docker学习笔记



0、 ssh docker@Boot2DockerVM\_IP ##进入到VM虚拟机Boot2Docker系统下，用户docker,密码tcuser 【windows Boot2Docker 虚拟机方式】

1、 docker-machine ls ## 列出docker机器【windows Boot2Docker 虚拟机方式】,比如有default这台docker机器

2、 docker-machine ssh default 这句命令加上序号1，是同上序号0的作用一样的。【windows Boot2Docker 虚拟机方式】

***tipe1***： 上面三步也只是相当于进入到安装了docker的linux系统

3、 sudo -i 在docker用户su到root用户【windows Boot2Docker 虚拟机方式,在设置一些变量时用到，比如配置国内的docker仓库】

## 使用Device Mapper存储驱动程序

*预计阅读时间： 26分钟*

Device Mapper是一个基于内核的框架，在Linux上支持许多先进的卷管理技术。Docker的devicemapper存储驱动程序利用此框架的精简配置和快照功能进行图像和容器管理。本文参考Device Mapper存储驱动程序devicemapper，并将内核框架作为Device Mapper。

对于支持的系统，devicemapperLinux内核中包含支持。但是，需要具体配置才能使用Docker。例如，在RHEL或CentOS的库存安装中，Docker将默认overlay为不支持的配置。

该devicemapper驱动程序使用专用于多克尔块设备和在块级，而不是文件级别运行。可以通过向Docker主机添加物理存储来扩展这些设备，并且它们的性能要优于在操作系统级使用文件系统。

**如果在RHEL，CentOS或Oracle Linux上使用Docker EE，则必须使用 devicemapper存储驱动程序。**

### 先决条件

* devicemapper存储驱动程序是RHEL，CentOS和Oracle Linux上Docker EE和商业支持Docker Engine（CS-Engine）唯一支持的存储驱动程序。请参阅 [产品兼容性矩阵](https://success.docker.com/Policies/Compatibility_Matrix)。
* devicemapper 在CentOS，Fedora，Ubuntu或Debian上运行的Docker CE也支持。
* 更改存储驱动程序将使您在本地系统上创建的任何容器无法访问。使用docker save保存的容器，并推动现有图像多克尔集线器或私人仓库，让你不必后重新创建它们。

### 配置Docker与devicemapper存储驱动程序

在遵循这些程序之前，您必须首先满足所有 [先决条件](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#prerequisites)。

#### 配置loop-lvm模式进行测试

此配置仅适用于测试。环回设备速度慢，资源密集，需要您以特定大小在磁盘上创建文件。他们也可以引入种族条件。它们应该用于测试，因为设置更容易。

有关生产系统，请参阅 [配置直接生产模式进行生产](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production)。

1. 停止码头
2. $ sudo systemctl stop docker
3. 编辑/etc/docker/daemon.json。如果还不存在，创建它。假设文件为空，请添加以下内容。
4. {
5. "storage-driver": "devicemapper"
6. }

如果daemon.json文件包含格式不正确的JSON，Docker将无法启动。

1. 启动Docker。
2. $ sudo systemctl start docker
3. 验证守护程序是否正在使用devicemapper存储驱动程序。使用 docker info命令并寻找Storage Driver。
4. $ docker info
5. Containers: 0
6. Running: 0
7. Paused: 0
8. Stopped: 0
9. Images: 0
10. Server Version: 17.03.1-ce
11. Storage Driver: devicemapper
12. Pool Name: docker-202:1-8413957-pool
13. Pool Blocksize: 65.54 kB
14. Base Device Size: 10.74 GB
15. Backing Filesystem: xfs
16. Data file: /dev/loop0
17. Metadata file: /dev/loop1
18. Data Space Used: 11.8 MB
19. Data Space Total: 107.4 GB
20. Data Space Available: 7.44 GB
21. Metadata Space Used: 581.6 kB
22. Metadata Space Total: 2.147 GB
23. Metadata Space Available: 2.147 GB
24. Thin Pool Minimum Free Space: 10.74 GB
25. Udev Sync Supported: true
26. Deferred Removal Enabled: false
27. Deferred Deletion Enabled: false
28. Deferred Deleted Device Count: 0
29. Data loop file: /var/lib/docker/devicemapper/devicemapper/data
30. Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
31. Library Version: 1.02.135-RHEL7 (2016-11-16)
32. <output truncated>

该主机正在loop-lvm节点中运行，生产系统**不**支持此主机。这是由Data loop file a和a Metadata loop file在文件下 的事实表明的/var/lib/docker/devicemapper/devicemapper。这些是环回安装的稀疏文件。有关生产系统，请参阅 [配置直接生产模式进行生产](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production)。

#### 配置direct-lvm模式进行生产

使用devicemapper存储驱动程序的生产主机必须使用direct-lvm 模式。此模式使用块设备创建精简池。这比使用环回设备更快，更有效地使用系统资源，并且块设备可以根据需要增长。但是，需要比loop-lvm 模式更多的设置。

满足[先决条件后](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#prerequisites)，请按照以下步骤配置Docker以使用devicemapper存储驱动程序 direct-lvm。

**警告**：更改存储驱动程序将使您在本地系统上创建的任何容器无法访问。使用docker save保存的容器，并推动现有图像多克尔集线器或私人仓库，这样就不需要以后重新创建。

##### 允许DOCKER配置直接LVM​​模式

在Docker 17.06及更高版本中，Docker可以为您管理块设备，简化了direct-lvm模式的配置。**这仅适用于新鲜的Docker设置。**您只能使用单个块设备。如果需要使用多个块设备，请[手动配置direct-lvm模式](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-manually)。添加了以下新配置选项：

| **选项** | **描述** | **需要？** | **默认** | **例** |
| --- | --- | --- | --- | --- |
| dm.directlvm\_device | 要设置的块设备的路径direct-lvm。 | 是 |  | dm.directlvm\_device="/dev/xvdf" |
| dm.thinp\_percent | 从传递的块设备中用于存储的空间的百分比。 | 没有 | 95 | dm.thinp\_percent=95 |
| dm.thinp\_metapercent | 元数据存储的空间百分比来自block = | 没有 | 1 | dm.thinp\_metapercent=1 |
| dm.thinp\_autoextend\_threshold | lvm应自动将瘦池扩展为占总存储空间百分比的阈值。 | 没有 | 80 | dm.thinp\_autoextend\_threshold=80 |
| dm.thinp\_autoextend\_percent | 触发自动延伸时增加精简池的百分比。 | 没有 | 20 | dm.thinp\_autoextend\_percent=20 |
| dm.directlvm\_device\_force | 是否格式化块设备，即使文件系统已经存在于其上。如果设置为false文件系统，则会记录错误，文件系统保持原样。 | 没有 | 假 | dm.directlvm\_device\_force=true |

编辑daemon.json文件并设置相应的选项，然后重新启动Docker以使更改生效。以下daemon.json设置上表中的所有选项。

{

"storage-driver": "devicemapper",

"storage-opts": [

"dm.directlvm\_device=/dev/xdf",

"dm.thinp\_percent=95",

"dm.thinp\_metapercent=1",

"dm.thinp\_autoextend\_threshold=80",

"dm.thinp\_autoextend\_percent=20",

"dm.directlvm\_device\_force=false"

]

}

重新启动Docker以使更改生效。Docker调用命令为您配置块设备。

**警告**：在Docker准备好您的块设备后，更改这些值不受支持，并将导致错误。

您仍然需要[执行定期的维护任务](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#manage-devicemapper)。

##### 手动配置DIRECT-LVM模式

以下过程将创建一个配置为精简池的逻辑卷，以用作存储池的后备。它假设您有一个备用块设备，/dev/xvdf具有足够的可用空间来完成任务。您的环境中的设备标识符和卷大小可能不同，您应该在整个过程中替换自己的值。该过程还假定Docker守护程序处于该stopped状态。

1. 识别要使用的块设备。该设备将位于/dev/（如/dev/xvdf）下方 ，并需要足够的可用空间来存储主机将运行的工作负载的映像和容器层。理想情况下，这将是一个坚实的驱动。
2. 停止码头
3. $ sudo systemctl stop docker
4. 安装以下软件包：
   * **RHEL / CentOS的**：device-mapper-persistent-data，lvm2，和所有的依赖
   * **Ubuntu的/ Debian的**：thin-provisioning-tools，lvm2，和所有的依赖
5. 使用pvcreate命令从步骤1在块设备上创建物理卷 。替换您的设备名称/dev/xvdf。

**警告**：接下来的几个步骤是破坏性的，因此请确保您指定了正确的设备！

$ sudo pvcreate /dev/xvdf

Physical volume "/dev/xvdf" successfully created.

1. docker使用该vgcreate 命令在同一设备上创建卷组。
2. $ sudo vgcreate docker /dev/xvdf
3. Volume group "docker" successfully created
4. 创建两个命名的逻辑卷thinpool并thinpoolmeta使用该 lvcreate命令。最后一个参数指定了如果空间不足，允许自动扩展数据或元数据的空闲空间，作为临时停顿。这些是推荐值。
5. $ sudo lvcreate --wipesignatures y -n thinpool docker -l 95%VG
6. Logical volume "thinpool" created.
7. $ sudo lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG
8. Logical volume "thinpoolmeta" created.
9. 使用命令将卷转换为精简池和瘦池的元数据的存储位置lvconvert。
10. $ sudo lvconvert -y \
11. --zero n \
12. -c 512K \
13. --thinpool docker/thinpool \
14. --poolmetadata docker/thinpoolmeta
15. WARNING: Converting logical volume docker/thinpool and docker/thinpoolmeta to
16. thin pool's data and metadata volumes with metadata wiping.
17. THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)
18. Converted docker/thinpool to thin pool.
19. 通过lvm配置文件配置瘦池的自动扩展。
20. $ sudo vi /etc/lvm/profile/docker-thinpool.profile
21. 指定thin\_pool\_autoextend\_threshold和thin\_pool\_autoextend\_percent 值。

thin\_pool\_autoextend\_threshold是lvm 尝试自动扩展可用空间之前使用的空间百分比（100 =禁用，不推荐）。

thin\_pool\_autoextend\_percent 是自动扩展时添加到设备的空间量（0 =禁用）。

当磁盘使用率达到80％时，下面的示例将增加20％的容量。

activation {

thin\_pool\_autoextend\_threshold=80

thin\_pool\_autoextend\_percent=20

}

保存文件。

1. 使用lvchange命令应用LVM配置文件。
2. $ sudo lvchange --metadataprofile docker-thinpool docker/thinpool
3. Logical volume docker/thinpool changed.
4. 启用对主机上逻辑卷的监视。没有此步骤，即使存在LVM配置文件，也不会发生自动扩展。
5. $ sudo lvs -o+seg\_monitor
6. LV VG Attr LSize Pool Origin Data% Meta% Move Log Cpy%Sync Convert Monitor
7. thinpool docker twi-a-t--- 95.00g 0.00 0.01 monitored
8. 如果您曾经在此主机上运行Docker，或者如果/var/lib/docker/ 存在，则将其移出方式，以便Docker可以使用新的LVM池来存储映像和容器的内容。
9. $ mkdir /var/lib/docker.bk
10. $ mv /var/lib/docker/\* /var/lib/docker.bk

如果以下步骤中的任何一个失败，并且需要还原，则可以将/var/lib/docker其删除 并替换/var/lib/docker.bk。

1. 编辑/etc/docker/daemon.json和配置devicemapper存储驱动程序所需的选项 。如果文件以前是空的，它现在应该包含以下内容：

**注意**：dm.use\_deferred\_deletion=true使用默认内核版本3.18时，RHEL，CentOS或Ubuntu 14.04尚不支持延迟删除选项。

{

"storage-driver": "devicemapper",

"storage-opts": [

"dm.thinpooldev=/dev/mapper/docker-thinpool",

"dm.use\_deferred\_removal=true",

"dm.use\_deferred\_deletion=true"

]

}

1. 启动Docker。

**系统**：

$ sudo systemctl start docker

**服务**：

$ sudo service docker start

1. 验证Docker是否正在使用新配置docker info。
2. $ docker info
3. Containers: 0
4. Running: 0
5. Paused: 0
6. Stopped: 0
7. Images: 0
8. Server Version: 17.03.1-ce
9. Storage Driver: devicemapper
10. Pool Name: docker-thinpool
11. Pool Blocksize: 524.3 kB
12. Base Device Size: 10.74 GB
13. Backing Filesystem: xfs
14. Data file:
15. Metadata file:
16. Data Space Used: 19.92 MB
17. Data Space Total: 102 GB
18. Data Space Available: 102 GB
19. Metadata Space Used: 147.5 kB
20. Metadata Space Total: 1.07 GB
21. Metadata Space Available: 1.069 GB
22. Thin Pool Minimum Free Space: 10.2 GB
23. Udev Sync Supported: true
24. Deferred Removal Enabled: true
25. Deferred Deletion Enabled: true
26. Deferred Deleted Device Count: 0
27. Library Version: 1.02.135-RHEL7 (2016-11-16)
28. <output truncated>

如果Docker配置正确，Data file并且Metadata file将为空，并且池名称将为空docker-thinpool。

1. 验证配置正确后，可以删除/var/lib/docker.bk包含上一个配置的 目录。

$ rm -rf /var/lib/docker.bk

### 管理devicemapper

#### 监控瘦池

不要单靠LVM自动扩展。卷组将自动扩展，但音量仍可以填满。您可以使用lvs或监视卷上的可用空间lvs -a。考虑在操作系统级别使用监视工具，如Nagios。

要查看LVM日志，可以使用journalctl：

$ journalctl -fu dm-event.service

如果在瘦池遇到重复问题，可以将存储选项设置 dm.min\_free\_space为一个值（代表一个百分比）/etc/docker.daemon.json。例如，将其设置为10确保在可用空间处于或接近10％时操作失败并发出警告。请参阅[引擎守护程序引用中](https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)的 [存储驱动程序选项](https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)。

#### 增加运行设备的容量

您可以在运行的瘦池设备上增加池的容量。如果数据的逻辑卷已满并且卷组处于满容量，这将非常有用。具体过程取决于您是否使用 [循环lvm精简池](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#resize-a-loop-lvm-thin-pool)或 [直接lvm​​精简池](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#resize-a-direct-lvm-thin-pool)。

##### 调整一个LOOP-LVM的精简池

调整loop-lvm精简池的最简单方法是 [使用device\_tool实用程序](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#use-the-device_tool-utility)，但可以[使用操作系统实用程序](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#use-operating-system-utilities) 。

###### 使用device\_tool实用程序

在Docker Github仓库device\_tool.go的contrib/目录中提供了 一个社区贡献的脚本。您可以使用此工具来调整loop-lvm精简池的大小，从而避免上述漫长过程。该工具不能保证工作，但您应该只loop-lvm在非生产系统上使用。

如果不想使用device\_tool，可以手动调整瘦池大小（＃use-operating-system-utilities）。

1. 要使用该工具，请克隆Github资源库，更改为 contrib/docker-device-tool并遵循README.md 编译工具中的说明。
2. 使用该工具。以下示例将精简池重新调整为200GB。
3. $ ./device\_tool resize 200GB

###### 使用操作系统实用程序

如果不想[使用设备工具实用程序](https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#use-the-device_tool-utility)，可以loop-lvm使用以下步骤手动调整瘦池的大小。

在loop-lvm模式下，环回设备用于存储数据，另一个用于存储元数据。loop-lvm模式仅用于测试，因为它具有显着的性能和稳定性缺点。

如果您使用的loop-lvm模式，输出docker info会显示文件路径Data loop file和Metadata loop file：

$ docker info |grep 'loop file'

Data loop file: /var/lib/docker/devicemapper/devicemapper/data

Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata

按照以下步骤来增加瘦池的大小。在这个例子中，瘦池是100 GB，增加到200 GB。

1. 列出设备的大小。
2. $ sudo ls -lh /var/lib/docker/devicemapper/devicemapper/
3. total 1175492
4. -rw------- 1 root root 100G Mar 30 05:22 data
5. -rw------- 1 root root 2.0G Mar 31 11:17 metadata
6. data使用truncate用于增加**或**减少文件大小的命令将文件的大小增加到200 G。请注意，减小大小是一个破坏性的操作。
7. $ sudo truncate -s 200G /var/lib/docker/devicemapper/devicemapper/data
8. 验证文件大小是否已更改。
9. $ sudo ls -lh /var/lib/docker/devicemapper/devicemapper/
10. total 1.2G
11. -rw------- 1 root root 200G Apr 14 08:47 data
12. -rw------- 1 root root 2.0G Apr 19 13:27 metadata
13. 环回文件在磁盘上已更改，但不在内存中。列出内存中环回设备的大小，以GB为单位。重新加载，然后再次列出大小。重新加载后，大小为200 GB。
14. $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]
15. 100
16. $ sudo losetup -c /dev/loop0
17. $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]
18. 200
19. 重新加载devicemapper精简池。

一个。先获取池名称。池名称是第一个字段，由`：`分隔。此命令提取它。

$ sudo dmsetup status | grep ' thin-pool ' | awk -F ': ' {'print $1'}

docker-8:1-123141-pool

湾 转储瘦池的设备映射表。

$ sudo dmsetup table docker-8:1-123141-pool

0 209715200 thin-pool 7:1 7:0 128 32768 1 skip\_block\_zeroing

C。使用输出的第二个字段计算精简池的总扇区。数字以512-k扇区表示。100G文件具有209715200 512k扇区。如果你把这个数字加倍到200G，你会得到419430400 512-k个扇区。

天。使用以下三个dmsetup 命令重新加载具有新扇区号的精简池。

$ sudo dmsetup suspend docker-8:1-123141-pool

$ sudo dmsetup reload docker-8:1-123141-pool --table '0 419430400 thin-pool 7:1 7:0 128 32768 1 skip\_block\_zeroing'

$ sudo dmsetup resume docker-8:1-123141-pool

##### 调整直接LVM​​精简池的大小

要扩展一个direct-lvm精简池，您需要首先将一个新的块设备附加到Docker主机上，并记下内核分配给它的名称。在这个例子中，新的块设备是/dev/xvdg。

按照这个步骤来扩展一个direct-lvm细小的池，用你的块设备和其他参数来代替你的情况。

1. 收集有关您的卷组的信息。

使用该pvdisplay命令查找您的瘦池当前正在使用的物理块设备以及卷组的名称。

$ sudo pvdisplay |grep 'VG Name'

PV Name /dev/xvdf

VG Name docker

在以下步骤中，根据需要替换您的块设备或卷组名称。

1. 扩展卷组，使用 上一步中的vgextend命令VG Name和**新的**块设备的名称。
2. $ sudo vgextend docker /dev/xvdg
3. Physical volume "/dev/xvdg" successfully created.
4. Volume group "docker" successfully extended
5. 扩展docker/thinpool逻辑卷。此命令立即使用100％的音量，无需自动扩展。要扩展元数据稀疏池，请使用docker/thinpool\_tmeta。
6. $ sudo lvextend -l+100%FREE -n docker/thinpool
7. Size of logical volume docker/thinpool\_tdata changed from 95.00 GiB (24319 extents) to 198.00 GiB (50688 extents).
8. Logical volume docker/thinpool\_tdata successfully resized.
9. 使用Data Space Available输出中的字段验证新的精简池大小docker info。如果扩展docker/thinpool\_tmeta逻辑卷，请查找Metadata Space Available。
10. Storage Driver: devicemapper
11. Pool Name: docker-thinpool
12. Pool Blocksize: 524.3 kB
13. Base Device Size: 10.74 GB
14. Backing Filesystem: xfs
15. Data file:
16. Metadata file:
17. Data Space Used: 212.3 MB
18. Data Space Total: 212.6 GB
19. Data Space Available: 212.4 GB
20. Metadata Space Used: 286.7 kB
21. Metadata Space Total: 1.07 GB
22. Metadata Space Available: 1.069 GB
23. <output truncated>

### 如何devicemapper存储驱动程序作品

**警告**：不要直接操纵任何文件或目录 /var/lib/docker/。这些文件和目录由Docker管理。

lsblk从操作系统的角度来看，使用命令查看设备及其池：

$ sudo lsblk

NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT

xvda 202:0 0 8G 0 disk

└─xvda1 202:1 0 8G 0 part /

xvdf 202:80 0 100G 0 disk

├─docker-thinpool\_tmeta 253:0 0 1020M 0 lvm

│ └─docker-thinpool 253:2 0 95G 0 lvm

└─docker-thinpool\_tdata 253:1 0 95G 0 lvm

└─docker-thinpool 253:2 0 95G 0 lvm

使用该mount命令查看Docker正在使用的挂接点：

$ mount |grep devicemapper

/dev/xvda1 on /var/lib/docker/devicemapper type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

使用时devicemapper，Docker将图像和图层内容存储在稀释池中，并将它们放在容器的子目录下/var/lib/docker/devicemapper/。

#### 磁盘上的图像和容器图层

该/var/lib/docker/devicemapper/metadata/目录包含有关Devicemapper配置本身以及存在的每个映像和容器层的元数据。该devicemapper存储驱动程序使用快照，这个元数据包含有关这些快照的信息。这些文件是JSON格式的。

该/var/lib/devicemapper/mnt/目录包含存在的每个图像和容器层的安装点。图像层安装点为空，但容器的安装点显示容器在容器内出现的文件系统。

#### 图像分层和共享

该devicemapper存储驱动程序使用专用的块设备，而不是格式化的文件系统，并在写入时复制（全体）操作上以获得最大性能的块级的文件进行操作。

##### 快照

另一个特点devicemapper是它使用快照（有时也称为 瘦设备或虚拟设备），它将每个层中引入的差异存储为非常小的轻量级精简池。快照提供了许多好处：

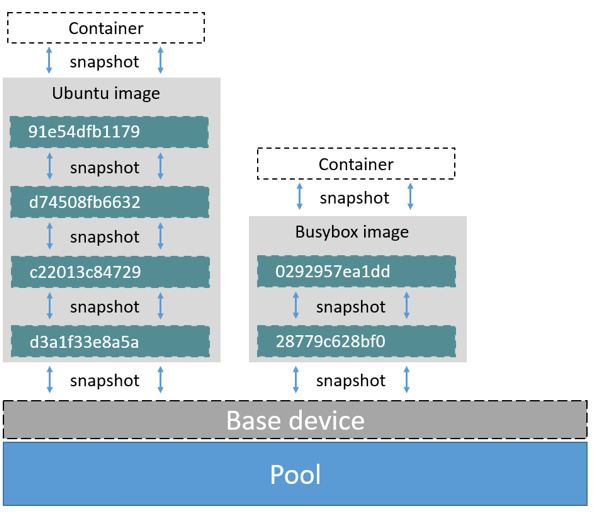
* 在容器之间共享的层只存储在磁盘上一次，除非它们是可写的。例如，如果您有10种不同的图像，这些图像都是基于的alpine，则alpine图像及其所有父图像仅在磁盘上存储一次。
* 快照是写时复制（CoW）策略的一个暗示。这意味着给定的文件或目录仅在容器的可写层被该容器修改或删除时复制。
* 因为devicemapper在块级别操作，可以同时修改可写层中的多个块。
* 可以使用标准操作系统级备份实用程序来备份快照。只是复制一份/var/lib/docker/devicemapper/。

##### DEVICEMAPPER工作流程

当您使用devicemapper存储驱动程序启动Docker时，与映像和容器层相关的所有对象都将存储在/var/lib/docker/devicemapper/一个或多个块级设备中，即环回设备（仅测试）或物理磁盘。

* 所述基部设备是最低级的对象。这就是瘦池本身。你可以使用它来检查它docker info。它包含一个文件系统。该基本设备是每个图像和容器层的起点。基本设备是Device Mapper实现细节，而不是Docker层。
* 有关基本设备和每个图像或容器层的元数据/var/lib/docker/devicemapper/metadata/以JSON格式存储 。这些层是写时复制快照，这意味着它们是空的，直到它们与父层分开。
* 每个容器的可写层都安装在安装点上 /var/lib/docker/devicemapper/mnt/。每个只读图像层和每个停止的容器都存在空目录。

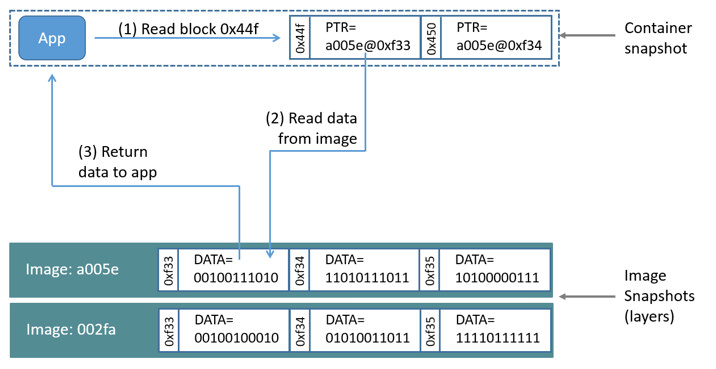
每个图像层是其下面的图层的快照。每个图像的最低层是存储在池中的基本设备的快照。运行容器时，它是容器所基于的映像的快照。以下示例显示具有两个运行容器的Docker主机。第一个是ubuntu 容器，第二个是busybox容器。



### 容器读写操作如何工作 devicemapper

#### 阅读文件

随着devicemapper读取在块级发生。下图显示了0x44f在示例容器中读取单个块（）的高级过程。



应用程序0x44f在容器中对块进行读取请求。因为容器是一个图像的薄片快照，它没有块，而是有一个指向最近的父图像中存在的块的指针，并且从那里读取该块。该块现在存在于容器的内存中。

#### 写文件

**编写新文件**：使用devicemapper驱动程序，通过按需分配操作完成向容器写入新数据。新文件的每个块都分配在容器的可写层中，并将块写入。

**更新现有文件**：从存在的最近层读取**文件**的相关块。当容器写入文件时，只有修改的块被写入容器的可写层。

**删除文件或目录**：当您删除容器可写层中的文件或目录时，或者当图像层删除其父层中存在的文件时，devicemapper存储驱动程序会拦截对该文件或目录的进一步读取尝试，并响应文件或目录不存在。

**写入然后删除文件**：如果容器写入文件并稍后删除文件，则所有这些操作都发生在容器的可写层中。在这种情况下，如果您正在使用direct-lvm，这些块将被释放。如果使用loop-lvm，则块可能不会被释放。这是loop-lvm生产中不使用的另一个原因 。

### 设备映射器和Docker性能

* **allocate-on demand性能影响**：

在devicemapper存储驱动程序使用的allocate-on-demand操作以从薄池新块分配到在容器的可写层。每个块是64KB，所以这是写入的最小空间量。

* **复制性能影响**：容器首次修改特定块时，将该块写入容器的可写层。因为这些写入发生在块而不是文件的级别，所以性能影响最小化。然而，编写大量块仍然会对性能产生负面影响，而且devicemapper在这种情况下，存储驱动程序实际上可能会比其他存储驱动程序恶化。对于写入繁重的工作负载，您应该使用数据卷，完全绕过存储驱动程序。

#### 性能最佳实践

记住这些事情，以便在使用devicemapper 存储驱动程序时最大限度地提高性能。

* **使用direct-lvm**：loop-lvm模式不是性能，不应该在生产中使用。
* **使用快速存储**：固态驱动器（SSD）比旋转磁盘提供更快的读取和写入。
* **内存使用情况**：devicemapper使用比其他一些存储驱动程序更多的内存。每个启动的容器将其文件的一个或多个副本加载到内存中，具体取决于同一文件的多少个块同时被修改。由于内存压力，devicemapper存储驱动程序可能不是高密度用例中某些工作负载的正确选择。
* **为繁重的工作负载使用卷**：卷**为重写繁重的工作负载**提供最佳和最可预测的性能。这是因为它们绕过存储驱动程序，并且不会产生由精简配置和写时复制引入的任何潜在开销。卷具有其他好处，例如允许您在容器之间共享数据，即使没有运行容器正在使用它们也会持续存在。

## ## 配置镜像加速器：

首先去阿里云加速器、DaoCloud 加速器、灵雀云加速器 申请加速地址《如 https://jxus37ad.mirror.aliyuncs.com》

### 4、 配置镜像加速器给docker引擎：

#### Ubuntu 14.04、Debian 7 Wheezy （对于使用 upstart 的系统而言）：

1、编辑 /etc/default/docker ，在DOCKER\_OPTS 中添加 : --registry-mirror=<加速器地址>.如：DOCKER\_OPTS="--registry-mirror=https://jxus37ad.mirror.aliyuncs.com"

2、再重启服务：sudo service docker restart

#### Ubuntu 16.04、Debian 8 Jessie、CentOS 7（对于使用 systemd 的系统）：

1、用 systemctl enable docker 启用服务

2、编辑 /etc/systemd/system/multi-user.target.wants/docker.service 文件(是/usr/lib/systemd/system/docker.service 文件的软连接)

找到 ExecStart= 这一行，在这行最后添加加速器地址 --registry-mirror=<加速器地址>，如：ExecStart=/usr/bin/dockerd --registry-mirror=https://jxus37ad.mirror.aliyuncs.com

或者：sudo tee -a /etc/docker/daemon.json <<-EOF

{

"registry-mirrors": [

"加速地址"

],

"insecure-registries": []

}

EOF

3、重新加载配置并且重新启动。

$ sudo systemctl daemon-reload

$ sudo systemctl restart docker

#### Windows 10：

对于使用 WINDOWS 10 的系统，在系统右下角托盘图标内右键菜单选择 Settings，打开配置窗口后左侧导航菜单选择 Docker Daemon。编辑窗口内的JSON串，填写如阿里云、DaoCloud之类的加速器地址，如：

{

"registry-mirrors": [

"https://sr5arhkn.mirror.aliyuncs.com",

"http://14d216f4.m.daocloud.io"

],

"insecure-registries": []

}

编辑完成，点击Apply保存后Docker服务会重新启动

检查配置是否成功：

Linux系统下配置完加速器需要检查是否生效，在命令行执行 ps -ef | grep dockerd，如果从结果中看到了配置的 --registry-mirror 参数说明配置成功。

$ sudo ps -ef | grep dockerd

root 5346 1 0 19:03 ? 00:00:00 /usr/bin/dockerd --registry-mirror=https://jxus37ad.mirror.aliyuncs.com

$

#### Windows10 以下版本:

docker-machine ssh default ; # 登陆到虚拟机的docker机器上

sudo -i ; # 切换到root的用户上

sudo sed -i "s|EXTRA\_ARGS='|EXTRA\_ARGS='--registry-mirror=加速地址 |g" /var/lib/boot2docker/profile;

exit

docker-machine restart default # 重启

## 获取镜像：

命令：docker pull 。

docker pull [选项] [Docker Registry地址]<仓库名>:<标签>

## 列出镜像：

docker images

## 从镜像中创建一个容器并进入容器：

docker run -it ubuntu #ubuntu 是镜像名称

-i 交互式

-t 命令tty方式

-d 后台运行

--name 容器名字

--rm 运行从容器交互式命令行退出删除容器

## 悬挂镜像：

在使用docker images是有时会出现既没有仓库名，也没有标签，均为<none> ---悬挂镜像。

其实这个镜像是有镜像名和标签,原来为mongo:3.2 ，只是后来官方发布新版本后，从新docker pull mongo:3.2时，mongo:3.2这个名称被转移到新下载的镜像身上。出来docker pull 命令docker build也会导致这种情况。

docker images -f dangling=true 此命令可以专门显示悬挂镜像。

悬挂镜像已经失去价值可以随意删除：docker rmi $(docker images -q -f dangling=true)

中间层镜像(被顶层镜像依赖，不应该删除。顶层镜像删除了中间层镜像自然消失了)：

为了加速镜像构建、重复利用资源，docker 会利用中间层镜像。所以在一段时间后会出现一些一些依赖的中间层镜像。docker images 命令显示的只是顶层镜像。

显示包括中间层镜像在内的命令：docker images -a

## 过滤镜像：

docker images ubuntu

docker images ubuntu:3.2

docker images -f since=ubuntu

docker images -f before=ubuntu

## 主机与docker容器映射端口：

docker run --name webserver -d -p 90:80 nginx

-p 是用于映射容器端口 # 90 主机端口 80 container端口

## 对比容器相比镜像更改了什么：

docker diff [container名称]

将container的更改保存为image

docker commit [选项] <容器ID或容器名> [<仓库名>:[<标签>]]

docker commit --author "author name" --message "修改了默认网页" webserver nginx:v2

--author 作者

--message 附加信息

webserver 要保存为镜像的container

## 查询镜像的修改历史记录：

dicker history nginx

nginx 要查看的镜像名称或id

*tipe：* ***慎用docker commit命令***，因为用命令docker diff webserver(container) 查看修改了什么时，我们知道我们只修改了一个文件，但是事实却是除了我们修改之外还有由于命令的执行被改的的文件，会导致新的image特别臃肿。

用docker commit 制作的镜像成为黑箱镜像，应为除了制作者本人谁也不知道改了什么，过一段时间作者也会忘记。虽然docker diff 可以告诉你一些线索，但远远不能生成一致的镜像。docker commit 可以用于比如机器被入侵保留现场。万万不可用于定制镜像

## 使用Dockerfile定制镜像：

### 先来个小总结：

#### 以非交互模式安装软件

在我们使用linux安装软件时，常常会弹出文本框让我们选择，但是在docker通过Dockerfile生成镜像时，它是非交互shell，可能会出现安装失败，要在运行Run命令时指定 DEBIAN\_FRONTEND=noninteractive，它明确告诉了linux的程序安装器当前是非交互式shell

RUN DEBIAN\_FRONTEND=noninteractive apt-get install -y apt-utils bash-completion vim cmake gcc g++ gdb git curl libcurl4-gnutls-dev unzip build-essential automake libtool python

#### 缓存

当我们在通过docker构建程序时，如果是从svn或者git服务器上拉取代码再编译，那么由于命令与上次相同，就会出现不更新代码的问题，为了防止这种情况，可以通过docker build –no-cache来编译镜像，但是这样所以文件都不会使用缓存，生成时间太长，我们可以通过在拉取代码时运行不同的命令达到效果：

RUN DEBIAN\_FRONTEND=noninteractive echo '2015121600' > /dev/null && git clone --depth=1 --recursive https://github.com/sails/saf.git

每次只用改变echo的内容即可；

#### 日期

docker容器的时间默认是美国时间，在我们的程序中如果不注意就会出问题，可以在创建的image的时间指定：

RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

#### coredump

为了生成coredump,可以在启动容器时指定–ulimit core=xx，不要用unlimited，要指定具体的值。

#### 生成coredump

docker run有一个–ulimit选项，可以指定资源的限制，如当我们要生成core文件时：

docker run --ulimit core=1000000000000 test /test

#### 使用coredump

当容器在正式环境中运行时，都是以后台-d选项启动的，那么这时如果生成了coredump，容器已经关闭了，那怎么调试呢？我们可以把coredump文件通过卷的方式放到host中，再另外启动一个容器，因为环境，服务都一样，所以这个coredump也适用于其它相同docker image生成的容器，所以在新的容器中调度就ok了:

sudo bash -c "echo /core/%e.core.%p > /proc/sys/kernel/core\_pattern"

docker run --ulimit core=1000000000000 -v /home/sails/core:/core test /test

#### 同时执行多个命令

在linux下，命令都是在当前shell中执行的，所以上次的更改会作用到下一次，常常会这样cd到目录，然后再编译程序，但是docker如果把多个命令分到不同的RUN指令中，不会成功，因为它的一个RUN就会打开一个shell进行，为了达到目的，可以把多个命令放在一起执行，但是这里要注意’&&’:

RUN DEBIAN\_FRONTEND=noninteractive cd saf && mkdir build && cd build && cmake ../ && make

***tiep:***

解决时区错误，在Dockerfile 中添加命令：

RUN cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

Dockerfile文件记录的每-条指令对应构建一层存储

一个基于nginx镜像构建镜像：

定制Dockerfile：

mkdir MyNginx && cd MyNginx

vi Dockerfile

FROM nginx ##基于nginx镜像

RUN echo '<h1>hello,docker!!!!!!</h1>' > /usr/share/nginx/html/index.html.index ##运行命令

FROM 指令是必须指令，并且用于Dockerfile文件的第一行

Docker还存在一个特殊镜像，名为scratch,这个镜像是虚拟的概念，并不实际存在，表示空白的镜像。如果其他镜像无法满足你需要的基础镜像你可以用FROM scratch 为基础定制

以scratch 为基础的镜像更小巧更适合golang使用

RUN 命令是执行命令的命令，其格式有两种：

shell 格式：RUN <命令>

exec 格式：RUN ["可执行文件","参数1","参数2"]

tipe: 由于Dockerfile 没执行一个命令会建立一层，所以如果命令能写成一条劲量一条来RUN。这样不仅减少了构建部署的时间，也不容易出错（Union FS 是有最大层数限制的）

Dockerfile 中鼓励是有 \ 作为命令换行方式，以及行首 # 进行注释。

***tipe：***重要的一点是 在一组命令的最后添加清理工作，删除为了编译构建所需要的软件，所下载的、展开的文件、apt缓存文件。因为每一层的东西并不会在下一层删除，会一直跟着image

### Dockerfile定制完成了，开始构建镜像：

docker build -t nginx:v3 .

# . 是指定上下文路径，一切都基于这个路径（Dockerfile除外），没有指定Dockerfile文件时，默认去上下文的路径下找。包括要是上传到镜像中的东西也要放到上下文中，build时会把上下文中的数据打包发送到服务端Docker引擎构建image。

docker build 还支持从URL构建：

docker build https://github.com/twang2218/gitlab-ce-zh.git\#:8.14

此命令构建目录为 /8.14/

用给定的tar压缩包构建：

docker build http://server/context.tar.gz

如果给的URL不是git repo，而是个tar包，Docke会下载解压并以其作为上下文构建

从标准输出中读取Dockerfile构建：

docker build - < Dockerfile 或者 cat Dockerfile | docker build -

如果Dockerfile是文本这种没有上下文，不能像其他方法将本地文件COPY进镜像；如果是Dockerfile压缩包,会解压并以其为上下文

### Dockerfile中的命令：

（除了FROM RUN EXEC）

#### COPY:

两种格式：

shell方式：COPY <源路径>...<目标路径>

exec方式： COPY ["<源路径1>",..."<目标路径>"]

原路径 可以有多个，也可以是符合GO的filepath.Match规则的通配符。

目标路径 可以是容器内的绝对路径，也可以是相对工作目录的相对路径（工作目录可以用WORKDIR指令来指定），$

COPY命令会让原路径的文件文件夹的权限一同拷贝

COPY有两种形式：

* COPY <src>... <dest>
* COPY ["<src>",... "<dest>"] （此窗体是包含空格的路径所必需的）

该COPY指令将复制新文件或目录<src> ，并将其添加到路径中容器的文件系统<dest>。

<src>可以指定多个资源，但它们必须相对于正在构建的源目录（构建的上下文）。

每个<src>可能包含通配符，并使用Go的[filepath.Match](http://golang.org/pkg/path/filepath#Match)规则进行匹配 。例如：

COPY hom\* /mydir/ # adds all files starting with "hom"

COPY hom?.txt /mydir/ # ? is replaced with any single character, e.g., "home.txt"

的<dest>是一个绝对路径，或相对于一个路径WORKDIR，到其中的源将在目标容器内进行复制。

COPY test relativeDir/ # adds "test" to `WORKDIR`/relativeDir/

COPY test /absoluteDir/ # adds "test" to /absoluteDir/

当复制包含特殊字符（如[ 和]）的文件或目录时，需要按照Golang规则转义这些路径，以防止它们被视为匹配模式。例如，要复制一个名为的文件arr[0].txt，请使用以下命令：

COPY arr[[]0].txt /mydir/ # copy a file named "arr[0].txt" to /mydir/

所有新的文件和目录都使用UID和GID为0创建。

**注意**：如果使用STDIN（docker build - < somefile）构建，则没有构建上下文，因此COPY不能使用。

可选地，COPY接受一个--from=<name|index>可用于将源位置设置为FROM .. AS <name>将被使用而不是由用户发送的构建上下文的创建阶段（创建）的标志。该标志还接受以FROM指令开始的所有以前构建阶段分配的数字索引 。如果无法找到具有指定名称的构建阶段，则尝试使用具有相同名称的映像。

COPY 遵守以下规则：

* 该<src>路径必须是内部*语境*的构建; 你不能COPY ../something /something，因为第一步 docker build是将上下文目录（和子目录）发送到docker守护程序。
* 如果<src>是目录，则会复制目录的全部内容，包括文件系统元数据。

**注意**：目录本身不被复制，只是其内容。

* 如果<src>是任何其他类型的文件，它将与其元数据一起单独复制。在这种情况下，如果<dest>以尾部斜杠结尾/，它将被视为目录，并将内容<src>写入<dest>/base(<src>)。
* 如果<src>指定了多个资源，直接或由于使用通配符，则<dest>必须是目录，并且必须以斜杠结尾/。
* 如果<dest>不以尾部斜杠结尾，则将被视为常规文件，并将内容<src>写入<dest>。
* 如果<dest>不存在，则在其路径中创建所有缺少的目录

#### ADD更高级的复制文件：

和COPY性质和格式基本一致。

在COPY指令上增加了功能：源文件可以是一个URL,由URL下载下来的文件权限设置为600，如果源文件是压缩包，会自动解压。

ADD会使镜像构建缓存失效，使构建速度缓慢。

tiep：不建议使用ADD.仅在需要解压缩是使用。

#### CMD 两种格式：

shell 方式：CMD <命令>

exec 方式（推荐使用）：CMD ["可执行文件","参数1","参数2"...]

CMD用于指定默认容器主进程的启动命令。在运行是可以指定新的命令来代替镜像设置中的默认命令

eg:docker run -it ubuntu #会直接进入bash

eg：docker run -it ubuntu cat /etc/os-release #替换默认的/bin/bash 命令输出文件内容啦

***tipe：***容器中的进程没有后台执行之说，只有前台。

#### ENTRYPOINT:入口点：

shell 方式：

exec 方式（推荐使用）：

ENTRYPOINT 和 CMD一样，都是在指定容器启动及参数。

ENTRYPOINT 在运行时也可替换，需要通过docker run 的参数 --entryport来指定。

当指定ENTYYPORT 后，CMD的含义发生变化，不再直接运行其命令，而是建CMD的内容作为参数传给 ENTYRPORT 的指令：<ENTRYPORT> "<CMD>"。

场景:

1、让镜像变成像命令一样使用。

Dockerfile:

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

CMD ["curl","-s","http://ip.cn"]

构建：docker build -t myip

运行：docker run myip # 会打印出响应的内容：当前ip:.......

我们如果想显示请求头信息呢？

这样运行？：docker run myip -i #不对，我们说过了，放在命令后的参数哦会作为命令替换CMD的默认命令。那么CMD就无法是参数灵活使用。

ENTRYPOINT 可以！

Dockerfile:

FROM ubuntu:16.04

RUN apt-get update \

&& apt-get install -y curl \

&& rm -rf /var/lib/apt/lists/\*

ENTRYPOINT ["curl","-s","http://ip.cn"]

构建：docker build -t myip

我们可以直接当做命令运行：

docker run myip #显示响应内容

我想知道请求头信息：docker run myip -i #此处的 i 就替换了原始命令中的 s

2、应用运行前的准备工作：

#### ENV 设置环境变量：

两种格式：

ENV <key> <value>

ENV <key1>=<value1> <key2>=<value2>...

tiep:有空格的值可以用""包裹

#### ARG 构建参数：

ARG <参数名>[=<默认值>]

和 ENV 的效果是一样，都是设置环境变量。不同事 ARG 所设置的构建环境变量将来容器运行时是不会存在这些环境变量的。但不要因此就是用ARG 保存密码之类信息，应为docker history 还是可以看到的。

ARG 是定义参数名称，及默认值，该默认值可以在构建命令docker build 中用 --build-arg <参数名>=<值>来覆盖。

#### VOLUME 定义匿名卷：

格式:

VOLUME ["<路径1>","<路径2>"...]

VOLUME <路径>

为防止运行时用户忘记将动态文件所保存目录挂载为卷，在Dockerfile中，我们可以事先指定某些目录挂载为匿名卷。

VOLUME /data 这里的/data 目录就会在运行自动挂载为匿名卷，任何向/data 中写信息都不会记录进容器存储存，从而保证容器存储存的无状态话。

运行时可以覆盖这个挂载设置：docker run -d -v mydata:/data xxxx 这就是用了mydata这个命名卷挂载到了 /data 这个位置。

#### EXPOSE 暴露端口：

格式：

EXPOSE <端口1>[<端口2>...]

***tipe：***1、这只是一个声明，使是用者理解镜像服务的守护端口，方便配置映射

2、在运行时是用随机端口映射时：docker run -P，会自动随机映射EXPOST的端口

3、在早前还有特殊用途：早前所有容器运行于默认桥接网络中存在所有容器之间可以相互直接访问安全隐患，于是有了Docker 引擎参数 --icc=false使容器间默认无法访问

除非使用了--links参数的容器才可以互通，并且只有镜像中EXPOST所声明的端口才可以被访问。--icc=false 在引用docker network后基本不用了。通过自定义网络可以轻松实现容器间的互联和隔离。

4、要将EXPOSE 和 运行时使用 -p <宿主端口>:<容器端口> 区分开来。-p是映射主机和容器端口。EXPOSE 仅仅是声明容器打算使用什么端口，并不会自动在宿主机和容器间进行端口映射。

#### WORKDIR 指定工作目录：

格式：

WORKDIR <工作目录路径>

使用WORKDIR 指令可以指定工作目录或者称为当前目录，如果该目录不存在会帮你创建

***tiep:***关于把Dockerfile作为Shell来执行在此在此声明：

如：RUN cd /app

RUN echo "hello" > world.txt

用上面这个dockerfile构建镜像会发现当WORKDIR不是/app 时 /app/world.txt 文件找不到。原因之前说过，在Dockerfile中两个RUN命令的执行环境根本不同，而是两个完全不同的容器。这就是在Dockerfile构建分层存储导致，之前说过，每一个RUN都是启动一个容器、执行命令、然后提交存储层。

因此如果需要改变以后各层的工作目录的位置，那么应该使用WORKDIR执行

WORKDIR /app

RUN echo "hello" > world.txt

#### USER 用于指定当前用户：

格式：

USER <用户名>

USER 和 WORKDIR相似,都是改变环境状态并影响以后的层.

不过这个用户名是事先创建好的.

希望以某个已建立好的用户来运行某个服务进程,不要使用su或者sudo,这些都需要比较麻烦的配置，而且在TTy缺失的环境下经常出错。建议使用gosu

见《docker\_practice》 P91:



#### HEALTHCHECK 健康检查：

格式：

HEALTHCHECK [选项] CMD <命令>:设置检查容器健康状态的命令

HEALTHCHECK NONE :如果基础镜像用健康检查指令，此命令可以屏蔽。

HEALTHCHECK 支持选项：

--interval=<间隔>：两次健康检查的间隔，默认30秒

--timeout=<时长> ：健康检查命令运行超时时间，默认30秒。

--retries=<次数>：当前连续失败指定次数后，则容器状态视为unhealthy，默认3次

如果此命令写了多个，最后一个有效 见《docker\_practice》P93



#### ONBUILD 为他人作嫁衣

ONBUILD <其它指令>

在ONBUILD后的命令在执行当前镜像是不会执行，而是在以此镜像为基础的镜像才会执行。

## 其他制作镜像的方式：

### 1、从rootfs压缩包导入：

格式: docker import [选项] <文件> | <URL> |- [<仓库名>[:<标签>]]

压缩包可以使本地文件、远程文件，甚至从标准输入中得到。压缩包将会在镜像目录展开，并直接作为镜像的第一层提交。

eg:docker import http://download.openvz.org/template/precreated/ubuntu-14.04-x86\_64-minimal.tar.gz \

openvz/ubuntu:14.04

### 2、docker load 和 docker save 命令：

用于加载和保存为tar镜像。在没有docker Registry 的情况下。

保存镜像为归档文件：

docker save apline | gzip > apline-latest.tar.gzip

加载镜像：

docker load -i aplin-latest.tar.gz

一个命令完成从一个机器将镜像迁移到另一个机器：

docker save <镜像名> | bzip2 | pv |ssh <用户名>@<主机名> 'cat | docker load'

## 删除本地镜像：

docker rmi [选项] <镜像1> [<镜像2>...]

tiep: docker rm 是删除容器

***tiep:*** 对于CentOs/RHEL 的用户需要注意：这类系统没有*UnionFS*[aufs、overlay2] 使用，一般使用devicemapper 驱动利用LVM的机制模拟分层存储。

这样会使性能和稳定性大打折扣，而且配置相对复杂。默认devicemapper 为了简化配置，其devicemapper是跑在一个稀疏文件模拟的块设备上，也被称为loop-lvm(这样不需要格外配置就可以运行).

loop-lvm 的做法是稳定性和性能大打折扣，无论日志还是docker info 中都会看到警告信息，而且/var/lib/docker/devicemapper/devicemapper/data 由于是稀疏文件，导致不段增长无法控制，甚至很多人为解决这个问题删除镜像，空间释放后基本不进行垃圾回收，也是见效甚微，不见文件减小。

对如何配置块设备给devicemapper驱动做存储层的做法官方有提供，这类做法也被称为配置direct-lvm。

或许有人注意到CentOs7中存在被backports 回来的overlay驱动，不过CentOs里的这个驱动到不到生产环境使用稳定程度，所以不推荐使用。

## 启动容器：

### 基于镜像新建并启动；

主要命令：docker run

eg1：输出"Hello World"之后终止容器：

sudo docker run ubuntu:14.04 /bin/echo 'Hello World'

eg2:启动一个终端，允许用户进行交互：

sudo docker run -it unbuntu:14.04 /bin/bash

-t 选项让Docker 分配一个伪终端并绑定到容器的标准输入上，

-i让容器标准输入保持打开。

-d 守护态运行（后台运行）需要回去守护态的输出可以用docker logs命令查看

### 将终止状态的容器重新启动：

主要命令docker start

docker restart 将一个容器终止后启动。

## 终止容器：

主要命了：docker stop

此外当Docke 容器中指定的应用（主进程）终结是，容器自动终止。

终止的容器可以用docker ps -a来查看

## 进入容器：

在使用-d参数时，容器启动后会进入后台运行。如果需要进入容器进行操作有很多方法，包括使用docker attach 命令或者nsenter 工具等。

attach 命令：

docker attach <镜像>

缺点：当多个窗口attach到同一个容器时所有窗口会同步显示，当一个窗口阻塞是，其他窗口也不能用。退出是不可以用Ctrl+c 或者 exit 会使容器进入stop状态。要用

exec 命令：

docker exec <容器名/ID> /bin/sh

nsenter 命令：

想用需要安装，自己查文档吧。

## 查看容器信息：

docker stat containerId #容器的内存，cpu，网络io信息

docker inspect containerId #容器的配置信息，这个信息更大而全

## 导出和导入容器：

导出容器：

docker export <容器id/容器名> > ubuntu.tar # 导出容器快照到本地文件

导入容器快照：

docker import

*eg:* cat ununtu.tar | sudo docker import - test/unbuntu:v1.0

用户可以使用docker load 来导入镜像存储文件到本地镜像库，也可以docker import 导入一个容器快照到本地镜像库。区别在于容器快照文件将丢弃所以历史记录和元数据信息（既仅保存容器当时的快照状态），但可以重新制定标签等元数据信息

而镜像存储文件将保存完整记录，体积也要大。

## 删除容器：

docker rm 删除一个处于终止状态的容器。

如果删除处于运行中的容器可以添加 -f 参数。

docker rm $(docker ps -a -q) 可以全部删除处于stop状态的容器

docker search ## 搜索镜像

docker pull ## 将镜像拉取到本地

docker push ## 将镜像推送到Docker Hub

## Docker Hub 中的镜像：

docker search <image name> #搜索镜像时镜像名（NAME）有两种，**一种是单个单词** 如：centos;**另一种如：tianon/centos。**

**第一种**是基础镜像也被称为根镜像；是有Docker公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。

**第二种**是有Docker用户创建并维护，往往带有用户名为前缀，可以通过前缀 user\_name/ 来指定使用某个用户提供的镜像。

## 自动创建：

有时候用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像，麻烦！。

而自动创建允许用户通过 DockerHub 指定跟踪一个目标网站（目前支持 GitHub 或 BitBucket）上的项目，一旦项目发生新的提交，则自动执行创建。

配置自动创建：

创建并登陆 Docker Hub 以及目标网站；

在目标网站中连接账户到 Docker Hub；

在 Docker Hub 中 配置一个自动创建；

选取一个目标网站中的项目（需要韩Dockerfile） 和分支；

指定 Dockerfile 的位置，并提交创建。

之后可以在 Docker Hub 的自动创建页面中的跟踪每次创建的状态。

## 私有仓库：

docker-registry 是官方提供的工具，用于构建私有的镜像仓库。

### 安装：

#### 容器运行：

前提：安装了Docker 后，可以通过获取官方registry 镜像来运行。

$ docker run -d -p 5000:5000 registry

用户也可以指定参数来配置私有仓库位置，例如配置镜像存储到Amazon S3服务。

$ docker run \

-e SERRINGS\_FLAVOR=s3 \

-e AWS\_BUCKET=acme-docker \

-e STORAGE\_PATH=/registry \

-e AWS\_KEY=AKIAHSHB43HS3J92MXZ \

-e AWS\_SECRET=xdDowwlK7TJajV1Y7EoOZrmuPEJlHYcNP2k4j49T \

-e SEARCH\_BACKEND=sqlalchemy \

-p 5000:5000 \

registry

此外还可以指定本地路径（如 /home/user/registry-conf）下的配置文件。

$ docker run -d -p 5000:5000 -v /home/user/registry-conf:/registry-conf -e DOCKER\_REGISTRY\_CONFIG=/registry-conf/config.yml registry

默认情况下，仓库会被创建在容器 /var/lib/registry(v1 中是/tmp/registry)下。可以通过 -v 参数来讲镜像文件存放在本地指定路径。

$ docker run -d -p 5000:5000 -v /opt/data/registry:/var/lib/registry registry

#### 本地安装：

##### Ubuntu：

sudo apt-get install -y build-essential python-dev libevent-dev python-pip liblzma-dev

sudo pip install docker-registry

##### ContOs:

sudo yum install -y gcc xz-devel python-devel libevent-devel python-pip

sudo python-pip install docker-registry

也可以源码安装(不累述)

配置：

主要修改dev 模板段的storage-path 到本地的存储仓库路径。

cp config/config\_sample.yml config/config.yml ## 示例配置文件内容

之后启动Web服务：

sudo gunicorn --access-logfile - --error-logfile - -k gevent -b 0.0.0.0:5000 -w 4 --max-requests 100 docker\_registry.wsgi:application

## Portainer 项目：

***docker 管理平台***

### 安装：

1. **生产加密密码：<123456789 是我们的密码>**

***$ docker run --rm httpd:2.4-alpine htpasswd -nbB admin 123456789 | cut -d ":" -f 2***

生产的密码加密串为：

'$2y$05$WHEnya02H8UNZp6/nSUtRe.ECPlqDqhy2BeMTUT2P3UO5u.GriW/u'

1. **容器安装：**

***$ docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock portainer/portainer --admin-password '$2y$05$WHEnya02H8UNZp6/nSUtRe.ECPlqDqhy2BeMTUT2P3UO5u.GriW/u'***

1. **浏览器动了本地9000端口进入管理界面**

## Docker Compose 项目

# *负责快速在集群中部署分布式应用*

Compose允许用户通过一个单独的 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。（比如实现一个 Web 项目：除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等）

Compose 中有两个重要的概念：

* **服务（service）**：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
* **项目(project)**：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

### 安装：

***tipe:*** Compose 依赖Docker Engine 1.7.1+.

***tipe:*** 三种安装方式：

1. pip安装（Compose是python项目）
2. 下载编译好的二进制文件使用。
3. 直接运行在Docker容器中。

#### pip安装：

$ sudo pip install -U docker-compose

（安装后可以查看docker-compose 的用法：$ docker-compose –h）

之后，可以添加 bash 补全命令：

$ curl -L https://raw.githubusercontent.com/docker/compose/1.8.0/contrib/completion/bash/docker-compose > /etc/bash\_completion.d/docker-compose

#### 二进制包：

二进制文件，下载后直接放到执行路径下，并添加执行权限即可。

$ sudo curl -L https://github.com/docker/compose/releases/download/1.8.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

$ sudo chmod a+x /usr/local/bin/docker-compose

#### 容器中执行：（环境无关性）

Compose 既然是一个 Python 应用，自然也可以直接用容器来执行它。

$ curl -L https://github.com/docker/compose/releases/download/1.8.0/run.sh > /usr/local/bin/docker-compose

$ chmod +x /usr/local/bin/docker-compose

run.sh 中其实是下载docker-compose 容器运行。

### 卸载：

如果是二进制包方式安装的，删除二进制文件即可。

$ sudo rm /usr/local/bin/docker-compose

如果是通过 python pip 工具安装的，则可以执行如下命令删除。

$ sudo pip uninstall docker-compose

### 术语：

* 服务（service）：一个应用容器，实际上可以运行多个相同镜像的实例。
* 项目(project)：由一组关联的应用容器组成的一个完整业务单元。

可见，一个项目可以由多个服务（容器）关联而成，Compose 面向项目进行管理。

#### example：

##### 场景：

一个web项目：一个Haproxy,挂载三个web容器。

###### 目录结构：

现在 compose-haproxy-web 目录是工作目录。

compose-haproxy-web

├── docker-compose.yml

├── haproxy

│ └── haproxy.cfg

└── web

├── Dockerfile

├── index.html

└── index.py

web目录下是项目和Dockerfile文件

haproxy目录下是haproxy的配置文件，用于挂载到Docker下的卷。

docker-compose.yml 是Docker-compose的配置文件。

***index.py***作为服务器文件，代码为:

#!/usr/bin/python

#authors: yeasy.github.com

#date: 2013-07-05

import sys

import BaseHTTPServer

from SimpleHTTPServer import SimpleHTTPRequestHandler

import socket

import fcntl

import struct

import pickle

from datetime import datetime

from collections import OrderedDict

class HandlerClass(SimpleHTTPRequestHandler):

def get\_ip\_address(self,ifname):

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

return socket.inet\_ntoa(fcntl.ioctl(

s.fileno(),

0x8915, # SIOCGIFADDR

struct.pack('256s', ifname[:15])

)[20:24])

def log\_message(self, format, \*args):

if len(args) < 3 or "200" not in args[1]:

return

try:

request = pickle.load(open("pickle\_data.txt","r"))

except:

request=OrderedDict()

time\_now = datetime.now()

ts = time\_now.strftime('%Y-%m-%d %H:%M:%S')

server = self.get\_ip\_address('eth0')

host=self.address\_string()

addr\_pair = (host,server)

if addr\_pair not in request:

request[addr\_pair]=[1,ts]

else:

num = request[addr\_pair][0]+1

del request[addr\_pair]

request[addr\_pair]=[num,ts]

file=open("index.html", "w")

file.write("<!DOCTYPE html> <html> <body><center><h1><font color=\"blue\" face=\"Georgia, Arial\" size=8><em>HA</em></font> Webpage Visit Results</h1></center>");

for pair in request:

if pair[0] == host:

guest = "LOCAL: "+pair[0]

else:

guest = pair[0]

if (time\_now-datetime.strptime(request[pair][1],'%Y-%m-%d %H:%M:%S')).seconds < 3:

file.write("<p style=\"font-size:150%\" >#"+ str(request[pair][1]) +": <font color=\"red\">"+str(request[pair][0])+ "</font> requests " + "from &lt<font color=\"blue\">"+guest+"</font>&gt to WebServer &lt<font color=\"blue\">"+pair[1]+"</font>&gt</p>")

else:

file.write("<p style=\"font-size:150%\" >#"+ str(request[pair][1]) +": <font color=\"maroon\">"+str(request[pair][0])+ "</font> requests " + "from &lt<font color=\"navy\">"+guest+"</font>&gt to WebServer &lt<font color=\"navy\">"+pair[1]+"</font>&gt</p>")

file.write("</body> </html>");

file.close()

pickle.dump(request,open("pickle\_data.txt","w"))

if \_\_name\_\_ == '\_\_main\_\_':

try:

ServerClass = BaseHTTPServer.HTTPServer

Protocol = "HTTP/1.0"

addr = len(sys.argv) < 2 and "0.0.0.0" or sys.argv[1]

port = len(sys.argv) < 3 and 80 or int(sys.argv[2])

HandlerClass.protocol\_version = Protocol

httpd = ServerClass((addr, port), HandlerClass)

sa = httpd.socket.getsockname()

print "Serving HTTP on", sa[0], "port", sa[1], "..."

httpd.serve\_forever()

except:

exit()

***index.html***

生成一个临时的 index.html 文件，其内容会被 index.py 更新。

***Dockerfile***

生成一个 Dockerfile，内容为

FROM python:2.7

WORKDIR /code

ADD . /code

EXPOSE 80

CMD python index.py

***haproxy.cfg 内容：***

global

log 127.0.0.1 local0

log 127.0.0.1 local1 notice

defaults

log global

mode http

option httplog

option dontlognull

timeout connect 5000ms

timeout client 50000ms

timeout server 50000ms

listen stats

bind 0.0.0.0:70

stats enable

stats uri /

frontend balancer

bind 0.0.0.0:80

mode http

default\_backend web\_backends

backend web\_backends

mode http

option forwardfor

balance roundrobin

server weba weba:80 check

server webb webb:80 check

server webc webc:80 check

option httpchk GET /

http-check expect status 200

***docker-compose.yml*** 内容为：

这个是 Compose 使用的主模板文件。内容十分简单，指定 3 个 web 容器，以及 1 个 haproxy 容器。

weba:

build: ./web

expose:

- 80

webb:

build: ./web

expose:

- 80

webc:

build: ./web

expose:

- 80

haproxy:

image: haproxy:latest

volumes:

- ./haproxy:/haproxy-override

- ./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro

links:

- weba

- webb

- webc

ports:

- "80:80"

- "70:70"

expose:

- "80"

- "70"

***运行compose项目：***

在compose-haproxy-web 目录下执行 docker-compose up 命

***访问本地的 80 端口，会经过 haproxy 自动转发到后端的某个 web 容器上，刷新页面，可以观察到访问的容器地址的变化。***

***访问本地 70 端口，可以查看到 haproxy 的统计信息。***

***当然，还可以使用 consul、etcd 等实现服务发现，这样就可以避免手动指定后端的 web 容器了，更为灵活。***

### Compose 命令说明

对于 Compose 来说，大部分命令的对象既可以是***项目本身***，也可以指定为***项目中的服务***或者***容器***。如果没有特别的说明，命令对象将是项目，这意味着项目中所有的服务都会受到命令影响。

执行***docker-compose [COMMAND] --help*** 或者***docker-compose help [COMMAND]*** 可以查看具体某个命令的使用格式。

Compose 命令的基本的使用格式是

***docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]***

* -f, --file FILE 指定使用的 Compose 模板文件，默认为 docker-compose.yml，可以多次指定。
* -p, --project-name NAME 指定项目名称，默认将使用所在目录名称作为项目名。
* --x-networking 使用 Docker 的可拔插网络后端特性（需要 Docker 1.9 及以后版本）。
* --x-network-driver DRIVER 指定网络后端的驱动，默认为 bridge（需要 Docker 1.9 及以后版本）。
* --verbose 输出更多调试信息。
* -v, --version 打印版本并退出。

#### 命令使用说明：

##### *build*

格式为**docker-compose build [options] [SERVICE...]。**

构建（重新构建）项目中的服务容器。

服务容器一旦构建后，将会带上一个标记名，例如对于 web 项目中的一个 db 容器，可能是 web\_db。

可以随时在项目目录下运行 docker-compose build 来重新构建服务。

选项包括：

* --force-rm 删除构建过程中的临时容器。
* --no-cache 构建镜像过程中不使用 cache（这将加长构建过程）。
* --pull 始终尝试通过 pull 来获取更新版本的镜像。

##### *kill*

格式为 **docker-compose kill [options] [SERVICE...]。**

通过发送 SIGKILL 信号来强制停止服务容器。

支持通过 -s 参数来指定发送的信号，例如通过如下指令发送 SIGINT 信号。

$ ***docker-compose kill -s SIGINT***

##### *logs*

格式为**docker-compose logs [options] [SERVICE...]。**

查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。可以通过 --no-color 来关闭颜色。

该命令在调试问题的时候十分有用。

##### *pause*

格式为**docker-compose pause [SERVICE...]。**

暂停一个服务容器。

##### *port*

格式为 **docker-compose port [options] SERVICE PRIVATE\_PORT。**

打印某个容器端口所映射的公共端口。

选项：

* --protocol=proto 指定端口协议，tcp（默认值）或者 udp。
* --index=index 如果同一服务存在多个容器，指定命令对象容器的序号（默认为 1）。

##### *ps*

格式为**docker-compose ps [options] [SERVICE...]。**

列出项目中目前的所有容器。

选项：

* -q 只打印容器的 ID 信息。

##### *pull*

格式为 **docker-compose pull [options] [SERVICE...]。**

拉取服务依赖的镜像。

选项：

* --ignore-pull-failures 忽略拉取镜像过程中的错误。

##### *restart*

格式为**docker-compose restart [options] [SERVICE...]。**

重启项目中的服务。

选项：

* -t, --timeout TIMEOUT 指定重启前停止容器的超时（默认为 10 秒）。

##### *rm*

格式为**docker-compose rm [options] [SERVICE...]。**

删除所有（停止状态的）服务容器。推荐先执行 docker-compose stop 命令来停止容器。

选项：

* -f, --force 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
* -v 删除容器所挂载的数据卷。

##### *run*

格式为**docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]。**

在指定服务上执行一个命令。

例如：

$ ***docker-compose run ubuntu ping docker.com***

将会启动一个 ubuntu 服务容器，并执行 ping docker.com 命令。

默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

两个不同点：

* 给定命令将会覆盖原有的自动运行命令；
* 不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 --no-deps 选项，例如

$ docker-compose run --no-deps web python manage.py shell

将不会启动 web 容器所关联的其它容器。

选项：

* -d 后台运行容器。
* --name NAME 为容器指定一个名字。
* --entrypoint CMD 覆盖默认的容器启动指令。
* -e KEY=VAL 设置环境变量值，可多次使用选项来设置多个环境变量。
* -u, --user="" 指定运行容器的用户名或者 uid。
* --no-deps 不自动启动关联的服务容器。
* --rm 运行命令后自动删除容器，d 模式下将忽略。
* -p, --publish=[] 映射容器端口到本地主机。
* --service-ports 配置服务端口并映射到本地主机。
* -T 不分配伪 tty，意味着依赖 tty 的指令将无法运行。

##### *scale*

格式为**docker-compose scale [options] [SERVICE=NUM...]。**

设置指定服务运行的容器个数。

通过 service=num 的参数来设置数量。例如：

$ docker-compose scale web=3 db=2

将启动 3 个容器运行 web 服务，2 个容器运行 db 服务。

一般的，当指定数目多于该服务当前实际运行容器，将新创建并启动容器；反之，将停止容器。

选项：

* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

##### *start*

格式为**docker-compose start [SERVICE...]。**

启动已经存在的服务容器。

##### *stop*

格式为 **docker-compose stop [options] [SERVICE...]**。

停止已经处于运行状态的容器，但不删除它。通过 docker-compose start 可以再次启动这些容器。

选项：

* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

##### *unpause*

格式为**docker-compose unpause [SERVICE...]。**

恢复处于暂停状态中的服务。

##### *up*

格式为 **docker-compose up [options] [SERVICE...]。**

该命令十分强大，它将尝试自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器的一系列操作。

链接的服务都将会被自动启动，除非已经处于运行状态。

可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况，docker-compose up 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。

当通过 Ctrl-C 停止命令时，所有容器将会停止。

如果使用 docker-compose up -d，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

默认情况，如果服务容器已经存在，docker-compose up 将会尝试停止容器，然后重新创建（保持使用 volumes-from 挂载的卷），以保证新启动的服务匹配 docker-compose.yml 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 docker-compose up --no-recreate。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。如果用户只想重新部署某个服务，可以使用 docker-compose up --no-deps -d <SERVICE\_NAME> 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

选项：

* -d 在后台运行服务容器。
* --no-color 不使用颜色来区分不同的服务的控制台输出。
* --no-deps 不启动服务所链接的容器。
* --force-recreate 强制重新创建容器，不能与 --no-recreate 同时使用。
* --no-recreate 如果容器已经存在了，则不重新创建，不能与 --force-recreate 同时使用。
* --no-build 不自动构建缺失的服务镜像。
* -t, --timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）。

##### *migrate-to-labels*

格式为**docker-compose migrate-to-labels。**

重新创建容器，并添加 label。

主要用于升级 1.2 及更早版本中创建的容器，添加缺失的容器标签。

实际上，最彻底的办法当然是删除项目，然后重新创建。

### Compose模板文件：

默认的模板文件名称为 ***docker-compose.yml***，格式为 YAML 格式。

在旧版本（版本 1）中，其中每个*顶级元素为服务名称*，*次级元素为服务容器的配置信息*，例如

webapp:

image: examples/web

ports:

- "80:80"

volumes:

- "/data"

版本 2 扩展了 Compose 的语法，同时尽量保持跟版本 1 的兼容，除了可以声明网络和存储信息外，最大的不同一是添加了版本信息，另一个是需要将所有的服务放到 services 根下面。

例如，上面例子改写为版本 2，内容为

version: "2"

services:

webapp:

image: examples/web

ports:

- "80:80"

volumes:

- "/data"

tip：**注意每个服务都必须通过 image 指令指定镜像或 build 指令（需要 Dockerfile）等来自动构建生成镜像。**

**如果使用 build 指令，在 Dockerfile 中设置的选项(例如：CMD, EXPOSE, VOLUME, ENV 等) 将会自动被获取，无需在 docker-compose.yml 中再次设置。**

#### 指令的用法。

##### build

指定 Dockerfile 所在文件夹的路径（可以是绝对路径，或者相对 docker-compose.yml 文件的路径）。 Compose 将会利用它自动构建这个镜像，然后使用这个镜像。

build: /path/to/build/dir

##### cap\_add, cap\_drop

指定容器的内核能力（capacity）分配。

例如，让容器拥有所有能力可以指定为：

cap\_add:

- ALL

去掉 NET\_ADMIN 能力可以指定为：

cap\_drop:

- NET\_ADMIN

##### command

覆盖容器启动后默认执行的命令。

command: echo "hello world"

##### cgroup\_parent

指定父 cgroup 组，意味着将继承该组的资源限制。

例如，创建了一个 cgroup 组名称为 cgroups\_1。

cgroup\_parent: cgroups\_1

##### container\_name

指定容器名称。默认将会使用 项目名称\_服务名称\_序号 这样的格式。

例如：

container\_name: docker-web-container

需要注意，指定容器名称后，该服务将无法进行扩展（scale），因为 Docker 不允许多个容器具有相同的名称。

##### devices

指定设备映射关系。

例如：

devices:

- "/dev/ttyUSB1:/dev/ttyUSB0"

##### dns

自定义 DNS 服务器。可以是一个值，也可以是一个列表。

dns: 8.8.8.8

dns:

- 8.8.8.8

- 9.9.9.9

##### dns\_search

配置 DNS 搜索域。可以是一个值，也可以是一个列表。

dns\_search: example.com

dns\_search:

- domain1.example.com

- domain2.example.com

##### dockerfile

如果需要指定额外的编译镜像的 Dockefile 文件，可以通过该指令来指定。

例如

dockerfile: Dockerfile-alternate

注意，该指令不能跟 image 同时使用，否则 Compose 将不知道根据哪个指令来生成最终的服务镜像。

##### env\_file

从文件中获取环境变量，可以为单独的文件路径或列表。

如果通过 docker-compose -f FILE 方式来指定 Compose 模板文件，则 env\_file 中变量的路径会基于模板文件路径。

如果有变量名称与 environment 指令冲突，则按照惯例，以后者为准。

env\_file: .env

env\_file:

- ./common.env

- ./apps/web.env

- /opt/secrets.env

环境变量文件中每一行必须符合格式，支持 # 开头的注释行。

# common.env: Set development environment

PROG\_ENV=development

##### environment

设置环境变量。你可以使用数组或字典两种格式。

只给定名称的变量会自动获取运行 Compose 主机上对应变量的值，可以用来防止泄露不必要的数据。

例如

environment:

RACK\_ENV: development

SESSION\_SECRET:

或者

environment:

- RACK\_ENV=development

- SESSION\_SECRET

注意，如果变量名称或者值中用到 true|false，yes|no 等表达布尔含义的词汇，最好放到引号里，避免 YAML 自动解析某些内容为对应的布尔语义。

http://yaml.org/type/bool.html 中给出了这些特定词汇，包括

y|Y|yes|Yes|YES|n|N|no|No|NO

|true|True|TRUE|false|False|FALSE

|on|On|ON|off|Off|OFF

##### expose

暴露端口，但不映射到宿主机，只被连接的服务访问。

仅可以指定内部端口为参数

expose:

- "3000"

- "8000"

##### extends

基于其它模板文件进行扩展。

例如我们已经有了一个 webapp 服务，定义一个基础模板文件为 common.yml。

# common.yml

webapp:

build: ./webapp

environment:

- DEBUG=false

- SEND\_EMAILS=false

再编写一个新的 development.yml 文件，使用 common.yml 中的 webapp 服务进行扩展。

# development.yml

web:

extends:

file: common.yml

service: webapp

ports:

- "8000:8000"

links:

- db

environment:

- DEBUG=true

db:

image: postgres

后者会自动继承 common.yml 中的 webapp 服务及环境变量定义。

使用 extends 需要注意：

* 要避免出现循环依赖，例如 A 依赖 B，B 依赖 C，C 反过来依赖 A 的情况。
* extends 不会继承 links 和 volumes\_from 中定义的容器和数据卷资源。

一般的，推荐在基础模板中只定义一些可以共享的镜像和环境变量，在扩展模板中具体指定应用变量、链接、数据卷等信息。

##### external\_links

链接到 docker-compose.yml 外部的容器，甚至 并非 Compose 管理的外部容器。参数格式跟 links 类似。

external\_links:

- redis\_1

- project\_db\_1:mysql

- project\_db\_1:postgresql

##### extra\_hosts

类似 Docker 中的 --add-host 参数，指定额外的 host 名称映射信息。

例如：

extra\_hosts:

- "googledns:8.8.8.8"

- "dockerhub:52.1.157.61"

会在启动后的服务容器中 /etc/hosts 文件中添加如下两条条目。

8.8.8.8 googledns

52.1.157.61 dockerhub

##### image

指定为镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉去这个镜像。

例如：

image: ubuntu

image: orchardup/postgresql

image: a4bc65fd

##### labels

为容器添加 Docker 元数据（metadata）信息。例如可以为容器添加辅助说明信息。

labels:

com.startupteam.description: "webapp for a startup team"

com.startupteam.department: "devops department"

com.startupteam.release: "rc3 for v1.0"

##### links

链接到其它服务中的容器。使用服务名称（同时作为别名）或服务名称：服务别名 （SERVICE:ALIAS） 格式都可以。

links:

- db

- db:database

- redis

使用的别名将会自动在服务容器中的 /etc/hosts 里创建。例如：

172.17.2.186 db

172.17.2.186 database

172.17.2.187 redis

被链接容器中相应的环境变量也将被创建。

##### log\_driver

类似 Docker 中的 --log-driver 参数，指定日志驱动类型。

目前支持三种日志驱动类型。

log\_driver: "json-file"

log\_driver: "syslog"

log\_driver: "none"

##### log\_opt

日志驱动的相关参数。

例如

log\_driver: "syslog"

log\_opt:

syslog-address: "tcp://192.168.0.42:123"

##### net

设置网络模式。使用和 docker client 的 --net 参数一样的值。

net: "bridge"

net: "none"

net: "container:[name or id]"

net: "host"

##### pid

跟主机系统共享进程命名空间。打开该选项的容器之间，以及容器和宿主机系统之间可以通过进程 ID 来相互访问和操作。

pid: "host"

##### ports

暴露端口信息。

使用宿主：容器 （HOST:CONTAINER）格式，或者仅仅指定容器的端口（宿主将会随机选择端口）都可以。

ports:

- "3000"

- "8000:8000"

- "49100:22"

- "127.0.0.1:8001:8001"

注意：当使用*HOST:CONTAINER*格式来映射端口时，如果你使用的容器端口小于 60 并且没放到引号里，可能会得到错误结果，因为*YAML*会自动解析*xx:yy*这种数字格式为 60 进制。为避免出现这种问题，建议数字串都采用引号包括起来的字符串格式。

##### security\_opt

指定容器模板标签（label）机制的默认属性（用户、角色、类型、级别等）。

例如配置标签的用户名和角色名。

security\_opt:

- label:user:USER

- label:role:ROLE

##### ulimits

指定容器的 ulimits 限制值。

例如，指定最大进程数为 65535，指定文件句柄数为 20000（软限制，应用可以随时修改，不能超过硬限制） 和 40000（系统硬限制，只能 root 用户提高）。

ulimits:

nproc: 65535

nofile:

soft: 20000

hard: 40000

##### volumes

数据卷所挂载路径设置。可以设置宿主机路径 （HOST:CONTAINER） 或加上访问模式 （HOST:CONTAINER:ro）。

该指令中路径支持相对路径。例如

volumes:

- /var/lib/mysql

- cache/:/tmp/cache

- ~/configs:/etc/configs/:ro

##### volumes\_driver

较新版本的 Docker 支持数据卷的插件驱动。

用户可以先使用第三方驱动创建一个数据卷，然后使用名称来访问它。

此时，可以通过 volumes\_driver 来指定驱动。

volume\_driver: mydriver

##### volumes\_from

从另一个服务或容器挂载它的数据卷。

volumes\_from:

- service\_name

- container\_name

##### 其它指令

此外，还有包括 cpu\_shares, cpuset, domainname, entrypoint, hostname, ipc, mac\_address, mem\_limit, memswap\_limit, privileged, read\_only, restart, stdin\_open, tty, user, working\_dir 等指令，基本跟 docker-run 中对应参数的功能一致。

例如，指定使用 cpu 核 0 和 核 1，只用 50% 的 CPU 资源：

cpu\_shares: 73

cpuset: 0,1

指定服务容器启动后执行的命令。

entrypoint: /code/entrypoint.sh

指定容器中运行应用的用户名。

user: nginx

指定容器中工作目录。

working\_dir: /code

指定容器中搜索域名、主机名、mac 地址等。

domainname: your\_website.com

hostname: test

mac\_address: 08-00-27-00-0C-0A

指定容器中

ipc: host

指定容器中内存和内存交换区限制都为 1G。

mem\_limit: 1g

memswap\_limit: 1g

允许容器中运行一些特权命令。

privileged: true

指定容器退出后的重启策略为始终重启。该命令对保持服务始终运行十分有效，在生产环境中推荐配置为 always 或者 unless-stopped。

restart: always

以只读模式挂载容器的 root 文件系统，意味着不能对容器内容进行修改。

read\_only: true

打开标准输入，可以接受外部输入。

stdin\_open: true

模拟一个假的远程控制台。

tty: true

##### example2:

***tipe:将创建的镜像pushed*** [a registry](https://docs.docker.com/get-started/part2/#share-your-image). 我们使用这个共享的image.并保证启动的容器可用。

###### docker-compose.yml

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repository:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "80:80"

networks:

- webnet

networks:

webnet:

***tipe:***

该docker-compose.yml文件告诉Docker执行以下操作：

* 从registry中拉取之前创建推送上去的镜像。
* 运行该映像的五个实例作为调用的服务web，限制每个实例使用，最多使用10％的CPU（跨所有内核）和50MB RAM。
* 如果发生故障，立即重新启动容器。
* 将端口80映射到主机web端口80。
* 指示web容器通过称为负载平衡网络共享端口80 webnet。（在内部，Docker本身将web在短暂的港口发布到 80号港口。）
* webnet使用默认设置（这是一个负载平衡的重叠网络）来定义网络。
* 我们可以使用docker stack deploy命令运行文件（也仅在Compose文件3.x及以上支持）。您可以使用*非群组*配置docker-compose up运行版本3文件，但是由于我们正在构建一个群体示例，因此我们专注于堆栈部署。
* 您可以将Compose文件命名为任何您希望使其在逻辑上对您有意义的任何内容; docker-compose.yml只是一个标准名称。我们可以很容易地将这个文件docker-stack.yml或更具体的项目称为我们的项目。

***堆栈部署：***

在我们可以使用docker stack deploy命令之前，我们先运行：

docker swarm init

现在我们来运行它 你必须给你的应用程序一个名字。在这里，它设置为 getstartedlab：

docker stack deploy -c docker-compose.yml getstartedlab

看到刚刚推出的五个容器的列表：

docker stack ps getstartedlab

###### 缩放应用程序

您可以通过更改replicas值docker-compose.yml，保存更改并重新运行docker stack deploy命令来缩放应用程序：

docker stack deploy -c docker-compose.yml getstartedlab

Docker将进行就地更新，无需首先撕下堆叠或者杀死任何容器。

现在，重新运行docker stack ps命令以重新配置已部署的实例。例如，如果您放大副本，则会有更多的运行容器。

###### 停止app和swarm

将应用程序停掉使用docker stack rm：

docker stack rm getstartedlab

这会删除应用程序，但是我们的单节点群组仍然运行（如下所示docker node ls）。停掉这个swarm使用docker swarm leave --force。

与Docker一样容易启动扩展您的应用程序。您在学习如何运行生产中的Docker方面迈出了巨大的一步。接下来，您将了解如何在Docker机器Swarm上运行此应用程序作为bonafide群集。

**注意**：撰写这样的文件可用于使用Docker定义应用程序，并可以使用[Docker Cloud](https://docs.docker.com/docker-cloud/)上传到云提供商，也可以使用[Docker Enterprise Edition](https://www.docker.com/enterprise-edition)选择的任何硬件或云提供商 。

docker stack ls # List all running applications on this Docker host

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker stack services <appname> # List the services associated with an app

docker stack ps <appname> # List the running containers associated with an app

docker stack rm <appname> # Tear down an application

### 读取环境变量

从 1.5.0 版本开始，Compose 模板文件支持动态读取主机的系统环境变量。

例如，下面的 Compose 文件将从运行它的环境中读取变量 ${MONGO\_VERSION} 的值，并写入执行的指令中。

db:

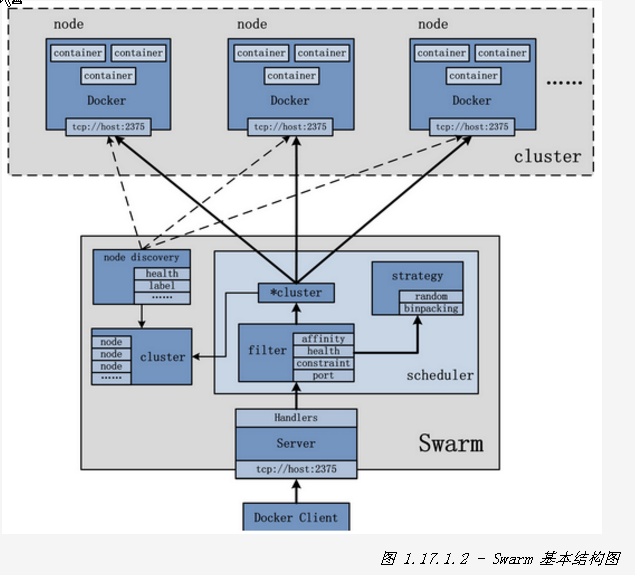
image: "mongo:${MONGO\_VERSION}"

如果执行 MONGO\_VERSION=3.0 docker-compose up 则会启动一个 mongo:3.0 镜像的容器；如果执行 MONGO\_VERSION=2.8 docker-compose up 则会启动一个 mongo:2.8 镜像的容器。

## Docker-Swarm项目：

提供 Docker 容器集群服务，是 Docker 官方对容器云生态进行支持的核心方案。

使用它，用户可以将多个 Docker 主机封装为单个大型的虚拟 Docker 主机，快速打造一套容器云平台。



### docker 1.12版本之前：

#### 下载swarm镜像：

***$ docker pull swarm***

验证版本：

***$ docker run –rm swarm –v***

#### 配置节点：

Docker 主机在加入 Swarm 集群前，需要进行一些简单配置，添加 Docker daemon 的网络监听。

##### 1、例如，在启动 Docker daemon 的时候通过 -H 参数：

$ sudo docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

<或者：sudo dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock >

注：Docker 1.8.0 版本之前不支持 daemon 命令，可以用 -d 代替。

##### 2、如果是通过服务方式启动，则需要修改服务的配置文件。

以 Ubuntu 14.04 为例，配置文件为 /etc/default/docker（其他版本的 Linux 上略有不同）。

在文件的最后添加：

DOCKER\_OPTS="$DOCKER\_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock"

以CentOs7 为例：

vi /lib/systemd/system/docker.service

在 ExecStart=/usr/bin/dockerd 行后加-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

如： ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock

（注意防火墙对端口的影响）

#### 启动集群：

Docker 集群管理需要使用服务发现（Service Discover）功能，Swarm 支持以下的几种方式：DockerHub、本地文件、etcd、consel、zookeeper 和手动指定节点 IP 地址信息等。

除了手动指定外，这些方法原理上都是通过维护一套数据库机制，来管理集群中注册节点的 Docker daemon 的访问信息。

本地配置集群推荐使用 consel 作为服务发现后端。利用社区提供的 Docker 镜像，整个过程只需要三步即可完成。

##### 启动 Consel 服务后端

启动 consel 服务容器，映射到主机的 8500 端口。

***$ docker run -d -p 8500:8500 --name=consul progrium/consul -server -bootstrap***

获取到本地主机的地址作为 consul 的服务地址：<consul\_ip>:8500

##### 启动管理节点

首先，启动一个主管理节点，映射到主机的 4000 端口，并获取所在主机地址为 <manager0\_ip>。其中 4000 端口是 Swarm 管理器的默认监听端口，用户也可以指定映射为其它端口。

***$ docker run -d -p 4000:4000 swarm manage -H :4000 --replication --advertise <manager0\_ip>:4000 consul://<consul\_ip>:8500***

为了提高高可用性，用户也可以启动从管理节点。假定获取所在主机地址为 <manager1\_ip>。

***$ docker run -d swarm manage -H :4000 --replication --advertise <manager1\_ip>:4000 consul://<consul\_ip>:8500***

##### 启动工作节点

需要在每个工作节点上启动 agent 服务。

获取节点的主机地址为 <node\_ip>，并指定前面获取到的 consel 服务地址。

***$ docker run -d swarm join --advertise=<node\_ip>:2375 consul://<consul\_ip>:8500***

节点启动后，用户可以指定 Docker 服务地址为 <manager0\_ip>:4000> 来测试各种 Docker 命令，可以看到整个 Swarm 集群就像一个虚拟的 Docker 主机一样正常工作。

由于 Swarm 实际上是通过 agent 调用了本地的 Docker daemon 来运行容器，当 Swarm 集群服务出现故障时，无法接受新的请求，但已经运行起来的容器将不会受到影响。

### docker 1.12版本之后：集成了swarm

相比于docker-compose可以管理当前宿主机上所有的服务，docker swarm可以统一管理多台负载机器上的所有服务

#### leader节点swarm初始化：

1. ***docker swarm init***

***docker swarm init --advertise-addr 10.128.112.132:2377***

初始化docker swarm，命令正确执行后的内容类型下面内容：

**[was@dqapp112132 docker-compose]$** *docker swarm init*

*Swarm initialized: current node (fbhzdzt24c9uih5fqchvwpv14) is now a manager.*

*To add a worker to this swarm, run the following command:*

*docker swarm join --token SWMTKN-1-0fviu393dhjftunv8vz6qsctcn34i2tg3mbii5bg8xms2d9y8u-03k73ziio3l8s913nbv0wt02g 10.128.112.132:2377*

*To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.*

请将里面的 docker swarm join --token SWMTKN-1-0fviu393dhjftunv8vz6qsctcn34i2tg3mbii5bg8xms2d9y8u-03k73ziio3l8s913nbv0wt02g 10.128.112.132:2377 备份好，如果忘记可以在master节点上使用命令***docker swarm join-token -q manager***  或者再worker节点上执行：d***ocker –H <leader节点ip> swarm join-token –q manager***显示出join token：。

***tipe:***请将示例里的IP替换成你宿主机器的IP，端口号不变。

执行完该命令，当前宿主机扮演docker swarm 的manager角色了，它负责统一管理所有node上的容器。

查看加入的节点：

在master节点执行：***docker node ls***

或者再worker执行：***docker –H <leader ip> node ls***

#### worker节点加入swarm集群：

如上master节点提示命令：

***docker swarm join --token SWMTKN-1-0fviu393dhjftunv8vz6qsctcn34i2tg3mbii5bg8xms2d9y8u-03k73ziio3l8s913nbv0wt02g <master 节点ip>:2377***

***tipe: 在worker机器上执行，注意token和master节点ip的替换。***

#### 使用与发布应用：

4.***docker stack deploy -c docker-compose.yml zhifu***

该命令用来发布、启动所有节点上的容器，需要用到docker-compose.yml文件，在该文件里，你配置你的所有服务，可以参考7中示例理解。最后一个参数zhifu自定义，指定你服务的名称，并且最好记住，因为你需要这个参数去rm、ps操作。

增加／减少 node／实例 ，更改配置等操作后，都只需要重新执行一下该命令即可生效，属于热部署，不会影响到正在运行的容器，除非你减少node、实例。

5. ***docker stack ps***

该命令跟docker ps具有相同的功能，只不过docker stack ps是用来查看docker swarm下的容器信息

示例： docker stack ps zhifu # 查看zhifu相关的容器信息

6. ***docker stack rm***

该命令用来停止响应的swarm服务，

示例：docker stack rm zhifu ＃ 该命令执行后，zhifu的集群就会停止，你使用docker stack ps zhifu 将不会看到任何容器，并且docker ps也不会看到响应的容器信息

7. ***docker-compose.yml 示例***

version: '3' ＃必需是3

services:

prod\_zhifu\_asyncquerygo: ＃ 服务名

image: 10.251.26.11:5000/prod-image-zhifu-asyncquerygo:v3 ＃镜像名，此处直接从仓库拉镜像

restart: always ＃自动重启

deploy:

replicas: 2 ＃实例数量

resources:

limits: ＃限制配置

cpus: "0.3" ＃cpu使用率不能超过0.3

memory: 5000M ＃内存使用率不能超过5000M

restart\_policy: ＃重启规则

condition: on-failure ＃ 重启条件：失败

ports:

- 8001:8001 ＃将宿主机8001 端口映射到容器8001端口

container\_name: prod-zhifu-asyncquerygo ＃ 容器名称，swarm里这个参数没啥用，因为swarm根本不用这个，而是采用 docker stack deploy -c docker-compose.yml zhifu 命令里的zhifu按照一定的规则给容器命名的

volumes:

- /app/logs/prod-lichunyan/go\_apps\_logs:/opt/logs

environment:

- LOG\_PREFIX=/opt/logs

prod\_zhifu\_osa:

image: 10.251.26.11:5000/prod-image-zhifu-osa:v1

restart: always

deploy:

replicas: 2

resources:

limits:

cpus: "0.2"

memory: 2000M

restart\_policy:

condition: on-failure

ports:

- 6101:6101

container\_name: prod-zhifu-osa

volumes:

- /app/logs/prod-lichunyan:/opt/logs

environment:

- LOG\_PREFIX=/opt/logs

prod\_zhifu\_osr:

image: 10.251.26.11:5000/prod-image-zhifu-osr:v1

restart: always

deploy:

replicas: 2

resources:

limits:

cpus: "0.2"

memory: 2000M

restart\_policy:

condition: on-failure

ports:

- 6201:6201

container\_name: prod-zhifu-osr

volumes:

- /app/logs/prod-lichunyan:/opt/logs

environment:

- LOG\_PREFIX=/opt/logs

#### swarm部署启动是报的错误：

* 1. 所有服务只在leader机器上启动，而worker上没有服务进程：
     + 1. 网络错误：
       - 解决：

在docker-compose文件中添加

networks:

webnet:

2．卷挂载错误：

解决：

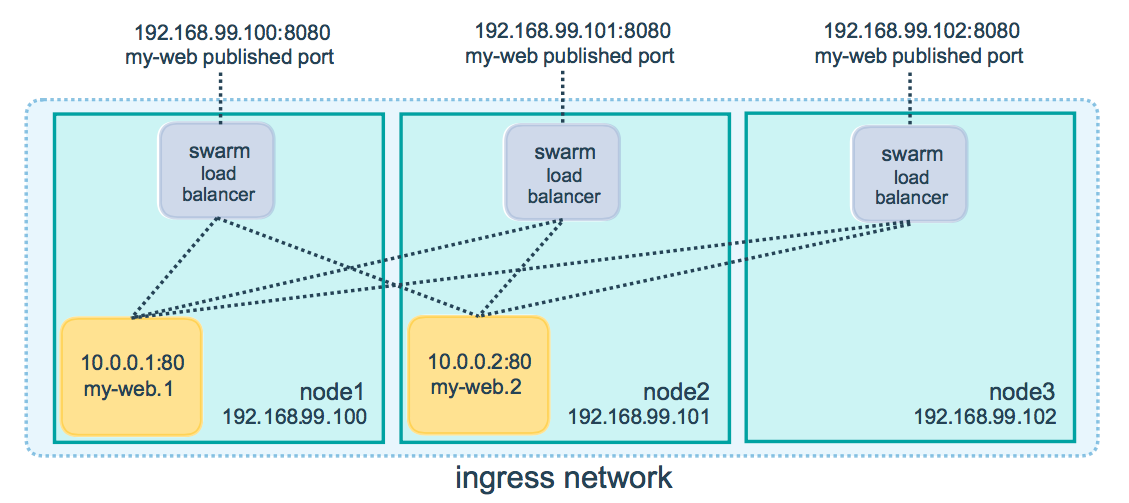
在worker节点上需要手动创建卷目录

#### swarm 负载原理：

##### 访问您的群集

你可以从IP地址来访问你的应用程序**要么** myvm1或myvm2。您创建的网络在它们之间共享，并且负载平衡。运行 docker-machine ls以获取您的虚拟机的IP地址，并在浏览器上访问它们，然后点击刷新（或仅仅curl它们）。您会看到五个可能的容器ID，它们都随机循环，显示了负载平衡。

IP地址工作的原因是群集中的节点参与入口**路由网格**。这样可以确保在群集中某个端口部署的服务始终将该端口保留给其自身，无论实际运行的是哪个节点。以下是在三节点群集my-web端口8080上发布的服务的路由网格如何显示：



**有连通性麻烦？**

请记住，为了在群集中使用入口网络，在启用群组模式之前，需要在群集节点之间打开以下端口：

* 端口7946用于容器网络发现的TCP / UDP。
* 端口4789 UDP用于容器入口网络。