架构师

ARCHITECT

月刊

推荐文章

甲骨文：有史以来最伟大的25个Java应用程序

Geekbang. 极客邦科技 | InfoQ

CONTENTS / 目录

架构师

2020 年 8 月刊

本期主编 小智

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@geekbang.com

商务合作 hezuo@geekbang.org

内容合作 editors@geekbang.com

HUAWEI | Geekbang. 极客邦科技 InfoQ

业务创新，生态聚合

迎接API经济你真的准备好了吗？

2020年API现状调研 邀你参与

《《《 扫码参与，赢取大奖

卷首语

Kubernetes是怎样席卷世界的？

作者 小智

CNCF创始企业之一的IBM前两天在其网站上发布了一个招聘JD，岗位是Cloud Native Infrastructure Engineer / Architect，翻译成中文是云原生基础设施工程师/架构师。

这份JD与常见的招聘广告并无二致，只是在岗位能力一栏上要求应聘者具备“至少12 年的Kubernetes 操作和管理经验。”

而Kubernetes从2014年诞生至今，也不过才6年时间。

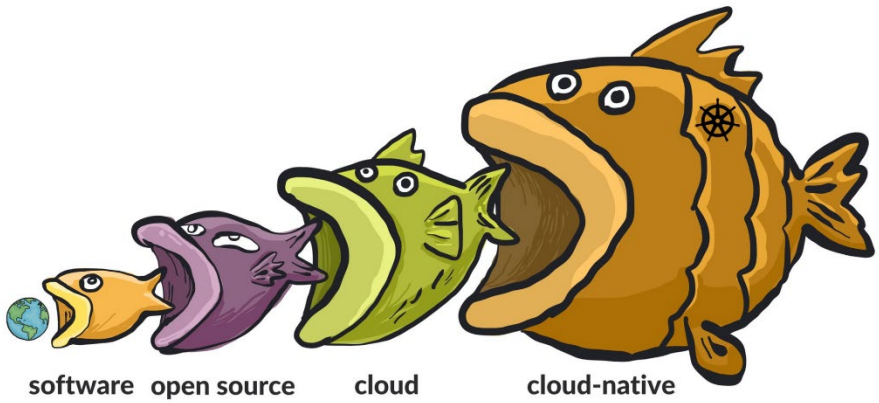
这则广告仍在IBM的招聘网站上挂着，没有人知道IBM对该岗位的能力要求是认真的还是在恶搞，就像没有人在一开始就能知道Kubernetes后来会席卷全球。

十年时间能给世界带来多大的变化？

2011年，Marc Andreessen说出了那句著名的论断——软件正在吞噬世界。

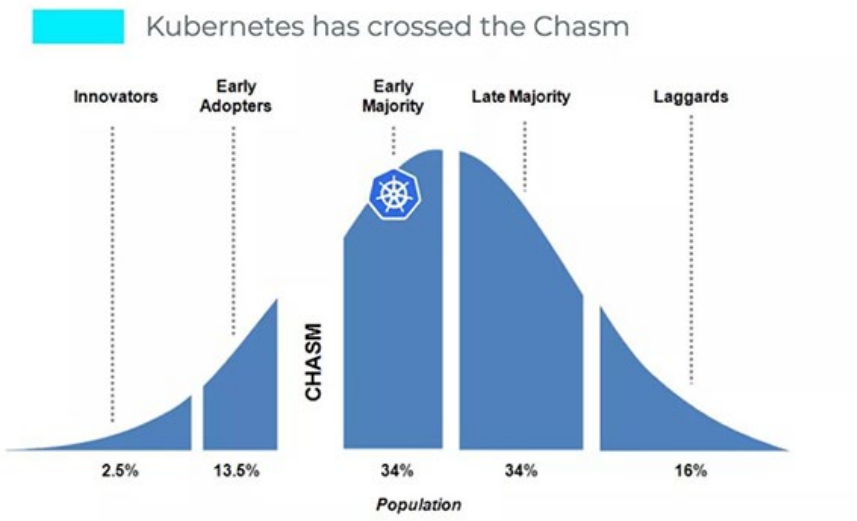
这在当年看起来耸人听闻的标题，在今天早已为人习以为常。十年互联网浪潮下，熠熠生辉的是IT技术的飞速发展，软件、开源、云计算、云原生，一环扣一环，成为了虚拟世界的真实规律。

今天吞噬世界的，是以Kubernetes为代表的“云原生”技术。

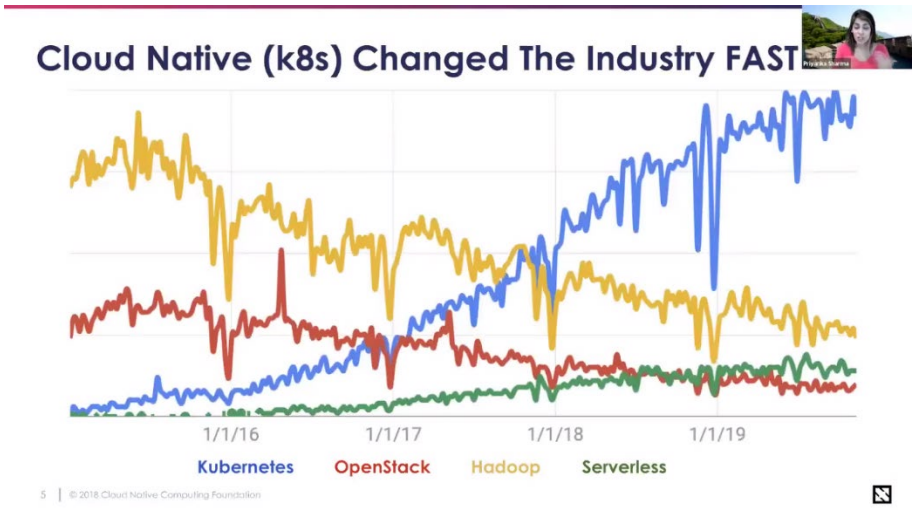


VMware 负责人 Pat Gelsinger 说：“从 Java 和虚拟机面世以来，这二十年里还未曾出现过像 Kubernetes 这样受业界欢迎、这么有活力的技术。我们从现在开始就应该押注 Kubernetes，它代表了下一个潮流。”

Kubernetes已经跨越了技术发展的那条鸿沟，进入早期采用者阶段。



与Hadoop、OpenStack的热度持续下降以及Serverless热度的缓慢提升不同，Kubernetes的上升态势长期保持着陡峭的曲线。以K8s技术为代表的云原生技术正在快速改变整个行业。



2019年，Datadog的一份样本超过数千家企业、容器超过15亿个的调研报告显示，运行容器的企业中，大约有 45%的用户在使用 Kubernetes，仅5年时间就占据了接近一半的市场份额。除了企业自用 Kubernetes，公有云上的 Kubernetes 同样成为了云厂商们的重要利润点，不管是国外的AWS、Azure、GCP，还是国内的阿里云、腾讯云、华为

云、京东云等厂商，都将云化 Kubernetes 当做自己的招牌产品对外输出。

背靠谷歌，Kubernetes发展速度十分迅猛，更新发版频率极高。而一个厂商中立的CNCF基金会，又给行业和社区带来了更具活力的生长土壤，Kubernetes 在这样的环境下野蛮生长，席卷世界。

Linus Torvalds：我们都老了，但 Linux 维护后继无人

作者 核子可乐，Tina

> Linux 之父非常担忧没人继续维护内核：“真的很难找到维护者！”

在本周召开的线上开源峰会与嵌入式 Linux 大会上，Linux 缔造者 Linus Torvalds 谈到了为开源操作系统寻找未来维护者时的种种挑战。Torvalds 近年来已经不再发表主题演讲，但这一次他与 VMware 公司首席开源官 Dirk Hohndel 展开了远程对话。

这次讨论很快就涉及到了一个令人不安的问题：在目前这一代维护者逐渐老去之后，Linux 项目将走向何处？面对 Torvalds 等这批五零后、六零后项目管理者，Hohndel 提到：“我们这个社区终归要考虑代际变更的问题。到那时，我们该怎么办？”

VMware 公司的 Dirk Hohndel（图左）在本届线上开源峰会中与 Linus Torvalds 进行了对话。

Torvalds 的回应是，Linux 内核社区的参与者们年纪不算太大。他表示，“很多新人都在 50 岁以下，他们才是目前开发工作的主力。当然，跟那些 30 岁上下的人们相比，我们确实是越来越老了。好在我们这些长期参与项目的早期成员还能做做维护与管理工作。”

维护者在社区内建立信任需要花费不少时间，Torvalds 指出，“这种信任不仅来自其他维护者，同时也来自所有代码贡献者……这肯定需要时间。”Torvalds 强调，“事实证明，维护者真的不好找。只要开始接管内核维护工作，就得一直坚持下去。每天都不能放松。我们得阅读电子邮件、做出回复，总之得一直待在那儿。而且维护工作属于那种要求不低但却需求面不大的小众岗位。”

“我们的维护者确实不够。能编写代码的人很多，能处理一部分维护工作的也不少，但很难找到那种可以吸纳他人代码贡献并立足上游将一切整合起来的人才。这也是我们目前面临的一大主要问题。”

另外，Linux 内核大部分是由 C 语言编写的。为此，Hohndel 问道：“C 语言是否会被 GO 及 Rust 语言取代，**我们这些用 C 的人有没有可能在二十年后变得像现在的 COBOL 程序员一样？**”Torvalds 的回应是，“C 语言目前仍是全球十大人气语言之一。但对语言的具体选择并不会对内核造成太大的影响。与驱动程序因此，内核团队正在研究多种语言接口，相信不久之后就能实现。总而言之，我们必将使用不同的模型编写 Linux 代码，而 C 绝对不会是其中唯一的模型选项。”

顶尖 Linux 开发者们已至暮年

上一代顶级程序员们确实在逐渐老去，Linus Torvalds 本人今年也超过了 50 岁。

Linux 社区需要新鲜血液，这也是事实。根据 Linux 基金会营销与开发人员计划副总裁 Amanda McPherson 所言，“目前 Linux 项目的参与者数量已经达到历史最高点。而且自 2005 年以来，已经有超过 8000 人为 Linux 内核做出贡献。”但从参与者数字来看，老一辈 Linux 程序员仍然是项目的主力。

软件开发分析公司 Bitergia 创始者之一 Jesús M González-Barahona 就发现，在以“参与项目的时间”作为“年龄”指标对 Linux 内核开发者进行统计时，可以看到新生

代程序员的占比一直在逐年下降。目前占比最高的参与者们，一般是十多年前就加入了 Linux 社区，之后几代的比例则呈现出下降趋势。

Linux 社区当然早就意识到了这个问题。

2010 年，资深 Linux 开发者兼 Linux Driver 项目负责人 Greg Kroah-Harman 就在 Linux 基金会协作峰会的内核小组讨论上指出，“项目高层的更迭一直没能成功完成。”

Parallels 公司服务器虚拟化 CTO James Bottomley 也表示，**“老一辈贡献者仍是项目主力。Linux 内核开发工作一直无法接棒，几年之后不知道还能剩下多少早期成员。”**

谷歌软件工程师兼高级 Linux 内核开发者 Andrew Morton 则总结道，“没错，**我们正在变老，精力也越来越差**。从现在来看，年轻一代也不像当初的贡献者们那样对内核开发充满热情。”

从多年前开始，Linux 基金会就一直试图解决问题。作为思路之一，Linux 基金会正努力吸引更多业余程序员加入进来。McPherson 补充道，“虽然 Linux 项目的参与者数量创下历史新高，但我们一直在努力吸引更多新的人才。而且大家基本达成了共识，**人才匮乏已经成为 Linux 实现进一步增长的最大障碍**。我们希望通过 LinuxCon 在新生代程序员中建立影响力，但目前看来这张网撒得太大，导致很多人搞不清 LinuxCon 到底是以开发者为中心、还是以系统管理员 / 架构师为中心。”

“怼天怼地怼空气”的 Linus

尽管 Linus Torvalds 有着无可置疑的天赋，但他对待社区参与者的方式使他成为一个极具争议性的人物。

对他行为的相关抱怨可以追溯到多年前。 2013 年，Intel 公司的内核开发人员 Sarah Sharp 称 Torvalds 的行为是不专业的，称 Torvalds“主张进行人身恐吓和暴力行为。”Torvalds 随后指责 Sharp 把自己描述成受害者博取同情，不接受任何劝他应该改变的建议。

在 2015 年发表演讲说到英伟达时，Torvalds 还曾转向一台摄像机说“so Nvidia fuck you”并竖起了中指。

2015 年底，Sarah Sharp 宣布退出（Closing a door）内核社区。Sarah Sharp 当时说道，过去一年多时间她已经逐步终止了手中的各项社区工作，转交了 USB 3.0 主控制器驱动的维护工作，不再担任开源会议的内核协调员。她不再递交任何补丁和 bug 报告，不再向内核邮件列表写任何的建议。她声称，Linux 内核社区的互动是一种“潜在有毒的背景辐射”，充满了性别歧视、语言暴力和不尊重人。

Torvalds 最终也意识到他的言行会伤害到社区发展。2018 年，他决定休假并反思自己的行为，在 4.19-rc4 版本发布公告中他写道：“我将抽出时间休息并寻求一些帮助，了解如何理解他人的情绪并做出适当的反应…我不是一个能对他人的感受感同身受的人，很多人对此也并不惊讶。 多年来，我误解了很多人，而我自己并没有意识到我对某些情况的判断有多么糟糕，这样造成了一种不专业的环境，这样不太好。”

写在最后

去年 8 月 7 日，首个专注于报道 Linux 内核及其发行版的杂志 Linux Journal 宣布停刊。杂志主编在官网公告上表示，因资金断裂，永久关停，并解雇了所有员工。很多人选择 Linux，是因为 Linux 开源免费。免费使用，但是又不愿意花钱或参与贡献。当时有人评论说：“今天死的是一个 Linux 杂志，明天就可能就是 Linux Mint 或者 LibreOffice。”

如今，Linux 作为最流行的操作系统，在超过 20 亿的设备上运行，已经成为人类技术发展中的不可或缺的一部分。我们难以想象无人维护的后果会是什么样。

参考链接：

https://www.zdnet.com/article/graying-linux-developers-look-for-new-blood/

https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/

高效软件开发团队的 4 个习惯

作者 Denise Yu 译者 平川

我经常需要费力地跟人解释，作为高效软件团队的一员到底意味着什么。当然，关于这一点已经有大量的资料，比如 LinkedIn 就有整套思想领导力理论介绍了各种帮助团队有效运作的指导方针和启发性思考，但根据我的经验，如果你从来都不知道什么样才算好，就很难内化这些想法并遵循别人的模式。

我非常幸运，到目前为止，我在职业生涯中已经直接与几十个（甚至是几百个）开发人员合作过。我曾在一些不健康的团队工作过：在那些团队里，人们会感到害怕，出于对职位的担忧，他们会把自己的底牌捂得很紧，不让其他人知道自己的计划或意图；我也在功能失调的团队工作过，那些团队由于工作优先级不明确而摇摆不定，或者协调成本太高而没有人愿意干活，许多天甚至数周的开发时间被浪费掉，团队成了一个个体的集合，而不再是一个工作单元。但幸运的是，我也曾在一些非常优秀的团队工作过。当我身处这些优秀团队的时候，我每天去上班时都很兴奋，对于那些比我年长的人，我也不怕公开提出反对意见，因为我认为自己的声音和工作很有影响力。

在这篇文章中，我将试着记录下，我所共事过的表现最好的团队所具有的特质和习惯。

本文最初发布于 Denise Yu 的个人博客，经原作者授权由 InfoQ 中文站翻译并分享。

高度的心理安全感

心理安全这个概念被提出来已经有一段时间了，所以我不打算花太多时间来解释它。如果你以前没有看过这个概念，请先阅读这篇文章。

软件团队由真实的人组成，他们在无形的社会和政治结构中生活和工作，他们从出生起就被社会化，变得更加自信，更加谦恭，更加直言不讳，更加礼貌，更加好争论，更善于安抚，等等。显然，这都是陈词滥调了，我说这些是为了证明，心理安全不仅指的是招聘一些顾问来开展员工培训、告诉大家这个概念意味着什么：建立真正的心理安全感需要领导者和管理者评估人与人之间交往的所有无形的社会规则，并理解这些规则如何影响一个人参与团队讨论、贡献团队动力的能力。简而言之：社交特权很重要。否则，当一些微不足道的小事情开始侵蚀团队凝聚力的时候，不要感到惊讶：攻击性的言语或小动作、刻板印象威胁、将幸存者偏差变成成就高效团队成员的信条。

根据我的观察，具有高度心理安全感的软件团队会有以下某些行为：

- 定期回顾，在“什么进展不顺利？”这一栏里列出适当数量的事项。那里不应该总是阳光和彩虹。那会让我怀疑回顾中是不是没有提出并讨论什么难题。健康的团队应该能够公开地反省并自我批评，因为每个人都明白，建设性的反馈是为了不断改进。
- 个人不会在一个问题上花很多时间。他们会有一个或明或暗的“奋斗时间窗”，在这个时间之后，他们会向同伴寻求帮助，他们知道，自己不会因此而得到负面评价。
- 个人从看得见的输出中分离出他们对团队的价值。我见过这样的情况，人们对自己的代码后来被删除或重构感到沮丧。但在健康的团队中，大家接受这样一个事实：改进是增量的、渐进的，贡献是对团队整体结果的贡献，而不是对特定输出（如代码行数）的贡献，表现出来就是“这是我们交付的”，而不是“这是我交付的”。此外，具有高度心理安全感的团队可以讨论价值的含义，以及价值为什么不仅仅是几行代码。
- 带薪休假时间和病假时间往往会更长，这很有趣。我认为，这是他们的信念的一种体现，即团队的其他成员可以在他们不在的情况下继续做出正确的决策，

> 因为他们之间已经进行了足够多的对话，这使得整个团队都认为，他们在产品和技术决策上非常一致。但我不太确定这其中的因果关系。因为人们如果发高烧，也会申请更多的带薪休假。

当一个团队有高度的心理安全感时，你可以尝试一些很酷的试验。我相信，这些试验会产生一种自我强化的积极反馈循环，创造出更高层次的信任和安全感。在我的第一个高绩效团队里，在一次回顾中，每个人都觉得一年两次的绩效评估周期不够频繁，也不够细致，无法促进职业发展，尤其是在我们团队的优先事项发展得如此之快的情况下。所以，我提出了一项试验：反馈周。

反馈周

这是一个为期一周的过程，每个人（包括团队负责人和项目经理）都被随机分配去收集对另一个人的反馈。这个过程进行得如此顺利，以至于在我的队友加入新的团队后，也带去了这个试验。最终，办公室里的其他团队开始模仿我们！我在这篇博文中更详细地介绍了这个试验。我还在 2019 年的多伦多 DevOpsDays 大会上就此做了一场演讲。

能够开展类似这样的反馈周，就说明你的团队处于高度的心理安全状态。如果你足够幸运参与其中，我的建议是：不要只站着不动。这是一个大胆试验你的过程和实践的机会，尝试像反馈周这样的事情，可以帮助你挖掘隐藏的积极反馈循环。如果你确实想出了一些很酷的东西，请告诉我！

良好的开发规范

> *随着系统复杂性的增加，系统中任何单个行为主体的自有模型的准确性都会迅速降低。*
>
> — Woods Theorem

我就直说了吧，我永远不会有一个完整的 GitHub 心智模型。它太庞大了，有太多的逻辑路径，坦率地说，花费过多的时间学习代码的所有部分，并不能使我的工作做得更好。而且，它可能明天就又变了。

因此，当我必须收集足够的上下文信息以实现下一个特性或 Bug 修复时，我会依

赖于代码中已有构件的准确性，这些构件是在我之前从事这些工作的人留下的。

我花了很多时间来研究代码：运行`git blame`，查看过去的提交、 有关问题，以及任何有助于我理解为什么某行代码这样写的信息。如果我看到一个难以理解的改动，提交信息是“WIP”，这就会变成效率杀手。

良好的软件开发规范意味着需要 **额外花一些时间来记录当前的上下文信息**，这可能表现在：

- 描述性的提交信息
  - 至少，做到每个提交信息都包含一个动词。有些团队甚至更进一步，要求每次提交都可以跟踪到问题编号。务必选择适合你的团队认知投资水平的方法；
- 遵循语言和框架约定、可以表明意图的类名和方法名；
- 单元测试带有有用的描述信息，使用符合实际的变量名和数据，而不是“foo”和“bar”这样的变量；
- 在问题跟踪系统中就相关特性反复沟通，而不是在 Slack DM 和其他地方。今后，入职不满 6 个月的团队新成员将无法访问这些地方。

最后，良好的开发规范事关同理心。工件越整洁，团队成员就可以越快地了解上下文，花在上下文切换和探查上的认知精力也就越少。此外，这其实对未来的你自己也是有好处的。道德哲学的一个分支认为，未来的你是一个有道德权利主张的不同实体，我想说：推广这些开发规范后续一定会得到回报，特别是在凌晨 3 点有人因为你写的一行代码给你打电话时！

主动重新分配“经验点”

我喜欢角色扮演游戏，尤其是《火焰徽章》和《口袋妖怪》系列，我最近还慢慢地喜欢上了《最终幻想》。

提这个是因为我认为，在《火焰徽章》中组建军队的方式和组建均衡的软件开发团队之间有很多相似之处。在 RPG 游戏中，我拥有自己的核心团队，我非常喜欢将所有

角色都均衡升级。如果我获得了一个等级较低的新角色，但他有一套技能或亲和力可以给队伍做补充，我就会对他进行投资，给他升一点级，这样他就可以在地图上到处移动而不用担心敌人的攻击。如果我的角色一开始就有一个很高的等级，我就会避免让他们与较弱的敌人战斗，因为这只会占用经验点，而这些经验点会让我的低等级角色受益更多。

我倾向于认为，软件团队中也存在类似的原则，但这些经验点不是为了增加力量、防御、魔法和抗性，每一项新工作都是一个“敌人”，一旦交付，就会扩大团队的领域上下文和信心。通常，团队中是没有核心“谋士”这样一个角色的，至此，这个类比就开始变得不恰当了（主管和项目经理不算，他们通常没有足够的视野或最新的上下文信息，无论如何，把如此复杂而又动态变化的事情都集中在一个人那里是个坏主意）。如果你的团队里已经有很多优秀的骑士和圣骑士——呃，我的意思是，高级开发人员——那么作为一个团队，你应该注意，不要总是只安排他们去处理困难的工作。在健康的团队中，上下文再分配也是他们工作的一部分，这样一来，一个缺乏经验的战士——我的意思是，工程师——也可以获得一些有价值的经验点。如果每个人都觉得自己至少在某种程度上具备了应对任何挑战的能力，那么这将提高整个团队的生产力和士气。如果没有，他们知道自己可以增加一个猎鹰骑士作为副官——换句话说，向更有经验的人寻求帮助。

慷慨大方地交流

关于最后一点，我想了很多。在对我的想法的所有描述中，我认为最好的是 Nat Friedman 在最近一次全体会议中所做的介绍。他说过类似这样的话：“我们彼此之间的交流应该遵循稳健性原则：发送时要保守，接收时要开放。”他还鼓励我们坚持宽容性原则，尤其是在非常困难的情况下与对方沟通的时候。但是我认为，高绩效团队会百分之百遵循这个原则。

我非常喜欢这种框架化，因为它让我想起了多年前我在codebar做志愿者时所经历的导师培训。“假设你接触到的每个学生都知识有限，但智慧无限。”这样一来，指导者就有责任确保他们的解释容易理解——考虑到老师和学生之间固有的权力不平衡，这是传达附加的情绪劳务的一种很好的方式。

同事之间慷慨大方地交流意味着，我们假设任何时候任何人问问题时：

- 已经做了基本的研究，如已经用谷歌进行了搜索；
- 他们是因为在任何地方都无法找到答案才找人问。因为这个地方很难找，或者根本不存在。

换句话说：假设你的同伴是一个有能力、聪明、通情达理的人，他们问问题是因为他们不了解上下文，虽然他们已经设法了解过。

当你开始寻求帮助时，你的上下文信息和工作经验被不断地“四舍五入”，我都不知道该怎么表达这是多么令人沮丧。是的，这里面有性别因素，但这超出了我们现在的讨论范围。以前，我曾发过多条推特，表达了当别人认为你实际上远没有那么经验丰富和知识渊博时的沮丧。当然，别人是不可能真正知道的，无所不知是不可能的！但是，这里有一个折中方案，也是一个合理的要求：慷慨。

像下面这样就**不够慷慨**：

> *我：嗨，这里为什么有一个负载均衡器？*
>
> *X：负载均衡器用于将请求分发到多个服务上，这样我们就不会遭受 DDoS 攻击了！这里有一些文章介绍负载均衡器的基础知识，以及从理论上讲我们为什么要使用它们！*
>
> *我：好的，但我问的不是这个。我知道负载均衡器是什么。我只是想了解下，在架构决策时，为什么要把 **HAproxy 放在这个特定的服务前面。*
>
> *X：奥！好吧，你应该直接这样问！*

相反，下面这样就是**慷慨**的：

> *我：嗨，这里为什么有一个负载均衡器？*
>
> *X：我猜你是在问我们为什么选择 HAproxy，以及为什么选择这些服务。如果不是*

> *的话，现在就告诉我。*
>
> *我：对，就是那样！谢谢你先确认我的问题。*
>
> *X：不用客气。是这样的，18 个月前，当我们构建这个系统时……*

上面两种互动方式的关键区别在于，在慷慨的互动中，在给出答案之前，回答者会确认他们对问题的假设，进而核实提问者的上下文层级和意图。采用慷慨的交流方式有非常积极的影响：首先，注意到交流时所使用的词汇减少了吗？他们实际上可以更快地得到答案。其次，没有产生不必要的摩擦。两个人团结一致消除了不确定性，而不是纠结于每个人知道多少，如果是后者，也许最终他们也可以得出答案，但同时也失去了一些信任和善意。

如果你有在高效软件开发团队工作的经验，或者花时间做过这方面的研究总结，请留言告诉我们你的感受和想法！

原文链接：Habits of High-Functioning Teams

从 0 到 1 搭建技术中台之组织架构篇

作者 伴鱼技术团队

> 自去年开始，中台话题的热度不减，很多公司都投入到中台的建设中，从战略制定、组织架构调整、协作方式变动到技术落地实践，每个环节都可能出现各种各样的问题。技术中台最坏的状况是技术能力太差，不能支撑业务的发展，其次是技术脱离业务，不能服务业务的发展。前者是能力问题，后者是意识问题。在本专题中，伴鱼技术团队分享了从 0 到 1 搭建技术中台的过程及心得。

引言

组织架构是围绕提高效率而设计的管理形式，任何新出现的组织架构一定是比之前的组织架构效率更高才有意义。中台是近来大家广泛讨论的一种组织架构，但是因为没有明确的定义，所以每一家公司对中台的理解可能会不尽相同。

但是不论什么样的组织架构，落到根本上都是服务好人才、做好事情，所以伴鱼在中台化落地的过程中，总是从这两个角度来评估当前组织架构对于效率的影响，以及应该怎么来优化当前的组织架构。本文回到组织架构的本质——效率的角度，分别从事情和人的角度讨论中台对组织架构的影响，以及伴鱼技术中台的建设与演进过程。

从事情的角度，中台组织架构应该关注提高复用率

每件事情只由一个团队负责

传统互联网公司的组织架构，一般是基于业务发展来设计的，开展新的业务线，组织架构是整体再建设一遍，这样会产生重复建设的问题。如果一个公司有多条业务线的时候，就会出现多个 DBA 团队，多个运维团队和多个基础架构团队等，这样一件事情

在公司内部有多个团队负责，每个团队都需要独立做人才管理和技术积累，从事情的角度来看，复用率是非常低的。

一件事情只由一个团队负责，很多重复的事情只需要做一遍，减少了很多不必要的工作量，并且集中了全公司在这一个方面的人才和资源，这样才能很专业、深入和系统性地把这件事情做好。

对于中台化的组织架构来说，每件事情只由一个团队负责是最基本的要求。这个在伴鱼的组织架构中是严格执行的，比如运维、DB 管理等，全公司只有技术中台的运维研发团队和 DBA 团队。

每件事情都要做成企业级服务

对于中台化的组织架构要求每件事情只由一个团队负责，其实这也对中台团队提出了一个要求，必须把自己负责的事情做成企业级的服务，因为整个公司只有你们负责这一件事情。这是权利与义务的关系，拥有了事情的独享权，那么就必须承担起服务全公司的责任。

目前伴鱼技术中台承担了所有的基础方面的事情，每一件事情都是企业级的服务，比如发布系统就承担了全公司所有业务线的所有端的发布服务，不论是伴鱼绘本还是伴鱼英语等不同业务的发布，还是服务器、安卓、ios 和 web 等不同端的发布；报警系统也一样，它被设计为一个信息的分发平台，前端、后端、运维和数据库等所有需要分发报警信息都先发送给它，由它完成报警信息到 Owner 的分发。

从人的角度，中台组织架构应该关注提高沟通效率

> Conway's law:
>
> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. - Melvin Conway(1967)

康威定律阐述了组织的沟通结构对系统设计的影响，沟通效率越高的组织，其系统设计也会更加合理。中台其实本质上也是通过从优化组织的沟通效率来影响公司的系统设计。比如帐号平台在中台化之前，提高一系列帐号相关服务，每个帐号相关的服务再

提供了一系列的 API ，这导致需要使用帐号能力的业务团队，都需要和多个帐号相关服务的 Owner 去对接和接入。这是一种很低效的沟通形式，其实接入帐号服务的流程是差不多的，通过将帐号相关服务成立一个团队，并且将之前多服务多 API 形式的接入方式抽象为提供帐号能力或者服务的形式，这其实就是中台化的核心目标。

所有的哲学是相通的，软件设计的哲学高内聚，低耦合，也是中台的哲学：打造高内聚，低耦合的沟通高效的组织架构，将一些频繁、重复和低效的组织之间的沟通变成高效简洁的组织内的沟通，并且高效简洁的组织内沟通更可能进化出新的能力。伴鱼技术中台在早期阶段建设了各种各样的平台，现在开始将一些平台打通融合，将多个零散的平台能力抽象为更高级的产品能力提供出来。

伴鱼技术中台之前提供了 APP 推送平台、短信推送平台和微信推送平台，现在开始往触达平台的方向演进，对不同的业务场景和不同的用户情况可以自动采用不同的触达方式，并且能通过触达反馈的数据来智能推荐不同的触达方式和提升触达能力，这就是从平台化到中台化的进化之路——从提供基础能力到提供强大的产品或服务能力。

伴鱼为什么需要进行中台化的组织架构

伴鱼是一家快速发展的在线教育公司，教育这个领域天生就有教学科目和教学场景两个维度，这两个维度的笛卡尔积就是它可能的业务类型数量，当然有多个科目和多个场景适合由一个业务线来负责，但总体上由于科目和场景的业务差别会非常大，出现多个业务线是必然的趋势，所以将一些基础和业务能力中台化，提高企业级的复用能力是目前最适应公司业务特点的组织架构。

伴鱼技术中台组织架构的演进

上面阐述了伴鱼对中台组织架构的理解：中台化组织架构应该提高事情的复用率和组织的沟通效率，并在这一基础上提供强大的产品或服务能力。下面我们讲述在这一原则下伴鱼技术中台组织架构的演进过程。

第一个阶段是基础架构阶段，这是伴鱼技术中台的初始阶段，当时只是服务器基础架构组，负责基础服务、微服务框架和服务治理相关的事情，不过当时 DBA 是属于基础架构组的，这是因为我们认为 DB 是基础架构非常重要的一部分，DBA 和基础架构

的需要密切沟通和深度配合的，比如数据库路由可以理解为服务发现的一部分，熔断降级等服务治理的方法其实对数据库治理同样非常关键，这是中台化思想在组织架构上落地最初形式。

第二个阶段是平台化阶段，这个时候已经正式成立了伴鱼技术中台部门，前面的服务器基础架构团队和 DBA 团队加上运维研发团队、WEB、安卓和 IOS 基础架构团队合并为伴鱼技术中台，并推行 SRE 文化，各团队都开始将一些需要人工管理的事情平台化，由于技术中台部门包含前端和后端的基础架构团队，所以沟通和配合效率非常高，这一段时间推出了各种各样的平台，并且所有的平台都提供企业级服务能力，比如发布系统、灰度系统、数据库平台、报警平台、APM 平台、账号平台、存储中心、安全中心等等

第三个阶段是中台化阶段，这个也是伴鱼技术中台目前所处的阶段，现在开始，对内我们挖掘平台之间的价值，对外关注业务使用上的研发体验。挖掘平台之间的价值，其实就是做平台之上的平台，打通多个相关平台的数据和接口，提供产品和服务能力更好的平台；另外也从研发体验的角度出发，从使用角度将一些场景相近和关联的平台相互融合，提供统一、简单的使用方式。并且，在平台的融合与调整中，为了获得最好的沟通效率，内部组织架构也会做响应的调整。现在正在进行的 A/B Testing 平台（从数据上报、指标分析和指标信度评估等全功能统一的 A/B Testing 平台）、触达平台（打造统一的触达用户的智能平台）和统一前后端的性能分析平台（全业务和全场景的性能分析平台）等都是在这个方向上的努力工作。

总结

在伴鱼技术中台的发展过程中，组织架构是不断演进的，其实不论有没有中台这一概念，只要不断优化组织的效率，最终都会呈现中台化的组织形态，好比不论使用什么样的软件架构理论，最终目标一定是一个高内聚，低耦合的系统。

只是，恰好，有了中台这一个概念，让伴鱼技术中台不需要再取一个新的名字了，这是非常幸运的事情，一直觉得写代码最难的就是命名。

甲骨文：有史以来最伟大的25个Java应用程序

作者 Alexa Morales 译者 刘雅

摘要：从太空探索到基因组学，从反向编译器到机器人控制器，Java 都是当今世界的核心。下面将介绍几个从众多 Java 应用程序中脱颖而出的优秀应用。

Java 的故事始于 1991 年，当时 Sun Microsystems 试图将其在计算机工作站市场的领先地位扩展到新兴且发展迅速的个人电子产品市场。几乎没有人预料到 Sun 即将创建的编程语言会使计算大众化，激发了一个全球范围的社区，并成为了一个由语言、运行时平台、SDK、开源项目以及许多工具组成的持久软件开发生态系统的平台。经过James Gosling领导的数年秘密开发之后，Sun 于 1995 年发布了具有里程碑意义的“一次编写，随处运行” 的 Java 平台，并将重点从最初的交互式电视系统设计转到了新兴的万维网应用程序上。在本世纪初，Java 就已经开始为从智能卡到太空飞行器的一切制作动画了。

如今，数以百万计的开发人员在使用 Java 编程，Java 仍然在以越来越快的步伐向

前发展。在Java 诞生25 周年之际，Java Magazine（Oracle 的双月刊）联合Oracle Java 开发团队，共同撰文回顾Java 是如何塑造我们这个星球的。

以下是迄今为止，最具创意和影响力的25 个Java 应用程序， 包含了从Wikipedia Search 到美国国家安全局的Ghidra 等。这些应用包罗万象，覆盖了包括：太空探索、视频游戏、机器学习、基因组学、汽车、网络安全等不同领域。

这份清单没有特定的顺序，也还不够详尽，可能会有遗漏，如果你认为文章中遗漏了哪个重要的Java 应用，可以在文末给我们留言！

最后的边界

1、Maestro 火星探测器控制器。 2004 年，Java 成为首个扩展人类星球影响力的编程语言。那年在三个月的时间里，美国国家航空航天局（NASA）的科学家在位于加利福尼亚州帕萨迪纳的喷气推进实验室（JPL）里，使用了由 JPL 机器人接口实验室建造的基于 Java 的Maestro Science Activity Planner来控制“勇气号”火星探测器（Spirit Mars Exploration Rover ）。在 JPL 里，Java 试验早在许多年前就开始，当时是为 1995 年的“逗留者”火星车（Mars Sojourner）创建了一个命令和控制系统。 Java 创始人 James Gosling 为 JPL 工作了很长时间，因此他成为了顾问委员会的一名成员。

2、JavaFX 深空轨迹探测器。计划进行一次太空飞行？您可能需要来自a.i.solutions 的工具，a.i.solutions 是一家美国航空承包商，其产品和工程服务已经被国防公司和民用航天机构使用了 20 多年。

该公司的JavaFX 深空轨道探测器（JavaFX Deep Space Trajectory Explorer）使轨迹设计人员可以计算深空三体系统的路径和轨道。该应用程序可以为任何行星 - 卫星系统或小行星生成多维视图和模型，并能在密集的视觉搜索中过滤数百万个点。

3、NASA WorldWind。NASA 发布了开源的WorldWind软件开发工具包，所有人都可

以免费使用火箭科学家的工作成果。WorldWind 是一个虚拟地球 SDK，允许程序员将美国航天局的地理渲染引擎添加到自己的 Java、Web 或 Android 应用程序中。 World-Wind 的地理空间数据远远超过了谷歌地球（Google Earth），它是由 NASA 工程师通过高程模型和其他数据源可视化地形的方式生产的。其网站称：“世界各地的组织都使用 WorldWind 监测天气模式，可视化城市和地形，跟踪车辆移动，分析地理空间数据以及对人类进行地球知识的教育。”

4、JMARS 和 JMoon。Java 任务 - 遥感计划和分析（JMARS）是一个地理空间信息系统，由亚利桑那州立大学火星空间飞行设施人员编写，自 2003 年开始公开发行，至今仍为 NASA 科学家所使用。月球JMARS（月球科学家称之JMoon）可以分析月球勘测轨道飞行器（Lunar Reconnaissance Orbiter，LOR）拍摄的广角图像，它是一种自动航天器，自2009 年发射以来，一直在50 至200 公里的月球轨道上运行，并能将观测结果发送给NASA 的行星数据系统（Planetary Data System）。

5、小体映射工具（ Small Body Mapping Tool ，SBMT）。SBMT在太空科学家中很受欢迎，是由约翰霍普金斯大学应用物理实验室开发的，它使用来自航天器的任务数据以 3D 的形式可视化小行星、彗星和小卫星等不规则天体。 SBMT 是用 Java 编写的，并使用了用于 Java 3D 图形处理的开源可视化工具包（VTK）。“黎明“号（Dawn）、“罗塞塔”号（Rosetta）、“奥西里斯 - 雷克斯”号（OSIRIS-REx）和“隼鸟二号”（Hayabusa2）飞行任务团队在探索彗星、小行星和矮行星时都使用了 SBMT。

数据的强度

6、Wikipedia Search。一部大众百科全书应该运行在开源软件上，并且具有一个由 Java 驱动的搜索引擎，这是再合适不过的了。Lucene 是由 Doug Cutting于 1999 年编写，并以他妻子的中间名命名的，它实际上是 Cutting 开发的第五个搜索引擎。他作为工程师先后为 Xerox PARC（施乐帕克研究中心）、Apple 和 Excite 创建了其他的引擎。

2014 年，Wikipedia 用Elasticsearch代替了 Lucene 引擎，Elasticsearch 是一个分布式的、支持 REST 的搜索引擎，也是用 Java 编写的。

7、Hadoop。Lucene 并不是唯一一个进入我们这个榜单的 Cutting 创建的作品。2003 年，Google 在一篇研究论文中描述了在大型商用计算机集群上处理数据的 MapReduce 算法，受该论文的启发，Cutting 用 Java 编写了一个 MapReduce 操作开源框架，并以他儿子的玩具大象命名，称为Hadoop。 Hadoop 1.0 于 2006 年发布，催生了大数据趋势，并激发了许多公司开始收集“数据湖”（data lakes），制定挖掘“数据排放”（data exhaust）的策略，并将数据描述为“新石油”（the new oil）。到 2008 年，Yahoo（当时的 Cutting 曾在该公司工作）宣称他们的 Search Webmap 运行在 10,000 个内核的 Linux 群集上，是现有的最大的产线 Hadoop 应用程序。到 2012 年，Facebook 声称在全球最大的 Hadoop 集群上拥有超过 100 PB 的数据。

8、并行图形分析（Parallel Graph AnalytiX，PGX）。图形分析是有关理解数据中的关系和连接的。根据基准测试，PGX是世界上速度最快的图形分析引擎之一。 PGX 是用 Java 编写的，由 Oracle Labs 研究员 Sungpack Hong 领导的团队于2014 年首次发布，PGX 允许用户加载图形数据并运行分析算法，比如，社区发现（Community Detection）、聚类、路径查找、页面排名、影响因素分析、异常检测、路径分析和模式匹配等算法。在健康、安全、零售和金融领域，它的用例比比皆是。

9、H2O.ai。机器学习（ML）的曲线非常陡峭，这可能会阻止领域专家实现伟大的 ML 想法。自动化 ML（AutoML）可以通过推断 ML 流程中的某些步骤（例如特征工程、模型训练和调整以及转译等）来提供一些帮助。 由 Java 冠军 Cliff Click 创建的基于 Java 的开源H2O.ai平台，旨在实现 AI 的大众化，并能为那些刚入门的人们充当虚拟数据科学家，同时能帮助 ML 专家提高效率。

有趣的世界

10、 Minecraft。该游戏的和平环境是由生物群落、人以及自己用积木搭建的住所组成的，它对世界各地的儿童和成人都有着持久的吸引力，这使得它成为历史上最受欢迎的视频游戏。 Minecraft及其 3D 宇宙是由 Markus “Notch” Persson 用 Java 开发的，并于 2009 年以 Alpha 版本发布，它是永无止境的创造力之源，因为没有两个衍生的世界是一样的。该视频游戏对Java的使用也可以让在家和学校的程序员创建自己的模块。

11、Jitter 机器人和 leJOS。在自动吸尘器 Roomba 出现之前，就已经有Jitter。Jitter 是一个用来吸取国际空间站（ISS）中漂浮颗粒的原型机器人，它能够在失重状态下导航，在墙壁上弹跳，并能使用回转仪进行自我定位。据报告称，俄罗斯宇航员发现该机器人的 x、y、z 旋转操作令人印象深刻，能让人联想到国际空间站自身是如何控制其方向的。Jitter 是leJOS最出类拔萃的原型，leJOS 是 Lego Mindstorms 的 Java 虚拟机，是 Lego 的硬件软件环境，可用于从积木玩具中开发可编程的机器人。玩具 OS 可以追溯到 1999 年由 José Solorzano 发起的 TinyVM 项目，该项目后来演变成 leJOS，由 Brian Bagnall、JürgenStuber 和 Paul Andrews 领导。这个功能齐全的环境具有许多特定于机器人编程的类，这些类使用 Java 的面向对象特性进行了简化，使得任何人都可以利用其高级控制器和行为算法。

12、Java 小程序。根据牛津英语词典，小程序（applet）一词最早出现在 1990 年的 PC Magazine（计算机杂志）上。但是直到 1995 年 Java 出现后，小程序才真正腾飞。Java 小程序可以在网页（Frame、新窗口、Sun 的AppletViewer或测试工具）中启动，并能运行在于浏览器相独立的 JVM 上。一些人将 Minecraft 的早期成功归功于这样一个事实：玩家可以通过 Java 小程序在 Web 浏览器中玩游戏，而不必下载并安装游戏。尽管 Java 小程序自 Java 9 以来就不被推荐，并且在 2018 年也被从 Java SE 11 中剔除掉了，但它们一度是最快的游戏。一个有趣的事实是：Java 小程序还可以访问 3D 硬件

加速，这使得它们在科学可视化方面很受欢迎。

荣誉代码

13、NetBeans 和 Eclipse IDE。最早进入 Java 集成开发环境世界的是NetBeans，NetBeans 于 1996 年在布拉格的查尔斯大学（以 Xelfi 的名义）创立，并于 1997 年由企业家 Roman Staněk 创立的同名公司进行商业化。 Sun 在 1999 年购买了支持所有 Java 应用程序类型的模块化 IDE，并于次年将其开源。 2016 年，Oracle 将整个 NetBeans 项目捐赠给了 Apache 软件基金会（Apache Software Foundation）。

另一个流行的基于 Java 的集成开发环境是开源的 Eclipse IDE，它不仅可以用于 Java 编码，还可用于从 Ada 到 Scala 的其他语言的编码。 Eclipse SDK 由 IBM 于 2001 年推出，是基于 IBM VisualAge 的，它是面向 Java 开发人员的，但是可以通过插件进行扩展。 Eclipse IDE 于 2004 年从 IBM 分离出来并加入 Eclipse 基金会，它目前仍然是可用的顶级 IDE 之一。

14、 IntelliJ IDEA。IDE 有很多，但 IntelliJ IDEA 在 2001 年推出后就成为了人们的最爱。如今，IntelliJ IDEA 已成为许多 IDE 的框架，这些 IDE 适用于 Python、Ruby 和 Go 等多种语言。IntelliJ IDEA 及其相关的JetBrains IDE 套件是使用Java 编写的，可以提高许多开发人员所依赖的生产力和导航功能。其中包括代码索引、重构、代码完成（这要早于智能手机上的文本自动完成）以及发现错误的动态分析（类似于拼写检查器）。“ IntelliJ IDEA 帮助克服了在某个框架下管理和调试基于Java 和JVM 的复杂应用程序的挑战，”驻英自由软件和数据工程师、Java 冠军Mani Sarkar 说。 “它们让开发人员在使用它们的工具时，感到高效、多产，最重要的是能感到快乐。”

15、Byte Buddy。开源 Java 库Byte Buddy的创建者，来自奥斯陆（挪威 Oslo）的软件工程师Rafael Winterhalter坦承，他的一生（有时令人发狂）专注于小众市场。尽管如此，他的贡献还是大受欢迎：Winterhalter 说，用于 Hibernate 和 Mockito 等 Java 工

具的 Byte Buddy 运行时代码生成和操作库每月下载量高达 2000 万次。

16、Jenkins。Jenkins由 Sun Microsystems 工程师 Kohsuke Kawaguchi 于 2004 年创建，是一个功能强大的开源的持续集成服务器。 Jenkins 用 Java 编写，可帮助我们快速自动地构建、测试和部署应用程序。它通常被认为是使“基础设施即代码”（Infrastructure As Code，Iac）成为可能的早期 DevOps 工具之一。 Jenkins 及其 1500 多个由社区贡献的插件可以处理各种各样的部署和测试任务，从与 GitHub 合作，到支持色盲开发人员，再到提供 MySQL Connector JAR 文件。

17、GraalVM。由Oracle Labs 的 Thomas Wuerthinger领导的一个苏黎世（Zurich）研究团队，花费了多年的时间来磨练三个想法：我们是否可以用 Java 编写编译器（原始 JVM 是用 C 编写的）呢？它是否可以运行以任何语言编写的程序呢？它是否够高效呢？在发表了 60 篇研究论文之后，GraalVM最终胜出，并成为了一个商业产品。 Twitter 是这项技术的狂热爱好者之一，它使用 GraalVM 来提高服务的速度和计算效率。

18、Micronaut。为云编写代码的开发人员需要仔细考虑其应用程序使用了多少内存，以及应用程序如何使用这些内存。Micronaut的创建者 Graeme Rocher 说： “您必须使应用程序对重启、故障自动切换、停机再恢复非常敏感，并在启动时间和内存消耗方面进行优化，” 。Micronau 是一个用于微服务的 Java 框架，该微服务需使用注解元数据，以便 JVM 能够有效地编译应用程序的字节码。

19、WebLogic Tengah。1997 年，WebLogic Tengah成为企业级 Java 服务器的首个实质性实现。 “它早于 Java 2 企业版，并成为 BEA 的主要产品，最终导致了 Oracle 收购 BEA Systems，”Java Magazine 和 Dr. Dobb's Journal 的前主编 Andrew Binstock 说。与此同时，IBM 在业务对象框架San Francisco Project上的成功，“使 Java 真正从酷孩子们正在玩的一个有趣的新事物中脱颖而出，并成为一种严肃的业务工具，”Binstock 说。如今，Oracle WebLogic Server 仍然是领先的 Java 应用程序服务器。然而，另一种选择仍在蓬勃发展：开源应用程序服务器GlassFish，它于 2005 年由 Sun 创建，于 2018 年捐赠给 Eclipse 基金会。

20、Eclipse Collections。工作在投资银行、证券交易所和其他金融服务公司的许多高薪开发人员都需要强大的 Java 技能，这是有原因的：Java 编程语言擅长处理并发，即管理高频交易和其他大规模金融事务中常见的多个执行线程。Eclipse Collections最初

称为 Goldman Sachs Collections，后来捐赠给了 Eclipse 基金会，它扩展了原生 Java 的性能密集型特性，“具有优化的数据结构和丰富的、功能强大的、流畅的 API”，Java 冠军 Mani Sarkar 说。 Sarkar 指出，Eclipse Collections 包含缓存、原语支持、并发库、通用注解、字符串处理、输入 / 输出等等。

21、NSA Ghidra。在旧金山举行的 2019 年 RSA 大会上，美国国家安全局（ U.S. National Security Agency）推出了一款基于 Java 的开源工具Ghidra，安全研究人员和从业者现在可以使用它来了解恶意软件的工作原理，并检查自己的代码是否存在漏洞。这个逆向工程平台可以将软件从机器语言反编译回源代码（例如 Java 语言）。该工具有一个故事，即使不是声名狼藉，也是传奇：2017 年 3 月，维基解密（WikiLeaks）将其存在公之于众。

绘制基因组图谱

22、集成基因组浏览器（ Integrated Genome Browser，IGB）。绘制人类基因组图谱的竞赛始于 1990 年，并在 13 年后结束。当时，医学研究人员成功地对生物技术专家 Craig Venter 的 30 亿个 DNA 碱基对进行了测序，这项工作历时 10 年，涉及 3,000 人，耗资 30 亿美元。测序完成后，科学家们很想深入研究我们这个物种的源代码，但是怎么做呢？进入基于 Java 的基因组浏览器，这是一个由包括生物信息学教授 Ann Loraine 在内的团队开发的可视化工具，可用于探索基础数据集和参考基因注解。开源的集成基因组浏览器（Integrated Genome Browser）允许研究人员放大、平移和绘制基因组数据，以便识别和注解遗传特征。为了配合这一全球努力，加州大学圣克鲁兹分校（University of California Santa Cruz）提供了一个类似的工具，即由Jim Kent 管理的基因组浏览器（Genome Browser）。

23、BioJava。BioJava于2000 年启动，至今仍很强大，它是一个用于处理生物数据的开源库，生物数据领域也被称为生物信息学。科学家使用该库可以处理蛋白质和核苷

酸序列，并可以研究有关基因到蛋白质翻译、基因组学、系统发育和大分子结构的数据。该项目得到了开放生物信息学基金会（Open Bioinformatics Foundation，OBF）的支持，其全球范围的贡献者得到了各种制药、医学和基因组学领域的资助。 “BioJava 是方法论和软件开发的一个热门选择，这要归功于Java 的可用工具及其跨平台的可移植性，”Aleix Lafita 及其同事在2019 年发表的一篇题为“ BioJava 5：社区驱动的开源生物信息库”的论文中写道。该论文进一步指出，自2009 年以来，BioJava 已经接受了65 个不同开发人员的贡献，并且在过去的一年中，它已在GitHub 上累积了224 个fork 和270 个star，并且下载次数超过了19,000 次。

最喜欢的“东西”

24、VisibleTesla。这款基于 Java 的应用程序是由特斯拉（Tesla）汽车爱好者 Joe Pasqua 于 2013 年创建的，它是一个免费程序，可用于监视和控制他的特斯拉 Model S。VisibleTesla的灵感来自于特斯拉汽车俱乐部社区，它提供与电动汽车制造商官方移动应用程序类似的功能。用户可以为诸如解锁的门或充电状态之类的东西设置地理围栏和通知，以及收集和处理行程数据。该项目的开源代码托管在 GitHub 上。

25、SmartThings 。该物联网（IoT）应用程序是由SmartThings开发的（SmartThings 是由 Alex Hawkinson 于 2012 年联合创立的，后来在 Kickstarter 上筹集了 120 万美元的资金），可以让我们通过智能手机或平板电脑控制和自动化所有的东西，从家用照明、锁、咖啡机、恒温器以及收音机到全部家用安全系统。该应用程序使用基于 Java 的 Micronaut 框架（请参阅#18），因此它基于云的服务可以以亚秒级的速度运行。该公司于 2014 年被三星电子（Samsung Electronics）以 2 亿美元的高价收购。

原文链接：

https://blogs.oracle.com/javamagazine/the-top-25-greatest-java-apps-ever-written

戳破微服务的七大谎言

作者 Scott Rogowski 译者 王强

本文最初发布于 scottrogowski.com 网站，经原作者 Scott Rogowski 授权由 InfoQ 中文站翻译并分享。

在现代技术公司（无论大小）的架构中，微服务已经无处不在。但是，它们真的比以前的开发模型更优秀吗？在这篇文章中，我将揭穿工程师们关于微服务所讲述的七大谎言，以及为什么它可能是一种反模式。

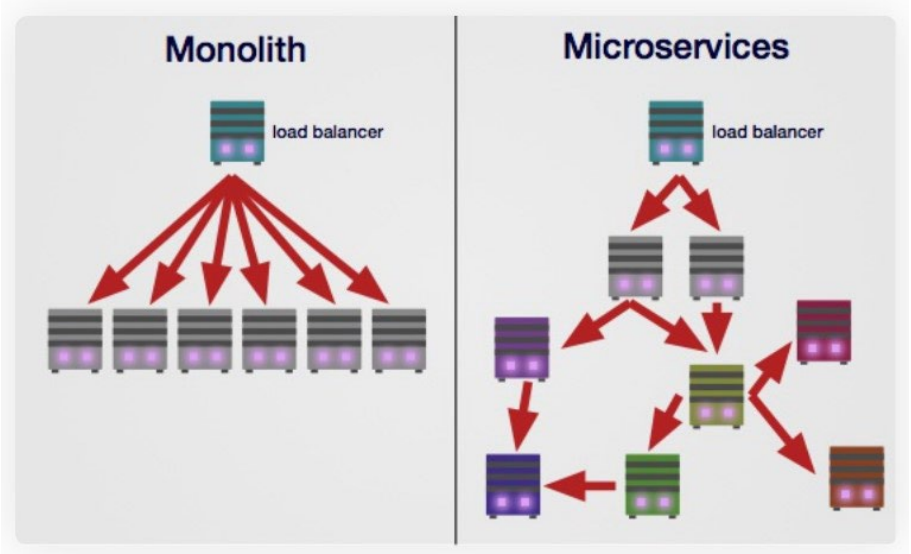
免责声明 1：我不是架构师，也没假装自己是架构师。本文内容只是我多年来作为软件开发人员 / 经理所做的观察总结。我曾见证两家公司在微服务架构的压力下陷入泥潭。由于很少有人深度质疑这种新生范式，因此我想表达自己的声音。不过，我经验有限，所以也欢迎反馈意见。

免责声明 2：互联网上也有其他标题相似的演讲 / 文章，这里就不纠结这种相似度了。

单体架构和微服务之间有何区别?

开始研究谎言前，我们先来定义一下术语。后端软件架构可以分为单体和微服务两种。单体架构指的是由一台或多台服务器运行单个应用程序，其通常从单个存储库中部署。使用多台服务器时，这些服务器将运行相同的代码。从 90 年代到 2000 年代，多数情况下这都是默认的架构。

随着互联网的发展，大型公司开始面临单体架构的局限。为解决这一问题，公司开始将其代码分割成在不同服务器上运行的多个组件。例如，一家公司可能会有运行日志记录的服务、调用外部 API 的服务以及管理数据库的服务。亚马逊 AWS 在这一风潮中扮演了重要角色，因为它让部署服务器和管理基础设施的工作变得非常容易。



随着时间流逝，中小型公司也开始接受这种新的发展范式。很快，一个围绕微服务的产业发展壮大起来，它逐渐成为寻求扩展的企业默认架构。不幸的是，现在有许多公司由于这种选择而掉进了各种之前想不到的坑里。

谎言 1：跨服务的关注点分离降低了复杂度

“[关注点分离]”指的是在不相关的代码间应存在隔离墙。当不相关的代码需要协同工作时，应该使用抽象良好的接口并尽量减少状态共享。很多入门编程课程都将其视为标准的软件开发公理。你的代码对其他代码的了解，越少越好。同样，一个函数执行的功能越多，你就越需要考虑运动部件之间的复杂关系（即复杂度）。而且，我们作为合格的工程师就应该努力降低复杂度。

我坚信这一公理。从逻辑上讲，分离关注点的最佳方法是否就是让你的无关代码运行在不同的服务（服务之间以 API 沟通）中呢？

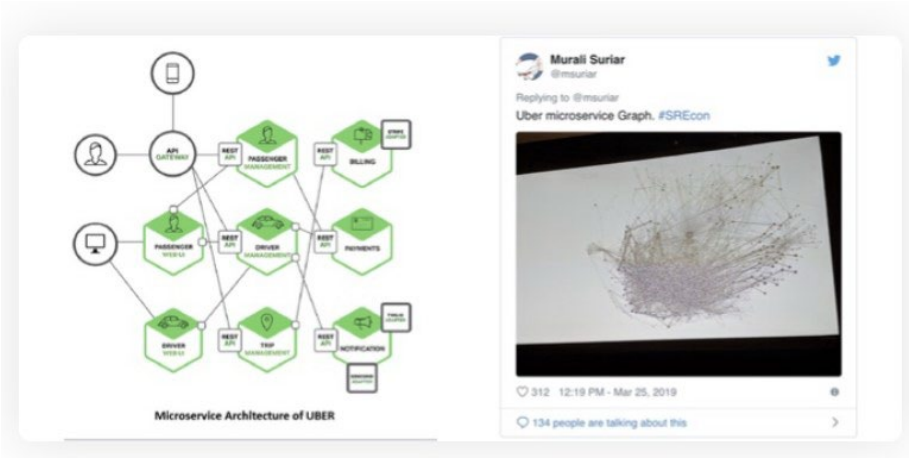
不，并非如此。

经典的单进程关注点分离之所以有效，是因为它可以最小化并简化不相关代码之间的接口。在设计良好的程序中，此接口可以只有带 return 语句的单个函数调用。不相关代码之间的边界本质上是复杂的，而简单的接口有助于管理这种复杂性。

相比之下，在微服务中，函数调用被替换为网络请求。这种新的服务间障碍严格来说更加复杂且更不可靠。首先，每个网络调用都需要一定数量的样板。其次，工程师现在需要默认任何服务随时会失效。相反，在单体中，当代码失败时整个服务都会失败。尽管这听起来很糟糕，但由于现在只有一种故障情况，因此它更易于管理。

在实践中更糟

左下方的图表是几年前 Uber 的微服务架构。它很简单，很容易理解。右边是 Uber 的实际服务地图。



我敢说 Uber 的任何人都不知道这个架构是如何工作的。曾在使用微服务架构的大型公司工作过的人都知道，Uber 的经验并不是特例。

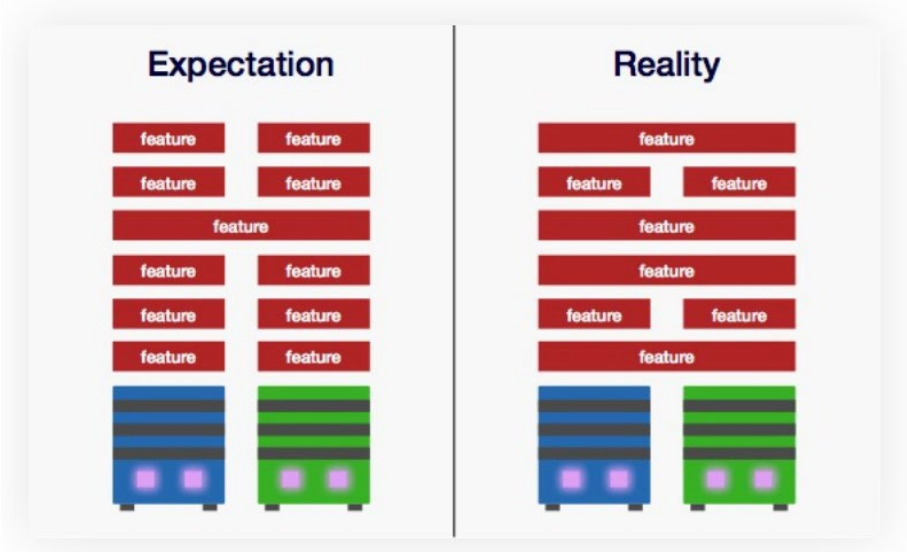
模型与现实之间存在这种差异的原因有两个。首先，这些图表通常过于简化。架构师设计这些图表是为了交流，而非完美反映现实。但因为它们隐藏了复杂性，也就容易误导决策。如果 Uber 的技术领导者知道自己的架构会变成什么样，他们还会走这条路吗？

其次，一旦你投身于微服务，随后的所有技术决策都将受其影响。因此，开发新功能时总要启动新的服务。架构图很快就会变得非常复杂，膨胀成上图那种密密麻麻的网状结构。

谎言 2：微服务提高开发速度

当你采用“关注点分离”公理并将其应用在开发人员头上时会发生什么？你会得到一些孤立的团队，他们之间各自独立。从表面上看这似乎是有益的。如果团队只需要操心自己的服务，那将减轻他们的认知负担，并提高他们的生产力。现在，工程师无需担心基础架构中其他部分的复杂性了。

问题在于，大多数新功能都需要一些跨多个服务的补丁。



许多功能需要在两个或多个服务上开发

开发多服务功能需要在具有不同优先级和能力的团队之间安排大量会议。考虑到他们从事的多个不同项目，这些团队可能需要异步协作。你现在还需要交付经理来分配工作和管理迭代。

从技术角度来看，实现多服务功能可能需要编辑多个存储库。至少，它需要一种方法来测试在多个服务上运行的代码。

对许多公司而言，这种多服务测试的需求是事后才会意识到的。架构师在设计技术栈时会假设大多数开发工作都将在单个服务上进行。多服务功能将很少见。你如何手动测试多服务功能？你需要在机器上启动多个容器，并仔细设置每个容器的状态。那单元测试呢？你将在哪里对多服务功能进行单元测试？是仓库 A 还是仓库 B？文档写好了吗？部署往往会破坏未调整好的服务。很容易想象，数据流中的一个小错误会破坏多个下游服务。我们应该期望工程师理解所有可能依赖其代码的下游服务吗？

如果你的组织没有投入大量的工程资源来构建多服务测试流程，那么除了最常见的功能之外，开发新功能的速度会像蜗牛般缓慢。如果没有质量测试框架，看似简单的任务（例如“添加分页”）也可能会变成历时数月、跨多个团队的工作。

谎言 3：部署众多小型服务比部署整个应用更安全

“回滚”是现代软件工程需要面对的现实。作为工程师，当你部署的代码会破坏某些功能时，必须回滚部署并还原提交。没有人想要回滚——尤其是部署代码的工程师。但是，好的公司知道错误的部署总有可能出现，因此必须对其进行管理。

微服务架构的一个观点是，部署多个独立服务比部署整个应用更安全。当一项服务中断时，其他服务还有回退可用。整个应用程序将继续运行，客户不会有什么感觉。

这种方法存在多个问题。

首先，这要假设你的服务可以容忍其他任何服务的随机消失。这是 Netflix 的“Chaos Monkey”方法。但是，将其构建到服务中并非易事，测试它需要资源，并且除非这是工程的最优先事项，否则实践中人们多大程度上会遵守这一要求就不一定了。

其次，部署多服务功能时，服务的上线时间会有所不同。在一段时间里，你的那些服务将有不同的版本。对此有多种处理方法。你是否在半夜部署？你是否并行维护不同的 API 版本？你是否使用托管流？所有解决方案都需要额外的工程资源。如果部署意外破坏了（甚至不是部署的一部分）服务中的状态，会发生什么情况？你是否有针对任何意外情况的预案？

虽然单体部署也会出错，但是有多种方法可以缓解这种情况（蓝色 / 绿色、金丝雀等等）。虽然这些方法也可用于微服务，但是设置和管理安全部署并非易事，应对一项服务总比应对多个服务要容易些。

谎言 4：分开扩展服务通常是有利的

在每个应用程序中，都有经常运行的部分和很少运行的部分。很少运行的部件比频繁运行的部件需要的资源要少一些。那么分开扩展这些部件是否有意义？

从根本上讲，扩展软件的原因是因为你的软件需要更多的核心资源。这些资源可能是 CPU 周期、内存、磁盘空间或网络。例如，当 CPU 以 100%运行时，可以启动另

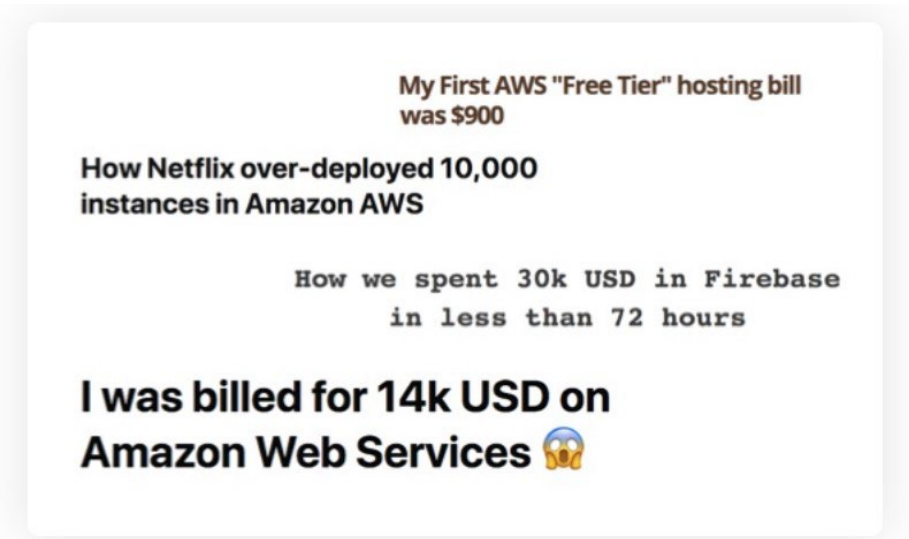
一个服务来减轻压力。

对于大多数应用，水平扩展（克隆单体）就足够了。水平扩展的复杂度较低，许多云服务都可以用很少的配置来做到这一点。

相比之下，选择分开扩展许多微服务有两个常见原因。首先，如果你的代码具有实质上并行的部分，则在某些情况下将计算块分配给不同的“worker”可能会有些意义。重要的是，相对于每个任务的总计算量，数据传输和加速的开销必须够低。因此，将十个计算块（每个计算耗时 10ms）发送到服务器，开销却为 100ms 就没有并行的价值了。因为顺序执行耗时是 10x10ms=100ms，而并行却是 10ms+100ms=110ms。但是，如果每次计算都花费 100 毫秒，则将它们并行化就能节省时间。

其次，如果资源需求在整个请求中出现变化，则单独扩展各个微服务可能是有意义的。例如，如果一个请求在开始时是受内存限制的，而在结束时是 CPU 限制的，那么就可以将请求的开始部分放在高内存服务中，将结束部分放在高 CPU 服务中。即便如此，除非你是独角兽级别的企业，否则分开扩展服务带来的财务优势可能也无法抵消额外的复杂性。

另外，你试图省钱的做法可能会适得其反：



快乐的云客户

谎言 5：微服务架构性能更高

我在学校学到了一些经验法则。

1. 读取内存所需的时间是读取二级缓存的 10 倍
2. 读取硬盘驱动器所需时间是读取内存的 10 倍。
3. 从网络读取所需的时间是从硬盘读取的 10 倍。

这些数字是粗略的近似值。我们来看一下数据中心中的网络通信与从内存读取之间的实际差异：2009 年，从内存中顺序读取 1MB 的耗时估计为250000ns；2019 年，在 AWS 数据中心中，两个 EC2 实例之间的通信速度可以达到5Gbps。

简单算一算：

- 2009 年的内存：0.25 毫秒内 1 兆字节
- 2019 年的数据中心：1 秒内 5Gbit=1 秒内 0.625GBytes=0.64 秒内 1 兆字节

觉得差距不算大？可我们要意识到：

- 上面的网络速度是最好的情况
- 我们正在对比 2009 年与 2019 年的指标
- 要通过超高速的 AWS 网络发送这一兆数据，我们仍然需要从内存中读取它。

假设你只有一个依赖项，那么这也意味着几千倍的速度差距。实际情况中这一差距还会大得多。

难怪我们现在使用字节流来让每个请求快那么几毫秒。当然，对于字节流来说，调试服务间通信也是需要工具的。

谎言 6：管理多个服务并不难

软件工程师喜欢自欺欺人。也许你以前听过，什么“不需要很长时间”“请给我几个小时”或“我可以在周末完成任务”，诸如此类。优秀的项目经理会理解这一点，并

将工程师的估算值乘以四（或四十……）。

使用微服务的决策也会有同样的乐观情绪。这项工作并不是 " 为所有人获取 AWS 证书并移动一些代码 " 那么简单。实践中会有大量意外开销。下面是你需要的一些人员和工具类别：

1. 架构师。你需要一些人来绘制美观且过于简化的图表并做演示。
2. 发布管理。现在，你需要协调各个部署并管理多个 pipelines。这种协调工作将需要通用工具链，还要有团队来维护这一工具链。
3. DevOps。“常规”工程师既没有专业知识，也没有意愿来正确配置他们的服务。他们也很难正确处理安全性问题。
4. 数据工程师。如果你很幸运能够按照这些建议来成功分解数据存储，那么你现在需要一个团队来将这些数据提取到一个地方进行分析。
5. 配置文件。虽然一些额外的 YAML 文件听起来并不那么糟糕，但这里会出现最危险的错误。它们也难以测试和调试。

你不仅需要支付所有这些额外人员的薪酬，而且还指数级增加了工程组织中的沟通渠道数量。这会拖慢所有人的步伐。

谎言 7：如果你从头开始精心设计微服务，它们将会起作用

这里我引用一段文字：

> 正常运作的复杂系统一定是从一个正常运作的简单系统演变而来的。从头开始设计的复杂系统永远无法正常工作，也无法靠打补丁来正常运作。你必须从一个简单系统起步。——Gall 定律

总结

既然有这么多如此明显的缺点，为什么微服务还这么受欢迎呢？

我相信大多数工程师（包括我本人）都有一定程度的自我能力否定倾向。很多时候，我们需要面对自身能力不足以应付的状况，却依旧要跨过眼前的障碍。在这种情况下，

依靠他人的成果和“最佳实践”是更安全的。但是，我们很快就认为这些“最佳实践”是经过深思熟虑的，或肯定适用于我们的问题。当你启用更多服务时，云供应商会受益。微服务倡导者在你购买他们出的书时也会赚钱。他们俩都有动力向你兜售你本来用不到的技术。

不管怎样，我认为在某些情况下微服务可能是正确的选择。如果你是谷歌或 Facebook 那样的企业，并且要应对数十种产品上数以十亿计的活跃用户，那么单体架构肯定是不够的。如果你有大量可并行化的任务，那么只用单体也是不行的。

我的目的是要告诉大家，后端服务设计是非常重要的，没有哪种选择是银弹。无论我们是在谈论微服务还是单体，SQL 还是 NoSQL，Python 还是 Node，本质都一样。任何技术都不可能完美适应所有用例。

因此，你应该认真思考各种想法，质疑所有假设并清醒地做出架构决策。你的选择可能会成就或拖垮你的公司。

英文原文：

The seven deceptions of microservices

不要盲目跟着JavaScript的趋势走

作者 Nikola Đuza 译者 平川

> 在生活中，炒作和兴奋有时是有用的。没有它，生活将会乏味而无聊。偶尔跟风可能会让你精神振奋，但是你首先应该自己做好调查。当尝试采用一个被大肆宣传的全新的库或框架时，要先进行研究和测试，并听取他人的意见。

本文最初发布于 Pragmatic Pineapple 博客，经原作者授权由 InfoQ 中文站翻译并分享。

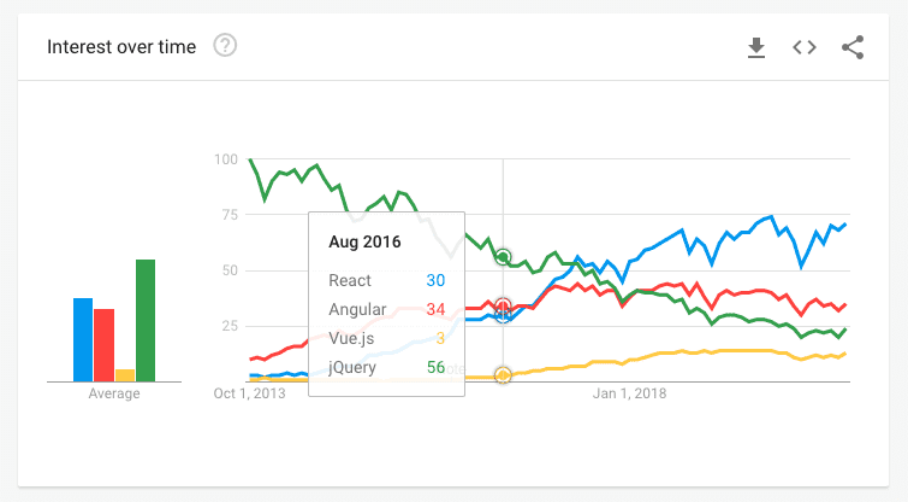
有一天，你在浏览器里输入了twitter.com，然后看到了某人发的一条关于如何使用 React Hooks 的新消息。但是，由于某些原因，你的公司或团队并没有转而使用 Hooks。或者，也许你正在使用它们，但不是以一种新的“符合潮流”的方式。也许你正在使用 Vue.js 或者 Angular，但是 React Hooks 无处不在。

这一天，你开始质疑你代码库中的内容是否正确？你是否应该用你刚刚读到的内容来重构那部分逻辑？得出答案后，你开始想象它在自己的代码中会是什么样子。

现在，你突然有了使用它的冲动。你告知团队负责人，或者向整个团队发送消息介绍这个又酷又新潮的方法，然后提出你要开始使用它。

重写代码

不久前，@ThePracticalDev的 Twitter 账户上出现了一本假想著作的封面。早在 2016 年，取笑多变的 JavaScript 世界就是一种时尚，虽然方式与今天有所不同。



嘘，我发明了时间机器（不要告诉任何人）！让我们闪回到 2016 年。嗖的一声！我们到了。JavaScript 生态圈看起来是这样的：

> 如果你正在使用 JavaScript 框架或是想要使用一个框架，你可能会选择 Angular.js。但是，你很快就会看到这样的消息：Angular 2 会需要你重写几乎所有的东西。而且，React.js 即将到来，并日渐成熟。当然，还是有使用 Vanilla JS 和不使用框架的人。2016 年，不使用框架仍然是一个流行的观点，但这个观点正在慢慢消失。

在了解了这一切之后，你会怎么做？你会选择哪条路？为什么？答案似乎很明显，因为你来自未来。但是，如果你之前决定使用 Angular.js，那么几年之后，你将尝试使用新的 Angular 版本并重写你的代码。如果你选择使用 React，你将成为一个幸运的赢家，因为现在每个人都搭了 React 的便车。现在，你可能想放弃类组件，借助那些妙不可言的钩子使用函数组件，对吧？好吧，至少它不像 Angular.js 到 Angular 2 的变化那么大，不需要学习全新的 API，对吧？

选择这么多，时间这么少。我们该怎么做？

不管我们现在选什么，过去选什么，这都不重要。我们仍然会被诱惑或者不得不重写我们的代码。这样做的理由可能有许多：

- 你的公司以前使用 [框架名]，但现在已经无法招聘到新人了；
- 你觉得以前的解决方案不再有效，需要引入一些新东西；
- 你屈从于行业趋势，想要使用最新最好的。

除非我们打破这个循环。

打破循环

不断改进并提供一个更好的新版本已经成为我们这个行业的基因。我们总是非常迫切地希望制定更高效、更简单、更巧妙、更健壮的解决方案。违背不断学习和进步的理念，就会走到现如今一切人和事的对立面。我现在不打算走这条路，但是如果你想在将来听到更多关于这方面的信息，可以考虑订阅这份简报。

学习新东西的想法是好的，我同意这一点，但是你应该多久学习一次呢？看看

JavaScript 的世界吧，这里经常会出现新的想法、博文、库、框架和某个不知名的新玩意。当它变得越来越流行，人们很快就会尝试采用它。我并不是说你不应该采用新的东西，也不是说你不应该考虑解决方案的不同方法，完全不是！我的意思是，降低下频次。

让我们更加务实点。我以前使用过axios，它非常棒。你可以适当地测试它，它获得了广泛的支持，有很多的点赞（GitHub 星），等等。然后，我看到一篇博文，它告诉你替换axios 并开发自己的获取逻辑。

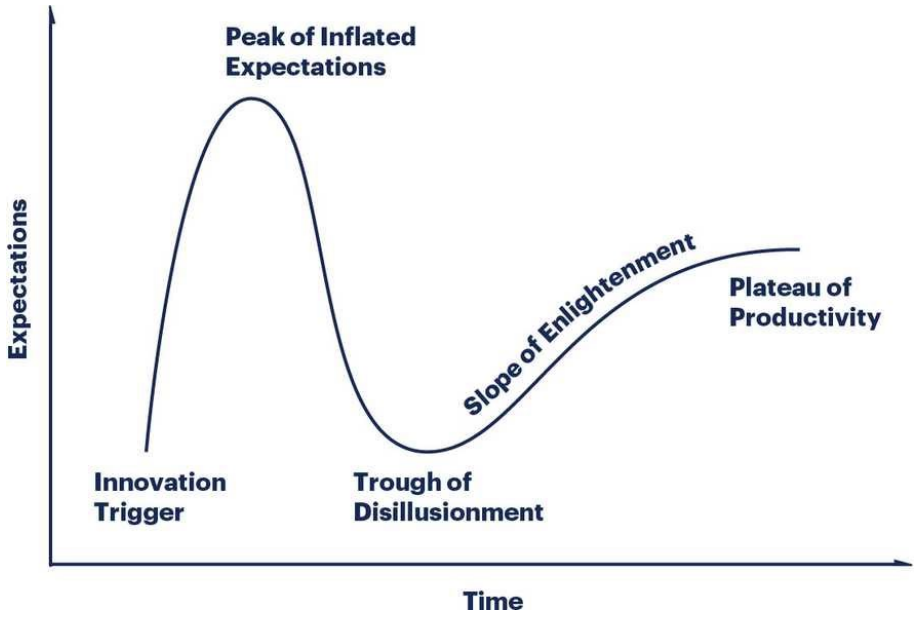
在读了这篇文章的标题“用一个简单的自定义fetch 包装器替换axios”之后，你会从头开始思考这个问题，质疑自己的选择。

我不会详细探讨你是否应该按照这篇博文所说的那样做，那篇文章本身就很好地做到了这一点。我可以帮你做基本的决定。你现在对axios 满意吗？如果答案是肯定的，那么最好不要考虑替代它。对你或你的团队来说，axios 会带来困难吗？如果答案是肯定的，那么就按照博文所说的去做，看看效果如何。

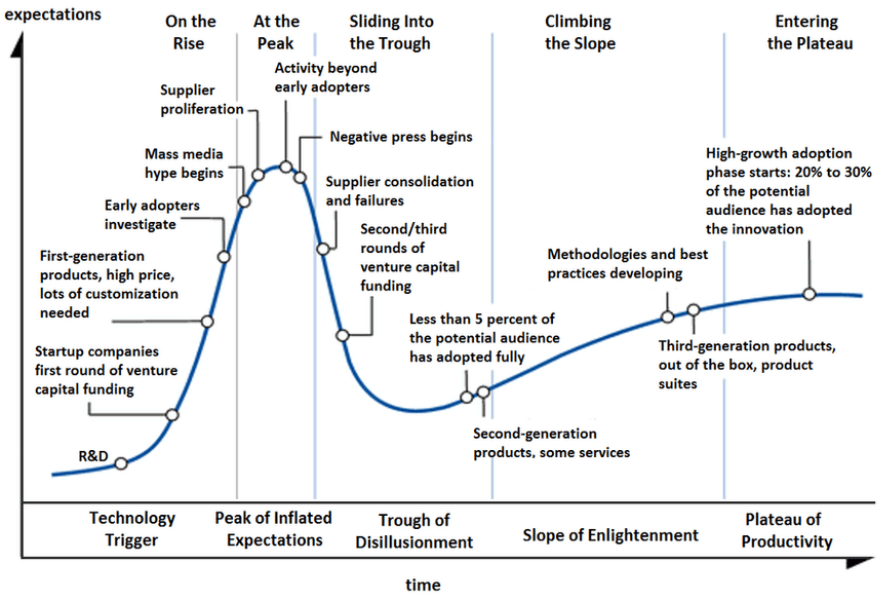
简而言之：不要轻信炒作。试着去“感受”什么对你有效，然后再去做。试着不要盲目跟随那些炫技的新推文、博客文章、Hacker News 热文、你应该或不应该做什么的热门话题标签。

炒作驱动的开发

炒作在我们行业很常见。还记得NoSQL 吗？或者是每个人都为之疯狂的微服务？或者是人工智能/ 机器学习的爆发？这样的例子不胜枚举。人们总是会对新的、突破性的技术和想法感到兴奋。Gartner 在描绘技术炒作周期方面做得非常出色：



上图展示了一个典型的新兴技术的生命周期。你是否意识到你现在使用的任何东西都可能会落在图表的某个部分？Ayman做了一个更详细的炒作周期图：



对照这张图回想一下，就最近的 JS 趋势来说，它处于哪个位置？

如何应对炒作

在生活中，炒作和兴奋有时是有用的。没有它，生活将会乏味而无聊。偶尔跟风可能会让你精神振奋，但是你应该首先自己做好调查。

当尝试采用一个被大肆宣传的全新的库或框架时，请记住这一点。问问你自己和你的团队：

在做决定之前研究并测试了吗？

阅读博客文章、推特和公告有帮助，但更好的做法是，不管某个东西是否适合你，你都要从中获得经验。如果你计划用什么，就尝试构建一个原型。看看它是如何与你正在做的其他事情“共舞”的。

如果你计划在团队层面上做一些事情，可以尝试团队黑客马拉松。黑客马拉松是与你的团队一起测试新技术的好方法，也是你为解决方案疯狂的地方。然后，你可以和团队进行某种回顾，讨论利弊。

它解决了你的问题吗？代价是什么？

你当前的实现有什么特别的问题吗？如果是的话，测试一下，看看新技术是否能解决这个问题。要花多少时间？学习它和重写你的解决方案值得吗？这会在多大程度上减缓团队的开发工作？

听取他人的意见了吗？

如果你在一家小公司工作，或者团队里的成员没那么有经验，这个问题可能会很棘手。试着征求架构师或高级工程师的意见。不能仅仅因为某个库适合 AirBnB 和他们的网站，你就要采用它，可能对你来说它不是最好的，你可能忽略了其中的某些方面。有时候，与有经验的人交谈是一种特权，如果你有，就好好利用它！

如果你是一名高级工程师，试着和一名初级工程师或者没你那么有经验的人交谈。许多公司都在实施所谓的“反向辅导”项目，由初级员工指导公司的资深员工。资深员工的经验可以换来初级员工的新观点。你会惊讶于自己能学到和分享的东西。

总之，尽量不要在你刚刚看到某个东西时就匆忙做出决定。

如果你喜欢这篇文章，可以把它分享给你的朋友和同事。如果你有什么想法，也可以通过推特（*@nikolalsvk*）联系作者。

作者介绍：

Nikola Đuza在塞尔维亚诺维萨德工作和生活，主要使用 JavaScript 和 Ruby 进行开发。你可以在Twitter 上关注他。

原文链接：

Do Not Follow JavaScript Trends

扫码关注InfoQ公众号

Geekbang. 极客邦科技 | InfoQ