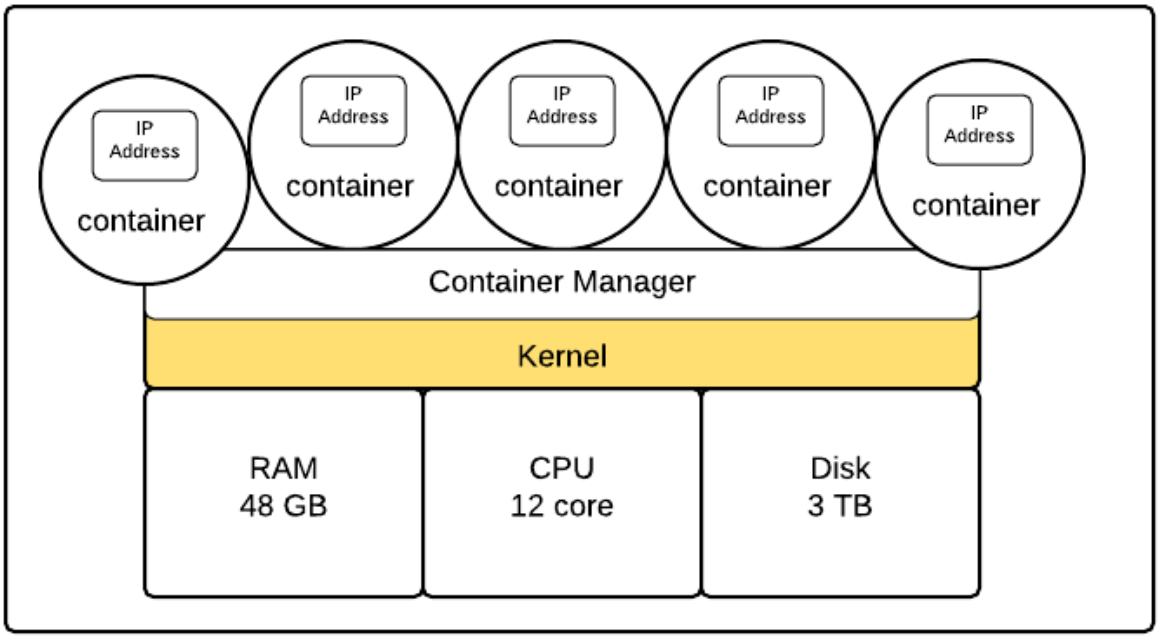
容器技术

1352837 倪雨婷

概念

概念上来说，容器是一个 Linux 进程，Linux 认为它只是一个运行中的进程。该进程只知道它被告知的东西。另外，在容器化方面，该容器进程也分配了它自己的 IP 地址。在容器化方面，容器进程有它自己的 IP 地址。一旦给予了一个 IP 地址，该进程就是宿主网络中可识别的资源。然后，你可以在容器管理器上运行命令，使容器 IP 映射到主机中能访问公网的 IP 地址。建立了该映射，无论出于什么意图和目的，容器就是网络上一个可访问的独立机器，从概念上类似于虚拟机。

下面是一个示意图：



容器/进程以动态、合作的方式共享主机上的资源。如果容器只需要 1GB 内存，它就只会使用 1GB。如果它需要 4GB，就会使用 4GB。CPU 和存储空间利用也是如此。CPU、内存和存储空间的分配是动态的，和典型虚拟机的静态方式不同。所有这些资源的共享都由容器管理器来管理。

最后，容器能非常快速地启动。

因此，容器的好处是：你获得了虚拟机独立和封装的好处，而抛弃了静态资源专有的缺陷。另外，由于容器能快速加载到内存，在扩展到多个容器时你能获得更好的性能。

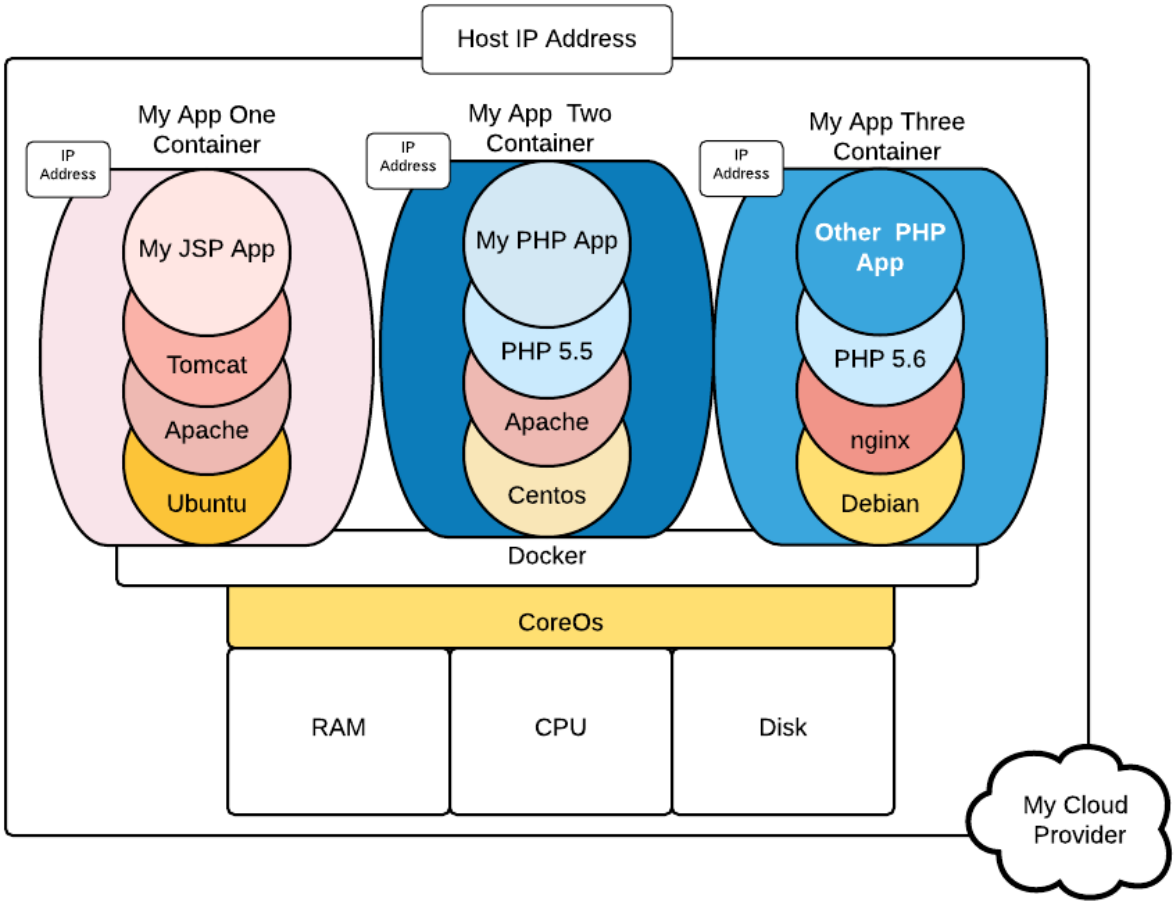
隔离

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。

管理

托管容器的计算机运行着被剥离的只剩下主要部分的某个 Linux 版本。现在，宿主计算机流行的底层操作系统是之前提到的 CoreOS。当然还有其它，例如 Red Hat Atomic Host 和 Ubuntu Snappy。

该 Linux 操作系统被所有容器所共享，减少了容器足迹的重复和冗余。每个容器只包括该容器特有的部分。下面是一个示意图：



你可以用它所需的组件来配置容器。一个容器组件被称为层

。层是一个容器镜像，（你会在后面的部分看到更多关于容器镜像的介绍）。你从一个基本层开始，这通常是你想在容器中使用的操作系统。（容器管理器只提供你所要的操作系统在宿主操作系统中不存在的部分。）当你构建你的容器配置时，你需要添加层，例如你想要添加网络服务器时这个层就是 Apache，如果容器要运行脚本，则需要添加 PHP 或 Python 运行时环境。

分层非常灵活。如果应用程序或者服务容器需要 PHP 5.2 版本，你相应地配置该容器即可。如果你有另一个应用程序或者服务需要 PHP 5.6 版本，没问题，你可以使用 PHP 5.6 配置该容器。不像虚拟机，更改一个版本的运行时依赖时你需要经过大量的配置和安装过程；对于容器你只需要在容器配置文件中重新定义层。

所有上面描述的容器的各种功能都由一个称为容器管理器

的软件控制。现在，最流行的容器管理器是 Docker 和 Rocket。（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。）上面的示意图展示了容器管理器是 Docker，宿主操作系统是 CentOS 的主机情景。

安全

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。

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

缺陷

1、不能应用在所有场景当中

Bittman认为虽然容器技术拥有很强的兼容性，但是仍然不能完全取代现有的虚拟机环境。就像虚拟化技术刚刚出现的时候，一些传统的应用程序更加适合运行在物理环境当中一样，现在，一些应用程序并不适合运行在容器虚拟化环境当中。

比如，容器技术非常适合用于开发微服务类型的应用程序——这种方式将复杂的应用程序拆分为基本的组成单元，每个组成单元部署在独立的容器当中，之后将相关容器链接在一起，形成统一

的应用程序。可以通过增加新的组成单元容器的方式对应用程序进行扩展，而不再需要对整个应用程序进行重新开发。

但是另一方面，一些应用程序只能以统一整体的形式存在——它们在最初设计时就采用了这种方式，很难实现高扩展性和快速部署等特性。对于这种情况来说，容器技术反而会对应用负载造成限制。最好的检验方式就是进行大量试验，查看哪种现有应用程序能够通过容器技术发挥最大优势。一般来说，新的应用程序研发过程很可能从容器技术当中获益。而那些不能被容器化的应用程序仍然可以运行在传统hypervisor的全功能虚拟机当中。一位来自知名保险提供商的IT架构师表示应该放缓应用程序容器化趋势。“虽然容器技术非常具有吸引力，但是软件开发团队需要一段时间及时跟进，才能够真正地高效利用容器技术所带来的优势。”

2、难以解决依赖关系问题

大多数虚拟机都是相对独立的，每台虚拟机都包含自己的操作系统、驱动和应用程序组件。只要拥有合适的hypervisor，还可以将虚拟机迁移到其他任何虚拟化平台当中。但是对比来说，容器运行在物理操作系统之上，相互之间共享大量底层的操作系统内核、库文件以及二进制文件。Bittman进一步解释说容器之间的现有依赖关系可能会限其在服务器之间的可移植性。比如，位于Linux操作系统上的Docker容器就不能运行在当前版本的Windows Server操作系统上。

对于这种问题来说，当前的解决方案并不止一种——容器可以在数秒钟之内完成复制过程，操作系统也在不断发展，开始提供“micro OS”和“nano OS”等多种类型，提供了高稳定性以及快速重启等特性。从容器自身的角度来说其更加适合于这些环境，只要数据中心当中的其他服务器可用，仍然能够对其进行迁移。

随着操作系统的逐渐发展，这些依赖关系问题也在不断得到解决。比如，Windows Server 2016承诺同时支持Docker和原生Hyper-V容器。除了Docker之外，还有许多其他容器平台可供选择，比如LXC、Parallels Virtuozzo、Joyent、Canonical LXD、Spoon等等，VMware也有可能随时加入到竞争行列中来。

3、较差的隔离性

基于hypervisor的虚拟机拥有完善的隔离特性，由于系统硬件资源完全是虚拟的，由hypervisor分配给虚拟机使用，因此bug、病毒或者入侵有可能影响一台虚拟机，但是不会蔓延到其他虚拟机上。

容器的隔离性较差因为其共享同一个操作系统内核以及其他组件，在开始运行之前就已经获得了统一的底层授权（对于Linux环境来说通常是root权限）。因此，漏洞和攻击更加有可能进入到底层的操作系统，或者转移到其他容器当中——潜在的传播行为远比最初的事件更加严重。

尽管容器平台也在不断发展，开始隔离操作系统权限、减少脆弱的安全特性等，但是Bittman仍然推荐管理员通过在虚拟机当中运行容器来提升安全性。比如，可以在Hyper-V当中部署一台Linux虚拟机，在Linux虚拟机当中安装Docker容器。这样即便虚拟机当中的容器出现问题，这种漏洞也只存在于当前虚拟机当中——限制了潜在的受攻击范围。

4、潜在的蔓延问题

就像虚拟机生命周期管理对于hypervisor环境来说十分重要一样，生命周期管理对于容器来说也是至关重要的。容器可以被大量快速复制，这是容器技术的重要优势之一，但是也有可能在管理员没有注意到的情况下消耗大量计算资源。如果应用程序所在的容器不再使用时能够被及时删除，那么情况还不算太坏。但是如果对一个容器化应用程序进行扩展之后忘记将其缩减回之前的规模，那么将会为企业带来大量的（并且不必要的）云计算开销。Bittman还表示云提供商十分高兴看到这种情况发生——因为他们就是通过出租计算资源而获利的——因此用户需要自己关注容器的部署情况。

5、缺乏工具

对于这个行业来说，用于监控和管理容器的工具种类仍然十分缺乏。这并不是一种最近产生的现象，在基于hypervisor虚拟化的早期也曾经出现过可用工具十分匮乏的情况。就像优秀的虚拟机监

控和管理工具逐渐增多一样，容器管理领域也在不断出现新的工具。其中包括谷歌的开源Docker管理工具Kubernetes，此外DockerUI使用基于web的前端界面替换Linux的命令行功能，Logspout能够将容器日志汇集到一个集中位置。

Bittman建议管理员可以通过将容器运行在虚拟机当中缓解容器管理工具缺乏的问题，这样就可以使用虚拟机工具来完成一些监控和管理功能了。因为虚拟机工具更加成熟和多样化，因此在容器工具逐渐成熟之前，可以将其作为临时的替代产品。

Bittman对于容器技术充满热情，认为其能够快速交付轻量级的应用程序，提升资源使用效率和扩展性；容器自身（非虚拟化I/O）还能够实现更好的性能表现；已经拥有像Docker这样优秀的开发架构，像GitHub这样吸引广泛关注的共享和协作平台。但是容器并不是一种能够满足所有虚拟化任务的解决方案，只是虚拟化工具箱提供的另外一种工具——通常可以和传统虚拟机很好地协同工作。

参考资料

1.使用容器技术时务必当心的几个问题
http://www.jifang360.com/news/2016225/n319278557.html
2.虚拟化 VS 容器化
http://www.oschina.net/news/61820/virtualization-vs-containerization
3.一位开发者的 Linux 容器之旅
https://linux.cn/article-6594-1.html
4.警惕容器技术的五大缺陷
http://cloud.51cto.com/art/201507/483300.htm