

架构师

ARCHITECT



推荐文章 | Article

- 大数据框架对比：
Hadoop、Storm、Samza、Spark和Flink
- 亿级用户PC主站的PHP7升级实践

理论派 | Theory

- 从容器想到去IOE、去库存和独角兽

观点 | Opinion

- Nginx何时取代Apache
- Web不是未来会赢，而是已经赢了



卷首语

什么样的人可以称为“架构师”

饿了么大前端部门总监 林建锋 (Sofish)

我曾问过很多自称热爱代码的程序员的发展规划，大多都回答说期望成为一名架构师。而在招聘一方，有的团队会过滤掉多次提起架构一词而一点不提具体内容的简历。可见，虽然在大多数程序员眼里，架构师是神圣的，但又不得不承认事实是：“架构”和“架构师”是最常被滥用的。那些写能 PPT 而不能写代码的人，只做和事佬而不考虑软件快、稳、便捷的人，都称不上做“架构”更别提“架构师”。

那么什么样的人可以称为“架构师”？

据称架构一词源于建筑行业，架构师这个职位，不管是前端还是后端，职责是相同的。而用规划一次房屋的装修来描述架构师这个职位的职责是非常合适的。

建立一套 Web API 就像在定装修风格。要选择注重重 CRUD 的 RESTful 式，还是请求自定义性更强的 GraphQL 式，又或者是简单的 JSON-RPC 式，这就像装修风格是选要简洁的日式、粗犷的美式还是奢华的欧式。定方向和选型这件事无处不在，架构师必须根据实际需求，做各种决策，为后面各部分整体结合打好基础。

灯光、墙面、家具等各个部分都需要根据风格精心设计、执行和不断修正，才可能达到原定目标，架构也一样。拿光线控制来说，施工人员可能会忽略你注重的一些细节：暖色的书房氛围；明亮且能切到影院模式的客厅；装在合适位置才不会刺眼的背景灯。在每个环节的执行上，架构师既要设计，又要保证对每个角色充分理解，必要时不排除动手编写重要环节的功能，而在经验或考虑不足的点上一旦出现问题就必须迅速调整。空有一个好的设计而没有好的执行，是非常让人惋惜的。

值得一提的是，选用最好的卫浴用品、最贵的过滤器并不是获得最佳洗浴室体验的关键点。同样，软件架构并不是说把每个部分做到最好再拼凑起来就能达到佳效果。最好洗浴室体验的关键点在于折中和妥协。例如，在水压不是特别高的情况下，把过滤器安装在总闸虽然能让用水达到最健康的状态，但会导致淋浴的水压不够，进而使体验大打折扣。把过滤器安装在厨房出水口可能是最佳的平衡，既保证水压又保证了用水的健康。分成多个部分是解耦，而协作的平衡是内聚。低耦合、高内聚是架构师处理软件各部分协作的终极目标。

装修有很多细节，例如，若不喜欢晾衣服且生活在有“黄梅天”的上海，可选洗烘一体机；房子面积不大，可选扩展型家具；对通风质量要求比较高，可安装新风系统。软件架构也需要考虑很多细节，例如客户需求、实际环境、技术可用黑科技之类、安全、重用、扩展等。而这些细节方面的考虑，并不是一个刚入门的新人能做到的。

总的来说，称得上架构师的人，必须是具备丰富系统设计经验且能保证设计执行的设计师和决策者；必须参与设计、开发执行和测试但又不局限于一个角色。也许架构师并不一定全是这样，这仅代表个人看法和期望。

QCon

全球软件开发大会2017

主办方 Geekbang InfoQ
极客邦科技

[北京站]

2017年4月16-18日 / 北京·国家会议中心

大会官网: www.qconbeijing.com

议题提交: qcon@cn.infoq.com

扫码关注大会官网, 获取更多大会信息



购票咨询请联系售票经理Hanna

联系电话: 010-64738142

电子邮箱: hanna@infoq.com



纵览20大热门专题

最低优惠7折进行时

现在报名每张立减2040元

团购享受更多优惠 截至2017年1月1日



关注阿里技术公众号
先睹为快！

《不一样的技术创新》

— 阿里巴巴2016双11背后的技术

全面收录双11各领域技术精萃，首次推出电子书！

30篇纯技术干货，共计10万余字巨著！

数十位顶尖技术专家精心策划选题！

阿里巴巴双11一线技术团队实战经验分享！



基础设施

存 储

中间件

业务架构

大数 据

人工智 能

电商云化

交互技术

安 全

书中全面阐述阿里双11背后的技术实践与创新：

- 为了满足电商业务的复杂性，特别是应对大促当天零点峰值，而被称为双11稳定性保障核武器的全链路压测；
- 充分发挥云计算弹性能力，实现资源快速腾挪，支撑交易峰值每秒17.5万笔、支付峰值每秒12万笔的混合云弹性架构；
- 在双11当天实现万亿级消息流转第三代分布式消息引擎RocketMQ；
- 交易核心应用容器化，撑起双11交易下单峰值，充分解放资源的超大规模Docker化技术；
- 支撑全球最大规模在线交易的数据实时和离线计算能力，包括承载阿里巴巴集团核心大数据的离线计算平台MaxCompute，以及双11保证每秒处理亿条日志的计算能力、毫秒级的计算延迟的实时流计算平台StreamCompute；
- 阿里人工智能在搜索、推荐以及客服场景下的创新应用：人工智能赋能的数千家品牌商家店铺的个性化运营和粉丝会员的精准营销；基于深度强化学习和在线自适应学习的推荐算法创新；以“智能+人工”模式提供智能导购、服务、助理拟人交互的阿里小蜜；
- 全球第一个端对端的虚拟购物环境背后的VR技术，全面解读如何解决VR购物三大技术挑战，全面覆盖浏览、下单及支付环节；
- 首度曝光基于阿里云的淘宝移动直播解决方案，完整阐述首屏秒开、互动玩法、全链路数据监控的一体化直播开放平台；
- 全面解密双11会场页面几乎全覆盖背后的Weex技术，如何在充分保证稳定性的前提下，实现全网首屏渲染，完美践行“秒开”体验。

CONTENTS / 目录

热点 | Hot

开源搜索引擎 Elasticsearch 5.0
版本正式发布

推荐文章 | Article

京东 Nginx 平台化实践
亿级用户 PC 主站的 PHP7 升级实践

理论派 | Theory

奇谈怪论：从容器想到去 IOE、
去库存和独角兽

观点 | Opinion

Nginx 何时取代 Apache
Web 不是未来会赢，而是已经赢了

特别专栏 | Column

腾讯云 11.11：十分钟内完成弹性伸缩，流量
清洗化解 DDoS 攻击

漫画 | Comic

究竟什么样的技术 Leader 是称职的



架构师 2016 年 12 月刊

本期主编 韩 婷

提供反馈 feedback@cn.infoq.com

流程编辑 丁晓昀

商务合作 sales@cn.infoq.com

发行人 霍泰稳

内容合作 editors@cn.infoq.com

开源搜索引擎

Elasticsearch 5.0 版本正式发布

作者 谢丽

近日，Elastic 在官方博客中[宣布](#)，Elasticsearch 5.0 正式发布。该版本基于 Lucene 6.2.0，已经在 [Elastic Cloud](#) 上完成了部署。据称，这是迄今为止最快、最安全、最易用的版本。

Elasticsearch 5.0 带来了许多增强功能和新特性，主要包括：

- 索引性能：得益于多项改进，其中包括更好的数值型数据结构，索引吞吐量大幅提升。根据应用场景的不同，索引吞吐量提升在 25% 到 80% 之间。
- Ingest 节点：向 Elasticsearch 添加数据更简单了。Logstash 是一个强大的工具，而一些较小的用户只需要过滤器，不需要它所提供的众多路由选项。因此，Elastic 将一些最流行的 Logstash 过滤器（如 grok、split）直接在 Elasticsearch 中实现为[处理器](#)。多个处理器可以组合成一个[管道](#)，在索引时应用到文档上。
- Painless 脚本：Elasticsearch 中很多地方用到了脚本，而出于安全考虑，脚本在默认情况下是禁用的，这令人相当失望。为此，Elastic 开发了一种新的脚本语言 Painless。该语言更快、

更安全，而且默认是启用的。不仅如此，它的[执行速度是Groovy的4倍](#)，而且正在变得更快。Painless已经成为默认脚本语言，而Groovy、Javascript和Python都遭到弃用。要了解有关这门新语言的更多信息，请点击[这里](#)。

- 新数据结构：Lucene 6带来了一个新的Points 数据结构[K-D树](#)，用于存储数值型和地理位置字段，彻底改变了数值型值的索引和搜索方式。基准测试表明，Points将查询速度提升了36%，将索引速度提升了71%，而占用的磁盘和内存空间分别减少了66%和85%（参见“在5.0中搜索数值”）。
- 搜索和聚合：借助即时聚合，Kibana图表生成速度显著提升。Elastic用一年的时间对搜索API进行了[重构](#)，Elasticsearch现在可以[更巧妙地执行范围查询](#)，只针对已经发生变化的索引重新计算聚合，而不是针对每个查询从头开始重新计算。在搜索方面，默認的相关性计算已经由TF/IDF换成了更先进的BM25。补全建议程序经过了完全重写，将已删除的文档也考虑了进来。
- 更友好：Elasticsearch 5.0更安全、更易用。他们采用了“尽早提示”的方法。如果出现了问题，则新版本会及早给出提示。例如，Elasticsearch 5.0会严格验证设置。如果它不能识别某项设置的值，就会给出提示和建议。不仅如此，集群和索引设置现在可以通过null进行解除。此外，还有其他的一些改进，例如，[rollover](#)和[shrink](#) API启用了一种新的模式来管理基于时间的索引，引入新的[cluster-allocation-explain API](#)，简化索引创建。
- 弹性：Elasticsearch分布式模型的每一部分都被分解、重构和简化，提升了可靠性。集群状态更新现在会等待集群中的所有

节点确认。如果一个“复制片（replica shard）”被“主片（primary）”标记为失败，则主片会等待“主节点（master）”的响应。索引现在使用数据路径中的UUID，而不是索引名，避免了命名冲突。另外，Elasticsearch现在进行启动检查，确保系统配置没有问题。配置比较麻烦，但如果只是试用，开发人员也可以选择localhost-only模式，避免繁琐的配置。另外，新版本还增加了[断路器](#)及其他一些[软限制](#)，限制请求使用的内存大小，保护集群免受恶意用户攻击。

此外，该版本还提供了一个底层的[Java REST/HTTP 客户端](#)，可以用于监听、日志记录、请求轮询、故障节点重试等。它使用 Java 7，将依赖降到了最低，比 Transport 客户端的依赖冲突少。而在基准测试中，它的[性能并不输于 Transport 客户端](#)。不过，这是一个底层客户端，目前还没有提供任何查询构建器或辅助器。它的输入参数和输出结果都是 JSON。需要注意的是，该版本引入了许多[破坏性更改](#)，好在他们提供了一个[迁移辅助插件](#)，可以帮助开发人员从 Elasticsearch 2.3.x/2.4.x 迁移到 Elasticsearch 5.0。如果是从更早的 Elasticsearch 版本向最新的 5.0 版本迁移，则请查阅[升级文档](#)。

大数据框架对比：Hadoop、Storm、Samza、Spark 和 Flink

作者 Justin Ellingwood 译者 大愚若智

简介

大数据是收集、整理、处理大容量数据集，并从中获得见解所需的非传统战略和技术的总称。虽然处理数据所需的计算能力或存储容量早已超过一台计算机的上限，但这种计算类型的普遍性、规模，以及价值在最近几年才经历了大规模扩展。

在之前的文章中，我们曾经介绍过有关大数据系统的常规概念、处理过程，以及[各种专门术语](#)，本文将介绍大数据系统一个最基本的组件：处理框架。处理框架负责对系统中的数据进行计算，例如处理从非易失存储中读取的数据，或处理刚刚摄入到系统中的数据。数据的计算则是指从大量单一数据点中提取信息和见解的过程。

下文将介绍这些框架。

- 仅批处理框架：

- o Apache Hadoop
- 仅流处理框架:
 - o Apache Storm
 - o Apache Samza
- 混合框架:
 - o Apache Spark
 - o Apache Flink

大数据处理框架是什么

处理框架和处理引擎负责对数据系统中的数据进行计算。虽然“引擎”和“框架”之间的区别没有什么权威的定义，但大部分时候可以将前者定义为实际负责处理数据操作的组件，后者则可定义为承担类似作用的一系列组件。

例如 Apache Hadoop 可以看作一种以 MapReduce 作为默认处理引擎的处理框架。引擎和框架通常可以相互替换或同时使用。例如另一个框架 Apache Spark 可以纳入 Hadoop 并取代 MapReduce。组件之间的这种互操作性是大数据系统灵活性如此之高的原因之一。

虽然负责处理生命周期内这一阶段数据的系统通常都很复杂，但从广义层面来看它们的目标是非常一致的：通过对数据执行操作提高理解能力，揭示出数据蕴含的模式，并针对复杂互动获得见解。

为了简化这些组件的讨论，我们会通过不同处理框架的设计意图，按照所处理的数据状态对其进行分类。一些系统可以用批处理方式处理数据，一些系统可以用流方式处理连续不断流入系统的数据。此外还有一些系统可以同时处理这两类数据。

在深入介绍不同实现的指标和结论之前，首先需要对不同处理类型的概念进行一个简单的介绍。

批处理系统

批处理在大数据世界有着悠久的历史。批处理主要操作大容量静态数据集，并在计算过程完成后返回结果。

批处理模式中使用的数据集通常符合下列特征。

- 有界：批处理数据集代表数据的有限集合
- 持久：数据通常始终存储在某种类型的持久存储位置中
- 大量：批处理操作通常是处理极为海量数据集的唯一方法

批处理非常适合需要访问全套记录才能完成的计算工作。例如在计算总数和平均数时，必须将数据集作为一个整体加以处理，而不能将其视作多条记录的集合。这些操作要求在计算进行过程中数据维持自己的状态。

需要处理大量数据的任务通常最适合用批处理操作进行处理。无论直接从持久存储设备处理数据集，或首先将数据集载入内存，批处理系统在设计过程中就充分考虑了数据的量，可提供充足的处理资源。由于批处理在应对大量持久数据方面的表现极为出色，因此经常被用于对历史数据进行分析。

大量数据的处理需要付出大量时间，因此批处理不适合对处理时间要求较高的场合。

Apache Hadoop

Apache Hadoop 是一种专用于批处理的处理框架。Hadoop 是首个在开源社区获得极大关注的大数据框架。基于谷歌有关海量数据处理所发表的多篇论文与经验的 Hadoop 重新实现了相关算法和组件堆栈，让大规模批

处理技术变得更易用。

新版 Hadoop 包含多个组件，即多个层，通过配合使用可处理批数据：

- HDFS：HDFS是一种分布式文件系统层，可对集群节点间的存储和复制进行协调。HDFS确保了无法避免的节点故障发生后数据依然可用，可将其用作数据来源，可用于存储中间态的处理结果，并可存储计算的最终结果。
- YARN：YARN是Yet Another Resource Negotiator（另一个资源管理器）的缩写，可充当Hadoop堆栈的集群协调组件。该组件负责协调并管理底层资源和调度作业的运行。通过充当集群资源的接口，YARN使得用户能在Hadoop集群中使用比以往的迭代方式运行更多类型的工作负载。
- MapReduce：MapReduce是Hadoop的原生批处理引擎。

批处理模式

Hadoop 的处理功能来自 MapReduce 引擎。MapReduce 的处理技术符合使用键值对的 map、shuffle、reduce 算法要求。基本处理过程包括：

- 从HDFS文件系统读取数据集；
- 将数据集拆分成小块并分配给所有可用节点；
- 针对每个节点上的数据子集进行计算（计算的中间态结果会重新写入HDFS）；
- 重新分配中间态结果并按照键进行分组；
- 通过对每个节点计算的结果进行汇总和组合对每个键的值进行“Reducing”；
- 将计算而来的最终结果重新写入 HDFS。

优势和局限

由于这种方法严重依赖持久存储，每个任务需要多次执行读取和写入操作，因此速度相对较慢。但另一方面由于磁盘空间通常是服务器上最丰富的资源，这意味着 MapReduce 可以处理非常海量的数据集。同时也意味着相比其他类似技术，Hadoop 的 MapReduce 通常可以在廉价硬件上运行，因为该技术并不需要将一切都存储在内存中。MapReduce 具备极高的缩放潜力，生产环境中曾经出现过包含数万个节点的应用。

MapReduce 的学习曲线较为陡峭，虽然 Hadoop 生态系统的其他周边技术可以大幅降低这一问题的影响，但通过 Hadoop 集群快速实现某些应用时依然需要注意这个问题。

围绕 Hadoop 已经形成了辽阔的生态系统，Hadoop 集群本身也经常被用作其他软件的组成部件。很多其他处理框架和引擎通过与 Hadoop 集成也可以使用 HDFS 和 YARN 资源管理器。

总结

Apache Hadoop 及其 MapReduce 处理引擎提供了一套久经考验的批处理模型，最适合处理对时间要求不高的非常大规模数据集。通过非常低成本的组件即可搭建完整功能的 Hadoop 集群，使得这一廉价且高效的处理技术可以灵活应用在很多案例中。与其他框架和引擎的兼容与集成能力使得 Hadoop 可以成为使用不同技术的多种工作负载处理平台的底层基础。

流处理系统

流处理系统会对随时进入系统的数据进行计算。相比批处理模式，这是一种截然不同的处理方式。流处理方式无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作。

流处理中的数据集是“无边界”的，这就产生了几个重要的影响：

- 完整数据集只能代表截至目前已经进入到系统中的数据总量。
- 工作数据集也许更相关，在特定时间只能代表某个单一数据项。
- 处理工作是基于事件的，除非明确停止否则没有“尽头”。处理结果立刻可用，并会随着新数据的抵达继续更新。

流处理系统可以处理几乎无限量的数据，但同一时间只能处理一条（真正的流处理）或很少量（微批处理，Micro-batch Processing）数据，不同记录间只维持最少量的状态。虽然大部分系统提供了用于维持某些状态的方法，但流处理主要针对副作用更少，更加功能性的处理（Functional processing）进行优化。

功能性操作主要侧重于状态或副作用有限的离散步骤。针对同一个数据执行同一个操作会忽略其他因素产生相同的结果，此类处理非常适合流处理，因为不同项的状态通常是某些困难、限制，以及某些情况下不需要的结果的结合体。因此虽然某些类型的状态管理通常是可行的，但这些框架通常在不具备状态管理机制时更简单也更高效。

此类处理非常适合某些类型的工作负载。有近实时处理需求的任务很适合使用流处理模式。分析、服务器或应用程序错误日志，以及其他基于时间的衡量指标是最适合的类型，因为对这些领域的数据变化做出响应对于业务职能来说是极为关键的。流处理很适合用来处理必须对变动或峰值做出响应，并且关注一段时间内变化趋势的数据。

Apache Storm

Apache Storm是一种侧重于极低延迟的流处理框架，也许是要求近实时处理的工作负载的最佳选择。该技术可处理非常大量的数据，通过比其他解决方案更低的延迟提供结果。

流处理模式

Storm 的流处理可对框架中名为 Topology（拓扑）的 DAG（Directed Acyclic Graph，有向无环图）进行编排。这些拓扑描述了当数据片段进入系统后，需要对每个传入的片段执行的不同转换或步骤。

拓扑包含：

- Stream：普通的数据流，这是一种会持续抵达系统的无边界数据。
- Spout：位于拓扑边缘的数据流来源，例如可以是API或查询等，从这里可以产生待处理的数据。
- Bolt：Bolt代表需要消耗流数据，对其应用操作，并将结果以流的形式进行输出的处理步骤。Bolt需要与每个Spout建立连接，随后相互连接以组成所有必要的处理。在拓扑的尾部，可以使用最终的Bolt输出作为相互连接的其他系统的输入。

Storm 背后的想法是使用上述组件定义大量小型的离散操作，随后将多个组件组成所需拓扑。默认情况下 Storm 提供了“至少一次”的处理保证，这意味着可以确保每条消息至少可以被处理一次，但某些情况下如果遇到失败可能会处理多次。Storm 无法确保可以按照特定顺序处理消息。

为了实现严格的一次处理，即有状态处理，可以使用一种名为 Trident 的抽象。严格来说不使用 Trident 的 Storm 通常可称之为 Core Storm。Trident 会对 Storm 的处理能力产生极大影响，会增加延迟，为处理提供状态，使用微批模式代替逐项处理的纯粹流处理模式。

为避免这些问题，通常建议 Storm 用户尽可能使用 Core Storm。然而也要注意，Trident 对内容严格的一次处理保证在某些情况下也比较有用，例如系统无法智能地处理重复消息时。如果需要在项之间维持状态，例如想要计算一个小时内有多少用户点击了某个链接，此时 Trident 将是

你唯一的选择。尽管不能充分发挥框架与生俱来的优势，但 Trident 提高了 Storm 的灵活性。

Trident 拓扑包含：

- 流批（Stream batch）：这是指流数据的微批，可通过分块提供批处理语义。
- 操作（Operation）：是指可以对数据执行的批处理过程。

优势和局限

目前来说 Storm 可能是近实时处理领域的最佳解决方案。该技术可以用极低延迟处理数据，可用于希望获得最低延迟的工作负载。如果处理速度直接影响用户体验，例如需要将处理结果直接提供给访客打开的网站页面，此时 Storm 将会是一个很好的选择。

Storm 与 Trident 配合使得用户可以用微批代替纯粹的流处理。虽然借此用户可以获得更大灵活性打造更符合要求的工具，但同时这种做法会削弱该技术相比其他解决方案最大的优势。话虽如此，但多一种流处理方式总是好的。

Core Storm 无法保证消息的处理顺序。Core Storm 为消息提供了“至少一次”的处理保证，这意味着可以保证每条消息都能被处理，但也可能发生重复。Trident 提供了严格的一次处理保证，可以在不同批之间提供顺序处理，但无法在一个批内部实现顺序处理。

在互操作性方面，Storm 可与 Hadoop 的 YARN 资源管理器进行集成，因此可以很方便地融入现有 Hadoop 部署。除了支持大部分处理框架，Storm 还可支持多种语言，为用户的拓扑定义提供了更多选择。

总结

对于延迟需求很高的纯粹的流处理工作负载，Storm 可能是最适合的

技术。该技术可以保证每条消息都被处理，可配合多种编程语言使用。由于 Storm 无法进行批处理，如果需要这些能力可能还需要使用其他软件。如果对严格的一次处理保证有比较高的要求，此时可考虑使用 Trident。不过这种情况下其他流处理框架也许更适合。

Apache Samza

Apache Samza 是一种与 Apache Kafka 消息系统紧密绑定的流处理框架。虽然 Kafka 可用于很多流处理系统，但按照设计，Samza 可以更好地发挥 Kafka 独特的架构优势和保障。该技术可通过 Kafka 提供容错、缓冲，以及状态存储。

Samza 可使用 YARN 作为资源管理器。这意味着默认情况下需要具备 Hadoop 集群（至少具备 HDFS 和 YARN），但同时也意味着 Samza 可以直接使用 YARN 丰富的内建功能。

流处理模式

Samza 依赖 Kafka 的语义定义流的处理方式。Kafka 在处理数据时涉及下列概念：

- Topic（话题）：进入 Kafka 系统的每个数据流可称之为一个话题。话题基本上是一种可供消耗方订阅的，由相关信息组成的数据流。
- Partition（分区）：为了将一个话题分散至多个节点，Kafka 会将传入的消息划分为多个分区。分区的划分将基于键（Key）进行，这样可以保证包含同一个键的每条消息可以划分至同一个分区。分区的顺序可获得保证。
- Broker（代理）：组成 Kafka 集群的每个节点也叫做代理。
- Producer（生成方）：任何向 Kafka 话题写入数据的组件可以叫做

生成方。生成方可提供将话题划分为分区所需的键。

- Consumer（消耗方）：任何从Kafka读取话题的组件可叫做消耗方。消耗方需要负责维持有关自己分支的信息，这样即可在失败后知道哪些记录已经被处理过了。

由于 Kafka 相当于永恒不变的日志，Samza 也需要处理永恒不变的数据流。这意味着任何转换创建的新数据流都可被其他组件所使用，而不会对最初的数据流产生影响。

优势和局限

乍看之下，Samza 对 Kafka 类查询系统的依赖似乎是一种限制，然而这也可为系统提供一些独特的保证和功能，这些内容也是其他流处理系统不具备的。

例如 Kafka 已经提供了可以通过低延迟方式访问的数据存储副本，此外还可以为每个数据分区提供非常易用且低成本的多订阅者模型。所有输出内容，包括中间态的结果都可写入到 Kafka，并可被下游步骤独立使用。

这种对 Kafka 的紧密依赖在很多方面类似于 MapReduce 引擎对 HDFS 的依赖。虽然在批处理的每个计算之间对 HDFS 的依赖导致了一些严重的性能问题，但也避免了流处理遇到的很多其他问题。

Samza 与 Kafka 之间紧密的关系使得处理步骤本身可以非常松散地耦合在一起。无需事先协调，即可在输出的任何步骤中增加任意数量的订阅者，对于有多个团队需要访问类似数据的组织，这一特性非常有用。多个团队可以全部订阅进入系统的数据话题，或任意订阅其他团队对数据进行过某些处理后创建的话题。这一切并不会对数据库等负载密集型基础架构造成额外的压力。

直接写入 Kafka 还可避免回压（Backpressure）问题。回压是指当负

载峰值导致数据流入速度超过组件实时处理能力的情况，这种情况可能导致处理工作停顿并可能丢失数据。按照设计，Kafka 可以将数据保存很长时间，这意味着组件可以在方便的时候继续进行处理，并可直接重启而无需担心造成任何后果。

Samza 可以使用以本地键值存储方式实现的容错检查点系统存储数据。这样 Samza 即可获得“至少一次”的交付保障，但面对由于数据可能多次交付造成的失败，该技术无法对汇总后状态（例如计数）提供精确恢复。

Samza 提供的高级抽象使其在很多方面比 Storm 等系统提供的基元（Primitive）更易于配合使用。目前 Samza 只支持 JVM 语言，这意味着它在语言支持方面不如 Storm 灵活。

总结

对于已经具备或易于实现 Hadoop 和 Kafka 的环境，Apache Samza 是流处理工作负载一个很好的选择。Samza 本身很适合有多个团队需要使用（但相互之间并不一定紧密协调）不同处理阶段的多个数据流的组织。Samza 可大幅简化很多流处理工作，可实现低延迟的性能。如果部署需求与当前系统不兼容，也许并不适合使用，但如果需要极低延迟的处理，或对严格的一次处理语义有较高需求，此时依然适合考虑。

混合处理系统：批处理和流处理

一些处理框架可同时处理批处理和流处理工作负载。这些框架可以用相同或相关的组件和 API 处理两种类型的数据，借此让不同的处理需求得以简化。

如你所见，这一特性主要是由 Spark 和 Flink 实现的，下文将介绍这两种框架。实现这样的功能重点在于两种不同处理模式如何进行统一，以

及要对固定和不固定数据集之间的关系进行何种假设。

虽然侧重于某一种处理类型的项目会更好地满足具体用例的要求，但混合框架意在提供一种数据处理的通用解决方案。这种框架不仅可以提供处理数据所需的方法，而且提供了自己的集成项、库、工具，可胜任图形分析、机器学习、交互式查询等多种任务。

Apache Spark

Apache Spark 是一种包含流处理能力的下一代批处理框架。与 Hadoop 的 MapReduce 引擎基于各种相同原则开发而来的 Spark 主要侧重于通过完善的内存计算和处理优化机制加快批处理工作负载的运行速度。

Spark 可作为独立集群部署（需要相应存储层的配合），或可与 Hadoop 集成并取代 MapReduce 引擎。

批处理模式

与 MapReduce 不同，Spark 的数据处理工作全部在内存中进行，只在一开始将数据读入内存，以及将最终结果持久存储时需要与存储层交互。所有中间态的处理结果均存储在内存中。

虽然内存中处理方式可大幅改善性能，Spark 在处理与磁盘有关的任务时速度也有很大提升，因为通过提前对整个任务集进行分析可以实现更完善的整体式优化。为此 Spark 可创建代表所需执行的全部操作，需要操作的数据，以及操作和数据之间关系的 Directed Acyclic Graph（有向无环图），即 DAG，借此处理器可以对任务进行更智能的协调。

为了实现内存中批计算，Spark 会使用一种名为 Resilient Distributed Dataset（弹性分布式数据集），即 RDD 的模型来处理数据。这是一种代表数据集，只位于内存中，永恒不变的结构。针对 RDD 执行的操作可生成新的 RDD。每个 RDD 可通过世系（Lineage）回溯至父级 RDD，

并最终回溯至磁盘上的数据。Spark 可通过 RDD 在无需将每个操作的结果写回磁盘的前提下实现容错。

流处理模式

流处理能力是由 Spark Streaming 实现的。Spark 本身在设计上主要面向批处理工作负载，为了弥补引擎设计和流处理工作负载特征方面的差异，Spark 实现了一种叫做微批（Micro-batch）* 的概念。在具体策略方面该技术可以将数据流视作一系列非常小的“批”，借此即可通过批处理引擎的原生语义进行处理。

Spark Streaming 会以亚秒级增量对流进行缓冲，随后这些缓冲会作为小规模的固定数据集进行批处理。这种方式的实际效果非常好，但相比真正的流处理框架在性能方面依然存在不足。

优势和局限

使用 Spark 而非 Hadoop MapReduce 的主要原因是速度。在内存计算策略和先进的 DAG 调度等机制的帮助下，Spark 可以用更快速度处理相同的数据集。

Spark 的另一个重要优势在于多样性。该产品可作为独立集群部署，或与现有 Hadoop 集群集成。该产品可运行批处理和流处理，运行一个集群即可处理不同类型的任务。

除了引擎自身的能力外，围绕 Spark 还建立了包含各种库的生态系统，可为机器学习、交互式查询等任务提供更好的支持。相比 MapReduce，Spark 任务更是“众所周知”地易于编写，因此可大幅提高生产力。

为流处理系统采用批处理的方法，需要对进入系统的数据进行缓冲。缓冲机制使得该技术可以处理非常大量的传入数据，提高整体吞吐率，但等待缓冲区清空也会导致延迟增高。这意味着 Spark Streaming 可能不适

合处理对延迟有较高要求的工作负载。

由于内存通常比磁盘空间更贵，因此相比基于磁盘的系统，Spark 成本更高。然而处理速度的提升意味着可以更快速完成任务，在需要按照小时数为资源付费的环境中，这一特性通常可以抵消增加的成本。

Spark 内存计算这一设计的另一个后果是，如果部署在共享的集群中可能会遇到资源不足的问题。相比 Hadoop MapReduce，Spark 的资源消耗更大，可能会对需要在同一时间使用集群的其他任务产生影响。从本质来看，Spark 更不适合与 Hadoop 堆栈的其他组件共存一处。

总结

Spark 是多样化工作负载处理任务的最佳选择。Spark 批处理能力以更高内存占用为代价提供了无与伦比的速度优势。对于重视吞吐率而非延迟的工作负载，则比较适合使用 Spark Streaming 作为流处理解决方案。

Apache Flink

Apache Flink 是一种可以处理批处理任务的流处理框架。该技术可将批处理数据视作具备有限边界的 data stream，借此将批处理任务作为流处理的子集加以处理。为所有处理任务采取流处理为先的方法会产生一系列有趣的副作用。

这种流处理为先的方法也叫做 Kappa 架构，与之相对的是更加被广为人知的 Lambda 架构（该架构中使用批处理作为主要处理方法，使用流作为补充并提供早期未经提炼的结果）。Kappa 架构中会对一切进行流处理，借此对模型进行简化，而这一切是在最近流处理引擎逐渐成熟后才可行的。

流处理模型

Flink 的流处理模型在处理传入数据时会将每一项视作真正的数据流。Flink 提供的 DataStream API 可用于处理无尽的数据流。Flink 可配

合使用的基本组件包括：

- Stream（流）是指在系统中流转的，永恒不变的无边界数据集
- Operator（操作方）是指针对数据流执行操作以产生其他数据流的功能
- Source（源）是指数据流进入系统的入口点
- Sink（槽）是指数据流离开Flink系统后进入到的位置，槽可以是数据库或到其他系统的连接器

为了在计算过程中遇到问题后能够恢复，流处理任务会在预定时间点创建快照。为了实现状态存储，Flink 可配合多种状态后端系统使用，具体取决于所需实现的复杂度和持久性级别。

此外 Flink 的流处理能力还可以理解“事件时间”这一概念，这是指事件实际发生的时间，此外该功能还可以处理会话。这意味着可以通过某种有趣的方式确保执行顺序和分组。

批处理模型

Flink 的批处理模型在很大程度上仅是对流处理模型的扩展。此时模型不再从持续流中读取数据，而是从持久存储中以流的形式读取有边界的数据集。Flink 会对这些处理模型使用完全相同的运行时。

Flink 可以对批处理工作负载实现一定的优化。例如由于批处理操作可通过持久存储加以支持，Flink 可以不对批处理工作负载创建快照。数据依然可以恢复，但常规处理操作可以执行得更快。

另一个优化是对批处理任务进行分解，这样即可在需要的时候调用不同阶段和组件。借此 Flink 可以与集群的其他用户更好地共存。对任务提前进行分析使得 Flink 可以查看需要执行的所有操作、数据集的大小，以及下游需要执行的操作步骤，借此实现进一步的优化。

优势和局限

Flink 目前是处理框架领域一个独特的技术。虽然 Spark 也可以执行批处理和流处理，但 Spark 的流处理采取的微批架构使其无法适用于很多用例。Flink 流处理为先的方法可提供低延迟，高吞吐率，近乎逐项处理的能力。

Flink 的很多组件是自行管理的。虽然这种做法较为罕见，但出于性能方面的原因，该技术可自行管理内存，无需依赖原生的 Java 垃圾回收机制。与 Spark 不同，待处理数据的特征发生变化后 Flink 无需手工优化和调整，并且该技术也可以自行处理数据分区和自动缓存等操作。

Flink 会通过多种方式对工作进行分许进而优化任务。这种分析在部分程度上类似于 SQL 查询规划器对关系型数据库所做的优化，可针对特定任务确定最高效的实现方法。该技术还支持多阶段并行执行，同时可将受阻任务的数据集合在一起。对于迭代式任务，出于性能方面的考虑，Flink 会尝试在存储数据的节点上执行相应的计算任务。此外还可进行“增量迭代”，或仅对数据中有改动的部分进行迭代。

在用户工具方面，Flink 提供了基于 Web 的调度视图，借此可轻松管理任务并查看系统状态。用户也可以查看已提交任务的优化方案，借此了解任务最终是如何在集群中实现的。对于分析类任务，Flink 提供了类似 SQL 的查询，图形化处理，以及机器学习库，此外还支持内存计算。

Flink 能很好地与其他组件配合使用。如果配合 Hadoop 堆栈使用，该技术可以很好地融入整个环境，在任何时候都只占用必要的资源。该技术可轻松地与 YARN、HDFS 和 Kafka 集成。在兼容包的帮助下，Flink 还可以运行为其他处理框架，例如 Hadoop 和 Storm 编写的任务。

目前 Flink 最大的局限之一在于这依然是一个非常“年幼”的项目。

现实环境中该项目的大规模部署尚不如其他处理框架那么常见，对于 Flink 在缩放能力方面的局限目前也没有较为深入的研究。随着快速开发周期的推进和兼容包等功能的完善，当越来越多的组织开始尝试时，可能会出现越来越多的 Flink 部署。

总结

Flink 提供了低延迟流处理，同时可支持传统的批处理任务。Flink 也许最适合有极高流处理需求，并有少量批处理任务的组织。该技术可兼容原生 Storm 和 Hadoop 程序，可在 YARN 管理的集群上运行，因此可以很方便地进行评估。快速进展的开发工作使其值得被大家关注。

结论

大数据系统可使用多种处理技术。

对于仅需要批处理的工作负载，如果对时间不敏感，比其他解决方案实现成本更低的 Hadoop 将会是一个好选择。

对于仅需要流处理的工作负载，Storm 可支持更广泛的语言并实现极低延迟的处理，但默认配置可能产生重复结果并且无法保证顺序。Samza 与 YARN 和 Kafka 紧密集成可提供更大灵活性，更易用的多团队使用，以及更简单的复制和状态管理。

对于混合型工作负载，Spark 可提供高速批处理和微批处理模式的流处理。该技术的支持更完善，具备各种集成库和工具，可实现灵活的集成。Flink 提供了真正的流处理并具备批处理能力，通过深度优化可运行针对其他平台编写的任务，提供低延迟的处理，但实际应用方面还为时过早。

最适合的解决方案主要取决于待处理数据的状态，对处理所需时间的需求，以及希望得到的结果。具体是使用全功能解决方案或主要侧重于某种项目的解决方案，这个问题需要慎重权衡。随着逐渐成熟并被广泛接受，在评估任何新出现的创新型解决方案时都需要考虑类似的问题。

亿级用户 PC 主站的 PHP7 升级实践

作者 侯青龙

背景

新浪微博在 2016 年 Q2 季度公布月活跃用户 (MAU) 较上年同期增长 33%，至 2.82 亿；日活跃用户 (DAU) 较上年同期增长 36%，至 1.26 亿，总注册用户达 8 亿多。PC 主站作为重要的流量入口，承载部分用户访问和流量落地，其中我们提供的部分服务(如：头条文章)承担全网所有流量。

随着业务的增长，系统压力也在不断的增加。峰值时，服务器 Hits 达 10W+，CPU 使用率也达到了 80%，远超报警阈值。另外，当前机房的机架已趋于饱和，遇到突发事件，只能对非核心业务进行降低，挪用这些业务的服务器来进行临时扩容，这种方案只能算是一种临时方案，不能满足长久的业务增长需求。再加上一年一度的三节（圣诞、元旦、春节），系统需预留一定的冗余来应对，所以当前系统面临的问题非常严峻，解决系统压力的问题也迫在眉急。

面对当前的问题，我们内部也给出两套解决方案同步进行。

- 方案一：申请新机房，资源统一配置，实现弹性扩容。
- 方案二：对系统进行优化，对性能做进一步提升。

针对方案一，通过搭建与新机房之间的专线与之打通，高峰时，运用内部自研的混合云 DCP 平台，对所有资源进行调度管理，实现了真正意义上的弹性扩容。目前该方案已经在部分业务灰度运行，随时能对重点业务进行小流量测试。

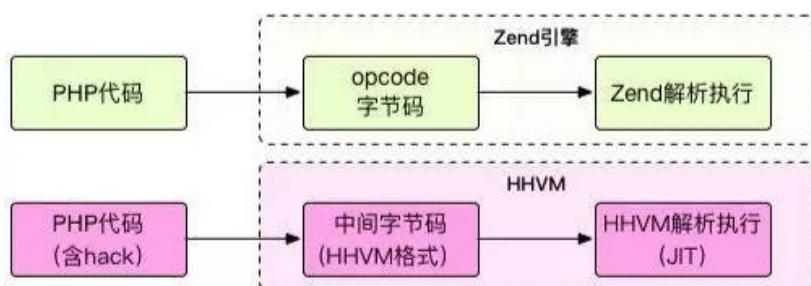
针对方案二，系统层面，之前做过多次大范围的优化，比如：

- 将Apache升级至Nginx
- 应用框架升级至Yaf
- CPU计算密集型的逻辑扩展化
- 弃用smarty
- 并行化调用

优化效果非常明显，如果再从系统层面进行优化，性能可提升的空间非常有限。好在业界传出了两大福音，分别为 HHVM 和 PHP7。

方案选型

在 PHP7 还未正式发布时，我们也研究过 HHVM（HipHop Virtual Machine），关于 HHVM 更多细节，这里就不再赘述，可参考官方说明。下面对它提升性能的方式进行一个简单的介绍。

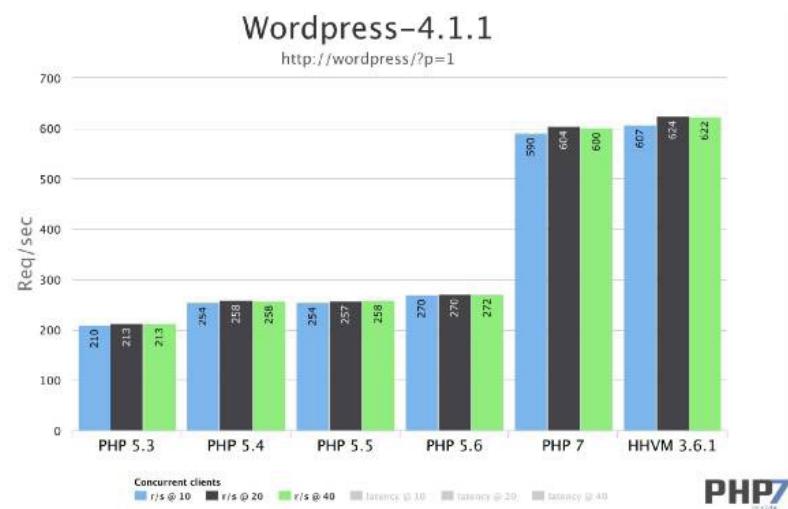


默认情况下，Zend 引擎先将 PHP 源码编译为 opcode，然后 Zend 解析引擎逐条执行。这里的 opcode 码，可以理解成 C 语言级的函数。而 HHVM 提升性能方式为替代 Zend 引擎将 PHP 代码转换成中间字节码（HHVM 自己的中间字节码，通常称为中间语言），然后在运行时通过即时（JIT）编译器将这些字节码转换成 x64 的机器码，类似于 Java 的 JVM。

HHVM 为了达到最佳优化效果，需要将 PHP 的变量类型固定下来，而不是让编译器去猜测。Facebook 的工程师们就定义一种 Hack 写法，进而来达到编译器优化的目的，写法类似如下：

```
<?hh
class point {
    public float $x, $y;
    function __construct(float $x, float $y) {
        $this->x = $x;
        $this->y = $y;
    }
}
```

通过前期的调研，如果使用 HHVM 解析器来优化现有业务代码，为了达到最佳的性能提升，必须对代码进行大量修改。另外，服务部署也比较复杂，有一定的维护成本，综合评估后，该方案我们也就不再考虑。



当然，PHP7 的开发进展我们也一直在关注，通过官方测试数据以及内部自己测试，性能提升非常明显。

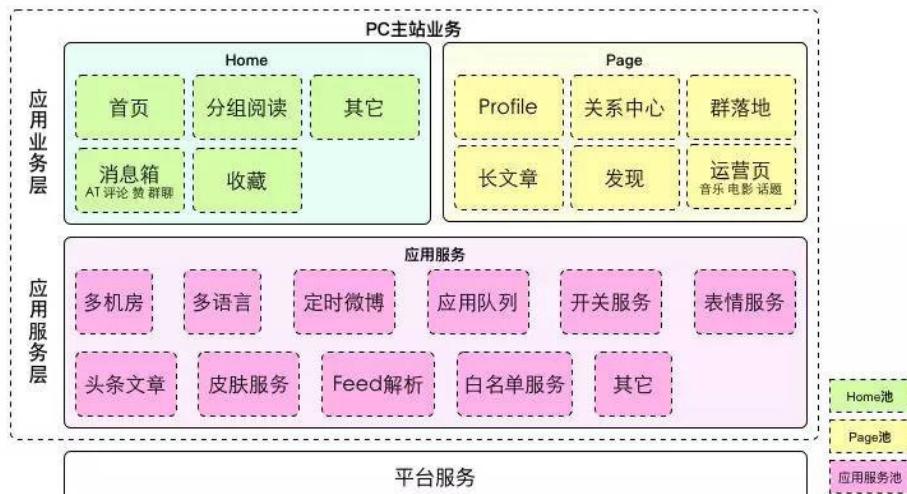
令人兴奋的是，在去年年底（2015 年 12 月 04 日），官方终于正式发布了 PHP7，并且对原生的代码几乎可以做到完全兼容，性能方面与 PHP5 比较能提升达一倍左右，和 HHVM 相比已经是不相上下。

	性能	开发成本	运维成本	问题响应
PHP7	高	低，99%代码 身后兼容	运维成本	快
HHVM	高	低	高	慢

无论从优化成本、风险控制，还是从性能提升上来看，选择 PHP7 无疑是我们的最佳方案。

系统现状以及升级风险

微博 PC 主站从 2009 年 8 月 13 日发布第一版开始，先后经历了 6 个大的版本，系统架构也随着需求的变化进行过多次重大调整。截止目前，系统部分架构如下。



从系统结构层面来看，系统分应用业务层、应用服务层，系统所依赖基础数据由平台服务层提供。

从服务部署层面来看，业务主要部署在三大服务集群，分别为 Home 池、Page 池以及应用服务池。

为了提升系统性能，我们自研了一些 PHP 扩展，由于 PHP5 和 PHP7 底层差别太大，大部分 Zend API 接口都进行了调整，所有扩展都需要修改。

所以，将 PHP5 环境升级至 PHP7 过程中，主要面临如下风险：

- 使用了自研的PHP扩展，目前这些扩展只有PHP5版本，将这些扩展升级至PHP7，风险较大。
- PHP5与PHP7语法在某种程度上，多少还是存在一些兼容性的问题。由于涉及主站代码量庞大，业务逻辑分支复杂，很多测试范围仅仅通过人工测试是很难触达的，也将面临很多未知的风险。
- 软件新版本的发布，都会面临着一些未知的风险和版本缺陷。这些问题，是否能快速得到解决。
- 涉及服务池和项目较多，基础组件的升级对业务范围影响较大，升级期间出现的问题、定位会比较复杂。

对微博这种数亿用户级别的系统的基础组件进行升级，影响范围将非常之大，一旦某个环节考虑不周全，很有可能会出现比较严重的责任事故。

PHP7 升级实践

1. 扩展升级

一些常用的扩展，在发布 PHP7 时，社区已经做了相应升级，如：Memcached、PHPRedis 等。另外，微博使用的 Yaf、Yar 系列扩展，由于鸟哥 (laruence) 的支持，很早就全面支持了 PHP7。对于这部分扩展，需要详细的测试以及现网灰度来进行保障。

PHP7 中，很多常用的 API 接口都做了改变，例如 HashTable API 等。

对于自研的 PHP 扩展，需要做升级，比如我们有个核心扩展，升级涉及到代码量达 1500 行左右。

新升级的扩展，刚开始也面临着各式各样的问题，我们主要通过官方给出的建议以及测试流程来保证其稳定可靠。

官方建议

在 PHP7 下编译你的扩展，编译错误与警告会告诉你绝大部分需要修改的地方。

在 DEBUG 模式下编译与调试你的扩展，在 run-time 你可以通过断言捕捉一些错误。你还可以看到内存泄露的情况。

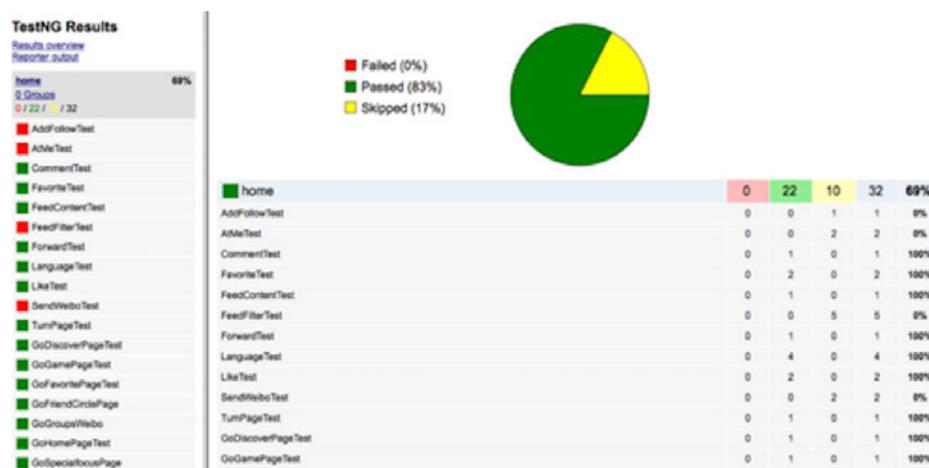
测试流程

首先通过扩展所提供的单元测试来保证扩展功能的正确性。

```
PASS Check for weibo presence [tests/001.php]
PASS Check for Weibo_Util::MidEncode & Weibo_Util::MidDecode [tests/002.php]
PASS Check for Weibo_Conf [tests/003.php]
PASS Check for Weibo_Conf [tests/004.php]
PASS Check for type converting bug for MidEncode/Decode [tests/033.php]
PASS Check for Weibo_Formatter::renderAt() [tests/weibo_formatter/at_001.php]
PASS Check for Weibo_Formatter::renderEmoji() [tests/weibo_formatter/emoji_001.php]
PASS Check for Weibo_Formatter::renderIcon() [tests/weibo_formatter/icon_001.php]
PASS Check for Weibo_Formatter::renderTag() [tests/weibo_formatter/tag_001.php]
PASS Check for Weibo_Formatter::renderTopic() [tests/weibo_formatter/topic_003.php]
```

其次通过大量的压力测试来验证其稳定性。

然后再通过业务代码的自动化测试来保证业务功能的可用性。



最后再通过现网流量灰度来确保最终的稳定可靠。

整体升级过程中，涉及到的修改比较多，以下只简单列举出一些参数变更的函数。

(1) addassocstringl 参数 4 个改为了 3 个。

```
//PHP5  
add_assoc_stringl(parray, key, value, value_len);  
//PHP7  
add_assoc_stringl(parray, key, value);
```

(2) addnextindex_stringl 参数从 3 个改为了 2 个。

```
//PHP5  
add_next_index_stringl(parray, value, value_len);  
//PHP7 add_next_index_string(parray, value); //PHP7
```

(3) RETURN_STRINGL 参数从 3 个改为了 2 个。

```
//PHP5  
RETURN_STRINGL(value, length, dup);  
//PHP7  
RETURN_STRINGL(value, length);
```

(4) 变量声明从堆上分配，改为栈上分配。

```
//PHP5  
zval* sarray_1;  
ALLOC_INIT_ZVAL(sarray_1);  
array_init(sarray_1);
```

```
//PHP7  
zval sarray_1;  
array_init(&sarray_1);
```

(5) zendhashgetcurrentkey_ex 参数从 6 个改为 4 个。

```
//PHP5  
ZEND_API int ZEND_FASTCALL zend_hash_get_current_key_ex (   
HashTable* ht,
```

```
char** str_index,
uint* str_length,
ulong* num_index,
zend_bool duplicate,
HashPosition* pos);

//PHP7
ZEND_API int ZEND_FASTCALL zend_hash_get_current_key_ex(
    const HashTable *ht,
    zend_string **str_index,
    zend_ulong *num_index,
    HashPosition *pos);
```

更详细的说明，可参考官方 PHP7 扩展迁移文档：<https://wiki.php.net/PHPng-upgrading>。

2. PHP代码升级

整体来讲，PHP7 向前的兼容性正如官方所描述那样，能做到 99% 向前兼容，不需要做太多修改，但在整体迁移过程中，还是需要做一些兼容处理。

另外，在灰度期间，代码将同时运行于 PHP5.4 和 PHP7 环境，现网灰度前，我们首先对所有代码进行了兼容性修改，以便同一套代码能同时兼容两套环境，然后再按计划对相关服务进行现网灰度。

同时，对于 PHP7 的新特性，升级期间，也强调不允许被使用，否则代码与低版本环境的兼容性会存在问题。

接下来简单介绍下升级 PHP7 代码过程中，需要注意的地方。

(1) 很多致命错误以及可恢复的致命错误，都被转换为异常来处理，这些异常继承自 Error 类，此类实现了 Throwable 接口。对未定义的函数进行调用，PHP5 和 PHP7 环境下，都会出现致命错误。

```
undefined_function();
```

错误提示：

```
PHP Fatal error: Call to undefined function undefined_function()
in /tmp/test.PHP on line 4
```

在 PHP7 环境下，这些致命的错误被转换为异常来处理，可以通过异常来进行捕获。

```
try {
    undefined_function();
}
catch (Throwable $e) {
    echo $e;
}
```

提示：

```
Error: Call to undefined function undefined_function() in /tmp/
test.PHP:5 Stack trace:
```

```
#0 {main}
```

(2) 被 0 除，PHP 7 之前，被 0 除会导致一条 E_WARNING 并返回 false。一个数字运算返回一个布尔值是没有意义的，PHP 7 会返回如下的 float 值之一。

```
+INF
```

```
-INF
```

```
NAN
```

如下：

```
var_dump(42/0); // float(INF) + E_WARNING
var_dump(-42/0); // float(-INF) + E_WARNING
var_dump(0/0); // float(NAN) + E_WARNING
```

当使用取模运算符（%）的时候，PHP7 会抛出一个 DivisionByZeroError 异常，PHP7 之前，则抛出的是警告。

```
echo 42 % 0;
```

PHP5 输出：

```
PHP Warning: Division by zero in /tmp/test.PHP on line 4
```

PHP7 输出：

```
PHP Fatal error: Uncaught DivisionByZeroError: Modulo by zero  
in /tmp/test.PHP:4 Stack trace:  
#0 {main}  
thrown in /tmp/test.PHP on line 4
```

PHP7 环境下，可以捕获该异常：

```
try {  
    echo 42 % 0;  
} catch (DivisionByZeroError $e) {  
    echo $e->getMessage();  
}
```

输出：

```
Modulo by zero
```

(3) pregrepate() 函数不再支持 "\e" (PREGREPLACEEVAL). 使用 pregreplace_callback() 替代。

PHP5：

```
$content = preg_replace("/#([^\#]+)\#/ies", "strip_  
tags('#\\1#')", $content);
```

PHP7：

```
$content = preg_replace_callback("/#([^\#]+)\#/is",  
"self::strip_str_tags", $content);  
public static function strip_str_tags($matches){  
    return "#".strip_tags($matches[1])."#";  
}
```

(4) 以静态方式调用非静态方法。

```
class foo {  
    function bar() {  
        echo 'I am not static!';  
    }
```

```

    }
}

foo::bar();

```

以上代码 PHP7 会输出：

```

PHP Deprecated: Non-static method foo::bar() should not be
called statically in /tmp/test.PHP on line 10
I am not static!

```

(5) E_STRICT 警告级别变更。

原有的 E_STRICT 警告都被迁移到其他级别。E_STRICT 常量会被保留，所以调用 errorreporting(EALL|E_STRICT) 不会引发错误。

场景	新的级别/行为
将资源类型的变量用作键来进行索引	E_NOTICE
抽象静态方法	不再警告, 会引发错误
重复定义构造器函数	不再警告, 会引发错误
在继承的时候, 方法签名不匹配	E_WARNING
在两个 trait 中包含相同的(兼容的)属性	不再警告, 会引发错误
以非静态调用的方式访问静态属性	E_NOTICE
变量应该以引用的方式赋值	E_NOTICE
变量应该以引用的方式传递(到函数参数中)	E_NOTICE
以静态方式调用实例方法	E_DEPRECATED

关于代码兼容 PHP7, 基本上是对代码的规范要求更严谨。以前写的不规范的地方, 解析引擎只是输出 NOTICE 或者 WARNING 进行提示, 不影响对代码上下文的执行, 而到了 PHP7, 很有可能会直接抛出异常, 中断上下文的执行。

如：对 0 取模运行时, PHP7 之前, 解析引擎只抛出警告进行提示, 但到了 PHP7 则会抛出一个 DivisionByZeroError 异常, 会中断整个流程的执行。

对于警告级别的变更, 在升级灰度期间, 一定要关注相关 NOTICE 或 WARNING 报错。PHP7 之前的一个 NOTICE 或者 WARNING 到了 PHP7, 一些报警级变成致命错误或者抛出异常, 一旦没有对相关代码进行优化处理, 逻

辑被触发，业务系统很容易因为抛出的异常没处理而导致系统挂掉。

以上只列举了 PHP7 部分新特性，也是我们在迁移代码时重点关注的一些点，更多细节可参考官方文档 <http://PHP.net/manual/zh/migration70.PHP>。

3. 研发流程变更

一个需求的开发到上线，首先我们会通过统一的开发环境来完成功能开发，其次经过内网测试、仿真测试，这两个环境测试通过后基本保证了数据逻辑与功能方面没有问题。然后合并至主干分支，并将代码部署至预发环境，再经过一轮简单回归，确保合并代码没有问题。最后将代码发布至生产环境。

为了确保新编写的代码能在两套环境（未灰度的 PHP5.4 环境以及灰度中的 PHP7 环境）中正常运行，代码在上线前，也需要在两套环境中分别进行测试，以达到完全兼容。

所以，在灰度期间，对每个环节的运行环境除了现有的 PHP5.4 环境外，我们还分别提供了一套 PHP7 环境，每个阶段的测试中，两套环境都需要进行验证。

4. 灰度方案

之前有过简单的介绍，系统部署在三大服务池，分别为 Home 池、Page 池以及应用服务池。

在准备好安装包后，先是在每个服务池分别部署了一台前端机来灰度。运行一段时间后，期间通过错误日志发现了不少问题，也有用户投诉过来的问题，在问题都基本解决的情况下，逐渐将各服务池的机器池增加至多台。

经过前期的灰度测试，主要的问题得到基本解决。接下是对应用服务

池进行灰度，陆续又发现了不少问题。前后大概经历了一个月左右，完成了应用服务池的升级。然后再分别对 Home 池以及 Page 池进行灰度，经过漫长灰度，最终完成了 PC 主站全网 PHP7 的升级。

虽然很多问题基本上在测试或者灰度期间得到了解决，但依然有些问题是全量上线后一段时间才暴露出来，业务流程太多，很多逻辑需要一定条件才能被触发。为此 BUG 都要第一时间同步给 PHP7 升级项目组，对于升级 PHP 引起的问题，要求必须第一时间解决。

5. 优化方案

(1) 启用 Zend Opcache，启用 Opcache 非常简单，在 PHP.ini 配置文件中加入：

```
zend_extension=opcache.so
opcache.enable=1
opcache.enable_cli=1"
```

(2) 使用 GCC4.8 以上的编译器来编译安装包，只有 GCC4.8 以上编译出的 PHP 才会开启 Global Register for opline and execute_data 支持。

(3) 开启 HugePage 支持，首先在系统中开启 HugePages，然后开启 Opcache 的 hugecodepages。

关于 HugePage

操作系统默认的内存是以 4KB 分页的，而虚拟地址和内存地址需要转换，而这个转换要查表，CPU 为了加速这个查表过程会内建 TLB(Translation Lookaside Buffer)。显然，如果虚拟页越小，表里的条目数也就越多，而 TLB 大小是有限的，条目数越多 TLB 的 Cache Miss 也就会越高，所以如果我们能启用大内存页就能间接降低这个 TLB Cache Miss。

PHP7 与 HugePage

PHP7 开启 HugePage 支持后，会把自身的 text 段，以及内存分配中的 huge 都采用大内存页来保存，减少 TLB miss，从而提高性能。相关实现可参考 Opcache 实现中的 accel_move_code_to_huge_pages() 函数。

```
static void accel_move_code_to_huge_pages(void)
{
    FILE *f;
    long unsigned int huge_page_size = 2 * 1024 * 1024;

    f = fopen("/proc/self/maps", "r");
    if (!f) {
        long unsigned int start, end, offset, inode;
        char perm[5], dev[6], name[MAXPATHLEN];
        int ret;

        ret = fscanf(f, "%lx-%lx %4s %lx %ld %s\n", &start, &end, perm, &offset, dev, &inode, name);
        if (ret == 7 && perm[0] == 'r' && perm[1] == '-' && perm[2] == 'x' && name[0] == '/') {
            long unsigned int seg_start = ZEND_MM_ALIGNED_SIZE_EX(start, huge_page_size);
            long unsigned int seg_end = (end & ~(huge_page_size-1));
            if (seg_end > seg_start) {
                zend_accel_error(ACCEL_LOG_DEBUG, "remap to huge page %lx-%lx %s\n", seg_start, seg_end, name);
                accel_remap_huge_pages((void*)seg_start, seg.end - seg.start, name, offset + seg.start - start);
            }
        }
        fclose(f);
    }
}
```

开启方法

以 CentOS 6.5 为例，通过命令：

```
sudo sysctl vm.nr_hugepages=128
```

分配 128 个预留的大页内存。

```
$ cat /proc/meminfo | grep Huge
AnonHugePages:      444416 kB
HugePages_Total:     128
HugePages_Free:      128
HugePages_Rsvd:       0
HugePages_Surp:       0
Hugepagesize:        2048 kB
```

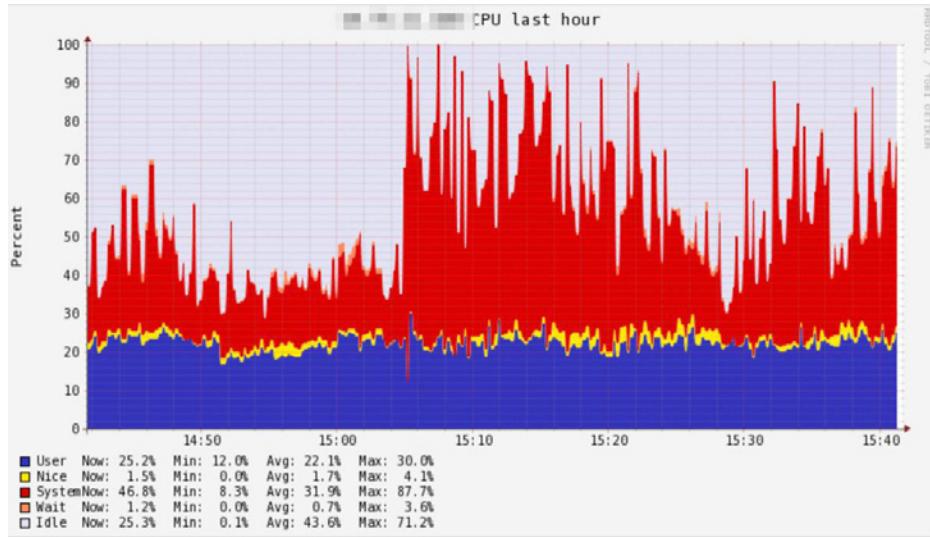
然后在 PHP.ini 中加入

```
opcache.huge_code_pages=1
```

6. 关于负载过高，系统CPU使用占比过高的问题

当我们升级完第一个服务池时，感觉整个升级过程还是比较顺利，当

灰度 Page 池，低峰时一切正常，但到了流量高峰，系统 CPU 占用非常高，如图：



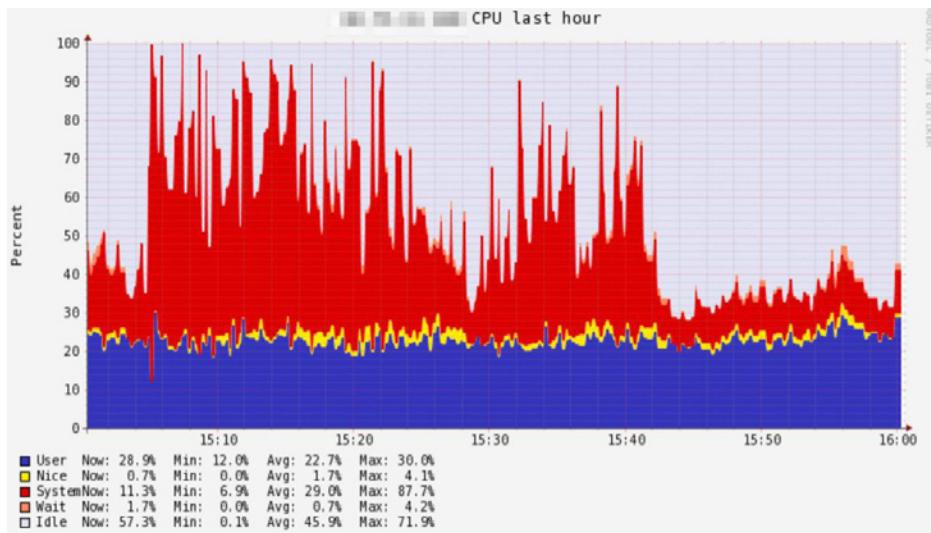
系统 CPU 的使用远超用户程序 CPU 的使用，正常情况下，系统 CPU 与用户程序 CPU 占比应该在 1/3 左右。但我们的实际情况则是，系统 CPU 是用户 CPU 的 2~3 倍，很不正常。

对比了一下两个服务池的流量，发现 Page 池的流量正常比 Home 池高不少，在升级 Home 池时，没发现该问题，主要原因是流量没有达到一定级别，所以未触发该问题。当单机流量超过一定阈值，系统 CPU 的使用会出现一个直线的上升，此时系统性能会严重下降。

这个问题其实困扰了我们有一段时间，通过各种搜索资料，均未发现任何升级 PHP7 会引起系统 CPU 过高的线索。但我们发现了另外一个比较重要的线索，很多软件官方文档里非常明确的提出了可以通过关闭 Transparent HugePages（透明大页）来解决系统负载过高的问题。后来我们也尝试对其进行了关闭，经过几天的观察，该问题得到解决，如图：

什么是 Transparent HugePages（透明大页）

简单的讲，对于内存占用较大的程序，可以通过开启 HugePage 来提



升系统性能。但这里会有个要求，就是在编写程序时，代码里需要显示的对 HugePage 进行支持。

而红帽企业版 Linux 为了减少程序开发的复杂性，并对 HugePage 进行支持，部署了 Transparent HugePages。Transparent HugePages 是一个使管理 Huge Pages 自动化的抽象层，实现方案为操作系统后台有一个叫做 khugepaged 的进程，它会一直扫描所有进程占用的内存，在可能的情况下会把 4kPage 交换为 Huge Pages。

为什么 Transparent HugePages（透明大页）对系统的性能会产生影响

在 khugepaged 进行扫描进程占用内存，并将 4kPage 交换为 Huge Pages 的这个过程中，对于操作的内存的各种分配活动都需要各种内存锁，直接影响程序的内存访问性能。并且，这个过程对于应用是透明的，在应用层面不可控制，对于专门为 4k page 优化的程序来说，可能会造成随机的性能下降现象。

怎么关闭 Transparent HugePages（透明大页）

(1) 查看是否启用透明大页。

```
[root@venus153 ~]# cat /sys/kernel/mm/transparent_hugepage/
```

```
enabled
```

```
[always] madvise never
```

使用命令查看时，如果输出结果为 [always] 表示透明大页启用了，
[never] 表示透明大页禁用。

(2) 关闭透明大页。

```
echo never > /sys/kernel/mm/transparent_hugepage/enable
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

(3) 启用透明大页。

```
echo always > /sys/kernel/mm/transparent_hugepage/enable
echo always > /sys/kernel/mm/transparent_hugepage/defrag
```

(4) 设置开机关闭。

修改 /etc/rc.local 文件，添加如下行：

```
if test -f /sys/kernel/mm/redhat_transparent_hugepage/enable;
then
    echo never > /sys/kernel/mm/transparent_hugepage/enable
    echo never > /sys/kernel/mm/transparent_hugepage/defrag
fi
```

升级效果

由于主站的业务比较复杂，项目较多，涉及服务池达多个，每个服务池所承担业务与流量也不一样，所以我们在对不同的服务池进行灰度升级，遇到的问题也不尽相同，导致整体升级前后达半年之久。庆幸的是，遇到的问题，最终都被解决掉了。最让人兴奋的是升级效果非常好，基本与官方一致，也为公司节省了不少成本。

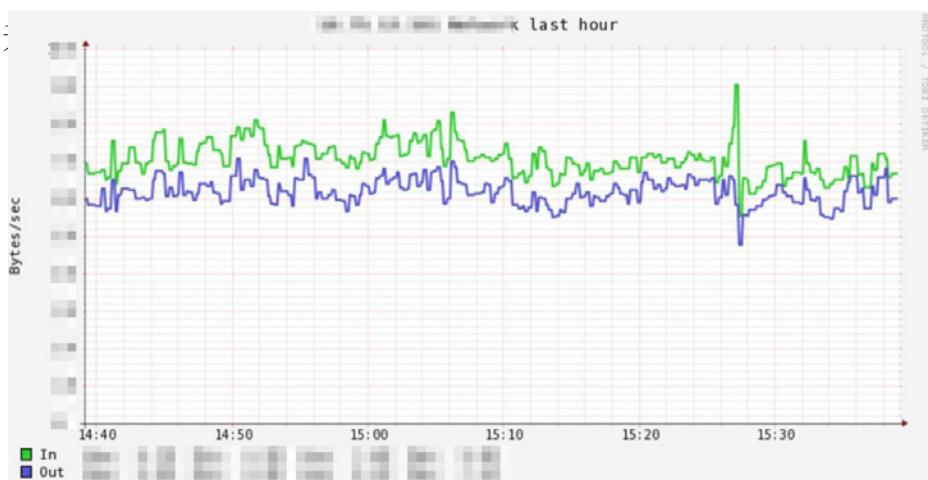
以下简单地给大家展示下这次 PHP7 升级的成果。

(1) PHP5 与 PHP7 环境下，分别对我们的某个核心接口进行压测（压测数据由 QA 团队提供），相关数据如下：

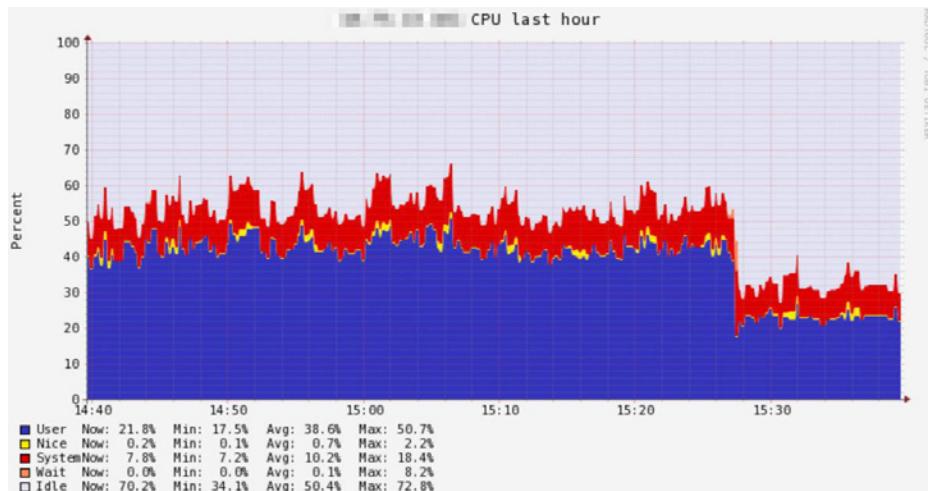
PHP版本	并发数	平均TPS	最大TPS
PHP5	70	95	101
PHP7	185	220	262

同样接口，分别在两个不同的环境中进行测试，平均 TPS 从 95 提升到 220，提升达 130%。

(2) 升级前后，单机 CPU 使用率对比如下。



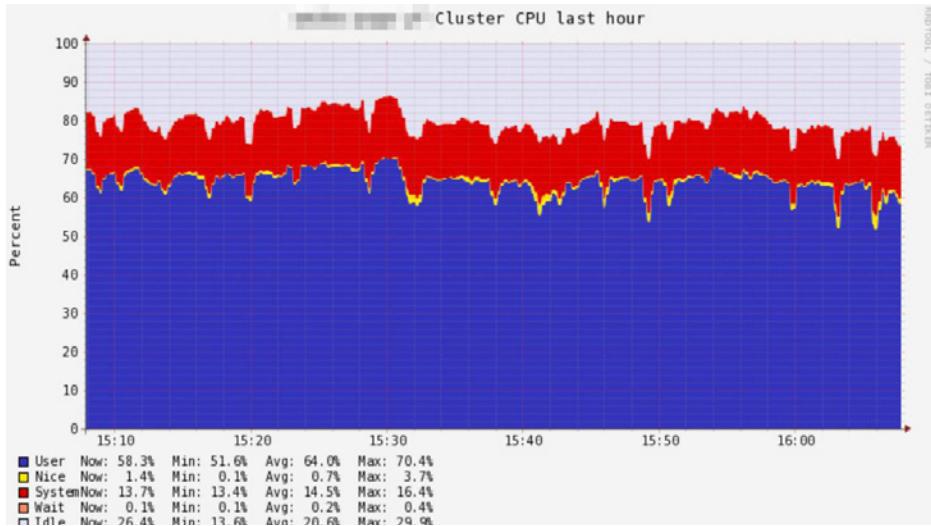
升级前后，1 小时 CPU 使用率变化：



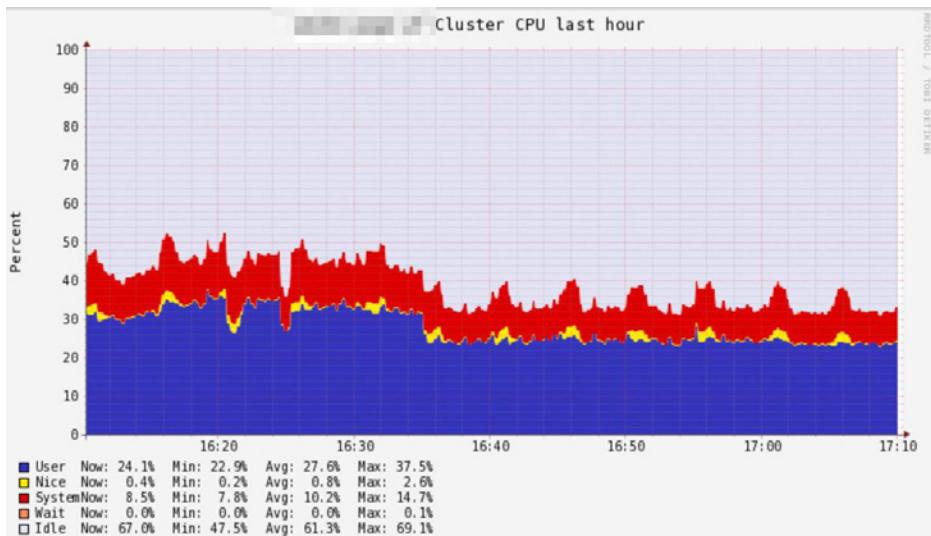
升级前后，在流量变化不大的情况下，CPU 使用率从 45% 降至 25%，CPU 使用率降低 44.44%。

(3) 某服务集群升级前后，同一时间段 1 小时 CPU 使用对比如下。

PHP5 环境下，集群近 1 小时 CPU 使用变化：



PHP7 环境下，集群近 1 小时 CPU 使用变化：



升级前后，CPU 变化对比（图见下页）：

升级前后，同一时段，集群 CPU 平均使用率从 51. 6% 降低至 22. 9%，
使用率降低 56. 88%。

以上只简单从三个维度列举了一些数据。为了让升级效果更加客观，
我们实际的评估维度更多，如内存使用、接口响应时间占比等。最终综合

PHP版本	CPU(min)	CPU(Avg)	CPU(Max)
PHP5	51.6%	64%	70%
PHP7	22.9%	27.6%	37.5
变化	55.62%	56.88%	46.43%

得出的结论为，通过本次升级，PC 主站整体性能提升在 48.82%，效果非常好。团队今年的职能 KPI 就算是提前完成了。

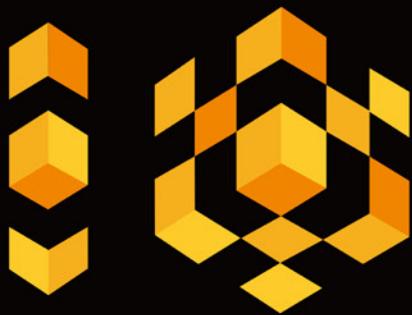
总结

整体升级从准备到最终 PC 主站全网升级完成，时间跨度达半年之久，无论是扩展编写、准备安装脚本、PHP 代码升级还是全网灰度，期间一直会出现各式各样的问题。最终在团队的共同努力下，这些问题都彻底得到了解决。

一直以来，对社区的付出深怀敬畏之心，也是因为他们对 PHP 语言性能极限的追求，才能让我们的业务坐享数倍性能的提升。同时，也让我们更加相信，PHP 一定会是一门越来越好的语言。

作者简介

侯青龙，微博主站研发负责人。2010 年加入新浪微博，先后参与过微博主站 V2 版至 V6 版的研发，主导过主站 V6 版以及多机房消息同步系统等重大项目的架构设计工作。致力于提升产品研发效率以及优化系统性能。



TEN YEARS
OF OPERATION

| 迷你书免费下载 |

十载运营 硕果累累
再接再厉 迈向未来



AWS BOOK

Geekbang InfoQ
极客邦科技

奇谈怪论： 从容器想到去 IOE、去库存和独角兽

作者：梁启鸿

2016 年，容器化技术如火如荼，诞生于 2013 年的 Docker 成了行业的宠儿，它让炒了 8 年的 DevOps 有了更具体可落地可执行的工具。虽然有一定程度的过火现象（所谓的 hype），虽然有很多 IT 人（尤其是在传统垂直行业信息技术部里）依然怀疑容器与虚拟机的差别，但总体来说，容器化可能算的上是软件开发领域的又一次“运动”。

每一次“运动”，都是有很多人追随、有很多技术架构被（一窝蜂的）重新设计、有很多系统被迁移，形成一种潮流（好像不那么干就被时代抛弃），代码的开发调试、架构设计和交付部署的方式发生巨大变化：

- Mainframe：大型机、大集中、傻终端（dumb Terminal）。
- 2 层架构 Client-server 和 4GL：整个 90 年代 - 中小型机/x86 服务器、工作站/PC 终端。Mainframe 应用很多被采用 C/S 架构重写。一

时间都是DCE RPC、DCOM、CORBA-IIOP。

- 3层架构通行：Web 1.0时代开始到现在，浏览器、中间件、关系型数据库服务器的架构依然在很多企业中通行。C/S架构应用很多被迁移至三层架构。Struts+Spring+Hibernate（传说中的SSH）成为这一时代的web应用标配，JEE应用服务器曾经是“State of the art”的技术，甚至成就了上市公司（例如WebLogic）。
- SOA + RIA：SOA最开始是互联网技术（例如HTTP、XML等等）生态回归企业IT，新的技术载体实现企业应用所需要的RPC语义、服务注册与发现机制，再结合Flash、AJAX技术实现的富客户端（Rich Client），实现企业类应用所需要的、比一般网站类应用交互复杂的交互。
- 在CAP定律制约下的分布式架构：Web 2.0时代产物，participation age 的社交网络、UGC，海量数据海量流量促成。
- 容器化的分布式架构：为什么这算的上一次里程碑式的运动？因为它可能导致ISV的软件产品（例如MongoDB、Redis等）被容器化、导致行业解决方案容器化（例如交易系统）、甚至导致操作系统容器化（例如RancherOS和其他“极简主义”下支持运行容器服务的操作系统），它促进DevOps工具链的发展，它和CI/CD深度整合，它直接影响开发人员的思考方式。
- Serverless架构：类似Amazon lambda、Google Compute Engine等，对于应用开发者而言交付方式部署方式都是有变化的，能否称的上“运动”，有待商榷，姑妄列出。
- 未知的运动与变革：技术的进步与变化，是加速度的，并且这个加速度本身的变化，也是指数级的，这是“奇点临近”作者库兹

威尔（Kurzweil）的观点。总之，新的变革一定会来，快到你还没学会容器，新技术的“飞饼”已经摔到脸上。

其实，“软件开发”这个世界，依稀也是遵循“合久必分”（Divide-and-conquer）、“分久必合”（Combine-and-conquer）的规律的，例如：

从 Mainframe 向 Client-Server 到 Web/3-tiered 架构的发展，可以算是一个分层分治的过程

Client-Server 架构下的系统一多，造成服务器端资源的浪费，然后以 IBM、Sun、HP 为首的厂商，在本世纪初开始推销所谓 Server Consolidation 的解决方案。随着虚拟化技术的成熟，物理服务器逐渐变成虚拟服务器，世界又回归一个逻辑上分布、物理上集中的“超级 Mainframe”

有些技术，往往是“似曾相识”，是新技术世代下用新技术手段对旧的理念的回归。例如 SOA 架构 +RIA，是对 C/S 架构某种程度上的回归、虽然技术载体大不一样。同理，Micro Services 也体现 SOA 的理念（虽然两者是巨大的不一样，见后文）

容器化，也许算的上近年来最重要的“运动”，很快成为一种潮流 – 无论有无必要，开发工程师的代码都以容器交付，否则就像 Web 1.0 时代还在开发 Client-Server 甚至 Mainframe 的应用都不好意思出去跟人说。这潮流里，有厂商的有目的性推波助澜，有一窝蜂的赶潮流，也有切实的业务场景驱动，无论如何，容器化将（1）常态化 – 尤其是当 ISV 也把它们的产品容器化后；（2）促进分布式架构在传统企业 IT 里的采用（此前，大部分垂直行业 IT 并不擅长互联网企业所擅长的分布式架构，现在只要接受了容器的概念，显然就走向分布式）；（3）促进已经讲了 8 年以上

	引爆点	开发技术载体	软件架构风格	客户端发布	服务器端交付
Mainframe	IBM System/360（第一代多用途计算主机），首次分离“架构”与“实现”的概念	打孔机、汇编、Fortran 77	一体	“笨终端” – Dumb Terminal	见过并且活着的人已经不多。。。
C/S	X86/PC、RISC、摩尔定律、HP/Sun/传说中的 SGI 和 NeXT 工作站	Unix、4GL、Sybase、高级语言、X/Motif、DCE RPC、DCOM、CORBA-IIOP	2 层架构、关系型数据库主导	软件 CD 安装、升级	软件 CD 安装升级、数据库迁移
Multi-tiered	互联网/Web	Struts+Spring+Hibernate、Tomcat、WebLogic、Websphere、Oracle、PHP、Ruby...	3 层至多层 – 展示层、整合层、业务逻辑层、持久层、存储层。。。	浏览器刷新一下页面	手工部署脚本、JAR、WAR、存储过程等等，数据库迁移。开始有 CI
SOA + RIA	互联网技术进入企业	SOAP、REST、Flash/AIR、AJAX、RMI/其他 Remoting、WSDL、UDDI。。。。	对于用户像 C/S，对于开发者是 Multi-tiered	浏览器刷新一下、升级（例如通过 AIR）等	同上
Distributed	Web 2.0、NoSQL（BigTable）、云	函数类、动态、脚本语言，非关系型数据库，一致性算法（Raft、Paxos、Zookeeper），响应式服务器（nginx、Node.js。。。）	Reactive 响应式架构、Heroku 12-factors	多元化 – 手机 App、内嵌浏览器（例如 Webkit、Chromium）的富终端、网站	自动化部署 – Chef、Puppet、Ansible、CI/CD、DevOps 开始
Containerized	LXC、Docker	同上，但更规范（通过 PaaS 如 K8S – 遵循其最佳实践）	同上，但更规范	同上	同上，加不可变基础设施（immutable infrastructure）运维，加基于容器编排技术的 CI/CD
Serverless?	Amazon Lambda	脚本类语言更容易	透明	同上	仅需交付源代码

的 DevOps 落地可操作。

说到传统企业 IT 的技术架构，就不得不提一下“去 IOE”，因为尤其在金融机构，IOE 是无可辩驳的存在。

容器化也许帮助你“去 IOE”

乍一听，这标题有点哗众取宠。但仔细想想，其实还真可以拉扯点关系。

首先，“去 IOE”本身是一个伪命题。企业 IT 降低对一些外国厂商和商业技术的依赖，固然从节省成本上有那么些好处，但如果不上升到“民族产业”、“信息安全”的层面，减少几个数据库软件的 license 对

于企业本身的效益是有限的，其伤筋动骨的技术迁移也许是得不偿失的。去了 IOE，也没什么值得自豪 - 别高估了自己在国家信息安全方面的重要性，如果没有为业务经营、客户利益带来价值，恐怕只能是“然并卵”。

如果我们不带偏见的把 IOE 看成一些象征性的符号而不是具体的某些公司的话，IOE 一定程度上代表了上一个世代的技术，对于只生存在开源技术世界里的互联网企业的年轻工程师尤其如此。IOE 甚至在一定程度上是 Client-Server 架构思潮下的产物，代表了以关系型数据库为中心、以中央存储阵列为主导、以品牌服务器硬件为载体的技术架构风格，这类技术的存在依然有充分必要的业务应用场景，盲目的“去”，只能是自找麻烦和浪费资源。

但换一个角度看，“去 IOE 思维”却又是有意义的，因为在实践中我们已经发现：

- 关系型数据库被企业里的应用开发者们过度滥用。事无大小，都被存入数据库，包括一些配置信息。事务型操作往往也没有控制好粒度，开发者为了“稳妥”起见囫囵吞枣的把CRUD操作都丢到事务里，期望由数据库来解决自己的不求甚解和懒惰。
- 很多问题，其实是可以不用关系型数据库解决的。
- 互联网时代尤其是Web 2.0开启后，从3-tiered向分布式架构演进，RDBMS为中心、高端硬件为依托的架构已经力不从心。
- 就算不搞互联网，传统业务系统在当今这个时代沿用旧的技术架构依然扛不住，2015年疯狂的股市下，高频、高并发、海量的交易就是股票交易系统的梦魇。

“去 IOE”其实最难的是观念的改变，传统企业 IT 的工程师，非常习惯于用关系型数据库的语义、概念作为对业务领域（business

domain) 的建模工具，一言不合就开始设计表结构、画 ER (Entity Relationship) 图。开发过程中，可能大部分时间消耗在 ORM (Object-Relational Mapping) 上，从数据模型出发封装一些对象以便于数据持久层和内存之间关联起来。这导致传统 IT 系统的升级动辄涉及数据库迁移，功能扩展通常导致表结构改变。实际上用关系型数据库的理念对世界进行建模 (modeling)，是有很多局限性的，一是无法对业务逻辑进行抽象，二是无法对业务数据进行封装。这样做的缺点，是所建立的模型无法低成本扩展、重构，以快速应对持续变化的业务场景，在当今这个“只有变化才是唯一的不变”并且变化频率本身是指数级改变（《奇点临近》作者 Ray Kurzweil 所言）的世界，这样的设计导致的显然是一个变更成本非常高的“脆弱系统”，无法拥抱改变应对黑天鹅（关于脆弱系统，见塔勒布《反脆弱》）。

“世界观决定方法论”，中医和西医对人体的建模差别，决定治病的方法截然不同。例如前者用经络、寒热、干湿、虚实、阴阳来描述病理，后者用细胞、基因、细菌、病毒来看待问题，导致同一个疾病的不同处理手段。有些问题用这种模型来看容易解决，有些问题则用另一种模型描述更有效。盲目的用关系型数据库看待一切，是很多问题的根源。

以关系型数据库为中心的应用，一般都是单体应用 (monolithic)，虽然它们可能也会融合一些分布式的技术元素，扩容、扩展、弹性伸缩、响应变更等等这些非功能性需求，依然是它们所难以满足的。在看到这类架构的问题后，技术界开始出现混合编程 (polyglot programming)、混合存储 (polyglot persistence) 和混合处理 (polyglot processing – 例如大数据里的 zeta 架构) 的潮流，微服务 (Micro Services) 的架构风格与理念也逐渐形成。

微服务具体实施的问题在于，停留在“理念”、“最佳实践”层面的东西，很难在一般垂直行业的 IT 落地，因为面向业务的工程师们，关注点不在底层技术细节，无法投入资源去研究自己的平台，凡是能在垂直行业推广的技术，必须是具体有形的工具、API、框架。容器类技术作为看得见摸得着的、同时被运维人员和开发人员使用的工具链，对微服务的开发和运维，提供了无比巨大的推动力。虽然微服务本质上不依赖于容器，但是没有容器技术的支持，微服务在一般企业 IT 里的落地是不乐观的。

这就产生一个非常有趣的副作用 – 一旦技术人员习惯了容器化的观念，他们很可能不知不觉就走上了分布式架构的道路、潜移默化接受了微服务的思维，我们知道技术人员是很容易“心为物役”的，他们的抽象思考往往需要寄托在有型的工具上。例如 Heroku 的 12-Factors（12 律），总结了 12 项在云上开发分布式应用的最佳实践，我们可以看到，采用容器化的架构，很自然的就吻合这些实践。

总而言之，“去 IOE”从它最开始的起源来看其实是去单体应用架构、去“数据库中心主义”，是分布式架构对传统企业技术套路的颠覆，而容器化一旦成为主流，分布式架构即会潜移默化不知不觉中成为企业 IT 架构的主流。

但不要把微服务和容器化本末倒置

容器化既然被说的这么玄乎，那么是不是我们就该蜂拥而去的采用？个人认为，如果阁下的企业环境，并无特别适合“微服务化”的应用，那么个人揣测，阁下也并无采用容器技术的必要，即便是已经采用了 SOA 的技术架构与技术治理，千万别以为就顺理成章可以换一个时髦点的名字“微服务”。

SOA 与微服务，其实有一些本质区别，哪怕是表面上有一些相似性 – 例如都叫“服务”（技术名词有时确实导致巨大的歧义）、都有服务注册与发现机制、甚至具体实施技术有一定重叠。在此列出一些区别（有商榷处，姑妄列出供讨论）：

- 从根本思想看，SOA是强调中央治理（central governance）的，服务之间大致松耦合但实践中其实有一定耦合，例如shared storage是常见的 – 同一个数据库上封装几个服务，避免数据直接暴露到外面，可后面依然是一个数据库；事实上当用到“治理”（governance）这样的字眼，本身就充满了中心化的意味。微服务则以去中心化为原则，强调share nothing、服务间高度松耦合，数据持久层被抽象为backing store – 甚至不需要是关系型数据库，每个服务各有自己的backing store。
- 总体架构设计，SOA是一个top-down思维：“顶层设计”，从业务切分模块，把模块之间关联变成封装服务之间的调用、集成。微服务是一个自下而上的bottom-up的思维，服务的粒度更小（否则何为“微”服务？），对服务本身上下文的假设更少，最重要一点，微服务通常是从当前服务的上下文衍生出来的。
- 组织结构上看，SOA类型的项目团队，模块负责团队隶属于一个更大项目之下，实际中几乎不会独立运营运维。微服务，粒度小，强调单一责任，独立部署运维、自有生命周期。某种角度看，不同SOA项目之间是不同的Silo型团队负责的，而微服务严格来说一个服务对应一个跨职能团队。
- 从关注点看，SOA强调的是SLA、compliance/regulation（合规）、audit（审计）等大型企业特色的“治理”，微服务关注点

从来都是快速响应客户要求和市场变化以及快速创新，是敏捷型的。（所以微服务并不替代SOA）

- 从部署运维角度看，SOA出现于云计算之前，自动化部署、自动化运维并不是它的天然基因。微服务几乎可以说是云计算时代的产物，高度碎片化（与SOA型服务比），高度依赖于解决底层非功能性技术问题的PaaS/CaaS平台，天然符合多租户、需要DevOps支持。

一个微服务通常很可能是通过从现有服务 fork（开分支）、clone（直接复制）、mutate（变种）出来，这很有可能是违反我们在软件工程中一个所谓 DRY（Don't Repeat Yourself）的原则 – 我们已经习惯于认为，剪贴代码是糟糕的、粗暴复制功能不好的，在理想主义的软件王国里，绝对没有拷贝粘贴的代码，也没有冗余的数据，更不应该有复制的服务。

然而，我们对世间事物的认识，往往是螺旋上升的，没有绝对的赞好、也不能武断的说坏，在现实这个不完美的世界里，除了编程大拿、代码洁癖者、技术原教旨主义者，我们必须接受一个现实，就是只关注快速实现业务需求的程序员、普通运维工程师是大多数，粘贴代码、复制部署服务可能就是大部分以业务功能为终极目标、以快速上线为首要任务的普通工程师的本能，quick-n-dirty 是任何团队绕不开的取舍。微服务，通过结合容器技术，一定程度上接受了普通程序员克隆服务、克隆代码的“陋习”。

- 系统升级，在条件允许的情况下，运维工程师最喜欢的可能是部署一套新的，旧的不碰。新的没问题，把旧的关闭；新的有问题，把旧的切换回来。微服务自然是考虑支持同一个服务的多个版本并存的，而容器则刚好是实现“不可变基础设施”（Immutable infrastructure）的最佳套路，这俩一拍即合。

- 同一个服务，也许会逐渐演变成需要接受不同的运营约束（operational constraints）、或者针对不同的用户群，是不断收集新需求、重构代码、升级服务以保持单一服务支持不同业务需求？还是复制代码、克隆服务，在不同环境各自独立发展？对于快速敏捷支持不同的业务线、产品线，也许复制代码克隆服务是一个更高效的做法。

Michael Nygard，一位曾任职私募股权交易机构的架构师，在他的“新常态”系列技术文章中，举了一个例子：他的公司有四十多个不同的交易席位（trading desk），分别在不同的市场采用不同的策略进行交易。每个交易席位都有自己的技术团队负责交易应用开发。如果他们像很多大企业一样采用一套单一的、集中式的交易系统，则任何交易组对系统的变更均会对其他组产生潜在的损害 – 因为变更影响他人、bug产生系统性风险、测试需要更繁复覆盖更充分、变更发布周期需要更长、升级需更复杂慎重、交易系统受影响面更，他把这些潜在能导致失败的影响称之为“失败域”（failure domain）。

Michael 的雇主选择让每个交易组独立维护自己的小型、单一、功能聚焦的交易应用 – 开发工程师和交易员坐在一起工作、小团队作战、快速迭代，以此来最大程度缩小各交易组被动关联产生的“失败域”。这个场景，对于从事证券交易系统开发的工程师，是非常熟悉的。在这里我们看到两个极端的取舍：

- 交易系统从架构和基本功能的角度上看都是大同小异的，以一套理想的、大而全的、集中式的系统服务各条业务线、各个交易市场、各种交易产品，好处也许是集中运维统一监控，服务归一、数据完备，消灭了信息孤岛，公司能获得跨市场、跨产品、跨业

务线的最完整的经营数据，轻易实现合规监管、统一风控。但是这种中心化系统，本身的任何变更都是牵一发动全身，任何缺陷都导致公司级风险，是一个典型的“脆弱系统”。也不利于任何业务线的单独敏捷运作。

- 类似上述私募股权公司的做法，则是完全去中心化，多套交易系统冗余建设，在一定规模的证券公司里，几十套交易系统是常见的，确实产生很多问题 – 例如一个合规要求或交易市场的新业务，必须在几十套系统里变更升级，硬件资源得不到充分的共享利用，不同交易系统往往是异构技术形成一个个竖井（Silo），信息孤岛的形成给统一风控统一经营造成巨大困难，等等。然而，这个去中心化的做法，却是符合”反脆弱“精神的，它把失败域变小。每条业务线有自己的交易员、业务专家、工程师以及系统，响应市场竞争的效率最高。

作为例子，微服务架构很可能对上述情形的解决是有帮助的。首先，在实际操作中，去中心化基本上是共识，避免全局、系统性风险比什么都重要，所以功能相同相似的服务在不同业务线、不同约束条件、不同目标用户环境下冗余部署是必须的；其次，越复杂、越大块头、越黑箱型的代码模块，越难弄明白，越怕被触碰，也就隐含越高风险，微服务化使之增加透明度；第三，一些服务例如交易引擎，架构大同小异但是细节很不一样，与其反复抽象、重构、支持一切交易市场、交易产品、资产类型，还不如克隆一下，把相互依赖以及对某些共同设施的依赖均降到最低，各自迭代发展，让每条业务线获得最不相互掣肘的、最高效率的技术支持。

但是正如我们常说的，“软件工程没有银子弹”，微服务架构带来的更多的碎片化，对架构设计、开发运维挑战更大，对开发者技能要求更高。

例如需要掌握一系列 Cloud-native patterns（原生云架构设计模式诸如 throttling、circuit-breaker、bulkhead 等）。容器技术作为一种工具链和微服务载体，则在此时发生了很大的促进作用，甚至是微服务化实际落地的可行性的一个重大保障。

显然，正如上述例子，我们是因为考虑微服务化单体应用而考虑容器工具，平白无故容器化一个系统则是毫无意义的，“容器化与否”本身也是一个伪命题。事实上，脱离业务特点谈“微服务”本身也是个伪命题——“如果你都不能构建一个有效的单体应用，你凭什么认为微服务能解决你 的问题？”（Simon Brown，“Distributed big balls of mud”）。

采用微服务类架构，则需要调整一些“传统”的观念，例如关系型数据的高度归一性（Normalization）、代码的可重用性（Re-usability）和系统的精益化（Lean），可能不再是无可辩驳的“美德”：越归一则数据关系的变更成本越高、代码越可重用则模块组件的依赖关系越复杂（dependency hell）、越精益的系统则变更越困难，这些都是对构建“反脆弱”型系统不友好的，而且很多时候，强韧性（Resiliency）、灵活性（Flexibility）和效率（Efficiency）之间是有冲突和代价的（还是 Fred Brook 的银子弹理论）。

事实上，随着大数据技术发展应运而生的 NoSQL 运动，一定程度可以说是对高度强调归一性的传统关系型数据库理论的“反动”。在微服务的思潮下，我们的服务首先是不共享任何东西（share nothing），每个服务可能都有自己的持久化存储（backing store），形成所谓的混合存储（polyglot persistence）；其次，基于领域建模的开发（Domain Driven Development – DDD），对于实现微服务更加重要，以完整业务逻辑为中心，并解耦了对开发语言、数据存储技术等等的假设。虽然单体架

构或者微服务架构更恰当的说只是不同的架构风格而不是架构本身，但是，一些开发语言、技术工具确实是更适合或者更导致实现出单体架构系统的，例如上述这位 Michael Nygard 先生就认为，传统的 Java 企业应用服务器市场里的技术，包括 Oracle Weblogic、IBM Websphere、Redhat JBoss 等等，是“鼓励”开发单体架构的技术，这又回归到本文关于“IOE”技术风格的讨论，分布式架构、微服务、容器化，是脱离技术分层（layering）的单体架构风格的，和大部分企业 IT 所熟悉的技术生态截然不同。

微服务、分布式架构是比单体架构更复杂的，可移动零部件特别多，零部件之间的关联关系复杂、通讯耗损大，为了解决这些问题，技术团队需要掌握此前不需要的知识。例如上述的 Cloud-native patterns、Heroku 12 要素、Reactive 技术风格与技术工具等等。一个企业如果没有具备这些知识与能力的技术团队，都不应该去考虑做微服务。

“架构风格”、“发展理念”、“最佳实践”、“技术原则与技术哲学”等这些，通常只适用于优秀的技术团队，对于普罗大众的、以业务功能为开发目标的团队而言，也就是听完一次、佩服一下，结束。只有通过工具才能把这些抽象的东西实际落地，否则它们也就只好留在教科书、别人的 PPT、网上文章里。幸运的是，容器工具链的发展，也许正在成为这么一系列工具，会促进微服务在传统企业 IT 的落地。

会计准则、代码“库存”与进化式架构

在微服务架构的思维下，归一性（normalization）不再是一个绝对的“健康目标”，服务们通过克隆、变异、分叉而诞生，然后迭代发展、存活、消亡，在这过程中冗余、重复是被允许的，而淘汰则是必须的，所谓“有生有灭”，有用的服务（被使用的频繁、被高度依赖）存活，没用的

服务关闭，我们设计微服务不是在一张白纸上凭空绘画，而是不断在现有的服务中派生、演变。这种架构，可以称之为进化式架构（evolutionary architecture）。

进化式架构有一个有趣的作用，就是代码“去库存”。这里说的“代码库存”，不是指代码的技术载体 Git、SVN 等代码库，而是一家企业多年开发积累下来的代码，究竟是“无形资产”（Intangible asset）还是“债务”（Liability）和“库存”（Inventory）。根据美国的会计准则（US GAAP），一家企业的内部自研发软件费用是这么被划分的（粗略罗列）：

- 项目启动前的相关内部和外部费用，被认为是开销（expensed）。
- 自研发供自己使用（internal-use）的应用软件所投入的相关外部和内部费用，被认为是资产（capitalized）。
- 让新系统接入历史数据、或者转换旧数据格式，所投入的费用，被认为是资产（capitalized）。
- 系统投入运营后的相关培训、运维费用，被认为是开销。

显然，一家企业 IT 的 R&D 产出，被认为是公司资产，软件研发的投入是一种投资，所以其产出软件承载着一种期望 – 增加企业的生产力。但软件毕竟不像机房、机器、网络设备那样看得见摸得着，对于大部分的企业管理者而言，无形资产恐怕总是有点说不清道不明，在这个连软件都不再以盒装 CD 加大部头手册的方式卖的时代，IT 管理者们量化自己的无形资产的一个本能，就是自豪的宣称，自己的系统含有多少百万行代码。可是，这几百万甚至上千万行代码，真的是资产吗？

首先，在现实中，一个系统的代码量基本上是逐年增加的，对于大部分团队，系统维护就是增加代码，真正有动力和能力对自己的已上线系统

进行重构的团队是很少的，当一个团队跟老板报告系统代码行数下降百分之十的时候，该老板除了疑惑不解之外可能还有恐惧不安，业务部门和用户也不会为节省了几十万行代码而感谢你给你发奖金。可是代码越多的另一个潜台词，是风险点越多。有些风险是程序员引入的，有些风险，则是因为世界发生了变化、原来的运行环境业务规则发生了变化，原来的假设忽然不再成立，例如用一个十年老的网站改造去支持智能手机出现后的移动应用，原来已经稳定的架构和逻辑在修修补补后就产生新的缺陷。总之，世界的变化总是快于软件的进化的，而积累了十年的几百万甚至几千万行代码，总体来说是很难对变更友好的。这个时候，代码库就变成了库存、技术债、阻碍业务创新的惯性。

过去十几年来的企业软件，我们可以认为大部分是单体架构的，它们中的许多是内部逻辑铁板一块的交织着，甚至没有做到代码级别的松耦合（例如通过良好的面向对象设计与设计模式实践），陷入依赖关系地狱（dependency hell），无用的代码很可能继续存活在系统中从来没被淘汰。新工程师对老系统是拒绝的，因为技术套路和理念两年就过时了；用户对老系统内心是崩溃的，因为让 IT 去维护修改它的效率永远是低下的。作为程序员，我们每个人心底里都讨厌接手他人的代码、抱怨前人的愚蠢设计或怪咖风格，我们基本上倾向于认为历史遗留代码（legacy code）是阻碍生产力的，总是恨不得除之而后快。

例如我们为了不触碰前人的代码，通过模拟接口绕开旧代码、换上自己的实现，然后旧代码就成为一段 dead code，但谁也无法确定它是否死透，因为它也有可能在其他地方被调用，于是谁都不敢把这段代码删除，于是代码只做加法越积累越多，而后来者则理解越来越困难… 可以想象，依赖这样的系统去创新，就像戴着脚镣跳舞。金融行业的应用系统，有很

多这样的东西，庞大而沉重，日益成为厌恶害怕变更的“脆弱系统”，“我就是喜欢你看我不惯却不得不和我一起建设社会主义的样子”，面对这样的技术库存我们很无奈。

普通程序员（尤其是被业务部门蹂躏着、无暇学什么理论的）的“本能”偏好和习惯是这样的：情不自禁的剪贴代码；上线新功能最好部署整套系统而别让我折腾里面的细节，出了问题马上切回旧的那套 – 还好好的在那跑着呢；永远喜欢做新项目写新代码，而不是去消化前一个傻帽的代码；有些代码就是没法子写的优雅的，尤其是临时性运营性的代码，能不能用完即丢… 别轻视这些草根的、不高大上的“陋习”，简单的、符合“本能”的、有“群众基础”的东西，才往往是有生命力的。

微服务架构风格之下，也许更符合草根习惯的开发平台终于有机会出现：高度碎片化，大家都在写小程序，除个别关键服务外一般难以对整个系统产生坍塌风险，任何小程序可以被丢弃被重写，小程序们可能多个版本同时存活但是监控程序会告诉我们哪些已经逐渐没人用可以被销毁，面向一个有略微差别的新客户也许就克隆一个新服务稍微修改一下、避免修改现有服务增加判断条件… 微服务基于业务需要、市场变化、经营策略而持续不断的出现和消失，这就是一个像流水一样的持续变更中的有生命力的系统。

当然，前提是希望容器编排技术结合监控技术和 cloud-native 的各种架构模式把服务底层的运行平台搞定，所以一个强的平台团队依然是需要的。把一个一百万行代码的系统拆成 100 个一万行代码的独立运行协同工作的小程序，绝对不是一件随便可行的事。

当我们有非常好的工具以支持我们在短时间内快速开发出代码、极大程度释放生产力的时候，我们才敢于丢弃过时代码，让服务派生与进化、

用完即扔，才可能不再把历史积累的代码当作什么珠宝般的贵重资产，才不再拖着充满风险的“库存”前行。

从生产工具到生产组织关系

新技术带来新工具，容器只是又一种工具而已。然而，鸟枪换炮不仅需要相应的新操作技能与观念，还需要与之相适应的组织结构，工具使用者之间的协作方式、运作流程、管理模式都需要作相应变更 – “二炮”变成“火箭军”，肯定不仅是改个名字那么简单，是和先进科技武器的发展相适应的。自从微服务开始流行的这两年来，网上越来越多架构师在讨论康威定律，不是偶然现象，是大家不约而同意识到，服务的切分、模块的解偶、技术系统的最终形态，很大程度取决于开发者所在组织的交流沟通形态。

单体架构应用开发的团队，很可能是这么一个 silo 团队：一到多个界面及交互设计师、几个前端开发（Web 1.0 时代，是 HTML 及模板编写者；Client-server 时代，是 MFC、JFC/Swing 甚至 X/Motif 开发者）、几个服务器端开发（EJB/JSP 的、更古老一点是 CORBA/IIOP 或者 DCOM 的）、一两个数据库的 DBA。这种团队虽然有多个角色对应多个技术层，但严格来说不算“跨职能”，因为他们基本上既不运营也不运维自己的应用，业务人员和运维人员并不在团队内。为什么称之为 silo 团队呢？因为多个这样的团队之间，信息是不透明不流动的。

微服务架构下的组织结构，适应如下特点：

- 有Gartner所谓的“外架构”与“内架构”之分，内架构主要围绕微服务的标准化设计，例如每个服务必须是实现单一责任（single responsibility）、服务边界（scope）清晰、接口约

定（contract）明确稳定。外架构主要是平台，聚焦于系统性解决基于内架构的微服务实例的注册发现、编排、资源伸缩、生命周期管理、监控、高可用等等，解决服务碎片化带来的各种共性问题。

- 平台团队负责平台的构建、优化、维护，持续整合新技术、持续向服务开发团队提供工具与公共设施便利，例如利用Kubernetes、Swarm或者Mesos之类的技术构建容器云支持多租户，基于行业（例如金融业）特有环境与要求定制网络与安全解决方案，用符合自身企业环境的技术解决Docker的网络问题，诸如此类。微服务团队则由业务专家和工程师组成，联合维护和保障一个服务的功能、有用性（没人用的服务就没有生命力）、运维，其中工程师很可能是全栈的、开发自运维的 – 虽然系统层面的高可用由平台团队支持，可是服务运行中的业务逻辑故障显然只有负责开发的人自己直接搞定。
- 平台团队主动向企业各业务线的团队布道、宣传自己的工具与平台，争取更多的租户。而微服务的团队，同样需要向其他部门、团队布道、宣讲自己的服务，找到更多的应用场景和使用方。最后，各业务线的应用开发项目，则是在产品开发过程中组合、集成各种服务（同时也可能提供自己的微服务）以服务终极客户。所以，这是一个服务型的、“人人为我、我为人人”的组织。衡量这些团队的表现也好办，看看平台团队争取了多少租户、微服务团队支持了多少应用。

结合容器技术，基于微服务架构的组织，都可以成为DevOps型组织，因为容器工具链也许是迄今为止真真正正让开发与运维共同使用的同一套

工具，把开发、测试与运维工程师通过 CI/CD 串在了一条协作生产线上；此前虽然有 Puppet、Chef、其他所谓实现“infrastructure as code”的工具，实际上不同角色的工程师依然是没有标准化的工具分享的。别少看了共用一套工具的厉害之处，工具所附带的整套技术语言、概念、词汇表，往往成为工程师们交流沟通、描述问题的标准语言，否则他们往往是“鸡同鸭讲”，同一个词汇说的完全不是一回事。

虽然我们一直强调工程师的抽象思维，避免“心为物役”（过度依赖于某种技术、某个开发商的系统所定义的概念词汇来作为自己分析思考问题的基础），可是实际中具备良好抽象思维的人并不多，所以依靠一套比较好的工具作为技术解决方案领域（solution domain）的思考依据，也不是一件糟糕的事情。

掌握新技术新工具，需要传统企业 IT 的组织思维与时俱进，作相应调整、重构。否则，无论打着“互联网+”的旗号还是标榜“科技”（例如所谓“科技券商”或者“金融科技机构”等等），都有点“意淫”的味道，生产组织基因不对，无法有效使用科技生产工具。实际上，任何复杂业务应用系统的架构，都涉及两大因素：技术和人。不考虑人与组织因素的架构，有伪架构之嫌。

纯粹卖容器技术解决方案的厂商，恐怕生意不好做，因为光卖武器而没有相适应的组织结构与操作方式，武器很难推广，尤其是现在传统行业甲方的 IT 恐怕依然没有把虚拟机和容器的本质区分好的环境下。

企业级 IT 向左、科技独角兽向右

企业级的 IT 方案，往往突出一个“大”字：总是庞大而沉重，总是大而全，总是显示出一副很强大的样子，它们强调的，往往是“治理”、

规范、流程、管理… 而不是互联网独角兽们所聚焦的敏捷、把握市场机会、以快打慢、以创新迅速争夺客户… 这两种理念的区别，体现在对变更的适应程度上。

仅就金融科技这个领域而言，技术能适应变更甚至从中获益壮大是非常必要的。Zvi Bodie 和 Robert C. Merton 合著的金融领域经典著作《Finance》一书，定义诠释了金融学：“金融学是一门研究人们在不确定环境下如何进行资源跨期配置的学科”。Nassim Taleb 则在《Anti-fragile》中从金融世界开始讨论如何从无序（disorder）中获利。可见金融世界天然关注不确定、无序、随机、紊乱，金融机构的本质是通过风控从不确定中获利，这就是 Taleb 所说的“反脆弱”。金融机构的科技，是否也需要具备“反脆弱”基因与之相适应？

有技术同行以适应变化的能力、喜爱不确定性的程度为标准，把公司分成两极，一端是拖着几千万行代码“库存”的“雷龙”俱乐部成员们，另一端是轻资产、迅猛、快捷逮住市场机会的“独角兽”们。侏罗纪的雷龙们长五十米重三十顿，最终在没有在物竞天择、适者生存的游戏中存活下来。

独角兽（一个由 Cowboy Ventures 创始人和风投家 Aileen Lee 所发明而被投资界广为使用的概念，专指发展迅猛、所做的创新为世界所未见、具有高度价值的公司），则是不断颠覆行业让雷龙们疲于奔命。一个把自己变成雷龙的例子是 GE，它把绝大部分的软件开发都外包出去。问题是，软件开发商的收入模式就靠替客户开发，开发任务越多、工作量越大、代码越庞大越好，所以，“不管你需要还是不需要，它们会继续替你构建东西”（Michael Nygard “新常态”文），最终这些代码形成一个“价值 15 亿美元的船锚，并且被咨询顾问们镀了一层金以显得有价值”，但实际上这

15 亿的锚不一定是你的资产，也可能是把你的船拖沉的债。

扯远了，脑洞有点大。回归到企业级 IT 的技术方案话题：作为一个在互联网企业和创业公司以及传统 IT 的甲方乙方都呆过很多很多年的技术人，个人虽然一本正经严谨讨论“企业级”解决方案，可是心底里喜欢的是互联网自由散漫那套，那就是开放(open)、轻量(light)、敏捷(agile)、精益(lean)，而不是“企业级套件”、重型中间件和大中央数据库技术的粉丝(它们确实有存在的道理，用的恰当的话，但个人喜好无关合理性)。个人粉的以下例子，上升一下下到哲理高度，是符合极简主义、精益的理念的：

- UNIX，以及此后的Linux，操作系统保持小的内核，开放接口，由大量小程序形成各种小工具，完成更复杂的任务可以通过脚本来粘合各种小工具。
- REST，在SOAP、UDDI、各种“WS-*”前缀的企业标准把Web服务搞的无比沉重后，REST真是一股技术清风，用最基本的HTTP原语，更简单透明轻量的描述了服务。
- Mechanical Sympathy，这个由Martin Thompson等人提倡的理念，指出要把一个高性能的程序做好，程序员必须对计算机底层硬件的基本原理（例如CPU的缓存机制）有一个认识，才能通过合理的数据结构与算法设计，去发挥运算能力（现在的企业软件开发，更像“考古”工作，应用开发者的代码只有一点点，下面是第三方框架层、JVM虚拟机层、并发线程库、操作系统的系统库、操作系统内核… 顶层的工程师可控的东西非常少，大部分对计算机原理不求甚解）。

微服务加容器，和小程序加 UNIX，是否有那么点哲理上的可比性？

正如写 UNIX 小工具的人需要直接了解操作系统，写微服务的人需要直接了解一点容器与容器编排、原生云架构模式 – 这是开发者真正第一次介入云计算，此前基于虚拟机的云计算只对运维有意义，对开发者透明（试问此前有多少开发工程师在自己的应用系统里调用过虚拟机的 API，基于业务需求控制云计算资源？又有多少人在意自己的代码跑在物理机还是虚拟机里？）。虽然不久的将来，容器技术很可能又被大技术厂商们搞成一个个冠以各种高冷术语命名的、不透明的“企业平台”。

套用网上“鸡汤”的常用语，技术的“初心”是提高效率增强生产力。不想变成雷龙的企业，需要引入科技工具。以前建房子用铁铲挖泥，现在用挖掘机。那么问题来了（还不是“挖掘机哪家强”，还没到那份上，挖掘机性能可能都差不多…），一个十年前或者更久远的企业组织、生产关系，能有效掌握新生产工具吗？

微服务、容器神马的，都是浮云，这种现象级的技术浪潮会一浪接一浪，等学会了，又已经过时。所以，本文想扯的观点其实是，去一家“兄弟单位”调研容器技术怎么用、和技术界大牛学习微服务最佳实践等，其实很可能没有什么显著功用，因为很快你会发现超容器、超超容器、去容器化又出来了，当我们总算丢弃了青铜剑进入汉朝斩马剑的时代，发现别人已经在舞弄连人带马都能劈碎的唐朝陌刀。

最可怕的是，还有几个哥们已经在比划着玄铁剑、叫嚣着“重剑无锋，大巧不工”的理论；但最最可怕的是，还有哥们已经在玩剑气搞“无招胜有招”……按照金庸金大侠的理论，最根本的还是内功。企业 IT 的内功是什么的？个人认为是组织结构、体制、文化，在这个时代，只能通过构建有科技基因的、对技术友好的组织，让团队和新技术共同成长（而不是等它成熟再去向“兄弟单位”学习），保持精益的技术文化和理念，才

能“去库存”，稍稍具备一点独角兽的特质：Swift（迅捷）、Nimble（灵巧）。

作者介绍

梁启鸿，广发证券 IT 研发，董事总经理首席架构师。哥伦比亚大学计算机科学系毕业，出道于纽约 IBM T. J. Watson 研究院，后投身华尔街，分别在纽约 Morgan Stanley、Merrill Lynch 和 JP Morgan 等投行参与交易系统研发。本世纪初加入 IT 界，在 Sun Microsystems 大中华区专业服务部负责金融行业技术解决方案。后加入雅虎担任北京研究院首席架构师。三年前回归金融业，企望做些能改变传统面貌的、有趣又有用的事情。

Nginx 何时取代 Apache

作者：刘志勇

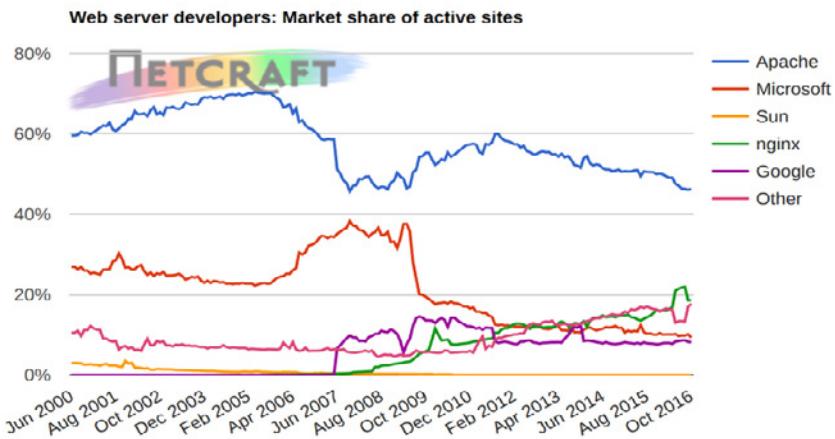
Nginx 和 Apache 都是流行的开源 Web 服务器。Apache 是世界使用排名第一的 Web 服务器软件，是 Apache 软件基金会的一个开源 Web 服务器，几乎所有的计算机平台都可以运行，由于其开放的 API 接口，使得 Apache 有超多的模块，基本想到的都可以找到；加之历史悠久，因此相关文档齐全，因此，长期雄踞 Web 服务器市场的巅峰。

而 Nginx 最初是俄罗斯程序员 Igor Sysoev 开发的轻量级开源 Web 服务器，同时也是一个反向代理服务器和电子邮件代理服务器，并在一个 BSD-like 协议下发行。

Apache 是顶级的 Web 服务器，但是 Nginx 持续增长，而 Microsoft IIS 几十年来首次下降到 10% 以下。

用户自然会关心，Nginx 会不会取代 Apache，以及何时能取代 Apache？Nginx 首席执行官 Gus Robertson 近日就表示，这两者的使用场景并不同，并不存在谁取代谁的问题。

Nginx 已经成为第二大 Web 服务器。它很久以前就超越了微软的



Internet Information Services (IIS)，长期以来，它一直逼近顶级 Web 服务器 Apache。但是，Nginx 首席执行官 Gus Robertson 在接受采访时表示，Apache 和 Nginx 的用户范围并不相同。

Robertson 表示：“我认为 Apache 是一个卓越的 Web 服务器。但 Nginx 和它不是一个相同的使用案例。我们不认为 Apache 是一个竞争对手，我们的客户使用 Nginx 来替换硬件负载均衡和构建微服务，这两者都不是 Apache 干的事。”

事实上，Robertson 发现许多用户同时使用两个开源 Web 服务器，他说，客户在 Apache 的前端使用 Nginx 进行负载平衡和应用，它们的架构完全不同，可以提供更好的并发性能。他还表示，在云配置方面，Nginx 表现会更好。

Robertson 总结道：“我们（Nginx）是唯一仍在增长的 Web 服务器，而其他 Web 服务器正在萎缩。”然而这不是事实。根据 10 月份 Netcraft 的 Web 服务器调查，Apache 本月活跃网站增长最多，获得 180 万；而 Nginx 增长 40 万，仅次于 Apache。

这些增长，加上微软损失的 120 万活跃网站，导致微软的活跃网站份

额下降到 9.27%，首次跌至 10% 以下。Apache 在市场份额提高了 0.19%，并继续占据主导地位，现在拥有 46.30% 的活跃网站。尽管如此，多年来 Apache 一直在慢慢下降，而 Nginx 现在只有 19%。

目前，Nginx 的开发人员正在继续改进开放核心商业网络服务器 Nginx Plus，以求提高其竞争力。Web 服务器使用最新版本的 Nginx Plus Release 11 (R11)，更易于扩展和自定义，并支持更广泛的部署。

最大的提升是动态模块 (dynamic modules) 的二进制兼容性。这意味着为开源 Nginx 软件编译的动态模块可以加载到 Nginx Plus 中。

还意味着开发人员可以利用大量的第三方 Nginx 模块来扩展和添加功能到 Nginx Plus，从一系列开源模块到商业模块。开发人员可以基于支持的 Nginx Plus 核心创建自定义扩展、附加组件和新产品。

Nginx Plus R11 还增加了其他增强功能：

- 改进的TCP/UDP负载平衡：新功能包括SSL服务器名称路由、新的日志功能、附加变量和改进的Proxy协议支持。这些新功能增强了调试功能，使开发者能够支持更广泛的企业应用程序。
- 通过IP地址更好地进行地理定位：第三方GeoIP2模块现已通过认证，并提供给Nginx Plus客户。相比原始GeoIP模块，新版本提供了更为本地化和更丰富的位置详细信息。
- 增强的nginxScript模块：nginxScript是基于JavaScript的Nginx Plus的下一代配置语言。开发者能使用新功能在Stream (TCP/UDP) 模块中即时修改请求和响应数据。

可以预见的是，在这场旷日持久的角逐顶级 Web 服务器的战争中，Nginx 会同 Apache 进行激烈的竞争。而微软的 IIS，则继续缓慢地衰落，走向消亡。

Web 不是未来会赢，而是已经赢了

作者 Niels Klom 译者 大愚若智

过去 20 多年来，Web 已由一个基本的文档共享网络发展成为诞生之初我们根本无法设想的，无所不能的平台。自面世之日起，Web 就在努力迎合用户的各种需求，虽然发展过程中也曾犯过各种错误，但这些问题陆续都已顺利解决。诸如 Flash 和 Silverlight 等插件本有机会统治整个市场，因为 Web 本身当时还在匍匐前进中，很多方面尚不能满足用户的需求。但当智能手机引发的革命使得大家逐渐忘却台式机（和各种插件），转为使用体积更小，同时性能不那么强大的便携设备后，大环境又一次发生了巨变。

“移动”的世界到底是咋回事

距离人们在“水果店”外排队购买初代 iPhone 到现在已经差不多 10 年了，Web 技术至今也还没有全面成为移动领域的“一哥”。“移动化”的 Web 能力依然落后于原生应用甚至桌面浏览器。Flash 虽然算是死了，

但 HTML5 依然没有因此而能称王。然而越来越多的开发者开始使用这种技术开发自己的移动体验，Gartner 称到 2016 年底，超过 50% 的移动应用将会使用混合方式开发。

诸如 React Native 和 Phonegap (Cordova) 等工具为 Web 开发者提供了一种使用 Web 技术构建原生应用的备选方案。这种混合方式在时间和成本方面的效益更高：开发者无需专门为每个移动平台从零开始构建应用，而是可以编写基于 HTML5 的代码，只须编写一次，少量调整后即可重复部署至 iOS、Android、Windows Phone 等平台。

谷歌也在想方设法推进 Web 平台技术的发展。他们通过自家的操作系统 Chrome OS 作为标杆向大家展示 Web 技术的各种可能性，甚至向我们展示了 App Store 本应能做到的一切。Chrome OS 完全依赖 Web 技术，除了可供开发者用在自己的 Chrome 应用中，由浏览器提供的 JavaScript API 之外，不包含任何类型的原生备选方案。但 Chrome OS 并不能算作一种移动平台，尤其是考虑到该系统内建了一个桌面级的浏览器，并且只安装在一些小型笔记本上，因此谷歌又提出了 Progressive Web Apps 的概念，借此打造易用性更强，更接近原生应用体验的 Web 应用。理论上这种应用的实际使用效果还不错，并且应该成为未来几年里 Web 技术的发展方向，但目前这依然仅仅是一个愿景创意，只能用于谷歌自己的 Android 版 Chrome 浏览器，该技术还有很长的路要走。

JavaScript 的革命

虽然大部分人依然认为 JavaScript 只是一种前端 Web 开发技术，但实际上 JavaScript 已经逐渐渗透至现代化计算领域的每个角落。在我看来，Node 是 PHP 的未来。Node 易于学习，易于安装和开发。Node.js 唯

一欠缺的是托管方面的广泛支持。大部分 Web 托管服务依然只支持 PHP，但 Node.js 也为想要尝试新技术的托管公司提供了一个好机会。

IoT（物联网）甚至 VR（虚拟现实）领域也可以见到 JavaScript 的身影。知名的 JavaScript 库 Johnny Five 就可以用于几乎所有新型的微处理控制器（Microcontroller，也许你喜欢与用其他名字称呼这种技术）。Tessel 也很好地证明了 JavaScript 可以实现的丰富用途，它已经不单纯是一种库，更像是一种依赖 JavaScript 的完整设备。

去年这时候 Mozilla 发布了 A-Frame。虽然该技术并未获得应有的广泛关注，但这实际上是一种非常让人吃惊的库。基于 Three.js 构建的 A-Frame 使得开发者可以创建基于 Web 的虚拟现实体验。我自己尝试过这种技术，不得不说这个库太伟大了。问题在于 VR 技术的重要性并不像大家想象中（本应该？）的那么高。也许有朝一日虚拟现实技术会变得极为平常普遍，但也许人们再也不会为其感到激动了。

社区

无论公司或个人，总是需要 Web 的，应用通常包含的内容还是不如网站那么丰富。例如手机银行，虽然可以在银行的应用中办理很多业务，但通常就有些业务无法支持。网站的成本更低，更易于访问，通常来说开发过程也相对更为简单，为 Web 技术背后的开源社区在近年来也已经对 Web 技术进行了大刀阔斧的革新。

诸如 jQuery 和 Bootstrap 等框架使得不同技术水平的开发者可以更容易地进行 Web 开发。一段时间来，似乎接下来需要考虑的就是 Angular 了，然而该技术在向后兼容性方面还没有达到前任所实现的高度。市面上还有大量都很不错的 MV* 库，例如 Backbone、Knockout、Ember，以及 Vue 等，

因此开发者并不需要固守于 Angular，尤其是该技术会使得他们无法继续使用原有的代码。

我认为可以说 Web 开发的未来主要还会落在 React 身上。Facebook 开发的库在 Web 开发社区中曾引起热议，经过不断的完善这些库已成为大家的首选，而不像 Angular 那样做的太过火。我认为，React 未来面临的最大挑战在于接受度。因此问题实际上就变成：网站真的需要 React 吗？

永不停歇的成长

Web 技术依然在以稳定的速率成长着，未来很长时间内还会继续维持这样的步伐。目前预计全球有 34 亿网民，数量已接近全球人口总数的一半。随着另一半人口开始上网，Web 技术将再一次遭遇自己最大的宿敌：浏览器的支持。目前大部分新晋网民居住在新兴市场国家，甚至第三世界国家，他们不可能使用装有 Chrome 53 的全新 Macbook 上网。如果够幸运，他们使用的可能也仅仅是运行老版 Android 浏览器的三星 Galaxy S2 手机。

这就给 Web 开发者造成了另一个问题：如何让自己的网站能被这些浏览器不支持最新功能，并且网速不快的网民顺利访问。很多企业已经因为类似这样的问题错失了大量潜在用户。据估计到 2020 年，印度将有另外 3.5 亿人开始上网，美国人口总数都没这么多。

从此往后

为了拉拢这些新用户，网站需要变得比以往任何时候更轻巧，更易于访问。虽然算不上一个网站，但 YouTube Go 就是个不错的例子，这是一种轻量级版本的 YouTube，专为诸如印度等国家网速不够快的用户打造。你可以关注一下他们是如何通过短时间内进行的一些小幅改动对自己的应用进行优化，使其更容易被网速不高的用户所使用的。这一过程中最难的

部分是如何在自己无法获得切身体会的前提下，发现应用中需要改动的部分，并决定具体该如何改动。谷歌 CEO Sundar Pichai 最近曾提到“对谷歌来说，解决诸如印度用户遇到的此类 [问题] 甚至让谷歌自己得到了创新灵感”，毕竟有些时候少就是多（Less is more）。因此我认为 React 在全球采用率方面无法实现 jQuery 那样的高度，不是因为 React 不够好，而是因为目前并不需要它。

针对未来的想法

有件事是确定的：Web 技术还会继续发展。Web 是一种依然在不断扩张的数字化领域，已在我们每个人的生活中占据了越来越大的比重。对于零零后（L 世代）来说，Web 已经不仅仅是一种技术创新，而成了自己身份的一部分。下一场战争将以 Web 为战场，下一次革命将发生在 Web 上，自由世界的下一任领袖将从现在的 Web 中评选而来。然而 Web 最令人称道的地方在于它是对所有人开放的。Web 不归任何人所拥有，或换句话说，任何人都都是 Web 的主人。但并非所有人都行使了自己的主人翁权利。

也许从今往后的某一天，文盲和非文盲之间的差别不再看文字的运用能力，而是看是否会写代码（书写），或至少能读懂代码（阅读）。那么多人严重依赖自己哪怕连最浅显的理解都不具备的东西，这件事想想就让人感觉恐怖。正如 Steve Jobs 曾经说过的：“在网上，哪怕全球最小的公司看起来也和最大规模的公司一样大”，这句话直到今天依然是真理。Web 比以往任何时候具备更多的潜力。随着越来越多的逻辑开始包含在客户端，随着 JavaScript 成为 Web 上的通用语言，你也需要想方设法保护这些网站，因为 Web 中也将出现越来越多的威胁。

Jscrambler 提供了一种运行时应用程序自保护（RASP）解决方案，

该方案可为客户端 Web 应用程序提供有效的保护，防范运行时攻击。在该解决方案的帮助下，用户的 Web 应用程序可通过针对 JavaScript 量身打造的反调试和反篡改技术（均为流行的应用程序保护方法）自我防护并检测篡改。

腾讯云 11.11： 十分钟内完成弹性伸缩，流量清洗化 解 DDoS 攻击

作者 罗志

一年一度的双十一狂欢节已经来临，这个特殊的节日因为电商们的集体疯狂促销而被赋予了特殊的含义。从往年的数据来看，双十一当天各家平台的成交额和成交量都是在创造着各种数字奇迹，并且每年都保持高速增长。惊人的数据背后也有着不可想象的挑战，特别是在零点之后的几分钟内，数千万的用户在同一时间进行着同样的购物流程，搜索、查看详情、下单、支付等等的每一个环节都需要保证高并发下的高可用性，扩容、冗余、多活、降级、防 DDos 等等的技术都需要渗入到相关的平台架构中。

而对于双十一这样的高并发访问场景，很多中小型电商并不具备相对应的运维能力，所以他们开始借助专业的云服务商来解决这些棘手的问题。这其中，腾讯云就为用户提供了 11.11 电商解决方案，核心能力包括弹性

扩容、抵御刷单等，蘑菇街、聚美优品、当当、有货、唯品会、小红书、楚楚街等知名电商平台都基于他们的云服务构建了自己的业务架构。为了了解腾讯云电商解决方案的技术细节，InfoQ 记者采访了其资深架构师罗志。

电商架构

每家电商都有自己各自的业务特色和技术栈，具体落实的架构不尽相同；但是在我看来，有一些规律可以遵循，这里对不同的电商架构情况进行下分类介绍。各电商业务的架构一般比较类似，都是 4 层架构，即接入层、逻辑层、cache 层和数据库层。

近年来，随着抢购、秒杀的兴起，要求电商后台需要能够有集中时间内应对大请求量的处理能力，因此，在对接入层、逻辑层的按需、实时、快速平行扩展能力就提出了较高的要求。同时由于国内日渐猖獗的羊毛党黑产的泛滥，因此电商业务对腾讯云的基于大数据的恶意用户识别——天御系统，也有了越来越多的需求。

在考虑电商业务架构的时候，以接入层为例，在以往非云架构时，业务往往只会选择一个接入中心，虽然有 BGP 的接入，但毕竟受限于物理距离，故对距离接入中心较远的地区的覆盖，依然是存在不足的，这个问题在移动端 APP 访问时，变得尤为突出。因此在这里，建议更多的考虑如何复用公有云在全国各可用区间强大的内网专线能力，通过反向代理等技术，实现全国多接入中心的接入能力。

但若按业务架构的物理部署而言，又可以大致分为传统和新兴电商两类。其中传统电商，在初创时即使用自有 IDC，故仅在公有云上部署其接入、逻辑、cache 层，充分复用公有云的 BGP 接入及弹性的能力，同时又以专

线将腾讯云与其私有 IDC 数据中心连接成为混合云。

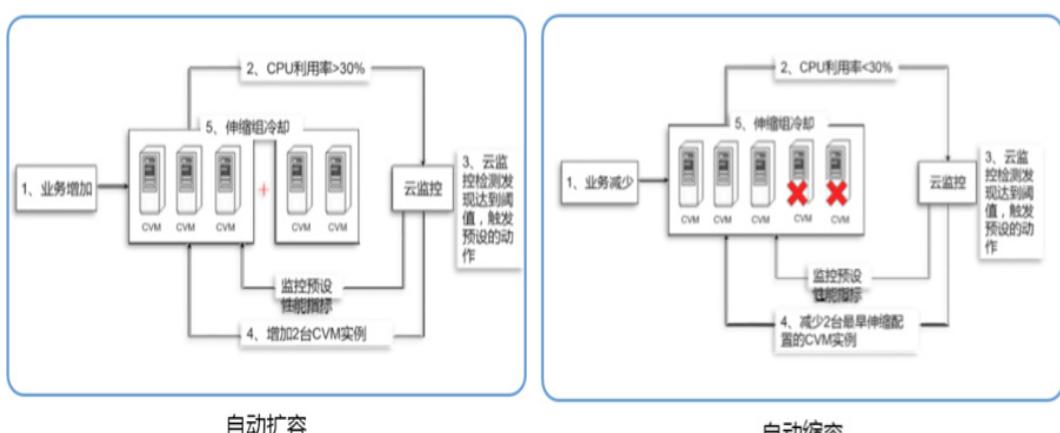
而以小红书为代表的新兴电商，由于其是完全在腾讯云上成长起来的业务架构，故会将全部业务架构部署在云端，充分享受云带给其的低运维成本和低架构成本，而将全部重心投入在业务发展上。

如果某家电商客户同时采用了其他家的云服务或者自有 IDC，那么我认为混合云是一个比较好的发展方向，无论是双云混合还是自有 IDC+ 云。如果是双云架构，建议以双云容灾，一个为主一个为备作为主要的考虑方向。而如果是构建混合云，则以充分利用云的弹性及 BGP+ 多地接入节点作为主要的考虑方向。

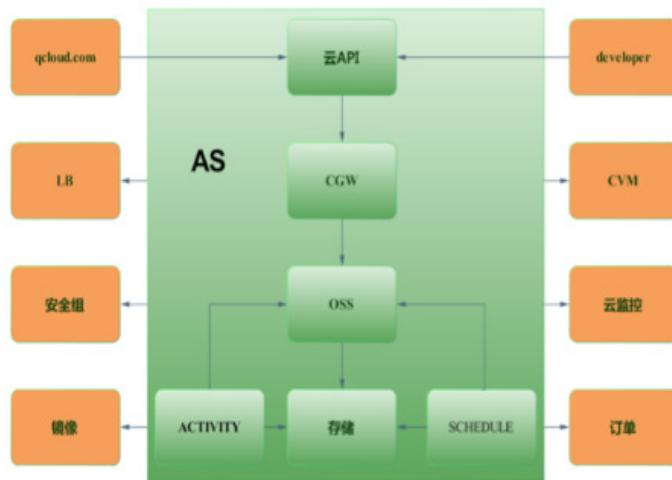
怎样做到弹性伸缩

弹性伸缩（Auto Scaling）根据业务需求和策略，自动调整计算资源。可根据定时、周期或监控策略，恰到好处地增加或减少 CVM 实例，并完成配置，保证业务平稳健康运行。

腾讯云目前的弹性伸缩请求响应时间是 5–10 分钟，扩容时间视客户镜像大小而定，一般 10 分钟以内。同时并发扩容数，无限制。我们的实现是基于业务无状态层的，可自定义策略的弹性伸缩能力。



整体架构如下图：



这里可以举一个我们的客户案例进行分享，江苏盛瑞面临的挑战有三点：

1. 动辄需要数千台规模的扩容，需要稳定、海量资源的后台
2. 过去需要2个以上运维人员晚上加班手工扩容，耗时大半天，费事费力
3. 机器数量巨大，希望找到计费更精确的云平台，节省费用

经过我们的努力最终实现了 2.5 小时完成两次自动扩容，共扩充接近 3000 台高配服务器，此后又完成自动缩容；全程零故障，零人工参与，顺利度过高峰。

尽可能地周全监控服务

对于监控，我们会考虑到云服务器 CPU 利用率、内存利用率、磁盘利用率以及云数据库、Memcached 高速存储等各项云服务负载和性能指标，通过直观图表展示出来。

在设计和实现监控时，我们考虑到了需要支持配置多种指标的告警触发阈值，每个策略可关联不同云产品。设为默认告警策略后，自动关联该

策略，然后还支持自定义告警接收人和发送渠道。关于自定义告警通道服务，我们预留了告警上报接口，客户可以自己通过监控脚本产生自定义的告警，通过告警上报接口，即可将告警内容上报给云监控，平台会及时推送给客户。

同时，不单是监控，云 API 几乎覆盖所有可以以 API 方式对外提供服务的云产品并持续迭代。新产品上线时均同步推出对应 API。所有接口均以 https 方式对外提供，并通过签名机制对调用者身份进行鉴权，充分保证 API 请求的安全性和合法性。在调试上，腾讯云提供了云 API 的调试工具，方便开发者快速调试 API 接口，并提供了丰富的代码示例共开发和参考。

电商业务离不开图片处理

作为电商，图片支持需求是绕不过去的；对应到技术上，需要对商品图片进行存储和识别。这里重点谈谈我们的图片内容识别技术：DeepEye 内容识别引擎。

DeepEye 是由 SNG 优图实验室、TEG 信息安全部联合研发的图像识别引擎，基于海量数据与深度学习技术研发，旨在打造百亿处理级别的高可用、高准确图片内容检测识别系统，对行业内各大产品 UGC 图片 / 视频类数据进行主动色情检测，改变了信息安全领域依赖人工发现恶意内容的模式，升级为机器自动发现 + 人工辅助的 AI 模式；DeepEye 技术在鉴黄上的精度达到 99.95%，在多次业务评测中保持业界领先；在腾讯公司内部已接入包括 Qzone、头像、Q 群、朋友圈等绝大部分图片与视频业务，极大降低了业务监管风险；并联合腾讯云搭建了万象鉴黄、天御等品牌向公司外部客户提供多媒体内容检测服务，迄今接入了包括快手、大众点评、

京东、大智慧、斗鱼等数十家领域内标杆公司，极大程度上解决了人力审核上的痛点并获得了好评。

这里不得不重点提一下 DeepEye 技术的创始团队：SNG 优图实验室，这支人工智能团队是腾讯公司布局前沿技术的一张王牌。从 2012 年成立至今，优图团队从寥寥数人已成长为具备近百人规模的算法研发团队，并于今年从原有社交平台部架构中独立出来，成立了优图实验室，专门从事机器学习、计算机视觉、音频分析等人工智能领域前沿科技的研发与产品落地。其实除了图像内容识别方面的能力，该团队的人脸识别、声纹识别等能力在国际比赛中创造了世界纪录。

越来越受到重视的安全技术

电商常见的三种困扰：恶意刷单、DDoS 攻击、域名劫持，这里和大家分享其中两个。

处理恶意刷单需要积累恶意用户库、有效判定、处理执行。恶意用户的积累需要长久作战，收集 IMEI、QQ、微信、手机号码、身份证号、IP 等全方位的信息；需要基于大数据做到一处作恶、多处识别。

关于 DDoS 攻击的处理，我想谈谈锤子科技的这个案例。2015 年锤子坚果手机发布会，遭到攻击导致直播暂时，高达数十 G 的量攻击，锤子商城一度面临全面瘫痪风险。

此时腾讯云介入，在锤子官网域名的 DNS 配置里面加一条 CNAME，将相应域名解析到大禹，CNAME 功能会把用户流量的请求先路由到大禹的加速点，然后由大禹的加速点清洗、过滤，再吐回到原站，顺利化解了攻击。从启动预案，接入客户网站，到启动清洗，13 分钟内一气呵成，数十 G 的攻击流量均被大禹系统成功清洗，让锤子科技官网恢复正常，保证了现

场发布会召开后 274 万在线用户的顺畅观看。



未来展望

又是一年电商狂欢时，能做的还有更多，更多的腾讯能力不断的跟广大云计算用户的需求相适应，结合电商我们可以做的更多，电商正在面临的转型问题和最新行业发展趋势，腾讯云电商解决方案还提供多样化的产物与服务。例如，许多电商正在尝试的“直播”模式，腾讯云的视频解决方案能够提供全方位、立体化的技术支持，支持直播应用在 24 小时内完成快速对接，拥有完整的直播能力，实现稳定的用户体验，这将为更多电商平台尝试“直播 + 电商”模式提供更多可能性。

而在人工智能层面，腾讯云基于腾讯 18 年的社交大数据经验提供准确的图片识别能力，可对不良内容进行准确甄别，识别精确度达 99.95%，节省用户 95% 的人力；同时，腾讯云提供智能语音服务，日核查语音时长提升 31 倍，服务满意度提升 23%，从资源设备到人力成本全面实现优化。使用腾讯云智能语音服务后，珍爱网语音质检业务的覆盖率从

3% 提升到了 100%，质检效率提升 31 倍。与传统人工方式相比，同样数量的通话录音，质检成本仅为原来的 10%。

嘉宾介绍

罗志，腾讯云资深架构师。 2005 年加入腾讯，先后在腾讯集团内负责过系统运维、DBA、游戏运维、游戏运营规划、腾讯云售前架构师团队。目前主要致力于帮助合作伙伴评估、规划、重构系统架构，建立满足其业务未来至少 2-3 年发展需要的云端架构整体解决方案。

究竟什么样的技术 Leader 是称职的？

常言道：「兵熊熊一个，将熊熊一窝」，Leader 常常决定着团队能力的上限，其重要性不言而喻。那么，究竟什么样的 Leader 是称职的？什么样的 Leader 是不称职的呢？

称职技术 Leader 的日常



作者 | 黄勇（特赞 CTO，已获得授权）
 漫画 | 王小威
 策划 | Betty

不称职技术 Leader 的日常





架构师 月刊 2016年11月

本期主要内容：如何使代码审查更高效，用10%的自主时间提升学习，京东 Nginx 平台化实践，携程基于 Storm 的实时大数据平台实践，基于 Lambda 架构的股票市场事件处理引擎实践，多形态 MVC 式 Web 架构的分类



云生态专刊 09

《云生态专刊》是 InfoQ 为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。



顶尖技术团队访谈录 第七季

本次的《中国顶尖技术团队访谈录》·第七季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱



架构师特刊 Apache Kylin实践

值此 Apache Kylin 开源两年之际，InfoQ 将之前发表的 Kylin 相关的文章集结成册，向社会发行电子书。