软件复用讨论课三

徐锦程 1353012

参考资料

http://baike.so.com/doc/8850626-9175652.html

<http://www.wtoutiao.com/p/X6fDUr.html>

<http://www.freebuf.com/articles/system/69809.html>

http://tech.chinaso.com/ejyjs/detail/20160303/1000200032857121456967201146855212\_1.html

http://toutiao.com/i6255090247101579777/

侧重于容器技术，讨论各种方案以及挑战

– 容器技术的理解

› 关键技术

- 隔离

- 安全

› 挑战

- 管理(Orchestrating at Scale)

– 如何复用

› 我们需要和希望解决的问题

› 选取什么样的容器技术方案

**一、容器技术**

**1、概念**

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口(类似 iPhone 的 app)。几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。

但是，不同的人对于Docker带来的价值有不同的理解。我们认为Docker带来了以下两个非常重要的价值：一个轻量级、可独立调度的运行环境以及一个标准的镜像模型。

如果观察现在容器相关的技术和产品基本也对应到这两点上。一部分容器技术与产品关注使用Docker来简化运行环境管理，打通从代码到容器镜像的整条链路。另一部分则关注如何更好的管理和编排容器运行时环境，让容器真正能够达到服务高可用、强弹性的业务场景。当然，还有连接两者的Docker Registry。

容器技术落地的一个重要产品方向就是“容器云”。一般来说，一个完整的容器云需要包括以上说到的三方面产品（镜像发布、镜像仓库和容器编排）。

说到“容器云”，就必然要联系到另一个云，即基础设施云（IaaS）。那这两者之间到底是什么关系？一般认为，“容器云”是一种特化（技术路线上）的平台云（PaaS），按照云计算普遍采纳的三层架构，其应该在IaaS之上。这也是我们今天要表达的第一个观点：我们认为只有运行在IaaS平台上的“容器云”才能够算真正意义上的容器云。直接运行在物理机以及虚拟化环境上的“容器云”则缺少底层基础设施弹性交付的能力，并不能拥有“云”最核心的优势：弹性交付，按需付费。当然，IaaS平台提供的丰富基础设施服务也为容器云的运营带来很多价值。例如，IaaS提供的云磁盘，云存储可以作为Kubernetes中pod的存储的一个选择。区别于宿主机硬盘，云磁盘在数据可靠性要好非常多。

类似与IaaS，容器云也可以有公有容器云、托管容器云或者私有容器云之分。但是无论那种形态，我们都认为其应该运行在某种IaaS层之上（可以是公有IaaS云：如青云QingCloud），也可以是私有云。

目前新兴的容器云，从功能上看主要有两类：一种是直接托管一个（系统）容器，让用户自己访问并管理，看到的是一个系统；另一种是用（应用）容器托管应用，用户只提供代码，看到的是部署好的业务服务。后者是典型的 PaaS 场景，也有很多优秀的开源项目，但实际上很多企业业务在往 PaaS 上迁移的过程中挑战比较大。

**2、隔离与安全性：**

**何谓安全性？**

单单就Docker来说，安全性可以概括为两点：

不会对主机造成影响

不会对其他容器造成影响

所以安全性问题90%以上可以归结为隔离性问题。而Docker的安全问题本质上就是容器技术的安全性问题，这包括共用内核问题以及Namespace还不够完善的限制：

/proc、/sys等未完全隔离

Top, free, iostat等命令展示的信息未隔离

Root用户未隔离

/dev设备未隔离

内核模块未隔离

SELinux、time、syslog等所有现有Namespace之外的信息都未隔离

当然，镜像本身不安全也会导致安全性问题。

**真的不如虚拟机安全？**

其实传统虚拟机系统也绝非100%安全，只需攻破Hypervisor便足以令整个虚拟机毁于一旦，问题是有谁能随随便便就攻破吗？如上所述，Docker的隔离性主要运用Namespace 技术。传统上Linux中的PID是唯一且独立的，在正常情况下，用户不会看见重复的PID。然而在Docker采用了Namespace，从而令相同的PID可于不同的Namespace中独立存在。举个例子，A Container 之中PID=1是A程序，而B Container之中的PID=1同样可以是A程序。虽然Docker可透过Namespace的方式分隔出看似是独立的空间，然而Linux内核（Kernel）却不能Namespace，所以即使有多个Container，所有的system call其实都是通过主机的内核处理，这便为Docker留下了不可否认的安全问题。

传统的虚拟机同样地很多操作都需要通过内核处理，但这只是虚拟机的内核，并非宿主主机内核。因此万一出现问题时，最多只影响到虚拟系统本身。当然你可以说黑客可以先Hack虚拟机的内核，然后再找寻Hypervisor的漏洞同时不能被发现，之后再攻破SELinux，然后向主机内核发动攻击。文字表达起来都嫌繁复，更何况实际执行？所以Docker是很好用，但在迁移业务系统至其上时，请务必注意安全性！

**如何解决？**

在接纳了“容器并不是全封闭”这种思想以后，开源社区尤其是红帽公司，连同Docker一起改进Docker的安全性，改进项主要包括保护宿主不受容器内部运行进程的入侵、防止容器之间相互破坏。开源社区在解决Docker安全性问题上的努力包括：

1. Audit namespace

作用：隔离审计功能

未合入原因：意义不大，而且会增加audit的复杂度，难以维护。

2. Syslognamespace

作用：隔离系统日志

未合入原因：很难完美的区分哪些log应该属于某个container。

3. Device namespace

作用：隔离设备（支持设备同时在多个容器中使用）

未合入原因：几乎要修改所有驱动，改动太大。

4. Time namespace

作用：使每个容器有自己的系统时间

未合入原因：一些设计细节上未达成一致，而且感觉应用场景不多。

5. Task count cgroup

作用：限制cgroup中的进程数，可以解决fork bomb的问题

未合入原因：不太必要，增加了复杂性，kmemlimit可以实现类似的效果。(最近可能会被合入)

6. 隔离/proc/meminfo的信息显示

作用：在容器中看到属于自己的meminfo信息

未合入原因：cgroupfs已经导出了所有信息，/proc展现的工作可以由用户态实现，比如fuse。

不过，从08年cgroup/ns基本成型后，至今还没有新的namespace加入内核，cgroup在子系统上做了简单的补充，多数工作都是对原有subsystem的完善。内核社区对容器技术要求的隔离性，本的原则是够用就好，不能把内核搞的太复杂。

**一些企业也做了很多工作，比如一些项目团队采用了层叠式的安全机制，这些可选的安全机制具体如下：**

**1、文件系统级防护**

文件系统只读：有些Linux系统的内核文件系统必须要mount到容器环境里，否则容器里的进程就会罢工。这给恶意进程非常大的便利，但是大部分运行在容器里的App其实并不需要向文件系统写入数据。基于这种情况，开发者可以在mount时使用只读模式。比如下面几个： /sys 、/proc/sys 、/proc/sysrq-trigger 、 /proc/irq、/proc/bus

写入时复制（Copy-On-Write）：Docker采用的就是这样的文件系统。所有运行的容器可以先共享一个基本文件系统镜像，一旦需要向文件系统写数据，就引导它写到与该容器相关的另一个特定文件系统中。这样的机制避免了一个容器看到另一个容器的数据，而且容器也无法通过修改文件系统的内容来影响其他容器。

**2、Capability机制**

Linux对Capability机制阐述的还是比较清楚的，即为了进行权限检查，传统的UNIX对进程实现了两种不同的归类，高权限进程（用户ID为0，超级用户或者root），以及低权限进程（UID不为0的）。高权限进程完全避免了各种权限检查，而低权限进程则要接受所有权限检查，会被检查如UID、GID和组清单是否有效。从2.2内核开始，Linux把原来和超级用户相关的高级权限划分成为不同的单元，称为Capability，这样就可以独立对特定的Capability进行使能或禁止。通常来讲，不合理的禁止Capability，会导致应用崩溃，因此对于Docker这样的容器，既要安全，又要保证其可用性。开发者需要从功能性、可用性以及安全性多方面综合权衡Capability的设置。目前Docker安装时默认开启的Capability列表一直是开发社区争议的焦点，作为普通开发者，可以通过命令行来改变其默认设置。

**3、NameSpace机制**

Docker提供的一些命名空间也从某种程度上提供了安全保护，比如PID命名空间，它会将全部未运行在开发者当前容器里的进程隐藏。如果恶意程序看都看不见这些进程，攻击起来应该也会麻烦一些。另外，如果开发者终止pid是1的进程命名空间，容器里面所有的进程就会被全部自动终止，这意味着管理员可以非常容易地关掉容器。此外还有网络命名空间，方便管理员通过路由规则和iptable来构建容器的网络环境，这样容器内部的进程就只能使用管理员许可的特定网络。如只能访问公网的、只能访问本地的和两个容器之间用于过滤内容的容器。

**4、Cgroups机制**

主要是针对拒绝服务攻击。恶意进程会通过占有系统全部资源来进行系统攻击。Cgroups机制可以避免这种情况的发生，如CPU的cgroups可以在一个Docker容器试图破坏CPU的时候登录并制止恶意进程。管理员需要设计更多的cgroups，用于控制那些打开过多文件或者过多子进程等资源的进程。

**5、SELinux**

SELinux是一个标签系统，进程有标签，每个文件、目录、系统对象都有标签。SELinux通过撰写标签进程和标签对象之间访问规则来进行安全保护。它实现的是一种叫做MAC（Mandatory Access Control）的系统，即对象的所有者不能控制别人访问对象。

**安全建议**

**最简单的就是不要把Docker容器当成可以完全替代虚拟机的东西。跑在Docker容器中的应用在很长一段时间内都将会是选择性的，通常只跑测试系统或可信业务。**

门槛再高一点，我们对系统做减法，通过各种限制来达到安全性。这也是最主流的、有效的安全加固方法，比如上一章节介绍的几种安全机制。同时一定要保证内核的安全和稳定。外部工具的监控、容错等系统也必不可少。

总之通过适配、加固的Docker容器方案，在安全性上完全可以达到商用标准。就是可能对实施人员的技术要求和门槛较高。

**3、管理**

容器技术发展出了另外一个概念：immutable infrastructure（不可变基础架构）。所谓不可变基础架构，就是说系统一旦部署，就不再更变升级。当服务/应用需要升级时，只要部署一个新版系统，摧毁旧版就好了。在这个过程中，系统对外服务几乎是持续的。从这个概念描述中，我们很容易想到容器及相应的编排管理框架可以自然地实现immutable infrastructure。 Google的Brendan Burns有一段对Kurbenetes介绍视频，其中一个非常直观的演示场景就是在几乎不影响对外服务的情况下如何迅速将Container封装的应用从1.0升级到1.1。这利用到了container轻量化，快速部署的特性，使得以新替旧的成本大大低于升级维护旧Container —— 听起来是个很不错的免维护的场景。

事实上，类似免维护的观念在硬件领域已非常普遍。比如电脑的显示屏不亮了，维修工程师要么换屏，要么换主板，没有人会再用仪器去检测显示芯片组了，因为这样的投入费时费工。这里，修（维护升级）还是换（去旧补新）基于很简单的成本逻辑，一旦换新的比升级旧的成本更低时，人们自然选择直接换新的。同样，Container标准化封装和快捷部署和销毁，让新应用/服务取代旧应用/服务，比研究如何给应用/服务打patch升级更为简化高效。由此，容器技术让应用的immutable infrastructure变成现实。

**二、如何复用**

**1、我们需要和希望解决的问题**

**容器在存储领域的短板：默认情况下没有永久性存储**

容器当前的吸引力来自与运行虚拟实例相关的问题和日常开支方面，即每个实例必须要有专用内存和存储资源。这些东西通常要么太多，要么太少，在需要快速扩展时，往往要花费很长的时间。

虚拟实例的设计存在孤立性，每个实例的升级都具有独特性，但是在运行相似或相同操作系统版本的大环境中，每个都运行着一套相同的程序以消费内存，保持近乎相同的启动卷。

在向“大规模计算转型”这一目标进发的征途中，传统的虚拟化被认为效率低下，浪费内存、CPU、存储、机架空间、电力、冷却等物理资源，以及浪费管理、IP地址等逻辑资源。由于它们与邻近的看起来好像容器拥有整个操作系统的组件相互隔离，因此容器带有一定程度的分离性。这种分离性允许它们与外部世界进行互动。

由于在2014年呈现指数增长，容器及其生态系统2015年在企业环境中获得了一些推动力，但是还远未达到大规模部署的程度。目前已经有少量的备份软件提供商开始提供容器备份支持，但是目前是否已经有了一种可以通过备份软件备份容器的方法呢?

与虚拟实例相比，这可能是非常短暂的，并且只适用于分配给它们的存储。与以前所使用的原始镜像相比，容器可以使用重叠式文件系统等功能执行写入时复制程序，以存储所有升级信息到容器的根文件系统。如果容器被删除了，那么这些变更也通常会丢失。因此，容器在默认情况下没有永久存储。

**Docker在存储领域的改进：使用卷和数据容器时仍存在问题**

然而，Docker提供了Docker卷和数据容器两个功能，这两个功能可以实现对更多永久存储资源的访问。

Docker卷允许数据被存储在启动卷外的容器中，根文件系统内，并且可以通过多种方式被执行。通过提供一个传递至“-v”交换机参数的共享名，容器能够被创建在一个或多个卷内。

这拥有在Docker配置文件夹(/var/lib/docker)中创建一个实体，以展现卷中内容的效果。卷中的配置数据被存储在/var /lib/docker/volumes文件夹内，通过每个子目录展现基于通用唯一标识符(UUID)的卷名。数据本身将被存储在以UUID名称为基础的 /var/lib/docker/vfs/dir文件夹中。

任何卷中的数据都能够通过主机操作系统和标准权限申请被浏览和编辑。卷的使用既有优势也有劣势。由于数据被存储在标准文件系统中，它们可以通过操作系统被备份、拷贝或是移入/移出。

不足之处是卷名采用的是UUID格式，这让人难以将它们与容器名称联系在一起。Docker让许多工作变得更加轻松，它通过提供“docker cp”命令，让文件和文件夹能够从主机目录拷贝至指定容器名称的容器目录路径。这与rsync很相似。

这为提供对在NFS共享上的外部共享存储或LUN的访问带来了可能。LUN使用卷选项访问创建在外部存储上的主机共享，不过这种方式并不推荐使用。

Docker卷也可以与主机目录联系在一起。这在“-v”交换机中被进行了详细说明，它使用了一个通过冒号将主机与容器分隔起来的格式，即：“-v /host:/container”。这种方法允许容器访问主机上的持久性数据。

因此，这使得访问存储在NFS共享上的外部共享存储，或者使用卷选项访问创建在外部存储上的主机共享的LUN成为了可能。这种方式也可用于备份由容器访问的数据。

对于管理Docker中数据的另一个选项是使用Docker数据容器。这一概念实质上是一个不活跃容器，其中创建有一个或多个卷。在启动额外容器时，这些卷能够被导出到一个或多个使用“-volumes-from”交换机的容器中。

数据卷容器实际上是作为一个内部的Docker NFS服务器，提供对来自中央挂载点的容器的访问。

这种方法的好处是它抽象了原始数据的位置，让数据容器成为了一个逻辑挂载点。在保持数据长久保存一个专用容器的同时，它们还允许访问数据容器卷的“应用”容器被创建和被破坏。

在使用卷和数据容器时我们会发现，其中存在许多问题。

**孤立的存储**

目前，在不删除相关卷的情况下，删除容器已经成为了可能。实际上，这是一种默认行为，除非明确进行重写。这使得终结没有相关容器的孤立卷变得更加容易。

清理孤立存储是一项艰巨的任务，这需要查找与容器匹配的容器配置文件和与它们相关的卷。

**安全性**

除了标准文件权限和配置只读或读写访问选项外，卷或数据容器没有额外的安全保证。这意味着容器中的用户文件访问权限需要与主机设置相匹配。

**数据完整性**

共享使用卷和数据容器的数据没有任何数据完整性保护措施。文件锁定等功能需要由容器自身进行管理。这是一项必须要加入到应用当中的额外开销。

容器不提供数据保护工具，例如快照或复制，因此数据管理必须要由主机或是容器负责。

目前还缺乏对外部存储的支持。除了主机操作系统提供的功能外，Docker内部也没有对外部存储的一些特殊支持。

容器卷默认存储在/var/lib/dockerdirectory下，这将成为容量和性能瓶颈。目前，在Docker后台程序启动中使用交换机改变这一位置已经变成了可能。

容器和存储目前存在的问题之一是，无法管理运行在独立物理主机上的容器之间的数据共享。

容器卷可位于外部存储上，但是目前的设计并不便于将来自一台主机的卷用在另一台主机上。为了解决这一问题，开始出现了类似ClusterHQ的 Flocker等解决方案，以解决卷的可移植性问题。目前，还有一些建议对Docker进行调整，增加一些关于对卷进行管理的功能。

虽然许多问题有望被快速解决，但是数据管理在短期内仍将是一个棘手的问题。

**2、选取什么样的容器技术方案**

Docker Datacenter（简称DDC）（http://www.docker.com/products/docker-datacenter）正式发布，这套可集成的端到端平台设计用于高效地将应用程序开发和管理工作由内部数据中心迁移至云环境下。

在Docker Datacenter的帮助下，企业将能够立足于内部或者虚拟私有云环境实现容器即服务（简称CaaS）的部署工作。CaaS能够提供一整套IT管理及安全应用的内容与基础设施环境，开发人员则以此为基础通过自助服务模式实现应用程序的构建与部署。

Docker Datacenter当中包含一系列领先的Docker开源项目、商用软件并同大量经过验证并受到支持配置相集成：

* Universal Control Plane (UCP) 1.0，嵌入Swarm以实现对Docker环境的管理与编排能力。
* Trusted Registry (DTR) 1.4.3，用于实现Docker镜像管理、安全保护与协作。
* Docker Engine 1.10对强大的container runtime提供商业支持。

下面我们将对其功能特性做出深入审视。

**易于设置及使用**

我们希望用户能够尽快利用Docker Datacenter打理各类实际性任务，因此我们投入大量时间以确保安装、配置与升级等流程能够快速简便地完成。DTR与UCP本身就属于Docker化应用程序，因此能够在Docker Datacenter中快速启动。而一旦投入运行，适用于UCP与DTR的Web管理员UI则开始负责后续配置工作，具体包括存储、凭证以及用户管理，且一切都可通过几次点击轻松实现。同一套UI还可作用于用户，保证他们便捷地同应用程序、repo、网络以及访问分卷进行交互。

**Docker原生配备Engine、Networking与Swarm**

Docker Datacenter支持Docker API，并可在平台中直接嵌入多种高人气Docker开源项目，具体包括Engine以及Swarm等等。这意味着应用程序开发人员能够利用一条简单的docker-compose up命令直接定义Docker Compose与UCP之间的协作。整个过程不涉及任何重写与调整——只需立足开发成果将其敏捷部署至Swarm即可。大家不仅能够实现对整个Swarm集群内的各应用、网络及分卷的可视能力与管理能力，Docker Datacenter在本质上还能够保证应用程序的可移植能力，包括由开发向生产的流程乃至跨越各网络与存储供应程序（插件）以及任意云环境（私有云与公有云）。

在以下截图当中，大家可以看到Networks作为UCP UI中的第一个类对象。我们可以直接在UI中创建网络，或者docker-compose up一个文件并在其中做出网络定义。在此之后，UCP将创建对应网络并将其显示在Networking屏幕当中。

**内置高可用性与安全性**

为了确保应用程序流水线的顺畅推进，Docker Datacenter还内置有面向应用程序环境的高可用性与安全性机制。UCP能够利用多台主机上的控制器，轻松设置以实现高可用性。一旦其中某台主机发生宕机或者故障，整体系统将继续保持Swarm集群同UCP设置、账户乃至权限状态的一致性。TLS亦会在加入该集群的同时在各Docker主机上实现自动化设置，这样大家就能够在无需额外调整的前提下在自己的Docker环境内确保安全通信。需要访问UCP的客户端能够通过用户指定型客户端绑定轻松接入——其中包含有UCP各自所需的凭证与认证密钥，进而确保对UCP之上运行的应用程序的正确管理权限。

**从开发到生产，全程配合集成化内容安全保护**

多层级的安全才是真正的安全; 从执行期到内容再到访问者身份乃至可执行的操作等等。Docker Datacenter将Docker Content Trust与DTR相结合，从而提供一整套贯穿应用程序生命周期始终的集成化内容安全保障机制。Content Trust使大家有能力通过数字化密钥进行镜像标记，而后对这些镜像签名加以验证。举例来说，中央IT团队能够创建基础镜像、对其进行标记并将它们上传至Trusted Registry受信注册表实例当中。而与DTR相集成则能够将签名状态显示在UI中以供开发人员及IT人员查阅。开发者们可以提取这些镜像，以其为基础构建应用程序并通过部署实现生产环境测试。当Content Trust被激活，环境中的Docker Engine将无法访问或者运行那些未被标记的镜像。

**贯穿整个应用程序生命周期的用户与访问管理机制**

要对运行在容器内的负载进行安全保护，首先需要通过命令来控制哪些负载能够运行在环境之内; 接下来，我们需要控制有资格对负载进行访问的用户身份，这代表着一种新的控制层级：谁有资格访问、允许其执行哪些操作、其能够访问哪些具体内容。UCP与DTR都允许大家通过GUI实现用户及团队管理，或者集成至现有LDAP/AD服务器以继承已定义的用户及群组成员。与DTR类似，UCP允许我们以基于角色的方式为团队分配指向特定容器组的权限（例如设置‘只读’以实现对容器的罗列/检查，而对设置‘全部控制’以开启/停止/删除/查看容器）。这种细粒度访问控制机制保证了每个团队都能够随时根据实际需要以适当方式访问应用程序及其资源。

**灵活选择插件、驱动程序与开放API**

每一家企业都拥有不同的系统、工具与流程。Docker Datacenter在设计上充分考虑到了当前运行环境的实际需要，并提供出色的灵活性以在无需进行应用程序代码重构的前提下对基础设施内的任意部分做出调整。举例来说，其网络插件能够帮助我们轻松利用Docker定义各应用容器网络的对接方式，同时选出特定数量的提供程序以交付底层网络基础设施。亦有多种面向存储体系的插件选项。存储驱动程序能够轻松将DTR与存储基础设施相结合，从而存储镜像及API允许我们从日志记录及监控系统中提取状态及日志等数据。这种模式建立起极具生命力的生态系统，目前已经有数百家合作伙伴为Docker用户提供各类网络、存储、监控以及工作流自动化等可行选项。