Docker 容器技术分享

一、初识 Docker

1.1 Docker 是什么？

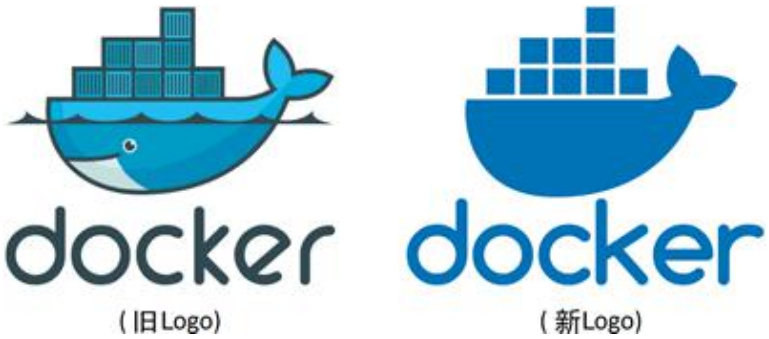
1.1.1 Docker 简介

中文版介绍：Docker 是一个用于**开发、发布和运行**应用程序的开放平台，基于 **Go 语言** 并遵从 Apache2.0 协议开源。**Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上**。Docker 使你能够将你的应用程序从你的基础设施中分离出来，这样你就可以快速地交付软件。使用 Docker，你可以像管理应用一样管理你的基础设施。通过利用 Docker 快速发布、测试和部署代码的方法，您可以显著减少编写代码和在生产环境中运行代码之间的延迟。

官方文档

1.1.2 Docker 历史由来

Docker 公司位于旧金山，由法裔美籍开发者和企业家 Solumon Hykes 创立，其标志如下图所示。



（旧Logo） （新Logo）

有意思的是，Docker 公司起初是一家名为 dotCloud 的平台即服务（Platform-as-a-Service, PaaS）提供商。

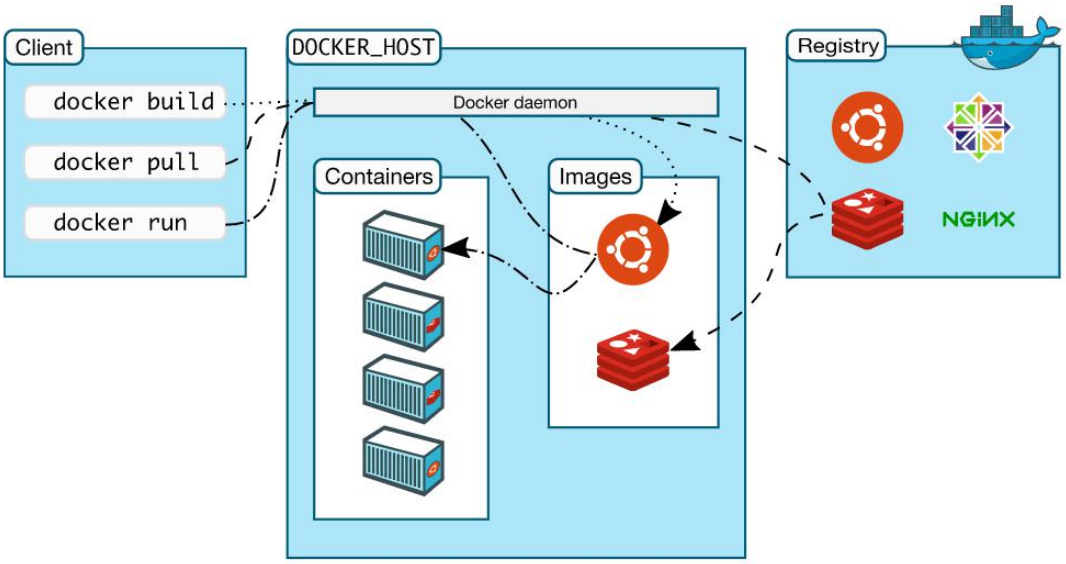
底层技术上，dotCloud 平台利用了 Linux 容器技术。为了方便创建和管理这些容器，dotCloud 开发了一套内部工具，之后被命名为“Docker”。Docker 就是这样诞生的！

2013 年，dotCloud 的 PaaS 业务并**不景气**，公司需要寻求新的突破。于是他们聘请了 Ben Golub 作为新的 CEO，将公司重命名为“Docker”，放弃 dotCloud PaaS 平台，怀揣着“将 Docker 和容器技术推向全世界”的使命，开启了一段新的征程。

如今 Docker 公司被普遍认为是一家创新型科技公司，据说其市场价

值约为 10 亿美元。Docker 公司已经通过多轮融资，吸纳了来自硅谷的几家风投公司的累计超过 2.4 亿美元的投资。几乎所有的融资都发生在公司更名为“Docker”之后。

1.1.3 Docker 架构



Docker 采用 client-server 架构。Docker 客户端与 Docker 守护进程对话，守护进程承担构建、运行和分发 Docker 容器的繁重工作。Docker 客户端和守护进程可以运行在同一个系统上，或者你可以连接一个 Docker 客户端到一个远程 Docker 守护进程。Docker 客户端和守护进程通过 UNIX 套接字或网络接口使用 REST API 进行通信。

docker 客户端和守护进程间通过 socket 进行连接，docker 提供了三种连接模式：

```
unix:///var/run/docker.sock
tcp://host:port
fd://socketfd
```

其中，`unix:///var/run/docker.sock` 是默认的连接方式们可以通过配置修改为其他的连接方式。

1.2 Docker 能干什么？

1.2.1 更高效的利用系统资源

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销，Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度，都要比传统虚拟机技术更高效。因此，相比虚拟机技术，**一个相同配置的主机，往往可以运行更多数量的应用**。

1.2.2 一致的运行环境

开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现 「这段代码在我机器上没问题啊」 这类问题。

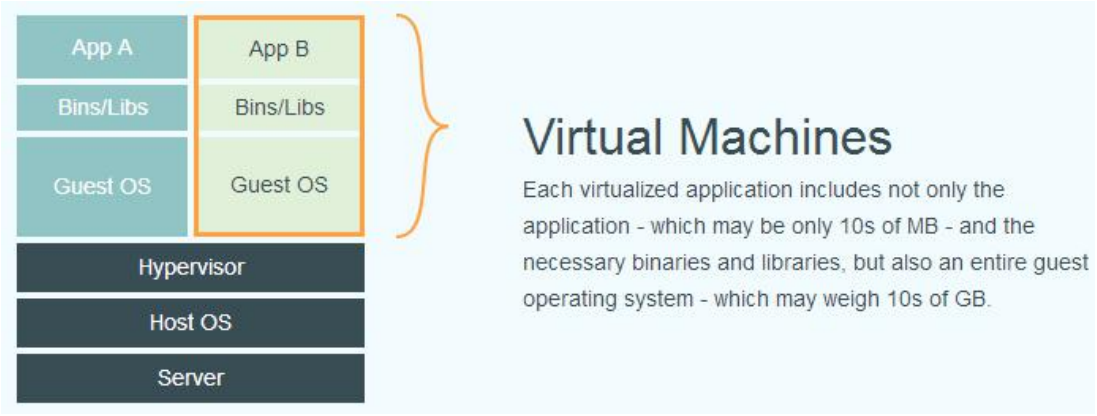
1.2.3 持续交付和部署

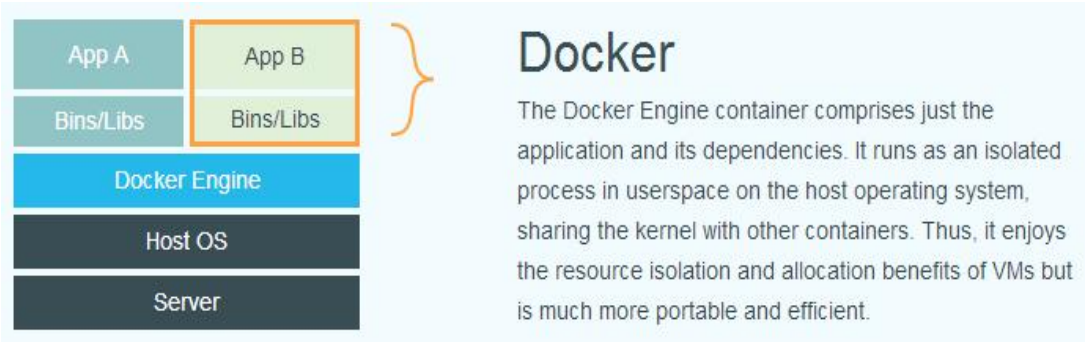
对开发和运维（DevOps）人员来说，最希望的就是一次创建或配置，可以在任意地方正常运行。使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付、部署。开发人员可以通过 Dockerfile 来进行镜像构建，并结合持续集成(Continuous Integration)系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合 持续部署(Continuous Delivery/Deployment) 系统进行自动部署。而且使用 Dockerfile 使镜像构建透明化，不仅仅开发团队可以理解应用运行环境，也方便运维团队理解应用运行所需条件，帮助更好的生产环境中部署该镜像。

1.2.4 更轻松的迁移

由于 Docker 确保了执行环境的一致性，使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。因此用户可以很轻易的将在一个平台上运行的应用，迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

1.3 VM vs Docker





1、虚拟机和容器都是在硬件和操作系统以上的，虚拟机有 Hypervisor 层，Hypervisor 是整个虚拟机的核心所在。他为虚拟机提供了虚拟的运行平台，管理虚拟机的操作系统运行。每个虚拟机都有自己的系统和系统库以及应用。

2、容器没有 Hypervisor 这一层，并且每个容器是和宿主机共享硬件资源及操作系统，那么由 Hypervisor 带来性能的损耗，在 linux 容器这边是不存在的。

3、但是虚拟机技术也有其优势，能为应用提供一个更加隔离的环境，不会因为应用程序的漏洞给宿主机造成任何威胁。同时还支持跨操作系统的虚拟化，例如你可以在 linux 操作系统下运行 windows 虚拟机。

4、从虚拟化层面来看，传统虚拟化技术是对硬件资源的虚拟，容器技术则是对进程的虚拟，从而可提供更轻量 级的虚拟化，实现进程和资源的隔离。

5、从架构来看，Docker 比虚拟化少了两层，取消了 hypervisor 层和 GuestOS 层，使用 Docker Engine 进行调度和隔离，所有应用共用主机操作系统，因此在体量上，Docker 较虚拟机更轻量级，在性能上优于虚拟化，接近裸机性能。从应用场景来 看，Docker 和虚拟化则有各自擅长的领域，在软件开发、测试场景和生产运维场景中各有优劣。

二、Docker 安装与卸载

2.1 安装 Docker

2.1.1 安装前准备

Docker 对 Linux 内核版本的最低要求是 3.10，如果内核版本低于 3.10 会缺少一些运行 Docker 容器的功能。这些比较旧的内核，在一定条件下会导致数据丢失和频繁恐慌错误。

所以在安装 Docker 前先将 Linux 内核升级到 3.10.x 或者更新的 Linux 内核版本。

查看 Linux 内核版本：`uname -r`

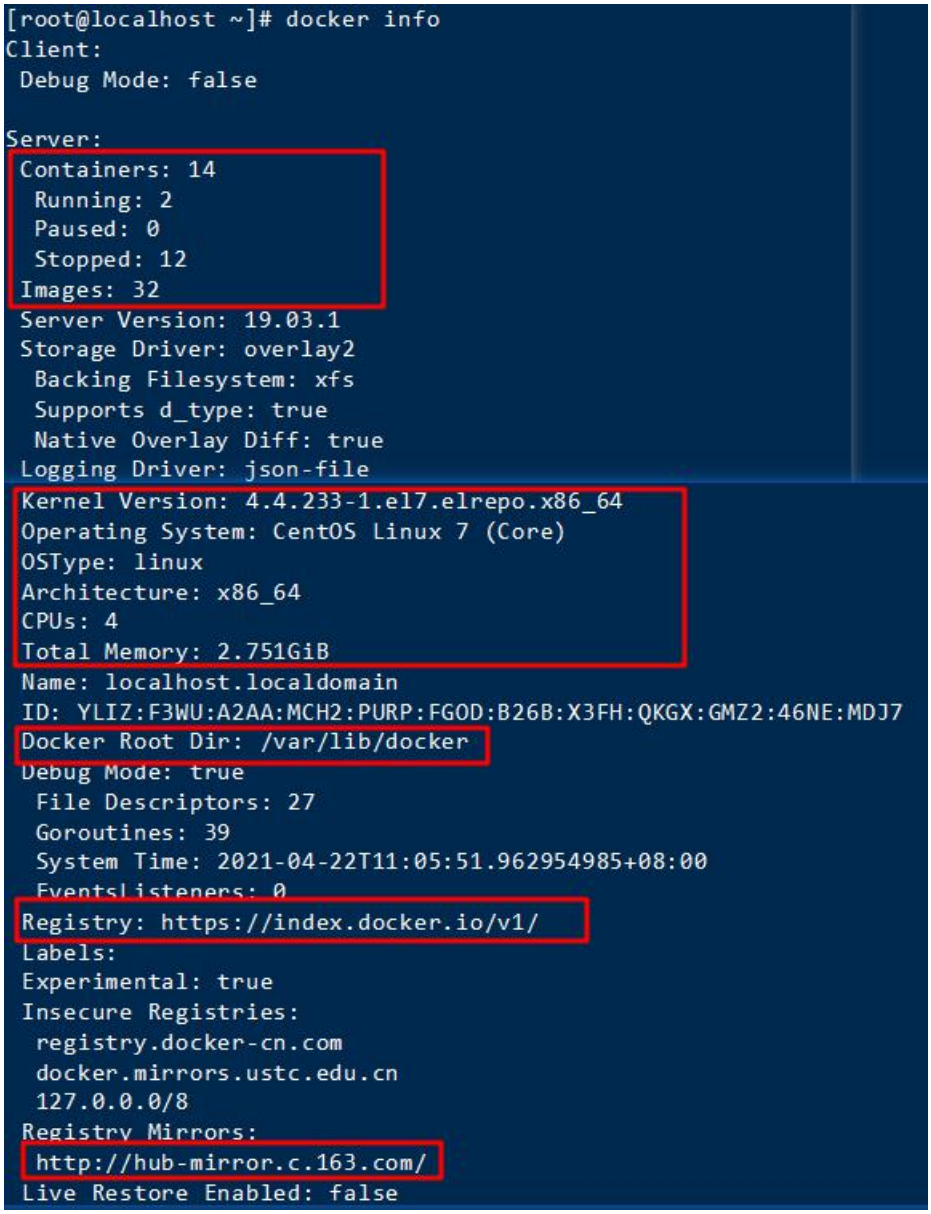
查看 Linux 发行版本：`cat /etc/redhat-release`

升级 Linux 系统内核

2.1.2 Linux 下安装 Docker

CentOS 下安装 Docker

安装并且启动了 Docker 后，通过 `docker info` 查看一下 docker 的信息：



2.2 卸载 Docker

```
yum remove docker \
        docker-client \
        docker-client-latest \
        docker-common \
        docker-latest \
        docker-latest-logrotate \
        docker-logrotate \
        docker-selinux \
        docker-engine-selinux \
        docker-engine
```

```
rm -rf /etc/systemd/system/docker.service.d

rm -rf /var/lib/docker

rm -rf /var/run/docker
```

三、镜像（Image）

3.1 Docker 镜像基本概念

Docker 镜像就是一个只读的模板。

例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。

镜像可以用来创建 Docker 容器。

Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。

3.2 Docker 镜像使用

3.2.1 拉取镜像

我们可以通过 **docker pull** 命令从镜像仓库拉取镜像。

```
12.04: Pulling from library/ubuntu
d8868e50ac4c: Pull complete
83251ac64627: Pull complete
589bba2f1b36: Pull complete
d62ecaceda39: Extracting [==================================================>]     680B/680B
6d93b41cfc6b: Download complete
```

该命令实际上相当于 `$ sudo docker pull registry.hub.docker.com/ubuntu:12.04` 命令，即从注册服务器 *registry.hub.docker.com* 中的 *ubuntu* 仓库来下载标记为 *12.04* 的镜像。

拉取好了镜像就可以使用该镜像创建一个容器：

```
[root@k8s-master1 ~]# docker run -t -i ubuntu:12.04 /bin/bash
root@2e3c6c7ca30d:/#
```

3.2.2 查看本地镜像

使用 `docker images` 显示本地已有的镜像：

```
[root@k8s-master1 ~]# docker images
REPOSITORY                                                        TAG                 IMAGE ID            CREATED             SIZE
centos                                                            latest              300e315adb2f        4 months ago        209MB
quay.io/coreos/flannel                                            v0.13.1-rc1         f03a23d55e57        4 months ago        64.6MB
registry.aliyuncs.com/google_containers/kube-apiserver            v1.16.3             df60c7526a3d        17 months ago       217MB
registry.aliyuncs.com/google_containers/kube-proxy                v1.16.3             9b65a0f78b09        17 months ago       86.1MB
registry.aliyuncs.com/google_containers/kube-controller-manager   v1.16.3             bb16442bcd94        17 months ago       163MB
registry.aliyuncs.com/google_containers/kube-scheduler            v1.16.3             98fecf43a54f        17 months ago       87.3MB
registry.aliyuncs.com/google_containers/etcd                      3.3.15-0            b2756210eeab        19 months ago       247MB
registry.aliyuncs.com/google_containers/coredns                   1.6.2               bf261d157914        20 months ago       44.1MB
registry.aliyuncs.com/google_containers/pause                     3.1                 da86e6ba6ca1        3 years ago         742kB
ubuntu                                                            12.04               5b117edd0b76        4 years ago         104MB
[root@k8s-master1 ~]#
```

通过这个命令可以获取到镜像的几个关键信息：

* REPOSITORY： 镜像仓库；当前镜像来自于哪个仓库，比如 ubuntu
* TAG：镜像的标记，比如 14.04 ，用于标记同一个仓库的不同镜像（类似镜像版本）
* IMAGE_ID：镜像的 （唯一）ID 号

* CREATED: 创建时间
* SIZE: 镜像大小

```
[root@k8s-master1 ~]# docker images
REPOSITORY                                                       TAG             IMAGE ID            CREATED             SIZE
centos                                                           latest          300e315adb2f        4 months ago        209MB
quay.io/coreos/flannel                                           v0.13.1-rc1     f03a23d55e57        4 months ago        64.6MB
registry.aliyuncs.com/google_containers/kube-proxy               v1.16.3         9b65a0f78b09        17 months ago       86.1MB
registry.aliyuncs.com/google_containers/kube-apiserver           v1.16.3         df60c7526a3d        17 months ago       217MB
registry.aliyuncs.com/google_containers/kube-controller-manager  v1.16.3         bb16442bcd94        17 months ago       163MB
registry.aliyuncs.com/google_containers/kube-scheduler           v1.16.3         98fecf43a54f        17 months ago       87.3MB
registry.aliyuncs.com/google_containers/etcd                     3.3.15-0        b2756210eeab        19 months ago       247MB
registry.aliyuncs.com/google_containers/coredns                  1.6.2           bf261d157914        20 months ago       44.1MB
registry.aliyuncs.com/google_containers/pause                    3.1             da86e6ba6ca1        3 years ago         742kB
ubuntu                                                           12.04           5b117edd0b76        4 years ago         104MB
ubuntu                                                           fl              5b117edd0b76        4 years ago         104MB
```

可以看到 TAG 为 12.04 和 fl 的镜像拥有同样的 IMAGE_ID,说明他们实际是同一个镜像。

*注：默认情况下 TAG 是 latest，表示最新的镜像。

3.2.3 创建镜像

Docker commit 命令：

我们可以基于已有的一个镜像，对齐进行修改，然后通过 `docker commit` 命令来提交更新后的镜像副本。

```
[root@k8s-master1 ~]# docker run -t -i ubuntu:12.04 /bin/bash
root@ff381b57f233:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  srv  sys  tmp  usr  var
root@ff381b57f233:/# vi abc.txt
bash: vi: command not found
root@ff381b57f233:/# mkdir abc
root@ff381b57f233:/# ls
abc  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  srv  sys  tmp  usr  var
root@ff381b57f233:/#
```

```
sudo docker commit -m "说明信息" -a "用户信息" ff381b57f233 ouruser/sinatra:v2
```

```
[root@k8s-master1 ~]# sudo docker commit -m "说明信息" -a "用户信息" ff381b57f233 ouruser/sinatra:v2
sha256:c4153fb9d0e796003036cdf41ce7719308747dbe33c16cd14a4cd69b86b7ce15
```

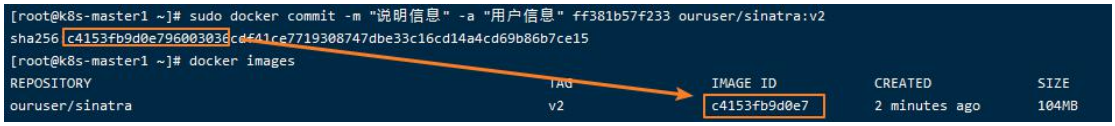
说明：

-m ：来指定提交的说明信息，跟我们使用的版本控制工具一样；

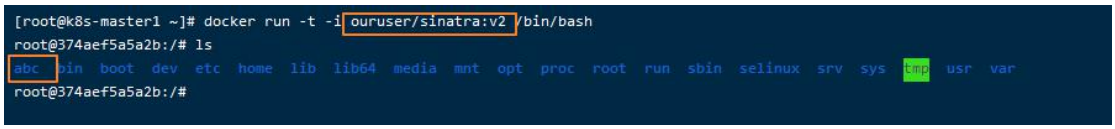
-a ：可以指定更新的用户信息；

之后是用来创建镜像的容器的 ID；最后指定目标镜像的仓库名和 tag 信息。创建成功后会返回这个镜像的 ID 信息。

通过 `docker images` 命令查看一下：



使用我们创建的镜像运行一个容器：



可以发现我们之前创建文件夹 abc 已经存在了。

虽然通过 docker commit 命令可以很简单的生成一个镜像，但是它不便于在团队协作中进行分享。

Dockerfile 方式（重要）：

我们可以通过 docker build 来创建一个新的镜像。首先需要创建一个 Dockerfile 文件；该文件中包含一些如何创建镜像的指令。

Dockerfile 语法

```
#    This is    a    comment
FROM    ubuntu:12.04
MAINTAINER    user info
RUN mkdir abc
```

```
[root@k8s-master1 ~]# vim Dockerfile
[root@k8s-master1 ~]# docker build -t="ouruser/ubuntu:v3" .
Sending build context to Docker daemon  2.967MB
Step 1/3 : FROM ubuntu:12.04
 ---> 5b117edd0b76
Step 2/3 : MAINTAINER   user info
 ---> Using cache
 ---> e3b41f42e67b
Step 3/3 : RUN     mkdir abc
 ---> Running in a5af9275099b
Removing intermediate container a5af9275099b
 ---> ce9e5483a168
Successfully built ce9e5483a168
Successfully tagged ouruser/ubuntu:v3
[root@k8s-master1 ~]#
```

```
docker build -t="ouruser/ubuntu:v3" .
```

其中 -t 标记来添加 tag，指定新的镜像的用户信息。

“.”是 Dockerfile 所在的路径（当前目录），也可以替换为一个具体的 Dockerfile 的路径（通过-f 指定具体的路径）。

可以看到 build 进程在执行操作。它要做的第一件事情就是上传这个 Dockerfile 内容，因为所有的操作都要 依据 Dockerfile 来进行。 然后 Dockfile 中的指令被一条一条的执行。每一步都创建了一个新的容器，在容器中执行指令并提交修改（就跟之前介绍过的 docker commit 一样）。当所有的指令都执行完毕之后，返回了最终的镜像 id。所有的中间步骤所产生的容器都被删除和清理了。

Dockerfile 官方文档　　最佳实践

*注意一个镜像不能超过 127 层

此外，还可以利用 ADD 命令复制本地文件到镜像；用 EXPOSE 命令来向外部开放端口；用 CMD 命令来 描述容器启动后运行的程序等。

下面通过 docker images 命令查看一下：

```
[root@k8s-master1 ~]# docker images
REPOSITORY                                                      TAG                 IMAGE ID            CREATED             SIZE
ouruser/ubuntu                                                  v3                  ce9e5483a168        3 minutes ago       104MB
ouruser/sinatra                                                 v2                  c4153fb9d0e7        39 minutes ago      104MB
<none>                                                          <none>              b7ade378ae94        39 minutes ago      104MB
centos                                                          latest              300e315adb2f        4 months ago        209MB
quay.io/coreos/flannel                                          v0.13.1-rc1         f03a23d55e57        4 months ago        64.6MB
registry.aliyuncs.com/google_containers/kube-proxy              v1.16.3             9b65a0f78b09        17 months ago       86.1MB
registry.aliyuncs.com/google_containers/kube-apiserver          v1.16.3             df60c7526a3d        17 months ago       217MB
registry.aliyuncs.com/google_containers/kube-controller-manager v1.16.3             bb16442bcd94        17 months ago       163MB
registry.aliyuncs.com/google_containers/kube-scheduler          v1.16.3             98fecf43a54f        17 months ago       87.3MB
registry.aliyuncs.com/google_containers/etcd                    3.3.15-0            b2756210eeab        19 months ago       247MB
registry.aliyuncs.com/google_containers/coredns                 1.6.2               bf261d157914        20 months ago       44.1MB
registry.aliyuncs.com/google_containers/pause                   3.1                 da86e6ba6ca1        3 years ago         742kB
ubuntu                                                          12.04               5b117edd0b76        4 years ago         104MB
ubuntu                                                          f1                  5b117edd0b76        4 years ago         104MB
[root@k8s-master1 ~]#
```

使用刚刚创建的镜像运行一个容器：

```
[root@k8s-master1 ~]# docker run -t -i ouruser/ubuntu:v3
root@0f66d807565d:/# ls
abc  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  srv  sys  tmp  usr  var
root@0f66d807565d:/#
```

3.2.4 导出与加载镜像

使用 docker save 命令导出镜像到本地文件系统：

```
docker save -o ubuntu.tar    ouruser/ubuntu:v3
```

```
[root@k8s-master1 ~]# docker save -o ubuntu.tar  ouruser/ubuntu:v3
[root@k8s-master1 ~]# ll
总用量 106980
-rw-------. 1 root root      1218 11月 16 09:43 anaconda-ks.cfg
-rw-r--r--  1 root root        77 4月  12 21:02 Dockerfile
-rw-------  1 root root 109536768 4月  12 21:13 ubuntu.tar
[root@k8s-master1 ~]#
```

-o:指定导出的文件名

可以使用 `docker load` 从导出的本地文件中再导入到本地镜像库：

```
docker load -i ubuntu.tar
```

```
[root@k8s-master1 ~]# docker rmi ouruser/ubuntu:v3
Error response from daemon: conflict: unable to remove repository reference "ouruser/ubuntu:v3" (must force) - container 0f66d807565d is using its referenced image ce9e5483a168
[root@k8s-master1 ~]# docker rmi -f  ouruser/ubuntu:v3
Untagged: ouruser/ubuntu:v3
Deleted: sha256:ce9e5483a168b4c7da439a368fc50b5047e32208de366d577b72b42c954e7345
[root@k8s-master1 ~]# docker rmi ouruser/ubuntu:v3^C
[root@k8s-master1 ~]# docker load -i ubuntu.tar
Loaded image: ouruser/ubuntu:v3
[root@k8s-master1 ~]# docker images
REPOSITORY                                                        TAG                 IMAGE ID            CREATED             SIZE
ouruser/ubuntu                                                    v3                  ce9e5483a168        15 minutes ago      104MB
<none>                                                            <none>              e3b41f42e67b        16 minutes ago      104MB
ouruser/sinatra                                                   v2                  c4153fb9d0e7        About an hour ago   104MB
<none>                                                            <none>              b7ade378ae94        About an hour ago   104MB
centos                                                            latest              300e315adb2f        4 months ago        209MB
quay.io/coreos/flannel                                            v0.13.1-rc1         f03a23d55e57        4 months ago        64.6MB
registry.aliyuncs.com/google_containers/kube-proxy                v1.16.3             9b65a0f78b09        17 months ago       86.1MB
registry.aliyuncs.com/google_containers/kube-apiserver            v1.16.3             df60c7526a3d        17 months ago       217MB
registry.aliyuncs.com/google_containers/kube-controller-manager   v1.16.3             bb16442bcd94        17 months ago       163MB
registry.aliyuncs.com/google_containers/kube-scheduler            v1.16.3             98fecf43a54f        17 months ago       87.3MB
registry.aliyuncs.com/google_containers/etcd                      3.3.15-0            b2756210eeab        19 months ago       247MB
registry.aliyuncs.com/google_containers/coredns                   1.6.2               bf261d157914        20 months ago       44.1MB
registry.aliyuncs.com/google_containers/pause                     3.1                 da86e6ba6ca1        3 years ago         742kB
ubuntu                                                            12.04               5b117edd0b76        4 years ago         104MB
ubuntu                                                            f1                  5b117edd0b76        4 years ago         104MB
[root@k8s-master1 ~]#
```

3.2.5 上传镜像

用户可以通过 `docker push` 命令，把自己创建的镜像上传到仓库中来共享。

例如，用户在 Docker Hub 上 完成注册后，可以推送自己的镜像到仓库中。

```
[root@k8s-master1 ~]# docker push ouruser/ubuntu:v3
The push refers to repository [docker.io/ouruser/ubuntu]
41a660d048fd: Preparing
3efd1f7c01f6: Preparing
73b4683e66e8: Preparing
ee60293db08f: Preparing
9dc188d975fd: Preparing
58bcc73dcf40: Waiting
denied: requested access to the resource is denied
[root@k8s-master1 ~]#
```

注意：这里推送失败，是由于没有配置好对应的 Docker 仓库地址。

3.2.6 删除镜像

如果要移除本地的镜像，可以使用 *docker rmi* 命令。注意 *docker rm* 命令是移除容器。

```
[root@k8s-master1 ~]# docker rmi ouruser/ubuntu:v3
Error response from daemon: conflict: unable to remove repository reference "ouruser/ubuntu:v3" (must force) - container 4f07f6a414b2 is using its referenced image ce9e5483a168
[root@k8s-master1 ~]# docker rmi -f ouruser/ubuntu:v3
Untagged: ouruser/ubuntu:v3
[root@k8s-master1 ~]#
```

-f:强制删除镜像。

*注意：在删除镜像之前要先用 *docker rm* 删掉依赖于这个镜像的所有容器。

3.3 镜像实现原理

Docker 镜像是怎么实现增量的修改和维护的？

每个镜像都由很多层次构成，Docker 使用 Union FS 将这 些不同的层结合到一个镜像中去。 通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个 disk 挂到同一个目录下,另一个更常用的就是将一个只读的分支和一个可写的分支联合在一起，Live

CD 正是基于此方法可以允许在镜像不 变的基础上允许用户在其上进行一些写操作。Docker 在 AUFS 上构建的容器也是利用了类似的原理。

四、容器（Container）

4.1 Docker 容器基本概念

Docker 利用容器来运行应用。容器是从镜像创建的运行实例，它是独立运行的一个或一组应用，以及它们的运行态环境。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的 Linux 环境（包括 root 用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

*注：镜像是只读的，容器在启动的时候创建一层可写层作为最上层。

4.2 容器使用

4.2.1 启动容器

4.2.1.1 一种是基于镜像新建一个容器并启动

```
docker run ouruser/ubuntu:v3 /bin/echo 'Hello Docker'
```

```
[root@k8s-master1 ~]# docker run ouruser/ubuntu:v3 /bin/echo 'Hello Docker'
Hello Docker
[root@k8s-master1 ~]#
```

这样的方式启动的容器，会在命令执行完毕后就结束运行。这样就像是在本地执行一样。

一般在实际应用中，我们会以为 Docker 分配一个伪终端并且绑定到容器的标准输入上，交互式的方式来使用：

```
docker run -i -t --name="d_01" ouruser/ubuntu:v3 /bin/bash
```

```
[root@k8s-master1 ~]# docker run -i -t ouruser/ubuntu:v3 /bin/bash
root@1438bbda1744:/# ls
abc  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  selinux  srv  sys  tmp  usr  var
root@1438bbda1744:/#
root@1438bbda1744:/#
root@1438bbda1744:/#
```

在交互模式下，用户可以通过所创建的终端来输入命令。

其中，

-t :选项让 *Docker* 分配一个伪终端（*pseudo-tty*）并绑定到容器的标准输入上，

-i :则让容器的标 准输入保持打开，*--name* 则为容器创建一个名称。

当利用 *docker run* 来创建容器时，*Docker* 在后台运行的标准操作包括：

* 利用镜像创建并启动一个容器
* 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
* 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
* 从地址池配置一个 ip 地址给容器
* 执行用户指定的应用程序
* 执行完毕后容器被终止

4.2.1.2 将在终止状态（stopped）的容器重新启动

利用 `docker start` 命令，直接将一个已经终止的容器启动运行。

Docker 容器是非常轻量级的所以我们可以随时删除和新创建容器。

4.2.1.3 守护态方式启动容器（重要）：

很多时候，我们需要让 Docker 容器在后台以守护态（Daemonized）运行。

通过添加 -d 参数来实现。

```
docker run -d ouruser/ubuntu:v3 /bin/sh -c "while true;do echo hello world;sleep 1;done"
```

```
[root@k8s-master1 ~]# docker run -d ouruser/ubuntu:v3 /bin/sh -c "while true;do echo hello world;sleep 1;done"
b4e108a091d45baf093380f3fe8ae8cd31ef8cb3de3cf4aae7b84b95b6522ed8
```

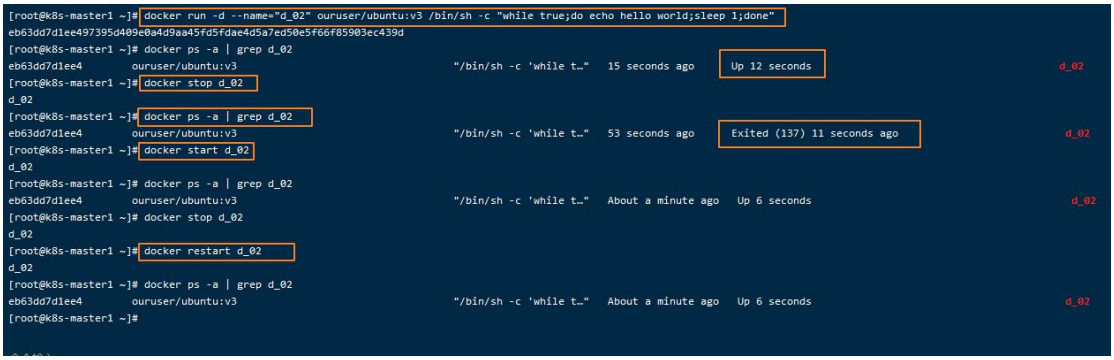
容器启动后回返回一个唯一的 ID，可以通过 docker ps 命令查看容器信息：

```
[root@k8s-master1 ~]# docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS               NAMES
b4e108a091d4        ouruser/ubuntu:v3                    "/bin/sh -c 'while t…"   About a minute ago   Up About a minute                       adoring_cray
```

4.2.2 终止容器

使用 `docker stop` 来终止一个运行中的容器。此外，当 Docker 容器中指定的应用终结时，容器也自动终止。

并且对于那些启动了一个终端的容 器，用户通过 exit 命令或 Ctrl+d 来退出终端时，所创建的容器立刻终止。



以通过 `docker start` 命令来重新启动。

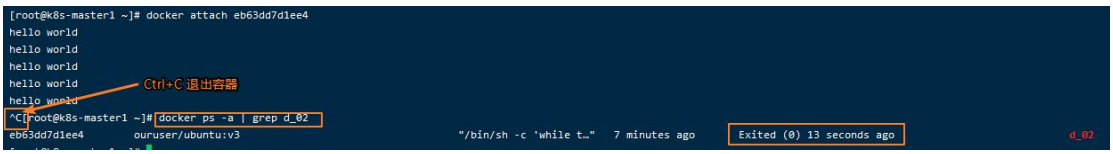
此外， `docker restart` 命令会将一个运行态的容器终止，然后再重新启动它。

4.2.3 进入容器

在使用 -d 参数时，容器启动后会进入后台。 某些时候需要进入容器进行操作。

看下之前我们创建的容器：

docker attach 命令：



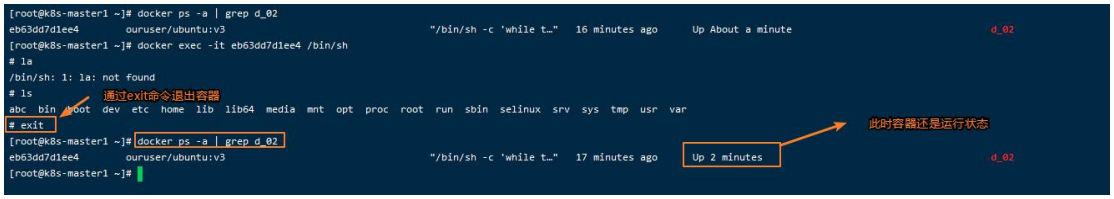
虽然通过 `docker attach` 命令可以进入到容器内容。但是该命令有个问题，使用该命令有一个问题。当多个窗口同时使用该命令进入该容

器时，所有的窗口都会同步显示。如果有一个窗口阻塞了，那么其他窗口也无法再进行操作；并且如果我们通过 Ctrl+C 退出了容器，会导致当前容器被停止掉。

所以 docker attach 命令不太适合于生产环境，平时自己开发应用时可以使用该命令。

docker exec 命令：

docker 在 1.3.X 版本之后提供了一个 docker exec 命令，用于进入容器。



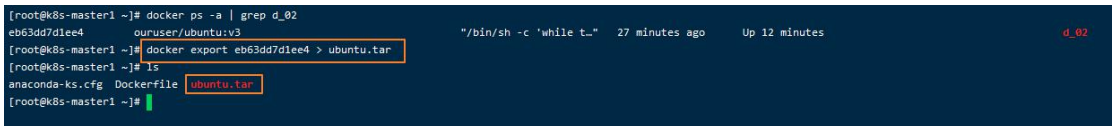
Docker exec 命令，进入容器后，即使执行 *exit* 命令退出容器，容器还是可以正常执行。

nsenter 进入 Docker 容器：

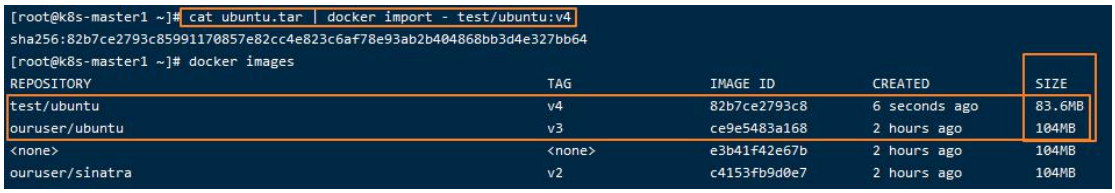
这种方式需要先安装好 nsenter，可以查阅相关文档。

4.2.4 导出和导入容器

如果要导出本地某个容器快照，可以使用 `docker export` 命令：



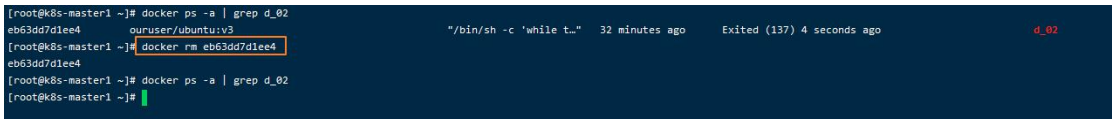
使用 *docker import* 从容器快照文件中再导入为镜像：



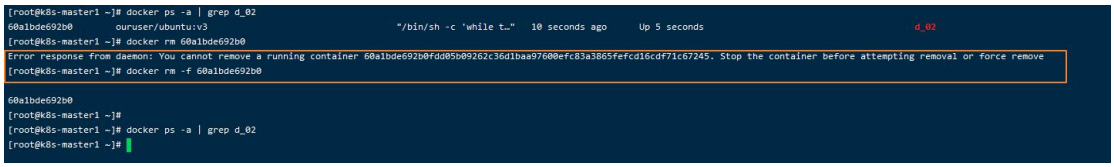
*注：用户既可以使用 *docker load* 来导入镜像存储文件到本地镜像库，也可以使用 *docker import* 来 导入一个容器快照到本地镜像库。这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息 （即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文 件导入时可以重新指定标签等元数据信息。

4.2.5 删除容器

可以使用 *docker rm* 来删除一个处于终止状态的容器：



如果要删除一个运行中的容器，可以添加 *-f* 参数。*Docker* 会发送 *SIGKILL* 信号给容器：



4.3 容器的重启策略

Docker 容器的重启策略是面向生产环境的一个启动策略，在开发过程中可以忽略该策略。

Docker 容器的重启都是由 Docker 守护进程完成的，因此与守护进程息息相关。

4.3.1 Docker 容器的重启策略

no，默认策略，在容器退出时不重启容器

on-failure，在容器非正常退出时（退出状态非 0），才会重启容器

on-failure:3，在容器非正常退出时重启容器，最多重启 3 次

always，在容器退出时总是重启容器

unless-stopped，在容器退出时总是重启容器，但是不考虑在 Docker 守护进程启动时就已经停止了的容器

4.3.2 Docker 容器的退出状态码

docker run 的退出状态码如下：

0，表示正常退出
非 0，表示异常退出（退出状态码采用 chroot 标准）
125，Docker 守护进程本身的错误
126，容器启动后，要执行的默认命令无法调用
127，容器启动后，要执行的默认命令不存在
其他命令状态码，容器启动后正常执行命令，退出命令时该命令的返回状态码作为容器的退出状态码

4.3.3 docker run 的--restart 选项

通过--restart 选项，可以设置容器的重启策略，以决定在容器退出时 Docker 守护进程是否重启刚刚退出的容器。

--restart 选项通常只用于 detached 模式的容器。
--restart 选项不能与--rm 选项同时使用。

显然，--restart 选项适用于 detached 模式的容器，而--rm 选项适

用于 foreground 模式的容器。

在 docker ps 查看容器时，对于使用了--restart 选项的容器，其可能的状态只有 Up 或 Restarting 两种状态。

示例：

```
docker run -d --restart=always centos
docker run -d --restart=on-failure:10 centos
```

五、仓库（Repository）

5.1 docker 仓库基本概念

仓库是集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 Docker Hub，存放了数量庞大的镜像供用户下载。 国内的公开仓库包括 Docker Pool 等，可以提供大陆用户更稳定快速的访问。

当然，用户也可以在本地网络内创建一个私有仓库。

当用户创建了自己的镜像之后就可以使用 push 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时

候，只需要从仓库上 pull 下来就可以了。

*注：Docker 仓库的概念跟 Git 类似，注册服务器可以理解为 GitHub 这样的托管服务。

5.2 Docker 仓库使用

5.2.1 从仓库搜索镜像

可以通过 docker search 命令查找官方仓库中的镜像，并通过 docker pull 命令将镜像下载到本地。

```
[root@k8s-master1 ~]# docker search centos
NAME                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
centos                            The official build of CentOS.                   6517      [OK]
ansible/centos7-ansible           Ansible on Centos7                              133                  [OK]
consol/centos-xfce-vnc            Centos container with "headless" VNC session…   128                  [OK]
jdeathe/centos-ssh                OpenSSH / Supervisor / EPEL/IUS/SCL Repos - …   117                  [OK]
centos/systemd                    systemd enabled base container.                 97                   [OK]
imagine10255/centos6-lnmp-php56   centos6-lnmp-php56                              58                   [OK]
tutum/centos                      Simple CentOS docker image with SSH access      47
kinogmt/centos-ssh                CentOS with SSH                                 29                   [OK]
pivotaldata/centos-gpdb-dev       CentOS image for GPDB development. Tag names…   13
guyton/centos6                    From official centos6 container with full up…   10                   [OK]
centos/tools                      Docker image that has systems administration…   7                    [OK]
drecom/centos-ruby                centos ruby                                     6                    [OK]
```

返回的信息包括：镜像名字、描述、星级（表示该镜像的受欢迎程度）、 是否官方创建、是否自动创建。

另外，在查找的时候通过 *-s N* 参数可以指定仅显示评价为 *N* 星以上的镜像。

5.2.2 从仓库下载镜像

使用 docker pull 命令从仓库下载镜像到本地

```
[root@localhost ~]# docker pull amd64/centos:latest
latest: Pulling from amd64/centos
Digest: sha256:dbbacecc49b088458781c16f3775f2a2ec7521079034a7ba499c8b0bb7f86875
Status: Downloaded newer image for amd64/centos:latest
docker.io/amd64/centos:latest
[root@localhost ~]#
```

通过 docker images 命令查看一下刚刚的镜像是否成功下载到本地了：

```
[root@localhost ~]# docker images | grep amd64
amd64/centos                                latest              300e315adb2f        4 months ago        209MB
```

5.2.3 向仓库推送镜像

当我们自己创建的镜像打包好了之可以推送到镜像仓库供别人使用。

推送之前先使用 `docker tag` 给要推送的镜像打上标记

格式为：`docker tag IMAGE[:TAG][REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]`

```
[root@localhost ~]# docker images | grep amd64
amd64/centos                                latest              300e315adb2f        4 months ago        209MB
[root@localhost ~]# docker tag amd64/centos 127.0.0.1:5000/centos:v1
[root@localhost ~]# docker tag amd64/centos 127.0.0.1:5000/push_centos:v1
[root@localhost ~]# docker images | grep push
127.0.0.1:5000/push_centos                  v1                  300e315adb2f        4 months ago        209MB
[root@localhost ~]#
```

然后使用 docker push 命令推送镜像到仓库；

```
[root@localhost ~]# docker push 127.0.0.1:5000/push_centos:v1
The push refers to repository [127.0.0.1:5000/push_centos]
2653d992f4ef: Pushed
v1: digest: sha256:dbbacecc49b088458781c16f3775f2a2ec7521079034a7ba499c8b0bb7f86875 size: 529
```

然后通过 `curl 127.0.0.1:5000/v2/_catalog`
查看一下是否上传成功：

```
[root@localhost ~]# curl 127.0.0.1:5000/v2/_catalog
{"repositories":["push_centos"]}
```

5.3 私有仓库

有时候使用 Docker Hub 这样的公共仓库可能不方便，用户可以创建一个本地仓库供私人使用。

常用的 Docker 私有仓库：

* Docker registry

* harbor

Docker-registry 私有仓库安装：

第一步：使用 docker 安装一个 registry 私有仓库

```
docker run -d -p 5000:5000 --restart=always --name registry \
-v /opt/data/registry:/var/lib/registry \
 registry
```

第二步：配置私有仓库

编辑/etc/docker/daemon.json 文件，添加下面的内容（没有该文件就新建一个）

```
{
 "insecure-registries" : [
   "192.168.17.137:5000"
 ]
}
```

第三步：重启下 docker

使用 service docker restart 重新一下 docker 服务。

Docker registry 安装参考（官网）

六、Docker 数据存储（volume）

在 *Docker* 容器中管理数据主要有两种方式：

* 数据卷（Data volumes）
* 数据卷容器（Data volume containers）

6.1 数据卷

数据卷是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

* 数据卷可以在容器之间共享和重用
* 对数据卷的修改会立马生效
* 对数据卷的更新，不会影响镜像
* 卷会一直存在，直到没有容器使用

数据卷的使用，类似于 Linux 下对目录或文件进行 mount。

6.1.1 创建一个数据卷

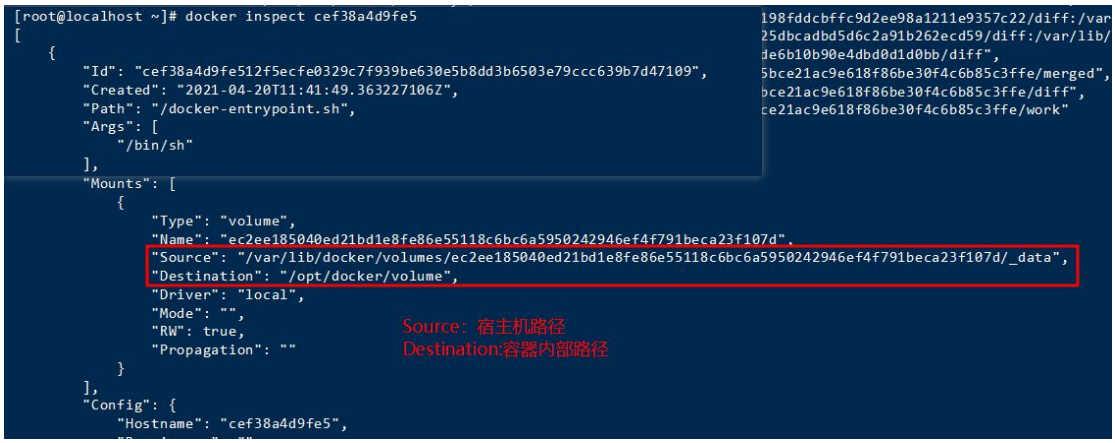
在用 docker run 命令的时候，使用 -v 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用 可以挂载多个数据卷。

```
docker run -d  -v /opt/docker/volume/ nginx:latest /bin/sh
docker exec -it cef38a4d9fe51 /bin/sh
docker inspect cef38a4d9fe51
```

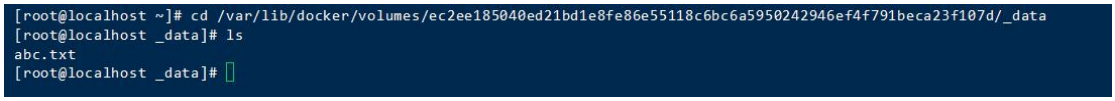
运行一个容器，在容器内部创建一个文件 abc.txt

```
[root@localhost _data]# docker run -it -d  -v /opt/docker/volume/ nginx:latest /bin/sh
cef38a4d9fe512f5ecfe0329c7f939be630e5b8dd3b6503e79ccc639b7d47109
[root@localhost _data]# docker exec -it cef38a4d9fe512f5ecfe0329c7f939be630e5b8dd3b6503e79ccc639b7d47109 /bin/sh
# cd /opt/docker/volume
# ls
# ls
# vi abc.txt
/bin/sh: 4: vi: not found
# toch abc.txt
/bin/sh: 5: toch: not found
# ls
# touch abc.txt
# ls
abc.txt
# ^[[2C
```

下面通过 docker inspect 命令查看一下容器挂载的数据卷：



查看宿主机对应目录：



*注意：也可以在 *Dockerfile* 中使用 *VOLUME* 来添加一个或者多个新的卷到由该镜像创建的任意容器。

挂载一个主机目录作为数据卷

使用 *-v* 标记也可以指定挂载一个本地主机的目录到容器中去。

```
docker run -it    -v /root/docker/volume:/opt/docker/volume2 nginx:latest /bin/sh
```

上面的命令加载主机的 /root/docker/volume 目录到容器的 /opt/docker/volume2 目录。本地目录的路径必须是绝对路径，如果目录不存在 *Docker* 会自动为你创建它。

Docker 挂载数据卷的默认权限是读写,用户也可以通过 :ro 指定为只读。

```
docker run -it    -v /root/docker/volume:/opt/docker/volume2:ro    nginx:latest /bin/sh
```

挂载一个本地主机文件作为数据卷

```
sudo docker run --rm -it -v ~/.bash_history:/.bash_history nginx:latest /bin/sh
```

*注意：如果直接挂载一个文件，很多文件编辑工具，包括 vi 或者 sed --in-place ，可能会造成文件 inode 的改变，从 Docker 1.1 .0 起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

6.2 数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。 数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

首先，创建一个命名的数据卷容器 data_volume：

```
docker run -d -v /opt/db --name data_volume centos:latest echo “data volume”
```

然后，在其他容器中使用 --volumes-from 来挂载 data_volume 容器中的数据卷。

```
docker run -it --name v1 --volumes-from data_volume  centos:latest /bin/sh

docker run -it --name v2 --volumes-from data_volume  centos:latest /bin/sh
```

还可以使用多个 --volumes-from 参数来从多个容器挂载多个数据卷。 也可以从其他已经挂载了数据卷的容器来挂载数据卷。

```
docker run -it --name v3 --volumes-from v2  centos:latest /bin/sh
```

*注意：使用 --volumes-from 参数所挂载数据卷的容器自己并**不需要**保持在运行状态。

如果删除了挂载的容器（包括 data_volume、v1 和 v2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。

七、Docker 网络

7.1 端口映射

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 -P 或 -p 参数来指定端口映射。

-P（大写）: Docker 会随机映射一个端口到内部容器开放的网络端口。

```
[root@k8s-master1 ~]# docker run -d --name web01 -P nginx:latest
a3e482867c1c6ae68df95b88abb74ce9ae0e55ac6fb9a268c3f64146c00b1f23
[root@k8s-master1 ~]# docker ps
CONTAINER ID        IMAGE                                   COMMAND                  CREATED             STATUS              PORTS                     NAMES
a3e482867c1c        nginx:latest                            "/docker-entrypoint.…"   5 seconds ago       Up 4 seconds        0.0.0.0:32768->80/tcp     web01
```

通过 docker ps 可以看到，本地主机的 32768 端口被映射到容器的 80 端口上去了。
此时访问本地的 32768 端口就可以访问到容器的 80 端口上对应的服务。

```
[root@k8s-master1 ~]# curl http://127.0.0.1:32768
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
[root@k8s-master1 ~]#
```

-p（小写）：可以指定要映射的端口，并且在一个指定端口上只可以绑定一个容器；

支持的各式有：

ip:hostPort:containerPort ：映射指定地址的指定端口

ip::containerPort：映射指定地址的任意端口

hostPort:containerPort：映射所有接口地址的指定端口

下面将本地的 5000 端口映射到容器的 80 端口：

```
[root@k8s-master1 ~]# docker run -d --name web02 -p 5000:80 nginx:latest
1807f9459e1f44f436db1aca8a97f601cc05ab428c5c591f925bd25a43c54068
[root@k8s-master1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
1807f9459e1f        nginx:latest        "/docker-entrypoint.…"   3 seconds ago       Up 3 seconds        0.0.0.0:5000->80/tcp     web02
```

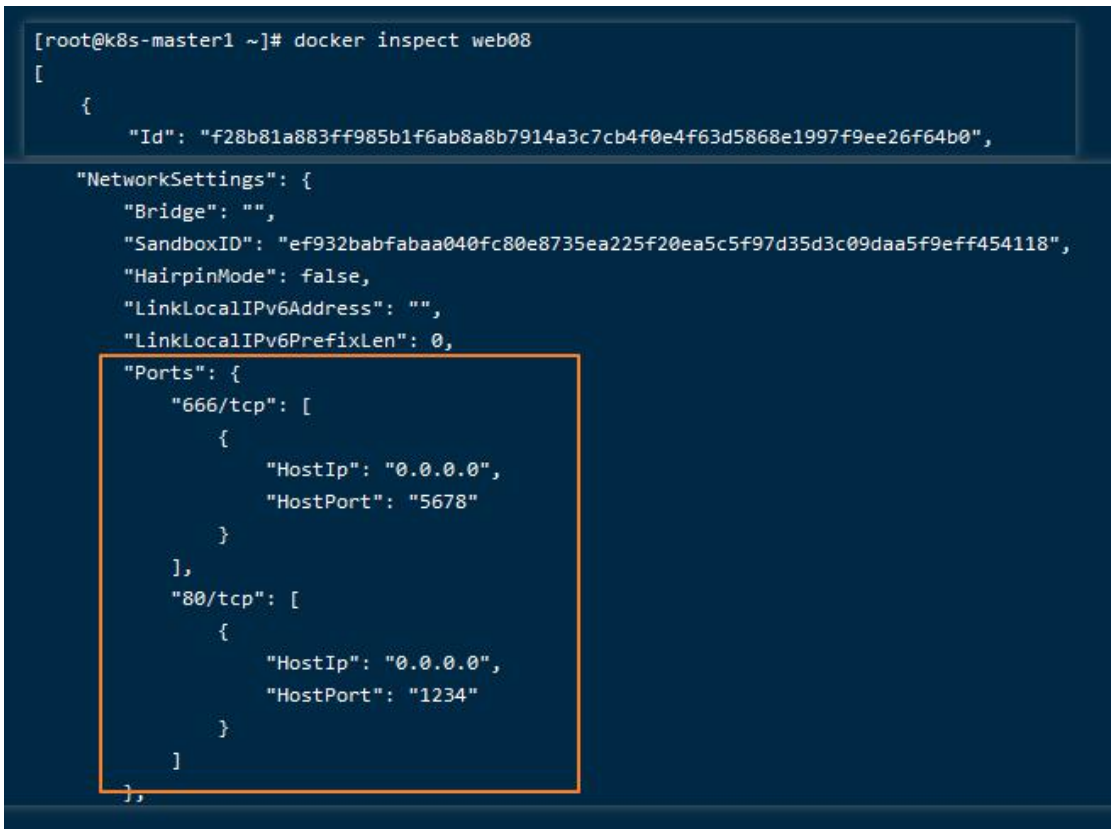
下面访问本地的 5000 端口，也是可以访问到容器 80 端口的：

```
[root@k8s-master1 ~]# curl http://localhost:5000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
[root@k8s-master1 ~]#
```

-p（小写）也可以多次使用来绑定多个端口：

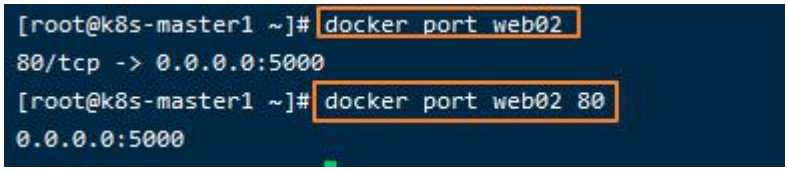
```
[root@k8s-master1 ~]# docker run -d --name web08 -p 1234:80 -p 5678:666 nginx:latest
f28b81a883ff985b1f6ab8a8b7914a3c7cb4f0e4f63d5868e1997f9ee26f64b0
[root@k8s-master1 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                                    NAMES
f28b81a883ff   nginx:latest   "/docker-entrypoint.…"   6 seconds ago    Up 5 seconds    0.0.0.0:1234->80/tcp, 0.0.0.0:5678->666/tcp   web08
```

通过 docker inspect 查看一下详细信息：



查看映射端口配置：

通过 docker port 命令查看当前映射的端口配置，也可以查看绑定的地址。



7.2 Docker 网络基础

我们使用 ifconfig 查看 docker 宿主机的网络信息，发现一个名为 docker0 的网络设备。

```
[root@k8s-master1 ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:5bff:fe2c:dce6  prefixlen 64  scopeid 0x20<link>
        ether 02:42:5b:2c:dc:e6  txqueuelen 0  (Ethernet)
        RX packets 14  bytes 2350 (2.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 26  bytes 1921 (1.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

docker 守护进程就是通过这个 docker0 来为 docker 容器提供网络连接的各种服务。

docker0 就是 Linux 的虚拟网桥

根据 OSI 七层模型，网桥是属于数据链路层的一种设备;

它通过 MAC 地址，也就是物理地址来对网络进行划分；并且在不同的网络之间传递数据。

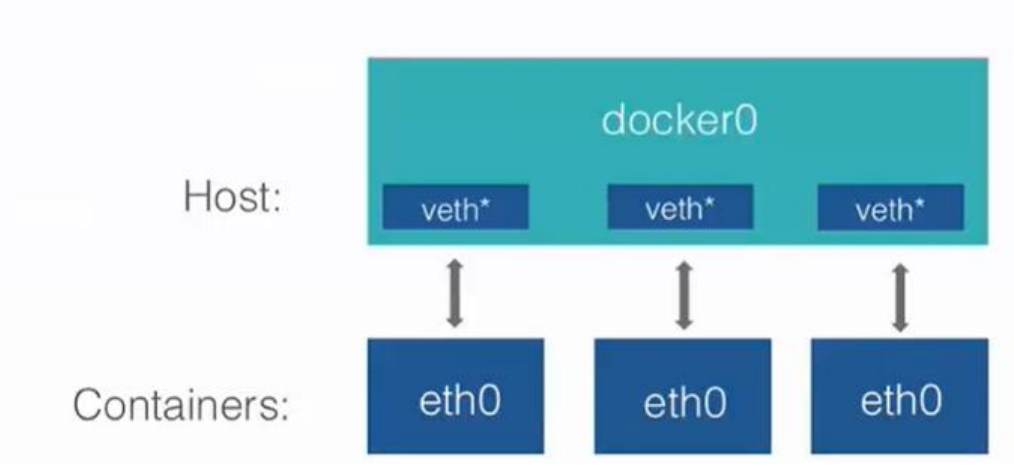
网桥是一个纯的数据链路层的设备；但是 Linux 的虚拟网桥，却有一些不同的特点：

它是可以设置 IP 地址

我们在 docker0 的设置中可以看到，实际它是有一个独立的 IP 地址；但通常而言，IP 地址是三层协议的内容；不应该出现在二层协议中；但是 Linux 的虚拟网桥是通用网络设备抽象的一种【只要是网络设备都是可以识别 IP 地址的】；当虚拟网桥拥有 IP 之后，Linux 便可以通过路由表或者 IP 表规则在网络层定位网桥

相对于用有一个隐藏的虚拟网卡
这就相当于拥有了一个隐藏的虚拟网卡,而这个网卡的名字,就是虚拟网桥的名字。也就是我们刚刚看到的 docker0.

docker 守护进程在一个容器启动时，实际上它要创建网络连接的两端：



- 一端是在容器中的网络设备【eth0】
- 另一端是在运行 docker 容器的宿主机上，打开一个名为【vethXXX】 的一个接口，来实现 docker0 这个网桥与容器的网络通信。

通过 ip addr 查看宿主机的 ip 信息：

```
[root@k8s-master1 ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:f8:c2:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.111/24 brd 192.168.0.255 scope global noprefixroute ens33
       valid_lft forever preferred_lft forever
    inet 192.168.0.158/32 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fef8:c2c3/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ea:aa:08:94 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:eaff:feaa:894/64 scope link
       valid_lft forever preferred_lft forever
```

发现宿主机中有一个 docker0 网桥；

下面创建一个新的容器：

```
docker run -d --name centos02  centos:latest /bin/sh -c "while true;do echo hello world;sleep 1;done"
```

```
[root@k8s-master1 ~]# docker run -d --name centos02  centos:latest /bin/sh -c "while true;do echo hello world;sleep 1;done"
4142de921d62dc97e63ec69e948e7812a52313a4aab5afdce66aba07eec060a6
[root@k8s-master1 ~]# docker ps
CONTAINER ID        IMAGE                                     COMMAND                  CREATED             STATUS              PORTS               NAMES
4142de921d62        centos:latest                             "/bin/sh -c 'while t…"   5 seconds ago       Up 4 seconds                            centos02
```

再通过 ip addr 查看一下

```
[root@k8s-master1 ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:f8:c2:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.111/24 brd 192.168.0.255 scope global noprefixroute ens33
       valid_lft forever preferred_lft forever
    inet 192.168.0.158/32 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fef8:c2c3/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ea:aa:08:94 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:eaff:feaa:894/64 scope link
       valid_lft forever preferred_lft forever
25: vethe686429@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 32:57:bc:61:90:40 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::3057:bcff:fe61:9040/64 scope link
       valid_lft forever preferred_lft forever
```

发现多了一个 25: vethe686429@if24 的网络信息。

这个名为vethe686429@if24 的的网络接口其实就是用来实现 docker0 和容器网络通信的。

进入到刚刚创建的 centos02 **容器再通过 ip add 命令看下：**

```
[root@k8s-master1 ~]# docker exec -it centos02 /bin/sh
sh-4.4# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
sh-4.4#
```

容器内部也有一个名为 24: eth0@if25 的的网络接口，它是 docker 为容器创建的网络通信的另一端，拥有同宿主机中的那个名为 vethe686429@if24 的网络接口通信的。

而且发现每当我们启动一个新的容器，在宿主机上就会多一个名为 vethxxx 的的网络接口卡，它于容器内部的 eth0xxx 的的网络接口卡是成对的；

7.3 Docker 网络模型

Docker 网络模型（官网）

docker 支持以下几种网络模型：

- `bridge` : The default network driver. If you don't specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See bridge networks.
- `host` : For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See use the host network.
- `overlay` : Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks.
- `macvlan` : Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.
- `none` : For this container, disable all networking. Usually used in conjunction with a custom network driver. `none` is not available for swarm services. See disable container networking.
- Network plugins: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

7.3.1 桥接式网络模式（Bridge）

bridge 模式是 docker 的默认网络模式，不写--net 参数，就是 bridge 模式。

使用 docker run -p 时，docker 实际是在 iptables 做了 DNAT 规则，实现端口转发功能。

可以使用 iptables -t nat -vnL 查看：

```
[root@localhost containers]# iptables -t nat -vnL
Chain PREROUTING (policy ACCEPT 33 packets, 3631 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2   104 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0           ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 33 packets, 3631 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 41 packets, 4504 bytes)
 pkts bytes target     prot opt in     out     source               destination
    5   300 DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8         ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 48 packets, 4908 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  *      !docker0  172.17.0.0/16       0.0.0.0/0
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.2           172.17.0.2           tcp dpt:50000
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.2           172.17.0.2           tcp dpt:8080
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.3           172.17.0.3           tcp dpt:5000

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
    0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:50000 to:172.17.0.2:50000
    0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8080 to:172.17.0.2:8080
    7   404 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:5000 to:172.17.0.3:5000
[root@localhost containers]#
```

Iptabls 命令

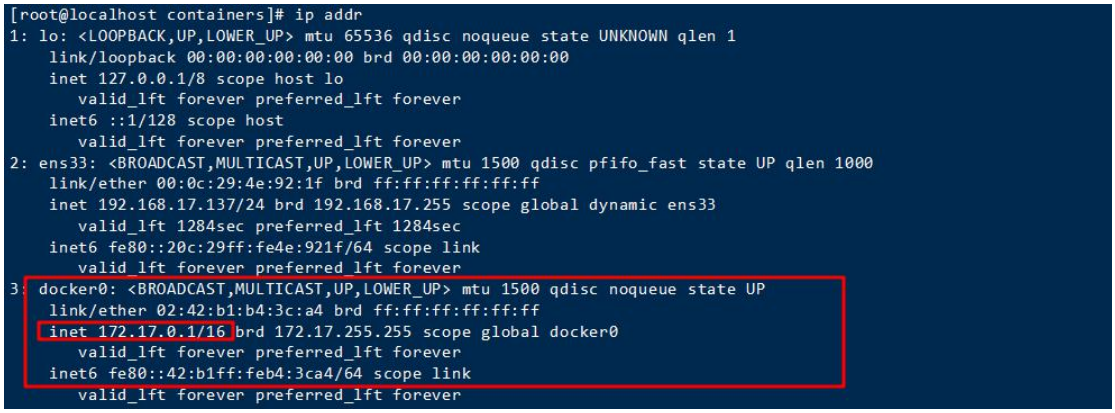
创建一个容器：

```
docker run -d -p 8088:8088 --name web_server nginx:latest
```

发现 iptables 规则新增了一条：

```
[root@localhost containers]# iptables -t nat -vnL
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    3   188 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0           ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    6   384 DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8         ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  *      !docker0  172.17.0.0/16       0.0.0.0/0
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.2           172.17.0.2           tcp dpt:50000
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.2           172.17.0.2           tcp dpt:8080
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.3           172.17.0.3           tcp dpt:5000
    0     0 MASQUERADE  tcp  --  *      *       172.17.0.4           172.17.0.4           tcp dpt:8088

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    1    84 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
    0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:50000 to:172.17.0.2:50000
    0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8080 to:172.17.0.2:8080
    7   404 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:5000 to:172.17.0.3:5000
    0     0 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0           tcp dpt:8088 to:172.17.0.4:8088
[root@localhost containers]#
```

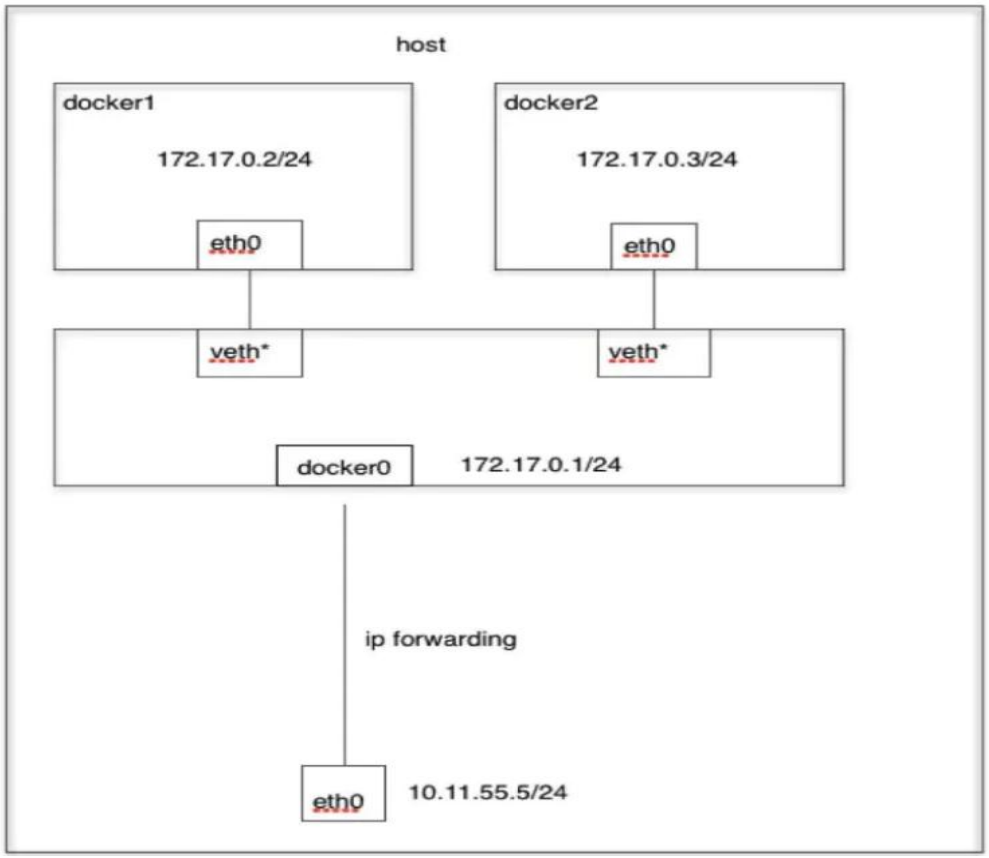
当 Docker 进程启动时，会在主机上创建一个名为 docker0 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上，所以有默认地址 172.17.0.0/16 的地址。



虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。

从 docker0 子网中分配一个 IP 给容器使用，并设置 docker0 的 IP 地址为容器的默认网关。在主机上创建一对虚拟网卡 veth pair 设备，Docker 将 veth pair 设备的一端放在新创建的容器中，并命名为 eth0（容器的网卡），另一端放在主机中，以 vethxxx 这样类似的名字命名，并将这个网络设备加入到 docker0 网桥中。

bridge 模式示意图：



小试牛刀：

```
docker run --name b1 -it --network bridge --rm busybox:latest
```

```
[root@localhost containers]# docker run --name b1 -it --network bridge --rm busybox:latest
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:04
          inet addr:172.17.0.4  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:508 (508.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
/ #
```

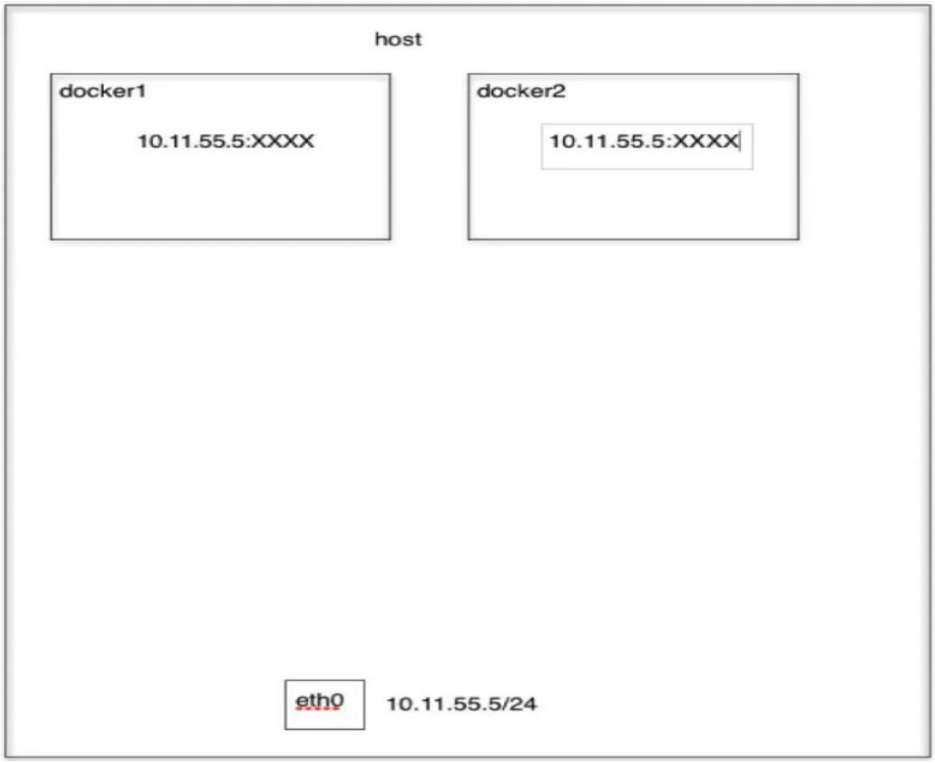
容器中 ping 以下宿主机：

```
/ # ping 192.168.17.137
PING 192.168.17.137 (192.168.17.137): 56 data bytes
64 bytes from 192.168.17.137: seq=0 ttl=64 time=0.244 ms
64 bytes from 192.168.17.137: seq=1 ttl=64 time=0.180 ms
64 bytes from 192.168.17.137: seq=2 ttl=64 time=0.421 ms
64 bytes from 192.168.17.137: seq=3 ttl=64 time=0.352 ms
^C
--- 192.168.17.137 ping statistics ---
```

7.3.2 开放式网络模式（Host）

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。

Host 模式示意图：



小试牛刀：

```
docker run --name b2 -it --network host --rm busybox:latest
```

```
[root@localhost containers]# docker run --name b2 -it --network host --rm busybox:latest
/ # ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:B1:B4:3C:A4
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:b1ff:feb4:3ca4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9478 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11071 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:428465 (418.4 KiB)  TX bytes:98185740 (93.6 MiB)

ens33     Link encap:Ethernet  HWaddr 00:0C:29:4E:92:1F
          inet addr:192.168.17.137  Bcast:192.168.17.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe4e:921f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1223770 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1647120 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:293685613 (280.0 MiB)  TX bytes:226065000 (215.5 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:9425 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9425 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:75707513 (72.2 MiB)  TX bytes:75707513 (72.2 MiB)

vethc764c19 Link encap:Ethernet  HWaddr 2E:C1:F2:73:E3:78
          inet6 addr: fe80::2cc1:f2ff:fe73:e378/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:59 errors:0 dropped:0 overruns:0 frame:0
          TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:5451 (5.3 KiB)  TX bytes:7327 (7.1 KiB)

vethf6dc1a9 Link encap:Ethernet  HWaddr 56:02:7B:CF:94:95
          inet6 addr: fe80::5402:7bff:fecf:9495/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:1002 (1002.0 B)
```

查看一下宿主机的网卡信息：

```
[root@localhost containers]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:b1ff:feb4:3ca4  prefixlen 64  scopeid 0x20<link>
        ether 02:42:b1:b4:3c:a4  txqueuelen 0  (Ethernet)
        RX packets 9478  bytes 428465 (418.4 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11071  bytes 98185740 (93.6 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.17.137  netmask 255.255.255.0  broadcast 192.168.17.255
        inet6 fe80::20c:29ff:fe4e:921f  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:4e:92:1f  txqueuelen 1000  (Ethernet)
        RX packets 1225303  bytes 293805395 (280.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1649391  bytes 226387232 (215.8 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 9425  bytes 75707513 (72.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9425  bytes 75707513 (72.2 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vethc764c19: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::2cc1:f2ff:fe73:e378  prefixlen 64  scopeid 0x20<link>
        ether 2e:c1:f2:73:e3:78  txqueuelen 0  (Ethernet)
        RX packets 59  bytes 5451 (5.3 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 74  bytes 7327 (7.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vethf6dc1a9: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::5402:7bff:fecf:9495  prefixlen 64  scopeid 0x20<link>
        ether 56:02:7b:cf:94:95  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1002 (1002.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

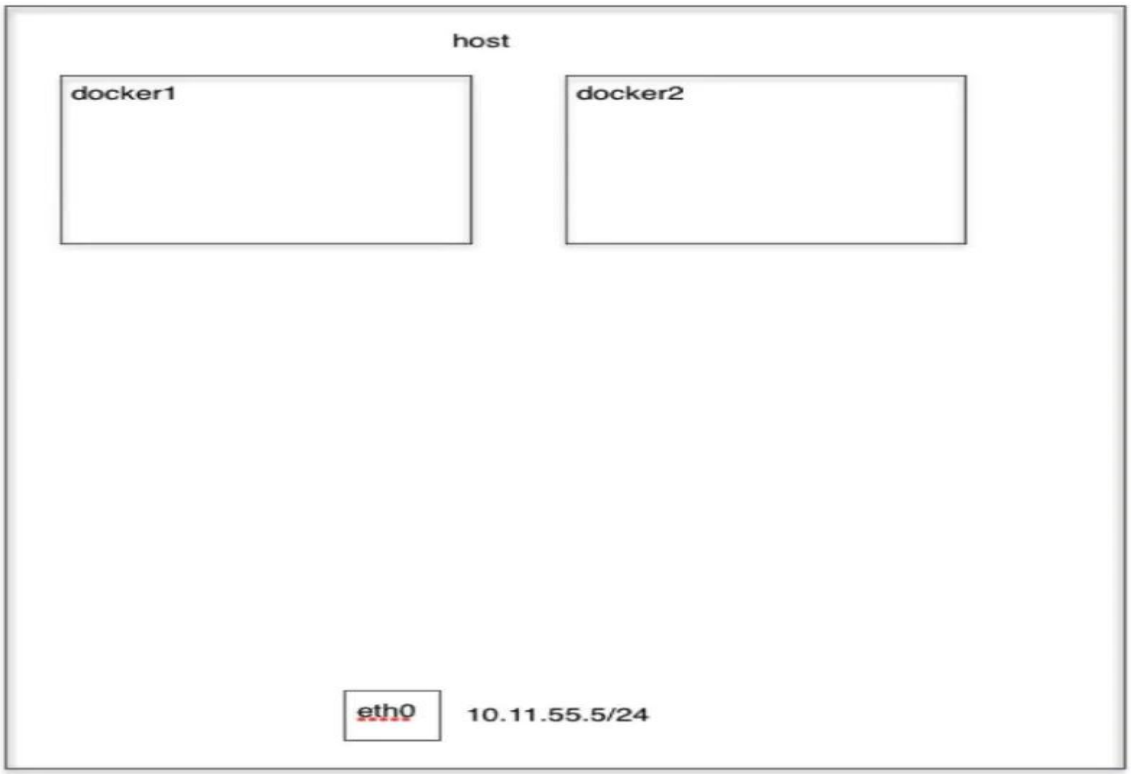
一樣勒。

7.3.3 封闭式网络模式（none）

使用 none 模式，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，只有 lo 网络接口。需要我们自己为 Docker 容器添加网卡、配置 IP 等。

不参与网络通信，运行于此类容器中的进程仅能访问本地回环接口；仅适用于进程无须网络通信的场景中，例如：备份、进程诊断及各种离线任务等。

None 模式示意图：



小试牛刀：

```
docker run --name b3 -it --network none --rm busybox:latest
```

```
[root@localhost containers]# docker run --name b11 -it --network none --rm busybox:latest
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ #
```

八、Docker 底层技术概览

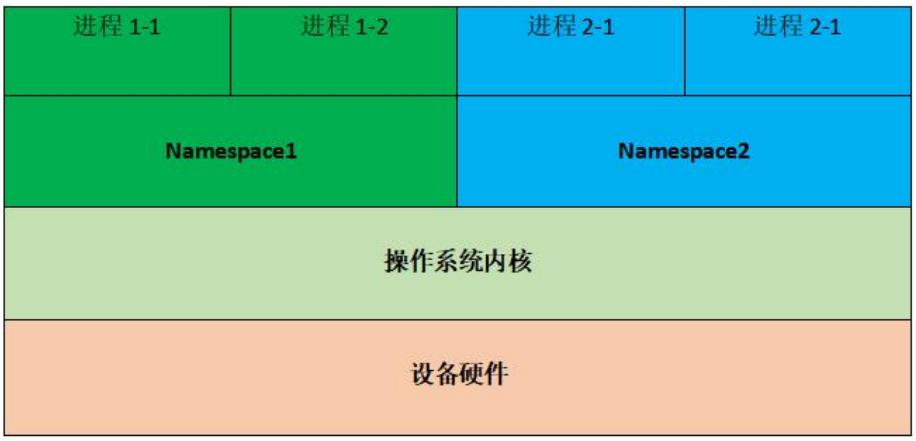
8.1 Linux Namespace

简介

Linux Namespace 是 Linux 提供的一种内核级别环境隔离的方法。

不知道你是否还记得很早以前的 Unix 有一个叫 chroot 的系统调用（通过修改根目录把用户加到一个特定目录下），chroot 提供了一种简单的隔离模式：chroot 内部的文件系统无法访问外部的内容。

Linux Namespace 在此基础上，提供了对 UTS、IPC、mount、PID、network、User 等的隔离机制。举个例子，我们都知道，Linux 下的超级父亲进程的 PID 是 1，所以，同 chroot 一样，如果我们可以把用户的进程空间 jail 到某个进程分支下，并像 chroot 那样让其下面的进程 看到的那个超级父进程的 PID 为 1，于是就可以达到资源隔离的效果了（不同的 PID namespace 中的进程无法看到彼此）



Linux 提供如下 Namespace:

Namespace	Constant	Isolates
Cgroup	CLONE_NEWCGROUP	Cgroup root directory
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name

以上 Namespace 分别对进程的 Cgroup root、进程间通信、网络、文件系统挂载点、进程 ID、用户和组、主机名域名等进行隔离。

6 种命名空间

UTS namespace

UTS namespace 对主机名和域名进行隔离。为什么要隔离主机名？因为主机名可以代替 IP 来访问。如果不隔离，同名访问会出冲突。通过 UTS 命名空间，可以为不同的 Namespaces 设置不同的主机名和网络域 能够简单的将程序隔离到一个独立的网络命名空间

IPC namespace

Linux 提供很多种进程通信机制，IPC namespace 针对 System V 和 POSIX 消息队列，这些 IPC 机制会使用标识符来区别不同的消息队列，然后两个进程通过标识符找到对应的消息队列。

IPC namespace 使得 相同的标识符在两个 namespace 代表不同的消息队列，因此两个 namespace 中的进程不能通过 IPC 来通信。用于隔离进程之间的调用，也就是隔离进程之间的通信；主要针对系统信号量，消息队列以及共享内存；但是需要注意的是，IPC 对于需要进行进程之间通信的程序，只能与同一个命名空间进行通信，无法做到不同命名空间进行信息交换通信；

PID namespace

隔离进程号，不同 namespace 的进程可以使用相同的进程号。 当创建一个 PID namespace 时，第一个进程的 PID 是 1，即 init 进程。它负责回收所有孤儿进程的资源，所有发给 init 进程的信号都会被屏蔽。进程命名空间，用于隔离进程的运行信息
进程是程序运行最直接的体现方式，要实现进程隔离，将进程信息进行隔离是必须的，为了最大的节约转换的损耗，每一个运行在 Namespaces 中的进程，其实就真是的运行在 Linux 系统中，我们虽然可以在宿主机中找到 Namespaces 隔离
的进程信息，但是 namespaces 中的 PID 与宿主机系统中你那个的进程 PID 并不相同，这也就是得益于 PIDNamespaces 实现的进程信息的隔离，PID Namespaces 为命名空间设置了一个独立的进程管理栈，其中就包括了独立的进程号管理，每个运行在 Namespaces 中的进程，会分配到一个属于这个命名空间的 PID。

Mount namespace

隔离文件挂载点，每个进程能看到的文件系统都记录在/proc/$$/mounts 里。在一个 namespace 里挂载、卸载的动作不会影响到其他 namespace。

为什么要通过 Mount namespace 隔离挂载目录？
如果说隔离在某个 namespace 中的程序，所挂载的目录进行修改，那么另一个 nemaspace 中运行的程序也能察觉到，这样就在无形之中影响了其他 Namespace 中程序的运行，显然达不达这样的隔离效果；所以要进行程序之间的隔离，首先是要把程序所使用的挂载目录进行隔离，让不同的 Nemaspaces 拥有独立挂载结构，而程序对挂载信息的修改，也不会影响到其他的 namespace 中程序的运行；

Network namespace

隔离网络资源。每个 namespace 都有自己的网络设备、IP、路由表、/proc/net 目录、端口号等。网络隔离可以保证独立使用网络资源，比如开发两个 web 应用可以使用 80 端口。

新创建的 Network namespace 只有 loopback 一个网络设备，需要手动添加网络设备。

User namespace

隔离用户和用户组。它的厉害之处在于，可以让宿主机上的一个普通用户在 namespace 里成为 0 号用户，也就是 root 用户。这样普通用户可以在容器内“随心所欲”，但是影响也仅限在容器内。

用于隔离用户和用户组信息
通过专门的用户隔离机制，防止运行在 Namespaces 中的程序直接操作宿主机系统中的用户，以避免影响其他 Namespaces 中的运行程序；

docker exec 的底层实现就是上面提过的 setns 。

系统调用

创建容器（进程）主要用到三个系统调用：

```
clone() – 实现线程的系统调用，用来创建一个新的进程，并可以通过上述参数达到隔离
unshare() – 使某进程脱离某个 namespace
setns() – 把某进程加入到某个 namespace
```

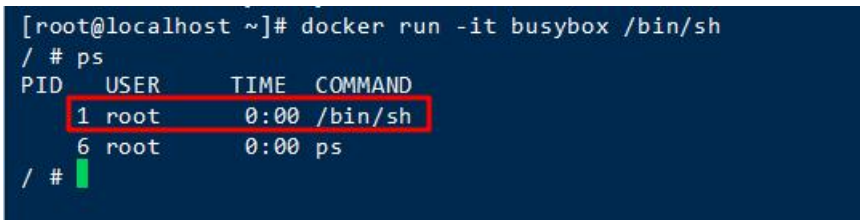
参考：【https://cloud.tencent.com/developer/article/1603569】

案例

下面通过一个栗子来感受下 PID Namespace 的作用：

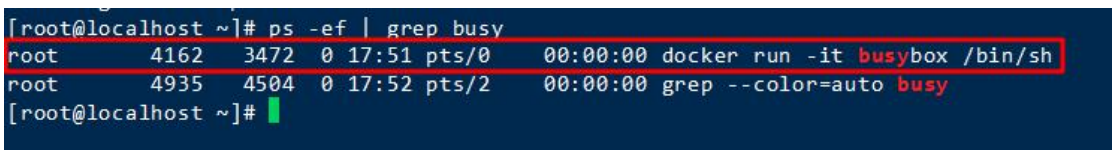
创建一个容器：

```
docker run -it busybox /bin/sh
```



启动一个容器，通过 ps 命令可以参考到，/bin/sh 的 pid=1

查看下宿主机中该/bin/sh 进程的 id:



可以看到，我们在 Docker 里最开始执行的/bin/sh，就是这个容器内部的第 1 号进程（PID=1），而在宿主机上看到它的 PID=4162。

这就意味着，前面执行的/bin/sh，已经被 Docker 隔离在了一个跟宿主机完全不同的世界当中。

而这就是 Docker 在启动一个容器（创建一个进程）时使用了 PID namespace。

```
int pid = clone(main_function, stack_size, CLONE_NEWPID | SIGCHLD, NULL);
```

这时候，Docker 就会在这个 PID=4162 的进程启动时给他施一个“障眼法”，让他永远看不到不属于它这个 namespace 中的进程。这种机制，其实就是对被隔离应用的进程空间做了手脚，使得这些进程只能看到重新计算过的进程编号，比如 PID=1。可实际上，他们在宿主机的操作系统里，还是原来的第 4162 号进程。然后如果你自己只用 PID namespace 使用上述的 clone()创建一个进程，查看 ps 或 top 等命令时，却还是能看到所有进程。

说明并没有完全隔离，这是因为，像 ps、top 这些命令会去读/proc 文件系统，而此时你创建的隔离了 pid 的进程和宿主机使用的是同一个/proc 文件系统，所以这些命令显示的东西都是一样的。

所以，我们还需要使其它的 namespace 隔离，如文件系统进行隔离。

docker 源码：

当启动一个 docker 容器时，会调用到 dockerd 提供的 `/containers/{name:.*}/start` 接口，然后启动一个容器，docker 服务收到请求后，调用关系如下：

[go]

```
//注册http handler
router.NewPostRoute("/containers/{name:.*}/start", r.postContainersStart)
//
func (s *containerRouter) postContainersStart(ctx context.Context, w http.ResponseWriter, r *http.Request, vars map[string]string) error
//
func (daemon *Daemon) ContainerStart(name string, hostConfig *containertypes.HostConfig, checkpoint string, checkpointDir string) error


//
func (daemon *Daemon) containerStart(container *container.Container, checkpoint string, checkpointDir string, resetRestartManager bool) (err error) {
    //...
    spec, err := daemon.createSpec(container)
          //...创建容器
    err = daemon.containerd.Create(context.Background(), container.ID, spec, createOptions)
          //...启动容器
          pid, err := daemon.containerd.Start(context.Background(), container.ID, checkpointDir,
                  container.StreamConfig.Stdin() != nil || container.Config.Tty,
```

```
                container.InitializeStdio)
        //...
        container.SetRunning(pid, true)
        //...}
```

可以看到在 Daemon.containerStart 接口中创建并启动了容器，而创建容器时传入的 spec 参数就包含了 namespace，我们再来看看 daemon.createSpec(container)接口返回的 spec 是什么：

[go]

```
func (daemon *Daemon) createSpec(c *container.Container) (retSpec *specs.Spec, err error) {
    s := oci.DefaultSpec()
    //...
    if err := setUser(&s, c); err != nil {
                return nil, fmt.Errorf("linux spec user: %v", err)
        }
        if err := setNamespaces(daemon, &s, c); err != nil {
                return nil, fmt.Errorf("linux spec namespaces: %v", err)
        }
        //...
        return &s}
//oci.DefaultSpec()会调用DefaultLinuxSpec,可以看到返回的spec中包含了namespace
func DefaultLinuxSpec() specs.Spec {
    s := specs.Spec{
                Version: specs.Version,
                Process: &specs.Process{
                        Capabilities: &specs.LinuxCapabilities{
                                Bounding:    defaultCapabilities(),
                                Permitted:   defaultCapabilities(),
                                Inheritable: defaultCapabilities(),
                                Effective:   defaultCapabilities(),
                        },
```

```
                },
                Root: &specs.Root{},
        }
        s.Mounts = []specs.Mount{
                {
                        Destination: "/proc",
                        Type:        "proc",
                        Source:      "proc",
                        Options:     []string{"nosuid", "noexec", "nodev"},
                },
                {
                        Destination: "/sys/fs/cgroup",
                        Type:        "cgroup",
                        Source:      "cgroup",
                        Options:     []string{"ro", "nosuid", "noexec", "nodev
"},
                },
                //...
        }

        s.Linux = &specs.Linux{
            //...
                Namespaces: []specs.LinuxNamespace{
                        {Type: "mount"},
                        {Type: "network"},
                        {Type: "uts"},
                        {Type: "pid"},
                        {Type: "ipc"},
                },
                //...
        //...
        return s}
```

```
//而在 setNamespaces 中还会根据其它配置对 namespace 进行修改
func setNamespaces(daemon *Daemon, s *specs.Spec, c *container.Container) error {
	userNS := false
	// user
	if c.HostConfig.UsernsMode.IsPrivate() {
		uidMap := daemon.idMapping.UIDs()
		if uidMap != nil {
			userNS = true
			ns := specs.LinuxNamespace{Type: "user"}
			setNamespace(s, ns)
			s.Linux.UIDMappings = specMapping(uidMap)
			s.Linux.GIDMappings = specMapping(daemon.idMapping.GIDs
())
		}
	}
	// network
	if !c.Config.NetworkDisabled {
		ns := specs.LinuxNamespace{Type: "network"}
		parts := strings.SplitN(string(c.HostConfig.NetworkMode), ":", 2)
		if parts[0] == "container" {
			nc, err := daemon.getNetworkedContainer(c.ID, c.HostConf
ig.NetworkMode.ConnectedContainer())
			if err != nil {
				return err
			}
			ns.Path = fmt.Sprintf("/proc/%d/ns/net", nc.State.GetPID
())
			if userNS {
				// to share a net namespace, they must also sha
re a user namespace
				nsUser := specs.LinuxNamespace{Type: "user"}
				nsUser.Path = fmt.Sprintf("/proc/%d/ns/user", n
c.State.GetPID())
```

```
                                        setNamespace(s, nsUser)
                                }
                        } else if c.HostConfig.NetworkMode.IsHost() {
                                ns.Path = c.NetworkSettings.SandboxKey
                        }
                        setNamespace(s, ns)
                }

                // ipc
                ipcMode := c.HostConfig.IpcMode
                switch {
                case ipcMode.IsContainer():
                        ns := specs.LinuxNamespace{Type: "ipc"}
                        ic, err := daemon.getIpcContainer(ipcMode.Container())
                        if err != nil {
                                return err
                        }
                        ns.Path = fmt.Sprintf("/proc/%d/ns/ipc", ic.State.GetPID())
                        setNamespace(s, ns)
                        if userNS {
                                // to share an IPC namespace, they must also share a user
 namespace
                                nsUser := specs.LinuxNamespace{Type: "user"}
                                nsUser.Path = fmt.Sprintf("/proc/%d/ns/user", ic.State.G
etPID())
                                setNamespace(s, nsUser)
                        }
                case ipcMode.IsHost():
                        oci.RemoveNamespace(s, specs.LinuxNamespaceType("ipc"))
                case ipcMode.IsEmpty():
                        // A container was created by an older version of the daemon.
                        // The default behavior used to be what is now called "shareable".
                        fallthrough
```

```
        case ipcMode.IsPrivate(), ipcMode.IsShareable(), ipcMode.IsNone():
                ns := specs.LinuxNamespace{Type: "ipc"}
                setNamespace(s, ns)
        default:
                return fmt.Errorf("Invalid IPC mode: %v", ipcMode)
        }

        // pid
        if c.HostConfig.PidMode.IsContainer() {
                ns := specs.LinuxNamespace{Type: "pid"}
                pc, err := daemon.getPidContainer(c)
                if err != nil {
                        return err
                }
                ns.Path = fmt.Sprintf("/proc/%d/ns/pid", pc.State.GetPID())
                setNamespace(s, ns)
                if userNS {
                        // to share a PID namespace, they must also share a user
namespace
                        nsUser := specs.LinuxNamespace{Type: "user"}
                        nsUser.Path = fmt.Sprintf("/proc/%d/ns/user", pc.State.G
etPID())
                        setNamespace(s, nsUser)
                }
        } else if c.HostConfig.PidMode.IsHost() {
                oci.RemoveNamespace(s, specs.LinuxNamespaceType("pid"))
        } else {
                ns := specs.LinuxNamespace{Type: "pid"}
                setNamespace(s, ns)
        }
        // uts
        if c.HostConfig.UTSMode.IsHost() {
                oci.RemoveNamespace(s, specs.LinuxNamespaceType("uts"))
```

```
                s.Hostname = ""
        }


        return nil}func setNamespace(s *specs.Spec, ns specs.LinuxNamespace) {
        for i, n := range s.Linux.Namespaces {
                if n.Type == ns.Type {
                        s.Linux.Namespaces[i] = ns
                        return
                }
        }
        s.Linux.Namespaces = append(s.Linux.Namespaces, ns)}
```

其实很早以前 Docker 创建一个容器，获取 namespace 是通过 CloneFlags 函数，后来有了开放容器计划(OCI)规范后，就改为了以上面代码中方式创建容器。

OCI 之前代码如下：

[go]

```
var namespaceInfo = map[NamespaceType]int{
        NEWNET:  unix.CLONE_NEWNET,
        NEWNS:   unix.CLONE_NEWNS,
        NEWUSER: unix.CLONE_NEWUSER,
        NEWIPC:  unix.CLONE_NEWIPC,
        NEWUTS:  unix.CLONE_NEWUTS,
        NEWPID:  unix.CLONE_NEWPID,}
// CloneFlags parses the container's Namespaces options to set the correct// flags on
 clone, unshare. This function returns flags only for new namespaces.func (n *Namespa
ces) CloneFlags() uintptr {
        var flag int
        for _, v := range *n {
                if v.Path != "" {
                        continue
                }
                flag |= namespaceInfo[v.Type]
        }
```

```
        return uintptr(flag)}func (c *linuxContainer) newInitProcess(p *Process, cmd *exec.Cmd, parentPipe, childPipe *os.File) (*initProcess, error) {     t := "_LIBCONTAINER_INITTYPE=standard"
    //
    //没错，就是这里～
    //
    cloneFlags := c.config.Namespaces.CloneFlags()
    if cloneFlags&syscall.CLONE_NEWUSER != 0 {
        if err := c.addUidGidMappings(cmd.SysProcAttr); err != nil {
            // user mappings are not supported
            return nil, err
        }
        enableSetgroups(cmd.SysProcAttr)
        // Default to root user when user namespaces are enabled.
        if cmd.SysProcAttr.Credential == nil {
            cmd.SysProcAttr.Credential = &syscall.Credential{}
        }
    }
    cmd.Env = append(cmd.Env, t)
    cmd.SysProcAttr.Cloneflags = cloneFlags
    return &initProcess{
        cmd:        cmd,
        childPipe:  childPipe,
        parentPipe: parentPipe,
        manager:    c.cgroupManager,
        config:     c.newInitConfig(p),
    }, nil}
```

现在，容器运行时，通过 OCI 这个容器运行时规范同底层的 Linux 操作系统进行交互，

即：把容器操作请求翻译成对 Linux 操作系统的调用（操作 Linux Namespace 和 Cgroups 等）。

8.2 Linux Cgroup

参考：【https://houbb.github.io/2019/12/18/docker-learn-32-core-cgroup】

Cgroup 是 Control group 的简称，是 Linux 内核提供的一个特性，用于限制和隔离一组进程对系统资源的使用。

子系统

对不同资源的具体管理是由各个子系统分工完成的。

子系统	作用
devices	设备权限控制
cpuset	分配指定的 CPU 和内存节点
CPU	控制 CPU 使用率
cpuacct	统计 CPU 使用情况
memory	限制内存的使用上限
freezer	暂停 Cgroup 中的进程
net_cls	配合流控限制网络带宽
net_prio	设置进程的网络流量优先级
perf_event	允许 Perf 工具基于 Cgroup 分组做性能检测
huge_tlb	限制 HugeTLB 的使用

在 Cgroup 出现之前，只能对一个进程做资源限制，如 ulimit 限制一个进程的打开文件上限、栈大小。而 Cgroup 可以对进程进行任意分组，如何分组由用户自定义。

cpuset 子系统

cpuset 可以为一组进程分配指定的 CPU 和内存节点。

cpuset 一开始用在高性能计算上，在 NUMA(non-uniform memory access) 架构的服务器上，通过将进程绑定到固定的 CPU 和内存节点上，来避免进程在运行时因跨节点内存访问而导致的性能下降。

cpuset 的主要接口如下：

cpuset.cpus: 允许进程使用的 CPU 列表

cpuset.mems： 允许进程使用的内存节点列表

cpu 子系统

cpu 子系统用于限制进程的 CPU 利用率。

具体支持三个功能

第一，CPU 比重分配。使用 cpu.shares 接口。

第二，CPU 带宽限制。使用 cpu.cfs_period_us 和 cpu.cfs_quota_us 接口。

第三， 实时进程的 CPU 带宽限制。使用 cpu_rt_period_us 和 cpu_rt_quota_us 接口。

cpuacct 子系统

统计各个 Cgroup 的 CPU 使用情况，有如下接口：

cpuacct.stat: 报告这个 Cgroup 在用户态和内核态消耗的 CPU 时间，单位是 赫兹。

cpuacct.usage： 报告该 Cgroup 消耗的总 CPU 时间。

cpuacct.usage_percpu：报告该 Cgroup 在每个 CPU 上的消耗时间。

memory 子系统

限制 Cgroup 所能使用的内存上限。

memory.limit_in_bytes：设定内存上限，单位字节。

默认情况下，如果使用的内存超过上限，Linux 内核会试图回收内存，如果这样仍无法将内存降到限制的范围内，就会触发 OOM，选择杀死该 Cgroup 中的某个进程。 memory.memsw,limit_in_bytes: 设定内存加上交换内存区的总量。

memory.oom_control： 如果设置为 0，那么内存超过上限时，不会杀死进程，而是阻塞等待进程释放内存；同时系统会向用户态发送事件通知。 memory.stat: 报告内存使用信息。

blkio

限制 Cgroup 对 阻塞 IO 的使用。

blkio.weight: 设置权值，范围在[100, 1000]，属于比重分配，不是绝对带宽。因此只有当不同 Cgroup 争用同一个阻塞设备时才起作用 blkio.weight_device： 对具体设备设置权值。它会覆盖上面的选项值。 blkio.throttle.read_bps_device: 对具体的设备，设置每秒读磁盘的带宽上限。
blkio.throttle.write_bps_device: 对具体的设备，设置每秒写磁盘的带宽上限。
blkio.throttle.read_iops_device: 对具体的设备，设置每秒读磁盘的 IOPS 带宽上限。
blkio.throttle.write_iops_device: 对具体的设备，设置每秒写磁盘的 IOPS 带宽上限。

devices 子系统

控制 Cgroup 的进程对哪些设备有访问权限 devices.list: 只读文件，显示目前允许被访问的设备列表，文件格式为 `类型 [a|b|c] 设备号 [major:minor] 权限 [r/w/m 的组合]`

a/b/c 表示 所有设备、块设备和字符设备。

devices.allow： 只写文件，以上述格式描述允许相应设备的访问列表。 devices.deny： 只写文件，以上述格式描述禁止相应设备的访问列表。

8.3 UnionFS

Linux 的命名空间和控制组分别解决了不同资源隔离的问题，前者解决了进程、网络以及文件系统的隔离，后者实现了 CPU、内存等资源的隔离，但是在 Docker 中还有另一个非常重要的问题需要解决 - 也就是镜像。

联合文件系统（UnionFS）是一种**分层、轻量级**并且**高性能**的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)。

联合文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承，基于基础镜像（没有父镜像），可以制作各种具体的应用镜像。

另外，不同 Docker 容器就可以共享一些基础的文件系统层，同时再加上自己独有的改动层，大大提高了存储的效率。

Docker 使用存储驱动程序来管理镜像层和可写容器层的内容，每个存储驱动程序的处理方式不同，但是所有的驱动成都使用可堆叠的镜像层和写时复制（Cow）策略，这些驱动程序

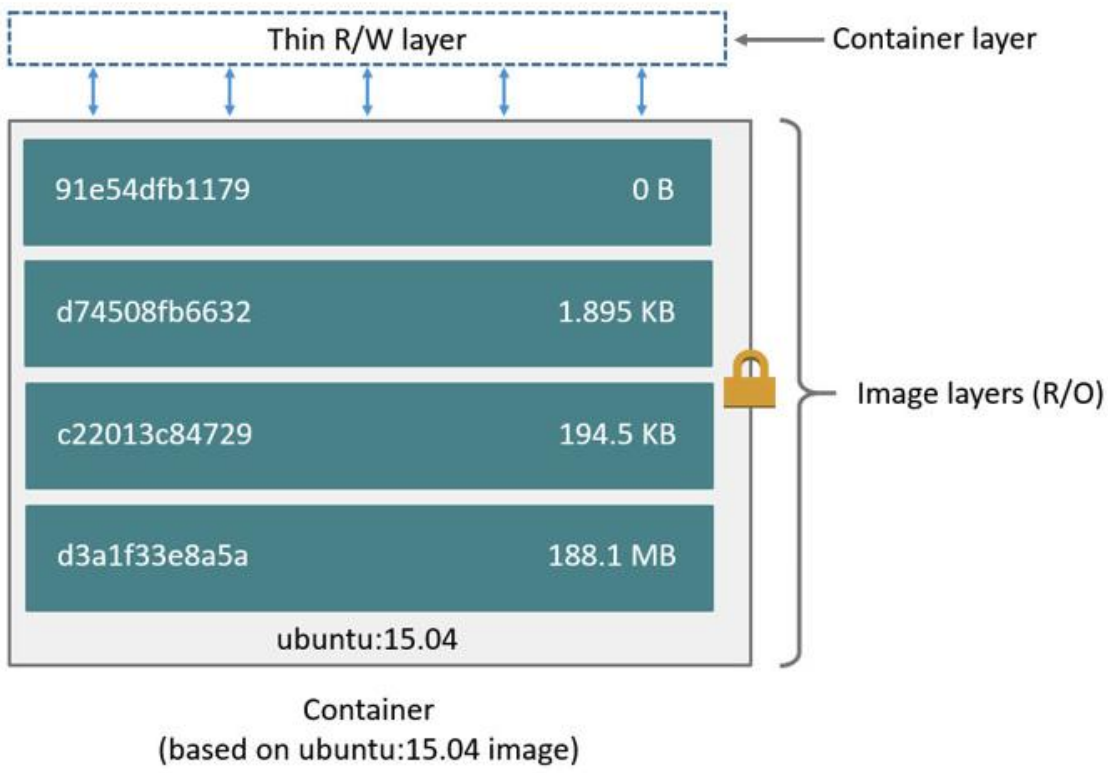
管理的这些层其实就是 UnionFS（联合文件系统），现在 Docker 主要支持的存储驱动有 aufs、devicemapper、overlay、overlay2、zfs 和 vfs 等等，在最新的 Docker 中，overlay2 取代了 aufs 成为了推荐的存储驱动。

如何选择存储驱动：

对于 Docker 如何选择一个合适的存储驱动程序，可以查看官方文档 Docker storage drivers。

Copy-on-write:

写时复制是一种共享和复制文件的策略，可以最大程度地提高效率，如果文件或目录位于镜像的较低层中，而另一层（包括可写层）需要对其进行读取访问，则它直接使用现有文件即可。另一层第一次需要修改文件时（在构建镜像或运行容器时），将文件复制到该层并进行修改。这样可以将 I/O 和每个后续层的大小最小化。



容器和镜像之间的主要区别就是容器在镜像顶部由一个可写层，在容器中的所有操作都会存储在这个容器层中，删除容器后，容器层也会被删除，但是镜像不会变化。正因为每个容器都有自己的可写容器层，所有更改都存储在自己的容器层中，所以多个容器之间可以共享同一基础镜像的访问，但仍然具有自己的数据状态。

附录

一、Dockerfile 语法

FROM
FROM 指定一个基础镜像， 一般情况下一个可用的 Dockerfile 一定是 FROM 为第一个指令。至于 image 则可以是任何合理存在的 image 镜像。
FROM 一定是首个非注释指令 Dockerfile.
FROM 可以在一个 Dockerfile 中出现多次，以便于创建混合的 images。
如果没有指定 tag ，latest 将会被指定为要使用的基础镜像版本。

MAINTAINER
这里是用于指定镜像制作者的信息

RUN
RUN 命令将在当前 image 中执行任意合法命令并提交执行结果。命令执行提交后，就会自动执行 Dockerfile 中的下一个指令。
层级 RUN 指令和生成提交是符合 Docker 核心理念的做法。它允许像版本控制那样，在任意一个点，对 image 镜像进行定制化构建。
RUN 指令缓存不会在下个命令执行时自动失效。比如 RUN apt-get dist-upgrade -y 的缓存就可能被用于下一个指令. --no-cache 标志可以被用于强制取消缓存使用。

ENV
ENV 指令可以用于为 docker 容器设置环境变量
ENV 设置的环境变量，可以使用 docker inspect 命令来查看。同时还可以使用 docker run --env <key>=<value>来修改环境变量。

USER
USER 用来切换运行属主身份的。Docker 默认是使用 root，但若不需要，建议切换使用者身分，毕竟 root 权限太大了，使用上有安全的风险。

WORKDIR

WORKDIR 用来切换工作目录的。Docker 默认的工作目录是/，只有 RUN 能执行 cd 命令切换目录，而且还只作用在当下的 RUN，也就是说**每一个 RUN 都是独立进行的**。如果想让其他指令在指定的目录下执行，就得靠 WORKDIR。WORKDIR 动作的目录改变是持久的，不用每个指令前都使用一次 WORKDIR。

COPY

COPY 将文件从路径 <src> 复制添加到容器内部路径 <dest>。

<src>

必须是想对于源文件夹的一个文件或目录，也可以是一个远程的 url，<dest> 是目标容器中的绝对路径。

所有的新文件和文件夹都会创建 UID 和 GID 。事实上如果 <src> 是一个远程文件 URL，那么目标文件的权限将会是 600。

ADD

ADD 将文件从路径 <src> 复制添加到容器内部路径 <dest>。

<src> 必须是想对于源文件夹的一个文件或目录，也可以是一个远程的 url。

<dest> 是目标容器中的绝对路径。

所有的新文件和文件夹都会创建 UID 和 GID。事实上如果 <src> 是一个远程文件 URL，那么目标文件的权限将会是 600。

VOLUME

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。

EXPOSE

EXPOSE 指令指定在 docker 允许时指定的端口进行转发。

CMD

Dockerfile.中只能有一个 CMD 指令。 如果你指定了多个，那么最后个 CMD 指令是生效的。

CMD 指令的主要作用是提供默认的执行容器。这些默认值可以包括可执行文件，也可以省略可执行文件。

当你使用 shell 或 exec 格式时， CMD

会自动执行这个命令。

ONBUILD
ONBUILD 的作用就是让指令延迟執行，延迟到下一个使用 FROM 的 Dockerfile 在建立 image 时执行，只限延迟一次。
ONBUILD 的使用情景是在建立镜像时取得最新的源码 (搭配 RUN) 与限定系统框架。

ARG
ARG 是 Docker1.9 版本才新加入的指令。
ARG 定义的变量只在建立 image 时有效，建立完成后变量就失效消失

LABEL
定义一个 image 标签 Owner，并赋值，其值为变量 Name 的值。(LABEL Owner=$Name)

ENTRYPOINT
是指定 Docker image 运行成 instance (也就是 Docker container) 时，要执行的命令或者文件。

注意：Dockerfile 中的每一个命令都会创建一个新的 layer，而一个容器能够拥有的最多 layer 数是有限制的[127]。所以尽量将逻辑上连贯的命令合并可以减少 layer 的层数，合并命令的方法可以包括将多个可以合并的命令（EXPOSE，ENV，VOLUME，COPY）合并,可以减少 layer 的层数，这也可以加快编译速度。

二、Docker 常用命令

Docker 命令使用：docker --help，等价与 docker -h

Docker 命令官方文档

三、参考文献

Docker 官方网站：https://docs.docker.com/
Docker 从入门到实践：https://yeasy.gitbook.io/docker_practice/
Docker 百度百科：https://baike.baidu.com/item/Docker/13344470?fr=aladdin