Docker 的一些介绍

一、Docker简介

Docker是一个由GO语言开发的，基于LXCs(Linux containers)的高级容器引擎，实现了一种应用程序级别的隔离，源代码托管在Github上,遵从Apache2.0协议开源。

Container技术并非Docker的创新，很多云服务商都采用了类似这种轻量级的虚拟化技术，但Docker是第一个将这这种Container技术大规模开源并被社区广泛接受的。

Docker使用以下操作系统的功能来提高容器技术的效率：

原理图

> Namespace 充当隔离的第一级。确保一个容器中运行一个进程而且不能看到或影响容器外的其它进程。

> Control Groups是LXC的重要组成部分，具有资源核算与限制的关键功能，实现了对资源的配额和度量。

> UnionFS（文件系统）作为容器的构建块。为了支持Docker的轻量级以及速度快的特性，它创建了用户层。

二、Docker自身的组件与元素

图1.运行示意图

Docker engine

图2.组件架构图

Docker architecture

几个概念：

1. Docker Client & Docker Daemon

Docker使用客户端-服务器(client-server)架构模式。Docker客户端会与Docker守护进程进行通信。Docker守护进程会处理复杂繁重的任务，例如建立、运行、发布你的Docker容器。Docker客户端和守护进程可以运行在同一个系统上，当然你也可以使用Docker客户端去连接一个远程的Docker守护进程。Docker客户端和守护进程之间通过socket或者RESTful API进行通信。

2. Docker镜像

Docker 镜像是 Docker 容器运行时的只读模板，每一个镜像由一系列的层 (layers) 组成。

正因为有了这些层的存在，Docker 是如此的轻量。当你改变了一个 Docker 镜像，比如升级到某个程序到新的版本，一个新的层会被创建。

因此，不用替换整个原先的镜像或者重新建立(在使用虚拟机的时候你可能会这么做)，只是一个新的层被添加或升级了。不用重新发布整个镜像，只需要升级，层使得分发 Docker 镜像变得简单和快速。

5. Docker容器

Docker 容器和文件夹很类似，一个Docker容器包含了所有的某个应用运行所需要的环境。每一个 Docker 容器都是从 Docker 镜像创建的。

Docker 容器可以运行、开始、停止、移动和删除。每一个 Docker 容器都是独立和安全的应用平台，Docker 容器是 Docker 的运行部分。

4. Docker仓库

类似于Github的一种代码仓库，同样，Docker 仓库也有公有和私有的概念。公有的 Docker 仓库名字是 Docker Hub。Docker Hub 提供了庞大的镜像集合供使用。这些镜像可以是自己创建，或者在别人的镜像基础上创建。Docker 仓库是 Docker 的分发部分。

三、Docker的运行流程

举例说明，执行如下命令：

```
$ docker run -i -t ubuntu /bin/bash
```

docker client会启动，然后使用后面的run参数来通知docker daemon启动一个新容器。

这个简短的命令将会通知docker daemon以下信息：

1、容器所需的image在哪里，这里image名称是ubuntu，是一个base image。

2、当容器启动时，你想让容器初始化的动作，这里我们需要容器启动时自动切换到/bin/bash下面。

所以当我们敲下回车后，docker将会如何处理呢？

1、Pulls the ubuntu image

Docker检测image是否存在，如果本地不存在，则默认从Dock Hub下载。如果本地存在，则使用本地的image启动容器。

2、Creates a new container

Docker加载image，然后创建容器。

3、Allocates a filesystem and mounts a read-write layer

容器开始创建文件系统，并且在image上面添加可读可写的数据层。

4、Allocates a network / bridge interface

Docker开始创建网络接口，并且允许容器同主机进行关联。

5、Sets up an IP address

Docker从IP资源池中挑选一个分配给容器。

6、Executes a process that you specify

Docker开始执行指定的应用或者命令

7、Captures and provides application output

Docker将执行过程当中的输出或者错误信息返回给Client。让用户可以知道当前应用执行的情况。

四、Docker的组网模式

1. host模式，使用--net=host指定

Docker使用了Linux的Namespaces技术来进行资源隔离，如PID Namespace隔离进程，Mount Namespace隔离文件系统，Network Namespace隔离网络等。一个Network Namespace提供了一份独立的网络环境，包括网卡、路由、Iptable规则等都与其他的Network Namespace隔离。一个Docker容器一般会分配一个独立的Network Namespace。

如果启动容器的时候使用host模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

2. container模式，使用--net=container:NAME_or_ID指定

这个模式指定新创建的容器和已经存在的一个容器共享一个Network Namespace，而不是和宿主机共享。

新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。

同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。

3. none模式，使用--net=none指定

在这种模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等。

4. bridge模式，使用--net=bridge指定，默认设置

Docker默认的网络设置，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker容器连接到一个虚拟网桥(如下图中的docker0)上。

4.1 Bridge模式下的拓扑

单机环境下的网络拓扑

4.2 Bridge模式下容器的通信

举一个例子说明一下。假设主机有一块网卡为eth0，IP地址为10.10.101.105/24，网关为10.10.101.254。从主机上一个IP为172.17.0.1/16的容器中ping百度（180.76.3.151）。IP包首先从容器发往自己的默认网关docker0，包到达docker0后，也就到达了主机上。然后会查询主机的路由表，发现包应该从主机的eth0发往主机的网关10.10.105.254/24。接着包会转发给eth0，并从eth0发出去（主机的ip_forward转发应该已经打开）。这时候，上面的Iptable规则就会起作用，对包做SNAT转换，将源地址换为eth0的地址。这样，在外界看来，这个包就是从10.10.101.105上发出来的，Docker容器对外是不可见的。

将容器的80端口映射到主机的80端口，对主机eth0收到的目的端口为80的tcp流量进行DNAT转换，将流量发往172.17.0.5:80，也就是我们上面创建的Docker容器。所以，外界只需访问10.10.101.105:80就可以访问到容器中得服务。（除此之外，我们还可以自定义Docker使用的IP地址、DNS等信息，甚至使用自己定义的网桥，但是其工作方式还是一样的）

说明：

Docker自身的网络功能比较简单，不能满足很多复杂的应用场景，有很多开源项目用来改善Docker的网络功能。如果应用组网场景不复杂，还是能够满足要求的。

演进中的网络模型

网络模型

Sandbox：对应一个容器中的网络环境，包括相应的网卡配置、路由表、DNS配置等。CNM很形象的将它表示为网络的『沙盒』，因为这样的网络环境是随着容器的创建而创建，又随着容器销毁而不复存在的；

Endpoint：实际上就是一个容器中的虚拟网卡，在容器中会显示为eth0、eth1依次类推；

Network：指的是一个能够相互通信的容器网络，加入了同一个网络的容器直接可以直接通过对方的名字相互连接。它的实体本质上是主机上的虚拟网卡或网桥。

5. 共享网络命名空间

共享网络

当容器共享其他容器的网络命名空间，则在这两个容器之间不存在网络隔离，而她们又与宿主机以及除此之外其他的容器存在网络隔离

在这种模式下的容器可以通过localhost来同一网络命名空间下的其他容器，传输效率较高。而且这种模式还节约了一定数量的网络资源，但它并没有改变容器与外界通信的方式。

在一些特殊的场景中非常有用，例如，kubernetes的pod，kubernetes为pod创建一个基础设施容器，同一pod下的其他容器都以其他容器模式共享这个基础设施容器的网络命名空间，相互之间以localhost访问，构成一个统一的整体。

五、Docker运行时资源限制

Docker基于Linux内核提供的cgroups功能，可以限制容器在运行时使用到的资源，比如内存、CPU、块I/O、网络等。

1. 内存限制

docker 提供的内存限制功能有以下几点：

- 容器能使用的内存和交换分区大小。
- 容器的核心内存大小。
- 容器虚拟内存的交换行为。
- 容器内存的软性限制。
- 是否杀死占用过多内存的容器。
- 容器被杀死的优先级

一般情况下，达到内存限制的容器过段时间后就会被系统杀死。

2.CPU限制

Docker的资源限制和隔离完全基于linux cgroups。对CPU资源的限制方式也和cgroups相同。

Docker提供的CPU资源限制选项可以在多核系统上限制容器能利用哪些CPU，而对容器最多能使用的 CPU 时间有两种限制方式：

- 有多个 CPU 密集型的容器竞争 CPU 时，设置各个容器能使用的 CPU 时间相对比例。
- 以绝对的方式设置容器在每个调度周期内最多能使用的 CPU 时间。

3.磁盘IO配额控制

相对于CPU和内存的配额控制，docker对磁盘IO的控制相对不成熟，大多数都必须在有宿主机设备的情况下使用。主要包括以下参数：

- –device-read-bps：限制此设备上的读速度（bytes per second），单位可以是kb、mb或者gb。
- –device-read-iops：通过每秒读IO次数来限制指定设备的读速度。
- –device-write-bps ：限制此设备上的写速度（bytes per second），单位可以是kb、mb或者gb。
- –device-write-iops：通过每秒写IO次数来限制指定设备的写速度。
- –blkio-weight：容器默认磁盘IO的加权值，有效值范围为10-100。
- –blkio-weight-device： 针对特定设备的IO加权控制。其格式为DEVICE_NAME:WEIGHT

4.磁盘空间限制

在docker使用devicemapper作为存储驱动时，默认每个容器和镜像的最大大小为10G。如果需要调整，可以在daemon启动参数中，使用dm.basesize来指定，但需要注意的是，修改这个值，不仅仅需要重启docker daemon服务，还会导致宿主机上的所有本地镜像和容器都被清理掉。

使用aufs或者overlay等其他存储驱动时，没有这个限制。

六、与虚拟机的对比

1. 实现原理

对比	对比

通过docker和虚拟机实现原理的比较，我们大致可以得出一些结论：

> 1. docker有着比虚拟机更少的抽象层。由于docker不需要Hypervisor实现硬件资源虚拟化，运行在

> docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上docker将会在效率上有优势，具体的效率对比在下几个小节里给出。在IO设备虚拟化上，docker的镜像管理有多种方案，比如利用Aufs文件系统或者Device Mapper实现docker的文件管理，各种实现方案的效率略有不同。

2. docker利用的是宿主机的内核，而不需要Guest OS。因此，当新建一个容器时，docker不需要和虚拟机一样重新加载一个操作系统内核。我们知道，引导、加载操作系统内核是一个比较费时费资源的过程，当新建一个虚拟机时，虚拟机软件需要加载Guest OS，这个新建过程是分钟级别的。而docker由于直接利用宿主机的操作系统，则省略了这个过程，因此新建一个docker容器只需要几秒钟。另外，现代操作系统是复杂的系统，在一台物理机上新增加一个操作系统的资源开销是比较大的，因此，docker对比虚拟机在资源消耗上也占有比较大的优势。事实上，在一台物理机上我们可以很容易建立成百上千的容器，而只能建立几个虚拟机。

2. 计算效率

计算效率

可见docker相对于物理机其计算能力几乎没有损耗，而虚拟机对比物理机则有着非常明显的损耗。虚拟机的计算能力损耗在50%左右。

为什么会有这么大的性能损耗呢？

> 一方面是因为虚拟机增加了一层虚拟硬件层，运行在虚拟机上的应用程序在进行数值计算时是运行在Hypervisor虚拟的CPU上的

> 另外一方面是由于计算程序本身的特性导致的差异。虚拟机虚拟的cpu架构不同于实际cpu架构，数值计算程序一般针对特定的cpu架构有一定的优化措施，虚拟化使这些措施作废，甚至起到反效果。

3. 内存访问效率

内存访问

内存访问效率的比较相对比较复杂一点，主要是内存访问有多种场景：

> （1）大批量的，连续地址块的内存数据读写。这种测试环境下得到的性能数据是内存带宽，性能瓶颈主要在内存芯片的性能上；

> （2）随机内存访问性能。这种测试环境下的性能数据主要与内存带宽、cache的命中率和虚拟地址与物理地址转换的效率等因素有关。

在应用程序内存访问上，虚拟机的应用程序要进行2次的虚拟内存到物理内存的映射，读写内存的代价比docker的应用程序高。

4. 启动时间及资源耗费

无论从启动时间还是从启动资源耗费角度来说。docker直接利用宿主机的系统内核，避免了虚拟机启动时所需的系统引导时间和操作系统运行的资源消耗。利用docker能在几秒钟之内启动大量的容器，这是虚拟机无法办到的。快速启动、低系统资源消耗的优点使docker在弹性云平台和自动运维系统方面有着很好的应用前景。

5. Docker的劣势

1. 资源隔离方面不如虚拟机，docker是利用cgroup实现资源限制的，只能限制资源消耗的最大值，而不能隔绝其他程序占用自己的资源。

2. 安全性问题。docker目前并不能分辨具体执行指令的用户，只要一个用户拥有执行docker的权限，那么他就可以对docker的容器进行所有操作，不管该容器是否是由该用户创建。比如A和B都拥有执行docker的权限，由于docker的server端并不会具体判断docker cline是由哪个用户发起的，A可以删除B创建的容器，存在一定的安全风险。

3. docker目前还在版本的快速更新中，细节功能调整比较大。一些核心模块依赖于高版本内核，存在版本兼容问题

4. 真实投入生产，还需要多种开源组件和技术的支持，如kubernetes管理容器、etcd管理存储、应用打包技术dockerfile、容器间的网络管理flannel、私有仓库的构建、持续集成jenkins的结合、监控docker的工具等等。

七、Docker的应用场景

1. 简化配置

这是Docker公司宣传的Docker的主要使用场景。虚拟机的最大好处是能在你的硬件设施上运行各种配置不一样的平台（软件、系统），Docker在降低额外开销的情况下提供了同样的功能。它能让你将运行环境和配置放在代码中然后部署，同一个Docker的配置可以在不同的环境中使用，这样就降低了硬件要求和应用环境之间耦合度。

2. 代码流水线（Code Pipeline）

管理前一个场景对于管理代码的流水线起到了很大的帮助。代码从开发者的机器到最终在生产环境上的部署，需要经过很多的中间环境。而每一个中间环境都有自己微小的差别，Docker给应用提供了一个从开发到上线均一致的环境，让代码的流水线变得简单不少。

3. 提高开发效率

不同的开发环境中，我们都想把两件事做好。一是我们想让开发环境尽量贴近生产环境，二是我们想快速搭建开发环境。理想状态中，要达到第一个目标，我们需要将每一个服务都跑在独立的虚拟机中以便监控生产环境中服务的运行状态。然而，我们却不想每次都需要网络连接，每次重新编译的时候远程连接上去特别麻烦。这就是Docker做的特别好的地方，开发环境的机器通常内存比较小，之前使用虚拟的时候，我们经常需要为开发环境的机器加内存，而现在Docker可以轻易的让几十个服务在Docker中跑起来。

另外一个场景就是，比如要设置mips的编译环境，而宿主机是debian.有些优秀的工具只存在于gentoo平台，而gentoo又不好安装，我们使用docker只要从docker hub上面pull下镜像，就可以完美使用，这使得开发人员效率最大化。

4. 隔离应用

有很多种原因会让你选择在一个机器上运行不同的应用，比如之前提到的提高开发效率的场景等。我们经常需要考虑两点，一是因为要降低成本而进行服务器整合，二是将一个整体式的应用拆分成松耦合的单个服务（即微服务架构）。

在权衡资源隔离(比如限制应用最大内存使用量，或者资源加载隔离，彼此资源占用互相不影响等)和低消耗(虚拟化本身带来的损耗需要尽量的低)上，Docker很好的权衡了两者，即拥有不错的资源隔离能力，又有很低的虚拟化开销。

5. 整合服务器

正如通过虚拟机来整合多个应用，Docker隔离应用的能力使得Docker可以整合多个服务器以降低成本。由于没有多个操作系统的内存占用，以及能在多个实例之间共享没有使用的内存，Docker可以比虚拟机提供更好的服务器整合解决方案。

6. 调试能力

Docker提供了很多的工具，这些工具不一定只是针对容器，但是却适用于容器。它们提供了很多的功能，包括可以为容器设置检查点、设置版本和查看两个容器之间的差别，这些特性可以帮助调试Bug。你可以在《Docker拯救世界》的文章中找到这一点的例证。

7. 多租户环境

另外一个Docker有意思的使用场景是在多租户的应用中，它可以避免关键应用的重写。我们一个特别的关于这个场景的例子是为IoT（译者注：物联网）的应用开发一个快速、易用的多租户环境。这种多租户的基本代码非常复杂，很难处理，重新规划这样一个应用不但消耗时间，也浪费金钱。使用Docker，可以为每一个租户的应用层的多个实例创建隔离的环境，这不仅简单而且成本低廉，当然这一切得益于Docker环境的启动速度和其高效的diff命令。你可以在这里了解关于此场景的更多信息。

8. 快速部署

在虚拟机之前，引入新的硬件资源需要消耗几天的时间。Docker的虚拟化技术将这个时间降到了几分钟，Docker只是创建一个容器进程而无需启动操作系统，这个过程只需要秒级的时间。这正是Google和Facebook都看重的特性。你可以在数据中心创建销毁资源而无需担心重新启动带来的开销。通常数据中心的资源利用率只有30%，通过使用Docker并进行有效的资源分配可以提高资源的利用率。

9. 从轻量级工具、持续集成演进到微服务架构

早在14年国内有一些公司，开始尝试docker，当时毕竟docker是一个新事物，很多新特性方面的优点，并没有被大大的利用起来，这个也可以理解。那时docker对一些企业的价值在于计算的轻量级，也就是对于一些计算型的任务，通过docker的形式来分发，部署快，隔离性好，这样的任务包括：消息传递，图像处理等。

14下半年到15年初，docker的价值被更大化，应用的运行，服务的托管，外界的接受度也变高，国内也出现了一些startup公司，比如DaoCloud,灵雀云等。但这些仅仅是这些公司的第一步，后续紧跟的更多的是基于代码与镜像之间的CI/CD,缩减开发测试发布的流程，这方面的实践逐渐成熟。

微服务架构的兴起。微服务会对现阶段的软件架构有一些冲击，同样也是软件系统设计方法论的内容。这些方面国外讨论的要多一些，相信这一点也会近年来多家公司发力的地方。

八、我们的需求及考虑

> - 与硬件的解耦

> - 硬件资源的最大化利用、应用与应用之间的安全隔离

- 开发与测试、外场环境的一致性
- 版本的快速分发和部署
- 微服务的引入