001
2016-01-13

架構
ARCHNOTES
高 可 用 架 构

创刊号

Docker 实战

参加那么多技术大会你学到了什么?

互联网金融创业公司的Docker实战

Docker在芒果TV的实践之路

微博基于Docker容器的混合云迁移实战

使用开源Calico构建Docker多租户网络

张磊：关于Docker、开源，以及教育的尝试

Joyent首席技术官关于容器的新年愿望

Dawn of space exploration! © Wojciech Tymicki

Docker实战

Docker实战

《高可用架构》创刊号卷首语：

参加那么多技术大会你学到了什么？

作者 / 杨卫华

杨卫华 (Tim Yang)，新浪微博研发副总经理，2008年加入新浪，2009年起参与新浪微博的技术架构工作，在海量及峰值访问、大数据、NoSQL存储、异地机房分布式架构及开放平台等方面参与并推动多次技术改进。工作之余喜欢各种技术交流，经常通过微博发表技术观点。

在2015年末北京一场技术大会上，一群网上相识的同行正在会场附近一家餐厅相聚午餐，来自广州的架构师小白率先起身赶往下午场，因为他需要回去公司转述参加本次大会的一些精华，尽管要完成这个作业、也就是总结一些对大家有启发的内容相比于前几年越来越难，对于部分参会人员，即使写一条140字的微博感悟可能也有挑战。

在另外一个会议，Tim 也碰到来自武汉一家技术公司的创始人猫总，他来北京参会主要为了定期感受一下帝都互联网的氛围，同时也是趁机拜访一些同行。对于大部分技术人员来说，参加技术会议是弥补自己日常岗位或者环境视野不足的一个有效途径，技术人员由于自身产品及业务需要，可能长期专注在技术某一个特定领域，但整个技术行业在发生日新月异的变化，通过技术大会可以在较短的时间内，了解行业的动态及发展方向，了解其他团队正在做的项目及研究成果，可以在较短时间内迅速补充自身视野的不足，并根据对技术行业新的理解，调整自身的技术研发方向。

但是技术大会也存在一些效率的问题，比如现在的技术会议出于商业的考虑，参会人数较多，一个分会场的演讲通常有数百人参加，

因此可能存在体验较难保证的问题，比如热门 Topic 的座位不够、后排屏幕看不清、音响声音不够清晰、会场空气流通差或温度不舒适的问题，导致获取知识的效果大打折扣。

在另外一方面，由于一场大会的讲师通常非常多元化，来自不同的行业及公司，分享的内容侧重点各不相同。但存在讲师对听众缺乏了解，不了解听众的诉求与期望，因此很容易出现讲师内容对大部分听众针对性不强的问题。听众需要具有代表性的行业发展动态，但由于编辑精力及视野的局限、讲师自身能力及视野的局限，这一点很难保证。讲师分享的内容有如在一些没有菜单的私房菜餐厅，餐厅提供什么顾客就吃什么，这种方式非常考验厨师的功底，对应就是策划编辑及讲师的功底。但是大多时候，组织方的主要重心只是保证上一桌菜并保证表面光鲜，能够进一步将顾客吸引过来，至于味道怎么样，只有吃完后大家才知道。这种情况下，参与技术大会是否能够获取优质内容成为了一件预期不可确定的活动。

由于能否获得干货不可确定，资深一些的技术群体往往将技术大会定位成社交活动，他们认为参加会议主要是为了结识讲师及不同领域的同行，平时不太有机会一次遇到这么多技术同行，而通过主办方的组织，同行有机会聚集在一起，通过线下交流来拓宽自己视野，了解不同团队最近的动态与变化。但是这只适合有一定人脉资源及自身有一定影响力的人物，他可以实现私下联系或者通过日常交流的论坛或群聊召集同圈子的人聚集在一起，然后再在会场周围进行小圈子活动。但对于大多数人来说，在一个人山人海的大会，缺乏

一些组织机制来达成社交的目的。讲师由于面对一大群提问的人员，很难与某一个人员进行长期或者深度的交流；参与者由于与讲师地位不对等。也很难短时间达成社交的目的。而由于人数及体验的考虑，主办方组织的的宴会大多针对 VIP 及讲师，普通参会人员很难参与其中，因此普通参会者很难去在一个上千人的大会中达成社交目的。

从上面来看，大部分参加技术人员的需求主要有：拓宽视野为主的知识需求、以及拓展人脉为主的社交需求。这些需求是否有更好、更简单的方法来实现？下面简单谈下获取高质量内容的一些思考。

首先知识获取的需求肯定可以互联网化，通过网络本来就可以获取多媒体，包括视频、语音、图片、文字等，而大屏幕显示器、iPad、Kindle 等设备可以让阅读及观赏保持一个较好的体验。在优质内容的广度和深度可以保证的前提下，这些数字化的内容可以更方便、更及时、更广泛的触达最终的读者，并通过评论、群聊等讨论等反馈交流机制获得更深度的理解。

如何通过互联网产生高质量内容是非常有挑战的问题，在心理程度上，大会讲师可以得到比一篇文章作者更高的光环和荣誉，因此有经验的讲师在有意愿的前提下，更愿意去参加线下会议来提升自己的专业声望。由于自媒体的流行，通过类似博客的工具写作门槛非常低，大部分文章的质量良莠不齐，因此大部分从业人员很难通过短期的写文章脱颖而出得到业界的认可。

不过情况也在发生微妙的变化，近半年，基于在线群聊的“微课堂”

突然在各个圈子开始流行，这些“微课堂”大多是组织方邀请各种背景的讲师，直接通过在线图片、文字或语音的方式跟所有群里面的成员介绍一个专业的话题，参与的成员可以实时的跟讲师互动提问。参加者可以更有效率的了解分享话题的内容，而且在不离开自身舒适的环境的情况下。这种方法看起来不太正式，但却可以通过非常低的成本获取相同量的内容。

微课堂也存在策划编辑及讲师视野局限及选题把关的问题，因此纯运营型的组织其实很难做好这件事情，有可能是将线下大会 Topic 质量参差不齐的情况再一次带到线上。如果有一批具有专业背景且有情怀的人，在一个合适在线平台的组织下，尝试不同的方式 (不局限目前的微课堂的形式)，更容易将这个事情简单的回归本身，通过互联网来轻量的产生及获取高质量内容。

在架构的软件交付领域，2015 年最突飞猛进的莫过于 Docker 了，在 2014 年末时，Tim 曾经计划将 “2015年主要精力投入到 Docker 上”，2015 年也已经过去，团队在 Docker 工作中也取得了不少收获。因此本期主要围绕 Docker 的架构，从 2015 年高可用架构多篇文章整理，以及业界对于 2016 年的 Docker 发展一些展望。

关于「高可用架构」

高可用架构由新浪微博 Tim Yang 发起创建的一个社区组织，主要关注互联网架构及高可用、可扩展及高性能领域的知识传播，订

阅用户覆盖主流互联网及软件领域系统首席架构师、技术负责人等技术从业群体。

用微信扫描关注公众号可以更多了解微博、腾讯、Twitter、Uber 等公司在 2015 年架构领域的分享及实践，以及获得高可用架构群的申请方法。■

高可用架构

改变互联网的构建方式

微信扫描二维码 添加高可用架构

架構 ARCHNOTES 高可用架构

Docker实战

互联网金融创业公司的Docker实战

作者 / 高磊

雪球运维架构师。2012年主持研发和运维 Redis 集群等分布式系统，专注于中小型公司运维架构和运维体系建设，目前在雪球负责技术保障工作，在Docker容器及其周边生态系统积累了大量一线经验。

背景介绍

雪球（http://xueqiu.com）是一家涉足证券行业的互联网金融公司，成立于2010年，去年获得C轮融资。雪球的产品形态包括社区、行情、组合、交易等，覆盖沪深港美市场的各个品种。

与传统社交网络的单一好友关系不同，雪球在用户、股票基金及衍生品、组合三个维度上都进行深度的相互连接。同时雪球的用户活跃度和在线时长极高，以致我们在进行技术方案选型和评估的时候必须提出更高的要求。

目前雪球的 DAU 为1M，带宽为1.5G，物理机数量为200+，云虚拟机数量约50个。雪球采用了 Docker 容器作为线上服务的一个基本运行单元，雪球的容器数量近1k。

容器选型

我们的服务面临的操作系统环境大致有五种：物理机、自建虚拟机、云虚拟机、LXC 容器、Docker 容器。

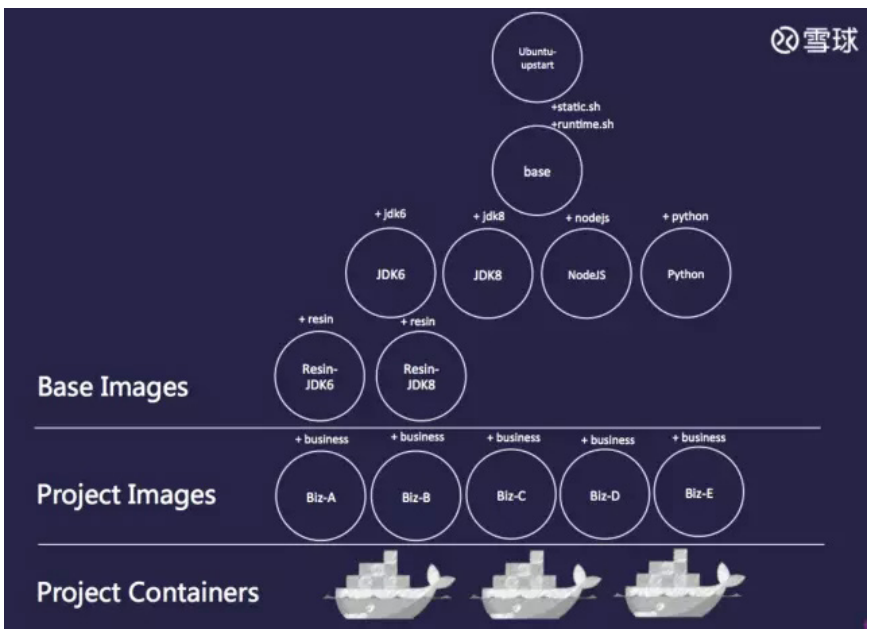
我们主要的考虑的选型指标包括“成本”、“性能”、“稳定性”三个硬性基础指标，以及“资源隔离能力”、“标准化能力”、“伸缩能力”这几个附加指标。这五类操作系统环境在雪球的实践如下：

操作系统环境	在雪球的使用场景	原因
物理机	Redis、Nginx、LVS、Hadoop等	性能最优；有kernel调优的能力
自建虚拟机	行情接收等windows服务	异构OS
云虚拟机	接入层Nginx代理	硬件资源优势；网络优势；基础设施和产品化程度高
LXC 容器	MySQL、运维工具	持久化不受AUFS约束；原生有Daemon能力
Docker 容器	无状态业务	极轻；标准化

应用迁移

我们 SRE 团队借助 Docker 对整个公司的服务进行了统一的标准化工作，在上半年已经把开发测试、预发布、灰度、生产环境的所有无状态服务都迁移到了 Docker 容器中。雪球借助 Docker 对服务进行的标准化和迁移工作，主要是顺着 Docker 的 Build -> Ship -> Run 这三个流程进行的。同时在迁移过程中填了许多坑，算是摸着石头过河的阶段。

我们首先设计了自己的 Docker Image 层级体系。在 Base 这一层，我们从 ubuntu-upstart 镜像开始制作 base 镜像。这里做了动静分离，静态的部分 (static.sh 和对应的 static 文件) 和动态的部分 (runtime.sh) 都会被添加进 base 镜像，静态部分会在构建 base 镜像过程中被执行，而动态部分会在启动 project 镜像的过程中被执行。详情如下图所示：



其中静态部分我们更推荐灵活的使用 shell 脚本来完成多个基础配置，而不是写冗长的 Dockerfile。下图是一个很好的对比，左面是gitlab官方的dockerfile，右面是我们的动静分离实践。

然后基于 base 镜像，我们又添加进入了 JDK、NodeJS 等运行环境，并在JDK的基础上进一步构建了 Resin 镜像。在上述镜像中再添加一层业务执行代码，则构成了业务镜像。不同的业务镜像是每个业务运行的最小单元。在雪球我们大力的推进了服务化，去状态。这样的一个业务镜像就具备了迁移部署、动态伸缩的能力。需要提醒的是 docker build 镜像的过程中会遇到临时容器的一些问题，主要涉及访问外网和dameon能力。Docker 构建这块就跟大家分享这么多。接下来咱们聊聊分发这一步。

分发的过程痛点不多，主要是 Registry 的一些与删除相关的 Bug、Push 镜像的性能，以及高可用问题。而本质上高可用依赖于存储的高可用。在雪球我们使用了硬件存储，接下来计划借助 Ceph 等分布式文件系统去解决。

说完分发，我们谈谈运行这个环节，Docker 运行的部分可以探讨的内容较多，这里分为网络模型、使用方式、运维生态圈三部分来介绍。绝大多数对 Docker 的网络使用模型可以汇总为三类：bridge（NAT）、bridge（去 NAT）、host。Docker 默认的桥接是用的第一种 NAT 方式，也即是把命名空间中的 veth 网卡绑定到自己的网桥 docker0。然后主机使用 iptables 来配置 NAT，并使用 DHCP 服务器 dnsmasq 来分配 IP 地址。

在雪球我们对 Bridge 模式去掉了 NAT，也即把宿主机的 IP 从物理网卡上移除，直接配置到网桥上去，并且使用静态的 IP 分配策略。据了解，有好多其他公司在自己的 IDC 机房中也是采用了去 NAT 的桥接模式，这样的好处是 Docker 的 IP 可以直接暴露到交换机上，性能最高。而端口映射方案不容易做服务发现，雪球并没有使用。其中去 NAT 的 Bridge 模式需要在宿主机上禁用 iptables 和 ip_forward，以及禁用相关的内核模块，以避免网络流量毛刺风暴问题。

说完网络，我们看下一些使用场景。我们在 docker run 的时候通过 —mac-address 传入了静态 MAC 地址，并通过前述 runtime.sh 在运行时修改了 docker 的 eth0 IP。其中计算 MAC 地址的算法是：

```
IP="aa.bb.cc.dd"
MAC_TXT=`echo "$IP" | awk -F'.' '{print "0x02 0x42 "$1" "$2" "$3" "$4}'`
MAC=`printf "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x" $MAC_TXT`
```

同时我们按照雪球统一的运维规范在运行时修改了容器 Hostname、Hosts、DNS 配置。

在交互上，我们在容器中提供了 sshd，以方便业务同学直接 ssh 方式进入容器进行交互。Docker 原生不提供 /sbin/init 来启动 sshd 这类后台进程的，一个变通的办法是使用带 upstart 的根操作系统镜像，并将 /sbin/init 以 entrypoint 参数启动，作为 PID=1 的进程，并且严禁各种其他 CMD 参数。这样其他进程就可以成为 /sbin/init 的子进程并作为后台服务跑起来。

然后跟大家介绍下我们在 docker 周边做的一些运维生态圈。

首先我们对容器做了资源限制。一个容器默认分配的是 4core 8G 标准。CPU 上，我们对 share 这种相对配额方式和 cpuset 这种静态绑定方式都不满意，而使用了 period + quota 两个参数做动态绝对配额。

在内存上我们禁用了 swap，原因是当一个服务 OOM 的时候，我们希望服务会 Fast Fail 并被监控系统捕捉到，而不是使用 swap 硬撑。死了比慢要好，这也是我们大力推进服务化和去状态的原因。对了，顺便提一下 docker daemon 挂掉，或者升级，我们也是靠应用层的分布式高可用方案去解决。只要去掉状态，什么都好说。

在日志方面，我们以 rw 方式映射了物理机上的一个与 Docker IP 对应的目录到容器的 /persist 目录，并把 /persist/logs 目录软连接为业务的相对 logs 目录。这样对业务同学而言，直接输出日志到相对路径即可，并不需要考虑持久化的事宜。这样做也有助于去掉 docker 的状态，让数据和服务分离。日志收集我们在 java 中使用 logback appender 方式直接输出。对于少数需要 tail -F 收集的，则在物理机上实现。

在监控方面，分成两部分，对于 CPU、Mem、Network、BlkIO 以及进程存活和TCP连接，我们把宿主机的 cgroup 目录以及一个统一管理的监控脚本映射到容器内部，这个脚本定期采集所需数据，主动上报到监控服务器端。

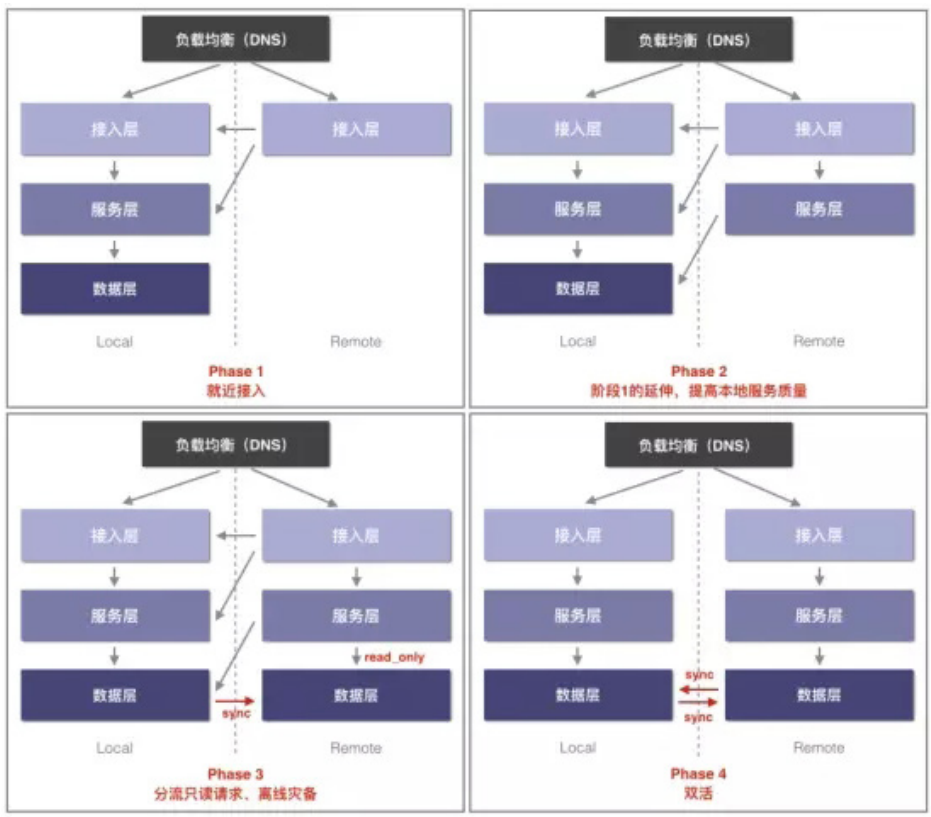
对于业务自身的 QPS、Latency 等数据，我们在业务中内嵌相关的 metrics 库来推送。不在宿主机上使用 docker exec API 采集的原因是性能太差。据说 docker 1.8 修过了这个 docker exec 的 bug，我们还没有跟进细看。

以上是我们在上半年的一些工作，主要方向是把业务迁移入 docker 中，并做好生态系统。

弹性扩容

大家知道上半年时，股市异常火爆，我们急需对业务进行扩容。而通过采购硬件实现弹性扩容的，都是耍流氓。雪球活跃度与证券市场的热度大体成正相关关系，当行情好的时候，业务部门对硬件资

源的需求增长是极其陡峭的。在无法容忍硬件采购的长周期后，我们开始探索私有云＋公有云混合部署的架构。我们对本地机房和远端云机房的流量请求模型做了一些抽象，如下：



我们把服务栈分为接入层、服务层、数据层三层。

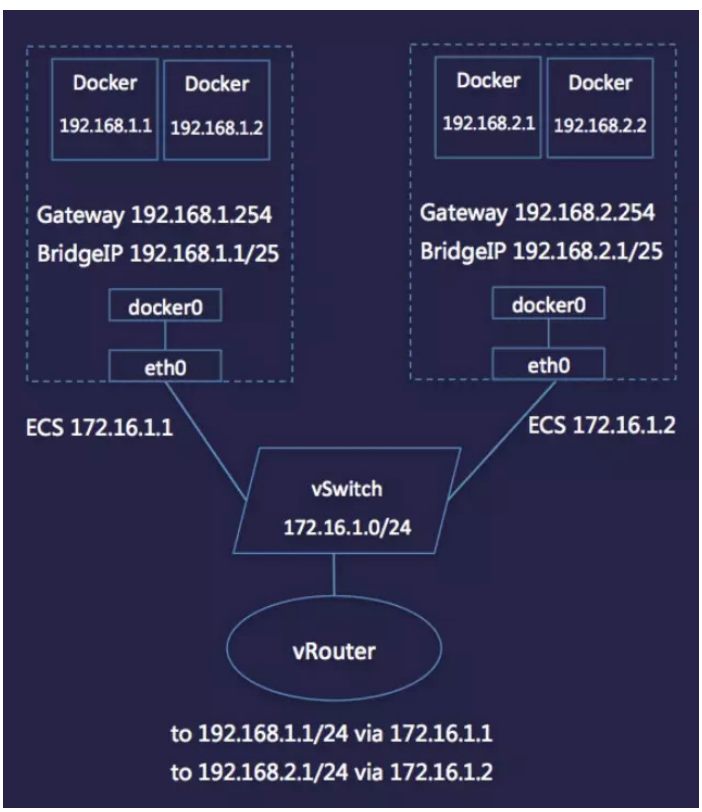
第一个阶段，只针对接入层做代理回源，目的可以是借助公有云全球部署的能力实现全球就近接入。

第二个阶段，我们开始给远端的接入层铺设当地的服务层。演进到第三个阶段，远端的服务层开始希望直接请求本地数据。

这里比较有趣的一点是，如果远端服务是只读逻辑，那么我们只需要把数据做单向同步即可。

如果要考虑双向同步，也即演进到第四个阶段，也即我们所说的双活。其中不同机房之间的流量切换可以使用 DNS 做负载均衡，在雪球我们开发了一套 HTTP DNS 较完美的解决此问题。

目前雪球的混合云架构演进到第三个阶段，也即在公有云上部署了一定量的只读服务，获得一定程度的弹性能力。在公有云上部署 Docker 最大的难题是不同虚拟机上的 Docker 之间的网络互通问题。我们与合作厂商进行了一些探索，采用了如下的 Bridge (NAT) 方案。



这本质上是一个路由方案。首先虚拟机要部署在同一个 VPC 子网中，然后在虚拟机上开启 iptables 和 ip_forward 转发，并给每个虚拟机创建独立的网桥。网桥的网段是独立的 C 段。最后在 VPC 的虚拟路由器上设置对应的目标路由。这个方案的缺点是数据包经过内核转发有一定的性能损失，同时在网络配置和网段管理上都有不小的成本。庆幸的是越来越多的云服务商都在将 Docker 的网络模型进行产品化，例如据我们了解，阿里云就在向 docker 提交 docker machine 的 driver。

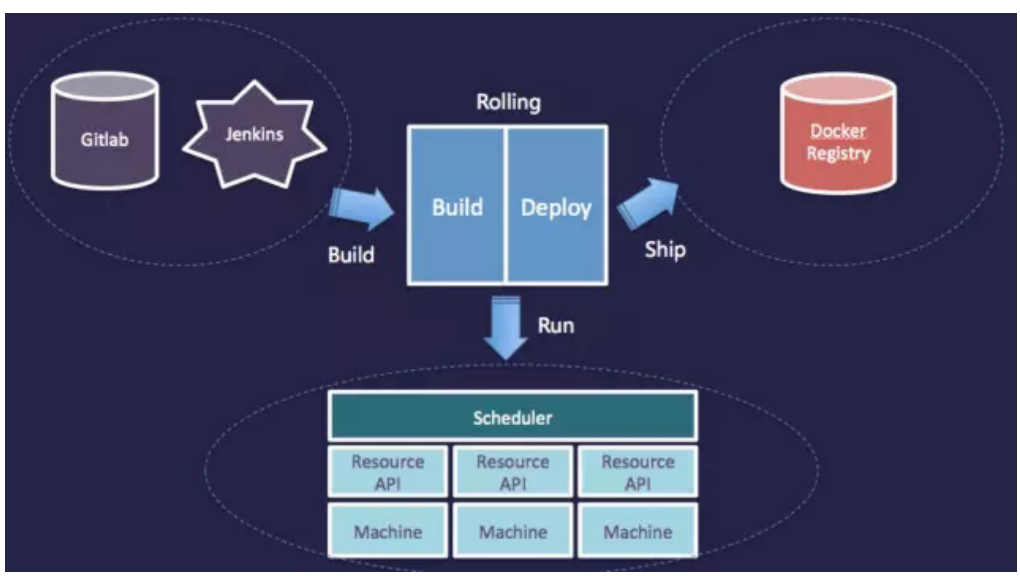
除了路由方案外，另一个方案是隧道方案。也即铺设 ovs 之类的 overlay。但是，在公有云上本来底层网络就是一层 overlay，再铺设一层软件 overlay，性能必然会大打折扣。我们最希望的，还是公有云自身的 SDN 能够直接支持去 NAT 的 bridge。

当使用 Docker 对服务进行标准化后，我们认为有必要充分发挥 Docker 装箱模型的优势来实现对业务的快速发布能力，同时希望有一个平台能够屏蔽掉本地机房与远端公有云机房的部署差异，进而获得跨混合云调度的能力。

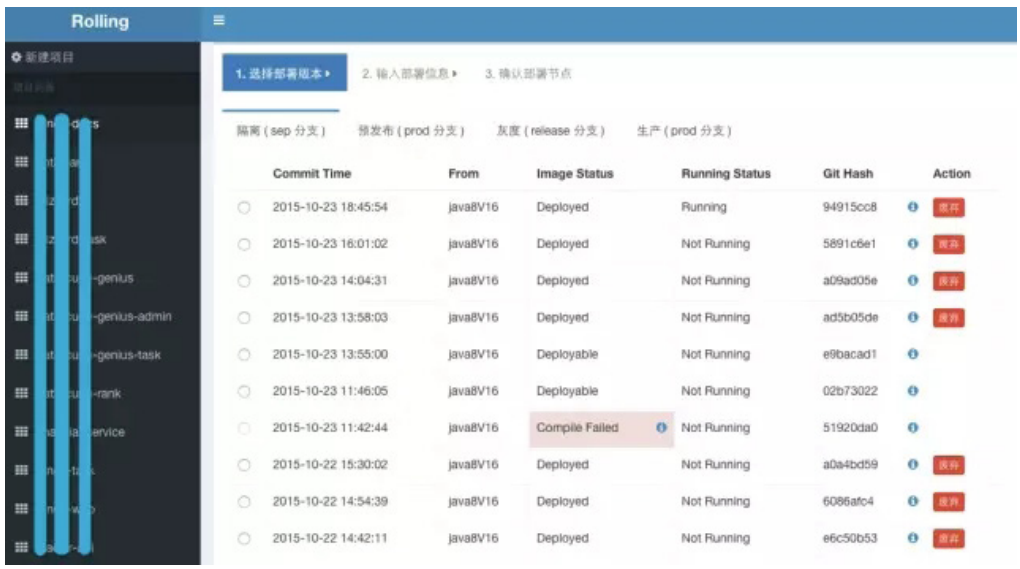
于是我们开发了一套发布系统平台，命名为 Rolling，意喻业务系统如滚雪球般不断向前。 在此之前，雪球有一套使用开源软件 Capistrano 构建的基于 ssh 分发的部署工具，Rolling 平台与其对比如下：

对比	Capistrano	Rolling
编译	部署现场人工触发编译（编译后置，费时费力）	代码提交时触发编译（编译前置，省时省心）
分发	基于 ssh	基于 Docker Registry 和 Docker API
运行	环境越跑越"脏"	环境是可重复的"干净"的
流程控制	Shell（能力过于开放）	Web（能力被规范化）
代码控制	线上代码和环境 =? RC 验证的代码和环境	线上镜像 == RC镜像
版本控制	在业务本地使用 xxx.bak 备份历史版本	在 Docker Registry 存储历史镜像
伸缩调度	慢(几十分钟级别)	快(秒级别)
Hook 功能	弱	强

大家可以点开图细看下，Rolling 帮助我们解决了非常多的痛点。像编译时机、环境干净程度、代码验证、版本控制，等等。Rolling 的上下游系统如下图所示：



下面截取了几张 Rolling 平台在部署过程中的几个关键步骤截图：



上图显示了 Rolling 在部署时的第三个步骤：资源选择。目前雪球仍然是靠人力进行调度配置，接下来会使用自动化的调度工具进行资源配置，而 Rolling 已经赋予我们这种可能性。在真正开始部署之后，还有一键暂停、强制回滚、灰度发布的功能。

上图显示了某业务在使用 Rolling 部署后的运行状态。

Rolling 平台带来的质变意义有两方面:

其一从运维同学的角度，Rolling 使得我们对服务的调度能力从静态跃迁为动态。而配合以大力推进的服务去状态化，Rolling 完全可以发展成为公司内部私有 PaaS 云平台的一款基石产品。

其二从业务同学的角度，其上线时不再是申请几台机器，而是申请多少计算和存储资源。

理想情况下业务同学甚至可以评估出自己每个 QPS 耗费多少 CPU 和内存，然后 Rolling 平台能够借助调度层计算出匹配的 Docker 容器的数量，进而进行调度和部署。也即从物理机（或虚拟机）的概念回归到计算和存储资源本身。

未来规划

一方面，我们非常看好公有云弹性的能力，雪球会和合作厂商把公有云部署 Docker 的网络模型做到更好的产品化，更大程度的屏蔽底层异构差别。另一方面，我们考虑在资源层引入相对成熟的开源基础设施，进而为调度层提供自动化的决策依据。■

Q&A

Q1：同时维护五类操作系统环境，是不是太多了？

多套环境，是确实有多种需求，很难砍掉。物理机，不用说了，上面要部署nginx/redis等等。KVM，因为雪球要装一些windows，来对接券商等传统行业（必须要求windows）。LXC，确实是历史遗留，不排除切换到物理机，但是目前没有动力。Docker，无状态服务。PS. 我们实践中，只把无状态的服务放到 Docker 中，有状态的不会放。

Q2：跨机房的事情如何处理的？

跨机房这块我们在本地机房与公有云之间铺设了专线。目前使用情况如我跨机房图的第三图所示，也即云端有只读服务。

Q3：公有云和私有云拉专线么，雪球处于哪个阶段？

跨机房这块我们在本地机房与公有云之间铺设了专线。目前使用情况如我跨机房图的第三图所示，也即云端有只读服务。

Q4：调度参考监控系统的哪些指标？

我们调度目前还没真正做起来，参考的值必然会包括操作系统层面的指标（CPU/Mem/Network/BlkIO），同时应该会参考业务的指标（QPS、Latency）等等。

Q5：能详细说一下用shell脚本配置如何做，雪球是不是已经自研一套shell脚本组件来管理配置，代替dockerfile？

shell 脚本（也即capistrano这套基于ssh的发布系统），逻辑上就是git拉取代码，然后执行编译，然后打包，然后基于不同项目的配置，scp到不同的目标机上，启动起来。我们并没有替代 dockerfile，而是简化dockerfile，而且这一步是在构建base镜像时做的。跟业务部署并没有关系。

Q6：目前雪球有多少个业务（核心的）跑在docker上？

我们的 docker 总体数量大概是1000个左右，其中大约有1/5是测试、预发布环境的，剩下的800个左右线上。我们总共有40个左右的业务，有的大，有的小。

Q7：rolling打算开源吗？

最初考虑过这个问题，可能拆除业务逻辑后有机会吧。我个人理解，在的许多公司一定有能力自行研发一套自己的 Rolling，重要的是去想清楚它的思想，从代码生产，到上线提供服务，中间到底需要什么，如何进行标准化，这是我们最受益的。

Docker实战

Docker在芒果TV的实践之路

作者 / 彭哲夫

芒果TV平台部核心技术团队负责人，主要负责Docker 和 Redis Cluster相关的基础设施开发。 前豆瓣 App Engine 核心主程，前金山快盘核心主程。在系统工程方面经验丰富。彭首席，知识丰富，功底深厚，语言幽默风趣，知乎上、简书上有不少彭首席的精彩大作和回复。

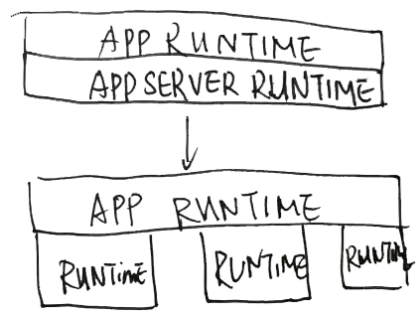
在这篇文章中，我将和大家分享一下最近几年我做平台的一些思考，并介绍一下目前我做的这个基于 Docker 的项目的一些技术细节，以及为什么我们会那么做。

豆瓣时期

我在豆瓣工作的时候，主要是写 Douban App Engine 。大体上它和 GAE 类似，有自己的 SDK 和服务端的 Runtime。因为是对内使用，所以在 SDK 和 Runtime 实现细节上，我们并没有像 GAE 那样做太多的 Mock 来屏蔽一些系统层面的 API (比如重写 OS 库等)。对于一家大部分都是使用 Python 的公司而言，我们只做了 Python 的 SDK 和 Runtime，我们基于 Virtualenv 这个工具做了运行时的隔离，使得 App 之间是独立分割的。但是在使用过程中，我们发现有些运行时的隔离做得并不是很干净，比如说我们自己在 Runtime 使用了 werkzeug 这个库来实现一些控制逻辑，然后叠加应用自身 Runtime 的时候可能因为依赖 Flask ，因此也安装了另一个 werkzeug 库，那么到底用哪个版本，就成了我们头疼的一个问题。

一开始我们考虑修改 CPython 来做这件事，包括一些 sys.path 的黑魔法，但是发现成本太高，同时要小心翼翼的处理依赖和路径关系，后面就放弃使用这种方法了，采用分割依赖来最小化影响，尽量使得 Runtime 层叠交集最小。

然后 App Runtime 和 AppServer Runtime 就成了这样：



进入 2013 年，Docker 在 3 月默默地发布了第一个版本，我们开始关注起来。紧接着我就离职出发去横穿亚洲大陆了，一路上提着 AK 躲着子弹想着 Docker（大雾）并思考如何通过它来做一个或者改造一个类似 DAE 一样的 PaaS，直到我回国。机缘巧合之下，我加入到芒果TV，隶属于平台部门，有了环境便开始尝试我在路上产生的这些想法。

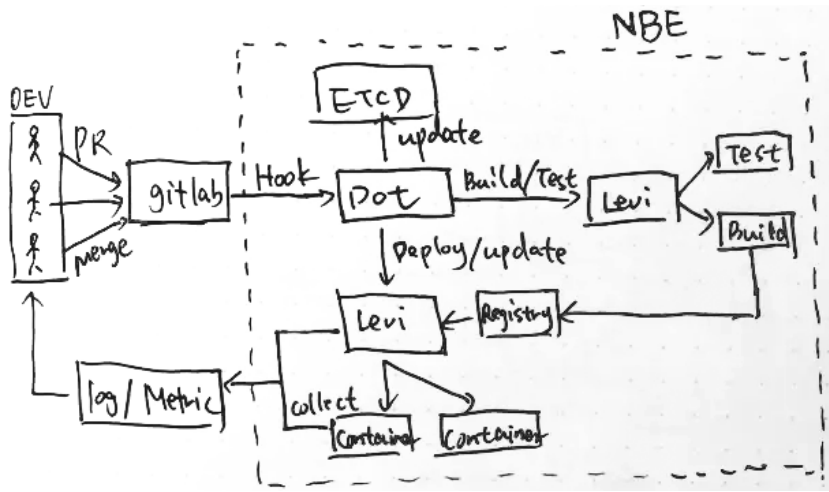
芒果 TV 的 Nebulium Engine

加入芒果TV之后，一开始我实现了类似 DAE 架构的一个新的 PaaS —— Nebulium Engine（a.k.a NBE），只不过运行时完全用 Docker 来隔离，控制层移到了 Container 之外。除此之外，整体架构上和 DAE 并未有太多差别。

同时我遇到了一个很是头疼的问题——那就是在芒果 TV ，我们并没有一个大一统的强势语言，我们必须把 Runtime 的控制权完全交给业务方去决定。综合大半年线上运行结果来看，在资源管理和工作流整合上面，其实 NBE 做得并不是很好。原因很多，一方面是基础设施和豆瓣比实在太糟糕，如果说豆瓣是21世纪互联网的话，我们现在还停留在19世纪的传声筒，另外一方面五花八门的语言都需要支持，放开 Runtime 之后，对资源竞争和预估我们是完全没有任何能力去做的，恰恰业务方又希望平台能 hold 住这些事。

举个例子，Python GIL 限定在非多进程模式下顶多吃死一个核，然后隔壁组上了个 Java 的 Middleware……然后我们就看到 Python 业务方过来哭天喊地了。

在这个时期，我们的架构是这样的：



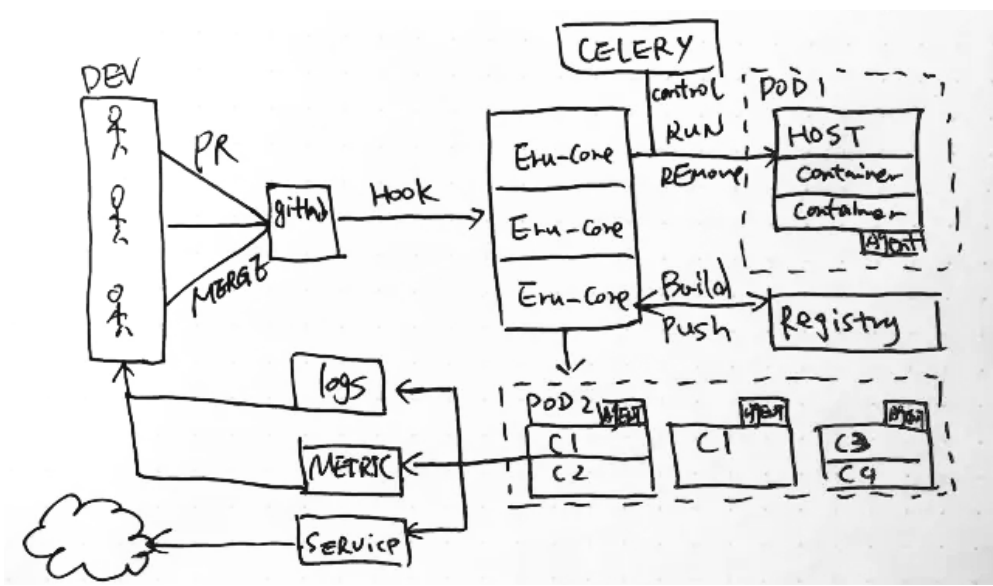
于是在挫折的 2014 年底，我们重新回顾了一遍 Borg 和 Omega 相关的信息，开始了第二代 NBE，也就是今天的主角——

Project Eru ——的开发。这一次我们抛弃了以前做一个 PaaS 的思路，而是决定去实现一个类似于 Borg 一样的服务编排和调度平台。

Project Eru

有了第一代 NBE 的开发经验，我们开发速度明显快了很多，第二周的时候就已经有了一个大体上能用的 demo。到目前为止，Eru 平台可以混编 Offline 和 Online 的服务 (binary/script)，对于资源尤其是 CPU 资源实现了自由维度 (0.1，0.01，0.001 等) 的弹性分配，使用 Redis 作为数据总线对外进行消息发布，动态感知集群所有的 Containers 状态并监控其各项数据等。基于 Docker 的 Image Layer 特性，我们把其和 Git version 结合起来，实现了自动化的 build/test 流程，统一了线上部署环境。同时顺便解决了 Runtime 的污染问题，使得业务能快速的扩容/缩容。

然后，我们的架构变成了这个样子：



看上去变化不大，实际上里面的设计和反馈回路什么的和第一代已经完全不一样了。

业务层方面在逻辑上我们使用了类似于 Kubernetes 的 Pod 来描述一组资源，使得 Eru 有了 Container 的组资源控制的能力。但是和 Kubernetes 不同的是，我们 Pod 仅仅是逻辑上的隔离，主要用于业务的区分，而实际的隔离则基于我们的网络层。对于 Dockefile，我们不允许业务方自行写 Dockerfile。通过标准化的 App.yaml 统一 Dockerfile 的生成，通用化的 Entrypoint 则满足了业务一份代码多个角色的复用和切换，使得任何业务几乎都可以完全无痛的迁移上来。

另外我不知道大家发现没有，之前第一代 NBE 是个完整的闭环，一个业务有生到死都有 NBE 本身各个组件的身影。但是在第二代 NBE 中我们放弃了以前考虑的完整闭环设计。之前实现的 NBE 第一代打通了项目整个生命周期的每一个环节，但实际上落地起来困难重重，并且使得 Dot (Master) 的状态太重没法 Scale Out，因为是它是单点部署，可靠性上会糟糕一些。所以 Eru 中每一个 Core 都是一个完整的无状态的逻辑核心，使得其可以 Scale Out 的同时可靠性上也比 NBE 第一代要健壮得多。所以第三幅图你们可以看到，我画了好多个 Eru-Core 来表明它是个可以 Scale out 的幺蛾子。

因此在这个体系下，我们推荐业务根据自身业务特性，通过监控自身数据，订阅 Eru 广播，调用 Eru-Core 的 API，实现复杂的自

定义的部署扩容等操作。我们并不会去强行干涉或者建立一系列规则去限定这些事情。这也是它不属于 PaaS 的原因。

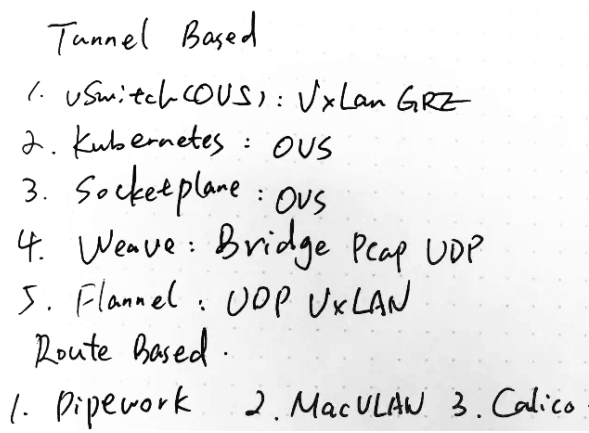
细节

大体介绍完这个项目之后，我来说说实现细节。主要有几个方面，首先是项目部署结构总体技术选择，然后是网络、存储、资源分配、服务发现等等。

首先 Eru 主要分 Core 和 Agent 两个部分。Agent 和 Core 并没有很强的耦合，通过 Redis 来交互信息（依赖于我们自己的 Redis Cluster 集群技术），主要用来汇报本机 Containers 情况和做一些系统层面的操作（比如增加减少 veth）。Core 则是刚才所说无状态的逻辑核心，控制所有的 Docker Daemon 并且和 Agent 进行控制上的交互。

容器内存储上，我们目前大部分使用了 Devicemapper 小部分是 Overlay，因此我们有的 Docker Host 上使用了内核 3.19 的内核，并外挂了一个 MooseFS 作为容器间数据共享的卷。考虑到 Docker 本身大部分时间是版本越新越靠谱（但是幺蛾子的 1.4 是个杯具），因此基本上我们使用的都是最新版的 Docker。网络方面对比了若干个解决方案之后，在隧道类（Weave/OVS 等）和路由类（MacVLAN/Calico 等）中我们选择了后者中的 MacVLAN。

下面是我当时做的方案对比图：



网络

既然说到网络，那我就从网络开始说起，为啥当时我们会选择 MacVLAN 相对而言这个比较简单和冷门的方案?

相比于 Route 方案，Tunnel 方案灵活度会更高，但是会带来两个问题:

- 性能，比如 Weave，通过 UDP 封装数据包然后广播到其他跑着 Weaver 的 Host，封包解包的过程就会带来一些开销。另外大多数 OVS 方案性能其实都不太乐观，之前和某公司工程师交流过，大体上会影响 20%~30% 左右的吞吐性能。(所以他们都用上 FPGA 了……)
- Debug 困难。Tunnel 的灵活是构建在 Host 间隧道上的，物理网络的影响其实还没那么大，但是带来一个弊端就是如果现在出了问题，我怎样才能快速的定位是物理链路还是隧道本身自己的问题。

而 Route 方案也会有自己的问题:

- Hook，Route 方案需要 Container Host 上有高权限进程去 Hook 系统 API 做一些事情。
- 依赖于物理链路，因此在公有云上开辟新子网做 Private SDN 使得同类 Containers 二层隔离就不可能了。
- 如果是基于 BGP 的 Calico，那么生效时间差也可能带来 Container 应用同步上的一些问题。

所以最后我们选择了 MacVLAN，一来考虑到我们组人少事多，隧道类方案规模大了之后 Debug 始终是一件比较麻烦的事情，二来 MacVLAN 从理解上和逻辑上算是目前最简单的一个方案了。

使用这种方案后，我们可以很容易在二层做 QoS，按照 IP 控流等，这样避免了使用 tc（或者修改内核加强 tc）去做这么一件事件，毕竟改了内核你总得维护对吧。因为是完全独立的网络栈，性能上也比 Weave 等方案表现得好太多，当然还有二层隔离带来的安全性。

某种意义上 MacVLAN 对 Container 耦合最小，但是同时对物理链路耦合最大。在混合云上，无论是 AWS 也好还是青云亦或者微软的 Azure，对二层隔离的亲和度不高，主要表现在不支持自定义子网上。因此选取这个方案后，在混合云上是没法用的。所以目前我们也支持使用 Host 模式，使得容器可以直接在云上部署，不过这样一来在云上灵活度就没那么高了。

存储

容器内存储我们目前对这个需求不是太高，小部分选取 Overlay 主要是为了我们 Redis Cluster 集群方案上 Eru 之后 Redis 的 AOF 模式需要，目前来看情况良好。考虑 Overlay 也是因为来自百度的一哥们告诉我们这货是 Container 杀人越货必备，尤其是小文件，所以我们的另外一个工程师也做了一个测试。

```
overlayfs 20g文件测试
dd if=/dev/zero of=/root/20Gb.file bs=1024 count=20000000

20000000+0 records in
20000000+0 records out
20480000000 bytes (20 GB) copied, 53.1718 s, 385 MB/s

device_mapper 20g文件读写

dd if=/dev/zero of=/root/20Gb.file bs=1024 count=20000000

20000000+0 records in
20000000+0 records out
20480000000 bytes (20 GB) copied, 69.7867 s, 293 MB/s
```

在 Devicemapper 和 Overlay 的性能对比上，大量小文件持续写 Overlay 的性能要高不少。然后我们就小部分上 Overlay 了。对于我们现有的 Redis Cluster 集群，我们采用内存分割的方式部署 Container。一个 Container 内部的 Redis 限制在 Host 总内存数 / Container 数这么大。举个例子，我们给 Redis 的 CPU

分配为 0.5 个，一台机器 24 Core 可以部署 48 个 Container，而我们的 Host 申请下来的一般只有 64G，因此基本上就是 1G 左右一个 Redis Container 了。

这样会有2个好处:

- AOF 卡顿问题得到缓解。
- 数据量或者文件碎片量远远达不到容器内存储的性能上限，意外情况可控。

在这个量级下，其实 DM 和 Overlay 还是差不多的

当然如果 instance 内存上限提高了，那么 Overlay 的优势就会很明显了。

Scale

扩容和缩容，我们更加希望是业务方去定制这个组件去做。我们所有的容器基础监控数据均存储在了 influxdb 上面，虽然现在来看它不是蛮靠谱 (研究 Open-falcon 中)。同时业务方也可以写自己想监控的一些数据按照自己喜欢的姿势到任何地方，然后通过读取这些数据，判断并决策，最后调取 Core 的 API 去干扩容和缩容。我们在第一代 NBE 中是我们自己定义了一套扩容缩容的规则，主要是效果不好，业务有时候要监控的并不是什么 CPU、MEM、IO，可能就是某个请求的耗时来决定是否扩容缩容。

因此 Eru 中，我们完全放开了权限。

核心宗旨就是“谁关心，谁做”。

资源分配和集群调度

资源的分配和集群调度，我们在第二代 NBE 中采取了以CPU 为主，MEM 半人工审核机制。磁盘 IO 暂时没有加入到 Eru 豪华午间套餐，而流量控制交给了二层控制器。之所以这么选择主要是因为考虑到一个机房建设成本的时候，CPU 的成本是比较高的，因此以 CPU 为主要调度维度。在 QCon 上和腾讯的讲师聊过之后，关于 CPU 的利用率上我们也实现了掰开几分用这么个需求，当然，我们是可以按照 Pod 来设置的，我们不会局限于 0.1 这个粒度，0.01 或者 0.5 都可以。但是内存方面明显我们的研发力量表示改内核并自行维护还不如在 520 陪女朋友看电影，所以我们没有实现腾讯讲师说的 softlimit subsystem 。主要还是通过数据判断 Host 内存余量和在上面 Container 内存使用量/申明量对比来做旁路 OOM Kill。

以 CPU 为主维度的来调度上，我们把应用申请 CPU 的数目计算为两类，一类称之为独占核，一类为碎片核。一个 Container 有且仅有一个碎片核，比如申请为 3.2 个 CPU 的话 (假定一个 Cpu 分为 10 份)，我们会通过 CpuSet 参数设定 4 个核给这个 Container，然后统一设定 CpuShare 为 1024*2。其中一个核会跟其他 5个 Container 共享，实现 Cpu 资源的弹性。

目前我们暂不考虑容器均匀部署这么个需求，因为我们对应的都是一次调度几台甚至几十台机器的情况，单点问题并不严重。

其实主要还是懒……

所以在应用上线时，会经过这些步骤：

- 申报内存最大使用量和单个 Container CPU 需求。比如 1G 1.3 个 CPU。
- 请求在那个 Pod 上部署（权限验证）。
- Core 计算 Pod 中资源是否满足上线需求（CPU 申明 * 上线数目，当然这其中算法就复杂了，并不是一个简单乘的关系）。
- 足够的话开始锁定 CPU 资源，调度 Host 上的 Docker Daemon 开始部署。

这个 By Core 的模型当然也会带来一些浪费，比如对一些不重要的业务。

因此我们加入了一个 Public Server 的机制，不对机器的 CPU 等资源做绑定，只从宏观的 Host 资源方面做监控和限制。使得 Eru 本身可以对服务进行降级操作，目前我们主要用 Public Server 来跑单元测试和镜像打包。

服务发现和安全

上线完容器后，那么接下来要考虑的就是服务发现和安全问题了，我们把控制权交给了业务方和运维部门。一般情况下同类

Container 将会在同一个子网之中（就是依靠 SDN 的网络二层隔离，一组 Container 理论上都会在一个或者多个同类子网中），调用者接入子网即可调用这些 Container。同时我们也把防火墙策略放到了二层上，保证其入口流量安全。因而整体上，对于业务部门而言，服务基本上说一个完整的黑箱（组件），他们并不需要关心服务的部署细节和分布情况，他们看到的是一组 IP（当然使用内网 DNS 的话会更加透明），同一子网内才有访问权限，直接调用就完了。我们认为一个自建子网内部是安全的。

另外我们基于 Dnscache 和 Skydns 构建了可以实时生效的内网 DNS 体系，分别部署在了我们现有的三个机房里面。业务方可以自行定义域名用来描述这个服务（其实也是 Eru App），完全不需要关心服务背后的物理链路物理机器等，实现了线上的大和谐。

举个例子

目前我们 Redis Cluster 有 400 个 instance，10 个集群，按照传统方式部署。每一次业务需求到我们这边之后我们需要针对业务需求调配服务器，初始化安装环境，并做 instance 部署的操作。在我们完成 Redis instance Dockerize 之后，Redis Cluster Administrator 只需要调取 API biu 一个最小集群，交付子网入口 IP 即可（就是我们的 Proxy 地址）。遇到容量不足则会有对应的 Redis Monitor 来自动调用 Eru API 扩容，如果过于清闲也能非常方便的去缩容。现在已经实现了秒级可靠的 Redis 服务响应和支撑。

此外我们另一个服务也打算基于这一套平台来解决自动扩容问题。通过 Eru 的 Broadcasting 机制结合 Openresty 的 lua 脚本动态的更新服务的 Upstream 列表，从而使得我们这样平时 500 QPS 峰值 150K QPS 的业务不再需要预热和准备工作，实现了无人值守。

总结

总的来说，我们整个 Docker 调度和编排平台项目 Eru 的设计思路是以组合为主，依托于现有的 Redis 解决方案，通过“消息”把各个组件串了起来，从而使得整个平台的扩展性和自由度达到我们的需求。除了一些特定的方法，比如构建 Image，其他的诸如构建 Dockerfile，如何启动应用等，我们均不做强一致性的范式去规范业务方 / 服务方怎么去做，当然这和我们公司本身体系架构有关，主要还是为了减少落地成本。毕竟不是每个公司业务线都有能力和眼界能接受和跟上。

最后我们现在主要在搞 Redis instance Dockerize 这么一件事，又在尝试把大数据组 Yarn task executor docker 化。在这个过程中我们搞定了 sysctl 的参数生效，容器内权限管理等问题，那又是另外一个故事了……我们计划今年年末之前，业务 / 服务 / 离线计算 3 个方向，都会开始通过 Eru 这个项目构建的平台开始 Docker 化调度和部署，并对基于此实现一个 PaaS。

P.S. 所有代码均公开在了 github.com/HunanTV 里面，欢迎大家围观。■

Q&A

Q1：首先是已有平台迁移到Docker过程中，有没有遇到过阻力，应用适配成本高么，能否分享下典型问题和阻力？

记得我说过 NBE 第一代就是因为完整闭环落地困难所以我们才决定做 Eru 的吧。我们是在14年开始做这件事的，当时对 Docker 的看法其实还不是很成熟。一方面业务方有很多的顾虑，另一方面我们是 AWS 大客户了，为毛要用 Docker，直接虚机起啊。

然后呢，NBE 第一代是一个纯种 PaaS 去做的，那么自然有强制性的范式和规则要求业务方遵守，其中我这边因为 DAE 带来的一些经验，比较推崇与服务拆分和微服务化。但业务方觉得我流量现在都快撑不住了，还拆分，所以很多那种一份代码按照启动命令不同来实现线上角色的切换。其实整体上 NBE 第一代只需要增加一个 App.yaml 就能使得大部分业务直接 Docker 化。

但是就是因为我们两方在这个业务代码角色和拆分上有重大矛盾，导致落地阻力非常大。以至于我本组的人都直接参与业务组开发里面去了。最后总结到这写现状之后，我们得出这么几个结论：

- 尽可能的先满足业务，再推动重构（Eru 支持一份代码多个角色）。
- 尽可能的降低自身平台耦合，服务发现安全交给上层去做（Eru 的消息广播机制）。
- 对于新技术的落地，不要先从制定规范开始做起，尽可能的猥琐的勾引他们落地后再去制定规范（目前平台架构部基于 Eru 在做芒果TV 自己的 PaaS）。

Q2：Docker平台上有没有推荐的监控系统，监控数据存储中InfluxDB上面，有没有遇到过坑？

cAdvisor 是 Google 出的一个监控 Agent，我们瞄了一眼代码后决定写到自己的 Agent 中整合进去。说到这个其实技术细节和有意思的事情就多了，两周前我还在看 libcontainer 的代码。

首先，cAdvisor 早期的实现里面，是通过 libcontainer 中的一个 struct 读取的 container 基础信息。但是，libcontainer 经过一次重构之后，那个 struct 没了……所以 cAdvisor 自行维护了一个版本的 libcontainer。

之前，我翻代码之后，认为 libcontainer.cgroups/fs 下面的 Manager 可以来做这件事件。但是会出问题，如果直接调用 libcontainer/cgourps/fs 下的 Manager 的话，会产生一个新的 cgroups profile，覆盖掉通过 Docker api 启动 container 的配置，导致某些设备不可读，举个例子 /dev/urandom 变为不可读状态，影响某些语言中的 random 包。

所以，目前我们自己的实现是直接通过 libcontainer 载入 /var/lib/docker/execute/native 这个目录，直接读取容器信息后找到这些 cgroups 状态文件来生成 influxdb 所需要的数据。

说到 InfluxDB 我觉得可以开一个吐槽大会了……InfluxDB 0.88 目前可以找到的最后一个 stable 版本，集群功能基本是废的，同时会有内存泄露和丢数据的问题……而且我们还修过他们 API 的

一些边界条件问题，但是发到 InfluxDB 组之后他们要我们不要管了，直接等 0.9。然而……InfluxDB 的官网本来写的是 0.9 stable 将会在 4 月发布……现在已经到了 coming soon，同时 master 里面的 rc 已经到了 35 还是多少来着了……对于 0.9 rc 而言，数据格式聚合函数和 0.88 已经有了很大的不同，举个例子，derivative 这个聚合函数也就是前几天才合入的。并且 rc 之间的落到磁盘数据并不通用，因此我们经常遇到升级后 influxdb 无法启动的问题，只好清理数据。

目前因为实现问题，我们在暂时还是用 InfluxDB 但只保证一周数据有效性（其实如果不升级可以保持很久）。同时我们在考虑第二方案，就是我前同事 laiwei 来总团队在小米大规模铺的 open-falcon 方案。

Q3：申请碎片核，对性能是否有影响相对于整数核？

对于这个问题，其实 cgroups 是这样的一个逻辑；假设一个 container 有 3 个 CPU，其中一个共享核设定了 20% 的使用量，那么在共享核，我们假设是 0 号核吧，没有其他 container 绑定的时候，这个 container 可以近似的看成绑定了 3 个核。它可以吃满 0 号核 100% 的用量，但是一旦 0 号还让其他 4 个 container 绑定后，只要在高峰期，那么它只能最多吃满 20%。等于就是说对于这个 container 而言，0 号核的资源是弹性的，最少能保证 20%。有一篇测试的文章，我给翻一下 https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/。

Q4：选择aws而非阿里，可否分享一下原因？

其实我不是运维部的，这个问题其实没那么简单，我们在役的公有云有2家：AWS 和 Azure。接洽过的有2家，QingCloud 和阿里云。选择 AWS，其实最大一个原因公司高层的看法是要做中国的 Netflix。其实这很好理解，因此比较看重未来对海外的扩张。另外一点是综合性能和价格等多个维度上，EC2 是一个比较不错的选择，前提是对于大客户。

Q5：Redis自动扩容，缩容，依据什么来判定？是业务方来控制，还是调度系统自己判定？

Redis 本身会提供很多监控信息，通过 info 命令即可，我们的 proxy 也实现了类似的原语，redis ctl daemon 通过它将整合后的集群信息发送到了 influxdb。monitor 会通过这些信息来决定是否调用 core 的 api 上新的 redis instance 并在部署完毕后通知 redis-ctl 把 instance 配置好加入到集群中。举个简单的例子：

info 可以拿到 instance 现在使用内存量和cpu使用率，那么集群所有 instance 使用内存量接近设定上限（现在是1G/500M一个）的时候，monitor 就会开始做这件事情。

业务方对我们集群没多大的控制权，他们只需要提出申请即可。

Q6：可以稍微介绍下芒果TV的业务么，对芒果台很熟，但对你们这个平台支撑的业务不清楚。

芒果TV 现在主要覆盖了3条线。

第一条线是湖南本地的 IPTV 内容，俗称 OTT 线，这一条线和传统互联网企业不一样的在于，拥有XXXXW真金白银的付费用户。

第二条线是内容产出，作为广电旗下的一个互联网企业，依托于自己的资源（广电的资源）。

第三条线就是网站线，因为独播策略内容为王，我们现在主要承担湖南卫视所有节目的互联网渠道转播工作。同时，我们另辟蹊径，在互联网直播技术上应该也是往前走了一大步，比如前几天的billboard，全球同步直播。比如跨年晚会，五机位自由选择视角互联网直播等。

Q7：启用Docker后，原有AWS的EC2主机有没有减少?

其实 EC2 主机减少和启用 Docker 没太多关系。我们目前平台大多数是在我们自己的机房中的。我们 EC2 也真的减少了，但是相信我，能做到这样绝对不是架构的问题，而是程序问题。至于有没有估算过资源利用率提升了多少，从 Redis 集群利用率来看，不太好估算。因为什么呢，以前是业务方说我需要15w QPS，那我们就按照 Redis Cluster 性能曲线开了一堆 instance 去支撑这个业务。

说回这个 Redis，那么业务方实际上能达到多少呢？即便是跨年演唱会，我们的峰值也没过9w QPS，等于就是说大多数 redis instance 是被浪费掉的。我们采取 Docker 之后，资源利用率是上升了的，具体多少就没估算了。对于上面那个例子，我们会采取给予一个基础性能的集群，让其自动扩容，满足业务需求，而不

是一开始就给一个能撑住 15W QPS 的集群。那么多出来的资源，就给其他业务了。毕竟预先估算和实际还是有差距的。

Q8：如何做到应用和配置分离的，同一份代码，在测试和产品上用不同的配置？

我们现在没做 UI，不过大家可以试想一下一个应用部署页面，应用需要选择运行时资源，网络，CPU/MEM（就像一个滑动条在拖动），以及启动的入口命令。选择的资源将会以 ENV 的形式写入到 container 中，所以是一种组合搭配的节奏，项目并不会去指定资源，而资源当然也是可以自定义的，就像买车，你可以定制座椅，轮毂，电子装置一样。

Q9：如果换云平台，比如从 AWS 到第三方，Eru 需要多大改造？

基本不需要，Eru 支持 host 模型，只不过就得在业务层去控制端口冲突了，不过好在目前我们上云的业务还比较纯粹，不会出现混排的需求。不过未来在这一块，我们有这样的计划：

- push 云支持自建子网（但不跟机器绑定），Azure 表示甲方（其实我们才是乙方）您出钱我出命。
- Calico 这个方案和 MacVLAN 同时支持，这个就等我前领导 @洪强宁@宜信 那边的实现了。

Docker实战

微博基于Docker容器的混合云迁移实战

作者 / 陈飞

微博研发中心技术经理及高级技术专家。2008年毕业于北京邮电大学，硕士学位。毕业后就职华为北研。2012年加入新浪微博，负责微博Feed、用户关系和微博容器化相关项目，致力于Docker技术在生产环境中的规模化应用。2015年3月，曾在QClub北京Docker专场分享《大规模Docker集群助力微博迎接春晚峰值挑战》。

为什么要采用混合云的架构

在过去很长的时间内，大部分稍大些的互联网系统包括微博都是基于私有体系的架构，可以在某种程度理解成私有云系统。混合云，顾名思义就是指业务同时部署在私有云和公有云，并且支持在云之间切换。实际上“为什么要采用混合云”这个问题，就等于“为什么要上公有云”。我们的考虑点主要有四个方面

业务场景

随着微博活跃度的提升，以及push常规化等运营刺激，业务应对短时极端峰值挑战，主要体现在两个方面：

- 时间短，业务需要应对分钟级或者小时级。
- 高峰值，例如话题经常出现10到20倍流量增长。

成本优势

对于短期峰值应对，常规部署，离线计算几个场景，我们根据往年经验进行成本对比发现公有云优势非常明显。

场景	私有云方案	公有云方案	预估总成本对比（私/公）
短期峰值应对	部分调度，部分采购	按小时付费	数十倍
日常常规部署	按月摊销	包月付费	1.0~1.2
离线计算	按月摊销	按小时付费	数十倍

效率优势

公有云可以实现5分钟千级别节点的弹性调度能力，对比我们目前的私有云5分钟百级别节点的调度能力，也具有明显优势。

业界趋势

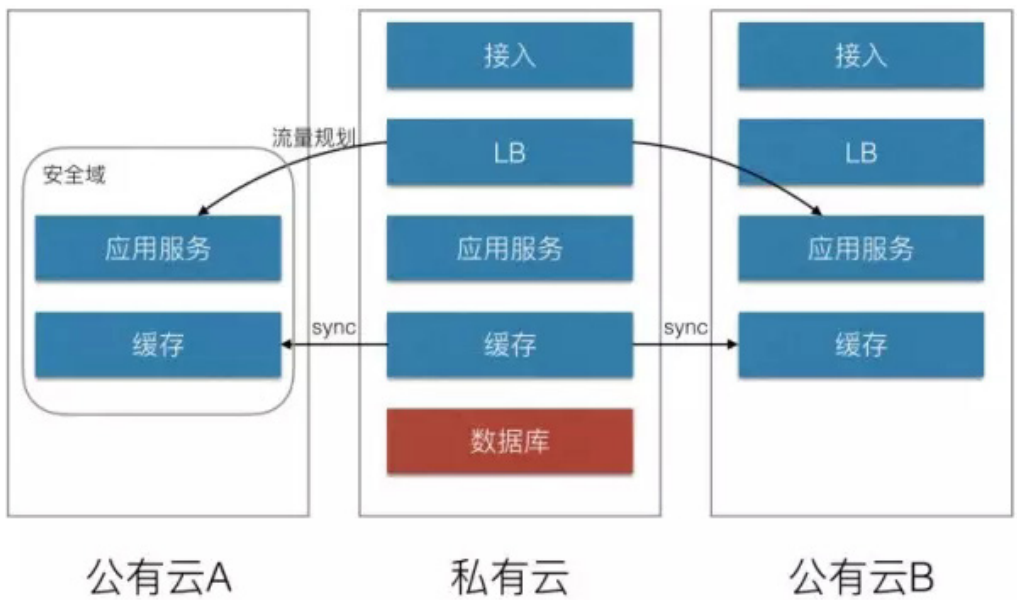
“Amazon首次公布AWS业绩：2014年收入51.6亿美元，2015年1季度AWS收入15.7亿美元，年增速超40%。”“阿里巴巴旗下云计算业务阿里云营收6.49亿元，比去年同期增长128%，超越亚马逊和微软的云计算业务增速，成为全球增速最快的云计算服务商。”

我们预计未来产品技术架构都会面临上云的问题，只是时间早晚问题。

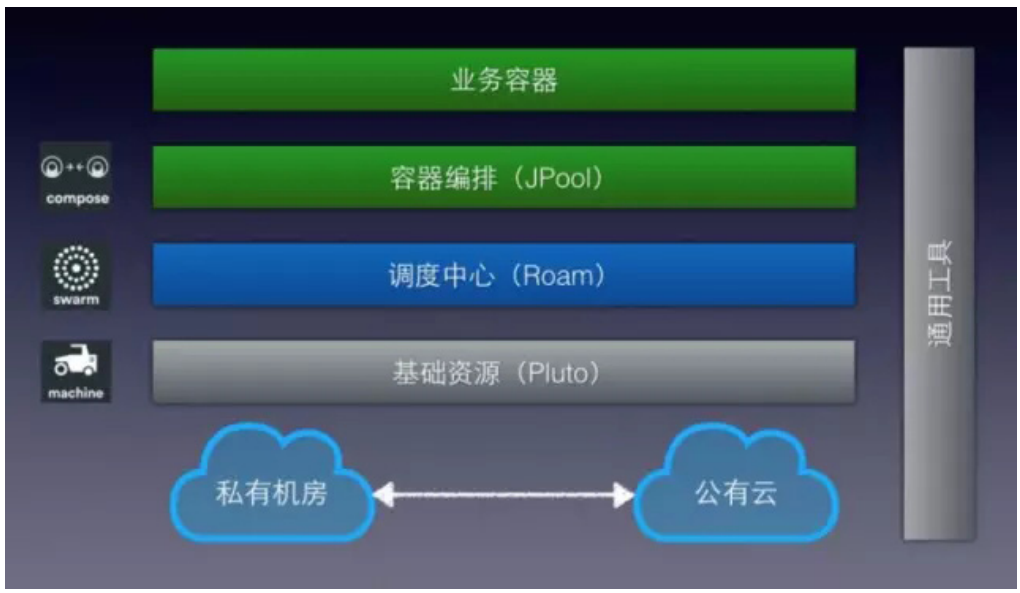
安全性

基于数据安全的考虑，我们现阶段只会把计算和缓存节点上云，核心数据还放在私有云；另外考虑到公有云的技术成熟度，需要支持在多个云服务直接进行业务灵活迁移。

基于上述几点考虑，我们今年尝试了以私有云为主，公有云为辅的混合云架构。核心业务的峰值压力，会在公有云上实现，业务部署形态可参考下图：



下面介绍介绍技术实现。整体技术上采用的是Docker Machine + Swarm + Consul的框架。系统分层如下图:

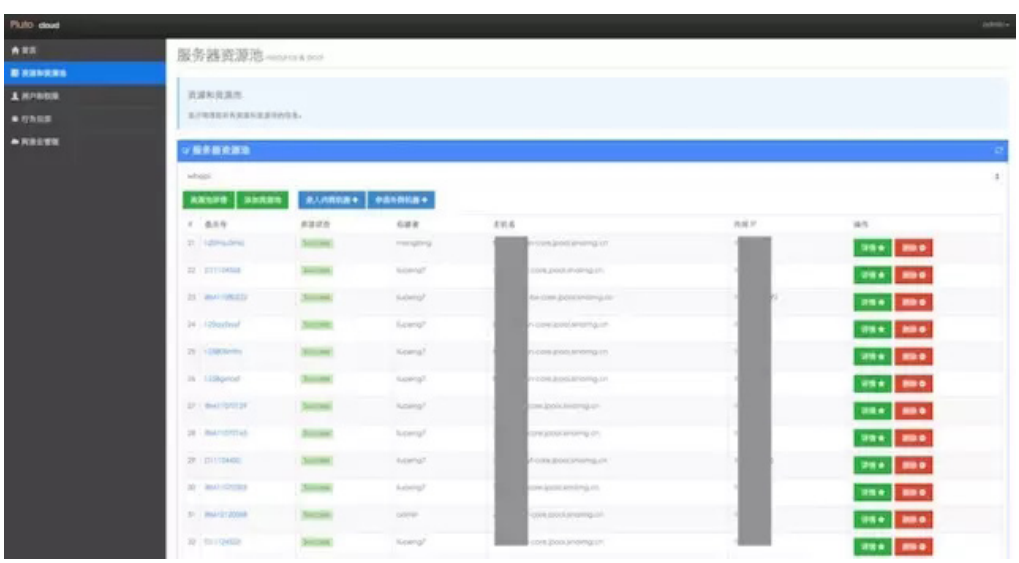


跨云的资源管理与调度

跨云的资源管理与调度（即上图中的pluto部分）的作用是隔离云服务差异，对上游交付统一的Docker运行环境，支持快速弹性扩缩容及跨云调度。功能上主要包括：

- 系统初始化
- 元数据管理
- 镜像服务
- 网络
- 云服务选型
- 命令行工具
- 其他

系统界面如下图：



系统初始化

最初技术选型时认为Machine比较理想，仅需要SSH通道，不依赖任何预装的Agent。我们也几乎与阿里云官方同时向Docker Machine社区提交了Driver的PR，然后就掉进了大坑中不能自拔。

例举几个坑：

- 无法扩展，Machine的golang函数几乎都是小写，即内部实现，无法调用其API进行功能扩展。
- 不支持并发，并发创建只能通过启动多个Machine进程的方式，数量多了无法承载。
- 不支持自定义，Machine启动Docker Daemon是在代码中写死的，要定义Daemon的参数就非常ugly。

目前我们采用的是Puppet的方案，采用去Master的结构。配置在GitLab上管理，变更会通过CI推送到pluto系统，然后再推送到各实例进行升级。在基础资源层面，我们目前正在进行大范围基础环境从CentOS 6.5升级到CentOS 7的工作。做这个升级的主要原因是由于上层基于调度系统依赖Docker新版本，而新版本在CentOS 6.5上会引发cgroup的bug，导致Kernel Panic问题。

元数据的管理

调度算法需要根据每个实例的情况进行资源筛选，这部分信息目前是通过Docker Daemon的Label实现的，这样做的好处是资源和调度可以Docker Daemon解耦。例如我们会在Daemon上记录

这个实例的归属信息：

```
—label idc=$provider #记录云服务提供商
—label ip=$eth0 #记录ip信息
—label srv=$srv #记录所属业务
—label role=ci/test/production…
…
```

目前Docker Daemon最大的硬伤是任何元数据的改变都需要重启。所以我们计划把元数据从Daemon迁移到我们的系统中，同时尝试向社区反馈这个问题，比如动态修改Docker Daemon Label的接口（PR被拒，官方虽然也看到了问题，但是比较纠结是否要支持API方式），动态修改registry（PR被拒，安全因素），动态修改Docker Container Expose Port（开发中）。

镜像服务

为了提升基础资源扩缩容的效率，我们正在构建虚机镜像服务。参考Dockerfile的思路，通过描述文件定义定义虚机的配置，支持继承关系，和简单的初始化指令。通过预先创建好的镜像进行扩缩容，可以节省大约50%的初始化时间。

描述文件示意如下：

```
centos 7.0:
- dns: 8.8.8.8
- docker:
- version: 1.6
- net: host
```

```
meta:
- service: $SRV
puppet:
- git: git.intra.weibo.com/docker/puppet/tags/$VERSION
entrypoint:
- init.sh
```

不过虚机镜像也有一些坑，例如一些云服务会在启动后自行修改一部分配置，例如router, dns, ntp, yum等配置。这就是上面entrypoint的由来，部分配置工作需要在实例运行后进行。

网络

网络的互联互通对业务来说非常关键，通常来说有三种方案：公网，VPC+VPN，VPC+专线。公网因为性能完全不可控，而且云服务通常是按照出带宽收费，所以比较适合相互通信较少的业务场景。VPC+VPN实际链路也是通过公网，区别是安全性更好，而且可以按照私有云的IP段进行网络规划。专线性能最好，但是价钱也比较好，且受运营商政策影响的风险较大。

网络上需要注意的是包转发能力，即每秒可以收发多少个数据包。一些云服务实测只能达到10万的量级，而且与CPU核数、内存无关。也就是说你花再多的钱，转发能力也上不去。猜测是云厂商出于安全考虑在虚机层面做了硬性限制。我们在这上面也中过枪，比如像Redis主从不同步的问题等等。建议对于QPS压力比较重的实例进行拆分。

云服务选型

我们主要使用的是虚机和软负载两种云服务。因为微博对缓存服务已经构建一套高可用架构，所以我们没有使用公有云的缓存服务，而是在虚机中直接部署缓存容器。

虚机的选型我们重点关注CPU核数，内存，磁盘几个方面。CPU调度能力，我们测试总结公有云是私有云的1.2倍，即假设业务在私有云上需要6个核，在公有云上则需要8个。

内存写入速度和带宽都不是问题，我们测试发现甚至还好于私有云，MEMCPY的带宽是私有云的1.2倍，MCBLOCK是1.7倍。所以内存主要考虑的是价钱问题。

磁盘的性能也表现较好，顺序读写带宽公有云是私有云的1.4倍，随机写是1.6倍。唯一要注意的是对于Redis这种业务，需要使用I/O优化型的虚机。

以上数据仅供参考，毕竟各家情况不一样，我们使用的性能测试工具：sysbench, mbw, fio，可自行测试。

CLI客户端（命令行工具）

为了伺候好工程师们，我们实现了简单的命令行客户端。主要功能是支持创建Docker容器（公有云或私有云），支持类SSH登陆。因为我们要求容器本身都不提供SSH（安全考虑），所以我们是用Ruby通过模拟docker client的exec命令实现的，效果如下图：

```
➜  bin git:(master) ✗ ./pluto
usage: dck [new | login | delete]
➜  bin git:(master) ✗ time ./pluto new
succeed
./pluto new  0.16s user 0.03s system 58% cpu 0.331 total
➜  bin git:(master) ✗ ./pluto login
login container successfully
root@b9713222124c:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.2 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.2 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@b9713222124c:/# exit
logout container successfully
➜  bin git:(master) ✗ time ./pluto delete
succeed
./pluto delete  0.15s user 0.03s system 67% cpu 0.260 total
➜  bin git:(master) ✗
```

其他方面

跨域的资源管理和调度还有很多技术环节需要处理，比如安全，基础设施（DNS、YUM等），成本核算，权限认证，由于这部分通用性不强，这里就不展开了。

容器的编排与服务发现

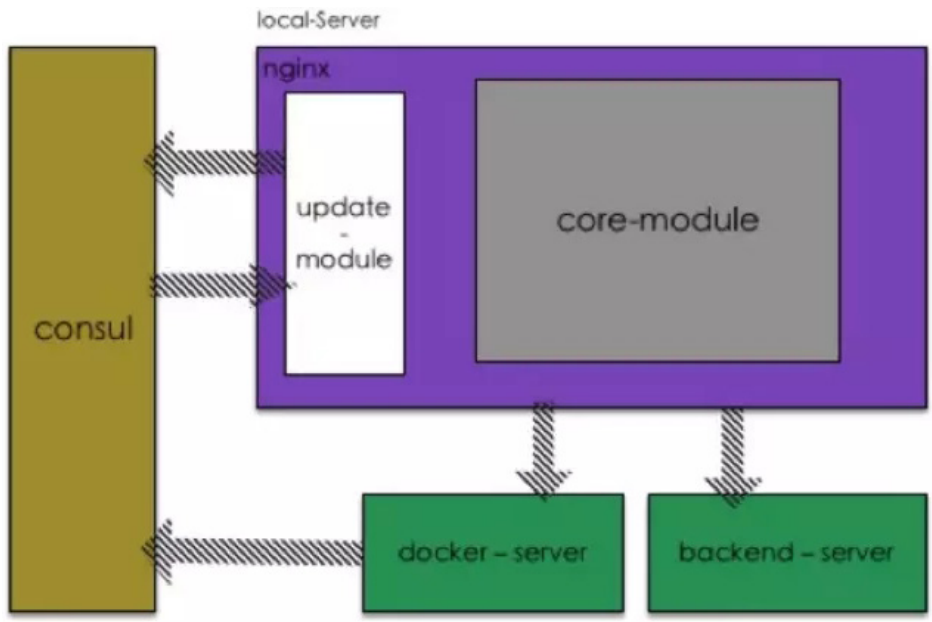
提到调度就离不开发现，Swarm是基于Consul来做节点发现的。Consul采用raft协议来保证server之间的数据一致性，采用gossip协议管理成员和传播消息，支持多数据中心。Consul集群中的所有请求都会重定向到server，对于非leader的server会将写请求转发给leader处理。

Consul对于读请求支持3种一致性模式：

- default：给leader一个time window，在这个时间内可能出现两个leader（split-brain情况下），旧的leader仍然支持读的请求，会出现不一致的情况。
- consistent：完全一致，在处理读请求之前必须经过大多数的follower确认leader的合法性，因此会多一次round trip。
- stale：允许所有的server支持读请求，不管它是否是leader。

我们采用的是default模式。

除了Swarm是基于Consul来做发现，业务直接也是通过Consul来做发现。我们服务采用Nginx来做负载均衡，开源社区的方案例如Consul Template，都需要进行reload操作，而reload过程中会造成大量的失败请求。我们现在基于Consul实现了一个nginx-upsync-module。



Nginx仅需在upstream配置中声明Consul集群即可完成后端服务的动态发现。

```
upstream test {
# fake server otherwise ngx_http_upstream will report error when startup
server 127.0.0.1:11111;
# all backend server will pull from consul when startup and will delete fake
server
consul 127.0.0.1:8500/v1/kv/upstreams/test update_timeout=6m update_
interval=500ms strong_dependency=off;
upstream_conf_path /usr/local/nginx/conf/upstreams/upstream_test.conf;
}
```

这个模块是用C语言实现的，效率已经经过线上验证，目前验证在20W QPS压力没有问题。而且这个模块代码已经开源在Github上，也欢迎大家提Issue：https://github.com/weibocom/nginx-upsync-module。

下图是我们做的几种方案的性能对比：

Environment	Qps	Total request	Timeout	Error
nginx(official)	71105	21323701	0	0
nginx-upsync	70023	21704181	0	0
consul-temp	1357	407253	0	135709

当然我们的RPC框架motan也会支持Consul，实现机制同样也是利用Consul的long polling机制，等待直到监听的X-Consul-Index位置发生变化，这个功能已经在我们内网验证通过，不过由于依赖整个motan框架，所以目前还没有开源。

Consul的监控

由于Consul处于系统核心位置，一旦出现问题会导致整体所有集群失联，所以我们对Consul做了一系列保障措施，其中所有Consul Server节点监控指标如下：

key	输出示例	含义（猜测）
consul.consul.fsm.register	Count: 2 Min: 0.195 Mean: 0.255 Max: 0.315 Stddev: 0.085 Sum: 0.511	
consul.consul.session_ttl.active	0.000	
consul.memberlist.gossip	Count: 70 Min: 0.003 Mean: 0.100 Max: 1.760 Stddev: 0.242 Sum: 6.981	
consul.memberlist.msg.alive	Count: 8 Sum: 8.000	
consul.memberlist.msg.dead	Count: 1 Sum: 1.000	
consul.memberlist.msg.suspect	Count: 1 Sum: 1.000	
consul.memberlist.probeNode	Count: 10 Min: 0.703 Mean: 1.407 Max: 2.605 Stddev: 0.621 Sum: 14.068	
consul.memberlist.pushPullNode	Count: 2 Min: 1.889 Mean: 2.542 Max: 3.196 Stddev: 0.924 Sum: 5.085	

consul.memberlist.tcp.accept	Count: 1 Sum: 1.000	
consul.memberlist.tcp.connect	Count: 2 Sum: 2.000	
consul.memberlist.tcp.sent	Count: 2 Min: 506.000 Mean: 561.500 Max: 617.000 Stddev: 78.489 Sum: 1123.000	
consul.memberlist.udp.received	Count: 19 Min: 20.000 Mean: 112.789 Max: 312.000 Stddev: 102.275 Sum: 2143.000	
consul.memberlist.udp.sent	Count: 36 Min: 20.000 Mean: 127.083 Max: 312.000 Stddev: 74.284 Sum: 4575.000	
consul.raft.fsm.apply	Count: 2 Min: 0.255 Mean: 0.308 Max: 0.360 Stddev: 0.074 Sum: 0.615	
consul.raft.rpc.appendEntries	Count: 200 Min: 0.003 Mean: 0.012 Max: 0.587 Stddev: 0.056 Sum: 2.327	
consul.raft.rpc.appendEntries.processLogs	Count: 2 Min: 0.004 Mean: 0.004 Max: 0.004 Stddev: 0.000 Sum: 0.007	
consul.raft.rpc.appendEntries.storeLogs	Count: 2 Min: 0.537 Mean: 0.553 Max: 0.569 Stddev: 0.023 Sum: 1.106	
consul.raft.rpc.processHeartbeat	Count: 65 Min: 0.007 Mean: 0.010 Max: 0.014 Stddev: 0.001 Sum: 0.672	
consul.runtime.alloc_bytes	3157640.000	
consul.runtime.free_count	9534843.000	
consul.runtime.total_gc_pause_ns	Count: 2 Min: 1039638.000 Mean: 1251945.000 Max: 1464252.000 Stddev: 300247.439 Sum: 2503890.000	
consul.runtime.heap_objects	9843.000	
consul.runtime.malloc_count	9544686.000	
consul.runtime.num_goroutines	64.000	
consul.runtime.sys_bytes	9836792.000	
consul.runtime.total_gc_pause_ns	1726423808.000	
consul.runtime.total_gc_runs	2094.000	
consul.serf.member.failed	Count: 1 Sum: 1.000	
consul.serf.member.join	Count: 2 Sum: 2.000	
consul.serf.queue.Event	Count: 20 Sum: 0.000	
consul.serf.queue.Intent	Count: 20 Sum: 0.000	
consul.serf.queue.Query	Count: 20 Sum: 0.000	
consul.serf.snapshot.appendLine	Count: 3 Min: 0.004 Mean: 0.010 Max: 0.019 Stddev: 0.008 Sum: 0.031	
consul.serf.snapshot.compact	Count: 1 Sum: 0.236	

Leader 节点监控指标如下图:

key	示例	含义（猜测）
consul.consul.catalog.register	Count: 2 Min: 1.549 Mean: 1.565 Max: 1.582 Stddev: 0.023 Sum: 3.130	
consul.consul.fsm.register	Count: 2 Min: 0.187 Mean: 0.226 Max: 0.264 Stddev: 0.054 Sum: 0.452	
consul.consul.leader.barrier	Count: 1 Sum: 1.541	
consul.consul.leader.reconcile	Count: 1 Sum: 2.047	
consul.consul.leader.reconcileMember	Count: 1 Sum: 1.762	
consul.consul.rpc.accept_conn	Count: 1 Sum: 1.000	
consul.consul.rpc.query	Count: 2 Sum: 2.000	
consul.consul.rpc.request	Count: 3 Sum: 3.000	
consul.consul.session_ttl.active	0.000	
consul.memberlist.gossip	Count: 70 Min: 0.002 Mean: 0.088 Max: 1.545 Stddev: 0.214 Sum: 6.186	
consul.memberlist.msg.alive	Count: 8 Sum: 8.000	
consul.memberlist.msg.dead	Count: 2 Sum: 2.000	
consul.memberlist.msg.suspect	Count: 2 Sum: 2.000	
consul.memberlist.probeNode	Count: 8 Min: 0.747 Mean: 1.409 Max: 1.926 Stddev: 0.533 Sum: 11.275	
consul.memberlist.pushPullNode	Count: 1 Sum: 3.572	
consul.memberlist.tcp.accept	Count: 1 Sum: 1.000	
consul.memberlist.tcp.connect	Count: 1 Sum: 1.000	
consul.memberlist.tcp.sent	Count: 2 Min: 575.000 Mean: 622.500 Max: 670.000 Stddev: 67.175 Sum: 1245.000	
consul.memberlist.udp.received	Count: 18 Min: 20.000 Mean: 107.833 Max: 285.000 Stddev: 82.970 Sum: 1941.000	
consul.memberlist.udp.sent	Count: 32 Min: 20.000 Mean: 132.281 Max: 312.000 Stddev: 91.603 Sum: 4233.000	
consul.raft.apply	Count: 2 Sum: 2.000	
consul.raft.barrier	Count: 1 Sum: 1.000	
consul.raft.commitTime	Count: 2 Min: 1.216 Mean: 1.241 Max: 1.267 Stddev: 0.036 Sum: 2.483	
consul.raft.fsm.apply	Count: 2 Min: 0.232 Mean: 0.269 Max: 0.307 Stddev: 0.054 Sum: 0.539	
consul.raft.leader.dispatchLog	Count: 2 Min: 0.459 Mean: 0.464 Max: 0.469 Stddev: 0.007 Sum: 0.929	
consul.raft.leader.lastContact	Count: 44 Min: 0.000 Mean: 29.295 Max: 76.000 Stddev: 20.504 Sum: 1289.000	

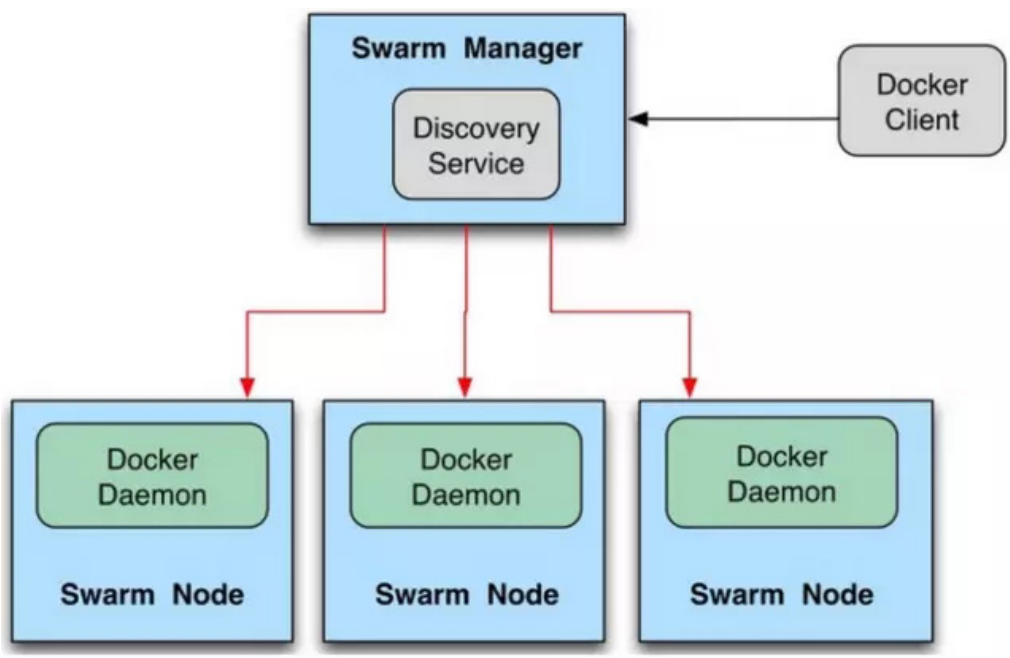
consul.raft.replication.appendEntries.logs.10.74.8.180:8300	Count: 135 Min: 0.000 Mean: 0.015 Max: 1.000 Stddev: 0.121 Sum: 2.000	
consul.raft.replication.appendEntries.logs.10.74.8.181:8300	Count: 135 Min: 0.000 Mean: 0.015 Max: 1.000 Stddev: 0.121 Sum: 2.000	
consul.raft.replication.appendEntries.rpc.10.74.8.180:8300	Count: 135 Min: 0.259 Mean: 0.400 Max: 0.938 Stddev: 0.066 Sum: 54.002	
consul.raft.replication.appendEntries.rpc.10.74.8.181:8300	Count: 135 Min: 0.347 Mean: 0.404 Max: 0.720 Stddev: 0.042 Sum: 54.589	
consul.raft.replication.heartbeat.10.74.8.180:8300	Count: 66 Min: 0.252 Mean: 0.379 Max: 0.481 Stddev: 0.029 Sum: 24.603	
consul.raft.replication.heartbeat.10.74.8.181:8300	Count: 66 Min: 0.338 Mean: 0.378 Max: 0.402 Stddev: 0.013 Sum: 24.918	
consul.runtime.alloc_bytes	3585448.000	
consul.runtime.free_count	100243648.000	
consul.runtime.gc_pause_ns	Count: 2 Min: 947504.000 Mean: 1117430.500 Max: 1287357.000 Stddev: 240312.361 Sum: 2234861.000	
consul.runtime.heap_objects	12448.000	
consul.runtime.malloc_count	100256096.000	
consul.runtime.num_goroutines	74.000	
consul.runtime.sys_bytes	10098936.000	
consul.runtime.total_gc_pause_ns	14956033024.000	
consul.runtime.total_gc_runs	16735.000	
consul.serf.member.failed	Count: 2 Sum: 2.000	
consul.serf.member.join	Count: 1 Sum: 1.000	
consul.serf.queue.Event	Count: 20 Sum: 0.000	
consul.serf.queue.Intent	Count: 20 Sum: 0.000	
consul.serf.queue.Query	Count: 20 Sum: 0.000	
consul.serf.snapshot.appendLine	Count: 1 Sum: 0.015	
consul.serf.snapshot.compact	Count: 1 Sum: 0.234	

Consul的坑

除了使用不当导致的问题之外，Consul Server节点通信通道UDP协议，偶发会出现server不停被摘除的现象，这个问题官方已在跟进，计划会增加TCP的通道保证消息的可靠性。

容器调度

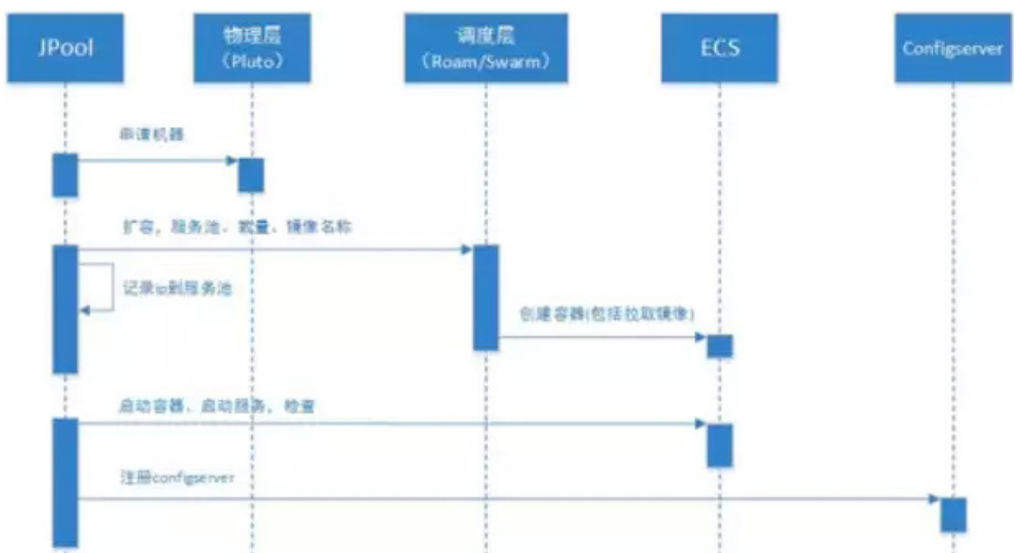
容器调度基于Swarm实现，依赖Consul来做节点发现（话说Swarm才刚刚宣布Production Ready）。容器调度分为三级，应用-应用池-应用实例，一个应用下有多个应用池，应用池可以按机房和用途等来划分。一个应用池下有多个Docker容器形式的应用实例。



我们利用Swarm的Filter机制，实现了适应业务的调度算法。整个调度过程分为两步：主机过滤：指定机房、内存、CPU、端口等条件，筛选出符合条件的主机集合；策略选择：对符合条件的主机集合进行打分，选择出最合适的主机，在其上创建容器以部署应用。调度子系统Roam实现了批量的容器调度与编排。

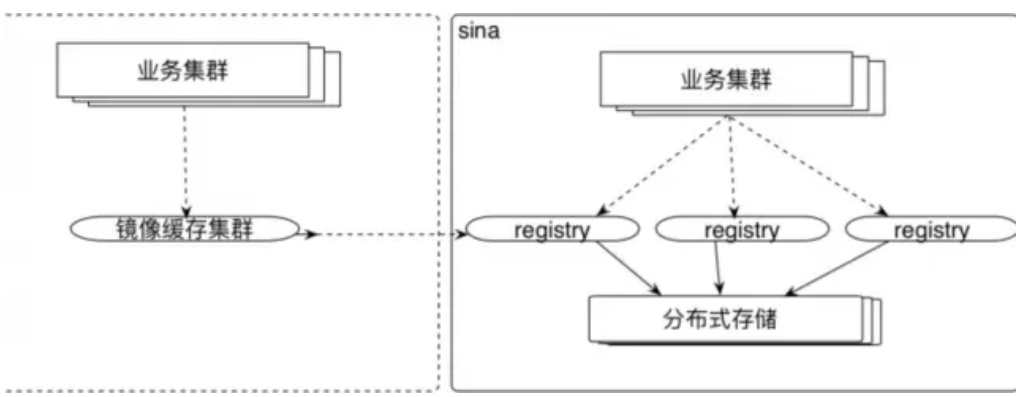
业务调度

容器调度是于业务无关，具体串联起资源管理，容器调度，发现等系统，完成业务容器最终跨云部署的是我们的JPool系统。JPool除了完成日常的业务容器上线发布之外，最重要的是完成动态扩缩容功能，使业务实现一键扩容、一键缩容，降低快速扩容成本。一次扩容操示意图如下：



围绕这调度和发现，需要很多工具的支撑。例如为了使业务接入更加方便，我们提供了自动打包工具，它包括代码打包、镜像打包的解决方案，支持svn、gitlab等代码仓库，业务仅需要在工程中定义pom.xml和Dockerfile即可实现一键打包代码，一键打包镜像，过程可控，接入简单。

我们还对Docker的Registry，我们进行了一些优化，主要是针对混合云跨机房场景，提供跨机房加速功能，整个服务架构如下：



混合云监控体系

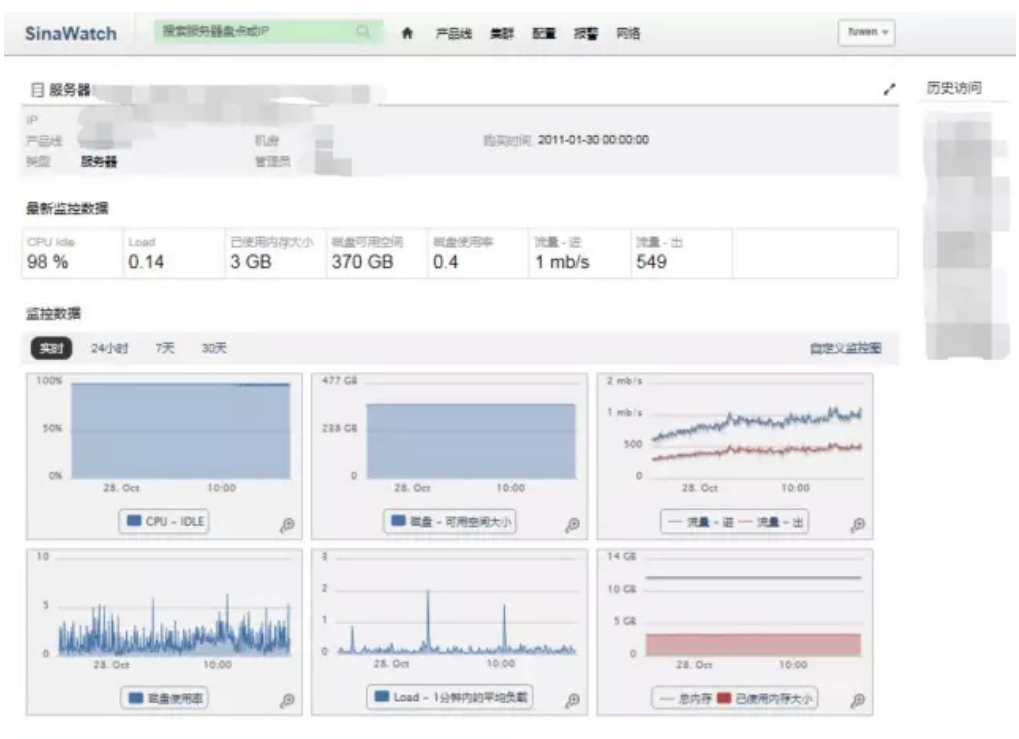
微博体系在经历了多年的IT建设过程后，已经初步建立了一套完整的信息化管理流程，极大地提升了微博业务能力。同时微博开展了大量的IT基础设施建设（包括网络、机房、服务器、存储设置、数据库、中间件及各业务应用系统等）。

针对于混合云体系，我们提供了一套完整的监控告警解决方案，实现对于云上IT基础架构的整体监控与预警机制，最大程度地保证了微博体系能够稳定地运行在混合云体系上，不间断地为用户提供优质的服务。监控告警解决方案实现了四个级别上的监控与预警：

- 系统级监控
- 业务级监控
- 资源级监控
- 专线网络监控

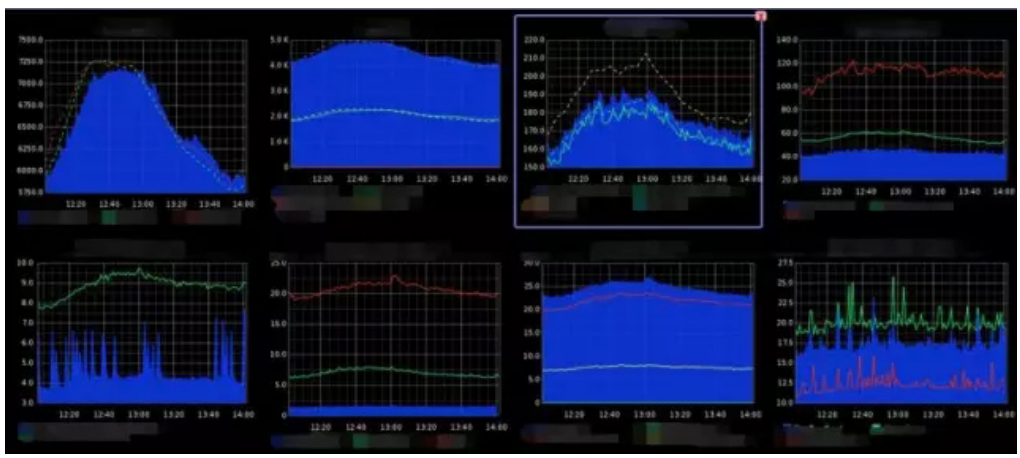
系统级监控

混合云体系支持的系统级监控（与新浪sinawatch系统的对接）包括：CPU，磁盘，网卡，IOPS，Load，内存。



业务监控

混合云体系集成了目前微博业务监控平台Graphite，自动提供了业务级别(SLA)的实时监控与告警。所有的配置与操作都是系统自动完成的，不需要用户进行额外的配置操作。



业务级别的监控包括：

- JVM监控：实时监控堆、栈使用信息、统计gc收集时间、检查JVM瓶颈等。
- 吞吐量监控：实时监控业务系统吞吐量(QPS或TPS)。
- 平均耗时监控：实时监控业务系统接口的平均耗时时间。
- 单机性能监控：实时监控单台服务器的各种业务指标。
- Slow监控：监控服务器集群，实时显示当前业务系统中最慢的性能瓶颈。

资源监控

混合云体系集成了目前微博资源监控平台sinadsp，自动提供了对各种底层资源的实时监控与告警。所有的配置与操作都是系统自动完成的，不需要用户进行额外的配置操作。

具体的监控指标包括：命中率，QPS/TPS，连接数，上行/下行带宽，CPU，内存。

前进路上遇到的那些坑

需要注意的坑，实际在各部分中都有提及。今天分享的主要内容就是这些了，当然业务上云，除了上面这些工作之外，还存在很多技术挑战要解决。比如跨云的消息总线，缓存数据同步，容量评估，流量调度，RPC 框架，微服务化等。■

Q&A

Q1：为什么选 Consul？看中有对比 Zookeeper、etcd，尤其是 etcd？

我们有对比 etcd 和 consul，主要还是看重了 consul 基于 K-V 之外额外功能，比如支持 DNS，支持 ACL。另外 etcd 不对 get request 做超时处理，Consul 对 blocking query 有超时机制。

Q2：上面提到的方案主要是 Java 体系，对于其他语言（php, nodejs, golang）体系系统的是否有很好的支持（开发、测试、发布部署、监控等）？

已经在开始 php 的支持。其实容器化之后，所有语言都是适用的。

Q3：Docker registry 底层存储用的是什么，怎么保障高可用的？

底层使用的 Ceph，前端无状态，通过 DNS 做跨云智能解析，加速下载。

Q4：Consul temple reload nginx时为什么会造成大量请求失败呢，不是graceful的吗？

是graceful的，在QPS压力较大的情况下，由于需要进行大量重连，过程中会产生较多失败请求。

Q5：Ceph IO情况怎么？高IO的是不是不太适合用Ceph？

针对Registry场景Ceph IO是可以胜任的。不过Ceph暂时还没有宣布Production Ready，所以对于极端业务场景，请谨慎。

使用开源Calico构建Docker多租户网络

作者 / 高永超 (flex)

宜信大数据创新中心云平台运维专家。目前专注于DevOps 和 PaaS 平台开发(基础设施方向)。曾在豆瓣担任过 SA Team Leader。《Pro Puppet》第一版中文译者。

PaaS 平台的网络需求

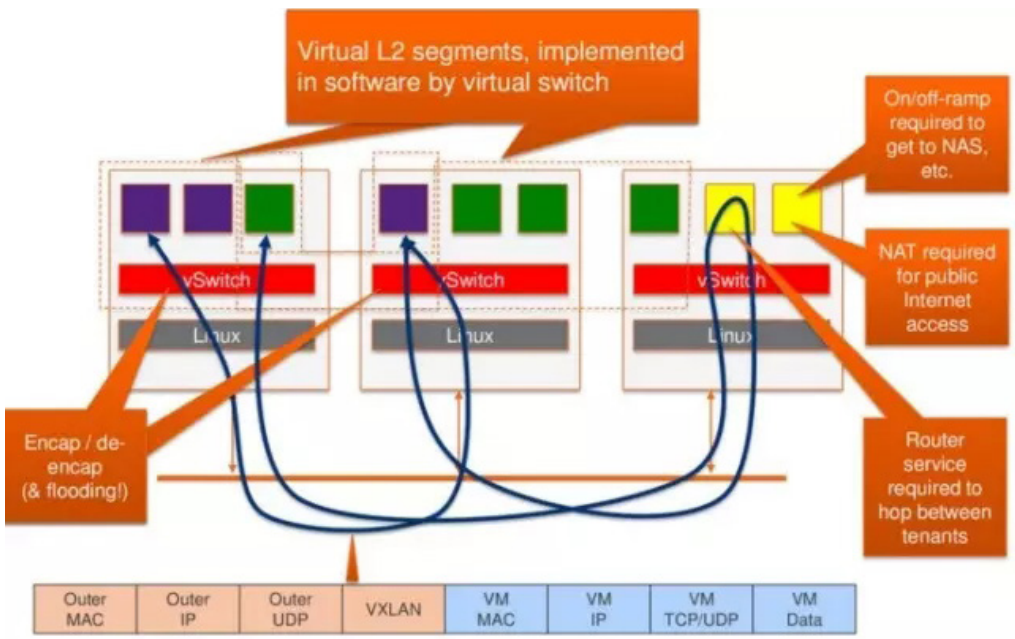
在使用 Docker 构建 PaaS 平台的过程中，我们首先遇到的问题是需要选择一个满足需求的网络模型：

- 让每个容器拥有自己的网络栈，特别是独立的 IP 地址。
- 能够进行跨服务器的容器间通讯，同时不依赖特定的网络设备。
- 有访问控制机制，不同应用之间互相隔离，有调用关系的能够通讯。

调研了几个主流的网络模型：

- Docker 原生的 Bridge 模型：NAT 机制导致无法使用容器 IP 进行跨服务器通讯 (后来发现自定义网桥可以解决通讯问题，但是觉得方案比较复杂)。
- Docker 原生的 Host 模型：大家都使用和服务器相同的 IP，端口冲突问题很麻烦。

- Weave OVS 等基于隧道的模型：由于是基于隧道的技术，在用户态进行封包解包，性能折损比较大，同时出现问题时网络抓包调试会很蛋疼。
- 在对上述模型都不怎么满意的情况下，发现了一个还不怎么被大家关注的新项目：**Project Calico**。



Project Calico 是纯三层的 SDN 实现，它基于 BPG 协议和 Linux 自己的路由转发机制，不依赖特殊硬件，没有使用 NAT 或 Tunnel 等技术。能够方便的部署在物理服务器，虚拟机（如 OpenStack）或者容器环境下。同时它自带的基于 Iptables 的 ACL 管理组件非常灵活，能够满足比较复杂的安全隔离需求。

使用 Calico 来实现 Docker 的跨服务器通讯

环境准备

- 两个 Linux 环境 node1|2 (物理机，VM 均可)，假定 IP 为: 192.168.78.21|22。
- 为了简单，请将 node1|2 上的 Iptables INPUT 策略设为 ACCEPT，同时安装 Docker。
- 一个可访问的 Etcd 集群 (192.168.78.21:2379)，Calico 使用其进行数据存放和节点发现

启动 Calico

在 node1|2 上面下载控制脚本:

```
# wget https://github.com/projectcalico/calico-docker/releases/download/v0.4.9/calicoctl
```

启动

```
# export ETCD_AUTHORITY=192.168.78.21:2379
# ./calicoctl node --ip=192.168.78.21|22
```

docker ps 能看到:

```
CONTAINER ID   IMAGE                 COMMAND           CREATED          STATUS
PORTS   NAMES
74cc20b90b0f   calico/node:v0.4.9   "/sbin/my_init"   24 seconds ago   Up 23
seconds          calico-node
```

部署测试实例

在 Calico 中，有一个 Profile 的概念（类似 AWS 的 Security Group），位于同一个 Profile 中的实例才能互相通讯，所以我们先创建一个名为 db 的 Profile：

在 node1 上执行：

```
[node1]# ./calicoctl profile add db
```

然后启动测试实例：

```
[node1]# export DOCKER_HOST=localhost:2377
[node1]# docker run -n container1 -e CALICO_IP=auto -e CALICO_PROFILE=db
-td ubuntu
```

这里大家注意，我们注入了两个环境变量：CALICO*IP* 和 *CALIC*OPROFILE。

前者告诉 CALICO 自动进行 IP 分配，后者将此容器加入到 Profile db 中。

那么 Calico 是怎么做到在容器启动的时候分配 IP 的呢？

大家注意我们在 run 一个容器前，先执行了一个 export，这里其实就是将 Docker API 的入口劫持到了 Calico 那里。Calico 内部是一个 twistd 实现的 Python Daemon，转发所有 Docker 的 API 请求给真正的 Docker 服务，如果发现是 `start` 则插入自己

的逻辑创建容器的网络栈。 容器启动后我们查看 container1 获取的 IP 地址:

```
[container1]# ip addr

...
8: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
link/ether 1e:48:3e:ec:71:52 brd ff:ff:ff:ff:ff:ff
inet 192.168.0.1/32 scope global eth1
   valid_lft forever preferred_lft forever
```

我们会看到 eth1 这个网络接口被设置了 IP `192.168.0.1`。同样在 node2 上面部署 container2。默认设置下 IP 会在 192.168.0.0/16 中按顺序分配，所以 container2 会是 `192.168.0.2`。然后我们就会发现 container1|2 能够互相 ping 通了!

路由实现

接下来让我们看一下在上面的 demo 中，Calico 是如何让不在一个节点上的两个容器互相通讯的:

- Calico 节点启动后会查询 Etcd，和其他 Calico 节点使用 BGP 协议建立连接。 [node1]# netstat -anpt | grep 179 tcp 0 0 0.0.0.0:179 0.0.0.0:* LISTEN 21887/bird tcp 0 0 192.168.78.21:46427 192.168.78.22:179 ESTABLISHED 21887/bird

- 容器启动时，劫持相关 Docker API，进行网络初始化。
  - 如果没有指定 IP，则查询 Etcd 自动分配一个可用 IP。
  - 创建一对 veth 接口用于容器和主机间通讯，设置好容器内的 IP 后，打开 IP 转发。
  - 在主机路由表添加指向此接口的路由。

  主机上：[node1]# ip link show ... 7: cali2466cece7bc: <BROADCAST,MULTICAST,UP,LOWER*UP*> *mtu* 1500 qdisc pfifofast state UP mode DEFAULT qlen 1000 link/ether 96:c4:86:4d:d7:2c brd ff:ff:ff:ff:ff:ff

  容器内：[container1]# ip addr ... 8: eth1: <BROADCAST,MULTICAST,UP,LOWER*UP*> *mtu* 1500 qdisc pfifofast state UP group default qlen 1000 link/ether 1e:48:3e:ec:71:52 brd ff:ff:ff:ff:ff:ff inet 192.168.0.1/32 scope global eth1 validlft forever preferredlft forever

  主机路由表：[node1]# ip route ... 192.168.0.1 dev cali2466cece7bc scope link
- 然后将此路由通过 BGP 协议广播给其他所有节点，在两个节点上的路由表最终是这样的：

  [node1]# ip route ... 192.168.0.1 dev cali2466cece7bc scope link 192.168.0.2 via 192.168.78.22 dev enp0s8 proto bird

  [node2]# ip route ... 192.168.0.1 via 192.168.78.21 dev enp0s8 proto bird 192.168.0.2 dev caliea3aaf5a7be scope link

大家看这个路由，node2 上面的 container2 要访问 container1（192.168.0.1），通过查路由表得知需要将包转给 192.168.78.21，也就是 node1。形象的展示数据流向是这样的：

```
container2[eth1] -> node2[caliea3aaf5a7be] -> route -> node1[cali2466cece7bc] -> container1[eth1]
```

至此，跨节点通讯打通，整个流程没有任何 NAT，Tunnel 封包。所以只要三层可达的环境，就可以应用 Calico。

利用 Profile 实现 ACL

在之前的 demo 中我们提到了 Profile，Calico 每个 Profile 都自带一个规则集，用于对 ACL 进行精细控制，如刚刚的 db 的默认规则集是：

```
[node1]# ./calicoctl profile db rule json

{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}
```

这个规则集表示入连接只允许来自 Profile 名字是 db 的实例，出连接不限制，最后隐含了一条默认策略是不匹配的全部 drop，所以同时位于不同 Profile 的实例互相是不能通讯的，这就解决了隔离的需求。

下面是一个更复杂的例子：

在常见的网站架构中，一般是前端 WebServer 将请求反向代理给后端的 APP 服务，服务调用后端的 DB：

```
WEB -> APP -> DB
```

所以我们要实现：

- WEB 暴露 80 和 443 端口；
- APP 允许 WEB 访问；
- DB 允许 APP 访问 3306 端口；
- 除此之外，禁止所有跨服务访问。

那么我们就可以如此构建 json：

对于 WEB：

```
[node1]# cat web-rule.json

{
  "id": "web",
  "inbound_rules": [
    {
```

```
      "action": "allow",
      "src_tag": "web"
    },
    {
      "action": "allow",
      "protocol": "tcp",
      "dst_ports": [
        80,
        443
      ]
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}

[node1]# ./calicoctl profile web rule update < web-rule.json
```

入站规则我们增加了一条允许 80 443。

对于 APP:

```
[node1]# cat app-rule.json

{
  "id": "app",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "app"
    },
    {
      "action": "allow",
      "src_tag": "web"
```

```
    }
  ],
  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}

[node1]# ./calicoctl profile app rule update < app-rule.json
```

对于后端服务，我们只允许来自 web 的连接。

对于 DB，我们在只允许 APP 访问的基础上还限制了只能连接 3306。

```
[node1]# cat db-rule.json

{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    },
    {
      "action": "allow",
      "src_tag": "APP",
      "protocol": "tcp",
      "dst_ports": [
        3306
      ]
    }
  ],
  "outbound_rules": [
```

```
    {
      "action": "allow"
    }
  ]
}

[node1]# ./calicoctl profile db rule update < db-rule.json
```

很简单的几条规则，我们就实现了上述需求。

Profile 高级特性：Tag

有同学可能说，在现实环境中，会有多组不同的 APP 都需要访问 DB，如果每个 APP 都在 db 中增加一条规则也很麻烦同时还容易出错。

这里我们可以利用Profile 的高级特性 Tag 来简化操作：

- 每个 Profile 默认拥有一个和 Profile 名字相同的 Tag。
- 每个 Profile 可以有多个 Tag，以 List 形式保存

利用 Tag 我们可以将一条规则适配到指定的一组 Profile 上。

参照上面的例子，我们给所有需要访问 DB 的 APP 的 Profile 都加上 db-users 这个 Tag：

```
[node1]# ./calicoctl profile app1 tag add db-users
[node1]# ./calicoctl profile app2 tag add db-users
[node1]# ./calicoctl profile app3 tag add db-users
...
```

然后修改 db-rule.json 为：

```
{
  "id": "db",
  "inbound_rules": [
    {
      "action": "allow",
      "src_tag": "db"
    },
    {
      "action": "allow",
      "src_tag": "db-users",
      "protocol": "tcp",
      "dst_ports": [
        3306
      ]
    }
  ],

  "outbound_rules": [
    {
      "action": "allow"
    }
  ]
}
```

将之前的 src*tag*: *app* 替换为 *srct*ag: db-users。这样所有打了 db-user 这个 Tag 的实例就都能访问数据库了。

Profile 的实现

Profile 的实现基于 Iptables 和 IPSet。我们以刚刚的 `db` 规则集中 `inbound` 部分为例：

Calico 在启动后会在 Iptables 中新建一些 Chain，数据包会在不同的 Chain 之间跳转，下面我截取了一些关键的规则列表：

```
[node1]# iptables -n -L -v

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
target         prot  in  out  source     destination
felix-FORWARD  all   *   *    0.0.0.0/0  0.0.0.0/0

Chain felix-FORWARD (1 references)
target             prot  in  out    source     destination
felix-TO-ENDPOINT  all   *   cali+  0.0.0.0/0  0.0.0.0/0

Chain felix-TO-ENDPOINT (1 references)
target                prot  in  out              source     destination
felix-to-2466cece7bc   all    *    cali2466cece7bc   0.0.0.0/0   0.0.0.0/0
[goto]

Chain felix-to-2466cece7bc (1 references)
target        prot  in  out  source     destination
felix-p-db-i  all   *   *    0.0.0.0/0  0.0.0.0/0

Chain felix-p-db-i (2 references)
target  prot  in  out  source     destination
RETURN  all   *   *    0.0.0.0/0  0.0.0.0/0    match-set felix-v4-db src
RETURN  tcp    *   *    0.0.0.0/0  0.0.0.0/0    match-set felix-v4-db-users
src multiport dports 3306
```

这个略复杂，我们慢慢看。基本上数据包是从上到下一步步跳转的。

当发给 container1 的数据包到达 node1 后，由于目标 IP 192.168.0.1 和 node1 自身 IP 不同，会被放入 FORWARD 链，然后跳转到 felix-FORWARD，通过查询路由表：

```
192.168.0.1 dev cali2466cece7bc  scope link
```

得知下一跳接口为 cali2466cece7bc，于是先跳转到 felix-TO-ENDPOINT，再跳转到 felix-to-2466cece7bc。

在这里，定义了具体的 ACL 列表，felix-p-db-i，这个 db 是不是很眼熟?

对，就是这个 container 所属 Profile 的名字，而 felix-p-db-i 中就是 Profile db 的 inbound 规则集。而 felix-p-db-i 的内容:

```
match-set felix-v4-db src
match-set felix-v4-db-users src multiport dports 3306
```

felix-v4-db 和 felix-v4-db-users 是不是也很熟悉?

在 db 规则集中的两个 Tag 在这里加了个前缀变成了 IPSet，它包括了所有打了这个 Tag 的 IP 列表:

```
[node1]# ipset list
...
Name: felix-v4-db
Type: hash:ip
Revision: 1
Header: family inet hashsize 1024 maxelem 65536
Size in memory: 16576
References: 1
Members:
192.168.0.1
192.168.0.2
```

至此，ACL 部分分析完毕。可见 Calico 灵活运用了 Iptables 的种种高级特性。

性能测试

来自官方测试结果:

测试环境

- 8 core CPU
- 64GiB RAM
- 10Gb 网卡直连
- Ubuntu 14.04.2 with 3.13 Kernel (3.10 版本的 Kernel 修复了一些 veth 性能的问题)
- 没有额外的内核参数调优

测试了四种场景:

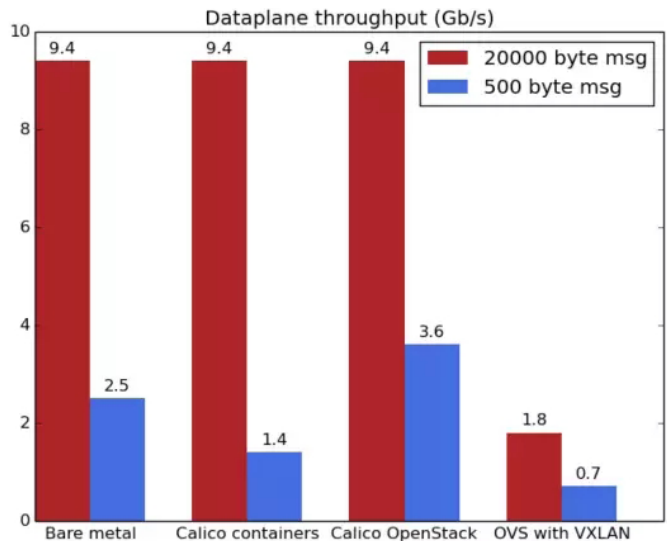
- 物理服务器 (基准)
- 部署了 Calico 的容器之间
- 部署了 Calico 的 OpenStack VM 之间
- 部署了 OVS with VxLAN 的 OpenStack VM 之间

测试了两种数据大小:

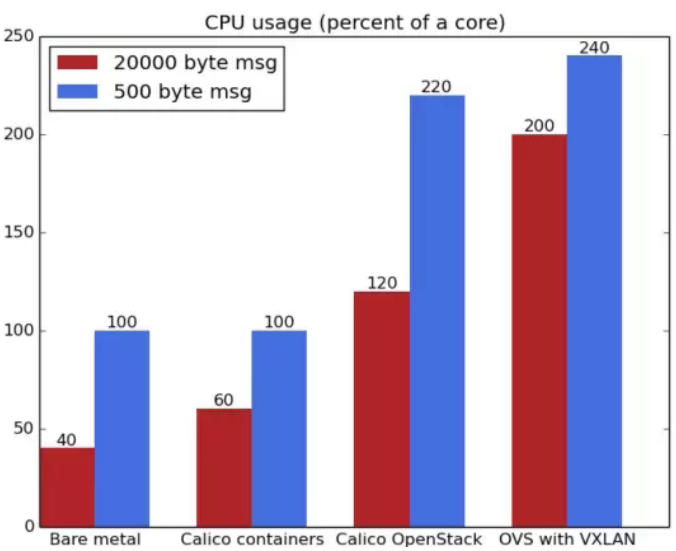
- 20000 byte
- 500 byte

吞吐量 & CPU 使用率测试

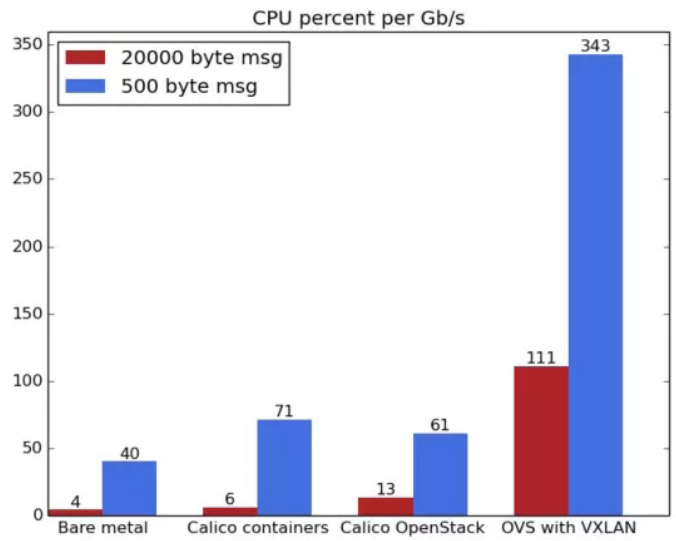
吞吐量极限:



同一时刻 CPU 的使用率:



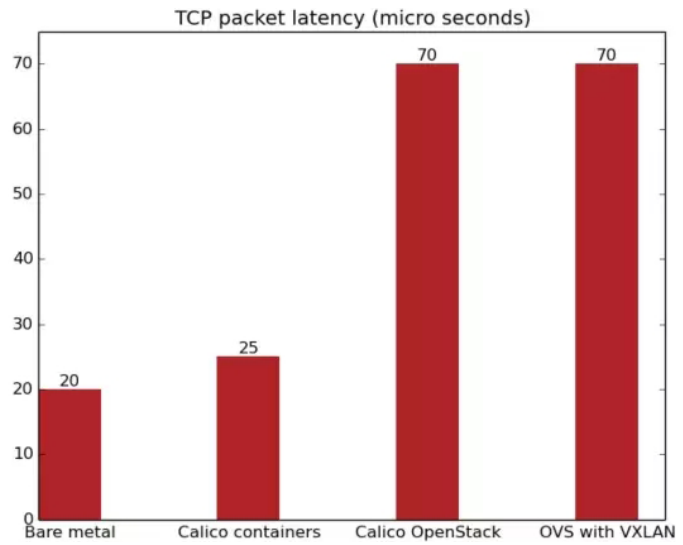
把它们合并成一张图就是每 Gb/s 的 CPU 使用率：



可以看到，部署了 Calico 的两个场景都非常贴近物理服务器的性能。

延迟测试

测试方法是：节点间交换 1 byte 数据包。



这个结果显示，Calico 容器非常接近物理服务器，而 OpenStack 场景由于网络虚拟化的缘故延迟稍大。

结论

测试结果表明，Calico 的性能非常接近物理服务器，比基于隧道的 OVS 性能好很多。

Calico 的发展

Calico 和 Docker 一样是很年轻的项目，但是坑比后者少多了，我遇到了一些，如 `docker inspect` 没有显示 Calico 分配的 IP，BGP 客户端重启姿势不正确导致路由周期性消失重建等等。但是他们的开发进度非常快，一个 issue 提出来到修复可能就一两天时间（再次鄙视 Docker）。

目前唯一一个比较麻烦的问题是，Calico 这种劫持 Docker API 的方式，容器的网络栈是在容器启动后才进行初始化，所以在头几秒其实是没有网络可用的，这会导致那些启动就要访问网络的容器挂掉。解决方案有两个：

升级 Docker 到支持 libnetwork 的版本，Calico 在新版本（>0.5）中支持了 libnetwork，理论上能够解决这个问题。但是代价要踩新版本 Docker 带来的更多的坑。

自定义容器的 CMD，实现一个 entry 脚本，待网络可用后再 exec 载入真正的进程。■

Q&A

Q1：自定义容器的 CMD，实现一个 entry 脚本，待网络可用后再 exec 载入真正的进程。 有没有具体的?

主要实现方式就是 Dockerfile 中的 CMD 可以这个样子：`/entry.sh your-cmd` 。这个 entry.sh 中判断 IP 是否已经分配好，如果没有就 sleep 重试。分配好后再用 exec 载入后面的 your-cmd 。

Q2：是否每增加一个容器宿主机就需要增加一条路由? 如果容器数量很多会有问题吗?

是的。关于这个问题我咨询过开发团队，他们表示压测过单机 10 万条路由，没有问题。同时将来会推出路由段的广播机制，即：每台服务器使用一小段，之间只需要广播此段即可。

Q3：如果要做容器间通讯限速，Calico 能做吗?

由于每个由 Calico 管理的容器在宿主机上面都有一个唯一的网络接口 (veth 的一端)，通过限制此接口流量即可进行限速。Calico 官方没有提供这个功能，我们可以用常规的其他手段解决。同时这个接口还有一个好处就是非常容易做容器的流量监控，只要看接口计数器即可。

Q4：Calico 和 Kubernetes 的整合你们尝试过吗? Calico 只是接管了容器间的通信，和 Kubernetes 的 Service Cluster IP 没有关系吧?

我们没有使用 Kubernetes 的解决方案，而是自行开发的调度编

排等组件。Cailco 官方是支持的并且有相关文档可以参考。具体请参考：https://github.com/projectcalico/calico-docker/blob/master/docs/kubernetes/README.md。

Docker实战

张磊：关于Docker、开源，以及教育的尝试

张磊，浙江大学计算机学院博士生，科研人员，VLIS实验室云计算组技术负责人、策划人。Kubernetes项目贡献者和维护者，Docker项目贡献者。前Cloud Foundry中国团队和百度私有云项目组成员。InfoQ、CSDN、《程序员》杂志等多篇浙大系技术文章的贡献者和策划人，他还是《Docker——容器与容器云》一书的主要作者之一。

作者 / 图灵访谈

欢迎关注图灵访谈微信公众号！对话国外知名技术作者，讲述国内码农精彩人生。你听得见他们，他们也听得见你。微信号：ituring_interview。

问：你现在的目标是成为一位计算机科学家吗？

是。

问：你是从什么时候开始想成为一位计算机科学家的？

大概是由于中学时候就开始玩电脑，然后自然而然就进入了计算机专业。由于我的兴趣主要在云计算以及Linux操作系统上，所以我的导师跟课题都是跟这些相关的。

2011年我们开始专注于开源云计算技术，当时开源的力量正在逐渐浮现。后来事实证明我们对趋势的判断是对的，因为从那个时候开始，软件的开发和发布以及整个生命周期就发生了改变。从此以后，可以说开源技术掌握了云计算行业，甚至是整个计算机行业的主流发展态势。于是，我们开始更深入地钻研Cloud Foundry, Docker, Kubernetes这样的技术，并且作为这些项目的贡献者成为了社区中的重要成员。

问：你认为在云计算领域，学术跟产业之间有没有明确的界限？

首先，云计算这个领域本身就比较特殊，它其实没有很多基础性研究，所以这个领域的学术跟工业是分不开的，中间没有一个明显的界限。比如，伯克利的AMP实验室做的一套高性能数据分析系统，最终开源出去就变成现在的明星项目Spark，成为工业界大数据的事实标准。在这些领域中，没有任何一个界限能够划定哪些技术是学术的，而工业界不能用。因为云计算本来就是“站在巨人肩膀上”的一种技术，它基于已有的分布式系统来做进一步的创新和整合。

问：你现在在SEL实验室的工作是什么？

我主要负责实验室云计算团队的技术工作，以及与技术相关的其他事宜，包括开源以及一些商业上的技术合作。

问：SEL实验室的前身是VLIS实验室，当初建立VLIS实验室的目的是什么？现在SEL实验室的关注点有没有变化？

我们实验室最开始的研究方向就是软件工程以及计算机软件。实验室一开始就注重从学校的角度跟知名企业建立强强联合的关系，致力于为工业界提供最好的大规模信息系统的开发技术和能力。在那个时候，我们的主要关注点是金融信息系统的开发，并且同全球最大的资金托管机构美国道富银行建立了紧密的合作研发关系。从这个时候开始，我们向工业界输出了大量的技术能力，整个北美市场的股票交易系统的后台都是我们实验室师生参与重新开发的。而软件技术发展到现在，新一代的大规模分布式系统开始更多地以知名开源项目作为表现形式，这些技术也就自然成为我们新的关注点。其中最重要的还是云计算技术，但是我们略有侧重，更关注轻量化的云计算技术，我们认为这将是一个新的变革。

问：你们实验室跟Cloud Foundry还有百度、思科等知名企业都是以什么形式合作的？

首先肯定有人才上的合作，因为学校本来是人才，我们会选择一些从事这个方向的优秀的学生，联合工业界的公司，比如去道富，VMware，思科总部或者百度等等，完成一个以技术研究为核心，以实际开发为途径的长期合作学习的过程。学生不是实习几个月然

后回来，而是从开始到毕业，从研究的方向到最后的毕业论文都跟这些公司的真实技术场景紧密相关，并且专注于这些IT巨头的核心技术以及云计算平台的开发和研究工作。

问：你们实验室为什么在Docker还不太完善的时候就敢去尝试这种技术？

容器技术不是一种新技术，它其实很早就存在了。在此之前，我们搞Linux内核的时候已经用过类似的技术，而且做PaaS用到的技术也是基于Linux容器的，所以我们团队很久以前就对这种技术有过很多接触和研究。2013年的时候，我们同学主要在从事的是VMware的Warden容器的研发，紧接着Docker就出现了，并且比前一代容器技术要完善很多，所以我们就自然而然地转到了Docker上。这就是为什么Docker一出现就会引起我们的关注。轻量化的云计算技术一定会以这样的方式实现，只不过实现不同，而我们肯定会选择更好的实现。

问：你本人是怎么成为Kubernetes和Docker项目的贡献者的？

我是从Cloud Foundry团队出来的，而Cloud Foundry是一个纯粹的开源项目，它没有市场人员和项目经理这样的角色来干扰工程师的工作，所有贡献者都是通过远程协作和结对编程来贡献代码的。我，以及我们实验室的大多数同学从一开始就是这样一种工作模式，所以对于我们来说，参与开源项目是很自然的，而且我们也不像其他公司那样寻求互等的商业利益。

我们认为开源项目是一定要参与的，不仅要参与，还要学会主导项目的方向，成为维护者和更重要的核心代码贡献者。我刚才讲过，我们的愿景就是要基于开源软件做事情，做研究也好，做进一步的商业活动也好，一定要进入到社区里面，而不仅仅是一个使用者。

问：Docker一直存在安全方面的问题，在这方面你有哪些经验可以和大家分享？

我也在《Docker——容器和容器云》里提到过，Docker本身确实有安全问题，但是一定要分场景讨论。比如，什么样的场景下我会在一台裸机上部署Docker；什么样的场景下我会让Docker容器跑在虚拟机里面。在目前这种情况下，如果你是一个公有云提供商，我认为你还是要把容器拷在你的虚拟机里面，防止出现逃逸状况。

我们在书里讲过，你的Docker容器和整个Docker Daemon环境最好做安全加固，在操作系统层面做很多加固，设置权限，并且在整个系统的设计上把权限设计和授权设计摆在第一位，逐层来把不正确的行为过滤掉。另外，一定要区分场景。对于私有云的话，安全要求满足第一点就可以了。但是对于公有云来说，一定要做最高等级的安全预案。

以上是从业务方面讲，但是从技术方面讲，容器本身的安全问题是很难解决的，但是有一些努力的方向非常值得我们关注。比如最新的Rocket集成了英特尔在CPU上的一些虚拟化技术来做到硬件加固，这就是等级很高的安全技术。另外还有像国内赵鹏他们做的Hyper，是一个基于虚拟化技术的容器，它跟虚拟机的安全系数几

乎是一样的。所以我觉得从另一方面说，这些技术应该得到大家的重视，并且集成到我们现有的解决方案里面。

问：现在在生产环境中使用Docker的人还是不太多，其背后的原因多种多样，你认为现在Docker面临的最大的阻碍是什么？

第一个问题在于Docker所属公司本身的强势，以及他们自己想做一揽子事情的态度。这个问题使得Docker现在变得非常臃肿，而且导致本来应该专注解决的问题没有解决掉。如果他们投入主要力量来解决容器的安全问题，我觉得反而要比现在的态势好。

第二，我们使用Docker也好，对它做二次开发也好，其实我们不要把自己的眼光局限在Docker上面。让Docker只做容器的事情其实是最好的选择，其余的事情交给更专业的人。我们不要过分相信来自某些商业化的宣传，比如一个人或者一个团队就能完成从开发到部署到运维的所有流程。哪怕你一开始可以，但是随着业务的正常增长，技术发展到一定程度之后你一定是做不到的。所以一定是专业的人做专业的事。

问：Docker和CoreOS一起创立了一个开放容器计划（OCP）。你认为OCP的成立对于软件开发行业会造成什么影响？

容器镜像不单指Docker镜像，它很久之前就已经存在了。容器镜像作为一种软件的发布方式，现在已经得到了大家的认可，成为了行业的事实标准。并且由于容器镜像本身已经存在了很久，所以它

本身标准的普及是比较容易的。所以OCP的成立其实是为这种镜像发布的方式提供了一个技术上的标准。

以前，这一套东西虽然可以做标准，但是没有技术来支撑它，现在有了。我们通过标准容器来支撑标准镜像。所以，这两个标准如果能够在OCP里面得到统一，我相信它对整个软件工程将来的发展都是有很大影响的。我们现在学校的课程里就已经引入了完全基于容器的软件工程设计模式，谷歌也提出了基于容器的编程模型，将来的软件工程一定会向这方向发展。

问：你有一篇文章叫做《从Borg到Kubernetes》。你觉得Kubernetes今后的发展会怎么样？它和Mesos分别会向什么方向发展？

我们最近也跟谷歌的人一起交流了很久，首先，Kubernetes确实背负了很多Borg之前的优秀的设计理念，其中包括Borg在谷歌内部大规模集群业务的应用。虽然现在还有一些应用我们看不到，但是Kubernetes将来的发展目标一定是用来解决这些问题。

Mesos和Kubernetes在一开始发展时其实是非常直接的竞争对手，因为这两者关注的事情有很多是一致的。但是随着这两个项目的继续发展，它们已经形成了合作关系。比如，Mesos本来就是一个优秀的调度器，那么接下来Mesos会更关注这个业务。并且Mesos可以被更方便地集成到Kubernetes里，作为Kubernetes的一个核心调度器来工作。

这两个项目现在的关注点其实是不一样的，使用的场景也不一样。

在今后的发展中，它们会逐渐融合对方的优点。互相之间的集成会越来越多，互相之间的重合会越来越少。

问：学术跟产业之间的脱轨问题一直以来经常被人们所诟病，浙大在这方面做得很不错，有没有什么经验可以分享？

首先，作为一所学校，要学会如何在我们国家的体制下，在不影响正常的教学、科研的前提下，获得企业的支持。学校需要鼓励学生在看似枯燥的学业中找到真正的个人兴趣点。比如我们实验室就涌现出了非常多的代码贡献者、作者、领域内的小专家。因为我们实验室从一开始就鼓励个人发展自己的兴趣，并且鼓励同学向大家分享你的工作成果。这样，工业界会自然而然地关注过来，合作也接踵而至。

另一方面，如果学校自身的硬件条件很强，技术水平很高的话，学校就可以为工业界做出很多贡献，无论是开源贡献，还是参与到工业的开发。并且还有一点很重要，就是学校要想办法把实验室的技术和研究经验转化成工程上可以应用到增值需求里的东西，而不只是埋头写论文。所以，我们实验室在硕士生阶段，不会提出苛刻的论文要求。我们更希望你的论文是对一个开源项目的贡献，或者是我们和产业界合作的课题的相关实际工程经验。

同时，我们也在浙大试点了一个学院，整个软件学院在以更加工程的方式推动科研的走向。我们课题组从十年前开始做这样的事情，所以学术跟产业之间的脱轨问题我们这边几乎是没有的。

问：浙大有这么好的环境，但是很多其他大学没有能力提供这样的环境。你建议在一般大学学习的学生怎样来丰富自己的专业知识?

首先对于一个学生，尤其是CS专业的学生，我认为实习是最重要的。你一定要想办法在跟导师融洽相处的前提下，寻求到与自己专业相关的实习机会，并且珍惜这些机会，因为实习能够使你的专业技能得到锻炼。并且你应该想办法把实习转化成毕业论文，或者是学校要求的课程设计。这样的经历会对你在工业界的影响力也好，工作也好，起到很大的积极作用。

问：你认为学习Docker需要几个阶段?

我们在书的后记里面讲过，不止是Docker，对于任何一个开源项目来说，都有这样的三个或四个阶段。 首先，你要去用，而且不只要用，还要变成一个优秀的玩家。对于开源项目的所有指令、所有设计，你应该有一个感性认识。

在这个基础之上就是源码，要读源码。读源码是一件非常有意思的事，但是在这个过程中，你要学会提问，带着问题去读源码，才会有收获。

然后就是转化，转化包括几种情况。比如，你可以将容器技术转化成你们实验室的某个项目的基础或者工作中的整个项目。另外，你要学会对项目做贡献。从最开始的找bug、解决bug、修改文档，到最后提出自己的特性、融入到社区里，只有这样你才能够获得最多的知识，以最快的速度提高自己在这个领域中的能力。

这三步之后，如果你在这个方面做得更多，可以考虑一些商业化的事情。比如你可以做一些相关的买卖，或者在你的公司里推广这些开源技术。

问：你觉得读者应该怎样使用《Docker——容器和容器云》这本书，读者在哪个阶段需要用到这本书？

这本书应该更适合在第二或第三阶段阅读。

这本书的一个特点就是它倾向于把原理帮你从源码中抽象出来，而不仅仅带你走读代码。因为代码很快会过时，所以我们特别注重抽象原理。在第三阶段的时候，你要去做开发，所以你对技术的熟悉程度和原理必然要有一个深刻的认识。而这本书的很多实践章节，包括我们对代码的整个框架结构的分析，会对你有很大的帮助。

问：《Docker——容器和容器云》的关注点和其他类似的书有什么不同？

首先，我们不认为容器就是Docker，我们认为它只是容器技术一种优秀的实现。所以我们的书叫《Docker——容器与容器云》，我们更关注所有基于容器的云平台的实现方式。你可以把容器理解为我们现在的虚拟机，把容器云理解为OpenStack，所以我们这本书肯定要先讲虚拟机原理，以此为基础我们才能讲清楚容器云，也就是容器的大规模管理方式。

很多目前市面上的书只关注于Docker本身，而我们更关注Docker背后的libcontainer也就是runc的工作原理，于此同时，

我们还非常关注所谓的容器技术与大规模容器集群管理的结合方式。我们非常想为大家解决的一个问题就是，真正的大规模容器集群管理应该是什么样的。我们认为 Kubernetes 现在的方向是非常好的，所以在书中我们对它做了一个非常详细深入的解读，这在国内外应该是首次，而且从谷歌和 CoreOS 工程师的反馈来看，甚至在国外可能也是第一次。■

Docker实战

Joyent首席技术官关于容器的新年愿望

作者 / Trevor Jones

Trevor Jones，TechTarget 数据中心和虚拟化媒体集团的新闻作者。

为了在云市场取得领先地位，Joyent 在容器上下了很大赌注。关于容器技术的现状和亟需改变的问题，Joyent 首席技术官 Bryan Cantrill 有话要说。

Joyent 公司在过去的十年中一直在生产环境中实践开源容器技术，当 Docker 横空出世之时，Joyent 是最早投身这种技术的供应商之一。

去年年底，这家云供应商把自己重新打造成了“原生容器基础设施”，从那时起，他们发布了多种产品，其中包括 Triton——用于管理 Docker 容器的容器基础设施；以及把传统应用容器化，使它们可以在任何地方运行的 Containerbuddy。

这家公司同时还是“原生云计算基金会”的创始成员之一，该基金会于 7 月成立，目的在于创造原生云应用和容器的参考架构。Joyent 的首席技术官 Bryan Cantrill 是该基金会技术指导委员会的成员。他最近和 SearchCloudComputing 谈到了容器技术的现状，他对未来一年期待，以及亟需改变的问题。

问：应用容器持续受到了很多关注，但是问题仍然存在。这方面你怎么看？

每个人都把容器看做通往下一个目的地的路径，而这样的情况到底意味着什么也引起了大量的恐慌。在 15 年前的虚拟机革命中，人们不需要改变思维方式——只需要把实体（机）虚拟化就好了。对

于容器来说，机会更多，而很多人也认为同时存在着不少陷阱，因为容器会让你改变思考问题的方式。

问：有些人会认为容器之所以能够勇往直前是因为它们有些类似于虚拟机。

没错，但是他们会发现容器只是和容器所在的环境水平相当。你有了能在生产中盛东西的容器并不代表你就有了服务发现。当我们把宠物转变成牲畜时，容器的意义可远远不止于改变基底。容器的真正意义在于更加快速和简单地构建大规模系统，但是同时还需要解决如何有效地把系统转变为微服务的问题。

问：现在容器市场的成熟度如何？人们对容器技术的理解水平如何？

KubeCon（最近谷歌的Kubernetes社区举行的大会）提出的问题之一就是："容器领域接近峰值混乱状态了吗？"有趣的是，每个和我交谈的人——无论他们在进行开发或运维，或者身为供应商——都认为我们没有进入峰值混乱。他们仍然期待成长，甚至有些人期待着加速成长，这种想法还挺令人不安的。

我不知道我们应该期待容器技术在近期达到多大程度的稳定性，因为所有的技术都是开源的。这里存在着各种各样的框架和理念，也有很多混乱以及和简洁相悖的气质。直白一点说，这里有很多领土争夺的意味，在这种情况下如果你故意想要制造出尽量多的领域，那么界定边界就会很难。

问：容器即服务（CaaS）是我们从供应商那里看到的最新趋势。你是否担心某些服务和容器带来的专有概念会让跨平台可移植性大打折扣？

人们确实对华而不实的容器即服务抱有成见。他们想要自己使用容器来构建东西。我们的信念就是，我们需要原生容器基础设施——并不是在虚拟机内部供应，而是在纯粹的容器上供应。在这种情况下，如果容器有一个IP地址的话某些事情就会简单得多，而你也不需要完成任何愚蠢的映射特技了。

我的确认为有些容器服务，特别是那些运行在虚拟机内部的服务，只是现有的云技术专家想要在容器领域“到此一游”而已。他们并不理解容器革命背后隐含的经济学原理。

问：容器市场接下来需要发生什么样的变化？

概念的扩张需要放缓一些，或者我们需要沉淀一些东西，让这些技术在生产环境中做出有意义的东西来。但是这方面我们还一筹莫展……

你可以真正在生产环境部署Triton。你可以用Docker Compose来支持适应性比较强的服务，但是除了Triton以外，很多技术感觉上还很稚嫩和原始，而这样的技术也招来了一些恶名。人们不应该再吹嘘自己收到多少代码提交，拥有多少贡献者以及下载量了。对于我来说，这样做从某种程度上代表着混淆视听。如果你面临着一两万个问题，你怎么能忍受项目上的任何东西以这样的速度迅速扩张呢？

明年的某个时间，我们至少会在心里巩固某些概念，并不是说未来将会产生一位唯一的冠军，而是说我们将会更好地了解某些概念在哪些方面合适，在哪些方面不合适。进一步的突变也很有可能会发生，尽管如此，我仍然相信我们将会看到更多对稳健性的强调，以及更少的不间断扩张。■

英文原文：Joyent CTO talks Docker containers and the work ahead in 2016

高可用架构部分分享讲师名单（按姓名首字母排序）

- 陈飞，新浪微博技术经理
- 常雷博士，Pivotal 中国研发中心研发总监，HAWQ 并行 Hadoop SQL 引擎创始人，Pivotal HAWQ 团队负责人
- 陈宗志，奇虎 360 基础架构组高级存储研发工程师
- 杜传赢，Google 研发工程师
- 董西成，Hulu 网高级研发工程师，dongxicheng.org 博主
- 付海军，时趣互动技术总监
- 冯磊，新浪微博技术保障架构师
- 高磊，雪球运维架构师
- 郭斯杰，Twitter 高级工程师
- 郭伟，腾讯安全架构师
- 高永超，宜信大数据创新中心云平台运维专家
- 黄东旭，Ping CAP CTO，开源项目 Codis co-author
- 霍泰稳，InfoQ、极客邦科技创始人兼 CEO
- 蒋海滔，阿里巴巴国际事业部 高级技术专家
- 金自翔，百度资深研发工程师
- 吕毅，前百度资深研发工程师
- 马利超，小米科技的系统研发与大数据工程师

- 马涛，前迅雷网络 CDN 系统研发工程师，前 EMC/Pivotal Hawq 研发工程师
- 彭哲夫，芒果 TV 平台部核心技术团队负责人
- 秦迪，新浪微博研发中心技术专家
- 沈剑，58 到家技术总监 / 技术委员会负责人
- 孙其瑞，得图技术总监
- 孙宇聪，Coding.net CTO，前 Google SRE
- 孙子荀，腾讯手机 QQ 公众号后台技术负责人
- 谭政，Hulu 网大数据基础平台研发工程师
- 唐福林，雪球首席架构师
- 田琪，京东云数据库技术负责人
- 王富平，1 号店搜索与精准化部门架构师
- 王劲，酷狗音乐大数据架构师
- 王晶昱，花名沈询，阿里资深技术专家
- 王康，奇虎 360 基础架构组资深工程师
- 王新春，大众点评网数据平台资深工程师
- 王晓伟，麦图科技
- 王渊命，Grouk 联合创始人及 CTO
- 王卫华，百姓网资深开发工程师及架构师
- 温铭，奇虎 360 企业安全服务端架构师，OpenResty 社区咨

询委员会成员

- 萧少聪，阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理
- 许志雄，腾讯云产品运营负责人
- 颜国平，腾讯云天御系统研发负责人
- 闫国旗，京东资深架构师，京东架构技术委员会成员
- 杨保华，IBM 研究院高级研究员
- 杨尚刚，美图公司数据库高级 DBA
- 尤勇，大众点评网资深工程师，开源监控系统 CAT 开发者
- 张开涛，京东高级工程师
- 赵磊，Uber 高级工程师
- 张亮，当当网架构师、当当技术委员会成员、消息中间件组负责人
- 张虔熙，Hulu 网 HBase contributor
- 赵星宇，新浪微博 Android 高级研发工程师
- 周洋，奇虎 360 手机助手技术经理及架构师

扫描二维码关注微信公众号
或搜索 [ArchNotes] 高可用架构

出品人 杨卫华
编辑/整理 李盼 臧秀涛 四正
余长洪 王杰
设计 大胖