

架构师 ARCHITECT



本期专题 : Docker

Docker 核心技术预览

Docker 命令行探秘

Docker 开源之路

推荐文章 | Article

Spark 的硬件配置

高密度 Java 应用部署的一些实践

特别专栏 | Column

豌豆荚质量总监分享云计算演进之路

腾讯大数据平台纵览

一些好的规则



卷首语：Docker，明天的明天会更好

在不到一年的时间里，Docker 已经家喻户晓，并相继得到了谷歌、微软、红帽、IBM 等大佬的支持。为什么一个容器能得到大家这么高的关注？前几天和一个朋友聊天，他说整个圈子都在吹捧 Docker，虽然实现 Docker 的愿景还有很长的路要走，但在这个高速发展的时代指不定哪天又会出来下一个“Docker”。Docker 要大规模在生产环境使用还需要一段时间，但是我坚信 Docker 一定是一次技术上的大的变革，原因如下：

- 商业运作，社区活跃：Docker 公司是以 dotCloud 起家的，为了能够把所有精力都集中到 Docker 身上，Docker 公司不惜卖掉了发展同样不错的 dotCloud，可见 Docker 公司的决心。今年 8 月，Docker 融资 4000 万美元以上，目前估值达 4 亿美元。有决心、有胆量、不缺钱、方向明确，这使我找不到一个理由来否定 Docker。另外，与 Docker 相关的社区同样非常活跃，Docker 1.0 发布时，官方公开的数据显示超过 460 位社区开发者参与了 Docker 的开发，现在可能会更多。
- 受众群体庞大，使用场景多：Docker 如此之火的原因之一就是它的受众群体大，可以应用到多个场景。比如在交付方面，Docker 可以极大地减少交付的时间成本和人力成本，传统的软件交付方式是复制、配置、运行，而 Docker 的交付方式是复制、运行，通过使用 Docker，可以避免因为环境而引起的程序问题。在 PaaS 方面，大家都知道 PaaS 的一大软肋就是很多软件不能无缝迁移到 PaaS 平台，有了 Docker，这一弊端便迎刃而解了。随着 Docker 安全性的日益完善，相信 Docker 未来会大规模应用到 PaaS 中。IaaS 方面，由于 Docker 之间内核共享，想做到彻底的安全是不可能的，所以这块我看好企业内部的 IaaS 平台使用 Docker。
- 操作系统级别厂商的支持：Docker 之后，CoreOS 也火了，同样拿到了不少融资。CoreOS 是一个轻量级的针对大规模服务器部署而优化的 Linux 发行版，它通过使用 Docker 来减少维护和管理 Linux 软件堆栈所带来的麻烦，可以说欲借 Docker 的春风革传统企业级操作系统的命。8 月，红帽在战战兢兢中发布了 Atomic 项目，也是一个用于运行 Docker 容器的轻量级系统。不管是谁革了谁的命，总之他们都视 Docker 为亲爹。
- 生态圈日益成熟 Docker 相关的开源项目越来越多，涉及 PaaS、CI、仓库托管、图形界面、管理工具等多个领域，每个领域都有与之对应的开源项目，并且社区关注度都很高。Kubernetes、Rudder、Panamax 等都是由大公司组织维护的，出发点和质量都很好。比如谷歌的 Kubernetes 已经得到多个云平台的支持，CenturyLink 的 Panamax 可以让用户通过图形界面来管理 Docker，CoreOS 的 Rudder 更是解决了集群中复杂的网络配置问题。同时，与 Docker 相关的创业公司也开始多起来了，比如 Quay.io、StackDock、Orchard。众人拾柴火焰高，这么多的人参与 Docker，火焰能不高么？

卷首语：Docker，明天的明天会更好

美团云的技术演变：先把云主机做稳定了再说别的

携程首席架构师谈 DevOps：找到合适的人最重要

ThoughtWorks 技术雷达 2014 年 7 月刊：JavaScript、微服务和去中心化的技术趋势

为什么 ZeroMQ 不应该成为你的第一选择

本期专题：深入浅出 Docker 系列

Docker 核心技术预览

Docker 命令行探秘

Docker 开源之路

HTML5、Web 引擎与跨平台移动 App 开发

Spark 的硬件配置

通过度量查询质量构建更佳的搜索引擎

高密度 Java 应用部署的一些实践

豌豆荚质量总监分享：从自建机房到云计算的演进之路

腾讯大数据平台纵览

有关云架构建设和选型的思考

一些好的规则

岑文初谈移动端开放插件平台的技术难点

Whitepages 的架构变迁：从 Ruby 到响应性更好的 Scala 和 Akka

封面植物：不知名的丽花球

美团云的技术演变：先把云主机做稳定了再说别的

作者 唐君毅 邱剑 朱晏

2013年上半年，美团发布了其公有云服务美团云。该产品一开始的背后支撑团队是美团系统运维组，到2014年6月，美团云业务部从系统运维组独立了出来，专门负责云计算方面的产品研发与运营。

近日，InfoQ中文站编辑与美团云业务部的三位工程师进行了交流，了解美团云目前的状态、过去的演变历程和下一步发展计划。以下内容根据这次交流整理而成。

背景

独立出来的美团云业务部目前有十几位工程师。虽然部门独立，但工作中仍然跟系统运维组有紧密的配合。美团系统运维组原本就有三分之二的开发工程师，而运维工程师也全部具备编写代码的能力，因此开发与运维工程师能够进行紧密的配合。

美团云的最初版本起步于2012年7月，一开始是作为私有云计算平台来构建。第一版开发了大约2个月的时间，之后用了大概10个月的时间将美团的所有业务迁移到云平台上。现在除了Hadoop、数据库仍跑在物理机上之外，美团网的所有业务都已经运行在美团云上，内部的研发、测试平台也运行在美团内部的办公云平台上。

2013年5月，美团云开始对外提供公有云服务，截止到目前仅对外提供云主机产品。对象存储、Redis、MySQL、负载均衡、监控、VPC等服务也在研发，部分产品已经在内部以及部分测试客户使用，未来会根据产品的成熟度和客户的需求程度逐步对外开放。

稳定是公有云服务的核心价值。自从公有云服务开放以来，美团云主要的工作都放在提升云主机的稳定性，完善云主机模板、备份、监控、安全等工作上。预计在2014年9月，美团云将发布其第三个版本的更新，继续打磨其产品细节。

技术架构

美团云最初选型的时候对OpenStack、CloudStack、Eucalyptus都做过调研，调研的结果是架构设计使用OpenStack的框架，网络架构参考CloudStack，主要组件由自己开发，部分组件在OpenStack原生组件上进行了二次开发。

- 核心的云主机管理系统是自己研发，没有使用Nova。采用Region-Zone-Cluster三层架构，支持跨地域、多数据中心的大规模集群部署。采用了基于KVM的主机虚拟化和基于OpenVSwitch+OpenFlow的网络虚拟化技术。
- 镜像管理用了Glance。有一定修改，例如，多数据中心分布式支持，以及镜像替换。
- 身份管理用了Keystone。有一定修改，例如，高并发性能改进，与美团帐户系统的集成。
- 对象存储用了Swift，但Swift在写延迟方面存在性能问题，同时在OAM方面的功能比较薄弱，所以也做了一些修改和研发。Swift现在已经在为美团网内部存储量几十TB量级的业务提供服务。

之所以没有整个引入OpenStack，是因为当时调研时，感觉OpenStack的设计比较脱离美团的实际情况。例如，网络架构需要有较大的调整，同时需要有共享存储的支持。当时对美团来说，现有业务的基础设施已经基本固化，为适应OpenStack而做这样的调整基本不可接受。另外，OpenStack在很多细节方面达不到需要的级别。比如，OpenStack对跨机房多Zones的设计，它假设你机房之间的网络是完备的，这也不太符合我们的网络现状。因此，我们基于美团现有的主机使用模式和网络架构，重新设计了网络、存储和主机管理模型，自主研发了虚拟化管理平台。同时，我们在从单机房做到多机房的时候，在Zones之间做了解耦，比如在每个机房里都放置Glance服务节点，减少对跨机房网络的依赖。正是由于这些研发工作，使得我们经过不到一年时间的磨练，就实现了美团整个基础设施完全运行在私有云上的目标。

当然，由于我们不使用Nova，就意味着OpenStack社区的很多依赖Nova的组件和功能我们基本无法直接使用。不过，相对于OpenStack大而全/兼容并包的架构，我们坚持在每一方面都集中精力用好一种技术，如主机虚拟化使用KVM，网络虚拟化使用OpenVSwitch+OpenFlow，使得整个系统的开发和维护成本相对较低，同时能深挖这些技术方案的功能特性，最大限度地压榨硬件性能。同时，由于我们掌握了基础系统的代码，使得我们可以较高的效率添加一些新的业务功能(例如，对虚拟IP, USB KEY的支持等)，以及实现系统架构的升级改造(例如，对多机房架构支持等)。另外，我们对使用的OpenStack组件做的一些修改，比如对Swift的优化，目前技术委员会也在提议如何回馈给上游社区。当然了，这个还需要看社区对我们的patch接受与否，而我们也还是以满足业务需求优先。

其他方面，块存储落在本地的SAS盘上并在本地做RAID。目前我们对美团网自己的业务做RAID5，对公有云用户做RAID10。这是考虑到美团网自己的业务在应用层已经做了较完备的高可用设计的，即使掉了单个节点也不会影响到业务；但对于公有云用户而言，他们用的那一台云主机就是一个单点，所以要对他们的云主机做更好的保护。使用RAID10当然成本会比较高。我们也在考虑共享存储，当然前提是先解决了上面的稳定性和性能问题。以后会使用SSD，使块存储的性能有更大的提升。

网络方面做了分布式设计，主机上用了OpenFlow，通过OpenFlow修改二层协议，让每个用户拥有一个独立的扁平网络，跟其他用户的网络隔离。通过DNS虚拟化技术，使得不

同的用户可以在各自的私有网络上使用相同的主机名字，并在每个宿主机上部署分布式DNS和DHCP以实现基础网络服务的去中心化。

运维

美团云的运维思路跟整个美团的运维思路是一致的。下面介绍的运维思路既适用于美团云，也适用于整个美团网。

运维框架可以概括为五横三纵。从横向来看，自底向上分为五个层次：

- 物理层，包括机房网络、硬件设施。我们已在开展多机房和城域网建设，从最底层保证基础设施的稳定性。为了应对大规模机房建设带来的运维成本，我们实现了Baremetal自动安装部署的Web化管理，从服务器上架之后，其他工作均由自动化完成，并可以和虚拟机一样管理物理机。
- 系统层，包括操作系统、虚拟化。我们在虚拟化基础之上采用了模板化（镜像）的方式进行管理，也对Linux内核做了一部分定制开发，例如针对OVS的兼容性做了优化。
- 服务层，包括Webserver、缓存、数据库等基础服务。我们基于Puppet工具做了统一配置管理，有自己的软件仓库，并对一部分软件包做了定制。统一配置管理的好处，一方面是避免不一致的修改，保证集群的稳定性，另一方面是提高运维效率。
- 逻辑层，包括业务逻辑、数据流。这一层的主要工作是发布和变更。在很多其他公司，业务的发布上线、数据库的变更管理都是由运维来做，我们认为这样对开发、运维的协作成本较高，所以一直往开发人员自助的方向做，通过代码发布平台、数据库变更平台实现开发和运维工作的轻耦合。在发布平台中，每个应用对应独立的集群，有一位开发作为应用owner有最高权限，有多位开发作为应用的成员可以自助发布代码。数据库变更平台也有类似的权限控制机制，并在任务执行层面有特殊的稳定性考虑，例如将大的变更任务自动调度到夜间执行，对删除数据表的任务在后台先做备份。
- 应用层，包括用户可见部分。除了跟逻辑层有类似的发布和变更之外，我们有统一前端平台，实现访问流量的进出分离、行为监测和访问控制，这对于整体的安全性有很大的好处。

从纵向来看，有三部分工作，对上述五个层次是通用的：

- 监控。从物理层到服务层的监控和报警都是运维来跟进、响应的。对于逻辑层和应用层，也是开发人员自助的思路，运维提供监控API的规范，开发可以自己创建监控项、设定报警规则、进行增删改查。监控报警之后的处理，现在有些做到了自动化，有些还没有。尤其是有些基础架构和业务之间的纵向链条还没有打通，包括建立业务容量模型，某种特定的业务形态在多少用户的情况下最高负载多少，不

同负载等级下的SLA应该是多少，等等，这些模型都建立起来之后就能够进行自动化的处理。

- 安全。我们很早就部署了统一的安全接入平台，所有线上的人工操作都需要登陆relay跳板机，每个人有独立的登陆帐号，所有线上操作都有审计日志。更多的安全工作由专门的信息安全组负责。
- 流程。早期基于Jira做了一些简单的流程，但仍需要改进。现在正在针对比较集中的需求，开发相应的流程控制系统，方向也是自动化、自助化。从业务部门申请VM资源，到业务扩容的整个流程，我们正在进行上下游的打通，未来可以在Web界面上通过很简单的操作实现，也提供服务化的API，方便其他业务平台进行集成。虚拟化覆盖全业务线之后，这些事情做起来都变得很方便。

总之，美团网整体的运维思路就是：保证业务稳定运行，同时推动全面自动化、自助化。涉及开发、运维沟通协作的部分，尽可能通过自动化平台的方式，由开发人员自助完成。运维人员除了基础环境、平台建设之外，帮助业务进行高可用架构的梳理，提高代码的可运维性，以及定位和解决业务中的各类问题。

改进与演变

美团云从对内服务开始到现在两年以来，最大的一次改进就是从单机房到多机房的建设，这是跟美团网的城域网建设同步开展的。

单机房的时候，美团网业务早期曾遇到过运营商网络中断几小时的情况，期间业务不可用，非常痛苦。多机房冗余做到最理想的情况下是，即使一个机房整个断电了，业务也不受影响，当然这就意味着需要100%的冗余量，成本是比较高的。不过对于美团网来说，冗余的成本是很愿意承担的，因为业务不可用造成的损失要大于做这些冗余的成本，所以我们现在物理资源都留有50%的冗余，带宽一般会预留30%的冗余。

因为美团网的发展速度很快，去年我们一度遇到资源不够用的情况，在这上面踩了很多坑之后，开始做一些长远规划。现在美团网业务的双机房冗余已经实施了一部分，美团云也有两个机房，如果公有云客户的业务支持横向扩展，那么也可以做跨机房部署。这种机房级高可用做好了，对稳定性又是一个很大的提升，大大减少网络抖动对业务的影响，可用性SLA可以从现在的4个9做到更高。有些规模比较大的客户对服务质量会有比较高的需求，所以美团的城域网、以及未来的广域网，也会共享给我们的公有云客户。

另外上面说到我们数据库跑在物理机上，这一块现在用的是SSD，读写性能顶得上早期的三台15000转SAS，瓶颈在千兆网卡上，所以我们现在也在做万兆网络的升级改造。数据库服务以后也会开放给公有云用户使用，基础设施跟美团自身业务一致。

未来的计划

由于使用本地存储，所以现在虚拟机迁移需要在夜间进行，以减少对用户服务的影响。为了提高服务的可用性，在确保稳定性和性能的前提下，共享存储是一个不错的选择，所以我们正在测试万兆网络下的共享存储方案。另外，我们底层虚拟化机制用的KVM，本身是没有热插拔的功能，这也是我们计划要做的一件事。

现在很多客户问我们，什么时候出Redis，什么时候出云数据库，一些客户对Redis和MongoDB会有需求，Web服务想要MySQL。我们的计划是由DBA团队提供一些模板，相当是一些专门针对Redis/MySQL做好优化的系统镜像，让客户可以直接拿来用。这可能会在下一个版本release的时候推出。

我们还会提供一些基础架构的咨询服务，这个咨询服务一方面是工程师提供的人工服务，另一方面是以工具+文档的形式，以互联网的方式将我们的最佳实践共享出去。美团网做到现在的几百亿规模，内部有很多经验积累，如果能把这些积累传递给我们的客户，能够帮助客户少走很多弯路。

分享者简介

唐君毅，美团云产品工程师。哈工大毕业，曾任积木恒硕产品总监。现负责美团云的产品研发和运营工作。

邱剑，美团云架构师。清华毕业，曾任安和创新副总经理。现负责美团云的平台研发和架构工作。

朱晏，美团网高级技术经理。清华、中科院计算所毕业，曾任百货网联合创始人。现负责美团网的系统运维、云计算工作。

感谢杨赛对本文的策划。

查看原文：[美团云的技术演变：先把云主机做稳定了再说别的](#)



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

携程首席架构师谈 DevOps：找到合适的人最重要

受访者 Eric Ye 作者 杨赛

受访人简介 叶亚明（Eric Ye），携程首席架构师，负责移动、Web、呼叫中心等部门的研发工作，领导开发的业务和领域包括酒店、机票、商务旅游、开放API、全球站、用户体验研究。他从过去十年的电子商务变革中，总结出六种有效的编程模型，目前被广泛应用于携程内部的产品研发过程中。此外，他还致力于升级携程网架构并创建新一代框架，以提高可扩展性和可用性。

InfoQ：大家好，我是 InfoQ 的主持人，现在在 ArchSummit 大会现场。今天十分高兴的邀请到携程的技术负责人高级技术副总裁 Eric Ye（叶亚明）来接受我们的采访。今天的话题是运维工程师在时代的职业发展，首先您是如何对运维的工作如何划分的？包括在携程这一块你们对运维工程师是怎样分工的？

Eric Ye：运维工程师在携程，或者是在大部分的中国互联网公司，一般是按照他们的功能划分的，比如说DBA、SA、应用运维、Security、还有Storage、network，加上Tooling，还有一部分是负责上架下架的siteops。

InfoQ：为大规模系统做运维，大概是大型互联网公司开始就是比较兴起的，然后您在这里也是有了十几年的经验了，然后您觉得这个大规模系统运维的思路和理念在过去几年有什么比较大的变化？

Eric Ye：这个问题，我结合我在海内外的一些经历，稍微展开来讲一下，大致分为3个阶段。

第一阶段：1995年到2005年，互联网刚刚兴起的这个十年，互联网公司随着用户量、流量增加，运维变得越来越复杂，大家还是凭着运维工程师的经验来做事情，按照前面的功能划分，那些工程师对某个方面经验会越来越深入，但是从本质上，还是依赖于人手工来做运维的。

第二阶段，过去凭经验的手工方式，一方面反应慢，另一方面有经验的人毕竟很少，怎么能提高运维的效率，缩短排障时间，他们开发了好多工具。但是有些工具要求基础设施必须要标准化，简单化；如果基础设施很复杂，五花八门的设备，加上架构也不一样，自然影响自动化程度，运维工程师的压力会越来越大。

第三阶段，最近几年，像Facebook, LinkedIn这些公司已经走在前列，负责运维的人很少。像Facebook这么大的公司，运维人员就20个左右，这么大的流量，它是怎么做到的？它们有DevOps这样一个很超前的理念，在开发人员完成功能开发之后到生产环境发布，基本上全自动起来。除了Facebook/LinkedIn还有一类是云计算的供应商，像亚马逊，Google，他们不光是把DevOps贯彻在公司里头，还要将DevOps展现给第三方开发者。

InfoQ: 那么既然讲到 DevOps，其实 DevOps 像您刚才也提到，它涉及一个全链路的，包括从测试，到最后的部署，整个环节。那么如果一个公司的运维团队，现在还没有开始这个阶段，那么他们从哪里开始介入会比较容易一些？

Eric Ye: 开发人员完成了一个功能之后到生产发布的所有环节，DevOps负责将其全自动化。

DevOps不光是个概念，它是这样一套系统，开发人员有了它，开发人员就是Ops人员，与传统概念“开发人员不管运维”是相反的。这套系统有哪些组成部分？第一是，你要测试，作为开发人员写完代码以后，要做Unit test、功能Test、CI test，这些对开发人员来说是蛮自然的。之后要到staging，通过整个功能集成测试，再将功能代码发布到生产环境。但生产环境，Dev的人员是不能直接接触的，通过 DevOps把生产环境抽象起来展现给开发人员，也就是说开发人员也不需要知道后面的生产环境是什么样子，通过DevOps平台，可以方便快捷的发布到标准的基础设施之上。

听起来很简单，但实际上发布需要非常严谨，有阶梯式的发布（灰度发布），尤其像携程这个规模，每个集群规模都不小，那这些发布过程中一旦碰到问题怎么办？如果一个代码在测试的环境没问题，但生产环境上又有问题怎么办？这背后需要有一套自动检测的工具，监控整个发布过程，当代码出错率高到一定程度，可以自动刹车，自动回滚，如果代码质量符合生产环境的要求，就往前推进，一直到整个生产环境完成发布，没有问题以后，开发人员才能离开这个生产环境。

这一整个过程都由开发去掌控，这样全自动的效率会非常之高； 刚才讲了DevOps整个流程的细节，这里头门槛还是蛮高的，如果是传统运维的人，会觉得这是蛮不可思议的事情。

从开发人员来说，这怎么弄？我以前只知道写代码，这个整个过程是怎么玩儿的。 对于成功的那些公司像LinkedIn、Google、Facebook，是怎么做这件事情的呢？它们有个组织架构，叫做Infrastructure Tooling，既做基础设施，又负责自动化工具研发，他们是蛮资深的工程师，还不是一般的产品开发的工程师，而是对系统、对基础设施的建设非常有兴趣的一些人。

现在来回答你的问题，作为一个公司，他如果没有DevOps怎么进入，怎么去开发这套系统？我觉得门槛是蛮高的，第一你要找到合适的人，这些人是对基础设施、对工具很有兴趣、很有研究的一些人；第二他们需要了解当前开源社区、业界的技术趋势，有很多与DevOps相关的开源系统已经非常流行，比如说代码管理工具Git，代码Code review工具Review Board/Gerrit，还有像CI方面有Jenkins。需要借助于现有开源系统来构建DevOps；第三基础设施的标准化非常重要，没有标准就无法实现最大程度自动化。有了基础设施标准，这套系统应该是每个开发工程师的开发、测试、发布用到的工具是一样的。如果每个团队、部门都不一致，那个DevOps又变成一个空的东西。

总之DevOps涉及到的细节蛮多的，如何打造，简而言之，就是要有合适的、有经验的人，起点要站的高一点，不要闭门造车，在标准的基础设施上构建各种涵盖各个环节的统一工具。

InfoQ: 第一步找到合适的人最重要的，而且这个人是不是我从企业内部，他本身了解这套系统会比较好，因为如果从外头引入的话，他可能不了解这套系统？

Eric Ye: 两方面的力量都要结合。内部工程师对现有系统比较了解，是有帮助的，因为我们做DevOps这样的系统，是要解决他们眼前碰到的问题。不过这还是不够，你需要引进知道DevOps的概念，知道DevOps怎么搭建的人才，这两个力量会产生合力，更快开发出DevOps系统。自己学的话还是比较慢，而且容易走弯路。

InfoQ: 刚才您也提到了云计算的那个运维跟之前的运维也不太一样，云计算是将很多这种边界的工作，就是以前开发、运维的边界，他移到了运维这个端。然后如果是传统的运维工程师他到了云计算会无法掌控，因为它要复杂很多，比如说很多运维吐槽 OpenStack，说我搞不定，他就没有办法去运维那套系统，您觉得这是因为云计算的系统做的还不够友好，还是说这些人的能力跟不上？

Eric Ye: 我觉得两个原因都有，第一云计算不算很成熟，也很复杂，涉及到的东西太多，刚才讲到有Storage、Network、Compute，也涉及到跟开发有关的发布，应用的运营，还有Scale UP, Scale down，有弹性计算能力，在世界上比较成功的云计算公司也是少数几家，都是以技术为导向的，驾驭这个产品对技术要求非常高，不是只靠一个运营团队就能搞定的，所以难度比较大。第二对于传统运维工程师，这些理念对他是蛮挑战的，因为他以前知道系统是怎么运维的，DB如何管理，但这些技能在云计算面前，它发不出力，有的时候都找不到门。那让他们去驾驭云计算平台，需要一个重新学习的过程，好比他以前是开车的，一下子要他开飞机，这个难度还是不小的。所以说他必须要学习了。

你刚才讲到的OpenStack最近是比较火，但OpenStack的历史来看相对短，没有多少年时间，并且涉及到的模块、技术非常广，质量还在提升过程中。如果你有一个团队去用OpenStack，三个月到六个月，只能知道一些皮毛，里面的水还是蛮深的。也可以这么说一些运维团队，要真正去学会OpenStack，驾驭OpenStack，难度非常之大，会觉得有点力不从心，容易觉得这个系统太庞大、不好用，或者是Bug比较多。

携程算是比较早就用OpenStack了，大概在一年半以前已经进入，我们现在很多的系统已经基于OpenStack来做，我前面提到基础设施标准化，就用OpenStack的方式去实现，而不是用一个文档规范来标准化。另外携程有一个比较独特的OpenStack应用场景，就是呼叫中心虚拟桌面云。所有的呼叫中心不再需要台式机，呼叫中心员工办公只需要一个云客户端加显示器即可，真正的桌面都运行在后端的云里面。虚拟桌面云整个平台，包括后端对桌面、云终端的运营管理、资源分配调度、动态伸缩等等功能，都是基于OpenStack来打造的。在整个过程中，我们也碰到了很多坑，但我们还是跃过去了，给OpenStack修复了很多bug。一旦研发到这个深度，OpenStack会对携程这样的企业，或者其他的企业很有价值。如果光是去做一个POC，十几个人去用用它，用了三个月以后觉得太复杂放弃，那么很难发现它的真正价值。

总体来说，造成不少对OpenStack吐槽的现状，不仅仅是运维的能力不够，也因为OpenStack还不够成熟，这两方面都有。

InfoQ: 最后一个问题，就是长远来看，有个预测，就是未来可能整个 IT 架构会逐渐往云的方向倾斜，可能像小的企业他也不太会倾向于自己架台机器跑跑，那么所以就有可能运维工程师都跑到云平台去做了，没有进云平台的就只好改行了。您对这个运维工程师这个工种未来是怎么看？

Eric Ye: 随着云技术越来越深入，越来越成熟以后，云技术这个趋势是不可避免的。一些小的创业公司，小的企业，把系统搬到云上，难度不是太大。举例来说，原本需要一百人的团队来开发、运维云系统，现在十个人就可以搞定。小公司对运维工程师的需求，会随着云技术的推进会减少。

所以说这种职业需求在这些公司会缩小。对于中型的公司，又面临这么一个选择，已经有了一定的规模的IT设施，把这些IT设施搬迁到云上去，从技术上说是可行的，但是需要很多改动，需要逐步落地，这个改动落地是一个成本，可能有这样的疑问，已经有成熟的团队，可以把现有基础设施维护好，那为什么还要搬迁到云上去？搬迁过程还有额外成本。但是因为一旦搬迁落地后，它对运维的需求就大大降低了，可以说是一个长痛还是短痛的选择问题。

云技术的好处除了整个运营成本会降低，还能支持弹性计算，当网站流量突然增大的时候，能快速扩容；所以很多中小企业非常愿意采用云技术。当然不是说有了云技术就不需要运维了，而是会直接减少运维的人员数量。

大企业接纳云技术，需要一个漫长的过程，一方面现在提供云技术的公司，还不能直接快速的把大企业的应用搬迁上来就能直接运行；背后还需要大量的工作，并且出于数据安全的考虑，支撑业务的核心应用会很长一段时间在由内部运维人员进行管理，但边缘的一些应用或者新开发的应用、比较符合标准的应用，搬到云技术上还是可行的。我认为这些大企业会分批的，逐步尝试云技术，这个过程应该会很漫长。

对于你的问题，那些运维的人员在云的前面会不会成为消失的工种？这个工种会长时间存在，但是长远来看，随着云计算的成熟，对于传统运维的技能的需求会逐渐的降低，所以如果一个运维工程师要去规划自己的职业，他应该去考虑这个问题，适应这个趋势，重新学习，才是长远之计。

查看原文：[携程首席架构师谈DevOps：找到合适的人最重要](#)

ThoughtWorks 技术雷达 2014 年 7 月刊： JavaScript、微服务和去中心化的技术趋势

作者 Abel Avram , 译者 李哲

ThoughtWorks最近发布了2014年7月刊的[技术雷达报告](#) ([PDF英文文件](#)) ([PDF中文版](#))，该报告关注了JavaScript生态系统、微服务、康威定律和基础设施去中心化这几个方面的重要发展趋势。

对于这一期的雷达报告，ThoughtWorks提到了如下几大趋势：

- JavaScript生态系统正在充满活力地不断发展。
- 人们对微服务的兴趣非常大并且很重视web API，用它来连接企业内部网络系统以及外部网络系统。
- 人们对[康威定律](#)的认识逐步深入。
- 在经过了互联网和云服务提供商的一段时间的合并之后，有必要对数据和基础设施进行去中心化设计。

和上一期的ThoughtWorks雷达报告的情况一样，该图表包含四个象限，每个象限又分为四个区域： Adopt——推荐采用； Trial——风险比较低，值得在项目中尝试； Assess——建议进行评估； Hold——谨慎运用。当数字标识的条目出现在和上一期的雷达报告相同的位置的时候，这些条目会放到圆形图案中；当这一条目是新的或者位置发生改变后，它们就会被放到圆角三角形中。下图所示就是技术（Technique）象限：

TECHNIQUES**ADOPT**

1. Forward Secrecy
2. Segregated DOM plus node for JS Testing

TRIAL

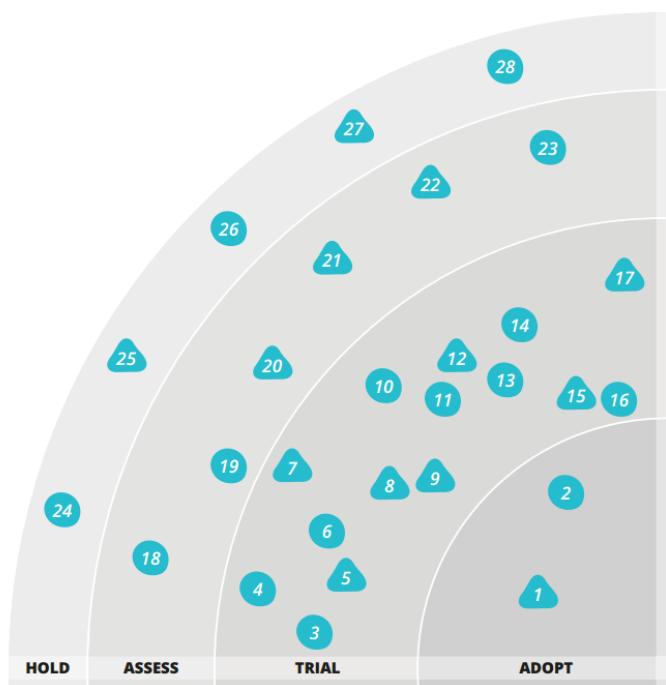
3. Capture domain events explicitly
4. Development environments in the cloud
5. Event Sourcing
6. Focus on mean time to recovery
7. Humans registry
8. Inverse Conway Maneuver
9. Living CSS Style Guides
10. Machine image as a build artifact
11. Masterless Chef/Puppet
12. Perimeterless enterprise
13. Provisioning testing
14. Real user monitoring
15. REST without PUT
16. Structured logging
17. Tailored Service Template

ASSESS

18. Bridging physical and digital worlds with simple hardware
19. Datensparsamkeit
20. Machine image pipelines
21. Pace-layered Application Strategy
22. Property-based unit testing
23. Tangible interaction

HOLD

24. Cloud lift and shift
25. DevOps as a team
26. Ignoring OWASP Top 10
27. Testing as a separate organization
28. Velocity as productivity



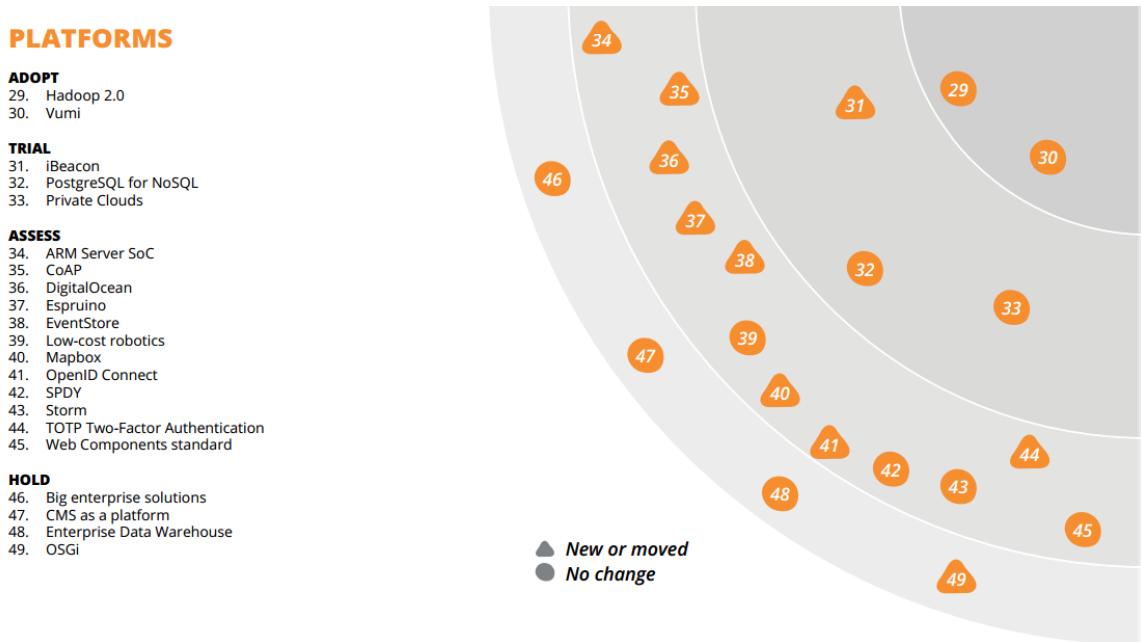
在这里，我们着重指出：

- (1) **Forward Secrecy (Adopt)** ——这是一种加密技术，当某个会话密钥被盗用后，之前的通信还是可以得到保护。
- (8) **反向康威操纵 (Inverse Conway Maneuver) (Trial)**——它建议“逐渐改进你的团队和组织结构来促进你所渴望的架构”，理想情况下，达到技术架构与业务架构的同构。
- (15) **没有PUT的REST(Trial)** —使用POST而非PUT，这是因为它分离了“命令和查询接口，并且强制调用方来支持最终一致性”
- (25) **DevOps作为一个团队 (Hold)**——这意味着要提醒的是，DevOps是一种文化观念的转变，组织机构不该吞下将DevOps作为一个团队而造成的苦果。

对于平台象限（下图所示），我们注意到如下内容：

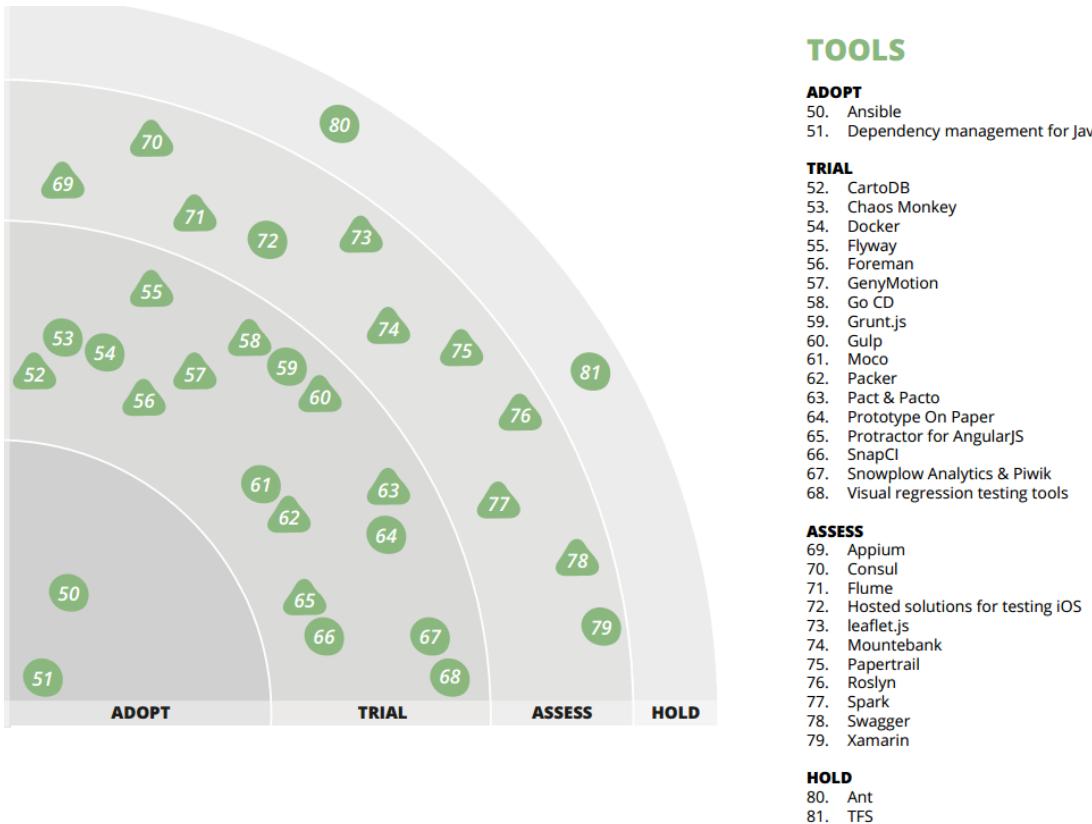
- (29) **Hadoop 2.0 (Adopt)** 已经从Trial移到Adopt区域。（图中Hadoop 2.0采用圆形图案来表示是错误的。）
- 大量平台都被建议进行评估（assessment），这其中包括ARM SoC、(35) **CoAP**——一个物联网(IoT)协议、(37) **Espruino**——一个包含了JavaScript解释器的控制器，此外还有两阶段认证（**Two-factor Authentication**）。

- 有趣的是，ThoughtWorks已经将OSGi置于Hold区域，这是因为它“只能解决整体问题中的一小部分，而它本身却经常给项目带来意外的复杂性，例如更加复杂的构建流程”。



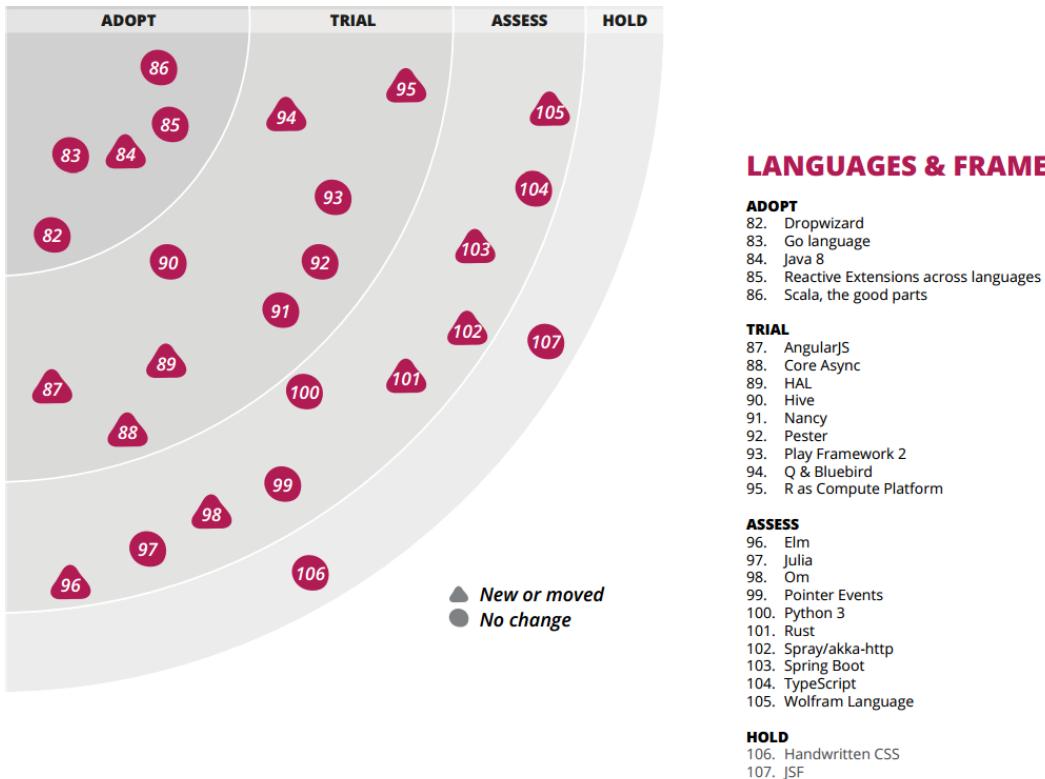
现在我们来到工具象限中，我们可以注意到：

- (50) [Ansible](#) 已经移到Adopt区域中。
- (58) [Go CD](#) 这是由ThoughtWorks在3月份开源的一个CD工具，对它的建议是Trial。
- 大量的工具被收录到Assess区域中，这包括(69) [Appium](#)——一套移动端自动化测试框架、(76) [Roslyn](#)——微软的编译器技术、(77)[Spark](#)——一款大数据分析工具和 (78) [Swagger](#)——一套RESTful API 标准。



对于最后一个语言和框架象限，我们注意到：

- (84) **Java 8 (Adopt)** ——ThoughtWorks认为Java 8成功地保持了向后的兼容性，并同时引入了“重大的语言改动而又能与现有的库和特性相保持一致”，建议进行采用。
 - (87) **AngularJS**、(88) **Core Async**以及(95) **R**都被认为是适合于放在Trial区域中。
 - 有一些更新的语言被建议为assessed，比如(96) **Elm**、(101) **Rust**以及(105) **Wolfram**。
 - (107) **JSF**依然停留在Hold位置，ThoughtWorks认为“JSF是有缺陷的，因为它试图将HTML、CSS和HTTP抽象出来，而这与现代的web框架所做的是相背离的”。



查看原文地址：<http://www.infoq.com/news/2014/07/thoughtworks-radar-july-2014>

感谢赵震一对本文的审校。

查看原文：[ThoughtWorks技术雷达 2014年7月刊：JavaScript、微服务和去中心化的技术趋势](#)

为什么 ZeroMQ 不应该成为你的第一选择

作者 马德奎

[Tyler Treat](#)是一名软件开发人员，他近日发表了一篇博文《[为什么ZeroMQ不应该成为你的第一选择](#)》。

文中，[Tyler Treat](#)对[nanomsg](#)和ZeroMQ进行了比较。[nanomsg](#)是一个套接字库，提供了多种常见的通信协议，其目标是使网络层更快、更具扩展性、更容易使用。它用C进行了彻底地重写，可以说是对ZeroMQ的重建。它构建在ZeroMQ的可靠性能之上，同时又提供了若干重要的改进。而且，它还试图消除ZeroMQ经常出现一些怪异行为。作者从以下几个方面对二者进行了比较：

- 用于新传输协议的API——对于ZeroMQ，人们经常抱怨的问题是它没有提供用于新传输协议的API，这从根本上把用户限制在TCP、PGM、IPC和ITC上。而[nanomsg](#)提供了一个可插拔的接口，用于新的传输（如WebSockets）和消息协议。
- POSIX兼容性——[nanomsg](#)完全兼容POSIX，而且API更简洁，兼容性更好。在ZeroMQ中，套接字用空指针表示，然后绑定到上下文；而在[nanomsg](#)中，只需要初始化一个新的套接字并使用它，一步即可完成。
- 线程安全——ZeroMQ在架构上有一个根本性缺陷：其套接字不是线程安全的。在ZeroMQ中，每个对象都被隔离在自己的线程中，因此不需要信号量和互斥锁。并发是通过消息传递实现的。[nanomsg](#)消除了对象与线程间的一对一关系，它不再依赖于消息传递，而是将交互建模为一组状态机。因此，[nanomsg](#)套接字是线程安全的。
- 内存和CPU使用效率——ZeroMQ使用一种很简单的Trie结构存储和匹配发布/订阅服务。当订阅数超过10000时，该结构很快就显现出不合理之处了。[nanomsg](#)则使用一种称为“基数树（radix tree）”的结构来存储订阅，并提供了真正的零复制API，允许内存从一台机器复制到另一台机器，而且完全不需要CPU的参与，这极大地提高了性能。
- 负载均衡算法——ZeroMQ采用了轮转调度算法。虽然该算法可以平均分配工作，但也有其局限性。比如，有两个数据中心，一个在伦敦，一个在纽约。在理想情况下，一个位于伦敦数据中心的网站，其请求不应该路由到纽约。但在ZeroMQ的负载均衡算法里，这完全有可能。而[nanomsg](#)避免了这种情况的出现。

除此之外，文中还提到，[nanomsg](#)提供了一个名为nanocat的命令行工具，用于与系统进行交互。

作者继续写道，nanomsg旨在实现“可扩展协议（Scalability Protocols）”，用于构建可扩展的高性能分布式系统。当前，它定义了六种不同的可扩展协议：PAIR、REQREP、PIPELINE、BUS、PUBSUB和SURVEY。

既然nanomsg在ZeroMQ的基础上做了如此多的改进，那我们为什么还要用ZeroMQ呢？针对这个疑问，作者指出，nanomsg还相对年轻，它还没有达到ZeroMQ的成熟度，没有像ZeroMQ那样有一个繁荣的开发者社区。另外，ZeroMQ有丰富的文档及其它资源，可以帮助开发人员使用它，而nanomsg的文档非常少。

尽管如此，作者还是认为nanomsg所做的改进，尤其是它的可扩展协议，使它非常有吸引力。从技术上讲，从三月份开始，nanomsg就已经开始beta测试，因此，生产就绪版本已经指日可待。

感谢郭蕾对本文的审校。

查看原文：[为什么ZeroMQ不应该成为你的第一选择](#)

QCon全球软件开发大会

2014年10月16-18日

上海光大会议中心国际大酒店

上海站 2014

9
折优惠

9月28日前

精彩早知道



一个程序员的创业寻梦坎坷之路

前JavaEye网站创始人 范凯

开放式移动端平台架构设计

阿里巴巴-研究员 岑文初

程序员与黑客

知道创宇技术副总裁 余弦

开放式移动端平台架构设计

IBM Multi-tenant JVM项目技术负责人 李三红

Netty架构剖析和行业应用

华为平台架构师 李林锋

技术人该做什么样的产品

上海泰尼网络科技有限公司创始人 郝培强

支付安全的实践和思考

小微金服（支付宝）安全专家 郑歆炜

Spark应用案例分析

Databricks软件工程师 连城



更多重磅嘉宾，精彩内容，敬请关注QCon上海官网网站：<http://www.qconshanghai.com/>

咨询热线：010-64738142 咨询邮箱：qcon@cn.infoq.com

本期专题：深入浅出 Docker 系列

Docker 是 PaaS 供应商 dotCloud 开源的一个基于 LXC 的高级容器引擎，源代码托管在 GitHub 上，基于 Go 语言开发并遵从 Apache 2.0 协议开源。Docker 提供了一种在安全、可重复的环境中自动部署软件的方式，它的出现拉开了基于云计算平台发布产品方式的变革序幕。

为了更好的促进 Docker 在国内的发展以及传播，我们决定开设《[深入浅出 Docker](#)》专栏，邀请 Docker 相关的布道师、开发人员、技术专家来讲述 Docker 的各方面内容，让读者对 Docker 有更深入的了解，并且能够积极投入到新技术的讨论和实践中。另外，欢迎加入 InfoQ Docker 技术交流群交流 Docker 的最佳实践，QQ 群号：365601355。

本期专题作者简介

肖德时, Red Hat Engineering Service/HSS 内部工具组Team Lead. Nodejs开源项目 nodejs-cantas Lead Developer。擅长企业内部工具的设计以及实现。开源课程Rails Starter 的发起人。rubygem: lazy_high_charts的Maintainer。twitter账号: xds2000, 邮箱: xiaods@gmail.com

Docker 核心技术预览

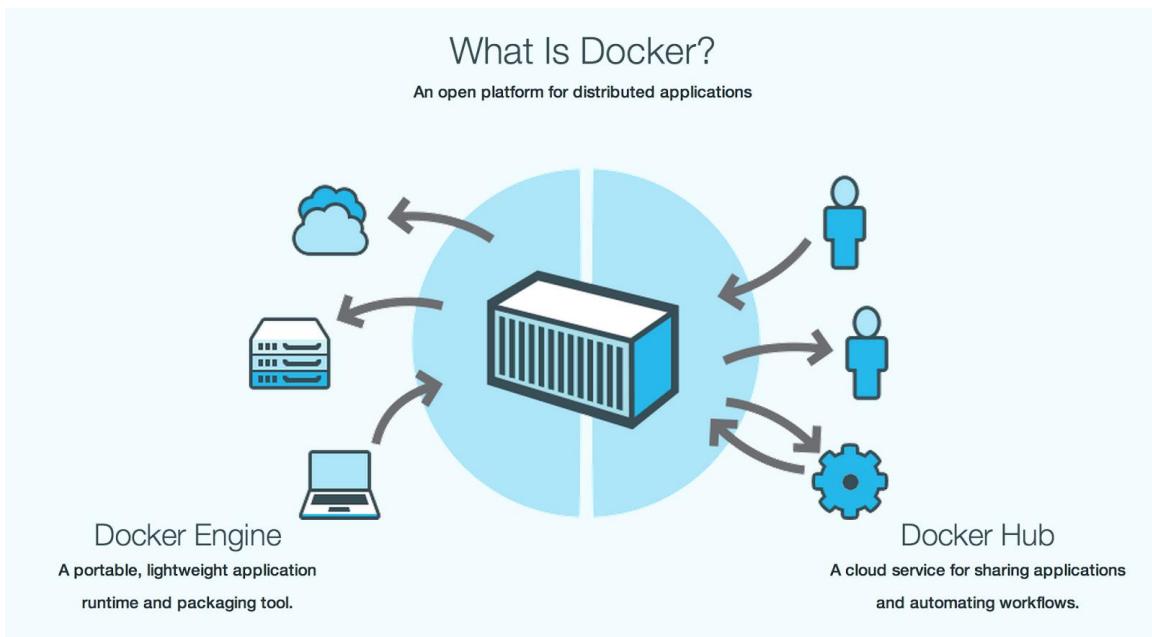
作者 肖德时

1. 背景

1.1. 由PaaS到Container

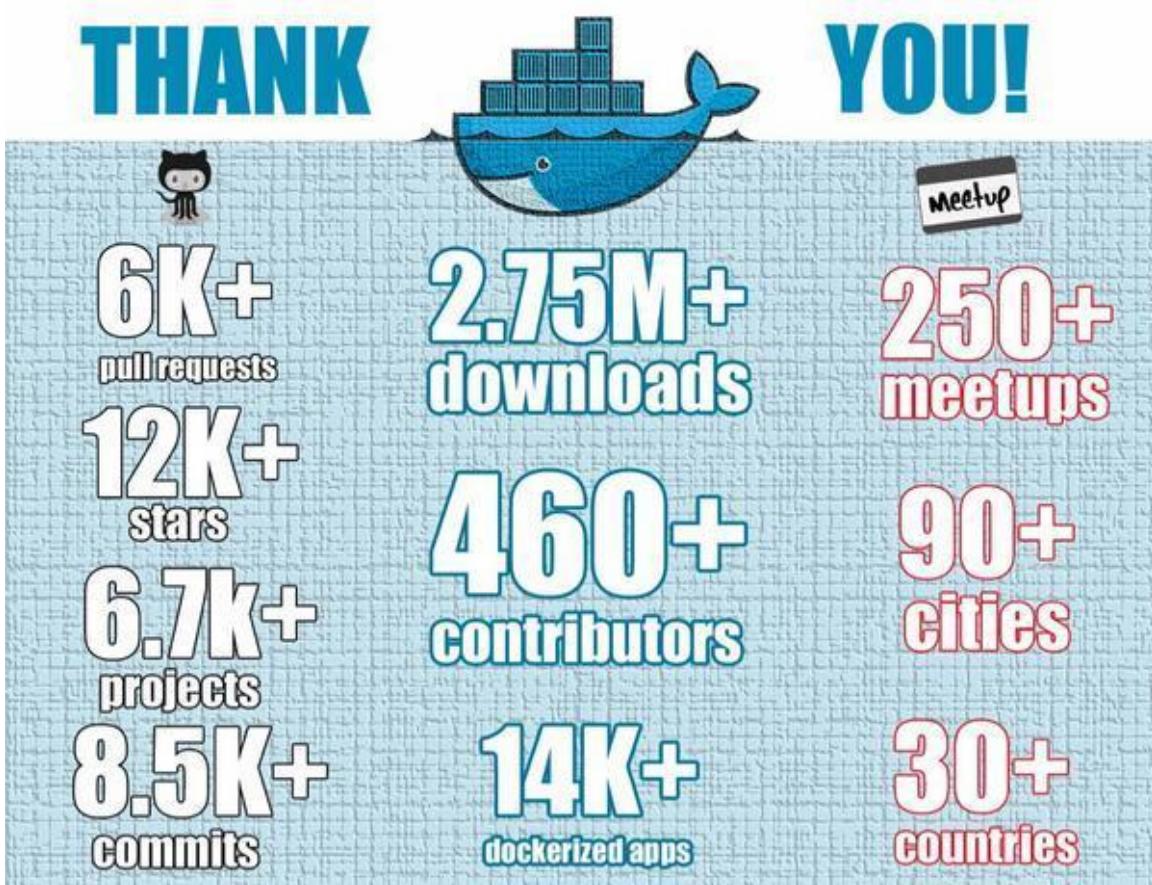
2013年2月，前Gluster的CEO Ben Golub和dotCloud的CEO Solomon Hykes坐在一起聊天时，Solomon谈到想把dotCloud内部使用的Container容器技术单独拿出来开源，然后围绕这个技术开一家新公司提供技术支持。28岁的Solomon在使用python开发dotCloud的PaaS云时发现，使用 LXC(Linux Container) 技术可以打破产品发布过程中应用开发工程师和系统工程师两者之间无法轻松协作发布产品的难题。这个Container容器技术可以把开发者从日常部署应用的繁杂工作中解脱出来，让开发者能专心写好程序；从系统工程师的角度来看也是一样，他们迫切需要从各种混乱的部署文档中解脱出来，让系统工程师专注在应用的水平扩展、稳定发布的解决方案上。他们越深入交谈，越觉得这是一次云技术的变革，紧接着在2013年3月Docker 0.1发布，拉开了基于云计算平台发布产品方式的变革序幕。

1.2 Docker简介



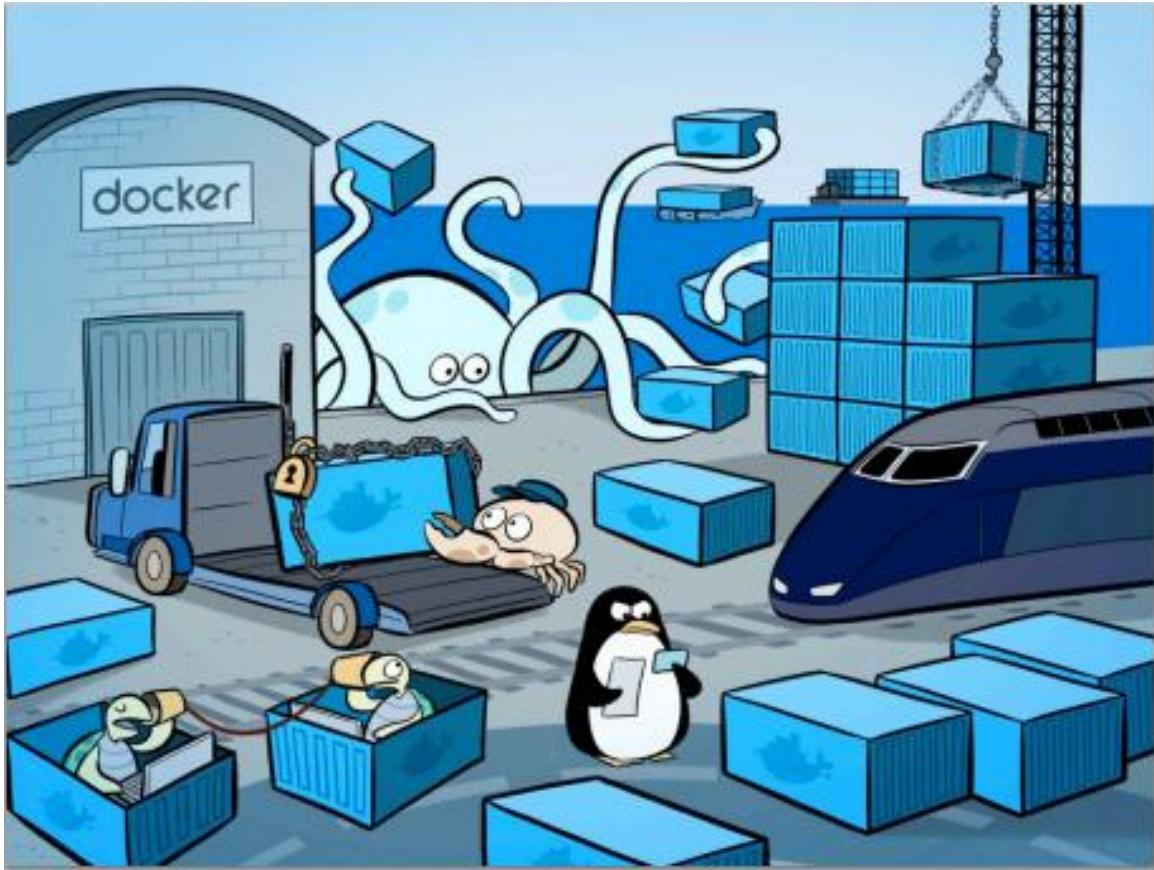
[Docker](#) 是 Docker.Inc 公司开源的一个基于 LXC技术之上构建的Container容器引擎，[源代码](#)托管在 GitHub 上，基于Go语言并遵从Apache2.0协议开源。 Docker在2014年6月

召开DockerConf 2014技术大会吸引了IBM、Google、RedHat等业界知名公司的关注和技术支持,无论是从 GitHub 上的代码活跃度,还是Redhat宣布在[RHEL7中正式支持Docker](#),都给业界一个信号,这是一项创新型的技术解决方案。就连 Google 公司的 Compute Engine 也[支持 docker 在其之上运行](#),国内“BAT”先锋企业百度Baidu App Engine(BAE)平台也是[以Docker作为其PaaS云基础](#)。



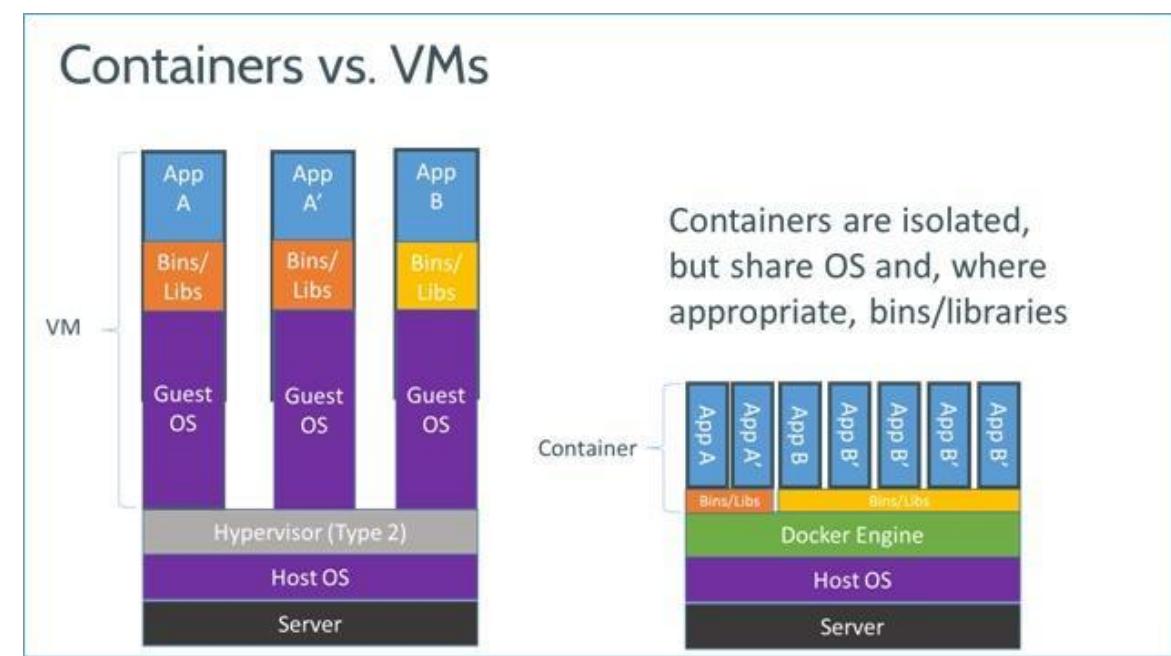
Docker产生的目的就是为了解决以下问题:

1) 环境管理复杂: 从各种OS到各种中间件再到各种App, 一款产品能够成功发布, 作为开发者需要关心的东西太多, 且难于管理, 这个问题在软件行业中普遍存在并需要直接面对。Docker可以简化部署多种应用实例工作, 比如Web应用、后台应用、数据库应用、大数据应用比如Hadoop集群、消息队列等等都可以打包成一个Image部署。如图所示:



2) 云计算时代的到来: AWS的成功,引导开发者将应用转移到云上,解决了硬件管理的问题,然而软件配置和管理相关的问题依然存在 (AWS cloudformation是这个方向的业界标准,样例模板可[参考这里](#))。Docker的出现正好能帮助软件开发者开阔思路,尝试新的软件管理方法来解决这个问题。

3) 虚拟化手段的变化: 云时代采用标配硬件来降低成本,采用虚拟化手段来满足用户按需分配的资源需求以及保证可用性和隔离性。然而无论是KVM还是Xen,在 Docker 看来都在浪费资源,因为用户需要的是高效运行环境而非OS, GuestOS既浪费资源又难于管理,更加轻量级的LXC更加灵活和快速。如图所示:



4) LXC的便携性: LXC在 Linux 2.6 的 Kernel 里就已经存在了, 但是其设计之初并非为云计算考虑的, 缺少标准化的描述手段和容器的可便携性, 决定其构建出的环境难于分发和标准化管理(相对于KVM之类image和snapshot的概念)。Docker就在这个问题上做出了实质性的创新方法。

1.3 Docker的Hello World

以Fedora 20作为主机为例, 直接安装docker-io:

```
$ sudo yum -y install docker-io
```

启动docker后台Daemon:

```
$ sudo systemctl start docker
```

跑我们第一个Hello World容器:

```
$ sudo docker run -i -t fedora /bin/echo hello world
```

```
Hello world
```

可以看到在运行命令行后的下一行会打印出经典的Hello World字符串。

2. 核心技术预览

Docker核心是一个操作系统级虚拟化方法，理解起来可能并不像VM那样直观。我们从虚拟化方法的四个方面：**隔离性、可配额/可度量、便携性、安全性**来详细介绍Docker的技术细节。

2.1. 隔离性: Linux Namespace(ns)

每个用户实例之间相互隔离，互不影响。一般的硬件虚拟化方法给出的方法是VM，而LXC给出的方法是container，更细一点讲就是kernel namespace。其中**pid、net、ipc、mnt、uts、user**等namespace将container的进程、网络、消息、文件系统、UTS("UNIX Time-sharing System")和用户空间隔离开。

1) pid namespace

不同用户的进程就是通过pid namespace隔离开的，且不同 namespace 中可以有相同pid。所有的LXC进程在docker中的父进程为docker进程，每个lxc进程具有不同的namespace。同时由于允许嵌套，因此可以很方便的实现 Docker in Docker。

2) net namespace

有了 pid namespace，每个namespace中的pid能够相互隔离，但是网络端口还是共享host的端口。网络隔离是通过net namespace实现的，每个net namespace有独立的 network devices, IP addresses, IP routing tables, /proc/net 目录。这样每个container的网络就能隔离开来。docker默认采用veth的方式将container中的虚拟网卡同host上的一个docker bridge: docker0连接在一起。

3) ipc namespace

container中进程交互还是采用linux常见的进程间交互方法(interprocess communication - IPC)，包括常见的信号量、消息队列和共享内存。然而同 VM 不同的是，container 的进程间交互实际上还是host上具有相同pid namespace中的进程间交互，因此需要在IPC资源申请时加入namespace信息 - 每个IPC资源有一个唯一的 32 位 ID。

4) mnt namespace

类似chroot，将一个进程放到一个特定的目录执行。mnt namespace允许不同namespace的进程看到的文件结构不同，这样每个 namespace 中的进程所看到的文件目录就被隔离开了。同chroot不同，每个namespace中的container在/proc/mounts的信息只包含所在namespace的mount point。

5) uts namespace

UTS("UNIX Time-sharing System") namespace允许每个container拥有独立的hostname和domain name，使其在网络上可以被视作一个独立的节点而非Host上的一个进程。

6) user namespace

每个container可以有不同的 user 和 group id, 也就是说可以在container内部用container内部的用户执行程序而非Host上的用户。

2.2 可配额/可度量 - Control Groups (cgroups)

cgroups 实现了对资源的配额和度量。cgroups 的使用非常简单, 提供类似文件的接口, 在 /cgroup 目录下新建一个文件夹即可新建一个 group, 在此文件夹中新建 task 文件, 并将 pid 写入该文件, 即可实现对该进程的资源控制。groups 可以限制 blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net_cls、ns 九大子系统的资源, 以下是每个子系统的详细说明:

1. blkio 这个子系统设置限制每个块设备的输入输出控制。例如: 磁盘, 光盘以及 usb 等等。
2. cpu 这个子系统使用调度程序为 cgroup 任务提供 cpu 的访问。
3. cpuacct 产生 cgroup 任务的 cpu 资源报告。
4. cpuset 如果是多核心的 cpu, 这个子系统会为 cgroup 任务分配单独的 cpu 和内存。
5. devices 允许或拒绝 cgroup 任务对设备的访问。
6. freezer 暂停和恢复 cgroup 任务。
7. memory 设置每个 cgroup 的内存限制以及产生内存资源报告。
8. net_cls 标记每个网络包以供 cgroup 方便使用。
9. ns 名称空间子系统。

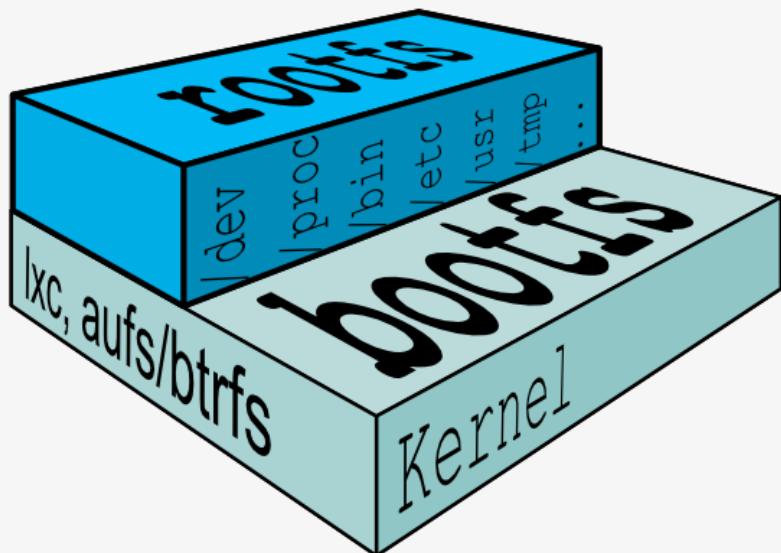
以上九个子系统之间也存在着一定的关系。详情请参阅 [官方文档](#)。

2.3 便携性: AUFS

AUFS (AnotherUnionFS) 是一种 Union FS, 简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统, 更进一步的理解, AUFS 支持为每一个成员目录(类似 Git Branch)设定 readonly、readwrite 和 whiteout-able 权限, 同时 AUFS 里有一个类似分层的概念, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下, 另一个更常用的就是将一个 readonly 的 branch 和一个 writeable 的 branch 联合在一起, Live CD 正是基于

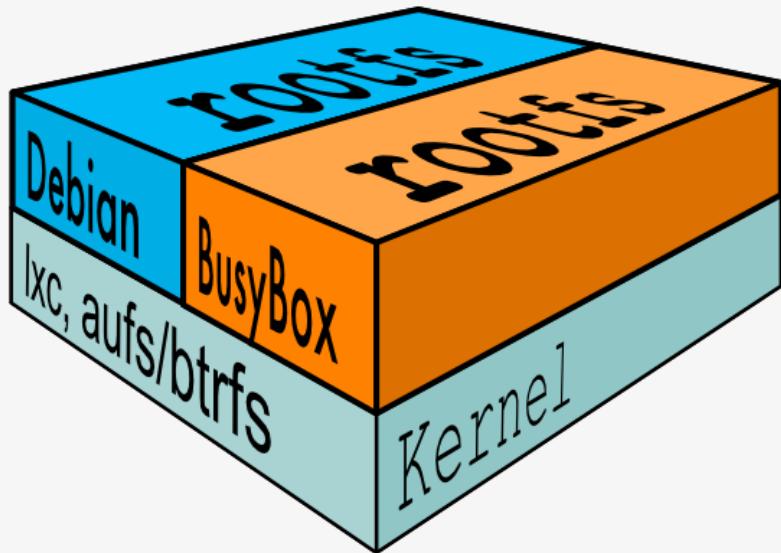
此方法可以允许在 OS image 不变的基础上允许用户在其上进行一些写操作。Docker 在 AUFS 上构建的 container image 也正是如此，接下来我们从启动 container 中的 linux 为例来介绍 docker 对AUFS特性的运用。

典型的启动Linux运行需要两个FS: bootfs + rootfs:

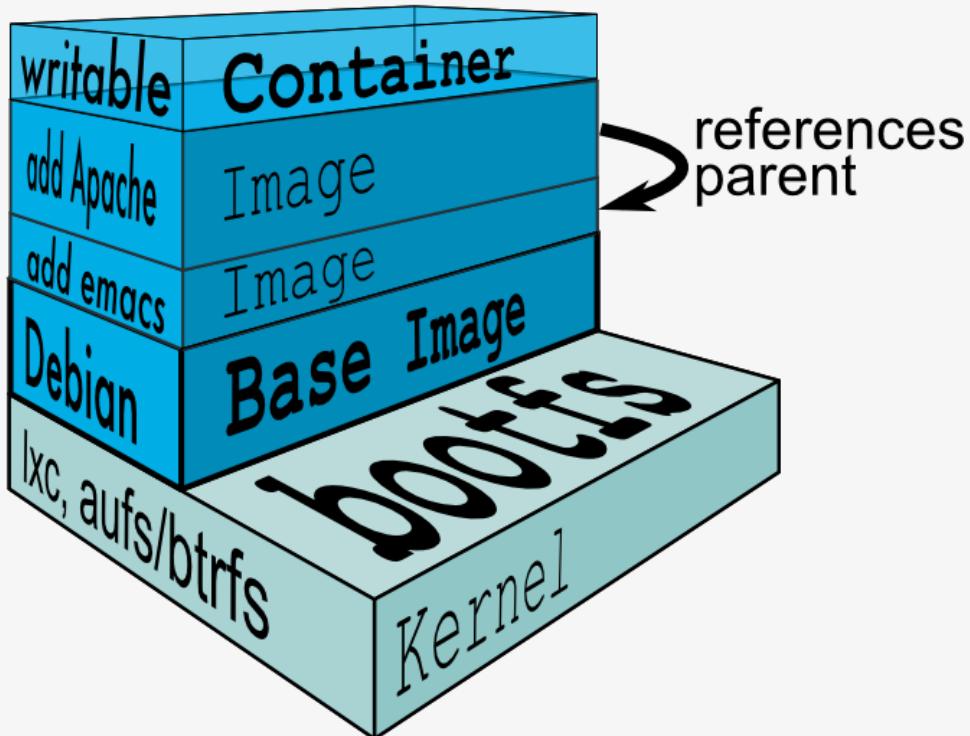


bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel,当boot成功后 kernel 被加载到内存中后 bootfs就被umount 了. rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc,/bin, /etc 等标准目录和文件。

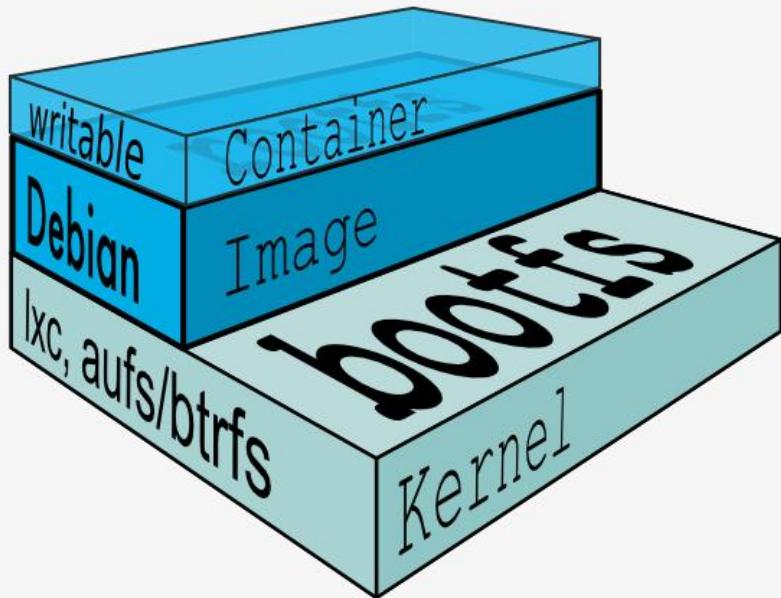
对于不同的linux发行版, bootfs基本是一致的, 但rootfs会有差别, 因此不同的发行版可以公用bootfs 如下图:



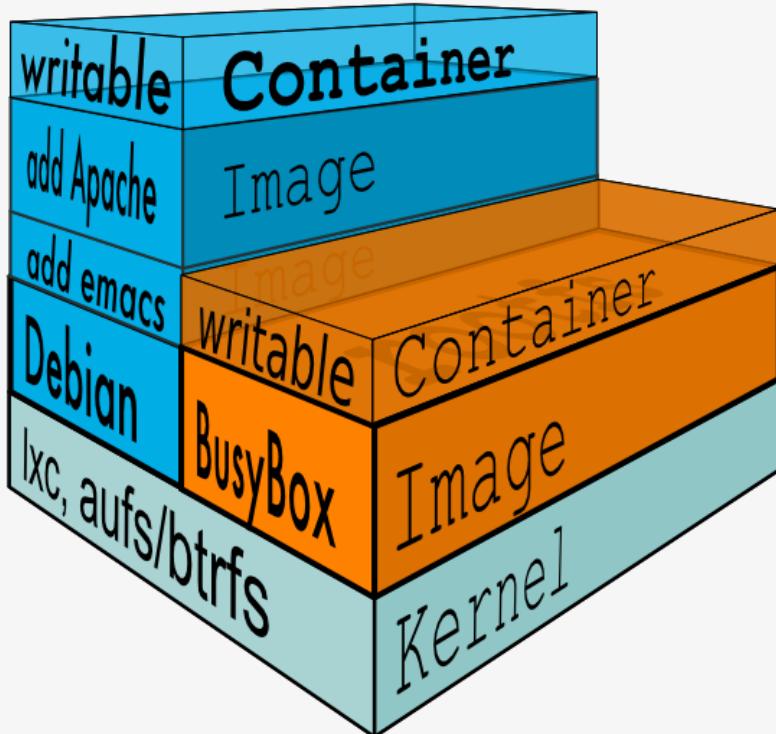
典型的Linux在启动后，首先将 rootfs 设置为 readonly，进行一系列检查，然后将其切换为 "readwrite" 供用户使用。在Docker中，初始化时也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加，这样一组readonly和一个writeable的结构构成一个container的运行时态，每一个FS被称作一个FS层。如下图：



得益于AUFS的特性，每一个对readonly层文件/目录的修改都只会存在于上层的writeable层中。这样由于不存在竞争，多个container可以共享readonly的FS层。所以Docker将readonly的FS层称作 **"image"** - 对于container而言整个rootfs都是read-write的，但事实上所有的修改都写入最上层的writeable层中，image不保存用户状态，只用于模板、新建和复制使用。



上层的image依赖下层的image，因此Docker中把下层的image称作父image，没有父image的image称作base image。因此想要从一个image启动一个container，Docker会先加载这个image和依赖的父images以及base image，用户的进程运行在writeable的layer中。所有parent image中的数据信息以及 ID、网络和lxc管理的资源限制等具体container的配置，构成一个Docker概念上的container。如下图：



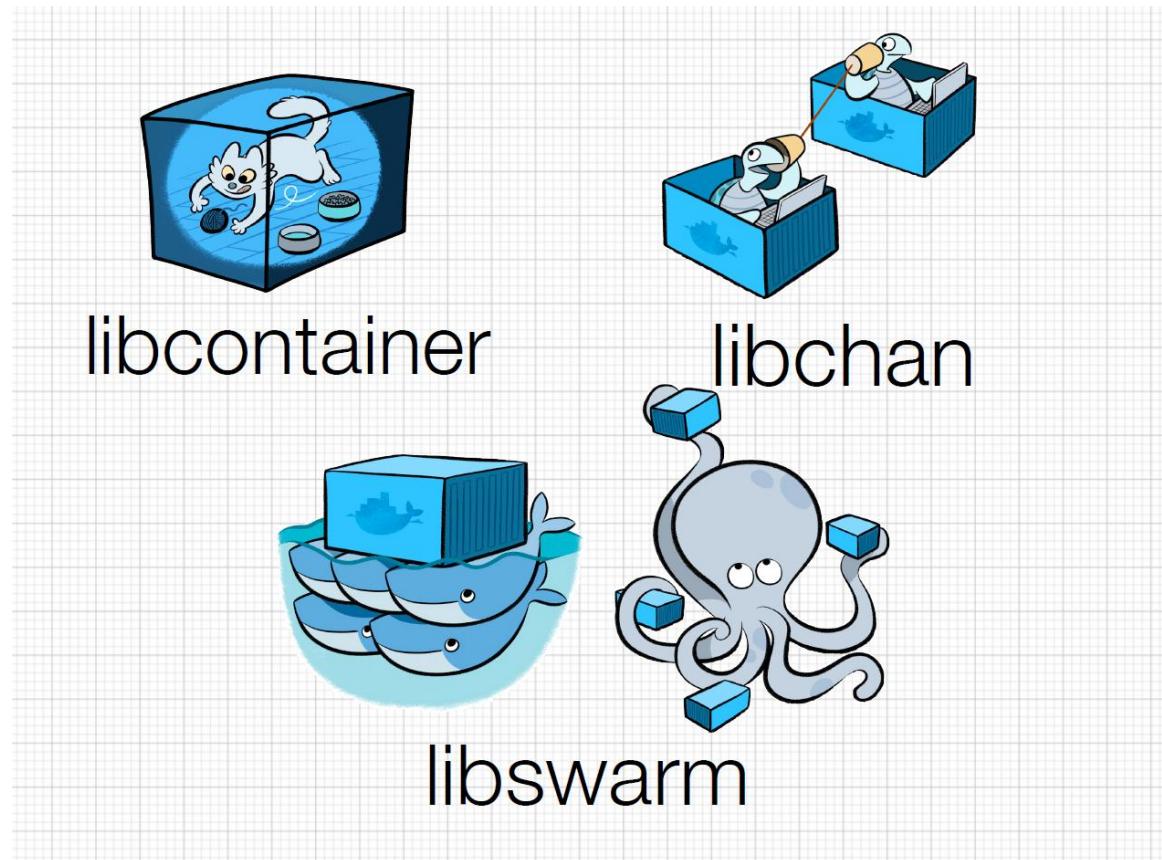
2.4 安全性: AppArmor, SELinux, GRSEC

安全永远是相对的，这里有三个方面可以考虑Docker的安全特性：

1. 由kernel namespaces和cgroups实现的Linux系统固有的安全标准；
2. Docker Deamon的安全接口；
3. Linux本身的安全加固解决方案,类如AppArmor, SELinux；

由于安全属于非常具体的技术，这里不在赘述，请直接参阅[Docker官方文档](#)。

3. 最新子项目介绍



我们再来看看Docker社区还有哪些子项目值得我们去好好研究和学习。基于这个目的，我把有趣的核心项目给大家罗列出来，让热心的读者能快速跟进自己感兴趣的项目：

1. Libswarm, 是Solomon Hykes (Docker的CTO) 在DockerCon 2014峰会上向社区介绍的新“乐高积木”工具：它是用来统一分布式系统的网络接口的API。Libswarm要解决的问题是，基于Docker构建的分布式应用已经催生了多个基于Docker的服务发现 (Service Discovery)项目，例如etcd, fleet, geard, mesos, shipyard, serf等等，每一套解决方案都有自己的通讯协议和使用方法，使用其中的任意一款都会局限在某一个特定的技术范围内。所以Docker的CTO就想用libswarm暴露出通用的API接口给分布式系统使用，打破既定的协议限制。目前项目还在早期发展阶段，值得参与。
 2. Libchan, 是一个底层的网络库，为上层 Libswarm 提供支持。相当于给Docker加上了ZeroMQ或RabbitMQ，这里自己实现网络库的好处是对Docker做了特别优化，更加轻量级。一般开发者不会直接用到它，大家更多的还是使用Libswarm来和容器交互。喜欢底层实现的网络工程师可能对此感兴趣，不妨一看。
 3. Libcontainer, Docker技术的核心部分，单独列出来也是因为这一块的功能相对独立，功能代码的迭代升级非常快。想了解Docker最新的支持特性应该多关注这个模块。
- ## 4. 总结

Docker社区一直在面对技术挑战，从容地给出自己的解决方案。云计算发展至今，有很多重要的问题没有得到妥善解决，Docker正在尝试让主流厂商接受并应用它。至此，以上Docker技术的预览到此告一段落，笔者也希望读者能结合自己的实际情况，尝试使用Docker技术。因为只有在亲自体会的基础之上，像Docker这样的云技术才会产生更大的价值。

参考文献：

1. <https://tiewei.github.io/cloud/Docker-Getting-Start/>
2. <http://docs.docker.com/articles/>
3. <http://www.slideshare.net/shykes/docker-the-road-ahead>
4. <http://www.centurylinklabs.com/meet-docker-ceo-ben-golub/>
5. <http://lwn.net/Articles/531114/>
6. <http://en.wikipedia.org/wiki/Aufs>
7. <http://docs.docker.io/en/latest/terms/filesystem/>
8. <http://docs.docker.io/en/latest/terms/layer/>
9. <http://docs.docker.io/en/latest/terms/image/>
10. <http://docs.docker.io/en/latest/terms/container/>
11. <https://stackoverflow.com/questions/17989306/what-does-docker-add-to-just-plain-lxc>

感谢郭蕾对本文的审校。

查看原文：[深入浅出Docker（一）：Docker核心技术预览](#)

Docker 命令行探秘

作者 肖德时

1. Docker命令行

Docker官方为了让用户快速了解Docker，提供了一个[交互式教程](#)，旨在帮助用户掌握Docker命令行的使用方法。但是由于Docker技术的快速发展，此交互式教程已经无法满足Docker用户的实际使用需求，所以让我们一起开始一次真正的命令行学习之旅。首先，Docker的命令清单可以通过运行 `docker`，或者 `docker help` 命令得到：

```
$ sudo docker
```

```
A self-sufficient runtime for linux containers.

Commands:
attach      Attach to a running container
build       Build an image from a Dockerfile
commit      Create a new image from a container's changes
cp          Copy files/folders from a container's filesystem to the host path
diff        Inspect changes on a container's filesystem
events     Get real time events from the server
export      Stream the contents of a container as a tar archive
history    Show the history of an image
images     List images
import      Create a new filesystem image from the contents of a tarball
info        Display system-wide information
inspect    Return low-level information on a container
kill        Kill a running container
load        Load an image from a tar archive
login      Register or log in to the Docker registry server
logs        Fetch the logs of a container
port        Lookup the public-facing port that is NAT-ed to PRIVATE_PORT
pause      Pause all processes within a container
ps          List containers
pull        Pull an image or a repository from a Docker registry server
push        Push an image or a repository to a Docker registry server
restart    Restart a running container
rm          Remove one or more containers
rmi        Remove one or more images
run         Run a command in a new container
save        Save an image to a tar archive
search     Search for an image on the Docker Hub
start      Start a stopped container
stop       Stop a running container
tag         Tag an image into a repository
top         Lookup the running processes of a container
unpause   Unpause a paused container
version   Show the Docker version information
wait       Block until a container stops, then print its exit code
```

在Docker容器技术不断演化的过程中，Docker的子命令已经达到34个之多，其中核心子命令(例如：run)还会有复杂的参数配置。笔者通过结合功能和应用场景方面的考虑，把命令行划分为4个部分，方便我们快速概览Docker命令行的组成结构：

功能划分	命令
环境信息相关	1. info 2. version
系统运维相关	1. attach 2. build 3. commit 4. cp 5. diff 6. export 7. images 8. import / save / load 9. inspect 10. kill 11. port 12. pause / unpause 13. ps 14. rm 15. rmi 16. run

	17. start / stop / restart 18. tag 19. top 20. wait
日志信息相关	1. events 2. history 3. logs
Docker Hub服务相关	1. login 2. pull / push 3. search

1.1 参数约定

单个字符的参数可以放在一起组合配置，例如

```
docker run -t -i --name test busybox sh
```

可以用这样的方式等同：

```
docker run -ti --name test busybox sh
```

1.2 Boolean

Boolean参数形式如： -d=false。注意，当你声明这个Boolean参数时，比如 docker run -d=true，它将直接把启动的Container挂起放在后台运行。

1.3 字符串和数字

参数如 --name="" 定义一个字符串，它仅能被定义一次。同类型的如-c=0 定义一个数字，它也只能被定义一次。

1.4 后台进程

Docker后台进程是一个常驻后台的系统进程，值得注意的是Docker使用同一个文件来支持客户端和后台进程，其中角色切换通过-d来实现。这个后台进程是用来管理容器的，使用Docker --help可以得到更详细的功能参数配置，如下图：

```
$ docker --help
Usage of docker:
  --api-enable-cors=false
  -b, --bridge=""
  --bip=""
  -D, --debug=false
  -d, --daemon=false
  --dns=[]
  --dns-search=[]
  -e, --exec-driver="native"
  -G, --group="docker"
  -g, --graph="/var/lib/docker"
  -H, --host=[:]
  --icc=true
  --ip="0.0.0.0"
  --ip-forward=true
  --iptables=true
  --mtu=0
  -p, --pidfile="/var/run/docker.pid"
  -r, --restart=true
  -s, --storage-driver=""
  --selinux-enabled=false
  --storage-opt={}
  --tls=false
  --tlscacert="/Users/dxiao/.docker/ca.pem"
  --tlscert="/Users/dxiao/.docker/cert.pem"
  --tlskey="/Users/dxiao/.docker/key.pem"
  --tlsv1=false
  -v, --version=false
  Print version information and quit

      Enable CORS headers in the remote API
      Attach containers to a pre-existing network bridge
      Use this CIDR notation address for the network bridge's IP, not compatible with -b
      use 'none' to disable container networking
      Enable debug mode
      Enable daemon mode
      Force Docker to use specific DNS servers
      Force Docker to use specific DNS search domains
      Force the Docker runtime to use a specific exec driver
      Group to assign the unix socket specified by -H when running in daemon mode
      use '' (the empty string) to disable setting of a group
      Path to use as the root of the Docker runtime
      The socket(s) to bind to in daemon mode
      specified using one or more tcp://host:port, unix:///path/to/socket, fd:///* or fd://socketfd.
      Enable inter-container communication
      Default IP address to use when binding container ports
      Enable net.ipv4.ip_forward
      Enable Docker's addition of iptables rules
      Set the containers network MTU
      if no value is provided: default to the default route MTU or 1500 if no default route is available
      Path to use for daemon PID file
      Restart previously running containers
      Force the Docker runtime to use a specific storage driver
      Enable selinux support
      Set storage driver options
      Use TLS; implied by tls-verify flags
      Trust only remotes providing a certificate signed by the CA given here
      Path to TLS certificate file
      Path to TLS key file
      Use TLS and verify the remote (daemon: verify client, client: verify daemon)
      Print version information and quit
```

Docker后台进程参数清单如下表：

参数	解释
--api-enable-cors=false	开放远程API调用的 CORS 头信息 。这个接口开关对想进行二次开发的上层应用提供了支持。
-b, --bridge=""	挂载已经存在的网桥设备到 Docker 容器里。注意，使用 none 可以停用容器里的网络。
--bip=""	使用 CIDR 地址来设定网络桥的 IP。注意，此参数和 -b 不能一起使用。
-D, --debug=false	开启Debug模式。例如： docker -d -D
-d, --daemon=false	开启Daemon模式。
--dns=[]	强制容器使用DNS服务器。例如： docker -d --dns 8.8.8.8

--dns-search=[]	强制容器使用指定的DNS搜索域名。例如: docker -d --dns-search example.com
-e, --exec-driver="native"	强制容器使用指定的运行时驱动。例如: docker -d -e lxc
-G, --group="docker"	在后台运行模式下，赋予指定的Group到相应的unix socket上。注意，当此参数 --group 赋予空字符串时，将去除组信息。
-g, --graph="/var/lib/docker"	配置Docker运行时根目录
-H, --host=[]	在后台模式下指定socket绑定，可以绑定一个或多个 tcp://host:port, unix:///path/to/socket, fd://* 或 fd://socketfd。例如： \$ docker -H tcp://0.0.0.0:2375 ps 或者 \$ export DOCKER_HOST="tcp://0.0.0.0:2375" \$ docker ps
--icc=true	启用内联容器的通信。
--ip="0.0.0.0"	容器绑定IP时使用的默认IP地址
--ip-forward=true	启动容器的 net.ipv4.ip_forward
--iptables=true	启动Docker容器自定义的iptable规则
--mtu=0	设置容器网络的MTU值，如果没有这个参数，选用默认 route MTU，如果没有默认route，就设置成常量值 1500。
-p, --pidfile="/var/run/docker.pid"	后台进程PID文件路径。
-r, --restart=true	重启之前运行中的容器
-s, --storage-driver=""	强制容器运行时使用指定的存储驱动，例如，指定使用devicemapper，可以这样： docker -d -s devicemapper

--selinux-enabled=false	启用selinux支持
--storage-opt=[]	配置存储驱动的参数
--tls=false	启动TLS认证开关
--tlscacert="/Users/dxiao/.docker/ca.pem"	通过CA认证过的的certificate文件路径
--tlscert="/Users/dxiao/.docker/cert.pem"	TLS的certificate文件路径
--tlskey="/Users/dxiao/.docker/key.pem"	TLS的key文件路径
--tlsverify=false	使用TLS并做后台进程与客户端通讯的验证
-v, --version=false	显示版本信息

注意，其中带有[] 的启动参数可以指定多次，例如

```
$ docker run -a stdin -a stdout -a stderr -i -t ubuntu /bin/bash
```

2. Docker命令行探秘

2.1 环境信息相关

info

使用方法： docker info

例子：

```
[fedora@docker-devel-cli docker]$ sudo docker -D info
Containers: 0
Images: 32
Storage Driver: devicemapper
  Pool Name: docker-252:1-130159-pool
  Data file: /var/lib/docker/devicemapper/devicemapper/data
  Metadata file: /var/lib/docker/devicemapper/devicemapper/metadata
  Data Space Used: 1616.9 Mb
  Data Space Total: 102400.0 Mb
  Metadata Space Used: 2.4 Mb
  Metadata Space Total: 2048.0 Mb
Execution Driver: native-0.2
```

```

Kernel Version: 3.11.10-301.fc20.x86_64
Debug mode (server): false
Debug mode (client): true
Fds: 11
Goroutines: 14
EventsListeners: 0
Init SHA1: 2c5adb59737b8a01fa3fb968519a43fe140bc9c9
Init Path: /usr/libexec/docker/dockerinit
Sockets: [fd://]

```

使用说明：

这个命令在开发者报告Bug时会非常有用，结合docker version一起，可以随时使用这个命令把本地的配置信息提供出来，方便Docker的开发者快速定位问题。

version

使用方法： docker version

使用说明：

显示Docker的版本号，API版本号，Git commit，Docker客户端和后台进程的Go版本号。

2.2 系统运维相关

attach

使用方法： docker attach [OPTIONS] CONTAINER

例子：

```

$ ID=$(sudo docker run -d ubuntu /usr/bin/top -b)
$ sudo docker attach $ID
top - 17:21:49 up 5:53, 0 users, load average: 0.63, 1.15, 0.78
Tasks: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.0 us, 0.7 sy, 0.0 ni, 97.7 id, 0.7 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 2051644 total, 723700 used, 1327944 free, 33032 buffers
KiB Swap: 0 total, 0 used, 0 free. 565836 cached Mem

PID USER      PR  NI    VIRT      RES      SHR S %CPU %MEM     TIME+ COMMAND
 1 root       20   0 19748    1280    1008 R  0.0  0.1  0:00.04 top
$ sudo docker stop $ID

```

使用说明：

使用这个命令可以挂载正在后台运行的容器，在开发应用的过程中运用这个命令可以随时观察容器内进程的运行状况。开发者在开发应用的场景中，这个命令是一个非常有用的命令。

build

使用方法： docker build [OPTIONS] PATH | URL | -

例子：

```
$ docker build .
Uploading context 18.829 MB
Uploading context
Step 0 : FROM busybox
--> 769b9341d937
Step 1 : CMD echo Hello world
--> Using cache
--> 99cc1ad10469
Successfully built 99cc1ad10469
```

使用说明：

这个命令是从源码构建新Image的命令。因为Image是分层的，最关键的Base Image是如何构建的是用户比较关心的，Docker官方文档给出了构建方法，请参考[这里](#)。

commit

使用方法： docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

例子：

```
$ sudo docker ps
ID                  IMAGE              COMMAND            CREATED           STATUS
PORTS
c3f279d17e0a      ubuntu:12.04       /bin/bash        7 days ago      Up
25 hours
197387f1b436      ubuntu:12.04       /bin/bash        7 days ago      Up
25 hours
$ docker commit c3f279d17e0a  SvenDowideit/testimage:version3
f5283438590d
$ docker images | head
```

REPOSITORY	TAG	ID	CREATED
VIRTUAL SIZE SvenDowideit/testimage seconds ago	version3 335.7 MB	f5283438590d	16

使用说明：

这个命令的用处在于把有修改的container提交成新的Image，然后导出此Image分发给其他场景中调试使用。Docker官方的建议是，当你在调试完Image的问题后，应该写一个新的Dockerfile文件来维护此Image。commit命令仅是一个临时创建Image的辅助命令。

cp

使用方法： cp CONTAINER:PATH HOSTPATH

使用说明：

使用cp可以把容器内的文件复制到Host主机上。这个命令在开发者开发应用的场景下，会需要把运行程序产生的结果复制出来的需求，在这个情况下就可以使用这个cp命令。

diff

使用方法： docker diff CONTAINER

例子：

```
$ sudo docker diff 7bb0e258aefe

C /dev
A /dev/kmsg
C /etc
A /etc/mtab
A /go
A /go/src
A /go/src/github.com
A /go/src/github.com/dotcloud
....
```

使用说明：

diff会列出3种容器内文件状态变化（A - Add, D - Delete, C - Change ）的列表清单。构建Image的过程中需要的调试指令。

export

使用方法: docker export CONTAINER

例子:

```
$ sudo docker export red_panda > latest.tar
```

使用说明:

把容器系统文件打包并导出来，方便分发给其他场景使用。

images

使用方法: docker images [OPTIONS] [NAME]

例子:

```
$ sudo docker images | head
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
<none> <none> 77af4d6b9913 19 hours ago 1.089 GB
committest latest b6fa739cedf5 19 hours ago 1.089 GB
<none> <none> 78a85c484f71 19 hours ago 1.089 GB
$ docker latest 30557a29d5ab 20 hours ago 1.089 GB
<none> <none> 0124422dd9f9 20 hours ago 1.089 GB
<none> <none> 18ad6fad3402 22 hours ago 1.082 GB
<none> <none> f9f1e26352f0 23 hours ago 1.089 GB
tryout latest 2629d1fa0b81 23 hours ago 131.5 MB
<none> <none> 5ed6274db6ce 24 hours ago 1.089 GB
```

使用说明:

Docker Image是多层结构的，默认只显示最顶层的Image。不显示的中间层默认是为了增加可复用性、减少磁盘使用空间，加快build构建的速度的功能，一般用户不需要关心这个细节。

import / save / load

使用方法:

```
docker import URL [- [REPOSITORY[:TAG]]]
```

```
docker save IMAGE
```

```
docker load
```

使用说明：

这一组命令是系统运维里非常关键的命令。加载(两种方法: import, load), 导出(一种方法: save)容器系统文件。

inspect

使用方法：

```
docker inspect CONTAINER|IMAGE [CONTAINER|IMAGE...]
```

例子：

```
$ sudo docker inspect --format='{{.NetworkSettings.IPAddress}}' $INSTANCE_ID
```

使用说明：

查看容器运行时详细信息的命令。了解一个Image或者Container的完整构建信息就可以通过这个命令实现。

kill

使用方法：

```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

使用说明：

杀掉容器的进程。

port

使用方法：

```
docker port CONTAINER PRIVATE_PORT
```

使用说明：

打印出Host主机端口与容器暴露出的端口的NAT映射关系

pause / unpause

使用方法：

```
docker pause CONTAINER
```

使用说明：

使用cgroup的freezer顺序暂停、恢复容器里的所有进程。详细freezer的特性，请参考[官方文档](#)。

ps

使用方法：

```
docker ps [OPTIONS]
```

例子：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
4c01db0b339c	ubuntu:12.04	bash	17 seconds
ago	Up 16 seconds	webapp	

d7886598dbe2	crosbymichael/redis:latest	/redis-server --dir	33
minutes ago	Up 33 minutes	6379/tcp	redis, webapp/db

使用说明：

docker ps打印出正在运行的容器， docker ps -a打印出所有运行过的容器。

rm

使用方法：

```
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

例子：

```
$ sudo docker rm /redis  
/redis
```

使用说明：

删除指定的容器。

rmi

使用方法:

```
docker rmi IMAGE [IMAGE...]
```

例子:

```
$ sudo docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
test1 latest fd484f19954f 23 seconds ago 7 B (virtual 4.964 MB)
test latest fd484f19954f 23 seconds ago 7 B (virtual 4.964 MB)
test2 latest fd484f19954f 23 seconds ago 7 B (virtual 4.964 MB)

$ sudo docker rmi fd484f19954f
Error: Conflict, cannot delete image fd484f19954f because it is tagged in multiple
repositories
2013/12/11 05:47:16 Error: failed to remove one or more images

$ sudo docker rmi test1
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8

$ sudo docker rmi test2
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8

$ sudo docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
test latest fd484f19954f 23 seconds ago 7 B (virtual 4.964 MB)

$ sudo docker rmi test
Untagged: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
Deleted: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
```

使用说明:

指定删除Image文件。

run

使用方法:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

例子：

```
$ sudo docker run --cidfile /tmp/docker_test.cid ubuntu echo "test"
```

使用说明：

这个命令是核心命令，可以配置的参数多达28个参数。详细的解释可以通过docker run --help列出。官方文档中提到的 [Issue 2702](#): "lxc-start: Permission denied - failed to mount" could indicate a permissions problem with AppArmor. 在最新版本的Docker中已经解决。

start / stop / restart

使用方法：

```
docker start CONTAINER [CONTAINER...]
```

使用说明：

这组命令可以开启(两个： start, restart)，停止(一个： stop)一个容器。

tag

使用方法：

```
docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/] [USERNAME/] NAME[:TAG]
```

使用说明：

组合使用用户名，Image名字，标签名来组织管理Image。

top

使用方法：

```
docker top CONTAINER [ps OPTIONS]
```

使用说明：

显示容器内运行的进程。

wait

使用方法：

```
docker wait CONTAINER [CONTAINER...]
```

使用说明：

阻塞对指定容器的其他调用方法，直到容器停止后退出阻塞。

2.3 日志信息相关

events

使用方法：

```
docker events [OPTIONS]
```

使用说明：

打印容器实时的系统事件。

history

使用方法：

```
docker history [OPTIONS] IMAGE
```

例子：

```
$ docker history docker
IMAGE CREATED CREATED BY SIZE
3e23a5875458790b7a806f95f7ec0d0b2a5c1659bfc899c89f939f6d5b8f7094 8 days ago
/bin/sh -c #(nop) ENV LC_ALL=C.UTF-8 0 B
8578938dd17054dce7993d21de79e96a037400e8d28e15e7290fea4f65128a36 8 days ago
/bin/sh -c dpkg-reconfigure locales && locale-gen C.UTF-8 &&
/usr/sbin/update-locale LANG=C.UTF-8 1.245 MB
be51b77efb42f67a5e96437b3e102f81e0a1399038f77bf28cea0ed23a65cf60 8 days ago
/bin/sh
-c apt-get update && apt-get install -y git libxml2-dev python build-essential
make gcc python-dev locales python-pip 338.3 MB
4b137612be55ca69776c7f30c2d2dd0aa2e7d72059820abf3e25b629f887a084 6 weeks ago
/bin/sh -c #(nop) ADD jessie.tar.xz in / 121 MB
750d58736b4b6cc0f9a9abe8f258cef269e3e9dced1146503522be9f985ada 6 weeks ago
/bin/sh -c #(nop) MAINTAINER Tianon Gravi <admwiggin@gmail.com>
-mkimage-debootstrap.sh -t jessie.tar.xz jessie http://http.debian.net/debian
0 B
```

511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158 9 months ago
0 B

使用说明：

打印指定Image中每一层Image命令行的历史记录。

logs

使用方法：

```
docker logs CONTAINER
```

使用说明：

批量打印出容器中进程的运行日志。

2.4 Docker Hub服务相关

login

使用方法：

```
docker login [OPTIONS] [SERVER]
```

使用说明：

登录Hub服务。

pull / push

使用方法：

```
docker push NAME[:TAG]
```

使用说明：

通过此命令分享Image到Hub服务或者自服务的Registry服务。

search

使用方法：

```
docker search TERM
```

使用说明：

通过关键字搜索分享的Image。

3. 总结

通过以上Docker命令行的详细解释，可以强化对Docker命令的全面理解。考虑到Docker命令行的发展变化非常快，读者可以参考官方的[命令行解释](#)文档更新相应的命令行解释。另外，通过以上Docker命令行的分析，可以知道Docker命令行架构设计的特点在于客户端和服务端的运行文件是同一个文件，内部实现代码应该是重用的设计。笔者希望开发者在开发类似的命令行应用时参考这样的设计，减少前后台容错的复杂度。

参考文献

- [1] <https://docs.docker.com/reference/commandline/cli/>
- [2] https://en.wikipedia.org/wiki/Cross-Origin_Resource_Sharing
- [3] https://en.wikipedia.org/wiki/CIDR_notation#CIDR_notation

感谢[郭董](#)对本文的策划和审校。

查看原文：[深入浅出Docker（二）：Docker命令行探秘](#)

Docker 开源之路

作者 肖德时

1. 背景

Docker从一开始的概念阶段就致力于使用开源驱动的方式来发展，它的成功缘于国外成熟的开源文化氛围，以及可借鉴的社区运营经验。通过本文详细的介绍，让大家可以全面了解一个项目亦或者一项技术是如何通过开源的方式发展起来的。为了更准确的描述Docker的社区状况，请先看一份来自Docker官方的数据：

Docker, the community

- >500 contributors
- ~20 core maintainers
- >8,000 Dockerized projects on GitHub
- >20,000 repositories on Docker Hub
- >250 meetups in >90 cities in >30 countries
- >500,000 downloads of boot2docker

图中数据的看点有：

1. 超过500个代码贡献者。代码的贡献者在社区发展过程中是非常重要的催化剂，它会不断加快产品迭代的速度，让项目更快的交付到最终用户的手里。
2. 20个全职开发。一般的开源项目一般都不会有如此多的全职开发人员，但是Docker却有20个全职开发人员来驱动一个社区项目。这也从侧面证明了国外公司对开源项目的支持力度，以至于一家初创公司都舍得投入20个全职开发来推动开源技术的发展。

3. 超过8000个创建在GitHub上的Docker相关项目。通过这个规模可以看到围绕Docker可以使用的项目已经非常丰富。
4. Docker技术聚会。30个国家超过90个城市举办超过250个Docker技术聚会，这样的技术聚会还在不断增加，Docker技术爱好者可以在线申请承办此类技术聚会。
5. 50万次的boot2docker下载。boot2docker为Docker官方推荐客户端，50万次也代表当前潜在的用户群体。

Docker Inc, the company

- Headcount: ~50
- Led by Open Source veteran Ben Golub (GlusterFS)
- Revenue:
 - t-shirts and stickers featuring the cool blue whale
 - SaaS delivered through Docker Hub
 - Support & Training

Docker公司目前正式员工50多名，由开源老手Ben Golub(前GlusterFS CEO)主持运营。现在主要有以下几个赢利点：

1. 印有蓝鲸的T恤和贴纸的品牌价值
2. 通过Docker Hub服务提供SaaS的分发服务
3. 提供Docker技术支持和培训

通过了解Docker的社区现状和公司的运营状况，可以发现其在运营的方式上并没有什么特别的地方。除了支付公司员工的正常开支外，它并没有像一般商业公司那样在推广上投入很多资金。Docker在推广上主要是将开源社区和社交网络作为基础推广平台，结合全球范围的Docker技术聚会，形成了良好的良性的客户互动和口口相传的品牌效应。在Docker的开源历程中，通过分析观察用户社区、源代码管理、合作伙伴的生态圈这三种形式，梳理出一套实践经验的案例，方便大家参考学习。

2. 用户社区维护

Docker技术首先考虑的是在技术社区里通过各种渠道来找到它的用户，而当前最流行的技术社交网络不是Twitter，而是GitHub。所以Docker第一步是向GitHub上提交自己的代码开始吸引自己的用户的。我们总说万事开头难，那么Docker的第一个Commit应该是什么，它是否需要包括测试、用户文档、用户开发指南、设计理念等一系列的文档和代码呢？如果参照常规的开源项目，它们都考虑的都很周到，完善的代码文档结构会让用户第一眼就知道这是一个成熟的项目，我们只要用就可以了。但Docker公司却不按套路出牌，第一个Commit仅包括6个主文件：

```
$ git log
commit a27b4b8cb8e838d03a99b6d2b30f76bdaf2f9e5d
Author: Andrea Luzzardi <aluzzardi@gmail.com>
Date:   Fri Jan 18 16:13:39 2013 -0800

    Initial commit
dxiao at dhcp-141-62 in ~/Documents/Code/go/docker on test
$ ll
total 88
drwxr-xr-x  2 dxiao  staff   68 Aug 21 17:36 bundles
-rw-r--r--  1 dxiao  staff  4573 Aug 27 15:41 container.go
-rw-r--r--  1 dxiao  staff  3530 Aug 27 15:41 container_test.go
-rw-r--r--  1 dxiao  staff  2509 Aug 27 15:41 docker.go
-rw-r--r--  1 dxiao  staff  3856 Aug 27 15:41 docker_test.go
-rw-r--r--  1 dxiao  staff  1068 Aug 27 15:41 filesystem.go
-rw-r--r--  1 dxiao  staff   766 Aug 27 15:41 filesystem_test.go
-rw-r--r--  1 dxiao  staff  2520 Aug 27 15:41 lxc_template.go
-rw-r--r--  1 dxiao  staff   738 Aug 27 15:41 state.go
-rw-r--r--  1 dxiao  staff  2041 Aug 27 15:41 utils.go
-rw-r--r--  1 dxiao  staff  2901 Aug 27 15:41 utils_test.go
```

没有README，没有开发环境指南，开始阶段用户无法有效的了解Docker项目。但是，这其实也是对的，因为在一个小众的开源项目的初期，很难吸引到社区用户来为它贡献代码。在接下来的很长一段时间，Docker主要是由Docker之父Solomon Hykes开发维护。在没有社区用户参与的情况下，他每天不分昼夜地提交代码，也许是为了生活，也许是爱好的，Solomon Hykes在拼命的实现心中那个目标：让LXC创建的容器更容易使用。用当下时髦的那句话讲就是不忘初心。



从 Solomon Hykes 贡献代码的趋势图，我们可以看到只有保证项目的活跃度（持续贡献代码）那么项目才有可能获得用户的认可和关注。试想，“三天打鱼，两天晒网”的开源项目会给用户怎么样的感觉？作者都不专注，用户怎么可能忠实？这种投入，并不是偶然性的，这是国外业界的开源文化，非常值得我们学习。

在代码贡献的同时，Docker.io的主站也在2013年的5月30日第一次提交到GitHub。这距离第一行代码提交到GitHub已经有5个月。总体来看，Docker从一开始仅仅是一个很酷的想法，到真正完成并可以对外发布版本，也是经历了小半年之久。开源的意义并不像国内大多数厂商发布开源项目一样，一定要在内部应用的很成功才发布到技术社区。我们通过以上数据，可以很容易的看出来，它一定是一个长期的、持续的过程。你越专注的投入，你的项目越有可能成功。

当然，如果仅仅Docker只在GitHub上提供代码来吸引用户，那也不会是最佳的实践。在网站建立后的下一步，它开始在全球各大城市的技术聚会上推广介绍Docker。Docker技术在全球推广的方法是通过线下的技术聚会，外加社交网络媒体传播才得到流行的。

Docker官网通过发布“[创建Docker技术聚会指南](#)”，引导Docker技术的爱好者自发申请承办技术聚会。Docker官网主要通过类似Twitter、Docker周报、用户论坛等渠道，及时的把每个城市即将开始的技术聚会时间和联系人发布出来，就可以把Docker的技术推广做下去。并且这种形式的最大好处是口碑相传，其长尾效应的结果是Docker技术开始在當地的技术圈得到关注。像这样的技术聚会，一般都在30到200人之间，对于商业项目的推广是无法参照这样推广的。但这种形式就像一颗种子散落在城市中间，一旦有人介绍Docker技术到这个城市的技术圈，就会有更多的用户去GitHub上关注Docker的项目进展。

3. 源代码管理

Docker的源码是按照模块划分的，每一个模块都会单独放在一个目录里。并且每个目录都会包含一个MAINTAINERS.md。当你提交一个Pull Request的时候，子模块的维护者就会站出来回答“LGTM (Looks Good To Me)” 表示已经看了你的代码并接受你提交的代码。

每一个想参与Docker开发的开发者，你需要进入一个特殊的目录hack，这个目录包括了所有你想知道的开发相关信息。它把开发者分为4类人，

1. 代码贡献者，详细指南看CONTRIBUTORS.md。
2. 代码维护者，详细指南看MAINTAINERS.md。
3. 程序打包者，详细指南看PACKAGERS.md。
4. 负责发布版本的维护者，详细指南看RELEASE-CHECKLIST.md。

所有的指南文档都非常详细完整，可以做到看完此文档就可以开始相应角色的任务。除了这些文档之外，hack目录还包括了发布，开发，测试等环节需要的辅助脚本，让开发者可以得到很多开发上的便利。

Docker从一开始就是围绕LXC开发的，但在版本稳定后，使用Go重写了一套类LXC接口实现。也就是 [libcontainer](#) 项目。它的代码管理方式与上面的Docker源代码管理方式一样，开发者可以很容易的导入到子项目libcontainer的开发。一致的管理方法可以提高流程的复用，这种管理方式希望能得到大家的借鉴参考。

4. 创建合作伙伴生态圈

首先，Docker从一开始使用Ubuntu版本作为开发环境，主要是Ubuntu上支持aufs(advanced multi layered unification filesystem)文件系统。所以，Docker对Ubuntu的支持是最好的。



还有就是大家非常熟悉的Google的GCE，在没有Docker之前，就已经使用自己开发的[类LXC容器](#)。在有了Docker之后，Google开发了[kubernetes](#)来管理Docker。



商业版本Linux的领导者RedHat，也是Docker生态圈非常重要的合作伙伴。并且它支持的Fedora社区、CentOS社区会对推广Docker技术起到关键性作用。



还有就是OpenStack的建立者Rackspace，在混合云解决方案上具有全球领导地位。Docker与它的合作，可以帮助Docker技术在混合云的解决方案中得到推广。



最后就是开源PaaS项目DEIS，基于Docker和CoreOS技术构建的类如Heroku发布流程的PaaS应用。与它的合作，可以让更多的企业看到一个使用Docker技术的范本。



5. 结论



Docker的开源之路可以说是开源项目的最佳实践，它从屌丝瞬间变为高富帅的历程并不是偶然。Docker的开源之路还在进行中，从它的模式中我们可以学习到的东西并不仅仅局限于以上罗列的几点，读者可以在参与开源项目的过程中多体会，找到自己的“痛点”，然后参考Docker等开源项目的做法，相信可以很快解决你的问题。

下期预告

Docker技术基本信息已经介绍的差不多了，接下来的系列将进入实战篇。下一篇将重点讲解Docker的集成测试部署方法，并结合众场景（数据库集成、测试、审查、部署）给出参考解决方案，敬请期待！

感谢郭蕾对本文的审校和策划。

查看原文：[深入浅出Docker（三）：Docker开源之路](#)

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

HTML5、Web引擎与跨平台移动App开发

作者 余枝强

移动端跨平台应用开发是个有趣的话题。纵观该领域目前各个开发商提供的多种方案，大致可以分为三大类：

1. 基于HTML5的方案。该方案以[PhoneGap/Cordova](#)为代表。其基本思路是针对HTML5标准目前功能上的不足，补充定义了一套比较实用的API（比如硬件访问/系统交互等），然后基于平台上自带的Web引擎（比如iOS的UIWebView等），通过扩展机制实现了这些API，在此基础上再提供一套应用打包部署系统。[Intel的XDK](#)也属于此类方案。
2. 将Native API映射封装成统一语言的API的方案。该方案以[Titanium](#)、[Xamarin](#)为代表，其中Titanium提供JavaScript API，Xamarin提供C# API。这样的好处是可以较容易达到和Native API类似的能力，编程模型/方式也和原生应用相似。
3. 有行业针对性的HTML5 API方案。比如Ludei的[CocoonJS](#)就是一个比较有意思的方法，它设计了一套专门针对2D/3D游戏开发的API（支持iOS和Android）。可以认为它是HTML5图形操作的子集（Canvas + WebGL），再加上一些扩展的API比如硬件访问能力/广告/应用内购买/社交网络整合等，以实现一个完整的游戏引擎。

本文重点介绍基于HTML5的方案相比其他方案的优缺点，如何实现更好的效果，以及目前的一些进展。

HTML5方案的特点

原生API映射的方案，如Titanium、Xamarin，其优点在于功能和性能与原生系统比较接近。但是，由于不同系统原生API设计上还是会有不少差异，API的映射还是需要不少的权衡取舍。同时，由于这些API是这些厂商自定义的，谈不上什么标准，相应的开发资源（程序库/技术支持/社区等等）也相对有限。

而另一方面，标准化、开发资源的丰富则是HTML5方案最大的优点，同时第三方的HTML5框架工具比如PhoneGap/Cordova也极大促进了HTML5应用的发展，它们提供了方便的跨平台应用打包/发布服务、实用的API、灵活的扩展机制、以及积累下来的丰富的第三方API实现。而上游的W3C一旦开始支持一些新的API，PhoneGap/Cordova也可以很快沿用这些标准的API将相关能力开放出去。

HTML5方案的主要不足则在于功能和性能方面，这主要是因为HTML5应用的能力严重依赖于系统自带的Web引擎：iOS的UIWebView、Android的WebView等，此类组件的HTML5

能力相比Safari for iOS、Chrome for Android都要差一截。另外在Android平台上，由于系统碎片化比较严重，不同Android版本的Webview的HTML5能力也有较大差异，导致相应的HTML5应用一致性难以保证。

好消息是，现在已经出现一些第三方的Web引擎以提供比系统默认的Webview更好的功能和性能，而PhoneGap/Cordova也正在改进架构以便引入这些更好的第三方Web引擎。另外对于Tizen、Firefox OS这样本身就是HTML5 Runtime加上扩展API的系统而言，HTML5应用是一等公民，在功能拓展方面相比iOS、Android上会增强不少。

而第三种方案，CocoonJS的优点是专注于2D/3D游戏开发，画图性能很好，比如同时画1000个精灵也能达到60FPS，这是绝大多数的浏览器/通用的HTML5引擎目前还做不到的。这个方案的缺点在于，由于它的画图操作简化了很多路径，它无法做到和HTML5 DOM元素的互操作，而且它的HTML5能力只是一个子集，功能比较受限。目前CocoonJS针对Android也引入了另外一种模式Webview+作为补充，Webview+基于Chromium的内核加上Cordova API的支持以实现更通用的HTML5能力。

总的来说，HTML5应用的能力很大程度上依赖于Web引擎的能力。因此，无论是移动操作系统开发商还是开发工具的开发商，都持续在Web引擎的方向投入了更多的努力。

Web引擎

Web引擎目前大致可分为三种方式：

1. 浏览器，比如Safari/Chrome/UC Browser等；
2. 系统自带的Webview组件，比如上面提到的iOS UIWebview和Android Webview
3. 专门的Web Engine，比如Intel的开源项目[Crosswalk](#)、Ludei的[Webview+](#)

浏览器方式很容易理解，一个HTML5应用就是一个Web页面，用户通过浏览器打开一个URL，然后进入浏览器的全屏模式/App模式进行操作，或者是通过点击一个事先创建好的快捷方式打开应用。这种方式的性能取决于浏览器本身对HTML5的支持情况，一般来说要优于Webview组件的方式，但是问题在于不同的浏览器有差异，而且通过浏览器运行HTML5较难做到类似原生应用的体验（应用切换/权限管理/系统资源访问/整合等）以及丰富的API支持。

Webview组件方式的一般用法是以Hybrid的方式发布HTML5应用，即上述提到的PhoneGap/Cordova方案所采用的方式。其问题已经在上面提到过，主要是Webview组件本身对HTML5的支持能力不足。

专门的Web引擎可以有较好的HTML5功能和性能支持，同时有较好一致性，类似原生应用的系统整合也可以做得较好。这种方式的缺点则在于开发者需要将Web引擎与应用程序一起打包，生成的应用大小会更大，因此有的Web引擎（如Crosswalk）也提供了一种“共

享模式”，让多个应用可以共享一个Web引擎，仅当应用第一次启动并且发现系统还没有相应Web引擎时才提示用户下载安装。

目前的发展趋势是：通过PhoneGap/Cordova方式得到丰富的API支持，通过专门开发的Web引擎去提升HTML5的能力。

Crosswalk和Ludei的Webview+在概念上比较类似。Webview+是闭源的，目前还不好评估；Crosswalk由我所在的团队开发，是开源的（BSD许可协议），基于Chromium内核，着重于对HTML5功能和支持，发布周期为六周一次，支持Cordova API。

目前Crosswalk正式支持的移动操作系统包括Android和Tizen，在Android 4.0及以上的系统中使用Crosswalk的Web应用程序体验和原生应用没有区别。该引擎现在已经成为众多知名HTML5平台和应用的推荐引擎，包括[Google Mobile Chrome App](#)、[Intel XDK](#)、[Famo.us](#)和[Construct2](#)等等，未来的Cordova 4.0也计划集成该Web引擎。不过比较遗憾的是，由于iOS的限制（iOS不允许应用使用除iOS UIWebView之外第三方的JIT--即时编译引擎），目前Crosswalk也没有办法提供直接的支持，但这也许会随着HTML5更广泛的进入移动市场而发生改变。

总结

现在的HTML5 App（加上API扩展）已经可以胜任很多事了，比如教育类应用，休闲游戏等等。不过对于那些实时性要求比较高的、计算量大的（比如涉及大量的元素绘制，或并行计算等）、复杂的3D游戏，多人在线游戏/应用等还有不少差距。另外，工具方面，如何能够更高效的调试/开发/性能内存调优 HTML5应用也是另外一个需要提高的地方。不过，这些方面也在不断的演进。相信不久的将来，HTML5终会成为主流移动开发平台。

作者介绍

余枝强目前是英特尔开源技术中心的软件技术经理。主要负责HTML5 引擎 – Crosswalk 在安卓平台的开发，以及一些其他和Web有关的新技术的研发工作（如 HTML5 并行技术, 3D Camera等）。他坚信Web是未来，也非常希望和大家一起努力，让这个未来能够更快更好的到来。

感谢杨赛对本文的审校。

查看原文：[HTML5、Web引擎与跨平台移动App开发](#)

Spark 的硬件配置

作者 张逸

从MapReduce的兴起，就带来一种思路，就是希望通过大量廉价的机器来处理以前需要耗费昂贵资源的海量数据。这种方式事实上是一种架构的水平伸缩模式——真正的以量取胜。毕竟，以现在的硬件发展来看，CPU的核数、内存的容量以及海量存储硬盘，都慢慢变得低廉而高效。然而，对于商业应用的海量数据挖掘或分析来看，硬件成本依旧是开发商非常关注的。当然最好的结果是：既要马儿跑得快，还要马儿少吃草。

Spark相对于Hadoop的MapReduce而言，确乎要跑得迅捷许多。然而，Spark这种In-Memory的计算模式，是否在硬件资源尤其是内存资源的消耗上，要求更高呢？我既找不到这么多机器，也无法租用多台虚拟instance，再没法测评的情况下，只要寻求Spark的官方网站，又或者通过Google搜索。从Spark官方网站，Databricks公司Patrick Wendell的演讲以及Matei Zaharia的Spark论文，找到了一些关于Spark硬件配置的支撑数据。

Spark与存储系统

如果Spark使用HDFS作为存储系统，则可以有效地运用Spark的standalone mode cluster，让Spark与HDFS部署在同一台机器上。这种模式的部署非常简单，且读取文件的性能更高。当然，Spark对内存的使用是有要求的，需要合理分配它与HDFS的资源。因此，需要配置Spark和HDFS的环境变量，为各自的任务分配内存和CPU资源，避免相互之间的资源争用。

若HDFS的机器足够好，这种部署可以优先考虑。若数据处理的执行效率要求非常高，那么还是需要采用分离的部署模式，例如部署在Hadoop YARN集群上。

Spark对磁盘的要求

Spark是in memory的迭代式运算平台，因此它对磁盘的要求不高。Spark官方推荐为每个节点配置4-8块磁盘，且并不需要配置为RAID（即将磁盘作为单独的mount point）。然后，通过配置spark.local.dir来指定磁盘列表。

Spark对内存的要求

Spark虽然是in memory的运算平台，但从官方资料看，似乎本身对内存的要求并不是特别苛刻。官方网站只是要求内存8GB之上即可（Impala要求机器配置在128GB）。当然，

真正要高效处理，仍然是内存越大越好。若内存超过200GB，则需要当心，因为JVM对超过200GB的内存管理存在问题，需要特别的配置。

内存容量足够大，还得真正分给了Spark才行。Spark建议需要提供至少75%的内存空间分配给Spark，至于其余的内存空间，则分配给操作系统与buffer cache。这就需要部署Spark的机器足够干净。

考虑内存消耗问题，倘若我们要处理的数据仅仅是进行一次处理，用完即丢弃，就应该避免使用cache或persist，从而降低对内存的损耗。若确实需要将数据加载到内存中，而内存又不足以加载，则可以设置Storage Level。0.9版本的Spark提供了三种Storage Level：MEMORY_ONLY（这是默认值），MEMORY_AND_DISK，以及DISK_ONLY。

关于数据的持久化，Spark默认是持久化到内存中。但它也提供了三种持久化RDD的存储方式：

- in-memory storage as deserialized Java objects
- in-memory storage as serialised data
- on-disk storage

第一种存储方式性能最优，第二种方式则对RDD的展现方式(Representing)提供了扩展，第三种方式则用于内存不足时。

然而，在最新版（V1.0.2）的Spark中，提供了更多的Storage Level选择。一个值得注意的选项是OFF_HEAP，它能够将RDD以序列化格式存储到Tachyon中。相比MEMORY_ONLY_SER，这一选项能够减少执行垃圾回收，使Spark的执行器(executor)更小，并能共享内存池。Tachyon是一个基于内存的分布式文件系统，性能远超HDFS。Tachyon与Spark同源同宗，都烙有伯克利AMPLab的印记。目前，Tachyon的版本为0.5.0，还处于实验阶段。

注意，RDDs是Lazy的，在执行Transformation操作如map、filter时，并不会提交Job，只有在执行Action操作如count、first时，才会执行Job，此时才会进行数据的加载。当然，对于一些shuffle操作，例如reduceByKey，虽然仅是Transformation操作，但它在执行时会将一些中间数据进行持久化，而无需显式调用persist()函数。这是为了应对当节点出现故障时，能够避免针对大量数据进行重计算。要计算Spark加载的Dataset大小，可以通过Spark提供的Web UI Monitoring工具来帮助分析与判断。

Spark的RDD是具有分区(partition)的，Spark并非是将整个RDD一次性加载到内存中。Spark针对partition提供了eviction policy，这一Policy采用了LRU(Least Recently Used)机制。当一个新的RDD分区需要计算时，如果没有合适的空间存储，就会根据LRU策略，将最少访问的RDD分区弹出，除非这个新分区与最少访问的分区属于同一个RDD。这也在一定程度上缓和了对内存的消耗。

Spark对内存的消耗主要分为三部分：

1. 数据集中对象的大小；
2. 访问这些对象的内存消耗；
3. 垃圾回收GC的消耗。

一个通常的内存消耗计算方法是：内存消耗大小= 对象字段中原生数据 * (2~5)。这是因为Spark运行在JVM之上，操作的Java对象都有定义的“object header”，而数据结构（如Map, LinkedList）对象自身也需要占用内存空间。此外，对于存储在数据结构中的基本类型，还需要装箱（Boxing）。Spark也提供了一些内存调优机制，例如执行对象的序列化，可以释放一部分内存空间。还可以通过为JVM设置flag来标记存放的字节数（选择4个字节而非8个字节）。在JDK 7下，还可以做更多优化，例如对字符编码的设置。这些配置都可以在spark-env.sh中设置。

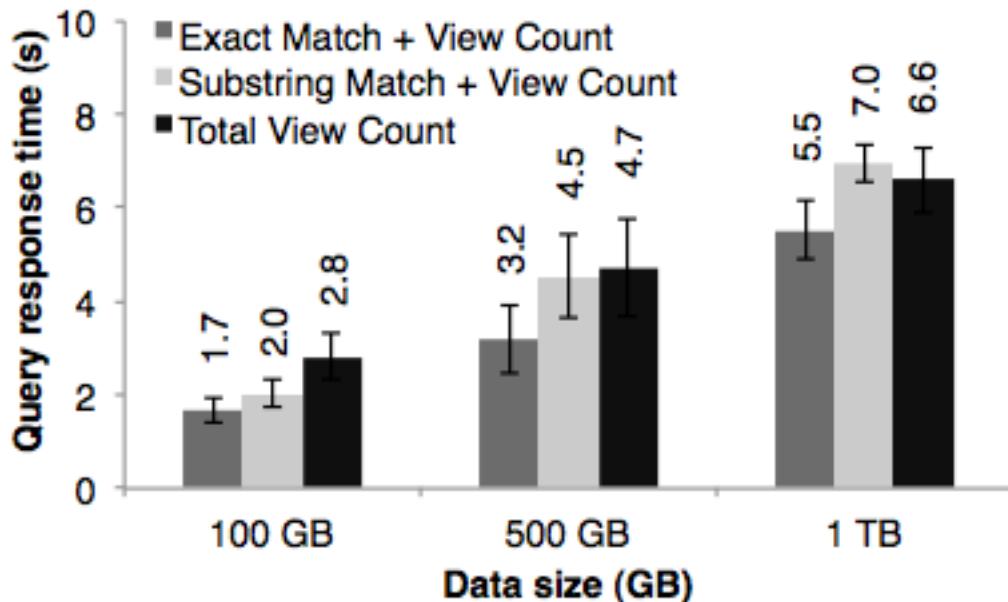
Spark对网络的要求

Spark属于网络绑定型系统，因而建议使用10G及以上的网络带宽。

Spark对CPU的要求

Spark可以支持一台机器扩展至数十个CPU core，它实现的是线程之间最小共享。若内存足够大，则制约运算性能的就是网络带宽与CPU数。

Spark官方利用Amazon EC2的环境对Spark进行了基准测评。例如，在交互方式下进行数据挖掘（Interative Data Mining），租用Amazon EC2的100个实例，配置为8核、68GB的内存。对1TB的维基百科页面查阅日志（维基百科两年的数据）进行数据挖掘。在查询时，针对整个输入数据进行全扫描，只需要耗费5-7秒的时间。如下图所示：



在Matei Zaharia的Spark论文中还给出了一些使用Spark的真实案例。视频处理公司Conviva，使用Spark将数据子集加载到RDD中。报道说明，对于200GB压缩过的数据进行查询和聚合操作，并运行在两台Spark机器上，占用内存为96GB，执行完全部操作需要耗费30分钟左右的时间。同比情况下，Hadoop需要耗费20小时。注意：之所以200GB的压缩数据只占用96GB内存，是因为RDD的处理方式，使得我们可以只加载匹配客户过滤的行和列，而非所有压缩数据。

查看原文：[Spark的硬件配置](#)

通过度量查询质量构建更佳的搜索引擎

作者 Ellen M. Voorhees, Paul Over, Ian Soboroff , 译者 陈菲

本文最先发表于[IT Professional](#)杂志，现由InfoQ & IEEE Computer Society在此为您呈现。

搜索引擎构建于对真实测试案例标准集的使用，该标准集允许开发人员度量其它替代方法的相对有效性。这篇文章讨论了NIST的文本检索会议（TREC）项目，该项目创建了用于度量查询结果质量的基础架构。

我们总是认为以母语进行的文本文件查询是有保障的，但是像Yahoo、Google和Bing这样的网页搜索引擎并非一天构成，而且网页内容也不是我们需要查询的唯一领域。随着数据变得越来越无处不在，搜索需求也相应地扩大了。人们为了不同的目的（比如：重新定位已知数据项、回答特定问题、学习特定问题、监测数据流、及浏览）在不同媒体（比如：文本、网页、Tweet、语音录音、静态图像和视频）中搜索所需内容。大多数情况下，用于支持这些不同搜索类型的技术还在不断完善中。可搜索技术是如何发展的？搜索引擎开发人员怎么知道哪些可行，又为什么可行呢？

来自大型、多元化搜索社区的参与者对标准的、真实测试中的搜索引擎性能的仔细度量已被证实是至关重要的；通过文本检索会议（TREC）项目，美国国家标准及技术研究所（NIST）在过去四分之一世纪以来一直通过收集社区评估促进着搜索和与搜索相关技术的发展。

TREC的来源

搜索算法通常通过测试集，即在基准任务上对比各个替代方法而发展起来的。第一个测试集是20世纪60年代从Cranfield学院的一系列关于航空索引语言的实验中获取而来的¹。Cranfield测试集包含一组航空杂志文章摘要、一组针对这些摘要的查询以及每个查询的正确答案要点。从今天的标准看，它可能微不足道，但Cranfield集打破了当时的记录，为信息检索系统创建了第一个共享的度量工具。研究人员可以自己编写搜索引擎来查询所要的摘要，其返回值可以通过对比答案要点来度量。

其它研究组开始遵循由Cranfield测试提出的实验型研究法，生成了其它多个用于70和80年代的测试集。但到了90年代，大家开始对该方法越来越不满意。尽管有些研究组使用了相同的测试集，但在以下几方面却各持己见：是否使用相同数据，是否使用相同评估度量，是否跨查询系统对比结果。商业搜索引擎公司并未将从查询系统中取得的研究结果整合到他们的产品中，因为在当时他们觉得研究界所用的测试集太小了，并不值得借用。

面对这一不满，NIST被要求构建一个大型的用于评估文本检索技术的测试集，在当时该技术已发展为美国国防部高级研究计划局（DARPA）Tipster项目²的一部分。NIST同意以研讨会形式来构建大型测试集，同时也支持关于测试集使用方面等更大问题的检测。该研讨会就是1992年举行的第一次TREC会议，此后每年都举办一次TREC会议。TREC在早期就完成了构建大型测试集的最初目标；事实上到目前为止，TREC已经构建了几十个正被整个国际研究界使用的测试集。TREC更大的成就是对研究范例的建立和验证，并在此后的每年都继续将其拓展到新任务和应用程序的具体内容中去。

社区评估

该研究范例以基于社区的评估为中心，称为“合作竞争（coopetitions）”，这一新词强调了竞争者间的合作，从而创造出更大的利益。

该范式的主要元素是评估任务，通常情况下，它是一个用户任务抽象，明确定义了我们对系统的期望。与评估项目相关联的是一或多个度量，它们在反映了系统的响应质量的同时，也是所有基础架构用来计算可构建度量标准的手段。其评估方法概括了任务、度量、以及对度量分数有效解释的声明。一个标准的评估方法允许横跨不同系统间进行结果比较，这就是检索竞赛无获奖者为什么这么重要了；相对于其他研究组所能解决的，它促进了更广泛类型研究成果的整合。

谈到范式的具体实例，我们可以考虑第一个TREC中的主要ad hoc项目，该项目拓展了当时的Cranfield方法。该ad hoc评估项目还检索了相关文档（更确切地说，创建了一个文档列表将所有相关文档罗列于无关文档之前），然后给出一个文档集和称为话题的对信息需求自然语言的声明。该检索输出以准确率（检索出的相关文档数与检索出的文档总数的比率）和召回率（检索出的相关文档数和文档库中所有相关文档数的比率）为指标，为每个已知话题（换句话说，就是答案要点）提供相关文档集。TREC的创新在于利用pooling³为大型文档集构建相关集合。

该pool是所有参与系统对给定话题的检索而查询出来的前X个文档的集合。只有来自话题pool中的文件才会被人工评审员采用并鉴定其相关性，其它的文档则与有效性评分的计算无关。尽管只参照了整个测试集中的一小部分，但是随后的测试证明在TREC中执行的pooling在文档集中所发现的文档绝大部分都是相关的。另外，该测试进一步验证了在通常情况下，这些在测试集上获得更高分数、构建于pooling上的检索系统往往比那些分数较低的系统在实际应用中更有效⁴。同时该测试也暴露了通过计算测试集所得分数只有有限的有效应用。因为分数的绝对值由多种因素决定，而非仅仅是检索系统（比如：使用不同人工评审员通常对分数会有一定的影响）。只有比较其它系统在完全相同测试集上计算出的分数才有效。这就意味着对比不同年度的TREC分数是无效的，因为每个TREC构建的测试集都是新的（不同的）。想让pooling成为有效的策略，在pool中拥有类型广泛的检索方法起着非常必要的作用。因此，TREC的社区因素 — 使用了多种检索方法用于检索不同文档集 — 是创建良好测试集的重要因素。社区因素同时也是TREC在其它方面成功的重要因素。TREC只有在所有检索方法都呈现时，才能检验当前的技术。一年一度的TREC会议不仅促进了不同研究组间，而且还促进不同研究开发组织间的技术交流。

该年度会议还为解决方法论方面的问题提供了有效的机制。最后，社区成员往往是新项目数据和用例的来源。

在TREC开始之初，大家还怀疑这个构建于实验室的统计系统（相对于在手动索引集合上使用布尔搜索的操作系统）是否能真正有效地从大型集合中检索出文档来。TREC中的ad hoc项目证明了90年代初的搜索引擎不仅确实拓展到了大型集合范畴，也不断得到了改进。其有效性无论在TREC测试集实验室中，还是在当下集合了该技术的操作系统中都得到了证明。此外，该技术现在所应用的测试集要比当初在1992年设想的大得多。网页搜索引擎是统计技术能力最好的例子。搜索引擎为用户带来他们想要的查询信息的能力已经成为网站成功的基础。正如之前所说的，搜索有效性的改进并不能简单地由每年的TREC分数来判断。然而，SMART搜索系统开发人员会将每八个他们曾经使用过的TREC ad hoc项目冻结起来，并保存系统副本⁵。每个TREC后，他们会在每个系统上执行所有测试集。针对每个测试集，新版SMART系统总是比旧版的更有效，而且分数也比旧版的大概要高出一倍。尽管该证据只能证实一个系统，但在每个TREC中，SMART系统结果始终追踪其它系统的成绩，因此SMART结果完全可以代表当前技术水平。

发展分支

尽管TREC的最初意图只是简单地为ad hoc检索构建一两个大型测试集，以及关于pooling方法论方面的问题的探索；但很快ad hoc项目就明显展露出可在多个方面调整发展的特性。从不同方向衍生出来的新项目都与经典项目息息相关，但又不同到足以要求评估方法论作出一定的变化。因此TREC引进了Track项目，从而每个给定的TREC都包含了多个检索子项目，每个子项目又是自身评估挑战的焦点。图1显示不同年度TREC所执行的Track项目，根据维度将Track项目分组从而将它们区分开来。Track项目位于下图的左边，展示了TREC所解决问题的宽度，而位于右边的单独Track项目则显示了项目在给定问题领域中的进展。

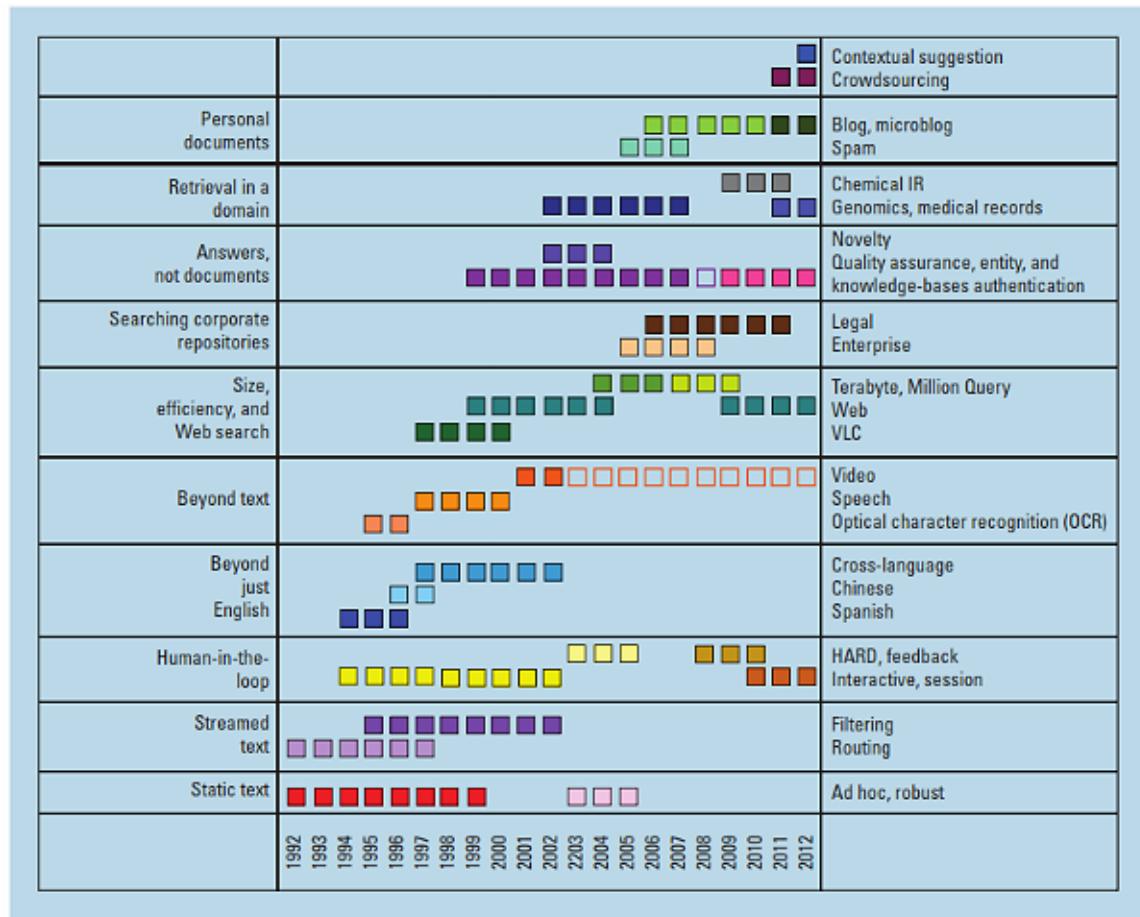


图1. 文本检索会议 (TREC) Track项目及它们的所属年份。Track项目名字位于右边，位于左边的则是Track项目关注的。空框所代表的Track项目是被分离出来成为一种评估方式用于运行当年的Track项目。颜色相同的Track项目则彼此密切相关。

现在，每个TREC会包含七或八个经常变换的track项目以保持新鲜度，及支持新的社区。多个TREC track项目已经是自己领域的首个大规模评估。在这种情况下，Track项目已经建立了相应的研究社区，并创建了首个用于支持其研究领域的特定测试集。有几个Track项目已经从TREC中分离出来，感兴趣的社区建立了自己的评估会议。比如：评估论坛会议和实验室（the Conference and Labs of the Evaluation Forum, [CLEF](#)）就于2000年从TREC中脱离出来，在欧洲拓展了关于跨语言检索的评估，此后它不断扩大，不仅包含了多语言，也涵盖了多模式（文本、图像和视频）信息。其它会议尽管没有从TREC中直接分离开来，却受到TREC的启发，将方法论拓展到其它领域。其中包括NII测试基地和信息访问研究社区（NII Testbeds and Community for Information Access Research）（NTCIR, [research.nii.ac.jp/ntcir](#)）关注了中文、日文和韩文文本；XML检索评估倡议（INEX, [inex.mmci.uni-saarland.de](#)）；以及信息检索评估论坛（Forum for Information Retrieval Evaluation, FIRE）关注了印度次大陆的语言。

由于空间上的局限性，我们无法为每个TREC Track项目展开哪怕是粗略的讨论。因此我们抽样强调以下一些Track项目 — 过滤、问题解答和法律电子取证 — 它们从某种程度

上部分解决了这些特别紧迫的搜索问题，其中还有视频检索Track项目，由于数字视频日益普及，它已发展出自己的NIST研讨会系列：TRECVID。

过滤

TREC的最初几年仅有Ad hoc和routing两个项目。Routing项目用于模拟用户监控文件流、选择相关文档及忽略无关文档。在TREC-4中，routing演变为过滤 — 一个更为复杂，却更真实的场景。正如电子邮件过滤系统实时处理传入的邮件流以删除垃圾邮件和执行归档规则一样，信息过滤系统处理传入的文件流，然后根据用户对之前交付文档的回馈而建立的用户兴趣模型配置文件，决定是否要将这些信息发送给用户⁶。

鉴于routing评估项目允许系统以批处理的集合形式处理所有文件，该过滤评估项目要求系统在文档到达文件流时就处理，并采用在线用户模型。如果系统选择显示文档给用户，还附有对该文件相关度的评价，那么该系统就被赋予了评估（模拟实时用户的回馈）。系统就可以根据这些信息进行及时的自我调整。如果系统决定不将这些文档显示给用户，那就是缺失了相关信息。过滤系统的有效性通过使用实用模型，一个基于返回相关文件数、并根据其返回的不相关文档数而被减分的系统。

过滤Track项目让与会者更好地理解过滤项目的执行难度。在实用模型中，系统因返回不相关信息而受处罚。如同现实生活一样，在过滤track集合时，在数以百万计的文档流中往往只有一小部分文档是相关的。因此，一些精明的系统会因为从未返回任何文档而得分很高 — 换句话说，它们决定不冒风险去浪费使用者的时间。由于系统在数据流初期只有少量培训数据，其初始性能往往比较差。为了完善该用户模型，系统必须给用户提供很多有希望却最终无关的文档。

该系统必须能够从反馈的原始成本中恢复，这些反馈经历了从执行特别好很快到分数特别好这一过程。

问题解答

尽管主题相关的文档列表毫无疑问是有益的，其本身所提供的信息就多于用户想要检测的。TREC问题-解答Track项目于1999年引入，主要关注返回问题确切答案这一类问题。最初的问题解答Track项目关注事实型问题 — 即那些答案短且基于事实的问题，比如：“泰姬陵在哪里？”随后，Track项目又引入更为复杂的问题类型，如：列表问题（其答案是所请求类型实例的不同集合。比如：“有哪些演员曾在《屋顶上的小提琴手（Fiddler on the Roof）》这部电影中扮演过Tevye这一角色？”）；以及定义型或传记型问题（比如：“什么是黄金降落伞？”或“谁是弗拉德三世（Vlad the Impaler）？”）。

问题解答Track项目是开放域问题解答系统的首个大型评估，而且利用了从其它TREC观察而获得的测试集评估优势来处理问题解答项目。该Track为检索和自然语言处理研究社区建立了通用项目，带来了问题解答研究的复兴。由于研究人员将复杂语言处理结合到

问题解答系统中，该研究潮流在自然语言自动理解上实现了显著的进步。比如：Watson，IBM的Jeopardy-playing电脑系统，TREC问题解答Track项目中就有该公司曾参与的原型⁷。

电子取证

法律Track项目始于2006年，专门关注电子取证问题，电子存储信息的有效生产是诉讼和监管设置的证据。当今组织更多地依赖电子记录，而非纸质记录。但相应数据量及其潜在的短暂本质已使传统司法发现流程和实践不堪重负。因此我们需要新的处理电子数据的发现实践。当该Track任务开始时，涉及的诉讼双方通常会讨论出一个布尔表达式用来定义发现结果集。然后再由人工检查每个检索文档，从而判定发现请求的响应能力。该Track的目的在于评估该基线方法和发现的其它查询技术的有效性。该Track使用了假设的投诉和相应的请求来生成由执业律师开发出作为主题的文档。指定的“主题权威”扮演了某案件中首席律师的角色，陈述关于具体是什么让文档响应请求的总体战略和指导方针。特定文档的相关性判断由法律专业人士来评估，他们会更根据自身的典型工作实践来审查文档。

该Track项目对法律社区有着重大的影响，其中包括司法意见中的引文（见 en.wikipedia.org/wiki/Paul_W._Grimm）。它的主要成果就是引发实时对话，该对话通过对迭代流程的展示指出应该完成哪个电子发现，这个迭代查询会包含一个人工到查询循环中，其效力往往优于一次性检索。在信息检索这方面，Track在标准测试集评估方法上表现出了不足。为了方便稳定的评估，尤其当使用pooling中构建的测试集时，标准方法所依赖的平均有效性是从一组每个主题只有一小部分相关文档的主题中获取的。但是当返回文档数量很大时，电子发现中的真实案例则从单个响应集中衡量其有效性。

视频

虽然不属于TREC研讨会范畴，但依然属于NIST，TRECVID从它出现于TREC Track项目中以来，已经以多种形式发展起来(请看图2)。TRECVID创建于2001年，将TREC/Cranfield理念扩展到基于具体内容的视频分析和检索。两年后，TRECVID成立了独立的系列研讨会，并开始了使用电视广播新闻（英文、中文和阿拉伯语）及4年为一周期的循环，还将测试数据从50小时增加三倍到了150小时。系统任务包含使用多媒体主题的查询，高层次特征提取，拍摄和传记边界确定，和相机运动检测。

2007年开始了以3年为周期的循环，采用来自荷兰声音和视觉研究所(Netherlands Institute for Sound and Vision)的教育和文化节目。测试数据也在2009年增加到280小时。针对BBC工作样品（未经编辑的节目素材）添加了一个综述任务，针对由英国内政部(UK Home Office)提供的机场监控录像添加了事件监测任务。从2010年开始，TRECVID就开始关注不同的、通常非专业的网络视频，这些由不同社区捐献的视频在数量上从几百到几千小时不等，在拓展了研究和功能/时间检测任务的同时，也在评估中添加了已知事项和实例查询（见图2a）。

TRECVID研究人员已经为全球科学同行在对艺术状态的判断上做出了显著的贡献。2009年，由都柏林城市大学的图书馆学家进行的文献计量学研究发现TRECVID参与者在2003年和2009年间生成了310个（未经审阅的）研讨会文章，与此同时，还生成了2073篇同行评审期刊文章和会议论文⁸。

尽管测试数据的变化导致我们难以度量系统的改进，但阿姆斯特丹大学的MediaMill团队在2010年进行的实验证实了，过去三年里特征检测有了三倍的改善 — 这对一个系统来说，已经是TRECVID中表现最佳的执行者⁹。2010年和2011年的检测测试数据是相同的，但测试查询（11, 256）却是随机创建的，这允许了系统间的比较。顶级团队在检测和定位上2011年的平均分数要比2010年好。

TRECVID研讨会系列汇集了来自多元化社区自我资助的研究人员，在多个领域激发起有趣工作的任务深深地吸引着他们。与此同时，研究人员也被数据和计分流程的可用性所吸引，因为这些允许他们关注研究任务，而非基础架构。与此同时，他们还被科学比较的开放论坛所吸引。来自世界各地能参与到竞赛项目中的团队数量也在增长，新的顶级执行者也不断出现。越来越多的知识分子将注意力放到了持久性问题上，比如：从视频中提取有意义的信息，只能从长远角度才有增加进展的可能性。

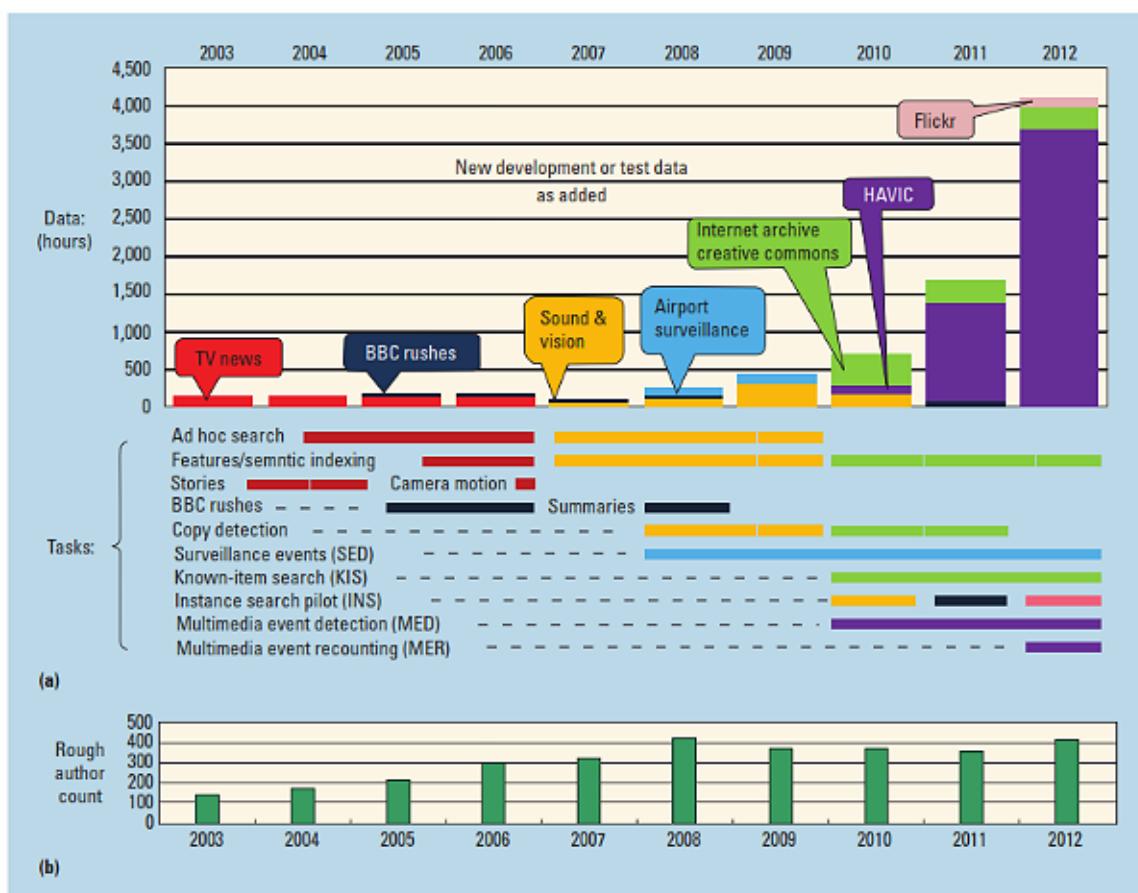


图2. TRECVID在（a）数据、项目和（b）参与者方面的演变。TRECVID中使用的数字视频包括广播新闻报道、未经编辑的电视节目素材、监控录像和非专业互联网视频。不

同数据类型支持不同的项目，比如：传统的ad hoc查询，复制检测和视频序列中特定活动模式的识别。TRECVID程序中文档的作者数量是TRECVID参与宽度的一个度量。

在其作为独立研讨会系列的头三年，TRECVID社区发展迅速，申请量从20组增加三倍到了60组，其中有40组至少完成了一个项目。从2007到2009，申请增加到100组左右，其中有60组已组队完成，社区参与还以该水平持续发展到现在。对研讨会文档合著者的粗略计算显示了大概有400名研究人员参与到每年的TRECVID实验中（见图2b）。虽然学术团队占据了主导地位，商业研究实验室却一直是组合的一部分。欧洲和亚洲的地区参与者最多，北美则紧随其后。

TRECVID社区所贡献的不仅限于研究。他们还为评估基础架构贡献了各种关键部分，其中包括地面实况注释系统和判断、镜头分割、自动语音识别、评估软件、数据托管和训练有素的探测器。如果没有这种合作，TRECVID是无法实现的。

以下是从2009年信息检索中基础和趋势¹⁰(Foundations and Trends in Information Retrieval)的一篇评论文章中找到的结论：

由于TRECVID在领域中广泛的接受度，从而获得了大量国际团队的参与，他们来自大学、研究机构和企业研究实验室。TRECVID基准可作为实际标准用来评估基于概念的视频搜索研究的性能。实际上该基准已经在视频搜索社区产生了巨大的影响，生成了大量的视频检索系统和出版物汇报了TRECVID中进行的实验。

包括利用多媒体搜索主题的创新自动决定将镜头作为检索的基本单元（允许有效地判断系统的输出），将平均精度应用程序作为视频查询和概念检测有效性的度量，采用基于成本的度量用于拷贝检测，及使用实用方法评估简单总结。

跨研究团队技术转换发生于TRECVID和更广泛的视频分析社区内。用于某年某系统中的方法通常会在下一年被其它系统做出一定修改后采用。由于实验室是通过原型系统练习的，因此TRECVID结果往往具有指导性，而非结论性。随着算法反复证明自己可以是多种系统的一部分，并能解决不断变化的测试数据，特定方法的可信证据逐步增多。相当数量的工程在某种情况下，需要做可用性测试以将实验室内的成功实例应用到现实世界的应用程序中。

荷兰声音和视觉研究所，是TRECVID主要的数据和用例捐赠者之一，是TRECVID记录在案、允许其加入大范围的研究者社区以低成本在他们自己的数据上探索他们所感兴趣的项目。与附近另一个TRECVID参与者（阿姆斯特丹大学）在所需的工程设计和用户测试从原型到可运行系统的转换上有更密切的合作，使得有前景的技术得到了更进一步的探讨¹¹。

过渡到现实世界运用的一个具体例子是荷兰一个公司对功能/概念探测器的开发和授权，它们将被集成到软件工具中允许警方搜索非法材料中应被没收的视频¹²。

向前迈进

TREC在常见问题集上所采用的评估竞争技术方法已被证明是提高当前技术及加速技术转移的一种强有力的方式。Google的首席经济学家Hal Varian曾在其Google博客2008年的一篇博文¹³中这样描述TREC的影响：

TREC在信息检索上对数据恢复的研究所具有的标准性、广泛可用性和精心构建的数据集为本领域进一步创新奠定了基础。一年一度的TREC会议促进了合作、创新和程度可控的竞争（当然还有吹牛的权利），从而引导了更好的信息检索。

NIST委员会RTI国际展开了对TREC影响更详细的研究¹⁴。在数量方面，该研究估算出在TREC上所投资的每一美元，信息检索研究人员能得到的累积回报利润在3到5美元之间。该项研究还列举了各种定性收益，总结起来其中一部分包括以下这些：

TREC活动同时还有其它好处，只是并不属于经济范畴。TREC帮助教育研究生和本科生，其中有一些还去领导IR公司，其它一些留在学术界教书和进行研究。TREC使得IR产品质量和可用性得到了提升 — 我们的研究显示TREC促进了IR研究大范围的拓展，从而带来了高质量的应用程序，比如：网页搜索，企业搜索和领域特定搜索产品和服务（例如：基因组分析）。更具体地说，该研究估算出从1999年到2009年间所观察到的网页搜索产品超过200%的提升中大约有三分之一是由于TREC的存在而带来的。

尽管有了这样的成功，我们依然有很多工作需要去做。虽然内容存储变得越来越大了，电脑依然无法真正理解由人类运用而产生的内容。

在可预见的将来TREC和TRECVID研讨会还会继续，关注于搜索研究社区和更广泛用户社区上具有重大影响的问题的搜索研究。

TREC网页[1](#)和[2](#)有关于TREC的丰富资料，其中包括每个研讨会的完整流程和关于如何获取测试集的详情。每个冬季想要参与的组织可以通过响应号召邀请参与到TREC中。

声明

某些商业实体、设备或材料可能会在本文中被识别出用于充分描述实验型流程或概念。这种识别的目的并不在于暗示他们被美国国家标准及技术研究所推荐或认可，也不在于暗示某些实体、材料或设备是解决问题的最好工具。

参考资料

1. C.W. Cleverdon, “The Cranfield Tests on Index Language Devices,” *Aslib Proc.*, vol. 19, no. 6, 1967, pp. 173–192. (Reprinted in *Readings in Information Retrieval*, K. Spärck-Jones and P. Willett, eds., Morgan Kaufmann, 1997.)
2. D. Harman, “The DARPA TIPSTER Project,” *ACM SIGIR Forum*, vol. 26, no. 2, 1992, pp. 26–28.

3. K. Spärck Jones and C. van Rijsbergen, *Report on the Need for and Provision of an “Ideal” Information Retrieval Test Collection*, report 5266, British Library Research and Development, Computer Laboratory, Univ. of Cambridge, 1975.
4. C. Buckley and E.M. Voorhees, “Retrieval System Evaluation,” *TREC: Experiment and Evaluation in Information Retrieval*, E.M. Voorhees and D.K. Harman, eds., MIT Press, 2005, chap. 3, pp. 53–75.
5. C. Buckley and J. Walz, “SMART at TREC-8,” *Proc. 8th Text Retrieval Conf. (TREC 99)*, 1999, pp. 577–582.
6. S. Robertson and J. Callan, “Routing and Filtering,” *TREC: Experiment and Evaluation in Information Retrieval*, E.M. Voorhees and D.K. Harman, eds., MIT Press, chap. 5, 2005, pp. 99–122.
7. D. Ferrucci et al., “Building Watson: An Overview of the DeepQA Project,” *AI Magazine*, vol. 31, no. 3, 2010, pp. 59–79.
8. C.V. Thornley et al., “The Scholarly Impact of TRECVID (2003–2009),” *J. Am. Soc. of Information Science and Technology*, vol. 62, no. 4, 2011, pp. 613–627.
9. C.G.M. Snoek et al., “Any Hope for Cross-Domain Concept Detection in Internet Video,” MediaMill TRECVID 2010, www-nlpir.nist.gov/projects/tvpubs/tv10/slides/mediamill.tv10.slides.pdf.
10. C.G.M. Snoek and M. Worring, “Concept-based Video Retrieval,” *Foundations and Trends in Information Retrieval*, vol. 2, no. 4, 2009, pp. 215–322.
11. J. Oomen et al., “Symbiosis Between the TRECVID Benchmark and Video Libraries at the Netherlands Institute for Sound and Vision,” *Int'l J. Digital Libraries*, vol. 13, no. 2, 2013, pp. 91–104.
12. P. Over, “Instance Search, Copy Detection, Semantic Indexing @ TRECVID,” US Nat'l Inst. Standards and Technology, Nov. 2012, www.nist.gov/oles/upload/8-Over_Paul-TRECVID.pdf.
13. H. Varian, “Why Data Matters,” blog, 4 Mar. 2008, <http://googleblog.blogspot.com/2008/03/why-data-matters.html>.
14. RTI Int'l, Economic Impact Assessment of NIST's Text Retrieval Conf. (TREC) program, 2010, www.nist.gov/director/planning/impact_assessment.cfm.

关于作者

Ellen Voorhees是美国国家标准及技术研究所的一名计算机科学家，她的主要工作职责在于管理TREC项目。她的研究关注于开发和验证合理的评估模式用于度量不同用户研究项目和自然语言处理项目的系统有效性。Voorhees从美国康奈尔大学获得了计算机科学博士学位，她在信息访问上的工作曾获得三个专利；于此同时，她还是西门子公司研究机构（Siemens Corporate Research）的技术成员之一。可以通过以下邮箱联系到Ellen：ellen.voorhees@nist.gov。

Paul Over是美国国家标准及技术研究所的一名计算机科学家，TREC视频检索（TREC VID）的项目领导人。他曾经在NIST中负责过文本检索评估（TREC）中交互文本检索系统的评估，也是文本摘要技术评估中自然语言处理的研究人员。总的说来，他曾发布过多篇关于视频片段、摘要和研究不同话题的文章。他在2011年获得了美国商务部颁发的杰出联邦服务的铜牌。可以通过以下邮箱联系到Paul：over@nist.gov。

Ian Soboroff是美国国家标准及技术研究所（NIST）检索组的一名计算机科学家和经理。他当前的研究兴趣包括为社会媒体环境和非传统检索项目构建测试集。Soboroff已为大范围的数据和用户任务开发了评估方法和测试集。可以通过以下邮件联系到Ian：ian.soboroff@nist.gov。

本文最先发布于*IT Professional*杂志。*IT Professional*提供了关于当今战略技术问题可靠的、并由同行审评的信息。为了满足运行可靠且灵活企业所面临的挑战，IT经理和技术管理者依赖IT Pro获取最先进的解决方案。

查看英文原文：[Building Better Search Engines by Measuring Search Quality](#)

查看原文：[通过度量查询质量构建更佳的搜索引擎](#)

高密度 Java 应用部署的一些实践

作者 李三红

传统的Java应用部署模式，一般遵循“硬件->操作系统->JVM->Java应用”这种自底向上的部署结构，其中JEE应用可以细化为“硬件->操作系统->JVM->JEE容器->JEE应用”的部署结构。这种部署结构往往比较重，操作系统、JVM和JEE容器造成的overhead很高，而很多时候一个Java应用并不需要跑满整个硬件的资源，导致这种模式的资源利用率是比较低的。

而另一方面，硬件虚拟化技术逐渐成熟：VMware Hypervisor、Xen、KVM、Power LPAR等技术能够帮助我们在同一个硬件上部署多个操作系统实例，而时下流行的OS Container技术如LXC、Docker等，则是把操作系统虚拟化为多个实例，实现更轻量级的虚拟化。无论哪个层面的虚拟化，其目的都是对资源利用率更加高效的追求，从而成为如今构建云计算平台底层架构的基础技术。

Java应用也可以通过同样的思路来实现高密度的部署。JVM虚拟化是比OS虚拟化更高一层的做法，可以更大程度的提高资源利用率，降低平均应用的部署成本。本文将介绍 Multi-tenant JVM这一方案实现高密度Java应用部署的一些特点和思路。

背景介绍

早在2004年，Sun公司就提出过Java应用虚拟化这方面的想法。当时Grzegorz Czajkowski领导了一个叫做巴塞罗那的研究项目，该项目基于Java HotSpot虚拟1.5版本开发了 Multi-Tasking Virtual Machine（MVM）。MVM的目的旨在提高Java程序的启动速度，节省内存开销。不过自从Sun被甲骨文收购后，我们没有听到关于该项目的任何新的进展。

尽管我们没有看到MVM成功产品化，不过它却留下两个JSR规范：[JSR121](#)和[JSR284](#)。对于JSR284，目前在java.net上有一个实现它的孵化项目。

从2009年开始，我所在的IBM Java团队开始研究Java应用的SaaS化方案，即让一个应用实例服务于多个租户。为了保证多个租户在使用同一个应用实例时候数据的隔离，该方案在应用这个层面做了一些Bytecode Instrument（BCI）的工作，主要通过改写getstatic/putstatic使每个租户有独立的类的静态数据拷贝而没有相互影响。但是，该方案在Bytecode层面更改带来的额外性能开销，以及Java Reflection等访问带来的安全性/正确性的问题。而且，除了数据上的隔离，也需要针对关键性的资源譬如CPU、Heap、IO等资源的使用进行管理，于是该方案下沉到了JVM层面，形成现在的[多租户JVM（Multi-tenant JVM）方案](#)。

Multi-tenant JVM是JVM层面的虚拟化，其思路是把多个Java应用部署在同一个JVM上，让这些应用共享底层的GC、JIT、Java运行时库等基础组件。除了IBM的团队之外，爱尔兰的[Waratek公司](#)也实现了多租户的JVM。和IBM Multi-tenant JVM类似，Waratek允许多个应用运行在同一个CloudVM上，每一个应用运行在一个叫Java Virtual Container (JVC) 的容器里。从现有公开的资料开看，IBM Multi-tenant JVM是基于Java 7的，而Waratek是基于Java 6的，两者支持的CPU架构和平台也有所不同。

此外，JEE方面在两年前也有讨论[计划增加对PaaS和多租户的支持](#)，这项提议旨在定义PaaS环境下如何使得JEE应用支持多租户，保证不同租户在使用这些应用时相互隔离，以及资源方面的管理（如JMS资源），不过该项提议已经推迟到JEE 8。

除了提升部署密度之外，多租户的另一项好处在于应用启动的加速。快速的程序启动受益于不同的应用共享同一个JVM，我们称之为javad。Java核心的类库在javad运行后，不再需要被重新装载和定义。你也许可以用Nailgun来加速你的启动时间，但Nailgun的问题是没有安全的数据隔离，这包括类的静态数据以及Java属性值，而且Nailgun在易用性等方面也不如Multi-tenant JVM。

多租户JVM的实现思路

跟传统JVM相比，多租户JVM的主要工作围绕隔离而进行，其针对JVM/JDK的改动主要实现三个方面的目标：

1. 租户之间的数据隔离
2. Java类库支持多租户语境
3. 资源管理隔离

租户之间的数据隔离

让每个租户应用拥有独立的类静态数据拷贝，这个目标主要通过修改getstatic/putstatic字节码指令实现。下面是一个简单的例子：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

每一个运行在Multi-tenant JVM上的程序都有不同System.out实例。就java.lang.System内部实现来说，out是其类静态变量：

```
public final class System {
```

```
// The standard input, output, and error streams.  
// Typically, these are connected to the shell which  
// ran the Java program.  
/**  
 * Default input stream  
 */  
public static final InputStream in = null;  
/**  
 * Default output stream  
 */  
public static final PrintStream out = null;  
/**  
 * Default error output stream  
 */  
public static final PrintStream err = null;  
.....  
}
```

Multi-tenant JVM对于标准的JVM行为进行的更改如下：

- 每一个租户第一次使用java/lang/System时，都会触发它的初始化，也就是<clinit>。而一般的JVM，java/lang/System只会被初始化一次。
- <clinit>的执行，对于每一个静态成员变量存取，都被重新定向到了具体的租户存储空间。比如对于out = null赋值，putstatic执行时实际上会找到当前的租户，然后把值存到该租户的空间去，getstatic有着类似的道理。

Java类库支持多租户语境

这部分主要通过改造类库实现，具体的功能包括：

- System.exit(code) 调用只会使当前租户退出，而不会令整个JVM退出。而租户申请的一些诸如File/Socket句柄之类系统资源，会随着租户的推出而被释放。
 - 租户A不可能通过类似如下

```
ThreadGroup group = Thread.currentThread().getThreadGroup();  
  
ThreadGroup parent = group.getParent();
```

枚举线程的办法获得租户B的线程。不同租户的线程分属于不同的线程组。

- Java属性值的隔离，比如同样的语句System.getProperty("name")对于不同的租户可能是不同的值。

资源管理隔离

这是Multi-tenant JVM很重要的功能。在Multi-tenant JVM上，Heap/CPU/Disk IO/Net IO这些资源的使用是受资源策略保护的，比如你可以去限制某个租户它的CPU最少可以使用20%，而在系统空闲时，最大可以用到100%。

Multi-tenant JVM通过[Token Bucket](#)来对IO(Disk/Net)和CPU资源进行管理。对于IO而言，Multi-tenant JVM截获IO有关的OS API调用，使得IO发生之前受制于我们预先规定的资源策略。我们举个网络IO的例子，例如Java程序从Socket的读取操作，JDK内部的实现通过JNI实际上会对应到系统的API调用

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

在recv调用发生之前，Multi-tenant JVM通过资源策略保证租户的IO使用带宽不会超过给它设定的限制。关于网络IO，我们这里有一个很好的[演示](#)：

用简单的-xlimit:netIO=6M参数限制运行在Multi-tenant JVM上的Ftp Server带宽上限读写各为6Mib/s。

关于对CPU管理，Multi-tenant JVM实现的基本的思路是，把租户线程所花费的CPU时间量化为Tokens，运行时每一个租户线程都会被周期性检查是否其当前CPU时间的使用超过了给它设定的限制。如果超过，当前线程会被挂起，直到满足限制为止。周期性检查的代码是由Multi-tenant JVM插入到租户线程里去的，对于用户程序而言完全是透明的。

Multi-tenant JVM对于Heap的管理建立在[Balanced GC Policy](#)基础之上。同一般的Java程序类似，你可以使用-Xms/-Xmx为租户程序设定最大/最小的堆内存值。Balanced GC Policy基于Region对Heap进行管理，每个租户程序根据-Xms/-Xmx的设定来为其分配Region，而租户对象的分配也必然只能发生在它自己拥有的区域内。

多租户JVM的用法与限制

IBM发布的Java 7 R1默认支持多租户JVM，在命令行上添加-xmt参数即可启用。由于多租户JVM对JVM的变更，JNI Native Libraries、JVMTI以及GUI programs在多租户状态下的使用是受限制的。Multi-tenant JVM并未实现对JNI的隔离，所以不同的租户应用不能装载依赖同样的JNI Native Lib，所有发生在JNI Native Lib里的IO，不会受限于该租户资源消费策略。同样的情况适用于CPU以及Memory。

Multi-tenant JVM目前没有实现对JVMTI Agent的改造用以支持我们前面所描述的静态数据的隔离，这可能会对用户如果想调试Java核心类库代码（不是用户代码）造成困扰。

关于GUI，Multi-tenant JVM没有实现底层对于UI程序消息队列的隔离，所以不支持在同一个Multi-tenant JVM运行大于1个的GUI程序。

还有一点，不要在非Daemon线程里写“暴力”的死循环代码，例如：

```
while(true)
{
try () {
    ...
} catch(Throwable t) {
{
}
}
```

最后需要注意的是，当开启IO资源控制时，尽量一次写出更多的字节，避免影响程序的IO性能。

总结

Multi-tenant JVM目前在应用启动时间和更小的内存占用开销方面已经被证实有效。根据目前的一些基准测试结果来看，对于简单的应用，相较于一般JVM，Multi-tenant JVM可以获得5~6倍的运行个数。后续计划发布的版本仍然会集中在提高启动时间、更小的内存开销这两个方面，也会陆续有一些性能的报告发布。

长远来看，Multi-tenant JVM会基于用户、IBM产品线以及技术社区等的反馈，做进一步的提高，以及解决一些目前所存在的局限性，比如对于JNI隔离的支持，JVMTi的多租户支持等等。

作者介绍

李三红，IBM资深软件工程师，Multi-tenant JVM项目技术负责人，目前供职于IBM Java技术中心，从事多租户Java虚拟机相关的研发工作。九年多的Java开发经验，2008年加入IBM，参与基于OSGi框架的安全方面的开发，2010年加入Java技术中心，参与IBM Java虚拟机 J9的开发。在Java技术领域拥有多项专利以及在developerWorks上发表十余篇文章。他的微博：[@sanhong_li](#) IBM Java技术中心微博：[@IBM_JTC](#)

李三红将在2014年10月16-18日的QCon上海大会上就本话题[进行分享](#)。

感谢[杨赛](#)对本文的审校。

查看原文：[高密度Java应用部署的一些实践](#)

AWS 解决方案&白皮书下载

白皮书推荐

《向AWS迁移或新建应用的最佳实践》，本文将重点强调创建新的云应用程序或将现有的应用程序移植到云端的概念、原则和最佳实践。您将了解云计算带来的一些业务与技术优势，以及目前已可利用的AWS服务。

《在AWS上保证应用安全的最佳实践》，在一个多租户环境中，云架构师常常就安全问题表示担忧。安全性问题应在云应用程序架构的每一层都能落实。物理安全问题通常是由您的服务提供商处理，这是使用云的一个额外优点。网络和应用程序层级的安全是您的责任，您应该执行最佳实践，以便使之尽可能地适用于您的业务。在本文中，您将了解一些特定的工具、功能和如何确保您的云计算应用程序在AWS环境中安全的指导方针。建议充分利用这些工具和功能，以便实现基本安全，然后再酌情使用标准的或合适的方法来执行附加的安全最佳实践。

更多白皮书

《我们如何应对风险？了解AWS安全概述》

《AWS与云计算：传统IT面临机遇和挑战》

《AWS的价格魅力：按需付费，无需签署长期合同》

《在AWS上保证应用安全的最佳实践》

《AWS云与自有IT基础设施的经济性比较》



豌豆荚质量总监分享：从自建机房到云计算的演进之路

作者 杨赛

以下内容整理自InfoQ中文站跟豌豆荚质量总监高磊的一次采访对话。

豌豆荚作为创新工场的首批孵化项目之一，从2009年12月发展至今，用户量已经增长至4.1亿。豌豆荚的主要业务在国内，帮助用户在手机上发现、获取和消费应用、游戏、视频、电子书、壁纸等娱乐内容，在东南亚地区等海外市场也做了类似的业务探索。

这样一个快速增长的系统，对IT的底层支持也是一个相当大的挑战。本文将介绍豌豆荚在IT基础架构、工具、流程方面做过的一些事，在不同需求之间如何平衡，团队职责的划分，以及遇到的一些挑战。

受访者简介

高磊，2012年4月加入豌豆荚，现任豌豆荚质量总监，负责豌豆荚的工程生产力部门（Engineering Productivity, EP）。长期关注[QA这个话题](#)，对流程、开发技术、自动化测试、敏捷、持续集成、运维、代码库、生产力工具等方向均有所涉猎。

基础设施的建设与增长

豌豆荚诞生于2009年12月，机房部署是从2010年年初开始。那时候因为还没有成熟的云服务可用，所以选择了自建机房的方案。到目前为止，我们已经在全国各地尤其是北京、天津地区建立了多个节点。

从对基础设施资源使用的情况来看，我们的主要业务对带宽和CDN资源用量会比较高；而从单一业务来看，各类数据挖掘和分析对服务器资源的占用是最大的。豌豆荚从创建一开始就是数据驱动的业务，有很强的用户行为导向，因此数据挖掘的工作量非常多。

数据挖掘主要是基于Hadoop集群。豌豆荚有一个数据挖掘团队专门做产品研发（主要是面向内部），而我们这个团队则提供硬件资源和底层的Hive、HBase等基础设施的支撑和维护。整体的数据量、计算量一直都在增长，一开始的几年增长极快，最近几年稍微慢一些，也有每年几倍的增长。

差不多在2011年左右，我们开始尝试做海外版的豌豆荚Snappa。当时评估过在海外自建机房的可行性，在考察过各个地方不同位置、不同IDC、不同运营商的选项之后，我们发

现即使在进展顺利的情况下，也至少需要两三个月才能建成，这个时间成本太高。如果不自建，那就只有公有云这一个选择，正好当时我们很多工程师都自己用过亚马逊的AWS，出于时间、知识门槛、成本的考量，就决定在海外使用AWS作为我们的基础支撑。

团队

EP团队的目标很明确：在主要产品的完整生命周期内，实现一流的效率、质量和服务稳定性；至于具体用什么技术或者方法，则并不做限制。一开始我们团队比较关注流程、开发工具等方面，现在我们对CI、代码库、自动化测试、运维、基础设施建设等各个方面都做了很多工作，有时候工程师要引入一些新的基础设施相关的技术或框架，我们也会进行review它们是不是靠谱，总的目标就是让产品从开始开发到线上生产环境运行这整条路径下，其稳定性和质量都有所保证。

现在整个团队的全职工程师有不到三十人，其中运维团队有十个人，而且他们也都会承担开发任务（我们叫做SRE，网站可靠性工程师），运维过程中需要什么工具、支持系统，都是由他们自己开发。运维团队的主要工作都在维护我们自建的机房系统上，AWS上面现在平均投入的维护人力差不多只有三分之一个人。这一方面是因为AWS的维护成本确实低，另一方面也是因为我们在AWS上面的规模还不是太大。

从代码库到生产环境

我们的产品发布流程还是相对成形的。不同的产品线有不同的发布频率，比较稳定的在一周两次，有些比较早期的项目可能一天一次，没有太大的压力。

产品下一个release要发布哪些feature、发布周期设置成多久间隔，主要是由产品经理和设计师来决定，工程师实现需求。到了发布日期截止之前，从代码库的主干拉一支发布分支出来做feature freeze和最后的验收测试，到发布分支上只能做bug修复，不再接受新的feature。

有的产品线有统一的测试机制，有的产品线则主要靠工程师自己做测试。无论是哪种测试模式，在进入CI做集成之前和之后都会统一进行静态检查和已有的单元测试用例，然后才上到staging环境。从staging环境开始就属于运维负责的领域了，我们的staging没有真实的流量，但是环境跟线上是一模一样的，可以说是一直处在最新版本的服务，然后staging再跟线上环境同步代码。

这一套自动发布、部署的流程虽然也不是很完善，比如持续集成的检查点还不够多，单元测试率还比较低，不过还算跑的不错。现在AWS上也是同样的一套部署过程，当时适配起来也很快，大概做了一个星期就跑上去了。

监控

我们的监控系统要实现的目的无非是两个：实时的报警，以及可以追溯的历史数据，其他都是衍生的功能。跟大部分互联网公司一样，我们一开始做监控也都是用开源软件搭起来的，不过开源的监控软件现在越来越不能满足我们的需求。

这里面有两个挑战：

1. 性能问题
2. 数据采集的定制化问题

数据采集的定制化主要是涉及到一些业务数据的采集，通用的开源软件也还是要做适配，需要自己去写实现，这个其实还好。性能问题是一个更加严重的问题，这个问题来自于三个方面：越来越多的机器、越来越多的采集项、越来越高的采集频率。

以前我们监控，可能5分钟抓一次数据就行；现在我们希望做到秒级的采集。监控系统需要有实时分析日志的能力，而到机器数量增长到千台以上之后，要做秒级的采集和分析，无论是数据收集的速度还是数据分析的速度都会遇到瓶颈。

所以现在我们正在自己重写一套监控的系统，专门针对我们自建的机房体系，包括对多机房架构的支持、与资产系统的对接等等。而AWS上我们直接使用了其CloudWatch监控功能，目前来讲完全够用了。

一些挑战

因为业务跟数据密切相关，而我们部门承担了给数据分析团队提供基础设施的责任。

业务对数据报告的需求一般有两类：

1、定制化的、定期的数据指标报告

此类报告有按天的、按周的、按月的或者按小时的，一般都是比较常规的监测指标，持续监控、持续分析，中间数据保留完整，需要的计算量和存储量容易预测。这种报告需求比较容易满足。

2、按需出报告

此类需求经常是针对之前没有中间数据的监测值，之前并不知道需要针对此类数值做分析，现在忽然发现需要了，业务部门会要求一次性分析过去半年到一年跟该数据相关的趋势。此类报告往往很耗时，有时候我们做估算，一年的数据分析完毕需要用多长时间，结果可能是用我们现在的计算资源，可能要分析一个月才能产出他想要的报告，但这是无法满足业务需求的。

要提高分析速度，最直接的做法就是投入更多的计算资源——放在我们自建的机房就是扩容，如果用公有云就是起更多的实例。一方面扩容是要做的，另一方面现在AWS进入国内，我们也在考察使用AWS来做这种任务的可能性。

实际上我们用了AWS以来，也逐渐发现我们之前系统设计的并不是那么好。比如我们在海外的数据分析，原本是想用EMR的，但是发现我们现在这套东西搬过去不好直接用，只好基于EC2来做这个事情。为什么呢？因为AWS的理念是让不同的组件做不同的事，比如EC2只做计算，数据持久化存储最好都放在S3；但是我们的系统一开始设计并没有考虑这些，数据都存在本地计算节点上，如果要重构，需要投入的时间还是挺多的。

包括scaling这件事也是，现在我们基本没有用到scaling，因为我们的应用上下游之间的依赖关系太重，所以对scaling机制支持的不好。这些都是需要努力的方向。一个比较好的事情是，我们豌豆荚的工程师都是比较有情怀的，对重构这件事情比较支持。当然，这里面有投入成本和产出的考量，我们首先要满足的还是业务需求，解决业务问题，至于重构的工作，会随着我们在AWS上的业务规模更大而变得优先级更高。

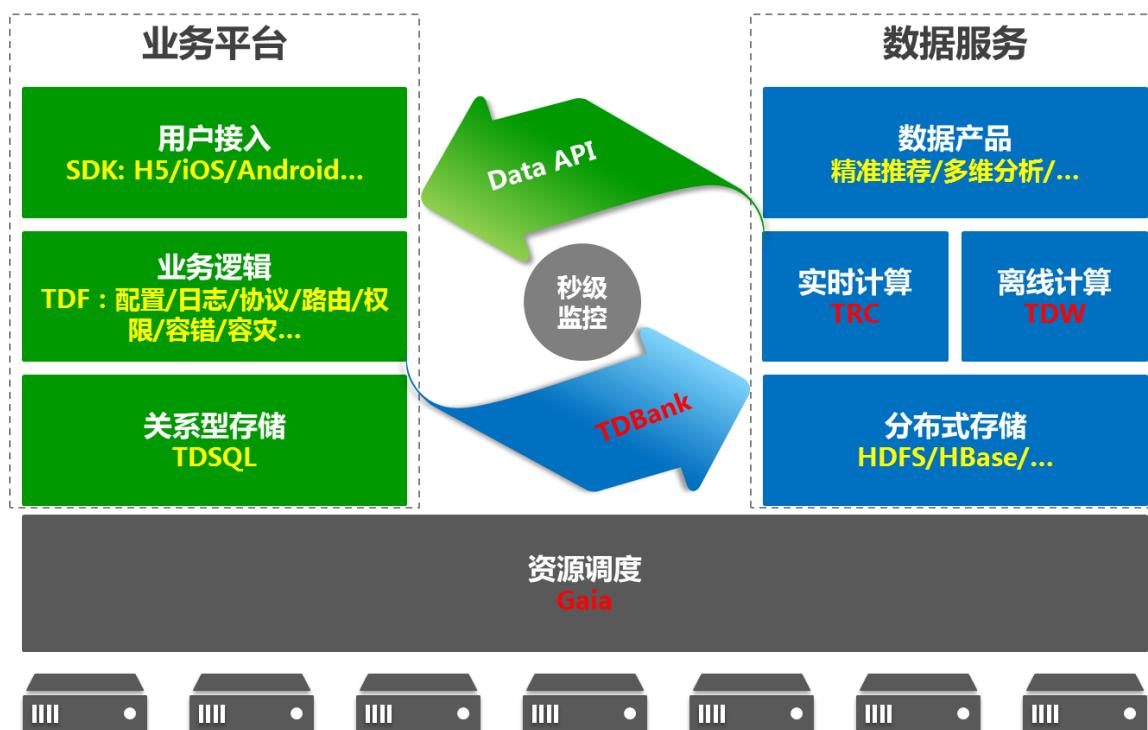
最后想分享的是，EC2的reserved instance如果用好了，能够比on demand模式节省很多。我们一开始不知道AWS除了on demand之外还有reserved instance、spot instance这些玩法，最近才刚知道。Reserved instance非常适合Web Service使用，临时性数据分析用spot instance则比较合适。

查看原文：[豌豆荚质量总监分享：从自建机房到云计算的演进之路](#)

腾讯大数据平台纵览

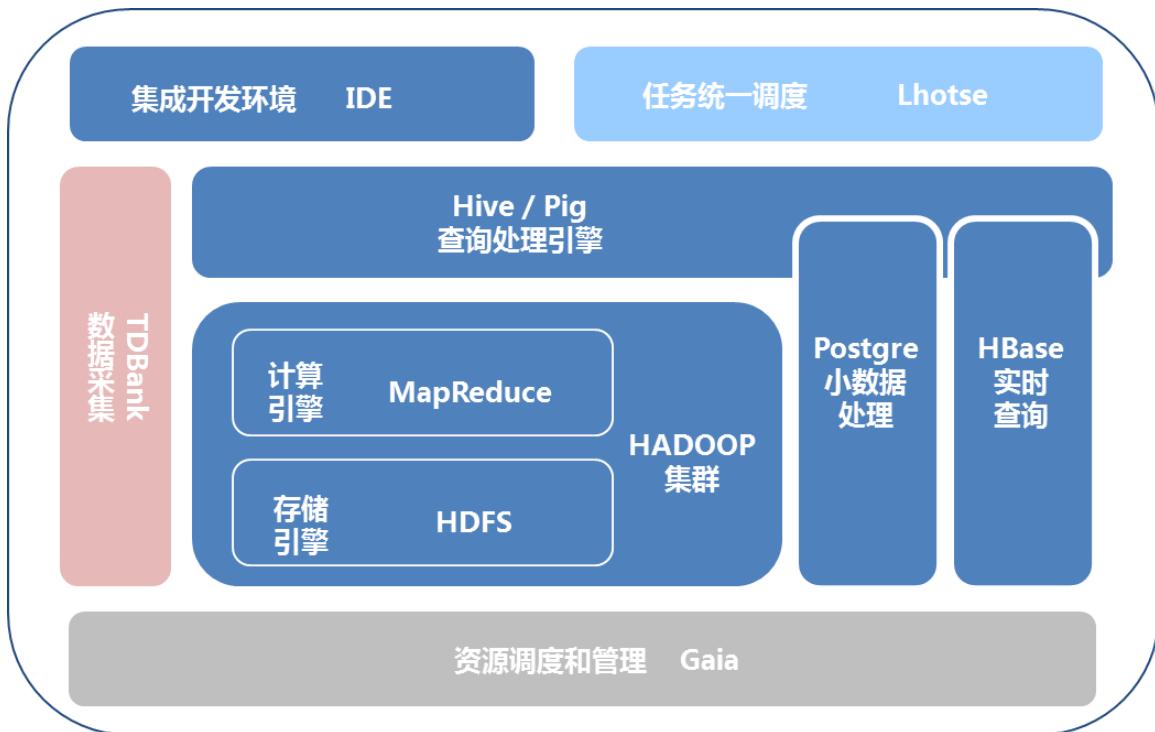
作者 刘煜宏

腾讯业务产品线众多，拥有海量的活跃用户，每天线上产生的数据超乎想象，必然会成为数据大户。特别是随着传统业务增长放缓，以及移动互联网时代的精细化运营，对于大数据分析和挖掘的重视程度高于以往任何时候，如何从大数据中获取高价值，已经成为大家关心的焦点问题。在这样的大背景下，为了公司各业务产品能够使用更丰富优质的数据服务，近年腾讯大数据平台得到迅猛发展。



从上图可以看出，腾讯大数据平台有如下核心模块：TDW、TRC、TDBank和Gaia。简单来说，TDW用来做批量的离线计算，TRC负责做流式的实时计算，TDBank则作为统一的数据采集入口，而底层的Gaia则负责整个集群的资源调度和管理。接下来，本文会针对这四块内容进行整体介绍。

TDW (Tencent distributed Data Warehouse)：腾讯分布式数据仓库。它支持百PB级数据的离线存储和计算，为业务提供海量、高效、稳定的大数据平台支撑和决策支持。目前，TDW集群总设备8400台，单集群最大规模5600台，总存储数据超过100PB，日均计算量超过5PB，日均Job数达到100万个。

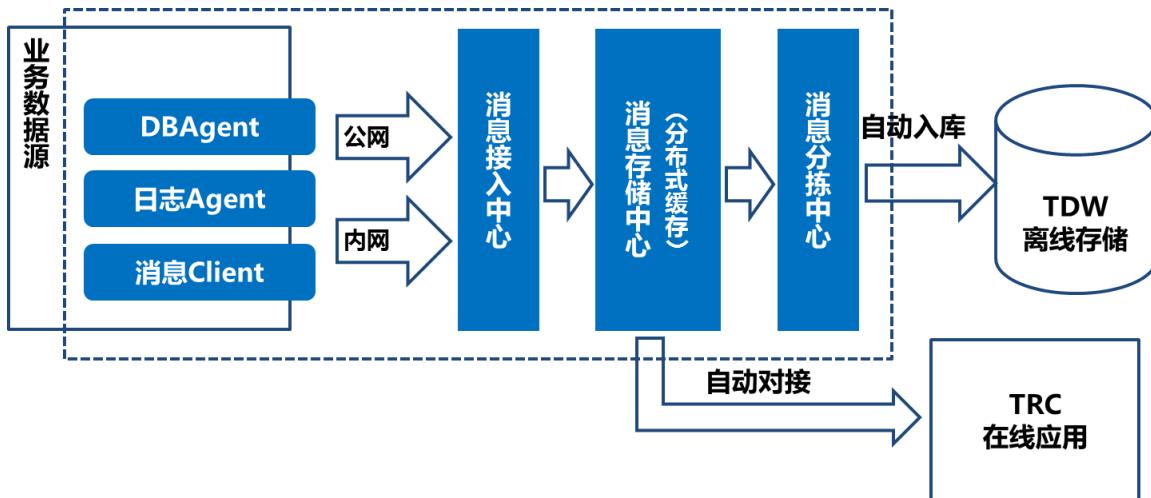


为了降低用户从传统商业数据库迁移门槛，TDW基于开源Hive进行了大量定制开发。在功能扩充方面，SQL语法兼容Oracle，实现了基于角色的权限管理、分区功能、窗口函数、多维分析功能、公用表表达式-CTE、DML-update/delete、入库数据校验等。在易用性方面，增加了基于Python的过程语言接口，以及命令行工具PLClient，并提供可视化的IDE集成开发环境，使得开发效率大幅度提升。另外，在性能优化方面也做了大量工作，包括Hash Join、按行split、Order by limit优化、查询计划并行优化等，特别是针对Hive元数据的重构，去掉了低效的JDO层，并实现元数据集群化，使系统扩展性提升明显。

为了尽可能促进数据共享和提升计算资源利用率，实施构建高效稳定的大集群战略，TDW针对Hadoop原有架构进行了深度改造。首先，通过JobTracker/NameNode分散化和容灾，解决了Master单点问题，使得集群的可扩展性和稳定性得到大幅度提升。其次，优化公平资源调度策略，以支撑上千并发job（现网3k+）同时运行，并且归属不同业务的任务之间不会互相影响。同时，根据数据使用频率实施差异化压缩策略，比如热数据lzo、温数据gz、冷数据gz+hdfs raid，总压缩率相对文本可以达到10-20倍。

另外，为了弥补Hadoop天然在update/delete操作上的不足，TDW引入PostgreSQL作为辅助，适用于较小数据集的高效分析。当前，TDW正在向着实时化发展，通过引入HBase提供了千亿级实时查询服务，并开始投入Spark研发为大数据分析加速。

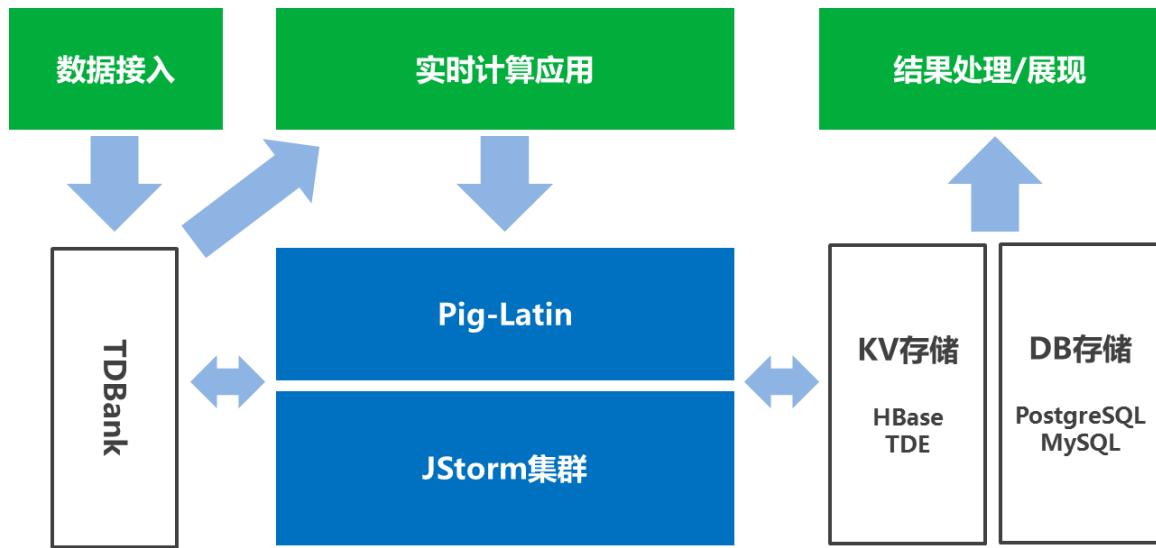
TDBank (Tencent Data Bank)：数据实时收集与分发平台。构建数据源和数据处理系统间的桥梁，将数据处理系统同数据源解耦，为离线计算TDW和在线计算TRC平台提供数据支持。



从架构上来看，TBank可以划分为前端采集、消息接入、消息存储和消息分拣等模块。前端模块主要针对各种数据形式（普通文件，DB增量/全量，Socket消息，共享内存等）提供实时采集组件，提供了主动且实时的数据获取方式。中间模块则是具备日接入量万亿级的基于“发布-订阅”模型的分布式消息中间件，它起到了很好的缓存和缓冲作用，避免了因后端系统繁忙或故障从而导致的处理阻塞或消息丢失。针对不同应用场景，TDBank提供数据的主动订阅模式，以及不同的数据分发支持（分发到TDW数据仓库，文件，DB，HBase，Socket等）。整个数据通路透明化，只需简单配置，即可实现一点接入，整个大数据平台可用。

另外，为了减少大量数据进行跨城网络传输，TDBank在数据传输的过程中进行数据压缩，并提供公网/内网自动识别模式，极大的降低了专线带宽成本。为了保障数据的完整性，TDBank提供定制化的失败重发和滤重机制，保障在复杂网络情况下数据的高可用。TDBank基于流式的数据处理过程，保障了数据的实时性，为TRC实时计算平台提供实时的数据支持。目前，TDBank实时采集的数据超过150+TB/日（约5000+亿条/日），这个数字一直在持续增长中，预计年底将超过2万亿条/日。

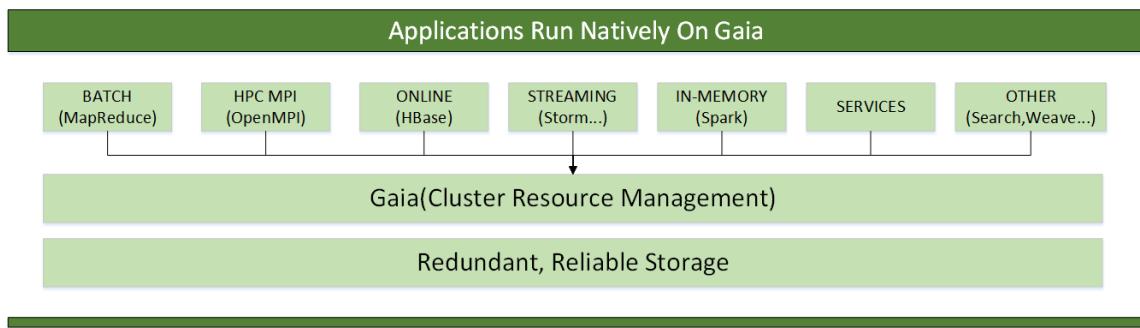
TRC (Tencent Real-time Computing)：腾讯实时计算平台。作为海量数据处理的另一利器，专门为对时间延敏感的业务提供海量数据实时处理服务。通过海量数据的实时采集、实时计算，实时感知外界变化，从事件发生、到感知变化、到输出计算结果，整个过程中秒级完成。



TRC是基于开源的Storm深度定制的流式处理引擎，用Java重写了Storm的核心代码。为了解决了资源利用率和集群规模的问题，重构了底层调度模块，实现了任务级别的权限管理、资源分配、资源隔离，通过和Gaia这样的资源管理框架相结合，做到了根据线上业务实际利用资源的状况，动态扩容&缩容，单集群轻松超过1000台规模。为了提高平台的易用性和可运维性，提供了类SQL和Pig Latin这样的过程化语言扩展，方便用户提交业务，提升接入效率，同时提供系统级的指标度量，支持用户代码对其进行扩展，实时监控整个系统运营环节。另外将TRC的功能服务化，通过REST API提供PaaS级别的开放，用户无需了解底层实现细节就能方便的申请权限，资源和提交任务。

目前，TRC日计算次数超过2万亿次，在腾讯已经有很多业务正在使用TRC提供的实时数据处理服务。比如，对于广点通广告推荐而言，用户在互联网上的行为能实时的影响其广告推送内容，在用户下一次刷新页面时，就提供给用户精准的广告；对于在线视频，新闻而言，用户的每一次收藏、点击、浏览行为，都能被快速的归入他的个人模型中，立刻修正视频和新闻推荐。

Gaia：统一资源调度平台。Gaia，希腊神话中的大地之神，是众神之母，取名寓意各种业务类型和计算框架都能植根于“大地”之上。它能够让应用开发者像使用一台超级计算机一样使用整个集群，极大地简化了开发者的资源管理逻辑。Gaia提供高并发任务调度和资源管理，实现集群资源共享，具有很高的可伸缩性和可靠性，它不仅支持MR等离线业务，还可以支持实时计算，甚至在线service业务。



为了支撑单集群8800台甚至更大规模，Gaia基于开源社区Yarn之上自研Sfair (Scalable fair scheduler)调度器，优化调度逻辑，提供更好的可扩展性，并进一步增强调度的公平性，提升可定制化，将调度吞吐提升10倍以上。为了满足上层多样化的计算框架稳定运行，Gaia除了CPU、Mem的资源管理之外，新增了Network IO，Disk space，Disk IO等资源管理维度，提高了隔离性，为业务提供了更好的资源保证和隔离。同时，Gaia开发了自己的内核版本，调整和优化CPU、Mem资源管理策略，在兼容线程监控的前提下，利用cgroups，实现了hardlimit+softlimit结合的方式，充分利用整机资源，将container oom kill机率大幅降低。另外，丰富的API也为业务提供了更便捷的容灾、扩容、缩容、升级等方式。

基于以上几大基础平台的组合联动，可以打造出了很多的数据产品及服务，如上面提到的精准推荐就是其中之一，另外还有诸如实时多维分析、秒级监控、腾讯分析、信鸽等等。除了一些相对成熟的平台之外，我们还在进行不断的尝试，针对新的需求进行更合理的技术探索，如更快速的交互式分析、针对复杂关系链的图式计算。此外，腾讯大数据平台的各种能力及服务，还将通过TOD (Tencent Open Data) 产品开放给外部第三方开发者。

作者简介

刘煜宏 (ehomeliu)：拥有10年以上的电信行业及互联网行业的从业经验，现就职于腾讯数据平台部，是腾讯实时数据接入平台（TDBank）及实时计算平台（TRC）的负责人，在大数据接入、计算及分析等方面有丰富经验。

感谢包研对本文的审校。

查看原文：[腾讯大数据平台纵览](#)

有关云架构建设和选型的思考

作者 罗立树

最近在负责公司内部私有云的建设，一直在思考怎么搞云计算，怎么才能够把云架构设计得好一些。本文尽量全面的列出了云架构建设和选型的考量因素。

我们主要从五个层面逐步评估云架构的建设和选型，分别是：

1. 行业生态
2. 企业需求
3. 云计算的能力
4. 潜在的挑战
5. 如何建设

一、行业生态

计算机云经过多年的发展，由一开始的概念，慢慢发展成熟并能够推向市场，提供多种多样的服务，市场空间非常之大。

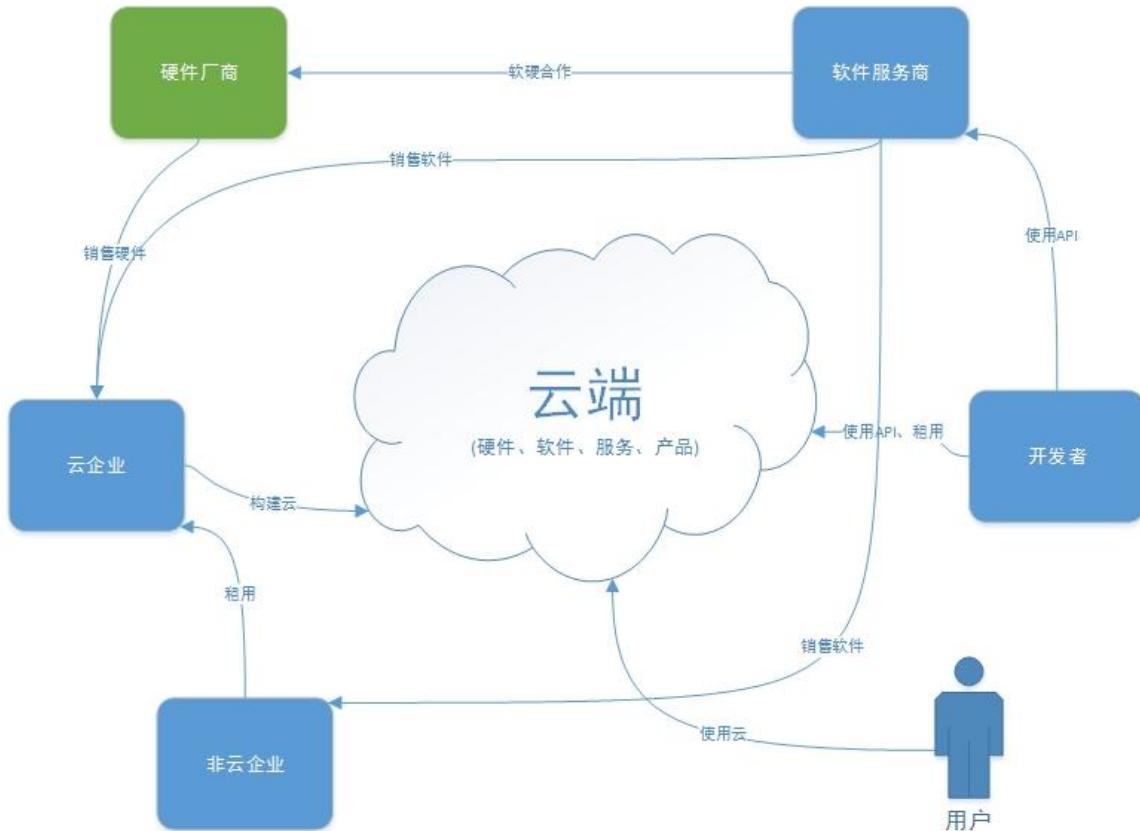
在云的发展过程中，亚马逊经过多年的深耕积累，发展成为了云行业的标杆企业，甚至可以说是建立了云解决方案的标准。之后，Google、IBM、思科、Oracle、HP、Intel、华为等IT巨头先后参与进来，在软件和硬件方面提供专门的面向企业的解决方案，纷纷打着云计算、大数据、智能等概念来吸引客户，拓展市场。

另外一方面，基于大数据、存储、云服务等，市场上也先后出现一些创新企业，如Dropbox、Rackspace，国内的七牛、青云、UnitedStack等。

当前的IT世界有一个常见的现象，就是只要某一个领域有一套成熟的商业软件，就同时也会有一套开源的解决方案，如Windows之于Linux，Google的MapReduce、GFS、大表之于Hadoop等。在云领域也存在相应的开源解决方案，目前最为著名的有Openstack和Cloudstack。开源行业的领导者RedHat此前

在企业操作系统的市场已经做的很好，RHEL的各个版本在企业级系统市场有相当高的市场份额。现在的RedHat特别重视云的发展，并将云操作系统作为未来10年的发展战略重点，在最近两年先后收购了Gluster以及Ceph等存储企业，以壮大自己在云领域的影响力。

随着云领域的发展，市场上也逐渐形成了面向企业提供硬件和软件产品的提供商、面向企业提供服务的提供商、面向市场初创企业提供基础服务的提供商、面向个人提供业务服务的提供商等一系列行业生态。



二、企业需求

需求是什么，也就是what people need这个问题。我们所说的people，即人或者公司实体，该对象的划分并不单纯，可粗浅的从三个角度来进行分类：

从企业角度看：

1) 小型企业

小型企业的技术储备不多，人员缺乏，没有独立的IT部门，但是在构建自己的IT系统过程中需要购置各种产品和服务，包括服务器、网络、CDN等等，而要完成这样的工作，需要投入大量的人力和财力。通过购买云服务可以更加方便快捷，简单的完成系统的搭建。

2) 中型企业

中型企业有一定的规模，需要在信息化、管理方面有所注重，一般内部都设立IT部门，但是和小型企业一样，IT部门大多数都是为了解决自身需求，很难能够有一个完整的解决方案。这样在服务器、网络、CDN、企业管理软件等等的需求还是比较大的。

3) 大型企业

大型企业人数规模在万人以上，特别是高新企业，都有一个实力不错的IT支撑部门，通过部门就可以完善对企业内部信息化建设。

从企业性质范围来看：

1) 传统行业企业

传统行业大多数是以服务业、制造业、生产性企业为主，在IT信息化方面相对比较落后，属于重资产行业。

2) 互联网企业

互联网行业是基于IT作为解决方案的

3) IT服务企业

以销售软件、硬件、以及技术咨询服务为主的企业。

针对市场中存在的企业、个体等的需求特点，市场上一般对软件服务进行如下分类：

1. 提供软件的服务，解决企业内部信息化问题，如ERP系统、进销存管理系统、人力资源管理系统、行政系统、财务系统等等。（SaaS）
2. 提供平台服务，解决行业共性问题，将SaaS迁移到云端，提供平台类的服务。如淘宝的开放平台、Facebook的开放平台、基于Salesforce的销售系统、云笔记、云盘等。（PaaS）
3. 提供基础设施服务。基础设施包括软件和硬件方面的，包括存储、虚拟机、网络、防火墙、缓存、负载均衡、数据库等等。（IaaS）

从企业内部人员角度来看：

企业内部，尤其是互联网企业内部，一般将角色分为如下几类：

1. 开发

2. 测试
3. 运维
4. DBA
5. 产品
6. 项目管理人员
7. 客服
8. 业务人员（销售、市场、BD、人力资源、行政等等）

不同的角色对于软件服务的需求也是不同的，下图大致描绘了互联网行业各个角色对云平台的需求：



三、云计算的能力

云计算能够解决什么，也就是what cloud offer这个问题。目前的云计算在应用中主要提供了以下八个能力：

1. 封装：将计算能力和软件放在云端，可以减少重复建设，将通用的服务封装起来，达到重用，减少资源的浪费，提高生产效率，并提供成熟的解决方案。在云端，云

提供商可以建立软件的标准，提供发布包的方式，用户可以通过软件包的方式进行购买使用，譬如目前开源领域的Docker。

2. 安全：云计算将数据和存储，软件逻辑都集中于云端，更能方便的统一构建安全体系，通过Iptables实现网络过滤，并在服务端做安全组件实现安全策略，并能够通过海量集群应对DDOS攻击等。
3. 灵活：云计算提供灵活的软件和服务端架构，用户不再需要自己构建应用运行环境，对资源的使用能够按需购买，并能够升级，并自由组合。举例来说：用户可以选择不同的存储方式（mysql、oracle，文件系统，kv等等）
4. 性能：通过集群的能力和云端的集成能够提高集群的性能处理，通过专业的云解决提供商，在云端的性能扩展更加方便，技术上更加专业。譬如服务端可以在用户毫不察觉的情况下完成添加机器、存储扩容等操作。
5. 伸缩能力：在存储和计算能力方面提供弹性的资源管理，能够按需使用，在使用过程中，可以通过动态的添加和减少物理资源，来提高响应能力或节约成本。
6. 运维：云计算在IaaS角度来看，重要的是运维，能够将运维更加集中化管理，并完全智能化，大大降低人力成本
7. 充分利用物理资源：通过云建设，能够将物理资源进行虚拟化处理，屏蔽物理硬件底层，并能够完成物理资源软化进行逻辑管理和分配调度
8. 大数据：大数据保存于云端，能够提供数据分析和智能处理

当然，云计算还有很多很多好处，给我们带来很多想像空间和IT技术的革命。

公有云与私有云

行业内将云分为“公有云”和“私有云”。在我们之前的需求分析过程中，大致了解了云的需求，“公有云”和“私有云”的差别最大的是需求的差异，因为需求的差异，导致了技术方案和产品决策的差异。

公有云需求上由于用户多种多样，导致需求存在不一样，特别需要更多的定制化，譬如：

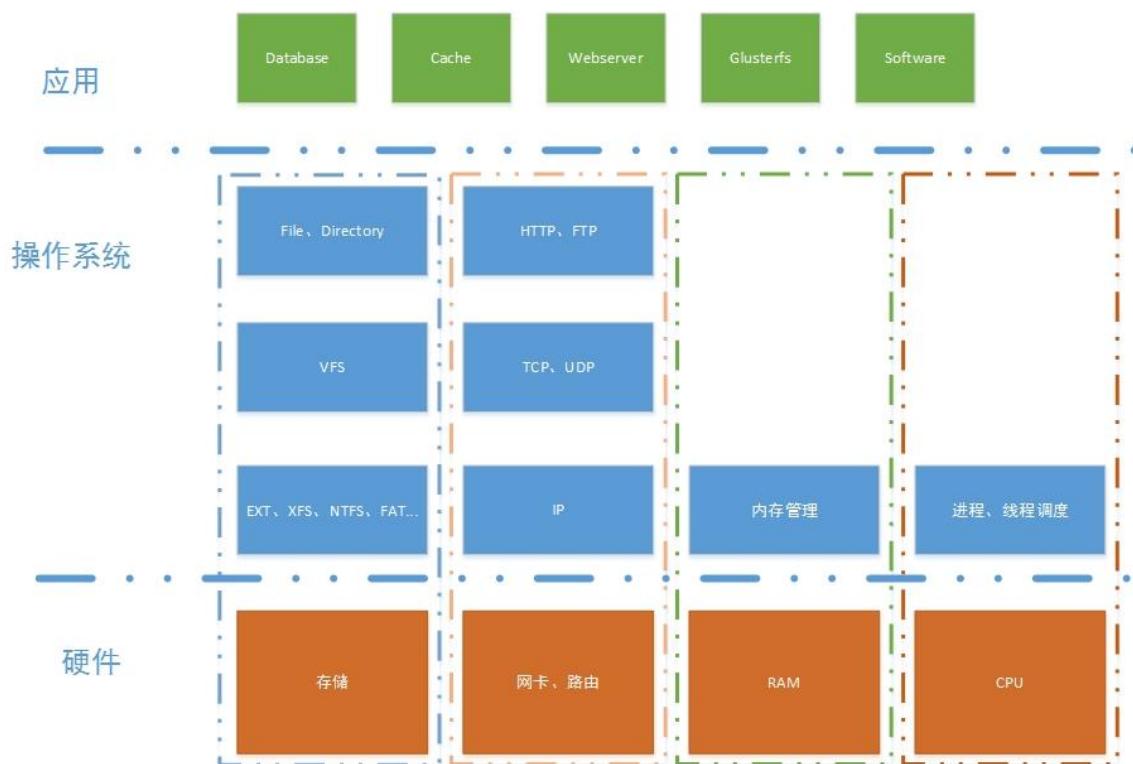
1. 存储个性化 云存储方面大概分为块存储和对象存储，块存储适合于vm运行环境，对象存储提供了KV的访问方式提供了海量扩展存储文件的能力，用户可以根据自己的需求选择不同的存储方式，选用不同的容量。在存储物理介质方面来说，因为在不同的物理介质，对性能和安全的要求，可以采用传统的SATA硬盘，或者SSD存储等。
2. 内存使用 内存方面，需要提供动态扩展内存的方式，用户能够自由扩展

3. 网络的定制化 公有云用户需要能够构建自己的内部网络，并能够自动组网
4. 数据库使用 公有云的用户分属不同的公司团体，各自的技术差异存在，因而有不同的数据库类型，譬如mysql, sqlserver, oracle等等。并能够定义存储大小，内存运行大小等等。并提供数据备份、恢复、高可用服务等
5. 缓存使用 公有云的用户可以选择不同的缓存方式，譬如增加CDN，采用不同的KV缓存方式并选择容量。
6. 安全问题 公有云对于云的安全和私有云差别较大，私有云大多数在安全问题上不需要公有云那么严格，大多数是内部系统之间的交互

以上仅限于IaaS层面的考虑，当然对于公有云来说还有很多细化的个性化需求，例如：数据分析，业务对接服务等等。

四、潜在的挑战

计算机自从诞生以来，一直按照冯·诺伊曼的体系发展在硬件的基础上的操作系统，也分为网络协议体系的实现、内存管理、文件管理体系等等。大致的抽象图如下：



要建设云，有几个重要的问题需要解决：

1. 管理问题 云计算的实施首先要解决运维的问题，在云环境下后端是大规模数量的物理节点的集群，对于同时维护数以千计算的计算节点，以及部署结构的复杂，需求的变化，光靠增加人力也难以解决复杂的问题。从而需要构建高效的计算资源管理系统，能够灵活简单的管理运系统，并能够及时的发现问题。
2. 计费问题(公有云) 对于公有云而言，因为是面向公众的，必然产生费用的问题，常用的收费方式多种多样，也因为产品的不同而计费方式不同，譬如：网络、存储、cpu、数据库容量等等
3. 资源隔离问题 云计算运行在云端，是通过虚拟化体系建立的，虚拟化是建立在硬件之上，多个虚拟化资源同时运行于同一节点(host)中，存在着资源的共享争用问题，这样就存在着资源使用的公平性问题，导致同一Host上的资源使用相互影响。为了使得彼此资源使用相互独立，我们要建立相应的隔离机制。资源的隔离包括：存储、内存、cpu、数据库、网络等，其中网络是最难控制的。
4. 安全问题 在云端的应用和基于客户端的安全，面临的环境不一样，客户端方面大多数是病毒问题引起的，而在云端，也存在一些服务器攻击的问题，以及数据相互独立相互影响的问题，以及一些服务端编程的安全问题等。
5. 性能问题 对于云来说，需要保证云端的性能问题，包括CPU处理性能，IO处理能力，资源的就近访问，资源数据同步的速度，还需要解决系统底层的性能问题，包括文件处理Cache，存储介质的优化，采用SSD等，或者采用SATA+SSD的混合方式节约资源和降低成本。
6. 存储问题 对于云来说，由于云端是将客户端的数据和运算转移到云端，必须要有足够的存储能力以及足够稳定的存储系统，保证用户数据的安全，对于存储来说，有提供VM虚拟机运行环境的block device（块存储），以及提供KV方式的对象访问存储，这些都需要保证数据复制、数据读写访问的性能和数据永久可用的能力
7. 网络问题 对于公有云以及私有云的一些应用场景，需要能够提供网络的逻辑隔离（SDN）或物理隔离，以及对网络的访问灵活问题。构建虚拟化网络，由于物理条件的限制，我们不得不从L2-L4层进行处理，我们常用的方式是：bridge，vlan，gre，sdn（openflow，.opendaylight），以及一些厂家的产品等等。
8. 高可用问题 高可用问题是在分布式系统中必须要处理的问题，正因为集群的问题，我们必须从多方面考虑解决的问题，包括保证云管理系统的高可用性，存储介质的高可用性，网络的高可用性，虚拟机高可用问题等等。
9. 提高资源利用率问题 对于物理资源的虚拟化，我们有很多种解决方法，KVM、Vmware、xen、Hyperv、LXC等等，在HVM的方式下，对于VM本身的启动需要占用大量的内存、cpu和存储资源，导致系统内存和cpu使用有一定的浪费，基于LXC的解决方案因为是机基于Host OS进程，通过namespace的方式进行隔离的，是一种轻量级的实现，能够在资源初始化，资源利用率方面能够最大化，对于各个应用场景来说，我们可以选用合适的解决方案。

五、如何建设

58同城经过多年的发展，探索了一条适合自身发展的技术架构体系。随着业务和技术的发展，团队规模不断壮大，在技术和管理上面临越来越多的挑战。在项目需求管理，开发效率、代码管理和质量建设，测试，线上发布，运维管理等方面需要有一套完整的解决方案，来提升公司的协作能力和整体能效。

58同城目前所有的应用在线上都是跑在物理机器上，采用物理机的方式，一方面会导致服务器资源得不到充分和合理的使用，譬如：有些物理机器cpu使用长期在10%以下，有些内存使用剩余很多；另外一方面，由于互联网的特点，存在着时段内的访问高峰问题，需要解决资源使用的伸缩问题；基于以上问题，架构部对现有的技术体系进行梳理和分析，采用资源虚拟化的方式进行私有云的建设，并在这基础上，完善公司整体技术体系，包括：开发、测试、上线、运维等一系列自动化和智能化方面的建设。

私有云的目标

1. 提高物理资源的利用率
2. 一套云管理系统，降低运维的复杂度，提高运维工作效率
3. 构建灵活的开发、测试集成环境
4. 提供海量的存储体系
5. 建立完善的监控体系
6. 建立基础应用环境、方便测试
7. 统一架构
8. 智能资源调度

实施方案：OpenStack

对于云计算来说，也存在着多种解决方案，如CloudStack和OpenStack等。在两种方案的比较之后，我们最终选择了OpenStack的解决方案。主要是出于以下几点原因：

1. OpenStack的社区成熟度：OpenStack经过几年的发展，社区已经越来越成熟，很多大公司都参与进来帮助完善，红帽公司未来十年也将OpenStack作为发展的战略重点。
2. 架构设计的选择：OpenStack采用了Python语言编写，并且设计上采用组件化的方式，各个组件独立发展，并相互解耦

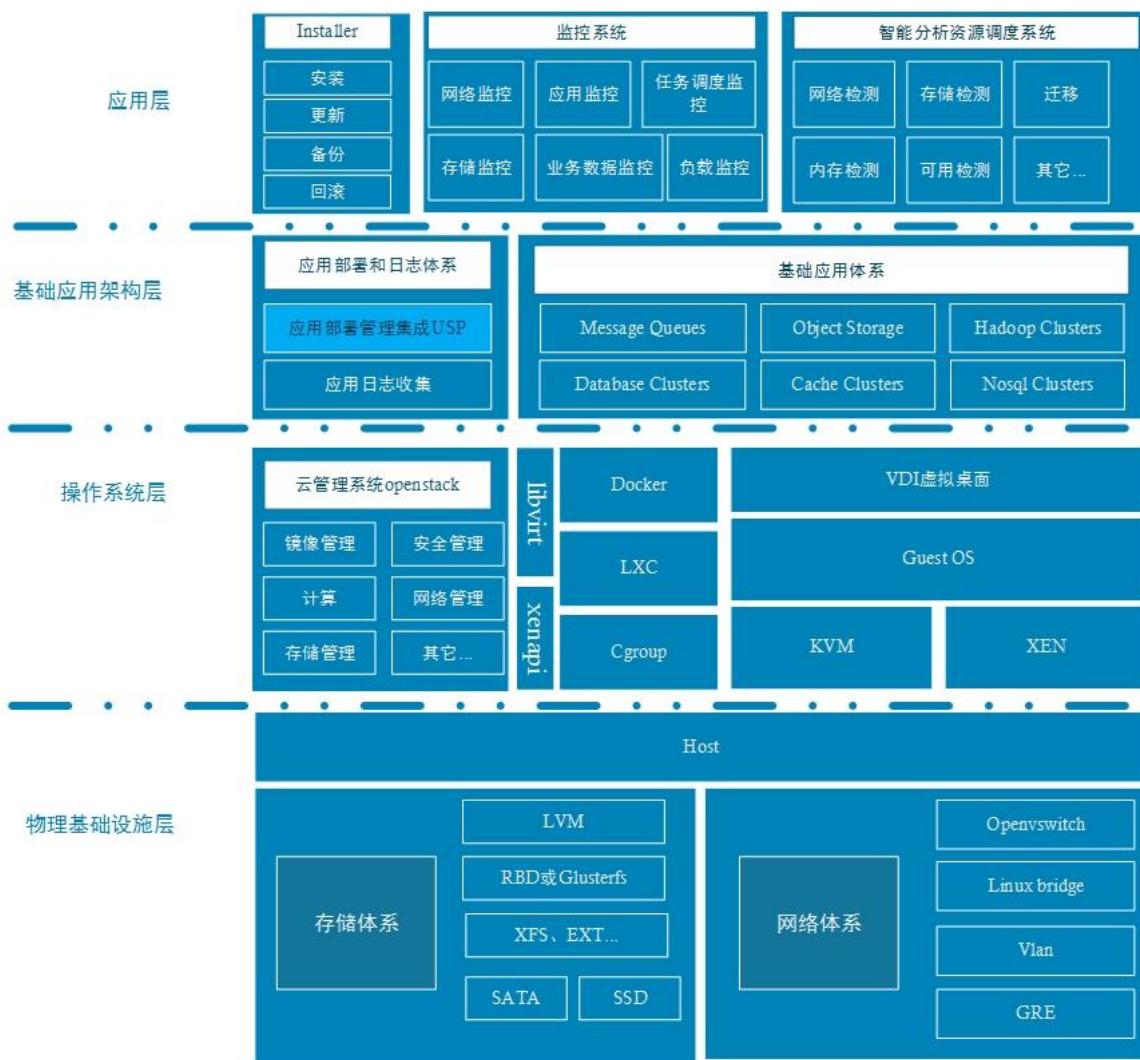
3. OpenStack提供了更加完整成熟的方案，能够满足多样的需求，同时已经有不少公司采用，已经经过生产上的验证

4. 文档问题：OpenStack文档化做的不错，网上能够找到多种多样的问题处理办法

5. 人员招聘问题，经过多年的发展和市场的培育，了解OpenStack的人越来越多，对于开发维护的人才建设和招聘相对成熟一些。

6. 发展比较迅速

下图是我们大致的架构规划



文章观点仅一家之言，欢迎大家一起交流探讨。我计划在下一篇文章《58同城私有云建设实践》中详细介绍我们私有云建设的思路和过程，中间遇到的问题，希望跟大家一起探讨。

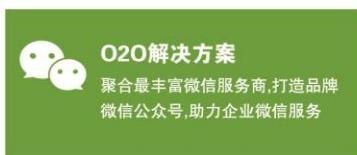
感谢杨赛对本文的审校。

查看原文：[有关云架构建设和选型的思考](#)



腾讯云简介

腾讯云致力于打造最高质量、最佳生态的公有云服务平台。基于QQ、微信、QQ空间、腾讯游戏等海量业务的技术架构和精细化互联网运营经验，腾讯云为广大企业和开发者提供云计算、云数据、云运营等一体化云端服务能力，助力企业建立灵活高效的IT架构，轻松连接未来。腾讯云提供的产品安全可靠、稳定易用，包括云服务器、云存储、云数据库和弹性web引擎等基础云计算服务以及腾讯云分析（MTA）、腾讯云推送（信鸽）等大数据运营服务。针对不同领域独特需求，腾讯云还推出一系列行业解决方案，例如微信解决方案、游戏解决方案、移动应用解决方案、视频解决方案等等。云端生态，价值共享，了解更多腾讯云信息，请见：<http://www.qcloud.com/>



云端生态·价值共享
www.qcloud.com



一些好的规则

作者 Peter Neumark , 译者 潘瑾瑜

什么是明智的标准化？

想象一下第一次和特别的人约会。当你到达最喜欢的餐馆时，所有的灯都熄灭了，你身处黑暗之中。奇怪的是，从厨房传来的声音又表明这里像往常一样正在营业中。你听到一位女服务员走来，等待着引导你到没有灯光照射到的座位上。你的同伴不知所措，并且有一点害怕。你是打算留下，还是找个正常点的地方吃饭？

Web应用就像餐馆一样，人们通过其所提供的体验对其进行评价。即使是短暂的中断也会影响服务提供商的口碑或服务水平。政策和指导方针在防止代价高昂的服务中断中扮演着重要的角色。不幸的是，它们也能导致不理智决策的产生，从而造成更大的损害。比如公司内“DevOps团队”的建立。这将导致所有的运维知识都被隔离在一个单独的团队中。尽管这样一个管理层指令可能预示着DevOps的到来，但它什么都不是。

工程师鄙视无逻辑的、官僚主义的规则。这些规则是前进的障碍物。然而，每家公司至少都会有一些这样的规则。在过去，可能有好的理由在一些问题上制定这样的规则。渐渐地，这些规则过时了。但是，规则制定者不能（或不敢）取消它们。当使用C++代码库时，由于历史原因，被告知不能使用STL；参与的Java项目被坚定地拒绝从1.4迁移到新版本。任何有过这样经历的人都明白有些措施可能会对生产力产生消极的影响。

我们应该忘记这些规则吗？

面对这些像障碍物一样的规则，我们都很想将它们废除。不幸的是，“无为”的公司通常都没有成功地废除它们。好的规则是一种重要的交流形式。这种形式关乎到长期策略、从过去吸取到惨痛教训、以及来自用户需求中的发现。理想情况下，一个组织制定的与时俱进的规则，可以帮助个人增强做出正确决定的信心。在实践中，这样的情况真的发生过吗？

一家公司是否拥有真正有效的指导方针，Netflix就是一个很好的例子。至少通过阅读他们的[博客](#)和[开源的代码](#)会给人留下这样的印象。比如，即使没有和Netflix的任何员工聊过，我也能确定，“[构建它，运行它](#)”是一个他们如何把开发和运维结合起来的不错的想法。另一个明确的原则是：写代码是为了构建一个可靠的、可扩展的服务，而不是为了其他目的。他们开源了所写的大部分后台软件。这个事实比任何事都更具有说服力。

Netflix已经构建了Netflix内部Web服务框架（NIWS）。这是一个自定义的软件栈，用于创建可靠地运行在云上的内部Web服务。NIWS采用了一些不太流行的技术和不太常用的技术。

方法。使用这种与最佳实践背道而驰的方法需要有相当强的自信。毫无疑问，部分可以归因于落后的政策。这些政策让工程师可以不受限制地考虑问题。

Netflix的负载均衡

在Netflix如何挑战常规的例子中，我最喜欢的是他们是如何在NIWS中实现负载均衡的。面向客户的流量仍使用传统的负载均衡处理器（一个标准的Amazon EC2 ELB），但对于Netflix服务器之间的流量，他们选择了一个完全不同的方案，称作客户端负载均衡（client-side loadbalancing）。基本思想很简单：取消专门用于负载均衡的节点。这些节点用于在Netflix服务器间转发流量。客户端本身维持着一张列表，记录了可用的后台节点。当客户端发送请求时，直接与所选的后台实例交互。而这样就没有必要使用专门的负载均衡器。

客户端负载均衡并不是Netflix发明的。但是，它是有名的公司里第一个在基础架构中完全使用这种技术的（公平地说，同一时期内，Twitter和Yahoo也在做基于相同概念的实验）。在多个后台服务器上做均衡的标准方法是：通过一个负载均衡器，如Amazon EC2 ELB，或者在服务器上运行类似HAProxy的软件。对于这么关键的组件，使用保守的方法和一种大多数工程师都熟悉的技术是很有意义的。但是，几乎没有公司在Netflix之前试验客户端负载均衡的方法。其真正原因是，他们甚至都没有考虑到这种方法。

对于从事大规模应用程序开发的软件工程师，每天都要和各种库和组件打交道。这有点像鱼和水的关系。在能使用一种特定方法成功地构建系统这么多年后（也许几十年），对已经经过考验的方法或者系统的构建模块提出疑问，这看起来是在浪费时间。在许多公司里，这些决定已经被写进政策中。这些政策基本上是不可变的。但是，Netflix采用了客户端负载均衡的方法，并因此取得了显著的成功。首先，他们从系统中移除了一个单点故障点（对于频繁地在没有警告下就停止服务的EC2实例，这是一个重大的胜利）。其次，通过将负载均衡的逻辑集成进客户端，负载均衡的策略可以参考客户端提供的信息。比如，考虑以下的负载均衡规则：

向客户端的EC2可访问区域（EC2 Availability Zone）中的可用节点发送请求。如果这样的实例不存在，则在当前区域中找一个运行已超过一天的实例替代。

传统的负载均衡器并没有被设计成用来执行这种自定义逻辑。它们也无法获取太多关于客户端的信息（比如一个客户端所属的EC2可访问区域）。自定义负载均衡逻辑变成了应用的一部分，使用相同的语言编写。这意味着编写代码的单元测试用例变得容易。而在传统上，这被认为是“基础设施的东西”。因此，这不仅让制定更复杂、更智能的决策成为可能；也使得人们对工作能如期完成更有信心。从某方面看，NIWS将DevOps带入了下一个层次：开发人员和运维工程师不仅坐在一起，在同一个团队中工作；而且他们使用相同的开发语言，向相同的代码库提交更新。

Prezi加入客户端负载均衡俱乐部

用一个内部的客户端负载均衡实现替代标准的负载均衡器，这种让Netflix受益的技术只适用于Netflix吗？不一定。在[prezi.com](#)，我们对内部流量也采用了这种技术。我们的一些应用服务器运行着若干个服务。当这些服务通信时，我们希望它们优先选择本地的服务实例，而不是向网络中发送请求。然而，如果需要访问的服务没有运行在同一台服务器上，那么就可以访问任何一个该服务的实例。对于Prezi，获得的好处是，尽可能地避免了网络流量、减少了在AWS上的支出和响应时间。目前运行于[prezi.com](#)产品上的负载均衡逻辑由以下的这段Scala代码实现：

```
override def choose(key: scala.Any): Server = Option(getLoadBalancer).map(lb =>
    lb.getServerList(true).filter{server =>
        server.getHost == config.getHostname && serverIsAvailable(lb, server)
    }.getOrElse(Seq())
) match {
    case Seq() => super.choose(key)
    case matchedServers => matchedServers(0)
}
```

Netflix的工程师可以设计出NIWS，而不用担心质疑当前技术所带来的后果。因为公司的规则授权他们这么做。即使任何人都可以获得NIWS的技术，只有那些有着类似思维的公司才能够使用这种技术去搭建产品。具体来讲，期望工程师基于技术价值做出决定的公司和完美主义的公司是无法利用这样的技术的。

Netflix验证（Netflix test）

期望所有的工程师在做决定时不受办公室政治、流行技术和害怕改变的制约，这是不可能的。然而，减少这些方面的影响，对确保开发不会误入歧途有很大的帮助。一堆武断的限制规定会让工程师的设计缺乏创新和效用。相比之下，一些好的规则限制了问题空间、明确了约束、改善了产品的质量。

基于NIWS栈的源代码，Netflix在决定如何实现一个组件时会考虑两件事：

1. 这个组件发生故障的可能性及后果是什么？
2. 当这个组件的设想场景发生改变时，是否容易修改这个组件的行为？

我将这些问题称为Netflix验证。这两个问题是紧密关联的。甚至可以说，第二个问题包含了第一个问题。这两个问题之所以意义重大，是在于他们如实地镜射出了Netflix的商业目标。这个目标就是提供可靠的、可扩展的服务。其他也有相同目标的公司也能从这个验证中受益。但是，这个验证的真正力量在于它没有提到的东西。它没有提到任何具体的技术或者[供应商](#)。

不适用于完美主义者

真正让我惊讶的是，Netflix的代码只专注于足够好即可，而无过之。别误解，目前我所看到的代码容易阅读，并且有很高的单元测试覆盖率。即便如此，我也没有预料到Netflix能专注在足够好这个级别。比如，在代码的许多地方，当后台线程启动之后，就再没有任何操作停止它们。这看起来有很大的问题，直到你意识到Netflix不在节点上进行软件升级。他们通过启动一个新的EC2实例集群来部署新版本的应用。当通过监控验证新版本应用运行正常后，就将老集群关闭。如果有人使用了这些部署工具（也是开源的），那么就没有僵尸线程的问题。然而，如果有人在一个像Glassfish的应用服务器上使用Netflix的库，那么每次重新部署都将会触发内存泄漏。

代码中包含大量单例模式的类，也是我未预料到的。当我们以一种Netflix未预见的方式使用一个NIWS库时，我们很快会发现自己在不断挣扎地使用错综复杂的技术来处理问题。包括使用多个类加载器。

最后，尽管wiki页面上关于代码的文档有很大的帮助，但是这样的文档太少了，很多细节都没有描述。通常，代码就是文档。有几次，我在github issue tracker上找到了一些解决NIWS相关问题的最佳建议。

我的许多同事，在第一次接触Netflix生态圈时都有点不知所措。他们的第一反应是谴责那些写代码的工程师未经训练或者太懒惰。“应该有一些规则关闭这些没用的线程”，我听他们这样说着。然而，对于Netflix，我们所列出的NIWS的缺点，都不算是一个真正的问题。用于处理线程关闭的时间被用在了其他更需要的地方。如果有人想要以不支持的方式重用代码，那么单例模式的类只是其中一个会遇到的问题。最后，尽管写文档是一件好事。但是，可读性高的代码和大量内部专业知识让文档成为了一个可选项。Netflix建立了关于线程管理、恼人的设计模式和最小化文档量的规则。通过建立这些规则，让工程师可以专注于其他主要问题。

事实上，我已经意识到Netflix的软件栈之所以成功，是因为它有着旺盛的精力。这不仅让Netflix“砍掉了软件栈的一些边角”的事实可以被接受，而且实际上也催生了一个更好的产品。编写了大量描述代码的文档还得保证文档不会过期，因为代码总是在不断的演进。编写不会用到的特性会使开发者失去动力、且难已为团队证明自己，对社区也没有什么好处，因为这部分代码不会在产品中被验证到。在Prezi，我们有一些一直想开源的项目。但由于缺乏时间加入一些我们希望的改进，目前还不能将它们开源。Netflix成功地开源了大量的代码却没有破产。因为它们一直专注于代码的可读性和单元测试，而不是加入过多的亮点，以及保证其不会过时。Netflix实施的这些合理规则，使得它的设计开发可以应对不断快速增长的用户；甚至是不断开源所写的代码。

因此所有的特定规则都不好吗？

如果用Netflix验证再形成一些指导方针，那么这些方针是相当通用的。例如，通过努力获得成功的名言，像“花10%的时间用于偿还技术债”，或者技术信息，如“0.6.1版的NodeJS使我们的Web应用变得不稳定，别使用它”。如果把从过去失败中总结获取的教训忘了，这难道不是一种浪费吗？

这样的建议，和最佳实践、知名的组件一样，是非常有价值的。在加速开发和简化系统的运维方面，通过多年的验证，这些建议已经获得了工程师的信任。比如在Prezi，大多数后台系统都是用Python写的，并使用了unicorn web服务器、Django web框架和MySQL数据库。在公司的初期，这个软件栈使得开发者能够专注于新产品的特性上。多年来，“使用Django和MySQL开发服务”就如同“不要在周五下午3点后部署”一样明确。这些都不是Prezi成文的规则，但却早已在实行中。

随着注册用户数从0攀升到4000万，许多当初采用这个平台的实际情况都已时过境迁。比如，当所有的网站流量都由一个应用处理时，将所有用户数据存入一个MySQL数据库中是有意义的。如今，Prezi拥有很多独立的服务。这些服务对响应延时、可靠性和一致性上都有着不同的需求。许多服务运行在EC2上，将数据库当做键-值存储的容器，通过主键访问数据。第一年制定的技术指导方针，尽管在那时有用，但没有一条能帮助我们应对目前工程上的挑战。

只要标准的技术和特定的规则没有过时，就能够激发工程师的产出。问题在于，当这些特定的规则不再适用时，仍然被强制实施。

固定的接口集

对于已经过时的规范而言，一个问题（而且很常见）是软件接口的过时。我最喜欢的例子是Java Servlet API。即使它并没有真的过时！实际上，它是一个优秀的接口：直观、稳定、有完整的文档、很多不同应用服务器都是使用它实现的。

当Prezi决定探索JVM，将其作为我们可靠的Django栈的一个可选平台时，我们选择了一个轻量级的代理应用作为我们的试用项目。我强烈地表明应该使用Jetty和Servlet API，而不是团队考虑的另一个可选方案。这个方案使用一个不知名的Scala Web服务器。6个月之后，我们关闭了原有的代理程序，而用一个基于[Spray](#)（这个技术我是投反对票的）写的程序取而代之。部分原因是：对于我们的用例，使用它可以获得更多效率。因为在我们的用例中，响应时间主要受发出的HTTP请求的响应延时的影响。我开始从代码层面思考：我们想要什么样的目标，想使用什么样的接口。我们如何写单元测试。开发者社区有多大。这正是Servlet API在抽象层面解决的问题。我本应该考虑（或谈论）关于它是如何利用硬件资源的。具体来说，瓶颈在于：处理请求时是否需要大量的CPU或者IO资源。由于在我们的用例中，大部分时间都花费在等待发出的HTTP请求的响应上，所以没有这样的资源要求。这就是代理程序的本质。鉴于我们的用例，使用Servlet的方法对每一个请求都创建一个专门的线程，不仅毫无必要地限制了处理请求的并行数，而且也无法高效地利用内存。

Servlet API不适用于这个问题的事实，并不能说明那些常用的接口或Java编程语言不好。数以千计的公司使用Servlet构建了令人惊叹的产品。其他编程语言也具有相似语义的Web服务器接口。这个故事想表明的是，我在使用特定的指导方针时脱离了实际的场景。接口是用来解决某一个特定问题的。当问题不再是尝试解决的问题时，使用给定的接口不是一个好的选择（无论这个接口有多流行或者多新颖）。

DevOps的规则

DevOps能量来自于合作中的人有着完全不同的技能。相比于成员技能单一的团队；一个拥有各种不同技能的团队，包括长满胡子的系统管理员、函数式编程的狂热粉丝，更有可能构建出可靠和可扩展的服务。

成员技术背景的不同使得团队更加需要明确的规则。开发者不需要知道为什么使用的自定义Linux内核有着一大串的编译参数。类似地，不是所有人都需要担心代码中有多少单例模式对象的存在。“写shell脚本时必须添加shebang行”，或者“解析用户数据的代码要有单元测试”。像这样的标准适用于团队中的每一个人，并且会帮助到那些在特定领域内没有足够经验做好事情的人。特定规则只有被适当的使用，才会对团队产生积极的作用。

更通用的规则，像这些Netflix验证只适用于制定高层级决策，但是能够应用地更久。管理团队既需要通用的规则也需要高层级的规则。诀窍是要及时发现我们制定的规则是否已不再发挥期望的作用。

如果我们回到文章开头的那个餐馆，打开冰箱门，不同盒子上有着不同的保质期时间。有的可能几个月，比如番茄酱；有的可能几个小时，比如鱼。做饭要用到不同的原料，而每种原料有自己的保质期。保持原料的新鲜，使得最终做出的食物可口，这是一个厨师的责任。同样，不仅在我们决定要将什么进行标准化这件事上需要智慧，在及时发现我们的标准是否已失去意义这件事上也需要真正的智慧。

关于作者

Peter Neumark是Prezi的一名devops人员。他和妻子Anna，以及两个儿子住在匈牙利的布达佩斯。当不用调试pyhton代码或换尿布时，Peter喜欢骑自行车。

参考英文原文：[A Few Good Rules](#)

感谢[赵震一](#)对本文的审校。

查看原文：[一些好的规则](#)

岑文初谈移动端开放插件平台的技术难点

作者 杨赛

千牛是阿里巴巴商家的多端开放式工作平台，每天服务500w+的活跃商家在移动和桌面端操作业务，同时千牛本身是一个开放的端体系架构，上面承载着上百家ISV的服务（还在不断增加），在端设计上做到开放和体验，开放和安全的平衡。

在2014年10月16-18日的[QCon上海大会](#)上，千牛团队技术负责人岑文初（放翁）将带来演讲《高效安全的开放式移动端平台架构设计》，介绍他们开放式移动工作平台的客户端及服务端架构演进，构建移动插件平台架构的思路和经验，并针对开放模式下移动客户端存在的速度、安全、稳定性等问题和案例进行分享。会前我们对放翁进行了邮件采访，邀请他介绍他们平台的最初设计思路、迭代流程、针对弱网络的优化等思路。

InfoQ：简单介绍一下千牛平台最初的设计思路，以及每次进行迭代是怎样一个流程去做？这是一个开放插件平台架构，是否跟其他的客户端在设计方面有所不同？

放翁：千牛起源于做卖家的移动工作台，希望解决卖家离开电脑也能够管理店铺，处理日常事务的问题。当时千牛技术产品设计方面考虑两个关键点：消息驱动行为，行为标准化开放。用一个简单的案例表述一下：一个卖家出门以后可以在手机上即时的收到一个买家拍下商品的消息，看到消息以后他可以立刻点击消息，平台会呼起订单管理插件并直接定位到具体订单，然后直接修改运费后发货（消息驱动行为）；订单管理插件可能是多家ISV开发，但都实现了平台的标准协议，当被设置为订单管理默认插件以后，平台负责消息路由（行为标准化开放，可以类比点击超链接会打开默认浏览器，因为浏览器实现了http协议的解析支持）。

最早千牛是支持Native插件和H5插件两种方式，分别提供了相关的SDK，这时候对于千牛应用来说，卖家感知的应用迭代背后其实是两部分产品的迭代：千牛平台自身产品发布（支持更多的服务和本地能力），ISV业务插件的产品发布（新业务的产生，老业务产品改进，基于千牛新服务和能力的二次开发产品更新），整体上来说千牛平台自身迭代速度就会相对来说比较慢（一个月一个版本），业务迭代比较快（一周就可以有一个迭代），同时千牛上的业务也是随时通过后台而扩展上线的。（具体的整体架构有三版的变迁，到时候QCon大会分享的时候可以更详细的和大家介绍这种开放式架构的变迁设计）

开放式的平台架构和普通App的差别，从业务上来说：平台应用对于业务的关注更抽象（标准化业务协议，数据互通的流程设计等），从技术上来说关注的更多不定因素：安全、三方插件的性能优化和可用性、多端统一接口差异实现等等。

InfoQ：现在千牛的移动活跃度和桌面活跃度大概是怎样的比例？有什么有意思发现吗？

放翁：千牛当前日活跃卖家数量暂时不方便透露，但PC和移动的活跃卖家人群当前重合度并不是很高，100w左右，同时通过对于不同用户的在千牛多端上的行为分析结合商家本身的特性可以看到，中偏小卖家已经基本可以脱离PC通过移动直接处理日常经营（这个在以前是很难想像的，卖家人群操作的复杂度是非常高的），大卖家很多决策人群使用移动端来看店铺的经营状况。而中大卖家的操作员角色大部分都使用PC千牛做复杂的业务处理，同时多端不同角色间已经产生了互动，千牛有任务转发来驱动一些业务处理场景由决策者在移动端接收，分配给PC端的操作者。

InfoQ：移动端插件平台架构的设计，跟桌面Web端开放平台的设计，有何异同？

放翁：移动端和PC端对于消息驱动行为，行为标准化开放的思路和实现都是保持一致的（他们只有一个大后台），但是不同端由于界面不同，操作入口的容器也不同，在移动端操作入口局限于首页、消息中心、数字中心，而在PC端除了移动端所拥有的这些，IM（旺旺）本身也是一个非常强大的容器，在聊天框右面形成了一个互动化非常强的插件容器。另外移动和PC的native的能力不同，通过SDK给多端应用开放的基础能力也有很大差异，例如移动端的摄像头用在很多业务插件上支持扫描条形码发货等功能，PC端的磁盘加网络能力可以用于客户端文件云化等等。

InfoQ：千牛针对弱网络环境有什么优化的经验可以分享吗？

放翁：千牛对于弱网络环境做了非常多的优化，但是都是针对千牛业务特点来定制的，千牛商家业务的特点就是**数据交互重，JS框架业务性强，图片变化量不大等**，千牛针对这些点做了Rainbow的设计（why no spdy?），为ISV插件支持Web容器可定制化缓存策略等。具体可以在这次QCon上海分享中做更多详细的介绍。

InfoQ：千牛平台在双十一应该也将面临大压力挑战，这方面目前是如何应对的？

放翁：当前日常的聚划算促销和各种节日大促都已经和双十一非常类似，服务端的推送也成为了移动和PC千牛最重要的环节。比如买家已经和卖家说“我拍了商品”，这一瞬间如果没有商品已拍的消息，那么用户明显能够感知，这个和很多社交类软件的信息不对称推送是完全不同的，因此千牛服务端推送系统即时性的需求就凸显出来，但是对于推送的数据还有非常多的复杂操作（数据补全，去重数字通知计算等），这又和业务即时性和稳定性产生了矛盾，加上客户端网络的不稳定性，大促推送的挑战可见一斑。

另外千牛IM模块压力非常大，例如小米家遇到大促，一个客服可能IM上就有几百个等着回复消息的买家闪亮着，同时这些消息不仅仅是文本消息，还需要把这些买家的业务信息带下来（是否有订单都必须标注在IM的买家闪亮黄条中），同时IM是一个大容器，IM每一个即时聊天客户的切换都会关联信息给到右边的插件，右边插件立刻联动的去获取用户信息，这样的内存消耗和网络消耗都是对一个客户端软件非常大的挑战。

同时双十一如何保证ISV软件不可用的时候快速切换到可用的其他ISV，并且在服务恢复的时候可以再次被切换回来；客户端模块在服务端业务不可用的时候如何快速降级，最后能够保证基础的聊天可以最高效率；这些都是千牛双十一的很大的挑战。这些挑战每

一个点都可以有非常多的内容可以讲，后续有机会可以让千牛团队的同学走出来给更多同行做分享。

InfoQ：说一个千牛平台做到现在，遇到的最大的挑战，以及解决的思路吧。

放翁：千牛平台的技术难点很多，就举Rainbow的例子吧。

当时千牛刚推出无线版本两个月（2013年初），一群人也没有无线开发经验，但是慢网络已经遏制了很多商家用千牛的动力了，因为大部分商家出外本身手机性能较差和网络非常不稳定（卖家地域遍布全国乡镇，卖家所处地点会是很多市场内部等等），因此千牛必须去做改进。但要知道千牛大部分的数据交互都是及时性的且无法缓存（交易订单等信息随时都在变化），同时业务都是交由不同ISV开发，开发模式不受控制，因此如何能够在弱网络下优化千牛平台自身和ISV对外的数据交互成了很大的问题。

当时有一个非常有利的点是，千牛最早SDK封装了所有的数据交换，也就是ISV不论是Native还是H5，都会通过千牛底层和远端做数据交换（这和业内已有的开放平台模式直接调用不同），当时是出于安全策略的需要，但这层的代理却给这种优化提供了收敛的策略保证，因此我们基于iOS和Android开发了基础层的数据交换，不改变上层的API调用模式，直接代理掉了Http协议，同时也把推送、配置更新、安全策略等都切换到了这个基础层，因此不仅提升了千牛上所有的Http数据交换可用性和效率，同时也实现了推送的全链路追踪，安全加密，压缩数据，数据变更通知（替代轮询）等功能。

打个广告，千牛是阿里内部的创业团队，当前负责阿里系所有的卖家（淘宝，天猫，1688，菜鸟）多端开放式工作平台。最早6个人从开放平台出来的服务端技术人员加上3个产品和一个运营，花了3个月时间打造了第一个移动版千牛，10个月时间移动端自然增长到100w DAU。当前团队包含开发、产品、运营（垂直化团队），技术涉及到移动端，PC客户端，高性能服务端，Webkit容器等内容，产品除了构建一站式商家工作台，还会建立商家互动服务平台，为商家生态服务圈提供更多有创新性的服务玩法。整个团队人员不多，欢迎优秀的同学加入一起来服务阿里巴巴所有的商家。联系方式：旺旺：放翁，微博：[@放翁_文初](#)

嘉宾简介

岑文初（放翁），2006年加入阿里巴巴集团，随后转入阿里软件创业团队，担任阿里软件基础平台架构师一职，负责了阿里软件基础服务平台架构设计与开发。2008年—2012年负责淘宝开放平台技术团队，为阿里集团构建对外的开放基础平台，开放服务数量从几十个发展到了上千个，业务类型从基础服务类扩展到阿里系新老买卖各条业务链，平台服务每日支撑60亿的调用量，并支持多终端多形态开发对接。2012年至今负责商家事业部千牛产品及技术团队，从无线和PC两端打造阿里商家开放式工作平台，千牛当前每日支撑500w活跃阿里商家（淘宝，天猫，1688等）在无线和桌面高效处理业务，同时千牛的开放式端和云设计模式为淘宝开放平台ISV提供了低成本高灵活性的新开发模式。

查看原文：[岑文初谈移动端开放插件平台的技术难点](#)

Whitepages 的架构变迁: 从 Ruby 到响应性更好的 Scala 和 Akka

作者 獡秀涛

Whitepages是位于美国的一家公司，主要负责提供个人和企业的联系信息，供用户搜索。其业务每个月要服务5000万独立用户，每天要完成3500万次搜索。其移动产品每个月也有超过1800万的活跃用户。

随着业务的增长，Whitepages的架构出现了瓶颈。经过评估，开发人员将出现瓶颈及代价较高的部分从原来的Ruby语言实现迁移到了更为现代、响应性更好的Scala语言和Akka框架。Whitepages的开发人员John Nestor和Dragos Manolescu[分享](#)了他们的经验。

在介绍了公司要应对的业务规模之后，他们提到了Ruby遗留系统存在的问题：

- 较高的延迟
- 较高的资源消耗，包括内存和处理器两个方面
- 对于上游服务的降级支持较差
 - 并发能力有限
 - 当阻塞在较慢的上游服务上时，工作线程会饥饿
 - 连接管理和恢复能力不佳

之所以选择Scala，是因为这门语言具有如下优点：

- 优雅地结合了函数式编程范型和面向对象编程范型
- 静态类型系统
 - 类型推导可以避免编写大量的Java样板代码
 - 编译器可以捕获很多错误
- 运行在JVM上
 - 速度快

- 几乎可以无缝地与JVM库互操作
- 相当成熟的工具支持
- 基于Actor的并发框架——Akka

Whitepages的响应式服务的特点：

- 面向服务的架构：通信采用Thrift或HTTP上的Json
- 延迟和吞吐量非常重要
- 对日志和监控有很高的要求
- 敏捷的开发、测试、构建和部署流程

在使用Scala和Akka迁移了服务之后，改进非常明显。

Service	p50 ms	p99 ms	through RPS/core
DirSvc - Scala	25	300	80
DirSvc - Ruby	140	1200	7

他们先从1个单一的后台服务入手，现在已经完成了4个服务的迁移；还有6个服务尚在开发之中。Scala开发人员也从最初的6个增加到20个。未来他们还将迁移更多服务。

他们总结的成功经验主要有以下几点：

- Scala简洁的语法提高了开发效率。
- 异步代码提高了性能。
- 不可变集合和函数式编程减少了bug。
- 强类型检查也有助于减少bug，并使代码的可维护性更好（不过元编程变困难了）。
- 并发能力提高。
- Spray具有极好的性能，而且提供了一个异步API。
- SBT能够根据需求轻松定制，尽管学习曲线有些陡峭。
- IntelliJ IDEA对Scala的支持非常好。

- Typesafe的开发者支持合约非常不错，Typesafe反馈非常快，对复杂的问题也可以给出很好的答案。

当然，迁移过程中也遇到了不少问题，比如：

- 差劲的文档，SBT就是个典型，很多时候还不得不阅读Scala和Akka库的源代码。
- API不稳定，升级步子太大。
- 缺乏好用的并发构件分析工具：尝试过Typesafe Console，但是一直没有完整地跑起来，最后放弃；虽然有些新工具，但没有时间一一评测。
- 生态系统不如Java，缺乏一些所需的组件；有时候选择太多，比如Json库就有10多款；GitHub上存在大量的Scala项目，但质量参差不齐。
- 难以调试，尤其是异步代码和Actor。
- 语言和库的问题：类型擦除是一个主要缺陷；Actor缺乏类型检查；某些Scala代码看上去简单直观，但是要了解其背后的机制也非常困难。

不过整体而言还是利大于弊，Scala/Akka非常适合构建响应式系统。

最后，他们讲到了开发人员这个关键因素。有经验的Scala开发人员还不够多。所以他们一方面招聘Scala开发人员，一方面培训现有的Ruby开发人员，促其转型。

更多细节，可以观看[讲座视频](#)或[下载讲稿](#)。

Whitepages并不是第一家尝试从Ruby向其他开发语言迁移的公司。Twitter早在2011年就开始[从Ruby向Scala和Java迁移](#)。[Iron.io从Ruby迁移到Go](#)，服务器从30台减少到2台。[LinkedIn从Rails迁移到Node](#)，服务器减少了27台，速度提升高达20倍。

项目创建初期，开发效率往往是首先要考虑的，以保证产品尽快推向市场。而随着业务规模的扩大，性能、可伸缩性方面的需求又会凸现出来，上述几家公司都选择了切换编程语言。

感谢[郭蕾](#)对本文的审校。

查看原文：[Whitepages的架构变迁：从Ruby到响应性更好的Scala和Akka](#)

架构师

www.infoq.com/cn/architect

每月8号出版

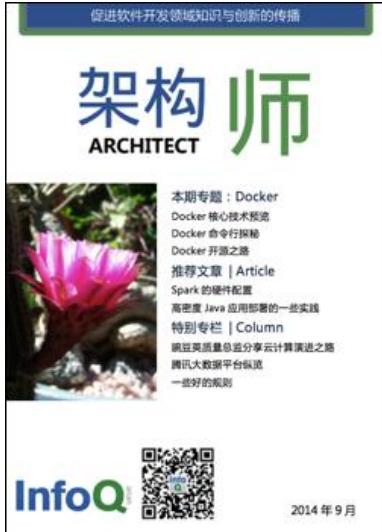


封面植物：不知名的丽花球



本期的封面植物照片来自 [Oregon Cactus Blog](#)，是杂交而生的丽花球（*Lobivia*），因此具体品种不明。

仙人掌科丽花球属，中型球状植株，单生或群生。棱沟很浅，棱又横分为不明显的斧状突起。刺细，数量多。花中等大，钟状至漏斗状，子房和花托筒密被鳞片和长软毛，少数种在于房腋部有刺或刚毛。丽花球喜爱寒冷而干燥的冬天，在这种环境下过冬后，它们在春季会开花。花色有红、紫红、洋红、堇粉、橘黄、黄、白等，非常艳丽，花朵可能会长大到覆盖整个植物。



架构师 2014 年 9 月刊

每月 8 号出版

本期主编：郭蕾

策划编辑：杨赛

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com

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

商务合作：sales@cn.infoq.com 15810407783



本期主编：郭蕾，InfoQ 技术编辑，文艺范儿程序员，并发编程网站长。在 CRM 行业厮混 3 年多，喜欢技术写作和社区运营，信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。

<mailto:editors@cn.infoq.com>