

架构师

ARCHITECT



热点 | Hot

Oracle的Java模块化系统保卫战

WebAssembly, 火狐赢了?

推荐文章 | Article

Python向来以慢著称,

为啥Instagram却唯独钟爱它

高负载微服务系统的诞生过程

观点 | Opinion

一语点醒技术人:你不是Google



卷首语

王晓波

什么是架构，什么是架构师？

——这似乎是聊架构话题时永恒的问题。

从内心讲我真的不想回答架构具体需要做什么，架构师应该具体负责什么。因为从实际情况看，**在不同的系统层级，不同的需求下架构师的职责也会不同；从不同的技术角度看**，架构师又是个变色龙——一时是技术的大拿，一时是技术的规划者，一时是技术团队的指挥者。

那么，该如何回答“什么是架构，什么是架构师”这个问题呢？这或许需要先搞清楚另外一个问题——一名程序员是如何走上架构师之路的？我从许多朋友那里了解到了很多实际案例，**程序员走上架构师之路，总结起来最多的原因是因为他早前代码写的好。**

那么，**代码写的好就是架构吗**？显然不是。代码写的好只是表象，做所有事情都需要规划，尤其是一个复杂的软件系统，这更需要规划，否则可能连一行代码都写不出。复杂的软件系统一定会需要做很多抽象设计、对象规划、接口规划等准备动作。也就是“上一辈程序员”口中所说的：详细设计。做架构主要的事情也依旧如此，需要对整个系统进行系统的规划：模块、通讯、边界、扩展、技术下沉等工作。这样的规划完成之后项目方能正常跑起来。



当然，**架构不仅仅是规划，还要做的另一件大事就是技术识别**。识别出系统中技术的难易区域，并分解复杂技术，使之成为一个个技术的黑盒子，在此之上再进行新的技术规划，使整个系统从技术角度来看是分层次的，从难到易，从大到小，但各层之间又是互相的黑盒。这也常说的让系统模块间达到“鸡犬相闻老死不相往来”的状态。

系统技术的识别完成之后还要对另一种技术进行识别，即人的技术。什么样的工程师适合写哪一层的代码，那一层的技术对程序员技术的深入程度要求到哪个点上。在做完这些事情整个架构表面上看是平稳进行了。

但实际上，架构的问题一定会再次前来打扰：首先是测试工程师来询问“对于整体系统架构而言这个应用该如何更好的被测试？”“我们需要用什么样的技术来更好地保证软件的质量？”然后是运维工程师来询问“该系统将跑在什么样的环境之上？”“我们应该提供什么样的服务器？”“服务器上我们会做哪些配置和安装哪些基础软件？”“我们需要提供一个什么样的网络环境？”“有什么样特殊的网络配置？”“我们需要做哪些安全策略？”……此时，架构师不时会像是一个掉入冰洞的猎人无比无助，头顶成群的苍蝇飞着，这些问题，有的懂点，有的不专业，还有的听说过没干过，有些仅限知道原理。其实这些**辣手的事情是考验架构师的一种能**

力：技术的宽度。

一个架构师需要足够的技术的宽度。从软件到硬件，从开发到测试，从运维到安全等都需要面面俱到的了解。当然你可能不是这单方面领域里面最深入的人，但是你需要知道它们是怎么做的（不仅仅是皮毛，要深入原理），并且要知道它们组合起来是个什么样的东西。

技术面也足够宽了之后，是不是就会成为完美架构师呢？答案是不会，因为还有新的问题要过来。这次的问题诸如“系统在未来的运行过程中运维需要做什么？”“系统在未来的功能迭代中如何更方便的扩展？”“系统应该怎么修改？”“系统应该被怎么样升级？”这时的你是不时很困惑？是不是感觉这个架构的世界好长啊，怎么像保姆一样什么都要管。但仔细想想这是应该的，因为一个系统初次开发并交付只是它生命周期中的一小部分而已。后面的维护、改造、升级才占了整个软件生命周期的绝大部分时间。你是它的架构设计者，是它灵魂之所在，你当然应该设计好它的未来。这也是架构师做好的最后一件事情：**系统未来的设计**。

仔细想想，上文提到的这些案例全是架构的糗事，但糗事其实是架构师成长路上的必经之路。因为一个没有经历失败的架构师一定不是个好的架构师。只有经历各种苦难，越过各种坑和各种痛苦之后才能成为一个优秀的架构师。架构师也是一个很独特职业，不像现代教育里已经很成熟的人文和物理教育体系，勤奋的人大都能经过系统的阅读和教育能走向成功。架构更像一种艺术、一门哲学，架构师们也仿佛经过多年积累后忽然间就像打通了任督二脉。那么走向架构师的路是不是无迹可寻呢？——这个问题留个大家来思考。

前瞻热点一睹为快

2017年10月17-19日 | 上海·宝华万豪酒店



前瞻热点 实践案例



《方圆并济：基于 Spark on Angel 的高性能机器学习》

黄明 腾讯数据平台部 T4 专家



《机器学习风控实践与发展》

武广柱 百度安全事业部首席架构师



《从“作坊”到“专业”，团队管理三要点》

乔梁 腾讯高级管理顾问



《越过山丘，已回不了头》

李宝泉 网易产品经理

技术大咖 重磅加盟



张雪峰
饿了么
CTO



施坚松
携程网
AVP



李双涛
饿了么
高级架构师



程军
饿了么
研发总监



凌云
携程
信息安全总监



Kingsum Chow
Alibaba Infrastructure Services Chief Scientist



梁飞 (虚极)
阿里巴巴
资深技术专家



贺师俊
百姓网
前端工程师



毕鹏
点融网
技术副总裁



张彭善
PayPal 大数据研发架构师
/ 资深数据科学家



孙志岗
网易
教育事业部战略总监



王伟钊
Google
Senior Staff Engineer

8折 优惠报名中, 每张立减1360元

截至2017年08月20日前

团购享受更多优惠

大会官网: 2017.qconshanghai.com
议题提交: speakers@cn.infoq.com
扫码关注大会官网, 获取更多大会信息



购票请联系售票经理Hanna
联系电话: 010-64738142
电子邮箱: hanna@infoq.com



CONTENTS / 目录

热点 | Hot

Oracle 的 Java 模块化系统保卫战

Apple 发布 Core ML，为 Apple 设备提供了机器学习功能

WebAssembly，火狐赢了？

理论派 | Theory

微服务，够了

推荐文章 | Article

Python 向来以慢著称，为啥 Instagram 却唯独钟爱它？

高负载微服务系统的诞生过程

观点 | Opinion

一语点醒技术人：你不是 Google



架构师 2017 年 7 月刊

本期主编 韩 婷

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

Oracle 的 Java 模块化系统保卫战

作者 Michael Redlich 译者 薛命灯



2017 年企业新兴技术 (ETE) 大会上最为及时的演讲之一要算由 Oracle JVM 负责人 Karen Kinnear 呈献的“[Java 的未来: 模块化及其他](#)”。在她演讲之前的这段时间发生了很多事情，其中最为引人瞩目的就是 5 月 8 号针对 JSR 376 的投票事件。

Kinnear 介绍了 Java 9 的目标，包括提升开发者效率和改进 Java 云 API。在对 [Java 9 相关的 JEP](#) 进行了一番评述之后，她开始专注于 Java 平台模块化系统 (JPMS) 的话题。JPMS 也就是 [Jigsaw 项目](#) 或 [JSR 376](#)。

Kinnear 向参会人员问了三个有关模块化的问题。

- 模块遗漏问题

模块系统在构建模块图时会检测遗漏的模块。

- 冲突问题

在构建模块图时包的冲突问题也会被检测到。

- 变更内部API是否安全

模块系统可以确保无法从模块外部访问内部的 API。

Kinnear 说，模块可以被集成到已有的应用程序里，她还演示了包与模块之间如何进行交互，解释了模块路径和类路径之间的区别。她还向开发者介绍了如何迁移到 Java 9，特别是如何在迁移过程中保持向后兼容。

Oracle Java 平台组的首席架构师 [Mark Reinhold](#) 在 2016 年 3 月份的[白皮书](#)中描述了 JPMS 的目标。

- 可靠的配置

使用声明式的程序组件依赖机制代替脆弱且容易出错的类路径机制。

- 强封装

组件可以声明自己的哪些公共类型是可以被其他组件访问哪些是不能被访问的。

社区的反应

Red Hat 的 JBoss 架构副主席 [Scott Stark](#) 表达了他对 JPMS 存在的一些[疑问](#)，Red Hat 认为这些问题影响 JPMS 无法达成 JSR 提交目标的主要因素。Stark 说：

Jigsaw 是一个全新的模块化系统，它可以很好地应用在 Java 上，但并没有在生产环境里那些基于 JVM 的真实应用上大规模尝试应用 JPMS。很多应用程序可能无法使用 Jigsaw，或者需要进行重大的重构才可以。

IBM 和 Red Hat 公开表示他们不会为当前的 JPMS 投赞成票。

尽管没有达成一致意见，Reinhold 仍然提交了 JSR 376 公开预览版，并声明“这对于广大的 Java 生态圈来说是最有益的，我们因此可以达成切实的目标”。在投票当天，他还向执行委员会（EC）提交了一封[公开信](#)，呼吁他们能够为 JSR 376 投赞成票。不过，最终 JSR 376 仍然没能通过投票。

投票之后

Twitter 的 JVM 和 GC 工程师 Tony Printezis 解释了 Twitter 投反对票的原因。

我们的主要疑问在于，JPMS 有可能会颠覆开发者，但却未能给他们带来直接的好处。我们担心因此会阻碍这项技术的大规模采用。我们希望 JPMS 能够对最初的目标做更全面的调整，从而真正地解决开发者的痛点。比如，非公开包名称冲突就与当前 JSR “不互相干扰” 和 “强封装”的目标不一致。而如果模块能够更加彻底地分离，那么就可以通过把包隐藏在模块内部来支持相同包的多个复本同时存在。如此直观的好处简化了开发人员模块化代码的工作，也因此能够加速 JPMS 的采用速度。

在与 InfoWorld 的一次[访谈](#)中，Reinhold 尝试着澄清人们对 JPMS 的误解。关于人们反对无法在 Java 中使用 Maven 这一问题上，Reinhold 说，这不是真的，“Maven 可以在 Java 9 里使用”。不过他承认，Maven 的插件可能无法正常运行，包括[Surefire 测试插件](#)。

Reinhold 确认了开发者最喜欢的一些库、框架和工具可能无法在 Java 9 中使用，这是因为当下的一些因素造成的，不过他说在正式发布时可能可以解决这些问题。他指出，这些项目的维护者已经在使用 Java 9 抢先版，所以他们会为这些项目做好支持 Java 9 的准备。这也就是为什么一些项目已经可以使用 Java 9，如 Spring Boot 和 Hibernate Validator。

很多开发团队认为，在他们将所有代码、框架和库模块化之前就不能使用 Java 9。Reinhold 说这也是不对的。

开发人员可以在 Java 9 里继续使用类路径，不过因为 Java 9 有了模块机制，所以开发人员就不再需要类路径了。

伦敦 Java 社区共同创始人及 jClarity CEO Martijn Verburg 与[InfoQ 交流](#)了他对 JSR 376 投票的看法。在谈到 JVM 模块化的好处时，他说：它为 Java 代码提供了更多的安全性；它隐藏了很多内部 API 或者不

应该暴露给开发人员的 API；不过需要为那些被隐藏的功能提供安全的替代方案。Java 的运行时将变得更小，因为运行时被拆分成更小的模块。Java 9 将提供 [jlink](#) 工具，用于将应用程序部署在更小的运行时上，只安装必要的组件。服务器端的应用程序就不需要把客户端的 GUI（如 AWT 或 Swing）也包含在内。这样，Java 可以启动得更快，可以在更小的设备上运行应用，在云端的部署也会更快。

IBM 的高级技术研究员 Tim Ellison 最近表达了他对如何在 JSR 376 上达成共识的看法。他说：

我们希望看到修订过的规范重新呈现给 JCP 执行委员会，也希望执行委员会能够支持专家组的结论。IBM 关心的是企业应用在迁移到 Java 9 时的兼容性问题。升级到 Java 9 对迁移有重大的影响，而 JPMS 的默认行为在这方面会提供很大的帮助。

JSR 376 即将进入到终稿阶段。在定稿之前可能还有一些小幅度的改动，不过整个过程充分展示了 JCP 致力于为 Java 提供更加强大的语言特性。感谢 Oracle 一直在主导这个规范，以及专家组在这个里程碑上所投入的大量精力。

Reinhold 最近针对 Java 9 的 GA 版本发布日期提出了一个新的提议。他说：

为了迎接各种可能的结果，我建议保持 6 月 22 号的 JDK 9 初始候选版本发布日期不变，不过将 GA 版本的发布日期向后延期，为通过 JCP 流程争取更多的时间。我提议将 GA 版本的发布日期向后延期 8 周，从 7 月 27 号调整为 9 月 21 号。

JSR 376 的下一个投票日期是周一，也就是 2017 年 6 月 26 号。

编辑后记

Michael Redlich 是 ETE 的积极参与者，他从 2008 年开始作为 ETE 的参会者和演讲者，2013 年成为 ETE 指导委员会的成员。

Apple 发布 Core ML， 为 Apple 设备提供了机器学习功能

作者 Roland Meertens 译者 Rays



Apple 在 [WWDC 2017 大会](#)上发布了一种使用机器学习的方式，以及一种开发人员在自身应用中添加机器学习的方式。

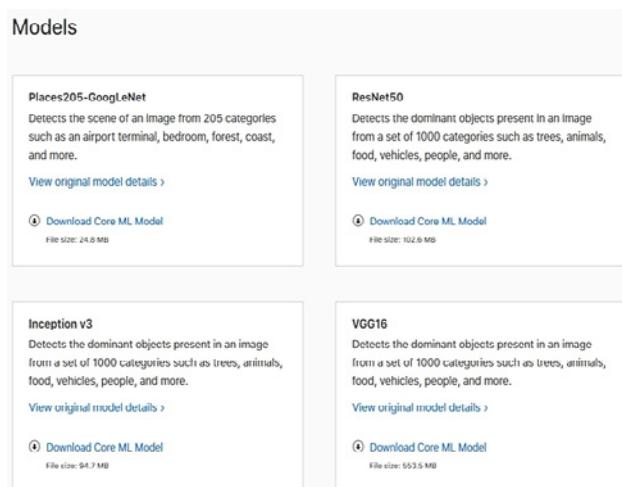
Apple 新发布的机器学习 API 称为 Core ML，允许开发人员将机器学习模型集成到 App 中，App 运行于采用 iOS、macOS、watchOS 和 tvOS 的 Apple 设备上。由于模型驻留在设备上，因此数据不会离开设备。

Core ML 提供了应用开发人员可用的多种 API 调用，无需开发人员在 App 中额外添加任何模型。例如，它所提供的计算机视觉算法包括了面部识别和追踪、特征点检测和事件识别。开发人员也可调用 Core ML 做自然语言分析，例如实现对电子邮件、文本和 Web 页面的分析。自然语言处理 API 调用包括了语言检测、标记化（Tokenization）、词性标注（POS

tagging) 抽取和命名实体识别等。

开发人员也可以设计并使用自己的机器学习模型。Core ML 支持超过 30 层的深度神经网络，也支持其他一些机器学习方法，例如 SVM 和线性模型。在设备上可以使用 CPU 和 GPU，这为在 Apple 设备上运行强大的算法提供了很大的空间。

Apple 提供了一些预先训练好的模型，开发人员可以下载它们到自己的 App 中。在 [Apple 开发者网站](#) 上提供的一个模型可检测 205 种图像场景（例如候机楼或卧室）。另外还提供了三种模型，可用于检测图像中的对象。开发人员也可以使用 Apple 提供的[转换工具](#)，将一些已有的模型转换为 Core ML 格式。该工具支持的机器学习工具包括：Keras（使用 Tensorflow 作为后端）、Caffe、Scikit-learn、libsvm 和 XGBoost。但是它不支持将已有的 Tensorflow 模型导入 Core ML 中，这在用于 Android 的 Tensorflow Lite 上是支持的。



对于那些想在自身 App 中添加人工智能的开发人员，可以访问 Core ML 的[官方文档](#)。

WebAssembly，火狐赢了？

作者 金灵杰



自从 WebAssembly 标准发布以及各大浏览器完成对其默认支持之后，WebAssembly 成为前端热门话题。在 WebAssembly 之前，类似的前端二进制标准有火狐主导的 asm.js 和 Chrome 主导的 PNaCl。二者均用于将后端 C/C++ 代码用于前端，作为它们折中方案，WebAssembly 标准更偏向于 asm.js 的实现。Chrome 在支持了 WebAssembly 标准之后，宣布将放弃对 [PNaCl 的支持](#)。

作为前端标准，PNaCl 在创立之初就有其先天不足。在设计上，PNaCl 代码和前端代码（Javascript、HTML 等）高度独立，并且 PNaCl 代码运行在独立进程中，这使得 PNaCl 代码和页面代码交互成本非常高（需要通过 IPC 方式）。另外，出于安全考虑，PNaCl 进程运行在沙箱环境中，Chrome 为此定义了一套 API，称为：Pepper。Pepper 定义的 API 中，有

许多和现行 Web 标准重复。

更加严重的问题是，不论是 Pepper 还是 PNaCl，都没有明确的二进制代码规范。因此非 Chrome 浏览器如果要兼容 PNaCl 插件，要么反向工程 Pepper 来自己实现一套接口实现，要么从 Chromium 工程中导入其中的实现代码。无论哪种方式，都需要和 Google 的修改同步。这对于开发者来说是不可接受的。

相反，asm.js 实现方式从一开始就尽量贴近前端开发和已有前端标准。asm.js 用 Javascript 数组来表示内存，并将 C/C++ 代码编译成 Javascript 以操作这个数组。这种实现方式相比 PNaCl 有一个很大的优势：所有代码在同一个 JS 虚拟机中运行，可以方便的和其他 Javascript 代码、DOM 进行交互。另外，这样的实现没有引入新的 API，因此文档相关的工作也比较少。

综上所述，WebAssembly 标准最终和 asm.js 比较接近，它实现在 JS 虚拟机中，可以和页面 Javascript 之间方便的进行调用。WebAssembly 标准除了新增加载和链接 WebAssembly 代码相关的 [API](#) 之外，没有定义新的平台相关 API。和 asm.js 不同的是，WebAssembly 完整定义了二进制代码规范，相关[规范文档](#)已经完成。

当然，Google 和其他团队在 WebAssembly 标准的制定上也功不可没。针对 PNaCl 插件，Google 已经发布了[迁移文档](#)。可以说，WebAssembly 标准的发布，真正的赢家是开发者！

微服务，够了

作者 Adam Drake 译者 薛命灯



资深架构师 [Adam Drake](#) 在他的博客上分享了他对微服务的看法，他从自己的经验出发，结合 Martin Fowler 对微服务的见解，帮助想要采用微服务的公司重新审视微服务。以下内容已获得作者翻译授权，查看英文原文 [Enough with the microservices](#)。

简介

关于微服务的优势和劣势已经有过太多的讨论，不过我仍然看到很多成长型初创公司对它进行着盲目崇拜。冒着“重复发明轮子”的风险（Martin Fowler 已经写过“Microservice Premium”的文章），我想把我的一些想法写下来，在必要的时候可以发给客户，也希望能够帮助人们避免犯下

我之前见过的那些错误。在进行架构或技术选型时，将网络上找到的一些所谓的最佳实践文章作为指南，一旦做出了错误的决定，就要付出惨重的代价。如果能够帮助哪怕一个公司避免犯下这种错误，那么写这篇文章都是值得的。

如今微服务是个热门技术，微服务架构一直以来都存在（面向服务架构也算是吧？），但对于我所见过的大部分公司来说，微服务不仅浪费了他们的时间，分散了他们的注意力，而且让事情变得更糟糕。

这听起来似乎很奇怪，因为大部分关于微服务的文章都会肯定微服务的各种好处，比如解耦系统、更好的伸缩性、消除开发团队之间的依赖，等等。如果你的公司有 Uber、Airbnb、Facebook 或 Twitter 那样的规模，那么就没有什么问题。我曾经帮助一些大型组织转型到微服务架构，包括搭建消息系统和采用一些能够提升伸缩性的技术。不过，对于成长型初创公司来说，很少需要这些技术和服务。

Russ Miles 在他的《[让微服务失效的八种方式](#)》这篇文章中表达了他的首要观点，而在我看来，这些场景却到处可见。成长型初创公司总是想模仿那些大公司的最佳实践，用它们来弥补自身的不足。但是，最佳实践是要视情况而定的。有些东西对于 Facebook 来说是最佳实践，但对于只有不到百人的初创公司来说，它们就不一定也是最佳实践。

如果你的公司比那些大公司小一些，在一定程度上你仍然能够从微服务架构中获益。但是，对于成长型初创公司来说，大规模地迁移到微服务是一种过错，而且对技术人来说是不公平的。

为什么选择微服务？

一般来说，成长型初创公司采用微服务架构最主要的目的为了减少或消除开发团队之间的依赖，或者提升系统处理大流量负载的能力（比如伸缩性）。开发人员经常抱怨的问题和常见的症状包括合并冲突、由未完整实现的功能引起的部署错误以及伸缩性问题。接下来让我们逐个说明这些问题。

依赖

在初创公司的早期阶段，开发团队规模不大，使用的技术也很简单。人们在一起工作，不会出现混乱，要实现一些功能也比较快。一切看起来都很美好。

随着公司的不断发展，开发团队也在壮大，代码库也在增长，然后就出现了多个团队在同一个代码库上工作的情况。这些团队的大部分成员都是公司早期的员工。因为初创公司的早期员工一般都是初级开发人员，他们并没有意识到一个问题，那就是在团队规模增长和代码库增长的同时，沟通效率也需要随之提升。对于缺乏经验的技术人员来说，他们倾向于通过技术问题来解决人的问题，并希望通过微服务来减少开发团队之间的依赖和耦合。

实际上，他们真正需要做的是通过有效的沟通来解决人的问题。当一个初创公司有多个开发团队时，团队之间需要协调，团队成员需要知道每个人都在做什么，他们需要协作。在这样规模的企业里，软件开发其实具备了社交的性质。如果团队之间缺乏沟通或者缺乏信息分享，不管用不用微服务，一样存在依赖问题，而就算使用了微服务，也仍然存在负面的技术问题。

将代码模块化作为解决这个问题的技术方案，确实能够缓解软件开发固有的团队依赖问题，但团队间的沟通仍然要随着团队规模的增长而不断改进。

不要混淆了解耦和分布式二者的含义。由模块和接口组成的单体可以帮助你达到解耦的目的，而且你也应该这么做。你没有必要把应用程序拆分成立体的多个独立服务，在模块间定义清晰的接口同样能达到解耦的目的。

部分功能实现

微服务里需要用到[功能标志](#)（feature flag），微服务开发人员需要

熟悉这种技术。特别是在进行快速开发（下面会深入讨论）的时候，你可能需要部署一些功能，这些功能在某些平台上还没有实现，或者前端已经完全实现，但后端还没有。随着公司的发展，部署和运维系统变得越来越自动化和复杂，功能标志也变得越来越重要。

水平伸缩

通过部署同一个微服务的多个实例来获得伸缩性，这是微服务的优点之一。不过，大多数过早采用微服务的公司在这些微服务背后使用了同一个存储系统。也就是说，这些服务具备了伸缩性，但整个应用并不具备伸缩性。如果你正打算使用这样的伸缩方式，那为什么不直接在负载均衡器后面部署多个单体实例呢？你可以用更简单的方式达到相同的目的。再者，水平伸缩应该被作为杀手锏来使用。你首先要关注的应该是如何提升应用程序的性能。一些简单的优化常常能带来数百倍的性能提升，这里也包括如何正确地使用其他服务。例如，我在一篇博文里提到的 [Redis 性能诊断](#)。

我们为微服务做好准备了吗？

在讨论架构选型时，人们经常会忽略这个问题，但其实却是最重要的。高级技术人员在了解了开发人员或业务人员的抱怨或痛点之后，在网上找寻找解决方案，他们总是宣称能解决这些问题。但在这些信誓旦旦的观点背后，有很多需要注意的地方。微服务有利也有弊。如果你的企业足够成熟，并且具有一定的技术积累，那么采用微服务所面临的挑战会小很多，并且能够带来更多正面好处。那么怎样才算已经为微服务做好准备了呢？Martin Fowler 在多年前表达了他对[微服务先决条件](#)的看法，但是从我的经验来看，大多数成长型初创公司完全忽略了他的观点。Martin 的观点是一个很好的切入点，让我们来逐个说明。

我敢说，大部分成长型初创公司几乎连一个先决条件都无法满足，更不用说满足所有的条件了。如果你的技术团队不具备快速配置、部署和监控能力，那么在迁移到微服务前必须先获得这些能力。接下来让我们更详

细地讨论这些先决条件。

快速配置

如果你的开发团队里只有少数几个人可以配置新服务、虚拟环境或其他配套设施，那说明你们还没有为微服务做好准备。你的每个团队里都应该要有几个这样的人，他们具备了配置基础设施和部署服务的能力，而且不需要求助于外部。要注意，光是有一个 DevOps 团队并不意味着你在实施 DevOps。开发人员应该参与管理与应用程序相关的组件，包括基础设施。

类似的，如果你没有灵活的基础设施（易于伸缩并且可以由团队里的不同人员来管理）来支撑当前的架构，那么在迁移到微服务前必须先解决这个问题。你当然可以在裸机上运行微服务，以更低的成本获得出众的性能，但在服务的运维和部署方面也必须具备灵活性。

基本的监控

如果你不曾对你的单体应用进行过性能监控，那么在迁移到微服务时，你的日子会很难过。你需要熟悉系统级别的度量指标（比如 CPU 和内存）、应用级别的度量指标（比如端点的请求延迟或端点的错误）和业务级别的度量指标（比如每秒事务数或每秒收益），这样才可以更好地理解系统的性能。在性能方面，微服务生态系统比单体系统要复杂得多，就更不用提诊断问题的复杂性了。你可以搭建一个监控系统（如 Prometheus），在将单体应用拆分成微服务之前对应用做一些增强，以便进行监控。

快速部署

如果你的单体系统没有一个很好的持续集成流程和部署系统，那么要集成和部署好你的微服务几乎是件不可能的事。想象一下这样的场景：10个团队和100个服务，它们都需要进行手动测试和部署，然后再将这些工作与测试和部署一个单体所需要的工作进行对比。100个服务会出现多少种问题？而单体系统呢？这些先决条件很好地说明了微服务的复杂性。

Phil Calcado 在 Fowler 的先决条件清单里添加了一些东西，不过我认为它们更像是重要的扩展，而不是真正的先决条件。

如果我们具备了这些先决条件呢？

就算具备了这些条件，仍然需要注意微服务的负面因素，确保微服务能够为你的业务带来真正的价值。事实上，很多技术人员对微服务中存在的[分布式计算谬论](#)视而不见，但为了确保能够成功，这些问题是非常需要考虑到的。对于大部分成长型初创公司来说，基于各种原因，他们应该避免使用微服务。

运营成本的增加

快速部署这一先决条件已经涵盖了一部分成本，除此之外，对微服务进行容器化（可能使用 Docker）和使用容器编排系统（比如 Kubernetes）也需要耗费很多成本。Docker 和 Kubernetes 都是很优秀的技术，但是对于大部分成长型初创公司来说，它们都是一种负担。我见过初创公司使用 rsync 作为部署和编排工具，我也见过很多的初创公司陷入运维工具的复杂性泥潭里，他们因此浪费了很多时间，而这些时间本来可以用于为用户开发更多的功能。

你的应用会被拖慢

如果你的单体系统里包含了多个模块，并且在模块间定义了良好的 API，那么 API 之间的交互就几乎没有额外开销。但对于微服务来说就不是这么一回事了，因为它们一般运行在不同的机器上，它们之间需要通过网络进行交互。这样会在一定程度上拖慢整个系统。如果一个请求需要多个服务进行同步交互，那么情况会变得更加糟糕。我曾经工作过的一个公司，他们需要调用将近 10 个服务才能处理完某些请求。处理请求的每一个步骤都需要额外的网络开销和延迟，但实际上，他们可以把这些服务放在单个软件包里，按照不同的模块来区分，或者把它们设计成异步的。

这样可以为他们节省大量的基础设施成本。

本地开发变得更加困难

如果你有一个单体应用，后端只有一个数据库，那么在开发过程中，在本地运行这个应用是很容易的。如果你有 100 个服务，并使用了多个数据存储系统，而且它们之间互相依赖，那么本地开发就会变成一个噩梦。即使是 Docker 也无法把你从这种复杂性泥潭中拯救出来。虽然事情原本可以简单一些，不过仍然需要处理依赖问题。理论上说，微服务不存在这些问题，因为微服务被认为是相互独立的。不过，对于成长型初创公司来说，就不是这么一回事了。技术人员一般需要在本地运行所有（或者几乎所有）的服务才能进行新功能的开发和测试。这种复杂性是对资源的巨大浪费。

难以伸缩

对单体系统进行伸缩的最简单方式是在负载均衡器后面部署单体系统的多个实例。在流量增长的情况下，这是一种非常简单的伸缩方式，而且从运维角度来讲，它的复杂性是最低的。你的系统在编排平台（如 [Elastic Beanstalk](#)）上运行的时间越长越好，你和你的团队就可以集中精力开发客户需要的东西，而不是忙于解决部署管道问题。使用合适的 CI/CD 系统可以缓解这个问题，但在微服务生态系统里，事情要复杂得多，而且这些复杂性所造成的麻烦已经超过了它们所能带来的好处。

然后呢？

如果你刚好身处一个成长型初创公司里，需要对架构做一些调整，而微服务似乎不能解决你的问题，这个时候应该怎么办？

Fowler 提出的先决条件可以说是技术领域的[能力成熟度模型](#)，Fowler 在他的文章里对成熟度模型进行过介绍。如果这种成熟度模型对于公司来说是说得通的，那么我们可以按照 Fowler 提出的先决条件，并

使用其他的一些中间步骤为向微服务迁移做好准备。下面的内容引用自 Fowler 的文章。

关键是要认识到，成熟度模型的评估结果并不代表你的当前水平，它们只是在告诉你需要做哪些工作才能朝着改进的目标前进。你当前的水平只是一种中间工作，用于确定下一步该获得什么样的技能。

那么，我们该做出怎样的改进，以及如何达成这些目标？我们需要经过一些简单的步骤，其中前面两步就可以解决很多在向微服务迁移过程中会出现的问题，而且不会带来相关的复杂性。

- 清理应用程序。确保应用程序具有良好的自动化测试套件，并使用了最新版本的软件包、框架和编程语言。
- 重构应用程序，把它拆分成多个模块，为模块定义清晰的API。不要让外部代码直接触及模块内部，所有的交互都应该通过模块提供的API来进行。
- 从应用程序中选择一个模块，并把它拆分成独立的应用程序，部署在相同的主机上。你可以从中获得一些好处，而不会带来太多的运维麻烦。不过，你仍然需要解决这两个应用之间的交互问题，虽然它们都部署在同一个主机上。不过你可以无视微服务架构里固有的网络分区问题和分布式系统的可用性问题。
- 把独立出来的模块移动到不同的主机上。现在，你需要处理跨网络交互问题，不过这样可以让这两个系统之间的耦合降得更低。
- 如果有可能，可以重构数据存储系统，让另一个主机上的模块负责自己的数据存储。

在我所见过的公司里，如果他们能够完成前面两个步骤就算万事大吉了。如果他们能够完成前面两个步骤，那么剩下的步骤一般不会像他们最初想象的那么重要了。如果你决定在这个过程的某个点上停下来，而系统仍然具有可维护性和比刚开始时更好的状态，那么就再好不过了。

结尾

我不能说这些想法都是独一无二的，也不能说是我所独有的。我只是从其他遭遇了相同问题的人那里收集想法，并连同观察到的现象在这里作了一次总结。还有其他很多比我更有经验的人也写过这方面的文章，他们剖析地更加深入，比如 Sander Mak 写的有关模块和[微服务的文章](#)。不管怎样，对于正在考虑对他们的未来架构做出调整的公司来说，这些经验都是非常重要的。认真地思考每一个问题，确保微服务对你们的组织来说是一个正确的选择。

最起码在完成了上述的前面两个步骤之后，再慎重考虑一下微服务对于你的组织来说是否是正确的方向。你之前的很多问题可能会迎刃而解。

Python 向来以慢著称， 为啥 Instagram 却唯独钟爱它？

作者 朱雷



PyCon 是全世界最大的以 Python 编程语言 为主题的技术大会，大会由 Python 社区组织，每年举办一次。在 Python 2017 上，Instagram 的工程师们带来了一个有关 Python 在 Instagram 的主题演讲，同时还分享了 Instagram 如何将整个项目运行环境升级到 Python 3 的故事。本文为该次演讲的内容摘要，由 Python 爱好者朱雷撰写，聊聊架构经授权发布。

Instagram 是一款移动端的照片与视频分享软件，由 Kevin Systrom 和 Mike Krieger 在 2010 年创办。Instagram 在发布后开始快速流行。于 2012 年被 Facebook 以 10 亿美元的价格收购。而当时 Instagram 的员工仅有区区 13 名。

如今，Instagram 的总注册用户达到 30 亿，月活用户超过 7 亿（作为对比，微信最新披露的月活跃用户为 9.38 亿）。而令人吃惊的是，这么高的访问量背后，竟完全是由以速度慢著称的 Python + Django 支撑。

为什么选择 Python 和 Django

Instagram 选择 Django 的原因很简单，Instagram 的两位创始人（Kevin Systrom and Mike Krieger）都是产品经理出身。在他们想要创造 Instagram 时，Django 是他们所知道的最稳定和成熟的技术之一。

时至今日，即使已经拥有超过 30 亿的注册用户。Instagram 仍然是 Python 和 Django 的重度使用者。Instagram 的工程师 Hui Ding 说到：“一直到用户 ID 已经超过了 32bit int 的限额（约为 20 亿），Django 本身仍然没有成为我们的瓶颈所在。”

不过，除了使用 Django 的原生功能外，Instagram 还对 Django 做了很多定制化工作：

- 扩展 Django Models 使其支持 Sharding（一种数据库分片技术）。
- 手动关闭 GC（垃圾回收）来提升 Python 内存管理效率，他们同样也写过一篇博客来说明这件事情：Dismissing Python Garbage Collection at Instagram。
- 在位于不同地理位置的多个数据中心部署整套系统。

Python 语言的优势所在

Instagram 的联合创始人 Mike Krieger 说过：“我们的用户根本不关心 Instagram 使用了哪种关系数据库，他们当然也不关心 Instagram 是用什么编程语言开发的。”

所以，Python 这种 简单 而且 实用至上 的编程语言最终赢得了 Instagram 的青睐。他们认为，使用 Python 这种简单的语言有助于塑造 Instagram 的工程师文化，那就是：

1. 专注于定位问题、解决问题，而不是工具本身的各种花花绿绿的特性；
2. 使用那些经过市场验证过的成熟技术方案，而不用被工具本身的问题所烦扰；
3. 用户至上：专注于用户所能看到的新特性，为用户带去价值。

但是，即使使用 Python 语言有这么多好处，它还是很慢，不是吗？

不过，这对于 Instagram 不是问题，因为他们认为：“Instagram 的最大瓶颈在于开发效率，而不是代码的执行效率”。

At Instagram, our bottleneck is development velocity, not pure code execution.

所以，最终的结论是：你完全可以使用 Python 语言来实现一个超过几十亿用户使用的产品，而根本不用担心语言或框架本身的性能瓶颈。

如何提升运行效率

但是，即使是选用了拥有诸多好处的 Python 和 Django。在 Instagram 的用户数迅速增长的过程中，性能问题还是出现了：服务器数量的增长率已经慢慢的超过了用户增长率。Instagram 是怎么应对这个问题的呢？

他们使用了这些手段来缓解性能问题：

- 开发工具来帮助调优：Instagram 开发了很多涵盖各个层面的工具，来帮助他们进行性能调优以及找到性能瓶颈。
- 使用 C/C++ 来重写部分组件：把那些稳定而且对性能最敏感的组件，使用 C 或 C++ 来重写，比如访问 memcache 的 library。
- 使用 Cython：Cython 也是他们用来提升 Python 效率的法宝之一。

除了上面这些手段，他们还在探索异步 I/O 以及新的 Python Runtime 所能带来的性能可能性。

为什么要升级到 Python 3

在相当长的一段时间，Instagram 都跑在 Python 2.7 + Django 1.3

的组合之上。在这个已经落后社区很多年的环境上，他们的工程师们还打了非常非常多的小 patch。难道他们要被永远卡在这个版本上吗？

所以，在经过一系列的讨论后，他们最终做出一个重大的决定：升级到 Python 3 ! !

事实上，Instagram 目前已经完成了将运行环境迁移到 Python 3 的工作 - 他们的整套服务已经在 Python 3 上跑了好几个月了。那么他们是怎么做到的呢？接下来便是由 Instagram 工程师 Lisa guo 带来的 Instagram 如何迁移到 Python 3 的故事。

对于 Instagram 来说，下面这些因素是推动他们将运行环境迁移到 Python 3 的主要原因。

1. 新特性：类型注解 Type Annotations

看看下面这段代码：

```
def compose_from_max_id(max_id):    '''@param str max_id'''
```

图中函数的 max_id 参数究竟是什么类型呢？int ? tuple ? 或是 list? 等等，函数文档里面说它是 str 类型。

但随着时间推移，万一这个参数的类型发生变化了呢？如果某位粗心的工程师修改代码的同时忘了更新文档，那就会给函数的使用者带来很大麻烦，最终还不如没有注释呢。

2. 性能

Instagram 的整个 Django Stack 都跑在 uwsgi 之上，全部使用了同步的网络 I/O。这意味着同一个 uwsgi 进程在同一时间只能接收并处理一个请求。这让如何调优每台机器上应该运行的 uwsgi 进程数成了一个麻烦事：

为了更好利用 CPU，使用更多的进程数？但那样会消耗大量的内存。而过少的进程数量又会导致 CPU 不能被充分利用。

为此，他们决定跳过 Python 2 中哪些蹩脚的异步 I/O 实现（可怜的 gevent、tornado、twisted 众），直接升级到 Python 3，去探索标

准库中的 `asyncio` 模块所能带来的可能性。

3. 社区

因为 Python 社区已经停止了对 Python 2 的支持。如果把整个运行环境升级到 Python 3, Instagram 的工程师们就能和 Python 社区走的更近，可以更好的把他们的工作回馈给社区。

迁移方案

在 Instagram, 进行 Python 3 的迁移需要必须满足两个前提条件：

- 不停机，不能有任何的服务因此不可用；
- 不能影响产品新特性的开发。

但是，在 Instagram 的开发环境中，要满足上面这两点来完成迁移 到 Python 3.6 这种庞大的工程是非常困难的。

基于主分支的开发流程

即便使用了以多分支功能著称的 git, Instagram 所有的开发工作都是主要在 `master` 分支上进行的, Instagram 所奉行的开发哲学是：『不管是多大的新特性或代码重构，都应该拆解成较小的 Commit 来进行。』

那些被合并进 `master` 分支的代码，都将在一个小时内被发布到线上环境。而这样的发布过程每天将会发生上百次。在这么频繁的发布频率下，如何在满足之前的那两个前提下来完成迁移变得尤其困难。

被弃用的迁移方案

创建一个新分支

很多人在处理这类问题时，第一个蹦进脑子的想法就是：『让我们 创建一个分支，当我们开发完后，再把分支合并进来』。但在 Instagram 这么高的迭代频率上，使用一个独立分支并不是好主意：

- Instagram 的 Codebase 每天都在频繁更新，在开发 Python 3 分支的过程中，让新分支与现有 `master` 分支保持同步开销极大，同时极易出错；

- 最终将 Python 3 分支这个改动非常多的分支合并回 Master 拥有非常高的风险；
- 只有少数几个工程师在 Python 3 分支上专职负责升级工作，其他想帮助迁移工作的工程师无法参与进来。

挨个替换接口

还有一个方案就是，挨个替换 Instagram 的 API 接口。但是 Instagram 的不同接口共享着很多通用模块。这个方案要实施起来也非常困难。

微服务

还有一个方案就是将 Instagram 改造成微服务架构。通过将那些通用模块重写成 Python 3 版本的微服务来一步步完成迁移工作。

但是这个方案需要重新组织海量的代码。同时，当发生在进程内的函数调用变成 RPC 后，整个站点的延迟会变大。此外，更多的微服务也会引入更高的部署复杂度。

所以，既然 Instagram 的开发哲学是：小步前进，快速迭代。他们最终决定的方案是：一步一步来，最终让 master 分支上的代码同时兼容 Python 2 和 Python 3。

正式迁移到 Python 3

既然要让整个 codebase 同时兼容 Python 2 和 Python 3，那么首先要符合这点的就是那些被大量使用的第三方 package。针对第三方 package，Instagram 做到了下面几点：

- 拒绝引入所有不兼容 Python 3 的新 package；
- 去掉所有不再使用的 package；
- 替换那些不兼容 Python 3 的 package。

在代码的迁移过程中，他们使用了工具 modernize 来帮助他们。

使用 modernize 时，有一个小技巧：每次修复多个文件的一个兼容问题，而不是一下修复一个文件中的多个兼容问题。这样可以让 Code

Review 过程简单很多，因为 Reviewer 每次只需要关注一个问题。

对于 Python 这种灵活性极强的动态语言来说，除了真正去执行代码外，几乎没有其他比较好的检查代码错误的手段。

前面提到，Instagram 所有被合并到 master 的代码提交会在一个小时上线到线上环境，但这不是没有前提条件的。在上线前，所有的提交都需要通过成千上万个单元测试。

于是，他们开始加入 Python 3 来执行所有的单元测试。一开始，只有极少数的单元测试能够在 Python 3 环境下通过，但随着 Instagram 的工程师们不断的修复那些失败的单元测试，最终所有的单元测试都可以在 Python 3 环境下成功执行。

但是，单元测试也是有局限性的：

- Instagram 的单元测试没有做到 100% 的代码覆盖率；
- 很多第三方模块都使用了 mock 技术，而 mock 的行为与真实的线上服务可能会有所不同。

所以，当所有的单元测试都被修复后，他们开始在线上正式使用 Python 3 来运行服务。

这个过程并不是一蹴而就的。首先，所有的 Instagram 工程师开始访问到这些使用 Python 3 来执行的新服务，然后是 Facebook 的所有雇员，随后是 0.1%、20% 的用户，最终 Python 3 覆盖到了所有的 Instagram 用户（见图 1）。

迁移过程的技术问题

Instagram 在迁移到 Python 3 时碰到很多问题，下面是最典型的几个：

Unicode 相关的字符串问题

Python 3 相比 Python 2 最大的改动之一，就是在语言内部对 unicode 的处理。

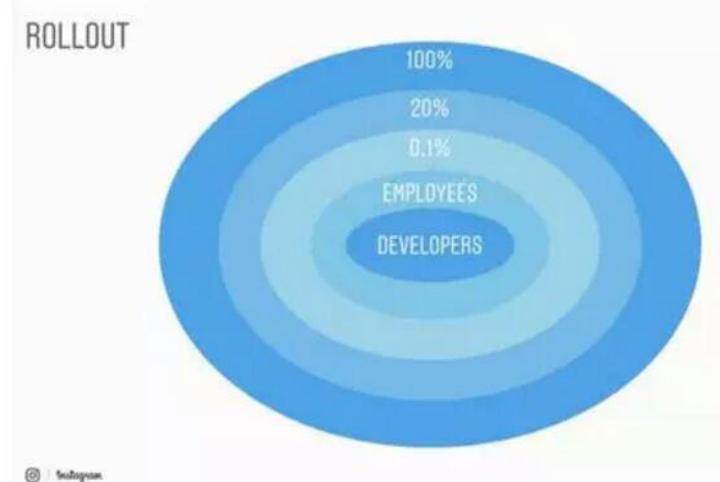


图 1

在 Python 2 中，文本类型（也就是 `unicode`）和二进制类型（也就是 `str`）的边界非常模糊。很多函数的参数既可以是文本，也可以是二进制。但是在 Python 3 中，文本类型和二进制类型的字符串被完全的区分开了。

于是，下面这段在 Python 2 下可以正常运行的代码在 Python 3 下就会报错：

```
mymac = hmac.new('abc')
TypeError: key: expected bytes or bytearray, but got 'str'
```

解决办法其实很简单，只要加上判断：如果 `value` 是文本类型，就将其转换为二进制。如下所示：

```
value = 'abc'
if isinstance(value, six.text_type):      value = value.encode(encoding='utf-8')
mymac = hmac.new(value)
```

但是，在整个代码库中，像上面这样的情况非常多。作为开发人员，如果需要在调用每个函数时都要想想： 这里到底是应该编码成二进制，或者是解码成文本呢？ 将会是非常大的负担。

于是 Instagram 封装了一些名为 `ensure_str()`、`ensure_binary()`、`ensure_text()` 的帮助函数，开发人员只需对那些不确定类型

的字符串，使用这些帮助函数先做一次转换就好。

```
mymac = hmac.new(ensure_binary('abc'))
```

不同 Python 版本的 pickle 差异

Instagram 的代码中大量使用了 pickle。比如用它序列化某个对象，然后将其存储在 memcache 中。如下面的代码所示：

```
memcache_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)  
data = pickle.loads(memcache_data)
```

问题在于，Python 2 与 Python 3 的 pickle 模块是有差别的。

如果上文的第一行代码，刚好是由 Python 3 运行的服务进行序列化后存入 memcache。而反序列化的过程却是由 Python 2 进行，那代码运行时就会出现下面的错误：

```
ValueError: unsupported pickle protocol: 4
```

这是由于在 Python 3 中，pickle.HIGHEST_PROTOCOL 的值为 4，而 Python 2 中的的 pickle 最高支持的版本号却是 2。那么如何解决这个问题呢？

Instagram 最终选择让 Python 2 和 Python 3 使用完全不同的 namespace 来访问 memcache。通过将二者的数据读写完全隔开来解决这个问题。

迭代器

在 Python 3 中，很多内置函数被修改成了只返成迭代器 Iterator：

```
map()  
filter()  
dict.items()
```

迭代器有诸多好处，最大的好处就是，使用迭代器不需要一次性分配大量内存，所以它的内存效率比较高。

但是迭代器有一个天然的特点，当你对某个迭代器做了一次迭代，访问完它的内容后，就没法再次访问那些内容了。迭代器中的所有内容都只

能被访问一次。

在 Instagram 的 Python 3 迁移过程中，就因为迭代器的这个特性被坑了一次，看看下面这段代码：

```
CYTHON_SOURCES = [a.pyx, b.pyx, c.pyx]
builds = map(BuildProcess, CYTHON_SOURCES)
while any(not build.done() for build in builds):    pending =
[build for build in builds if not build.started()]    <do some
work>
```

这段代码的用处是挨个编译 Cython 源文件。当他们把运行环境切换到 Python 3 后，一个奇怪的问题出现了：CYTHON_SOURCES 中的第一个文件永远都被跳过了编译。为什么呢？

这都是迭代器的锅。在 Python 3 中，`map()` 函数不再返回整个 list，而是返回一个迭代器。

于是，当第二行代码生成 `builds` 这个迭代器后，第三行代码的 `while` 循环迭代了 `builds`，刚好取出了第一个元素。于是之后的 `pending` 对象便里面永远少了那第一个元素。

这个问题解决起来也挺简单的，你只要手动的吧 `builds` 转换成 list 就可以了：

```
builds = list(map(BuildProcess, CYTHON_SOURCES))
```

但是这类 bug 非常难定位到。如果用户的 feeds 里面永远少了那最新的第一条，用户很少会注意到。

字典的顺序

看看下面这段代码：

```
>>> testdict = {'a': 1, 'b': 2, 'c': 3}
>>> json.dumps(testdict)
```

它会输出什么结果呢？

```
# Python2
'{"a": 1, "c": 3, "b": 2}'
# Python 3.5.1
```

```
'{"c": 3, "b": 2, "a": 1}'      # or  
'{"c": 3, "a": 1, "b": 2}'  
# Python 3.6  
'{"a": 1, "b": 2, "c": 3}'
```

在不同的 Python 版本下，这个 json.dumps 的结果是完全不一样的。甚至在 3.5.1 中，它会完全随机的返回两个不同的结果。Instagram 有一段判断配置文件是否发生变动的模块，就是因为这个原因出了问题。

这个问题的解决办法是，在调用 json.dumps 传入 sort_keys=True 参数：

```
>>> json.dumps(testdict, sort_keys=True)  
'{"a": 1, "b": 2, "c": 3}'
```

迁移到 Python 3.6 后的性能提升

当 Instagram 解决了这些奇奇怪怪的版本差异问题后，还有一个巨大的谜题困扰着他们：性能问题。

在 Instagram，他们使用两个主要指标来衡量他们的服务性能：

- 每次请求产生的 CPU 指令数（越低越好）；
- 每秒能够处理的请求数（越高越好）。

所以，当所有的迁移工作完成后，他们非常惊喜的发现：第一个性能指标，每次请求产生的 CPU 指令数居然足足下降了 12% !!!

但是，按理说第二个指标：每秒请求数也应该获得接近 12% 的提升。不过最后的变化却是 0%。究竟是出了什么问题呢？

他们最终定位到，是由于不同 Python 版本下的内存优化配置不同，导致 CPU 指令数下降带来的性能提升被抵消了。那为什么不同 Python 版本下的内存优化配置会不一样呢？

这是他们用来检查 uwsgi 配置的代码：

```
if uwsgi.opt.get('optimize_mem', None) == 'True':    optimize_  
mem()
```

注意到那段 == 'True' 了吗？在 Python 3 中，这个条件

判断总是不会被满足。问题就在于 `unicode`。在将代码中的 `'True'` 换成 `b'True'`（也就是将文本类型换成二进制，这种判断在 Python 2 中完全不区分的）后，问题解决了。

所以，最终因为加上了一个小小的字母 `'b'`，程序的整体性能提升了 12%。

完美切换



图 2

在今年二月份，Instagram 的后端代码的运行环境完全切换到了 Python 3 下：

当所有的代码都都迁移到 Python 3 运行环境后：

- 节约了 12% 的整体 CPU 使用率 (`Django/uwsgi`)；
- 节约了 30% 的内存使用 (`celery`)。

同时，在整个迁移期间，Instagram 的月活用户经历了从 4 亿到 6 亿的巨大增长。产品也发布了评论过滤、直播等非常多新功能。

那么，那几个最开始驱动他们迁移到 Python 3 的目的呢？

- **类型注解：**Instagram 的整个 codebase 里已经有 2% 的代码添加上了类型注解，同时他们还开发了一些工具来辅助开发者添加类型提示。

- `asyncio`: 他们在单个接口中利用 `asyncio` 平行的去做多件事情，最终降低了 20-30% 的请求延迟。
- 社区：他们与 Intel 的工程师联合，帮助他们更好的对 CPU 利用率进行调优。同时还开发了很多新的工具，帮助他们进行性能调优。

Instagram 带给我们的启示

Instagram 的演讲视频时间不长，但是内容很丰富，在编写此文前，我完全没有想到最终的文章会这么长。

那么，Instagram 的视频可以给我们哪些启示呢？

- Python + Django 的组合完全可以负载用户数以 10 亿记的服务，如果你正准备开始一个项目，放心使用 Python 吧！
- 完善的单元测试对于复杂项目是非常有必要的。如果没有那『成千上万的单元测试』。很难想象 Instagram 的迁移项目可以成功进行下去。
- 开发者和同事也是你的产品用户，利用好他们。用他们为你的新特性发布前多一道测试。
- 完全基于主分支的开发流程，可以给你更快的迭代速度。前提是拥有完善的单元测试和持续部署流程。
- Python 3 是大势所趋，如果你正准备开始一个新项目，无需迟疑，拥抱 Python 3 吧！

好了，就到这儿吧。Happy Hacking !

高负载微服务系统的诞生过程

作者 Vadim Madison 译者 薛命灯



在 2016 LighLoad++ 大会上，“M-Tex”的开发经理 Vadim Madison 讲述了从一个由数百个微服务组成的系统到包含数千个微服务的高负载项目的发展历程。

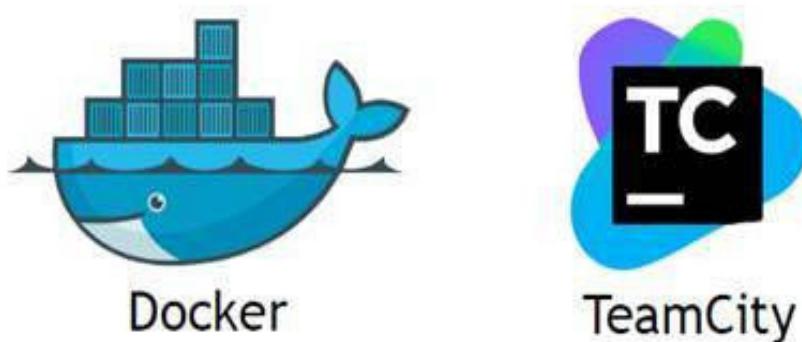
我将告诉大家我们是如何开始一个高负载微服务项目的。在讲述我们的经历之前，先让我们简单地自我介绍一下。

简单地说，我们从事视频输出方面的工作——我们提供实时的视频。我们负责“NTV-Plus”和“Match TV”频道的视频平台。该平台有 30 万的并发用户，每小时输出 300TB 的内容。这是一个很有意思的任务。那么我们是如何做到的呢？

这背后都有哪些故事？这些故事都是关于项目的开发和成长，关于我

们对项目的思考。总而言之，是关于如何提升项目的伸缩能力，承受更大的负载，在不宕机和不丢失关键特性的情况下为客户提供更多的功能。我们总是希望能够满足客户的需求。当然，这也涉及到我们是如何实现这一切，以及这一切是如何开始的。

在最开始，我们有两台运行在 Docker 集群里的服务器，数据库运行在相同机器的容器里。没有专用的存储，基础设施非常简单。

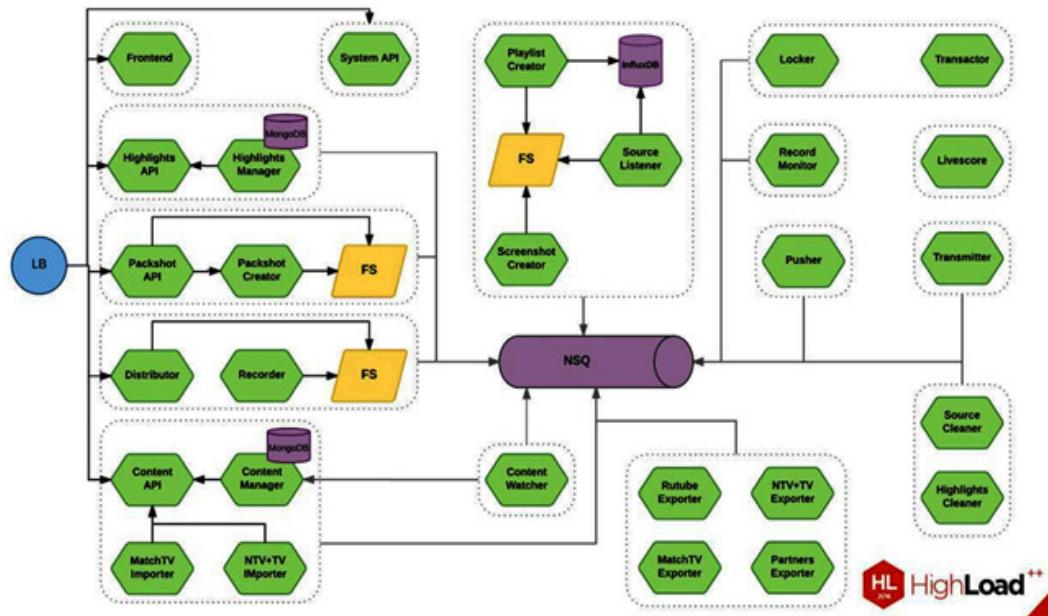


我们就是这样开始的，只有两台运行在 Docker 集群里的服务器。那个时候，数据库也运行在同一个集群里。我们的基础设施里没有什么专用的组件，十分简单。

我们的基础设施最主要的组件就是 Docker 和 TeamCity，我们用它们来交付和构建代码。

在接下来的时期——我称其为我们的发展中期——是我们项目发展的关键时期。我们拥有了 80 台服务器，并在一组特殊的机器上为数据库搭建了一个单独的专用集群。我们开始使用基于 CEPH 的分布式存储，并开始思考服务之间的交互问题，同时要更新我们的监控系统。

现在，让我们来看看我们在这一时期都做了哪些事情。Docker 集群里已经有数百台服务器，微服务就运行在它们上面。这个时候，我们开始根据数据总线和逻辑分离原则将我们的系统拆分成服务子系统。当微服务越来越多时，我们决定拆分我们的系统，这样维护起来就容易得多（也更容易理解）。

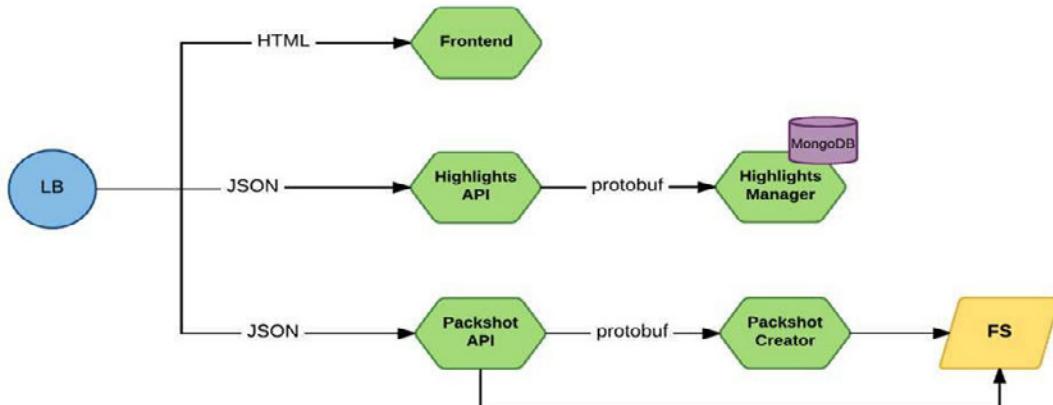


这张图展示的是我们系统其中的一小部分。这部分系统负责视频剪切。半年前，我在“RIT++”也展示过类似的图片。那个时候只有 17 个绿色的微服务，而现在有 28 个绿色的微服务。这些服务只占我们整个系统的二十分之一，所以可以想象我们系统大致的规模有多大。

深入细节

服务间的通信是一件很有趣的事情。一般来说，我们应该尽可能提升服务间通信效率。我们使用了 protobuf，我们认为它就是我们需要的东西。

它看起来是这样的：



微服务的前面有一个负载均衡器。请求到达前端，或者直接发送给提供了 JSON API 的服务。protobuf 被用于内部服务之间的交互。

protobuf 真是一个好东西。它为消息提供了很好的压缩率。现如今有很多框架，只要使用很小的开销就能实现序列化和反序列化。我们可以将其视为有条件的请求类型。

但如果从微服务角度来看，我们会发现，微服务之间也存在某种私有的协议。如果只有一两个或者五个微服务，我们可以为每个微服务打开一个控制台，通过它们来访问微服务，并获得响应结果。如果出现了问题，我们可以对其进行诊断。不过这在一定程度上让微服务的支持工作变得复杂。

在一定时期内，这倒不是什么问题，因为并没有太多的微服务。另外，Google 发布了 gRPC。在那个时候，gRPC 满足了所有我们想做的事情。于是我们逐渐迁移到 gRPC。于是我们的技术栈里出现了另一个组件。



实现的细节也是很有趣的。gRPC 默认是基于 HTTP/2 的。如果你的环境相对稳定，应用程序不怎么发生变更，也不需要在机器间迁移，那么 gRPC 对于你来说就是个不错的东西。另外，gRPC 支持很多客户端和服务端的编程语言。

现在，我们从微服务角度来看待这个问题。从一方面来看，gRPC 是一个好东西，但从另一方面来看，它也有不足之处。当我们开始对日志进行标准化（这样就可以将它们聚合到一个独立的系统里）时，我们发现，从 gRPC 中抽取日志非常麻烦。

于是，我们决定开发自己的日志系统。它解析消息，并将它们转成我们需要的格式。这样我们才可以获得我们想要的日志。还有一个问题，添

加新的微服务会让服务间的依赖变得更加复杂。这是微服务一直存在的问题，这也是除版本问题之外的另一个具有一定复杂性的问题。

于是，我们开始考虑使用 JSON。在很长的一段时间里，我们无法相信，在使用了紧凑的二进制协议之后会转回使用 JSON。有一天，我们看到一篇文章，来自 DailyMotion 的一个家伙在文章里提到了同样的事情：“我们知道该如何使用 JSON，每个人都可以使用 JSON。既然如此，为什么还要自寻烦恼呢？”



于是，我们逐渐从 gRPC 转向我们自己实现的 JSON。我们保留了 HTTP/2，它与 JSON 组合起来可以带来更快的速度。

现在，我们具备了所有必要的特性。我们可以通过 cURL 访问我们的服务。我们的 QA 团队使用 Postman，所以他们也感觉很满意。一切都变得简单起来。这是一个有争议性的决定，但却为我们带来了很多好处。

JSON 唯一的缺点就是它的紧凑性不足。根据我们的测试结果，它与 MessagePack 之间有 30% 的差距。不过对于一个支持系统来说，这不算是个大问题。

况且，我们在转到 JSON 之后还获得了更多的特性，比如协议版本。有时候，当我们在新版本的协议上使用 protobuf 时，客户端也必须改用 protobuf。如果你有数百个服务，就算只有 10% 的服务进行了迁移，这也会引起很大的连锁反应。你在一个服务上做了一些变更，就会有十多个服务也需要跟着改动。

因此，我们就会面临这样的一种情况，一个服务的开发人员已经发布了第五个、第六个，甚至第七个版本，但生产环境里仍然在运行第四个版

本，就因为其他相关服务的开发人员有他们自己的优先级和截止日期。他们无法持续地更新他们的服务，并使用新版本的协议。所以，新版本的服务虽然发布了，但还派不上用场。然后，我们却要以一种很奇怪的方式来修复旧版本的 bug，这让支持工作变得更加复杂。

最后，我们决定停止发布新版本的协议。我们提供协议的基础版本，可以往里面添加少量的属性。服务的消费者开始使用 JSON schema。

标准看起来是这样的：

Protocol Versioning

V1, V2, ... → V1 + schema

```
/api/v1/content  
/api/v2/content      →      /api/v1/content + JSON Schema v1.X  
/api/v3/content
```

我们没有使用版本 1、2 和 3，而是只使用版本 1 和指向它的 schema。

这是从我们服务返回的一个典型的响应结果。它是一个内容管理器，返回有关广播的信息。这里有一个消费者 schema 的例子。

最底下的字符串最有意思，也就是“required”那块。我们可以看到，这个服务只需要 4 个字段——id、content、date 和 status。如果我们使用了这个 schema，那么消费者就只会得到这样的数据。

它们可以被用在每一个协议版本里，从第一个版本到后来的每一个变更版本。这样，在版本之间迁移就容易很多。在我们发布新版本之后，客户端的迁移就会简单很多。

下一个重要的议题是系统的稳定性问题。这是微服务和其他任何一个系统都需要面临的问题（在微服务架构里，我们可以更强烈地感觉到它的

```
{
  "id": "507f1f77bcf86cd7994390",
  "projectId": "507f1f77bcf86cd7994390",
  "content": "broadcast",
  "name": "Children interrupt BBC News interview",
  "date": {
    "start": "2005-08-09T18:31:42-03:30",
    "end": "2005-08-09T18:31:42-03:30"
  },
  "source": "http://.../playlist.m3u8",
  "extra": {
    "videoType": "funny",
    "description": "There was an unexpected
      distraction for Professor Robert Kelly
      when he was being interviewed live on
      BBC News about South Korea. But he
      managed to keep his composure and
      complete the interview successfully."
  },
  "listeningStatus": "fail",
  "status": "enabled"
}
```

重要性）。系统总会在某个时候变得不稳定。

如果服务间的调用链只包含了一两个服务，那么就没有什么问题。在这种情况下，你看不出单体和分布式系统之间有多大区别。但当调用链里包含了 5 到 7 个调用，那么问题就会接踵而至。你根本不知道为什么会这样，也不知道能做些什么。在这种情况下，调试会变得很困难。在单体系统里，你可以通过逐步调试来找出错误。但对于微服务来说，网络不稳定性或高负载下的性能不稳定性也会对微服务造成影响。特别是对于拥有大量节点的分布式系统来说，这些情况就更加显而易见了。

在一开始，我们采用了传统的办法。我们监控所有的东西，查看问题和问题的发生点，然后尝试尽快修复它们。我们将微服务的度量指标收集

```
{  
  "$schema": "...",  
  "type": "object",  
  "properties": {  
    "id": { "type": "string" },  
    "content": { "type": "string" },  
    "date": {  
      "type": "object",  
      "properties": {  
        "start": { "type": "string" },  
        "end": { "type": "string" }  
      },  
      "required": ["start", "end"]  
    },  
    "source": { "type": "string" },  
    "status": { "type": "string" },  
    ...  
  },  
  "required": ["id", "content",  
              "date", "status"]  
}  
  
{  
  "id": "507f1f77bcf86cd7994390",  
  "content": "broadcast",  
  "date" : {  
    "start": "2005-08-09T18:31:42-03:30",  
    "end": "2005-08-09T18:31:42-03:30"  
  },  
  "status": "enabled"  
}
```

到一个独立的数据库里。我们使用 Diamond 来收集系统度量指标。我们使

用 cAdvisor 来分析容器的资源使用情况和性能特征。所有的结果都被保存到 InfluxDB，然后我们在 Grafana 里创建仪表盘。

于是，我们现在的基础设施里又多了三个组件。



我们比以往更加关注所发生的一切。我们对问题的反应速度更快了。不过，这并没有阻止问题的出现。

奇怪的是，微服务架构的主要问题出在那些不稳定的服务上。它们有的今天运行正常，明天就不行，而且有各种各样的原因。如果服务出现超载，而你继续向它发送负载，它就会宕机一段时间。如果它在一段时间不提供服务，负载就会下降，然后它就又活过来了。这类系统很难维护，也很难知道到底出了什么问题。

最后，我们决定把这些服务停掉，而不是让它们来回折腾。我们因此需要改变服务的实现方式。

我们做了一件很重要的事情。我们对每个服务接收的请求数量设定了一个上限。每个服务知道自己可以处理多少个来自客户端的请求（我们稍后会详细说明）。如果请求数量达到上限，服务将抛出 503 Service Unavailable 异常。客户端知道这个节点无法提供服务，就会选择另一个节点。

当系统出现问题时，我们就可以通过这种方式来减少请求时间。另外，我们也提升了服务的稳定性。

我们引入了第二种模式——回路断路器（Circuit Breaker）。我们在客户端实现了这种模式。

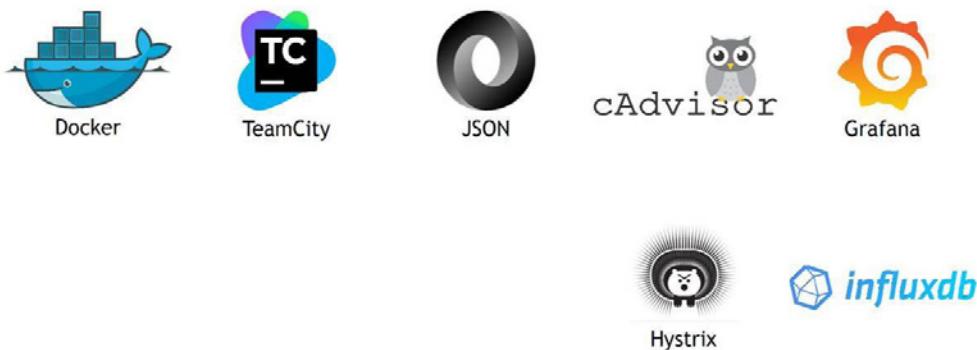
假设有一个服务 A，它有 4 个可以访问的服务 B 的实例。它向注册中

心索要服务 B 的地址：“给我这些服务的地址”。它得到了服务 B 的 4 个地址。服务 A 向第一个服务 B 的实例发起了请求。第一个服务 B 实例正常返回响应。服务 A 将其标记为可访问：“是的，我可以访问它”。然后，服务 A 向第二个服务 B 实例发起请求，不过它没有在期望的时间内得到响应。我们禁用了这个实例，然后向下一个实例发起请求。下一个实例因为某些原因返回了不正确的协议版本。于是我们也将其禁用，然后转向第四个实例。

总得来说，只有一半的服务能够为客户端提供服务。于是服务 A 将会向能够正常返回响应的两个服务发起请求。而另外两个无法满足要求的实例被禁用了一段时间。

我们通过这种方式来提升性能的稳定性。如果服务出现了问题，我们就将其关闭，并发出告警，然后尝试找出问题所在。

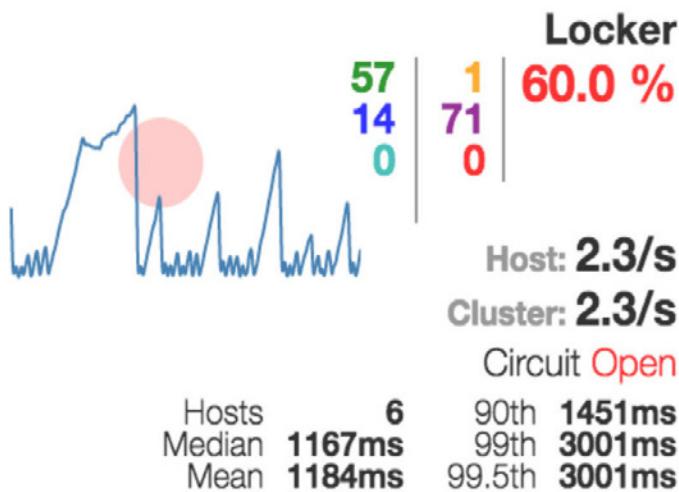
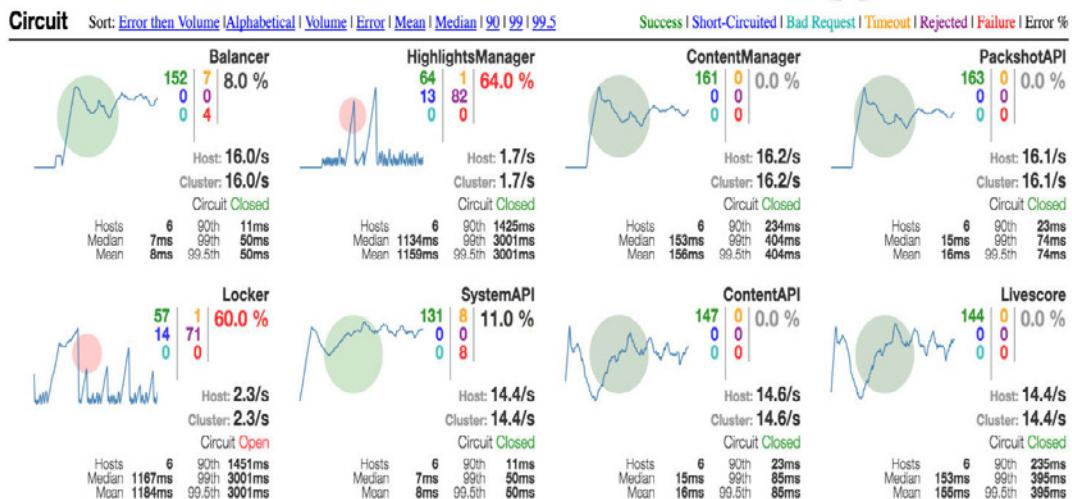
因为引入了回路断路器模式，我们的基础设施里又多了一个组件——[Hystrix](#)。



Hystrix 不仅实现了回路断路器模式，它也有助于我们了解系统里出现了哪些问题：

圆环的大小表示服务与其他组件之间的流量大小。颜色表示系统的健康状况。如果圆环是绿色的，那么说明一切正常。如果圆环是红色的，那么就有问题了。

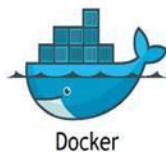
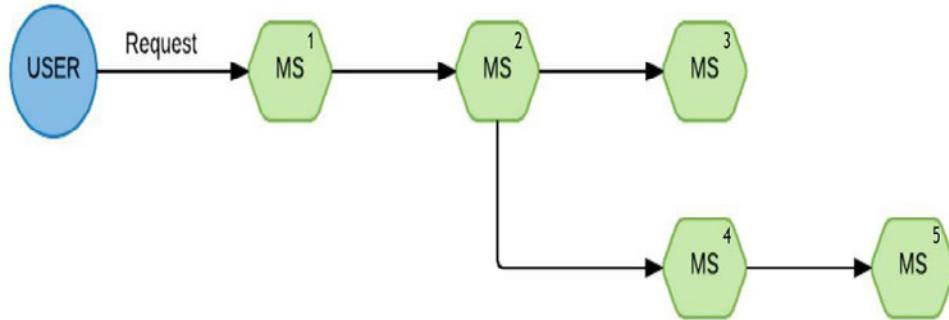
如果一个服务应该被停掉，那么它看起来是这个样子的。圆环是打开的。



我们的系统变得相对稳定。每个服务至少都有两个可用的实例，这样我们就可以选择停掉其中的一个。不过，尽管是这样，我们仍然不知道我们的系统究竟发生了什么问题。在处理请求期间如果出现了问题，我们应该怎样才能知道问题的根源是什么呢？

这是一个标准的请求：

这是一个处理链条。用户发送请求到第一个服务，然后是第二个。从第二个服务开始，链条将请求发送到第三个和第四个服务。



Docker



TeamCity



JSON



cAdvisor



Grafana



ZIPKIN



Hystrix



然后一个分支不明原因地消失了。在经历了这类场景之后，我们尝试着提升这种场景的可见性，于是我们找到了 Appdash。Appdash 是一个跟踪服务。



Docker



TeamCity



JSON



cAdvisor



Grafana

appdash

Appdash



Hystrix

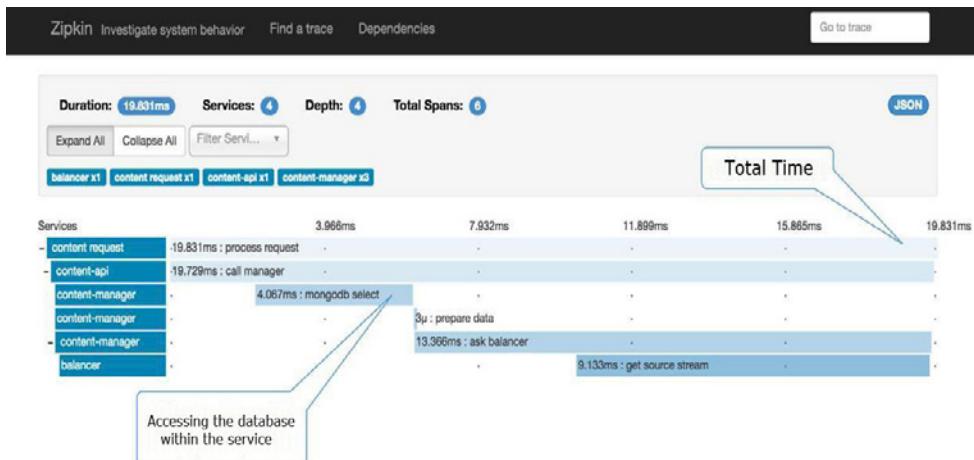


它看起来是这个样子的。

可以这么说，我们只是想尝试一下，看看它是否适合我们。将它用在我们的系统里是一件很容易的事情，因为我们那个时候使用的是 Go 语言。Appdash 提供了一个开箱即用的包。我们认为 Appdash 是一个好东西，只

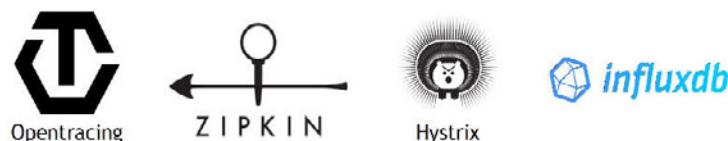
是它的实现并不是很适合我们。

于是，我们决定使用 Zipkin 来代替 Appdash。Zipkin 是由 Twitter 开源的。它看起来是这个样子的。



我认为这样会更清楚一些。我们可以从中看到一些服务，也可以看到我们的请求是如何通过请求链的，还可以看到请求在每个服务里都做了哪些事情。一方面，我们可以看到服务的总时长和每个分段的时长，另一方面，我们完全可以添加描述服务内容的信息。

我们可以在这里添加一些与数据库的调用、文件系统的读取、缓存的访问有关的信息，这样就可以知道请求里哪一部分使用了最多的时间。TraceID 可以帮助我们做到这一点。稍后我会介绍更多细节。

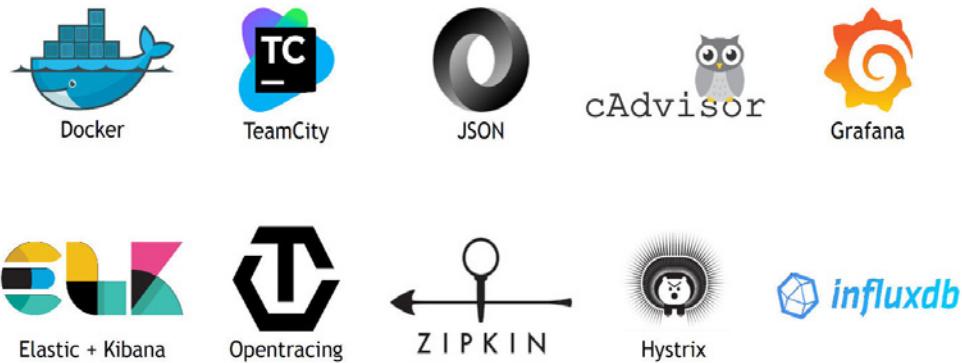


我们就是通过这种方式知道请求在处理过程中发生了什么问题，以及为什么有时候无法被正常处理。刚开始一切都正常，然后突然间，其中的

一个出现了问题。我们稍作排查，就知道出问题的服务发生了什么。

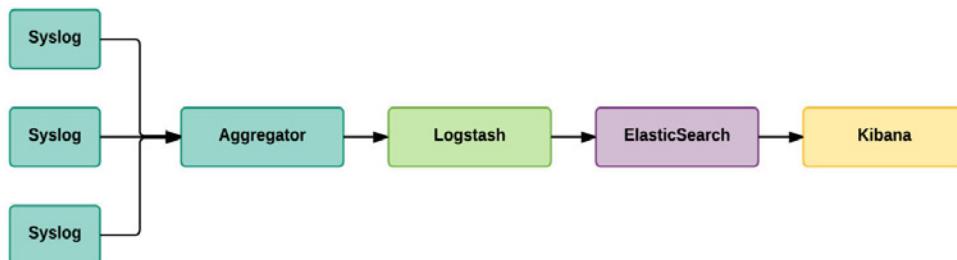
不久前，一些厂商推出了一个跟踪系统的标准。为了简化系统的实现，主要的几个跟踪系统厂商在如何设计客户端 API 和客户端类库上达成了一致。现在已经有了 OpenTracing 的实现，支持几乎所有的主流开发语言。现在就可以使用它了。

我们已经有办法知道那些突然间崩溃的服务。我们可以看到其中的某部分在垂死挣扎，但是不知道为什么。光有环境信息是不够的，我们还需要日志。是的，这应该成为标准的一部分，它就是 Elasticsearch、Logstash 和 Kibana (ELK)。不过我们对它们做了一些改动。



我们并没有将大量的日志直接通过 forward 传给 Logstash，而是先传给 syslog，让它把日志聚合到构建机器上，然后再通过 forward 导入到 Elasticsearch 和 Kibana。这是一个很标准的流程，那么巧妙的地方在哪里呢？

Logging: Collecting Logs



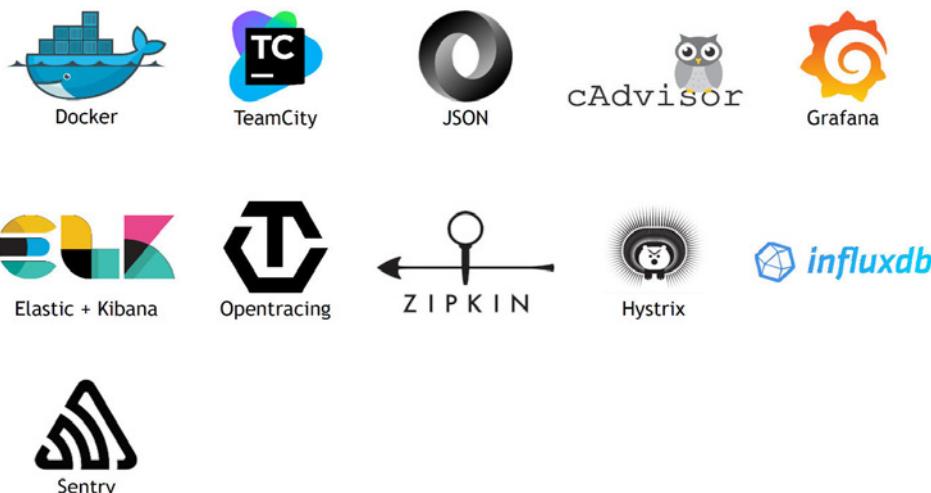
巧妙的是，我们可以在任何可能的地方往日志里加入 Zipkin 的 TraceID。

这样一来，我们就可以在 Kibana 仪表盘上看到完整的用户请求执行情况。也就是说，一旦服务进入生产环境，就为运营做好了准备。它已经通过了自动化测试，如果有必要，QA 可以再进行手动检查。它应该没有什么问题。如果它出现了问题，那说明有一些先决条件没有得到满足。日志里详细地记录了这些先决条件，通过过滤，我们可以看到某个请求的跟踪信息。我们因此可以快速地查出问题的根源，为我们节省了很多时间。

我们后来引入了动态调试模式。现在的日志数量还不是很大，大概只有 100 GB 到 150 GB，我记不太清楚具体数字了。不过，这些日志是在正常的日子模式下生成的。如果我们添加更多的细节，那么日志就可能变成 TB 级别的，处理起来就很耗费资源。

当我们发现某些服务出现问题，就打开调试模式（通过一个 API），看看发生了什么事情。有时候，我们找到出现问题的服务，在不将它关闭的情况下打开调试模式，尝试找出问题所在。

最后，我们在 ELK 端查找问题。我们还对关键服务的错误进行聚合。服务知道哪些错误是关键性的，哪些不是关键性的，然后将它们传给 [Sentry](#)。

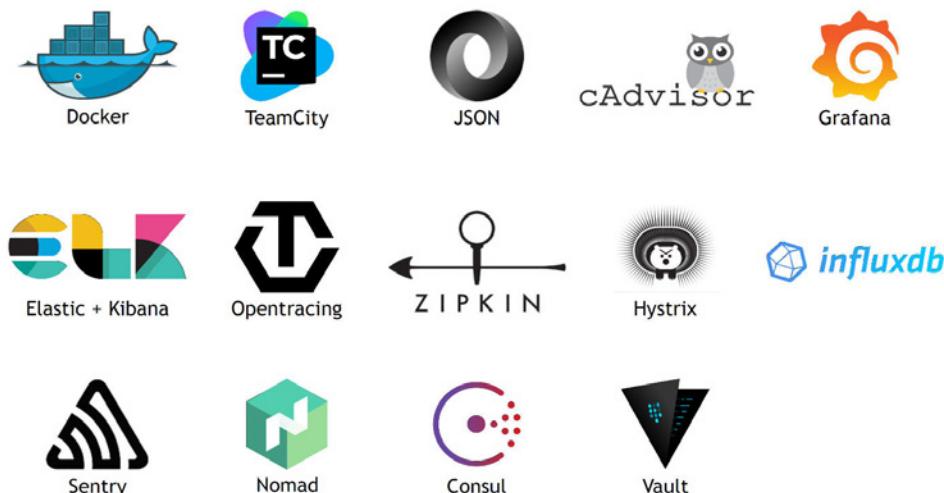


Sentry 能够智能地收集错误日志，并形成度量指标，还会进行一些基本的过滤。我们在很多服务上使用了 Sentry。我们从单体应用时期就开始使用它了。

那么最有趣的问题是，我们是如何进行伸缩的？这里需要先介绍一些概念。我们把每个机器看成一个黑盒。



我们有一个编排系统，最开始使用 [Nomad](#)。确切地说，应该是 [Ansible](#)。我们自己编写脚本，但光是这些还不能满足要求。那个时候，Nomad 的某些版本可以简化我们的工作，于是我们决定迁移到 Nomad。



同时还使用了 [Consul](#)，将它作为服务发现的注册中心。还有 Vault，用于存储敏感数据，比如密码、秘钥和其他所有不能保存在 Git 上的东西。

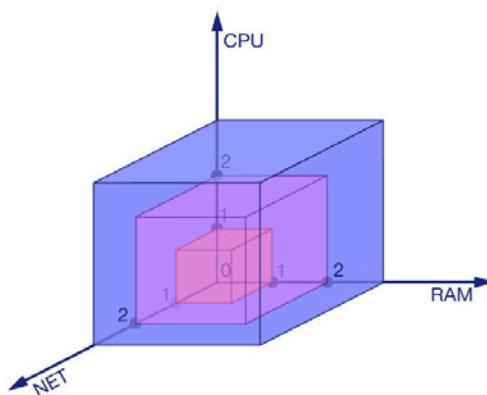
这样，所有的机器几乎都变得一模一样。每个机器上都安装了 Docker，还有 Consul 和 Nomad 代理。总的来说，每一个机器都处于备用状态，可以在任何时候投入使用。如果不用了，我们就让它们下线。如果你构建了云平台，你就可以先准备好机器，在高峰期时将它们打开，在负载下降时将它们关闭。这会节省大量的成本。

后来，我们决定从 Nomad 迁移到 Kubernetes，Consul 也因此成为了集中式的配置系统。

这样一来，部分栈可以进行自动伸缩。那么我们是怎么做的呢？

第一步，我们对内存、CPU 和网络进行限制。

Standardization: RAM-CPU-NET

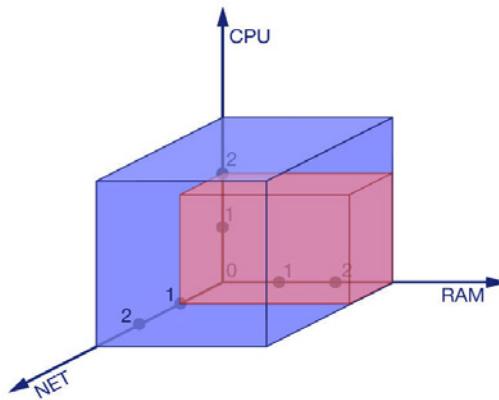


我们分别将这三个元素分成三个等级，砍掉其中的一部分。例如，R3-C2-N1，我们已经限定只给某个服务一小部分网络流量、多一点点的 CPU 和更多的内存。这个服务真的很耗费资源。

我们在这里使用了助记符，我们的决策服务可以设置很多的组合值，这些值看起来是这样的：

事实上，我们还有 C4 和 R4，不过它们已经超出了这些标准的限制。标准看起来是这样的：

Standardization: R3-C2-N1



Standardization: Dimension Matrix

CPU	RAM	NET
C1 = 500 MHz	R1 = 128 MB	N1 = 10 Mb
C2 = 1000 MHz	R2 = 256 MB	N2 = 100 Mb
C3 = 3000 Mhz	R3 = 512 MB	N3 = 1 Gb

Standardization

- N1C3R1 → [10 Mb] [3000 MHz] [128 MB]
- N1C2R3 → [10 Mb] [1000 MHz] [1 GB]

下一步开始做一些预备工作。我们先确定服务的伸缩类型。

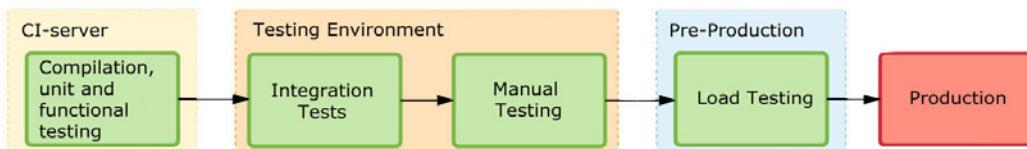
独立的服务最容易伸缩，它可以进行线性地伸缩。如果用户增长了两倍，我们就运行两倍的服务实例。这就万事大吉了。

第二种伸缩类型：服务依赖了外部的资源，比如那些使用了数据库的服务。数据库有它自己的容量上限，这个一定要注意。你还要知道，如果

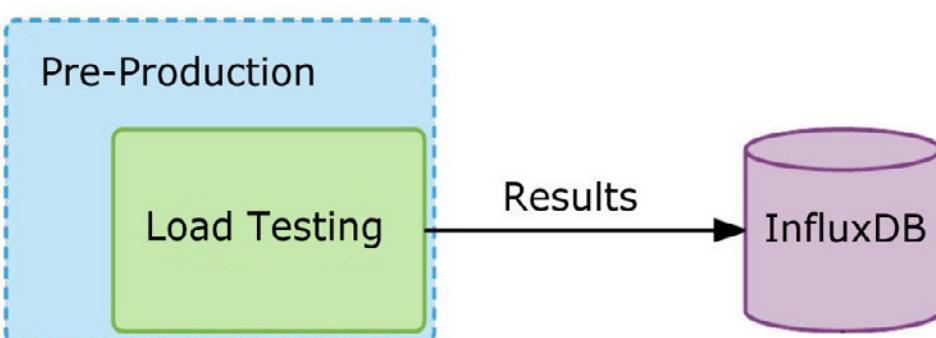
系统性能出现衰退，就不应该再增加更多的实例，而且你要知道这种情况会在什么时候发生。

第三种情况是，服务受到外部系统的牵制。例如，外部的账单系统。就算运行了 100 个服务实例，它也没办法处理超过 500 个请求。我们要考虑到这些限制。在确定了服务类型并设置了相应的标记之后，是时候看看它们是如何通过我们的构建管道的。

Preparation: Load Testing



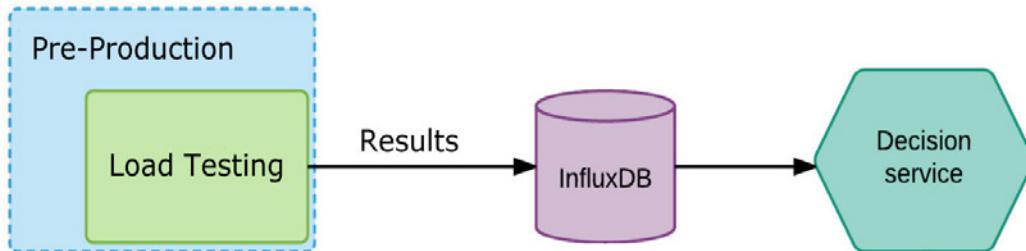
我们在 CI 服务器上运行了一些单元测试，然后在测试环境运行集成测试，我们的 QA 团队会对它们做一些检查。在这之后，我们就进入了预生产环境的负载测试。



如果是第一种类型的服务，我们使用一个实例，并在这个环境里运行它，给它最大的负载。在运行了几轮之后，我们取其中的最小值，将它存入 InfluxDB，将它作为该服务的负载上限。

如果是第二种类型的服务，我们逐渐加大负载，直到出现了性能衰退。我们对这个过程进行评估，如果我们知道该系统的负载，那么就比较当前

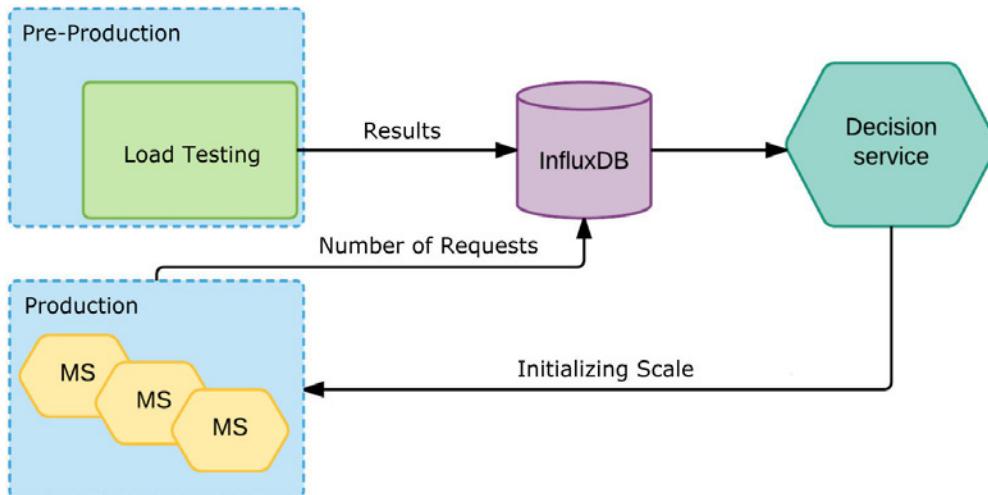
负载是否已经足够，否则，我们就会设置告警，不会把这个服务发布到生产环境。我们会告诉开发人员：“你们需要分离出一些东西，或者加进去另一个工具，让这个服务可以更好地伸缩。”



因为我们知道第三种类型服务的上限，所以我们只运行一个实例。我们也会给它一些负载，看看它可以服务多少个用户。如果我们知道账单系统的上限是 1000 个请求，并且每个服务实例可以处理 200 个请求，那么就需要 5 个实例。

我们把这些信息都保存到了 InfluxDB。我们的决策服务开始派上用场了。它会检查两个边界：上限和下限。如果超出了上限，那么就应该增加服务实例。如果超出下限，那么就减少实例。如果负载下降（比如晚上的时候），我们就不需要这么多机器，可以减少它们的数量，并关掉一部分机器，省下一些费用。

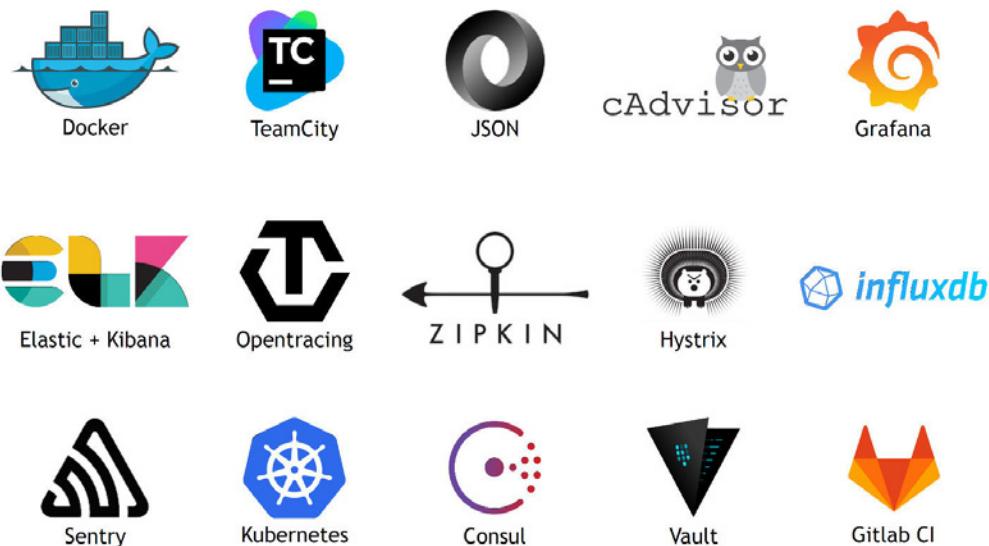
整体看起来是这样的：



每个服务的度量指标表明了它们当前的负载。负载信息被保存到 InfluxDB，如果决策服务发现服务实例达到了上限，它会向 Nomad 和 Kubernetes 发送命令，要求增加服务实例。有可能在云端已经有可用的实例，或者开始做一些准备工作。不管怎样，发出要求增加新服务实例的告警才是关键所在。

一些受限的服务如果达到上限，也会发出相关的告警。对于这类情况，我们除了加大等待队列，也做不了其他什么事情。不过最起码我们知道我们很快就会面临这样的问题，并开始做好应对措施。

这就是我想告诉大家有关伸缩性方面的事情。除了这些，还有另外一个东西——[Gitlab CI](#)。



我们一般是通过 TeamCity 来开发服务的。后来，我们意识到，所有的服务都有一个共性，这些服务都是不一样的，并且知道自己该如何部署到容器里。要生成这么的项目真的很困难，不过如果使用 yml 文件来描述它们，并把这个文件与服务放在一起，就会方便很多。虽然我们只做了一些小的改变，不过却为我们带来了非常多的可能性。

现在，我想说一些一直想对自己说的话。

关于微服务开发，我建议一开始就使用编排系统。可以使用最简单的编排系统，比如 Nomad，通过 nomad agent -dev 命令启动一个编排系统，

包括 Consul 和其他东西。

我们仿佛是在一个黑盒子工作。你试图避免被绑定到某台特定的机器上，或者被附加到某台特定机器的文件系统上。这些事情会让你开始重新思考。

在开发阶段，每个服务至少需要两个实例，如果其中一个出现问题，就可以关掉它，由另一个接管继续服务。

还有一些有关架构的问题。在微服务架构里，消息总线是一个非常重要的组件。

假设你有一个用户注册系统，那么如何以最简单的方式实现它呢？对于注册系统来说，需要创建账户，然后在账单系统里创建一个用户，并为他创建头像和其他东西。你有一组服务，其中的超级服务收到了一个请求，它将请求分发给其他服务。经过几次之后，它就知道该触发哪些服务来完成注册。

不过，我们可以使用一种更简单、更可靠、更高效的方式来实现。我们使用一个服务来处理注册，它注册了一个用户，然后发送一个事件到消息总线，比如“我已经注册了一个 yoghurt，ID 是……”。相关的服务会收到这个事件，其中的一个服务会在账单系统里创建一个账户，另一个服务会发送一封欢迎邮件。

不过，系统会因此失去强一致性。这个时候你没有超级服务，也不知道每个服务的状态。不过，这样的系统很容易维护。

现在，我再说一些之前提到过的问题。不要试图修复出问题的服务。如果某些服务实例出现了问题，将它找出来，然后把流量定向到其他服务实例（可能是新增的实例）上，然后再诊断问题。这样可以显著提升系统的可用性。

通过收集度量指标来了解系统的状态自然不在话下。

不过要注意，如果你对某个度量指标不了解，不知道怎么使用它，或者它对你来说没有什么意义，就不要收集它。因为有时候，这样的度量指标会有数百万个。你在这些无用的度量指标上面浪费了很多资源和时间。

这些是无效的负载。

如果你认为你需要某些度量指标，那么就收集它们。如果不需要，就不要收集。

如果你发现了一个问题，不要急着去修复。在很多情况下，系统会对此作出反应。当系统需要你采取行动的时候，它会给你发出告警。如果它不要求你在半夜跑去修复问题，那么它就不算是一个告警。它只不过是一种警告，你可以在把它当成一般的问题来处理。

一语点醒技术人：你不是 Google

作者 Ozan Onay 译者 薛命灯



在为问题寻找解决方案时要先充分了解问题本身，而不是一味地盲目崇拜那些巨头公司。Ozan Onay 以 Amazon、LinkedIn 和 Google 为例，为执迷不悟的人敲响警钟。以下内容已获得作者翻译授权，查看原文：[You Are Not Google](#)。

软件工程师总是着迷于荒唐古怪的事。我们看起来似乎很理性，但在面对技术选型时，总是陷入抓狂——从 Hacker News 到各种博客，像一只飞蛾一样，来回折腾，最后精疲力尽，无助地飞向一团亮光，跪倒在它的前面——那就是我们一直在寻找的东西。

真正理性的人不是这样做决定的。不过工程师一贯如此，比如决定是否使用 MapReduce。

Joe Hellerstein 在他的大学数据库教程视频中说道：

世界上只有差不多 5 个公司需要运行这么大规模的作业。至于其他公司……他们使用了所有的 I/O 来实现不必要的容错。在 2000 年代，人们狂

热地追随着 Google：“我们要做 Google 做过的每一件事，因为我们也运行着世界上最大的互联网数据服务。”

超出实际需求的容错没有什么问题，但我们却为此付出了惨重的代价：不仅增加了 I/O，还有可能让原先成熟的系统——包含了事务、索引和查询优化器——变得破碎不堪。这是一个多么严重的历史倒退！有多少个 Hadoop 用户是有意识地做出这种决定的？有多少人知道他们的决定到底是不是一个明智之举？

MapReduce 已经成为一个众矢之的，那些盲目崇拜者也意识到事情不对劲。但这种情况却普遍存在：虽然你使用了大公司的技术，但你的情况却与他们大不一样，而且你的决定并没有经过深思熟虑，你只是习以为常地认为，模仿巨头公司就一定也能给你带来同样的财富。

是的，这又是一篇劝大家“不要盲目崇拜”的文章。不过这次我列出了一长串有用的清单，或许能够帮助你们做出更好的决定。

很酷的技术？UNPHAT

如果你还在使用 Google 搜索新技术来重建你的软件架构，那么我建议你不要再这么做了。相反，你可以考虑应用 UNPHAT 原则。

1. 在彻底了解（Understand）你的问题之前，不要急着去寻找解决方案。你的目标应该是在问题领域内“解决”问题，而不是在方案领域内解决问题。
2. 列出（eNumereate）多种方案，不要只把眼睛盯在你最喜欢的方案上。
3. 选择一个候选方案，并阅读相关论文（Paper）。
4. 了解候选方案的产生背景（Historical context）。
5. 比较优点（Advantages）和缺点，扬长避短。
6. 思考（Think）！冷静地思考候选方案是否适合用于解决你的问题。要出现怎样异常的情况才会让你改变主意？例如，数据要少到什么程度才会让你打消使用Hadoop的念头？

你不是 Amazon

UNPHAT 原则十分直截了当。最近我与一个公司有过一次对话，这个公司打算在一个读密集的系统里使用 Cassandra，他们的数据是在夜间加载到系统里的。

他们阅读了 Dynamo 的相关论文，并且知道 Cassandra 是最接近 Dynamo 的一个产品。我们知道，这些分布式数据库优先保证写可用性（Amazon 是不会让“添加到购物车”这种操作出现失败的）。为了达到这个目的，他们在一致性以及几乎所有在传统 RDBMS 中出现过的特性上做出了妥协。但这家公司其实没有必要优先考虑写可用性，因为他们每天只有一次写入操作，只是数据量比较大。

他们之所以考虑使用 Cassandra，是因为 PostgreSQL 查询需要耗费几分钟的时间。他们认为是硬件的问题，经过排查，我们发现数据表里有 5000 万条数据，每条数据最多 80 个字节。如果从 SSD 上整块地读取所有数据大概需要 5 秒钟，这个不算快，但比起实际的查询，它要快上两个数量级。

我真的很想多问他们几个问题（了解问题！），在问题变得愈加严重时，我为他们准备了 5 个方案（列出多个候选方案！），不过很显然，Cassandra 对于他们来说完全是一个错误的方案。他们只需要耐心地做一些调优，比如对部分数据重新建模，或许可以考虑使用（当然也有可能没有）其他技术……但一定不是这种写高可用的键值存储系统，Amazon 当初创建 Cassandra 是用来解决他们的购物车问题的！

你不是 LinkedIn

我发现一个学生创办的小公司居然在他们的系统里使用 Kafka，这让我感到很惊讶。因为据我所知，他们每天只有很少的事务需要处理——最好的情况下，一天最多只有几百个。这样的吞吐量几乎可以直接记在记事本上。

Kafka 被设计用于处理 LinkedIn 内部的吞吐量，那可是一个天文数字。即使是在几年前，这个数字已经达到了每天数万亿，在高峰时段每秒钟需要处理 1000 万个消息。不过 Kafka 也可以用于处理低吞吐量的负载，或许再低 10 个数量级？

或许工程师们在做决定时确实是基于他们的预期需求，并且也很了解 Kafka 的适用场景。但我猜测他们是抵挡不住社区对 Kafka 的追捧，并没有仔细想过 Kafka 是否适合他们。要知道，那可是 10 个数量级的差距！

再一次，你不是 Amazon

比 Amazon 的分布式数据库更为著名的是它的可伸缩架构模式，也就是面向服务架构。Werner Vogels 在 2006 年的一次访谈中指出，Amazon 在 2001 年时就意识到他们的前端需要横向伸缩，而面向服务架构有助于他们实现前端伸缩。工程师们面面相觑，最后只有少数几个工程师着手去做这件事情，而几乎没有人愿意将他们的静态网页拆分成小型的服务。

不过 Amazon 还是决定向 SOA 转型，他们当时有 7800 个员工和 30 亿美元的销售规模。

当然，并不是说你也要等到有 7800 个员工的时候才能转向 SOA……只是你要多想想，它真的能解决你的问题吗？你的问题的根源是什么？可以通过其他的方式解决它们吗？

如果你告诉我说，你那 50 个人的公司打算转向 SOA，那么我不禁感到疑惑：为什么很多大型的公司仍然在乐此不彼地使用具有模块化的大型单体应用？

甚至 Google 也不是 Google

使用 Hadoop 和 Spark 这样的大规模数据流引擎会非常有趣，但在很多情况下，传统的 DBMS 更适合当前的负载，有时候数据量小到可以直接放进内存。你是否愿意花 10,000 美金去购买 1TB 的内存？如果你有十亿个用户，每个用户仅能使用 1KB 的内存，所以你的投入远远不够。

或许你的负载大到需要把数据写回磁盘。那么你需要多少磁盘？你到底有多少数据量？Google 之所以要创建 GFS 和 MapReduce，是要解决整个 Web 的计算问题，比如重建整个 Web 的搜索索引。

或许你已经阅读过 GFS 和 MapReduce 的论文，Google 的部分问题在于吞吐量，而不是容量，他们之所以需要分布式的存储，是因为从磁盘读取字节流要花费太多的时间。那么你在 2017 年需要使用多少设备吞吐量？你一定不需要像 Google 那么大的吞吐量，所以你可能会考虑使用更好的设备。如果都用上 SSD 会给你增加多少成本？

或许你还想要伸缩性。但你有仔细算过吗，你的数据增长速度会快过 SSD 降价的速度吗？在你的数据撑爆所有的机器之前，你的业务会有多少增长？截止 2016 年，Stack Exchange 每天要处理 2 亿个请求，但是他们只用了 4 个 SQL Server，一个用于 Stack Overflow，一个用于其他用途，另外两个作为备份副本。

或许你在应用 UNPHAT 原则之后，仍然决定要使用 Hadoop 或 Spark。或许你的决定是对的，但关键的是你要用对工具。Google 非常明白这个道理，当他们意识到 MapReduce 不再适合用于构建索引之后，他们就不再使用它。

先了解你的问题

我所说的也不是什么新观点，不过或许 UNPHAT 对于你们来说已经足够了。如果你觉得还不够，可以听听 Rich Hickey 的演讲“吊床驱动开发”，或者看看 Polya 的书《How to Solve It》，或者学习一下 Hamming 的课程“The Art of Doing Science and Engineering”。我恳请你们一定要多思考！在尝试解决问题之前先对它们有充分的了解。最后送上 Polya 的一个金句名言：

回答一个你不了解的问题是愚蠢的，到达一个你不期望的终点是悲哀的。



CNUTCon 2017

全球运维技术大会

上海·光大会展中心大酒店 | 2017年9月10-11日

部分出品人



曲显平
百度运维部技术经理



张晓强
携程基础架构运维总监



肖世广
腾讯QQ技术运营总监



徐巍
饿了么高级运维经理



涂彦
腾讯游戏运维总监



刘建
搜狗资深架构师



王旭
HYPER COFOUNDER & CTO



许晓斌
阿里巴巴高级技术专家

.....

部分精彩议题

搜狗配置中心架构演化与实践

郭理勇 搜狗资深高级工程师

机器学习模型训练的KUBERNETES实践

袁晓沛 七牛云技术总监

从理论到实践，深度解析 MYSQL GROUP REPLICATION

徐春阳 民生银行科技部项目经理

机器学习在大规模服务器治理 复杂场景的实践

陈立波 阿里巴巴高级技术专家

基于资产配置业务场景下的全链路监控平台

王晔倞 好买财富技术总监

构建微服务下的性能监控

吴晟 华为软件开发云分布式应用性能监控产品专家

麻袋理财安全与合规建设

王耀 麻袋理财首席安全官

京东物流系统自动化运维平台技术揭秘

赵玉开 京东资深架构师

苏宁大数据平台运维实践

王志强 苏宁云商IT总部技术总监

阿里一键建站技术解密

谢吉宝（唐三） 阿里巴巴高级技术专家

7月29日前购票立减**720**元（团购更优惠）

Geekbang

InfoQ



售票咨询 18504256269 hedy.hu@geekbang.org





本期主要内容：“从此社区再无 Docker？”
那“Moby”又是什么？Kotlin 成为正式的
Android 编程语言；如何构建一套高可用的
移动消息推送平台？阿里巴巴为什么要选择星
际争霸作为 AI 算法研究环境？人工智能与软
件架构



架构师特刊 大前端

本期主要内容：当我们在谈大前端的时候，我们谈的是什么；如何落地和管理一个“大前端”团队？



顶尖技术团队访谈录 第九季

本次的《中国顶尖技术团队访谈录》·第八季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



架构师特刊 用户画像实践

本电子书中几个作者介绍一个公司如何从无到有的搭建用户画像系统，以及其中的技术难点与实际操作中的注意事项，实为用户画像的实操精华之选，推荐各位收藏阅读。