Table of Contents

Introduction

[TOC]

一、什么是 Docker

Docker 是基于 GO 语言实现的开源容器项目，诞生于 2013 年年初，最初发起者是 dotCloud 公司，Docker 项目已经加入了 Linux 基金会，并遵循 apache2.0 协议，全部代码开源在 github 上，docker 的构想是要实现 "Build ship and run any,anywhere", 即通过对应用的封装、分发、部署、运行 生命周期进行管理，达到应用组件 "一次封装，到处运行" 的目的。

Docker 是一种容器虚拟化，Docker 容器可以理解为一种轻量级的沙盒，每个容器内运行着一个应用，不同的容器相互隔离，容器之间也可以通过网络互相通信，也可以说 docker 就是轻量级及互相隔离应用的虚拟化技术

二、Docker 的优势

更快的交付和部署

更高效的资源利用

更轻松的迁移和扩展

更简单的更新管理

docker 容器很快，启动和停止可以在秒级实现

docker 容器对系统资源需求很少，一台主机上可以同时运行数千个容器

docker 通过类型 git 设计理念的操作来方便用户获取、分发、更新应用镜像，存储复用，增量更新

docker 利用 Linux 系统上的多种防护技术实现了严格的隔离可靠性，并且可以整合众多的安全工具

三、Docker 容器技术和传统虚拟机技术的区别

传统的方式是在硬件层面实现虚拟化，需要有额外的虚拟机管理应该和虚拟机操作系统层

docker 容器是在操作系统层面上实现的虚拟化，直接复用本地主机的操作系统，因此更加的轻量级

四、Docker 核心三大概念

docker 的大部分操作都是围绕着它的三大核心概念 -- 镜像、容器、仓库来展开的 这个就相当于 docker 的基础很重要

4.1. docker 镜像

docker 镜像类似于虚拟机的镜像文件，可以将它理解为一个只读的模板，例如一个镜像可以包含一个基本的操作系统，里面安装这个应用程序镜像是创建 docker 容器的基础

详细信息请点击 -----> 镜像详情

4.2. docker 容器

docker 容器就像一个轻量级的沙盒，docker 利用容器来运行和隔离应用，容器是从镜像创建的应用运行实例，可以启动、停止、删除，而这些容器都是互相隔离，互不可见的

详细信息请点击 -----> 容器详情

4.3. docker 仓库

docker 仓库类型于代码仓库，它是集中存放镜像文件的地方

详细信息请点击 -----> 仓库详情

PS：镜像自身是只读的，容器从镜像启动的时候，会在镜像的最上层创建一个可写层

五、安装 Docker

```
yum -y install docker-io
或者
curl -fsSL https://get.docker.com/ | sh

安装指定版本的docker
在使用centos7，并使用yum安装docker的时候，往往不希望安装最新版本的doc
# 安装依赖包
yum install -y yum-utils device-mapper-persistent-data lvm2 libs
# 添加Docker软件包源
yum-config-manager --add-repo http://mirrors.aliyun.com/docker-c
# 关闭测试版本list（只显示稳定版）
yum-config-manager --enable docker-ce-edge
yum-config-manager --enable docker-ce-test
# 更新yum包索引
yum makecache fast
# 找到需要安装的
yum list docker-ce --showduplicates|sort -r
# 指定版本安装
yum install docker-ce-17.09.0.ce -y
# 启动docker
systemctl start docker && systemctl enable docker

查看docker版本
docker version
```

六、Docker 服务配置项

```
配置文件：/etc/default/docker
进程ID：/var/run/docker.pid
日志文件:/var/log/upstart/docker.log
```

[TOC]

一. 获取镜像

1.1. docker pull

镜像是运行容器的前提，也就是说没有镜像就没有办法创建容器

获取镜像的命令:
docker pull
这个命令可以直接在 docker Hub 镜像源下载镜像

该命令的格式是:
docker pull NAME[:TAG]
其中 name 是镜像仓库的名称 (用来区分镜像)tag 是镜像的标签 (用来标注版本)，通常情况下我们描述一个镜像需要包括 "名称 + 标签" 信息

```
我们pull一个centos的镜像
[root@rsync131 ~]# docker pull centos
Using default tag: latest
latest: Pulling from library/centos
256b176beaff: Downloading [==========>

这里我没有指定标签，它的下载的默认标签是latest
```

下载镜像的过程解析：

下载过程我们可以看出，镜像文件一般由若干层组成 256b176beaff 这样的串是这个镜像的唯一 ID(这里只是一个短的 ID，实际上完整的 ID 包括 256 比特，由 64 个十六进制字符组成)。使用 docker pull 命令下载时会获取并输出镜像的各层信息。当不同的镜像包括相同的层时，本地仅存储层的一份内容，减小的存储需要的空间。

```
pull 子命令支持的选项主要包括：
-a,--all-tags=true|false：是否获取仓库中的所有镜像，默认是否。
```

二. 查看镜像信息

2.1. docker images

使用 docker images 命令可以列出本地主机上的已有镜像基本信息

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we

REPOSITORY：来自哪个仓库
TAG：镜像的标签
IMAGE ID：镜像的唯一ID
CREATED：创建的时间
SIZE：镜像的大小


其中镜像的ID信息是十分重要的，因为它是镜像的唯一值
```

```
images子命令主要支持的选项如下：
-a,--all=true|false：列出所有的镜像文件(包括临时文件)，默认为否
--digests=true|false：列出镜像的数字摘要值，默认为否
-f,--filter=[]：过滤列出的镜像，如dangling=true只显示没有被使用的镜
--format="TEMPLATE"：控制输出格式，如.ID代表ID信息
--no-trunc=true|false：对输出格式中太长的部分是否进行截断，默认为是
-q,--quiet=true|false：仅输出ID信息，默认为否

更多的子命令可以通过man docker-images来查看
```

2.2. docker tag

使用 docker tag 可以给镜像添加标签信息

```
[root@rsync131 ~]# docker tag centos:latest mycentos:1.0
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
mycentos            1.0                 5182e96772bf        6 we

我们可以看到mycentos和centos的ID、大小完全一致，实际上他们就是一个镜
```

2.3. docker inspect

使用 docker inspect 命令可以查看镜像的详细信息，包括制作者、适应架构、各层的数字摘要等很多信息

```
[root@rsync131 ~]# docker inspect centos
[
    {
        "Id": "sha256:5182e96772bf11f4b912658e265dfe0db8bd314475
        "RepoTags": [
            "centos:latest",
            "mycentos:1.0"
        ],
        "RepoDigests": [
            "centos@sha256:6f6d986d425aeabdc3a02cb61c02abb2e78e5
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2018-08-06T19:21:48.235227329Z",
        "Container": "d60ffc9ddd12462af4bdcdbe45b74f3b3f99b46607
        "ContainerConfig": {
            "Hostname": "d60ffc9ddd12",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/bash\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:748eacc0f236df2fc9ba87c4d76a66cb107
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20180804",
                "org.label-schema.license": "GPLv2",
```

```
                "org.label-schema.name": "CentOS Base Image",
                "org.label-schema.schema-version": "1.0",
                "org.label-schema.vendor": "CentOS"
            }
        },
        "DockerVersion": "17.06.2-ce",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
            ],
            "Cmd": [
                "/bin/bash"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:748eacc0f236df2fc9ba87c4d76a66cb107
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {
                "org.label-schema.build-date": "20180804",
                "org.label-schema.license": "GPLv2",
                "org.label-schema.name": "CentOS Base Image",
                "org.label-schema.schema-version": "1.0",
                "org.label-schema.vendor": "CentOS"
            }
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 199723824,
        "VirtualSize": 199723824,
        "GraphDriver": {
            "Data": {
                "MergedDir": "/var/lib/docker/overlay2/8d76bee4a
```

```
                "UpperDir": "/var/lib/docker/overlay2/8d76bee4ae
                "WorkDir": "/var/lib/docker/overlay2/8d76bee4ae9
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:1d31b5806ba40b5f67bde96f18a181668348934a
            ]
        },
        "Metadata": {
            "LastTagTime": "2018-09-22T12:25:06.648773262+08:00"
        }
    }
]
```

其实返回的就是一个 json 格式的信息，我们也可以用 -f 参数来获取指定的信息

```
[root@rsync131 ~]# docker inspect -f {{".Architecture"}} centos
amd64
```

回到顶部

2.4. docker history

使用 docker history 命令可以查看镜像的历史信息

既然镜像文件是由多层组成的，那么怎么知道每一层都干了什么呢，可以使用 history 命令来查看

```
[root@rsync131 ~]# docker history ubuntu
IMAGE               CREATED             CREATED BY
cd6d8154f1e1        2 weeks ago         /bin/sh -c #(nop)  CMD [
 2 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo 'd
 2 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)
 2 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*
 2 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' >
 2 weeks ago         /bin/sh -c #(nop) ADD file:3df374a69ce696c2


这里有一些长的命令被自动截断了，可以使用--no-trunc选项来显示完整的信息
```

三. 搜索镜像

3.1. docker search

使用 docker search 命令可以搜索远端仓库的共享镜像，默认是搜索官方仓库中的镜像文件

```
语法格式：


docker search TERM

支持的选项包括：

--automated=true|false：仅显示自动创建的镜像，默认为否

--no-trunc=true|false：输出信息不截断显示，默认为否

-s,--stars=X：指定仅显示评论为指定星级以上的镜像，默认为0，即输出所有
```

搜索所有自动创建评价为 1 + 的带 nginx 的镜像

```
[root@rsync131 ~]# docker search --automated -s 3 nginx
Flag --automated has been deprecated, use --filter=is-automated=
Flag --stars has been deprecated, use --filter=stars=3 instead
NAME                                                    DESCRIPTI
jwilder/nginx-proxy                                     Automated
richarvey/nginx-php-fpm                                 Container
jrcs/letsencrypt-nginx-proxy-companion                  LetsEncry
webdevops/php-nginx                                     Nginx wit
zabbix/zabbix-web-nginx-mysql                           Zabbix fr
bitnami/nginx                                           Bitnami n
1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5    ubuntu-16
tobi312/rpi-nginx                                       NGINX on
blacklabelops/nginx                                     Dockerize
wodby/drupal-nginx                                      Nginx for
nginxdemos/hello                                        NGINX web
webdevops/nginx                                         Nginx con
1science/nginx                                          Nginx Doc
behance/docker-nginx                                    Provides
```

四. 删除镜像

4.1. docker rmi

使用 docker rmi 命令可以删除指定的镜像文件

```
语法格式：
docker rmi IMAGE [IMAGE....]

PS：当本地一个镜像有多个标签的时候，docker rmi 只会删除指定的标签镜像

支持的选项参数：
-f ： 强制删除镜像文件
```

通过标签删除

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
mycentos            1.0                 5182e96772bf        6 we
centos              latest              5182e96772bf        6 we
[root@rsync131 ~]# docker rmi mycentos:1.0
Untagged: mycentos:1.0
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
```

通过 ID 删除

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
[root@rsync131 ~]# docker rmi 5182e96772bf
Untagged: centos:latest
Untagged: centos@sha256:6f6d986d425aeabdc3a02cb61c02abb2e78e5735
Deleted: sha256:5182e96772bf11f4b912658e265dfe0db8bd314475443b64
Deleted: sha256:1d31b5806ba40b5f67bde96f18a181668348934a44c9253b
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
```

强制删除

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
[root@rsync131 ~]# docker rmi -f 5182e96772bf
Untagged: centos:latest
Untagged: centos@sha256:6f6d986d425aeabdc3a02cb61c02abb2e78e5735
Deleted: sha256:5182e96772bf11f4b912658e265dfe0db8bd314475443b64
Deleted: sha256:1d31b5806ba40b5f67bde96f18a181668348934a44c9253b
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
```

五. 创建镜像

其实创建镜像有三种方式，基于已有的镜像的容器创建，基于本地模板导入，基于 Dockerfile 创建，这里只介绍下前两种，因为 dockerfile 是一个很高级的东西会有专门的文章来介绍：点击 Dockerfile

5.1. docker commit

docker commit 是基于已有的镜像的容器创建

```
语法格式：
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

支持的选项包括：
-a,--author="": 作者的信息
-c,--change=[]: 提交的时候执行的Dockerfile指令，包括CMD|ENTRYPOINT|
-m,--message="": 提交的信息
-p,--pause=true: 提交时暂停容器运行
```

因为是基于容器创建的镜像，所以要先创建个容器

```
# 创建个容器，touch个文件，这里相对源镜像已经发生了改变 c86d860188f3
[root@rsync131 ~]# docker run -it centos bash
[root@c86d860188f3 /]# touch a.txt
[root@c86d860188f3 /]# exit
exit

# 创建镜像基于c86d860188f3 容器
[root@rsync131 ~]# docker commit -m "add a.txt" -a "zhujingzhi"
sha256:64b8d4f3a1901353deeb31e8b646c0f04dd58accddc951814fdce237e

# 查看镜像
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
a_centos            1.0                 64b8d4f3a190        19 s
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
```

5.2. docker import

也可以直接从一个操作系统文件导入一个镜像，只要使用 docker import 命令

```
语法格式：

docker import [OPTIONS] file|URL|-[REPOSITORY[:TAG]]
```

给朋友们个链接下载模板：
https://download.openvz.org/template/precreated/

```
[root@rsync131 ~]# ll
总用量 345664
-rw-------. 1 root root      1513 8月  20 20:25 anaconda-ks.cfg
-rw-r--r--  1 root root 145639219 9月  22 14:12 centos-7-x86_64-

[root@rsync131 ~]# cat centos-7-x86_64-minimal.tar.gz | docker i
sha256:4dec53e6e02af646126879ac69c8bee4ef6bdcf9fe0cc86d3a92959ce

[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
centos-mini         1.0                 4dec53e6e02a        4 se
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
```

六. 镜像的导出和导入

6.1. docker save

使用 docker save 可以把镜像导出

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
a_centos            1.0                 64b8d4f3a190        17 m
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
[root@rsync131 ~]# docker save -o centos_bak.tar.gz centos
[root@rsync131 ~]# ll
总用量 203436
-rw-------. 1 root root      1513 8月  20 20:25 anaconda-ks.cfg
-rw-------  1 root root 208301056 9月  22 13:47 centos_bak.tar.g
```

6.2. docker load

使用 docker load 可以把使用 docker save 导出的文件导入

```
[root@rsync131 ~]# docker load --input centos_bak.tar.gz
或者
[root@rsync131 ~]# docker load < centos_bak.tar.gz
1d31b5806ba4: Loading layer [==================================
Loaded image: centos:latest
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
ubuntu              latest              cd6d8154f1e1        2 we
centos              latest              5182e96772bf        6 we
```

[TOC]

一、创建容器

容器是 Docker 另一个核心的概念，简单来说，容器是镜像的一个运行实例，所不同的是，镜像是静态的只读文件，而容器带有运行时需要的可写文件层

1.1. 新建容器

使用 docker create 命令来新建容器

```
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
docker.io/centos    latest              5182e96772bf        6 we
[root@rsync131 ~]# docker create centos
b9dd06b5dcba561b178b1892631cd318f0babd4a9a3f067761963a6ab61fe078
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
b9dd06b5dcba        centos              "/bin/bash"         9 se
```

1.2. 启动容器

使用 docker start 命令来启动容器

```
[root@rsync131 ~]# docker start b9dd06b5dcba
b9dd06b5dcba
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
b9dd06b5dcba        centos              "/bin/bash"         Abou
```

PS: 不知道为什么我使用这样的方式是启动不了容器的，待解释

1.3. 新建并启动容器

一般在创建容器的是我们不会使用上面的两步的方式来进行创建，有个更简单的方式就是在创建的时候直接启动

使用 docker run 命令，创建并启动容器

```
[root@rsync131 ~]# docker run -it -d --name test_centos centos
159a08c11fece853626516ab94b401464efd85d69c1f88706ec17a29178fee54
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         3 se


docker run：死命令
-i：让容器的标准输入保持打开
-t：分配个伪终端
-d：守护状态运行
--name：指定容器名字参数
test_centos：容器名字
centos：镜像名或者镜像ID
```

这里我们说一下 docker run 执行的时候都干了什么

检查本地镜像是否存在，不存在则会自动去公有仓库去下载

利用镜像创建容器，并启动容器

分配一个文件系统给容器，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

从网桥的地址池配置一个 IP 给容器

执行用户指定的应用程序

1.4. 查看容器运行日志

使用命令 docker logs 来查看日志

```
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         21 m
[root@rsync131 ~]# docker logs 159a08c11fec
```

回到顶部

二、终止容器

2.1. 终止容器

使用 docker stop 来终止正在运行的容器

```
语法格式：
docker stop 容器名

常用选项：
-t, --time[=10] : 10秒后发送终止信号
```

使用 docker kill 命令可以直接终止容器

docker kill 命令会直接发送 SIGKILL 信号来强行终止容器

```
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         29 m
[root@rsync131 ~]# docker stop 159a08c11fec
[root@rsync131 ~]# docker kill 159a08c11fec
```

2.2. 查看终止的容器

使用 docker ps 只是可以查看正在运行的容器

使用 docker ps -a 则可以看到所有的容器包括正在运行的和已经终止的容器

```
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         30 m
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         30 m
b9dd06b5dcba        centos              "/bin/bash"         36 m
```

2.3. 启动和重启已经终止的容器

已经终止的容器我们可以使用 docker start 命令来启动

也可以使用 docker restart 命令将一个正在运行的容器先 stop 后 start

```
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         30 m
b9dd06b5dcba        centos              "/bin/bash"         36 m
[root@rsync131 ~]# docker start b9dd06b5dcba
b9dd06b5dcba
[root@rsync131 ~]# docker restart 159a08c11fec
159a08c11fec
```

回到顶部

三、进入容器

在使用 -d 选项来运行容器的时候，容器会在后台运行，用户是无法看到容器的信息的，也无法进入操作

这个时候如果我们要进入容器进行操作，应该怎么办呢？在 docker 中进入 docker 中的方法有很多种，比如官方的 attach 和 exec 命令，以及第三方的 nsenter 工具 (这个没有用过 最常用的还是 exec) 等

3.1. attach 命令

```
attach命令是docker自带的命令
语法格式：
docker attach [--detach-keys[=[]]] [--no-stdin] [--sig-proxy[=tr

支持的常用主要选项：
--detach-keys[=[]]：指定退出attach模式的快捷键序列，默认是CTRL+p CT
--no-stdin=true|false :是否关闭标准输入，默认true
--sig-proxy=true|false ： 是否代理收到的系统信号给应用程序，默认是tru
```

```
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         41 m
[root@rsync131 ~]# docker attach 159a08c11fec
[root@159a08c11fec /]#
```

PS：使用 attach 命令进入容器是很不方便的，就是这个命令只允许一个人使用容器，在开多个窗口同时用 attach 进入容器的时候，窗口执行的命令会同步，当一个窗口阻塞了，其他窗口就不能玩了，所有不用这个命令，太坑了

3.2. exec 命令

为了解决上面的坑，docker 在 1.3.0 版本提供了 exec 这个命令解决了 attach 命令的坑

```
语法格式：
docker exec  [-d|--detach] [--detach-keys[=[]]] [-i|--interactiv

主要选项参数：
-i,  --interactive=true|false：打开标准输入接收用户输入的命令，默认
--privileged=true|false ：是否给执行命令以高权限，默认false
-t, --tty=true|false：分配伪终端 默认false
-u, --user="" ：  执行命令的用户或者ID
```

```
[root@rsync131 ~]# docker exec -it 159a08c11fec bash
[root@159a08c11fec /]# ls
anaconda-post.log  bin  dev  etc  home  lib  lib64  media  mnt
[root@159a08c11fec /]#
```

可以看到，一个 bash 终端打开了，在不影响容器内其他应用的前提下，用户可以很容易的与容器进行交互

PS：通过制定 -it 参数保持标准输入，并且分配个伪终端，是很推荐的方式哦

回到顶部

四、删除容器

使用 docker rm 命令来删除处于终止或者退出状态的容器

```
语法格式:
docker rm [-f|--force] [-l|--link] [-v|--volumes] CONTAINER [CON

主要选项参数:
-f,--force=false: 是否强行终止并删除运行中的容器
-l,--link=false: 删除容器的链接，但是保留容器
-v,--volumes=false: 删除容器挂载的数据卷
b
```

```
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         2 ho
b9dd06b5dcba        centos              "/bin/bash"         2 ho
[root@rsync131 ~]# docker rm b9dd06b5dcba
b9dd06b5dcba
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         2 ho
```

默认情况下，docker rm 命令只能删除已经终止的容器，不能删除运行中的容器，想要删除运行中的容器可以使用 -f 参数，来强制的删除

回到顶部

五、容器的导入与导出

在做容器迁移的时候我们就要先把容器导出，然后在导入到其他的地方

5.1. 导出容器

使用命令 docker export 命令来导出容器

```
语法格式:
docker export [-o|--output[=""]] CONTAINER

主要选项参数:
-o,--output: 指定导出的tar文件名
```

```
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
159a08c11fec        centos              "/bin/bash"         2 ho
[root@rsync131 ~]# docker export -o centos_test.tar.gz 159a08c11
```

或者

```
[root@rsync131 ~]# docker export 159a08c11fec > centos_test.tar.
[root@rsync131 ~]# ll
总用量 273536
-rw-------. 1 root root      1513 8月  20 20:25 anaconda-ks.cfg
-rw-------  1 root root 280093696 9月  23 11:51 centos_test.tar.
```

5.2. 导入容器

使用命令 docker import 导入变成镜像

语法格式:

```
docker import [-c|--change[=[]]] [-m|--messge[=MESSAGE]] file|UR
```

用户可以通过-c 选项在导入的同时执行对容器进行修改的Dockerfile指令

```
[root@rsync131 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREA
[root@rsync131 ~]# ll
总用量 273536
-rw-------. 1 root root      1513 8月  20 20:25 anaconda-ks.cfg
-rw-r--r--  1 root root 280093696 9月  23 11:52 centos_test.tar.
[root@rsync131 ~]# docker import centos_test.tar.gz centos:1.0
sha256:f54ee7febeda5d3d8af11cc4756c1e6661feb84b2af6280654ef07ddf
[root@rsync131 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
centos              1.0                 f54ee7febeda        5 se
docker.io/centos    latest              5182e96772bf        6 we
```

[TOC]

一、什么是仓库

仓库就是集中存放镜像的地方，分为公共仓库和私有仓库，一个容易混淆的概念是注册服务器，实际上注册服务器是存放仓库的具体服务器，一个注册服务器可以有多个仓库，而每个仓库下面有多个镜像

二、使用 Docker 官方公共仓库

官方仓库地址:DockerHub

2.1. 登录仓库

使用 docker login 命令来登录官方的公共仓库

```
[root@rsync131 ~]# docker login
Login with your Docker ID to push and pull images from Docker Hu
Username: brianzjz
Password:
Login Succeeded

没有用户名密码的可以去官方注册一个
```

2.2. 下载镜像

使用 docker pull 命令下载镜像

```
[root@rsync131 ~]# docker pull nginx
Using default tag: latest
Trying to pull repository docker.io/library/nginx ...
latest: Pulling from docker.io/library/nginx
802b00ed6f79: Pull complete
e9d0e0ea682b: Pull complete
d8b7092b9221: Pull complete
Digest: sha256:24a0c4b4a4c0eb97a1aabb8e29f18e917d05abfe1b7a7c078
Status: Downloaded newer image for docker.io/nginx:latest
```

2.3. 推送镜像

使用 docker push 命令推送镜像

```
[root@rsync131 ~]# docker push brianzjz/nginx:v1.0
The push refers to a repository [docker.io/brianzjz/nginx]
579c75bb43c0: Mounted from library/nginx
67d3ae5dfa34: Mounted from library/nginx
8b15606a9e3e: Mounted from library/nginx
v1.0: digest: sha256:c0b69559d28fb325a64c6c8f47d14c26b95aa047312
```

三、国内第三方公共仓库

使用国内的仓库就是为了能提高下载的速度：想要达到火箭的速度请点击：火箭加速器

四、搭建自己的本地仓库

这个私有仓库才是重点，在我们的企业中很多都会是自己去定制自己的镜像文件，有一些还必须不能对外开放的，也是能够为了提高在内网的下载速度，这样我们就要搭建自己的私有仓库了

4.1. 使用 registry 镜像创建私有仓库

安装 docker 后，可以通过官方提供的 registry 镜像来简单的搭建一套本地的私有仓库

```
[root@rsync131 ~]# mkdir -p /opt/data/docker/
[root@rsync131 ~]# docker run -it -d -p 5000:5000 -v /opt/data/d
Unable to find image 'registry:latest' locally
Trying to pull repository docker.io/library/registry ...
latest: Pulling from docker.io/library/registry
d6a5679aa3cf: Pull complete
ad0eac849f8f: Pull complete
2261ba058a15: Pull complete
f296fda86f10: Pull complete
bcd4a541795b: Pull complete
Digest: sha256:5a156ff125e5a12ac7fdec2b90b7e2ae5120fa249cf622483
Status: Downloaded newer image for docker.io/registry:latest
6df20cdacf8d4a40ad3cbd3d310299650d41d0757b535130a0daa19f21ee1901
[root@rsync131 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND
6df20cdacf8d        registry            "/entrypoint.sh /e..."
```

4.2. 管理私有仓库

用另一台机器来测试上传下载私有仓库

```
[root@nfs133 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
[root@nfs133 ~]# docker pull nginx
Using default tag: latest
Trying to pull repository docker.io/library/nginx ...
latest: Pulling from docker.io/library/nginx
802b00ed6f79: Pull complete
e9d0e0ea682b: Pull complete
d8b7092b9221: Pull complete
Digest: sha256:24a0c4b4a4c0eb97a1aabb8e29f18e917d05abfe1b7a7c078
Status: Downloaded newer image for docker.io/nginx:latest
[root@nfs133 ~]# docker tag nginx 192.168.73.131:5000/nginx_test
[root@nfs133 ~]# docker images
REPOSITORY                        TAG                 IMAGE ID
192.168.73.131:5000/nginx_test    latest              06144b28784
docker.io/nginx                   latest              06144b28784
================================================
[root@nfs133 ~]# docker push 192.168.73.131:5000/nginx_test
The push refers to a repository [192.168.73.131:5000/nginx_test]
Get https://192.168.73.131:5000/v1/_ping: http: server gave HTTP
```

这里执行的时候报错了，原因就是新版本的docker对安全性要求高了，会要求仓

解决方法：
修改daemon配置文件

```
[root@nfs133 ~]# vim /etc/sysconfig/docker
OPTIONS='--selinux-enabled --log-driver=journald --signature-ver
```

修改为：

```
OPTIONS='--selinux-enabled --log-driver=journald --signature-ver
```

保存后，重启docker

```
[root@nfs133 ~]# systemctl restart docker

================================================
# 重新执行push
[root@nfs133 ~]# docker push 192.168.73.131:5000/nginx_test
The push refers to a repository [192.168.73.131:5000/nginx_test]
579c75bb43c0: Pushed
67d3ae5dfa34: Pushed
8b15606a9e3e: Pushed
latest: digest: sha256:c0b69559d28fb325a64c6c8f47d14c26b95aa0473


# curl 测试
```

```
# 温馨提示：下面的执行是错的 因为registry现在已经是2.0版本了
[root@nfs133 ~]# curl http://192.168.73.131:5000/v1/search
404 page not found

# 正确的方式
[root@nfs133 ~]# curl -X GET http://192.168.73.131:5000/v2/_cata
{"repositories":["nginx_test"]}

# 下载私有仓库镜像
[root@nfs133 ~]# docker pull 192.168.73.131:5000/nginx_test
Using default tag: latest
Trying to pull repository 192.168.73.131:5000/nginx_test ...
latest: Pulling from 192.168.73.131:5000/nginx_test
Digest: sha256:c0b69559d28fb325a64c6c8f47d14c26b95aa047312b29c69
Status: Downloaded newer image for 192.168.73.131:5000/nginx_tes
[root@nfs133 ~]# docker images
REPOSITORY                         TAG                 IMAGE ID
192.168.73.131:5000/nginx_test     latest              06144b28784
docker.io/nginx                    latest              06144b28784
```

[TOC]

一、什么是数据卷

生成环境中使用 docker 的过程中，往往需要对数据进行持久化，或者需要多个容器之间进行数据共享，这个就涉及到了容器数据管理

容器中管理数据主要有两种方式：

1. 数据卷：容器内数据之间映射到本地主机环境
2. 数据卷容器：使用特定的容器来维护数据卷

数据卷是一个可供容器使用的特殊目录，它将主机操作系统目录直接映射进容器，类似 Linux 的 mount 挂载

数据卷的特性

1. 数据卷可以在容器之间共享和重用，容器间传递数据将变得高效方便
2. 对数据卷内数据的修改立马生效，无论是容器内还是本地操作
3. 对数据卷的更新不会影响镜像，解耦了应用和数据
4. 卷会一直存在，直到没有容器使用，可以安全的卸载它

二、创建数据卷

在用 docker run 命令的时候，使用 -v 标记可以在容器内创建一个数据卷。多次重复使用 -v 标记可以创建多个数据卷

```
[root@rsync131 ~]# docker run -it -P -d --name web -v /opt/web/
b0772ffc49ea226536a23ca5d73f1a69c4212407ad54ab406b1b098138e64a65
```

PS：-P 参数是将容器服务暴露的端口，是自动映射到本地主机的临时端口

也可以挂载主机目录做我数据卷 (经常使用的方式)

```
[root@rsync131 ~]# docker run -it -P -d --name web1 -v /opt/web:
acb1b0f0c804ca2b9bbe8c1fe0373c6025c51342e138204e40e74ac9123c8854
```

上面的命令是将主机的 / opt/web 目录挂载到容器的 / opt / 下，这样是很方便的在你进行数据修改的是，直接修改本地文件就行了，容器直接就会同步了

docker 挂载数据卷的默认权限是读写 (rw) , 我们也可以改权限

```
[root@rsync131 ~]# docker run -it -P -d --name web2 -v /opt/web:
8400a6999dbafe14dd6cb28796dc84595282361e619a8d183db405106e07d85d
```

加了: ro 容器内对所挂载的数据卷内的数据就不能修改了

也可以只挂载本地主机的单个文件到容器中作为数据卷 (强烈不推荐，生产也很少用，可忽略)

```
[root@rsync131 ~]# docker run -it -P -d --name web4 -v /root/.ba
b5a45865eca2ad16cde207c9a35702667f3bd9b947383a33451b67098548331e
```

三、数据卷容器

如果用户需要在多个容器之间共享一些持续更新的数据，最简单的方式是使用数据卷容器，数据卷容器也是个容器，但是它的目的是专门用来提供数据卷供其他容器挂载的

首先创建一个数据卷容器

```
[root@rsync131 /]# docker run -it -d -v /dbdata --name dbdata ce
ea8c0b5014166f57afc5d7cf52b1ca532e0e0a259f414d2891d0ebf9dc397624
[root@rsync131 /]# docker exec -it ea8c0b5014166f57afc5d7cf52b1c
[root@ea8c0b501416 /]# ls
anaconda-post.log  bin  dbdata  dev  etc  home  lib  lib64  medi
```

挂载

```
[root@rsync131 /]# docker run -it -d --volumes-from dbdata --nam
ad8b9458a39c85516f2dc88645af8f3725418d5f4bf058566b0cbc977352490c
[root@rsync131 /]# docker run -it -d --volumes-from dbdata --nam
b07899fb1a46aa5538fc82dceebc4e53d7abf74ab870e14f916a59a03d1fab45
```

此时，容器 db1 和 db2 都挂载同一个数据卷到相同的 / dbdata 目录，三个容器任何一方在该目录下进行操作，其他的容器都能看见

测试

```
# db1 容器
[root@rsync131 /]# docker exec -it db1 bash
[root@ad8b9458a39c /]# cd /
[root@ad8b9458a39c /]# ls
anaconda-post.log  bin  dbdata  dev  etc  home  lib  lib64  medi
[root@ad8b9458a39c /]# cd dbdata/
[root@ad8b9458a39c dbdata]# ls
[root@ad8b9458a39c dbdata]# touch  a.txt
[root@ad8b9458a39c dbdata]# ls
a.txt
[root@ad8b9458a39c dbdata]# exit
exit

# db2 容器
[root@rsync131 /]# docker exec -it db2 bash
[root@b07899fb1a46 /]# cd /d
dbdata/ dev/
[root@b07899fb1a46 /]# cd /dbdata/
[root@b07899fb1a46 dbdata]# ls
a.txt
```

其实在有容器也可以对 db1 或者 db2 进行挂载

```
[root@rsync131 /]# docker run -it -d --volumes-from db1 --name d
95547b8422226a2f18dd71db4e8c551fd8df72771cf6001cfdfb963899b71477
[root@rsync131 /]# docker exec -it db3 bash
[root@95547b842222 /]# ls /dbdata/
a.txt
```

--volumes-from 参数所挂载数据卷的容器自身并不需要保持正在运行

如果删除了挂载的容器 (包括 dbdata、db1 和 db2)，数据卷并不会被自动的删除，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

四、利用数据卷容器来迁移数据

4.1. 备份

使用下面的命令来备份 dbdata 数据卷容器内的数据卷

```
[root@rsync131 ~]# docker run --volumes-from dbdata -v $(pwd):/b
/dbdata/
/dbdata/a.txt
[root@rsync131 ~]# ll
总用量 273540
-rw-------. 1 root root      1513 8月  20 20:25 anaconda-ks.cfg
-rw-r--r--  1 root root       142 9月  23 14:13 backup.tar.gz

# 具体的意思是：利用centos镜像创建一个容器，使用--volumes-from dbdat
```

4.2. 恢复

为了恢复我们创建一个容器

```
[root@rsync131 ~]# docker run -it -d -v /dbdata --name db8 cento
042d3c9567154d047977e8097939215bef441d3206c81c6bca99b6d227501169
[root@rsync131 ~]# docker run --volumes-from db8 -v $(pwd):/back
dbdata/
dbdata/a.txt
[root@rsync131 ~]# docker exec -it db8 bash
[root@042d3c956715 /]# ls
anaconda-post.log  bin  dbdata  dev  etc  home  lib  lib64  medi
[root@042d3c956715 /]# cd dbdata/
[root@042d3c956715 dbdata]# ls
a.txt
```

[TOC]

一、容器端口映射

1.1. 外部访问容器

在启动容器时候，如果不指定参数，在容器外部是无法通过网络来访问容器内的服务的

当容器运行一些网络服务的时候，我们可以通过指定 - p 或者 - P 参数来实现能够让外部访问的效果

1. -P(大 P)： Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口
2. -p(小 p)： 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

```
[root@rsync131 ~]# docker run -it -P -d --name web -v /opt/web/
[root@rsync131 ~]# docker run -it -d -p 5000:5000 -v /opt/data/d
```

1.2. 映射到指定地址的指定端口

可以使用 IP:HostPort:ContainerPort 格式指定映射使用一个特定的地址

```
[root@rsync131 ~]# docker run -it -d -p 127.0.0.1:2000:2000 cent
```

还可以绑定 udp 端口

```
[root@rsync131 ~]# docker run -it -d -p 127.0.0.1:2000:2000/udp
```

1.3. 映射到指定地址的任意端口

使用 IP::ContainerPort 绑定 IP 的任意端口到容器的 2000 端口，本地主机会自动的分配端口

```
[root@rsync131 ~]# docker run -it -d -p 127.0.0.1::2000 centos
```

1.4. 查看映射端口的配置

使用 docker port 命令来查看当前映射的端口配置

```
[root@rsync131 ~]# docker port 6df20cdacf8d
5000/tcp -> 0.0.0.0:5000
```

容器有自己内部的 IP 和网络，可以使用 docker inspect + 容器名或者容器 ID 查看具体的信息

二、容器与容器直接实现互联

2.1. 自定义容器名

容器的连接系统是根据容器的名字来执行的，所以要先给容器起一个好记得名字

在使用 docker run 创建容器的是 指定 --name 来定义容器的名字

```
[root@rsync131 ~]# docker run -it -P -d --name web1 -v /opt/web:
```

可以使用 docker inspect 来看容器的名字

```
[root@rsync131 ~]# docker inspect -f  042d3c956715
/db8
```

2.2. 容器互联

使用 --link 参数可以实现容器之间的安全交互

```
[root@rsync131 ~]# docker run -it -d -P --name link --link db1:d
```

[TOC]

一、什么是 dockerfile

Dockerfile 是一个文本格式的配置文件，用户可以使用 Dockerfile 自定义快速创建属于自己的镜像，Dockerfile 是通过很多的参数指令编写的文件，通过 docker build 命令来创建镜像

二、基本语法和结构

Dockerfile 由一行行的命令语句组成，并且支持以 #号注释

一般情况，Dockerfile 分为四部分：

1. 基础镜像信息
2. 维护者信息
3. 镜像操作指令
4. 容器启动执行的指令

其中，一开始必须指定所基于的镜像信息名称，接下来一般是说明的维护者信息，后面则是镜像的操作指令，例如 RUN 指令，RUN 指令将对镜像执行跟随的命令，每运行一条 RUN 指令，镜像就会添加新的一层，并提交，最后是 CMD 指令，用来指定运行容器时操作的指令

三、参数指令说明

指令	说明
FROM	指定创建镜像的基础镜像
MAINTAINER	指定维护者信息
RUN	运行命令
CMD	指定启动容器时默认执行的命令
LABEL	指定生成镜像的元数据标签信息
EXPOSE	声明镜像内服务所监听的端口
ENV	指定环境变量
ADD	复制指定的 <src> 路径下的内容到容器中的 < dest > 下，<src > 可以为 URL，如果是 tar 文件，会自动解压到 < dest > 路径下
COPY	复制本地主机的 <src> 路径下的内容到镜像中的 < dest > 路径下，一般情况下这个常用
ENTRYPOINT	指定镜像的默认入口
VOLUME	创建数据卷挂载点
USER	指定运行容器的用户名或者 UID
WORKDIR	配置工作目录
ARG	指定镜像内使用的参数 (例如版本号等信息)
ONBUILD	配置当所创建的镜像作为其他镜像的基础镜像时，所执行的创建操作命令
STOPSIGNAL	容器退出的信号值
HEALTHCHECK	如何进行健康检查
SHELL	指定使用 shell 时默认 shell 类型

3.1. FROM (小写 from)

指定所创建的镜像的基础镜像，如果不存在，会去DockerHub去下载

格式：
FROM
或者
FROM：或者
FROM@ 任何Dockerfile中的第一条指令必须为FROM指令，并且如果在同一个Do

3.2. MAINTAINER (小写 maintainer)

指定维护者信息

格式：
MAINTAINER 该信息会写入到生成镜像的Author属性域中

例如：

```
MAINTAINER zhujingzhi@126.com
```

3.3. RUN (小写 run)

运行指令命令

格式：
RUN
或者
RUN ["executable","paraml","param2"]
注意后一个指令会被解析为JSON的数组，因此必须要用双引号

前者默认将在shell终端中运行命令，即/bin/sh -c
后者则是用exec执行，不会启动shell环境

指定使用其他的终端类型可以用第二种方式，例如 RUN ["/bin/sh","-c","ec

每条RUN指令将在当前镜像的基础上执行指定命令，并提交为新的镜像，命令长的

例如：

```
RUN yum update \
    && yum -y install net-tools openssh openssl \
    && rm -rf /var/log/a.log
```

3.4. CMD (小写 cmd)

```
CMD 指令用来指定启动容器时默认执行的命令，有三种格式：
1、CMD ["executable","param1","param2"] 使用exec执行，推荐使用

2、CMD command param1 parma2 在/bin/sh中执行，提供给要交互的应用

3、CMD ["parma1","parma2"] 提供给ENTRYPOINT 的默认参数

每个Dockerfile只有一条CMD命令，如果指定了多条命令，只有最后一条会执行
```

3.5. LABEL (小写 label)

```
LABEL 指令是用来指定生成镜像的元数据标签

格式：
LABEL = = = .......
```

例如：

```
LABEL version="1.0"
LABEL description="Zhu Jingzhi's mirror image"
```

3.6. EXPOSE (小写 expose)

```
EXPOSE 声明镜像内服务监听的端口

格式：
EXPOSE [......]

注意，该指定是声明的作用，不会自动的完成端口的映射
在启动容器的时候需要使用-P 或 -p 来自动分配一个临时端口或者指定具体的
```

例如：

```
EXPOSE 22 443 80
```

3.7. ENV (小写 env)

```
ENV 指定环境变量，在镜像生成的过程中会被后续的RUN 使用，在镜像启动的容

格式:
ENV 或者
ENV=.......

注意在 指令指定的环境变量在运行时可以被覆盖掉
如:
docker run --env = centos
```

例如:

```
ENV PY_VERSION 3.6.1
RUN curl -sSL http://python.org/ftp/python/3.6.1/Python-$PY_VERS
ENV PATH /usr/src/python=$PY_VERSION/bin:$PATH
```

3.8. ADD (小写 add)

```
该命令将复制指定的路径下的内容到容器中的路径下

格式:
ADD 其中可以是Dockerfile所在目录的一个相对路径(文件或者目录),也可以是

路径支持正则格式
例如:
ADD *.tar /code/

tar压缩包用这个还是很方便的
```

3.9. COPY (小写 copy)

```
COPY 复制本地主机的(为Dockerfile所在目录的相对路径、文件或者目录)下的

格式:
COPY 路径同样支持正则
当使用本地目录为源目录的时候，非常推荐使用户CMD
```

例如:

```
COPY /opt/data/ /opt/
```

3.10. ENTRYPOINT (小写 entrypoint)

```
ENTRYPOINT 指定镜像的默认入口，该入口命令会在启动容器时作为根命令执行

格式:
ENTRYPOINT ["executable","param1","param2"]
或者
ENTRYPOINT command param1 param2

此时，CMD指令指定的值将作为根命令的参数
每个Dockerfile 中只能有一个ENTRYPOINT 当指定多个的时候，只有最后一个

在运行时，可以被--entrypoint参数覆盖
```

3.11. VOLUME (小写 volume)

```
VOLUME 创建一个挂载点

格式:
VOLUME ["/data"]

可以从本地主机或者其他容器挂载数据卷，一般用来存放数据库和需要保存的数
```

3.12. USER (小写 user)

```
USER 指定运行容器时的用户名或UID ，后续的RUN指令也是使用指定的用户

格式:
USER daemon

当服务不需要管理员权限的时候，可以使用该命令指定运行用户，并且可以在之
```

```
要临时获取管理员权限可以是用sudo
```

例如:

```
RUN groupadd -r nginx && useradd -r -g nginx nginx
```

3.13. WORKDIR (小写 workdir)

```
WORKDIR 为后续的RUN CMD ENTRYPOINT 指令配置工作目录

格式:
WORKDIR /path/to/workdir

可以使用多个WORKDIR指令，后续命令如果参数是相对路径，则会基于之前的命
```

例如:

```
WORKDIR /a
WORKDIR b
WORKDIR c

最终的路径是 /a/b/c
```

3.14. ARG (小写 arg)

```
ARG 指定一些镜像内使用的参数(例如版本信息)

格式:
ARG=[=]

也可以用docker build --build-arg=来进行指定参数值
```

3.15. ONBUILD (小写 onbuild)

```
ONBUILD 配置当所创建的镜像作为其他镜像的基础镜像时，所执行的创建操作指

格式：
ONBUILD [INSTRYCTION]
```

例如创建一个镜像 A：

```
[......]
ONBUILD ADD ./app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[......]
```

如果基于镜像 A 创建新的镜像，新的 Dockerfile 中使用 FROM 镜像 A 指定基础镜像，会自动执行镜像 A 中的 ONBUILD 指令的内容，等价在后面添加了两条指令

```
FROM 镜像A

# 等价于：
ADD . /app/src
RUN /usr/local/bin/python-build --dir /app/src
```

3.16. STOPSIGNAL (小写 stopsignal)

```
STOPSIGNAL 指定所创建的镜像启动的容器接收的退出的信号值

例如：
STOPSIGNAL signal
```

3.17. HEALTHCHECK (小写 healthcheck)

```
HEALTHCHECK 配置所启动容器如何进行健康检查，Docker1.12 才开始支持

两种格式：
1、HEALTHCHECK [OPTIONS] CMD command
# 根据所执行命令的返回值是否为0来判断

OPTIONS支持的参数：
--interval=DURATION(默认30s) ：过多久检查一次
--timeout=DURATION(默认30s) ：  每次检查的超时时间
--retries=N(默认为：3) ：  如果失败了重试的次数

2、HEALTHCHECK NONE
# 禁止基础镜像的中的健康检查
```

3.18. SHELL (小写 shell)

```
SHELL 指定其他命令使用shell时默认shell类型

格式：
SHELL ["executable","parameters"]

默认值为["/bin/sh","-c"]
```

四、创建镜像

创建完 dockerfile 文件后 可以使用 docker build 命令来创建镜像

基本的格式：

docker build [选项] Dockerfile 路径

该命令会读取指定路径下 (包括子目录) 的 Dockerfile，并将该路径下的所有的内容发给 docker 服务端，由服务端来创建镜像，因此建议除非生成镜像需要，否则一般吧 Dockerfile 放到一个空的目录中

两点经验：

1. 如果使用非内容路径下的 Dockerfile，可以通过 -f 参数来指定路径
2. 要指定生成镜像的标签信息，可以使用 -t 参数

例如：

```
指定Dockerfile所在的路径为/opt/docker_builder,并且希望生成镜像的标签

[root@nfs133 ~]# docker build -t zhujingzhi/nginx1.8.1 /root/doc

如果是在Dockerfile的目录下执行就是

[root@nfs133 ~]# docker build -t zhujingzhi/nginx1.8.1 .   # 一定
```

五、Dockerfile 实战文件

说了一堆的每个参数的语法格式，下面来做个实战的 dockerfile 文件，来生成一个镜像，并使用这个镜像创建个容器，并运行起来，我们来使用 nginx 服务来做实战

5.1. 下载基础镜像

```
[root@rsync131 ~]# docker pull centos
Using default tag: latest
Trying to pull repository docker.io/library/centos ...
latest: Pulling from docker.io/library/centos
256b176beaff: Pull complete
Digest: sha256:6f6d986d425aeabdc3a02cb61c02abb2e78e57357e92417d6
Status: Downloaded newer image for docker.io/centos:latest
```

5.2. 编写 Dockerfile 文件

```
# 创建存储Dockerfile文件的目录
[root@rsync131 ~]# mkdir docker_builder
[root@rsync131 ~]# cd /root/docker_builder/
# 下载需要的包
[root@rsync131 docker_builder]# wget http://nginx.org/download/n
[root@rsync131 docker_builder]# wget http://dl.fedoraproject.org
```

编写 Dockerfile 文件

```
[root@rsync131 ~]# cd /root/docker_builder/
[root@rsync131 docker_builder]# vim Dockerfile

# This my first nginx Dockerfile
# Version 1.0

# Base images 基础镜像
FROM centos

#MAINTAINER 维护者信息
MAINTAINER zhujingzhi

#ENV 设置环境变量
ENV PATH /usr/local/nginx/sbin:$PATH

#ADD  文件放在当前目录下，拷过去会自动解压
ADD nginx-1.8.1.tar.gz /usr/local/
ADD epel-release-7-11.noarch.rpm /usr/local/


#RUN 执行以下命令
RUN rpm -ivh /usr/local/epel-release-7-11.noarch.rpm
RUN yum install -y wget lftp gcc gcc-c++ make openssl-devel pcre
RUN useradd -s /sbin/nologin -M nginx

#WORKDIR 相当于cd
WORKDIR /usr/local/nginx-1.8.1

RUN ./configure --prefix=/usr/local/nginx --user=nginx --group=n

RUN ln -s /usr/local/nginx/sbin/* /usr/local/sbin/

#EXPOSE 映射端口
EXPOSE 80

#CMD 运行以下命令
CMD ["nginx","-g","daemon off;"]
```

5.3. 构建镜像并启动容器

构建镜像

```
[root@rsync131 docker_builder]# cd /root/docker_builder/
[root@rsync131 docker_builder]# docker build -t zhujingzhi/nginx
```

过程就不粘贴了 因为太多了 大家执行自己看一下吧，会有Dockerfile的每一步

查看

```
[root@rsync131 docker_builder]# docker images
REPOSITORY              TAG                 IMAGE ID
zhujingzhi/nginx1.8.1   latest              236535a1cdd2
docker.io/registry      latest              2e2f252f3c88
docker.io/centos        latest              5182e96772bf
```

已经构建好了镜像

启动容器

```
[root@rsync131 docker_builder]# docker run -itd --name nginx1 -p
8e23f4f849a33515c27e0bad92ff29442b7b2822be30dc235f30bf200d663f64

[root@rsync131 docker_builder]# docker ps
CONTAINER ID        IMAGE               COMMAND
8e23f4f849a3        236535a1cdd2        "nginx -g 'daemon ..."
```

5.4. 访问测试

```
[root@rsync131 docker_builder]# curl 127.0.0.1

Welcome to nginx!

    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }

Welcome to nginx!
=================

If you see this page, the nginx web server is successfully insta
working. Further configuration is required.



For online documentation and support please refer to
[nginx.org](http://nginx.org/).
Commercial support is available at
[nginx.com](http://nginx.com/).



_Thank you for using nginx._
```

[TOC]

一、什么是 Docker Compose

Compose 项目是 Docker 官方的开源项目，负责实现 Docker 容器集群的快速编排，开源代码在 https://github.com/docker/compose 上

我们知道使用 Dockerfile 模板文件可以让用户很方便的定义一个单独的应用容器，其实在工作中，经常会碰到需要多个容器相互配合来完成的某项任务情况，例如工作中的 web 服务容器本身，往往会在后端加上数据库容器，甚至会有负责均衡器，比如 LNMP 服务

Compose 就是来做这个事情的，它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)

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 中定义

二、基本原理

Compose 项目是由 Python 编写的，实际上就是调用了 Docker 服务提供的 API 来对容器进行管理，因此，只要所在的操作系统的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理.

三、安装

3.1. 二进制包安装

```
[root@operation ~]# curl -L https://github.com/docker/compose/re
% Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent
100   617    0   617    0     0    373      0 --:--:--  0:00:01
100 11.1M  100 11.1M    0     0   368k      0  0:00:31  0:00:31
[root@operation ~]# chmod +x /usr/local/bin/docker-compose
[root@operation ~]# docker-compose version
docker-compose version 1.23.0-rc2, build 350a555e
docker-py version: 3.5.0
CPython version: 3.6.6
OpenSSL version: OpenSSL 1.1.0f  25 May 2017
```

3.2. pip 安装 (安装的是最新稳定版本)

```
Compose 既然是用python编写的那么肯定是可以用pip install 进行安装的
[root@operation ~]# pip install docker-compose
# 安装完需要做个软链接
[root@operation ~]# ln -s /usr/bin/docker-compose /usr/local/bin
[root@operation ~]# docker-compose version
docker-compose version 1.22.0, build f46880f
docker-py version: 3.5.0
CPython version: 2.7.5
OpenSSL version: OpenSSL 1.0.1e-fips 11 Feb 2013
```

3.3. 容器安装

```
[root@operation ~]# curl -L https://github.com/docker/compose/re
% Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent
100   596    0   596    0     0    158      0 --:--:--  0:00:03
100  1670  100  1670    0     0    343      0  0:00:04  0:00:04
[root@operation ~]# chmod +x /usr/local/bin/docker-compose
[root@operation ~]# docker-compose
Unable to find image 'docker/compose:1.23.0-rc2' locally
1.23.0-rc2: Pulling from docker/compose
3489d1c4660e: Pull complete
2e51ed086e7d: Pull complete
07d7b41c67a1: Pull complete
Digest: sha256:14f5ad3c2162b26b3eaafe870822598f80b03ec36fd451269
# 实际上就是下的镜像(可以看下下载的run.sh脚本)
[root@operation ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
docker/compose      1.23.0-rc2          dc59a0b5e981        5 da
alpine              latest              196d12cf6ab1        4 we
[root@operation ~]# docker-compose version
docker-compose version 1.23.0-rc2, build 350a555e
docker-py version: 3.5.0
CPython version: 3.6.6
OpenSSL version: OpenSSL 1.1.0f  25 May 2017
```

四、命令

```
Compose 大部分命令的对象即可以是项目的本身，也可以是指定为项目中的服务
执行docker-compose [COMMAND] --help 或者docker-compose help [COMM
具体的使用格式
docker-compose [-f=...] [options] [COMMAND] [ARGS]
参数选项
-f,--file file指定模板文件，默认是docker-compose.yml模板文件,可以多
-p,--project-name name指定项目名称，默认使用所在目录名称作为项目名称
--x-networking 使用Docker的后端可插拔网络特性
--x-networking-driver driver指定网络的后端驱动，默认使用bridge
--verbose 输入更多的调试信息
-v,--version 输出版本信息
```

```
Compose所支持的命令
build              Build or rebuild services (构建项目中的服务容器
bundle             Generate a Docker bundle from the Compose fil
config             Validate and view the Compose file (验证并查看
create             Create services (为服务创建容器)
down               Stop and remove containers, networks, images,
events             Receive real time events from containers (为项
exec               Execute a command in a running container (这相
help               Get help on a command (获得一个命令的帮助)
images             List images ()
kill               Kill containers (通过发送SIGKILL信号来强制停止服
logs               View output from containers (查看服务容器的输出
pause              Pause services (暂停一个容器)
port               Print the public port for a port binding (打印
ps                 List containers (列出项目中目前所有的容器)
pull               Pull service images (拉取服务依赖镜像)
push               Push service images (推送服务镜像)
restart            Restart services (重启项目中的服务)
rm                 Remove stopped containers (删除所有停止状态的服
run                Run a one-off command (在指定服务上执行一个命令
scale              Set number of containers for a service (设置指
start              Start services (启动已存在的服务容器)
stop               Stop services (停止已存在的服务容器)
top                Display the running processes (显示容器正在运行
unpause            Unpause services (恢复处于暂停状态的容器)
up                 Create and start containers (自动完成包括构建镜
version            Show the Docker-Compose version information (
```

这些命令的使用方法可以使用执行 docker-compose [COMMAND] --help 或者 docker-compose help [COMMAND] 可以查看命令的帮助信息

因为太多的原因我这里就不写了，有时间我在把每个命令的使用方法补上吧

针对模板文件的使用才是重中之重，我在后面会对模板文件详细讲解

官方链接：https://docs.docker.com/compose/reference/build/

五、环境变量

```
环境变量可以用来配置 Compose 的行为,以DOCKER_开头的变量和用来配置 Do

COMPOSE_PROJECT_NAME    设置通过 Compose 启动的每一个容器前添加的项

COMPOSE_FILE            设置要使用的 docker-compose.yml 的路径。默

DOCKER_HOST             设置 Docker daemon 的地址。默认使用 unix:/

DOCKER_TLS_VERIFY       如果设置不为空，则与 Docker daemon 交互通过

DOCKER_CERT_PATH        配置 TLS 通信所需要的验证（ca.pem、cert.pe
```

在使用的时候在做解释和操作吧，因为一般不会改环境变量的东西，默认的就 OK，做个简单的了解

官方链接：https://docs.docker.com/compose/reference/envvars/#compose_project_name

六、模板文件

模板文件时 Compose 的核心，涉及的指令关键字比较多，但是大部分的指令与 docker run 相关的参数的含义是类似的

默认的模板名是 docker-compose.yml

官网链接：https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples

Compose和Docker兼容性:

Compose 文件格式有3个版本,分别为1, 2.x 和 3.x

目前主流的为 3.x 其支持 docker 1.13.0 及其以上的版本

常用参数:

```
version           # 指定 compose 文件的版本
services          # 定义所有的 service 信息, services 下面的第

    build                 # 指定包含构建上下文的路径, 或作为一个
        context               # context: 指定 Dockerfile 文件
        dockerfile            # dockerfile: 指定 context 指定
        args                  # args: Dockerfile 在 build 过
        cache_from            # v3.2中新增的参数, 指定缓存的镜
        labels                # v3.3中新增的参数, 设置镜像的元
        shm_size              # v3.5中新增的参数, 设置容器 /de

    command               # 覆盖容器启动后默认执行的命令, 支持

    configs               # 不知道怎么用

    cgroup_parent         # 不知道怎么用

    container_name        # 指定容器的名称 (等同于 docker run

    credential_spec       # 不知道怎么用

    deploy                # v3 版本以上, 指定与部署和运行服务相
        endpoint_mode         # v3.3 版本中新增的功能, 指定服务
            vip                   # Docker 为该服务分配了一个虚
            dnsrr                 # DNS轮询, Docker 为该服务设
        labels                # 指定服务的标签, 这些标签仅在服务
        mode                  # 指定 deploy 的模式
            global                # 每个集群节点都只有一个容器
            replicated            # 用户可以指定集群中容器的数
        placement             # 不知道怎么用
        replicas              # deploy 的 mode 为 replicated
        resources             # 资源限制
            limits                # 设置容器的资源限制
                cpus: "0.5"           # 设置该容器最多只能使用
                memory: 50M           # 设置该容器最多只能使用
            reservations          # 设置为容器预留的系统资源(随
                cpus: "0.2"           # 为该容器保留 20% 的 CP
```

```
                memory: 20M             # 为该容器保留 20M 的内存
        restart_policy          # 定义容器重启策略，用于代替 restart 命令
            condition               # 定义容器重启策略(接受三个参数)
                none                    # 不尝试重启
                on-failure              # 只有当容器内部应用程序出现问题才会重启
                any                     # 无论如何都会尝试重启(默认)
            delay                   # 尝试重启的间隔时间(默认为 0)
            max_attempts            # 尝试重启次数(默认一直尝试重启)
            window                  # 检查重启是否成功之前的等待时间
        update_config           # 用于配置滚动更新配置
            parallelism             # 一次性更新的容器数量
            delay                   # 更新一组容器之间的间隔时间
            failure_action          # 定义更新失败的策略
                continue                # 继续更新
                rollback                # 回滚更新
                pause                   # 暂停更新(默认)
            monitor                 # 每次更新后的持续时间以监视更新是否失败
            max_failure_ratio       # 回滚期间容忍的失败率(默认值)
            order                   # v3.4 版本中新增的参数，回滚期间的操作顺序
                stop-first              #旧任务在启动新任务之前停止
                start-first             #首先启动新任务，并且正在运行
        rollback_config         # v3.7 版本中新增的参数，用于定义回滚策略
            parallelism             # 一次回滚的容器数，如果设置为0，则所有容器将同时回滚
            delay                   # 每个组回滚之间的时间间隔(默认为0)
            failure_action          # 定义回滚失败的策略
                continue                # 继续回滚
                pause                   # 暂停回滚
            monitor                 # 每次回滚任务后的持续时间以监视失败
            max_failure_ratio       # 回滚期间容忍的失败率(默认值)
            order                   # 回滚期间的操作顺序
                stop-first              # 旧任务在启动新任务之前停止
                start-first             # 首先启动新任务，并且正在运行

        注意:
            支持 docker-compose up 和 docker-compose run 但不支持在 Swarm 环境中使用
            security_opt  container_name  devices  tmpfs  stop_signal
            network_mode  external_links  restart  build  user

    devices                 # 指定设备映射列表（等同于 docker run --device）

    depends_on              # 定义容器启动顺序（此选项解决了容器之间的依赖关系）
        示例:
            docker-compose up 以依赖顺序启动服务，下面例子中 redis 和 db 服务在 web 服务之前启动
            默认情况下使用 docker-compose up web 这样的方式启动 web 服务时，也会启动 redis 和 db 两个服务
```

```
        version: '3'
        services:
            web:
                build: .
                depends_on:
                    - db
                    - redis
            redis:
                image: redis
            db:
                image: postgres

dns                     # 设置 DNS 地址(等同于 docker run -

dns_search              # 设置 DNS 搜索域(等同于 docker run

tmpfs                   # v2 版本以上，挂载目录到容器中，作为

entrypoint              # 覆盖容器的默认 entrypoint 指令 (等

env_file                # 从指定文件中读取变量设置为容器中的环
    文件格式:
        RACK_ENV=development

environment             # 设置环境变量，  environment 的值可以

expose                  # 暴露端口，但是不能和宿主机建立映射关

external_links          # 连接不在 docker-compose.yml 中定义

extra_hosts             # 添加 host 记录到容器中的 /etc/host

healthcheck             # v2.1 以上版本，定义容器健康状态检查
    test                    # 检查容器检查状态的命令，该选项必
        NONE                    # 禁用容器的健康状态检测
        CMD                     # test: ["CMD", "curl", "-
        CMD-SHELL               # test: ["CMD-SHELL", "cur
    interval: 1m30s         # 每次检查之间的间隔时间
    timeout: 10s            # 运行命令的超时时间
    retries: 3              # 重试次数
    start_period: 40s       # v3.4 以上新增的选项，定义容器启
    disable: true           # true 或 false，表示是否禁用健康
```

```
image                 # 指定 docker 镜像，可以是远程仓库镜

init                  # v3.7 中新增的参数，true 或 false

isolation             # 隔离容器技术，在 Linux 中仅支持 de

labels                # 使用 Docker 标签将元数据添加到容器

links                 # 链接到其它服务中的容器，该选项是 do

logging               # 设置容器日志服务
    driver                # 指定日志记录驱动程序，默认 jso
    options               # 指定日志的相关参数（等同于 doc
        max-size              # 设置单个日志文件的大小，当
        max-file              # 日志文件保留的数量

network_mode          # 指定网络模式（等同于 docker run --

networks              # 将容器加入指定网络（等同于 docker
    aliases               # 同一网络上的容器可以使用服务名称
    ipv4_address      # IP V4 格式
    ipv6_address      # IP V6 格式

    示例:
        version: '3.7'
        services:
            test:
                image: nginx:1.14-alpine
                container_name: mynginx
                command: ifconfig
                networks:
                    app_net:
                    ipv4_address: 172.16.238.10
        networks:
            app_net:
                driver: bridge
                ipam:
                    driver: default
                    config:
                        - subnet: 172.16.238.0/24

pid: 'host'           # 共享宿主机的 进程空间(PID)

ports                 # 建立宿主机和容器之间的端口映射关系，
```

```
    SHORT 语法格式示例:
        - "3000"                                # 暴露容器的
        - "3000-3005"                           # 暴露容器的
        - "8000:8000"                           # 容器的 8000
        - "9090-9091:8080-8081"
        - "127.0.0.1:8001:8001"                 # 指定映射宿主
        - "127.0.0.1:5000-5010:5000-5010"
        - "6060:6060/udp"                       # 指定协议

    LONG 语法格式示例:(v3.2 新增的语法格式)
        ports:
            - target: 80                        # 容器端口
              published: 8080                   # 宿主机端口
              protocol: tcp                     # 协议类型
              mode: host                        # host 在每个

secrets             # 不知道怎么用

security_opt        # 为每个容器覆盖默认的标签（在使用 sw

stop_grace_period   # 指定在发送了 SIGTERM 信号之后，容器

stop_signal         # 指定停止容器发送的信号（默认为 SIGT

sysctls             # 设置容器中的内核参数（在使用 swarm

ulimits             # 设置容器的 limit

userns_mode         # 如果Docker守护程序配置了用户名称空间

volumes             # 定义容器和宿主机的卷映射关系，其和
    SHORT 语法格式示例:
        volumes:
            - /var/lib/mysql                    # 映射容器内的
            - /opt/data:/var/lib/mysql          # 映射容器内的
            - ./cache:/tmp/cache                # 映射容器内的
            - ~/configs:/etc/configs/:ro        # 映射容器宿主
            - datavolume:/var/lib/mysql         # datavolume

    LONG 语法格式示例:(v3.2 新增的语法格式)
        version: "3.2"
        services:
            web:
                image: nginx:alpine
```

```
                    ports:
                        - "80:80"
                    volumes:
                        - type: volume                  # mo
                            source: mydata              # 宿
                            target: /data               # 容
                            volume:                     # 配
                                nocopy: true
                        - type: bind                    # vo
                            source: ./static
                            target: /opt/app/static
                            read_only: true             # 设
            volumes:
                mydata:                                 # 定

restart                 # 定义容器重启策略(在使用 swarm 部署
    no                    # 禁止自动重启容器(默认)
    always                # 无论如何容器都会重启
    on-failure            # 当出现 on-failure 报错时，容器
```

其他选项：

```
domainname, hostname, ipc, mac_address, privileged,
```

上面这些选项都只接受单个值和 docker run 的对应参数类似

对于值为时间的可接受的值：

```
2.5s
10s
1m30s
2h32m
5h34m56s
```

时间单位：us，ms，s，m， h

对于值为大小的可接受的值：

```
2b
1024kb
2048k
300m
1gb
```

单位：b，k，m，g 或者 kb，mb，gb

```
networks          # 定义 networks 信息
    driver                # 指定网络模式，大多数情况下，它 bri
        bridge                # Docker 默认使用 bridge 连接单
        overlay               # overlay 驱动程序创建一个跨多个
        host                  # 共享主机网络名称空间(等同于 do
        none                  # 等同于 docker run --net=none

    driver_opts           # v3.2以上版本，传递给驱动程序的参数，

    attachable            # driver 为 overlay 时使用，如果设置

    ipam                  # 自定义 IPAM 配置. 这是一个具有多个
        driver                # IPAM 驱动程序，bridge 或者 de
        config                # 配置项
            subnet                # CIDR格式的子网，表示该网络

    external              # 外部网络，如果设置为 true 则 docke

    name                  # v3.5 以上版本，为此网络设置名称
```

文件格式示例：

```
version: "3"
services:

  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - frontend
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure

  db:
    image: postgres:9.4
```

```
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]
```

七、Compose 使用

举个简单的例子来具有的说明一下 Compose 的使用 (也是官网的一个入门小例子)

先决条件：确保您已经安装了 Docker Engine 和 Docker Compose。您不需要安装 Python 或 Redis，因为两者都是由 Docker 镜像提供的。

7.1. 创建一个目录 (里面包含需要的文件)

```
# 创建目录
[root@operation ~]# mkdir composetest
[root@operation ~]# cd composetest/

# 创建一个Python应用， 使用Flask，将数值记入Redis
[root@operation composetest]# cat app.py
import time

import redis
from flask import Flask


app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)


def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)


@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(co

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)


# 创建requirements.txt文件，里面是需要安装的Python包
[root@operation composetest]# cat requirements.txt
flask
redis

# 创建Dockerfile文件
```

```
# 在此步骤中，您将编写一个构建Docker镜像的Dockerfile。该图像包含Pyth
[root@operation composetest]# cat Dockerfile
FROM python:3.6-alpine
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]

# 这告诉Docker：
    从Python 3.6映像开始构建映像。
    将当前目录添加.到/code映像中的路径中。
    将工作目录设置为/code。
    安装Python依赖项。
    将容器的默认命令设置为python app.py。

# 创建docker-compose.yml文件
[root@operation composetest]# cat docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"

此Compose文件定义了两个服务，web和redis。该web服务：
使用从Dockerfile当前目录中构建的图像。
将容器上的公开端口5000转发到主机上的端口5000。我们使用Flask Web服务器
该redis服务使用从Docker Hub注册表中提取的公共 Redis映像。
```

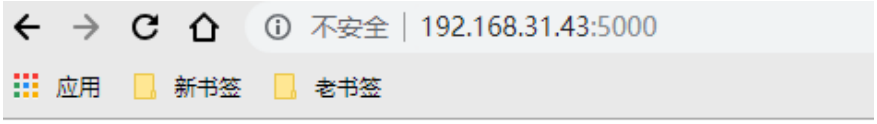
7.2. 使用 Compose 构建并运行您的应用程序

```
[root@operation composetest]# docker-compose up
# 出现下面说明成功了
redis_1_bfd9eb391c58 | 1:M 14 Oct 08:29:53.581 * Ready to accept
web_1_6f42e21c34dd |  * Serving Flask app "app" (lazy loading)
web_1_6f42e21c34dd |  * Environment: production
web_1_6f42e21c34dd |    WARNING: Do not use the development serv
web_1_6f42e21c34dd |    Use a production WSGI server instead.
web_1_6f42e21c34dd |  * Debug mode: on
web_1_6f42e21c34dd |  * Running on http://0.0.0.0:5000/ (Press C
web_1_6f42e21c34dd |  * Restarting with stat
web_1_6f42e21c34dd |  * Debugger is active!
web_1_6f42e21c34dd |  * Debugger PIN: 160-344-502
```

7.3. 测试访问

在浏览器访问 IP:5000 我这里是 192.168.31.43:5000

每刷新一次就会加一



Hello World! I have been seen 6 times.

[TOC]

一、什么是 Docker Machine

Docker Machine 是 Docker 官方编排项目之一，使用 go 语言编写的，使用不同引擎在多种平台上快速的安装 Docker 环境，开源地址:https://github.com/docker/machine。

Docker Machine 是一个工具，它允许你在虚拟宿主机上安装 Docker，并使用 docker-machine 命令管理这个宿主机，可以使用 Docker Machine 在本地的 MAC 或者 windows box、公司网络，数据中心或者 AWS 这样的云提供商上创建 docker。

使用 docker-machine 命令，可以启动、审查、停止、重启托管的 docker 也可以升级 Docker 客户端和守护程序并配置 docker 客户端和宿主机通信。

Docker Machine 也可以集中管理所以得 docker 主机。

二、为什么要使用 Docker Machine

Docker Machine 使你能够在各种 Linux 上配置多个远程 Docker 宿主机。此外，Machine 允许你在较早的 Mac 或 Windows 系统上运行 Docker，如上一主题所述。
Docker Machine 有这两个广泛的用例。

- 我有一个较旧的桌面系统，并希望在 Mac 或 Windows 上运行 Docker

如果你主要在不符合新的 Docker for Mac 和 Docker for Windows 应用程序的旧 Mac 或 Windows 笔记本电脑或台式机上工作，则需要 Docker Machine 来在本地 “运行 Docker”（即 Docker Engine）。在 Mac 或 Windows box 中使用 Docker Toolbox 安装程序安装 Docker Machine 将为 Docker Engine 配置一个本地的虚拟机，使你能够连接它、并运行 docker 命令。

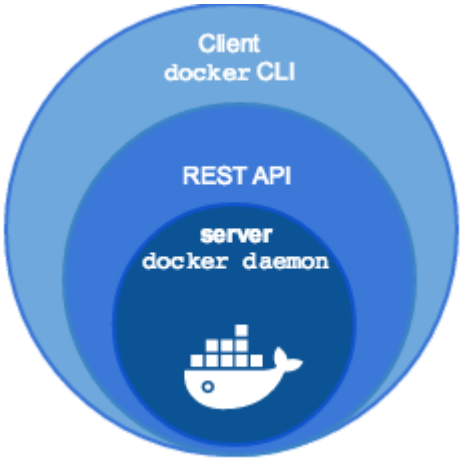
- 我想在远程系统上配置 Docker 宿主机

Docker Engine Linux 系统上原生地运行。如果你有一个 Linux 作为你的主系统，并且想要运行 docker 命令，所有你需要做的就是下载并安装 Docker Engine 。然而，如果你想要在网络上、云中甚至本地配置多个 Docker 宿主机有一个有效的方式，你需要 Docker Machine。

无论你的主系统是 Mac、Windows 还是 Linux，你都可以在其上安装 Docker Machine，并使用 docker-machine 命令来配置和管理大量的 Docker 宿主机。它会自动创建宿主机、在其上安装 Docker Engine 、然后配置 docker 客户端。每个被管理的宿主机（“machine”）是 Docker 宿主机和配置好的客户端的结合。

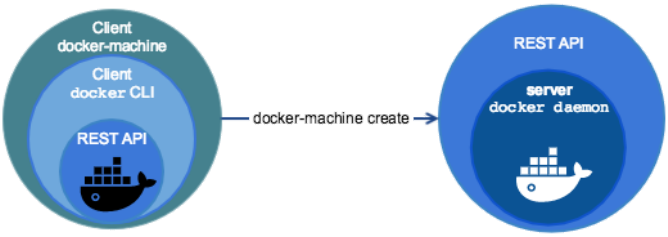
三、Docker 和 Docker Machine 之间的区别

当人们说 “Docker” 时，他们通常是指 Docker Engine，它是一个客户端 - 服务器应用程序，由 Docker 守护进程、一个 REST API 指定与守护进程交互的接口、和一个命令行接口（CLI）与守护进程通信（通过封装 REST API）。Docker Engine 从 CLI 中接受 docker 命令，例如 docker run 、docker ps 来列出正在运行的容器、docker images 来列出镜像，等等。



Docker Machine 是一个用于配置和管理你的宿主机（上面具有 Docker Engine 的主机）的工具。通常，你在你的本地系统上安装 Docker Machine。Docker Machine 有自己的命令行客户端 docker-machine 和 Docker Engine 客户端 docker。你可以使用 Machine 在一个或多个虚拟系统上安装 Docker Engine。

这些虚拟系统可以是本地的（就像你在 Mac 或 Windows 上使用 Machine 在 VirtualBox 中安装和运行 Docker Engine 一样）或远程的（就像你使用 Machine 在云提供商上 provision Dockerized 宿主机一样）。Dockerized 宿主机本身可以认为是，且有时就称为，被管理的 “machines”。



四、安装

Docker Mechine 可以在多种平台上安装使用，包括 Linux 、MacOS 已经 windows

Docker Mechine 安装非常的简单：GitHub 地址:https://github.com/docker/machine/releases/ 里面有安装教程 (在写这篇文章的时候最新版本是 v0.15.0)

安装 Docker Mechine

```
[root@operation ~]# curl -L https://github.com/docker/machine/re
  % Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent
100   617    0   617    0     0    462      0 --:--:--  0:00:01
100 26.8M  100 26.8M    0     0  1000k      0  0:00:27  0:00:27
[root@operation ~]# chmod +x /tmp/docker-machine
[root@operation ~]# cp /tmp/docker-machine /usr/local/bin/docker

# 查看版本确认是否安装成功
[root@operation ~]# docker-machine -v
docker-machine version 0.15.0, build b48dc28d

# 安装自动补全功能
[root@operation ~]# yum -y install bash-completion
[root@operation ~]# scripts=( docker-machine-prompt.bash docker-

# 添加以下
[root@operation ~]# cat ~/.bashrc
# .bashrc

# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

source /etc/bash_completion.d/docker-machine-wrapper.bash
source /etc/bash_completion.d/docker-machine-prompt.bash
source /etc/bash_completion.d/docker-machine.bash

PS1='[\u@\h \W$(__docker_machine_ps1)]\$ '

# 使之生效
[root@operation ~]# source ~/.bashrc
```

到此位置 docker-machine 就安装完成了！

五、参数

支持命令

命令	说明
active	查看当前激活状态的 Docker 主机
config	查看当前激活状态 Docker 主机的连接信息
creat	创建 Docker 主机
env	显示连接到某个主机需要的环境变量
inspect	以 json 格式输出指定 Docker 的详细信息
ip	获取指定 Docker 主机的地址
kill	直接杀死指定的 Docker 主机
ls	列出所有的管理主机
provision	重新配置指定主机
regenerate-certs	为某个主机重新生成 TLS 信息
restart	重启指定的主机
rm	删除某台 Docker 主机，对应的虚拟机也会被删除
ssh	通过 SSH 连接到主机上，执行命令
scp	在 Docker 主机之间以及 Docker 主机和本地主机之间通过 scp 远程复制数据
mount	使用 SSHFS 从计算机装载或卸载目录
start	启动一个指定的 Docker 主机，如果对象是个虚拟机，该虚拟机将被启动
status	获取指定 Docker 主机的状态 (包括：Running、Paused、Saved、Stopped、Stopping、Starting、Error) 等
stop	停止一个指定的 Docker 主机
upgrade	将一个指定主机的 Docker 版本更新为最新
url	获取指定 Docker 主机的监听 URL
version	显示 Docker Machine 的版本或者主机 Docker 版本
help	显示帮助信息

支持的平台及驱动引擎

```
# 平台
1.常规Linux操作系统;

2.虚拟化平台-VirtualBox, VMware, Hyper-V

3.Openstack

4.公有云-Amazon Web Services, Microsoft Azure, Google Compute Eng

Docker Machine为这些环境起了一个统一的名字：provider

对于特定的某个provider, Docker Machine使用相应的driver安装配置docke

# 驱动引擎
amazonec2
azure
digitalocean
exoscale
generic
google
hyperv
none
openstack
rackspace
softlayer
virtualbox
vmwarevcloudair
vmwarefusion
vmwarevsphere

# 指定方式
使用参数 -d 或者 --driver 驱动引擎名称
```

说白了都是虚拟化平台和云平台的驱动文件

六、使用

通过 Docker Machine 创建 docker(我用了两台机器)

- 192.168.31.43 安装 docker machine 的机器 主机名：operation
- 192.168.31.188 被管理的机器 主机名：client1

创建步骤：

1. 配置主机间的 SSH 免密 (在 192.168.31.43 上面创建)

```
# 生成keys并配置可以免密登录主机(这个是必须要做的)
[root@operation ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:0Fq7VlowSsDqSStOn4veIoTxFbW2RB059qXMSzLblKg root@operatio
The key's randomart image is:
+---[RSA 2048]----+
|   ...o..o       |
|    oo o=   .    |
|   . .*.=* +     |
|. o .+ *++O      |
|.= +  +.SBo.     |
|oo=   E .=o      |
|+.. .   +        |
|.ooo   .         |
|oo.o.            |
+----[SHA256]-----+

# 将keys拷贝到client1上去
[root@operation ~]# ssh-copy-id root@192.168.31.188
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/
The authenticity of host '192.168.31.188 (192.168.31.188)' can't
ECDSA key fingerprint is SHA256:6MKhx743bCMD3Ay+ELNpKnq1+3/wltcr
ECDSA key fingerprint is MD5:e8:6d:14:7e:41:da:96:4b:2c:92:f8:61
Are you sure you want to continue connecting (yes/no)? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new ke
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- i
root@192.168.31.188's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'root@192.168.31.
and check to make sure that only the key(s) you wanted were adde

# 测试是否可以免密登录
[root@operation ~]# ssh root@192.168.31.188
```

```
Last login: Fri Oct 12 15:27:45 2018 from 192.168.31.104
[root@client1 ~]# exit
```

2. 使用 docker machine 创建 docker host

```
# 使用docker machine 创建
# 对于docker machine来将，术语Machine就是运行docker daemon的主机，包
# 执行docker-macine ls查看当前的machine
[root@operation ~]# docker-machine ls
NAME    ACTIVE   DRIVER   STATE   URL   SWARM   DOCKER   ERRORS

# 当前还没有一个machine，接下来我们创建第一个machine：docker188-192.
[root@operation ~]# docker-machine create --driver generic --gen
Running pre-create checks...
Creating machine...
(docker188) No SSH key specified. Assuming an existing key at th
Waiting for machine to be running, this may take a few minutes..
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with centos...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine ru
```

注意：这里会出现Error creating machine: Error running provisioning

```
# 创建成功执行ls查看
[root@operation ~]# docker-machine  ls
NAME        ACTIVE   DRIVER    STATE     URL
docker188   -        generic   Running   tcp://192.168.31.188:23

# 登录到client查看配置项
[root@operation ~]# ssh root@192.168.31.188
Last login: Fri Oct 12 16:19:10 2018 from 192.168.31.43
[root@docker188 ~]# cat /etc/systemd/system/docker.service.d/10-
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix:///var/
Environment=
```

注：-H tcp://0.0.0.0:2376 使docker daemon接受远程连接

--tls*对远程连接启用安全认证和加密

注：大家可能会发现这里的主机名变成了docker188 原因就是docker-machine

```
# 查看docker188的环境变量
[root@operation ~]# docker-machine env docker188
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.31.188:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/docker18
export DOCKER_MACHINE_
# Run this command to configure your shell:
# eval $(docker-machine env docker188)

# 根据提示执行
[root@operation ~]# eval $(docker-machine env docker188)
[root@operation ~ [docker188]]#
```

可以看到，命令提示符变成了docker188，其原因是我们之前在/root/.bashrc
注：如果我们输入eval $(docker-machine env docker1)没有显示出docker
在此状态下执行的docker命令其效果都相当于在docker188上执行

```
[root@operation ~ [docker188]]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREA
[root@operation ~ [docker188]]# docker images
REPOSITORY          TAG                 IMAGE ID            CREA
```

3. 其他命令操作

```
# 其他命令
# create 命令
选项包括:
---driver, -d"none"                                    指定驱动类型;
---engine-install-url"https://get.docker.com"          配置Dokcer主机时
---engine-opt option                                   以键值对格式指定所
---engine-insecure-registry option                     以键值对格式指定所
---engine-registry-mirror option                       指定使用注册仓库镜
---engine-label option                                 为所创建的Docker引
---engine-storage-driver                               存储后端驱动类型;
---engine-env option                                   指定环境变量;
---swarm                                               指定使用Swarm;
---swarm-image"swarm: latest"                          使用Swarm时候采用
---swarm-master                                        配置机器作为Swarm
---swarm-discovery                                     Swarm集群的服务发
---swarm-strategy"spread"                              Swarm默认调度策略
---swarm-opt option                                    任意传递给Swarm的
---swarm-host"tcp://0.0.0.0: 3376"                     指定地址将监听 Swa
---swarm-addr                                          从指定地址发送广播

实例:
docker-machine create -d virtualbox \
--engine-storage-driver overlay \
--engine-label name=testmachine \
--engine-label year=2018 \
--engine-opt dns=8.8.8.8 \
--engine-env HTTP_PROXY=http://proxy.com:3128 \
--engine-insecure-registry registry.private.com \
mydockermachine


# active命令
[root@operation ~]# docker-machine ls
NAME        ACTIVE   DRIVER    STATE     URL
docker188   -        generic   Running   tcp://192.168.31.188:23

# 这里的状态是没有被激活
[root@operation ~]# docker-machine env docker188
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.31.188:2376"
export DOCKER_CERT_PATH="/root/.docker/machine/machines/docker18
export DOCKER_MACHINE_
# Run this command to configure your shell:
```

```
# eval $(docker-machine env docker188)

[root@operation ~]# export DOCKER_HOST="tcp://192.168.31.188:237
[root@operation ~]# docker-machine ls
NAME        ACTIVE   DRIVER    STATE     URL
docker188   *        generic   Running   tcp://192.168.31.188:23
[root@operation ~]# docker-machine active
docker188

# config命令
[root@operation ~]# docker-machine config docker188
--tlsverify
--tlscacert="/root/.docker/machine/machines/docker188/ca.pem"
--tlscert="/root/.docker/machine/machines/docker188/cert.pem"
--tlskey="/root/.docker/machine/machines/docker188/key.pem"
-H=tcp://192.168.31.188:2376

# inspect命令
[root@operation ~]# docker-machine inspect docker188
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "192.168.31.188",
        "MachineName": "docker188",
        "SSHUser": "root",
        "SSHPort": 22,
        "SSHKeyPath": "",
        "StorePath": "/root/.docker/machine",
        "SwarmMaster": false,
        "SwarmHost": "",
        "SwarmDiscovery": "",
        "EnginePort": 2376,
        "SSHKey": ""
    },
    "DriverName": "generic",
    "HostOptions": {
        "Driver": "",
        "Memory": 0,
        "Disk": 0,
        "EngineOptions": {
            "ArbitraryFlags": [],
            "Dns": null,
            "GraphDir": "",
            "Env": [],
```

```
            "Ipv6": false,
            "InsecureRegistry": [],
            "Labels": [],
            "LogLevel": "",
            "StorageDriver": "",
            "SelinuxEnabled": false,
            "TlsVerify": true,
            "RegistryMirror": [],
            "InstallURL": "https://get.docker.com"
        },
        "SwarmOptions": {
            "IsSwarm": false,
            "Address": "",
            "Discovery": "",
            "Agent": false,
            "Master": false,
            "Host": "tcp://0.0.0.0:3376",
            "Image": "swarm:latest",
            "Strategy": "spread",
            "Heartbeat": 0,
            "Overcommit": 0,
            "ArbitraryFlags": [],
            "ArbitraryJoinFlags": [],
            "Env": null,
            "IsExperimental": false
        },
        "AuthOptions": {
            "CertDir": "/root/.docker/machine/certs",
            "CaCertPath": "/root/.docker/machine/certs/ca.pem",
            "CaPrivateKeyPath": "/root/.docker/machine/certs/ca-
            "CaCertRemotePath": "",
            "ServerCertPath": "/root/.docker/machine/machines/do
            "ServerKeyPath": "/root/.docker/machine/machines/doc
            "ClientKeyPath": "/root/.docker/machine/certs/key.pe
            "ServerCertRemotePath": "",
            "ServerKeyRemotePath": "",
            "ClientCertPath": "/root/.docker/machine/certs/cert.
            "ServerCertSANs": [],
            "StorePath": "/root/.docker/machine/machines/docker1
        }
    },
    "Name": "docker188"
}
```

```
# ssh命令
[root@operation ~]# docker-machine ssh docker188 docker images
REPOSITORY          TAG                 IMAGE ID            CREA
alpine              latest              196d12cf6ab1        4 we

[root@operation ~]# docker-machine ssh docker188
Last login: Fri Oct 12 16:36:49 2018 from 192.168.31.43
[root@docker188 ~]#

# url命令
[root@operation ~]# docker-machine url docker188
tcp://192.168.31.188:2376

# status命令
[root@operation ~]# docker-machine status docker188
Running

# version命令
[root@operation ~]# docker-machine version docker188
18.06.1-ce
```

注：还有一些命令就不一一列出了，可以查看上面的参数命令表，具体命令的使用方法可以通过 --help 查看

注：Machine 安装 docker 环境中会因网络或其他情况造成安装失败，使用中发现，这种安装失败频率很高，感觉没有使用的价值，说白了，一个公司操作系统一般不会超过两个发行版，写个脚本一键安装也许会更方便！

七、总结

Docker Machine 最主要有两个作用：

- 使用 Docker Machine 方便在不同的环境中使用 Docker ，比如：Win/Mac
- 使用 Docker Machine 方便在云环境下批量部署 Docker 环境，比如：私有云，公有云批量安装 Docker 环境

八、参考链接

官方地址：https://docs.docker.com/machine/

官方驱动详细使用方法：https://docs.docker.com/machine/drivers/

参考文档：https://www.cnblogs.com/lkun/p/7781157.html

[TOC]

一、什么是 Docker Swarm

Swarm 是 Docker 公司推出的用来管理 docker 集群的平台，几乎全部用 GO 语言来完成的开发的，代码开源在 https://github.com/docker/swarm， 它是将一群 Docker 宿主机变成一个单一的虚拟主机，Swarm 使用标准的 Docker API 接口作为其前端的访问入口，换言之，各种形式的 Docker

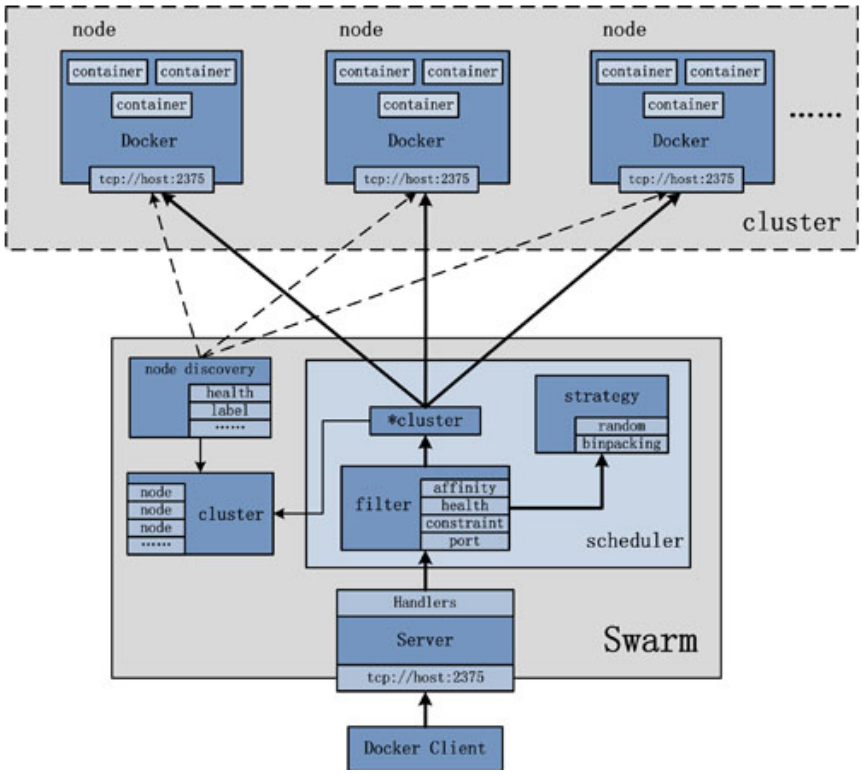
Client(compose,docker-py 等) 均可以直接与 Swarm 通信，甚至 Docker 本身都可以很容易的与 Swarm 集成，这大大方便了用户将原本基于单节点的系统移植到 Swarm 上，同时 Swarm 内置了对 Docker 网络插件的支持，用户也很容易的部署跨主机的容器集群服务。

Docker Swarm 和 Docker Compose 一样，都是 Docker 官方容器编排项目，但不同的是，Docker Compose 是一个在单个服务器或主机上创建多个容器的工具，而 Docker Swarm 则可以在多个服务器或主机上创建容器集群服务，对于微服务的部署，显然 Docker Swarm 会更加适合。

从 Docker 1.12.0 版本开始，Docker Swarm 已经包含在 Docker 引擎中（docker swarm），并且已经内置了服务发现工具，我们就不需要像之前一样，再配置 Etcd 或者 Consul 来进行服务发现配置了。

Swarm deamon 只是一个调度器 (Scheduler) 加路由器 (router),Swarm 自己不运行容器，它只是接受 Docker 客户端发来的请求，调度适合的节点来运行容器，这就意味着，即使 Swarm 由于某些原因挂掉了，集群中的节点也会照常运行，放 Swarm 重新恢复运行之后，他会收集重建集群信息。

二、Docker Swarm 基本结构图



在结构图可以看出 Docker Client 使用 Swarm 对 集群 (Cluster) 进行调度使用。

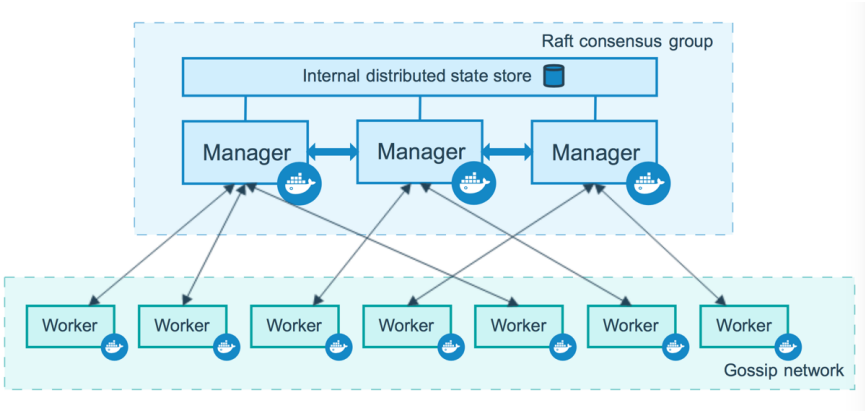
上图可以看出，Swarm 是典型的 master-slave 结构，通过发现服务来选举 manager。manager 是中心管理节点，各个 node 上运行 agent 接受 manager 的统一管理，集群会自动通过 Raft 协议分布式选举出 manager 节点，无需额外的发现服务支持，避免了单点的瓶颈问题，同时也内置了 DNS 的负载均衡和对外部负载均衡机制的集成支持

三. Swarm 的几个关键概念

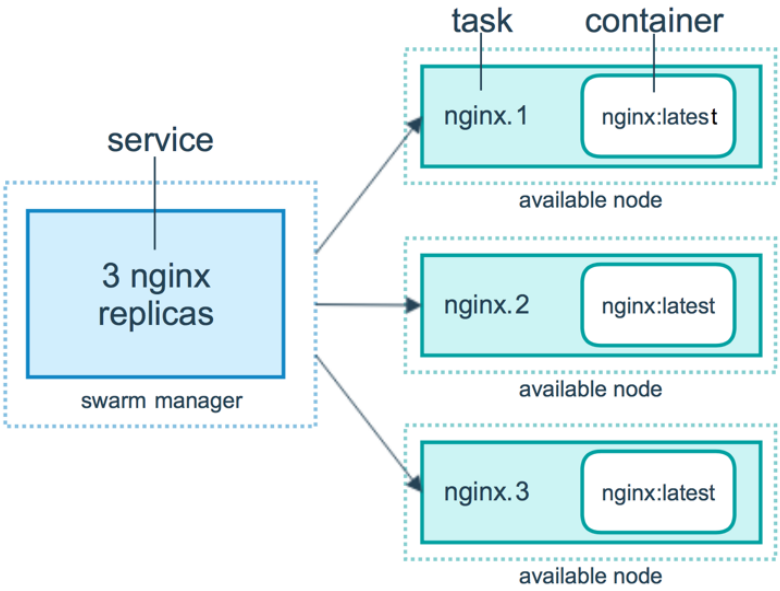
```
1.Swarm
集群的管理和编排是使用嵌入docker引擎的SwarmKit，可以在docker初始化时

2.Node
一个节点是docker引擎集群的一个实例。您还可以将其视为Docker节点。您可以
要将应用程序部署到swarm，请将服务定义提交给 管理器节点。管理器节点将称
Manager节点还执行维护所需群集状态所需的编排和集群管理功能。Manager节点
工作节点接收并执行从管理器节点分派的任务。默认情况下，管理器节点还将服

3.Service
一个服务是任务的定义，管理机或工作节点上执行。它是群体系统的中心结构，

4.Task
任务是在docekr容器中执行的命令，Manager节点根据指定数量的任务副本分配

----------------------------------------使用方法---------------
docker swarm：集群管理，子命令有init，join，leave，update。（docke
docker service：服务创建，子命令有create，inspect，update，remove，
docker node：节点管理，子命令有accept，promote，demote，inspect，u

node是加入到swarm集群中的一个docker引擎实体，可以在一台物理机上运行多
manager nodes，也就是管理节点
worker nodes，也就是工作节点

1）manager node管理节点：执行集群的管理功能，维护集群的状态，选举一个
2）worker node工作节点：接收和执行任务。参与容器集群负载调度，仅用于承
3）service服务：一个服务是工作节点上执行任务的定义。创建一个服务，指定
   service是运行在worker nodes上的task的描述，service的描述包括使用
4）task任务：一个任务包含了一个容器及其运行的命令。task是service的执行
```

四、Swarm 的工作模式

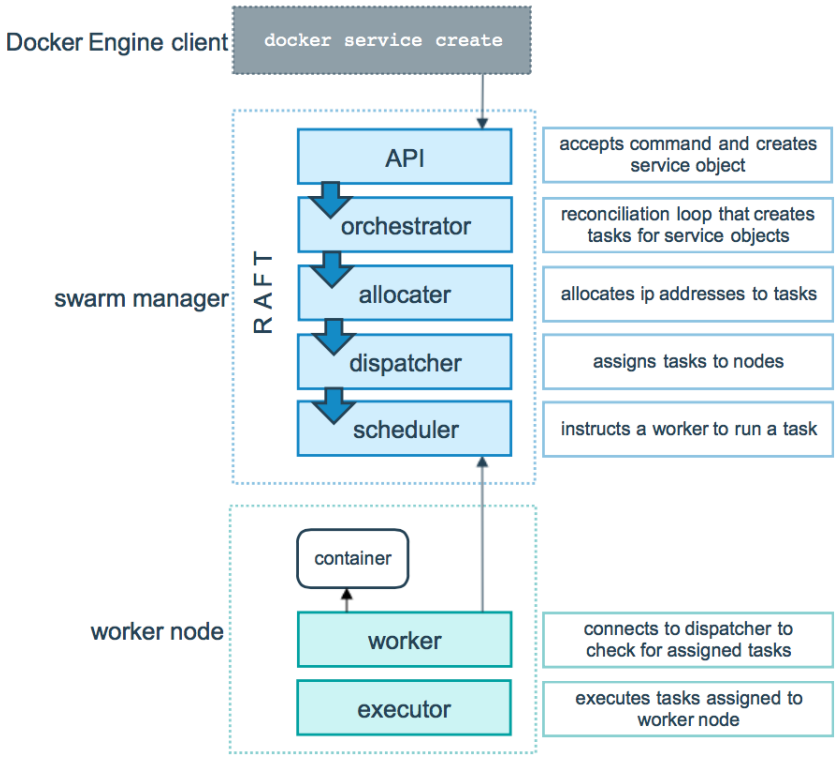
1. Node



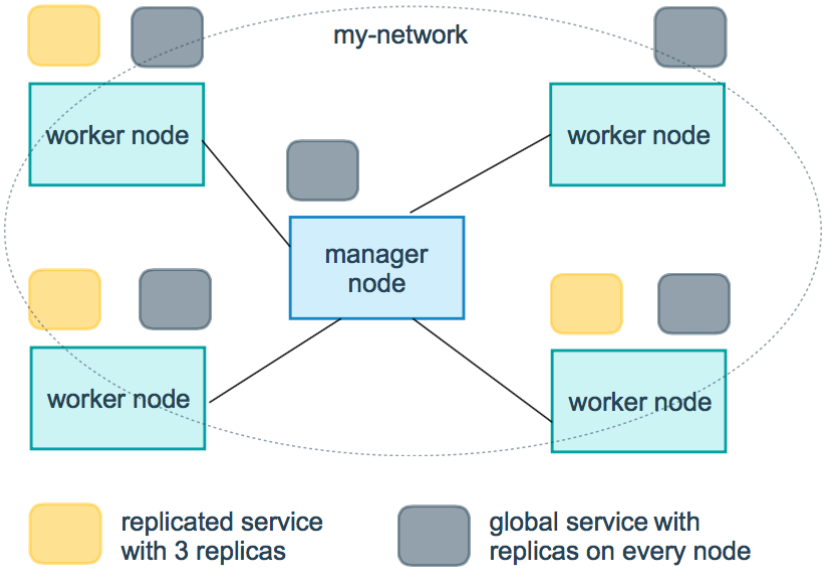
1. Service



1. 任务与调度



1. 服务副本与全局服务



五、Swarm 的调度策略

Swarm在调度(scheduler)节点（leader节点）运行容器的时候，会根据指定的

1）Random

顾名思义，就是随机选择一个Node来运行容器，一般用作调试用，spread和bin

行的容器的数量来计算应该运行容器的节点。

2）Spread

在同等条件下，Spread策略会选择运行容器最少的那台节点来运行新的容器，bi

使用Spread策略会使得容器会均衡的分布在集群中的各个节点上运行，一旦一个

3）Binpack

Binpack策略最大化的避免容器碎片化，就是说binpack策略尽可能的把还未使用

一个节点上面。

六、Swarm Cluster 模式特性

1）批量创建服务
建立容器之前先创建一个overlay的网络，用来保证在不同主机上的容器网络互

2）强大的集群的容错性
当容器副本中的其中某一个或某几个节点宕机后，cluster会根据自己的服务注
在集群中剩余的空闲节点上，重新拉起容器副本。整个副本迁移的过程无需人工
不难看出，docker service其实不仅仅是批量启动服务这么简单，而是在集群中
并维护集群的高可用性。

3）服务节点的可扩展性
Swarm Cluster不光只是提供了优秀的高可用性，同时也提供了节点弹性扩展或
参数即可复制出新的副本出来。

仔细观察的话，可以发现所有扩展出来的容器副本都run在原先的节点下面，如果
其实很简单，只需要在命令中将"--replicas n"更换成"--mode=global"即可

复制服务（--replicas n）
将一系列复制任务分发至各节点当中，具体取决于您所需要的设置状态，例如“-

全局服务（--mode=global）
适用于集群内全部可用节点上的服务任务，例如“--mode global”。如果大家在

4．调度机制
所谓的调度其主要功能是cluster的server端去选择在哪个服务器节点上创建并
组合而成。每次通过过滤器（constraint）启动容器的时候，swarm cluster

------------------Swarm cluster的创建过程包含以下三个步骤----------
1）发现Docker集群中的各个节点，收集节点状态、角色信息，并监视节点状态
2）初始化内部调度（scheduler）模块
3）创建并启动API监听服务模块

一旦创建好这个cluster，就可以用命令docker service批量对集群内的容器进

在启动容器后，docker 会根据当前每个swarm节点的负载判断，在负载最优的节
可以看到任务运行在哪个节点上。容器启动后，有时需要等待一段时间才能完成

七、Dcoker Swarm 集群部署

温馨提示：

机器环境 (三台机器，centos 系统)

IP：192.168.31.43 主机名：manager43 担任角色：swarm manager

IP：192.168.31.188 主机名：node188 担任角色：swarm node

IP：192.168.31.139 主机名：node139 担任角色：swarm node

1、准备工作

```
1）修改主机名
# 192.168.31.43  主机上执行
[root@manager43 ~]# hostnamectl set-hostname manager43

# 192.168.31.188 主机上执行
[root@node188 ~]# hostnamectl set-hostname node188

# 192.168.31.139 主机上执行
[root@node139 ~]# hostnamectl set-hostname node139

2)配置hosts文件(可配置可不配置)
[root@manager43 ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost
::1         localhost localhost.localdomain localhost6 localhost

192.168.31.43 manager43
192.168.31.188 node188
192.168.31.139 node139

# 使用scp复制到node主机
[root@manager43 ~]# scp /etc/hosts root@192.168.31.188:/etc/host
[root@manager43 ~]# scp /etc/hosts root@192.168.31.139:/etc/host

3）设置防火墙
关闭三台机器上的防火墙。如果开启防火墙，则需要在所有节点的防火墙上依次
[root@manager43 ~]# systemctl disable firewalld.service
[root@manager43 ~]# systemctl stop firewalld.service

4）安装docker并配置加速器(在三台主机都要安装哟...)
[root@manager43 ~]# yum -y install docker
[root@node188 ~]# yum -y install docker
[root@node139 ~]# yum -y install docker
```

也可以安装最新版 docker，可查考：docker 安装教程

加速器配置，可查考:docker 加速器配置教程

2、创建 Swarm 并添加节点

1）创建Swarm集群

```
[root@manager43 ~]# docker swarm init --advertise-addr 192.168.3
Swarm initialized: current node (z2n633mty5py7u9wyl423qnq0) is n

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

    # 这就是添加节点的方式(要保存初始化后token，因为在节点加入时要使用
    docker swarm join --token SWMTKN-1-2lefzq18zohy9yr1vskutf1sf

To add a manager to this swarm, run 'docker swarm join-token man
```

上面命令执行后，该机器自动加入到swarm集群。这个会创建一个集群token，获
其中，--advertise-addr参数表示其它swarm中的worker节点使用此ip地址与r

这里无意中遇到了一个小小的问题：

```
# 在次执行上面的命令，回报下面的错误
[root@manager43 ~]# docker swarm init --advertise-addr 192.168.3
Error response from daemon: This node is already part of a swarm
# 解决方法
[root@manager43 ~]# docker swarm leave -f
```

这里的leave就是在集群中删除节点，-f参数强制删除，执行完在重新执行OK

2）查看集群的相关信息

```
[root@manager43 ~]# docker info
```

上面的命令执行后 找到Swarm的关键字，就可以看到相关信息了

```
[root@manager43 ~]# docker node ls
ID                            HOSTNAME            STATUS
3jcmnzjh0e99ipgshk1ykuovd *   manager43           Ready
```

上面的命令是查看集群中的机器(注意上面node ID旁边那个*号表示现在连接到

3）添加节点主机到Swarm集群

上面我们在创建Swarm集群的时候就已经给出了添加节点的方法

```
# 192.168.31.188 主机上执行
[root@node188 ~]# docker swarm join --token SWMTKN-1-2lefzq18zoh
This node joined a swarm as a worker.

# 192.168.31.139 主机上执行
[root@node139 ~]# docker swarm join --token SWMTKN-1-2lefzq18zoh
This node joined a swarm as a worker.
```

如果想要将其他更多的节点添加到这个swarm集群中，添加方法如上一致

在manager43主机上我们可以看一下集群中的机器及状态

```
[root@manager43 ~]# docker node ls
ID                            HOSTNAME            STATUS
3jcmnzjh0e99ipgshk1ykuovd *   manager43           Ready
vww7ue2xprzg46bjx7afo4h04     node139             Ready
c5klw5ns4adcvumzgiv66xpyj     node188             Ready
```

温馨提示：更改节点的availablity状态

swarm集群中node的availability状态可以为 active或者drain，其中：

active状态下，node可以接受来自manager节点的任务分派；

drain状态下，node节点会结束task，且不再接受来自manager节点的任务分派

```
[root@manager43 ~]# docker node update --availability drain node
node139
[root@manager43 ~]# docker node ls
ID                            HOSTNAME            STATUS
3jcmnzjh0e99ipgshk1ykuovd *   manager43           Ready
vww7ue2xprzg46bjx7afo4h04     node139             Ready
c5klw5ns4adcvumzgiv66xpyj     node188             Ready
```

如上，当node1的状态改为drain后，那么该节点就不会接受task任务分发，就算

再次修改为active状态（及将下线的节点再次上线）

```
[root@manager43 ~]# docker node update --availability active nod
node139
[root@manager43 ~]# docker node ls
ID                            HOSTNAME            STATUS
3jcmnzjh0e99ipgshk1ykuovd *   manager43           Ready
vww7ue2xprzg46bjx7afo4h04     node139             Ready
c5klw5ns4adcvumzgiv66xpyj     node188             Ready
```

3、在 Swarm 中部署服务 (nginx 为例)

Docker 1.12版本提供服务的Scaling、health check、滚动升级等功能，并提

1）创建网络在部署服务

```
# 创建网络
[root@manager43 ~]# docker network create -d overlay nginx_net
a52jy33asc5o0ts0rq823bf0m
[root@manager43 ~]# docker network ls | grep nginx_net
a52jy33asc5o        nginx_net           overlay             swar

# 部署服务
[root@manager43 ~]# docker service create --replicas 1 --network
olexfmtdf94sxyeetkchwhehg
overall progress: 1 out of 1 tasks
1/1: running   [==================================================
verify: Service converged
```

在manager-node节点上使用上面这个覆盖网络创建nginx服务：

其中，--replicas 参数指定服务由几个实例组成。

注意：不需要提前在节点上下载nginx镜像，这个命令执行后会自动下载这个容器

```
# 使用 docker service ls 查看正在运行服务的列表
[root@manager43 ~]# docker service ls
ID                  NAME                MODE                REPL
olexfmtdf94s        my_nginx            replicated          1/1
```

2）查询Swarm中服务的信息

-pretty 使命令输出格式化为可读的格式，不加 --pretty 可以输出更详细的信

```
[root@manager43 ~]# docker service inspect --pretty my_nginx
ID:             zs7fw4ereo5w7ohd4n9ii06nt
Name:           my_nginx
Service Mode:   Replicated
 Replicas:      1
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
```

```
ContainerSpec:
 Image:         nginx:latest@sha256:b73f527d86e3461fd652f62cf47e
 Init:          false
Resources:
Networks: nginx_net
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress

# 查询到哪个节点正在运行该服务。如下该容器被调度到manager-node节点上
[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
```

温馨提示：如果上面命令执行后，上面的 STATE 字段中刚开始的服务状态为 Pr

有上面命令可知，该服务在manager-node节点上运行。登陆该节点，可以查看到

```
[root@manager43 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND
0dc7103f8030        nginx:latest        "nginx -g 'daemon of…"
```

3）在Swarm中动态扩展服务(scale)

当然，如果只是通过service启动容器，swarm也算不上什么新鲜东西了。Servi

比如将上面的my_nginx容器动态扩展到4个

```
[root@manager43 ~]# docker service scale my_nginx=4
my_nginx scaled to 4
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service converged
```

和创建服务一样，增加scale数之后，将会创建新的容器，这些新启动的容器也会

```
[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
mlprstt9ds5x        my_nginx.2          nginx:latest        node
y09lk90tdzdp        my_nginx.3          nginx:latest        node
clolfl3zlvj0        my_nginx.4          nginx:latest        node
```

可以看到，之前my_nginx容器只在manager-node节点上有一个实例，而现在又

这4个副本的my_nginx容器分别运行在这三个节点上，登陆这三个节点，就会发

4）模拟宕机node节点
特别需要清楚的一点：
如果一个节点宕机了（即该节点就会从swarm集群中被踢出），则Docker应该会

比如：
将node139宕机后或将node139的docker服务关闭，那么它上面的task实例就会
只能等别的节点出现故障后转移task实例到它的上面。使用命令"docker node

```
[root@node139 ~]# systemctl stop docker
[root@manager43 ~]# docker node ls
ID                            HOSTNAME            STATUS
ppk7q0bjond8a58xja7in1qid *   manager43           Ready
mums8azgbrffnecp3q8fz70pl     node139             Down
z3n36maf03yjg7odghikuv574     node188             Ready
```

然后过一会查询服务的状态列表

```
[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
wb1cpk9k22rl        my_nginx.2          nginx:latest        node
mlprstt9ds5x         \_ my_nginx.2      nginx:latest        node
rhbj4bcr4t2c        my_nginx.3          nginx:latest        mana
y09lk90tdzdp         \_ my_nginx.3      nginx:latest        node
clolfl3zlvj0        my_nginx.4          nginx:latest        node
```

上面我们可以发现node139故障后，它上面之前的两个task任务已经转移到node

登陆到node188和manager43节点上，可以看到这两个运行的task任务。当访问

```
[root@manager43 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
ae4c5c2e6f3f        nginx:latest        "nginx -g 'daemon of…"
0dc7103f8030        nginx:latest        "nginx -g 'daemon of…"

[root@node188 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
a63ef253f7dd        nginx:latest        "nginx -g 'daemon of…"
74a1a1db81d4        nginx:latest        "nginx -g 'daemon of…"
```

再次在node188和manager43节点上将从node139上转移过来的两个task关闭

```
[root@manager43 ~]# docker stop my_nginx.3.rhbj4bcr4t2c3y2f8vyfm
my_nginx.3.rhbj4bcr4t2c3y2f8vyfmbi21
```

```
[root@node188 ~]# docker stop my_nginx.2.wb1cpk9k22rl1ydab7aozl2
my_nginx.2.wb1cpk9k22rl1ydab7aozl2b5
```

再次查询服务的状态列表，发现这两个task又转移到node139上了

```
[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
j2q61f8jtzba        my_nginx.2          nginx:latest        node
wb1cpk9k22rl         \_ my_nginx.2      nginx:latest        node
mlprstt9ds5x         \_ my_nginx.2      nginx:latest        node
oz9wyjuldw1t        my_nginx.3          nginx:latest        mana
rhbj4bcr4t2c         \_ my_nginx.3      nginx:latest        mana
y09lk90tdzdp         \_ my_nginx.3      nginx:latest        node
clolfl3zlvj0        my_nginx.4          nginx:latest        node
```

结论：即在swarm cluster集群中启动的容器，在worker node节点上删除或停

5) Swarm 动态缩容服务(scale)

同理，swarm还可以缩容，同样是使用scale命令

如下，将my_nginx容器变为1个

```
[root@manager43 ~]# docker service scale my_nginx=1
my_nginx scaled to 1
overall progress: 1 out of 1 tasks
1/1:
verify: Service converged

[root@manager43 ~]# docker service ls
ID                  NAME                MODE                REPL
zs7fw4ereo5w        my_nginx            replicated          1/1

[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
wb1cpk9k22rl        my_nginx.2          nginx:latest        node
mlprstt9ds5x         \_ my_nginx.2      nginx:latest        node
rhbj4bcr4t2c        my_nginx.3          nginx:latest        mana
y09lk90tdzdp         \_ my_nginx.3      nginx:latest        node
```

通过docker service ps my_nginx 可以看到node节点上已经为Shutdown状态

在登录到node节点主机上查看

```
[root@node188 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
f93c0a27374a        nginx:latest        "nginx -g 'daemon of…"
```

```
a63ef253f7dd        nginx:latest        "nginx -g 'daemon of…"
[root@node139 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND
e8ac2e44f5c4        nginx:latest        "nginx -g 'daemon of…"
5b031aa5a2cc        nginx:latest        "nginx -g 'daemon of…"
```

登录node节点，使用docker ps -a 查看，会发现容器被stop而非rm

6）除了上面使用scale进行容器的扩容或缩容之外，还可以使用docker servi

```
[root@manager43 ~]# docker service update --replicas 3 my_nginx
my_nginx
overall progress: 3 out of 3 tasks
1/3: running   [==================================================
2/3: running   [==================================================
3/3: running   [==================================================
verify: Service converged

[root@manager43 ~]# docker service ls
ID                  NAME                MODE                REPL
zs7fw4ereo5w        my_nginx            replicated          3/3

[root@manager43 ~]# docker service ps my_nginx
ID                  NAME                IMAGE               NODE
yzonph0zu7km        my_nginx.1          nginx:latest        mana
j3hduzd9pret        my_nginx.2          nginx:latest        node
wb1cpk9k22rl         \_ my_nginx.2      nginx:latest        node
mlprstt9ds5x         \_ my_nginx.2      nginx:latest        node
gng96vc5vqpv        my_nginx.3          nginx:latest        node
rhbj4bcr4t2c         \_ my_nginx.3      nginx:latest        mana
y09lk90tdzdp         \_ my_nginx.3      nginx:latest        node
```

docker service update 命令，也可用于直接 升级 镜像等

```
[root@manager43 ~]# docker service update --image nginx:new my_n

[root@manager43 ~]# docker service ls
ID                  NAME                MODE                REPL
zs7fw4ereo5w        my_nginx            replicated          3/3
```

注意IMAGE列 变成了nginx:new

7）为了下面的直观显示，我这里把my_nginx服务直接删除了

```
[root@manager43 ~]# docker service rm my_nginx
```

这样就会把所有节点上的所有容器（task任务实例）全部删除了

4、Swarm 中使用 Volume(挂在目录，mount 命令)

1) 查看volume的帮助信息

```
[root@manager43 ~]# docker volume --help

Usage:  docker volume COMMAND

Manage volumes

Commands:
  create      Create a volume
  inspect     Display detailed information on one or more volume
  ls          List volumes
  prune       Remove all unused local volumes
  rm          Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a com
```

2) 创建一个volume

```
[root@manager43 ~]# docker volume create --name testvolume
testvolume

# 查看创建的volume
[root@manager43 ~]# docker volume ls
DRIVER              VOLUME NAME
local               testvolume

# 查看volume详情
[root@manager43 ~]# docker volume inspect testvolume
[
    {
        "CreatedAt": "2018-10-21T10:50:02+08:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/testvolume/_data"
        "Name": "testvolume",
        "Options": {},
        "Scope": "local"
    }
]
```

3) 创建新的服务并挂载testvolume(nginx为例)

```
[root@manager43 ~]# docker service create --replicas 3 --mount t
sh7wc8yzcvr0xaedo4tnraj7l
```

```
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>
2/3: running   [==================================================>
3/3: running   [==================================================>
verify: Service converged
```

温馨提示：
参数src写成source也可以；dst表示容器内的路径，也可以写成target

```
# 查看创建服务
[root@manager43 ~]# docker service ls
ID                  NAME                MODE                REPL
sh7wc8yzcvr0        test_nginx          replicated          3/3
[root@manager43 ~]# docker service ps test_nginx
ID                  NAME                IMAGE               NODE
m7m41kwt4q6w        test_nginx.1        nginx:latest        node
kayh81q1o1kx        test_nginx.2        nginx:latest        node
eq11v0rcwy38        test_nginx.3        nginx:latest        mana

# 查看有没有挂载成功(登录各个节点的容器看看有没有指定的目录并创建文件测
# 容器中操作
[root@manager43 ~]# docker exec -it 63451219cb4e /bin/bash
root@63451219cb4e:/# cd /zjz/
root@63451219cb4e:/zjz# ls
root@63451219cb4e:/zjz# echo "gen wo xue docker" > docker.txt
root@63451219cb4e:/zjz# ls
docker.txt
```

执行docker volume inspect testvolume 可以看到本地的路径(上面已经执行
本地路径：/var/lib/docker/volumes/testvolume/_data

```
[root@manager43 ~]# cd /var/lib/docker/volumes/testvolume/_data
[root@manager43 _data]# ls
docker.txt
[root@manager43 _data]# cat docker.txt
gen wo xue docker
```

还可以将node节点机上的volume数据目录做成软链接

```
[root@manager43 _data]# ln -s /var/lib/docker/volumes/testvolume
[root@manager43 _data]# cd /zjz/
[root@manager43 zjz]# ls
docker.txt
[root@manager43 zjz]# echo "123" > 1.txt
[root@manager43 zjz]# ll
总用量 8
```

```
-rw-r--r-- 1 root root  4 10月 21 11:04 1.txt
-rw-r--r-- 1 root root 18 10月 21 11:00 docker.txt

# 容器中查看
[root@manager43 zjz]# docker exec -it 63451219cb4e /bin/bash
root@63451219cb4e:/# cd /zjz/
root@63451219cb4e:/zjz# ls
1.txt  docker.txt
root@63451219cb4e:/zjz# cat 1.txt
123
root@63451219cb4e:/zjz# cat docker.txt
gen wo xue docker

# 还有一种挂载方式简单说一下吧，上面的会了下面的肯定简单
命令格式:
docker service create --mount type=bind,target=/container_data/,
其中，参数target表示容器里面的路径，source表示本地硬盘路径

# 示例创建并挂载并使用网络
[root@manager43 ~]# docker service create --replicas 1 --mount t
```

5、多服务 Swarm 集群部署

问：上面我们只是对单独的一个 nginx 服务进行的集群部署，那如果要统一编排多个服务呢?
答：docker 三剑客中有个 compose 这个就是对单机进行统一编排的，它的实现是通过 docker-compose.yml 的文件，这里我们就可以结合 compose 和 swarm 进行多服务的编排 (docker compose 教程)

温馨提示:
我们这里要部署的服务有三个(nginx服务，visualizer服务，portainer服务)
docker service部署的是单个服务，我们可以使用docker stack进行多服务编

1）编写docker-compose.yml文件

```
[root@manager43 ~]# mkdir testswarm
[root@manager43 ~]# cd testswarm/
[root@manager43 testswarm]# cat docker-compose.yml
version: "3"
services:
  nginx:
    image: nginx
    ports:
      - 8888:80
    deploy:
      mode: replicated
      replocas: 3

  visualizer:
    image: dockersamples/visualizer
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager]

  portainer:
    image: portainer/portainer
    ports:
      - "9000:9000"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager]
```

2）通过这个yml文件部署服务

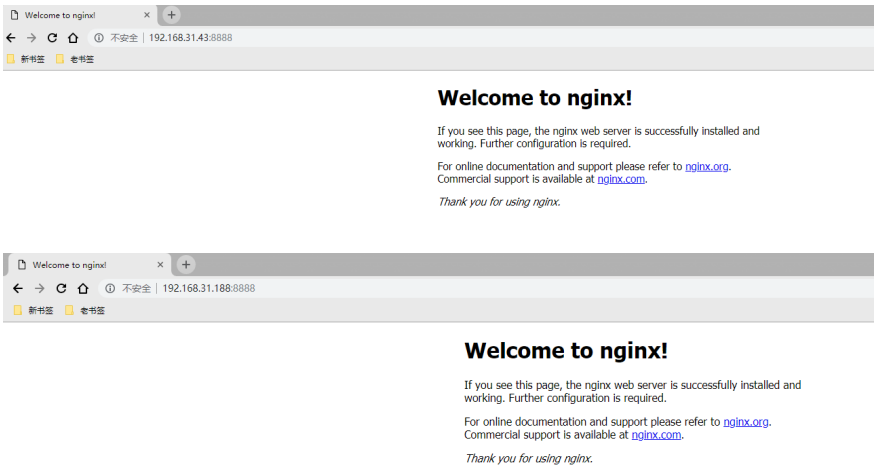
```
[root@manager43 testswarm]# docker stack deploy -c docker-compos
Creating network deploy_deamon_default
```

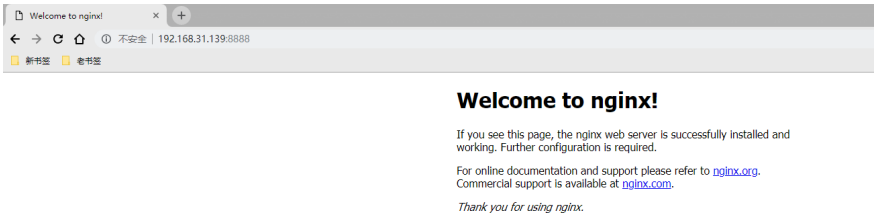
```
Creating service deploy_deamon_portainer
Creating service deploy_deamon_nginx
Creating service deploy_deamon_visualizer
```

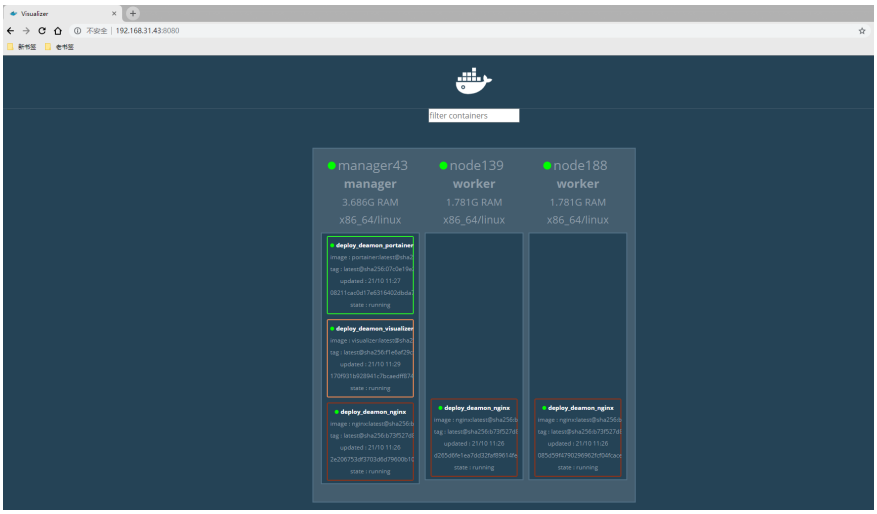
通过上面的执行过程可以看出这样创建会默认创建一个网络并使用它，名字都是

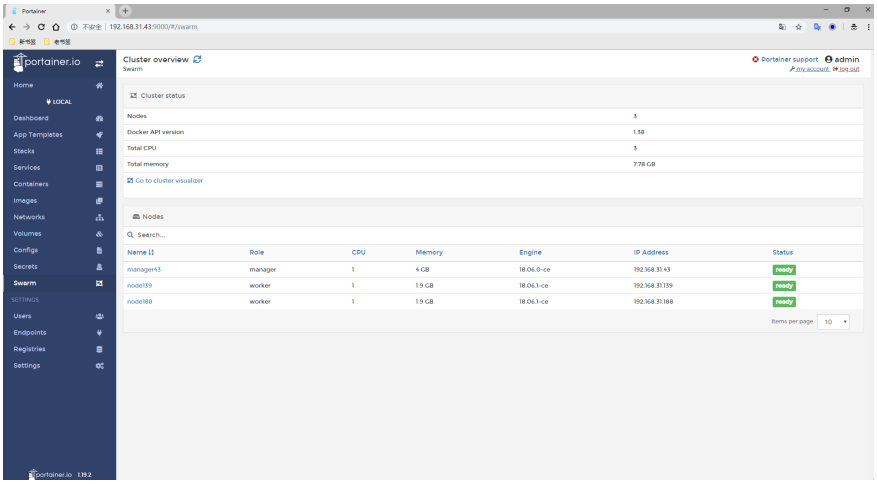
```
# 查看创建服务
[root@manager43 testswarm]# docker service ls
ID                  NAME                        MODE
xj2f1t5ax3nm        deploy_deamon_nginx         replicated
ky9qpldr5abb        deploy_deamon_portainer     replicated
r47ff177x1ir        deploy_deamon_visualizer    replicated

[root@manager43 testswarm]# docker service ps deploy_deamon_ngin
ID                  NAME                      IMAGE
z3v4uc1ujsnq        deploy_deamon_nginx.1     nginx:latest
jhg3ups0cko5        deploy_deamon_nginx.2     nginx:latest
3e6guv791x21        deploy_deamon_nginx.3     nginx:latest

[root@manager43 testswarm]# docker service ps deploy_deamon_port
ID                  NAME                          IMAGE
whyuvy82cvvw        deploy_deamon_portainer.1     portainer/portai

[root@manager43 testswarm]# docker service ps deploy_deamon_visu
ID                  NAME                          IMAGE
wge5w1eqykg3        deploy_deamon_visualizer.1    dockersamples/v
```

测试









八、Docker Swarm 容器网络

在Docker版本1.12之后swarm模式原生支持覆盖网络(overlay networks)，可
这样只要是这个覆盖网络内的容器，不管在不在同一个宿主机上都能相互通信，

swarm模式的覆盖网络包括以下功能：
1）可以附加多个服务到同一个网络。
2）默认情况下，service discovery为每个swarm服务分配一个虚拟IP地址(vi
3）可以配置使用DNS轮循而不使用VIP
4）为了可以使用swarm的覆盖网络，在启用swarm模式之间你需要在swarm节点之
5）TCP/UDP端口7946 – 用于容器网络发现
6）UDP端口4789 – 用于容器覆盖网络

实例如下：
----------在Swarm集群中创建overlay网络------------
```
[root@manager-node ~]# docker network create --driver overlay --
```

参数解释：
–opt encrypted 默认情况下swarm中的节点通信是加密的。在不同节点的容器
--subnet 命令行参数指定overlay网络使用的子网网段。当不指定一个子网时，

```
[root@manager-node ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOP
d7aa48d3e485        bridge              bridge              loca
9e637a97a3b9        docker_gwbridge     bridge              loca
b5a41c8c71e7        host                host                loca
7f4fx3jf4dbr        ingress             overlay             swar
3x2wgugr6zmn        ngx_net             overlay             swar
0808a5c72a0a        none                null                loca
```

由上可知，Swarm当中拥有2套覆盖网络。其中"ngx_net"网络正是我们在部署容
Swarm 管理节点会利用 ingress 负载均衡以将服务公布至集群之外。
在将服务连接到这个创建的网络之前，网络覆盖到manager节点。上面输出的SC
在将服务连接到这个网络后，Swarm只将该网络扩展到特定的worker节点，这个
在那些没有运行该服务任务的worker节点上，网络并不扩展到该节点。

------------------将服务连接到overlay网络-------------------
```
[root@manager-node ~]# docker service create --replicas 5 --netw
```

上面名为"my-test"的服务启动了3个task，用于运行每个任务的容器都可以彼此
```
[root@manager-node ~]# docker service ls
ID            NAME     REPLICAS  IMAGE  COMMAND
dsaxs6v463g9  my-test  5/5       nginx
```

在manager-node节点上，通过下面的命令查看哪些节点有处于running状态的任

```
[root@manager-node ~]# docker service ps my-test
ID                          NAME        IMAGE  NODE          DESIR
8433fuiy7vpu0p80arl7vggfe   my-test.1   nginx  node2         Runni
f1h7a0vtojv18zrsiw8j0rzaw   my-test.2   nginx  node1         Runni
ex73ifk3jvzw8ukurl8yu7fyq   my-test.3   nginx  node1         Runni
cyu73jd8psupfhken23vvmpud   my-test.4   nginx  manager-node  Runni
btorxekfix4hcqh4v83dr0tzw   my-test.5   nginx  manager-node  Runni
```

可见三个节点都有处于running状态的任务，所以my-network网络扩展到三个节

可以查询某个节点上关于my-network的详细信息：

```
[root@manager-node ~]# docker network inspect ngx_net
[
    {
        "Name": "ngx_net",
        "Id": "3x2wgugr6zmn1mcyf9k1du27p",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.10.19.0/24",
                    "Gateway": "10.10.19.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "00f47e38deea76269eb03ba13695ec0b0c740601c85019546d6
                "Name": "my-test.5.btorxekfix4hcqh4v83dr0tzw",
                "EndpointID": "ea962d07eee150b263ae631b8a7f8c195
                "MacAddress": "02:42:0a:0a:13:03",
                "IPv4Address": "10.10.19.3/24",
                "IPv6Address": ""
            },
            "957620c6f7abb44ad8dd2d842d333f5e5c1655034dc43e49abb
                "Name": "my-test.4.cyu73jd8psupfhken23vvmpud",
                "EndpointID": "f33a6e9ddf1dd01bcfc43ffefd19e1951
                "MacAddress": "02:42:0a:0a:13:07",
                "IPv4Address": "10.10.19.7/24",
                "IPv6Address": ""
```

```
            }
        },
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "2
        },
        "Labels": {}
    }
]
```

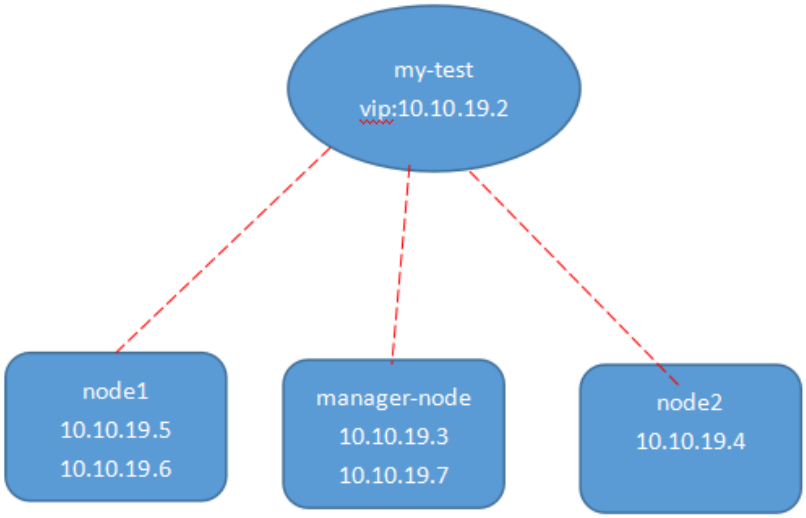
从上面的信息可以看出在manager-node节点上，名为my-test的服务有一个名为my-test.4.cyu73jd8psupfhken23vvmpud的task连接到名为ngx_net的网络上

```
[root@node1 ~]# docker network inspect ngx_net
.......
        "Containers": {
            "7d9986fad5a7d834676ba76ae75aff2258f840953f1dc633c3e
                "Name": "my-test.3.ex73ifk3jvzw8ukurl8yu7fyq",
                "EndpointID": "957ca19f3d5480762dbd14fd9a6a1cd01
                "MacAddress": "02:42:0a:0a:13:06",
                "IPv4Address": "10.10.19.6/24",
                "IPv6Address": ""
            },
            "9e50fceada1d7c653a886ca29d2bf2606debafe8c8a97f2d791
                "Name": "my-test.2.f1h7a0vtojv18zrsiw8j0rzaw",
                "EndpointID": "b1c209c7b68634e88e0bf5e100fe03435
                "MacAddress": "02:42:0a:0a:13:05",
                "IPv4Address": "10.10.19.5/24",
                "IPv6Address": ""
            }
        },
.........

[root@node2 web]# docker network inspect ngx_net
........
        "Containers": {
            "4bdcce0ee63edc08d943cf4a049eac027719ff2dc14b7c3aa85
                "Name": "my-test.1.8433fuiy7vpu0p80arl7vggfe",
                "EndpointID": "df58de85b0a0e4d128bf332fc783f6528
                "MacAddress": "02:42:0a:0a:13:04",
                "IPv4Address": "10.10.19.4/24",
                "IPv6Address": ""
            }
        },
```

可以通过查询服务来获得服务的虚拟IP地址，如下：

```
[root@manager-node ~]# docker service inspect --format='' my-tes
[{"NetworkID":"7f4fx3jf4dbrp97aioc05pul4","Addr":"10.255.0.6/16"
```

由上结果可知，10.10.19.2其实就是swarm集群内部的vip，整个网络结构如下



加入 ngx_net 网络的容器彼此之间可以通过 IP 地址通信，也可以通过名称通信。

```
[root@node2 ~]# docker ps
CONTAINER ID    IMAGE           COMMAND                  CREATED
4bdcce0ee63e    nginx:latest    "nginx -g 'daemon off"   22 minu

[root@node2 ~]# docker exec -ti 4bdcce0ee63e /bin/bash
root@4bdcce0ee63e:/# ip addr
1: lo: mtu 65536 qdisc noqueue state UNKNOWN group default
    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
1786: eth0@if1787: mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:ff:00:08 brd ff:ff:ff:ff:ff:ff link-netn
    inet 10.255.0.8/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.255.0.6/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:feff:8/64 scope link
       valid_lft forever preferred_lft forever
1788: eth1@if1789: mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netn
    inet 172.18.0.3/16 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:3/64 scope link
       valid_lft forever preferred_lft forever
1791: eth2@if1792: mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:0a:13:04 brd ff:ff:ff:ff:ff:ff link-netn
    inet 10.10.19.4/24 scope global eth2
       valid_lft forever preferred_lft forever
    inet 10.10.19.2/32 scope global eth2
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe0a:1304/64 scope link
       valid_lft forever preferred_lft forever

root@4bdcce0ee63e:/# ping 10.10.19.3
PING 10.10.19.3 (10.10.19.3): 56 data bytes
64 bytes from 10.10.19.3: icmp_seq=0 ttl=64 time=0.890 ms
64 bytes from 10.10.19.3: icmp_seq=1 ttl=64 time=0.622 ms
.....-
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.622/0.756/0.890/0.134 ms
```

```
root@4bdcce0ee63e:/# ping 10.10.19.6
PING 10.10.19.6 (10.10.19.6): 56 data bytes
64 bytes from 10.10.19.6: icmp_seq=0 ttl=64 time=0.939 ms
64 bytes from 10.10.19.6: icmp_seq=1 ttl=64 time=0.590 ms
```

----------使用swarm模式的服务发现----------

默认情况下，当创建了一个服务并连接到某个网络后，swarm会为该服务分配一个
所以网络上的任意容器可以通过服务名访问服务。

在同一overlay网络中，不用通过端口映射来使某个服务可以被其它服务访问。S
active的task上。

如下示例：
在同一个网络中添加了一个centos服务，此服务可以通过名称my-test访问前面

```
[root@manager-node ~]# docker service create --name my-centos --
```

查询centos运行在哪个节点上（上面创建命令执行后，需要一段时间才能完成这

```
[root@manager-node ~]# docker service ps my-centos
ID                         NAME             IMAGE   NODE   DESIR
e03pqgkjs3l1qizc6v4aqaune  my-centos.1      centos  node2  Runni
```

登录centos运行的节点（由上可知是node2节点），打开centos的交互shell：

```
[root@node2 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND
e4554490d891        centos:latest            "/bin/bash"

[root@node2 ~]# docker exec -ti my-centos.1.9yk5ie28gwk9mw1h1jov
root@4bdcce0ee63e:/# nslookup my-test
Server: 127.0.0.11
Address 1: 127.0.0.11

Name: my-test
Address 1: 10.10.19.2 10.10.19.2
```

从centos容器内部，使用特殊查询 查询DNS，来找到my-test服务的所有容器的

```
root@4bdcce0ee63e:/# nslookup tasks.my-test
Server: 127.0.0.11
Address 1: 127.0.0.11

Name: tasks.my-test
Address 1: 10.10.19.4 my-test.1.8433fuiy7vpu0p80arl7vggfe
Address 2: 10.10.19.5 my-test.2.f1h7a0vtojv18zrsiw8j0rzaw
Address 3: 10.10.19.6 my-test.3.ex73ifk3jvzw8ukurl8yu7fyq
```

```
Address 2: 10.10.19.7 my-test.4.cyu73jd8psupfhken23vvmpud
Address 3: 10.10.19.3 my-test.5.btorxekfix4hcqh4v83dr0tzw
```

从centos容器内部，通过wget来访问my-test服务中运行的nginx网页服务器

```
root@4bdcce0ee63e:/# wget -O- my-test
Connecting to my-test (10.10.19.2:80)

Welcome to nginx!
...
```

Swarm的负载均衡器自动将HTTP请求路由到VIP上，然后到一个active的task容

----------为服务使用DNS round-robin-----

在创建服务时，可以配置服务直接使用DNS round-robin而无需使用VIP。这是

当你想要使用自己的负载均衡器时可以使用这种方式。

如下示例（注意：使用DNS round-robin方式创建服务，不能直接在命令里使用

```
[root@manager-node ~]# docker service create --replicas 3 --name

[root@manager-node ~]# docker service ps my-dnsrr-nginx
ID                         NAME              IMAGE  NODE
65li2zbhxvvoaesndmwjokouj  my-dnsrr-nginx.1  nginx  node1
5hjw7wm4xr877879m0ewjciuj  my-dnsrr-nginx.2  nginx  manager-node
afo7acduge2qfy60e87liz557  my-dnsrr-nginx.3  nginx  manager-node
```

当通过服务名称查询DNS时，DNS服务返回所有任务容器的IP地址：

```
root@4bdcce0ee63e:/# nslookup my-dnsrr-nginx
Server:    127.0.0.11
Address 1: 127.0.0.11

Name:      my-dnsrr-nginx
Address 1: 10.10.19.10 my-dnsrr-nginx.3.0sm1n9o8hygzarv5t5eq46ok
Address 2: 10.10.19.9  my-dnsrr-nginx.2.b3o1uoa8m003b2kk0ytl9law
Address 3: 10.10.19.8  my-dnsrr-nginx.1.55za4c83jq9846rle6eigiq1
```

需要注意的是：一定要确认VIP的连通性

通常Docker官方推荐使用dig，nslookup或其它DNS查询工具来查询通过DNS对

[TOC]

一、docker-machine

命令	说明
docker-machine create	创建一个 Docker 主机（常用 `-d virtualbox` ）
docker-machine ls	查看所有的 Docker 主机
docker-machine ssh	SSH 到主机上执行命令
docker-machine env	显示连接到某个主机需要的环境变量
docker-machine inspect	输出主机更多信息
docker-machine kill	停止某个主机
docker-machine restart	重启某台主机
docker-machine rm	删除某台主机
docker-machine scp	在主机之间复制文件
docker-machine start	启动一个主机
docker-machine status	查看主机状态
docker-machine stop	停止一个主机

二、docker-compose

命令	说明
docker-compose build	建立或者重建服务
docker-compose config	验证和查看 Compose 文件
docker-compose create	创建服务
docker-compose down	停止和删除容器，网络，镜像和卷
docker-compose events	从容器接收实时事件
docker-compose exec	登录正在运行的容器执行命令
docker-compose images	镜像列表
docker-compose kill	杀掉容器
docker-compose logs	查看容器的输出
docker-compose pause	暂停容器
docker-compose port	为端口绑定打印公共端口
docker-compose ps	容器列表
docker-compose pull	下载服务镜像
docker-compose push	上传服务镜像
docker-compose restart	重启容器
docker-compose rm	删除停止的容器
docker-compose run	运行一次性的命令
docker-compose scale	设置服务的容器数量
docker-compose start	启动服务
docker-compose stop	停止服务
docker-compose top	显示运行过程
docker-compose unpause	暂停服务
docker-compose up	创建并启动容器

三、docker swarm

命令	说明
docker swarm init	初始化集群
docker swarm join-token worker	查看工作节点的 token
docker swarm join-token manager	查看管理节点的 token
docker swarm join	加入集群中

四、docker node

命令	说明
docker node ls	查看所有集群节点
docker node rm	删除某个节点（ `-f` 强制删除）
docker node inspect	查看节点详情
docker node demote	节点降级，由管理节点降级为工作节点
docker node promote	节点升级，由工作节点升级为管理节点
docker node update	更新节点
docker node ps	查看节点中的 Task 任务

五、docker service

命令	说明
docker service create	部署服务
docker service inspect	查看服务详情
docker service logs	产看某个服务日志
docker service ls	查看所有服务详情
docker service rm	删除某个服务（ `-f` 强制删除）
docker service scale	设置某个服务个数
docker service update	更新某个服务

六、docker stack

命令	说明
docker stack deploy	部署新的堆栈或更新现有堆栈
docker stack ls	列出现有堆栈
docker stack ps	列出堆栈中的任务
docker stack rm	删除堆栈
docker stack services	列出堆栈中的服务
docker stack down	移除某个堆栈（不会删除数据）