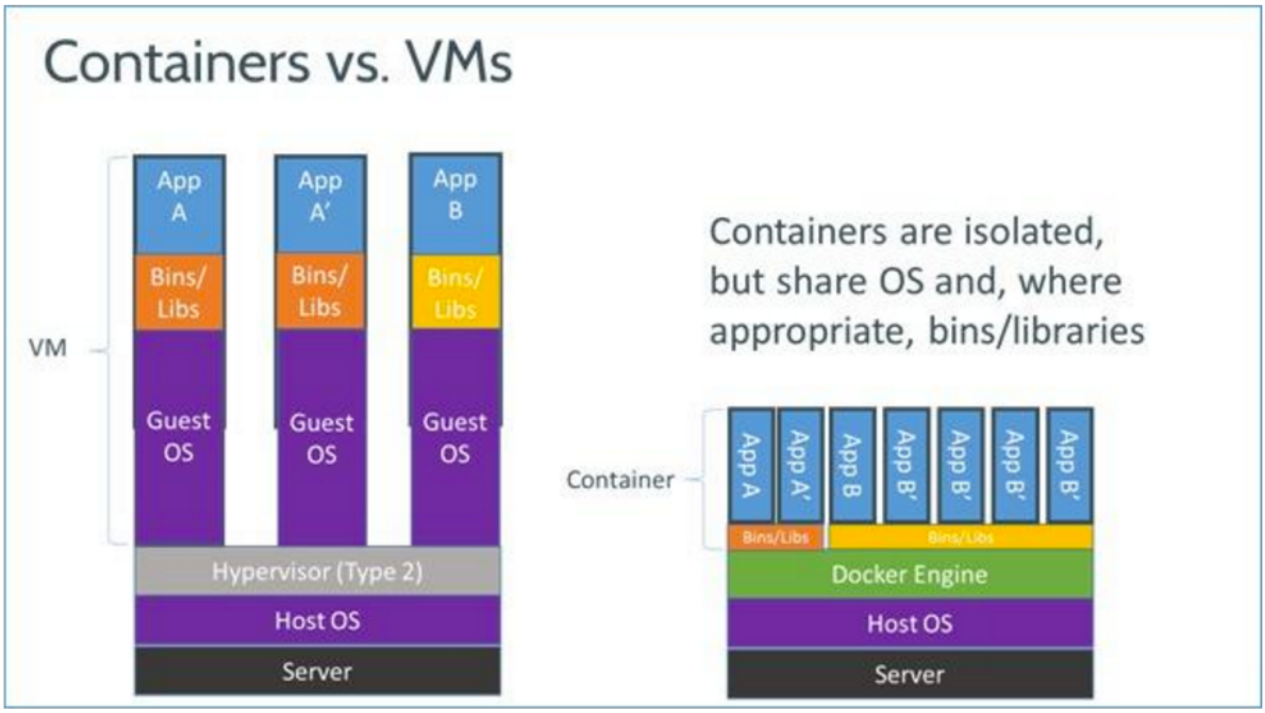
复用解决方案3 —复用云技术

1352920 冯夏令

容器技术

容器技术是一种操作系统层虚拟化（Operating system–level virtualization）技术，为Linux内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器（Container），内含应用软件本身的代码，以及所需要的操作系统核心和库。通过统一的名字空间和共用API来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得Linux用户可以容易的创建和管理系统或应用容器。

时下最为流行的容器技术莫过于Docker，Docker是对LXC(Linux Container)技术的一种封装，下图表示了Docker与传统的虚拟化技术如VM的区别。



隔离&&安全

容器的隔离指的是与Host环境和与其他容器环境的隔离；我个人认为容器的安全性是指，在一个容器里运行的进程不应该影响到Host和其他容器。所以容器的隔离问题和安全问题本质上是相同的，故在此仅仅讨论容器的隔离技术。

容器技术是一种操作系统层级的虚拟化，它创造出了一种类似于virtual machine的环境，与传统虚拟化技术相比相比，既不需要指令级模拟，也不需要即时编译，大大提高了虚拟化的速度。想要实现容器技术所面对的第一个挑战就是需要提供传统虚拟化技术所提供的隔离环境，那么在LXC是如何做到将各个容器之间的运行环境与本机上的运行环境隔离开来的呢，其实是通过Linux内核提供的Linux namespace来实现的。

环境隔离

Linux namespace

Linux Namespace是Linux提供的一种内核级别环境隔离的方法。Linux namespace在系统调用chroot的基础上，提供了对UTS、IPC、mount、PID、network、User等的隔离机制，分别对UTS（Unix Time-Sharing）、进程间通信（interprocess communication）、文件系统、用户进程、网络和用户控件进行了隔离。

· 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。

· IPC namespace

IPC(Inter-Process Communication)，即Unix/Linux下进程间通信的方式，IPC有共享内存、信号量、消息队列等方法。在各个容器中的进程需要相互通信，而不在同一个容器内的进程不能进行相互通信，所以要做到隔离，需要对IPC进行隔离，所以我们需要使用IPC namespace对IPC进行隔离。同时，IPC需要有一个全局的ID，既然是全局的，那么就意味着我们的Namespace需要对这个ID隔离，不能让别的Namespace的进程看到。

· PID namespace

我们知道，在传统的UNIX系统中，PID为1的进程是init，地位非常特殊。他作为所有进程的父进程，有很多特权（比如：屏蔽信号等），另外，其还会为检查所有进程的状态，我们知道，如果某个子进程脱离了父进程（父进程没有wait它），那么init就会负责回收资源并结束这个子进程。所以，要做到进程空间的隔离，首先要创建出PID为1的进程，最好就像chroot那样，把子进程的PID在容器内变成1。所以我们需要PID namespace将各个容器中的进程空间给隔离开来，使得每一个容器间可以有相同PID的进程，然而它们不能互相影响。

· mount namespace

在使用PID对进程空间进行隔离之后，我们会发现，在容器的shell里输入ps,top等命令，我们还是可以看得到所有进程。说明并没有完全隔离。这是因为，像ps, top这些命令会去读/proc文件系统，所以，因为/proc文件系统在父进程和子进程都是一样的，所以这些命令显示的东西都是一样的。所以，为了创建容器内部独立的进程空间，我们还需要对文件系统进行隔离。

mount namespace 将一个进程放到一个特定的目录执行并允许不同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中的容器在/proc/mounts的信息只包含所在namespace的mount point。

· User namespace

使用user namespace使得每个容器有不同的 user 和 group id, 也就是说可以在容器内部用容器内部的用户执行程序而非Host上的用户。

· Network 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。

资源隔离

CGroup

容器技术有效地将由单个操作系统管路的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求。但是以上的Linux namespace主要解决的是环境隔离的问题，但这只是虚拟化中最最基础的一步，我们还需要解决对计算机资源使用上的隔离。也就是说，若仅拥有一个隔离

的环境，而不对各个容器所使用的系统资源进行隔离，在各个容器中的进程使用CPU、内存、磁盘等这些计算资源其实还是可以随心所欲的。所以，我们希望对进程进行资源利用上的限制或控制，这就要用到Linux内核提供的CGroup技术。

Linux CGroup全称Linux Control Group， 是Linux内核的一个功能，用来限制，控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）。

CGroup实现了从单个进程的资源控制到操作系统层面的虚拟化，主要有以下功能：

- 资源限制（Resource Limitation）： cgroups可以对进程组使用的资源总额进行限制。如设定应用运行时使用内存的上限，一旦超过这个配额就发出OOM（Out of Memory）。
- 优先级分配（Prioritization）：通过分配的CPU时间片数量及硬盘IO带宽大小，实际上就相当于控制了进程运行的优先级。
- 资源统计（Accounting）： cgroups可以统计系统的资源使用量，如CPU使用时长、内存用量等等，这个功能非常适用于计费。
- 进程控制（Control）：cgroups可以对进程组执行挂起、恢复等操作。

在容器技术的实现中，CGroup的作用就是把系统运行的进程按用户自定义的群组区分，也就是一个容器使用一个group，从而达到容器间资源隔离的目的。

管理

在我的理解中，这里的管理指的是对大规模容器集群的管理。那么在容器的管理方面，我们可以参考或者使用开源的容器管理工具Kubernetes。

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

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

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

用户可以将一组 containers 以 “POD” 形式通过 Kubernetes 部署到集群之中。Kubernetes 以 “POD” 为单位管理一系列彼此联系的 Containers，这些 Containers 被部署在同一台物理主机中、拥有同样地网络地址并共享存储配额。

Kubernetes的架构被定义为由一个master服务器和多个minons服务器组成。命令行工具连接到master服务器的API端点，其可以管理和编排所有的minons服务器，Docker容器接收来自master服务器的指令并运行容器。

Master：Kubernetes API 服务所在，多Master的配置仍在开发中。

Minons：每个具有Kubelet服务的Docker主机，Kubelet服务用于接收来自Master的指令，且管理运行容器的主机。

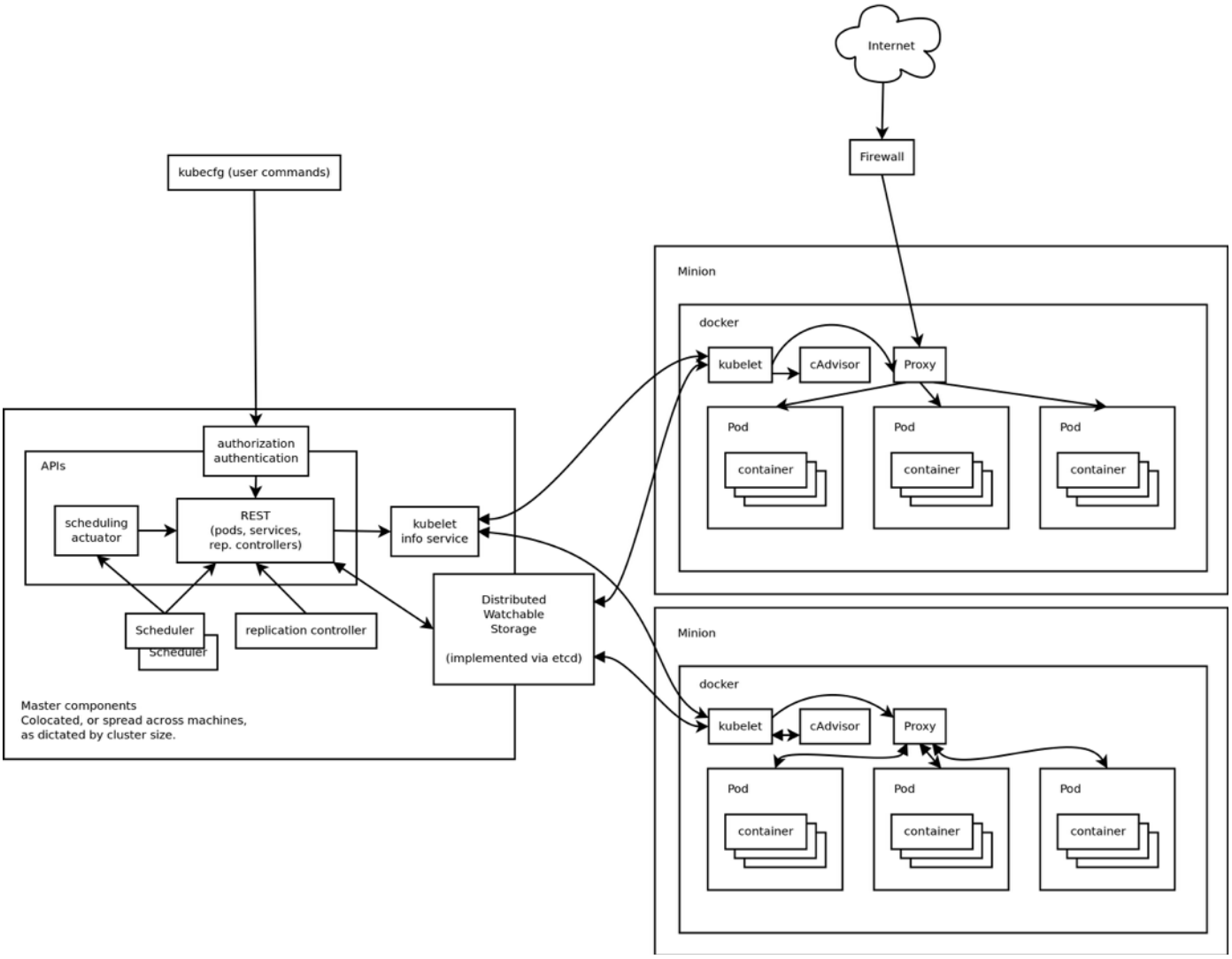
Pod：定义了一组绑在一起的容器，可以部署在同一Minons中，例如一个数据库或者是web服务器。

Replication controller：定义了需要运行多少个Pod或者容器。跨多个minons来调度容器。

Service：定义了由容器所发布的可被发现的服务／端口，以及外部代理通信。服务会映射端口到外部可访问的端口，而所映射的端口是跨多个minons的Pod内运行的容器的端口。

kubecfg：命令行客户端，连接到master来管理Kubernetes。

下图是Kubernetes的架构图:


Internet
Firewall
kubecfg (user commands)
Minion
docker
kubelet
cAdvisor
Proxy
Pod
container
Pod
container
Pod
container
authorization
authentication
APIs
scheduling
actuator
REST
(pods, services,
rep. controllers)
kubelet
info service
Distributed
Watchable
Storage
(implemented via etcd)
Scheduler
Scheduler
replication controller
Master components
Colocated, or spread across machines,
as dictated by cluster size.
Minion
docker
kubelet
cAdvisor
Proxy
Pod
container
Pod
container
Pod
container

参考资料：

Docker核心技术预览：http://www.infoq.com/cn/articles/docker-core-technology-preview

Docker基础技术：Linux namespace：http://coolshell.cn/articles/17010.html

http://coolshell.cn/articles/17029.html

Docker基础技术：Linux CGroup：http://coolshell.cn/articles/17049.html

LXC维基百科：https://zh.wikipedia.org/wiki/LXC

Linux UTS：http://dockone.io/article/76

CGroup维基百科：https://zh.wikipedia.org/wiki/Cgroups

http://www.infoq.com/cn/articles/docker-kernel-knowledge-cgroups-resource-isolation

Kubernetes容器管理技术变迁：http://www.csdn.net/article/1970-01-01/2826140

Kubernetes系统架构简介：http://www.infoq.com/cn/articles/Kubernetes-system-architecture-introduction

Docker背后的容器集群管理——从Borg到Kubernetes：http://www.infoq.com/cn/articles/docker-container-cluster-management-part-01?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk