

架构师

ARCHITECT



理论派 | Theory

集群调度框架的架构演进之路

观点 | Opinion

如何成为架构师？

推荐文章 | Article

Airbnb的大数据平台架构

Twitter的A/B测试实践

热点 | Hot

在微服务中保证服务的一致性

Jigsaw终于添加到JDK 9中了



旧金山 伦敦 北京 圣保罗 东京 纽约 上海
San Francisco London Beijing Sao Paulo Tokyo New York Shanghai

QCon

全球软件开发大会

2016年4月21-23日 | 北京·国际会议中心

主办方 **Geekbang** 极客邦科技 **InfoQ**



余锋(褚霸)

阿里云研究员

《阿里云高质量云数据库服务背后——AliCloudDB 智能化支撑系统天象》



郭斯杰

Twitter Staff Software Engineer

《Twitter Messaging的架构演化之路》



更多精彩专题内容

敬请关注: www.qconbeijing.com

咨询热线: 010-64738142

购票咨询: qcon@cn.infoq.com

扫码关注InfoQ官方微信

了解更多技术内容



想了解

其他牛逼哄哄的嘉宾

请扫描二维码

移步至大会官网吧



企业落地微服务必备的三个能力

作者 王磊

微服务架构，虽然诞生的时间不长，但其在各种演讲、文章、书籍上所出现的频率已经让很多人意识到它对软件架构领域带来的影响。经过2015年的快速普及，微服务被越来越多的组织和企业所熟识。2016年，将会有更多的企业将目光聚焦在如何实践并有效将落地这个核心问题上。在这里我来谈下微服务实施的三种策略以及落地时企业必备的三个能力。

通常，微服务的实施有如下几种策略。

1. 新产品微服务化

对于新产品构建而言，使用微服务构建存在一定优势：

- 不必担心同现有遗留系统的整合，在实施上能更加灵活，便于尝试业界先进的开发框架和工具；
- 从零开始构建新产品，易于保持代码结构清晰及制定团队规范。

不过，新产品微服务化也存在一定风险。通常新产品都处于快速试错阶段，需求在不断的变化中，微服务的引入必然会增加额外的开发、运维以及交付成本。

2. 遗留系统的新功能微服务化

对于遗留系统而言，可以将新功能的上线作为微服务化实施的尝试。这类方式实施的优势在于虽然是遗留系统，但新功能部分相对独立，上下文清晰，实施时比较灵活，也便于尝试业界先进的开发框架和工具。不过，这类方式的风险在于：

同现有遗留系统的集成存在风险，尤其是数据部分的访问与共享。

需要与遗留系统共存并为用户提供价值，因此需要构建代理机制，处理不同的请求。

3. 遗留系统的旧功能微服务化

对于遗留系统而言，也可以选择将现有的功能改造成微服务。不过这类方式的风险较大，主要考虑：

- 如何选择合适的部分进行微服务化。这类部分通常包括但不限于功能独立的模块、频繁使用的模块以及占用特殊资源的模块等；
- 如何保证改造过程中不对现有的功能造成破坏。对于某些时间较长的遗留系统，通常功能交错复杂，人员更换频繁，在改造的过程中难免；
- 需要同遗留系统共存为用户提供价值，因此需要构建代理机制，处理不同的请求。

微服务的概念看似浅显易懂，但实际上却涉及架构演进、领域建模、持续交付，虚拟化以及 DevOps 等多个维度的方法论与实践。在微服务的实践过程中，如下几点也将成为企业实施微服务架构的必备能力。

1. 持续交付是内功

十年以前，软件在一年中的交付次数屈指可数。

过去的十年间，交付的过程一直被不断地优化和改进。从早期的 RUP 模型、敏捷、持续集成，再到近几年的 DevOps，力求能更有效地降低交付过程所耗费的成本并提高效率，从而尽早实现价值。持续交付的提出，极大的优化了软件交付的流程，帮助企业更有效的验证业务想法，并通过快速迭代的方式持续为用户提供价值。

对于构建微服务而言，持续交付机制建立的顺畅与否，直接决定了微服务构架实施的成本与效率，稳固的持续交付体系能让微服务架构的实施事半功倍。

持续交付和微服务只有组合在一起才能展现出相互的价值。

2. 演进式架构是核心

架构是 IT 领域经久不衰的话题之一。架构的本质是对业务、技术、团队以及可维护性等多重因素下做的平衡。纵览 IT 架构发展的历史长河，存在诸多行之有效的模式与方法，譬如企业架构模式、设计模式及近几年的领域驱动设计，帮助 IT 团队完成架构设计。

在如今市场激烈竞争的环境下，业务快速变化，架构也需要不断的变化。没有完美的架构，只有恰当的平衡。

对于微服务架构的实施过程，很多朋友纠结如何定义“完美的”服务划分，其实架构的演进过程中，必然会经历服务的定义、拆分、合并以及组合，微服务实施的过程也是一个架构不断动态平衡的过程，不必太纠结初期多么“完美”。业务在变，技术在变，架构也在变，拥抱变化，完善内部的微服务生态圈（开发流程、交付流程、自动化机制、部署以及监控），寻找演进式架构的动态平衡才是核心。

3. DevOps 是动力

运维能力是企业实施微服务的关键动力。相比于传统架构，微服务的落地增加了大量的运维工作。随着服务的增多，有更多的服务需要部署、监控。另外，随着服务的增多，出错的可能性大大增加，出错时如何快速

恢复也是件很复杂的事。因此，在实施微服务的过程中，运维能力直接决定了实施的效率和产出。

另外，在加强运维能力本身的同时，微服务的实施还需要很强的 DevOps 文化。如果开发团队和运维团队之间无法密切协作，那实施过程将会存在很大的沟通成本。因此，运维能力和 DevOps 是企业实施微服务的关键动力。

除此之外，微服务带来的不仅仅是技术上的变革，也带来企业组织结构上的变革。传统的按照技能划分部门的方式，将越来越难以适应当今业务的激烈竞争和市场的快速变化。通过将大部门基于业务、基于服务，组织成适当的的小团队，不仅能有效提高成员的主人翁精神，也容易在实现层面给予团队充分的自由度，这也是目前硅谷诸多顶尖 IT 公司青睐的方式。

综上所示，微服务的实施注定不是一个简单的过程，是需要企业潜心积累、循序渐进的过程。企业需要根据不同的业务，不同的发展阶段，选择不同的实施策略，并在实施的过程中不断提高关键能力，并构建企业内部的微服务生态圈，才能享受到微服务架构带来的巨大价值。

CONTENTS / 目录

热点 | Hot

为了实现一致性，我们从事务方案转移到流处理方案

在微服务中保证服务的一致性

AutoMapper 及静态类之争论

Jigsaw 终于添加到 JDK 9 中了

推荐文章 | Article

Airbnb 的大数据平台架构

Twitter 的 A/B 测试实践：为什么要测试以及测试的意义

理论派 | Theory

集群调度框架的架构演进之路

观点 | Opinion

如何成为架构师？7 个关键的思考、习惯和经验

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

5月15日前
8 折购票
团购优惠更多

sz2016.archsummit.com
购票电话:010-89880682



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 活动专区全新上线

更多活动·更多选择

一站获取所有活动信息

Geekbang

极客邦科技

InfoQ | EGO | StuQ | GiT
技术媒体 职业社交 在线教育 企业培训



扫描二维码
前往专区

为了实现一致性， 我们从事务方案转移到流处理方案

作者 Jan Stenberg 译者 张卫滨

当系统变得越来越复杂，数据库会被拆分为多个更小的库，如果借助这些衍生库实现像全文搜索这样的功能，那么如何保证所有的数据保持同步就是一项很有挑战性的任务了，在最近的 QCon 伦敦会议上，Martin Kleppmann 通过[演讲](#)阐述了他的观点。

使用多个数据库时，最大的问题在于它们并不是互相独立的。相同的数据会以不同的形式进行存储，所以当数据更新的时候，具有对应数据的所有数据库都需要进行更新。保证数据同步的最常用方案就是将其视为应用程序逻辑的责任，通常会对每个数据库进行独立的写操作。这是一个脆弱的方案，如果发生像网络故障或服务器宕机这样的失败场景，那么对一些数据库的更新可能会失败，从而导致这些数据库之间出现不一致性。

Kleppmann 认为这并不是能够进行自我纠正的最终一致性，至少相同的数
据再次进行写操作之前，无法实现一致性。

这不是最终一致性，它更像是持续的不一致性。

传统的方案使用事务来实现原子性，但是 Kleppmann 认为这只有在一个数据库的时候才有效，如果是两个不同的数据存储的话，那么这就不太可行了。分布式事务（又称为[两阶段提交](#)）支持跨多个存储系统，但是 Kleppmann 认为它也面临自身的挑战，如较差的性能和运维问题。

我们重新回过头来看一下这个问题，Kleppmann 认为有一种很简单的解决方案，那就是按照系统的顺序对所有的写操作进行排序，并且确保所有人在随后读取时遵循相同的顺序。他将其与[确定性的状态机复制](#)（deterministic state machine replication）进行了类比，对于相同的起始状态，给定的输入流在多次运行时将会始终产生相同的状态转换。

在 leader（主）数据库中，同时会将所有的写入操作按照处理的顺序存储为流，然后一个或多个 follower 数据库就能读取这个流并按照完全相同的顺序执行写入。这样的话，这些数据库就能更新自己的数据并成为 leader 数据库的一致性备份。对于 Kleppmann 来说，这是一个非常具有容错性的方案。每个 follower 都遵循它在流中的顺序，在出现网络故障或宕机时，follower 数据库能够从上一次的保存点开始继续进行处理。

Kleppmann 还提到在实现上述场景时，使用 [Kafka](#) 作为工具之一。目前，他正在编写一个实现，[Bottled Water](#)，在这个实现中，他使用了 PostgreSQL 来抽取数据变化，然后将其中继到 Kafka 中，代码可以在 [GitHub](#) 上获取到。

InfoQ 最近也[发布](#)了一个关于使用 Kafka 进行开发的演讲。

QCon 的参会者已经聆听了 Kleppmann 的演讲，InfoQ 的读者稍后也将能看到。他还将在演讲的[讲稿](#)发布了出来。

在微服务中保证服务的一致性

作者 Jan Stenberg 译者 邵思华

当在近期举办的 QCon London 大会上，Ben Stopford 在他的[演讲](#)中极力主张拥抱去集中化的思想、构建基于服务的系统，并通过[流处理](#)工具解决分布式状态所引起的问题。

Stopford 目前任职于 Confluent，参与[Kafka](#) 的开发工作。对他来说，构建基于服务的系统的理由有很多，包括松耦合、边界上下文、易于扩展等等，这些特性让我们能够构建出可以随着时间的推移而不断改进的系统。但是，通过这种方式，我们实质上是在创建分布式的系统，而分布式系统自有其本身的复杂性，并且在延迟与故障等方面还存在着种种问题。

Stopford 描述了分布式的两种基本模式：

以[请求 - 响应](#)的方式对服务进行解耦，通常使用 REST 的服务实现。它很适合于 UI 以及提问的场景。

事件驱动，这种模式的特征是异步的，或是“fire and forget”的消息传递。它非常适合于设计跨服务的复杂依赖。

这两种模式可以结合使用，例如使用请求 - 响应模式实现一个 REST 接口，随后以事件的方式进行后台处理。

Stopford 随后对异步与基于事件的通信，例如队列的使用展开了讨论。他认为这种模型非常简单，只要做到一次只取出一条消息，就能够保证消息的次序。即使将这一方式进行一定程度的扩展，仍然可以保证它的

次序，但 Stopford 指出，在某些情况下，我们或许会失去可用性或是次序的保证。他还指出了该方式的另一个缺点，即消息的存在是短期的，因此服务一旦出现故障，就无法回到之前的时间点再次读取这些消息了。

Stopford 认为，更好的方法是使用某种分布式日志支持服务的开发，并以 Kafka 为例。Kafka 是基于日志的概念而设计的，而日志是一种只增的数据结构。因此读写操作都非常高效，对于读操作来说，只需定位到某个位置，并进行顺序读取。而对写操作来说，所做的只是简单的添加而已。

分布式的日志系统还能够为微服务带来以下好处：

- 始终在线，这依赖于某种容错的代理，例如 Kafka。
- 负载均衡，每个服务实例都将从一个代理中读取数据。
- 容错性，这是因为服务可能会产生故障转移，但消息的次序仍然保持不变。
- 倒带和回放，当系统发现了某个错误并修复之后，服务可以找回原始的消息，并进行回放。

但这种方式仍然有一个未解决的问题，即保持服务的一致性。因为在系统发生故障等一些情况下，很难避免出现一些重复的消息（“即至少一次”的提交机制）。因此，服务在处理他们收到的消息时必须保证幂等性（idempotent）。从逻辑上说，这相当于创建了一种“正好一次”的提交机制。Stopford 表示，这一功能在 Kafka 中尚未实现（但相关功能已经在开发中了）。

Martin Kleppmann 在本次大会的另一场演讲中也提到了服务一致性的问题。

QCon 的参会者已经可以欣赏 Stopford 的演讲了，而 InfoQ 的读者很快也能够欣赏到演讲的内容。Stopford 同时也发布了这次演讲的幻灯片。

AutoMapper 及静态类之争论

作者 Jonathan Allen 译者 邵思华

在进行 API 设计时，静态类的使用有时会为设计者带来一些烦恼。应该将某个函数暴露为静态函数还是实例方法，这一点常常会造成人们的争论。

静态函数的主要优点在于其简便性。调用者可以在代码中的任意位置使用静态函数，而无需为实例的创建、管理以及依赖注入等问题而烦恼。并且由于没有创建新的实例，因而也不存在垃圾回收的问题，从而使性能也得以提高。

如果没有维护状态的需求，以上的论点确实是成立的。如若不然，则设计者必须保证静态函数的线程安全，而这往往牵涉到开销较大的加锁与同步等技术。而且即便独立的调用是线程安全的，但调用者也往往需要将一系列调用过程封装为一个原子性的事务。AutoMapper 目前也遇到了这方面的麻烦。

AutoMapper 最初是围绕着静态函数而设计的，但随着时间的推移，它的可配置性也在逐步提高。每当出现新的配置选项，就需要管理更多的状态，而潜在的线程问题也在逐渐加剧。因此，今年 1 月，Jimmy Bogard 将 AutoMapper 4.2 版本中的静态函数一律标记为过时（obsolete）[方法](#)，并打算最终完全移除这些函数。

在我开发 AutoMapper 4.2 版本的过程中，脑海中突然有灵光一闪。过去这十年间，我多次在讲座与播客中谈到了如何长期维护开源代码的问题。对于 AutoMapper，我最大的遗憾就是在一开始设计了一套静态的 API。AutoMapper 最初的测试与原型中都是通过“Mapper.CreateMap”与“Mapper.Map”等方法调用的。当时我向我的老板 Jeffrey Palermo 展示了我的代码，并询问他对代码的看法。他当时说道：“这看上去很棒 Jimmy，不过 API 似乎不应该设计成静态的”，而我则回应说：“开玩笑吧，这不可能！”。

之后，我开始意识到静态函数的问题，至今都为此感到懊悔。在即将发布的新版本中，我利用这次机会设计了一个不再使用静态方法的原型，它表现得很出色，我也准备好将整个静态 API 标记为过时方法。

这一改动也确实造成了某些问题。AutoMapper 的特性之一是支持 fluent API，它能够配合 LINQ 表达式链工作。这一特性需要用到扩展方法，而扩展方法往往都是通过静态函数的方式定义的。

我选择的临时方案是仍然提供对 LINQ 的支持，但改变了它的方式，使其不再利用全局的状态。使用者需要将 AutoMapper 配置信息传递给 LINQ 表达式，这种方式稍嫌冗长，但从某些方面来看，它提供了更大的灵活性。

示例代码是从“[静态 API 迁移指南](#)”中所摘录的一段。

可就在一个月后，Jimmy Bogard 又决定让这些[静态函数重新回归](#)。他写道：

静态 API 的一大困扰在于使用者可以随时对配置进行改动，而我却无法强制要求使用者对配置的步骤进行清理。但在进一步思考之后，我发现静态 API 的使用并没有任何问题，它只是要求使用者在进行映射之前必须

```
001 public class ProductsController : Controller {  
002     public ProductsController(MapperConfiguration config) {  
003         this.config = config;  
004     }  
005     private MapperConfiguration config;  
006  
007     public ActionResult Index(int id) {  
008         var dto = dbContext.Products  
009                         .Where(p => p.Id == id)  
010                         .ProjectTo(config)  
011                         .SingleOrDefault();  
012  
013         return View(dto);  
014     }  
015 }
```

完成初始化工作。因此我决定在后续版本中仍然允许这种使用方式。实例 API 如今已经彻底完善了，而静态 API 实际上只是一种轻量级的封装，使用者可以简单地调用静态 Initialize 方法，而无需直接调用实例的构造函数。

新发布的版本移除了某些过时属性，并且恢复了在 LINQ 映射时使用静态配置的特性。

关于应当使用有状态的静态函数，还是只允许使用实例方法，InfoQ 希望聆听读者的意见。

大数据生态构建

北京国际会议中心 203AB
2016年4月22日

Jigsaw 终于添加到 JDK 9 中了

作者 Alex Blewitt 译者 张卫滨

在发往 jdk9-dev 邮件列表的一封[邮件](#)中，Alan Bateman 宣布 Jigsaw 模块系统的一个快照版本将会合并到 JDK9 开发分支的主线中。模块系统的状态[文档](#)最近进行了更新来表明它的进展，这是从 Jigsaw 项目启动以来，它的变更第一次合并回来。

在历史上，Java 的开发在不同的分支下进行，因为它们是 Mercurial 工具下[不同的 tree](#)。Hotspot 编译器（以及底层的 VM 内容）是在 Hotspot tree 中开发的，而 Java 开发的主线是在 jdk tree 中完成的。在 Open JDK 的集合中，还有不同的 tree 用于 nashorn 甚至 corba。这样造成的结果就是，每个分组就被称为 forest，因为它们是相互关联的 tree。（这种结构很大程度上是因为在创建项目的时候，Mercurial 没有轻量级的分支功能，并且它没有很好的跨分支 push 的功能，所以它们都被分割为不同的 repository，这也是为什么会有 jdk8 和 jdk8u forest 的原因，jdk8 实际上是主开发分支，而 jdk8u 包含了其他 repository 的 fork，而且带上了特定的标签，如 8u40 和 8u60。幸好，所有的这些内容能够全部放到一个[Git repository](#)之中，这其实是非常有价值的。）

Jigsaw 所引入的变化是很明显的，底层的包被划分成了不同的模块，每个模块包含一个或多个包。java.base 模块包含了标准库的主要内容（java.lang、java.util 以及 java.time 等）。这种划分会带来一定的破坏性，java.beans 包因为它与 AWT 和 Applets 的密切关联，

被转移到了 java.desktop 模块中，这就意味着在 java.base 包中，为 PropertyChangeListener 实现监听器注册的类被移除掉了。

为了保证 JDK 9 和 Jigsaw 的开发不会遇到太多的困难，Jigsaw 实现是位于它自己的 forest 之中的。这样的话，通过 Oracle 和 [Azul Zulu](#) 得到的主线 OpenJDK Java 构建版本是不包含 Jigsaw 的，开发人员也无法进行尝试。而我们可以得到一个特殊的“早期可访问”（early access）构建版本，这个版本提供了 Jigsaw 的内容，但是并不包含 JDK 9 分支的一些其他变更。

所以，这封邮件确认 Oracle 将会尝试将并行开发的 Jigsaw forest 合并到 JDK9 forest 之中。一些重要的变更（如移除 PropertyChangeListeners）已经完成了，[自动化的差异对比](#)显示没有其他的功能移除了，不过包含了几百项对 API 的添加 / 变更。为了尽可能减少额外出现的问题，JDK9 forest 将会锁定两周的时间，以便于将 Jigsaw 相关的变更合并到主线的 JDK9 forest 中。预期将会只有一个构建版本——也就是 3 月 21 日的 JDK9+111——不过，Jigsaw 团队也申请预留了 3 月 27 日的 JDK9+112 构建版本，以应对 Jigsaw 可能出现的变化。

Jigsaw 模块后续的变更将会继续在 Jigsaw forest 中进行，因此针对 JDK9 主线的开发人员将不会处理由此导致的不稳定。但是，这个构建版本是第一次把 Java 的基础库划分为不同的模块，并且会在一个能够感知模块的系统中进行编译，因此希望能够避免将来在模块间添加功能所导致的问题。同时，Jigsaw forest 未来将会完全合并到 JDK9 中，它本身会被移除，不过这可能需要等到年底才会进行。

在[合并之前](#)，已经进行了一些评估，InfoQ 将会持续关注模块系统和 Jigsaw 合并到 JDK9 事宜的进展。

Airbnb 的大数据平台架构

作者 侠天



Airbnb 成立于 2008 年 8 月，拥有世界一流的客户服务和日益增长的用户社区。随着 Airbnb 的业务日益复杂，其大数据平台数据量也迎来了爆炸式增长。

本文为 Airbnb 公司工程师 James Mayfield 分析的 Airbnb 大数据平台构架，提供了详尽的思想和实施。

Part 1：大数据架构背后的哲理

Airbnb 公司提倡数据信息化，凡事以数据说话。收集指标，通过实验证假设、构建机器学习模型和挖掘商业机会使得 Airbnb 公司高速、灵活的成长。

经过多版本迭代之后，大数据架构栈基本稳定、可靠和可扩展的。本文分享了 Airbnb 公司大数据架构经验给社区。后续会给出一系列的文章来讲述分布式架构和使用的相应的组件。James Mayfield 说，“我们每天使用着开源社区提供的优秀的项目，这些项目让大家更好的工作。我们在使用这些有用的项目得到好处之后也得回馈社区。”

下面基于在 Airbnb 公司大数据平台架构构建过程的经验，给出一些有效的观点。

多关注开源社区：在开源社区有很多大数据架构方面优秀的资源，需要去采用这些系统。同样，当我们自己开发了有用的项目也最好回馈给社区，这样会良性循环。

多采用标准组件和方法：有时候自己造轮子并不如使用已有的更好资源。当凭直觉去开发出一种“与众不同”的方法时，你得考虑维护和修复这些程序的隐性成本。

确保大数据平台的可扩展性：当前业务数据已不仅仅是随着业务线性增长了，而是爆发性增长。我们得确保产品能满足这种业务的增长。

多倾听同事的反馈来解决问题：倾听公司数据的使用者反馈意见是架构路线图中非常重要的一步。

预留多余资源：集群资源的超负荷使用让我们培养了一种探索无限可能的文化。对于架构团队来说，经常沉浸在早期资源充足 的兴奋中，但 Airbnb 大数据团队总是假设数据仓库的新业务规模比现有机器资源大。

Part 2：大数据架构预览

图 1 是大数据平台架构一览图。

Airbnb 数据源主要来自两方面：数据埋点发送事件日志到 Kafka；

AIRBNB DATA INFRA

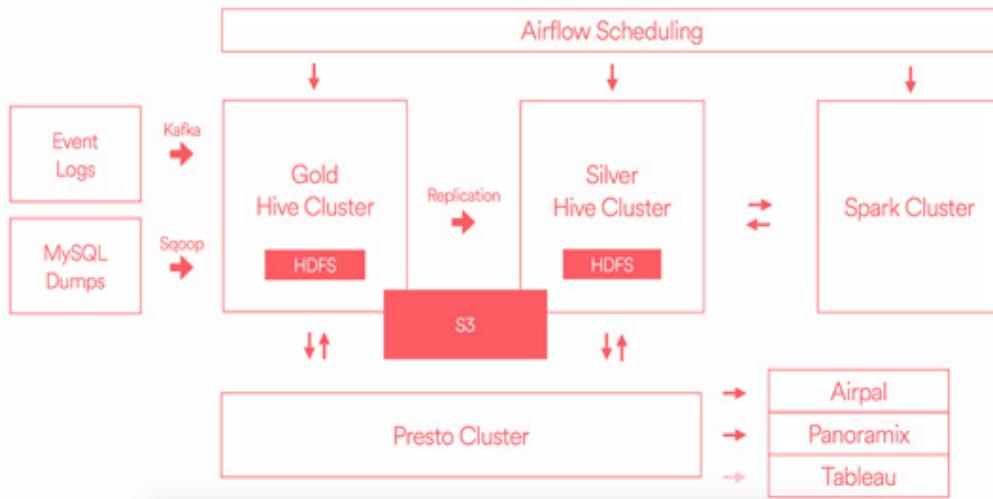


图 1

MySQL 数据库 dumps 存储在 AWS 的 RDS，通过数据传输组件 Sqoop 传输到 Hive “金” 集群（其实就是 Hive 集群，只是 Airbnb 内部有两个 Hive 集群，分别为 “金” 集群和 “银” 集群，具体分开两个集群的原因会在文章末尾给出。）。

包含用户行为以及纬度快照的数据发送到 Hive “金” 集群存储，并进行数据清洗。这步会做些业务逻辑计算，聚合数据表，并进行数据校验。

在以上架构图中，Hive 集群单独区分 “金” 集群和 “银” 集群大面上的原因是为了把数据存储和计算进行分离。这样可以保证灾难性恢复。这个架构中，“金” 集群运行着更重要的作业和服务，对资源占用和即席查询可以达到无感知。“银” 集群只是作为一个产品环境。

“金” 集群存储的是原始数据，然后复制 “金” 集群上的所有数据到 “银” 集群。但是在 “银” 集群上生成的数据不会再复制到 “金” 集群。你可以认为 “银” 集群是所有数据的一个超集。由于 Airbnb 大部分数据分析和报表都出自 “银” 集群，所以得保证 “银” 集群能够无延迟的复制数据。

更严格的讲，对于“金”集群上已存在的数据进行更新也得迅速的同步到“银”集群。集群间的数据同步优化在开源社区并没有很好的解决方案，Airbnb 自己实现了一个工具，后续文章会详细的讲。

在 HDFS 存储和 Hive 表的管理方面做了不少优化。数据仓库的质量依赖于数据的不变性（Hive 表的分区）。更进一步，Airbnb 不提倡建立不同的数据系统，也不想单独为数据源和终端用户报表维护单独的架构。以以往的经验看，中间数据系统会造成数据的不一致性，增加 ETL 的负担，让回溯数据源到数据指标的演化链变得异常艰难。Airbnb 采用 Presto 来查询 Hive 表，代替 Oracle、Teradata、Vertica、Redshift 等。在未来，希望可以直接用 Presto 连接 Tableau。

另外一个值得注意的几个事情，在架构图中的 Airpal，一个基于 Presto，web 查询系统，已经开源。Airpal 是 Airbnb 公司用户基于数据仓库的即席 SQL 查询借口，有超过 1/3 的 Airbnb 同事在使用此工具查询。任务调度系统 Airflow，可以跨平台运行 Hive，Presto，Spark，MySQL 等 Job，并提供调度和监控功能。Spark 集群时工程师和数据分析师偏爱的工具，可以提供机器学习和流处理。S3 作为一个独立的存储，大数据团队从 HDFS 上收回部分数据，这样可以减少存储的成本。并更新 Hive 的表指向 S3 文件，容易访问数据和元数据管理。

Part 3：Hadoop 集群演化

Airbnb 公司在今年迁移集群到“金和银”集群。为了后续的可扩展，两年前迁移 Amazon EMR 到 EC2 实例上运行 HDFS，存储有 300 TB 数据。现在，Airbnb 公司有两个独立的 HDFS 集群，存储的数据量达 11PB。S3 上也存储了几 PB 数据。

下面是遇到的主要问题和解决方案：

A. 基于 Mesos 运行 Hadoop

早期 Airbnb 工程师发现 Mesos 计算框架可以跨服务发布。在 AWS c3.8xlarge 机器上搭建集群，在 EBS 上存储 3TB 的数据。在 Mesos 上运行所有 Hadoop、Hive、Presto、Chronos 和 Marathon。

基于 Mesos 的 Hadoop 集群遇到的问题：

- Job 运行和产生的日志不可见
- Hadoop 集群健康状态不可见
- Mesos 只支持 MR1
- task tracker 连接导致性能问题
- 系统的高负载，并很难定位
- 不兼容 Hadoop 安全认证 Kerberos

解决方法：不自己造轮子，直接采用其它大公司的解决方案。

B. 远程读数据和写数据

所有的 HDFS 数据都存储在持久性数据块级存储卷（EBS），当查询时都是通过网络访问 Amazon EC2。Hadoop 设计在节点本地读写速度会更快，而现在的部署跟这相悖。

Hadoop 集群数据分成三部分存储在 AWS 一个分区三个节点上，每个节点都在不同的机架上。所以三个不同的副本就存储在不同的机架上，导致一直在远程的读数据和写入数据。这个问题导致在数据移动或者远程复制的过程出现丢失或者崩溃。

解决方法：使用本地存储的实例，并运行在单个节点上。

C. 在同构机器上混布任务

纵观所有的任务，发现整体的架构中有两种完全不同的需求配置。

Hive/Hadoop/HDFS 是存储密集型，基本不耗内存和 CPU。而 Presto 和 Spark 是耗内存和 CPU 型，并不怎么需要存储。在 AWS c3.8xlarge 机器上持久性数据块级存储卷（EBS）里存储 3 TB 是非常昂贵的。

解决方法：迁移到 Mesos 计算框架后，可以选择不同类型的机器运行不同的集群。比如，选择 AWS c3.8xlarge 实例运行 Spark。AWS 后来发布了“D 系列”实例。从 AWS c3.8xlarge 实例每节点远程的 3 TB 存储迁移数据到 AWS d2.8xlarge 4 TB 本地存储，这给 Airbnb 公司未来三年节约了上亿美元。

D. HDFS Federation

早期 Airbnb 公司使用 Pinky 和 Brain 两个集群联合，数据存储共享，但 mappers 和 reducers 是在每个集群上逻辑独立的。这导致用户访问数据需要在 Pinky 和 Brain 两个集群都查询一遍。并且这种集群联合不能广泛被支持，运行也不稳定。

解决方法：迁移数据到各 HDFS 节点，达到机器水平的隔离性，这样更容易容灾。

E. 繁重的系统监控

个性化系统架构的严重问题之一是需要自己开发独立的监控和报警系统。Hadoop、Hive 和 HDFS 都是复杂的系统，经常出现各种 bug。试图跟踪所有失败的状态，并能设置合适的阈值是一项非常具有挑战性的工作。

解决方法：通过和大数据公司 Cloudera 签订协议获得专家在架构和运维这些大系统的支持。减少公司维护的负担。Cloudera 提供的 Manager 工具减少了监控和报警的工作。

最后陈述

在评估老系统的问题和低效率后进行了系统的修复。无感知的迁移 PB 级数据和成百上千的 Jobs 是一个长期的过程。作者提出后面会单独写相关的文章，并开源对于的工具给开源社区。

大数据平台的演化给公司减少大量成本，并且优化集群的性能，下面是一些统计数据：

- 磁盘读写数据的速度从 70 - 150 MB / sec 到 400 + MB / sec;
- Hive 任务提高了两倍的 CPU 时间；
- 读吞吐量提高了三倍；
- 写吞吐量提高了两倍；
- 成本减少百分之七十。

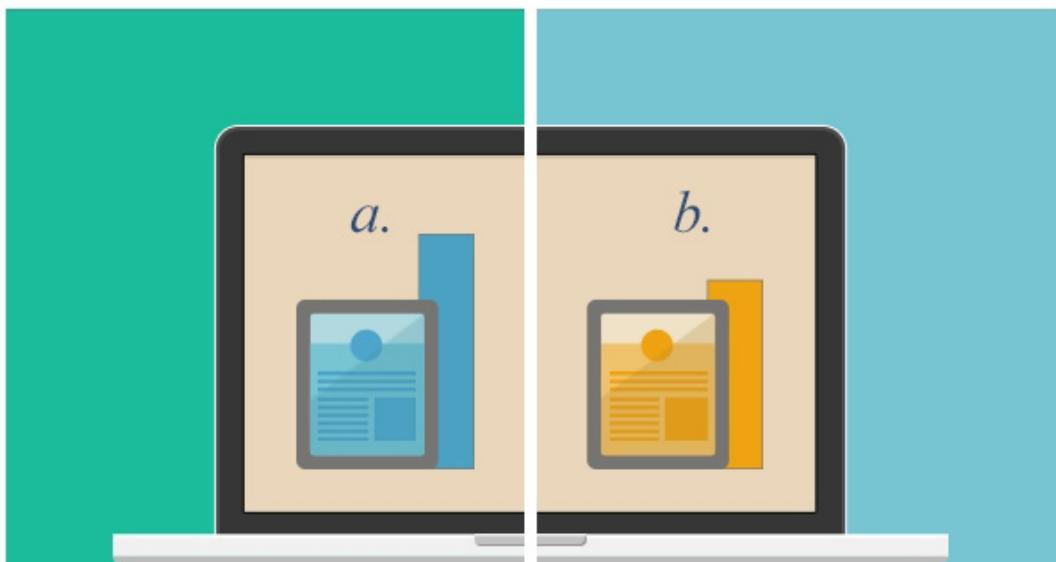


运维与监控

| 2016年4月23日
| 北京国际会议中心203AB

Twitter 的 A/B 测试实践（一）： 为什么要测试以及测试的意义

作者 陆志伟



【编者的话】 A /B 测试曾在多个领域产生深远的影响，其中包括医药、农业、制造业和广告。 在软件开发中， A/B 测试实验提供了一个有价值的方式来评估新特性对客户行为的影响。在这个系列中，我们将描述 Twitter 的 A/B 测试系统的技术和统计现状。

本文是该系列的第一篇，主要介绍了为什么进行 A/B 测试和如何避免其中的陷阱。

注：本文最初发布于 Twitter 博客， InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

实验是 Twitter 产品开发周期的核心。这种实验文化可能是因为 Twitter 对工具、研究和培训进行了大量的投资，以确保特性团队能够对他们的想法进行无缝、严格的测试和验证。

Twitter 实验的规模无论数量还是种类都是庞大的——从细微的 UI/UX 变更，到新特性，到机器学习模型的改进。我们喜欢将实验看作是一个无尽的学习环。（见图 1）

- 建立假设：提出新特性想法或者为现有特性提出改进建议。
- 定义成功指标：评估“机会大小（opportunity size）”——受该变更影响的用户数量。正式定义实验成功和失败的指标；考虑可接受的折衷方案。
- 检验假设：实现拟定的变更，”检测（instrument）“相应的日志，并执行合理性检查以确保实验正确配置。
- 学习：检查实验中收集的数据，吸取其中的经验教训，并与其他 Twitter 团队共享。
- 发布：收集完数据后，判断实验是否验证了假设，并决定发布或者不发布。

建立另一个假设：结合实验中的新想法，为更多的改进建立更多的假设。

A/B 测试、决策制定和创新

Twitter 的“产品检测和实验（Product Instrumentation and Experimentation）”（PIE）团队对实验哲学进行了大量的思考。A/B 测试可以带来很多好处，但是它也有很多众所周知的、容易陷入的陷阱。它

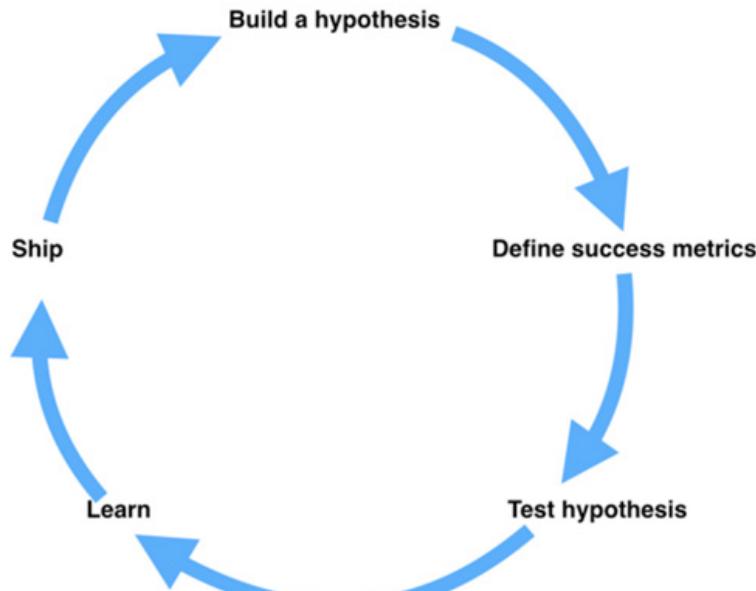


图 1

的结果往往是出人意料、违反直觉的。我们如何避免这种陷阱？什么时候我们应该建议进行 A/B 测试，从而试验特性或者拟定的变更？在决策制定过程中我们如何保持敏捷，且在承受重大风险的时候仍能保持严谨？

测试的好处以及增量测试

特性变更 A/B 测试文化重点关注的是它带来的是小增量收益，大部分实验只能带来个位数百分比的改进，或者甚至是分数百分比。因此，有些观点认为，这有什么意义呢？为什么不从事一些更有影响力、更有革命性的事情呢？

这是事实：如果存在实验能够提升指标，那么到目前为止，大多数实验在以最低限度的方式提升着指标；某个实验如果能够为大部分用户将某个核心指标提升数个百分点就被视为一种非凡的成功。

这与 A/B 测试的基本原理无关。这是因为一个成熟的产品很难通过

大幅度改进指标的方式改变。很多人认为的全垒打想法根本没有带来一点点的改进：人类原来极其不善于预测什么可行（更多信息请参阅 “Seven Rules of Thumb for Web Site Experimenters”）。大多数时候，不理想的 A/B 测试结果让我们能够及早发现，看上去不错的想法其实可能不怎么样。因此我们更愿意尽可能快的获得坏消息，重新回到绘图板；这就是我们实验的原因。

A/B 测试是一种可以确保好想法不会夭折的方法，使好的想法有机会全面充分开发。当我们真正的信任某个想法，而初步实验结果不能满足我们的期望的时候，我们可以对产品做进一步改进，持续改进直到符合期望可以发布给数百万的人使用。另一种方法是，构建一些感觉良好的特性并发布，之后开发其它新的想法，一年后有人意识到根本没人在使用这个特性，它就这样静静地陨落了。

在我们从事各种各样原型开发的时候，快速迭代和衡量拟定的变更带来的影响让我们团队能够及早将隐式用户反馈吸收进产品中。我们可以发布一个变更，研究哪些能够产生改进，哪些不能，接着为能够进一步改进产品的变更建立假设，然后发布变更，周而复始直到拥有能够推送给更广泛用户的变更。

有些人可能认为这种增量变更效率太低。当然，发布“大创意”听起来远比小改进好太多了。然而仔细想一下，将许多小变更叠加起来就能产生复合效果。回避增量变更的产品改进方法很大程度上不是一个好方针。一个好的金融投资组合要能够平衡尽管回报不那么高但可预测的无风险赌注和高风险、高收益赌注。在这方面产品组合管理没有什么不同。

这就是说，有很多东西我们不能，或者不应该测试。有些变更被设计用于形成网络效应，而这些基于用户分桶的 A/B 测试不会捕获（尽管确实存在其它技术能够量化这种影响）。当只针对一个随机比例的人群时，某些特性可能会失灵。例如，在简单的 A/B 测试中，Group DMs 就不是一个可使用的特性，因为可能那些有幸获得此特性的人想要给那些没有获得此特性的人留言，这使得该特性基本无用。其它特性可能是完全的正交——例如推出像 Periscope 一样的新应用程序就不是 Twitter 应用实验。但是一旦推出，A/B 测试就成为一种重要的驱动应用程序内可度量增量和不容易度量的增量变更的方法。

还有另一类变更是，主要的新特性在内部构建过程中通过用户研究进行测试，但是考虑市场战略原因，在特定爆炸性时刻发布给所有的用户。作为一个组织，我们在自认为对产品和用户都有利的时候做出这个决定。我们相信尽管增量变更可能带来更好的初始版本，使更多用户尝试和使用，但是我们能够从大版本中获得更多收益。这是产品负责人需要权衡取舍的。那么当这么一个新特性发布后，我们要对其增量变更进行 A/B 测试吗？当然啦！随着想法的成熟，我们使用完善的科学原理指导它们的演化——并且实验是该过程的关键部分。

实验的可靠性

既然我们已经对运行实验进行了案例说明，接着让我们来讨论怎么做可以避免陷阱。实验的配置和分析是复杂的。即使是正常的人类行为也很容易引起对结果的偏差和误解。这里有几种实践方法，可以降低风险。

需求假设

通常实验工具能够揭示大量数据，常常允许实验者设计自定义的指标来衡量变更的影响。但这可能触发 A/B 测试中最隐匿的陷阱之一：“cherry-picking” 和 “HARKing” ——从许多数据点中选择仅仅支持你的假设的指标，或者看到数据后调整假设，从而让它匹配实验结果。在 Twitter，一个实验收集上百个指标是很常见的，这些指标可以分解成大量的维度（用户属性、设备类型、国家等等），生成数以千计的观测值——如果你希望拟合使其适应任何故事，就需要从中挑选。

我们指导实验者远离 cherry-picking 的一个方法是，要求他们在配置阶段明确指定希望改进的指标。实验者愿意跟踪多少指标都可以，但是只有少数指标可以用这种方式明确标记。然后工具在结果页面突出显示这些指标。实验者可以自由探索所有其它已经被收集的数据，建立新的假设，但是最初的主张应该是固定的，并且要容易检查。

实验过程

无论工具有多好，一套配置不完善的实验仍然会交付不理想的结果。在 Twitter，我们已经对创建实验过程进行了投资，从而提高实验成功、正确运行的概率。在这个过程中大多数步骤是可选的——但是我们发现，使其可用和详细记录能够大大降低重新运行实验的时间损失从而收集更多数据，并降低等待 App Store 发布周期的时间损失，等等。

所有的实验者都被要求记录他们的实验。你在改变什么？你期望的结果是什么？期望的“受众规模”（将要看到这一特性的用户比例）？收集

这些数据不仅保证了实验者考虑过这些问题，而且让我们能够建立一个制度性学习的资料库——一份已经实验的正式记录和实验结果，包括负面结果。我们可以用此提醒后面的实验。

实验者还可以利用实验牧羊人的优势。实验牧羊人都是经验丰富的工程师和数据科学家，负责评审实验假设和拟定的指标，以减少实验出错的几率。这是可选的，建议不具有约束力。随着人们对实验正确配置、跟踪正确的指标、能够正确分析实验结果有了更多的信心，该项目也从参与者中收到了大量的反馈。

一些团队也会举行每周例会，在例会上评审实验结果以便决定哪些应该哪些不应该发布给更广泛的受众。这有助于解决 cherry-picking 和误解统计显著性的问题。重要的是要注意这不是一场“给我一个理由说不”的例会——我们已经明确，“红色”实验发布，“绿色”实验不发布。这里重要的是坦诚及明确我们引入变更的期望和结果，而不是容忍停滞和奖励短期收益。引入这些评审显著提高了我们发布的变更的整体质量。同时这也是个有趣的会议，因为我们能够看到团队正在进行的所有工作以及人们对产品的看法。

另一个我们经常使用的实践是使用“holdbacks”，如果可能——向 99%（或者其它高百分比）的用户推送该特性，并观测随着时间推移关键指标是如何偏离被阻止的 1% 的。这使得我们能够快速迭代和发布，同时密切关注实验的长期影响。这也是一种很好的验证实验中真正实现的收益的方法。

实验培训

确保实验者提防陷阱最有效的方法之一就是培训他们。Twitter 数据科学家会开设多门实验和统计直觉课程，其中统计直觉是所有新工程师加入公司后的前几周都要参加的课程。目标是使工程师、PM、EM 和其它一些角色熟悉实验过程、警告、陷阱和最佳实践。增强实验品质和陷阱的意识有助于我们避免在可避免的错误和误解上浪费时间，让人们更快洞察和改进节奏和质量。

即将推出

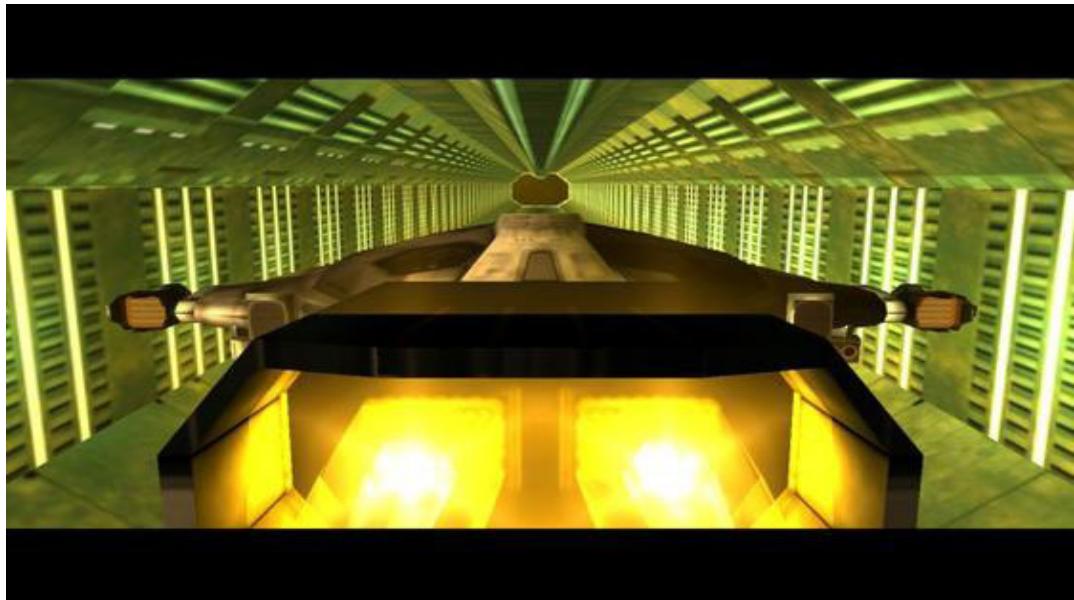
在后面的文章中，我们将介绍我们的实验工具 DDG 是如何工作的；我们将直接介绍一些我们遇到的有趣的统计问题——检测有偏差的分桶，使用（或不使用）第二控制（control）进行合理性检查，自动确定合适的 bucket 大小，基于会话的指标和处理异常值。

致谢

感谢 Lucile Lu, Robert Chang, Nodira Khoussainova 和 Joshua Lande 对此文章的反馈。很多人都为 Twitter 实验背后的理念和工具作出了贡献。我们想特别感谢 Cayley Torgeson, Chuang Liu, Madhu Muthukumar, Parag Agrawal, 和 Utkarsh Srivastava。

集群调度框架的架构演进之路

作者 杨峰



【编者的话】 集群架构是现代数据中心非常重要的组件，在最近几年中有长足发展。架构也从单体式设计转向更加灵活、去中心化和分布式设计。然而，许多现代开源实现仍然是单体式设计或者缺少很多功能，而这些功能对实际用户非常有用。本文经授权转载自 DockOne 社区。

这篇博客是关于大型机群任务调度系列的第一篇，资源调度在 Amazon、Google、Facebook、微软或者 Yahoo 已经有很好实现，在其它地方的需求也在增长。调度是很重要的课题，因为它直接跟运行集群的投入有关：一个不好的调度器会造成低利用率，昂贵投入的硬件资源被白白浪费。而光靠它自己也无法实现高利用率，因为资源利用相抵触的负载必须要仔细配置，正确调度。

架构演进

这篇博客讨论了最近几年调度架构如何演进，为什么会这样。图一演示了不同的方法：灰色方块对应一个设备，不同颜色圈对应一个任务，有“S”的方形代表一个调度器。箭头代表调度器的调度决定；三种颜色代表不同工作负载（例如，网站服务，批量分析，和机器学习）。

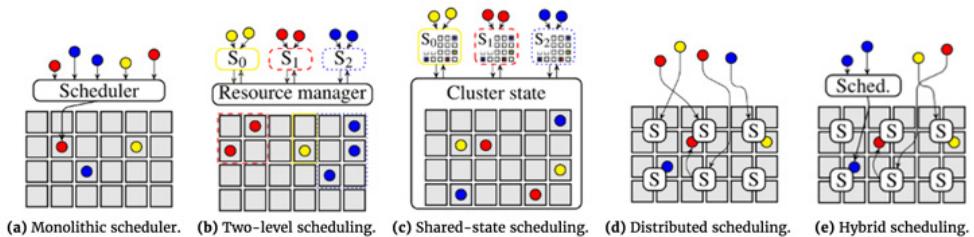


Figure 1: Different cluster scheduler architectures. Gray boxes represent cluster machines, circles correspond to tasks and S_i denotes scheduler i .

图 1：不同调度架构，灰色框代表集群设备，圆圈代表任务， S_i 代表调度器。（a）单体式调度器（b）二级调度（c）共享状态调度（d）分布式调度（e）混合式调度

许多集群架构，例如大量高性能计算（HPC），使用的是 Borg 调度器，它跟 Hadoop 调度器和 Kubernetes 调度器完全不同，是单体式调度。

单体式调度

单一调度进程运行在一台物理机内（例如 Hadoop V1 中 JobTacker，Kubernetes 中的 kube-scheduler），将任务指派给集群内其它物理机。所有负载都服从于一个调度器，所有任务都通过这一个调度逻辑运行（参见图一 a）。这种架构最简单格式唯一，在此基础上发展起了很多负载的调度器，例如 Paragon 和 Quasar 调度器，采用机器学习方法来避免不同负载之间资源竞争。

今天集群都运行着不同类型的应用（与之相对应的是 MapReduce 早期作业场景），然而，采用单一调度器来处理这么复杂异构负载会很棘手，有几个原因：

1. 调度器必须区分对待长期运行服务作业和批量分析作业，这是合理的需求。
2. 因为不同应用有不同的需求，催生调度器内加入更多功能，增加业务逻辑和部署方式。
3. 调度器处理任务顺序变成一个问题：如果调度器不仔细设计，队列效果（例如线头阻塞 head-of-line blocking）和回滚会成为问题。

总之，这听起来像是给工程师带来噩梦，调度器维护者面对的没完没了的功能请求证实了这点。

二级调度

二级调度通过将资源调度和任务调度分离解决了这个问题，这使得任务调度逻辑不仅可以根据不同应用要求而进行裁剪，而且保留了在集群之间共享资源的可能性。尽管侧重点不同，Mesos 和 YARN 集群管理都使用了这种方法：Mesos 中，资源是主动提供（offer）给应用层调度，而 YARN 则允许应用层调度请求（request）资源（，并且随后接受被分配资源）。图一 b 展示了这一概念，作业负责调度（S0-S2）跟资源管理器交互，资源管理器则给每个作业分配动态资源。这一方案赋予客户灵活调度作业策略的可能性。

然而，通过二级调度解决问题也有问题：应用层调度将资源全局调度

隐藏起来，也就是说，不再能看到全局性的可选资源配置。相反，只能看到资源管理器主动提供 (offer, 对应于 Mesos) 或者请求 / 分配 (request/allocate, 对应于 YARN) 给应用的资源。由此带来一些问题：

- 重入优先权（也就是高优先权会将低优先权任务剔除）实现变的很困难。在基于 offer 模式，被运行中任务占用的资源对高一级调度器不可见；在基于 request 模式，底层资源管理器必须理解重入策略（跟应用相关）。
- 调度器不能介入运行中业务，有可能减低资源使用效率（例如，“饥饿邻居”占据了 IO 带宽），因为调度器看不见他们。
- 应用相关调度器更关注底层资源使用的不同情况，但是他们唯一选择资源的方法就是资源管理器提供的 Offer/request 接口，这个接口很容易变的很复杂。

共享状态架构

共享状态架构通过采用半分布式模式来解决这个问题，这种架构下集群状态多副本会被应用层次调度器独立更新，如图一 C 中所示。一旦本地有更新，调度器发布一个并发交易更新所有共享集群状态。有时候因为另外一个调度器发出了一个冲突交易，交易更新有可能失败。

最重要的共享状态架构实例是 Google 的 Omega 系统，以及微软的 Apollo 和 Hashicorp 的 Nomad 容器调度。这些例子中，共享集群状态架构都是通过一个模块实现，也就是 Omega 中的“cell state”，Apollo 中的“resource monitor”，以及 Nomad 中的“plan queue”。Apollo 跟其他两个不同之处在于共享状态是只读的，调度交易直接提交到集群设

备；设备自身会检查冲突，来决定接受或者拒绝更新，使得 Apollo 即使在共享状态暂时不可用情况下也可以继续执行。

逻辑上来说，共享状态设计不一定必需将全状态分布在其它地方，这种方式（有点像 Apollo）每个物理设备维护自己的状态，将更新发送到其它感兴趣的代理，例如调度器，设备健康监控，和资源监控系统。每个物理设备本地状态就成为一个全局共享状态的“沟通片”（shard）。

然而，共享状态架构也有一些缺点，必须作用在稳定的（过时的，stale）信息（这点跟中心化调度器不同），有可能在高竞争情况下造成调度器性能下降（尽管对其他架构也有这种可能）。

全分布式架构

看起来这种架构更加去中心化：调度器之间没有任何协调，使用很多各自独立调度器响应不同负载，如图一 d 所示。每个调度器都作用于自己本地（部分或者经常过时的【stale】）集群状态信息。典型的，作业可以提交到任何调度器，调度器可以将作业发布到任何集群节点上执行。跟二级调度器不同的是，每个调度器并没有负责的分区，全局调度和资源分区是服从统计意义和随机分布的，这有点像共享状态架构，但是没有中央控制。

尽管说去中心化底层概念（去中心化随机选择）是从 1996 年出现，现代意义上分布式调度应该是从 Sparrow 论文开始的，当时有一个讨论是：合适粒度（fine-grained）任务有很多优势，Sparrow 论文的关键假设是集群上任务周期可以变得很短；接下来，作者假定大量任务意味着调度器必须支持很高通量的决策，而单一调度器并不能支持如此高的决策量（假

定每秒上百万任务量），Sparrow 将这些负载分散到许多调度器上。

这个实现意义重大：去中心化理论上意味着更多的仲裁，但是这非常适合某类负载，我们会在后面的连载中讨论。现在，足够理由证明，由于分布式调度是无协调的，因此相对于复杂单体式调度，二级调度或者分布状态时调度，更适合于简单逻辑。例如：

1. 分布式调度是基于简单的“时间槽（slot）”概念，将每台设备分成 n 个标准时间槽，同时运行 n 个并发任务，尽管这种简化忽略了任务资源需求是各自不同的事实。
2. 在任务端（worker side）使用服从简单服务规则的队列方式（例如 Sparrow 中 FIFO），这样调度器的灵活性受到限制，调度器只需决定在哪台设备上将任务入队。
3. 因为没有中央控制，分布式调度器对于全局变量设置（例如，fairness policies 或者 strict priority precedence 等）有一定难度。
4. 因为分布式调度是基于最少知识做出快速决策而设计，因此无法支持或承担复杂应用相关调度策略，因此避免任务之间干扰，对于全分布式调度来说很困难。

混合式架构

混合式架构是为了解决全分布式架构缺陷，最近（发端于学院派）提出的解决方式，它综合了单体式或者共享状态的设计。这种方式，例如 Tarcil，Mercury 和 Hawk，一般有两条调度路径：一条是为部分负载设计的分布式路径（例如，短时间任务或者低优先级批量负载），另外一条集

中式调度，处理剩下负载，如图一 e 所示。对于所描述的负载来说，混合架构中发生作用的调度器都是唯一的。实际上，据我所知，目前还没有真正的混合架构部署于生产系统中。

实际意义

不同调度架构相对价值，除了有很多研究论文外，其讨论不仅仅局限在学院内，从行业角度对于 Borg, Mesos 和 Omega 论文的深入讨论，可以参见 Andrew Wang 的专业博客。然而，很多以上讨论的系统都已经部署在大型企业生产系统中（例如，微软的 Apollo, Google 的 Borg, Apple 的 Mesos），反过来这些系统激励了其它可用开源项目。

如今，很多集群系统运行容器化负载，因此有一系列面向容器的“调度框架”（Orchestration Frameworks）出现，他们跟 Google 以及其它被称为“集群管理系统”类似。然而，很少关于这些调度框架和设计原则的讨论，更多是集中于面向用户调度的 API（例如，这篇 Armand Grillet 的报道，比较了 Docker Swarm, Mesos/Marathon 和 Kubernetes 的默认调度器）。然而，很多客户既不懂不同调度架构的不同，也不知道哪个更适合自己的应用。

图 2 展示了一部分开源编排框架的架构和调度器支持的功能。图表底部，也包括 google 和微软闭源系统作比较。资源粒度一列展示调度器分配任务给固定大小时间槽，还是按照多维需求（例如 CPU, memory, 磁盘 I/O, 网络带宽等）分配资源。

决定一个合适调度架构主要因素在于你的集群是否运行一个异构

	Framework	Architecture	Resource granularity	Multi-scheduler	Pluggable logic	Priority preemption	Re-scheduling	Oversubscription	Resource estimation	Avoid interference
O P E N	Kubernetes	monolithic	multi-dimensional	N ^[v1.2, DD, issue]	Y ^[DD]	N ^[Issue]	N ^[Issue]	Y ^[DD]	N	N
	Swarm	monolithic	multi-dimensional	N	N	N ^[Issue]	N	N	N	N
	YARN	two-level	RAM/CPU slots	Y	Y ^[framework-lvl.]	N ^[IIRA]	N	N ^[IIRA]	N	N
	Mesos	two-level	multi-dimensional	Y	Y ^[framework-lvl.]	N ^[IIRA]	N	Y ^[v0.23, Doc]	N	N
	Nomad	shared-state	multi-dimensional	Y	Y	N ^[Issue]	N ^[Issue]	N ^[Issue]	N	N
	Sparrow	fully-distributed	fixed slots	Y	N	N	N	N	N	N
C L O S E D	Borg	monolithic ^[2]	multi-dimensional	N ^[2]	N ^[2]	Y	Y	Y	Y	N
	Omega	shared-state	multi-dimensional	Y	Y	Y	Y	Y	Y	N
	Apollo	shared-state	multi-dimensional	Y	Y	Y	Y	N	N	N

图 2：常用开源编排框架分类和功能比较，以及与闭源系统比较

(或者说混合的) 负载。例如，前端服务（例如，负载均衡 Web 服务和 memcached）和批量数据分析（例如，MapReduce 或者 Spark）混合在一起，这种组合对于提高系统利用率是有意义的，但是不同应用需要不同调度方式。在混合设定下，单体式调度很可能导致次优任务分配，因为基于应用需求，单体调度逻辑不能多样化，而此时二级或者共享状态调度可能更加适合一些。

许多面向用户服务负载运行的资源一般是满足容器的峰值需求，但是实际上资源都是过分配的。这种情况下，能够有机会降低给低优先级负载过分配资源对高效集群来说是关键。尽管 Kubernetes 拥有相对成熟方案，Mesos 是目前唯一支持这种过分配策略的开源系统。这个功能未来应该有

更大改善空间，因为根据 Google borg 集群来看很多集群利用率任然小于 60–70%。后续博客我们将就资源预估，过分配和有效设备利用等方面展开讨论。

最后，特殊分析和 OLAP 应用（例如，Dremel 或者 SparkSQL）非常适合全分布式调度。然而，全分布式调度（例如 Sparrow）内置相对严格功能设置，因此当负载是同构（也就是，所有任务同时运行）、配置时间（set-up times）很短（也就是，任务调度后长时间运行，如同 MapReduce 应用任务运行于 YARN）、任务通量（churn）很高（也就是，许多调度决定必须很短时间内做出）时非常合适。我们将在下一个博客中详细讨论这些条件，以及为什么全分布式调度（以及混合架构中分布模块）只对这种应用场景有效。

现在，我们可以证明分布式调度比其他调度架构更加简单，而且不支持其他资源维度，过分配或者重新调度。

总之，图二中表格表明，相对于更高级但是闭源的系统来说，开源框架仍然有很大提升空间。可以从如下几方面采取行动：功能缺失，使用率不佳，任务性能不可预测，邻居干扰（noisy neighbours）降低效率，调度器精细调整以支持某些客户忒别需求。

然而，也有很多好消息：尽管今天还有很多集群仍然使用单体式调度，但是也已经有很多开始迁移到更加灵活架构。Kubernetes 今天已经可以支持可插入式调度器（kube-scheduler pod 可以被其它 API 兼容调度 pod 替代），更多调度器从 1.2 版本开始会支持“扩展器”提供客户化策略。Docker Swarm，据我理解，在未来也会支持可插入式调度器。

下一步

下一篇博客将讨论全分布式架构对于可扩展式集群调度是否关键技术创新（反对声音说：不是必须的）。然后，我们会讨论资源适配策略（提高利用率），最后讨论我们 Firmament 调度平台如何组合和共享状态架构和单体式调度质量，以及全分布调度器性能问题。



如何成为架构师？ 7 个关键的思考、习惯和经验

作者 秦迪



工作了挺久，发现有个挺有意思的现象，从程序员、高级程序员，到现在挂着架构师、专家之类的头衔，伴随着技术和能力的提高，想不明白的事情反而越来越多了。这些疑问有些来自于跟小伙伴交流，有些是我的自问自答，有些到现在也想不清楚，这篇文章就来写一写这些问题。

如何更高效的学习

很多新人程序员一开始在学习上找不到方向，但我想在渡过了一段时间的新手期之后这类问题大多都会变得不再那么明显，工作的方向也会逐渐变得清晰起来。

但是没过多久，能了解到的资料就开始超过每天学习的能力，像是买了没看的书、收藏没读的贴、mark 了之后再也没有关注过的文章越积越多，更别提每天面对各种技术分享或者微博里的新鲜玩意了。

大多数人每天能留给自己学习的时间有限，这个阶段如何提升学习效率就成了要解决的重点。

说说自己提升学习效率的心得，其实非常简单：体系化的学习。

我曾经很喜欢看一些博客或者是一些“看起来”比较通俗易懂的文章，每天在微博微信里刷到什么技术文章就 Mark 下来，基本上几分钟就能读完。可一段时间下来，虽然读了不少东西，但是还是有种在原地打转的状态，并没有感受到有什么实际的提高。

最后实在忍不住，抱着厚书硬啃了一遍，突然有种豁然开朗的感觉：读书时自己学到的是一张完整的知识网络，每个知识点和其它内容相互联结和区别。这种全方位的理解比起一篇篇独立的文章，不知要高到哪里去了。

而读了一段时间书之后，渐渐原本不在一个体系之内的知识也会慢慢联系起来，比如说后端服务的开发，简单梳理一下，就成了图 1 这样。

在重复了几次痛苦的学习 - 梳理过程后，再去看一些独立的文章或者资料往往事半功倍，因为能在体系内找到相对应的知识，甚至有时候一本书里一页只需要看一句话，点破那层窗户纸，就可以掌握新的知识。

你是怎么知道这些的

工作中总是会遇到各种各样的问题，有几次把问题处理过程总结了一

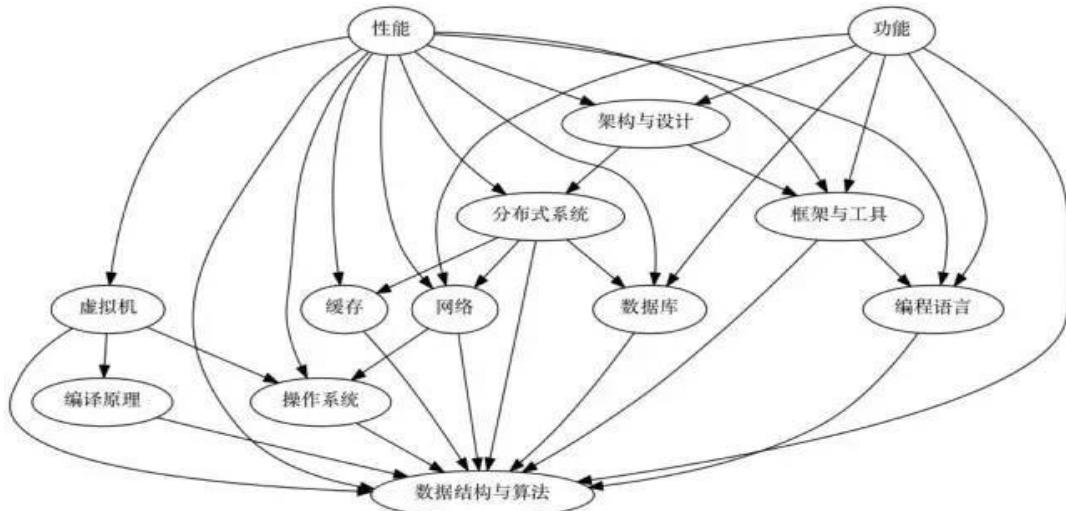


图 1

下，发了出来，之后就像滚雪球一样，有越来越多的小伙伴来咨询问题，比如说：

- 前一阵帮忙排查一个性能问题，系统压力稍微一大就会频繁 Full GC，压力降低之后又恢复了。
- 某个小伙伴接入代码质量检查系统之后发现每次构建会报一个莫名其妙的错误，打不了包。
- 某次代码有 Bug，小伙伴跑来问我 Git 怎么才能回滚代码。

每次查完这种问题的时候，一些刚毕业没多久小伙伴们就会用一种崇拜的眼神看着我，然后大多会问：“你是怎么知道这些的？”

实际上，虽然我一直在不断的学习，但是面对工作中无穷无尽的新问题，大部分问题还是会命中我没有掌握的那部分区域。每次有人问到我不了解的知识时我都会非常开心：还有什么比带着问题学习更有效率的学习方法呢？

而且幸运的是，在建立了自己的知识体系的基础上，学习新的知识通

常都能很快的上手，解决一个问题往往只需要多了解一个知识点而已。

举个例子，频繁 Full GC 的问题，以前查过很多次 GC 的问题，大多数是 Java 程序或 JVM 内存泄露问题，而这次内存没有泄露，GC 吞吐量也正常，那么我只需要查一下如何查看一段时间内创建的最多的对象的方法就可以了。

回到刚才的问题，小伙伴们问我：“你是怎么学到这些的知识的？”

答案是：在你问我问题之后现学的。

架构师应不应该写代码

似乎隔三差五就能看到一些关于架构师应不应该写代码的文章。我是属于写代码派，因为我本身就喜欢写代码。但是，当工作职责发生变化之后，如何保持写代码和其它工作之间的平衡就成了问题。

从个体效率上来看，我自己亲自写代码，和很多人相比没有什么绝对优势，甚至有些人码代码的速度比我还快一些。

但作为架构师，参与写代码还是会有一些不大不小的收益。

一般来说合格的程序员对于明确分配的任务会完成的很好，但是大部分情况下“架构”这个词意味着架构师并不会涉及太多细节，架构图和代码实现之间总还是有些距离，你无法保证所有人都会正确的理解你的设计，或者是程序员写代码时遇到障碍时会立刻想出足够优雅的解决方案。

之前写过一篇关于烂代码的文章，大部分烂代码并不是架构师的设计问题，如果程序员没能很好的理解设计或者是经验不足，往往会做出一些非常匪夷所思的东西。比如我见过刚毕业的程序员为了防止模块耦合而

将耦合的代码又拷贝了一份，或者为了“优化性能”而尽量把所有逻辑写在一个函数里。

如果不能及时发现并改正这些问题，那么这些地方就会变成“正确的错误代码”，或者“不是我写的”代码，或者“我靠我也看过那段代码”之类足以被挂上耻辱柱的玩意。这种问题算是架构师的责任吗？作为一个视名声如命的架构师，我认为是的。

在我看来，写代码的架构师更像是在做后勤保障的工作：在代码中第一时间发现可能存在的问题，向其他人提出警告，或是给予其他人改进的意见，必要的时候或是给其他人演示一下正确的姿势。

大部分情况下我作为架构师并不需要揽下“核心模块”开发这种工作，毕竟我能调配的时间太零散了，效率难以保证，很多人在专注的情况下比我做的好很多，我只需要保持大局观需要适度参与就可以了。

总的来说，架构师和程序员在某些方面上有点像产品经理和用户的关系，大部分程序员并不会主动告诉你他们想要什么、哪里需要优化，甚至自己也不知道这些。想要做出好的产品，捷径之一就是跟用户做同样的事情。

实践：开会是个技术活吗

我觉得应该没有人喜欢开会，身为一个程序员，没有几个人的志向是当什么职场交际花。

但是会议邀请就这么一个个的跳了出来：开发需求要跟产品开会、项目方案要跟技术开会、新人转正要去开评审会、别的公司来了几个大牛正

在开分享会、出了故障要开总结会、小组有周会、部门有周会，大项目每周开两次碰头会不过分吧？小项目启动的时候开个会不过分吧？调试的时候发现有个坑大家赶紧讨论讨论吧？

有时候参加的会议整场下来跟我毛关系都没有，全程神游俩钟头，最后突然有人一拍桌子：“还有问题没？好，散了！”“

也有可能有个什么会没叫你，过了俩礼拜突然收到封邮件催开发进度，”当时那个会你没参加，大家都说应该是你们做……你没看会议纪要吗？”“

吐槽了这么多，但我还是认为开会是个技术活，对于架构师来说尤其如此。

大多数技术人员开会并不是那种新闻里的工作汇报或者长者们的会议，他们真的需要通过开会议论一个具体方案，或者解决什么具体问题。可惜的是我参加过很多会议，大多数的会议都是在毫无意义的交流中浪费时间：几方人坐在一个屋里互相说一些对方理解不了的话，最后得出一个“我们会后再捋一捋”之类的结论。

这并不是会议才有的问题，在程序员日常的沟通中，也有很多人并不懂得如何交流，比如偶尔会收到一些写的非常认真的邮件，打开之后是密密麻麻的一屏幕文字，但是从第一句开始就不知道他在说什么，后面的东西连看的动力都没有了。

大多数时候，沟通的核心不是你说了什么，而是你想要让对方了解什么、让他做什么。良好的沟通能在工作中显著提升效率，但很多人忽略了这个事情。

想要恰到好处的进行沟通是一件不那么轻松的事情，但是简单来说有几条原则：

1. 确保各方对背景的理解一致，比如开会之前先简单通过邮件交流一下，对新加入会议的人花个 30 秒钟做个前情提要，或者在讨论过程中让对方说一下他的理解。
2. 去掉对方不能 / 不需要理解的内容，比如跟产品说“这个队列在高并发下因为锁的实现有问题导致 CPU 性能瓶颈”不如改成“我们发现了性能问题，持续 10 分钟了，10 万用户收不到运营发的无节操广告，大概 5 分钟后扩容解决”。
3. 确保在对方失去注意力前尽快说出重点，比如排查问题的总结邮件，如果第一段是这样：“某某框架内部使用的是 xxx 技术，这个技术的架构是这样：blabla”，那么对方可能完全不知道你在讲什么。可以换成这样：“我发现了某某框架的 bug，需要尽快升级，否则在 xxx 情况下有可能会出现 yyy 问题，具体排查过程如下：blabla”。
4. 不要说没有意义的内容浪费其他人的时间，比如“这需求做不了”或者“这里不可能出 Bug”，没有人想听到这些废话。

为什么别人的系统总是那么烂

很多程序员解决问题的能力很强，说要解决一个什么问题，下午就能写出几百行代码把功能实现了。但是做出来的东西有种少考虑了什么东西的感觉，我花了挺久去想一个词来形容“这个东西”，最后想出了个勉强

可以表达的词：程序的生命力。

大部分程序都能实现功能，但是如果把“时间”这个也作为一个考虑的维度的话，就会意识到一个合格的项目需要考虑更多的东西：更通用的使用方式、易于理解的文档、简单而易于扩展的设计，等等。而想要毁掉程序的生命力也很简单：做的更复杂，更定制化，让更少的人参与。

我跟很多程序员提过程序的生命力，比如说要让自己写的工具的操作方式跟其它 Linux 命令类似，或者要用一些更容易理解但不是性能最优的设计方式，又或者要他去参考现在业界主流的做法，很多人认为提这种需求的意义不大，我觉得这里还是举个例子吧。

很多公司应该都会有一些遗留系统，它们庞大、笨重、难用、几乎无法维护，所有人都在抱怨这些系统，并且每天都在想方设法换掉那些遗留系统。但是一段时间过去之后，又会发现身边的新人又开始吐槽当时替代遗留系统的那个系统了。

“大多数系统当初都很好使，功能当时够用，扩展性看起来也可以，但是这些系统都是开发的人离职之后变坏的。”

还有更好的办法吗

成为技术专家之后的工作可以说是痛并快乐着，会有很多人找你咨询问题，另一方面，会有太多人找你咨询问题。

甚至有一段时间我每天的工作就是解答问题，小到工具使用中到疑难 Bug，大到架构设计，从早上到晚上基本都是在给各种各样的小伙伴提供咨询服务。

我很快发现有些地方不对头：有些问题实在是太简单了，以至于我甚至都不用思考就可以给出答案，为什么会有这种问题？

后来我在每次回答之前先问一句：“你还有更好的办法吗？”

一小部分人立刻能给出优化后的版本，甚至我连续问几次之后，他能给出好几个优化后的版本；另小一部分人会斩钉截铁的说优化不了了，就这样了。但是大部分人会犹犹豫豫的说出一些完全不着调的回答。

后来我改成在每次回答之前先问两句：“你要解决什么问题？”

“还有更好的办法吗？”

效果好了很多，很多小伙伴发现要解决的问题并不复杂，只是做法跑偏了。

再后来我改成了在每次回答之前先问三句：“他们要你解决什么问题？”“你解决的是什么问题？”“还有更好的办法吗？”

现在第三句已经很少问到了。

成为架构师最困难的门槛是什么

跟一些程序员交流的过程中，有不少人问我怎么成为一名牛逼的架构师。

我最近几年面试的人挺多，发现一个有意思的现象：很多人自称架构师的人跟你讲一个架构时简直滔滔不绝，各种技术名词像是说相声一样从他嘴里说出来，三句话不离高并发大数据，但是稍微追问一下，就会发现很多基本概念的缺失，例如自称精通高并发的人说不清楚他所谓的高并发系统的瓶颈在哪里，自称精通架构设计的人说不明白他的系统怎么保证高

可用，自称超大数据量的系统实际上只有不到 100 万条数据，等等。

架构师虽然听起来很高大上，但本质上仍然是工程师，不是科学家，也不是忽悠人的江湖骗子。学习再多，也需要实践落地。设计架构方案更多的是在做一些抽象和权衡：把复杂的需求抽象成简单的模型，从功能、性能、可用性、研发成本等方面规划如何构建一个系统，这些内容需要更多的实践练习。

很多人没有工作在类似微博平台这种天天需要接触架构设计的地方，而很多公司没有架构方面的工作可供他们练级，于是就想办法从理论上下功夫，这类人的特征非常明显：在信息不足，甚至不了解实际场景的情况下就开始做架构设计，这种所谓的架构往往理解比较肤浅，经不住推敲。

每年招人之后我们都会做一些针对新人的架构方面的培训，课程材料基本上包括了高可用架构相关的主要方面，但是学完这些材料之后就能成为独当一面的架构师了吗？并没有。相反，这仅仅是开始，新人真正做了几个并发量上万的系统之后才算是正式入门：面对压力时才会懂得权衡，走过弯路之后才会寻找捷径。

所以我认为在架构师（和其它很多）的工作中最重要的部分是实践，夸夸其谈很容易，与其拽一些技术名词，不如把你正在做的系统真正的做好。

我和大牛之间有多少距离

跟很多人一样，刚毕业时我觉得作为程序员，只要努力，加上少许天赋便可以获得一些成绩。

工作一段时间后，对自己和其他人的认识也越来越清晰，逐渐的发现程序员之间的差距或许比人和猴子之间的差距还大，接受这个事实这让我郁闷了很久。

再过一段时间，发现自己已经能够客观的评价自己的能力，也意识到了距离并不是那么重要，只要想办法跑的更快，就足够了。

另外，我所在的部门正在招聘 Java 工程师和实习生。在这里，你可以和高手一起工作、交流和学习。在这里，上到架构设计，下至内核原理，你所知道的都将有用武之地。在这里，鼓励研究和实践新技术。在这里，MacBook 标配，给你最好的待遇。感兴趣的同学请发送简历到 qindi@staff.weibo.com。

作者介绍

秦迪，微博平台及大数据技术专家，13 年加入微博，负责微博平台通讯系统的设计和研发、微博平台基础工具的开发和维护，并负责微博平台的架构改进工作，在工作中擅长排查复杂系统的各类疑难杂症。

爱折腾，喜欢研究从内核到前端的所有方向，近几年重点关注大规模系统的架构设计和性能优化，重度代码洁癖：以 Code Review 已任，重度工具控：有现成工具的问题就用工具解决，没有工具能解决的问题就写个工具解决。业余时间喜欢偶尔换个语言写代码放松一下。

数人云

“下一代 DCOS ”

基于 Mesos 的大规模 Docker 生产环境



秒级扩容

秒级扩展 1000 个容器实例
轻松应对高并发



混跑多种应用

将 Docker 容器化应用与
Spark 等大数据应用混跑
在同一集群 提高资源利用率



支持混合云环境

支持公有云，私有云，混合云
支持物理机，虚拟机



一键部署

一键部署 Spark, Hadoop 等
分布式应用



简化运维

统一监控各种应用和集群资源
简化运维管理



保障高可用

自动迁移故障服务器上的应用实例
保障服务高可用



关注数人云，获取更多 Docker Mesos 实践

InfoQ 中文站 迷你书



架构师 月刊 2016年3月

本期主要内容：2016年会成为Java EE微服务年吗？Netflix Spinnaker：实现全局部署，.NET开源现状，Oracle宣布：2017年将废弃Java浏览器插件，针对架构设计的几个痛点，我总结出的架构原则和模式，问答系统的前世今生，每个架构师都应该研究下康威定律，电商网站的初期技术选型。



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。



架构师特刊 架构漫谈

架构漫谈是由资深架构师王概凯Kevin执笔，InfoQ 架构垂直社群策划的系列专栏，专栏以Kevin的架构经验为基础，逐步讨论什么是架构、怎样做好架构、软件架构如何落地、如何写好代码等问题。