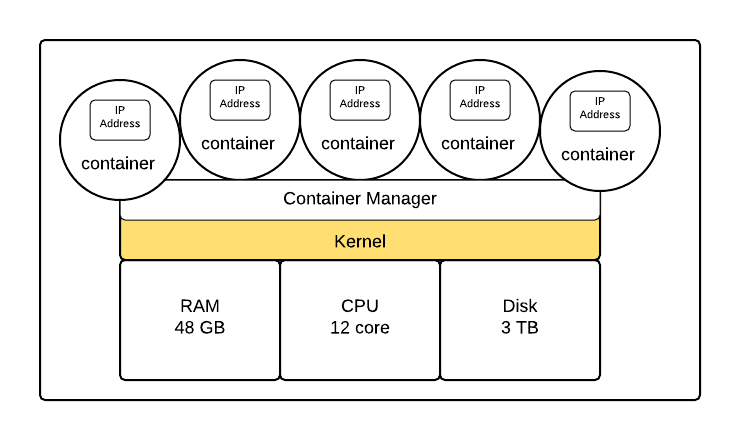
容器技术

1.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。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，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。

容器是拥有不同 IP 地址从而使其成为网络上可识别的独立 Linux 进程。下面是一个示意图：

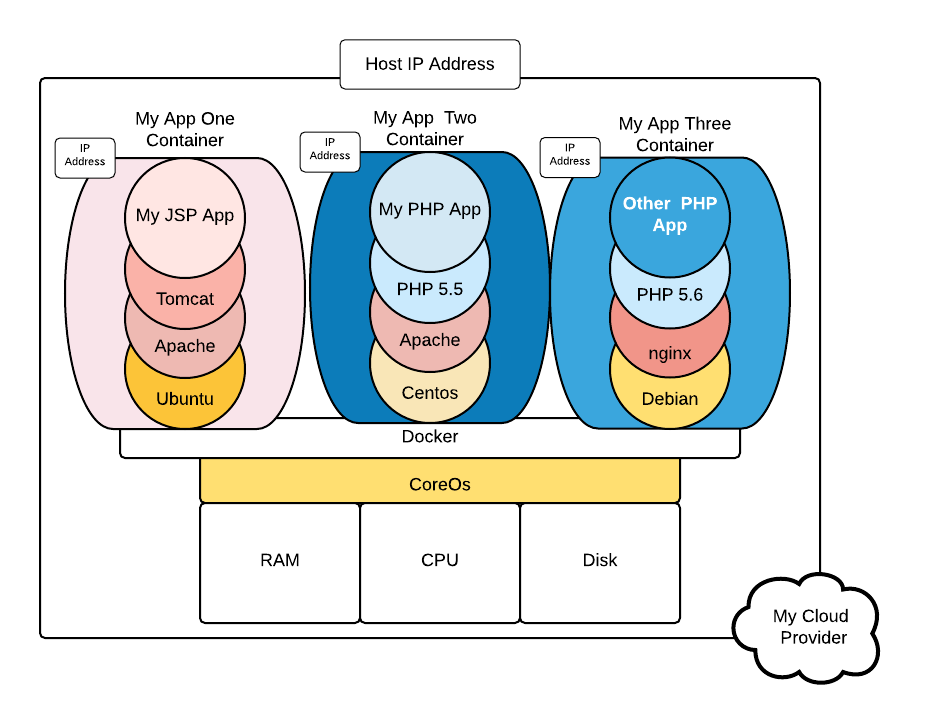


容器/进程以动态、合作的方式共享主机上的资源。如果容器只需要 1GB 内存，它就只会使用 1GB。如果它需要 4GB，就会使用 4GB。CPU 和存储空间利用也是如此。CPU、内存和存储空间的分配是动态的，和典型虚拟机的静态方式不同。所有这些资源的共享都由容器管理器来管理。最后，容器能非常快速地启动。

因此，容器的好处是：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能获得更好的性能。

2管理

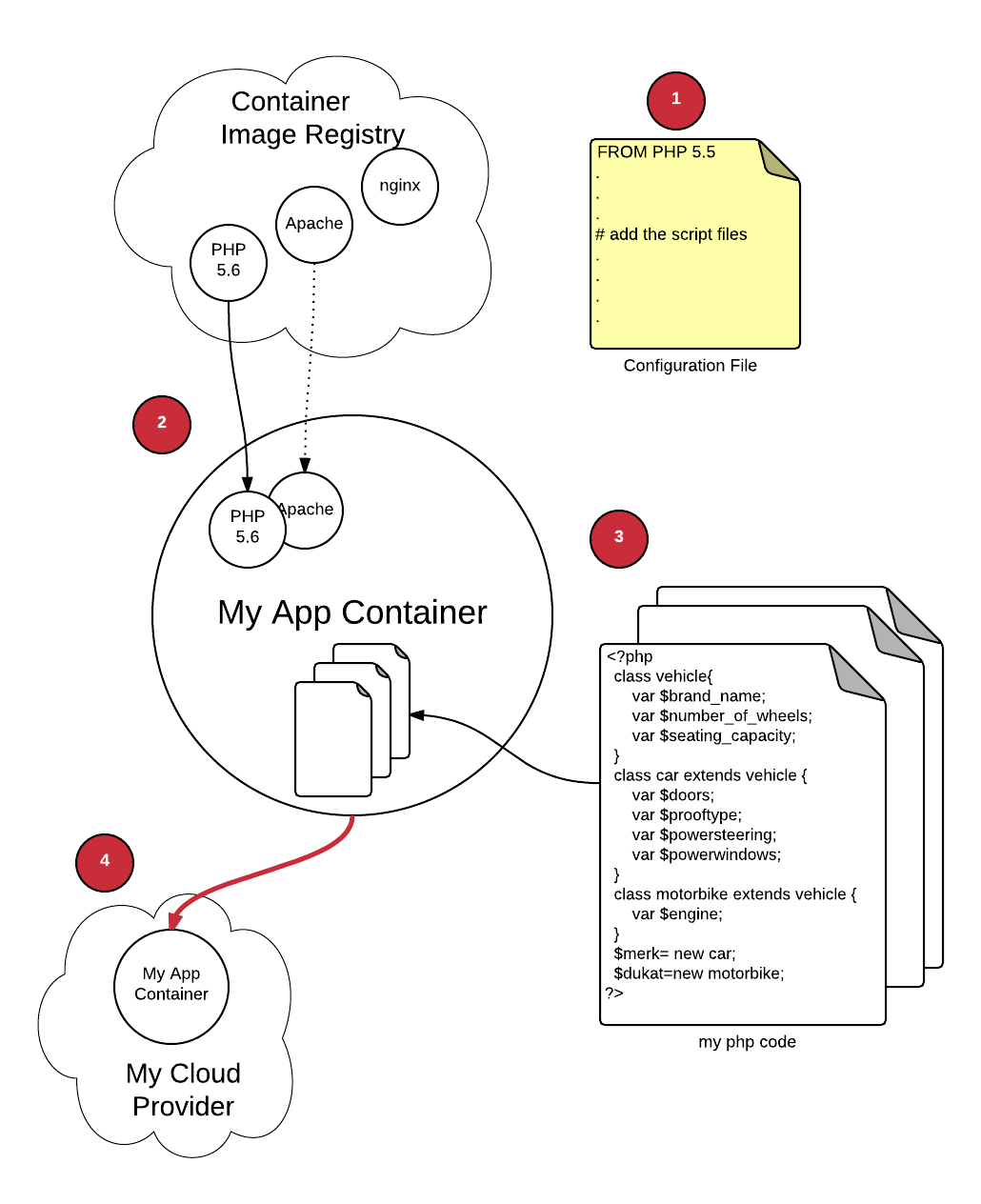
托管容器的计算机运行着被剥离的只剩下主要部分的某个 Linux 版本。现在，宿主计算机流行的底层操作系统是之前提到的CoreOS。当然还有其它，例如Red Hat Atomic Host和Ubuntu Snappy。该 Linux 操作系统被所有容器所共享，减少了容器足迹的重复和冗余。每个容器只包括该容器特有的部分。下面是一个示意图：



可以用它所需的组件来配置容器。一个容器组件被称为层layer。层是一个容器镜像。从一个基本层开始，这通常是想在容器中使用的操作系统。（容器管理器只提供所要的操作系统在宿主操作系统中不存在的部分。）当构建容器配置时，需要添加层，例如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，则相应地配置该容器即可。如果有另一个应用程序或者服务需要 PHP 5.6 版本，那么可以使用 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需要经过大量的配置和安装过程；对于容器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container manager的软件控制。现在，最流行的容器管理器是Docker和Rocket。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。

容器由镜像构成，当需要将应用程序构建到容器时，就要编译镜像。镜像代表了容器需要完成其工作的容器模板。镜像存储在注册库registry中，注册库通过网络访问。从概念上讲，注册库类似于一个使用 Java 的人眼中的Maven仓库、使用 .NET 的人眼中的NuGet服务器。创建一个列出了应用程序所需镜像的容器配置文件。然后使用容器管理器创建一个包括了应用程序代码以及从容器注册库中下载的部分资源。例如，如果应用程序包括了一些 PHP 文件，那么容器配置文件会声明从注册库中获取 PHP 运行时环境。另外，还要使用容器配置文件声明需要复制到容器文件系统中的 .php 文件。容器管理器会封装应用程序的所有东西为一个独立容器，该容器将会在容器管理器的管理下运行在宿主计算机上。这是一个容器创建背后概念的示意图：



1. 代表一个定义了容器所需东西以及容器如何构建的容器配置文件。当在主机上运行容器时，容器管理器会读取该配置文件，从云上的注册库中获取需要的容器镜像。
2. 将镜像作为层添加到容器中。另外，如果组成镜像需要其它镜像，容器管理器也会获取这些镜像并把它们作为层添加进来。
3. 容器管理器会将需要的文件复制到容器中。如果使用了配置provisioning服务，例如Deis，则刚刚创建的应用程序容器做成镜像。
4. 配置服务会将它部署到选择的云供应商上，比如类似 AWS 和 Rackspace 云供应商。
5. 安全性

单单就Docker来说，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：

/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

内核社区对容器技术要求的隔离性，本的原则是够用就好，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。

1. 注意
2. 不要在容器中存储数据

容器可能被停止，销毁，或替换。一个运行在容器中的程序版本1.0，应该很容易被1.1的版本替换且不影响或损失数据。有鉴于此，如果你需要存储数据，请存在卷中，并且注意如果两个容器在同一个卷上写数据会导致崩溃。确保你的应用被设计成在共享数据存储上写入。

1. 不要将你的应用发布两份

一些人将容器视为虚拟机。他们中的大多数倾向于认为他们应该在现有的运行容器里发布自己的应用。在开发阶段这样是对的，此时你需要不断地部署与调试；但对于质量保证与生产中的一个连续部署的管道，你的应用本该成为镜像的一部分。记住：容器应该保持不变。

1. 不要创建超大镜像

一个超大镜像只会难以分发。确保你仅有运行你应用/进程的必需的文件和库。不要安装不必要的包或在创建中运行更新（yum更新）。

1. 不要使用单层镜像

要对分层文件系统有更合理的使用，始终为你的操作系统创建你自己的基础镜像层，另外一层为安全和用户定义，一层为库的安装，一层为配置，最后一层为应用。这将易于重建和管理一个镜像，也易于分发。

1. 不要为运行中的容器创建镜像

换言之，不要使用“docker commit”命令来创建镜像。这种创建镜像的方法是不可重现的也不能版本化，应该彻底避免。始终使用Dockerfile或任何其他的可完全重现的S2I（源至镜像）方法。

1. 不要只使用“最新”标签

最新标签就像Maven用户的“快照”。标签是被鼓励使用的，尤其是当你有一个分层的文件系统。你总不希望当你2个月之后创建镜像时，惊讶地发现你的应用无法运行，因为最顶的分层被非向后兼容的新版本替换，或者创建缓存中有一个错误的“最新”版本。在生产中部署容器时应避免使用最新。

1. 不要在单一容器中运行超过一个进程

容器能完美地运行单个进程（http守护进程，应用服务器，数据库），但是如果你不止有一个进程，管理、获取日志、独立更新都会遇到麻烦。

1. 不要在镜像中存储凭据。使用环境变量

不要将镜像中的任何用户名/密码写死。使用环境变量来从容器外部获取此信息。有一个不错的例子是postgres镜像。

1. 使用非root用户运行进程

“docker容器默认以root运行。（…）随着docker的成熟，更多的安全默认选项变得可用。现如今，请求root对于其他人是危险的，可能无法在所有环境中可用。你的镜像应该使用USER指令来指令容器的一个非root用户来运行。”

1. 不要依赖IP地址

每个容器都有自己的内部IP地址，如果你启动并停止它地址可能会变化。如果你的应用或微服务需要与其他容器通讯，使用任何命名与（或者）环境变量来从一个容器传递合适信息到另一个。