

架构师

ARCHITECT



热点 | Hot

OpenAPI规范3.0版即将发布
春节期间你可能错过的IT大事件

推荐文章 | Article

谈谈技术选型

服务拆分与架构演进

观点 | Opinion

左耳朵耗子：

GitLab误删除数据库的几点思考



卷首语

微服务是一种文化

可能有人会不以为然：微服务不是技术架构吗？我们自己的系统就正在微服务化呢！

不要急，我们慢慢聊。

2016 年，“微服务体系”几乎成为了各家架构师们宣讲必备的技术热点。现在回头来看，这一年其实正是容器集群编排与管理集中爆发的一年，而容器，这个可以把应用程序完整封装起来的技术实现，恰如其分地描绘出了一个看得见摸得着的运行单位。相比以前焦头烂额也解释不明白的“服务”两个字，一个敲敲命令行就可以执行起来的容器镜像实在是“天地良心”。

很多组织都公开宣讲过自己的微服务架构实践和落地经验，其中十有八九都能归结于容器化架构的实施和上线。也难怪一些老牌的微服务架构的先驱（比如 Netflix 和 Pivotal）纷纷表示摸不着头脑：咱前好几年费老鼻子劲折腾出来的微服务体系，真就这么被一家容器创业公司给实现了？

其实，微服务架构之所以能够被现在的架构师们所逐渐接纳，成体系的服务编排与管理理论的成型和落地确实是背后最值得关注的推动力量。



有趣的是，这套理论本身成型已经多年，而能够真正落地，还真是多亏了容器技术这趟“东风”。在此之前，Spring 框架通过依赖注入和切面编程，对应用程序强制灌输良好的架构规范和配置标准几乎是我们能接触到的唯一的微服务落地手段。而且这套标准的强制力在应用的开发周期结束后就戛然而止了，只有在容器技术成熟之后，微服务理念才得以扩展到了运维的范畴，再加上 Google Kubernetes 等集群管理项目的推波助澜，各种各样的微服务落地实践才得以应运而生。得益于这些开源容器集群管理技术的繁荣，当我们再谈起微服务，诸如服务注册、发现、负载均衡、服务容错和健康检查等一系列以往只出现在 Borg、Netflix 基础设施里的核心理念，我们国内的技术人员们也能信手拈来了。

容器技术确实从头到脚改变了微服务理念的落地方式，我们现在的架构师们也已经习惯于在技术分享时用“微服务”来代替以前的“应用”、“Tomcat”，但这远远不够。我们必须认识到，将 Tomcat 放进容器里运行绝对不可能修正一个拙劣的架构设计；反过来，优秀的架构师分离出来的实现单位也完全不需要容器的包装就能满足单一职责和高可扩展的要求。容器技术在微服务体系里的本质作用是提供了微服务单元部署的便利

性，而微服务思想的实现则必须依赖于从设计初始就贯彻在程序中的良好的“变成言行”。这些言行包括了小到变量命名和代码规范的制定，大到系统接口的设计和分布式协调的选型，都缺一不可。这也是为什么我们没必要把微服务和容器技术绑定起来，因为在贯彻微服务理念的能力上，容器技术恐怕还不如一套 Spring 框架来的有效。归根到底，容器只能为我们提供实施微服务的形体，而这个形体的灵魂，却只能扎根于我们每个技术人的头脑中。

有位 Google 的资深工程师曾经这么说过：“我不理解你们所谈论的微服务，因为我认为程序本来就应该是那么设计的”。微服务的思想本身并不玄妙，它更像是一种文化，生长在每个技术团队当中，潜移默化地影响着我们的技术和产品。在这个圈子里，有太多的微服务实践能够让我们学习和模仿，但一位有着良好编程习惯的技术人员恐怕才是解决这个问题的关键。

2016 年，容器技术的繁荣推动了微服务架构风格的第二次普及，却不应该掩盖容器技术本身在良好系统设计规范中的无能为力。在新的一年，相比继续执着于各种酷炫的微服务架构建设和推广，我们不妨尝试去向优秀的开源社区学习强制而高效的 Code Review 和 CI 体系，借此培养技术人员良好的“编程言行”，这恐怕能够让我们在系统架构优化的不懈努力中，取得事半功倍的效果。

张磊, HyperHQ 项目成员,

Kubernetes 项目 PM 和 Feature Maintainer

[深圳站]



主办方 Geekbang · InfoQ
极客邦科技

2017年7月7日 - 7月8日
——
深圳·华侨城洲际酒店

7 折购票(截至3月5日)
立减2040

19大专题，全新出炉

- ◆ 区块链以及金融新技术
- ◆ 推荐系统架构实践
- ◆ 低延迟系统架构设计
- ◆ 移动以及轻应用
- ◆ 创新的智能应用
- ◆ 机器学习架构
- ◆ 大规模存储系统
- ◆ 大数据框架
- ◆ 架构师成长路线
- ◆ 研发团队建设和工程文化
- ◆ 基于微服务的软件新架构
- ◆ 社交网络与视频直播
- ◆ 运维新挑战
- ◆ 云架构新动态
- ◆ 大规模企业级性能优化
- ◆ 安全之战
- ◆ 电商之核心架构
- ◆ 技术创业
- ◆ 研发工具

咨询: 010-89880682 / 18515221946
Q Q: 2332883546
微信: 497788321

进入官网了解详情
archsummit.com



CONTENTS / 目录

热点 | Hot

OpenAPI 规范 3.0 版接近最终发布

独家盘点：春节期间你可能错过的 IT 技术大事件

推荐文章 | Article

谈谈技术选型

2016 年 JavaScript 领域中最受欢迎的“明星”们

服务拆分与架构演进

观点 | Opinion

左耳朵耗子：我对 GitLab 误删除数据库事件的几点思考

漫画 | Comic

程序员过年最怕问到的二三事



架构师 2017 年 2 月刊

本期主编 孟 夕

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

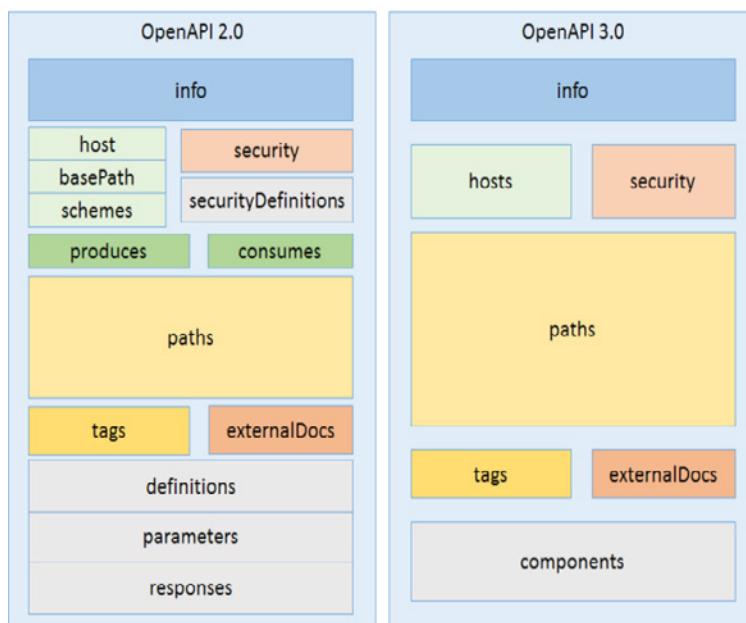
OpenAPI 规范 3.0 版接近最终发布

作者 Abel Avram 译者 Rays

“[开放 API 战略](#)”(Open API Initiative) 发布了 OpenAPI 规范 3.0 版的预览，并规划于今年二月底发布实施草案(Implementer Draft)。

新的 [OpenAPI 规范 3.0](#) 带来了如下重大改进。

为实现更好的可重用性，对规范的整体架构进行了重构，重构后的架构如下图所示。



- 支持oneOf、anyOf和not的JSON模式。
- 可使用模式的参数。
- 引入了Cookie参数，去除了dataForm参数。
- 具有自身实体的主体参数（Body Parameter）。
- 可协商的内容类型。
- 多主机支持。
- 简化了安全定义，改进的WebHooks不再通过回调机制描述。

“开放 API 战略”作为负责规范开发的组织，计划在今年二月底前给出首个带来了。

已公布的 OpenAPI 规范目标是：

定义标准的、独立于语言的指向 REST API 的接口，使得服务能力无需访问源代码、文档，或是借助于网络流量检查，就可被人类和计算机发现并理解。通过对 OpenAPI 做适当定义后，消费者可使用最小数量的实现逻辑理解远程服务，并与远程服务交互。

OpenAPI 基于 Swagger 2.0 构建，Swagger 是 SmartBear 贡献给 Linux 基金会的。意在构建具有中立管理模型的新组织，以引领 Swagger 更上一层楼。“开放 API 战略”的创始成员包括对贡献具有兴趣的 Google、IBM 和 Microsoft。同时还成立了一个技术开发者社区（TDC，Technical Developer Community）以对规范做开发。TDC 对于任何有意向做出贡献的个人都是开放的，无需会员身份。

在对“上百个任务单和上千条评论”近一年的处理工作后，现在 OpenAPI 规范已接近于最终发布，不会再有任何其他重大改进。

独家盘点：春节期间你可能错过的 IT 技术大事件

作者 郭蕾

InfoQ 编辑在这里整理了春节期间的一些重要的技术事件，供大家参考了解，及时充电！

Google 开源 iOS 版本的 Chrome 浏览器

Google 宣布 iOS 版本的 Chrome 浏览器正式加入开源项目 Chromium 中，希望通过这项举措可以让开发者使用 iOS 版本的 Chrome 进行各项测试，并加速 Chrome 浏览器的功能开发。

Google 在 2015 年 5 月将 Android 版本的 Chrome 浏览器开源后，终于在 2017 年 1 月 31 日，正式宣布 iOS 版本 Chrome 加入 Chromium 开源项目。之所以等了这么久，是因为 iOS 系统要求浏览器必须使用 WebKit 渲染引擎，所以 Google 必须完成对 WebKit 和 Blink 两个引擎的支持。

Google 将于 3 月份开源 Google Earth 企业版

Google 官方博客宣布将于 3 月份在 Apache 2 许可证下开源 Google Earth Enterprise（Google 地球企业版，简称 GEE）。GEE 允许开发者构建和托管自己的私有地图和 3D 地球仪。开源的产品包括了 GEE Fusion、

GEE Server 和 GEE Portable Server。Google 是在 2015 年 3 月宣布弃用和终止企业版销售。为了给客户足够的时间过渡，Google 提供了两年的维护期，这一维护期将于 3 月 22 日结束。Google 称，开源 GEE 将给予客户社区继续改进和推进该项目。Google 表示，Google Earth Enterprise 客户端、Google Maps JavaScript® API V3 和 Google Earth API 并不开源。

Facebook 正式关停其旗下 MBaaS 云服务 Parse

1 月 17 日，Parse 联合创始人 Kevin Lacker 最后一次发出提醒，表示 Parse 将在 1 月 30 日正式关闭。在去年 Parse 宣布关闭之后，许多不同的提供商，包括 AWS、Rackspace、Heroku 等都发布了自己的迁移指南。此外，托管公司例如 mLab、ObjectRocket、Buddy 等公司都开始帮助迁移 Parse。Parse 没有公开任何有关已成功迁移的应用程序数量的统计信息，也没有公开还有多少应用程序仍然在访问其数据库，因此很难评估 Parse 的关闭可能会产生的影响。

思科斥资 37 亿美元收购 AppDynamics

1 月 25 日，思科宣布，将以 37 亿美元收购应用性能管理初创企业 AppDynamics，交易预计将在今年 Q3 结束。作为一家应用性能管理公司，AppDynamics 由前 CA 员工 Jyoti Bansal 创办于 2008 年，总部位于旧金山。截至 2016 年 10 月 31 日止，AppDynamics 前 9 个月的净亏损为 9500 万美元，收入为 1.58 亿美元。而上一年同期的数据分别为亏损 1.023 亿美元，收入为 1.02 亿美元。AppDynamics 在公告中表示，与思科的合并将可提供从网络到应用端到端的可视性和智能，再加上安全和伸缩性，可帮助 IT 把业务结果推向新的高度。

微软第二财季盈利超预期：Azure 云服务营收大增 93%

微软 1 月 26 日公布的财报显示，第二财季公司利润超过市场预期，云服务业务成为本季业绩亮点。数据显示，去年 10 月至 12 月，微软总收入 261 亿美元，净利润 52 亿美元。包括云办公软件服务在内的“生产力和商业处理”类业务营收增长 10% 至 74 亿美元；包括 Windows 操作系统、手机和游戏业务在内的个人电脑部门贡献了 118 亿美元营收，同比下跌 5%，包括微软 Azure 云服务在内的“智能云”业务营收增长 8% 至 69 亿美元。Azure 云服务的营收同比增幅高达 93%。

TensorFlow 1.0 预览版本发布

近日，谷歌开源深度学习框架 TensorFlow 发布了完整的 1.0 版本（TensorFlow 1.0.0-rc0）。新版本不仅改进了 TensorFlow 库中的机器学习功能，而且对 Python 和 Java 用户开放了 TensorFlow 开发，并且提升了 debugging。新版本中，一个对 TensorFlow 的计算进行了优化的新编译器，为一系列能够在智能手机级别的硬件上运行机器学习应用程序打开了大门。

Kernel.org 宣布将关闭 FTP 服务器

Linux 内核官网 kernel.org 宣布将在 3 月 1 日关闭 FTP 服务器 ftp.kernel.org，12 月 1 日关闭 mirrors.kernel.org。kernel.org 解释了关闭 FTP 服务器的理由：FTP 协议是低效的，需要向防火墙和负载均衡守护进程添加复杂的程序；FTP 服务器不支持缓存和加速器，严重影响性能；绝大多数 FTP 协议的软件实现已经陷入停滞，很少更新。它因此决定在年底前关闭所有 FTP 服务器，但为了最小化潜在干扰而决定将关闭过程分成两个阶段完成。

甲骨文预计裁员超 1800 人，多为硬件和软件开发人员

甲骨文发布裁员消息称，总共约有 1800 名员工会收到解雇通知书，

大多数都是硬件和软件开发人员。甲骨文在其圣克拉拉的硬件系统部门裁员约 450 名员工。该公司声称不会因为裁员而关闭圣克拉拉的设备。相反，“甲骨文正在重新调整其硬件系统业务，因此，决定裁掉硬件系统部门的一些员工。”虽然也解雇了管理和工作人员，但绝大多数是硬件和软件开发人员。

苹果正开发全新 ARM Mac 芯片：只负责低功耗任务

根据彭博社消息，苹果正在为 Mac 电脑开发一款基于 ARM 架构的芯片，这款 ARM 芯片将于英特尔芯片共存，并且将只负责低功耗任务。这款 ARM Mac 芯片的代号为 T310，开发从去年开始。T310 的设计与 2016 款 MacBook Pro 上的 Touch Bar 驱动芯片相似。这款芯片采用 ARM 技术打造，将与标准的英特尔芯片共同协作，负责处理器 Mac 电脑的 Power Nap 低功耗模式。

Gitlab.com 因误删数据宕机，目前已经恢复访问

2017 年 1 月 31 日 18:00 (UTC 时间)，GitLab 通过推特发文承认 300GB 生产环境数据因为 UNIX SA 的误操作，已经被彻底删除（后发文补充说明已经挽回部分数据），引起业界一片哗然。2017 年 2 月 1 日 18:14 (UTC 时间)，GitLab.com 恢复在线。通过使用一个之前的 6 小时 备份数据库，GitLab 申明 1 月 31 日下午 17:20 (UTC 时间) 至晚上 23:25 (UTC 时间) 之间的数据已经被恢复并可以在生产环境使用，包括项目、问题、 合并请求、用户、注释，等等。

谈谈技术选型

作者 Marek Kirejczyk

本文的主题是技术选型，作者总结了他见过的各种不靠谱的技术选型方式，比如有的团队会根据社交媒体上的讨论来决定选择哪种架构，也有的团队会跟风走，哪个热门就选哪个。但总体来看，这样简单粗暴的方式一定会为未来埋下隐患。那为什么这样说？我们应该如何做正确的选择？且听作者细细道来。另本文译者余晟，曾经是主力程序员、技术文章的写作和翻译爱好者，现在在沪江负责研发和软件架构。欢迎关注他的个人公众号“余晟以为”，了解一位非典型 IT 人员对世界的看法。

软件开发团队所做的软件架构或技术栈的决策，很多并没有经过踏实的研究和对目标成果的认真思考，而是不准确的意见、社交媒体的信息，或者就些是“热闹”的玩意。我称这种作派为“热闹驱动开发（Hype Driven Development, HDD）”，眼见它的危害，我赞成更专业的做法，就是“脚踏实地的软件工程”。下面我们一起看看 HDD 的来龙去脉，想想能如何改进。

新技术带来新希望

开发团队把最新最热的技术应用到项目里，这样的情景你见过吗？有人是因为读到了相关的博客，有人是看到了 Twitter 上的潮流，还有人是刚刚在技术大会上听到了关于某门技术的精彩演讲。不久，开发团队就开始采用这种时髦的新技术（或者软件架构设计范式），结果他们却没法更快（就像之前说的那样）开发出更优秀的产品，反而身陷囹圄。开发的速度降下来了，信心受挫了，后续版本的交付也出问题了。有些团队甚至干脆专心修 bug，停止开发新功能。他们“只需要多花几天”就能把事情搞定。

热闹驱动开发

热闹驱动开发有很多流派，也有很多渠道介入大家的项目：

- Reddit 驱动开发。在选择技术、架构、设计方案时，团队和个人的决策依据是知名博主的文章，或者 Reddit、Hacker News、博客、Twitter、Facebook、GitHub 以及其它社交媒体上的热门信息。
- 技术会议驱动开发。仔细观察观察，参会回来的家伙们有什么表现。他们听了演讲兴致高涨。然而这是双刃剑。他们没有做足够的研究，就开始使用最新最热的类库、框架、架构范式，于是可能踏上通往地狱的高速公路。
- 噪门驱动开发。有人整天谈论新框架/类库/技术，他自己其实没有实际经验，但是反复念经终于让团队决定采纳他的意见。
- Gem/类库/插件驱动开发。在 RoR 社区里特别流行这种情况，有时候我会发现一个 gemfile 太长，唯一比它更长的只有程序启动时的装载时间。这种流派源自下面的观念：Rails 里的每个问题都应当

有个gem来解决。有时候分明只要自己动手写几行代码就能解决，但是我们还是一个劲地添加类库/插件/gem/框架。

- 我还希望提到热闹驱动开发的一个常见流派，StackOverflow 驱动开发。开发人员从StackOverflow（总之就是互联网上）拷贝代码，而没有真正弄懂这些代码。

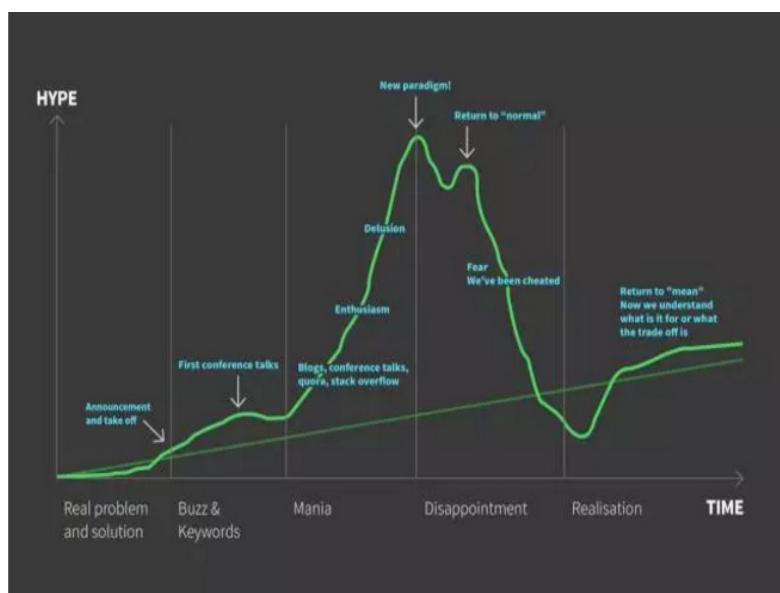
HDD 就是开发团队自掘坟墓

凑热闹的问题是：它很容易导致错误决策。无论是糟糕的架构决策，还是糟糕的技术栈决策，对团队的影响都常常持续数月甚至数年。最坏的结果是造成极其严重的软件工程问题，只能推倒重来。但推倒重来而成功的案例几乎没有。

一切罪恶的根源似乎都是社交媒体——新观点传播得太快，都还没来得及经过检验。大家还来不及细想有哪些利弊，这些观点就已经传播开了。

凑热闹的起承转合

大多数凑热闹的过程是相同的，像下面这样：



第一阶段：真实问题和解决方案

热闹的来源是，某些公司真的遇到了问题。这些公司里的开发团队认为，现成的技术栈、流程、架构并不能解决这个问题，必须自己动手。所以他们研发了新的框架、类库、范式，问题迅速解决了。

第二阶段：宣示、推广、包装关键词

团队热衷于向他人展示自己的成果。很快他们就发布了博客文章，也去技术会议上演讲。这些问题通常是有难度的，所以解决方案是有分量的，结果也是很可观的，开发团队对此很自豪。其它人也开始对这项新技术有了兴致。唯一的问题是，并非所有兴致勃勃的人都能彻底理解问题本身和解决方案的细节。毕竟，问题有难度，解决方案也有分量，所以不是一条推文、一次碎碎念、甚至是一篇博客就能讲清楚的。利用博客文章、技术大会的主题演讲之类的社交工具，原始信息就很容易走样。

第三阶段：狂热现身

HDD 阴影下的开发人员都会阅读博客、参加技术会议。然后，世界各地的开发团队都开始使用新技术了。因为信息已经走样了，所以有些人会在框架问题上做草率的决定。哪怕新框架没有解决任何具体问题，开发团队仍然期望新的框架会带来帮助。

第四阶段：心灰意冷

新鲜劲头过去了，新技术并没有给团队带来期望的改进，反而增加了很多额外的工作。大家得重写很多代码，花不少时间专门学习。工作的速度慢下来，管理者也没耐心了。大家都感觉被骗了。

第五阶段：反省领悟

最终团队做了复盘，认清了追逐这项新技术的代价，也认清了新技术适合解决的问题。大家都变聪明了，直到再次凑热闹为止。

HDD 举例

来看看几个热闹驱动开发的例子，看看它是怎么发生的。

举例 1：React.js

- Facebook遇到了一个问题，Facebook自己的复杂单页面应用里会出现各种状态改变的事件，必须追踪到发生了什么，并且保持状态的连贯一致。
- Facebook用几个时髦的词包装新范式：函数式、虚拟DOM、组件。
- 追逐热闹的人说：Facebook创造了未来的前端框架。我们现在就把一切用react重写吧。
- 等等！要做的工作很多，但这项投资看不到什么短期回报。
- React非常适用于包含很多实时通知的复杂单页面应用程序，但是对简单应用来说，它不见得合适。

举例 2：TDD被DHH杀死了

- David Heinemeier Hansson (DHH, Ruby on Rails框架的创造者)意识到，Rails的框架里没有对OOP支持很好的架构，所以很难做测试驱动开发。于是他做了个现实的选择：不要提前写测试代码。
- DHH的博客和会议演讲引发了热潮。关键词是：TDD is DEAD。
- 忘了测试吧！我们的领袖说过。一个测试也不要写。我们可不是在假装，而是在虔诚地执行。
- 等等！以前一些能正常运行的代码现在都出问题了。我们新写的代码错误百出。
- TDD无所谓生死。TDD是需要权衡的，权衡因素包括API变化的风

险、既有设计、参与者的水平——Kent Beck。

举例3：微服务

- 庞大的单体系统很难扩展。在某个时候我们可以把它拆成多个服务。如果各个都用QPS之类的指标来衡量，扩展就容易很多，也更容易拆分给多个团队。
- 热闹关键词：可伸缩性、松耦合、单体系统。
- 让我们重写所有的服务！我们的单体系统已经是一锅粥了。得把所有东西都拆成微服务。
- 见鬼！现在系统开发的速度变慢了，部署的难度提高了，我们还花了不少时间在多个系统之间追踪bug。
- 微服务需要团队有充分的DevOps能力，还需要权衡增加系统和团队扩展性，保证投入划算。在你遇到严重的规模问题之前，这样的投资是超前的。微服务是提炼出来的，不是重写出来的。按照Martin Fowler的说法，微服务的门槛可不低。

举例 4：NoSQL

- 在应对高压力和处理非结构化数据时，关系型数据库有不少问题。全世界的团队都在研究新一代数据库。
- 热闹关键词：可伸缩性、大数据、高性能
- 我们的数据库太慢，而且容量不够。我们需要NoSQL。
- 我们还需要联表查询？这可不行。简单的SQL操作现在都越来越有挑战了。开发速度越来越慢，我们的核心问题还没解决。
- NoSQL是用来解决特定问题的（要么是海量的非结构化数据，要么是非常高的负载）。如果专业水平足够高，关系数据库也是应对高负载和处理海量数据的好工具。非得使用NoSQL的情况，在2016

年仍然不多见。

举例 5：Elixir和Phoenix（或者是你喜欢的语言/框架组合）

- RoR之类的Web框架不能很好地应付高性能应用、分布式应用、Websockets。
- 热闹关键词：可伸缩性、高性能、分布式、容错性。
- 噢，我们的系统太慢，我们的聊天系统不是可伸缩的。
- 才发现，学习函数式编程和分布式解决方案没那么容易，我们进展真慢。
- Elixir和Phoenix是很优秀的框架，但学习成本太高。如果你确实需要高性能的系统，它的益处要很长时间才会显现。

推而广之

在软件开发的小小天地里，已经有太多领域是热闹非凡的了。在 JavaScript 里，几乎每天都有新框架诞生。Node.js（关键词：事件编程），React 编程，Meteor.js（关键词：共享状态），前端 MVC，React.js……你可以随便举例。软件工程领域里新概念也层出不穷：领域驱动开发，六边形架构理论，DCI 架构（数据 - 场景 - 交互）。你最喜欢哪一种呢？

正面的例子

如果我们不能相信网上的言论或是其他人的说法，那如何做出聪明的选择？下面是一些好的建议：

先测试、研究，再决定

快速搭建原型，不要从博客学习，而要从经验学习。针对新技术提供的功能，在决定采用之前花一两天搭个原型，然后组织大家分析利弊。你可能会遇到若干能彼此替代的技术，可以让团队里不同人用不同的技术来

搭原型。

黑客马拉松，这也是不错的办法，它让大家真正感受到不同技术的代价。对所有兼具风险和诱惑力的技术，都让整个团队花一两天来把玩。这会让大家自主做出聪明的选择，根据自己的经验来决策。

何时开始？

原则上说，应当选择投资回报巨大的时间点开始。大多数技术是用来解决特定问题的。你遇到了那个问题吗？那个问题重要不重要？会不会节省很多时间？新技术带来的好处能不能抵消学习成本和重新的成本？如果我们的开发速度从一开始就降低到正常水平 $1/2$ 甚至 $1/4$ ？想想新技术还值得吗？

优秀的团队有更多自主权——一些团队确实比其他团队更快出成果，他们也更容易厌烦自己手头的工作。这些团队可以更多更快地引入新技术。但这不是省略快速搭建原型或者黑客马拉松的理由。相反，如果这样的团队在交付上遇到了麻烦，一定要加倍小心。

找到对的人

有良好技术背景的人——那些人了解不同的范式，理解编程的理论（算法和并发），受过良好工程文化熏陶，这样的人很少去凑热闹。

有经验的人——年轻的开发人员更喜欢凑热闹。如果有多年的经验，见过许多技术，踩过许多坑，在技术决策时就更容易做出客观的判断。

2016 年 JavaScript 领域中最受欢迎的“明星”们

作者 刘志勇

JavaScript 社区的发展正如盛壮之时的骐骥，一日而驰千里，趋势如长江后浪推前浪。

2016 年已经过去了，你是否会担心错过一些重要的东西？无须担心，JS.ORG 不久前分享了一篇博文，为大家回顾了去年主要的趋势。

InfoQ 翻译并整理这篇博文，以飨 JavaScript 的开发者。

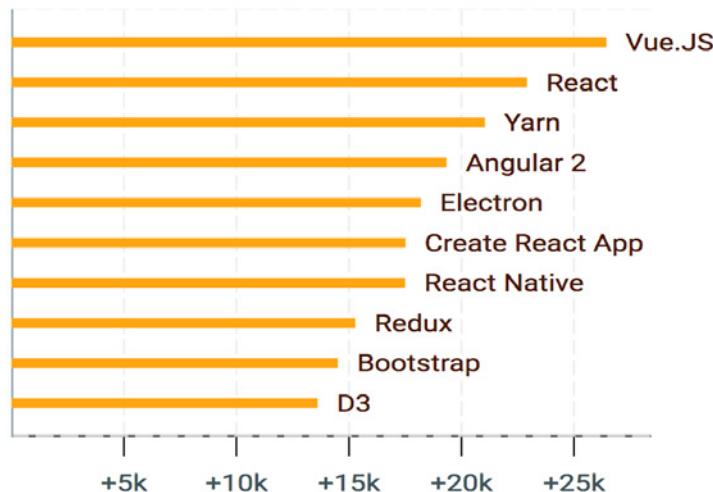
JS.ORG 通过比较过去 12 个月里，在 Github 上增加的星标数，告诉你 2016 年的趋势。

在 2015 年，[React](#) 是社区之王，[Redux](#) 在与 [Flux](#) 的大战中获胜。那么，谁是 2016 年的 JavaScript 的新星？

以下图表比较了 Github 在过去 12 个月中增加的星标数量。[JS.ORG](#) 分析了 [bestof.js.org](#) 的项目，这是一个与网络平台相关的最佳项目的精选列表。

一. 2016 年最受欢迎的项目

#1 Most Popular Projects in 2016



概述

通过一年中最热门的 10 个项目，由此可以很好地了解 2016 年的 Web 开发环境，因为您会发现：

- 3 个 UI 框架：[Vue.js](#)、[React](#)、[Angular 2](#)
- 新的 Node.js 包管理器：[Yarn](#)
- 构建桌面应用程序的领先解决方案：[Electron](#)
- 快速启动新的 React 项目的解决方案：[Create React App](#)
- 移动框架：[React Native](#)
- 最著名的 CSS 工具包：[Bootstrap](#)
- 基于函数概念的状态管理库：[Redux](#)
- 强大灵活的图表库：[D3](#)

以上展现了 2016 年中，JavaScript 表现出了无处不在，功能多样的特性。

而 2016 年的王者是……

Vue. JS 项目在去年的 Github 上获得了超过 25,000 颗星标，将包括 React 和 Angular 的其他框架甩在身后，一骑绝尘。

10 月发布的 Vue. JS 的[版本 2](#)，带来了虚拟 DOM 的性能。

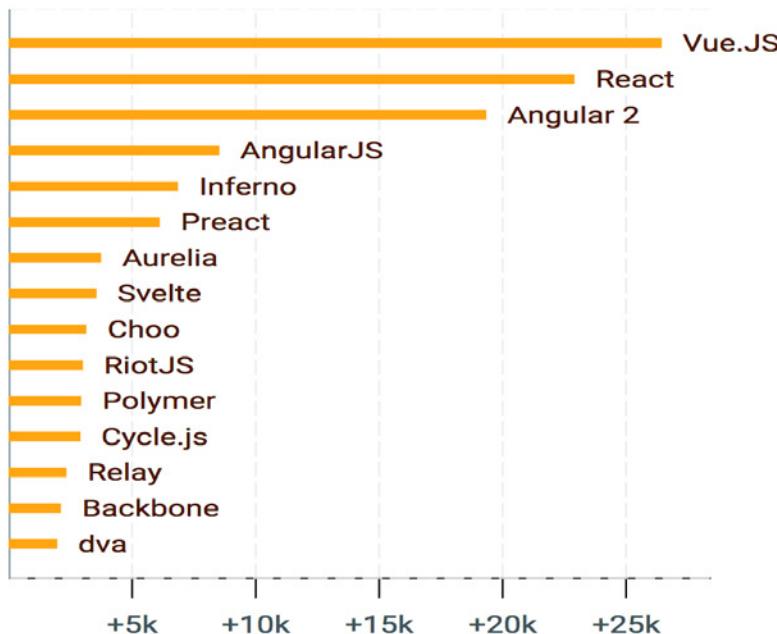
Vue. JS 用于大公司（包括阿里巴巴，中国最大的电子商务公司）的生产，所以你可以认为它是一个安全的选择。

它已经有一个相当成熟的生态系统，包括路由（[vue-router](#)）和状态管理库（[Vuex](#)）。

看来 Vue. JS 采用了最好的 React（组件方法）和 AngularJS（模板是 html 代码增强的框架特性）。

二. 前端框架

#2 Front-end Frameworks



前端框架类可能是 2016 年 [JavaScript 最累的一个排行榜](#)，几乎每个

月都会出现一个竞争者，但是，这推动了创新的步伐。

确切地说，在这个类别中混合了两种类型的项目：

- 完整的框架包含了所有功能，能够创建一个现代的Web应用程序（路由、数据提取、状态管理）。[AngularJS](#)、[Angular 2](#)、Ember 或Aurelia都属于这一类。
- 更轻量级的解决方案专注于UI层，如React、Vue.js、[Inferno](#)……

我们已经提到了总体排名第一的Vue.js，让我们看看其他竞争者。

React及其竞争者

React 总体排名第二，前端开发者没有谁可以忽略 React 及其丰富的生态系统。

React 如此受欢迎，它激发了很多其他库，旨在采取最好的React，没有臃肿，提高在浏览器的性能和构建时间。

Inferno 是这个类别中最受欢迎的项目，它声称是 React 最快的替代品。

在我们的排名中，紧跟 Inferno 之后，[Preact](#) 也是 React 的一个不错的替代品。它的生态系统是相当成熟的，例如有一个具有离线功能的Bolierplates、路由、Compat 模块，以便您可以使用 Preact 项目中的任何现有 React 库。

Angular 1和2

Angular 项目已经拆分为 2 个存储库，因为 Angular 2 是 Angular 1 的完全重写，即使一些概念保持不变。

Angular 2 是用 TypeScript 编写的，并且利用 ES6 提供一个现代和彻底的框架。

AngularJS 项目是 1.x 的分支，它在许多项目中仍然使用，并将继续流行一段时间。

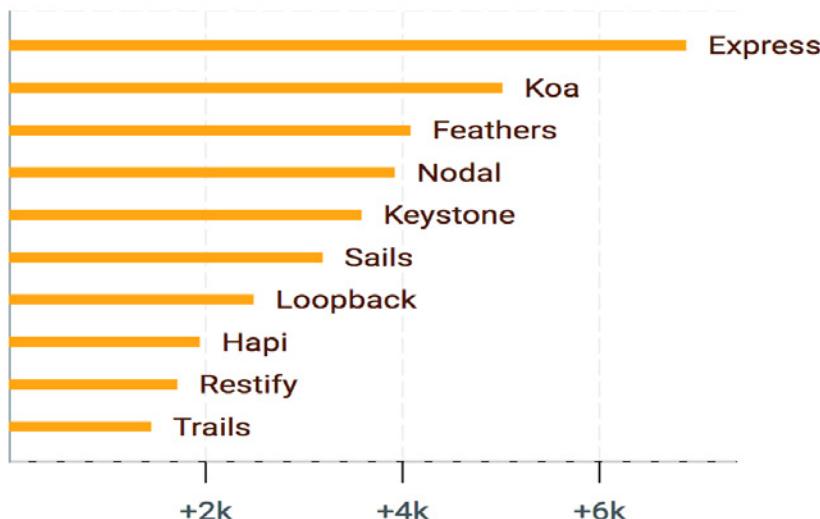
值得一提的是，[Ember](#) 虽然拥有庞大的生态系统，但它的社区并不在前十名。

因此看起来，与其选择“开箱即用”的所有功能的完整框架，2016 年开发商更倾向轻量级的方案，并喜欢组成自己的方案——“点菜”。

在 2016 年调味的更轻的方法，并更喜欢组成自己的解决方案“点菜”。

三. Node.js 框架

#3 Node.js Frameworks



2016 年，使用以下解决方案创建和部署 node.js 应用程序从未如此简单：

- [Now](#)
- [Webtask.io](#)
- [Stdlib](#)

像 [Gomix](#) 这样的项目甚至降低了 Node.js 世界的门槛，使得任何人都可以在浏览器中轻松点击几下来编写和共享 Node.js 代码。

如果你必须构建一个 web 应用程序，你会选择哪个框架？

Express

当你使用 Node.js 构建 Web 应用程序时，[Express](#) 通常被视为事实上的 Web 服务器。它的哲学（一个可以使用中间件包扩展的简约核心）是大多数 Node.js 开发人员熟悉的。

Koa

Koa 的哲学接近 Express，但它是使用 ES6 生成器，以避免有时被称为回调地狱的问题。

Feathers

Feathers 是一个非常灵活的解决方案，创建一个“面向服务”的架构，它是一个很好的适合创建 Node.js 微服务。

Nodal

Nodal 框架以目标无状态和分布式服务连接到 PostgreSQL 数据库。

Keystone

Keystone 是我所知得到一个管理客户端并运行得最好的解决方案之一，以便管理来自 MongoDB 数据库的内容。管理界面自动从模型生成，具有所有 CRUD 操作和精细的过滤器。

Sails

Sails 是一个完整的 MVC 框架，受 Ruby on Rails 的启发（因此名为 Sails！）。它已经存在了很长时间。它可以与任何类型的数据库（SQL 或无 SQL）良好工作。

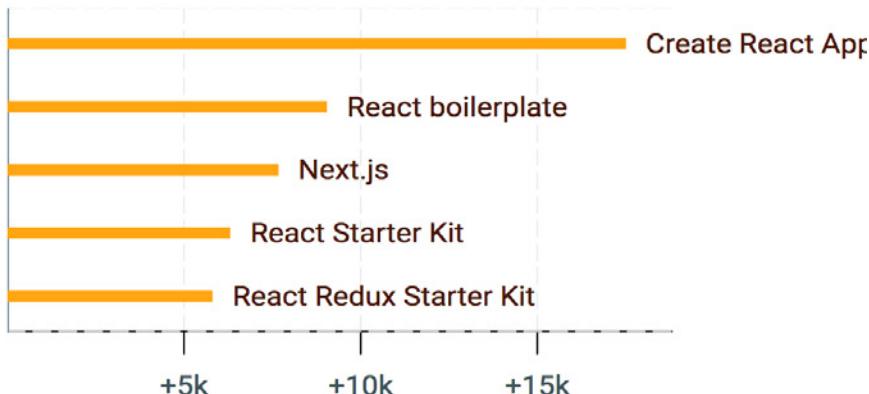
Loopback

Loopback 是另一个成熟的框架，内置许多函数，包括使用令牌和到任何类型的数据库的连接器的认证。

它的杀手级功能是 API 浏览器功能，允许开发人员以直观的方式检查所有 API 端点，并能检查任何用户的令牌。如果你必须构建一个 API，这绝对是一个不错的选择。

四 . React Boilerplates

#4 React Boilerplates



React 是一个伟大的 UI 库，但使用 React 和现代 Web 开发工作流工具需要大量的配置。那么如何开始创建一个应用程序呢？

这是 React 的“Boilerplates”和其他“Starter Kits”提供的答案：

Create React App

Facebook 通过提供一个称为 Create React App 的轻量级方法来解决这个需求，这是一个非常方便的启动一个新的 React 项目。

Dan Abramov (Redux 的创造者，现在为 Facebook 工作) 做了一个伟大的工作，在简单性和功能找到了正确的平衡点。例如，没有花哨的样式

解决方案（只是简单的 CSS），没有服务器端渲染，但是所有的一切，都很好地打包了，开发人员的体验非常棒。

与其竞争者的主要区别是，如果使用 Create React App，它将成为项目的依赖项，所有的魔法是隐藏的，你看到的只是你的应用程序代码。您可以随时升级依赖关系，它并非只是一个起点。

React boilerplate

命名为 [React boilerplate](#) 具有您需要的一切，包括 [Redux](#) 和一些漂亮的离线功能，利用 web workers 技术。

它让开发人员创建所谓的渐进式 Web 应用程序（Progressive Web Applications, PWA）：离线运行的 Web 应用程序，使用一种名为 Service Worker 的技术，请阅读 Nicolás Bevacqua 的这篇[文章](#)。

Next.js

Next.js 由 [Zeit](#) 创建，具有可用于创建通用应用程序的服务器端呈现功能（或同构应用程序，如我们在 2015 年所说），也就是说客户端和服务器端运行的应用程序使用相同代码。

五 . Mobile

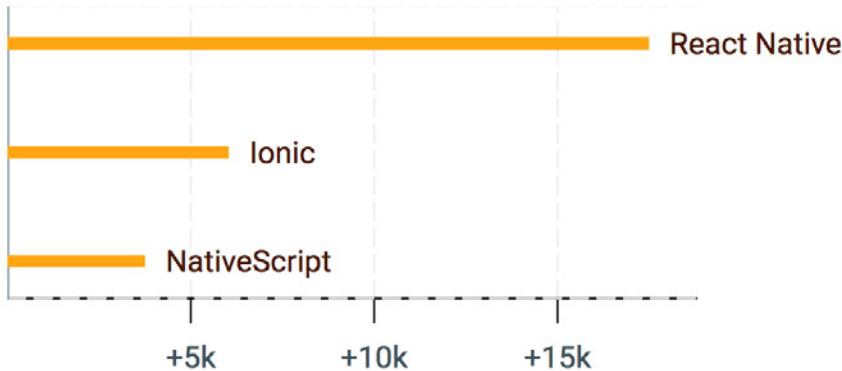
JavaScript 无处不在，你可以使用技术 Web 开发人员已知的任何技术（HTML、JavaScript、CSS）构建移动应用程序。

React Native

使用 React Native，你可以从相同的代码库使用 React 开发人员熟悉的概念构建 iOS 和 Android 真正的原生移动应用程序。要了解有关构建 iOS 和 Android 应用程序的更多信息，请阅读这本[教程](#)。

其他基于 Cordova 的解决方案，依靠 Webview 来渲染屏幕，并且不如

#5 Mobile



原生解决方案那么高效。“一次编写，随处运行”，这是开发人员的梦想成真！

Ionic

Ionic 是“混合”应用程序概念的先驱。在后台中，它基于 Cordova 访问移动设备功能。这是一个非常成熟的大型生态系统。

NativeScript

NativeScript 旨在实现与 React Native 相同的目标（使用 Web 技术构建真正的移动应用程序）。它有两种风格：NativeScript Core 和 NativeScript + Angular 2。

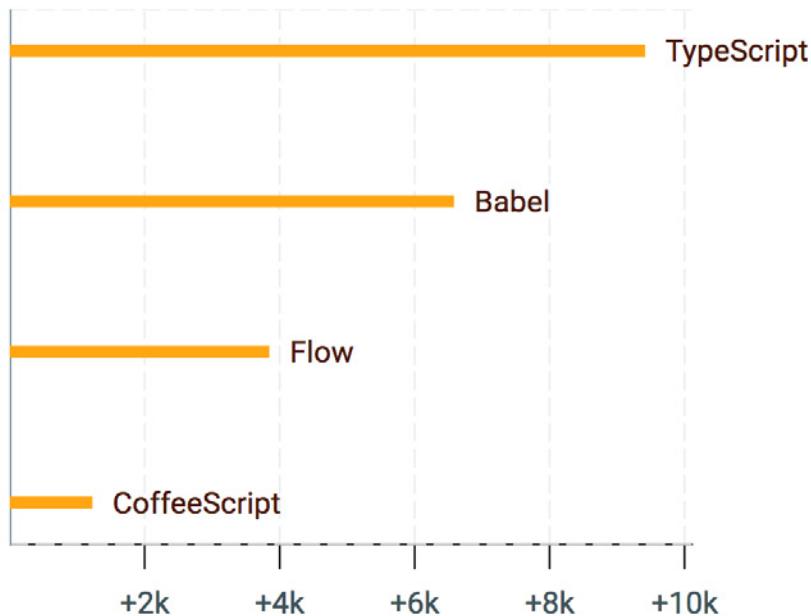
前瞻

在 2017 年密切关注的一个项目：[Weex](#)，一个构建在 Vue.js 之上的移动跨平台 UI 框架。

六. Compilers

我们在这里谈论生成任何语言（或 JavaScript 的任何变体）的 JavaScript 的编译器（或“transpilers”）。它们将代码转换为浏览器（或 node.js）可以执行的“标准 JavaScript”代码。

#6 Compilers



例如，编译器允许开发人员使用最新版本的 JavaScript（ES6）编写代码，而无须担心浏览器的支持。

TypeScript

最时髦的转换器是 [TypeScript](#)，它为 Web 开发人员提供了 Java 和 C# 开发人员使用的静态类型。事实上，Angular 2 使用 TypeScript 增加了更多的牵引力。在 JavaScript 中使用类型有优缺点，阅读这些文章，使你自己的观点：

- [你可能不需要TypeScript](#)
- [TypeScript：缺失的介绍](#)

Babel

Babel 与 Webpack 一起，几乎成为编译 ES5 代码和标准 JavaScript 中的库（如 React（JSX））使用的模板的标准。最初创建用于编译 ES6，

它成为一个更通用的工具，可以完成任何代码转换，拜一个系统的插件所赐。

Flow

Flow 不是一个编译器，它是一个用于“注释”JavaScript 代码的静态类型检查器。基本上在代码库中使用 Flow 意味着添加注释来描述期望的类型（点[阅读](#)更多了解使用 Flow 编写模块）。

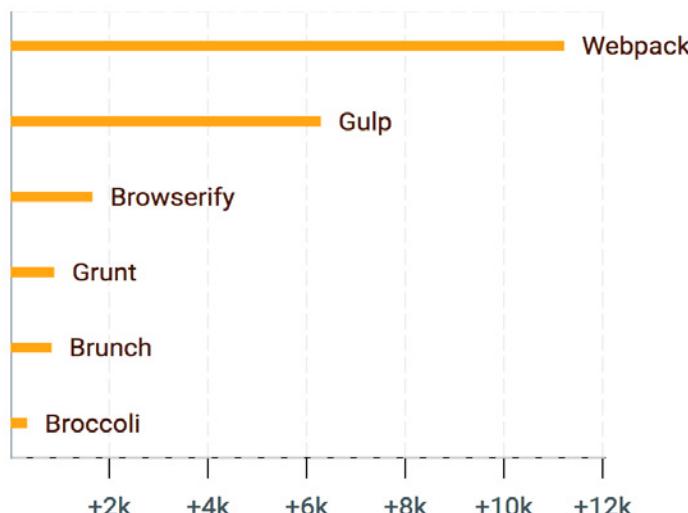
它在 Facebook 项目的代码源内使用。因为 Facebook 成为开源世界的主要角色之一（像 React、React Native、[Flux](#)、[Immutable](#)、[Jest](#) 等项目），这意味着很多。

CoffeeScript

多年来，[CoffeeScript](#) 由于其精简语法（灵感来自 Python 和 Ruby 语法），成为最受欢迎的编译器，但它在 2016 年不太流行，很多开发人员从 CoffeeScript 迁移到 ES6 与 Babel。

七. Build Tools

#7 Build Tools



在 2016 年，很难想象一个没有任何构建过程的 Web 应用程序。通常需要一个构建过程来编译模板和优化资源，以便在生产环境中运行 Web 应用程序。

Webpack

Webpack 是用于构建单页应用程序的主要工具，它与 React 生态系统一起使用。新发布的版本 2 带来了一些令人鼓舞的增强功能（查看这份[介绍](#)）。

Gulp

Gulp 是一个通用的任务运行器，可以用于涉及文件系统的任何类型的自动过程，因此它不是 Webpack 或 Browserify 的直接竞争者。

像 [Grunt](#) 一样，[Gulp](#) 通过聚合工作：你可以要求它缩小和连接资源列表，但是它不会像 [Webpack](#) 或 [Browserify](#) 那样处理模块化 JavaScript 本身。

然而，它可以很好地与 webpack 一起工作，即使开发人员倾向于使用 npm 脚本。

Browserify

Browserify 由于其简单性，受到了 Node. js 开发人员喜爱。

基本上，它需要几个 Node. js 包作为输入，并为浏览器生成一个单一的“构建”文件作为输出。但是似乎一个更有见地的工具像 Webpack 是一个更好地适合 Web 应用程序工作流。

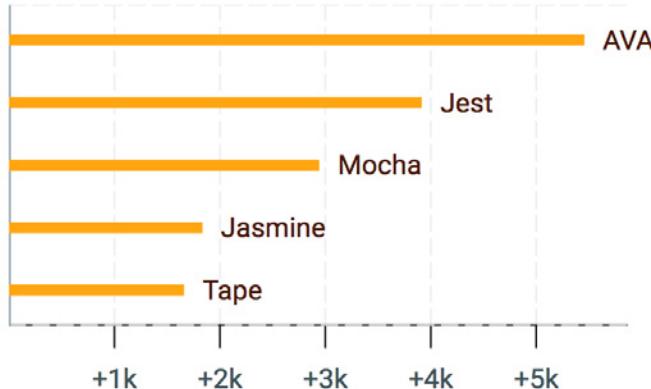
前瞻

2017 年的模块捆绑包，强调性能：汇总 ([rollup](#))。

它使用 ES6 模块与一个称为树摇动 (Tree shaking) 功能创建捆绑包，只包括您在代码中使用的功能，而不是搬运完整的库。

八 . Testing Frameworks

#8 Testing Frameworks



最著名的两个测试框架是 [Jasmine](#) 和 [Mocha](#)，但最近的两个项目在 2016 年有更多的牵引力： [AVA](#) 和 [Jest](#)。

AVA

AVA，由多产的 [Sindre Sorhus](#) 创建的强调性能（并行测试）和 ES6。 AVA 的语法接近标准测试框架，如 [Tape](#) 和 [Node-tap](#)。

Jest

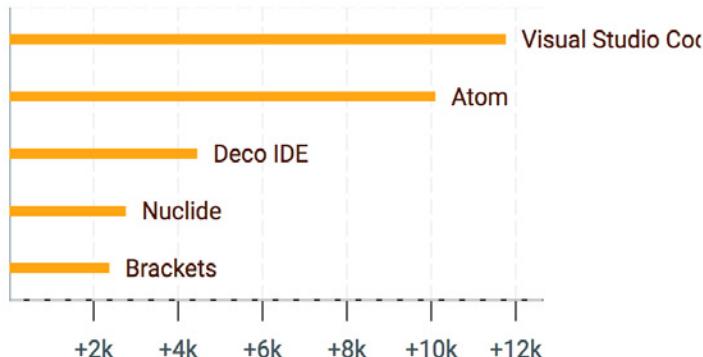
Jest，另一个 Facebook 项目，在过去的几周里得到了很大的牵引力。它在 React 社区中是众所周知的，越来越多的人转向 Jest（阅读[这篇故事](#)来了解），它可能成为 2017 年最流行的测试框架。

Jest 有内置的良好的模拟能力，而其他测试框架通常依赖于像 [Sinon.JS](#) 这样的库。

九 . IDE

关于 IDE（Integrated Development Environment，集成开发环境），值得一提的是，两个最流行的 IDE 是使用 Web 技术开发的开源项目。

#9 IDE



Visual Studio Code

在我们的结果中，Microsoft 凭借 [Visual Studio Code](#) 遥遥领先。

它提供了一个与 TypeScript 和 node.js 的很好的集成。一些开发人员提到关于开发速度，很感谢 IntelliSense 功能（高亮和自动完成的混合）。

在同一句话中提到“开源”和“微软”不再矛盾了！

Atom

Atom 是由 Github 推动的、并且由 [Electron](#) 构建（像其他一些桌面应用程序，包括 Slack 桌面客户端），并非远远落后 Visual Studio Code。关于 [Atom](#) 的一个有趣的事：它的主要语言是 CoffeeScript！

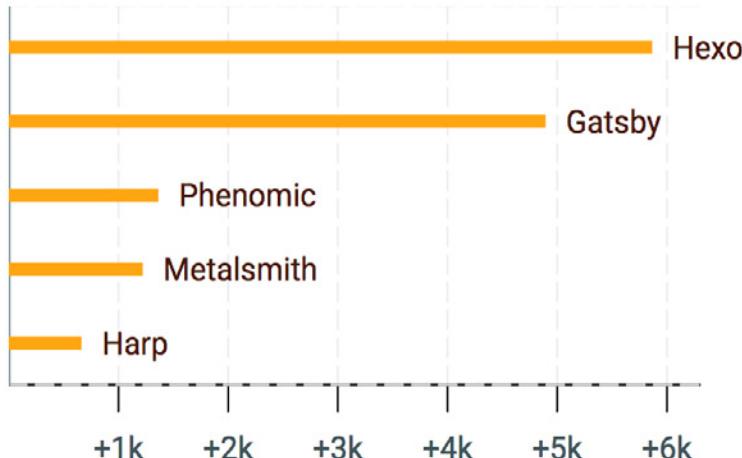
十. Static Site Generators

静态网站生成器 (Static site generators, SSG) 是生成一系列 .html、.css 和 JavaScript 文件的工具，您可以在任何简单的 Web 服务器 (Apache 或 NGNX) 上部署，而不必大惊小怪，或者设置数据库或任何网络框架。正如 [Gatsby](#) 网站所说：

就像 1995 年那样建立网站。

静态网站具备快速性、鲁棒性和易维护性。

#10 Static Site Generators



SSG 非常受欢迎，因为有很多很好的解决方案来主持静态网站免费：

- [Github pages](#)
- [Gitlab pages](#)
- [Netlify](#)
- [Surge](#)
- [Now static](#)

Hexo

在 2016 年，使用 Node.js 构建的最流行的 SSG 是 [Hexo](#)。它是一个彻底的 SSG，接近 CMS 系统，可用于构建一个博客，如 Wordpress。它有很多功能，包括国际化插件。

Gatsby

新来的 [Gatsby](#) 是一个非常有趣的解决方案，它从竞争对手脱颖而出，因为它使用 React 生态系统来生成静态 html 文件。事实上，您可以组合 React 组件，Markdown 文件和服务器端渲染使它非常强大。

总结

尽管存在 JavaScript 疲劳™和戏剧（记住“[左键门](#)”），但对于社区而言，随着像 Vue.js 和 React Native 项目的兴起，以及像 [Yarn](#) 或 Create React App 的新项目，2016 年仍然不啻为一个伟大的年份。

我们一直在谈论的项目，2016 年在 Github 得到了吸睛，但真正重要的是开发者的满意度。所以，如果你想要一个更定性的方法，上 Sacha Greif 查看 JavaScript 调查的[结果](#)，它收集了超过 9,000 的反馈。

以下是我的年度十大选择，代表了在 2016 年我所喜欢的项目和想法，将在 2017 年持续增长：

- Vue.js：势头强劲，不会停止；
- Electron；
- Create React App；
- React Native；
- Gatsby；
- Yarn：一个快速，可靠和安全的依赖管理，可以取代npm，点[此处](#)了解Node.js包管理器的状态；
- 渐进式Web应用程序；
- Node.js微服务使用像[Now](#)这样的托管解决方案很容易部署；
- Node.js的演变：最新版本对ES6语法提供良好的支持；
- 还有一个选择是[GraphQL](#)：据我了解，GraphQL将有大动作。

服务拆分与架构演进

作者 糜娴静

前言

上文提到，微服务帮助企业提升其响应力，而企业需要从 DevOps、服务构建、团队和文化四点入手，应对微服务带来的复杂度和各种挑战，从而真正获益。如果说运维能力是微服务的加油站，服务则是其核心。

企业想要实施微服务架构，经常问到的第一个问题是，怎么拆？如何从单体到服务化的结构？第二个问题是拆完后业务变了增加了怎么办？另外，我们想要改变的系统往往已经成功上线，并有着活跃的用户。那么对其拆分还需要考虑现有的系统运行，如何以安全最快最低成本的方式拆分也是在这个过程中需要回答的问题。

本文会针对以上问题，介绍我们团队在服务拆分和演进过程中的实践和经验总结。

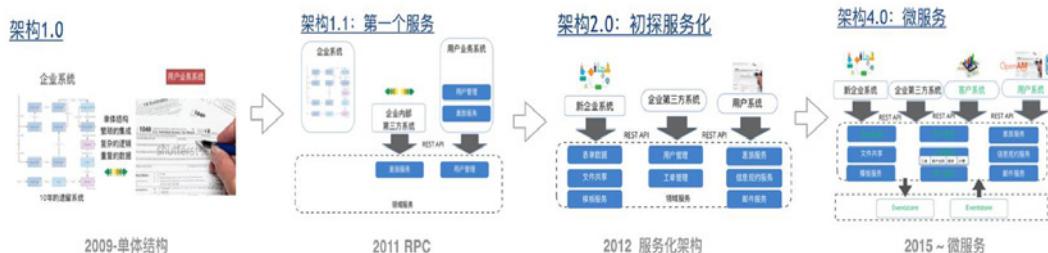
我们项目架构的演化历程

该项目始于 2009 年，到现在已有 7 年的时间。在这 7 年中覆盖的业



务线不断扩大，从工单、差旅、计费、文件、报表、增值业务等；业务流程从部分节点到用户端的全线延伸；系统用户来自 198 个国家，从 10 万增长到 41 万。7 年间打造多个产品，架构经历了多次调整，从单体架构、RPC、服务化、规模化到微服务。

主要架构变迁如下图所示：



在这 7 年架构演进路上，我们遇到的主要挑战如下：

- 如何拆？即如何正确理解业务，将单体结构拆分为服务化架构？
- 拆完后业务变了增加了怎么办？即在业务需求不断发展变化的前提下，如何持续快速地演进？

- 如何安全地持续地拆？即如何在不影响当下系统运行状态的前提下，持续安全地演进？
- 如何保证拆对了？
- 拆完了怎么保证不被破坏？

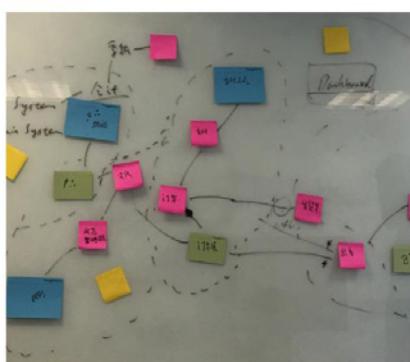
问题 1：如何将单体结构拆分为服务化架构？

就如庖丁解牛一样，拆分需要摸清内部的构造脉络，在筋骨缝隙处下刀。那么微服务架构中，我们认为服务是业务能力的代表，需要围绕业务进行组织。拆分的关键在于正确理解业务，识别单体内部的业务领域及其边界，并按边界进行拆分。

1. 识别业务领域及边界

首先需要将客户、体验设计师、业务分析师、技术人员集结在一起对业务需求进行沟通，随后对其进行领域划分，确定限界上下文（Boundary Context），也称战略建模。

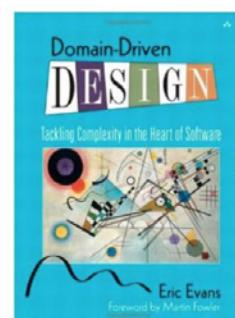
以下我们经常使用的方法和参考的红蓝宝书：



业务流程与四色建模



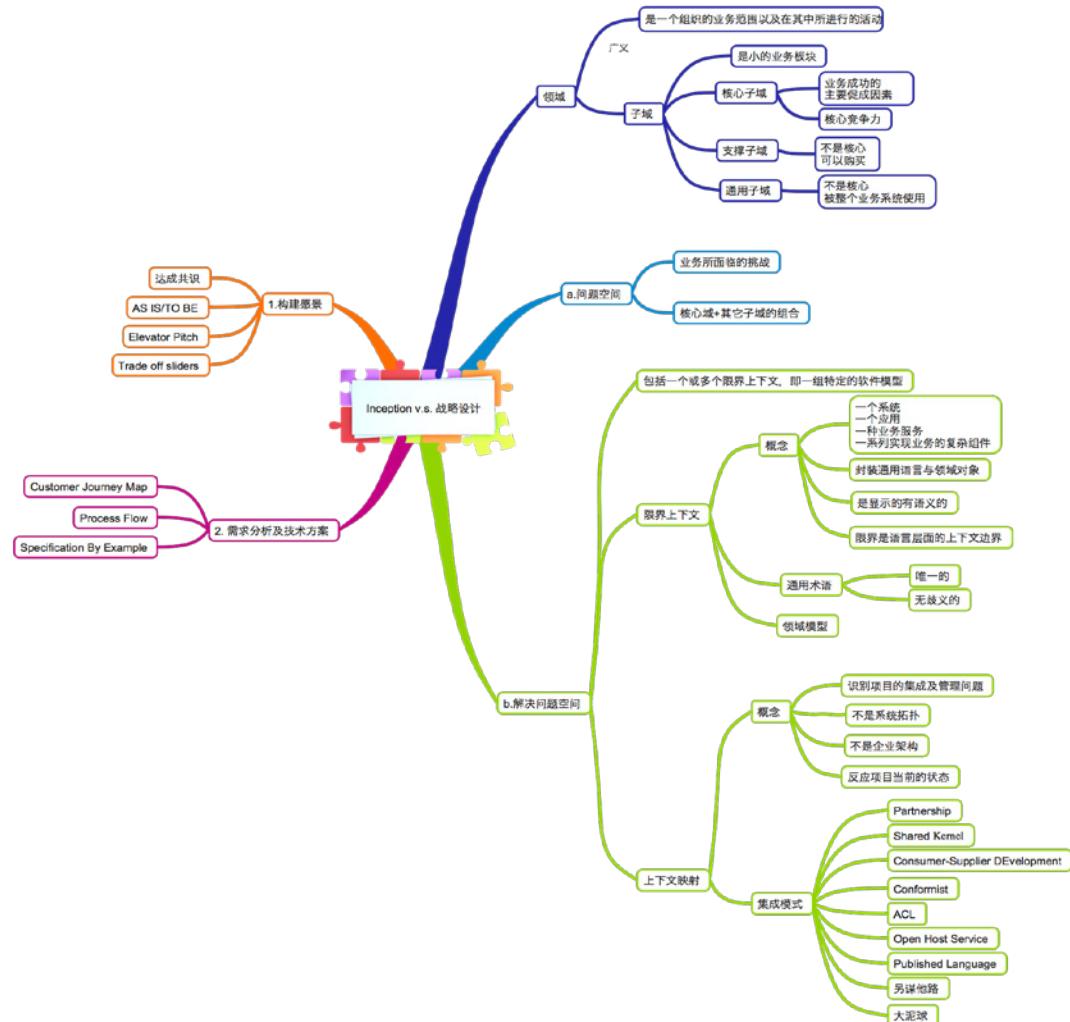
统一语言 领域/子域 限界上下文



- Inception-> [User Journey](#) | Scenarios，用于梳理业务流程，由粗粒度到细粒度逐一场景分析。

- 四色建模，用于提取核心概念、关键数据项和业务约束。
- 领域驱动设计-战略设计，用于划分领域及边界、进行技术验证。
- Eventstorming，用于提取领域中的业务事件，便于正确建模。

下图是 Inception 与 DDD 战略设计的对比。



一个业务领域或子域是一个企业中的业务范围以及在其中进行的活动，核心子域指业务成功的主要促成因素，是企业的核心竞争力；通用子域不是核心，但被整个业务系统所使用；支撑子域不是核心，不被整个系统使用，该能力可从外部购买。一个业务领域和子域可以包括多个业务能力，一个业务能力对应一个服务。领域的边界即限界上下文，也是服务的

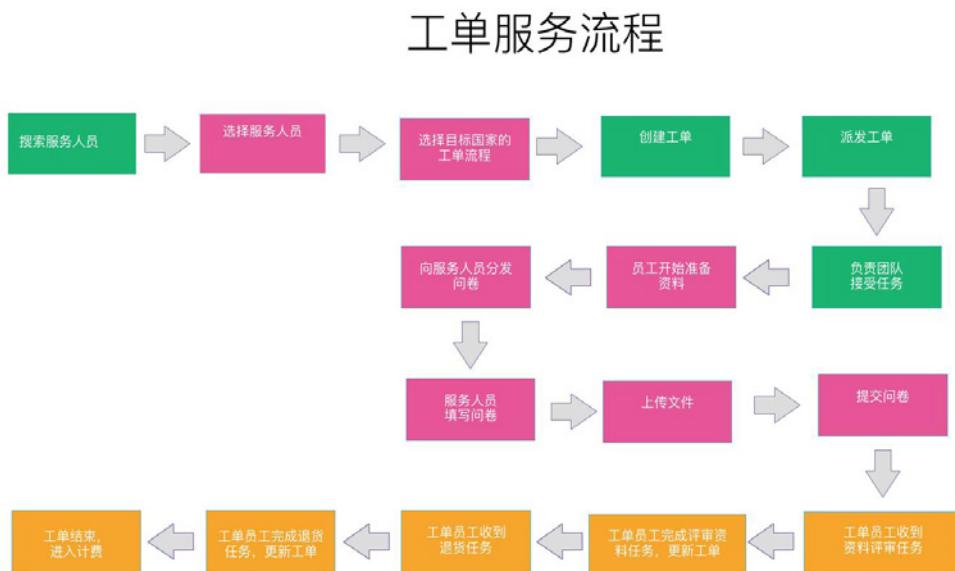
边界，它封装了一系列的领域模型。

一个业务流程代表了企业的一个业务领域，业务流程所涉及的数据或角色或是通用子域，或是支撑子域，由其在企业的核心竞争力的角色所决定。比如企业有统一身份认证，决策不同部门负责不同的流程任务，那么身份认证子域并不产生业务价值，不是业务成功的促成因素，但是所有流程的入口，因而为通用子域，可为单独服务；而部门负责的业务则为核心子域。

举个例子

工单业务流程

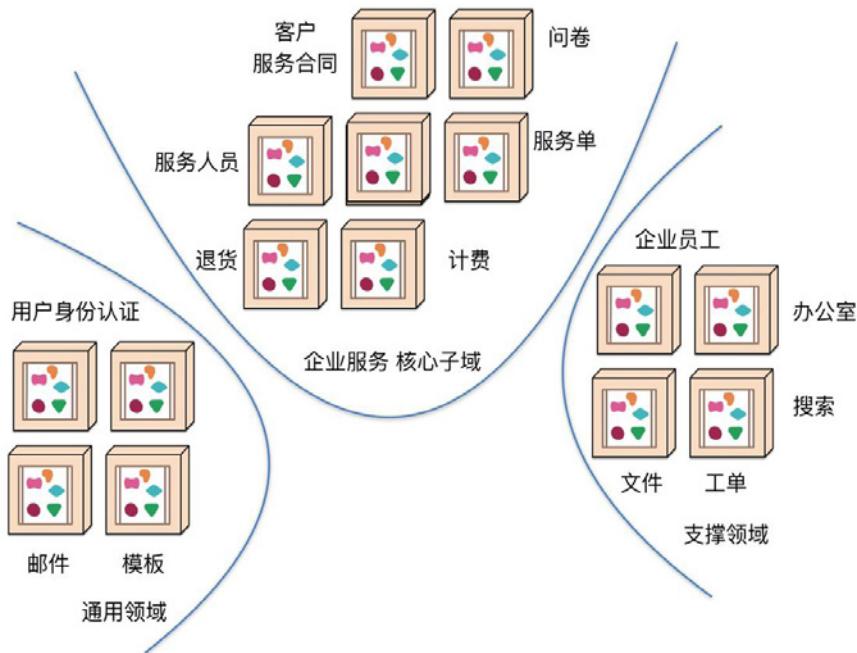
某企业为服务人员提供工单服务的业务流程简化如下。首先搜索服务人员，选取服务人员购买的服务，基于目标国家的工单流程，向服务人员收取资料，对其进行审计，最后发送结果。



识别的领域

其中服务为其核心竞争能力，包括该企业对全球各国的政策理解，即法律流程，服务资料（问卷），计算服务，资料审计服务，相比其他竞争

对手的服务（价位 / 效率等），这些都为改企业提供核心的业务价值，自然也是核心子域。而其他用于统计改企业员工工作的工单，组织结构和员工为支撑子域，并不直接产生业务价值



领域划分的原则

在划分的过程中，经常纠结的一个问题是：这个模型（概念或数据）看起来放这个领域合适，放另一个也合适，如何抉择呢？

第一，依据该模型与边界内其他模型或角色关系的紧密程度。比如，是否当该模型变化时，其他模型也需要进行变化；该数据是否通常由当前上下文中的角色在当前活动范围内使用。

第二，服务边界内的业务能力职责应单一，不是完成同一业务能力的模型不放在同一个上下文中。

第三，划分的子域和服务需满足正交原则。领域名字代表的自然语言上下文保持互相独立。

第四，读写分离的原则。例如报表需有单独报表子域。核心子域的划

分更多基于来自业务价值的产生方，而非不产生价值的报表系统。

第五，模型在很多业务操作中同时被修改和更新。

第六，组织中业务部分的划分也是一种参考，一个业务部门的存在往往有其独特的业务价值。

简单打个比方，同一个领域上下文中的模型要保持近亲关系，五福以内，同一血统（业务）。

领域划分的误区和建议

- 业务能力还是计算能力？往往在划分一些貌似通用的领域，其实只是通用的计算能力并不是真正的业务能力，只需采用通用库的方式进行封装，而无需使用服务的方式。如我们系统的模板服务，是构建通用的模板服务，服务于整个平台的服务；还是每个服务拥有独立的模板模块？
- 尽早识别剥离通用领域，如身份认证与鉴权领域，是企业系统中最复杂有相对多变的领域，需要及早隔离它对核心业务的干扰。
- 时刻促成技术人员与客户、业务人员的对话。业务领域的划分离不开对业务意图的真正理解。而需求人员和体验设计师对于User Journey的使用更熟悉，而技术人员、架构师对领域驱动设计、Eventstorming更熟悉。不管哪种方法都要求跨角色的群体的协同工作，即客户人员、业务分析师、体验设计师与技术人员、架构师。而现实的情况中，User Journey更多的在Inception，在需求阶段进行，而领域驱动设计、Eventstorming更多的在开发设计阶段被使用，故而需求阶段经常缺失技术人员，而开发设计阶段经常缺失客户、业务人员的参与。

另一个常见的现象是，Inception 的参与人员和真正的开发团队有可

能不是同一个群体，那么 Inception 中的业务沟通往往以 UI 的方式作为传递，因此在开发中经常只能通过 UI 设计来理解业务的真正意图。

所以要想将正确的理解业务，做对软件，需要时刻促成技术人员与客户、业务人员的对话。

识别了被拆对象的结构和边界，下一步需要决定拆分的策略和拆分的步骤。

2. 拆分方法与策略

拆分方法需要根据遗留系统的状态，通常分为绞杀者与修缮者两种模式。

绞杀者模式

指在遗留系统外围，将新功能用新的方式构建为新的服务。随着时间的推移，新的服务逐渐“绞杀”老的一流系统。对于那些老旧庞大难以更改的遗留系统，推荐采用绞杀者模式。

修缮者模式

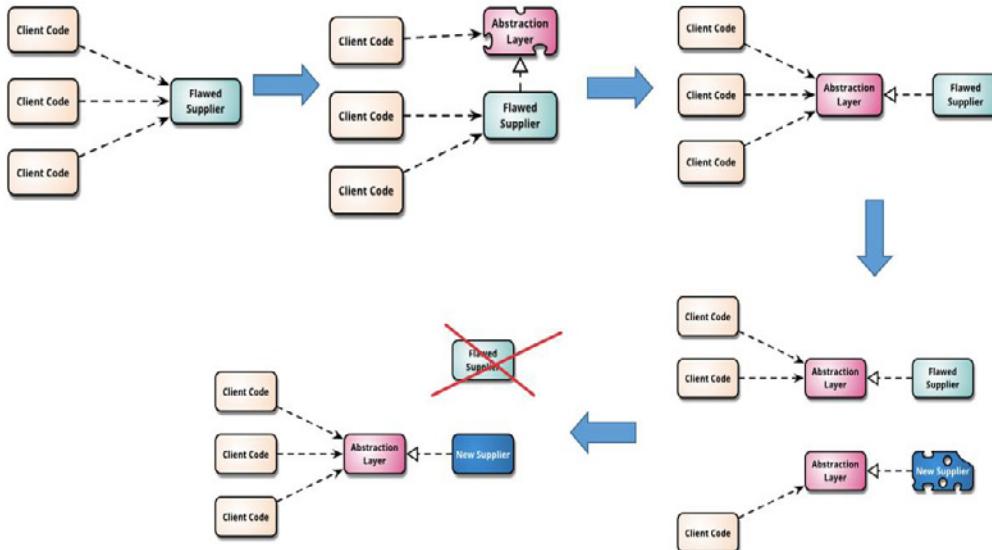
就如修房或修路一样，将老旧待修缮的部分进行隔离，用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同功能。

我们过去所做的拆分中多为修缮者模式，其基本原理来自 Martine Fowler 的 branch by abstraction 的重构方法，如上图所示。

就如我们团队所总结的 16 字重构箴言，我觉得十分的贴切：“**旧的不变，新的创建，一步切换，旧的再见**”。

通过识别内部的被拆模块，对其增加接口层，将旧的引用改为新接口调用；随后将接口封装为 API，并将对接口的引用改为本地 API 调用；最后将新服务部署为新进程，调用改为真正的服务 API 调用。

同时，拆分建议从业务相对独立、耦合度最小的地方开始。待团队获



取相应经验和基础设施平台构建完善后，再进行核心应用迁移和大规模的改造。另外，核心通用服务尽量先行，如身份认证服务。

3. 拆分步骤

对于模块的拆分包括两部分数据库与业务代码，可以先数据库后业务代码，亦可先业务代码后数据库。然我们的项目拆分中遇到的最大挑战是数据层的拆分。在 2015 年的拆分中发现，数据库层由于当时系统性能调优的驱动，在代码中出现了跨模块的数据库连表查询。这导致后期服务的拆分非常的困难。因此在拆分步骤上我们更多的推荐数据库先行。

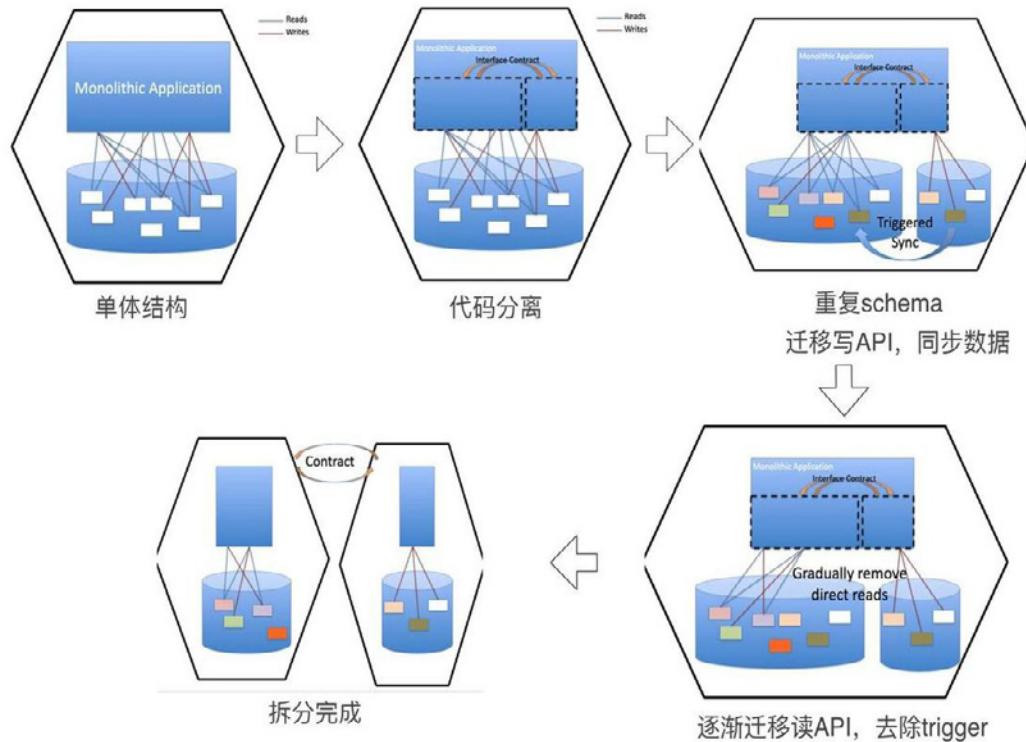
4. 数据库拆分

我们借鉴了重构数据库一书中提到的方法，通过重复 schema 同步数据，对数据库的读写操作分别进行迁移。如下图所示。

虽然技术上是可行的，然这仍然消耗了大量不必要的工作，包括大量的数据迁移。这也是导致当时的拆分无法在给定时间内完成的很大因素。

5. 我们的结果

系统架构图。



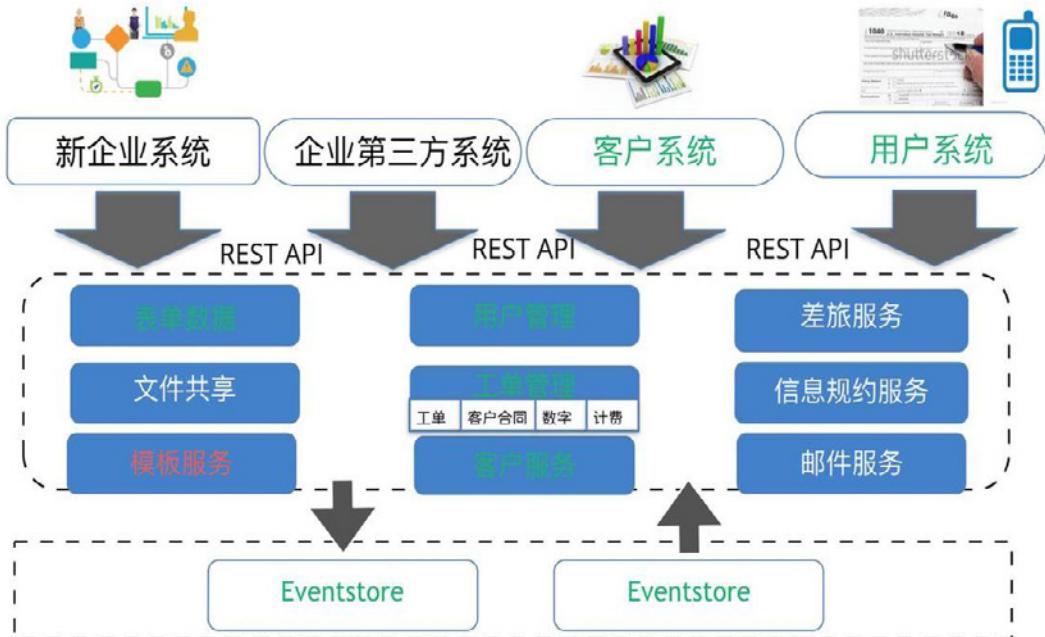
问题 2：拆分后业务变了增加了怎么办？

随着客户业务的变化，我们的服务也在持续的增加，而其中碰到了一个特大的服务，MonkeyAPI。服务的大小如何衡量呢？该服务生产代码 7 万行 +，测试代码 14 万行 +，测试运行时间 2 个小时。团队中 7 个 stream 每天 50% 工作需要对这个服务进行更改，使得团队间的依赖非常严重，独立功能无法单独快速前行，交付速度及质量都受到了影响。

我们的总结

客户的业务是在变化的，我们对业务的认知也是逐渐的过程，所以 Martin Fowler 在他的文章中提出，系统的初期建议以单体结构开始，随业务发展决定其是否被拆分或合并。那么这也意味着这样构建的服务在它的生命周期中必然会持续被拆分或合并。那么为了实现这样一个目标，使系统拥有快速的响应力，也要求这样的拆分必然是高效的低成本的。

因此，服务的设计需要满足如下的原则。



- 服务要有明确的业务边界，以单体开始并不意味着没有边界。
- 服务要有边界，即使以单体开始也要定义单体时期的边界。我们系统中有一个名为MonkeyAPI的服务，Monkey是企业系统的名字，是在中国虎年启动的，由此它并不是一个业务概念。当这个服务的名字为MonkeyAPI时，可以想象5年来它变成了什么？几乎所有和这个产品相关的功能都放入了这个服务中。脱离平台来看这一个产品的系统，其实它只是做了前后端分离而已。这个例子告诉我们，没有边界就会导致大杂烩，之后对其进行整理和重造的代价很大，可能需要花费几代人的努力。
- 服务要有明确清晰的契约设计，即对外提供的业务能力。
- 服务内部要保持高度模块化，才能够容易的被拆分。
- 可测试。

问题 3：如何安全地持续地拆？

就如前言中提到的，系统已经上线大量的用户正在使用，如何在不影

响当下系统运行状态的前提下，持续安全地演进？其实持续演进就是一场架构层次的重构，在这样的路上同样需要：

- 坏味道驱动，架构的坏味道是代码坏味道在更高层次的展现，也就意味着架构的混乱程度同样反映了该系统代码层的质量问题。
- 安全小步的重构。
- 有足够的测试进行保护 – 契约测试。
- 持续验证演进的方向。
- 真正有挑战的问题4：如何保证拆对了？

拆分不能没有目标，尤其在具有风险的架构层次拆分更需谨慎。那么我们如何验证拆分的结果和收益？或许它可以提高开发效率，交付速度快，上线快，宕机时间也短，还能提高开发质量，可扩展性好，稳定，维护成本低，新人成长快，团队容易掌握等等。然而软件开发是一个复杂的事情，拆分可以引起多个维度的变化，度量的难度在于如何准确定位由拆分这一单一因素引起的价值的变化（增加或降低）。

其实要回答这个问题，还是要回到拆分之初：为什么而拆？

我所见过的案例中有因为政治原因拆的，，业务发展需要的，系统集成驱动的等等；有因之而成功的，也有因之而失败的。拆并不是一件容易的事，有诸多的因素。我认为不管表象是什么，拆之前需要弄清拆分的价值所在，这也是我们可以保证拆分结果的源头。

总结

系统可由单体结构开始，不断的演进。而团队需要对业务保持敏感，与客户、业务人员进行业务对话，不断修炼领域驱动设计和重构的能力。

在拆分的路上，我们的经验显示其最大的障碍来自意大利面一样的系

统。不管我们是什么样的架构风格，高内聚低耦合的模块化代码内部质量仍然是我们架构演进的基石。具有夯实领域驱动设计和重构功底的团队才可以应对这些挑战，持续演进，保持其生命力。而架构变迁之前需要弄清背后的变迁动因与价值，探索性前进，及时反馈验证，才是正解。那么我们如何保证架构不被破坏呢？这个问题会在后续的文章中持续探讨。

最后，勿忘初心，且行且演进。



智 能 时 代 的 大 前 端

GMTC 2017

GLOBAL MOBILE
TECH CONFERENCE

全 球 移 动 技 术 大 会

2017.6.9-10 北京·国际会议中心

五大专题全新出炉^{new}

Android新技术与实践

iOS新技术与实践

前端技术实践

大前端技术探索

AI应用实践

3月26日前购票低至**2160**元/张 (全价3600元/张) 团购更加优惠

票务咨询: 18618231445 / alfred@infoq.com

> 扫码了解更多



左耳朵耗子：我对 GitLab 误删除数据库事件的几点思考

作者 陈皓

太平洋时间 2017 年 1 月 31 日晚上，GitLab 通过推特发文承认 300GB 生产环境数据因为 UNIX SA 的误操作，已经被彻底删除（后发文补充说明已经挽回部分数据），引起业界一片哗然。知名博主陈皓在其博客中详细回顾了此次事件，并做了思考和总结，聊聊架构经原作者授权发布此文。

昨天，GitLab.com 发生了一个大事，某同学误删了数据库，这个事看似是个低级错误，不过，因为 GitLab 把整个过程的细节都全部暴露出来了，所以，可以看到很多东西，而对于类似这样的事情，我自己以前也干过，而在最近的两公司中我也见过（Amazon 中见过一次，阿里中见过至少四次），正好通过这个事来说一下自己的一些感想和观点吧。我先放个观点：你觉得有备份系统就不会丢数据了吗？

事件回顾

整个事件的回顾 GitLab.com 在第一时间就放到了 Google Doc 上，事后，又发了一篇 Blog 来说明这个事，在这里，我简单的回顾一下这个事

件的过程。

首先，一个叫 YP 的同学在给 GitLab 的线上数据库做一些负载均衡的工作，在做这个工作时的时候突发了一个情况，GitLab 被 DDoS 攻击，数据库的使用飙高，在 block 完攻击者的 IP 后，发现有个 staging 的数据库 (db2. staging) 已经落后生产库 4GB 的数据，于是 YP 同学在 Fix 这个 staging 库的同步问题的时候，发现 db2. staging 有各种问题都和主库无法同步，在这个时候，YP 同学已经工作的很晚了，在尝试过多个方法后，发现 db2. staging 都 hang 在那里，无法同步，于是他想把 db2. staging 的数据库删除了，这样全新启动一个新的复制，结果呢，删除数据库的命令错误的敲在了生产环境上 (db1. cluster)，结果导致整个生产数据库被误删除（陈皓注：这个失败基本上就是“工作时间过长” + “在多数终端窗口中切换中迷失掉了”）。

在恢复的过程中，他们发现只有 db1. staging 的数据库可以用于恢复，而其它的 5 种备份机制都不可用，第一个是数据库的同步，没有同步 webhook，第二个是对硬盘的快照，没有对数据库做，第三个是用 pg_dump 的备份，发现版本不对（用 9.2 的版本去 dump 9.6 的数据）导致没有 dump 出数据，第四个 S3 的备份，完全没有备份上，第五个是相关的备份流程是问题百出的，只有几个粗糙的人肉的脚本和糟糕的文档，也就是说，不但是人肉的，而且还是完全不可执行的（陈皓注：就算是这些备份机制都 work，其实也有问题，因为这些备份大多数基本上都是 24 小时干一次，所以，要从这些备份恢复也一定是要丢数据的了，只有第一个数据库同步才会实时一些）。

最终，GitLab 从 db1. staging 上把 6 个小时前的数据 copy 回来，结果发现速度非常的慢，备份结点只有 60Mbits/S，拷了很长时间（陈皓注：

为什么不把 db1. staging 给直接变成生产机？因为那台机器的性能很差）。数据现在的恢复了，不过，因为恢复的数据是 6 小时前的，所以，有如下数据丢失掉了：

- 粗略估计，有4613 的项目， 74 forks， 和 350 imports 丢失了；但是，因为Git仓库还在，所以，可以从Git仓库反向推导数据库中的数据，但是，项目中的issues等就完全丢失了。
- 大约有±4979 提交记录丢失了（陈皓注：估计也可以用git仓库中反向恢复）。
- 可能有 707 用户丢失了，这个数据来自Kibana的日志。
- 在1月31日17:20 后的Webhooks 丢失了。

因为 GitLab 把整个事件的细节公开了出来，所以，也得到了很多外部的帮助，2nd Quadrant 的 CTO - Simon Riggs 在他的 blog 上也发布文章 Dataloss at GitLab 给了一些非常不错的建议：

- 关于PostgreSQL 9.6的数据同步hang住的问题，可能有一些Bug，正在fix中。
- PostgreSQL有4GB的同步滞后是正常的，这不是什么问题。
- 正常的停止从结点，会让主结点自动释放WALSender的链接数，所以，不应该重新配置主结点的 max_wal_senders 参数。但是，停止从结点时，主结点的复数连接数不会很快的被释放，而新启动的从结点又会消耗更多的链接数。他认为，GitLab配置的32个链接数太高了，通常来说，2到4个就足够了。
- 另外，之前GitLab配置的max_connections=8000太高了，现在降到2000个是合理的。
- pg_basebackup 会先在主结点上建一个checkpoint，然后再开始

同步，这个过程大约需要4分钟。

- 手动的删除数据库目录是非常危险的操作，这个事应该交给程序来做。推荐使用刚release 的 repmgr。
- 恢复备份也是非常重要的，所以，也应该用相应的程序来做。推荐使用 barman （其支持S3）。
- 测试备份和恢复是一个很重要的过程。

看这个样子，估计也有一定的原因是——GitLab 的同学对 PostgreSQL 不是很熟悉。

随后，GitLab 在其网站上也开了一系列的 issues，其 issues 列表在这里 Write post-mortem（这个列表可能还会在不断更新中）。

- infrastructure#1094 - Update PS1 across all hosts to more clearly differentiate between hosts and environments
- infrastructure#1095 - Prometheus monitoring for backups
- infrastructure#1096 - Set PostgreSQL's max_connections to a sane value
- infrastructure#1097 - Investigate Point in time recovery & continuous archiving for PostgreSQL
- infrastructure#1098 - Hourly LVM snapshots of the production databases
- infrastructure#1099 - Azure disk snapshots of production databases
- infrastructure#1100 - Move staging to the ARM environment
- infrastructure#1101 - Recover production replica(s)

- infrastructure#1102 - Automated testing of recovering PostgreSQL database backups
- infrastructure#1103 - Improve PostgreSQL replication documentation/runbooks
- infrastructure#1104 - Kick out SSH users inactive for N minutes
- infrastructure#1105 - Investigate pgbarman for creating PostgreSQL backups

从上面的这个列表中，我们可以看到一些改进措施了。挺好的，不过我觉得还不是很够。

因为类似这样的事，我以前也干过（误删除过数据库，在多个终端窗口中迷失掉了自己所操作的机器……），而且我在亚马逊里也见过一次，在阿里内至少见过四次以上（在阿里人肉运维的误操作的事故是我见过最多的），但是我无法在这里公开分享，私下可以分享。在这里，我只想从非技术和技术两个方面分享一下我的经验和认识。

我的思考：技术方面 人肉运维

一直以来，我都觉得直接到生产线上敲命令是一种非常不好的习惯。我认为，一个公司的运维能力的强弱和你上线上环境敲命令是有关的，你越是喜欢上线敲命令你的运维能力就越弱，越是通过自动化来处理问题，你的运维能力就越强。理由如下：

1. 如果说对代码的改动都是一次发布的话，那么，对生产环境的任何改动（包括硬件、操作系统、网络、软件配置……），也都算是一次发布。那么这样的发布就应该走发布系统和发布流程，要

被很好的测试、上线和回滚计划。关键是，走发布过程是可以被记录、追踪和回溯的，而在线上敲命令是完全无法追踪的。没人知道你敲了什么命令。

2. 真正良性的运维能力是一人管代码，代码管机器，而不是人管机器。你敲了什么命令没人知道，但是你写个工具做变更线上系统，这个工具干了什么事，看看工具的源码就知道了。

另外，有人说，以后不要用 rm 了，要用 mv，还有人说，以后干这样的事时，一个人干，另一个人在旁边看，还有人说，要有一个 checklist 的强制流程做线上的变更，还有人说要增加一个权限系统。我觉得，这些虽然可以 work，但是依然不好，因为：

如果要解决一个事情需要加更多的人来做的事，那这事就做成劳动密集型了。今天我们的科技就是在努力消除人力成本，而不是在增加人力成本。而做为一个技术人员，解决问题的最好方式是努力使用技术手段，而不是使用更多的人肉手段。人类区别于动物的差别就是会发明和使用现代化的工具，而不是使用更多的人力。另外，这不仅仅因为是，人都是会有这样或那样的问题（疲惫、情绪化、急躁、冲动……），而机器是单一无脑不知疲惫的，更是因为，机器干活的效率和速度是比人肉高出 N 多倍的。

增加一个权限系统或是别的一个 watch dog 的系统完全是在开倒车，权限系统中的权限谁来维护和审批？不仅仅是因为多出来的系统需要多出来的维护，关键是这个事就没有把问题解决在 root 上。除了为社会解决就业问题，别无好处，故障依然会发生，有权限的人一样会误操作。对于 GitLab 这个问题，正如 2nd Quadrant 的 CTO 建议的那样，你需要的是一个自动化的备份和恢复的工具，而不是一个权限系统。

像使用 mv 而不 rm，搞一个 checklist 和一个更重的流程，更糟糕。

这里的逻辑很简单，因为，1) 这些规则需要人去学习和记忆，本质上来说，你本来就不相信人，所以你搞出了一些规则和流程，而这些规则和流程的执行，又依赖于人，换汤不换药，2) 另外，写在纸面上的东西都是不可执行的，可以执行的就是只有程序，所以，为什么不把 checklist 和流程写成代码呢（你可能会说程序也会犯错，是的，程序的错误是 consistent，而人的错误是 inconsistent）？

最关键的是，数据丢失有各种各样的情况，不单单只是人员的误操作，比如，掉电、磁盘损坏、中病毒等等，在这些情况下，你设计的那些想流程、规则、人肉检查、权限系统、checklist 等等统统都不管用了，这个时候，你觉得应该怎么做呢？是的，你会发现，你不得不用更好的技术去设计出一个高可用的系统！别无它法。

关于备份

一个系统是需要做数据备份的，但是，你会发现，GitLab 这个事中，就算所有的备份都可用，也不可避免地会有数据的丢失，或是也会有很多问题。理由如下：

1. 备份通常来说都是周期性的，所以，如果你的数据丢失了，从你最近的备份恢复数据里，从备份时间到故障时间的数据都丢失了。
2. 备份的数据会有版本不兼容的问题。比如，在你上次备份数据到故障期间，你对数据的 scheme 做了一次改动，或是你对数据做了一些调整，那么，你备份的数据就会和你线上的程序出现不兼容的情况。
3. 有一些公司或是银行有灾备的数据中心，但是灾备的数据中心没

有一天 live 过。等真正灾难来临需要 live 的时候，你就会发现，各种问题让你 live 不起来。你可以读一读几年前的这篇报道好好感受一下《以史为鉴，宁夏银行 7 月系统瘫痪最新解析》。

所以，在灾难来临的时候，你会发现你所设计精良的“备份系统”或是“灾备系统”就算是平时可以工作，但也会导致数据丢失，而且可能长期不用的备份系统很难恢复（比如应用、工具、数据的版本不兼容等问题）。

我之前写过一篇《分布式系统的事务处理》，你还记得下面这张图吗？看看 Data Loss 那一行的，在 Backups, Master/Slave 和 Master/ Master 的架构下，都是会丢的。

	Backups	M/S	MM	2PC	Paxos
Consistency	Weak	Eventual		Strong	
Transactions	No	Full	Local	Full	
Latency	Low		High		
Throughput	High		Low Medium		
Data loss	Lots	Some		None	
Failover	Down	Read only	Read/write		

所以说，如果你要让你的备份系统随时都可以用，那么你就要让它随时都 Live 着，而随时都 Live 着的多结点系统，基本上就是一个分布式的高可用的系统。因为，数据丢失的原因有很多种，比如掉电、磁盘损坏、中病毒等等，而那些流程、规则、人肉检查、权限系统、checklist 等等都只是让人不要误操作，都不管用，这个时候，你不得不用更好的技术去设计出一个高可用的系统！别无它法。（重要的事，得再说一篇）

另外，你可以参看我的另一篇《关于高可用系统》，这篇文章中以 MySQL 为例，数据库的 replication 也只能达到两个 9。

AWS 的 S3 的的高可用是 4 个加 11 个 9 的持久性（所谓 11 个 9 的持久性 durability，AWS 是这样定义的，如果你存了 1 万个对象，那么丢一个的时间是 1000 万年），这意味着，不仅仅只是硬盘坏，机器掉电，整个机房挂了，其保证可以承受有两个设施的数据丢失，数据还是可用的。试想，如果你把数据的可用性通过技术做到了这个份上，那么，你还怕被人误删一个结点上的数据吗？

我的思考：非技术方面 故障反思

一般说来，故障都需要反思，在 Amazon，S2 以上的故障都需要写 COE (Correction of Errors)，其中一节就是需要 Ask 5 Whys，我发现 GitLab 的故障回顾的 blog 中第一段中也有说要在今天写个 Ask 5 Whys。关于 Ask 5 Whys，其实并不是亚马逊的玩法，这还是算一个业内常用的玩法，也就是说不断的为自己为为什么，直到找到问题的根本原因，这会逼着所有的当事人去学习和深究很多东西。在 Wikipedia 上有相关的词条 5 Whys，其中罗列了 14 条规则：

- 你需要找到正确的团队来完成这个故障反思。
- 使用纸或白板而不是电脑。
- 写下整个问题的过程，确保每个人都能看懂。
- 区别原因和症状。
- 特别注意因果关系。
- 说明Root Cause以及相关的证据。
- 5个为什么的答案需要是精确的。
- 寻找问题根源的频，而不是直接跳到结论。
- 要基础客观的事实、数据和知识。

- 评估过程而不是人。
- 千万不要把“人为失误”或是“工作不注意”当成问题的根源。
- 培养信任和真诚的气氛和文化。
- 不断的问“为什么”直到问题的根源被找到。这样可以保证同一个坑不会掉进去两次。
- 当你给出“为什么”的答案时，你应该从用户的角度来回答。

工程师文化

上述的这些观点，其实，我在我的以往的博客中都讲过很多遍了，你可以参看《什么是工程师文化？》以及《开发团队的效率》。其实，说白了就是这么一个事——如果你是一个技术公司，你就会更多的相信技术而不是管理。相信技术会用技术来解决问题，相信管理，那就只会有制度、流程和价值观来解决问题。

这个道理很简单，数据丢失有各种各样的情况，不单单只是人员的操作，比如，掉电、磁盘损坏、中病毒等等，在这些情况下，你设计的那些流程、规则、人肉检查、权限系统、checklist 等等统统都不管用，这个时候，你觉得应该怎么做呢？是的，你会发现，你不得不用更好的技术去设计出一个高可用的系统！别无它法（重要的事得说三遍）。

事件公开

很多公司基本上都是这样的套路，首先是极力掩盖，如果掩盖不了了就开始撒谎，撒不了谎了，就“文过饰非”、“避重就轻”、“转移视线”。然而，面对危机的最佳方法就是——“多一些真诚，少一些套路”，所谓的“多一些真诚”的最佳实践就是——“透明公开所有的信息”，GitLab 此次的这个事给大家树立了非常好的榜样。AWS 也会把自己所有的故障和

细节都批露出来。

事情本来就做错了，而公开所有的细节，会让大众少很多猜测的空间，有利于抵制流言和黑公关，同时，还会赢得大众的理解和支持。看看 GitLab 这次还去 YouTube 上直播整个修复过程，是件很了不起的事，大家可以到他们的 blog 上看看，对于这样的透明和公开，一片好评。



鸡年到！程序员们996了一整年，终于能回家好好过个年，家人团聚、推杯换盏自是美哉。但怕就怕那其乐融融时的话锋一转。你走过最长的路，不是跟产品经理斗智斗勇的路，而是三姑六婆们的各种套路啊！

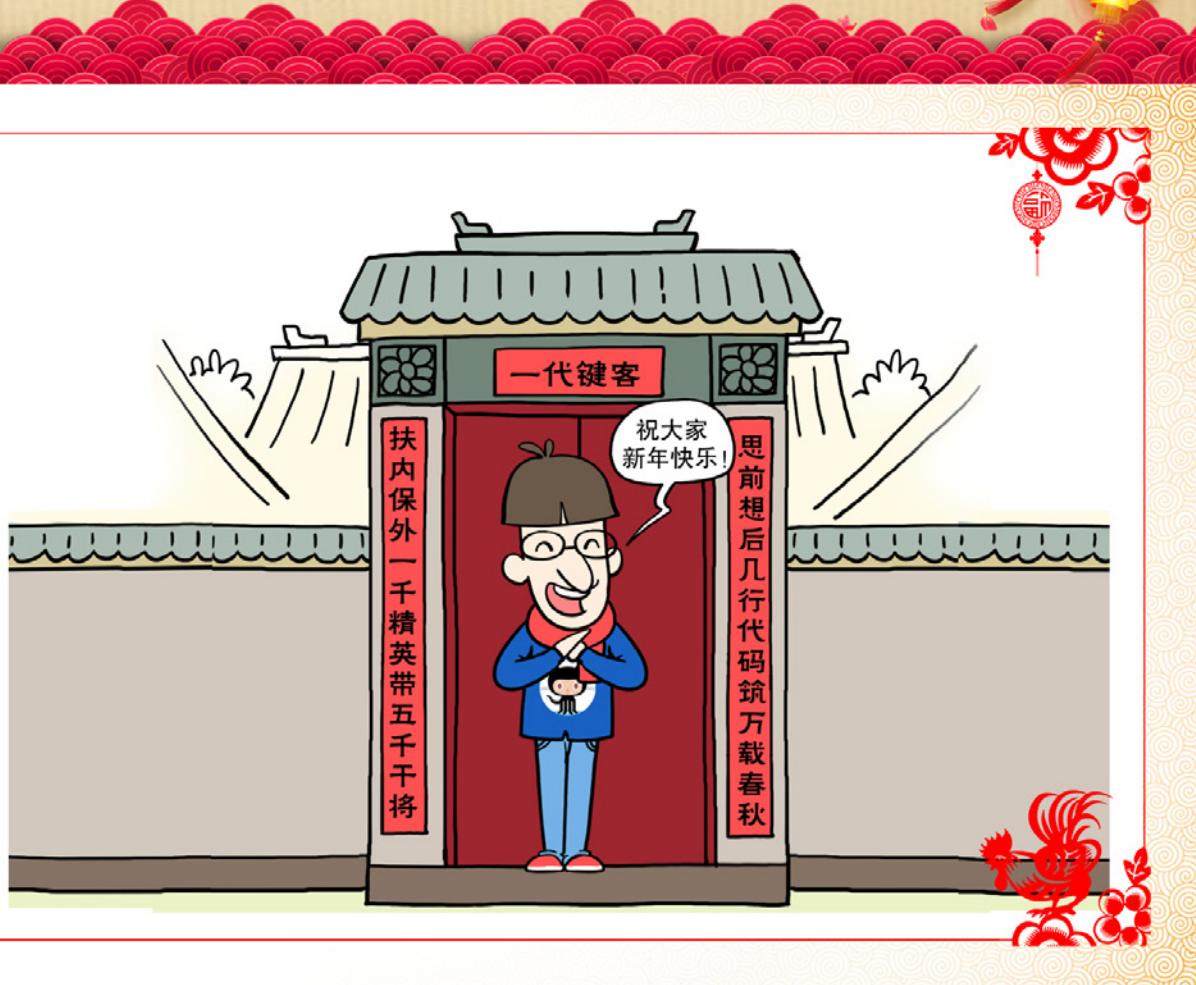
策划：Gary Betty 小智
脚本：小智
漫画：王小威





鸡年到！程序员们996了一整年，终于能回家好好过个年，家人团聚、推杯换盏自是美哉。但怕就怕那其乐融融时的话锋一转。你走过最长的路，不是跟产品经理斗智斗勇的路，而是三姑六婆们的各种套路啊！

策划：Gary Betty 小智
脚本：小智
漫画：王小威





架构师 月刊 2017年1月

本期主要内容：Facebook 开源跨平台前端布局引擎 Yoga；腾讯大数据宣布开源第三代高性能计算平台 Angel；支持十亿维度；前端组件化开发方案及其在 React Native 中的运用；深入浅出 Redis client/server 交互流程；微信 PaxosStore；深入浅出 Paxos 算法协议；我们为什么选择 Vue.js 而不是 React



解读2016

许多年后，如果我们回过头来评点，也许 2016 年是非常重要的一个时间节点。



顶尖技术团队访谈录 第七季

本次的《中国顶尖技术团队访谈录》·第七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



架构师特刊 机器学习实践

值机器学习，趁着近年来深度学习的热潮强势复苏，很快地从一个很少被大众关注的技术主题，转变为被很多人使用的管理工具和开发工具。