Docker 架构

Docker网络

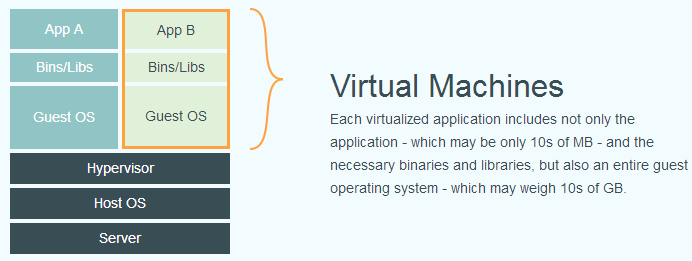
Docker存储

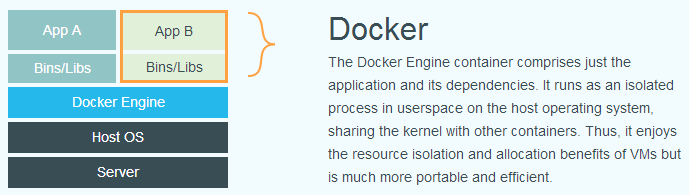
Docker 集群

Docker 优化

Docker 是一个开源项目，诞生于 2013 年初，最初是 dotCloud 公司内部的一个业余项目。它基于 Google 公司推出的 Go 语言实现。 项目后来加入了 Linux 基金会，遵从了 Apache 2.0 协议，项目代码在 GitHub 上进行维护。

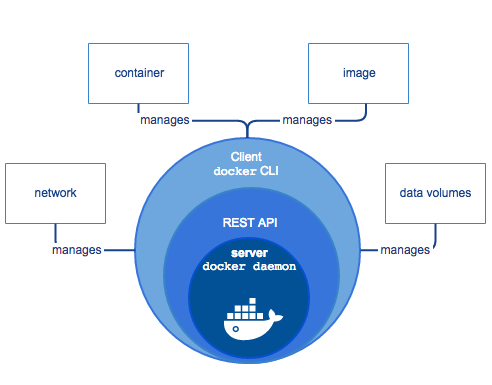
Docker 项目的目标是实现轻量级的操作系统虚拟化解决方案。 Docker 的基础是 Linux 容器（LXC）等技术





容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，而传统方式则是在硬件层面实现。

Docker架构图



Server是一个常驻进程

REST API 实现了client和server间的交互协议

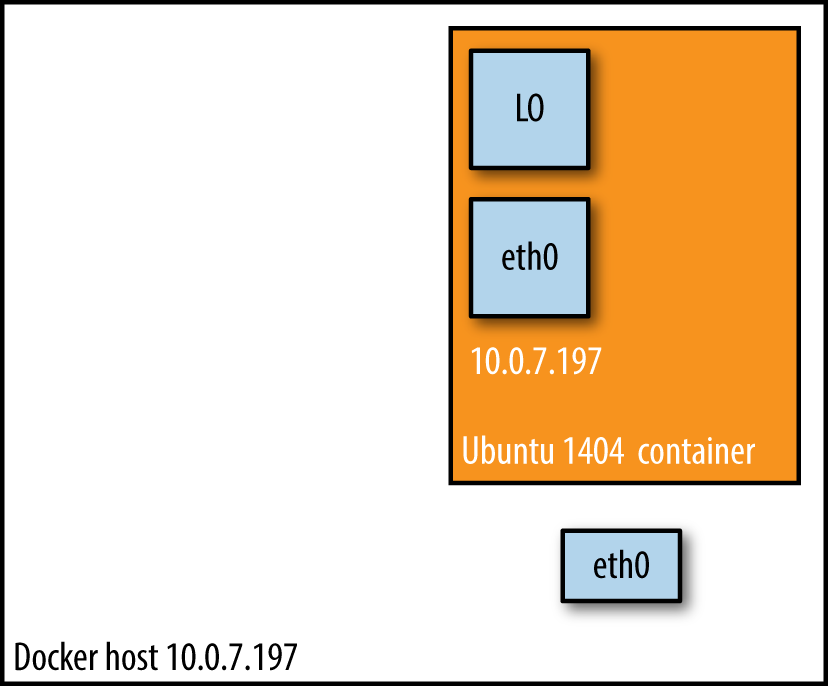
CLI 实现容器和镜像的管理，为用户提供统一的操作界面

------------------------------------------------------------------------------------------------

Docker 网络

五种网络模式(host, none, container, bridge,overlay)

Host模式



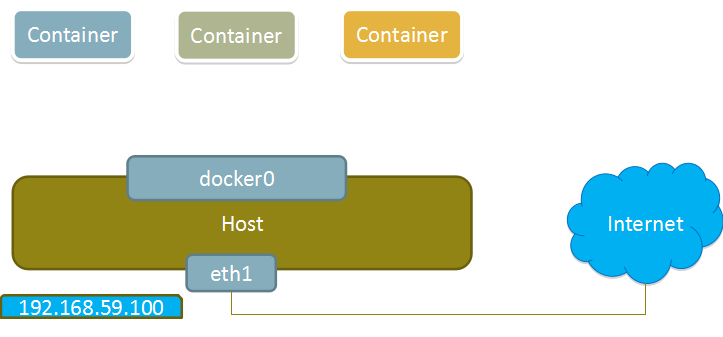
容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP等，而是使用宿主机的IP和端口。

在这种网络模式下，创建的容器大多数都是查看主机上的网络设备，且不安全。

命令是：

Docker run -it --net=host nginx /bin/sh

none模式：

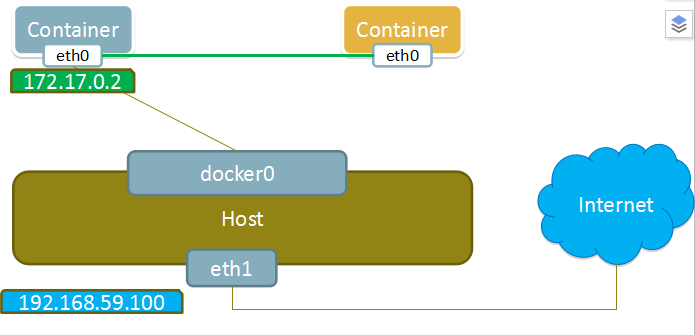


在none模式下，Docker容器拥有自己的Network Namespace，但是，并不为Docker容器进行任何网络配置。也就是说，这个Docker容器没有网卡、IP、路由等信息。需要我们自己为Docker容器添加网卡、配置IP等

命令是：

Docker run –it –net=none nginx /bin/sh

container模式：



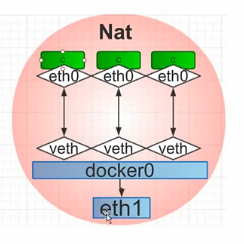
这个模式指定新创建的容器和已经存在的一个容器共享一个Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和一个指定的容器共享IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过lo网卡设备通信。

命令是：

Docker run –it –net=container nginx /bin/sh

bridge模式

bridge模式是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，并将并将一个主机上的Docker容器连接到一个虚拟网桥上。当Docker server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。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。接下来就要为容器分配IP了，Docker会从RFC1918所定义的私有IP网段中，选择一个和宿主机不同的IP地址和子网分配给docker0，连接到docker0的容器就从这个子网中选择一个未占用的IP使用。

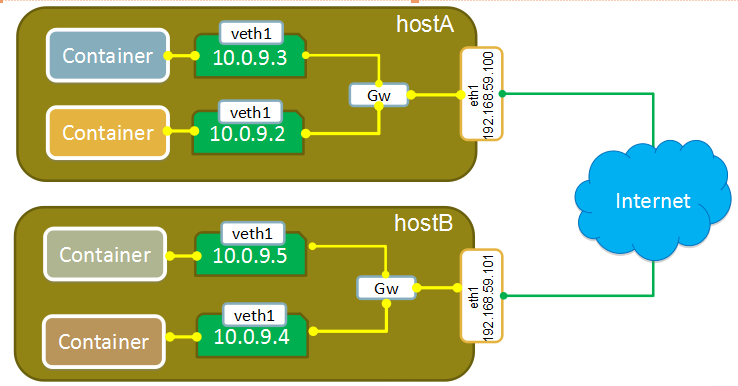


1. 在主机上创建一对虚拟网卡veth pair设备。veth设备总是成对出现的，它们组成了一个数据的通道，数据从一个设备进入，就会从另一个设备出来。因此，veth设备常用来连接两个网络设备。

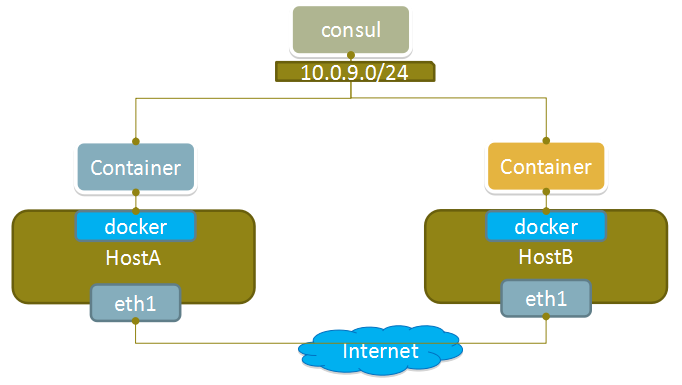
2. Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0。另一端放在主机中，以veth65f9这样类似的名字命名，并将这个网络设备加入到docker0网桥中，可以通过brctl show命令查看。

3. 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。

Overlay模式



容器在两个跨主机进行通信的时候，是使用overlay network这个网络模式进行通信，如果使用host也可以实现跨主机进行通信，直接使用这个物理的ip地址就可以进行通信。overlay它会虚拟出一个网络比如10.0.9.3这个ip地址，在这个overlay网络模式里面，有一个类似于服务网关的地址，然后把这个包转发到物理服务器这个地址，最终通过路由和交换，到达另一个服务器的ip地址。



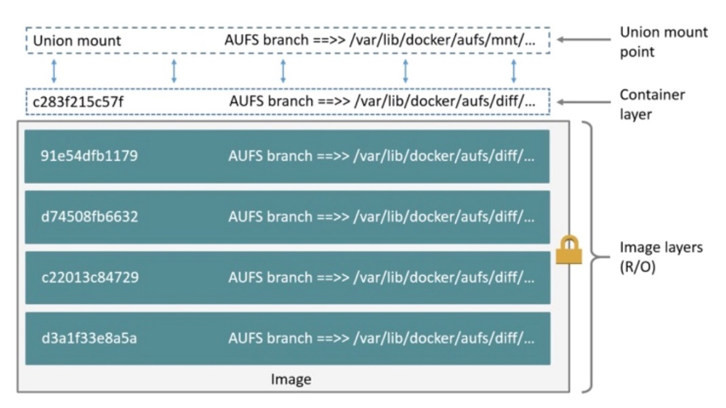
我们会有一个服务发现，比如说是consul，会定义一个ip地址池，比如10.0.9.0/24之类的，上面会有容器，容器的ip地址会从上面去获取，获取完了后，会通过eth1进行通信，实现跨主机的通信。

------------------------------------------------------------------------------------------------

在Docker 0.7版本中引入了存储驱动， 目前，Docker支持AUFS、Btrfs、Device mapper、OverlayFS、ZFS五种存储驱动。就如Docker官网上说的，没有单一的驱动适合所有的应用场景，要根据不同的场景选择合适的存储驱动，才能有效的提高Docker的性能。

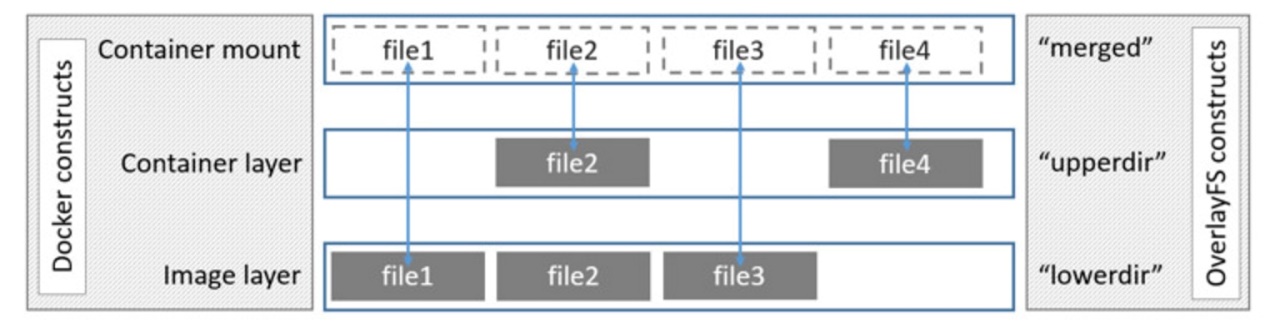
AUFS（文件级存储）

AUFS（AnotherUnionFS）是一种Union FS，是文件级的存储驱动。AUFS能透明覆盖一或多个现有文件系统的层状文件系统，把多层合并成文件系统的单层表示。简单来说就是支持将不同目录挂载到同一个虚拟文件系统下的文件系统。这种文件系统可以一层一层地叠加修改文件。无论底下有多少层都是只读的，只有最上层的文件系统是可写的。当需要修改一个文件时，AUFS创建该文件的一个副本，使用CoW将文件从只读层复制到可写层进行修改，结果也保存在可写层。在Docker中，底下的只读层就是image，可写层就是Container。如图:



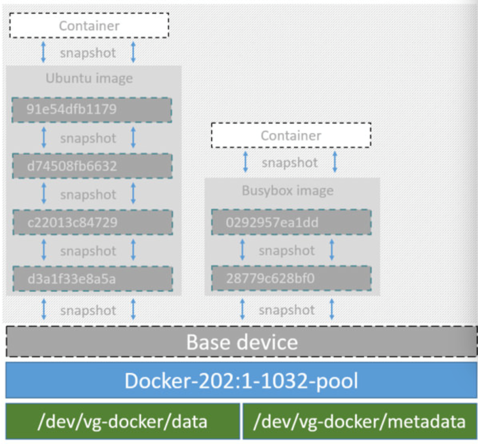
Overlay（文件级存储）

Overlay是Linux内核3.18后支持的，也是一种Union FS，和AUFS的多层不同的是Overlay只有两层：一个upper文件系统和一个lower文件系统，分别代表Docker的镜像层和容器层。当需要修改一个文件时，使用CoW技术将文件从只读的lower复制到可写的upper进行修改，结果也保存在upper层。在Docker中，底下的只读层就是image，可写层就是Container。结构如下图所示：



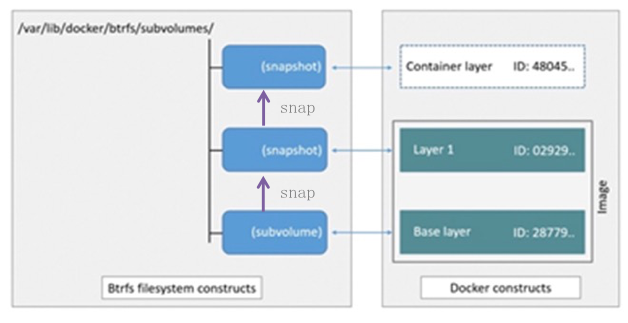
Device mapper(块级存储)

Device mapper是Linux内核2.6.9后支持的，提供的一种从逻辑设备到物理设备的映射框架机制，在该机制下，用户可以很方便的根据自己的需要制定实现存储资源的管理策略。前面讲的AUFS和OverlayFS都是文件级存储，而Device mapper是块级存储，所有的操作都是直接对块进行操作，而不是文件。Device mapper驱动会先在块设备上创建一个资源池，然后在资源池上创建一个带有文件系统的基本设备，所有镜像都是这个基本设备的快照，而容器则是镜像的快照。所以在容器里看到文件系统是资源池上基本设备的文件系统的快照，并不为容器分配空间。当要写入一个新文件时，在容器的镜像内为其分配新的块并写入数据，这个叫用时分配。当要修改已有文件时，再使用CoW为容器快照分配块空间，将要修改的数据复制到在容器快照中新的块里再进行修改。Device mapper 驱动默认会创建一个100G的文件包含镜像和容器。每一个容器被限制在10G大小的卷内，可以自己配置调整。结构如下图所示：



Btrfs（文件级存储）

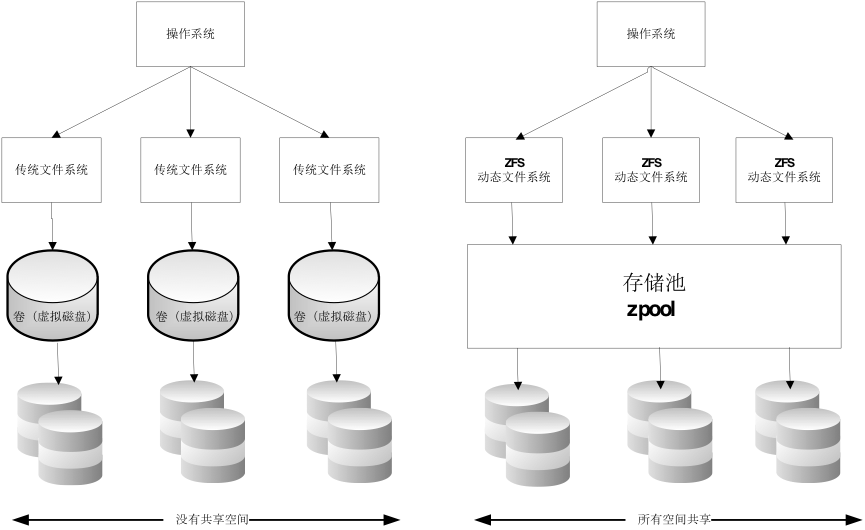
Btrfs被称为下一代写时复制文件系统，并入Linux内核，也是文件级级存储，但可以像Device mapper一直接操作底层设备。Btrfs把文件系统的一部分配置为一个完整的子文件系统，称之为subvolume 。那么采用 subvolume，一个大的文件系统可以被划分为多个子文件系统，这些子文件系统共享底层的设备空间，在需要磁盘空间时便从底层设备中分配，类似应用程序调用 malloc()分配内存一样。为了灵活利用设备空间，Btrfs 将磁盘空间划分为多个chunk（块） 。每个chunk可以使用不同的磁盘空间分配策略。比如某些chunk只存放metadata，某些chunk只存放数据。这种模型有很多优点，比如Btrfs支持动态添加设备。用户在系统中增加新的磁盘之后，可以使用Btrfs的命令将该设备添加到文件系统中。Btrfs把一个大的文件系统当成一个资源池，配置成多个完整的子文件系统，还可以往资源池里加新的子文件系统，而基础镜像则是子文件系统的快照，每个子镜像和容器都有自己的快照，这些快照则都是subvolume的快照。



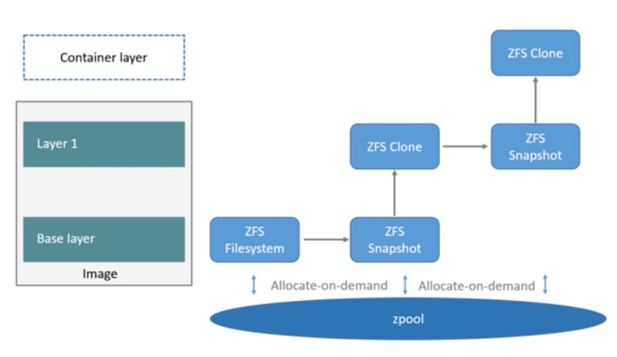
当写入一个新文件时，为在容器的快照里为其分配一个新的数据块，文件写在这个空间里，这个叫用时分配。而当要修改已有文件时，使用CoW复制分配一个新的原始数据和快照，在这个新分配的空间变更数据，变结束再更新相关的数据结构指向新子文件系统和快照，原来的原始数据和快照没有指针指向，被覆盖。

ZFS（存储池）

ZFS 文件系统是一个革命性的全新的文件系统，它从根本上改变了文件系统的管理方式，ZFS 完全抛弃了“卷管理”，不再创建虚拟的卷，而是把所有设备集中到一个存储池中来进行管理，用“存储池”的概念来管理物理存储空间。过去，文件系统都是构建在物理设备之上的。为了管理这些物理设备，并为数据提供冗余，“卷管理”的概念提供了一个单设备的映像。而ZFS创建在虚拟的，被称为“zpools”的存储池之上。每个存储池由若干虚拟设备（virtual devices，vdevs）组成。这些虚拟设备可以是原始磁盘，也可能是一个RAID1镜像设备，或是非标准RAID等级的多磁盘组。于是zpool上的文件系统可以使用这些虚拟设备的总存储容量。



下面看一下在Docker里ZFS的使用。首先从zpool里分配一个ZFS文件系统给镜像的基础层，而其他镜像层则是这个ZFS文件系统快照的克隆，快照是只读的，而克隆是可写的，当容器启动时则在镜像的最顶层生成一个可写层。如下图所示



当要写一个新文件时，使用按需分配，一个新的数据快从zpool里生成，新的数据写入这个块，而这个新空间存于容器（ZFS的克隆）里。

当要修改一个已存在的文件时，使用写时复制，分配一个新空间并把原始数据复制到新空间完成修改。

存储驱动的对比及适应场景：



AUFS VS Overlay

AUFS和Overlay都是联合文件系统，但AUFS有多层，而Overlay只有两层，所以在做写时复制操作时，如果文件比较大且存在比较低的层，则AUSF可能会慢一些。而且Overlay并入了linux kernel mainline，AUFS没有，所以可能会比AUFS快。但Overlay还太年轻，要谨慎在生产使用。而AUFS做为docker的第一个存储驱动，已经有很长的历史，比较的稳定，且在大量的生产中实践过，有较强的社区支持。目前开源的DC/OS指定使用Overlay。

Overlay VS Device mapper

Overlay是文件级存储，Device mapper是块级存储，当文件特别大而修改的内容很小，Overlay不管修改的内容大小都会复制整个文件，对大文件进行修改显示要比小文件要消耗更多的时间，而块级无论是大文件还是小文件都只复制需要修改的块，并不是整个文件，在这种场景下，显然device mapper要快一些。因为块级的是直接访问逻辑盘，适合IO密集的场景。而对于程序内部复杂，大并发但少IO的场景，Overlay的性能相对要强一些。

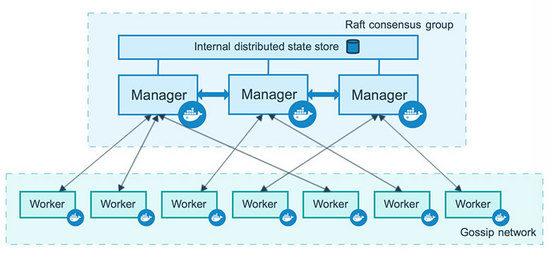
Device mapper VS Btrfs Driver VS ZFS

Device mapper和Btrfs都是直接对块操作，都不支持共享存储，表示当有多个容器读同一个文件时，需要生活多个复本，所以这种存储驱动不适合在高密度容器的PaaS平台上使用。而且在很多容器启停的情况下可能会导致磁盘溢出，造成主机不能工作。Device mapper不建议在生产使用。Btrfs在docker build可以很高效。

ZFS最初是为拥有大量内存的Salaris服务器设计的，所在在使用时对内存会有影响，适合内存大的环境。ZFS的COW使碎片化问题更加严重，对于顺序写生成的大文件，如果以后随机的对其中的一部分进行了更改，那么这个文件在硬盘上的物理地址就变得不再连续，未来的顺序读会变得性能比较差。ZFS支持多个容器共享一个缓存块，适合PaaS和高密度的用户场景。

Docker集群：

Swamp



Swarm中的管理节点都是通过实现Raft一致性算法来管理所有节点的状态。通过实现Raft Consensus Algorithm，可以确保集群中的所有任务、存储等都是一致的状态

它分为两类节点，manager 和 worker。

manager 节点处理集群管理任务：

维护集群状态（配置）

调度服务

提供集群模式 HTTP API 端点服务

管理节点共同维护一个强一致状态数据（你可以认为每个管理节点拥有一个同步的集群状态/配置数据库），任何一个经理退出都不会导致集群失效。

高可靠模式 是指容器集群拥有 3 个以上 manager。 manager 的奇数（如 3、5、7），一半以下 manager 失效都不会导致集群状态失效。Docker为群组建议最多七个管理器节点。

多个 7 个 manager 不会提高可靠性，反而会提高开销

强一致状态数据库常见实现包括： etcd 、consul 、zookeeper 等等。

worker 节点的唯一目的就是运行容器（服务）。默认情况下，manager 同时承担 worker 的职能，所以一个 manager 可以单节点运行，所有容器运行在一个节点中。

对于大型的容器集群，需要设置管理器节点的可用性设置为 Drain 以阻止服务的任务运行。

docker node update 是一个很有用的命令，可以改变节点的 role，label 等

docker优化

镜像优化：

1运行自己私有的Docker registry用于存储和分发Docker镜像。

2改变dockerfile语序以复用缓存（即将dockerfile中含有update或者install的语句提前）

当我们在构建Dockerfile时，docker会检查dockerfile中正在处理的指令，判断在它的缓存机制中是否存在可以复用的镜像，而不是重复创建一个一样的镜像。通过改变语句如“RUN apt-get update”或者"RUN bundle install"或者“RUN yum -y update”等更新程序包等语句，对依赖库进行一次操作而为重复操作。在大多数情况下。我们更新的只是应用程序中的核心部分

将更新语句集结到Dockerfile最前段，可有效节省修改后构建的时间

内核参数优化

Docker安装：

docker安装：

关闭防火墙

systemctl stop firewalld.service && systemctl disable firewalld.service

setenforce 0

修改配置文件

sed -i.bak's/SELINUX=enforcing/SELINUX=permissive/'/etc/selinux/config

创建一个文件

vi /etc/yum.repos.d/docker.repo

在文件中添加

[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 -y docker-engine

Docker常用命令：

docker格式

docker [option][command][arguments]

option:

build 生成一个image文件，通过Dockerfile

comit 将现有的容器保存为一个镜像

create 创建一个容器

diff 检测容器系统的变更

exec 在运行的容器中运行容器的内部命令

export 将一个容器的文件系统输出为一个压缩包

images 列出所有容器的镜像列表

import 从export的tar包中导入一个容器

info 显示系统信息

inspect 返回容器的底层信息

kill 删除容器

load 从export 的tar包中加载容器

logs 查看容器的日志

network 查看容器的网络信息

port 查看容器的暴露到前端（主机服务器）的端口

ps 列出容器 -a docker ps --help

pull 拉镜像，从网络上（HUB，分为公有和私有）

push 推镜像

rm 删除容器命令

stop 停止容器

rmi 删除容器的镜像

save 保存镜像至tar包

search 从互联网上找镜像文件

star 启动

tag 给镜像打标记

top 显示容器的进程

volume 管理容器的卷

run 运行容器

comment：