Docker

Docker归根到底是一种容器虚拟化技术(操作系统虚拟化)，其依赖的核心技术主要包括Linux操作系统的命名空间(Namespaces),控制组(Control Groups),联合文件系统(Union File Systems)和Linux虚拟网络支持。

安装

在线安装

centos/redhat

docker对于系统内核有严格要求，因此建议使用尽量新的操作系统

> 建议使用CentOS7以后的版本

1. 可以选择使用rpm包进行安装

```
  # 请自行选择最新docker版本
  $ wget
https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-
ce-17.12.0.ce-1.el7.centos.x86_64.rpm
  $ yum install docker-ce-17.06.2.ce-1.el7.centos.x86_64.rpm
```

2. 也可以设置yum源，并通过

```
yum install
```

进行安装

```
  $ cat /etc/yum.repos.d/docker.repo
  [dockerrepo]

  name=Docker Repository

  baseurl=https://yum.dockerproject.org/repo/main/centos/7/

  enabled=1

  gpgcheck=1

  gpgkey=https://yum.dockerproject.org/gpg

  $ yum install docker-engine -y
```

安装之后，使用`docker version`进行验证和版本确认。

ubuntu

同CentOS相同，ubuntu也尽量使用新的版本。然后通过设置apt源并在线安装

```
$ sudo apt-get update
$ sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
$ sudo apt-get update
```

离线安装

二进制安装

1. 首先去官网下载二进制文件`https://download.docker.com/linux/static/stable/x86_64/`。
2. 然后解压并将文件移动到`/usr/bin`目录下

   ```
   $ tar zxf docker-18.06.1-ce.tgz
   $ mv docker/* /usr/bin/
   $ rm -rf docker*.tgz
   ```

3. 然后编辑docker.service文件

   ```
   $ cat /etc/systemd/system/docker.service
   [Unit]
   Description=Docker Application Container Engine
   Documentation=https://docs.docker.com
   After=network-online.target firewalld.service
   Wants=network-online.target

   [Service]
   Type=notify
   # the default is not to use systemd for cgroups because the delegate issues still
   # exists and systemd currently does not support the cgroup feature set required
   # for containers run by docker
   ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock
   ExecReload=/bin/kill -s HUP $MAINPID
   # Having non-zero Limit*s causes performance problems due to accounting overhead
   ```

```
  # in the kernel. We recommend using cgroups to do container-local
accounting.
  LimitNOFILE=infinity
  LimitNPROC=infinity
  LimitCORE=infinity
  # Uncomment TasksMax if your systemd version supports it.
  # Only systemd 226 and above support this version.
  #TasksMax=infinity
  TimeoutStartSec=0
  # set delegate yes so that systemd does not reset the cgroups of docker
containers
  Delegate=yes
  # kill only the docker process, not all processes in the cgroup
  KillMode=process
  # restart the docker process if it exits prematurely
  Restart=on-failure
  StartLimitBurst=3
  StartLimitInterval=60s

  [Install]
  WantedBy=multi-user.target
```

4. 授予docker可执行权限

```
  $ chmod +x /etc/systemd/system/docker.service
  $ systemctl daemon-reload   //重载systemd下 xxx.service文件
  $ systemctl start docker       //启动Docker
  $ systemctl enable docker.service   //设置开机自启
```

5. 通过`docker version`验证其版本及安装情况

CentOS通过RPM包安装

1. 首先确认安装

```
yum-plugin-downloadonly
```

```
  yum install yum-plugin-downloadonly
```

2. 然后选择某个文件，并将离线文件下载到其中

```
  # 选择/opt文件夹
  yum  install --downloadonly --downloaddir=/opt/docker docker
```

3. 当需要安装时，执行rpm命令

```
  rpm -Uvh /opt/docker/*.rpm --nodeps --force
```

基础架构

Docker采用了标准的C/S架构，包括了客户端和服务端两大部分。

> 客户端和服务端既可以运行在一个机器上，也可以通过socket或RESTful API来进行通信

1. 服务端

Docker daemon一般在宿主主机后台运行，作为服务端接收来自客户的请求。Docker服务端默认监听本地的`unix:///var/run/docker.sock`套接字，只允许本地root用户访问。可以通过`-H`选项来修改监听的方式。

1. 客户端

Docker客户端则为用户提供一系列可执行命令，用户用这些命令实现与Docker Daemon的交互

命名空间

命名空间Namespace是Linux内核针对实现容器虚拟化引入的一个强大特性。

每个容器都可以拥有自己独特的命名空间，运行在其中的应用都像是在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。

在操作系统中，包括内核、文件系统、网络、PID、UID、IPC、内存、硬盘、CPU等资源，所有的资源都是应用进程直接共享的。要实现虚拟化，除了要实现对内存、CPU、网络IO、硬盘ID、存储空间等的限制外。还要实现文件系统、网络、PID、UID、IPC等的隔离。

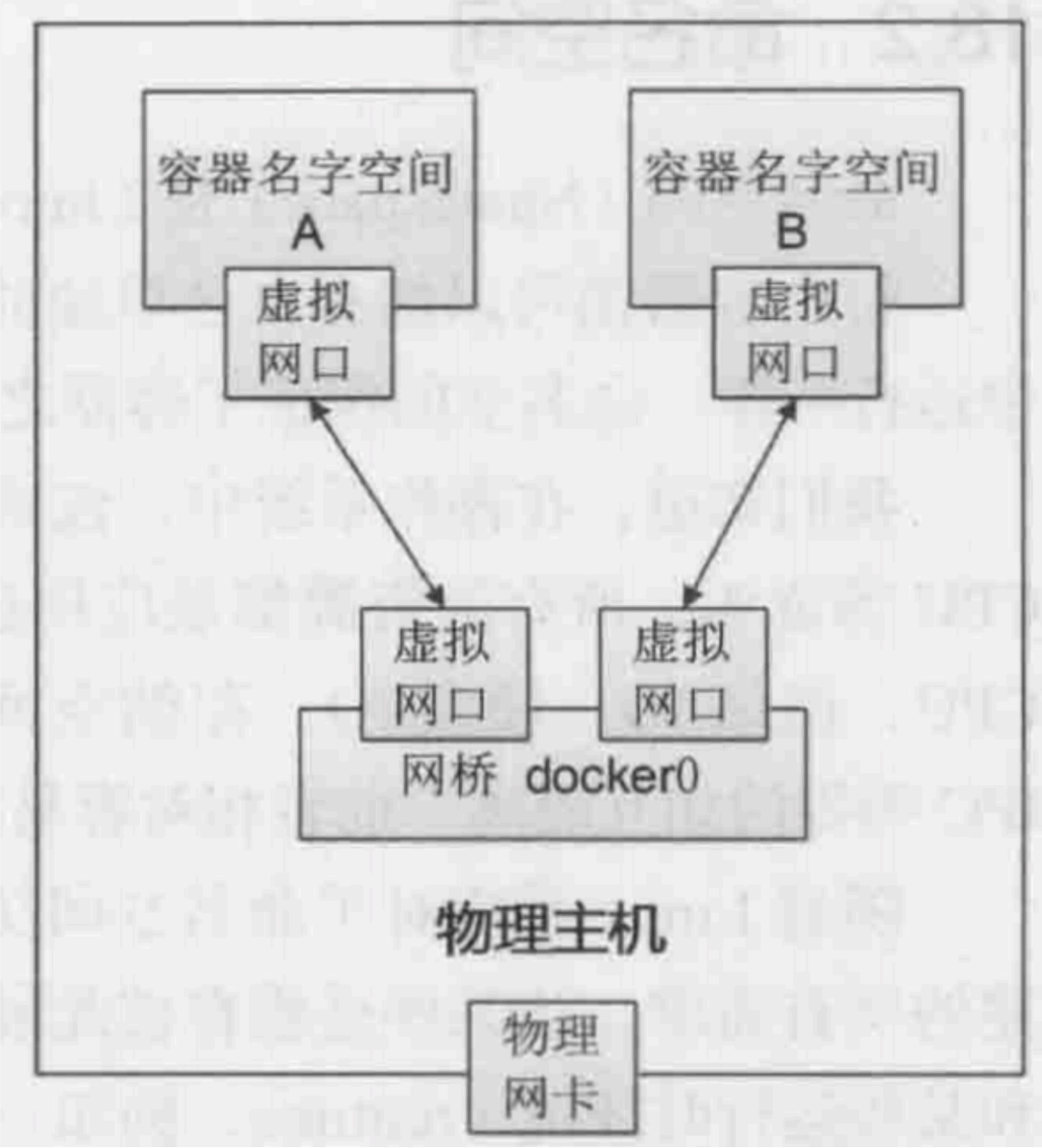
1. 进程命名空间

Linux通过命名空间管理进程号PID，对于同一进程(同一个task_struct)，在不同的命名空间中，看到的进程号不相同，每个进程命名空间有一套自己的进程号管理方法。进程命名空间是一个父子关系的结构，子空间中的进程对于父空间是可见的。新fork出的进程在父命名空间和子命名空间将分别有一个进程号来对应。(容器启动的初始进程，其父进程正是Docker的主进程)。

通过PID命名空间，每个名字空间中的进程就可以互相隔离

1. 网络命名空间

网络命名空间为进程提供了一个完全独立的网络协议栈的视图(包括网络设备接口、IPv4和IPv6协议栈、IP路由表、防火墙规则、sockets等等)。这样每个容器的网络就可以隔离开来。Docker通过虚拟网络设备的方式，将不同命名空间的网络设备连接到一起。在默认情况下，容器的虚拟网卡将同本地主机上的docker0网桥连接在一起。



1. IPC命名空间

容器中的进程交互(IPC)还是采用了Linux常见的进程间交互方法，包括信号量、消息队列和共享内存等。PID的命名空间和IPC命名空间可以组合起来一起使用，同一个IPC命名空间内的进程彼此可见和交互，但不同命名空间的进程无法交互

1. 挂载命名空间

类似`chroot`，将一个进程放到一个特定的目录执行。挂载命名空间允许不同命名空间的进程看到的文件结构不同，这样每个命名空间看到的文件目录彼此被隔离

1. UTS命名空间

UTS命名空间允许每个容器拥有独立的主机名和域名，从而可以虚拟出一个独立主机名和网络空间的环境，就跟网络上一台独立的主机一样。

1. 用户命名空间

每个容器都可以由不同的用户和组id，即容器内使用特定的内部用户执行程序，而非本地系统上存在的用户。

> 每个容器内部都可以有root账号，跟宿主主机不在一个命名空间中。

控制组

控制组(CGroups)是Linux内核的一个特性，主要用来对共享资源进行隔离、限制、审计等。只有能控制分配到容器的资源，Docker才能避免多个容器同时运行时的系统资源竞争。

CGroups可以提供对容器的内存、CPU、磁盘IO等资源进行限制和计费管理。其功能主要有如下:

- 资源限制
- 优先级
- 资源审计
- 隔离
- 控制

可以在`/sys/fs/cgroup/memory/docker/`目录下看到对Docker组应用的各种限制项。

联合文件系统

联合文件系统(UnionFS)是一种轻量级的高性能分层文件系统，它支持将文件系统中的修改信息作为一次提交，并层层叠加，同时可以将不同目录挂载到同一个虚拟文件系统下。

> 联合文件系统是实现Docker镜像的技术基础。镜像可以通过分层来进行继承。

Docker中使用`AUFS`。当Docker利用镜像启动一个容器时，将利用镜像分配文件系统并且挂载一个新的可读写的层给容器，容器会在这个文件系统中创建，并且这个可读写的层被添加到镜像中。

容器网络

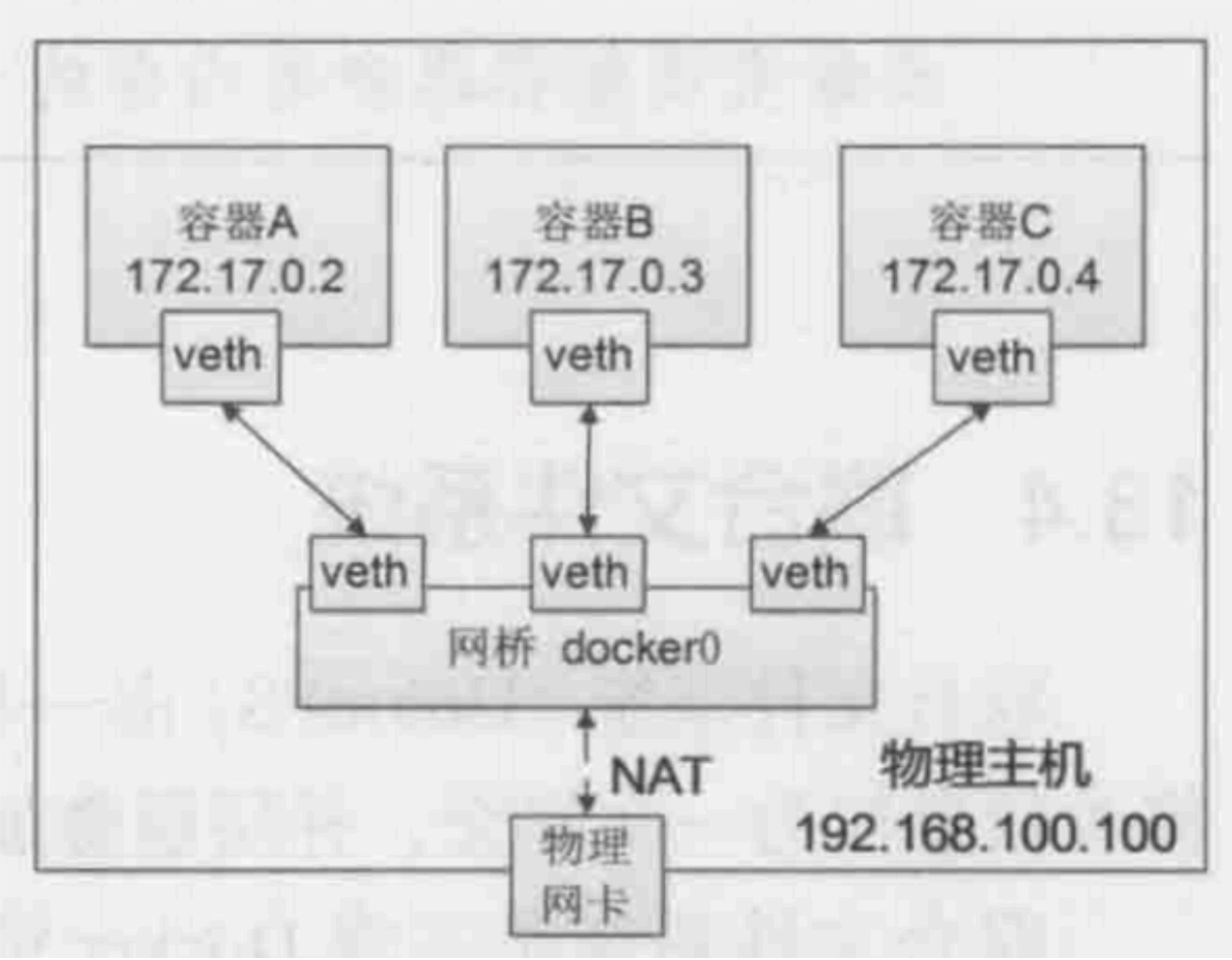
Docker的网络实现就是利用了Linux上的网络命名空间和虚拟网络设备(veth pair)

1. 基本原理

要实现网络通信，机器至少需要一个网络接口与外界通信，并可以收发数据包；此外，如果不同子网之间要进行通信，需要额外的路由机制。

Docker中的网络接口默认都是虚拟的接口（转发效率极高，因为内核中进行数据复制来实现虚拟接口之间的数据转发)，对于本地系统和容器内系统来看，虚拟接口跟一个正常的以太网卡相比没有区别，只是速度快得多。

Docker容器网络利用了Linux虚拟网络技术，它在本地主机和容器内分别创建了一个虚拟接口，并让它们彼此相连通



1. 网络创建过程

Docker创建一个容器的时候，会执行如下操作:

- 创建一对虚拟接口，分别连接本地主机和新容器的命名空间中
- 本地主机一端连接默认Docker0网桥或指定网桥上，并具有一个以`veth`开头的唯一ID
- 容器另一端的虚拟接口将放到新创建的容器中，并修改名字为`eth0`，这个接口仅在容器的命名空间中可见
- 从网桥可用地址段中获取一个空闲地址分配给容器的`eth0`,并配置默认路由网关为docker0网卡的内部接口dockr0的IP地址。

这样连接后，容器就可以使用它所能看到的eth0虚拟网卡来连接其他容器和访问外部网络。

此外，容器的运行可以通过`--net`来指定容器的网络配置

- bridge 默认值，在Docker网桥上为容器创建新的网络栈
- host 告诉Docker不要将容器网络放到隔离的命名空间中，即不要容器化容器内的网络。此时容器完全使用本地主机的网络，拥有本地主机接口访问权限。
- container 让docker将新建容器的进程放到一个已存在的容器网络栈中，新容器进程会有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享IP地址和端口等网络资源，两者进程可以通过lo环回接口通信
- none 让Docker将新容器放到隔离的网络栈中，但不进行网络配置，之后，用户可以自己进行配置。

跨节点网络

docker 实现跨节点网络需要解决的问题

1. 容器地址重复

   由于不同节点上的docker服务相互不知道对方的存在，因此在分配网络地址时就有可能出现鸡群里两个不同主机各自运行的docker容器获得相同的ip地址的情况。即使将这些局部网络直接连在一起也无法正确通信。

2. 容器地址 不可达

   即使所有节点的IP分配没有重复，各个节点上的主机路由表也不包含通往其他节点上的容器地址的路由信息，因此发往跨节点容器地址的信息还是会被丢弃。

覆盖网络（Overlay network）

覆盖网络在有些地方也被称为隧道网络，它的原理是在不改变现有网络基础设施的前提下，将传输层协议附加到另一种传输层协议或者更高层的协议之上，通过中继设备或者终端设备对这种额外的封装数据加以处理，从而实现扩展现有标准传输协议的目的。

常见基于覆盖网络实现的开源跨节点容器解决方案:

- Docker内置的 “Overlay” 网络插件：基于Vxlan实现的高效易用的跨节点网络，仅支持CNM接口。
- Weave 支持应用层封装的覆盖网络和基于Vxlan的覆盖网络，支持CNM,CNI网络模型。
- Flannel的UDP和Vxlan模式:CoreOS公司的通用覆盖网络解决方案，持应用层封装的覆盖网络和基于Vxlan封装的覆盖率，也支持基于扩展路由的网络连通方法，支持CNM,CNI网络模型。
- 开启IPIP模式的Calico 这是一种将IP包封装在IP数据包内，目的是为了在无法使用BGP协议的网络里面实现分段路由，从而将数据包挎节点投递到目标容器的IP地址，支持CNM和CNI网络模型。

扩展路由

跨节点网络的另一种实现思路是扩展路由。这种思路的原理是通过某种低成本的机制让容器IP和主机IP一样，可以直接在集群的网络基础设施上进行路由，从而解决容器跨节点通信的问题。特点是：在网络传输过程中没有额外的封装，没有NAT地址转发，性能普遍很好，接近原始网络带宽，并且由于传输过程使用的都是标准网络协议，在出现丢包等网络问题时，可以利用传统的网络工具进行排查。目前较多实际应用的扩展路由方案有：Macvlan，节点网关路由和节点BGP路由。

Macvlan

Macvlan技术能够在主机的网卡上添加多个mac地址(实际是虚拟出多个子网卡，如eth0.1 etho0.2 ...)所以在主机的外部网卡创建多个子网卡，让他们分别绑定ip地址和mac地址，再将每个子网卡分配给不同的Network Namespace ,这就相当于把所有容器的网络都直连到主机网卡上，使得每个容器都有一个外部可访问的Mac地址和IP地址，此外Macvlan本身支持划分虚拟局域网，因此可以避免广播风暴的发生。

节点网关路由

在每个节点上运行一个agent服务，然后用这个agent监听容器IP段分配的变化，一旦有变化就将新的路由规则刷新到本地的内核路由表中。这种被称为“主机网关”，其实在二层网络的环境中这是没有问题的，但一旦节点之间隔着另一个路由器，情况就没有这么顺利了。由于agent服务只存在于主机，但数据经过网络的路由器时，这个路由器没有相关的路由信息，就会将数据包丢弃。

节点BGP

主机网关由于无法跨越三层网络的原因在于，它无法改变在三层网络上其他路由器设备的路由规则。那么是否有一种方式能做到这点?

BGP协议,它是目前唯一能够处理因特网规模的网络路由协议，因此广泛存在于各种基础设施网络里。简单的来说，节点BGP路由网络的思路就是让安放在各个节点上的agent不仅修改本地内核路由表，还能实现路由路由BGP协议，把自己变成容器所在IP子网段的末端路由器，利用bgp协议具有的路由信息传播机制，让网络里的其他三层设备学习到容器网络段的路由信息，这就是节点BGP路由方案。

Calico是目前最主流的企业级节点BGP路由容器网络产品。

常见的基于扩展路由实现的开源跨节点容器网络解决方案:

- Docker自带的 Macvlan 网络插件，仅支持CNM接口。
- Calico(非IPIP模式) 基于节点BGP协议的路由的方案，支持很细致的ACL网络访问规则控制。
- Flannel 的HostGW模式 基于内核路由表和Iptables规则路由的方案。

容器命令

容器是Docker的另一个核心概念，它是镜像的一个运行实例，所不同的是，它带有额外的可写文件层。

如果认为虚拟机是模拟运行的一整套操作系统(提供了运行态环境和其他系统环境)和跑在上面的应用，那么Docker容器就是独立运行的一个或一组应用，以及它们的必须运行环境

创建容器

1. 可以使用`docker create`创建一个新的容器

```
docker create ubuntu:latest
```

使用`docker start`创建的容器会出与停止状态，可以使用`docker start`来启动它。

1. 启动容器还可以基于容器直接启动，需要使用`docker run`, 等价于先执行`docker create`,再执行`docker start`命令

```
docker run ubuntu /bin/echo "helloworld"
```

> 关于守护态与交互模式: 当使用`docker run -d`时，Docker会将容器以守护态的形式运行，容器会在后台持续运行命令直到进程结束；当使用`docker run -t -i`时，容器会分配一个伪终端(-t)，并绑定到容器的标准输入上(-i)，这样用户可以直接与容器进行命令交互

中止容器

可以使用`docker stop`来终止一个运行中的容器，它会首先向容器发送`SIGTERM信号`,过一段时间之后(默认为10s),再发送`SIGKILL信号`终止容器。当Docker容器应用终结时，容器也自动终止。

```
docker stop mycontainer
```

可以使用`docker ps -a`来查看到处于终止状态的容器ID信息。

处于终止状态的容器，可以通过`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，然后再启动它

进入容器

在启动容器时，如果使用`-d`参数，容器启动后会进入后台。要进入容器，有以下方法

1. attach

比如要进入`mycontainer`容器

```
docker attach myconainer
```

但是使用attach命令并不方便，当多个窗口同时attach到同一个容器，所有窗口都会同步显示，如果某个窗口阻塞，则所有窗口无法执行操作。

1. exec -ti

使用`exec`来执行命令，可以通过shell来访问容器

```
docker exec -ti mycontainer /bin/bash
```

删除容器

使用`docker rm`可以删除处于终止状态的容器

```
docker rm mycontainer
```

数据管理

用户在使用Docker的过程中，往往需要能查看容器内应用产生的数据，或者需要把容器内的数据进行备份，甚至多个容器之间进行数据的共享，这必然涉及容器的数据管理操作。

容器中管理数据的方式主要有两种:

- 数据卷
- 数据卷容器

数据卷

数据卷是一个可供容器使用的特殊目录，它绕过文件系统，提供:

- 可以在容器之间共享和重用
- 对数据卷的修改会马上生效

- 对数据卷的更新，不会影响到镜像
- 卷会一直存在，知道没有容器使用

当使用`docker run`的时候，使用`-v`可以在容器内创建一个数据卷，多次使用`-v`可以创建多个数据卷

```
docker run -d -v /webapp ubuntu
```

其实，以上只是容器在默认目录创建了一个空白卷，然后将空白卷映射到容器内的路径。如果要指定挂载本地一个已有的目录，

```
docker run -d -v /webapp:/root/webapp ubuntu
```

以上命令会将本地`/root/webapp`目录挂载到容器的`/webapp`目录中。

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

```
docker run -d -v /webapp:/root/webapp:ro ubuntu
```

> 如果直接挂载一个文件到容器，对文件进行编辑时，可能会造成文件inode的改变，所以推荐直接挂载文件所在的目录

数据卷容器

如果用户需要在容器直接共享一些持续更新的数据，可以使用数据卷容器。数据卷容器实际就是一个普通的容器。

1. 首先创建一个数据卷容器mydata,并创建一个空白卷

```
docker run -d -v /mydata --name mydata ubuntu
```

1. 然后其他容器就可以使用`--volumes-from`来挂载`mydata`容器中的数据卷了

```
docker run -ti --volumes-from mydata --name db1 ubuntu
docker run -ti --volumes-from mydata --name db2 ubuntu
```

1. 此时，`db1`的容器数据修改时，`db2`也可见其修改，同理，`db2`的修改对`db1`也是可见的

清理

1. 清理所有未使用的容器

```
docker rm `docker ps -a | grep Exited | awk '{print $1}'`
```

1. 清理所有标签为none的镜像

```
docker rmi -f  `docker images | grep '<none>' | awk '{print $3}'`
```

1. 删除未使用的volume

```
docker volume rm $(docker volume ls -qf dangling=true)
```

1. 删除所有没有用到的镜像

```
docker image prune -a
```