

# 架构师

ARCHITECT



## 人物 | People

Databricks连城谈Spark的现状

## 观点 | Opinion

PaaS，不是银弹

Spark的现状与未来发展

Shellshock漏洞证明是时候放弃CGI技术了

## 特别专题 | Topic

微观SOA：服务设计原则及其实践方式

基于微服务架构，改造企业核心系统之实践



# 卷首语

## 角度的不同

最近跟业内某大哥聊天，聊到企业级 IT 和互联网 IT 做事情和看事情的不同。

**企业级 IT：**企业业务层逻辑庞大而复杂，企业软件开发人员懂业务层逻辑但不懂系统不懂数据库更不懂硬件，IT 管理员了解各个部门每天发来的各种新需求但不会编程，整个体系规模往往没有激烈增长因此对可伸缩性要求一般。业务因 IT 中断造成的损失往往远大于 IT 方案的采购费用，所以该体系对可用性、数据健壮性的要求较高。越是庞大复杂的体系，能满足该体系的 IT 解决方案就越是有限，买方对该方案的需求越是难以回避，买方对高价格的耐受力也越高，买方跟卖方的服务关系就越持久、忠诚度越高而难以替代。

**互联网 IT：**大部分业务不是 Web App 就是移动 App，企业规模往往比较小，核心成员当中多半有一个可充当全栈工程师的极客，运用通用的开源技术栈可解决绝大多数需求。整个体系规模可能呈现不可预测的大涨大跌，因此对可伸缩性有强需求。业务中断往往不会造成什么严重损失，因此对可用性、数据健壮性的要求不高。越是通用性强的业务，对技术上越是不会产生太特别的需求，买方越是倾向于选择价格相对低廉的服务，忠诚度低。

**企业级 IT 公司做云计算：**我们的老客户们说想要“云”这个东西增加效率，那这样，我们可以把原来的解决方案给服务化，做一个可以自助操作的控制台给他好了。另外那个啥，出去把市面上那些看起来做的不错、报价几亿几十亿美金的云计算初创企业给我挑一两个买回来吧。

**互联网公司做云计算：**我们处理网站应用的可伸缩性已经经验丰富了，我们可以先把那些要做网站的都招呼到我们这儿来。如果有企业用户来用，提出需要一些企业级 IT 的功能，我们就按照用户呼声高低的次序尽快做给他们吧。如果实在赚不到钱、做不下去的话，就卖了吧。

.....

当然了，“企业级”、“互联网”只是两个标签——而且多半是 IT 历史发展历程中的两个临时性标签。现在大企业也做移动 App，互联网公司也做金融，所以这两个象征性的标签，无非是代表业务系统不同的复杂度、不同的变更模式、不同的技术需求而已。老牌企业级 IT 方案的积累厚但是迭代慢，互联网技术的底子薄但是更新的快；双方的云计算在现阶段还谈不上抢客户竞争的关系，因为两者的客户群还远没有到重叠的地步。像是 CIA 的那个单子的情况虽然也有，毕竟还是个例，只是擦枪走火，还算不上是战争爆发。

继续再往下发展下去，老牌企业级 IT 如果能展示更强的创新能力，并不难把客户的心多留住几年；互联网做的云如果能快速吸收企业级 IT 的更多能力，也会自然而然的吸收更多的企业客户入驻。光是做这两件事情就已经忙不过来了，更远的事情，谁能说得清呢？

本期主编：杨赛

# ArchSummit

全球架构师峰会 2014

2014.12.19-20 北京国际会议中心



互联网金融  
研发体系构建  
云计算解决方案专场  
电商，不是搭个平台就能赢

- 转型中的SNS
- 智能硬件，更懂你
- 移动互联网，随时随地
- 云计算与大数据，从技术选型说起

# 目 录

## 人物 | People

Databricks 连城谈 Spark 的现状

## 观点 | Opinion

关于 Web-Scale IT 的一些观点

Shellshock 漏洞证明是时候放弃 CGI 技术了

Pivotal 和 EMC 谈下一代数据湖技术：Tachyon + Spark 将极为重要  
PaaS，不是银弹

## 专题 | Topic

Spark 的现状与未来发展

微观 SOA：服务设计原则及其实践方式（上篇）

微观 SOA：服务设计原则及其实践方式（下篇）

基于微服务架构，改造企业核心系统之实践

## 推荐文章 | Article

Apache Tez 是什么？

万台规模下的 SDN 控制器集群部署实践

OS X 和 iOS 中的多线程技术

## 特别专栏 | Column

Kubernetes 系统架构简介

一个 OpenStack 访问请求在各组件之间的调用过程

## 避开那些坑 | Void

你的数据库危机四伏

说说远程团队协作的故事

项目初始会议 —— 如何在一次会议中达成共识

封面植物——巨人柱仙人掌

# Databricks 连城谈 Spark 的现状

作者 张天雷

连城目前就职于 DataBricks，曾工作于网易杭州研究院和百度，也是《Erlang/OTP 并发编程实战》及《Erlang 并发编程（第一部分）》的译者。近日，InfoQ 中文站编辑跟连城进行了邮件沟通，连城在邮件中分享了自己对 Spark 现状的解读。

**InfoQ：有专家侧重 Storm，您则是侧重 Spark，请简单谈谈这两者的区别和联系？**

**连城：**Storm 是一个流处理系统，而 Spark 的适用范围则宽泛得多，直接涵盖批处理、流处理、SQL 关系查询、图计算、即席查询，以及以机器学习为代表的迭代型计算等多种范式。所以我想这个问题的初衷可能是想问 Storm 和 Spark 的流处理组件 Spark Streaming 之间的区别和联系？

Spark Streaming 相对于传统流处理系统的主要优势在于吞吐和容错。在吞吐方面，包括 Storm 在内的大部分分布式流处理框架都以单条记录为粒度来进行处理和容错，单条记录的处理代价较高，而 Spark Streaming 的基本思想是将数据流切成等时间间隔的小批量任务，吞吐量显著高于 Storm。在容错方面，Storm 等系统由于以单条记录为粒度进行容错，机制本身更加复杂，错误恢复时间较长，且难以并行恢复；Spark Streaming 借助 RDD 形成的 lineage DAG 可以在无须 replication 的情况下通过并行恢复有效提升故障恢复速度，且可以较好地处理 straggler。

除此之外，由于 Spark 整体建立在 RDD 这一统一的数据共享抽象结构之上，开发者不仅可以在单套框架上实现多种范式，而且可以在单个应用中混用多种范式。在 Spark 中，可以轻松融合批量计算和流计算，还可以在交互式环境下实现流数据的即席查询。Storm 相对于 Spark Streaming 最主要的优势在于处理延迟，但百毫秒至秒级延迟已经可以覆盖相当多的用例。更详细的分析比较可以参考 Matei Zaharia 博士的论文 An Architecture for Fast and General Data Processing on Large Clusters 的第四章。

**InfoQ:** 2014年初您加入 Databricks 这个数据初创公司，当时是怎样一个契机触动了您？

**连城：**我于 2013 年六月第一次接触 Spark。此前函数式语言和分布式系统一直是我最为感兴趣的两个技术方向，而 Spark 刚好是这二者的一个很好的融合，这可以算是最初的契机。而深入接触之后，我发现 Spark 可以在大幅加速现有大数据分析任务的同时大幅降低开发成本，从而使得很多原先不可能的工作成为可能，很多困难的问题也得到了简化。Spark 的社区活跃度也进一步增强了我对 Spark 的信心。有鉴于此，去年十月份刚得知 Databricks 成立时便有心一试，并最终得偿所愿。

**InfoQ：**您之前翻译的图书都是跟并发和分布式相关，请您介绍一下 Spark 在并发和分布式上的设计？

**连城：**分布式系统设计的一大难点就是分布式一致性问题。一旦涉及可变状态的分布式同步，系统的复杂性往往会陡然上升。而 Spark 则较好地规避了这个问题。个人认为原因主要有二。

1、Spark 是一个大数据分析框架，其本身并不包含任何（持久）存储引擎实现，而是兼容并包现有的各种存储引擎和存储格式，所以规避了分布式存储系统中的分布式数据一致性问题。

虽然函数式语言还远未成为主流，但在大数据领域，以不可变性（*immutability*）为主要特征之一的函数式编程却已经深入骨髓。扎根于函数式编程的 MapReduce 固然是一大原因，但我猜想另一方面可能是因为在大数据场景下，单一节点出错概率较高，容错代价偏大，因此早期工程实践中一般不会在计算任务中就地修改输入数据，而是以新增和/或追加文件内容的方式记录中间结果和最终结果，以此简化容错和计算任务的重试。这种对不可变性的强调，大大削减了大数据分析场景下的数据一致性问题的难度。上层框架也因此得以将注意力集中在容错、调度等更为高层的抽象上。

2、在并发方面，和 Hadoop 的进程级并行不同，Spark 采用的是线程级并行，从而大大降低了任务的调度延迟。借助于 Akka 的 actor model，Spark 的控制位面和数据位面并发通讯逻辑也相对精简（Akka 本身也的确是跟 Erlang 一脉相承）。

**InfoQ:** Spark 社区现在是空前火爆，您觉得其流行的主要原因是什么？

**连城：**可能是大家受 Hadoop 压迫太久了吧，哈哈，开玩笑的。我觉得原因有几点：

- Spark 在大大提升大数据分析效率的同时也大大降低了开发成本，切实解决了大数据分析中的痛点。
- 通过 RDD 这一抽象解决了大数据分析中的数据共享这一重要问题，从而使得开发者得以在单一应用栈上混合使用多种计算范式打造一体化大数据分析流水线，这大大简化了应用的开发成本和部署成本。
- 简洁明了的接口。我曾经碰到这么一个真实案例，一位 Spark contributor 用 Spark 来做数据分析，但他的数据量其实很小，单机完全可以处理，他用 Spark 的主要理由就是接口简洁明了，写起来代码来“幸福指数”高。当然另一个重要原因是因为今后数据量大起来之后可以很方便地 scale out。
- 对兼容性的极致追求。面对资产丰富的 Hadoop 生态，Spark 的选择是全面兼容，互惠共赢。用户无须经历痛苦的 ETL 过程即可直接部署 Spark。这也是 Cloudera、MapR、Pivotal、Hortonworks 等 Hadoop 大厂商全面拥抱 Spark 的重要原因之一。

**InfoQ:** Spark 目前好像还没有完全大规模应用，您觉得开发者主要的顾虑在什么地方？

**连城：**由于大数据本身的重量，大数据分析是一个惯性很大的技术方向，相关新技术的推广所需要的时间也更加长久。我个人接触到的案例来看，Spark 用户的主要顾虑包括两点：

- 1、Scala 相对小众，认为相关人才培养和招聘上会比较吃力。这个问题我认为正在缓解，而且有加速的趋势。
- 2、对数百、上千节点的大规模集群的稳定性的顾虑。实际上一千节点以上 Spark 集群的用例已经出现多个，其中 eBay 的 Spark 集群节点数已超过两千。

**InfoQ:** 您最希望 Spark 下一版本能解决的技术难题是什么？

**连城：**我近期的工作主要集中于 Spark SQL，在 1.2 的 roadmap 中最为期待的还是正在设计当中的外部数据源 API。有了这套 API，用户将可以在

Spark SQL 中采用统一的方式注册和查询来自多种外部数据源的数据。Cassandra、HBase 等系统的深度集成将更加统一和高效。

## InfoQ：在部署 Spark 集群、设计 Spark 应用时有哪些方面的问题需要考量？

### 连城：

- 集群部署方式，standalone、Mesos 和 YARN 各有千秋，需要按需选用。
- 在单集群规模上，也可以按需调整。Yahoo 和腾讯采用的是多个小规模卫星集群的部署模式，每个集群都有专用的目的，这种模式故障隔离更好，可以保证更好的 QoS。同时业内也不乏 eBay 这样的单体大集群案例，其主要点在于更高的集群资源利用率以及对大规模计算的应对能力。
- 与现存数据分析系统的对接。对于常见系统如 Kafka、Flume、Hive、支持 JDBC 的传统数据库，可以利用 Spark 提供的现成接口；对于 Spark 项目本身尚未涵盖的，或是私有系统的对接，可以考虑开发自定义数据源 RDD。在 1.2 版本以后，也可以考虑通过 Spark SQL 的外部数据源 API 来对接现有结构化、半结构化数据。
- 合理挑选、组织需要 cache 的数据，最大限度地发挥 Spark 内存计算的优势。
- 熟悉并合理选用恰当的组件。Spark 提供了多个可以互操作的组件，可以极方便的搭建一体化的多范式数据流水线。
- 和所有其他基于 JVM 的大数据分析系统一样，规避 full GC 带来的停顿问题。

### 采访者简介

张天雷（@小猴机器人），清华大学计算机系博士，熟悉知识挖掘，机器学习，社交网络舆情监控，时间序列预测等应用。目前主要从事国产无人车相关的研发工作。

查看原文：[Databricks 连城谈 Spark 的现状](#)

# 关于 Web-Scale IT 的一些观点

作者 阮志敏

[FIT2CLOUD](#) 联合创始人阮志敏近日通过邮件向 InfoQ 中文站分享了他对 Web-Scale IT 的一些观点，当中列举了 Web-Scale IT 的典型特征，并对企业实现 Web-Scale IT 提供了一些建议。以下是邮件内容。

---

Gartner 近日发布了 [2015 年十大 IT 趋势预测](#)，其中包括 Web-Scale IT。Gartner 指出：未来将有更多的企业以亚马逊、谷歌、Facebook 等互联网科技巨头的方式去思考、行动和打造应用程序和基础设施。传统企业渴望拥有和互联网公司一样的 IT 能力去进行业务创新，同时提高 IT 运营效率、降低费用。Web-Scale IT 和国内的"去 IOE"运动有着相同的逻辑，都是互联网企业影响、颠覆传统 IT 解决方案的一种趋势。

Web-Scale IT 和传统 IT 不仅在技术上有差异，而且在文化上也有差异，所以企业也并不是要完全从传统 IT 转向 Web-Scale IT。比如，一些交易型应用/System of Record 应用仍然离不开传统 IT，传统 IT 有其存在的价值和理由。但是，企业应该开始用 Web-Scale IT 的方式去构建一些新型的社交化、移动化、面向外部用户的应用。企业应该怎么做才能实现 Web-Scale IT 呢？

我认为 Web-Scale IT 有 5 个典型特征：

1. 拥有成千上万台机器的计算能力。
2. 只需少数工程师就可以运维大量机器。
3. 应用可以承载非常高的用户访问量。
4. 即使有些机器出现故障，应用仍然工作正常。
5. 应用每天可以升级部署应用好几次。

## 1、拥有成千上万台机器的计算能力

这里的机器是指商品化服务器



(Commodity)。企业可以通过两种路径实现这个目标，一是采用公有云，二是自建方式/私有云。目前我们观察到的情况是，国内对 IT 需求稍大一些的企业，都倾向于通过采用自建方式。而在国外，采用公有云的比例会高一些，比如像 Netflix 等，在 AWS 上面的虚机数量超出 1 万台，仍不自建数据中心/私有云。随着国内公有云服务的完善、API 的开放、费用的进一步下调，更多的企业应该会优先考虑采用公有云。

很多企业实施"Go to Cloud"战略的第一步就是构建自己的私有云服务，而不是采用公有云，这种策略值得商榷：对企业而言，基于云的可编程特性，学习在云中开发、部署和运营大规模、分布式的云应用是更重要的，而不是将全部精力放在构建私有云基础设施上。一种更好的做法是，比如，企业可以先使用青云公有云，如果业务取得成功、使用的规模越来越大，可以考虑基于青云构建自己的私有云并托管给青云来运维。

## 2、只需少数工程师就可以运维大量机器

如果用户采用公有云，那么这个问题就转变成：如何高效管理成千上万台虚机？这包括：

1. 如何快速启动部署一个应用所需要的一系列资源。
2. 如何给这些虚机做配置。
3. 如何一键给这些虚机打 patch。
4. 如何对这些虚机进行统一监控和告警。
5. 如何进行安全和费用统一管理。

实现这些的核心是要充分利用云的 API 来实现自动化管理，用户可以采用云服务商提供的工具（比如 Cloudformation、OpsWorks），也可以采用第三方管理工具（如 Rightscale）也可以自己基于云服务 API 构建自己的管理部署工具（比如 Netflix）。

若用户采用私有云，用户可以自己运维也可以采用托管方式。如果是自己运维，则不仅需要对整个私有云解决方案非常熟悉，而且需要投入相当的开发和运维人力，同时伴随着很大的风险。而想减少运维人员，托管、外包方式应该是更好的选择。

## 3、应用可以承载非常高的用户访问量

Web-Scale IT 通过以下两个方面来实现这个特点：

1. Web architecture/SOA 服务架构，把整个系统分为松耦合的组件，尽量实现无状态，使得组件可以水平扩展。
2. 充分利用云来动态、快速地创建资源（scale-out），根据各层次的监控数据来实现自动伸缩。

传统 IT 则是更多地通过 Scale-up 方式来进行扩容，和 Web-Scale IT 相比在架构方面也有些差异，比如，采用 session 复制方式也多于基于 Redis/Memcached 的集中式 session 方案。

在云环境下，云服务除了提供计算、网络和存储资源外，还提供负载均衡服务、缓存服务、RDS 服务等很多上层服务。云服务商负责这些服务的创建并提供自动化运维方案以实现确保安全性、可用性和扩展性。企业也可以通过 API 动态管理这些服务。从这个角度看，在云中开发、运维应用和传统方式有很多不同，企业需要进行相应的转变。

#### 4、即使有些机器出现故障，应用仍然工作正常

传统 IT 可以通过 Infra 层面的冗余来保证系统的高可用性，而 Web-Scale IT 是构建在商品化硬件上，其冗余度比较低。Web-Scale IT 在 Infra 层次的 SLA 是无法保证的，这就需要用户 Design for Failure。这不仅仅需要在 Infra 层次部署架构方面做相应的设计，如把应用部署在两个可用区、使用负载均衡服务等，在应用本身的逻辑架构、中间件、数据库存储层等方面做针对性的设计，同时在应用本身的管理上也需要有相应的设计，如数据的备份、恢复、动态地更改 DNS、更改负载均衡的后端配置等。

#### 5、应用每天可以升级部署应用好几次

传统 IT 对变更有着严格的控制和要求，开发和运维团队有着明确的分工和责任，应用要实现每天十几次的变更是不太可能的。

Web-Scale IT 能够实现持续部署和交付，不单是靠自动化工具，更重要的是依靠流程、组织、文化上面的变革。很多时候，思维方式、文化方面的改变比技术上更重要。因此 Gartner 说，开发与运营之间的高效协作配合(DevOps)是实现 Web-Scale IT 的第一步。

### 小结

从上述分析可以看出，要向 Web-Scale IT 转变，企业需要实现在 Infra、平台和管理部署工具、应用架构，流程、组织和文化等各个层面的改变。云计算的进一步普及会加速企业在 Infra 层面的转变，而另一方面，随着各个行业云标杆用户的出现，会示范和引导更多企业去实现其他层面转变。

---

## 作者简介

阮志敏是 AWS 认证解决方案架构师(专业级别), [FIT2CLOUD](#) 联合创始人, 长期关注于如何使用云服务进行业务创新。[FIT2CLOUD](#) 是一个云管理及 DevOps 协作平台, 旨在帮助开发人员、运维人员实现应用全生命周期的自动化管理, 提升云使用成熟度。

查看原文: [关于 Web-Scale IT 的一些观点](#)

# Shellshock 漏洞证明是时候放弃 CGI 技术了

作者 曹知渊

最近，被类 UNIX 系统广泛使用的 Bash 软件曝出了一系列已经存在数十年的漏洞（[Shellshock](#)），在业界引起了非常大的影响。不少 Linux 发行版本连夜发布了修复版本的 Bash，在服务器领域占有不少份额的 FreeBSD 和 NetBSD 已经默认关闭了引起漏洞的功能。InfoQ 也及时带来了关于 Shellshock 的[详细报道](#)。

在这个漏洞的风波逐渐平息之余，不少业内人士也在思考，它为何波及如此之广，影响如此之大。[InfoWorld](#) 的专栏作者 Andrew C. Oliver 在一篇[文章](#)中表达了自己看法，他认为 CGI 技术的普及是个错误，正是因为 CGI 技术的不合理之处，Shellshock 才有机可乘。

CGI 技术是 Web 技术刚兴起的时候发明的，它是最早的可以创建动态网页内容的技术之一。它会把一个 HTTP 请求转化为一次 shell 调用。而 Shellshock 的原理是利用了 Bash 在导入环境变量函数时候的漏洞，启动 Bash 的时候，它不但会导入这个函数，而且会误把函数定义后面的命令也执行一遍。在有些 CGI 脚本的设计中，数据是通过环境变量来传递的，这样就给了数据提供者利用 Shellshock 漏洞的机会。对此，Oliver 抱怨道：

为什么有人会认为，通过 HTTP 请求给一个陌生人访问 shell（哪怕是受限的）的机会是一个好主意呢？我不理解。

Oliver 把 CGI 技术比作“上了膛的武器”，程序员必须非常谨慎地使用它，写出优秀的脚本。但在现代的商业实践中，雇佣优秀程序员已经不是一个必选项，大量的廉价程序员很多时候也能合力完成工作。能写出考虑周全的 CGI 脚本的人越来越少，这也使得 CGI 技术更不合时宜了。

Oliver 甚至觉得，用 C 语言编写的动态网页程序都要比 CGI 好一些，因为避免 C 语言的缓冲溢出问题其实并不难，如果程序员不给自己挖坑，并且认真做好单元测试的话，问题不会太大。

编写水平糟糕的 CGI 脚本，确实是互联网上可利用漏洞最多的技术之一了。当然网络、计算机架构等也有可能存在漏洞，但是 CGI 从诞生之初就是一个设计错误，经历此次 Shellshock 风波，其弱点也再次暴露在公众面前。Oliver 也呼吁大家，逐步放弃 CGI 技术，“先移除那些暴露在公众访问之下，又需要非常仔细编写才能不出问题的脚本”。

---

感谢郭董对本文的审校。

查看原文：[Shellshock 漏洞证明是时候放弃 CGI 技术了](#)

# 2014AWS技术峰会北京 晚场活动之讲师见面会

2014AWS技术峰会将于12月12日在北京国际饭店会议中心召开，这是AWS技术峰会首次来到中国。AWS将携手客户展示AWS一系列最新的产品及服务，分享客户的成功案例及实践经验，交流国内外最新技术，现场更有由AWS讲师亲自讲授的动手实验培训课程。

在2014AWS技术峰会晚间，InfoQ主办了讲师见面会，让参会者可以近距离的与大会讲师进行交流。活动将围绕大数据、游戏、网站应用、企业云、电商、移动应用、运维、扩展性可用性与高性能等9大话题进行Lean Coffee式的分享讨论。以讲师和现场参会者为中心，现场选出大家有意愿参与或者分享的话题，然后在不同的时间段进行讨论，其目的是以一种轻松的方式，让大家社交起来，沟通起来。

同时，在现场设置了有趣的闯关游戏及丰富的奖品，包括AWS定制T恤、技术图书、AWS服务抵扣卷、AWS Summit纪念品等。

时间：12月12日 18:30~20:50

地点：北京国际饭店会议中心

详细内容您可以到InfoQ官方网站了解。



# Pivotal 和 EMC 谈下一代数据湖技术：Tachyon + Spark 将极为重要

作者 杨赛

在纽约举办的 2014 年 [Strata+Hadoop World](#) 大会开幕的前一天，Pivotal 在官方博客上发布了一篇名为《[数据湖（Data Lake）的未来架构：基于 Tachyon 和 Apache Spark 的 In-memory 数据交换平台](#)》的文章，表达 Pivotal 与 EMC 对下一代数据湖技术的展望：

下一代数据湖技术的关键在于 In-memory 处理的普及+能够在单一环境下支持多重数据分析负载的架构。

文中表示，[Pivotal Big Data Suite](#) 一直以来的理念是将数据湖作为企业内所有数据的中心化仓库，这样的好处是可以对所有的数据——无论是内存数据还是磁盘数据——进行 SQL 级别的处理，同时具有将多种计算范式持久化的能力。然而随着高性能内存的性价比越来越高，内存数据库的相关技术与企业越来越成熟，Pivotal 认为未来的数据湖将基于一种组合式的新架构：磁盘存储+内存处理的混合框架。

Pivotal 选择了 [Tachyon](#) 和 [Spark](#) 这两个开源项目作为此新架构的基础。其中，Tachyon 作为其内存数据交换平台，而 Spark 作为内存计算层。文章在末尾处表示，Pivotal 相信 Tachyon 会给 HDFS 这样的文件存储与内存处理的交互方式带来革命性的变化，并展望 Tachyon 会成为其 [Pivotal Big Data Suite](#) 的中心数据交换层。

InfoQ 此前对 Spark 项目有过[为数不少的报道和介绍](#)，这个孵化自 AMPLab 的项目在过去两年间受到业界的广泛关注，被视为实时数据处理的一个优先选项。Tachyon 项目也是来自于 AMPLab，最早[在 2012 年底对外发布](#)，是一个相对年轻的项目，在最近也受到了越来越多公司的关注，这包括雅虎、红帽、Intel 还有 EMC——EMC 已经在其[闪存产品 DSSD](#) 和 [Isilon](#) 中尝试集成 Tachyon，Pivotal 的文章中称 Tachyon 是 AMPLab 历史上成长最快的项目。

根据[该项目官网](#)的介绍，Tachyon 是一个内存分布式文件系统，效果是“在 Spark 或 MapReduce 等集群框架中实现内存级速度的跨集群文件共享”。它具有类 Java 的文件 API、兼容 Hadoop MapReduce 和 Spark、底层文件系统可插拔等特性。

InfoQ 中文站针对此事采访了 Tachyon 项目的负责人、UC Berkeley AMPLab 的博士候选人李浩源，沟通内容如下。

**InfoQ：**很高兴看到 Tachyon 得到越来越多的关注。Tachyon 是你的博士研究方向，当时为何选择了这样一个课题？

**李浩源：**一方面是个人兴趣，一直以来，我对存储有很大的兴趣，因为计算机数据处理流程分为读取、处理分析、以及写入，前后都是都是有存储系统来完成。另一方面是机会，我是 3 年前加入 UC Berkeley AMPLab 的，实验室有过很多成功的项目，比如 Apache Mesos 和 Apache Spark，但是它们一个是计算机集群资源调度层，一个是并行计算层，还没有一个存储层的支撑。两方面结合，我就选择了这个课题。

**InfoQ：**在 Github 上看到现在参与 Tachyon 项目的开发者和企业也有将近 50 人，你从大家的 patch 来看，是否感觉各个公司的侧重点有些不同呢？目前项目是如何管理的？

**李浩源：**参与 Tachyon 项目的开发者实际上大于 Github 上的统计数据（其中包括不少来自国内的开发者），并且还有一些比较大的功能在一些公司和科研机构已经内部测试过，正在提交的过程中。

因为每个公司机构的战略不一样，所以的确侧重点是不一样。比如说，网络硬件公司就会对 Tachyon 的网络层更加有兴趣，系统集成公司对 Tachyon 的兼容性更有兴趣。

从项目管理上来讲，Tachyon 有一个开放的社区，很欢迎更多的开发者加入。目前的流程是比较小规模的改动，开发者会自己提交一个 Patch。而对于比较大的功能，开发者和我会紧密的合作，确保功能和项目的总体方向切合，并有一个初步设计意向。而后社区会对设计和代码提出建议，经过一些轮的改动之后，我会尽快把代码融合(merge)到项目的公共代码库中。

随着项目的发展，越来越多的公司和机构已经或者开始投入全职员工对项目进行开发，其中包括很多在其领域领先的上市公司，从长期来讲，我们

会像 Apache Mesos 以及 Apache Spark 一样，进入 Apache Software Foundation。这里欢迎更多的开发者加入。

**InfoQ：**你以前说过，在学校做东西需要有学术价值，但企业更注重将东西产品化、商业化。现在 **Tachyon** 毫无疑问是越来越商业化了，你现在觉得出论文和商业化有很大的差别吗？

**李浩源：**差别的确是有的，但是不一定‘很大’。我的经验还很浅薄，但是在做的领域，学术是需要有前瞻性（比如需要预测未来的趋势，根据趋势来指引方向），可以作为工业化的基础，但是工业化在此基础上还需要做大量的工作。这两件事情在 Tachyon 这个项目上目前来看是相辅相成的。比如我们 Tachyon 第一篇论文只是项目中的一个功能、或者说是一个点，从这个点出发，我们做大量的工作来工业化。这些工作的直观成果是，今年以来，使用 Tachyon 的公司数量在指数增长，根据三个月前的调查，已经有至少 50-100 家公司已经在使用 Tachyon。这些工作的间接成果是，通过更多公司的使用，使得学术机构看到很多不同的应用案例，从而进行相对应的研究。目前我们实验室内部，以及和其他高校合作，就有不少基于目前 Tachyon 项目的相关科研工作，明年应该会看到更多的成果。因为这些科研距离实际案例很近，所以会相对更加容易的增强 Tachyon 在产业界的应用和价值。

李浩源将在 10 月 16 日的 Strata 大会上就 Tachyon 项目进行分享。

查看原文：[Pivotal 和 EMC 谈下一代数据湖技术：Tachyon + Spark 将极为重要](#)

# PaaS，不是银弹

作者 王利俊

## 摘要

首先这篇文章并非攻击 PaaS，也不是否定 PaaS 的价值。相反，笔者是想通过本文对 PaaS 有一个更加明确的界定，它是什么，能处理哪些问题，不能解决哪些问题。这样可以对所有正在探索 PaaS 或准备上 PaaS 的企业，能有一个参考。

本文作为笔者过去十年的工作总结，对 PaaS 的实践和思考。笔者曾在新浪供职九年时间，参与并负责研发内部动态平台(私有 PaaS)的建设并在后来领导了整个 SAE(公有 PaaS)项目的发展，因为有了动态平台的实践经验，也才有了后来 SAE 的诞生，两者有因果联系。

## 动态平台（Dynamic Pool）

这个名词是和静态池相对的。因为新浪在很早就为新闻业务构建了一个静态池（目前仍在沿用）。

### 起源

动态平台的立项在 2004 年，当时 CTO 李嵩波先生负责新浪技术工作，他对这个项目非常支持。童剑当时是这个项目的带头人。

当时的动态平台解决的问题：

- 资源共用：避免一个应用一堆机器。
- 开发有规范：不能按照每个开发人员的好恶。
- 统一的运维管理：开发人员不管理机器，只负责代码编写和数据库设计。

### 发展

动态平台的发展初期，得益于公司领导的支持和成本管理的加强。这使得新项目申请设备预算变得困难，进而促进了动态平台的快速发展。

发展过程中遇到的主要难题：

- 资源争抢冲突问题。
- 故障排查难度大。
- 数据库管理面临挑战。
- 开发和运维的协作配合。

这些难题在动态平台不同的发展时期，表现程度也不尽相同。在不同时期，都有相应的流程或技术来解决这些问题。

## 壮大

2009年，微博技术负责人决定使用动态平台，这使得动态平台的承载规模在随后几年都呈现了井喷式的高速发展。并使得动态平台的适应能力更强。

动态平台快速发展壮大的根本原因在于公司领导支持和严格的成本管理，削减业务部门IT预算。这一点可供想搞私有IaaS或私有PaaS的企业参考，如果你们的预算很多，那么搞私有云，十有八九是要失败的！很明显，业务部门的IT预算足够，是没有能动性去使用私有云的。

如果要问全球业务规模最大的PaaS是哪一家，那一定是新浪研发的动态平台！

## SAE

2008年Google GAE发布。笔者当时正负责动态平台的日常管理。当时的GAE我看到后非常惊艳，开发人员可以自助管理自己的应用，写好代码提交后就直接运行。而当时动态平台还是工单时代，开发人员需要提交应用申请，我们在后台进行手工配置后开通。当时就有一股冲动，想要搞一个类似的产品。这在2009年成为现实。2009年11月SAE如愿上线，并很快发布了alpha1、alpha2、beta等多个版本。随着微博的蓬勃发展，2011年微博开放平台应用的蓬勃发展，有力地带动了SAE的飞速发展，当时的微博投票、粉丝汇、微博数据分析、聊天工具等大量第三方的应用快速地在SAE上诞生，并且日访问量都可以轻松过千万。

## 挑战

SAE的技术架构，有很多动态平台的影子。其运营维护也得益于过去多年成熟的经验。但外部用户和内部用户的差别，对SAE的影响很大，特别是后来IaaS和云主机在国内快速发展，SAE发展速度放缓。

- 外部业务的差异性大，内部业务相对要整齐。
- 外部客户的协作难度更高 外部客户数量庞大，在服务支持上只能侧重于重要的客户。

- 敏感应用监管难度高。
- DDoS 攻击每日不绝 这是所有做公有云的人都面临的痛苦。
- 恶意应用多，比如恶意的淘宝客。

## 用户使用 SAE 的理由

毫无疑问，SAE 是国内最早的 PaaS 平台，也是目前国内最成熟、用户规模最大的 PaaS 平台。即使是在目前云计算用户争抢越来越激烈的今天，每天仍然有大量用户注册使用 SAE 平台。之所以有用户愿意使用 SAE，核心的原因：

- 快速获取 app 运行环境 虽然说用户搭建一套 Lamp 或 Tomcat 环境并不复杂，但如果不是很熟练，看文档去做，几个小时还是需要的。
- 免运维 这个是最关键和核心的。使用 SAE 后，你完全不需要关心运维了，只要负责写代码，这对很多开发人员来说，很有吸引力。
- 便宜 SAE 的实现方式，决定了它的密度最高，目前没有其他模式可以相比。这也是为什么使用 SAE 会很便宜的原因。这对很多个人开发者而言很有吸引力。

## PaaS 解密

### 定义

维基百科的解释： In this model, the consumer creates the software using tools and/or libraries from the provider. The consumer also controls software deployment and configuration settings. The provider provides the networks, servers, storage, and other services that are required to host the consumer's application.

上面的定义，应该是对多家 PaaS 供应商的产品的一个总结。包括 GAE、Heroku、CloudFoundry、OpenShift、SAE 等。翻译为中文的意思就是：使用者只要提交应用代码，其余所有事由 PaaS 供应商搞定。

这是多么美好的愿景！我想这也是所有开发者的梦想，只关心代码，其他的都不用管，服务还都能运行得很好，99.99% 的可用性，不用担心半夜出故障还得爬起来，不用担心数据库忘记了备份导致数据丢失，不用担心访问量突然倍增，服务抗不住，不用担心网络故障来回切换服务。世界变得好有秩序。

上面描述的愿景，令人十分向往。如果真的有这样的 PaaS 存在，如果 GAE 真的做到了这些，为何云计算的领导者是 AWS，不是 GAE？

我不禁怀疑，这样的万能的包治百病的 PaaS 真的存在吗？不论是作为先行者的 GAE、Heroku、SAE，还是后来的 CloudFoudry、OpenShift，还是现在的基于 Docker 的 Flynn、Deis。

如果让我现在给一个 PaaS 的定义，我会这样写：**PaaS 是一套开发、运维的规范和流程，可以通过一些辅助工具将规范、流程沉淀下来。**但同时业务和技术总是处于不断变化的时代。流程和规范也需要适应变化。没有一套流程规范能让你用一辈子，也没有什么工具可以帮助你一劳永逸地解决所有问题。新浪动态平台已经有不到 10 年的历史，一直都处于不断的演进、变化、调整中，之所以需要不断演进变化，因为技术在变化、业务在变化、组织在变化，不要期待不变，那是不可能也是做不到的。

## PaaS 能解决什么问题

要谈 PaaS 能够解决哪些问题，取决于 PaaS 提供哪些能力，一般而言，目前的 PaaS 提供：

- 代码部署能力。
- 代码运行时环境，如 Java、PHP、Ruby 等。
- 各种应用运行所需的服务 典型的是数据库。

从上面的功能看，PaaS 主要解决的问题是应用的部署以及执行。

## PaaS 不能解决什么问题

- PaaS 不能做到全自动、无故障的运维管理。
- PaaS 也不能代替你实施开发和运维流程的梳理，而这个我认为对企业才是最核心的，是一个开发和运维观念的变化，光有工具是不行的。
- PaaS 需要的运营维护工具，仍然是需要你自己开发或者购买的。PaaS 无法提供全套的管理工具。
- PaaS 提供的服务仍然是有限的。比如你需要 LBS 服务，或者消息推送服务，可能某个 PaaS 提供，但另外的就没有。没有全能厂商可以提供所有服务，如果他提供了，也一定是个花架子。

看到上面几点，大家是不是觉得 PaaS 没什么用？其实不是，PaaS 只是个工具，你需要首先变革你的理念，或者你不使用 CloudFoundry 这么复杂的系统，但如果你已经将你的开发和运维流程规范做得很到位，那么确实是不需要 PaaS 的，或者你在实施你的流程时，就已经自觉或不自觉地使用了某些工具，你可以非常快速地部署软件、实施监控、有条理地进行备份，那么你确实无需再去引入一个 PaaS 平台了。

PaaS 最终应该是解决方案，适应客户需求的解决方案，而且是需要随着业务需求的变化可以不断演变。而不是客户削足适履去适应 PaaS 这个工具。那样的话，PaaS 之路必定是多灾多难。

## NiceScale

离开老东家新浪后，当时我立志做一个灵活性很强的 PaaS，可以支持任意的软件栈，能够帮用户管理维护好他的所有软件栈。这个项目设定的目标比 CloudFoundry 要大，当然我们在 PaaS 运营上的经验足够。但是 Docker 发展如火如荼后，一个通用的 PaaS 意义还有多大？而且要解决 PaaS 的运管方面的需求，其复杂度也很高。但最关键还是，用户真的需要这么复杂的工具吗？

我重读 Unix 经典著作，思考前辈们是如何处理这样复杂的工程的。我们承认，服务运行的管理确实非常复杂，但是如果使用了复杂的工具去管理，那么也只能带来更高的复杂度。解决复杂的问题，只有简单，任何复杂的事情，都是可以分解为简单。

从简单入手，于是有了新的 NiceScale。但 NiceScale 的目标没有变，降低用户使用云计算的复杂度一直是我们的追求，是我们矢志不渝的目标！

这个新的产品，前期只解决一个小问题，帮助你非常容易地管理多个服务器。通过批量在多个机器上执行脚本，并将行为记录下来。功能虽少，但是相信你使用过后，会体验到它的强大与方便。

原来服务器管理也可以不再枯燥，变得有趣、很酷！

初心未变，但我们选择了另外一条路，简单的路。

Keep it simple, stupid ...

## 作者简介

[@IT人](#)，曾负责新浪研发私有 PaaS（动态平台）和公有 PaaS(SAE)。混合云管理平台 [NiceScale.com](#) 创始人。

查看原文：[PaaS，不是银弹](#)

# Spark 的现状与未来发展

作者 张逸

## Spark 的发展

对于一个具有相当技术门槛与复杂度的平台，Spark 从诞生到正式版本的成熟，经历的时间如此之短，让人感到惊诧。2009 年，Spark 诞生于伯克利大学 AMPLab，最开始属于伯克利大学的研究性项目。它于 2010 年正式开源，并于 2013 年成为了 Apache 基金项目，并于 2014 年成为 Apache 基金的顶级项目，整个过程不到五年时间。

由于 Spark 出自伯克利大学，使其在整个发展过程中都烙上了学术研究的标记，对于一个在数据科学领域的平台而言，这也是题中应有之义，它甚至决定了 Spark 的发展动力。Spark 的核心 RDD（resilient distributed datasets），以及流处理，SQL 智能分析，机器学习等功能，都脱胎于学术研究论文，如下所示：

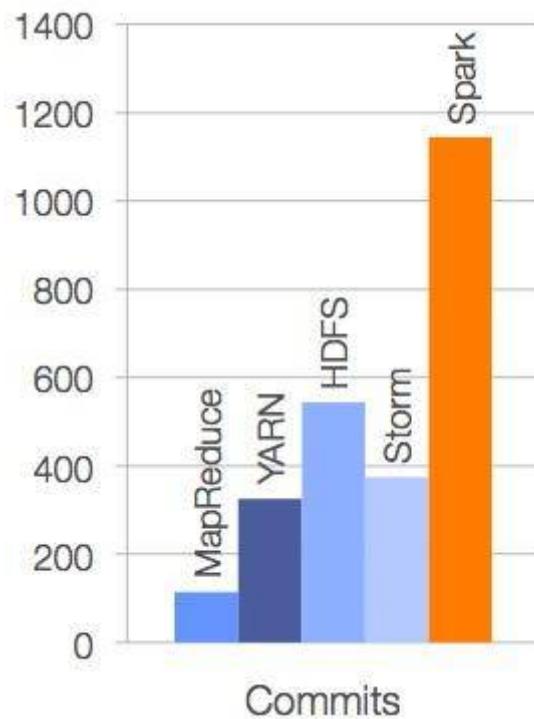
- Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica. SOSP 2013. November 2013.
- Shark: SQL and Rich Analytics at Scale. Reynold Xin, Joshua Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica. SIGMOD 2013. June 2013.
- Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. HotCloud 2012. June 2012.
- Shark: Fast Data Analysis Using Coarse-grained Distributed Memory (demo). Cliff Engle, Antonio Luper, Reynold Xin, Matei Zaharia, Haoyuan Li, Scott Shenker, Ion Stoica. SIGMOD 2012. May 2012. Best Demo Award.
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI 2012. April 2012. Best Paper Award and Honorable Mention for Community Award.

- Spark: Cluster Computing with Working Sets. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. HotCloud 2010. June 2010.

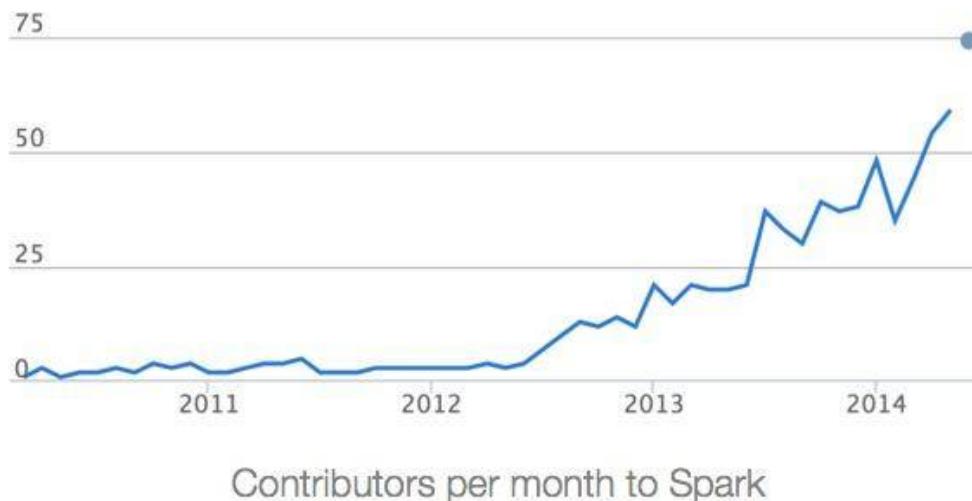
在大数据领域，只有深挖数据科学领域，走在学术前沿，才能在底层算法和模型方面走在前面，从而占据领先地位。Spark 的这种学术基因，使得它从一开始就在大数据领域建立了一定优势。无论是性能，还是方案的统一性，对比传统的 Hadoop，优势都非常明显。Spark 提供的基于 RDD 的一体化解决方案，将 MapReduce、Streaming、SQL、Machine Learning、Graph Processing 等模型统一到一个平台下，并以一致的 API 公开，并提供相同的部署方案，使得 Spark 的工程应用领域变得更加广泛。

## Spark 的代码活跃度

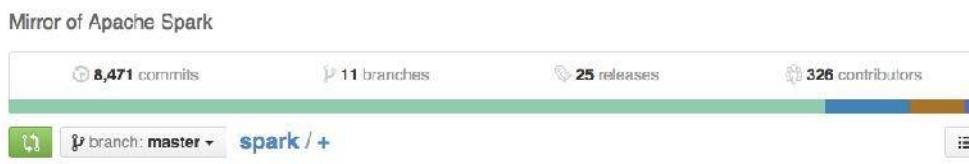
从 Spark 的版本演化看，足以说明这个平台旺盛的生命力以及社区的活跃度。尤其在 2013 年来，Spark 进入了一个高速发展期，代码库提交与社区活跃度都有显著增长。以活跃度论，Spark 在所有 Aparche 基金会开源项目中，位列前三。相较于其他大数据平台或框架而言，Spark 的代码库最为活跃，如下图所示：



从 2013 年 6 月到 2014 年 6 月，参与贡献的开发人员从原来的 68 位增长到 255 位，参与贡献的公司也从 17 家上升到 50 家。在这 50 家公司中，有来自中国的阿里、百度、网易、腾讯、搜狐等公司。当然，代码库的代码行也从原来的 63,000 行增加到 175,000 行。下图为截止 2014 年 Spark 代码贡献者每个月的增长曲线：



下图则显示了自从 Spark 将其代码部署到 Github 之后的提交数据，一共有 8471 次提交，11 个分支，25 次发布，326 位代码贡献者。



目前的 Spark 版本为 1.1.0。在该版本的代码贡献者列表中，出现了数十位国内程序员的身影。这些贡献者的多数工作主要集中在 Bug Fix 上，甚至包括 Example 的 Bug Fix。由于 1.1.0 版本极大地增强了 Spark SQL 和 MLib 的功能，因此有部分贡献都集中在 SQL 和 MLib 的特性实现上。下图是 Spark Master 分支上最近发生的仍然处于 Open 状态的 Pull Request：

281 Open ✓ 2,588 Closed		Author	Labels	Milestones	Assignee	Sort
[SQL]	Correct a variable name in JavaApplySchemaSuite.applySchemaToJSON	#2869 opened 31 minutes ago by yhuai				1
[SPARK-3161][MLLIB]	Adding a node Id caching mechanism for training deci...	#2868 opened 38 minutes ago by codedefft				1
[SPARK-4016]	Allow user to show/hide UI metrics.	#2867 opened an hour ago by kayousterhout				1
[SPARK-4019]	Fix MapStatus compression bug that could lead to empty results	#2866 opened an hour ago by JoethRosen				1
[SPARK-4020]	Do not rely on timeouts to remove failed block managers	#2865 opened 3 hours ago by andrewor14				3
[SPARK-4012]	call tryOrExit instead of logUncaughtExceptions in ContextCleaner	#2864 opened 4 hours ago by CodingCat				6
[SPARK-4013]	Do not create multiple actor systems on each executor	#2863 opened 5 hours ago by andrewor14				8
[SQL]	redundant methods for broadcast	#2862 opened 5 hours ago by sowt				10
[SQL]	replace awaitTransformation with awaitTermination in scaladoc/javadoc	#2861 opened 7 hours ago by holderkn				3

可以看出，由于 Spark 仍然比较年轻，当运用到生产上时，可能发现一些小缺陷。而在代码整洁度方面，也随时在对代码进行着重构。例如，淘宝技术部在 2013 年就开始尝试将 Spark on Yarn 应用到生产环境上。他们在执行数据分析作业过程中，先后发现了 DAGScheduler 的内存泄露，不匹配的作业结束状态等缺陷，从而为 Spark 库贡献了几个比较重要的 Pull Request。具体内容可以查看淘宝技术部的博客文章：“Spark on Yarn：几个关键 Pull Request (<http://rdc.taobao.org/?p=525>)”。

## Spark 的社区活动

Spark 非常重视社区活动，组织也极为规范，定期或不定期地举行与 Spark 相关的会议。会议分为两种，一种为 Spark Summit，影响力巨大，可谓全球 Spark 顶尖技术人员的峰会。目前，已于 2013 年和 2014 年在 San Francisco 连续召开了两届 Summit 大会。2015 年，Spark Summit 将分别在 New York 与 San Francisco 召开，其官方网站为：<http://spark-summit.org/>。

在 2014 年的 Spark Summit 大会上，我们看到除了伯克利大学以及 Databricks 公司自身外，演讲者都来自最早开始运用和尝试 Spark 进行大数据分析的公司，包括最近非常火的音乐网站 Spotify，全球最大专注金融交易的 Sharethrough，专业大数据平台 MapR、Cloudera，云计算的领先者 Amazon，以及全球超大型企业 IBM、Intel、SAP 等。

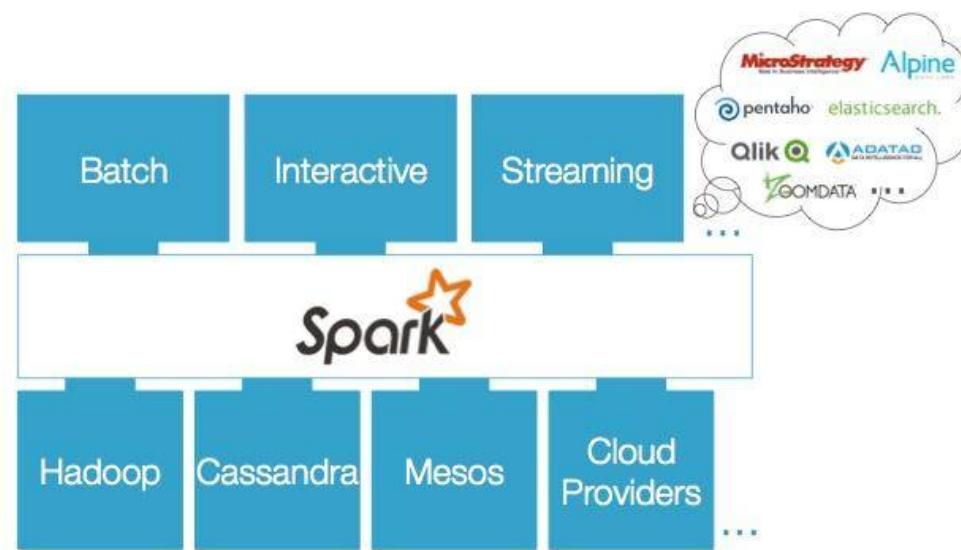
除了影响力巨大的 Spark Summit 之外，Spark 社区还不定期地在全球各地召开小型的 Meetup 活动。Spark Meetup Group 已经遍布北美、欧洲、亚洲和大洋洲。在中国，北京 Spark Meetup 已经召开了两次，并将于今年 10 月 26 日召开第三次 Meetup。届时将有来自 Intel 中国研究院、淘宝、TalkingData、微软亚洲研究院、Databricks 的工程师进行分享。下图为 Spark Meetup Groups 在全球的分布图：

**Spark Meetup Groups**



## Spark 的现在和未来

Spark 的特色在于它首先为大数据应用提供了一个统一的平台。从数据处理层面看，模型可以分为批处理、交互式、流处理等多种方式；而从大数据平台而言，已有成熟的 Hadoop、Cassandra、Mesos 以及其他云的供应商。Spark 整合了主要的数据处理模型，并能够很好地与现在主流的大数据平台集成。下图展现了 Spark 的这一特色：

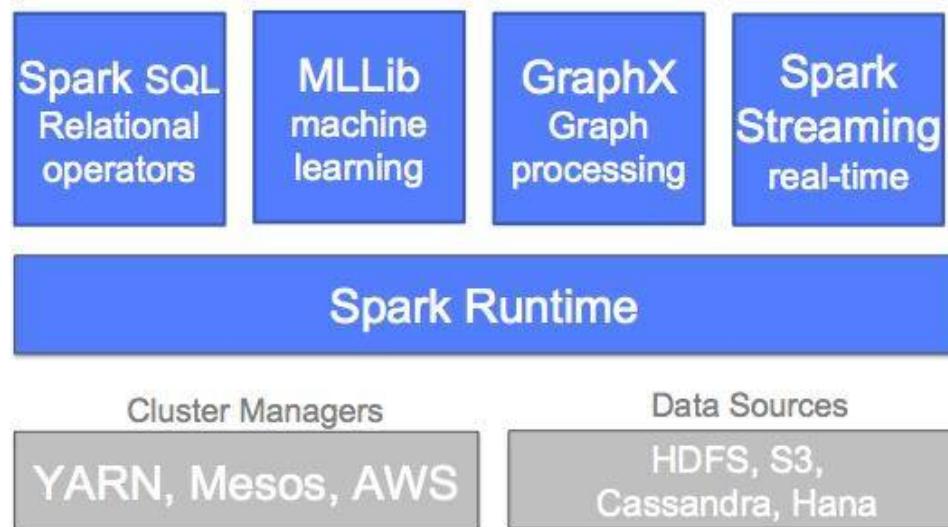


这样的一种统一平台带来的优势非常明显。对于开发者而言，只需要学习一个平台，降低了学习曲线。对于用户而言，可以很方便地将 Spark 应用运行在 Hadoop、Mesos 等平台上面，满足了良好的可迁移性。统一的数据处理方式，也可以简化开发模型，降低平台的维护难度。

Spark 为大数据提供了通用算法的标准库，这些算法包括 MapReduce、SQL、Streaming、Machine Learning 与 Graph Processing。同时，它还提供了对 Scala、Python、Java（支持 Java 8）和 R 语言的支持：



在最新发布的 1.1.0 版本中，对 Spark SQL 和 Machine Learning 库提供了增强。Spark SQL 能够更加有效地在 Spark 中加载和查询结构型数据，同时还支持对 JSON 数据的操作，并提供了更加友好的 Spark API。在 Machine Learning 方面，已经包含了超过 15 种算法，包括决策树、SVD、PCA，L-BFGS 等。下图展现了 Spark 当前的技术栈：



在 2014 年的 Spark Summit 上，来自 Databricks 公司的 Patrick Wendell 展望了 Spark 的未来。他在演讲中提到了 Spark 的目标，包括：

- Empower data scientists and engineers.
- Expressive, clean APIs.
- Unified runtime across many environments.
- Powerful standard libraries.

在演讲中，他提到在 Spark 最近的版本中，最重要的核心组件为 Spark SQL。接下来的几次发布，除了在性能上更加优化（包括代码生成和快速的 Join 操作）外，还要提供对 SQL 语句的扩展和更好的集成（利用 SchemaRDD 与 Hadoop、NoSQL 以及 RDBMS 的集成）。在将来的版本中，要为 MLLib 增加更多的算法，这些算法除了传统的统计算法外，还包括学习算法，并提供与 R 语言更好的集成，从而能够为数据科学家提供更好的选择，根据场景来选择 Spark 和 R。

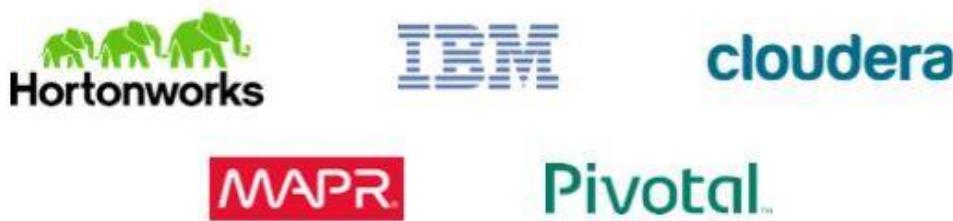
Spark 的发展会结合硬件的发展趋势。首先，内存会变得越来越便宜，256GB 内存以上的机器会变得越来越常见，而对于硬盘，则 SSD 硬盘也将慢慢成为服务器的标配。由于 Spark 是基于内存的大数据处理平台，因而在处理过程中，会因为数据存储在硬盘中，而导致性能瓶颈。随着机器内存容量的逐步增加，类似 HDFS 这种存储在磁盘中的分布式文件系统将慢慢被共享内存的分布式存储系统所替代，诸如同样来自伯克利大学的 AMPLab 实验室的 Tachyon 就提供了远超 HDFS 的性能表现。因此，未来的

Spark会在内部的存储接口上发生较大的变化，能够更好地支持SSD、以及诸如Tachyon之类的共享内存系统。事实上，在Spark的最近版本里，已经开始支持Tachyon了。

根据Spark的路线图，Databricks会在近三个月陆续发布1.2.0和1.3.0版本。其中，1.2.0版本会对存储方面的API进行重构，在1.3.0之上的版本，则会推出结合Spark和R的SparkR。除了前面提到的SQL与MLLib之外，未来的Spark对于Streaming、GraphX都有不同程度的增强，并能够更好地支持YARN。

## Spark的应用

目前，Spark的正式版本得到了部分Hadoop主流厂商的支持，如下企业或平台发布的Hadoop版本中，都包含了Spark：



这说明业界已经认可了Spark，Spark也被许多企业尤其是互联网企业广泛应用于商业项目中。根据Spark的官方统计，目前参与Spark的贡献以及将Spark运用在商业项目的公司大约有80余家。在国内，投身Spark阵营的公司包括阿里、百度、腾讯、网易、搜狐等。在San Francisco召开的Spark Summit 2014大会上，参会的演讲嘉宾分享了在音乐推荐（Spotify）、实时审计的数据分析（Sharethrough）、流在高速率分析中的运用（Cassandra）、文本分析（IBM）、客户智能实时推荐（Graphflow）等诸多在应用层面的话题，这足以说明Spark的应用程度。

但是，整体而言，目前开始应用Spark的企业主要集中在互联网领域。制约传统企业采用Spark的因素主要包括三个方面。首先，取决于平台的成熟度。传统企业在技术选型上相对稳健，当然也可以说是保守。如果一门技术尤其是牵涉到主要平台的选择，会变得格外慎重。如果没有经过多方面的验证，并从业界获得成功经验，不会轻易选定。其次是对SQL的支持。传统企业的数据处理主要集中在关系型数据库，而且有大量的遗留系统存在。在这些遗留系统中，多数数据处理都是通过SQL甚至存储过程来完成。如果一个大数据平台不能很好地支持关系型数据库的SQL，就会导致迁移数据分析业务逻辑的成本太大。其三则是团队与技术的学习曲线。如果没有熟悉该平台以及该平台相关技术的团队成员，企业就会担心开发进度、成本以及可能的风险。

Spark 在努力解决这三个问题。随着 1.0.2 版本的发布，Spark 得到了更多商用案例的验证。Spark 虽然依旧保持年轻的活力，但已经具备堪称成熟的平台功能。至于 SQL 支持，Spark 非常。在 1.0.2 版本发布之前，就认识到基于 HIVE 的 Shark 存在的不足，从而痛下决心，决定在新版本中抛弃 Shark，而决定引入新的 SQL 模块。如今，在 Spark 1.1.0 版本中，Spark SQL 的支持已经相对完善，足以支持企业应用中对 SQL 迁移的需求。关于 Spark 的学习曲线，主要的学习内容还是在于对 RDD 的理解。由于 Spark 为多种算法提供了统一的编程模型、部署模式，搭建了一个大数据的一体化方案，倘若企业的大数据分析需要应对多种场景，那么，Spark 这样的架构反而使得它的学习曲线更低，同时还能降低部署成本。Spark 可以很好地与 Hadoop、Cassandra 等平台集成，同时也能部署到 YARN 上。如果企业已经具备大数据分析的能力，原有掌握的经验仍旧可以用到 Spark 上。虽然 Spark 是用 Scala 编写，官方也更建议用户调用 Scala 的 API，但它同时也提供了 Java 和 Python 的接口，非常体贴地满足了 Java 企业用户或非 JVM 用户。如果抱怨 Java 的冗赘，则 Spark 新版本对 Java 8 的支持让 Java API 变得与 Scala API 同样的简洁而强大，例如经典的字数统计算法在 Java 8 中的实现：

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

显然，随着 Spark 的逐渐成熟，并在活跃社区的推动下，它所提供的强大功能一定能得到更多技术团队和企业的青睐。相信在不远的将来会有更多传统企业开始尝试使用 Spark。

查看原文：[Spark 的现状与未来发展](#)

# 微观 SOA：服务设计原则及其实践方式（上篇）

作者 沈理

大量互联网公司都在拥抱 SOA 和服务化，但业界对 SOA 的很多讨论都比较偏向高大上。本文试图从稍微不同的角度，以相对接地气的方式来讨论 SOA，集中讨论 SOA 在微观实践层面中的缘起、本质和具体操作方式，另外也用相当篇幅介绍了当今互联网行业中各种流行的远程调用技术等等，比较适合从事实际工作的架构师和程序员来阅读。

为了方便阅读，本话题将分为两篇展现。本文是上篇，着眼于微观 SOA 的定义，并简单分析其核心原则。

## 亚马逊 CEO 杰夫·贝佐斯：鲜为人知的 SOA 大师

由于 SOA 有相当的难度和门槛，不妨先从一个小故事说起，从中可以管窥一点 SOA 的大意和作用。

按照亚马逊前著名员工 Steve Yegge 著名的“酒后吐槽”，2002 年左右，CEO 贝佐斯就在亚马逊强制推行了以下六个原则（摘自[酷壳](#)）：

1. 所有团队的程序模块都要以通过 Service Interface 方式将其数据与功能开放出来。
2. 团队间的程序模块的信息通信，都要通过这些接口。
3. 除此之外没有其它的通信方式。其他形式一概不允许：不能使用直接链接程序、不能直接读取其他团队的数据库、不能使用共享内存模式、不能使用别人模块的后门等，唯一允许的通信方式只能是能过调用 Service Interface。
4. 任何技术都可以使用。比如：HTTP、Corba、Pubsub、自定义的网络协议、等等，都可以，贝佐斯不管这些。
5. 所有的 Service Interface，毫无例外，都必须从骨子里到表面上设计成能对外界开放的。也就是说，团队必须做好规划与设计，以便未来把接口开放给全世界的程序员，没有任何例外。
6. 不这样的做的人会被炒鱿鱼。

据说，亚马逊网站展示一个产品明细的页面，可能要调用 200-300 个 Service，以便生成高度个性化的内容。

Steve 还提到：

Amazon 已经把文化转变成了“一切以 Service 第一”为系统架构的公司，今天，这已经成为他们进行所有设计时的基础，包括那些绝不会被外界所知的仅在内部使用的功能。

那时，如果没有被解雇的的恐惧他们一定不会去做。我是说，他们今天仍然怕被解雇，因为这基本上是那儿每天的生活，为那恐怖的海盗头子贝佐斯工作。不过，他们这么做的确是因为他们已经相信 Service 这就是正确的方向。他们对于 SOA 的优点和缺点没有疑问，某些缺点还很大，也不疑问。但总的来说，这是正确的，因为，SOA 驱动出来的设计会产生出平台（Platform）。

今天，我们都知道亚马逊从世界上最大图书卖场进化为了世界上最成功的云平台……

贝佐斯的六原则展示出高度的远见和超强的信念，即使放到十几年后的今天，依然觉得振聋发聩……想起一句老话：“不谋万世者，不足以谋一时；不谋全局者，不足以谋一隅。”

当然，像贝佐斯这种将神性与魔性集于一身的专横人物，既可能创造划时代的进步，也可能制造前所未有的灾难。

## SOA 漫谈：宏观与微观

SOA 即面向服务架构，是一个特别大的话题。

为了方便讨论，我在此先草率的将 SOA 分为两个层面（大概模仿宏观和微观经济学，但这里的划分没有绝对界限）。

- 宏观 SOA：**面向高层次的部门级别、公司级别甚至行业级别；涉及商业、管理、技术等方面综合的、全局的考虑；架构体系上包括服务治理（governance，如服务注册，服务监控），服务编排（orchestration，如 BPM，ESB），服务协同（choreography，更多面向跨企业集成）等等。我认为 SOA 本身最主要是面向宏观层面的架构，其带来益处也最能在宏观高层次上体现出来，同时大部分 SOA 的业界讨论也集中在这方面。
- 微观 SOA：**面向有限的、局部的团队和个人；涉及独立的、具体的服务在业务、架构、开发上的考虑。

很多业界专家都认为 SOA 概念过于抽象，不接地气，我认为主要是宏观 SOA 涉及面太广，经常需要做通盘考虑，而其中很多方面距离一般人又比较远。而在微观层面的 SOA 更容易达到涛哥过去提出的“三贴近”：贴近实际、贴近生活、贴近群众。

同时，宏观 SOA 要取得成功，通常的前提也是 SOA 在微观层面的落地与落实，正如宏观经济学一般要有坚实的微观基础（比如大名鼎鼎的凯恩斯主义曾广受诟病的一点就是缺乏微观基础）。

因此，我们着眼于 SOA 落地的目的，着重来分析微观 SOA，也算是对业界主流探讨的一个小小的补充。

## SOA 定义

按照英文维基百科定义：SOA 是一种“软件”和“软件架构”的设计模式（或者叫设计原则）。它是基于相互独立的软件片段要将自身的功能通过“服务”提供给其他应用。

什么是“服务”？按照 OASIS 的定义：Service 是一种按照既定“接口”来访问一个或多个软件功能的机制（另外这种访问要符合“服务描述”中策略和限制）。

## Service 示例（代码通常以 java 示例）

```
public interface Echo {  
    String echo(String text);  
}  
  
public class EchoImpl implements Echo {  
    public String echo(String text) {  
        return text;  
    }  
}
```

可能每个开发人员每天都在写类似的面向对象的 Service，难道这就是在实施 SOA 吗？

## SOA 设计原则

既然 SOA 是设计原则（模式），那么它包含哪些内容呢？事实上，这方面并没有最标准的答案，多数是遵从著名 SOA 专家 Thomas Erl 的归纳：

标准化的服务契约 Standardized service contract 服务的松耦合 Service loose coupling 服务的抽象 Service abstraction 服务的可重用性 Service reusability 服

务的自治性 Service autonomy 服务的无状态性 Service statelessness 服务的可发现性 Service discoverability 服务的可组合性 Service compositability.....

这些原则总的来说要达到的目的是：提高软件的重用性，减少开发和维护的成本，最终增加一个公司业务的敏捷度。

但是，业界著名专家如 Don Box, David Orchard 等人对 SOA 又有各自不同的总结和侧重。

SOA 不但没有绝对统一的原则，而且很多原则本身的内容也具备相当模糊性和宽泛性：例如，所谓松耦合原则需要松散到什么程度才算是符合标准的呢？这就好比一个人要帅到什么程度才算是帅哥呢？一栋楼要高到多少米才算是高楼呢？可能不同人心中都有自己的一杆秤.....部分由于这些理论上的不确定因素，不同的人理解或者实施的 SOA 事实上也可能有比较大的差别。

## 浅析松耦合原则

SOA 原则比较多，真正的理解往往需要逐步的积累和体会，所以在此不详细展开。这里仅以服务的松耦合为例，从不同维度来简单剖析一下这个原则，以说明 SOA 原则内涵的丰富性。

- **实现的松耦合：**这是最基本的松耦合，即服务消费端不需要依赖服务契约的某个特定实现，这样服务提供端的内部变更就不会影响到消费端，而且消费端未来还可以自由切换到该契约的其他提供方。
- **时间的松耦合：**典型就是异步消息队列系统，由于有中介者（broker），所以生产者和消费者不必在同一时间都保持可用性以及相同的吞吐量，而且生产者也不需要马上等到回复。
- **位置的松耦合：**典型就是服务注册中心和企业服务总线（ESB），消费端完全不需要直接知道提供端的具体位置，而都通过注册中心来查找或者服务总线来路由。
- **版本的松耦合：**消费端不需要依赖服务契约的某个特定版本来工作，这就要求服务的契约在升级时要尽可能的提供向下兼容性。

## SOA 与传统软件设计

我们可以认为：**SOA 模块化开发 + 分布式计算**

将两者传统上的最佳实践结合在一起，基本上可以推导出 SOA 的多数设计原则。SOA 从软件设计（暂不考虑业务架构之类）上来讲，自身的新东西其实不算很多。

## SOA 原则的应用

基于 SOA 的原则，也许我们很难说什么应用是绝对符合 SOA 的，但是却能剔除明显不符合 SOA 的应用。

用上述标准化契约，松耦合和可重用这几个原则来尝试分析一下上面 Echo 示例：

- Echo 的服务契约是用 Java 接口定义，而不是一种与平台和语言无关的标准化协议，如 WSDL，CORBA IDL。当然可以抬杠，Java 也是行业标准，甚至全国牙防组一致认定的东西也是行业标准。
- Java 接口大大加重了与 Service 客户端的耦合度，即要求客户端必须也是 Java，或者 JVM 上的动态语言（如 Groovy、Jython），等等。
- 同时，Echo 是一个 Java 的本地接口，就要求调用者最好在同一个 JVM 进程之内……
- Echo 的业务逻辑虽然简单独立，但以上技术方面的局限就导致它无法以后在其他场合被轻易重用，比如分布式环境，异构平台，等等。

因此，我们可以认为 Echo 并不太符合 SOA 的基本设计原则。

## 透明化的转向 SOA？

修改一下上面的 Echo，添加 Java EE 的@WebServices 注解（annotation）：

```
@WebServices
public class EchoImpl implements Echo {
    public String echo(String text) {
        return text;
    }
}
```

现在将 Echo 发布为 Java WebServices，并由底层框架自动生成 WSDL 来作为标准化的服务契约，这样就能与远程的各种语言和平台互操作了，较好的解决了上面提到的松耦合和可重用的问题。按照一般的理解，Echo 似乎就成为比较理想的 SOA service 了。

但是……即使这个极端简化的例子，也会引出不少很关键的问题，它们决定 SOA 设计开发的某些难度：

- 将一个普通的 Java 对象通过添加注解“透明的”变成 WebServices 就完成了从面向对象到面向服务的跨越？
- 通过 Java 接口生成 WSDL 服务契约是好的方式吗？
- WebServices 是最合适远程访问技术吗？

## 面向对象和面向服务的对比

面向对象（OO）和面向服务（SO）在基础理念上有大量共通之处，比如都尽可能追求抽象、封装和低耦合。

但 SO 相对于 OO，又有非常不同的典型应用场景，比如：

- 多数 OO 接口（interface）都只被有限的人使用（比如团队和部门内），而 SO 接口（或者叫契约）一般来说都不应该对使用者的范围作出太多的限定和假设（可以是不同部门，不同企业，不同国家）。还记得贝佐斯原则吗？“团队必须做好规划与设计，以便未来把接口开放给全世界的程序员，没有任何例外”。
- 多数 OO 接口都只在进程内被访问，而 SO 接口通常都是被远程调用。

简单讲，就是 SO 接口使用范围比一般 OO 接口可能广泛得多。我们用网站打个比方：一个大型网站的 web 界面就是它整个系统入口点和边界，可能要面对全世界的访问者（所以经常会做国际化之类的工作），而系统内部传统的 OO 接口和程序则被隐藏在 web 界面之后，只被内部较小范围使用。而理想的 SO 接口和 web 界面一样，也是变成系统入口和边界，可能要对全世界开发者开放，因此 SO 在设计开发之中与 OO 相比其实会有很多不同。

## 小结

在前述比较抽象的 SOA 大原则的基础上，我们可尝试推导一些较细化和可操作的原则，在具体实践中体现 SO 的独特之处。请关注本系列文章的下篇！

查看原文：[微观 SOA：服务设计原则及其实践方式（上篇）](#)

# 微观 SOA：服务设计原则及其实践方式（下篇）

作者 沈理

在[上一篇文章](#)中，我说到 SOA 是一个特别大的话题，不但没有绝对统一的原则，而且很多原则本身的内容也具备相当模糊性和宽泛性。虽然我们可以说 SOA 模块化开发 + 分布式计算，但由于其原则的模糊性，我们仍然很难说什么应用是绝对符合 SOA 的，只能识别出哪些是不符合 SOA 的。

本篇将对 8 种可操作的服务设计原则进行细化的分析，作为 SOA 实践的参考。

## 服务设计原则 1：优化远程调用

这里的远程调用特指 RPC (Remote Procedure Call)。当然更面向对象的说法应该是远程方法调用或者远程服务调用，等等。

由于 SO 接口通常要被远程访问，而网络传输，对象序列化/反序列化等开销都远远超过本地 Object 访问几个数量级，所以要加快系统的响应速度、减少带宽占用和提高吞吐量，选择高性能的远程调用方式经常是很重要的。

但是远程调用方式往往又要受限于具体的业务和部署环境，比如内网、外网、同构平台、异构平台等等。有时还要考虑它对诸如分布式事务，消息级别签名/加密，可靠异步传输等方面的支持程度（这些方面通常被称为 SLA: service level agreement），甚至还包括开发者的熟悉和接受程度，等等。

因此，远程调用方式往往需要根据具体情况做出选择和权衡。

以 Java 远程 Service 为例分析不同场景下，传输方式的某些可能较好选择：

- 内网 + 同框架 Java 客户端 + 大并发：多路复用的 TCP 长连接 + kryo (二进制序列化) (kryo 也可以用 Protostuff, FST 等代替)
- 内网 + 不同框架 Java 客户端：TCP + Kryo
- 内网 + Java 客户端 + 2PC 分布式事务：RMI/IOP (TCP + 二进制)

- 内网 + Java 客户端 + 可靠异步调用: JMS + Kryo (TCP + 二进制)
- 内网 + 不同语言客户端: thrift (TCP + 二进制序列化)
- 外网 + 不同语言客户端 + 企业级特性: HTTP + WSDL + SOAP (文本)
- 外网 + 兼顾浏览器、手机等客户端: HTTP + JSON (文本)
- 外网 + 不同语言客户端 + 高性能: HTTP + ProtocolBuffer (二进制)

简单来说，从性能上讲，tcp 协议 + 二进制序列化更适合内网应用。从兼容性、简单性上来说，http 协议 + 文本序列化更适合外网应用。当然这并不是绝对的。另外，tcp 协议在这里并不是限定远程调用协议一定只能是位于 OSI 网络模型的第四层的原始 tcp，它可以包含 tcp 之上的任何非 http 协议。

所以，回答上面提到的问题，WebServices（经典的 WSDL+SOAP+HTTP）虽然是最符合前述 SOA 设计原则的技术，但并不等同于 SOA，我认为它只是满足了 SOA 的底线，而未必是某个具体场景下的最佳选择。这正如一个十项全能选手在每个单项上是很难和单项冠军去竞争的。更理想的 SOA Service 最好能在可以支持 WebServices 的同时，支持多种远程调用方式，适应不同场景，这也是 Spring Remoting, SCA, Dubbo, Finagle 等分布式服务框架的设计原则。

### 远程调用技术解释：HTTP + JSON 适合 SOA 吗

JSON 简单易读，通用性极佳，甚至能很好支持浏览器客户端，同时也常被手机 APP 使用，大有取代 XML 之势。

但 JSON 本身缺乏像 XML 那样被广泛接受的标准 schema，而一般的 HTTP + JSON 的远程调用方式也缺乏像 Thrift, CORBA, WebServices 等等那样标准 IDL（接口定义语言），导致服务端和客户端之间不能形成强的服务契约，也就不能做比如自动代码生成。所以 HTTP + JSON 在降低了学习门槛的同时，可能显著的增加复杂应用的开发工作量和出错可能性。

例如，新浪微博提供了基于 HTTP + JSON 的 Open API，但由于业务操作比较复杂，又在 JSON 上封装实现了各种语言的客户端类库，来减少用户的工作量。

为了解决这方面的问题，业界有很多不同方案来为 HTTP + JSON 补充添加 IDL，如 RSDL、JSON-WSP、WADL、WSDL 2.0 等等，但事实上它们的接受度都不太理想。

另外值得一提的是，JSON 格式和 XML 一样有冗余，即使做 GZIP 压缩之类的优化，传输效率通常也不如很多二进制格式，同时压缩、解压还会引入额外的性能开销。

## 远程调用技术解释：Apache Thrift 多语言服务框架

Thrift 是最初来自 facebook 的一套跨语言的 service 开发框架，支持 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, JavaScript, Node.js, Smalltalk, Delphi 等几乎所有主流编程语言，具有极好的通用性。

Thrift 被 facebook, twitter 等巨头以及开源社区都广泛使用，是非常成熟的技术。

Thrift 的服务契约通过类似如下形式的 IDL 定义：

```
struct User {  
    1: i32 id,  
    2: string name,  
    3: string password  
}  
  
service UserService {  
    void store(1: User user),  
    UserProfile retrieve(1: i32 id)  
}
```

非常类似于 C 语言，易读易写，比 WSDL 简单明了得多。比用 java 之类的编程语言也更方便，有时候可以把所有相关的接口和数据结构定义放到同一个文件，发布出去的时候不用再打一个压缩包之类，甚至可以直接粘贴到文档中。

Thrift 还提供工具，可以基于 IDL 自动生成各种语言对应的服务端和客户端代码：

```
[lishen@dangdang thrift]thrift --gen java user.thrift  
[lishen@dangdang thrift]$ thrift --gen cpp user.thrift  
[lishen@dangdang thrift]$ thrift --gen php user.thrift  
[lishen@dangdang thrift]$ thrift --gen csharp user.thrift
```

我认为 thrift 是比 WebServices 更简单高效的技术，是在 SOA 中对 WebServices 最具有替代性的技术之一。

## 远程调用技术解释：多路复用的 TCP 长连接

这是一种追求极致高性能高伸缩的方式，这里只做简要介绍。

比较典型的是 twitter 的 Mux RPC 协议以及 google 的 SPDY 协议，在其中多个请求同时共用同一个长连接，即一个连接交替传输不同请求的字节块。它既避免了反复建立连接开销，也避免了连接的等待闲置从而减少了系统连接总数，同时还避免了 TCP 顺序传输中的线头阻塞（head-of-line blocking）问题。

另外，国内比较著名的开源 dubbo 框架的默认 RPC 协议，以及业界许多小型开源 RPC 框架也都是类似的思路。

采用多路复用机制后，一般就要求服务器端和客户端都支持额外的类似于会话层（即 OSI 网络模型第六层）的语义，导致它们必须要依赖于同一套 RPC 框架。

其他很多 RPC 机制都是使用 TCP 短连接。即使有些 RPC 使用了长连接，但一个连接同一时间只能发送一个请求，然后连接就处于闲置状态，来等待接收该请求的响应，待响应完毕，该连接才能被释放或者复用。

HTTP 1.1 也支持一种基于 pipeline 模式的长连接，其中多个 HTTP 请求也可共用一个连接，但它要求响应（response）也必须按照请求（request）的顺序传输返回，即 FIFO 先进先出。而在完全多路复用的连接中，哪个的响应先 ready 就可以先传输哪个，不用排队。

当然，短连接、长连接和多路复用长连接之间不存在绝对的好坏，需要取决于具体业务和技术场景，在此不详细展开了。

## 远程调用技术解释：Java 高效序列化

最近几年，各种新的 Java 高效序列化方式层出不穷，不断刷新序列化性能的上限，例如 Kryo，FST 等开源框架。它们提供了非常高效的 Java 对象的序列化和反序列化实现，相比 JDK 标准的序列化方式（即基于 Serializable 接口的标准序列化，暂不考虑用诸如 Externalizable 接口的定制序列化），在典型场景中，其序列化时间开销可能缩短 20 倍以上，生成二进制字节码的大小可能缩减 4 倍以上。

另外，这些高效 Java 序列化方式的开销也显著少于跨语言的序列化方式如 thrift 的二进制序列化，或者 JSON，等等。

## 远程调用技术解释：RMI/IOP 和分布式事务

RMI/IOP 是 Java EE 中标准的远程调用方式，IOP 是 CORBA 的协议，只有 IOP 上的 RMI 才支持两阶段提交的分布式事务，同时提供和 CORBA 的互操作。

当然，严格的两阶段提交事务并不高效，还可能严重影响系统伸缩性甚至可用性等等，一般只应用在非常关键的业务中。

## 远程调用技术解释：Google ProtocolBuffer 跨语言序列化

ProtocolBuffer 是 google 开发的跨语言的高效二进制序列化方式，其序列化性能和 thrift 比较类似。事实上 thrift 最初就是 ProtocolBuffer 的仿制品。但它和 thrift 最大的不同是他没有自带的 RPC 实现（因为 google 没有将 RPC 部分开源，但有大量第三方实现）。

由于不和 RPC 方式耦合，反而使得 ProtocolBuffer 能被方便的集成进大量已有的系统和框架中。在国内它也被百度、淘宝等广泛的应用在 Open API 中，和 HTTP 搭配作为一种高效的跨平台跨组织的集成方式。

## 服务设计原则 2：消除冗余数据

同样由于 service 的远程调用开销很高，所以在它的输入参数和返回结果中，还要尽量避免携带当前业务用例不需要的冗余的字段，来减少序列化和传输的开销。同时，去掉冗余字段也可以简化接口，避免给外部用户带来不必要的业务困惑。

比如 article service 中有个返回 article list 的方法

```
List<Article> getArticles(...)
```

如果业务需求仅仅是要列出文章的标题，那么在返回的 article 中就要避免携带它的 contents 等字段。

这里经典解决方案就是引入 OO 中常用的 Data Transfer Object (DTO)模式，专门针对特定 service 的用例来定制要传输的数据字段。这里就是添加一个 ArticleSummary 的额外数据传输对象：

```
List<ArticleSummary> getArticleSummaries(...)
```

额外的 DTO 确实是个麻烦，而一般 OO 程序通常则可直接返回自己的包含冗余的业务模型。

## 服务设计原则 3：粗粒度契约

同样由于远程调用开销高，同时 service 的外部使用者对特定业务流程的了解也比不上组织内部的人，所以 service 的契约（接口）通常需要是粗粒度的，其中的一个操作就可能对应到一个完整的业务用例或者业务流程，这样既能减少远程调用次数，同时又降低学习成本和耦合度。

而 OO 接口通常可以是非常细粒度的，提供最好的灵活性和重用性。

例如，article service 支持批量删除文章，OO 接口中可以提供：

```
deleteArticle(long id)
```

供用户自己做循环调用（暂不考虑后端 SQL 之类优化），但 SO 接口中，则最好提供：

```
deleteArticles(Set<Long> ids)
```

供客户端调用，将可能的 N 次远程调用减少为一次。

例如，下订单的用例，要有一系列操作：

```
addItem -> addTax -> calculateTotalPrice -> placeOrder
```

OO 中我们完全可以让用户自己来灵活选择，分别调用这些细粒度的可复用的方法。但在 SO 中，我们需要将他们封装到一个粗粒度的方法供用户做一次性远程调用，同时也隐藏了内部业务的很多复杂性。另外，客户端也从依赖 4 个方法变成了依赖 1 个方法，从而大大降低了程序耦合度。

顺便值得一提的是，如果上面订单用例中每个操作本身也是远程的 service（通常在内网之中），这种粗粒度封装就变成了经典的 service composition（服务组合）甚至 service orchestration（服务编排）了。这种情况下粗粒度 service 同样可能提高了性能，因为对外网客户来说，多次跨网的远程调用变成了一次跨网调用 + 多次内网调用。

对这种粗粒度 service 封装和组合，经典解决方案就是引入 OO 中常用的 Facade 模式，将原来的对象屏蔽到专门的“外观”接口之后。同时，这里也很可能要求我们引入新的 service 参数/返回值的数据结构来组合原来多个操作的对象模型，这就同样用到前述的 DTO 模式。

一个简单 Facade 示例（FooService 和 BarService 是两个假想的本地 OO service，façade 将它们的结果值组合返回）：

```
class FooBarFacadeImpl implements FooBarFacade {  
    private FooService fooService;  
    private BarService barService;  
  
    public FooBarDto getFooBar() {  
        FooBarDto fb = new FooBarDto();  
        fb.setFoo(fooService.getFoo());  
        fb.setBar(barService.getBar());  
        return fb;  
    }  
}
```

```
    fb.setBar(barService.getBar());
    return fb;
}
}
```

当然，有的时候也可以不用 facade 和 DTO，而在是 FooService 和 BarService 之外添加另一个本地 service 和 domain model，这要和具体业务场景有关。

## 服务设计原则 4：通用契约

由于 service 不假设用户的范围，所以一般要支持不同语言和平台的客户端。但各种语言和平台在功能丰富性上有很大差异，这就决定了服务契约必须取常见语言、平台以及序列化方式的最大公约数，才能保证 service 广泛兼容性。由此，服务契约中不能有某些语言才具备的高级特性，参数和返回值也必须是被广泛支持的较简单的数据类型（比如不能有对象循环引用）。

如果原有的 OO 接口不能满足以上要求，则在此我们同样需要上述的 Facade 和 DTO，将 OO 接口转换为通用的 SO 契约。

例如原有对象模型：

```
class Foo {
    private Pattern regex;
}
```

Pattern 是 Java 特有的预编译好的，可序列化的正则表达式（可提高性能），但在没有特定框架支持下，可能不好直接被其他语言识别，所以可添加 DTO：

```
class FooDto {
    private String regex;
}
```

## 服务设计原则 5：隔离变化

虽然 OO 和 SO 都追求低耦合，但 SO 由于使用者范围极广，就要求了更高程度的低耦合性。

比如前述的 article service，OO 中可以直接返回 article 对象，而这个 article 对象在 OO 程序内部可能做为核心的建模的 domain model，甚至作为 O/R mapping 等等。而在 SO 如果还直接返回这个 article，即使没有前面所说的冗余字段，复杂类型等问题，也可能让外部用户与内部系统的核心对象模型，甚至 O/R mapping 机制，数据表结构等等产生了一定关联度，这样一来，内部的重构经常都会可能影响到外部的用户。

所以，这里再次对 Facade 和 DTO 产生了需求，用它们作为中介者和缓冲带，隔离内外系统，把内部系统变化对外部的冲击减少到最小程度。

## 服务设计原则 6：契约先行

Service 是往往涉及不同组织之间的合作，而按照正常逻辑，两个组织之间合作的首要任务，就是先签订明确的契约，详细规定双方合作的内容，合作的形式等等，这样才能对双方形成强有力的约束和保障，同时大家的工作也能够并行不悖，不用相互等待。因此 SOA 中，最佳的实践方式也是契约先行，即先做契约的设计，可以有商务，管理和技术等不同方面的人员共同参与，并定义出相应的 WSDL 或者 IDL，然后在开发的时候再通过工具自动生成目标语言的对应代码。

对于 WSDL 来说，做契约先行的门槛略高，如果没有好的 XML 工具很难手工编制。但对于 Thrift IDL 或者 ProtocolBuffer 等来说，由于它们和普通编程语言类似，所以契约设计相对是比较容易的。另外，对于简单的 HTTP + JSON 来说（假设不补充使用其他描述语言），由于 JSON 没有标准的 schema，所以是没法设计具有强约束力的契约的，只能用另外的文档做描述或者用 JSON 做输入输出的举例。

但是，契约先行，然后再生成服务提供端的代码，毕竟给 service 开发工作带来了较大的不便，特别是修改契约的时候导致代码需要重写。因此，这里同样可能需要引入 Facade 和 DTO，即用契约产生的都是 Facade 和 DTO 代码，它们负责将请求适配和转发到其他内部程序，而内部程序则可以保持自己的主导性和稳定性。

另外，契约先行可能会给前面提到的多远程调用支持带来一些麻烦。

当然契约先行也许并不是能被广泛接受的实践方式，就像敏捷开发中“测试先行”（也就是测试驱动开发）通常都是最佳实践，但真正施行的团队却非常之少，这方面还需要不断摸索和总结。但我们至少可以认为 Echo 中 Java2WSDL 并不被认为是 SOA 的最佳实践。

## 服务设计原则 7：稳定和兼容的契约

由于用户范围的广泛性，所以 SO 的服务契约和 Java 标准 API 类似，在公开展布之后就要保证相当的稳定性，不能随便被重构，即使升级也要考虑尽可能的向下兼容性。同时，如果用契约先行的方式，以后频繁更改契约也导致开发人员要不断重做契约到目标语言映射，非常麻烦。

这就是说 SO 对契约的质量要求可能大大高于一般的 OO 接口，理想的情况下，甚至可能需要专人（包括商务人员）来设计和评估 SO 契约（不管是否用契约先行的方式），而把内部的程序实现交给不同的人，而两者用 Facade 和 DTO 做桥梁。

## 服务设计原则 8：契约包装

前述原则基本都是针对 service 提供端来讲的，而对 service 消费端而言，通过契约生成对应的客户端代码，经常就可以直接使用了。当然，如果契约本身就是 Java 接口之类（比如在 Dubbo, Spring Remoting 等框架中），可以略过代码生成的步骤。

但是，service 的返回值（DTO）和 service 接口（Facade），可能被消费端的程序到处引用到。

这样消费端程序就较强的耦合在服务契约上了，如果服务契约不是消费端定义的，消费端就等于把自己程序的部分主导权完全让渡给了别人。

一旦契约做更改，或者消费端要选择完全不同的 service 提供方（有不同的契约），甚至改由本地程序自己来实现相关功能，修改工作量就可能非常大了。

另外，通过契约生成的客户端代码，经常和特定传输方式是相关的（比如 webservices stub），这样给切换远程调用方式也会带来障碍。

因此，就像在通常应用中，我们要包装数据访问逻辑（OO 中的 DAO 或者 Repository 模式），或者包装基础服务访问逻辑（OO 中的 Gateway 模式）一样，在较理想的 SOA 设计中，我们也可以考虑包装远程 service 访问逻辑，由于没有恰当的名称，暂时称之为 Delegate Service 模式，它由消费端自己主导定义接口和参数类型，并将调用转发给真正的 service 客户端生成代码，从而对它的使用者完全屏蔽了服务契约，这些使用者甚至不知道这个服务到底是远程提供的还是本地提供的。

此外，即使我们在消费端是采用某些手工调用机制（如直接构建和解析 json 等内容，直接收发 JMS 消息等），我们同样可以用 delegate service 来包装相应的逻辑。

delegate service 示例 1：

```
// ArticleService 是消费端自定义的接口
class ArticleServiceDelegate implements ArticleService {
    // 假设是某种自动生成的 service 客户端 stub 类
    private ArticleFacadeStub stub;
```

```
public void deleteArticles(List<Long> ids) {  
    stub.deleteArticles(ids);  
}  
}
```

delegate service 示例 2:

```
// ArticleService 是消费端自定义的接口  
class ArticleServiceDelegate implements ArticleService {  
  
    public void deleteArticles(List<Long> ids) {  
        // 用 JMS 和 FastJson 手工调用远程 service  
        messageClient.sendMessage(queue, JSON.toJSONString(ids));  
    }  
}
```

## 从面向对象到面向服务，再从面向服务到面向对象

总结上面的几个原则，虽然只是谈及有限的几个方面，但大致也可看出 OO 和 SO 在实际的设计开发中还是有不少显著的不同之处，而且我们没有打算用 SO 的原则来取代过去的 OO 设计，而是引入额外的层次、对象和 OO 设计模式，来补充传统的 OO 设计。

其实就是形成了这种调用流程：

- service 提供端：OO 程序 <- SOA 层（Facade 和 DTO）<- 远程消费端
- service 消费端：OO 程序 -> Delegate Service -> SOA 层（Facade 和 DTO 或者 其他动态调用机制）-> 远程提供端

Facade、DTO 和 Delegate Service 负责做 OO 到 SO 和 SO 到 OO 的中间转换。

现在，可以回答 Echo 示例中的问题：通过“透明的”配置方式，将 OO 程序发布为远程 Service，虽然可能较好的完成了从本地对象到远程对象的跨越，但通常并不能较好的完成 OO 到 SO 的真正跨越。

同时，透明配置方式也通常无法直接帮助遗留应用（如 ERP 等）转向 SOA。

当然，在较为简单和使用范围确定很有限应用（比如传统和局部的 RPC）中，透明式远程 service 发布会带来极大的便利。

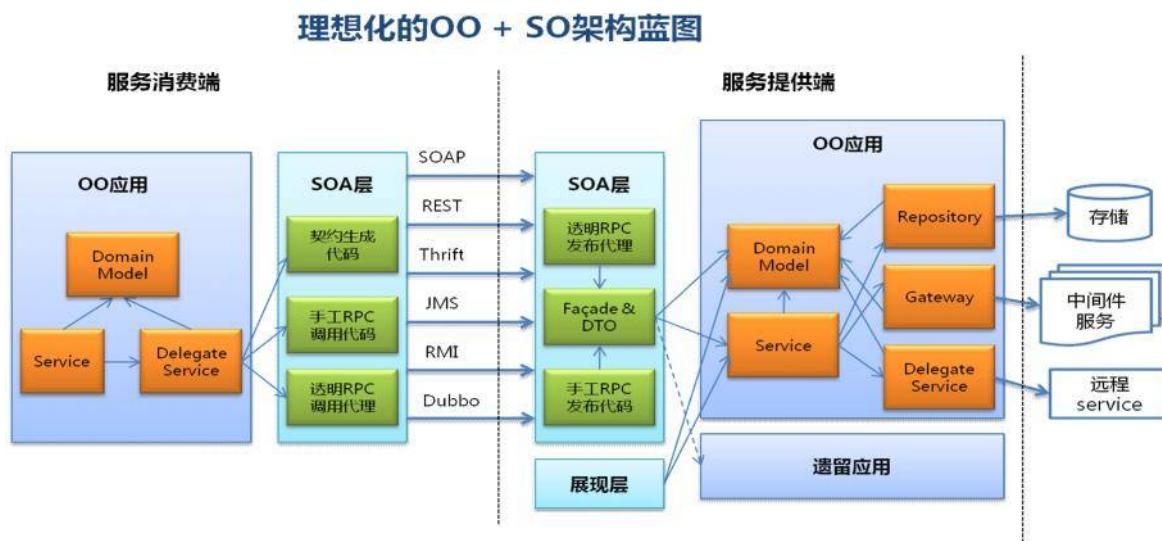
另外，上面对 SO 的所有讨论都集中在 RPC 的方式，其实 SO 中也用 message 的方式做集成，它也是个大话题，暂时不在此详论了。

# 为什么不能放弃面向对象

SO是有它的特定场景的，比如远程的，范围不定的客户端。所以它的那些设计原则并不能被借用来指导一般性的程序开发，比如很多 OO 程序和 SO 原则完全相反，经常都要提供细粒度接口和复杂参数类型以追求使用的使用灵活性和功能的强大性。

就具体架构而言，我认为 SOA 层应该是一个很薄的层次（thin layer），将 OO 应用或者其他遗留性应用加以包装和适配以帮助它们面向服务。其实在通常的 web 开发中，我们也是用一个薄的展现层（或者叫 Web UI 层之类）来包装 OO 应用，以帮助它们面向浏览器用户。因此，Facade、DTO 等不会取代 OO 应用中核心的 Domain Model、Service 等等（这里的 service 是 OO 中 service，未必是 SO 的）。

综合起来，形成类似下面的体系结构：



## 理想和现实

需要特别指出的是，上面提到的诸多 SO 设计原则是在追求一种相对理想化的设计，以达到架构的优雅性，高效性，可重用性，可维护性，可扩展性，等等。

而在现实中任何理论和原则都可能是需要作出适当妥协的，因为现实是千差万别的，其情况远比理论复杂，很难存在放之四海而皆准的真理。

而且很多方面似乎本来也没有必要追求完美和极致，比如如果有足够能力扩充硬件基础设施，就可以考虑传输一些冗余数据，选择最简单传输方式，并多来几次远程调用等等，以减轻设计开发的工作量。

那么理想化的原则就没有意义了吗？比如领域驱动设计（Domain-Driven Design）被广泛认为是最理想的 OO 设计方式，但极少有项目能完全采用它；测试驱动开发也被认为是最佳的敏捷开发方式，但同样极少有团队能彻底采用它。但是，恐怕没有多少人在了解它们之后会否认它们巨大的意义。

理想化的原则可以更好的帮助人们理解某类问题的本质，并做为好的出发点或者标杆，帮助那些可以灵活运用，恰当取舍的人取得更大的成绩，应付关键的挑战。这正如孔子说的“取乎其上，得乎其中；取乎其中，得乎其下；取乎其下，则无所得矣”。

另外，值得一提的是，SOA 从它的理念本身来说，就带有一些的理想主义的倾向，比如向“全世界”开放，不限定客户端等等。如果真愿意按 SOA 的路径走，即使你是个土豪，偷个懒比浪费网络带宽重要，但说不定你的很多用户是土鳖公司，浪费几倍的带宽就大大的影响他们的利润率。

## 延伸讨论：SOA 和敏捷软件开发矛盾吗？

SOA 的服务契约要求相当的稳定性，一旦公开发布（或者双方合同商定）就不应该有经常的变更，它需要对很多方面有极高的预判。而敏捷软件开发则是拥抱变化，持续重构的。软件设计大师 Martin Fowler 把它们归结为计划式设计和演进式设计的不同。

计划理论（或者叫建构理论）和演进理论是近代哲学的两股思潮，影响深远，派生出了比如计划经济和市场经济，社会主义和自由主义等各种理论。

但是，计划式设计和演进式设计并不绝对矛盾，就像计划经济和市场经济也不绝对矛盾，非此即彼，这方面需要在实践中不断摸索。前面我们讨论的设计原则和架构体系，就是将 SOA 层和 OO 应用相对隔离，分而治之，在 SOA 层需要更多计划式设计，而 OO 应用可以相对独立的演进，从而在一定程度缓解 SOA 和敏捷开发的矛盾。

## 延伸讨论：SOA 和 REST 是一回事吗？

从最本质的意义上讲，REST（Representational State Transfer）实际是一种面向资源架构（ROA），和面向服务架构（SOA）是有根本区别的。

例如，REST 是基于 HTTP 协议，对特定资源做增（HTTP POST）、删（HTTP DELETE）、改（HTTP PUT）、查（HTTP GET）等操作，类似于 SQL 中针对数据表的 INSERT、DELETE、UPDATE、SELECT 操作，故 REST 是以资源（资源可以类比为数据）为中心的。而 SOA 中的 service 通常不包含这种针对资源（数据）的细粒度

操作，而是面向业务用例、业务流程的粗粒度操作，所以 SOA 是以业务逻辑为中心的。

但是在实际使用中，随着许多 REST 基本原则被不断突破，REST 的概念被大大的泛化了，它往往成为很多基于 HTTP 的轻量级远程调用的代名词（例如前面提到过的 HTTP + JSON）。比如，即使是著名的 Twitter REST API 也违反不少原始 REST 的基本原则。

在这个泛化的意义上讲，REST 也可以说是有助于实现 SOA 的一种轻量级远程调用方式。

## SOA 架构的进化

前面讨论的 SOA 的所有问题，基本都集中在 service 本身的设计开发。但 SOA 要真正发挥最大作用，还需要不断演进成更大的架构（也就是从微观 SOA 过渡到宏观 SOA），在此略作说明：

- 第一个层次是 **service 架构**：开发各种独立的 service 并满足前面的一些设计原则，我们前面基本都集中在讨论这种架构。这些独立的 service 有点类似于小孩的积木。
- 第二个层次是 **service composition**（组合）架构：独立的 service 通过不同组合来构成新的业务或者新的 service。在理想情况下，可以用一种类似小孩搭积木的方式，充分发挥想象力，将独立的积木（service）灵活的拼装组合成新的形态，还能够自由的替换其中的某个构件。这体现出 SOA 高度便捷的重用性，大大提高企业的业务敏捷度。
- 第三个层次是 **service inventory**（清单）架构：通过标准化企业服务清单（或者叫注册中心）统一的组织和规划 service 的复用和组合。当积木越来越多了，如果还满地乱放而没有良好的归类整理，显然就玩不转了。
- 第四个层次是 **service-oriented enterprise** 架构……

## 总结

至此，我们只是简要的探讨了微观层面的 SOA，特别是一些基本设计原则及其实践方式，以期能够略微展示 SOA 在实践中的本质，以有助于 SOA 更好的落地，进入日常操作层面。

最后，打个比方：SOA 不分贵贱（不分语言、平台、组织），不远万里（通过远程调用）的提供服务（service），这要求的就是一种全心全意为人民服务的精神……

---

## 作者简介

沈理，当当网架构师和技术委员会成员，主要负责当当网的 SOA 实施（即服务化）以及分布式服务框架的开发。以前也有在 BEA、Oracle、Redhat 等外企的长期工作经历，从事过多个不同 SOA 相关框架和容器的开发。他的邮箱：[shenli@dangdang.com](mailto:shenli@dangdang.com)

---

感谢马国耀对本文的审校。

查看原文：[微观 SOA：服务设计原则及其实践方式（下篇）](#)

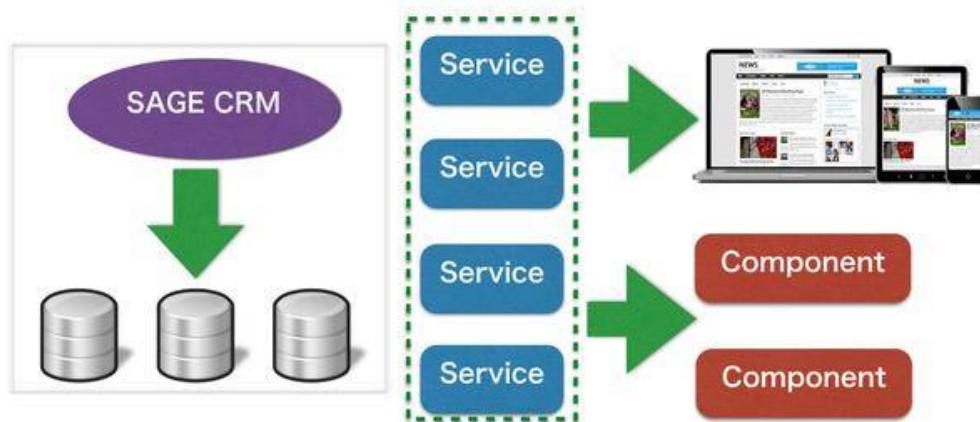
# 基于微服务架构，改造企业核心系统之实践

作者 王磊

## 1. 背景与挑战

随着公司国际化战略的推行以及本土业务的高速发展，后台支撑系统已经不堪重负。在吞吐量、稳定性以及可扩展性上都无法满足日益增长的业务需求。对于每 10 万元额度的合同，从销售团队准备材料、与客户签单、递交给合同部门，再到合同生效大概需要 3.5 人天。随着业务量的快速增长，签订合同的成本急剧增加。

合同管理系统是后台支撑系统中重要的一部分。当前的合同系统是 5 年前使用.NET 基于 [SAGE CRM](#) 二次开发的产品。一方面，系统架构过于陈旧，性能、可靠性无法满足现有的需求。另一方面，功能繁杂，结构混乱，定制的代码与 SAGE CRM 系统耦合度极高。由于是遗留系统，熟悉该代码的人早已离职多时，新团队对其望而却步，只能做些周边的修补工作。同时，还要承担着边补测试，边整理逻辑的工作。



在无法中断业务处理的情况下，为了解决当前面临的问题，团队制定了如下的策略：

- 在现有合同管理系统的外围，构建功能服务接口，将系统核心的功能分离出来。
- 利用这些功能服务接口作为代理，解耦原合同系统与其调用者之间的依赖。
- 通过不断构建功能服务接口，逐渐将原有系统分解成多个独立的服务。

- 摒弃原有的合同管理系统，使用全新构建的（微）服务接口替代。

## 2. 什么是微服务

多年来，我们一直在技术的浪潮中不断乘风破浪，扬帆奋进，寻找更好的方式构建 IT 系统。微服务架构(Micro Service Architect)是近一段时间在软件体系架构领域里出现的一个新名词。它通过将功能分解到多个独立的服务，以实现对解决方案或者复杂系统的解耦。

微服务的诞生并非偶然：[领域驱动设计](#)指导我们如何分析并模型化复杂的业务；[敏捷方法论](#)帮助我们消除浪费，快速反馈；[持续交付](#)促使我们构建更快、更可靠、更频繁的软件部署和交付能力；虚拟化和基础设施自动化( Infrastructure As Code)则帮助我们简化环境的创建、安装；[DevOps](#)文化的流行以及特性团队的出现，使得小团队更加全功能化。这些都是推动微服务诞生的重要因素。

实际上，微服务本身并没有一个严格的定义。不过从业界的讨论来看，微服务通常有如下几个特征：

### 1、小，且专注于做一件事情

每个服务都是很小的应用，至于有多小，是一个非常有趣的话题。有人喜欢 100 行以内，有人赞成 1000 行以内。数字并不是最重要的。仁者见仁，智者见智，只要团队觉得合适就好。只关注一个业务功能，这一点和我们平常谈论的面向对象原则中的”单一职责原则”类似，每个服务只做一件事情，并且把它做好。

### 2、运行在独立的进程中

每个服务都运行在一个独立的操作系统进程中，这意味着不同的服务能被部署到不同的主机上。

### 3、轻量级的通信机制

服务和服务之间通过轻量级的机制实现彼此间的通信。所谓轻量级通信机制，通常指基于语言无关、平台无关的这类协议，例如 XML、JSON，而不是传统我们熟知的 Java RMI 或者 .Net Remoting 等。

### 4、松耦合

不需要改变依赖，只更改当前服务本身，就可以独立部署。这意味着该服务和其他服务之间在部署和运行上呈现相互独立的状态。

综上所述，微服务架构采用多个服务间互相协作的方式构建传统应用。每个服务独立运行在不同的进程中，服务与服务之间通过轻量的通讯机制交互，并且每个服务可以通过自动化部署方式独立部署。

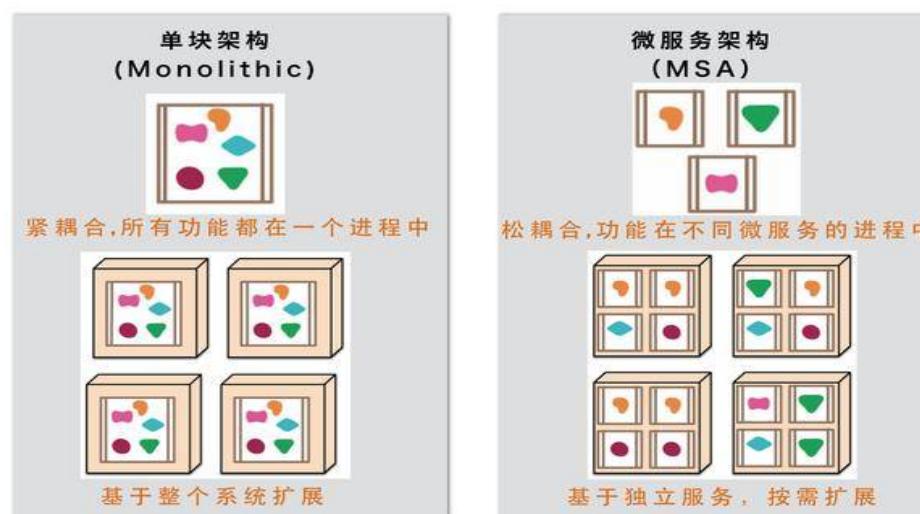
### 3. 微服务的优势

相比传统的单块架构系统（monolithic），微服务在如下诸多方面有着显著的优势。

#### 异构性

问题有其具体性，解决方案也应有其针对性。用最适合的技术方案去解决具体的问题，往往会事半功倍。传统的单块架构系统倾向采用统一的技术平台或方案来解决所有问题。而微服务的异构性，可以针对不同的业务特征选择不同的技术方案，有针对性的解决具体的业务问题。

对于单块架构的系统，初始的技术选型严重限制将来采用不同语言或框架的能力。如果想尝试新的编程语言或者框架，没有完备的功能测试集，很难平滑的完成替换，而且系统规模越大，风险越高。基于微服务架构，使我们更容易在遗留系统上尝试新的技术或解决方案。譬如说，可以先挑选风险最小的服务作为尝试，快速得到反馈后再决定是否试用于其他服务。这也意味着，即便对一项新技术的尝试失败，也可以抛弃这个方案，并不会对整个产品带来风险。



该图引用自 Martin Fowler 的 [Microservices](#) 一文。

## 独立测试与部署

单块架构系统运行在一个进程中，因此系统中任何程序的改变，都需要对整个系统重新测试并部署。而对于微服务架构而言，不同服务之间的打包、测试或者部署等，与其它服务都是完全独立的。对某个服务所做的改动，只需要关注该服务本身。从这个角度来说，使用微服务后，代码修改、测试、打包以及部署的成本和风险都比单块架构系统降低很多。

## 按需伸缩

单块架构系统由于单进程的局限性，水平扩展时只能基于整个系统进行扩展，无法针对某一个功能模块按需扩展。而服务架构则可以完美地解决伸缩性的扩展问题。系统可以根据需要，实施细粒度的自由扩展。

## 错误隔离性

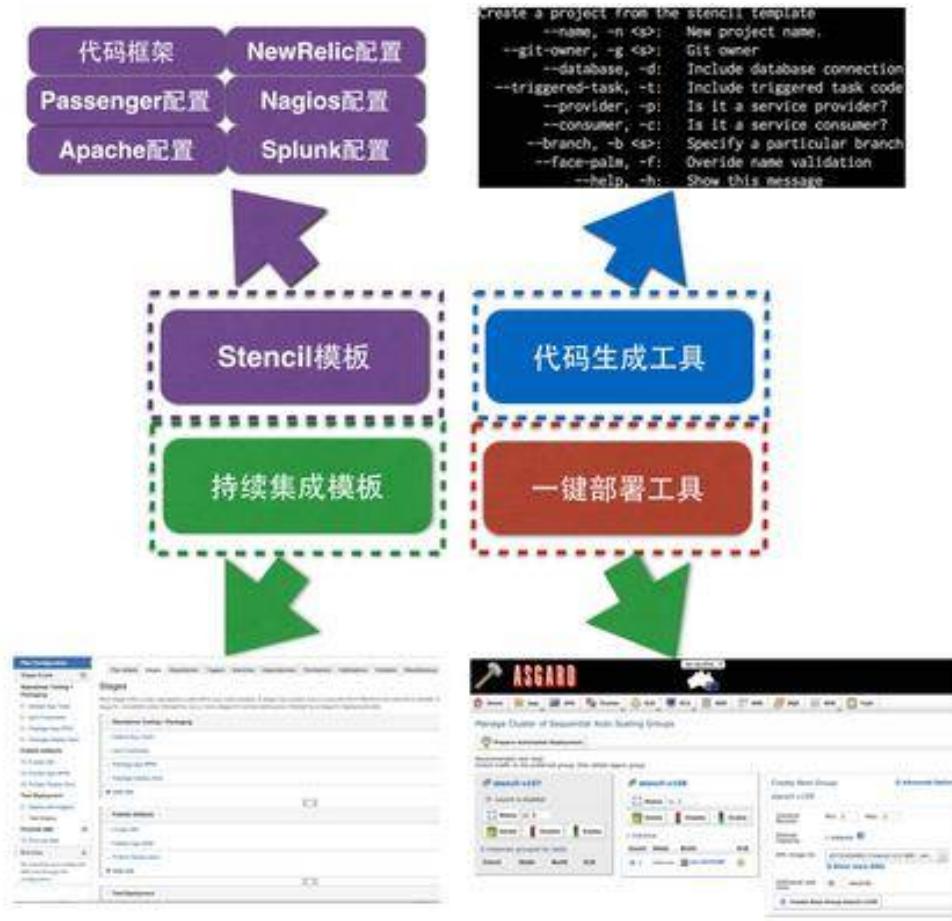
微服务架构同时也能提升故障的隔离性。例如，如果某个服务的内存泄露，只会影响自己，其他服务能够继续正常地工作。与之形成对比的是，单块架构中如果有一个不合格的组件发生异常，有可能会拖垮整个系统。

## 团队全功能化

康威定律（Conway's law）指出：一个组织的设计成果，其结构往往对应于这个组织中的沟通结构（*organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations*）。传统的开发模式在分工时往往以技术为单位，比如 UI 团队、服务端团队和数据库团队，这样的分工可能会导致任何功能上的改变都需要跨团队沟通和协调。而微服务则倡导围绕服务来分工，团队需要具备服务设计、开发、测试到部署所需的所有技能。

## 4. 微服务快速开发实践

随着团队对业务的理解加深和对微服务实践的尝试，数个微服务程序已经成功构建出来。不过，问题同时也出现了：对于这些不同的微服务程序而言，虽然具体实现的代码细节不同，但其结构、开发方式、持续集成环境、测试策略、部署机制以及监控和告警等，都有着类似的实现方式。那么如何满足 [DRY 原则](#)并消除浪费呢？带着这个问题，经过团队的努力，Stencil 诞生了。Stencil 是一个帮助快速构建 Ruby 微服务应用的开发框架，主要包括四部分：Stencil 模板、代码生成工具，持续集成模板以及一键部署工具。



## Stencil 模板

Stencil 模板是一个独立的 Ruby 代码工程库，主要包括代码模板以及一组配置文件模板。

代码模板使用 [Webmachine](#) 作为 Web 框架，[RESTful](#) 和 JSON 构建服务之间的通信方式，RSpec 作为测试框架。同时，代码模板还定义了一组 Rake 任务，譬如运行测试，查看测试报告，将当前的微服务生成 RPM 包，使用 Koji 给 RPM 包打标签等。

除此之外，该模板也提供了一组通用的 URL，帮助使用者查看微服务的当前版本、配置信息以及检测该微服务程序是否健康运行等。

```
[
  {
    rel: "index",
    path: "/diagnostic/"
  },
  {
    rel: "version",
    path: "/diagnostic/version"
  }
]
```

```
{  
    rel: "config",  
    path: "/diagnostic/config"  
},  
{  
    rel: "hostname",  
    path: "/diagnostic/hostname"  
},  
{  
    rel: "heartbeat",  
    path: "/diagnostic/status/heartbeat"  
},  
{  
    rel: "nagios",  
    path: "/diagnostic/status/nagios"  
}  
]
```

配置文件模板主要包括 [NewRelic](#) 配置, [Passenger](#) 配置、[Nagios](#) 配置、[Apache](#) 配置以及 [Splunk](#) 配置。通过定义这些配置文件模板, 当把新的微服务程序部署到验收环境或者产品环境时, 我们立刻就可以使用 Nagios、NewRelic 以及 Splunk 等第三方服务提供的功能, 帮助我们有效的监控微服务, 并在超过初始阈值时获得告警。

## 代码生成工具

借助 Stencil 代码生成工具, 我们能在很短时间内就构建出一个可以立即运行的微服务应用程序。随着系统越来越复杂, 微服务程序的不断增多, Stencil 模板和代码生成工具帮助我们大大简化了创建微服务的流程, 让开发人员更关注如何实现业务逻辑并快速验证。

```
Create a project from the stencil template (version 0.1.27)  
  --name, -n <s>:  New project name. eg. things-and-stuff  
  --git-owner, -g <s>:  Git owner (default: which team or owner)  
  --database, -d:  Include database connection code  
  --triggered-task, -t:  Include triggered task code  
  --provider, -p:  Is it a service provider? (other services use this  
service)  
  --consumer, -c:  Is it a service consumer? (it uses other services)  
  --branch, -b <s>:  Specify a particular branch of Stencil  
  --face-palm, -f:  Override name validation  
  --help, -h:  Show this message
```

如上代码所示, 通过指定不同参数, 我们能创建具有数据库访问能力的微服务程序, 或者是包含异步队列处理的微服务程序。同时, 我们也可以标记该服务是数据消费者还是数据生产者, 能帮助我们理解多个微服务之间的联系。

## 持续集成模板

基于持续集成服务器 [Bamboo](#)，团队创建了针对 Stencil 的持续集成模板工程，并定义了三个主要阶段：

- 打包：运行单元测试，集成测试，等待测试通过后生成 RPM 包。
- 发布：将 RPM 包发布到 [Koji](#) 服务器上，并打上相应的 Tag。然后使用 [Packer](#) 在亚马逊 AWS 云环境中创建 AMI，建好的 AMI 上已经安装了当前微服务程序的最新 RPM 包。
- 部署：基于指定版本的 AMI，将应用快速部署到验收环境或者产品环境上。

利用持续集成模板工程，团队仅需花费很少的时间，就可以针对新建的微应用程序，在 Bamboo 上快速定义其对应的持续集成环境。

## 一键部署工具

所有的微服务程序都部署并运行在亚马逊 AWS 云环境上。同时，我们使用 [Asgard](#) 对 AWS 云环境中的资源进行创建、部署和管理。Asgard 是一套功能强大的基于 Web 的 AWS 云部署和管理工具，由 Netflix 采用 [Groovy on Grails](#) 开发，其主要优点有：

- 基于 B/S 的 AWS 部署及管理工具，使用户能通过浏览器直接访问 AWS 云资源，无需设置 Secret Key 和 Access Key。
- 定义了`Application`以及`Cluster`等逻辑概念，更清晰、有效地描述了应用程序在 AWS 云环境中对应的部署拓扑结构。
- 在对应用的部署操作中，集成了 AWS Elastic Load Balancer、AWS EC2 以及 AWS Autoscaling Group，并将这些资源自动关联起来。
- 提供 RESTful 接口，能够方便地与其他系统集成。
- 简洁易用的用户接口，提供可视化的方式完成一键部署以及流量切换。

由于 Asgard 对 RESTful 的良好支持，团队实现了一套基于 Asgard 的命令行部署工具，只需如下一条命令，提供应用程序的名称以及版本号，就可自动完成资源的创建、部署、流量切换、删除旧的应用等操作。

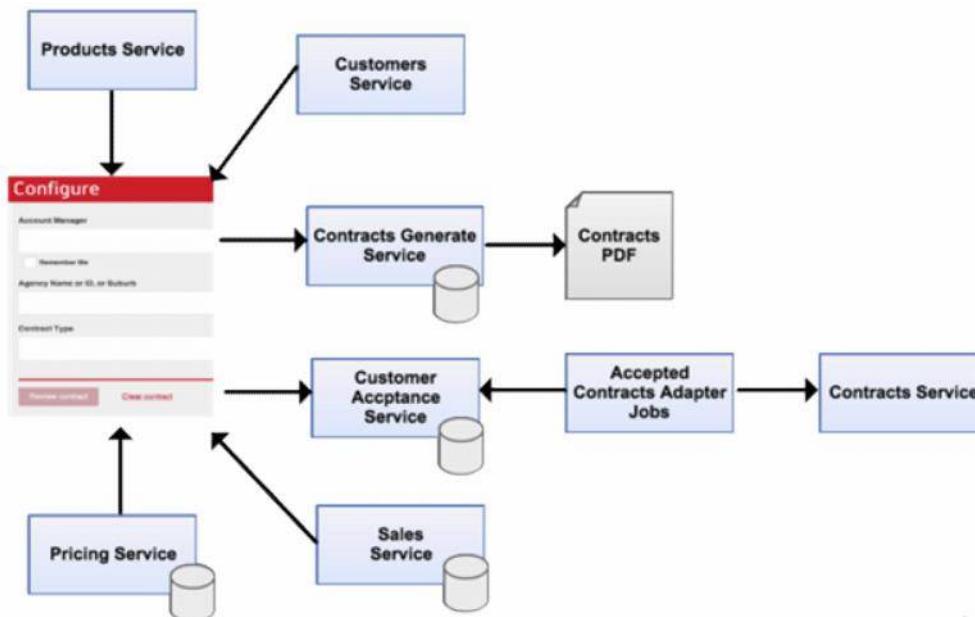
```
asgard-deploy [AppName] [AppVersion]
```

同时，基于命令行的部署工具，也可以很容易的将自动化部署集成到 Bamboo 持续集成环境。

通过使用微服务框架 Stencil，大大缩短了团队开发微服务的周期。同时，基于 Stencil，我们定义了一套团队内部的开发流程，帮助团队的每一位成员理解并快速构建微服务。

## 5、微服务架构下的新系统

经过 5 个月的努力，我们重新构建了合同管理系统，将之前的产品、价格、销售人员、合同签署、合同审查以及 PDF 生成都定义成了独立的服务接口。相比之前大而全、难以维护的合同管理系统而言，新的系统由不同功能的微服务组成，每个微服务程序只关注单一的功能。每个微服务应用都有相关的负责人，通过使用 [Page Duty](#) 建立消息通知机制。每当有监控出现告警的时候，责任人能立即收到消息并快速做出响应。



由于微服务具有高内聚，低耦合的特点，每个应用都是一个独立的个体。当出现问题时，很容易定位问题并解决问题，大大缩短了修复缺陷的周期。另外，通过使用不同功能的微服务接口提供数据，用户接口（UI）部分变成了一个非常简洁、轻量级的应用，更关注如何渲染页面以及表单提交等交互功能。

## 总结

通过使用微服务架构，在不影响现有业务运转的情况下，我们有效的将遗留的大系统逐渐分解成不同功能的微服务接口。同时，通过 Stencil 微服务开发框架，我们能够快速地构建不同功能的微服务接口，并能方便地将其部署到验收环境或者产品环境。最后，得益于微服务架构的灵活性以及扩展性，使得我们能够快速构建低耦合、易扩展、易伸缩性的应用系统。

## 参考文献

1. <http://martinfowler.com/articles/microservices.html>
  2. <http://jaxenter.com/cracking-microservices-practices-50692.html>
  3. <http://microservices.io/patterns/microservices.html>
- 

## 作者简介

王磊，ThoughtWorks 公司的程序员，咨询师。开源软件的爱好者和贡献者，社区活动的参与者，Practical RubyGems 的译者。于 2012 年加入 ThoughtWorks，为国内外诸多客户提供项目交付和咨询服务；在加入 ThoughtWorks 之前，曾就职过多家知名外企，具有丰富的敏捷项目实战经验。目前致力于微服务架构、高可用性的 Web 应用以及 Devops 的研究与实践。

感谢张逸对本文的审校。

查看原文：[基于微服务架构，改造企业核心系统之实践](#)



InfoQ在线课堂

# AWS存储服务特点 与典型应用场景

2014年11月18日 20:30–21:30 讲师：张波

## 课程介绍

存储在应用架构设计中非常关键的一环。不同的应用，对于存储的并发访问量、响应时间、数据持久性保护、高可用、访问接口以及成本等都有不同的需求。本期在线讲座，将从互联网时代应用程序对数据存储的多种需求出发，全面介绍AWS云平台所提供的多种数据存储服务，其中包括完全基于非结构化数据存储的简单存储服务S3，侧重于磁盘性能的弹性块存储（EBS），传统的关系型数据库服务，NoSQL 数据库DynamoDB以及缓存集群服务等，并以客户案例为例说明这些服务的实际应用场景。

# Apache Tez 是什么？

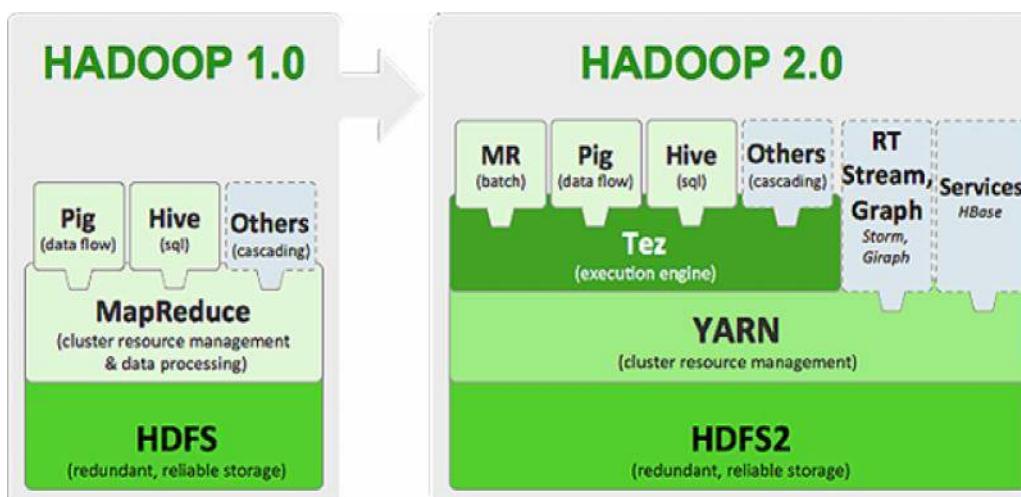
作者 Roopesh Shenoy，译者 孙镜涛

你可能听说过 Apache Tez，它是一个针对 Hadoop 数据处理应用程序的新分布式执行框架。但是它到底是什么呢？它的工作原理是什么？哪些人应该使用它，为什么？如果你有这些疑问，那么可以看一下 Bikas Saha 和 Arun Murthy 提供的呈现“Apache Tez: 加速 Hadoop 查询处理”，在这个呈现中他们讨论了 Tez 的设计，它的一些突出亮点，同时还分享了通过让 Hive 使用 Tez 而不是 MapReduce 而获得的一些初步成果。

——呈现记录由 [Roopesh Shenoy](#) 编辑

Tez 是 Apache 最新的支持 DAG 作业的开源计算框架，它可以将多个有依赖的作业转换为一个作业从而大幅提升 DAG 作业的性能。Tez 并不直接面向最终用户——事实上它允许开发者为最终用户构建性能更快、扩展性更好的应用程序。Hadoop 传统上是一个大量数据批处理平台。但是，有很多用例需要近乎实时的查询处理性能。还有一些工作则不太适合 MapReduce，例如机器学习。Tez 的目的就是帮助 Hadoop 处理这些用例场景。

Tez 项目的目标是支持高度定制化，这样它就能够满足各种用例的需要，让人们不必借助其他的外部方式就能完成自己的工作，如果 [Hive](#) 和 [Pig](#) 这样的项目使用 Tez 而不是 MapReduce 作为其数据处理的骨干，那么将会显著提升它们的响应时间。Tez 构建在 [YARN](#) 之上，后者是 Hadoop 所使用的新资源管理框架。



## 设计哲学

Tez产生的主要原因是绕开 MapReduce 所施加的限制。除了必须要编写 Mapper 和 Reducer 的限制之外，强制让所有类型的计算都满足这一范例还有效率低下的问题——例如使用 HDFS 存储多个 MR 作业之间的临时数据，这是一个负载。在 Hive 中，查询需要对不相关的 key 进行多次 shuffle 操作的场景非常普遍，例如 join - grpby - window function - order by。

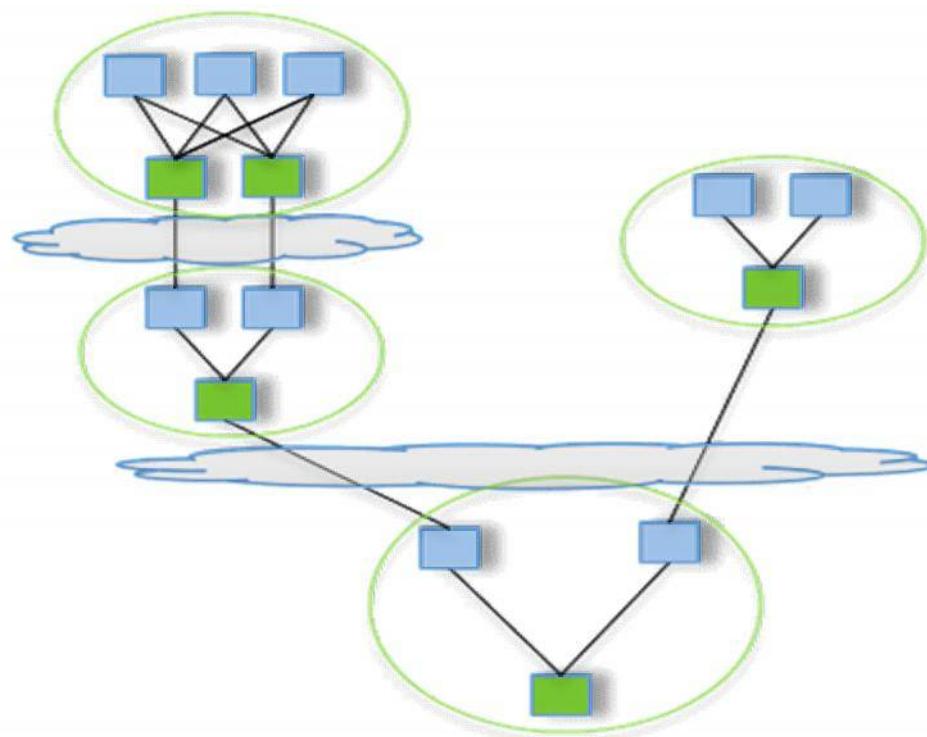
Tez 设计哲学里面的关键元素包括：

- 允许开发人员（也包括最终用户）以最有效的方式做他们想做的事情。
- 更好的执行性能。

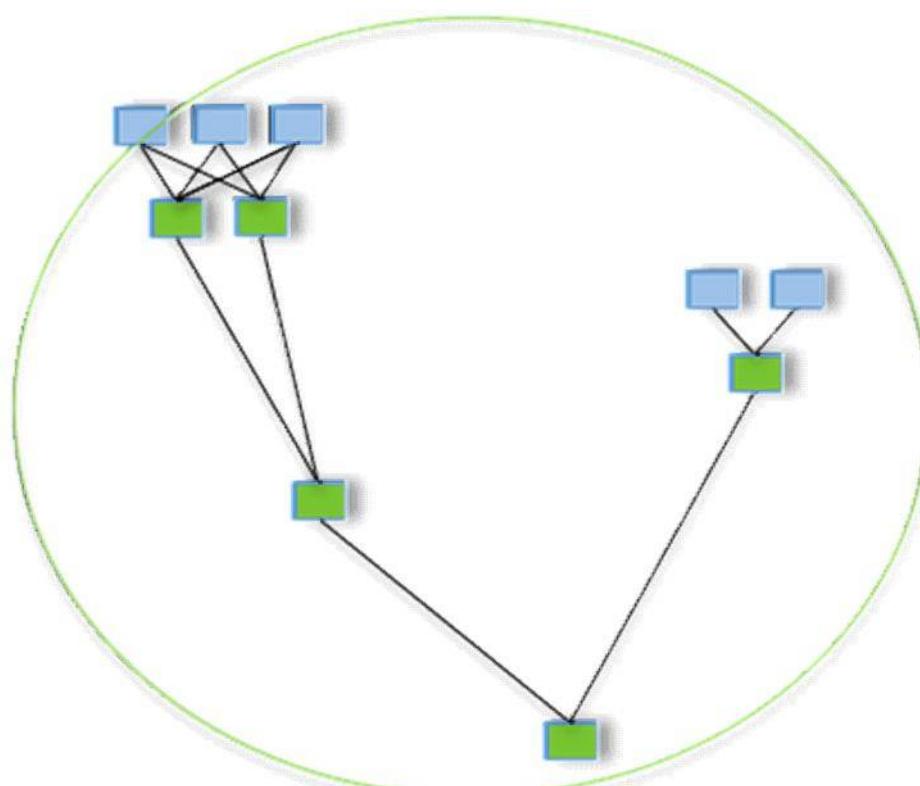
Tez之所以能够实现这些目标依赖于以下内容：

- 具有表现力的数据流 API——Tez 团队希望通过一套富有表现力的数据流定义 API 让用户能够描述他们所要运行计算的有向无环图（DAG）。为了达到这个目的，Tez 实现了一个结构化类型的 API，你可以在其中添加所有的处理器和边，并可视化实际构建的图形。
- 灵活的输入—处理器—输出（Input-Processor-Output）运行时模型——可以通过连接不同的输入、处理器和输出动态地构建运行时执行器。
- 数据类型无关性——仅关心数据的移动，不关心数据格式（键值对、面向元组的格式等）。
- 动态图重新配置。
- 简单地部署——Tez 完全是一个客户端应用程序，它利用了 YARN 的本地资源和分布式缓存。就 Tez 的使用而言，你不需要在自己的集群上部署任何内容，仅需要将相关的 Tez 类库上传到 HDFS 上，然后使用 Tez 客户端提交这些类库即可。
- 你甚至可以在你的集群上放置两份类库。一份用于产品环境，它使用稳定版本供所有的生产任务使用；另一份使用最新版本，供用户体验。这两份类库相互独立，互不影响。
- Tez 能够运行任意 MR 任务，不需要做任何改动。这样能够让那些现在依赖于 MR 的工具实现分布迁移。

接下来让我们详细地探索一下这些表现力丰富的数据流 API——看看可以使用它们做什么？例如，你可以使用 MRR 模式而不是使用多个 MapReduce 任务，这样一个单独的 map 就可以有多个 reduce 阶段；并且这样做数据流可以在不同的处理器之间流转，不需要把任何内容写入 HDFS（将会被写入磁盘，但这仅仅是为了设置检查点），与之前相比这种方式性能提升显著。下面的图表阐述了这个过程：



Pig/Hive - MR



Pig/Hive - Tez

第一个图表展示的流程包含多个 MR 任务，每个任务都将中间结果存储到 HDFS 上——前一个步骤中的 reducer 为下一个步骤中的 mapper 提供数据。第二个图表展示了使用 Tez 时的流程，仅在一个任务中就能完成同样的处理过程，任务之间不需要访问 HDFS。

Tez 的灵活性意味着你需要付出比 MapReduce 更多的努力才能使用它，你需要学习更多的 API，需要实现更多的处理逻辑。但是这还好，毕竟它和 MapReduce 一样并不是一个面向最终用户的应用程序，其目的是让开发人员基于它构建供最终用户使用的应用程序。

以上内容是对 Tez 的概述及其目标的描述，下面就让我们看看它实际的 API。

## Tez API

Tez API 包括以下几个组件：

- 有向无环图 (DAG) —— 定义整体任务。一个 DAG 对象对应一个任务。
- 节点 (Vertex) —— 定义用户逻辑以及执行用户逻辑所需的资源和环境。一个节点对应任务中的一个步骤。
- 边 (Edge) —— 定义生产者和消费者节点之间的连接。
- 边需要分配属性，对 Tez 而言这些属性是必须的，有了它们才能在运行时将逻辑图展开为能够在集群上并行执行的物理任务集合。下面是一些这样的属性：
  - 数据移动属性，定义了数据如何从一个生产者移动到一个消费者。
  - 调度 (Scheduling) 属性 (顺序或者并行)，帮助我们定义生产者和消费者任务之间应该在什么时候进行调度。
  - 数据源属性 (持久的，可靠的或者暂时的)，定义任务输出内容的生命周期或者持久性，让我们能够决定何时终止。

如果你想查看一个 API 的使用示例，对这些属性的详细介绍，以及运行时如何展开逻辑图，那么可以看看 [Hortonworks 提供的这篇文章](#)。

[运行时 API 基于输入—处理器—输出模型](#)，借助于该模型所有的输入和输出都是可插拔的。为了方便，Tez 使用了一个基于事件的模型，目的是为了让任务和系统之间、组件和组件之间能够通信。事件用于将信息（例如任务失败信息）传递给所需的组件，将输出的数据流（例如生成的数据位置信息）传送给输入，以及在运行时对 DAG 执行计划做出改变等。

Tez 还提供了各种开箱即用的输入和输出处理器。

这些富有表现力的 API 能够让更高级语言（例如 Hive）的编写者很优雅地将自己的查询转换成 Tez 任务。

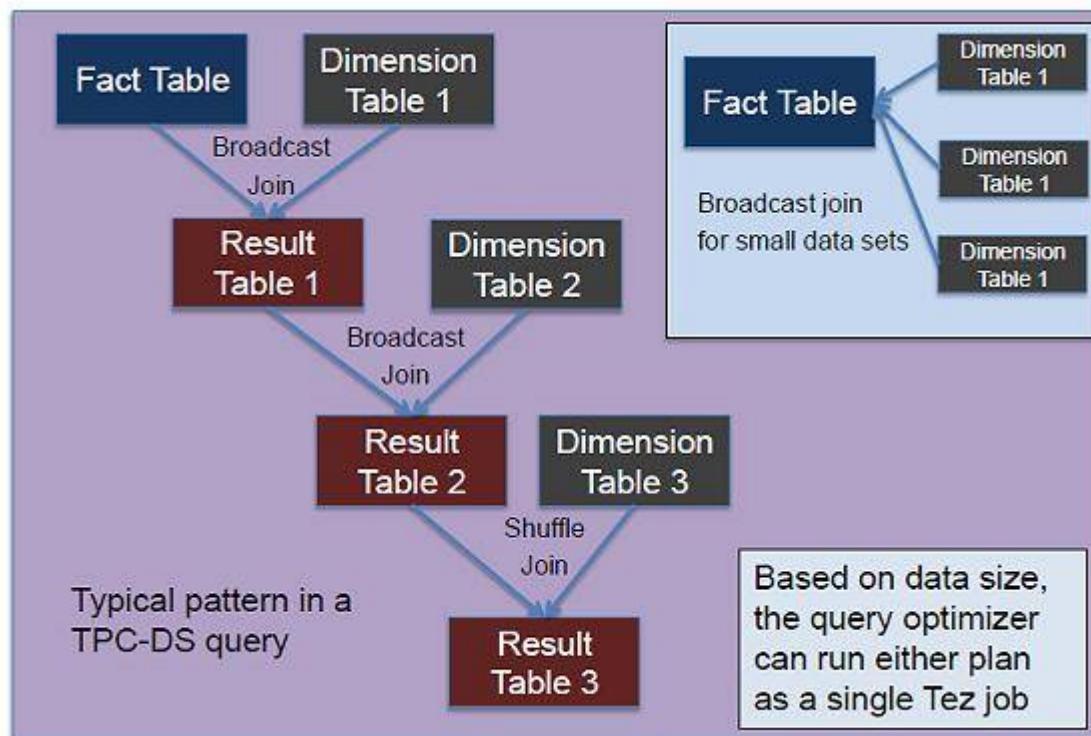
## Tez 调度程序

在决定如何分配任务的时候，Tez 调度程序考虑了很多方面，包括：任务位置需求、容器的兼容性、集群可利用资源的总量、等待任务请求的优先级、自动并行化、释放应用程序不再使用的资源（因为对它而言数据并不是本地的）等。它还维护着一个使用共享注册对象的预热 JVM 连接池。应用程序可以选择使用这些共享注册对象存储不同类型的预算算信息，这样之后再进行处理的时候就能重用它们而不需要重新计算了，同时这些共享的连接集合及容器池资源也能非常快地运行任务。

如果你想了解更多与容器重利用相关的信息，那么可以[查看这里](#)。

## 扩展性

总体来看，Tez 为开发人员提供了丰富的扩展性以便于让他们能够应对复杂的处理逻辑。这可以通过示例“Hive 是如何使用 Tez 的”来说明。



让我们看看这个经典的 TPC-DS 查询模式，在该模式中你需要将多个维度表与一个事实表连接到一起。大部分优化器和查询系统都能完成该图右上角部分所描述的场景：

如果维度表较小，那么可以将所有的维度表与较大的事实表进行广播连接，这种情况下你可以在 Tez 上完成同样的事情。

但是如果这些广播包含用户自定义的、计算成本高昂的函数呢？此时，你不可能都用这种方式实现。这就需要你将自己的任务分割成不同的阶段，正如该图左边的拓扑图所展示的方法。第一个维度表与事实表进行广播连接，连接的结果再与第二个维度表进行广播连接。

第三个维度表不再进行广播连接，因为它太大了。你可以选择使用 shuffle 连接，Tez 能够非常有效地导航拓扑。

使用 Tez 完成这种类型的 Hive 查询的好处包括：

- 它为你提供了全面的 DAG 支持，同时会自动地在集群上完成大量的工作，因而它能够充分利用集群的并行能力；正如上面所介绍的，这意味着在多个 MR 任务之间不需要从 HDFS 上读/写数据，通过一个单独的 Tez 任务就能完成所有的计算。
- 它提供了会话和可重用的容器，因此延迟低，能够尽可能地避免重组。

使用新的 Tez 引擎执行这个特殊的 Hive 查询性能提升将超过 100%。

## 路线图

- 更加丰富的 DAG 支持。例如，Samza 是否能够使用 Tez 作为其底层支撑然后在这上面构建应用程序？为了让 Tez 能够处理 Samza 的核心调度和流式需求开发团队需要做一些支持。Tez 团队将探索如何在我们的 DAG 中使用这些类型的连接模式。他们还想提供更好的容错支持，更加有效地数据传输，从而进一步优化性能，并且改善会话性能。
- 考虑到这些 DAG 的复杂度无法确定，需要提供很多自动化的工具来帮助用户理解他们的性能瓶颈。

## 总结

Tez 是一个支持 DAG 作业的分布式执行框架。它能够轻而易举地映射到更高级的声明式语言，例如 Hive、Pig、Cascading 等。它拥有一个高度可定制的执行架构，因而我们能够在运行时根据与数据和资源相关的实时信息完成动态性能优化。框架本身会自动地决定很多棘手问题，让它能够顺利地正确运行。

使用 Tez，你能够得到良好的性能和开箱即用的效率。Tez 的目标是解决 Hadoop 数据处理领域所面对的一些问题，包括延迟以及执行的复杂性等。Tez 是一个开源的项目，并且已经被 Hive 和 Pig 使用。

---

## 作者简介

**Arun Murthy** 是 Apache Hadoop 下的 MapReduce 项目的领导者，自 2006 年成立以来他一直是 Apache Hadoop 项目的全职贡献者。他是一个长期的提交者，是 Apache Hadoop PMC 的成员，共同保持着目前使用 Apache Hadoop 排序的世界记录。在共同创建 Hortonworks 之前，Arun 在 Yahoo 公司负责所有的 MapReduce 代码以及 42,000+ 台服务器的配置部署工作。

**Bikas Saha** 致力于 Apache Hadoop 项目有一年多了，他是该项目的一个提交者。对于让 Hadoop 在 Windows 系统上运行这方面他是一个关键的提交者，他关注 YARN 和 Hadoop 计算栈。在从事 Hadoop 项目之前，他在 Dryad 分布式数据处理框架上做了大量工作，该框架作为 Microsoft Bing 基础设施的一部分运行在全世界最大的一些集群上。

查看英文原文：[What is Apache Tez?](#)

查看原文：[Apache Tez 是什么？](#)

# 万台规模下的 SDN 控制器集群部署实践

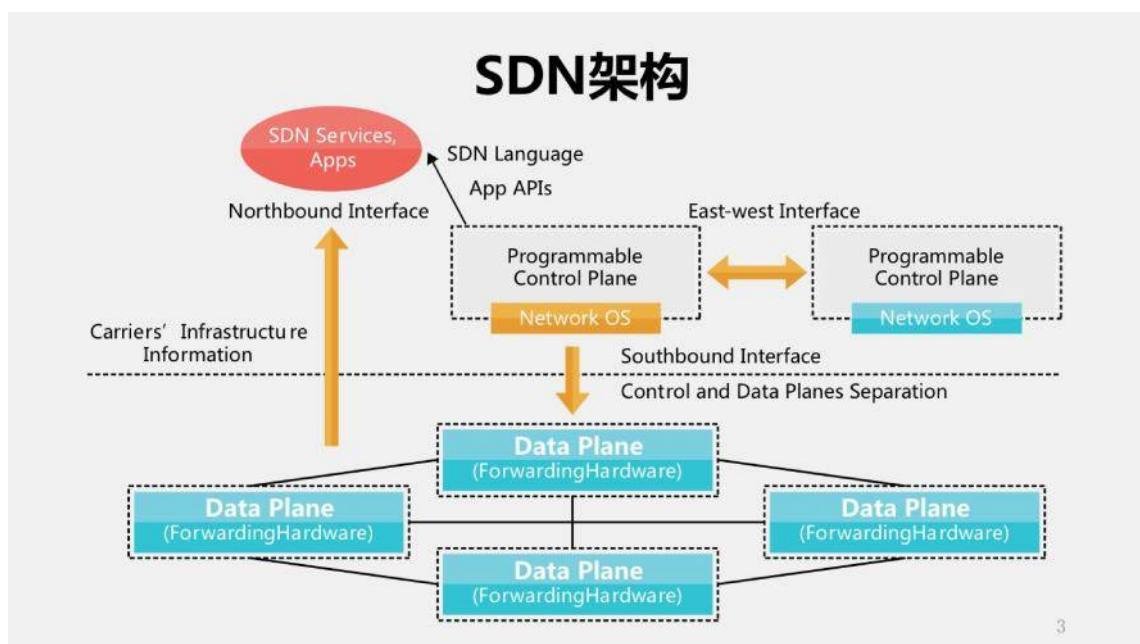
作者 王飓

本文根据华三通信研发副总裁王飓在 2014 年 QCon 上海的主题演讲《SDN 控制器集群中的分布式技术实践》整理而成。

目前在网络世界里，云计算、虚拟化、SDN、NFV 这些话题都非常热。今天借这个机会我跟大家一起来一场 SDN 的深度之旅，从概念一直到实践一直到一些具体的技术。

本次分享分为三个主要部分：

- SDN & NFV 的背景介绍
- SDN 部署的实际案例
- SDN 控制器的集群部署方案



我们首先看一下 SDN。其实 SDN 这个东西已经有好几年了，它强调的是什么？控制平面和数据平面分离，中间是由 OpenFlow 交换机组成的控制器，再往上就是运行在

SDN 之上的服务或者是应用。这里强调两个，控制器和交换机的接口——我们叫做南向接口；另一个是往上的北向接口。

SDN 的核心理念有三个，第一个控制和转发分离，第二个集中控制，第三个开放的 API——可编程、开放的 API 接口。单纯看这三个概念，我们很难理解为什么 SDN 在网络业界现在这么火。这三个概念就能够支撑起 SDN 的成功吗？所以我们要探寻一下 SDN 背后的故事。

## SDN 背后的故事

其实 SDN 在诞生之初，我们这些做网络的人对它不重视，最开始认为就是大学的教授搞出来的实验室里的玩具，并不认为会对产业界有大的影响，可是几年下来以后让我们每个人都大吃一惊，它发展太快了。这个背后有什么呢？

实际上在 SDN 的发展的几年当中有另外一个技术在迅速的发展，铺天盖地来到每个人的身边，就是云计算。说云计算我还想跟大家分享一个小故事，我前几天在公司准备 QCon 的胶片，我们公司的负责保洁的师傅说了，你做云计算啊？我说是啊，你也知道云计算？他说当然知道了，我经常用云计算，那我问他，你都怎么用的？他说上淘宝啊，经常买东西。然后我就问他了，那你知道淘宝应该算什么云吗？其实我问他这句话背后的含义是因为，在我们这个圈子里面把云分为公有云、私有云、混合云，我想问他，结果他的答案非常让我震惊，淘宝你不知道吗？马云啊！所以我就深深的体会到了，起一个好的名字是非常重要的。

刚刚这个只是个笑话，体现了两个问题：第一个就是云计算现在地球人都知道，第二个就是每个人对云计算的理解又是不同的。我给云计算下了一个定义，就是使计算分布在海量的分布式节点上并且保持弹性，这么说可能比较抽象，再说的稍微形象一点就是说使资源池化，在你需要的时候可以按需索取、动态管理。

当人们围绕着按需索取动态管理做文章的时候，什么技术能达到这个要求呢？虚拟化技术。所以现在看云管理平台，OpenStack、CloudStack 也好，都是围绕这三个方面——计算、存储还有网络的虚拟化做文章的。

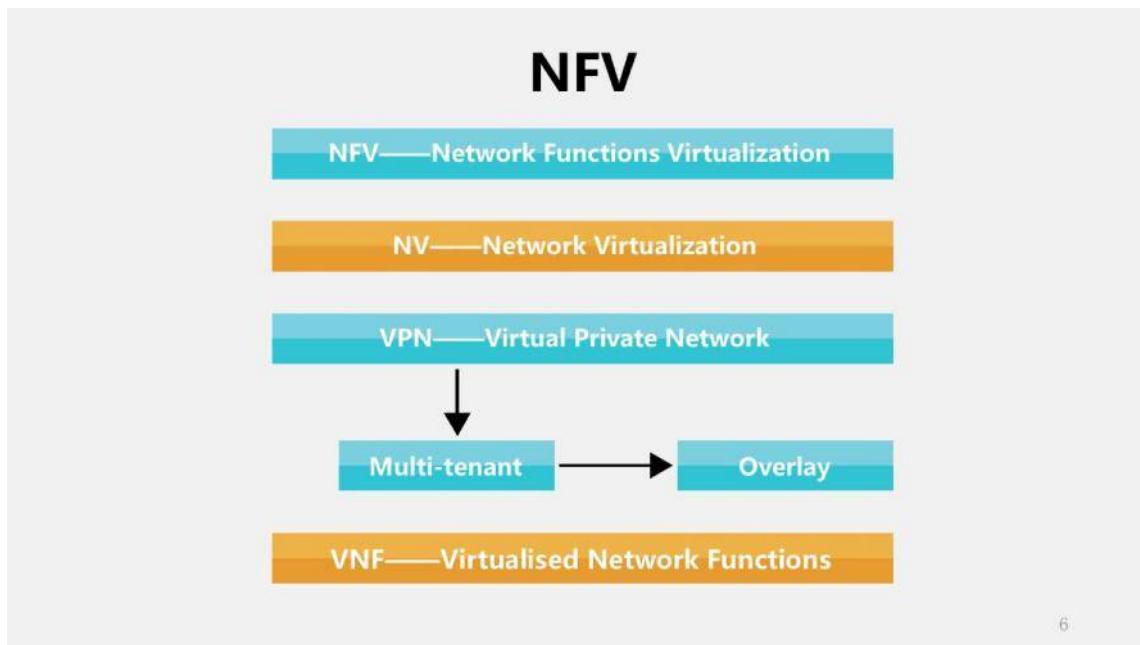
在这股虚拟化的浪潮前面，计算虚拟化发展的最早也发展的最快，网络和存储的虚拟化就相对滞后一点。当人们把目光聚焦到网络虚拟化的时候，人们寻找解决网络虚拟化的方法和工具，这个时候 SDN 就出现在人们的视野里了。

刚刚讲的 SDN 三个理念：控制和转发分离可以使控制层面脱离对网络设备的依赖，可以快速发展；集中控制就很方便对资源进行池化和控制；开放的 API——南向和北向接口——可以催生产业链，推动整个产业的快速发展。

开放的云计算数据中心解决方案都离不开 SDN，从某种程度上讲，是云计算架构里面的基石，再讲另外一个话题，NFV 也是比较热的话题，是网络功能虚拟化，和刚刚我讲的网络虚拟化就差一个字母，但是实际上阐述的是两个不同层面的概念。

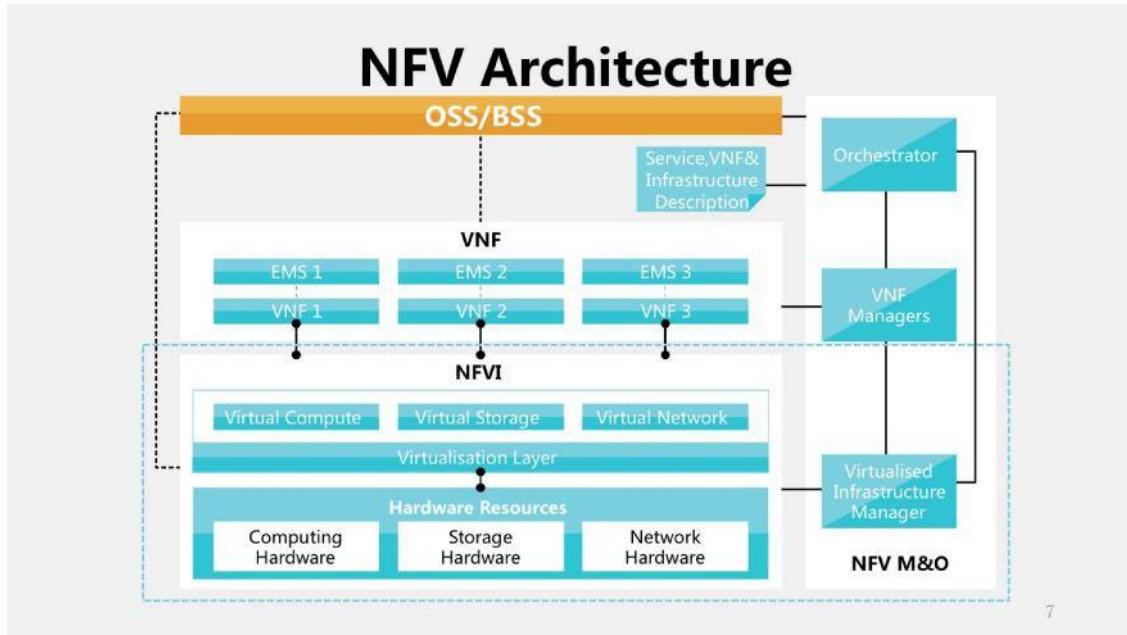
我们讲了云计算需要网络虚拟化，实际上不是今天才有的，像我们做网络的人都知道很久以前人们就有这种网络虚拟化的要求了，不过在那个时候我们管它叫 VPN，虚拟专用网，以前使用的都是在一个公共的网络上虚拟出来一个专用的网络，让使用者以为这个网络就是给我专用的。

那么到了云计算时代，在我们讨论云数据中心的时候，就不再用原先的词了，现在说 multi-tenant，多租户，其实道理是一样的，也是在一个公共的网络里隔离出来各个专用的网络。这个技术是什么？就是 Overlay。数据层面看这两项技术没有本质的区别，都是实现了数据封装，方式就是在网状建立隧道把各个网络隔离开。

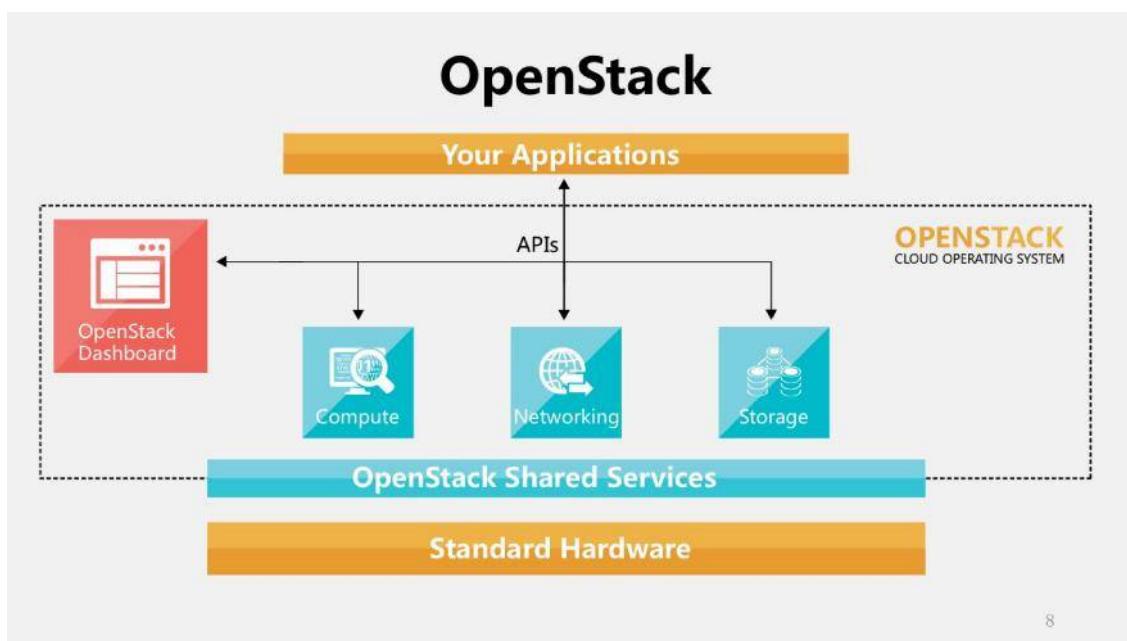


回头看 NFV 这个概念，就有点区别了：NFV 是欧洲电信联盟提出来的。我们知道在运营商的机房里面看到成片的服务器、存储设备、还有大量的不同厂商不同的网络设备，云计算的时代这些运营商也不干了，这么搞太烦了，维护起来成本高，部署起来复杂，新业务上线很慢，他就强调能不能把这个世界搞得干净一点，机房当中只剩下

三个设备：标准的交换机，标准的服务器还有标准的存储设备，除了这些设备，其他通通消失，把所有功能挪到标准的服务器上实现，强调网络功能的虚拟化。



所谓功能的虚拟化就是说把这个功能从传统的网络设备里拿出来。这个是欧洲电信联盟给出的 NFV 的架构图，我们看这个架构图的时候发现，这个下面的部分称为 NFVI，就是基础设施，进行管理和虚拟化，目标是为了在上面提供这些他称之为 VNF 的功能。强调一下，这是一个一个的功能单元，这些功能单元运行在虚拟化出来的虚拟机或容器里。最右边这一块就是整个系统的管理。



我们看这个图可能会感觉很熟悉，这个就是 OpenStack 的架构图。云管理平台玩的就是虚拟化，管理的是计算、存储和网络。对比前后这两张图我们发现，其实 NFV 的架构就是云计算的架构，只不过它所强调的仅仅是运行在云计算上的服务，而不像我们普通的比如 Hadoop 服务为了大数据，或者 WEB 和数据库等等。NFV 的服务都是为网络功能服务的，包括 DHCP 地址分配、NAT 地址转换、防火墙、无线接入、宽带接入、3G 核心网等等，都可以用虚拟的容器实现。所以其实网络功能和我们大家平时所熟悉的这些标准的 APP 或者是这些服务都是一样的，都可以一样的进行虚拟化和云化。



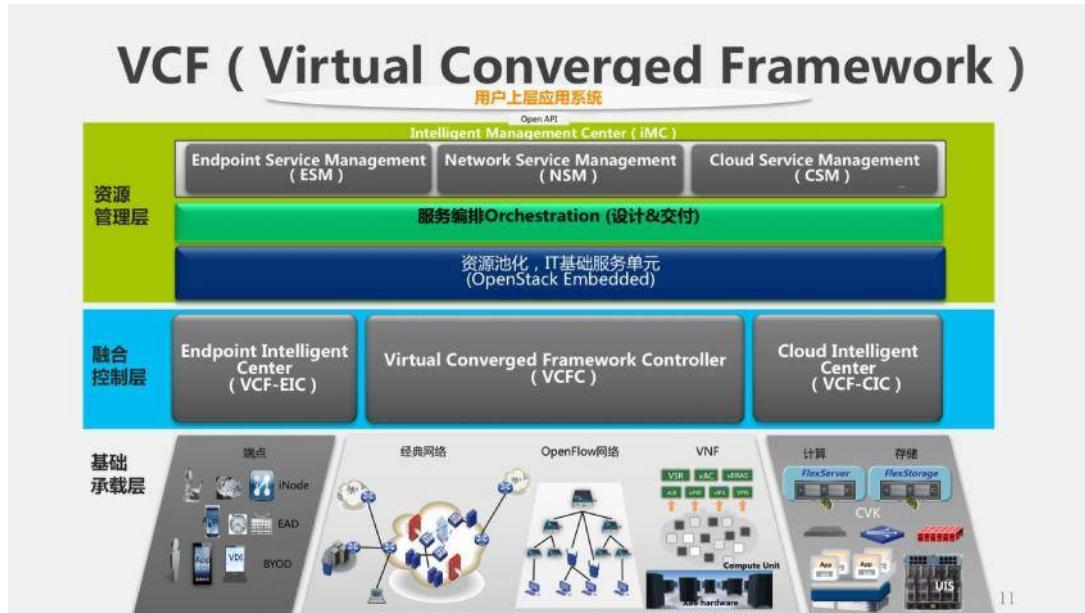
9

所以从这个层面上讲，一个广泛意义上的网络虚拟化会看到几个热门技术：SDN、NFV、Overlay，都是从不同的层面支撑虚拟化——SDN 定义了一种控制和管理的网络架构，Overlay 提供了一种解决数据平面转发和多租户隔离的技术手段，NFV 指出了我的网络功能如何借助这个架构实现虚拟化。这里就有一个循环：网络虚拟化包含了网络功能虚拟化，网络功能虚拟化又依赖于云计算的架构，一旦这个循环形成了，这些新技术在彼此之间不停的碰撞，相互结合也相互竞争，也构成了我们今天这样一个网络世界变革的大时代。

## SDN 的部署案例

下面给大家介绍一下我们在 SDN 领域的实际的案例。

在讲实际的案例之前，想先跟大家分享一下我们公司对未来网络发展的一个理解。我们知道现在 SDN、NFV 很热，但是传统的网络并不会消失，在很长的时间内这些技术会并存。我们强调一个什么概念呢？就是一个虚拟的融合架构。



这个图是一个三横三纵的结构，我们从三纵开始。最右边就是云，解决的是计算存储的问题；最左边是端。有人最近提出来一个概念，说现在的网络是云计算和移动互联网的时代，云计算就是云，而移动互联网就是端。对端的管理，不管是什么样的网络，我们最终检验网络质量的标准是什么？最终用户的体验。如果你的网络解决不好这个问题，你这个网络就很难说是成功的网络——所以中间就是云和网的结合体，就是网络的主体。

我们再横向看三层。底下一层就是基础的设备，包括终端设备、网络设备还有计算存储，包括 NFV 虚拟出来的网络单元也可以当成逻辑的网络设备放这里。第二层就是所谓的融合控制层：我们认为在网络上应该有这样一个层次，既可以管网，也可以管云，也可以管端，这些都需要进行一定的集中控制。最上边的一层就是所谓的资源管理层，在这个里面第一要对所有的资源进行池化，比如 OpenStack 这样的管理平台；在这个之上要提供一个业务编排的系统，把这些逻辑的分散的资源单元串在一起，才能够为用户提供服务；再之上就是针对不同的网络服务提供一些管理组件。

这个 VCF 架构强调每个层次都要对上一层次提供开放的接口，最终对最上面用户的应用提供可编程、可控制的能力。这实际上强调的是什么？就是对端和云中的应用提供了一种自动化的编排和管理的能力。

这么说可能比较抽象，我举个具体的例子，比如说你的手机拿起来了要上网，当你的手机上网的时候，传统的网络就是 AC，无线控制器做 Wifi 认证；但是在我们这样一个融合架构里面，对这个手机进行认证的设备就不再是 AC 控制器了，而是 VCF 的网络控制器。在这个网络控制器对这个手机进行认证、允许上网以后，就知道这个手机是谁的，应该有什么权限，可以获取什么资源，这个时候就要执行一个我们称之为 user profile 的服务模板，执行之后会控制整个网络里面所有的设备，根据这个用户上网这一个动作，它就可以对这个用户所需要的所有的资源进行调整。这在传统的网络里是很难实现的。

另一个例子，在云数据中心一款 APP 上线，一般就是 VM 或者是一个容器的创建。一般情况下在云管理平台里面，容器创建的时候就要分配和指定资源，包括什么资源呢？CPU、内存、硬盘、网络出口带宽、还有这个服务前面要不要加防火墙、是不是大的集群中的成员前面要放负载均衡、要不要给它备份管理等等等，这些功能实际上存在一个叫做 app profile 的文件里。这样在整个云里面，不论是存储还是计算资源还是网络资源都被这个 APP 上线调动了，他会根据你预先编排好的需求，动态对所有的资源进行调整。我们以前做应用的人对网络施加一些控制是很难的，只能对网络管理员提出要求，让他实现；但是现在，我们就具备一种动态管理的能力，这就是这样一个概念带来的一个变革。

所以说，我们认为 VCF 这样的概念实际上就是我们对 SDN 这个概念的发展和补充，是我们认为未来网络发展的趋势。

像这样一个整网融合的方案会比较大，我们真正在商用的时候往往只会使用一部分。比如说我们在给一个数据中心做方案的时候，可能重点关注于你的云和网；如果给一个城域网做方案，可能就是关注网；如果是给园区做网，就是关注端和网的管理。

下面我就给大家讲两个实际的案例。

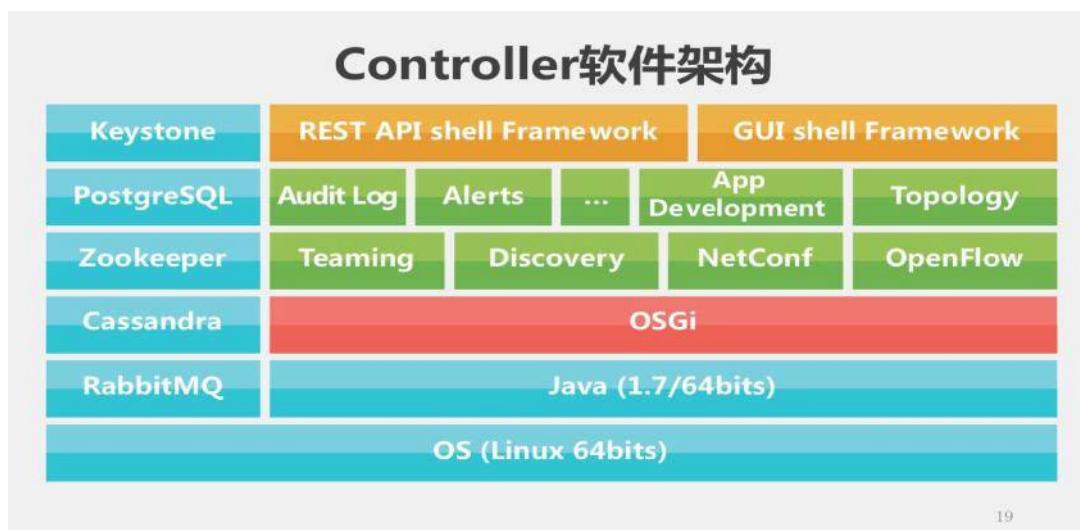
第一个，浙江政务云。这个项目包含两个部分，一个是公有云，一个是政务云，公有云由阿里云承担，政务云由我们公司承建。我们看一下结构，在这个图里面我们会发现，有计算、存储，中间是一个由核心交换机和边缘交换机构成的网络把这些全部连接起来，同时还有网络的管理控制器和云的管理控制器，之上就是 iMC——一个更高层的资源编排和管理软件。上面的 OpenStack 没有直接管理交换机，而是通过往 OpenStack 里面注入插件，把控制功能转给了控制器，包括云控制器和网络控制器，然后再去管理物理的设备。这样有什么好处呢？保留了开源云管理平台 OpenStack 的开放性，第三方应用可以用同一个 API 来做控制；而同时因为使用了专用的控制器，效率会有进一步的提升。

这个专用控制器就是 SDN 和 Overlay 技术的实现，可以对外控制三种网络角色：VxLAN VTEP 控制虚拟化的 vSwitch，VxLAN GW 控制数据中心内的边缘交换机，VxLAN IP GW 控制对外界连接的网关——核心交换机。

Overlay 这个技术有一个特点，就是它初始化的时候，所有节点上的流表是空的。在什么时候才形成转发控制的能力呢？是随着业务的部署形成的。比如说当有一个 VM 想跟另外一个通讯的时候，第一个报文就被 vSwitch 捕获，然后分析一下，就知道应该从哪个虚机到哪个虚机，在源和目的的之间建立一个隧道下发流表，把这个初始的报文返给 vSwitch，这样就过去了。

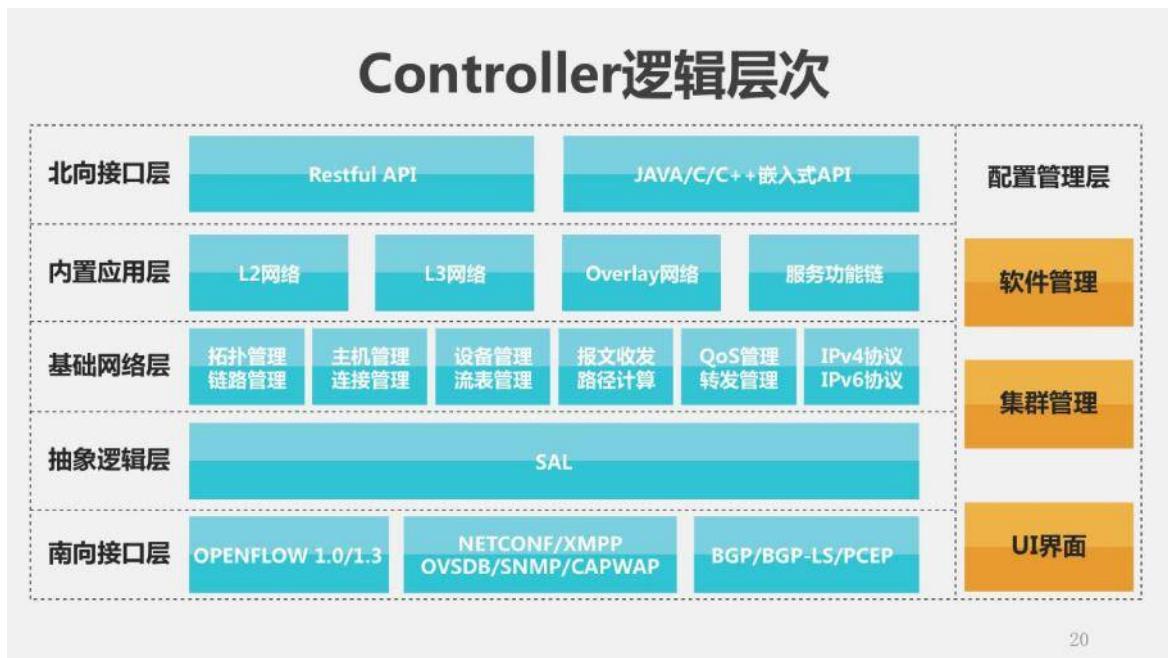
这样处理有甚么好处呢？最大的好处是节约资源。我们知道像这样的数据中心可能有几千或者几万个节点，就是几十万个虚拟机，如果让任意两个 VM 之间都可以通的话，大家算一下要多少的流表——这个资源是有限的。实际上不会所有的 VM 之间都有通讯的要求，根据业务部署可能只有很少数量的 VM 之间才会通信的要求，所以这样的方案很节省流表的资源。这个方案如果说有缺点的话是什么呢，因为它的这个首包上送给了控制器，越到后期在控制器这块的压力就会越来越大。这个问题怎么解决我们后面讲。

再看腾讯的方案。腾讯数据中心的情况是，他们自己已经有云管理平台，有自己的 vSwitch，只是需要我们的物理交换机和控制器。这个方案展开一看大家会发现跟我刚刚讲的这个浙江政务云的方案其实是很类似的，也是 SDN 加 Overlay 的方案。只不过在这个方案里面，第一，不是所有的设备都是我们的，所以需要我们在我们的控制器上面有一些东西跟腾讯的云管理平台进行对接；第二，就是规模的问题，腾讯让我们建立这样的数据中心到什么规模呢？物理的服务器一万五千台。这给我们整个管理带来了很多的挑战，我们怎么才能部署控制器管理一万五千台的服务器，几十万的虚机？下面讲一下我们这个集群管理的部署方案以及具体的优化。



## SDN Controller 集群部署方案以及优化

讲解控制器部署之前，我们花一点时间进入控制器软件的内部，看看这个 Controller 的软件架构。可以看到我们也用一些开源的工具，然后之后呢还有一些各种层面的模块。我们去看一下具体的逻辑图，可能看的更清楚一点。



20

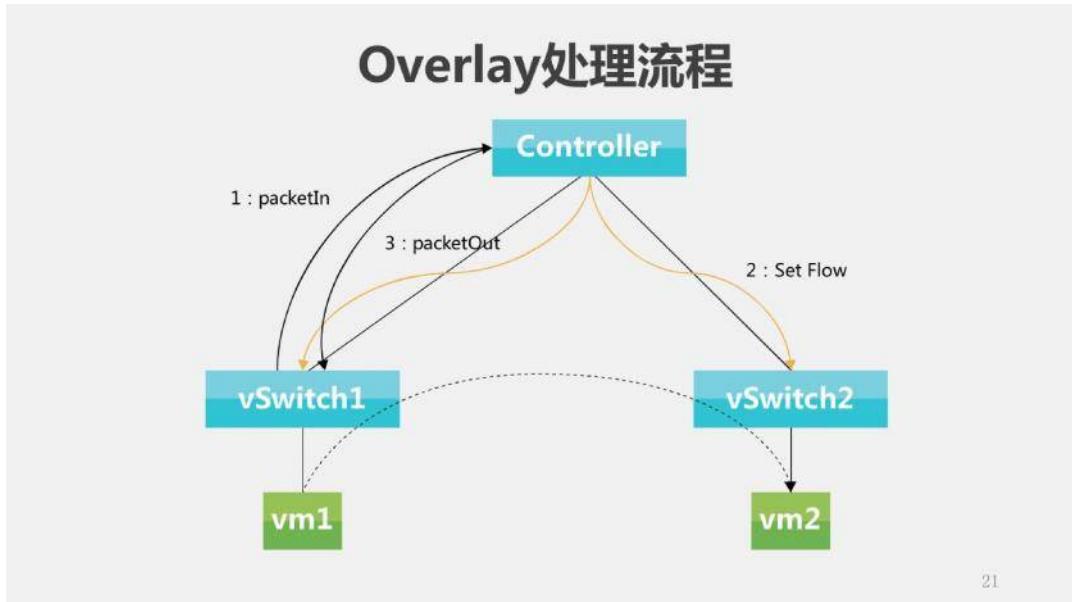
我们把控制器分成不同的层次：最下面我们称之为南向接口层，有 OpenFlow、NetConf/XMPP、BGP 等等不同的数据协议，这是因为控制器往下要管理不同的节点，这些不同的角色（vSwitch 或物理交换机）使用的协议不一样。

第二层是 SAL 适配层，这一层屏蔽了不同的厂商/不同的设备对南向提供接口的差别，让上层的模块运行起来可以仅仅针对他关心的业务处理，而不用关心不同厂商的 API 有什么差别。

再往上就是基础的网络功能模块，这一块没什么说的。再往上就是内置应用。在 SDN 里面有两种应用，一种是内置的应用——就运行在 SDN 的控制器上，还有一种外置的应用——在上面。我们看到这里有 Overlay 模块：最关键的计算都是由这个 Overlay 模块完成的。

再往上就是北向接口层了，就是可编程的控制器要对外提供一个良好的编程接口。

还有一部分就是管理层，有软件的管理、软件自身的升级、增加模块，还有生命周期管理、集群的管理，还有一些 UI 的界面。



讲完这个层次图以后，回到刚刚的问题上。我们 Overlay 的过程——送给控制器，下发流表把这个包返回给 vSwitch 进行转发，这个是一次首包上送。这个方案所造成的问题就是对控制器的计算能力提出挑战。所以我们在这个方案里面重点优化首包上送的处理能力，对刚刚的结构不停的优化、进行重构。

最后我们做到什么性能呢？标准的 Intel i7 4 核处理器上，可以做到 500k 的处理能力。在这个基于 Java 的架构上，我想再做出质的提升恐怕就很困难了。

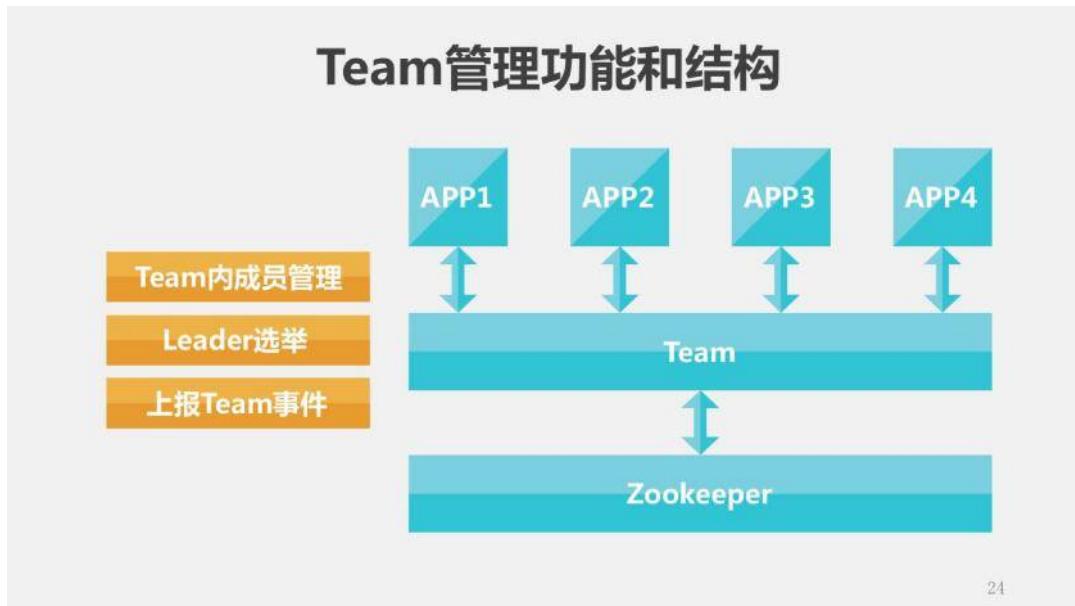
那么这一个控制器能管理多大的网络？瓶颈在首包上送的能力，我们可以计算一下：一个服务器有 1 个 vSwitch，跑 20-30 个 VM，每秒大概可以产生 500 以上的新流，就是每秒有 500 次跟一个新的、不同的设备通信。那么用我们刚刚的首包上送一除就知道，500K 的 TPS 性能，一个控制器大概可以管理一千个 Host；当一个数据中心规模在 15k 的时候，单节点控制器肯定搞不定了，就需要控制器集群。

我们把所有的控制器就称之为一个团队（team），一部分成员是领导者（leader），一部分是成员（member）。Leader 对上提供北向的访问接口，负责对 cluster 进行管理；Member 就负责管理控制交换机，连接交换机的方式就是刚刚讲的南向接口。

单一的节点可能会不安全或者是不可靠，所以就提供了另外一个东西就是 Region。我们把所有的 leader 放在一个 Region 里，作为主集群，其他的作为备份，这样就保持这个集群有一个持续的不间断的对外提供北向接口的能力。下面这些 member 划了一个一个 Region，一个 Region 中有多个 Member，Switch 要同时连接到一个 Region 中的所有 Member 上，并选取一个作为主。这样的好处是什么呢？一个 switch 有多个

member，如果我的主域宕掉了，这个 switch 发现了以后就可以从剩余的里面选择一个新的，备变为主，这样就可以提供一个不间断的服务能力。

我们拿两台服务器做主，然后划分 15 个 Region，一个控制器可以管理一千台的服务器，15 个正好是 15000 台。总的来说就是对控制器进行分层的设计，让 leader 提供向北接口，member 提供向南接口。



24

简单介绍一下我们实现这个集群采用的技术。Team 管理功能，是在 Zookeeper 之上封装的，这个 Team 实现了成员管理、leader 选举、上报 Team 事件，具体的方式是很标准的 Zookeeper 使用方式，这个就不多说了。

那么还有一个重要的问题，不是说你的成员加入集群就完事儿了，我刚刚讲腾讯的方案的时候，腾讯的云管理平台上有大量的 VM 的信息，这些需要你做一个模块抓取过来，要在你所有的控制器之间共享，所以说就需要有一些数据在所有的控制器间共享，也就是 HA。按照我们做网络的习惯，我们把 HA 分成两种功能，一个是实时备份，一个是批量备份，目标就是希望这个 HA 系统对上述的 APP 是不可见的，具体看一下实现。

第一个就是 BUS，它提供通讯的通道，当你写入一个数据的时候，就在主那里创建一个单元，发现这个节点发生变化，把这个单元读出来，这个数据就传过去了。

KeyStore，实现了一个非常简单的数据库功能，采用 Key-Value 机制，不支持范围查找，只提供了设置和获取接口，没有通知接口。有的同学会问了，说你们跟

Zookeeper 干上了是吧？那我们做开发的人都知道，当我们熟悉一个工具的时候，就会很自然的重复使用，尽量用熟为止。

实际使用当中的实时备份过程就是这样的，很简单，集群中某个成员业务数据变化时，发送 bus 消息通知其他成员，同时将本成员的运行数据以 key&value 的形式保存在 KeyStore 中。

批量备份就是当新的控制节点加入时，KeyStore 就会自动将其他节点的数据备份到本地，App 需要先从 KeyStore 中恢复数据，当恢复完成后，再开始接收 bus 消息。做 keyStore 数据恢复时，要求 bus 可以从批备开始的时间点开始缓存 bus 消息，等恢复完成后补报这些 bus 消息，这样就可以保证了最终数据的同步。

今天跟大家分享的话题就到这里，谢谢大家！

---

## 嘉宾简介

王飓，华三研发副总裁，从事数据通讯设备软件开发长达 14 年，作为资深的网络协议专家和软件系统架构师，熟悉多个层面的数据通讯协议，擅长做通信协议设计以及实现，对嵌入式系统和复杂软件系统设计，以及对实时系统的性能优化有着十分丰富的经验。此外，对网络安全有着比较深入的研究，对各种网络攻击和防护有着丰富的经验。近年来开始关注并投入 SDN 相关领域的研究和开发。对 OpenStack、OpenDaylight、OpenVswitch、NFV 等都有一定的研究，对云计算时代的网络通信有着深刻的理解。

在这个云计算的时代，很多传统的通信技术都会经历一个痛苦的解构重建的过程，如何把已有的网络经验融合到现在的 SDN 世界当中，充分利用历史的积累，是他目前最为关心的问题。

---

感谢杨赛对本文内容的整理。

查看原文：[万台规模下的 SDN 控制器集群部署实践](#)

# OS X 和 iOS 中的多线程技术

作者 郭麟

## 多线程技术

我们为何需要多线程呢？多线程其实是为了实现并发执行，而且线程是并发执行多个代码路径的多种技术之中比较轻量级的一种（对应较重的实现是多进程）。

在单核 CPU 时代，支持多线程的操作系统会通过分配 CPU 计算时间，来实现软件层面的多线程。创建线程，线程间切换都是有成本开销的。但由于多线程可以避免阻塞所造成的 CPU 计算时间浪费，所以多线程所带来的开销成本总体看来是值得的。任务一般都可以被拆分成多个子任务，如果一个子任务发生了阻塞，计算时间就可以分配给其他子任务。这样就提高了 CPU 的利用率。

在多核 CPU 时代，就更好理解了。由于硬件上就支持多线程技术，就可以让多个线程真正同时地运行。如果任务能够被拆分，各个子任务就能并行地在 CPU 上运行，这就能够显著加快运行速度。

总结说来，多线程的目的是，通过并发执行提高 CPU 的使用效率，进而提供程序运行效率。

OS X 和 iOS 是多线程操作系统，它们追随 UNIX 系统使用了 POSIX 线程模型。OS X 和 iOS 都提供了一套底层的 C 语言 POSIX 线程 API 来创建和管理线程。但实际应用开发中，除非需要跨平台，我们并不常直接使用 POSIX 线程 API，而是使用系统或语言提供的其他一些更为简单的方案，下一节中会讨论它们。

## Objective-C 中实现多线程

performSelectors

NSObject 提供了以 `performSelector` 为前缀的一系列方法。它们可以让用户在指定线程中，或者立即，或者延迟执行某个方法调用。这个方法给了用户实现多线程编程最简单的方法。下面有一些例子：

在当前线程中执行方法：

```
- (void)performSelector:(SEL)aSelector withObject:(id)anArgument  
afterDelay:(NSTimeInterval)delay  
  
- (void)performSelector:(SEL)aSelector withObject:(id)anArgument afterDelay:  
(NSTimeInterval)delay inModes:(NSArray *)modes
```

在指定线程中执行方法：

```
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thread  
withObject:(id)arg waitUntilDone:(BOOL)wait  
  
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thread  
withObject:  
(id)arg waitUntilDone:(BOOL)wait modes:(NSArray *)array
```

在主线程中执行方法：

```
- (void)performSelectorOnMainThread: (SEL)selector withObject:(id)argument  
waitUntilDone:(BOOL)wait  
  
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(id)arg  
waitUntilDone:(BOOL)wait modes:(NSArray *)array
```

在后台线程中执行方法：

```
- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg
```

这一系列方法简单易用，但只提供了有限的几个选择：指定执行的方法（但传入方法的参数数量有限制）；指定是在当前线程，还是在主线程，还是在后台线程执行；指定是否需要阻塞当前线程等待结果。

例如，以下代码使得方法 foo: 在一个新的后台线程执行，并传入了 object 参数：

```
SEL selector = @selector(foo:);  
[self performSelectorInBackground:selector withObject:object];
```

以下代码使得 updateUI 方法在主线程内得到执行，并且当前线程会被阻塞，直到主线程执行完该函数：

```
[self performSelectorOnMainThread:@selector(updateUI) withObject:nil  
waitUntilDone:YES];
```

## NSThread

NSThread 是 OS X 和 iOS 都提供的一个线程对象，它是线程的一个轻量级实现。在执行一些轻量级的简单任务时，NSThread 很有用，但用户仍然需要自己管理线程生命周期。

期，进行线程间同步。比如，线程状态，依赖性，线程间同步等线程相关的主题 NSThread 都没有涉及。比如，涉及到线程间同步仍然需要配合使用 NSLock，NSCondition 或者 @synchronized。所以，遇到复杂任务时，轻量级的 NSThread 可能并不合适。

提供一个模拟多线程运作的简单例子：两个人同时一起到烤箱抢面包。我们启动两个线程，来代表两个人。由于烤箱门比较小，同时只能有一个人去拿面包。由于 NSThread 不处理线程同步，所以为了模拟这个过程，你还需要一把线程锁（即类型为 NSLock 的实例变量 \_lock）。在后面的 run 方法中会用到这把线程锁：

```
_lock = [[NSLock alloc] init];

NSThread *geroge = [[NSThread alloc] initWithTarget:self selector:@selector(run) object:nil];
[geroge setName:@"Geroge"];
[geroge start];

NSThread *totty = [[NSThread alloc] initWithTarget:self selector:@selector(run) object:nil];
[totty setName:@"Totty"];
[totty start];
```

受到线程锁保护的拿面包过程可以用下面的 run 方法表示：

```
- (void)run {
    while (TRUE) {

        [_lock lock];
        if(_cake > 0){
            [NSThread sleepForTimeInterval:0.5];
            _cake--;
            _occupied = kSum - _cake;
            NSLog(@"Taken by %@\nCurrent free:%ld, occupied:%ld",
                  [[NSThread currentThread] name], _cake, _occupied);

        }
        [_lock unlock];
    }
}
```

## NSOperation

NSOperation 做的事情比 NSThread 更多一些。通过继承 NSOperation，可以使子类获得一些线程相关的特性，进而可以安全地管理线程生命周期。比如，以线程安全的方式建立状态，取消线程。配合 NSOperationQueue，可以控制线程间的优先级和依赖性。这就给出了一套线程管理的基本方法。

NSOperation 代表了一个独立的计算单元。一般，我们会把计算任务封装进 NSOperation 这个对象。NSOperation 是抽象类，但同时也提供了两个可以直接使用的实体子类：NSInvocationOperation 和 NSBlockOperation。NSInvocationOperation 用于将计算任务封装进方法，NSBlockOperation 用于将计算任务封装进 block。

NSOperationQueue 则用于执行计算任务，管理计算任务的优先级，处理计算任务之间的依赖性。NSOperation 被添加到 NSOperationQueue 中之后，队列会按优先级和进入顺序调度任务，NSOperation 对象会被自动执行。

仍然使用上一节 NSThread 中的模拟两人抢面包的例子。由于计算任务没有变化，所以 run 方法并不改变。但这里需要使用 NSOperation 和 NSOperationQueue 来代表两个抢面包的人，并给予他们不同的优先级。由于 NSOperation 也不处理线程间同步问题，所以你仍然需要一把在 run 方法中会用到的线程锁：

```
_lock = [[NSLock alloc] init];

NSInvocationOperation *geroge = [[NSInvocationOperation
alloc] initWithTarget:self
selector:@selector(run:) object:@"Geroge"];
geroge.queuePriority = NSOperationQueuePriorityHigh;

NSInvocationOperation *operationTwo = [[NSInvocationOperation
alloc] initWithTarget:self
selector:@selector(run:) object:@"Totty"];
totty.queuePriority = NSOperationQueuePriorityLow;

NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue setMaxConcurrentOperationCount:2];
[queue addOperation:geroge];
[queue addOperation:totty];
```

NSOperation 提供以下任务优先级，以这些优先级设置变量 queuePriority 即可加快或者推迟操作的执行：

- NSOperationQueuePriorityVeryHigh
- NSOperationQueuePriorityHigh
- NSOperationQueuePriorityNormal
- NSOperationQueuePriorityLow
- NSOperationQueuePriorityVeryLow

NSOperation 使用状态机模型来表示状态。通常，你可以使用 KVO (Key-Value Observing) 观察任务的执行状态。这是其他多线程工具所不具备的功能。NSOperation 提供以下状态：

- isReady
- isExecuting
- isFinished

NSOperation 对象之间的依赖性可以用如下代码表示：

```
[refreshUIOperation addDependency:requestDataOperation];
[operationQueue addOperation:requestDataOperation];
[operationQueue addOperation:refreshUIOperation];
```

除非 requestDataOperation 的状态 isFinished 返回 YES，不然 refreshUIOperation 这个操作不会开始。

NSOperation 还有一个非常有用功能，就是“取消”。这是其他多线程工具（包括后面要讲到的 GCD）都没有的。调用 NSOperation 的 cancel: 方法即可取消该任务。当你知道这个任务没有必要再执行下去时，尽早安全地取消它将有利于节省系统资源。

## GCD

GCD (Grand Central Dispatch) 是 Apple 公司为了提高 OS X 和 iOS 系统在多核处理器上运行并行代码的能力而开发的一系列相关技术，它提供了对线程的高级抽象。GCD 是一整套技术，包含了语言级别的新功能，运行时库，系统级别的优化，这些一起为并发代码的执行提供了系统级别的广泛优化。所以，GCD 也是 Apple 推荐的多线程编程工具。

GCD 是系统层面的技术，除了可以被系统级应用使用，也可以被更普通的高级应用使用。使用 GCD 之后，应用就可以轻松地在多核系统上高效运行并发代码，而不用考虑繁琐的底层问题。GCD 在系统层面工作，能很好地满足所有应用的并行运行需求，将可用系统资源平衡地分配给它们。

GCD 提供了一套纯 C API。但是，它提供的 API 简单易用并且有功能强大的任务管理和多线程编程能力。GCD 需要和 blocks (Objective-C 的闭包) 配合使用。block 是 GCD 执行单元。GCD 的任务需要被拆解到 block 中。block 被排入 GCD 的分发队列，GCD 会为你排期运行。GCD 创建，重用，销毁线程，基于系统资源以它认为合适的方式运行每个队列。所以，用户需要关心的细节并不多。

GCD 的使用也很简单，假设抢面包是个耗时操作，前面例子中的 Geroge 和 Totty 的工作都可以实现如下：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^{
    // 并发队列中做耗时操作
    while (TRUE) {
        if(_cake > 0) {
            // 耗时操作
        }
    }
});
```

```
[NSThread sleepForTimeInterval:0.5];
_cake--;
_occupied = kSum - _cake;
} else {
    break;
}
}

// 主队列中刷新界面
dispatch_async(dispatch_get_main_queue(), ^{
    [self updateUI];
});
});
```

## GCD 分发队列

GCD 分发队列是执行任务的有力工具。使用分发队列，你可以异步或者阻塞执行任意多个 block 的代码。你可以使用分发队列来执行几乎任何线程任务。GCD 提供了简单易用的接口。

在 GCD 中存在三种队列。

### 1 串行分发队列（Serial dispatch queue）

串行分发队列又被称为私有分发队列，按顺序执行队列中的任务，且同一时间只执行一个任务。串行分发队列常用于实现同步锁。下面代码创建了一个串行分发队列：

```
dispatch_queue_t serialQueue = dispatch_queue_create("com.example.MyQueue",
NULL);
```

### 2 并发分发队列（Concurrent dispatch queue）

串行分发队列又被称为全局分发队列，也按顺序执行队列中的任务，但是顺序开始的多个任务会并发同时执行。并发分发队列常用于管理并发任务。下面代码创建了一个并发分发队列：

```
dispatch_queue_t concurrentQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

### 3 主分发队列（Main dispatch queue）

主分发队列是一个全局唯一的特殊的串行分发队列。队列中的任务会被在应用的主线程中执行。主分发队列可以用于执行 UI 相关的操作。取得主分发队列的方法：

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
```

## GCD 任务执行方式

GCD 中有两种任务执行方式：

- 异步执行, `dispatch_async`, 意味将任务放入队列之后, 主线程不会等待 `block` 的返回结果, 而是立即继续执行下去。
- 阻塞执行, `dispatch_sync`, 意味将任务放入队列之后, 主线程被阻塞, 需要等待 `block` 的执行结果返回, 才能继续执行下去。

## GCD 的其他主题

GCD 有着丰富的功能, 比如分发组 (dispatch group), 信号 (semaphores), 分发栅栏 (dispatch barrier), 分发源 (dispatch source) 等等。这些可以用于完成更复杂的多线程任务。详细可以查阅 Apple 关于 GCD 的[文档](#)。

## 使用建议

在能够使用 GCD 的地方, 尽量使用 GCD。

Apple 公司宣称其在 GCD 技术中为更好地利用多核硬件系统做了很多的优化。所以, 在性能方面 GCD 是不用担心的。而且 GCD 也提供了相当丰富的 API, 几乎可以完成绝大部分线程相关的编程任务。所以, 在多线程相关主题的编程中, GCD 应该是首选。下面举一些可以推荐使用 GCD 的实际例子。

### 1. 使用 GCD 的 `dispatch queue` 实现同步锁

同步锁的实现方案有不少, 比如, 如果仅仅是想对某个实例变量的读写操作加锁, 可以使用属性 (property) 的 `atomic` 参数, 对于一段代码加锁可以使用 `@synchronized` 块, 或者 `NSLock`。

`@synchronized` 和 `NSLock` 实现的同步锁:

```
// Method 1
- (void)synchronizedMethod {
    @synchronized(self) {
        // safe
    }
}

// Method 2
_lock = [[NSLock alloc] init];
```

```
- (void)synchronizedMethod {
    [_lock lock];
    // Safe
    [_lock unlock];
}
```

@synchronized 一般会以 self 为同步对象。重复调用 @synchronized(self) 是很危险的。如果多个属性这么做，每一个属性将会被和其它所有属性同步，这可能并不是你所希望的，更好的方法是每个属性的锁都是相互独立的。

另一种方法是使用 NSLock 实现同步锁，这个方法不错，但是缺点是在极端环境下同步块可能会导致锁死，而且这种情况下处理锁死状态会有麻烦。

一个替代方法是使用 GCD 的分发队列。将读和写分发到相同并发队列中，这样读操作会是并发的，多个线程可以同时执行写操作；而对于写操作，以分发栅栏（dispatch barrier）保证同时只有一个线程可以执行写操作，并且由于写操作无需返回，写操作还是异步马上返回的。这样，就得到了一个高效且线程安全的锁。代码看起来会像这样：

```
_syncQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

- (NSInteger)cake {
    __block NSInteger localCake;
    dispatch_sync(_syncQueue, ^{
        localCake = _cake;
    });
    return localCake;
}

- (void)setCake:(NSInteger)cake {
    dispatch_barrier_async(_syncQueue, ^{
        _cake = cake;
    });
}
```

简单而言，上面的代码可以使读操作被竞争执行；写操作被互斥执行，并且异步返回。使用 GCD 实现的这个同步锁应该是效率最优且最安全的。

## 2. 使用 GCD 替代 performSelector 系列方法

NSObject 的 performSelector 系列方法有很多限制。传给要执行的方法的参数的数量是有限制的，也没法方法保证能正确地取得要执行的方法的返回值。这些限制在使用 block 的 GCD 中都不存在。

下面是使用 GCD 替代 performSelector 的例子。使用 performSelector 系列方法：

```
[self performSelector:@selector(cake)
    withObject:nil
    afterDelay:5.0];
```

使用 GCD 完成相同的事情：

```
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(5.0 *
NSEC_PER_SEC));
dispatch_after(time, dispatch_get_main_queue(), ^(void){
    [self cake];
});
```

### 3. 使用 `dispatch_once` 实现线程安全单一执行要求

线程安全单一执行典型例子是单例，GCD 的 `dispatch_once` 能够保证传入的 block 被线程安全地唯一执行：

```
+ (id)sharedInstance {
    static AdivseDemoController *sharedInstance = nil;
    static dispatch_once_t onceToken = @“token”;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

这是现在 Objective-C 中实现单例较为推荐的一种方法。

在需要更细粒度控制线程时，考虑 NSOperation。

GCD 虽然在很多地方值得提倡，但并不是任务管理和多线程地唯一解决方案，并不是说所有的地方都应该使用 GCD。GCD 是一个纯 C API，NSOperation 是 Objective-C 类，在一些地方对象编程是有优势的。NSOperation 也提供了一些 GCD 无法实现，或者 GCD 所没有的功能。

以下是你需要考虑使用 NSOperation 的一些理由：

- 当你需要取消线程任务时，GCD 无法提供取消任务的操作。而 NSOperation 提供了取消任务的操作。
- 当你需要更细的粒度地观察任务改变了状态时，由于 NSOperation 是一个对象，比较 GCD 使用的 block 而言，通过对 NSOperation 对象进行键值观察（KVO）能很容易观察到任务的状态改变。
- 当你需要重用线程任务时，NSOperation 作为一个普通的 Objective-C 对象，可以存储任何信息。对象就是为重用而设计的，这时，NSOperation 比 GCD 使用的 block 要更方便。

## 总结

OS X 和 iOS 系统提供了丰富的多线程工具。这些工具中，最新最现代的是 GCD（Grand Central Dispatch）。GCD 也是 Apple 公司推荐的多线程解决方案。所以，对于多线程技术的选择，总结下来有这两条建议：

- 能够使用 GCD 的地方，尽量使用 GCD。
- 在需要更细粒度控制线程时，考虑 NSOperation。

## 参考文档

- [Grand Central Dispatch \(GCD\) Reference](#)
  - [Concurrency Programming Guide](#)
  - [Effective Objective-C 2.0](#)
- 

查看原文：[OS X 和 iOS 中的多线程技术](#)

# Kubernetes 系统架构简介

作者 杨章显

*Together we will ensure that Kubernetes is a strong and open container management framework for any application and in any environment, whether in a private, public or hybrid cloud.*

——Urs Hözle, Google

## 1. 前言

Kubernetes 作为 Docker 生态圈中重要一员，是 Google 多年大规模容器管理技术的开源版本，是产线实践经验的最佳表现[G1]。如 Urs Hözle 所说，无论是公有云还是私有云甚至混合云，Kubernetes 将作为一个为任何应用，任何环境的容器管理框架无处不在。正因为如此，目前受到各大巨头及初创公司的青睐，如 Microsoft、VMWare、Red Hat、CoreOS、Mesos 等，纷纷加入给 Kubernetes 贡献代码。随着 Kubernetes 社区及各大厂商的不断改进、发展，Kubernetes 将成为容器管理领域的领导者。

接下来我们会用一系列文章逐一探索 Kubernetes 是什么、能做什么以及怎么做。

## 2. 什么是 Kubernetes

Kubernetes 是 Google 开源的容器集群管理系统，其提供应用部署、维护、扩展机制等功能，利用 Kubernetes 能方便地管理跨机器运行容器化的应用，其主要功能如下：

- 使用 Docker 对应用程序包装（package）、实例化（instantiate）、运行(run）。
- 以集群的方式运行、管理跨机器的容器。
- 解决 Docker 跨机器容器之间的通讯问题。
- Kubernetes 的自我修复机制使得容器集群总是运行在用户期望的状态。

当前 Kubernetes 支持 GCE、vSphere、CoreOS、OpenShift、Azure 等平台，除此之外，也可以直接运行在物理机上。

接下来本文主要从以下几方面阐述 Kubernetes：

- Kubernetes 的主要概念。
- Kubernetes 的构件，包括 Master 组件、Kubelet、Proxy 的详细介绍。

## 3. Kubernetes 主要概念

### 3.1. Pods

Pod 是 Kubernetes 的基本操作单元，把相关的一个或多个容器构成一个 Pod，通常 Pod 里的容器运行相同的应用。Pod 包含的容器运行在同一个 Minion(Host)上，看作一个统一管理单元，共享相同的 volumes 和 network namespace/IP 和 Port 空间。

### 3.2. Services

Services 也是 Kubernetes 的基本操作单元，是真实应用服务的抽象，每一个服务后面都有很多对应的容器来支持，通过 Proxy 的 port 和服务 selector 决定服务请求传递给后端提供服务的容器，对外表现为一个单一访问接口，外部不需要了解后端如何运行，这给扩展或维护后端带来很大的好处。

### 3.3. Replication Controllers

Replication Controller 确保任何时候 Kubernetes 集群中有指定数量的 pod 副本(replicas)在运行，如果少于指定数量的 pod 副本(replicas)，Replication Controller 会启动新的 Container，反之会杀死多余的以保证数量不变。Replication Controller 使用预先定义的 pod 模板创建 pods，一旦创建成功，pod 模板和创建的 pods 没有任何关联，可以修改 pod 模板而不会对已创建 pods 有任何影响，也可以直接更新通过 Replication Controller 创建的 pods。对于利用 pod 模板创建的 pods，Replication Controller 根据 label selector 来关联，通过修改 pods 的 label 可以删除对应的 pods。Replication Controller 主要有如下用法。

#### 1) Rescheduling

如上所述，Replication Controller 会确保 Kubernetes 集群中指定的 pod 副本(replicas)在运行，即使在节点出错时。

#### 2) Scaling

通过修改 Replication Controller 的副本(replicas)数量来水平扩展或者缩小运行的 pods。

#### 3) Rolling updates

Replication Controller 的设计原则使得可以一个一个地替换 pods 来 rolling updates 服务。

#### 4) Multiple release tracks

如果需要在系统中运行 multiple release 的服务，Replication Controller 使用 labels 来区分 multiple release tracks。

#### 3.4. Labels

Labels 是用于区分 Pod、Service、Replication Controller 的 key/value 键值对，Pod、Service、Replication Controller 可以有多个 label，但是每个 label 的 key 只能对应一个 value。Labels 是 Service 和 Replication Controller 运行的基础，为了将访问 Service 的请求转发给后端提供服务的多个容器，正是通过标识容器的 labels 来选择正确的容器。同样，Replication Controller 也使用 labels 来管理通过 pod 模板创建的一组容器，这样 Replication Controller 可以更加容易，方便地管理多个容器，无论有多少容器。

### 4. Kubernetes 构件

Kubernetes 整体框架如图 1 所示，主要包括 kubecfg、Master API Server、Kubelet、Minion(Host)以及 Proxy。

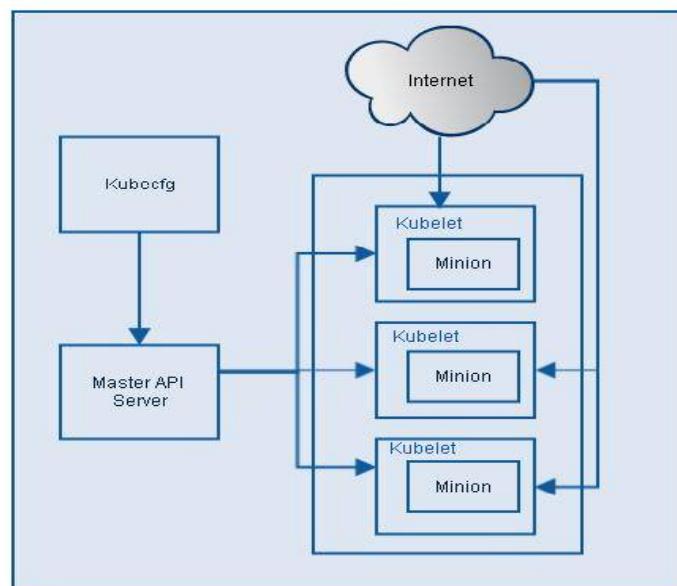


图 1 Kubernetes High Level 构件

## 4.1. Master

Master 定义了 Kubernetes 集群 Master/API Server 的主要声明，包括 Pod Registry、Controller Registry、Service Registry、Endpoint Registry、Minion Registry、Binding Registry、RESTStorage 以及 Client，是 client(Kubecfg)调用 Kubernetes API，管理 Kubernetes 主要构件 Pods、Services、Minions、容器的入口。Master 由 API Server、Scheduler 以及 Registry 等组成。从下图 3-2 可知 Master 的工作流主要分以下步骤。

- a. Kubecfg 将特定的请求，比如创建 Pod，发送给 Kubernetes Client。
- b. Kubernetes Client 将请求发送给 API server。
- c. API Server 根据请求的类型，比如创建 Pod 时 storage 类型是 pods，然后依此选择何种 REST Storage API 对请求作出处理。
- d. REST Storage API 对的请求作相应的处理。
- e. 将处理的结果存入高可用键值存储系统 Etcd 中。
- f. 在 API Server 响应 Kubecfg 的请求后，Scheduler 会根据 Kubernetes Client 获取集群中运行 Pod 及 Minion 信息。
- g. 依据从 Kubernetes Client 获取的信息，Scheduler 将未分发的 Pod 分发到可用的 Minion 节点上。

下面是 Master 的主要构件的详细介绍。

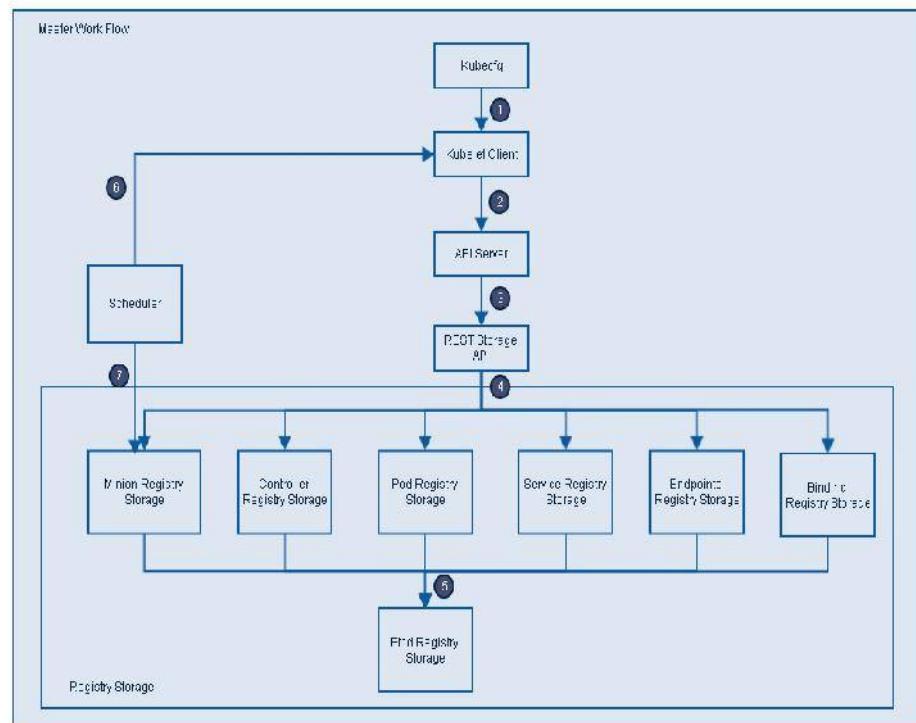


图 2 Master 主要构件及工作流

#### 4.1.1. Minion Registry

Minion Registry 负责跟踪 Kubernetes 集群中有多少 Minion(Host)。Kubernetes 封装 Minion Registry 成实现 Kubernetes API Server 的 RESTful API 接口 REST，通过这些 API，我们可以对 Minion Registry 做 Create、Get、List、Delete 操作，由于 Minion 只能被创建或删除，所以不支持 Update 操作，并把 Minion 的相关配置信息存储到 etcd。除此之外，Scheduler 算法根据 Minion 的资源容量来确定是否将新建 Pod 分发到该 Minion 节点。

#### 4.1.2. Pod Registry

Pod Registry 负责跟踪 Kubernetes 集群中有多少 Pod 在运行，以及这些 Pod 跟 Minion 是如何的映射关系。将 Pod Registry 和 Cloud Provider 信息及其他相关信息封装成实现 Kubernetes API Server 的 RESTful API 接口 REST。通过这些 API，我们可以对 Pod 进行 Create、Get、List、Update、Delete 操作，并将 Pod 的信息存储到 etcd 中，而且可以通过 Watch 接口监视 Pod 的变化情况，比如一个 Pod 被新建、删除或者更新。

#### 4.1.3. Service Registry

Service Registry 负责跟踪 Kubernetes 集群中运行的所有服务。根据提供的 Cloud Provider 及 Minion Registry 信息把 Service Registry 封装成实现 Kubernetes API Server 需要的 RESTful API 接口 REST。利用这些接口，我们可以对 Service 进行 Create、Get、List、Update、Delete 操作，以及监视 Service 变化情况的 watch 操作，并把 Service 信息存储到 etcd。

#### 4.1.4. Controller Registry

Controller Registry 负责跟踪 Kubernetes 集群中所有的 Replication Controller，Replication Controller 维护着指定数量的 pod 副本(replicas)拷贝，如果其中的一个容器死掉，Replication Controller 会自动启动一个新的容器，如果死掉的容器恢复，其会杀死多出的容器以保证指定的拷贝不变。通过封装 Controller Registry 为实现 Kubernetes API Server 的 RESTful API 接口 REST，利用这些接口，我们可以对 Replication Controller 进行 Create、Get、List、Update、Delete 操作，以及监视 Replication Controller 变化情况的 watch 操作，并把 Replication Controller 信息存储到 etcd。

#### 4.1.5. Endpoints Registry

Endpoints Registry 负责收集 Service 的 endpoint，比如 Name: "mysql"，Endpoints: ["10.10.1.1:1909", "10.10.2.2:8834"]，同 Pod Registry，Controller Registry 也实现了

Kubernetes API Server 的 RESTful API 接口，可以做 Create、Get、List、Update、Delete 以及 watch 操作。

#### 4.1.6. Binding Registry

Binding 包括一个需要绑定 Pod 的 ID 和 Pod 被绑定的 Host，Scheduler 写 Binding Registry 后，需绑定的 Pod 被绑定到一个 host。Binding Registry 也实现了 Kubernetes API Server 的 RESTful API 接口，但 Binding Registry 是一个 write-only 对象，所有只有 Create 操作可以使用，否则会引起错误。

#### 4.1.7. Scheduler

Scheduler 收集和分析当前 Kubernetes 集群中所有 Minion 节点的资源(内存、CPU)负载情况，然后依此分发新建的 Pod 到 Kubernetes 集群中可用的节点。由于一旦 Minion 节点的资源被分配给 Pod，那这些资源就不能再分配给其他 Pod，除非这些 Pod 被删除或者退出，因此，Kubernetes 需要分析集群中所有 Minion 的资源使用情况，保证分发的工作负载不会超出当前该 Minion 节点的可用资源范围。具体来说，Scheduler 做以下工作：

- 实时监测 Kubernetes 集群中未分发的 Pod。
- 实时监测 Kubernetes 集群中所有运行的 Pod，Scheduler 需要根据这些 Pod 的资源状况安全地将未分发的 Pod 分发到指定的 Minion 节点上。
- Scheduler 也监测 Minion 节点信息，由于会频繁查找 Minion 节点，Scheduler 会缓存一份最新的信息在本地。
- 最后，Scheduler 在分发 Pod 到指定的 Minion 节点后，会把 Pod 相关的信息 Binding 写回 API Server。

### 4.2. Kubelet

根据图 3 可知 Kubelet 是 Kubernetes 集群中每个 Minion 和 Master API Server 的连接点，Kubelet 运行在每个 Minion 上，是 Master API Server 和 Minion 之间的桥梁，接收 Master API Server 分配给它的 commands 和 work，与持久性键值存储 etcd、file、server 和 http 进行交互，读取配置信息。Kubelet 的主要工作是管理 Pod 和容器的生命周期，其包括 Docker Client、Root Directory、Pod Workers、Etcd Client、Cadvisor Client 以及 Health Checker 组件，具体工作如下：

- 通过 Worker 给 Pod 异步运行特定的 Action。
- 设置容器的环境变量。
- 给容器绑定 Volume。
- 给容器绑定 Port。
- 根据指定的 Pod 运行一个单一容器。

- f. 杀死容器。
- g. 给指定的 Pod 创建 network 容器。
- h. 删除 Pod 的所有容器。
- i. 同步 Pod 的状态。
- j. 从 Cadvisor 获取 container info、 pod info、 root info、 machine info。
- k. 检测 Pod 的容器健康状态信息。
- l. 在容器中运行命令。

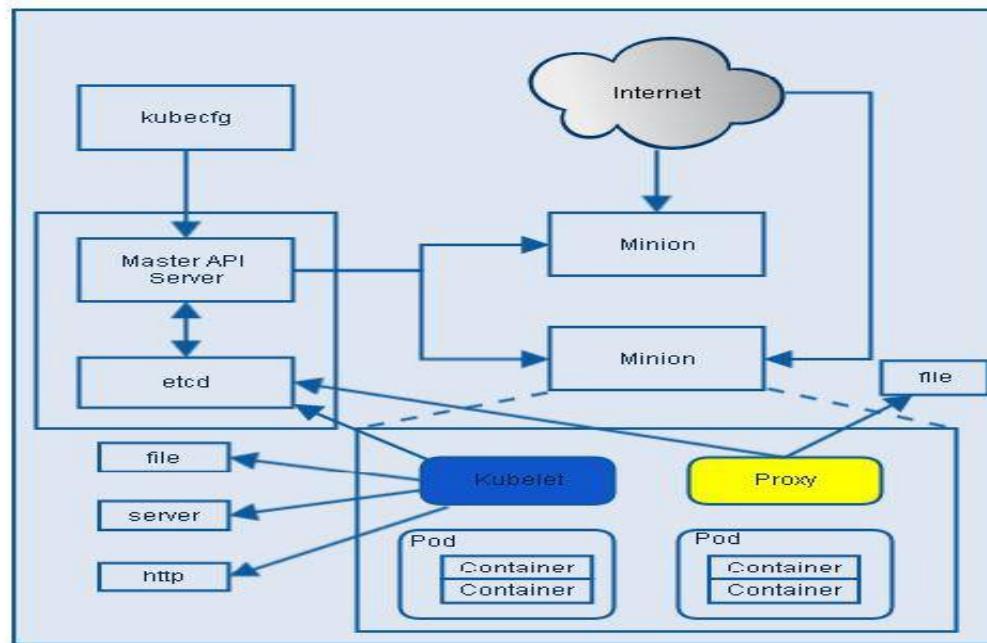


图 3 Kubernetes 详细构件

### 4.3. Proxy

Proxy 是为了解决外部网络能够访问跨机器集群中容器提供的应用服务而设计的，从图 3 可知 Proxy 服务也运行在每个 Minion 上。Proxy 提供 TCP/UDP sockets 的 proxy，每创建一种 Service，Proxy 主要从 etcd 获取 Services 和 Endpoints 的配置信息，或者也可以从 file 获取，然后根据配置信息在 Minion 上启动一个 Proxy 的进程并监听相应的服务端口，当外部请求发生时，Proxy 会根据 Load Balancer 将请求分发到后端正确的容器处理。

## 5. 下篇主题

下篇讲述在 CentOS7 上用 Kubernetes 来管理容器。

## 6. 个人简介

杨章显，现就职于 Cisco，主要从事 WebEx SaaS 服务运维，系统性能分析等工作。特别关注云计算，自动化运维，部署等技术，尤其是 Go、OpenvSwitch、Docker 及其生态圈技术，如 Kubernetes、Flocker 等 Docker 相关开源项目。Email:  
[yangzhangxian@gmail.com](mailto:yangzhangxian@gmail.com)。

## 7. 参考资料

1. <https://github.com/GoogleCloudPlatform/kubernetes/tree/master/docs>
2. <http://www.slideshare.net/rajdeep>
3. <http://www.docker.com>

---

感谢郭蕾对本文的策划和审校。

查看原文：[Kubernetes 系统架构简介](#)

# 一个 OpenStack 访问请求在各组件之间的调用过程

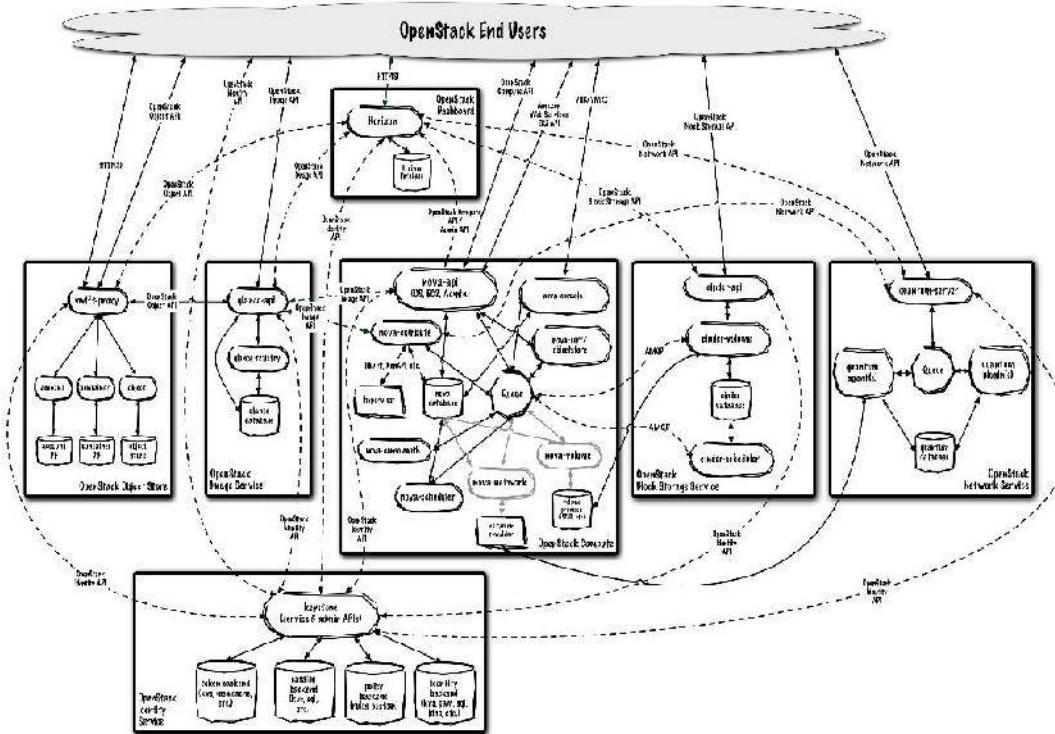
作者 乔立勇

## OpenStack 各个组件之间的关系

OpenStack 是一套资源管理软件的集合，也是当前最热的开源虚拟化管理软件之一，有一个全球 139 个国家将近两万开发者参与的开源社区（[www.openstack.org](http://www.openstack.org)）作为支持。OpenStack 项目的目的是快速建设一个稳定可靠的公有云或私有云系统。整个项目涵盖了计算，存储，网络以及前端展现等关于云管理的全部方面，包含了众多子项目，其中主要的子项目有：

- OpenStack Compute (code-name Nova) 计算服务
- OpenStack Networking (code-name Neutron) 网络服务
- OpenStack Object Storage (code-name Swift) 对象存储服务
- OpenStack Block Storage (code-name Cinder) 块设备存储服务
- OpenStack Identity (code-name Keystone) 认证服务
- OpenStack Image Service (code-name Glance) 镜像文件服务
- OpenStack Dashboard (code-name Horizon) 仪表盘服务
- OpenStack Telemetry (code-name Ceilometer) 告警服务
- OpenStack Orchestration (code-name Heat) 流程服务
- OpenStack Database (code-name Trove) 数据库服务

OpenStack 的各个服务之间通过统一的 REST 风格的 API 调用，实现系统的松耦合。下图是 OpenStack 各个服务之间 API 调用的概览，其中实线代表 client 的 API 调用，虚线代表各个组件之间通过 rpc 调用进行通信。松耦合架构的好处是，各个组件的开发人员可以只关注各自的领域，对各自领域的修改不会影响到其他开发人员。不过从另一方面来讲，这种松耦合的架构也给整个系统的维护带来了一定的困难，运维人员要掌握更多的系统相关的知识去调试出了问题的组件。所以无论对于开发还是维护人员，搞清楚各个组件之间的相互调用关系是怎样的都是非常必要的。



## 从 nova-client 入手

nova-client 是一个命令行的客户端应用，终端用户可以从 nova-client 发起一个 api 请求到 nova-api，nova-api 服务会转发该请求到相应的组件上。同时，nova-api 支持对 cinder、neutron 的请求转发，也就是你可以在 nova-client 直接向 cinder、neutron 发送请求。

我们可以在调用 nova-client 增加--debug 选项打印更多的 debug 消息，通过这些 debug 信息可以了解到如果我们需要发起一个完整的业务层面上请求，都需要跟那些服务打交道。

以 boot 一个新实例为例子，以下是执行代码以及 debug 输出：

```
[tagett@stack-01 devstack]$ nova --debug boot t3 --flavor m1.nano --image 44c37b90-0ec3-460a-bdf2-bd8bb98c9fdf --nic net-id=b745b2c6-db16-40ab-8ad7-af6da0e5e699

...
REQ: curl -i 'http://cloudcontroller:5000/v2.0/tokens'

REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/images/44c37b90-0ec3-460a-bdf2-bd8bb98c9fdf'
```

```

REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/flavors/m1.n
ano'
REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/servers' -X
POST
-H "Accept: application/json" -H "Content-Type: application/json" -H "User-
Agent:
python-novaclient" -H "X-Auth-Project-Id: admin" -H "X-Auth-Token:
{SHA1}15d9e554b7456f1043732bb8df72d1521c5f6aa1" -d '{
"server": {
"name": "t3",
"imageRef": "44c37b90-0ec3-460a-bdf2-bd8bb98c9fdf",
"flavorRef": "42",
"max_count": 1,
"min_count": 1,
"networks": [
{"uuid": "b745b2c6-db16-40ab-8ad7-af6da0e5e699"}]
}''

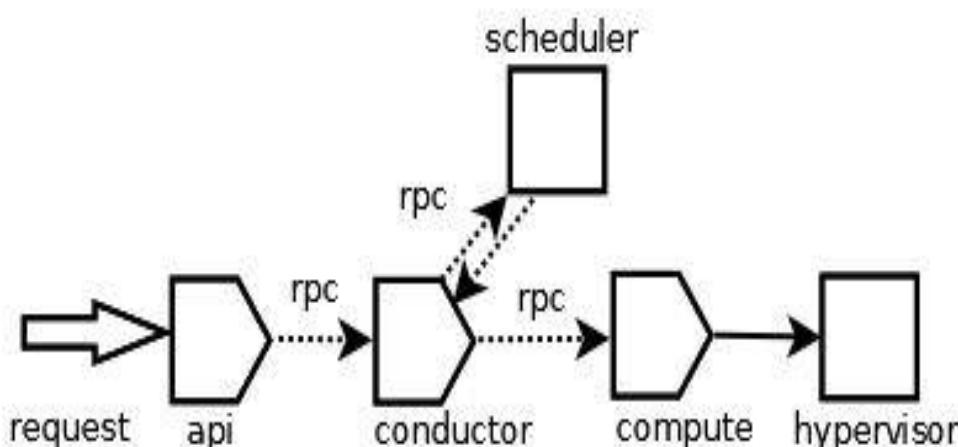
```

从以上 debug 输出我们可以清楚看到，执行一个 boot 新实例的操作需要发送如下几个 api 请求：

1. 向 keystone 发送请求，获取租户（d7beb7f28e0b4f41901215000339361d）的认证 token。
2. 通过拿到的 token，向 nova-api 服务发送请求，验证 image 是否存在。
3. 通过拿到的 token，向 nova-api 服务发送请求，验证创建的 flavor 是否存在。
4. 请求创建一个新的 instance，需要的元数据信息通过包含在请求 body 中。

nova-client 帮我们把需要的全部请求放到一起，而最重要的就是 4。如果用户想自己通过 rest api 直接发送 http 请求的话，可以直接使用 4，当然，前提是先通过调用 keystone 服务得到认证 token。

下面结合代码重点叙述一下 4 的请求数据流动在整个 stack 中的过程。



上图是一个全局的流程图，图中每个服务是一个单独的进程实例，他们之间通过 rpc 调用（广播或者调用）另一个服务。nova-api 服务是一个 wsgi 服务实例，创建新 instance 的入口代码是在 nova /api/openstack/compute/servers.py，处理函数为：

```

def create(self, req, body):
    """Creates a new server for a given user."""
    ...
    (instances, resv_id) = self.compute_api.create(context, ...)

```

做一些参数验证之后，调用 compute api 的 create 函数（代码在 nova/compute/api.py 中）：

```

@hooks.add_hook("create_instance")
def create(self, context, instance_type,
          ...):
    return self._create_instance(...)

```

创建 instance 对象实例，\_create\_instance 会调用 compute\_task api 的 build\_instances 方法对刚创建的 instances 实例进行构建：

```
self.compute_task_api.build_instances(context, ...)
```

compute\_task api 是一个 nova-conductor 服务的 rpc api 请求，处理代码在 nova/conductor/manager.py 中：

```

def build_instances(self, context, instances, image, filter_properties,
                   ...):
    hosts = self.scheduler_rpcapi.select_destinations(context,
                                                     ...)
    self.compute_rpcapi.build_and_run_instance(context, ...)

```

它做了两件事情：调用 scheduler 的 rpc api 选择在那些主机上创建新实例，并最终通过 rpc 请求 nova-compute 服务去构建和运行新实例。

处理函数在 nova/compute/manager.py 中：

```

def build_and_run_instance(self, context, instance, image, request_spec,
                           filter_properties, admin_password=None,
                           injected_files=None, requested_networks=None,
                           security_groups=None, block_device_mapping=None,
                           node=None, limits=None):

```

最终调用配置文件中配置的 hypervisor 类型进行虚拟机的创建和运行，一个实例就这样构建好了。

以下是上面涉及到的服务的主要功能：

1. nova-api：接受 http 请求，并响应请求，当然还包括请求信息的验证。
2. nova-conductor：与数据库交互，提高对数据库访问的安全性。

3. nova-scheduler: 调度服务，决定最终实例要在哪个服务上创建。迁移，重建等都需要通过这个服务。

4. nova-compute: 调用虚拟机管理程序，完成虚拟机的创建和运行以及控制。

以上基本包含 nova 项目的全部服务，但一个请求有的时候并不需要经过全部的服务。继续看 shelve 一个实例的过程。

```
[tagett@stack-01 devstack]$ nova --debug shelve t2

REQ: curl -i 'http://cloudcontroller:5000/v2.0/tokens' ...

REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/servers'...

REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/servers/r'...

...
REQ: curl -i
'http://cloudcontroller:8774/v2/d7beb7f28e0b4f41901215000339361d/servers/
00be783d-bef5-46b1-bfdc-316618c76e92/action'
-X POST -H "Accept: application/json" -H "Content-Type: application/json"
-H "User-Agent: python-novaclient" -H "X-Auth-Project-Id: admin"
-H "X-Auth-Token: {SHA1}0634ea0ef1c3994e1f496c5d8890d32610cf11e9"
-d '{"shelve": null}'...
```

1. 向 keystone 发送请求，获取租户（d7beb7f28e0b4f41901215000339361d）的认证 token。
2. 通过拿到的 token，向 nova-api 服务发送请求，显示该租户的全部服务实例。
3. 通过拿到的 token，向 nova-api 服务发送请求，查询准备 shelve 的实例 uuid 的详细信息。
4. 请求一个 server 操作 action，执行 shelve 操作(request body 为‘{"shelve": null}’)。

nova-api 返回 http 202，成功接受请求，转为后台进行异步执行。

```
RESP: [202] CaseInsensitiveDict({'date': 'Thu, 18 Sep 2014 04:03:09 GMT',
'content-length': '0', 'content-type': 'text/html;
charset=UTF-8', 'x-compute-request-id': 'req-4be7dc9a-21da-4050-9310-3ee58ca93569'}) RESP BODY: null
```

上面 4 中的 shelve 操作代码在 nova/api/openstack/compute/contrib/shelve.py :

```
@wsgi.action('shelve')
def _shelve(self, req, id, body):
    """Move an instance into shelved mode."""
    context = req.environ["nova.context"]
    auth_shelve(context)
```

```

instance = self._get_instance(context, id)
try:
    self.compute_api.shelve(context, instance)
except exception.InstanceIsLocked as e:
    raise exc.HTTPConflict(explanation=e.format_message())

```

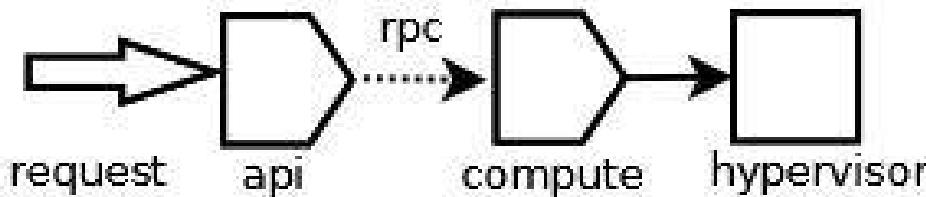
从代码可以看出，nova-api 服务直接调用了 compute\_api，代码位于 nova/compute/api.py：

```

if not self.is_volume_backed_instance(context, instance):
    name = '%s-shelved' % instance['display_name']
    image_meta = self._create_image(context, instance, name,
                                    'snapshot')
    image_id = image_meta['id']
    self.compute_rpcapi.shelve_instance(context, instance=instance,
                                         image_id=image_id)
else:
    self.compute_rpcapi.shelve_offload_instance(context,
                                                instance=instance)

```

compute\_api 直接调用 rpc 消息请求，所以，直接将消息发送给了 nova-compute 服务，所以最终各个组件之间的调用关系如下：



## 结论

本文介绍了 OpenStack 项目的所有组件以及组件之间的调用关系，并从 nova-client 入手，结合代码分析了两个具体实例，从实例的 debug 消息分析得出如果我们需要完成一个完整的业务请求，需要调用那些 api 请求；从代码分析，可以得出 api 调用的大致关系。rpc 请求用于实现一个组件内部的各个服务，如 nova 组件中的 nova-api、nova-compute、nova-conductor、nova-scheduler 等。而不同组件之间的调用则是通过 rest api 请求实现，如 nova 组件的某一服务需要调用 cinder 服务，则是在 nova 组件引入改 cinder 服务的 client api，实现 rest-api 请求。

---

## 作者简介

乔立勇，IBM 高级系统软件研发工程师，2011 年加入 IBM 中国 Linux 技术中心，一直从事 Linux 虚拟化（KVM）相关方向的研发工作。曾在 [IBM developerworks](#) 发表多篇关于虚拟化方面的文章，[Linux 与虚拟化实验室社区](#)的编辑。现在是 openstack 开源社区一名代码贡献者（launchpad id 'Eli Qiao'）。

查看原文：[一个 OpenStack 访问请求在各组件之间的调用过程](#)

# 你的数据库危机四伏

作者 Yaniv Yehuda，译者 汪佳南

我们已经给予了数据库充分的关注，因此它们不应成为 IT 风险因素。但即便为 DRP（灾难恢复计划）准备预算、备份机制并且拥有一流的 DBA，数据库仍然造成了重大威胁。这是为什么呢？

## 变得敏捷

在快速发展、充满竞争的市场中，如果你的竞争对手能够更快更好地发布相关产品，那就意味着你终将失去市场份额。这也就是为什么公司需要变得敏捷。公司需要更好地掌控信息、更快地制定决策、加快软件交付以及优化质量控制。敏捷生来就是处理这些挑战的。它一方面使组织能够更快地发展，并处理不断变化的需求，另一方面又能确保最佳质量。把敏捷概念带入产品（产品环境、客户站点等）当中，并使得开发和运营紧密相连，这样的需求导致了 DevOps 的诞生。

敏捷和 DevOps 的出现并未使软件开发生命周期（SDLC）的最佳实践发生彻底的改变。因为它们是软件开发演进的步骤，而不是一次彻头彻尾的变革。然而，如果你正行驶在敏捷世界的快车道上的话，如何贯彻执行这些最佳实践将显得更为重要。此外，自动化是极其重要的，它既提升了总体效率，又减少了频繁变更和发布的固有风险。

流程的自动化、持续集成、持续交付以及持续部署的实现，这些实践已经被反复证明过了。它们确保了总体流程的高效和可靠性。

## 对数据库需要特别关注

基于自动化的可重复性和可持续性流程共同驱动着更安全和更少出错的部署工作。要想在频繁变更中立于不败之地，那就不能依赖于个人能力去记住所有已完成的步骤，也不能依赖于群体能力去识别出可能受到当前变更影响的所有范围。关键在于要尽可能减少人工操作。

但不同于其他软件组件和代码（或者是编译过的代码），数据库可不只是堆文件。数据库是你最有价值的资产，它包含了需要妥善保管的业务数据。因此我们不能将它

从开发环境拷贝粘贴到测试环境再到产品环境。大多数情况下，数据库的开发与应用代码（.Net 或者 Java 开发）显得截然不同：开发人员和 DBA 访问并更改共享资源，即一个中央数据库，而不是工作站上的一个本地拷贝。

数据库开发和普通应用代码开发是如此不同，最终导致了孤岛的产生。

应用开发人员正勇往直前，他们采用优秀的新开发工具、实践敏捷方法、自动化以及持续交付。相比之下，数据库开发采用的却是一个相对不受控制的流程。数据库开发与其说是工程实践，倒更像是一门与整个 SDLC 流程相隔绝的艺术。在这里，人们从未共享过开发工具、流程和最佳实践。

数据库作为有价值的资产，极易成为摇晃的车轮从而破坏了整车的平衡（即成为短板）。

## 数据库变更应如何进行处理

我们不应该仅仅因为数据库构造的不同而视它为特别的事物。我们应该确保所有团队、代码开发人员、DBA 和数据库开发人员都采用相同的流程。如同在代码开发中做的那样，我们应当遵循这些已得到证明的软件开发最佳实践，并将其应用到数据库开发当中去。

在整个开发周期中，为了将数据库造成的风险最小化，我们需要按照以下方式解决问题。

- 敏捷开发——实行短小的开发周期，同时将数据库变更作为基于任务的开发工作的一部分进行管理。这样做能够确保代码变更和数据库变更同时完成，并在之后能够基于变更请求一同部署。  
——像 Jira 这样的任务管理解决方案能够将任务（task）、问题（issue）、变更需求（change requirements）或者问题记录（trouble ticket）与实际需要完成的工作对应起来。它能帮助并引导你朝着明确定义的可交付成果方向进行努力，并在之后采用以任务为中心的视角来部署这些变更。
- 像 Chef、Puppet 和 Jenkins 这样的工具使得软件开发环境的配置变得更加高效。但是数据库方面又是如何呢？你如何配置大型数据库环境？即便是配置小型数据库环境？数据库仍然是敏捷跨不过去的槛。  
——使用 [Delphix](#) 可以方便地创建虚拟数据库分支。创建开发环境将变得轻而易举，并行开发也变得触手可及。生产环境的虚拟拷贝允许你使用最新产品环境的真实数据来测试产品部署，而无需处理存储问题或是真实的产品风险。

● 假如你尝试去构造一个良好的开发、测试、UAT（用户接受性测试）或是产品预发布环境，那么内容屏蔽确确实实是一个挑战。测试数据越真实，你对质量保障也就越充满自信。但是如果开发人员可以不加约束的访问产品拷贝，那会怎样呢？你的信用卡号码是否有可能正好留在其中一台笔记本上？它会泄露出去吗？另一方面，如果采用模拟数据或是根本不用任何数据，那将无法营造一个良好的测试环境，同时性能测试也会不准确。

——像 DMsuite 这样的解决方案能够用逼真、但也是虚构的数据来替换机密数据，而不需要任何编码。如果为了达到某种目的（健康保健亦或是财务方面的内容）或者正好存在一些敏感信息而使得你需要这样做的时候，它会通过在非产品环境中替换机密数据的方式从而减少你的“可侵入的足迹”。

● 数据库变更管理——运用数据库工具以确保数据库变更能够作为 SDLC 的一部分妥善管理，而不是隔绝在主要开发流程之外。另外，确保每一次变更都和一个变更请求关联起来。这有助于将数据库管理工作作为通用开发或变更工作的一部分。

——通过使用 DBmaestro 可以加强版本控制并同时追踪所有变更。一切尽在你掌控之中：谁在做什么事，在哪里，为什么这样做。它将消除流程外的变更。

一旦开发周期结束，DBmaestro 会将相关的变更自动部署到集成环境中去，识别出变更冲突并且合并它们，如果没有冲突则直接推送到产品环境中去。

● 测试自动化——建立合适的自动化测试需要巨大的投资。你需要将之视为长远目标，而非一次性项目，并开始为每次新的开发活动或是主要变更创建并积累自动化测试。

单元测试：建立坚固的安全网络需要采用以变更为重点的测试方法，同时确保那些变更没有造成破坏。

——通过采用 Oracle SQL Developer 或者 Microsoft 平台的 tSQLt 这样的单元测试工具就可以引进数据库单元测试，而无须任何花费。

回归测试：另一种测试方法，它将数据库和应用代码作为一个整体来进行测试。

——使用 TestComplete 验证那些常见的场景，之后也可以加入对不太常见的场景的验证，这将使你自信满满地交付发布。

● 发布自动化——缓解风险的关键角色。自动化的部分越多，你就越能做到精确掌控：重现开发中引入的变更，并在测试环境中测试，最终安全地应用到产品环境中去。一旦你自动化了 SDLC 的流程并且成功解决日常工作中的挑战，那么一个健康的 SDLC 就水到渠成了，它能同时兼顾最小风险和最高效率。

——使用 IBM UrbanCode 部署工具能够帮助你管理发布活动和配置，并精心策划总体流程。

最好的消息是，这些不同类型的解决方案完全可以合作无间。因而可以使用 Delphix 创建虚拟开发分支，使用 DMsuite 屏蔽机密内容，使用 Jira 管理开发任务，使用 DBmaestro 对变更和发布进行版本控制，使用 tSQLt 或者 SQL Developer 进行自动化测

试发布，以及使用 uDeploy 来策划发布流程，以上所有都能汇总成一个精巧的、自动化的并且安全的流程。

## 总结

当着手采用最佳实践的时候，数据库才展现出真正的挑战。这也正说明了孤岛是如何产生的。开发部门或许会采用不同的流程来分别对待代码开发和数据库开发，或仅在部分变更工作中采用最佳实践。

消除主要风险的最佳方法是，不要以技术眼光去看待这些差异，同时找到解决方法——流程或者工具——去采用并实施最佳实践和自动化。

作为一份具有如此高价值的资产，数据库不应成为你的业务的短板。

---

## 作者简介

**Yaniv Yehuda** 是 DBmaestro 公司的联合创始人兼 CTO。DBmaestro 是一家以数据库开发和部署技术为重点的企业级软件开发公司。Yaniv 同时也是 Extreme Technology 公司的联合创始人兼开发部主管。Extreme Technology 是一家面向以色列市场的 IT 服务供应商。Yaniv 曾是以色列国防军队计算机中心的上尉。他在那里担任软件工程经理。

查看英文原文：[Your Database: The Threat That Lies Within](#)

查看原文：[你的数据库危机四伏](#)

# 说说远程团队协作的故事

作者 Ben Linders , 译者 郑柯

Lisette Sutherland 和 Elinor Slomba 在一起收集一些人的故事，这些人的业务模式需要依靠远程团队正确完成工作。故事中体现出远程团队如何协作，如何跨越距离的障碍，如何建立信任，如何完成任务。即将出版的《高能协作：远程战地指南》一书将会讲述这些故事。

InfoQ 采访了 Lisette 和 Elinor，请她们讲讲人们如何在远程团队中工作，他们使用何种工具协作、沟通，以及远程团队哪些东西。

## InfoQ：为什么人们选择远程团队工作方式？

**Lisette:** 吸引合适的人才，将有特定技能的人聚合在一起，这是一个重要推动因素。无论小企业，大企业，还是跨国公司，都是如此。人们不必呆在一个办公室里，要想跟有特定能力的人工作，还不需要旅行或是通勤。这样一来，团队可以得到来自多个工种的快速反馈。

办公室有潜在的社交习惯。一种工作方式对你来说高效，对我可不一定，所以大家的工作效率就被拉低到最低的平均值。如果想要让远程工作得以成功，就得让大家可以用自己最高效的方式自由工作。**Elinor:** 不必受限于特定办公地点，还可以降低启动和维持商业投资的风险。远程团队不需要处理那么多现实问题，这也很有好处。

面临改变的组织，以前会外聘一个咨询顾问，一段时间之内，就要让一只毛毛虫变成蝴蝶（Mike Sutton 提供了这个比喻）。然而，企业发现，要想让一些东西稳定、持续下来，变更过程并不遵循严格的线性回归方式。随着时间不断推移，某些远程人员能够为组织提供更多外部专业知识，令组织收益良多。

如果想让合适的人能够长时间参与，完成有价值的事情，他们不需要挣扎于通勤路上，或者远离自己的家庭。在全新的职场中，某些定量的改进正在发生，这在不久之前似乎还是不可能的！当然，也不是到处都如此，但确实有一些地方值得研究，成功者的故事可以让更多人看到。

**InfoQ:** 远程团队与坐在一起的团队之间有什么区别？

**Elinor:** 当然，面对面沟通还是不一样。习惯下来也需要一定时间，但当然有其长处。有些结对的程序员喜欢真正坐在一起，连续紧密工作几个小时。如果用点儿脑子，只要有高速宽带支持彼此的互动，远程连接也可以很流畅，而且满足个人的特定需求。比如，在会议中，为了测试某些假设，远程团队成员可以立即联系对方。要联系一个负责业务的副总裁，与联系开发人员或团队其他成员没有任何区别。

远程团队的关注点在于正在创造的价值，而不是某些公司的条条框框，或是担心流程是否符合坐在一起的经理的要求。如果对比分布式办公环境中的可能产出，一帮人在监督之下，于固定时间出现在固定地点，天长日久，这看起来像是工业革命时期过时的废气遗迹。

远程工作会让大家更加享受与珍惜和同事站在一起分享实际空间的机会。因为这样的时间不多，所以人们不再将彼此的行动视为理应如此。在分布式团队的成员看来，任何可以一起工作或者交流的机会，都应得到仔细规划和珍惜。

**InfoQ:** 你们访谈了不少远程工作的团队，并整理成故事分享出来。是什么促使你们这么做？

**Elinor:** 我们都有多年远程工作经验，而且发现每个职场人士都要跨越某种形式的远程距离，即使是在同一个地方。我们希望讲述这些故事，让更多人看到这个问题，创建一个平台，分享解决方案。每个公司和团队，都需要组合特定于自己情形的沟通方法、个性、技术和能力，以适应自己的远程情况。从 IT 到艺术，跨越多个地点共同学习，我们希望让远程协作者和主持推动者共同发出声音，让优秀远程团队的特质得以深入展现。

**Lisette:** 我们写的这本书，将会展现针对成功远程协作团队的广泛调查结果。不过，我们在构建的社区要更广泛。这个空间充满活力，人们在尝试许许多多不一样的东西。我们希望能够促进对话，让人们彼此相识，并留住有趣的故事，让大家可以分享知识，一起变得更加有力量。

**InfoQ:** 大家使用什么样的工具来远程协作？这些工具真得有用吗？

**Lisette:** 简单的工具比如即时消息，复杂的完整的协作平台比如 Yammer，现在有几百种工具，可以让远程工作富有成效。我们现在也在实验和尝试很多其他工具，比如：Squiggle、Trello、Boardthing、Hangouts、Sococo、Second Life 等。它们都有自己的优势和不足，不过总的来说技术层面总是比较简单的。

最重要的事情，是要选择一组符合团队需要的工具，然后不断地调整让它们适应团队。然而，沟通比工具更重要。远程工作需要积极沟通！

有一种积极沟通的技巧，名为“大声工作（Working out loud）”，就是要讲述你的工作，让别人都知道你做了什么。就其本质而言，这就是记录你正在做的工作，然后让与你一起工作的人们知道你在干什么。有很多种不同方法可以实现这个目的，再次强调，选择一种适合你的团队的方法很重要。

**Elinor:** 如果一个团队能够达到自组织的水平，他们就一定能够找到最适合自己的工具。但是我们知道，自组织需要组织架构层面的配合。当你与几个不同大陆的人一起，开发一些很重要的项目，而且能进入随心所欲的状态，那确实很容易提升提振士气。不过，挑战确实存在，而且我们也不会掩饰这一点。

管理人员可以提供很多支持，其中最好的做法，就是创建一种异步聊天机制。在这里，团队成员可以放进去所有的东西，从新页面的设计，到下个季度应有的业务目标。你不必担心邮件应该抄送给谁，而且也不会丢失任何上下文。还有一个额外的好处，编写方针手册的压力也变小了，因为现在有记录，可以搜索到最新决策。

**Lisette:** 保持简单很重要。沟通应该简单、频繁，而且要轻量级。不管怎么做，焦点都要放在如何让沟通更加高效，同时尽量减少摩擦。

**InfoQ:** 当人们在时间和空间都分散的团队中工作时，他们都会使用什么样的实践？能给出一些例子吗？

**Lisette:** Spotify 的大部分团队都是坐在一起的。然而，其中一个远程团队决定使用 Google Hangouts，团队会一直开着它。团队通过视频互相联系，

麦克风会静音。如果有人有问题，他们就会取消静音。这让他们觉得自己在同一个房间内工作。

**Elinor:** Treehouse Learning 的一切事务都基于项目。如果你想启动某项工作，就将其作为项目提出草案，然后识别出需要的角色，比如：设计一个新页面，你可能需要一个开发人员、一个设计师和一个数据分析人员。很多人会了解这个项目，也可以选择接收通知，了解哪些新项目需要自己的技能。如果有足够的人填充项目角色，项目就会往前推进。

很多跨越多个时区的团队，尝试不同的工作安排时间表，让团队有尽量多的时间在一起工作。

**Lisette:** 对于这种通过时间表覆盖解决空间分散问题，我们还观察到一个很有趣的方式，就是在 Sococo 的环境中。在 Sococo 里面，你可以在空间上设计一个虚拟办公室，其地理分布由工作流决定。在规划视图中，你可以看到所有团队成员，知道他们在哪个房间，或者他们在干什么。团队成员也可以很方便地开关视频或麦克风，与个人或小组通话，分享屏幕。Agile Dimension 的 Billy Krebs 编写了一套完整的分布式工具云图，并运转了一个分布式敏捷研究组（Distributed Agile Study Group）。

**Elinor:** 我们接触过这样一位管理人员，他设计了一个实验：平常在一起工作的人们，在家呆上一整个星期，然后尝试像你平常一样完成他们的工作。这么做的目的，是希望让坐在一起的团队成员能够更多了解远程团队的感受。人们需要理解：要是觉得被排斥了，每个人都会很敏感；而且对于坐在身边的人，很容易就会投放更多注意力。

有意识地制造真正的面对面接触，这是很重要的实践。我们访谈过一位远程工作者，他为马拉松提供技术支持。去办公室的时候，与 CEO 见面的方法，是一起跑步。这对建立个人联系非常有帮助，而且在业务策略层面，也能与高层保持对话。

**InfoQ:** 在远程团队中工作，需要具备哪些条件？是不是得有某些特定技能？这些东西可以学习吗？

**Lisette:** 从性格层面说，有些人喜欢在安静的个人空间中工作，他们对此很敏感，而且产出更高，远程工作对他们来说至关重要。相反，喜欢社交、性格外向的远程工作者，他们更适合于联合工作空间（coworking），现在很多城市都有类似环境，而且有一些全球目录可以找到类似地方，比

如 Sharedesk。知道自己是什么样的人很有帮助，你就可以找到适合自己个性特点的解决办法。

不过，纪律这件事儿就没那么容易了。你必须知道自己什么时候犯了拖延症，然后要有意识地制止自己。有时候你得激励自己挺过去，有时候不妨休息一下。你要有自知之明，同事知道什么样的东西能够激发自己。

与之相关的是，工作空间也很重要。如果你想在家工作，需要考虑如何安排你的环境，满足自己的工作要求。确保你有一个合适的工作空间，光线要好，还要能为视频会议提供好的音质。如果你在外旅行，要保证了解在路上展开工作需要的技术需求，以及你能否具备相应的网络连接。

**Elinor:** 对于管理者，远程团队建设的基础，是对慢慢涌现的产出的信任，以及对团队成员的激励。至于学习能力，我们现在做的一些事情，在以前的职场中从未出现过。所以如果有人是某些特定方法论的“纯粹主义者”，或者将某些宣言视为“天条”，一个字都不能动，他们就会成为障碍，阻碍这个工作领域的演化。

**InfoQ:** 对于希望远程工作的人们，你们能提供一些建议吗？有什么事情是他们必须要做，或者不应该做的？

**Elinor:** 每个人都应该有一种健康的荒谬感，知道事情总会出问题。我们提倡多用几种方法。不要依赖特定的技术，而是重点保证对话可以一直进行。在 2013 年 10 月那个通过病毒式传播的 Stoos in Action 大会上，我们观察到这一点。这个大会非常依赖 Google Hangouts，当大会将要开始直播时，系统因为要维护就掉线了！所以只能使用本地卫星开始会议，直到系统上线。

**Lisette:** 团队日程表中，确保既有正式时间，也有非正式时间。正式时间就是会议或是工作时段，有非常明确的安排。安排明确了，就可以快速而流畅地开始或结束工作。

正式时间没有为直觉和问题的有机产生留出余地……如果有人产生某些感觉，也没有时间表达，无法让其他人了解。非正式时间可以替代“饮水机时间”，大家可以随意聊天。远程团队采取虚拟咖啡或是虚拟午餐方式，可以达成同样目的。如果整个团队都是远程的，你必须仔细考虑如何管理工作环境，让人们产生自发的联系。我们将其称为“设计的巧合（engineered serendipity）”。

## 作者简介：

**书籍作者：****Lisette Sutherland** 创立了多个在线协作社区，非常了解基于 web 的协作工具和线上社区管理。她的目标是让最好的人一起工作，不受地域限制。她目前担任 Happy Melly 的社区构建者（Community Builder），协调全球的市场工作，并组织整理世界各地的知识，将工作变为出色而令人满意的体验。

**主要贡献者：****Elinor Slomba** 将艺术与创业的世界联结在一起，以深入理解创新必要的原则。她的家在康涅提格州纽黑文市，与其他职业人士远程协作，关注视觉艺术、行为艺术和文学艺术等多个领域，还有城市设计、软件和其他技术领域。Elinor 将自己的艺术管理背景与民族志领域研究的调查、敏捷框架结合在一起，帮助创意十足的人们分享跨领域的模型，创造价值。

查看英文原文：[Stories of Collaboration in Remote Teams](#)

查看原文：[说说远程团队协作的故事](#)

# 项目初始会议——如何在一次会议中达成共识

作者 James Bayer，译者 李清玉

在启动一个项目之前预先达成团队共识，这一点在效能和效率上是非常必要的。项目对发起人的重要性体现在哪里？项目如何适用于整个组织的蓝图？项目的最高优先级条目有哪些？以及项目发起人愿意在哪些方面进行妥协？如果团队成员在这些方面没有和发起人达成一致，就有可能导致项目进展脱离进度，或者项目成果无法满足项目发起人的预期。同样地，如果项目发起人与团队无法达成一致，项目发起人就可能会对团队的能力，项目发布的质量与时间抱有不现实的预期。那么，你该如何让这个广义上的团队达成一致呢？



通常情况下，团队成员个人很少会在日常工作中与项目发起人产生大量的互动。通过与整个团队高忠诚度的交流来达到团队共识，远比大量的邮件、文档和电话会议更为有效。通常情况下，由于地理位置，干系人的时间安排和项目安排的原因，保持整个广义上的团队每天都能够保持高忠诚度的交流是不可能的，但整个广义上的团队肯定可以保证在某一个指定的日子里聚在一起。

在 Pivotal 公司，特别是在我们 Cloud Foundry 团队，我们保证团队达成共识的一个有效方法就是开一整天的独立的会议，我们称之为[初始会议](#)。通过本文，你将会学到与这个方法有关的“为什么”，“什么”和“如何”的有关答案，并且回顾一些从这个方法受益的具体例子，这些案例都来自于真实的 Cloud Foundry 项目和团队。

## 什么是初始会议

典型的初始会议（Inception）就是，团队用一个工作日的大部分时间，专注于为启动一个新的项目做准备。在需要对进行中的项目进行重新讨论的时候，初始会议一样也可以用于进行了几个月的项目上。理想情况下，在初始会议之后的第二天，研发团队就可以开始那些高优先级的、具体的和可操作的用户故事了。

有的时候，启动会议可以使我们团队在开工之前，更早的发现干系人或者是项目的目标不清晰，需要做一些额外工作，从而对共同的目标进行提炼和达成一致。在团队开始工作之前对问题进行澄清和达成一致，远远好于团队已经工作了数周，甚至更长时间之后再重新研究，这些做过的工作迟早也会被丢弃。

## 初始会议与其他流程和方法有关吗

极限编程（Extreme Programming）有一个概念叫[规划策略](#)（Planning Game）。初始会议的很多元素都是受这个方法启发。不过在 Pivotal 公司，我们启动项目通常都是用非常统一的会议议程，它非常有效。这与和松散定义的规划策略的议程不同，后者往往贯穿于项目的整个阶段，而不仅仅是一天。

一些人也许会将“初始”这个单词与统一软件开发过程（Rational Unified Process（RUP））中的[初始阶段](#)（Inception Phase）联系起来。初始会议和 RUP 中的初始阶段在语言或目的上都比较相近，但是初始会议是更为轻量级的方法，它能够在一天内达到类似的目标。

## 为什么要像这样开一整天的会

会议的主要目的是达到团队共识和更好的启动项目。Graham Siener 说：初始会议就是让团队关注于“[知道项目要创建什么并且从哪里开始](#)”。我的经验是，初始会议可以让团队达成一致，从而更好的开始一个项目，快速交付价值，并且不会在错误的事情上浪费时间。

## 谁应该参加这个初始会议

初始会议的参会者应该包括做这个工作的核心团队、发起项目的干系人或者他们的代表。特别地，还要包括业务、产品、开发，也许还有像运维与支持等其它团队。也有可能包括上下游团队的代表，他们是这个团队开发产品的制造商或者是客户。实践表示，当会议人数超过 10 人，会议的效率就会开始下降，那是因为这会产生许多小组讨

论，而且让 20 个人有效地参与是很困难的。如果被邀请者列表变得太长，初始会议的引导者或者项目发起人就可以要求团队参与度不高的受邀者，让可以代表他观点的其他参与者参加这个会议。有时候，大规模的初始会议是不可避免的，其代价就是降低每个参会者的参与度并有可能降低共识度。

项目发起人或干系人应该参加，或者应该派代表参加，代表是可以代表他们的观点并且是给予授权的。如果发起人没有时间参加这个重要的会议，但又想对项目的决定施加重要的影响或控制，那么这就发出一个信号，这个团队并没有获得对现在和将来所需要的和应有的支持与关注。

获得一位有经验的团体引导师是重要的，他可以公平地主持会议。对引导师而言，拥有高效的主持能力是至关重要的。他们负责高效地按照议程主持会议，保证团队进行有效的沟通，理解团队应该在哪些地方深入讨论片刻，并且知道哪些讨论占用了太长的时间，应该暂停，并在之后用小组讨论的方式得出结论。

## 初始会议应该安排在什么时间

理想情况下，在新的项目开始即将之前，或者对于已存在的长期项目，即将开始一个新的主要工作之前，应该启动初始会议。如果一个团队有相当数量的人加入或者离开、或者在方向上有很大的变化，那么重新主持一个初始会议可以帮助团队达成共识。

### 初始会议应该怎样主持？

会议引导师应该要求参与者全身心的投入，除了休息时间，不允许打开电脑或者接电话从而分心。引入以下规则：“如果你待在这里，你就全心全意的待着。”可以极大地提高会议的有效性。

理想情况下，初始会议应该在一个大的会议室进行，有白板和大的白板纸。推荐使用多种颜色的马克笔、各种颜色的索引卡，胶带和一些像零食或糖果形状的可以用来投票的东西。会议中间应该经常休息，每小时休息五到十分钟。

## 现场与远程参与者

相比远程参与者，现场参与者的好处如何强调都不为过。远程参与者更容易受到干扰，从而影响沟通的忠诚度。我以前曾看到过初始会议有几个远程参与者的情况，相对于现场初始会议来说，我觉得远程的初始会议参与度不够，团队从我的参与中所获

得的好处则更少。有时候，由于受到现实情况的限制，远程参与的方式确实无法避免。在这种情况下，请尝试用最好的远程在线技术，例如[高品质的麦克风](#)和独立的摄像机。用笔记本自带的低音质的麦克风和摄像头容易让参与者受到邮件和浏览网页的干扰。

## 典型的初始会议议程

**介绍：**保持介绍简短，因为你要花几乎一天的时间和团队在一起，并且你们会通过一天的会议彼此很好地认识。让引导师提醒每位参与者简单的介绍几个关键信息会非常有帮助，例如他是谁、为什么在这里等。

**愿景和目标：**项目发起人和 Product Owner 应该阐明简洁的项目长期愿景和接下来几个月的近期目标。对于愿景和目标的讨论需要给团队安排一定的时间。

**非目标：**和目标一样重要，明确的说明什么不是我们的短期目标非常重要。清晰的说明非目标，是我们限定当前工作范围的一种方式，这样给了团队快速发布价值的许可，暂时不考虑那些今后想要、但不是现在必须要有的东西。要确保为小组讨论非目标预留时间。

**风险：**房间里的每一个人都要用索引卡片识别项目的主要风险。让每个人针对每一个风险写一张索引卡片，需要多少张风险卡片都没问题。引导师应该对风险分类，把相关的风险卡片放到一个类别里叠成一堆。随后把风险读出来，并且给参与者机会，让他们解释在卡片中写下的风险。这还不是对每一个风险进行小组讨论的时机，只是试图理解可能有哪些风险存在。然后，引导师指示每一个人用糖果或其他投票道具对风险投票，每个人三票，找出你认为对于达成项目目标风险最高的类别。你可以对某一个风险类别投出所有的三票，也可以给三个风险类别分别投票。投票结束之后，我们应该从投票数最多的风险分类开始讨论。将每个分类的风险的投票得分记在白板上，或者是一张图片上。团队在当天结束之前应该对其重新投票，看看有没有什么变化。通常情况下是有变化的。

**角色/情景人物：**引导师应该进行一个小组练习，用来识别与项目有关的[角色](#)或[情景人物](#)。我曾见到过多种不同的识别方式，既可以通过 Product Owner 简单的陈述来选出角色与人物，也可以让大家一起进行头脑风暴讨论以得出各种不同的角色和情景人物。关键的目标就是让团队的每一个人理解用户都是谁。

**活动/工作流程：**针对项目短期目标范围内的每一个用户角色或情景人物，引导师应该与讨论组一起定义每个角色或情景人物的活动与工作流程。引导师应该针对讨论组的不同情况选择一种合适的方法。可以使用简单的方式，例如让 Product Owner 定义

关键的活动，然后允许组内讨论，也可以是像游戏一样更有创意的方法。用户与系统如何交互的高层次的描述形成了项目功能的基础，通常在之后可以直接分解为用户故事。例如“学生 Bobby 在书店的目录里面查找一本书，把找到的条目放到他的购物车中，并且完成支付。”就是一个高层次的工作流程的例子。

**用户故事：**如果有时间，可以把活动和工作流程分解到更小的、具体可操作的用户故事。不要担心写的太过具体化，Product Owner 可以在事后提炼这些语言和细节。有时候在初始会议中并没有时间做这个活动，所以估算和排序必须在高层次的活动与工作流程的粒度上进行。这样也是可以的，因为这是 Product Owner 的工作，和开发团队一起工作，引导他们遵循初始会议的流程，确保尽快地写出用户故事。

**估算：**对于非常重要的用户故事，需要与会的开发人员快速地给出粗略的估算。如果你的估算是在用户故事的层面，可以试图使用团队在开发阶段使用的故事点系统。如果你的估算处于史诗、活动或工作流程级别的，可以尝试使用几个开发人员周的粗略估算方式。Martin Fowler 的 [Thrown Estimate](#) 技巧非常有效，可以快速地得到粗略估算。如果你有大量的条目需要估算，我的同事 [Evan Willey](#) 建议使用 [Affinity Estimating](#) 方法。重要的是，客户端发起人和 Product Owner 可以从负责实现的团队那里直接得到估算。

**优先级：**让 Product Owner 把用户故事按照优先顺序排列，并且允许小组讨论和验证。Product Owner 应该能够根据业务价值来判断团队工作的优先级。这些功能在当前阶段是“必须的”，还只是“可有可无”？可以根据团队的大小，看看这个基于故事点的估算是否可以转换成项目周数。在我参加的几乎所有的初始会议中，估算和优先级几乎都预示着团队必须减少在几个月内发布的内容。现在就是一个很好的时机，与项目发起人和 Product Owner 讨论一些取舍，因为他们很可能是首次从负责交付项目的团队那里得到半精确的估算。

**风险：**重复之前的风险讨论环节，看看风险是否发生了变化。

**下一步：**对接下来几天或几周应该做些什么进行小组讨论。这是为了让团队的每个成员达成一致或者告诉每一个人，接下来团队要使用的开发与签入的流程。通常情况下，引导师和 Product Owner 会把白板的内容拍下来、收集活动中使用的索引卡片、捕获行动项，并对谁应该发出一篇总结达成统一意见。我偶尔看到团队把初始会议产生的那些白板纸、索引卡片和白板的图片放到团队的工作区域。随着时间的流逝，这些内容就变得过时，价值也开始降低。

**回顾：**用[敏捷回顾会议](#)对初始会议进行讨论，哪些做得好，哪些东西人们有问题或者有困惑，哪些做得不好，以及对下次会议有什么好的主意等。

**放松：**大家在房间里已经待了一整天，每个人都已经非常疲惫。当天的工作结束时间或许已经快到了，最好让团队在办公室以外，例如选择有利于社交的某个集合地，做一些放松娱乐的活动。通常情况下，那些不能参加初始会议的那些人想知道会议的结论并希望与团队沟通，所以邀请那不能参加初始会议的人也是个不错的主意。让这个活动保持随意是比较重要的，因为某些团队成员需要时间放松减压。

## 持续的达成一致

初始会议在某一点上及时地达成一致是非常好的，但为了确保持续的达成一致，在项目中也应该使用其他类型的循环反馈回路。有效的反馈回路方式包括每日站立会议，每周的迭代计划会议，每周与项目发起人的会议，每周或每两周的回顾会议，以及项目功能的持续发布。在几个月或者重要的里程碑之后，我建议启动另一个初始会议。因为达成一致的团队才是一个有效的团队。

## Cloud Foundry 项目的初始会议

我曾在 [Pivotal](#) 公司的 [Cloud Foundry](#) 团队中工作过 2 年，我相信[我们工作的方式](#)给我们带来了高效的生产力，拥有优秀功能团队，成员们也在工作中获得了乐趣。每一个 Cloud Foundry 的子项目在启动时都会进行初始会议，包括一些最有创意的功能项目，例如 [Loggregator](#)。Loggregator 是从所有的应用中直接添加一些聚合的日志给用户，并且从远程终结点的系统日志中排出。在初始会议中，我们明确地从团队中得到半精确的估算，我们确保功能范围只限于流日志，并不扩展到日志的持久化和搜索功能。用 Golang 重写 Cloud Foundry [命令行接口](#)是我们得益于初始会议的另一个项目。在 Inception 会议中，团队达成一致，我们将从 Ruby 版本的命令行接口（CLI）开始再考虑改进用户体验，从而帮助我们减少了重写的范围，并有了更好的用户体验。

在我以前很多的软件开发经验中，都是使用更传统的瀑布式流程，有大规模的提前设计和文档编写，大规模的团队，以及持续一年或更久的长期项目。我再也不想回到那样的工作方式上了。自从 [Pivotal 实验室创建以来](#)，我们就



用务实的现实项目和持续的改进，Pivotal 公司现在使用的方法是基于有 25 年历史的 [极限编程](#) 和 [敏捷原则](#)。这个流程保证我们达成共识，让核心团队与外部发起人及干系人对项目有共同的理解。在你下一次开始新的项目和方案的时候，不妨试着启动一个初始会议或类似的<sup>1</sup>会议吧。

---

## 作者简介

**James Bayer** 是 Pivotal 公司产品管理部门的资深总监，负责领导产品管理团队开发 Cloud Foundry 这一开源项目。在开发 Cloud Foundry 项目之前，他曾是 Oracle 公司的 WebLogic Server 产品管理团队的一员，他的大量工作都是与 Java EE 技术有关。他还曾从事其它企业级中间件开发相关的职位，如 VMware，BEA Systems，Cordys 和 IBM。James 拥有美国内布拉斯加大学的数学与计算机科学的学士学位。

查看英文原文：<http://www.infoq.com/articles/project-inception-meeting>

---

感谢[邵思华](#)对本文的审校。

查看原文：[项目初始会议 – 如何在一次会议中达成共识](#)

---

<sup>1</sup> 对于这个方法更多更全面的讨论，我的同事 Andrew Clay Shafter 推荐阅读 Diana Larsen 和 Ainsley Nies 写的 [Liftoff: Launching Agile Teams & Projects](#) 一书。



# 封面植物——巨人柱仙人掌

仙人掌原产美洲，全球约有 2000 多种，集中分布在南北美洲赤道两侧的热带地区。它们的体态各异。圆的，如球如桔；长的，如鞭如杖；有的像桂林山水，孤峰突起；有的像十万大山，峰峦重迭。它们的大小也相差悬殊，小的只有孩子们玩的弹子那么大，大的可比人还高，其中最大的一种要算生长在美国南部和墨西哥沙漠里的巨人柱。巨人柱仙人掌的植株硕大无比，高达 15~18 米，最高的可达 21 米以上，直径 30~60 多厘米，重 10 余吨。如果把它锯倒弄断，要两辆大卡车才能把它拖走！

墨西哥以盛产仙人掌驰名世界其品种占世界总品种的一半以上。其中巨人柱仙人掌是世界上最高的仙人掌。

促进软件开发领域知识与创新的传播

# 架构师

ARCHITECT



人物 | People  
Databricks连城谈Spark的现状

观点 | Opinion  
PaaS, 不是银弹  
Spark的现状与未来发展  
Shellshock漏洞证明是时候放弃CGI技术了

特别专题 | Topic  
微观SOA：服务设计原则及其实践方式  
基于微服务架构，改造企业核心系统之实践

InfoQ logo and QR code

2014年11月

## 架构师 11 月刊

每月 8 号出版

本期主编：杨赛

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：[editors@cn.infoq.com](mailto:editors@cn.infoq.com)

InfoQ 中文站新浪微博：<http://weibo.com/infoqchina>

商务合作：[sales@cn.infoq.com](mailto:sales@cn.infoq.com)



### 本期主编：杨赛，InfoQ 高级策划编辑

对软件开发、系统运维、互联网产品、移动技术、项目管理等方向感兴趣。写过一点 Flash 和前端，现在只是个伪码农。曾在 51CTO 创办了《Linux 运维趋势》电子杂志，偶尔也自己折腾系统。曾混迹于英联邦国家，学过物理，做过一些游戏汉化，练过点长拳，玩过足球、篮球、羽毛球等各类运动和若干乐器。喜欢读《失控》。