不要假装努力，结果不会陪你演戏！

Docker 与 Kubernetes 基础

目录

第1章　Docker 容器

本章节会讲述什么是 Docker，容器与虚拟机有哪些区别，容器有哪些优劣势，Docker 的基本安装，Docker C/S 模式讲解及实现,最后总结运行容器需要哪些条件。

1.0　什么是 Docker?

Docker 最初是 dotCloud 公司创始人在法国期间发起的一个公司内部项目，它是基于 dotCloud 公司多年云服务技术的一次革新，并于 2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub 上进行维护。

在 2013 年底，dotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持。

Docker 使用 Google 公司推出的 Go 语言进行开发实现，基于操作系统内核中的 Cgroup（资源控制）、Namespace（资源隔离）与 OverlayFS（数据存储）等技术，实现基于**操作系统**层面的(**应用**)虚拟化技术。

最初实现是基于 LXC 技术，从 0.7 版本以后开始去除 LXC，转而使用自行开发的 libcontainer(容器管理技术)。

Docker 运行层次图解:
前提条件: 操作系统需要支持运行 Docker
每个操作系统中运行**一个** Docker 进程

1.2　Docker 发行版本

Docker 从 1.13.x 版本开始，版本分为企业版 EE 和社区版 CE，版本号也改为按照时间线来发布，比如 17.03 就是 2017 年 3 月，有点类似于 ubuntu 的版本发布方式。

Docker EE（企业版），**Docker CE（社区版）**

在 CentOS 系统中可以使用 **docker version** 查看版本信息

```
[root@master ~]# docker version
Client: Docker Engine - Community
 Version:           19.03.5
 API version:       1.40
```

```
Go version:     go1.12.12
Git commit:     633a0ea
Built:          Wed Nov 13 07:25:41 2019
OS/Arch:        linux/amd64
Experimental:   false
```

1.3 Docker 基本实现原理

通过三个方面实现容器化技术的前置:

1) 操作系统的 NameSpace 隔离系统资源技术，通过隔离网络、PID 进程、系统信号量、文件系统挂载、主机名与域名，来实现在同一宿主机系统中，运行不同的容器，而每个容器之间**相互隔离**，**运行互不干扰**。
2) 使用系统的 Cgroups 系统资源配额功能，限制资源包括: CPU、Memory、Blkio(块设备)、Network。
3) 通过 OverlayFS 数据存储技术，实现容器**镜像**的物理存储与新建**容器**存储。

1.3.0 讲解容器基本实现原理

1.3.1 Linux NameSpace

当一台物理主机(宿主机)运行容器的时候，为了避免容器所需系统资源之间相互干扰与。所以 Docker 利用操作系统的隔离技术-NameSpace，来实现在同一个操作系统中，不同容器之间的资源独立运行。

Linux Namespace 是 Linux 系统提供的一种资源隔离机制，可实现系统资源隔离的列表如下:

Mount	-	用于隔离文件系统的挂载点
UTS	-	用于隔离 HostName 和 DomianName
IPC	-	用于隔离进程间通信
PID	-	用于隔离进程 ID
Network	-	用于隔离网络
User	-	用于隔离用户和用户组 UID/GID

1.3.1.1 查看系统资源隔离

```
1.查找进程
[root@master jdk]# ps aux | grep ssh
root        727  0.0  0.1 112796  4296 ?        Ss   01:24   0:00 /usr/sbin/sshd -D

2.查看 NS
[root@master-1 prometheus]# ll /proc/727 /ns/
total 0
```

```
lrwxrwxrwx 1 root root 0 Nov 23 22:48 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Nov 23 22:48 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Nov 23 22:48 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 Nov 23 22:48 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Nov 23 22:48 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Nov 23 22:48 uts -> uts:[4026531838]
```

1.3.2 Cgroups (资源控制)

在操作系统解决了资源相互隔离的问题以后，还需要解决资源限制的问题，也就是避免在同一个操作系统中，防止有些资源消耗较大的容器，将整个物理机器(宿主机)的硬件资源(CPU, Memory) 占满。

在 Linux 系统中能够控制的资源列表如下:

Memory	- 内存限制
hugetlb	- huge pages 使用量
cpu	- CPU 使用率
cpuacct	- CPU 使用率
cpuset	- 绑定 cgroups 到指定 CPUs 和 NUMA 节点
innodb_lock_wait_timeout	- block 设备的 IO 速度
net_cls	- 网络接口设置优先级
devices	- mknode 访问设备权限
freezer	- suspend 和 restore cgroups 进程
perf_event	- 性能监控
pids	- 限制子树 cgroups 总进程数

1.3.2 查看系统实现的限制资源

```
[root@master-1 prometheus]# cat /proc/cgroups
#subsys_name    hierarchy       num_cgroups     enabled
cpuset  10      1       1
cpu     3       1       1
cpuacct 3       1       1
memory  8       1       1
devices 2       14      1
freezer 5       1       1
net_cls 4       1       1
blkio   7       1       1
perf_event      9       1       1
hugetlb 11      1       1
pids    6       1       1
net_prio        4       1       1
```

1.3.3 OverlayFS

OverlayFS 是一种堆叠文件系统,它依赖并建立在其它的文件系统之上(例如 ext4fs 和 xfs 等),

并不直接参与磁盘空间结构的划分,仅仅将原来系统文件中的文件或者目录进行"合并一起",**最后向用户展示"合并"的文件是在同一级的目录**，这就是联合挂载技术，相对于 AUFS（<1.12 早期使用的存储技术），OverlayFS 速度更快,实现更简单。

Linux 内核为 Docker 提供的 OverlayFS 驱动有两种：Overlay 和 Overlay2。而 Overlay2 是相对于 Overlay 的一种改进,在 Inode 利用率方面比 Overlay 更有效。但是 Overlay 有环境需求：Docker 版本 17.06.02+,**宿主机文件系统需要是 EXT4 或 XFS 格式。**

1.3.4 OverlayFS 实现方式

OverlayFS 通过三个目录：lower 目录、upper 目录、以及 work 目录实现,其中 lower 目录可以是多个, upper 目录为可以进行读写操作的目录, work 目录为工作基础目录,挂载后内容会被清空,且在使用过程中其内容用户不可见,最后联合挂载完成给用户呈现的统一视图称为 merged 目录。

1.3.5 Overlay2 命令行挂载操作

```
1.创建文件
[root@master ~]# mkdir /lower{1..3}
[root@master ~]# mkdir /upper /work /merged
2.挂载文件系统
[root@master ~]# mount -t overlay overlay -o lowerdir=/lower1:/lower2:/lower3,upperdir=/upper,workdir=/work /merged
3.查看挂载
[root@master ~]# mount | grep merged

4.在/upper 目录中写入文件,在 merged 中可以显示
[root@master /]# touch /upper/upper.txt
[root@master /]# ll /merged/
total 0
-rw-r--r-- 1 root root 0 Mar 14 02:17 upper.txt
5. 在 merged 中写入文件，实际存储到了/uppper
[root@master /]# touch /merged/d.txt
[root@master /]# ll /upper/
total 0
-rw-r--r-- 1 root root 0 Mar 14 02:19 d.txt

注:如果没有 upperdir， merged 是只读的
[root@node-2 overlay2]# umount /merged
[root@node-2 overlay2]# mount -t overlay overlay -o lowerdir=/lower1:/lower2 /merged
[root@master /]# touch /merged/c.txt
touch: cannot touch '/merged/c.txt': Read-only file system
```

1.3.6 NameSpace、Cgroup 与 OverlayFS 关系

在操作系统中，可以根据宿主机的资源情况，创建不同应用与数量不同的容器。每个容器的 CPU、Memory、Network 由系统的内核 NameSpace 进行隔离，相互之间不影响。

为了防止某个正在运行的容器大量占用宿主机的系统资源（CPU、Memory、Network），那么将由操作系统的 Cgroups 功能进行资源限制（防止独占）。

容器运行的基础是需要镜像，并且新容器的运行也是需要存储支持的。在 Docker 中使用 OverlayFS 解决这一问题。

关系列表图:

1.4 Docker 优势与局限性

优势

1）Docker 让软件开发者与维护人员可以非常方便的启动应用程序以及将程序的依赖，包到一个容器中，然后启动 Docker 应用到支持 Docker 的系统平台中，就可以实现**应用**虚拟化。

2）Docker 镜像中包含了应用运行环境和配置文件，所以 Docker 可以简化部署多种应用的工作。比如说 Web 应用、后台应用（Java/C++）、数据库应用、Hadoop 集群、消息队列等等都可以打包成一个个独立的 Docker 应用镜像来部署。

3）提升宿主机(物理服务器/云虚拟机)，系统资源的利用率。

局限性

1) 基本的 Docker 网络管理模式比较简单，主要是基于系统使用 Namespace 隔离。

2) 与其他系统的网络连通性，使用自定义的地址网段，需要借助其他插件实现与其他网段的互通，提高了网络的整体复杂度。

3) 容器中应用程序日志不方便查看与收集。

4) 容器中无法运行 Windows

1.5 容器与虚拟机区别

下图比较 Docker 和传统虚拟化方式的不同之处，可见容器是在操作系统层面上实现虚拟化，直接复用宿主机的操作系统，而传统虚拟化（KVM/XEN/VMware）方式则是在硬件层面实现,并且**完全虚拟一个操作系统**，这是两者最大的不同之处。

1.5.1 详细对比说明

1) Docker 容器的启动可以在**秒级**实现，相比传统的虚拟机方式要快很多（分钟级）。 其次，容器对系统资源的利用率很高，一台主机上可以同时运行**数百**个 Docker 应用。
2) 容器基本不消耗额外的系统资源，使系统的开销尽量小。传统虚拟机方式运行应用与虚拟机的数量 1:1 (应用级别)
3) 虚拟机技术依赖物理 CPU 和内存，属于硬件级别的(特别是桌面虚拟化)；而 Docker 构建在操作系统上，**利用操作系统的系统隔离技术运行，甚至 Docker 可以在虚拟机中运行。**
4）无需要硬件是否支持，在大多数主流的 Linux/Unix 与 Windows 系统上都支持。

1.5.1 虚拟化发展历程

1.6 Dokcer 架构与术语

Docker 使用客户端-服务器 (C/S) 架构模式，即可使用远程 API 来管理服务端和创建容器。
Docker 容器需要通过 Docker **镜像**来创建。

1.6.1 Docker C/S 架构逻辑图
1) Docker 分为客户端与服务端，客户端可以管理本地的服务端（默认），也可以管理远程的服务端。
2) Docker 服务端在**启动容器**时需要从仓库获取启动镜像。
3)运行镜像分为两个部分:
a.默认的公共仓库。b.自建与第三方的私有仓库。
4)私有仓库有两种实现方式:
a. 使用 Docker Registry 部署。b. Vmware Harbor

1.6.2 Docker 基本术语

名称	解释
Docker 镜像(Images)	Docker 镜像是用于创建 Docker 容器的模板。
Docker 容器(Container)	容器是独立运行的一个或一组应用。
Docker 客户端(Client)	Docker 客户端通过命令行或者其他工具使用 Docker API (https://docs.docker.com/reference/api/docker_remote_api) 与 Docker 的守护进程通信。

Docker 主机(Host)	一个物理或者虚拟的机器用于执行 Docker 守护进程和容器

1.7 Docker 基本安装

1.7.1 操作系统版本要求

CentOS 7 要求系统为 64 位、系统内核版本为 3.10 以上。
CentOS-6.5 或更高的版本的 CentOS 上，要求系统为 64 位、系统内核版本为 2.6.32-431 或者更高版本。

1.7.2 查看系统内核版本

```
[root@docker ~]# cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

1.7.3 关闭系统防火墙与 Selinux

```
[root@docker ~]# service firewalld stop
[root@docker ~]# service firewalld status
[root@docker ~]# getenforce
[root@docker ~]# setenforce 0
[root@docker ~]# reboot
```

1.7.3.1 安装 Docker 的准备

```
[root@docker ~]# yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine
```

#device-mapper-persistent-data and lvm2 are required by the devicemapper storage driver.

```
[root@docker ~]# yum install -y yum-utils device-mapper-persistent-data lvm2
```

1.7.4 下载 Docker CE Yum 源 (社区版)

```
root@docker ~]# yum-config-manager \
    --add-repo https://download.docker.com/linux/centos/docker-ce.repo
```

1.7.5 安装开源版

```
[root@master ~]# yum install wget net-tools vim -y
[root@docker ~]# yum search docker-ce              #可省略
```

```
[root@docker ~]# yum install docker-ce-19.03.6 docker-ce-cli-19.03.6 containerd.io
[root@docker ~]# servie docker start
[root@docker ~]# chkconfig docker on              #开机启动
```

1.7.6 Docker 信息查询

Docker 是传统的 CS 架构分为 Docker Client 和 Docker Server

```
[root@docker ~]# docker version
Client: #客户端
Version: 18.09.0
API version: 1.39
Go version: go1.10.4
Git commit: 4d60db4
Built: Wed Nov 7 00:48:22 2018
OS/Arch: linux/amd64
Experimental: false
Server: Docker Engine - Community #服务端
Engine:
Version: 18.09.0
API version: 1.39 (minimum version 1.12)
Go version: go1.10.4
Git commit: 4d60db4
Built: Wed Nov 7 00:19:08 2018
OS/Arch: linux/amd64
Experimental: false
```

1.7.8 docker 开机自启动

```
[root@docker ~]# systemctl enable docker
```

1.7.9 #查看 ip 地址

Docker 启动后，会自动创建一个 docker 0 的网桥

```
[root@docker ~]# ip a
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
inet 172.17.0.1 netmask 255.255.0.0 broadcast 0.0.0.0
ether 02:42:e1:99:6c:ef txqueuelen 0 (Ethernet)
```

1.7.9.1 Docker 状态信息查看

```
[root@docker01 ~]# docker info
Docker Root Dir: /var/lib/docker          #容量
Labels:
Experimental: false
Insecure Registries:
```

```
127.0.0.0/8
Registry Mirrors:
https://registry.docker-cn.com/
Live Restore Enabled: false
Product License: Community Engine       #社区版本
```

1.7.9.2 修改 Docker 存储数据存储目录

```
[root@docker01 ~]# vi /usr/lib/systemd/system/docker.service
[Service]
ExecStart=/usr/bin/dockerd  --graph=/data/docker  -H fd:// --containerd=/run/containerd/containerd.sock
[root@docker ~]# systemctl restart docker          #重启服务
[root@docker ~]# docker info                              #查看是否修改成功
```

1.7.9.2.1 添加阿里云镜像加速器

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://plqjafsr.mirror.aliyuncs.com"]
}
EOF
[root@master ~]# systemctl daemon-reload
[root@master ~]# systemctl restart docker
```

1.7.9.3 运行第一个 Docker 应用

检验 Docker 是否可以正常运行

```
[root@docker ~]# docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

1.8 Docker C/S 模式

Docker 客户端和服务端是使用 Socket 方式连接，主要有以下几种方式：

1） 本地的 socket 文件 unix:///var/run/docker/sock （默认）
2） tcp://host:prot （演示）
3） fd://socketfd

1.8.1 Docker 默认连接方式

#未启动的状态，说明 Docker 在默认情况下使用本地的 var/run/docker.sock 连接

```
[root@master ~]# service docker stop
[root@ master ~]# docker info
Client:
 Debug Mode: false
Server:
ERROR: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
errors pretty printing info
```

1.8.2 设置 Docker 远程使用 TCP 的连接方式

#打开 sock 与 tcp 连接方式

```
[root@ master ~]#   vim /usr/lib/systemd/system/docker.service
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
#修改为:
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock -H fd:// --containerd=/run/containerd/containerd.sock
```

1.8.3 重启 Docker 服务

```
[root@master ~]# systemctl daemon-reload
[root@master ~]# service docker restart
```

1.8.4 查看 Docker 运行状态

```
[root@master-1 ~]# service docker status
Redirecting to /bin/systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
Main PID: 2218 (dockerd)
    Tasks: 12
   Memory: 42.0M
   CGroup: /system.slice/docker.service
           └─2218 /usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock -H fd:// --containerd=/run/containerd/containerd.sock
```

1.8.5 查看地址端口监听

```
[root@master ~]# netstat -nltup |grep 2375
tcp6       0      0 :::2375                 :::*                    LISTEN      2218/dockerd
```

1.8.6 远程连接 Docker TCP 查看 Docker 信息

在另外一台安装 Docker 的宿主机连接(从 192.168.91.9 客户端远程到 192.168.91.8 服务端)
192.168.91.9 需要安装 Docker 客户端

```
[root@node-1 ~]# docker -H 192.168.91.8:2375 info
Containers: 12
 Running: 0
 Paused: 0
 Stopped: 12
Images: 1
Server Version: 19.03.4
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay
Swarm: inactive
Kernel Version: 3.10.0-862.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
Number of Docker Hooks: 3
CPUs: 1
Total Memory: 3.685 GiB
Name: master-1
Docker Root Dir: /var/lib/docker
Registry: https://index.docker.io/v1/
```

1.8.7 远程连接查看 Docker Images

```
[root@node-1 ~]# docker -H 192.168.91.8:2375 images
REPOSITORY      TAG                 IMAGE ID            CREATED             SIZE
hello-world             latest              fce289e99eb9            14 months ago           1.84kB

#远程启动容器
[root@node-1 ~]# docker -H 192.168.91.8:2375 ps --all

[root@node-1 ~]# docker -H 192.168.91.8:2375 start 8a9ec62bea66
```

1.9 Docker 应用程序运行条件

1) 计算机硬件: CPU、内存、磁盘、显卡、网卡(物理机/虚拟机)。
2) 支持运行 Docker 的操作系统 (NS、Cgroups、OverlayFS)。
3) 安装 Docker 服务，并且能够正常运行。
4) 需要可以运行在 Docker 里面的镜像，镜像来自本地、docker hub、远程私有仓库。
5) 在镜像加载需要运行的程序（最终目的）。

1.9.1 程序在容器中运行条件逻辑图

第 2 章 Docker 镜像

在上章节对 Docker 有了初步的了解之后，在本章将学习容器镜像的加载流程,镜像的基本操作以及与系统存储技术 OverlayFS 的关系。

2.0 容器运行基础

镜像是 Docker 运行的基础，就好比计算机硬件需要安装操作系统. 使用 Docker Images 命令可以看到当前系统中存在的镜像。当运行容器时，使用的镜像如果在本地系统中不存在，Docker 就会自动从 Docker 镜像仓库或者配置的私有仓库中下载；默认是从 Docker Hub 公共镜像源下载。

上节在 Docker C/S 架构逻辑图中提到了私有仓库。这节主要讲镜像的基本操作。

2.1 Docker 加载镜像流程

1）检查本地是否有与启动镜像相匹配的镜像。
2）查询与镜像地址中是否有启动镜像
3）如果在镜像地址中没有完整的地址，则从默认的 Docker Hub 下载。
Docker 加载镜像的两种方式:
公共仓库与私有仓库

2.2 Docker 镜像基本操作

2.2.1 搜索 Docker Hub 有哪些镜像

docker search [镜像名字]

```
[root@master jdk]# docker search nginx
```

2.2.2 从 Docker Hub 下载镜像

下载镜像，（没有指定版本，默认会下载最新版 latest）

docker pull nginx:latest = docker pull nginx

```
[root@master jdk]# docker pull nginx:latest
```

从第三方docker 镜像仓库或者私有仓库下载镜像方法

```
[root@docker ~]# docker pull repo.abc.com/httpd:latest #地址不存在
```

2.2.3 镜像加速器

#在下载 Docker Hub 镜像非常慢的情况下，可以使用镜像加速功能。配置加速可以用官网提供的，也可以使用私有仓库。

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://plqjafsr.mirror.aliyuncs.com"]
}
EOF
[root@master ~]# systemctl daemon-reload
[root@master ~]# systemctl restart docker
```

2.2.4 基于容器创建镜像系统

#类似虚拟机的克隆功能

#在运行当中的容器基础上创建一个新的镜像（相当于 vm 的**完整快照**）

```
[root@docker ~]# docker ps -a                    #显示所有的容器, ps 只是显示正在运行的容器
[root@master ~]# docker run -d -P nginx   #启动容器
[root@master /]# docker container commit 960d3f0e61eb centos-nginx:1.0 #容器运行 ID
sha256:7af7ffdea5e8ef703fb7cb30baaff3b3e75a1297bfbc75bb15454dd7eb622215

#检查
[root@master /]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos-nginx        1.0                 7af7ffdea5e8        21 seconds ago      127MB

#不能删除原来的镜像(父镜像)
[root@master /]# docker rmi 6678c7c2e56c
```

```
Error response from daemon: conflict: unable to delete 6678c7c2e56c (cannot be forced) - image has dependent child images

#使用快照镜像启动新容器
[root@master /]# docker run -d -p 8090:80 centos-nginx:1.0
1dabb29c5a87a4ec8d58040e6ea4a21dcb17dda006168ce56021fd232208c18d
```

2.2.5 删除镜像

```
[root@master ~]# docker image rm centos-nginx
Error: No such image: centos-nginx

[root@docker ~]# docker image rm 07ddb4d9a8ab
Untagged: centos-nginx:latest
Deleted: sha256:07ddb4d9a8abfd51849f6b2747c115a28e2dd9587a56af881c43cc890e874861
Deleted: sha256:41c4debe112f55832a841e976e96d61ad170c7646460560799313c82f9f36f44

[root@docker ~]# docker rmi fdf13fa91c6e
Error response from daemon: conflict: unable to delete fdf13fa91c6e (must be forced) - image is
being used by stopped container f30520fafcbf
#注：如果镜像系统正在运行，需要停止进程才可以删除镜像系统，否则无法删除！
或者选择强制删除 (注意镜像之间的依赖关系)
[root@docker ~]# docker rmi --force fdf13fa91c6e
[root@docker ~]# docker rmi -f fdf13fa91c6e #缩写 -f 参数 == --force
```

2.2.6 导出镜像

```
#导出镜像为 tar 包
[root@master /]# docker image save nginx:latest > nginx.tar.gz
```

2.2.7 导入镜像

#直接使用tar 包导入镜像,注意如果原来导出的源镜像还是在存在于系统中，那么新导入的镜像在系统将不可见。

```
[root@master /]# docker image load -i nginx.tar.gz
Loaded image: nginx:latest

#强制删除原来的镜像
[root@master /]# docker rmi nginx:latest --force

#再执行导入
[root@master /]# docker image load -i nginx.tar.gz
Loaded image: nginx:latest
```

```
#最后查看
[root@master /]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               latest              6678c7c2e56c        9 days ago          127MB
```

2.2.8 镜像tag标签

语法：docker image tag 源_IMAGE[:TAG] 目标_IMAGE[:TAG]

类似linux 硬链接,在推送镜像到私有仓库时，需要添加tag。

```
[root@docker01 ~]# docker image tag nginx:latest repo.abc.com/nginx:v1.0
[root@master /]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               latest              6678c7c2e56c        9 days ago          127MB
repo.abc.com/nginx  v1.0                6678c7c2e56c        9 days ago          127MB

#可以使用新的 tag 标签启动容器
[root@master /]# docker run -d -p 9091:80 repo.abc.com/nginx:v1.0
1f5f20b763ccb7661b41701b9319876a0bd074da12e4e77ba0ff7879c2b19627
```

2.2.9 构建Java基础镜像

之前使用为Docker自带镜像仓库中的镜像，现在我们自己来构建私有镜像

#在java 镜像中为什么要构建alpine-glibc？

```
#Java Dockerfile
[root@master /]# mkdir -p /root/jdk/
[root@master /]# vim Dockerfile
FROM frolvlad/alpine-glibc          #模板镜像
MAINTAINER Tony                                 #创建者
RUN echo "https://mirror.tuna.tsinghua.edu.cn/alpine/v3.4/main/" > /etc/apk/repositories    #修改源
RUN apk add --no-cache bash                     #安装 bash
ADD jre1.8.0_211.tar.gz /usr/java/jdk/          #添加文件
ENV JAVA_HOME /usr/java/jdk/jre1.8.0_211        #设置环境变量
ENV PATH ${PATH}:${JAVA_HOME}/bin
RUN chmod +x /usr/java/jdk/jre1.8.0_211/bin/java
WORKDIR /opt                                    #工作目录
```

2.3.0 构建镜像

```
#需要把 jre1.8.0_211.tar.gz 放到与 Dockerfile 同级目录
[root@docker01 ~]# docker pull frolvlad/alpine-glibc    #拉取镜像
[root@master jdk]# docker build -t jre8:1.0 .                   #编译镜像
#查看镜像
[root@master jdk]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
jre8                1.0                 c65645093f97        16 seconds ago      136MB
```

```
#添加 tag
[root@master jdk]# docker tag jre8:1.0 repo.abc.com/jre8:v1.0
```

2.3.0.1 运行镜像

```
[root@docker01 ~]# docker run -it jre8:1.0 bash                    #启动并且进入容器
bash-4.3# java -version
java version "1.8.0_211"
Java(TM) SE Runtime Environment (build 1.8.0_211-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.211-b12, mixed mode)
```

2.3.1 镜像构建过程

构建容器：通过父镜像上创建一个容器，重新封装成一个镜像，变成下一个新启动容器的只读层(LOWER)。 但是如果当前容器需要对镜像进行修改，就需要加载所有镜像的只读层到容器中进行修改。

#特别注意之前提到的容器启动，修改镜像内容是不会修改源镜像；只是修改启动容器的内容。在下一节将详细讲述镜像存储的原理。

#查看容器的构建历史

missing 为当前镜像的**源镜像**构建历史 FROM frolvlad/alpine-glibc

f83c32dc03f3 为当前镜像构建的历史

```
[root@master jdk]# docker history jre8:1.3
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
ea65e9cde677   About a minute ago   /bin/sh -c #(nop) WORKDIR /opt                 0B
d0afad7e579e   About a minute ago   /bin/sh -c chmod +x /usr/java/jdk/jre1.8.0_2…   8.46kB
12d49a4a8a61   6 minutes ago        /bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:…   0B
256df34ef73f   6 minutes ago        /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/java/j…   0B
dba16cec6c49   7 minutes ago        /bin/sh -c #(nop) ADD file:d614f1012c68493ba…   116MB
22b0f5f4b551   17 minutes ago       /bin/sh -c apk add --no-cache bash              3.56MB
c97cdc4cfa33   17 minutes ago       /bin/sh -c echo "https://mirror.tuna.tsinghu…   54B
f26f217e4a3f   17 minutes ago       /bin/sh -c #(nop)  MAINTAINER Tony              0B
f83c32dc03f3   2 days ago           /bin/sh -c apk add --update curl &&  curl -…   11.7MB
<missing>      2 days ago           /bin/sh -c #(nop)  ENV GLIBC_VERSION=2.31-r0    0B
<missing>      2 days ago           /bin/sh -c #(nop)  MAINTAINER Jean Blanchard…   0B
<missing>      7 weeks ago          /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      7 weeks ago          /bin/sh -c #(nop) ADD file:e69d441d729412d24…   5.59MB
```

2.3.2 查询镜像详情

#存储信息

```
[root@master jdk]# docker image inspect jre8:1.3
```

2.4 Docker 镜像和 Overlay2 关系

Docker 需要OverlayFS文件系统存储

1) Docker中的镜像采用分层构建设计,每个层可以称之为"layer", 镜像文件默认存在目录:/var/lib/docker/overlay2

2) Docker支持的文件存储如下: AUFS、OverlayFS、VFS、Brtfs等。

3)通过Docker Info命令查看当前的存储驱动

```
[root@node-2 overlay2]# docker info | grep -E  " Storage Driver|Server Version"
Server Version: 19.03.4
Storage Driver: overlay2
```

4)通常ubuntu类的系统默认采用的是AUFS, centos7.1+系列采用的是OverlayFS。

第 3 章 Docker 镜像存储机制

本章节是对上章节Docker镜像原理理解的巩固,从Linux系统运行基础到OverlayFS存储机制去了解与分析;在底层,镜像是怎样实现存储的; 并且会详细说明存储文件的作用。

3.1 Linux 系统运行基础

Linux 系统正常运行, 通常需要两个文件系统:

3.1.1 boot file system (bootfs)

1) 包含 Boot Loader与Kernel文件,用户不能修改这些文件。并且在系统启动过程完成之后,整个系统的内核都会被加载进内存。此时bootfs会被卸载, 从而释放出所占用的系统内存。

2) 在容器中可以运行不同版本的Linux, 说明对于同样内核版本的不同的 Linux 发行版的 bootfs 都是一致的, 否则会无法启动。因此可以推断, Docker运行是需要内核支持的。

3) Linux系统中典型的bootfs目录: (核心) /boot/vmlinuz、(核心解压缩所需 RAM Disk) /boot/initramfs

3.1.2 root file system (rootfs)

1) 不同的Linux发行版本, bootfs相同, rootfs不同(二进制文件)。

2）每个容器有自己的 **rootfs**，它来自不同的 Linux 发行版的基础镜像,包括 Ubuntu，Debian 和 SUSE 等。

3) 使用不同的rootfs 就决定了，在构建镜像的过程中，可以使用哪些系统的命令。

4) 典型的rootfs 目录: /dev、/proc、/bin、/etc、 /lib、/usr

3.2 OverlayFS 存储原理

OverlayFS 结构分为三个层: LowerDir、Upperdir、MergedDir

1) LowerDir （只读）

只读的 image layer，其实就是 rootfs，在使用 Dockfile 构建镜像的时候, Image Layer 可以分很多层,所以对应的 lowerdir 会很多（源镜像）。

2) Upperdir （读写）

upperdir 则是在 lowerdir 之上的一层，为读写层。容器在启动的时候会创建，所有对容器的修改，都是在这层。**比如容器启动写入的日志文件，或者是应用程序写入的临时文件。**

3) MergedDir （展示）

merged 目录是容器的挂载点,在用户视角能够看到的所有文件，都是从这层展示的。

LowerDir、Upperdir、MergedDir 关系图:

3.3 分析镜像存储结构

3.3.1 获取镜像存储路径

#通过镜像信息获取到物理存储位置

```
[root@master jdk]# docker image inspect jre8:1.3
 "Architecture": "amd64",
 "Os": "linux",
 "Size": 136406576,
 "VirtualSize": 136406576,
 "GraphDriver": {
        "Data": {
            "LowerDir":
"/var/lib/docker/overlay2/ba469a9497fe6894e9022c2cb8b4217cd5aa1d0b35653ccd927a247bff3d2a81/diff:/var/lib/docker/overlay2/785564c2852e5b5b8f53d84ab4350c7aec6cb5a0
f44e457779877f06a95354ad/diff:/var/lib/docker/overlay2/101c2e9852e1b3e4a593f1b4ca8770fdd2b2c4656b3447dc96799527f03767c6/diff:/var/lib/docker/overlay2/fb940d476e39c
51cace728b5d709fff21c5c6421227c7ec7b2c421875767bc26/diff:/var/lib/docker/overlay2/271a364d00b2aabce86f2cb7d6c1abead3e1c8903cdd25e07a901964e3534978/diff",
            "MergedDir": "/var/lib/docker/overlay2/6700cb0c98e3e7af8bdd97d4e40d673e3a0cf0fc6768323e8d4372afde12aff2/merged",
            "UpperDir": "/var/lib/docker/overlay2/6700cb0c98e3e7af8bdd97d4e40d673e3a0cf0fc6768323e8d4372afde12aff2/diff",
```

```
            "WorkDir": "/var/lib/docker/overlay2/6700cb0c98e3e7af8bdd97d4e40d673e3a0cf0fc6768323e8d4372afde12aff2/work"
        },
        "Name": "overlay2"
    },
    "RootFS": {
        "Type": "layers",
        "Layers": [
            "sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10",
            "sha256:3219209e108e14824bd77c247110bbdb0e8ab392a016c634b8f031b610673fac",
            "sha256:d5aea7d9c0743772a38d8d81c4f1be709890827bc915465a7048a7d718fbf859",
            "sha256:3ff10a379107fb9e679ac8c001e5edbc4a01ec7b3bd86162e5932aa5ea2f8808",
            "sha256:c86fc31cf85edab6d1b9aa4750656496e0925186c112092ddd0a5909ba4a8b9b",
            "sha256:27d22f85ea93e4cbdf47b565433be94f6ce46f4e822abe6352c0f5b028bd2150"
        ]
```

3.3.2 分析Lower层

#LowerDir 层的存储是不允许创建文件，此时的LowerDir实际上是其他的镜像的UpperDir层，也就是说在构建镜像的时候，如果发现构建的内容相同，那么不会重复的构建目录，而是使用其他镜像的Upper 层来作为本镜像的Lower

```
[root@master jdk]# touch /var/lib/docker/overlay2/ba469a9497fe6894e9022c2cb8b4217cd5aa1d0b35653ccd927a247bff3d2a81/diff/lower.txt
```

3.3.3 分析Upper层

#在Upper层创建文件

```
[root@master jdk]# touch /var/lib/docker/overlay2/6700cb0c98e3e7af8bdd97d4e40d673e3a0cf0fc6768323e8d4372afde12aff2/diff/upper.txt
```

3.4 运行中容器的存储结构

3.4.1 启动容器

#前台启动，直接进入到容器

```
[root@master jdk]# docker run -it jre8:1.3 bash
```

3.4.2 查看容器挂在信息

#容器启动以后，挂载merged、lowerdir、upperdir以及workdir目录

#lowerdir是只读的image layer，其实就是rootfs

```
#获取容器 ID
[root@master ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
NAMES
39d421864be6        jre8:1.3            "bash"              11 seconds ago      Up 10 seconds                           agitated_heyrovsky
```

3.4.3 查看容器存储目录信息

#注意在所有的启动容器中会自动添加init目录，此目录是存放系统的hostname与域名解析文件。


```
[root@master ~]# docker inspect 39d421864be6
        "GraphDriver": {
            "Data": {
                "LowerDir": "/var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d-init/diff:/var/lib/docker/overlay2/6700cb0c98e3e7af8bdd97d4e40d673e3a0cf0fc6768323e8d4372afde12aff2/diff:/var/lib/docker/overlay2/ba469a9497fe6894e9022c2cb8b4217cd5aa1d0b35653ccd927a247bff3d2a81/diff:/var/lib/docker/overlay2/785564c2852e5b5b8f53d84ab4350c7aec6cb5a0f44e457779877f06a95354ad/diff:/var/lib/docker/overlay2/101c2e9852e1b3e4a593f1b4ca8770fdd2b2c4656b3447dc96799527f03767c6/diff:/var/lib/docker/overlay2/fb940d476e39c51cace728b5d709fff21c5c6421227c7ec7b2c421875767bc26/diff:/var/lib/docker/overlay2/271a364d00b2aabce86f2cb7d6c1abead3e1c8903cdd25e07a901964e3534978/diff",
                "MergedDir": "/var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d/merged",
                "UpperDir": "/var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d/diff",
                "WorkDir": "/var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/java/jdk/jre1.8.0_211/bin",
                "GLIBC_VERSION=2.31-r0",
                "JAVA_HOME=/usr/java/jdk/jre1.8.0_211"
            ],
            "Cmd": [
                "bash"
            ],
            "Image": "jre8:1.3",
            "Volumes": null,
            "WorkingDir": "/opt",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        }
    }
]
```


3.5 容器文件存储解析

3.5.1 容器运行时的UpperDir目录结构

```
[root@master ~]# touch /var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d/diff/c1.txt
#在进入到操作查看
bash-4.3# cat /c1.txt
```

3.5.1.1 Work 目录

work目录用于联合挂载指定的工作目录,在overlay 把文件挂载到 upperdir后, work内容会被清空,且在使用过程中（为空）其内容用户不可见。

3.5.1.3 用户视角层Merged

#最后给用户展示的层,一般看到为一个完整的操作系统文件系统结构

```
[root@master ~]# ll   /var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d/merged
total 4
....省略.....
-rw-r--r-- 1 root root      0 Mar 14 03:58 lower.txt
-rw-r--r-- 1 root root      0 Mar 14 04:03 upper.txt
drwxr-xr-x 1 root root     18 Mar 14 03:10 usr
```

3.5.2 Lower 层

#Lower 包括两个层:a. 系统的init。b.容器的镜像层

```
Lower 记录父层的链接名称
[root@node-2 ~]# cat /var/lib/docker/overlay2/d4dc057329ecbf5a2f97293b6d49078e9cce6869a9f049ba5bc365f6fba424d2/lower
l/QCXVWDWYPFM5NRVMB2ZC2BE5WU:l/PUSOZBTJKJ2OBNKK2UQDNQLHCU
init 层 / 容器镜像层
```

3.5.2.1 查看init层地址指向

#容器在启动的过程中, Lower 会自动挂载init的一些文件

```
[root@master ~]# ls /var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d-init/diff/etc/
hostname  hosts  mtab  resolv.conf
```

3.5.2.2 init层主要内容是什么?

init层是以一个uuid+-init结尾表示，放在只读层(Lower)和读写层(upperdir)之间,
作用只是存放/etc/hosts、/etc/resolv.conf 等文件,

3.4.2.3 为什么需要init层?

1)容器在启动以后，默认情况下lower层是不能够修改内容的，但是用户有需求需要修改主机名与域名地址，那么就需要添加init层中的文件(hostname, resolv.conf)，用于解决此类问题.
2) 修改的内容只对当前的容器生效，而在docker commit提交为镜像时候,并不会将init层提交。
3) init 文件存放的目录为/var/lib/docker/overlay2/<init_id>/diff

3.5.2.4 查看init层文件

#hostname与resolv.conf 全部为空文件，在系统启动以后由系统写入.

```
[root@master ~]# ll /var/lib/docker/overlay2/7397527ac0e1088be96ee714bb6caf78e88badd567477f7bfbec9717a975794d-init/diff/etc/
total 0
-rwxr-xr-x 1 root root  0 Mar 14 04:09 hostname
-rwxr-xr-x 1 root root  0 Mar 14 04:09 hosts
lrwxrwxrwx 1 root root 12 Mar 14 04:09 mtab -> /proc/mounts
-rwxr-xr-x 1 root root  0 Mar 14 04:09 resolv.conf
```

#总结

1) 镜像所挂载的目录层为 Lower 层，然后通过 Merged 展示所有的文件目录与文件。用户写入的所有文件都是在 UpperDir 目录,并且会在 UpperDir 建立于 Merged 层展示的文件目录结构，所以用户就可以看到写入的文件。并且底层的镜像是不能被修改(如果挂载目录为 UpperDir,则可以修改源镜像)。
2) 在下次重新启动已经停止的容器的时候，如果容器的 ID 没有发生改变，那么所写入的文件是存在物理系统中的；反之就会是一个新的容器，之前手工创建的文件是不存在的。
3) 基于容器创建的镜像，就相当于容器的快照，可以删除原来的容器，但是不能删除原来的镜像
4) 基于镜像创建的镜像，原来的镜像就是新镜像的 low 层（build), tag 则是没有区别
5) 容器启动以后，镜像就存在于容器的 lower 层，所有的写入都是在 upper

第 4 章 容器基本操作

本章节主要讲解 Docker 容器的基本操作指令，学会查看指令的帮助;并且掌握容器常用的操作命令，对之后容器的操作会有很大的帮助。

4.0 查看 Docker 命令行帮助

#查看帮助的方法

Docker 操作命令分为：管理命令与直接命令参数

1) 管理命令为区分每个项目的命令，比如说镜像操作，就是以docker image 开头
2) 直接命令参数就是在docker 命令之后直接的命令，比如说删除镜像 docker rmi
3) 管理命令相对于直接命令参数,更加严谨。

```
[root@node-2 ~]# docker --help
```

4.1 常用基本操作列表

命令行	解释

创建container	docker container create image_name
创建并运行container 及进入交互终端参数	docker container run -it image_id CMD -i 交互模式 -t 终端 -it 为分配一个交互式终端 -d 放在后台 -p 端口映射 -v 源地址(宿主机):目标地址(容器) --name 指定容器的名字 -h 指定容器主机名 --restart=always 每次重启服务，容器跟着重启
创建并运行container 并让其在后台运行，并端口映射	docker container run -p [port in container]:[port in physical system] -d [image] [command]
查看正在运行的所有container 信息	docker container ps
查看最后创建的container	docker container ps -l
查看所有container，包括正在运行和已经关闭的	docker container ps -a
输出指定container 的stdout 信息（用来看log，效果和tail -f 类似，会实时输出。）	docker container logs -f [container]
获取container 指定端口映射关系	docker container port [container] [port]
查看container 进程列表	docker container top [container]
查看container 详细信息	docker container inspect [container]
停止continer	docker container stop [container]
强制停止container	docker container kill [container]
启动一个已经停止的container	docker container start [container]
重启container(若container 处于关闭状态，则直接启动)	docker container restart [container]
删除container	docker container rm [container] -f 强制参数

4.2 常用容器操作命令详解

4.2.0 查看容器日志

```
#启动容器, -P 生成随机映射端口
[root@localhost ~]# docker run --name t1 -d -P nginx

#查看日志
[root@localhost ~]# docker logs -f t1
```

```
#获取到端口
[root@localhost ~]# docker ps -a

#访问容器
[root@localhost ~]# curl http://localhost:32770

#显示日志
[root@localhost ~]# docker logs -f t1
172.17.0.1 - - [20/Mar/2020:03:59:44 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
```

4.2.1 创建容器

语法：docker container run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]

-i 交互模式

-t 终端

-it 为分配一个交互式终端

-d 放在后台 （nohup command &）

-p 端口映射

-v 源地址(宿主机):目标地址(容器) (本地的宿主机的目录挂载到容器目录)

-h 指定容器主机名

--name 指定容器的名字

--restart=always 每次重启服务，容器跟着重启

#创建容器，并且挂载本地目录到容器目录

#把宿主机的/test目录挂载到容器的/mnt目录（新开终端）

```
#宿主机建立目录
[root@docker ~]# mkdir /test
#启动新容器，指定主机名为 test，容器名为 t11，在终端打开-it
[root@localhost ~]# docker run -it --name t11 -h test -v /test:/mnt centos /bin/bash

#宿主机建立文件
[root@localhost test]# touch /test/1.txt

#容器查看文件
[root@1444e17a8998 mnt]# ls /mnt/1.txt
/mnt/1.txt
```

#创建一台新容器，并进入镜像系统

```
[root@docker ~]# docker container run -it centos /bin/bash
[root@f30520fafcbf /]#
```

Docker 容器内的第一个进程必须一直处于前台运行的状态（必须挂起），否则这个容器，就会处于退出状态（-d 后台运行）。
当执行 docker run 时，Docker会启动一个新进程，同时给这个进程分配其独占的文件系统。
注：使用run选项会打开另外一个会话窗口，进入容器，ID不同，并不是同一个容器，相当于在系统当中又开启一个新容器，但是镜像是使用相同的镜像。

4.2.2 容器启动、停止

```
语法：docker container start [OPTIONS] CONTAINER [CONTAINER...]
完整信息，请查看:docker container start --help
启动一个 container。

#列出所有容器，包括运行与停止的容器，获取到容器 id
[root@docker ~]# docker ps -a

#根据容器 id 启动容器
[root@docker ~]# docker start c06fc46a1176
c06fc46a1176

[root@docker ~]# docker ps # 列出正在运行的容器
```

4.2.3 进入docker 容器
4.2.3.1 exec
语法：docker container exec [OPTIONS] CONTAINER [CONTAINER...]
完整信息，请查看:docker exec–help
exec 会分配一个新的终端(pts)
docker container exec -it 容器id 或容器名字 /bin/bash
#执行进入必须要带参数和COMMAND（如：/bin/bash）

```
#进入容器可以用容器运行的 id，也可以用容器名称，其目的都是唯一定位到容器
[root@localhost ~]# docker container exec -it t11 /bin/bash (父镜像)
```

4.2.3.2 attach
语法：docker container attach CONTAINER

```
#如果 Docker 容器是使用/bin/bash 命令启动的，则可以使用 attach
[root@localhost temp]# docker attach t11
```

#exec 与attach 区别
EXEC：在一个正在运行的容器中执行命令，exec是针对已运行的容器实例进行操作，在已运行的容器中执行命令，不创建和启动新的容器，**退出shell不会导致容器停止运行。**

Attach：将本机的标准**输入（键盘）、标准输出（屏幕）、错误输出（屏幕）**附加到一个运行

的容器，也就是说本机的输入直接输到容器中，容器的输出会直接显示在本机的屏幕上，
如果退出容器的shell，容器会停止运行。

4.2.4 重启服务，容器自启动

当Docker重启时，容器能自动启动

命令行添加参数docker run --restart=always

```
[root@master ~]# docker run -d --restart=always nginx
[root@master ~]# service docker restart
```

#查看状态

```
[root@master ~]# docker ps | grep nginx
5b9a13d0aedb        nginx    "nginx -g 'daemon of…"   About a minute ago   Up 48 seconds
80/tcp              fervent_lehmann
```

4.2.5 容器其他操作

```
# 关闭所有正在运行的容器
[root@master ~]#    docker kill $(docker ps -q)

# 移除所有停止的容器
[root@master ~]# docker rm $(docker ps -a -q)
[root@demo ~]# docker container prune

# 根据状态移除
[root@master ~]# docker rm $(docker ps -q -f 'status=exited' -n 3)
[root@master ~]# docker rm $(docker ps -q -f 'status=exited')

# 根据标签移除
[root@master ~]# docker rm $(docker ps -a | grep nginx | awk '{print $1}')
```

4.3 容器资源限制

#查看容器资源情况

```
[root@master ~]# docker stats 67d1b66c71ce
```

4.3.1 #CPU 资源限制

#根据CPU核心去绑定

```
# 容器可占用的 CPU 核编号，0-3 表示占用四个核，0,1 表示占用两个核
[root@master ~]# docker run --name nginx-cpu-t1 --cpuset-cpus=0-1 -d   -P   nginx
[root@master ~]# docker run --name nginx-cpu-t2 --cpuset-cpus=0 -d   -P   nginx
```

4.3.2 #内存限制

```
[root@master ~]# docker run --name nginx-memoy-t3   -d --cpuset-cpus=0 -m 300M -P nginx
```

4.3.3 #容器空间限制

#限制单个容器使用的磁盘空间（非全局）

```
root@master ~]# docker run -it --storage-opt size=12m alpine:latest /bin/df -h | grep overlay
```

Dokcer 中使用overlay2.size （注意存储驱动overlay），限制每个容器可以占用的磁盘空间，并且需要xfs(CentOS 7默认的文件)文件系统支持，xfs挂载时使用pquota参数， 在实际的生产中，建议用**独立的磁盘**作为docker的存储盘.

#限制单个容器使用的磁盘空间（全局）

#增加一块硬盘格式化为xfs

#https://github.com/moby/moby/issues/40667

```
[root@demo ~]# mkdir /data
[root@demo jdk]# docker info | grep "Storage Driver"
 Storage Driver: overlay2
[root@demo ~]#    docker info | grep "Backing"
   Backing Filesystem: <unknown>
[root@demo ~]# fdisk /dev/sdb
[root@master ~]# mkfs.xfs /dev/sdb1
[root@demo ~]# ll /dev/disk/by-uuid/*
```

#正常情况

```
[root@demo yum.repos.d]# docker info
Server Version: 19.03.6
 Storage Driver: overlay2
   Backing Filesystem: xfs
```

#设置/etc/fstab

```
UUID=c620987d-9e05-441d-b67b-c6015c51975a /data xfs rw,pquota 0 0
Mount -a
```

#查看挂载

```
[root@master ~]# cat /proc/mounts   | grep sdb
/dev/sdb1 /data xfs rw,relatime,attr2,inode64,prjquota 0 0
```

#配置docker

```
[root@master ~]# mkdir -p /data/docker
[root@master ~]# vi /etc/docker/daemon.json
{
  "registry-mirrors": ["https://plqjafsr.mirror.aliyuncs.com"],
```

```
    "data-root": "/data/docker",
     "storage-driver": "overlay2",
     "storage-opts": [
        "overlay2.override_kernel_check=true",
        "overlay2.size=1G"
     ]
}
```

#重启docker

```
[root@master ~]# service docker restart
Redirecting to /bin/systemctl restart docker.service
```

#查看docker

```
[root@master ~]# docker info | grep 'Docker Root Dir'
WARNING: API is accessible on http://0.0.0.0:2375 without encryption.
         Access to the remote API is equivalent to root access on the host. Refer
         to the 'Docker daemon attack surface' section in the documentation for
         more information: https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
 Docker Root Dir: /data/docker
```

#启动容器测试

```
[root@master ~]# docker run -it centos /bin/bash
[root@f6b9617d3736 /]# dd if=/dev/zero of=/test.txt bs=130M count=10
dd: error writing '/ test.txt': No space left on device
```

#启动新容器，写入文件

```
[root@master ~]# docker run -it centos /bin/bash
[root@4dd62c6ef62e /]# dd if=/dev/zero of=/test1.txt bs=230M count=1
1+0 records in
1+0 records out
```

4.4 容器定时任务清理架构

第 5 章 Docker 基本网络管理

本章节讲解 Docker 的四种网络模式: Bridge、Host、 None、Container；并且会详细解释这几种网络模式的工作模式方式。

5.0 网络模式基本介绍

Docker 单机网络模式分为以下几种:
1）bridge NetWork，启动容器时使用--net=bridge参数指定，默认设置。
2）Host NetWork ，启动容器时使用--net=host参数指定。
3）None NetWork， 启动容器时使用--net=none参数指定。
4）Container NetWork，启动容器时使用--net=container:NAME_or_ID参数指定。

5.1 Docker 网络模式详解

5.1.1 host 模式

如果启动容器的时候使用host 模式，那么这个容器将不会获得一个独立的Network Namespace，而是和宿主机共用一个Network Namespace。容器将不会虚拟出自己的网卡，配置自己的IP 等，而是使用宿主机的IP 和端口。但是容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。
用主机网络的时候，一个宿主机，相同的容器，只能启动一个?

5.1.1.1 案例:容器网络host模式

#注意如果是host模式，命令行参数不能带-p/-P 主机端口:容器端口
WARNING: Published ports are discarded when using host network mode

```
[root@master ~]# docker run --name host_demo -it --network host gliderlabs/alpine   /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:d0:f0:73 brd ff:ff:ff:ff:ff:ff
    inet 192.168.91.8/24 brd 192.168.91.255 scope global ens32
       valid_lft forever preferred_lft forever
    inet6 fe80::c634:c8f0:327e:10a8/64 scope link
```

```
        valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:da:53:55:ff brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
/ # ping www.baidu.com
PING www.baidu.com (14.215.177.38): 56 data bytes
64 bytes from 14.215.177.38: seq=0 ttl=128 time=10.177 ms

[root@demo ~]# docker run --name nginx_host1111 -d --network host nginx
```

5.1.1.2 案例:容器网络host模式

#运行nginx服务，如果运行多次，由于使用主机模式，会造成端口冲突，所以只会存在一个容器在运行。

```
[root@master ~]# docker run --name host_demo_nginxq --network host -d   nginx
6378f4b3a9c045b92a19cf0aad6443010726962ad9c4731926bea75c4fa9ab7f
```

#通过主机IP访问容器服务

```
[root@master ~]# curl 192.168.91.8
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

5.1.2 bridge 模式

bridge 模式是Docker 默认的网络设置，此模式会为每一个容器分配Network Namespace、设置IP等，并将一个主机上的Docker 容器连接到一个虚拟网桥上。

当Docker进程启动时，会在主机上创建一个名为docker0的虚拟网桥，此主机上启动的Docker容器会连接到这个虚拟网桥上。虚拟网桥(根据MAC地址进行数据交换)的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。

从docker0子网中分配一个IP给容器使用，并设置docker0的IP地址为容器的默认网关。在主机上创建一对虚拟网卡veth pair设备，Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0（容器的网卡），另一端放在主机中，以vethxxx这样类似的名字命名，并将这个网络设备加入到docker0网桥中。可以通过brctl show命令查看。**如果不写--net参数，就是bridge模式**。使用docker run -p时，docker实际是在iptables做了DNAT规则，实现端口转发功能。可以使用iptables -t nat -vnL查看。

5.1.2.1 案例:容器网络模式桥接

#此模式可以添加端口映射参数:-p 2000:22

```
[root@master ~]# docker run --name host_demo_bridge -it --network bridge gliderlabs/alpine    /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

5.1.3 container 模式

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

5.1.3.1 案例:容器网络使用container

#使用其他容器的桥接网卡出外网，此模式不支持-p主机端口:容器端口

```
[root@master ~]# docker run --name host_demo_container -it \
--network container:host_demo_bridge gliderlabs/alpine    /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

5.1.4 none 模式

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

5.1.4.1 案例容器网络模式为none模式

```
[root@master ~]# docker run --name host_demo_none -it --network none gliderlabs/alpine    /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
       link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
       inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
```

5.2 容器网络操作

5.2.1 容器网络配置查看

语法:docker container inspect 容器ID
#系统默认分配的IP 地址段为172.17.0.0/16

```
[root@docker chapter]# docker container inspect 658273be34ee
.....省略.....
"NetworkSettings": {
"Bridge": "",
"SandboxID": "d2e6b8e57c55a552abb825b11ed3faff4541fe28fa20244caa29ee2b2ed8ce74",
"HairpinMode": false,
"LinkLocalIPv6Address": "",
"LinkLocalIPv6PrefixLen": 0,
"Ports": {
"80/tcp": [
{
"HostIp": "0.0.0.0",
"HostPort": "80"
}
]
},
"SandboxKey": "/var/run/docker/netns/d2e6b8e57c55",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "73023ec84983ee451e065700c61c8b43bc67fafb3997946002a7a48ca24923cf",
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
```

5.2.2 容器端口映射

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 -P 或 -p 参数来指定端口映射。
当使用 -P(大写) 标记时，Docker 会随机映射一个端口到内部容器开放的网络端口。
当使用 -p（小写）标记时，Docker 指定一个宿主机端口映射到到内如容器开放的网络端口。
使用 docker ps 可以看到，本地主机的端口被映射到了容器端口。

在一个指定端口上只可以绑定一个容器。
指定映射(docker 会自动添加一条iptables 规则来实现端口映射)
-p hostPort:containerPort
#映射主机指定端口8080到容器的端口80

```
[root@docker master ]# docker run --name nginx-demo -p 8080:80 -d nginx
#查看 iptables
[root@master ~]# iptables -L -n -t nat | grep 8080
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:8080 to:172.17.0.4:80
```

-p ip:hostPort:containerPort
#映射主机的指定IP 与指定 8180端口到容器80端口

```
[root@master ~]# docker run --name nginx-demo-8180 -p 192.168.91.8:8180:80 -d nginx
#查看 iptables
[root@master ~]# iptables -L -n -t nat | grep 8180
DNAT       tcp  --  0.0.0.0/0            192.168.91.8        tcp dpt:8180 to:172.17.0.5:80
```

-p ip::containerPort(随机端口)

```
[root@master ~]# docker run --name nginx-1 -p 192.168.91.8::80 -d nginx
[root@master ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                           NAMES
f5fb8604f330        nginx               "nginx -g 'daemon of…"   7 seconds ago       Up 6 seconds
192.168.91.8:32768->80/tcp      nginx-1
```

-p hostPort:containerPort/udp

```
 [root@master ~]# docker run --name dns-udp -p 53:53/udp -d andyshinn/dnsmasq #命令成功
```

-p 81:80 –p 443:443 可以指定多个-p

```
[root@master ~]# docker run --name nginx-demo-81 -p 81:80 -p 32:22 -d nginx
```

-P 宿主机随机端口映射

```
[root@master ~]# docker run --name nginx-port -P -d nginx
[root@master ~]# docker ps | grep nginx-port
```

```
e08c51faced1        nginx               "nginx -g 'daemon of…"   7 seconds ago       Up 5 seconds
0.0.0.0:32769->80/tcp
```

#查看容器的端口映射

```
[root@master ~]# docker port 4a52c7b5035b
80/tcp -> 192.168.91.8:8110
```

第 6 章 数据卷管理

本章节主要介绍 Docker 数据卷的使用场景；复制宿主机文件到容器中，怎样创建数据卷，以及数据卷的权限管理。

6.0 数据卷特点

数据卷是一个可供一个或多个容器使用的本地文件目录，主要特性如下：
1)数据卷可以在容器之间共享和重用
2)对数据卷的修改会立即生效
3)对数据卷的更新，不会影响镜像
4)数据卷默认会一直存在，即使容器被删除
提示:数据卷的使用，类似于 Linux 下对目录进行 mount。

6.1 使用场景

1) 程序目录
2) 程序日志
3) 集群数据目录

6.2 数据卷操作

#从容器内拷贝文件到主机上
6.2.1 语法：
docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-
#将容器的/test.txt 拷贝到主机的 /tmp目录中

```
[root@localhost ~]# docker run -it --name c1 -h c1-test centos /bin/bash
[root@c1 /]# echo "docker" > /test.txt
#cp 也可以拷贝目录
[root@master ~]# docker cp c1:/test.txt /tmp/
```

```
[root@master ~]# ls /tmp/test.txt
/tmp/test.txt
[root@master ~]# cat /tmp/test.txt
docker
```

6.2.2 #推送文件至容器/tmp 目录下，通过容器ID

```
[root@docker ~]# echo 'ceshi' >ceshi.txt
#复制本机的 ceshi.txt 文件到容器 c06fc46a1176
#注意复制到容器中的文件与挂载本机的目录到容器有不同，如果是挂载的模式，在本机目录写入文件那么也会同步到容器，如果是复制文件，则内容不会同步
[root@master ~]# docker cp ./ceshi.txt c1:/tmp
```

6.2.3 #检查

```
[root@docker ~]# docker attach c1                              #进入容器
或者:
[root@docker ~]# docker container exec -it c1 /bin/bash          #进入容器
root@c1:/tmp# ls
ceshi.txt
root@c1:/tmp# cat ceshi.txt
ceshi
```

6.3 创建数据卷

使用docker run 命令中使用-v 标识来给容器内添加一个数据卷，也可以在一次docker run 命令中，多次使用-v 标识挂载多个数据卷。

数据卷的使用场景

6.3.1 创建一个新容器并挂载本地目录

```
[root@master ~]# echo "from local txt" > /mnt/fromlocal.txt  #宿主机操作

#将本地目录/mnt 挂载到容器的/mnt
[root@docker ~]# docker run -it -v /mnt/:/mnt centos /bin/bash
[root@97f04ce38226 mnt]# cat fromlocal.txt
from local txt
#在宿主机/mnt 创建目录，会在容器挂载目录中实时看到
```

6.4 数据卷权限设置

docker 默认情况下是对数据卷有读写权限，但是通过这样的方式让数据卷只读

```
[root@demo ~]# docker run -it --name centos-ro -v /mnt:/mnt:ro  centos /bin/bash #创建文件报错
[root@875aaa01cccc mnt]# touch t1.txt
touch: cannot touch 't1.txt': Read-only file system
```

第 7 章 Dockerfile

本章节主要讲解Dockerfile的语法结构，并且会详细讲解在Dockerfile中常用指令的使用方法。

7.0 为什么需要 Dockerfile

Docker运行的基础是需要镜像，那么构建镜像的基础是Dockerfile；也就是说如果需要运行自己的应用程序，那么需要编写Dockerfile来构建镜像。

7.1 Dockfile 结构

Dockerfile是用来构建Docker镜像的语法文件，是由一系列命令和参数构成Docker所特有的脚本命令。
Dockerfile 结构如下:

7.2 Dockerfile 构建案例

#典型Java构建基础镜像Dockerfile

```
#模板镜像
FROM centos
#镜像的创建者
MAINTAINER Tony
#  建立一个目录
RUN mkdir /usr/local/java
# copy gz  压缩文件到容器目录
ADD jdk-8u141-linux-x64.tar.gz /usr/local/java/
```

```
# 建立软链接
RUN ln -s /usr/local/java/jdk1.8.0_141 /usr/local/java/jdk
# 设置环境变量
ENV JAVA_HOME /usr/local/java/jdk
ENV JRE_HOME ${JAVA_HOME}/jre
ENV CLASSPATH .:${JAVA_HOME}/lib:${JRE_HOME}/lib
ENV PATH ${JAVA_HOME}/bin:$PATH
#定义容器端口
# EXPOSE 8080
#启动命令
CMD ["java","-version"]
```

7.3 Dockerfile 语法详细解析

7.3.1 FROM（指定基础image）

构建指令，**必须指定且需要在Dockerfile 其他指令的前面**。后续的指令都依赖于该指令指定的image。FROM **指令指定的基础image** 可以是官方远程仓库中的，也可以位于本地仓库，该模板镜像决定了在**RUN 指令**中可以执行哪些系统的指令。

该指令有两种格式：

FROM <image>

指定基础image 为该image 的最后修改的版本。或者：

FROM <image>:<tag>

指定基础image 为该image 的一个tag 版本。

```
#指定镜像版本
FROM centos: latest
#注意: 如果是从私有仓库拉取镜像，如果有配置权限，那么需要先登录到私有仓库.
```

7.3.2 MAINTAINER（用来指定镜像创建者信息）

构建指令，用于将image 的制作者相关的信息写入到image 中。当我们对该image 执行docker inspect 命令时，输出中有相应的字段记录该信息。

格式：

MAINTAINER <name>

7.3.3 RUN（执行构建命令）

构建指令，RUN 可以运行任何被基础image 支持的命令。如基础image 选择了ubuntu，那么软件管理部分只能使用ubuntu 的命令。

该指令有两种格式：

RUN <command> (the command is run in a shell - `/bin/sh -c`)

RUN ["executable", "param1", "param2" ...] (exec form)
#类似于shell 模块
RUN yum install java -y

注：容器在启动时，会挂载三个配置文件-init
/etc/hostname
/etc/hosts
/etc/resolv.conf
dockerfile 每执行一个run 会临时创建一个容器，每次从创建都会重新挂载这三个配置文件。所以有对于此三个配置文件有依赖操作的命令需要处于同一个RUN（也可以在容器启动以后，执行脚本修改），如下命令。

```
RUN echo "192.168.20.200 mirrors.aliyun.com" >> /etc/hosts & \
curl -o /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-6.repo & \
yum install php php-cli php-mbstring php-gd unzip -y
```

7.3.4 CMD（设置container 启动时执行的操作）
设置指令，用于container 启动时指定的操作。该操作可以是执行自定义脚本，也可以是执行系统命令。该指令只能在文件中存在一次，如果有多个，则只执行最后一条。容易被替换。
该指令有两种格式:
CMD ["executable","参数","参数"]
CMD command param1 param2
CMD ["/bin/bash","/init.sh"]
或者
CMD ["tail","-F","/etc/hosts"]

7.3.5 ENTRYPOINT（设置container 启动时执行的操作）
设置指令，指定容器启动时执行的命令，可以多次设置，但是只有最后一个有效。
两种格式:
ENTRYPOINT ["executable", "param1", "param2"] 格式与CMD 相同
ENTRYPOINT command param1 param2
容器启动后执行的命令（无法被替换，启容器的时候指定的命令，会被当成参数）
该指令的使用分为两种情况，一种是独自使用，另一种和CMD 指令配合使用。
当独自使用时，如果你还使用了CMD 命令且CMD 是一个完整的可执行的命令，那么CMD 指令和ENTRYPOINT 会互相覆盖只有最后一个CMD 或者ENTRYPOINT 有效。
CMD 指令将不会被执行，只有ENTRYPOINT 指令被执行

7.3.6 USER（设置container 容器的用户）
设置指令，设置启动容器的用户，默认是root 用户。
指定memcached 的运行用户
#USER = su – user11(centos)
ENTRYPOINT ["memcached"]

USER daemon
或
ENTRYPOINT ["memcached", "-u", "daemon"]

7.3.7 EXPOSE（指定容器需要映射到宿主机器的端口）
设置指令，该指令会将容器中的端口映射成宿主机器中的某个端口。当你需要访问容器的时候，可以不是用容器的IP 地址而是使用宿主机器的IP 地址和映射后的端口。要完成整个操作需要两个步骤，首先在Dockerfile 使用EXPOSE 设置需要映射的容器端口，然后在运行容器的时候指定-p 选项加上EXPOSE 设置的端口，这样EXPOSE 设置的端口号会被随机映射成宿主机器中的一个端口号。也可以指定需要映射到宿主机器的那个端口，这时要确保宿主机器上的端口号没有被使用。EXPOSE 指令可以一次设置多个端口号，相应的运行容器的时候，可以配套的多次使用-p 选项。
格式:
EXPOSE <port> [<port>...]
映射一个端口
EXPOSE port1

7.3.8 ENV（用于设置环境变量）
构建指令，在image 中设置一个环境变量。
格式:
ENV <key> <value>
设置了后，后续的RUN 命令都可以使用，container 启动后，可以通过docker inspect 查看这个环境变量，也可以通过在docker run --env key=value 时设置或修改环境变量。
假如你安装了JAVA 程序，需要设置JAVA_HOME，那么可以在Dockerfile 中这样写:
ENV JAVA_HOME /path/to/java/dirent
ENV JAVA_HOME /usr/java/latest
ENV PATH $JAVA_HOME/bin:$PATH
ENV LANG en_us.UTF-8

7.3.9 ADD（从src 复制文件到container 的dest 路径）
构建指令，所有拷贝到container 中的文件和文件夹权限为0755，uid 和gid 为0；如果是一个目录，那么会将该目录下的所有文件添加到container 中，不包括目录；如果文件是可识别的压缩格式，则docker 会帮忙解压缩（注意压缩格式tar.gz）；
格式:
ADD <src> <dest>
<src> 是相对被构建的源目录的相对路径，可以是文件或目录的路径，也可以是一个远程的文件url;
<dest> 是container 中的绝对路径

```
ADD ./apache-tomcat-9.0.11.tar.gz /opt/server/tomcat
```

7.3.9.1 VOLUME（指定挂载点）
设置指令，使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使

用，也可以共享给其他容器使用。我们知道容器使用的是AUFS(overlay)，这种文件系统不能持久化数据，当容器关闭后，所有的更改都会丢失。当容器中的应用有持久化数据的需求时可以在Dockerfile 中使用该指令。

格式:

VOLUME ["<mountpoint>"]

FROM base

VOLUME ["/tmp/data"]

运行通过该Dockerfile 生成image 的容器，/tmp/data 目录中的数据在容器关闭后，里面的数据还存在。VOLUME不会存在于宿主机。

#编译JDK Dockerfile

```
[root@master jdk]# mkdir /jdk
[root@master jdk]# cat /jdk/Dockerfile
FROM docker.io/jeanblanchard/alpine-glibc
MAINTAINER Tony
RUN echo "https://mirror.tuna.tsinghua.edu.cn/alpine/v3.4/main/" > /etc/apk/repositories
RUN apk add --no-cache bash
ADD jre1.8.0_211.tar.gz /usr/java/jdk/
ENV JAVA_HOME /usr/java/jdk/jre1.8.0_211
ENV PATH ${PATH}:${JAVA_HOME}/bin
RUN chmod +x /usr/java/jdk/jre1.8.0_211/bin/java
RUN mkdir /zz
USER root
VOLUME ["/tmp/data"]
WORKDIR /opt
#EXPOSE 22
CMD ["java","-version"]
# ENTRYPOINT ["java", "-version"]
```

#编译镜像

```
[root@master jdk]# docker build -t jre8:1.5 .
```

#启动容器

```
[root@localhost images]# docker run -it --name t19 jre8:1.5 bash
bash-4.3# cd /tmp/data/
bash-4.3# touch dd.txt
```

#退出容器

#启动容器，然后再进入到容器

```
[root@master jdk]# docker start t19
[root@master jdk]# docker attach t19 (exec -it, 前提条件 bash)
bash-4.3# cd /tmp/data
```

```
bash-4.3# ls
dd.txt
```

#其他容器共享VOLUME内容

```
[root@localhost test]# docker run -it --name t20 --volumes-from t19 jre8:1.5 bash
bash-4.3# cd /tmp/data/
bash-4.3# ls
dd.txt

#在 t19 写入/tmp/data/ dd.txt 内容, t20 也可以看到
```

7.3.9.2 WORKDIR（工作目录）

设置指令，可以多次切换(相当于cd 命令)，对RUN,CMD,ENTRYPOINT 生效。

格式:

WORKDIR /path/to/workdir ，相当于cd /path/to/workdir

7.4 实战编译 Nginx 镜像

7.4.1 配置Dockerfile

```
FROM centos:centos7
MAINTAINER Tony
RUN yum install -y wget openssl epel-release
RUN yum install -y nginx
COPY nginx.conf /etc/nginx/nginx.conf
VOLUME [ "/data/www" ]
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

7.4.2 配置Nginx.conf

```
user nginx;
worker_processes    1;

error_log    /var/log/nginx/error.log warn;
pid              /var/run/nginx.pid;

events {
      worker_connections    1024;
}
```

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;
    sendfile        on;
    #tcp_nopush     on;
    keepalive_timeout  65;
    #gzip  on;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   /data/www;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
}
```

7.4.3 编译镜像

```
[root@demo nginx]# docker build -f Dockerfile -t nginx-c1:v1.0 .
```

7.4.4 配置Index.html

```
[root@demo nginx]# cat index.html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Welcome</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
    Welcome Nginx
```

```
</body>
</html>
```

7.4.4 复制Nginx index.html 到宿主机

```
[root@demo nginx]# mkdir -p /data/www/
[root@demo nginx]# cp index.html /data/www/
[root@demo nginx]# docker run --name nginx-demo    -d -p 8001:80 -v /data/www:/data/www nginx-c1:v1.0
```

7.4.5 访问服务

```
[root@demo nginx]# curl localhost:8001
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Welcome</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
    Welcome Nginx
</body>
</html>
```

7.4.6 使用自动分配端口访问

```
[root@demo nginx]# docker run --name nginx-demo1    -d -P -v /data/www:/data/www nginx-c1:v1.0
```

7.4.7 获取端口

```
[root@demo nginx]# docker port nginx-demo1
80/tcp -> 0.0.0.0:32768
```

7.4.8 访问服务

```
[root@demo nginx]# curl localhost:32768
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Welcome</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
```

```
    Welcome Nginx
</body>
</html>
```

第 8 章 Registry 私有镜像仓库

本章节讲解私有仓库的工作流程，主要是了解Docker怎么获取镜像的过程，怎样搭建属于自己的私有仓库，客户端怎样推送镜像到私有仓库，以及私有仓库的管理方法。

学过之前的章节应该会明白，容器运行是需要启动镜像，那么有了镜像，就需要有仓库存放，提供给客户端下载使用，所以这个就是私有仓库的由来。

前面的章节已经讲了镜像的拉取流程，以及镜像的存储机制。这节主要讲私有仓库Registry的搭建。而在实际的生产过程中会使用到Harbor，在Spring Cloud 与 Kubernetes 课程中会详细的讲解搭建与部署规划。

本章节的主要目的:
1）自定义打包镜像tag，推送到私有仓库。
2）熟悉Registry的基本操作。

8.0 私有镜像仓库简介

Docker Registry: 官方docker镜像存储、管理和分发工具。

部署私有仓库可以解决以下问题:
1) 下载镜像时所带来的网络延时。
2) 方便镜像版本的更新与维护
3）部署私有应用程序

8.1 私有镜像仓库工作流程

1)用户本地构建镜像，将镜像推送到Registry仓库.
2) Docker 用户使用的时候，直接从Registry 下载，无须从Docker Hub 下载.

8.2 搭建私有仓库

官方提供的私有仓库镜像，无需调整，直接使用。将宿主机目录挂载到容器中，提供数据持久化。
镜像名称：registry，默认使用最新版。
#挂载宿主机/opt/myregistry目录到容器目录/var/lib/registry

```
[root@docker01 ~]# docker run -d -p 5000:5000 --restart=always --name registry -v /opt/myregistry:/var/lib/registry registry
```

8.3 上传本地镜像至私有仓库

#给镜像打tag 标签，要写全镜像仓库地址。

```
[root@master mnt]# docker image tag centos:latest 192.168.91.8:5000/centos7:v1.0
```

#上传镜像到私有仓库

```
[root@master mnt]# docker push 192.168.91.8:5000/centos7:v1.0
```

注：报错

```
The push refers to repository [192.168.91.8:5000/centos7]
Get https://192.168.91.8:5000/v2/: http: server gave HTTP response to HTTPS client
```

出现https 错误解决方法有两种:
1） 修改Docker 节点配置文件 (本案例中)
2）添加Nginx 反向代理

```
解决办法：修改配置文件，建立信任
[root@docker01 ~]#   vi /etc/docker/daemon.json
{
"insecure-registries": ["192.168.91.8:5000"]
}
```

#重启docker 服务

```
[root@docker01 ~]# systemctl restart docker

#再次推送镜像
[root@master mnt]# docker push 192.168.91.8:5000/centos7:v1.0
The push refers to repository [192.168.91.8:5000/centos7]
0683de282177: Pushing [=========>                                         ]     38.8MB/237.1MB
```

8.4 查看私有仓库镜像列表

```
[root@master mnt]# curl http://192.168.91.8:5000/v2/_catalog
{"repositories":["centos7"]}
```

8.5 删除私有仓库中镜像

1）进入docker registry 的容器中

```
[root@docker01 ~]# docker exec -it registry /bin/sh
```

2) 删除指定镜像目录文件

```
/ # rm -fr /var/lib/registry/docker/registry/v2/repositories/centos7
```

3) 清理掉blob

```
/ # registry garbage-collect /etc/docker/registry/config.yml
```

4) 再次查看版本库

```
[root@master mnt]#    curl http://192.168.91.8:5000/v2/_catalog
{"repositories":[]}
```

8.6 生产容器容量估算

第 9 章 Docker 高级网络操作

在 Docker 的网络模式章节讲述了 Docker 支持的四中网络模式: host、container、none、bridge 以及图形解释了在这几种网络模式下, Dokcer IP 的配置情况。那么在本章节主要讲解容器之间通信的原理,Dokcer0 网桥详解,以及在 None 模式下怎样添加自定义网卡，并且还要实现添加正在运行容器的 IP 地址。

注:本章节有一定难度，可以选修。

9.1 网络通信基础

在网络通信中，两台主机的唯一标识为 **IP 地址**(V4/V6)，（类似身份证号码）基于 IP 地址实现全世界的网络互联。

计算机通信最初发展，并未实现系统化与标准化，不同厂商产出各自的网络来实现通信，这样导致了计算机缺乏灵活性和可扩展性，为了解决该问题,ISO（国际标准化组织）制定了一个国际标准 OSI（开放式通信系

统互联参考模型）, (IETF（国际互联网工程任务组）TCP/IP 模型)
OSI 模型

9.2 局域网互联技术

9.2.1 以太网发展
1973 年，美国加利福尼亚州的 Xerox 公司实现了最初的以太网(3Mbps)
1980 年, Xerox 与 DEC、Intel 实现了 10Mbps 以太网
1983 年, IEEE 标准委员会通过第一个 802.3 标准
1990 年, IEEE 通过了使用双绞线介质的以太网标准传输 10Mbps

9.2.2 局域网通信设备
集线器（HUB），**物理层的设备**

网桥又称桥接器，(Network Bridge)，**数据链路层设备**

交换机 Switch，**数据链路层设备(网管 snmp/telnet 与非网管)**

9.2.3 局域网(二层)数据转发原理
192.168.1.10 向 192.168.1.11 发送邮件，根据 OSI 七层模型，数据转发是需要封装数据包（1500 字节/分片重组 MTU）
1. 192.168.1.10 查询本机的地址缓存表（arp -a），是否有 IP 地址对应的 mac 地址，如果有直接发送数据包,如果没有则发送 ARP 广播
2. 192.168.1.10 向网络发送 ARP 广播，询问 192.168.1.11 的 MAC 地址是多少?
3. 在未隔离**广播域**的交换机上的端口都会接收到这个广播.
4. 192.168.1.11 接收到请求以后，比对目标网络地址，是否是地址，然后再回应给 192.168.1.10
5. 192.168.1.10 收到 mac 地址以后，封装数据包，发送到交换机，交换机根据目标 mac 转发数据到 192.168.1.11

9.3 容器通信基础

在Linux系统中Namespace (> 2.6.x 内核版本) 主要用于资源的隔离。在有了Namespace功能之后，在Linux系统中就可以**抽象**出多个网络子系统，并且各子系统间都有自己的网络设备，协议栈等，**彼此之间互不影响**。

如果相互隔离的Namespace之间需要通信时，则用veth-pair来做连接作为连接桥梁，使原

本互不相干的容器之间能够相互通信。

根据网络连接的方式与规模，可分为“直接相连”、“Bridge 相连”和“OVS 相连”。下面会详细讲解几种模式的实现方式。

9.3.1 直接连接

直接相连是最简单的方式，如下图，一对 veth-pair 直接将两个 namespace 连接在一起。

#同一宿主机

9.3.1.1配置不同NS，相同网段的直接连接互通

9.3.1.2 创建不同的Namespace

```
# 创建 namespace
[root@docker01 ~]# ip netns a ns1
[root@docker01 ~]# ip netns a ns2
```

9.3.1.3 创建veth-pair（veth0/veth1）

```
#创建一对 veth-pair veth0 veth1
#创建 veth0 类型为 veth，对端的地址为 veth1
[root@docker01 ~]# ip l a veth0 type veth peer name veth1
```

9.3.1.4 添加veth0/veth1至两个不同的NS

```
[root@docker01 ~]# ip l s veth0 netns ns1
[root@docker01 ~]# ip l s veth1 netns ns2
```

9.3.1.5 配置veth IP 地址

```
#给两个 veth0 veth1 配上 IP 并启用，必须为一个网段
[root@docker01 ~]# ip netns exec ns1 ip a a 10.1.1.2/24 dev veth0
[root@docker01 ~]# ip netns exec ns1 ip l s veth0 up
[root@docker01 ~]# ip netns exec ns2 ip a a 10.1.1.3/24 dev veth1
[root@docker01 ~]# ip netns exec ns2 ip l s veth1 up
```

9.3.1.6 测试veth 连通性

```
#从 veth0 ping veth1
[root@localhost ~]# ip netns exec ns1 ping 10.1.1.3
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=0.073 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.068 ms
--- 10.1.1.3 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14000ms
```

```
rtt min/avg/max/mdev = 0.068/0.084/0.201/0.032 ms
```

9.3.2 Linux Bridge连接

Linux Bridge 相当于一个**网桥**，可以中转两个namespace的流量。

如下图，两对 veth-pair 分别将两个 namespace 连到 Bridge 上。

9.3.2.1 配置不同NS，相同网络通过网桥的互通

```
# 创建 namespace
[root@docker01 ~]# ip netns a ns1
[root@docker01 ~]# ip netns a ns2
```

9.3.2.2 创建网桥br0

```
[root@docker01 ~]# ip l a br0 type bridge
[root@docker01 ~]# ip l s br0 up
```

创建两对 veth-pair

```
[root@docker01 ~]# ip l a veth01 type veth peer name br-veth0
[root@docker01 ~]# ip l a veth11 type veth peer name br-veth1
```

9.3.2.3 配置veth pair，并且加入到独立的NS与相同的br0

9.3.2.3.1 #配置veth01的namespace为ns1，设置br-veth0的桥接器为br0

```
[root@docker01 ~]# ip l s veth01 netns ns1
[root@docker01 ~]# ip l s br-veth0 master br0
[root@docker01 ~]# ip l s br-veth0 up
```

9.3.2.3.2 配置veth11的namespace为ns2，设置br-veth1的桥接器为br0

```
[root@docker01 ~]# ip l s veth11 netns ns2
[root@docker01 ~]# ip l s br-veth1 master br0
[root@docker01 ~]# ip l s br-veth1 up
```

9.3.2.4 配置veth01与veth11 IP 地址

```
# 给两个 ns 中的 veth 配置 IP 并启用
[root@docker01 ~]# ip netns exec ns1 ip a a 11.1.1.2/24 dev veth01
[root@docker01 ~]# ip netns exec ns1 ip l s veth01 up

[root@docker01 ~]# ip netns exec ns2 ip a a 11.1.1.3/24 dev veth11
[root@docker01 ~]# ip netns exec ns2 ip l s veth11 up
```

9.3.2.5 测试连通性

#veth01 ping veth11

```
[root@localhost ~]# ip netns exec ns1 ping 11.1.1.3
PING 11.1.1.3 (11.1.1.3) 56(84) bytes of data.
```

```
64 bytes from 11.1.1.3: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 11.1.1.3: icmp_seq=2 ttl=64 time=0.105 ms
--- 11.1.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.060/0.082/0.105/0.024 ms
```

9.3.3 通过OVS 连接

OVS 是第三方开源的 Bridge，功能比 Linux Bridge 要更强大
如下图所示：

9.3.3.1 安装OVS

```
[root@localhost ~]# yum install wget openssl-devel   \
python-sphinx gcc make python-devel openssl-devel kernel-devel graphviz kernel-debug-devel \
autoconf automake rpm-build redhat-rpm-config libtool python-twisted-core python-zope-interface \
PyQt4 desktop-file-utils libcap-ng-devel groff checkpolicy python-six selinux-policy-devel -y
[root@localhost ~]# mkdir -p /root/rpmbuild/SOURCES
[root@localhost ~]# cd /root/rpmbuild/SOURCES
[root@localhost ~]# wget http://openvswitch.org/releases/openvswitch-2.9.2.tar.gz
[root@localhost ~]# tar xvf openvswitch-2.9.2.tar.gz
[root@localhost ~]# rpmbuild -bb --nocheck openvswitch-2.9.2/rhel/openvswitch-fedora.spec
[root@localhost ~]# yum localinstall /root/rpmbuild/RPMS/x86_64/openvswitch-2.9.2-1.el7.centos.x86_64.rpm
-y
[root@demo SOURCES]# systemctl start openvswitch.service
[root@demo SOURCES]# systemctl enable openvswitch.service
```

9.3.4 配置veth通过OVS连接通信

9.3.4.1 通过OVS命令创建一个bridge

```
[root@localhost ~]# ovs-vsctl add-br ovs-br
```

9.3.4.2 创建两对veth-pair

```
[root@localhost ~]# ip l a veth011 type veth peer name ovs-veth0
[root@localhost ~]# ip l a veth111 type veth peer name ovs-veth1
```

9.3.4.3 配置veth-pair添加到NS与OVS Bridge 中

```
#操作 veth011
#添加 veth011 到 ns1
[root@localhost ~]# ip l s veth011 netns ns1
#将 ovs-veth0 添加到 ovs-br 桥接器
[root@localhost ~]# ovs-vsctl add-port ovs-br ovs-veth0
[root@localhost ~]# ip l s ovs-veth0 up
#操作 veth1
```

```
#添加 veth1 到 ns2
[root@localhost ~]# ip l s veth111 netns ns2
#将 ovs-veth1 添加到 ovs-br 桥接器
[root@localhost ~]# ovs-vsctl add-port ovs-br ovs-veth1
[root@localhost ~]# ip l s ovs-veth1 up
```

9.3.4.4 配置veth IP 地址

```
[root@localhost ~]# ip netns exec ns1 ip a a 12.1.1.2/24 dev veth011
[root@localhost ~]# ip netns exec ns1 ip l s veth011 up
[root@localhost ~]# ip netns exec ns2 ip a a 12.1.1.3/24 dev veth111
[root@localhost ~]# ip netns exec ns2 ip l s veth111 up
```

9.3.4.5 测试连通性

veth011 ping veth111

```
[root@localhost ~]# ip netns exec ns1 ping 12.1.1.3
PING 12.1.1.3 (12.1.1.3) 56(84) bytes of data.
64 bytes from 12.1.1.3: icmp_seq=1 ttl=64 time=0.311 ms
64 bytes from 12.1.1.3: icmp_seq=2 ttl=64 time=0.087 ms
--- 12.1.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.087/0.199/0.311/0.112 ms
```

9.4 Docker0 网桥详解

Docker 服务默认会创建一个 docker0 网桥（其上有一个 docker0 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和宿主机都在同一个物理网络。

Docker 默认指定了 docker0 接口 的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），以太网通常是 1500 Bytes，或宿主机网络路由上支持的默认值。这些值都可以在服务启动的时候进行配置。

--bip=CIDR -- IP 地址加掩码格式，例如 192.168.100.5/24
--mtu=BYTES -- 覆盖默认的 Docker mtu 配置
也可以在配置文件中配置 DOCKER_OPTS，然后重启服务。 由于目前 Docker 网桥是 Linux 网桥，用户可以使用 brctl show 来查看网桥和端口连接信息。
基本流程图:

9.4.1 查看docker0网桥

9.4.2 安装网络工具

```
#如果已经安装 Docker CE 则无需要再安装
[root@node-2 ~]#  yum install -y yum-utils device-mapper-persistent-data lvm2
[root@node-2 ~]#  yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
[root@node-2 ~]#  yum install docker-ce-19.03.6 docker-ce-cli-19.03.6 containerd.io -y
#安装网络查看工具
[root@node-2 ~]#  yum install bridge-utils
```

9.4.3 查看网络

```
#查看系统网络接口
[root@node-2 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:14:38:91 brd ff:ff:ff:ff:ff:ff
    inet 192.168.91.136/24 brd 192.168.91.255 scope global noprefixroute ens32
       valid_lft forever preferred_lft forever
    inet6 fe80::7dfd:e7a3:7683:20e7/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:2f:4c:4a:2b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

9.4.4 查看网桥

```
[root@node-2 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02422f4c4a2b       no
```

9.4.5 启动nginx demo容器

```
#运行 nginx demo
[root@node-2 ~]# docker run --name nginx-demo -p 8081:80 -d nginx
```

9.4.6 获取容器进程id

```
[root@node-2 ~]# docker inspect -f  '{{.State.Pid}}' nginx-demo
1370
```

9.4.7 查看命名空间

#系统所支持的网络命名空间(在内核中根据进程号隔离)

```
[root@node-2 ~]# ll /proc/1370/ns/
total 0
lrwxrwxrwx 1 root root 0 Nov  6 11:51 ipc -> ipc:[4026532449]
lrwxrwxrwx 1 root root 0 Nov  6 11:51 mnt -> mnt:[4026532447]
lrwxrwxrwx 1 root root 0 Nov  6 11:50 net -> net:[4026532443]
lrwxrwxrwx 1 root root 0 Nov  6 11:51 pid -> pid:[4026532450]
lrwxrwxrwx 1 root root 0 Nov  6 11:51 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Nov  6 11:51 uts -> uts:[4026532448]
```

9.4.8 通过进程id 获取到网络命名空间

```
[root@demo ~]# ll /proc/1370/ns/net
lrwxrwxrwx 1 root root 0 Mar 23 23:35 /proc/1370/ns/net -> net:[4026532443]
```

9.4.9 查看系统网络接口

```
#查看网络结构，多了一块网卡
[root@node-2 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:14:38:91 brd ff:ff:ff:ff:ff:ff
    inet 192.168.91.136/24 brd 192.168.91.255 scope global noprefixroute ens32
       valid_lft forever preferred_lft forever
    inet6 fe80::7dfd:e7a3:7683:20e7/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:2f:4c:4a:2b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2fff:fe4c:4a2b/64 scope link
       valid_lft forever preferred_lft forever
5: veth7ebd9ef@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether b2:12:dc:e7:a5:17 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::b012:dcff:fee7:a517/64 scope link
       valid_lft forever preferred_lft forever
```

9.4.9.1 查看网卡桥接

```
[root@demo ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242c736b32f       no              veth7ebd9ef
```

9.4.9.2 查看Iptables

#iptables 存在有地址转换

```
[root@node-2 ~]# iptables -L -n -t nat
Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:8081 to:172.17.0.2:80
```

9.5 创建 Docker 自定义网桥

Docker 自定义网桥

除了默认的 docker0 网桥，用户也可以指定网桥来连接各个容器。

在启动 Docker 服务的时候，使用 -b BRIDGE或--bridge=BRIDGE 来指定使用的网桥。

如果Docker服务已经运行，那需要先停止服务。

9.5.1 自定义网桥逻辑架构图

容器通过自定义的网桥bridge0访问外网

9.5.2 停止docker 服务

```
[root@node-2 ~]# service docker stop
```

9.5.3 建立自定义网桥bridge0

#设置bridge0网段为192.168.6.0/24

```
#添加网桥
[root@node-2 ~]# brctl addbr bridge0
#设置网桥 IP 地址段
[root@node-2 ~]# ip addr add 192.168.6.1/24 dev bridge0
#设置网桥状态 UP
[root@node-2 ~]# ip link set dev bridge0 up
```

9.5.4 查看网络

#查看系统增加桥接器bridge0

```
[root@node-2 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:14:38:91 brd ff:ff:ff:ff:ff:ff
    inet 192.168.91.136/24 brd 192.168.91.255 scope global noprefixroute ens32
       valid_lft forever preferred_lft forever
    inet6 fe80::7dfd:e7a3:7683:20e7/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:2f:4c:4a:2b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2fff:fe4c:4a2b/64 scope link
       valid_lft forever preferred_lft forever
6: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether 12:e7:be:6c:aa:1c brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
    inet6 fe80::10e7:beff:fe6c:aa1c/64 scope link tentative
       valid_lft forever preferred_lft forever
```

9.5.5 查看系统所有桥接器

```
[root@node-2 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
bridge0         8000.000000000000       no
docker0         8000.02422f4c4a2b       no
```

9.5.6 查看网桥状态

```
[root@node-2 ~]# ip addr show bridge0
6: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether 12:e7:be:6c:aa:1c brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
```

```
    inet6 fe80::10e7:beff:fe6c:aa1c/64 scope link
       valid_lft forever preferred_lft forever
```

9.5.7 配置Docker使用新网桥bridge0

```
[root@node-2 ~]# cat /etc/docker/daemon.json
{
  "bridge": "bridge0"
}

#完整配置
{
    "bridge": "bridge0",
  "registry-mirrors": ["https://plqjafsr.mirror.aliyuncs.com"],
    "data-root": "/data/docker",
     "storage-driver": "overlay2",
     "storage-opts": [
        "overlay2.override_kernel_check=true",
        "overlay2.size=1G"
     ]
}
```

9.5.8 重启docker 服务

```
#重载
[root@node-2 ~]# systemctl daemon-reload
[root@node-2 ~]# service docker restart
```

9.5.8.1 启动容器

#启动一个已经退出的容器

```
[root@demo ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
400fb9f1f415 centos              "/bin/bash"         27 hours ago        Exited (1) 26 hours ago                         happy_lumiere
#启动容器
[root@demo ~]# docker start 400fb9f1f415
#进入终端
[root@demo ~]# docker exec -it 400fb9f1f415 /bin/bash
[root@400fb9f1f415 /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

```
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:06:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.6.2/24 brd 192.168.6.255 scope global eth0
       valid_lft forever preferred_lft forever
```

9.5.9 运行容器，是否可以使用新网桥

9.5.9.1 重新运行容器

```
[root@node-2 ~]#   docker run -it --name centos-d1 centos /bin/bash
```

9.5.9.2 使用 ping 测试网络连通性

```
/ # ping www.baidu.com
PING www.baidu.com (163.177.151.109): 56 data bytes
64 bytes from 163.177.151.109: seq=0 ttl=127 time=9.102 ms
64 bytes from 163.177.151.109: seq=1 ttl=127 time=9.358 ms
64 bytes from 163.177.151.109: seq=2 ttl=127 time=9.284 ms
```

9.5.9.3 查看容器ip

#新容器使用bridge0网段192.168.6.0/24, docker0为172.17.0.0/16 网段

```
[root@node-2 ~]# docker run -ti centos
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
30: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:c0:a8:06:03 brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.3/24 brd 192.168.6.255 scope global eth0
       valid_lft forever preferred_lft forever
```

9.5.9.4 查看系统Iptables 转发规则

MASQUERADE:

通过MASQUERADE自动实现SNAT，把192.168.6.0网段的地址，转换发送到外网。

也就是说在自定义添加了网桥以后, docker 会自动添加网桥的网段到iptables SNAT 的规则中,实现容器启动以后能够自动的连接到外网。

```
[root@node-2 ~]# iptables -L -n -t nat
.....省略
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.6.0/24      0.0.0.0/0
```

9.6 容器 None 网络模式添加网卡

Docker 的网络实现其实就是利用了 Linux 上的网络名字空间隔离技术和虚拟网络设备（veth pair）连接技术,实现容器对外部网络的访问。

9.6.1 创建容器流程回顾(桥接模式)

Docker 创建一个容器的时候，会执行如下操作:

创建一对虚拟接口(veth pair)，分别放到本地主机和新容器中;
本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，比如 veth65f9；容器一端放到新容器中，并修改名字作为 eth0，这个接口只在容器的名字空间可见；从网桥可用地址段中获取一个空闲地址分配给容器的 eth0，并配置默认路由到桥接网卡 veth65f9。完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

9.6.1.1 本案例实现的效果

为容器busybox容器添加自定义的网卡，容器可以通过网卡连接到外部网络。

9.6.2 操作过程

9.6.2.1 创建一个容器，设置网络模式是none

#启动一个 /bin/bash 容器，指定 --net=none 参数。

```
[root@demo ~]# docker run -it --name=demo-none  --net=none busybox
```

9.6.3 查看容器IP

#此时的容器是没有任何IP

```
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

9.6.4 获取容器id

#在本地主机查找容器的进程 id

```
[root@node-2 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                  PORTS                  NAMES
38078e9a638d busybox   "sh" 13 minutes ago Up 13 minutes                                  youthful_chandrasekhar
```

9.6.5 获取到容器的进程id

```
[root@node-2 ~]# docker inspect -f '{{.State.Pid}}' demo-none
13034
```

9.6.6 创建网络命名空间挂载点

```
[root@node-2 ~]# pid=13034

#创建网络空间挂载点(本地文件目录)
[root@node-2 ~]# mkdir -p /var/run/netns

#绑定网络命名空间到挂载点
[root@node-2 ~]# ln -s /proc/$pid/ns/net   /var/run/netns/$pid
```

9.6.7 查看需要添加的桥接器的网段与桥接器名称

```
#检查桥接网卡的 IP 和子网掩码信息。
[root@node-2 ~]# ip addr show bridge0
6: bridge0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.1/24 scope global bridge0                    #容器网关地址
       valid_lft forever preferred_lft forever
    inet6 fe80::10e7:beff:fe6c:aa1c/64 scope link
       valid_lft forever preferred_lft forever
```

9.6.8 建立veth-pair

```
namespace 之间需要通信，用 veth-pair 来做桥梁
创建一对“veth pair”接口 A 和 B，绑定 A 到网桥 bridge0，并启用。
A 为: veth 接口（宿主机）, B 为容器网卡
[root@node-2 ~]# ip link add A type veth peer name B

#设置 A veth 桥接到 bridge0
[root@node-2 ~]# brctl addif bridge0 A

#启动 veth A
[root@node-2 ~]# ip link set A up
```

9.6.9 设置网络与ip

#将B放到容器的网络命名空间，命名为 eth10，启动它并配置一个可用 IP（桥接网段）和默认网关。

```
#添加网卡 B 到网络命名空间 13034
[root@node-2 ~]# ip link set B netns $pid

#设置容器网卡的名称为 eth10
[root@node-2 ~]# ip netns exec $pid ip link set dev B name eth10
```

```
#设置网卡的状态为 up
[root@node-2 ~]# ip netns exec $pid ip link set eth10 up

#设置容器网卡的 ip 地址
[root@node-2 ~]# ip netns exec $pid ip addr add 192.168.6.100/24 dev eth10

#设置默认路由
[root@node-2 ~]# ip netns exec $pid ip route add default via 192.168.6.1
```

9.6.9.1 查看容器ip

```
#查看容器是否有 eth10
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
20: eth10@if21: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue qlen 1000
    link/ether 8e:6b:02:17:36:0a brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.100/24 scope global eth10
       valid_lft forever preferred_lft forever
```

9.6.9.2 查看宿主机网桥

```
[root@node-2 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
bridge0         8000.1651b5e3adba       no              A
docker0         8000.02422f4c4a2b       no
```

9.6.9.3 查看系统网卡

```
[root@node-2 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen
1000
    link/ether 00:0c:29:14:38:91 brd ff:ff:ff:ff:ff:ff
    inet 192.168.91.136/24 brd 192.168.91.255 scope global noprefixroute ens32
       valid_lft forever preferred_lft forever
    inet6 fe80::7dfd:e7a3:7683:20e7/64 scope link noprefixroute
```

```
        valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:2f:4c:4a:2b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:2fff:fe4c:4a2b/64 scope link
        valid_lft forever preferred_lft forever
6: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen
1000
    link/ether 16:51:b5:e3:ad:ba brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.1/24 scope global bridge0
        valid_lft forever preferred_lft forever
    inet6 fe80::10e7:beff:fe6c:aa1c/64 scope link
        valid_lft forever preferred_lft forever
21: A@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master bridge0 state UP
group default qlen 1000
    link/ether 16:51:b5:e3:ad:ba brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::1451:b5ff:fee3:adba/64 scope link
        valid_lft forever preferred_lft forever
```

9.7 运行容器 IP 添加地址

#拉取镜像 CentOS

```
[root@ demo ~]# docker pull centos
```

9.7.1 运行容器

#前台运行centos

```
[root@demo ~]# docker run --name=centos-IP1 -it centos /bin/bash
```

9.7.2 获取到容器启动进程号

```
[root@ demo ~]# docker inspect -f '{{.State.Pid}}' centos-IP1
13575
```

9.7.3 容器修改之前的ip

```
[root@95d78806950d /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:06:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

```
    inet 192.168.6.3/24 brd 192.168.6.255 scope global eth0
       valid_lft forever preferred_lft forever
```

9.7.4 创建容器网络命名空间挂载点

```
[root@ demo ~]# pid=13575

#无法直接操作必须要有网络空间挂载点
[root@demo ~]# ip netns exec $pid ip a
Cannot open network namespace "13575": No such file or directory

[root@ demo ~]# mkdir -p /var/run/netns
#创建网络空间挂载点
[root@ demo ~]# ln -s /proc/$pid/ns/net    /var/run/netns/$pid

#查看现在容器 IP
[root@ demo ~]# ip netns exec $pid ip a
```

9.7.5 设置容器新IP

#设置容器的ip 地址

```
[root@ demo ~]# ip netns exec $pid ip addr add 192.168.6.10/24 dev eth0
```

9.7.6 容器内部检验IP

#检验新的ip 地址是否添加

```
[root@ demo ~]# ip netns exec $pid ip a
```

9.8 总结 Docker 实现原理

1）底层硬件支撑，也可以用虚拟机运行Docker。
2）在硬件或虚拟机内安装支持容器运行的操作系统
3） Docker 通过NS、Cgroups与OverlayFs 技术实现容器的加载启动.
4）容器通过利用系统中的Docker0网桥与外部网络互通。

第 10 章 Kubernetes 介绍与部署

本章节主要讲为什么需要容器编排工具，Kubernetes 的基本特征以及怎样部署；在部署之后

会安装一些基本的组件，比如: DNS，管理界面与监控系统。

10.0 什么是容器编排?

概括：支持Docker在各个宿主机节点自动部署、自动扩容、负载均衡、滚动升级等等这些操作是通过编排工具去完成，支持这些功能的工具，称为容器编排。流行的开源容器编排工具包括Kubernetes、Docker Swarm以及Mesos等。

10.1 为什么需要容器编排?

1)应用程序如何实现透明化发布，比如：发布100个容器， 10个节点，那么10个节点应该怎么样分布这100个容器?
2)已经部署的程序，怎么样实现快速的扩容? 使用脚本? 还是自动化工具? 假设某个服务已经运行10个容器，但是根据用户访问情况反馈，系统访问速度慢。那么怎么样实现快速的扩容？甚至是秒级扩容?
3)一个应用已经运行100个容器，怎么做到从客户端的请求能够根据某种算法均衡分布到容器中?
.....

10.2 什么是 Kubernetes

Kubernetes (通常称为K8s，K8s 是将8个字母“ubernete”替换为“8”的缩写) 是一个以容器为中心的基础架构，可以实现在物理机集群或**虚拟机**集群上编排与运行容器，并且提供容器自动部署、扩展和管理的开源平台。满足了应用程序在生产环境中的一些实际需求：应用实例副本、水平自动扩展、命名与发现、负载均衡、滚动升级、资源监控等。

10.3 Kubernetes 发展史

Kubernetes 最初由 Google 的工程师开发和设计，并由谷歌在2014 年首次对外开源 。它的开发和设计都深受谷歌的Borg系统的影响，它的许多顶级贡献者之前也是Borg系统的开发者, Kubernetes v1.0 于2015 年7 月21 日发布。

版本发行历史:
https://github.com/kubernetes/kubernetes/releases

10.4 Kubernetes 特点

1) 自动化: 自动部署，自动重启，自动复制，自动伸缩/扩展。
2) 快速部署应用，快速扩展应用。
3) 无缝对接与部署新的应用功能。
4) 节省系统资源，优化硬件资源的使用。

10.5 Kubernets 编排流程

1）用户通过Yaml编排文件，向服务端的Api Server发起请求，需要建立新容器。
2）服务端Api Server会把新建立容器信息写入到Etcd中。
3）服务端Scheduler会根据调度策略，分配运行此容器的节点。
4）服务端Api Server会监听Scheduler哪些新的Pod的变化信息，并且更新状态到Etcd中.
5) 在服务端确定了哪些节点运行容器以后，会调用节点的Kuelet服务，通过Kuelet调用Docker拉取镜像,启动镜像.
6) 服务端Api Server也会监听Kuelet服务，实时更新Pod状态到etcd中

10.6 Kubernetes 核心组件介绍

Kubernetes 遵循master-slave architecture。Kubernetes 的组件可以分为管理单个的 node 组件和控制平面的一部分的组件。

Kubernetes Master 是集群的主要控制单元，用于管理其工作负载并指导整个系统的通信。Kubernetes 控制平面由**各自**的进程组成，每个组件都可以在单个主节点(Master)上运行，也可以在支持high-availability clusters 的多个主节点上运行。

Kubernetes 主要由以下几个核心组件组成:

etcd	保存了整个集群的状态与节点网段分配;
apiserver	提供了资源操作的唯一入口，并提供认证、授权、访问控制、API 注册和发现等机制;
controller manager	负责维护集群的状态，比如故障检测、自动扩展、滚动更新等;
scheduler	负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上;
kubelet	负责维护容器的生命周期，同时也负责Volume（CVI）和网络（CNI的管理;
kube-proxy	负责为Service 提供cluster 内部的服务发现和负载均衡;

核心组件结构图
#每个组件的基本介绍:

除了核心组件，还有一些其他的组件:

kube-dns 负责为整个集群提供DNS 服务
Ingress Controller 为服务提供外网入口
Heapster 提供资源监控
Dashboard 提供GUI
Federation 提供跨可用区的集群

10.7 Kubernetes 部署

省略.....

第 11 章 Kubernetes 核心概念

在上章完成基本的部署之后，但是对Kubernetes的组件还不能完全理解，尤其是各个组件的有哪些特征，用到什么位置，都不是很明确。所以本章节会详细讲解各个组件的核心概念。

11.0 kubectl 语法

```
$ kubectl [command] [TYPE] [NAME] [flags]
  kubectl get pod ninx -A
```

#参数解释:

1)**command**：子命令，用于操作Kubernetes集群资源对象的命令,比如create(创建)、delete(删除)、describe(描述)、get(获取)、apply(更新)等.

2)**TYPE**：资源对象的类型(pod、ns、svc、rs)，区分大小写，可以简写.

3)**NAME**：资源对象的名称,区分大小写.

4)**flags**：kubectl子命令的可选参数，例如使用"-A" 返回所有的资源。

11.1 Kubernetes 对象

服务网络访问: Ingress、Services
容器管理类型: Deployment、ReplicaSet、 ReplicationController、DaemonSet、StatefulSet
容器存储类型: Volume、Secret、ConfigMap、PersistentVolume

#逻辑层次

11.2 集群资源管理

11.2.0 Node

Node是kubernetes集群的工作节点，可以是物理机也可以是虚拟机

11.2.1 Namespace

节省资源的一种逻辑隔离技术

Namespace类似于Linux系统中用户的概念，通过将系统内部的对象分配到不同的Namespace中，形成逻辑上的区分，便于不同的分组在共享集群资源的同时还能被分别管理。同一Namespace下的Kubenetes对象的Name必须唯一。

命名空间为名称提供了一个范围。 资源的名称需要在**命名空间内是唯一**的，但不能跨命名空间。命名空间不能嵌套在另外一个命名空间内，而且每个 Kubernetes 资源只能属于一个命名空间。

默认情况下，Kubernetes 集群会在配置集群时实例化一个默认命名空间default，用以存放集群所使用的默认 Pods、Services 和 Deployments 集合。

注意:不需要使用多个命名空间来分隔轻微的且不同的资源，比如同一项目的不同发行版本。

11.2.1.1 案例:Namespace操作

#创建NS

```
[root@master-1 ~]# kubectl create namespace ns-system
```

#通过Yaml 文件创建

#test-ns.yaml

```
[root@master-1 test]# cat test-ns.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: test-ns
```

#查看NS列表

```
[root@master-1 ~]# kubectl get namespace
NAME                   STATUS   AGE
default                Active   6d12h
ns-system              Active   21s
```

#查看ns详细信息

```
[root@master-1 ns]# kubectl describe  namespace/test-ns
Name:         test-ns
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration: {"apiVersion":"v1","kind":"Namespace","metadata":{"annotations":{},"name":"test-ns"}}
Status:       Active
```

#NS与Service（10.0.0.0/8）关系

创建 Service 时，它会创建相应的 DNS记录。此记录的格式为 <servicename>.<namespace-name> .svc.cluster.local，这意味着如果容器只使用 <service-name>，它将解析为本地服务到命名空间(default)。 如果要跨命名空间访问，则需要使用完全限定的域名（FQDN）。

```
#dns 解析的名称是 svc (service 名称，非 pod 名称)
dnstools# nslookup nginx
Server:         10.0.0.2
Address:        10.0.0.2#53

Name:   nginx.default.svc.cluster.local
Address: 10.0.0.55
```

#非默认的default，需要写全名称

#nginx-n1.kube-system

```
dnstools# nslookup nginx-n1
Server:         10.0.0.2
Address:        10.0.0.2#53
** server can't find nginx-n1: NXDOMAIN
```

```
dnstools# nslookup nginx-n1
Server:         10.0.0.2
Address:        10.0.0.2#53

#解决方法(默认解析为 default 空间)
dnstools# nslookup nginx-n1.kube-system.svc.cluster.local
Server:         10.0.0.2
Address:        10.0.0.2#53

Name:   nginx-n1.kube-system.svc.cluster.local
Address: 10.0.0.196
```

11.2.2 Label

Label 机制是Kubernetes 中的一个重要设计，通过Label 进行对象(pod,service,node)的关联，可以灵活地进行分类和选择。对于Pod而言，需要设置其自身的Label 来进行标识，Label 是一系列的Key/value 对，在Pod-->metadata-->labels 中进行设置。Label 的定义是任一的，但是Label 必须具有可标识性，比如设置Pod 的应用名称和版本号等。另外Lable 是不具有唯一性的，为了更准确的标识一个Pod，应该为Pod 设置多个维度的label。如下：

"release" : "stable", "release" : "canary"

"environment" : "dev", "environment" : "uat", "environment" : "production"

"partition" : "customerA", "partition" : "customerB"

11.2.2.1 案例一：设置容器 Label

在 Job、Deployment、ReplicaSet 和 DaemonSet 中，通常在容器中使用 metadata.labels 字段，来为对象添加 Label，例如：

#配置标签

```
[root@master-1 ingress]# cat traefik-deploy.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: traefik-ingress-controller
  labels:
    app: traefik
```

#查看标签

```
[root@master-1 ingress]# kubectl describe pod   traefik-ingress-controller
Name:           traefik-ingress-controller-4cn9t
Namespace:      default
```

```
Priority:         0
Node:             192.168.91.21/192.168.91.21
Start Time:       Sat, 18 Apr 2020 16:02:08 +0800
Labels:           app=traefik
                  controller-revision-hash=846985d876
                  pod-template-generation=4
```

#界面查看

11.2.2.2 案例二: 设置Node Label

#设置Node的label标签为IngressProxy=true

```
[root@master-1 ingress]# kubectl label nodes 192.168.91.21 IngressProxy=true
```

#检查是否成功

```
[root@master-1 ingress]# kubectl get nodes --show-labels
[root@master-1 nginx]# kubectl get nodes -l IngressProxy=true
```

```
[root@master-1 ingress]#    kubectl label nodes 192.168.91.21 isNode=true
node/192.168.91.21 labeled

[root@master-1 ingress]# kubectl get nodes --show-labels
NAME              STATUS    ROLES     AGE     VERSION     LABELS
192.168.91.21     Ready     <none>    13d     v1.15.1
IngressProxy=true,beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,isNode=true,kubernetes.io/arc
h=amd64,kubernetes.io/hostname=192.168.91.21,kubernetes.io/os=linux
```

11.3 容器控制器

11.3.1 Pod (容器控制器)

Pod 是Kubernetes 的基本操作的最小单元，也是应用运行的载体。整个Kubernetes 系统都是围绕着Pod 展开的;比如,如何部署运行Pod、如何保证Pod的数量、如何访问Pod 等。

.

可以将 Pod 看作单个容器的包装，并且 Kubernetes 直接管理 Pod，而不是容器.

11.3.1.1 案例一:创建Nginx Pod

#删除之前的nginx pod

```
kubectl delete deployment nginx
kubectl delete pods nginx
```

```
kubectl delete svc -l run=nginx
kubectl delete deployment.apps/nginx
```

```
[root@master-1 nginx]# cat nginx.yaml
apiVersion: v1          #pod 接口
kind: Pod               #控制器名称 pod
metadata:
    name: nginx         #pod 名称
    labels:
      app: nginx        #pod  标签  app=nginx
spec:
  containers:
  - name: nginx         #容器名称
    image: nginx        #镜像名称
    ports:
    - containerPort: 80     #容器端口
      hostPort: 3000        #node  端口
```

#运行

```
[root@master-1 nginx1]# kubectl apply -f nginx.yaml
pod/nginx created
```

#查看运行状态

```
[root@master-1 nginx]# kubectl describe pod nginx
[root@master-1 nginx1]# kubectl describe pod/nginx
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                        {"apiVersion":"v1","kind":"Pod"
```

11.3.2 ReplicationController与ReplicaSet(容器控制器)

ReplicationController 用来确保容器应用的副本数始终保持在用户定义的**副本数**（容器数量），即如果有容器异常退出，会自动创建新的 Pod 来替代；而如果异常多出来的容器也会自动回收。

在新版本的Kubernetes中建议使用**ReplicaSet**来取代**ReplicationController**。ReplicaSet跟ReplicationController没有本质的不同，名称不同，并且ReplicaSet支持集合式（多标签选择）的selector。

11.3.2.1 案例一：创建ReplicationController

#通过**template.metadata.labels**字段为即将新建的Pod附加Label，即为app=nginx

#通过**spec.selector**字段来指定这个RC管理哪些Pod

```
[root@master-1 nginx-rc]# cat nginx-rc.yaml
```

```
apiVersion: v1
kind: ReplicationController   #容器控制器 RC
metadata:
  name: nginx-rc              #RC 名称
spec:
  replicas: 1                 #副本数 1
  selector:
    app: nginx-rc             #控制器选择 app=nginx-rc (label)
  template:
    metadata:
      labels:
        app: nginx-rc             #模板的 label: app=nginx
    spec:
      containers:
      - name: nginx-rc #容器名称
        image: nginx   镜像名称
        ports:
        - containerPort: 80 #容器端口
```

#部署RC

```
[root@master-1 demo]# kubectl apply -f nginx-rc.yaml
replicationcontroller/nginx-rc created
```

#查看状态

```
[root@master-1 nginx1]# kubectl describe rc nginx-rc
[root@master-1 nginx1]# kubectl describe replicationcontroller/nginx-rc
……..
Events:
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  56s   replication-controller  Created pod: nginx-rc-lwghm
```

#访问测试

```
[root@master-1 ~]# kubectl run -it --rm --restart=Never --image=infoblox/dnstools:latest dnstools
dnstools# curl 172.17.98.13
```

11.3.4 调整RC Pod 数量

```
[root@master-1 demo]#   kubectl scale rc nginx-rc --replicas=3
replicationcontroller/nginx-rc scaled
```

#RC容器控制器

#pod

#pod 不支持扩容(命令扩容)

```
[root@master-1 nginx1]#    kubectl scale pod nginx    --replicas=2
error: could not find proper group-version for scale subresource of /v1, Resource=pods: could not find scale subresource for /v1, Resource=pods in discovery information
```

#pod 不支持扩容(yaml)

```
[root@master-1 nginx1]# cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
    name: nginx          #pod 名称
    labels:              #label 标签
      app: nginx         #定义 pod 的 label: app=nginx
spec:
  replicas: 2 #  容器数量
  containers:
  - name: nginx    #容器名称
    image: nginx  #镜像名称
    ports:
    - containerPort: 80#容器端口
      hostPort: 3000         #node 接口端口
```

#出现错误

#接口不支持replicas

```
[root@master-1 nginx1]# kubectl apply -f    nginx.yaml
error: error validating "nginx.yaml": error validating data: ValidationError(Pod.spec): unknown field "replicas" in io.k8s.api.core.v1.PodSpec; if you choose to ignore these errors, turn validation off with --validate=false
```

11.3.2.2 案例二：创建ReplicaSet

selector除了可以使用matchLabels，还支持集合式的操作。

#通过修改.spec.replicas的值可以实时修改RS运行的Pod数量。

#镜像下载有问题

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
```

```
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        #定义多个 Label
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 80
```

#查看是否启动成功

```
[root@master-1 ~]# kubectl describe rs/frontend
```

11.3.3 Deployment

Deployment同样为Kubernetes的一个核心内容，主要职责同样是为了保证pod的数量和健康，90%的功能与Replication Controller完全一样，可以看做新一代的Replication Controller。但是，它又具备了Replication Controller之外的新特性：

1) Deployment继承了上面描述的Replication Controller(副本数保持与扩容)全部功能。
2) 容器在升级或者创建的过程中，可以查看详细进度和状态。
3) 当升级pod镜像或者相关参数的过程中，如果出现问题，可以使用回滚操作回滚到上一

个版本或者指定的版本。

4) 每一次对Deployment的操作，都能保存下来（有操作记录），便于之后的回滚.
5) 每一次pod升级，都能够随时暂停和启动（容易控制）。
6) pod多种升级方案：
   a) Recreate：删除所有已存在的pod,重新创建新的;
   b) RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用pod数量，最小升级间隔时间等等。

#容器控制器创建pod流程

1.使用Deployment来创建ReplicaSet,ReplicaSet则在在后台创建pod;检查pod启动状态,是否成功。

2.通过更新Deployment的PodTemplateSpec字段来声明Pod的新状态。之后会创建一个新的ReplicaSet，

Deployment会按照默认的更新策略(RollingUpdate 所需副本数的25%，生产建议配置为:1)先启动新的ReplicaSet.

3.如果新的ReplicaSet启动失败（检查端口策略），则旧的ReplicaSet还会继续存在。

4.如果扩容成功，则会清理掉旧的ReplicaSet

#系统查看Deployment

```
[root@master-1 ~]# kubectl   get Deployment
NAME                          READY    UP-TO-DATE    AVAILABLE    AGE
nfs-client-provisioner    1/1       1                1              6d22h
redis-tcp                      1/1       1                1              96m
```

#查看Deployment详细信息

```
[root@master-1 ~]# kubectl describe pod   redis-tcp
Name:                redis-tcp-5fcb54cd64-b5fsx
Namespace:           default
Priority:            0
Node:                192.168.91.22/192.168.91.22
Start Time:          Sun, 19 Apr 2020 12:35:35 +0800
Labels:              app=redis-tcp
                     pod-template-hash=5fcb54cd64
Annotations:         <none>
Status:              Running
IP:                  172.17.20.6
Controlled By:   ReplicaSet/redis-tcp-5fcb54cd64
```

11.3.3 Pod 升级策略与检测机制

11.3.3.1 升级策略:

```
  strategy:
    type: RollingUpdate      #默认更新策略Deployment
    rollingUpdate:
      maxSurge: 1 # Pod的总数最多比所需的Pod数多1个
      maxUnavailable: 1 # 更新过程中最多1个不可用Pod数
```

11.3.3.2 pod 启动与存活机制

#pod 就绪探测 (Deployment控制器)

```
    readinessProbe: #来确定容器是否已经启动正常
      tcpSocket:
        port: 8080 #检查端口
      initialDelaySeconds: 5 # kubelet 会在容器启动 5 秒后发送第一个就绪探测
      periodSeconds: 10 #每 10 秒执行一次存活探测
```

如果探测成功，这个 Pod 会被标记为就绪状态，kubelet 将继续每隔 10 秒运行一次检测。

#pod 存活探测 (Deployment控制器)

```
    livenessProbe: #存活探测
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15 #kubelet 会在容器启动 15 秒后进行第一次存活探测
      periodSeconds: 20 #每 10 秒执行一次存活探测
```

与就绪探测一样，Kubelet会尝试连接容器的 8080 端口。如果存活探测失败，这个容器会被重新启动。

11.3.3.1 案例一: 创建Deployment

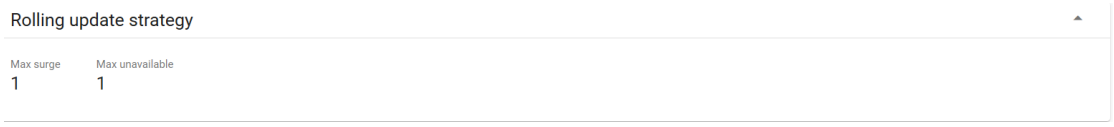
```
[root@master-1 demo]# cat nginx-deployment.yaml
apiVersion: apps/v1      #控制器接口版本
kind: Deployment         #控制器类型
metadata:
  name: nginx-deployment # Deployment 控制器名称
  labels:
    app: nginx-deployment # Deployment 控制器标签
spec:
  replicas: 3                    # Deployment 副本数(容器数量)
  selector:
    matchLabels:
```

```
      app: nginx-deployment    #标签选择器
  minReadySeconds: 1           #pod  准备时间
  strategy:
    type: RollingUpdate        #pod 升级策略
    rollingUpdate:
      maxSurge: 1              #最大允许 1 个 pod 升级
      maxUnavailable: 1        #最多允许 1 个 pod 失效
  template:
    metadata:
      labels:
        app: nginx-deployment#模板标签
    spec:
      containers:
      - name: nginx-deployment     #容器名称
        image: nginx:latest        #镜像名称
        ports:
        - containerPort: 80      #容器端口
        readinessProbe:                #容器就绪检测
          tcpSocket:
            port: 80                   #检测端口
          initialDelaySeconds: 5#容器启动之后 5 秒检测
          periodSeconds: 10            #容器初始化之后每隔 10 秒检测
```

#部署

```
[root@master-1 demo]# kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

#界面查看更新策略



#查看状态

```
[root@master-1 demo]# kubectl get pods -w
[root@master-1 demo]# kubectl describe deployment.apps/nginx-deployment
```

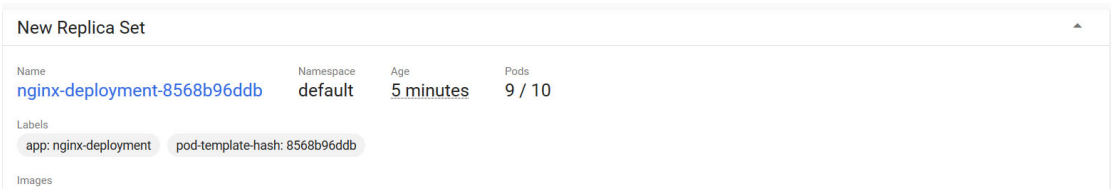
#扩容

#scale 支持容器控制器类型Deployment, ReplicaSet, Replication Controller, Job

```
[root@master-1 ~]# kubectl scale deployment nginx-deployment --replicas 10
```

#查看Deployments

#查看RS控制器



#pod 正在扩容创建

#修改配置文件，检测更新策略
#把镜像修改为不存在的镜像
image: nginx111111:latest

```
[root@master-1 ~]# kubectl apply -f nginx-deployment.yaml
```

#会创建一个新的RS

#新的RS关联的pod出错

#根据更新的策略，还会存在2个旧的pod，因为应用了新的配置文件，所以原来的10个pod，只会留下3个，其中1个正在更新,所以只能看到两个pod是正常的.

#查看更新的过程

```
[root@master-1 ~]# kubectl describe    deployment nginx-deployment
  ----              ------  ------
  Available         True    MinimumReplicasAvailable
  Progressing       False   ProgressDeadlineExceeded
OldReplicaSets:   nginx-deployment-8568b96ddb (2/2 replicas created)
NewReplicaSet:    nginx-deployment-65cf75487b (2/2 replicas created)
Events:
  Type    Reason              Age                  From                   Message
```

```
  ----     ------                  ----                  ----                    -------
  Normal   ScalingReplicaSet       21m                   deployment-controller   Scaled up replica set nginx-deployment-8568b96ddb to 3
  Normal   ScalingReplicaSet       15m                   deployment-controller   Scaled down replica set nginx-deployment-8568b96ddb to 1
  Normal   ScalingReplicaSet       14m (x2 over 16m)     deployment-controller   Scaled up replica set nginx-deployment-8568b96ddb to 10
  Normal   ScalingReplicaSet       11m                   deployment-controller   Scaled down replica set nginx-deployment-8568b96ddb to 3
  Normal   ScalingReplicaSet       11m                   deployment-controller   Scaled up replica set nginx-deployment-65cf75487b to 1
  Normal   ScalingReplicaSet       11m                   deployment-controller   Scaled down replica set nginx-deployment-8568b96ddb to 2
  Normal   ScalingReplicaSet       11m                   deployment-controller   Scaled up replica set nginx-deployment-65cf75487b to 2
```

#删除nginx

```
[root@master-1 ~]# kubectl delete deployment.apps/nginx-deployment
[root@master-1 ~]# kubectl delete -f nginx-deployment.yaml
```

11.3.4 DaemonSet（容器控制器）

一个 DaemonSet 对象能确保其创建的 Pod 在集群中的每一台（或指定 Label）Node 上都运行一个副本。如果集群中动态加入了新的 Node，DaemonSet 中的 Pod 也会被添加在新加入 Node 上运行。删除一个 DaemonSet 也会级联删除所有其创建的 Pod。

使用场景:

1) 运行集群存储 daemon，例如在每个 Node 上运行 glusterd、ceph。
2) 在每个 Node 上运行**日志收集** daemon，例如fluentd、logstash。

3) 在每个 Node 上运行**监控 daemon**，例如 Prometheus Node Exporter

11.3.4.1 案例一：创建DaemonSet

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: log-pilot
  namespace: kube-system
  labels:
    k8s-app: log-pilot
    kubernetes.io/cluster-service: "true"
spec:
  template:
    metadata:
      labels:
        k8s-app: log-es
```

```
        kubernetes.io/cluster-service: "true"
        version: v1.22
```

log-pilot

#每个节点运行一个pod

11.3.5 StatefulSet

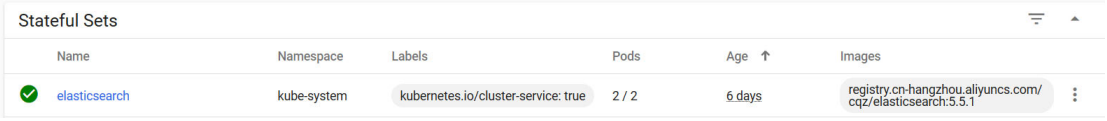
Kubernetes从v1.4版本开始引入了PetSet这个新的资源对象，并且在v1.5版本时更名为StatefulSet，从本质上来讲，可以看作Deployment/RC的一个特殊类型。

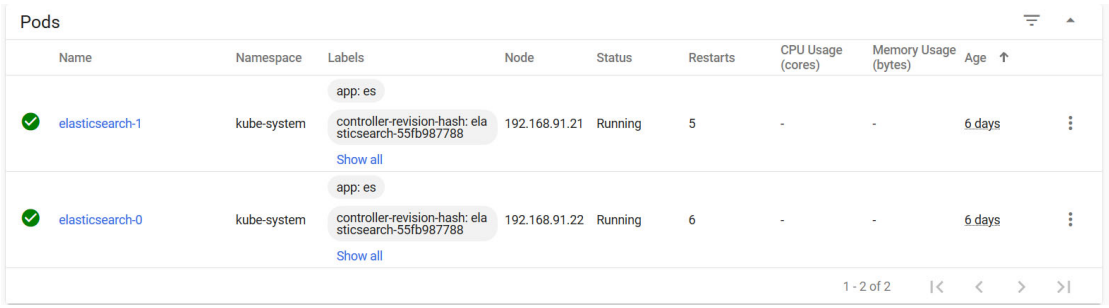
StatefulSet 适合有状态的服务； StatefulSet有唯一的pod名称标识，支持**持久化存储(集群需要存储数据)**与滚动更新。

假设StatefulSet运行集群的名字叫elasticsearch，那么第一个Pod叫elasticsearch -0，第二个Pod叫elasticsearch -1，以此类推。正是因为有序的pod 的名称，才能够组件集群(需要安装DNS)。

11.3.5.1 案例一：StatefulSet组建 Elasticsearch 集群

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: elasticsearch
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 2
  serviceName: "elasticsearch-service"
  selector:
    matchLabels:
      app: es
  template:
    metadata:
      labels:
        app: es
```





#资源请求和限制

请求和限制是Kubernetes用于控制CPU、Memory与磁盘等资源的机制。

#Kubernetes 字段

1) Requests 请求是保证容器能够得到的资源。 如果容器请求资源，Kubernetes会将其调度到可以为其提供该资源的节点上。

   requests用于schedule阶段，在调度pod保证所有pod的requests总和小于node能提供的计算能力

   requests.cpu被转成docker的--cpu-shares参数，与cgroup cpu.shares功能相同；该参数在CPU资源不足时生效，根据容器requests.cpu的比例来分配cpu资源，当CPU资源充足时，requests.cpu不会限制container占用的最大值，container可以独占CPU

   requests.memory没有对应的docker参数，作为k8s调度依据，使用requests来设置各容器需要的最小资源

2) Limits 限制则是确保容器使用的资源不会超过限制值。 容器只允许达到限制设定的资源值，无法获得更多资源。

limits.cpu 会被转换成docker的–cpu-quota参数。与cgroup cpu.cfs_quota_us功能相同
限制容器的最大CPU使用率.

limits.memory 会被转换成docker的–memory参数。用来限制容器使用的最大内存
当容器申请内存超过limits时会被终止

3) 设置单位

#cpu表示方法

spec.containers[].resources.requests.cpu 为 0.5 的容器保证了一半 CPU 要求 1 CPU的一半

表达式 0.1 等价于表达式 100m，可以看作“100 millicpu”。“一百毫 cpu”

#内存表示方法

内存的限制和请求以字节为单位，128974848, 129M（1000）, 123Mi（1024）

#注

“GiB”、“KiB”、“MiB”等是于1999年由国际电工协会（IEC）拟定了“KiB”、“MiB”、“GiB”的二进制单位，专用来标示“1024进位”的数据大小。而GB是以1000进位的数据单位。

11.3.6 POD基本操作

Kubernetes中Pod常用命令

#创建资源对象

根据yaml配置文件一次性创建service和rs:

#kubectl create -f nginx.yaml -f nginx-svc.yaml

1)根据<directory>目录下所有.yaml、.yml、.json文件的定义进行创建操作:

#kubectl create -f <directory>

#更新资源对象

#kubectl apply -f <directory> 或者 yaml 文件

#查看资源对象

1)查看所有Pod列表(default命名空间):

#kubectl get pods

1)查看pod所在的物理节点

#kubectl get pods -o wide

2)查看rc和service列表:

#kubectl get rc,svc

3)查看pod 日志
#需要控制台输出日志
#kubectl log -f yourPodName

#查看最新20行日志
#kubectl logs --tail=20 nginx

4)获取pod详细信息
#kubectl describe pod nginx

#资源对象操作
1)进入到pod
kubectl exec nginx-demo-68749b58dc-rr9rj -i -t bash -n default

2)查看pod 内容(非进入)
#kubectl exec nginx-deployment-8568b96ddb-psr74 -n default -i -t -- cat /etc/resolv.conf
#kubectl exec coredns-66db855d4d-26bvw -n kube-system -i -t -- cat /etc/resolv.conf

3)监听pods状态
#kubectl get pods -w

#删除容器与对象
#kubectl delete -f ./
#kubectl delete rs nginx-deployment-8568b96ddb-psr74

11.3.7 POD弹性伸缩
注意：POD 所支持弹性伸缩的类型: Deployment, ReplicaSet, ReplicationController

弹性伸缩是指适应负载变化，以弹性可伸缩的方式提供资源。反映到Kubernetes 中，指的是可根据负载的高低动态调整Pod 的副本数量。调整Pod 的副本数是通过修改RC 中Pod 的副本是来实现的。(HPA章节详细讲解)

11.3.8 POD的管理方式

11.3.9 pod 状态信息
1) 挂起（Pending）：Pod 已被 Kubernetes 系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括调度 Pod 的时间和通过网络下载镜像的时间，这可能需要花点时间。
2) 运行中（Running）：该 Pod 已经绑定到了一个节点上，Pod 中所有的容器都已被创

建。至少有一个容器正在运行，或者正处于启动或重启状态。

3) 成功（Succeeded）：Pod 中的所有容器都被成功终止，并且不会再重启。

4) 失败（Failed）：Pod 中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非0状态退出或者被系统终止。

5) 未知（Unknown）：因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败。

11.3.9.1 查看pod 运行状态

```
[root@master-1 job]# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
demo-app-57756548d8-sltsg           1/1     Running   1          3d11h
dnstools                            1/1     Running   0          122m
nfs-client-provisioner-5978c5f69c-k9lkb   1/1     Running   1          5d
```

1.3.9.2 POD重启策略

当某个容器异常退出或者健康检查失败, kubelet将根据RestartPolicy的设置来进行相应的操作,重启策略有Always , OnFailure, Never

1) Always: 当容器失效时，由kubelet自动重启该容器

2) OnFailure: 当容器终止运行且退出码不为0时，由kubelet自动重启该容器

3) Never: 不论容器运行状态如何, kubelet都不会重启该容器

#重启策略使用场景

Deployment、RC和DaemonSet：必须设置为Always，需要保证该容器持续运行。

Job和CronJob：OnFailure或Never，确保容器执行完成后不再重启。

kubelet：在Pod失效时自动重启它，不论将RestartPolicy设置为什么值，也不会对Pod进行健康检查。

1.3.9.2.1 配置pod重启策略

#RC控制器和DeamonSet控制器建议设置为Always,需要保证该容器持续运行

#Job: OnFailure或Never，确保容器执行完成后不再重启

```
[root@linux-master1 ~]# vim nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
  labels:
    app: nginx
spec:
  selector:
```

```
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      restartPolicy: Always  #容器重启策略
      containers:
      - name: nginx
        image: nginx:1.12
        imagePullPolicy: IfNotPresent      #镜像拉取策略
        ports:
        - containerPort: 80
```

11.4 镜像策略

默认的镜像拉取策略是“IfNotPresent”，在镜像已经存在的情况下，kubelet将不在去拉取镜像。 如果总是想要拉取镜像，必须设置拉取策略为“Always”或者设置镜像标签为“:latest”。如果没有指定镜像的标签，它会被假定为“:latest”,同时拉取策略为“Always”.
#在生产环境中，运行应用的时候(微服务)，建议设置为Always

注意在拉取自建私有仓库镜像时，需要通过创建Secret，才能拉取镜像。

在kubernetes 中有以下三种下载策略：
Always：　　每次都下载最新的镜像　　#微服务建议用此项
Never：　　只使用本地镜像，从不下载　　#单节点
IfNotPresent：　　只有当本地没有的时候才下载镜像 #开源的第三方的应用建议用此项

11.4.1 案例一：配置镜像拉取策略

```
[root@linux-master1 ~]# vim nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
  labels:
    app: nginx-deployment
spec:
```

```
  selector:
    matchLabels:
      app: nginx-deployment
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-deployment
    spec:
      restartPolicy: Always
      containers:
      - name: nginx-deployment
        image: nginx:1.12
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
```

11.5 定时任务

11.5.1 Job

Job 负责批处理任务，即仅**执行一次**的任务，它保证批处理任务的一个或多个 Pod 成功结束。

使用场景:清理数据，同步数据

11.5.1.1 案例一: Job

Job 配置文件，只包含一个 pod，输出圆周率小数点后 2000 位，运行时间大概为 10s

```
[root@master-1 test]# kubectl create -f job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

```
  backoffLimit: 4
```

#运行

```
[root@master-1 test]# kubectl create -f job.yaml
```

#查看任务

```
[root@master-1 job]# kubectl describe jobs/pi
Name:                pi
Namespace:           default
Selector:            controller-uid=a7e9d6cd-9b57-4864-9a87-40d864cfe34b
Labels:              controller-uid=a7e9d6cd-9b57-4864-9a87-40d864cfe34b
                     job-name=pi
Annotations:         kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"batch/v1","kind":"Job","metadata":{"annotations":{},"name":"pi","namespace":"defa
ult"},"spec":{"backoffLimit":4,"template":...
Parallelism:         1
Completions:         1
Start Time:          Sun, 08 Dec 2019 23:23:58 +0800
Pods Statuses:       1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=a7e9d6cd-9b57-4864-9a87-40d864cfe34b
           job-name=pi
  Containers:
```

#导出镜像

```
[root@node-1 ~]# docker save aebad195b013 -o perl.tar
```

#分发镜像到node-2节点

```
[root@node-1 ~]# scp perl.tar node-2:/root/
```

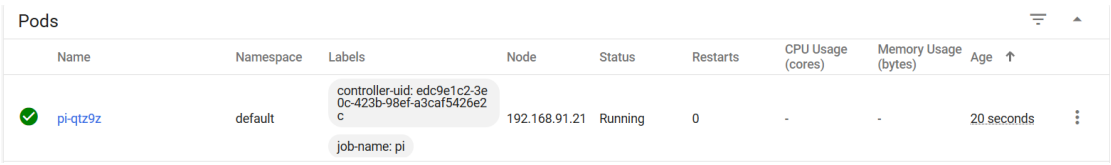
#导入镜像

```
[root@node-2 ~]# docker load < perl.tar
```

#获取job

```
[root@master-1 test]#pods=$(kubectl get pods \
--selector=job-name=pi --output=jsonpath={.items..metadata.name})

[root@master-1 ingress]# echo $pods
pi-qtz9z
```



#获取结果

```
[root@master-1 job]# kubectl get pods | grep $pods
pi-t596w                                  0/1     Completed   0          3m29s
[root@master-1 job]# kubectl logs $pods
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612 8
.....
```

#容器的状态为Completed

```
[root@master-1 ingress]#    kubectl get pods | grep $pods
pi-qtz9z                                  0/1     Completed   0          94s
```

11.5.2 CronJob

CronJob是用来管理基于时间的Job，即:

1) 在给定的时间点运行一次Job

2) 周期性的在给定的时间点运行

3）Crontab的基本格式

<分钟> <小时> <日> <月份> <星期> <命令>

分钟 值从 0 到 59.

小时 值从 0 到 23.

日 值从 1 到 31.

月 值从 1 到 12.

星期 值从 0 到 6, 0 代表星期日

多个时间可以用逗号隔开，范围可以用连字符给出，*可以作为通配符。空格用来分开字段。

11.5.2.1 案例一：创建定时任务

#定时清理数据

#每个一分钟echo “Hello from the Kubernetes cluster”

```
[root@master-1 test]#cat cronjob.yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
```

```
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

#创建任务

```
[root@master-1 test]#kubectl create -f cronjob.yaml
```

#获取任务列表

```
[root@master-1 job]# kubectl get CronJob
NAME      SCHEDULE      SUSPEND     ACTIVE     LAST SCHEDULE     AGE
hello     */1 * * * *     False       0          <none>            17s
```

#获取任务

```
[root@master-1 job]# kubectl get jobs
NAME                     COMPLETIONS     DURATION     AGE
hello-1575819060     1/1             9s           38s
```

#获取job

```
[root@master-1 job]# pods=$(kubectl get pods --selector=job-name=hello-1575819060 --output=jsonpath={.items..metadata.name})
[root@master-1 job]# echo $pods
hello-1575819060-ql48t
```

#获取结果

```
[root@master-1 job]# kubectl logs $pods
Sun Dec  8 15:31:12 UTC 2019
Hello from the Kubernetes cluster
```

#通过界面查看任务

#定时任务输出的结果

#删除Cron Job

```
[root@master-1 test]# kubectl delete cronjob hello
```

11.6 服务发现

11.6.1 Service

此Service不能直译，在kubernetes中为了更好的定位每一个或者每组POD (Label 标签），才出现了Service 的概念。因为我们知道POD 的IP地址是在每次重启或者是容器重新部署之后是有变化的。而且容器的IP 地址是无法固定的，所以需要Service来定位POD。

11.6.1.1 案例一: 定义服务的Service

#通过节点的Nodeport 端口访问应用

```
[root@master-1 java]# cat provider-passport-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: passport-svc #Service 名称
spec:
  type: NodePort        #Service 端口类型,默认为 ClusterIP
  ports:
  - port: 8081          #Service (ClusterIP)端口
    targetPort: 8081    #容器端口
    protocol: TCP       #端口协议
  selector:             #根据 Label 选择容器控制器
    app: passport
```

11.6.2 Service 网络代理模式

用户通过Service访问服务的时候，实际是通过node节点上kube-proxy访问pod服务. 在 Kubernetes v1.2 中，kube-proxy 的 iptables 模式成为默认设置。 Kubernetes v1.8 添加了 ipvs 代理模式。

1) Iptables模式

1、Node节点客户端直接访问ServiceIP，Linux根据Iptables协议栈规则策略匹配。

2、ServiceIP根据标签直接访问Backend Pod。

2) IPVS 代理模式

IPVS是LVS一个组件，提供高性能、高可靠性的四层负载均衡器。IPVS 是IP Virtual Server的简写。IPVS构建在netfilter上，作为Linux 内核的一部分，从传输层实现了负载均衡。IPVS能直接转发 基于Services的TCP 和UDP到真实服务器。IPVS能直接构建一个VIPs 并通过负载均衡算法，把请求转发到backend Pod。

IPVS支持的负载均衡算法：

rr: 轮询
lc: 最小连接数
dh: 目的地址hash
sh: 源地址hash
sed: 最短期望延迟
nq: 无须队列等待

11.6.3 Service服务类型

1) ClusterIP：通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。

2) NodePort：通过每个 Node 上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求 <NodeIP>:<NodePort>，可以从集群的外部访问一个 NodePort 服务。

3) LoadBalancer：使用**云提供商的负载局衡器**，可以向外部暴露服务。外部的负载均衡器可以路由到 NodePort 服务和 ClusterIP 服务。

11.6.4 Service 端口类型

port: service的的端口
targetport: pod也就是容器的端口
nodeport: 容器所在宿主机的端口（实质上也是通过service暴露给了宿主机）与Service**服务类型**为NodePort结合使用(注意端口范围).还要区别与容器控制器的**hostPort**

11.6.5 访问Service方式

Service 的虚拟IP 是由Kubernetes 虚拟出来的内部网络(默认是Iptables实现)，外部是无法寻址到的。但是有些服务又需要被外部访问到，例如web前端。这时候就需要添加一层网

络转发，即外网到内网的转发。Kubernetes 提供了NodePort、LoadBalancer、Ingress 三种方式访问Service。
访问方式图解：

1） NodePort， Kubernetes 会在每一个Node 上暴露出一个端口：NodePort，外部网络可以通过 [NodeIP]:[NodePort]访问到后端的Service。

2） LoadBalancer，在NodePort 基础上，Kubernetes 可以请求底层云平台创建一个负载均衡器，将每个Node作为后端，进行服务分发。该模式需要底层云平台（例如GCE）支持。

3） Ingress，是一种HTTP 方式的路由转发机制，由Ingress Controller 和HTTP 代理服务器组合而成。Ingress Controller 实时监控Kubernetes API，实时更新HTTP 代理服务器的转发规则。HTTP 代理服务器有GCE Load-Balancer、HaProxy、Nginx 等开源方案。

#Kubernetes 访问 Pod 中服务的方式

1) hostNetwork（Pod）
在 Pod 中使用 hostNotwork:true 配置,在 Pod 中可以看到所在 Node 节点的 IP 地址.适合集群组建.Pod 的数量等于 Node 数量. Debug 使用）

2) hostPort（Pod）
在 Pod 中配置 hostPort，那么在 Node 中将占用一个端口，Pod 的数量等于 Node 数量.（Debug 使用）

3) NodePort（SVC）
SVC 占用 Node 节点的端口,可以指定也可以随机生成.

4) LoadBalancer（SVC）
公有云提供的负载均衡器，如 AWS、Azure、腾讯、GCE.

5) Ingress （SVC）
4-7 层负载均衡，典型代表:Ingress Controller 与 Traefik

11.7 DNS

DNS 服务器监视着创建新 Service （SVC） 的 Kubernetes API，从而为每一个 Service 创建一组DNS记录。 如果整个集群的 DNS 一直被启用，那么所有的 Pod 应该能够自

对 Service 进行名称解析。

例如，有一个名称为 "my-service" 的 Service，它在 Kubernetes 集群中名为 "my-ns" 的 Namespace 中，为 "my-service.my-ns" 创建了一条 DNS 记录。 在名称为 "my-ns" 的 Namespace 中的 Pod 应该能够简单地通过名称查询找到 "my-service"。

11.7.1 案例一: 通过Service名称访问服务

#获取Service名称

```
[root@master-1 job]# kubectl get svc
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)      AGE
demo-app                ClusterIP      10.0.0.220     <none>          8080/TCP     3d11h
kubernetes              ClusterIP      10.0.0.1       <none>          443/TCP      9d
```

#访问服务

#服务访问的正确格式: SERVICE_NAME.NS

```
[root@master-1 job]# kubectl run -it --rm --restart=Never --image=infoblox/dnstools:latest dnstools
dnstools# curl demo-app.default:8080
Hello! My name is demo-app-57756548d8-sltsg. I have served 2279 requests so far.
```

注意:Service 在没有修改的情况下，是不会修改的; 而cluster ip 在每次重建Service的时候，会修改的。所以建议使用service name 连接内部服务。这也是为什么需要安装DNS的原因。

11.8 存储- Secret

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

Secret有三种类型:

1）Service Account ： 用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的/run/secrets/kubernetes.io/serviceaccount目录中;

2）Opaque ： base64编码格式的Secret，用来存储密码、密钥等;

3）kubernetes.io/dockerconfigjson ： 用来存储私有docker registry的认证信息

11.10.1 案例一: 导入Docker Registry认证信息

#在Kubernetes中需要导入Docker私有镜像仓库的登录信息，才能从私有仓库拉取镜像，登录信息的存储使用Secret加密存储。

```
#创建 secret 密钥
#创建 secret (注意解析 docker-server （hosts）)
[root@master-1 ~]# kubectl create secret docker-registry registry-secret --docker-server=reg.abc.com --
```

```
docker-username=user --docker-password=Aa123456 --docker-email=user@abc.com

#下载密钥
[root@master-1 ~]# kubectl get secret registry-secret -o yaml
apiVersion: v1
data:
  .dockerconfigjson:
eyJhdXRocyI6eyJyZWcuYWJjLmNvbSI6eyJ1c2VybmFtZSI6InVzZXIiLCJwYXNzd29yZCI6IkFhMTIzNDU2IiwiZW1haWwiOiJ1c2VyQGFiYy5jb20iLCJhdXRoIjoiZFhObGNqcEJZVEV5TXpRMU5nPT0ifX19
kind: Secret
metadata:
  creationTimestamp: 2019-06-27T13:02:44Z
  name: registry-secret
  namespace: default
  resourceVersion: "1231172"
  selfLink: /api/v1/namespaces/default/secrets/registry-secret
  uid: dacbce87-98db-11e9-b5f4-000c292ae7f4
type: kubernetes.io/dockerconfigjson

#修改为需要导入的格式
apiVersion: v1
kind: Secret
metadata:
  name: regis-secre
  namespace: default
data:
  .dockerconfigjson:
eyJhdXRocyI6eyJyZWcuYWJjLmNvbSI6eyJ1c2VybmFtZSI6InVzZXIiLCJwYXNzd29yZCI6IkFhMTIzNDU2IiwiZW1haWwiOiJ1c2VyQGFiYy5jb20iLCJhdXRoIjoiZFhObGNqcEJZVEV5TXpRMU5nPT0ifX19
type: kubernetes.io/dockerconfigjson

#创建 Secret
kubectl create -f registry-secret.yaml

[root@master-1 ~]# cat registry-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: regsecret
  namespace: default
data:
  .dockerconfigjson:
```

eyJhdXRocyI6eyJyZXBvLmhvc3RzY2MuY29tIjp7InVzZXJuYW1lIjoidXNlciIsInBhc3N3b3JkIjoiQWExMjM0NTY3OCIsImVtYWlsIjoidXNlckBhYmMuY29tIiwiYXV0aCI6ImRYTmxjanBCWVRFeU16UTFOamM0In19fQ==

type: kubernetes.io/dockerconfigjson

11.9 基于角色的访问控制- RBAC

基于角色（Role）的访问控制（RBAC）是一种基于企业中用户的角色来调节控制对计算或网络资源的访问方法。

RBAC使用rbac.authorization.k8s.io API Group 来实现授权决策，允许管理员通过Kubernetes API 动态配置策略.

启用RBAC，需要在 apiserver 配置文件中添加参数--authorization-mode=RBAC

11.9.1 RBAC 可操作资源列表

Pods	#容器
ConfigMaps	#配置
Deployments	#容器控制器
Nodes	#节点
Secrets	#凭证
Namespaces	#命名空间

11.9.2 资源列表可操作的权限列表

Create、get、delete、list、update、edit、watch、exec

资源列表与权限关系:

11.9.3 RBAC中的其他对象

1) Rule：规则，规则是一组属于不同 API Group 资源上的一组操作的集合

2) **Role 和 ClusterRole**：角色和集群角色，这两个对象都包含上面的 Rules 元素，二者的区别在于，在 **Role** 中，定义的规则只适用于**单个命名空间**，也就是和 namespace 关联的，而 **ClusterRole** 是集群范围内的，因此定义的**规则不受命名空间**的约束。

1) Subject：主题，对应在集群中尝试操作的对象，集群中定义了3种类型的主题资源：
   a) **User Account**：用户，这是有外部独立服务进行管理的，管理员进行私钥的分配，用户可以使用 KeyStone或者 Goolge 帐号，甚至一个用户名和密码的文件列表也可以。对于用户的管理集群内部没有一个关联的资源对象，所以用户不能通过集群

内部的 API 来进行管理

b) Group：组，这是用来关联多个账户的，集群中有一些默认创建的组，比如cluster-admin

c) Service Account：服务帐号，通过Kubernetes API 来管理的一些用户帐号，和 namespace 进行关联的，适用于集群内部运行的应用程序，需要通过 API 来完成权限认证，所以在集群内部进行权限操作，都需要使用到 ServiceAccount.

2) RoleBinding 和 ClusterRoleBinding：角色绑定和集群角色绑定，简单来说就是把声明的 Subject 和我们的 Role 进行绑定的过程(给某个用户绑定上操作的权限)，二者的区别也是作用范围的区别：RoleBinding 只会影响到当前 namespace 下面的资源操作权限，而 ClusterRoleBinding 会影响到所有的 namespace。

11.9.3.1 授权角色绑定图解
Service Account 集群的内部的鉴权.

11.9.4 案例一:创建登录Dashboard的用户

创建登录dashboard的用户，并且只能操作kube-system这个命名空间下面的 pods 和 deployments，所以需要创建ServiceAccount对象.

#创建SA对象

```
[root@master-1 ~]#    kubectl create sa demo-sa -n kube-system
```

#创建角色kube-role

```
[root@master-1 ~]# cat kube-sa-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
   name: kube-sa-role
   namespace: kube-system
rules:
- apiGroups: [""]                          #找不到资源 pods 对应的 apiGroup
   resources: ["pods"]                     #可以操作的资源
   verbs: ["get", "watch", "list"]         #对资源可以操作的列表
```

```
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

#创建角色

```
[root@master-1 ~]# kubectl   apply -f kube-sa-role.yaml
role.rbac.authorization.k8s.io/kube-sa-role created

#查询角色
[root@master-1 ~]# kubectl   get role -n kube-system
```

#常用的apiGroups列表查阅:
https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/

```
[root@master-1 ~]# cat kube-sa-rolebinding.yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: kube-sa-rolebinding
  namespace: kube-system
subjects:
- kind: ServiceAccount
  name: demo-sa
  namespace: kube-system
roleRef:
  kind: Role
  name: kube-sa-role
  apiGroup: rbac.authorization.k8s.io
```

ServiceAccount 会生成一个 Secret 对象和它进行映射，这个 Secret 里面包含一个 token，如果dashboard也时属于kube-system，则这个token 可以用来登录到dashboard

#创建角色绑定

```
[root@master-1 ~]# kubectl   apply -f kube-sa-rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/kube-sa-rolebinding created
```

#获取base64

```
[root@master-1 ~]# kubectl get secret $( kubectl get secret -n kube-system |grep demo-sa | awk '{print $1}') -o jsonpath={.data.token} -n kube-system |base64 -d
```

#登录dashbosh

#输入kube-system 命名空间

11.9.5 案例二:创建一个不受限制的SA账号，用于集群内部的认证

#创建SA

```
[root@master-1 job]# cat super-sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: super-sa
  namespace: kube-system
```

#创建账号

```
[root@master-1 ~]# kubectl  apply -f super-sa.yaml
serviceaccount/super-sa created
```

#创建ClusterRole (控制权限)
使用的cluster-admin这个对象，这是Kubernetes集群内置的 ClusterRole 对象，无需创建。

#查看cluster-admin 定义的权限
#所有的资源不受限制，相当于超级管理员

```
[root@master-1 job]# kubectl get clusterrole cluster-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2019-11-29T02:16:49Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  resourceVersion: "40"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/cluster-admin
  uid: df4c2dff-d3d9-4184-a72c-7604d964c2bf
rules:
- apiGroups:
```

```
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

#创建角色绑定

RoleBinding 与 ClusterRoleBinding，最大的不同就是, RoleBingding 受限制于NS, ClusterRoleBinding反之.

```
[root@master-1 job]# cat super-sa-ClusterRoleBinding.yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: super-sa-clusterrolebinding
subjects:
- kind: ServiceAccount
  name: super-sa
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

#创建权限绑定

```
[root@master-1 ~]# kubectl   apply -f super-sa-ClusterRoleBinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/super-sa-clusterrolebinding created
```

#获取base64编码与案例一中相同

```
[root@master-1 ~]# kubectl get secret -n kube-system
super-sa-token-8c2td              kubernetes.io/service-account-token     3         86s
```

#获取token

```
[root@master-1 ~]# kubectl get secret super-sa-token-8c2td -o jsonpath={.data.token} -n kube-system
|base64 -d
```

#登录系统

11.9.6 案例三: RBAC之Yaml授权文件讲解

```
[root@master-1 job]# cat prometheus-rabc.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus-rbac
subjects:
  - kind: ServiceAccount
    name: prometheus-k8s
    namespace: monitoring
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

11.9.7 案例四: 实际操作授权命令操作

kubectl create clusterrolebinding system:anonymous --clusterrole=cluster-admin --user=system:anonymous

#解释

#集群角色绑定用户system:anonymous，拥有cluster-admin的权限，集群角色名称为system:anonymous

11.10 Yaml 文件与 Pod、Service 属性值

11.10.1 Yaml 是什么?

Yaml 是一种用来写配置文件的语言。结构上它有两种可选的类型: Lists [1,2,3,4] 和 Maps {1:111}。List 用 -（破折号）来定义每一项，Map 则是一个 key:value 的键值对来表示。

11.10.1 YAML语法规则

大小写敏感(区分大小写)

使用缩进表示层级关系

缩进时不允许使用Tal键，只允许使用空格

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
"#"表示注释，从这个字符一直到行尾，都会被解析器忽略
"---"" 为可选的分隔符

11.10.2 Yaml结构类型图解
在Kubernetes中，Yaml支持两种结构类型即可:Lists、Maps

#注
1）Yaml 转 Properties (https://www.toyaml.com/index.html)
2）Yaml 转 Json (http://nodeca.github.io/js-yaml/)

11.10.3 Kubernetes通过Yaml创建资源
如果需要通过yaml 文件创建Kubernetes 对象，需要配置如下的字段:
apiVersion - 创建该对象所使用的 Kubernetes API 的版本
kind - 想要创建的对象的类型
metadata - 帮助识别对象唯一性的数据，包括一个 name 字符串、UID 和可选的 namespace

11.10.4 案例一: Kubernetes pod编排文件讲解
#Pod在程序中运行的必须的编排文件

```
apiVersion: v1 #指定 api 版本，此值必须在 kubectl apiversion 中
kind: Pod #指定创建资源的角色/类型
metadata: #资源的元数据/属性
  name: test-pod #资源的名字，在同一个 namespace 中必须唯一
  labels: #设定资源的标签
    k8s-app: apache
    version: v1
  annotations:                #自定义注解列表
    - name: String            #自定义注解名字
spec: #specification of the resource content 指定该资源的内容
  restartPolicy: Always #表明该容器一直运行，默认 k8s 的策略，在此容器退出后，会立即创建一个相同的容器
  nodeSelector:       #节点选择，先给主机打标签 kubectl label nodes kube-node1 zone=node1
    zone: node1
  containers:
  - name: test-pod #容器的名字
    image: 10.192.21.18:5000/test/chat:latest #容器使用的镜像地址
    imagePullPolicy: Never # Kubernetes 拉取镜像策略
    command: ['sh'] #启动容器的运行命令，将覆盖容器中的 Entrypoint,对应 Dockefile 中的 ENTRYPOINT
    args: ["$(str)"] #启动容器的命令参数，对应 Dockerfile 中 CMD 参数
```

```
    env: #指定容器中的环境变量
    - name: str #变量的名字
      value: "/etc/run.sh" #变量的值
    resources: #资源管理
      requests: #容器运行时，最低资源需求，也就是说最少需要多少资源容器才能正常运行
        cpu: 0.1 #CPU 资源（核数），两种方式，浮点数或者是整数+m，0.1=100m，最少值为 0.001 核（1m）
        memory: 32Mi #内存使用量
      limits: #资源限制
        cpu: 0.5
        memory: 1000Mi
    ports:
    - containerPort: 80 #容器开发对外的端口
      name: httpd  #名称
      protocol: TCP
    livenessProbe: #pod 内容器健康检查的设置
      httpGet: #通过 httpget 检查健康，返回 200-399 之间，则认为容器正常
        path: / #URI 地址
        port: 80
        scheme: HTTP
      initialDelaySeconds: 180 #表明第一次检测在容器启动后多长时间后开始
      timeoutSeconds: 5 #检测的超时时间
      periodSeconds: 15  #检查间隔时间
      #也可以用这种方法
      #exec:  执行命令的方法进行监测，如果其退出码不为 0，则认为容器正常
      #  command:
      #    - cat
      #    - /tmp/health
      #也可以用这种方法
      #tcpSocket: //通过 tcpSocket 检查健康
      #  port: number
    lifecycle: #生命周期管理
      postStart: #容器运行之前运行的任务
        exec:
          command:
            - 'sh'
            - 'yum upgrade -y'
      preStop:#容器关闭之前运行的任务
        exec:
          command: ['service httpd stop']
    volumeMounts:  #挂载持久存储卷
    - name: volume #挂载设备的名字，与 volumes[*].name  需要对应
      mountPath: /data #挂载到容器的某个路径下
```

```
        readOnly: True
  volumes: #定义一组挂载设备
  - name: volume #定义一个挂载设备的名字
    #meptyDir: {}
    hostPath:
      path: /opt #挂载设备类型为 hostPath，路径为宿主机下的/opt,这里设备类型支持很多种
    #nfs
```

11.10.4.1 Pod属性值列表

属性名称	取值类型	是否必选	取值说明
version	String	Required（必）	版本号，例如 v1
kind	String	Required	pod (资源类型)
metadata	Object	Required	元数据
metadata.name	String	Required	pod 的名称，命名规范须符合 RFC 1035 规范
metadata.namespace	String	Required	pod 的所属命名空间，默认值为 default
metadata.labels[]	List		自定义标签列表
metadata.annotation[]	List		自定义注解列表
Spec	Object	Required	pod 中容器的详细定义
spec.containers[]	List	Required	Pod 容器列表
spec.containers[].name	String	Required	容器名称
spec.containers[].image	String	Required	容器镜像名称
spec.containers[].imagePullPolicy	String		获取镜像策略，可选值包括：Always、Never、IfNOtPresent，默认值为 Always
			Always：表示每次都尝试重新下载镜像
			IfNotPresent：表示如果本地有镜像，使用本地镜像，本地镜像不存在时下载镜像
			Never：表示仅使用本地镜像
spec.containers[].command[]	List		容器启动命令列表，如果不指定，则使用镜像打包是的启动命令
spec.containers[].args[]	List		容器启动命令参数列表
spec.containers[].workingDir	String		容器工作目录
spec.containers[].volumeMounts[]	List		挂载到容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String		引用 Pod 定义的共享存储名称，需要使用 volumes[]部分定义的共享存储名称
spec.containers[].volumeMounts[].mountPath	String		存储卷在容器内 Mount 的绝对路径，应少于 512 字符
spec.containers[].volumeMounts[].readOnly	string		是否为只读模式，默认读写模式
spec.containers[].ports[]	list		容器需要暴露端口号列表

spec.containers[].ports[].name	String		端口名称
spec.containers[].ports[].containerPort	Int		容器需要监听的端口号
spec.containers[].ports[].hostPort	Int		容器所在主机需要监听的端口号，默认与 containerPort 相同。设置 hostPost 时，同一台主机无法启动该容器的第二个副本
spec.containers[].ports[].protocol	String		端口协议，支持 TCP 和 UDP，默认 TCP
spec.containers[].env[]	list		容器运行前需要设置的环境变量列表
spec.containers[].env[].name	String		环境变量的名称
spec.containers[].env[].value	String		环境变量值
spec.containers[].resources	Object		资源限制和资源请求的请求设置
spec.containers[].resources.limits	Object		资源限定的设置
spec.containers[].resources.limits.cpu	String		CPU 限制，单位为 core 数，将用于 docker run --cpu-shares 参数
spec.containers[].resources.limits.memory	String		内存限制，单位为 MIB/GiB 等，将用于 docker run --memory 参数
spec.containers[].resources.requests	Object		容器初始化的资源限制设置
spec.containers[].resources.requests.cpu	String		CPU 请求，单位为 core 数，容器启动时初始化可用数量
spec.containers[].resources.requests.memory	String		内存请求，单位为 MIB、GiB 容器启动的初始化可用数量
spec.volumes[]	list		在该 Pod 定义共享存储列表
spec.volumes[].name	String		共享存储卷的名称；在同一个 Pod 中每个存储卷定义一个名称，应符合 RFC 1035 规范。容器定义部分的 containers[]. volumeMount[].name 将引用该存储卷法人名称
spec.volumes[].emptyDir	Object		类型为 emptyDir 的存储卷，表示与 pod 同生命周期的一个临时目录，其值为一个空对象：emptyDir

spec.volumes[].hostPath.path	String		类型为 hostpash 的存储卷，表示挂着 pod 所在宿主机的目录
spec.volumes[].secret	Object		类型为 secret 的存储卷，表示挂载集群预定义的 secret 对象到容器内部
spec.volumes[].configMap	Object		类型为 configMap 的存储卷，表示挂载集群预定义的 configMap 对象到容器内部
spec.volumes[].livenessProbe	Object		对 pod 内容器设置健康状态检查的设置，当探测几次无响应后，系统自动重启该容器。可以设置的方法包括：exec、httpGet、
			和 tcpSocket。对一个容器仅设置一种健康检查方法
spec.volumes[].livenessProbe.exec	Object		对 pod 内部健康状态检查设置 exec 方式
spec.volumes[].livenessProbe.exec.command[]	String		exec 需要指定的命令或者脚本
spec.volumes[].livenessProbe.httpGet	Object		对 Pod 内个容器健康状态检查，设置 HTTPGet 方式。需要指定 Path、pod
spec.volumes[].livenessProbe.tcpSocket	Object		对 pod 内各个容器健康检查的设置，tcpSocket 方式
spec.volumes[].livenessProbe.initialDelaySeconds	Number		容器启动完成后进行首次探测的时间，单位为 s
spec.volumes[].livenessProbe.timeoutSeconds	Number		对容器健康状态检查的等待响应的超时时间。单位为 s，默认为 1s,超过该超时时间设置，将认为该容器不健康，将重启容器
spec.volumes[].livenessProbe.periodSeconds	Number		对容器健康检查的定期时间设置，单位为 s 默认为 10s 探测一次
spec.restartPolicy	String		pod 的重启策略，可选值为 Always、OnFailure、默认值 Always
			Always：pod 一旦停止运行,则无论容器是如何终止的。kubelet 都将它重启

			OnFailure:只有 pod 以非零吗终止时，kubelet 才会重启容器，如果正常退出则不会重启
			Never：pod 终止后将该 pod 退出吗；报告给 Master，不会再重启 pod
spec.nodeSelector	Object		设置 NodeSelector 表示将该 pod 调度到包含这些 label 的 Node 上。以 key:value 格式指定
spec.imagePullSecret	Object		pull 镜像时使用 secret 名称，以 name:secretkey.格式指定
spec.hostNetwork	Boolean		是否使用主机网络模式，默认值为 false.。如果设置为 true，则表示容器使用宿主机网络，使用 Docker 网桥，该 pod 将无法在同一
			台主机启动第二个副本

11.10.5 案例二: Kubernetes deployment运行

```
apiVersion: apps/v1     # 1.9.0 之前的版本使用 apps/v1beta2，可通过命令 kubectl api-versions 查看
kind: Deployment        #指定创建资源的角色/类型
metadata:       #资源的元数据/属性
  name: web-server      #资源的名字，在同一个 namespace 中必须唯一
spec:
  replicas: 2       #pod 副本数量
  selector:         #定义标签选择器
    matchLabels:
      app: web-server           #符合目标的 pod 的标签
  template:                     # Deployment 扩容 pod 的时候根据此模板
    metadata:
      labels:                   #Pod 的 label
        app: web-server         #pod 副本的标签，selector 根据此标签选择 pod 副本
    spec:                       # 指定该资源的内容
      containers:               # pod 内容器定义的部分
      - name: nginx             #容器的名字
        image: nginx:1.12.1     #容器的镜像地址
        ports:
        - containerPort: 80   #容器暴露的端口
```

#部署

```
[root@master ~]# kubectl create -f nginx.yaml
```

11.10.6 案例三: Kubernetes Services 运行

```
[root@master ~]# cat nginx-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service          #service 名称
spec:
  selector:
    app: web-server            #匹配 service 选择器标签的 pod
  ports:
  - protocol: TCP
    port: 80                   #service 端口
    targetPort: 80             #pod 端口
```

11.10.6.1 Services 属性值列表

属性名称	取值类型	是否必选	取值说明
version	String	Required（必）	版本号，例如 v1
kind	String	Required	pod (资源类型)
metadata	Object	Required	元数据
metadata.name	String	Required	Service 的名称，命名规范须符合 RFC 1035 规范
metadata.namespace	String	Required	Service 的所属命名空间，默认值为 default
metadata.labels[]	List		自定义标签列表
metadata.annotation[]	List		自定义注解列表
Spec	Object	Required	详细描述
spec.selector: []	List	Required	Label Selector 配置，将选择具有指定 Label 标签的 Pod 作为管理范围
spec.type	String	Required	Service 的类型，指定 Service 的访问方式，默认为 ClusterIP
			ClusterIP：虚拟的服务 IP 地址，改地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置 iptables 规则进行转发

			NodePort: 使用宿主机的端口，使能够访问个 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务
			LoadBalancer: 使用外接负载均衡完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，并同时定义 nodePort 和 clusterIP，用于公有云环境
spec.clusterIP	String		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系统自动分配，也可以手动指定；当 type=LoaderBalancer 需要指定
spec.sessionAffinity	String		是否支持 Session，可选值为 ClientIP，默认为空
spec.ports[]	List		需要暴露的端口列表
spec.ports[].name	String		端口名称
spec.ports[].protocol	String		端口协议 支持 tcp 和 udp，默认为 tcp
spec.ports[].port	Int		服务监听的端口号
spec.ports[].targetPort	Int		需要转发到后端 Pod 的端口号
spec.ports[].nodePort	Int		当 spec.type=NodePort 时，指定映射到物理机的端口号
spec.status			当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址，用于公有云环境
spec.status.loadBalancer	String		外部负载均衡器
spec.status.loadBalancer.ingress	String		外部负载均衡器
spec.status.loadBalancer.ingress.ip	String		外部负载均衡器的 IP 地址
spec.status.loadBalancer.ingress.hostname	String		外部负载均衡器的主机名

11.11 ConfigMap

ConfigMap 允许您将配置文件与属性文件分离，以使容器化的应用程序具有可移植性。

1. Kubernetes 空间都可以使用
2. 可以作为变量或者路径文件使用
3. Pod支持自动更新内容

#获取系统中的ConfigMap

```
[root@master1 ~]# kubectl get configmap
```

11.11.1 案例一：将ConfigMap中的所有键值对配置为容器环境变量

#创建一个包含多个键值对的 ConfigMap。

```
[root@master-1 configmap]# cat configmap-multikeys.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: L1
  SPECIAL_TYPE:  Nginx
```

#创建ConfigMap

```
[root@master-1 configmap]# kubectl apply -f    configmap-multikeys.yaml
configmap/special-config created
```

#查看ConfigMap

```
[root@master-1 configmap]# kubectl get configmap/special-config -n default
NAME                  DATA   AGE
special-config   2      65s
```

#查看详细内容

```
[root@master-1 configmap]# kubectl describe configmap special-config
Name:            special-config
Namespace:       default
Labels:          <none>
Annotations:     kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","data":{"SPECIAL_LEVEL":"L1","SPECIAL_TYPE":"Nginx"},"kind":"ConfigMap","metadata":{"annotations":{},"name":"spe
cial-co...

Data
====
SPECIAL_TYPE:
----
```

```
Nginx
SPECIAL_LEVEL:
----
L1
Events:    <none
```

#创建容器引用ConfigMap

```
[root@master-1 configmap]# cat pod-configmap.yaml
apiVersion: v1
kind: Pod
metadata:
   name: pod-test-configmap
spec:
   containers:
      - name: pod-test-configmap
         image: busybox
         command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
         env:
            - name: SPECIAL_LEVEL_KEY
              valueFrom:
                 configMapKeyRef:
                    name: special-config
                    key: SPECIAL_LEVEL
            - name: SPECIAL_TYPE_KEY
              valueFrom:
                 configMapKeyRef:
                    name: special-config
                    key: SPECIAL_TYPE
   restartPolicy: Never
```

#创建pod

```
[root@master-1 configmap]# kubectl apply -f pod-configmap.yaml
pod/pod-test-configmap created
```

#查看pod日志

```
[root@master-1 configmap]# kubectl log pod-test-configmap
log is DEPRECATED and will be removed in a future version. Use logs instead.
L1 Nginx
```

11.11.2 案例二：将 ConfigMap 数据添加到容器中的特定路径

#Nginx pod 文件

```
[root@master-1 configmap]# cat nginx.yaml
apiVersion: apps/v1beta2 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx-configmap
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-configmap
    spec:
      containers:
      - name: nginx-configmap
        image: nginx
        volumeMounts:
          - name: config-volume
            mountPath: /mnt/
        ports:
        - containerPort: 80
      volumes:
        - name: config-volume
          configMap:
            name: special-config
            items:
            - key: SPECIAL_LEVEL
              path: keys
```

#创建

```
[root@master-1 configmap]# kubectl apply -f  nginx.yaml
deployment.apps/nginx-deployment created
```

#查看容器内容

```
[root@master-1 configmap]# kubectl exec -i -t nginx-deployment-69fd86756f-7m2jj bash
root@nginx-deployment-69fd86756f-7m2jj:/ # cd /mnt
root@nginx-deployment-69fd86756f-7m2jj:/mnt# cat keys
L1root
```

#修改ConfigMap 内容（验证pod自动更新）
#修改内容

```
[root@master-1 configmap]# cat configmap-multikeys.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: L111
  SPECIAL_TYPE:  Nginx
```

#应用新配置

```
[root@master-1 configmap]# kubecel apply -f configmap-multikeys.yaml
[root@master-1 configmap]# kubectl describe configmap special-config
```

#查看容器是否可以自动更新
#需要等待1分钟(默认)

```
[root@master-1 configmap]# kubectl exec -it nginx-deployment-69fd86756f-hh6hm -- cat /mnt/keys
L111
```

11.11.3 案例三: 将ConfigMap中的所有键值对配置为容器环境变量
#定义配置文件
#把redis 配置文件存放到configmap
kustomize 应用配置管理

```
[root@master-1 redis]# cat kustomization.yaml
configMapGenerator:
- name: redis-master-config
  files:
  - redis-config
resources:
- redis-pod.yaml
```

#定义redis配置文件

```
[root@master-1 redis]# cat redis-config
maxmemory 2mb
maxmemory-policy allkeys-lru
port 6380
```

#定义redis pod 文件

```
[root@master-1 redis]# cat redis-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis-pod
spec:
  hostNetwork: true
  containers:
  - name: redis-pod
    image: redis:5.0.4
    imagePullPolicy: IfNotPresent
    command:
      - redis-server
      - "/redis-master/redis.conf"
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6380
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: redis-master-config
        items:
        - key: redis-config
          path: redis.conf
```

#创建redis

```
[root@master-1 redis]# kubectl apply -k .
configmap/redis-master-config-k66gtchdm4 configured
```

```
pod/ redis-pod configured
```

#获取redis节点地址

```
[root@master-1 redis]# kubectl get pods -o wide | grep redis
redis-pod   1/1     Running   0          6m10s   192.168.91.21   192.168.91.21   <none>           <none>
```

#连接访问redis

```
[root@master-19 ~]# yum install epel-release -y
[root@master-19 ~]# yum install redis -y
[root@master-19 ~]# redis-cli -h 192.168.91.21 -p 6380
192.168.91.21:6380> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152" #字节
192.168.91.21:6380> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

第 12 章　Kubernetes 存储

本章节主要讲述在 Kubernetes 中为什么需要使用存储，Volume 的基本管理方式，以及怎样创建 PVC 与 PV，结合实际的场景怎样使用 NFS 后端存储。

12.1　Volume 介绍与管理

Pod里面定义的一种属性值

在Docker的设计实现中，容器中的数据是临时的，即当容器被销毁时，其中的数据将会丢失。如果需要（有状态服务）持久化数据（集群或者日志），那么就需要使用Docker数据卷挂载宿主机上的文件或者目录到容器中。在Kubernetes中，当Pod重建的时候，数据会丢失；另外，一个 Pod 中同时运行多个容器时，容器之间需要共享文件. 所以Kubernetes通过数据卷挂载的方式来提供Pod数据的持久化。

12.1.0 区别于Docker中的Volume

Docker 也有 Volume 的概念，但对它只有少量且松散的管理。 在 Docker 中，Volume 是磁盘上或者另外一个容器内的一个目录。 虽然 Docker 现在也能提供 Volume 驱动程序，但是目前功能还非常有限（例如，截至 Docker 1.7，每个容器只允许有一个 Volume 驱动程序，并且无法将参数传递给卷，并且支持的存储驱动有限）。

12.1.1 Kubernetes 中支持的卷类型

目前，Kubernetes 支持的数据卷,类型如下：

1) EmptyDir (单节点存储)　　2) HostPath (单节点存储)

3) GCE Persistent Disk　　4) AWS Elastic Block Store

5) NFS　　6) iSCSI

8) GlusterFS　　9) RBD

10) Persistent Volume Claim

图解Volume在Kubernetes中位置:

12.1.2 Kubernetes 哪些资源对象使用Volume?

Pod

12.1.2.1 案例一: 挂载hostpath到pod

#把主机中的/data/nginx/log 目录挂载到pod中的/var/log/nginx 目录

#需要在所有的节点创建目录: /data/nginx/log

```
[root@master-1 ~]# cat demo-mount-hostpath.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-hostpath
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: nginx-log
      mountPath: /var/log/nginx
  volumes:
  - name: nginx-log
    hostPath:
      path: /data/nginx/log
```

#创建pod

```
[root@master-1 volume]# kubectl   apply -f demo-mount-hostpath.yaml
pod/pod-hostpath created
```

#通过访问nginx，可以看到日志记录

```
[root@master-1 ingress]# kubectl run -it --rm --restart=Never --image=infoblox/dnstools:latest dnstools
```

#Node节点查看日志

```
[root@node-21 log]# pwd
```

```
/data/nginx/log
[root@node-21 log]# ls
access.log    error.log
```

12.1.2.2　案例二：挂载emptyDir到pod

emptyDir 类型的 volume 在 pod 分配到 node 上时被创建，kubernetes 会在 node 上自动分配 一个目录，因此无需指定宿主机 node 上对应的目录文件。这个目录的初始内容为空，**当 Pod 从 node 上移除时，emptyDir 中的数据会被永久删除。**

emptyDir Volume 主要用于某些应用程序无需永久保存的临时目录，多个容器的共享目录等。

```
[root@master-1 redis]# cat redis.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

#创建

```
[root@master-1 volume]# kubectl apply -f redis-empty.yaml
pod/redis created
```

#进入容器,创建文件

```
[root@master-1 redis]# kubectl exec -it redis -- /bin/bash
root@redis:/data# cd /data/redis/
root@redis:/data/redis# echo "test file" > t1.txt
```

#删除容器

```
[root@master-1 ~]# kubectl delete -f redis/redis.yaml
pod "redis" deleted
```

#创建新容器

```
[root@master-1 ~]# kubectl apply -f redis/redis.yaml
pod/redis created
```

#查看数据（空）

```
[root@master-1 redis]# kubectl exec -it redis -- ls /data/redis/
```

12.2 Persistent Volume 与 Persistent Volume Claim

12.2.1 PersistentVolume

PersistentVolume（PV）是由管理员设置的存储卷，它是群集的一部分。就像节点是集群中的资源一样，PV 也是集群中的资源。 PV 是 Volume 之类的卷插件，但具有独立的生命周期。

12.2.1.0 PersistentVolume回收策略

PersistentVolumes 可以有多种回收策略，包括 **“Retain”、”Recycle”** 和 **“Delete”**。
对于动态配置的 PersistentVolumes 来说，默认回收策略为 “Delete”。这表示当用户删除对应的 PersistentVolumeClaim 时，动态配置的 volume 将被自动删除。

使用 “Retain” 时，如果用户删除 PersistentVolumeClaim，对应的 PersistentVolume 不会被删除。相反，它将变为 Released 状态，表示所有的数据可以被手动恢复。

12.2.1.1 列出集群中的 PersistentVolumes

```
[root@master-1 ~]# kubectl get pv
```

12.2.1.2 修改PV回收策略

选择你的 PersistentVolumes 中的一个并更改它的回收策略：
kubectl patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'

12.2.1.3 PV的访问模式

1. ReadWriteOnce: 是最基本的方式，可读可写，但只支持被**单个Pod**挂载。

2. ReadOnlyMany: 可以以**只读的方式**被**多个Pod**挂载。
3. ReadWriteMany: 可以以**读写的方式被多个Pod**共享。

注意:不是每一种存储都支持这三种访问模式，像共享方式，目前支持的还不多(数据一致性问题)，通常简单的实现为NFS。所以在PVC绑定PV时通常根据两个条件来绑定，一个是存储的大小，另一个就是访问模式。

12.2.2 PersistentVolumeClaim

PersistentVolumeClaim（PVC）是用户存储的请求。它与 Pod 相似。Pod 消耗节点资源，PVC 消耗 PV 资源。Pod 可以请求特定级别的资源（CPU 和内存）。声明可以请求特定的大小和访问模式（例如，可以以读/写一次或 只读多次模式挂载）。集群管理员还可以使用 StorageClasses 来设置动态提供存储.

12.2.2.1 PersistentVolumeClaim存储挂载过程:

1.集群管理员创建由物理存储支持的 PersistentVolume。管理员不将卷与任何 Pod 关联。
2.群集用户创建一个 PersistentVolumeClaim，它将自动绑定到合适的 PersistentVolume。
3.用户创建一个使用 PersistentVolumeClaim 作为存储的 Pod。

12.2.3 Persistent Volume与 **Volume** 的区别:

a) PV 只能是网络存储，不属于任何的节点，必须可以在每个节点上可以访问。
b) PV 不是定义在pod 之上， 而是独立于pod，是属于kubernetes 存储的一部分。
c) PV 目前支持的类型: **NFS、ISCSI、RBD、HostPath** ……
d) Volume与使用它的Pod之间是一种静态绑定关系，在定义Pod里面的一种属性，同时定义了它使用的Volume的类型.
e) Volume无法单独创建，因为它不是独立的Kubernetes资源对象

PV/PVC 关系图:

12.2.4 案例一: PVC 挂载NFS 存储

12.2.4.1 安装NFS

#master节点安装nfs

```
[root@master-1 ~]# yum -y install nfs-utils
```

#创建NFS目录

```
[root@master-1 ~]# mkdir -p /data/nfs
```

#修改权限

```
[root@master-1 ~]# chmod -R 777 /data/nfs
```

#修改配置文件

```
[root@master-1 ~]# vim /etc/exports
```

```
/data/nfs *(rw,no_root_squash,sync)
```

#修改启动文件

```
[root@master-1 ~]# cat >/etc/systemd/system/sockets.target.wants/rpcbind.socket<<EOFL
[Unit]
Description=RPCbind Server Activation Socket
[Socket]
ListenStream=/var/run/rpcbind.sock
ListenStream=0.0.0.0:111
ListenDatagram=0.0.0.0:111
[Install]
WantedBy=sockets.target
EOFL
```

#生效配置

```
[root@master-1 ~]# exportfs -r
```

#查看生效

```
#查看生效
[root@master-1 ~]# exportfs
```

#启动服务

```
[root@master-1 ~]# systemctl restart rpcbind && systemctl enable rpcbind
[root@master-1 ~]# systemctl restart nfs && systemctl enable nfs
```

#客户端挂载测试，客户端需要安装nfs-utils

```
[root@master-1 ~]# showmount -e 192.168.91.143
```

12.2.4.2 创建PV

#关联NFS到PV

```
[root@master-1 ~]# cat nfs-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs    #pv 的名称
spec:
  capacity: #容量
    storage: 10Gi    #pv 可用的大小
  accessModes: #访问模式
    - ReadWriteOnce    #PV 以 read-write 挂载到一个节点
  persistentVolumeReclaimPolicy: Recycle #持久卷回收策略
  storageClassName: nfs    #存储类名称
```

```
  nfs:
    path: /ifs/kubernetes          #NFS 的路径
    server: 192.168.91.18      #NFS 的 IP 地址
```

PV回收策略:
保留(Retain):允许人工处理保留的数据.
删除(Delete):将删除pv和外部关联的存储资源,需要插件支持.
回收(Recycle):将执行清除操作,之后可以被新的pvc使用,需要插件支持.

#部署PV

```
[root@master-1 ~]# kubectl apply -f nfs-pv.yaml
```

#获取PV信息

```
[root@master3 nfs]# kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS   CLAIM                 STORAGECLASS    REASON   AGE
pv-nfs    10Gi       RWO            Recycle          Available                                   nfs                     4s
```

PV 的状态信息:
Available – 资源尚未被claim使用
Bound – 卷已经被绑定到claim了
Released – claim被删除，卷处于释放状态，但未被集群回收
Failed – 卷自动回收失败

PV 的访问模式
ReadWriteOnce – PV以 read-write 挂载到一个节点
ReadWriteMany – PV以read-write方式挂载到多个节点
ReadOnlyMany – PV以read-only方式挂载到多个节点

12.2.4.3 创建PVC
#关联PV 到PVC

```
[root@master3 nfs]# vim pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc #PVC 的名称
spec:
  accessModes:
    - ReadWriteOnce #PVC 以 read-write 挂载到一个节点
  resources:
    requests:
      storage: 10Gi
```

```
  storageClassName: nfs #绑定 PV
```

#部署

```
[root@master3 nfs] kubectl apply -f pvc.yaml
```

12.2.4.4 查看PVC状态

```
[root@master3 nfs]# kubectl get pvc
NAME      STATUS    VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nfs-pvc   Bound     pv-nfs    10Gi       RWO            nfs            109s
```

#查看pv状态

```
[root@master-1 pv]# kubectl get pv
NAME                CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   REASON   AGE
pv-nfs              10Gi       RWO            Recycle          Bound    default/nfs-pvc       nfs                     2m
```

12.2.4.5 创建Deployment

```
[root@master3 nfs]# cat nginx-deployment.yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-pvc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-pvc
  template:
    metadata:
      labels:
        app: nginx-pvc
    spec:
      containers:
      - name: nginx-pvc
        image: nginx
        ports:
          - containerPort: 80
        volumeMounts:
          - name: nginx     #申请 pvc 的名称,要和 spec.volumes.name 匹配
            mountPath: /var/log/nginx/    #pod 内部的目录
            subPath: logs/nginx #挂载到 pv 目录下的 logs/nginx,如不指定此参数则挂在到 pv 的/目录下
      volumes:
```

```
      - name: nginx #申请的名称,要和 spec.spec.containers.volumes.name 匹配
        persistentVolumeClaim:    #PVC 方式
          claimName: nfs-pvc    #PVC 的名称
```

#部署

```
[root@master1~]# kubectl apply -f nginx-deployment.yaml
```

#查看日志

```
[root@master-1 pv]# ll /ifs/kubernetes/logs/nginx/
total 0
-rw-r--r-- 1 root root 0 May    3 14:48 access.log
-rw-r--r-- 1 root root 0 May    3 14:48 error.log
```

12.2.4.6 挂载流程图

12.3 StorageClass

#为PVC动态分配PV
每个 StorageClass 都有一个分配器，用来决定使用哪个卷插件分配 PV
#不区分NS

12.3.1 StorageClass的作用

在Kubernetes中，用户使用pod需要持久化数据(集群)，通常情况下，需要集群管理员手工创建PV，然后用户绑定在pod 中，提供给pod中的集群使用。假设在大规模的情况下，如果每次都是需要管理员人工创建PV，那么很严重的影响效率；所以引入了StorageClass，让PVC通过StorageClass 自动创建PV。

12.3.2 StorageClass 支持的卷插件

1) AWSElasticBlockStore
2) RBD
3) Local
4) Glusterfs
5) CephFS

12.3.1 StorageClass 创建PV 流程:

1）集群管理员预先创建存储类（StorageClass）；
2）用户创建POD, POD中使用存储类的持久化存储声明(PVC)；
3）存储持久化声明（PVC）通知系统，需要一个持久化存储(PV)；
4）系统读取存储类的信息；
5）系统基于存储类的信息，在后台自动创建PVC需要的PV；

6）在后台创建完成PV之后, PVC 绑定到PV，进行数据的持久化处理。
7）POD 通过PVC 持久化数据存储。

12.3.2 查询集群中的 PersistentVolumes

```
[root@master-1 demo]# kubectl get sc
NAME                    PROVISIONER          AGE
managed-nfs-storage     fuseim.pri/ifs     3d3h
```

12.3.1 案例一：通过StorageClass自动创建NFS类型PV

#创建nfs pod

```
[root@master1~]# cat nfs-deployment.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-client-provisioner
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: quay.io/external_storage/nfs-client-provisioner:latest
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: fuseim.pri/ifs
            - name: NFS_SERVER
```

```
                value: 192.168.91.143
              - name: NFS_PATH
                value: /ifs/kubernetes
        volumes:
          - name: nfs-client-root
            nfs:
              server: 192.168.91.143
              path: /ifs/kubernetes
```

#创建StoreClass
#绑定NFS后台

```
[root@master1~]# cat nfs-class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-nfs-storage
provisioner: fuseim.pri/ifs # or choose another name, must match deployment's env PROVISIONER_NAME'
parameters:
  archiveOnDelete: "true"
```

#创建PVC
#注意PVC与挂载的pod在同一个NS

```
[root@master1~]# cat grafana-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: grafana
  namespace: monitoring
spec:
  storageClassName: managed-nfs-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

prometheus 与 grafana 调用

```
      - name: grafana-storage
        persistentVolumeClaim:
          claimName: grafana
```

#查看系统PV

```
[root@master-1 demo]# kubectl get pv -A
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM                              STORAGECLASS          REASON   AGE
pvc-138a8cef-d058-4900-bb94-7fccfe8c9661   5Gi        RWX            Delete
Bound    monitoring/prometheus-data         managed-nfs-storage            3d3h
pvc-762ac09e-b270-4000-bf35-80db30dd6a9e   5Gi        RWO            Delete
Bound    monitoring/grafana                 managed-nfs-storage            3d3h
```

#查看PVC

```
[root@master-1 demo]# kubectl get pvc -A
NAMESPACE    NAME                                                    STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS          AGE
monitoring   grafana                 Bound    pvc-762ac09e-b270-4000-bf35-80db30dd6a9e   5Gi
RWO            managed-nfs-storage   3d3h
monitoring   prometheus-data         Bound    pvc-138a8cef-d058-4900-bb94-7fccfe8c9661   5Gi
RWX            managed-nfs-storage   3d3h
```

第 13 章 Kubernetes 网络

本章节讲述 Docker 原生网络问题，通过 Kubernetes 怎样解决这些网络问题，并且在 Kubernetes 中能够加载哪些网络组件，每个类型的网络组件的优劣势，最后总结应用服务的访问流程。

13.0 Docker 原生网络模型

原生网络模型中所有主机节点上的容器都会连接到主机内部的 Docker0 （Linux Bridge），主机的 IP 它默认会分配 172.17.0.0/16 网段中的某一个 IP，因为有了 Docker0，所以在一个**单一主机**上的所有容器可以实现内部互联互通。

但是因为IP分配的范围是基于**单一主机**的，所以你会发现在其他主机上，也会出现完全相同的IP地址。很明显，这两个IP地址肯定没办法直接通信。所以为了解决这个问题，可以使用 iptables 的DNAT功能，设置端口映射。但是也存在弊端，同一主机上不能启动两个相同的应用端口。

总结原生网络问题：

1） 默认情况下,只能实现单一主机内部容器的通信。
2） 在不同的节点主机容器通信时，需要进行 DNAT 端口映射,在应用较多的情况下，非常繁琐，也非常容器出错。
3） 不同主机节点默认分配给 Docker0 的网段有可能存在重复的情况。所以只能用 DNAT 的方式。

13.1 Kubernetes 网络组件

正是因为**原生网络**存在多个节点之间容器互通的问题,那么Kubernetes 通过引入**第三方**的网络组件的方法，来解决这一问题。

常见的网络组件有：Flannel、Calico、Open Vswitch，本内容会重点讲解前面两种网络组件，并且解释数据包的封装过程。

13.1.1 Flannel 网络

Flannel实现的功能有两点。
1) 协助Kubernetes给每一个Node上的Docker0分配互相不冲突的IP地址段。
2) 能在每个Node节点容器IP地址段之间建立一个覆盖网络（Overlay Network：将TCP数据包装在另一种网络包里面进行路由转发和通信），通过这个覆盖网络，将数据包原封不动地传递到不同Node节点的目标容器内。

13.1.1.1 Flannel 数据包封装过程

1) 源容器172.17.18.9向目标容器172.17.19.9发送数据，数据通过Veth Pair到达Docker0网桥172.17.18.1。（可以在容器内通过查看路由表查看到网关为Docker0）
2) Docker0网桥接收到数据包后，根据系统路由表信息，将数据包转发给flannel.1虚拟网卡。

3）Flannel.1 接收到数据包以后，对数据进行二层以太网数据包封装，
Ethernet Header的信息(Flannel节点MAC):
From: { fa:16:3e:f0:2c:11 }
To: { fa:16:3e:f0:2c:43 }

4）Flannel.1 封装第一层MAC数据包以后，系统内核还会进行第二层的(MAC、IP)数据包封装，最后转发数据包到目标容器Node2的em1。
再次封装包的格式如下:
Ethernet Header的信息（Node）:
From: { fa:16:3e:f0:2c:c6}
To: { f1:16:3e:10:11:19}
IP Header的信息:
1) From: {192.168.91.134}
2) To: {192.168.91.135}

13.1.1.1 Flannel 数据包解封装过程

1）目标容器宿主机的em1接收到数据后，对数据包进行拆封，并转发给Flannel.1虚拟网卡；
2）Flannel.1 虚拟网卡接受到数据，将数据发送给Docker0网桥；
3）最后，数据包通过veth pair到达目标容器，完成容器之间的数据通信。

13.1.2 Calico

Calico 与其他虚拟网络最大的不同是，它没有采用Overlay 网络做报文的转发，提供了纯 3 层的网络模型。三层通信模型表示每个容器都通过 IP 直接通信，中间通过路由转发找到对方。在这个过程中，容器所在的节点类似于传统的路由器，提供了路由查找的功能。要想路由工作能够正常，每个虚拟路由器（容器所在的主机节点）通过BGP（Border Gateway Protocol） 路由协议知道整个集群的路由信息。

数据包三层转发基础?

Calico 在每个计算节点利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发。每个 vRouter 通过 BGP 协议把在本节点上运行的容器的路由信息向整个 Calico 网络广播，并自动设置到达其他节点的路由转发规则。Calico 保证所有容器之间的数据流量都是通过 IP 路

由的方式完成互联互通的。Calico 节点组网可以直接利用数据中心的网络结构（L2 或者 L3），不需要额外的 NAT、隧道或者 Overlay Network，没有额外的封包解包，能够节约 CPU 运算，提高网络通信效率。

Calico 一端作为容器网卡加入到容器的网络命名空间，并设置 IP 和掩码；另一端直接暴露在宿主机上。

13.1.2.1 Calico数据包分装过程

1）从节点1容器（172.17.18.9）发送数据包到节点2（172.17.19.9），数据包会通过veth pair到达宿主机的cali网卡上。

2）系统在接收到数据包以后， 根据容器要访问的IP所在的子网CIDR和通过查找主机上的路由规则，找到下一跳要到达的宿主机IP 地址。

3）数据包到达下一跳的宿主机后，查找当前宿主机的路由表信息，直接转发数据veth pair的网卡，最后数据包转发到容器中。

13.1.2.2 总结

Flannel 通过Overlay技术（数据包再封装）发送数据包到达目标容器。

Calico 使用直接路由的模式传递数据包。

13.2 kubernetes 中应用服务访问流程

1） 用户需要访问https://api.abcd.com

2） 用户请求到达物理主机的负载均衡器，转发Ingress端口 80

3） Ingress 根据请求的域名，查找到匹配相对应的Service名称获取Cluster IP与port

4） Ingress 转发请求到Service port

5） Service 根据容器label转发请求到相匹配的pod