Schauenberg](#) 在伦敦 QCon 大会上描述了 [Etsy](#) 是如何做到每天 50 次部署的。Etsy 以其 DevOps 和持续交付的实践而闻名。一个完全自动化的部署管道、全面的应用监控和基于 IRC（互联网中继聊天）的协作是能达到这个变更频率的同时又保持最小风险的重要因素。

Etsy 的开发策略是反复围绕着做许多小的、连续的变更。这样做的一个直接后果就是需要每天做很多次部署。用 Daniel Schauenberg 的话来说，在任何时刻每个 Etsy 的开发人员都需要知道下面这个问题的答案：“我现在有足够的信心来部署这个变更吗？”。为了能够每次都可以轻松部署，Etsy 采取了一系列的工具和做法：强制基于 IRC 的沟通、开发者虚拟机、持续集成、一键式部署、全面的应用和系统监控、对于开发和运营团队都采取免责怪的事后检查（post-mortem）和随叫随到的政策。

每个开发人员都拥有自己的通过 [Chef](#) 配置的 [KVM](#)（基于内核的虚拟机）。在线上运营中使用的 [cookbooks](#) 也同样地用在了开发人员的虚拟机中，这意味着每个开发者都有自己完整的 Etsy 栈。任何人都可以通过 [Virtual Madness](#)（一个可以实现整个过程自动化的 Web 应用）来提供一个虚拟机。

在持续集成方面，Daniel 解释了 [Try](#) 是如何成为所有过程的核心。Try 是一个工具，它允许开发人员在 [Jenkins](#)（在 Etsy 中使用的持续集成工具）中 [测试他的代码变更](#)，而不需要先提交到 trunk 中。Try 有助于保持 trunk 干净从而实现可部署，而同时让开发人员能够快速、可靠地测试他们的变更。CI(持续集成)集群必须强大到足以支持 150 名工程师，以及每天超过 [14000 个测试集的运行](#)。[LXC](#)（Linux 容器）会平衡工作的负载，它们还提供了隔离机制，确保不同测试的执行之间不会冲突。

部署管道会经过 Princess，或者工作台，这是一种上线前的环境。Princess 的所有意图和目的就是上线环境，但只有 Etsy 的员工可以访问它。[Deployinator](#) 是由 Etsy 构建并使用的部署工具，提供一键式部署。

配置标志，也被称为功能标志，是部署过程中的一个主要组成部分。[通过其功能 API](#)，Etsy 能够做 [A/B 测试](#)，即完全启用或者禁用某一功能或一个给定功能的变体。

监控是 Etsy 团队建立信心做持续交付的关键。开发人员做自己的功能监控，而且每个人都可以通过仪表盘看到所有的监控图表。Etsy 具有这样一个策略，在默认情况下所有可以被绘制的信息都会被制成[图表](#)。随着时间的推移，指标的数量一直稳步增加，所以 Etsy 建立了 [Kale](#) 来帮助检测异常模式。所有的日志都可以通过 [Supergrep](#) 展现出来，Supergrep 是一个基于 Web 的日志展示工具，这增加了日志的信噪比。

IRC 是贯穿 Etsy 的主要沟通工具，也是 Etsy 协作文化的关键。里面有很多不同的聊天室，每一个都具有特定目的。例如，在聊天室 #warroom 中仅允许中断事故有关的会谈。聊天室是用来协调调查，讨论应对措施和解决方案监控的。与其他的聊天室一样，# warroom 是个鼓励新工程师们潜水的地方，因为这些聊天室被认为是学习的好地方。

每次中断发生，或接近中断的时候，所有人被邀请来做事后检查。事后检查是一个重要的文化活动，如果财务和支持人员需要，他们也可以参与。事后检查是为了成为一个学习的机会，所以它们不应该包含责备。所有事后检查相关的信息被记录在 [Morgue](#) 里面：日期、严重性、IRC 日志、图表和补救措施。Morgue 是 Etsy 为了保存事后检查记录而特别构建的另一种工具。

对于运营、开发、支付和支持部门的员工都采取随叫随到的政策。开发人员通常轮流地在每四个星期里的某一周提供随叫随到的支持。该政策的目的是让每个人都意识到上线产品每天所面临的问题，使他们能够在开发新功能或改进现有流程时考虑到这些问题。

Etsy 拥有约 60 万的月访问量和每月 15 亿页面浏览量。

查看英文原文：[How Etsy Deploys More Than 50 Times a Day](#)

感谢[崔康](#)对本文的审校。

查看原文：[Etsy 是如何做到每天 50 次以上部署的](#)

跟 Monty Taylor 和 Jim Blair 聊 OpenStack 的持续集成与自动化测试

作者 Sai Yang , 译者 杨赛 发布于 2014 年 8 月 5 日

[OpenStack](#) 社区有一个 CI 和自动化测试小组，该小组为 OpenStack 社区的开发者们提供服务，而该服务所用的工具正是他们自己维护的一个 OpenStack 云环境。

对于这样一个囊括了十数个子项目，每月有 300 多位开发者提交代码的复杂项目，普通的 CI 系统是难以处理的。

我们跟该小组的负责人 [Monty Taylor](#) 和 [James Blair](#) 沟通，了解他们在构建和测试过程中所面临的挑战，以及他们是如何解决这些挑战的。

InfoQ：你们的 CI 系统每天处理多少次提交？你预计到 Icehouse 版本发布时会有多少？

(注：本采访完成于 2013 年 11 月，当时距离 Icehouse 发布还有半年。)

Monty：印象中，我们的系统最高处理过每日 400 次提交。这些仅仅是通过测试的部分，实际上我们的测试量要大于这个数字，因为只有通过测试的代码才会进入 CI。

Jim：每次提交被审查之后，我们在实施合并之前会再做一轮测试。

Monty：对于每个被合并的提交，我们都会对其做 8-10 个不同的测试任务。因为测试会在上传的时候和合并之前各做一次，相当于每次变更我们都会跑将近 20 个测试任务。有一段时间我们的系统一天就跑了 10 000 个任务。

从 Grizzly 到 Havana，我们的集成、测试量基本上增加了一倍。基本上每个新版本我们都会增加一倍的量，到 Icehouse 应该也是如此。

InfoQ：你们都跑哪些测试任务？

Jim：首先是代码风格检测。因为我们的协作开发者人数众多，因此代码风格统一是非常重要的，我们需要确保大家都使用同样的编码方式。这是一个很简单的任务，但很重要。

然后是单元测试，仅仅测试被变更的子项目，不考虑跟其他子模块之间有网络交互的情况。我们针对几个不同的平台做测试，包括 2.6、2.7 和 3.3，基本上我们在 CentOS 上跑 2.6，在 Ubuntu 上跑 2.7。

然后是集成测试。我们用 DevStack 将所有的组件安装起来，然后在安装起来的这个单节点云实例上跑不同的模板。不同的模板对不同的模块进行不同的设置，比如使用不同的数据库、不同的消息队列。可以选择的种类很多，不过基本上我们只测试那些常用的，比如 MySQL、PostgreSQL、RabbitMQ 这些。

Monty: 我们最近也在考虑引入 ZeroMQ 的测试。

Jim: 如果社区里认为某个子模块比较重要，使用的人也越来越多，也有更多的人愿意参与到 debug 工作当中，那我们也会将这个模块加入。

InfoQ: 测试任务是由谁来写的？

Monty: 开发者自己写。我们的 QA 团队很小，基本上只关注测试系统本身的工作，不会有太多精力去关注测试任务本身。所以我们要求开发者自己提供单元测试和集成测试。

Jim: 我们最近在讨论的一个话题就是在这方面做更严格的限制，即只有写好了集成测试的变更提交才能够被接受。

Monty: 我们总觉得未经测试的变更就是有问题的。一般来说的确是这样。

Jim: 现在项目发展的这么快，有这么多组件，这里或那里的一个小错误可能就把整个系统搞死。

InfoQ: 性能测试有在做吗？

Jim: 还没有，不过我觉得可能差不多可以启动了。我听说 Boris Pavlovic 正在做一个叫做 Rally 的测试系统，Joe Gordon 则在进行一些可扩展性测试的工作——跟性能测试不太一样，不过关联比较大。这都是我们希望做的事情。

我们的测试显然没有覆盖所有的方面，不过我们最终希望测试所有的东西，当然这需要时间。

在本次发布周期内，我们关注于升级测试。现在我们已经在做一些，不过做的还不够，需要做更多。

InfoQ：在一个实例上运行一个测试任务大概需要多久？

Monty：一般在 20~40 分钟，具体时间长短跟实例的配置有关。

Jim：我们花了很多精力让测试变得并行化。我们构建了一个叫做 Test Repository 的框架，大多数单元测试在这个框架中已经可以并行处理，测试结果出的很快。

Monty：还有 Jim 写的 Zuul，这个工具可以一方面并行的测试成套的变更，同时又保持他们的测试顺序不变。

InfoQ：运行测试用到了多少机器？用于运行测试用例的实例配置是怎样的？

Monty：我们自己是没有机器的。所有的测试都跑在公有云上，有些来自 Rackspace，有些来自 HP，都是赞助的。他们没找我们要钱，而我们需要多少就可以用多少。

Jim：上一个版本周期内，最高的时候我们并行跑了 340 个实例，一个实例就是一个 VM。集成测试一般使用很基础的 VM——8GB 内存，系统是 Ubuntu Precise。我们把这个节点搞起来，然后让 DevStack 在这个 VM 上安装 OpenStack。

Monty：实际情况要比这个复杂，不过大概意思就是这样。我们有一个 nodepool 用来管理这些 VM，通过缓存来预备这些机器。我们需要将 DevStack 需要的依赖等东西都预先下载到本地，这样测试本身就可以离线运转。

Jim：测试跑完之后，我们再销毁这些 VM。实际创建的 VM 数量要比跑成功的测试数量多，因为 Zuul 的随机机制，有些时候它的测试跑到一半的时候才发现还需要一些其他东西，于是测试跑不下去了，我们会干掉这个 VM，起一个新的。一个大致的比例是，如果一天跑 10 000 个任务，那么启动的 VM 数量差不多在 100 000 的量级。

InfoQ: 可以认为用于 OpenStack 的 Zuul 模式是 nvie git 分支模式的一个改进吗？感觉 Zuul 似乎不适合分支过多的情况。

Monty: 实际上我们是不采用 nvie git 分支模式的，因为我们用了 Gerrit，所以我们的代码提交模式跟 Linux 内核的模式更像：人们在邮件里交换补丁。我们的做法不是建立很多的分支然后做合并，而是让每一个变更形成一个虚拟的私有分支。相对于将每一次变更生成一个新的 commit 并增添至分支的顶端的做法，我们的做法是：在之前的一次修改之上再进行修改。我们的测试针对每一个独立的 commit，而不是针对一个分支。

每一个开发者可以建立本地的分支，这些分支是私有的，没有什么发布机制。我并不知道 Jim 的笔记本上的分支是什么样的。我自己用 git 的方式比较奇葩，我不用分支，而是每次在我的 master 上重置 ref——这是个非主流的用法，git 新手最好还是不要这么尝试。

所以，OpenStack 的 git 补丁流程其实是基于 Gerrit 的。

Jim: 另外，我们需要确保审查人员审查的对象是每一个 commit（而不是分支）。理想状态下，每一个进入项目的 commit 都被人仔细的检查过。分支的话就会比较混乱。把每一个 commit 把关好，把好的 commit 合并，是比较精细的做法。

InfoQ: 除了 Zuul 之外，你还提到了在 Jenkins 上使用 Gearman 来提高可扩展性，使用 Logstash 做 debug，还有你上面提到的 Test Repository 将测试输出自动发给 committer。目前的反馈机制是如何运转的？理想的情况是怎样的？

Monty: 反馈机制整体来说是越来越好的。你的问题涉及到几个方面。有关用 Gearman 来提高 Jenkins 的可扩展性这一点，首先 Jenkins 本身的设计是针对一个 master 的情况，让它支持多个节点是通过 hack 的方式来完成的。我们一开始的用法是跑一个 Jenkins master 和若干个 slave，并行跑的测试任务数量要比正常的 Jenkins 用法要多很多。Jenkins 在设计当中涉及到很多全局锁，所以要像我们这样用起来，会遇到很多可扩展性的问题。

Jim: 因为 Jenkins 在设计的时候根本没考虑过我们这样的用法。

Monty: 所以我们就写了 Gearman 插件，这个插件的作用是让 Jenkins 将所有任务注册为潜在的 Gearman 任务，标记在 Gearman 服务器上。这样一来我们就可以针对一组测试任务建立多个 Jenkins master，让 Gearman 来做任

务分发，如果一个 Jenkins master 开始遇到瓶颈，我们就让 Gearman 把任务分发到下一个 Jenkins master 上。

Jim: 一般来说，一个 Jenkins master 带 100 个 slave 之后就会遇到问题。我们要同时跑 340 个任务，那就需要 3.4 个 Jenkins master 来处理。

Monty: Logstash 集群是个很有意思的东西。每一次 DevStack 安装的是整个的云环境，然后针对这个小环境跑测试。仅仅是安装的过程就会制造很多日志，包括 Nova、Glance 等等。如果遇到问题，开发者根本无从下手去 debug，能够依赖的只有日志。所以，我们把所有的日志丢到一个很大的 Logstash 集群当中，这个集群通过 elastic search 的方式给所有的 log 建索引。这样，开发者就可以进去查看日志，了解到底发生了什么问题。这里面的 Elastic Recheck 是 Joe Gordon、Sean Dague 和 Clark Boylan 写的。

Joe: 那个图表功能是我写的。

Monty: 比如我们发现有一个任务导致测试跑失败了，我们会在 LogStash 上运行脚本，来检测这是否是我们之前见到过的错误类型。如果有匹配，我们在邮件通知里将之前的 bug 报告附上，这样会帮助开发者更快的定位问题。

Jim: 这其实是很酷，也很独特的。世界上像这种规模的项目是很少的，这种规模的测试、这种规模的日志，开发者很少能够在其他项目获取到。云平台这样的项目，开发者在自己的机器上是很难去发现代码可能会引起的问题的，因为很多问题都是要跑很多次不同的测试才能抓到——而我们的测试平台可以做到这一点！下一个发布周期内，我们会尝试让问题识别变得更加自动化，将变更和行为的特征更多的抽取出来，帮助开发者更快的定位问题。

InfoQ: 你们做的这一大堆自动化测试的工作，感觉最难的地方是在哪里？

Monty: 开发者很多，代码很多，测试需求量每 6 个月都会增长一倍。面对 commit 数量如此众多、快速增长的情况，我们需要提前预见到可能发生的问题，做好准备——因为如果真的遇到了问题，那么那个时候再去开发系统来解决问题就来不及了。自动化解决的问题不是今天的问题，而是三个月之后的问题。

正因为所有的测试都在我们这里，我们就必须确保这个系统一直能够正常运转。你的测试一天跑 10 000 次，万一系统出了问题，给开发者发邮件说你的代码有错（而实际上根本不是他们代码出了错，是系统本身出了错），那就会很糟糕。误报比不报更糟糕，所以自动化必须做的非常靠谱。

还有就是，我们总是会遇到网络中断的问题——基本上我们有一半的时间都用来处理这个问题。所有的网站都会连不上：平时你自己去刷网页是感觉不到的，但如果你一天跑 10 000 次自动化测试呢？如果 Github 平均有 1% 的时间是不可用的，你作为用户去刷页面没打开，重试一次就好；而我的测试系统每天从 Github 做 10 000 次抓取，1% 的不可用就相当于 100 次失败。

由于我们在跑的这个系统，我们也成了 RackSpace 和 HP 云的性能监控器。很多时候我们发现有一个问题，就去问他们的运维：“你们这个数据中心是不是网络出问题了？”然后他们会说：“对啊！我们也刚刚发现！”

Jim: Rackspace 和 HP 云都是基于 OpenStack 的系统，所以我们的测试系统是在 OpenStack 上运行、为 OpenStack 做测试。用自己测试自己的代码，同时又测试自己的运行状态，这是个很酷的事情。

受访者简介

Monty Taylor 是 HP 杰出工程师，OpenStack 技术委员会成员、OpenStack 基金会个人董事。他带领 OpenStack 基础架构项目、Ironic 项目和 TripleO 项目。

Jim Blair 现在是 OpenStack 基础软件组的核心开发者，也是 OpenStack CI 项目的核心开发者。他也是 OpenStack 技术委员会成员，OpenStack 基础架构项目的技术领导者。他目前任职于 OpenStack 基金会。

查看英文原文：[Monty Taylor and Jim Blair on CI and Test Automation at OpenStack](#)

查看原文：[跟 Monty Taylor 和 Jim Blair 聊 OpenStack 的持续集成与自动化测试](#)

Amazon DynamoDB 在游戏开发中的应用

作者 Nate Wiger 发布于 2014 年 10 月 9 日

Amazon DynamoDB 正迅速成为世界上发展势头最强劲的游戏数据库。《水果忍者》（由 Halfbrick 工作室开发）、《战斗营地》（由 PennyPop 开发）等游戏都充分利用 Amazon DynamoDB 的一键式扩展性功能，支撑游戏高速的发展，为全球数百万玩家提供服务。Amazon DynamoDB 还得到包括 Supervillain 工作室在内的众多开发人员的赞赏，该工作室的知名作品包括《塔炮战争》与《特隆：进化》。

在今天的文章中，大家将了解 Amazon DynamoDB 如何帮助大家为自己的移动游戏快速建立起可靠且极具可扩展性的数据库层。我们将分步剖析设计示例并了解如何以每天不足一杯咖啡钱的成本为游戏提供弹性资源支持。我们还将模拟一家快速发展的客户，观察 Amazon DynamoDB 如何在时间与成本效率的前提下将玩家支持规模扩展至数百万之巨。

数据库层的重要性

在为规模化应用程序设计架构时，一大关键性因素在于数据库层。这一点对于游戏尤为重要，毕竟属于写入密集型应用。游戏数据会随着玩家收集道具、击败敌人、获取金币、角色升级以及完成成就而不断更新。每一个事件都必须被写入到数据库层，从而保证内容不会丢失。可以想见，一旦进度损坏玩家将变得极为暴躁。

游戏与 Web 应用开发人员通常会使用 MySQL 等开源关系型数据库作为自己的数据库层，这是因为此类方案更为人们所熟悉。遗憾的是，以 MySQL 为代表的关系型技术方案在开发之初更多考虑到的是高强度读取工作负载，而这种机制并不太适合游戏、社交媒体应用以及图片分享站点。有鉴于此，NoSQL 解决方案应运而生，它利用强大的写入数据吞吐能力与横向扩展能力替代了传统关系型数据库在查询灵活性领域的优势。

Amazon DynamoDB 适合游戏开发人员需求的三个理由

AMAZON 包揽运营任务

开发游戏本身就很累人，对吧？Amazon DynamoDB 是一项托管服务，其中包含全方位运营支持以及多数据中心高可用性。大家用不着再为软件安装、硬件故障处理或者性

能表现调整而烦心。只需调用单一 API，大家就能对 Amazon DynamoDB 进行动态缩放。

每一个 Amazon DynamoDB 数据库表都与数据吞吐能力密切相关。大家可以将每秒写入操作设定为 1000 次，而 Amazon DynamoDB 会处理全部后台数据库调整工作。根据用户需求的变化，大家可以更新该容量，Amazon DynamoDB 则会依要求完成资源重新分配。这种弹性能力对于游戏开发者帮助巨大：当游戏推出之后，大家可能需要在短时间内将玩家支持规模由数千位增加到数百万位。同样重要的是，大家可以快速调低资源规模——这种调整对于 MySQL 数据库来说颇具挑战。

无论游戏玩家规模如何变动，性能表现都将保持稳定

Amazon DynamoDB 可在任意规模水平下保持可预测、低延迟性能表现。如果大家的游戏对于延迟较为敏感、并且需要面对数百万玩家，那么这种特性将变得至关重要。使用 Amazon DynamoDB，大家不必再为性能调整浪费任何精力。

在 Amazon DynamoDB 中保存游戏数据

我们不妨想象一下，大家希望创建一款角色扮演游戏。游戏设计遵循常见机制：与怪兽战斗、收集战利品并进行角色升级。在这种情况下，各位显然需要保存用户的当前进度，这时我们应该为每位玩家创建一个键-值对配置文件，其中包括现有道具、角色等级以及赚得金币的数量。大家的数据结构可能如下所示：

```
{  
    player_id: 3612458,  
    name: "Gunndor",  
    class: "thief",  
    gold: 47950,  
    level: 24,  
    hp_max: 320,  
    hp_current: 292,  
    mp_max: 180,  
    mp_current: 180,  
    xp: 582180,  
    num_plays: 538,  
    last_play_date: "2014-06-30T16:27:39.921Z"  
}
```

在这个示例中，player_id 应该是个独特的值。将其与 MySQL 等关系型数据库相映射非常简单：为每个键创建一个列。这种方式当然可行，但对结构中的每一列进行解析及检索将给数据库带来沉重负担，而且主键（即 player_id）几乎每时每刻都要接收查询请求。很明显，大家不太可能利用玩家的记录点位置或者经验值进行记录查询。

好的，下面来看将其映射至 Amazon DynamoDB 会是什么。在 Amazon DynamoDB 当中，我们只需要定义需要进行检索的列即可。在这种情况下，我们将创建出单一散列键作为主键，并借由 player_id 的独特性利用它实现记录查询。我们会定义一套名为 player_profiles 的表，并为 player_id 设定散列键。下面来看 Python 语言编写的示例：

```
player_profiles = Table.create('player_profiles', schema=[  
    HashKey('player_id', data_type=STRING)  
, throughput={  
    'read': 5,  
    'write': 5,  
},
```

我们已经创建了一套包含 5 个读取容量单位与 5 个写入容量单位的表，其被包含在 AWS Free Usage Tier 当中。大家也可以利用 AWS 管理控制台创建这套表。如果不知道该如何操作，请[点击此处](#)查看指导信息。

该表创建完成之后，我们的配置文件将如下所示：

```
player_profiles.put_item(data={  
    'player_id': '3612458',  
    'name': 'Gunnidor',  
    'class': 'thief',  
    ...  
})  
  
profile = player_profiles.get_item(player_id='3612458')
```

这只是 Python 环境下的示例，大家也可以使用任何 AWS SDK 以 put/get Amazon DynamoDB 中的键。

主键值

关系型数据库使用自动递增的整数，其典型会被作为主键。在规模化场景下，自动递增的主键往往成为性能瓶颈，因此 Amazon DynamoDB 等 NoSQL 并不会将其进行保

存。那么我们该如何生成独特的 player_id 值呢？我们使用 UUID，由于 UUID 的内容彼此不同、因此不同客户端总会生成独立的相应数值。UUID 能够为我们生成相当长的字符串，例如 a8098c1a-f86e-11da-bd1a-00112444be1e。UUID 与 Amazon DynamoDB 匹配效果极好，这是因为它有助于确保主键的随机分布与访问、从而让 Amazon DynamoDB 始终拥有良好的性能表现。

生成 UUID 非常简单：

```
player_id = uuid.uuid1()

player_profiles.put_item(data={

    'player_id': player_id,
    'name': 'Gunndor',
    'class': 'thief',
    ...
})
```

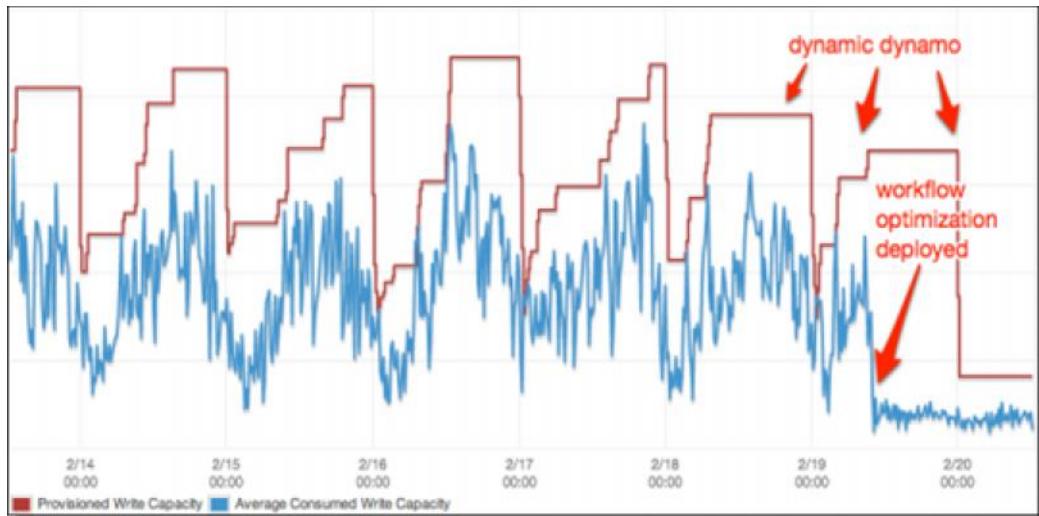
原子递增

除了 put 与 get，Amazon DynamoDB 还支持原子递增。这种机制在值发生变化之后的更新流程中非常实用，因为来自应用程序的多项请求不会出现冲突——正是此类状况引发了在线游戏中的大部分进度丢失问题。如果玩家拾取到 100 金币，大家可以直接要求 Amazon DynamoDB 自动将 100 金币增量以原子化方式进行添加，而无需经历获取记录、添加金币再将其返回 Amazon DynamoDB 的过程。

选择合适的容量水平

Amazon DynamoDB 允许大家指定自己所需要的数据吞吐能力容量。但如果大家不清楚这一水平该怎么办？当大家开始游戏开发时，先创建自己的第一套表（例如 5 个写入容量单位与 10 个读取容量单位）。随着流量的增长，我们可以在 Amazon DynamoDB 控制台中利用 CloudWatch 图形监控使用情况并作出调整。

[Dynamic DynamoDB](#) 是另一款实用性工具，这是一套开源库、旨在帮助我们对表容量进行自动扩展。我们的客户之一 [tadaa](#) 公司利用 Dynamic DynamoDB 在流量水平下降时及时调整资源、从而控制成本支出。



我们真能用每天一杯咖啡的成本为数千玩家提供支持吗

是的！我们利用保存游戏这个例子来估算成本。假设游戏每个月平均玩家数量为 10 万名，其中大部分玩家并不会在同一时间登录游戏，因此我们粗略估算其中的十分之一将同时在线。另外，我们假设这 1 万名玩家每一分钟保存一次游戏，而每位玩家的进度数据记录不足 1KB。最后，我们假设每位玩家在游戏过程中每一分钟需要从数据库中读取一次游戏状态。由于一分钟里有 60 秒，相当于我们的 Amazon DynamoDB 表每秒必须能够支持 167 次写入与读取操作（10 000 除以 60）。大多数企业都会保留一部分缓冲容量，因此我们将每秒写入与读取接纳能力提升到 200 次，而总存储空间则设定为 50GB。

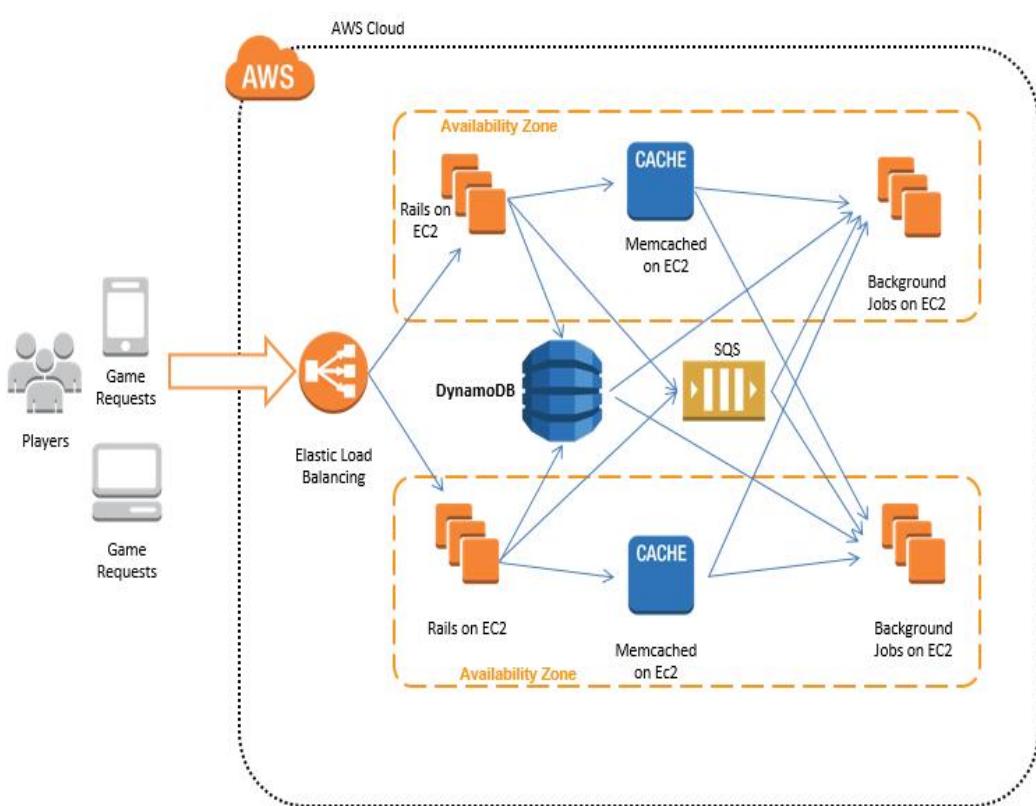
根据目前 US-EAST-1 区域的资源计费标准，这样的资源每天只会带来 4.16 美元的开支！也就是说，每天一杯咖啡的价钱完全可以为 10 万名玩家提供支持（当然，这里指的是花式意大利浓缩那种相对较贵的咖啡类型）。作为起步，大家也可以先免费使用 AWS Free Usage Tier 与 Amazon DynamoDB 相配合。

客户示例：《战斗营地》

作为一款由 PennyPop 公司开发的高人气手机游戏，《战斗营地》利用 Amazon DynamoDB 作为其首选数据存储机制。《战斗营地》截至目前的下载总量已经超过 1000 万次，而且在超过四十个国家的应用程序商店中占据下载榜的前百名位置。在比较了其它几套 NoSQL 选项之后，PennyPop 公司的技术人员选择了 Amazon DynamoDB，因为他们希望能够将精力集中在应用程序编程、而非服务器维护与扩展方面。

《战斗营地》的开发人员首先下载了 [fake dynamo](#)，一款开源客户端，来进行本地开发。（Amazon 还发布并支持一款本地 Dynamo 客户端。）他们构建起自己的对象关系映射 (ORM) 以使其能够与 Ruby on Rails 协作。这套 ORM 方案对 DynamoDB 的功能进行了大幅度简化，因为开发人员们只需要在应用程序层中用到键-值检索。该 ORM 将值作为 JSON 对象进行保存——所谓 JSON 对象，也就是经过压缩的 base64 字符串。这种机制允许他们将 JSON 对象压缩至原始体积的不足十分之一。大多数 Web 应用程序数据库查询操作可以通过获取与保存简化实现削减，而整个迁移工作也在数周后彻底完成了——在此期间他们还构建起自己的定制 ORM。

下图所示为《战斗营地》游戏如何将 Amazon DynamoDB 整合至自己的架构当中。



结果如何？极大节约了时间与成本。正如 PennyPop 公司联合创始人 Charles Ju 的解释：

Amazon DynamoDB 的成本节约效果一方面借由效率与易用性实现，同时也体现在了维护成本的显著缩减身上。构建、维护以及拆分以数据为中心的大型实时项目一直非常困难，项目的创建与维护也一直要求大量技术人员的参与。然而现在我们在为数百万玩家提供满意的游戏体验的同时，仍然只拥有两位服务器工程师。我们的规模比已知的任何一家 MMORPG 厂商都更具精简特性。

他们还发现，DynamoDB 在其它规模更大的场景中同样运作良好、甚至能够与 MapReduce 分析协议顺利对接，这要归功于其出色的扩展灵活性。他们完全可以构建内部 MapReduce 协议，进而以并行方式执行数据分析。

最后一项：分析

除了实时游戏服务之外，Amazon DynamoDB 还集成了一系列其它 AWS 服务，其中包括 Amazon Elastic MapReduce（简称 EMR）以及 Amazon Redshift。Amazon EMR 与 Amazon Redshift 都能够直接从 Amazon DynamoDB 当中加载分析数据，从而简化分析工作流的构建方式。如果大家想了解更多在 AWS 上对游戏内容加以分析的信息，请在评论栏中发表意见、我们将在未来的文章中专门探讨这个话题。

本文原载于 [AWS 博客](#)。

查看原文：[Amazon DynamoDB 在游戏开发中的应用](#)