

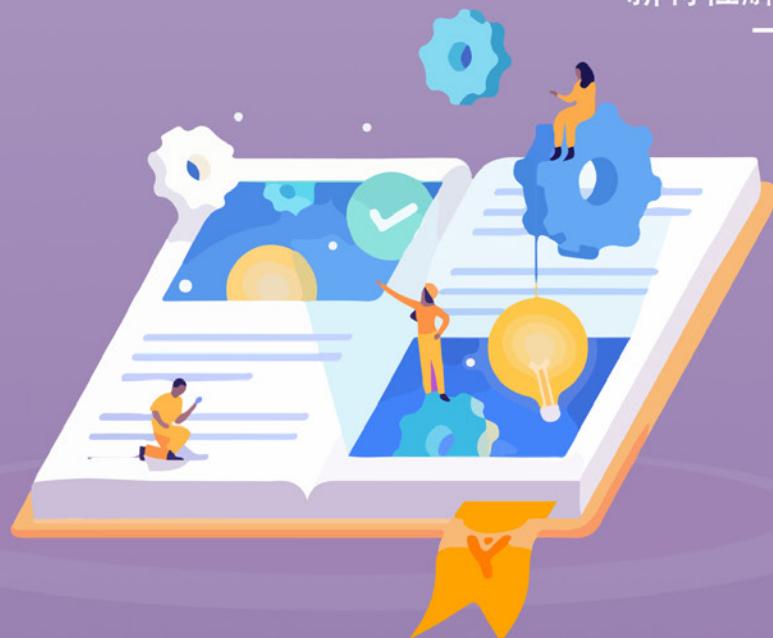
# 架构师

ARCHITECT

10月刊

## 热点

Java 11正式发布，  
新特性解读



Geekbang | 极客邦科技

InfoQ



## 全球软件开发大会

### ▶ 聚焦

- 互联网高可用架构
- 国际化互联网业务架构
- 工程师个人成长与技术领导力
- 架构设计
- 后移动互联网时代的技术思考与实践
- 大规模基础设施DevOps探索
- 硅谷人工智能
- 人工智能与业务实践
- Java生态与创新
- 大数据系统架构
- 区块链技术与应用
- 前端新趋势
- 深度学习技术与应用
- 微服务架构 & Serverless
- 产品经理必修之用户细分与产品定位

### ▶ 实践

**Facebook /** 硅谷公司的互联网计算性能优化经验谈

**LinkedIn /** 推荐系统：提升用户增长与参与的利器

**Uber /** 核心Trip Flow容量管理

**Confluent /Apache Kafka /** 从0.8到2.0：那些年我们踩过的坑

**快手 /** 如何快速打造高稳定千亿级别对象存储平台

**微软 /** 集成AI开发平台实践

会议：2018年10月18–20日

培训：2018年10月21–22日

地址：上海·宝华万豪酒店

100+ 技术  
的实

## ▶ 大咖助阵



专题：硅谷人工智能

夏磊 / LinkedIn高级工程师，  
湾区同学技术沙龙Board Member



专题：Java生态与创新

张建锋 / 永源中间件 共同创始人



专题：互联网高可用架构

吴其敏 / 平安银行  
零售网络金融事业部首席架构师



专题：研发效率提升

徐毅 / 华为 技术专家



专题：产品经理必修之用户细分与产品定位

袁店明 / Dell EMC  
敏捷与精益创业咨询师

.....

## ▶ 分享嘉宾



David Cheney  
Heptio  
资深工程师  
著名Go语言专家



Julien Viet  
Red Hat  
首席软件工程师



李怀根  
广发银行  
研发中心总经理



庄振运  
Facebook  
计算机性能高级工程师



邹欣  
微软亚洲研究院  
首席研发经理



李欣 (Bruce Li)  
Paypal  
PayPal Risk Infra  
Director level architect



Hien Luu  
LinkedIn  
项目经理



Jonathan Giles  
Microsoft  
Senior Cloud Developer Advocate

.....



# CONTENTS / 目录

## 热点 | Hot

微软开源 Sketch2Code，草图秒变代码

Java 11 正式发布，新特性解读

秒杀 Redis 的 KVS 数据库上云了！伯克利重磅开源 Anna 1.0

## 理论派 | Theory

架构师如何判断技术演进的方向？

## 推荐文章 | Article

后 Kubernetes 时代的微服务

## 观点 | Opinion

GitHub：我们为什么会弃用 jQuery？



## 架构师 2018 年 10 月刊

本期主编 覃 云

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

# 卷首语

## 数据浪潮之间的前端工程师

阿里南京研发中心前端工程师 王下邀月熊

十年来，波澜壮阔的移动互联网浪潮促进了 Web 技术的迅猛发展，随着浏览器性能、网络带宽等基础设施的提升，Web 也能够承载起包含复杂交互、可视化、计算逻辑需求的富客户端应用。同时 RN、Weex、小程序为代表的混合式开发日趋成为与 Android、iOS 原生开发并肩的开发模式之一；而 VR、AR、IoT 等新的交互方式或者媒介也正步入消费级市场，原本前端之间的隔阂逐渐消亡，我们慢慢进入了大前端的时代。笔者认为大前端不仅仅是横向地指泛 GUI 的接入端，纵向来看基于 Node.js 的全栈开发、中台化背景下的 BFF 模式，微前端架构等也是大前端的有机组成，也给予了前端工程师更广阔地舞台。

### DT 时代

繁多的互联网接入端也催生了海量数据的产生与富集，开启了所谓的 DT 时代；我们利用云计算、人工智能、深度学习等手段分析数据、利用数据，将数据作为燃料，赋能新的商业模式。算法、数据与工程是优秀的智能化产品不可或缺的组成部分，前端作为与数据的产生源头--用户最贴近的部分，也在未来全连接的架构里迸发出更绚烂的火花。

前端首先能够通过埋点、监控等方式，采集到用户行为、偏好、应用运行状态等丰富的数据，我们团队(阿里南京 NUE)也自研了高性能 Web

实时录屏与回放的产品，赋能客户服务的体验提升与前端开发者的线上调试。基于数据与算法，前端也可以设计更好地人机交互模式，譬如人脸识别的登录方式、智能问答客服、智能音箱的语音控制；数据可视化也是典型的前端与数据水乳交融的领域，ECharts、G2、D3、Three.js 等框架允许我们更便捷、友好地、深刻地展现统计数据、关系数据、地理空间数据、时间序列数据与文本数据等多源异构数据集。此外，TensorFire, TensorFlow.js 等深度学习框架利用客户端的 GPU 计算能力，pix2code 或者 SketchCode 利用算法来快速实现原型界面，Guess.js 能够帮助优化构建好的包体与智能添加预抓取策略。

值得一提的是，近几年区块链技术的爆炸性发展也促进了 Web 3.0 概念的思辨与实践，IPFS、Ethereum dApp 等工具或者开发框架允许我们便捷地编写去中心化的 Web 应用。Web 3.0 提倡以人为本，看重隐私，反垄断网络，旨在更开放的网络上进行集体贡献并实现共享利益，这也给予了前端开发者更多样化的未来。在 DT 时代，我们或许不能站在浪潮之巅，但是随波逐流顺势而下也可以找到自己的位置，或高或低地翱翔于天。

## 数据流驱动的界面

数据的核心操作是存储与计算，传统的 Web 应用因为单线程与离线不可用性往往是即用即走，而 PWA 这样的应用设计模式，提倡使用 Service Worker 添加离线支持，充分利用 IndexDB, CacheAPI 等进行灵活地数据存储与检索，并且给予用户贴近原生的体验。另一方面，Web Worker, WebAssembly 等亦从不同的方面释放或者增强前端的计算能力，不仅使得 Web 中运行高性能要求的应用或动画，也可以借鉴边缘计算的理念，未来将更多地数据聚合、计算的逻辑前移。感性地说，当数据逐渐活跃、富集，如百川汇海，自然需要流动起来。

广义的数据流驱动的界面有很多的理解，其一是界面层的从以 DOM 操作为核心到逻辑分离，其二是数据交互层的前后端分离。在 jQuery 时代，我们往往将 DOM 操作与逻辑操作混杂在一起，再加上模块机制的缺

乏使得代码的可读性、可测试性与可维护性极低；随着项目复杂度的增加、开发人员的增加与时间的推移，项目的维护成本会以几何级数增长。随着 ES6 Modules 的广泛应用，我们在前端开发中更易于去实践 SRP 单一职责原则，也更方便地去编写单元测试、集成测试等来保证代码质量。而像 React、Vue 这样现代的视图层库为我们提供了声明式组件，托管了从数据变化到 DOM 操作之间的映射，使得开发者能够专注于业务逻辑本身。并且 Redux, MobX 这样独立的状态管理库，又可以将产品中的视图层与逻辑层剥离，保证了逻辑代码的易于测试性与跨端迁移性，促进了前端的工程化步伐。

近两年来随着无线技术的发展和各种智能设备的兴起，互联网应用演进到以 API 驱动的无线优先(Mobile First)和面向全渠道体验(Omni-channel Experience Oriented)的时代，BFF 这样前端优先的 API 设计模式与 GraphQL 这样的查询语言也得到了大量的关注与应用。GraphQL 是由 Facebook 开源的查询语言标准，包含了数据格式、数据关联、查询方式定义与实现等等一揽子东西的数据抽象层。GraphQL 并不能消融业务内在的复杂度，而是通过引入灵活的数据抽象层，尽量解耦前后端之间的直接关联或者阻塞；在满足日益增长不断变化的 Web/Mobile 端复杂的数据需求的同时，尽可能避免服务端内部逻辑复杂度的无序增加。

## 工程化与微前端

编程生态往往经历三个阶段：涌现大量工具的原始阶段、复杂度提升后引入大量设计模式的框架阶段、具有更好的团队组织与协调机制的工程化阶段。大部分时候我们谈论到工程化这个概念的时候，往往指的是工具化；工具的存在是为了帮助我们应对复杂度，在技术选型的时候我们面临的抽象问题就是应用的复杂度与所使用的工具复杂度的对比。而工程化，即是面向某个产品需求的技术架构与项目组织，致力于尽可能快的速度实现可信赖的产品；尽可能短的时间包括开发速度、部署速度与重构速度，而可信赖又在于产品的可测试性、可变性以及 Bug 的重现与定位。

在 DT 时代中，很多公司也开启了大中台，小前台的战略，即在中台中完成一系列可开放能力的聚合，赋能前端业务，加速迭代开发。工程化是中台化的基石，通过制定标准化的规范、基于元数据的可配置业务流等，完成前后端的业务衔接；而统一的服务中台又是在复杂业务场景下实现微前端/微服务的保障。微服务与微前端，都是希望将某个单一的单体应用，转化为多个可以独立运行、独立开发、独立部署、独立维护的服务或者应用的聚合，从而满足业务快速变化及分布式多团队并行开发的需求。微前端的落地，需要考虑到产品研发与发布的完整生命周期；我们会关注如何保证各个团队的独立开发与灵活的技术栈选配，如何保证代码风格、代码规范的一致性，如何合并多个独立的前端应用，如何在运行时对多个应用进行有效治理，如何保障多应用的体验一致性，如何保障个应用的可测试与可依赖性等方面。

最后，对于个人而言，随着团队技术栈的相对稳定，关注点也会逐步从组件库的建设变化为基础设施的建设，从考虑选择怎样的技术栈到如何在立足某个技术栈更好地服务于业务规划。这个知识爆炸与终身学习/碎片化学习为主的时代，我们要进行更有效地学习，从知识广度，编程能力与知识深度等方面提升自己。

# 前端要凉？微软开源 Sketch2Code，草图秒变代码

作者 微软 ML 博客团队，译者 无明



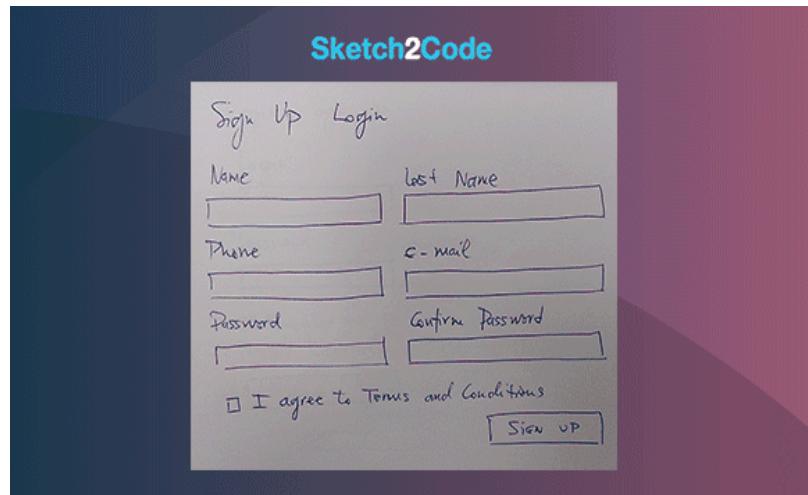
用户界面设计过程涉及大量创造性的迭代工作。这个过程通常从在白板或白纸上画草图开始，设计师和工程师分享他们的想法，尽力表达出潜在的客户场景或工作流程。当他们在某个设计上达成一致之后，通过照片的形式将草图拍下来，然后手动将草图翻译成 HTML 代码。翻译过程需要耗费很多时间和精力，通常会减慢设计过程。

如果可以将白板上手绘的设计立即反映在浏览器中，那会怎样？如果我们能够做到这一点，在设计头脑风暴结束时，我们就可以拥有一个已经由设计师、开发人员甚至客户验证过的现成原型，这将为网站和应用程序开发者省不少时间。现在，微软已经借助 AI 做到了这一点，同时他们还将这个项目在 Github 上开源了。

## Sketch2Code 是什么？

Sketch2Code 是一个基于 Web 的解决方案，使用 AI 将手绘的用户界面草图转换为可用的 HTML 代码。Sketch2Code 由微软和 Kabel、Spike Techniques 合作开发。读者可以在 GitHub 上找到与 Sketch2Code 相关的代码、解决方案开发过程和其他详细信息。

Sketch2Code [项目地址](#)。下图演示了利用 Sketch2Code 将手绘草图转换成代码的操作过程。在微软官方[网站](#)上可以做更多尝试。

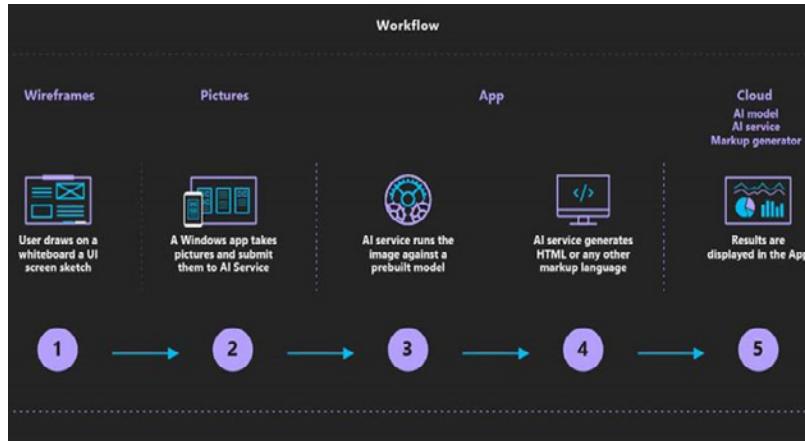


## Sketch2Code 是如何工作的？

让我们来看看使用 Sketch2Code 将手绘草图转换成 HTML 代码的过程：

- 用户将图片上传到网站上。
- 自定义视觉模型预测在图像中出现的 HTML 元素，并将它们的位置标出来。
- 手写文本识别服务读取预测元素中的文本。
- 布局算法根据预测元素的边框空间信息生成网格结构。
- HTML 生成引擎使用上述信息来生成 HTML 代码。

工作流程如下所示：



## Sketch2Code 的架构设计

Sketch2Code 使用了以下组件：

**微软自定义视觉模型（Custom Vision）**：这个模型是基于不同的手绘稿的图象训练得出的，并标记了与常见 HTML 元素（如文本框、按钮、图像等）相关的信息。

**微软计算机视觉服务**：用于识别设计元素中的文本。

**Azure Blob Storage**：保存与 HTML 生成过程的每个步骤相关的信息，包括原始图像、预测结果、布局和分组信息等。

**Azure Function**：它作为后端入口点，通过与其他服务发生交互来协调生成过程。

**Azure Website**：用户界面前端，用户可以在这里上载设计图，并查看生成的 HTML。

以上组件通过如下架构组合在一起：

是不是感觉跃跃欲试？

你可以在这里找到 Sketch2Code 的[开源代码](#)。

也可以在[这里](#)对 Sketch2Code 的实际效果进行验证。

# Java 11 正式发布，新特性解读

作者 杨晓峰



北京时间 9 月 26 日，Oracle 官方宣布 Java 11 正式发布。这是 Java 大版本周期变化后的第一个长期支持版本，非常值得关注。你可以点击[这里](#)下载。

最新发布的 Java11 将带来 ZGC、Http Client 等重要特性，一共包含 17 个 JEP（JDK Enhancement Proposals，JDK 增强提案）。

除了这些重要特性以外，Java 11 还有哪些值得关注的点呢？Java 现在更新那么频繁，是否要考虑升级呢？工程师们还应该继续学，学得动吗？一起来看！

不知不觉 JDK 11 已经发布了，从 9 开始，JDK 进入了让人学不动的更新节奏，对于广大 Java 工程师来说，真是又爱又恨，Java 演进快速意

- 181: Nest-Based Access Control
- 309: Dynamic Class-File Constants
- 315: Improve Aarch64 Intrinsics
- 318: Epsilon: A No-Op Garbage Collector
- 320: Remove the Java EE and CORBA Modules
- 321: HTTP Client (Standard)
- 323: Local-Variable Syntax for Lambda Parameters
- 324: Key Agreement with Curve25519 and Curve448
- 327: Unicode 10
- 328: Flight Recorder
- 329: ChaCha20 and Poly1305 Cryptographic Algorithms
- 330: Launch Single-File Source-Code Programs
- 331: Low-Overhead Heap Profiling
- 332: Transport Layer Security (TLS) 1.3
- 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
- 335: Deprecate the Nashorn JavaScript Engine
- 336: Deprecate the Pack200 Tools and API

意味着它仍将能够保持企业核心技术平台的地位，我们对 Java 的投入和饭碗是安全的，但同时也带来了学习、选择的困惑。

所以，今天我们不准备做个流水账的介绍，一起来看看工程师甚至是 IT 决策者最关心的问题：

- JDK 更新如此频繁，我是否要考虑升级？新的发布模式下，企业的 IT 策略需要做出什么样的调整？
- 除了要酷，JDK 11，或者说最近的 JDK 版本，有什么真正值得生产环境中应用的特性？工程师要跟进吗？

对于第一个问题，本人十分确信 JDK 11 将是一个企业不可忽视的版本。

首先，从时间节点来看，JDK 11 的发布正好处在 JDK 8 免费更新到期的前夕，同时 JDK 9、10 也陆续成为“历史版本”，请看下面的 Oracle JDK 支持路线图。

JDK 更新很重要吗？非常重要，在过去的很多年中，Oracle 和 OpenJDK 社区提供了接近免费的午餐，导致人们忽略了其背后的海量工作和价值，这其中包括但不限于：

| Java SE Public Updates            |                |                                    |                                       |                                     |
|-----------------------------------|----------------|------------------------------------|---------------------------------------|-------------------------------------|
| Release                           | GA Date        | End of Public Updates Notification | Commercial User End of Public Updates | Personal User End of Public Updates |
| 7                                 | July 2011      | March 2014                         |                                       | April 2015                          |
| 8                                 | March 2014     | September 2017                     | January 2019****                      | December 2020****                   |
| 9 (non-LTS)                       | September 2017 | September 2017                     |                                       | March 2018                          |
| 10 (18.3 <sup>4</sup> ) (non-LTS) | March 2018     | March 2018                         |                                       | September 2018                      |

最新的安全更新，如，安全协议等基础设施的升级和维护，安全漏洞的及时修补，这是 Java 成为企业核心设施的基础之一。安全专家清楚，即使开发后台服务，而不是前端可直接接触，编程语言的安全性仍然是重中之重。

大量的新特性、Bug 修复，例如，容器环境支持，GC 等基础领域的增强。很多生产开发中的 Hack，其实升级 JDK 就能解决了。

不断改进的 JVM，提供接近零成本的性能优化

“Easy is cheap”？Java 的进步虽然“容易”获得，但莫忽略其价值，这得益于厂商和 OpenJDK 社区背后的默默付出。

第二，JDK 11 是一个长期支持版本（LTS, Long-Term-Support）。

对于企业来说，选择 11 将意味着长期的、可靠的、可预测的技术路线图。有人说，那是对于付费订阅客户，不订阅是不是就不用考虑 11 了呢？

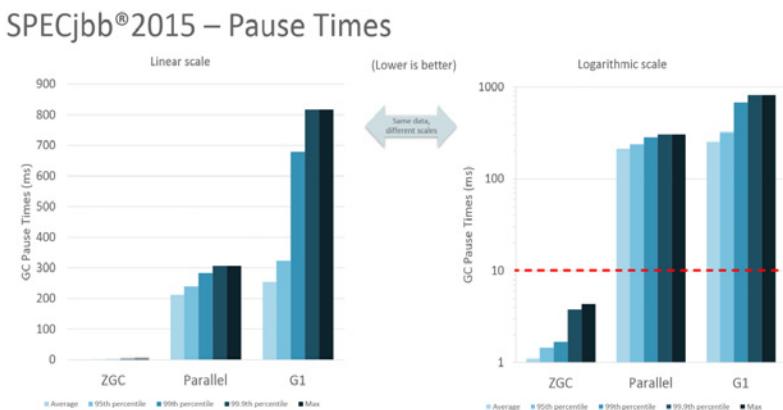
不，请放心，11 确定将得到 OpenJDK 社区的长期支持，目前 Oracle 提供了 OpenJDK build，虽然后续计划未定，但是承诺“至少维护到明年”。即使是停止发布后续 JDK11 更新，Andrew Haley 等社区专家也已经明确保证，会组建并领导“JDK-11-updates”项目，并且保证：

“please let me assure you of one thing: whether by Oracle or Red Hat or someone else, JDK LTS releases will continue to be supported. We all have a lot invested in Java, and we won't let it fall.”

所以，LTS 版本将是企业 IT 决策者可以放心选择的版本。

回到第二个问题，我们一起来看看 JDK 11 的有哪些能力上的突破，能够让我们觉得升级到 JDK 11 是超值的。

从 JVM GC 的角度，JDK 11 引入了两种新的 GC，其中包括也许是划时代意义的 ZGC，虽然其目前还是实验特性，但是从能力上来看，这是 OpenJDK 的一个巨大突破，为特定生产环境的苛刻需求提供了一个可能的选择。例如，对部分企业核心存储等产品，如果能够保证不超过 10ms 的 GC 暂停，可靠性会上一个大的台阶，这是过去我们进行 GC 调优几乎做不到的，是能与不能的问题。相信你对下面的数据已经不再陌生。



关于 ZGC 特性的解读已经非常多，本文不再重复。我在这里主要介绍那些不起眼，但更具生产系统价值的部分。例如，对于 G1 GC，相比于 JDK 8，升级到 JDK 11 即可免费享受到：并行的 Full GC，快速的 CardTable 扫描，自适应的堆占用比例调整（IHOP），在并发标记阶段的类型卸载等等。这些都是针对 G1 的不断增强，其中串行 Full GC 等甚至是曾经被广泛诟病的短板，你会发现 GC 配置和调优在 JDK11 中越来越方便。

云计算时代的监控、诊断和 Profiling 能力，这是个人认为比 ZGC 更具生产实践意义的特性。

不知道你有没有注意到，不知不觉中 Java 的应用场景发生了天翻地覆

的变化，从单机长时间运行的 Java 应用，发展成为分布式、大的单体应用或小的 function、瞬时或长时间运行等，应用场景非常复杂。

我们还用什么工具诊断 Java 应用？JConsole，JProfiler，还是“自研”的各种工具？目前的工具大多是从对单个 Java 应用的诊断视角出发，试想如果我们的集群中有几百、数千台机器或容器，每台机器有几个或者几十个 Java 进程，那么：

怎么在不干扰生产系统的情况下，高效地跟踪海量的 Java 进程，准确定位可以优化的性能空间？

如何保证“随机出现”的 JVM 问题，不需要进行额外的、令人头痛的“重现”？

JDK 11 悄悄地为我们提供了更加强大的基础能力，主要是两部分：

JEP 328: Flight Recorder (JFR) 是 Oracle 刚刚开源的强大特性。我们知道在生产系统进行不同角度的 Profiling，有各种工具、框架，但是能力范围、可靠性、开销等，大都差强人意，要么能力不全面，要么开销太大，甚至不可靠可能导致 Java 应用进程宕机。

而 JFR 是一套集成进入 JDK、JVM 内部的事件机制框架，通过良好架构和设计的框架，硬件层面的极致优化，生产环境的广泛验证，它可以做到极致的可靠和低开销。在 SPECjbb2015 等基准测试中，JFR 的性能开销最大不超过 1%，所以，工程师可以基本没有心理负担地在大规模分布式的生产系统使用，这意味着，我们既可以随时主动开启 JFR 进行特



定诊断，也可以让系统长期运行 JFR，用以在复杂环境中进行“After-the-fact”分析。还需要苦恼重现随机问题吗？JFR 让问题简化了很多哦。

在保证低开销的基础上，JFR 提供的能力也令人眼前一亮，例如：

我们无需 BCI 就可以进行 Object Allocation Profiling，终于不用担心 BTrace 之类把进程搞挂了。

- 对锁竞争、阻塞、延迟，JVM GC、SafePoint 等领域，进行非常细粒度分析。
- 甚至深入 JIT Compiler 内部，全面把握热点方法、内联、逆优化等等。
- JFR 提供了标准的 Java、C++ 等扩展 API，可以与各种层面的应用进行定制、集成，为复杂的企业应用栈或者复杂的分布式应用，提供 All-in-One 解决方案。

而这一切都是内建在 JDK 和 JVM 内部的，并不需要额外的依赖，开箱即用。

另一方面，就是同样不起眼的 JEP 331: Low-Overhead Heap Profiling。我们知道，高效地了解在 Java 堆上都进行了哪些对象分配，是诊断内存问题的基本出发点之一。JEP 331 来源于 Google 等业界前沿厂商的一线实践，通过获取对象分配的 Call-site，为 JDK 补足了对象分配诊断方面的一些短板，工程师可以通过 JVMTI 使用这个能力增强自身的工具。

从 Java 类库发展的角度来看，JDK 11 最大的进步也是两个方面：

首先，是难得的现代 HTTP/2 Client API，Java 工程师终于可以摆脱老旧的 HttpURLConnection 了。新的 HTTP API 提供了对 HTTP/2 等业界前沿标准的支持，精简而又友好的 API 接口，与主流开源 API（如，Apache HttpClient，Jetty，OkHttp 等）对等甚至更高的性能。与此同时它是 JDK 在 Reactive-Stream 方面的第一个生产实践，广泛使用了 Java Flow API 等，终于让 Java 标准 HTTP 类库在扩展能力等方面，满足了现代互联网的需求。

第二，就是安全类库、标准等方面的大范围升级，其中特别是 JEP

332: Transport Layer Security (TLS) 1.3，除了在安全领域的重要价值，它还是中国安全专家范学雷所领导的 JDK 项目，完全不同于以往的修修补补，是个非常大规模的工程。

除此之外，JDK 还在逐渐进行瘦身工作，或者偿还 JVM、Java 规范等历史欠账，例如

- 181: [Nest-Based Access Control](#)
- 309: [Dynamic Class-File Constants](#)
- 320: [Remove the Java EE and CORBA Modules](#)
- 330: [Launch Single-File Source-Code Programs](#)
- 335: [Deprecate the Nashorn JavaScript Engine](#)
- 336: [Deprecate the Pack200 Tools and API](#)

其中最值得关注的是 JEP 335，它进一步明确了 Graal 很有可能将成为 JVM 向前演进的核心选择，Java-on-Java 正在一步步的成为现实。

JDK 11 还有什么遗憾吗？很多 Valhalla、Loom、Panama 等项目中继续补齐的短板，如协程、Value Type 等，目前还是可望而不可即，也许令人欣慰地是，我们能看到 Java 能够正视自身存在的不足，不断飞速发展。

从 1995 年第一个版本发布到现在，Java 语言已经在跌宕起伏中走过了 23 年。这 23 年，既有 Java 连续霸榜多年的风头无两，也有近两年 Java 不可忽视的颓势。Java 是最好的语言吗？不是，每个领域都有更合适的编程语言，没有无所不能的存在。

Java 语言到底有什么优势可以占据排行榜第一的位置呢？

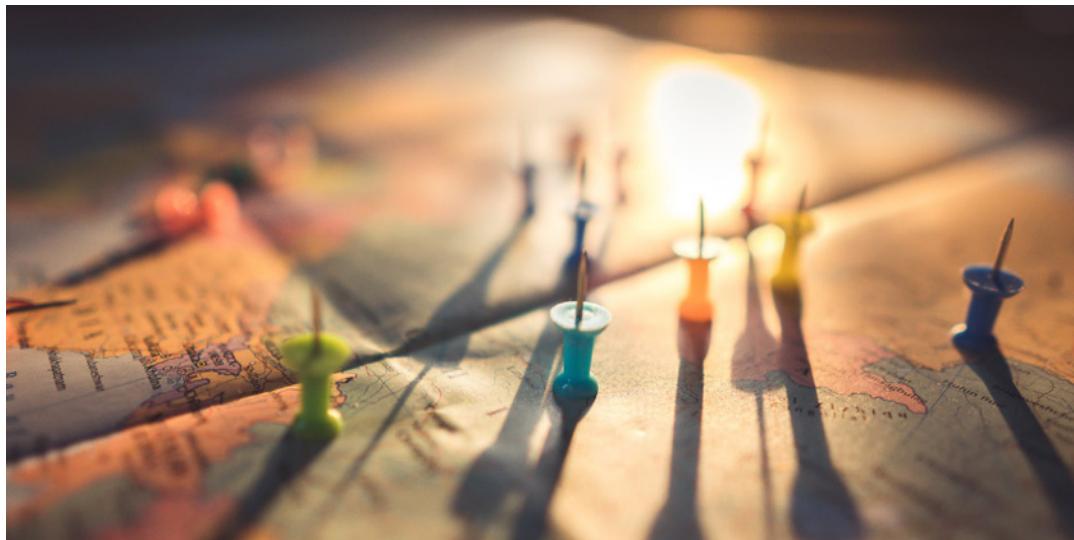
其一，语法比较简单，学过计算机编程的开发者都能快速上手，JVM 也为开发者屏蔽了大量复杂的细节。其二，能力过硬，在若干个领域的竞争力都非常强，比如服务端编程，高性能网络程序，企业软件事务处理，大数据，分布式计算，移动、嵌入终端应用开发等等。最重要的一点是其吸收了业界领先的工程实践，构建从嵌入式设备到超大规模软件系统的能力，充分得到了实践验证。所有这些都使得 Java 成为企业软件公司的

首选，也得到很多互联网公司的青睐。时移世易，Java 正在也必须改变。

Java 改为每半年发版一次以后，在对合并关键特性、快速得到开发者反馈等方面，做得越来越好，我们明显能够看到各厂商和社区对 Java 投入的提高。最新发布的 JDK 11 虽然谈不上划时代的进步，但一定是 JDK 发展历程中的一个重要版本，升级 JDK 就可以获得的性能等各种提高，基础能力的全面进步和突破，这一切无不说明，是时候开始评估并开始计划升级到 JDK 11 了。

# 秒杀 Redis 的 KVS 数据库上云了！伯克利重磅开源 Anna 1.0

作者 Anna 团队，译者 无明



天下武功，唯快不破。今年3月份，伯克利 RISE 实验室推出了最新的键值存储数据库 Anna，提供了惊人的存取速度、超强的伸缩性和史无前例的一致性保证。Anna论文也被评为"Best of ICDE 2018"，其加长版即将应邀发表在IEEE TKDE期刊上。Anna一经推出即在业界引发热烈讨论（报道回顾 "[伯克利推出世界最快的KVS数据库Anna：秒杀Redis和Cassandra](#)"）。不少读者关心它何时开源、后续有什么新的进展。今天，AI前线给大家带来了Anna的最新消息，过去这半年里，伯克利RISE实验室对Anna的设计进行了重大变更，新版本的Anna能够更好地在云端扩展。实验表明，无论是在性能还是成本效益方面，Anna都表现突出，它明显优于AWS ElastiCache的memcached以及较早之前的Masstree，也

比AWS DynamoDB更具成本优势。与此同时，Anna所有源码也正式登陆Github，开放给所有开发者。

## 背景

在之前的一篇博文中，我们介绍了Anna系统，它使用了一个核心对应一个线程的无共享线程架构，通过避免线程间的协调来实现闪电般的速度。Anna还使用晶格组合来实现多样的无协调一致性级别。第一个版本的Anna吊打现有的内存KV存储系统：它的性能优于Masstree 700倍，优于TBB 800倍。你可以重温之前的博文（<https://databeta.wordpress.com/2018/03/09/anna-kvs/>），或者阅读完整的论文（[http://db.cs.berkeley.edu/jmh/papers/anna\\_ieee18.pdf](http://db.cs.berkeley.edu/jmh/papers/anna_ieee18.pdf)）。我们将第一个Anna版本称为“Anna v0”。在这篇文章中，我们介绍如何将这个最快的KV存储系统变得极具成本效益和适应性。

现如今，公共基础设施云用户可选择的存储系统真是太多了。AWS提供两种对象存储服务（S3和Glacier）和两种文件系统服务（EBS和EFS），另外还有七种不同的数据库服务，从关系数据库到NoSQL键值存储。真是花样繁多，令人眼花缭乱，用户自然会问，哪个服务才适合他们。最直接（然而并不乐观）的答案是，把它们全都用起来就对了。

这些存储服务都提供了非常有限的成本与性能之间的权衡。例如，AWS ElastiCache速度很快，但很贵，而AWS Glacier虽然便宜，但速度较慢。因此，用户们面对一个窘境：他们必须要么放弃节约成本的目标，大规模部署高性能存储集群，要么放弃性能，利用低成本的系统例如DynamoDB或者S3。

更糟糕的是，大多数实际应用展现出偏斜的数据访问模式。频繁被访问的数据是“热”数据，其他则为“冷”数据，而这些服务要么是专门为“热数据”而设计，要么专门为“冷数据”而设计。因此，不想在性能或成本上妥协的用户必须手动将这些解决方案拼凑在一起，跟踪服务间的数据和请求，以及管理不同的API，并做出一致性保证。

更糟糕的是，高性能的云存储产品不具备弹性：向集群添加资源或从集群中移除资源都需要人工干预。这意味着云开发者们设计并实现自定义解决方案来监控工作负载变化、修改资源分配以及在存储引擎之间移动数据。

这是非常糟糕的。应用程序开发人员不断被迫重新发明轮子，而不是把精力放在他们最关心的指标上：性能和成本。我们想要改变这种现状。

## Anna v1

我们借助Anna v0这个内存存储引擎来解决上述的云存储问题。我们的目标是将最快的Anna同时发展成为最具适应性和成本效益的KV存储系统。我们向Anna中添加了3个关键的机制：垂直分层、水平弹性和选择性复制。

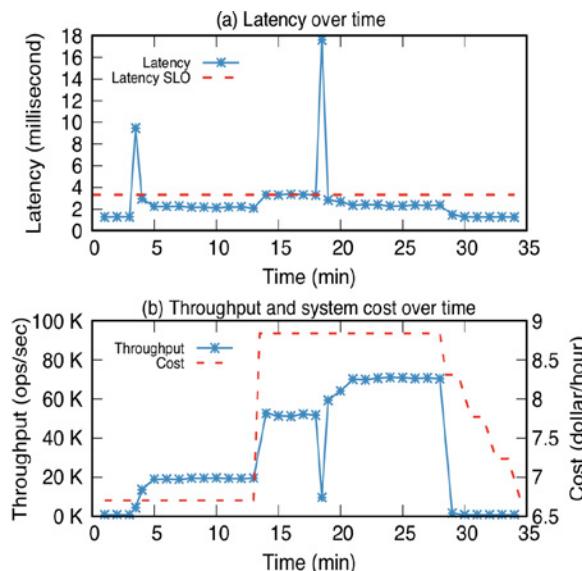
Anna v1的核心组件是监控系统和策略引擎，可实现工作负载的响应性和适应性。为了满足用户定义的性能目标（请求延迟）和成本，监控服务对工作负载变化进行监控和调整。存储服务器会收集请求和数据的统计信息。监视系统定期搜索和处理这些数据，策略引擎基于这些统计信息执行上述的三个操作。操作的触发规则很简单：

- 弹性：为了让系统适应变化的工作负载，系统需要能够根据工作量自动扩展和收缩。当一个层达到计算或存储容量最大值时添加节点，当某个节点明显得不到充分利用时将其移除。
- 选择性复制：在实际工作环境下，访问量很大的key需要被广泛的复制到多个节点和CPU核从而提高性能。Anna v0启用了key的多主复制，但在复制所有key时提供了一个固定的复制因子。正如读者们想象的，这是一个成本很大的设计。在Anna v1中，监控服务仅选择性的复制访问次数高的“热”key而不复制“冷”key，从而减小存储成本。
- 升级和降级：与传统的内存层次结构中一样，热数据应该保存在更快的存储介质中提高访问效率，冷数据应该保存在较慢的存储

介质中节约成本。我们的监测服务会根据数据的热度变化自动将它们移至合适的存储介质。

为了实现这些机制，我们不得不对Anna的设计做出两个重大变更。首先，我们让存储引擎支持多种存储介质——目前是内存和闪存。与传统的存储层次结构类似，这些存储层的成本与性能权衡是不一样的。我们还实现了一个路由服务，它将用户请求发送到目标层的服务器上。无论数据存储在什么地方，都可以为用户提供统一的API。这些层都从第一版Anna继承了同等丰富的一致性模型，因此开发者可以灵活挑选并自定义合适的一致性模型。

我们的实验表明，无论是在性能还是成本效益方面，Anna都达到了令人印象深刻的水平。在同一成本下，Anna提供优于AWS ElastiCache 8倍的吞吐量和优于DynamoDB 355倍的吞吐量。Anna还能够通过添加节点和恰到好处的数据复制来应对工作负载的变化：



这篇文章只提供了Anna的设计概述，如果你有兴趣了解更多，可以在[这里](#)和[这里](#)找到完整的论文和代码。这个项目的进展让我们很满意，我们也很乐意收到你的反馈。后续，我们会有更多的计划，将Anna的高性能和灵活性拓展到其他的系统中，敬请关注！

# 架构师如何判断技术演进的方向？

作者 李运华



互联网的出现不但改变了普通人的生活方式，同时也促进了技术圈的快速发展和开放。在开源和分享两股力量的推动下，最近10多年的技术发展可以说是目不暇接，你方唱罢我登场，大的方面有大数据、云计算、人工智能等，细分的领域有NoSQL、Node.js、Docker容器化等。各个大公司也乐于将自己的技术分享出来，以此来提升自己的技术影响力，打造圈内技术口碑，从而形成强大的人才吸引力，典型的有，Google的大数据论文、淘宝的全链路压测、微信的红包高并发技术等。

对于技术人员来说，技术的快速发展当然是一件大好事，毕竟这意味着技术百宝箱中又多了更多的可选工具，同时也可以通过学习业界先进的

技术来提升自己的技术实力。但对于架构师来说，除了这些好处，却也多了“甜蜜的烦恼”：面对层出不穷的新技术，我们应该采取什么样的策略？

架构师可能经常会面临下面这些诱惑或者挑战：

- 现在Docker虚拟化技术很流行，我们要不要引进，引入Docker后可以每年节省几十万元的硬件成本呢？
- 竞争对手用了阿里的云计算技术，听说因为上了云，业务增长了好几倍呢，我们是否也应该尽快上云啊？
- 我们的技术和业界顶尖公司（例如，淘宝、微信）差距很大，应该投入人力和时间追上去，不然招聘的时候没有技术影响力！
- 公司的技术发展现在已经比较成熟了，程序员都觉得在公司学不到东西，我们可以尝试引入Golang来给大家一个学习新技术的机会。

类似的问题还有很多，本质上都可以归纳总结为一个问题：我们应该如何判断技术演进的方向？

关于这个问题的答案，基本上可以分为几个典型的派别：

### 1. 潮流派

潮流派的典型特征就是对于新技术特别热衷，紧跟技术潮流，当有新的技术出现时，迫切想将新的技术应用到自己的产品中。

例如：

- NoSQL很火，咱们要大规模地切换为NoSQL。
- 大数据好牛呀，将我们的MySQL切换为Hadoop吧。
- Node.js使得JavaScript统一前后端，这样非常有助于开展工作。

### 2. 保守派

保守派的典型特征和潮流派正好相反，对于新技术抱有很强的戒备心，稳定压倒一切，已经掌握了某种技术，就一直用这种技术打天下。就像有句俗语说的，“如果你手里有一把锤子，那么所有的问题都变成了钉子”，保守派就是拿着一把锤子解决所有的问题。

例如：

- MySQL咱们用了这么久了，很熟悉了，业务用MySQL，数据分析也用MySQL，报表还用MySQL吧。
- Java语言我们都很熟，业务用Java，工具用Java，平台也用Java。

### 3. 跟风派

跟风派与潮流派不同，这里的跟风派不是指跟着技术潮流，而是指跟着竞争对手的步子走。

简单来说，判断技术的发展就看竞争对手，竞争对手用了咱们就用，竞争对手没用咱们就等等看。

例如：

- 这项技术腾讯用了吗？腾讯用了我们就用。
- 阿里用了Hadoop，他们都在用，肯定是好东西，咱们也要尽快用起来，以提高咱们的竞争力。
- Google都用了Docker，咱们也用吧。

不同派别的不同做法本质上是价值观的不同：潮流派的价值观是新技术肯定能带来很大收益；稳定派的价值观是稳定压倒一切；跟风派的价值观是别人用了我就用。这些价值观本身都有一定的道理，但如果不考虑实际情况生搬硬套，就会出现“橘生淮南则为橘，生于淮北则为枳”的情况。

下面我们来看一下不同的派别可能存在的问题。

#### 1. 潮流派

首先，新技术需要时间成熟，如果刚出来就用，此时新技术还不怎么成熟，实际应用中很可能遇到各种“坑”，自己成了实验小白鼠。

其次，新技术需要学习，需要花费一定的时间去掌握，这个也是较大的成本；如果等到掌握了技术后又发现不适用，则是一种较大的人力浪费。

#### 2. 保守派

保守派的主要问题是不能享受新技术带来的收益，因为新技术很多都

是为了解决以前技术存在的固有缺陷。就像汽车取代马车一样，不是量变而是质变，带来的收益不是线性变化的，而是爆发式变化的。如果无视技术的发展，形象一点说就是有了拖拉机，你还偏偏要用牛车。

### 3. 跟风派

可能很多人都会认为，跟风派与“潮流派”和“保守派”相比，是最有效的策略，既不会承担“潮流派”的风险，也不会遭受“保守派”的损失，花费的资源也少，简直就是一举多得。

看起来很美妙，但跟风派最大的问题在于如果没有风可跟的时候怎么办。如果你是领头羊怎么办，其他人都准备跟你的风呢？另外一种情况就是竞争对手的这些信息并不那么容易获取，即使获取到了一些信息，大部分也是不全面的，一不小心可能就变成邯郸学步了。

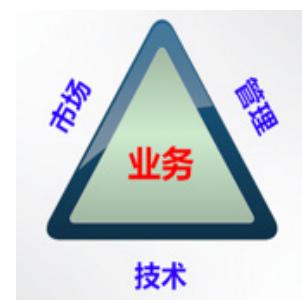
即使有风可跟，其实也存在问题。有时候适用于竞争对手的技术，并不一定适用于自己，盲目模仿可能带来相反的效果。

既然潮流派、保守派、跟风派都存在这样或者那样的问题，那架构师究竟如何判断技术演进的方向呢？

## 技术演进的动力

这个问题之所以让人困惑，关键的原因还是在于不管是潮流派、保守派，还是跟风派，都是站在技术本身的角度来考虑问题的，正所谓“不识庐山真面，只缘身在此山中”。因此，要想看到“庐山真面目”，只有跳出技术的范畴，从一个更广更高的角度来考虑这个问题，这个角度就是企业的业务发展。

无论是代表新兴技术的互联网企业，还是代表传统技术的制造业；无论是通信行业，还是金融行业的发展，归根到底就是业务的发展。而影响一个企业业务的发展主要有3个因素：市场、技术、管理，这三者构成支撑业务发展的铁三角，任何一个因素的不足，都可能导致企业的业务停滞不前。



在这个铁三角中，业务处于三角形的中心，毫不夸张地说，市场、技术、管理都是为了支撑企业业务的发展。在专栏里，我主要探讨“技术”和“业务”之间的关系和互相如何影响。

我们可以简单地将企业的业务分为两类：一类是产品类，一类是服务类。

产品类：360的杀毒软件、苹果的iPhone、UC的浏览器等都属于这个范畴，这些产品本质上和传统的制造业产品类似，都是具备了某种“功能”，单个用户通过购买或者免费使用这些产品来完成自己相关的某些任务，用户对这些产品是独占的。

服务类：百度的搜索、淘宝的购物、新浪的微博、腾讯的IM等都属于这个范畴，大量用户使用这些服务来完成需要与其他人交互的任务，单个用户“使用”但不“独占”某个服务。事实上，服务的用户越多，服务的价值就越大。服务类的业务符合互联网的特征和本质：“互联”+“网”。

对于产品类业务，答案看起来很明显：技术创新推动业务发展！

例如：

- 苹果开发智能手机，将诺基亚推下王座，自己成为全球手机行业的新王者。
- 2G时代，UC浏览器独创的云端架构，很好地解决了上网慢的问题；智能机时代，UC浏览器又自主研发全新的U3内核，兼顾高速、安全、智能及可扩展性，这些技术创新是UC浏览器成为了全球最大的第三方手机浏览器最强有力的推动力。

为何对于产品类的业务，技术创新能够推动业务发展呢？答案在于用户选择一个产品的根本驱动力在于产品的功能是否能够更好地帮助自己完成任务。用户会自然而然地选择那些功能更加强大、性能更加先进、体验更加顺畅、外观更加漂亮的产品，而功能、性能、体验、外观等都需要强大的技术支撑。例如，iPhone手机的多点触摸操作、UC浏览器的U3内核等。

对于“服务”类的业务，答案和产品类业务正好相反：业务发展推动技术的发展！

为什么会出现截然相反的差别呢？主要原因是用户选择服务的根本驱动力与选择产品不同。用户选择一个产品的根本驱动力是其“功能”，而用户选择一个服务的根本驱动力不是功能，而是“规模”。

例如，选择UC浏览器还是选择QQ浏览器，更多的人是根据个人喜好和体验来决定的；而选择微信还是Whatsapp，就不是根据它们之间的功能差异来选择的，而是根据其规模来选择的，就像我更喜欢Whatsapp的简洁，但我的朋友和周边的人都用微信，那我也不得不用微信。

当“规模”成为业务的决定因素后，服务模式的创新就成为了业务发展的核心驱动力，而产品只是为了完成服务而提供给用户使用的一个载体。以淘宝为例，淘宝提供的“网络购物”是一种新的服务，这种业务与传统的到实体店购物是完全不同的，而为了完成这种业务，需要“淘宝网”“支付宝”“一淘”和“菜鸟物流”等多个产品。随便一个软件公司，如果只是模仿开发出类似的产品，只要愿意投入，半年时间就可以将这些产品全部开发出来。但是这样做并没有意义，因为用户选择的是淘宝的整套网络购物服务，并且这个服务已经具备了一定的规模，其他公司不具备这种同等规模服务的能力。即使开发出完全一样的产品，用户也不会因为产品功能更加强大而选择新的类似产品。

以微信为例，同样可以得出类似结论。假如我们进行技术创新，开发一个耗电量只有微信的1/10，用户体验比微信好10倍的产品，你觉得现在的微信用户都会抛弃微信，而转投我们的这个产品吗？我相信绝大部分人都不会，因为微信不是一个互联网产品，而是一个互联网服务，你一个人换到其他类微信类产品是没有意义的。

因此，服务类的业务发展路径是这样的：提出一种创新的服务模式→吸引了一批用户→业务开始发展→吸引了更多用户→服务模式不断完善和创新→吸引越来越多的用户，如此循环往复。在这个发展路径中，技术并没有成为业务发展的驱动力，反过来由于用户规模的不断扩展，业务的不

断创新和改进，对技术会提出越来越高的要求，因此是业务驱动了技术发展。

其实回到产品类业务，如果我们将观察的时间拉长来看，即使是产品类业务，在技术创新开创了一个新的业务后，后续的业务发展也会反向推动技术的发展。例如，第一代iPhone缺少对3G的支持，且只能通过Web发布应用程序，第二代iPhone才开始支持3G，并且内置GPS；UC浏览器随着功能越来越强大，原有的技术无法满足业务发展的需求，浏览器的架构需要进行更新，先后经过UC浏览器7.0版本、8.0版本、9.0版本等几个技术差异很大的版本。

综合这些分析，除非是开创新的技术能够推动或者创造一种新的业务，其他情况下，都是业务的发展推动了技术的发展。

## 技术演进的模式

明确了技术发展主要的驱动力是业务发展后，我们来看看业务发展究竟是如何驱动技术发展的。

业务模式千差万别，有互联网的业务（淘宝、微信等），有金融的业务（中国平安、招商银行等），有传统企业的业务（各色ERP对应的业务）等，但无论什么模式的业务，如果业务的发展需要技术同步发展进行支撑，无一例外是因为业务“复杂度”的上升，导致原有的技术无法支撑。

按照专栏前面所介绍的复杂度分类，复杂度要么来源于功能不断叠加，要么来源于规模扩大，从而对性能和可用性有了更高的要求。既然如此，判断到底是什么复杂度发生了变化就显得至关重要了。是任何时候都要同时考虑功能复杂度和规模复杂度吗？还是有时候考虑功能复杂度，有时候考虑规模复杂度？还是随机挑一个复杂度的问题解决就可以了？

所以，对于架构师来说，判断业务当前和接下来一段时间的主要复杂度是什么就非常关键。判断不准确就会导致投入大量的人力和时间做了对业务没有作用的事情，判断准确就能够做到技术推动业务更加快速发展。

那架构师具体应该按照什么标准来判断呢？

答案就是基于业务发展阶段进行判断，这也是为什么架构师必须具备业务理解能力的原因。不同的行业业务发展路径、轨迹、模式不一样，架构师必须能够基于行业发展和企业自身情况做出准确判断。

假设你是一个银行IT系统的架构师：

- 90年代主要的业务复杂度可能就是银行业务范围逐渐扩大，功能越来越复杂，导致内部系统数量越来越多，单个系统功能越来越复杂。
- 2004年以后主要的复杂度就是银行业务从柜台转向网上银行，网上银行的稳定性、安全性、易用性是主要的复杂度，这些复杂度主要由银行IT系统自己解决。
- 2009年以后主要的复杂度又变化为移动支付复杂度，尤其是“双11”这种海量支付请求的情况下，高性能、稳定性、安全性是主要的复杂度，而这些复杂度需要银行和移动支付服务商（支付宝、微信）等一起解决。

而如果你是淘宝这种互联网业务的架构师，业务发展又会是另外一种模式：

- 2003年，业务刚刚创立，主要的复杂度体现为如何才能快速开发各种需求，淘宝团队采取的是买了一个PHP写的系统来改。
- 2004年，上线后业务发展迅速，用户请求数量大大增加，主要的复杂度体现为如何才能保证系统的性能，淘宝的团队采取的是用Oracle取代MySQL。
- 用户数量再次增加，主要的复杂度还是性能和稳定性，淘宝的团队采取的是Java替换PHP。
- 2005年，用户数量继续增加，主要的复杂度体现为单一的Oracle库已经无法满足性能要求，于是进行了分库分表、读写分离、缓存等优化。
- 2008年，淘宝的商品数量在1亿以上，PV2.5亿以上，主要的复杂

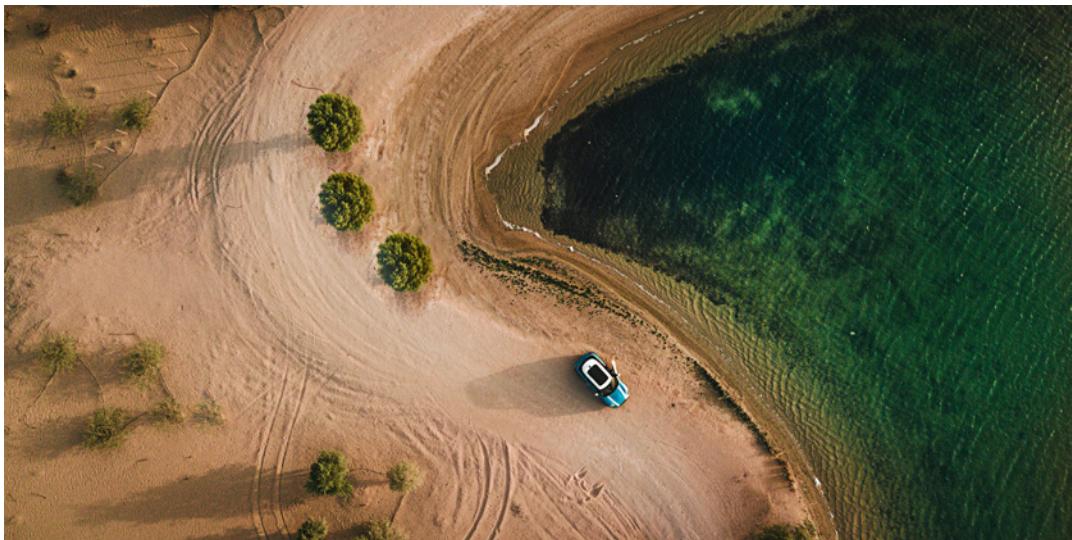
度又变成了系统内部耦合，交易和商品耦合在一起，支付的时候又和支付宝强耦合，整个系统逻辑复杂，功能之间跳来跳去，用户体验也不好。淘宝的团队采取的是系统解耦，将交易中心、类目管理、用户中心从原来大一统的系统里面拆分出来。

## 小结

今天我为你讲了架构师该如何判断技术演进的方向，希望对你有所帮助。最后留一道思考题给你吧，如果业界已经有了一个明显的参照对象（例如电商企业可以参考淘宝），那架构师是否还需要按照步骤逐步演进，还是直接将架构一步到位设计好？

# 后 Kubernetes 时代的微服务

作者 Bilgin Ibryam , 译者 盖磊



微服务的关注热度起源于一大堆极端的想法，涉及组织的结构、团队的规模、服务的规模、重写和抛出服务而不是修复、避免单元测试等。依我的经验看，其中大部分想法已被证明是错误的、不实用的，或者至少在一般情况下是不适用的。当前能残存下来的微服务原则和实践，大部分是非常通用和宽松定义的。虽然它们适合未来许多年内的发展，但在实践中并没有多大的意义。

早在Kubernetes横空出世的几年前，微服务理念就得到了采用，目前，它仍然是一种最流行的分布式系统架构风格。Kubernetes和云原生运动已大规模地重定义了应用设计和开发的一些方面。在本文中，我试图提

出一些原始的微服务理念。我将指出，这些理念在后Kubernetes时代已不再像以前那样强大。

## 服务不仅应可观测，而且应是自动化的

可观测性(Observability) 是微服务自一开始就提出的一个基本原则。可观测性虽然适用于一般的分布式系统，但是当前，尤其是在Kubernetes上，可观测性的主要涉及平台层的开箱即用，例如进程运行状况检查、CPU和内存消耗等。应用能以JSON格式登录控制台，这就是可观测性的最低要求。此外，平台应可以在无需过多服务层开发的情况下，实现跟踪资源的消耗、开展请求追踪、收集全部类型的指标、计算错误率等。

在云原生平台上，服务仅具备可观测性是不够的。更基本的先决条件是使用检查健康、响应信号、声明资源消耗等手段实现微服务的自动化。任何应用都可以置于容器中并运行。但是要创建一个可通过云原生平台自动化和协调编排容器的应用，则需要遵循一定的规则。遵循这些原则和模式，可确保所生成的容器作为云本地成员在大多数容器编排引擎中表现为优秀，并支持对容器进行自动化的调度、扩展和监视。

我们希望平台不仅可观测服务中发生的情况，而且希望平台能检测到异常，并按照声明情况做出协调。纠正措施可以是通过停止引导流量到服务实例、重新启动、向上/向下扩展，也可以是将服务迁移到另一台健康的主机、重试失败的请求或是其它一些操作。如果服务实现了自动化，那么所有这些纠正措施都会自动做出，我们只需要描述所需的状态，而不是去观测并做出响应。服务应该是可观测的，但也应在无需人工干预的情况下由平台实现问题整改。

## 具备正确职责的智能平台和智能设备

在从SOA转向微服务的过程中，在服务交互上发生的另一个根本转变就是“智能端点哑管道”（smart endpoints and dumb pipes）这一[理念](#)。在微服务领域，服务不依赖于所具有的集中式智能路由层，而是依赖于具有

某些平台级功能的智能端点。服务的实现是通过在每个微服务中嵌入传统ESB的部分功能，并转为使用不具有业务逻辑元素的一些轻量级协议。

这仍然是一种惯常采用的方法，即在不可靠的网络层（使用诸如[Hystrix](#)之类的库）实现服务交互，但在当前的后Kubernetes时代，服务交互已完全被服务网格（[Service Mesh](#)）技术取代。服务网格吸引人之处在于，它甚至要比传统的ESB更智能。网格可以执行动态路由、服务发现、基于延迟的负载平衡、响应类型、指标和分布式跟踪、重试、超时，以及我们所能想到的所有特性。

与ESB的不同，服务网格只有一个集中路由层，每个微服务通常都具有自己的路由器，即一个使用额外中央管理层执行代理逻辑的“跨斗模式容器”（Sidecar Container）。更重要的是，管道（即平台和服务网格）中并不维持任何业务逻辑。管道完全聚焦于基础架构问题，而让服务聚焦于业务逻辑。下图表示了为适应云环境的动态和不可靠特性，ESB和微服务在认知上的演变情况。

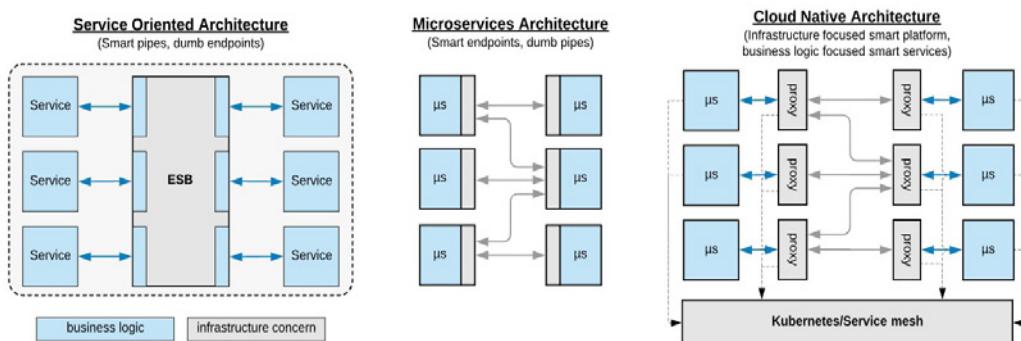


图 从SOA到MSA和CNA

如果查看服务的其他一些方面，我们就会注意到云原生不仅影响了端点和服务交互。Kubernetes平台（及其所有附加技术）还负责资源的管理、调度、部署、配置管理、扩展和服务交互等。与其再次称之为“智能代理哑管道”，我认为更好的描述应是一种具备正确职责的智能平台和智能服务。它不仅是与端点相关，而且也是一个完整的平台，实现主要聚焦于业务功能的服务在所有基础架构上的自动化。

## 设计不应针对“故障”，而应针对“恢复”

毫无疑问，要在基础架构和网络本身并非可靠的云原生环境中运行微服务，我们必须针对故障做出设计。但是越来越多的故障是由平台检测并处理的，而人们对如何从微服务中捕获故障的考虑较少。相反，我们应通过考虑从多个维度实现幂等性，设计我们的恢复服务。

容器技术、容器编排和服务网格（service mesh）可以检测许多故障，并从中进行恢复。例如无限循环（分配CPU份额）、内存泄漏和OOM（运行状况检查）、磁盘占用（配额问题）、Fork炸弹（进程限制），批量处理和进程隔离（限制内存份额）、延迟和基于响应的服务发现、重试、超时、自动扩展等。同样，在过渡到无服务器模型后，服务必须在几毫秒内处理一个请求。看上去对垃圾回收、线程池、资源泄漏等问题的关注，越来越成为一些毫不相关的问题……

使用平台处理所有诸如此类的问题，会将服务视为一个密封黑盒子。该黑盒子应支持多次启动和停止（支持服务重新启动）、服务按比例的放大和缩小（通过将服务成为无状态的以支持安全扩展）、假定许多传入请求最终会超时（使端点具有幂等性）、假定许多传出请求将暂时失败并且平台将会做出重试（确保我们使用了幂等服务）。

为实现自动化在云原生环境中适用自动化，服务必须满足下列条件：

- 对重启的幂等（服务支持多次被杀掉并启动）。
- 对向上/向下扩展幂等（服务可实现多个实例的自动扩展）。
- 对服务生成者幂等（其它服务可重试调用）。
- 对服务消费者幂等（服務或服务网格可以重试传出请求）。

如果服务在一次或是多次执行上述行为中总是表现出同一方式，那么平台就可以在无需人工干预的情况下从故障中恢复服务。

最后请记住，平台提供的所有恢复只是一些本地优化。正如Christian Posta所指出的，分布式系统中应用的安全性和正确性仍然是应用的责任。对于设计一个整体稳定的系统，业务流程整体范围中的思维模式（可

能跨越多个服务) 十分重要的。

## 双重开发职责

越来越多的微服务原则已被Kubernetes及一些补充项目实施和提供。因此, 现代开发人员必须做到不仅要精通编程语言去实现业务功能, 而且同样也要精通云原生技术去完全满足一些非功能性基础架构层上的需求。

业务需求和基础架构(操作上的需求、跨功能的需求, 或是一些系统质量属性)之间的界限通常是模糊不清的, 我们不可能只采取其中的某个方面, 而期望其他人去实现另一个方面。例如, 如果要在服务网格层实现重试逻辑, 那么必须使服务中的业务逻辑或数据库层所使用的服务具有幂等性。如果在服务网格层使用超时, 那么必须在服务中实现服务使用者超时的同步。如果必须要实现服务的定期执行, 那么必须配置Kubernetes作业去按时间执行。

展望未来, 一些服务功能应作为业务逻辑实现在服务中, 而其它一些服务功能则应作为平台功能提供。虽然使用正确的工具去完成正确的任务是一种很好的责任分离, 但新技术不断出现极大地增加了整体的复杂性。要在业务逻辑方面实现简单的服务, 我们需要很好地理解分布式技术堆栈, 因为开发职责是分散在各个层上的。

事实证明, Kubernetes支持向上扩展到数千个节点, 数万个Pod和每秒数百万事务。但它是否同样支持向下扩展? 对我来说, 我并不清楚应用的规模、复杂性或关键性的阈值应该是多少, 才能证明我们引入复杂的“云原生”是正确的。

## 结论

看到微服务运动为采用Docker和Kubernetes等容器技术提供了如此巨大的动力, 这是非常有意思的。虽然一开始是微服务实践推动了这些技术的发展, 但现在是Kubernetes重新定义了微服务架构的原则和实践。

从最近的一些实例看, 我们将很快采纳功能模型作为有效的微服务原

语，而不是将微服务视为纳米（nanoservice）服务的反模式。我们并没有充分质疑云原生技术对于中小型案例的实用性和适用性，而是出于兴奋有些随意地投身到这个领域中。

Kubernetes从ESB和微服务中汲取了大量经验，因此它将会成为最终的分布式系统平台。它是一种用于定义建筑风格的技术，而不是反之。究竟是好是坏，只有时间才能证明。

## 作者简介

Bilgin Ibryam (@bibryam) 是Red Hat的首席架构师、提交者和ASF成员。他也是一名开源布道师、博客作者，《Camel设计模式》（Camel Design Patterns）和《Kubernetes模式》（Kubernetes Patterns）等书的作者在他的日常工作中，Bilgin 喜欢指导、编码和领导开发人员成功地构建云解决方案。他目前的工作重点是应用程序集成、分布式系统、消息传递、微服务、DevOps 和云原生的挑战。可通过[Twitter](#)、[Linkedin](#)和[个人博客](#)联系Bilgin。

# GitHub：我们为什么会弃用 jQuery？

作者 GitHub 前端工程团队，译者 无明



我们最近完成了一个里程碑，将jQuery完全从GitHub.com的前端代码中移除。这标志着我们数年来逐步移除jQuery这个渐进式的过程终于结束了，我们现在已经完全移除了这个库。这篇文章将介绍我们是如何依赖上jQuery的，以及后来我们意识到不再需要它，到最后我们并没有使用另一个库或框架取代它，而是使用标准的浏览器API实现了我们所需要的一切。

## 为什么在早期 jQuery 对我们来说是有意义的

GitHub.com在2007年底开始使用jQuery 1.2.1。那是谷歌发布Chrome

浏览器的前一年。当时还没有通过CSS选择器来查询DOM元素的标准方法，也没有动态渲染元素样式的标准方法，而Internet Explorer的XMLHttpRequest接口与其他很多API一样，在浏览器之间存在不一致性问题。

jQuery让DOM操作、创建动画和“AJAX”请求变得相当简单——基本上，它让Web开发人员能够创建更加现代化的动态Web体验。最重要的是，使用jQuery为一个浏览器开发的代码也适用于其他浏览器。在GitHub的早期阶段，jQuery让小型的开发团队能够快速进行原型设计并开发出新功能，而无需专门针对每个Web浏览器调整代码。

基于jQuery简单的接口所构建的扩展库也成为GitHub.com前端的基础构建块：[pjax](#)和[facebox](#)。

我们将永远不会忘记John Resig和jQuery贡献者创建和维护的这样一个有用的基本库。

## 后来的 Web 标准

多年来，GitHub成长为一家拥有数百名工程师的公司，并逐渐成立了一个专门的团队，负责JavaScript代码的规模和质量。我们一直在排除技术债务，有时技术债务会随着依赖项的增多而增长，这些依赖项在一开始会为我们带来一定的价值，但这些价值也随着时间的推移而下降。

我们可以将jQuery与现代浏览器支持的Web标准的快速演化进行比较：

- \$(selector)模式可以使用querySelectorAll()来替换；
- 现在可以使用Element.classList来实现CSS类名切换；
- CSS现在支持在样式表中而不是在JavaScript中定义可视动画；
- 现在可以使用Fetch Standard执行\$.ajax请求；
- addEventListener()接口已经足够稳定，可以跨平台使用；
- 我们可以使用轻量级的库来封装事件委托模式；
- 随着JavaScript语言的发展，jQuery提供的一些语法糖已经变得多余。

另外，链式语法不能满足我们想要的编写代码的方式。例如：

```
$('.js-widget')
    .addClass('is-loading')
    .show()
```

这种语法写起来很简单，但是根据我们的标准，它并不能很好地传达我们的意图。作者是否期望在当前页面上有一个或多个js-widget元素？另外，如果我们更新页面标记并意外遗漏了js-widget类名，浏览器是否会抛出异常会告诉我们出了什么问题？默认情况下，当没有任何内容与选择器匹配时，jQuery会跳过整个表达式，但对我们来说，这是一个bug。

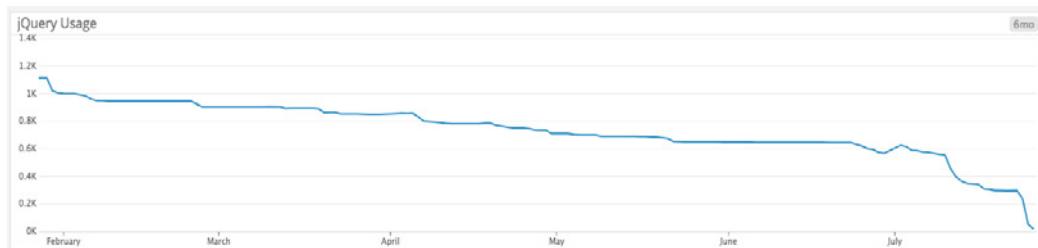
最后，我们开始使用Flow来注解类型，以便在构建时执行静态类型检查，并且我们发现，链式语法不适合做静态分析，因为几乎所有jQuery方法返回的结果都是相同的类型。我们当时之所以选择Flow，是因为@flow weak模式等功能可以让我们逐步将类型应用于无类型的代码库上。

总而言之，移除jQuery意味着我们可以更多地依赖Web标准，让MDN Web文档成为前端开发人员事实上的默认文档，在将来可以维护更具弹性的代码，并且可以将30KB的依赖从我们的捆绑包中移除，加快页面的加载速度和JavaScript的执行速度。

## 逐步解耦

虽然定下了最终目标，但我们也知道，分配所有资源一次性移除jQuery是不可行的。这种匆匆忙忙的做法可能会导致网站功能出现回归。相反，我们采取了以下的策略：

设定指标，跟踪整一行代码调用jQuery的比率，并监控指标走势随时间变化的情况，确保它保持不变或下降，而不是上升。



我们不鼓励在任何新代码中导入jQuery。为了方便自动化，我们创建了[eslint-plugin-jquery](#)，如果有人试图使用jQuery功能，例如\$.ajax，CI检查将会失败。

旧代码中存在大量违反eslint规则的情况，我们在代码注释中使用特定的eslint-disable规则进行了注解。看到这些代码的读者，他们都该知道，这些代码不符合我们当前的编码实践。

我们创建了一个拉请求机器人，当有人试图添加新的eslint-disable规则时，会对拉取请求留下评论。这样我们就可以尽早参与代码评审，并提出替代方案。

很多旧代码使用了pjax和facebox插件，所以我们在保持它们的接口几乎不变的同时，在内部使用JS重新实现它们的逻辑。静态类型检查有助于提升我们进行重构的信心。

很多旧代码与rails-behavior发生交互，我们的Ruby on Rails适配器几乎是“不显眼的”JS，它们将AJAX生命周期处理器附加到某些表单上：

```
// 旧方法
$(document).on('ajaxSuccess', 'form.js-widget',
function(event, xhr, settings, data) {
    // 将响应数据插入到DOM中
})
```

我们选择触发假的ajax\*生命周期事件，并保持这些表单像以前一样异步提交内容，而不是立即重写所有调用，只是会在内部使用fetch()。

我们自己维护了jQuery的一个版本，每当发现我们不再需要jQuery的某个模块的时候，就会将它从自定义版本中删除，并发布更轻量的版本。例如，在移除了jQuery的CSS伪选择器之后（如:visible或:checkbox）我们就可以移除Sizzle模块了，当所有的\$.ajax调用都被fetch()替换时，就可以移除AJAX模块。这样做有两个目的：加快JavaScript执行速度，同时确保不会有新代码试图使用已移除的功能。

我们根据网站的分析结果尽快放弃对旧版本Internet Explorer的支持。每当某个IE版本的使用率低于某个阈值时，我们就会停止向它提供

JavaScript支持，并专注支持更现代的浏览器。尽早放弃对IE 8和IE 9的支持对于我们来说意味着可以采用很多原生的浏览器功能，否则的话有些功能很难通过polyfill来实现。

作为GitHub.com前端功能开发新方法的一部分，我们专注于尽可能多地使用常规HTML，并且逐步添加JavaScript行为作为渐进式增强。因此，那些使用JS增强的Web表单和其他UI元素通常也可以在禁用JavaScript的浏览器上正常运行。在某些情况下，我们可以完全删除某些遗留的代码，而不需要使用JS重写它们。

经过多年的努力，我们逐渐减少对jQuery的依赖，直到没有一行代码引用它为止。

## 自定义元素

近年来一直在炒作一项新技术，即自定义元素——浏览器原生的组件库，这意味着用户无需下载、解析和编译额外的字节。

从2014年开始，我们已经基于v0规范创建了一些自定义元素。然而，由于标准仍然在不断变化，我们并没有投入太多精力。直到2017年，Web Components v1规范发布，并且Chrome和Safari实现了这一规范，我们才开始更广泛地采用自定义元素。

在移除jQuery期间，我们也在寻找用于提取自定义元素的模式。例如，我们将用于显示模态对话框的facebox转换为<details-dialog>元素。

我们的渐进式增强理念也延伸到了自定义元素上。这意味着我们将尽可能多地保留标记内容，然后再标记上添加行为。例如，<local-time>默认显示原始时间戳，它被升级成可以将时间转换为本地时区，而对于<details-dialog>，当它被嵌在<details>元素中时，可以在不使用JavaScript的情况下具备交互性，它被升级成具有辅助增强功能。

以下是实现<local-time>自定义元素的示例：

```
// local-time根据用户的当前时区显示时间。
// 例如：
```

```
//   <local-time datetime="2018-09-06T08:22:49Z">Sep 6, 2018</
local-time>
//
class LocalTimeElement extends HTMLElement {
    static get observedAttributes() {
        return ['datetime']
    }

    attributeChangedCallback(attrName, oldValue, newValue) {
        if (attrName === 'datetime') {
            const date = new Date(newValue)
            this.textContent = date.toLocaleString()
        }
    }
}

if (!window.customElements.get('local-time')) {
    window.LocalTimeElement = LocalTimeElement
    window.customElements.define('local-time', LocalTimeElement)
}
```

我们很期待Web组件的Shadow DOM。Shadow DOM的强大功能为Web带来了很多可能性，但也让polyfill变得更加困难。因为使用polyfill会导致性能损失，因此在生产环境中使用它们是不可行的。

# 从0开始学架构

资深技术专家的  
实战架构心法 50 讲



你将获得

1. 理解架构设计的本质和目的
2. 掌握高性能和高可用架构模式
3. 走进 BAT 标准技术架构实战
4. 从编程到架构，实现思维跃迁

订阅附赠：

《架构师成长技能图谱》

\*扫码获取领取方式

26000 +

技术人已加入学习，邀你一起！



李运华  
资深技术专家

¥99 / 50期全集

新人立减 ¥30



智能时代的新运维

# CNUTCon

## 全球运维技术大会

### ▶ 聚焦12大专题

- AIOps实践与探索
- 自动化运维平台实践
- 监控与分析 (APM)
- 日志处理
- 大规模系统的性能优化
- 智能+时代下的运维破局
- 数据库运维
- CI/CD实践
- 微服务架构与实践
- Kubernetes实战
- 运维新技术
- SRE实践与思考

会议：2018年11月16–17日

培训：2018年11月18–19日

地址：上海 光大会展中心大酒店



8折报名

团购享更



## ▶ 联席主席



**刘国华**  
阿里巴巴 研究员



**吴其敏**  
平安银行  
零售网络金融事业部首席架构师



**涂彦**  
腾讯 游戏运维总监



**李涛**  
百度 工程效率部负责人  
百度平台化委员会秘书长

## ▶ 部分演讲嘉宾



**陈云**  
百度 智能云事业部  
资深研发工程师



**不畏**  
阿里云  
视频云运维专家



**刘伟**  
腾讯  
技术运营部高级工



**夏舰波**  
京东商城  
技术架构部资深架构师



**顾宇**  
ThoughtWorks  
高级咨询师



**谭宇 (茂七)**  
阿里巴巴  
高级技术专家



**闫鹏**  
阿里云  
ARMS技术负责人



**刘林**  
搜狗  
资深软件工程师



**李浩**  
eBay  
主管工程师

报名中，立减720元

多优惠，截止2018年10月21日



联系我们：

售票咨询(电话)：13269078023

售票咨询(邮箱)：piaowu@geekbang.com



## ► 聚焦

- 高性能业务架构
- 微服务治理
- 数据基础平台构建技术
- 数据驱动产品和用户增长
- 短视频架构和算法
- 金融技术框架
- 数据工程 & 大数据智能处理
- 大前端技术
- 区块链技术实践
- 信息安全与隐私保护
- 智能高效运维
- CTO技术选型思维

## ► 实践

[菜鸟网络](#) / 菜鸟仓储物流平台架构演进及技术挑战

[Pinterest](#) / Build a dynamic and responsive Pinterest on AWS:  
a system engineer's perspective

[京东](#) / 分布式BaaS的设计与实践

[Netflix](#) / Netflix API Gateway that handles 2M requests per second

[LinkedIn](#) / 大规模机器学习在LinkedIn预测模型中的应用

[满帮集团](#) / 货车帮云原生平台架构设计思路和实践

[阿里巴巴](#) / 深度学习在智慧餐厅中的应用

会议：2018年12月07-08日

培训：2018年12月09-10日

地址：北京·国际会议中心

8折报

团购享更

## ▶ 出品人



**沈剑**  
58速运 CTO



**何径舟**  
百度 自然语言处理部高级经理  
技术专家



**张磊**  
阿里巴巴集团  
高级技术专家 & 技术顾问



**冯湧**  
美团 美团研发总监

## ▶ 分享嘉宾



**Susheel Aroskar**  
Netflix  
Software Engineer



**李刚**  
百度 可用性工程  
技术负责人



**谷雪梅**  
菜鸟网络  
CTO&副总裁&技术  
产品负责人



**长纪**  
阿里巴巴  
高级技术专家



**刘波**  
Pinterest  
Engineering Manager



**郭平（坤宇）**  
阿里巴巴  
高级技术专家



**张镭**  
LinkedIn  
Engineering Manager



**陈皓（左耳朵耗子）**  
MegaEase  
创始人&CTO



**杨巍威**  
Hortonworks  
YARN/Staff Enginee

名中，立减1360元

多优惠，截止2018年10月28日



联系我们：

电话：17326843116 （灰灰）

微信：aschina666

# AiCon

全球人工智能与机器学习技术大会

## 大会聚焦

- 机器学习应用和实践
- 计算机视觉
- NLP和语音技术
- 搜索推荐与算法
- 数据智能驱动业务
- AI+行业案例



8折报名中，立减720元

联系热线：18514549229

2018年12月20-23 / 北京·国际会议中心



## ▶ 大咖助阵



**颜水成**  
360人工智能研究院  
院长及首席科学家



**华先胜**  
阿里巴巴达摩院机器智能技术实  
验室副主任  
城市大脑人工智能技术负责人



**裴健**  
京东集团副总裁  
大数据与智能供应链事业部总裁



**马维英**  
今日头条副总裁  
人工智能实验室负责人



**崔宝秋**  
小米人工智能与云平台副总裁  
首席架构师

## ▶ 分享嘉宾



**薄列峰**  
京东金融  
AI实验室首席科学家



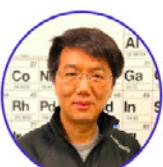
**袁进辉 (老师木)**  
一流科技创始人



**张俊林**  
新浪微博  
AI Lab资深算法专家



**王兴星**  
美团点评技术总监  
外卖商业技术负责人



**陈博兴**  
阿里巴巴达摩院  
资深算法专家



**鲍捷**  
文因互联CEO  
人工智能领域知名专家

更多嘉宾陆续上线中 ...

**8折报名中，立减720元**

联系热线：18514549229

2018年12月20-23 / 北京·国际会议中心





### 架构师 月刊 2018年9月

本期主要内容：Kafka 2.0 重磅发布，新特性独家解读；都去炒 AI 和大数据了，落地的事儿谁来做？我用 Vue 和 React 构建了相同的应用程序，这是他们的差异；人工智能与区块链初探：交集与前瞻



### Kubernetes指南

《Kubernetes 指南》开源电子书旨在整理平时在开发和使用 Kubernetes 时的参考指南和实践心得，更是为了形成一个系统化的参考指南以方便查阅。



### 2018，进击的大前端

全栈与大前端，前端工程师进阶该如何抉择？



### 《英雄联盟》 在线服务运维之道

拳头公司基础设施团队的工程师们分享了他们运维在线服务的发展历程，介绍了他们从手动部署到自动运维的演变过程。