促进软件开发领域知识与创新的传播

架构师 ARCHITECT

推荐文章 | Article

高可扩展分布式应用程序架构

Hadoop集群实现大规模分布式学习

专题 | Topic

SND&OpenStack漫谈

Mockito设计解析

观点 | Opinion

先把平台做扎实，再来微服务吧

特别专栏 | Column

如何构建交互型分析系统

InfoQ queue

2015年11月

高可扩展分布式应用程序的架构原则

Elastisys云平台官方网站发表了一篇文章，介绍他们在高可扩展分布式应用程序设计和开发方面的经验。

如何构建交互型分析系统

本篇文章详细介绍了TalkingData如何构建交互型分析系统。

SND&OpenStack漫谈

本篇文章是SDN & OpenStack的实践总结，里面介绍了多种解决方案以及对未来技术发展的看法。

Mockito设计解析

原则在总结出来之后，不应该成为僵硬的教条，根据需求灵活地应用这些原则，才能达成好的设计。在这方面，Mockito堪称一个经典案例。

如何在Hadoop集群上实现大规模分布式深度学习

雅虎Big ML团队的Cyprien Noel、Jun Shi和Andy Feng撰文介绍了雅虎在Hadoop集群上构建大规模分布式深度学习的实践。

先把平台做扎实，再来微服务吧

微服务有很多的优势，比如技术的多样性、模块化、独立部署等，但也带来了相应的成本，比如运维成本、服务管理成本等。

架构师 2015年11月刊

本期主编 杜小芳

流程编辑 丁晓昀

发行人 霍泰稳

联系我们

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

刘之 野狗科技 CEO

卷首语

互联网业务具有快速发展和变化，对性能和请求响应时间敏感，要求 24 小时稳定提供服务，必须随时应对并发访问量急剧增长等特点。为了解决这些问题，技术架构作为应用开发中的整体设计和抽象，必须以高开发和维护效率，高性能，高可用性，高可扩展性为目标。同时由于互联网的开放性特点，还必须兼顾安全性的考虑。这是我们对“架构”的理解。

野狗技术团队对技术架构进行了许多摸索和实践，沉淀出我们的架构观：

技术架构要与具体的业务相结合，避免脱离业务空谈架构。

技术架构始终是要为业务服务的，不同的业务有不同的特点。举个简单的例子，我们面向移动端提供数据存储和实时同步的服务。从面向实现的角度讲，仅需要一个云端的数据库，并实现一个数据通信协议就可以了。但是考虑到移动端的特性，我们在架构上做了许多优化：SDK 端拥有可以进行本地操作的本地数据副本，与服务端采用尽可能压缩数据流量的传输协议，在长连接不稳定时退化为 long-polling 等。

以最简单和高效的方式去实现目标，不做过多的假设，避免过度设计。

架构应该是从简单到复杂，不断演进的一个过程。工程师思维驱使人们去开发出一个理想化的系统。然而最好的架构却通常不是最理想的架构，而是最适合的架构。我们见过许多因为过度的架构设计而浪费宝贵的时间的案例，更有甚者因过多不切实际的假想而导致架构迟迟

无法落定，这种现象有个术语描述，叫做“分析瘫痪”。小团队资源有限，不可能把所有的架构考虑都实现到最理想的状态。即使在大公司，技术的实现也需要考虑周期和成本的因素。所以优秀的架构应该结合业务目标，在权衡之中寻求到一个最优的点。

我们认为未来应用架构会有两大发展趋势。首先是后端服务的云化。随着技术的发展和积累，技术的垂直领域越来越多，每个垂直技术领域也越来越具有深度。因此一定会出现越来越多垂直领域的云服务，提供更专业的技术和产品服务。例如现在进行移动端 app 的开发，可能用到的云服务就包括推送，数据分析和处理等服务。这些技术领域都已有相当深度和积累，全部选择自行开发的话，造轮子成本会比较高。其次是长连接在架构设计中将会占据越来越重要的地位。随着移动互联网和物联网的发展，应用对双向实时的数据通信的需求越来越强烈。而长连接服务在高可用性，负载均衡，安全性等策略上与传统的无状态的 HTTP 协议是完全不同的，这将引起更多架构的变革。

创业团队必须尽可能降低成本，提升将产品推向市场的速度。野狗致力于加速应用开发，将和伙伴们携手并肩，面对挑战，共创明天。

ArchSummit 全球架构师峰会 2015

[北京站]

2015年12月18日-19日
北京·国际会议中心
www.archsummit.com

11月27日前 9折优惠 立减680元 团购享受更多优惠

大会介绍

交易量逐年暴涨，双十一电商架构今年有何不同？经历了多起安全事件之后，互联网安全架构有何警示？云服务遍地开花，云架构如何能承受爆发式的增长？智能硬件产业方兴未艾，智能设备设计有何玄机？互联网银行蓄势待发，互联网架构跟银行业务又是如何结合？答案尽在ArchSummit北京2015全球架构师峰会。ArchSummit聚集了互联网行业各领域的一线知名架构师及高级技术管理者等，期待与您一起交流承载繁华业务的架构智慧。

主题演讲

梁胜
Rancher Labs创始人
兼全球CEO

袁泳
Uber软件工程师

部分演讲嘉宾（排名不分先后）

陈锐锋
学霸君研发副总裁
如何提供让人耳目一新的在线答疑服务的核心技术？

刘宁
腾讯云安全技术副总监
从甲方到乙方
--腾讯云安全实践之路

李靖
微众银行架构师
微众银行基于自主可控技术的分布式架构实践

黄正强
Marvell研发总监
Wi-Fi SoC 芯片在IoT智能设备中的应用

段念
宜人贷CTO
为行驶中的汽车升级
——快速业务发展环境下的研发团队调优

沈剑
58赶集集团技术总监
58速运-百万单量用车O2O架构实践

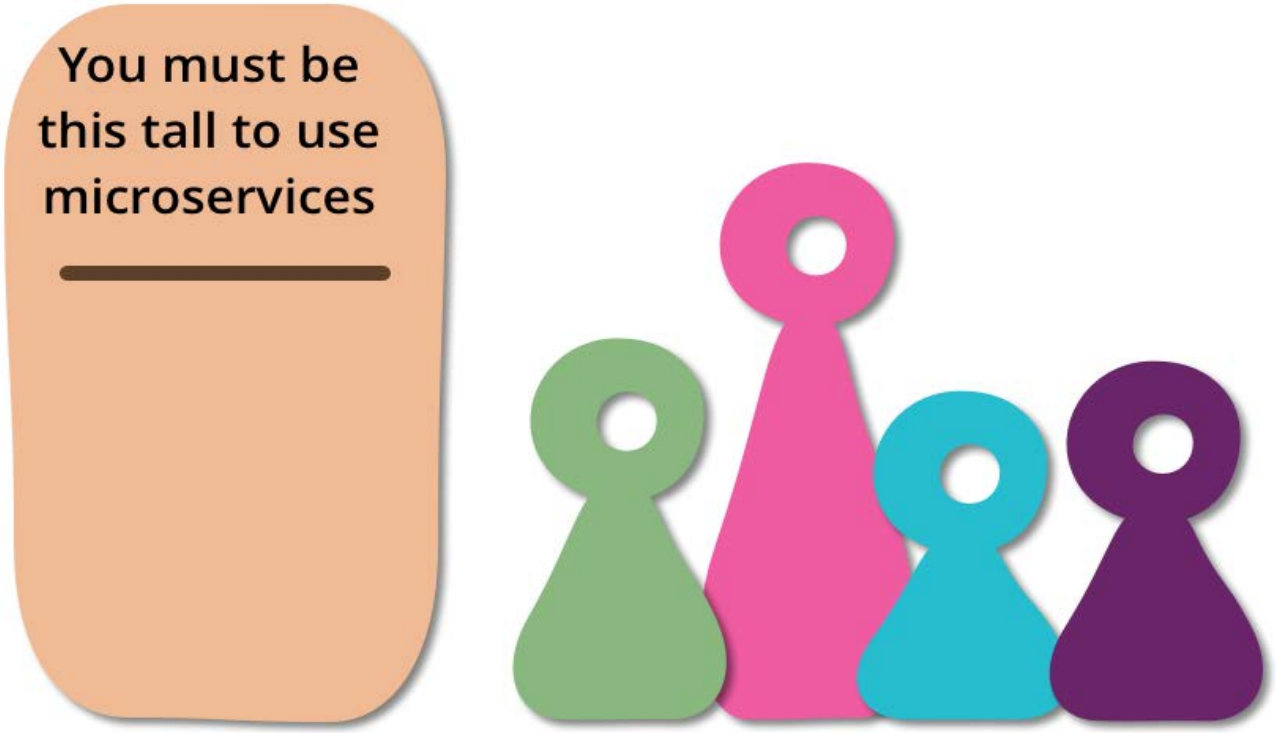
陈浩然
携程移动开发总监
移动应用架构优化之旅

李庆丰
新浪微博高级技术经理
新浪微博高可用服务保障体系演进

垂询电话：010-89880682
QQ 咨询：2332883546
E-mail: arch@cn.infoq.com
更多精彩内容，请持续关注archsummit.com

Brought by Geekbang. 极客邦科技 InfoQ

先把平台做扎实，再来微服务吧



作者 郭蕾

微服务已经成为当下最热门的话题之一。它是一种新的架构风格，涉及组织架构、设计、交付、运维等方面的变革，核心目标是为了解决系统的交付周期，并降低维护成本和研发成本。相比传统的 SOA 架构或者单块架构，微服务有很多的优势，比如技术的多样性、模块化、独立部署等，但也带来了相应的成本，比如运维成本、服务管理成本等。

网上有很多讨论微服务优缺点的文章，而在这个新的东西面前，很多企业不知道如何抉择。那传统的架构有什么挑战？微服务可以为企业带来什么？切换到微服务架构之前，需要做出哪些准备？ 10 月 16 日，在 QCon 上海全球软件架构师大会上，InfoQ 组织了一场闭门会议，深入讨论和交流了微服务相关的问题。阿里巴巴、大众点评、唯品会、华为、普元、中国移动、点融等公司的代表参与了此次讨论，本文根据会议中的精彩观点整理而成。

传统架构面临的挑战

1. **稳定性**。很多公司都是从零开始构建自己的系统的，一开始，为了尽快的实现业务，所以系统的架构也相对比较简单。但当业务和用户量开始大规模增长之后，反过来又对系统的性能和稳定性有较高的要求。而传统的单体式（本文均指 monolithic）应用把所有的服务都堆到一起，可能会由于一些不重要的服务出问题而影响系统的核心服务。另外，系统对服务的要求也不一样，分开部署可以减小它们之间的相互影响。
2. **可维护性**。移动的系统是从 2001 年开始做

的，到现在已经十多年了，这么多年，核心的系统一直是在完善、堆叠。而经过这么多年的维护，系统代码的可维护性非常差，开发人员和技术都已经迭代过几波，系统中代码之间的调用也很多，不是很了解系统的人也不敢轻易动。而由于系统的代码非常多，所以了解系统也不是一件容易的事情。

3. **API 的版本迭代**。单体应用中，一个系统会对应出很多的 API，而根据业务的不断迭代，肯定会衍生出很多版本的 API，特别是现在很多系统都有不同的端（手机、网站）。API 的升级和维护都需要有相应版本的管理，而单体式的应用在这一块，有明显的短板。另外，API 的发布周期也会很慢，因为版本的发布涉及到整体的协调和测试。
4. **持续集成**。单体式的应用由于比较大，应用内部的依赖非常多，涉及的业务逻辑也比较复杂。在 CI 流程中，如果没有很好的约定的话，失败的次数也会比较多。另外，由于代码基比较大，所以构建的时间也会非常长，如果构建错误，排查问题也会比较困难。

对微服务的理解

1. 微服务的概念里，有两个重点，快速发布和解耦。这两点都可以和 OO 中的架构设计原则相对应，一个是单一职责（single responsibility），也就是把一件事情做好，一个是关注分离（Separation of concerns）。康威定律说：设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构。对应到这里，也就是说微服务和公司的组织架构有非常紧密的关系。另外，很多公司都已经是微服务了，只是他们没有提，比如亚马逊、阿里巴巴。像他们这么大的公司，没有微服务根本玩不转。
2. 微服务革新了软件的生产过程，包括开发、测试和部署各个阶段。但服务的切分并不是要遵守单一原则，因为未来的服务不可能完全垂直切分。从另外一个角度看，微服务的过程其实也是工业化的工程，每个微服务都是生产线上的一个零件，但要注意，零件的组装和拼接难度更大，所以在谈论微服务时一定要注意，它是需要一个大平台支撑的。
3. 首先微服务并不是指代码本身，它包括从代码开发到部署到运维这一系列的工作流程。其次，谈到服务拆分，需要注意的是服务的 SLA（Service-Level Agreement）是不一样的，拆分时需要考虑到哪些是一级服务，哪些是二级服务。最底层的服务直接决定了上层服务的稳定性。
4. 服务化说白了是组件化的一种形式，所有的组件都来源于重构。流程是从上往下写的，写到一定程度时，就会遇到分离、解耦的问题，然后就是组件化，这时才会出现重构。不要上来就追时髦，服务化没有价值。
5. 如康威定律所说，系统架构和团队的组织架构有直接的关系。但并不是说为了微服务而调整组织架构，一般是考虑到业务的需求才调整团队组织架构。微服务的团队推荐是 7 个人左右，这也是 Netflix 的最佳实践。另外，团队与团队之间不要有太多依赖。中小型的创业公司不推荐使用微服务，因为开销成本很高，对平台的要求也很高。微服务涉及到你是否能有快速提供环境的能力、文化的改变、快速部署、监控等多方面的能力，当这些条件都具备之后，再考虑是否要微服务。

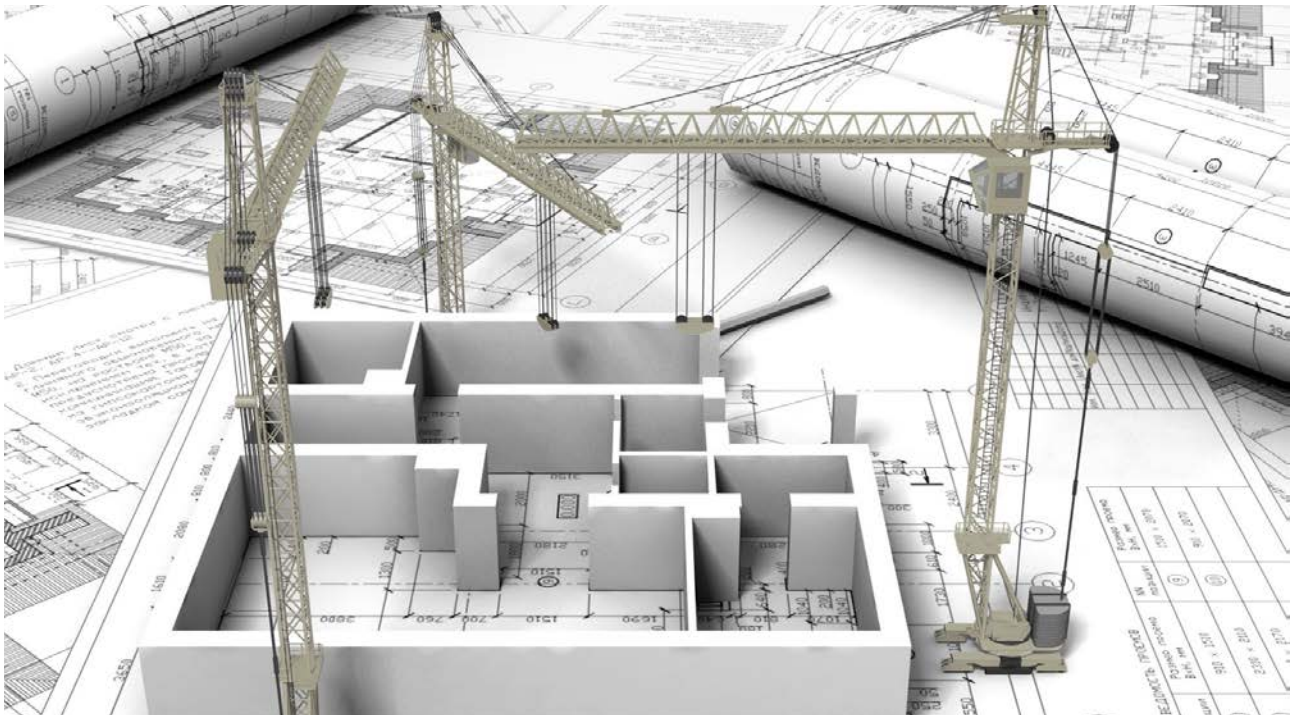
服务治理

1. 服务治理这块，根据经验，有三个需要注意的。一是接口的统一性，当公司发展到几百人的时候，如果没有接口规范，那效率马上就会下降。之前可以像游击队一样打，但人多了之后，还是像正规军一样战

斗力会比较强。二是容错一定要做好，这块可以参考 Netflix 开源的几个组件。容错如果没有做好，在服务化这样的系统里，很可能会造成雪崩效应。容错的方案也有很多，比如限流、回退、隔离、熔断。三是监控，在微服务出错之后，团队需要快速定位出错的位置和原因。

2. 服务治理这块，阿里巴巴有个不错的框架 Dubbo，已经开源。抛开服务治理不说，我来举个服务化的反例，Facebook 内部是反服务化的。Facebook 所有的代码只有两个库，并且所有的发布都是同时进行的，每个机器都一模一样。所以 Facebook 可以做到零运维，并且服务根本没有版本的概念。阿里巴巴现在的服务也准备往回收，在没有服务和过多的服务之间找到一个平衡点。
3. 服务方需要知道服务会被多少人调用，被哪些业务调用，每个时间点上的状态是怎么样的。一个经验是为每个服务都起一个名字，当出现问题时，可以快速定位到。另外，阿里的 eagleye 和点评的 CAT 监控系统，支持对服务调用链和依赖关系进行可视化监控，可以参考学习。
4. 微服务中，服务的测试是一个非常大的挑战。服务和服务之间会存在依赖，所以环境的搭建可能就会涉及到多个团队，这个时候就需要能快速部署环境，当然现在比较推荐的方案是容器。
5. 微服务，其实对应的应该是微业务，是为了适应业务的多变 / 面向最终用户的个性化需求。而微服务的力度，很大程度上是取决于完成一项业务所要设计的数据存储所在的区域。微服务的监控，和原来的服务监控力度应该是类似的，包括业务、应用、系统多个层次，日志、Metrics、调用链、告警等多个维度。
6. 提到监控，大家一般都是从系统、应用等层面去考虑。我这里抛一个思路给大家：舆情监控。很多时候，当系统出问题的时候，我们并不是在监控那里得到消息，而是从微博、微信等渠道中收到用户反馈（XX 系统真垃圾，又挂了之类…..）。我们努力了几年了，就是想让我们的监控系统能在舆论之前监测到系统的异常状态，但很难，所以现在我们也在重点考虑舆情监控。

高可扩展分布式应用程序的架构原则



作者 谢丽

Elastisys 云平台诞生于瑞典默奥大学的分布式系统研究小组。它由一组以预测性扩展引擎为中心的工具组成，可以自动扩展云部署。近日，其官方网站发表了一篇文章，介绍他们在高可扩展分布式应用程序设计和开发方面的经验。

他们将可扩展性分成了如下四个维度：

- 性能可扩展：性能无法完全实现线性扩展，但要尽量使用具有并发性和异步性的组件。具备完成通知功能的工作队列要优于同步连接到数据库。
- 可用性可扩展：CAP 理论表明，分布式系统无法同时提供一致性、可用性和分区容错性保证。许多大规模 Web 应用程序都为了可用性和分区容错性而牺牲了强一致性，而后者则有赖于最终一致性来保证。
- 维护可扩展：软件和服务器都需要维护。在使用平台&工具监控和更新应用程序时，要尽可能地自动化。
- 成本可扩展：总拥有成本包括开发、维护和运营支出。在设计一个系统时，要在重用现有组件和完全新开发组件之间进行权衡。现有组件很少能完全满足需求，但修改现有组件的成本还是可能低于开发一个完全不同的方案。另外，使用符合行业标准的技术使组织更容易聘到专家，而发布独有的开源方案则可能帮助组织从社区中挖掘人才。

以上各项，他们在设计应用程序时都会考虑和

权衡。下面是他们根据上述内容总结出的 10 个设计原则：

1. 避免单点故障：任何东西都要有两个。这增加了成本和复杂度，但却能在可用性和负载性能上获益。而且，这有助于设计者采用一种分布式优先的思维。
2. 横向扩展，而不是纵向扩展：升级服务器（纵向）的成本是指数增长的，而增加另一台商用服务器（横向）的成本是线性增长的。
3. 尽量减少应用程序核心所需要完成的工作。
4. API 优先：将应用程序视为一个提供 API 的服务，而且，不假定服务的客户端类型（手机应用、Web 站点、桌面应用程序）。
5. 总是缓存。
6. 提供尽可能新的数据：用户可能不需要立即看到最新的数据，最终一致性可以带来更高的可用性。
7. 设计时要考虑维护和自动化：不要低估应用程序维护所需要的时间和工作量。软件首次公开发布是一个值得称赞的里程碑，但也标志着真正的工作要开始了。
8. 宁异步，不同步。
9. 努力实现无状态：状态信息要保存在尽可能少的地方，而且要保存在专门设计的组件中。
10. 为故障做好准备：将故障对终端用户的影响最小化。

关于分布式系统的设计，InfoQ 曾有过一些报道（1,2），感兴趣的读者可以对照阅读。

雅虎如何在Hadoop集群上实现大规模分布式深度学习

作者 谢丽

过去十年中，雅虎在 Apache Hadoop 集群构建和扩展方面投入了很大的精力。目前，雅虎有 19 个 Hadoop 集群，其中包含 4 万多台服务器和超过 600PB 的存储。他们在这些集群上开发了大规模机器学习算法，将 Hadoop 集群打造成了雅虎首选的大规模机器学习平台。近日，雅虎 Big ML 团队的 Cyprien Noel、Jun Shi 和 Andy Feng 撰文介绍了雅虎在 Hadoop 集群上构建大规模分布式深度学习的实践。

深度学习（DL）是雅虎的许多产品都需要的功能。比如，Flickr 的场景检测、对象识别、计算审美等功能均依赖于深度学习。为了使更多产品从机器学习中受益，他们最近将 DL 功能引入到了 Hadoop 集群本地。在 Hadoop 上进行深度学习主要有以下好处：

- 深度学习直接在 Hadoop 集群上执行，可以避免数据在 Hadoop 集群和单独的深度学习集群之间移动；
- 同 Hadoop 数据处理和 Spark 机器学习管道一样，深度学习也可以定义为 Apache Oozie 工作流中的一个步骤；
- YARN 可以与深度学习很好地协同，深度学习的多个实验可以在单个集群上同时进行。与传统方法相比，这使得深度学习极其高效。

DL on Hadoop 是一种新型的深度学习方法。为了实现这种方法，雅虎主要做了如下两个方面的工作。

- 增强 Hadoop 集群：他们向 Hadoop 集群添加了 GPU 节点。每个节点有 4 个 Nvidia

Tesla K80 卡，每个卡有 2 个 GK 210 GPU。这些节点的处理能力是传统商用 CPU 节点的 10 倍。GPU 节点有两个独立的网络接口 Ethernet 和 Infiniband。前者作为外部通信接口，后者速度要快 10 倍，用于连接集群中的 GPU 节点以及为通过 RDMA 直接访问 GPU 内存提供支持。借助 YARN 最新推出的节点标记功能，可以在作业中指定容器是在 CPU 上运行还是在 GPU 上运行。

- 创建 Caffe-on-Spark：这是他们基于开源软件库 Apache Spark 和 Caffe 创建的一个分布式综合解决方案。借助它，通过几条简单的命令就可以将深度学习作业提交到 GPU 节点集群，并且可以指定需要启动的 Spark executor 进程数量、分配给每个 executor 的 GPU 数量、训练数据在 HDFS 上的存储位置以及模型的存储路径。用户可以使用标准的 Caffe 配置文件指定 Caffe solver 和深层网络拓扑。Spark on YARN 会启动指定数量的 executor，每个 executor 会分得一个 HDFS 训练数据分区，并启动多个基于 Caffe 的训练线程。

上述工作完成后，他们在两个数据集上对该方法进行了基准测试。在 ImageNet 2012 数据集上的测试显示，与使用一个 GPU 相比，使用 4 个 GPU 仅需要 35% 的时间就可以达到 50% 的准确度。而在 GoogLeNet 数据集上的测试表明，8 个 GPU 达到 60% top-5 准确度的速度是 1 个 GPU 的 6.8 倍。

这说明他们的方法是有效的。为了使 Hadoop 集群上的分布式深度学习更加高效，他们计划继续在 Hadoop、Spark 和 Caffe 上的投入。

雅虎已经将部分代码发布在 GitHub 上，感兴趣的读者可以进一步了解。

环信™
即时通讯云

专为App打造的智能客服平台

客户在哪儿，客户服务就要去哪儿

在经历千万移动端用户同时在线，每天亿级消息量的考验后，环信在成熟的即时通讯平台基础上又打造出新的移动客服平台。
Gartner预测，60%的客服请求将来自移动端。环信为您准备好了全套移动客服解决方案，你准备好了吗？

高效的会话支撑系统

让企业与客户的沟通更顺畅，提升客服代表工作效率

- 自定义信息
- 自动回复
- 快捷回复
- 多客服协作
- 富媒体消息交互
- 接线节奏调整

强大的移动端 SDK

与环信即时通讯 SDK 共享核心代码，历经 2 年研发迭代，2 万家 APP 实际验证

- 欢迎词
- 轨迹跟踪
- 富媒体消息交互
- 常见问题及自动回复
- 极简集成
- 移动端能力极致优化

开放平台

开放的平台，支持差异化的客服业务需求

- 全插件设计
- 开放的接口
- 第三方集成

透明的质检系统

事后的考核评价，实时纠正、协助客服代表

- 实时质检
- 历史会话质检
- 统计报表

精准的客户画像

卓越的用户体验从精准的了解用户开始

- 客户分类
- 轨迹分析
- 会话小结

多渠道接入

以移动端为核心，多渠道接入的统一客户服务平台

- APP 内置客服
- 微信公共帐号
- 微博
- 网页

智能应答机器人

知识库+智能聊天机器人
可以为人工坐席挡住80%的常见问题

- 灵活可定制智能会话、自定义菜单导航功能
- 预置的行业知识库，行业相关常见问答 一键拥有
- 与现有知识库系统对接，机器学习，智能优化知识库
- 人机无缝配合，更少的成本，更好的客户体验

官方微信

服务热线：400-036-8099
www.easemob.com

QINIU

产品 = 七牛云 . 服务(想法)

七大行业解决方案

娱乐　广电　教育　智能硬件　O2O　旅游　安防

七牛四大产品

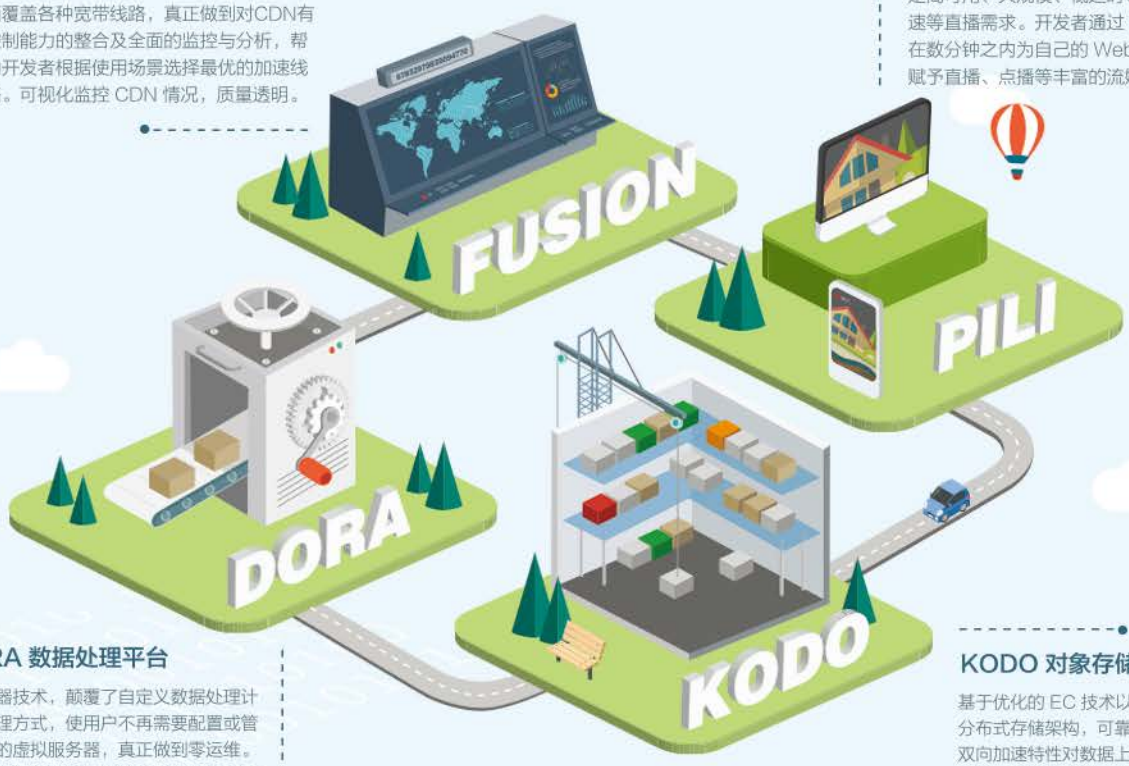
七牛重新定义了云存储

FUSION 融合 CDN 管理平台

持续引入国内外主流 CDN，双向加速，全面覆盖各种宽带线路，真正做到对CDN有控制能力的整合及全面的监控与分析，帮助开发者根据使用场景选择最优的加速线路。可视化监控 CDN 情况，质量透明。

PILI 直播云服务

全球首个用 Go 语言实现的企业级直播流媒体云服务，充分利用七牛的海量网络节点资源满足高可用、大规模、低延时、跨终端和全球加速等直播需求。开发者通过 RESTful API，能在数分钟之内为自己的 Web 和 Mobile App 赋予直播、点播等丰富的流媒体功能。



DORA 数据处理平台

基于容器技术，颠覆了自定义数据处理计算的管理方式，使用户不再需要配置或管理单一的虚拟服务器，真正做到零运维。为用户提供了按需弹性伸缩的计算力和高自由度的开发语言环境，能无缝衔接用户原有的业务技术栈。

KODO 对象存储服务

基于优化的 EC 技术以及七牛自主研发的全分布式存储架构，可靠性高达16个9。首创双向加速特性对数据上传下载均加速，镜像存储、客户端直传、断点续上传等功能，最大程度减少了服务器资源浪费。

SDN & OpenStack漫谈

作者 马力

首先，这片文章初衷是为了自我总结，我不会从感情上偏袒其中某个解决方案，但我确实有自己的Preference和对未来技术发展的看法。

从真正开始参与Neutron开发，已经有近2年的时间，起初的半年内（Icehouse周期里），无时无刻不在修复Neutron在生产环境中发现的各类Bug，其中有一些核心的问题处理比较费时，甚至至今也没有一个很好的解决方案。

1. DB模块的稳定性和性能

众所周知，OpenStack所有的数据持久化（除了Ceilometer外），全部依赖官方推荐的MySQL（包括Percorna、MariaDB等），但是开源关系型数据库引擎的扩展性一直都是问题，从生产环境监控中可以很容易看到，Neutron对数据库的读写压力，在OpenStack所有组件中，仅次于Nova（Keystone可以通过内存数据库来缓解75%以上的压力）。其中主要是三个部分的原因，一方面数据库引擎单点或者HA都无能解决扩展性问题，当Neutron-server进程越来越多时，就会出现，虽然有Galera集群方案，但其事务处理性能一直都是瓶颈，而且乐观锁机制也会导致API失败（Nova有db-retry，Neutron在那个时候还没有）。第二个方面，是SQLAlchemy框架本身的限制，Python社区一直都是松耦合的纯开源社区，所以导致很多企业级应用所必须的框架，没有得到企业级的维护，其中首当其冲就是SQLAlchemy，这个Python世界里排行第一的ORM解决方案，并不是一个优秀的性能出众的ORM方案（至少和我在Java世界里使用的Hibernate，各方面都差了不止一截），记得其在0.90-0.99的

每个版本都存在或多或少的 Bug 没有修复，而且它的 ORM 引擎对 SQL 本身的优化也不是特别到位。第三个方面是 Neutron 社区，一直以来都存在错用 SQLAlchemy 的情况，在事务处理过程中使用 RPC 引起不必要的事务内协程切换，间接增加无谓的数据库引擎长时间维护事务的压力，大规模环境下经常导致事务 Timeout、Deadlock 等情况。这些情况集中于 ML2 和 L3RouterPlugin 插件的 DB 模块，每个 Release 都有大量的 Race Condition 曝出，然后就是 Case by Case 去修复，虽然这种情况自从 Icehouse 曝出多达 10-20 个同样的问题后，慢慢随着 Juno、Kilo 已经修复了大多问题（本人也参与讨论和修复了其中的几个 Bug），但是，由于初期设计失误导致的这种情况，在后期，却需要 200%-300% 的时间去分析和修复，让人汗颜。

2. RPC 系统的稳定性和性能

Neutron 的控制平面是基于 RPC 实现的，官方推荐基于 AMQP0.9 的 RabbitMQ 方案，但是大规模环境下，RabbitMQ 集群的扩展性也成为了障碍，虽然存在 Kernel 调优和 Ram Node-Disk Node 的优化方案，但一方面 RabbitMQ 本身对于 Devops 是黑盒，其次官方的指南在当时并不清晰（后期官方也在不断完善 Best Practice 之类的文档）。为了处理当时的生产环境，切换到了基于 ZeroMQ 的分布式消息队列，在 Neutron 项目中替换了 RabbitMQ 的方案，确有奇效。

关于这个方案，在 Vancouver 峰会上我有一个演讲，详细描述了一个新的分布式消息队列系统。当前，社区除了官方推荐能在小规模 Region 稳定运行的 RabbitMQ，oslo.messaging 团队正在发展三个更有前途的希望能够稳定运行在大规模环境的 RPC 驱动，分别是 ZeroMQ、Kafka、AMQP1.0（记得是基于 Apache Qpid dispatch router 实现的）。

3. Eventlet 本身

Eventlet 是 OpenStack 依赖的 greenthread 管理库，它的优点是开发模型简单，能有效重用 iowait 时的计算资源，缺点是缺少细粒度的调度控制。在 OpenStack 世界里，需要显式调度的时候，需要使用 eventlet.sleep。

但是基于 iowait 的调度粒度太粗。某些情况下，我确实需要在 iowait 下禁止当前协程切换呢？还有些情况下，我需要基于优先级的调度呢？在某些特殊情况下，我需要抢占式调度呢？对于大型系统来说，开发模型简单易学，隐藏了大量细节，并不是好事。

4. 集群资源协调问题

（1）Scheduler

在 Neutron 中，存在一个并不是特别起眼的模块，Scheduler，这个模块负责把 Neutron core Resource 和 Agent 进行绑定和解绑，也就是把资源调度到不同的 Agent 上去配置。这个 Scheduler 目前常用的是 DHCP、L3、LB（负载均衡）。这个模块存在一个较大的缺陷，就是每个调度器都是独立运行，其中没有任何资源协调，导致在生产环境下，我即便是有针对性地实现了更细粒度调度算法，也无法使得不同的调度器间进行交互，实现更高层次的资源统一调度（比如，把一组资源统一调度到同一台 Host，当前无法实现，在写 DB 时会出现 Race Condition）。那个时候，我自己基于 Redis 开发了一个分布式锁管理器，在各个调度器之间实现了统一的调度控制。

（2）AgentGroup

Neutron 目前还没有 AgentGroup 的概念，没有办法实现统一的集群管理和可靠的健康检查机制（RPC+DB，只能算 Demo）。

基于以上两个问题，都是由于Neutron项目内不存在Cluster Coordination机制导致（虽然我实现了一个，但并不通用，当时还没有Tooz），这个机制目前已经有了一个BP（通过Tooz实现，但是只有我觉得可以+1，其他Reviewer并没有反馈），而且其实Nova社区也有了ServiceGroup的参考实现，但是Neutron社区方面却一直Delay，不知道是因为Core Team把握不准，还是觉得这个功能比较复杂，要放到M周期讨论。

5. Agent Loop、External Commands

Neutron中最常用的（User Survey调研占到75%的案例）是Openvswitch和Linuxbridge，其中的Agent框架都是基于Daemon-loop的轮询机制，这套机制在大规模生产环境下，基本不可用，因为一次轮询的时间太长，导致一些需要立即更新的端口配置延时到下一个轮询周期，对于任一端口配置错误都会导致full-sync，系统开销极大。另外，所有对于虚拟网络设备的配置都是通过Subprocess调用外部Command的方式，导致Host Kernel需要维护大量并发的Subprocess，带给Host Kernel很大的压力。幸好在Liberty周期里，已经有人在实现基于事件的Callback机制，并且我也联合另一个Neutron开发者向社区提出了使用Netlink Library Call代替Subprocess的技术方案，还在初期讨论过程中，但是从Icehouse到Liberty，确实够久的。

当然，除了处理Neutron的一系列事故外，我还在思考这么一个问题，就是Neutron的SDN方案，到底该如何发展？当前的情况是，不停地修Bug，平均一个BP的引入会同时带来10+的Bugs（话说回来，DVR的引入带来了30+的Bugs，给HP和Review BP的人狠狠点赞）？OpenStack提供了一个广阔的开放的平台，定义了业界认可的API集合，但是，就如同存储技术需要Domain Knowledge，网络技术同样需要，当前Neutron除了其Plugin框架之外的实现，更多是Reference Design，而不是大规模生产环境下的专用系统。

在这个时候，也有思考过SDN和OpenStack的关系。

（1）Neutron到底是不是实现了SDN？

这个问题不是我抛出的，而是去年就有很多云计算架构师和网络架构师，包括一些业界专家一起在QQ群里论战。其实这个问题，需要从两个角度去看。第一，从云计算技术的角度，Neutron实现了租户网络拓扑的自定义，确实是SDN思想的体现。第二，从网络技术的角度看，Neutron仅仅是实现了网络通路（保证网络是通的，汗），还没有针对流量的细粒度管控，职责也没有非常清晰地分离，与现实世界里大量专业的SDN系统所实现的网络功能相比，确实差了很远，所以要说Neutron是SDN的初级起步阶段也未尝不可，所有特别是很多业界的网络专家对Neutron的实现不屑一顾也情有可原。最新Neutron社区主导的项目，像Dragonflow、RouterVM、ML3，以及与外部SDN开源方案的互动更为频繁（OpenDayLight、OVN等），可以看出OpenStack也在朝好的方向努力。

（2）SDN技术如何定位？

有网络的地方，就可以有SDN发挥的余地，在我眼里，SDN技术所能应用的领域，囊括了整个CT领域，从范围上讲，包含卫星网、全球互联网、骨干网、城域网、园区网、最后100M接入网，局域网；从传输介质上讲，包含双绞线、光纤（单模、多模）、同轴电缆、无线通信（微波、Wifi、2/3/4/5G移动网）所形成的各类网络；当然还包括IT领域的DCN（数据中心网络）、DCI（数据中心互联），NFV领域，以及在未来

DT 领域会大力发展的物联网、传感器网络等等。

云计算网络（比如 OpenStack Neutron）环境仅仅是 SDN 众多北向应用中的一个，不需要谈到 SDN 必谈云。离开了云，SDN 技术还是可以在其它领域发亮发光，反倒离开了 SDN，云就更加虚无缥缈了。

之后 1 年的时间，其实也就是业界各类开源 SDN 解决方案陆续出现的阶段，OpenContrail、Calico、OpenDayLight、ONOS、Midonet、OVN、Neutron DVR、Dragonflow 等。之后我的更多的精力就放在了真正能处理生产环境问题的 SDN 解决方案上。

6. OpenContrail

当时，OpenContrail 的横空出世，确实是 SDN 业界的一颗重磅炸弹。记得该项目刚开源，就开始评估和测试。OpenContrail 确实是一个非常专业的 SDN 解决方案，其架构并无 SPOF，用 MPLS VPN 实现了隔离，包括在那个时候率先设计并实现了基于 Policy 的服务链定义，非常值得学习。不过最终还是放弃跟进，原因如下：

1. 其软件框架纯粹是 Juniper 设计并实现了，没有任何指导开发的文档（门槛确实较高）。
2. 其转发面模块 vRouter 是 Juniper 设计并实现的 Linux 内核模块，虽然只是简单的查询和转发数据包，但当时在 Mailing List 上也有公司测试出有 crash 的现象。当时我特地发消息给社区，建议社区尽量把内核态模块提交给 Linux Upstream，保证代码质量。毕竟 datapath crash 的后果可想而知，没有稳定性，谈何性能。
3. 开源社区并不开放，这个是大多厂商主导的开源社区的通病。
4. 毕竟在创业公司，很难依靠个人之力去维护一套庞大的没有文档的开源系统。
5. 听说当时国内 Juniper 并无对应的技术支持团队，就算企业愿意购买商业版本，也需国外远程支持，说明 Juniper 并没有很好去建设针对企业的服务体系，连企业级支持都没，风险挺大。

这些都是很早之前的评估了，之后 OpenContrail 这个产品发展越来越好，包括也看到其和 Mirantis 合作落地了一些项目，不过还都是以 Contrail 商业版本为主。

7. Calico

这个开源项目是比较有趣的针对 Neutron SDN 的方案，当时投入了一些精力在其开源代码上，受到了很大的启发。Calico 设计的思路其实非常简单直白，它在每台计算节点上安装了一个 BGP Speaker，通过软件实现了 Virtualized L3 Fabric。然后在架构上又引入了 BGP Reflector 解决 full mesh 的问题。虽然 API 是用的 Neutron，但是大多 Neutron extension 都没有实现，也不好实现，它是 Flat Network 的解决方案，当前还不支持 Overlapping-IP，最多实现个安全组、但是完美支持了 IPv6，其架构和 Nova-network multihost 模式更为接近，而且由于其控制平面基于 BGP，计算开销小并且运行可靠，运维成本低，所以在我眼里，他是 Neutron 版本的 L3-based multihost 模式。这个项目设计更多是考虑数据中心网络架构，把 Neutron 的虚拟网络与现实世界的 DCN 实现了整合，是大规模企业级私有云的一个可靠的解决方案。

8. OpenDayLight

OpenDayLight 是一直在跟踪的项目，它是基于动态模块系统构建的插件式网络操作系统（SDN-OS），是我见过的功能最为强大的 SDN 平台（没有之一）。从架构角度，它是纯粹地利用了 Java OSGI 框架实现了一个动态模块系统，各个网络服务模块（无论是南向 plugin

还是北向 plugin）都可以进行热部署，这对于生产环境运维是极大的帮助，因为整个 OSGI 框架不变的基础上，每个网络服务都能做到独立升级和修改，并且动态生效，不影响平台稳定性。而且它设计并实现了南北向交互的服务抽象层（AD-SAL 和 MD-SAL），大大方便了定制开发。唯一的问题在于 AD-SAL 和 MD-SAL 本身，OpenDayLight 项目最早是实现了 AD-SAL，其很多成熟的模块都是基于 AD-SAL 开发的，但是之后，发现设计的 AD-SAL 存在扩展性和代码重用的问题，就重新借助 Yang 模型设计了 MD-SAL，然后希望逐步把 AD-SAL 实现的模块重写变成 MD-SAL 的，费时费力，当前应该还是存在两个不同的 SAL 框架，建议新的插件都以 MD-SAL 为主开发。OpenDayLight 还初步实现了集群功能，基于 Akka 框架实现集群，并且也设计了支持分布式内存数据库，但是其扩展性我没有评估过，不妄下判断。从社区发展角度讲，其代码核心是 Cisco 实现的，之前确实担心 Cisco 独裁，但是目前整个项目全部由独立的基金会操控，包括代码贡献上，Ericsson、Intel、Brocade、Huawei、ETRI 等公司和组织都在持续贡献，而且该开源项目已经纳入了 Linux 基金会的合作项目。从功能实现的角度讲，OpenDayLight 是我用过的第一个纯粹的网络操作系统，兼容并包各类网络技术，包括 OpenFlow1.0 和 1.3、OVSDB、BGP、LISP、Netconf、PCEP、SNMP、OpenDove 等，可以做到统管整个 DCN（数据中心网络基础架构），从上层服务上，提供了包括虚拟多租户、服务链、有线电缆通信服务（DOCSIS）、OpenStack Neutron 服务接口等模块。与 OpenStack 的对接仅仅是其众多应用中的一个，相信其在数据中心网络领域、NFV 领域，乃至三网融合领域，都会取得良好的发展。

最新版本是 Lithium，在这个版本中，第一次看到了完整的使用文档和开发文档，这也是我评价一个开源项目是否值得跟进的一个重要指标（本人痛恨开源社区耍流氓）。目前其案例更多来自 DCN 领域，包括华为和 Brocade 都有基于 OpenDayLight 的数据中心解决方案产品对外在销售，而腾讯的数据中心网络优化，也在使用 OpenDayLight，说明其生产化已经成为可能，但从我的角度讲，确实需要一个专业的网络技术团队作为服务支撑才行。OpenDayLight 生产环境运维，以及基于 OSGI 模型的二次研发，都是需要投入一定的成本的。 当前 OpenDayLight 与 OpenStack 的整合，也在按部就班的进行，并且完全跟随着 OpenStack 社区的研发脚步，完整提供了 ML2 Plugin 和一部分 Service Plugin。话说，Neutron 前 PTL Kile 就是 OpenDayLight 粉，并且在多个公共场合极力推广整合解决方案。

9. ONOS

ONOS 是另一个庞大的开源网络操作系统，它的设计的目标和实现的功能几乎和 OpenDayLight 完全相同，得到了 ONF 组织的大力支持（OpenFlow 标准制定者），是 OpenDayLight 唯一的市场竞争对手（目前两个项目都在试探性地开展技术合作，竞合关系值得期待）。其发展滞后于 OpenDayLight，代码主要由 Huawei、ETRI、 ON.Lab 等公司和组织参与贡献，公司数目和质量还不及 OpenDayLight（Huawei 在开源道路上战略很明确，使用人海战术不放过任何一个可能的发展方向）。它的技术架构都和 OpenDayLight 类似，也是通过 Yang 模型定义其抽象层对象，集群实现的发展从使用 Zookeeper、 Hazelcast 到最后听说选择了 Raft，由于我没有尝试使用过和分析过该系统源码，对其技术不妄下判断，但是从其对 OpenStack 的支持力度上看，还只是 Demo 阶段，似乎 Plugin 也没有完善，希望能尽快看到整合方案，并且从其官网的宣传上看，落地的生产项目也几乎没有。

10. Midonet

Midonet 是去年下半年开源的企业级 OpenStack

SDN 解决方案，也是我介绍的主流方案中唯一一个全球范围内认可的企业级解决方案，毕竟这个是人家 Midokura 公司卖了 2 年的产品，非常成熟，提供了完整的 OpenStack 网络解决方案。

从架构上讲，第一次将分布式 SDN 控制器的概念落地，使用了 Zookeeper 和 Cassandra 作为持久化存储方案（网络拓扑数据库），其控制器分布在每一台计算节点上，实现了一个纯分布式的控制平面，管理接口通过 Restful HTTP Server 实现，数据平面利用了 Openvswitch datapath，通过分布式路由实现东西向的流量调度，通过 BGP 发布外网网段，是一个没有 SPOF 的解决方案，更加精妙的是，其流表的下发完全使用 Proactive 模型，其安全组和防火墙是基于 Ingress Filtering 的设计方案，大大降低了数据网络的无效流量，其还存在一个简单的防 DDoS 模块；另外它实现了 Virtual Single Hop，虚拟网络内任意两点间均只存在单跳。所以从架构、性能、稳定性上，Midonet 都是最适合大规模生产环境使用的 SDN 解决方案，并且已经得到了 OpenStack 业界权威 RedHat 和 Mirantis 的认证支持。但是，它也不是完美的。第一，和 OpenDayLight/ONOS 不同，Midonet 仅支持 OpenStack 和 VMware，无法脱离 OpenStack 和 vSphere 单独使用；第二，其虽然开源，但是从 governance model 看，完全是掌握在 Midokura 公司手里，并且没有指导开发的技术文档（其在 Github 上的开发文档都是完全落后于其当前代码设计的，我在阅读源码过程中发现大量无效内容，让我莫名走了很多的弯路）；第三，它的技术堆栈主体是基于 JVM 的 Scala 语言，任务管理框架基于 Akka，在云计算技术中比较冷门，一般需要较长的学习周期才能适应这门函数式编程语言及其框架。 当然，我和 Midonet 的核心开发人员以及社区管理员都进行过多次面谈和电话交流，他们承诺会尽自己努力构建一个完善的开源生态，希望他们能越走越开放。

11. OVN

OVN 这个项目之所以会抛出，就是因为发现 Neutron 并不适合于作为完整的 SDN 控制器使用，仅适合作为整个虚拟网络层的北向应用（定义 API 接口）。 （就如我之前说的，专业的系统让专业的人去开发，OpenStack 社区做好应用层和生态的管理就行了）。

最终，因为 OVS 在那个时候就已经成为 de facto，具有一定的话语权，所以这个社区也就承担了为 OpenStack 设计并实现一套能大规模生产化使用的网络系统的任务。从技术架构讲，这个项目最初的设计就是参考了 Midonet（当时 Midonet 已经得到了大家的注意），但是由于 OVS 社区是基于 C stack 的，所以架构虽然类似，但却更多利用了已有的 OVS 代码进行改造，比如它的数据持久化方案，完全借助其已经在使用的 OVSDB-server 作为全局的网络拓扑数据库和本地状态存储；部署架构上，每台计算节点上都部署了 ovn-controller 作为本地 SDN 控制进程负责 OpenFlow 和 OVSDB 的通信。

从产品成熟度来讲，毕竟这个是一个全新刚起步的方案，当前还仅仅实现了和 Neutron ML2 的对接，依照它的 Roadmap，明年初就可以完善对接到 Neutron ML2 Plugin+Service Plugins，并且实现分布式路由；另一个方面，OVSDB-server 是一个不支持 HA、Cluster 的单点 NoSQL 数据存储系统，所以除非 OVN 引入更可靠的分布式系统，比如 Zookeeper、Cassandra、Raft，否则肯定无法生产环境使用。还是希望 OpenStack 东京峰会能有更多利好的消息，以及待到明年开始具体进行测试评估工作。然后从社区角度讲，毕竟和 OVS 同源，所以确实是一个值得期待和信任的方案，但是由于其设计初就绑定了 OpenStack，所以和 Midonet 类似，无法扩展到非 CMS（云管理系统）的应用场景。另外，这个项目的核心推

动者之一，还是 Neutron 前 PTL Kile，看来他对 Neutron 当前实现怨念很深啊。

12. Neutron DVR 和 Dragonflow

这两个项目是当前 Neutron 社区设计并实现的分布式路由解决方案，专门解决东西向流量的问题。

（1）DVR

这个是 Neutron 最初的分布式路由方案，由 HP 主导，将 L3-agent 管理的网络拓扑分布到所有的计算节点上，看似很牛，实则纯粹就是个玩具。为什么那么说？因为首先，它的设计并没有创新，而是将整个 L3 的控制和转发面复制到了所有的计算节点，确实开发省心省力，并且设计之初就没有考虑其他高级网络服务对 L3Router 的依赖，导致兼容性问题；其次，运维复杂度上升不止一个数量级，针对虚拟路由器所定义的本地状态（tap, veth peer, router namespace, flow table 等等）原本仅分布在某几台网络节点（L3-agent）上，现在却分布在所有的计算节点上，看似没有了 SPOF，但运维难度并没有降低；然后也是最让人郁闷的是，整个 BP 的推动到代码的编写充斥着不确定性，一方面代码的实现并不优雅，实现过程中都是硬生生地嵌入 Neutron 已有的代码中，之后再发现问题，提交大量的重构去满足 DVR 的落地；该功能的代码写完提交给社区进行测试后，才发现原来和社区已经存在的 L3 HA、甚至与存在了 1 年多的 L2pop 均有兼容性问题；单元测试和功能测试用例极其不完善；每个 Release 都曝出各种 High Level 的 Bugs，据统计大于 30+，所以就如我之前说的，给 HP 和 Review BP 的人狠狠点赞，各种刷 Commits 和 Reviews 的机会啊。最后从 SDN 技术发展的角度上说，DVR 的推动是 Neutron SDN 的退步，它虽然解决了东西向流量集中路由的问题，但是在 packet tracing 没有可靠工具的前提下，不仅其增加了 datapath 的复杂度，间接导致增加了运维的复杂度，提高了企业网络运维的隐形成本，与利用 SDN 技术降低 OpEx 的初衷相反。

（2）Dragonflow

这个是 Neutron 社区之后推动的第二个分布式路由方案，完全由 Huawei 主导，OpenFlow Reactive 的设计思路，利用纯流表实现的分布式路由，设计比 DVR 优雅太多，我也没有时间去做测试评估，但是从设计文档看，确实非常清晰，但由于一方面该项目参与的开发者并不多，另一方面项目起步也比较晚、到目前社区 CI 系统还没能支持该项目的集成测试，所以当前是否能直接上生产，我持保留意见，反正东京峰会快要举行了，在峰会上应该会有一些确定的结论。最后要说明 Dragonflow 只是替代 L3-agent 的方案，包括与 FWaaS、VPNaaS 的兼容性问题，更重要是首包上送的性能，还有待生产环境去考证。所以，和 DVR 一样，它的应用范围实在有限，不是一个完整的 SDN 解决方案。

本文纯属个人观点，请自由拍砖，但本人仅关注在文章中技术层面存在严重失误的拍砖。

作者简介

马力，海云捷迅（AWcloud）架构师，关注领域：大规模云计算架构、数据中心基础架构、SDN、OpenStack，Email：mali@awcloud.com

亚马逊 aws | InfoQ

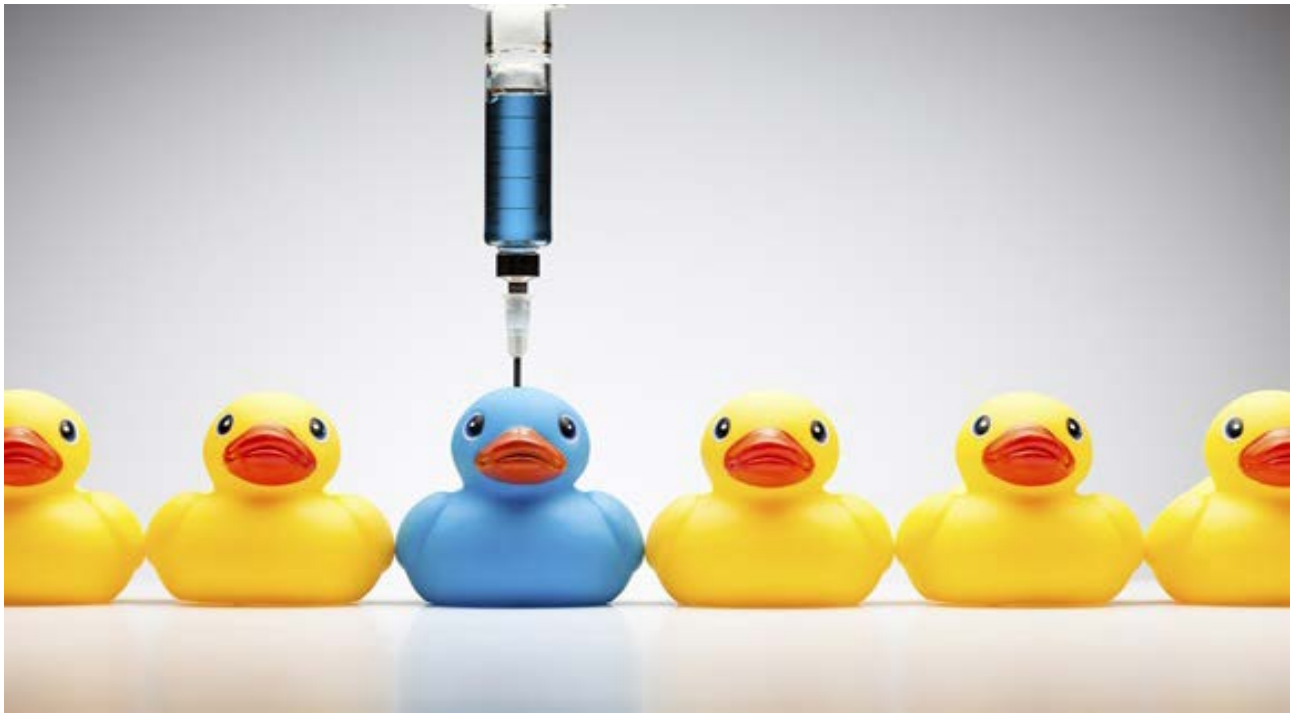
InfoQ在线课堂

走近AWS IoT: 技术解析

亚马逊物联网云服务平台

2015年11月17日（周二）20:00-21:00

反模式的经典——Mockito设计解析



作者 吴以均

测试驱动的开发（Test Driven Design, TDD）要求我们先写单元测试，再写实现代码。在写单元测试的过程中，一个很普遍的问题是，要测试的类会有很多依赖，这些依赖的类 / 对象 / 资源又会有别的依赖，从而形成一个大的依赖树，要在单元测试的环境中完整地构建这样的依赖，是一件很困难的事情。

所幸，我们有一个应对这个问题的办法：Mock。简单地说就是对测试的类所依赖的其他类和对象，进行 mock — 构建它们的一个假的对象，定义这些假对象上的行为，然后提供给被测试对象使用。被测试对象像使用真的对象一样使用它们。用这种方式，我们可以把测试的目标限定于被测试对象本身，就如同在被测试对象周围做了一个划断，形成了一个尽量小的被测试目标。关于Mock在单元测试中的作用，Martin Fowler有过专门的叙述。

Mockito的设计

Mock的框架有很多，最为知名的一个是Mockito，这是一个开源项目，使用广泛。官网。示例代码见下。

代码中的注释描述了代码的逻辑：先创建mock对象，然后设置mock对象上的方法get，指定当get方法被调用，并且参数为0的时候，返回”one”；然后，调用被测试方法（被测试方法会调用mock对象的get方法）；最后进行验证。逻辑很好理解，但是初次看到这个代码的人，会觉得有点儿奇怪，总感觉这个代码跟一般的代码不太一样。

```
001 import org.mockito.Mockito;
002
003 // 创建mock对象
004 List mockedList = Mockito.mock(List.
      class);
005
006 // 设置mock对象的行为 — 当调用其get方法获
      取第0个元素时，返回”one”
007 Mockito.when(mockedList.get(0)).
      thenReturn(“one”);
008
009 // 使用mock对象 — 会返回前面设置好的
      值”one”，即便列表实际上是空的
010 String str = mockedList.get(0);
011
012 Assert.assertTrue(“one”.equals(str));
013 Assert.assertTrue(mockedList.size() ==
      0);
014
015 // 验证mock对象的get方法被调用过，而且调用
      时传的参数是0
016 Mockito.verify(mockedList).get(0);
```

让我们仔细想想看，下面这个代码：

```
001 // 设置mock对象的行为 — 当调用其get方法获
      取第0个元素时，返回”one”
002 Mockito.when(mockedList.get(0)).
      thenReturn(“one”);
```

如果按照一般代码的思路去理解，是要做这么一件事：调用 mockedList.get 方法，传入 0 作为参数，然后得到其返回值（一个 object），然后再把这个返回值传给 when 方法，然后针对 when 方法的返回值，调用 thenReturn。好像有点不通？ mockedList.get(0) 的结果，语义上是 mockedList 的一个元素，这个元素传给 when 是表示什么意思？所以，我们不能按照寻常的思路去理解这段代码。实际上这段代码要做的是描述这么一件事情：当 mockedList 的 get 方法被调用，并且参数的值是 0 的时候，返回”one”。很不寻常，对吗？如果用平常的面向对象的思想来设计 API 来做同样的事情，估计结果如下面代码所示。第一个参数描述要返回的结果，第二个参数指定 mock 对象，第三个参数指定 mock 方法，后面的参数指定 mock 方法的参数值。这样的代码，更符合我们看一般代码时候的思路。

```
001 Mockito.returnValueWhen(“one”,
      mockedList, “get”, 0);
```

第一个参数描述要返回的结果，第二个参数指定 mock 对象，第三个参数指定 mock 方法，后面的参数指定 mock 方法的参数值。这样的代码，更符合我们看一般代码时候的思路。

但是，把上面的代码跟 Mockito 的代码进行比较，我们会发现，我们的代码有几个问题：

1. 不够直观
2. 对重构不友好

第二点尤其重要。想象一下，如果我们要做重构，把 get 方法改名叫 fetch 方法，那我们要把”get”字符串替换成”fetch”，而字符串替换没有编译器的支持，需要手工去做，或者查找替换，很容易出错。而 Mockito 使用的是方法调用，对方法的改名，可以用编译器支持的重构来进行，更加方便可靠。

实际上，Mockito 的设计还有很多其他的好处，Mockito 的作者写了一篇文章描述它背后的设计思想。

实现分析

明确了 Mockito 的方案更好之后，我们来看看 Mockito 的方案是如何实现的。首先我们要知道，Mock 对象这件事情，本质上是一个 Proxy 模式的应用。Proxy 模式说的是，在一个真实对象前面，提供一个 proxy 对象，所有对真实对象的调用，都先经过 proxy 对象，然后由 proxy 对象根据情况，决定相应的处理，它可以直接做一个自己的处理，也可以再调用真实对象对应的方法。Proxy 对象对调用者来说，

可以是透明的，也可以是不透明的。

Java 本身提供了构建 Proxy 对象的 API：Java Dynamic Proxy API。Mockito 就是用 Java 提供的 Dynamic Proxy API 来实现的。

下面我们来看看，到底如何实现文章开头的示例中的 API。如果我们仔细分析，就会发现，示例代码最难理解的部分是建立 Mock 对象(proxy 对象)，并配置好 mock 方法（指定其在什么情况下返回什么值）。只要设置好了这些信息，后续的验证是比较容易理解的，因为所有的方法调用都经过了 proxy 对象，proxy 对象可以记录所有调用的信息，供验证的时候去检查。下面我们重点关注 stub 配置的部分，也就是我们前面提到过的这一句代码：

```
001 // 设置mock对象的行为 — 当调用其get方法获取第0个元素时，返回”one”
002 Mockito.when(mockedList.get(0)).
      thenReturn(“one”);
```

当 when 方法被调用的时候，它实际上是没有办法获取到 mockedList 上调用的方法的名字(get)，也没有办法获取到调用时候的参数(0)，它只能获得 mockedList.get 方法调用后的返回值，而根本无法知道这个返回值是通过什么过程得到的。这就是普通的 java 代码。为了验证我们的想法，我们实际上可以把它重构成下面的样子，不改变它的功能：

```
001 // 设置mock对象的行为 — 当调用其get方法获取第0个元素时，返回”one”
002 String str = mockedList.get(0);
003 Mockito.when(str).thenReturn(“one”);
```

这对 Java 开发者来说是常识，那么这个常识对 Mockito 是否还有效呢。我们把上面的代码放到 Mockito 测试中实际跑一遍，结果跟前面的写法是一样的，证明了常识依然有效。

有了上面的分析，我们基本上可以猜出来 Mockito 是使用什么方式来传递信息了 —— 不是用方法的返回值，而是用某种全局的变量。当 get 方法被调用的时候（调用的实际上是 proxy 对象的 get 方法），代码实际上保存了被调用的方法名（get），以及调用时候传递的参数（0），然后等到 thenReturn 方法被调用的时候，再把”one”保存起来，这样，就有了构建一个 stub 方法所需的所有信息，就可以构建一个 stub 方法了。

上面的设想是否正确呢？Mockito 是开源项目，我们可以从代码当中验证我们的想法。下面是 MockHandlerImpl.handle() 方法的代码。代码来自 Mockito 在 Github 上的代码。

注意第 1 行，第 6-9 行，可以看到方法调用的信息(invocation)对象被用来构造 invocationMatcher 对象，然后在第 19-21 行，invocationMatcher 对象最终传递给了 ongoingStubbing 对象。完成了 stub 信息的保存。这里我们忽略了 thenReturn 部分的处理。有兴趣的同学可以自己看代码研究。

看到这里，我们可以得出结论，mockedList 对象的 get 方法的实际处理函数是一个 proxy 对象的方法（最终调用 MockHandlerImpl.handle 方法），这个 handle 方法除了 return 返回值之外，还做了大量的处理，保存了 stub 方法的调用信息，以便之后可以构建 stub。

总结

通过以上的分析我们可以看到，Mockito 在设计时实际上有意地使用了方法的“副作用”，在返回值之外，还保存了方法调用的信息，进而在最后利用这些信息，构建出一个 mock。而这些信息的保存，是对 Mockito 的用户完全透明的。这是一个经典的“反模式”的使用案例。“模式”告诉我们，在设计方法的时候，应该避免副作用，一个方法在被调用时候，除了

```
001 public Object handle(Invocation invocation) throws Throwable {
002      if (invocationContainerImpl.hasAnswersForStubbing()) {
003          ...
004     }
005      ...
006      InvocationMatcher invocationMatcher = matchersBinder.bindMatchers(
007             mockingProgress.getArgumentMatcherStorage(),
008            invocation
009     );
010    mockingProgress.validateState();
011     // if verificationMode is not null then someone is doing verify()
012    if (verificationMode != null) {
013    ...
014       }
015     // prepare invocation for stubbing   invocationContainerImpl.
     setInvocationForPotentialStubbing(invocationMatcher);
016   OngoingStubbingImpl<T> ongoingStubbing =
017   new OngoingStubbingImpl<T>(invocationContainerImpl);
018 mockingProgress.reportOngoingStubbing(ongoingStubbing);
019    ...
020 }
```

return 返回值之外，不应该产生其他的状态改变，尤其不应该有“意料之外”的改变。但 Mockito 完全违反了这个原则，Mockito 的静态方法 Mockito.anyString()，mockInstance.method()，Mockito.when()，thenReturn()，这些方法，在背后都有很大的“副作用”——保存了调用者的信息，然后利用这些信息去完成任务。这就是为什么 Mockito 的代码一开始会让人觉得奇怪的原因，因为我们平时不这样写代码。

然而，作为一个 Mocking 框架，这个“反模式”的应用实际上是一个好的设计。就像我们前面看到的，它带来了非常简单的 API，以及编译安全，可重构等优良特性。违反直觉的方法调用，在明白其原理和一段时间的熟悉之后，也显得非常的自然了。设计的原则，终究是为设计目标服务的，原则在总结出来之后，不应该成为僵硬的教条，根据需求灵活地应用这些原则，才能达成好的设计。在这方面，Mockito 堪称一个经典案例。

参考资料

- Mockito - A Simple, Intuitive Mocking Framework
- Mockito: drink it without moderation
- Mockito 作者自己的话
- Explanation how proxy based Mock Frameworks work
- How does mockito when() invocation work?
- Explore the Dynamic Proxy API
- Mockito for Spring
- Mockto 2.0 doc

作者简介

吴以均，浙江大学硕士，专注于 Java 服务端的开发，在 IBM，唯品会等公司从事后端开发工作多年。理念是用技术的手段解决“非技术”的问题。对于能够提升开发人员的效率和改进开发流程的技术抱有极大的兴趣。

TalkingData如何构建交互型分析系统

作者 周海鹏

统计型分析系统 Vs. 交互型分析系统

首先讲讲我对认知、思维的理解：逻辑思维的方式无非以下几种：抽象与概括、分析与综合，归纳与演绎，对比（求同、求异），原因与结果（正推：原因推理结果，逆推：结果推理原因，因果链：原因产生结果，结果作为原因产生下一个结果。）

传统的统计型分析系统，一般会对数据进行收集、清洗、计算、展现。这个过程属于抽象和概况的过程。从传统的 BI（基于 Relation DB）到如今的新一代大数据工具（例如基于 Hadoop 的 Hive），基本能完成上述加工过程。

但是随着业务的不断发展，客户提出了更多的问题。当仅仅以查看固定的报表需求被满足后，客户需要更进一步了解数据，更进一步能动态的分析数据。数据分析师的需求从统计型系统进化到分析型系统。这也是人类认知的更高层次的天然需求。这种天然需求，在认知、思维范畴，就是“分析与综合、归纳与演绎”，我们可以将这种对数据的探索，称为交互式查询。**目的是通过对数据的不断咀嚼，逐步从数据中提炼出知识，推理得到结论**。例如：数据分析师需要对比能力（求同、求异），在业务中，利用这种能力在时间维度上对比、在渠道上对比，分析时间、渠道对产品、服务的影响；业务分析人员需要了解原因与结果、因果链：例如在广告领域期待了解广告点击的心理原因、环境原因，以提高广告转化率。

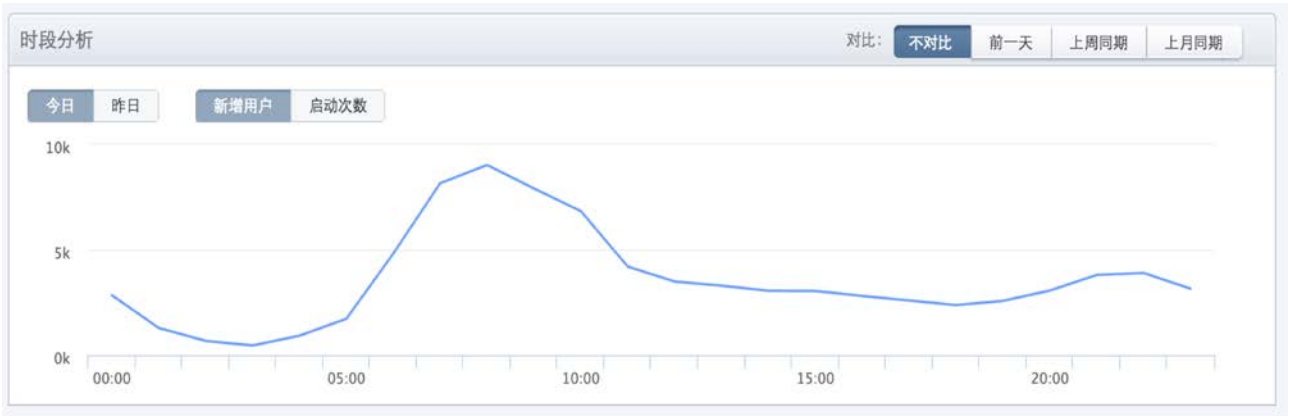


图 1

交互型分析系统的核心特征

像仪表板（Dashboard）和记分卡（Scorecard）的使用，一般是为了提供快速、直观的、对特定的业务领域的，高度汇聚信息的数据。用途和目的是让核心业务人员，从影响业务的、整体的角度，有效的监视业务的状态。交互式查询需要从海量数据中提炼出这些核心数据，那么交互式查询系统必须拥有以下特征：

- 可以灵活的执行上卷、下钻、横向、纵向切分的功能：这可以使业务人员，可以透过表面的图表、图形、趋势和数字，向更深的信息下潜，可以更好的综合观察数据的细节，以便更深理解数据的本质，甚至可以找到更有趣的度量、更隐秘的答案。
- 可以附加业务和技术元数据、并提供上下文的线索，这将有助于更好地了解各种指标、数据信息来源。将多种信息汇总，目前计算机系统还不能像人类大脑一样纵横捭阖，因此在交互式查询过程中加入人类的智能活动，就是必须的能力。
- 收集的数据容易，收集大数据也不难，但分析大数据就不那么容易了。一个交互式查询系统需要汇聚企业中各种数据来源到一个数据载体中，如企业数据库，社交媒体，传感器数据等，同时又有能力提供搜索、计算能力，以图标、图形、数字等方式展现信息。

如何利用 Elastic Search 构建交互型分析系统

总体架构如下图 2 所示。

- TalkingData 技术团队从 2013 年开始，构建了基于 Kafka 的 DSB（Data Service Bus），以获得高吞吐、可复用的数据传输能力。关于 Kafka，这里不再赘述，可以参考相应的文档（http:// kafka.apache.org）。
- river 代表 Elastic Search 的一个数据源，也是其它存储方式（如：数据库）同步数据到 Elastic Search 的一个方法。它是以插件方式存在的一个 Elastic Search 服务，通过读取 river 中的数据并把它索引到 Elastic Search 中，官方的 river 有 couchDB 的，RabbitMQ 的，Twitter 的，Wikipedia 的。 在 river 插件中，有我们急需的 Kafka 插件（https://github.com/endgameinc/elasticsearch-river-kafka）可以先安装插件到 Elastic Search，完成配置文件，用 curl 命令提交任务到 Elastic Search，这样 Kafka 和 Elastic Search 就可以完成数据集成。配置的方法见图 3。
- Elastic Search head（https://github.com/mobz/elasticsearch-head） 是一个 Elastic Search 的集群管理工具，它是完全由 html5 编写的独立网页程序，可以通过插件把它集成到 Elastic Search，进而

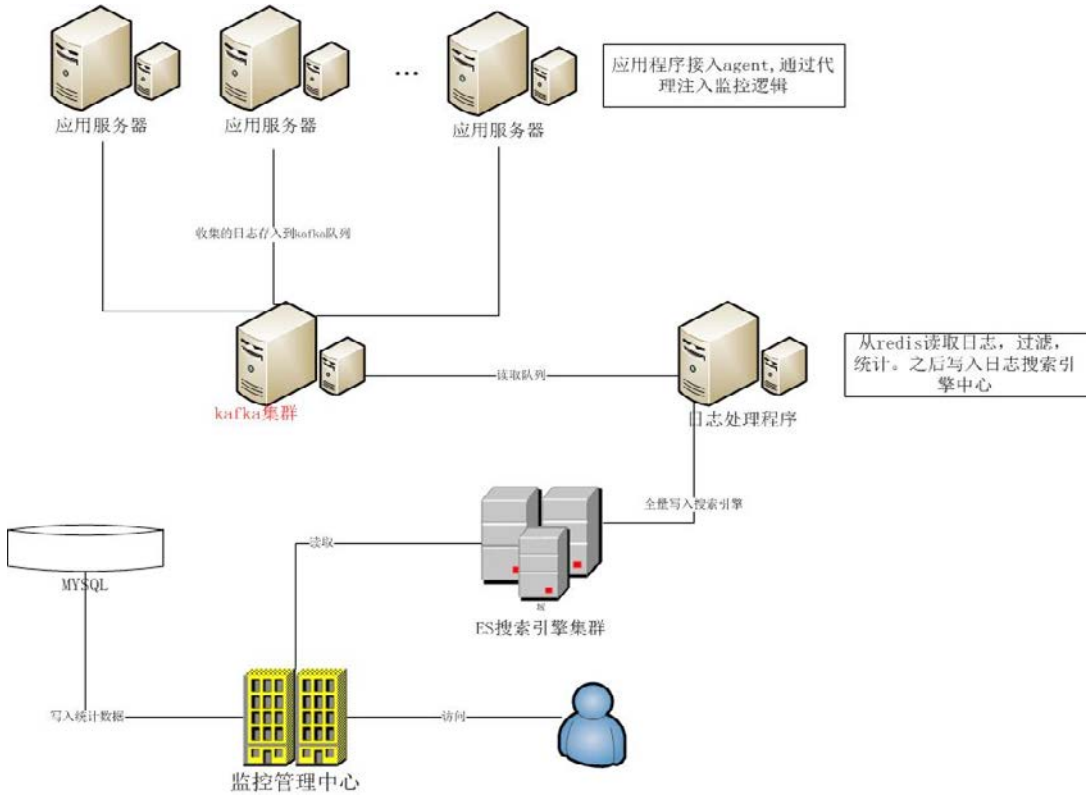


图 2

```
curl -XPUT 'localhost:9200/_river/<river-name>/_meta' -d '
{
    "type" : "kafka",
    "kafka" : {
        "zookeeper.connect" : <zookeeper.connect>,
        "zookeeper.connection.timeout.ms" : <zookeeper.connection.timeout.ms>,
        "topic" : <topic.name>,
        "message.type" : <message.type>
    },
    "index" : {
        "index" : <index.name>,
        "type" : <mapping.type.name>,
        "bulk.size" : <bulk.size>,
        "concurrent.requests" : <concurrent.requests>,
        "action.type" : <action.type>,
        "flush.interval" : <flush.interval>
    },
    "statsd" : {
        "host" : <statsd.host>,
        "prefix" : <statsd.prefix>,
        "port" : <statsd.port>,
        "log.interval" : <statsd.log.interval>
    }
}'
```

图 3

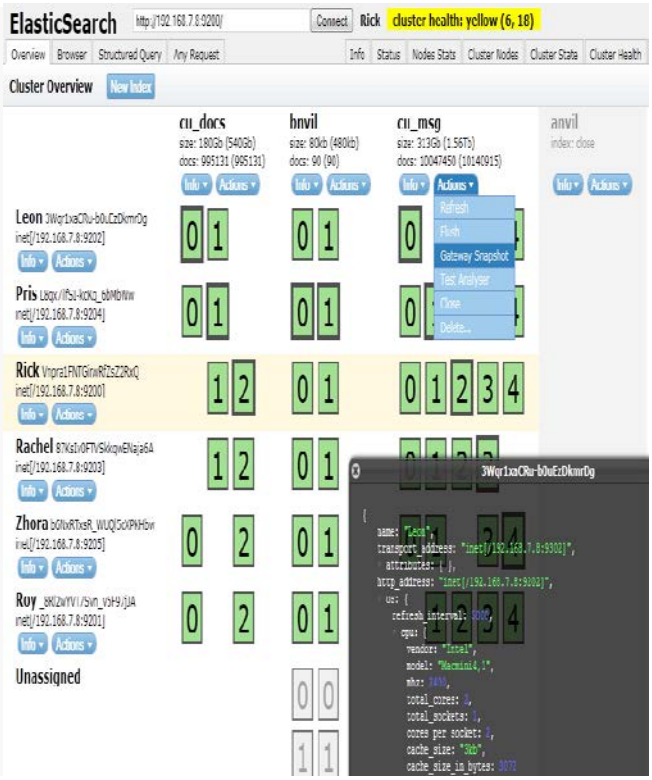


图 4

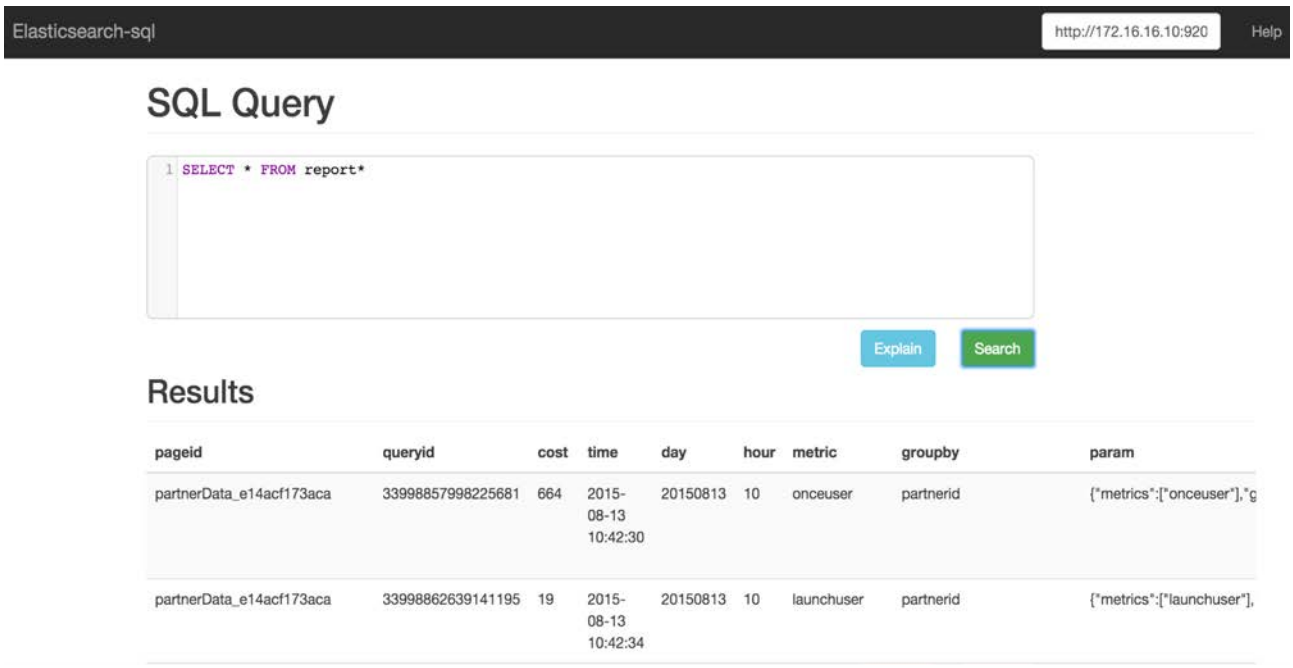


图 5

可以通过该工具对集群索引、状态等进行管理。（见图 4）

- 虽然 Elastic Search 有非常好的 API 体系，但是要想理解清楚这些 API，并基于这些 API 开发出系统还是有些难度的。因此，TalkingData 技术团队为了让有一定技术能力（SQL）的数据分析人员可以尽快使用 Elastic Search 对数据进行分析，引进了 Elastic Search SQL 插件（https://github.com/NLPchina/elasticsearch-sql/），见图 5。

成功案例

经过以上的配置，我们很轻松就建立起一个可以进行交互式查询、可以方便管理、监控的 Elastic Search 环境。我们可以通过一个案例来证实一下交互式查询对解决问题的巨大帮助。

在 TalkingData App Analytics 1.0 系统中，我们基于 Hive、Mysql 实现了“App 错误报告”的功能：可以统计异常的出现次数、时间、机型等等。但是客户在使用后提出这样的疑问：我们能不能对“异常堆栈”中的信息进行细分的查询，例如同样是“Null point Exception”（空指针）这样的问题，希望可以进一步查询造成这样问题的原因。在传统分析中（例如基于 Mysql），会基于“Like”这样的语义进行分析。但是海量数据下，这种方式完全不可行。在 TalkingData App Analytics 2.0 中我们用 Elastic Search 完成了这样的功能，见图 6。

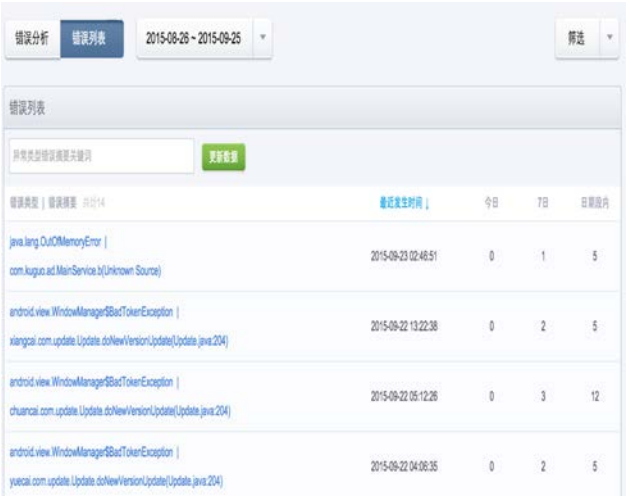


图 6

通过使用 Elastic Search 的交互式查询能力，我们可以对 Exception 进行不断的挖掘，直到挖到真正导致系统问题的开发原因。

UCLOUD

数据方舟

时间的任意门,让数据失而复得

12小时内任意1秒

24小时内任意1时

3日以内任意0点

InfoQ 中文站　2015迷你书

架构师 月刊 2015年10月

本期主要内容：作为一名Java程序员，我为什么不在生产项目中转向Go，LinkedIn是如何优化Kafka的，深入浅出React（四）：虚拟DOMDiff算法解析，Apache Calcite：Hadoop中新型大数据查询引擎，专访阿里钱磊：如何打造第三方的开放账号体系，OPPO Monitor Platform：从应用请求到后端处理，自研解决服务化架构系统监控难题

开源启示录 第二季

开源软件的未来在于建立一个良性循环，以参与促进繁荣，以繁荣促进参与。在这里，我们为大家呈现本期迷你书，在揭示些许开源软件规律的之外，更希望看到有更多人和企业参与到开源软件中来。

顶尖技术团队访谈录 第三季

本次的《中国顶尖技术团队访谈录》·第三季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。

云生态专刊

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