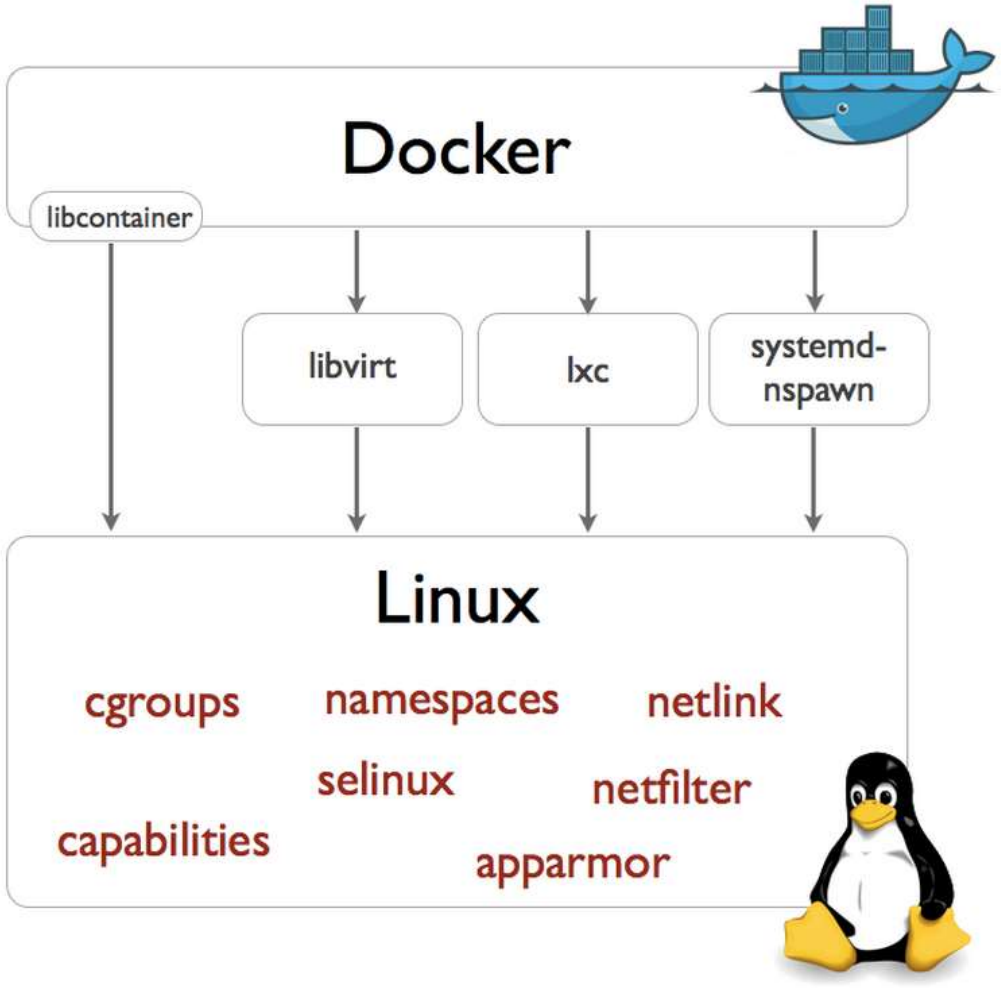
软件复用讨论课 03 – 复用云技术

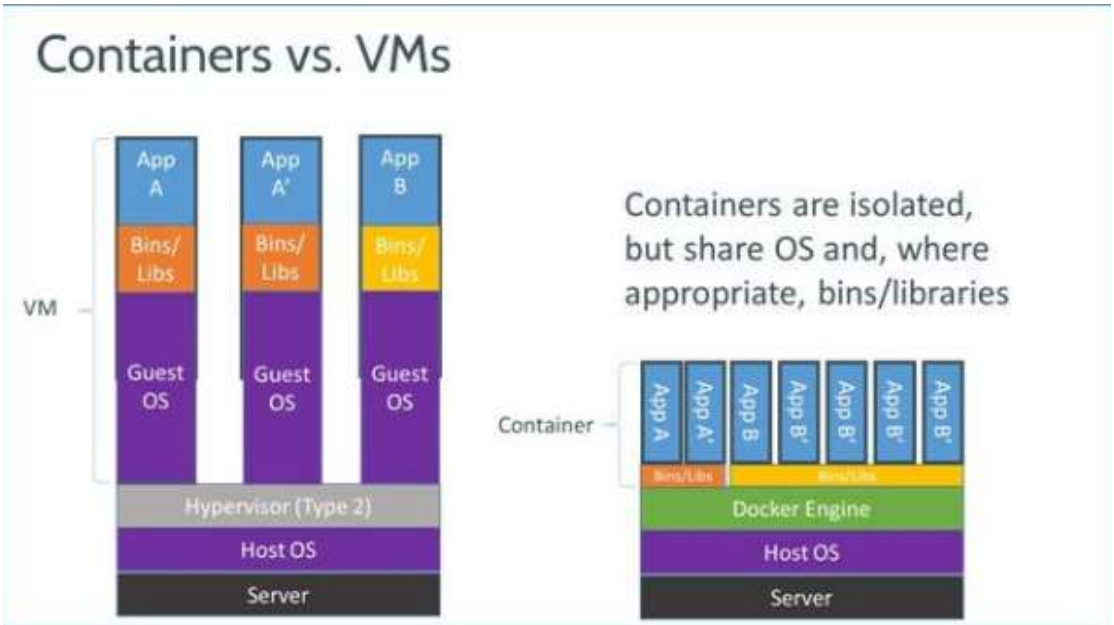
1352921 廖山河

一、容器技术的概念

1. 容器技术是一种操作系统层虚拟化技术，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容器软件（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。容器可以提供轻量级的虚拟化，以便隔离进程和资源，而且不需要提供指令解释机制以及全虚拟化的其他复杂性。
   目前容器技术代表 Docker：



2. 容器技术与虚拟化的区别：



容器为应用程序提供了隔离的运行空间：每个容器内都包含一个独享的完整用户环境空间，并且一个容器内的变动不会影响其他容器的运行环境。为了能达到这种效果，容器技术使用了一系列的系统级别的机制诸如利用 Linux namespaces 来进行空间隔离，通过文件系统的挂载点来决定容器可以访问哪些文件，通过 cgroups 来确定每个容器可以利用多少资源。此外容器之间共享同一个系统内核，这样当同一个库被多个容器使用时，内存的使用效率会得到提升。

对于系统虚拟化技术来说，虚拟层为用户提供了一个完整的虚拟机：包括内核在内的一个完整的系统镜像。CPU 虚拟化技术可以为每个用户提供一个独享且和其他用户隔离的系统环境，虚拟层可以为每个用户分配虚拟化后的 CPU、内存和 IO 设备资源。

二、关键技术—隔离与安全

1. 安全性

以 Docker 为例，安全性可以概括为两点：①不会对主机造成影响；②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：①/proc、/sys 等未完全隔离；②Top, free, iostat 等命令展示的信息未隔离；③Root 用户未隔离；④/dev 设备未隔离；⑤内核模块未隔离；⑥SELinux、time、syslog 等所有现有 Namespace 之外的信息都未隔离；⑦镜像本身不安全。

2. 与虚拟机对比

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。

3. 解决安全性的关键技术

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。

三、容器的管理

参考 Kubernetes。Kubernetes 是一款开源的项目，管理 Linux 容器集群，并可将集群作为一个单一的系统来对待。其可跨多主机来管理和运行 Docker 容器、提供容器的定位、服务发现以及复制控制。它由 Google 发起，现在得到了如微软、红帽、IBM 和 Docker 等众多厂商的支持。该项目是为了解决两个问题：一、如何跨多个 Docker 主机扩展和启动容器，且能够在主机间平衡这些容器。二、还需要高度抽象出 API 来定义如何从逻辑上组织容器、定义容器池、负载均衡以及关联性。

Kubernetes 概念：

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。

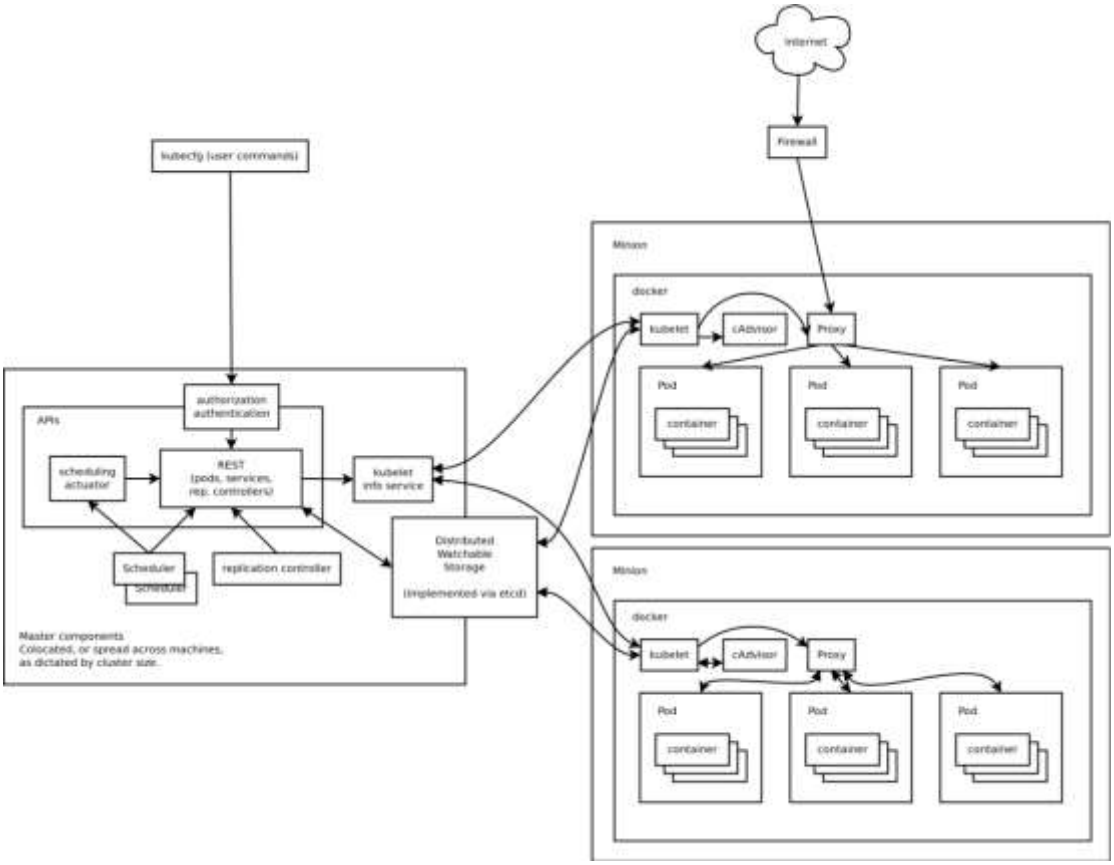
Pod：定义了一组绑在一起的容器，可以部署在同一 Minons 中，例如一个数据库或者是 web 服务器。

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。

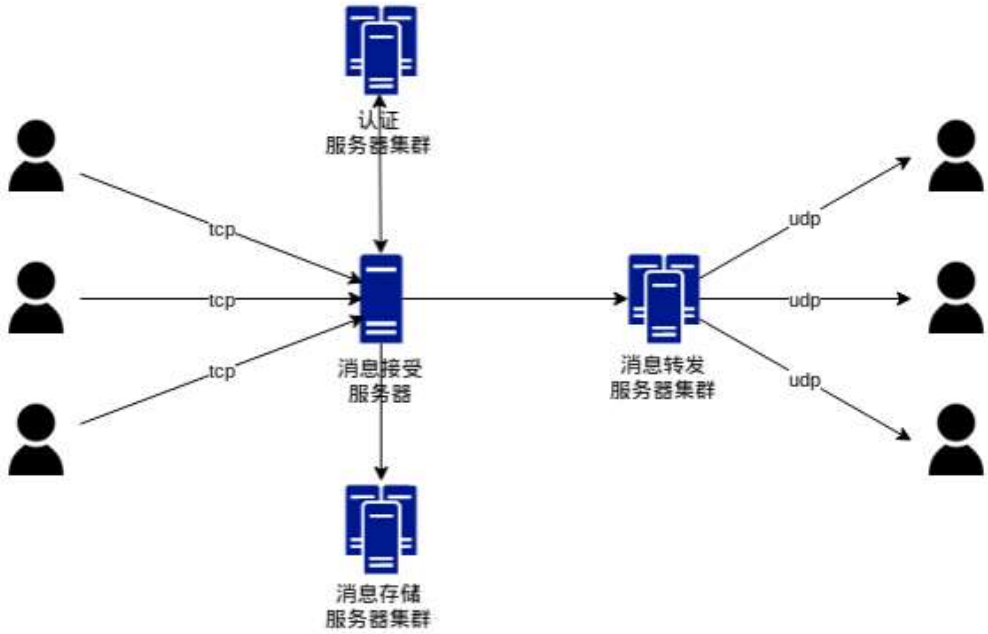
下图所示为利用 Kubernetes 管理 Docker：



Kubernetes 由状态所定义，而不是进程。当你定义了一个 pod 时，Kubernetes 会设法确保它会一直运行。如果其中的某个容器挂掉了，Kubernetes 会设法再启动一个新的容器。如果一个复制控制器定义了 3 份复制，Kubernetes 会设法一直运行这 3 份，根据需要来启动和停止容器。

四、如何复用

在上次上机实践后我们的项目的整体架构图如下所示：



因此在系统负载增加而需要增加某种服务器时，可以使用容器技术对服务器进行方便地部署与管理。