

架构师 ARCHITECT



理论派 | Theory

每个架构师都应该研究下康威定律

观点 | Opinion

电商网站的初期技术选型

推荐文章 | Article

架构设计原则和模式

问答系统的前世今生

热点 | Hot

2016年会成为Java EE微服务年吗

.NET开源现状



电商网站的初期技术选型

完全从0到1建设一个电商网站，技术选型和注意事项有哪些？

2016年会成为Java EE微服务年吗

去年12月，来自C2B2的Steve Millidge预测，2016年将会成为Java EE微服务年。

架构师 ARCHITECT



理论派 | Theory
每个架构师都应该研究下康威定律
观点 | Opinion
电商网站的初期技术选型
推荐文章 | Article
架构设计原则与模式
问答系统的前世今生
热点 | Hot
2016年会成为Java EE微服务年吗
.NET开源现状

2016年3月

InfoQ

www.infoq.cn

InfoQ

针对架构设计的几个痛点，我总结出的架构原则和模式

作者介绍了架构设计的原则以及什么是架构，并分析了4种常用的软件架构模式，分别是分层架构、事件驱动架构、微内核架构和微服务架构。

Netflix Spinnaker：实现全局部署

Netflix最近将他们的持续交付平台Spinnaker作为开源项目进行了发布。

.NET开源现状

部分开源贡献者最近对于.NET开源的现状提出了一些顾虑，他们围绕着个人与企业对于项目的贡献展开了讨论。

每个架构师都应该研究下康威定律

今天的分享主要来自我之前的工作经验以及平时的学习总结和思考。

架构师 2016年3月刊

本期主编 郭蕾

流程编辑 丁晓昀

发行人 霍泰稳

联系我们

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

2016.3.27

北京 · 新云南皇冠假日酒店



技术社群大会

TECHNICAL COMMUNITIES CONFERENCE (TCC2016)

— 源于社区，服务社区，共建社区 —



扫描二维码报名

或登录 shequn.geekbang.org 报名

2016年04月21-23日 北京·国际会议中心

www.qconbeijing.com

限时
折扣 9 折

仅限3月27日前

团购可享更多优惠!

精彩早知道

中外一线技术专家
加入QCon大会



Paul Butcher

《七周七并发模型》作者

《编程语言演变
对开发者的意义》



Leo Li

LinkedIn Sr Manager
Business Analytics and Data Science

《创造数据产品驱动商业价值》



庄振运

LinkedIn Staff Software Engineer

《OS造成的长时间非典型
JVMGC停顿：深度分析和解决》



覃超

峰瑞资本技术合伙人
前 Facebook 工程师

《Facebook的项目开发流程
和工程师的绩效管理机制》



Paul Kinlan

Google Staff
Developer Associate

《移动Web的未来》



Dylan Schiemann

SitePen CEO
Dojo Toolkit 联合创始人

《移动Web的未来》



林文

高盛技术部VP

《高盛如何使用Java》



李文哲

普惠金融首席数据科学家

《大数据和人工智能在互联网
金融上的应用》



欧阳辰

小米研发架构师

《后SOA主义，微服务
架构演化之道》

精彩内容策划中，欢迎自荐/推荐讲师
线索提供: speakers@cn.infoq.com
更多精彩专题内容
敬请关注: www.qconbeijing.com
抢票热线: 010-64738142
咨询邮箱: qcon@cn.infoq.com



扫描关注QCon官微

主办方 **Geekbang**. **InfoQ**
极客邦科技

卷首语 | 架构发展趋势和现状

作者 王庆友

最近几年，软件系统越来越大，越来越复杂，相应地，架构扮演的角色也越来越重要。无架构，不系统，因此这里和大家交流下架构的发展趋势，简单概括有几点：

技术平台轻量化

十年前，企业级应用大行其道，商业应用服务器如 Weblogic/Websphere 扮演中心的角色，它们内含各种系统级组件，如 web 容器、EJB 容器、数据源和事务管理等，一站式搞定企业级软件所需的性能、可扩展性、高可用性，整个系统平台是很重的 all in one 模式，走的是 scale out 路线。

开源社区的兴起为这些系统级组件提供免费实现，并且更轻量级和更易用，比如 Tomcat、MySQL、Struts、Spring，IBatis 等。同时应用更加互联网化，互联网业务的快速变化，要求开发模式和系统实现更轻量更敏捷，所以技

术平台一般是 DIY，如目前比较流行的 LAMP 和 SSH 组合，系统设备也从高大上的 IOE 变成平民化的 X86+Mysql+Linux)，走的是 scale up 路线。

软件设计服务化

传统的企业级应用是单体应用 (monolith application)，一般是分层结构，如表现层 / 应用层 / 领域层 / 数据层，这主要是水平切分的思想。

随着互联网应用的发展，特别是大型电商系统，业务非常复杂。这种巨型系统，首先要关注的是如何根据业务划分子系统，然后是子系统间如何协作，最后才是子系统内部实现。SOA 设计可以很有效支持前面两步，在 SOA 体系里，每个子系统是独立的服务，服务接口体现子系统协作关系，至于子系统内部，直接作为黑盒子处理。

所以对于复杂系统，首先采用 SOA 垂直切分子系统，然后使用分层设计水平切分单个子系统，服务化把传统的分层设计往前更推进一步。

当然 SOA 本身也在不断发展，最初跨企业的 Web service 交互可认为 1.0 时代；支持企业内部系统间轻量级访问，支持服务治理，可认为 2.0 时代；服务进一步分层和微服务化可认为 3.0 时代。

应用系统生态化

软件是人类活动的虚拟化模拟，目前这种模拟越来越深入，逐渐覆盖吃穿住行、旅游、购物、娱乐、社交等方方面面，应用也从开始单一的

信息管理系统，到覆盖整个企业业务，成为企业级应用，到了互联网时代，应用更超出企业边界，通过开放平台互相链接成一个巨大的生态系统。

业务的上下游关联要求应用系统内外一体化，内部系统调用、APP 接口、开放平台接口尽可能地一致和复用。例如对于 APP 接口和 Open API，只需要提供简单处理，如安全和数据格式转换，核心实现转发到内部 SOA 服务作统一处理。

业务在变，技术在变，架构也在变。变的是形式，不变的是本质，架构为了系统更有序，系统为了业务更快速，业务为了生活更美好。

2016年会成为Java EE微服务年吗



作者 Mark Little 译者 谢丽

进入 2016 年时间还不是很长，让我们回顾下去年年底的一个预言。去年 12 月，来自 C2B2 的 Steve Millidge [预测](#)，2016 年将会成为 Java EE [微服务](#) 年。在一定程度上，这是基于 Steve 在 JavaOne 上的[演讲](#)，他在演讲中详细地讨论了这个主题。此外，Steve 还是 [Payara](#) 的联合创始人，Payara 的目标用户也是对微服务感兴趣的 Java EE 开发人员。Steve 还认为，SOA 和微服务之间的差别很小，这种观点我们以前听说并且[报道过](#)。他在视频中指出：

“微服务与 SOA 没什么不同。它还是关于 SOA”。

当然，现在还存在争论，因为他的背景和当前的工作重心，Steve 可能会发现自己很难保

持客观的态度。不过，早在 2014 年，微服务还处于起步阶段，Adam Bien 就描述了[理想的 Java EE 微服务](#)。

理想的 Java EE 微服务是一个单 [实体控制边界] 组件，在一个 WAR 包中，部署在单台服务器 / 域中。在这种情况下，开发人员可以单独地发布和重新部署单个组件（又称微服务）。WAR 包之间不可能直接调用方法，因此，WAR 包将不得不使用比如 JAX-RS 来彼此通信。

我们在去年年底就微服务、DevOps 和 Java EE 相关内容[采访](#)了 Markus Eisele，他详细论述了自己为什么[认为](#) Java EE 将会在微服务生态圈的发展中扮演重要的角色。还有一些其他使用 Java EE 编写微服务的方法，包括 [TomEE](#) 和

[WildFly](#)。[KumuluzEE](#)是JavaOne 2015 Duke 选择奖的其中一个获奖者，该框架是一个 Java EE 微服务框架。该框架的联合创建者 Matjaz Juric 解释说：

KumuluzEE 是第一个使用标准 Java API 的微服务框架。微服务架构的重点是将应用程序开发成服务并将这些服务单独部署；没有一个框架提供自动化部署和配置，是不可能使用 Java EE 实现真正的微服务架构的。

让我们看一些人们如何看待微服务和 Java EE 的其他例子，这会非常有趣，因为有些人严格来讲并不属于传统的 Java EE 领域。例如，早在 2014 年，Alex Soto 就论述了为什么 Java EE 和 RxJava 是一个很棒的方案。不过，并不是每个人都认可使用 Java EE 能使开发人员采用微服务。正如 Rick Hightower 所说的那样：

如果你将一个 WAR 文件部署到一个 Java EE 容器，那么你很可能不是在做微服务开发。如果你在一个容器或 EAR 文件中包含超过一个 WAR 文件，那么你肯定不是在做微服务开发。如果你将服务部署为 AMI 或 Docker 容器，而且你的微服务有一个 main 方法，那么你可能是在编写微服务。

而且，Rick 也不认为微服务与 SOA 相同：

事实上，它们在许多方面是完全相反的。例如，SOA 往往采用 WSDL，后者是一种非常严格的、强类型的服务端点定义方式。WSDL 和 XML 模式中所有的未知量都来自 XML。

当然，我们已经多次讨论过，SOA 和 Web Service 常常没有关系。不过，Rick 及其他一

些人确信，Java EE 太过臃肿或者说笨拙，以其为基础构建微服务并不合适。Jeppe Cramon 认为，Java EE 之所以是一个糟糕的基础还有更为根本的原因：

如果我们将两路（同步）通信与小 / 微服务结合使用，并根据比如“1 个类=1 个服务”的原则，那么我们实际上回到了使用 Corba、J2EE 和分布式对象的 20 世纪 90 年代。遗憾的是，新生代的开发人员没有使用分布式对象的经验，因此也就没有认识到这个主意多么糟糕，他们正试图重复历史，只是这次使用了新技术，比如用 HTTP 取代了 RMI 或 IIOP。

如果微服务和 SOA 密切相关，那么可能会有一种观点，就是微服务可以像 SOA 那样采用一种 [技术无关](#) 的方式。你认为呢？2016 年会成为 Java EE 微服务年吗？如果有的话，Java EE 会在微服务中扮演什么角色？

Netflix Spinnaker：实现全局部署



作者 Matt Raible 译者 邵思华

Netflix 最近将他们的持续交付平台 [Spinnaker](#) 作为[开源项目](#)进行了发布。Spinnaker 允许使用者通过创建管道（pipeline）的方式展现一个交付流程，并执行这些管道以完成一次部署。

Spinnaker 能够向前兼容 [Asgard](#)，因此无需一次性完全迁移至 Spinnaker。

用户可在 Spinnaker 中从创建一个部署单元（例如一个 JAR 文件或是 Docker 镜像）开始，直至将应用部署至云环境中。Spinnaker 支持多种云平台，包括 AWS、Google Could Platform 以及 Cloud Foundry。Spinnaker 通常是在一个持续集成作业完成之后启动的，但也可以通过一个 cron 作业、一个 Git 库或者由其他管道进行手动触发。

Spinnaker 还为用户提供了管理服务器集群的功能，通过应用视图，用户可以对新的服务器组、负载均衡器以及安全组进行编辑、规模调整、删除、禁用以及部署等操作。

Spinnaker 是由基于 JVM 的服务（由 Spring Boot 和 Groovy 所实现），以及由 AngularJS 所创建的 UI 所组成的。

为了进一步了解 Spinnaker 及其开源现状，InfoQ 与来自 Netflix 的 Spinnaker 团队进行了一次访谈，受访者包括负责交付工程的经理 Andy Glover，以及高级软件工程师 Cameron Fieber 和 Chris Berry。

InfoQ: Spinnaker 发布已经有一个多月了，社区对此的反响如何？

Glover: 社区对 Spinnaker 的接纳程度令人震惊！这个平台内置了对多个云提供商的兼容，并且能够通过一种可扩展的模型接入其中。这意味着我们打造了一个大型社区，而不是一系列专注于不同分支的微型社区。这种方式的优势在于社区中的每个人都可以利用各种创新的特性。我们已看到许多来自于新社区成员的 pull request，并且我相信，随着我们继续提升项目的可适配性，将会看到越来越多的贡献。

InfoQ: 许多云提供商似乎都建议使用者上传单一的部署文件，并通过他们的 API 或 UI 进行扩展。Spinnaker 的不同之处又体现在哪里呢？

Fieber: Spinnaker 推荐使用不可变基础设施风格的部署方式，它提供了对各种云提供商（AWS AMI、Google Compute Engine Images 等等）的镜像格式的原生支持。

Spinnaker 还支持通过 Quick Patch 进行已排编代码的 push，让团队能够快速地迭代，在现有的实例中进行软件包的推送以及安装，从而减免了新虚拟机上线的等待时间。常见的使用方式是快速地部署一个测试环境以运行测试，或发布一些有状态服务，例如数据存储的补丁。

InfoQ: 你知道是否有用户已经开始使用 Spinnaker 对多个云环境进行部署吗？

Glover: 我知道有一家非常著名的公司已经在多个云提供商环境中进行部署了，不过他们希望我不要提起他们的名字。我觉得应该有其他用户也会这样做，并且随着社区的发展，我们将进一步了解有哪些公司将采取多个云环境的策略。

InfoQ: 你怎样比较 Spinnaker 与 Heroku 的管道特性？

Glover: 我认为 Spinnaker 与 [Heroku](#) 的管道相比最大的区别在于：(1) Spinnaker 支

The screenshot shows the Spinnaker UI with the 'Clusters' tab selected. On the left, there are filters for 'ACCOUNT', 'REGION', 'STACK', 'STATUS', and 'AVAILABILITY ZONES'. The main area displays two regions: 'US-EAST-1' and 'US-WEST-2'. Each region has a grid of green squares representing instances. A specific instance in the US-EAST-1 grid is highlighted with a black border and labeled 'i-abcdedfgh'. To the right of the grid, there is a detailed view for this instance, showing its launch date (2016-02-17), server group ('api-prod-v218'), and various status metrics like 'Type: c3.8xlarge' and 'Status: Starting'. There are also sections for 'Discovery', 'DNS', 'Security Groups', and 'Logs'.

持多种可适配的部署端点，例如 AWS、GCE、Pivotal CloudFoundry 等等。（2）Spinnaker 的管道模型非常之灵活，它支持多种不同类型的阶段（stage），而且社区也可以自行开发各种管道并将其接入 Spinnaker 平台。Heroku 管道的设计目标是为了 Heroku 本身服务的，并且他们的管道模型非常僵化。另一方面，Heroku 的管道是通过命令行驱动的。我们目前还没有发布 Spinnaker 的命令行客户端。

InfoQ：从 Spinnaker 在 GitHub 上的项目来看，“gate”这个项目似乎是由 Groovy 编写的，并且使用了 Spring Boot。为什么你们选用了 Groovy 而不是 Java 8 呢？

Fieber：Spinnaker 其实就是 Asgard 项目的后继者（我们还有一个名为 Mimir 的内部工具。译注：Asgard 与 Mimir 都来源于北欧神话），他们都是由 Grails 编写的应用。我们团队对于 Groovy 有充分的了解，感觉它比 Grails 更为轻量级，并且更专注于操作性，因此值得投入精力进行研究。Spring

Boot 是一种很自然的选择，并且 Groovy 很适合应用在这个环境中。由于选择了 Groovy，我们就能够从 Asgard 中选取经过了充分测试的 AWS 代码并在 Spinnaker 中重用。

InfoQ：Spinnaker 的 UI 项目“deck”是由 AngularJS（1.4 版本）编写的，你们的开发过程是否顺利？

Berry：刚开始的时候是比较顺利的。当我们在 18 个月之前启动这个项目的时候，Angular 表现得十分稳定。并且有大量的库（UI Bootstrap、UI Router 和 Restangular 等等）让 UI 能够十分快速地进行创建与迭代。React 也是一门非常激动人心的技术，但当时它才刚刚出现不久，而且它的规范与模式还没有 Angular 那么充实。

但随后这个开发过程逐渐变得令人痛苦起来。其中部分原因在于 Netflix 的规模很大，我们某些应用需要在一个屏幕中渲染上千种元素，而 Angular 1.x 在处理这种数量的 DOM 节点时

性能跟不上。对于这些页面，我们选择以纯 JS 进行重写，再用一些比较粗糙的方式进行性能对比。最终发现纯 JS 的结果能够满足性能的要求，即便一次性渲染几千个实例也没问题。但这种方式写出来的代码非常脆弱，毕竟 Angular 已经为你完成各种任务铺平了道路。

另一个难题在于如何让 UI 实现模块化与可适配性，让不同的云提供商能够按照他们的需求创建 UI 模块，并且让外部用户能够创建自定义的管道组件。我们在这两方面的工作做得还可以，它不算很差，但也绝对谈不上出色。我们从 UI Router 中直接抄用了大量的代码与概念，让我们的代码能够运行起来，但除了我们团队之外，我并没有看到像 Google、微软和 Pivotal 尝试开发任何自定义的实现。我想一定有某些人已经在做这件事了，只是我们还没看到罢了。

以上这些并不是说我们对于选择 Angular 1.x 感到后悔。在当时来说，它对于我们确实是正确的选择。现在回过头来看，如果我们能够回到 18 个月之前，那我们或许会对代码进行一些重写，但大概还是会用 Angular 吧。

InfoQ：你们是否计划将 UI 项目迁移至 Angular 2？

Berry： 我们确实有进行迁移的打算，但估计要到 5 至 6 个月之后才会开始。毕竟 Angular 2 还只发布了 beta 版本，并且在工具方面也缺乏支持。那些编写 UI 特性的非 Netflix 用户有许多都不是专职的前端开发者，我们希望确保他们能够轻易地找到构建特性的正确方式，并且在遇到问题时能够方便地进行调试。

我很乐于看到 Angular 2 在明年的发展，并且想多了解一些从 1.x 迁移至 2 的案例。我们只是想对此采取一种相对谨慎的态度，并且从其他人身上多学习一点经验。

InfoQ：Spinnaker 是怎样改善 Netflix 的部署工作的？

Glover： 首先，也是最重要的一点是它为所有人提供了一个标准的交付平台。Spinnaker 让用户能够方便地进行交付，并且对于流程具备充分的信心，这正是团队最需要的东西。通过这个平台，整个 Netflix 服务能够更频繁地进行部署，并且在运维上具备更大的弹性。Spinnaker 本身与来自 Netflix 的大量其他服务与工具进行了集成，使这些特性更易于为用户所用。举例来说，我们有一个名为 ACA (Automated Canary Analysis —— 自动化金丝雀分析) 的内部服务，这是由 Netflix 的另一个团队所维护的。尽管如此，它也是一个原生的 Spinnaker 管道阶段，能够提供测试服务。在 Spinnaker 出现之前，如果有哪个团队需要使用 ACA，就不得不自行寻找将 ACA 集成进自己的管道的方式。如今随着 Spinnaker 的出现，就为 ACA 的使用定义了一种标准方法，这也最终使 ACA 的使用得到了突飞猛进式的增长，这也提高了我们在 AWS 上的生产环境的可靠性。如果新创建的工具与服务能够提供更好的测试、数据采集或运维的弹性，就可以将它们接入 Spinnaker 平台，让每个人都能够充分利用这些工具与服务。

InfoQ：你对 Spinnaker 的哪个特性最中意？

Glover： Spinnaker 支持一种表达式语言，能够让用户对管道进行参数化。它允许用户创建一些非常复杂的管道，最重要的是还能够进行重用。它们能够在全球范围内进行构建的提送 (promote)、测试与部署。

InfoQ：你对于 Spinnaker 还有什么想补充的吗？

Glover： 虽然 Spinnaker 是由 Netflix 所开发的，但是这个项目的成功离不开与 Google、

微软、Pivotal 和 Kenzan 良好的合作与他们的贡献。我们目前的良好发展状况以及将来的发展前景让我们非常振奋。我们目前正在开发的内容包括对容器更深层的支持、整体可适配性与灵活性的增长、以及 UI 和 UX 的改进。而 Spinnaker 社区的发展也让我们觉得非常激动。

Greg Turnquist 是来自 Pivotal 的高级软件工程师，他在一篇[博客文章](#)中描述了 Spinnaker 如何与 Cloud Foundry 进行结合工作。我们很有兴趣了解其他人是如何整合使用 Spinnaker 的。

InfoQ：你在什么时机下会建议 Cloud Foundry 用户尝试使用 Spinnaker 进行部署工作？

Turnquist：对于 Cloud Foundry 的支持是在 Spinnaker 的 master 分支中开发的，其中包括大量的特性。我们目前正在计划通过活跃的客户进行 beta 级别的测试。在我看来，这对于 Cloud Foundry 的用户，无论是 PCF、PWS 还是其他 CF 的认证实例都已经成熟了。

如果你觉得目前手动将新的版本发布到 CF 的时间太长，而希望转而使用管道进行部署、冒烟测试与验证，那么现在正是使用 Spinnaker，剔除你的发布流程中低效部分的时机。

InfoQ：Spinnaker 能否简便地与 Cloud Foundry 进行整合？

Turnquist：我觉得“简便”这种表述或许不够准确，这个词似乎暗示着整合这两个平台只需很少的工作。实际上我花了很多时间去学习 Spinnaker 的底层概念，并将这些概念与 Cloud Foundry 的概念进行一一对应。随着经验的积累，我开始认识到 Cloud Foundry 能够完美地与 Spinnaker 平台进行配合。我需要学习大量 CF 的 API 方面的知识（实际上我是在

Spring 团队工作，而不是在 CF 团队中工作），但我学到的东西越多，这两者的结合就做得越好。

Cloud Foundry 与 Spinnaker 两者都支持将应用的多个版本进行分组以进行统一的升级或回滚、在新版本与旧版本之间实现负载均衡，并且支持开发实例、预发布实例与生产环境的实例。它展现了 Spinnaker 架构的长处与灵活性，并且也展现了 Cloud Foundry 这个平台强大的能力。

InfoQ：Greg，你对于 Spinnaker 的哪个特性最中意？

Turnquist：当我谈到这个平台的时候，给我最多惊喜的是 UI 的管道编辑器，它让我能够进行各种随意的变更。在“Cloud Foundry After Dark”这个 webcast 中，我设计了一个简单的管道，其中只包含一个步骤：部署至生产环境。在我进行描述的同时，主持人 Andrew 要求我进行一些调整，让它能够实现部署至预发布环境、进行冒烟测试以及部署至生产环境。每当他话音刚落，我就已经完成了调整。随后我们开始运行管道并通过一个对用户十分友好的界面阅读它的输出。这个平台让用户能够随意塑造流程，这是我们不应低估的一个强大特性。

InfoQ 再次感谢 Spinnaker 团队与 Greg Turnquist 能够回答我们的这些问题。在 GitHub 上可以找到 Spinnaker 的源代码。如果读者想要与 Spinnaker 社区进行交流，可以加入它的 Slack 频道、[查看 Stack Overflow](#) 上有关 Spinnaker 的问题，或关注它的 Twitter 账号 [@spinnakerio](#)。

.NET开源现状



作者 Pierre-Luc Maheu 译者 邵思华

部分开源贡献者最近对于.NET开源的现状提出了一些顾虑，他们围绕着个人与企业对于项目的贡献展开了讨论。而微软在.NET生态环境中所扮演的角色也成为辩论的焦点。

Itamar Syn-Hershko 目前是 Lucene.NET 项目的贡献者，[他表示](#).NET 生态系统的传统发展方式对于开发者只知利用开源软件，却不知回报的思想负有一定的责任：

微软曾是一家产品公司，因此它的生存依赖于产品的销售。操作系统、文字处理器、开发工具、数据库，这些产品都是收费的，而且往往价格不菲。在利用微软产品栈开展工作或进行开发

时，免费的工具始终遭到人们的忽视。

而这种状况会让人产生一种危险的心态，即免费即意味着可能无法胜任有一定难度的工作。虽然我可能会对这些工具表示感谢，但这种东西的出现就应该为我所用，并且就应该是免费的。人们也不会产生回报或是成立社区项目的想法，免费就意味着不用自掏腰包，而分辨“免费啤酒”与“免费讲座”有什么区别也是毫无意义的。反正它就是免费的，管它呢。

在 Twitter 上的回应大多数是关于项目资助的想法。Jimmy Boggard[写道](#)：

库与框架的需求有着巨大的差别，必须有人来资助框架的发展。

Christos Matskas 也[写道](#):

许多公司在开源软件的贡献方面设定了一些愚蠢的知识产权限制，这一点必须得到改变。

而在 Reddit 上，[Manitcor](#) 也表达了对于开源项目缺乏企业资助的不满：

我所知的一些使用 .NET 技术的公司对于开源软件并不支持，他们认为那些开发者的行为是离经叛道的，并且不会为他们提供任何支持。因此，虽然对开源软件的支持是一个值得骄傲的目标，但我看不会有太多人愿意为此丢了自己饭碗。

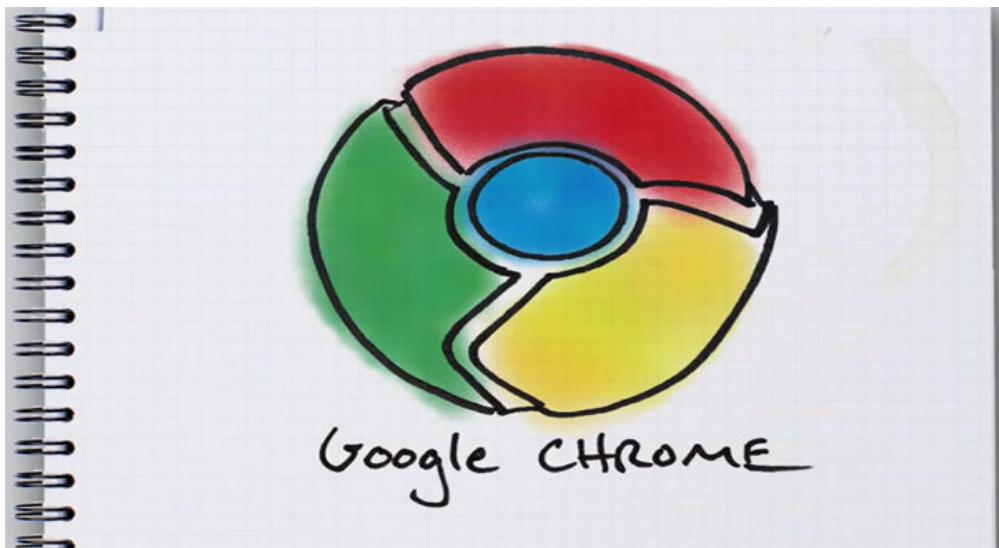
虽然部分用户将 .NET 的开源现状一定程度上归咎于微软的作法，但也有部分人认为微软决定对 .NET 框架与产品开源的做法将起到很大的正面作用。JustMake[写道](#):

微软对于 .NET 的开源开了一个好头，它的影响不应被人们低估。不久之前，我提交了一个关于 VS 2015 的问题，一位开发者随后给我发送了一封邮件，告诉我他已经修复了这个问题。他在邮件中甚至还将相应的 GitHub issue 的链接告诉了我。这与微软之前的做法已经产生了很大的变化，在过去，如果你要与开发者进行交谈，不仅要提交一个问题报告，还必须接受为此付款的可能。这样的变化将改变那些认为代码不能共享的人们的看法。

Sean Killeen 同样认为目前的状况正在逐步改善中，他[表示](#) .NET 的开源正在加速发展：

我看到身边有越来越多的开发者对于 .NET 生态系统的开源表现出兴奋之情，并且他们已经认识到回馈的重要性。我觉得他们现在已经卷起了袖子和裤管，一到时机成熟就准备大干一场，或者更深入地参与其中。我认为新一代的 .NET 开发者正在成熟起来，或者说经历了浴火重生。现在，这些开源项目背后有大量热情的人们提供支持。而这段时间以来，微软本身的参与程度也有很大的提升。我希望我们不要只看到像 Itamar 等人的劳动成果，还要主动帮助他们完善这些项目。

Oracle宣布：2017年将废弃Java浏览器插件



作者 Charles Humble 译者 夏雪

Oracle [宣布](#)，预计在 2017 年发布的 JDK 9 中将废弃 Java 浏览器插件。这个废弃的技术将从未来 Java 版本的 Oracle Java 开发工具 (JDK) 和 Java 运行期环境 (JRE) 中彻底移除，但是 Oracle 还未表明将是哪个版本。

供应商建议仍需要 Java 客户端的组织可以将 Java Web Start 作为 applet 的替代品，并为帮助他们迁移发布了一份[白皮书](#)。

许多最终用户肯定不会怀念这个插件的。它不但要为大量的安全性问题负责，而且自从 Sun Microsystems 启动这一做法之后就有大量的“捆绑”软件内置到安装包中。在不同时期谷歌工具栏、微软 MSN 工具栏、McAfee 安全、雅虎工具栏以及臭名昭著的 Ask toolbar 全都和 Java 捆绑到一起了。

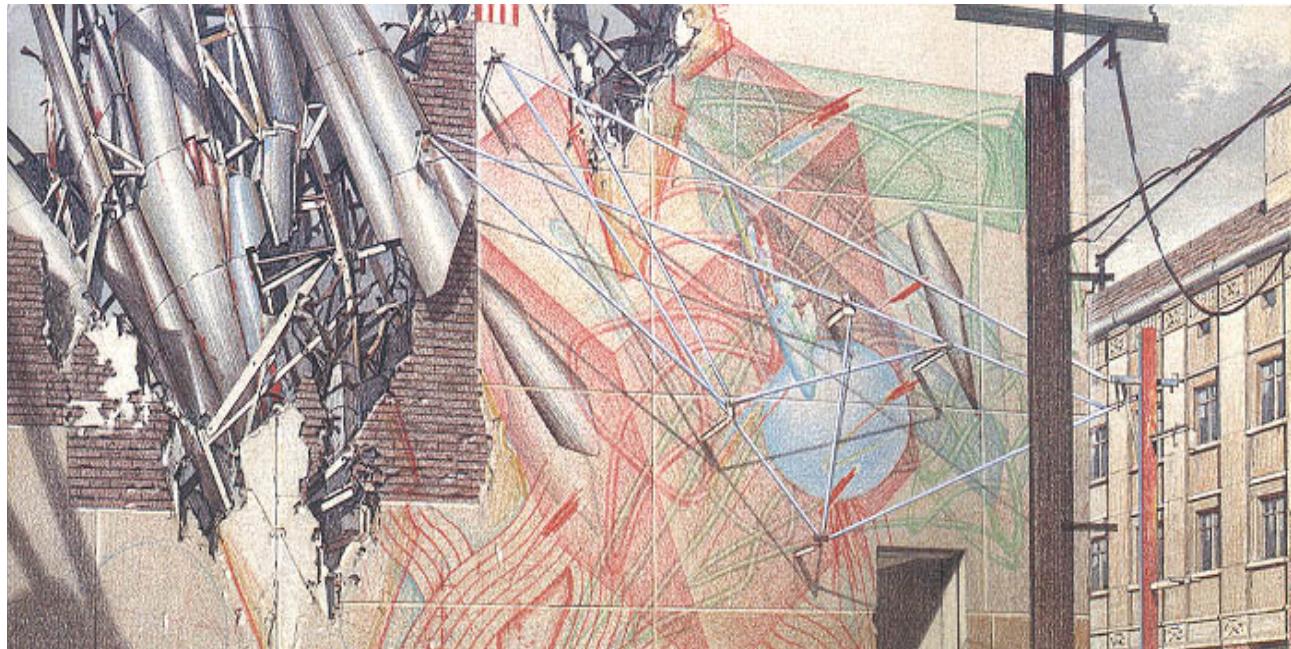
Ask 尤其难以避免安装，因为用户不得不记得每次安装和更新时都选择不进行安装，这些更

新普遍都存在安全性问题。有一封要求 Oracle 与它解除捆绑的请愿书已经超过了 21000 个签名，其中包括像 [Joshua Bloch](#) 这样的业内知名人物，可见它是多么不受欢迎。Oracle 在 2014 年针对这些问题进行了处理，增加了一个禁止第三方绑定软件的配置选项。

Java 和 Flash 是众所周知的可以广为利用的攻击向量，而附着物使这一点更加显著了，所以浏览器厂商已经在警惕带有[附着物](#)的插件了。谷歌在去年四月份已经开始废弃浏览器插件了，Mozilla 在七月份也宣布了类似的计划。微软最新的[Edge 浏览器也不支持插件](#)了。

Oracle 此举大受欢迎，但直到 Adobe 宣布 Flash 生命周期结束之前插件的安全问题仍将存在。

针对架构设计的几个痛点，我总结出的架构原则和模式



作者 Firat 译者 韩陆

本文作者介绍了架构设计的原则以及什么是架构，并分析了4种常用的软件架构模式，分别是分层架构、事件驱动架构、微内核架构和微服务架构。

分层架构

分层架构是最常见的架构，也被称为n层架构。多年以来，许多企业和公司都在他们的项目中使用这种架构，它已经几乎成为事实标准，因此被大多数架构师、开发者和软件设计者所熟知。

分层架构中的层次和组件是水平方向的分层，每层扮演应用程序中特定的角色。根据需求和软件复杂度，我们可以设计N层，但大多数应用程序使用3-4层。有太多层的设计会很糟糕，

将导致复杂度的上升，因为我们必须维护每一层。在传统的分层架构中，分层包括表现层、业务或者服务层，以及数据访问层。表现层负责应用程序的用户交互和用户体验（外观和视觉）。通常我们会使用数据传输对象（Data Transfer Object）将数据带到这一层，然后使用视图模型（View Model）渲染到客户端。业务层接收请求并执行业务规则。数据访问层负责操作各种类型的数据库，每个访问数据库的请求都要经过这一层。

分层无需知道其他层如何去做，比如业务层无需知道数据访问层是如何查询数据库的，相反，业务层在调用数据层的特定方法时，只需关注需要部分数据还是全部数据。这就是我们所说的关注点分离。这是非常强大的功能，每层负

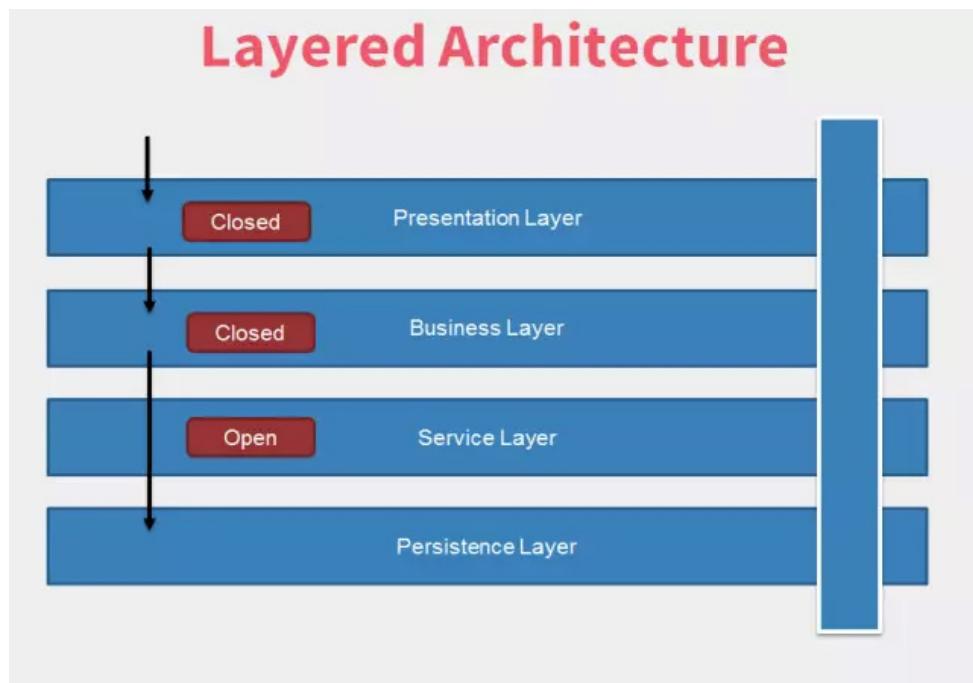


图 1

责其所负的责任。

根据开闭原则来简化实现。

分层架构中的核心概念是管理依赖。如果我们使用依赖倒置原则和测试驱动开发 (Test Driven Development)，我们的架构会有更好的健壮性。因为，我们要保证所有可能的用例都有测试用例。

我们需要这样的冗余，即使业务层没有处理业务规则，也要通过业务层来调用数据层，这叫分层隔离。对于某些功能，如果我们从表现层直接访问数据层，那么数据层后续的任何变动都将影响到业务层和表现层。（见图 1）

分层架构中的一个重要的概念就是分层的开闭原则。如果某层是关闭的，那么每个请求都要经过着一层。相反，如果该层是开放的，那么请求可以绕过这一层，直接到下一层。

分层隔离有利于降低整个应用程序的复杂度。某些功能并不需要经过每一层，这时我们需要

分层架构是 SOLID 原则的通用架构，当我们不确定哪种架构更合适的时候，分层架构将是一个很好的起点。我们需要注意防止架构陷入污水池反模式。这种反模式描述了请求经过分层，但没做任何事或者只处理了很少的事。如果我们的请求经过所有分层而没有做任何事，这就是污水池反模式的征兆。如果 20% 的请求只是经过各层，而 80% 的请求实际做事，这还好，如果这个比率不是这样的，那么我们已经患上反模式综合征。（见图 2）

此外，分层架构可以演变为巨石应用 (Monolith)，导致代码库难以维护。

分层架构分析：

- 敏捷性：总体敏捷性是指对不断变化的环境作出反应的能力。由于其整体风格 (Monolith) 的性质，可能会变得难以应



图 2

对通过所有层的变化，开发者需要注意依赖性和分层分离。

- 易于部署：大型应用程序的部署会是个麻烦。一个小要求，可能需要部署整个应用程序。如果能做好持续交付，可能会有所帮助。
- 可测试性：使用 Mocking 和 Faking，每一层可以独立测试，因此测试上很容易。
- 性能：虽然分层应用程序可能表现良好，但是因为请求需要经过多个分层，可能会存在性能问题。
- 可伸缩性：因为耦合太紧以及整体风格（Monolith）的天生特质，很难对分层应用程序进行伸缩。然而，如果分层能够被构建为独立的部署，还是可以具备伸缩能力的。但是，这样做的代价可能很昂贵。
- 易于开发：这种模式特别易于开发。许多企业采用这种模式。大多数开发者也都知道、了解，并且可以轻松学习如何使用它。

事件驱动架构

事件驱动架构（Event Driven Architecture）是一种流行的分布式异步架构模式，用于创建可伸缩的应用程序。这种模式是自适应的，可

用于小规模或者大规模的应用程序。事件驱动架构可以与调停者拓扑（Mediator Topology）或者代理者拓扑（Broker Topology）一起使用。理解拓扑的差异，为应用程序选择正确的拓扑是必不可少的。

调停者拓扑

调停者拓扑需要编排多种事件。比如在交易系统中，每个请求流程必须经过特定的步骤，如验证、订单、配送，以及通知买家等。在这些步骤中，有些可以手动完成，有些可以并行完成。

通常，架构主要包含 4 种组件，事件队列（Event Queue）、调停者（Mediator）、事件通道（Event Channel）和事件处理器（Event Processor）。客户端创建事件，并将其发送到事件队列，调停者接收事件并将其传递给事件通道。事件通道将事件传递给事件处理器，事件最终由事件处理器处理完成。（见图 3）

事件调停者不会处理也不知道任何业务逻辑，它只编排事件。事件调停者知道每种事件类型的必要步骤。业务逻辑或者处理发生在事件处

Event Driven Architecture

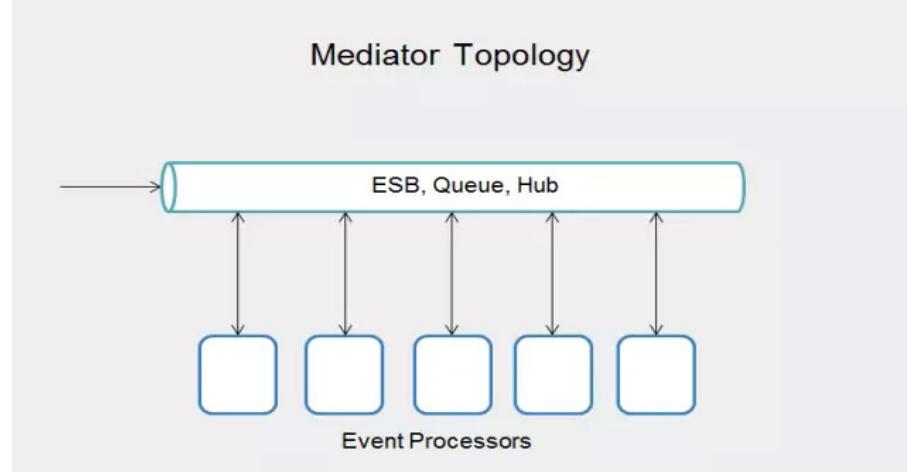


图3

Event Driven Architecture

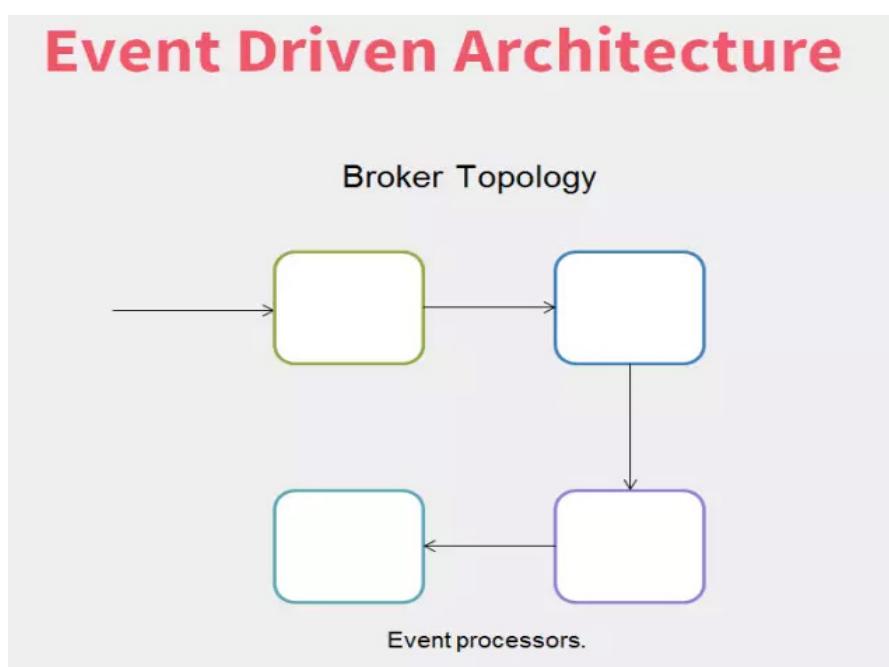


图4

理器中，事件通道、消息队列或者消息主题用于传递事件给事件处理器。事件处理器是自包含和独立的，解耦于架构。理想情况下，每种事件处理器应只负责处理一种事件类型。

通常，企业服务总线、队列或者集线器可以用作事件调停者。正确选择技术和实现能够降低风险。

代理者拓扑

不像调停者拓扑，代理者拓扑不使用任何集中的编排，而是在事件处理器之间使用简单的队列或者集线器，事件处理器知道处理事件的下一个事件处理器。（见图 4）

因其分布式和异步的性质，事件驱动架构的实

现相对复杂。我们需要面对很多问题，比如网络分区、调停者失败、重新连接逻辑等。由于这是一个分布式且异步的模式，如果你需要事务，那就麻烦了，你得需要一个事务协调器。分布式系统中的事务非常难以管理，很难找到标准的工作单位模式。

另一个充满挑战的概念是契约。架构师声称服务的契约应该预先定义，而应变是非常昂贵的。

事件驱动架构分析：

敏捷性：由于事件和事件处理器之间解耦，并且可独立维护，因此这种模式的敏捷性很高。变化可以快速、轻松地完成，而不会影响整个系统。

- **易于部署：**由于架构是解耦的，因此很容易部署。组件可以独立部署，并且可以在调停者上注册。部署在代理者拓扑上也相当简单。
- **可测试性：**虽然独立测试组件很容易，但

测试整个应用程序很有挑战。因此端到端的测试是很难的。

- **性能：**事件驱动架构性能非常好，因为它是异步的。此外，事件通道和事件处理器可以并行工作，因为它们是解耦的。
- **可伸缩性：**事件驱动架构的伸缩性非常好，因为组件之间解耦，组件可以独立扩展。
- **易于开发：**这种架构的开发不是很容易。需要明确定义契约，错误处理和重试机制得处理得当。

微内核架构

微内核架构 (Microkernel architecture) 模式也被称为插件架构 (plugin architecture) 模式。这是产品型应用程序的理想模式，由两部分组成：核心系统和插件模块。核心系统通常包含最小的业务逻辑，并确保能够加载、卸载和运行应用所需的插件。许多操作系统使用这种模式，因此得名微内核。

插件彼此独立，因此解偶。核心系统持有注册器，插件将自己注册其上，因此核心系统知道

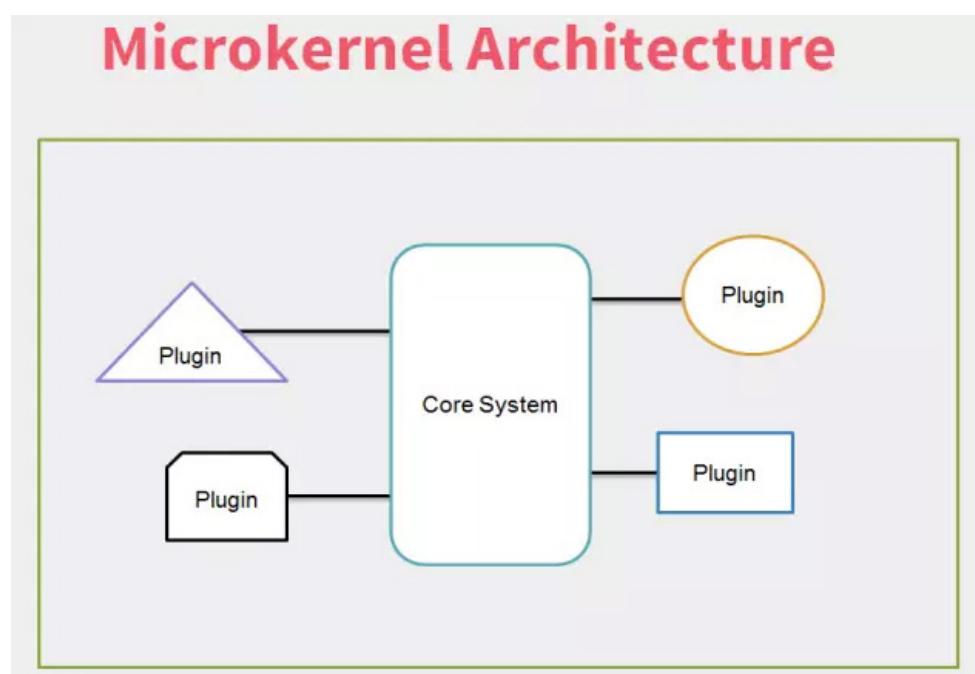


图 5

哪里可以找到它们以及如何运行它们。(见图 5)

这种模式非常适合桌面应用程序，但是也可以在 Web 应用程序中使用。事实上，许多不同的架构模式可以作为整个系统的一个插件。对于产品型应用程序来说，如果我们想将新特性和功能及时加入系统，微内核架构是一种不错的选择。

微内核架构分析：

- 敏捷性：由于插件可以独立开发并注册到核心系统，微内核架构具有很高的敏捷性。
- 易于部署：依赖于核心系统的实现，能做到不需要重新启动整个系统来完成部署。
- 可测试性：如果插件开发是独立的，测试就可以独立且隔离地进行。还可以 Mock 核心系统来测试插件。
- 性能：这取决于我们有多少插件在运行，但性能可以调优。
- 可伸缩性：如果整个系统被部署为单个单元，这个系统将难以扩展。
- 易于开发：这种架构不容易开发。实现核心系统和注册会很困难，而且插件契约和数据交换模型增加了难度。

微服务架构

尽管微服务的概念还相当新，但它确实已经快速地吸引了大量的眼球，以替代整体应用和面向服务架构（SOA）。其中的一个核心概念是具备高可伸缩性、易于部署和交付的独立部署单元（Separately Deployable Units）。最重要的概念是包含业务逻辑和处理流程的服务组件（Service Component）。拿捏粒度设计服务组件是必要而具有挑战性的工作。服务组件是解耦的、分布式的、彼此独立的，并且可以使用已知协议来访问。

微服务的发展是因为整体应用和面向服务应用的缺陷。整体应用程序通常包含紧耦合的层，难以部署和交付。比如，如果应用程序总在每次应对变化时垮掉，这是一个因耦合而产生的大问题。微服务将应用程序分解为多个部署单元，因此很容易提升开发和部署能力，以及可测性。虽然面向服务架构非常强大，具有异构连接和松耦合的特性，但是性价比不高。它很复杂、昂贵，难于理解和实现，通常对于大多数应用程序来说矫枉过正。微服务简化了这种复杂性。（见图 6）

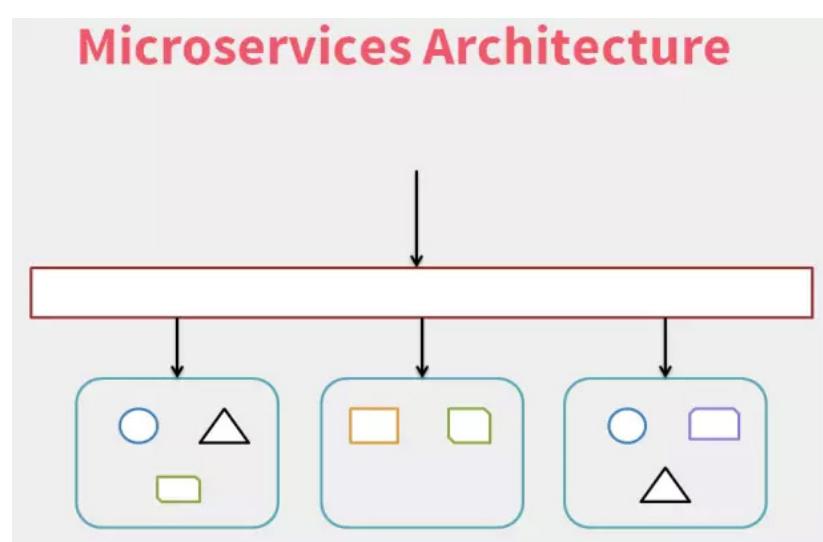


图 6

跨服务组件的代码冗余是完全正常的。开发微服务时，为了受益于独立的部署单元，以及更加容易的部署，我们可以违反 DRY 原则。其中的挑战来自服务组件之间的契约，以及服务组件的可用性。

微服务架构分析：

- 敏捷性：由于服务组件可以各自独立开发，彼此没有耦合，因此微服务架构具有很高的敏捷性。独立部署单元能够对变化作出

迅速的反应。

- 易于部署：相比其他的架构模式，微服务的优势是服务组件即是单独部署单元。
- 可测试性：服务组件的测试可以独自完成。微服务的可测试性很高。
- 性能：依赖于服务组件和这种特定模式的分布式性质。
- 可伸缩性：独立部署单元天然具备很好的伸缩性。
- 易于开发：每个服务组件可以各自独立实现。（见图 7）

		Comparison			
		Layered	Event Driven	Microkernel	MicroService
	Agility	:frowny:	:smiley:	:smiley:	:smiley:
	Deployment	:frowny:	:smiley:	:smiley:	:smiley:
	Testability	:smiley:	:frowny:	:smiley:	:frowny:
	Performance	:frowny:	:smiley:	:smiley:	:smiley:
	Scalability	:frowny:	:smiley:	:frowny:	:smiley:
	Development	:smiley:	:frowny:	:smiley:	:frowny:

图 7

问答系统的前世今生



作者 李维

一、前世

传统的问答系统是人工智能（AI: Artificial Intelligence）领域的一个应用，通常局限于一个非常狭窄专门的领域，基本上是由人工编制的知识库加上一个自然语言接口而成。由于领域狭窄，词汇总量很有限，其语言和语用的歧义问题可以得到有效的控制。问题是可预测的，甚至是封闭的集合，合成相应的答案自然有律可循。著名的项目有上个世纪 60 年代研制的 LUNAR 系统，专事回答有关阿波罗登月返回的月球岩石样本的地质分析问题。SHRDLE 是另一个基于人工智能的专家系统，模拟的是机器人在玩具积木世界中的操作，机器人可以回答这个玩具世界的几何状态的问题，并听从语言指令进行合法操作。

这些早期的 AI 探索看上去很精巧，揭示了一个有如科学幻想的童话世界，启发人的想象力和好奇心，但是本质上这些都是局限于实验室的玩具系统（Toy Systems），完全没有实用的可能和产业价值。随着作为领域的人工智能之路越走越窄（部分专家系统虽然达到了实用，基于常识和知识推理的系统则举步维艰），寄生其上的问答系统也基本无疾而终。倒是有一些机器与人的对话交互系统（Chatterbot）一路发展下来至今，成为孩子们的网上玩具（我的女儿就很喜欢上网找机器人对话，有时故意问一些刁钻古怪的问题，程序应答对路的时候，就夸奖它一句，但更多的时候是看着机器人出丑而哈哈大笑。不过，我个人相信这个路子还大有潜力可挖，把语言学与心理学知识交融，应该可以编制出质量不错的机器人心理治疗

师。其实在当今的高节奏高竞争的时代，很多人面对压力需要舒缓，很多时候只是需要一个忠实的倾听者，这样的系统可以帮助满足这个社会需求。要紧的是要消除使用者“对牛弹琴”的先入为主的偏见，或者设法巧妙隐瞒机器人的身份，使得对话可以敞开心扉。扯远了，打住。）

二、重生

产业意义上的开放式问答系统完全是另一条路子，它是随着互联网的发展以及搜索引擎的普及应运而生的。准确地说，开放式问答系统诞生于 1999 年，那一年搜索业界的第八届年会（TREC-8: Text REtrieval Conference）决定增加一个问答系统的竞赛，美国国防部有名的 DARPA 项目资助，由美国国家标准局组织实施，从而催生了这一新兴的问答系统及其 community。问答系统竞赛的广告词写得非常精彩，恰到好处地指出搜索引擎的不足，确立了问答系统在搜索领域的价值定位。记得是这样写的（大体）：**用户有问题，他们需要答案。** 搜索引擎声称自己做的是信息检索（information retrieval），其实检索出来的并不是所求信息，而只是成千上万相关文件的链接（URLs），答案可能在也可能不在这些文件中。无论如何，总是要求人去阅读这些文件，才能寻得答案。问答系统正是要解决这个信息搜索的关键问题。**对于问答系统，输入的是问题，输出的是答案，就是这么简单。**

说到这里，有必要先介绍一下开放式问答系统诞生时候的学界与业界的背景。

从学界看，传统意义上的人工智能已经不再流行，代之而来的是大规模真实语料库基础上的

机器学习和统计研究。语言学意义上的规则系统仍在自然语言领域发挥作用，作为机器学习的补充，而纯粹基于知识和推理的所谓智能规则系统基本被学界抛弃（除了少数学者的执着，譬如 Douglas Lenat 的 Cyc）。学界在开放式问答系统诞生之前还有一个非常重要的发展，就是信息抽取（Information Extraction）专业方向及其 Community 的发展壮大。与传统的自然语言理解（Natural Language Understanding）面对整个语言的海洋，试图分析每个语句求其语义不同，信息抽取是任务制导，任务之外的语义没有抽取的必要和价值：每个任务定义为一个预先设定的所求信息的表格，譬如，会议这个事件的表格需要填写会议主题、时间、地点、参加者等信息，类似于测试学生阅读理解的填空题。这样的任务制导的思路一下子缩短了语言技术与实用的距离，使得研究人员可以集中精力按照任务指向来优化系统，而不是从前那样面面俱到，试图一口吞下语言这个大象。到 1999 年，信息抽取的竞赛及其研讨会已经举行了七届（MUC-7: Message Understanding Conference），也是美国 DARPA 项目的资助产物（如果说 DARPA 引领了美国信息产业研究及其实用化的潮流，一点儿也不过誉），这个领域的任务、方法与局限也比较清晰了。发展得最成熟的信息抽取技术是所谓实体名词的自动标注（Named Entity: NE tagging），包括人名、地名、机构名、时间、百分比等等。其中优秀的系统无论是使用机器学习的方法，还是编制语言规则的方法，其查准率查全率的综合指标都已高达 90% 左右，接近于人工标注的质量。这一先行的年轻领域的技术进步为新一代问答系统的起步和开门红起到了关键的作用。

到 1999 年，从产业来看，搜索引擎随着互联

网的普及而长足发展，根据关键词匹配以及页面链接为基础的搜索算法基本成熟定型，除非有方法学上的革命，关键词检索领域该探索的方方面面已经差不多到头了。由于信息爆炸时代对于搜索技术的期望永无止境，搜索业界对关键词以外的新技术的呼声日高。用户对粗疏的搜索结果越来越不满意，社会需求要求搜索结果的细化 (More Granular Results)，至少要以段落为单位 (Snippet) 代替文章 (URL) 为单位，最好是直接给出答案，不要拖泥带水。虽然直接给出答案需要等待问答系统的研究成果，但是从全文检索细化到段落检索的工作已经在产业界实行，搜索的常规结果正从简单的网页链接进化到 highlight 了搜索关键词的一个个段落。

新式问答系统的研究就在这样一种业界急切呼唤、学界奠定了一定基础的形势下，走上历史舞台。美国标准局的测试要求系统就每一个问题给出最佳的答案，有短答案 (不超过 50 字节) 与长答案 (不超过 250 字节) 两种。下面是第一次问答竞赛的试题样品：

- Who was the first American in space?
- Where is the Taj Mahal?
- In what year did Joe DiMaggio compile his 56-game hitting streak?

三、昙花

这次问答系统竞赛的结果与意义如何呢？应该说是结果良好，意义重大。最好的系统达到 60% 多的正确率，就是说每三个问题，系统可以从语言文档中大海捞针一样搜寻出两个正确答案。作为学界开放式系统的第一次尝试，这是非常令人鼓舞的结果。当时正是 dot com

的鼎盛时期，IT 业界渴望把学界的这一最新研究转移到信息产品中，实现搜索的革命性转变。里面有很多有趣的故事，参见我的相关博文：《朝华午拾：创业之路》。

回顾当年的工作，可以发现是组织者、学界和业界的天时地利促成了问答系统奇迹般的立竿见影的效果。美国标准局在设计问题的时候，强调的是自然语言的问题 (English questions, 见上)，而不是简单的关键词 queries，其结果是这些问句偏长，非常适合做段落检索。为了保证每个问题都有答案，他们议定问题的时候针对语言资料库做了筛选。这样一来，文句与文本必然有相似的语句对应，客观上使得段落匹配 (乃至语句匹配) 命中率高 (其实，只要是海量文本，相似的语句一定会出现)。设想如果只是一两个关键词，寻找相关的可能含有答案的段落和语句就困难许多。当然找到对应的段落或语句，只是大大缩小了寻找答案的范围，不过是问答系统的第一步，要真正锁定答案，还需要进一步细化，pinpoint 到语句中那个作为答案的词或词组。这时候，信息抽取学界已经成熟的实名标注技术正好顶上来。为了力求问答系统竞赛的客观性，组织者有意选择那些答案比较单纯的问题，譬如人名、时间、地点等。这恰好对应了实名标注的对象，使得先行一步的这项技术有了施展身手之地。譬如对于问题 “In what year did Joe DiMaggio compile his 56-game hitting streak?”, 段落语句搜索很容易找到类似下列的文本语句：Joe DiMaggio’s 56 game hitting streak was between May 15, 1941 and July 16, 1941. 实名标注系统也很容易锁定 1941 这个时间单位。An exact answer to the exact question, 答案就这样在海量文档中被搜得，好像大海捞针一般神

奇。沿着这个路子，11 年后的 IBM 花生研究中心成功地研制出打败人脑的电脑问答系统，获得了电视智能大奖赛 Jeopardy! 的冠军（见报道《COMPUTER CRUSHES HUMAN 'JEOPARDY!' CHAMPS》），在全美观众面前大大地出了一次风头，有如当年电脑程序第一次赢得棋赛冠军那样激动人心。

当年成绩较好的问答系统，都不约而同地结合了实名标注与段落搜索的技术：证明了只要有海量文档，Snippet+NE 技术可以自动搜寻回答简单的问题。

四、现状

1999 年的学界在问答系统上初战告捷，我们作为成功者也风光一时，下自成蹊，业界风险投资商蜂拥而至。很快拿到了华尔街千万美元的风险资金，当时的感觉真地好像是在开创工业革命的新纪元。可惜好景不长，互联网泡沫破灭，IT 产业跌入了萧条的深渊，久久不能恢复。投资商急功近利，收紧银根，问答系统也从业界的宠儿变成了弃儿（见《朝华午拾 - 水牛风云》）。主流业界没人看好这项技术，比起传统的关键词索引和搜索，问答系统显得不稳定、太脆弱（Not Robust），也很难 Scale Up，业界的重点从深度转向广度，集中精力增加索引涵盖面，包括所谓 Deep Web。问答系统的研制从业界几乎绝迹，但是这一新兴领域却在学界发芽生根，不断发展壮大，成为自然语言研究的一个重要分支。IBM 后来也解决了 Scale Up（用成百上千机器做分布式并行处理）和适应性培训的问题，为赢得大奖赛做好了技术准备。同时，学界也开始总结问答系统的各种类型。一种常见的分类是根据问题的种类。

我们很多人都在中学语文课上，听老师强调过阅读理解要抓住几个 WH 的重要性：Who/What/When/Where/How/Why (Who did what when, where, how and why?)。抓住了这些 WH，也就抓住了文章的中心内容。作为对人的阅读理解的仿真，设计问答系统也正是为了回答这些 WH 的问题。值得注意的是，这些 WH 问题有难有易，大体可以分成两类：有些 WH 对应的是实体专名，譬如 Who/When/Where，回答这类问题相对容易，技术已经成熟。另一类问题则不然，譬如 What/How/Why，回答这样的问题是问答学界的挑战。简单介绍一下这三大难题如下。

What is X？类型的问题是所谓定义问题，譬如 What is iPad II?（也包括作为定义的 who: Who is Bill Clinton?）。这一类问题的特点是问题短小，除去问题词 What 与联系词 is 以外（搜索界叫 Stop Words，搜索前应该滤去的，问答系统在搜索前利用它理解问题的类型），只有一个 X 作为输入，非常不利于传统的关键词检索。回答这类问题最低的要求是一个有外延和种属的定义语句（而不是一个词或词组）。由于任何人或物体都是处在与其他实体的多重关系之中（还记得么，马克思说人是社会关系的总和），要想真正了解这个实体，比较完美地回答这个问题，一个简单的定义是不够的，最好要把这个实体的所有关键信息集中起来，给出一个全方位的总结（就好比是人的履历表与公司的简介一样），才可以说是真正回答了 What/Who is X 的问题。显然，做到这一步不容易，传统的关键词搜索完全无能为力，倒是深度信息抽取可以帮助达到这个目标，要把散落在文档各处的所有关键信息抽取出来，加以整合才有希望（【立委科普：信息抽取】）。

How 类型的问题也不好回答，它搜寻的是解决方案。同一个问题，往往有多种解决档案，譬如治疗一个疾病，可以用各类药品，也可以用其他疗法。因此，比较完美地回答这个 How 类型的问题也就成为问答界公认的难题之一。

Why 类型的问题，是要寻找一个现象的缘由或动机。这些原因有显性表达，更多的则是隐性表达，而且几乎所有的原因都不是简单的词或短语可以表达清楚的，找到这些答案，并以合适的方式整合给用户，自然是一个很大的难题。

可以一提的是，我来硅谷九年帮助设计开发 deploy 了两个产品，第一个产品的本质就是回答 How-question 的，第二个涉及舆情挖掘和回答舆情背后的 Why-question。问答系统的两个最大的难题可以认为被我们的深层分析技术解决了。



InfoQ 活动专区全新上线

更多活动·更多选择

MORE ACTIVITIES MORE OPTIONS

一站获取所有活动信息

Geekbang 技客帮

极客邦科技

InfoQ

四庫全書

在她歌喉

600



扫描二维码
前往专区

云计算

让数据管理变得更轻松



扫描下载迷你书



老杨聊架构：每个架构师都应该研究下康威定律



作者 杨波

嘉宾介绍：杨波具有超过 10 年的互联网分布式系统研发和架构经验，曾先后就职于：eBay 中国研发中心(eBay CDC)，任资深研发工程师，参与亿贝开放 API 平台研发，携程旅游网(Ctrip)，任技术研发总监，主导携程大规模 SOA 体系建设，唯品会 (VIPShop)，任资深云平台架构师，负责容器 PaaS 平台的调研和架构，目前就职于法国 LVMH 集团中国区的垂直电商部门，任职电商首席架构师，帮助传统 IT 向互联网转型。

背景

今天的分享主要来自我之前的工作经验以及平时的学习总结和思考。我之前的背景主要是做框架、系统和平台架构，之前的工作过的公司 eBay、携程、唯品会都是平台型互联网公司，所以今天主要带着平台架构视角和大家分享心得体会。架构的视角每个人都一样，可以说

一万种眼光，有业务架构、安全架构、平台架构、数据架构，各不相同，这里仅是我的一家之言，欢迎大家加入『聊聊架构』社群参与讨论。今天聊的话题主要包括以下几点：

- 我对架构定义的理解
- 架构的迭代和演化性
- 构建闭环反馈架构 (Architecting for

closed loop feedback)

- 谈谈微服务架构和最新主题
- 架构和组织文化关系
- 架构师心态和软技能
- 我对一些架构师争议主题的看法

我对架构定义的理解

大概在 7~8 年前，我曾经有一个美国对口的架构师导师，他对我讲架构其实是发现利益相关者（stakeholder），然后解决他们的关注点（concerns），后来我读到一本书《软件系统架构：使用视点和视角与利益相关者合作》，里面提到的理念也是这样说：系统架构的目标是解决利益相关者的关注点。（见图 1）

这是从那本书里头的一张截图，我之前公司分享架构定义常常用这张图，架构是这样定义的：

1. 每个系统都有一个架构
2. 架构由架构元素以及相互之间的关系构成
3. 系统是为了满足利益相关者(stakeholder)

的需求而构建的

4. 利益相关者都有自己的关注点 (concerns)
5. 架构由架构文档描述
6. 架构文档描述了一系列的架构视角
7. 每个视角都解决并且对应到利益相关者的关注点。

架构系统前，架构师的首要任务是尽最大可能找出所有利益相关者，业务方，产品经理，客户 / 用户，开发经理，工程师，项目经理，测试人员，运维人员，产品运营人员等等都有可能是利益相关者，架构师要充分和利益相关者沟通，深入理解他们的关注点和痛点，并出架构解决这些关注点。架构师常犯错误是漏掉重要的利益相关者，沟通不充分，都会造成架构有欠缺，不能满足利益相关者的需求。利益相关者的关注点是有可能冲突的，比如管理层(可管理性) vs 技术方(性能)，业务方(多快好省) vs 技术方(可靠稳定)，这需要架构师去灵活平衡，如何平衡体现了架构师的水平和价值。

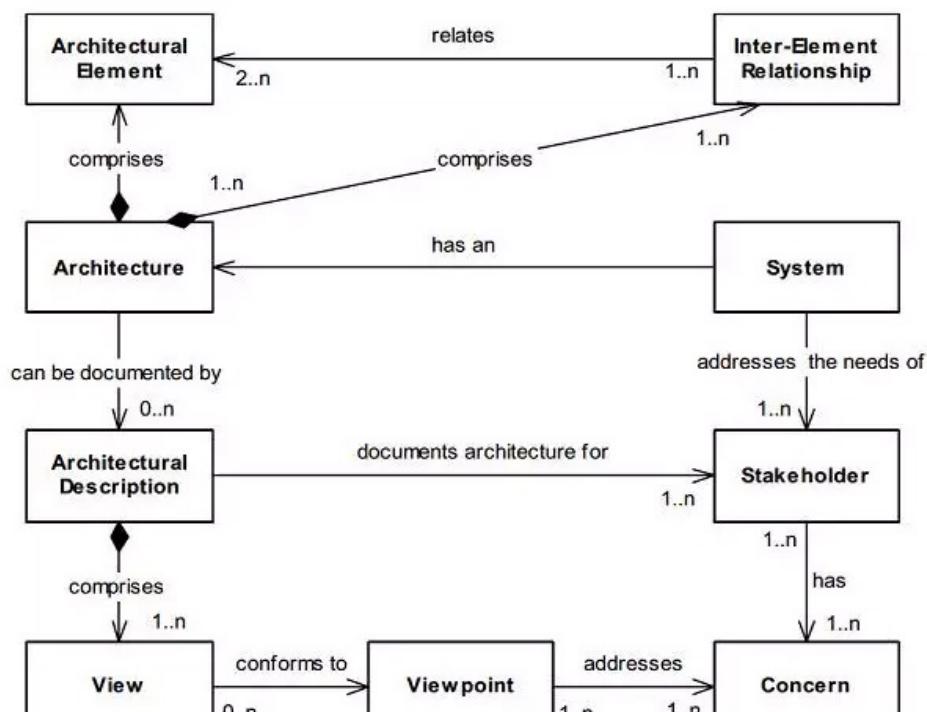


图 1

关于架构的第二点定义是说架构主要关注非功能性需求（non-functional requirements），即所谓的 -abilities。

图 2 是我上次公司内分享的一个图。

图 3 是 slideshare 一个 ppt 里头截取的，两个图都是列出了架构的非功能性关注点；关于架构的水平该如何衡量，去年我看到一句话，对我影响很大。

Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

翻译为中文就是，架构表示对一个系统的成型起关键作用的设计决策，架构定系统基本就成型了，这里的关键性可以由变化的成本来决定。这句话是 Grady Booch 说的，他是 UML 的创始人之一。

进一步展开讲，架构的目标是用于管理复杂性、易变性和不确定性，以确保在长期的系统演化过程中，一部分架构的变化不会对架构的其它部分产生不必要的负面影响。这样做可以确保业务和研发效率的敏捷，让应用的易变部分能够频繁地变化，对应用的其它部分的影响尽可能的小。



图 2

Architectural requirements

- Easy to separate → Autonomy
- Easy to understand → Understandability
- Easy to extend → Extensibility
- Easy to change → Changeability
- Easy to replace → Replaceability
- Easy to deploy → Deployability
- Easy to scale → Scalability
- Easy to recover → Resilience
- Easy to connect → Uniform interface
- Easy to afford → Cost-efficiency
(for development & operations)

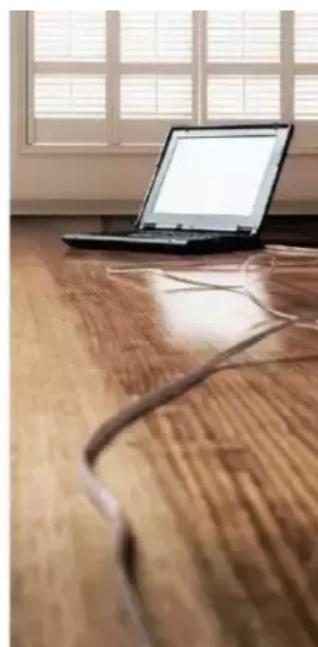


图 3

我刚入软件开发这个行业之初，谈的架构主要是性能，高可用等等。现在，见过无数遗留系统，特别是国内企业 IT 的现状，无数高耦合的遗留系统，不良的架构像手铐一样牢牢地限制住业务，升级替换成本非常巨大，所以我更加关注可理解，可维护性，可扩展性，成本。我想补充一句，**创业公司创业之初获得好的架构师或技术 CTO 非常重要。**

架构的迭代和演化性

我是属于老一代的架构师，99 年参加工作。职业初学了很多 RUP，统一软件过程的理念。RUP 的理念对我的架构有很深的影响，RUP 总结起来就是三个特点：

1. 用例和风险驱动 Use Case and risk driven
2. 架构中心 Architecture centric
3. 迭代和增量 Iterative and incremental

RUP 很注重架构，提倡以架构和风险驱动，项目开始一定要做端到端的原型（prototype）；通过压测验证架构可行性，然后在原型基础上持续迭代和增量式开发，开发 → 测试 → 调整架构 → 开发，循环，如下图 4 所示：

从上图可以看出架构师要尽可能写代码，做测试，纸上谈兵式做架构而后丢给团队的作法非常不靠谱（除非是已经非常清晰成熟的领域）。另外，做技术架构的都有点完美主义倾向，一开始往往喜欢求大求全，忽视架构的演化和迭代性，这种倾向易造产品和用户之间不能形成有效快速的反馈，产品不满足最终用户需求，继续看后面图 5、图 6。

第一个图是讲最小可用产品（Minimum Viable Product，MVP）理念，做出最小可用产品，尽快丢给用户试用，快速获取客户反馈，在此基础上不断迭代和演化架构和产品。第二个图是过度工程（Over Engineering）的问题，其实也是讲产品架构和用户之间没有形成有效的反馈闭环，架构师想的和客户想的不在一个方向上，通过最小可用产品，快速迭代反馈的策略，可以避免这种问题。**注意，在系统真正地投入生产使用之前，再好的架构都只是假设，产品越晚被使用者使用，失败的成本和风险就越高，而小步行进，通过 MVP 快速实验，获取客户反馈，迭代演化产品，能有效地减少失败的成本和风险。**

另外，多年的经验告诉我，架构，平台不是买来的，也不是用一个开源就能获得的，也不是

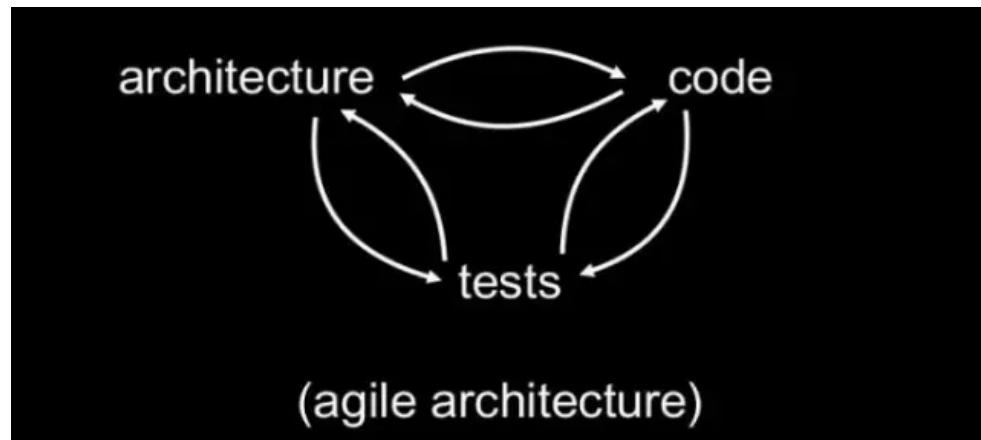


图 4

Minimum Viable Product

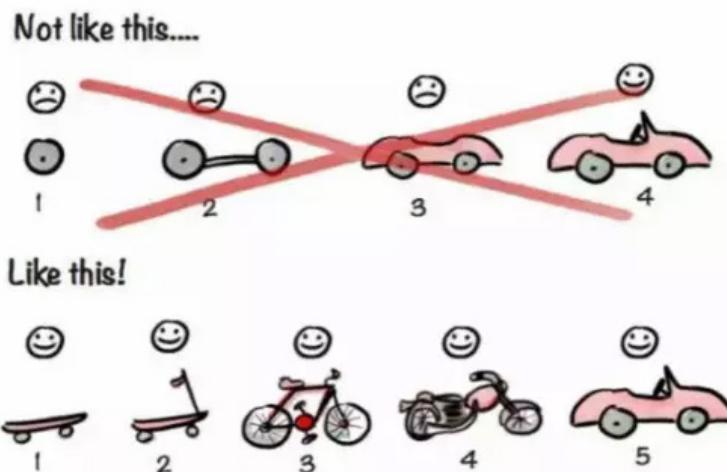


图5

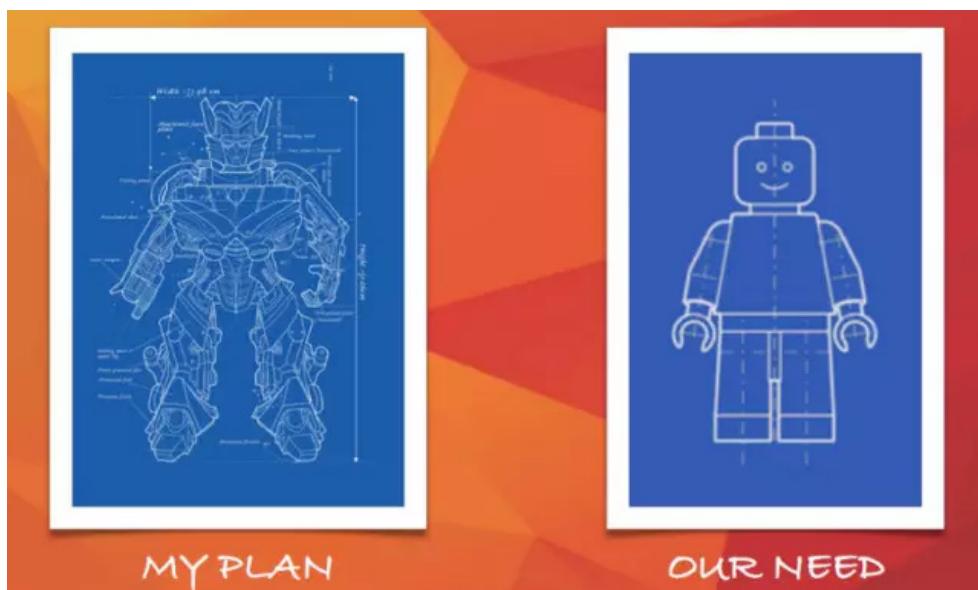


图6

设计出来，而是长期演化才能落地生根的。

给大家推荐两篇不错的微信文章（微信不能插入链接，根据题目 Google 下即可）：

1. 58 同城沈剑：好的架构源于不停地衍变，而非设计
2. 宜人贷系统架构 - 高并发下的进化之路

两篇文章其实都是讲架构的迭代和演化性，值得每个架构师学习吸收。

构建闭环反馈架构

先分享一个链接，这几年对我架构影响最深的一篇文章。这篇文章是关于 DevOps 的，但对系统架构同样适用：

<http://itrevolution.com/the-three-ways-principles-underpinning-devops/>

这篇文章讲述了企业通向 DevOps 的三条必经之路，我们来看看这三条道路对架构师的启示。（见图 7）

第一条道路，系统思维，开发驱动的组织机体，其能力不是制作软件，而是持续的交付客户价值，架构师需要有全局视角和系统思维（System Thinking），深入理解整个价值交付链，从业务需求、研发、测试、集成，到部署运维，这条价值链的效率并不依赖于单个或者几个环节，局部优化的结果往往是全局受损，架构师要站在系统高度去优化整个价值交付链，让企业和客户之间形成快速和高效的价值传递。（见图 8）

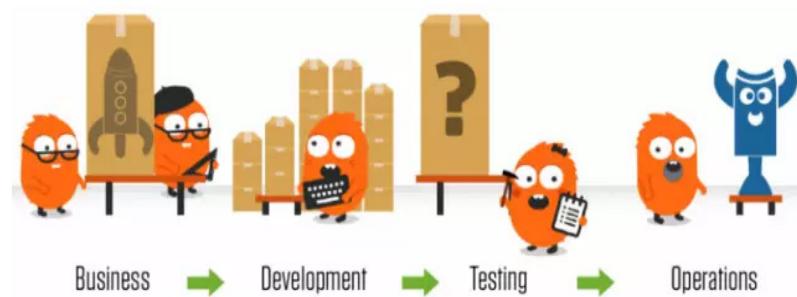
第二条道路，**强化反馈环**，任何过程改进的目标都是加强和缩短反馈环。刚入行的工程师，也是中国学生的普遍问题，就是生产运维意识不足（监控是系统反馈的重要环节）。有两句话这样讲的：

1. no measurement, no improvement 没有测量，就没有改进和提升
2. What you measure is what you get 你测量什么，就得到什么

The First Way: Systems Thinking



图 7



The Second Way: Amplify Feedback Loops

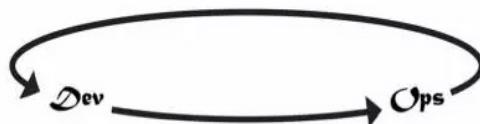


图 8

没有监控或者监控不完善的系统相当于裸奔，开车上高速无仪表盘。有一篇很不错的关于测量驱动开发的文章，在InfoQ上的，向大家推荐：

<http://www.infoq.com/cn/articles/metrics-driven-development>

这篇文章提出了度量驱动开发的理念，即所谓MDD，在系统、应用和业务三个层次，通过三级监控，构建三个反馈环，在监控测量基础上持续改进系统和架构，我最近也在参考这个理念设计一个电商平台的技术架构，见图9：

这是一个电商平台的架构，整个体现了闭环架构的思想，右侧是整个平台的反馈监控环节。具体如下：

1. 系统层监控计算网络存储，构建系统层的反馈环
2. 应用服务层，监控业务、应用、服务，甚

至整个研发流程，构建应用和服务层的反馈环

3. 客户体验层，监控端用户和分析网站用户的行为，构建和客户的反馈环

下面这个图展示了系统提升和改进的一般方法，见图10：

收集→测量→调整→闭环重复，在有测量数据和反馈的基础上，系统、应用、流程和客户体验才有可能获得持续的提升和改善，否则没有数据的所谓改进只能靠拍脑袋或者说猜测。（见图11）

第三条道路，鼓励勇于承担责任，冒险试错和持续提升的文化。这点是最难的，一般和企业人才密度有关。工具、技术、流程只是一个公司的冰山浮出水面的部分，而真正对企业效能影响大的则是冰山水下的部分，即企业的人和



图9

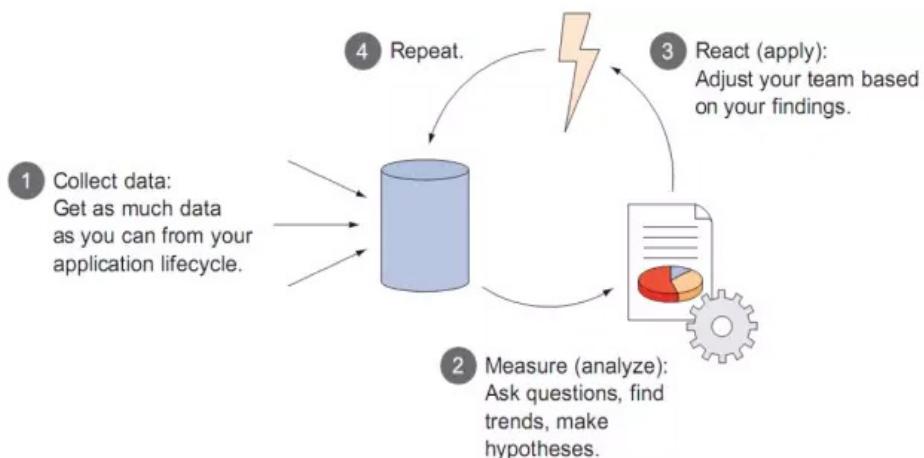


图 10

The Third Way: Culture Of Continual Experimentation And Learning

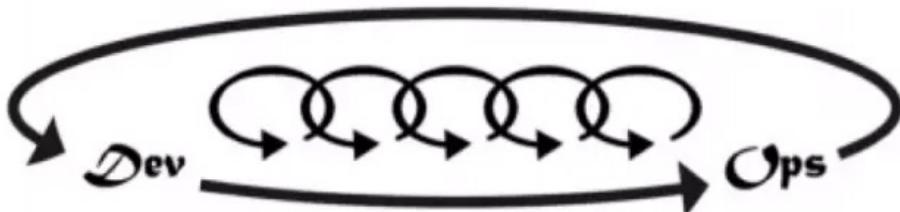


图 11

文化，架构师作为技术和架构的布道者，有责任义务鼓励和推动试错文化。

架构师要深入领会这三条道路，关注整个价值交付链的效率，关注每个环节的闭环反馈，鼓励和推动公司的试错文化，打造全系统的闭环架构（Architecting for closed loop feedback），提升整个系统效能。下图的 DevOps 和每日交付是每一个互联网系统架构师的梦想和努力的方向。（见图 12）

谈谈微服务架构

微服务我想大家都听得很多了，我本人也非常

关注和推崇微服务，从技术角度讲，我认为微服务主要体现的是单一职责和关注分离的思想，从单进程模块化进一步拓展到跨进程分布式的模块化。微服务是独立的开发、测试、部署和升级单元，正如我在第一点架构定义中提到的，微服务中每个服务可以独立演变，它的 cost of change 比较小，整体架构比较灵活，是一种支持创新的演化式架构。

去年 MartinFowler 写了两篇文章，给微服务泼冷水的，建议大家好好读下。

1. <http://martinfowler.com/bliki/MicroservicePrerequisites.html>
2. <http://martinfowler.com/bliki/MicroservicePremium.html>

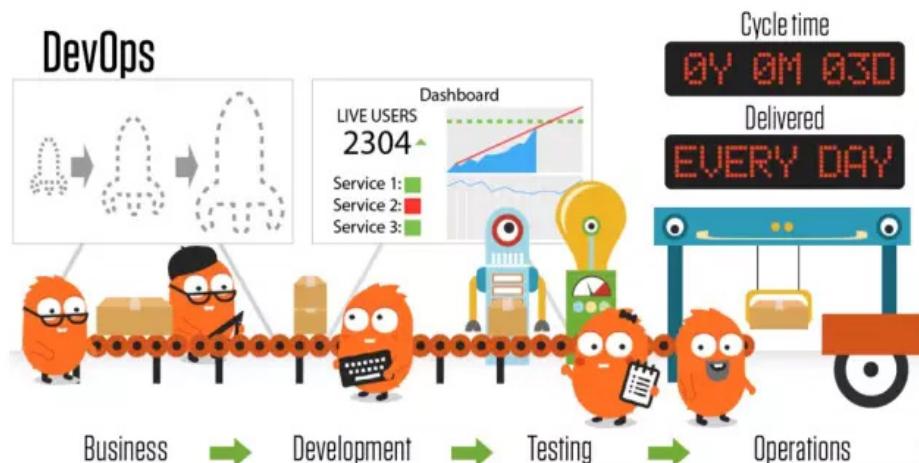


图 12

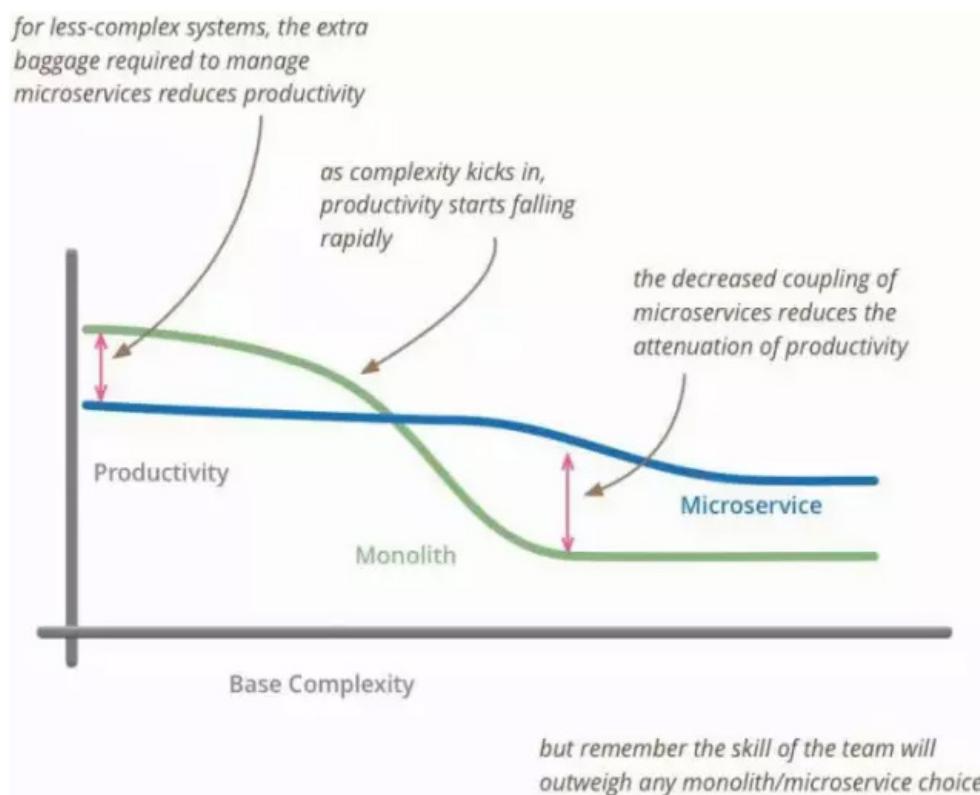


图 13

图 13 讲什么时候该引入微服务。微服务有额外成本的，需要搭建框架、发布、监控等基础设施。初创和小规模团队不建议采用。主要决定是因素系统复杂性和团队规模，当到达一个点，单块架构严重影响效率才考虑。另外补充一点，**微服务更多是关于组织和团队，而不是技术。**

架构和组织文化关系

再谈一下康威定律：

Conway's law: Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.

（设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。）

从单块架构到微服务架构是这个定律的很好体现。（见图 14）

团队是分布式的，系统架构是单块的，开发，测试，部署协调沟通成本大，严重影响效率，严重时团队之间还容易起冲突。（见图 15）

将单块架构解耦成微服务，每个团队开发，测试和发布自己负责的微服务，互不干扰，系统效率得到提升。

可见，组织和系统架构之间有一个映射关系（ ~ 1 mapping），两者不对齐就会出各种各样的问题，一方面，如果你的组织结构和文化结构不支持，你也无法成功建立高效的系统架构，例如集中式和严格职能（业务，Dev，QA，Deployment，Ops）的企业，很难推行微服务和 DevOps，推行 Docker/PaaS 平台也会比较

困难，这样的组织职能之间都倾向于局部优化（local optimization），无法形成有效的合作和闭环。

反过来也是成立的，如果你的系统设计或者架构不支持，那么你就无法成功建立一个有效的组织；作为系统架构师，一定要深入领会康威定律，设计系统架构之前，先看清组织结构，也要看组织文化（民主合作式，集权式，丛林法则式，人才密度），再根据情况调整系统架构或者组织架构。

架构师心态和软技能

空杯，或者说开放心态（open minded）是一个成熟架构师的应有心态，stay hungry, stay foolish，心态有多开放，视野就有多开阔。来自《高效能人士的七个习惯》史蒂芬

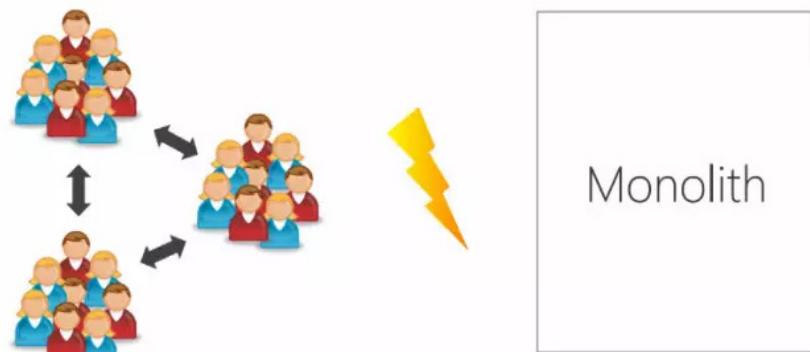


图 14

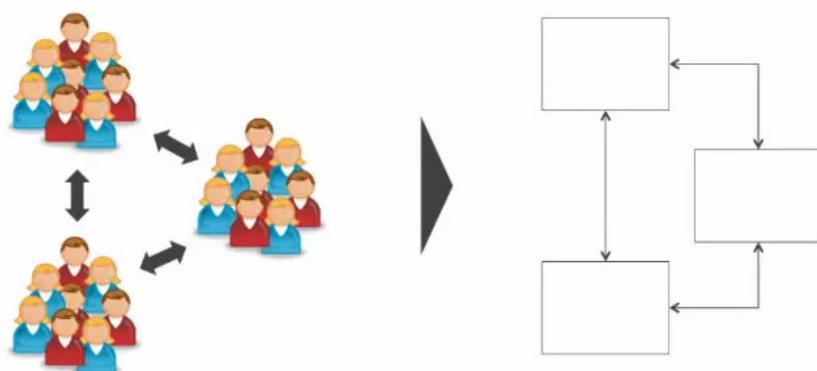


图 15

~ 柯维》的一句话送给每一个架构师：

如果一位具有相当聪明才智的人跟我意见不同，那么对方的主张必有我尚未体会的奥秘，值得加以了解。与人合作最重要的是，重视不同个体的不同心理、情绪与智能，以及个人眼中所见的不同世界。与所见略同的人沟通，益处不大，要有分歧才有收获。

另外再推荐一个本书《软件架构师的 12 项修炼》，这书中三个观点很值得每个架构师学习领会：

1. soft skills are always hard than hard skills, 软技能比硬技能难
2. choosing relationship over correctness , 注重关系重于谁对谁错
3. 架构的政治性，在中大型公司里工作的架构师尤其要学习

政治指的是和他人协作将事情搞定的艺术，架构是一种社交活动，在技术的世界里，个人主义很容易被打败，即使你的目的是好的技术是最优的，技术决策是政治决策（technical decisions are political decisions），一个技术产品，一波人可以做，另一波人也可以做，到底谁做的好，真不好说，不管谁做，都给业务套上了一副手铐。

架构师如何提升？实战，实战，实战！规划职业，找好的团队和项目，总结分享，学习 GitHub 开源项目，尽可能参与和开创自己的开源项目和产品，并长期积累。

我对一些架构师争议主题的看法

主要争议是两个话题：

1. 技术和业务的关系。
2. 架构师要写代码吗？

架构师怎么回答这类问题？一个成熟架构师的口头禅：视情况而定，不一定，是也不是，it depends。技术和业务，架构师要理解业务吗？看产品和客户，如果是业务性产品，肯定要理解业务，如果是技术型产品，就不一定。

架构师要写代码？也不一定，个人觉得尽可能要写，如果你写过十年以上代码了，每年不少于 2 万行，都玩通了，可以不写。另外架构师如果写代码，要把控度，不要一头钻入代码，花大量时间解决细节和复杂性问题，忽视全局和系统性问题。

最后

我想说中国现在的互联网发展趋势很好，越来越多的人加入架构师这个行业，这个行业正在“万物生长”。但是我们现在还没有马丁福勒，adrian cockcroft 这样的架构牛人物，我辈需不断努力，期待中国 10~20 年后出现超过十个马丁福勒，adrian cockcroft 这样的大牛神级人物。我们必不可停止探索，而一切探索的尽头，就是重回起点，并对起点有首次般的认识。

电商网站的初期技术选型



作者 崔康

今天在架构师俱乐部 3 群（由 ArchSummit 全球架构师峰会运营）里，大家围绕着一个话题讨论地很热烈——完全从 0 到 1 建设一个电商网站，技术选型和注意事项有哪些？群友们都结合自己的实际工作经历分享了很多经验教训，这里是其中的精选。

青岛海尔 Jan 给大家分享了一个失败案例的教训：

1. 没有准确估计实际业务量或者说就没有估计过，导致技术选型直接参考京东、淘宝一线大公司，实现较复杂，技术铺的也很大。（教训：技术够用就好，选型的目标

是能够快速实现产品的迭代）

2. 因为缺少经验，前期业务没有明确的规划，技术选型也没有考虑高内聚、低耦合，导致系统之间依赖太强，导致现在想拆分很难。
3. 选择了一些较新的技术框架，过于依赖几位关键的技术牛人，结果这些人一旦离职，就陷入迷茫。（教训：关键人员一定要留住，或者有备份人选）

还有群友表示，对于早期的技术选型，不要抄大公司经验，按需求出发，先活下来再说。要考虑到哪些是可以省的，那些是可以用现成的，哪些是需要有特色自己开发的。早期手段可以

粗糙，尽量考虑云。云服务真的可以解决很多创业初期的一些棘手问题，而且可以省下成本。

0到1解决的是卖什么、怎么卖、卖给谁的问题。思考的角度分为技术化域和商业化域。技术域，是实用生存为主，不求高大上，但求快速实用；商业域，就是经营变现了，细分领域夺城拔寨。重要的是，寻找到盈利点，建立合适的商业模式，然后通过技术来实现，除非技术驱动产品这样的公司完全以技术为核心，掌握核心就掌握市场。

如果不及时考虑盈利问题，可能面临以下的问题：公司跟风耗费巨资投入一个项目，技术部门找了很多，人，并且采用了各种高大上的项目，结果盈利没跟上来，最终公司决定不再急需投入，项目就黄了，研发团队也解散了……

上海微肯 CTO 孔燕斌则认为，流量是电商的最大成本和成功的关键因素之一，关于流量的问题其实就是怎么卖和卖给谁的问题。现在线上流量的成本是非常高的，而且传统的线上流量都是一次性的，为什么叫一次性，是因为即使是同一个用户，要再次唤醒，大部分时候还是要支付额外的成本，流量的成本谁一值跟着运营高企不下。这里非常推荐微信上的流量，微信的流量有几个好处，一个是用户充足，第二个是有公众号，可以免费唤醒用户，第三个是有社交属性，可以通过朋友圈、券、微信支付等微信能力进行营销。其中对初创的企业最重要的是可以通过微信的近场能力在线下拉人，使用很低的成本高效地拉人，快速验证。

线下拉人的最大好处是可以通过选择地点，来圈定自己的潜在用户。要做到这一点，系统架构的时候需要增加微信的模块，实现和微信的

和和相关的营销功能。快速找到第一批客户验证业务，完成 0 到 1，完成之后是 1 到 100，1000，10000 的复制都可以在微信这个流量里面很好地展开，并且有效地降低运营中的流量的成本。基于篇幅的限制，在此不一一展开。

Jan 认为，关于网站的流量，严格来说流量 ≠ 客户量，当然这也得看如何定义。流量更多看的是网站访问者或是 App 使用者的访问情况，从进入第一个页面到最后退出，这样一个全流程。客户量，相对于网站来说，我的注册客户多少，日访问客户多少（流量分析工具可以实现），成交客户量等等，需要结合实际公司的对客户量的定义，结合流量分析。

初期除了购买流程上不能有技术短板外，产品为核心的营销数据流也很重要。提升流量，用流量测试转化率和动销率，然后想办法提升这两点。一旦转化率稳定，才是买大流量的时候。这些都要有数据支撑试错。

LAANTO 王巍表示，架构其实是妥协的结果，受投入、团队技术水平多方面影响的，够用就好。从基础做好上云的准备，比如用 memcache，redis 等分布式缓存系统，把应用改造成与状态无关，一方面可以做到容易扩容，随时增加节点，另一方面可以足够的可靠性，从而降低各方面成本；在成本有限的情况下，使用成熟技术，达到最优性价比即可，力争达到 good，不放弃对 perfect 的追求；片面要求百分百可靠的都是异端。满足 80% 的高质量用户需求就够了。技术还得结合投入的多寡，凡是都有个投入产出比，因此要管理好老板的期望和用户的期望，所谓量力而行，做人如此，技术也是如此，做企业更是如此。秉承恰当的技术做恰当的事的理念。

就 App 而言，很多时候做 App 是为了估值。当然，依附与微信等高流量入口可以快速获取用户，缺陷在于人家的地盘听人家的，有着诸多限制，当用户积累到一定程度，业务受限于其平台的时候，做 APP 就成了必然的选择，所谓因时而动，顺势而为。

孔燕斌：从 0 到 1 的时候需求上的假设都没有验证，没必要去折腾 App，集中力量，快速把微信搞定，验证需求，累积用户，收集用户反馈。然后才能确定是否真的需要 App，绝大部分的 App 都是伪命题。一个 App 如果需求不找对，并且没有竞争对手，可以自然增长，靠补贴的话，一个用户 20 块钱都不一定够。所以需求需要验证的，觉得很美妙的未必可行，不咋样的其实会很不错，是驴子是马都得拉出来溜过才知道。

速普母婴 Martin 说，我是做母婴电商这块的，从去年 4 月份到现在，也是经历了团队从 0 到 1，产品从 0 到 1 的过程，说说我的一点理解：

1. 人是最重要的，有个靠谱的 CTO 其实已经成功了一大半，CTO 的经验决定了未来产品的技术栈啊。一些小创业公司仰慕某些巨头的技术架构，技术专家，然后不惜花重金请来，专家出了各种高大上的方案，对么？巨头专家当然说的方案不能说不对，但是创业公司有可能还没到那个体量和基础，最重要的是，干活的技术人员，有可能连最基本的优化逻辑都没掌握呢。
2. 业务。产品初期能正常下单，库存能锁住，服务器稳定高可用就可以了。
3. 技术。我的理解是拿来主义，有现成的或者自己能掌握的技术千万不要去用那些最新的，一是新技术会引入时间成本，创业

公司一般耗不起啊，另外新技术的把控不住可能会在未来造成难以预估的灾难。

我们第一期做的比较简单，主要分三块：前端、业务层、数据层。前端分移动端（Android、IOS）、PC 端，业务层开放 restful 接口给前端调用，http 协议 json 传输数据，前后端分离，分开部署，接口文档工具采用了阿里的 rap，减少前端后端人员的沟通成本。其中前端主要 nginx 分流，当然，还没用现在主流电商采用的 nginx+lua，因为 lua 大家都没底把控不了。其次图片类的静态文件对接了三方的文件存储系统（又拍）。

后端业务层采用了 springmvc+mybatis，应用服务器是 tomcat，搜索业务采用了 solr，还有几台队列服务器 rabbitmq（用在订单业务上）。至于数据层，则分为分布式缓存和持久化数据。分布式缓存采用了豌豆荚开源的 codis 方案，那时候 redis3.0 刚出来，不敢踩坑果断放弃了，其实也可以直接用 ssdb 双主，毕竟 redis 太耗内存了，尤其对创业型公司来说，省钱是最主要的，ssdb 和 redis 对比，读性能差的不大，并且 ssdb 采用 leveldb 做文件存储（当然也可以用 rocksdb 存储），摆脱了内存的限制，在京东等一些网站都有成功的案例。

至于持久化数据这层（mysql），考虑到电商业务初期，采用了读写分离，选择了 MHA 方案（LVS+Atals+MHA），还有数据库设计，不要用数据库特有的，比如存储过程，还有反范式设计，减少表的关联查询，对后期的分离、服务、可读性做考虑。

谢文渊表示，从 0 到 1 建电商上面同学把一些关键点都说了非常清楚，我做过几个这种从 0 到 1 的电商，说说我的几点看法：

从 0 到 1，说明是一个企业的一个新的 IT 领域，很多业务策略和基础根基都是不成熟，不管是业务还是技术架构，同时还有个共同特征：上线周期短，新团队在上面的情况下，有几个方面是需要重点关注的：

1. 业务流程

这一块是所有工作的基础，包括调研和梳理业务流程，主要涵盖正向流程如：采购、会员管理、商品价格、上下架、购买、订单管理、发货、财务等，逆向的更麻烦，如退换货、退款等另一个核心就是促销规则，如套餐、团购、满赠、买赠、折扣、优惠券等，这个可先从简单入手，只是在架构设计考虑扩展项目周期原因，必须的关键活动：会员注册、登录、购买支付、订单审核、发货、对帐。

2. 应用架构

一开始业务量小，应用拆分适可而止，初期建议有商城前端、后台管理、订单管理、物流发货商城前端的演变将会是快速的，不管是业务模式还是用户量规模，都会促使商城前端的快速迭代，其技术要求也是最高的，大多数在行业内分享的技术也都集中在这一块后台管理主要处理运营商城的需求，在线配置是重点，包括 CMS 订单管理也是后台应用，接口相对也多一些，如与 ERP、WMS 等，很多企业也在第三方电商平台销售，如天猫、亚马逊等，可以接口接入物流发货比较规范，可以考虑外购，如 WMS，也可外包，可省很多事。

3. 技术领域

具体的技术细节不谈了，非常同意前面同学说的够用即可，不要追求高大上，也不要学大平台的架构，这些架构也是从最简单的架构开始的，到现在这样也是被业务迫使的。一定要用团队最熟悉的技术或架构师最熟悉的，有几点可以参考：确定技术标准，如分层，开发规范，采用的开源框架等，并培训抽取基础包或框架包，这个可以在边开发应用边抽取，如通用的 Util、缓存操作类、数据访问等（这个好象所有软件项目都是这样，但很关键）初期不建议按模块拆分系统，做好模块划分，在配置管理上做区分即可虽然拒绝过度设计，但扩展性和安全性一定要考虑，提前考虑扩展性会让你在后续演变过程中如鱼得水，尤其是商城前端的水平扩展，通常受到数据（配置或可卖数等）和会话的一致性限制，会话可以用 memcache 来管理，数据可加载到缓存如 redis，一个可减少 DB 压力，二个能屏蔽 DB 层的演变，如分表分库等。

安全性是互联网上应用的永恒主题，可在框架层置入 XSS 和 SQL 注入的过滤器静态资源和动态内容做分离，商品详情页可做成伪静态化，静态和伪静态资源都可发布到 CDN 上，对用户体验还是很有帮助的，万一流量大时，也能保护后台服务，并且可减少带宽 CMS 可以从一些开源框架上做些改造来用，主要针对一些活动、首页的一些配置，如果有 WAP 和 APP，可以用下阿里的 RAP 来管理接口，会大大提高接口的可管理性值得好好设计的几个地方：商品模型、促销规则引擎、多类型订单模型、第三方订单接入适配层部署上一般采用 LVS（土豪可用商用的，如 F5/Array）nginx（or other web server）app server DB（or cache），一开始

一般每层搞个双节点就好了。

另外，为了提高业务连续性，可以采用灰度发布，可以简单写些脚本，不一定要高大上的工具如果仅仅是从 0 到 1，甚至在性能上都是可演进的，很多在并发容量、性能及高可用方面，是 1 之后的事情了以上只是简单从 0-1 的描述，但实际上细节还是挺多的。对了，电商也算是

互联网应用，对流量统计和转化率是运营的抓手，这些数据一定要有，流量可以用百度的就行，转化率得从后台出数据了，做些简单报表也是必须的。



ArchSummit

技术峰会,让学习交流畅通无阻!

ArchSummit传承经典案例,引领未来技术!

2016年7月15-16日/深圳南山区华侨城洲际酒店
中国·深圳



传承经典

云服务架构探索、发展中的移动架构技术、社交网络等专题带您领略经典行业的技术变化



跟进前沿

大数据和个性化及高可用架构专题与您一同探索热门技术方向的更迭起伏



引领趋势

智能硬件、机器学习、虚拟现实于您携手观看最新技术前沿的波澜壮阔

3月20日前 **7** 折购票
团购优惠更多

sz2016.archsummit.com



线上专家零距离沟通

想有更多的思想碰撞吗?

请扫描上方二维码

“ArchSummit技术关注”

为您提供足够的专家交流平台!

InfoQ 中文站 迷你书



本期主要内容：OpenJDK将对Android开发产生怎样的影响？，Python将迁移到GitHub，鏖战双十一-阿里直播平台面临的技术挑战，从无到有：微信后台系统的演进之路，推荐系统和搜索引擎的关系，关于云迁移的经验总结



解读2015

希望“解读2015”年终技术盘点系列文章，能够给您清晰地梳理出技术领域在这一年的发展变化，回顾过去，继续前行。



顶尖技术团队访谈录
第四季

本次的《中国顶尖技术团队访谈录》·第四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。



10个精选的容器
应用案例

《10个精选的容器应用案例》是InfoQ旗下的CNUT (容器技术俱乐部) 推出的容器应用白皮书，旨在通过优秀的案例引导国内社区和企业正确使用相关的容器技术，以加速容器技术在国内的普及。