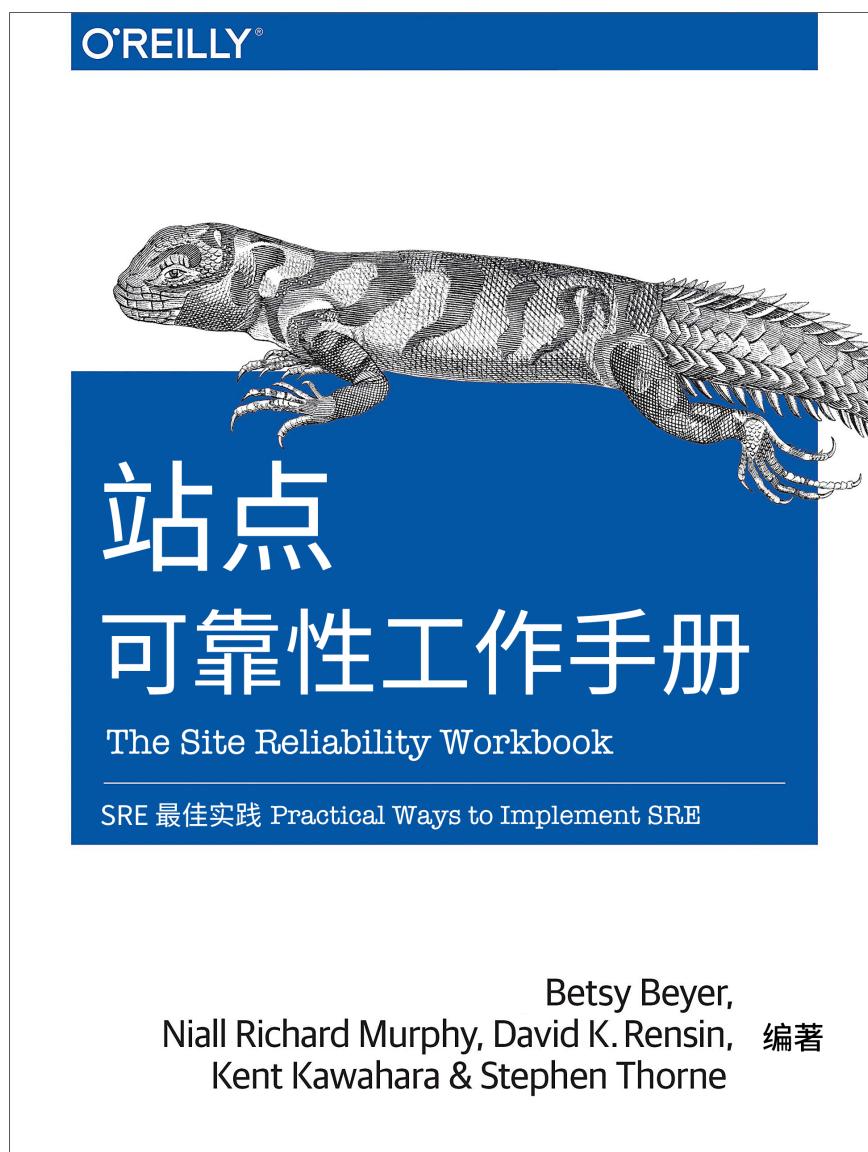


# Table of Contents

Introduction	1.1
前言 I	1.2
前言 II	1.3
序言	1.4
第1章 SRE和DevOps的关系	1.5
第一部分 基础	1.6
第2章 实施SLO	1.6.1
第3章 SLO工程案例研究	1.6.2
第4章 监控	1.6.3
第5章 基于SLO发出警报	1.6.4
第6章 消除琐事工作	1.6.5
第7章 简单化	1.6.6
第二部分 实践	1.7
第8章 值班	1.7.1
第9章 事件响应	1.7.2
第10章 事后总结文化: 从失败中学习	1.7.3
第11章 管理负载	1.7.4
第12章 介绍非抽象大型系统设计	1.7.5
第13章 数据处理管道	1.7.6
第14章 配置设计和最佳实践	1.7.7
第15章 配置细节	1.7.8
第16章 金丝雀发布	1.7.9
第三部分 流程	1.8
第17章 识别过载并从中恢复	1.8.1
第18章 SRE参与模型	1.8.2
第19章 SRE-超越自己	1.8.3
第20章 SRE团队生命周期	1.8.4
第21章 SRE中的组织变革管理	1.8.5
总结	1.9
附录A SLO文档示例	1.10
附录B 错误预算政策示例	1.11
附录C 事后分析的结果	1.12
关于编者	1.13

# The Site Reliability Workbook 站点可靠性工作手册 中文版



Betsy Beyer,  
Niall Richard Murphy, David K. Rensin, 编著  
Kent Kawahara & Stephen Thorne

[前言 I](#)

[前言 II](#)

[序言](#)

[第1章 SRE和DevOps的关系](#)

[第一部分 基础](#)

- [• 第2章 实施SLO](#)
- [• 第3章 SLO工程案例研究](#)
- [• 第4章 监控](#)
- [• 第5章 基于SLO发出警报](#)
- [• 第6章 消除琐事工作](#)
- [• 第7章 简单化](#)

## 第二部分 实践

- 第8章 值班
- 第9章 事件响应
- 第10章 事后总结文化: 从失败中学习
- 第11章 管理负载
- 第12章 介绍非抽象大型系统设计
- 第13章 数据处理管道
- 第14章 配置设计和最佳实践
- 第15章 配置细节
- 第16章 金丝雀发布

## 第三部分 流程

- 第17章 识别过载并从中恢复
- 第18章 SRE参与模型
- 第19章 SRE-超越自己
- 第20章 SRE团队生命周期
- 第21章 SRE中的组织变革管理

## 总结

附录A SLO文档示例

附录B 错误预算政策示例

附录C 事后分析的结果

关于编者

## 前言 |

**Mark Burgess**

介绍了O'Reilly的第一本SRE书籍后，我很荣幸被邀请回到续集。在这本书中，写作团队将抛弃第一本书的历史，为自己说话，并向更广泛的受众延伸，提供直接的经验，案例研究和非正式的指导。广泛的主题对于IT领域的任何人来说都是熟悉的，可能会被重新标记和重新确定优先级，并且具有现代的业务意识。在这里，代替技术说明，我们提供了面向用户的服务及其承诺或目标。我们看到人机系统起源于不断发展的业务，是其宗旨所固有的，而不是外国陨石影响着毫无戒心的原始基础设施。人机各部分的合作是重点。确实，这本书可以总结如下：

- 承诺明确设定服务目标，期望和水平。
- 根据指标和预算限制，持续评估这些承诺。
- 快速做出反应，以兑现并兑现承诺，随时待命，并保持自主权，以避开新的看门人。

(对所有利益相关者)可靠地兑现承诺取决于其所有依赖，意图和所涉人员生活的稳定性(例如，请参阅[思考 承诺](#))。值得注意的是，人机系统的人为因素只会随着规模增长的威胁而增长：事实证明，毕竟自动化并不能消灭人。相反，它从各个想法的产生到代表全球用户群的大规模部署不断挑战着我们，在各个层面上重新确立人类的需求。

教授这些课程本身就是一项服务挑战-就像任何服务一样，来之不易的知识是一个反复的过程。我们通过质疑，尝试，失败，排练和完善它们来总结自己的经验。书中有很多材料值得思考和适应，所以让我们开始吧。

## 前言II

*Andrew Clay Shafer*

当我发现人们正在写第二本SRE书时，我伸出手问我是否可以写几句话。第一本SRE书中的原则与我一直想像的DevOps非常吻合，并且即使在Google之外并非100%适用时，这些实践也很有见地。首次阅读第一本SRE书中的原则之后---[拥抱风险\(第3章\)](#), [服务级别目标\(第4章\)](#), 和[减少琐事\(第5章\)](#)---我想在屋顶上大声喊叫。“拥抱风险”之所以引起共鸣，是因为我多次使用类似的语言来帮助传统组织推动变革。[第6章](#)一直是一个隐含的DevOps目标，既可以让人们有更多的时间进行更有创意的高阶工作，又可以使他们变得更加人性化。但是我真的爱上了“服务水平目标(SLO)”。我喜欢这种语言和流程在运维注意事项和提供新功能之间建立了无懈可击的契约。SRE，SWE(软件工程师)和企业都同意，该服务必须具有最高价值，并且SRE解决方案量化了目标，以驱动行动和优先事项。该解决方案将服务级别作为目标，当您低于目标时，优先考虑可靠性而不是功能-消除了运维和开发人员之间的经典冲突。这是一个简单而优雅的框架，它可以通过解决问题来解决问题。自那以后，我将这三章作为作业分配给几乎我遇到的每个人。他们就是那么好。每个人都应该知道。告诉你所有的朋友。我已经告诉了我所有的人。

我职业生涯的最后十年一直专注于帮助人们使用更好的工具和流程来交付软件。有时人们会说我为发明DevOps做出了贡献，但我只是可以从许多不同的组织和项目中借鉴和窃取成功的模式。当人们说“DevOps”的发明人可以是任何人，但特别是我发明的，我感到很尴尬。除了好奇心外，我不认为自己是专家。我理想化的DevOps总是仿制我可以从朋友那里提取或推断出的所有信息，而我的朋友恰巧正在建立互联网。我有进行幕后访问的特权适用于部署和运维世界上最令人难以置信的基础架构和应用程序的代表示例的人员。DevOps象征着通过互联网快速交付高可用性软件所需的紧急和现有优化的方面。从物理介质上交付的软件到服务即交付的软件的转变迫使工具和流程不断发展。这种演变提高了运维对价值链的贡献。如果系统关闭，则该软件没有价值。好消息是，您不必等待下一个封装盒子就可以更换软件。对于某些人来说，这也是个坏消息。我只是有机会和观点阐明了新方法最成功的方式来吸引听众。

2008年，在像现在这样使用DevOps一词之前，我曾经历过互联网泡沫破裂，研究生院的发展，以及作为开发人员的几笔由风险投资资助的过山车之旅-每天都在Google中寻找答案。我全职从事Puppet的工作，而被自动化改造IT组织的潜力所吸引。Puppet推动我解决了运维领域的问题。当时，Google使用Puppet来管理其公司Linux和OS X工作站，其规模足以推动Puppet服务器的功能。我们与Google保持着良好的合作关系，但是Google出于政策原因将其内部运营的某些细节保密。我知道这一点是因为我天生好奇，并一直在寻求更多信息。我一直都知道Google必须拥有出色的内部工具和流程，但是这些工具和流程的含义并不总是很明显。最终，我接受了问有关Borg的深层问题可能意味着当前的对话进行得并不顺利的事实。我很想知道有关Google如何完成所有工作的信息，但是当时根本不允许这样做。2008年的意义还包括第一次O'Reilly Velocity会议以及我遇见Patrick Debois的那一年。“DevOps”还是一个方法论，但快了。时间到了。世界已经准备好了。DevOps象征着一种新的方式，一种更好的方式。如果那时Site Reliability Engineering发布

了，那么我相信组成的社区将团结起来悬挂“消除琐事”的旗帜，而DevOps一词可能从未存在过。尽管事实并非如此，但我知道第一本SRE本书亲自提高了我对可能性的理解，并且我已经仅凭SRE原理就为许多其他人提供了帮助。

在DevOps运动的早期，我们有意识地避免编纂惯例，因为一切都在迅速发展，我们不想为DevOps的发展设定限制。另外，我们明确地不希望任何人“把控”DevOps。当我在2010年撰写有关DevOps的文章时，我提出了三个不同的观点。首先，开发人员和运维部门可以并且应该一起工作。其次，系统管理将越来越像软件开发。最后，与全球实践社区共享可以加速并增加我们的集体能力。大约在同一时间，我的朋友达蒙·爱德华(Damon Edward)和约翰·威利斯(John Willis)为文化，自动化，度量和共享创建了缩写CAMS。Jez Humble later通过添加精益持续改进，将该首字母缩写词扩展为CALMS。这些词在上下文中可能意味着什么，这本书应该是一本完整的书，但是我在那里提到它们是因为*Site Reliability Engineering*明确引用了Culture，Automation，Metrics和Sharing以及有关Google不断改进之路的轶事。通过出版第一本SRE书籍，Google与全球社区分享了他们的原则和实践。现在，我将DevOps定义为“优化人员绩效以及使用软件以及人员来实践运维软件”。我不想在任何人的嘴上说些什么，但这似乎也是描述SRE的好方法。

最终，当我看到DevOps以及Google的SRE时，从理论上和实践上我都知道它们是最先进的实现之一。良好的IT运维始终取决于良好的工程设计，而使用软件解决运维问题始终是DevOps的核心。站点可靠性工程使工程方面更加明确。当我听到有人说“SRE vs DevOps”时，我感到非常畏缩。对我来说，它们在时间和空间上是密不可分的，就像标签所描述的那样，它们通过软件为现代基础设施提供了社会技术系统。我认为DevOps是一组宽松的通用原则，而SRE是高级的显式实现。一个类似的比喻是敏捷与极限编程(XP)之间的关系。的确，XP是敏捷的，可以说是敏捷中最好的，但是并非所有组织都有能力或愿意采用XP。

有人说“软件正在吞噬世界”，我知道为什么会这样，但是仅“软件”并不是正确的框架。如果没有与高速网络连接的计算硬件的普遍存在，我们认为“软件”的大部分功能将无法实现。这是不可否认的真理。我认为在这次有关技术的对话中，许多人未察觉到的是人类。技术的存在是由于人类的存在，并希望人类能够“为之”，但是，如果您更深入地了解，您还会意识到我们所依赖的软件(并且可能认为是理所当然的)在很大程度上取决于人类。我们依靠软件，但是软件也依靠我们。这是一个不完美的硬件的互连系统，软件和人类依靠自己来构建未来。可靠性正在蚕食世界。但是，可靠性不仅与技术有关，而且与人有关。人与技术形成一个单一的技术社会系统。让Google与其他行业共享SRE的一个不错的特色是，关于大规模运行所有流程工作的任何借口都是无效的。Google为可靠性和规模设定了最高标准。关于为什么有人不能直接采用Google SRE做法可能存在有效的论点，但指导原则仍应适用。当我着眼于构建未来的可能性以及利用软件转变人类体验的雄心壮志时，我看到了许多雄心勃勃的项目，这些项目实际上将所有内容都连接到了互联网。我的数学认为，成功的项目将发现自己正在摄取难以置信的数据并将其编入索引。很少(如果有的话)会超过Google的规模，但有些会与Google创办SRE时的规模相同，并且会需要解决同样的可靠性问题。我认为，在这些情况下，采用看起来像SRE的工具和过程不是可选的，而是必然的-尽管由于SRE的原理和实践适用于各种规模，所以不必等待那场危机。

SRE通常是按照Google的运作方式来构架的，但这错过了更大的前景：SRE实际上支持软件工程，但也可以改变体系结构，安全性，治理和合规性。当我们利用SRE专注于提供服务平台时，所有其他这些注意事项都将得到一流的重视，但是这种情况发生的地点和方式可能完全不同。就像SRE(并希望DevOps)将越来越多的

负担转移到软件工程上一样，现代体系结构和安全性实践也从幻灯片，检查表开始发展，并希望通过运行代码实现正确的行为。在不重新审视其他方面的情况下采用SRE原则和实践的组织将失去巨大的改进机会，并且如果没有将相关责任人转化为盟友，他们也可能会遇到内部抵制。

我总是很喜欢学习。我直接阅读了第一本SRE书中的每个字。我喜欢这种语言。我喜欢轶事。我喜欢更多地了解Google的看法。但是对我来说，问题始终是：“我将改变哪种行为？”

学习不是在收集信息。学习正在改变行为。在某些学科中，这很容易确定甚至量化。当您可以播放歌曲时，您已经学会了播放新歌曲。就好比国际象棋比赛中与实力较强的玩家对抗会更好。像DevOps一样，站点可靠性工程不仅应该更改标题，还应该进行明确的行为更改，重点放在结果和明显的可靠性上。《网站可靠性工作手册》承诺从Google列举的Google原则和实践向更多上下文行动和行为迈进。网站的可靠性适合所有人，但可靠性并非来自于读书。这是拥抱风险并消除琐事的方法。

## 序言

当我们撰写 *Site Reliability Engineering* 时，我们的目标是解释 Google 的生产工程和运营的理念和原则。这本书是我们试图与其他计算机世界分享我们团队的最佳实践和经验教训的尝试。我们假设 SRE 本书可能吸引少量的工程师从事注重可靠性的大型工作，并且内容的数量和重点都将限制本书的吸引力。

事实证明，我们在这两个方面都被误解了。

令我们感到惊讶和高兴的是，SRE 本书在发行后令人振奋的时期是计算机上的畅销书，不仅出售或下载很多，而且还很畅销。它正在被很多人阅读。我们收到了来自世界各地有关本书，团队，实践和成果的问题。我们被要求谈论各章节，方法和事件。我们发现自己处于意外状况，因为我们处于非周期状态，因此不得不拒绝外部请求。

像大多数成功的灾难一样，SRE 本书创造了一个机会去选择是可以用人工来应对（“雇佣更多人！进行更多的口语交流！”）还是采用更具扩展性的方式。作为 SRE，我们会偏向于后一种方法，这会让少数读者感到惊讶。我们决定编写第二本 SRE 书-扩展了我们最常被要求谈论的内容，并解决了读者对第一本书最常见的问题。

关于第一本 SRE 书，我们收到了许多不同的问题，要求和意见，其中有两个主题对我们特别有趣：如果不加以解决，它们是将 SRE 的课程用于生产的障碍。这些主题通俗地概括为：

- 原理很有趣，但是如何在我的项目/团队/公司中将其付诸实践？
- SRE 的方法对我不起作用；这仅在 Google 的文化中可行，并且仅在 Google 的规模上才有意义。

这本第二本 SRE 书籍的目的是(a)在第一卷中概述的原则上添加更多实施细节，以及(b)消除仅在“Google 规模”或“Google 文化”下可以实施 SRE 的想法。

该卷是以前工作的“伴侣”，而不是新版本。这两本书应该合为一体。如果您已经熟悉本书的前身，您将会从本书中获得最大收益。第一本 SRE 书籍 [免费在线提供](#)。

通过设计，本书的结构大致遵循第一卷的结构。我们希望您能够一并阅读各章。本卷的每一章都假定您熟悉上一本书中的相应内容。我们的目标是让您随时随地在原理和实践之间来回切换。这样，您可以将两本书都用来参考正在进行的工作。

接下来，谈谈精神：我们从一些读者那里得知，在描述 Google 改善运营的过程时，我们过于专注于“只是我们”。一些读者建议说，我们也远离 Google 之外的世界的实际情况，并且没有用 DevOps 的原理解决 [我们的思想的相互作用](#)。在本书中，我们试图引起人们的注意，这是一种完全公平的批评。

但是，我们确实认为 SRE 的高度自觉性有助于其作为一门学科的有用性。对我们来说，这是一个特点，而不是错误。我们不主张 SRE 是构建和运行高度可靠的系统的唯一方法（甚至通常是最佳方法）。这是对我们最成功的方式。

我们还将花一些时间谈论 SRE 和 DevOps 之间的关系。要牢记的重要一点是它们没有冲突。

我们想预先确认这本书一定是不完整的。即使在Google的内部，SRE学科也是一个广阔的领域，并且由于它在Google以外的广泛实践，它的发展速度甚至更快。我们没有将内容广泛和肤浅，而是集中讨论了第一本书中最需要的实施细节。

最后，该书及其前身不是福音。请不要那样对待他们。即使经过了这么多年，我们仍在寻找条件和情况，使我们调整(或在某些情况下替代)以前坚定的信念。SRE既是一门学科，又是一段旅程。

我们希望您喜欢在这些页面中阅读的内容，并找到有用的书。组装它是一种爱的劳动。我们很高兴，有一个不断壮大的SRE专业人士社区，我们可以与他们一起学习和改进。

与往常一样，非常感谢您的直接反馈。每当您做出贡献时，它都会教给我们一些有价值的东西。

## 如何阅读这本书

本书是Google第一本书*Site Reliability Engineering*的配套书。为了充分利用此书，我们建议您阅读或参考第一本SRE书籍(可通过[google.com/sre](http://google.com/sre))免费在线阅读。这两部作品在以下方面互为补充：

- 以前的工作是对原理和哲学的介绍。本册集中于如何应用这些原理。(在某些领域，尤其是配置管理和金丝雀发布，我们还涵盖了一些新领域，为其他主题的实际处理提供了背景。)
- SRE仅着重于Google如何实践SRE。这项工作包括许多其他公司的观点-从传统企业(包括The Home Depot和《纽约时报》)到数字本地人(Evernote，Spotify等)。
- SRE没有直接涉及到更大的运营社区，尤其是DevOps，而这本书直接谈到了SRE和DevOps如何相互联系。

该书假定您将在此书与其前身之间切换。例如，您可以阅读第一本书中的第4章，“服务级别目标”，然后在本卷中阅读其实现补充(第2章)。

本书假定每一章只是更长的讨论和旅程的起点。因此，这本书旨在成为对话的发起者，而不是硬道理。

-编者

## 本书中使用的惯例

本书使用以下印刷约定：

**斜体**

表示新的术语，URL，电子邮件地址，文件名和文件扩展名。

**恒定宽度**

用于程序清单，以及在段落中用于引用程序元素，例如变量或函数名称，数据库，数据类型，环境变量，语句和关键字。

**恒定宽度加粗**

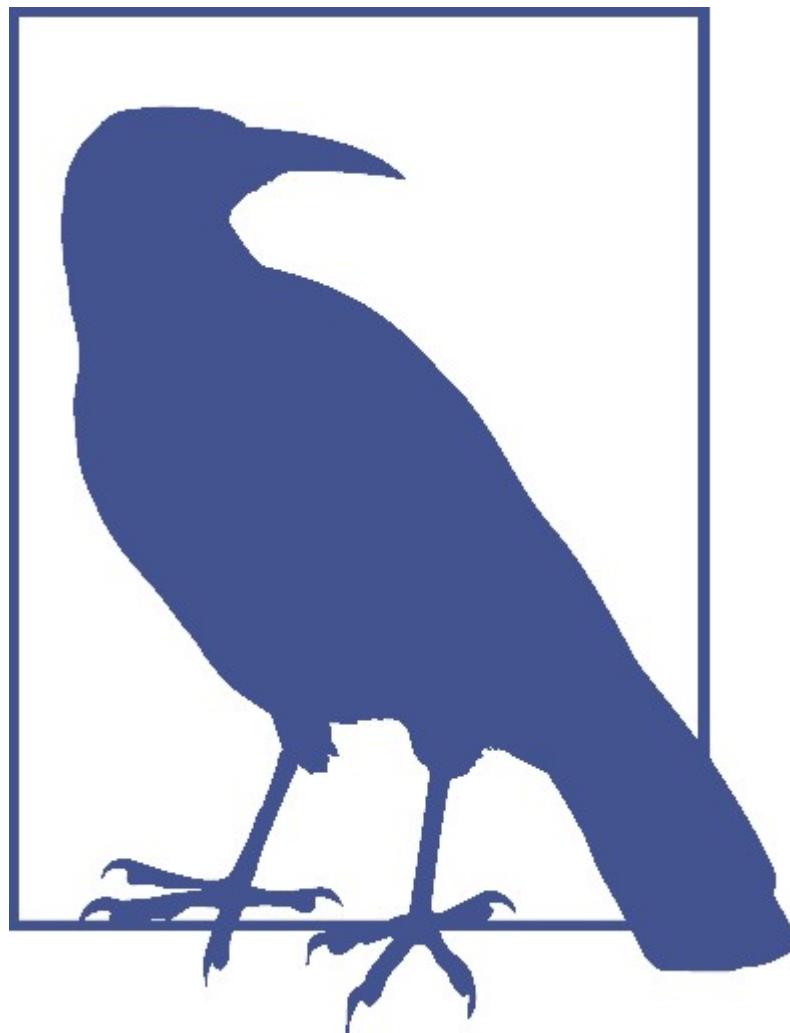
显示用户应按字面意义键入的命令或其他文本。

等宽斜体

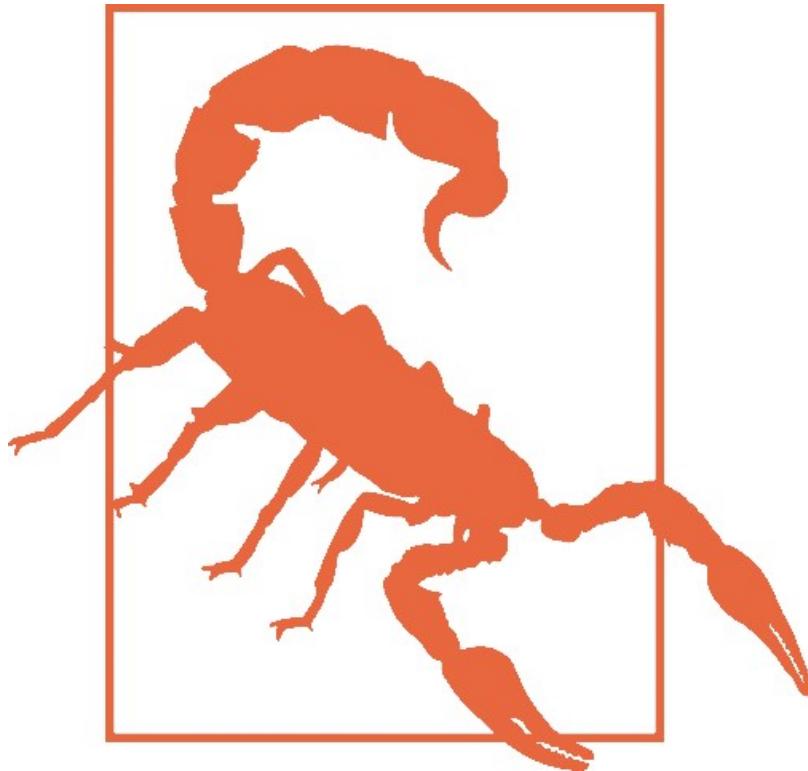
显示应由用户提供的值或由上下文确定的值替换的文本。



该元素表示一个提示或建议。



此元素表示一般注释。



此元素表示警告或注意。

## 使用代码示例

可从<http://g.co/SiteReliabilityWorkbookMaterials>下载补充材料(代码示例，练习等)。

这本书可以帮助您完成工作。通常，如果本书提供了示例代码，则可以在程序和文档中使用它。除非您要复制大部分代码，否则无需与我们联系以获取许可。例如，编写使用本书中若干代码段的程序无需许可。出售或分发O'Reilly书籍中的示例CD-ROM确实需要获得许可。引用本书并引用示例代码来回答问题无需许可。确实需要将本书中的大量示例代码合并到产品的文档中。

我们感谢但不要求注明出处。出处通常包括标题，作者，出版者和ISBN。例如：“《站点可靠性工作手册》，由Betsy Beyer，Niall Richard Murphy，David K.Rensin，Kent Kawahara和Stephen Thorne(O'Reilly)编辑。版权所有2018 Google LLC，978-1-492-02950-2。”

如果您认为使用代码示例超出合理使用范围或获得上述允许，请随时通过[permissions@oreilly.com](mailto:permissions@oreilly.com)与我们联系。

**O'Reilly Safari**



*Safari*(以前称为Safari Books Online)是会员-面向企业，政府，教育工作者和个人的培训和参考平台。

成员可以访问来自250多家发行商的数千本书，培训视频，学习路径，互动教程以及精选的播放列表，其中包括O'Reilly Media，《哈佛商业评论》，Prentice Hall Professional，Addison-Wesley Professional，Microsoft Press，Sams，Que，Peachpit Press，Adobe，Focal Press，Cisco Press，John Wiley & Sons，Syngress，Morgan Kaufmann，IBM Redbooks，Packt，Adobe Press，FT Press，Apress，Manning，New Riders，McGraw-Hill，Jones & Bartlett和Course技术，等等。

有关更多信息，请访问<http://oreilly.com/safari>。

## 如何联系我们

请将有关本书的评论和问题发送给出版商：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

塞巴斯托波尔，CA 95472

800-998-9938(在美国或加拿大)

707-829-0515(国际或本地)

707-829-0104(传真)

我们为这本书提供了一个网页，其中列出了勘误表，示例以及所有其他信息。您可以通过<http://bit.ly/siteReliabilityWkbk>访问此页面。

要对本书发表评论或提出技术问题，请发送电子邮件至  
[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)。

有关我们的书籍，课程，会议和新闻的更多信息，请访问我们的网站  
<http://www.oreilly.com>。

在脸书上找我们:<http://facebook.com/oreilly>

在推特上关注我们:<http://twitter.com/oreillymedia>

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

## 致谢

本书是作者，技术作家和评论家等100多人的热情和慷慨捐助的产物。每章都有一个供个人作者和技术作家使用的标题。我们还要花点时间感谢此处未列出的所有人。

我们要感谢以下审稿人提供了宝贵的(有时是针对性的)反馈:安倍·哈桑(Abe Hassan)，亚历克斯·佩里(Alex Perry)，卡拉·唐纳利(Cara Donnelly)，克里斯·琼斯(Chris Jones)，科迪·史密斯(Cody Smith)，德莫特·达菲(Jerrod Todd)，杰伊·贾德·科维兹(Jay Judkowitz)，约翰·T·里斯(John T.Reese)，莉兹·芳·琼斯(Liz Fong-Jones)，迈克·丹尼斯(Mike Danese)，穆拉里·苏里尔(Nuralan Desai)，尼科尔·卡斯卡拉诺(NiccolòCascarano)，拉尔夫·皮尔森(Ralph Pearson)，萨利姆(Salim Virji)，Todd Underwood，Vivek Rau和Zoltan Egyed。

我们想对以下人员表示最深切的感谢，他们将其作为我们此卷的整体质量标准。他们在整个著作中做出了巨大贡献:Alex Matey，Max Luebbe，Matt Brown和JC van Winkel。

作为Google SRE的领导人，Benjamin Treynor Sloss和Ben Lutch是这本书在Google内部的主要执行赞助商；他们坚信坚定不移的后续项目是SRE第一本书的重要伴侣，这对于使本书得以实现至关重要。

尽管在每一章中都特别感谢作者和技术作家，但我们还是想通过提供周到的意见，讨论和评论来认可对每一章做出贡献的人。按照章节顺序，它们是:

- **第2章:**哈维尔·科恩(Javier Kohen)，帕特里克·伊顿(Patrick Eaton)，理查德·邦迪(Richard Bondi)，亚尼夫·阿克宁(Yaniv Aknin)
- **第4章:**Alex Matey，Clint Pauline，Cody Smith，JC van Winkel，OlaKlapcińska，ŠtěpánDavidovič
- **第5章:**Alex Matey，Clint Pauline，Cody Smith，Iain Cooke，JC van Winkel，ŠtěpánDavidovič
- **第6章:**Dermot Duffy，James O'Keeffe，Stephen Thorne
- **第七章:**马克·布罗迪
- **第8章:**Alex Perry，Alex Hidalgo，David Huska，Sebastian Kirsch，Sabrina Farmer，Steven Carstensen，Liz Fong-Jones，Nandu Shah(Evernote)，Robert Holley(Evernote)
- **第9章:**Alex Hidalgo，Alex Matey，Alex Perry，Dave Rensin，Matt Brown，Tor Gunnar Houeland，Trevor Strohman
- **第10章:**约翰·T·里斯
- **第11章:**Daniel E. Eisenbud，Dave Rensin，Dmitry Nefedkin，DževadTrumić，Edward Wu(Niantic)，JC van Winkel，Lucas Pereira，Luke Stone，Matt Brown，Natalia Sakowska，Niall Richard Murphy，Phil Keslin(Niantic)，Rita Sodt，Scott Devold，Simon Donovan，TomaszKulczyński
- **第12章:**伊沃·克卡(Ivo Krka)，马特·布朗(Matt Brown)，尼基·尼科洛西(Nicky Nicolosi)，丹妮亚·赖莉

- **第13章:**Bartosz Janota(Spotify) , Cara Donnelly , Chris Farrar , JohannesRußek(Spotify) , Max Charas , Max Luebbe , Michelle Duffy , NelsonArapé(Spotify) , Riccardo Petrocco(Spotify) , Rickard Zwahlen(Spotify) , Robert Stephenson(Spotify) , 史蒂文·瑟古德(Steven Thurgood)
- **第14章:**夏琳·佩雷斯(Charlene Perez) , 戴夫·坎宁安(Dave Cunningham) , 戴夫·雷辛(Dave Rensin) , 杰西·范·温克尔(JC van Winkel) , 约翰·里斯(John Reese) , 斯蒂芬·索恩
- **第15章:**Alex Matey , Bo Shi , Charlene Perez , Dave Rensin , Eric Johnson , Juliette Benton , Lars Wander , Mike Danese , Narayan Desai , Niall Richard Murphy , ŠtěpánDavidovič , Stephen Thorne
- **第16章:**Alex Matey , Liz Fong-Jones , Max Luebbe
- **第17章:**安德鲁·哈维(Andrew Harvey) , 亚历山大·史基曼(Aleksander Szymanek) , 布拉德·克拉托奇维尔(Brad Kratochvil) , 埃德·韦尔温(Ed Wehrwein) , 邓肯·萨金(Duncan Sargeant) , 杰西卡·赖斯兰(Jessika Reissland) , 马特·布朗(Pattr Sieklucki)和托马斯·亚当西克
- **第18章:**Brian Balser(*纽约时报*) , Deep Kapadia(*纽约时报*) , Michelle Duffy , XavierLlorà
- **第19章:**马特·布朗
- **第20章:**Brian Balser (*New York Times*) , Christophe Kalt , Daniel Rogers , Max Luebbe , Niall Richard Murphy , Ramón Medrano Llamas , Richard Bondi , Steven Carstensen , Stephen Thorne , Steven Thurgood , Thomas Wright
- **第21章:**Dave Rensin , JC Van Winkel , Max Luebbe , Ronen Louvton , Stephen Thorne , Tom Feiner , Tsiki Rosenman

我们也感谢以下贡献者，他们提供了丰富的专业知识或资源，或者对这项工作产生了其他出色的效果:卡莱布·唐纳森(Caleb Donaldson) , 夏琳·佩雷斯(Charlene Perez) , 埃文·伦纳德(Evan Leonard) , 珍妮弗·佩托夫(Jennifer Petoff) , 朱丽叶·本顿(Juliette Benton)和莉亚·米勒(Lea Miller)

我们非常感谢从行业评论家那里获得的深思熟虑的反馈:Mark Burgess , David Blank-Edelman , John Looney , Jennifer Davis , BjörnRabenstein , Susan Fowler , Thomas A.Limoncelli , James Meickle , Theo Schlossangle , Jez Humble , Alice Goldfuss , Arup Chakrabarti , John Allspaw , Angus Lees , Eric Liang , Brendan Gregg和Bryan Liles。

我们要特别感谢Shylaja Nukala，他慷慨地投入了SRE技术写作团队的时间和技能。她热情地支持他们的必要和宝贵的努力。

还要感谢O'Reilly Media团队-弗吉尼亚·威尔逊(Virginia Wilson) , 克里斯汀·布朗(Kristen Brown) , 雷切尔·莫纳汉(Rachel Monaghan) , 尼克·麦克唐纳(Nikki McDonald) , 梅兰妮·雅伯洛(Melanie Yarbrough)和格洛里亚·卢克斯(Gloria Lukos)-的帮助和支持，使这本书在我们雄心勃勃的时间表中成为现实。

还要特别感谢Niall Richard Murphy:尽管他是在这本书问世之前就离开Google的，但他的持续洞察力和奉献精神对于在终点线获得大量有意义的内容至关重要。他的领导才能，体贴，坚韧和机智无不令人振奋！最后，编辑们还要亲自感谢以下人员:

- **贝茜·拜尔(Betsy Beyer)**:对于祖母，我是鼓励，灵感，爆米花，鼓舞人心和令人困惑的首选资源。您使这本书和我的日常生活都变得更好了！对于Duzzie，Hammer，Joan，Kiki和Mini(注意字母顺序--哈！)，他们帮助我成为了一个沉迷于作家的人，使我成为今天的人。当然，Riba也提供了DMD和其他必要的规定来推动这项工作。
- **尼尔·理查德·墨菲(Niall Richard Murphy)**:献给Léan，Oisín，Fiachra和Kay的北极星。对于那些对自身利益的抗议完全与他的行为格格不入的人。对沙龙来说，影响力比她知道的还要多。在一个光线充足的客厅里，给亚历克斯(Alex)喝杯茶，一本书，一盒骰子和你。
- **斯蒂芬·索恩(Stephen Thorne)**:对我的妈妈和爸爸来说，他们一直鼓励我努力奋斗。对我的妻子，埃尔斯佩思。对于那些给予我比我认为我应得的更多的尊重和鼓励的同事:Ola，Štěpán，Perry和David。
- **Dave Rensin**:写完第一本书后，我发誓永远不会再写另一本书。那是六本书前，我每次都说完全一样的话。对于我的妻子利亚(Lia)，后者给了我足够的空间去做，却从不说“我告诉过你”。(即使她告诉我了。)对于Google的同事们-尤其是SRE的家人-在过去的几年中，我比过去20多学到了更多有关大规模生产工程的知识。最后，本杰明·特雷诺·斯洛斯(Benjamin Treynor Sloss)采访了我，并说服我首先加入Google。
- **川原健太(Kent Kawahara)**:对于我的父母，Denby和Setsuko，以及我的Asako麻子，帮助我到达了现在的位置。感谢我的兄弟姐妹Randy和Patti多年来的支持。感谢我的妻子安吉拉(Angela)和我的儿子瑞安(Ryan)，伊桑(Ethan)和布雷迪(Brady)的爱戴和支持。最后，对于Dave，Betsy，Niall，Juliette和Stephen的核心团队，我很荣幸能与您合作进行此项目。

# 第1章

## SRE如何与DevOps相关

***class SRE implements interface DevOps***

*By Niall Richard Murphy, Liz Fong-Jones, and Betsy Beyer, with Todd Underwood, Laura Nolan, and Dave Rensin*

运维，作为一门学科，是“难”的。<sup>1</sup>不仅不存在如何很好地运行系统的这种普遍未解决的问题，而且，已经发现有效的最佳实践高度依赖于上下文，并且离被广泛采用还差很远。关于如何更好地运营团队，也存在一个很大程度上未解决的问题。一般认为，对这些主题的详细分析源自[Operational Research](#)，致力于在第二次世界大战期间改善盟军的工艺流程和产出，但实际上，我们一直在考虑如何[千年来更好地运作事物](#)。

然而，尽管付出了所有这些努力和思想，但可靠的生产操作仍然难以捉摸-尤其是在[信息技术](#)和[软件可操作性](#)的领域中。例如，企业界经常将运维视为成本中心，<sup>2</sup>这使得即使不是不可能，也很难实现有意义的结果改善。这种方法的巨大的短视性尚未得到广泛的理解，但是对此方法的不满引发了一场如何组织我们在IT中所做的工作的革命。

这场革命源于试图解决一系列共同的问题。解决这些问题的最新方法有两个不同的名称-DevOps和站点可靠性工程(SRE)。尽管我们分别讨论它们，就像它们是对上述企业心态的完全独立反应一样，<sup>3</sup>我们希望说服您，实际上它们更相似，并且每个实践者的共同点都比您假设的要多。

但是首先，需要了解每个主题的主要背景。

## DevOps的背景

DevOps是一组松散的实践，指南和文化，旨在打破IT开发，运维，网络和安全性中的孤岛。[CA\(L\)MS](#)---由约翰·威利斯(John Willis)，达蒙·爱德华兹(Damon Edwards)和杰兹·汉布尔(Jez Humble)共同阐述，它代表精益文化，自动化，精益(如[精益管理](#)，另请参阅[持续交付](#))，度量和共享-是记住DevOps哲学要点的有用缩写。共享和协作是这项运动的重中之重。在DevOps方法中，您可以改进某些内容(通常通过使其自动化)，衡量结果并与同事共享这些结果，从而可以改善整个组织。支持文化有助于所有CALMS原则。DevOps，敏捷以及各种其他业务和软件再造技术都是关于如何在现代世界中最好地开展业务的一般世界观的例子。DevOps哲学中的任何元素都不容易彼此分离，这本质上是设计使然。但是，有一些关键思想可以相对隔离地进行讨论。

### 不再需要孤岛

第一个关键思想是“不再孤岛”。这是对以下两个想法的反应：

- 历史悠久但现在越来越老套的独立运维和开发团队

- 在许多情况下，极端的[知识孤立化](#)激励纯粹用于局部优化，并且[缺乏协作](#)对业务不利<sup>4</sup>

### 事故是正常的

第二个关键思想是，事故不仅是个人孤立行为的结果，而且是由于不可避免地在事情不可避免地出错时缺少保障措施而造成的。<sup>5</sup>例如，不良的界面会在无意中导致压力下的错误行为。如果出现(未明确说明的)错误情况，系统功能失误将不可避免地导致故障；中断的监控功能使您无法知道是否出了什么问题，不必担心“什么”是错误的。一些传统思维更强的企业具有根除犯错者并予以惩罚的文化本能。但是这样做有其自身的后果：最明显的是，它创造了诱使人们混淆问题，掩盖真相并责备他人的诱因，所有这些最终都是无利可图的干扰。因此，专注于加快恢复速度比预防事故更为有利。

### 变更应循序渐进

第三个关键思想是[较小且频繁的变更是最好的](#)。在主环境中提交那些变更委员会每月开讨论详细记录在案的更改是一个激进的想法。但是，这不是一个新主意。事实证明，所有变更都必须由经验丰富的人员来考虑，并且要进行有效的考虑并分批处理，这一事实与最佳实践几乎相反。更改是冒险的，是正确的，但是正确的应对方法是将更改尽可能地拆分为较小的子组件。然后，您可以根据产品，设计和基础结构变更的常规输出构建稳定的低风险变更流水线。<sup>6</sup>此策略，再加上对较小变更的自动测试和对不良变更的可靠回滚，导致了变更方法管理，例如[持续集成\(CI\)](#)和[持续交付或部署\(CD\)](#)。

### 工具和文化相互关联

工具是DevOps的重要组成部分，特别是考虑到正确管理变更的重要性，如今，变更管理依赖于高度特定的工具。总体而言，DevOps的支持者强烈强调组织文化-而非工具-是成功采用新工作方式的关键。良好的文化可以解决残破的工具，但相反的情况很少成立。俗话说，[文化能把战略当早餐吃](#)。像操作一样，改变本身很难。

### 度量是至关重要的

最后，度量在整个业务环境中尤其重要，例如，打破孤岛和事件解决方案。在上述每种环境中，您都可以通过客观度量来确定正在发生的事情的真实性，验证您是否正在按预期方式更改情况，并为不同功能达成共识的对话创建客观基础。(这适用于业务和其他情况，例如On-Call。)

## SRE背景

Site Reliability Engineering(SRE)是Google工程副总裁Ben Treynor Sloss创造的一个术语(及相关的工作角色)。<sup>7</sup>如上一节所述，

DevOps是有关运维和产品开发之间的全生命周期协作的广泛原则。SRE是一种工作角色，是我们发现行之有效的一系列实践(如下所述)，以及使这些实践具有生气勃勃的一些信念。如果您将DevOps视为一种工作原理和工作方法，则可以辩称SRE实施了DevOps所描述的一些哲学，并且比“DevOps工程师”更接近于工作或角色的具体定义。<sup>8</sup>因此，在某种程度上，*class SRE implements interface DevOps*。

与DevOps运动源于多家公司的领导者和从业者之间的合作不同，Google的SRE在SRE这个术语在整个行业广泛普及之前，就从周围的公司那里继承了许多文化。在这种情况下，默认情况下，该学科作为一个整体目前不会像DevOps那样将文化变革作为前景。(当然，这并不意味着在任意组织中进行SRE是否需要文化变革。)

SRE由以下具体原则定义。

### 运维是软件问题

SRE的基本原则是，做好运维是一个软件问题。因此，SRE应该使用软件工程方法来解决该问题。这是一个广泛的领域，涵盖了从流程和业务变更到类似复杂但更为传统的软件问题的所有内容，例如重写堆栈以消除业务逻辑中的单点故障。

### 按服务水平目标(SLO)管理

SRE不会尝试为所有内容提供100%的可用性。正如我们在第一本书*Site Reliability Engineering*中所讨论的，出于多种原因，这是错误的目标。取而代之的是，产品团队和SRE团队为服务及其用户群选择适当的可用性目标，然后将服务管理到该SLO。<sup>9</sup>确定此类目标需要业务部门的大力协作。SLO也具有文化含义:随着利益相关者之间的协作决策，违反SLO的行为无可厚非地将团队带回了项目白板阶段。

### 努力减少琐事

对于SRE而言，任何手动的，结构强制性的操作任务都是令人讨厌的。(这并不意味着我们没有任何此类操作:我们有很多这样的操作。我们只是不喜欢他们。)我们认为，如果机器可以执行所需的操作，则通常应该机器去做。[这是一个区别\(和价值所在\)](#)在其他组织中并不常见，琐事工作是工作，而这就是您付钱给人做的事情。对于Google环境中的SRE，“琐事”不能算是“工作”。花在运维任务上的任何时间都意味着是没有花在项目正常工作上的时间，而项目工作就是我们使服务更可靠和可扩展的方式。

但是，执行运维任务确实可以通过“真实生产环境的智慧”为决策提供重要的输入。这项工作通过提供给定系统的实时反馈使我们保持基础。需要确定琐事的来源，以便可以将其最小化或消除。但是，如果您发现自己对于运维操作不够多，则可能需要更频繁地推送新功能和更改，以便工程师对所支持服务的工作方式保持熟悉。

## 生产的智慧

关于“生产的智慧”的注释:用这个短语，我们的意思是您从生产中运行的某种事物中获得的智慧-杂乱的细节，如“实际生产情况”如何表现，以及软件应“实际生产情况”如何设计，而不是将服务与实际情况隔离开的白板视图。您获得的所有页面，团队获得的罚单等等，都是与现实的直接联系，应该可以更好地指导系统设计和行为。

### 自动化今年的工作

该领域的实际工作是确定要在什么条件下进行自动化以及如何对其进行自动化。

按照Google的惯例，SRE对于团队成员可以花多少时间在琐事上有一个硬性限制，相对于工程方案维持在:50%。许多人将此限制视为上限。实际上，将其视为保证一种明确的声明和启用机制，对于采用基于工程的方法来解决问题，而不是一遍又一遍地解决问题，要有用得多。

当我们考虑自动化和琐事时，此基准与其如何发挥作用之间存在一种直观而有趣的互动。随着时间的流逝，SRE团队结束了对服务的所有可能的自动化工作，遗留下了无法自动化的事情([墨菲-拜尔效应](#))。在其他条件相同的情况下，除非采取其他措施，否则这将主导SRE团队的工作。在Google环境中，您倾向于添加更多的服务，直至达到仍支持50%工程时间的某些限制，或者您在自动化方面非常成功，因此可以去做其他完全不同的事情。

#### 通过降低失败成本来快速行动

SRE参与的主要好处之一是不一定能提高可靠性，尽管显然确实如此。它实际上是提高了产品开发的产出。为什么？嗯，减少常见故障的平均维修时间(MTTR)可以提高产品开发人员的速度，因为工程师不必浪费时间并集中精力解决这些问题。这源于众所周知的事实，即在产品生命周期中发现问题的时间越晚，[修复它的成本就越高](#)。SRE专门负责改善不良问题的发现，从而为整个公司带来收益。

#### 与开发者共享所有权

“应用程序开发”和“生产”(有时称为程序员和运维人员)之间的严格界限适得其反。如果将职责分开和运维作分类为成本中心会导致权力不平衡，尊重或报酬尤其存在差异。

SRE倾向于将重点放在生产问题上，而不是业务逻辑问题上，但是由于他们的方法带来了软件工程工具来解决该问题，因此它们与产品开发团队共享技能。通常，SRE在他们正在照看的服务的可用性，延迟，性能，效率，变更管理，监控，紧急响应和容量规划方面具有特殊的专业知识。那些特定的(通常是定义明确的)能力是SRE为产品以及相关产品开发团队所做的工作的基础。[理想情况下，产品开发和SRE团队都应该对以下方面有整体了解堆栈-前端，后端，库，存储，内核和物理机-且任何团队都不应嫉妒拥有单个组件。事实证明，如果您“模糊界限”<sup>11</sup>并拥有SRE工具JavaScript，或者产品开发人员使内核合格，那么您将可以做更多的事情：关于如何进行更改的知识以及进行更改的权限将更加广泛。]，并取消了嫉妒任何特定功能的奖励措施。

在[Site Reliability Engineering](#)中，我们没有清楚表明Google的产品开发团队默认持有其服务。尽管SRE原则仍然可以指导在整个Google范围内管理服务的方式，但SRE既不提供服务也不保证提供大量服务。<sup>12</sup>当SRE团队与产品开发团队合作时，所有权模型最终也是共享模型。

#### 无论功能或职称如何，都使用相同的工具

工具是决定行为的极其重要的决定因素，并且假设SRE在Google上下文中的功效与可广泛使用的[统一代码库](#)无关各种各样的软件和系统工具，[高度优化和专有生产堆栈](#)，等等。然而，我们与DevOps共享这一绝对假设：关注服务<sup>13</sup>的团队应使用相同的工具，无论他们在组织中的角色如何。没有一种好的方法来管理一项具有一个针对SRE的工具以及一个针对产品开发人员不同的工具的服务，并且在不同情况下的行为不同(并且可能是灾难性的)。您的分歧越大，您的公司从每次改进每个工具的努力中获得的利益就越少。

## 比较和对比

回顾前面的原理，我们立即在概述的要点上看到很多共性：

- DevOps和SRE都取决于必须接受变更才能改进。没有这些，就没有太多的操作空间。[14](#)
- 协作是DevOps工作的重中之重。有效的共享所有权模型和合作伙伴团队关系对于SRE起作用是必不可少的。像DevOps一样，SRE在整个组织中也具有很强的共享价值，这可以使退出基于团队的孤岛变得更加容易。
- 最好以小的，连续的动作来进行变更管理，理想情况下，大多数动作都是自动测试和应用的。变更与可靠性之间的关键相互作用使得这对于SRE尤其重要。
- 正确的工具至关重要，而工具在一定程度上决定了行为的范围。但是，我们决不能太集中精力使用某些特定的工具来实现某些目标。归根结底，面向系统管理的API是一个更重要的理念，它比任何特定实现都更长久。
- 度量是DevOps和SRE如何工作的绝对关键。对于SRE，SLO在确定为改善服务而采取的措施中占主导地位。当然，没有度量标准就不可能有SLO(以及跨团队辩论-理想情况是产品，基础架构/ SRE和业务之间)。对于DevOps，度量行为通常用于了解过程的输出是什么，反馈循环的持续时间是什么，等等。DevOps和SRE都是面向数据的事物，无论它们是专业还是哲学。
- 管理生产服务的残酷现实意味着坏事偶尔会发生，您必须谈论原因。SRE和DevOps都实行[对事不对人的事后检查](#)，以抵消无助的，充满肾上腺素的反应。
- 最终，实施DevOps 或 SRE是一项整体举措；双方都希望通过高度明确的合作方式使整个团队(或单位或组织)变得更好。对于DevOps和SRE，应该有更好的速度。[15](#)

如您所见，DevOps和SRE之间有很多共同点。

但是也存在重大差异。从某种意义上说，DevOps是更广泛的哲学和文化。因为与SRE相比，它带来的变化更大，所以DevOps对上下文更敏感。DevOps对如何详细运行操作相对保持沉默。例如，它不是关于服务的精确管理的规定。相反，它选择专注于打破更广泛组织中的障碍。这具有很大的价值。

另一方面，SRE的职责相对狭窄，其职责通常面向服务(和面向最终用户)，而不是面向整个业务。结果，它为如何有效运行系统的问题带来了一个固执己见的知识框架(包括[错误预算](#)等概念)。尽管SRE作为一种专业非常了解激励措施及其影响，但它反而对孤岛化和信息障碍等话题相对沉默。它不一定会由于业务案例而支持CI和CD，而是因为所涉及的改进的操作实践会支持CI和CD。或者，换句话说，SRE 相信与DevOps相同，但原因略有不同。

## 组织环境与成功采用者的培养

DevOps和SRE在操作方式上有很大的概念重叠。正如您可能期望的那样，它们在组织中还必须具备一组相似的条件，才能使它们a)首先实现，并且b)从该实现中获得最大收益。正如[Tolstoy马上但从来没有说过](#)，有效的操作方法都是相似的，而破损的方法都是以自己的方式破损的。激励可以部分解释为什么会这样。

如果组织的文化重视DevOps方法的好处并愿意承担这些费用-通常表示为招聘困难，维持团队流动性和职责所需的精力以及专用于补偿技能的财务资源这种情况肯定更罕见了-然后，组织还必须确保奖励正确无误，才能充分利用此方法。

具体来说，在DevOps和SRE的上下文中，以下内容均应适用。

### **有限，严格的激励措施会使您的成功**

许多公司意外地定义了破坏集体绩效的正式激励措施。为避免此错误，请不要将激励措施与发起相关或可靠性相关的结果紧密联系在一起。[任何经济学家](#)都可以告诉您，如果存在数字度量标准，人们会找到一种方法来对它产生不良影响，有时甚至是一种完全出于善意的方式。<sup>16</sup>相反，您应该让您的人民自由地找到正确的权衡。如前所述，DevOps或SRE总体上可以充当产品团队的促进剂，从而使其余的软件组织能够以持续可靠的方式向客户提供功能。这种动态变化还解决了传统的和分散的系统/软件组方法的一个持续存在的问题：在设计和生产之间缺乏反馈回路。具有早期SRE参与度(理想情况下是在设计时)的系统通常在部署后在生产中会更好地运行，而不管是谁负责管理服务。(没有任何事情会像丢失用户数据一样拖慢功能开发。)

### **最好自己修复；别怪别人**

此外，避免任何诱因将生产事件或系统故障归咎于其他群体。在很多方面，非常规的动因是工程运营传统模型的核心问题，因为将运维和软件团队分开可以产生单独的激励机制。相反，请考虑采用以下做法来打击组织级别中的责任过失

- 不仅要允许工程师，而且要积极鼓励工程师在产品需要时更改代码和配置。还应允许这些团队在其任务范围内具有激进的权威，从而消除了进行得更慢的动机。
- 支持对事不对人的事后检查。<sup>17</sup>这样做消除了淡化或掩盖问题的动机。这一步对于充分理解产品并实际优化其性能和功能至关重要，它依赖于前面提到的生产智慧。

允许支持从难以操作的产品上移开。威胁要撤消支持促使产品开发人员在启动支持过程中以及在产品本身获得支持时解决问题，从而节省了所有人的时间。取决于您的上下文，“难以克服的运维困难”的含义可能会有所不同-此处的动态应该是相互理解的职责之一。对其他组织的驳回应该要比较温和，比如“我们认为拥有这种技能的人会更有价值地使用时间，”或者“这些人如果承担太多的运维工作任务而没有机会使用他们的工程技能会辞职”在Google，完全从此类产品中撤回支持的做法已成为一种制度。

### **将可靠性工作视为专门角色**

在Google，SRE和产品开发是独立的组织。每个小组都有自己的重点，优先级和管理，而不必互相竞标。但是，当产品成功时，产品开发团队将通过新员工有效地资助SRE的增长。这样，产品开发与SRE团队的成功息息相关，就像SRE与产品开发团队的成功息息相关。SRE还很幸运地获得了管理层的高层支持，这确保了工程团队对“SRE方式”支持服务的异议通常是短期的。不过，您无需拥有组织结构图即可以不同的方式进行操作-您只需要一个不同的实践社区涌现出来。

无论您分叉组织结构图还是使用更多非正式机制，认识到专业化会带来挑战非常重要。DevOps和SRE的从业者受益于拥有支持和职业发展的同龄人社区，而工作阶梯则奖励他们<sup>18</sup>他们带来的独特技能和观点。

请务必注意，Google所采用的组织结构以及上述某些激励措施在某种程度上依赖于规模较大的组织。例如，如果您的20人创业公司只有一种(相对较小的)产品，那么允许撤回运维支持就没有多大意义。仍然可以采用DevOps风格的方法，<sup>19</sup>但是，

如果从字面上看，您能做的所有事情就是帮助它发展，那么改善可操作性差的产品的能力就会受到损害。不过，通常情况下，人们如何满足这些增长需求以及技术债务积累的速度比他们想象的要多。<sup>20</sup>

#### ***When Can Substitute for Whether***

但是，当您的组织或产品规模增长到一定程度时，您可以在支持哪些产品或如何“优先”分配该支持方面行使更大的自由度。如果很明显，对系统X的支持比对系统Y的支持要早得多，则隐式条件的作用与在SRE世界中“不支持”服务的选择几乎相同。

在Google，事实证明，SRE与产品开发之间的牢固合作至关重要：如果您的组织中存在这种关系，则可以基于有关比较运营特征的客观数据来决定是否撤回(或提供)支持，从而避免产生非生产性的影响。

SRE与产品开发之间的有效关系还有助于避免组织上的反模式，在这种模式下，产品开发团队必须在产品或功能准备就绪之前就将其交付。相反，SRE可以与开发团队一起对产品进行改进，然后再将维护工作的负担转移到最不擅长修理的人员身上。

#### **争取平等对待：职业与财务**

最后，请确保采取正确行动的职业动机是：我们希望我们的DevOps / SRE组织与产品开发部门保持同样的自尊心。因此，每个团队的成员应采用大致相同的方法进行评级，并具有相同的经济动机。

## **结论**

在实践和哲学上，DevOps和SRE在许多方面都在IT运营的总体环境中彼此非常接近。

DevOps和SRE都需要讨论，管理支持和实际工作人员的支持，以取得重大进展。实施它们中的任何一个都是一个过程，而不是一个快速的解决方案：重命名和羞耻<sup>21</sup>的做法是一个空洞的，不太可能产生收益。鉴于它是如何执行操作的更自以为是的实现方式，尽管需要进行特定的调整，但SRE对于如何在此旅程的早期更改工作实践提出了更具体的建议。具有更广泛关注点的DevOps在某种程度上很难推理并转化为具体步骤，但是正是由于该关注点更广泛，它可能会遇到较弱的初始阻力。

但是，每个人的实践者都使用许多相同的工具，相同的变更管理方法以及相同的基于数据的决策思维方式。归根结底，无论我们叫什么名字，我们所有人都面临着同样的持续问题：生产，和使其变得更好。

对于那些有兴趣进一步阅读的人，以下建议应有助于您对当前发生的运维革命的文化，业务和技术基础有更广泛的了解：

- [站点可靠性工程](#)
- [有效的DevOps](#)
- [The Phoenix Project](#)
- [云系统管理实践：Web 服务的DevOps和SRE实践，第2卷](#)
- [加速：精益软件和DevOps科学](#)

1. 请注意，此讨论出现在有关SRE的书中，其中一些讨论特定于软件服务操作，而不是IT操作。 ↵
2. 玛丽·波彭迪克(Mary Poppendieck)在这方面有一篇出色的文章，名为"成本中心陷阱"。这种方法失败的另一种方式是，当一场非常大且不可能的灾难完全抵消了您采用低等级运营模式所节省的成本时(参见2017年5月英国航空公司的停运)。 ↵
3. 当然，还有许多其他潜在的反应。例如，ITIL®是另一种IT管理方法，倡导更好的标准化。 ↵
4. 还要注意，因为这是一个复杂的世界，所以也存在正效应到分区，孤岛等，但是在运营领域，不利因素似乎尤其严重。 ↵
5. 请参阅[https://en.wikipedia.org/wiki/Normal\\_Accidents](https://en.wikipedia.org/wiki/Normal_Accidents)。 ↵
6. 较高风险的更改或无法通过自动方式确认的更改，显然应该由人类审核，即使不是由他们实施的。 ↵
7. Google的SRE历史来自前身的团队，该团队更注重运营，Ben从工程的角度提供了解决问题的动力。 ↵
8. 这在很多方面都是用词不当，也许最根本的原因是您不能仅仅雇用一些人，称他们为"DevOps工程师"，并立即期望收益。您必须接受改变工作方式以受益的全部理念。正如Andrew Clay Shafer所说一样，"人们出售DevOps，但您买不到。"而且，正如Seth Vargo在"DevOps的十大神话"中指出的那样，您不能"雇用DevOp来修复您的组织"。 ↵
9. 服务水平目标是特定指标性能的目标(例如，有99.9%的时间可用)。 ↵
10. 当然，不是每个团队都做所有事情，但是这些是SRE工作的最常见标题。 ↵
11. 如果您将其视为分层堆栈，请执行分层冲突。 ↵
12. 实际上，有任何内容\*的生产准备情况审查；SRE不仅会从一开始就提供车载服务。 ↵
13. 服务通常被定义为运行以执行某些业务需求的软件，通常具有可用性约束。 ↵
14. 在Google内部，这个问题已得到解决，服务始终会更改状态，配置，所有权，方向等。在一定程度上，Google的SRE曾是"抗争是必要的"论点的受益者，该论点过去曾被抗争并赢得过多次胜利。但是，正如威廉·吉布森(William Gibson)可能说的那样，并不是完全均匀分布。 ↵
15. 请参阅<https://devops-research.com/research.html>上的相关研究。 ↵
16. 请参阅[http://en.wikipedia.org/wiki/Goodhart%27s\\_law](http://en.wikipedia.org/wiki/Goodhart%27s_law)和<https://skybrary.aero/bookshelf/books/2336.pdf>。 ↵
17. 参见例如<https://codeascraft.com/2012/05/22/blameless-postmortems/>。 ↵

18. 在具有发达文化的组织中。早期公司可能没有建立奖励这些工作角色的方法。[←](#)

19. 确实，可以说，除非您外包操作，否则这是您的唯一选择。[←](#)

20. 有关如何在不同上下文中应用SRE原则的讨论，请参见第20章。[←](#)

21. 换句话说，简单地淘汰一组DevOps或SRE，而无需改变他们的组织位置，导致在无法实现预期的改进时不可避免地使团队蒙羞。[←](#)

## PART I 基础

每个实施指南都需要从一个共同的基础开始。在这种情况下，SRE的基本基础包括 *SLO*，*监控*，*警报*，*减少人工*和*简化*。正确掌握这些基础知识将为您在SRE旅途中取得成功做好准备。

以下各章探讨了将这些核心原则转变为组织的具体实践的技术。

## 第2章

### 实施SLO

*史蒂文·瑟古德(Steven Thurgood)和大卫·弗格森(David Ferguson)以及亚历克斯·希达尔戈(Alex Hidalgo)和贝茜·拜尔(Betsy Beyer)*

服务水平目标(SLO)指定了服务可靠性的目标水平。由于SLO是做出以数据为依据的可靠性决策的关键，因此它们是SRE实践的核心。在许多方面，这是本书中最重要的章节。

一旦掌握了一些准则，设置初始SLO以及完善它们的过程就很简单了。[第4章](#)介绍了SLO和SLI(服务级别指标)的主题，并提供了有关如何使用它们的一些建议。

在讨论了SLO和错误预算背后的动机之后，本章提供了逐步指南，可让您开始考虑SLO，并提供了一些有关如何从那里进行迭代的建议。然后，我们将介绍如何使用SLO做出有效的业务决策，并探讨一些高级主题。最后，我们将为您提供一些针对不同类型服务的SLO的示例，以及一些有关在特定情况下如何创建更复杂的SLO的指针。[22](#)

## 为什么SRE需要SLO

即使是最大的组织，工程师也是稀缺资源。应将工程时间用于最重要服务的最重要特征。要在赢得新客户或留住现有客户的功能性投资与将使这些客户满意的可靠性和可扩展性方面进行投资之间取得正确的平衡，是很难的。在Google，我们了解到，经过深思熟虑并采用的SLO是就可靠性工作的机会成本做出数据明智的决定并确定如何适当地确定工作优先级的关键。

SRE的核心职责不仅仅是使“所有事物”自动化并保持On-Call。他们的日常任务和项目由SLO推动：确保SLO在短期内得到捍卫，并在中长期内得到维护。甚至可以说没有SLO，就不需要SRE。SLO是一种工具，可帮助您确定要优先处理的工程工作。例如，请考虑两个可靠性项目的工程设计权衡：自动回滚和移至复制的数据存储。通

过计算对错误预算的估计影响，我们可以确定哪个项目对我们的用户最有利。有关更多详细信息，请参见第37页的“使用SLO和错误预算进行决策”，以及\*《站点可靠性工程》中的“[管理风险](#)”。

## 入门

作为建立一组基本SLO的起点，让我们假设您的服务是某种形式的代码，已被编译和发布，并且在用户通过Web访问的联网基础结构上运行。系统的成熟度级别可能是以下之一：

- 未开发项目，目前未部署
- 一个生产中的系统，带有一些监控功能，可在出现问题时通知您，但没有正式的目标，没有错误预算的概念，并且有100%正常运行时间的潜意识目标
- 正在运行的部署SLO低于100%，但是对其重要性或如何利用其做出持续改进选择没有共识，换句话说，SLO毫无用处

为了对站点可靠性工程采用基于错误预算的方法，您需要达到以下条件成立的状态：

- 组织中的所有利益相关者都已批准了适用于该产品的SLO。
- 负责确保服务符合其SLO的人员已同意在正常情况下可以满足此SLO。
- 组织已承诺将错误预算用于决策和确定优先级。此承诺已在错误预算政策中正式化。
- 有完善SLO的流程。

否则，您将无法采用基于错误预算的方法来提高可靠性。SLO合规性将仅仅是另一个KPI(关键绩效指标)或报告指标，而不是决策工具。

### 可靠性目标和错误预算

制定适当的SLO的第一步是讨论SLO应该是什么以及应该涵盖什么。

SLO为服务的客户设置了目标可靠性级别。高于此阈值，几乎所有用户都应该对您的服务感到满意(假设他们对服务的实用性感到满意)。<sup>23</sup>低于此阈值，用户可能会开始抱怨或停止使用该服务。最终，用户的幸福才是最重要的-幸福的用户使用该服务，为您的组织创造收入，对您的客户支持团队提出低要求并向其朋友推荐该服务。我们保持可靠的服务，以使客户满意。

客户满意度是一个相当模糊的概念。我们无法精确测量。通常我们对此几乎没有任何了解，那么我们如何开始呢？我们的第一个SLO使用什么？

我们的经验表明，100%的可靠性是错误的目标：

- 如果您的SLO与客户满意度保持一致，那么100%并不是一个合理的目标。即使具有冗余组件，自动运行状况检查和快速故障转移，一个或多个组件同时发生故障的可能性也不为零，从而导致可用性不到100%。
- 即使您可以在系统中实现100%的可靠性，您的客户也不会体验到100%的可靠性。您和客户之间的系统链通常很长而且很复杂，并且其中任何一个组件都可能发生故障。<sup>24</sup>这也意味着，当您从99%变为99.9%到99.99%的可靠性时，每增加9个可靠性成本增加，但对客户的边际效用却稳定地接近零。

- 如果您确实设法为客户创造了100%可靠的体验，并且想要保持这种可靠性，那么您将永远无法更新或改善服务。停机的第一大原因是变化:推出新功能，应用安全补丁，部署新硬件以及扩大规模以满足客户需求，将对100%的目标产生影响。迟早，您的服务将停滞，您的客户将前往其他地方，这对任何人的底线都不利。
- 100%的SLO意味着您只有时间进行反应。实际上，您只能对<100%的可用性做出反应，这是可以保证的。100%的可靠性不是工程文化的SLO，而是运维团队SLO。

一旦SLO目标低于100%，它就必须由组织中有权在速度特征和可靠性之间进行权衡的人来把控。在小型组织中，这可能是CTO。在大型组织中，通常是产品所有者(或产品经理)。

### 度量方法:使用SLI

一旦您同意100%是错误的数字，您将如何确定正确的数字？您到底在测量什么？服务水平指标器(SLI)在这里起作用:SLI是您提供的服务水平的“指示器”。尽管许多数字可以用作SLI，但我们通常建议将SLI视为两个数字的比值:好事件的数量除以事件的总数。例如:

- 成功的HTTP请求数/总HTTP请求数(成功率)
- 在<100毫秒内成功完成的gRPC调用次数/总gRPC请求数
- 使用了整个语料库的搜索结果数/搜索结果总数，包括正常降低的搜索结果数
- 使用少于10分钟的库存数据的产品搜索发出的“库存检查计数”请求数/库存检查请求总数
- 根据该指标的某些扩展标准列表得出的“良好用户分钟数”/用户分钟总数

这种形式的SLI具有几个特别有用的属性。SLI的范围是0%到100%，其中0%表示没有任何作用，而100%表示没有损坏。我们发现这种规模很直观，并且这种风格很容易将其应用于错误预算的概念:SLO是目标百分比，错误预算是100%减去SLO。例如，如果您有99.9%的成功SLO，则在四个星期的时间内收到300万个请求的服务在此期间的预算错误为3,000(0.1%)。如果一次中断造成1,500个错误，则该错误将花费错误预算的50%。<sup>25</sup>

此外，使所有SLI遵循一致的样式可以使您更好地利用工具:可以编写警报逻辑，SLO分析工具，错误预算计算和报告以期望输入相同的值:分子，分母和阈值。在这里简化是一个好处。

首次尝试制定SLI时，您可能会发现将SLI进一步分为SLI规范和SLI实现很有用:

#### SLI规格

您认为对用户至关重要的服务结果的评估，与评估结果的方式无关。例如:  
<100毫秒内加载的主页请求的比率

#### SLI实施

SLI规范及其度量方法。例如:

- 从服务器日志的“延迟”列中测算，首页请求在<100毫秒内加载的比率。此度量将错过未能到达后端的请求。
- 在小于100毫秒内加载的首页请求的比例，由在虚拟机中运行的浏览器中执行JavaScript的探测器测量。当请求无法到达我们的网络时，此度量将捕获错误，但可能会丢失仅影响一部分用户的问题。
- 主页请求在<100毫秒内加载的比率，该比率通过主页上JavaScript本身的检测进行测量，并报告回专用遥测记录服务。尽管我们现在需要修改代码以捕获此信息并构建用于记录该信息的基础结构，但此度量将更准确地捕获用户体验，该规范具有其自身的可靠性要求。

如您所见，单个SLI规范可能具有多个SLI实现，每个实现都有自己的一套利弊，包括质量(他们捕获客户体验的准确程度)，覆盖范围(他们捕获所有客户的体验的程度客户)和成本。

您第一次尝试SLI和SLO不一定正确。最重要的目标是进行适当的测量并建立反馈环路，以便您进行改进。(我们在第34页的“持续改进SLO目标”中更深入地探讨了此主题。)

在[我们的第一本书](#)中，我们建议不要根据当前的性能来选择SLO，因为这会使您不必要地严格执行SLO。尽管该建议是正确的，但是如果您没有其他信息，并且有适当的迭代过程(稍后将进行介绍)，则可以将当前的性能作为一个不错的起点。但是，在优化SLO时，不要让当前的性能受到限制：客户还会期望您的服务在其SLO上运行，因此，如果您的服务在不到10ms的时间内成功返回了99.999%的请求，则从该基线明显下降可能会使他们感到不高兴。

要创建第一组SLO，您需要确定一些与服务相关的关键SLI规范。可用性和延迟SLO非常普遍；新鲜度，耐用性，正确性，质量和覆盖范围SLO也应有的地位(我们将在后面详细讨论)。

如果您在弄清要开始使用哪种SLI时遇到问题，则可以从简单开始：

- 选择一个您要为其定义SLO的应用程序。如果您的产品包含许多应用程序，则可以在以后添加。
- 在这种情况下，明确确定谁是“用户”。这些是您正在优化其幸福感的人。
- 考虑用户与系统交互的常见方式-常见任务和关键活动。
- 绘制系统的高级体系结构图；显示关键组件，请求流，数据流和关键依赖项。将这些组件分为以下部分列出的类别(可能存在一些重叠和模糊之处；请运用直觉，不要让完美成为商品的敌人)。

您应该仔细考虑所选择的SLI，但也不要过于复杂。特别是如果您刚刚开始SLI之旅，请选择与系统相关但易于评估的方面-您以后可以随时进行迭代和完善。

### 组件类型

设置SLI的最简单方法是将系统抽象为几种常见的组件类型。然后，您可以使用每个组件的推荐SLI列表来选择与您的服务最相关的SLI：

#### 请求驱动

用户创建某种类型的事件并期望得到响应。例如，这可以是HTTP服务，用户可以在其中与浏览器进行交互，或者可以与移动应用程序的API进行交互。

### 管道

一个将记录作为输入，对其进行变异并将其输出放置在其他位置的系统。这可能是在单个实例上实时运行的简单过程，也可能是耗时数小时的多阶段批处理过程。示例包括：

- 定期从关系数据库读取数据并将其写入分布式哈希表以优化服务的系统
- 一种视频处理服务，可将视频从一种格式转换为另一种格式
- 从许多来源读取日志文件以生成报告的系统
- 监控系统，可从远程服务器获取指标并生成时间序列和警报

### 存储

接受数据(例如字节，记录，文件，视频)并使之可在以后获取的系统。

## 工作示例

考虑一个简化的手机游戏架构，如图2-1所示。

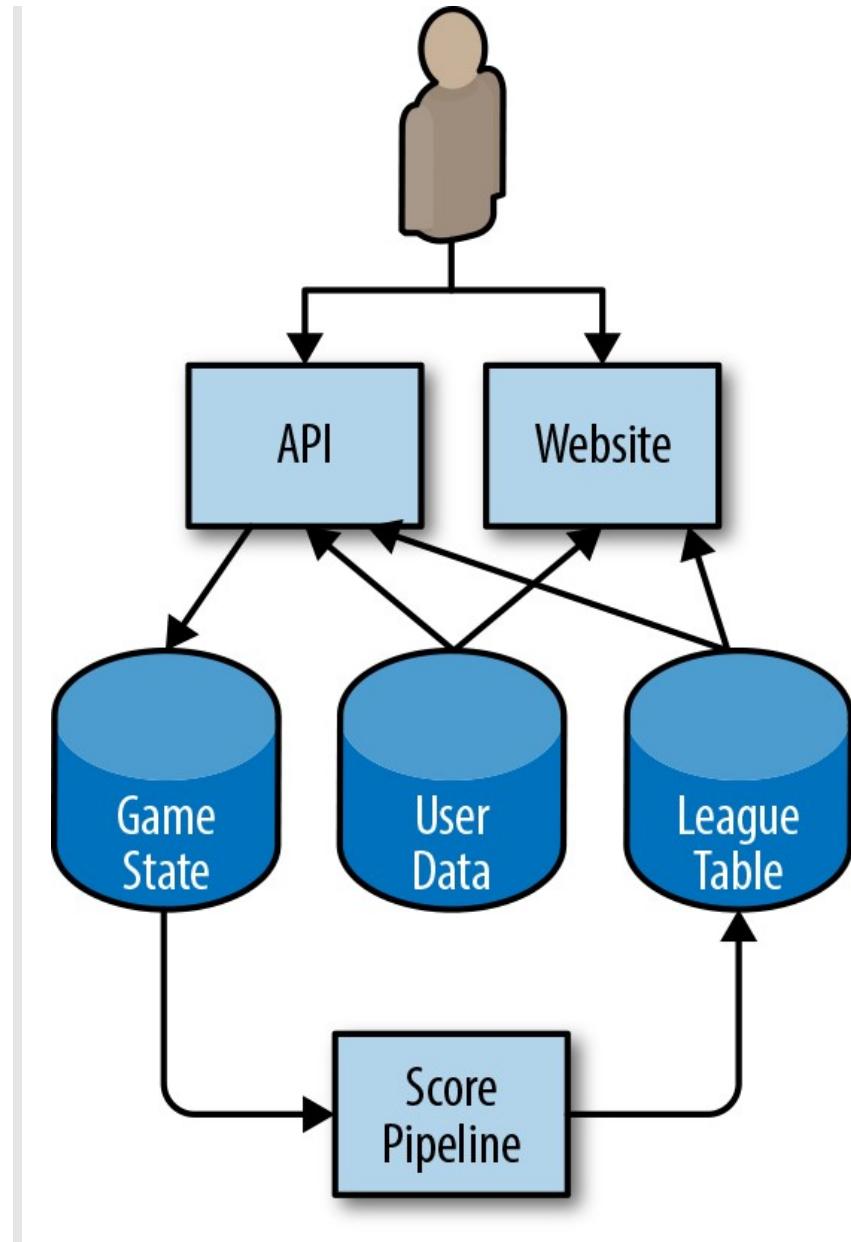


图2-1.示例手机游戏的架构

用户手机上运行的应用程序与云中运行的HTTP API进行交互。API将状态更改写入永久存储系统。管道定期运行这些数据以生成排行榜，该排行榜为今天，本周和所有时间提供高分。此数据将写入单独的联赛表数据存储，结果可通过移动应用程序（用于游戏中的得分）和网站获得。用户可以将自定义头像上传到“用户数据”表中，这些头像通过API在游戏中和高分网站上均可使用。

有了这个设置，我们就可以开始考虑用户如何与系统交互，以及哪种SLI可以衡量用户体验的各个方面。

这些SLI中的某些可能重叠：请求驱动的服务可能具有正确性SLI，管道可能具有可用性SLI，而持久性SLI可能被视为正确性SLI的变体。我们建议选择少量（五种或更少）的SLI类型，这些类型代表对客户最关键的功能。

为了捕捉典型的用户体验和长尾，我们还建议对某些类型的SLI使用多个等级的SLO。例如，如果90%的用户请求在100毫秒内返回，而其余10%的请求花了10秒钟，则许多用户将不满意。一个延迟SLO可以通过设置多个阈值来捕获此用户群：90%的请求快于100毫秒，而99%的请求快于400毫秒。该原理适用于所有具有衡量用户不快乐参数的SLI。

表2-1提供了用于不同类型服务的一些常见SLI。

表2-1. 适用于不同类型组件的潜在SLI

服务类型	SLI类型	说明
请求驱动	可用性	成功响应的请求比例
请求驱动	延迟	超过某个阈值的请求比例
请求驱动	品质	如果在过载或后端不可用时服务正常降级，则需要测量以未降级状态提供服务的响应的比例。例如，如果用户数据存储区不可用，则游戏仍可玩，但使用同类图像
管道	新鲜度	比某个时间阈值更新的数据比例。 理想情况下，此指标计算用户访问数据的次数，以便最准确地反映用户体验
管道	正确性	进入管道的产生正确的值的记录所占的比例。
管道	覆盖范围	对于批处理，在某些目标数据量以上进行处理的作业比例。 对于流处理，在某个时间窗口内成功处理的传入记录的比例
储存	耐用性	可以成功读取的已写入记录的比例。 请特别注意耐用性SLI： 用户所需的数据可能只是存储的数据的一小部分。 例如，如果您在过去的10年中有10亿条记录，但是用户只想要今天的记录(不可用)，那么即使几乎所有数据都是可读的，它们也会让您感到不满意

### 从SLI规范过渡到SLI实施

现在我们知道我们的SLI规范，我们需要开始考虑如何实现它们。

对于您的第一个SLI，请选择最少的工程工作。如果您的Web服务器日志已经可用，但是设置探针将花费数周，而对JavaScript进行检测则将花费数月，请使用日志。

您需要足够的信息来衡量SLI:对于可用性，您需要成功/失败状态。对于缓慢的请求，您需要花费时间来处理请求。您可能需要重新配置Web服务器以记录此信息。如果您使用的是基于云的服务，则某些信息可能已在监控仪表板中提供。

对于我们的示例体系结构，SLI实现有多种选择，每种选择都有其优缺点。以下各节详细介绍了系统中三种类型的组件的SLI。

#### API和HTTP服务器的可用性和延迟

对于所有考虑的SLI实现，我们将响应成功基于HTTP状态代码。5XX响应应为SLO，而其他所有请求均视为成功。可用性SLI是成功请求的比例，而延迟SLI是比定义的阈值快的请求的比例。

您的SLI应该是具体且可衡量的。总结“测量方法:使用SLI”(第20页)，您的SLI可以使用以下一种或多种来源:

- 应用程序服务器日志
- 负载均衡器监控
- 黑盒监控
- 客户端工具

我们的示例使用负载平衡器监控功能，因为这些指标已经可用，并且提供的SLI比应用程序服务器日志中的SLI更接近用户体验。

#### 管道新鲜度，覆盖范围和正确性

当我们的管道更新League表时，它会记录一个水位标志，其中包含更新数据的时间戳。一些示例SLI实现:

- 在超级联赛表格中运行定期查询，计算新记录的总数和记录的总数。不管有多少用户看到了数据，这都会将每个陈旧记录视为同等重要。
- 当所有联盟客户请求新数据时，请他们检查水位标志，并增加一个表示已请求数据的指标计数器。如果数据比预定的阈值新，则增加另一个计数器。

从这两个选项中，我们的示例使用客户端实现，因为它提供了与用户体验紧密相关且易于添加的SLI。

为了计算我们的覆盖率SLI，我们的管道将导出应处理的记录数和成功处理的记录数。由于配置错误，此指标可能会丢失我们的管道不知道的记录

我们有几种可能的方法来衡量正确性:

- 将具有已知输出的数据注入系统，并计算输出符合我们期望的次数的比例。
- 使用一种方法来根据与管道本身不同的输入(可能更昂贵，因此不适合我们的管道)来基于输入计算正确的输出。使用它来采样输入/输出对，并计算正确输出记录的比例。该方法论假设创建这样一个系统既可能又可行。

我们的示例将其正确性SLI基于游戏状态数据库中一些手动管理的数据，并在每次管道运行时测试已知的良好输出。我们的SLI是测试数据正确输入的比例。为了使此SLI能够代表实际的用户体验，我们需要确保手动整理的数据可以代表真实的数据。

#### 测量SLI

图2-2显示了我们的白盒监控系统如何从示例应用程序的各个组件中收集指标。

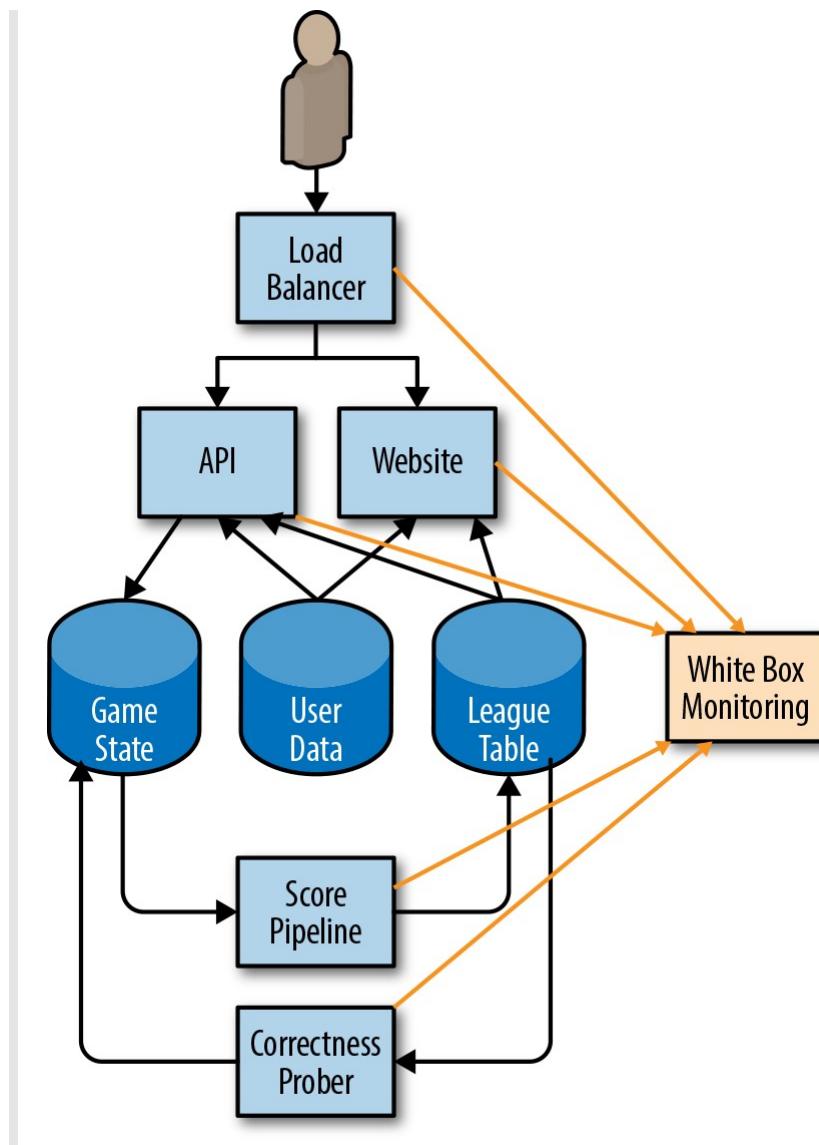


图2-2. 我们的监控系统如何收集SLI指标

让我们来看一个示例，该示例使用监控系统中的指标来计算启动器SLO。尽管我们的示例使用了可用性和延迟指标，但相同的原理也适用于所有其他潜在的SLO。有关我们系统使用的指标的完整列表，请参阅附录A。我们所有的示例都使用[Prometheus表示法](#)。

### 负载均衡器指标

后端请求总数("api"或"web")和响应代码:

```
http_requests_total{host="api", status="500"}
```

总延迟，作为累积直方图；每个存储桶都会计算花费少于或等于该时间的请求数:

```
http_request_duration_seconds{host="api", le="0.1"}
http_request_duration_seconds{host="api", le="0.2"}
http_request_duration_seconds{host="api", le="0.4"}
```

一般而言，对慢速请求进行计数要比用直方图对其进行近似更好。但是，由于该信息不可用，因此我们使用了监控系统提供的直方图。另一种方法是将显式慢请求计数基于负载平衡器配置中的各种慢阈值(例如，对于100 ms和500 ms的阈值)。此策略将提供更准确的数字，但需要更多配置，这使得追溯更改阈值变得更加困难。

```
http_request_duration_seconds{host="api", le="0.1"}  
http_request_duration_seconds{host="api", le="0.5"}
```

### 计算SLI

使用前面的指标，我们可以计算前7天的当前SLI，如表2-2所示。

表2-2. 过去7天中的SLI计算

指标	表达式
可用性	sum(rate(http_requests_total{host="api",status!="5.."}[7d])) / sum(rate(http_requests_total{host="api"})[7d])
延迟	histogram_quantile(0.9, rate(http_request_duration_seconds_bucket[7d])) histogram_quantile(0.99, rate(http_request_duration_seconds_bucket[7d]))

### 使用SLI计算入门级SLO

我们可以将这些SLI四舍五入为可管理的数量(例如，两个重要的可用性数字，或最多50 ms ^5^的延迟)，以获得我们的起始SLO。例如，在过去的四个星期中，API指标显示：

- 请求总数:3,663,253
- 成功请求总数:3,557,865(97.123%)
- 第90个百分点的延迟:432毫秒•第99个百分点的延迟:891毫秒

我们对其他SLI重复此过程，并为API创建一个建议的SLO，如表2-3所示。

5 50 ms，因为用户不太可能察觉到50 ms的延迟变化，但是适当的窗口显然取决于服务和用户。报表服务将不同于实时游戏。

表2-3. 为API建议的SLO

SLO类型	目标
可用性	97%
延迟	90%的请求 <450毫秒
延迟	99%的请求 <900毫秒

附录A提供了SLO文档的完整示例。本文档包括SLI实现，为简洁起见，在此省略。

根据提议的SLI，我们可以计算出这四个星期的错误预算，如表2-4所示。

**表2-4.四周内的错误预算**

SLO	允许的故障
97%可用性	109,897
90%的请求速度超过450毫秒	366,325
99%的请求速度超过900毫秒	36,632

## 选择适当的时间窗口

可以在各种时间间隔内定义SLO，并且可以使用滚动窗口或日历对齐的窗口(例如，一个月)。选择窗口时，需要考虑几个因素。

滚动窗口与用户体验更加接近:如果您在一个月的最后一天发生了大故障，则用户不会在下个月的第一天突然忘记它。我们建议将此时间段定义为整数周，以便它始终包含相同的周末数。例如，如果使用30天的窗口，则某些期间可能包括四个周末，而另一些期间则包括五个周末。如果周末流量与平日流量有显着差异，则出于不必要的原因，您的SLI可能会有所不同。

日历窗口与业务计划和项目工作更加紧密地结合在一起。例如，您可以每个季度评估您的SLO，以确定将下一个季度的项目人数集中在哪里。日历窗口还引入了一些不确定性元素:在本季度的中间，无法知道在本季度的剩余时间内您将收到多少个请求。因此，在本季度中期做出的决策必须推测您将在本季度剩余时间内花费多少错误预算。

较短的时间范围使您可以更快地做出决策:如果您错过了前一周的SLO，则进行细微的课程更正-例如，优先处理相关的错误-可以帮助避免在未来几周违反SLO。

较长的时间段适合进行更具战略性的决策:例如，如果您只能选择三个大型项目中的一个，那么您会迁移到高可用性分布式数据库，自动化您的前滚和回滚过程，还是将重复的应用部署到另一个区？您需要超过一周的数据来评估大型多季度项目。所需的数据量大致与为解决该问题而建议的工程工作量相称。

我们发现四个星期的滚动窗口是一个很好的通用间隔。

我们通过每周摘要(用于任务优先级)和季度总结报告(用于项目计划)来补充此时间范围。

如果数据源允许，则可以使用该提议的SLO计算该时间间隔内的实际SLO性能:如果您根据实际测量值设置了初始SLO，则设计达到了SLO。但是我们也可以收集有关分布的有趣信息。在过去的四个星期中，有没有几天我们的服务没有达到其SLO？这些天是否与实际事件相关？在那几天是否(或应该曾经)采取了一些措施来应对事故？

如果您没有日志，指标或其他任何历史性能源头，则需要配置数据源。例如，作为HTTP服务的低保真解决方案，您可以设置远程监控服务，该服务对服务执行某种定期的运行状况检查(ping或HTTP GET)并报告成功请求的数量。许多在线服务可以轻松实现这一方案。

## 获得利益相关者协议

为了使拟议的SLO有用且有效，您需要使所有利益相关者都同意：

- 产品经理必须同意这个阈值对用户来说足够好-低于该值的性能太低了，值得花时间进行修复。
- 产品开发人员必须同意，如果错误预算用尽了，他们将采取一些措施降低用户风险，直到服务恢复预算为止(如第31页的“建立错误预算策略”中所述)。
- 负责捍卫本SLO的负责生产环境的团队已同意，无需费力，琐事工作和疲倦就可以辩护-所有这些都会损害团队和服务的长期健康。

一旦所有这些点都达成共识，最困难的部分就完成了。<sup>26</sup>您已经开始了SLO之旅，其余步骤需要从这个起点开始进行迭代。

为了保护您的SLO，您将需要设置监控和警报(请参阅第5章)，以便工程师在错误威胁变为赤字之前及时收到错误预算威胁的通知。

### **建立错误预算政策**

拥有SLO后，就可以使用SLO得出错误预算。为了使用此错误预算，您需要一个策略概述在服务预算用尽时该怎么办。

获得所有关键利益相关者(产品经理，开发团队和SRE)批准的错误预算政策，是SLO是否适合目标的良好测试：

- 如果SRE认为SLO没有过多的琐事就无法辩护，那么他们可以为放宽某些目标提供理由。
- 如果开发团队和产品经理认为他们将增加用于修复可靠性的资源会使功能发布速度降至可接受的水平以下，那么他们也可以主张放宽目标。请记住，降低SLO也会降低SRE应对情况的数量；产品经理需要了解这一权衡。
- 如果产品经理在错误预算政策提示任何人解决问题之前，认为SLO将给大量用户带来不良体验，则SLO可能不够紧。

如果所有三个方面均不同意执行错误预算政策，则需要迭代SLI和SLO，直到所有涉众满意为止。决定如何前进，以及需要做出什么决定：更多数据，更多资源，或者对SLI或SLO进行更改？

当我们谈论执行错误预算时，是指一旦耗尽(或接近耗尽)错误预算，就应该采取措施以恢复系统的稳定性。

要制定错误预算执行决策，您需要先制定书面政策。此策略应涵盖在给定的时间段内服务耗尽了整个错误预算后必须采取的特定操作，并指定将采取哪些措施。共同所有者和行动可能包括：

- 在过去的四个星期中，开发团队将与可靠性问题相关的错误排在第一位。
- 开发团队只专注于可靠性问题，直到系统进入SLO之内。该责任伴随着高层批准，以推迟外部功能请求和任务。
- 为了减少更多停机的风险，生产变更冻结将暂停对系统的某些更改，直到有足够的错误预算来恢复更改为止。

有时，服务会消耗掉整个错误预算，但并非所有利益相关者都同意制定错误预算策略是适当的。如果发生这种情况，您需要返回到错误预算政策批准阶段。

### **记录SLO和错误预算政策**

适当定义的SLO应该记录在醒目的位置，其他团队和利益相关方可以对其进行审查。该文档应包括以下信息：

- SLO的作者，审阅者(检查其技术准确性)和批准者(对是否为正确的SLO做出商业决策)。
- 批准的日期以及下次审核的日期。
- 为读者提供上下文的服务的简短描述。
- SLO的细节：目标和SLI实施。
- 有关错误预算的计算和使用方式的详细信息。
- 数字背后的原理，以及它们是来自实验数据还是观察数据。即使SLO完全是临时性的，也应记录这一事实，以便将来阅读该文档的工程师不会基于临时性数据做出错误的决定。

您审阅SLO文档的频率取决于您SLO文化的成熟程度。开始时，您可能应该经常检查SLO，也许每个月都要检查一次。一旦确定了SLO的适当性，您就可以减少每季度甚至更少一次的审核。

错误预算政策也应记录在案，并应包括以下信息：

- 政策制定者，审核者和批准者
- 批准日期和下次审核日期
- 为读者提供上下文的服务的简要说明
- 为应对预算用尽而采取的行动
- 如果在计算上存在分歧或商定的行动是否适合这种情况，则应遵循明确的升级路径
- 根据受众的错误预算经验和专业知识水平，对错误预算进行概述可能会有所帮助。

有关SLO文档和错误预算策略的示例，请参见附录A。

### 仪表板和报告

除了已发布的SLO和错误预算政策文档之外，拥有报告和仪表板还可以提供有关服务的SLO合规性的及时快照，与其他团队进行沟通以及发现有问题的地方，这非常有用。

图2-3中的报告显示了几种服务的总体合规性：它们是否满足了上一年的所有季度SLO(括号中的数字表示已达到的目标数量，以及目标总数)，以及其SLI相对于上一季度和去年同一季度呈上升还是下降趋势。

Service ↑	2017				2018		Trends
	Q1 ↑	Q2 ↑	Q3 ↑	Q4 ↑	Q1 ↑	QoQ	YoY
FooService	✓ (1/1)	✓ (1/1)	✓ (1/1)	✓ (1/1)	✓ (1/1)	→	↗
ServiceApi		✓ (1/1)	✓ (1/1)	✓ (1/1)	✓ (1/1)	→	
ThingDoer	✓ (1/1)	✓ (1/1)	✓ (1/1)	✓ (1/1)	✓ (1/1)	↗	↗
NewService				✓ (1/1)	✓ (1/1)	↘	
BrokenService	✓ (9/9)	⚠ (84/87)	⚠ (122/123)	⚠ (148/150)	⚠ (169/172)	↘	↘

图2-3.SLO合规报告

具有显示SLI趋势的仪表板也很有用。这些仪表板指示您是否正在以比平常高的速度消耗预算，或者是否需要了解某些模式或趋势。

图2-4中的仪表板显示了该季度中途一个季度的错误预算。在这里，我们看到一个事件在两天内消耗了错误预算的15%左右。

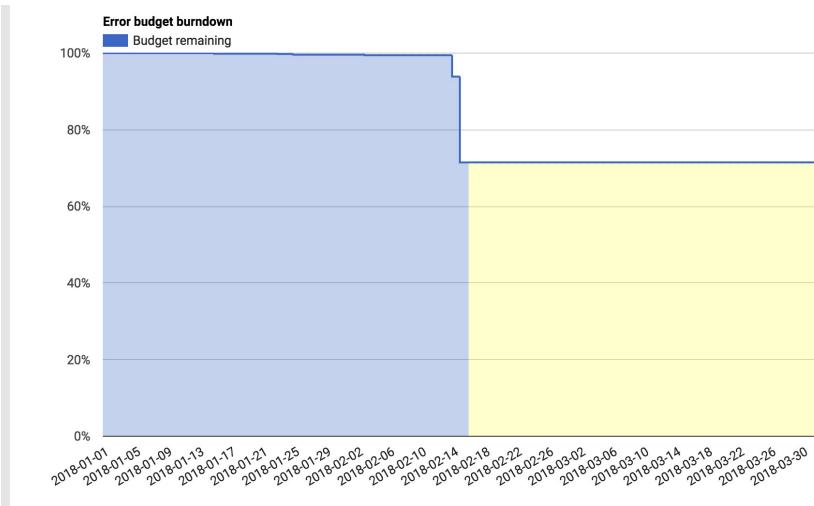


图2-4. 错误的预算信息中心

错误预算对于量化这些事件很有用-例如，“此中断消耗了我季度错误预算的30%，或“这是本季度发生的前三起事件，按它们所消耗的错误预算来排序”。

## 持续改进SLO目标

每项服务都可以从不断改进中受益。例如，这是ITIL中的中心服务目标之一。

在提高SLO目标之前，您需要有关用户对服务满意度的信息源。有多种选择：

- 您可以计算手动发现的停机，在公共论坛上发帖，支持故障单以及致电客户服务的次数。
- 您可以尝试衡量社交媒体上的用户情绪。
- 您可以向系统中添加代码以定期采样用户满意度。
- 您可以进行面对面的用户调查和样本。

可能性无穷无尽，最佳方法取决于您的服务。我们建议您从开始测量起点进行收集和迭代，这样成本最低。让您的产品经理将可靠性纳入他们与客户有关价格和功能的现有讨论中，是一个很好的起点。

### 提高您的SLO的质量

计算您手动检测到的中断。如果您已经为一些故障单提供了支持，也算进去。查看发生已知故障或事件的时间段。检查这些时间段是否与错误预算的急剧下降相关。同样，请查看您的SLI指示问题或您的服务退出SLO的时间。这些时间段是否与已知的故障或支持通知单增加相关？如果您熟悉统计分析，则[Spearman等级相关系数](#)可能是量化此关系的有用方法。

图2-5显示了每天创建需要支持的单据数量与当天错误预算中测得的损失的关系图。尽管并非所有故障单都与可靠性问题相关，但是故障单和错误预算损失之间存在相关性。我们看到两个异常值：一天只有5张故障单，而我们损失了10%的错误预算；一天有40张故障单，在那一天我们没有损失任何错误预算。两者都需要进一步调查。

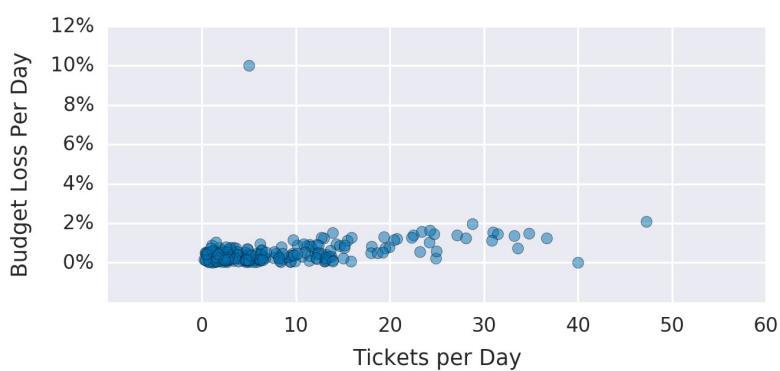


图2-5. 该图显示了每天支持的故障单数量与当天的预算损失

如果在任何SLI或SLO中都未捕获到您的某些中断和故障单，或者您的SLI下降和SLO未命中未映射到面向用户的问题，则表明您的SLO缺乏覆盖面。这种情况是完全正常的期望。您的SLI和SLO应该随着时间的流逝而变化，因为它们所代表的服务的现实已经发生了变化。不要害怕随着时间的推移来检查和完善它们！如果您的SLO缺乏覆盖范围，您可以采取几种措施：

### 更改您的SLO

如果您的SLI指示问题，但是您的SLO没有提示任何人注意或响应，则您可能需要收紧SLO。

- 如果该日期的事件足够大以至于需要解决，请查看相关时间段内的SLI值。计算在这些日期SLO会导致什么通知。将该SLO应用于您的历史SLI，并查看此调整还将捕获哪些其他事件。如果降低精度，使团队必须不断响应不重要的事件，那么提高系统的召回率就毫无意义。<sup>27</sup>
- 同样，对于误报的日子，请考虑放宽SLO。

如果在任一方向上更改SLO都会导致误报或误报过多，那么您还需要改进SLI实施。

### 更改您的SLI实施

有两种方法可以更改您的SLI实施方式:将度量移动到更靠近用户的位置以提高度量的质量，或者提高覆盖率，以便您捕获更高比例的用户交互。例如:

- 与其在服务器上衡量成功/等待时间，不如在负载平衡器或客户端上衡量它。
- 与其通过简单的HTTP GET请求来评估可用性，使用运行状况检查处理程序来行使系统的更多功能，或者使用执行所有客户端JavaScript的测试。

#### 制定有抱负的SLO

有时您确定需要更严格的SLO才能使用户满意，但是要改进产品以达到该SLO要求花费一些时间。如果实施更严格的SLO，您将永远缺乏SLO，并受制于错误预算政策。在这种情况下，您可以使精简的SLO成为理想的SLO ---与您当前的SLO一起进行测量和跟踪，但是在错误预算策略中明确指出它不需要采取措施。这样，您就可以跟踪达到理想的SLO的进度，但不会处于永久的紧急状态。

#### 迭代

有很多不同的迭代方法，您的审查会议将确定许多潜在的改进。选择最有可能带来最高投资回报的选项。特别是在最初的几次迭代中，错误会变得更快，更低成本。这样做可以减少指标的不确定性，并帮助您确定是否需要更昂贵的指标。根据需要重复多次。

## 使用SLO和错误预算进行决策

一旦有了SLO，就可以开始使用它们进行决策。

显而易见的决定始于不满足SLO时的处理方式，也就是耗尽了错误预算后的处理方式。如前所述，错误预算政策应涵盖耗尽错误预算时的适当措施。通用策略包括停止功能启动，直到该服务再次位于SLO中为止，或者将部分或全部工程时间专门用于处理与可靠性相关的错误。

在极端情况下，团队可以在获得高层批准后宣布紧急情况，以降低所有外部需求(例如来自其他团队的请求)的优先级，直到该服务满足退出标准为止。通常情况下，该服务在SLO范围内并且您已经采取了减少SLO错过机会的机会。这些步骤可能包括改善监控，改善测试，消除危险依赖性或重新配置系统以消除已知的故障类型。

您可以根据所消耗的错误预算的比例来确定事件的规模，并使用此数据来确定需要更深入调查的最关键事件。

例如，假设发布一个新的API版本会导致100%的NullPointerException，直到可以在四个小时后还原系统为止。<sup>28</sup> 检查原始服务器日志表明该问题导致了14,066个错误。使用先前我们97%的SLO中的数字以及109,897个错误的预算，此单个事件使用了我们错误预算的13%。

也许存储我们唯一宿主状态数据库的服务器发生故障，并且从备份还原需要20个小时。我们估计(根据该时期的历史流量)该中断导致我们发生72,000个错误，占错误预算的65%。

想象一下，我们的示例公司在五年内只有一台服务器发生故障，但是通常会遇到每年需要回滚的两到三个不良版本。我们可以估计，平均而言，错误推送的错误预算 是数据库故障的两倍。这些数字证明，解决发布问题比调查服务器故障方面的资源投入要多得多。

如果服务运行正常且几乎不需要监督，那么可能是时候将服务移至较少的动手支持层了。您可能会继续提供事件响应管理和高级监督，但是您不再需要每天与产品紧密联系。因此，您可以将精力集中在需要更多SRE支持的其他系统上。

表2-5根据三个关键维度提供了建议的行动方案：

- 针对SLO的表现
- 操作该服务所需的劳动量
- 客户对服务的满意程度

表2-5.SLO决策矩阵

SLOs	琐事	客户满意度	动作
达标	低	高	选择(a)放宽发布和部署过程并提高速度，或者(b)退出参与，将工程时间集中在需要更高可靠性的服务上。
达标	低	低	收紧SLO
达标	高	高	如果警报产生误报，请降低灵敏度。否则，请暂时放松SLO(或减轻负担)并修复产品和/或改进自动故障缓解功能。
达标	高	低	收紧SLO
未捕获	低	高	放松SLO。
未捕获	低	低	提高警报灵敏度
未捕获	高	高	放松SLO
未捕获	高	低	减轻琐事并修复产品和/或改进自动故障缓解功能。

## 高级主题

一旦有了健康和成熟的SLO和错误预算文化，您就可以继续改进和完善衡量和讨论服务可靠性的方式。

### 模拟用户旅程

尽管本章讨论的所有技术都将对您的组织有所帮助，但最终SLO应该以改善客户体验为中心。因此，您应该按照以用户为中心的操作编写SLO。

您可以使用“关键用户旅程”来帮助捕获客户的体验。关键的用户旅程是一系列任务，这些任务是给定用户体验的核心部分，也是服务的重要方面。例如，对于在线购物体验而言，关键的用户旅程可能包括：

- 搜索产品
- 将产品添加到购物车
- 完成购买

这些任务几乎肯定不会很好地映射到您现有的SLI。每个任务都需要多个复杂的步骤，这些步骤在任何时候都可能失败，并且从日志中推断这些操作的成功(或失败)可能非常困难。(例如，您如何确定用户是否在第三步失败，还是只是在另一个标签中被猫视频分散了注意力？)但是，我们必须先确定对用户重要的内容，然后才能开始确保服务的各个方面都是可靠的。

一旦确定了以用户为中心的事件，就可以解决度量它们的问题。您可以通过将不同的日志事件结合在一起，使用高级JavaScript探测，使用客户端工具或其他一些过程来进行度量。一旦可以衡量一个事件，它便会成为另一个SLI，您可以与现有的SLI和SLO一起对其进行跟踪。关键的用户旅程可以提高您的召回率，而不会影响您的准确性。

### 评分互动的重要性

并非所有请求都被视为相等。来自移动应用程序的用于检查帐户通知(通过日常管道生成通知)的HTTP请求对您的用户很重要，但不如广告客户的与计费相关的请求那么重要。

我们需要一种区分某些类别的请求与其他类别的方法。您可以使用*bucketing*来完成此操作-即向SLI添加更多标签，然后将不同的SLO应用于这些不同的标签。表2-6给出了一个示例。表2-6.按等级进行分组

客户等级	可用性SLO
高级	99.99%
免费	99.9%

您可以按期望的响应速度拆分请求，如表2-7所示。

表2-7.通过预期的响应性进行分类

响应能力	延迟SLO
交互式(即，请求阻止页面加载)	90%的请求在100毫秒内完成
CSV下载	90%的下载在5秒内开始

如果您有可用的数据将SLO独立地应用于每个客户，则可以跟踪任何给定时间在SLO中的客户数量。请注意，此数字可能变化很大-发送少量请求的客户将具有100%的可用性(因为他们很幸运没有失败)或非常低的可用性(因为他们遇到的一个失败是他们请求的很大一部分)。个别客户可能会出于无趣的原因而无法满足他们的SLO，但是总的来说，跟踪影响大量客户SLO合规性的问题可能是一个有用的信号。

### 建模依赖

大型系统具有许多组件。单个系统可以具有表示层，应用程序层，业务逻辑层和数据持久层。这些层中的每一层都可以包含许多服务或微服务。

当您最关心的是实现覆盖整个堆栈的以用户为中心的SLO时，SLO也是在堆栈中不同组件之间协调和实现可靠性要求的有用方法。

例如，如果单个组件是对于特别高价值的交互的关键依赖项<sup>29</sup>，则其可靠性保证应至少与该依赖动作的可靠性保证一样高。运行该特定组件的团队需要以与总体产品SLO相同的方式拥有和管理其服务的SLO。

如果特定组件具有固有的可靠性限制，则SLO可以传达该限制。如果依赖于此的用户旅程需要的可用性级别高于该组件无法合理提供的可用性，则您需要针对这种情况进行设计。您可以使用其他组件，也可以添加足够的防御措施(缓存，脱机存储和转发处理，正常降级等)来处理该组件中的故障。

尝试用数学方法解决这些问题可能很诱人。如果您的服务在单个区域中提供99.9%的可用性，并且您需要99.95%的可用性，那么只需在两个区域中部署该服务就可以满足该要求。两种服务同时停机的可能性非常低，以至于两个区域应提供99.9999%的可用性。但是，这种推论假定这两种服务都是完全独立的，这几乎是不可能的。您的应用程序的两个实例将具有通用的依赖关系，通用的故障域，共享的命运以及全局控制平面，所有这些都可能导致两个系统都中断，无论其设计和管理的谨慎程度如何。除非仔细枚举并说明每种依赖性和故障模式，否则任何此类计算都是具有欺骗性的。

当故障是由另一个团队处理的依赖性引起的时，关于错误预算策略应如何解决错过SLO，有两种思路：

- 您的团队不应暂停发布或花更多的时间来提高可靠性，因为您的系统没有引起问题。
- 您应该执行更改冻结，以最大程度地减少将来发生中断的机会，无论造成中断的原因如何。

第二种方法将使您的用户更快乐。您在应用此原理方面有一定的灵活性。根据中断和依赖性的性质，冻结更改可能不切实际。确定最适合您的服务及其依赖项的内容，并在记录在案的错误预算中记录后继决定。有关如何实际操作的示例，请参阅附录B中的示例错误预算策略。

### 通过放宽SLO进行实验

您可能需要试验应用程序的可靠性，并评估可靠性方面的哪些变化(例如，将延迟添加到页面加载时间中)对用户行为(例如，完成购买的用户百分比)产生了可衡量的不利影响。我们建议仅在确信有足够的错误预算时才执行这种分析。延迟，可用性，客户，业务领域和竞争(或缺乏竞争)之间存在许多微妙的相互作用。要有选择地降低感知到的客户体验是Rubicon的考虑点，即使有的话，也要经过深思熟虑。

尽管此练习看起来很可怕(没人想失去销售！)，但是通过进行此类实验可以获得的知识将使您以可能导致将来更好的性能(和更高的销售！)的方式改善服务。此过程可以使您数学上确定关键业务指标(例如，销售)和可衡量的技术指标(例如，延迟)之间的关系。如果是这样，您将获得非常有价值的数据，可用于为以后的服务做出重要的工程决策。

此练习不应是一次性的活动。随着服务的发展，客户的期望也将随之提高。确保定期检查关系的持续有效性。

这种分析也是有风险的，因为您可能会误解所获得的数据。例如，如果您人为地将页面速度降低了50毫秒，并且注意到没有相应的转换损失发生，则可能会得出结论，您的延迟SLO太严格了。但是，您的用户可能不满意，只是目前缺少您的服务的替代品。一旦竞争对手出现，您的用户就会离开。确保测量正确的指标，并采取适当的预防措施。

## 结论

本书涵盖的每个主题都可以与SLO联系起来。现在，您已经阅读了本章，我们希望您也同意，即使是部分形式化的SLO(清楚地说明了您对用户的承诺)也提供了一个框架，可以更清晰地讨论系统行为，并可以在服务失败时帮助您找到可采取的补救措施。达到期望。

总结一下：

- SLO是衡量服务可靠性的工具。
- 错误预算是平衡可靠性与其他工程工作的工具，也是决定哪些项目影响最大的好方法。
- 您应该立即开始使用SLO和错误预算。

有关示例SLO文档和示例错误预算策略，请参阅附录A和B。

<sup>22</sup>. 关于术语的注释: 在本章中，我们始终使用"可靠性"一词来谈论服务在其所有SLI方面的性能。这可能表示许多情况，例如可用性或延迟。 ↵

<sup>23</sup>. 这不同于服务水平协议(SLA)，后者是一种商业合同，当您的用户非常不满意时，您必须以某种方式对其进行赔偿。 ↵

<sup>24</sup>. 有关将依赖项分解为服务可靠性的更多详细信息，请参阅Ben Treynor，Mike Dahlin，Vivek Rau和Betsy Beyer，“服务可用性的微积分”，ACM队列15，否。2(2017)，<https://queue.acm.org/detail.cfm?id=3096459>。 ↵

<sup>25</sup>. 如果您在一个日历周期(例如一个季度)内衡量SLO，那么如果它基于流量等不可预测的指标，那么您可能不知道该季度末的预算将是多少。有关报告期间的更多讨论，请参见第29页“选择合适的时间窗口”。 ↵

<sup>26</sup>. 免责声明: 您将来可能会遇到更艰巨的任务。 ↵

<sup>27</sup>. Recall是SLI捕获的重大影响用户事件的比例。Precision是SLI捕获的严重影响用户的事件的比例。 ↵

<sup>28</sup>. 这里值得重申的是，错误预算是用户满意度的近似值。每30天四小时的中断可能会导致不满意的用户少于每30天四次单独的一小时中断，这反过来会导致不满意的用户少于0.5%的恒定错误率，但是我们的错误预算将相同。这些阈值因服务而异。 ↵

<sup>29</sup>. 如果依赖项不可用，则意味着依赖也很关键。 ↵

## 第3章

### SLO工程案例研究

本·麦考马克(Evernote)和

威廉·邦纳尔(家得宝)与加勒特·普拉斯基(Evernote),亚历克斯·希达戈,贝茜·拜尔  
和戴夫·雷辛

尽管SRE的许多宗旨是在Google的大门内塑造的，但其原则却长期存在于我们的门外。并行发现了许多标准的Google SRE惯例，或者以其他方式被业内其他许多组织所采用。

SLO是SRE模型的基础。自从我们启动客户可靠性工程(CRE)团队(一组经验丰富的SRE来帮助Google Cloud Platform(GCP)客户构建更可靠的服务)以来，几乎每个客户互动都始于SLO。

在这里，我们介绍了两个不同的公司讲述的两个故事，概述了他们与Google CRE团队一起采用基于SLO和错误预算的方法的过程。有关SLO和错误预算的更一般性讨论，请参见本书的第2章和第一本书的第3章。

### Evernote的SLO故事

由Evernote的Ben McCormack提供

Evernote是一个跨平台的应用程序，可帮助个人和团队创建，组合和共享信息。该平台在全球拥有超过2.2亿用户，我们存储了超过120亿条信息-包括基于文本的便笺，文件和附件/图像。在幕后，Evernote服务由750多个MySQL实例支持。

我们在Evernote中引入了SLO的概念，这是更广泛的技术改造的一部分，旨在提高工程设计速度，同时保持服务质量。我们的目标包括：

- 将工程重点从数据中心毫无区别的繁重工作转移到客户真正关心的产品工程工作上。为此，我们停止了运行物理数据中心，并转移到公共云。
- 修改运营和软件工程师的工作模型，以支持功能速度的提高，同时保持整体服务质量。
- 改进我们对SLA的看法，以确保我们更加关注故障如何影响我们庞大的全球客户群。

对于许多行业的组织来说，这些目标可能看起来很熟悉。尽管没有一种单一的方法可以全面进行这些类型的更改，但我们希望分享我们的经验将为面临类似挑战的其他人提供有价值的见解。

为什么Evernote采用SRE模型？

过渡开始时，Evernote的特点是传统的运维分离:运维团队保护生产环境的神圣性，而开发团队则负责为客户开发新产品功能。这些目标通常是冲突的:开发团队受到冗长的运维需求的束缚，而运维团队则因新规范在生产中引入新问题而感到沮丧。当我们在这两个目标之间疯狂转换时，运维和开发团队之间建立了沮丧而紧张的关系。我们希望找到一个更快乐的媒介，以更好地平衡参与团队的需求变化。

在过去的五年多的时间里，我们试图以各种方式解决传统二分法中的空白。在尝试了"您编写，你运行它"(开发)模型和"您编写后，我们为您运行"模型(操作)之后，我们转向了以SLO为中心的SRE方法。

那么什么促使Evernote朝这个方向发展呢？

在Evernote，我们将运维和开发的核心学科视为工程师可以专门研究的独立专业领域。一种途径是与客户近24/7的持续交付服务有关。另一个与服务的扩展和发展有关，以满足将来客户的需求。近年来，随着诸如SRE和DevOps之类的运动强调将软件开发应用于运营，这两个学科已经相互接近。(随着数据中心自动化的发展和公共云的发展，这种融合得到了进一步的发展，这两者都为我们提供了一个可以完全由软件控制的数据中心。)另一方面，全栈所有权和持续部署越来越多地应用于软件开发。

我们之所以被SRE模型吸引，是因为它完全拥抱并接受了运营与开发之间的差异，同时鼓励团队朝着一个共同的目标努力。它不会尝试将运维工程师转变为应用程序开发人员，反之亦然。而是提供了共同的参考框架。根据我们的经验，错误预算/SLO方法导致两个团队在陈述相同时做出相似的决定，因为它消除了对话中的很多主观性。

### SLO的介绍:进行中的旅程

我们旅程的第一部分是从物理数据中心到Google Cloud Platform的迁移。<sup>30</sup>

Evernote服务在GCP上启动并运行并稳定后，便引入了SLO。我们的目标是双重的:

- 围绕Evernote SLO内部调整团队，确保所有团队都在新框架内工作。
- 将Evernote的SLO纳入我们与Google Cloud团队的合作方式，该团队现在负责我们的基础架构。由于我们现在在整体模型中有了新的合作伙伴，因此我们需要确保迁移到GCP不会稀释或掩盖我们对用户的承诺。

在活跃使用SLO大约9个月之后，Evernote已经在其SLO实践的第3版中使用！

在了解SLO的技术细节之前，从客户的角度开始对话非常重要:您要坚持哪些承诺？与大多数服务类似，Evernote具有许多功能和选项，可供我们的用户以各种创造性方式使用。我们想确保我们最初专注于最重要和最常见的客户需求:Evernote服务的可用性，供用户跨多个客户端访问和同步其内容。我们的SLO旅程就是从这个目标开始的。通过专注于正常运行时间，我们使第一个SLO变得简单。使用这种简单的第一种方法，我们可以清楚地说明我们正在测量的内容以及如何进行测量。

我们的第一份SLO文件包含以下内容:

#### SLO的定义

这是正常运行时间的尺度:为某些服务和方法设置的按月窗口正常运行时间为99.95%。我们根据与内部客户支持和产品团队的讨论以及(更重要的是)用户反馈的讨论选择了这个数字。我们特意选择将我们的SLO绑定到一个日历月而不是一个滚动期，以使我们在进行服务审核时保持专注和有条理。

## 测量什么，以及如何测量

### 测量内容

我们指定了可以调用的服务端点，以测试服务是否按预期运行。在我们的例子中，我们的服务内置了一个状态页面，该页面可以执行大部分堆栈，如果一切正常，则返回200状态代码。

### 如何测量

我们需要一个探针来定期调用状态页面。我们希望该探针完全位于环境之外并且与我们的环境无关，以便我们可以测试我们的所有组件，包括负载平衡堆栈。我们的目标是确保我们正在测量GCP服务和Evernote应用程序的所有故障。但是，我们不希望随机的互联网问题触发误报。我们选择使用专门从事构建和运行此类探针的第三方公司。我们选择了Pingdom，但市场上还有很多其他产品。我们进行以下测量：

- **探针频率:** 我们每分钟都会轮询前端节点。
- **探针位置:** 此设置是可配置的。我们目前在北美和欧洲使用多个探针。
- **"停机"的定义:** 如果探针检查失败，则将该节点标记为"未确认关闭"，然后由第二个地理位置独立的探针执行检查。如果第二次检查失败，则将该节点标记为向下以用于SLO计算。只要连续的探测请求注册错误，该节点将继续被标记为已关闭。

### 如何根据监控数据计算SLO

最后，我们仔细记录了如何根据从Pingdom收到的原始数据计算SLO。例如，我们指定了如何维护窗口的方法：我们不能假设所有亿万用户都知道我们发布的维护窗口。因此，不知情的用户会将这些窗口视为无法解释的一般停机时间，因此我们的SLO计算将维护视为停机时间。

一旦定义了我们的SLO，就必须对它们进行一些处理。我们希望SLO推动软件和运维的变化，这将使我们的客户更加快乐并使他们满意。如何做到最好？

我们使用SLO/错误预算概念作为分配资源的方法。例如，如果我们错过了上个月的SLO，那么这种行为有助于我们优先考虑相关的修复，改进和错误修复。我们保持简单：Evernote和Google的团队每月对SLO的性能进行一次审查。在本次会议上，我们将回顾上个月的SLO绩效，并对任何中断进行深入分析。根据过去一个月的分析，我们设置了可能无法通过常规的"根本原因"分析过程获得的改进措施。

在整个过程中，我们的指导原则一直是"完美是良好的敌人"。即使SLO不够完善，它们也足以指导随着时间的推移进行改进。"完美"的SLO可以衡量用户与我们服务的所有可能互动，并考虑所有边缘情况。尽管这是一个好主意，但要花很多时间才能实现（如果可以实现完美），这是我们用来改善服务的时间。取而代之的是，我们选择了一个初始SLO，该SLO涵盖了大多数（但不是全部）用户交互，这是服务质量的良好代理人。

自从我们开始以来，根据内部服务审查的信号以及对影响客户的停机的响应，我们对SLO进行了两次修订。因为我们一开始并不是针对完美的SLO，所以我们很乐意进行更改以更好地与业务保持一致。除了我们每月对Evernote/Google进行SLO绩

效审查之外，我们还确定了六个月的SLO审查周期，该周期可以在过于频繁地更改SLO和使它们变得陈旧之间取得适当的平衡。在修订我们的SLO时，我们还了解到平衡想要的测量和可能的测量很重要。

自引入SLO以来，我们的运维和开发团队之间的关系已经微妙但明显改善。团队现在拥有共同的成功衡量标准：消除对服务质量(QoS)的人为解释，可以使两个团队保持相同的观点和标准。仅举一个例子，当我们不得不在2017年的压缩时间表中促进多个版本发布时，SLO提供了一个共同点。当我们解决一个复杂的错误时，产品开发要求我们在多个单独的窗口中分配正常的每周发布时间，每个窗口都可能影响客户。通过对问题应用SLO计算并从场景中消除人类的主观性，我们能够更好地量化对客户的影响，并将我们的发布窗口从五个减少到两个，以最大程度地减少客户的痛苦。

### 打破客户与云提供商之间的SLO墙

客户和云提供商之间的虚拟墙似乎是自然的或不可避免的。Google在我们运行Evernote的GCP平台上拥有SLO和SLA(服务水平协议)，而Evernote拥有自己的SLO和SLA。不能总是期望两个这样的工程团队会被告知彼此的SLA。

Evernote从未想要过这样的隔离墙。当然，我们可以根据基础GCP指标将SLO和SLA建立在隔墙上。相反，从一开始，我们希望Google了解哪些性能特征对我们最重要，以及为什么。我们希望使Google的目标与我们的目标保持一致，并且两家公司都将Evernote的可靠性成功与失败视为共同的责任。为此，我们需要一种方法：

- 调整目标
- 确保我们的合作伙伴(在本例中为Google)确实了解对我们重要的事情
- 分享成功与失败

大多数服务提供商都针对其云服务管理发表的SLO/SLA。在这种情况下工作很重要，但是不能从整体上说明我们的服务在云提供商环境中的运行状况。

例如，给定的云提供商可能在全球范围内运行成千上万个虚拟机，它们可以管理这些虚拟机的正常运行时间和可用性。GCP承诺Compute Engine(即其虚拟机)的可用性为99.95%。即使GCP SLO图表为绿色(即高于99.95%)，Evernote对同一SLO的看法也可能有很大不同：因为我们的虚拟机覆盖区仅占全球GCP数量的一小部分，所以中断对我们区域是孤立的(或孤立的由于其他原因)可能会在整体汇总到全球范围内“丢失”。

为了纠正这种情况，我们与Google分享了SLO和SLO的实时性能。结果，Google CRE团队和Evernote都使用相同的性能仪表板。这似乎很简单，但是事实证明，这是驱动真正以客户为中心的行为的强大方法。因此，Google不会收到通用的“Service X运行缓慢”类型的通知，而是向我们提供了更特定于我们环境的通知。例如，除了通用的“GCP负载平衡环境今天运行缓慢”之外，我们还将被告知，此问题对Evernote的SLO造成了5%的影响。这种关系还可以帮助Google内部的团队了解他们的行为和决定如何影响客户。

这种双向关系也为我们提供了一个非常有效的框架来支持重大事件。在大多数情况下，P1-P5故障单和常规支持渠道的常规模型效果很好，可以使我们保持良好的服务并与Google建立良好的关系。但是我们都清楚，有时候P1故障单(“对我们业务的重大影响”)还不够-您的整个服务都在网上，您将面临更大的业务影响。

在这样的时刻，我们共同的SLO和与CRE团队的关系得以实现。我们有一个共同的理解，即如果SLO的影响足够高，双方将把该问题视为经过特殊处理的P1故障单。通常，这意味着Evernote和Google的CRE团队使用共享会议迅速动员。Google CRE团队监控我们共同定义并达成的SLO，使我们能够在优先级划分和适当的响应方面保持同步。

### 当前状态

在积极使用SLO约9个月之后，Evernote已经在其SLO实践的第3版中使用。下一版本的SLO将在我们简单的正常运行时间SLO的基础上发展。我们计划开始探究单个API调用，并考虑客户端/客户端对度量/性能的看法，以便我们可以更好地表示用户QoS。

通过提供衡量QoS的标准和定义方式，SLO使Evernote可以更好地关注我们的服务运行方式。现在，我们可以在内部以及与Google进行数据驱动的对话，以讨论中断的影响，这使我们能够推动服务的改进，最终使支持团队更有效，客户更满意。

## 家得宝(Home Depot)的SLO故事

由Home Depot的William Bonnell提供

家得宝(THD)是世界上最大的家居装饰零售商:我们在北美拥有2200多家商店，每个商店都拥有35,000多种独特产品(并在线提供了超过150万种产品)。我们的基础架构托管着各种软件应用程序，这些软件应用程序支持近40万名员工，每年处理超过15亿笔客户交易。这些商店与全球供应链和一个电子商务网站紧密集成，每年访问量超过20亿。

在最近一次旨在提高软件开发速度和质量的运营方法更新中，THD不仅转向敏捷软件开发，而且还改变了我们设计和管理软件的方式。我们从支持大型单片软件包的集中支持团队转变为由小型独立运维的软件开发团队领导的微服务架构。结果，我们的系统现在具有较小块的不断变化的软件，这些软件也需要在整个堆栈中进行集成。

我们向微服务的迁移得到了全栈所有权的新“自由和责任文化”的补充。这种方法使开发人员可以在需要时自由地推送代码，还可以使他们共同负责服务的操作。为了使这种共有制的工作模式，运维和开发团队需要说一种通用的语言，这种语言可以促进问责制并跨越复杂性:服务水平目标(SLO)。相互依赖的服务需要了解以下信息:

- 您的服务有多可靠？它是为三个9，三个半9或四个9(或更高级)而构建的？有计划的停机时间吗？
- 我可以期望达到什么样的延迟？
- 您能处理我要发送的请求量吗？您如何处理过载？您的服务是否已逐步达到其SLO？

如果每个服务都可以为这些问题提供透明且一致的答案，那么团队将对他们的依存关系有清晰的了解，从而可以更好地进行沟通，并提高团队之间的信任度和责任感。

### SLO文化项目

在我们开始转变服务模式之前，Home Depot没有SLO的文化。监控工具和仪表板数量很多，但分布在各处，并且不会随时间推移跟踪数据。我们并非总是能够将服务定位在出现中断的根源。通常，我们开始在用户界面服务处进行故障排除，然后进行反向工作，直到发现问题为止，这浪费了无数小时。如果某项服务需要计划内停机，则其相关服务会感到惊讶。如果团队需要构建3个半9的服务，那么他们将不知道他们所依赖的服务是否可以为他们提供更好的正常运行时间(四个9)。这些脱节导致了我们的软件开发和运营团队之间的困惑和失望。

我们需要通过建立SLO的共同文化来解决这些脱节问题。这样做需要一种总体策略来影响人员，流程和技术。我们的工作涵盖了四个大致领域：

#### 普通白话

在THD上下文中定义SLO。定义如何以一致的方式测量它们。

#### 布道

在整个公司范围内传播信息。

- 制作培训材料以销售SLO为何重要，公司各处路演，内部博客以及T恤和贴纸等促销材料。
- 招募一些早期采用者来实施SLO，并向其他人展示其价值。
- 建立醒目的缩写(VALET；稍后讨论)，以帮助传播该想法。
- 创建培训计划(FiRE学院:可靠性工程的基础)，对开发人员进行SLO和其他可靠性概念的培训。[31](#)

#### 自动化

为减少采用过程中的摩擦，请实施一个指标收集平台，以自动收集部署到生产中的任何服务的服务水平指标。这些SLI以后可以更容易地转换为SLO。

#### 奖励

为所有发展经理制定年度目标，以设置和衡量其服务的SLO。

建立一个通用的语言对于使每个人意见统一至关重要。我们还希望保持此框架尽可能简单，以帮助该想法更快地传播。首先，我们仔细研究了跨各种服务监控的指标并发现了一些模式。每个服务都监控其“流量”，“等待时间”，“错误”和“利用率”的某种形式-与Google SRE的[四个黄金信号](#)密切相关的指标。此外，许多服务对正常运行时间或可用性的监控明显不同于错误。不幸的是，所有指标的整体监控都不一致，名称不同或数据不足。

我们的服务都没有SLO。我们的生产系统最接近面向客户SLO的指标是支持故障单。我们衡量部署到商店的应用程序的可靠性的主要(也是唯一的)方法是跟踪内部支持部门收到的支持电话数量。

## 我们的第一批SLO

我们无法为系统中可以衡量的各个方面创建SLO，因此我们必须决定哪些指标或SLI也应具有SLO。

#### API调用的可用性和延迟

我们决定，每个微服务必须具有由其他微服务调用的API调用可用性和延迟的SLO。例如，购物车微服务调用库存微服务。对于这些API调用，库存微服务发布了SLO，购物车微服务(以及其他需要库存的微服务)可以咨询SLO，以确定库存微服务是否可以满足其可靠性要求。

### 基础设施利用率

THD的团队以不同的方式衡量基础架构的利用率，但是最典型的衡量标准是一分钟粒度的实时基础架构利用率。由于某些原因，我们决定不设置利用率SLO。首先，微服务并不需要过分关注此指标-您只要使用可处理的流量，微服务启动，快速响应，不会引发错误，您的用户就不会真正在意利用率。您不会有容量不足的危险。此外，我们即将迁移到云意味着利用率将不再是问题，因此成本规划将使容量规划显得逊色。(我们仍然需要监控利用率和执行容量规划，但是我们不需要将其包含在我们的SLO框架中。)

### 流量

由于THD尚未具备容量规划的文化，因此我们需要一种机制，供软件和运维团队交流其服务可以处理的容量。流量很容易定义为对服务的请求，但是我们需要确定是否应该跟踪每秒的平均请求，每秒的峰值请求或在报告时间段内的请求量。我们决定跟踪所有这三个指标，并让每个服务选择最合适的目标。我们辩论是否要为流量设置SLO，因为该指标取决于用户行为，而不是我们可以控制的内部因素。最终，我们决定作为零售商，我们需要针对黑色星期五之类的高峰确定服务规模，因此我们根据预期的高峰容量设置了SLO。

### 潜伏

我们让每个服务定义其SLO的延迟，并确定在哪里对其进行最佳测量。我们唯一的要求是，服务应该用黑盒监控来补充我们常见的白盒性能监控，以捕获由网络或其他层(如微服务外部发生故障的缓存和代理)引起的问题。我们还认为，百分数比算术平均值更合适。至少，服务需要达到90%的目标；面向用户的服的服务的首选目标是95%和/或99%。

### 错误

错误的解释有些复杂。由于我们主要处理Web服务，因此我们必须标准化什么构成错误以及如何返回错误。如果Web服务遇到错误，我们自然会在HTTP响应代码上进行标准化：

- 服务不应在2xx响应的正文中指示错误；而是应该抛出4xx或5xx。
- 由服务问题(例如，内存不足)引起的错误应引发5xx错误。
- 由客户端引起的错误(例如，发送格式错误的请求)应引发4xx错误。

经过深思熟虑，我们决定同时跟踪4xx和5xx错误，但仅将5xx错误用于设置SLO。与我们针对其他SLO相关元素的方法类似，我们将此维度保持通用，以便不同的应用程序可以在不同的上下文中利用它。例如，除HTTP错误外，批处理服务的错误可能是无法处理的记录数。

### 故障单

如前所述，故障单最初是我们评估大多数生产软件的主要方式。由于历史原因，我们决定继续与其他SLO一起跟踪故障单。您可以将此指标视为类似于"软件操作级别"之类的指标。

## VALET

我们将新的SLO总结为一个方便的缩写:VALET

### 容量(流量)

我的服务可以处理多少业务量？

### 可用性

我需要时可以启动服务吗？

### 延迟

使用该服务时，服务是否快速响应？

### 错误

使用该服务时是否抛出错误？

### 故障单

服务是否需要人工干预才能完成我的请求？

## 宣传SLO

带着易于记忆的首字母缩写，我们着手向企业宣传SLO:

- 为什么SLO很重要
- SLO如何支持我们的"自由与责任"文化
- 应该测量什么
- 做什么去处理结果

由于开发人员现在要负责其软件的运行，因此他们需要建立SLO，以展示其构建和支持可靠软件的能力，还需要与服务的消费者和面向客户服务的产品经理进行沟通。但是，大多数受众不熟悉SLA和SLO之类的概念，因此需要对他们进行新的VALET框架教育。

由于我们需要获得高级管理层的支持才能加入SLO，因此我们的教育运动始于高级领导层。然后，我们与开发团队一对一会晤，以拥护SLO的价值。我们鼓励团队从其自定义的指标跟踪机制(通常是手动的)过渡到VALET框架。为了保持发展势头，我们每周以VALET格式发送SLO报告，并与有关高级可靠性概念和从内部事件中学到的经验教训一起进行评论。这也有助于制定业务指标，例如，根据VALET创建的采购订单(容量)或未能处理的采购订单(错误)。

我们还通过多种方式扩大了宣传的范围:

- 我们建立了一个内部WordPress网站来托管有关VALET和可靠性的博客，并链接到有用的资源。
- 我们进行了内部技术讲座(包括Google SRE演讲嘉宾)，讨论了一般的可靠性概念以及如何使用VALET进行测量。
- 我们举办了一系列VALET培训讲习班(以后将演变为FiRE学院)，并向想要参加的任何人开放了邀请。这些讲习班的出席人数持续了几个月。
- 我们甚至创建了VALET笔记本电脑贴纸和T恤，以支持全面的内部营销活动。

很快，公司中的每个人都了解了VALET，我们的SLO新文化开始流行起来。SLO的实施甚至开始正式纳入THD对开发经理的年度绩效评估中。虽然每周大约有50个服务定期按其SLO进行捕获和报告，但我们仍将临时指标存储在电子表格中。尽管VALET的想法像野火一样流行，但我们需要使数据收集自动化以促进广泛采用。

## 自动VALET数据收集

现在，我们的SLO文化已经站稳脚跟，但自动化VALET数据收集将加速SLO的采用。

### TPS报告

我们建立了一个框架来自动捕获已部署到我们新GCP环境中的任何服务的VALET数据。我们称此框架为TPS报告，是我们用来进行容量和性能测试(每秒事务处理)的术语的玩法，当然，幽默<sup>32</sup>的解释是多个管理者可能希望查看此数据。我们在GCP的BigQuery数据库平台上构建了TPS报告框架。我们的网络服务前端生成的所有日志均被馈入BigQuery中，以供TPS报告进行处理。最终，我们还纳入了其他各种监控系统的指标，例如Stackdriver的可用性探针。

TPS报告将这些数据转换为任何人都可以查询的每小时VALET指标。新创建的服务会自动注册到TPS报告中，因此可以立即查询。由于数据全部存储在BigQuery中，因此我们可以跨时间段有效地报告VALET指标。我们使用这些数据来构建各种自动报告和警报。最有趣的集成是一个聊天机器人，它使我们可以直接在商业聊天平台中报告服务的价值。例如，任何服务都可以显示最后一小时的VALET，VALET与前一周的对比，SLO之外的服务以及聊天频道中的各种其他有趣数据。

### VALET服务

我们的下一步是创建一个VALET应用程序，以存储和报告SLO数据。由于SLO可以最好地用作趋势工具，因此该服务每天，每周和每月对SLO进行跟踪。请注意，我们的SLO是一种趋势分析工具，可用于错误预算，但未直接连接到我们的监控系统。相反，我们有各种不同的监控平台，每个监控平台都有自己的警报。这些监控系统每天汇总其SLO，并发布到VALET服务以进行趋势分析。这种设置的缺点是，监控系统中设置的警报阈值未与SLO集成在一起。但是，我们可以根据需要灵活地更改监控系统。

考虑到需要将VALET与不在GCP中运行的其他应用程序集成，我们创建了一个VALET集成层，该层提供了一个API，用于每天收集服务的汇总VALET数据。TPS报告是第一个与VALET服务集成的系统，我们最终与各种本地应用程序平台(在VALET中注册的服务的一半以上)集成。

### VALET仪表板

VALET仪表板(如图3-1所示)是我们的UI，用于可视化和报告此数据，并且相对简单。它允许用户：

- 注册新服务。这通常意味着将服务分配给一个或多个URL，这些URL可能已经收集了VALET数据。
- 为五个VALET类别中的任何一个设定SLO目标。
- 在每个VALET类别下添加新的指标类型。例如，一项服务可以跟踪99%的延迟，而另一项服务可以跟踪90%(或两者)的延迟。或者，后端处理系统可以跟踪每天的交易量(一天创建的购买订单)，而客户服务前端可以跟踪每秒的高峰

交易。

VALET仪表板使用户可以立即报告许多服务的SLO，并以多种方式对数据进行切片和切块。例如，一个团队可以查看过去一周不满足SLO的所有服务的统计信息。寻求查看服务性能的团队可以查看所有服务及其所依赖服务的延迟。VALET仪表板将数据存储在简单的Cloud SQL数据库中，开发人员使用流行的商业报告工具来构建报告。

这些报告成为开发人员采取新的最佳实践的基础：定期对其服务进行SLO审核（通常是每周或每月）。基于这些审查，开发人员可以创建操作项以将服务返回到其SLO，或者可以决定需要调整不切实际的SLO。

SLOs	Total Request Per Day	Peak TPS	Overall Availability	90th Percentile(ms)	95th percentile(ms)	Sxms (%)	Overall Tickets
	2018-01-31	4,135,564	89.9	100%	496.58	884.47	0
2018-02-01	4,050,507	93.53	99.99%	491.45	892.83	0.01	0
2018-02-02	4,051,150	81.25	100%	445.86	842.1	0	0
2018-02-03	4,238,050	81.17	100%	437.92	827.31	0	0
2018-02-04	4,343,915	144.48	99.3%	857.52	1,194.88	2.4	2
2018-02-05	4,616,110	109.8	100%	474.43	864.79	0	0
2018-02-06	4,304,866	98.63	99.99%	484.78	871.01	0.01	0

## SLO扩散

一旦将SLO牢牢地扎根于组织，并且有了有效的自动化和报告功能，新的SLO就会迅速扩散。在年初跟踪了大约50种服务的SLO之后，到今年年底，我们跟踪了800种服务的SLO，每月大约有50种新服务在VALET注册。

由于VALET允许我们在THD上扩展SLO的使用，因此开发自动化所需的时间非常值得。但是，如果其他公司不能开发类似复杂的自动化技术，则不应害怕采用基于SLO的方法。尽管自动化为THD提供了额外的好处，但首先编写SLO还是有好处的。

### 将VALET应用于批处理应用

当我们围绕SLO开发可靠的报告时，我们发现VALET有其他用途。稍作调整，批处理应用程序就可以适合此框架，如下所示：

#### 容量

处理的记录量

#### 可用性

在一定时间内完成工作的频率(以百分比为单位)

### 延迟

作业运行所需的时间

### 错误

无法处理的记录

### 故障单

操作员必须手动修复数据并重新处理作业的次数

## 在测试中使用VALET

由于我们同时在开发SRE文化，因此我们发现VALET支持在演示环境中进行破坏性测试(混沌工程)自动化。有了TPS报告框架，我们可以自动运行破坏性测试并记录对服务的VALET数据的影响(或希望没有影响)。

### 未来的愿望

随着800项服务(并且还在不断增长)收集VALET数据，我们可以使用许多有用的运营数据。我们对未来有几个愿望。

现在，我们正在有效地收集SLO，我们希望使用此数据来采取行动。我们的下一步是建立类似于Google的错误预算文化，即当服务超出SLO时，团队将停止推出新功能(可靠性方面的改进除外)。为了保护我们的业务发展的速度要求，我们必须努力在SLO报告时间范围(每周或每月)与SLO违反频率之间找到良好的平衡。就像许多采用错误预算的公司一样，我们正在权衡滚动窗口与固定窗口的优缺点。

我们希望进一步完善VALET以跟踪详细的端点和服务的使用者。当前，即使特定服务具有多个端点，我们也仅在整个服务中跟踪VALET。结果，很难区分不同的操作(例如，对目录的写入与对目录的读取；虽然我们分别监控和警告这些操作，但不跟踪SLO)。同样，我们也希望针对服务的不同使用者区分VALET结果。

尽管我们目前在Web服务层跟踪延迟SLO，但我们也希望为最终用户跟踪延迟SLO。此度量将捕获第三方标记，Internet延迟和CDN缓存等因素如何影响页面开始渲染和完成渲染所需的时间。

我们还希望将VALET数据扩展到应用程序部署。具体来说，我们希望在将更改推出到下一个服务器，区域或区域之前，使用自动化功能来验证VALET是否在允许范围内。

我们已经开始收集有关服务依赖项的信息，并已制作了可视化图表的原型，该图表显示了在调用树上未达到VALET指标的位置。随着新兴的服务网格平台，这种类型的分析将变得更加容易。

最后，我们坚信，服务的SLO应该由服务的业务所有者(通常称为产品经理)根据其对业务的重要性来设置。至少，我们希望企业所有者为服务的正常运行时间设置要求，并将SLO用作产品管理和开发之间的共同目标。尽管技术人员发现VALET很直观，但是对于产品经理来说，这个概念并不是那么直观。我们正在努力使用与之相关的术语来简化VALET的概念：我们既简化了正常运行时间的选择数量，又提供了示例指标。我们还强调了从一个级别升级到另一个级别所需的大量投资。以下是我们可能提供的简化的VALET指标示例：

- 99.5%: 商店助理或新服务的MVP未使用的应用程序
- 99.9%: 适用于THD的大多数非销售系统

- 99.95%: 销售系统(或支持销售系统的服务)
- 99.99%: 共享基础设施服务

以业务术语转换指标并在产品和开发之间共享可见的目标(SLO!)，将减少在大型公司中经常看到的对可靠性的错位期望。

## 摘要

向一家大公司引入新流程，这需要一个好的策略，高管接受度，强大的宣传，易于采用的模式以及(最重要的是)耐心等待等等，更不用说新文化了。像SLO一样的重大变革可能要几年才能在公司中牢固确立。我们想强调一下，Home Depot是一家传统企业；如果我们能够成功引入如此大的变化，您也可以。您也不必一次完成所有任务。在我们逐步实施SLO的同时，制定了全面的宣传战略和明确的激励结构促进了快速的转变：我们在不到一年的时间内将SLO支持的服务从0升级到800。

## 结论

SLO和错误预算是功能强大的概念，可帮助解决许多不同的问题集。Evernote和Home Depot的这些案例研究提供了非常真实的示例，说明了实施SLO文化如何使产品开发和运维更紧密地联系在一起。这样做可以促进沟通并更好地为开发决策提供依据。最终，它将为您的客户带来更好的体验-无论这些客户是内部，外部，人工还是其他服务。

这两个案例研究突显了SLO文化是一个持续的过程，而不是一次性解决方案或解决方案。尽管它们具有相同的哲学基础，但THD和Evernote的度量方式，SLI，SLO和实现细节却截然不同。通过证明SLO实施不必特定于Google，这两个故事补充了Google对SLO的看法。正如这些公司为自己的独特环境定制SLO一样，其他公司和组织也可以。

<sup>30</sup>. 但这是另一本书的故事---在<http://bit.ly/2spqgcl>上查看更多详细信息。 ↩

<sup>31</sup>. 培训选择范围从一个小时的入门到半天的工作坊，再到与成熟的SRE团队进行为期四周的沉浸式学习，并附有毕业典礼和FiRE徽章。 ↩

<sup>32</sup>. 如在1999年的电影*Office Space*中出名。 ↩

## 第4章

### 监控

由Jess Frame , Anthony Lenton , Steven Thurgood

安东·托尔恰诺夫(Anton Tolchanov)和内伊·特金(Nejt Trdin)与卡梅拉·奎尼托(Carmela Quinito)提供

监控可以包括许多类型的数据，包括指标，文本日志记录，结构化事件日志记录，分布式跟踪和事件自省。尽管所有这些方法本身都是有用的，但本章主要介绍度量标准和结构化日志记录。根据我们的经验，这两个数据源最适合SRE的基本监控需求。

在最基本的级别上，监控使您能够了解系统，这是判断服务运行状况并在出现问题时诊断服务的核心要求。[SRE第一本书中的\[第6章\]](#)提供了一些基本的监控定义，并说明了SRE监控其系统以：

- 提醒需要注意的条件。
- 调查并诊断这些问题。
- 直观显示有关系统的信息。
- 深入了解资源使用或服务运行状况的趋势以进行长期规划。
- 比较更改前后，或实验中两组之间系统的行为。

这些用例的相对重要性可能会导致您在选择或构建监控系统时进行权衡。

本章讨论了Google如何管理监控系统，并提供了一些准则，以指导您选择和运行监控系统时可能出现的问题。

### 监控策略的理想功能

选择监控系统时，重要的是要了解和优先考虑与您相关功能。如果您要评估不同的监控系统，则本节中的属性可以帮助指导您思考哪种解决方案最适合您的需求。如果已经有了监控策略，则可以考虑使用当前解决方案的其他一些功能。取决于您的需求，单一监控系统可以解决您的所有用例，或者您可能希望使用多个系统的组合。

### 速度

在“数据的新鲜度”和“数据检索的速度”方面，不同的组织会有不同的需求。

数据应在需要时可用：刷新会影响出现问题时监控系统分页的时间。此外，缓慢的数据可能会导致您意外地对不正确的数据进行操作。例如，在事件响应期间，如果原因(采取措施)与结果(看到您的监控反映了该措施)之间的时间太长，则您可能会

认为更改没有效果，或者推断出因果之间存在错误的关联。失效时间超过四到五分钟的数据可能会严重影响您对事件做出响应的速度。

如果要基于此条件选择监控系统，则需要提前弄清速度要求。查询大量数据时，数据检索的速度通常是一个问题。如果图形必须汇总来自许多受监控系统的大量数据，则可能需要花费一些时间来加载图形。为了加快速度较慢的图形的速度，如果监控系统可以根据传入的数据创建和存储新的时间序列，将很有帮助；然后就可以预先计算常见查询的答案。

## 计算

对计算的支持可以跨越各种复杂性的各种用例。至少，您可能希望您的系统“保留多个月时间范围内的数据”。如果没有长期的数据视图，就无法分析诸如系统增长之类的长期趋势。就粒度而言，摘要数据(即您无法深入研究的汇总数据)足以促进增长计划。保留所有详细的个人指标可能有助于回答以下问题：“以前是否发生过这种异常行为？”但是，数据存储起来可能很昂贵，或者检索起来不切实际。

理想情况下，您保留的有关事件或资源消耗的指标应该是单调递增的计数器。使用计数器，您的监控系统可以计算一段时间内的窗口化功能，例如，报告该计数器每秒的请求速率。在更长的时间范围内(最多一个月)计算这些费率，可以实现基于SLO消耗的警报的构建基块(请参阅第5章)。

最后，支持更全面的统计功能可能很有用，因为琐碎的操作可能掩盖不良行为。一个在记录延迟时支持计算百分数(例如50%、95%、99%百分数)的监控系统将使您看到是否请求的50%，5%或1%太慢，而算术平均值只能告诉您请求时间较慢--没有具体说明。或者，如果您的系统不直接支持计算百分数，则可以通过以下方法实现：

- 通过将请求中花费的秒数相加并除以请求数得出平均值
- 记录每个请求并通过扫描或采样日志条目来计算百分比值

您可能希望将原始指标数据记录在一个单独的系统中以进行脱机分析-例如，用于每周或每月报告中，或者执行更复杂的计算，而这些计算在监控系统中很难计算。

## 接口

强大的监控系统应使您可以在图表中简洁地显示时间序列数据，并在表格或一系列图表样式中构造数据。仪表板将成为显示监控的主要界面，因此，选择最清楚显示所需数据的格式非常重要。一些选项包括热图，直方图和对数比例图。

您可能需要根据受众群体提供相同数据的不同视图；高级管理层可能希望查看与SRE完全不同的信息。具体说明如何创建对使用内容的人有意义的仪表板。对于每组仪表板，始终如一地显示相同类型的数据对于进行通信很有价值。

您可能需要实时绘制跨度量的不同聚合的信息，例如计算机类型，服务器版本或请求类型。对您的团队来说，最好是对数据进行临时的深入研究。通过

根据各种指标对数据进行切片，可以在需要时查找数据中的相关性和模式。

## 警报

能够对警报进行分类很有帮助: 警报的多种类别允许按比例响应。为不同的警报设置不同的严重性级别的功能也很有用: 您可以提交故障单以调查持续一个多小时的低错误率, 而100%的错误率是紧急情况, 应立即采取措施。

警报抑制功能使您避免不必要的噪音分散On-Call工程师的注意力。例如:

- 当所有节点都经历相同的高错误率时, 您可以针对全局错误率仅发出一次警报, 而不是为每个节点发送单独的警报。
- 当您的服务依赖项之一具有触发警报(例如, 后端速度缓慢)时, 您无需为服务的错误率发出警报。

您还需要确保事件结束后不再抑制警报。

您对系统所需的控制级别将决定您使用第三方监控服务还是部署并运行自己的监控系统。Google内部开发了自己的监控系统, 但是有大量的开源和商业监控系统可用。

## 监控数据来源

您选择的监控系统将由您将使用的特定监控数据源来通知。本节讨论两种常见的监控数据源: 日志和度量。还有其他一些有价值的监控资源, 例如[分布式跟踪](#)和运行时内省, 我们在这里将不介绍。

度量是代表属性和事件的数值度量, 通常是通过多个数据点以固定的时间间隔收集的。日志是事件的仅追加记录。本章的讨论重点在于结构化日志, 该结构化日志启用了丰富的查询和聚合工具, 而不是纯文本日志。

Google的基于日志的系统处理大量的高度精细的数据。在事件发生和在日志中可见之间存在固有的延迟。对于不敏感的分析, 可以使用批处理系统处理这些日志, 可以使用临时查询对其进行查询, 并可以使用仪表板对其进行可视化。此工作流程的一个示例是使用[Cloud Dataflow](#)处理日志, 并使用[BigQuery](#)临时查询和[Data Studio](#)来获取信息中心。

相比之下, 我们基于指标的监控系统从Google的每项服务中收集了大量指标, 但提供的粒度信息要少得多, 但几乎是实时的。这些特性在其他基于日志和基于指标的监控系统中是相当典型的, 尽管有例外, 例如实时日志系统或高基数指标。

我们的警报和仪表板通常使用指标。我们基于指标的监控系统的实时性意味着可以非常迅速地将问题通知工程师。我们倾向于使用日志来查找问题的根本原因, 因为我们经常无法获得所需的信息。

当报告对时间不敏感时, 我们经常使用日志处理系统生成详细的报告, 因为日志几乎总是比指标产生更准确的数据。

如果您基于指标进行警报, 则可能很想根据日志添加更多警报, 例如, 即使在发生单个异常事件时也需要通知您。在这种情况下, 我们仍然建议基于指标的警报: 您可以在发生特定事件时增加计数器指标, 并根据该指标的值配置警报。该策略将所有警报配置都放在一个位置, 使管理起来更加容易(请参阅第67页的“管理监控系统”)。

## 例子

以下真实示例说明了如何在监控系统之间进行选择的过程中进行推理。

### 将信息从日志移至指标

**问题.** HTTP状态代码是向App Engine客户调试错误的重要信号。该信息在日志中可用，但在指标中不可用。指标仪表板只能提供所有错误的全局比率，并且不包括有关确切错误代码或错误原因的任何信息。结果，调试问题的工作流程涉及：

1. 查看全局错误图以查找发生错误的时间。
2. 读取日志文件以查找包含错误的行。
3. 尝试将日志文件中的错误与图表相关联。

日志记录工具没有给出规模感，因此很难知道是否在一个日志行中经常发生错误。日志还包含许多其他不相关的行，因此很难找到根本原因。

**建议的解决方案.** App Engine开发人员团队选择将HTTP状态代码导出为指标上的标签(例如，`requests_total{status=404}`与`requests_total{status=500}`)。因为不同的HTTP状态代码的数量相对有限，所以这并未将度量标准数据的数量增加到不切实际的大小，而是使最相关的数据可用于图形化和警报。

**结果.** 这个新标签意味着团队可以升级图形以针对不同的错误类别和类型显示单独的行。客户现在可以根据暴露的错误代码快速地对可能的问题进行推测。现在，我们还可以为客户端和服务器错误设置不同的警报阈值，从而使警报触发更为准确。

### 改善日志和指标

**问题.** 一个Ads SRE团队维护了约50项服务，这些服务是用多种不同的语言和框架编写的。团队使用日志作为SLO合规性的标准真实来源。为了计算错误预算，每个服务都使用了日志处理脚本，其中包含许多特定于服务的特殊情况。这是处理单个服务的日志条目的示例脚本：

```
If the HTTP status code was in the range (500, 599)
    AND the 'SERVER ERROR' field of the log is populated
    AND DEBUG cookie was not set as part of the request
    AND the url did not contain '/reports'
    AND the 'exception' field did not contain 'com.google.ads.PasswordException' Then
        Log 'An error occurred during a request to a service' to the error log
```

这些脚本难以维护，并且还使用了基于指标的监控系统无法使用的数据。因为度量标准驱动了警报，所以有时警报不会对应于面向用户的错误。每个警报都需要一个明确的分类步骤来确定它是否面向用户，这减慢了响应时间。

**建议的解决方案.** 该团队创建了一个库，该库与每个应用程序的框架语言的逻辑息息相关。该库确定该错误是否在请求时影响了用户。工具将这个决定记录在日志中，并同时将其导出为指标，以提高一致性。如果度量标准表明服务已返回错误，则日志中将包含确切的错误以及与请求相关的数据，以帮助重现和调试问题。相应地，日志中出现的任何影响SLO的错误也会更改SLI指标，然后团队可以对其进行警报。

**结果.** 通过跨多个服务引入统一的控制界面，该团队重用了工具和警报逻辑，而不是实施多个自定义解决方案。移除复杂的，特定于服务的日志处理代码后，所有服务都因此受益，从而提高了可伸缩性。将警报直接与SLO绑定在一起后，它们就可

以更明确地采取行动，因此，误报率显着降低。

### 保留日志作为数据源

**问题.** 在调查生产问题时，一个SRE团队通常会查看受影响的实体ID，以确定用户影响和根本原因。与之前的App Engine示例一样，此调查需要的数据仅在日志中可用。该团队在响应事件时必须对此执行一次性日志查询。此步骤增加了事件恢复的时间：几分钟可以正确组合查询，还有查询日志的时间。

**建议的解决方案** 团队最初讨论了指标是否应替换其日志工具。与App Engine示例不同，实体ID可以采用数百万个不同的值，因此它不适合用作指标标签。

最终，团队决定编写一个脚本来执行他们需要的一次性日志查询，并记录下要在警报电子邮件中运行的脚本。然后，如有必要，他们可以将命令直接复制到终端中。

**结果.** 团队不再承担管理正确的一次性日志查询的认知负担。因此，他们可以更快地获得所需的结果(尽管不如基于指标的方法那样快)。他们还有一个备份计划：警报触发后，他们可以自动运行脚本，并使用小型服务器定期查询日志以不断检索半新数据。

## 管理您的监控系统

您的监控系统与您运行的任何其他服务一样重要。因此，应给予适当程度的护理和关注。**以代码形式处理您的配置**

将系统配置视为代码并将其存储在版本控制系统中是常见的做法，它们提供了一些明显的好处：更改历史记录，从特定更改到任务跟踪系统的链接，更容易的回滚和棉签检查，<sup>33</sup> 以及强制执行的代码审查程序。

我们强烈建议还将监控配置视为代码(有关配置的更多信息，请参见第14章)。与仅提供Web UI或CRUD样式API的系统相比，支持基于意图的配置的监控系统更为可取。对于许多仅读取配置文件的开源二进制文件，此配置方法是标准的。一些第三方解决方案，例如grafanalib，可为传统上使用UI配置的组件启用此方法。

## 鼓励一致性

拥有多个使用监控功能的工程团队的大型公司需要达到一个很好的平衡：集中式方法可以提供一致性，但是另一方面，各个团队可能希望完全控制其配置的设计。正确的解决方案取决于您的组织。随着时间的流逝，Google的方法已朝着集中在作为服务集中运行的单个框架上发展。由于一些原因，该解决方案对我们来说效果很好。单个框架使工程师在切换团队时能够更快地提高工作效率，并使调试过程中的协作更加轻松。我们还提供集中式仪表板服务，每个团队的仪表板都是可发现和可访问的。如果您容易理解另一个团队的仪表板，则可以更快地调试问题。

如果可能的话，使基本的监控范围不费吹灰之力。如果您所有的服务<sup>34</sup> 都导出了一致的基本指标集，则可以在整个组织中自动收集这些指标并提供一致的仪表板集。这种方法意味着您自动启动的任何新组件都具有基本监控。公司中的许多团队(甚至是非工程团队)也可以使用此监控数据。

### 首选松散耦合

业务需求发生变化，并且您的生产系统从现在开始一年后将有所不同。同样，您的监控系统也需要随着时间的推移而发展，因为它所监控的服务会通过不同的故障模式而发展。

我们建议使监控系统的组件保持松散耦合。您应该具有用于配置每个组件和传递监控数据的稳定接口。单独的组件应负责收集，存储，警告和可视化监控。稳定的接口可以更轻松地交换任何给定的组件，以获得更好的替代方案。

将功能拆分为单个组件在开源世界中正变得越来越流行。十年前，诸如[Zabbix](#)之类的监控系统将所有功能组合为一个组件。现代设计通常涉及到将收集和规则评估分开(使用诸如[Prometheus服务器](#)之类的解决方案)，长期时间序列存储([InfluxDB](#))，警报汇总([Alertmanager](#))和仪表板([Grafana](#))。

在撰写本文时，至少有两种流行的开放标准可用于对软件进行检测和公开指标:

#### *statsd*

度量标准聚合守护程序最初由Etsy编写，现在已移植到大多数编程语言中。

#### *Prometheus*

开源监控解决方案，具有灵活的数据模型，支持度量标准标签和强大的直方图功能。其他系统现在正在采用Prometheus格式，并且已标准化为[OpenMetrics](#)。

可以使用多个数据源的单独的仪表板系统提供了服务的集中统一概述。Google最近在实践中看到了这一好处: 我们的旧版监控系统(Borgmon<sup>35</sup>)在与警报规则相同的配置下组合了仪表板。在迁移到新系统([Monarch](#))时，我们决定将仪表板移至单独的服务([Viceroy](#))。由于Viceroy不是Borgmon或Monarch的组成部分，因此Monarch的功能需求较少。由于用户可以使用Viceroy基于来自两个监控系统的数据显示图形，因此他们可以逐渐从Borgmon迁移到Monarch。

## 目的指标

第5章介绍了如何在系统的错误预算受到威胁时使用SLI指标进行监控和警报。SLI指标是基于SLO的警报触发时要检查的第一个指标。这些指标应显着显示在服务的仪表板中，最好在其目标页面上。

在调查违反SLO的原因时，您很可能不会从SLO仪表板获得足够的信息。这些仪表板表明您违反了SLO，但不一定要这样做。监控仪表板还应显示哪些其他数据？

我们发现以下准则有助于实施指标。这些指标应提供合理的监控，使您能够调查生产问题，并提供有关服务的广泛信息。

## 预期的变化

诊断基于SLO的警报时，您需要能够从通知您有关用户影响问题的警报指标转变为告诉您是什么引起这些问题的指标。最近打算对服务进行的更改可能有误。添加监控以通知您生产中的任何变化。<sup>36</sup>要确定触发器，我们建议以下操作:

- 监控二进制文件的版本。
- 监控命令行标志，尤其是在使用这些标志启用和禁用服务功能时。

- 如果将配置数据动态推送到您的服务，请监控此动态配置的版本。

如果未对系统的任何这些版本进行版本控制，则您应该能够监控上次构建或打包该系统的时间戳。

当您尝试将中断与部署相关联时，查看警报中链接的图形/仪表板比事后浏览CI/CD(持续集成/持续交付)系统日志要容易得多。

## 依赖

即使您的服务没有更改，其任何依赖关系也可能会更改或出现问题，因此您还应该监控来自直接依赖关系的响应。

导出每个依赖项的请求和响应大小(以字节，等待时间和响应代码为单位)是合理的。选择要绘制图形的度量时，请记住四个[黄金信号](#)。您可以在指标上使用其他标签，以按响应代码，RPC(远程过程调用)方法名称和对等作业名称对它们进行分类。

理想情况下，您可以检测较低级别的RPC客户端库以一次导出这些指标，而不是要求每个RPC客户端库将其导出。[37](#) 检测客户端库可提供更高的一致性，并允许您免费监控新的依赖项。

有时您会遇到提供非常狭窄的API的依赖项，其中所有功能都可以通过名为Get，Query或同等无益的单个RPC来使用，并且实际命令被指定为该RPC的参数。客户端库中的单个检测点缺乏这种类型的依赖关系：您将观察到时延的巨大差异以及一定百分比的错误，这些错误可能表示也可能不表示此不透明API的某些部分完全失败。如果此依赖关系很关键，则有两种方法可以很好地监控它：

- 导出单独的度量标准以适应依赖性，以便度量标准可以解包收到的请求以获取实际信号。
- 要求该依赖所有者执行重写以导出更广泛的API，该API支持在单独的RPC服务和方法之间划分的单独功能。

## 饱和

旨在监控和跟踪服务所依赖的每种资源的使用情况。某些资源具有您不能超过的硬限制，例如分配给应用程序的RAM，磁盘或CPU配额。其他资源-例如打开文件描述符，任何线程池中的活动线程，队列中的等待时间或书面日志的数量-可能没有明确的硬限制，但仍需要管理。

根据所使用的编程语言，您应该监控其他资源：

- 在Java中：堆和[元空间](#)的大小，以及更具体的指标，具体取决于您使用的垃圾收集类型
- 在Golang中：goroutine的数量

语言本身为跟踪这些资源提供了各种支持。

除了按第5章中所述对重大事件进行警报外，您还可能需要设置在耗尽特定资源时将触发的警报，例如：

- 当资源有硬限制时
- 超过使用量阈值会导致性能下降

您应该具有监控指标来跟踪所有资源，甚至是服务管理良好的资源。这些指标对于容量和资源规划至关重要。

## 服务流量状态

最好添加指标或指标标签，以允许仪表板按状态代码细分服务流量(除非您的服务用于SLI的指标已包含此信息)。以下是一些建议:

- 对于HTTP流量，请监控所有响应代码，即使它们没有提供足够的警报信号，因为某些响应代码可能是由错误的客户端行为触发的。
- 如果您对用户应用速率限制或配额限制，请监控由于配额不足而拒绝的请求总数。

这些数据的图形可以帮助您确定在生产变更期间错误量何时发生明显变化。

## 实施有目的的指标

每个暴露的指标都应达到目的。抵制仅因易于生成而导出少量指标的诱惑。相反，请考虑如何使用这些指标。指标设计或缺乏指标设计太含蓄。

理想情况下，用于警报的度量标准值仅在系统进入问题状态时才发生重大变化，而在系统正常运行时则不会发生变化。另一方面，调试指标不具有这些要求-它们旨在提供有关警报触发时正在发生的情况的见解。良好的调试指标将指向可能导致问题的系统某些方面。撰写事后评估时，请考虑可以使用哪些其他指标更快地诊断问题。

## 测试警报逻辑

在理想的情况下，监控和警报代码应遵循与代码开发相同的测试标准。虽然Prometheus开发人员正在[讨论开发用于监控的单元测试](#)，但目前尚没有广泛采用的系统可以执行此操作。

在Google，我们使用特定于域的语言来测试我们的监控和警报，该语言允许我们创建综合时间序列。然后，我们根据派生的时间序列中的值或特定警报的触发状态和标签存在情况来编写断言。

监控和警报通常是一个多阶段的过程，因此需要多个系列的单元测试。尽管该领域仍很不发达，但是如果您想在某个时候实施监控测试，我们建议采用三层方法，如图4-1所示。

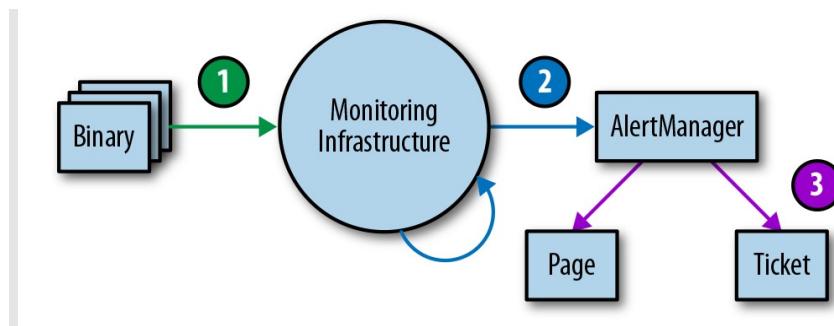


图4-1. 监控测试环境层

1. **二进制报告:** 检查导出的度量变量在某些条件下是否按预期变化。

2. **监控配置:** 确保规则评估产生预期结果，并且特定条件产生预期警报。
3. **允许的配置:** 根据警报标签值测试是否将生成的警报路由到预定的目的地。

如果您无法通过综合手段来测试监控，或者某个阶段您根本无法测试，请考虑创建一个正在运行的系统，该系统可以导出众所周知的指标，例如请求数和错误数。您可以使用此系统来验证派生的时间序列和警报。您的警报规则很可能在您配置警报规则后的几个月或几年内不会触发，并且您需要确信，当指标超过特定阈值时，将会向有意义的工程师发出有意义的通知。

## 结论

由于SRE角色负责生产中系统的可靠性，因此经常需要SRE熟悉服务的监控系统及其功能。没有这些知识，SRE可能不知道在何处看待，如何识别异常行为或如何在紧急情况找到他们需要的信息。

我们希望通过指出我们发现有用的监控系统功能以及原因，可以帮助您评估监控策略满足您的需求的程度，探索您可能能够利用的其他功能以及考虑可能要进行的更改。您可能会发现将一些度量标准来源和日志记录结合到监控策略中很有用；您需要的确切组合在很大程度上取决于上下文。确保收集用于特定目的的指标。该目的可能是为了更好地进行容量规划，协助调试或直接将问题通知您。

监控到位后，它必须可见且有用。为此，我们还建议您测试监控设置。一个好的监控系统可以带来好处。值得进行大量投资，以深入思考哪种解决方案最能满足您的需求，并进行迭代直到您找到正确的解决方案。

<sup>33</sup>. 例如，使用promtool来验证您的Prometheus配置在语法上是正确的。 ↵

<sup>34</sup>. 您可以通过公共库导出基本指标：诸如[OpenCensus](#)之类的检测框架，或[Istio](#)之类的服务网格。 ↵

<sup>35</sup>. 有关Borgmon的概念和结构，请参见[Site Reliability Engineering](#)的Chapter 10。 ↵

<sup>36</sup>. 在这种情况下，通过日志进行监控很有吸引力，特别是因为生产更改相对很少。无论您使用日志还是指标，这些更改都应在仪表板上显示，以便调试生产问题时可以轻松访问。 ↵

<sup>37</sup>. 请参阅<https://opencensus.io/>，以获取提供此功能的一组库。 ↵

## 第5章

### 基于SLO报警

由史蒂文·瑟古德

与Jess Frame , Anthony Lenton , Carmela Quinito , Anton Tolchanov和Nejc Trdin撰写

本章说明如何在重大事件上将SLO变成可操作的警报。我们的[第一本SRE书](#)和这本书都谈到了实现SLO。我们相信，拥有良好的SLO(可以衡量您的平台的可靠性)，如客户所体验到的那样，可以为呼叫工程师何时响应提供最高质量的指示。在这里，我们提供了有关如何将这些SLO转换为警报规则的具体指导，以便您可以在消耗过多错误预算之前对问题进行响应。

我们的示例介绍了一系列用于度量指标和逻辑警报的日益复杂的实现。我们讨论了它们的效用和缺点。尽管我们的示例使用了简单的请求驱动服务和[Prometheus语法](#)，但是您可以在任何警报框架中应用此方法。

## 注意事项

为了从服务水平指示器(SLI)和错误预算生成警报，您需要一种将这两个元素组合为特定规则的方法。您的目标是收到[重要事件](#)的通知: 重大事件消耗了错误预算的大部分。

评估警报策略时，请考虑以下属性:

**精确**

检测到的重大事件的比例。如果每个警报都对应一个重大事件，则精度为100%。请注意，在低流量时段，警报可能对不重要的事件特别敏感(在第86页的“低流量服务和错误预算警报”中进行了讨论)。

**召回**

检测到的重大事件的比例。如果每个重大事件都导致警报，则召回率为100%。

**检测时间**

在各种情况下发送通知需要多长时间。较长的检测时间会对错误预算产生负面影响。

**重置时间**

解决问题后，警报触发多长时间。较长的重置时间可能导致混乱或问题被忽略。

## 重要事件提醒方式

为您的SLO构造警报规则可能会变得非常复杂。在这里，我们介绍了六种方法，以提高保真度的顺序配置对重大事件的警报，以提供一个选项，可以同时对精度，调用，检测时间和重置时间这四个参数进行良好控制。以下每种方法都可以解决一个不同的问题，并且某些方法最终可以同时解决多个问题。前三种不可行的尝试朝着后三种可行的警报策略迈进，方法6是最可行和最推荐的选择。第一种方法易于实施，但不足，而最佳方法则可以为长期和短期内的SLO防御提供完整的解决方案。

在此讨论中，“错误预算”和“错误率”适用于所有SLI，而不仅仅是名称上带有“错误”的SLI。在“要测量的内容”部分中：第20页上的“使用SLI”，我们建议使用SLI来捕获好事件与总事件的比率。错误预算给出了允许的不良事件的数量，错误率是不良事件与总事件的比率。

## 1: 目标错误率 $\geq$ SLO阈值

对于最简单的解决方案，您可以选择一个较小的时间窗口(例如10分钟)，并在该时间窗口的错误率超过SLO时发出警报。

例如，如果30天的SLO为99.9%，则在过去10分钟内的错误率 $\geq 0.1\%$ 时发出警报：

```
- alert: HighErrorRate
expr: job:slo_errors_per_request:ratio_rate10m{job="myjob"} >= 0.001
```

在Prometheus中使用记录规则计算得出的这10分钟平均值：

```
record: job:slo_errors_per_request:ratio_rate10m expr:
sum(rate(slo_errors[10m])) by (job) /
sum(rate(slo_requests[10m])) by (job)
```

如果您不从工作中导出slo\_errors和slo\_requests，则可以通过重命名指标来创建时间序列：

```
record: slo_errors
expr: http_errors
```

当最近的错误率等于SLO时发出警报意味着系统检测到以下预算支出：

```
(alerting window size) / (reporting period)
```

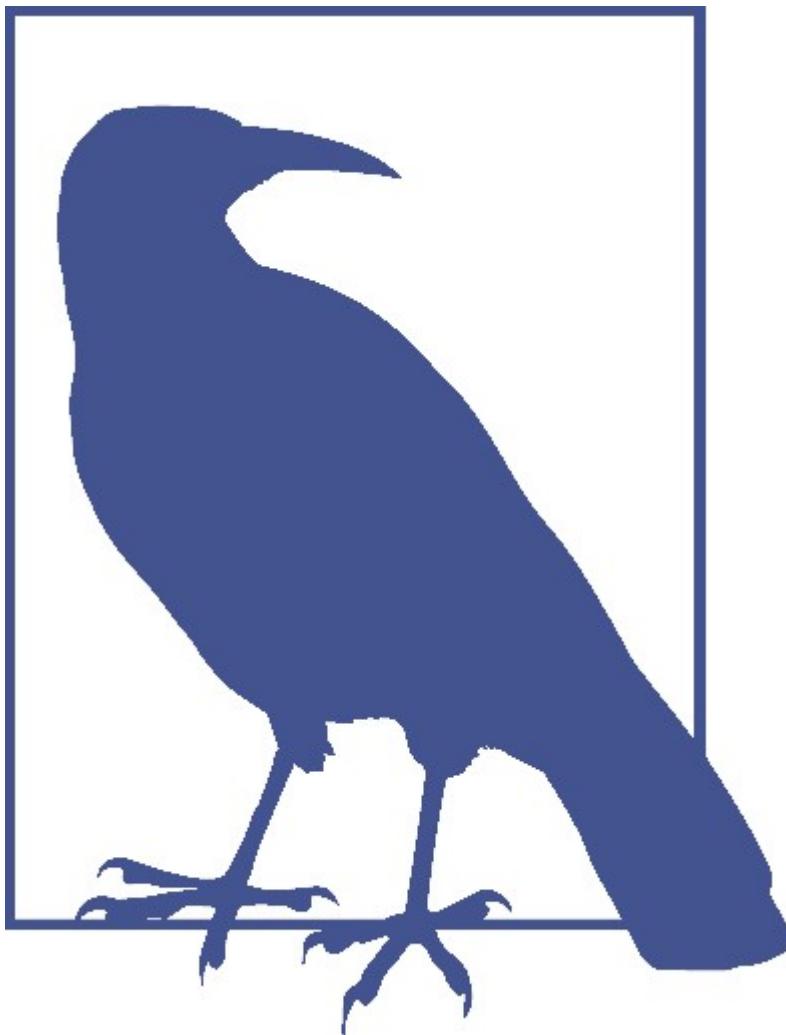


图5-1显示了具有10分钟警报窗口和99.9%SLO的示例服务的检测时间与错误率之间的关系。

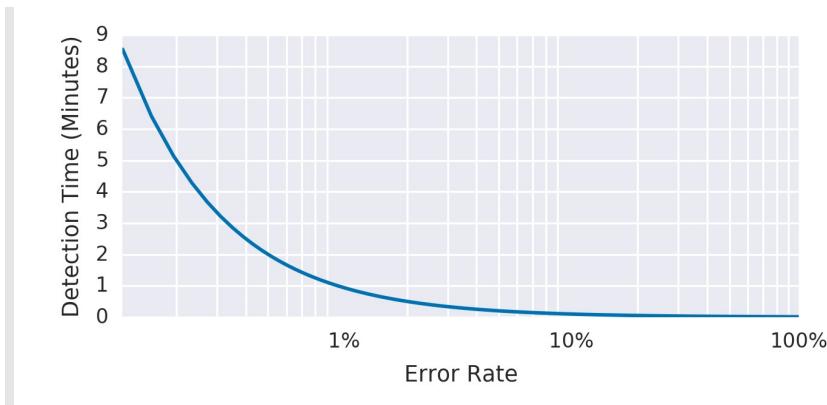


图5-1. 具有10分钟警报窗口和99.9 %SLO的示例服务的检测时间

表5-1显示了即时错误率过高时发出警报的优点和缺点。

表5-1. 当即时错误率过高时发出警报的利弊

优点	缺点
检测时间好: 完全中断0.6秒。	精度低: 该警报在许多不会威胁到SLO的事件上触发。10分钟内0.1%的错误率会发出警报，而仅消耗每月错误预算的0.02%.
此警报会在威胁到SLO的任何事件时触发，显示出良好的召回效果。	举个例子，您每天可能最多收到144条警报，不对任何警报采取行动，并且仍然符合SLO。

## 2: 警报窗口增加

我们可以通过更改警报窗口的大小来提高精度，以此为基础。通过增加窗口大小，您将在触发警报之前花费更高的预算金额。

为了保持[可管理的警报率](#)，您决定仅在事件消耗了30天错误预算的5%(36小时的窗口)时才收到通知:

```
- alert: HighErrorRate
  expr: job:slo_errors_per_request:ratio_rate36h{job="myjob"} > 0.001
```

现在，检测时间为:

```
(1 - SLO) / (error ratio) × alerting window size
```

表5-2显示了在较大的时间范围内错误率过高时发出警报的优点和缺点。

表5-2. 在较大的时间范围内错误率过高时发出警报的利弊

优点	缺点
检测时间还是不错的: 2分钟10秒完全中断。	重置时间很差: 在100%中断的情况下，警报会在2分钟后立即触发，并在接下来的36小时内继续触发。
比前面的示例更好的精度: 通过确保错误率持续更长的时间，警报很可能对错误预算构成严重威胁。	由于存在大量数据点，因此在更长的窗口中计算速率在内存或I/O操作方面可能会非常昂贵。

图5-2显示，尽管36小时内的错误率已降至可以忽略的水平，但36小时平均错误率仍高于阈值。

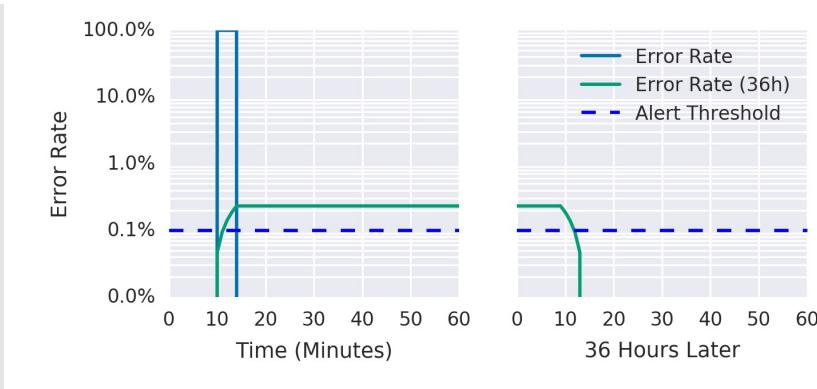


图5-2.36小时内的错误率

### 3: 递增警报持续时间

大多数监控系统都允许您将持续时间参数添加到警报条件，因此除非该值在一段时间内保持高于阈值，否则不会触发警报。您可能会想将此参数用作添加较长窗口的相对便宜的方法：

```
- alert: HighErrorRate
  expr: job:slo_errors_per_request:ratio_rate1m{job="myjob"} > 0.001 for: 1h
```

表5-3显示了使用持续时间参数进行警报的优缺点。

表5-3. 使用持续时间参数发出警报的利弊

优点	缺点
警报可以更高精度。	召回不良和检测时间差：由于持续时间不随事件的严重性而定，因此一小时后会发出100%的停机警报，而检测时间与0.2%的停机时间相同。100%的停机将消耗该小时30天预算的140%。
在触发之前要求持续的错误率意味着警报更有可能对应于重大事件。	如果度量标准甚至立即恢复到SLO内的水平，则持续时间计时器将重置。在丢失的SLO和通过的SLO之间波动的SLI可能永远不会发出警报。

由于表5-3中列出的原因，我们不建议将持续时间用作基于SLO的警报条件的一部分。<sup>38</sup>

图5-3显示了具有以下条件的服务在5分钟内的平均错误率：

警报触发前10分钟。每10分钟持续5分钟的一系列100%错误峰值，即使消耗了35%的错误预算，也永远不会触发警报。

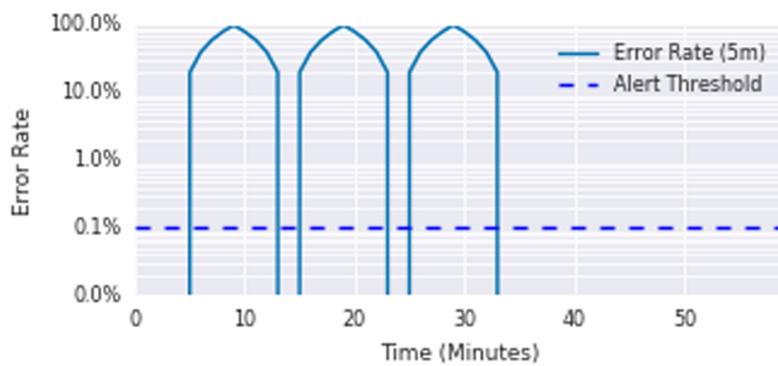


图5-3. 每10分钟出现100%错误峰值的服务

每个高峰都消耗了30天预算的近12%，但警报从未触发。

## 4: 消耗速率警报

为了改进先前的解决方案，您想要创建具有良好检测时间和高精度的警报。为此，您可以引入消耗率以减小窗口大小，同时保持警报预算支出不变。

消耗速率是相对于SLO，服务消耗错误预算的速度。图5-4显示了消耗率和错误预算之间的关系。

该示例服务使用1的消耗率，这意味着它消耗错误预算的速度使得在SLO时间窗口结束时，您的预算恰好为零(请参阅[第4章](#))。在30天的时间范围内，如果SLO为99.9%，则恒定的0.1%错误率将使用所有错误预算：消耗率为1。

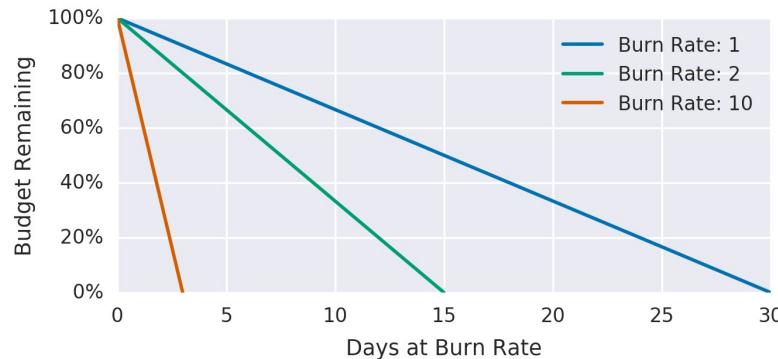


图5-4. 相对于消耗率的错误预算

表5-4显示了消耗率，其相应的错误率以及用尽SLO预算所需的时间。

表5-4. 完成预算用尽的消耗率和时间

消耗率	99.9% SLO的错误率	消耗时间
1	0.1%	30天
2	0.2%	15天
10	1%	3天
1,000	100%	43分钟

通过将警报窗口固定为一小时，并确定5%的错误预算支出足以通知某人，您可以得出要用于警报的消耗率。

对于基于消耗率的警报，触发警报所需的时间为：

```
(1 - SLO) / (error ratio) × alerting window size × burn rate
```

警报触发时消耗的错误预算为：

```
burn rate × alerting window size / period
```

一小时内花费30天的错误预算的5%，需要的消耗率为36。现在，警报规则变为：

```
- alert: HighErrorRate
  expr: job:slo_errors_per_request:ratio_rate1h{job="myjob"} > 36 * 0.001
```

表5-5. 基于消耗率的警报的利弊

优点	缺点
精度好：该策略选择了错误预算支出的重要部分来提醒。	召回率低：35倍的消耗速度永远不会发出警报，但会在20.5小时内消耗掉30天的所有错误预算。
时间窗口更短，计算起来更便宜。	重置时间：58分钟仍然太长。
良好的检测时间。	
更好的重置时间：58分钟。	

## 5: 多个消耗速率警报

您的警报逻辑可以使用多个消耗率和时间窗口，并在消耗率超过指定阈值时发出警报。此选项保留了警告消耗率的好处，并确保您不会忽略较低(但仍然很重要)的错误率。

为通常不会引起注意的事件设置故障单通知也是一个好主意，但是如果不去检查会耗尽您的错误预算-例如，三天内预算消耗为10%。该错误率捕获了重大事件，但是由于预算消耗率为解决该事件提供了足够的时间，因此您无需寻呼某人。

我们建议一小时内2%的预算消耗和六小时内5%的预算消耗，作为合理的寻呼起始编号，三天之内将10%的预算消耗作为故障通知的良好基准。适当的数字取决于服务和基准页面负载。为了获得更繁忙的服务，并且根据周末和节假日的On-Call职责，您可能希望在六小时窗口内收到故障单警报。

表5-6显示了消耗的SLO预算百分比的相应消耗率和时间窗口。

表5-6. 建议的时间窗口和消耗率，用于消耗的SLO预算的百分比

SLO预算消耗	时间窗口	消耗率	通知
2%	1小时	14.4	呼叫
5%	6小时	6	呼叫
10%	3天	1	工单

警报配置可能类似于：

```
expr: (
    job:slo_errors_per_request:ratio_rate1h{job="myjob"} > (14.4*0.001)
    or
    job:slo_errors_per_request:ratio_rate6h{job="myjob"} > (6*0.001)
)
severity: page
expr: job:slo_errors_per_request:ratio_rate3d{job="myjob"} > 0.001 severity: t:
```

图5-5根据错误率显示了检测时间和警报类型。



图5-5. 错误率，检测时间和警报通知

多种消耗速率使您可以根据警报的响应速度来调整警报，使其具有适当的优先级。如果问题会在几个小时或几天内耗尽错误预算，则发送活动通知是适当的。否则，基于故障单的通知在下一个工作日处理警报更为合适。<sup>39</sup>表5-7列出了使用多种消耗率的优缺点。

表5-7. 使用多种消耗速率的优缺点

优点	缺点
能够根据紧急情况使监控配置适应多种情况: 如果错误率很高, 则迅速发出警报; 如果错误率较低但持续存在, 则最终发出警报。	更多数字, 窗口大小以及要管理和推理的阈值。
与所有固定预算部分警报方法一样, 精度很高。	为期三天的窗口会导致更长的重置时间。为了避免在所有条件都满足的情况下触发多个警报, 您需要实施警报抑制。例如: 五分钟内支出10%的预算也意味着六小时内支出了5%的预算, 一小时内支出了2%的预算。除非监控系统足够智能以阻止其运行, 否则此方案将触发三个通知。
好的召回, 因为该窗口为期三天。	
根据某人为捍卫SLO做出反应的速度来选择最合适的警报类型。	

## 6: 多窗口，多消耗率警报

我们可以在第5次迭代中增强多消耗率警报, 以便仅在我们仍“还在”积极消耗预算时通知我们-从而减少误报的数量。为此, 我们需要添加另一个参数: 一个较短的窗口, 以检查触发警报时是否仍在使用错误预算。

一个好的指导原则是使短窗口的时间为长窗口的持续时间的1/12, 如图5-6所示。该图显示了两个警报阈值。在经历15%的错误10分钟后, 短窗口平均值立即超过警报阈值, 而长窗口平均值在5分钟后超过阈值, 此时警报开始触发。错误停止5分钟后, 短窗口平均值下降到阈值以下, 此时警报停止触发。错误停止60分钟后, 长窗口平均值下降到阈值以下。

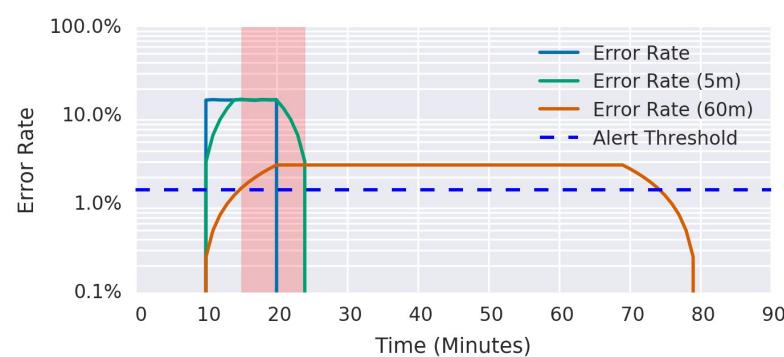


图5-6. 短窗口和长警报窗口

例如，当您在前一小时和前五分钟内的消耗速度超过14.4倍时，可以发送页面级警报。仅当您用完预算的2%时才会触发此警报，但是通过在五分钟后而不是一小时后停止触发，可以显示更好的重置时间：

```

expr: (
  job:slo_errors_per_request:ratio_rate1h{job="myjob"} > (14.4*0.001)
  and
  job:slo_errors_per_request:ratio_rate5m{job="myjob"} > (14.4*0.001)
) or
(
  job:slo_errors_per_request:ratio_rate6h{job="myjob"} > (6*0.001)
  and
  job:slo_errors_per_request:ratio_rate30m{job="myjob"} > (6*0.001)
)
severity: page
expr: (
  job:slo_errors_per_request:ratio_rate24h{job="myjob"} > (3*0.001)
  and
  job:slo_errors_per_request:ratio_rate2h{job="myjob"} > (3*0.001)
) or
(
  job:slo_errors_per_request:ratio_rate3d{job="myjob"} > 0.001
  and
  job:slo_errors_per_request:ratio_rate6h{job="myjob"} > 0.001
)
severity: ticket

```

我们建议将表5-8中列出的参数作为基于SLO的警报配置的起点。

表5-8.99.9 %SLO警报配置的推荐参数

严重程度	长期窗口	短期窗口	消耗率	消耗的错误预算
呼叫	1小时	5分钟	14.4	2%
呼叫	6小时	30分钟	6	5%
故障单	3天	6小时	1	10%

我们发现基于多种消耗速率的警报是实施基于SLO的警报的有效方法。

表5-9显示了使用多个消耗速率和窗口大小的好处和局限性。

表5-9. 使用多种消耗速率和窗口大小的优缺点

优点	缺点
灵活的警报框架，可让您根据事件的严重性和组织的要求来控制警报的类型。	需要指定许多参数，这会使警报规则难以管理。有关管理警报规则的更多信息，请参阅第89页“大规模警报”。
与所有固定预算部分警报方法一样，精度很高。	
好召回，时间窗口为三天。	

## 低流量服务和错误预算警报

当出现问题时，足够高的传入请求速率提供有意义的信号时，多窗口，多消耗速率的方法就可以很好地发挥作用。但是，这些方法会对接收请求率较低的系统造成问题。如果系统的用户数量较少或自然人流量少(例如晚上和周末)，则可能需要更改方法。

在低流量服务中自动区分不重要的事件比较困难。

例如，如果系统每小时接收10个请求，则单个失败请求的每小时错误率将为10%。对于99.9%的SLO，此请求构成了1,000倍的消耗率，并且由于消耗了30天错误预算的13.9%，因此会立即进行分页。此方案在30天内仅允许七个失败的请求。单个请求可能会因大量短暂且无趣的原因而失败，而这些原因不一定能以与大型系统停机相同的方式来解决。

最佳解决方案取决于服务的性质: 单个失败请求的影响<sup>40</sup>是什么？如果失败的请求是一次性的，不重试的高价值请求，则高可用性目标可能是合适的。从业务角度来看，调查每个失败的请求可能很有意义。但是，在这种情况下，警报系统将错误通知您为时已晚。

我们建议一些关键选项来处理低流量服务:

- 产生人为流量，以补偿来自实际用户的信号不足。
- 将较小的服务合并为较大的服务以进行监控。
- 修改产品，以便:
  - 需要更多的请求才能将单个事件视为失败。
  - 单个故障的影响较小。

## 产生人工流量

系统可以综合用户活动以检查潜在的错误和高延迟请求。在没有实际用户的情况下，您的监控系统可以检测到综合错误和请求，因此您的On-Call工程师可以在问题影响太多实际用户之前做出响应。

人工流量提供了更多可用的信号，并允许您重用现有的监控逻辑和SLO值。您甚至可能已经拥有大多数必要的流量生成组件，例如黑盒探针和集成测试。

产生人工负载确实有一些缺点。大多数需要SRE支持的服务都是复杂的，并且具有较大的系统控制界面。理想情况下，应设计和构建用于使用人工流量进行监控的系统。即使对于非平凡的服务，您也只能合成用户请求类型总数的一小部分。对于有状态服务，更多的状态会加剧这种问题。

此外，如果问题影响实际用户但不影响人工流量，则成功的人工请求会隐藏真实用户信号，因此不会通知您用户看到错误。

## 合并服务

如果多个低流量服务有助于一项整体功能，则将其请求组合到一个较高级别的组中可以更准确地检测到重大事件，并且误报更少。为了使这种方法起作用，服务必须以某种方式关联-您可以组合构成同一产品一部分的微服务，或组合由同一二进制文件处理的多种请求类型。

组合服务的不利之处在于，单个服务的完全失败可能不算是重大事件。通过选择具有共享故障域的服务(例如公共后端数据库)，可以增加故障对整个组产生影响的可能性。您仍然可以使用较长时间的警报，这些警报最终会针对单个服务捕获这100%的故障。

## 进行服务和基础架构更改

对重大事件发出警报旨在提供足够的通知，以在问题用尽整个错误预算之前缓解问题。如果您不能将监控调整为对短暂事件不太敏感，并且无法生成综合流量，则可以考虑更改服务以减少单个失败请求对用户的影响。例如，您可能会：

### 低流量服务和错误预算警报

- 将客户端修改为重试，具有指数补偿和抖动。<sup>41</sup>
- 设置可捕获最终执行请求的后备路径，该路径可以在服务器或客户端上进行。

这些更改对于高流量的系统很有用，但对于低流量的系统则更是如此：它们允许错误预算中出现更多的失败事件，来自监控的更多信号以及在事件变得严重之前有更多的时间响应事件。

### 降低SLO或增加窗口

您可能还需要重新考虑单个故障对错误预算的影响是否准确反映了其对用户的影响。如果少量错误导致您损失错误预算，您是否真的需要派遣工程师立即解决问题？如果不是这样，那么用户将对较低的SLO感到同样满意。如果SLO较低，则仅将较大的持续停机通知工程师。

一旦与服务的利益相关者协商降低SLO(例如，将SLO从99.9%降低到99%)，实施更改就非常简单：如果您已经有基于报告的系统来进行报告，监控和警报SLO阈值，只需将新的SLO值添加到相关系统即可。

降低SLO确实有一个缺点：这涉及产品决策。更改SLO会影响系统的其他方面，例如对系统行为的期望以及何时制定错误预算策略。这些其他要求对产品而言可能比避免一些低信号警报更为重要。

以类似的方式，增加用于警报逻辑的时间窗口可确保警报触发页面更重要且值得关注。

在实践中，我们使用以下方法的组合来提醒低流量服务：

- 可能时会产生虚假流量，并且可以实现良好的覆盖率
- 修改客户端，使短暂的故障不太可能造成用户伤害
- 汇总共享某些故障模式的较小服务
- 设置与失败请求的实际影响相称的SLO阈值

## 极端可用性目标

具有极低或极高可用性目标的服务可能需要特别考虑。例如，考虑具有90%可用性目标的服务。表5-8对在一个小时内消耗了2%的错误预算时说了说。因为100%的停机仅消耗了该小时预算的1.4%，所以该警报永远不会触发。如果您的错误预算长时间设置的，则可能需要调整警报参数。

对于具有极高可用性目标的服务，100%中断所需的时间非常短。目标每月可用性为99.999%的服务100%中断将在26秒内用尽预算，这比许多监控服务的度量标准收集间隔要小，更不用说端到端时间来生成警报并通过诸如电子邮件和SMS的通知系统传递。即使警报直接进入自动解决系统，问题也可能会完全消耗错误预算，然后才能缓解它。

收到仅剩26秒预算的通知并不一定是一个坏策略。这对于捍卫SLO毫无用处。捍卫这种可靠性水平的唯一方法是设计系统，以使100%中断的可能性极低。这样，您可以在消耗预算之前解决问题。例如，如果最初将更改仅推广到仅1%的用户，并且以1%的比率消耗错误预算，则现在您有43分钟的时间来耗尽错误预算。有关设计这样的系统的策略，请参见第16章。

## 大范围报警

扩展服务时，请确保您的警报策略同样可扩展。您可能很想为单个服务指定自定义警报参数。如果您的服务包含100个微服务(或等效地，一个具有100种不同请求类型的服务)，则这种情况会很快累积无法扩展的工作量和认知负担。

在这种情况下，我们强烈建议您不要为每个服务分别指定警报窗口和消耗速率参数，因为这样做很快就会变得不堪重负。[42](#) 确定警报参数后，请将其应用于所有服务。

一种用于管理大量SLO的技术是将请求类型分组为具有大致相似的可用性要求的存储桶。例如，对于具有可用性和延迟SLO的服务，可以将其请求类型分组到以下存储桶中：

### CRITICAL

对于最重要的请求类型，例如用户登录服务时的请求。

### HIGH\_FAST

对于具有高可用性和低延迟要求的请求。这些请求涉及核心交互功能，例如当用户单击按钮以查看其广告资源本月赚了多少钱时。

### HIGH\_SLOW

对于重要但对延迟不太敏感的功能，例如当用户单击按钮以生成过去几年中所有广告活动的报告时，并且不希望数据立即返回。

### LOW

对于必须具有一定可用性的请求，但对于用户而言，中断几乎是不可见的-例如，轮询处理程序以获取可能长时间失败而对用户没有影响的帐户通知。

### NO\_SLO

对于用户完全不可见的功能-例如，暗启动或明确位于任何SLO之外的alpha功能。

通过对请求进行分组而不是对所有请求类型都设置唯一的可用性和延迟目标，可以将请求分为五个存储桶，如表5-10中的示例。

表5-10.根据相似的可用性要求和阈值请求类别存储桶

请求类别	可用性	延迟 @ 90% <sup>a</sup>	延迟 @ 99%
CRITICAL	99.99%	100毫秒	200毫秒
HIGH_FAST	99.9%	100毫秒	200毫秒
HIGH_SLOW	99.9%	1,000毫秒	5,000毫秒
LOW	99%	无	无
NO_SLO	无	无	无

<sup>a</sup>a 90%的请求[比此阈值更快](#)。

这些存储桶提供了足够的保真度来保护用户的满意度，但是与“更可能”更精确地映射到用户体验且管理起来更为复杂和昂贵的系统相比，琐事要少多了。

## 结论

如果您设置的SLO有意义，可以理解并以指标表示，则可以将警报配置为仅在错误预算有可操作的特定威胁时通知On-Call的人。

在重大事件上发出警报的技术范围包括从错误率超过SLO阈值时发出警报到使用多个级别的消耗率和窗口大小。在大多数情况下，我们认为多窗口，多消耗率警报技术是捍卫应用程序SLO的最合适方法。希望我们为您提供了为您自己的应用程序和组织做出正确的配置决策所需的上下文和工具。

<sup>38</sup>. 当您在很短的持续时间内过滤掉短暂的噪声时，Duration子句有时会很有用。但是，您仍然需要注意本节中列出的缺点。[←](#)

<sup>39</sup>. 如“站点可靠性工程”的[简介](#)中所述，页面和故障单是促使人们采取行动的唯一有效方法。[←](#)

<sup>40</sup>. “测量内容：第20页上的“使用SLI”建议一种SLI样式，该样式将根据对用户的影响进行缩放。[←](#)

- 41. 请参阅《站点可靠性工程》中的"过载和故障"。 ↵
- 42. 除临时更改警报参数外，这是解决持续中断的必要条件，并且在此期间不需要接收通知。 ↵

## 第6章

### 消除琐事

由David Challoner , Joanna Wijntjes , David Huska

马修·萨特威尔，克里斯·科肯德尔，克里斯·谢里尔

约翰·鲁尼和维维克·劳

与Betsy Beyer , Max Luebbe , Alex Perry 和 Murali Suriar 撰写

Google SRE花费大量时间进行优化-通过项目工作和开发人员协作，从系统中获取每一点性能。但是优化的范围不仅限于计算资源: SRE优化他们如何度过时间也很重要。首先，我们要避免执行归类为琐事的任务。有关琐事的全面讨论，请参阅《站点可靠性工程》中的第5章。在本章中，我们将“琐事”定义为与维护服务有关的重复，可预测，恒定的任务流。

对于任何管理生产服务的团队来说，琐事似乎都是不可避免的。系统维护不可避免地需要一定数量的部署，升级，重新启动，警报分类等。如果不加检查和不加考虑，这些活动会很快消耗团队。Google将SRE团队在运维工作(包括劳动密集型和非劳动密集型工作)上花费的时间限制为50%(有关原因的更多信息，请参见我们的第一本书中的第5章)。尽管此目标可能不适合您的组织，但将琐事设置为上限仍然是一个优势，因为识别和量化辛勤工作是优化团队时间的第一步。

### 什么是琐事？

劳动趋于落在通过以下特征度量的范围上，这些特征在我们的第一本书中有所描述。在此，我们为每种琐事特性提供一个具体示例：

#### 手动

当Web服务器上的tmp目录达到95%的利用率时，工程师Anne登录到服务器并搜索文件系统以删除多余日志文件。

#### 重复

完整的tmp目录不太可能是一次性事件，因此修复该目录的任务是重复的。

#### 可自动<sup>43</sup>

如果您的团队拥有包含诸如“登录X，执行此命令，检查输出，如果看到...，则重新启动Y”之类的内容的补救文档，这些说明实际上是具有软件开发技能的人的伪代码！在tmp目录取例中，该解决方案已部分自动化。通过不需要人工来运行脚本来完全自动化问题检测和补救，会更好。更妙的是，提交补丁，以使软件不再以此方式破坏。

#### 非战术/反应性

当您收到过多的“磁盘已满”和“服务器关闭”警报时，它们会使工程师从高价值的工程设计中分散注意力，并可能掩盖其他更高严重性的警报。结果，服务的健康受到损害。

#### 缺乏持久价值

完成任务通常会带来令人满意的成就感，但是从长远来看，这种重复的满足感不是积极的。例如，关闭该警报生成的故障单可确保用户查询继续进行，HTTP请求继续以状态代码<400进行服务，这很好。但是，今天解决故障单并不能防止将来出现此问题，因此投资回报期很短。

#### 增长速度至少与其来源一样快

许多类型的运维工作的增长速度(或快于基础架构的大小)。例如，您可以预期执行硬件维修所花费的时间会随着服务器机群的大小而逐步增加。物理维修工作可能会不可避免地随着机器数量的增加而扩展，但是辅助任务(例如，更改软件/配置)不一定必须如此。

琐事的来源可能并不总是符合所有这些标准，但请记住，琐事的形式多种多样。除上述特征外，还要考虑特定工作对团队士气的影响。人们是喜欢做某件事并发现它有回报吗？还是因为被认为是无聊或无趣的工作而经常被人们忽视的工作类型？<sup>44</sup>琐事可以慢慢削弱团队士气。花在琐事工作上的时间通常不是花在批判性思考或表达创造力上的时间。减少琐事是承认工程师的努力在可能进行人类判断和表达的领域中得到了更好的利用。

### 例: 手动响应琐事

由Facebook生产工程经理John Looney撰写

始终在一个SRE的心中

并非总是很清楚某些工作是琐事。有时，"创造性"的解决方案-编写临时解决方法-是不正确的选择。理想情况下，您的组织应该对根本原因修复给予奖励，而不是仅仅掩盖问题的修复。

加入Google之后(2005年4月)，我的第一个任务是登录损坏的计算机，调查其损坏的原因，然后进行修复或将其发送给硬件技术人员。这项任务似乎都很简单，直到我意识到任何时候都有超过20,000台损坏的机器！

我研究的第一台损坏的计算机的根文件系统已完全装有来自Google修补的网络驱动程序的数以十亿计的废话日志。我发现又有数千台损坏的机器存在相同的问题。我与队友分享了解决该问题的计划：我会编写一个脚本ssh到所有损坏的机器中，并检查根文件系统是否已满。如果文件系统已满，脚本将截断/var/log中大于1MB的所有日志，然后重新启动syslog。

我的队友对我的计划的热情不足使我停下来。他指出，最好在可能的情况下解决根本原因。从中长期来看，编写掩盖问题严重性的脚本会浪费时间(通过不解决"实际"问题)，并可能在以后引起更多问题。

分析表明，每台服务器每小时的成本约为1美元。根据我的思路，成本不应该是最重要的指标吗？我没有考虑过如果解决此症状，就没有动力解决根本原因：内核团队的发布测试套件没有检查这些机器产生的日志量。

高级工程师将我引向内核源代码，以便我可以找到令人反感的代码行，并针对内核团队记录一个错误以改进他们的测试套件。我的客观成本/收益分析表明，该问题导致Google每小时损失1000日元，这使开发人员确信开发人员可以使用我的补丁程序解决此问题。

那天晚上，我的补丁程序变成了一个新的内核版本，第二天，我将其发布到了受影响的机器上。内核团队在下周晚些时候更新了他们的测试套件。现在，我不再感到每天早上都需要修复这些机器而短期内啡肽打击了，而是让我更加高兴地知道，我已经正确地解决了该问题。

## 测量琐事

您如何知道您的琐事工作量是多少？一旦您决定采取行动减少劳力，您如何知道自己的努力成功还是合理？许多SRE团队结合经验和直觉来回答这些问题。尽管此类策略可能会产生结果，但我们可以对其进行改进。

经验和直觉不是可重复的，客观的或可转移的。同一团队或组织的成员通常会因为琐事而付出的工程努力量得出不同的结论，因此对补救工作的优先级不同。此外，减少琐事的工作可能长达数个季度甚至数年(如本章中的一些案例研究所示)，在此期间，团队的工作重点和人员可能会发生变化。为了长期保持关注并证明成本合理，您需要客观地衡量进度。通常，团队必须从几名候选人中选择一个减少琐事的项目。客观的工作量衡量方法使您的团队可以评估问题的严重性并确定优先级，以实现最大的工程投资回报。

在开始减少琐事的项目之前，重要的是分析成本与收益的关系，并确认通过消除琐事所节省的时间(至少)与在首先开发然后维护自动化解决方案所花费的时间成比例(图6-1)。通过将节省的小时数与投资的小时数进行简单的比较，看起来“无利可图”的项目可能仍然很值得进行，因为自动化有许多间接或无形的好处。潜在的好处可能包括：

- 随着时间的推移，工程项目工作的增长，其中一些将进一步减少工作量
- 提高团队士气，减少团队损耗和倦怠
- 更少的中断上下文切换，提高了团队生产力
- 提高了流程的清晰度和标准化
- 增强团队成员的技术技能和职业发展
- 减少培训时间

-更少的人为失误造成的停机

- 提高了安全性

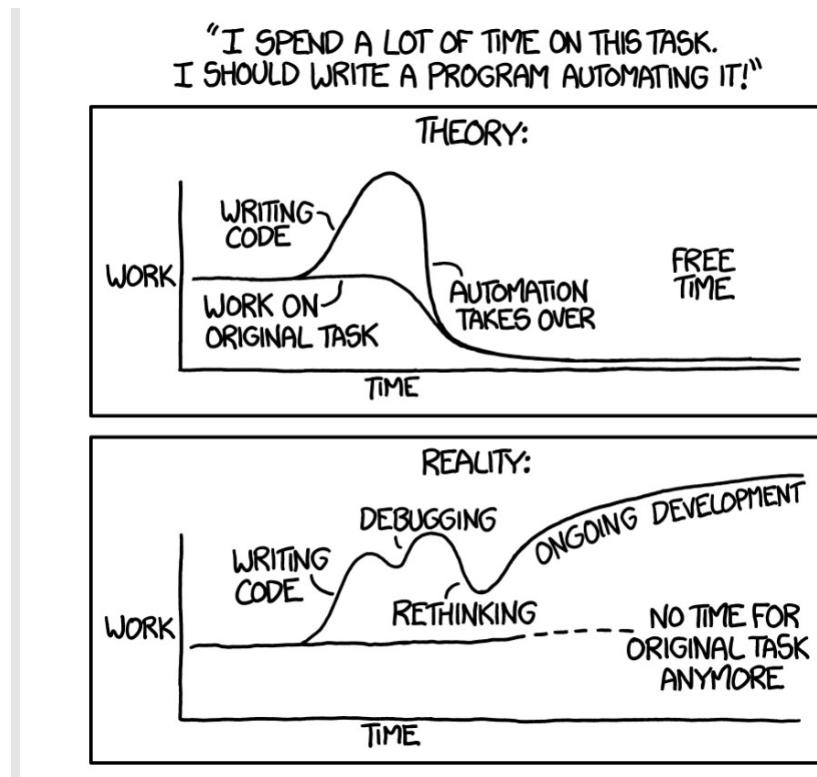


图6-1. 估算您将花费在减少琐事上的时间，并确保收益超过成本(来源:[xkcd.com/1319/](http://xkcd.com/1319/))

那么，我们如何建议您测量琐事呢？

1. 识别它。第一本SRE书籍的第5章提供了有关在操作中识别工作负担的指南。最能确定琐事工作的人员取决于您的组织。理想情况下，他们将是利益相关者，包括将执行实际工作的利益相关者。

2. 选择一个适当的度量单位，以表示应用于此琐事的“人力”总量。分钟和小时是自然的选择，因为它们是客观的并且得到普遍理解。确保考虑上下文切换的成本。对于分散的或分散的工作，可能需要使用其他易于理解的工作方式。度量单位的一些示例包括应用的补丁，完整的故障单，手动生产更改，可预测的电子邮件交换或硬件操作。只要该单位是客观，一致且广为人知的，它就可以用来衡量工作量。
3. 在减少琐事之前，期间和之后，持续跟踪这些测量结果。使用工具或脚本简化测量过程，以便收集这些测量结果不会造成额外的麻烦！

## 琐事分类法

琐事都藏在一天一天乏味中，就像摇摇欲坠的桥梁或漏水的大坝一样。本节中的类别并不详尽，但代表了一些常见的辛苦类别。这些类别中的许多似乎都是“正常”的工程工作，而且确实如此。将琐事视为频谱而不是二分类会很有帮助。

## 业务流程

这可能是最常见的琐事来源。也许您的团队会管理一些计算资源-计算，存储，网络，负载平衡器，数据库等等-以及提供该资源的硬件。您将与入职用户打交道，配置和保护他们的计算机，执行软件更新以及添加和删除服务器以达到中等容量。您还努力将成本或资源浪费降至最低。您的团队是机器的人机界面，通常与内部客户互动，后者根据需求为其提供故障单。您的组织甚至可能有多个故障单系统和工作摄取系统。

故障单琐事工作有点阴险，因为故障单驱动的业务流程通常可以实现其目标。用户得到了他们想要的东西，并且由于琐事工作通常分散在整个团队中，因此琐事工作不会很大声，显然需要进行补救。无论存在故障单驱动的过程如何，琐事工作都可能在附近悄悄积累。即使您没有明确计划自动化流程，您仍然可以执行诸如简化和精简流程之类的流程改进工作-这些流程以后将更易于自动化，同时也更易于管理。

## 生产中断

中断是对时间敏感的护卫任务，可以使系统保持运行状态。例如，您可能需要通过手动释放磁盘空间或重新启动泄漏内存的应用程序来解决某些资源(磁盘，内存，I/O)的严重短缺问题。您可能正在提出更换硬盘驱动器的请求，“踢掉”无响应的系统或手动调整容量以满足当前或预期的负载。通常，中断会将注意力从更重要的工作上移开。

## 发布指导

在许多组织中，部署工具会自动管理从开发到生产的发布。即使具有自动化，全面的代码覆盖范围，代码审查以及多种形式的自动化测试，此过程也并非总是能够顺利进行。根据工具和发布节奏，发布请求，回滚，紧急补丁以及重复或手动的配置更改，发布仍可能会产生麻烦。

## 迁移

您可能会发现自己经常从一种技术迁移到另一种技术。您可以手动执行此工作，也可以使用有限的脚本执行此工作，因为希望您只从X移到Y。迁移有多种形式，但一些示例包括数据存储，云供应商，源代码控制系统，应用程序库和工具的更改。

如果您手动进行大规模迁移，则迁移很可能涉及琐事。您可能倾向于手动执行迁移，因为这是一次性的工作。尽管您甚至可能会将其视为"项目工作"而不是"琐事工作"，但迁移工作也可以满足许多琐事的条件。从技术上讲，修改备份工具以使一个数据库可以与另一个数据库一起工作是软件开发，但这项工作基本上只是重构代码以将一个接口替换为另一个接口。这项工作是重复的，并且在很大程度上，备份工具的业务价值与以前相同。

## 成本工程和产能计划

无论您是拥有硬件还是使用基础架构提供商(云)，成本工程和容量规划通常都需要一些相关的工作。例如：

- 确保跨计算，资源或IOPS(每秒输入/输出操作)等资源的未来需求具有成本效益的基准或可突发功能。这可能会转换为采购订单，AWS预留实例或云/基础架构即服务合同谈判。
- 为产品发布或假期之类的关键高流量事件做准备(并从中恢复)。
- 审查下游和上游服务水平/限制。
- 针对不同的占用空间配置优化工作负载。(您要购买一个大盒子还是四个小盒子？)
- 根据专有云服务产品(适用于AWS的DynamoDB或适用于GCP的云数据存储)的计费细节优化应用程序。
- 重构工具，以更好地利用便宜的"现货"或"可抢占"资源。
- 处理超额预订的资源，无论是在上游与基础架构提供商还是与下游客户。

## 不透明架构的故障排除

分布式微服务体系结构现在很普遍，并且随着系统的分布越来越广泛，出现了新的故障模式。组织可能没有资源来构建复杂的分布式跟踪，高保真监控或详细的仪表板。即使企业确实拥有这些工具，它们也可能不适用于所有系统。故障诊断甚至可能需要登录到单个系统并使用脚本工具编写临时日志分析查询。

故障排除本身并不是天生的坏事，但您应该致力于将精力集中在新颖的故障模式上，而不是每周因系统结构脆弱而导致的同一类型的故障。对于可用性P的每个新的关键上游依赖性，由于故障的综合可能性，可用性降低了 $1 - P$ 。现在，添加了9个关键的4个9依赖关系的应该具有4个9可用性的服务现在是3个9服务。<sup>45</sup>

## 琐事管理策略

我们发现，如果您正在运行任何规模的生产系统，那么执行人工管理至关重要。一旦确定并量化了琐事，就需要制定消除琐事的计划。这些工作可能要花几周或几个季度才能完成，因此制定一个可靠的总体策略很重要。

从根本上消除琐事是最佳的解决方案，但是如果无法做到这一点，那么您就必须通过其他方式来处理琐事。在我们深入研究两个深入案例研究的细节之前，本节提供了一些一般策略，供您在计划减少工作量时考虑。正如您将在这两个故事中观察到的那样，每个团队(以及每个公司)的琐事之处都各不相同，但是不管具体情况如何，某些大小或口味的组织都可以采用某些通用策略。在至少一个后续案例研究中以具体方式说明了以下每种模式。

## 鉴别与衡量琐事

我们建议您采用一种数据驱动的方法来识别和比较琐事的来源，做出客观的补救决策，并量化减少琐事项目所节省的时间(投资回报)。如果您的团队遇到劳力超负荷的情况，请将减少劳力作为自己的项目。Google SRE团队经常跟踪bug中的琐事工作，并根据修复问题的成本和这样做所节省的时间来对琐事工作进行排名。有关技术和指导，请参见第96页上的“测量琐事”部分。

## 工程师从系统中解放出来

处理琐事的最佳策略是从源头上消除它。在投入精力管理现有系统和流程产生的琐事工作之前，请检查是否可以通过更改系统来减少或消除琐事工作。

在生产环境中运行系统的团队对于该系统的工作方式具有宝贵的经验。他们知道导致最多琐事的杂项和乏味的事情。SRE团队应通过与产品开发团队合作来应用此知识，以开发易于操作的软件，该软件不仅费力，而且可扩展性，安全性和灵活性更高。

## 拒绝琐事

被琐事拖累的团队应根据数据决定如何最佳地利用他们的时间和工程工作。根据我们的经验，虽然看起来似乎适得其反，但您应该考虑拒绝琐事的工作。对于给定的琐事工作，分析对琐事做出响应与不做出响应的成本。另一种策略是故意延迟琐事，以便为批处理或并行处理积累任务。以更大的总量使用琐事工作可以减少中断，并帮助您确定琐事模式，然后您可以将其消除。

## 使用SLO减少琐事

如第2章所述，服务应具有[文档化的服务水平目标\(SLO\)](#)。定义明确的SLO使工程师能够做出明智的决定。例如，如果您执行某些操作任务不会消耗或超过服务的错误预算，则可能会忽略它们。随着服务的增长，关注整体服务健康状况而不是单个设备的SLO更加灵活和可持续。有关编写有效SLO的指导，请参见第2章。

## 从人为支持的接口开始

如果您在许多极端情况或类型的请求中遇到特别复杂的业务问题，请考虑将部分自动化的办法作为迈向完全自动化的过渡步骤。通过这种方法，您的服务通常通过已定义的API接收结构化数据，但是工程师仍然可以处理一些结果操作。即使仍然需要一些人工工作，这种“幕后工程师”方法也使您可以逐步实现完全自动化。利用客户的意见，以更统一的方式收集这些数据；通过减少自由格式的请求，您可以更进一步以编程方式处理所有请求。这种方法可以与客户(现在可以清楚地指示所需的信息)来回节省，并避免您在完全映射和了解域之前过度设计大型解决方案。

## 提供自助服务方法

通过类型化的接口定义了服务产品之后(请参阅第102页的"以人为支持的界面开始")，转向为用户提供自助服务方法。您可以提供Web表单，二进制文件或脚本，API，甚至可以提供文档，告诉用户如何向服务的配置文件发出拉取请求。例如，与其要求工程师提交故障单以为其开发工作提供新的虚拟机，不如给他们提供触发提供的简单Web表单或脚本。允许该脚本优雅地降级为用于特殊请求或发生故障的故障单。<sup>46</sup> 人为支持的接口是与琐事斗争的良好开端，但服务所有者应始终致力于使自己的产品尽可能的成为自助服务。

## 获得管理层和同事的支持

在短期内，减少琐事的项目会减少可用于解决功能需求，性能改进和其他运维任务的人员。但是，如果成功地减少了工作量，从长远来看，团队将更健康，更快乐，并且有更多的时间进行工程改进。

对于组织中的每个人来说，重要的是要同意减少琐事是一个值得的目标。经理的支持对于保护员工免受新需求打扰至关重要。使用有关琐事劳动的客观指标来进行推论。

## 把减少琐事劳动提升为一个新的功能点

要创建减少琐事的强大业务案例，请寻找将您的策略与其他理想功能或业务目标结合在一起的机会。如果补充性目标(例如安全性，可扩展性或可靠性)对您的客户具有吸引力，那么他们将更愿意为不产生琐事的新目标放弃其现有的产生琐事系统。然后，减少琐事工作只是帮助用户的一个不错的副作用！

## 从小处着手然后改进

不要试图设计出消除所有琐事的完美系统。首先将一些高优先级的项目自动化，然后利用消除这种琐事的过程中所学到的经验教训，利用节省的时间改进解决方案。选择诸如MTTR(平均修复时间)之类的明确指标来衡量您的成功。

## 提高一致性

从规模上讲，多样化的生产环境变得难以管理。特殊设备需要耗时且容易出错的持续管理和事件响应。您可以使用"宠物与牛"的方法<sup>47</sup> 添加冗余并在您的环境中增强一致性。选择什么场景去考虑使用牛取决于组织的需求和规模。将网络链接，交换机，机器，机器机架乃至整个集群视为可互换单元可能是合理的。

将设备转换为牛的方法可能会有很高的初始成本，但可以在中长期内降低维护，灾难恢复和资源利用的成本。为多个设备配备相同的接口意味着它们具有一致的配置，可互换且需要较少的维护。各种设备的一致接口(用于转移流量，还原流量，执行关闭等)可实现更灵活，可扩展的自动化。

Google调整了商业激励措施，以鼓励工程团队在我们不断发展的内部技术和工具套件中实现统一。团队可以自由选择自己的方法，但是他们必须拥有由不受支持的工具或旧系统产生的琐事。

## 评估自动化范围内的风险

自动化可以节省大量的人力，但在错误的情况下，它也可能导致中断。通常，防御性软件始终是一个好主意。当自动化发挥管理员级别的权力时，防御性软件至关重要。在执行之前，应对每个动作的安全性进行评估。这包括可能会减少服务容量或冗余的更改。在实现自动化时，我们建议您遵循以下做法：

- 即使用户输入来自上游系统，也应谨慎处理用户输入-也就是说，请确保在上下文中仔细验证输入。
- 建立等同于人类操作员可能收到的间接警报类型的防护措施。防护措施可能像判断命令超时一样简单，也可能是对当前系统指标或当前中断次数的更复杂的检查。因此，机器和人工操作人员均应使用监控，警报和仪表系统。
- 请注意，即使是天真的实现的读取操作，也可能会增加设备负载并触发中断。随着自动化规模的扩大，这些安全检查最终可能会占据工作量。
- 最大限度地减少由于不完整的自动化安全检查而造成的停机影响。如果自动化在不安全的情况下运行，则应默认为人工操作。

## 自动完成琐事响应

一旦确定琐事劳动是可自动化的，就值得考虑如何最好地反映软件中的人工工作流程。您很少希望将人工工作流程直接转换为机器工作流程。还要注意，自动化不应消除人类对出现的问题的理解。

一旦对过程进行了详细记录，请尝试将手动工作分解为可以单独实现的组件，并使用这些组件来创建可组合的软件库，其他自动化项目以后可以重用这些库。正如即将进行的数据中心维修案例研究所说明的那样，自动化通常为重新评估和简化人工工作流程提供了机会。

## 使用开源和第三方工具

有时，您不必做所有工作来减少琐事。诸如一次性迁移之类的许多工作可能无法证明构建自己的定制工具是合理的，但是您可能不是第一个踏上这条路的组织。寻找机会使用或扩展第三方或开放源代码库以降低开发成本，或至少帮助您过渡到部分自动化。

## 使用反馈来改善

积极寻求与您的工具，工作流和自动化进行交互的其他人的反馈很重要。您的用户将根据对基础系统的理解，对您的工具做出不同的假设。用户对这些系统的了解越少，主动寻求用户的反馈就越重要。利用调查，用户体验(UX)研究和其他机制来了解您的工具的使用方式，并集成此反馈以在将来产生更有效的自动化。

人为输入只是您应考虑的反馈的一个维度。还要根据延迟，错误率，返工率和节省的人工时间(跨过程中的所有组)来衡量自动化任务的有效性。理想情况下，找到在自动化或减少琐事工作之前和之后可以比较的高级措施。

### 遗留系统

大多数负责SRE的工程师在工作中都遇到了至少一个遗留系统。这些较旧的系统通常会带来用户体验，安全性，可靠性或可伸缩性方面的问题。它们往往像一个神奇的黑匣子一样工作，因为它们“大部分工作”，但是很少有人了解他们的工作方式。它们令人恐惧且修改起来很昂贵，而且要使其保持运行通常需要大量繁琐的操作习惯。

脱离传统系统的旅程通常遵循以下路径：

1. **避免:** 直接解决这个问题的原因有很多：您可能没有资源来替换此系统。您认为您的业务成本和风险不值得付出替换成本。商业上可能没有任何更好的解决方案。规避实际上是选择接受技术债务，并从SRE原则转向系统管理。
2. **封装/增强:** 您可以带上SRE来构建围绕这些旧系统的抽象API，自动化，配置管理，监控和测试的外壳，这将使SA的工作分担。遗留系统仍然难以更改，但是现在您至少可以可靠地识别不良行为并在适当时候回滚。这种策略仍然可以避免，但是有点像将高息技术债务转换为低息技术债务。通常，为逐步替换做准备是权宜之计。
3. **替换/重构:** 更换旧系统可能需要大量的确定，耐心，沟通和记录。最好以增量方式进行。一种方法是定义一个公共接口，该接口位于旧系统的前面并对其进行抽象。此策略可帮助您使用诸如金丝雀或蓝绿色部署之类的发布工程技术[缓慢而安全地将用户迁移到替代方案](#)。通常，遗留系统的“规格”实际上仅由其历史用途来定义，因此，构建具有历史预期输入和输出的生产规模数据集有助于建立新系统不会偏离预期行为的信心（或以预期的方式出现分歧）。
4. **退役/监护权:** 最终，大多数客户或功能迁移到一个或多个替代方案。为了调整业务激励措施，尚未迁移的散客可以承担对遗留系统残余物的保管所有权。

## 实例探究

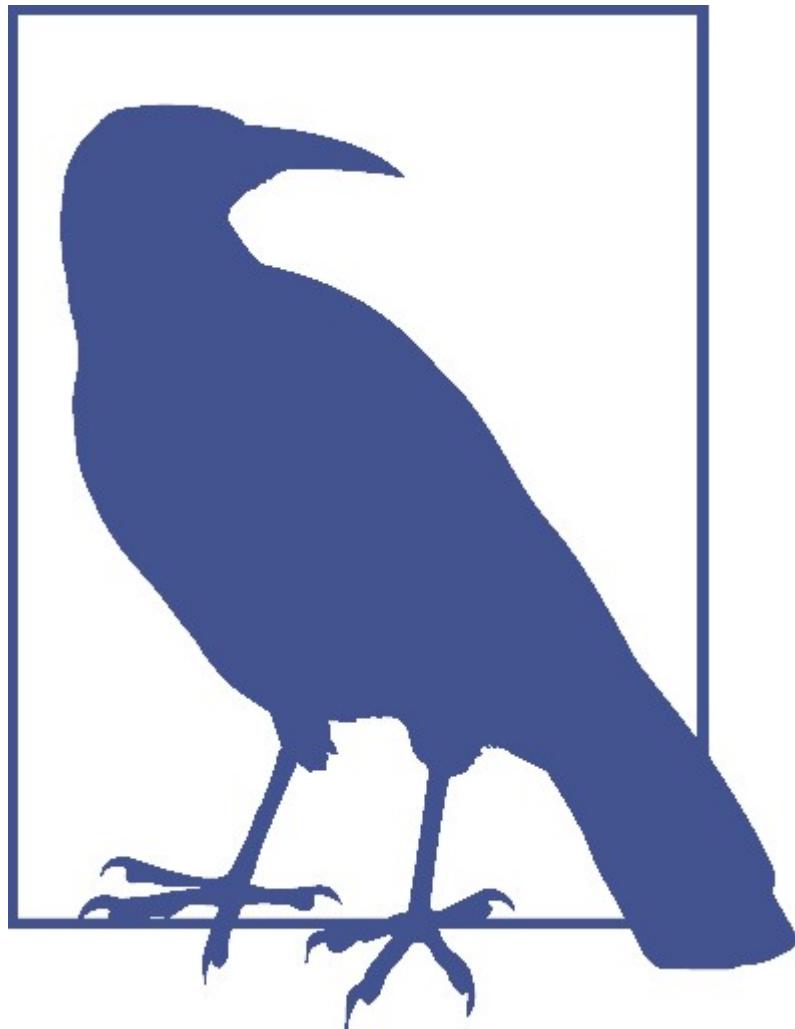
以下案例研究说明了刚才讨论的减少琐事的策略。每个故事都描述了Google基础架构的一个重要领域，该领域已经到了无法再通过人工工作来线性扩展的地步。随着时间的流逝，工程师工时的增加导致该投资的回报减少。您现在将大部分努力视为琐事。对于每个案例研究，我们都会详细介绍工程师如何识别，评估和缓解这种麻烦。我们还将讨论结果以及我们在此过程中学到的教训。

在第一个案例研究中，Google的数据中心网络存在扩展问题：我们拥有大量Google设计的组件以及用于监控，缓解和修复的链接。我们需要一种策略来最大程度地减少数据中心技术人员这项工作的繁琐性。

第二个案例研究的重点是一个团队，他们运行自己的“异常”专用硬件，以支持在Google内部已根深蒂固的劳动密集型业务流程。该案例研究说明了重新评估和替换昂贵的业务流程的好处。它显示出，即使稍有坚持和毅力，即使在大型组织机构的惯性约束下，也有可能转向其他选择。

综合起来，这些案例研究提供了前面介绍的每种减少劳动量策略的具体示例。每个案例研究均以相关的减少琐事策略列表开头。

## 案例研究1: 通过自动化减少数据中心的工作量



案例研究1中强调的琐事减少策略:

- 工程师琐事离开系统
- 从小开始，然后改善
- 增加一致性
- 使用SLO减少琐事
- 评估自动化中的风险
- 使用反馈来改善
- 自动化琐事响应

### 背景

该案例研究在Google的数据中心进行。与所有数据中心类似，Google的计算机连接到交换机，而交换机又连接到路由器。流量通过链接从这些路由器流入和流出，而这些链接又连接到Internet上的其他路由器。随着Google处理互联网流量的要求

不断增长，为该流量提供服务所需的计算机数量急剧增加。由于我们想出了如何高效，经济地服务大量流量的数据中心，因此其数据中心的范围和复杂性也在不断增长。这种增长将数据中心手动维修的性质从偶尔的有趣更改为频繁而死板的-这是两个琐事的信号。

Google首次开始运行自己的数据中心时，每个数据中心的网络拓扑都以少量的网络设备为特征，这些设备管理着大量机器的流量。单个网络设备故障可能会严重影响网络性能，但是相对较小的工程师团队却可以对少量设备进行故障排除。在此早期阶段，工程师调试了问题，并手动将流量从出现故障的组件转移开了。

我们的下一代数据中心拥有更多的计算机，并引入了具有折叠Clos拓扑的软件定义网络(SDN)，这大大增加了交换机的数量。图6-2显示了小型数据中心Clos交换网络的业务流的复杂性。设备数量的按比例增加意味着大量组件现在可能发生故障。尽管每个单独的故障对网络性能的影响都比以前少，但大量的问题开始使工程人员不堪重负。

除了要调试大量新问题外，复杂的布局还使技术人员感到困惑：哪些确切的链接需要检查？他们需要更换哪个线卡<sup>48</sup>？什么是一层交换机，而不是二层或三层交换机？关闭交换机会给用户带来麻烦吗？

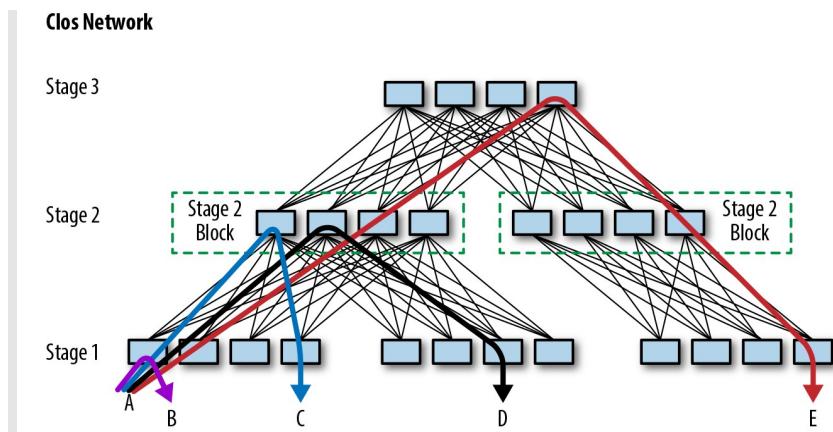


图6-2. 小型Clos网络，可支持第一层以下连接的480台机器

修复出现故障的数据中心线卡是一个明显的工作积压，因此我们将这项任务作为创建数据中心网络修复自动化的第一步。此案例研究描述了我们如何为第一代线卡（名为Saturn）引入维修自动化。然后，我们讨论下一代Jupiter fabrics线卡引入的改进。

如图6-3所示，在执行自动化项目之前，数据中心线卡修复工作流程中的每个修复都需要工程师执行以下操作：

1. 检查从受影响的交换机移出流量是否安全。
2. 将流量从故障设备移开("引流"操作)。
3. 执行重新引导或修复(例如更换线卡)。
4. 将流量转移回设备("引入"操作)。

排水，排水和修理设备的这项不变而重复的工作是琐事的教科书示例。这项工作的重复性质带来了它自己的问题-例如，工程师可能需要在处理线卡时进行多任务处理，同时还要调试更具挑战性的问题。结果，分心的工程师可能会意外地将未配置的交换机引入网络。

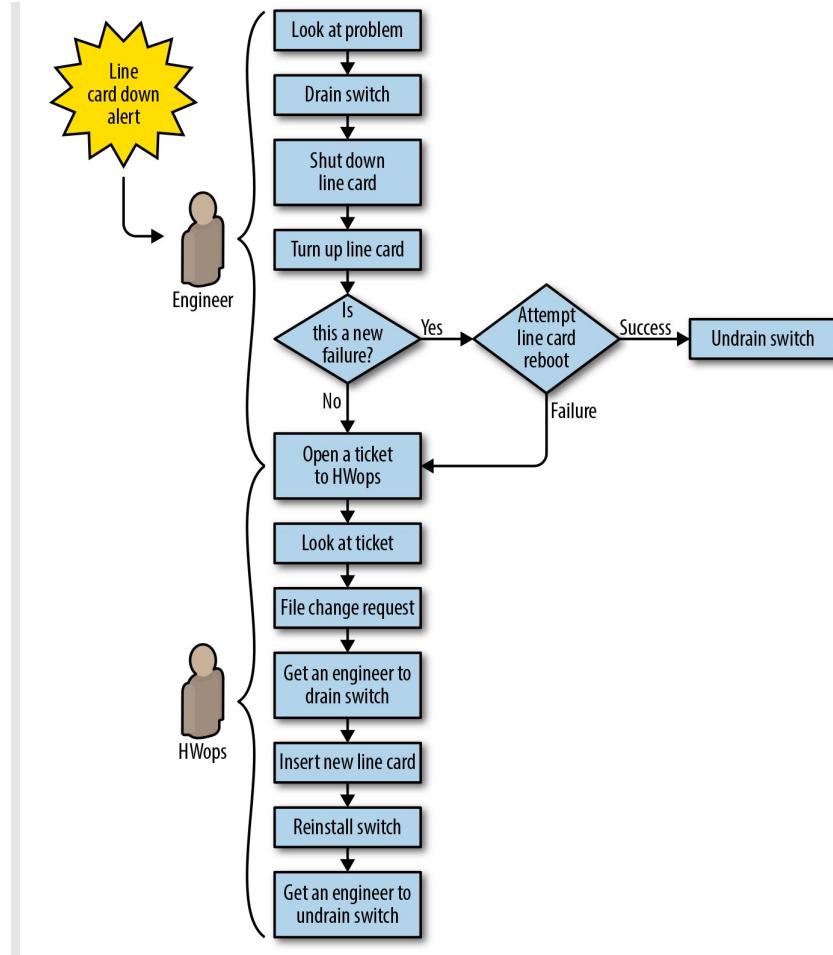


图6-3. 自动化之前的数据中心(Saturn)线卡维修工作流程: 所有步骤都需要人工操作

## 问题陈述

数据中心维修问题空间具有以下维度:

- 我们无法以足够快的速度发展团队，以跟上失败的数量，我们也不能足够快地解决问题，以防止对架构造成负面影响。
- 重复执行相同的步骤，经常会引入过多的人为错误。
- 并非所有的线卡故障都有相同的影响。我们没有办法确定更严重的故障的优先级。
- 有些故障是暂时的。我们希望可以选择重新启动线卡或重新安装交换机，作为维修时的第一步。理想情况下，如果问题再次发生，我们可以以编程方式捕获该问题，并标记设备以进行替换
- 新拓扑要求我们在采取行动之前手动评估隔离能力的风险。每次手动风险评估都是人为错误的机会，可能会导致中断。地板上的工程师和技术人员没有很好的方法来评估计划的维修会影响多少设备和链接。

## 我们决定要做的

我们没有将每个问题分配给工程师进行风险评估，流失，取消和确认，而是决定创建一个自动化框架，在适当时候与现场技术人员配合使用，可以以编程方式支持这些操作。

## 设计第一次尝试: Saturn线卡维修

我们的高级目标是构建一个系统，以响应网络设备上检测到的问题，而不是依靠工程师对这些问题进行分类和修复。我们没有向工程师发送“线路卡已关闭”警报，而是编写了该软件来请求排空(以消除流量)并为技术人员创建案例。新系统具有一些值得注意的功能:

- 我们尽可能利用了现有工具。如图6-3所示，我们的警报已经可以检测到结构线路卡上的问题；请参阅第11页的“警报”。我们将警报重新用于触发自动修复。新的工作流程还重新调整了我们的故障单系统的用途，以支持网络维修。
- 我们内置了自动风险评估功能，以防止在排水期间意外隔离设备，并在需要时触发安全机制。这消除了人为错误的巨大根源。
- 我们采用了由软件跟踪的警告政策：第一次失败(或警告)仅重新启动了卡并重新安装了软件。第二次故障触发了卡的更换，并完全退还给了供应商。

## 实施

新的自动化工作流程(如图6-4所示)进行如下：

1. 检测到有问题的线卡，并将症状添加到数据库中的特定组件。
2. 维修服务发现问题并启用交换机维修。该服务执行风险评估以确认操作不会隔离任何容量，然后：
  - a. 排空整个交换机的流量。
  - b. 关闭线卡。
  - c. 如果这是第一次失败，请重新引导卡并拔下交换机的电源，将服务恢复到交换机。至此，工作流程完成。
  - d. 如果这是第二次失败，则工作流程进入步骤3。
3. 工作流管理器检测到新案例并将其发送到维修案例库，以供技术人员提出索赔。
4. 技术人员声明了这种情况，在用户界面中看到一个红色的“停止”(表明在开始维修之前需要先排空交换机)，并分三步执行维修：
  - a. 通过技术人员UI中的“预备组件”按钮启动机箱排水。
  - b. 等待红色的“停止”清除，表明排水已完成且箱体可操作。
  - c. 装回卡并合上机箱。
5. 自动维修系统使线卡再次启动。稍等片刻，让卡有时间进行初始化之后，工作流管理器触发一项操作，以恢复到交换机的流量并关闭维修案例。

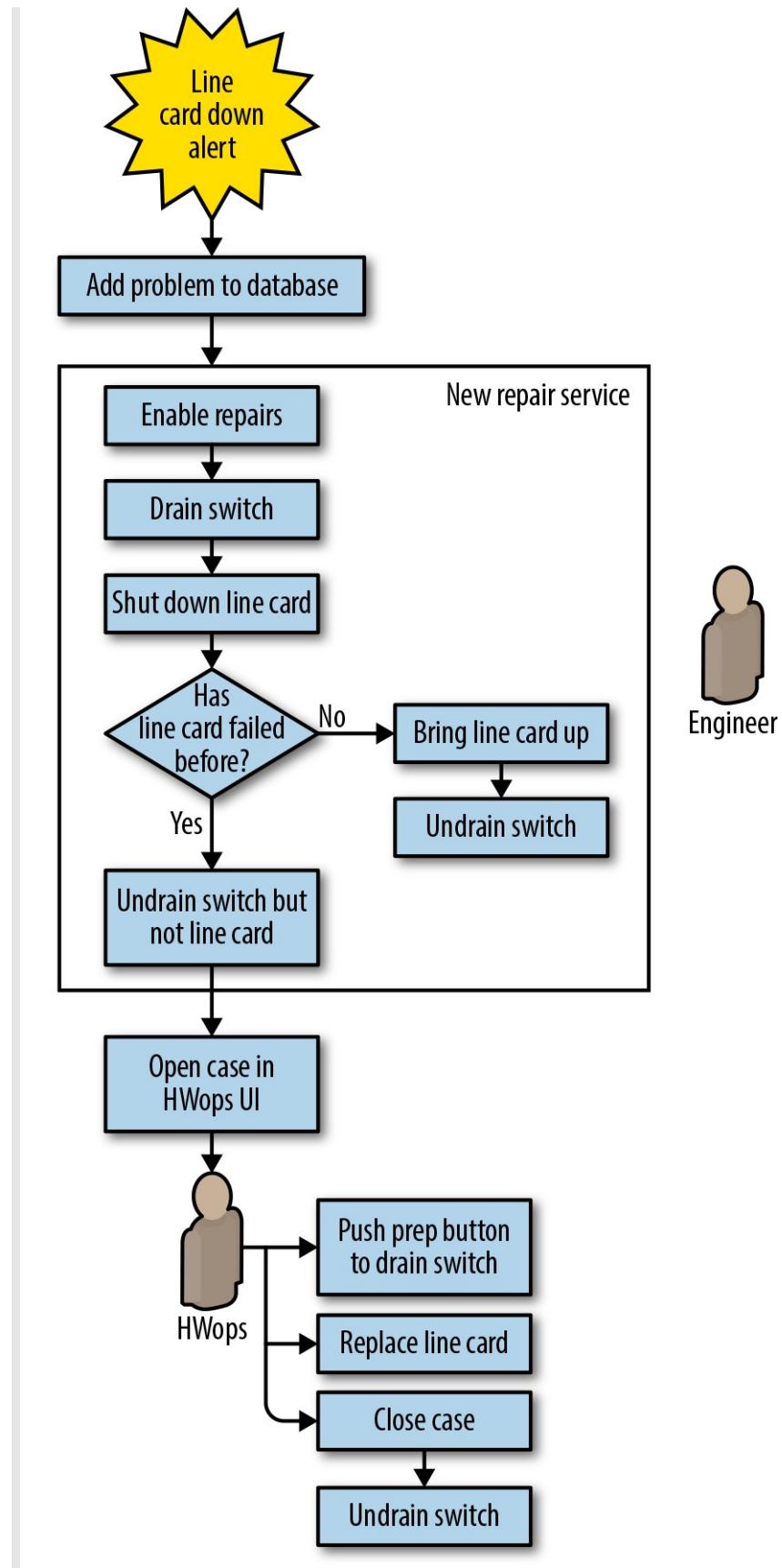


图6-4. 具有自动化功能的Saturn线卡维修工作流程: 只需按一下按钮并更换线卡即可进行手动操作

新系统使工程团队摆脱了繁重的工作，使他们有更多时间从事其他地方的生产性项目：研究下一代Clos拓扑Jupiter。

## 设计第二效能：Saturn线卡维修与Jupiter线卡维修

数据中心的容量需求几乎每12个月就翻一番。结果，我们的下一代数据中心架构Jupiter比以前的任何Google架构大六倍以上。问题的数量也增加了六倍。Jupiter提出了维修自动化的扩展挑战，因为每层中数千个光纤链路和数百个线卡可能会发生故障。幸运的是，潜在故障点的增加伴随着更大的冗余度，这意味着我们可以实施更具雄心的自动化。如图6-5所示，我们保留了Saturn的一些常规工作流程，并添加了一些重要的修改：

- 在自动排水/重新启动周期确定我们要更换硬件后，我们将硬件发送给了技术人员。但是，我们并不需要技术人员通过“按下准备按钮以排放开关”来启动排放，而是在发生故障时自动排放掉整个开关。
- 我们增加了自动化功能，用于安装和推送在更换组件后启用的配置。
- 我们启用了自动化功能，以在不卸下交换机电源之前验证维修是否成功。
- 除非绝对必要，否则我们将精力集中在恢复交换机上，而无需技术人员参与。

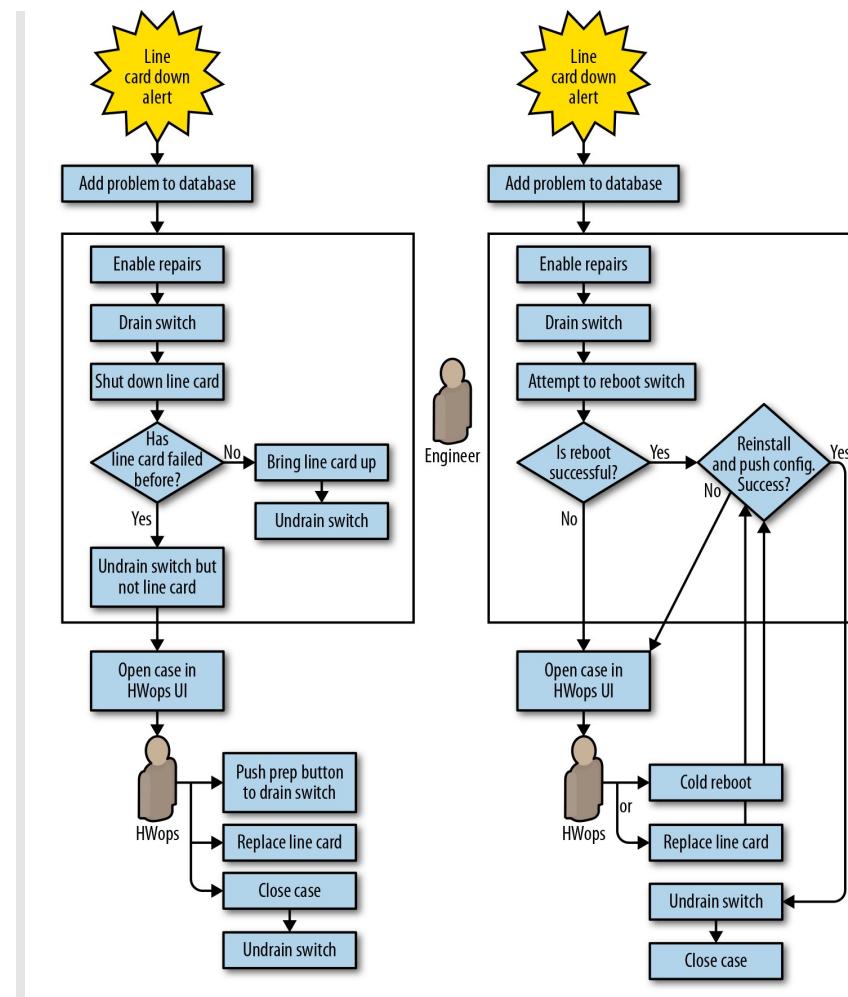


图6-5.Saturn线卡自动化下线(左)与Jupiter自动化(右)

## 实施

对于Jupiter交换机上的每个线路卡问题，我们采用了一种简单统一的工作流程: 断开交换机的电源，将其排空并开始维修。

自动化执行以下操作:

1. 检测到问题关闭，并将症状添加到数据库。
2. 维修服务解决了问题并启用了交换机的维修服务: 耗尽整个交换机，并增加耗尽原因。
  - a. 如果这是六个月内第二次失败，请继续执行步骤4。
  - b. 否则，请继续执行步骤3。
3. 尝试(通过两种不同的方法)对交换机重新通电。
  - a. 如果重启成功，请运行自动验证，然后安装和配置交换机。删除修复原因，从数据库中清除问题，然后拔出交换机电源。
  - b. 如果之前的健全性检查操作失败，请通过说明消息将案件发送给技术人员。
4. 如果这是第二次失败，则将案例直接发送给技术人员，要求提供新的硬件。发生硬件更改后，请运行自动验证，然后安装和配置交换机。删除修复原因，从数据库中清除问题，然后拔出交换机电源。

这种新的工作流程管理是对先前维修系统的完全重写。同样，我们尽可能利用现有工具:

- 配置新交换机的操作(安装和验证)与验证交换机已被替换所需的操作相同。
- 快速部署新结构需要以编程方式对BERT<sup>49</sup>和电缆审核<sup>50</sup>的能力。在恢复流量之前，我们重用了该功能，以在已经修复的链路上自动运行测试模式。这些测试通过识别故障链接进一步提高了性能。

下一步的逻辑改进是自动减轻和修复Jupiter交换线路卡上的内存错误。如图6-6所示，在自动化之前，此工作流程在很大程度上取决于工程师来确定故障是硬件还是软件相关的，然后排空并重启交换机，或者安排适当的维修。

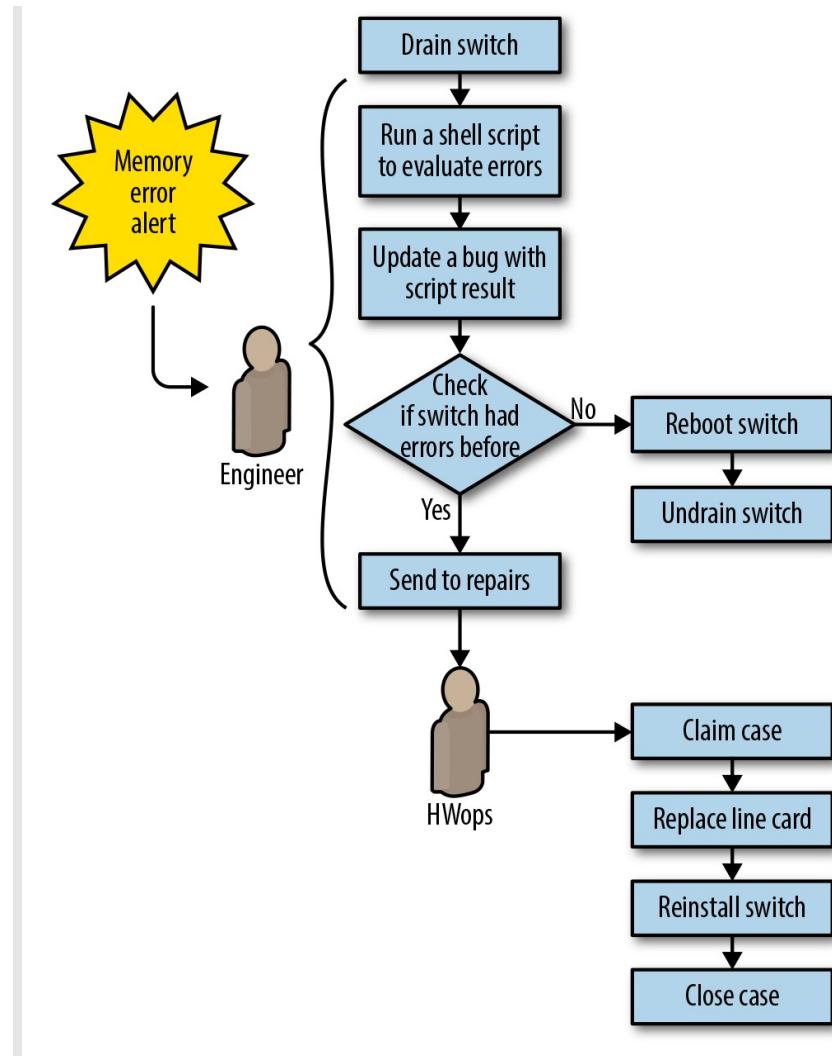


图6-6. 自动化之前的Jupiter内存错误修复工作流程

我们的自动化不再需要尝试对内存错误进行故障排除，从而简化了维修工作流程（请参阅第119页的“有时不完美的自动化就足够好了”，以了解为什么这样做很有意义）。相反，我们以与处理故障线卡相同的方式处理内存错误。为了将自动化扩展到内存错误，我们仅需在配置文件中添加另一种症状，以使其对新的问题类型起作用。

图6-7描述了内存错误的自动工作流程。

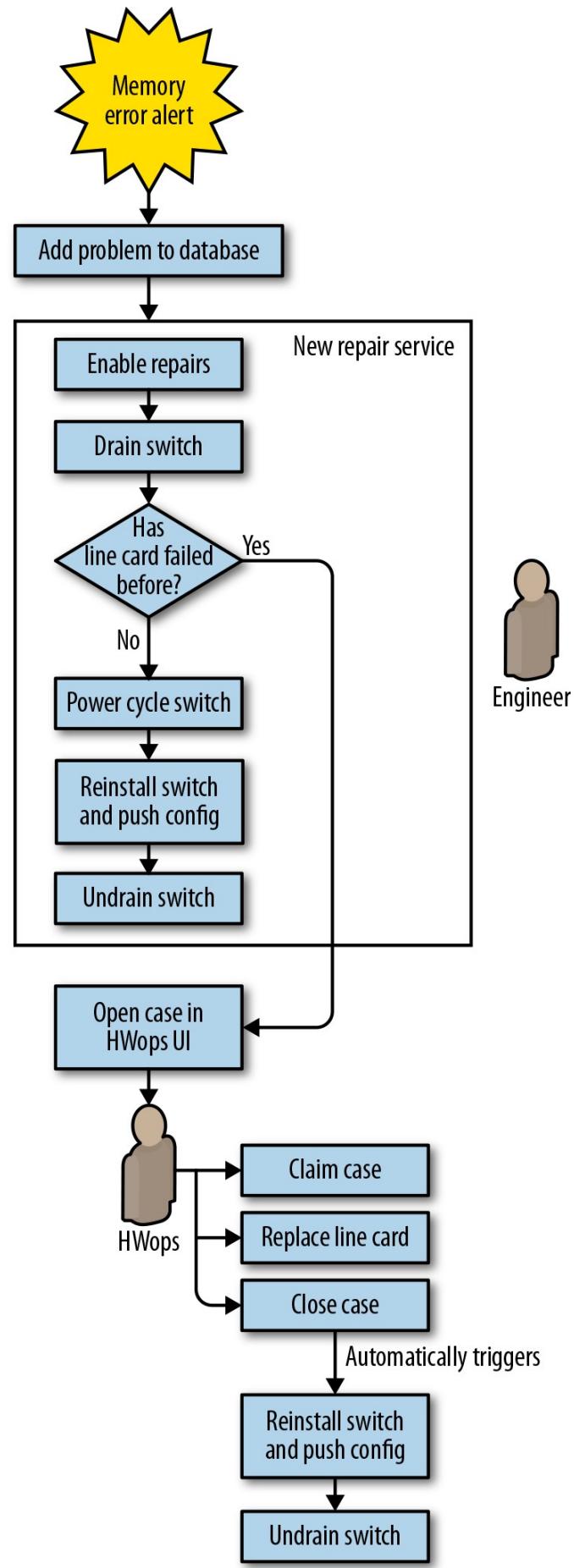


图6-7. 具有自动化功能的Jupiter内存错误修复工作流程

## 经验教训

在我们致力于网络修复自动化的几年中，我们学到了很多有关如何有效减少劳力的一般课程。

### UI不应引入过度开销或复杂性

对于基于Saturn的线卡，更换线卡需要排干整个交换机。在维修过程中尽早排空整个交换机意味着在等待更换零件和技术人员的同时会丢失交换机上所有线卡的工作能力。我们在用户界面中引入了一个称为"Prep组件"的按钮，该按钮使技术人员可以在准备更换卡之前就从整个交换机中抽出流量，从而消除了其余交换机的不必要的停机时间(请参阅"将准备按钮按下排水开关"(图6-5)。

UI和修复工作流的这一方面引入了许多意外问题:

- 按下按钮后，技术人员没有获得有关排水进度的反馈，而只是需要等待许可才能继续进行。
- 按钮未与开关的实际状态可靠地同步。结果，有时排水的交换机得不到维修，或者技术人员通过不排水的交换机来中断流量。
- 出现问题时，未启用自动化功能的组件返回通用的"联系工程"消息。较新的技术人员不知道与可以帮助的人联系的最佳方法。可以联系的工程师并不总是立即空闲。

为了响应用户报告以及功能复杂性导致的回归问题，我们设计了未来的工作流程，以确保在技术人员到达交换机之前，交换机是安全的并且可以维修。

### 不要依赖人类的专业知识

我们过于依赖经验丰富的数据中心技术人员来确定我们系统中的错误(例如，当软件指示可以安全进行维修但实际上不费力地进行更换时)。这些技术人员还必须手动执行多项任务，而不会受到自动化的提示。

经验很难复制。在一个影响特别大的事件中，技术人员决定通过在等待数据中心维修的每个线卡上启动并发消耗来加快"按下按钮并等待结果"的体验，从而导致网络拥塞和用户可见的数据包丢失。我们的软件没有预料到并阻止此操作，因为我们没有与新技术人员一起测试自动化。

### 设计可重复使用的组件

尽可能避免一体化设计。从可分离的组件构建复杂的自动化工作流，每个组件处理一个明确的任务。我们可以轻松地为每个连续的织物世代重复使用或适应早期Jupiter自动化的关键组件，并且当我们可以基于已经存在的自动化进行构建时，添加新功能会更加容易。Jupiter型网络的连续变化可以利用早期迭代中完成的工作。

### 不要想太多问题

我们对Jupiter线卡的内存错误问题进行了过度分析。在我们进行精确诊断的尝试中，我们试图将软件错误(可通过重新启动修复)与硬件错误(需要更换卡)区别开来，并确定影响流量的错误与未影响流量的错误。我们花了近三年的时间(2012-2015)收集了有关650多个离散内存错误问题的数据，然后才意识到此练习可能是过大的，或者至少不应该阻止我们的维修自动化项目。

一旦决定对发现的任何错误采取行动，就可以使用我们现有的维修自动化程序来实施简单的策略，以响应内存错误而耗尽，重新引导和重新安装交换机。如果问题再次发生，我们得出的结论是该故障很可能是基于硬件的，因此需要更换组件。我们在四分之一的过程中收集了数据，发现大多数错误是暂时的-多数交换机在重新引导和重新安装后都已恢复。我们不需要额外的数据来执行修复，因此延迟三年实施自动化是不必要的。

### **有时不完美的自动化就足够好了**

虽然可以在不断开BERT的情况下验证链接的能力很方便，但是BERT工具不支持网络管理链接。我们将这些链接添加到现有的链接修复自动化中，并进行了检查，以使它们可以跳过验证。我们很乐意绕过验证，因为这些链接不会承载客户流量，如果验证很重要，我们可以稍后添加此功能。

### **维修自动化并非一劳永逸**

自动化的寿命可能很长，因此请确保在人们离开并加入团队时计划项目的连续性。新工程师应在旧系统上接受培训，以便他们可以修复错误。由于Jupiter的零件短缺，基于Saturn的网络在最初预定的寿命终止日期后仍存在很长时间，因此要求我们在Saturn的整个使用寿命中进行一些改进。

一旦被采用，自动化可能会在很长一段时间内根深蒂固，带来积极和消极的后果。尽可能将自动化设计为以灵活的方式发展。依靠灵活的自动化使系统难以更改。基于策略的自动化可以通过将意图与通用实现引擎明确分开来提供帮助，从而使自动化能够更加透明地发展。

### **建立深入的风险评估和防御机制**

在为Jupiter构建了新工具之后，我们决定了执行排水操作的风险之后，遇到的复杂性使我们引入了深度防御的二次检查。第二次检查为受影响的链接数设置了上限，为受影响的设备设置了另一个限制。如果我们超过了两个阈值，则会自动打开一个跟踪错误，要求进一步调查。我们随着时间的推移调整了这些限制，以减少误报。虽然我们最初认为这是暂时的措施，直到主要风险评估稳定下来，但事实证明，辅助检查对于确定由于断电和软件错误而引起的非典型维修率很有用(例如，请参见《站点可靠性工程》中的自动化: "大规模启用失败" )。

### **获得错误预算和经理支持**

维修自动化有时会失败，尤其是在首次引入时。管理支持对于保护项目和增强团队的毅力至关重要。我们建议为琐事自动化建立错误预算。您还应该向外部利益相关者解释，尽管存在故障风险，但是自动化是必不可少的，并且自动化可以不断提高可靠性和效率。

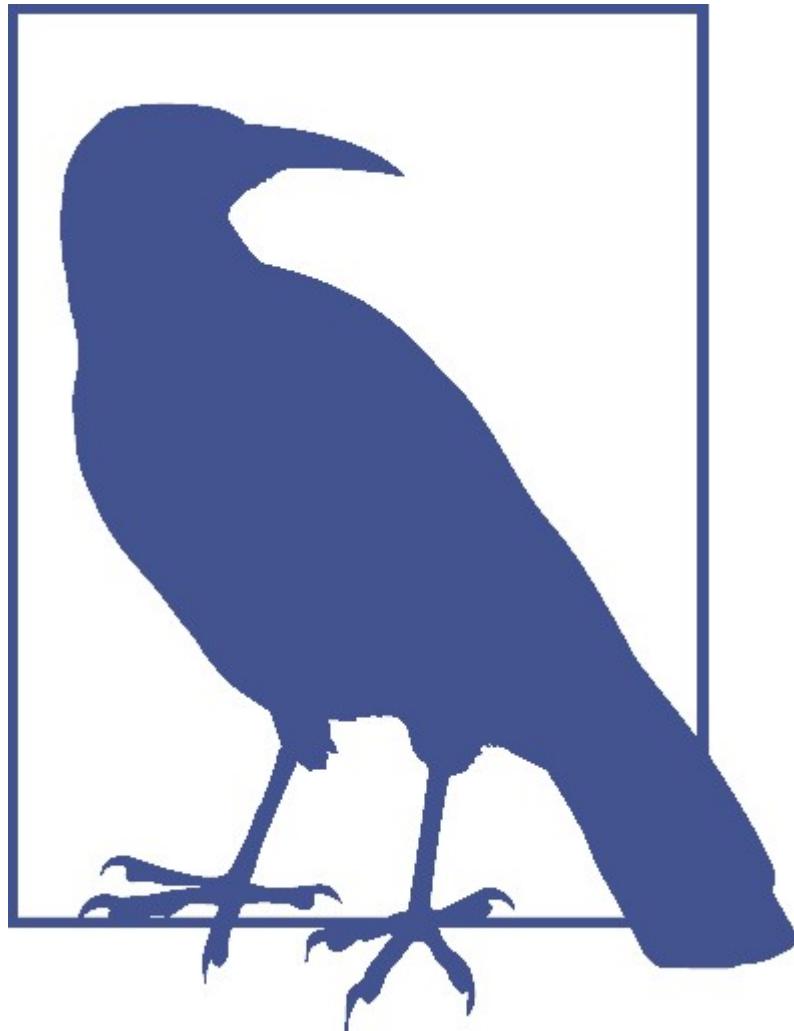
### **全面思考**

最终，要实现自动化的方案的复杂性是要克服的真正障碍。在进行自动化之前重新检查系统-可以先简化系统或者工作流程吗？

请注意您要自动化的工作流程的所有方面，而不仅是那些会亲自为您造成麻烦的方面。与直接参与工作的人员进行测试，并积极寻求他们的反馈和帮助。如果他们犯了错误，请找出如何使您的UI更加清晰，或需要进行哪些其他安全检查。确保您的自动化不会造成新的麻烦-例如，通过打开需要人注意的不必要的故障单。为其他团队创造问题将增加对未来自动化工作的抵制。

## 案例研究2: 停运由Filer支持的主页

### 目录



案例研究2中强调的琐事减少策略:

- 考虑停用旧系统
- 促进减少琐事作为主要特点
- 获得管理层和同事的支持
- 拒绝琐事
- 从人为支持的接口开始
- 提供自助服务方式
- 从小开始，然后改善
- 使用反馈来改善

### 背景

在Google成立初期，公司数据存储(CDS)SRE团队向所有Google员工提供了主目录。与企业IT中常见的Active Directory漫游配置文件相似，Google员工可以在工作站和平台之间使用相同的主目录。CDS团队还为共享存储空间中的跨团队协作提供了“团队共享”。我们通过NFS/CIFS(或“文件管理器”)上的Netapp存储设备提供了主目录和团队共享。此存储在操作上非常昂贵，但为Google员工提供了急需的服务。

## 问题陈述

随着时间的流逝，这些文件管理器解决方案大多被其他更好的存储解决方案所淘汰：我们的版本控制系统(Piper<sup>51</sup> / Git-on-borg<sup>52</sup>)，Google Drive，Google Team Drive，Google Cloud Storage，以及一个内部的，共享的，全局分布的文件系统x20。这些替代方案之所以优越，有以下几个原因：

- NFS/CIFS协议从未设计为可以在WAN上运行，因此用户体验会迅速下降，甚至只有几十毫秒的延迟。这给远程工作人员或遍布全球的团队带来了问题，因为数据只能存在于一个位置。
- 与替代产品相比，这些设备的运行和扩展成本很高。
- 要使NFS/CIFS协议与Google的Beyond Corp<sup>53</sup>网络安全模型兼容，将花费大量工作。

与本章最相关的是，主目录和团队共享需要大量的琐事工作。存储供应的许多方面都是凭故障单驱动的。尽管这些工作流程通常是部分编写脚本的，但它们代表了CDS团队的大量工作。我们花了很多时间来创建和配置共享，修改访问权限，对最终用户问题进行故障排除以及执行开和关来管理容量。除配置，更新和备份外，CDS还管理该专用硬件的设置，机架和电缆连接过程。由于延迟的要求，我们经常不得不在远程办公室而不是Google数据中心进行部署-有时这需要团队成员走很长一段距离才能管理部署。

## 我们决定要做什么

首先，我们收集数据：CDS创建了一个名为Moonwalk的工具，用于分析员工如何使用我们的服务。我们收集了传统的商业智能指标，例如每日活跃用户(DAU)和每月活跃用户(MAU)，并提出了以下问题：“哪些工作族实际上使用他们的主目录？”和“每天使用文件管理器的用户中，他们最多访问哪种文件？”Moonwalk与用户调查相结合，证实了归档者当前所满足的业务需求可以通过具有较低运营开销和成本的替代解决方案得到更好的满足。另一个引人入胜的商业原因导致我们离开了文件管理器：如果我们可以将大多数文件管理器用例迁移到G Suite/GCP，那么我们可以利用我们学到的经验教训来改进这些产品，从而使其他大型企业也可以迁移到G Suite/ GCP。

没有哪个替代方案可以满足当前所有文件管理器用例。但是，通过将问题分解为较小的可寻址组件，我们发现总体而言，少数替代方案可以覆盖我们的所有用例。替代解决方案更加专业，但是与通用文件管理器解决方案相比，每种解决方案都提供了更好的用户体验。例如：

x20<sup>54</sup>

是团队在全球范围内共享二进制等静态工作的绝佳方式

G Suite Team Drive

可以很好地用于Office文档协作，并且比NFS更能容忍用户延迟

#### Google的Colossus文件系统

允许团队比NFS更安全，更可扩展地共享大型数据文件

#### Piper/Git-on-Borg

更好地同步点文件(工程师的个性化工具首选项)

#### 一种新的历史即服务工具

可以托管跨工作站的命令行历史记录

当我们对用例进行分类并找到替代方案时，退役计划就形成了。

## 设计与实现

远离文件编档员是一项持续不断的，反复的，多年的努力，需要多个内部项目：

#### Moira

主目录退役

#### Tekmor

迁移主目录用户的长尾巴

#### Migra

团队共享停运

#### Azog

淘汰主目录/共享基础结构和关联的硬件

本案例研究的重点是第一个项目Moira。随后的项目建立在我们从Moira那里学到的东西上并为Moira创建的。

如图6-8所示，Moira由四个阶段组成。

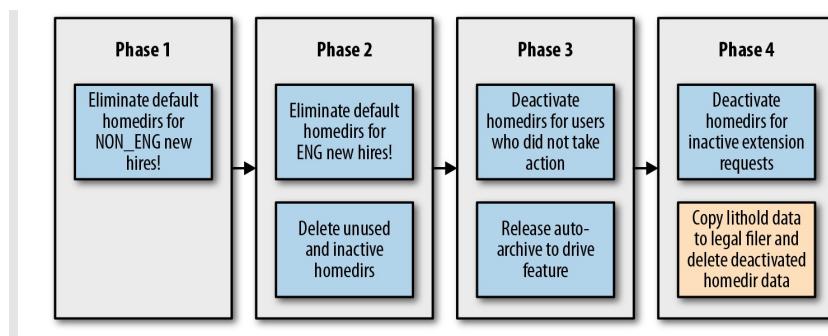


图6-8.Moira项目的四个阶段

淘汰旧系统的第一步是停止或(通常更现实地)减慢或阻止新的采用。从用户那里拿走某些东西比一开始从未提供它要痛苦得多。Moonwalk数据显示，非工程学的Google员工使用共享主目录的次数最少，因此我们的初始阶段针对的是这些用户。随着阶段范围的扩大，我们对替代存储解决方案以及我们的迁移过程和工具的信心也随之增强。项目的每个阶段都有一个相关的设计文档，该文档按照安全性，可伸

缩性，测试和启动等维度检查了提案。我们还特别注意了用户体验，期望和沟通。我们的目标是确保受每个阶段影响的用户了解退役项目的原因以及最简单的归档或迁移数据的方法。

## 关键部件

### Moonwalk

虽然我们掌握了有关用户份额的基本统计信息(例如，共享数量)，但我们需要了解用户的工作流程，以帮助推动围绕弃用的业务决策。我们建立了一个名为Moonwalk的系统来收集和报告此信息。

Moonwalk在BigQuery中存储了有关谁正在访问哪些文件以及何时访问的数据，这使我们能够创建报告并执行临时查询以更好地了解我们的用户。使用BigQuery，我们总结了使用300 TB磁盘空间的25亿个文件的访问模式。该数据归全球60个地理位置的124个NAS设备上400个磁盘卷中的60,000名POSIX用户所有。

### Moira Portal

我们庞大的用户群使基于主故障单的手动流程无法管理主目录退役工作。我们需要使整个过程-尽可能地降低接触程度-对用户进行调查，交流退役项目的原因，并逐步浏览归档其数据或迁移至其他方法。我们的最终要求是:

- 能描述项目的登录页面
- 持续更新的FAQ
- 与当前用户份额相关联的状态和使用情况信息
- 请求，停用，存档，删除，扩展或重新激活共享的选项

我们的业务逻辑变得相当复杂，因为我们不得不考虑许多用户方案。例如，用户可能离开Google，暂时休假或处于诉讼保留状态。图6-9提供了一个示例设计文档状态图，说明了这种复杂性。

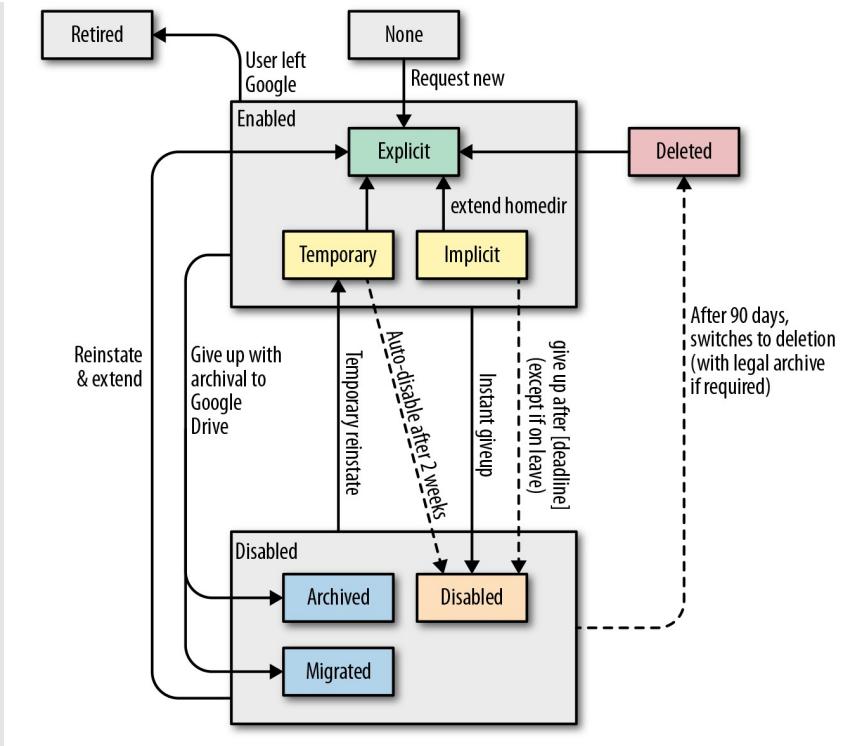


图6-9. 基于用户场景的业务逻辑

支持门户的技术相对简单。它使用Flask框架用Python编写，可以对Bigtable进行读写，并使用了许多后台作业和调度程序来管理其工作。

### 归档和迁移自动化

我们需要大量辅助工具来将门户和配置管理粘合在一起，并与用户进行查询和沟通。我们还需要确保我们为正确的通信确定了正确的用户。错误肯定(需要错误地报告操作)或错误否定(未能通知用户您正在取走某件东西)都是不可接受的，并且此处的错误将意味着丧失信誉和客户服务的形式进行额外的工作。

我们与其他存储系统所有者合作，将功能添加到其路线图中。结果，随着项目的进行，不太成熟的替代方案变得更适合文件使用案例。我们还可以使用和扩展其他团队的工具。例如，作为门户网站自动存档功能的一部分，我们使用了另一个团队内部开发的工具将数据从Google Cloud Storage迁移到Google Drive。

这项工作需要在项目的整个生命周期内进行大量的软件开发。为了响应下一阶段的需求和用户反馈，我们构建并迭代了每个组件-Moonwalk报告管道，门户和自动化，以更好地管理退出和归档份额。我们仅在第三阶段(将近两年)才达到特征完成状态；即使这样，我们仍需要其他工具来处理大约800个用户的“长尾巴”。这种缓慢而缓慢的方法具有明显的好处。它使我们能够：

- 维持精益团队(平均三个CDS团队成员)
- 减少对用户工作流程的干扰
- 限制Techstop(Google内部技术支持组织)的琐事
- 在需要的基础上构建工具，以避免浪费工程精力

与所有工程决策一样，也要权衡取舍：该项目将长期存在，因此，在设计这些解决方案时，团队必须忍受与文件管理器相关的操作繁琐。

该计划于2016年正式完成。在撰写本文时，我们已经将主目录从65,000减少到了50。(当前的Azog项目旨在淘汰这些最后的用户并完全停用文档编纂硬件。)我们的用户体验得到了改善，CDS已淘汰了运营成本高昂的硬件和流程。

## 经验教训

虽然没有人能替代Googlers使用超过14年的文件支持的存储，但我们并不一定需要批发替代。通过有效地将堆栈从通用但受限的文件系统级解决方案上移到多个特定于应用程序的解决方案，我们交易了灵活性以提高可伸缩性，延迟容忍度和安全性。Moira团队必须预测各种用户旅程，并在成熟的各个阶段考虑替代方案。我们必须围绕这些替代方案来管理期望：总体而言，它们可以提供更好的用户体验，但是实现目标并非易事。我们学习了以下有关沿途有效减少琐事的课程。

### 挑战性假设并淘汰昂贵的业务流程

业务需求日新月异，新的解决方案不断涌现，因此值得定期质疑劳动密集型业务流程。正如我们在第101页的“琐事管理策略”中所讨论的那样，拒绝琐事(决定不执行繁琐的任务)通常是消除此问题的最简单方法，即使这种方法并不总是那么快捷或容易。用户分析和业务依据不仅可以减少劳力，还可以为您提供支持。取消文件管理器的主要业务依据归结为Beyond Corp安全模型的好处。因此，虽然Moira是减少CDS团队辛苦工作的好方法，但它强调了退役申报人的许多安全益处，使商业案例更具吸引力。

### 构建自助服务界面

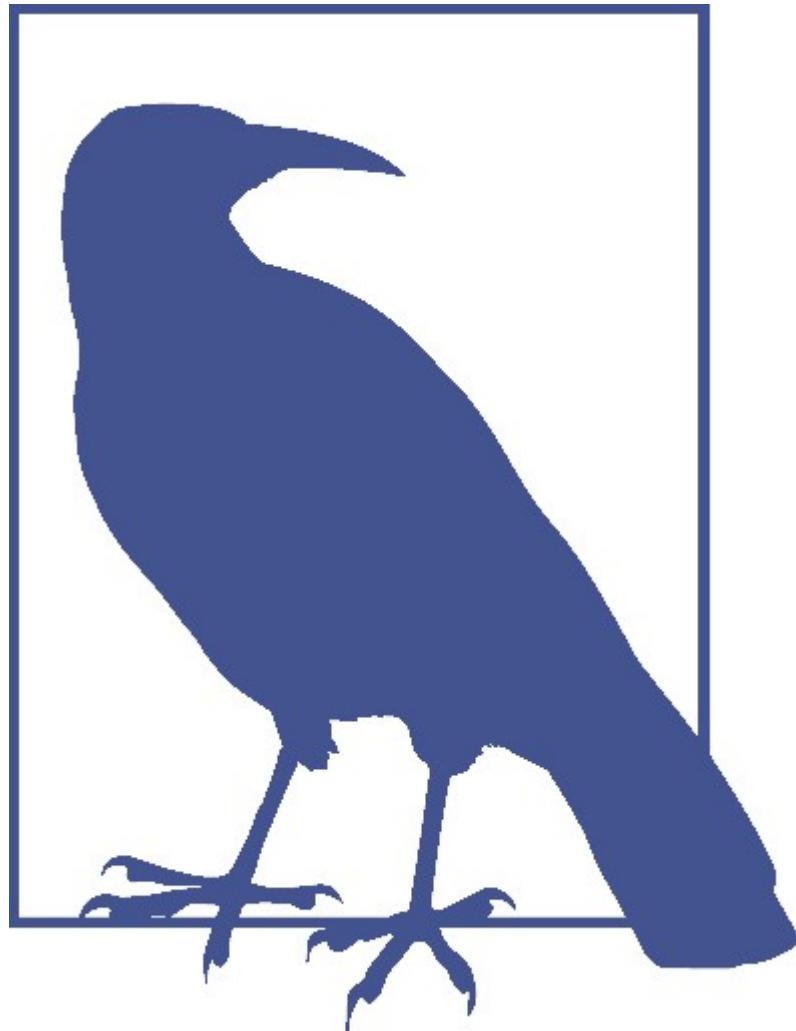
我们为Moira构建了一个自定义门户(这成本相对高)，但是通常有更容易的选择。Google的许多团队都使用版本控制来管理和配置其服务，并以拉取请求的形式处理组织的请求(称为变更列表或CL)。这种方法几乎不需要服务团队的参与，但是却为我们提供了代码检查和持续部署过程的优势，以验证，测试和部署内部服务配置变化。

### 从人工支持的接口开始

Moira团队在某些方面采用了“幕后工程师”方法，将自动化与工程师的手工工作结合在一起。例如，共享请求打开了正在跟踪bug，我们自动化更新这个bug为我们处理过这些请求。系统还分配了最终用户错误，以提醒他们解决其共享问题。故障单可以作为一个快速且肮脏的GUI为自动化服务：它们可以记录工作日志，更新涉众并在自动化出现问题时提供简单的人工后备机制。在我们的案例中，如果用户在迁移方面需要帮助，或者自动化无法处理其请求，则该错误会自动路由到SRE手动处理的队列。

### 融化的雪花

自动化渴望符合标准。Moira的工程师选择重新配置我们的自动化工具，以专门处理份额边缘情况，或者删除/修改不合格份额以符合工具期望。这使我们能够在许多迁移过程中实现零接触自动化。



有趣的事實：在Google，這種改變現實以適應代碼而不是相反的做法被稱為“購買侏儒”。這句話引用了有關Froogle的傳說，這是公司成立之初的購物搜索引擎。

在Froogle生命的早期，一個嚴重的搜尋品質錯誤導致對[跑鞋]的搜尋返回了一個花園侏儒(穿着跑鞋)作為高等級結果。在嘗試修復該錯誤幾次失敗之後，有人注意到該侏儒不是大量生產的物品，而是一個帶有“立即購買”選項的eBay清單。他們購買了侏儒(圖6-10)。



图6-10. 不会消失的花园侏儒

#### 聘请组织机构

寻找吸引新用户以采用更好的(可能希望减少琐事劳动强度)替代方案的方法。出于这种考虑，Moira要求对新的份额或配额请求进行升级，并认识能退役他们的分享的用户。提供有关服务设置，最佳做法以及何时使用服务的良好文档也很重要。Google团队经常采用代码实验室或食谱来教用户如何针对常见用例设置和使用其服务。结果，大多数用户入职不需要拥有该服务的团队的帮助。

## 结论

至少，与运行生产服务相关的工作量随其复杂性和规模呈线性增长。自动化通常是消除琐事的黄金标准，并且可以与许多其他策略结合使用。即使不值得花全力以赴的精力，您也可以通过部分自动化或更改业务流程等策略来减少工程和运维工作量。

本章中描述的消除琐事的模式和方法可以推广到各种其他大规模生产服务。消除琐事，可以节省工程时间来专注于服务的更持久方面，并且随着现代服务体系结构的复杂性和规模的不断增加，团队可以将人工任务保持在最低限度。

重要的是要注意，消除琐事并非总是最好的解决方案。如本章通篇所述，您应考虑与确定，设计和实施琐事相关的流程或自动化解决方案相关的可衡量成本。一旦确定了琐事工作，就必须使用指标，投资回报率(ROI)分析，风险评估和迭代开发来确定何时应减少琐事工作。

琐事通常从小事开始，并且会迅速成长以消耗整个团队。SRE团队必须坚持不懈地消除琐事，因为即使任务看起来艰巨，收益通常也超过成本。我们描述的每个项目都需要各自团队的毅力和奉献精神，有时他们会与怀疑论或机构抵制作斗争，并且总是面临着相互竞争的高度优先事项。我们希望这些故事能够鼓励您识别自己的琐事，对其进行量化，然后努力消除它。即使您今天不能投资大型项目，也可以从一个小的概念证明入手，这可以帮助您的团队改变工作的意愿。

<sup>43</sup>. 这里列出的最主观的特征是否可以自动化。通过自动完成工作来积累经验，您的观点将不断发展。一旦您习惯了"让机器人来完成工作"，那么一个看起来难以解决(或过于冒险)的问题空间将变得可行。 ↵

<sup>44</sup>. 一些工程师不介意长时间工作-并非每个人的劳动容忍阈值都相同。从长远来看，琐事会导致职业停滞，同时促进职业倦怠引起的离职。一定程度的琐事是不可避免的，但我们建议在可行的情况下减少辛勤工作，以维护团队，服务和个人的健康。 ↵

<sup>45</sup>. 换句话说，如果一项服务及其九个依赖项各自具有99.99%的可用性，则该服务的总可用性将为 $0.9999^{10} = 99.9\%$ 。有关依赖关系如何影响服务可用性的更多信息，请参阅"服务可用性计算"。 ↵

<sup>46</sup>. 当然，您将无法通过自助服务来处理一些一次性的情况("您想让一个VM具有多少个RAM?")，但旨在涵盖大多数用例。将80--90%的请求转移到自助服务上仍然大大减少了工作量！ ↵

<sup>47</sup>. 简而言之，从单个专用设备向具有公共接口的设备群过渡。请参阅"案例研究1: 有关对此类比的详细说明，请参见第107页的"通过自动化减少数据中心中的工作量"。 ↵

<sup>48</sup>. 线卡是一个模块化组件，通常为网络提供多个接口。它与其他线卡和组件一起位于机箱的底板中。模块化网络交换机由一个机箱组成，该机箱包括一个背板，电源输入模块，控制卡模块以及一个或多个线卡。每个线卡都支持到计算机或其他线卡(在其他交换机中的)的网络连接。与USB网络接口适配器一样，您可以在不关闭整个交换机电源的情况下更换任何线卡，前提是线卡已"排空"，这意味着已告知其他接口停止向其发送流量。 ↵

<sup>49</sup>. 误码率测试: 在恢复服务之前检查链路是否不正常。 ↵

<sup>50</sup>. 检查连接错误的端口。 ↵

<sup>51</sup>. Piper是Google的内部版本控制系统。有关更多信息，请参阅Rachel Potvin和Josh Levenberg, "Google为什么要在单个存储库中存储数十亿行代码"，ACM通讯 59，第1页。 7(2016): 78--87，<http://bit.ly/2J4jgMi>。 ↵

<sup>52</sup>. Google还具有可扩展的自助式Git托管，用于托管Piper中不存在的代码。 ↵

<sup>53</sup>. Beyond Corp是一项计划，旨在从传统的基于边界的安全性模型转变为基于密码身份的模型。当Google笔记本电脑连接到内部Google服务时，该服务通过识别笔记本电脑的加密证书，用户拥有的第二个因素(例如USB安全密钥)，客户端设备的配置/状态和用户的凭证。 ↵

<sup>54</sup>. x20是一个内部全局共享的，高度可用的文件系统，具有类似POSIX的文件系统语义。[←](#)

## 第7章

### 简单化

**By John Lunney, Robert van Gent, and Scott Ritchie with Diane Bates and Niall Richard Murphy**

一个有效的复杂系统总是从一个简单的有效系统演化而来的。

--加尔定律

简单化是SRE的重要目标，因为它与可靠性紧密相关:简单的软件中断的次数较少，并且在中断时更容易更快速地进行修复。简单的系统更易于理解，易于维护和测试。

对于SRE，简单化是端到端的目标:应该超越代码本身，扩展到系统体系结构以及用于管理软件生命周期的工具和过程。本章探讨了一些示例，这些示例演示了SRE如何衡量，思考和鼓励简单性。

### 测量复杂度

衡量软件系统的复杂性并不是一门绝对的科学。测量软件代码复杂度的方法有很多种，其中大多数是相当客观的。<sup>55</sup>也许最著名，使用最广泛的标准是[循环代码复杂度](#)，它通过一组特定的语句来测量不同代码路径的数量。例如，没有循环或条件的代码块的循环复杂度数(CCN)为1。实际上，软件社区非常擅长于测量代码的复杂性，并且有许多用于集成开发环境(包括Visual Studio，Eclipse和IntelliJ)的测量工具。我们不太善于理解所测得的复杂度是必要的还是偶然的，一种方法的复杂性如何影响更大的系统，以及哪种方法最适合重构。

另一方面，用于度量系统复杂性的正式方法很少见。<sup>56</sup>您可能会想尝试一种CCN类型的方法来计算不同实体(例如微服务)的数量以及它们之间可能的通信路径。但是，对于大多数规模较大的系统，该数字可能会非常快地增长到无可救药的地步。

有关系统级复杂性的一些更实际的代表包括:

**训练时间**

新的团队成员需要多长时间才能上班？文档不足或缺失可能是主观复杂性的重要来源。

**解释时间**

向新的团队成员解释服务的全面高级视图(例如，在白板上绘制系统架构并解释每个组件的功能和依赖性)需要花费多长时间？

**管理多元化**

有多少种方法可以在系统的不同部分中配置相似的设置？配置存储在集中的位置还是多个位置？

#### 部署配置的多样性

生产中部署了多少个独特的配置(包括二进制文件，二进制版本，标志和环境)？

#### 年龄

系统多大了？[Hyrum定律](#)指出，随着时间的流逝，API的用户取决于其实现的各个方面，从而导致脆弱和不可预测的行为。

尽管有时测量复杂性是值得的，但很难。但是，似乎没有对以下观点的严重反对：

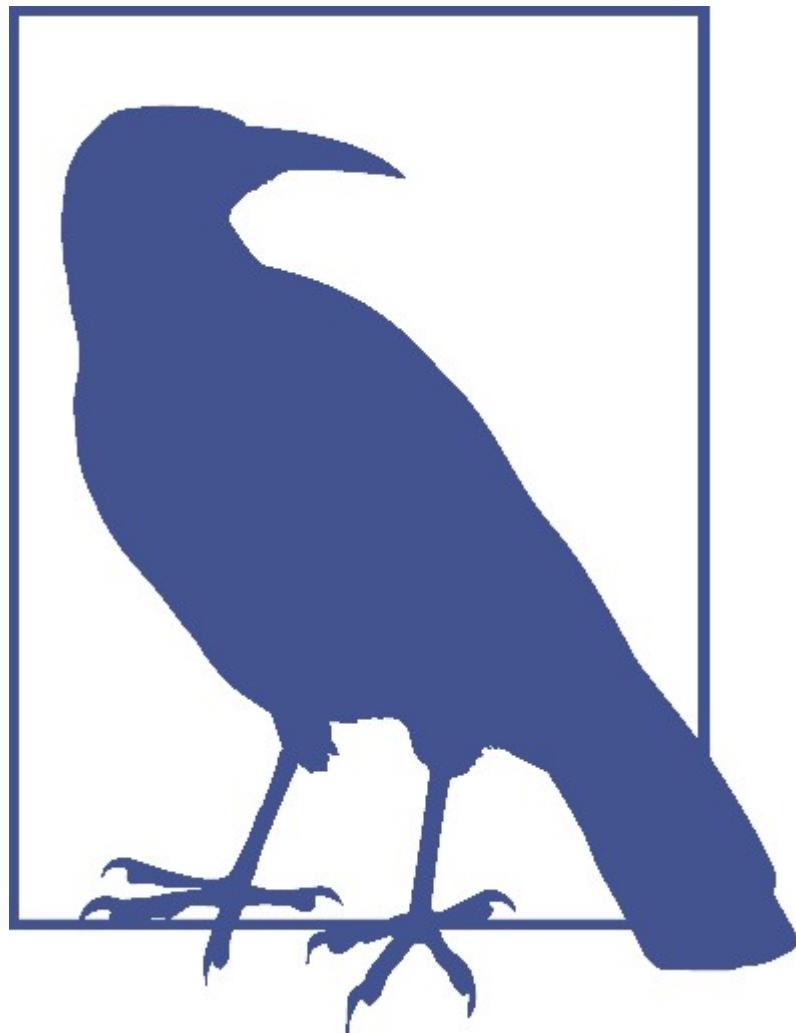
- 通常，除非做出与之抗衡的努力，否则现有软件系统的复杂性将会增加。
- 付出努力是值得的。

## 简单化是端到端的，而SRE对此非常有用

通常，生产系统的设计不是整体的。相反，它们有机地生长。随着团队增加新功能并推出新产品，他们会随着时间的推移积累组件和连接。尽管单个更改可能相对简单，但是每个更改都会影响其周围的组件。因此，总体复杂性很快就会变得不堪重负。例如，在一个组件中添加重试可能会使数据库超载并破坏整个系统的稳定性，或者使推理基于系统的给定查询的路径变得更加困难。

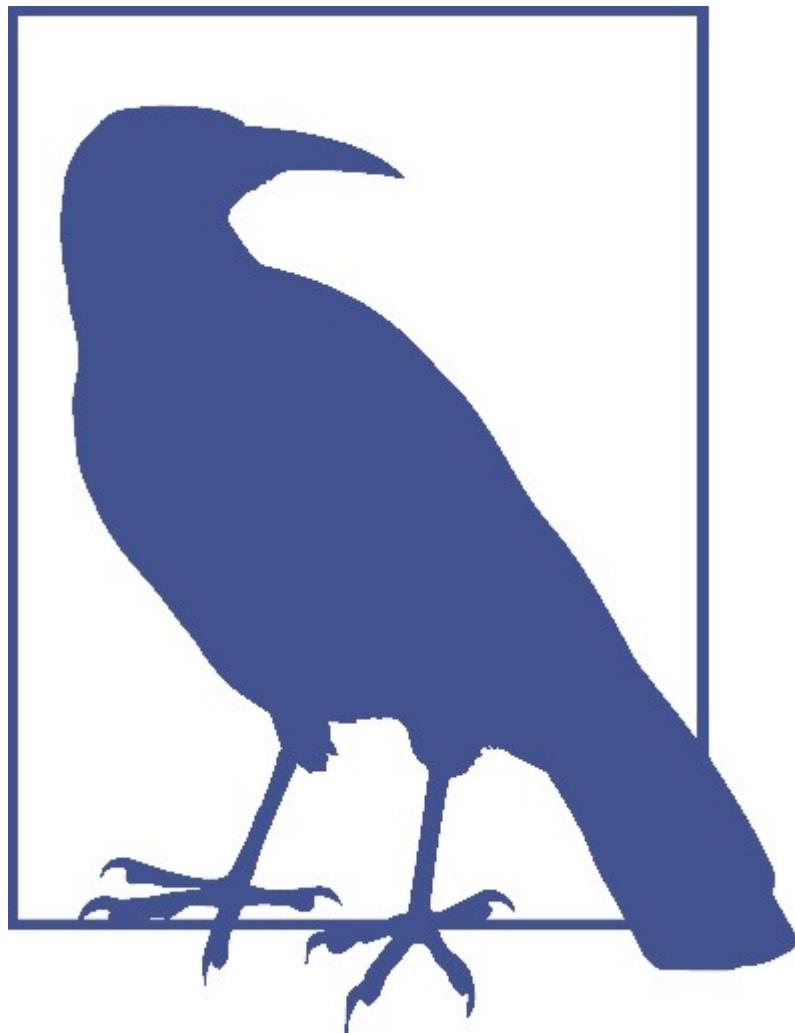
通常，复杂性的成本不会直接影响引入该复杂性的个人，团队或角色-从经济角度来讲，复杂性是外在的。相反，复杂性会影响那些继续在其中及其周围工作的人。因此，拥有端到端系统简化的拥护者很重要。

SRE很自然地适合担任此职务，因为他们的工作要求他们将系统视为一个整体。<sup>57</sup>除了支持自己的服务外，SRE还必须洞悉与其服务交互的系统。Google的产品开发团队通常对生产范围的问题一无所知，因此他们认为向SRE咨询有关系统设计和运维的建议很有价值。



**读者行动:** 在工程师初次上线之前，鼓励他们绘制(并重新绘制)系统图。在您的文档中保留一套规范的图表:它们对新工程师很有用，并可以帮助更有经验的工程师跟上变化。

根据我们的经验，产品开发人员通常最终只能在狭窄的子系统或组件中工作。结果，他们没有整个系统的思维模型，他们的团队也没有创建系统级的架构图。这些图很有用，因为它们可以帮助团队成员可视化系统交互并使用通用词汇表述问题。通常，我们发现该服务的SRE团队绘制了系统级体系结构图。



**读者行动:** 确保SRE审核所有主要设计文档，并确保团队文档显示新设计如何影响系统体系结构。如果设计增加了复杂性，那么SRE可能会提出简化系统的替代方案。

## 案例研究1: 端到端API简化

### 背景

上一章的作者曾在一家初创公司工作，该公司在其核心库中使用键/值袋数据结构。RPC(远程过程调用)拿了一个袋子并归还了一个袋子；实际参数作为键/值对存储在包装袋中。核心库支持对包的常规操作，例如序列化，加密和日志记录。所有核心库和API都非常简单和灵活-成功，对吧？

遗憾的是，没有库的客户最终为核心API的抽象性质付出了代价。对于每个服务，键和值(以及值类型)的集合需要仔细记录，但通常不需要。另外，随着时间的增加，删除或更改参数，保持后向/向前兼容性变得困难。

### 经验教训

[Google's Protocol Buffers](#)或[Apache Thrift](#)之类的结构化数据类型似乎比其抽象的通用替代品更为复杂，但是由于它们可以强制进行前期设计决策和文档化，因此它们可提供更简单的端到端解决方案。

## 案例研究2: 项目生命周期的复杂性

当您查看现有系统这个混乱的意大利面条时，可能会很想用一个新的，干净，简单的系统来替换它，以解决相同的问题。不幸的是，在维持现有系统的同时创建新系统的成本可能不值得。

### 背景

[Borg](#)是Google的内部容器管理系统。它运行着大量的Linux容器，并具有多种使用模式:批处理与生产，管道与服务器等。多年来，Borg及其周围的生态系统随着硬件的变化，功能的添加以及规模的增长而增长。

[Omega](#)旨在成为Borg的原则更清晰的版本，支持相同的用例。但是，从Borg到Omega的计划中的转换存在一些严重的问题:

- 随着Omega的发展，Borg不断发展，因此Omega一直在追求一个不断变化的目标。
- 对改善Borg的难度的早期估计被证明过于悲观，而对Omega的期望被证明过于乐观(实际上，草并不总是较绿)。
- 我们不完全理解从Borg迁移到Omega会有多么困难。跨数千个服务和许多SRE团队的数百万行配置代码意味着，迁移在工程和日历时间方面将是极其昂贵的。在可能需要数年的迁移期间，我们必须支持和维护这两个系统。

### 我们决定要做的

[最终](#)，我们将在设计Omega时出现的一些想法反馈回了Borg。我们还使用了Omega的许多概念来启动[Kubernetes](#)，一个开源容器管理系统。

### 经验教训

在考虑重写时，请考虑整个项目生命周期，包括朝着不断变化的目标进行开发，完整的迁移计划以及在迁移时间段内可能产生的额外费用。具有很多广泛用户的API很难迁移。不要将预期结果与当前系统进行比较。相反，如果将相同的精力投入到改进它上，则将预期结果与当前系统的外观进行比较。有时重写是最好的方法，但是请确保已权衡成本和收益，并且不要低估成本。

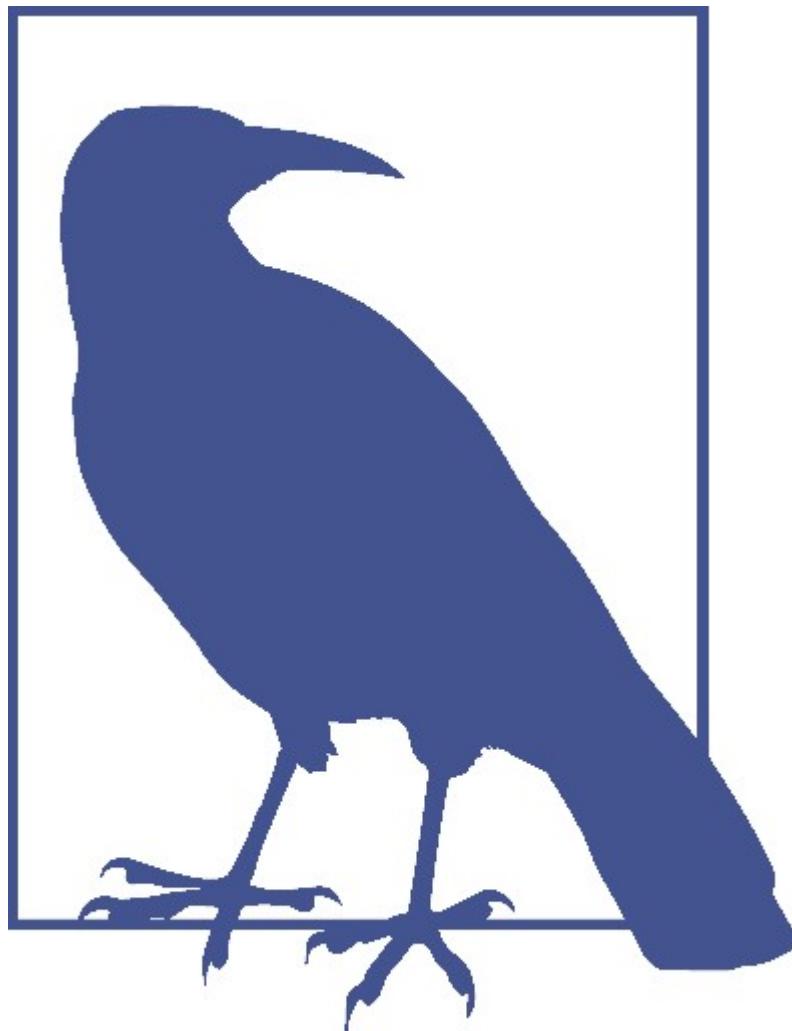
## 保持简单

大多数简化工作包括从系统中删除元素。有时，简化是直接的(例如，消除对从远程系统获取的未使用数据的依赖)。其他时候，简化需要重新设计。例如，系统的两个部分可能需要访问相同的远程数据。而不是两次获取数据，一个更简单的系统可能会一次获取数据并转发结果。

无论做什么工作，领导层都必须确保简化工作得到庆祝并明确地确定优先次序。简化就是效率-代替节省计算或网络资源，它可以节省工程时间和认知负担。就像对待有用的功能启动一样对待成功的简化项目，并平等地衡量和庆祝代码的添加和删除。<sup>58</sup>例如，Google的内网为删除大量代码的工程师显示“僵尸代码杀手”徽章。

简化是一个功能。您需要确定简化项目的优先级并为其配备人员，并为SRE留出时间来处理它们。如果产品开发人员和SRE认为简化项目对他们的职业不利，那么他们将不会进行这些项目。考虑将简单化作为特别复杂的系统或超负荷团队的明确目

标。创建一个单独的时间来完成这项工作。例如，为“简单化”项目预留工程项目时间的10%。<sup>59</sup>



**读者行动:** 让工程师集思广益，讨论系统中已知的复杂性，并讨论降低这些复杂性的想法。

随着系统复杂性的增加，有一种诱惑来分裂SRE团队，将每个新团队集中在系统的较小部分上。尽管有时这是必要的，但新团队的规模缩小可能会降低他们推动更大的简化项目的动力或能力。考虑指定一个SRE轮值小组，他们维护整个堆栈的工作知识(可能深度较小)，但可以推动整个堆栈的一致性和简化性。

如前所述，对系统进行图表绘制可以帮助您识别更深层的设计问题，这些问题会妨碍您理解系统并预测其行为。例如，在绘制系统示意图时，您可能会寻找以下内容：

#### 放大

当调用返回错误或超时并在多个级别上重试时，它将导致RPC总数增加。

#### 循环依赖

当组件(通常是间接地)依赖于自身时，系统完整性可能会受到严重损害-尤其是整个系统的冷启动可能变得不可能。

## 案例研究3: 简化展示广告Spiderweb

## 背景

Google的展示广告业务拥有许多相关产品，其中包括一些源自收购的产品（DoubleClick，AdMob，Invite Media等）。这些产品必须经过修改才能与Google基础架构和现有产品一起使用。例如，我们希望使用DFP广告管理系统的网站能够展示Google AdSense选择的广告；同样，我们希望使用DoubleClick Bid Manager的投标人可以访问在Google Ad Exchange上进行的实时拍卖。

这些独立开发的产品形成了一个很难推理的互连后端系统。观察流量通过组件时发生的情况很困难，并且为每块资源提供适当的容量是不方便且不精确的。在某一时刻，我们添加了测试以确保我们删除了查询流中的所有无限循环。

## 我们决定要做什么

投放SRE的广告是标准化的自然推动力：尽管每个组件都有一个特定的开发人员团队，但SRE在整体堆栈中处于待命状态。我们的首要任务之一是起草统一性标准，并与开发团队合作逐步采用它们。这些标准为：

- 建立了复制大型数据集的单一方法
- 建立了执行外部数据查找的单一方法
- 提供了用于监控、配置和配置的通用模板

在此倡议之前，单独的程序为每种产品提供了前端和拍卖功能。如图7-1所示，当一个广告请求可能同时到达两个定位系统时，我们重新编写了该请求以满足第二个系统的期望。这需要额外的代码和处理，并且还开辟了不良循环的可能性。

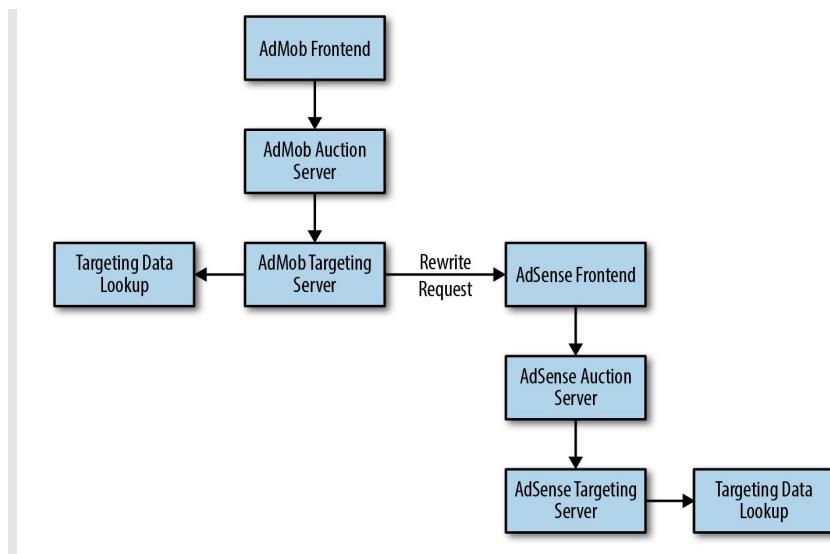


图7-1.以前，广告请求可能会同时在AdMob和AdSense系统上运行

为了简化系统，我们在满足所有用例的通用程序中添加了逻辑，并添加了标记来保护程序。随着时间的流逝，我们删除了这些标记并将功能合并到更少的服务器后端中。

服务器统一后，拍卖服务器可以直接与两个目标服务器通信。如图7-2所示，当多个目标服务器需要数据查找时，在统一拍卖服务器中查找只需进行一次。

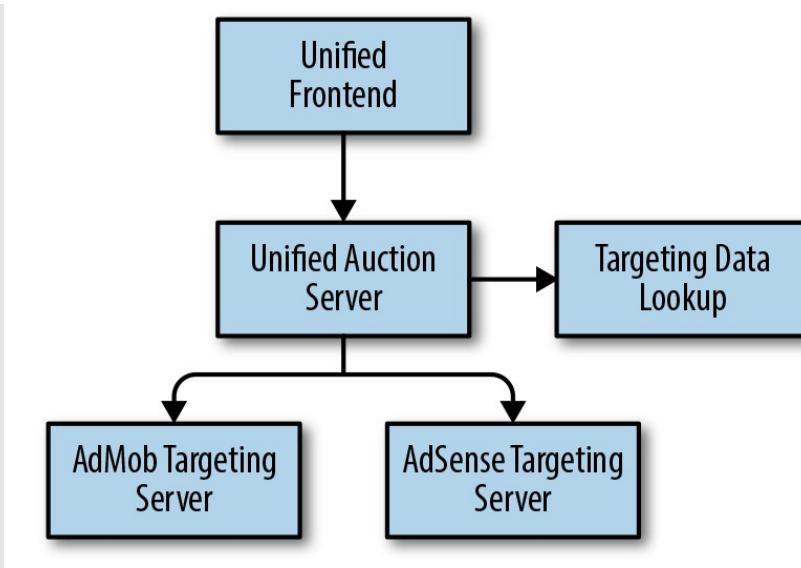


图7-2.统一拍卖服务器现在仅执行一次数据查找

### 经验教训

最好将已经运行的系统逐步集成到您自己的基础架构中。

就像在单个程序中存在非常相似的功能表示“代码臭气”表明更深的设计问题一样，在单个请求中的冗余查找也表示“系统臭气”。

当您通过SRE和开发人员的支持来创建定义明确的标准时，您可以提供清晰的蓝图，以消除管理人员更可能认可和奖励的复杂性。

## 案例研究4: 在共享平台上运行数百个微服务

由迈克·柯蒂斯(Mike Curtis)撰写

### 背景

在过去的15年中，Google开发了多个成功的产品垂直市场(仅举几个例子，包括Search，Ads和Gmail)，并产生了稳定的新系统和重构系统。这些系统中的许多系统都有专门的SRE团队和相应的特定领域生产堆栈，其中包括定制开发工作流程，持续集成和持续交付(CI/CD)软件周期以及监控。这些独特的生产堆栈会在维护，开发成本和独立的SRE投入方面产生大量的开销。它们也使在团队之间移动服务(或工程师！)或添加新服务变得更加困难。

### 我们决定要做什么

在社交网络领域中的一组SRE团队致力于将其服务的生产堆栈聚合到一个由一组SRE管理的单一托管微服务平台中。共享平台符合最佳实践，并且捆绑并自动配置了许多以前未充分使用的功能，这些功能可提高可靠性并促进调试。无论SRE参与程度如何，都需要使用SRE团队范围内的新服务来使用公共平台，并且旧服务必须迁移到新平台或被淘汰。

在社交网络领域获得成功后，共享平台已被Google的其他SRE和非SRE团队采用。

### 设计

我们使用了微服务，因此我们可以快速更新和部署功能-单个整体服务的更改缓慢。服务是“被管理(managed)”的，而不是“被托管(hosted)”的：不是让单个团队摆脱控制和责任，而是让他们自己有效地管理他们的服务。我们提供了服务团队可以用来发布，监控以及更多功能的工作流工具。

我们提供的工具包括一个UI，API和一个命令行界面，SRE和开发人员可以使用它们与他们的堆栈进行交互。这些工具使开发人员的体验变得统一，即使涉及许多基础系统也是如此。

## 结果

该平台的高质量和功能集带来了意想不到的好处：开发人员团队可以在无需任何SRE参与的情况下运行数百种服务。

通用平台还改变了SRE与开发人员的关系。因此，“分层”SRE参与在Google中变得很普遍。分层参与使一系列SRE被包含其中，遍布从轻度咨询和设计审查到深度参与(即，SRE共同承担On-Call职责)。

## 经验教训

从稀疏或定义不明确的标准过渡到高度标准化的平台是一项长期投资。每个步骤都可能是渐进式的，但最终，这些步骤会减少开销，并使大规模运行服务成为可能。

开发人员在这种过渡中看到价值非常重要。目标是在开发的每个阶段都释放出提高生产率的胜利。不要试图说服人们进行巨大的重构，而这种重构只有在最后才有回报。

# 案例研究5：pDNS不再依赖于自己

## 背景

当Google生产中的客户想要查找服务的IP地址时，通常会使用称为Svelte的查找服务。过去，为了查找Svelte的IP地址，客户端使用了名为pDNS(生产DNS)的Google命名服务。通过负载平衡器访问pDNS服务，该负载平衡器使用Svelte查找实际pDNS服务器的IP地址。

## 问题陈述

pDNS本身具有传递性依赖关系，在某个时候无意中引入了它，直到后来才将其确定为可靠性问题。通常不会出现问题，因为复制了pDNS服务，并且破坏了数据

依赖循环之外的功能始终可以在Google产品中的某个位置使用。但是，冷启动是不可能的。用一个SRE的话来说，“我们就像是洞穴居民，他们只能用最后一次篝火点燃的火炬来点燃油。”

## 我们决定要做什么

我们修改了Google生产中的低级组件，以维护所有Google生产计算机在本地存储中附近Svelte服务器的当前IP地址列表。除了打破前面描述的循环依赖关系之外，此更改还消除了大多数其他Google服务对pDNS的隐式依赖关系。

为了避免类似的问题，我们还介绍了一种将允许与pDNS通信的服务集列入白名单的方法，并逐步减少了该服务集。结果，生产中的每个服务查找现在都具有通过系统的更简单和更可靠的路径。

## 经验教训

请注意您的服务的依赖性-使用明确的白名单来防止额外意外。另外，请注意循环依赖。

## 结论

对于SRE而言，简单化是一个自然的目标，因为简单的系统倾向于可靠且易于运行。从数量上衡量分布式系统的简单化(或它的逆向性，复杂性)不容易，但是有一些合理的代表，值得选择一些并进行改进。

由于SRE对系统具有端到端的了解，因此它们非常适合识别，预防和修复复杂性源，无论它们是在软件设计，系统体系结构，配置，部署过程还是其他位置发现的。SRE应该尽早参与设计讨论，以提供其对替代方案的成本和收益的独特见解，并特别注重简单化。SRE还可主动制定标准以使生产同质化。

作为SRE，追求简单是您工作描述的重要组成部分。我们强烈建议SRE领导授权SRE团队推动简单化，并明确奖励这些努力。系统在发展过程中不可避免地会趋于复杂，因此为简单起见，需要不断的关注和投入-但它非常值得追求。

<sup>55</sup>. 如果您有兴趣了解更多信息，请阅读此软件复杂性趋势的最新[review](#)，或阅读Horst Zuse，*软件复杂性: 措施和方法*(柏林: Walter de Gruyter，1991年)。[←](#)

<sup>56</sup>. 尽管有一些示例，例如-"[AWS系统的自动化形式推理](#)"。[←](#)

<sup>57</sup>. 结果，SRE对于希望攻击复杂性作为产品技术债务代理的产品开发主管来说可能是一项有用的投资，但发现很难在现有团队的范围内证明这项工作的合理性。[←](#)

<sup>58</sup>. 正如[Dijkstra](#)所说，"如果我们希望对代码行进行计数，则不应将它们视为"产生的行"，而应视为"花费的行"。"[←](#)

<sup>59</sup>. 为简单性项目保留一部分时间(例如10%)并不意味着团队就允许其他90%的团队引入复杂性开了绿灯。这只是意味着您将精力放在简化的特定目标上。[←](#)

## PART II 实践

在第一部分所涵盖的SRE原则的坚实基础上，第二部分深入探讨了如何开展Google认为对于大规模运维至关重要的SRE相关活动。

其中一些主题(例如数据处理管道和管理负载)并不适用于所有组织。其他主题，例如通过配置和金丝雀发布安全地处理变更，On-Call的实践以及出现问题时的处理方式，对于任何SRE团队都包含了宝贵的经验教训。

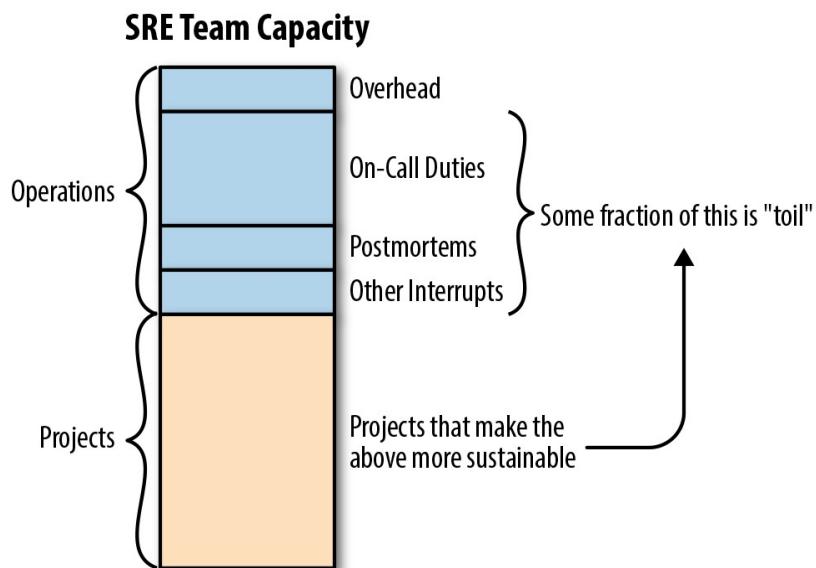
本部分还介绍了一项重要的SRE技能-非抽象大型系统设计(NALSD)，并提供了有关如何实践此设计过程的详细示例。

当我们从SRE基础转向实践时，我们想提供更多有关运维职责与项目工作之间的关系的背景信息，以及从战略上完成这两项工作所需的工程技术。

### 定义运维工作(相对于项目工作和开销)

在我们从基础转到实践之前，我们想介绍一下运维和项目工作之间的区别，以及这两种类型的工作如何相互影响。该主题是SRE社区中哲学辩论的一个领域，因此本插曲介绍了我们如何在本书的上下文中定义两种类型的工作。

SRE实践将软件工程解决方案应用于运维问题。由于我们的SRE团队负责我们支持的系统的日常运行，因此我们的工程工作通常专注于可能在其他地方进行操作的任务:我们使发布流程自动化，而不是手动执行；我们实施分片，以使我们的服务更加可靠，并减少了对人类注意力的要求；我们利用算法方法进行容量规划，因此工程师不必执行容易出错的手动计算。尽管工程和运维工作确实可以相互告知，但我们可以将任何给定的SRE团队执行的工作概念化为两个单独的类别，如图II-1所示。多年来，我们一直在努力使每个工作桶的效率和可扩展性最大化。



图II-1。SRE工作的两个类别

我们可以将运维工作分为四大类:

- 值班工作
- 客户请求(最常见的是故障单)
- 事件响应
- 事后总结

在我们的第一本书中([第11章](#); [第14章](#); [第15章](#))和这本书([第6章](#) ; 第8章 ; 第9章 ; 第10章), 每个类别都具有自己的详细处理方法。在这里，我们展示了这四个方面是如何相互联系的，以及将这些类型的工作视为紧密相关的重要性。

运维工作中不包括哪些类型的工作？如图II-1所示，项目工作是SRE工作的另一个主要任务。如果团队的中断工作得到良好的管理，他们就有时间进行长期的工程工作，以实现稳定性，可靠性和可用性目标。这可能包括旨在提高服务可靠性的软件工程项目，也可能包括诸如将新功能安全地推出到全球复制的服务之类的系统工程项目。

如图II-1所示，"日常开销"是在公司必需的行政管理工作:会议，培训，回复电子邮件，跟踪您的绩效，填写文书工作等等。日常开销对于手头的讨论并不立即重要，但是所有团队成员都花时间在上面。

您可能会注意到，我们没有专门将文档工作作为单独的活动。这是因为我们认为，健康的文档记录程序已融入您的所有工作中。您无需考虑将代码，剧本和服务功能文档化-甚至无需确保故障单和bug包含其应有的所有信息-与项目或操作任务无关。这只是这些任务的另一个方面。

在Google，我们指定SRE应该将至少50%的时间用于项目工作；少做任何事情都会导致不可持续的工程化和过度消耗，疲惫不堪的团队。尽管每个团队和组织都需要找到自己的健康平衡，但我们发现，大约三分之一的时间花在了运维任务上，而三分之二的时间花在了项目工作上是正确的(这个比例也说明了轮流值班的规模，您的工程师只有三分之一的时间在On-Call中)。

我们鼓励团队进行定期审查，以跟踪您是否在工作类型之间取得适当的平衡。在Google，我们会定期进行卓越生产(ProdEx)审核，这些审核使SRE的高级领导者可以使用明确定义的规则来查看每个SRE团队的状态。您需要根据自己的限制和组织成熟度来确定适当的时间间隔和标题，但是这里的关键是生成可以随时间跟踪的有关团队健康的指标。

当找到理想的平衡点时，请记住一个警告:一个团队在运维任务上花费的时间太少，可能会导致运维不足。在这种情况下，工程师可能会开始忘记他们所负责的服务的关键方面。您可以承担更多的风险和更快的行动来应对运维不足，例如，缩短发布周期，为每个发布推送更多功能或执行更多灾难恢复测试。如果您的团队永远负载不足，请考虑加入相关服务或将不再需要SRE支持的服务交还给开发团队(有关团队规模的更多讨论，请参见第8章)。

### 运维工作与项目工作之间的关系

尽管它们是不同的工作类别，但运维和项目工作并非完全分开。实际上，前者中提出的问题应纳入后者:SRE项目工作应该是使系统更加高效，可扩展和可靠，这意味着减少运维负荷和琐事工作量。如图II-1所示，在运维负荷源头与项目工作之间应该存在一个连续的反馈回路，以系统地提高产量。这项较长期的工作可能涉及迁移到更强大的存储系统，重新设计框架以减少脆弱性或维护负担，或解决系统中断和事件的源头。这些举措是通过项目开发和实施的，这些项目被定义为实现特定目标或可交付成果的临时性工作(具有明确的起点和终点)。

## 第8章

### 值班

由Ollie Cook，Sara Smollett，Andrea Spadaccini，

Cara Donnelly，Jian Ma和Garrett Plasky(Evernote)与Stephen Thorne和Jessie Yang撰写

值班意味着可以在规定的时间内有空，并准备在紧急时间内响应生产事件。经常需要站点可靠性工程师(SRE)参与轮班。在值班期间，SRE可根据需要诊断，缓解，修复或升级事件。此外，SRE定期负责非紧急生产职责。

在Google，值班是SRE的定义特征之一。SRE团队可以减轻事故，修复生产问题并自动化操作任务。由于我们的大多数SRE团队尚未将其所有操作任务完全自动化，因此故障升级需要人员的直接联系，即值班工程师。根据所支持系统的关键程度或系统所处的开发状态，可能并非所有SRE团队都需要值班。根据我们的经验，大多数SRE团队都会安排值班时间。

值班是一个庞大而复杂的主题，受许多限制和反复试验的限制。[我们的第一本书(*Site Reliability Engineering*)，“On-Call轮值”的[第11章](#)已经探讨了这个主题。本章介绍了有关该章的具体反馈和问题。其中包括：

- "我们不是Google；我们要小得多。我们轮换的人不多，而且我们在不同时区的没有站点。你在第一本书中描述的与我无关。"
- "我们将开发人员和DevOps混合在一起，可以随时轮值。最好的组织方式是什么？分担责任？"
- "我们的值班工程师通常在24小时轮班中被寻呼一百次。很多页面都被忽略了，所以真正的问题却被埋没了。我们应该从哪里开始？"
- "我们的电话轮换率很高。您如何解决团队内部的知识鸿沟？"
- "我们希望将DevOps团队重组为SRE。<sup>60</sup> SRE值班，DevOps值班和开发人员值班之间有什么区别？请具体说明，因为DevOps团队对此非常关注。"

我们针对这些情况提供实用建议。Google是一家拥有成熟的SRE组织的大公司，但是我们多年来所学到的许多知识都可以应用于任何公司或组织，无论其规模或成熟度如何。Google跨各种规模的服务提供了数百个on-call轮值，以及从简单到复杂的各种值班设置。值班不仅是SRE功能：许多开发人员团队都直接为他们的服务进行值班。每个值班设置都可以满足特定服务的需求。

本章介绍了Google内部和Google外部的值班设置。尽管您的设置和情况可能与我们的特定示例有所不同，但是我们涵盖的基本概念是广泛适用的。

然后，我们深入研究呼叫负载的结构，解释导致呼叫负载的原因。我们建议优化值班设置并最大程度地减少负载的策略。

最后，我们分享了Google内部实践的两个示例：值班灵活性和值班团队动态。这些实践表明，无论从数学上讲值班设置的听起来如何，您都不能仅依靠值班设置作为后勤保障。激励和人性起着重要作用，也应予以考虑。

## 第一本SRE图书"On-Call轮值"章节的回顾

[*Site Reliability Engineering*]中的"[Be-on-Call](#)"中，解释了Google进行值班轮换的原理。本节讨论该章的要点。

在Google，值班的总体目标是提供关键服务的覆盖范围，同时确保我们永远不会以牺牲值班工程师的健康为代价来实现可靠性。结果就是，SRE团队努力实现“平衡”。SRE工作应该是职责的健康组合：值班和项目工作。指定SRE在项目工作上花费至少50%的时间，意味着团队有时间解决战略上解决生产中发现的任何问题所需的项目。团队人员必须足够确保项目时间。

我们的目标是每个值班班次最多发生两次事故，[61](#)以确保有足够的时间进行跟进。如果值班负载过高，则必须采取纠正措施。（我们将在本章后面探讨值班的负载。）

心理安全 [62](#)对于有效的通话轮换至关重要。由于随时待命可能会令人生畏且压力很大，因此应通过一系列程序和逐步升级途径为值班工程师提供全面支持，以使他们的生活更轻松。

值班通常意味着一定数量的非工作时间。我们认为这项工作应得到补偿。虽然不同的公司可能选择以不同的方式处理此问题，但Google提供了休假或现金补偿，但不超过总工资的一定比例。补偿方案提供了参与值班的动机，并确保工程师出于经济原因不会进行过多的值班班次。

## 在Google内部和外部进行值班设置的示例

### 谷歌

本部分描述了Google和Evernote(加利福尼亚州一家公司，开发跨平台应用程序，可帮助个人和团队创建，组合和共享信息)中值班设置的真实示例。对于每家公司，我们都会探讨值班设置，通用值班哲学和值班惯例背后的原因。

### 谷歌：组建新团队

#### 初始方案

几年前，Google Mountain View的SRE萨拉(Sara)成立了一个新的SRE团队，该团队需要在三个月内开始值班。从这个角度来看，Google的大多数SRE团队都不希望新员工在三到九个月之前就可以随时待命。新的Google Mountain View SRE团队将支持三项Google Apps服务，以前由华盛顿州柯克兰的SRE团队提供支持(距离Google Mountain View只有两个小时的飞行时间)。Kirkland团队在伦敦拥有一个姐妹SRE团队，该团队将继续与新的Mountain View SRE团队以及分布式产品开发团队一起为这些服务提供支持。[63](#)

新的Mountain View SRE团队迅速聚集在一起，聚集了七个人：

- SRE技术负责人Sara
- Mike，来自另一个SRE团队的经验丰富的SRE
- 来自SRE的新产品开发团队的转移
- 四名新员工("Nooglers")

即使团队已经成熟，为新服务值班总是很困难的，新的Mountain View SRE团队是一个相对较小的团队。尽管如此，新团队仍能够在不牺牲服务质量或项目速度的情况下启动服务。他们立即对服务进行了改进，包括将机器成本降低了40%，并通过金丝雀和其他安全检查使发布发布完全自动化。新团队还继续提供可靠的服务，目标是99.98%的可用性，或者每个季度大约26分钟的停机时间。

新的SRE团队如何引导自己完成这么多任务？通过入门项目，指导和培训。

### 培训路线图

尽管新的SRE团队对他们的服务并不了解，但是Sara和Mike熟悉Google的生产环境和SRE。当四个Nooglers完成公司定位后，Sara和Mike编制了一份清单，列出了两打重点领域，供人们在值班前进行练习，例如：

- 管理生产工作
- 了解调试信息
- 从集群中“引流”流量
- 回滚不良的软件推送
- 阻止或限制不必要的流量
- 提高服务能力
- 使用监控系统(用于警报和仪表板)
- 描述服务的体系结构，各种组件和依赖性

Nooglers通过研究现有的文档和code labs(指导，动手编码教程)，自己找到了一些信息，并通过从事其入门项目来获得对相关主题的理解。当团队成员了解了与Nooglers入门项目有关的特定主题时，该人主持了一个简短的即席会议，与团队的其他成员分享该信息。萨拉和迈克介绍了其余主题。该团队还举行了实验会议，以执行常见的调试和缓解任务，以帮助每个人建立肌肉记忆并增强对其能力的信心。

除了清单之外，新的SRE团队还进行了一系列“深潜”以深入研究其服务。该团队浏览了监控控制台，确定了正在运行的作业，并尝试调试最近的页面。萨拉(Sara)和迈克(Mike)解释说，工程师并不需要多年的专业知识来使每项服务都变得相当熟练。他们指导团队从最初的原则探索服务，并鼓励Nooglers熟悉服务。他们对知识的局限持开放态度，并教别人何时寻求帮助。

在整个升级过程中，新的SRE团队并不孤单。Sara和Mike前往与其他SRE团队和产品开发人员会面，并向他们学习。新的SRE团队通过举行视频会议，交换电子邮件和通过IRC聊天与Kirkland和London团队会面。此外，该团队参加了每周的生产会议，阅读每日的值班交接和事故报告，并浏览了现有的服务文档。对Kirkland SRE进行了访问，以进行演讲和回答问题。伦敦SRE整理了一套完整的灾难场景，并在Google的灾难恢复培训周期间进行了测试(请参阅*Site Reliability Engineering*中的第33章“[Preparedness and Disaster Testing](#)”)。

团队还通过“轮盘赌”培训练习进行了值班训练(请参阅《站点可靠性工程》第28章中的“[灾难角色扮演](#)”部分，其中他们扮演了最近发生的事件的角色，以练习调试生产问题。在这些会议期间，鼓励所有SRE提供有关如何解决模拟生产故障的建议。每个人都准备好之后，团队仍然举行这些会议，轮流在每个团队成员中担任会议负责人。团队记录了这些内容，以备将来参考。

在进行呼叫之前，该团队检查了有关值班工程师职责的精确指南。例如：

- 在每个班次开始时，值班工程师都会读取上一个班次的切换。
- 值班工程师首先将对用户的影响降到最低，然后确保问题得到完全解决。
- 在轮班结束时，值班工程师将越区切换电子邮件发送给下一个工程师值班。

该准则还规定了何时升级给他人，以及如何为大事件写事故报告。

最后，团队阅读并更新了值班中的剧本。剧本包含有关如何响应自动警报的高级说明。它们解释了警报的严重性和影响，并包括调试建议以及为减轻影响并完全解决警报而可能采取的措施。在SRE中，每当创建警报时，通常都会创建一个相应的剧本条目。这些指南可减少压力，平均修复时间(MTTR)和人为错误的风险。

### 维护剧本

剧本中的详细信息以与生产环境变化相同的速度过时。对于日常发行，剧本可能需要在任何一天进行更新。像任何形式的交流一样，编写良好的文档非常困难。那么，您如何维护剧本？

Google的一些SRE主张保持剧本条目的一般性，以使它们变化缓慢。例如，对于所有“RPC Errors High”警报，它们可能只有一个条目，供受过训练的值班工程师阅读，并结合当前警报服务的体系结构图。其他SRE提倡逐步编写手册，以减少人为变异并降低MTTR。如果您的团队对剧本内容有不同的看法，则剧本可能会在许多方面被拉扯。

这是一个有争议的话题。如果您没有其他意见，请至少与您的团队一起决定您的剧本必须包含哪些最小的结构化细节，并尝试注意何时您的剧本积累了许多超出这些结构化细节的信息。参与一个项目，将来之不易的新生产知识转化为自动化或监控控制台。如果您的剧本是每次特定警报触发时应召唤工程师运行的确定性命令列表，建议您实施自动化。

两个月后，Sara，Mike和SRE调动掩盖了即将离任的Kirkland SRE团队的值班班次。在三个月的时间里，它们成为了主要的值班服务，而Kirkland SRE作为备份。这样，如果需要，他们可以轻松升级到Kirkland SRE。接下来，Nooglers屏蔽了经验更丰富的本地SRE，并加入了轮换。

良好的文档资料和前面讨论的各种策略都帮助团队奠定了坚实的基础并迅速成长。尽管随叫随到可能会带来压力，但车队的信心已增长到足以采取行动，而不必再事后批评自己了。知道他们的反应是基于团队的集体知识，并且即使他们逐步升级，仍可以将值班工程师视为合格的工程师，这在心理上是安全的。

### 后记

在Mountain View SRE不断增加的同时，他们了解到他们经验丰富的姐妹SRE在伦敦的团队将继续进行新项目，并且正在苏黎世组建新的团队以支持以前由SRE伦敦团队提供的服务。对于第二次转换，Mountain View SRE使用的相同基本方法被证明是成功的。Mountain View SRE先前的投资是用于开发入门和培训材料，这有助于苏黎世SRE新团队的建立。

虽然当一群SRE组成团队时，Mountain View SRE所使用的方法很有意义，但是当在指定时间只有一个人加入团队时，他们需要一种更轻量级的方法。为了预期将来的营业额，SRE创建了服务架构图，并将基本培训清单正式化为一系列练习，这些练习可以半独立地完成，而导师的参与则很少。这些练习包括描述存储层，执行容量增加以及检查如何路由HTTP请求。

## Evernote：在云端找到我们的双脚

将我们的本地基础架构迁移到云中

我们没有打算重新设计我们的通话流程，但是就像生活中的许多事情一样，必要性是发明之母。在2016年12月之前，Evernote仅在为支持我们的整体应用程序而构建的本地数据中心上运行。我们的网络和服务器在设计时考虑了特定的体系结构和数据流。再加上许多其他限制，意味着我们缺乏支持水平架构所需的灵活性。

Google Cloud Platform(GCP)为我们的问题提供了具体的解决方案。但是，我们仍然要克服一个主要障碍:将我们所有的生产和支持基础架构迁移到GCP。快进70天。通过艰苦的努力和许多非凡的壮举(例如，移动了数千台服务器和3.5 PB的数据)，我们很高兴地安家落户。但是，在这一点上，我们的工作还没有完成:我们将如何监控，警报并(最重要的是)如何应对新环境中的问题？

### 调整我们的值班政策和流程

向云的迁移释放了我们的基础设施快速增长的潜力，但是我们尚未建立值班的政策和流程来应对这种增长。迁移结束后，我们便着手解决问题。在我们以前的物理数据中心中，我们几乎在每个组件中都内置了冗余。这意味着，尽管考虑到我们的规模，组件故障很常见，但通常没有单个组件能够对用户产生负面影响。因为我们控制了基础架构，所以基础架构非常稳定-任何小小的碰撞都不可避免地是由于系统某处的故障引起的。我们在构造警报策略时考虑到了这一点:丢弃了一些数据包，导致JDBC(Java数据库连接)连接异常，始终意味着VM(虚拟机)主机处于故障边缘，或者开关控制平面处于故障状态。甚至在我们进入云的第一天之前，我们就意识到这种类型的警报/响应系统今后将难以为继。在实时迁移和网络延迟的世界中，我们需要采取更加全面的方法进行监控。

根据第一原则来重新定义寻呼事件，并将这些原则写下为我们的显式SLO(服务级别目标)，有助于使团队明确警告的重要内容并允许我们减少监控基础结构中的负担。我们专注于API响应度等较高级别的指标，而不是MySQL中InnoDB行锁等待之类的较低级基础结构，这意味着我们可以将更多的时间集中在用户在中断期间所遭受的实际痛苦。对于我们的团队来说，这意味着更少的时间花费在寻找瞬态问题上。

这转化为更多的睡眠，有效性以及最终的工作满意度。

### 重组我们的监控和指标

我们的主要值班轮换人员由一小组但零散的工程师组成，他们负责我们的生产基础架构和少数其他业务系统(例如，暂存和构建管道基础架构)。我们有一个每周24/7的时间表，其中安排了良好的交接程序，并且每天早上都会对事故进行每日回顾。我们的团队规模较小，责任范围相对较大，因此我们必须尽一切努力保持流程负担轻，并着重于尽快关闭警报/分类/补救/分析循环。我们实现这一目标的方法之一是通过维护简单但有效的警报SLA(服务水平协议)来保持低信噪比。我们将由指标或监控基础结构生成的任何事件分为三类:

#### P1:立即处理

- 应立即采取行动
- 呼叫值班人员
- 指挥事件分类
- 影响SLO

#### P2:在下一个工作日处理

- 通常不面向客户，或范围非常有限

- 向团队发送电子邮件并通知事件流频道

#### P3: 活动仅供参考

- 信息收集在仪表板，被动电子邮件等中
- 包括容量规划相关信息

任何P1或P2事件都附加有事件凭单。该故障单用于明显的任务，例如事件分类和跟踪补救措施，以及SLO的影响，出现的次数和事后报告文档链接(如果适用)。

当事件(类别P1)出现时，值班人员的任务是评估对用户的影响。将事件分为1至3级。对于严重性为1(严重性为1)的事件，我们维护了一组有限的标准，以使对响应者的升级决策尽可能简单。一旦事件升级，我们将组建一个事件团队并开始我们的事件管理流程。寻找这个事故经理，选择一名记录员和一名沟通负责人，并打开我们的沟通渠道。解决此事件后，我们会进行自动事故报告并在公司内部和全世界共享结果。对于等级为2或3的事件，值班响应者将处理事件生命周期，包括简短的事后回顾以进行事件审查。

保持流程轻量化的好处之一是，我们可以明确地将值班从项目工作的任何期望中解放出来。这将授权并鼓励待命人员立即采取后续行动，并在完成事后审查后确定工具或过程中的主要缺陷。通过这种方式，我们在每次通话时都实现了一个持续不断的改进和灵活性循环，与环境的快速变化保持同步。目标是使每次值班时的班次都比上次班好。

#### 随着时间的推移跟踪我们的表现

随着SLO的引入，我们希望跟踪一段时间内的绩效，并与公司内的利益相关者共享该信息。我们实施了每月一次的服务审查会议，向有兴趣的任何人开放，以审查和讨论服务的前一个月。我们还使用此论坛来审查我们的通话负担，以此作为团队健康的晴雨表，并讨论超出值班预算时的补救措施。该论坛的双重目的是在公司内部传播SLO的重要性，并使技术组织对维护我们的服务和团队的健康负责。

#### 与CRE合作

用SLO表示我们的目标，为与Google客户可靠性工程(CRE)团队互动打下了基础。在我们与CRE讨论了SLO以了解它们是否现实和可衡量之后，两个团队都决定将CRE与我们自己的工程师一起分入进行影响SLO的活动。查明隐藏在云抽象层背后的根本原因可能很困难，因此让我们的Googler帮您摆脱黑匣子事件分类的猜测是很有帮助的。更重要的是，此练习进一步降低了我们的MTTR，这最终是我们用户所关心的。

#### 保持自我延续的周期

现在，我们没有更多的时间花在分类/根本原因分析/事后分析周期中，而是有更多的团队时间来思考如何推动业务发展。具体而言，这转化为诸如改善我们的微服务平台和为我们的产品开发团队建立生产准备标准的项目。后者包含了我们在调整值班结构时遵循的许多原则，这对于第一个“搭载传呼机”牛仔竞技表演的团队特别有用。因此，我们使每个人都可以改善值班时间。

## 实际实施细节

到目前为止，我们已经讨论了Google内部和Google外部有关值班设置的详细信息。但是关于值班的具体考虑呢？以下各节将更深入地讨论这些实现细节：

- 值班负载-它是什么，它如何工作以及如何对其进行管理
- 如何在值班安排中考虑灵活性，以为SRE建立更健康的工作/生活平衡
- 在给定的SRE团队内部以及与合作伙伴团队一起改善团队动力的策略

### 值班负载的剖析

您的传呼机很吵，这使您的团队不高兴。您已经阅读了*Site Reliability Engineering* 中的[第31章](#)，并与您的团队和所支持的开发人员团队举行了定期的生产会议。现在，每个人都知道您的值班工程师不满意。接下来是什么？

**呼叫负载**是值班工程师在典型的轮班时间内(例如每天或每周)收到的寻呼事件数。一个事件可能涉及多个页面。在这里，我们将逐步了解各种因素对值班负载的影响，并提出最小化未来值班负载的技术。

#### 适当的响应时间

除非有充分的理由，否则工程师不必在接到呼叫后的几分钟内就使用计算机处理问题。虽然完全停止面向客户的创收服务通常需要立即做出响应，但是您可以在几个小时内处理不太严重的问题(例如，备份失败)。

我们建议您检查当前的呼叫设置，以查看是否实际应该\*页面当前触发呼叫的所有内容。您可能正在寻找可以通过自动修复更好地解决的问题(因为通常，计算机解决问题要比要求人工解决问题更好)或故障单(如果实际上不是高优先级)。表8-1显示了一些示例事件和适当的响应。

表8-1. 实际响应时间示例

事件描述	响应时间	SRE影响
影响收入的网络中断	5分钟	SRE必须始终保持在经过充电且经过身份验证且可以访问网络的笔记本电脑的范围内；不能旅行；在任何时候都必须与副手密切协调
客户订单批处理系统卡住	30分钟	SRE可以离开家进行快速差事或短途通勤；副手无需在此期间提供掩护
用于启动前服务的数据库备份失败	工单(工作时间内的回复)	无

#### 场景: 团队超负荷

(假设的)Connection SRE团队负责前端负载平衡和终止最终用户连接，发现自己处于高值班负载的位置。他们已经确立了每班两次呼叫事件的值班预算，但是在过去的一年中，他们经常每班接收五次呼叫事件。分析显示，三分之一的班次超出了其值班的预算。团队成员英勇地回应了每天的页面攻击，但无法跟上。一天中根本没有足够的时间来找到根本原因并正确解决即将出现的问题。一些工程师离开了团队，加入了较少运维负担的团队。高质量的事件跟踪很罕见，因为应召唤的工程师只有时间来缓解眼前的问题。

团队的视野并不完全黯淡:他们拥有遵循SRE最佳实践的成熟监控基础架构。警报阈值设置为与其SLO相符，并且寻呼警报本质上是基于症状的，这意味着它们仅在客户受到影响时才触发。当与高级管理层取得所有这些信息时，他们同意团队处于运营超负荷状态，并审查了项目计划以使团队恢复健康状态。

不太乐观的是，随着时间的流逝，Connection团队已经从10多个开发人员团队获得了软件组件的运维所有权，并且对Google面向客户的边缘网络和骨干网络非常依赖。大量的团体间关系是复杂的，并且已经变得难以管理。尽管团队遵循了构建监控的最佳实践，但是他们所面临的许多呼叫都不在他们的直接控制之下。例如，黑匣子探针可能由于网络拥塞而失败，从而导致数据包丢失。团队可以采取的缓解骨干网拥塞的唯一措施是升级到直接负责该网络的团队。

除了运维负担之外，团队还需要向前端系统交付新功能，所有Google服务都将使用这些新功能。更糟的是，他们的基础架构已从拥有10年历史的旧框架和群集管理系统迁移到支持更好的替代品。团队的服务发生了前所未有的变化，变化本身造成了很大的值班负担。

团队显然需要使用多种技术来应对这种过多的值班负载。团队的技术项目经理和人员经理与高级管理人员提出了项目建议，高级管理人员对此项目进行了审查和批准。团队全神贯注于减少值班负荷，并在此过程中吸取了一些宝贵的经验教训。

### 呼叫负载输入

解决高值班负载的第一步是确定是什么原因造成的。值班负载受三个主要因素影响:生产，警报和人工流程中的错误<sup>64</sup>。这些因素中的每一个都有几个输入，我们将在本节中更详细地讨论其中的一些输入。生产:

- 生产中现有错误的数量
- 将新的错误引入生产
- 识别新引入的错误的速度
- 漏洞消除和从生产中删除的速度

警报:

- 触发寻呼警报的警报阈值
- 引入了新的呼叫警报
- 服务的SLO与它所依赖的服务的SLO的对齐

对于人工流程:

- 严格的修复和错误的跟进
- 收集有关呼叫警报的数据的质量
- 注意传呼机负载趋势
- 人为改变生产

**先前存在的错误。**没有系统是完美的。生产中总会有bug:在您自己的代码，构建的软件和库或它们之间的接口中。这些错误可能暂时不会引起呼叫警报，但是肯定存在。您可以使用一些技术来识别或阻止尚未引起页面警报的错误:

- 确保系统尽可能复杂，不要再复杂了(请参阅第7章)。

- 定期更新构建系统所基于的软件或库，以利用错误修复(但是，请参阅下一节有关新错误的信息)。
- 进行定期的破坏性测试或模糊测试(例如，使用Netflix的Chaos Monkey)。
- 除了集成和单元测试外，还执行常规的负载测试。

**新错误.**理想情况下，SRE团队及其合作伙伴开发团队应该在甚至将其投入生产之前就检测出新的错误。实际上，自动化测试会遗漏许多错误，然后将其发布到生产环境中。

软件测试是一个在其他地方(例如[Martin Fowler on Testing](#)上广泛涉及的主题。但是，软件测试技术对于减少到达生产环境的错误数量以及在生产中保留的时间特别有用:

- 随着时间的推移改进测试。特别是，对于在生产中发现的每个错误，请询问"我们如何检测到该错误的预生产？"确保进行必要的工程跟进(请参阅第164页的"跟进工作")。
- 不要忽略负载测试，负载测试通常比功能测试的优先级低。许多错误仅在特定的负载条件下或在特定的请求混合下才会显现。
- 在类似生产的环境中运行场景(使用类似生产的但综合的流量进行测试)。在本书的第5章中，我们简要讨论了生成综合流量。
- 在生产环境中执行金丝雀(第16章)。
- 对新错误的容忍度较低。遵循"检测，回滚，修复和前滚"策略，而不要遵循"检测，识别出错误，修复并再次前滚但继续向前滚"策略。(有关更多详细信息，请参见第162页的"缓解延迟"。)

这种回滚策略需要可预测且频繁的发行，因此回滚任何一个发行的成本很小。我们在[站点可靠性工程](#)中讨论了此主题和相关主题。《[站点可靠性工程](#)》

某些错误可能仅是由于更改客户端行为引起的。例如:

- 仅在特定负载水平下才会显示的错误-例如，9月返校流量，黑色星期五，网络星期一或夏令时表示欧洲和北美距离一小时的一年中的那一周，意味着您的更多用户同时处于清醒状态和在线状态。
- 仅在特定的请求混合中出现的错误-例如，由于亚洲字符集的语言编码，距离亚洲较近的服务器的通信量比较昂贵。
- 仅当用户以意外方式使用系统时才会出现的错误-例如，航空公司预订系统正在使用日历！因此，重要的是扩展您的测试方案以测试每天都不会发生的行为。

当生产系统遇到多个并发错误时，要确定给定页面是用于"现有"还是"新"错误会更加困难。最大限度地减少生产中的错误数量不仅减少了值班的负担，而且还使识别和分类新的错误变得更加容易。因此，至关重要的是尽快从系统中删除生产错误。在提供新功能之前优先解决现有的错误；如果这需要跨团队合作，请参阅第18章。

诸如自动化运行状况检查，自我修复和减载之类的架构或程序问题可能需要大量的工程工作才能解决。请记住，为简单起见，即使这些问题的大小，复杂性或解决这些问题所需的工作非常重要，我们也将它们视为"错误"。

[Site Reliability Engineering]的第3章描述了错误预算如何有效地管理将新错误发布到生产中的速率。例如，当一项服务的SLO违规超出其季度总错误预算的某个比例时(通常是开发人员和SRE团队之间事先达成的协议)，可以暂时停止新功能的开发和与功能相关的部署，以专注于稳定化系统并减少页面的频率。

我们示例中的Connection团队采用了严格的策略，要求每次中断都必须具有跟踪错误。这使团队的技术项目经理可以检查其新错误的根本原因。这些数据表明，人为错误是生产中新错误的第二大最常见原因。

因为人是容易出错的，所以最好对生产系统进行的所有更改都是由(人为开发的)意图配置提供的自动化来完成的。在更改生产之前，自动化可以执行人类无法执行的其他测试。Connection团队正在半手动进行生产的复杂更改。毫不奇怪，团队的手动更改有时会出错。团队引入了新错误，这些错误导致了页面的出现。进行相同更改的自动化系统将确定，更改在进入生产并成为呼叫事件之前是不安全的。技术项目经理将这些数据带给团队，并说服他们确定自动化项目的优先级。

**识别延迟。**及时识别警报的原因很重要，因为识别页面的根本原因花费的时间越长，重复出现和再次被呼叫的机会就越大。例如，给定一个仅在高负载下显示的页面，例如在每天的峰值下，如果在下一个每天的峰值之前未识别出有问题的代码或配置，则该问题很可能会再次发生。您可以使用多种技术来减少识别延迟：

#### 使用良好的警报和控制台

确保页面链接到相关的监控控制台，并且该控制台会突出显示系统超出规范运行的位置。在控制台中，将黑盒和白盒呼叫警报关联在一起，并对其关联的图形执行相同的操作。确保剧本是最新的，并且具有响应每种警报的建议。当相应页面触发时，值班工程师应使用最新信息更新剧本。

#### 实践应急响应

运行“轮盘赌”练习(在Site Reliability Engineering中进行了描述)，与您的同事共享常规的和特定于服务的调试技术。

#### 执行小发布

如果您执行频繁，较小的发布，而不是进行罕见的整体更改，则将错误与引入错误的相应更改关联起来会更容易。第16章中描述的金丝雀版本强烈地表明了新错误是否是由于新版本引起的。

#### 日志更改

将变更信息汇总到可搜索的时间轴中，可以更轻松(并且希望更快)将新错误与引入它们的相应变更相关联。诸如Jenkins的Slack插件之类的工具可能会有所帮助。

#### 请求帮忙

在Site Reliability Engineering (站点可靠性工程) "Managing Incidents"中，我们讨论了共同管理大型停机的问题。值班工程师永远不会孤单；鼓励您的团队在寻求帮助时感到安全。

**缓解延迟**一旦发现漏洞，缓解漏洞所需的时间越长，再次出现和再次呼叫的机会就越大。考虑以下减少缓解延迟的技术：回滚更改

-如果错误是在最近的代码或配置首次发布中引入的，请在安全且适当的情况下立即通过回滚从生产中删除该错误(例如，仅进行回滚可能是必要的，但如果该错误导致数据损坏，则不足够)。请记住，即使是“快速修复”也需要时间进行测试，构建和推出。测试对于确保快速修复程序能够真正修复该错误，并且不会引入其他错误或其他意外后果至关重要。通常，最好是“后退，修复和前滚”而不是“前滚，修复和前滚”。

- 如果您希望99.99%的可用性，则每个季度大约有15分钟的错误预算。前滚的构建步骤可能需要15分钟以上的时间，因此回滚对用户的影响要小得多。

(99.999%的可用性提供了每季度80秒的错误预算。此时，系统可能需要自我修复属性，这超出了本章的范围。)

- 尽可能避免使用无法回滚的更改，例如与API不兼容的更改和锁步发布。

#### 使用功能隔离

- 设计系统，以便如果功能X出了问题，您可以通过例如功能标记将其禁用，而不会影响功能Y。此策略还可以提高发布速度，并使禁用功能X变得更加简单，您可以-无需检查您的产品经理是否也可以禁用功能Y。

#### 排空请求

- 将请求(即重定向客户请求)从显示错误的系统元素中删除。例如，如果该错误是代码或配置推出的结果，并且您逐步将其投入生产，则您可能有机会删除已经收到更新的基础结构元素。这使您可以在几秒钟内减轻对客户的影响，而无需回滚，因为回滚可能需要几分钟或更长时间。

**警报** Google SRE每12小时轮班最多发生两次不同的事件，这鼓励我们对如何配置呼叫警报以及如何引入新警报保持谨慎和谨慎的态度。*Site Reliability Engineering*，“[监控分布式系统](#)”描述了Google定义呼叫警报阈值的方法。严格遵守这些准则对于保持健康的轮班至关重要。值得强调本章讨论的一些关键元素：

- 所有警报应立即可操作。我们希望人们在收到系统无法自行处理的页面后立即采取措施。信噪比应该很高，以确保几乎没有误报。低信噪比会增加值班工程师产生警报疲劳的风险。
- 如果团队完全订阅基于SLO的警报，或仅在消耗错误预算后才进行呼叫(请参阅[站点可靠性工程中的“黑盒子与白盒子”](#))，至关重要的是，所有参与开发和运维服务的团队都必须同意满足SLO的重要性，并据此确定其工作的优先级。
- 如果团队完全订阅基于SLO和基于症状的警报，则放松警报阈值未必是对呼叫的适当响应。
- 就像新的代码一样，新的警报也应该得到彻底和认真的审查。每个警报应具有一个对应的剧本条目。

接收呼叫会产生负面的心理影响。为了最大程度地减少这种影响，请仅在确实需要时才引入新的呼叫警报。团队中的任何人都可以编写新的警报，但是整个团队都会审查建议的警报添加内容并可以提出其他建议。全面测试生产中的新警报，以检查误报，然后再将其升级为呼叫警报。例如，您可以在警报触发时通过电子邮件向警报的作者发送电子邮件，而不是呼叫工程师。

新警报可能会发现您不知道的生产问题。解决了这些生产错误之后，警报将仅在新错误上呼叫，其功能类似于回归测试。

确保在测试模式下运行新警报的时间足够长，以经历典型的定期生产条件，例如常规软件推出，云提供商的维护事件，每周负载高峰等。一周的测试大概是正确的。但是，此适当的窗口取决于警报和系统。

最后，在测试期间使用警报的触发率来预测由于新警报而导致的值班预算的预期消耗。明确批准或禁止以团队为单位的新警报。如果引入新的寻呼警报导致您的服务超出其寻呼预算，则需要特别注意系统的稳定性。

**后续措施。**旨在确定每个页面的根本原因。“根本原因”从机器延伸到团队的流程。中断是由单元测试所捕获的错误引起的吗？根本原因可能不是代码中的错误，而是团队中围绕代码审查的过程中的错误。

如果您知道根本原因，则可以修复并防止它再次困扰您或您的同事。如果您的团队无法找出根本原因，请添加监控和/或日志记录，这将帮助您在下次出现该页面时找到该根本原因。如果您没有足够的信息来确定该错误，则可以随时做一些事情来帮助下次进一步调试该呼叫。您应该不常断定呼叫是由“原因未知”触发的。请记住，作为一名随时待命的工程师，您并不孤单，因此请同事检查您的发现，看看是否有任何遗漏。通常，最容易在触发警报并获得新证据后立即找到警报的根本原因。

将页面解释为“瞬态”，或者因为系统“自行修复”或错误莫名其妙地“消失”而没有采取任何措施，导致错误再次发生并导致另一个呼叫，这给下一位值班的工程师带来了麻烦。

简单地修复即时错误(或进行“点”修复)会错过千载难逢的机会，以防止将来出现类似的警报。使用呼叫警报作为表面工程工作的机会，以改善系统并消除整个类别的未来可能的错误。为此，请在团队的生产组件中提交项目错误，并主张通过收集有关该项目将删除多少个单独的错误和呼叫的数据来确定其实现的优先级。如果您的提案需要3个工作周或120个工作小时来实施，而一个呼叫的平均处理时间为4个小时，则30个呼叫后会有一个明显的收支平衡点。

例如，假设出现以下情况:同一故障域上的服务器过多，例如数据中心中的交换机，导致经常发生多个同时发生的故障:

#### 点修复

在更多故障域上重新平衡当前的覆盖范围，并在此处停止。

#### 系统修复

使用自动化来确保此类服务器和所有其他类似服务器始终分布在足够的故障域中，并在必要时自动重新平衡。

#### 监控(或预防)修复

当故障域多样性低于预期水平但尚未影响服务时，会先发警报。理想情况下，该警报将是故障单警报，而不是呼叫警报，因为它不需要立即响应。尽管冗余程度较低，该系统仍可以幸福地服务。

为了确保您对呼叫警报的后续工作彻底，请考虑以下问题:

- 如何防止再次发生此特定错误？
- 如何防止此错误再次在此系统和我负责的其他系统中再次发生？
- 哪些测试可以阻止此错误发布到生产环境？
- 哪些故障单警报可能已触发操作，以防止错误在呼叫之前变得很严重？

- 在控制台上的bug变得严重之前，哪些信息警报可能已经在控制台上暴露了该bug？
- 我是否已最大限度地发挥修复作用？

当然，值班工程师仅记录与相关的呼叫警报是不够的。快速处理SRE团队发现的错误，以减少它们再次发生的可能性，这一点非常重要。确保SRE和开发人员团队的资源规划都考虑到响应错误所需的工作。

我们建议您保留一小部分SRE和开发人员时间，以应对生产错误。例如，一个Google值班工程师通常在值班期间不处理项目。相反，他们致力于改善系统运行状况的错误。确保您的团队例行地将生产错误排在其他项目工作之上。SRE经理和技术负责人应确保及时处理生产错误，并在必要时上报给开发团队决策者。

当呼叫事件严重到需要进行事后检查时，采用这种方法对后续行动项目进行分类和跟踪就显得尤为重要。(有关更多详细信息，请参见第10章。)

**数据质量** 一旦确定了导致呼叫的系统错误，自然就会产生许多问题：

- 您怎么知道首先要修复哪个错误？
- 您怎么知道系统中的哪个组件导致了大多数呼叫？
- 您如何确定呼叫工程师采取哪些重复的手动操作来解决呼叫？
- 您如何知道还有多少原因不明的警报？
- 如何判断哪些错误是"真正的"错误，而不仅仅是轶事，或者更严重的事件？

答案很简单：收集数据！

在建立数据收集流程时，您可能会跟踪和监控呼叫负载中的模式，但是这种努力无法扩展。为您的错误跟踪系统(例如，Jira，IssueTracker)中的每个寻呼警报提交占位符错误，并且由值班工程师创建链接，这种做法更具可持续性当监控系统的呼叫警报和错误跟踪系统中的相关错误意识到每个警报都是预先存在的问题的症状时，就会在此之间进行切换。最后，您将在一列中列出一个尚未理解的错误，并在下一列中列出每个错误被认为导致的所有呼叫。

一旦构造了有关呼叫原因的数据，就可以开始分析该数据并生成报告。这些报告可以回答以下问题：

- 哪些错误导致最多的呼叫？理想情况下，我们会立即回滚并修复错误，但是有时，找到根本原因并部署修复程序会花费很长时间，并且有时使关键警报静默不是一个合理的选择。例如，前面提到的Connection SRE团队可能会遇到持续的网络拥塞，这种拥塞无法立即解决，但仍需跟踪。收集导致生产问题最多的页面上的数据，并给团队带来压力，这有助于以数据为驱动力的对话，系统地优先安排工程工作。
- 导致大多数呼叫警报(付款网关，身份验证微服务等)的原因是系统的哪个组件？
- 当与您的其他监控数据关联时，特定呼叫是否对应于其他信号(请求率峰值，同时进行的客户会话次数，注册次数，提款次数等)？

将结构化数据绑定到错误和页面的根本原因还有其他好处：

- 您可以自动填充现有错误(即已知问题)的列表，这可能对您的支持团队有用。

- 您可以根据每个错误导致的呼叫数自动确定修复错误的优先级。

您收集的数据的质量将决定人或自动机可以做出的决策的质量。为了确保高质量的数据，请考虑以下技术：

- 定义并记录您的团队对页面数据收集的政策和期望。
- 从监控系统设置非呼叫警报，以突出显示未根据那些期望处理呼叫的位置。经理和技术负责人应确保满足期望。
- 当交接不符合预期时，队友应互相跟进。积极评论，例如“可能与错误123有关”，“我已将您的发现与错误一并提交，以便我们进行更详细的跟进”或“这很像上周三我上班时发生的情况”。

**警觉** 很多时候，团队因削减一千次而陷入运维超负荷状态。为了避免使温水煮青蛙，重要的是要注意随时间推移的值班工程师的健康状况，并确保SRE和开发人员团队。

以下技术可以帮助团队密切关注值班的负载：

- 在生产会议上(请参阅 "[Site Reliability Engineering](#)，第31章中的“沟通: 生产会议”，定期根据收集的结构化数据讨论值班负载的趋势。我们发现21天的追踪平均水平很有用。
- 设置故障单警报，可能针对技术主管或经理，以使传呼机负载超过您的团队事先同意的“警告”阈值。
- 在SRE团队和开发人员团队之间举行定期会议，讨论生产的当前状态以及呼叫了SRE的突出生产错误。

## 值班灵活性

### 班次长度

每天必须处理一次或多于一次呼叫的值班轮换必须以可持续的方式进行：我们建议将轮班时间限制为12小时。换班时间越短，对工程师的心理健康越好。团队成员长时间轮班时会感到精疲力尽，而当人们感到疲倦时，他们会犯错。如果不断有人值班，大多数人根本无法生产出高质量的作品。许多国家/地区都有关于最长工作时间，休息时间和工作条件的法律。

虽然在团队的白天分散值班轮换时间是理想的选择，但12小时的轮班系统并不需要在全球范围内分布团队。夜间值守12小时的值班时间比24小时或更长的值班时间更可取。您甚至可以在一个地点进行12小时轮班工作。例如，与其让单个工程师在整个整整一周的轮班中每天24小时待命，不如让两个工程师在一周的待命时间内安排一个人白天待命，一个夜间待命。

根据我们的经验，24小时值班不是可持续的设置。虽然不理想，但偶尔隔夜12小时轮班至少可以确保工程师休息。另一种选择是将轮班时间缩短到不到一周-大约3天，4天休息。

### 场景: 个人情况的改变

想象一下，您是一个大型服务的值班团队的成员，该服务在两个站点之间分布有一个24/7的全天候式的服务。该安排对您来说效果很好。尽管您对每早晨6点的呼叫的可能性并不感到激动，但您和您的团队为保持可管理的工作负载同时提高服务可靠性所做的工作感到满意。

一切都很好...直到有一天，您意识到值班时间安排和您个人生活的需求开始发生冲突。原因有很多潜在的原因-例如，成为父母，需要在短时间内出差并请假或生病。

您需要自己的值班职责与新的个人日程安排共存。

随着成熟，许多团队和组织都面临着这一挑战。人们的需求会随着时间的推移而变化，保持不同队友背景的健康平衡会导致以各种需求为特征的值班轮换。保持值班工作和个人生活之间健康，公平和公正的平衡的关键是灵活性。

您可以通过多种方式将灵活性应用于值班轮换，以满足团队成员的需求，同时仍然确保覆盖服务或产品。不可能写下一套全面，适合所有人的准则。我们鼓励将灵活性作为"原则"，而不是简单地采用此处列出的示例。

**自动安排值班时间。**随着团队的成长，考虑到日程安排的限制-假期计划，平日与周末的值班时间分配，个人喜好，宗教信仰等等-变得越来越困难。您无法手动管理此任务；根本找不到任何解决方案，更不用说公平的解决方案了。"公平"并不意味着团队成员中每种轮班的分配完全均匀。不同的人有不同的需求和不同的偏好。因此，团队共享这些偏好并尝试以一种明智的方式满足他们，这一点很重要。团队的组成和偏好决定了您的团队是希望采用统一分布还是更个性化的方式来满足计划偏好。

使用自动化工具安排值班班次将使这项任务变得更加容易。该工具应具有一些基本特征：

- 应该重新安排值班时间，以适应团队成员不断变化的需求。
- 它应响应任何更改自动重新平衡通话中的负载。
- 它应该通过考虑个人喜好(例如" 4月的周末不上班")以及历史信息(例如每个工程师的最新值班负荷)来尽力确保公平。
- 为了让值班工程师能够计划他们的值班时间，绝不能更改已经生成的时间表。

日程安排生成可以是完全自动化的，也可以是人工安排的。同样，有些团队希望让成员明确按时退出计划，而另一些团队则对完全自动化的过程感到满意。如果您的需求很复杂，则可以选择内部开发自己的工具，但是有许多商业和开源软件包可以帮助自动安排值班时间。

**计划进行短期互换。**对值班时间表进行短期更改的请求经常发生。没有人可以保证星期一不会感染流感。否则，您可能需要在值班期间进行一些不可预见的紧急任务。

出于非紧急原因，您可能还希望促进值班调换，例如，允许值班者参加运动训练课程。在这种情况下，团队成员可以交换部分值班时间(例如，周日的一半)。非紧急交换通常是最大的努力。

具有严格值班响应SLO的团队需要考虑通勤范围。如果您的值班响应SLO为5分钟，而通勤时间为30分钟，则您需要确保上班时其他人可以响应紧急情况。

为了灵活地实现这些目标，我们建议使团队成员能够更新值班轮换。另外，应有成文的政策来描述交换的工作方式。权力下放的选择范围从完全集中的策略(只有经理可以更改计划)到完全分散的策略(任何团队成员都可以独立更改策略)。根据我们的经验，对更改进行同行评审可以在安全性和灵活性之间取得良好的平衡。

**计划长期休假。**有时，由于个人情况的变化或精疲力尽，团队成员需要停止按轮换上任。重要的是，团队的结构应允许值班者暂时离开轮换。理想情况下，团队规模应允许减少(临时)人员，而不会导致团队的其他成员承担过多的运营负担。根据我们的经验，在多站点24/7配置中，每个站点至少需要五个人来维持值班，而在24/7配置中的单站点中则需要八个人。因此，可以安全地假设每个站点将需要一名额外的工程师来防止人员减少，从而使最少的人员配备达到每个站点六名工程师(多站点)或每个站点九名工程师(单站点)。

**计划兼职工作时间表。**兼职与值班安排似乎不兼容，但我们发现如果采取某些预防措施，则值班和兼职工作安排可以兼容。以下讨论假定，如果您的轮班值班人员兼职工作，那么他们将无法在其兼职工作之外进行轮班。兼职工作有两种主要模式：

- 每周减少全天工作量，例如每周工作8小时，而不是5天
- 每天减少工作时间-例如每天6小时，而不是每天8小时

两种模型都可以与值班中的工作兼容，但是需要对值班安排进行不同的调整。

第一个模型很容易与值班中的工作共存，特别是在非工作日数随时间变化的情况下。作为响应，您可以采用每周少于7天(例如，周一至周四或周五至周日)的值班时间，并配置自动调度程序从而不将兼职工程师安排在他们不工作的时候值班。第二种模型有两种可能：

- 与另一位工程师一起分配值班时间，这样就不会有人在值班时间不在。例如，如果值班工程师需要从上午9点到下午4点工作，则可以将轮班的前半部分(从上午9点到下午3点)分配给他们。轮值团队中的下半场(下午3点至晚上9点)，方法与其他轮换值班的方法相同。
- 兼职工程师可以在上班日全职工作，如果值班轮换不太频繁，这可能是可行的。

如Site Reliability Engineering的[第十一章](#)所述，根据当地劳动法和其他法规，Google SRE通过简化的小时的报酬或休假率来补偿正常工作时间以外的支持。确定值班补偿时，请考虑兼职工程师的简化时间表。

为了在项目时间和值班时间之间保持适当的平衡，减少工作时间的工程师应按比例分配较少的值班时间。规模较大的团队比规模较小的团队更容易吸收额外的值班负载。

## 值班团队动态

我们的第一本书探讨了诸如高值班负载和时间压力之类的压力因素如何迫使值班工程师采用基于直觉和启发法而非基于原因和数据的决策策略(请参阅第11章“感觉安全”部分(<http://bit.ly/2JgUBU7>)。通过对团队心理学的讨论，您如何建立具有积极动力的团队？考虑一个具有以下假设问题的值班团队。

### 场景：“生存一周”的文化

公司始于几个创始人和少数员工，全部都是功能开发人员。每个人都认识其他人，每个人都呼叫。

公司发展壮大。值班人员仅限于经验较少的，对系统有更好了解的一小部分经验丰富功能开发人员。

公司发展壮大。他们添加了运维角色来解决可靠性。这个团队负责生产健康，工作角色集中在运维上，而不是编码上。值班成为功能开发人员和运维人员之间的联合轮换。功能开发人员在维护服务方面拥有最终决定权，而运维的输入仅限于操作任务。到此时，有30名工程师处于值班轮换状态：25个功能开发人员和5个运维人员都位于同一地点。

大量的值班困扰着该团队。尽管遵循了本章前面介绍的建议以最大程度地减少值班负荷，但是该团队的士气低落。由于功能开发人员优先开发新功能，因此后续值班需要很长时间才能实施。

更糟的是，功能开发人员担心自己子系统的健康状况。尽管团队中其他人对此有所抱怨，但一个功能开发人员坚持基于错误率呼叫而不是只对其关键任务模块的错误率。这些警报嘈杂，并返回许多误报或无法操作的呼叫。

值班轮换的其他成员不会因高值班量而特别烦恼。当然，有很呼叫，但是大多数呼叫不需要花费很多时间来解决。正如一位值班工程师所说：“我快速浏览了呼叫主题，并知道它们是重复的。所以我只是无视它们。”听起来有点熟？

一些Google团队在成立初期就遇到了类似的问题。如果处理不当，这些问题可能会使功能开发人员和运维团队分崩离析，并阻碍呼叫操作。没有解决这些问题的灵丹妙药，但是我们发现了一些特别有用的方法。尽管您使用的方法可能有所不同，但您的总体目标应该是相同的：建立积极的团队动力，并谨慎避免陷入困境。

**提案一：授权您的运维工程师。**您可以根据本书和*Site Reliability Engineering*(站点可靠性工程)中概述的准则来重塑运维组织，甚至可以包括名称更改(SRE或类似名称)以指示角色更改。简单地解除运维组织并不是万能的灵丹妙药，但是它可以帮助您摆脱以运维中心为中心的旧模式，从而传达责任的“实际”变化。向团队和整个公司清楚，SRE负责站点运营。这包括为可靠性定义共享的路线图，推动问题的全面解决，维护监控规则等。功能开发人员是必要的协作者，但并不掌控这些工作。

对我们假设的团队来说，此公告带来了以下运维变化：

- 操作项仅分配给五位DevOps工程师-他们现在是SRE。SRE与主体专家(其中许多具有开发人员)一起工作以完成这些任务。SRE通过与功能开发人员协商警报更改来承担前面提到的“错误率与错误率”的争论。
- 如果可能，鼓励SRE深入研究代码以自行进行更改。他们将代码检查发送给主体专家。这样做的好处是，可以在SRE之间建立主人翁意识，并提高他们的技能和权威，以备将来之需。

通过这种安排，功能开发人员将成为可靠性功能的明确合作者，而SRE则有责任把控和改善站点。

**提案二：改善团队关系。**另一个可能的解决方案是在团队成员之间建立更牢固的团队联系。Google指定了一个“娱乐预算”，专门用于组织场外活动以加强团队联系。

我们发现，更稳固的团队关系营造了增进队友之间理解和协作的精神。结果，工程师更有可能修复错误，完成操作项并帮助同事。例如，假设您关闭了每晚的管道作业，但是却忘记了关闭检查管道是否成功运行的监控。结果，您不小心在凌晨3点呼叫了一位同事。如果您与该同事共度了一段时间，您会对所发生的事情感到更糟，并通过日后的谨慎努力来变得体贴。“我保护我的同事”的心态转化为更具生产力的工作氛围。

我们还发现，无论职位名称和职能领域如何，让所有值班轮换成员坐在一起，都可以极大地改善团队关系。鼓励团队彼此一起吃午餐。不要低估此类相对简单的更改的功能。它直接影响团队动态。

## 结论

SRE值班与传统运维角色不同。SRE不仅完全专注于日常运维，还完全掌控生产环境，并通过定义适当的可靠性阈值，开发自动化并执行战略工程项目来寻求改善。值班对站点运维至关重要，正确处理对公司的底线至关重要。

值班是造成个人和集体紧张的根源。但是，如果您凝视怪物的眼睛足够长的时间，就会看到智慧的一面。本章说明了我们通过艰苦学习所学到的有关值班的一些教训。我们希望我们的经验可以帮助其他人避免或解决类似问题。

如果您的值班团队淹没在无尽的警报中，我们建议您退后一步，从更高层次观察情况。与其他SRE和合作伙伴团队比较笔记。收集必要的信息后，请系统地解决问题。为值班工程师，值班团队和整个公司花费大量时间进行周到的组织。

<sup>60</sup>. 请注意，对于实际上没有实践DevOps的组织，此示例通常是一个危险信号，在这种情况下，更名不会解决更多的结构性问题。 ↵

<sup>61</sup>. 一个"事件"被定义为一个"问题"，无论针对同一"问题"发出了多少警报。一班是12个小时。 ↵

<sup>62</sup>. David Blank-Edelman(O'Reilly)的*Seeking SRE*中有关于此主题的更多信息。 ↵

<sup>63</sup>. Google的SRE团队在各个时区配对，以确保服务的连续性。 ↵

<sup>64</sup>. 在这种情况下，"错误"是指由软件或配置错误引起的任何不良系统行为。代码中的逻辑错误，二进制文件的错误配置，不正确的容量计划，错误配置的负载平衡器或新发现的漏洞都是造成值班负载的"生产错误"的有效示例。 ↵

## 第9章

### 事件响应

由Jennifer Mace , Jelena Oertel , Stephen Thorne和Arup Chakrabarti(PagerDuty)与马剑和Jessie Yang撰写

每个人都希望他们的服务始终保持平稳运行，但是我们生活在一个不完美的世界，在这个世界中，确实发生了中断。当一个非常紧急的问题需要多个人或团队来解决时，会发生什么？您突然面临着同时管理事件响应和解决问题的挑战。

解决事件意味着减轻影响和/或将服务恢复到以前的状态。管理事件意味着高效地协调响应团队的工作，并确保响应者之间以及对事件进展感兴趣的人员之间进行交流。包括Google在内的许多科技公司已经采用并采用了最佳实践来管理来自应急响应组织的事件，这些组织已经使用这些实践多年。

事件管理的基本前提是结构化的方式响应事件。大规模事件可能会造成混乱；团队事先确定的结构可以减少混乱。制定有关如何在灾难发生之前进行沟通和协调工作的规则，使您的团队能够集中精力解决突发事件。如果您的团队已经练习并熟悉了沟通和协调，那么他们在事件发生时就不必担心这些因素。

设置事件响应流程并不一定是一项艰巨的任务。有许多广泛可用的资源可以提供一些指导，例如第一本SRE手册中的[管理事件](#)。事件响应的基本原则包括：

- 保持命令清晰。
- 指定明确定义的角色。
- 随时进行调试和缓解的工作记录。
- 尽早并经常公布事件。

本章说明了如何在Google和PagerDuty上设置事件管理，并提供了一些示例说明了正确执行此过程的地方以及不正确之处。如果您还没有一个简单的清单，请参阅第191页的“将最佳实践付诸实践”，可以帮助您开始创建自己的事件响应实践。

## Google事件管理

事件响应提供了一种用于响应和管理事件的系统。框架和一组已定义的过程使团队可以有效地响应事件并扩大响应。Google的事件响应系统基于[事件指挥系统\(ICS\)](#)。

### 事件指挥系统

ICS是由消防员于1968年成立的，旨在管理野火。该框架提供了在事件发生期间进行通信和明确指定角色的标准化方法。基于该模型的成功，公司后来改编了ICS以响应计算机和系统故障。本章探讨了两个这样的框架:[PagerDuty的事件响应过程](#)和[Google的事件管理\(IMAG\)](#)。

事件响应框架具有三个共同的目标，也称为事件管理的"三个C"(3C):

- 协调响应工作。
- 在组织内部和外部事件响应者之间进行沟通。
- 保持对事件响应的"控制"。

当事件响应出现问题时，罪魁祸首可能是这些其中之一。掌握3C对有效的事件响应至关重要。

### 事件响应中的主要作用

事件响应中的主要角色是事件指挥官(IC)，通信主管(CL)和运维或运维主管(OL)。IMAG将这些角色组织成一个层次结构:IC负责事件响应，CL和OL向IC报告。

当灾难袭来时，发布事件的人员通常会担任IC角色并指挥事件的高级别状态。IC专注于3C，并执行以下操作:

- 命令和"协调"事件响应，根据需要委派角色。默认情况下，IC承担尚未委派的所有角色。
- 有效沟通。
- 保持对事件响应的"控制"。
- 与其他响应者一起解决该事件。

IC可以将其角色移交给其他人并担任OL角色，也可以将OL角色分配给其他人。OL通过应用运维工具缓解或解决事件来响应事件。

在IC和OL致力于缓解和解决事件的过程中，CL是事件响应团队的公开代表。CL的主要职责包括向事件响应团队和利益相关者提供定期更新，以及管理有关事件的查询。

CL和OL都可以领导一个团队来帮助管理他们特定的事件响应区域。这些团队可以根据需要扩展或收缩。如果事件足够小，则可以将CL角色归还给IC角色。

## 实例探究

以下四个大规模事件说明了事件响应在实践中如何工作。其中三个案例研究来自Google，最后一个案例来自PagerDuty，该案例研究提供了其他组织如何使用ICS派生框架的观点。Google的示例始于未能有效管理的事件，然后发展为管理良好的事件。

### 案例研究1: 软件Bug---灯亮但没有人在家(Google)

此示例说明了未能尽早发布事件如何使团队无法使用快速有效地响应事件的工具。尽管这一事件在没有重大灾难的情况下得到解决，但提早升级将产生更快，更有组织的响应和更好的结果。

#### 上下文

Google Home是智能扬声器和家庭助理，可响应语音命令。语音命令与Google Home的软件(称为Google Assistant)进行交互。

当用户说出*hotword*(触发Google Assistant的给定短语)时，便开始与Google Home进行交互。通过训练助手来收听给定的热词，多个用户可以使用同一Google Home设备。识别说话者的热词模型在客户端上进行了训练，但是训练数据(即说话者识别文件)存储在服务器上。服务器处理双向数据流。为了处理繁忙时间的过载，服务器具有Google助手的配额策略。为了保护服务器免受太大的请求值的影响，配额限制明显高于给定设备上Google Assistant的基准使用量。

Google Assistant 1.88版本中的一个错误导致说话者识别文件的获取频率比预期的高50倍，超过了此配额。最初，美国中部的Google Home用户仅遭受很小的流量损失。但是，随着向所有Google Home设备的推广逐步增加，用户在2017年6月3日的周末丢失了一半的请求。

### 事件

在5月22日星期一PST上午11:48，Jasper(正在开发Google Home的电话)碰巧正在查看每秒查询(QPS)图，并发现了一些奇怪的地方:Google助理一直每30分钟ping一次训练数据，而不是每天一次。他停止了1.88版本的发布，该版本已扩展到25%的用户。他利用Google的错误跟踪系统创建了一个bug-我们将其称为错误12345-以探究这种情况的发生原因。关于该错误，他指出Google助手一天要ping 48次数据，从而使其超出其QPS容量。

另一位开发人员Melinda将问题与以前报告的错误联系在一起，我们将其称为错误67890:每当应用刷新设备身份验证和注册状态时，语音处理器就会重新启动。该错误计划在1.88版发布后修复，因此该团队要求暂时增加该模型的配额，以减轻额外查询带来的过载。

重新启动了1.88版，并继续推出，到5月31日(星期三)已达到50%的用户。不幸的是，该团队后来得知，错误67890虽然造成了一些额外的流量，但并不是Jasper注意到的频繁获取的真正根本原因。

当天早上，客户开始向Google支持团队报告问题:每当有人说"OK Google"(或任何其他激活Google Home的关键词)时，设备就会返回错误消息。此问题阻止用户向Google Assistant发送命令。该团队开始调查可能导致用户报告的错误的原因。他们怀疑配额问题，因此他们要求再次增加配额，这似乎可以缓解问题。

同时，团队继续调查错误12345，以查看导致错误的原因。尽管配额连接是在调试过程的早期建立的，但是客户端和服务器开发人员之间的沟通不畅导致开发人员在故障排除过程中走错了路，并且整个解决方案仍然遥不可及。

该团队还对Google Assistant的访问量为何不断达到配额限制感到困惑。客户端和服务器开发人员对客户端错误感到困惑，这些错误似乎不是由服务器端的任何问题触发的。开发人员在下一个版本中添加了日志记录，以帮助团队更好地理解错误，并有望在解决事件方面取得进展。

到6月1日(星期四)，用户报告该问题已解决。没有新问题的报道，因此版本1.88继续推出。但是，尚未确定原始问题的根本原因。

到6月3日星期六星期六凌晨，1.88版的发布率已超过50%。上线活动是在一个周末进行的，当时开发人员不容易获得。该团队没有遵循仅在工作日内执行首次上线的最佳做法，以确保开发人员可以在出现问题的情况下及时到达。

当6月3日(星期六)1.88版的发布率达到100%时，客户端再次达到服务器限制Google Assistant流量的要求。客户的新报告开始出现。Google员工报告说，他们的Google Home设备抛出错误。Google Home支持团队收到了许多有关此问题的

客户电话，推文和Reddit帖子，并且Google Home的帮助论坛显示了一个[正在增长的帖子](#)来讨论此问题。尽管有所有用户报告和反馈，但该错误并未升级为更高的优先级。

在6月4日星期日，随着客户报告数量的不断增加，支持团队最终将Bug优先级提高到了最高级别。该团队没有发布事件，而是继续使用Bug跟踪系统通过“常规”方法对问题进行故障排除。值班的开发人员注意到一个数据中心群集中的错误率，并对SRE进行了ping操作，要求它们引流它。同时，团队又提出了增加配额的请求。之后，开发团队的一名工程师注意到，引流将错误推到了其他单元中，这为配额问题提供了更多证据。下午3:33，开发人员团队经理再次增加了Google Assistant的配额，并且停止了对用户的影响。事件已结束。此后不久，团队确定了根本原因(请参见前面的“上下文”部分)。

### 评论

事件处理的某些方面确实进展顺利，而其他方面则有改进的余地。

首先，开发者在周末聚会，并提供了宝贵的意见来解决问题。这既好又坏。尽管团队重视这些人在周末所付出的时间和精力，但成功的事件管理不应依赖于个人的英勇努力。如果开发人员无法访问怎么办？归根结底，Google支持良好的工作与生活平衡-不应在空闲时间利用工程师来解决与工作有关的问题。相反，我们应该在上班时间进行首次上线或组织一次轮班，以便在下班时间提供有偿服务。

接下来，团队努力减轻该问题。Google始终致力于首先阻止事件的影响，然后找到根本原因(除非恰好在早期就确定了根本原因)。缓解问题后，了解根本原因对于防止问题再次发生也同样重要。在这种情况下，缓解措施在三个不同的情况下成功地阻止了影响，但是团队只有在发现根本原因后才能阻止问题再次发生。首次缓解后，最好将部署推迟到根本原因完全确定之前，避免周末发生重大干扰。

最后，当问题首次出现时，团队没有发布事件。我们的经验表明，受到管理事件的解决速度更快。尽早发布事件可确保：

- 防止客户端和服务器开发人员之间的通讯错误。
- 根本原因识别和事件解决更快地发生。
- 相关团队会在较早的时间加入，从而使外部沟通更加快捷顺畅。

集中通信是IMAG协议的重要原则。例如，发生灾难时，SRE通常会聚集在“作战室”中。作战室可以是会议室等物理位置，也可以是虚拟位置：团队可能会聚集在IRC频道或群聊上。这里的关键是将所有事件响应者聚集在一个地方，并进行实时沟通以管理(并最终解决)事件。

## 案例研究2：服务故障-如果可以，请缓存我

以下事件说明了当一个专家团队尝试调试具有如此多交互作用的系统时，会发生什么情况，以至于没有一个人可以“全部”掌握所有细节。听起来有点熟？

### 上下文

Kubernetes是由许多公司和个人贡献者共同构建的开源容器管理系统。Google Kubernetes Engine或GKE是由Google管理的系统，可为用户创建，托管和运行Kubernetes集群。此托管版本可操作控制平面，而用户则以最适合他们的方式上载和管理工作负载。

用户首次创建新集群时，GKE将获取并初始化其集群所需的Docker映像。理想情况下，这些组件是在内部获取并构建的，因此我们可以对其进行验证。但是，由于Kubernetes是一个开源系统，因此有时会在缝隙中插入新的依赖项。

### 事件

太平洋标准时间(PST)的一个星期四上午6:41，针对位于几个区域的CreateCluster探针故障，在伦敦针对ZKE GKE的SRE进行了呼叫。没有成功创建新集群。Zara检查了探测器的仪表板，发现两个区域的故障率均超过60%。她验证了此问题是否影响了用户创建新集群的尝试，尽管到现有集群的流量没有受到影响。Zara遵循GKE记录的程序，并于上午7:06发布事件。最初，有四人参与了该事件：

- Zara，他首先注意到了问题，因此被指定为默认的突发事件指挥官
- Zara的两名队友
- Rohit，客户支持工程师按事件程序进行了呼叫

由于Rohit总部位于苏黎世，因此Zara(IC)开设了GKE Panic IRC频道，团队可以在此进行调试。当其他两个SRE陷入监控和错误消息时，Zara解释了停机情况及其对Rohit的影响。到早上7:24时，Rohit向用户发布了一条通知，告知CreateCluster在欧美地区出现故障。这变成了一个大事件。

在上午7点至8:20之间，Zara，Rohit和其他人共同致力于解决问题。他们检查了集群启动日志，发现了一个错误：

```
error: failed to run Kubelet: cannot create certificate signing request: Post t
```

他们需要确定证书创建的哪一部分失败。SRE调查了网络，资源可用性和证书签名过程。所有人似乎都可以单独正常工作。上午8:22，Zara将事件调查摘要发布到了事件管理系统，并寻找可以帮助她的开发人员。

值得庆幸的是，GKE安排了一个开发人员待命，可以在紧急情况被呼叫。开发者Victoria加入了该频道。她要求提供跟踪错误，并要求团队将该问题上报给基础架构的值班团队。

现在是上午8:45。西雅图的第一个SRE Il-Seong到达办公室，喝了点咖啡，为当天的工作做好了准备。Il-Seong是一位资深的SRE，在事件响应方面拥有多年经验。当他被告知正在进行的事件时，他跳了起来帮助。首先，Il-Seong根据警报的时间检查了当天的释放情况，并确定当天的版本发布没有引发故障。然后，他开始了一份工作文件<sup>65</sup>以收集笔记。他建议Zara将事件升级到基础架构，云网络和计算引擎团队，以消除这些区域的根本原因。由于Zara的升级，更多人加入了事件响应：

- GKE节点的开发负责人
- 云网络值班团队
- 计算引擎值班团队
- Herais，另一个西雅图SRE

上午9:10，事件频道有十几位参与者。该事件发生了2.5小时，没有根本原因，也没有缓解措施。沟通正成为挑战。通常，从伦敦到西雅图的应召移交是在上午10点发生的，但是扎拉决定在上午10点之前将事件指挥权移交给Il-Seong，因为他对

IMAG有更多的经验。

作为事件指挥官，Il-Seong建立了一个正式的机构来处理事件。然后，他指定Zara为业务负责人，任命Herais为沟通(Comms)负责人。Rohit仍然是外部沟通主管。Herais立即向包括所有开发人员领导的GKE列表发送了“紧急集合”的电子邮件，并请专家加入事件响应。

到目前为止，事件响应者了解以下信息：

- 如果节点尝试向主节点注册，则集群创建失败。
- 错误消息表明证书签名模块是罪魁祸首。
- 欧洲所有集群创建均失败；所有其他大洲都很好。
- 欧洲没有其他GCP服务出现网络或配额问题。

多亏了紧急集合的呼吁，GKE安全团队成员Puanani参与了这项工作。她注意到证书签名器尚未启动。证书签名器试图从DockerHub中提取镜像，并且该镜像似乎已损坏。Victoria(正在召集GKE的开发人员)在两个地理位置运行了Docker的pull命令来处理该镜像。当它在欧洲的某个集群上运行而在美国的一个集群上成功时，它失败了。这表明欧洲集群是问题所在。在上午9:56，团队确定了一个合理的根本原因。

由于DockerHub是外部依赖项，因此缓解和定位根本原因将特别具有挑战性。缓解的第一个选择是让Docker中的某人快速修复映像。第二个选项是重新配置群集，以从其他位置获取镜像，例如Google的安全图像托管系统Google Container Registry(GCR)。所有其他依赖项，包括对镜像的其他引用，都位于GCR中。

Il-Seong指派负责人同时推动这两种选择。然后，他委托一个小组研究修复损坏的群集。对于IRC来说，讨论变得过于密集，因此详细的调试移到了共享文档上，IRC成为了决策的中心。

对于第二个选项，推送新配置意味着重建二进制文件，这花费了大约一个小时。上午10:59，当团队完成90%的重建时，他们发现了另一个使用错误的镜像获取路径的位置。作为响应，他们不得不重新启动构建。

在IRC的工程师研究两种缓解方案时，SRE的Tugay有了一个主意。与其重新构建配置并将其发布(这是一个繁琐且冒险的过程)，假如他们拦截了Docker的pull请求并用内部缓存的映像替换了Docker的响应该怎么做？GCR有一个镜像站点可以做到这一点。Tugay与GCR的SRE团队联系，他们确认该团队可以在Docker配置上设置`--registry-mirror=https://mirror.gcr.io`。Tugay开始设置此功能，并发现镜像站点已经就位！

在上午11:29，Tugay向IRC报告说这些镜像是从GCR镜像站点而不是DockerHub中提取的。上午11:37，事件指挥官宣传呼了GCR。上午11:59，GCR待命，从其欧洲存储层清除了损坏的镜像。到12:11 pm，所有欧洲区域的误差均降至0%。

中断已经结束。剩下的就是清理工作，并写出一部真正的史诗般的事后报告。

在修复之前，CreateCluster在欧洲失败了6个小时40分钟。在IRC中，整个事件中出现了41个独立用户，而IRC日志一直扩展到

26,000个字。这项工作在不同的时间组建了7个IMAG工作队，并且在任何给定时间同时工作多达4个。来自六个团队的值班人员，这还不包括在“紧急集合”呼叫中。事后总结包含28个动作项。

## 回顾

按任何人的标准，GKE CreateCluster中断都是一个大事件。让我们探索进展顺利的地方，以及可以更好地处理的地方。

**进展顺利吗？**该团队有许多记录在案的上报路径，并且熟悉事件响应策略。GKE待命Zara迅速验证了这种影响正在影响实际客户。然后，她使用事先准备好的事件管理系统来引进Rohit，后者将中断情况传达给客户。

**本来可以更好地处理的？**服务本身有一些需要关注的领域。复杂性和对专家的依赖是有问题的。日志记录不足以进行诊断，并且团队因DockerHub的损坏而分心，这并不是真正的问题。

在事件开始时，事件指挥官没有建立正式的事件响应结构。当Zara担任这个角色并将对话转移到IRC时，她本可以在协调信息和制定决策方面更加主动。结果，少数急救人员在没有协调的情况下进行了自己的调查。Il-Seong在首页之后两个小时就建立了正式的事件响应结构。

最后，该事件表明GKE的灾难恢复工作存在差距：该服务没有任何早期的通用缓解措施可以减轻用户的痛苦。通用缓解是指即使在根本原因尚未完全了解之前，急救人员采取的缓解疼痛的措施。例如，当中断与发布周期相关联时，响应者可以回滚最近发布的版本，或者重新配置负载均衡器以避免在本地化错误时出现故障。重要的是要注意，通用缓解措施是钝器，可能会导致其他服务中断。但是，尽管它们可能比精确解决方案具有更广泛的影响，但可以在团队发现并解决根本原因的同时迅速部署它们以止血。

让我们再次查看该事件的时间表，以了解通用缓解措施可能在哪些地方有效：

- **上午7点(评估影响)**。Zara确认用户受到中断的影响。
- **上午9:56(找到可能的原因)**。潘安妮和维多利亚发现了流氓镜像。
- **上午10:59(定制缓解方案)**。几个团队成员致力于重建二进制文件，以推动新的配置，该配置将从另一个位置获取镜像。
- **上午11:59(找到了根本原因并解决了问题)**。Tugay和GCR待命呼叫停用了GCR缓存，并从其欧洲存储层清除了损坏的镜像。

步骤2(发现可能的原因)之后的一般缓解措施在这里非常有用。如果响应者一旦发现问题的大致位置，便已将所有图像回滚到已知的良好状态，则该事件将在上午10点之前缓解。要缓解事件，您不必完全了解详细信息-仅需要知道根本原因的位置。在完全了解故障原因之前具有缓解中断的能力对于运行具有高可用性的强大服务至关重要。

在这种情况下，响应者会从某种有助于回滚的工具中受益。缓解工具确实需要花费工程时间来开发。创建通用缓解工具的正确时间是在事件发生之前，而不是在响应紧急情况时。浏览事后检查是发现缓解措施和/或工具(在回顾中可能有用)并将其构建为服务的一种好方法，以便将来更好地管理事件。

重要的是要记住，第一响应者必须首先将缓解措施放在第一位，否则解决时间会受到影响。采取通用的缓解措施，例如回滚和引流，可以加快恢复速度，并带来更高兴的客户。最终，客户不在乎您是否完全了解造成中断的原因。他们想要的是停止接收错误。

以缓解为重中之重，应对活动中的事件的处理方法如下：

1. 评估事件的影响。
2. 减轻影响。
3. 对事件执行根本原因分析。
4. 事件结束后，请解决导致事件的原因并撰写事后报告。

之后，您可以运行事件响应演练来演练系统中的漏洞，工程师可以在项目上进行工作以解决这些漏洞。

### **案例研究3: 停电---雷电不会击中两次...直到它做到了**

前面的示例显示了当您没有适当的事件响应策略时可能会出问题的地方。下一个示例说明了已成功管理的事件。当您遵循定义明确且清晰的响应协议时，您甚至可以轻松处理罕见或异常事件。

#### **上下文**

诸如雷击之类的电网事件会导致进入数据中心设施的功率发生巨大变化。影响电网的雷击很少见，但并非意外。Google使用备用发电机和备用电池来防止突然的，意外的停电，这些备用发电机和电池经过了充分的测试，可以在这些情况下工作。

Google的许多服务器都附有大量磁盘，这些磁盘位于服务器上方或下方的单独托盘中。这些托盘有自己的不间断电源电池(UPS)。发生断电时，备用发电机将激活，但需要几分钟才能启动。在此期间，连接到服务器和磁盘托架的备用电池将一直供电，直到备用发电机完全运行为止，从而防止电网事件影响数据中心的运行。

#### **事件**

2015年中，闪电在两分钟内四次击中比利时Google数据中心附近的电网。数据中心的备用生成器已激活，可以为所有计算机供电。在备用发电机启动时，大多数服务器用备用电池运行几分钟。

磁盘托架中的UPS电池在第三和第四次雷击时未将电源切换为备用电池电力，因为这些雷击间距太近。结果，磁盘托架断电，直到备用发电机启动为止。服务器没有断电，但无法访问已关闭电源再打开的磁盘。

在永久性磁盘存储上丢失大量磁盘托盘会导致在Google Compute Engine(GCE)上运行的许多虚拟机(VM)实例发生读写错误。这些错误将立即通知持久化磁盘SRE值班团队。一旦Persistent Disk SRE小组确定了影响，便宣布了一个重大事件并向所有受影响的各方宣布。持久化磁盘SRE值班人员承担了突发事件指挥官的角色。

经过初步调查和利益相关者之间的沟通，我们确定：

- 由于暂时断电而丢失磁盘托盘的每台计算机都需要重新启动。
- 在等待重新启动时，某些客户VM在读取和写入磁盘时遇到问题。
- 任何既有磁盘托盘又有客户VM的主机都不能简单地“重新启动”而不丢失没有受到影响的客户VM。持久化磁盘SRE要求GCE SRE将不受影响的VM迁移到其他主机。

持久化磁盘SRE的主要值班人员仍然是IC角色，因为该团队对客户影响具有最佳可见性。

运维团队成员的任务如下：

- 安全地恢复电源，以使用电网电源代替备用发电机。
- 重新启动所有未托管VM的计算机。
- 在持久化磁盘SRE和GCE SRE之间进行协调，以在重新启动VM之前将其安全地移离受影响的计算机。

前两个目标已明确定义，充分理解并记录在案。数据中心运维值班人员立即开始工作，以安全地恢复供电，并向IC提供定期状态报告。持久化磁盘SRE已定义了用于重新启动所有未托管虚拟机的计算机的过程。团队成员开始重新启动这些计算机。

第三个目标比较模糊，任何现有程序都没有涵盖。事件指挥官分配了一个专门的运维团队成员来与GCE SRE和持久化磁盘SRE进行协调。这些团队进行了协作，以安全地将VM从受影响的计算机上移开，以便可以重新启动受影响的计算机。IC密切监控他们的进度，并意识到这项工作要求快速编写新工具。IC组织了更多的工程师向运维团队报告，以便他们可以创建必要的工具。

沟通负责人观察并询问了所有与事件相关的活动，并负责向多个受众报告准确的信息：

- 公司领导者需要有关问题范围的信息，并确保问题已得到解决。
- 担心存储的团队需要知道何时可以再次完全使用他们的存储。
- 需要主动通知外部客户有关此云区域中磁盘的问题。
- 已提交支持凭单的特定客户需要更多有关他们所遇到的问题的信息，以及有关解决方法和时间表的建议。

在减轻最初对客户的影响之后，我们需要进行一些跟进，例如：

- 诊断磁盘托盘使用的UPS失败的原因，并确保不再发生。
- 更换发生故障的数据中心中的电池。
- 手动清除由于同时丢失多个存储系统而导致的“卡死”操作。

事件后的分析表明，只有极少数的写操作(在事件期间断电的机器上未决的写操作)从未写入磁盘。由于持久化磁盘快照和所有Cloud Storage数据都存储在多个数据中心以实现冗余，因此仅丢失了0.000001%的来自运行中的GCE计算机的数据，并且只有来自运行中的实例的数据处于危险之中。

## 回顾

通过尽早宣布事件并组织有明确领导的对策，一组精心管理的人员可以有效地处理这一复杂事件。事件指挥官将恢复电源和重新启动服务器的正常问题委托给适当的操作负责人。工程师致力于解决问题，并将其进度报告给了运维负责人。

要同时满足GCE和持久化磁盘的需求，更复杂的问题需要协调决策和多个团队之间的互动。事件指挥官确保从这两个团队中分配适当的操作团队成员，并直接与他们合作以寻求解决方案。事件指挥官明智地专注于事件的最重要方面：尽快解决受影响客户的需求。

## 案例研究4: PagerDuty的事件响应

由*PagerDuty的Arup Chakrabarti撰写*

几年来，PagerDuty已经开发并完善了我们的内部事件响应实践。最初，我们在公司范围内任命了一名永久性的事件指挥官，并为每种服务配备了专门的工程师来参与事件响应。随着PagerDuty的员工人数增加到400多人和数十个工程团队，我们的事件响应流程也发生了变化。每隔几个月，我们都会仔细检查流程，并对其进行更新以反映业务需求。我们学到的几乎所有内容都记录在<https://responsepagerduty.com>上。我们的事件响应流程故意不是静态的；他们就像我们的业务一样不断变化和发展。

### PagerDuty的重大事件响应

通常，小事件仅需要一名值班工程师来响应。对于较大的事件，我们非常重视团队合作。在高压力和高影响的情况下，工程师不应感到孤独。我们使用以下技术来促进团队合作：

#### 参加模拟练习

我们教团队合作的一种方式是参加失败星期五。PagerDuty从Netflix Simian Army汲取了灵感。最初，“失败星期五”是一个手动失败注入练习，旨在更多地了解我们的系统可能崩溃的方式。今天，我们还使用每周一次的练习来重现生产和事件响应场景中的常见问题。

在“失败星期五”开始之前，我们将任命一名事故指挥官(通常是经过培训成为IC的人员)。在进行故障注入练习时，它们会表现得像真实的IC一样。在整个演练中，主题专家将使用与实际事件相同的过程和语言。这种做法既使新的值班工程师熟悉事件响应语言和流程，又为经验丰富的值班工程师提供了进修课程。

#### 玩限时模拟游戏

尽管“失败星期五”演习对于培训工程师如何使用不同的角色和流程有很大帮助，但他们无法完全复制实际重大事件的紧迫性。我们使用具有时间紧迫性的模拟游戏来捕获事件响应的这一方面。

“没人会被炸掉”是我们充分利用的一款游戏。它要求玩家共同努力在规定的时间内消灭炸弹。游戏的压力和交流密集性迫使玩家进行有效的协作和合作。

#### 从以前的事件中学习

从先前的事件中学习可以帮助我们更好地应对未来的重大事件。为此，我们进行并定期检查事后报告。

PagerDuty的事后处理涉及公开会议和详尽的文档。通过使这些信息易于访问和发现，我们旨在减少解决未来事件的时间，或完全杜绝未来事件的发生。

我们还会记录重大事件中涉及的所有电话，以便我们可以从实时通信提要中学习。

让我们看一下最近的事件，其中PagerDuty必须利用我们的事件响应过程。该事件发生于2017年10月6日，持续了10多个小时，但对客户的影响却很小。

- **7:53 pm** PagerDuty SRE团队的一位成员被警告说，PagerDuty内部NTP服务器出现时钟漂移。值班的SRE验证了所有自动恢复操作均已执行，并完成了相关运行手册中的缓解步骤。SRE团队专用的Slack频道记录了这项工作。

- **晚上8:20** PagerDuty软件小组A的成员收到有关其服务中时钟漂移错误的自动警报。软件团队A和SRE团队致力于解决该问题。
- **9:17 pm** PagerDuty软件团队B的成员收到有关其服务上时钟漂移错误的自动警报。B团队的工程师加入了Slack渠道，在那里该问题已经被分类和调试。
- **9:49 pm** SRE值班宣布发生了重大事件，并提醒了事件指挥官待命。
- **9:55 pm** IC组建了响应团队，其中包括每位提供依赖NTP服务的值班工程师，以及PagerDuty的客户支持值班人员。IC让响应团队加入了专门的电话会议和Slack频道。

在接下来的八个小时中，响应团队致力于解决并缓解该问题。当我们的运行手册中的过程无法解决问题时，响应团队开始有条不紊地尝试新的恢复选项。

在此期间，我们每四个小时轮换一名值班工程师和IC。这样做可以鼓励工程师休息，并将新的想法带入响应团队。

- **5:33 am** 通话中的SRE对NTP服务器进行了配置更改。
- **6:13 am** IC确认所有服务已由其各自的值班工程师恢复。验证完成后，IC将关闭电话会议和Slack通道，并宣布事件已完成。考虑到NTP服务的广泛影响，必须进行事后报告。在结束事件之前，IC将事后分析分配给了SRE团队以进行服务。

### 用于事件响应的工具

我们的事件响应流程利用了三个主要工具：

#### *PagerDuty*

我们将所有通话信息，服务所有权，事后报告，事件元数据等存储在PagerDuty中。这使我们可以在出现问题时迅速组建合适的团队。

#### *Slack*

我们为所有主题专家和突发事件指挥官提供了一个专用渠道(#incident-war-room)作为集会场所。该频道主要用于

作为抄写员的信息分类帐，他记录了操作，所有者和时间戳。

#### 电话会议

当要求加入任何事件响应时，值班工程师需要拨打静态电话会议号码。我们希望所有协调决策都在电话会议中做出，并且决策结果记录在Slack中。我们发现这是做出决定的最快方法。我们还记录每个电话，以确保在抄写员错过重要细节时我们可以重新创建任何时间表。

虽然Slack和电话会议是我们选择的沟通渠道，但您应该使用最适合您的公司及其工程师的沟通方式。

在PagerDuty，我们如何处理事件响应直接关系到公司的成功。我们没有面对未准备好的此类事件，而是通过进行模拟演练，回顾先前的事件并选择合适的工具来有针对性地为事件做准备，以帮助我们抵御可能遇到的任何重大事件。

## 将最佳实践付诸实践

我们已经看到了处理得很好的事件的例子，有些则没有。当呼叫向您发出问题警报时，考虑如何处理该事件为时已晚。开始考虑事件管理过程的时间是在事件发生之前。那么，在灾难来临之前，您如何准备理论并将其付诸实践？本节提供一些建议。

## 事件响应培训

我们强烈建议培训响应者以组织事件，这样他们就可以在真正的紧急情况下采取行动。知道如何组织事件，在整个事件中使用通用语言以及共享相同的期望可以减少发生误解的机会。

完整的“事件指挥系统”方法可能远远超出您的需要，但是您可以通过选择事件管理过程中对组织重要的部分来开发用于处理事件的框架。例如：

- 让值班人员中知道他们可以在事件发生期间委派和升级。
- 鼓励以缓解为先。
- 定义事件指挥官，沟通主管和操作主管角色。

您可以调整和总结事件响应框架，并创建幻灯片组以向新团队成员展示。我们了解到，人们可以将事件响应理论与实际场景和具体行动联系起来，因此更愿意接受事件响应培训。因此，一定要包括动手练习并分享过去事件中发生的事情，分析进展顺利和不顺利的情况。您也可以考虑使用专门从事事件响应课程和培训的外部机构。

### 事前准备

除了事件响应培训之外，它还有助于事前准备事件。使用以下提示和策略可以更好地做好准备。

#### 决定沟通渠道

事先确定并同意通信渠道(Slack，电话桥，IRC，HipChat等)---事件指挥官不想在事件发生时做出此决定。练习使用它，所以不会感到惊讶。如果可能，选择团队已经熟悉的沟通渠道，以便团队中的每个人都可以轻松使用它。

#### 让听众知情

除非您确认事件正在发生并得到积极解决，否则人们会自动认为没有任何事情可以解决。同样，如果您在问题得到缓解或解决后忘记取消呼叫响应，则人们会认为事件仍在继续。您可以通过定期更新状态来使事件发生期间的听众保持知情，从而避免这种动态变化。准备好联系人列表(请参阅下一个技巧)可以节省宝贵的时间，并确保您不会错过任何人。

请提前考虑如何起草，审阅，批准和发布公共博客文章或新闻稿。在Google，团队寻求公关团队的指导。另外，准备两个或三个可随时使用的模板以共享信息，以确保通话中知道如何发送它们。没有人愿意在没有指导的情况下承受巨大压力。模板使与公众共享信息变得容易且压力最小。

#### 准备联系人列表

事先准备好要发送电子邮件或页面的人员列表，可以节省关键的时间和精力。在“案例研究2：服务错误--如果可以，请给我缓存”，第180页上的沟通Lead通过向预先准备的几个GKE列表发送电子邮件来进行“紧急集合”的呼叫。

### 建立事件准则

有时候，很明显呼叫问题确实是一个事件。有时还不清楚。拥有确定的标准列表来确定问题是否确实是有帮助的。团队可以通过查看过去的中断情况，并考虑已知的高风险区域，得出可靠的标准列表。

总之，在应对事件时建立协调和沟通的共同基础很重要。确定沟通事件的方式，受众是谁，以及谁对事件负责。这些准则易于设置，并且对缩短事件的解决时间有很大影响。

### 操练

事件管理过程的最后一步是练习事件管理技能。通过在不太紧急的情况下进行练习，您的团队会在雷击时养成良好的习惯和行为方式，无论是从形象上还是从字面上看。通过培训介绍事件响应理论之后，实践可确保您的事件响应技能保持最新状态。

有几种方法可以进行事件管理演练。Google在全公司范围内进行了弹性测试(称为灾难恢复测试或DiRT；请参阅Kripa Krishnan的文章"应对意外情况"[66](#))，在该测试中，我们创建了可控制的紧急情况，实际上并未影响客户。团队对受控紧急情况的响应就像是真正的紧急情况一样。之后，团队将审查应急程序并讨论发生了什么。接受失败作为学习的一种方法，在已发现的差距中寻找价值，并在我们的领导层中发挥领导作用，对于在Google成功建立DiRT计划至关重要。在较小的规模上，我们使用诸如轮盘赌(Wheel of Fortune)之类的练习来练习对特定事件做出响应(请参阅《站点可靠性工程》中的"灾难角色扮演")。

您还可以通过有意将小问题视为需要大规模响应的重大问题来练习事件响应。这样一来，您的团队就可以在较低风险的实际情况下使用过程和工具进行练习。

演习是尝试新的事件响应技能的一种友好方式。团队中任何可以卷入事件响应的人-SRE，开发人员，甚至客户支持和营销合作伙伴-都应该对这些策略感到满意。

要进行演习，您可以制造中断，并让您的团队对事件进行响应。您还可以从事后总结中创建中断，其中包含事件管理演练的大量想法。尽可能使用真实的工具来管理事件。考虑破坏您的测试环境，以便团队可以使用现有工具进行真正的故障排除。

如果定期运行所有这些演练，它们将更加有用。您可以通过跟进每个练习并附上一份报告来使演习具有影响力，该报告详细说明了哪些方面进展顺利，哪些方面进展不好以及如何更好地处理问题。进行演习最有价值的部分是检查其结果，这可以揭示出事件管理方面的许多空白。一旦知道它们是什么，就可以努力将其关闭。

### 结论

为灾难袭来做好准备。如果您的团队定期练习并刷新事件响应程序，那么在不可避免的中断发生时，您就不会惊慌。

事件期间您需要与之合作的人员圈子随着事件的规模而扩大。当您与不认识的人一起工作时，过程可帮助您创建快速迈向解决方案所需的结构。我们强烈建议在世界末"着火"时提前建立这些程序。定期检查并迭代事件管理计划和手册。

事件指挥系统是一个易于理解的简单概念。它根据公司的规模和事件的规模来扩大或缩小。尽管很容易理解，但实施起来并不容易，尤其是在紧急情况突然超过您的情况下。在紧急情况下保持镇静并遵循响应结构需要练习，练习可以建立“肌肉记忆”。这使您有信心在真正的紧急情况下需要。

我们强烈建议您在团队繁忙的时间中安排一些时间来定期练习事件管理。在专门的练习时间获得领导的支持，并确保他们了解事件响应的工作方式，以防您需要让他们参与实际事件。备灾可以节省宝贵的几分钟或几小时的响应时间，并为您提供竞争优势。没有公司能一直做到正确---从错误中吸取教训，继续前进，并在下一次做得更好。

65. 当三个或三个以上的人员处理一个事件时，启动一个协作文档很有用，该文档列出了工作原理，消除的原因以及有用的调试信息，例如错误日志和可疑图形。该文档会保留此信息，因此不会在对话中迷路。 ↵

66. 克里帕·克里山(Kripa Krishan)，"出乎意料的风雨"，ACM通讯 10，否。  
9(2012)，<https://queue.acm.org/detail.cfm?id=2371516>。 ↵

# 第10章

## 事后文化: 从失败中学习

由丹尼尔·罗杰斯(Daniel Rogers) , 穆拉里·苏里尔(Murali Suriar) , 苏·路德(Sue Lueder)

Pranjal Deo和Divya Sudhakar以及Gary O'Connor和Dave Rensin撰写

我们的经验表明，真正对事不对人的事后总结文化可以产生更可靠的系统-这就是为什么我们认为这种做法对创建和维护成功的SRE组织很重要。

将事后总结引入组织既是一种文化变革，又是技术变革。做出这样的转变似乎令人生畏。本章的主要结论是，进行此更改是可能的，并且似乎不是一个无法克服的挑战。不要因为希望您的系统最终能够自我修复而发生事故。您可以通过引入一个非常基本的事后总结程序开始，然后反思和调整您的过程以使其最适合您的组织-从很多方面来说，没有一种方法可以适合所有情况。

如果编写得当，采取行动并广为分享，则事后总结可以是驱动积极的组织变革并防止重复停机的非常有效的工具。为了说明良好的事后撰写原则，本章介绍了有关Google发生实际停机的案例研究。事后总结笔迹不佳的一个例子突出了"不良"事后总结习惯对试图创建健康的事后总结文化的组织造成损害的原因。然后，我们将不良的事后总结与事件发生后写的真实的事后总结进行比较，突出显示高质量事后总结的原理和最佳实践。

本章的第二部分分享了我们所学到的有关创建激励机制来培育强大的事后总结文化的知识，以及如何识别(和补救)该文化正在崩溃的早期迹象。

最后，我们提供了可用于引导事后文化的工具和模板。

有关对事不对人的事后总结哲学的全面讨论，请参阅第一本书*Site Reliability Engineering*中的[第15章](#)。

## 案例分析

此案例研究具有例行的机架退役功能，导致我们的用户的服务延迟增加。我们维护自动化中的一个错误，再加上速率限制不足，导致承载生产流量的数千台服务器同时脱机。

虽然Google的大多数服务器都位于我们专有的数据中心中，但我们在托管设备(或"托管服务器")中也有许多代理/缓存计算机。包含我们的代理服务器的colos机架被称为卫星。由于卫星需要定期维护和升级，因此在任何时间点都会安装或停用许多卫星机架。在Google，这些维护流程在很大程度上是自动化的。

停用过程使用称为"磁盘擦除\*"的过程覆盖机架中所有驱动器的全部内容。一旦将机器发送到磁盘擦除，它曾经存储的数据将不再可检索。典型的机架停用步骤如下：

```
# Get all active machines in "satellite"
machines = GetMachines(satellite)

# Send all candidate machines matching "filter" to decom
SendToDecom(candidates=GetAllSatelliteMachines(),
            filter=machines)
```

我们的案例研究从标记为退役的卫星机架开始。停用过程的磁盘擦除步骤已成功完成，但是负责其余机器停用的自动化操作却失败了。为了调试故障，我们重试了退役过程。第二次停用运行为如下：

```
# Get all active machines in "satellite"
machines = GetMachines(satellite)

# "machines" is an empty list, because the decom flow has already run.

# API bug: an empty list is treated as "no filter", rather than "act on no # machines"
# Send all candidate machines matching "filter" to decom
SendToDecom(candidates=GetAllSatelliteMachines(),
            filter=machines)

# Send all machines in "candidates" to diskerase.
```

几分钟之内，全球所有卫星机器的磁盘都被擦除。这些机器呈惰性状态，无法再接受用户的连接，因此后续的用户连接被直接路由到我们的数据中心。结果，用户的等待时间略有增加。由于进行了良好的容量规划，在为时两天的时间内，我们很少有用户注意到此问题，因此我们在受影响的colo机架中重新安装了机器。事件发生后，我们花费了数周的时间进行审核，并在自动化系统中添加了更多的健全性检查，以使退役工作流成为幂等。

断电三年后，我们遇到了类似的事件：大量卫星被排空，导致用户延迟增加。从最初的事后总结中实施的行动项目极大地降低了冲击半径和第二次事件的发生率。

假设您是负责编写此案例研究的事后分析的人。您想知道什么，以及您打算采取什么行动来防止再次发生这种中断？

让我们从这次事后总结还不算大的问题开始。

## 不良事后总结

## 事后总结: 所有发送给卫星机器磁盘擦除

2014年8月11日

**所有者:** maxone@ , logantwo@ , sydneythree@ , dylanfour@ **与以下人员共享:** Satellite-infra-team@ **状态:** 终结 **事件发生日期:** 2014年8月11日 **发布日期:** 2014年12月30日

### 执行摘要

**影响:** 所有的卫星机器都被发送到磁盘擦除，而后者实际上清除了Google Edge。 **根本原因:** dylanfour@忽略了自动化设置，并手动运行了群集启动逻辑，从而触发了一个现有的错误。

### 问题摘要

**问题持续时间:** 40分钟 **受影响的产品:** 卫星基础设施团队

**受影响的产品百分比:** 所有卫星集群。

**用户影响:** 正常情况下，所有向卫星发送的查询都由核心提供，导致延迟增加。**收益影响:** 由于丢失了查询，因此无法投放某些广告。目前尚不清楚确切的收入影响。**侦测:** 监控警报。**解决方案:** 将流量转移到核心，然后手动修复边缘群集。

### 背景(可选)

#### 影响

##### 对用户的影响

通常会从核心服务所有通常用于卫星的查询，这会增加用户流量的延迟。

##### 收入影响

由于查询丢失，某些广告无法投放。

### 根本原因和触发因素

集群开启/关闭自动化并不是幂等的。该工具具有安全措施，可确保某些步骤不能多次运行。不幸的是，没有什么可以阻止某人手动运行他们想要的次数。没有文档提到此陷阱。结果，大多数团队成员认为，如果该过程不起作用，则可以多次运行该过程。

这就是在常规停用机架期间发生的事情。机架已被新的基于Iota的卫星所取代。dylanfour@完全忽略了该周转已经执行一次并被卡在第一次尝试中的事实。由于粗心大意的无知，他们引发了糟糕的互动，将所有卫星机器分配给了磁盘擦除团队。

### 恢复工作

#### 经验教训

##### 正常运行

- 警报立即发现了问题。
- 事件管理进展顺利。

**非正常运行**

- 团队(尤其是maxone@，logantwo@)从未编写任何文档来告诉SRE不要多次运行自动化，这很荒谬。
- 值班人员中行动不足以阻止大多数卫星机器被删除。这不是值班中未能及时做出反应的第一次。

**幸运的地方**

- Core能够处理通常会到达Edge的所有流量。我简直不敢相信我们幸免于难！！！

**动作项**

动作项	类型	优先级	所有者	跟踪错误
使自动化更好。	缓解	P2	logantwo@	
改进呼叫和警报	发现	P2		
sydneythree@需要学习正确的跨站点切换协议，因此没有人需要处理重复的问题。	缓解	P2		BUG6789
培训人员不要执行不安全的命令。	预防	P2		

**词汇表****为什么此事后分析不好？**

示例“不良”事后总结包含许多我们试图避免的常见故障模式。以下各节说明如何改进此事后总结。

**缺少上下文**

从一开始，我们的示例事后总结介绍了专门针对流量服务的术语(例如，“卫星”)和Google较低级别的机器管理自动化(例如，“磁盘擦除”)。如果您需要在事后提供其他上下文，请使用“背景”和/或“词汇表”部分(可以链接到更长的文档)。在这个实例下，两个部分均为空白。

如果您在撰写事后总结报告时没有正确地对内容进行上下文文化，则文档可能会被误解甚至被忽略。重要的是要记住，您的听众不仅限于直属团队。

**省略关键细节**

多个部分包含高级摘要，但缺少重要的细节。例如：

**问题摘要**

对于影响多种服务的中断，您应该提供数字以一致地表示影响。我们的示例提供的唯一数值数据是问题的持续时间。没有足够的细节来估计中断的规模或影响。即使没有具体数据，一个有根据的估计也比没有数据要好。毕竟，如果您不知道如何测量它，那么您将不知道它是被修复的！

#### 根本原因和触发因素

查明根本原因和触发因素是撰写事后总结报告的最重要原因之一。我们的示例包含一小段描述了根本原因和触发因素，但是并未探讨该问题的底层细节。

#### 恢复工作

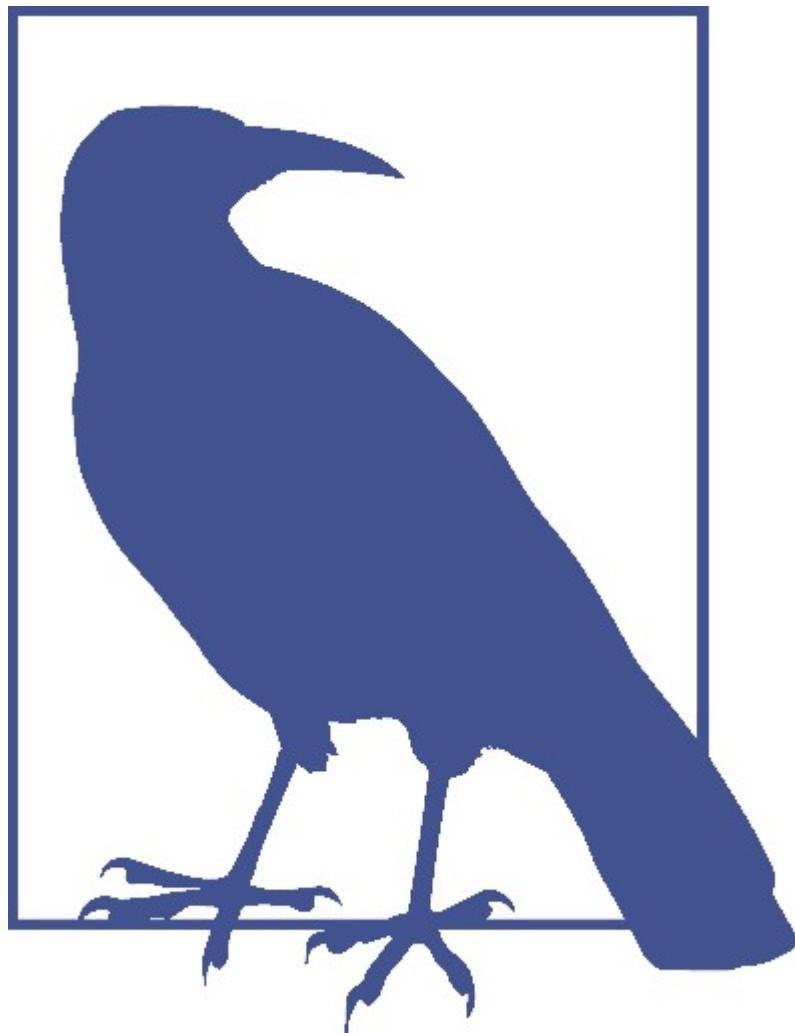
事后记录为其读者记录事件。一个好的事后总结报告将使读者知道发生了什么事情，如何缓解该问题以及如何影响用户。这些问题的答案通常可以在"Recovery Efforts"部分找到，在我们的示例中为空。

如果停机需要进行事后分析，则您还应该花时间准确捕获并记录必要的详细信息。读者应全面了解停机情况，更重要的是，要学习新的知识。

#### 缺少关键行动项目特征

我们的示例的"操作项"部分缺少可操作的计划以防止再次发生的核心方面。例如：

- 操作项大多是缓解措施。为了最大程度地减少停电再次发生的可能性，您应该包括一些预防措施和解决方案。一个"预防"行动建议表明我们"使人类减少了容易出错的事件"。通常，尝试更改人类行为并不如更改自动化系统和流程可靠。(或者像丹·米尔斯坦[曾经打过电话](#)："就像我们今天很愚蠢一样来为我们未来制定计划。")
- 所有动作项均具有与之相等的优先级标记。无法确定首先要采取的行动。
- 列表中的前两个操作项使用了模棱两可的词组，例如"改进"和"变得更好"。这些术语含糊且不易于解释。使用不清楚的语言会使衡量和理解成功标准变得困难。
- 仅向一个操作项分配了跟踪错误。如果没有正式的跟踪流程，事后总结后的行动项目通常会被遗忘，从而导致停运。



用Google 24/7运维副总裁Ben Treynor Sloss的话来说：“对于我们的用户，没有后续行动的事后总结与没有事后总结是一样的。因此，在影响用户的停机之后进行的所有事后总结必须至少具有一个与之关联的P[01]错误。我亲自审查例外情况。很少有例外。”

### 适得其反的指责

每个事后剖析都有可能陷入怪异的叙述中。让我们看一些例子：

#### 情况不佳

当两名成员被征召的时候(maxone@和logantwo@),整个团队都应该因为这个故障受到谴责。

#### 行动项目

列表中的最后一项使sydneythree@被压力压垮，对跨站点切换疏于管理。

#### 根本原因和触发因素

dylanfour@对中断负责。

在事后总结中突出个人似乎是一个好主意。相反，这种做法会导致团队成员规避风险，因为他们害怕被公众羞辱。他们可能有动机掩盖对理解和防止再次发生至关重要的事实。

### **情绪化的语言**

事后总结是一种事实物品，应该没有个人判断和主观语言。它应该考虑多种观点，并尊重他人。我们的示例事后分析包含多个不良语言示例：

#### **根本原因和触发因素**

多余的语言(例如"粗心的无知")

#### **情况不佳**

情绪化的文字(例如，"荒谬")

#### **幸运的地方**

令人难以置信的感叹(例如，"我不敢相信我们能幸免于难！")

情绪化的语言和事件的戏剧性描述分散了关键信息的注意力，削弱了心理安全性。取而代之的，要提供可验证的数据来证明语句的严重性。

#### **缺少所有权**

宣布指挥此行动的官方所有权的所有者用来建立问责机制。我们的示例事后总结包含缺少所有权的几个示例：

- 事后总结列出了四个所有者。理想情况下，所有者是负责事后检查，跟进和完成的单一联系人。
- "操作项"部分的条目几乎没有所有权。没有明确所有者的操作项目不太可能被分辨。

最好有一个所有者和多个合作者。

#### **限量受众**

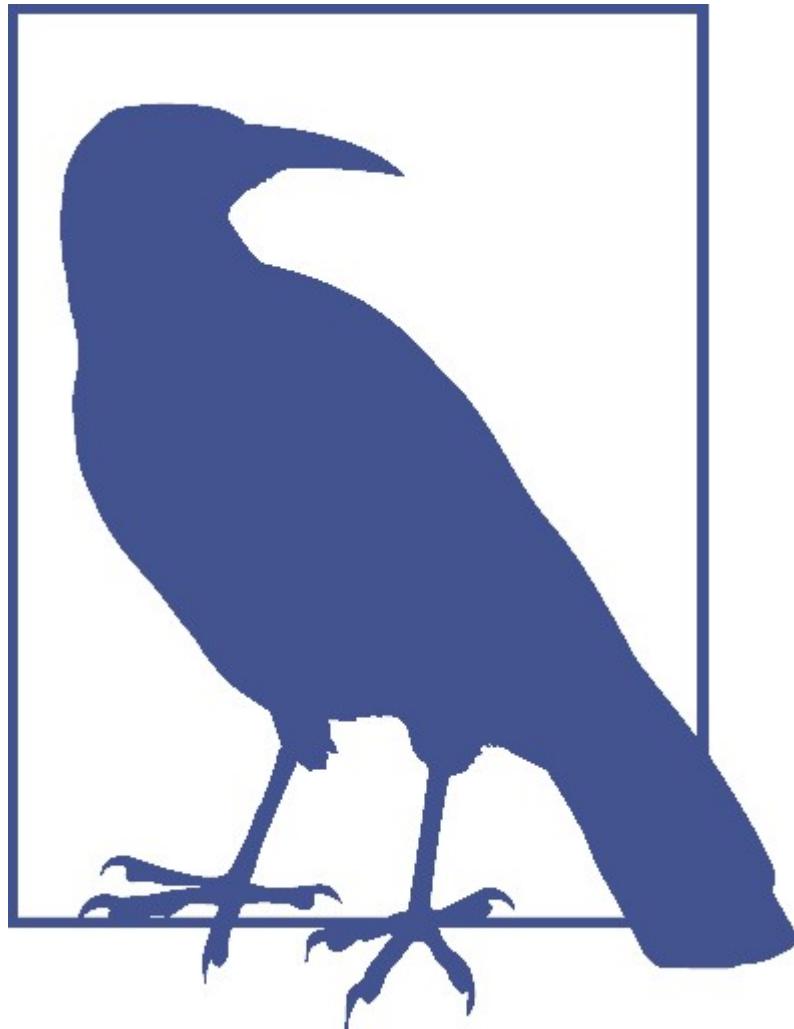
我们的示例事后总结仅在团队成员之间共享。默认情况下，公司中的每个人都应该可以访问该文档。我们建议您尽可能广泛地主动分享您的事后信息，甚至与您的客户分享。事后总结的价值与其创造的学习成正比。从过去的事情中学到的东西越多，重复发生的可能性就越小。进行深思熟虑的诚实事后总结也是恢复动摇的信任的关键工具。

随着经验和舒适度的提高，您也有可能将"受众"扩展到非人类。成熟的事后总结文化通常会添加机器可读的标签(和其他元数据)以启用下游分析。

#### **延迟发表**

我们的事后总结示例在事件发生四个月后才发表。在此期间，如果事件再次发生(实际上确实发生了)，团队成员可能会忘记及时的事后调查本应捕获的关键细节。

## **好的事后总结**



这是实

际的事后总结。在某些情况下，我们将个人和团队的名称虚构。我们还用占位符替换了实际值，以保护敏感的容量信息。在为内部消费创建的事后总结中，绝对应包括特定数字！

### 事后总结: 已将所有卫星机器发送给磁盘擦除

2014年8月11日

#### 所有者:

事后总结:maxone@ , logantwo@ 数据中心自动化:sydneythree@ 网络:dylanfour@ 服务器管理:fifive@

分享: [all\\_engineering\\_employees@google.com](mailto:all_engineering_employees@google.com)

状态: 最终

事件发生日期: 2014年8月11日，星期一，17:10至17:50 PST8PDT

时间: 2014年8月15日，星期五

#### 执行摘要

**影响:** 前端查询下降 一些广告未投放 将近两天通常由卫星提供的所有服务的延迟都增加了

**根本原因:** 一个停止自动化并会导致所有卫星计算机被擦除的错误，相反只有一机架卫星计算机并被擦除。这导致所有卫星计算机进入分解工作流程，从而擦除了磁盘。结果是全球卫星前端中断。

#### 问题摘要

**问题持续时间:** 主要中断: PST8PDT，8月11日星期一，17:10至17:50, 通过 PST8PDT，8月13日，星期三，进行重建工作和残留的痛苦，然后事件被关闭。

**受影响产品:** 前端基础结构，尤其是所有卫星位置。

**受影响产品百分比:** 全球-通常由卫星提供的所有流量(通常占全球查询的 60%)。

**用户影响:** [可编辑的值]前端查询在40分钟的时间内下降([可编辑的值]这段时间内的平均QPS，占总流量的[可编辑的值]%)。通常由卫星提供的所有服务的延迟都会增加近两天。

**收益影响:** 目前尚不清楚确切的收入影响。 **检测:** 黑盒警报:世界上的每颗卫星都向"流量小组"报告"satellite a12bcd34 failing too many HTTP

requests"。 **解决方案:** 通过将Google的所有存在额外用户流量的延迟开销的前端流量移至核心群集，可以快速缓解故障本身.

#### 背景(可选)

如果您不熟悉Google的前端流量服务和较低层的服务自动化，请先阅读词汇表，然后再继续。

#### 影响

对用户的影响

- [可编辑的值]前端查询在[可编辑的值]分钟内删除。[可编辑的值]此期间的QPS平均值，占全球流量的[可编辑的值]%。我们的监测表明，火山口更大。但是，由于停止监控仍在运行的卫星，并认为它们已关闭，因此数据不可靠。附录描述了如何估算上述数字。

- 将近两天通常由卫星提供的所有服务的延迟都增加了:
  - [可编辑的值] ms RTT峰值，用于靠近核心集群的国家
  - 对于更严重依赖卫星的位置(例如，澳大利亚，新西兰，印度)，最多+[可编辑的值] ms

#### 收入影响

由于查询丢失，某些广告无法投放。目前尚不清楚确切的收入影响:

- 显示和视频:由于每天的波动，数据具有非常宽的误差线，但我们估计停运当天收入损失的[可编辑的值]%和[可编辑的值]%之间。
- 搜索:[可编辑的值]%到[可编辑的值]%的损失在17:00到18:00之间，同样带有宽误差条。

#### 团队影响

- 沟通小组花了大约48小时的时间，全力以赴地重建卫星。
- NST的中断/值班负载高于正常水平，因为它们需要对过载的对等链接进行流量工程设计。
- 由于GFE中的缓存命中率降低，某些服务可能在其前端提供了更多的响应。
  - 例如，关于[与缓存相关的服务]，请参见此思路[link]。
  - [依赖于缓存的服务]看到它们在GFE上的缓存命中率从[可编辑的值]%下降到[可编辑的值]%，然后才慢慢恢复。

#### 事件文档

[指向我们的事件跟踪文档的链接已被编辑。]

#### 根本原因和触发因素

手动重新执行工作流程以停用a12bcd34卫星，触发了Traffic Admin服务器中长期存在的输入验证错误。该错误消除了对decom动作的机器限制，从而使所有卫星机器退役。

从那里开始，数据中心自动化执行了decom工作流程，在停止该操作之前先擦除大多数卫星计算机的硬盘驱动器。

Traffic Admin服务器提供ReleaseSatelliteMachines RPC。该处理程序使用三个MDB API调用启动卫星退役:

- 查找与边缘节点关联的机架名称(例如a12bcd34 -> \)。
- 查找与机架关联的机器名称(\ -> \, \等)。
- 将这些计算机重新分配给磁盘擦除，从而间接触发停用工作流程。

由于MDB API的已知行为以及缺少安全性检查，因此该过程不是幂等的。如果先前已成功将卫星节点成功发送给decom，则上面的步骤2将返回一个空列表，该列表在步骤3中被解释为对机器主机名没有约束。

这种危险的行为已经存在了一段时间，但是被调用不安全操作的工作流所掩盖:调用RPC的工作流步骤被标记为"运行一次"，这意味着一旦成功，工作流引擎将不会重新执行RPC。。

但是，"运行一次"语义不适用于工作流的多个实例。当"集群启动"团队手动启动a12bcd34的工作流的另一次运行时，此操作触发了admin\_server错误。

### 时间表/恢复工作

[因为图书的出版,指向我们的时间轴日志的链接已删除。在真实的事后总结中，始终会包含此信息。]

### 经验教训

#### 表现良好的

- 疏散边缘。核心中的GFE进行了明确的容量规划，以允许这种情况发生，生产主干网也是如此(除了对等链接；请参阅下一节的停机列表)。这种边缘疏散使沟通团队得以迅速缓解而无所畏惧。
- 自动减轻灾难性卫星故障。备份路线会自动将流量从发生故障的卫星拉回到核心集群，并且在检测到异常流失时，卫星会自动引流。
- 卫星decom/磁盘擦除的运作非常有效且迅速，尽管它是糊涂的代理人。
- 中断通过OMG触发了IMAG快速响应，并且该工具被证明可用于持续的事件跟踪。跨团队的响应非常出色，OMG进一步帮助保持了彼此之间的交流。

#### 表现不佳的事情

#### 停机

- Traffic Admin服务器缺少对发送到MDB的命令的适当检查。所有命令都应该是幂等的，或者至少在重复调用时具有故障保护功能。
- MDB没有反对所有权更改请求中缺少主机名约束。
- 分解工作流程不会与其他数据源(例如计划中的机架分解)交叉检查分解请求。结果，没有人反对丢弃(许多)地理上不同的机器的请求。
- Decom工作流程不受速率限制。机器进入反解析后，磁盘擦除和其他反分解步骤将以最大速度进行。
- 当卫星停止服务时，出口流量转移到其他位置，导致Google与世界之间的某些对等链接超载，而其查询却是从核心服务的。这导致拥挤的短暂爆发，直到选择了对等方，直到恢复卫星为止，NST的缓解工作也得以实现。

#### 恢复

- 卫星机器的重新安装缓慢且不可靠。重新安装使用TFTP传输数据，当在高延迟链路末端将数据传输到卫星时，该数据将无法正常工作。
- 停电时，Autoreplacer基础设施无法处理GFE的同时安装。要使自动设置的速度与之匹敌，需要并行执行手动设置的许多SRE的工作量。以下因素导致了自动化初期的缓慢:

--- 过于严格的SSH超时阻止了在非常远程的卫星上可靠的Autoreplacer操作。

--- 无论计算机是否具有正确的版本，都会执行缓慢的内核升级过程。

--- Autoreplacer中的并发恢复阻止了每台工作计算机运行两个以上的计算机设置任务。

--- 关于Autoreplacer行为的困惑浪费了时间和精力。

- 删除23%的目标时，不会触发监控配置增量安全检查(更改25%)，但是当读取相同的内容(剩余量的29%)时，触发监控配置增量安全检查。这导致重新启用对卫星的监控要延迟30分钟。
- "安装程序"的人员有限。结果，进行更改既困难又缓慢。
- 使用超级用户的权限将机器从磁盘擦除收回，从而留下了很多僵尸状态，从而导致持续的清理痛苦。

#### 我们幸运的地方

- 核心集群中的GFE的管理与卫星GFE的管理非常不同。结果，它们不受横冲直撞的分解的影响。
- 同样，YouTube的CDN作为不同的基础架构运行，因此YouTube视频投放不受影响。如果此操作失败，则中断将更加严重且持续时间更长。

#### 动作项

由于此事件的广泛性，我们将行动项目分为五个主题:

1. 预防/风险教育
2. 紧急应变
3. 监控/报警
4. 卫星/边缘配置
5. 清理/杂项

表10-1。预防/风险教育

动作项	类型	优先级	所有者	跟踪错误
审核所有能够将实时服务器转换为废机箱的系统(即，不仅是维修和磁盘擦除工作流程)。	调查	P1	sydneythree@	BUG1234
文件记录错误，以跟踪对BUG1234中标识的所有系统的错误输入拒绝的实现。	预防	P1	sydneythree@	BUG1235
禁止任何单个操作影响跨越namespace/class边界的服务器。	减轻	P1	maxone@	BUG1236
流量管理服务器需要进行安全检查，以确保不能在超过[可编辑的值]个节点上进行操作。	减轻	P1	dylanfour@	BUG1237
流量管理服务器应询问<安全检查服务>批准破坏性的工作。	预防	P0	logantwo@	BUG1238
MDB应该拒绝不提供预期约束条件值的操作。	预防	P0	louseven@	BUG1239

表10-2。紧急响应

动作项	类型	优先级	所有者	跟踪错误
确保从核心服务不会使出口网络链接过载。	维修	P2	rileysix@	BUG1240
确保在[紧急链接文档的链接已被删除]和[升级联系页面的链接已被删除]下注明了decom工作流问题。	减轻	P2	logantwo@	BUG1241
添加一个大红色按钮^a禁用分解工作流的方法。	减轻	P0	maxone@	BUG1242

<sup>a</sup>a 灾难性情况下使用的关闭开关(例如，紧急关闭按钮)的总称，以避免进一步的损坏。

表10-3。监控/警报

动作项	类型	优先级	所有者	跟踪错误
监控目标安全检查不应允许您推送无法回滚的更改。	缓解	P2	dylanfour@	BUG1243
当我们的机器中有超过[可编辑值]%的机器被影响时，添加警报。机器是在16:38从卫星上取走的，而世界仅在17:10左右才开始呼叫。	发现	P1	rileysix@	BUG1244

表10-4。卫星/边缘配置

动作项	类型	优先级	所有者	跟踪错误
使用iPXE使用HTTPS使重新安装更加可靠/更快。	缓解	P2	dylanfour@	BUG1245

表10-5。清理/杂项

动作项	类型	优先级	所有者	跟踪错误
在我们的工具中查看与MDB相关的代码，并用管理服务器备份恢复正常。	维修	P2	rileysix@	BUG1246
安排DiRT测试: - 磁盘擦除后回归卫星。 - 对YouTube CDN执行相同操作。	缓解	P2	louseven@	BUG1247

&gt;

## 词汇表

### 管理服务器

RPC服务器，该服务器使自动化能够为前端服务基础结构执行特权操作。自动化服务器最明显地参与了PCR和群集开/关的实施。

### 自动替换

一种将不在Borg管理的服务器在计算机之间移动的系统。它用于在遇到机器故障时保持服务运行，还用于支持forklifts和colo重新配置。

### Borg

集群管理系统，旨在大规模管理任务和机器资源。Borg拥有Borg单元中的所有计算机，并将任务分配给具有可用资源的计算机。

### 分解(Decom)

分解(decommissioning)的缩写。设备分解是与许多运营团队相关的过程。

### 磁盘擦除

在生产硬盘离开Google数据中心之前安全擦除其过程(以及相关的硬件/软件系统)。磁盘擦除是decom工作流程中的一个步骤。

### GFE(Google 前端)

外部连接到的服务器(几乎)用于所有Google服务。

### IMAG(Google 事件管理)

一个程序，该程序建立了一种标准的，一致的方式来处理所有类型的事件-从系统中断到自然灾害-并组织有效的响应。

### MDB(机器数据库)

一种存储有关Google机器清单状态的各种信息的系统。

### OMG(Google 的停机管理)

事件管理仪表板/工具，可作为集中位置来跟踪和管理Google上所有正在进行的事件。

### 卫星

小型廉价的机器机架，仅可服务来自Google网络边缘的非视频前端流量。卫星几乎没有传统的生产集群基础设施。卫星与CDN截然不同，CDN从Google的网络边缘以及更广泛的互联网上的其他地方提供YouTube视频内容。YouTube CDN不受此事件影响。

### 附录

#### 为什么 释放卫星系统服务器 不幂等？

[对这个问题的回答已被忽略。]

#### 在Admin Server将所有卫星分配给磁盘擦除团队之后发生了什么？

[对这个问题的回答已被忽略。]

中断期间真正的QPS损失是什么？

[对这个问题的回答已被忽略。]

IRC日志

[IRC日志已被删除。]

图表

更快的延迟统计---卫星为我们做了什么？

从这次中断的经验来看，卫星使核心集群附近的许多位置的[可编辑值] ms延迟减少了，并且距骨干网的位置减少了[可编辑值] ms延迟：

[省略了图形的解释。]

核心与边缘服务负载

很好地说明了重建工作。能够再次从50%的边缘流量提供服务大约需要36个小时，而恢复正常流量平衡则需要额外的12个小时(请参见图10-1和图10-2)。

流量变化带来的压力

[图表消失。]

该图显示了按网络区域汇总的数据包丢失情况。事件本身出现了一些短暂的峰值，但是损失的大部分是由于各个地区进入高峰期而几乎没有/完全没有卫星覆盖而发生的。

《人与机器》，GFE版

[省去了对人机对自动机器设置率的图示说明。]

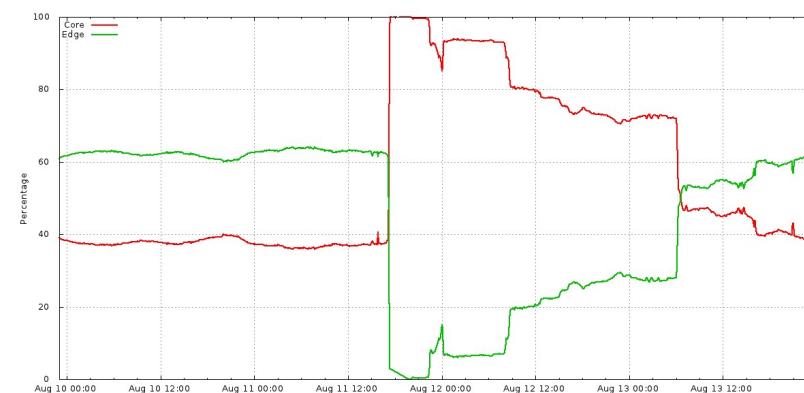


图10-1。核心与边缘QPS细分

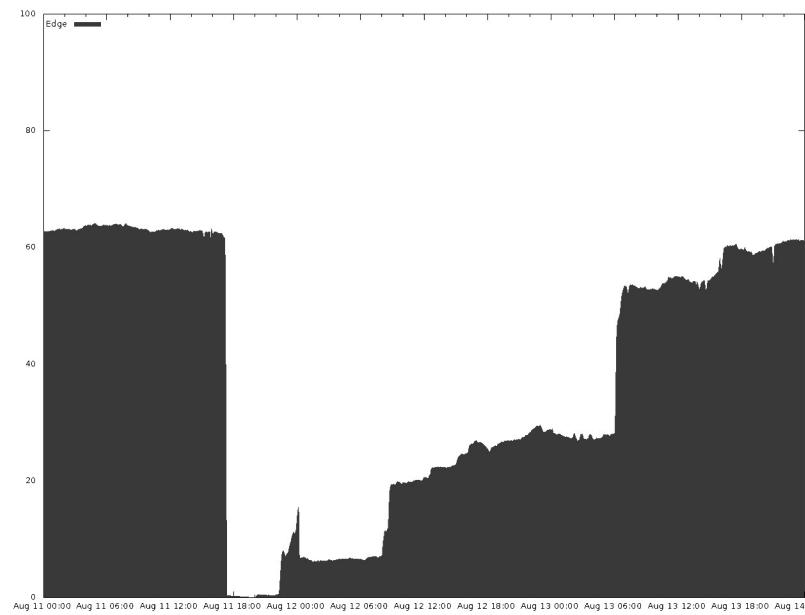


图10-2

## 为什么这个事后总结更好？

此事后总结示例了几种良好的写作习惯。

### 明晰

事后组织良好，并详细解释了关键术语。例如：

### 词汇表

精心编写的词汇表使广泛的读者可以方便地理解和理解事后记录。

### 行动项目

这是一个重大事件，涉及许多行动项目。按主题对操作项进行分组可以更轻松地分配所有者和优先级。

### 量化指标

事后分析会显示有关事件的有用数据，例如缓存命中率，流量级别和影响持续时间。数据的相关部分带有指向原始源的链接。这种数据透明性消除了歧义，并为读者提供了上下文。

### 具体行动项目

没有操作项的事后总结是无效的。这些操作项具有一些显着特征：

### 所有权

所有操作项都有所有者和跟踪号。

### 优先级

为所有操作项目分配了优先级。

### 可衡量性

操作项具有可验证的最终状态(例如，"当我们从我们手中夺走X%以上的机器时，添加警报")。

### 预防措施

每个操作项目"主题"都有"预防/减轻"操作项目，这些操作项目有助于避免中断再次发生(例如，"禁止任何单个操作影响跨越名称namespace/class边界的服务器")。

### 无瑕疵

作者专注于系统设计的空白，这些空白允许出现不希望的故障模式。例如：

#### 情况不佳

没有人或团队对此事件负责。

#### 根本原因和触发

关注"什么"出了问题，而不是"谁"引起了事件。

#### 动作项

旨在改善系统，而不是改善人员。

### 深度

事后调查不仅研究了系统故障的最主要方面，还探索了多个团队的影响和系统缺陷。特别说明：

#### 影响

本节从各个角度包含许多细节，使之具有平衡性和客观性。

#### 根本原因和触发

本节深入分析该事件，并找出根本原因和触发因素。

#### 以数据为依据的结论

提出的所有结论均基于事实和数据。用于得出结论的任何数据均与文档链接。

#### 其他资源

这些以图表的形式提供了更多有用的信息。解释了图以向不熟悉系统的读者提供上下文。

### 及时性

事件发生后不到一周的时间，便已撰写并分发了事后总结报告。即时事后总结往往更准确，因为信息在贡献者的脑海中是新鲜的。受故障影响的人们正在等待解释和证明您的事情得到控制的证明。您等待的时间越长，他们将用他们的想象力填补更多空白。这很少能帮到您！

### 简明

该事件是全球性事件，影响了多个系统。结果，事后记录并随后解析了许多数据。提取了冗长的数据源，例如聊天记录和系统日志，并从主文档链接了未编辑的版本。总体而言，事后分析在详细程度和可读性之间取得了平衡。

## 组织奖励

理想情况下，高层领导应该支持和鼓励有效的事后总结。本节介绍组织如何激励健康的事后总结文化。我们着重指出文化失败的警告信号，并提供一些解决方案。我们还提供工具和模板，以简化和自动化事后总结过程。

### 示范并执行无责行为

为了适当地支持事后文化，工程主管应一如既往地以对事不对人行为为榜样，并在事后的各个方面鼓励不责怪他人。

事后总结讨论。您可以使用一些具体策略来强制组织中的对事不对人行为。

#### 不使用指责型的语言

指责型的语言扼杀了团队之间的协作。请考虑以下情形：

Sandy错过了Foo服务培训，并且不确定如何运行特定的更新命令。延迟最终延长了停机时间。

SRE杰西[Sandy的经理]说：“你是经理；为什么不确保每个人都完成培训？”

这个回应的主要问题将立即使对方处于自卫状态。更加优雅的回应是：

SRE杰西[Sandy的经理]说：“阅读事后总结后，我发现求职者错过了一项重要的培训，该培训会使他们更快地解决中断问题。也许应该要求团队成员完成此培训，然后才能加入轮班安排？或者，我们可以提醒他们，如果他们陷入困境，请迅速升级。毕竟，升级并不是罪过—尤其是如果它有助于减轻客户的痛苦！从长远来看，我们不应该真的太依赖培训，因为很容易在问题发生时忘记。”

#### 包括事后创作中的所有事件参与者

当停机的事后报告由单个人或由单个团队撰写时，很容易忽略造成停机的关键因素。

#### 收集反馈

清晰的事后总结流程和沟通计划可帮助防止指责性语言和观点在组织内传播。有关建议的结构化检查过程，请参阅第221页的“事后检查清单”部分。

## 奖励事后总结结果

事后总结记录撰写得当，行之有效并得到广泛分享时，是推动积极的组织变革和防止重复停机的有效工具。

考虑以下激励事后总结文化的策略。

#### 依据操作项目奖励

如果您奖励撰写事后报告的工程师，而不是奖励完成相关的操作项的工程师，则您可能会面临未完成事后报告的无情循环。确保在撰写事后报告和成功执行其行动计划之间保持动力平衡。

#### 奖励积极的组织变革

通过将事后总结报告作为在整个组织范围内扩大影响的机会，可以激励事后总结课的广泛实施。通过适当的奖金，积极的绩效评估，晋升等方法实施奖励。

### 突出显示可靠性的提升

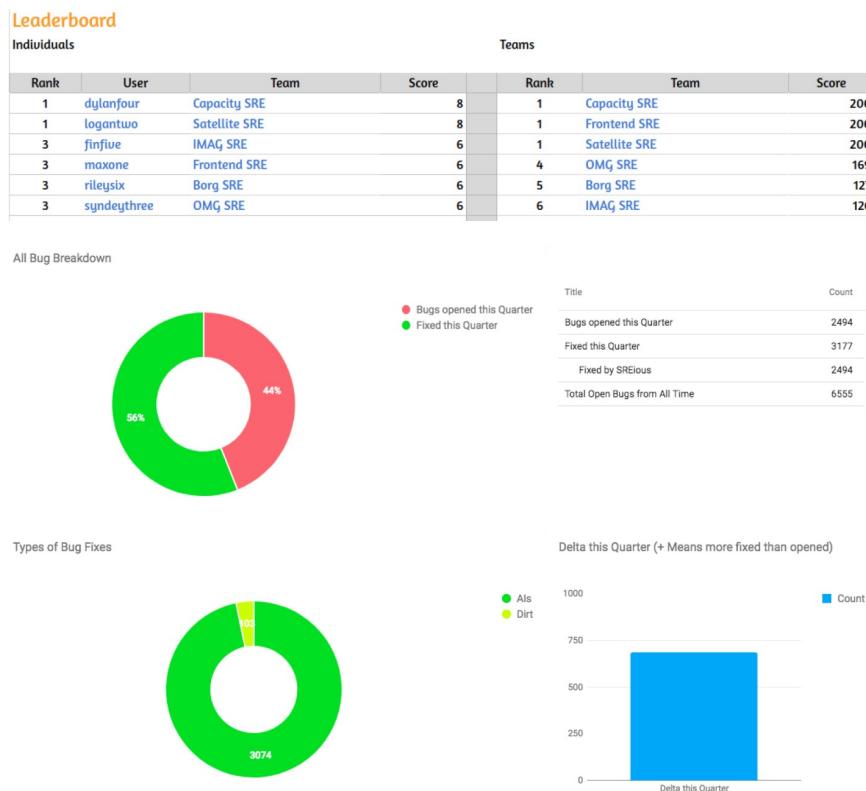
随着时间的流逝，有效的事后总结文化将带来更少的停机时间和更可靠的系统。因此，团队可以专注于功能而不是基础架构修补。从本质上讲，要在报告，演示文稿和绩效评估中强调这些改进。

### 让事后报告所有者担任领导者

通过电子邮件或会议来庆祝事后总结完成，或者通过给作者一个向听众介绍所学课程的机会，可以吸引欣赏公众赞誉的个人。将所有者设置为故障类型的“专家”并避免故障的发生，对于寻求同行认可的许多工程师而言，都是有益的。例如，您可能会听到有人说：“与Sara对话，她现在是专家。她与他人共同撰写了一份事后总结报告，报告里面她指出了解决该问题的方法！”

### 游戏化

某些人会因成就感和朝着更大的目标的进步(例如，解决系统缺陷和提高可靠性)而受到激励。对于这些人，记分板或完成事后行动项可能是一种诱因。在Google，我们每年两次举行“FixIt”活动。完成事后行动项目最多的SRE会获得少量赞赏和(当然)吹牛的权利。图10-3显示了事后排行榜的示例。



为了在组织内维持健康的事后总结文化，重要的是尽可能广泛地共享事后总结。即使采用以下策略之一也可以提供帮助。

### 在整个组织中分享公告

在您的内部沟通渠道，电子邮件，Slack等上宣布事后总结报告的可用性。如果您拥有一家正规公司，那么请实践一下分享近期的事后调查利益。

### 进行跨团队审核

对事后总结进行跨团队审查。在这些评论中，一个团队走过他们的事件，而其他团队则提出问题并进行替代学习。在Google，有几个办公室设有非正式的事后读书俱乐部，向所有员工开放。

此外，由跨职能的开发人员，SRE和组织负责人组成的小组审查了整个事后评估过程。这些人每月开会，审查事后总结程序和模板的有效性。

### 进行训练

培训新工程师时，请使用[不幸之轮](#):一组工程师会重演以前的事后总结，并假定事后总结中规定的角色。最初的事件指挥官出席会议是为了使体验尽可能“真实”。

### 每周报告一次事件和中断

创建每周停机报告，其中包含过去7天的事件和停机。与尽可能多的受众共享报告。从每周中断中，编制并共享定期的最大点击率报告。

## 应对事后文化故障

事后文化的崩溃可能并不总是显而易见的。以下是一些常见的故障模式和建议的解决方案。

### 避免形成联盟

脱离事后总结程序表明组织的事后总结文化正在失败。例如，假设SRE总监Parker听到了以下对话：

SWE山姆:哇，您听说过这么大的爆发吗？

SWE莱利:是的，这太可怕了。他们现在必须写一个事后总结报告。

SWE山姆:Oh，不！我很高兴我没有参与。

SWE莱利:是的，我真的不想参加讨论该会议的会议。

确保对高可见度的事后总结进行审查，可以帮助避免这种回避。此外，分享高质量的例子并讨论如何奖励参与其中的人可以帮助重新吸引个人。

### 未能加强文化

当高级管理人员使用指责性语言时做出回应可能具有挑战性。考虑一下高级领导在一次会议上关于停电的以下声明：

VP Ash:我知道我们应该无罪，但这是一个安全的空间。一定有人事先知道这是个坏主意，那么为什么不听那个人呢？

通过向更具建设性的方向发展叙事来减轻损害。例如：

SRE Dana:嗯，我敢肯定，每个人的意图都是最好的，所以都是无罪的，也许我们一般地问是否有任何我们可能会注意的警告信号，以及为什么我们会忽略它们。

个人以真诚的态度行事，并根据可获得的最佳信息做出决定。调查误导性信息的来源对组织而言比指责更为有益。(如果您遇到了敏捷原则，则应该[熟悉](#)。)

### 没有时间写事后总结

优质的事后总结报告需要花费时间。当团队负担过多其他任务时，事后总结的质量会受到影响。行动项不完整的不合格事后总结更容易复发。事后总结是您写给未来团队成员的信：保持一致的质量水准非常重要，以免您无意间给未来的队友们上了一个不好的教训。优先安排事后总结工作，跟踪事后总结完成和审查，并留出足够的时间来实施相关的行动计划。我们在第220页的“工具和模板”部分中讨论的工具可以帮助完成这些活动。

### 重复事件

如果团队遇到的错误反映出以前的事件，那么该是更深入地研究了。考虑问以下问题：

- 动作项目需要太长时间才能关闭吗？
- 新功能的速度胜过可靠性吗？
- 首先要捕捉正确的行动项目吗？
- 错误的服务是否应该进行重构？
- 人们是否将创可贴放在一个更严重的问题上？

如果您发现了系统性过程或技术问题，就该退后一步，考虑一下整个服务的运行状况。将每个类似事件的事后协作者聚集在一起，讨论防止重复的最佳行动方案。

## 工具和模板

一组工具和模板可以通过简化事后编写和管理关联数据的工作来引导事后文化。在这个领域，您可以利用Google和其他公司提供的许多资源。

### 事后总结模板

使用模板，可以更轻松地编写完整的事后总结并在整个组织中共享。使用标准格式可以使域外的读者更容易使用事后总结。您可以自定义模板以满足您的需求。例如，捕获特定于团队的元数据（如数据中心团队的硬件品牌/型号或受移动团队影响的Android版本）可能很有用。然后，随着团队的成熟和执行更复杂的事后分析，您可以添加自定义项。

### Google的模板

Google已在<http://g.co/SiteReliabilityWorkbookMaterials>上共享了Google文档格式的事后总结模板版本。在内部，我们主要使用Docs来编写事后总结，因为它可以通过共享的编辑权限和注释促进协作。我们的一些内部工具会使用元数据预先填充此模板，以使事后编写更容易。我们利用[Google Apps脚本](#)来自动化部分创作，并将大量数据捕获到特定的部分和表格中，以简化事后存储库解析数据进行分析。

### 其他行业模板

其他几家公司和个人也共享了他们的事后分析模板：

- [Pager Duty](#)
- [An adaptation of the original Google Site Reliability Engineering book template](#)

- [GitHub上托管的四个模板的列表](#)
- [GitHub用户Julian Dunn](#)
- [Server Fault](#)

## 事后总结工具

在撰写本文时，Google的事后管理工具尚未对外提供(请查看我们的[blog](#)了解最新更新)。但是，我们可以解释我们的工具如何促进事后文化。

### 事后总结制作

我们的事件管理工具收集并存储了大量有关事件的有用数据，并将这些数据自动推送到事后分析中。我们推送的数据示例包括：

- 事件指挥官和其他角色
- 详细的事件时间表和IRC日志
- 受影响的服务和根本原因服务
- 事件严重性
- 事件检测机制

### 事后检查清单

为了帮助作者确保正确完成事后总结，我们提供了事后总结清单，以指导所有者完成关键步骤。这只是列表中的一些示例检查：

- 对事件影响进行全面评估。
- 进行足够详细的根本原因分析，以推动行动计划的制定。
- 确保操作项目由服务的技术负责人审核和批准。
- 与更广泛的组织共享事后总结报告。

完整的核对清单可在<http://g.co/SiteReliabilityWorkbookMaterials>中找到。

### 事后总结保存

我们将事后总结存储在一个名为Requiem的工具中，因此任何Google员工都可以轻松找到它们。我们的事件管理工具会自动将所有事后总结信息推送至Requiem，组织中的任何人都可以发布其事后总结信息，以供所有人查看。自2009年以来，我们已经存储了成千上万的事后总结。Requiem从各个事后分析中解析出元数据，并将其用于搜索，分析和报告。

### 工具和模板

#### 事后追踪

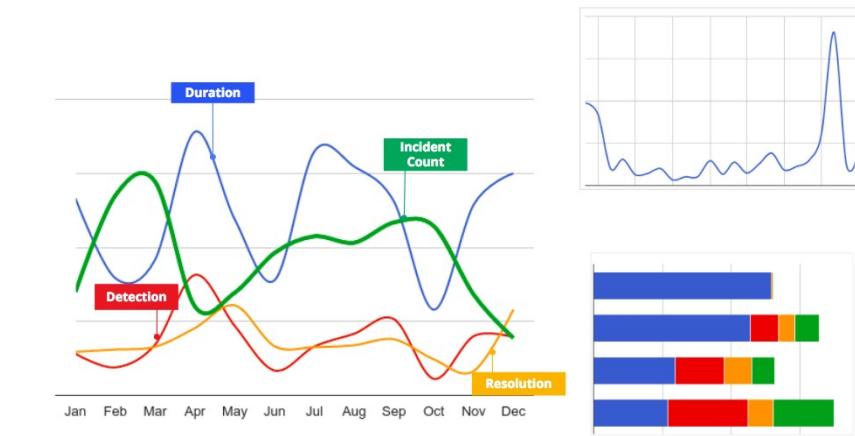
我们的事后总结信息存储在Requiem的数据库中。任何产生的操作项都会作为错误提交到我们的集中式错误跟踪系统中。因此，我们可以监控每个事后行动项的关闭情况。通过这种级别的跟踪，我们可以确保操作项不会漏掉间隙，导致服务变得越来越不稳定。图10-4显示了我们的工具启用的事后行动项目监控的模型。

Bug Priority	Incident Severity	Bug	Bug Summary	Linked Incident	Incident Date	Bug Create Date	Bug Age (days)	Bug Days Since Modified	Owner
P0	S0	205080437	SFC191 Overheating						
P0	S0	20440840	Create new release process and track exit criteria for the DMServer feature launches						
P0	S0	69431618	Ensure the dimmap is set up correctly (split, group precreate) before ranging further traffic to Spanner.						
P0	S0	20546188	AdmissionControl: issue credits and manage blocklists for the span events						
P0	S0	69505302	Customer API's should respect BHC security requirements, when possible						
P0	S0	12273659	Active monitoring on Checkpoint and other sites for potential PR fire. Rely to relevant teams.						
P0	S0	20208444	Review Spanner roll-out procedure including incremental steps with Docs eng leads						
P0	S0	68041622	Destroy SSD containing leaked key						
P0	S0	20221387	Chrome devices losing connectivity to manage networks						
P0	S1	28912652	Megastore Migration: Social/Graph/Chelfdex schema						
P0	S1	69922653	Extend Max queryability check to "height" until newer schema gets used by F1 (Instead of just verifying that F1 can consume newer schema through Datascap)						
P0	S1	65778172	Enable lean spam functionality for XTA pipeline to retro-blacklist BWM traffic post-serve (rollup)						
P0	S1	20488485	https://github.com/GoogleCloudPlatform/ComputeMetadata introduced a bug in Passwords.CreateMetadata						
P0	S1	20420241	Request for additional resource for Creator Academy from <a href="http://qa.us">http://qa.us</a>						

图10-4. 事后行动项目监控

### 事后分析

我们的事后管理工具将其信息存储在数据库中进行分析。团队可以使用该数据编写有关其事后趋势的报告，并确定其最易受攻击的系统。这有助于我们发现潜在的不稳定性或事件管理功能障碍的根本原因，而这些原因否则可能不会引起注意。例如，图10-5显示了使用我们的分析工具构建的图表。这些图表向我们显示了趋势，例如每个组织每个月有多少个事后调查，事件平均持续时间，检测时间，解决时间以及冲击半径。



Google

以下是一些第三方工具，可以帮助您创建，组织和分析事后总结:

- [Pager Duty Postmortems](#)
- [Morgue by Etsy](#)
- [VictorOps](#)

尽管不可能完全自动化编写事后总结的每个步骤，但是我们发现事后总结模板和工具可以使过程运行更加流畅。这些工具可以节省时间，使作者可以集中精力进行事后分析的关键方面，例如根本原因分析和行动项计划。

## 结论

持续进行的培养事后文化的投资以减少中断，为用户提供更好的整体体验以及来自依赖您的人们的更多信任的形式带来了回报。始终如一地应用这些实践可以带来更好的系统设计，更少的停机时间以及更有效，更快乐的工程师。如果确实发生了最坏的情况并且事件再次发生，您将遭受的损失更少，恢复得更快，并且拥有更多的数据来继续加强生产。

# 第11章

## 管理负载

*库珀·贝塞(Cooper Bethea)，格林·希林(Gráinne Sheerin)，珍妮弗·梅斯(Jennifer Mace)和露丝·金(Ruth King)以及加里·罗(Gary Luo)和加里·奥康纳(Gary O'Connor)撰写*

没有服务会在100%的时间内100%可用：客户可能会考虑不足，需求可能会增长50倍，服务可能会因流量高峰而崩溃，或者锚可能会拉跨大西洋电缆。有些人依赖您的服务，作为服务所有者，我们关心用户。当面对这些中断触发因素时，我们如何使我们的基础架构尽可能地自适应和可靠？

本章介绍了Google的流量管理方法，希望您可以使用这些最佳实践来提高服务的效率，可靠性和可用性。多年以来，我们发现没有均衡和稳定网络负载的单一解决方案。取而代之的是，我们结合使用多种工具，技术和策略，这些工具，技术和策略可以协同工作，以确保我们的服务可靠。

在进入本章之前，我们建议您阅读第19章([“前端的负载平衡”](#)和20([“数据中心的负载平衡”](#))。

## Google Cloud负载平衡

如今，大多数公司不再开发和维护自己的全局负载平衡解决方案，而是选择使用大型公共云提供商的负载平衡服务。我们将讨论Google Cloud Load Balancer(GCLB)作为大规模负载平衡的具体示例，但是我们介绍的几乎所有最佳实践也适用于其他云提供商的负载平衡器。

Google在过去的18年中一直在建立基础架构，以使我们的服务快速可靠。今天，我们使用这些系统来提供YouTube，地图，Gmail，搜索以及许多其他产品和服务。GCLB是我们的公共消耗性全局负载平衡解决方案，是我们内部开发的全局负载平衡系统之一的外部化。

本节描述了GCLB的组件以及它们如何协同工作来满足用户请求。我们跟踪从创建到到达目的地的典型用户请求。NianticPokémonGO案例研究提供了真实世界中GCLB的具体实现。

我们的第一本SRE书中的[第19章](#)描述了基于DNS的负载平衡是如何在用户开始连接之前平衡负载的最简单，最有效的方法。我们还讨论了这种方法的地域性的问题：它依赖于客户端的合作才能正确到期并重新获取DNS记录。因此，GCLB不使用DNS负载平衡。

相反，我们使用anycast，这是一种将客户端发送到最近的群集而不依赖DNS地理位置的方法。Google的全球负载平衡器知道客户端的位置，并将数据包定向到最近的Web服务，从而在使用单个[虚拟IP(VIP)]时为用户提供低延迟。

(<http://bit.ly/2kFchKX>)使用单个VIP意味着我们可以增加DNS记录的生存时间(TTL)，从而进一步减少延迟。

### Anycast

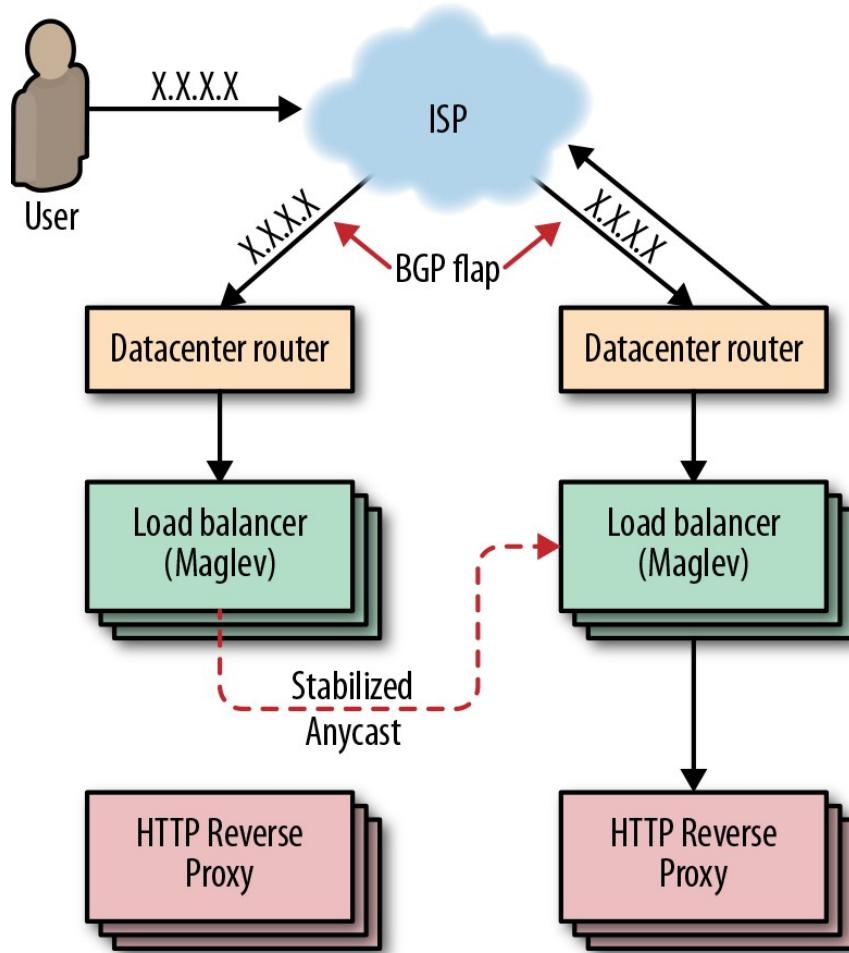
Anycast是一种网络寻址和路由方法。它将数据报从单个发送方路由到一组潜在接收方中拓扑最近的节点，这些接收方都由同一目标IP地址标识。Google通过[边界网关协议\(BGP\)](#)从我们网络中的多个点播报IP。我们依靠BGP路由网格将数据包从用户传递到可以终止传输控制协议(TCP)会话的最近的前端位置。这种部署消除了单播IP扩散的问题，并为用户找到了最接近的前端。仍然存在两个主要问题：

- 太多附近的用户会淹没前端站点。
- BGP路由计算可能会重置连接。

考虑一个ISP，它经常重新计算其BGP路由，以使其用户之一偏爱两个前端站点之一。每次BGP路由“震荡”时，所有进行中的TCP流都会重置，因为不幸的用户数据包将被定向到没有TCP会话状态的新前端。为了解决这些问题，我们利用了连接级负载平衡器Maglev(稍后将进行介绍)来协调TCP流，即使路由发生振荡也是如此。我们将此技术称为“稳定的Anycast”。

### 稳定的Anycast

如图11-1所示，Google使用我们的自定义负载平衡器Maglev实现了稳定的Anycast。为了稳定的Anycast，我们为每台Maglev机器提供了一种将客户端IP映射到最近的Google前端站点的方法。有时，Maglev(Maglev)会处理发往与另一个前端站点更近的客户端的Anycast VIP的数据包。在这种情况下，Maglev将数据包转发到位于最接近的前端站点的计算机上的另一个Maglev以进行传送。然后，最靠近前端站点的Maglev机器就像对待其他任何数据包一样简单地对待数据包，然后将其路由到本地后端。



Maglev，如图11-2所示，是Google的自定义分布式数据包级负载均衡器。Maglev机器是我们云架构不可或缺的一部分，它管理进入集群的流量。它们在我们的前端服务器之间提供有状态的TCP级负载平衡。Maglev在一些关键方面与其他传统的硬件负载平衡器不同：

- 通过等价多路径(ECMP)转发，可以将目的地为给定IP地址的所有数据包平均分配到Maglev计算机池中。这使我们能够通过简单地将服务器添加到池中来提高Maglev的容量。均匀地扩展数据包还可以使Maglev冗余模型化为  $N + 1$ ，与传统的负载平衡系统(通常依靠主动/被动对来提供1 + 1冗余)相比，提高了可用性和可靠性。
- Maglev是Google的自定义解决方案。我们端到端地控制系统，这使我们能够进行实验和快速迭代。
- Maglev在我们数据中心的商用硬件上运行，从而大大简化了部署。

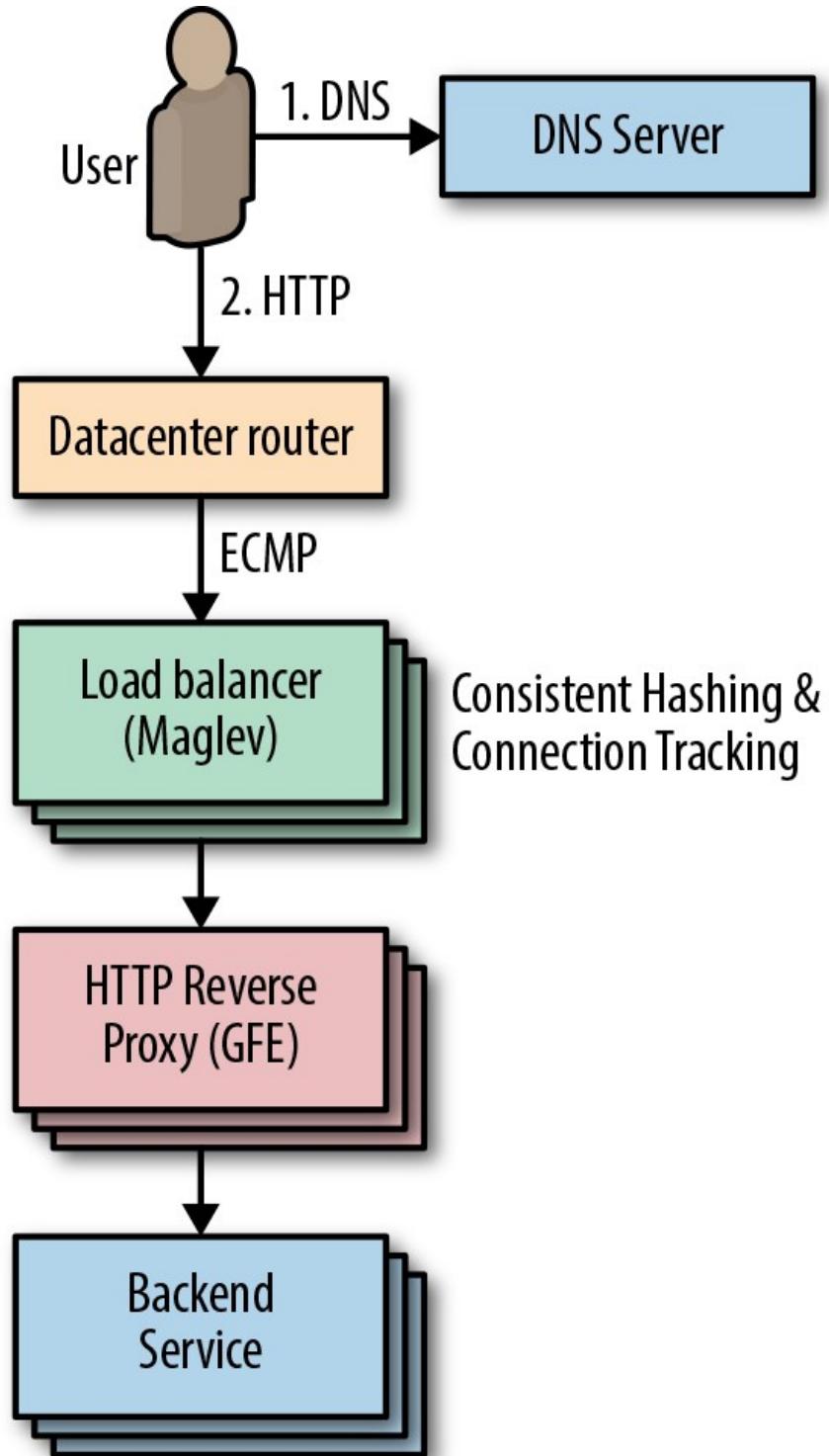


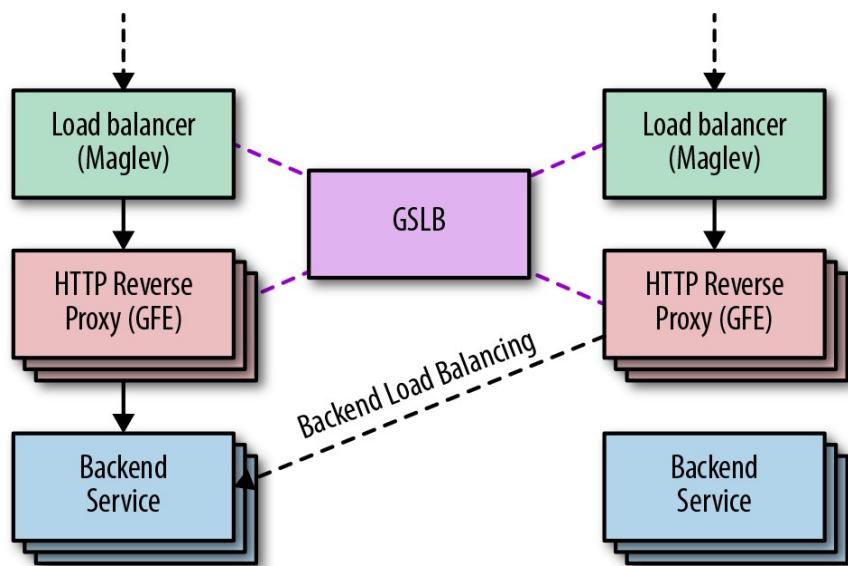
图11-2.Maglev

Maglev数据包传递使用一致性哈希和连接跟踪。这些技术在终止TCP会话的HTTP反向代理(也称为Google Front Ends，或\* GFEs)中合并TCP流。一致的哈希和连接跟踪是Maglev能够按数据包而不是按连接数进行扩展的能力的关键。当路由器收到发往Maglev托管的VIP的数据包时，路由器会通过ECMP将数据包转发到群集中的任何Maglev机器。Maglev接收到数据包时，将计算该数据包的5元组hash<sup>67</sup>，并在其连接跟踪表中查找哈希值，该表中包含最近连接的路由结果。如果Maglev找到一

个匹配项，并且所选的后端服务仍然正常，则它将重新使用该连接。否则，Maglev会退回到一致的散列来选择后端。这些技术的结合消除了在各个Maglev机器之间共享连接状态的需要。

### 全球软件负载平衡器

GSLB是Google的全球软件负载平衡器。它使我们能够平衡集群之间的实时用户流量，从而使用户需求与可用服务容量相匹配，从而可以以对用户透明的方式处理服务故障。如图11-3所示，GSLB控制与GFE的连接分配以及对后端服务的请求分配。GSLB允许我们从运行在不同集群中的后端和GFE服务用户。除了前端和后端之间的负载平衡外，GSLB还了解后端服务的运行状况，并且可以自动将流量从出现故障的群集中转移出去。



如图11-4所示，GFE位于外界和各种Google服务(网络搜索，图像搜索，Gmail等)之间，并且通常是客户端HTTP(S)请求遇到的第一个Google服务器。GFE终止客户端的TCP和SSL会话，并检查HTTP标头和URL路径，以确定哪个后端服务应处理该请求。GFE决定将请求发送到何处后，它将重新加密数据并转发请求。有关此加密过程如何工作的更多信息，请参见我们的白皮书["Google Cloud传输中的加密"](#)。

GFE还负责对其后端进行健康检查。如果后端服务器返回否定确认("NACKs"请求)或使运行状况检查超时，则GFE会停止向失败的后端发送流量。我们使用此信号更新GFE后端，而不会影响正常运行时间。通过将GFE后端置于一种模式，使它们在继续运行中请求时仍无法通过运行状况检查，我们可以在不中断任何用户请求的情况下从服务中正常删除GFE后端。我们称之为"蹩脚鸭"模式，我们将在第一本SRE书的[第20章](#)中对其进行详细讨论。

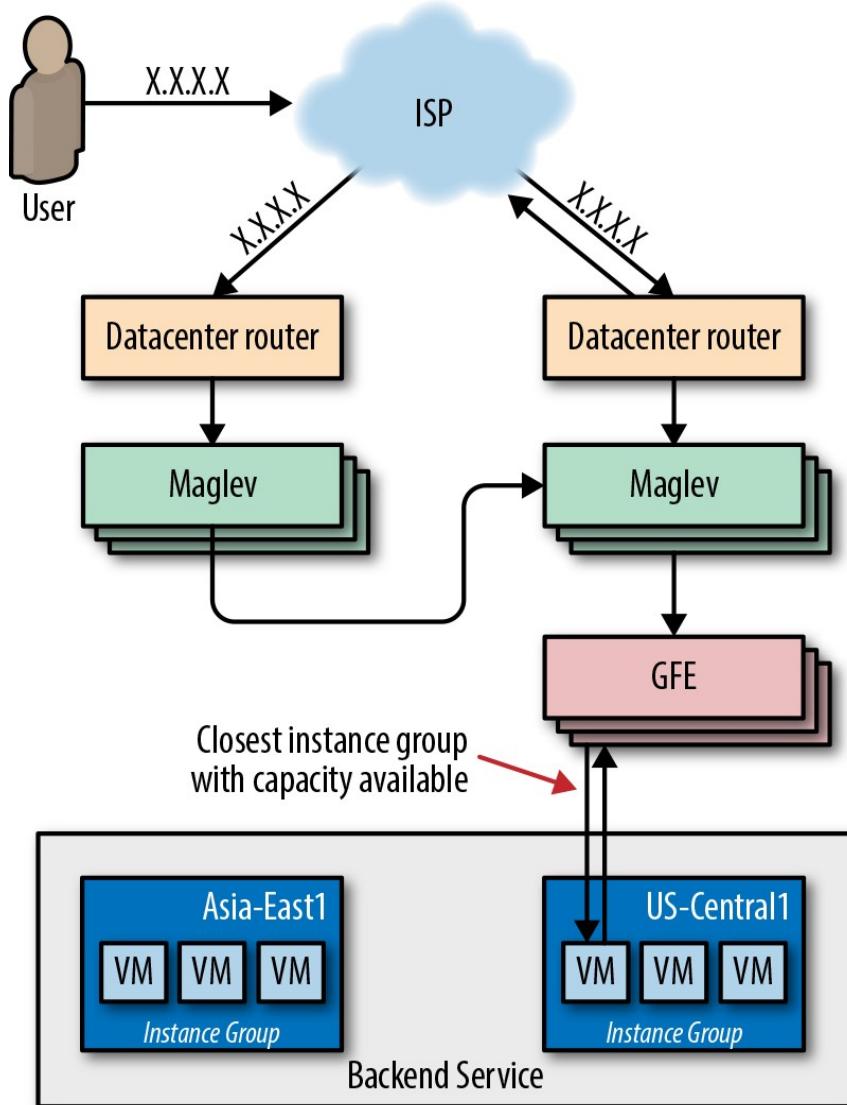


图11-4.GFE

GFE还维护与其最近活动的所有后端的持久会话，以便在请求到达时就可以使用连接。此策略有助于减少用户的延迟，尤其是在我们使用SSL来保护GFE与后端之间的连接的情况下。

#### GCLB·低延迟

我们的网络供应策略旨在减少最终用户对我们服务的延迟。由于要通过HTTPS协商安全连接，需要在客户端和服务器之间进行两次网络往返，因此特别重要的是，我们应尽量减少这部分请求时间的延迟。为此，我们扩展了网络的边缘以托管Maglev和GFE。这些组件会在尽可能接近用户的位置终止SSL，然后通过长期存在的加密连接将请求转发到我们网络内更深的后端服务。

我们在该Maglev/GFE组合边缘网络的顶部构建了GCLB。当客户创建负载均衡器时，我们将提供一个Anycast VIP，并对Maglev进行编程，以在我们网络边缘的GFE上对其进行全局负载均衡。GFE的作用是终止SSL，接受和缓冲HTTP请求，将这些请求转发到客户的后端服务，然后将响应代理回用户。GSLB在每一层之间提供了glue：它使Maglev能够找到具有可用容量的最近的GFE位置，并使GFE能够将请求路由到具有可用容量的最近的VM实例组。

### GCLB:高可用性

为了向我们的客户提供高可用性，GCLB提供了99.99%的可用性SLA<sup>68</sup>。此外，GCLB还提供支持工具，使我们的客户可以改善和管理自己的应用程序的可用性。将负载平衡系统视为一种流量管理器很有用。在正常运行期间，GCLB将流量路由到具有可用容量的最近后端。当您的一个服务实例发生故障时，GCLB会代表您检测到故障，并将流量路由到正常的实例。

金丝雀渐进式和逐步发布可帮助GCLB保持高可用性。金丝雀是我们的标准发布程序之一。如第16章所述，此过程涉及将新应用程序部署到数量很少的服务器上，然后逐渐增加流量并仔细观察系统行为以验证是否没有恢复。这种做法通过在金丝雀阶段的早期捕获它们来减少任何恢复的影响。如果新版本崩溃或以其他方式使运行状况检查失败，则负载平衡器将绕过它。如果您检测到非致命的回归，则可以从负载均衡器中删除实例组，而无需接触应用程序的主版本。

### 案例研究1: GCLB上的PokémonGO

Niantic在2016年夏天推出了PokémonGO。这是多年来第一款新的神奇宝贝游戏，第一款官方的神奇宝贝智能手机游戏，也是Niantic与一家大型娱乐公司合作的第一个项目。这款游戏大受欢迎，而且比任何人都想像的更受欢迎-那个夏天，您经常会看到玩家在虚拟世界中的神奇宝贝体育馆周围地标对决。

神奇宝贝GO的成功大大超出了Niantic工程团队的期望。在发布之前，他们对软件堆栈进行了负载测试，以处理最乐观的流量估算值的5倍。实际的每秒启动请求(RPS)速率是估计值的近50倍-足以对几乎所有软件堆栈提出扩展挑战。使事情变得更加复杂的是，《PokémonGO》的世界是高度互动的，并且在其用户之间全球共享。给定区域中的所有玩家都可以看到游戏世界的相同视图，并在该世界中彼此互动。这就要求游戏产生并向所有参与者共享的状态分发近乎实时的更新。

将游戏扩展到50倍以上的用户需要Niantic工程团队做出真正令人印象深刻的努力。此外，Google的许多工程师都在协助扩展服务以成功启动方面提供了帮助。在迁移到GCLB的两天之内，Pokemon GO应用成为了最大的GCLB服务，轻松与其他前十名GCLB服务相提并论。

如图11-5所示，PokémonGO在启动时使用Google的区域性Network Load Balancer(NLB)来平衡Kubernetes群集。每个群集包含Nginx实例的容器，这些实例用作第7层反向代理，可终止SSL，缓冲HTTP请求并在应用程序服务器后端的容器之间执行路由和负载平衡。

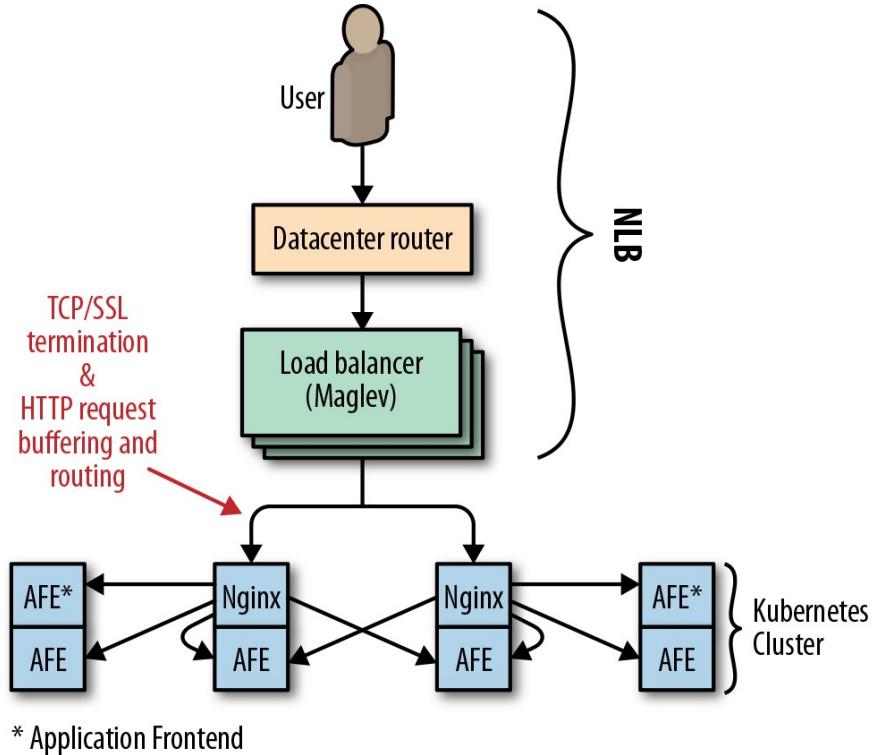


图11-5.神奇宝贝GO(GCLB之前的版本)

NLB负责IP层的负载平衡，因此有效使用NLB的服务将成为Maglev的后端。在这种情况下，依靠NLB对Niantic有以下影响：

- Nginx后端负责终止客户端的SSL，这需要从客户端设备到Niantic的前端代理进行两次往返。
- 需要缓冲来自客户端的导致代理层上的资源耗尽的HTTP请求，尤其是当客户端只能缓慢发送字节时。
- 数据包级代理无法有效地缓解低级网络攻击，例如`SYN Flood`。

为了适当地扩展，Niantic需要在大型边缘网络上运行的高级代理。NLB无法解决此问题。

### 迁移到GCLB

大型SYN洪水攻击使将PokémonGO迁移到GCLB成为优先事项。此次迁移是Niantic与Google客户可靠性工程(CRE)和SRE团队的共同努力。最初的过渡发生在流量低谷期间，当时并不明显。但是，随着流量增加到高峰，Niantic和Google都出现了无法预料的问题。Google和Niantic都发现，真正的客户对PokémonGO流量的需求比以前观察到的高200%。Niantic前端代理收到了如此多的请求，以致他们无法跟上所有入站连接的速度。以这种方式拒绝的任何连接都不会在入站请求的监控中浮出水面。后端从来没有机会。

这种流量激增导致了典型的级联故障情况。大量的API支持服务-Cloud Datastore，PokémonGO后端和API服务器以及负载平衡系统本身-超出了Niantic的云项目可用的容量。过载导致Niantic的后端变得非常慢(而不是拒绝请求)，表现为请求超时到负载均衡层。在这种情况下，负载均衡器将重试GET请求，从而增加系统负载。极

高的请求量和增加的重试功能相结合，使GFE中的SSL客户端代码达到前所未有的水平，因为它试图重新连接到无响应的后端。这导致GFE的性能严重下降，从而使GCLB的全球产能实际上降低了50%。

由于后端失败，PokémonGO应用程序尝试代理用户重试失败的请求。当时，该应用程序的重试策略是一次立即重试，然后不断退避。随着中断的继续，服务有时会返回大量快速错误-例如，当重新启动共享后端时。这些错误响应可有效地使客户端重试同步，从而产生“雷群”问题，在该问题中，许多客户端请求实际上是在同一时间发出的。如图11-6所示，这些同步的请求峰值极大地增加到了以前的全局RPS峰值的20倍。

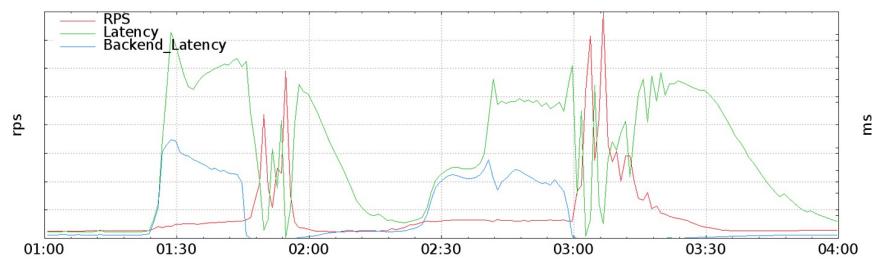


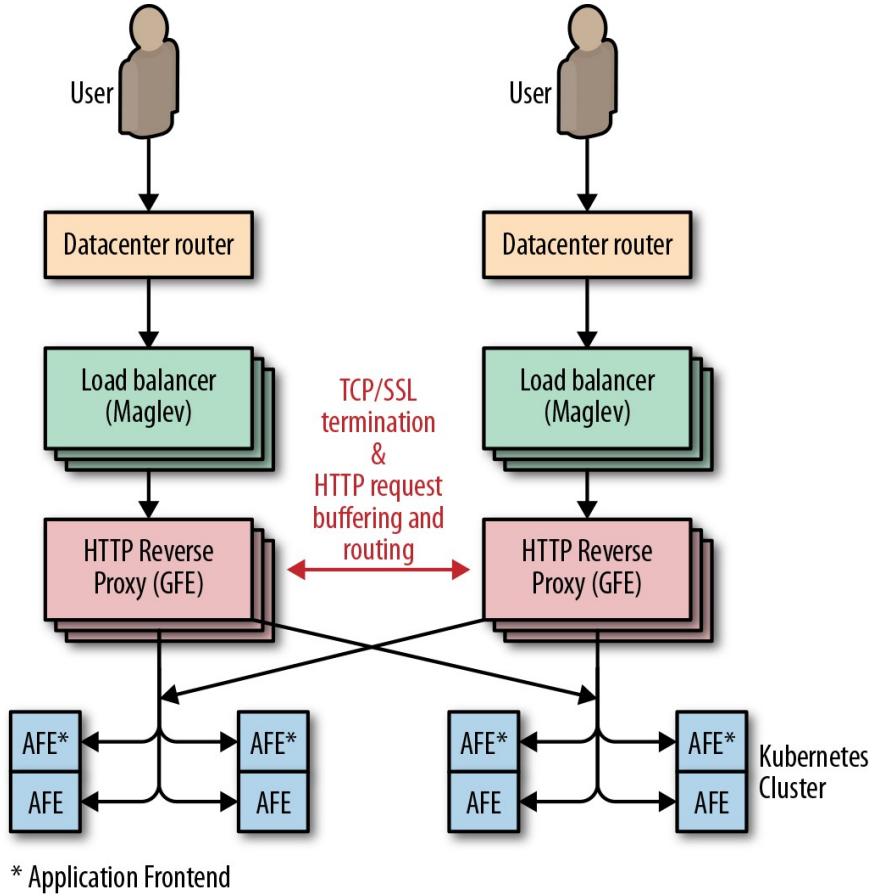
图11-6. 同步客户端重试导致的流量高峰

### 解决问题

这些请求高峰，再加上GFE容量回归，导致所有GCLB服务的排队和高延迟。Google的流量值班SRE通过执行以下操作来减少对其他GCLB用户的附带损害：

1. 从负载均衡器主池中隔离可以为PokémonGO流量提供服务的GFE。
2. 扩大隔离的PokémonGO池，直到尽管性能下降也可以处理高峰流量。此操作将容量瓶颈从GFE移到了Niantic堆栈，在该堆栈中服务器仍在超时，特别是在客户端重试开始同步和高峰时。
3. 在Niantic的祝福下，流量SRE实施了管理覆盖，以限制负载平衡器代表PokémonGO接受的流量。该策略包含的客户需求足以使Niantic重新建立正常运营并开始扩大规模。

最终的网络配置如图11-7所示。



\* Application Frontend

在此事件之后，Google和Niantic都对其系统进行了重大更改。Niantic向其客户端引入了抖动，截去了指数级重试<sup>69</sup>，从而抑制了级联失败期间经历的大量同步重试峰值。Google学会了将GFE后端视为潜在的重要负载来源，并制定了资格鉴定和负载测试实践，以检测由于后端缓慢或行为异常而导致的GFE性能下降。最终，两家公司都意识到他们应该尽可能地靠近客户端测量负载。如果Niantic和Google CRE能够准确预测客户的RPS需求，那么我们将比切换到GCLB之前抢先分配给Niantic的资源更多。

### 自动缩放

GCLB之类的工具可以帮助您有效地平衡整个机队的负载，使您的服务更加稳定和可靠。有时，您根本没有足够的资源来管理现有流量。您可以使用[自动扩展](#)从战略上扩展您的舰队。无论您是增加每台计算机的资源(垂直扩展)还是增加池中的计算机总数(水平扩展)，自动扩展都是一个功能强大的工具，如果使用正确，它可以提高服务的可用性和利用率。相反，如果配置错误或使用不当，自动缩放可能会对您的服务产生负面影响。本节介绍了一些最佳实践，常见的故障模式以及自动缩放的当前限制。

### 处理不健康的机器

自动缩放通常会将所有实例的利用率平均化，而不管其状态如何，并假定实例在请求处理效率方面是同质的。当计算机不服务(称为运行不正常的实例)但仍计入利用率平均值时，自动缩放会遇到问题。在这种情况下，根本就不会发生自动缩放。各种问题均可触发此故障模式，包括：

- 需要很长时间才能准备服务的实例(例如，加载二进制文件或进行预热时)

- 卡在非服务状态的实例(即僵尸)

我们可以使用多种策略来改善这种情况。您可以组合或单独进行以下改进:

#### 负载均衡

使用负载均衡器观察到的容量度量标准进行自动缩放。这将自动从不正常的实例中扣除不正常的实例。

#### 请等待新实例稳定后再收集指标

您可以配置自动缩放器以仅在新实例运行状况良好时收集有关新实例的信息(GCE将该非活动时间段称为"冷却时间段")。

#### 自动缩放和自动修复

自动修复会监控您的实例，并在实例不正常时尝试重新启动它们。通常，您将自动修复程序配置为监控实例公开的运行状况指标。如果自动修复程序检测到实例已关闭或运行不正常，它将尝试重新启动。在配置您的自动修复器时，

重要的是要确保您有足够的时间让实例在重新启动后恢复健康。

结合使用这些解决方案，您可以优化水平自动缩放比例，以仅跟踪运行状况良好的机器。请记住，在运行服务时，自动缩放器会不断调整您的设备规模。创建新实例从来都不是即时的。

#### 使用状态系统

有状态系统将用户会话中的所有请求一致地发送到同一后端服务器。如果这些途径不堪重负，则添加更多实例(即水平缩放)将无济于事。对于状态系统而言，分散负载的智能任务级路由(例如，使用一致的哈希<sup>70</sup>)是一种更好的策略。

垂直自动缩放在有状态系统中很有用。当与任务级平衡结合使用以均衡系统负载时，垂直自动缩放可以帮助吸收短期热点。请谨慎使用此策略:由于垂直自动缩放通常在所有实例之间是一致的，因此，低流量实例可能会不必要地变大。

#### 保守地配置

与自动缩小相比，使用自动缩放进行放大更重要且风险更低，因为缩小错误再放大会导致过载和流量下降。通过设计，大多数自动缩放器实现对流量的跃迁比对流量的下降更敏感。扩大规模时，自动缩放器倾向于快速增加额外的服务容量。进行缩减时，它们会更加谨慎，并等待更长的时间才能满足缩放条件，然后再慢慢减少资源。

随着服务远离瓶颈，您可以吸收的负载峰值会增加。我们建议配置自动缩放器，以使服务远离关键的系统瓶颈(例如CPU)。自动缩放器还需要足够的时间来做出反应，尤其是在新实例无法启动并无法立即使用时。我们建议面向用户的服务保留足够的备用容量，以用于过载保护和冗余。<sup>71</sup>

#### 设置约束

自动定标器是一个强大的工具。如果配置错误，它可能会失控。您可能会因引入错误或更改设置而无意中引发严重后果。例如，请考虑以下情形:

- 您已配置自动缩放以根据CPU利用率进行缩放。您发布了系统的新版本，其中包含一个错误，该错误导致服务器消耗CPU而不进行任何工作。自动缩放器会通过不断放大该作业来作出反应，直到浪费所有可用的配额。
- 您的服务没有任何改变，但是依赖项失败了。此故障会导致所有请求都卡在您的服务器上，并且永远无法完成，从而始终消耗资源。自动扩展器将扩大工作量，导致越来越多的流量被阻塞。失败的依赖项上增加的负载可能会阻止依赖项恢复。

限制允许自动缩放器执行的工作非常有用。设置扩展的最小和最大界限，确保您有足够的配额来扩展到设置的限制。这样做可以防止您耗尽配额并提供帮助

进行容量规划。

#### 包括终止开关和手动优先

最好包括终止开关，以防自动缩放出现问题。确保您的值班工程师了解如何禁用自动缩放以及在必要时如何手动缩放。您的自动缩放终止开关功能应该简单，明显，快速并且有据可查。

#### 避免后端过载

正确配置的自动缩放器将根据流量的增加而扩展。流量增加将对堆栈产生影响。后端服务(例如数据库)需要承担服务器可能创建的任何其他负载。因此，在部署自动缩放器之前，最好对后端服务进行详细的依赖性分析，特别是因为某些服务可能比其他服务线性地扩展。确保您的后端有足够的额外容量来服务增加的流量，并在过载时能够正常降解。使用分析数据来通知自动缩放器配置的限制。

服务部署通常运行共享配额的各种微服务。如果微服务因应流量高峰而扩大规模，则可能会使用大部分配额。如果单个微服务上的流量增加意味着其他微服务上的流量增加，则将没有可用的配额来扩展其余微服务。在这种情况下，依赖关系分析可以帮助您引导您先行实施有限的扩展。或者，您可以为每个微服务实现单独的配额(这可能需要将服务拆分为单独的项目)。

#### 避免流量不平衡

某些自动缩放器(例如AWS EC2，GCP)可以在区域实例组(RMiGs)之间平衡实例。除了常规自动缩放外，这些自动缩放器还运行单独的作业，该作业不断尝试使区域中每个区域的大小均匀。以这种方式重新平衡流量避免了一个大区域。如果您使用的系统按区域分配配额，则此策略会使您的配额使用量均匀化。此外，跨区域的自动缩放可为故障域提供更多多样性。

## 组合策略来管理负载

如果您的系统变得足够复杂，则可能需要使用多种负载管理。例如，您可能会运行几个托管实例组，这些实例组会随负载扩展，但会在多个区域中进行克隆以提高容量；因此，您还需要平衡区域之间的流量。在这种情况下，您的系统需要同时使用负载均衡器和基于负载的自动缩放。

或者，也许您在遍布全球的三个托管设施中运行一个网站。您希望在本地提供缩小延迟服务，但是由于部署更多计算机需要花费数周的时间，因此溢出容量需要溢出到其他位置。如果您的网站在社交媒体上受到欢迎，并且流量突然增加了五倍，那

么您希望能够满足您的要求。因此，您可以实施减载以减少多余的流量。在这种情况下，您的系统需要同时使用负载均衡器和负载削减。

或者，您的数据处理管道位于一个云区域中的Kubernetes集群中。当数据处理速度大大降低时，它会提供更多的Pod来处理工作。但是，当数据输入速度如此之快以至于读取数据导致您用尽内存或减慢垃圾收集速度时，您的Pod可能需要暂时或永久释放负载。在这种情况下，您的系统需要同时使用基于负载的自动缩放和基于负载减少的技术。

负载平衡，负载减少和自动缩放都是针对相同目标设计的：均衡和稳定系统负载。由于这三个系统通常是分别实现，安装和配置的，因此它们似乎是独立的。但是，如图11-8所示，它们并不完全独立。以下案例研究说明了这些系统如何相互作用。

### 结合策略来管理负载

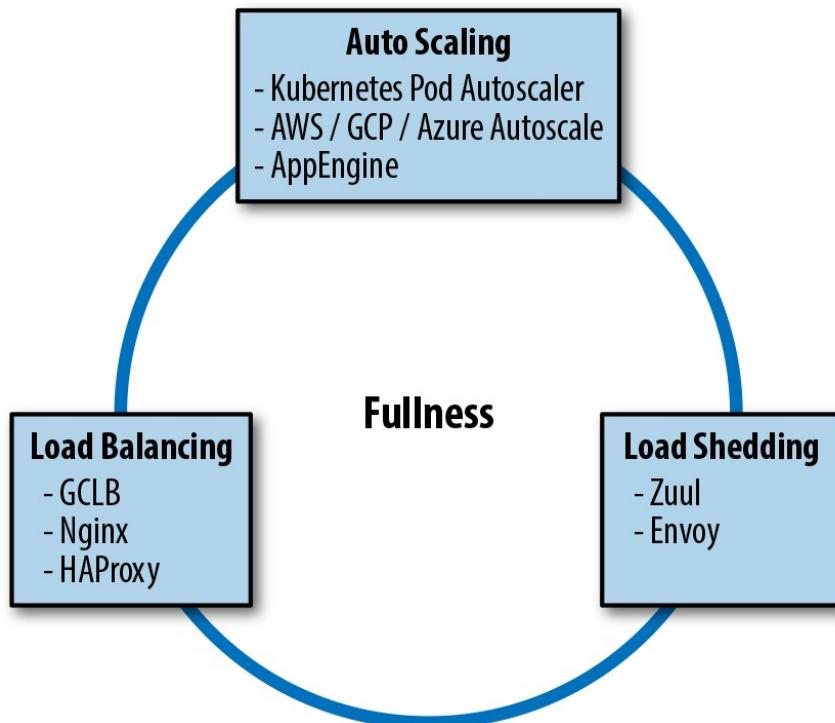


图11-8.完整的交通管理系统

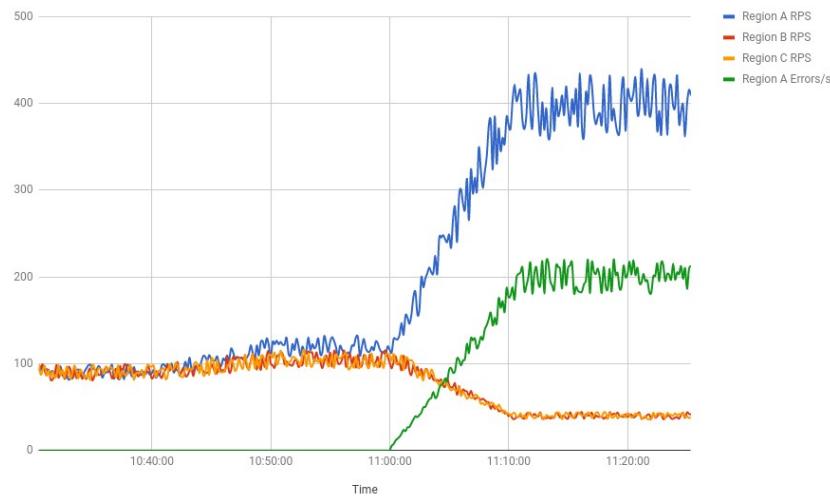
### 案例研究2:减载攻击时

想象一个虚构的公司Dressy，该公司通过应用程序在线销售礼服。由于这是一项流量驱动的服务，因此Dressy的开发团队在三个区域中部署了他们的应用程序。通过这种部署，他们的应用可以快速响应用户请求并应对单区域故障-或他们想这么做。

Dressy的客户服务团队开始收到客户无法访问该应用程序的投诉。Dressy的开发团队正在调查并注意到一个问题：他们的负载平衡正在莫名其妙地将所有用户流量吸引到区域A，即使该区域已满到溢出并且B和C都是空的(并且都很大)。事件的时间表(见图11-9)如下：

1. 一天开始时，流量图显示所有三个群集稳定在90 RPS。
2. 上午10:46，这三个地区的流量开始增加，因为渴望购物的人开始寻找便宜货。

3. 上午11:00，区域A在区域B和C之前达到了120 RPS。
4. 在上午11:10，区域A继续增长到400 RPS，而区域B和C下降到40 RPS。
5. 负载均衡器稳定在此状态。
6. 命中区域A的大多数请求都返回503错误。
7. 请求到达此群集的用户开始抱怨。



**图11-9. 区域交通**

如果开发团队参考了他们的负载均衡器的完全图，他们会发现一些非常奇怪的东西。负载平衡器可感知利用率：它正在从Dressy的容器中读取CPU利用率，并使用此信息来估计填充度。据其所知，区域A中每个请求的CPU利用率比区域B或区域C低10倍。负载均衡器确定所有区域均等地负载，并且其工作已完成。

### 发生了什么事？

在本周早些时候，为了防止级联过载，该团队启用了减载功能。只要CPU利用率达到某个阈值，服务器就会针对收到的任何新请求返回错误，而不是尝试处理它们。在这一天，区域A达到该阈值，稍稍超过其他区域。每个服务器开始拒绝接收到的请求的10%，然后拒绝20%的请求，然后拒绝50%。在此时间段内，CPU使用率保持不变。

就负载均衡器系统而言，每个连续的丢弃请求都减少了每个请求的CPU成本。区域A的效率远远高于区域B和C。在80%的CPU(脱落上限)下，区域A的服务速度为240 RPS，而区域B和C的管理速度仅为120 RPS。从逻辑上讲，它决定向A发送更多请求。

### 结合策略来管理负载

#### 什么地方出了错？

简而言之，负载均衡器不知道“有效”请求是错误的，因为负载削减和负载均衡系统没有通信。每个系统可能是由不同的工程师分别添加和启用的。没有人将它们作为一个统一的负载管理系统进行检查。

### 经验教训

为了有效地管理系统负载，我们需要认真设计-既要配置各个负载管理工具，又要管理它们之间的交互。例如，在Dressy案例研究中，向负载均衡器逻辑中添加错误处理将解决此问题。假设每个“错误”请求计为120%的CPU利用率(任何超过100的数字都可以使用)。现在，区域A看起来超载。请求将传播到B和C，并且系统将均衡。

您可以使用类似的逻辑将该示例推断为负载管理策略的任意组合。在采用新的负载管理工具时，请仔细检查它如何与系统已在使用的其他工具进行交互，并检测它们的交集。添加监控以检测反馈循环。确保可以在整个负载管理系统中协调紧急关闭触发器，并在这些系统严重失控的情况下考虑添加自动关闭触发器。如果您没有事先采取适当的预防措施，则可能在事后总结时必须这样做。

容易说“采取预防措施”。更具体地说，根据部署的负载管理类型，您可能会考虑以下预防措施：

#### 负载均衡

负载平衡通过路由到距离用户最近的位置来最大程度地减少延迟。自动缩放可以与负载均衡一起使用，以增加靠近用户的位置的大小，然后将更多的流量路由到那里，从而形成积极的反馈回路。

如果需求主要是最接近某个位置，则该位置的大小将不断增加，直到所有服务容量都集中在一个位置。如果此位置下降，其余位置将变得超载，流量可能会下降。扩大这些位置将不是立即的。您可以通过为每个位置设置最小实例数来保留故障转移的备用容量，从而避免这种情况。

#### 减载

设置阈值是一个好主意，以使系统在负载剥落开始之前自动扩展。否则，您的系统可能会开始减少如果首先扩展就可以满足的流量。

#### 使用RPC管理负载

处理正确的请求对于提高效率很重要：您不希望自动扩展以服务那些对用户无益的请求，或者因为处理不重要的请求而不必要地减轻了负载。同时使用自动缩放和减载时，重要的是为RPC请求设置截止日期。

进程为所有进行中的请求保留资源，并在请求完成后释放这些资源。在没有特定截止日期的情况下，系统将为所有进行中的请求保留资源，最大可能的限制。默认情况下，该截止日期非常长(取决于语言的实现方式-某些语言API的工作时间是固定的，而其他语言API的工作时间是持续的)。此行为导致客户端，最终用户，经历更高的延迟。该服务还存在耗尽资源(如内存)和崩溃的风险。

要优雅地处理此情况，我们建议服务器终止耗时太长的请求，并且客户端取消对他们不再有用的需求。例如，如果客户端已经向用户返回错误，则服务器不应启动昂贵的搜索操作。要设置服务的行为期望，您只需在API的.proto文件中提供注释即可建议默认截止日期。另外，设置客户端的故意期限(例如，请参阅我们的博客文章[“gRPC和Deadlines”](#))。

## 结论

根据Google的经验，没有完善的流量管理配置。自动缩放是一个功能强大的工具，但很容易出错。除非进行了仔细的配置，否则自动缩放会导致灾难性的后果，例如，在隔离配置这些工具时，负载平衡，减载和自动缩放之间可能存在灾难性的反馈循环。正如《PokémonGO》案例研究所说明的那样，当流量管理基于系统之间交互的整体视图时，其效果最佳。

我们一次又一次地看到，当服务全部同步失败时，没有任何负载减少，自动扩展或节流将节省我们的服务。例如，在《PokémonGO》案例研究中，同步客户端重试与负载平衡器(等待后端服务器无响应)的结合产生了"雷鸣般的冲击"。要使服务正常运行，您需要提前计划以减轻潜在的问题。您的缓解策略可能涉及设置标志，更改默认行为，启用昂贵的日志记录或公开流量管理系统用于决策的参数的当前值。

我们希望本章中提供的策略和见解可以帮助您管理自己服务的流量并保持用户满意。

<sup>67</sup>. 5元组包括以下内容: 源地址，目标地址，源端口，目标端口和传输协议类型。 ↵

<sup>68</sup>. 有关SLO和SLA之间差异的说明，请参见Google Cloud Platform博客文章"[SLO，SLI，SLA，哦，我的CRE生活课程](#)"。 ↵

<sup>69</sup>. 有关此主题的更多信息，请参见第一本SRE书的[第22章](#)。 ↵

<sup>70</sup>. 参见《[站点可靠性工程](#)》，[第19章](#) ↵

<sup>71</sup>. 有关容量规划中冗余的更多信息，请参见《[站点可靠性工程](#)》，[附录B](#)。  
 ↵

## 第12章

### 介绍非抽象的大型系统设计

由*Salim Virji , James Youngman , Henry Robertson*

*斯蒂芬·索恩(Stephen Thorne) , 戴夫·雷恩(Dave Rensin)和佐尔坦·埃吉(Zoltan Egyed)与理查德·邦迪(Richard Bondi)撰写*

SRE担负着跨生产运营和产品工程的职责，在调整业务案例要求和运营成本方面处于独特的位置。产品工程团队可能没有意识到他们设计的系统的维护成本，特别是如果该产品团队正在构建一个将更大的生产生态系统纳入考虑范围的组件时，尤其如此。

根据Google在开发系统方面的经验，我们认为可靠性是任何生产系统中最关键的功能。我们发现推迟设计过程中的可靠性问题等同于以更高的成本接受更少的功能。通过遵循系统设计和实现的迭代方式，我们以较低的运维成本实现了强大且可扩展的设计。我们称这种风格为**非抽象大型系统设计(NALSD)**。

### 什么是NALSD？

本章介绍了NALSD方法:我们从问题陈述开始，收集需求，并遍历越来越复杂的设计，直到找到可行的解决方案。最终，我们得到了一个可以抵御多种故障模式的系统，该系统既满足初始要求，又满足了我们反复提出的更多细节。

NALSD描述了SRE的一项关键技能:评估，设计和评估大型系统的能力。实际上，NALSD结合了容量规划，组件隔离和正常系统降级的要素，这些要素对于高可用性生产系统至关重要。期望Google SRE能够从系统的基本白板图开始资源规划，仔细考虑各种扩展和故障域，并将其设计集中于具体的资源建议。由于这些系统会随时间而变化，因此SRE能够分析和评估系统设计的关键方面至关重要。

#### 为什么要使用“非抽象”？

所有系统最终都将必须使用真实网络在真实数据中心的真实计算机上运行。Google已经了解到(艰难的方式)，设计分布式系统的人员需要开发并不断锻炼，以将白板设计转化为过程中多个步骤的具体资源估算。如果没有这种严格性，那么创建在现实世界中无法完全转换的系统就太诱人了。

前期的额外工作通常会减少最后一刻对系统设计的更改，从而解决一些无法预料的物理限制。

请注意，虽然我们将这些练习推向离散的结果(例如，机器数量)，但声音推理和假设制定的示例比任何最终值都重要。早期的假设会严重影响计算结果，而完美的假设并非NALSD的要求。这项工作的价值在于将许多不完美但合理的结果结合在一起，从而更好地理解设计。

#### AdWords示例

Google AdWords服务在Google Web搜索上显示文字广告。"点击率"(CTR)指标可告诉广告客户其广告效果如何。点击率是广告点击次数与广告点击次数之比。

这个AdWords范例旨在设计一个能够衡量和报告每个AdWords广告准确点击率的系统。我们计算点击率所需的数据记录在搜索和广告投放系统的日志中。这些日志分别记录为每个搜索查询显示的广告和被单击的广告。

### 设计过程

Google使用迭代方法来设计符合我们目标的系统。每次迭代都会定义一个潜在的设计并检查其优缺点。该分析要么进入下一次迭代，要么表明设计何时足以推荐。

概括地说，NALSD过程分为两个阶段，每个阶段都有两到三个问题。

在基本设计阶段，我们尝试发明一种"原理上可行"的设计。我们问两个问题：

**可能吗？**

设计是否可行？如果我们不必担心足够的RAM，CPU，网络带宽等等，那么我们将如何设计以满足需求？

**我们可以做得更好吗？**

对于任何此类设计，我们都会问："我们可以做得更好吗？"例如，我们可以使系统有意义地更快，更小，更高效吗？如果设计在O( $N$ )时间内解决了问题，我们可以更快地解决它吗？-例如，O( $\ln(N)$ )？

在下一阶段，我们尝试扩大我们的基本设计-例如，通过大幅增加需求。我们问三个问题：

**可行吗？**

考虑到资金，硬件等方面限制，是否可以扩展此设计？如有必要，哪种分布式设计可以满足要求？

**弹性吗？**

设计可以优雅地失败吗？该组件发生故障时会发生什么？当整个数据中心发生故障时，系统如何工作？我们可以做得更好吗？

虽然我们通常以大致的顺序涵盖这些阶段和问题，但实际上，我们会在问题和阶段之间跳来跳去。例如，在基本设计阶段，我们常常会不断增长和扩展。

然后我们进行迭代。一种设计可能成功地通过了大多数阶段，直到后来陷入困境。发生这种情况时，我们将重新开始，修改或更换组件。最终的设计是一个曲折故事的结尾。

考虑到这些概念，让我们逐步进行NALSD迭代过程。

### 初始要求

每个广告商可以具有多个广告。每个广告均由ad\_id键控，并与广告商选择的搜索字词列表相关联。

向广告客户显示仪表板时，我们需要了解每个广告和搜索字词的以下内容：

- 此搜索字词触发该广告展示的频率
- 看过广告的人点击了广告多少次

利用这些信息，我们可以计算出点击率:点击次数除以展示次数。

我们知道，广告客户关心的是两件事:仪表板快速显示，并且数据是最新的。因此，在进行设计迭代时，我们将根据SLO来考虑我们的要求(有关更多详细信息，请参见第2章):

- 99.9%的仪表板查询在<1秒内完成。
- 99.9%的时间显示5分钟内的点击率数据。

这些SLO提供了我们应该能够持续实现的合理目标。它们还提供了一个错误预算(请参阅*Site Reliability Engineering* 中的第4章)，我们将在设计的每次迭代中将我们的解决方案与之进行比较。

我们旨在创建一个既可以满足我们的SLO要求，又可以支持数百万希望在信息中心上查看其点击率的广告客户的系统。对于事务处理速率，我们假设每秒500,000个搜索查询和每秒10,000次广告点击。

### 一台机器

最简单的起点是考虑在单个计算机上运行我们的整个应用程序。

对于每个网络搜索查询，我们记录:

time

    |  查询发生的时间

query\_id

    |  唯一查询标识符(查询ID)

search\_term

    |  查询内容

ad\_id

    |  为搜索显示的所有AdWords广告的广告ID

这些信息一起构成了"查询日志"。每次用户点击广告时，我们会将点击时间，查询ID和广告ID记录在点击日志中。

您可能想知道为什么我们不只是将search\_term添加到点击日志中以降低复杂性。在我们示例的任意缩小范围内，这是可行的。但是，实际上，点击率实际上只是从这些日志计算得出的许多洞察中的一种。单击日志是从URL派生的，URL具有固有的大小限制，这使单独的查询日志成为更具可伸缩性的解决方案。与其通过在练习中添加类似CTR的额外要求来证明这一点，我们将只是承认这一假设并继续前进。

显示仪表板需要两个日志中的数据。我们需要证明我们可以实现我们的SLO，即在一秒钟内在仪表板上显示新数据。要达到此SLO，系统会处理大量的点击和查询，因此，计算CTR的速度必须保持恒定。

为了满足我们在一秒钟内显示仪表板的SLO，我们需要快速查找给定ad\_id的每个search\_term点击和显示的query\_id数量。我们可以从查询日志中提取每个search\_term 显示 query\_id和ad\_id的细目分类。点击率仪表板需要ad\_ids的查询日志和点击日志中的所有记录。

如果我们有多个广告客户，则扫描查询日志和单击日志以生成仪表板将非常低效。因此，我们的设计要求我们的一台计算机创建适当的数据结构，以便在接收日志时进行快速的点击率计算。在单台计算机上，使用在query\_id和search\_term上具有索引的SQL数据库应该能够在一秒钟内提供答案。

通过将这些日志加入query\_id并按search\_term分组，我们可以报告每次搜索的点击率。

### 计算

我们需要计算需要多少资源来解析所有这些日志。为了确定我们的扩展限制，我们需要做一些假设，从查询日志的大小开始：

time

64位整数，8字节

query\_id

64位整数，8字节

ad\_id

三个64位整数，8个字节

search\_term

长字符串，最多500个字节

### 其他元数据

500--1,000字节的信息，例如投放广告的机器，搜索所用的语言以及搜索项返回的结果数

为了确保我们不会过早达到限制，我们积极地将每条查询日志条目视为2 KB。点击日志量应大大小于查询日志量：由于平均点击率是2%(10,000次点击 / 500,000次查询)，因此点击日志的记录数将是查询日志的2%。请记住，我们选择了大量的数字来说明这些原理可以扩展到任意大型实现。这些估计值似乎很大，因为它们应该是。

最后，我们可以使用科学计数法来限制由不一致单元上的算术引起的错误。24小时内生成的查询日志的数量为：

$(5 \times 10^5 \text{ 查询/秒}) \times (8.64 \times 10^4 \text{ 秒/天}) \times (2 \times 10^3 \text{ 字节}) = 86.4 \text{ TB/天}$

因为我们获得的点击次数是查询的2%，并且我们知道数据库索引会增加一些合理的开销，所以我们可以将每天86.4 TB的空间四舍五入到存储一天的日志数据所需的100 TB空间。

由于总存储需求为~100 TB，我们需要做出一些新的假设。这种设计是否仍可以在单台机器上使用？虽然可以将100 TB的磁盘连接到一台计算机，但是我们很可能受到该计算机读取和写入磁盘的能力的限制。

例如，一个普通的4 TB HDD可能每秒能够维持200个输入/输出操作(IOPS)。如果每个日志条目可以平均每个日志条目存储一个磁盘并为其建立索引，那么我们会发现IOPS是查询日志的限制因素：

$(5 \times 10^5 \text{ 查询/秒}) / (200 \text{ IOPS/磁盘}) = 2.5 \times 10^3 \text{ 磁盘或} 2,500 \text{ 磁盘}$

即使我们可以以10:1的比例批量查询以限制操作，在最佳情况下，我们也需要数百个HDD。考虑到查询日志写入只是设计IO要求的一个组成部分，我们需要使用一种比传统HDD更好地处理高IOPS的解决方案。

为简单起见，我们将直接评估RAM，而跳过对其他存储介质(如固态磁盘(SSD))的评估。一台机器无法完全在RAM中处理100 TB的占用空间:假设我们拥有16核，64 GB RAM和1 Gbps网络吞吐量的标准机器占用空间，则需要:

$$(100 \text{ TB}) / (64 \text{ GB RAM} / \text{计算机}) = 1,563 \text{ 台计算机}$$

### 评估

暂时忽略我们的计算，并想象我们可以将这种设计安装在一台机器上，我们真的想要吗？如果我们通过询问“该组件发生故障时会发生什么情况”来测试我们的设计，我们会确定一长串单点故障(例如，CPU，内存，存储，电源，网络，散热)。如果这些组件之一发生故障，我们是否可以合理地支持我们的SLO？几乎可以肯定，即使是简单的电源重启也会严重影响我们的用户。

回到我们的计算中，我们的单机设计再次显得不可行，但是这一步并没有浪费时间。我们发现了关于如何合理解释系统约束及其初始要求的宝贵信息。我们需要改进设计以使用多台机器。

### 分布式系统

我们需要的search\_terms在查询日志中，而ad\_ids在点击日志中。既然我们知道我们将需要多台机器，那么加入它们的最佳设计是什么？

#### MapReduce

我们可以使用[MapReduce](#)处理和连接日志。我们可以定期获取累积的查询日志和单击日志，然后MapReduce将产生一个由ad\_id整理的数据集，该数据集显示每个search\_term收到的点击次数。

MapReduce用作批处理程序:其输入是一个大型数据集，它可以使用许多计算机通过工作程序处理该数据并产生结果。所有机器处理完数据后，就可以将其输出进行合并-MapReduce可以直接为每个AdWords广告和搜索词创建每个点击率的摘要。我们可以使用这些数据来创建所需的仪表板。

**评估** MapReduce是一种广泛使用的计算模型，我们相信它将水平扩展。无论我们的查询日志和单击日志输入有多大，添加更多计算机都将始终使该过程成功完成，而不会耗尽磁盘空间或RAM。

不幸的是，这种类型的批处理无法在收到日志后的5分钟内达到我们的SLO加入日志可用性。为了在5分钟内提供结果，我们需要小批量运行MapReduce作业-一次只需记录几分钟。批次的任意性和非重叠性使小批量不切实际。如果一个记录的查询在批次1中，并且其单击在批次2中，则该单击和查询将永远不会合并。虽然MapReduce可以很好地处理独立的批次，但并未针对此类问题进行优化。此时，我们可以尝试使用MapReduce找出潜在的解决方法。但是，为简单起见，我们将继续研究另一种解决方案。

#### LogJoiner

用户点击的广告数量明显少于投放的广告数量。直观地讲，我们需要专注于扩展两者中的较大者:查询日志。为此，我们引入了一个新的分布式系统组件。

与其像MapReduce设计中那样小批量查找query\_id，如果我们创建了所有查询的存储库，可以按需通过query\_id查找该查询，该怎么办？我们将其称为QueryStore。它包含查询日志的完整内容，并以query\_id为关键字。为避免重复，我们将假设单机设计中的计算将应用于QueryStore，并将对QueryStore的审查限于我们已经介绍的内容。有关此类组件如何工作的更深入讨论，建议阅读有关Bigtable的文章。<sup>72</sup>

由于点击日志也具有query\_id，因此我们的处理循环的规模现在要小得多：它只需要遍历点击日志并拉入引用的特定查询。我们将此组件称为LogJoiner。

LogJoiner从点击日志中获取连续的数据流，将其与QueryStore中的数据结合起来，然后存储该信息（按ad\_id进行组织）。一旦点击的查询被ad\_id存储并建立索引，我们就拥有生成点击率信息中心所需数据的一半。我们将其称为ClickMap，因为它是从ad\_id映射到点击次数的。

如果找不到点击的查询（接收查询日志可能会变慢），我们将其搁置一段时间，然后重试，直到时限。如果在该时间限制之前找不到查询，我们将放弃该点击。

对于每个ad\_id和search\_term对，CTR仪表板都需要两个组件：展现数和点击广告数。ClickMap需要一个合作伙伴来保存按ad\_id进行组织的查询。我们将其称为QueryMap。QueryMap直接从查询日志中获取所有数据，并通过ad\_id为条目建立索引。

图12-1描述了数据如何流经系统。

LogJoiner设计引入了几个新组件：LogJoiner、QueryStore、ClickMap和QueryMap。我们需要确保这些组件可以扩展。

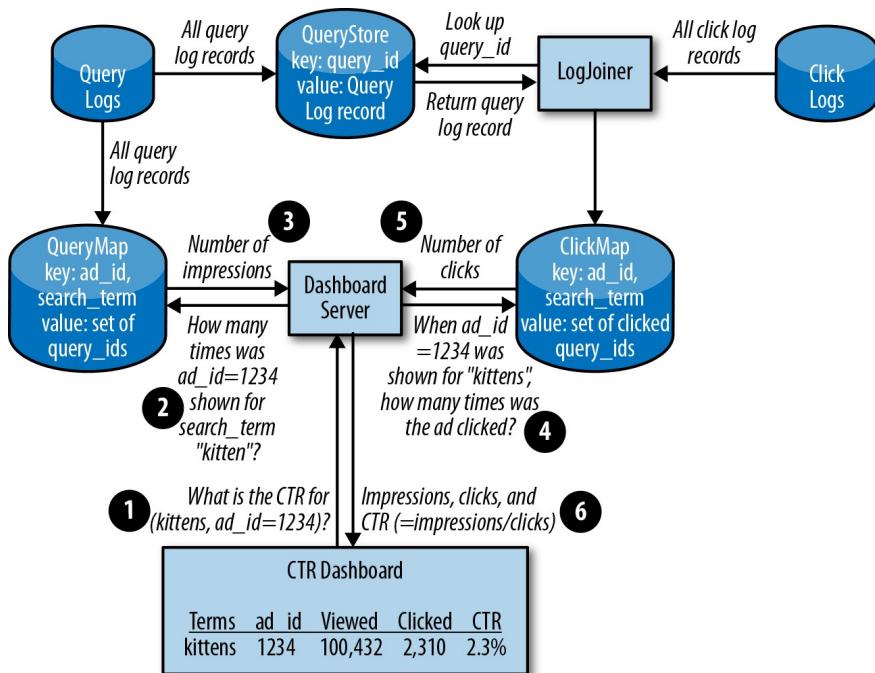


图12-1. 基本的LogJoiner设计；点击数据得到处理和存储，以便仪表板可以检索它

**计算** 根据我们在先前迭代中执行的计算，我们知道QueryStore在一天的日志中将有大约100 TB的数据。我们可以删除太旧而没有价值的数据。

LogJoiner应该在单击时处理单击，并从QueryStore检索相应的查询日志。

LogJoiner处理日志所需的网络吞吐量取决于我们日志中每秒的点击次数，再乘以2 KB记录大小：

$$(10^4 \text{ 点击/秒}) \times (2 \times 10^3 \text{ 字节}) = 2 \times 10^7 = 20 \text{ Mb/秒} = 160 \text{ Mbps}$$

QueryStore查找会导致额外的网络使用情况。对于每个点击日志记录，我们查找 query\_id 并返回完整的日志记录：

- $(10^4 \text{ 点击/秒}) \times (8 \text{ 字节}) = 8 \times 10^4 = 80 \text{ Kb/秒} = 640 \text{ Kbps}$
- $(10^4 \text{ 单击/秒}) \times (2 \times 10^3 \text{ 字节}) = 2 \times 10^7 = 20 \text{ Mb/秒} = 160 \text{ Mbps}$

LogJoiner还将结果发送到ClickMap。我们需要存储query\_id，ad\_id和时间。搜索词。time和query\_id均为64位整数，因此数据将小于1 KB：

$$(10^4 \text{ 点击/秒}) \times (10^3 \text{ 字节}) = 10^7 = 10 \text{ Mb/秒} = 80 \text{ Mbps}$$

总计~400 Mbps是我们机器的可控制数据传输速率。

ClickMap必须存储每次单击的时间和query\_id，但不需要任何其他元数据。我们将忽略ad\_id和search\_term，因为它们是很小的线性因子(例如，广告商数量×广告数量×8字节)。即使是1000万个广告客户，每个广告有10个，也只有~800 MB。一天的ClickMap价值为：

$$(10^4 \text{ 次点击/秒}) \times (8.64 \times 10^4 \text{ 秒/天}) \times (8 \text{ 字节} + 8 \text{ 字节}) = 1.4 \times 10^{10} = 14 \text{ GB/天}$$

我们会将ClickMap的舍入额提高到每天20 GB，以解决所有开销和我们的ad\_id。

填写QueryMap时，我们需要为显示的每个广告存储query\_id。我们的存储需求有所增加，因为每个搜索查询都可能会点击三个ad\_id，因此我们最多需要在三个条目中记录query\_id：

$$3 \times (5 \times 10^5 \text{ 查询/秒}) \times (8.64 \times 10^4 \text{ 秒/天}) \times (8 \text{ 字节} + 8 \text{ 字节}) = 2 \times 10^{11} \text{ TB/天} \overset{73}{=} 2$$

2 TB足够小，可以在使用HDD的单台计算机上托管，但是从单机迭代中我们知道，单个写操作太频繁而无法存储在硬盘驱动器上。虽然我们可以计算使用更高IOPS驱动器(例如SSD)的影响，但我们的练习重点是演示系统可以扩展到任意大的尺寸。在这种情况下，我们需要围绕一台计算机的IO限制进行设计。因此，扩展设计的下一步是“分片”输入和输出：将传入的查询日志和单击日志分成多个流。

### 分片的LogJoiner

我们在此迭代中的目标是运行多个LogJoiner实例，每个实例都在不同的数据分片上。<sup>72</sup>为此，我们需要考虑以下几个因素：

#### 数据管理

要加入查询日志和单击日志，我们必须将每个单击日志记录与其在query\_id上的相应查询日志记录进行匹配。该设计应防止网络和磁盘吞吐量随扩展规模而限制我们的设计。

#### 可靠性

我们知道机器随时可能发生故障。当运行LogJoiner的计算机发生故障时，我们如何确保我们不会丢失正在进行的工作？

### 效率

我们可以在没有浪费的情况下扩大规模吗？我们需要使用最少的资源来满足我们对数据管理和可靠性的关注。

我们的LogJoiner设计表明，我们可以连接我们的查询日志和单击日志，但是生成的数据量非常大。如果我们根据query\_id将工作划分为多个分片，则可以并行运行多个LogJoiners。

如果提供了合理数量的LogJoiner实例，则如果我们平均分配日志，则每个实例只能通过网络接收一小部分信息。随着点击流量的增加，我们通过添加更多LogJoiner实例来水平扩展，而不是通过使用更多的CPU和RAM来垂直扩展。

如图12-2所示，为了使LogJoiners接收正确的消息，我们引入了一个称为*log sharder*的组件，该组件会将每个日志条目定向到正确的目的地。对于每条记录，我们的点击日志分片都将执行以下操作：

1. 散列记录的query\_id。
2. 用N(分片数)对结果取模，然后加1以得到1到 N 之间的数字。
3. 将记录发送到步骤2中的分片号。

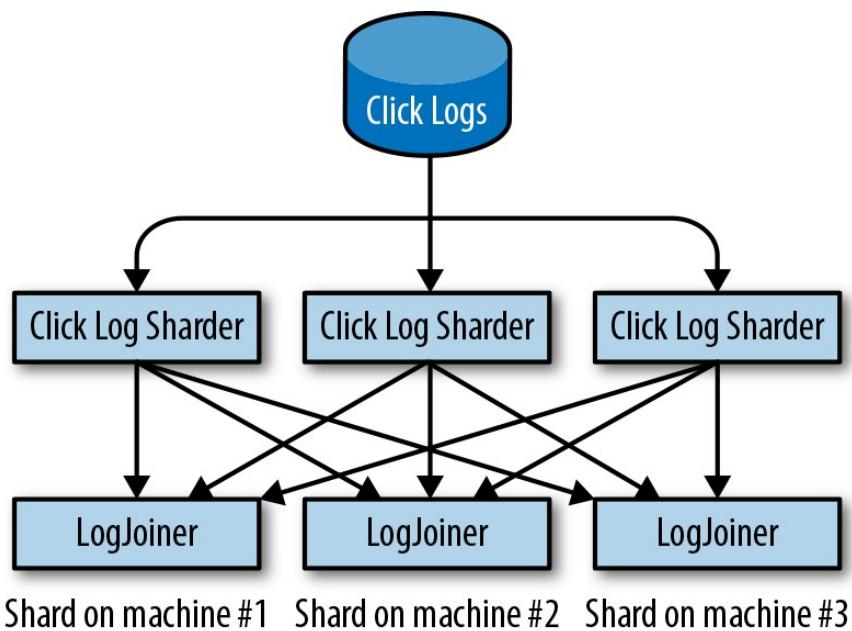


图12-2. 分片应如何工作？

现在，每个LogJoiner都将获得由query\_id分解的传入日志的一致子集，而不是完整单击日志。

QueryMap也需要分片。我们知道，要维持QueryMap所需的IOPS，将需要许多硬盘驱动器，而且一天的QueryMap(2 TB)的大小对于我们的64 GB计算机而言无法存储在RAM中。但是，我们不像LogJoiner那样通过query\_id进行分片，而是对ad\_id进行分片。ad\_id在任何读取或写入之前就已知道，因此使用与LogJoiner和CTR仪表板相同的哈希方法将提供一致的数据视图。

为了使实现保持一致，我们可以将ClickMap的日志分片设计与QueryMap重复使用，因为ClickMap小于QueryMap。

现在我们知道我们的系统可以扩展，我们可以继续解决系统的可靠性。我们的设计必须能够抵抗LogJoiner故障。如果LogJoiner在收到日志消息之后但在加入日志消息之前失败，则必须重做其所有工作。这会延迟准确数据到达仪表板的时间，这将影响我们的SLO。如果我们的日志分片程序将重复的日志条目发送到两个分片，则即使LogJoiner发生故障(可能是因为其所在的计算机发生故障)，系统仍可以继续全速运行并处理准确的结果。

通过以这种方式复制工作，我们减少(但不消除)丢失那些合并日志的机会。两个碎片可能会同时断裂并丢失连接的日志。通过分配工作量以确保没有重复的碎片落在同一台计算机上，我们可以减轻很多这种风险。如果两台机器同时发生故障，并且我们丢失了这两个分片的副本，则系统的错误预算(请参阅第一本SRE手册中的[第4章](#))可以弥补其余的风险。确实发生灾难时，我们可以重新处理日志。在短暂的时间范围内，仪表板只会显示比5分钟还早的数据。

图12-3显示了我们对分片及其副本的设计，其中LogJoiner，ClickMap和QueryMap都建立在这两个分片上。

从连接的日志中，我们可以在每个LogJoiner计算机上构造一个ClickMap。为了显示我们的用户仪表板，所有ClickMap都需要组合和查询。

**评估**将分片组件托管在一个数据中心中会造成单点故障：如果恰巧不幸机器对或数据中心断开连接，我们将失去所有ClickMap的工作，并且用户仪表板将完全停止工作！我们需要改进设计以使用多个数据中心。

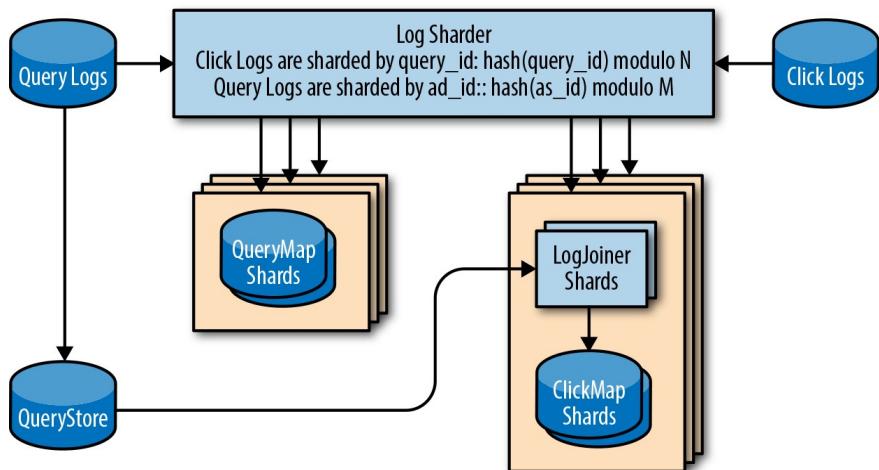


图12-3. 具有相同query\_id的日志分片以复制分片

### 多数据中心

在不同地理位置的数据中心之间复制数据使我们的服务基础架构能够承受灾难性的故障。如果一个数据中心宕机(例如，由于一天的停电或网络中断)，我们可以故障转移到另一个数据中心。为使故障转移正常工作，ClickMap数据必须在部署系统的所有数据中心中都可用。

这样的ClickMap可能吗？我们不想将计算需求乘以数据中心的数量，但是如何才能有效地同步站点之间的工作以确保足够的复制而又不产生不必要的重复？

我们刚刚描述了分布式系统工程中众所周知的[consensus](#)问题的示例。有许多解决此问题的复杂算法，但是基本思想是：

1. 制作三个或五个您要共享的服务的副本(如ClickMap)。

2. 让副本使用共识算法，例如Paxos，以确保如果发生数据中心大小的故障，我们可以可靠地存储计算状态。
3. 在参与节点之间实现至少一个网络往返时间以接受写操作。此要求限制了系统的顺序吞吐量。我们仍然可以并行处理对分布式基于共识的映射的某些写操作。

按照上面列出的步骤，现在原则上看来多数据中心设计是可行的。它也可以在实践中起作用吗？我们需要什么类型的资源，我们需要多少？

**计算** 使用故障隔离的数据中心执行Paxos算法的等待时间意味着完成每个操作大约需要25毫秒。这种延迟假设是基于至少相距数百公里的数据中心。因此，就顺序处理而言，我们每个操作只能执行一个操作

25毫秒或每秒40次操作。如果我们需要每秒执行 $10^4$  次的连续处理(单击日志)，则每个数据中心至少需要250个处理，并由ad\_id进行分拣，以进行Paxos操作。实际上，我们希望添加更多的流程来提高并行度-在任何停机时间或流量高峰后处理累积的积压。

基于我们先前对ClickMap和QueryMap的计算，并使用每秒40次连续操作的估计，我们的多数据中心设计需要多少台新机器？

因为我们经过分派的LogJoiner设计为每个日志记录引入了一个副本，所以我们将每秒事务数增加了一倍，以创建ClickMap和QueryMap:每秒20,000次点击和每秒1,000,000次查询。

我们可以通过将每秒的总查询数除以每秒的最大操作数来计算所需的最小进程数或"任务"数: $(1.02 \times 10^6 \text{ 查询/秒}) / (40 \text{ 操作/秒}) = 25,500 \text{ 个任务}$

每个任务的内存量(2 TB QueryMap的两个副本):

$$(4 \times 10^{12} \text{ 字节}) / (25,500 \text{ 个任务}) = 157 \text{ MB/任务}$$

每台计算机的任务:

$$(6.4 \times 10^{10} \text{ 字节}) / (1.57 \times 10^8 \text{ 字节}) = 408 \text{ 个任务/机器}$$

我们知道我们可以在一台计算机上完成许多任务，但是我们需要确保不会出现IO瓶颈。ClickMap和QueryMap的总网络吞吐量(使用每个条目2 KB的高估算值):

$$(1.02 \times 10^6 \text{ 查询/秒}) \times (2 \times 10^3 \text{ 字节}) = 2.04 \text{ Gb/秒} = 16 \text{ Gbps}$$

每个任务的吞吐量:

$$16 \text{ Gbps} / 25,500 \text{ 个任务} = 80 \text{ KB/秒} = 640 \text{ Kbps/任务}$$

每台机器的吞吐量:

$$408 \text{ 个任务} \times 640 \text{ Kbps/任务} = 256 \text{ Mbps}$$

157 MB内存和640 Kbps每个任务的组合是可管理的。每个数据中心大约需要4 TB RAM，以托管分片的ClickMap和QueryMap。如果每台计算机有64 GB的RAM，那么我们只能从64台计算机提供数据，并且将仅使用每台计算机25%的网络带宽。

**评估** 现在我们已经设计了一个多数据中心系统，让我们回顾一下数据流是否有意义。

图12-4显示了整个系统设计。您可以查看如何将每个搜索查询和广告点击传达给服务器，以及如何收集日志并将其推送到每个组件中。

我们可以根据我们的要求检查该系统：

**每秒10,000次广告点击**

LogJoiner可以水平缩放以处理所有日志单击，并将结果存储在ClickMap中。

**每秒500,000个搜索查询**

QueryStore和QueryMap被设计为以这种速率处理一整天的数据。

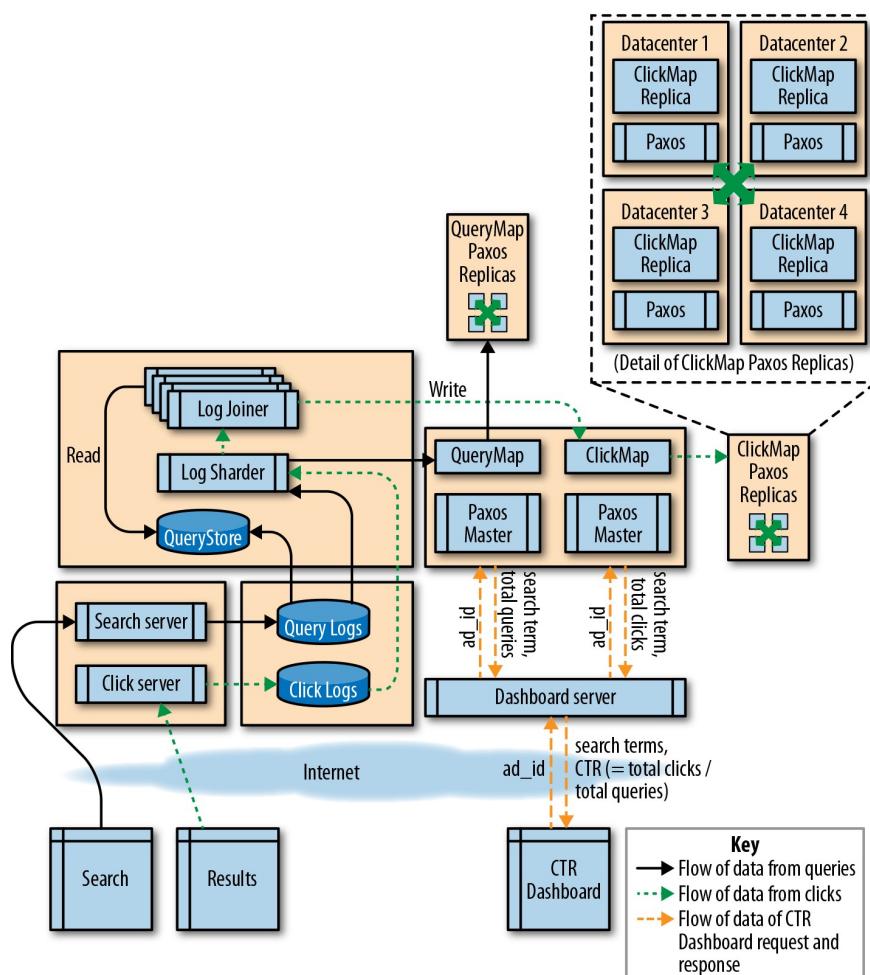
- 99.9%的仪表板查询在<1秒内完成\*

CTR仪表板从QueryMap和ClickMap中获取数据，这些数据以ad\_id为键，从而使该事务变得快速而简单。

- 99.9%的时间显示的点击率数据不到5分钟\*

每个组件均设计为水平扩展，这意味着如果流水线速度太慢，则添加更多计算机将减少端到端流水线延迟。

我们相信这种系统架构可以扩展以满足我们对吞吐量，性能和可靠性的要求。



## 总结

NALSD描述了Google用于生产系统的系统设计的迭代过程。通过将软件分解为逻辑组件，并将这些组件放入具有可靠基础架构的生产生态系统中，我们得到了为数据一致性，系统可用性和资源效率提供合理和适当目标的系统。NALSD的实践使我们能够改进设计，而不必在各种各样的设计迭代中为每次迭代重新开始。

本章介绍的内容满足了我们最初的问题陈述，每次迭代都揭示了新的要求，可以通过扩展以前的工作来满足这些要求。

在整个过程中，我们根据对系统增长的期望来分离软件组件。这种策略使我们能够独立扩展系统的不同部分，并消除了对单个硬件或单个软件实例的依赖，从而产生了一个更可靠的系统。

在整个设计过程中，我们继续通过询问NALSD的四个关键问题来改进每次迭代：

可能吗？

—— 我们能在没有“魔法”的情况下建造它吗？

我们可以做得更好吗？

—— 它是否尽我们所能地简单？

可行吗？

—— 是否符合我们的实际限制(预算，时间等)？

弹性吗？

—— 它会在偶尔但不可避免的中断中生存吗？

NALSD是一项博学的技能。与任何技能一样，您需要定期练习以保持您的熟练程度。Google的经验表明，从抽象需求到具体资源近似的推理能力对于构建健康且寿命长的系统至关重要。

<sup>72</sup>. Fay Chang等人，“大表：一种用于结构化数据的分布式存储系统”，《计算机系统上的ACM事务处理》(TOCS)26，第2号(2008)，<http://bit.ly/2J22BZv>。 ↵

<sup>73</sup>. 本部分基于Rajagopal Ananthanarayanan等人的“光子：连续数据流的容错和可伸缩联接”，在SIGMOD '13中：2013 ACM SIGMOD国际数据管理国际会议论文集(纽约：ACM，2013年)，<http://bit.ly/2Jse3Ns>。 ↵

# 第13章

## 数据处理管道

由Rita Sodt和Igor Maravić(Spotify)与Gary Luo , Gary O'Connor和Kate Ward合作撰写

数据处理是一个复杂的领域，它不断发展以满足更大的数据集，大量的数据转换以及对快速，可靠和廉价的结果的需求。当前的格局具有从各种来源生成和收集的数据集-从移动使用情况统计信息到集成的传感器网络再到Web应用程序日志，等等。数据处理管道可以将这些通常无限制，无序的全球规模的数据集转变为结构化的索引存储，可以帮助您确定重要的业务决策或解锁新产品功能。除了提供对系统和用户行为的洞察力之外，数据处理通常还对业务至关重要。管道中的数据延迟或不正确可能会显示在用户界面上，这些问题既昂贵，费力又费时，而且修复起来很费时间。本章从使用产品示例开始，以研究大数据处理管道的一些常见应用程序类型。然后，我们探索如何确定管道需求和设计模式，并列举一些在整个开发生命周期中管理数据处理管道的最佳实践。我们将介绍您可以进行的权衡，以优化管道以及测量管道运行状况重要信号的技术。为了使服务在部署后保持健康和可靠，SRE(以及开发人员)应该能够浏览所有这些任务。理想情况下，SRE应该从早期开始就参与这项工作:Google的SRE小组会定期与开发数据处理管道的小组进行磋商，以确保可以轻松发布，修改和运行该管道而不会对客户引起任何问题。

最后，Spotify案例研究概述了其事件传递处理管道，该管道使用内部，Google Cloud和其他第三方解决方案的组合来管理复杂的，对业务至关重要的数据处理管道。无论您是直接掌控管道，还是掌控其他依赖于管道产生的数据的服务，我们都希望您可以使用本章中的信息来帮助使管道(和服务)更加可靠。

有关Google关于数据处理管道的哲学的全面讨论，请参阅我们的第一本SRE书的[第25章](#)。

## 管道应用

管道应用程序种类繁多，每种都有其自身的优势和用例。流水线可以涉及多个阶段。每个阶段都是一个独立的过程，依赖于其他阶段。一个管道可能包含多个阶段，这些阶段通过一个高级规范被抽象出来。例如，在Cloud Dataflow中:用户使用相对高级的API编写业务逻辑，[管道技术](#)本身将这些数据转换为一系列步骤，或者一个人的输出是另一个人的输入的阶段。为了让您大致了解管道应用程序的广度，接下来我们介绍几种管道应用程序及其推荐用途。我们使用两家具有不同管道和实施要求的示例公司来演示满足其各自数据需求的不同方法。这些示例说明您的特定用例如何定义项目目标，以及如何使用这些目标就哪种数据管道最适合您做出明智的决定。

[订单或结构数据的事件处理/数据转换](#)

提取转换加载(ETL)模型是数据处理中的常见范例:从源中提取数据，对其进行转换并可能进行非规范化，然后“重新加载”为专用格式。在更现代的应用程序中，这可能看起来像一个认知过程:从某种传感器(实时或回放)中获取数据以及选择和编组阶段，然后“训练”专用数据结构(例如机器学习网络)。

ETL管道以类似的方式工作。从单个(或多个)源中提取数据，进行转换，然后将其加载(或写入)到另一个数据源中。转换阶段可以满足各种用例，例如:

- 更改数据格式以添加或删除字段
- 跨数据源聚合计算功能
- 将索引应用于数据，以便为使用数据的作业提供更好的特性

通常，ETL管道会准备数据以进行进一步分析或提供服务。如果使用正确，ETL管道可以执行复杂的数据操作，并最终提高系统效率。ETL管道的一些示例包括:

- 机器学习或商业智能用例的预处理步骤
- 计算，例如计算给定事件类型在特定时间间隔内发生了多少次
- 计算和准备帐单报告
- 为Google的网络搜索提供支持的索引管道

## 数据分析

商业智能是指用于收集，集成，分析和呈现大量信息以做出更好决策的技术，工具和做法。<sup>74</sup>无论您是零售产品，手机游戏还是物联网传感器跨多个用户或设备汇总数据，可以帮助您确定发生故障或运行良好的位置。

为了说明数据分析用例，让我们研究一个虚构的公司及其最近发布的手机和网络游戏*Shave the Yak*。拥有者想知道他们的用户如何在他们的移动设备和网络上与游戏进行交互。第一步，他们生成游戏的数据分析报告，以处理有关玩家事件的数据。该公司的业务负责人要求提供有关该游戏最常用功能的每月报告，以便他们可以计划新功能开发并进行市场分析。游戏的移动分析和网络分析存储在Google Cloud BigQuery表中，该表每天由Google Analytics更新三次。团队设置了一个作业，每当将新数据添加到这些表时，该作业便会运行。完成后，该作业将在公司的每日汇总表中输入。

## 机器学习

机器学习(ML)应用程序用于多种目的，例如帮助预测癌症，对垃圾邮件进行分类以及为用户提供个性化的产品推荐。通常，ML系统具有以下阶段:

1. 数据特征及其[标签](#)从较大的数据集中提取。
2. ML算法在提取的特征上训练模型。
3. 在测试数据集上评估模型。
4. 该模型可用于(服务)其他服务。
5. 其他系统使用模型提供的响应来做出决策。

为了演示实际操作中的ML管道，让我们考虑一个虚构的公司Dressy的示例，该公司在线销售礼服。该公司希望通过向用户提供有针对性的建议来增加收入。当新产品上载到网站时，Dressy希望他们的系统在12小时内开始将该产品纳入用户推荐

中。最终，Dressy希望向用户提供与网站互动并为着装评分的近实时建议。作为他们推荐系统的第一步，Dressy研究以下方法：

#### 协作

显示彼此相似的产品。

#### 分类

显示相似用户喜欢的产品。

#### 基于内容

显示与用户浏览或喜欢的其他产品相似的产品。

作为在线商店，Dressy具有用户配置文件信息和评论的数据集，因此他们选择使用聚类过滤器。上载到系统中的新产品没有结构化数据或一致的标签(例如，某些供应商可能会使用不同的类别和格式提供有关衣服的颜色，尺寸和功能的额外信息)。因此，他们需要实施管道以将数据预处理为与TensorFlow兼容的格式，该格式将产品信息和用户个人资料数据连接在一起。ML系统包括用于预处理训练模型所需的多个来源的数据的管道。Dressy的开发团队使用培训数据创建TensorFlow模型，以向客户提供适当的建议。图13-1显示了完整的ML解决方案。之后，我们将详细说明每个步骤。

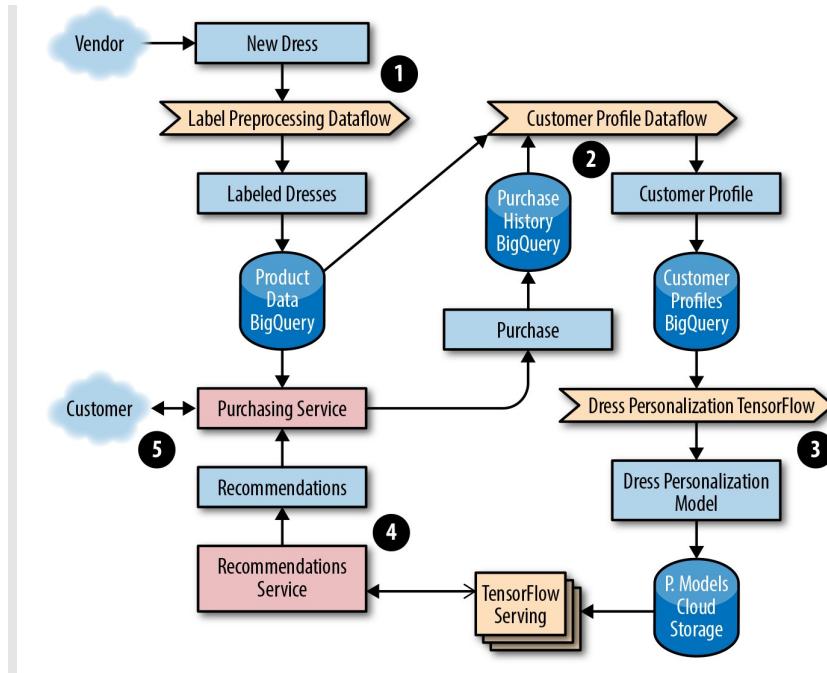


图13-1.ML数据处理管道

1. 开发团队选择使用流数据流管道Google Cloud Dataflow，通过将图像发送到返回特征列表的图像分类服务，将数据预处理为带标签的连衣裙格式。
2. 团队对来自多个来源的数据进行预处理，这些数据将用于训练一个模型，该模型返回前五名最相似的服饰。他们的工作流程根据服装产品数据和来自BigQuery中存储的客户资料的购买历史记录生成ML模型。
3. 他们选择使用流数据流管道将数据预处理为客户个性化配置文件的格式。这些配置文件用作训练TensorFlow模型的输入。经过训练的TensorFlow模型二进制文件存储在Google Cloud Storage(GCS)存储桶中。在推广到生产之前，该团

队确保针对用于模型评估的预处理数据的测试集进行评估时，模型可以通过准确性检查。

4. 服务为网络和移动前端使用的给定客户提供建议。团队将TensorFlow与[Cloud ML](#)在线预测服务一起使用。

1. 用于购买的面向客户的前端服务根据来自预测服务的最新着装建议为用户数据提供服务。

Dressy已注意到，新模型偶尔会在24小时内不会发布，并且建议会触发间歇性错误。首次部署新模型时，这是一个常见问题。但是，您可以采取一些简单的步骤来解决此问题。如果您开始发现决策，分类或建议没有浮出水面，陈旧或不正确，请问自己：

- 在对数据进行预处理以训练模型之前，数据是否一直进入管道？
- 我们是否由于软件错误而导致ML模型不佳？有很多垃圾邮件吗？用于训练模型的功能选择不正确吗？
- 最近是否生成了ML模型的新版本，或者该模型的过时版本正在生产中运行？

幸运的是，Dressy有一套工具可以在客户遇到任何问题之前对其进行监控和检测。如果发生中断，这些工具可以帮助他们快速修复或回滚任何有问题的代码。有关实施监控和警报的更多详细信息，请参阅第4章。

### 管道最佳实践

以下管道最佳实践适用于作为服务运行的管道(即负责及时正确处理数据以供其他系统使用的管道)。要正确地将管道作为服务部署，需要执行多个步骤。这些步骤包括从使用SLO定义和评估客户需求到从容应对降级和故障，编写文档并创建开发生命周期以在问题到达生产之前就发现问题，这些步骤就可以了。

### 定义和衡量服务水平目标

自动检测管道何时不正常以及是否无法满足客户需求非常重要。当您有可能超出错误预算的危险时接收通知(有关错误预算的更多详细信息，请参阅第2章)有助于最大程度地减少对客户的影响。同时，重要的是要在可靠性和功能发布之间取得舒适的平衡-您的客户关心这两者。本节的其余部分提供了管道SLO的示例以及如何维护它们。

### 数据新鲜度

大多数管道数据新鲜度SLO均采用以下格式之一：

- X%的数据以Y[秒,天,分钟]处理。
- 最旧的数据不超过Y[秒,天,分钟]。
- 管道作业已在Y[秒,天,分钟]内成功完成。

例如，*Shave the Yak*手机游戏可以选择以SLO为目标，要求在30分钟内将影响用户分数的所有影响用户分数的用户动作的99%反映在记分板上。

### 数据正确性

创建SLO以确保数据正确性可确保在管道中收到有关潜在数据错误的警报。例如，计费管道中的正确性错误可能导致向客户收取过多或过少的费用。正确性目标可能很难衡量，尤其是在没有预定的正确输出的情况下。如果您无权访问此类数据，

则可以生成它。例如，使用测试帐户来计算预期的输出。一旦有了这些“黄金数据”，就可以比较预期和实际输出。从那里，您可以创建对错误/差异的监控，并在测试数据流经实际生产系统时实施基于阈值的警报。

另一个数据正确性SLO涉及向后分析。例如，您可以设置一个目标，即每季度不超过0.1%的发票不正确。您可以为从管道输出数据提供错误数据或错误的小时/天数设置另一个SLO目标。数据正确性的概念因产品和应用程序而异。

### 数据隔离/负载平衡

有时，您将拥有优先级较高的数据段或需要更多资源来处理的数据段。如果您承诺在高优先级数据上采用更严格的SLO，则很重要的一点是，如果资源受到限制，则必须先处理此数据，然后再处理低优先级数据。此实现因管道而异，但通常表现为基于任务的系统中的不同队列或不同的作业。可以将管道worker配置为承担最高优先级的任务。也就是有多个使用不同资源配置(例如内存，CPU或[网络层](#))运行的管道应用程序和管道工作程序在配置较低的worker上无法成功，则可以使用配置较高的worker重试。在资源或系统受限的情况下，当不可能快速处理所有数据时，这种分离使您可以优先处理优先级较高的项目，而不是优先级较高的项目。

### 端到端测量

如果您的管道有一系列阶段，则可能很想测量每个阶段或每个组件的SLO。但是，以这种方式测量SLO并不能捕获客户的经验或系统的端到端运行状况。例如，假设您有一个基于事件的管道，例如Google Analytics(分析)。端到端SLO包括日志输入收集以及在数据达到服务状态之前发生的任何数量的管道步骤。您可以单独监控每个阶段并在每个阶段提供一个SLO，但是客户只关心SLO的所有阶段之和。如果要测量每个阶段的SLO，则将不得不加强每个组件的警报，这可能会导致出现更多无法模拟用户体验的警报。

此外，如果仅按每个阶段衡量数据的正确性，则可能会错过端到端数据损坏错误。例如，管道中的每个阶段都可以报告一切正常，但是一个阶段会引入一个字段，期望下游作业可以处理。此上游阶段假定多余的数据已被处理并用于向用户提供请求。下游作业不需要附加字段，因此会删除数据。两项工作都认为它们是正确的，但是用户看不到数据。

### 依赖失败计划

定义SLO后，优良作法是确认您不会过分依赖于其他无法满足其承诺的产品的SLO/SLA。许多产品(例如Google Cloud Platform)都在其网站上列出了其[SLA承诺](#)。一旦确定了任何第三方依存关系，就至少应针对其公布的SLA中说明的最大故障进行设计。例如，在定义SLO时，向Cloud Storage读取或写入数据的管道的所有者将确保[广告发布的正常运行时间和保证](#)合适。如果单区域正常运行时间保证少于管道满足其SLO的数据处理时间要求，则管道所有者可以选择跨区域复制数据以获得更高的可用性。

当服务提供商的基础架构中断其SLA时，结果可能会对依赖的管道产生负面影响。如果您的管道依赖于比服务提供商更严格的保证，则即使服务提供商仍在其SLA中，您的服务也可能会失败。有时，切实地为依赖项失败进行计划可能意味着接受较低级别的可靠性并为您的客户提供更宽松的SLA。[75](#)

在Google，为了鼓励考虑到依赖失败的管道开发，我们安排了计划内的中断。例如，Google的许多管道取决于它们运行所在的数据中心的可用性。我们的灾难恢复测试(DiRT)经常针对这些系统，以模拟区域中断。发生区域中断时，计划进行故障

处理的管道会自动故障转移到另一个区域。其他管道将延迟，直到发生故障的管道的操作员受到监控警报并手动进行故障转移为止。成功的手动故障转移假定管道可以获取足够的资源以在另一个区域启动生产堆栈。在最佳情况下，不成功的手动故障转移会延长中断时间。在最坏的情况下，处理作业可能会继续处理过时的数据，这会在任何下游管道中引入过时或不正确的数据。此类事件的恢复策略因您的设置而异。例如，如果正确的数据被错误的数据覆盖，则可能必须从以前的备份版本还原数据并重新处理所有丢失的数据。

总而言之，良好的做法是为您所依赖的系统不可用的那一天做好准备。即使是最好的产品也会失败，并遭受停运。定期练习灾难恢复方案，以确保您的系统能够应对常见和罕见的故障。评估您的依赖关系并尽可能自动执行系统响应。

### 创建和维护管道文档

如果编写和维护得当，系统文档可以帮助工程师可视化数据管道及其依赖项，了解复杂的系统任务，并有可能缩短停机时间。我们建议您的管道使用三类文档。

### 系统图

系统图类似于图13-2，可以帮助值班工程师快速找到潜在的故障点。在Google，我们鼓励团队绘制系统图来显示每个组件(管道应用程序和数据存储)，以及每个步骤发生的转换。图表中显示的每个组件和转换都可能卡住，从而导致数据停止流经系统。每个组件还可能引入影响数据正确性的软件或应用程序配置错误。

系统图应包含指向不同管道阶段的其他监控和调试信息的快速链接。理想情况下，这些链接应从实时监控信息中提取，以显示每个阶段的当前状态(例如，等待相关作业完成/处理/完成)。显示历史运行时信息还可以指示管道阶段花费的时间是否比预期的长。这种延迟可能预示性能下降或中断。

最后，即使在复杂的系统中，系统图也使开发人员可以更轻松地分析功能启动期间应了解的数据依赖性。

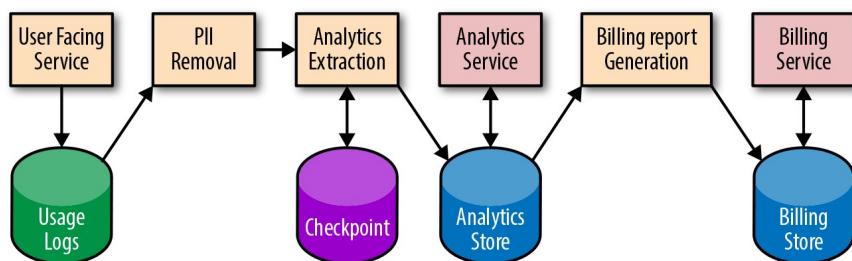


图13-2. 管道系统图(PII = 个人身份信息)

### 过程文档

重要的是记录如何执行常见任务，例如发布新版本的管道或对数据格式进行更改。理想情况下，您还应该记录一些不太常见(通常是手动)的任务，例如在新区域中进行初始服务启动或最终服务关闭。记录完任务后，请调查将所有手动工作自动化的可能性。如果任务和系统是自动化的，请考虑直接从源代码生成文档，以便使两者保持同步。

### 剧本条目

系统中的每个警报条件都应具有一个相应的剧本条目，该条目描述了恢复步骤。在Google，我们发现将本文档链接到发送给值班工程师的任何警报消息中非常有用。在[我们的第一本书的第11章](#)中详细讨论了剧本条目。

### 映射您的开发生命周期

如图13-3所示，管道的开发生命周期(或对管道的更改)与其他系统的开发生命周期没有太大差异。本节遵循管道开发各个阶段的典型发布流程生命周期。



图13-3. 带有发布工作流程的管道开发生命周期

#### 原型

开发的第一阶段涉及对管道进行原型设计并验证语义。原型设计可确保您表达执行管道所需的业务逻辑。您可能会发现一种编程语言可以使您更好地表达业务逻辑，或者一种特定的语言可以更轻松地与现有库集成。特定的编程模型可能适合您的特定用例(例如，数据流与MapReduce，批处理与流传输)。有关完整的编程模型比较的示例，请参见我们的博客文章["Dataflow/Beam & Spark: 编程模型比较"](#)。如果要将功能添加到现有管道中，建议在原型阶段添加代码并运行单元测试。

#### 以1%的演习进行测试

完成原型后，使用生产数据在整个堆栈上运行一个小的设置会很有帮助。例如，使用实验集运行管道，或者在非生产环境中使用1%的生产数据空运行。逐步扩大规模，跟踪您的管道性能，以确保您不会遇到任何瓶颈。产品发布给客户后，请运行性能测试。这些测试是不可或缺的开发步骤，有助于防止因推出新功能而造成停机。

#### 演练

在部署到生产之前，在预生产(或演练)环境中运行系统很有用。演练环境中的数据应尽可能接近实际生产数据。我们建议保留生产数据的完整副本或至少有代表性的子集。单元测试不能解决所有管道问题，因此让数据端到端流经系统以解决集成问题非常重要。

集成测试还可以识别错误。使用单元测试和集成测试，对新生成的数据与先前生成的已知正常数据进行A/B比较。例如，在对发行版进行认证并将其标记为可以投入生产之前，请检查您的先前发行版是否存在预期或意外的差异。

#### 金丝雀

与无状态作业相比，管道测试和验证需要更多的时间和精力-数据得以保留，并且转换通常很复杂。如果生产版本中出现问题，请务必尽早发现问题，这样可以限制影响。修改您的管道会有所帮助！金丝雀是一个过程，通过该过程可以部分部署服务(在本例中为管道应用程序)并监控结果。有关金丝雀的详细讨论，请参见第16章。金丝雀绑定到整个管道，而不是单个进程。在金丝雀阶段，您可以选择处理与实时管道相同的实际生产数据，但是跳过对生产存储的写入。两阶段突变等技术可以提供帮助(请参见第279页的“幂等和两阶段突变”一节)。通常，您必须等待整个处理过程完成才能发现任何影响客户的问题。试运行(或两阶段突变)后，将金丝雀管道的结果与实时管道的结果进行比较，以确认其运行状况并检查数据差异。

有时可以通过逐步更新作业的任务或先在一个区域中然后在另一个区域中进行更新来在金丝雀中前进，但这并不总是通过管道实现。使用复制数据的管道(例如 Dataproc<sup>76</sup> 和 Dataflow,<sup>77</sup>)支持区域终结点<sup>78</sup>，并防止这种金丝雀般的进展-您无法

隔离地重新加载一个单元。如果您运行多宿主管道，则可能无法像执行服务工作一样将其部署到单个区域(或一定百分比的服务器)。而是先对一小部分数据执行转储，或者如前所述，先在空运行模式下转储。

在验证金丝雀或生产前环境期间，重要的是评估管道的运行状况。通常，您可以使用与跟踪SLO相同的指标。验证金丝雀是一项很适合自动化的任务。

### 执行部分部署

除了取消更改之外，您可能还希望执行部分部署，尤其是在存在主要功能启动或更改会影响系统性能和资源使用的情况下。如果不先对实际流量的子集测试更改，就很难预测这些发布的影响。您可以在接受允许的数据子集的管道中将部分部署实现为标志或配置选项。考虑首先处理一个或两个帐户的新功能，然后逐渐增加数据量(例如，样本数据的~1%，~10%，~50%，最后是100%)。

部分部署可能会以多种方式出错:输入数据可能不正确或被延迟，数据处理可能存在错误，或者最终存储可能存在错误。这些问题中的任何一个都可能导致中断。避免将一组损坏的数据推广到您的低延迟前端。力争在用户遇到问题之前尽早发现这些问题。

### 部署到生产

在将新的管道二进制文件和/或配置完全推广到生产环境之后，您应该有足够的信心确信您已经审查了任何潜在的问题(如果确实发生了问题，则监控将警报您)。如果您的部署出错，则能够从已知的良好状态快速还原(例如，回滚二进制文件)，并将任何可能损坏的数据标记为不良数据(例如，将不良数据替换为先前备份版本中的数据，请确保没有作业读取受影响的数据，和/或在必要时重新处理数据)。

### 减少热点和工作负载模式

当资源由于过多的访问而变得超载而导致操作失败时，就会发生热点。管道容易受到工作负载模式的影响(包括通过读取和写入)，这会导致隔离的数据区域出现延迟。热点的一些常见示例包括:

- 由于多个管道工人正在访问一个服务任务而引发错误，从而导致过载。
- 由于并发访问仅在一台计算机上可用的一条数据，导致CPU耗尽。通常，数据存储的内部组件具有最低级别的粒度，如果进行大量访问，粒度可能会变得不可用(例如，Spanner数位板可能由于有问题的数据段而变得超载)即使大多数数据存储都可以)。
- 由于数据库中行级锁争用而导致的延迟。
- 由于同时访问硬盘驱动器而导致的延迟，这超出了驱动器磁头足够快地移动以快速定位数据的物理能力。在这种情况下，请考虑使用固态驱动器。
- 需要大量资源的大型工作单位。

热点可以隔离到一部分数据。为了解决热点问题，您还可以阻止诸如单个记录之类的细粒度数据。如果该数据被阻止，则其他管道可以继续进行。通常，您的基础架构可以提供此功能。如果大量的处理工作消耗了不成比例的资源，则管道框架可以通过将工作分解为较小的部分来动态地重新平衡。为了安全起见，最好还是将紧急关机内置到客户端逻辑中，以允许您停止处理并隔离以大量资源使用或错误为特征的细粒度处理工作。例如，您应该能够快速设置一个标志或推送一个配置，以使您可以跳过与特定模式或有问题的用户匹配的输入数据。其他减少热点的策略包括:

- 重组数据或访问模式以平均分散负载
- 减少负载(例如，静态分配部分或全部数据)
- 减少锁粒度以避免数据锁争用

### 实施自动缩放和资源规划

工作负载峰值很常见，如果您不做好准备，可能会导致服务中断。自动缩放可以帮助您处理这些峰值。通过使用自动缩放，您不必在100%的时间内准备峰值负载(有关自动缩放的更多详细信息，请参见第11章)。不断运行峰值容量所需的worker数量是昂贵且效率低下的资源使用。自动缩放可减少空闲的工作节点，因此您无需为不需要的资源付费。该策略对于可变的流管道和工作负载特别重要。批处理管道可以同时运行，并且将消耗尽可能多的资源。

预测系统的未来增长并相应地分配容量可确保您的服务不会耗尽资源。权衡资源成本和使管道更高效所需的工程量也很重要。在进行资源规划并估算未来的增长时，请记住，成本可能不会仅仅与运行管道作业隔离开来。您还可能要支付数据存储和网络带宽成本，以便跨区域或跨区域的读写操作复制数据。另外，某些数据存储系统比其他系统昂贵。即使单位存储成本很低，这些成本也可以迅速增加，以用于非常大的数据集或使用存储服务器上大量计算资源的昂贵数据访问模式。优良作法是通过定期检查数据集并修剪未使用的内容来帮助降低成本。

尽管应根据其端到端SLO衡量一系列管道阶段的有效性，但应在每个单独阶段对管道效率和资源使用情况进行度量。例如，假设您有许多使用BigQuery的工作，并且注意到发布后BigQuery资源使用量显着增加。如果您可以快速确定哪些工作是不可靠的，则可以将工程重点放在这些工作上以降低成本。

### 遵守访问控制和安全策略

数据流经您的系统，并且通常会一路被持久化。在管理任何持久数据时，我们建议您遵循以下隐私，安全性和数据完整性原则：

- 避免将个人身份信息(PII)存储在临时存储区中。如果需要临时存储PII，请确保数据已正确加密。
- 限制对数据的访问。仅授予每个管道阶段所需的最小访问权限，以读取上一个阶段的输出数据。
- 对日志和PII设置生存时间(TTL)限制。

考虑一个与GCP项目绑定的BigQuery实例，该实例的访问权限可以使用Google Cloud Identity和访问管理来管理(例如，前面描述的Dressy示例)。为每个函数创建不同的项目和实例，可以进行更精细的作用域限制访问。表可以具有主项目，并可以在客户端项目之间交叉创建视图，以允许它们进行受控访问。例如，Dressy限制了对包含来自特定项目角色的工作的敏感客户信息的表的访问。[计划升级路径](#)

重要的是，将管道设计为具有弹性的，这样系统故障(例如计算机或区域中断)永远不会触发SLO违规页面。到您被呼叫时，由于所有自动化措施都已用尽，因此您需要手动进行干预。如果您拥有定义明确的SLO，可靠的指标和警报检测，则在客户注意到或报告问题之前会收到警报。违反SLO时，重要的是要迅速做出反应并向客户发送主动通信。

### 管道需求和设计

当今的市场提供了许多管道技术和框架选项，要确定哪种最适合您的用例可能会很困难。一些平台提供了完全托管的管道。其他给您更大的灵活性，但需要更多的动手管理。在SRE中，我们通常在设计阶段花费大量时间来评估哪种技术最合适。我们根据用户需求，产品要求和系统约束来比较和对比各种设计选项。本节讨论可用于评估管道技术选项和对现有管道进行改进的工具。

### **您需要什么功能？**

表13-1提供了我们建议您在管理数据处理管道时优化的功能的列表。其中一些功能可能已存在于您现有的管道技术中(例如，通过托管管道平台，客户端应用程序逻辑或操作工具)。您的应用程序可能不需要其中一些功能-例如，如果您的工作单位是幂等的并且可以为相同的结果执行一次以上，则您不需要"精确地一次"语义。

*表13-1.推荐的数据管道功能*

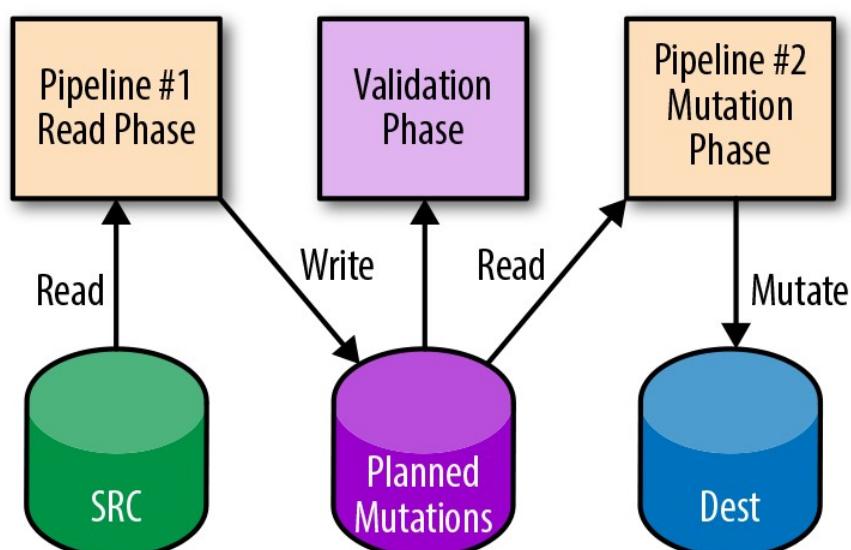
管道项目	功能
延迟	使用支持流传输，批处理或同时支持两者的API。在支持低延迟应用程序方面，流处理通常比批处理更好。如果选择批处理，但有时可能需要流传输，那么可互换的API可能会在以后降低迁移成本。
数据正确性	<p>全局一次语义。您可以要求(最多)处理一次数据以获得正确的结果。</p> <p>两阶段突变。</p> <p>窗口功能用于事件处理和聚合。您可能希望使用固定的时间，会话或滑动窗口来划分数据(因为并非总是按接收顺序处理数据)</p> <p>您可能还需要有序保证。</p> <p>黑盒监控。</p> <p>控制多个作业或管道阶段的流程的能力。此控件应允许您门控一个作业，直到另一个作业完成为止，这样该作业就不会处理不完整的数据。</p>
高可用性	多宿主。自动缩放。
解决数据处理中的事件的平均时间(MTTR)	<p>将您的代码更改绑定到发行版，从而可以快速回滚。</p> <p>已测试数据备份和还原程序到位。</p> <p>如果发生中断，请确保您可以轻松耗尽某个区域的服务或处理能力。</p> <p>具有有用的警报消息，仪表板和日志以进行调试。特别是，您的监控应快速确定管道延迟和/或数据损坏的原因。</p> <p>使用数据检查点来帮助在管道中断时更快地恢复。</p>
平均检测时间(MTTD)中断	确保已安装SLO监控。监控SLO外警报使您能够检测到影响客户的问题。在 <a href="#">症状(相对于原因)</a> 上发出警报可以减少监控差距。
生命周期以防止错误到达生产阶段	发展历程我们建议在部署到生产环境之前，在金丝雀环境中运行所有更改。此策略降低了更改影响生产中的SLO的可能性。
检查并预测资源使用或成本	<p>创建(或使用现有的)资源账单仪表板。确保包括存储和网络等资源。</p> <p>创建一个指标，使您可以关联或预测增长。</p>

管道项目	功能
易于发展	<p>支持最适合您的用例的语言。通常，管道技术会将您的选择限制为一种或两种语言。</p> <p>使用简单的API定义数据转换并表达管道逻辑。考虑一下简单性和灵活性之间的权衡。</p> <p>重用基础库，指标和报告。创建新管道时，可重用资源使您可以将开发重点放在任何新业务逻辑上。</p>
易于操作	<p>尽可能使用现有的自动化和操作工具。这样做可以降低运营成本，因为您不需要维护自己的工具。</p> <p>自动化尽可能多的操作任务。</p> <p>很少执行的较大任务，例如包括一系列依赖关系和先决条件，这些依赖关系和先决条件可能太多或太复杂，以至于人类无法及时进行评估(例如，将数据和管道堆栈从区域A移至区域B，然后调低区域A)。为了简化这样的过渡，请考虑投资自动化。在投入生产之前，可能会对区域B中的管道堆栈进行一些管道运行状况检查。</p>

### 幂等和两阶段突变

管道可以处理大量数据。当管道发生故障时，必须重新处理一些数据。您可以用"幂等突变"设计模式来防止存储重复或不正确的数据。[幂等突变](#)是一种可以多次应用且具有相同结果的突变。实施此设计模式允许使用相同的输入数据单独执行管道，以始终产生相同的结果。

在测试或修复管道时，您需要根据预期的输出来知道所应用的突变是否对管道所有者是可接受的。"两阶段突变"设计模式可以为您提供帮助。通常，从源读取数据并进行转换，然后应用突变。对于两阶段突变，突变本身存储在一个临时位置。可以针对这些潜在突变运行单独的验证步骤(或管道)，以验证它们的正确性。后续流程步骤仅在突变通过验证后才应用已验证的突变。图13-4显示了两阶段突变的示例。



通常，管道是长时间运行的流程，可分析或变异大量数据。无需特殊考虑，提前终止的管道将丢失其状态，从而需要再次执行整个管道。对于创建AI模型的管道来说尤其如此，因为模型计算的每次迭代都依赖于先前的计算。*Checkpointing*是一项技术，它使诸如管道之类的长期运行的进程能够定期将部分状态保存到存储中，以便以后可以恢复该进程。

尽管检查点通常用于故障情况，但在需要抢占或重新安排作业(例如，更改CPU或RAM限制)时，它也很有用。可以完全关闭该作业，并且在重新计划之后，它可以检测到已经处理了哪些工作单元。*Checkpointing*的另一个优点是使管道可以跳过潜在的昂贵读取或计算，因为它已经知道工作已经完成。

## 代码模式

一些常见的代码模式可以使您的管道更有效地管理并减少进行更改或更新所需的工作。

### 重用代码

如果您操作多个相似的管道并希望实施新的监控功能或度量标准，则必须对每个单独的系统进行检测。如果使用正确的策略，则这种通用的工作流程并不困难。实施可重用的代码库使您可以在一个位置添加用于监控的度量，并在多个管道或阶段之间共享它。共享库使您能够：

- 以标准方式获得对所有数据管道的洞察力。
- 为每个管道重用其他数据分析系统(例如，适用于所有管道的流量报告)。
- 针对多个作业在同一度量上发出警报，例如通用数据更新警报。

### 使用微服务方法创建管道

使用微服务时，让服务执行一项任务并做好它很重要。与使用许多核心服务相比，使用一组相同的核心库(仅在业务逻辑上有所不同)的微服务的操作要容易得多。类似的模式可以应用于管道。与其创建一个整体式管道应用程序，不如创建可以分别发布和监控的较小的管道。这样，您将获得与从微服务架构中获得的相同的收益。

### 管道生产准备就绪

正如第18章所讨论的那样，PRR(生产准备情况审查)是Google SRE团队用于推出新服务的过程。本着同样的精神，在咨询管道技术的选择或设计时，我们使用了“管道成熟度矩阵”。

### 管道成熟度矩阵

表13-2中的矩阵衡量了五个关键特征(但是您可以扩展矩阵以测量要优化或标准化的其他特征)：

- 容错能力
- 可扩展性
- 监控和调试
- 透明和易于实施
- 单元和集成测试

成熟度矩阵代表了Google许多管道专家的集体知识。这些人负责跨多个Google产品运行管道并生产关联的系统。

每个特征的测量范围为1到5，其中1代表“混乱”(计划外，临时，有风险，完全手动)，而5代表“持续改进”。要对系统评分，请阅读以下每个特性的说明，然后选择最匹配的里程碑。如果应用了多个里程碑，则使用中间的分数(即2或4)。完整的评分表将为您清楚地说明您的系统需要改进的地方。

我们建议您花时间对矩阵确定的任何薄弱环节进行改进。矩阵建议的仪器监控，警报和其他工具可能会很耗时。进行改进时，您可以从寻找适合您需求的现有产品或开源工具开始，而不是创建自己的产品。在Google，我们鼓励团队使用现有的管道技术或工具来提供现成的管道支持和功能。可以重复使用的工具和流程越多越好。

表13-2. 管道成熟度矩阵，包括开始，中期和高级成熟度的示例里程碑

	1.混乱	3。功能	5。持续的提升
<b>容错能力</b>			
故障转移	不支持故障转移	重试工作单元的一些支持(即使是手动)	具有自动故障转移功能的多宿主
全局工作调度	不支持全局工作调度，多宿主或故障转移	支持hot/hot/hot处理 (在所有三个区域中处理相同的工作，因此，如果任何区域都不可用，则至少有一个仍在运行)	支持有效的warm/warm/warm处理 (在所有三个区域中分配工作并集中存储工作以应对任何区域损失)
任务失败管理	不支持失败的工作单元	---	自动重试失败的工作单元 自动隔离不良工作单位
<b>可伸缩性</b>			
自动扩展可用工作池	无自动缩放；需要人工干预	自动缩放功能可通过使用其他手动工具来实现。	内置的自动化自动缩放支持，无需在第三方工具中进行配置
自动动态重新分片以在整个池中实现均衡负载	工作单元是固定的，无法进行更改	支持工作的手动重新分片，或者可以使用其他代码自动完成重新分片	对动态子分片的内置支持，以平衡可用工作节点池中的工作
减载/任务优先级	不存在工作单元优先级	工作单元优先排序的某些功能	存在一个易于使用的工作单元优先级排序功能 内置的减载支持工作节点了解抢占通知，之后工人将进行清理(完成工作)/减轻压力)
<b>监控和调试</b>			
调试工具和能力	无日志；没有办法功能识别或工作单元跟踪失败	有一种解决方案可以识别出故障的工作单元并提取相关日志	有一种解决方案允许用户在工作单元发生故障的时间访问日志。直接从失败的工作单元中检索此数据 有一种解决方案可以自动隔离并重放发生故障的工作单元。

	1.混乱	3。功能	5。持续的提升
仪表板和可视化	无仪表板或可视化方案支持管道信息展示	存在易于配置的可视化解决方案，支持显示以下信息： • 工作单元数 • 每个阶段的延迟和老化信息	管道整个执行图的细粒度可视化视图 延迟到每个阶段的可视化视图 节流和回退的可视化以及基本原理 由于资源使用而导致的限制因素的信息
易于实施且透明			
可发现性	没有功能支持可发现性(正在运行的管道列表和他们的状态)	可发现性的一些支持;可发现性可能需要手动设置，否则并非所有管道都可被发现	内部支持自动发现；允许配置管道列表的全局数据注册服务
代码	使用该技术的大量安装成本	一些可用的可重用组件	可用的基础框架和最小化代码 管道可配置为机器可读的格式 零配置 与相关团队大量使用的其他管道解决方案库文件相似
文档和最佳实践	文档和稀疏或过时的最佳实践文档	每个组件的最小化安装文档。	全面且及时更新的文档和为用户准备的培训示例
单元和集成测试			
单元测试框架	没有支持或单元测试框架	测试需要很长时间才能运行，并且经常会超时。 太多资源需求 没有代码覆盖支持轻松切换数据源以测试资源	与sanitizers(ASAN，TSAN等)一起运行。 尽可能小构建依赖关系图 代码覆盖范围支持 提供调试信息和结果 没有外部依赖 内置测试数据生成库

	1.混乱	3。功能	5。持续的提升
易于配置 (这与管道的可伸缩性方面直接相关)	除了了解管道本身以外，不支持测试配置或需要大量时间来学习测试-例如，测试使用的编程语言和API与管道	第一次集成测试需要大量的设置，但是后续测试要么是复制/粘贴，要么是对第一个扩展的扩展，具有最小的覆盖-例如，可以最小化调整测试数据生成器以支持为许多不同的管道应用程序收集测试数据	与生产配置脱钩，同时不阻止重用；更容易定义集成测试配置并重用产品配置的相关部分
支持集成测试框架 (这描述了管道应如何交互并支持各种集成测试方法，这些方法不一定是在管道本身的一部分)	使用Cloud或第三方开源工具进行差异，监控，集成测试等可能难以实施和/或占用大量资源；需要内部工具或工具不存在	与管道结合使用的最少的文档以及集成测试工具和方法的示例，即使对于输入最少的数据集，也需要大量时间 难以触发测试方案的按需执行很难将生产与非生产问题分开(例如，所有事件日志都转到生产日志服务中)	内置支持按比例缩小的输入数据 支持差异化输出数据(例如，保留输出测试数据) 可配置的监控以进行测试运行验证 为管道构建的大量文档和集成测试示例

## 管道故障：预防和应对

管道可能由于多种原因而失败，但是最常见的罪魁祸首是数据延迟和数据损坏。发生中断时，迅速发现并修复问题将大大减少其影响。在Google，如果发生中断或违反SLO的情况，我们会跟踪MTTD和MTTR的指标。跟踪这些指标表明我们在检测和修复问题方面的效率。在Google发生停电后的事后分析中，我们分析了停电的原因，以找出任何模式并解决操作上的麻烦。

本节介绍一些常见的故障模式，有效响应管道故障的方法，以及有助于您防止将来发生管道故障的策略。

### 潜在故障模式

#### 数据延迟

如果管道的输入或输出被延迟，则管道可能会失败。如果没有适当的预防措施，即使下游作业没有必要的数据，它也可能开始运行。陈旧的数据几乎总是比错误的数据要好。如果您的管道处理不完整或损坏的数据，则错误将传播到下游。恢复或重

新处理不良数据需要花费时间，并且可能会延长中断时间。相反，如果您的管道停滞不前，等待数据，然后在数据可用后恢复运行，则数据将保持高质量。创建所有阶段都重视的数据依赖关系很重要。

根据管道的类型，延迟数据的影响范围可能从过时的应用程序数据到停滞的管道。在批处理管道中，每个阶段在开始之前都等待其前任完成。流系统更加灵活:使用事件时间处理(例如Dataflow)，下游阶段可以在相应的上游部分完成后立即开始工作的一部分，而不是等待所有部分都完成。

发生此类中断时，您可能需要通知任何相关服务。如果停机是用户可见的，则可能还必须通知客户。在调试管道中断时，查看当前和过去管道运行的进度以及直接链接到日志文件和数据流图非常有帮助。能够在分析系统的计数器和统计信息时跟踪整个系统的工作单元也很有用。

### **数据损坏**

如果未检测到，损坏的管道数据(输入 和/或 输出)可能会导致面向用户的问题。您可以通过进行适当的测试来识别损坏的数据，并使用提醒您可能发生损坏的逻辑来规避许多面对用户的问题。例如，管道系统可以实施阻止策略和检测滥用/垃圾邮件，以自动或手动过滤掉不良数据源。

损坏的数据可能有多种原因:软件错误，数据不兼容，不可用的区域，配置错误等。  
修复损坏的数据涉及两个主要步骤:

1. 通过防止进一步的损坏数据进入系统来减轻影响。
2. 从以前的正确版本还原数据，或重新处理以修复数据。

如果单个区域正在提供损坏的数据，则可能需要从该区域中引流您的服务工作和/或数据处理。如果软件或配置错误有问题，则可能需要快速回滚相关的二进制文件。通常，数据损坏会导致数据窗口不正确，一旦解决了基本问题(例如，修复了管道二进制文件中的软件错误)，就需要重新处理这些数据。为了减少重新处理的成本，请考虑选择性地进行重新处理-读入并仅处理受数据损坏影响的用户或帐户信息。或者，您可以保留一些中间数据，这些数据可以用作检查点，以避免从头到尾重新处理管道。

如果管道的输出损坏，则下游作业可能会传播损坏的数据，或者正在服务的作业可能会提供不正确的数据。即使进行了最佳测试，开发

### **管道故障:预防和应对**

修改和发布实践，软件或配置错误可能会导致数据损坏。我们建议您为这种情况做准备，并能够快速重新处理和还原数据。从这种数据损坏中恢复是劳动密集型的并且难以自动化。

## **潜在原因**

### **管道依赖项**

当您尝试确定中断的原因时，调查管道依赖项(例如存储，网络系统或其他服务)很有用。这些依赖关系可能会限制您的请求/流量，或者，如果资源不足，它们会拒绝任何新数据。输入/输出的速率可能会由于多种原因而变慢:

- 输出接收器或存储设备可能拒绝写入数据。
- 可能存在无法完成的特定热点数据范围。

- 可能存在存储错误。

一些管道依赖问题无法解决。提交故障单或错误，并留出足够的时间来添加更多资源或解决流量模式，这一点很重要。实施负载均衡并删除低优先级数据可能有助于减轻影响。

### 管道应用或配置

管道故障可能是瓶颈，管道作业中的错误或配置本身中的错误(例如，CPU密集型处理，性能回归，内存不足故障，易于出现热点的滥用数据，或指向错误的输入/输出位置的配置)。根据原因，有几种可能的解决方案：

- 回退二进制文件/配置，选择修补程序，或修复任何权限问题。
- 考虑重组导致问题的数据。

应用程序或配置错误可能会导致数据不正确或导致数据延迟。这些类型的错误是最常见的中断原因。我们建议您花一些时间进行管道开发，并确保新的二进制文件和配置在非生产环境中正常运行，然后再进行全面部署。

### 意外的资源增长

系统负载的突然变化和计划外变化可能会导致管道发生故障。您可能需要其他计划外资源，以保持服务运行。自动缩放

您的应用程序作业可以帮助满足新负载的需求，但是您还应该意识到，增加的管道负载也会给下游依赖性带来压力-您可能还需要计划更多的存储和/或网络资源。

良好的资源规划和准确的增长预测可以在这些情况下提供帮助，但是这种预测可能并不总是正确的。我们建议您熟悉请求其他紧急资源的过程。根据管道部署的性质和所需的资源数量，获取这些资源所需的时间可能很长。因此，我们建议准备临时解决方案以保持您的服务正常运行-例如，通过管道优先处理不同类别的数据。

### 区域级中断

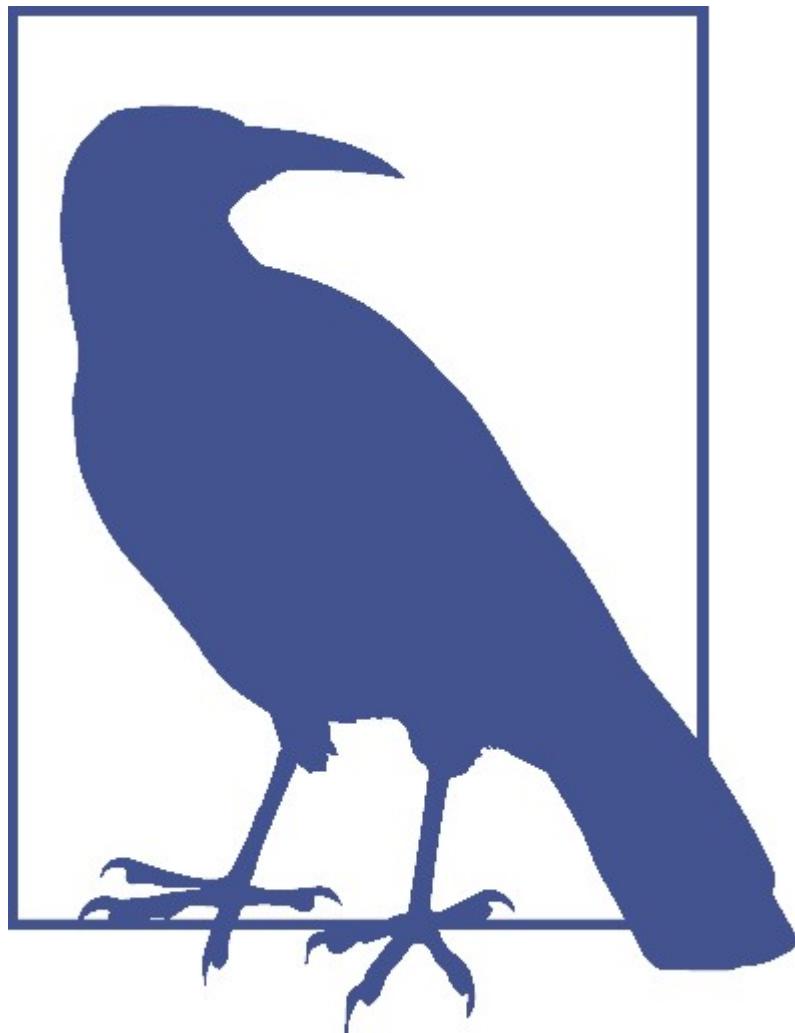
区域中断对所有管道都是不利的，但单归管道尤其容易受到影响。如果您的管道在突然变得不可用的单个区域中运行，则管道将停止直到该区域恢复正常。如果您拥有具有自动故障转移功能的多宿主管道，则响应可能很简单，例如引流处理或从受影响的区域进行服务，直到中断结束。当区域关闭时，数据可能会搁浅或延迟，从而导致管道输出不正确。结果，可能损害从任何相关作业或服务输出的数据的正确性。

## 案例分析:Spotify

由伊戈尔·马拉维奇(Igor Maravić)撰写

Spotify是全球领先的音乐流媒体公司。每天，成千上万人使用Spotify收听自己喜欢的歌曲，与朋友分享音乐并发掘新的艺术家。

本案例研究描述了我们的事件传递系统，该系统负责可靠地收集从Spotify应用程序生成的检测数据。该系统产生的数据有助于我们更好地了解最终用户，并在适当的时间为他们提供适当的音乐。



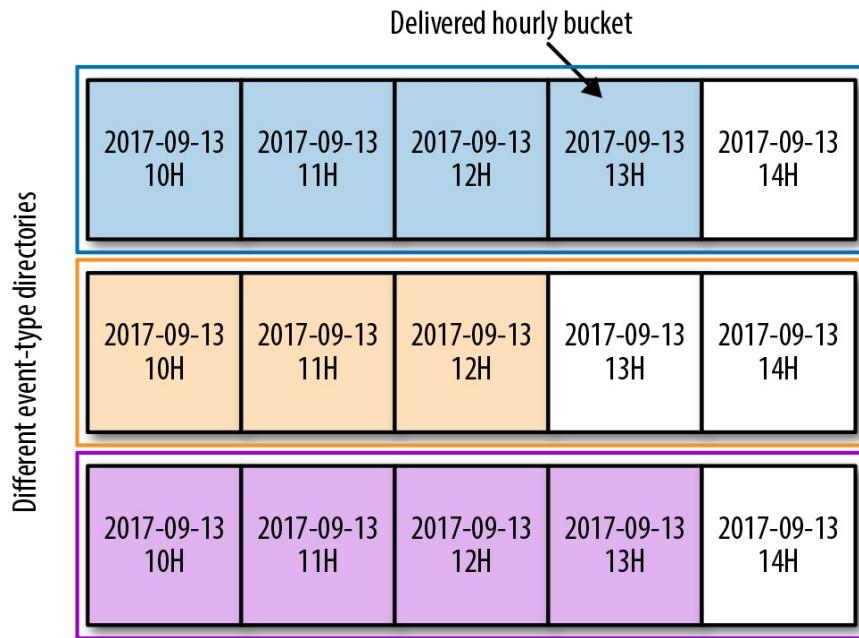
在此案例研究中，“客户”是指Spotify中使用事件传递系统中的数据的开发团队。“最终用户”是指使用Spotify服务收听音乐的个人。

### 事件交付

我们将最终用户交互称为“事件”。每当用户收听歌曲，点击广告或跟随播放列表时，我们都会记录一次事件。Spotify每天捕获并向我们的服务器发布数千亿个事件（多种类型）。这些事件在Spotify中有许多用途，从A/B测试分析到显示播放计数再到增强个性化发现播放列表。最重要的是，我们根据交付的事件向艺术家支付版税。<sup>79</sup>我们必须拥有可靠的事件存储和交付方式。

在我们处理事件数据之前，需要收集该数据并将其传递到持久性存储中。我们使用事件传递系统来可靠地收集和保留所有已发布事件。事件传递系统是我们数据基础结构的核心支柱之一，因为我们几乎所有的数据处理（直接或间接）都依赖于它传递的数据。

所有传递的事件均按类型和发布时间进行划分。如图13-5所示，在任何给定小时内发布的事件都被分组在一起并存储在指定的目录中，该目录称为“已交付事件的每小时存储桶”。然后将这些存储桶分为事件类型目录。这种分区方案简化了Spotify的数据访问控制，所有权，保留和使用。



每小时存储桶是我们的数据作业与事件传递系统唯一的接口。因此，我们会根据每个事件类型的小时时段交付情况来衡量性能并为事件交付系统定义SLO。

### 事件交付系统的设计和架构

我们的每小时存储区位于Google Cloud Storage(GCS)上。在设计过程的早期，我们决定将数据收集与系统内的数据传递分离。为此，我们使用了一个全局分布的持久队列Google Cloud Pub/Sub作为中间层。分离后，数据收集和传递将充当独立的故障域，从而限制了任何生产问题的影响，并导致了更具弹性的系统。图13-6描述了我们的事件传递系统的体系结构。

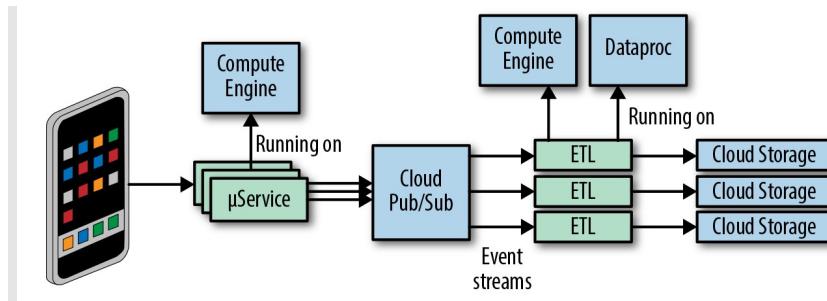


图13-6. 事件交付系统架构

### 数据采集

产生的事件<sup>80</sup>按事件类型分组。每种事件类型都描述了Spotify应用程序中的用户操作。例如，一种事件类型可以指代订阅播放列表的用户，而另一种事件类型可以指代开始播放歌曲的用户。为了确保不同的事件类型不会互相影响，系统具有完全的事件类型隔离。来自不同事件类型的各个事件在Google Cloud Pub/Sub中发布到其已分配的主题。发布是由我们的微服务执行的，这些微服务在Spotify数据中心和Google Compute Engine(GCE)上运行。为了进行交付，每个已发布的事件流都由ETL流程的专用实例处理。

### 抽取变换加载(ETL)

ETL流程负责将已发布的事件传递到GCS上正确的每小时时段。ETL过程包含三个步骤/组成部分:

1. 专用微服务使用事件流中的事件。
2. 另一个微服务将事件分配给它们的每小时分区。
3. 在Dataproc上运行的批处理数据作业会从其每小时的分区中删除事件的重复数据，并将其保留到GCS上的最终位置。

每个ETL组件都有一个责任，这使这些组件更易于开发，测试和操作。

### 数据传送

事件类型传递是由我们的客户(Spotify的其他工程团队)直接动态启用或禁用的。通过简单的配置即可控制投放。在配置中，客户定义应传递的事件类型。随着打开或关闭每种事件类型的传递，微服务会动态获取并释放运行ETL的Google GCE资源。以下代码显示了客户可以启用/禁用的示例事件类型:

```
events:  
-CollectionUpdate  
-AddedToCollection  
-RemovedFromCollection
```

当客户启用新事件类型的交付时，我们事先并不知道需要多少资源来保证交付。因此，手动确定必要的资源非常昂贵。为了实现交付不同事件类型的最佳资源利用率，我们使用了GCE [Autoscaler](#)。

### 事件传递系统运维

为我们的事件传递系统定义和传达SLO可以通过三种方式提供帮助:

#### 设计和开发

在开发我们的系统时，拥有明确的SLO可为我们朝着实现的目标迈进。这些目标可帮助我们做出务实的设计选择并优化系统以简化操作。

#### 确定性能问题

一旦将系统部署到生产环境中，SLO将帮助我们确定系统的哪些部分运行不佳以及需要集中精力进行哪些工作。

#### 设定客户期望

SLO使我们能够管理客户的期望并避免不必要的支持请求。当客户清楚我们系统的局限性时，他们有权决定如何设计，构建和操作依赖于我们数据的自己的系统。

我们为客户提供用于事件传递系统的三种SLO类型:及时性，完整性和偏度(接下来讨论)。这些SLO基于GCS提供的每小时数据段。为了尽可能客观地避免与事件无关的事件传递，我们使用独立的外部系统(例如，Datamon，下一节介绍的数据可视化工具)对所有SLO进行了测量。

#### 及时性

我们的及时性SLO定义为每小时传送一桶数据的最大延迟。输送延迟的计算方式为：输送铲斗的时间与铲斗可能被关闭的最早的理论时间之间的时差。图13-7提供了此传递延迟的示例。该图显示了12、13和14小时的时段。如果第13小时的时段在14:53关闭，则可以说关闭延迟为53分钟。

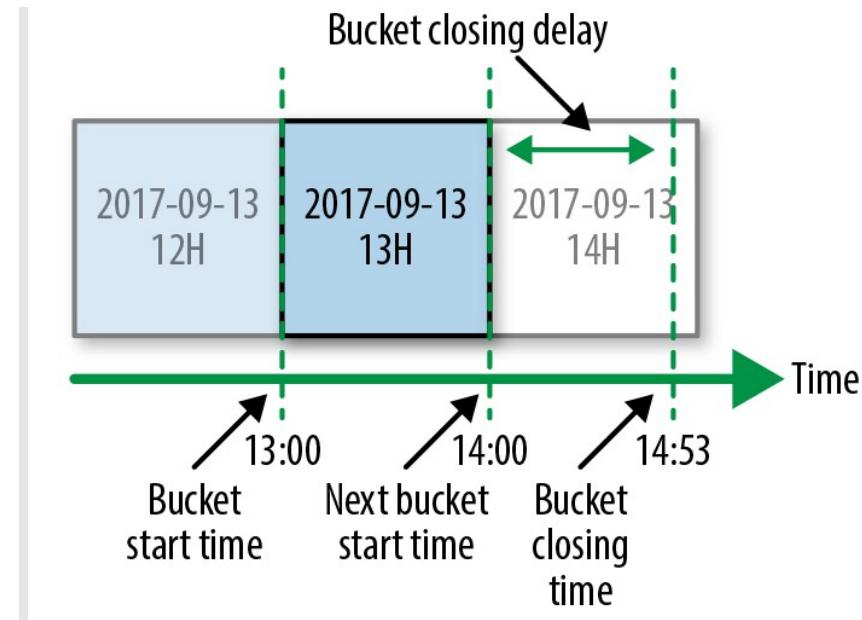


图13-7.事件时间分区

数据传递的及时性是我们用来评估数据管道性能的指标。为了衡量和可视化及时性，我们使用了一个名为Datamon的工具，这是我们的内部数据监控工具，它是围绕小时桶的概念而构建的。图13-8显示了典型的Datamon UI。每个绿色矩形(在灰色图像中，绝大多数矩形)代表一个按小时计时的时段。灰色矩形(在此处群集在右侧)表示尚未交付的存储桶，而红色矩形(在最上面一行的3个黑色矩形)表示未在要求的SLO中交付的存储桶。成功交付所有小时数的天数显示为单个绿色矩形。

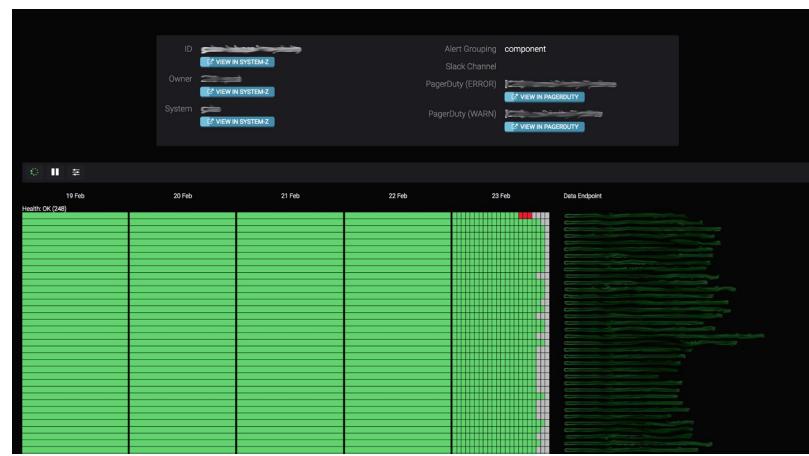


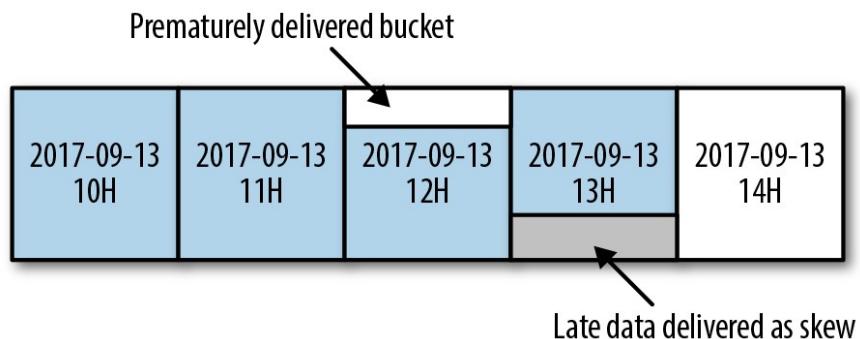
图13-8.用于Spotify数据监控系统的Datamon

下游数据作业要等到它们所依赖的每小时存储桶交付后才能开始处理。每个数据作业在处理数据之前都会定期检查其依赖项的传递状态。交货的任何延迟都会影响下游作业的及时性。我们的客户非常关心及时发送数据。为了帮助我们确定事件期间事件的优先级，我们的事件交付系统的及时性SLO分为三个优先级：高，正常和低。我们的客户为他们的事件类型配置了适当的层。

## 偏度

我们将偏度SLO定义为每天可放错位置的数据的最大百分比。偏斜(和完整性)是事件传递系统特有的概念，其他数据管道中不存在。在设计事件传递系统时，为这些概念定义SLO是一项关键要求，因为它可以处理(在其他事件类型中的)财务事件。对于所有其他事件，尽力而为的交付已经足够了，我们没有公开相应的SLO。事件是否具有财务能力取决于客户配置。

为了确定何时应每小时发送一次，我们的事件传递系统使用启发式方法。根据定义，启发式方法并不总是完全正确的。结果，先前交付的存储桶中的未交付事件可能会被传输到不正确的将来的每小时存储桶中。此放错位置的事件称为“偏斜”。偏斜可能会对作业产生负面影响，因为它们可能会先低估然后在某些时间段内高估价值。图13-9显示了偏斜数据传递的示例。



事件可以通过多种方式在分布式系统中丢失-例如，我们软件的新版本可能包含错误，云服务可能出现故障或开发人员可能意外删除了一些持久事件。为了确保我们收到有关数据丢失的警报，我们会评估完整性。我们将完整性定义为事件成功发布到系统后交付的百分比。

我们每天都会报告偏度和完整性。为了测量这些值，我们使用了内部审核系统，该系统比较所有已发布和已交付事件的计数。任何不匹配都会被报告，我们将采取适当的措施。

为了获得SLO的及时性，偏度和完整性，我们在服务器上收到事件时(而不是在客户端上生成事件时)将事件分配给我们的每小时存储桶。如果我们的用户处于离线模式，则所产生的事件在客户端上最多可以缓冲30天，然后再发布。此外，用户可以在其设备上修改系统时间，这可能导致带有时间戳的事件不正确。由于这些原因，我们使用来自Spotify服务器的时间戳。

我们不提供有关通过事件传递系统传递的事件的数据质量或准确性的任何SLO。我们观察到，在大多数情况下，质量取决于每个事件的内容，而这些事件的内容由客户的业务逻辑组成。为了使我们的系统能够随着客户数量的增长而扩展，我们将其始终专注于交付数据。在这方面，我们使用一个比喻，事件传递的行为应类似于邮政服务:您的邮件应按时，完整无缺地传递。我们有责任向拥有业务逻辑并因此了解数据内容的内部团队提供高质量的SLO。

## 客户集成与支持

许多Spotify团队每天都与事件传递系统进行交互。为了鼓励采用和减少学习曲线，我们采取了以下步骤来简化用户与事件传递系统的交互:

**事件交付作为完全托管的服务**

我们希望避免向客户暴露系统的复杂性，让他们专注于他们要解决的特定问题。我们努力将任何系统复杂性隐藏在定义明确且易于理解的API后面。

#### 有限的功能

为了使我们的API简单，我们仅支持有限的功能集。事件只能以特定的内部格式发布，并且只能以单一序列化格式传递到每小时存储桶。这些简单的API涵盖了我们的大多数用例。

#### 每个事件的传递都需要明确启用

当客户启用事件的交付时，他们定义事件是否具有财务责任及其相关的及时性要求。此外，事件所有权需要明确地定义为启用过程的一部分。我们坚信，让我们的内部团队对他们产生的事件负责，可以提高数据质量。事件的明确所有权也使我们在事件发生时有清晰的沟通渠道。

#### 文档化

无论与系统的交互多么简单，都需要良好的文档来提供良好的客户体验。在诸如Spotify之类的快节奏公司中，低劣而过时的文档编制是一个常见问题。为了解决此问题，我们将文档与其他任何软件产品一样对待：收到我们团队的所有支持请求都将被视为文档问题或实际产品中的问题。大多数支持请求都与系统的公共API有关。我们在编写文档时尝试回答的一些问题示例包括：

- 如何启用事件类型的传递？
- 数据传递到哪里？
- 数据如何分区？
- 我们的SLO是什么？
- 事件期间我们的客户应该期望什么样的支持？

我们的目标是随着客户群的增长而减少收到的支持请求量。

#### 系统监控

监控我们的SLO可提供对系统总体运行状况的高级洞察。我们可靠的全面监控解决方案可确保在出现问题时始终向我们发出警报。使用SLO违规作为监控标准的主要问题是，我们在我们的客户受到影响之后会收到警报。为避免这种情况，我们需要对系统进行足够的操作监控，以在SLO损坏之前解决或缓解问题。

我们从基本的系统指标开始，然后发展到更复杂的指标，分别监控系统的各个组成部分。例如，我们监控CPU使用情况作为实例运行状况的信号。CPU使用率并非始终是最关键的资源，但它可以作为基本信号正常运行。

当我们试图了解和解决生产问题时，有时系统监控还不够。为了补充我们的监控数据，我们还维护应用程序日志。这些日志包含与它们描述的组件的运行和运行状况相关的重要信息。我们非常注意确保仅收集正确数量的日志数据，因为无关的日志很容易淹没有用的日志。例如，错误的日志记录实现可能会记录处理传入请求的高容量组件的所有传入请求。假设大多数请求是相似的，则记录每个请求不会增加太多价值。此外，如果记录了太多请求，将很难找到其他日志条目，磁盘填充速度更快，并且我们的服务的整体性能开始下降。更好的方法是对记录的请求数量进行速率限制，或者仅记录感兴趣的请求（例如导致未处理异常的请求）。

通过阅读应用程序日志来调试生产中的组件具有挑战性，应该是最后的手段。

## 容量规划

事件传递系统的全天候可靠运行需要正确数量的已分配资源，尤其是因为组件已部署到单个GCP项目中，并且它们共享一个公共配额池。我们使用容量规划来确定每个系统组件需要多少资源。

对于我们大多数系统组件，容量规划是基于CPU使用率的。我们规定每个组件在高峰时段的CPU使用率为50%。此规定起到了安全边界的作用，使我们的系统能够处理突发的流量。当Spotify运行自己的数据中心时，我们为每个组件提供了静态资源。这导致在非高峰时段浪费资源，并且无法处理大量的流量突发事件。为了提高资源利用率，我们对某些无状态组件使用了GCE Autoscaler。

在实施Autoscaler的初期，我们有些痛苦。在某些情况下，Autoscaler可能会导致故障。例如，我们使用CPU使用率作为执行自动缩放的指标。自动缩放器本身取决于CPU使用率与每个组件实例执行的工作量之间的密切关系。如果这种关系被破坏了-通过向每个组件实例添加需要大量CPU的守护程序，或者由于组件实例在不做任何工作的情况下大量消耗了CPU的资源-Autoscaler将启动太多实例。

当Autoscaler呈现出不断增加的CPU使用率且与执行的工作量没有任何关系时，它将无限期扩展，直到使用它能找到的所有资源为止。为了防止Autoscaler耗尽我们的所有配额，我们实施了一些解决方法：

- 我们限制了Autoscaler可以使用的最大实例数。
- 我们严格限制了实例上运行的所有守护程序的CPU使用率。
- 一旦我们发现没有任何有用的工作在做，我们就会积极地限制组件的CPU使用率。

即使使用Autoscaler，我们也要进行容量规划。我们需要确保有足够的配额，并且将Autoscaler可以使用的最大实例数设置得足够高，以在高峰期间为流量提供服务，但又要足够低，以限制“失控”自动缩放的影响。

## 开发过程

为了迅速发布新功能和改进，我们在[持续集成](#)和[持续交付](#)之后开发了事件交付系统(如图13-10所示)(CI / CD)进程。根据此过程，将在进行有效，经过验证或审核的系统更改后立即对其进行部署。拥有足够的测试覆盖范围是成功进行每个更改而不会对我们的SLO造成负面影响的前提。

我们按照“[测试金字塔”的哲学](#)编写测试。这意味着对于我们的每个组件，我们都有大量针对组件内部工作的单元测试，此外，还有少量针对组件公共API的集成测试。在测试金字塔的最高级别，我们进行了系统范围的端到端测试。在此端到端测试中，所有组件都被视为黑盒子，因此测试环境中的系统尽可能类似于生产环境中的系统。

初步开发后，每项更改都要经过同行评审。作为检查过程的一部分，所有测试都在共享的CI/CD服务器上执行，并将结果呈现给开发人员。只有在审阅者批准更改并成功通过所有测试之后，才能合并更改。合并更改后，将立即触发部署过程。

事件交付系统是Spotify基础架构中的关键组件。如果停止传送数据，Spotify中的所有数据处理将停止。因此，我们决定对部署进行更保守的评估，并分阶段部署每个变更。在部署可以从一个阶段转移到另一个阶段之前，我们需要人工批准。

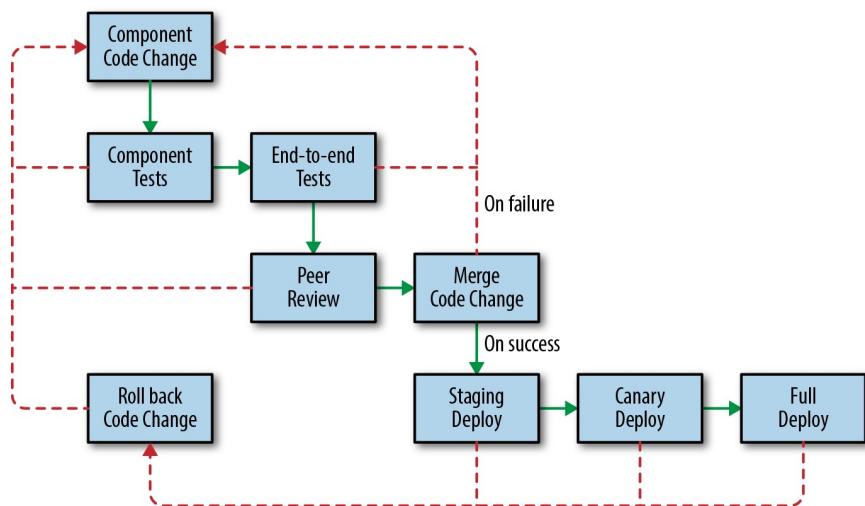


图13-10. 开发过程

在第一个部署阶段，将更改部署到演练环境。这种低风险的演练系统无法处理生产流量。出于测试目的，生产流量的代表部分被镜像到演练系统中，该系统是生产中运行的系统的副本。在第二个部署阶段，将更改部署到生产实例或金丝雀的一小部分。只有在确保一切都顺利进行之后，我们才执行完整的生产部署，无论是演练还是金丝雀(请参阅第16章)。

故障处理

处理事件时，我们的首要任务是减轻损害，并使系统恢复到稳定的先前状态。为了避免使情况变得更糟，我们在事件发生期间请勿对组件进行任何重大更改。该规则的例外是，如果我们得出的结论是事件是由最近部署的新代码引起的。在这种情况下，我们会立即将系统回滚到以前的工作版本。

今天，我们遇到的最常见的操作故障是由系统故障(例如，我们引入软件错误或性能下降)或我们所依赖的外部服务故障(例如，对服务API的更新不能向后兼容或服务打破了其SLO)。我们使用了许多经过考验的Google Cloud和内部Spotify服务，例如Cloud Pub/Sub和[Helios](#)，以加快我们系统的开发速度并减轻我们的运维负担。如果因外部服务而导致故障，我们将有一个专职的值班团队<sup>81</sup>提供支持。使用外部服务的缺点之一是我们无法自己缓解问题。此外，在发生事件期间，将问题传达给第三方需要花费宝贵的时间。不过，我们认为，把偶尔无力的感觉委派给第三方是值得的。

重负载下意外的系统行为是操作失败的另一个常见原因。在精确的生产条件下测试服务是不可能的，因此很难预测所有可能发生的情况。模仿我们的组件在生产中面临的负载也可能很困难。高负载加上无法预料的边缘情况可能会导致有趣的故障情况，例如前面在第295页的“容量规划”中描述的Autoscaler示例。

操作系统故障可能会导致我们的SLO中断。如果我们的数据新鲜的SLO损坏，则不希望客户采取任何措施；客户仅需等待数据到达即可。但是，如果违反了我们的偏斜或完整性SLO，由于数据质量受到影响，我们可能需要让客户参与。当我们检测到完整性或偏斜性问题时，受影响的事件需要正确地进行重新处理：

- 为了处理不完整性，需要从已知良好的最后一个检查点重新发送事件。
  - 为了解决过度偏斜的问题，已交付的事件将重新安排并分配给其正确的每小时存储桶。

重新交付和重新安排事件都是手动完成的。修改已交付事件后，我们强烈建议客户对其进行重新处理以产生足够质量的数据。

### 摘要

多年来，Spotify的事件传递系统已经发展。由于以前的迭代远不够可靠，因此我们的工程师每隔几个晚上就被呼叫一次。我们将大多数开发冲刺花费在事件补救和事后总结上。在设计当前版本时，我们专注于构建一个模块化系统，该系统可以很好地完成一项核心任务：交付事件。此外，我们希望将事件交付作为产品提供给其余的Spotify。为此，我们需要定义并满足SLO，以便我们可以为客户设定明确的期望。

我们采用了各种策略来保持服务的正常运行-从有据可查的值班程序到使用经过验证的外部服务(例如Google Cloud Pub/Sub)。此外，一个团队负责整个系统整个生命周期的开发和运行。

这种开发结构使我们能够利用从维护系统中获得的团队经验来不断改进它。

由于这些努力，我们现在有了一个可靠的系统，使我们可以将时间集中在满足更雄心勃勃的完整性，偏斜度和及时性的SLO上。这导致更好的可用性和更好的整体客户体验。

## 结论

将SRE最佳实践应用于管道可以帮助您做出明智的设计选择并开发自动化工具，从而使管道易于操作，更有效地扩展和更可靠。Spotify的事件传递系统就是一个管道的示例，该管道的建立和运维都牢记核心SRE原则，并使用多种技术(包括内部，Google Cloud和第三方)来满足客户对及时性的需求数据处理。如果没有适当地注意操作最佳实践，则管道可能更容易出现故障，并且需要大量的人工工作，尤其是在增长，迁移，功能启动或停运后清理期间。与任何复杂的系统设计一样，了解您的要求和选择要保留的SLO，评估可用技术并记录设计以及如何执行常见任务非常重要。

<sup>74</sup>. 请参阅第12届国际扩展数据库技术会议的会议记录中的Umeshwar Dayal等人的文章《商业智能的数据集成流程》：数据库技术的进步(纽约: ACM，2000)，1-1-1。 ↵

<sup>75</sup>. 有关将依赖关系纳入服务可靠性的更多详细信息，请参阅Ben Treynor等人的“服务可用性计算”，ACM队列/15，否。2(2017)，<https://queue.acm.org/detail.cfm?id=3096459>。 ↵

<sup>76</sup>. Cloud Dataproc是用于运行Apache Spark和Apache Hadoop集群的完全托管的云服务。 ↵

<sup>77</sup>. Cloud Dataflow是一种完全托管的云服务，用于以流模式和批处理模式转换数据，并且具有相同的可靠性和表现力。 ↵

<sup>78</sup>. 区域端点控制工作人员，并存储和处理Cloud Dataflow作业的元数据。 ↵

<sup>79</sup>. 交付的事件是已交付到持久性存储并以事件交付系统的客户可以使用的方式公开的事件(例如，数据作业)。 ↵

^

80. 产生的事件既是已传递的事件，也是当前流经事件传递系统的事件。 ↵

81. 外部服务提供商保证SLA，并拥有自己的待命团队以确保产品满足这些SLA。 ↵

# 第14章

## 配置设计和最佳实践

由Štěpán Davidovič

与Niall Richard Murphy，Christophe Kalt和Betsy Beyer撰写

在任何地方，配置系统都是常见的SRE任务。这可能是一项令人厌烦且令人沮丧的细致的活动，尤其是如果工程师对他们正在配置的系统不太熟悉，或者在设计配置时没有考虑清楚性和可用性。最常见的情况是，您在以下两种情况之一中执行配置：在有足够时间的初始设置期间，或在需要处理事件的紧急重新配置期间。

本章从设计和维护基础结构系统的人员的角度检查配置。它描述了我们以安全和可持续的方式设计配置的经验和策略。

## 什么是配置？

当我们部署软件系统时，我们并不认为它们是固定不变的。不断变化的业务需求，基础架构要求和其他因素意味着系统不断变化。当我们需要快速更改系统行为，并且更改过程需要昂贵，冗长的重建和重新部署过程时，代码更改就不够用。相反，配置(我们可以将其宽松定义为用于修改系统行为的人机界面)提供了一种低开销的方式来更改系统功能。在部署系统，调整其性能以及事件响应期间，SRE会定期利用此优势。

我们可以认为系统具有三个关键组成部分：

- 该软件
- 系统使用的数据集
- 系统配置

尽管我们可以直观地识别出每个组件，但它们之间通常相距很远。例如，许多系统使用编程语言进行配置，或者至少具有引用编程语言的能力。示例包括Apache和模块，例如[mod\\_lua](#)及其请求挂钩，或窗口管理器XMonad及其[基于Haskell的配置](#)。同样，数据集可能包含代码，例如SQL存储过程，这些代码可能构成复杂的应用程序。

良好的配置界面可实现快速，可靠和可测试的配置更改。如果用户没有直接的方法来更新配置，则更容易出错。用户会经历增加的认知负荷和显着的学习曲线。

### 配置和可靠性

因为我们的系统最终由人管理，所以人负责配置。系统配置的人机界面的质量会影响组织可靠运行该系统的能力。精心设计(或不良设计)的配置界面的影响类似于代码质量随时间对系统可维护性的影响。

但是，配置往往在几个方面与代码有显着差异。通过代码更改系统的功能通常是一个漫长而复杂的过程，涉及小的增量更改，代码检查和测试。相比之下，更改单个配置选项可能会对功能产生重大变化-例如，一个错误的防火墙配置规则可能使您无法使用自己的系统。与代码不同，配置通常生活在未经测试(甚至无法测试)的环境中。

系统配置更改可能需要在很大的压力下进行。在发生事故期间，必须能够简单，安全地调整配置系统。考虑一下早期飞机的界面设计:混乱的控件和指示器导致事故。当时的研究表明，操作员经常发生故障，而与飞行员的技能或经验无关。<sup>82</sup>可用性和可靠性之间的联系转化为计算系统。考虑一下如果我们为.conf文件和监控视图交换控制杆和拨号指示器会发生什么。

### 分离哲学和技术性细节

我们通常在设计新软件或使用现有软件组件组装新系统时讨论配置。我们将如何配置它？配置将如何加载？我们将配置的首要主题分为两部分:配置原理和配置机制。

配置哲学涉及完全独立于所选语言和其他机制的配置方面。我们对哲学的讨论包括如何构造配置，如何实现正确的抽象级别以及如何无缝支持各种用例。

我们对机制的讨论涵盖了诸如语言设计，部署策略以及与其他系统的交互之类的问题。本章将重点放在技术性细节上，部分原因是诸如语言选择之类的话题已经在整个行业中进行了讨论。另外，由于给定的组织可能已经具有强大的外部需求，例如预先存在的配置基础结构，因此配置机制不容易推广。下一章有关Jsonnet的示例提供了现有软件中的配置机制(特别是语言设计)的实际示例。

分别讨论哲学和技术性细节，可以使我们对配置进行更清晰的推理。实际上，配置是否需要大量难以理解的用户输入，诸如配置语言(XML或Lua)之类的实现细节并不重要。相反，即使最简单的配置输入也必须输入到非常繁琐的界面中，也会引起问题。考虑一下(非常)旧的Linux内核配置过程:必须通过命令行终端进行配置更新，该命令行终端需要一系列命令来设置每个参数。为了进行最简单的校正，用户必须从头开始配置过程。<sup>83</sup>

### 配置哲学

本节讨论完全独立于实现的配置方面，因此这些主题适用于所有实现。

按照以下原则，我们理想的配置是完全没有配置。在这个理想的世界中，系统会根据部署，工作量或部署新系统时已经存在的配置片段，自动识别正确的配置。当然，对于许多系统来说，这种理想在实践中是不可能实现的。但是，它强调了理想的配置方向:远离大量的可调参数，而转向简单。

从历史上看，关键任务系统提供大量控制(相当于系统配置)，但也需要大量的操作员培训。考虑图14-1中NASA航天器控制中心中复杂的操作员控制阵列。在现代计算机系统中，这种培训对大多数行业而言不再可行。



图14-1.NASA航天器控制中心的控制面板，说明了可能非常复杂的配置

这种理想的做法减少了我们可以对系统执行的控制量，同时减少了错误的表面积和操作员的认知负担。随着系统复杂性的增加，操作员的认知负担变得越来越重要。

当我们将这些原理应用到Google的实际系统中时，它们通常会导致内部用户支持的便捷，广泛采用和低成本。

#### 配置询问用户问题

无论您要配置什么以及如何配置，人机交互最终都归结为一个界面，该界面询问用户问题，要求输入有关系统应如何运行的信息。无论用户是在编辑XML文件还是使用配置GUI向导，这种概念化模型都适用。

在现代软件系统中，我们可以从两个不同的角度来处理此模型：

##### 以基础设施为中心的视图

提供尽可能多的配置旋钮很有用。这样做使用户可以根据自己的实际需求调整系统。旋钮越多越好，因为可以将系统调整到完美。

##### 以用户为中心的视图

配置询问有关基础结构的问题，用户必须先回答这些问题，然后他们才能恢复实际业务目标。旋钮越少越好，因为回答配置问题比较麻烦。

在我们最小化用户输入的最初理念的推动下，我们赞成以用户为中心的观点。

该软件设计决策的含义超出了配置范围。以用户为中心的配置意味着您的软件需要针对主要用户设计一套特定的用例。这需要用户研究。相反，以基础架构为中心的方法意味着您的软件有效地提供了基础架构，但是将其转变为实际的系统需要用户进行大量配置。这些模型并不存在严格的冲突，但是要调和它们可能非常困难。也许与直觉相反，与极其通用的软件相比，有限的配置选项可以导致更好的接受性-

入门工作量大大降低，因为该软件通常是“开箱即用”的。通过以各种方式删除一些配置旋钮(其中一些将在后续部分中讨论)，随着系统的成熟，从以基础结构为中心的视图开始的系统可能会朝着以用户为中心的焦点移动。

### 问题应该接近用户目标

当我们遵循以用户为中心的配置理念时，我们希望确保用户可以轻松地与我们提出的问题联系起来。我们可以想到频谱上用户输入的性质:一方面，用户用自己的方式描述他们的需求(更少的配置选项)；另一方面，用户准确描述了系统应如何实现其需求(更多配置选项)。

让我们以制作茶为例来配置系统。使用较少的配置选项，用户可以要求“热的绿茶”并大致获得他们想要的东西。在频谱的另一端，用户可以指定整个过程:水量，沸腾温度，茶的品牌和风味，浸泡时间，茶杯类型和杯中的茶量。使用更多配置选项可能更接近完美，但坚持此类细节所需的努力可能要比接近完美饮料的边际收益花费更多。

这种类比对在配置系统上工作的用户和开发人员都有帮助。当用户指定确切步骤时，系统需要遵循这些步骤。但是，当用户改为描述其高级目标时，系统可以随着时间的推移而发展，并改善其实现这些目标的方式。在这里，对系统的用户目标有一个很好的预先了解是必要的第一步。

有关该频谱如何发挥作用的实际说明，请考虑作业调度。想象一下，您有一个一次性的分析过程要运行。像[Kubernetes](#)或[Mesos](#)使您可以实现运行分析的“实际”目标，而不会像确定决定在哪个物理机上运行这样的细微细节给您带来负担。

### 必填和可选问题

给定的配置设置可能包含两种类型的问题:必填问题和可选问题。必须回答问题才能使配置完全提供任何功能。一个例子可能是谁来为一项操作收费。可选问题并不决定核心功能，但是回答它们可以提高功能的质量，例如，设置多个工作进程。

为了保持以用户为中心并易于采用，您的系统应尽量减少必填配置问题的数量。这不是一件容易的事，但是很重要。人们可能会争辩说，增加一个或两个小步骤所花费的成本很少，但工程师的生命往往是无穷无尽的单个小步骤链。原则上减少这些小步骤可以大大提高生产率。

最初的强制性问题集通常包括您在设计系统时考虑的问题。减少必填问题的最简单方法是将其转换为可选问题，这意味着提供默认答案，这些答案可以安全有效地应用于大多数(如果不是全部)用户。例如，我们不需要简单地默认执行空运行，而不是要求用户定义执行是否应为空运行。

尽管此默认值通常是静态的硬编码值，但不一定必须如此。可以根据系统的其他属性动态确定。利用动态确定可以进一步简化您的配置。

对于上下文，请考虑以下动态默认值示例。计算密集型系统通常可能会决定通过配置控件部署多少个计算线程。它的动态默认部署与系统(或容器)具有执行核心的线程数量一样多。在这种情况下，单个静态默认值将无用。动态默认意味着我们不需要让用户确定要在给定平台上部署的系统的正确线程数。同样，单独部署在容器中的Java二进制文件可以根据容器中可用的内存自动调整其堆限制。这两个动态默认值示例反映了常见的部署。如果您需要限制资源使用，则能够覆盖配置中的动态默认值很有用。

实施的动态默认值可能不适用于每个人。随着时间的流逝，用户可能会喜欢不同的方法，并要求对动态默认值进行更好的控制。如果很大一部分配置用户报告动态默认问题，则您的决策逻辑可能不再符合当前用户群的要求。考虑实施广泛的改进，使动态默认设置可以运行而无需其他配置旋钮。如果只有一小部分用户不满意，最好手动设置配置选项。在系统中实现更多复杂性将为用户带来更多工作(例如，增加阅读文档的认知负担)。

在为可选问题选择默认答案时，无论选择静态还是动态默认值，都应仔细考虑选择的影响。经验表明，大多数用户将使用默认值，因此这既是机会也是责任。您可以朝着正确的方向微妙地推动人，但是指定错误的默认值会造成很大的伤害。例如，考虑配置默认值及其对计算机科学以外的影响。器官捐赠者默认为选择加入的国家(如果愿意的话，个人可以选择退出)有比默认值选择不加入的国家[84更多的动器官捐献者动态比例](#)。指定默认选项会对整个系统的医疗选择产生深远的影响。

一些可选问题在没有明确用例的情况下开始。您可能希望完全删除这些问题。大量的可选问题可能会使用户感到困惑，因此，只有在真正需要的情况下，才应添加配置旋钮。最后，如果您的配置语言恰好使用继承的概念，则对于叶配置中的任何可选问题，能够将其恢复为默认值很有用。

### 逃避简单性

到目前为止，我们已经讨论了将系统配置简化为最简单的形式。但是，配置系统可能还需要考虑高级用户。回到我们的茶类比，如果我们真的需要在特定时间内浸泡茶会怎样？

容纳高级用户的一种策略是找到普通用户和高级用户所需的最大公约数，并将默认复杂度定为该水平。不利的一面是，这一决定影响到所有人。即使是最简单的用例，现在也都需要从低级的角度来考虑。

通过根据默认行为的可选替代来考虑配置，用户可以配置“绿茶”，然后添加“将茶浸泡五分钟”。在此模型中，默认配置仍然是高级的并且接近用户的目标，但是用户可以微调低级方面。这种方法不是新颖的。我们可以与C++或Java之类的高级编程语言进行比较，这使程序员能够将机器(或VM)指令包含在用高级语言编写的代码中。在某些消费者软件中，我们看到带有高级选项的屏幕，这些屏幕可以提供比典型视图更多的细粒度控制。

考虑优化整个组织的配置时间是很有用的。不仅要考虑配置本身的行为，还要考虑用户在看到许多选项时可能会遇到的决策瘫痪，错误的转弯后纠正配置所花费的时间，由于置信度较低而导致的更改速度变慢等等。当您考虑配置设计替代方案时，如果可以简化对最常见用例的支持，那么以较少但难度较大的步骤完成复杂配置的选项可能更可取。

如果发现超过一小部分用户需要复杂的配置，则可能是错误地标识了常见的用例。如果是这样，请重新查看系统的初始产品假设并进行其他用户研究。

### 配置机制

至此，我们的讨论涵盖了配置哲学。本节将重点转移到用户如何与配置交互的机制上。

### 单独的配置和结果数据

用哪种语言存储配置是不可避免的问题。您可以选择具有纯数据，例如INI，YAML或XML文件中的数据。可替代地，可以以允许更加灵活的配置的高级语言来存储配置。

从根本上讲，所有询问用户的问题都归结为静态信息。这显然可以包括对诸如“应使用多少个线程？”之类问题的静态答案。但是，甚至“应该为每个请求使用什么功能？”只是对函数的静态引用。

为了回答配置是代码还是数据这一古老的问题，我们的经验表明，同时拥有代码和数据两者，但将两者分开是最佳选择。系统基础结构应在纯静态数据上运行，其格式可以为Protocol Buffers，YAML或[JSON] (<https://www.json.org/>)。这种选择并不意味着用户实际上需要与纯数据进行交互。用户可以与生成此数据的更高级别的界面进行交互。但是，API可以使用此数据格式，以允许进一步堆叠系统和自动化。

这个高级界面几乎可以是任何东西。它可以是高级语言，例如基于Python的域特定语言(DSL)，Lua或专用语言，例如Jsonnet(我们将在第15章中进行详细讨论)。我们可以将这样的接口视为编译器，类似于我们对待C++代码的方式。<sup>85</sup>高级接口也可能根本不是语言，并且Web UI会提取配置。

从故意与静态数据表示形式分离的配置UI开始，意味着系统具有部署的灵活性。各种组织可能有不同的文化规范或产品要求(例如使用公司内的特定语言或需要将配置外部化到最终用户)，并且这种通用的系统可以适应各种支持的配置要求。这样的系统还可以毫不费力地支持多种语言。<sup>86</sup>见图14-2。

用户可以完全看不见这种分离。用户的通用路径可能是使用配置语言编辑文件，而其他所有事情都在幕后发生。例如，一旦用户向系统提交更改，新存储的配置就会自动编译为原始数据。<sup>87</sup>

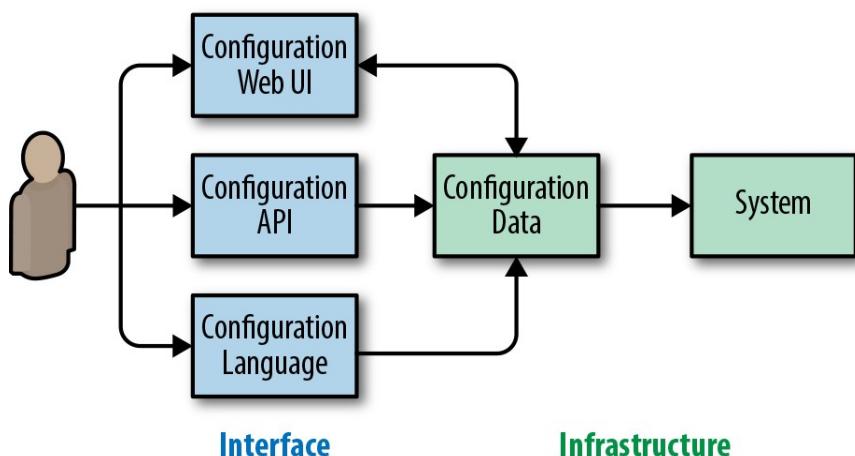


图14-2.具有单独的配置界面和配置数据基础结构的配置流。请注意，Web UI通常还会显示当前配置，从而使关系成为双向。

一旦获得静态配置数据，它也可以用于数据分析。例如，如果生成的配置数据为JSON格式，则可以将其加载到PostgreSQL中并通过数据库查询进行分析。作为基础架构所有者，您可以快速轻松地查询正在使用哪些配置参数以及由谁使用。该查询对于识别可以删除的功能或评估故障选项的影响很有用。

使用最终配置数据时，您会发现还存储有关如何提取配置的元数据很有用。例如，如果您知道数据来自Jsonnet中的配置文件，或者在原始数据被编译为数据之前拥有完整路径，则可以跟踪配置作者。

配置语言也可以是静态数据。例如，您的基础架构和接口都可能使用纯JSON。但是，请避免在用作接口的数据格式与内部使用的数据格式之间紧密耦合。例如，您可以在内部使用一个数据结构，其中包含从配置中使用的数据结构。内部数据结构可能还包含完全特定于实现的数据，这些数据永远不需要在系统外部显示。

### 工具的重要性

工具可以使混乱的噩梦和可持续的可扩展系统之间产生差异，但是在设计配置系统时通常会忽略它。本节讨论最佳配置系统应使用的关键工具。

#### 语义验证

尽管大多数语言都提供了开箱即用的语法验证，但是请不要忽略语义验证。即使您的配置在语法上有效，它也可能会做有用的事情？还是用户引用了一个不存在的目录(由于输入错误)，或者需要比其实际拥有的内存多一千倍的内存(因为单位不是用户期望的)？

在最大可能范围内验证配置在语义上的意义可以帮助防止中断并降低运营成本。对于每种可能的错误配置，我们应该问自己是否可以在用户提交配置时而不是在提交更改后阻止它。

#### 配置语法

确保配置完成用户所需的关键是关键，但消除机械障碍也很重要。从语法角度来看，配置语言应提供以下内容：

##### 在编辑器中突出显示语法(在公司内部使用)

通常，您已经通过重用现有语言来解决此问题。但是，特定于域的语言可能具有其他“语法糖”，可以从专门的突出显示中受益。

##### Linter

使用Linter来识别语言使用中的常见不一致的地方。[Pylint](#)是一种流行的语言示例。

##### 自动语法格式化器

内置的标准化可最大程度地减少相对不重要的格式化讨论，并减少贡献者切换项目时的认知负担。标准格式还可以简化自动编辑的过程，这在大型组织中广泛使用的系统中很有用。现有语言中自动格式化程序的示例包括[clang-format<sup>88</sup>](#)和[autopep8](#)。

这些工具使用户能够确信自己的语法正确，从而可以编写和编辑配置。[89](#)面向空白缩进的配置中的缩进不正确可能会带来严重的后果-其中一些是标准格式可以阻止。

#### 所有权和变更跟踪

因为配置可能会影响公司和机构的关键系统，所以重要的是要确保良好的用户隔离，并了解系统中发生了什么变化。如第10章所述，有效的事后文化应避免责怪个人。但是，这在事件发生期间和进行事后分析时都很有用，它有助于了解谁更改了配置并了解配置更改如何影响系统。无论事件是由于事故还是恶意行为者，这都是正确的。

系统的每个配置摘要都应具有明确的所有者。例如，如果使用配置文件，则它们的目录可能归一个生产组所有。如果目录中的文件只能有一个所有者，那么跟踪谁进行更改要容易得多。

无论版本配置如何执行，版本控制配置都允许您及时返回以查看在任何给定时间点的配置情况。如今，将配置文件检入到Subversion或Git之类的版本控制系统中已成为一种常见的做法，但是这种做法对于Web UI或远程API提取的配置同样重要。您可能还希望在配置和正在配置的软件之间建立更紧密的联系。这样，您可以避免无意中配置了软件不可用或不再支持的功能。

与此相关的是，将对配置的更改以及由此产生的应用程序都记录到系统中很有用(有时是必需的)。提交配置的新版本的简单动作并不总是意味着可以直接应用配置(稍后会详细介绍)。如果在事件响应期间怀疑系统配置更改是罪魁祸首，则能够快速确定更改所涉及的整套配置集很有用。这可以实现可靠的回滚，并能够通知配置受到影响的各方。

### **安全配置更改应用**

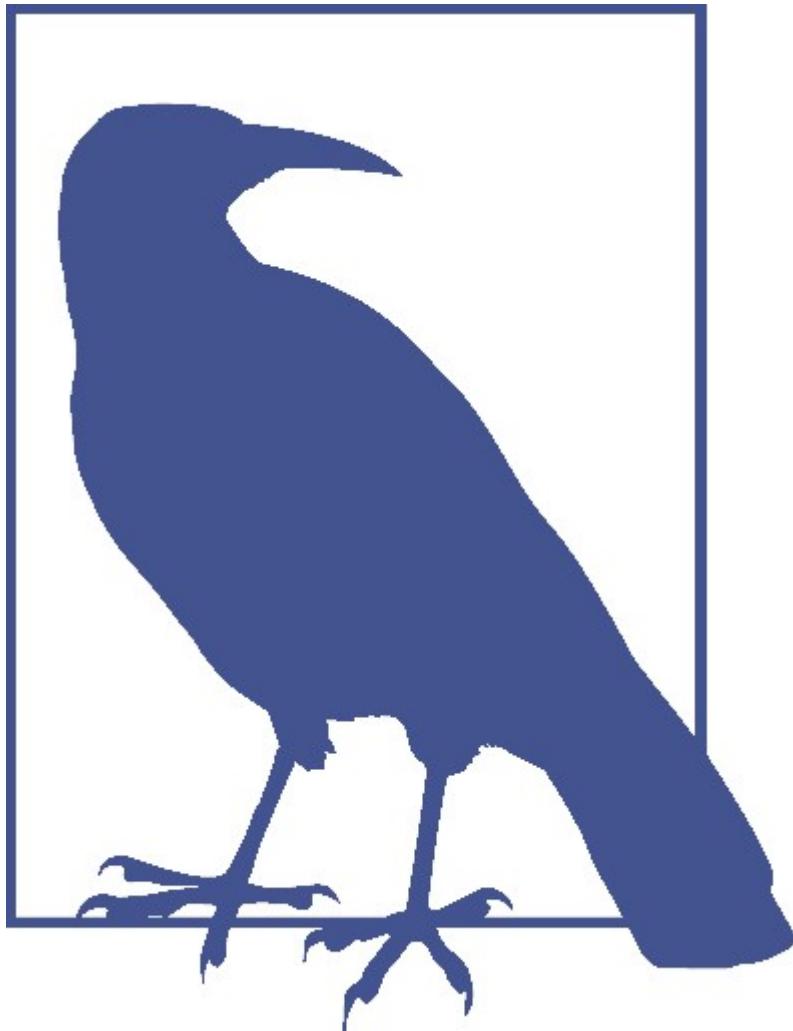
如前所述，配置是对系统功能进行重大更改的简便方法，但是它通常不是经过单元测试的，甚至不是易于测试的。由于我们希望避免发生可靠性事故，因此我们应该检查配置更改的安全应用意味着什么。

为了使配置更改安全，它必须具有三个主要属性：

- 逐步部署的能力，避免了全有或全无的变化
- 如果发现危险，则可以回滚更改
- 如果更改导致操作员失去控制，则自动回滚(或至少停止进度的能力)

部署新配置时，避免全局一次推送很重要。取而代之的是逐步推出新配置-这样做可以让您检测问题并在造成100%的停机之前中止有问题的推送。这就是为什么Kubernetes之类的工具使用滚动更新策略来更新软件或配置而不是一次更新每个Pod的原因之一。(有关讨论，请参见第16章。)

回滚的能力对于减少事件持续时间很重要。与尝试使用临时修复程序对其进行修补相比，回滚有问题的配置可以更快地缓解中断-人们对修补程序可以改善情况的信心较低。



为了能够前滚和后滚配置，它必须是封闭的。需要外部资源可以在其封闭环境之外更改的配置可能很难回滚。例如，存储在引用网络文件系统上数据的版本控制系统中的配置不是封闭的。

最后但并非最不重要的一点是，在处理可能导致突然失去操作员控制的更改时，系统应格外小心。在台式机系统上，如果用户不确认更改，屏幕分辨率的更改通常会提示倒计时并重置。这是因为不正确的监控器设置可能会阻止用户还原更改。同样，系统管理员通常会将自己意外地用防火墙阻隔出他们当前正在设置的系统。

这些原理不是唯一的配置，并且适用于更改已部署系统的其他方法，例如升级二进制文件或推送新数据集。

## 结论

微小的配置更改会以戏剧性的方式影响生产系统，因此我们需要刻意设计配置以减轻这些风险。配置设计包含API和UI设计两个方面，并且应该有目的性-不仅仅是系统实现的副作用。将配置分为原理和技术性细节两方面，有助于我们在设计内部系统时更加清晰，并使我们能够正确地进行讨论。

应用这些建议需要花费时间和精力。有关如何在实践中应用这些原理的示例，请参见[金丝雀 Analysis Service上的ACM队列文章](#)。在设计这个实用的内部系统时，我们花了大约一个月尝试减少必答问题并为可选问题找到良好答案。我们的努力创建

了一个简单的配置系统。由于易于使用，因此在内部被广泛采用。我们几乎不需要用户支持，因为用户可以轻松了解系统，因此可以放心地进行更改。当然，我们还没有完全消除配置错误和用户支持，也从未期望过。

<sup>82</sup>. Kim Vicente，《人为因素》(纽约: Routledge，2006)，71--6。 ↵

<sup>83</sup>. 虽然显然是夸张的，但有关最差音量控制接口的Reddit线程可以洞悉回答同一问题“音量应为多少”的好坏机制之间的区别。? ↵

<sup>84</sup>. 全文: <http://www.dangoldstein.com/papers/DefaultsScience.pdf>。 ↵

<sup>85</sup>. Jsonnet用于编译为Kubernetes YAML，为该并行提供了真实示例。请参阅 <http://ksonnet.heptio.com/>。 ↵

<sup>86</sup>. 当组织执行向新技术的迁移，整合并购或专注于共享基础结构，但在其他方面具有不同的开发和系统管理实践时，这将非常有用。 ↵

<sup>87</sup>. 有多种实用方法可以自动将新配置编译为原始数据。例如，如果将配置存储在版本控制系统中，则提交后挂钩可以简化此过程。可替代地，定期更新过程可以以一些延迟为代价来执行该操作。 ↵

<sup>88</sup>. 尽管C ++不太可能用于配置，但是[clang-format](#)很好地证明了，甚至比大多数用于配置的语言更复杂的语言也可以完全自动格式化。 ↵

<sup>89</sup>. 在非常大型的组织中，能够注释已弃用的广泛复用的配置也很有用。当有可用的替代产品时，自动重写工具可促进集中式更改，从而有助于避免遗留问题。 ↵

# 第15章

## 配置细节

**戴夫·坎宁安(Dave Cunningham)和米莎·布鲁克曼(Misha Brukman)与克里斯托弗·卡尔特(Christophe Kalt)和贝茜·拜尔(Betsy Beyer)**

管理生产系统是SRE为组织提供价值的多种方式之一。在生产环境中配置和运行应用程序的任务需要深入了解这些系统如何组合在一起以及它们如何工作。当出现问题时，值班工程师需要准确知道配置在哪里以及如何更改它们。如果团队或组织没有投资于解决与配置相关的工作，那么这种责任可能成为负担。

本书详细介绍了琐事的话题(见第6章)。如果您的SRE团队负担了许多与配置相关的工作，那么我们希望实施本章中介绍的一些想法将有助于您节省一些时间来进行配置更改。

## 配置引起的琐事

在项目生命周期的开始，配置通常是相对轻量和简单的。您可能有一些纯数据格式的文件，例如INI，JSON，YAML，或XML。管理这些文件几乎不需要费力。随着应用程序，服务器和变体的数量随时间增加，配置可能变得非常复杂和冗长。例如，您最初可能通过编辑一个配置文件来“更改设置”，但是现在您必须在多个位置更新配置文件。读取这样的配置也很困难，因为重要的差异隐藏在无关紧要的重复细节中。我们可以将与配置相关的工作描述为**重复琐事**:这是管理在系统中复制的配置的日常任务。这种工作不仅仅限于大型组织和大型系统，在具有许多独立配置组件的微服务架构中尤其常见。

工程师通常通过构建自动化或配置框架来应对复制工作。它们旨在消除配置系统中的重复项，并使配置更易于理解和维护。这种方法重用了软件工程中的技术，通常使用“配置语言”。Google SRE创建了许多配置语言，旨在减少我们最大，最复杂的生产系统的工作量。

不幸的是，这种策略并不一定消除配置方面的麻烦。从大量的个人配置中解放出来，该项目(及其配置语料库)随着新的活力而增长。不可避免地，您遇到了“复杂的琐事”:应对复杂的自动化的新的，有时是不良的行为的挑战性和令人沮丧的任务。这种琐事通常会在大型组织(超过10位工程师)中实现，并且会随着增长而增加。您越早解决复杂性问题就越好；配置的大小和复杂性只会随着时间的推移而增长。

### 减少配置引起的工作量

如果您的项目充斥着与配置相关的工作，那么您有几种改善情况的基本策略。

在极少数情况下，并且如果您的应用程序是定制构建的，则您可能会选择完全删除配置。在处理某些方面的配置时，该应用程序自然可以比配置语言更好:因为它可以访问有关机器的信息，或者可以动态地更改一些值，因为它可以根据负载进行扩展，因此可以为应用程序分配默认值。

如果删除配置不是一种选择，并且重复琐事成为问题，请考虑自动化以减少配置语料库中的重复。您可能集成了新的配置语言，或者可能需要改进或替换现有的配置设置。[90](#)下一部分，“配置系统的关键属性和陷阱”，第317页，提供了有关选择或设计该配置文件的一些指导。系统。

如果您要设置新的配置框架，则需要将配置语言与需要配置的应用程序集成在一起。“集成现有应用程序”第322页的“Kubernetes”将Kubernetes用作要集成的现有应用程序的示例，第326页的“集成自定义应用程序(内部软件)”提供了一些更一般的建议。这些部分介绍了一些使用Jsonnet的示例(出于示例目的，我们选择Jsonnet作为代表配置语言)。

一旦有了一个配置系统来帮助您进行复制工作-无论您是否已经致力于现有的解决方案，还是选择实现一种新的配置语言-都是“有效地操作配置系统”中的最佳做法第329页的“何时评估配置”和第333页的“防止滥用配置”应该有助于优化设置，无论使用哪种语言。采用这些流程和工具可以帮助最大程度地减少复杂性。

## 配置系统的关键属性和陷阱

第14章概述了任何配置系统的一些关键属性。除了轻量级，易学性，简单性和表达能力等通用理想要求之外，高效的配置系统还必须：

- 通过用于管理配置文件(工具，调试器，格式化程序，IDE集成等)的工具支持配置运行状况，工程师的信心和生产率。
- 提供密封评估配置，以实现回滚和一般重播性。
- 单独的配置和数据，可以轻松分析配置和一系列配置接口。

人们尚未普遍了解这些属性至关重要，因此达成我们目前的理解确实是一个旅程。在此过程中，Google发明了几种缺少这些关键属性的配置系统。我们也不是孤独的。尽管流行的配置系统种类繁多，但很难找到一个不会犯以下至少一个陷阱的系统。

### 陷阱1：无法将配置识别为编程语言问题

如果您不是故意设计一种语言，那么最终获得的“语言”就不太可能是一种好语言。

尽管配置语言描述数据而不是行为，但它们仍然具有编程语言的其他特征。如果我们的配置策略以仅使用数据格式为目标开始，则编程语言功能往往会在后门悄悄蔓延。该格式不是保留“仅数据”语言，而是成为一种深奥而复杂的“编程”语言。

例如，某些系统将count属性添加到要配置的虚拟机(VM)的架构中。此属性不是VM本身的属性，而是表示您想要多个。虽然有用，但这是

### 配置系统的关键属性和陷阱

编程语言，不是数据格式，因为它需要外部评估器或解释器。传统的编程语言方法将使用工作件外部的逻辑(例如for循环或列表理解)来根据需要生成更多的VM。

另一个示例是一种配置语言，它使用[字符串插值规则](#)而不支持通用表达式。尽管这些字符串实际上可以包含复杂的代码，包括数据结构操作，校验和，base64编码等，但它们似乎“仅仅是数据”。

流行的YAML + Jinja解决方案也有缺点。XML，JSON，YAML和文本格式的协议缓冲区等简单的纯数据格式是纯数据用例的绝佳选择。同样，文本模板引擎(例如Jinja2或Go模板)也非常适合HTML模板。但是，当将它们与配置语言结合使用时，对于人工和工具而言，它们都变得难以维护和分析。在所有这些情况下，这种陷阱使我们陷入了不适合工具使用的复杂而深奥的"语言"。

### 陷阱2: 设计偶发或临时语言功能

当大规模使用操作系统时，SRE通常会感到配置可用性问题。一种新语言将没有良好的工具支持(IDE支持，良好的linters)，并且如果该语言具有未记录的或深奥的语义，则开发自定义工具将很痛苦。

随着时间的推移将临时编程语言功能添加到简单的配置格式可能会创建功能完整的解决方案，但是临时语言比其正式设计的等效语言更复杂并且通常具有较低的表达能力。他们还冒着发展陷阱和特质的风险，因为他们的作者无法提前考虑要素之间的相互作用。

与其希望您的配置系统不会变得复杂到需要简单的编程构造，不如在初始设计阶段考虑这些要求。

### 陷阱3: 构建过多的特定领域优化

用于特定于域的新解决方案的用户群越小，您需要等待更长的时间来积累足够的用户来证明构建工具的合理性。工程师不愿意花时间正确理解该语言，因为它在该领域之外几乎没有适用性。[Stack Overflow](#)之类的学习资源不太可能获得。

### 陷阱4: 将"配置评估"与"副作用"交织在一起

副作用包括在配置运行期间更改外部系统或咨询带外数据源(DNS，VM ID，最新内部版本)。

允许这些副作用的系统违反了密闭性，并且还防止了配置与数据的分离在极端情况下，如果不花时间通过保留云资源来调试配置是不可能的。为了允许配置和数据分离，请首先评估配置，然后将结果数据提供给用户进行分析，然后再考虑副作用。

### 陷阱5: 使用现有的通用脚本语言(例如Python，Ruby或Lua)

这似乎是避免前四个陷阱的简单方法，但是使用通用脚本语言的实现是重量级的，并且/或者需要使用侵入式沙箱来确保密封性。由于通用语言可以访问本地系统，因此出于安全考虑，也可能需要沙盒。

此外，我们不能假设维护配置的人员会熟悉所有这些语言。

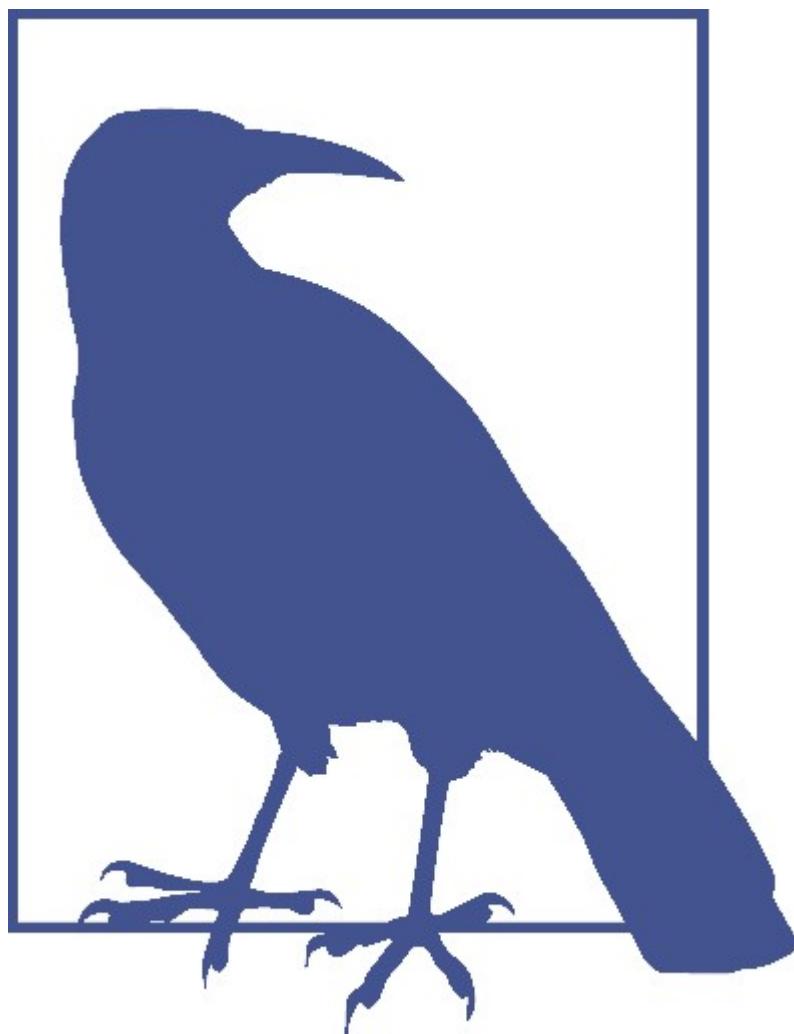
避免这些陷阱的愿望导致了可配置的可重用领域特定语言(DSL)的开发，例如[HOCON](#)，[Flabbergast](#)、[Dhall](#)和[Jsonnet](#)。我们建议使用现有DSL进行配置。即使DSL似乎无法满足您的需求，您有时还是需要其他功能，并且您可以始终使用内部样式指南来限制该语言的功能。

### Jsonnet快速入门

Jsonnet是一种封闭的开源DSL，可用作库或命令行工具以为任何应用程序提供配置。它在Google内部和外部均得到广泛使用。[91](#)

该语言旨在使程序员熟悉：它使用类似Python的语法，面向对象和功能构造。它是JSON的扩展，意味着JSON文件只是一个输出自身的Jsonnet程序。与JSON相比，Jsonnet在引号和逗号中更宽容，并且支持注释。更重要的是，它增加了计算结构。

尽管您无需特别熟悉Jsonnet语法即可遵循本章的其余部分，但是花一些时间阅读[在线教程](#)可以帮助您进行定向。



Google或我们的读者群中没有主流的配置语言，但是我们需要选择某种语言这样允许我们提供例子。本章使用Jsonnet展示了我们在第14章中提供的建议的实际示例。

如果您尚未使用特定的配置语言，并且想使用Jsonnet，则可以直接应用本章中的示例。在所有情况下，我们已尽力使您尽可能轻松地从代码示例中提取基础课程。

另外，一些示例探讨了您可能希望在编程书中找到的概念(例如图灵完整性)。我们非常小心地将深度降至要求的深度，以解释实际上已经在生产中咬住我们的微妙之处。在大多数复杂的系统中(当然还有配置方面)，故障都在

边缘。

## 集成配置语言

本节使用Jsonnet讨论如何将配置语言与需要配置的应用程序集成在一起，但是相同的技术也可以转移到其他配置语言。

### 以特定格式生成配置

配置语言可能以正确的格式本地输出。例如，Jsonnet输出与许多应用程序兼容的JSON。对于使用JSON，JavaScript，YAML或[HashiCorp的配置语言](#)之类的扩展JSON的消费者，JSON也足够了。如果是这种情况，则无需执行任何进一步的集成工作。

对于本机不支持的其他配置格式：

1. 您需要找到一种在配置语言中表示配置数据的方法。通常，这并不难，因为配置值(如映射，列表，字符串和其他原始值)是通用的，并且在所有语言中均可使用。
2. 一旦这些数据以配置语言表示，您就可以使用该语言的构造来减少重复(从而减少琐事)。
3. 您需要为必要的输出格式编写(或重用)序列化函数。例如，Jsonnet标准库具有用于从其内部类似JSON的表示中输出INI和XML的功能。如果配置数据拒绝使用配置语言(例如Bash脚本)表示，则可以使用基本的字符串模板技术作为最后的手段。您可以在<http://bit.ly/2La0zDe>上找到实际示例。

### 驱动多个应用程序

一旦可以使用配置语言驱动任意现有应用程序，您就可以从同一配置中定位多个应用程序。如果您的应用程序使用不同的配置格式，则需要执行一些转换工作。一旦能够以必要的格式生成配置，就可以轻松地统一，同步并消除整个配置语料库中的重复。考虑到JSON和基于JSON的格式的普遍性，您甚至不必生成其他格式-例如，如果您使用[GCP Deployment Manager](#)，[AWS Cloud Formation](#)或[Terraform](#)用于基础架构，以及用于容器的Kubernetes。此时，您可以：

- 仅定义一次端口的Jsonnet计算中输出Nginx Web服务器配置和Terraform防火墙配置。
- 从同一文件配置监控仪表板，保留策略和警报通知管道。
- 通过将初始化命令从一个列表移动到另一个列表，来管理VM启动脚本和磁盘映像构建脚本之间的性能折衷。

将完全不同的配置集中到一处之后，您就有很多机会可以优化和抽象配置。配置甚至可以嵌套-例如，[Cassandra配置](#)可以嵌入其基础基础架构的Deployment Manager配置中或Kubernetes ConfigMap中。好的配置语言可以处理任何尴尬的字符串引用，并且通常使此操作自然而简单。

为了使为各种应用程序编写许多不同的文件变得容易，Jsonnet提供了一种模式，该模式期望配置执行产生一个JSON对象，该JSON对象将文件名映射到文件内容(根据需要设置格式)。您可以使用其他配置语言来模拟此功能，方法是在字符串之间发出映射，并使用后处理步骤或包装脚本编写文件。

## 集成配置语言

### 集成现有应用程序:Kubernetes

Kubernetes进行一个有趣的案例研究有两个原因:

- 需要配置在Kubernetes上运行的作业，其配置可能会变得复杂。
- Kubernetes没有附带捆绑的配置语言(谢天谢地，甚至还没有一种特定的配置语言)。

具有最小复杂对象的Kubernetes用户只需使用YAML。具有较大基础架构的用户可以使用Jsonnet之类语言扩展其Kubernetes工作流，以提供该规模所需的抽象工具。

#### Kubernetes提供什么

Kubernetes是一个开源系统，用于在机器集群上协调容器化工作负载。它的API使您可以管理容器本身以及许多重要的细节，例如容器之间的通信，集群内外的通信，负载平衡，存储，渐进式部署和自动扩展。每个配置项都由一个JSON对象表示，该对象可以通过API端点进行管理。命令行工具kubectl允许您从磁盘读取这些对象并将其发送到API。

在磁盘上，JSON对象实际上被编码为YAML流。<sup>92</sup> YAML易于阅读，并可以通过通用库轻松转换为JSON。开箱即用的用户体验包括编写代表Kubernetes对象的YAML文件，并运行kubectl将它们部署到集群。

要了解配置Kubernetes的最佳实践，请参阅[关于该主题的Kubernetes文档](#)。

#### 示例Kubernetes配置

YAML是Kubernetes配置的用户界面，提供了一些简单的功能(例如注释)，并且具有简洁的语法，大多数人都喜欢原始JSON。但是，YAML在抽象方面不够完善:它仅提供锚点，<sup>93</sup> 在实践中很少使用，Kubernetes不支持。

假设您要使用不同的名称空间，标签和其他较小的变体来复制Kubernetes对象四次。遵循不变基础架构的最佳实践，您将存储所有四个变体的配置，从而复制配置的其他相同方面。以下代码段提供了一种变体(为简洁起见，我们省略了其他三个文件):

```

# example1.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: guestbook
    tier: frontend
  name: frontend
  namespace: prod
spec:
  externalTrafficPolicy: Cluster
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: NodePort

```

这些变体很难阅读和维护，因为重要的差异被掩盖了。

### 集成配置语言

如第315页的“配置导致的负担”中所述，管理大量的YAML文件可能会花费大量时间。配置语言可以帮助简化此任务。最直接的方法是，在每次执行Jsonnet时都发出一个Kubernetes对象，然后将生成的JSON直接传递到kubectl中，后者像处理YAML一样处理JSON。或者，您可以发出YAML流(此类对象的序列<sup>94</sup>)或单个kubectl列表对象，或者让Jsonnet从同一配置中发出多个文件。有关更多讨论，请参见[Jsonnet网站](#)。

开发人员应注意，通常，YAML允许您编写在JSON中无法表达的配置(因此，Jsonnet无法生成)。YAML配置可以包含异常的IEEE浮点值(如NaN)或具有非字符串字段的对象(如数组，其他对象或null)。实际上，这些功能很少使用，而且Kubernetes不允许使用这些功能，因为在将配置发送到API时必须对其进行JSON编码。

以下代码片段显示了我们的示例Kubernetes配置在Jsonnet中的显示效果：

```

// templates.libsonnet
{
  MyTemplate:: {
    local service = self,
    tier:: error 'Needs tier',
    apiVersion: 'v1',
    kind: 'Service',
    local selector_labels = { app: 'guestbook', tier: service.tier },
    metadata: {
      labels: selector_labels,
      name: 'guestbook-' + service.tier, namespace: 'default',
    },
    spec: {
      externalTrafficPolicy: 'Cluster',
      ports: [{
        port: 80,
        protocol: 'TCP',
        targetPort: 80,
      }],
      selector: selector_labels, sessionAffinity: 'None', type: 'NodePort',
    },
  },
}

// example1.jsonnet
local templates = import 'templates.libsonnet';
templates.MyTemplate { tier: 'frontend',
}

// example2.jsonnet
local templates = import 'templates.libsonnet';
templates.MyTemplate { tier: 'backend', metadata+: {
  namespace: 'prod',
},
}

// example3.jsonnet
local templates = import 'templates.libsonnet';
templates.MyTemplate { tier: 'frontend', metadata+: {
  namespace: 'prod',
  labels+: { foo: 'bar' },
},
}

// example4.jsonnet
local templates = import 'templates.libsonnet';
templates.MyTemplate { tier: 'backend',
}

```

请注意以下几点:

- 我们通过实例化抽象模板四次来表示所有四个变体，但是您也可以使用功能抽象。
- 虽然我们为每个实例使用单独的Jsonnet文件，但您也可以将它们合并到一个文件中。
- 在抽象模板中，名称空间默认为默认，并且层必须被覆盖。
- 乍一看，Jsonnet稍微冗长一些，但是随着模板实例化数量的增加而减少了工作量。

在MyTemplate中，local关键字定义了一个变量服务，该服务被初始化为self(对最近的封闭对象的引用)。这样，您就可以从嵌套对象(重新定义self)中引用该对象。

tier字段具有两个冒号(而不是常规JSON单个冒号)，并且在生成的JSON中隐藏(不输出)。否则，Kubernetes将拒绝tier作为无法识别的字段。隐藏的字段仍然可以被覆盖和引用-在这种情况下，作为service.tier。

该模板本身不能使用，因为引用service.tier会触发错误构造，从而使用给定的文本引发运行时错误。为避免该错误，模板的每个实例都使用其他一些表达式覆盖tier字段。换句话说，此模式表示类似于纯虚拟/抽象方法的内容。

使用函数进行抽象意味着只能对config进行参数化。相反，模板允许您覆盖父项中的任何字段。如中所述

### 集成现有应用程序:Kubernetes

第14章，虽然简单性是设计的基础，但逃避简单性的能力很重要。模板替代提供了一个有用的转义填充，用于更改通常被认为太低级的特定细节。例如:

```
templates.MyTemplate { tier: 'frontend', spec+: {
    sessionAffinity: 'ClientIP',
},
}
```

这是将现有模板转换为Jsonnet的典型工作流程:

1. 将YAML变体之一转换为JSON。
2. 通过Jsonnet格式化程序运行结果JSON。
3. 手动添加Jsonnet构造以抽象和实例化代码(如示例中所示)。

该示例说明了如何在保留某些不同字段的同时删除重复项。随着差异变得更加微妙(例如，字符串略有不同)或难以表达(例如，配置具有结构上的差异，如数组中的其他元素，或应用于数组中所有元素的相同差异)，使用配置语言将变得更具吸引力。

通常，跨不同配置抽象公共性可促进关注点分离，并具有与编程语言中的模块化相同的好处。您可以在许多不同的用例中利用抽象功能:

- 一个团队可能需要创建几乎相同(但不是完全相同)的多个版本的配置-例如，在跨各种环境(产品/阶段/开发/测试)管理部署时，在不同的架构，或在不同地区调整容量。

- 组织可能拥有一个基础架构团队，该团队维护着由应用程序团队使用的可重用组件-API服务框架，缓存服务器或MapReduces。对于每个组件，基础架构团队都可以维护一个模板，该模板定义大规模运行该组件所需的Kubernetes对象。每个应用程序团队都可以实例化该模板以添加其应用程序的详细信息。

### 集成自定义应用程序(内部软件)

如果您的基础架构使用了任何自定义应用程序(即，内部开发的软件，而不是现成的解决方案)，则可以设计这些应用程序与可重用的配置语言共存。当您编写配置文件或与生成的配置数据进行交互时(例如，出于调试目的或与其他工具集成时)，本节中的建议应改善总体用户配置体验。他们还应该简化应用程序的设计，并将配置与数据分开。

处理自定义应用程序的广泛策略应该是：

- 让配置语言处理其被设计用来处理的部分:问题的语言方面。
- 让您的应用程序处理所有其他功能。

以下最佳实践包括使用Jsonnet的示例，但相同的建议也适用于其他语言：

- 使用一个纯数据文件，然后让配置语言使用导入将配置拆分为文件。这意味着配置语言实现仅需发出(并且应用程序仅需使用)单个文件。而且，由于应用程序可以以不同的方式组合文件，因此该策略明确地描述了如何组合文件以形成应用程序配置。
- 使用对象表示命名实体的集合，其中字段包含对象名称，值包含实体的其余部分。避免使用每个元素都有名称字段的对象数组。

错误的JSON:

```
[  
  { "name": "cat", ... },  
  { "name": "dog", ... }  
]
```

良好的JSON:

```
{  
  "cat": { ... },  
  "dog": { ... }  
}
```

此策略使集合(和单个动物)更易于扩展，您可以按名称引用实体(例如，`Animals.cat`)，而不是引用脆弱

- 避免在顶层按类型对实体进行分组。结构化JSON，以便将与逻辑相关的配置分组在同一子树中。这允许抽象(在配置语言级别)遵循功能边界。

错误的JSON:

```
```
{
  "pots": { "pot1": { ... }, "pot2": { ... } },
  "lids": { "lid1": { ... }, "lid2": { ... } }
}
```
```

```

良好的JSON:

```
```
{
  "pot_assembly1": { "pot": { ... }, "lid": { ... } },
  "pot_assembly2": { "pot": { ... }, "lid": { ... } }
}
```
```

```

在配置语言级别，此策略启用如下的抽象:

```
```
local kitchen = import 'kitchen.libsonnet';
{
  pot_assembly1: kitchen.CrockPot,
  pot_assembly2: kitchen.SaucePan { pot+: { color: 'red' } },
}
```
```

```

- 通常使数据表示设计简单:

---避免在数据表示形式中嵌入语言功能(如“陷阱1:未能将配置识别为编程语言问题”(第317页)。这些类型的抽象将功能不足，只会造成混乱，因为它们迫使用户决定是在数据表示形式还是在配置语言中使用抽象功能。

---不用担心过于冗长的数据表示。减少冗长的解决方案会带来复杂性，并且可以使用配置语言来解决问题。

---避免在您的应用程序中解释自定义字符串插值语法，例如条件或字符串中的占位符引用。有时解释是不可避免的-例如，当您需要描述在生成配置的纯数据版本(警报，处理程序等)之后执行的操作时。但是否则，请让配置语言尽可能多地执行语言级别的工作。

如前所述，如果您可以完全删除配置，那么这样做始终是您的最佳选择。尽管配置语言可以通过使用具有默认值的模板来隐藏基础模型的复杂性，但是生成的配置数据并未完全隐藏-可以由工具进行处理，由人工检查或加载到配置数据库中。出于同样的原因，不要依赖配置语言来解决基础模型中不一致的命名，复数或错误-而是将它们修复在模型本身中。如果您无法解决模型中的不一致问题，那么最好在语言级别上使用它们，以避免更多的不一致之处。

根据我们的经验，随着时间的推移，配置更改往往会导致系统的中断根本原因(请参阅附录C中的主要中断原因列表)。验证配置更改是保持可靠性的关键步骤。我们建议配置执行后立即验证生成的配置数据。仅语法验证(即检查JSON是否可解析)不会发现很多错误。通用模式验证后，检查特定于应用程序域的属性-例如，是否存在必填字段，是否存在引用的文件名以及所提供的值在允许的范围内。

您可以使用[JSONSchema](#)验证Jsonnet的JSON。对于使用[protocol buffers](#)的应用程序，您可以轻松地从Jsonnet生成这些缓冲区的规范JSON形式，并且协议缓冲区实现将在反序列化期间进行验证。

不管您决定如何验证，都不要忽略无法识别的字段名称，因为它们可能表示在配置语言级别上有错字。Jsonnet可以屏蔽不应使用::语法输出的字段。在precommit hook中执行相同的验证也是一个好主意。

## 有效地操作配置系统

在以任何语言实现“代码配置”时，我们建议您遵循有助于整体软件工程的规则和流程。

### 版本化

配置语言通常会触发工程师编写模板和实用程序功能的库。通常，一个团队维护这些库，但是许多其他团队可能会使用它们。当需要对库进行重大更改时，有两种选择：

- 提交所有客户端代码的全局更新，以重构代码以使其仍然有效(这在组织上可能不可行)。
- 对库进行版本控制，以便不同的使用者可以使用不同的版本并独立迁移。选择使用不推荐使用的版本的用户将无法获得新版本的好处，并且会招致技术性债务-某天，他们将不得不重构代码以使用新的库。

包括Jsonnet在内的大多数语言都没有提供对版本控制的任何特定支持。相反，您可以轻松使用目录。有关Jsonnet中的实际示例，请参见[ksonnet-lib存储库](#)，其中版本是导入路径的第一个组件：

```
local k = import 'ksonnet.beta.2/k.libsonnet';
```

### 源代码控制

第14章主张保留配置更改的历史记录(包括进行更改的人)，并确保回滚容易且可靠。将配置检入到源代码管理中将带来所有这些功能，以及通过代码查看配置更改的功能。

### 工具

考虑如何实施样式和配置，并调查是否存在将这些工具集成到工作流程中的编辑器插件。您的目标是在所有作者之间保持一致的风格，提高可读性并检测错误。一些编辑器支持post-write hook，这些hook可以为您运行格式化程序和其他外部工具。您也可以使用precommit hook来运行相同的工具，以确保签入的配置是高质量的。

### 测试

我们建议对上游模板库实施单元测试。确保以各种方式实例化这些库时，它们会生成预期的具体配置。同样，功能库应包括单元测试，以便可以放心地对其进行维护。

在Jsonnet中，您可以将测试编写为Jsonnet文件，这些文件包括：

1. 导入要测试的库。
2. 练习这个库。
3. 使用assert语句或标准库assertEqual函数来验证其输出。后者会在其错误消息中显示任何不匹配的值。

以下示例测试joinName函数和MyTemplate：

```
// utils_test.jsonnet
local utils = import 'utils.libsonnet';
std.assertEqual(utils.joinName(['foo', 'bar']), 'foo-bar') && std.assertEqual(utils.
```

对于较大的测试套件，您可以利用[由Jsonnet社区成员开发的更全面的单元测试框架](#)。您可以使用此框架以结构化的方式定义和运行测试套件-例如，报告所有失败测试的集合，而不是在第一个失败的断言时中止执行。

## 何时评估配置

我们的关键特性包括密封性；也就是说，无论配置语言在何处或何时执行，都必须生成相同的配置数据。如第14章所述，如果系统依赖于可以在其封闭环境之外更改的资源，则可能难以回滚或无法回滚。通常，封闭性意味着Jsonnet代码始终可以与其表示的扩展JSON互换。因此，您可以在更新Jsonnet到需要JSON的任何时间（甚至每次都需要JSON时）从Jsonnet生成JSON。

我们建议将配置存储在版本控制中。然后，您最早来验证配置的机会是在签入之前。另一方面，应用程序可以在需要JSON数据时评估配置。作为中间选项，您可以在构建时进行评估。这些选项中的每一个都有各种折衷，您应该根据用例的具体情况进行优化。

### 很早期：签入JSON

您可以从Jsonnet代码生成JSON，然后再将其签入版本控制。典型的工作流程如下：

1. 修改Jsonnet文件。
2. 运行Jsonnet命令行工具（可能包装在脚本中）以重新生成JSON文件。
3. 使用precommit hook来确保Jsonnet代码和JSON输出始终保持一致。
4. 将所有内容打包到请求请求中以进行代码审查。

#### 优点

- 审阅者可以全面检查具体更改-例如，重构完全不影响生成的JSON。

- 您可以在生成和抽象级别检查多个作者在不同版本上的行注释。这对于审核更改很有用。
- 您不需要在运行时运行Jsonnet，这可以帮助限制复杂性，二进制文件大小和/或风险暴露。

#### 缺点

- 生成的JSON不一定是可读的-例如，如果它嵌入了长字符串。
- JSON可能由于其他原因而不适用于版本控制-例如，如果它太大或包含秘密。
- 如果用于分隔Jsonnet文件的许多并发编辑收敛到单个JSON文件，则可能会发生合并冲突。

### 中期:在构建时评估

您可以通过在构建时运行Jsonnet命令行实用程序并将生成的JSON嵌入到发布工作中(例如，作为tarball)来避免将JSON检入到源代码控制中。应用程序代码仅在初始化时从磁盘读取JSON文件。如果您使用的是Bazel，则可以使用[Jsonnet Bazel规则]轻松实现这一目标。<http://bit.ly/2xz0QxH>在Google上，由于下面列出的优点，我们通常偏爱这种方法。

#### 优点

- 您可以控制运行时复杂性，二进制文件大小和风险敞口，而不必在每个拉取请求中重建JSON文件。
- 在原始Jsonnet代码和生成的JSON之间不存在失去同步的风险。

#### 缺点

- 构建更加复杂。
- 在代码检查期间很难评估具体更改。

### 后期:在运行时评估

链接Jsonnet库可让应用程序本身随时解释配置，从而生成一个生成的JSON配置的内存表示形式。

#### 优点

- 更简单，因为您不需要事先评估。
- 您可以在执行期间评估用户提供的Jsonnet代码。

#### 缺点

- 任何链接的库都会增加占用空间和风险。
- 可能在运行时发现配置错误，为时已晚。
- 如果Jsonnet代码不受信任，则必须格外小心。(我们在第333页的"防范滥用配置"中讨论了原因。)

遵循我们的运行示例，如果要生成Kubernetes对象，什么时候应该运行Jsonnet？

答案取决于您的实现。如果您要构建类似ksonnet(从本地文件系统运行Jsonnet代码的客户端命令行工具)之类的东西，最简单的解决方案是将Jsonnet库链接到该工具并评估过程中的Jsonnet。这样做是安全的，因为代码在作者自己的计算机上运行。

Box.com的基础结构使用Git hook将配置更改推送到生产环境。为了避免在服务器上执行Jsonnet，Git hook会对存储在存储库中的生成的JSON起作用。对于像Helm或Spinnaker这样的部署管理守护程序，您唯一的选择是在运行时评估服务器上的Jsonnet(下一节将描述警告)。

### 防止滥用配置

与长期运行的服务不同，配置执行应以生成的配置迅速终止。不幸的是，由于错误或故意攻击，配置可能会花费任意数量的CPU时间或内存。为了说明原因，请考虑以下非终止Jsonnet程序:

```
local f(x) = f(x + 1); f(0)
```

使用无限制内存的程序类似:

```
local f(x) = f(x + [1]); f([])
```

您可以使用对象而不是函数或其他配置语言来编写等效的示例。

您可能会通过限制语言来避免过度消耗资源，从而使其不再图灵完备。但是，强制所有配置终止并不一定可以防止过度消耗资源。编写消耗足够时间或内存以至于几乎无法终止的程序很容易。例如:

```
local f(x) = if x == 0 then [] else [f(x - 1), f(x - 1)]; f(100)
```

实际上，即使使用简单的配置格式(例如XML和YAML)，[此类程序也存在](#)。

### 防止滥用配置

实际上，这些场景的风险取决于情况。在问题较少的方面，假设命令行工具使用Jsonnet构建Kubernetes对象，然后部署这些对象。在这种情况下，Jsonnet代码是受信任的:非终止的事故很少，您可以使用Ctrl-C缓解它们。偶然的内存耗尽是极不可能的。另一方面，对于像Helm或Spinnaker这样的服务，该服务从最终用户接受任意配置代码并在请求处理程序中对其进行评估，您必须非常小心，以避免可能会占用请求处理程序或耗尽内存的DOS攻击。

如果在请求处理程序中评估不受信任的Jsonnet代码，则可以通过对Jsonnet执行沙盒操作来避免此类攻击。一种简单的策略是使用单独的进程和ulimit(或它的非UNIX等效项)。通常，您需要派生到命令行可执行文件，而不是链接Jsonnet库。结果，未在给定资源内完成的程序将安全地失败并通知最终用户。为了进一步防御C++内存漏洞，可以使用Jsonnet的本地Go实现。

## 结论

无论您使用Jsonnet，采用其他配置语言还是自行开发，我们希望您可以应用这些最佳实践来自信地配置生产系统所需的复杂性和操作负载。

至少，配置语言的关键属性是良好的工具，**密封配置以及配置和数据的分隔**。

您的系统可能不够复杂，不需要配置语言。过渡到特定领域的语言(如Jsonnet)是一种在复杂性增加时要考虑的策略。这样做将使您能够提供一致且结构合理的界面，并腾出您的SRE团队用于其他重要项目的时间。

<sup>90</sup>. 请注意，您可能可以编写软件将旧的配置语言转换为新的语言。但是，如果您的原始源语言是非标准语言或已损坏，则这不是可行的选择。 ↵

<sup>91</sup>. 从计算生物学到视频游戏的领域都使用Jsonnet，但最热心的采用者来自Kubernetes社区。Box.com使用Jsonnet来描述在其基于Kubernetes的内部基础架构平台上运行的工作负载。Databricks和Bitnami也广泛使用该语言。

↵

<sup>92</sup>. YAML流是一个文件，其中包含许多由\" -- \"分隔的YAML文档。 ↵

<sup>93</sup>. YAML规范§6.9.2。 ↵

<sup>94</sup>. 在YAML规范中，对象被称为文档。 ↵

# 第16章

## 金丝雀发布

由Alec Warner和Štěpán Davidovič

与Alex Hidalgo , Betsy Beyer , Kyle Smith和Matt Duftler撰写

"发布工程"是一个术语，我们用来描述与将代码从存储库获取到正在运行的生产系统有关的所有过程和工作。自动化发布可以帮助避免与发布工程相关的许多传统陷阱:重复性和手动任务的繁琐工作，非自动化过程的不一致，无法知道发布的确切状态以及回滚的困难。发布工程的自动化在其他文献中也有很好的论述，例如，关于持续集成和持续交付(CI/CD)的书籍。<sup>95</sup>

我们将金丝雀定义为部分或有限时间内的服务变更部署和评估该评估有助于我们确定是否继续进行推广。服务中接收更改的部分是"金丝雀"，而服务的其余部分是"对照组"。支持这种方法的逻辑是，通常在较小的生产子集上执行金丝雀部署，或者与控制部分相比，对用户群的影响要小得多。金丝雀实际上是A/B测试过程。

我们将首先介绍发布工程的基础知识以及将发布自动化以建立共享词汇表的好处。

## 发布工程原理

发布工程的基本原理如下:

### 可复制的版本

构建系统应该能够获取构建输入(源代码，资产等)并产生可重复的工作。上周使用相同输入构建的代码本周应产生相同输出。

### 自动构建

检入代码后，自动化应产生构建工件并将其上传到存储系统。

### 自动测试

一旦自动构建系统构建了工件，则应使用某种测试套件确保其正常工作。

### 自动部署

部署应由计算机而非人员执行。

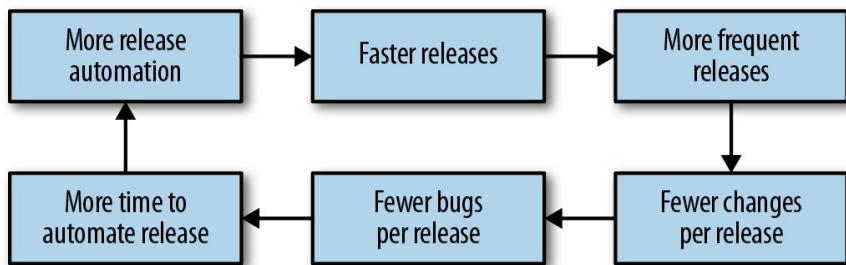
### 小型部署

构建工件应包含小的，独立的更改。这些原则为运维带来了特殊的好处:

- 通过消除手动和重复操作来减轻工程师的操作负担
- 任务。
- 强对等审查和版本控制，因为自动化通常基于代码。

- 建立一致，可重复的自动化流程，减少错误。
- 启动发布管道的监控，通过解决以下问题来进行评估和持续改进：
  - 发布需要多长时间才能投入生产？
  - 发布多久成功一次？成功发布是指没有严重缺陷或违反SLO的客户可以使用的发布。
  - 可以进行哪些更改以尽早在管道中发现缺陷？
  - 哪些步骤可以并行化或进一步优化？

CI/CD与发布自动化相结合可以持续改进开发周期，如图16-1所示。自动发布时，您可以更频繁地发布。对于变更率不高的软件，发布频率更高意味着在任何给定的发布工件中捆绑的更改都更少。较小的，自包含的发布工件使在发生错误时回滚任何给定的发布工件变得成本更低，更容易。更快的发布节奏意味着漏洞修复可以更快地到达用户。



## 平衡发布速度和可靠性

发布速度(以下称为"运输")和可靠性通常被视为相反的目标。该企业希望以100%的可靠性尽快交付新功能和产品改进！虽然这个目标是无法实现的(因为100%永远不是可靠性的正确目标；请参阅第2章)，但可以在满足给定产品特定可靠性目标的情况下尽快发货。

朝此目标迈出的第一步是了解运输对软件可靠性的影响。根据Google的经验，大多数事件是由二进制或配置推送触发的(请参阅附录C)。许多软件更改都可能导致系统故障-例如，基础组件的行为更改，依赖项(例如API)的行为更改或DNS之类的配置更改。

尽管更改软件存在固有的风险，但是这些更改(错误修复，安全补丁和新功能)对于业务成功而言是必不可少的。您可以使用SLO和错误预算的概念来衡量版本对可靠性的影响，而不是提倡更改。您的目标应该是在满足用户期望的可靠性目标的同时尽快交付软件。下一节讨论如何使用金丝雀过程来实现这些目标

目标。

### 以不同的速率分离变化的组件

您的服务由变化率不同的多个组件组成:二进制或代码，JVM，内核/OS，库，服务配置或标志等环境，功能/实验配置以及用户配置。如果只有一种部署变更方式，实际上很难让这些组件独立更改。

功能标记或实验框架，例如[Gertrude](#)，[Feature](#)和[PlanOut](#)允许您将功能启动与二进制发布分开。如果二进制发行版包含多项功能，则可以通过更改实验配置来一次启用一项功能。这样，您不必将所有这些更改批量处理为一个大更改，也不必为每个功能执行单独的发行。更重要的是，如果只有一些新功能无法按预期运行，则可以选择地禁用这些功能，直到下一个构建/发布周期可以部署新的二进制文件为止。

您可以将功能标记/实验的原理应用于服务的任何类型的更改，而不仅仅是软件版本。

## 什么是金丝雀发布？

术语“金丝雀发布”是指[将金丝雀带入煤矿以确定煤矿对人类是否安全的做法](#)。由于鸟类比人类更小，呼吸速度更快，因此被危险气体中毒的速度比人类操作者更快。

即使您的发布管道是完全自动化的，您也将无法检测到所有与发布相关的缺陷，直到真正的流量到达服务为止。在准备好将发行版部署到生产中时，您的测试策略应灌输合理的信心使发行版安全并且可以按预期工作。但是，您的测试环境与生产环境并非100%相同，并且您的测试可能无法涵盖100%的可能情况。一些缺陷将达到生产。如果一个版本立即部署到任何地方，则任何缺陷都将以相同的方式部署。

如果您可以快速检测并解决缺陷，则这种情况是可以接受的。

但是，您有一个更安全的选择:最初使用金丝雀发布将您的部分生产流量暴露给新版本。通过金丝雀发布，部署管道可以尽快检测到缺陷，而对服务的影响则尽可能小。

## 发布工程和金丝雀发布

在部署新版本的系统或其关键组件(例如配置或数据)时，我们会将更改捆绑在一起，这些更改通常是未公开给现实世界的输入，例如面向用户的流量或用户的批处理提供的数据。变更带来了新的特性和功能，但同时也带来了部署时暴露的风险。我们的目标是通过测试流量的一小部分变化来减轻这种风险，从而使人们确信它不会产生不良影响。我们将在本章后面讨论评估过程。

金丝雀过程还使我们对变化感到信心十足，因为我们面临着越来越多的流量。将更改引入实际生产流量还使我们能够识别在单元测试或负载测试之类的测试框架中可能看不到的问题，而这些问题通常是更人为的。

我们将通过一个实际的例子来研究金丝雀的创作过程及其评估，同时避免深入了解统计学。相反，我们专注于整个过程和典型的实际考虑。我们在App Engine上使用一个简单的应用程序来说明部署的各个方面。

### 金丝雀过程的要求

给一个确定的服务执行金丝雀发布需要特定功能:

- 一种将金丝雀变化部署到服务总体子集的方法。[96](#)
- 评估过程，以评估标准变更是“好”还是“坏”。
- 将金丝雀评估整合到发布过程中。

最终，金丝雀过程会在金丝雀以高置信度检测到不良发布候选者，并识别出没有误报的良好发布时证明其价值。

### 我们的示例设置

我们将使用一个简单的前端Web服务应用程序来说明一些简单的概念。该应用程序提供了一个基于HTTP的API，消费者可以使用它来处理各种数据(诸如产品价格之类的简单信息)。该示例应用程序具有一些可调参数，我们可以使用它们来模拟各种生产症状，并由金丝雀过程对其进行评估。例如，我们可以使应用程序为20%的请求返回错误，或者我们可以规定5%的请求至少需要两秒钟。

我们使用部署在Google App Engine上的应用程序说明了金丝雀过程，但是原理适用于任何环境。尽管该示例应用程序确实是精心设计的，但在实际场景中，类似的应用程序与我们的示例共享共同的信号，这些信号可以在金丝雀过程中使用。

我们的示例服务有两个潜在的版本：实时和发布候选。实时版本是当前在生产环境中部署的版本，候选发布版本是新建版本。我们使用这些版本来说明各种推广概念，以及如何实现金丝雀发布以使上线过程更加安全。

## 前滚部署与简单的金丝雀部署相比

首先让我们看一下没有执行金丝雀流程的部署，以便稍后可以将其与金丝雀部署进行比较，以节省错误预算和发生破损时的总体影响。我们的部署过程具有开发环境。一旦感觉到代码在开发环境中可以工作，就可以在生产环境中部署该版本。

部署后不久，我们的监控开始报告高错误率(请参见图16-2，在此我们故意将示例应用程序配置为20%的请求失败

用于模拟示例服务中的缺陷)。就本例而言，假设我们的部署过程没有向我们提供回滚到先前已知的良好配置的选项。修复错误的最佳选择是在生产版本中发现缺陷，修补缺陷并在中断期间部署新版本。这种做法几乎可以肯定会影响用户对该错误的影响。

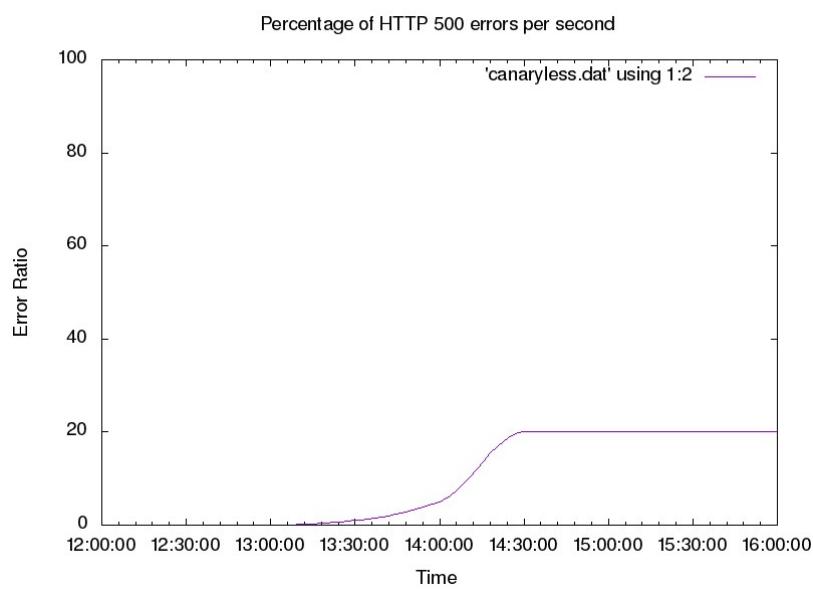
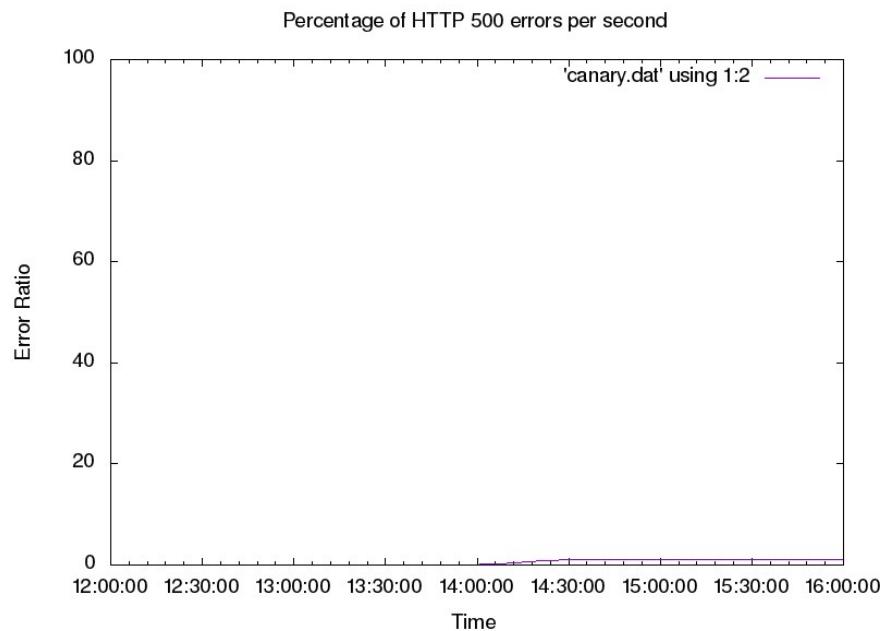


图16-2. 部署后错误率高

为了改进此初始部署过程，我们可以在部署策略中利用金丝雀来减少错误代码推送的影响。我们需要一种方法来创建一小部分可以运行我们的候选发布产品的产品，而不是一次全部部署到生产中。然后，我们可以将流量的一小部分发送到该生产部分(金丝雀)，并将其与其他部分(对照组)进行比较。使用这种方法，我们可以在影响所有产品之前发现候选版本中的缺陷。

我们的App Engine示例中的简单金丝雀部署可以在应用程序的特定标记版本之间[分割流量](#)。您可以使用App Engine或任何其他方法(例如负载平衡器上的后端权重，代理配置或轮询DNS记录)分配流量。

图16-3显示，使用金丝雀时，变更的影响已大大降低；实际上，错误几乎看不见！这就提出了一个有趣的问题：与总体流量趋势相比，金丝雀的评价很难看到和跟踪。

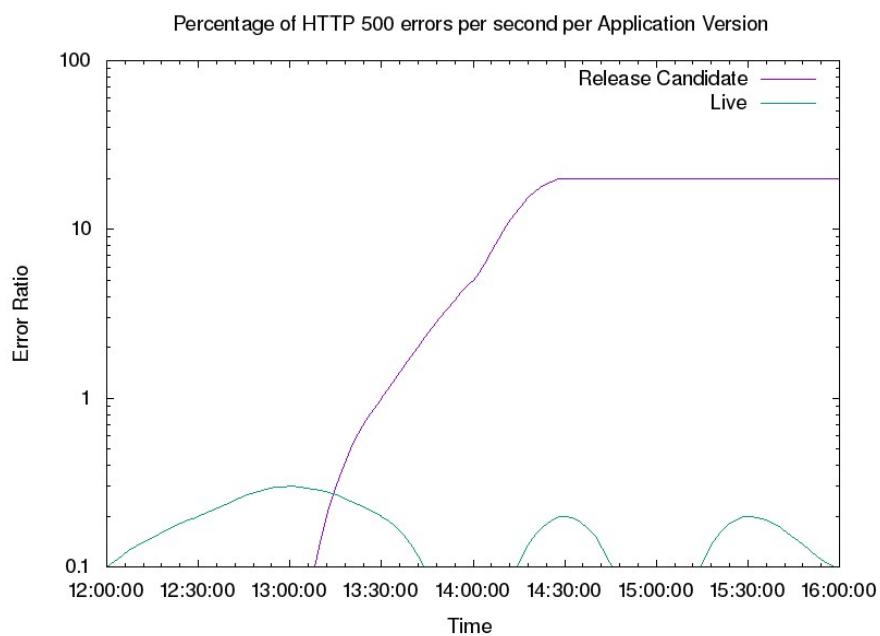


**图16-3.金丝雀部署的错误率；由于金丝雀种群只占生产的一小部分，因此总体错误率降低了**

为了更清晰地了解我们需要以合理的规模进行跟踪的错误，我们可以按App Engine应用程序版本查看关键指标(HTTP响应代码)，如下所示：

如图16-4所示。当我们查看每个版本的细分时，我们可以清楚地看到新版本引入的错误。从图16-4我们还可以观察到实时版本提供的错误很少。

现在，我们可以调整部署，以根据App Engine版本的HTTP错误率自动做出反应。如果金丝雀度量标准的错误率与控制错误率相距太远，则表明金丝雀部署是“错误的”。作为响应，我们应该暂停并回滚部署，或者与人员联系以解决问题。如果错误率相似，我们可以照常进行部署。在图16-4的情况下，我们的金丝雀部署显然很糟糕，我们应该回退它。



**图16-4.HTTP响应代码(按App Engine版本)；候选版本可解决绝大多数错误，而实时版本在稳定状态下会产生少量错误(请注意:图表使用以10为底的对数刻度)**

## 金丝雀实现

既然我们已经看到了相当简单的金丝雀部署实现，那么让我们更深入地研究成功的金丝雀流程需要了解的参数。

### 最大程度地降低SLO和错误预算的风险

第2章讨论了SLO如何反映围绕服务可用性的业务需求。这些要求也适用于金丝雀实现。金丝雀的生产过程仅会冒着我们错误预算的一小部分风险，这受时间和金丝雀种群数量的限制。

全局部署会很快使SLO面临风险。如果我们从琐碎的示例中部署候选人，我们将有可能失败20%的请求。如果我们改用5%的金丝雀种群，则为5%的流量提供20%的错误，导致总错误率为1%(如图16-3所示)。这种策略使我们可以节省错误预算-

对预算的影响与存在缺陷的流量成正比。我们可以假定，朴素的部署和金丝雀部署的检测和回滚大约需要相同的时间，但是当我们将在金丝雀流程集成到我们的部署中时，我们将以更低的成本获得有关新版本的有价值的信息。

这是一个非常简单的模型，假定负载均匀。它还假设我们可以将全部错误预算(超出我们已经包括在当前可用性的有机度量中的预算)花在金丝雀上。在这里，我们仅考虑新版本引入的不可用性，而不是“实际”可用性。我们的模型还假设故障率为100%，因为这是最坏的情况。<sup>97</sup>金丝雀部署中的缺陷很可能不会影响100%的系统使用率。在金丝雀部署期间，我们还允许整体系统可用性低于SLO。<sup>98</sup>

此模型有明显的缺陷，但是这是您可以调整以适应业务需求的坚实起点。<sup>99</sup>我们建议使用满足您的技术和业务目标的最简单模型。根据我们的经验，专注于使模型在技术上尽可能正确通常会导致对模型的过度投资。对于具有高复杂度的服务，过于复杂的模型可能导致不断的模型调整，而没有真正的好处。

### 选择金丝雀种群和持续时间

选择合适的金丝雀持续时间时，您需要考虑开发速度。如果每天发布，则一次只能运行一个金丝雀部署时，您的金丝雀不能持续一周。如果每周部署一次，您将有时间执行相当长的金丝雀。如果连续部署(例如一天20次)，则金丝雀的持续时间必须大大缩短。值得一提的是，虽然我们可以同时运行多个金丝雀部署，但这样做会增加大量精力来跟踪系统状态。当快速推理系统状态很重要时，这在任何非标准情况下都可能会成为问题。如果金丝雀重叠，同时运行金丝雀也会增加信号污染的风险。我们强烈建议一次只运行一个金丝雀部署。

对于基本评估，我们不需要庞大的金丝雀种群即可检测关键的关键条件。<sup>100</sup>但是，具有代表性的金丝雀过程需要跨多个维度的决策：

#### 大小和持续时间

它应该足够大并且持续足够长的时间才能代表整个部署。仅收到少量查询后终止金丝雀部署，对于以功能多样的查询为特征的系统来说，并不是一个有用的信号。处理速率越高，获取代表性样本所需的时间就越少，以确保观察到的行为实际上归因于规范变化，而不仅仅是随机工作。

#### 流量

我们需要在系统上接收足够的流量，以确保它已处理了代表性样本，并且系统有机会对输入做出负面反应。通常，请求越均匀，您需要的流量就越少。

#### 一天中的时间

性能缺陷通常仅在高负载下才会显示，<sup>101</sup>因此，在非高峰时间进行部署可能不会触发与性能相关的缺陷。

#### 评估指标

金丝雀的代表性与我们选择评估的指标紧密相关(我们将在本章稍后讨论)。我们可以快速评估诸如查询成功之类的琐碎指标，但其他指标(例如队列深度)可能需要更多时间或大量金丝雀来提供清晰的信号。

令人沮丧的是，这些要求可能相互矛盾。通过对最坏情况的冷分析和系统过去的实际记录，金丝雀是一种平衡的行为。从过去的金丝雀收集指标后，您可以根据典型的金丝雀评估失败率而不是假设的最坏情况来选择金丝雀参数。

### 选择和评估指标

到目前为止，我们一直在研究成功率，这是金丝雀评估的一个非常明显的指标。但是从直觉上讲，我们知道对于一个有意义的金丝雀过程，这个单一的指标还不够。如果我们以10倍的延迟来处理所有请求，或者在使用10倍的内存时这样做，那么我们可能会遇到问题。并非所有指标都适合评估金丝雀。指标的哪些属性最适合评估金丝雀的好坏？

### **指标应表明问题**

首先，该指标必须能够指示服务中的问题。这很棘手，因为构成“问题”的原因并不总是客观的。我们可能会认为失败的用户请求是有问题的。<sup>102</sup>但是，如果一个请求花费的时间增加10%，或者系统需要的内存增加10%，该怎么办？我们通常建议使用SLI作为开始考虑金丝雀指标的地方。良好的SLI倾向于对服务健康有强烈的归因。如果已经在衡量SLI以推动SLO遵从性，我们可以重复使用这项工作。

几乎所有的度量标准都可能会出现问题，但在您的金丝雀过程中添加过多的度量标准也要付出代价。我们需要为每个指标正确定义可接受行为的概念。如果可接受的行为的概念过于严格，我们将得到很多误报；也就是说，我们认为金丝雀部署是不好的，即使事实并非如此。相反，如果可接受行为的定义过于宽松，我们将更有可能让不良的金丝雀部署无法被发现。正确选择可接受的行为可能是一个昂贵的过程-这既耗时又需要分析。但是，如果做得不好，您的结果可能会完全误导您。另外，随着服务，其功能集及其行为的发展，您需要定期重新评估期望。

我们应该根据我们对度量指标表示系统中实际用户可感知问题的程度的意见进行排序。选择用于金丝雀评估的头几个指标(也许不超过十二个)。太多的指标会带来递减的收益，并且在某些时候，收益被维护成本所抵消，或者如果维护成本不高，则会对发布过程中的信任产生负面影响。

为了使该指南更加切实，让我们看一下示例服务。它具有许多我们可以评估的指标：CPU使用率，内存占用量，HTTP返回码(200s，300s等)，响应延迟，正确性等。在这种情况下，我们最好的指标可能是HTTP返回码和响应延迟，因为它们的降级最接近于影响用户的实际问题。在这种情况下，CPU使用率的度量标准没有用：资源使用率的增加并不一定会影响服务，并且可能导致金丝雀跃的过程变得脆弱或嘈杂。这可能导致金丝雀过程被操作员禁用或忽略，这可能会破坏首先具有金丝雀过程的意义。对于前端服务，我们直观地知道，速度变慢或响应失败通常是可靠的信号，表明服务存在问题。

HTTP返回代码包含有趣的棘手情况，例如代码404，它告诉我们找不到资源。发生这种情况的原因可能是用户获得了错误的URL(想象一个损坏的URL在一个受欢迎的讨论板上共享)，或者因为服务器错误地停止了提供资源。通常，我们可以通过从金丝雀评估中排除400级别的代码并添加黑盒监控来测试特定URL的存在来解决此类问题。然后，我们可以将黑匣子数据作为金丝雀分析的一部分，以帮助将金丝雀过程与奇怪的用户行为隔离开。

### **指标应具有代表性且可归属的**

所观察到的度量标准变化的根源应该明确归因于我们正在讨论的变化，并且不应受到外部因素的影响。

在大规模(例如，许多服务器或许多容器)中，我们可能会有异常值，即超额订购的计算机，运行具有不同性能特征的不同内核的计算机或网络过载的计算机。金丝雀种群与控制种群之间的差异与我们部署的变更的功能以及部署的两种基础设施之间的差异一样大。

管理金丝雀是许多力量之间的一种平衡行为。增加金丝雀种群的大小是减少此问题影响的一种方法(如前所述)。当达到我们认为适合系统的金丝雀种群的合理数量时，我们需要考虑我们选择的指标是否可以显示高方差。

我们还应该意识到金丝雀和控制环境之间共享的故障域；不好的金丝雀可能会对控制产生负面影响，而系统中的不良行为可能导致我们错误地评估了金丝雀。同样，请确保指标完全隔离。考虑一个同时运行我们的应用程序和其他进程的系统。整个系统中CPU使用率的急剧增加将导致衡量指标不佳，因为系统中的其他进程(数据库负载，日志轮换等)可能会导致这种增加。更好的指标是进程为请求提供服务时所花费的CPU时间。甚至更好的指标是在服务进程实际安排在CPU上的时间范围内，服务请求所花费的CPU时间。虽然与我们的流程并置的大量超额订购的机器显然是一个问题(监控应该抓住它！)，但这不是我们正在进行的变更引起的，因此不应将其标记为金丝雀部署失败。

金丝雀也需要归因；也就是说，您还应该能够将金丝雀指标与SLI绑定在一起。如果一个指标可以在不影响服务的情况下发生巨大变化，那么就不可能制定一个好的金丝雀指标。

### 评估前后有风险

金丝雀之前/之后的过程是归因问题的延伸。在此过程中，旧系统将被新系统完全替换，您的金丝雀评估会比较在设定的时间段内更改前后的系统行为。人们可能将此过程称为“时空中的金丝雀部署”，您可以通过分段时间来选择A/B组，而不是通过机器，Cookie或其他方式来对种群进行分段。由于时间是观察到的指标变化的最大来源之一，因此很难通过评估前后来评估性能下降。

虽然金丝雀的部署可能导致了性能下降，但性能下降也很可能在控制系统中发生过。如果我们尝试在更长的时间内运行金丝雀部署，这种情况将变得更加棘手。例如，如果我们在星期一进行发布，则可能会将工作日的行为与周末的行为进行比较，从而产生大量噪音。在此示例中，用户在周末使用服务的方式可能非常不同，从而在金丝雀过程中引入了噪音。

之前/之后过程本身会引入一个问题，即大的错误峰值(由评估之前/之后引入)是否比小但可能更长的错误率(由小金丝雀引入)更好。如果新版本完全损坏，我们可以多快检测和还原？金丝雀之前/之后可能会更快地发现问题，但是恢复的总时间可能仍然相当长，类似于较小的金丝雀。在此期间，用户会受苦。

### 使用渐进金丝雀以更好地选择指标

不符合我们理想属性的指标可能仍会带来巨大的价值。我们可以通过使用更细微的金丝雀过程来引入这些指标。

我们可以使用包含多个阶段的金丝雀来反映我们对指标进行推理的能力，而不是简单地评估单个金丝雀阶段。在第一阶段

### 选择和评估指标

我们对此版本的行为没有信心或知识。因此，我们希望使用一个较小的阶段，以最大程度地减少负面影响。在一个很小的金丝雀中，我们更喜欢可以最清楚地表明问题的指标-应用程序崩溃，请求失败等。一旦这一阶段成功通过，下一阶段的金丝雀种群将增加，以增加我们对变化影响的分析的信心。

### 依赖关系和隔离

被测系统将无法在完全真空中运行。出于实际原因，金丝雀种群和对照组可能共享后端，前端，网络，数据存储和其他基础结构。与客户之间甚至可能存在极其不明显的交互。例如，假设一个客户端发送了两个连续的请求。第一请求可以由金丝雀部署处理。金丝雀的响应可能会更改第二个请求的内容，而第二个请求的内容可能会落在对照组上，从而更改对照组的行为。

不完善的隔离会带来多种后果。最重要的是，我们需要意识到，如果金丝雀的发布过程提供的结果表明我们应该停止生产变更并调查情况，那么金丝雀的部署不一定存在错误。总的来说，这个事实确实适用于金丝雀，但实际上，隔离问题经常强制执行。

此外，不完善的隔离意味着金丝雀部署的不良行为也会对对照组产生负面影响。金丝雀发布是A/B的比较，A和B可能会同时更改。这可能会导致金丝雀评估混乱。同样重要的是，也要使用绝对测量(例如定义的SLO)来确保系统正常运行。

### 非交互式系统中的金丝雀发布

本章重点讨论了交互式请求/响应系统，该系统在许多方面都是最简单且讨论最多的系统设计。其他系统(例如异步处理管道)也同样重要，但是具有不同的金丝雀发布考虑因素，我们将简要列举一下。有关与数据处理管道有关的金丝雀跃变的更多信息，请参见第13章。

首先，金丝雀的持续时间和部署本质上取决于工作单元处理的持续时间。在涉及交互式系统时，我们忽略了这个因素，假设工作单元的处理时间不超过几秒钟，这比金丝雀的持续时间短。非交互系统中的工作单元处理(例如渲染管道或视频编码)可能需要更长的时间。因此，请确保金丝雀的最小持续时间跨过单个工作单元的持续时间。

对于非交互式系统，隔离可能变得更加复杂。许多管道系统只有一个工作分配器和一组带有应用程序代码的worker。在多阶段流水线中，工作单元由worker处理，然后返回到池中以供同一worker或另一个worker执行下一阶段的处理。对于金丝雀分析，确保始终将处理特定工作单元的worker从同一个worker池(金丝雀池或对照组池)中拉出很有帮助。否则，信号将变得越来越混杂(有关解开信号的更多信息，请参见第349页的“监控数据的要求”)。

最后，指标选择可能更加复杂。我们可能对处理工作单元的端到端时间(类似于交互式系统中的延迟)以及处理本身的质量(当然，这完全是特定于应用程序)感兴趣。

考虑到这些警告，金丝雀的一般概念仍然可行，并且适用相同的高级原则。

### 监控数据要求

在进行金丝雀评估时，您必须能够将金丝雀信号与对照组信号进行比较。通常，在构建监控系统时需要格外小心-有效的比较是直接的，并且会产生有意义的结果。

考虑我们前面的示例，该示例将金丝雀部署到5%的人口，错误率达到20%。由于监控可能会从整个系统角度出发，因此它将检测到的总体错误率仅为1%。根据系统的不同，此信号可能与其他错误源没有区别(见图16-3)。

如果我们通过对请求进行总体处理来分解指标(金丝雀与对照组)，我们可以观察到单独的指标(见图16-4)。我们可以清楚地看到对照组与金丝雀的错误率，这是完整部署将带来什么的鲜明说明。在这里，我们看到监控整个服务的充分理由不足以分析我们的金丝雀。收集监控数据时，重要的是能够执行细粒度的细分，使您能够区分金丝雀和对照种群的指标。

收集指标的另一个挑战是金丝雀的部署受设计的限制。当指标在特定时期内汇总时，这可能会导致问题。考虑度量每小时错误。我们可以通过汇总过去一个小时的请求来计算该指标。如果使用此指标评估我们的金丝雀，则可能会遇到问题，如以下时间轴所述：

### 监控数据要求

1. 不相关的事件导致发生一些错误。
2. 金丝雀部署到总量的5%；金丝雀的时长是30分钟。
3. 金丝雀系统开始监控每小时错误度量，以查看部署的好坏。
4. 由于每小时错误度量标准与对照组的每小时错误显着不同，因此将部署检测为不良。

这种情况是使用每小时计算一次的指标评估仅30分钟长的部署的结果。结果，金丝雀过程提供了非常混乱的信号。使用指标评估金丝雀的成功时，请确保指标的间隔等于或小于金丝雀的持续时间。

### 相关概念

通常，我们与客户的对话涉及在生产中使用蓝/绿部署，人工负载生成和/或流量准备。这些概念类似于金丝雀，所以虽然它们不是严格的金丝雀过程，但可以按原样使用。

#### 蓝/绿部署

蓝/绿部署维护系统的两个实例：一个正在为流量提供服务（绿色），另一个准备为流量提供服务（蓝色）。在蓝色环境中部署新版本后，可以将流量移至该版本。切换不需要停机，回滚是对路由器更改的简单逆转。缺点是此设置使用的资源是更“传统”部署的两倍。在此设置中，您将有效地执行金丝雀之前/之后（之前讨论过）。

通过同时（而不是独立）利用蓝色和绿色部署，您可以或多或少地使用蓝/绿部署作为普通金丝雀。在此策略中，您可以将金丝雀部署到蓝色（备用）实例，并在绿色和蓝色环境之间缓慢分配流量。您的评估以及将蓝色环境与绿色环境进行比较的指标均应与流量控制联系在一起。此设置类似于A/B金丝雀，其中绿色环境是对照组，蓝色环境是金丝雀部署，金丝雀数量由发送给每个人的流量控制。

#### 人工负载生成

与其将实时用户流量暴露给金丝雀部署，还不如在安全的层面诱发错误，并使用人工负载生成。通常，您可以在多个部署阶段（QA，预生产，甚至在生产中）运行负载测试。尽管根据我们的定义，这些活动不算是金丝雀，但它们仍然是发现缺陷的可行方法。

使用合成负载进行测试可以最大程度地提高代码覆盖率，但是不能提供良好的状态覆盖率。在可变系统（具有缓存，cookie，请求关联性等的系统）中人为地模拟负载可能尤其困难。人工负载也可能无法准确地模拟真实系统中发生的自然流量变化。某些回归可能仅在未包含在人工负载中的事件期间才显示出来，从而导致覆盖范围出现缺口。

人工负载在可变系统中也无法正常工作。例如，尝试在计费系统上产生人为负载可能是非常危险的：系统可能开始向信用卡提供商发送标注，然后信用卡提供商将开始主动向客户收费。虽然我们可以避免测试危险的代码路径，但是在这些路径上缺少测试会减少我们的测试范围。

### 流量开球

如果人工负载不具有代表性，我们可以复制流量并将其发送到生产系统和测试环境中的系统。此技术称为“开球”。在生产系统为实际流量提供服务并向用户传递响应的同时，金丝雀部署为副本提供服务并丢弃响应。您甚至可以将金丝雀的响应与实际响应进行比较，然后进行进一步的分析。

这种策略可以提供有代表性的流量，但是设置起来通常比更简单的金丝雀过程更为复杂。流量开球也无法充分识别状态系统中的风险；流量副本可能会在看似独立的部署之间引入意想不到的影响。例如，如果金丝雀部署和生产系统共享高速缓存，则人为增加的高速缓存命中率会使金丝雀指标的性能度量无效。

## 结论

您可以使用多种工具和方法来自动化发布，并在管道中引入金丝雀发布。没有哪一种测试方法是万能的灵丹妙药，因此应根据系统的要求和行为来告知测试策略。金丝雀发布是一种补充测试的简单，可靠且易于集成的方式。当您尽早发现系统缺陷时，对用户的影响最小。金丝雀发布还可以为频繁发布提供信心并提高开发速度。正如测试方法必须与系统需求和设计一起发展一样，必须金丝雀。

95. 本章的作者是Jez Humble和David Farley的书的爱好者连续交付: 通过构建，测试和部署自动化发行可靠的软件(波士顿: 皮尔逊(Pearson)，2011年)。 ↵

96. 您在金丝雀上运行的总服务负载的比例应与金丝雀种群的大小成比例。  
↵

97. 至少就可用性而言。显然，该分析未涵盖事件的影响，例如数据泄漏。  
↵

98. 对于足够小的金丝雀，其中服务部分等于实际可用性和SLO之间的差，我们可以充满信心地直达金丝雀。这是我们对错误预算的使用。 ↵

99. 正如英国统计学家乔治·博克斯(George Box)所说: “本质上，所有模型都是错误的，但有些模型是有用的。”George EP Box和Norman R. Draper，“经验模型构建和响应面”(纽约: John Wiley and Sons，1987年)。  
↵

100. 请参阅<http://bit.ly/2LgorFz>上有关实际中断的讨论。 ↵

101. 例如，考虑资源争用问题，例如数据库写冲突或在多线程应用程序中锁定。 ↵

102. 失败的用户请求不一定有问题。用户请求可能会失败，因为用户请求了一些不合理的内容，例如访问不存在的URL。我们需要纪律严明，以区分此类错误和系统中的问题。 ↵

## PART III 流程

SRE不仅仅是技术实践的集合-它是一种需要一致且逻辑的流程的文化。本节介绍了SRE中应用的流程的一些重要元素:管理运维和开发工作，从过载中恢复，构建和管理与开发合作伙伴和客户的关系以及实施实际的变更管理。

如果您是SRE的负责人，那么本部分将为您提供指导，以帮助您创建团队成功实施第二部分所涵盖的实践所需的空间和环境。

## 第17章

### 识别并从过载中恢复

由*Maria-Hendrike Peetz , Luis Quesada Torres 和 Marilia Melo*与*Diane Bates*撰写

当SRE团队运行平稳时，团队成员应该感觉自己可以轻松地处理所有工作。他们应该能够进行故障单工作，并且还有时间从事长期项目，这使得将来管理服务变得更加容易。

但是有时候情况会影响团队的工作目标。团队成员长期生病请假或转入新团队。组织为SRE交付了新的生产范围的程序。对服务或更大系统的更改带来了新的技术挑战。随着工作量的增加，团队成员开始工作更长的时间来处理故障单和呼叫，并花费更少的时间进行工程工作。整个团队在努力工作时开始感到压力和沮丧，但感觉不到自己正在取得进步。反过来，压力会使人们犯更多错误，从而影响可靠性，并最终影响最终用户。简而言之，团队失去了调节日常工作和有效管理服务的能力。

此时，团队需要找到摆脱这种超载状态的方法。他们需要重新平衡工作量，以便团队成员可以专注于基本的工程工作。

"运维负载"是一个术语，用于描述使系统和服务保持最佳性能运行的持续维护任务。有三种不同类型的操作负载:[呼叫](#)，[故障单](#)和[正在进行的运维职责](#)。呼叫通常需要立即关注，与紧急问题相关的故障单可能会有紧迫的期限。呼叫和紧急故障单都会中断SRE从事支持团队运维职责的工程项目。因此，我们将它们称为"中断"。

[Site Reliability Engineering]的[第29章](#)讨论了管理团队将复杂系统维持在功能状态时自然产生的中断的技术。

当操作负载超过团队的管理能力时，团队最终会处于运维过载(也称为工作过载)状态。当团队无法在关键优先事项上取得进展时，团队就处于运维超负荷状态，因为紧急问题不断抢占项目工作。除了降低团队的优先级和改善服务质量之外，过载还会增加工程师犯错的机会。[103](#)

团队之间的操作过载阈值可能会有所不同。Google SRE团队将运维工作的时间限制为工程师的50%。一个成功的SRE团队必须有信心，从长远来看，他们将能够完成所需的工程项目，以减轻他们管理的服务的运维负担。

本章介绍了Google的团队如何从以运维过载为特征的困难局势发展为管理良好的工作负荷。两个案例研究显示了运维超负荷对团队健康的有害影响，以及团队如何更改其日常任务，以便他们可以专注于长期影响重大的项目。在案例研究1中，当一个正收缩团队的剩余成员无法跟上工作量时，就会导致过载。在案例研究2中，一个团队遭受感知超载一种与运维超载具有相同效果的状态，但其开始时对实际工作负载的误解。

这些案例研究着重介绍了适用于两个Google SRE团队的特定操作，但第366页的“缓解过载的策略”部分提供了适用于任何公司或组织的识别和缓解过载的实践。因此，本章对过载团队的经理或任何关心过载的SRE团队都应该是有用的。

## 从负载到过载

无论其起源如何，过载都是一种职业压力，会削弱生产力。如果任其发展，[可能会导致严重疾病](#)。对于SRE团队，运维负荷通常是认知上困难的任务(例如调试内存泄漏或分段错误)和许多小任务的组合需要频繁的上下文切换(通过配额请求，启动二进制部署等)。

当团队没有足够的时间来处理所有这些任务时，通常会发生工作超负荷的情况，这是客观的现实，即分配给团队的任务数量无法在给定的期限内完成每个任务。感觉到的超负荷更为主观，并且在团队中的每个人感到他们的工作过多时发生。通常在短时间内发生几项组织或工作变更时会发生这种情况，但是团队几乎没有机会与领导层就变更进行沟通。

不清楚在值班时会出现什么问题，或者工作量会是多少。一方面，一张关于磁盘空间用尽的看似无辜的故障单可能会导致对重复发生的垃圾收集作业的深入研究。另一方面，20次呼叫以上的值班风暴可能是监控不佳的情况。当难以估计或预测工作量时，您很容易成为认知偏见的受害者，并错误地估计工作量-例如，您可能认为故障单队列太大而无法在轮班期间完成。即使您可以快速完成所有故障单并且实际工作量很低，但是当您初次查看故障单队列时，您还是会感到超负荷。这种感觉超负荷[104](#)本身具有影响您对工作的态度和态度的心理因素。如果您不以有太多工作而先入为主，那么您很可能会潜入并开始尝试通过故障单队列。也许您整天工作并没有完成工作量(因此面临工作超负荷)，但是与开始时不知所措相比，您取得了更大的进步。

累积许多中断会导致工作过载，但这不是必须的。但是，当频繁的中断与外部压力因素结合使用时，大的工作量(甚至是小的工作量)很容易变成感知的过载。这种压力可能是由于担心其他团队成员感到失望，工作缺乏安全感，与工作有关或与个人有冲突，疾病或与健康有关的问题(如缺乏睡眠或运动)而引起的。

如果您的工作没有得到适当的优先排序，那么每个任务似乎都同样紧迫，从而导致实际和可感知的过载。在实际超负荷的情况下，工单和警报的紧急性可能会导致团队成员继续工作，直到他们解决问题为止，即使这样做意味着连续的长时间工作。当团队面临明显的超负荷时，重新安排优先级可以帮助减少紧急工作量，为他们创造空间来通过项目工作解决超负荷的根源。

在分析您的特定情况时，您不必假定工作负载本身需要更改。相反，我们建议您首先量化您的团队所面临的工作，以及它随着时间的变化(或没有变化)。例如，您可以通过团队处理的故障单和呼叫数来衡量工作量。如果您的工作量实际上并没有随着时间的推移而改变，那么团队可能会因为他们认为工作繁重而感到超负荷。为了更全面地了解团队当前的工作量，您可以通过要求每个成员列出他们面临的所有工作任务来收集一次性快照。然后查看团队面临的心理压力因素，例如组织变更或重新安排优先级。完成研究后，您将有一个稳定的基础来做出有关更改工作负载的决策。

第366页的"减轻过载的策略"更多地讨论了如何识别实际过载和感知过载。首先，我们对团队进行了两个案例研究，这些团队认识到他们处于超负荷状态，并采取了缓解措施。

## 案例研究1:半个团队离开时的工作超负荷

### 背景

Google的内部存储SRE团队之一负责多种服务的后端，包括Gmail，Google云端硬盘和Google网上论坛以及许多其他内部或面向用户的服务。我们在2016年中期经历了一次危机，当时三分之二的团队(包括最高级的工程师(经理))出于相对完全不相关的原因，在相对较短的时间内转移到其他机会。显然，此事件导致了巨大的工作负载管理问题:可用于覆盖相同运营和项目工作的SRE更少，从而导致过载。我们的工作也遇到了瓶颈，因为每个团队成员的专业知识都被隔离到不同的生产领域。从长远来看，增加新的团队成员和三个实习生可以改善我们的工作量，而增加这些工程师则需要花费大量时间和精力。

### 问题陈述

前述因素大大降低了团队生产力。我们开始在项目工作上落后，与我们管理的许多服务相关的故障单开始堆积起来。我们没有足够的带宽来解决此积压工作，因为我们的所有工作都被优先级较高的任务占用。不久之后，我们将无法完成我们需要做的所有关键和紧急工作。同时，我们的团队计划很快接受更多的高优先级工作。

如果我们不做一些工作，那么我们不小心开始放弃重要的工作只是时间问题。但是，一旦我们开始卸货，就会遇到一些心理障碍:

- 放弃正在进行的任何工作，就好像我们浪费了我们的精力。大部分待办事项似乎对我们来说都是至关重要的，也值得我们付出努力，因此无限期取消或延迟项目是不合适的。我们没有意识到自己陷入了[沉没成本谬误](#)。
- 致力于自动化流程或解决工作负载的根本原因并不像立即处理高优先级中断那样重要。当这项工作被添加到一个已经很大的堆的顶部时，所有的工作都感觉不堪重负。

### 我们决定要做什么

我们将团队聚集在一个房间里，并列出了团队的所有职责，包括项目积压，运维工作和工单。然后，我们对每个列表项进行了分类。查看我们的每一项工作任务(即使列表几乎不适合白板)也有助于我们确定并重新定义我们的实际优先事项。然后，我们能够找到最小化，移交或消除优先级较低的工作项的方法。

### 实施

我们确定了不费吹灰之力的自动化技术，尽管它并不重要，但一旦部署，它将大大减少操作负荷。

我们还确定了可以记录的，可以实现自助服务的常见问题。编写客户所需的过程并不需要很长时间，并且从我们的队列中删除了一些重复的工作。

我们尽可能地关闭了许多积压的工单。这些故障单大多数已过时，多余或不像他们所声称的那么紧急。一些故障单是不可操作的监控工件，因此我们修复了相关监控。在某些情况下，我们正在积极解决非关键问题。我们将这些问题放在一边，以便处理更紧急的故障单，但首先记录了我们的进度，因此在再次进行处理之前，我们不会失去上下文。

如有疑问，我们放弃了一项任务，但将其标记为第二阶段的分类。一旦我们的餐盘(几乎)是空的，我们重新访问了这个暂定清单，以决定要恢复的任务。事实证明，这些任务几乎都没有影响力或重要到足以恢复。

在两天的时间内-密集分类的一天，外加记录过程和实现自动化的一天-我们的小得多的团队处理了几个月中断的积压。然后，我们可以处理剩余的少量中断，这些中断与生产中的活动问题有关。

### 经验教训

我们的团队了解到，鉴别并确定超载是解决它的第一步。我们需要让每个人都在一个房间里并重新评估积压，然后才能帮助我们的团队恢复健康。

为了避免新的中断堆积，我们每两周开始对中断进行一次分类。我们的技术主管会定期检查任务队列，并评估团队是否有超负荷工作的风险。我们决定每个团队成员应拥有10张或更少张open的故障单，以避免超载。如果团队负责人发现团队成员拥有超过10张故障单，他们可以执行以下一项或多项操作：

- 提醒团队关闭过期故障单。
- 与超负荷的团队成员同步并从他们那里卸载故障单。
- 提示各个团队成员解决他们的故障单队列。
- 组织整个团队的一日集中故障单修复。
- 分配工作以修复工单来源，或分配工作以减少将来的工单。

## 案例研究2:组织和工作负载更改后的感知超载

### 背景

在此案例研究中，Google SRE团队被划分为两个地点，每个站点都有六到七名值班工程师(有关团队规模的更多讨论，请参见以下内容的[站点可靠性工程第11章](#))。在悉尼团队运作健康的同时，苏黎世团队却超负荷运转。

在苏黎世团队超负荷工作之前，我们很稳定并且很满足。我们管理的服务数量相对稳定，并且每种服务种类繁多且维护程度很高。虽然我们支持的服务的SLO与它们的外部依赖项的SLO不匹配，但是这种不匹配并没有引起任何问题。我们正在进行许多项目，以改善我们管理的服务(例如，改善负载平衡)。

同时触发因素使苏黎世团队陷入超负荷状态:我们开始采用噪音较大且与Google的通用基础架构集成程度较低的新服务，技术负责人经理和另一位团队成员离开了我们的团队，使两个人短缺。额外的工作量和知识流失的结合引发了更多的问题:

- 对新服务的监控调整和与迁移相关的监控导致每班更多呼叫。这种积累是渐进的，因此我们没有注意到上升的发生。
- SRE对新服务感到相对无助。我们对它们的了解不足，无法做出适当的反应，因此经常需要向开发团队提出问题。尽管超负荷可能需要将服务交还给开发人员，但我们的团队从未将服务交还给开发人员，因此我们并不认为这是可行的选择。
- 一个更小的五人的值班轮调减少了我们通常在运维工作上花费的时间。
- 新的故障单警报解决了最近的团队变更之前存在的问题。尽管过去我们只是简单地忽略了这些问题，但现在我们需要将被忽略的电子邮件警报移至故障单。项目计划没有考虑到这种新的技术债务来源。
- 新的故障单SLO要求我们在三天内处理故障单，这意味着应召集人必须更早解决其在轮班时创建的故障单。[105](#) SLO旨在减少添加到我们的故障单中的故障单数量(几乎被忽略)积压，但产生的副作用可能更糟。现在，SRE感到他们在轮班后无法获得所需的其余部分，因为他们必须立即处理后续工作。这些故障单具有更高的优先级，这也意味着SRE没有足够的时间进行其他运维工作。

在这段时间里，我们的团队被分配了一位新经理，他还管理了另外两个团队。新经理不是值班轮换的一部分，因此没有直接感觉到团队成员正在经历压力。当团队向经理解释情况时，什么都没有改变。团队成员感到自己没有被听到，这使他们感到与管理团队相距遥远。

罚单的超载持续了几个月，使团队成员脾气暴躁，直到整个团队中散发出一连串的不满。

### 问题陈述

在失去了两个人并接受了更多不同的工作之后，我们的团队感到不堪重负。当我们试图将这种感觉传达给我们的直接经理时，经理不同意。随着长时间的工作，疲惫开始了。生产力正在下降，任务开始累积的速度超过了团队所能解决的速度。现在，感觉到的过载已变成客观过载，使情况变得更糟。

过载造成的情绪压力正在降低士气，并导致一些团队成员精疲力尽。当个人处理过度劳累造成的身体影响(疾病和生产率降低)时，团队中的其他人必须承担更多的工作。每周团队会议分配的工作尚未完成。

然后我们开始假设我们不能依靠别人来完成他们的工作，这削弱了团队内部的信任感和可靠性。结果，我们对人际风险承担(这是影响心理安全的重要因素)感到不安全(请参阅[Site Reliability Engineering]中的[第11章](#))。团队成员没有感到其他团队成员的接受和尊重，因此他们没有彼此自由协作。由于团队的心理安全性下降，协作停止了，从而减慢了信息共享的速度，并进一步导致了效率低下。

团队调查还显示出心理安全感的丧失-团队成员说，他们不觉得自己属于团队。他们不再关心自己的职业发展，团队的晋升率降至历史最低水平。

当高层管理人员为我们分配了新的强制性全公司项目时，我们终于达到了一个突破点。在这一点上，我们重新与管理层进行了有关以新的活力进行过载的对话。一系列讨论表明，我们的不愉快情况不仅仅是工作量过多的结果-我们对团队安全的看

法导致我们停止相互信任和合作。

### 我们决定要做什么

高层管理人员为我们的团队分配了一位新的经理，这三个团队之间没有人共享。新经理使用[参与式管理风格](#)改善了团队的心理安全性，以便我们可以再次进行协作。这种方法使团队成员能够积极参与解决团队问题。整个团队，包括我们的直接经理，都进行了一系列简单的团队建设练习，以提高我们团队的效率(其中有些简单到一起喝茶)。<sup>106</sup>因此，我们能够起草一套目标：

#### 短期

减轻压力，改善心理安全，建立健康的工作氛围。

#### 中期

通过培训建立团队成员的自信心。查找导致过载的问题的根本原因。

#### 长期

解决导致级联的持续性问题。

为了设定这些目标，我们必须首先在团队中实现某种基准的心理安全。随着士气的提高，我们开始分享知识，并在彼此的思想基础上寻求解决方案，以控制工作量。

### 实施

#### 短期行动

长期压力，无论是由于劳累过度还是团队安全感引起的，都会降低生产率并影响人们的健康。因此，我们最重要的短期措施是减轻压力，改善信任和心理安全。一旦减轻了一些压力，团队成员就可以更加清晰地思考并参与推动整个团队前进。在确定过载的一个月内，我们实施了以下措施：

- 启动了半常规圆桌会议来讨论问题。该团队释放了挫败感，并集体讨论了可能导致超载的原因。
- 找到了一个更好的衡量负载的指标。我们决定改进我们的原始呼叫数指标。我们自动将故障单分配给了呼叫者，即使转移结束后，呼叫者仍要负责这些故障单。我们的新指标衡量了呼叫者在轮班后解决故障单所需的时间。
- 审核并删除了垃圾邮件警报。我们查看了警报，并删除了那些不代表用户面临问题的警报。
- 慷慨的安静的警报。该团队故意不为每个警报寻找来源，而是致力于减轻我们已经知道的被连续寻呼和发出故障单问题的压力。我们使用以下策略：
  - 浮出水面的警报将保持静音，直到它们被修复。
  - 警报只能在有限的时间段(通常是一天，有时长达一周)内保持静音。否则，它们可能掩盖中断。
  - 在几分钟内无法修复的警报已分配给跟踪故障单。
- 增加了一个专门负责单个团队的直接经理。使新经理成为受人尊敬的团队成员，重新建立了对管理层的信任。新经理可以将更多的时间放在单个团队及其成员身上，而不是管理三个团队。

- **重新平衡团队。**通过添加对团队或组织没有先入为主的技术经验丰富的SRE，我们引入了新的视角和值班人员。寻找合适的人绝非易事，但值得付出努力。
- **举办团队活动，例如午餐和棋盘游戏。**谈论与工作无关的话题并一起大笑可以缓解团队的紧张感并提高心理安全性。

### 中期行动

仅短期解决方案将无法维持健康的氛围-例如，我们的短期策略之一是在没有真正解决问题的情况下使警报保持沉默。在三个月内，我们还采取了以下行动:

- **将运维工作尽可能地限制在值班时间**(请参阅Site Reliability Engineering中的[第29章](#))，以便团队可以专注于永久性修复和项目工作。
- **将一项服务的责任交还给其开发团队。**
- **彼此培训(和新团队成员)**。尽管培训需要投入时间和精力，但传播知识意味着所有团队成员(以及将来的员工)可以在将来更快地解决问题并解决问题。培训同事提高了我们的信心，因为我们意识到我们实际上对服务非常了解。随着知识的积累，团队成员开始寻找管理服务，提高可靠性和减少过载的新方法。
- **从远程团队引进了SRE，以为我们的一些值班人员配备人员并参加培训。**他们注意到了团队的压力，并提供了一些有价值的新视角。
- **回补了团队中的两个公开职位。**
- **随着沉默期满，处理每个警报。**我们讨论了重复的呼叫，这些呼叫在每周的生产会议上没有采取任何行动，导致我们调整警报和/或解决潜在的问题。尽管这些是重要的(也是显而易见的)操作，但是只有在警报静音且不会产生持续的噪音后，我们才有空间进行分析并采取措施。
- **有组织的倾听活动。**管理层(包括跳过级经理和团队负责人)有意识地倾听团队的痛点并找到团队驱动的解决方案。
- **增加的视角。**希望不是一种策略，但它无疑有助于团队士气。有了新成员加入呼叫轮换的承诺，转移到更明确的优先级并结束了产生噪声的项目，团队的情绪得到了改善。

### 长期行动

为了帮助维持新发现的稳定性，我们目前正在使SLO与它们的服务后端的SLO保持一致，并致力于使服务更加统一。一致性有双重好处:减轻SRE的认知负担，并使编写可跨服务使用的自动化更加容易。我们还将审查已经存在很长时间的服务，并将其更新为当前的生产标准。例如，某些服务在负载下运行不佳，而这些负载在过去几年中显着增加。某些服务需要根据其后端服务策略的更改进行更新。其他服务几年来一直没有更新。

### 效果

在我们第一次集思广益会议之后的几个月，结果开始浮出水面:呼叫班次变得更加安静，我们的团队设法以小组的形式快速有效地协作处理了一个困难事件。稍后，新的团队成员到达了。当我们在圆桌会议上讨论心理安全性时，新成员表示他们无法想象团队曾经遇到过此类问题。实际上，他们将我们的团队视为温暖，安全的工作场所。在最初的升级大约一年后，最初的超负荷仍然有少量存在，匿名调查显示，团队成员现在认为团队是有效且安全的。

## 经验教训

更换工作场所可能会对团队中的人员产生心理影响-毕竟，您的队友不是机器。您需要注意团队的压力水平，以便人们开始互相信任，可以一起工作；否则，团队可能会进入过载的恶性循环，从而导致压力，从而阻止您应对过载。

感知上的超负荷实际上是超负荷，对团队的影响与其他因素引起的工作超负荷一样大。在我们的案例中，我们在悉尼的姐妹团队没有遇到相同的问题，与往年相比，我们提供的呼叫数量实际上没有太大变化。取而代之的是，失去了两名团队成员，增加了认知负担，增加了工单噪音以及新的为期三天的工单SLO导致他们感到超载。最后，目标过载和感知过载之间的区别并不重要：少数团队成员的感知过载会很快导致整个团队的过载。

## 缓解过载的策略

有时，外部视角可以很容易地确定团队何时过载。同样，很容易就回顾过去应该采取的行动发表评论。但是，当您经历过载时，如何识别过载呢？当您陷入超负荷状态时，通往健康，友好，快乐的工作氛围的途径很难想象。本节介绍识别和减轻团队负担的实践。

### 识别过载症状

如果您知道超载的症状，很容易确定超载的团队：

#### 降低团队士气

过载可能表现为骚动和抱怨。有关主题(工作条件，工作满意度，项目，同事和经理)的调查通常反映团队的士气，并在团队超负荷时产生更多的负面结果。与团队负责人定期进行积极的聆听会议可能会发现您没有意识到的问题。主动聆听的基本要素是不加判断地聆听。

#### 团队成员长时间工作和/或生病时工作

无偿加班可能会成为社会心理压力源。领导者应树立一个好榜样：按时工作，生病时待在家里。

#### 更常见的疾病<sup>107</sup>

过度劳累的团队成员往往更容易遭受挫折和生病。

#### 不健康的任务队列

我们建议您定期检查团队的任务队列，以查看积压了多少故障单，谁在处理哪些问题以及哪些任务可以延迟或放弃。如果团队没有按时完成任务，或者紧急情况使您无法定期执行此检查，则团队积累的中断很可能比他们能处理的快。

#### 指标不平衡

一些关键指标可能表明您的团队超负荷了：

- 关闭一个问题的时间周期长
- 花很多时间在琐事上
- 大量时间来解决由呼叫中产生的问题

团队应该共同决定要使用的度量。没有一种万能的方法。每个团队的超载都以不同的方式反映出来。作为经理，在不了解每个人的工作量和工作习惯的情况下，不要对团队施加压力。如果您坚持使用特定的措施，团队成员可能会觉得您不理解工作。例如，如果您根据修复问题所需的天数评估负载，则一个人可能会整天工作以解决问题，而另一个人可能将工作与其他工作一起分配几天。

### 减少超载并恢复团队健康

阅读标准后，您可能会认为您的团队已经超负荷工作。别失望！本节提供了使您的团队恢复健康状态的想法列表。

总的来说，赋予团队成员更多的控制权和权力可以减少感官上的负担。<sup>108</sup> 虽然经理可能会在压力很大的情况下诉诸于微观管理，但重要的是使团队保持循环并共同进行优先排序。<sup>109</sup> 此模型假设一个您在管理与团队成员之间以及团队成员之间[功能团队的基线](#)。

### 确定并缓解心理压力源

在修复功能失调的团队时，最重要的是，每个团队成员都需要重新获得心理上的安全感。团队只能行使其个人成员的职责。

您可以从确定和减轻每个人和整个团队的心理压力源<sup>110</sup> 开始。您实际上可以控制以下哪些因素？您无法控制团队成员是否患有重大疾病，但是您可以\*控制团队的积压(如案例研究1中所示)或静音呼叫(如案例研究2中的大小)。

与您的合作伙伴产品开发团队进行沟通，并让他们知道您的团队超负荷。他们也许能够伸出援手，提供同情心，甚至接管整个项目。

当您的团队成员相互依赖并达到一定水平的[心理安全](#)(这样他们就可以承担人际交往的风险)时，您可以给单个团队成员更多的责任。发现专业领域并为关键人员和技术线索分配特定技术可以增强他们的自信心，从而使他们能够承担风险。

决策应该透明，并且在可能的情况下民主。每个团队成员都应该对局势有控制感。例如，案例研究2中的集思广益会议帮助团队确定并讨论了问题。

### 在四分之一时间内对优先级进行分类

一个健康的团队可以对问题进行优先排序和分类。案例研究1提供了此练习的一个很好的例子：团队坐在一个房间里，并查看他们的积压工作。审查帮助他们意识到他们已经超负荷。他们重新安排了工作的优先级，并致力于可以减少某些超负荷工作的任务。案例研究2中的团队现在在每个季度末开会，以计划并确定现有和未来的合作重点。

如果可能，我们建议SRE在其日历上安排“无中断时间”(无呼叫时间)，以便他们有时间从事定性困难的任务，例如开发自动化系统和调查中断的根本原因。在案例研究2中，当远程团队给值班人员一些救济时，团队成员便有宝贵的时间专注于他们的项目。

如果绝对必要，则放弃工作：在案例研究2中，团队将这一职责交给开发团队，从而放弃了对其中一项服务的值班支持。

### 保护未来的自己

我们强烈建议建立度量标准以评估团队的工作量。定期检查指标，以确保它们衡量正确的事物。

一旦您的团队摆脱了过载，您可以通过采取措施监控或解决潜在问题来防止将来的过载。例如，案例研究1中的团队现在维护一个轻量级的分类流程，以检测不断增加的任务积压。案例研究2中的团队目前正在制定一项长期计划，以使后端和服务SLO保持一致。

当您的团队处于超负荷状态时，对项目工作进行优先级排序，在你没过载的情况下及时交付比你要交付的更多的重复琐事工作。您将来会获利。

最后，团队中的每个人都应对指示可能出现过载情况的早期预警信号负责(请参阅第366页的“识别过载症状”)。如果经理认为团队正在超负荷工作，经理应该坐下来与他们交谈。

### 结论

在理想的情况下，SRE团队将始终能够使用第一本书中描述的策略来管理中断。但是我们只是人，有时我们的团队达不到理想。本章研究了过载可能消耗团队的一些方式，并讨论了如何在过载时进行检测和响应。

特别是在进行运维工作时，过多的中断很容易导致团队从正常的工作量转移到过载。频繁的中断会导致过载，并且过载会对运行状况和生产率产生负面影响。过载会给团队成员带来心理压力，这会进一步影响工作，导致自我执行周期。

感观的过载是过载的一种特殊形式，无法通过琐事的工作或工作量来衡量。很难查明和消除。

为了使团队的工作负载保持平衡，重要的是不断监控(感观或非感观的)过载。为了更好地服务于用户并做好工作，您需要首先对自己和团队表示尊重。在日常工作中保持健康的平衡对帮助您和您的团队实现这一目标大有帮助。

103. Kara A. Latorella，调查中断: 对Flightdeck性能的影响(弗吉尼亚州汉普顿: 兰里研究中心，1999年)，<https://go.nasa.gov/2Jc50Nh>；NTSB，飞机事故报告: NWA DC-9-82 N312RC，底特律都会区，1987年8月16日(编号NTSB / AAR-88 / 05)(华盛顿特区: 全国交通安全局，1988年)，<http://libraryonline.erau.edu/online-full-text/ntsb/aircraft-accident-reports/AAR88-05.pdf>。 ↵

104. Emmanuelle Brun和Malgorzata Milczarek，与职业安全与健康有关的新兴社会心理风险的专家预测(西班牙毕尔巴鄂: 欧洲工作安全与卫生局，2007年)，[https://\\*\\*osha.europa.eu/zh-CN/tools-and-publications/publications/reports/7807118](https://**osha.europa.eu/zh-CN/tools-and-publications/publications/reports/7807118)；M. Melchior，I. Niedhammer，LF Berkman和M. Goldberg，“社会心理工作因素和社会关系是否对疾病缺席发挥独立影响? GAZEL队列的六年前瞻性研究”，《流行病学与社区健康杂志》57，第4期(2003年): 285–93，<http://jech.bmjjournals.org/content/jech/57/4/285.full.pdf>。 ↵

105. 就上下文而言，根据估计，SRE每班需要至少再增加一天的故障单跟踪。 ↵

106. 我们使用了基于Project Aristotle的Google程序: <http://bit.ly/2LPemR2>。 ↵

107. Kurt GI Wahlstedt和Christer Edling，"邮政分拣站的组织变革-它们对工作满意度，心身投诉和病假的影响"，《工作与压力》，第11期，第11页。3(1997): 279--91。 ↵
108. 小罗伯特·卡拉塞克(Robert A. Karasek)，"工作需求，工作决策自由度和精神紧张-对工作重新设计的影响"，《管理科学季刊》，第24期，第24页。2(1979): 285--308。 ↵
109. 弗兰克·邦德(Frank W. Bond)和大卫·邦斯(David Bunce)，"工作控制在减少压力的工作重组干预中发挥了作用"，《职业健康心理学杂志》6，第6期。4(2001): 290--302；Toby D. Wall, Paul R. Jackson和Keith Davids，"操作员的工作设计和机器人系统性能: 偶然的田野研究，"《应用心理学杂志》77，第3号(1992年): 353--62。 ↵
110. Brun和Malgorzata，专家预测。 ↵

# 第18章

## SRE参与模式

由迈克尔·怀尔德潘纳(Michael Wildpaner)，格兰妮·希林(Gráinne Sheerin)，丹尼尔·罗杰斯(Daniel Rogers)和苏里亚·普拉山斯·萨纳加瓦拉普(Surya Prashanth Sanagavarapu)(《纽约时报》)与阿德里安·希尔顿(Adrian Hilton)和Shylaja Nukala撰写

第32章介绍了SRE团队可以用来分析和提高服务可靠性的技术和程序方法。这些策略包括生产准备情况审查(PPR)，尽早参与和持续改进。

简而言之，SRE原则旨在最大程度地提高开发团队的工程速度，同时保持产品的可靠性。这两个目标对产品用户和公司都有利。但是，即使是最好的SRE团队能完成多少工作也是有限制的，并且当域太大且过于复杂时，SRE模型的效率就会降低。当前的微服务运动使这种动态变得更加严峻-小公司可以轻松拥有比单个SRE团队可以处理的更多的微服务。鉴于生产环境广阔，并且知道他们无法涵盖所有服务，因此SRE团队必须决定将精力集中在哪里以取得最佳效果。产品开发和SRE团队可以合作确定正确的焦点。

本章采用了SRE团队的观点，该团队打算为新服务提供支持。我们研究如何最有效地与服务以及拥有该服务的开发人员和产品团队互动。尽管SRE参与通常围绕一项或多项服务构建，但参与所涉及的不仅仅是服务本身-它着重于了解开发人员和产品团队的目标并找到支持它们的正确方法。

无论组织的规模如何，大部分讨论都适用。尽管我们经常使用“团队”一词，但从理论上讲，团队可以从一个人开始(尽管那个人会很忙)。无论您的团队规模大小，主动定义SRE的角色并管理与产品开发的沟通与协作都是很重要的。

## 服务生命周期

如第一本SRE书籍的序言(<http://bit.ly/2LexAhJ>)中所述，SRE团队对服务可靠的贡献贯穿于服务生命周期的所有阶段。他们的生产知识和经验的应用可以在SRE接起服务呼叫之前大大提高服务的可靠性。

图18-1显示了服务生命周期内SRE参与的理想水平。但是，SRE团队可能会在生命周期的任何阶段开始参与服务。例如，如果开发团队开始为SRE支持的服务计划替换服务，则SRE可能会很早就参与新服务。或者，一旦服务普遍可用几个月或几年，并且现在正面临可靠性或扩展性挑战，则SRE团队可以正式参与该服务。本节提供有关SRE团队如何在每个阶段有效做出贡献的指南。

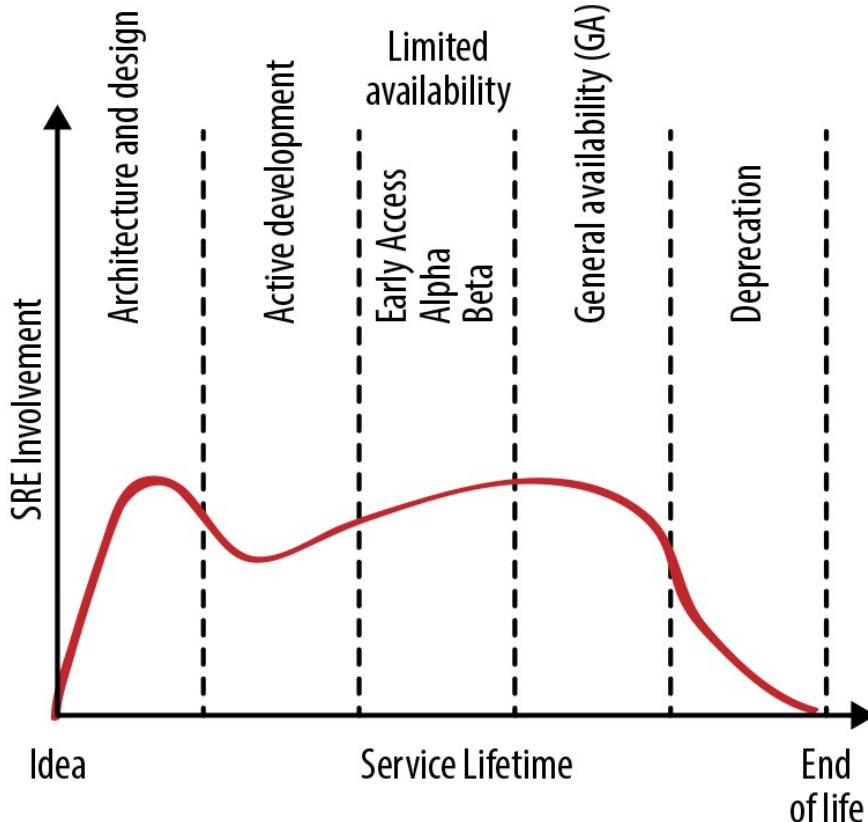


图18-1.服务生命周期中的SRE参与水平

### 阶段1:建筑与设计

SRE可以以不同方式影响软件系统的体系结构和设计:

- 创建最佳实践，例如对各种单点故障的适应能力，开发人员团队在构建新产品时可以采用
- 文档化特定基础架构系统的能做与不能做什么(基于先前的经验)，以便开发人员可以明智地选择其构建基块，正确使用它们并避免已知的陷阱
- 提供早期参与咨询，以详细讨论特定的架构和设计选择，并在目标原型的帮助下帮助验证假设
- 加入开发团队并参与开发工作
- 共同设计服务的一部分

在开发周期的后期，解决架构错误变得更加困难。早期的SRE参与可以帮助避免昂贵的重新设计，当系统与实际用户互动并且需要根据服务增长而扩展时，重新设计就变得非常必要。

### 阶段2:积极发展

当产品在积极开发过程中成形时，SRE可以开始“生产”服务--即使其成形并投入生产。生产化通常包括容量规划，为冗余设置额外的资源，针对尖峰和过载处理进行规划，实现负载平衡以及实施诸如监控，警报和性能调整之类的可持续运维实践。

### 阶段3:有限的可用率

随着服务向Beta的发展，用户数量，用例的多样性，使用强度以及可用性和性能需求都在增加。在此阶段，SRE可以帮助测量和评估可靠性。我们强烈建议在一般可用性(GA)之前定义SLO，以便服务团队可以客观地衡量服务的可靠性。产品团队仍然可以选择退出不能达到其目标可靠性的产品。

在此阶段中，SRE团队还可以通过建立容量模型，为即将启动的阶段获取资源以及自动执行开通和就地调整服务大小来帮助扩展系统。SRE可以确保适当的监控范围，并帮助创建与即将到来的服务SLO理想匹配的警报。

在服务使用情况仍在变化的同时，SRE团队可以期望在事件响应和操作职责期间增加工作量，因为该团队仍在学习服务的工作方式以及如何管理其故障模式。我们建议在开发人员和SRE团队之间共享这项工作。这样，开发人员团队可以获得该服务的运维经验，而SRE则可以用一般的服务获得经验。运维工作和事件管理将告知系统更改和更新服务拥有者在Google Analytics(分析)之前需要进行的工作。

#### 阶段4:一般可用性

在此阶段，该服务已通过“生产准备情况审查”(有关更多详细信息，请参阅站点可靠性工程中的[第32章](#))，并且正在接受所有用户。尽管SRE通常执行大部分运维工作，但开发人员团队应继续从事所有运营和事件响应工作的一小部分，以使他们不会对服务的这些方面失去认识。他们可能会在呼叫轮换中永久包含一名开发人员，以帮助开发人员跟踪运维负荷。

在GA的早期阶段，由于开发团队专注于使服务成熟并启动第一批新功能，因此它也需要保持循环以了解实际负载下的系统属性。在Google Analytics(分析)的后期阶段，开发人员团队会提供一些小的增量功能和修补程序，其中一些功能是根据运维需求和发生的任何生产事件告知的。

#### 阶段5:弃用

没有系统会永远运行。如果有更好的替换系统可用，则对新用户关闭现有系统，并且所有工程都致力于将用户从现有系统过渡到新系统。SRE主要在没有开发人员团队参与的情况下运维现有系统，并通过开发和运维工作来支持过渡。

虽然减少了现有系统所需的SRE工作量，但SRE有效地支持了两个完整的系统。员工人数和人员配备应作相应调整。

#### 阶段6:放弃

放弃服务后，开发团队通常会恢复运维支持。SRE尽力支持服务事件。对于具有内部用户的服，SRE将服务管理移交给其他所有用户。本章提供了两个案例研究，说明SRE如何将服务交还给开发人员团队。

#### 阶段7:不支持

没有更多的用户，并且该服务已关闭。SRE帮助删除生产配置和文档中对服务的引用。

## 建立关系

服务不是凭空存在的:SRE团队与构建服务的开发团队以及确定其发展方式的产品团队合作。本节为建立和维持与这些团队的良好工作关系提出了一些策略和策略。

#### 沟通业务和生产优先级

在可以帮助某人之前，您需要了解他们的需求。为此，SRE需要了解产品开发人员期望SRE参与实现的目标。与开发人员团队互动时，SRE应该建立对产品和业务目标的深刻理解。SRE应该能够阐明自己的角色，以及SRE的参与如何使开发人员能够朝着这些目标执行。

团队需要定期就业务和生产优先级进行讨论。SRE和开发人员领导团队在理想情况下应该作为一个整体工作，定期开会并就技术和优先级挑战交换意见。有时SRE领导会加入产品开发领导团队。

### **识别风险**

由于SRE团队专注于系统可靠性，因此他们可以很好地识别潜在风险。尽可能准确地衡量这些风险的可能性和潜在影响非常重要，因为破坏常规开发和功能流程的成本对于产品和工程师而言都是很重要的。

### **统一目标**

开发人员和SRE团队都关心可靠性，可用性，性能，可伸缩性，效率以及功能和启动速度。但是，SRE在不同的激励机制下运作，主要是支持服务的长期生存能力而不是新功能的发布。

根据我们的经验，开发人员和SRE团队可以在这里保持适当的平衡，同时保持各自的工作重点，同时也明确支持其他团队的目标。SRE可以有一个明确的目标，以支持开发人员团队的发布速度，并确保所有已批准发布的成功。例如，SRE可能会声明：“我们将支持您尽快安全发布”，其中“安全”通常表示保持在错误预算之内。然后，开发人员应致力于在工程上花费合理的时间来修复和防止破坏可靠性的事项：在设计和实施级别解决持续存在的服务问题，偿还技术债务，并尽早在新功能开发中包括SRE，以便他们可以参加设计对话。

### 共享目标:《纽约时报》的SRE参与

由Surya Prashanth Sanagavarapu(纽约时报)撰写

在我们的组织中，当涉及到云迁移，生产加速以及应用程序向容器迁移时，对SRE资源的需求很高。此外，SRE团队有自己的待办事项。面对有限的资源，这些相互竞争的优先事项决定了SRE团队的成功。雇用SRE是解决SRE时间需求的一种显而易见的方法，但并不是每个团队都有能力，经验或时间。

在《纽约时报》上，SRE功能的核心任务是通过产品和工具赋予产品开发团队以使支持我们新闻编辑室的应用程序的可靠性和弹性最大化的方法，从而向读者分发高质量的新闻。我们采用了“共享目标模型”，以在减少自动化积压工作和其他团队合作之间取得平衡。

在与团队合作之前，我们会查看当前季度/年度的总体积压工作，并明确定义其工作项和类别。例如，我们的积压项目可能包括：

- 通过点击应用程序的服务状态端点来增加自动化以设置基线监控和警报。
- 实施更可靠和/或更快速的构建管道。

当团队向SRE寻求帮助时，在对请求进行优先级排序时，我们考虑的因素之一是联合参与是否可以帮助减少积压。

### 定义参与度

我们的SRE根据两种不同的模式与产品开发团队合作：

- 全职
- 简短和受限项目的兼职支持

我们根据SRE团队的带宽来定义参与类型。对于全职工作，我们希望将SRE嵌入产品开发团队中。这有助于提供重点和时间来减轻产品工程团队的负担。随着开发人员提高SRE技能和能力，SRE和产品团队有最大的时间互相学习。对于长期合作，我们会优先考虑最适合我们公司战略的应用程序。

在定义参与范围时，我们尝试评估与SRE实践相关的团队或应用程序的成熟度。我们发现，在考虑SRE做法和原则时，各个团队的成熟程度不同。我们正在努力应用成熟度模型来提供帮助。

### 设定共同的目标和期望

设定正确的期望对于满足截止日期和任务完成至关重要。为此，我们根据以下原则进行工作：

- 我们强调，应用程序所有者(而不是SRE)直接负责对应用程序进行更改。
- SRE参与是为了整个公司的利益。任何新的自动化或工具都应改进整个公司使用的通用工具和自动化，并避免一次性脚本开发。
- SRE应该使开发人员团队了解参与可能引入的任何新流程(例如，负载测试)。

- 如*Site Reliability Engineering*的第32章中所述，约定可能涉及应用程序准备情况审查(ARR)和生产准备情况审查(PRR)。开发人员和SRE必须共同优先考虑ARR和PRR的建议更改。
- SRE不是传统的运维工程师。他们不支持手动工作，例如运行要部署的作业。

设定共享目标时，我们将与开发团队一起编写这些目标，并将目标划分为多个里程碑。如果您是一家基于敏捷的公司，则可以编写epics或故事。然后，SRE团队可以将这些目标映射到自己的待办事项中。设定目标时，我们常见的模式是：

1. 定义参与范围。
  - **范例1:**在下个季度，我希望我的团队中的所有成员都能够处理GKE/GAE部署，适应生产环境并能够处理生产中断。
  - **范例2:**在下个季度中，我希望SRE与开发团队合作，以在扩展和监控方面稳定该应用程序，并为停机创建运行手册和自动化。
2. 确定最终结果成功案例，并明确指出。
  - **示例:**参与之后，产品开发团队可以在Google Kubernetes Engine中处理我们的服务中断，而无需升级。

### 冲刺和沟通

与产品开发团队的任何接触都始于启动和计划会议。在启动之前，我们的SRE团队会审查应用程序体系结构和我们的共同目标，以验证预期结果在给定时间范围内是切合实际的创建史诗和故事的联合计划会议可以成为参与的良好起点。

参与的路线图可能是：

1. 查看应用程序体系结构。
2. 定义共同的目标。
3. 举行启动和计划会议。
4. 实施开发周期以达到里程碑。
5. 设置回顾以征集参与反馈。
6. 进行生产准备情况审查。
7. 实施开发周期以达到里程碑。
8. 计划并执行发射。

我们要求团队定义一种反馈方法并就其频率达成共识。SRE和开发团队都需要有关什么有效和什么无效的反馈。为了使这些合作取得成功，我们发现通过协商一致的方法在敏捷冲刺审阅之外提供恒定的反馈循环很有用，例如，每两周进行一次回顾或与团队经理签到。如果SRE参与没有奏效，我们希望团队不要回避计划脱离参与。

### 衡量影响

我们发现，衡量参与的影响以确保SRE正在开展高价值工作非常重要。我们还将衡量每个合作伙伴团队的成熟度，以便SRE可以确定与他们合作的最有效方式。我们与Google的客户可靠性工程(CRE)团队合作采取的一种方法是，在开始参与之前，与产品工程团队的负责人进行“时间点评估”。

一个时间点评估包括遍历一个成熟度矩阵，沿着SRE的各个关注轴衡量服务的成熟度(如站点可靠性工程的[第32章](#)中所述)，并就可观察性，容量规划，变更管理和事件响应等功能领域的分数达成一致。在我们进一步了解团队的优势，劣势和盲点之后，这也有助于更恰当地调整参与度。

参与结束后，开发团队将自己完成工作，我们将再次执行评估以衡量SRE增加的价值。如果我们有一个成熟度模型，我们将对该模型进行衡量，以查看参与度是否会导致更高的成熟度。随着参与活动的结束，我们计划举行庆祝活动！

## 设置基本规则

在Google，每个SRE团队都有两个主要目标：

### 短期

通过提供可操作的稳定系统来满足产品的业务需求，该系统可用并根据需求扩展，并着眼于可维护性。

### 长期

将服务运维优化到不再需要正在进行的人工工作的水平，因此SRE团队可以继续进行下一次高价值的工作。

为此，团队应商定一些合作原则，例如：

- 运维工作的定义(以及硬性限制)。
- 服务的商定和可衡量的SLO，用于优先考虑开发人员和SRE团队的工程工作。  
您可以在没有适当的SLO的情况下开始工作，但是我们的经验表明，没有在关系建立之初建立此上下文，这意味着您稍后必须回溯到此步骤。有关在没有SLO的情况下工程工作进展不理想的示例，请参阅“案例研究1：缩放Waze---从临时更改到计划更改”(第427页)。
- 商定的季度误差预算，用于确定释放速度和其他安全参数，例如用于处理意外使用增长的超额服务能力。
- 开发人员参与日常操作，以确保可以看到持续存在的问题，并确定解决其根本原因的优先次序。

## 计划与执行

主动的计划和协调的执行确保SRE团队达到期望和产品目标，同时优化运维并降低运维成本。我们建议在两个(关联)级别进行规划：

- 在开发人员的领导下，确定产品和服务的优先级并发布年度路线图。
- 定期检查和更新路线图，并得出与路线图一致的目标(每季度或其他)。

路线图可确保每个团队都拥有长期清晰，影响重大的工作。有充分的理由放弃路线图(例如，如果开发组织的变更太快)。但是，在稳定的环境中，缺少路线图可能是SRE团队可以与另一个团队合并，将服务管理工作移回开发团队，扩大范围或解散的信号。

与开发人员领导进行持续的战略对话有助于快速确定重点转移，讨论SRE为业务增加价值的新机会或停止对产品而言不合算的活动。

路线图的重点不仅限于改进产品。他们还可以解决如何应用和改进通用SRE技术和流程以降低运维成本的问题。

### 保持有效的持续关系

健康有效的关系需要不断的努力。本节概述的策略对我们来说效果很好。

### 投入时间更好地合作

花时间互相交流的简单行为可帮助SRE和开发人员更有效地协作。我们建议SRE与他们的同行定期会面以了解他们所提供的服务。对于SRE来说，定期与其他运行服务的SRE团队开会也是个好主意，这些团队要么向服务发送流量，要么提供服务使用的通用基础结构。然后，SRE团队可以在出现故障或分歧时自信而迅速地升级，因为这两个团队彼此了解并且已经设定了如何启动和管理升级的期望。

### 保持沟通畅通

除了团队之间的日常沟通外，我们发现了几种更正式的信息交换方法，在参与过程中特别有用。

SRE可以每季度与产品开发负责人进行一次“生产状态”演讲，以帮助他们了解应该在哪里投资资源以及SRE如何准确地帮助他们的产品或服务。同样，开发人员可以向SRE团队进行定期的“产品状态”演讲，也可以让SRE参与开发团队的执行演示。这使SRE团队可以大致了解开发人员团队在上个季度中所取得的成就(并让SRE看看他们自己的工作是如何实现的)。它还提供了有关产品在接下来的几个季度中的发展状况以及产品负责人认为SRE致力于实现这一目标的最新信息。

### 执行常规服务审核

作为服务未来的决策者，SRE和负责服务的开发团队负责人应至少每年面对面一次。比这更频繁地开会可能具有挑战性-例如，因为可能涉及洲际旅行。在本次会议期间，我们通常会分享未来12--18个月的路线图，并讨论新的项目和发布。

SRE团队有时会促进回顾性练习，牵头人讨论团队想要“停止做”，“继续做”和“开始做”的事情。项目可以出现在多个区域中，并且所有意见都是有效的。这些会议需要积极的协助，因为最好的结果来自全团队的参与。在服务会议中，这通常被认为是最有用的会议，因为它会产生可以推动重大服务更改的详细信息。

### 当基本规则开始下滑时重新评估

如果在任何已达成共识的领域中的合作(请参阅第379页的“设置基本规则”)开始退步，则开发人员和SRE都需要更改优先级以使服务恢复正常。我们发现，根据紧急程度，这可能意味着以下任何情况：

- 团队确定了特定的工程师，这些工程师必须放弃其较低优先级的任务以专注于回归。

- 两个团队都将其称为"可靠性黑客马拉松"，但通常在黑客马拉松日之后一般团队优先级会继续。
- 宣布冻结功能，并且两个团队中的大多数都专注于解决回归问题。
- 技术领导者确定产品的可靠性正处于严重风险中，并且团队称其为"全力以赴"。

#### 根据您的SLO和错误预算调整优先级

精心定义明确的SLO的巧妙技巧可帮助团队适当地确定优先级。如果某项服务有丢失SLO的危险或用尽了错误预算，则两个团队都可以高优先级地工作以使服务恢复安全。他们可以通过战术措施(例如，为解决与流量相关的性能下降而过度配置)和更具战略意义的软件修复(例如优化，缓存和正常降级)来解决这种情况。

如果服务在SLO范围内，并且有足够的错误预算，我们建议使用备用错误预算来提高功能速度，而不是花费过多的精力进行服务改进。

#### 适当处理错误

人类不可避免地会犯错误。与我们的[事后总结文化](#)一致，我们不会怪罪别人，而是将注意力集中在系统行为上。您的里程可能会有所不同，但是我们在以下策略上取得了成功。

#### 考虑一晚上

如果可能，在您感到疲倦或情绪激动时不要进行跟进对话。在压力很大的情况下，人们很容易误解电子邮件等书面交流中的语气。读者会记住单词的感觉，而不一定记住所写的内容。当您在各地交流时，通常值得花时间进行视频聊天，以便您可以看到面部表情并听到有助于消除单词歧义的语气。

#### 亲自(或尽可能接近)开会以解决问题

仅通过代码审查或文档进行的交互就会很快被抽出并令人沮丧。当另一个团队的行为或决定与我们的期望相抵触时，我们会与他们讨论我们的假设，并询问缺失的背景。

#### 要积极

感谢人们的积极行为。这样做很简单-例如，在代码审查，设计审查和失败场景培训期间，我们要求工程师指出什么是好的，然后解释原因。您也可能会认可良好的代码注释，或者感谢人们投入大量时间进行严格的设计审查。

#### 了解沟通差异

不同的团队对如何传播信息有不同的内部期望。了解这些差异可以帮助加强关系。

## 将SRE扩展到更大的环境

到目前为止，我们讨论的场景涉及一个SRE团队，一个开发人员团队和一个服务。较大的公司，甚至使用微服务模型的小型公司，可能都需要扩展其中一些或全部数量。

#### 通过一个SRE团队支持多种服务

由于SRE具有专业技能并且是稀缺资源，因此Google通常将SRE与开发人员的比率保持在<10%。因此，一个SRE团队通常在其产品领域(PA)中与多个开发人员团队合作。

如果相对于需要SRE支持的服务数量而言，SRE稀缺，则SRE团队可以将精力集中在一项服务或少数开发人员团队的少数服务上。

根据我们的经验，如果那些服务具有以下特征，则可以将有限的SRE资源扩展到许多服务：

- 服务是单个产品的一部分。这提供了用户体验的端到端所有权，并与用户保持一致。
- 服务建立在类似的技术堆栈上。这样可以最大程度地减少认知负担，并有效地重用技术技能。
- 服务由同一开发人员团队或少数相关开发人员团队构建。这样可以最大程度地减少关系的数量，并可以更轻松地调整优先级。

### 构建多个SRE团队环境

如果您的公司规模足够大，可以拥有多个SRE团队，也许还有多个产品，则需要选择一种结构来确定SRE与产品组之间的关系。

在Google内部，我们支持复杂的开发人员组织。如图18-2所示，每个PA包含多个产品组，每个产品组包含多个产品。SRE组织在层次上以共同的优先级和最佳实践来覆盖开发人员组织。当一个组中的所有团队或PA中的所有组共享相同或相似的特定业务目标时，并且每个产品组都有产品负责人和SRE负责人时，此模型将起作用。

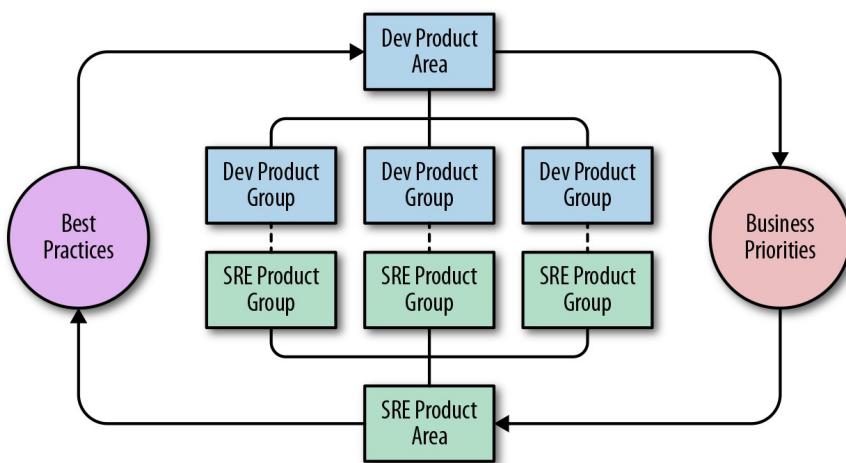


图18-2.大规模的开发人员到SRE团队关系(按产品区域)

如果您的组织有多个SRE团队，则需要以某种方式将它们分组。我们看到的两种有效的主要方法是：

- 将产品中的团队进行分组，因此他们不必与太多不同的开发人员团队进行协调。
- 将团队归入技术堆栈(例如"存储"或"网络")中。

为了防止开发人员重组期间SRE团队流失，我们建议根据技术而不是开发人员PA报告结构来组织SRE团队。例如，许多支持存储系统的团队的结构和运作方式都相同。将存储系统分组为以技术为重点的产品组可能更有意义，即使它们来自开发人员组织的不同部门。

### 使SRE团队结构适应不断变化的情况

如果您需要修改SRE团队的结构以反映不断变化的PA需求，我们建议根据服务需求以及工程和运维负载来创建，拆分(分片)，合并和解散SRE团队。每个SRE团队都应有清晰的章程，以反映其服务，技术和运维。当单个SRE团队提供的服务过多时，而不是从头开始构建新团队，我们更愿意将现有团队分为多个团队，以转移文化并增强现有领导能力。这样的更改不可避免地会破坏现有团队，因此我们建议您仅在必要时重组团队。

### 运行内聚分布式SRE团队

如果您需要确保24/7全天候覆盖范围和业务连续性，并且业务遍及全球，则值得尝试将您的SRE团队分布在全球各地，以提供均匀的覆盖范围。如果您有许多遍布全球的团队，我们建议根据相邻性以及服务和共享技术的相似性来安排团队。我们发现，单例团队通常效率较低，并且更容易受到团队外部重组的影响-只有在明确定义的业务需求需要他们并且我们考虑了所有其他选择的情况下，我们才创建这样的团队。

许多公司没有足够的资源来覆盖全球，但是即使您仅分布在各个建筑物之间(不要介意各大洲)，创建和维护两个地点的安排也很重要。

创建和维护组织标准以驱动计划和执行，并促进和维护共享的团队文化也很重要。为此，我们发现定期将整个团队聚集在一个物理位置很有用-例如，每12--18个月组织一次跨组织范围的峰会。

有时，团队中的每个人都承担某些职责是没有意义的-例如，从备份执行常规测试还原或实施跨公司的技术任务。在团队的分布式站点之间平衡这些职责时，请记住以下策略：

- 将个人职责分配给单个地点，但要定期轮换(例如，每年一次)。
- 分担地点之间的所有责任，积极努力平衡参与和工作量。
- 不要常年将责任锁定在单个位置。我们发现，这种配置的成本最终超过了收益。尽管该位置通常会变得非常善于履行这些职责，但这会养成“我们与他们对立”的心态，阻碍知识的分配，并带来业务连续性的风险。

所有这些策略都需要位置以维持战术和策略沟通。

## 结束关系

SRE参与不一定是无限期的。SRE通过进行有影响力的工程工作来提供价值。如果工作不再具有影响力(即SRE约定的价值主张消失了)，或者如果大部分工作不再在工程方面(相对于运维)，则可能需要重新审视正在进行的SRE约定。通常，各个SRE将从繁重的团队转移到从事更有趣的工程工作的团队。

在团队级别，如果SRE不再提供相对于成本足够的业务价值，您可能会退还服务。例如：

- 如果服务已优化到不再需要持续参与SRE的水平
- 如果服务的重要性或相关性降低了
- 如果服务即将终止

以下案例研究说明了两种Google SRE参与模式是如何结束的。第一个结局以积极的结果告终，而其他的结局则以更为细致的告终。

### 案例研究1:Ares

Google的滥用行为SRE和通用滥用工具(CAT)团队为大多数Google产品提供了反滥用保护，并与面向客户的产品合作，确保用户安全。滥用SRE团队应用工程工作来减轻CAT的运营支持负担，从而使开发人员能够直接为用户提供支持。这些用户是操作由CAT捍卫的财产的Google员工，他们对CAT的功效及其对问题或新威胁的响应时间寄予厚望。

面对新的威胁和攻击，有效的对抗滥用行为需要不断关注，快速的适应性变化和灵活的灵活性。这些要求与可靠的和计划的功能开发的通用SRE目标相冲突。CAT团队通常需要实施快速开发并将新的保护部署到受攻击的属性。但是，滥用SRE推迟了请求的更改，要求对每个新保护措施对整个生产系统的后果进行更深入的分析。团队之间的磋商和评审的时间限制加剧了这种紧张关系。

为了希望改善情况，Abuse SRE和CAT领导层参与了一个多年项目，以在CAT中创建专门的基础架构团队。新组建的"Ares"团队负责统一Google财产的反滥用基础架构。该团队由CAT工程师组成，他们具有生产基础架构方面的知识，并且具有构建和运行大型服务的经验。这些团队启动了一项交流计划，以将生产管理知识从"滥用SRE"转移到CAT基础结构团队成员。

滥用SRE告知Ares团队，在生产环境中启动新服务的最简单方法(当您已经在运行大型分布式服务时)是最大程度地减少服务施加的额外认知负担。为了减少这种认知负担，系统应尽可能均质。一起部署和管理一系列生产服务意味着它们可以共享相同的发布结构，容量计划，用于访问存储的子服务，等等。根据此建议，Ares重新设计了整个反滥用堆栈，应用了模块化概念以转向微服务模型。他们还构建了一个新层，为开发人员提供了抽象，因此他们不必担心较低级别的生产详细信息，例如监控，日志记录和存储。

在这一点上，Ares团队开始通过管理新的反滥用基础设施，开始更像CAT的SRE团队。同时，Abuse SRE专注于整个反滥用基础设施的生产部署和高效的日常运维。

Ares工程师和Abuse SRE之间的协作带来了以下改进：

- 由于CAT团队现在拥有"内部"生产专家，他们也是打击滥用的专家，因此Abuse SRE不再需要审查新功能集成。这大大减少了生产新功能的时间。同时，由于新的基础架构抽象了生产管理细节，因此CAT团队的开发人员速度有所提高。
- 滥用SRE团队现在从CAT团队发出新功能的请求减少了，因为大多数请求都不需要更改基础结构。由于很少需要更改基础架构，因此该团队还需要较少的知识来评估新功能的影响。当需要更改基础架构时，Abuse SRE仅需要澄清对基础架构的含义，而不是特定功能。
- 需要集成到反滥用基础架构中的产品具有更快，更可预测的周转时间，因为产品集成现在的功能相当于功能发布。

在该项目结束时，负责滥用SRE不再直接支持CAT，而是专注于基础架构。这不会损害CAT的可靠性，也不会增加CAT团队的额外工作负担；相反，它提高了CAT的整体开发速度。

目前，Ares保护着众多Google产品和服务的用户。自团队成立以来，SRE和产品开发已就基础架构在生产中的工作方式进行合作决策。之所以建立这种伙伴关系，是因为Ares的努力营造了共同的命运感。

### 案例研究2:数据分析管道

有时，维护SRE支持关系的成本高于SRE提供的价值(感官的或可衡量的)。在这些情况下，通过解散SRE团队来终止关系是有意义的。[111](#)

当关系的价值随时间下降时，很难确定终止该关系有意义的时间点。Google的两个支持收入关键型数据分析管道的团队必须面对这一挑战。弄清楚分开方式是不容易的，特别是在十年的合作之后。回顾过去，我们能够确定团队互动中的几种模式，这是我们重新考虑SRE团队与产品团队之间关系的有力指标。

### 关键

关闭之前的三年，所有参与方都意识到他们的主要数据分析管道正在遇到扩展限制。那时，开发团队决定开始计划他们的新系统，并让少数工程师致力于新工作。随着这项工作的开始，有必要对现有系统的大型，复杂或高风险功能的开发进行优先级排序，以支持新系统的工作。随着时间的推移，这产生了两个重要影响：

- 对新项目采用了非正式规则：如果项目的复杂性或修改现有系统以适应项目的风险很高，那么最好在新系统上进行投资。

随着资源转移到开发新系统，即使对现有系统进行相对保守的更改也变得更加困难。然而，使用量继续以极高的速度增长。

### 沟通破裂

在同时设计，建造和发布替换系统的同时，要保持现有系统的正常运行对任何工程团队来说都是一个挑战。专注于新系统和旧系统的人员之间自然会建立压力，团队需要做出困难的优先级决策。当团队在组织上分开时，这些困难会变得更加复杂-例如，专注于维护和运维现有系统的SRE团队和致力于下一代系统的开发团队。

在整个周期中，定期，开放和合作的交流对于维持和保持团队之间良好的工作关系至关重要。在此示例中，沟通上的差距导致团队之间的关系破裂。

### 退役

花了一些时间才意识到SRE和开发团队之间的脱节是无法克服的。最终，最简单的解决方案是消除组织上的障碍，并使开发人员团队可以完全控制优先级划分新旧系统的工作。在完全淘汰旧系统之前，预计这些系统会重叠18--24个月。

将SRE和产品开发功能组合到一个团队中，可以使高层管理人员最大程度地响应其责任范围。同时，团队可以决定如何平衡运维需求和速度。尽管使两个SRE团队退役不是一件令人愉快的经历，但这样做解决了在哪里投入工程精力的持续压力。

尽管开发人员团队不可避免地要承担额外的操作负担，但通过对服务内部知识更深入的人员重新调整旧系统的所有权，仍可以更快地解决操作问题。该团队还对潜在的中断原因有更深入的了解，这通常导致更有效的故障排除和更快的问题解决。但

是，在开发人员团队了解到在短时间内支持该服务所需的操作工作的细微差别时，会产生一些不可避免的负面影响。SRE团队的最后一项工作是使这些知识的传递尽可能顺利，从而使开发人员团队能够承担这项工作。

值得注意的是，如果工作关系更健康-团队有效地合作以解决问题-那么SRE将在短时间内将生产工作交还给开发团队。在为预期的增长需求对系统进行了稳定和加固之后，SRE通常会重新承担系统的责任。SRE和开发团队需要愿意直接解决问题，并确定需要重新设置的紧张点。SRE的工作之一是在面对不断变化的业务需求时帮助保持卓越的生产，这通常意味着与开发人员合作以找到解决难题的解决方案。

## 结论

SRE团队参与的形式会在服务生命周期的各个阶段发生变化。本章提供了针对每个阶段的建议。Google和"纽约时报" SRE团队的示例表明，有效管理参与度与制定良好的技术设计决策同样重要。有时，参与SRE会得出自然结论。来自Ares和数据分析管道团队的案例研究提供了如何发生这种情况以及如何最好地终止参与的示例。

关于在SRE和产品开发团队之间建立有效关系的最佳实践，通过定期和公开的沟通来共享目的和目标至关重要。您可以通过多种方式扩展SRE团队的影响力，但这些关系管理原则应始终成立。为了维持参与的长期成功，投资于调整团队目标和理解彼此的目标与捍卫SLO一样重要。

<sup>111</sup>. Google HR通过在发生这种转变时寻找新的机会来支持员工。 ↵

# 第19章

## SRE:跨越高墙

由戴夫·伦森

与Betsy Beyer, Niall Richard Murphy和Liz Fong-Jones撰写

从我们开始在Google实践SRE至今已有14年了。回顾过去，当时发生的某些事情似乎显而易见，而其他事态发展令人震惊。自[我们的第一本SRE书](#)出版以来的过去两年中，特别有趣。现在，实践SRE准则的公司数量以及我们在会议上和与客户讨论该问题所花费的时间已经超出了我们以前的想象。

尤其是这种变化-围绕SRE的非Google生态系统的迅速扩展-是最令人振奋的发展，但它使预测SRE行业的未来变得更加困难。尽管如此，在我们自己在Google的SRE工作中，我们开始看到一些趋势，这些趋势可能会为该行业的未来提供一个轮廓。本章介绍了我们与全球SRE同事分享的经验以及迄今为止所得出的结论。

## 我们坚持不言而喻的真理

充分利用未来的唯一方法就是从一系列原则开始并向前迈进。接下来的一些事情应该没有争议。不是很多。不过，在每种情况下，这些原则都是基于[我们在世界上看到的](#)真实事物。

### 可靠性是最重要的功能

当我们断言"可靠性是任何系统的最重要特征"时，人们通常不会与我们意见相左。只要我们注意指出"可靠性"通常涵盖广泛的领域即可。

论据很简单：

- 如果系统不可靠，用户将不会信任它。
- 如果用户不信任系统，则在选择时将不会使用该系统。
- 由于所有软件系统都受网络效应的支配，因此，如果一个系统没有用户，那么它一文不值。
- 您就是要衡量的对象，因此请仔细选择指标。

### 您的用户,而不是您的监控来决定您的可靠性

由于系统的价值与用户有关，因此，唯一重要的可靠性衡量标准就是用户如何体验可靠性。如果您的用户担心您的平台是造成他们不稳定的原因，那么告诉他们"我们的监控看起来很好；问题一定在您自己身上"不会使他们变得脾气暴躁。他们觉得您的系统不稳定，这就是当您和竞争对手之间进行选择时他们会记住的事情。(这种现象称为[峰值规则](#))。

您的监控，日志和警报仅在帮助您先于客户发现问题之前，才有价值。

### **如果您运行平台，那么可靠性就是合作伙伴**

如果某人使用您的系统的唯一方法是通过可视用户界面(例如网页)，并且您的系统仅由实际的人(而不是机器)使用，那么您的可靠性用户体验几乎完全取决于您作为SRE保持系统健康所完成的工作。

但是，一旦添加了API，并且某些"用户"实际上是其他机器，则您正在运行平台，并且规则也会更改。

当您的产品扮演平台时，用户体验的可靠性不仅限于您做出的选择。可靠性成为一种伙伴关系。如果您的用户在平台上构建或运行的系统永远无法获得高于99%的可用性-即使您以99.999%的可用性运行平台-则他们的[best-case体验](#)是98.99901%。

这些用户做出的选择将直接影响他们的体验并与您的服务相关联。您可能不喜欢它，但是即使他们不是您的错，他们也会让您对自己经历的一切负责。

### **一切最终都会成为平台**

由于系统的价值会随着使用该系统的人数的增加而增加，因此，您将想方设法找到其他庞大的已建立用户池。随着您吸引更多用户，其他软件系统也将希望吸引您的受众。

这是其他公司开始使他们的机器通过API与您的机器对话的时候。如果您的系统甚至不受欢迎，那么集成是您发展的必然步骤。

即使您决定不关心其他用户社区，并且决定从不创建机器消费的API，您仍然无法避免这种未来。其他人只会将您的UI[包装](#)到机器API中并使用它。唯一的区别是您将无法控制结果。

一旦您的系统成为通往大量用户的门户，它就变得很有价值。API(正式或非正式)将成为您未来的一部分。

### **当客户遇到困难时，您必须放慢速度**

当您的客户遇到困难时，他们的挫败感会为您带来磨擦。即使您没有传统的支持方式(麻烦的工单，电子邮件，电话等)，您仍然会花时间对问题进行分类，并通过StackOverflow甚至Twitter，Facebook和其他社交平台来答复投诉。

无论您投入什么精力来帮助用户度过难关，都无法投入精力来改善系统。我们已经看到许多团队(和公司)允许他们的时间被打破/修复客户问题所占用，从而使创新预算不断减少。这些团队被琐碎工作消耗。

一旦处于这种状态，就很难进行挖掘(请参阅第6章)。更好的计划是领先于即将到来的工作。您可能正在阅读此书，并在想，*Gee，我在一个内部平台团队中。这不适用于我！*

很抱歉通知您，此"加倍"适用于您！在您的情况下，您的客户就是您公司内部系统的客户。这使我们得出下一个结论。

### **您需要与客户一起练习SRE**

如果您希望客户使用平台设计和运行可靠的系统，则必须教他们如何做。是的，这也包括您的内部客户。仅仅因为您在内部平台团队中工作并不意味着您会摆脱这种动态状态-实际上，您最有可能首先遇到这种动态情况。

即使您可以将信息完美地提取为高度缩放的一对多形式(书籍，博客文章，体系结构图，视频等)，您仍然需要一种方法来

### 我们坚持不言而喻的真理

找出要包括的内容和培训。随着您成长和改进平台，这些课程将改变。您将始终需要一种方法来防止这些资源过时。

学习这些课程的最好方法是与客户"做SRE"。

这不一定意味着您需要为客户的系统使用呼叫，但是您确实需要承担通常导致呼叫切换(意味着系统已满足某些最低可行可靠性要求)的大部分工作。至少是您的用户的代表性样本。

### 如何:与您的客户SRE

与客户一起进行SRE旅程的想法似乎有些艰巨。您可能正在阅读这本书，因为您不确定自己如何走路！别担心。可以同时做两个。实际上，前者可以帮助您加速后者。

这是我们要遵循的步骤。它们对我们来说效果很好，我们认为它们对您也很有用。

#### 第1步:SLO和SLI是您的说话方式

您希望客户将您的系统视为可靠的。否则，您可能会失去它们。因此，有理由推论，您应该非常在乎它们如何形成这些意见。他们测量什么？他们如何衡量？最重要的是，他们向他们的客户做出什么承诺？

如果您的客户测量SLI和[关于SLO的警报](#)，并且与您共享这些测量，那么您的生活就会好很多。否则，您将在以下对话中花费大量精力：

**客户:**API调用X通常花费时间T，但是现在花费时间U。我认为您遇到了问题。请调查一下，并立即与我联系。

**您:**该性能似乎符合我们的期望，并且一切看起来都很好。API调用X花这么长时间会不会有问题？

**客户:**我不知道。通常不需要很长时间，因此很明显有些变化，我们对此感到担心。

这次谈话将循环往复，永远不会获得满意的答案。您将花费大量时间说服客户不要理会他们，或者您将花费大量时间从根本上引起更改，以便说服客户不要理会。无论哪种情况，您都将花费很多精力在其他地方使用。

此问题的根本原因是客户没有使用SLO来确定他们是否应该关心他们所看到的性能。他们只是注意到一个意想不到的变化，并决定担心它。请记住，在没有明确的SLO的情况下，您的客户将不可避免地发明一个，并且直到您没达到这个SLO才告诉您！您宁愿进行以下对话：

**客户:**我们对于应用程序FOO的SLO燃烧得太快了，应用程序处于危险之中。SLI X和Y似乎从悬崖上掉下来了。它们都取决于您的API X。

**您:**好的。让我研究一下API X在我们系统中的运行情况和/或特定于您的运行情况。

这是一个更加富有成效的对话，因为(a)仅在SLO受到威胁时才会发生，并且(b)它依赖于相互理解的指标(SLI)和目标(SLO)。

如果您正在使用SRE实践来运行系统，那么您内部就是在说SLO。如果他们也说SLO，那么您的生活会更好，客户也会更加快乐，因为这使你们两个之间的交谈变得更加容易。

我们建议您做一个简单的练习，以改善与客户的工作关系：与客户坐下来。解释SLO，SLI和错误预算-尤其是您在团队中的实践方式。然后帮助他们以这些术语描述他们在平台上构建的关键应用程序。

#### 第2步:审核监控和构建共享仪表板

一旦您的客户为他们的应用选择了一些基本的SLO，下一个问题是他们是否正在测量正确的事务以确定他们是否满足这些目标。您应该帮助他们确定他们使用的测量是否合适。

根据我们的经验，您的客户要衡量(和发出警报)的事情中，多达一半对他们的SLO零影响。当您向他们指出这一点时，您的生活会更好，他们会关闭令人讨厌的警报。对于他们和您来说，这意味着更少的呼叫！

其余的测量值是有用的候选SLI。帮助您的客户组合这些度量以计算其SLO。

一旦开始此练习，您将很快发现SLO的某些没有被覆盖的部分-没有适当的度量值可以说出这些维度的有用信息-被“发现”了。您还应该帮助客户涵盖其SLO的这些部分。

现在，您的客户可以开始在您的平台上谈论他们的应用程序的SLO性能。

最后，与您的客户构建一组共享的SLO仪表板。您应该能够看到他们的应用程序SLO，并且应该共享与他们体验系统性能有关的任何信息。你的目标是

## 如何:与您的客户SRE

每当您的客户因为他们的SLO受到威胁而与您联系时，您不必交换太多其他信息。所有这些信息都应该在共享监控中。

#### 第3步:衡量并重新谈判

整理好测量值后，您应该收集一个月或两个月的数据。为您的客户的猛然觉醒做好准备。与他们闪亮的新SLO相比，他们认为应用程序以“五个9”(99.999%；每个人认为他们得到五个9s)运行的应用程序只能达到99.5%-99.9%。

在最初的冲击消散之后，这是个绝佳的时机，指出他们的用户并没有一直在大喊大叫，因此他们可能永远不真正需要从来没有得到的五个9。

关键问题是，他们的用户对应用程序的性能有多满意？如果他们的用户感到满意，并且没有证据表明提高性能或可用性会增加用户的采用/保留/使用率，那么您就完成了。您应该定期问自己这个问题，以确保您的预算和优先级仍然正确。(有关此主题的更深入的处理，请参阅第2章。)

如果客户认为他们仍然需要使事情变得更好，请继续下一步。

#### 第4步:设计审查和风险分析

与您的客户坐下来，真正了解他们的应用程序是如何设计和操作的。他们是否有任何隐藏的单点故障(SPOF)？他们的部署和回滚是手动的吗？基本上，请执行与内部应用程序相同的练习。

接下来，根据每个项目消耗的错误预算来对找到的问题进行排名。(有关更多操作方法的信息，请访问[Google Cloud Platform博客](#)。请注意客户选择修复的项目，以"赚回9" (例如，从99.5%变为99.9%)。您将从这些评论中学到的内容将告诉您:

- 您的客户如何使用您的平台
- 这样做时会犯什么可靠性错误
- 他们在尝试改善时会选择哪些权衡

此练习还将帮助您的客户围绕他们在当前应用程序中应经历的可靠性设定切合实际的期望。他们的期望会影响他们的看法，因此适当地设置他们只会有助于赢得并保持他们的信任。

### 第5步:练习，练习，练习

最后一步是与您的客户建立一些严格的操作。练习模拟的问题(灾难轮演，[灾难和恢复测试](#)，呼叫游戏日等)。

在团队之间建立健康的肌肉记忆，以在危机期间进行有效沟通。这是建立信任，降低MTTR并了解可作为增强功能集成到平台功能中的怪异操作边缘情况的好方法。

当确实发生事件时，不要仅仅与客户分享您的事后报告。实际进行一些[联合事后总结](#)。这样做还将建立信任，并教给您一些宝贵的经验教训。

### 深思和纪律

超过一小部分客户将很快无法执行这些步骤。请不要尝试将此模型扩展到所有人。相反，请就如何选择做出一些原则性的决定。以下是一些常用方法:

#### 收入覆盖率

选择最少的客户数量以占您收入的XX%。如果您的收入主要集中在几个大客户上，那么这可能是您的正确选择。

#### 功能覆盖

选择最少数量的客户来覆盖您平台功能的XX%以上。如果您运行高度多样化的平台，并且有很多客户在做很多不同的事情，那么这种方法将帮助您避免意外。

#### 工作量覆盖

您平台的使用情况可能由几个不同的用例或客户类型决定。在这些类型中，也许没有单个客户占主导地位，但是您可以轻松地将它们归为一组。在这种情况下，从每个同类群组中抽样一个或两个客户是获取平台覆盖范围并发现用例之间运营差异的一种好方法。

无论选择哪种方法，都请坚持下去。混合和匹配将使您的涉众困惑，并迅速使您的团队不知所措。

### 结论

在过去的几年中，SRE的职业和角色已广泛传播到Google之外。尽管我们从未期望过这一点，但是我们对此感到非常兴奋。对于我们认为该学科将如何在Google内部发展，我们也许可以说些可信的方法，但是在大千世界，这是["令人不舒服的兴奋的"提议](#)。

我们非常确定的一件事是，当您在组织中采用SRE原则时，您将遇到许多我们曾经做过的相同的拐点(有些还没有！)–包括需要在各个地方之间进一步划分界限您的客户结束了，您从哪里开始。

在这种运维深度级别上与单个客户进行互动对于我们而言是一个令人振奋的新领域，而我们仍处在前进的道路上。(您可以在线访问[Google Cloud Platform Blog](#)。但是，我们走的越远，我们就越确信这也是您需要采取的旅程。

## 第20章

### SRE团队生命周期

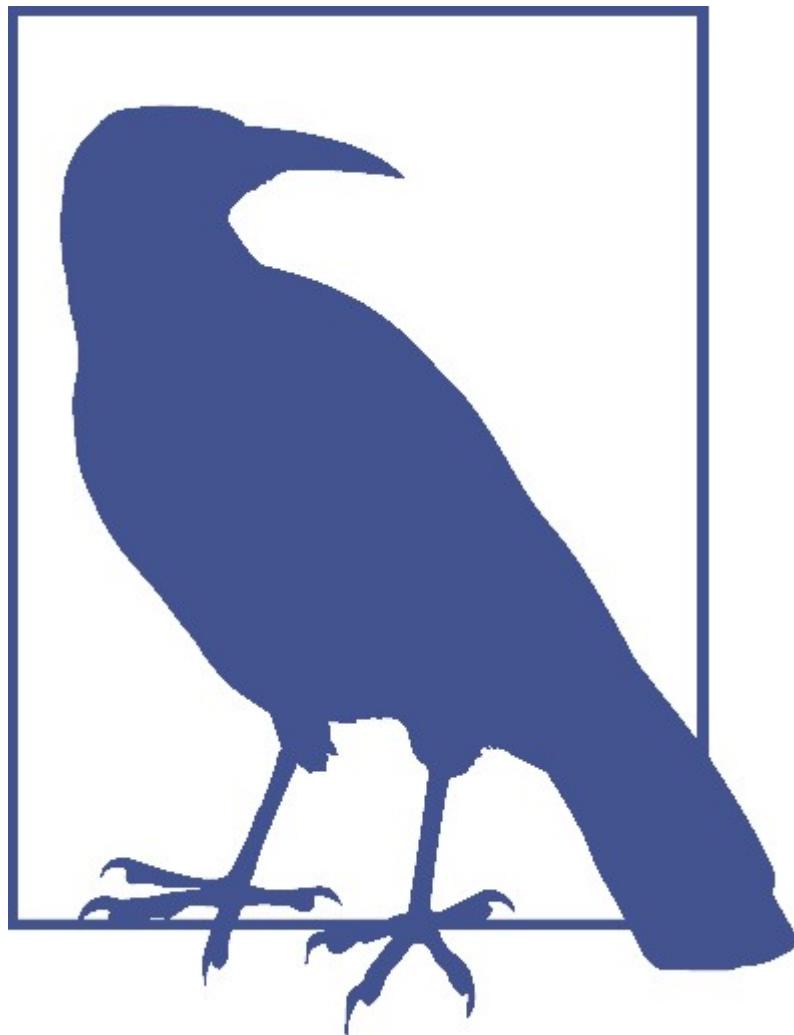
由大卫·弗格森(David Ferguson)和普拉珊特·拉汉(Prashant Labhane)与Shylaja Nukala撰写

本书的序言设定了一个目标，以“消除仅在‘Google规模’或‘Google文化’下可以实施SRE的想法。”本章列出了使SRE组织成熟的路线图，从人手不足但有抱负的人，到成熟的各个阶段，再到强大的(可能是)全球分布的SRE团队。不论您身在SRE组织中的哪个阶段，本章都将帮助您确定发展SRE组织的策略。

我们讨论了在此旅程的每个阶段都需要制定的SRE原则。虽然您自己的旅程会因组织的规模，性质和地理分布而异，但我们描述的成功应用SRE原则和实施SRE做法的路径应可推广到许多不同类型的组织。

### 没有SRE的SRE实践

即使您没有SRE，也可以通过使用SLO来采用SRE做法。如第2章所述，SLO是SRE实践的基础。因此，它们告知了我们SRE的第一个原则：



原则 #1 SRE需要具有结果的SLO。

相对于SLO的服务性能应指导您的业务决策。

我们认为，以下实践-您甚至不需要一个SRE就可以实现-是实施SRE实践的关键步骤：

- 确认您不希望100%的可靠性。
- 设定合理的SLO目标。此SLO应该衡量对用户最重要的可靠性。
- 同意将有助于捍卫用户体验的错误预算政策。使用错误预算来帮助指导：
  - 减少停机或管理使系统恢复到可靠状态的变更的战术措施
  - 对工作进行长期优先排序，以使系统更加可靠并减少错误预算的使用
- 测量SLO，并承诺遵循错误预算政策。此承诺需要公司领导的同意。

即使组织没有SRE员工，我们也认为值得为关键客户应用程序设置SLO并实施错误预算政策，这仅是因为隐含的100%SLO意味着团队只能是被动的。该SRE原理使您可以就如何确保应用程序的可靠性做出数据明智的决定。

## 开始担任SRE角色

### 查找您的第一个SRE

您的第一批SRE员工可能没有作为SRE的明确经验。我们发现以下与SRE角色相关的领域，因此适合在采访中涉及：

#### 运维

在生产环境中运行应用程序可以提供宝贵的见解，而这些见识是无法轻易获得的。

#### 软件工程

SRE需要了解他们所支持的软件，并有权对其进行改进。

#### 监控系统

SRE原则要求SLO可以衡量和解释。

#### 生产自动化

扩展操作需要自动化。

#### 系统架构

扩展应用程序需要良好的架构。

您的第一个SRE可能会在速度和可靠性目标之间处于困难和模棱两可的位置。他们将需要具有弹性和灵活性，以便在实现产品开发和捍卫客户体验之间达到适当的平衡。

### 放置您的第一个SRE

雇用了第一个SRE之后，现在需要确定将它们嵌入组织中的位置。您有三个主要选择：

- 在产品开发团队中
- 在运维团队中
- 在横向角色中，跨多个团队进行咨询

我们建议您在阅读本章后评估以下三个选项各自的优缺点，并考虑到：

#### 您自己的角色和影响范围。

如果您能够有效地影响产品开发团队，那么将SRE嵌入运维或横向角色工作中可以帮助您尽早解决烦人的生产问题。

#### 您面临的直接挑战。

如果存在需要动手工作以减轻技术问题或业务风险的挑战，那么将SRE嵌入运维或产品团队中可能会很有优势。这样做消除了组织孤岛，并促进了团队成员之间的轻松沟通。

#### 您期望在未来12个月内面临的挑战。

例如，如果您专注于发布，可以将SRE嵌入产品开发团队中。如果您专注于基础架构的变化，那么将SRE嵌入运维团队可能更有意义。

#### 您打算如何更改组织的计划。

如果您打算转变为集中式SRE组织，则可能不希望最初将SRE嵌入产品开发团队中-稍后可能很难将其从这些团队中删除。

**您确定为第一个SRE的人。**

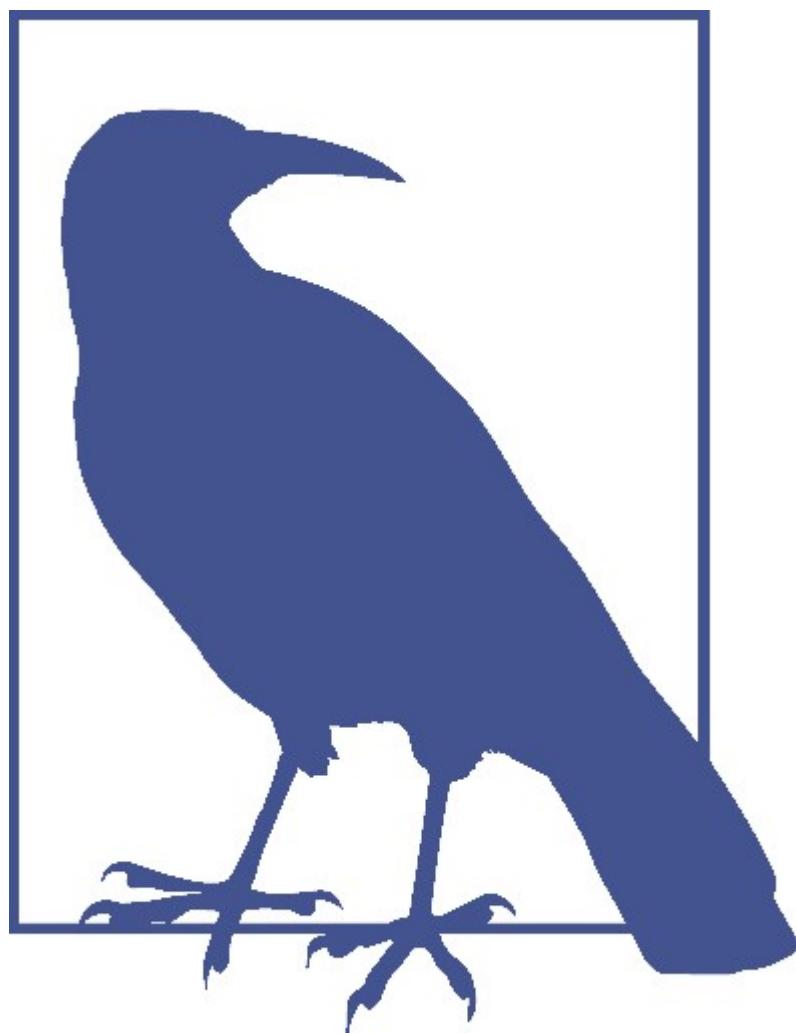
根据他们的背景和技能，决定第一个SRE在哪生产力最高。

### **开始担任SRE角色**

在确定哪种方法最适合您时，尝试使用不同的模型可能很有意义。但是，我们强烈建议您长期使用一种稳定且一致的模型；否则，不稳定会破坏SRE的有效性。

### **引导您的第一个SRE**

您的第一个SRE的初始任务是加快服务速度。为了产生积极影响，SRE需要了解服务的当前问题，所需的工作量(请参阅第6章)以及将系统保留在SLO中所需的工程技术。如果您的组织还没有遵循原则 \#1的SLO和错误预算，则您的第一个SRE需要执行设计和实施这些工具所需的工程。至此，我们的第二个SRE原则开始发挥作用：



原则 #2

SRE必须有时间使明天比今天更好。

没有这个原理，琐事只会随着服务使用的增加而增加，并且系统相应地变得更大和更复杂。在运维责任和项目工作之间保持健康的平衡至关重要-如果琐碎的工作变得繁重，有才华的工程师将逃离团队。有关SRE团队如何获得平衡的更多指导，请参阅第17章。

最初的项目工作可能集中在以下其中一项:

- 改进监控，以便在出现问题时可以更好地了解系统。
- 处理最近事后总结中确定的任何高优先级操作(请参阅第10章)。
- 实施自动化以减少运行服务所需的特定琐事。

SRE发挥独特作用，并使他们的项目使整个团队受益，这至关重要。留意SRE工作进展不佳的迹象:

- 他们的工作混合与其他工程工作没有区别。
- 如果您的第一个SRE在产品开发团队中，则他们所做的工作远远超出了其公平的运维工作份额，或者他们是唯一进行服务配置更改的人。
- SLO并没有受到重视，SRE在衡量和捍卫客户体验方面也没有取得进展。

### 分布式SRE

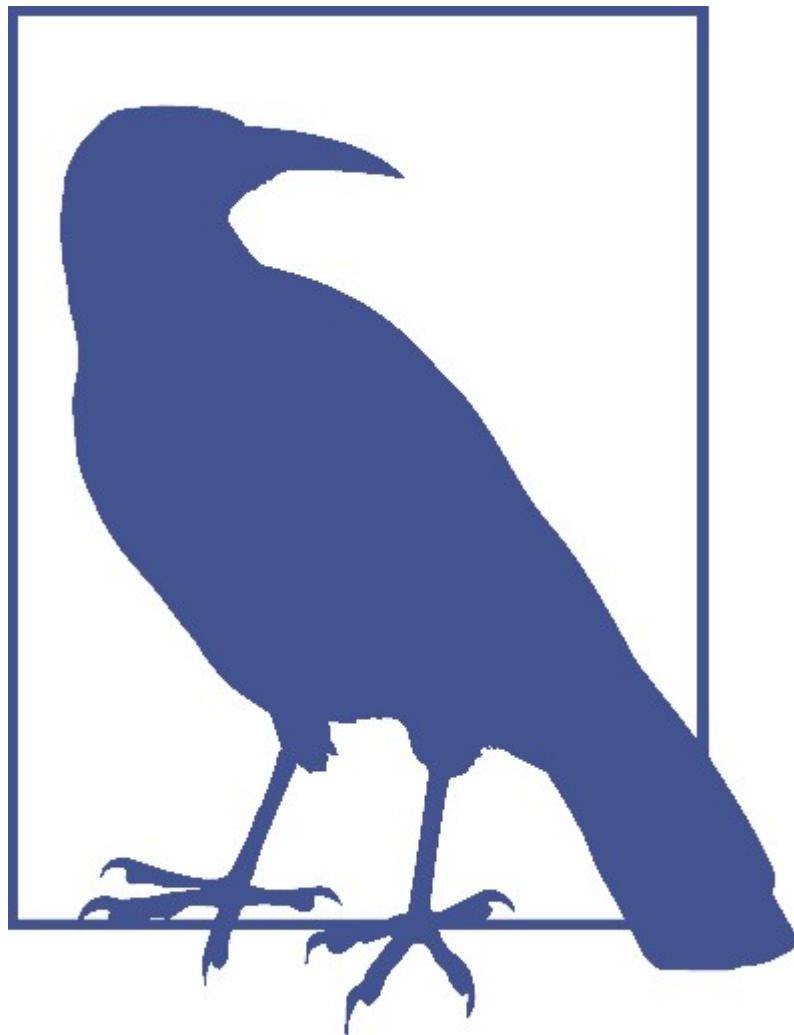
如果您的组织没有(或不打算拥有)离散的SRE团队，那么为分布式SRE构建社区很重要。该社区应倡导SRE的独特作用，并推动团队中以可靠性为中心的技术或实践的不断变化。没有社会团体，各个SRE可能会感到非常孤立。

## 您的第一个SRE团队

您可以通过多种方式组建SRE团队。从最小到最复杂，我们在Google使用的方法包括:

- 在重大项目中创建新团队
- 建立横向SRE团队
- 转换现有团队(例如，运维团队)

最适合您的组织的方法是高度受环境影响的。团队需要足够的SRE来处理运行服务所需的操作任务。解决这一工作量使我们遵循了第三项原则:



原则 #3

SRE团队有能力调节其工作量。

在大型SRE组织之外，团队从一开始可能就无法接受这个概念。该原则易于解释，可能难以在组织上付诸实践。这也是我们三项原则中最微妙的，并且经得起推敲。以下各节使用塔克曼的绩效模型以及组建，激荡期，规范和表现的阶段来逐步进行团队建设。[112](#)

## 组建

您组建的团队应具有丰富的经验和专业知识，其中包括：

- 更改应用程序软件以提高可靠性和性能。
- 编写软件至：
  - 加快发现和缓解生产中的问题。
  - 自动执行手动过程。
- 建立并使用强大的软件实践和标准来促进长期可维护性。
- 采取有条理和谨慎的方法进行业务变更：能够描述为什么某些做法可靠的原因。
- 了解系统架构(分布式系统设计和操作)。

理想情况下，您的团队将准备采用一种新的工作方式，并具有技能平衡和其他团队建立的个人关系。如果可能，我们建议您通过内部转移为团队提供种子。这样可以减少团队启动和运行所需的时间。

### **创建新团队作为重大项目的一部分**

您可能会为一个大型项目创建一个新的SRE团队，该团队的规模足以证明新员工的数量，并且已将可靠性和运维能力确定为项目风险。例如，可能包括创建新服务或技术发生重大变化(例如，迁移到公共云)。

### **组建水平的SRE团队**

通过这种方法(在第一本书的[第27章](#)中有充分记录)，由SRE组成的小型团队会跨多个团队进行咨询。该团队可能还会为配置管理，监控和警报建立最佳实践和工具。

### **将团队转换到位**

您也许可以将现有团队转换为SRE团队。现有团队可能不是产品开发团队。典型的候选人包括运维团队或负责管理组织大量使用的流行开源组件的团队。请注意避免在未先应用SRE惯例和原则的情况下将团队从"运维"重命名为"SRE"！如果您的品牌重塑工作失败，那么将来您的组织可能会完全不顾SRE的整个概念。

## **震荡期**

组建后，团队需要开始协作:团队成员需要彼此之间以及与其他团队一起良好地工作。

您可以采用多种策略来促进这种凝聚力。在Google，我们已经成功地提供了一个定期的论坛来学习和讨论SRE的实践，并反思团队的表现。例如，您可能会定期举行电视午餐，在那里您会播放SREcon的视频或读书俱乐部，在那里您都预读了一些相关内容，然后讨论了如何应用它。

在此阶段中，鼓励您的新SRE团队进行自我扩展。您的新SRE应该轻松地讲出组织中不适合的SRE做法，以及是否值得进行更改以使其适合。

### **风险和缓解措施**

在SRE旅程的初期阶段，团队可能会采用多种方式失败。接下来，我们介绍一些风险和可能的缓解策略，并按新团队的组成方式对其进行细分。对于每种风险，您可以使用一种或多种缓解策略。

### **新团队是重大项目的一部分**

#### **风险**

团队:

- 一次承担太多服务的责任，使自己的业务分散。
  - 一支不断战斗的团队没有时间以更持久的方式应对风险。
- 过于内省地试图理解SRE原理以及如何实现它们。结果，它的交付不足。
  - 例如，在开发完美的SLO定义时，团队可能会变得精疲力尽，同时忽略了服务的需求。
- 没有彻底检查其工作。结果，服务管理恢复为以前的行为。
  - 团队每天被呼叫100次。由于呼叫并不表示需要立即干预，因此它们会忽略页面。
- 放弃SRE原则和实践以达到产品里程碑。
  - 捍卫SLO的可靠性改进(例如体系结构更改)可能永远不会实现，因为它们会延迟开发时间表。
- 由于与现有团队的冲突而分心，他们认为新的SRE团队会失去影响力或力量。
- 没有必要的技能范围，因此只能提供部分必要的改进。
  - 如果没有编程能力，SRE可能无法对产品进行仪器测量以测量其可靠性。

**缓解措施**

团队:

- 最初从事一项重要服务。
- 尽早参与项目，最好是在设计阶段。
- 参与设计，特别侧重于定义SLO和分析设计固有的可靠性风险。
- 与产品开发团队合作，致力于特定于可靠性的功能以及与现有运维平台的集成。
- 预计第一天不承担运维责任。相反，此责任最初由产品开发团队或项目团队负责。这可能是重大的文化变革，需要管理层的支持。
- 在SRE启用服务之前必须达成明确的协议(请参阅*Site Reliability Engineering*的[第32章](#))。

另外:

- 如果项目涉及迁移，则团队应该对当前和将来的环境有深入的了解。如果您需要从外部招聘团队成员，请考虑具有软件工程和未来环境知识的候选人。
- 继续将新员工人数保持在不到团队的三分之一，以使培训工作不会淹没现有团队成员。

**横向SRE团队**

### 风险

团队被认为是一个新的“召唤”组织，没有任何实际工作或没有任何实际价值。

### 缓解措施

团队：

- 拥有具有相关主题专业知识的受人尊敬的工程师。
- 进行专注于交付工具(用于监控，警报，部署，最佳实践，清单)的项目工作。这些工具应至少对另外两个团队产生短期的有益影响。
- 交流成功和利益。应该赞扬一个SRE团队，该团队实现了效率突破，自动完成琐事或永久消除了系统不可靠性的根源。
- 将自己视为推动者，而不是看门人。关注解决方案，而不仅仅是问题。

## 一支已转换的团队

### 风险

团队：

- 意识到随着自动化取代人类，转换过程是缓慢失业的开始。
- 不支持到SRE团队的变更。
- 没有平衡的人力来改变团队的日常活动。
- 几个月后，他们的日常工作没有任何好处。
- 运维不支持脚本或自动化的系统。
- 没有软件工程技能来自动化他们当前的工作量。
- 并非一贯具有发展至SRE所需的技能，或对获得技能的兴趣。

### 缓解措施

团队:

- 获得高层领导对变更的支持。
- 重新协商责任以创造实现变更所需的人力。
- 非常仔细地管理变更的沟通。
- 在整个过渡过程中都能获得强大的个人和技术支持。
- 应对失业问题。在许多环境中，自动化消除了部分工作，但不是整个工作。尽管这可能是迈向失业的一步，但它的确至少可以腾出时间来做比非自动劳动更好的事情(并且对未来的雇主更容易销售)。
- 可以避免操作过载并产生更大的影响。如果工程师减少了琐事的工作量而需要一个较小的团队，那么他们的经验应该可以在组织中的其他地方高度复用。如果他们的经验不能在内部使用，则应该在其他地方找工作提供优势。
- 接受培训以获取SRE所需的技能。您的产品开发团队可以提供产品培训，而SRE指导可以利用本书和其他外部资源。
- 更改绩效评估方式-评估团队和个人的指标。前者应与SLO保持一致，并采用其他SRE做法；后者应与SRE技能的证据保持一致。
- 将经验丰富的SRE或开发人员添加到团队中。
- 可以自由(预算或时间)识别并引入新的开源或基于云的监控和警报系统以实现自动化。确定现有系统是否足够应该是当务之急。
- 定期在内部并与利益相关者一起审查进度。

## 规范化

规范需要解决第405页"风险和缓解"中提出的问题，并就组织SRE团队的最佳实践达成广泛共识。团队需要就可接受的琐事水平，适当的警报阈值以及重要和相关的SRE做法达成共识。团队还需要自给自足，以主动识别服务之前的挑战，并设定中期和长期目标以改善服务。

在规范阶段，团队应达到以下成熟度级别:

- 已制定SLO和错误预算，并在发生重大事件后执行错误预算政策。领导层对SLO度量很感兴趣。
- 值班轮换是可持续的(请参阅第8章)。值班工程师的通话时间得到补偿。有足够的工具，[文档<sup>113</sup>](#)和培训可在重要工单工作中为任何团队成员提供支持。
- 琐事要被记录，有边界和被管理。结果，SRE完成了具有影响力项目，从而提高了可靠性和效率。
- 事后总结文化已经建立。(请参阅第10章。)
- 该小组展示了第1章中列出的大多数原则。
- 当团队解决第405页"存储"中列出的初始问题时，他们会捕获所学内容并防止重复出现问题。该团队定期进行培训练习，例如不幸之轮或DiRT(灾难恢复测试)。(有关呼叫培训的更多信息，请参见第一本书的[第11章](#)和本书的第18章。)
- 产品开发团队将从继续参与值班轮换中受益。

- 团队为利益相关者生成定期报告(例如，季度报告)，涵盖报告期间的重点，不足之处和关键指标。

## 将现有团队转变为SRE团队

由《纽约时报》的Brian Balser撰写

当《纽约时报》成立其交付和站点可靠性工程部门时，我们由具有SRE类型技能的工程师组建了SRE团队，例如工具和运维生产系统团队。有些团队是“有潜力的”:他们在人才，远见和责任方面考虑了SRE。其他团队已经存在了几年，由于技能，兴趣和机会的结合，最终运行了生产架构。

### 挑战

过渡到SRE的现有团队之一处于非常具有挑战性的位置。多年来，该团队已拥有所有权和责任来管理配置，更改请求以及我们整个站点范围体系结构的核心组件的操作。他们有效地成为了支持我们所有产品开发团队的服务团队。他们的工作受到故障单和生产问题的驱动，并且处于连续反应模式。他们没有时间进行改进，创新或其他高价值的战略工作。

尽管团队有很多好主意，但工作量很大，而且通常与产品发布相关的许多高优先级“阻止程序”服务请求也是如此。这种模式是不可持续的，因此团队需要与产品线性增长一同成长以跟上这种支持负担。为了加剧这种情况，这支小团队多年来积累了丰富的机构知识。庞杂的信息对团队进行了大量的打扰，加剧了团队的负担，[公共要素](#)笼罩着团队。

### 以第一原则为基础

我们的SRE组织的一项指导原则是使自己摆脱关键的道路，并赋予产品开发团队自助服务解决方案的能力。考虑到这一点，我们的目标变得很明确:反转责任模型，使产品开发团队能够推动自己的变革。此策略将:

- 加快交货速度。
- SRE无需管理配置变更，从而可以对整个系统进行真正的改进。

### 流程改进

我们通过几个变更阶段改进了流程:

1. 我们在开发团队中嵌入了SRE，以帮助缓解压力。
2. 为了使产品开发团队能够独立获取其服务配置的所有权，我们将每种服务配置分为一个基于团队的存储库。
3. 我们将每项服务从传统CI系统迁移到我们的标准Drone CI/CD管道。开发人员友好的工作流程完全由GitHub事件驱动。
4. 我们将每个产品团队加入了新的工具和工作流程，以便他们可以提交自己的变更请求，而不会受到服务单的阻塞。

尽管这些改进是向前迈出的一大步，但我们尚未达到理想的最终状态。审核拉取请求仍然经常需要SRE专业知识。为了使耗时的审核中断更易于管理，我们安排了每天的办公时间。这种一致的做法使我们能够以更具预测性的方式来批处理问题和讨论，还为与正在加入的团队共享知识提供了场所。

### 最终结果和后续步骤

现在，SRE团队正在实现其最初目标:> 50%的项目工作(与支持相关的工作相比)。团队仍然拥有丰富的机构知识，但是现在知识正在更广泛地传播，从而逐渐改善了公共因素并减少了中断。

现在我们已经为项目工作腾出了空间，我们的下一步将集中在添加更多高级功能上，例如金丝雀部署，更好的测试工具以及可观察性和弹性功能。这样做将使产品开发团队更加自信地对其服务配置行使完全自主权，而无需依赖SRE进行变更管理。与您的产品开发团队建立健康的关系是许多缓解策略的基础。团队应根据您组织的计划周期来计划工作。

在继续下一步之前:停下来，庆祝这次成功，并写一篇回顾您迄今为止的旅程的回顾。

### 表现

SRE团队到现在为止的生产和工作经验，应该赢得了更广泛组织的尊重和关注，并为战略性前进奠定了基础。在塔克曼绩效模型的最后阶段，表现，您应该期望:

**所有架构设计和变更的合作伙伴。**

从最初的设计阶段开始，SRE应该定义用于构建和构造软件以确保可靠性的模式。

**具有完整的工作量自决。**

团队应始终遵循原则3，以确保系统的整体健康。

### 在架构层合作

产品开发团队应开始与合作伙伴SRE团队联系，以寻求有关所有重大服务变更的建议。SRE团队现在有机会发挥一些最大的影响。

例如，SRE团队可能会为新服务架构的设计提供早期投入，以减少日后进行高成本重新设计的可能性。产品开发和SRE团队可以承认他们在体系结构决策方面的差异，以达到良好的设计过程。成功的参与可以通过以下方式增加价值:

- 改善了可靠性，可扩展性和可操作性
- 更好地重用现有模式
- 更简单的迁移(如果需要)

### 自我调节的工作量

随着时间的流逝，体系结构上的伙伴关系应该有机地融合在一起，而SRE团队则必须向其伙伴明确提出原则 # 3。这样做需要强大的团队领导能力和高级管理层的明确，前期承诺。调节自己的工作负载的能力确保了SRE团队作为工程团队的地位，该团队可以使用与组织产品开发团队同等的组织最重要的服务。

实际上，SRE团队如何确定自己的工作量取决于SRE与之交互的团队。在Google，SRE团队通常与独特的产品开发团队互动。在这种情况下，该关系具有以下特征:

- SRE团队选择是否以及何时启用服务(请参阅Site Reliability Engineering的第32章)。
- 如果出现运营超负荷的情况，团队可以通过以下方式减少工作量:

--- 降低SLO

--- 将运维工作转移到另一个团队(例如，产品开发团队)

- 如果在商定的劳力约束内无法在SLO上操作服务，则SRE团队可以将服务移交  
给产品开发团队。

- SRE参与不是永久的-它通过大规模解决问题并提高服务的可靠性来满足自  
己。如果SRE团队为服务解决了所有此类问题，则您需要:

--- 有目的地考虑SRE团队需要解决的其他可靠性挑战。

--- 做出有意的决定，将服务移交给产品开发团队。

否则，随着SRE转向更多有趣的机会，您的团队将面临人员流失的风险。摩擦  
造成的缓慢流失可能使生产面临风险。

并非所有SRE团队都有合作伙伴产品开发团队。一些SRE团队还负责开发其运行的  
系统。一些SRE团队打包第三方软件，硬件或服务(例如，开源软件包，网络设  
备，即服务)，并将这些资产转化为内部服务。在这种情况下，您将无法选择将工  
作转移回另一个团队。相反，请考虑以下策略:

- 如果服务不符合其SLO，请停止与功能相关的项目工作，而转而关注可靠性的  
项目工作。
- 如果在商定的琐事约束内无法在SLO上运维服务，请减少您的SLO，除非管理  
人员提供更多的能力(人员或基础架构)来处理这种情况。

### 组成更多SRE团队

一旦第一个SRE团队启动并运行，您可能想组建一个额外的SRE团队。您可能由于  
以下原因之一而这样做:

#### 服务复杂度

随着服务获得用户和功能，单个SRE团队无法有效地支持它变得越来越复杂  
和困难。您可能需要将团队划分为专门负责服务部分的子团队。

#### SRE推广

如果您的第一个SRE团队已经成功并取得了明显的成就，那么在更多服务中  
采用这种方法可能会对组织产生兴趣。

#### 按地理位置划分

您想将团队分成两个不同的时区，并转移到12小时的值班时间。

在创建新的SRE团队时，我们建议您执行以下操作:

- 阅读其他团队成立后写的任何事后总结报告。识别并重复进行得很好的事情，  
并针对不顺利的事情进行修复和探索。
- 从现有团队中为SRE注入新团队-可能会挑战的一些最佳SRE和潜力最大的  
SRE。根据我们的经验，很难找到合格的SRE候选人，因此快速发展新员工团  
队通常是不现实的。
- 标准化建立团队和入职服务的框架(请参阅第18章)。
- 缓慢更改值班职责。例如:

--- 为了避免突然失去熟练的值班工程师，请在过渡期内让团队成员在其先前团队的系统中处于值班状态。

--- 团队分割后，请等待三至六个月以分割值班轮换。

### 服务复杂度

#### 哪里拆分

如果一项服务变得过于复杂而无法由一个团队进行管理，则可以采用多种方法来拆分工作。考虑以下选项以简化团队成员的认知负担：

#### 基础设施分离

例如，计算，存储和网络；前端和后端；前端和数据库；客户端和服务器；前端和管道。

#### 语言拆分

SRE原则不依赖于编程语言。但是，如果您的SRE深入地参与了源代码，那么按照这些思路进行拆分可能会带来一些好处。

#### 位置拆分

如果您组织的工程跨越多个办公室，则可能需要使SRE团队位置与应用程序开发保持一致。

#### 陷阱

当团队拆分时，有时新团队都不会对原团队拥有的组件负责。为了减轻这种风险，您可以：

- 指定一个小组负责第二小组章程未涵盖的所有事务。
- 在两个团队中任命高级SRE担任总体技术负责人。

#### SRE推广

如果最初的SRE团队成功，则您的组织可能需要更多。我们建议您仔细确定获得SRE支持的服务的优先级。请考虑以下几点：

- 优先考虑可靠性对财务或声誉有重大影响的服务。影响越大，优先级越高。
- 定义为使产品正常运行所需的最少可行服务集。确定这些服务的优先级，并确保其他服务正常降级。
- 仅仅因为服务不可靠，不应将服务作为SRE的优先事项。SRE应该在战术上与业务最相关地应用。您也不想让开发人员在使用SRE之前忽略可靠性。

#### 地理上分离

正如我们第一本书的[第11章](#)中所述，Google通常会为不同大陆的姐妹SRE团队配备人员。我们这样做的原因有很多：

#### 服务可靠性

如果重大事件(例如自然灾害)阻止一个团队运作，则另一个团队可以继续为服务提供支持。

#### 值班压力

将值班轮换分为12小时轮班，以便为待命工程师提供适当的休息时间。

### 招聘和留住人才

与正常工作日重叠的值班时间扩大了我们可以招聘到SRE职位的工程师的基础，并强调了我们职位的工程部分。

### 生产成熟度

随着对文档，培训和标准化的需求变得越来越重要，将服务责任划分给两个办公室往往会导致成熟度的提高。

如果您的组织很幸运能够在多个大洲拥有工程团队，我们建议为多站点SRE团队配备人员。可以将SRE团队与开发团队设在不同的办公室，但是根据我们的经验，集中办公可以通过健康而强大的团队间对话的形式带来好处。否则，SRE很难了解服务的发展方式或技术基础架构的使用方式，产品开发人员对基础架构的改进持乐观态度也很难。

### 布局:团队应该相隔几个时区？

假设您有选择的余地，则时区分离是确定两支队伍的位置时的重要考虑因素。不幸的是，目标是互斥的：

- 尽量减少值班者在正常办公时间以外必须工作的时间
- 最大化两个团队在线时的重叠时间，以便他们可以定期相互交流

夏令时使情况变得复杂。

根据我们的经验，时区间隔为六到八个小时的工作人员团队可以很好地工作，并且避免了凌晨12点至早上6点的值班时间。您可以使用<https://www.timeanddate.com/worldclock/meeting.html>等在线资源来可视化各个位置的时区重叠。

### 人员和项目:为团队播种

当您按地理位置划分团队时，新办公室中的第一个SRE团队将为以后的SRE团队设定规范。如果您可以确定一个或多个愿意暂时或长期从原始站点迁移以建立SRE实践并招募和培训新团队的SRE，则成功的可能性会更高。新团队还应执行一项高价值的项目，该项目应促进团队内部的协作，并需要与其姊妹团队进行互动。

### 平衡:在办公室之间分配工作并避免"夜班"

通常，两个姐妹SRE团队之一与产品开发团队(或称为"办公室1")一起(至少在同一时区)。如果是这种情况，请保持警惕，以确保不在同一地点的团队("Office 2")不会成为夜班，而该夜班与产品开发团队的联系很少，所付出的辛苦超过其应有的工作份额，或者仅分配给那些不太有趣或影响较小的项目。这两个办公室的工作量会有一些自然的差异：

- 您的服务可能每天都有高峰，在该高峰期间将有一个办公室待命。结果，两个站点的值班体验将有所不同。
- 您的开发过程将产生具有特定节奏的新版本。一个办公室可能会承担与部署和回滚相关的更多负担。
- Office 1更可能在工作日被产品开发团队的问题打扰。

- Office 1更容易进行与主要版本相关的项目工作。相反，Office 2更容易进行与近期产品目标分离的项目工作。

您可以通过以下做法帮助保持平衡:

- 平衡办公室之间的值班负载。指定故障单百分比较高办公室承接更低比例的呼叫数。
- 将开发区域与特定办公室中的SRE团队相关联。这可以是短期的(例如，根据项目)或长期的(例如，根据服务)。否则，产品开发团队可能会依靠Office 1，而不能有效地与Office 2中的SRE接触。
- 将较高比例的内部服务改进项目(可能需要较少的产品开发团队参与)分配给Office 2。
- 在两个办公室之间公平地传播最有趣，最有影响力项目。
- 在两个办公室之间保持相似的团队规模和资历组合。
- 在两个站点之间划分项目，以故意促进SRE之间的部门间交互。虽然从一个办公室运行一个主要项目可能会提高效率，但将项目分布在两个站点之间既有助于传播知识，也可以在办公室之间建立信任。
- 允许工程师定期前往其他办公室。这样可以建立更好的融洽关系，并因此愿意为另一方做好工作。

### 布局:三班倒怎么办？

我们试图将SRE团队分散到三个地点的尝试导致了各种问题:

- 不可能召开所有SRE都能参加的部门间生产会议(请参阅第一本书的[第31章](#))。
- 很难确保三个办公室之间知识，能力和运维响应的均等。
- 如果所有的值班职责仅在办公时间内进行，则没有动力去激励低层次的工作和低价值的呼叫。在办公时间内，成为解决简单问题的英雄很有趣。但是，如果有一定的个人成本，那么确保它不再发生的动机是敏锐而直接的。

### 时间选择:团队的两半都应该同时开始吗？

您可以使用以下任一模型来组建姐妹团队:

- 两个半部同时开始。
- 首先设置与产品开发团队位于同一地点的站点。这使SRE可以更早地参与产品生命周期。
- 首先设置与产品开发团队不在同一地点的站点，或者，如果某项服务已经投入生产已有一段时间，则SRE团队和产品开发团队可以共享值班。
- 根据合适的人在正确的时间开始进行更改。

### 经费:差旅预算

为团队的两半之间的高质量互动创造机会非常重要。尽管视频会议在日常会议中非常有效，但我们发现，定期的面对面互动对于促进健康的关系和信任大有帮助。我们建议:

- 站点1中的每个SRE，产品开发经理和技术主管每年(至少)每年访问站点2，反之亦然。
- 在站点1中担任管理或技术领导角色的每个SRE每年至少两次访问站点2，反之亦然。
- 所有SRE每年至少召开一次会议。

#### 领导力:共同的服务所有权

如果您有多个SRE站点，则每个办公室中可能都有决策者。这些各方应定期面对面并通过视频会议开会。只有建立牢固的个人关系，他们才能:

- 讨论团队面临的挑战的解决方案。
- 解决意见分歧，并商定共同前进的道路。
- 代表彼此的团队提倡(以防止"我们与他们对抗"的心态)。
- 支持彼此团队的健康。

## 建议管理多个团队的做法

随着您的组织积累更多的SRE和SRE团队，出现了新的挑战。例如，您必须:

- 确保为SRE提供他们所需的职业机会。
- 鼓励实践和工具的一致性。
- 处理无法证明完全参与SRE的服务。

本节介绍了Google在处理这些问题时采用的许多做法。根据您组织的具体情况，一些或许也可能为您工作。

#### 任务控制

Google的Mission Control计划使来自产品开发团队的工程师有机会在SRE团队中度过六个月的时间。我们通常将这些工程师与SRE团队相匹配，他们的工作范围与他们的专业知识截然不同。该软件工程师接受了生产系统和实践方面的培训，并最终为该服务上线。六个月后，一些工程师决定留在SRE。其他人则回到他们的旧团队，对生产环境和SRE实践有了更好的理解。SRE团队可以从其他工程资源中受益，并且可以深入了解培训材料和文档中的差距和不准确性。

#### SRE交换

Google的SRE Exchange计划允许SRE与其他SRE团队一起工作一个星期。来访的SRE观察宿主团队的工作方式，并分享其归属团队的实践，这可能对宿主团队有用。在交流结束时，来访的SRE编写了旅行报告，描述了他们的星期，观察结果以及对两个团队的建议。该计划对Google很有用，因为我们的SRE团队非常专业。

#### 培训

培训对于SRE操作系统的能力建至关重要。尽管大多数都是以团队形式提供的(请参阅第8章第150页的"培训路线图")，但请考虑为所有SRE建立标准的培训课程。在Google，所有新的SRE都会参加SRE EDU，这是一个为期一周的沉浸式培训，其中介绍了几乎所有SRE都可以使用的关键概念，工具和平台。这提供了所有新SRE

的基础知识水平，并简化了特定于团队和特定于服务的培训目标。几个月后，SRE EDU团队还将举办第二系列的课程，涵盖了我们用于管理重大事件的通用工具和流程。我们的绩效管理流程特别认可促进培训的SRE。

### 横向项目

由于SRE团队与一组服务紧密结合，因此团队倾向于构建专有的解决方案以应对他们所遇到的挑战-例如，监控，软件推广和配置工具。这可能导致团队之间的工作大量重复。虽然允许多种解决方案竞争"市场"有价值，但在某些时候，融合到以下标准解决方案是有意义的：

- 符合大多数团队的要求
- 提供稳定且可扩展的平台，可在其上构建下一层创新

Google通过使用水平团队来开展这些工作，水平团队通常由经验丰富的SRE组成。这些水平团队建立并运行标准解决方案，并与其他SRE团队合作以确保顺利采用。(有关横向软件开发的更多信息，请参阅"案例研究2:第21页第432页，"SRE中的通用工具采用"。)

### SRE移动性

Google尽最大努力确保工程师积极参与他们各自的团队。为此，我们确保SRE能够(并且能够知道)在团队之间转移。假设没有性能问题，SRE可以自由转移到其他开放招聘的SRE团队。还通过了我们的软件工程师职位招聘标准的SRE可自由转移到产品开发团队(请参阅<http://bit.ly/2xyQ4aD>)。

出于多种原因，这种移动性对于个人和团队而言非常健康：

- 工程师能够确定并占据感兴趣的角色。
- 如果个人情况发生变化并且值班职责变得不切实际，SRE可以在要求更少的值班职责的团队中探索机会。他们可以通过与其他团队交谈并查看团队的值班统计信息来获取此信息。
- 在团队之间移动的SRE扩大了他们加入的团队的经验。
- 在办公室之间移动的SRE帮助建立或保持不同办公室之间的文化一致性。
- SRE不会被迫从事不健康的服务或为不支持个人发展的经理而工作。

此策略还具有使您的SRE经理专注于健康快乐的服务和团队的副作用。

### 旅行

除了维持地理位置分散的团队健康所需的差旅费用外(请参阅"财务:"旅行预算"(第417页)，请考虑为以下项目提供资金：

- 建立内部感兴趣的公司社区，其中包括来自多个办事处的SRE。这些小组可以通过电子邮件和视频会议进行大量合作，但至少每年进行一次面对面的会面。
- 参加并参加全行业的SRE和SRE相关会议，以扩大知识，了解其他组织如何解决类似问题，并希望受到鼓舞和激发活力。

### 启动协调工程团队

正如我们第一本书的[第27章](#)中所述，启动协调工程(LCE)团队可以将SRE原理应用于更广泛的产品开发团队-即不需要值得SRE参与关注的构建服务的团队。就像其他SRE团队一样，LCE团队也应积极参与使其日常操作自动化。例如，开发标准工具和框架使产品开发团队能够在生产环境中设计，构建和启动其服务。

### 卓越生产

随着组织中SRE团队数量的增长，将会出现许多最佳实践。每个SRE团队的发展都不同，因此评估他们需要高级SRE并具有对多个团队的洞察力。

在Google，我们会进行定期的服务审核，称为“卓越生产”。SRE高级领导者会定期审查每个SRE团队，并根据许多标准措施(例如，值班负载，错误预算使用，项目完成，错误关闭率)对他们进行评估。审查既赞扬了出色的表现，又为表现欠佳的团队提供了建议。

经验丰富的SRE可以评估细微的场景。例如，要弄清由于团队合并或团队分裂而造成的项目完成率下降与真正的团队绩效问题相比可能具有挑战性。如果团队有不堪重负的风险，那么审阅者可以并且应该利用其组织地位来支持团队的领导者来纠正这种情况。

### SRE资金和招聘

在Google，我们采用两种做法来确保每个SRE都贡献可观的价值：

- SRE的大部分资金与产品开发团队的资金来自同一来源。类似于测试或安全性，可靠性是产品开发的核心支柱，并为此提供资金。
- 根据我们的经验，SRE的供应数量总是小于对它们的需求。这种动态机制确保我们定期审查获得SRE支持的服务并确定其优先级。

简而言之，您的SRE应该少于组织想要的数量，并且只有足够的SRE才能完成其专门工作。

在Google，产品开发团队中SRE与工程师的比例从大约1:5(例如，底层基础设施服务)到大约1:50(例如，具有大量使用标准框架构建的面向消费者的应用程序)。许多服务以大约1:10的比率处于此范围的中间。

### 结论

我们认为，任何规模的组织都可以通过应用以下三个原则来实施SRE实践：

- SRE需要具有结果的SLO。
- SRE必须有时间使明天比今天更好。
- SRE团队有能力调节其工作量。

自Google开始公开谈论SRE以来，它已从Google特定的生产实践发展为许多公司所从事的职业。这些原则通常被证明是正确的-在我们多年的大规模直接经验中，以及在我们最近与客户合作采用SRE做法的经验中。因为我们已经看到这些做法在Google内部和外部都可以使用，所以我们认为这些建议对于各种类型和规模的组织都非常有用。

...

<sup>112</sup>. 布鲁斯·塔克曼(Bruce W. Tuckman)，"小组中的发展顺序"，《心理公报》 \* 63，第1期。6(1965): 384--99。↔

<sup>113</sup>. Shylaja Nukala和Vivek Rau，"为什么SRE文件很重要"，ACM队列(2018年5月至6月): 即将出版。↔

## 第21章

### SRE中的组织变更管理

由Alex Bramley , Ben Lutch , Michelle Duffy和Nir Tarcic与Betsy Beyer撰写

在[第一本SRE书简介](#)中Ben Treynor Sloss将SRE团队描述为"以快速创新和对变革的广泛接受为特征"，并将组织变革管理指定为SRE团队的核心职责。本章探讨了理论如何在SRE团队中实践应用。在回顾了一些关键的变更管理理论之后，我们探索了两个案例研究，这些案例研究说明了如何以具体方式在Google发挥不同的变更管理风格。

注意，术语"变更管理"具有两种解释：组织变更管理和变更控制。本章将变更管理视为所有用于准备和支持个人，团队和业务部门进行组织变更的方法的统称。我们不会在项目管理环境中讨论此术语，因为在该术语中它可能用于指代变更控制过程，例如变更审查或版本控制。

## SRE 拥抱 变革

2,000多年前，希腊哲学家赫拉克利特(Heraclitus)宣称变化是唯一不变的。这种公理在今天仍然适用，尤其是在技术方面，尤其是在快速发展的互联网和云领域。

产品团队的存在是为了制造产品，交付功能并取悦客户。在Google，采用"启动和迭代"的方法后，大多数变更都是快节奏的。执行此类更改通常需要跨系统，产品和全球分布的团队进行协调。站点可靠性工程师经常身处这种复杂且瞬息万变的环境之中，负责平衡变更固有的风险与产品可靠性和可用性。错误预算(请参阅第2章)是实现这种平衡的主要机制。

## 变革管理简介

自1940年代Kurt Lewin在该领域开展基础工作以来，变革管理已成为研究和实践领域。理论主要集中于开发用于管理组织变更的框架。对特定理论的深入分析超出了本书的范围，但是为了在SRE领域中对它们进行情境化，我们简要介绍一些常见的理论以及每种理论如何适用于SRE类型的组织。尽管这些理论框架中隐含的正式流程尚未由Google的SRE应用，但通过这些框架的角度考虑SRE活动有助于我们改进了管理变更的方法。在讨论之后，我们将介绍一些案例研究，以说明其中一些理论的要素如何应用于由Google SRE领导的变更管理活动。

### Lewin的三阶段模型

用于管理变更的[库尔特·勒温\(Kurt Lewin\)](#)的"解冻-更改-冻结"模型是该领域中最古老的相关理论。这个简单的三阶段模型是用于管理流程审查以及由此产生的组动态变化的工具。第1阶段需要说服一个小组认为必须进行更改。一旦他们适应变更的想法，第二阶段便执行该变更。最后，当变更大致完成后，第3阶段将行为和思想的

新模式制度化。该模型的核心原理将小组视为主要的动态工具，认为当小组计划，执行和完成任何变更期间时，应将个人和小组的交互作为一个系统进行检查。因此，Lewin的工作对于在宏观层面计划组织变革最为有用。

### 麦肯锡的7-S模型

麦肯锡的七个S代表结构，策略，系统，技能，风格，员工和共同价值观。与Lewin的工作类似，此框架也是计划中的组织变革的工具集。尽管Lewin的框架是通用的，但7-S的明确目标是提高组织效率。两种理论的应用都始于对当前目的和过程的分析。但是，7-S还明确涵盖业务元素(结构，策略，系统)和人员管理元素(共享价值，技能，风格，员工)。该模型对于考虑从传统系统管理重点向更全面的站点可靠性工程方法转变的团队可能有用。

### Kotter领导变革的八步程序

《时间》杂志将约翰·P·科特(John P. Kotter)的1996年著作《领导变革》(哈佛商学院出版社)命名为“有史以来前25名最具影响力的商业管理书籍之一”。图21-1描述了Kotter变更管理过程中的八个步骤。



图21-1.Kotter的变更管理模型(来源:<https://www.kotterinc.com/8-steps-process-for-leading-change/>)

Kotter的流程与SRE团队和组织特别相关，但有一个小例外:在许多情况下(例如，即将进行的Waze案例研究)，无需产生紧迫感。支持产品和系统快速增长的SRE团队经常面临紧迫的规模，可靠性和运维挑战。组件系统通常由多个开发团队拥有，这些开发团队可能跨越多个组织部门。扩展问题还可能需要与从物理基础架构到产品经理的团队进行协调。由于发生问题时SRE通常位于前线，因此它有独特的动机来领导进行必要的更改以确保产品24/7/365可用。SRE的许多工作(隐式地)包含Kotter的过程，以确保受支持产品的持续可用性。

### Prosci ADKAR模型

[Prosci ADKAR模型](#)专注于平衡变更管理的业务和人员方面。ADKAR是个人必须达到的-实现成功的组织变革:意识，欲望，知识，能力和强化-的目标的缩写。

原则上，ADKAR提供了一个有用的，周到的，以人为本的框架。但是，由于操作责任通常会施加相当大的时间限制，因此它在SRE中的适用性受到限制。在ADKAR的各个阶段中逐步进行迭代并提供必要的培训或指导，需要在交流方面进行调整和投入，而这在全球分布，以运维为中心的团队中很难实现。也就是说，Google已成功使用ADKAR风格的流程来引入和构建对高层变更的支持-例如，在SRE管理团队中引入全球组织变更，同时保留实施细节的本地自治权。

### 基于情感的模型

[桥梁过渡模型](#)描述了人们对变化的情感反应。虽然它是人员管理的有用管理工具，但它不是变更管理的框架或过程。同样，库伯勒-罗斯变化曲线描述了人们面对变化时可能会感觉到的各种情绪。它是由伊丽莎白·库伯勒·罗斯(Elisabeth Kübler-Ross)对死亡和垂死的研究发展而来，[114](#)已用于理解和预测员工对组织变革的反应。两种模型都可用于在整个变更期间保持较高的员工生产率，因为不高兴的人很少有生产力。

### Deming周期

统计人员Edward W.Deming的这一过程也称为计划-行动-检查-表现(或PDCA)周期，通常在DevOps环境中用于改进过程-例如，采用持续集成/持续交付技术。它不适合组织变更管理，因为它不涵盖变更的人为方面，包括动机和领导风格。Deming的重点是采用现有流程(机械，自动化或工作流)并周期性地应用持续改进。我们在本章中引用的案例研究处理的是较大的组织变革，而迭代会适得其反:频繁，棘手的组织结构图变革会削弱员工的信心，并对公司文化产生负面影响。

### 这些理论如何适用于SRE

没有变更管理模型可以普遍适用于每种情况，因此Google SRE并未专门针对一种模型进行标准化就不足为奇了。就是说，这就是我们想将这些模型应用于SRE中的常见变更管理方案的方式:

- Kotter的“八步流程”是针对SRE团队的变更管理模型，他们必须将变更作为核心责任。
- Prosci ADKAR 模型是SRE管理层可能需要考虑的框架，以协调全球分布的团队之间的变更。
- 所有个体SRE经理都将受益于*Bridges Transition Model*和*Kübler-Ross Change Curve*，它们在组织变革时为员工提供支持的工具。

现在，我们已经介绍了这些理论，下面让我们看两个案例研究，这些案例研究显示了变更管理在Google中的表现。

## 案例研究1:缩放Waze-从临时到计划变革

### 背景

Waze是Google于2013年收购的基于社区的导航应用。收购之后，Waze进入了活跃用户，工程人员和计算基础架构显着增长的时期，但继续在Google内部相对自主地运作。增长带来了许多技术和组织方面的挑战。

Waze的自治权和创业精神促使他们以小组工程师的基层技术回应来应对这些挑战，而不是上一节中讨论的正式模型所隐含的管理化，结构化的组织变革。但是，他们在整个组织和基础架构中传播变更的方法与Kotter的变更管理模型非常相似。本案例研究考察了Kotter的流程(我们追溯适用)如何恰当地描述Waze在收购后增长时面临的一系列技术和组织挑战。

#### 消息队列:在保持可靠性的同时更换系统

Kotter的模型以“紧迫感”开始变革的周期。当Waze消息队列系统的可靠性严重下降，导致越来越频繁且严重的中断时，Waze的SRE团队需要迅速果断地采取行动。如图21-2所示，消息排队系统对操作至关重要，因为Waze的每个组件(实时，地理编码，路由等)都使用它与内部的其他组件进行通信。

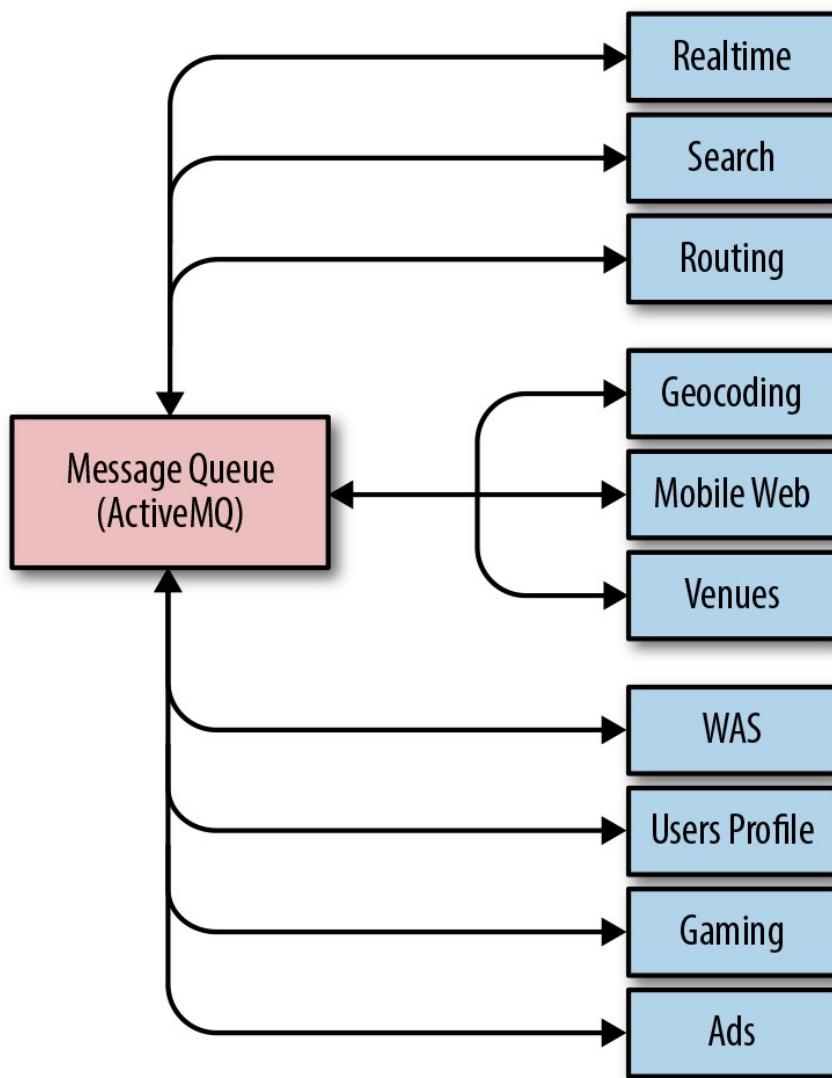


图21-2.Waze组件之间的通信路径

随着消息队列中吞吐量的显着增长，系统根本无法满足日益增长的需求。SRE需要手动干预以在越来越短的间隔内保持系统稳定性。在最糟糕的情况下，整个Waze SRE团队花了整整两周的24/7消防时间，最终每小时采取一次重新启动消息队列的某些组件的方式，以保持消息的畅通和数千万用户的快乐。

由于SRE还负责构建和发布Waze的所有软件，因此该操作负载对功能速度产生了显著影响-当SRE花费全部时间扑灭火灾时，他们几乎没有时间支持新功能的发布。通过强调这种情况的严重性，工程师说服Waze的领导重新评估优先级，并花费一些工程时间进行可靠性工作。由两个SRE组成的指导联盟和一名高级工程师共同形成了对未来的战略构想，该未来将不再需要SRE琐事来保持消息畅通。这个小型团队评估了现成的消息队列产品，但很快决定，使用定制解决方案只能满足Waze的扩展性和可靠性要求。

如果没有某种方式同时维护操作，则内部开发此消息队列将是不可能的。联盟从使用当前消息队列的团队中招募了一支由开发人员组成的“志愿军”，从而消除了这种“行动壁垒”。每个团队都检查了其服务的代码库，以确定减少其发布消息量的方法。整理不必要的消息并在旧队列之上部署压缩层可减少系统的某些负载。通过为负责30%以上系统流量的一个特定组件建立专用的消息传递队列，该团队还获得了更多的操作喘息空间。这些措施暂时搁置了足够的时间，以便有两个月的时间来组装和测试新消息系统的原型。

即使没有即将发生的服务崩溃的压力，迁移每秒处理数万条消息的消息队列系统也是一项艰巨的任务。但是逐渐减少旧系统上的负载将减轻一些压力，从而为团队提供更长的时间来完成迁移。为此，Waze SRE重建了用于消息队列的客户端库，以便它们可以使用集中控制界面切换流量来使用一个或两个系统发布和接收消息。

一旦新系统被证明可以正常工作，SRE就开始了迁移的第一阶段：他们确定了一些低流量，高重要性的消息流，这些消息流造成了消息中断。对于这些流，写入两个消息传递系统将提供备用路径。几近失误，在旧系统步履蹒跚的情况下，备份路径使核心Waze服务保持运行，提供了短期获胜，证明了最初的投资合理。

大规模迁移到新系统需要SRE与使用它的团队紧密合作。团队需要弄清楚如何最好地支持其用例以及如何协调流量切换。由于SRE团队自动化了迁移流量的过程，并且默认情况下新系统支持更多的用例，因此迁移速度“显着加快了”。

Kotter的变更管理流程以“发起变更”结束。最终，在采用新系统的背后有足够的动力，SRE团队可以宣布旧系统已弃用并且不再受支持。他们在几个季度后迁移了最后一批流浪者。如今，新系统处理的负载是上一个系统的1000倍以上，并且几乎不需要SRE的人工干预即可提供持续的支持和维护。

### 下一轮变革：改善部署流程

作为“周期”的变化过程是Kotter的关键见解之一。当涉及到SRE面临的技术变更类型时，有意义的变更的周期性本质尤其明显。消除系统中的一个瓶颈通常会突出另一个问题。随着每个变更周期的完成，最终的改进，标准化和自动化将节省工程时间。工程团队现在可以更仔细地检查他们的系统并确定更多的痛点，从而触发下一轮变革。

当Waze SRE最终可以从与消息传递系统相关的消防问题中退后一步时，出现了新的瓶颈，并带来了新的“紧迫感”：SRE对发行版的唯一所有权明显并严重阻碍了开发速度。发布的手动性质需要大量的SRE时间。更糟的是，本来已经很不理想的情况更加严重，系统组件很大，而且由于发行成本很高，因此相对来说很少。结果，每个发行版都代表一个较大的增量，从而大大增加了主要缺陷需要回滚的可能性。

由于Waze SRE没有第一阶段的总体规划，因此逐步改善了发布流程。为了简化系统组件，以便团队可以更快地进行迭代，一位Waze高级开发人员创建了一个用于构建微服务的框架。这提供了一个标准的“包含电池”平台，使工程组织可以轻松地将其组件拆开。SRE与该开发人员合作，加入了一些以可靠性为中心的功能-例

如，一个通用的控制界面和一系列适合自动化的行为。因此，SRE可以开发一套工具来管理发布过程中以前昂贵的部分。其中一种工具通过捆绑使用框架创建新的微服务所需的所有步骤来激励采用。

这些工具起初是很简单的-最初的原型是由一个SRE在几天的时间内构建的。随着团队从其父组件中分离出更多微服务，SRE开发的工具的价值很快就为整个组织所认识。SRE花费更少的时间将精简的组件投入生产，并且新的微服务单独发布的成本更低。

尽管发布过程已经得到了很大的改善，但是新的微服务的激增意味着SRE的整体负担仍然令人担忧。在发布的负担减轻之前，工程领导者不愿对发布过程承担责任。

作为回应，一小部分SRE和开发人员的联盟制定了一个[战略构想]，以使用开源，多云，连续交付平台[Spinnaker](#)转向持续部署策略。用于构建和执行部署工作流。借助我们的引导工具节省的时间，该团队现在能够设计这个新系统，以实现一键构建和部署成百上千个微服务的功能。新系统在技术上在各个方面都优于以前的系统，但是SRE仍然不能说服开发团队进行转换。这种勉强是由两个因素造成的：明显的抑制因素是必须将自己的发行版推向生产，以及由于对发布过程的可见性较差而导致的变更厌恶。

Waze SRE通过展示新流程如何增加价值来消除这些"接受障碍"。该团队构建了一个集中式仪表板，该仪表板显示了二进制文件的发布状态以及微服务框架导出的许多标准指标。开发团队可以轻松地将其发布与这些指标的更改关联起来，这使他们对部署成功感到充满信心。SRE与一些面向志愿系统的开发团队紧密合作，将服务移至[Spinnaker](#)。这些"胜利"证明了新系统不仅可以满足其要求，而且可以在原始发行过程之外增加价值。此时，工程领导者为所有团队设定了使用新的[Spinnaker](#)部署管道执行发布的目标。

为了促进迁移，Waze SRE为具有复杂需求的团队提供了整个组织的大三角帆培训课程和咨询课程。当早期采用者熟悉新系统时，他们的积极经验引发了"加速采用"的连锁反应。他们发现新流程比等待SRE推动发布更快，更轻松。现在，工程师开始对没有移动的依赖项施加压力，因为它们阻碍了更快的开发速度，而不是SRE团队！

如今，Waze超过95%的服务都使用[Spinnaker](#)进行连续部署，并且几乎无需人工干预就可以将变更推向生产。尽管[Spinnaker](#)并非万能解决方案，但如果使用微服务框架构建新服务，则配置发布管道就变得微不足道了，因此新服务强烈希望以此解决方案标准化。

### 经验教训

Waze在消除技术变更瓶颈方面的经验为其他团队提供了许多有益的经验教训，这些团队正在尝试以工程为主导的技术或组织变更。首先，变更管理理论并不浪费时间！通过Kotter的过程查看此开发和迁移过程，证明了该模型的适用性。当时对Kotter模型进行更正式的应用可能有助于简化和指导变更过程。

从基层发起的变革需要SRE与开发之间的紧密合作，以及高层领导的支持。与来自组织各个部门(SRE，开发人员和管理人员)的成员建立一个专注的小型小组，是团队成功的关键。类似的合作对于发起变更至关重要。随着时间的流逝，这些临时小组可以而且应该发展成更加正式和结构化的合作，在这些合作中，SRE会自动参与设计讨论，并可以为在整个产品生命周期中的生产环境中构建和部署健壮的应用程序的最佳实践提供建议。

增量更改更易于管理。直接跳到“完美”解决方案的步子太大了，无法一次完成所有操作(更不用说如果您的系统即将崩溃，这可能是行不通的)，并且“完美”的概念很可能会随着新信息的出现而在变更过程中演变。迭代方法可以证明早期的成功，这可以帮助组织理解变更的愿景并证明进一步的投资合理性。另一方面，如果早期的迭代没有显示出价值，那么当您不可避免地放弃更改时，您将浪费更少的时间和更少的资源。由于增量更改不会一次全部发生，因此拥有总体规划非常宝贵。广义地描述目标，保持灵活性，并确保每次迭代都朝着目标迈进。

最后，有时您当前的解决方案不能满足您的战略远景要求。建造新的东西会花费大量的工程成本，但是如果项目使您脱离当地的最大局限并实现长期增长，那将是值得的。作为一项思想实验，找出随着业务和组织在未来几年内增长，系统和工具中可能出现瓶颈的地方。如果您怀疑任何元素都无法横向扩展，或者相对于核心业务指标(例如每日活跃用户)没有超线性(或更糟糕的是，指数增长)，则可能需要考虑重新设计或替换它们。

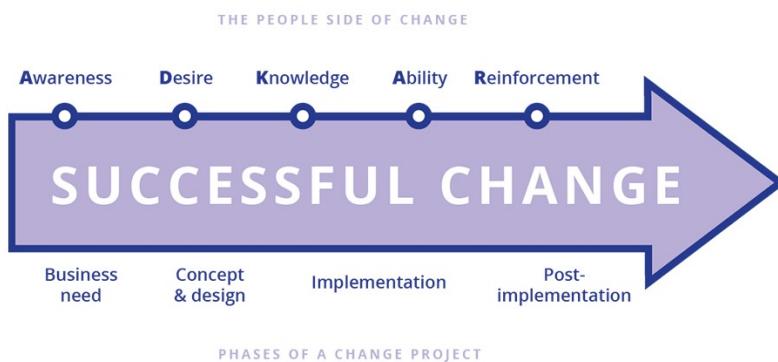
Waze对新的内部消息队列系统的开发表明，一小组坚定的工程师有可能进行更改，从而使针的服务可靠性更高。将Kotter的模型映射到变更上可以看出，即使在小型的，以工程为主导的组织中，对变更管理策略的一些考虑也可以为成功提供一个公式。而且，正如下一个案例研究还表明的那样，当变革促进标准化的技术和流程时，整个组织就可以收获可观的效率提升。

## 案例研究2：在SRE中采用通用工具

### 背景

SRE对他们可以并且应该用来管理生产的软件持谨慎态度。多年的经验，观察行之有效的方法以及从事后总结的角度审视过去的经验，为SRE提供了深厚的背景和强大的直觉。尤其，构建和实施软件以自动完成今年的工作是SRE的核心价值。特别是Google SRE最近将我们的工作重点放在了横向软件上。大量用户和开发人员采用相同的解决方案会产生良性循环，并减少对轮子的重新发明。否则可能无法交互的团队共享使用同一软件自动执行的实践和策略。

此案例研究基于组织的演变，而不是对系统扩展或可靠性问题的回应(如Waze案例研究中所述)。因此，Prosci ADKAR模型(如图21-3所示)比Kotter模型更适合，因为它可以识别变更过程中组织/人员管理的明确特征和技术考虑因素。



© Prosci Inc. All Rights Reserved.

图21-3. Prosci ADKAR 变更管理模型

### 问题陈述

几年前，Google SRE发现自己使用了多个独立的软件解决方案来解决多个问题空间中的大致相同的问题：监控，发布和部署，事件响应，容量管理等。产生这种最终状态的部分原因是，用于SRE的人员构建工具与他们的用户及其需求无关。工具开发人员并不总是对问题陈述或整个生产情况有最新的了解，因为随着几乎每天都有新的软件，硬件和用例投入使用，生产环境以新的方式迅速变化。此外，工具的消费者也各不相同，有时有正交的需求（“此发布必须快速；近似即可。”与“此发布必须100%正确；可以缓慢进行”）。

结果，这些长期项目都无法完全满足任何人的需求，并且每个项目都具有不同水平的开发工作，功能完整性和持续的支持。那些等待大型用例（一种非特定的，未来的载歌载舞的解决方案）的人等待了很长时间，感到沮丧，并使用自己的软件工程技能来创建自己的小环境解决方案。那些具有较小特定需求的人不愿采用并非针对他们的广泛解决方案。更加通用的解决方案的长期，技术和组织上的好处是显而易见的，但是客户，服务和团队因为等待并没有雇佣人员或采取奖励措施。为了使这种情况更加复杂，大型和小型客户团队的需求都随时间而变化。

### 我们决定要做什么

为了将这种情况视为一个具体的问题空间，我们问自己：如果所有的Google SRE都可以使用一个通用的监控引擎和一组仪表板，它们易于使用并支持各种用例而无需自定义，该怎么办？

同样，我们可以将这种思维模式扩展到发布和推出，事件响应，容量管理等等。如果产品的初始配置能够广泛满足各种功能需求，那么随着时间的流逝，我们通用且信息灵通的解决方案将不可避免。在某个时候，与生产交互的工程师的临界数量将超过他们使用的，自行选择的解决方案，从而无法迁移到通用的，得到良好支持的工具和自动化集，从而放弃了定制工具和相关的维护成本。

Google的SRE很幸运，它的许多工程师都有软件工程背景和经验。鼓励像专家这样对特定问题有深刻见解的工程师，从负载均衡到部署工具，再到事件管理和响应，要像一个虚拟团队一样工作，自发的具有长期视野，这是第一步，这是一个很自然的第一步。这些工程师将他们的愿景转化为可以工作的，真正的软件，这些软件最终将被SRE以及整个Google所采用，作为生产的基本功能。

为了返回到用于变更管理的ADKAR模型，到目前为止讨论的步骤（识别问题并确认机会）是ADKAR启动“意识”步骤的教科书示例。Google SRE领导团队同意需求（desire），并且具有足够的知识和能力，可以相当快地着手设计解决方案。

### 设计

我们的首要任务是集中讨论一些我们认为很重要的主题，这将受益于一致的愿景：提供适合大多数用例的解决方案和采用计划。从65个以上的拟议项目清单开始，我们花了几个月的时间收集客户需求，验证路线图并进行市场分析，最终将我们的工作范围限定在少数经过审查的主题上。

我们的初始设计围绕这些主题创建了一个由SRE专家组成的虚拟团队。这个虚拟团队将把大部分时间（约80%）用于这些横向项目。80%的时间和一个虚拟团队背后的想法是确保我们在与生产保持持续联系的情况下设计或构建解决方案。但是，我们（也许可以预见）通过这种方法发现了一些痛点：

- 协调虚拟团队非常困难，因为虚拟团队的工作重点是经常在多个时区待命，因此很难集中精力。在运行服务和构建严肃的软件之间可以交换很多状态。
- 从收集共识到代码审查的所有内容都受到缺少中心位置和公共时间的影响。
- 横向项目的人员最初必须来自现有团队，而现在他们的工程资源越来越少，无法解决自己的项目。即使在Google，在委派人员以支持系统与委派人员以构建具有前瞻性的基础架构之间也存在紧张关系。

有了足够的数据，我们意识到我们需要重新设计方法，并选择了更为熟悉的集中式模型。最重要的是，我们删除了要求团队成员将他们的时间80/20分配在项目工作和值班职责之间的要求。现在，大多数SRE软件开发工作都是由少数具有丰富值班经验，但头脑冷静，专注于根据这些经验来构建软件的高级工程师完成的。我们还通过招募或调动工程师将这些团队中的许多人集中起来。在一个房间内进行小组(6-10人)开发效率更高(但是，这种说法并不适用于所有小组，例如远程SRE团队)。通过视频会议，电子邮件和良好的老式旅行，我们仍然可以实现在整个Google工程组织中收集需求和观点的目标。

因此，我们的设计演变实际上是在一个熟悉的地方-小型，敏捷，大多是本地的，快速发展的团队-进行，但是更加强调选择和构建自动化工具，以供60%的Google工程师采用(我们确定的数字是对“几乎Google的所有人”目标的合理解释)。成功意味着大多数Google都在使用SRE构建的内容来管理其生产环境。

ADKAR模型在以人为中心的知识和能力之间映射了变更项目的实施阶段。此案例研究证明了这种映射。我们有许多敬业，有才华和知识渊博的工程师，但是我们要让专注于SRE问题的人员通过关注客户需求，产品路线图和交付承诺来像产品软件开发工程师一样工作。我们需要重新审视此更改的实施，以使工程师能够展示他们对这些新特性的能力。

### 实施:监控

回到上一节中提到的监控空间，第一本SRE书中的[第31章](#)描述了Viceroy--Google SRE如何创建适用于以下方面的单个监控仪表板解决方案:每个人-解决了不同的自定义解决方案的问题。几个SRE团队共同创建和运行了初始迭代，随着Viceroy逐渐成为Google的事实上的监控和仪表板解决方案，一个专门的集中式SRE开发团队承担了该项目的所有权。

但是，即使Viceroy框架将SRE统一在一个通用框架下，当团队针对他们的服务构建复杂的自定义仪表板时，也要付出很多重复的努力。Viceroy提供了一种标准的托管方法来设计和构建数据的可视显示，但仍然需要每个团队决定要显示的数据以及如何组织数据。

现在集中的软件开发团队开始了第二个并行工作，以提供通用的仪表板，在较低级别的“定制”系统之上构建了一个自以为是的零配置系统。该零配置系统基于给定服务是以几种流行样式之一进行组织的假设，提供了一组标准的全面监控显示。随着时间的推移，大多数服务都迁移到使用这些标准仪表板，而不是投资自定义布局。大型，独特或其他特殊服务仍可以根据需要在托管系统中部署自定义视图。

回到ADKAR模型，Google的监控工具整合始于基层工作，由此带来的运维效率改进为开展更广泛的工作提供了可量化的基础([意识和愿望](#)):SRE自资成立了一个软件开发团队，以为整个Google建立生产管理工具。

### 经验教训

设计相互依赖的零件的迁移通常比白板上设计复杂。但是在现实生活中，最艰巨的工程工作最终是将许多小型/受约束的系统演变为更少，更通用的系统-而不打扰许多客户所依赖的已经运行的服务。同时，除现有系统外，还添加了新的小型系统-其中一些最终通过扩展为大型系统而使我们感到惊讶。从大的设计重新开始有一个智力上的吸引力，只有真正必要的回到约束中，但是到目前为止，迁移系统和团队是最困难的工作。

设计横向软件需要大量听取潜在最终用户的意見，并且在许多方面，构建和采用的任务看起来很像产品经理的角色。为了使这项工作取得成功，我们必须确保我们吸收并确定了优先事项。满足SRE和其他生产用户的客户需求，也是成功的关键要素。重要的是要认识到向通用工具的转变仍在进行中。我们对构建共享技术的团队的结构和人员进行了迭代，以更好地满足客户需求，并增加了产品管理和用户体验人才(解决了缺失的"知识")。

在过去的一两年中，我们在Google的众多团队中看到了这些由SRE设计和制造的产品的使用。我们已经了解到，要获得成功，相对于新的通用解决方案的净收益，迁移成本(从较旧，分散但专业的解决方案中)需要很小。否则，迁移本身将成为采用的障碍。我们将继续与构建这些产品的各个团队合作，以"增强"使团队提供的通用解决方案使客户满意的行为。

我们在横向软件开发项目中发现的一个共同主题是，无论新软件和产品的质量如何，迁移成本(从已经运行的新产品迁移到新产品)始终被认为非常高。尽管具备易于管理且缺乏专门深度知识的诱人之处，但从熟悉的环境(包括所有累赘和琐事)中迁移出来的成本通常是一个障碍。此外，个别工程师经常有类似的内部独白:"我不是在改进或更改系统；我是在将一个工作换成另一个工作。"ADKAR将这种抵制描述为"知识与能力的鸿沟"。在人的方面，为了认识和拥抱变化，人们需要时间，指导和培训新工具和技能。在技术方面，实施更改需要了解采用成本，并包括在启动过程中将这些成本降至最低的工作。

结果，团队的迁移成本几乎为零("只需重新编译，您就可以使用new \$thing")，并且需要向团队，个人，和公司明确收益("现在您已收到\$foo漏洞的保护")。

SRE通常用于以"尽力而为"的方式来构建我们承诺的产品，这意味着我们将产品投入的时间与我们正在做的其他一切工作(管理主要服务，容量计划，处理中断，等等。)。结果，我们的执行不是很可靠。无法预测功能或服务何时可用。通过扩展，我们的产品的消费者对最终结果的信任度较低，因为它被永久地拖延了，并且产品经理和个人工程师轮流换人。当单个SRE或SRE团队构建供自己使用的工具时，重点是解决单个问题以减少维护受支持系统的SLO的成本。为了在Google上为大多数用例构建通用工具，我们需要将重点转移到衡量产品采用方面这项工作是否成功。

由于我们的组织文化和丰富的资源，我们以自下而上而不是自上而下的方式进行此项目。我们没有强制用户迁移到我们的新监控系统，而是通过证明我们的新产品比现有解决方案更好来争取用户。

随着时间的流逝，我们了解到，我们如何进行开发过程会告知潜在的内部用户如何看待最终结果。仅当由具有生产经验的工程师100%致力于构建软件，时间表和支持与Google其余软件开发相同时，这些项目才真正吸引人。透明地构建通用软件，例如clockwork，良好的通信性("我们将在Y日期之前完成X个工作")，大大提高了向新系统的迁移速度。人们已经信任新系统，因为他们可以从早期观察到它是如何开发的。\*对香肠制作方式的认识比我们一开始所期望的要重要得多。我们最初的想法

法是"如果您打造出了不起的东西，那么人们自然会蜂拥而至"。相反，这些项目必须明确定义，事先做好广告，针对众多用户案例(首先针对最脾气暴躁的采用者)进行评估，跨越式发展要比现有方案更好，并且可以毫不费力地采用。

您拥有更多用于通用工具和采用工具的消费者，您实际上需要花费更多的时间在编写代码以外的事情上。回想起来，这听起来似乎很明显，但最终目标明确，日期可信，定期更新以及与消费者保持经常联系至关重要。常常持怀疑态度的消费者会问:"如果我当前的一次性shell脚本可以正常工作，我真的需要吗?"采用通用软件或流程类似于将可靠性作为一项功能-您可以构建世界上最好的东西，但是如果人们不采用它(或者如果不靠谱就不能使用它)，那么它对任何人就没有用。制定采用计划-从champions到Beta测试人员再到执行赞助商再到专门的了解最小化采用障碍重要性的工程师-既是构建和采用通用工具的最终目标，也是起点。

这是因为采用会带来网络效应:随着通用软件工具的规模和范围的扩大，对这些工具的渐进式改进对组织而言更有价值。随着工具价值的增加，专用于它们的开发工作也趋于增加。这些开发工作中的一部分自然会朝着进一步降低迁移成本，鼓励更多采用的方向发展。广泛采用鼓励以一致的，类似于产品的方式建立组织范围内的改进，并证明需要配备完整的团队来长期支持这些工具。这些工具应具有快速开发，功能稳定，通用控制界面和可自动化API的特点。

在衡量此类努力的影响时，我们可以提出类似以下的问题:

- 新产品开发人员可以多快构建和管理世界范围的服务？
- 在通用工具和实践的支持下，一个域中的SRE如何轻松迁移到另一个域？
- 可以使用相同的原语来管理多少项服务，如端到端用户体验与单独的服务？

这些都是衡量影响的可能且极有价值的方法，但是我们的第一个衡量标准必须是采用。

## 结论

正如Waze和横向软件案例研究所证明的那样，即使在一家公司内部，SRE变更管理也可能需要解决各种问题空间和组织环境。结果，可能没有单一的正式变更管理模型可以完美地应用于任何给定组织可能解决的变更范围。但是，这些框架，尤其是Kotter的八步流程和Prosci ADKAR模型，可以为应对变化提供有用的见解。在像SRE一样动态的环境中，所有必要更改的共同点是不断进行重新评估和迭代。尽管许多变更可能是从基层开始的，但随着变更的成熟，大多数变更都可以从结构化的协调和计划中受益。

<sup>114</sup>. 伊丽莎白·库伯勒·罗斯(Elisabeth Kübler-Ross)，*关于死亡和垂死：垂死者必须教医生，护士，神职人员及其家属*(纽约: Scribner，1969)。 ↪

# 总结

由戴夫·伦森(Dave Rensin)，贝茜·拜尔(Betsy Beyer)，尼尔·理查德·墨菲(Niall Richard Murphy)

斯蒂芬·索恩(Stephen Thorne)和肯特·川原(Kent Kawahara)撰写

## 一路向前...

美国经济学家埃德加·菲德勒(Edgar Fiedler)曾经说过“靠水晶球生活的人很快就会学会吃毛玻璃了”。预测未来很危险。

尽管如此，我们仍将冒着碎玻璃的危险，并尝试对本书之后的内容说几句有用的话。

## 未来属于过去

自第一本SRE书出版以来，我们的经验以及该书的编纂过程清楚地表明了大小企业对SRE的巨大积压需求。由此，出现了一些有趣的观察。

首先，大型企业倾向于以健壮的方式采取SRE做法。在这种情况下，大型企业变化缓慢具有相似的文化基因肯定是不正确的。我们希望明年这些公司在SRE领域看到很多有趣的创新，这确实令人兴奋。

接下来，规模较小的公司正在寻找采用SRE做法的方法，无论他们是否可以配备一支遍布全球的成熟SRE团队。我们长期以来一直在猜测，尽管菜肴的实质和顺序至关重要，但人们不必一次吃掉整个SRE餐。现在，我们看到该概念已付诸实践，这真的很令人兴奋，因为它允许每个人都参与。

最后，如果您正在阅读本书，并且想知道您是否有市场机会提供服务或制造可帮助企业采用SRE的产品，那么答案是肯定的。实际上，如果您决定这样做，请通过bookquestions@oreilly.com上的O'Reilly告诉我们。我们想跟上您的进度。

## SRE + \

我们从许多有思想的人那里听说，SRE的原则和实践似乎也应该适用于其他学科，尤其是安全性。我们在第一本书(或本书)中没有涉及SRE/安全性的重叠，但这显然是一個迅速发展的重点领域。如果您想知道为什么我们在本卷中没有深入介绍它，那是因为我们不确定我们要说些什么(尚未)。

在撰写本文时(2018年初春季)，我们已经看到DevSecOps一词的出现-市场认识到开发，安全性和操作都是相互依赖的。如果您是SRE，正在寻找一个有用且未经探索的空间来度过一段时间，那么这是一个非常不错的选择！

一般而言，我们认为这是向“如何将SRE应用到\？毫无疑问，某些SRE原则和实践可能在其他领域具有价值。我们很高兴看到结果...

## 细流，溪流和洪水

提供对此内容进行内容贡献的人数比以前的人数要多得多。这似乎很容易解释-第一本书很受欢迎，所以人们渴望为第二本书做出贡献。但是，提供各种服务的人的“多样性”很有趣。我们评估的内容提案的类型和范围令我们感到惊讶。例如，一个律师团队就如何将错误预算纳入非技术性法律协议与我们联系。SRE如何应用于法律专业是本书显然不涉及的主题，但对于其他所有专业人员而言，可能确实很有趣。

由于种类繁多，我们决定本卷将不仅仅是上一本书的实现伴侣。这也将是全新的SRE相关内容流的起点。这意味着更多的书籍，文章，播客，视频，还有谁知道。此卷是开始，而不是结束。

### SRE属于我们所有人

当我们开始撰写第一本书时，我们的主要动机是解释SRE，这是我们做的一件有趣的事，目的是使Google更好地为用户服务。传播这些知识似乎很有用。

在我们热情地解释我们对世界的看法时，我们立足于我们的直接经验。结果，我们无意中疏远了较大的DevOps社区中的一些人，他们认为我们忽略了其他组织对该领域的贡献。

本卷旨在通过花费大量时间讨论DevOps和SRE以及它们为何不矛盾的方法来纠正该错误。

令我们感到非常高兴的是，SRE是一个不断发展的从业者社区，如今它已经扩展到Google之外。实际上，到您阅读本文时，非Google SRE的数量很可能会大大超过Google SRE的数量。

无论其历史如何，SRE现在都是一个全球社区，而Google就是其中的一员。它属于我们所有人，这是一件好事。通过诸如SREcon，meetups之类的会议和其他出版物，我们已经非常清楚地知道，我们每个人都有很多可以分享和相互学习的地方。为此，我们希望本书能够进一步扩展正在进行的对话。

### 感激

这本书一直是我们的挚爱之作，对于您决定阅读它，我们深表感谢。这个过程为我们每个人提供了一些令人惊讶的经验教训，说明SRE已经成为什么，并使我们所有人都非常高兴看到下一步的发展。

(由于我们是SRE，因此我们将对此过程进行漫长的事后总结。谁知道，也许您甚至会在这些天之一的博客中看到它……)

希望您的查询继续进行，而传呼机保持沉默。---戴夫，贝茜，尼尔，斯蒂芬和肯特

## 附录A SLO文档示例

本文介绍了示例游戏服务的SLO。

| Status        | Published       |
|---------------|-----------------|
| Author        | Steven Thurgood |
| Date          | 2018-02-19      |
| Reviewers     | David Ferguson  |
| Approvers     | Betsy Beyer     |
| Approval Date | 2018-02-20      |
| Revisit Date  | 2019-02-01      |

### 服务概述

示例游戏服务允许Android和iPhone用户彼此玩游戏。该应用程序在用户的手机上运行，并且移动通过REST API发送回API。数据存储区包含所有当前和先前游戏的状态。得分管道读取该表并生成今天，本周和所有时间的最新联赛表。排行榜结果可在应用程序中通过API获得，也可以在公共HTTP服务器上获得。

SLO使用四个星期的滚动窗口。

### SLI和SLO

| Category     | SLI                                                                                                                                                                                                                                                                                                                                                                                                                                         | SLO                                                  |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| API          |                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                      |
| Availability | <p>The proportion of successful requests, as measured from the load balancer metrics. Any HTTP status other than 500–599 is considered successful.</p> <pre>count of "api" http_requests which do not have a 5XX status code divided by count of all "api" http_requests</pre>                                                                                                                                                              | 97% success                                          |
| 延迟           | <p>The proportion of sufficiently fast requests, as measured from the load balancer metrics.</p> <p>"Sufficiently fast" is defined as &lt; 400 ms, or &lt; 850 ms. count of "api" http_requests with a duration less than or equal to "0.4" seconds divided by count of all "api" http_requests</p><br><p>count of "api" http_requests with a duration less than or equal to "0.85" seconds divided by count of all "api" http_requests</p> | 90% of requests < 400 ms<br>99% of requests < 850 ms |
| HTTP server  |                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                      |
| 可用性          | <p>根据负载平衡器指标衡量的成功请求的比例。除500–599以外的任何HTTP状态均被视为成功。</p> <pre>count of "web" http_requests which do not have a 5XX status code divided by count of all "web" http_requests</pre>                                                                                                                                                                                                                                                               | 99%                                                  |

| Category       | SLI                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | SLO                                                                                                                              |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Latency        | <p>The proportion of sufficiently fast requests, as measured from the load balancer metrics.</p> <p>“Sufficiently fast” is defined as &lt; 200 ms, or &lt; 1,000 ms.</p> <pre>count of "web" http_requests with a duration less than or equal to "0.2" seconds divided by count of all "web" http_requests</pre><br><pre>count of "web" http_requests with a duration less than or equal to "1.0" seconds divided by count of all "web" http_requests</pre>                                                | <p>90% of requests &lt; 200 ms</p> <p>99% of requests &lt; 1,000 ms</p>                                                          |
| Score pipeline |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                  |
| Freshness      | <p>The proportion of records read from the league table that were updated recently. “Recently” is defined as within 1 minute, or within 10 minutes.</p> <p>Uses metrics from the API and HTTP server:</p> <pre>count of all data_requests for "api" and "web" with freshness less than or equal to 1 minute divided by count of all data_requests</pre><br><pre>count of all data_requests for "api" and "web" with freshness less than or equal to 10 minutes divided by count of all data_requests</pre> | <p>90% of reads use data written within the previous 1 minute. 99% of reads use data written within the previous 10 minutes.</p> |

| Category     | SLI                                                                                                                                                                                                                                                                                                                                                         | SLO                                                                       |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Correctness  | <p>The proportion of records injected into the state table by a correctness prober that result in the correct data being read from the league table.</p> <p>A correctness prober injects synthetic data, with known correct outcomes, and exports a success metric: count of all data_requests which were correct divided by count of all data_requests</p> | 99.99999% of records injected by the prober result in the correct output. |
| Completeness | <p>The proportion of hours in which 100% of the games in the data store were processed (no records were skipped).</p> <p>Uses metrics exported by the score pipeline: count of all pipeline runs that processed 100% of the records divided by count of all pipeline runs</p>                                                                               | 99% of pipeline runs cover 100% of the data.                              |

## 理论

可用性和延迟SLI基于2018年1月1日至2018年1月28日之间的测量。可用性SLO向下舍入到最接近的1%，而延迟SLO时序被舍入到最接近的50 ms。作者选择了所有其他数字，并验证了这些服务正在或高于这些级别运行。

尚未尝试验证这些数字是否与用户体验密切相关。[115](#)

## 错误预算

每个目标都有一个单独的错误预算，定义为该目标的100%减去（-）目标。例如，如果在四周中有1,000,000个请求发送到API服务器，则API可用性错误预算为1,000,000中的3%（100%-97%）：30,000个错误。

当我们的任何目标用尽错误预算时，我们将制定错误预算政策（请参阅附录B）。

## 说明和警告

- 请求指标是在负载均衡器上测量的。此度量可能无法准确度量用户请求未到达负载均衡器的情况。
- 我们仅将HTTP 5XX状态消息视为错误代码；其他一切都算成功。
- 正确性探测器使用的测试数据包含大约200个测试，每1秒注入一次。我们的错误预算是每四周48个错误。

<sup>115</sup>. 即使SLO中的数字不是严格依据证据的，也有必要对此进行记录，以使将来的读者可以理解这一事实，并做出适当的决定。他们可能会决定值得收集更多证据的投资。 ↵

## 附录B 错误预算政策示例

| Status        | Published       |
|---------------|-----------------|
| Author        | Steven Thurgood |
| Date          | 2018-02-19      |
| Reviewers     | David Ferguson  |
| Approvers     | Betsy Beyer     |
| Approval Date | 2018-02-20      |
| Revisit Date  | 2019-02-01      |

### 服务概述

示例游戏服务允许Android和iPhone用户彼此玩游戏。后端代码的新版本每天发布一次。每周都会推送新版本的客户。此策略同时适用于后端和客户端版本。

### 目标

该政策的目标是：

- 保护客户免受重复的SLO遗漏
- 鼓励平衡可靠性与其他功能

### 非目标

此政策无意作为对丢失的SLO的惩罚。停止更改是不可取的；该政策允许团队专注于

### 示例错误预算政策

当数据表明可靠性比其他产品功能更重要时的可靠性。

#### SLO小姐政策

如果服务在其SLO或更高的性能下运行，则将根据发布策略继续进行发布（包括数据更改）。

如果该服务超出了前四个星期的错误预算，我们将停止所有更改并发布P0 [116](#) 问题或安全修补程序以外的其他问题，直到该服务返回其SLO之内。

根据SLO未命中的原因，团队可能会投入更多资源来进行可靠性工作，而不是进行功能工作。在以下情况下，团队必须致力于可靠性：

- 代码错误或程序错误导致服务本身超出错误预算。
- 事后总结表明有机会软化严格的依赖关系。
- 错误分类的错误无法消耗可能导致服务错过其SLO的预算。

在以下情况下，团队可能继续研究非可靠性功能：

- 中断是由公司范围内的网络问题引起的。
- 中断是由另一个团队维护的服务引起的，另一个团队冻结了发布以解决其可靠性问题。
- 错误预算被用户超出了SLO的范围（例如，负载测试或渗透测试仪）。
- 错误分类的错误会消耗预算，即使没有用户受到影响。

### 停运政策

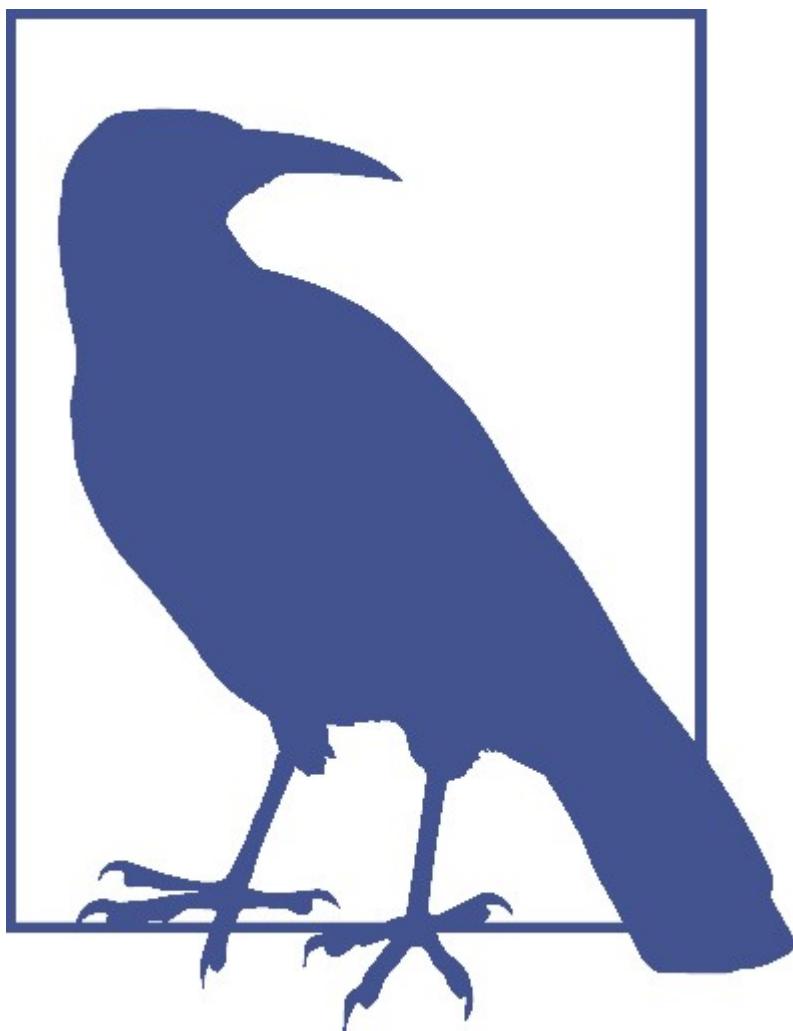
如果单个事件在四个星期内消耗了错误预算的20%以上，则团队必须进行事后调查。事后总结必须包含至少一个PO操作项以解决根本原因。

如果一类中断在一个季度中消耗了超过20%的错误预算，则团队必须在其季度计划文档中有一个PO项目<sup>117</sup>，以解决下一季度的问题。

### 升级政策

如果各方在错误预算的计算或其定义的具体措施上存在分歧，则应将此问题上报给CTO做出决定。

### 背景



本节是样板，旨在向不熟悉这些错误预算的人简要概述错误预算。

错误预算是SRE用于平衡服务可靠性和创新步伐的工具。变更是不稳定的主要来源，约占我们中断的70%，功能开发工作与稳定性工作竞争。错误预算构成了一种控制机制，可根据需要转移对稳定性的关注。

错误预算为1减去服务的SLO。99.9%的SLO服务的错误预算为0.1%。

如果我们的服务在四个星期内收到1,000,000个请求，则99.9%的可用性SLO会为我们提供该期间1,000个错误的预算。

116. P0是错误的最高优先级：所有人都在甲板上；放下其他所有东西，直到解决为止。[←](#)

117. 在Google，季度计划是公开的，团队要对其计划负责。[←](#)

## 附录C 事后分析的结果

在Google，我们有一个标准的事后总结模板，使我们能够始终如一地捕获事件的根本原因和触发因素，从而进行趋势分析。我们使用这种趋势分析来帮助我们确定针对系统性根本原因类型的改进，例如错误的软件界面设计或不成熟的变更部署计划。表C-1显示了过去7年中成千上万笔事后抽样的情况，显示了我们造成停机的八大诱因。

表C-1。2010--2017年排名前8位的中断触发因素

|                         |     |
|-------------------------|-----|
| ---                     | --- |
| Binary push             | 37% |
| Configuration push      | 31% |
| User behavior change    | 9%  |
| Processing pipeline     | 6%  |
| Service provider change | 5%  |
| Performance decay       | 5%  |
| Capacity management     | 5%  |
| Hardware                | 2%  |

表C-2列出了最重要的五个根本原因类别。

表C-2。停机的前五种根本原因类别

|                             |        |
|-----------------------------|--------|
| ---                         | ---    |
| Software                    | 41.35% |
| Development process failure | 20.23% |
| Complex system behaviors    | 16.90% |
| Deployment planning         | 6.74%  |
| Network failure             | 2.75%  |

## 关于编辑

**Betsy Beyer**是纽约市Google网站可靠性工程的技术作家。她以前曾为Google数据中心和硬件运营团队撰写过文档。在移居纽约之前，Betsy是斯坦福大学的技术写作讲师。Betsy在目前的职业生涯中学习了国际关系和英语文学，并获得了斯坦福大学和杜兰大学的学位。

尼尔·理查德·墨菲（Niall Richard Murphy）\*\*在互联网基础设施领域工作了20年。他是公司创始人，出版作家，摄影师，并拥有计算机科学与数学和诗歌研究学位。

**Dave Rensin**是Google SRE总监，之前是O'Reilly的作者，还是连续企业家。他拥有统计学学位。

**Kent Kawahara**是Google网站可靠性工程团队的项目经理，该团队的重点是Google Cloud Platform客户，其总部位于加利福尼亚州桑尼维尔。在担任Google以前的职位时，他管理技术和设计团队以开发广告支持工具，并与大型广告商和代理商合作制定战略性广告计划。加入Google之前，他曾在两家成功的电信初创公司从事产品管理，软件质量检查和专业服务。他拥有加州大学伯克利分校的电气工程和计算机科学学士学位。

**Stephen Thorne**是Google的高级网站可靠性工程师。他目前在客户可靠性工程部门工作，帮助将Google的Cloud客户的运营与Google SRE集成在一起。Stephen学习了如何在运行Google广告客户和发布商用户界面的团队中成为SRE，并随后在App Engine上工作。在加入Google之前，他曾在自己的祖国澳大利亚与垃圾邮件和病毒作斗争，在那里他还获得了计算机科学学士学位。