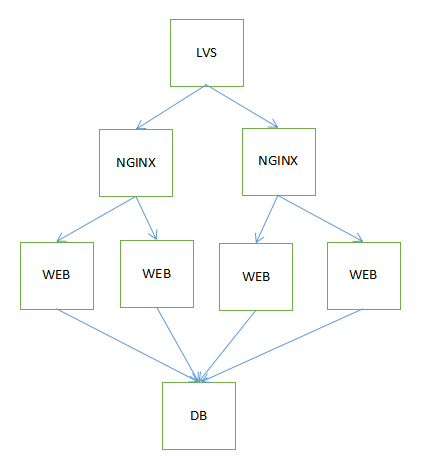
# Kubernetes

## 第一章：互联网架构的演变

随着1946年世界上第一台电子计算机的问世网络就随之出现了，只不过当初只是为了解决多个终端之间的连接，这就是局域网的雏形。后来，随着美国国防部高级研究计划局（ARPA）的无线、卫星网的分组交换技术的研究问世，一不小心搞出了TCP/IP，由此出现了我们所熟知的Internet互联网。

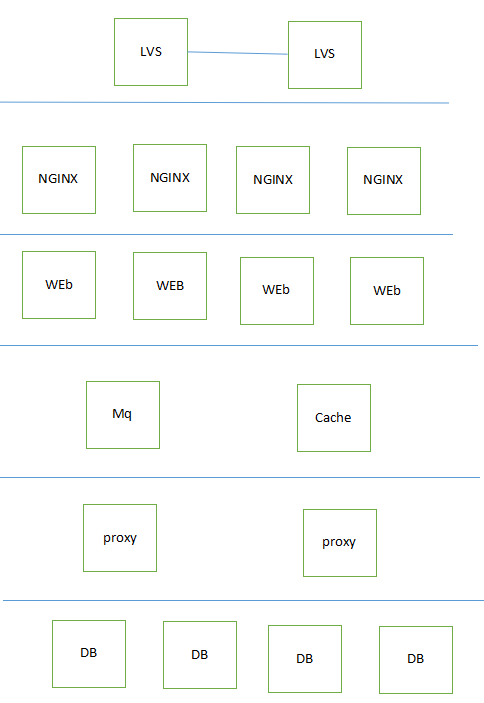
早期的电脑体积大，价格昂贵，一般只用在科学研究领域，后来随着集成电路，半导体的发展，电脑逐渐出现在我们的日常生活中。由此，我们对互联网的依赖越来越多，到现在已经成为了我们生活中的一部分，可以说互联网极大地改变了我们的生活方式。

在互联网刚刚进入我们生活之中的时候，架构还比较简单，一般的只有一个WEB服务器，提供几个简单的页面。由此我们的第一代互联网架构就产生了。这种结构有一个致命的缺陷，就是一旦web服务器挂掉，服务就会终止。于是产生了主备和负载均衡，这种架构是第一代架构的补充，如图：



这种架构简单，但是可拓展性差，耦合性高。

随着我们业务的不断发展，用户量的不断增加，第一代架构显然无法支持我们业务的需求，于是逐渐过渡到了我们第二代互联网架构，第二代互联网架构的特点是分层，所以我们也称之为分层架构。

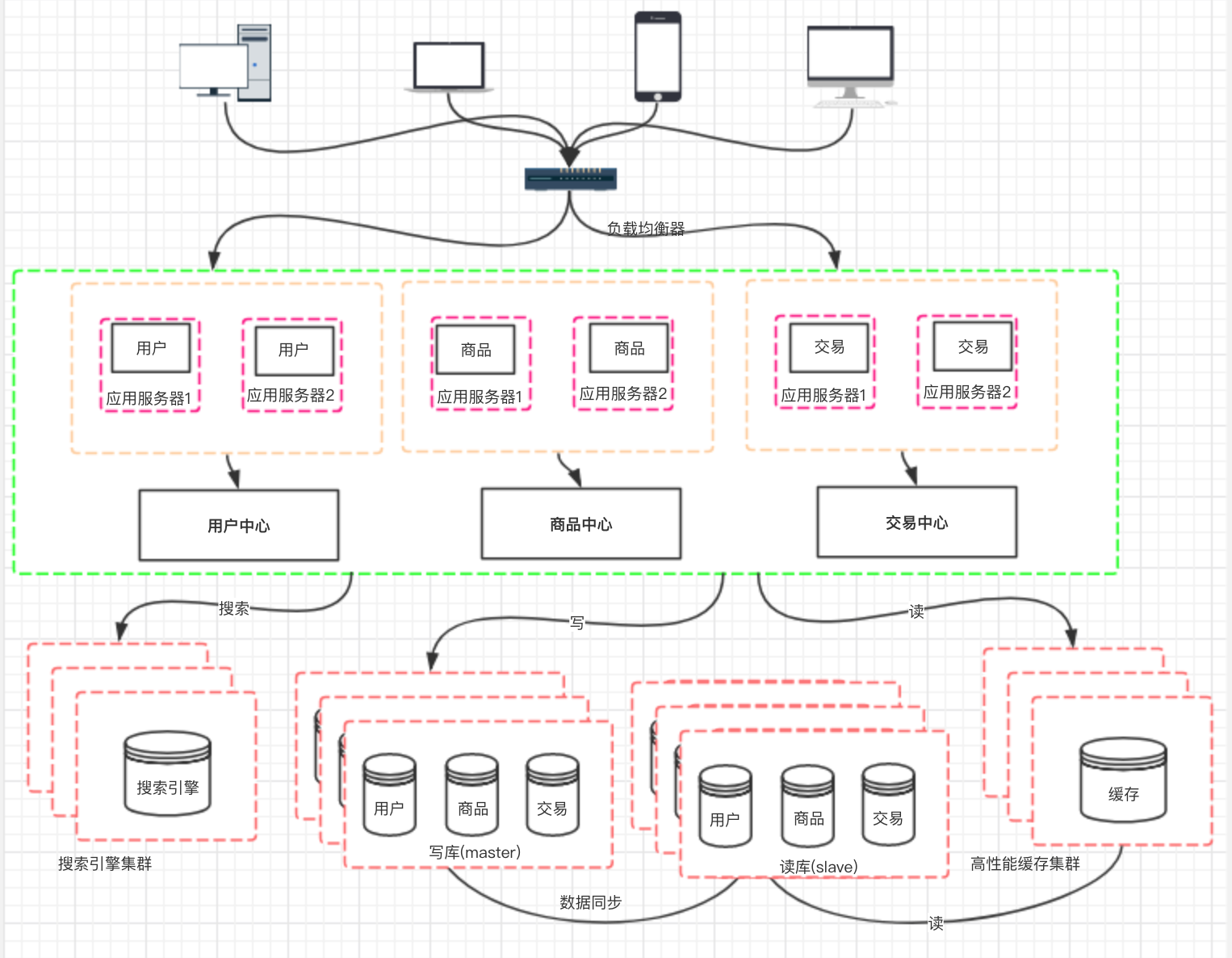


这种架构的优点是可以横向扩展，但是耦合性相对也很高。一个IP操作读写。

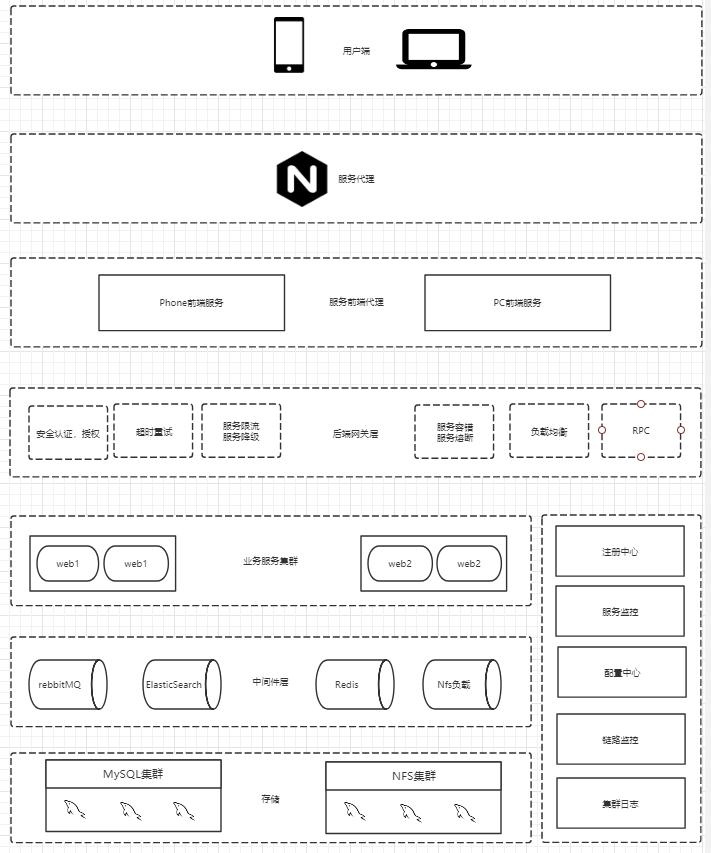
直至现在，业务越来越复杂，用户量越来越多，那么找到一个高性能，解耦合的架构就尤为迫不及待，所以随之而出的先是分布式架构，紧接着出现的就是微服务架构。

1. 分布式架构

分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。



1. 微服务架构



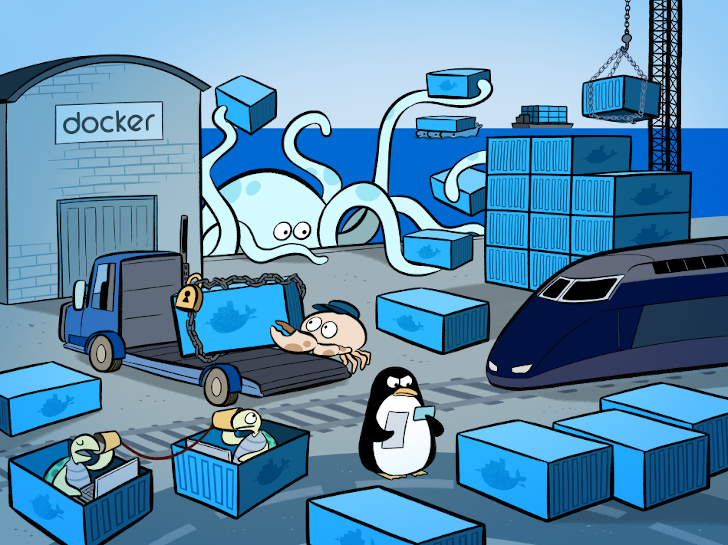
1. 分布式，微服务的区别？Filebeat -> logstach -> elasticsearch -> kibana

* 分布式：将一个大的系统划分为多个业务模块，业务模块分别部署到不同的机器上，各个业务模块之间通过接口进行数据交互。区别分布式的方式是根据不同机器不同业务。
* 微服务：微服务的设计是为了不因为某个模块的升级和BUG影响现有的系统业务。微服务与分布式的细微差别是，微服务的应用不一定是分散在多个服务器上，他也可以是同一个服务器。

## 第二章：Docker

### Docker简介

Docker是一个用于开发，交付和运行应用程序的开放平台。Docker使应用程序与基础架构分开，从而可以快速交付软件。借助Docker，可以和管理应用程序相同的方式来管理基础架构。通过利用Docker的方法来快速交付，测试和部署代码，大大减少编写代码和在生产环境中运行代码之间的延迟。



Docker提供了在简单、隔离的环境（称为容器）中打包和运行应用程序的功能。隔离和安全性使您可以在给定主机上同时运行多个容器。容器是轻量级的，因为它们不需要虚拟机管理程序的额外负载，而是直接在主机的内核中运行。这意味着与使用虚拟机相比，可以在给定的硬件组合上运行更多的容器。您甚至可以在虚拟机的主机中运行Docker容器！

Docker镜像和Docker容器的区别？

1. docker镜像相当于启动容器的模板（相当于Python中的类）
2. docker容器相当于启动起来的应用程序（相当于Python中的实例化好的容器）

Docker提供了工具和平台来管理容器的生命周期：

1. 使用容器开发应用程序及其支持组件。
2. 容器成为分发和测试应用程序的单元。
3. 准备就绪后，可以将应用程序作为容器或协调服务部署到生产环境中。无您的生产环境是本地数据中心，云提供商还是两者的混合，其工作原理都相同。

### Docker架构

Docker引擎是具有以下主要组件的客户端-服务器应用程序：

1. 服务器是一种长期运行的程序，称为守护程序进程（dockerd命令）。
2. REST API，它指定程序可以用来与守护程序进行通信并指示其操作的接口。
3. 命令行界面（CLI）客户端（docker命令）。



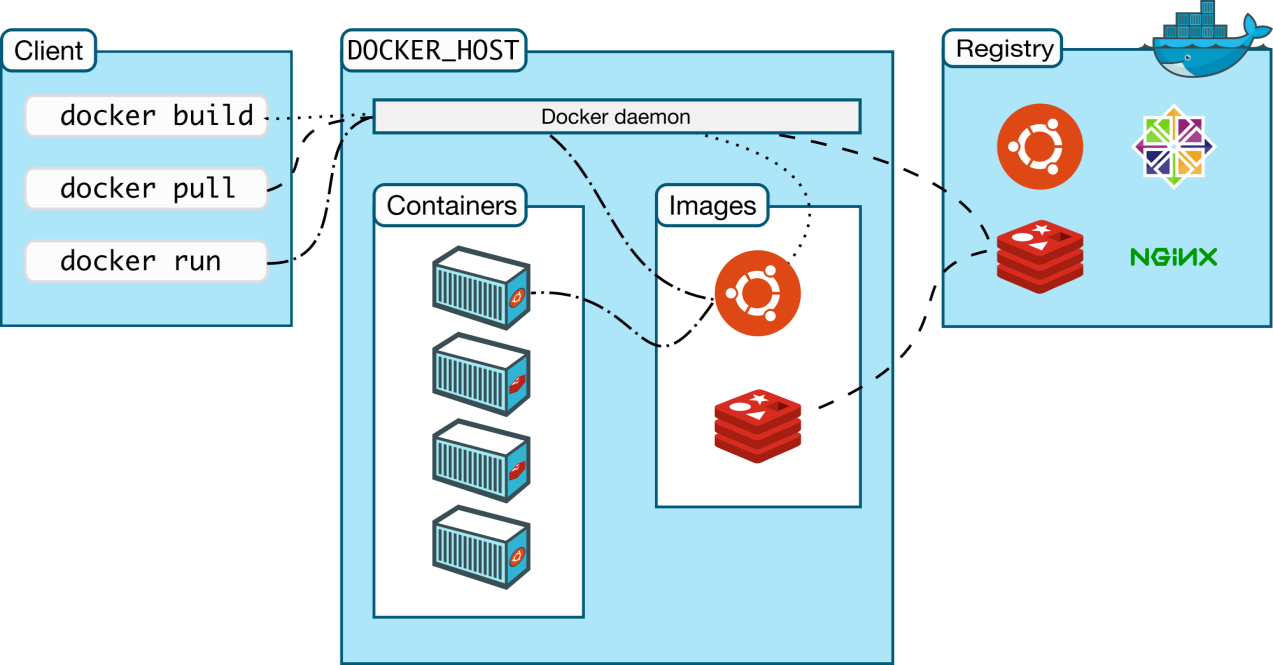
Docker使用客户端-服务器架构。Docker 客户端与Docker 守护进程进行对话，该守护进程完成了构建，运行和分发Docker容器的繁重工作。Docker客户端和守护程序可以在同一系统上运行，也可以将Docker客户端连接到远程Docker守护程序。Docker客户端和守护程序在UNIX套接字或网络接口上使用REST API进行通信。

* Image和container的区别？

Image是镜像（类）

Container是容器（对象）

首先创建image镜像，然后通过image镜像实例化成我们所需的容器（container）。



### 安装

Docker基本可以运行在任何Linux内核的主机之上。

* 1. CentOS
     1. 在阿里云上下载docker-ce.repo

yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

* + 1. 安装必要的包

yum install -y yum-utils device-mapper-persistent-data lvm2

* + 1. 刷新

yum makecache fast

* + 1. 安装

yum -y install docker-ce

* + 1. 启动

systemctl start docker && systemctl enable docker

* + 1. 验证

[root@localhost ~]# docker info

Client:

Debug Mode: false

Server:

Containers: 15

Running: 5

Paused: 0

Stopped: 10

Images: 26

Server Version: 18.09.9

Storage Driver: overlay2

Backing Filesystem: xfs

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: systemd

Plugins:

Volume: local

Network: bridge host macvlan null overlay

Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: 894b81a4b802e4eb2a91d1ce216b8817763c29fb

runc version: 425e105d5a03fabd737a126ad93d62a9eeede87f

init version: fec3683

Security Options:

seccomp

Profile: default

Kernel Version: 4.18.0-80.11.2.el8\_0.x86\_64

Operating System: CentOS Linux 8 (Core)

OSType: linux

Architecture: x86\_64

CPUs: 4

Total Memory: 1.92GiB

Name: localhost.localdomain

ID: FLZM:42YD:NC2K:NATO:YPCA:T6IY:G647:OKPA:WJKK:Z4TQ:AN7R:TMSB

Docker Root Dir: /var/lib/docker

Debug Mode: false

Registry: https://index.docker.io/v1/

Labels:

Experimental: false

Insecure Registries:

127.0.0.0/8

Registry Mirrors:

https://hjvrgh7a.mirror.aliyuncs.com/

Live Restore Enabled: false

Product License: Community Engine

* 1. Ubuntu
     1. 安装必要的一些系统工具

apt-get -y install apt-transport-https ca-certificates curl software-properties-common

* + 1. 安装GPG证书

curl -fsSL https://mirrors.aliyun.com/docker-ce/linux/ubuntu/gpg | sudo apt-key add -

* + 1. 写入软件源信息

sudo add-apt-repository "deb [arch=amd64] https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb\_release -cs) stable"

* + 1. 更新并安装Docker-CE

sudo apt-get -y update

sudo apt-get -y install docker-ce

* + 1. 添加配置

cat > /etc/docker/daemon.json <<EOF

{

"exec-opts": ["native.cgroupdriver=systemd"],

"registry-mirrors": ["https://hjvrgh7a.mirror.aliyuncs.com"],

"log-driver": "json-file",

"log-opts": {

"max-size": "100m"

},

"storage-driver": "overlay2"

}

EOF

要添加我们harbor仓库需要在添加下面一行

"insecure-registries": ["10.0.0.201"],

### 基本使用

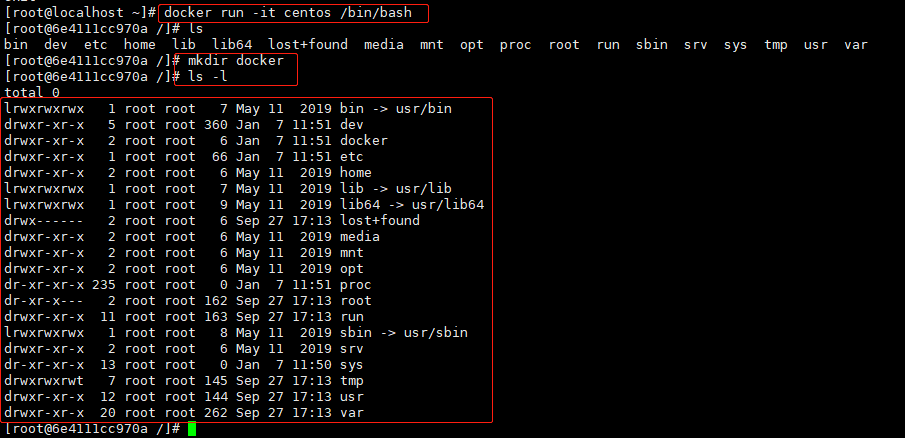
Docker容器中至少有一个应用程序一直运行在前台。

#### 创建容器

docker run -it centos /bin/sh

docker run -it --name centos centos:7 bash

# -it: 建立一个伪终端



* docker run:有两个过程：

1. 检测当前使用的镜像本地是否有，没有则去远程仓库拉取。
2. 将镜像实例化成容器。

* 在docker容器中，至少要有一个应用程序运行在前台，否则在启动的一瞬间生命周期就结束了。
* 为什么加上-it参数之后，就可以了呢？因为伪终端是运行在前台的。

#### 进入容器

##### Acctch

docker attach nginx

使用该命令有一个问题。当多个窗口同时使用该命令进入该容器时，所有的窗口都会同步显示。如果有一个窗口阻塞了，那么其他窗口也无法再进行操作，当所有窗口退出时，容器结束。

##### Exec

docker exec -it nginx /bin/bash

这个命令相当于在容器中执行一个命令。

##### Nsenter

需要配合docker inspect来使用（企业当中最长用的方式之一）

nsenter --target $( docker inspect -f {{.State.Pid}} caf8813adda2 ) --mount --uts --ipc --net --pid

Docker是用golang语言开发，所以它也支持go语言的摸版语法。

##### SSH

在生产环境中排除了使用docker attach命令进入容器之后，相信大家第一个想到的就是ssh。在镜像（或容器）中安装SSH Server，这样就能保证多人进入容器且相互之间不受干扰了，相信大家在当前的生产环境中（没有使用Docker的情况）也是这样做的。但是使用了Docker容器之后不建议使用ssh进入到Docker容器内。

总结：进入docker container中一般情况下有4种方式，最常用的是exec和nsenter这两种。

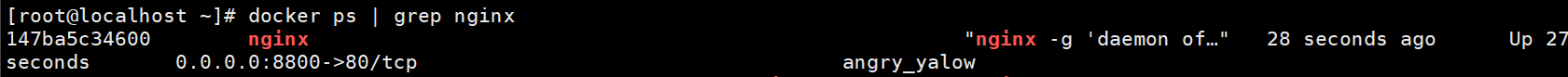
* Nsenter和exec之间的区别？
  + Exec是docker自带的命令，Nsenter是Linux提供的命令。
  + Exec相当于在容器内执行一个命令，而Nsenter是仅仅进入容器之中而已。

#### 端口映射

端口映射一般使用（-p/-P）

##### -p 指定端口映射

docker run -d -p 8800:80 nginx



##### -P 随机端口映射

docker run -d -P nginx



* -p和-P的区别是什么？
  + -p是指定固定端口的
  + -P是随机端口的
  1. 目录及文件映射

docker run -v /root/html:/usr/share/nginx/html nginx

docker inspect jolly\_payne

"Mounts": [

{

"Type": "bind",

"Source": "/root/html",

"Destination": "/usr/share/nginx/html",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

],

#### COPY文件

* + 1. Copy到容器外

docker cp [OPTIONS] CONTAINER:SRC\_PATH DEST\_PATH|-

* + 1. Copy到容器内

docker cp [OPTIONS] SRC\_PATH|- CONTAINER:DEST\_PATH

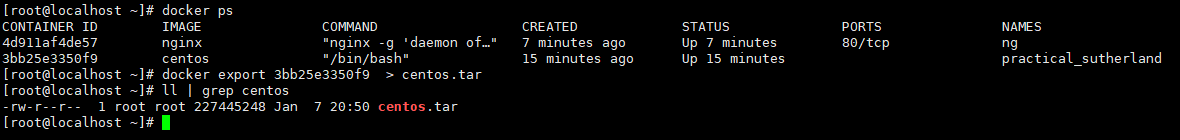
#### 4.5导入和导出

* + 1. 导出

如果要导出本地某个容器，可以使用 docker export 命令。

docker ps

docker export 3bb25e3350f9 > centos.tar



* + 1. 导入

可以使用 docker import 从容器快照文件中再导入为镜像

[root@localhost ~]# cat centos.tar | docker import - test/centos:v1.0

sha256:e1e54eebb51a5ddd8f3d4bc1ed9cb8a4ce2841a7e265bf

### 网络

主要解决容器与容器之间，容器与主机之间的网络互通问题。

* 1. --link

--link 参数的格式为 --link name:alias，其中 name 是要链接的容器的名称，alias 是这个连接的别名。

docker -d --link 容器名:连接别名 nginx

案例：

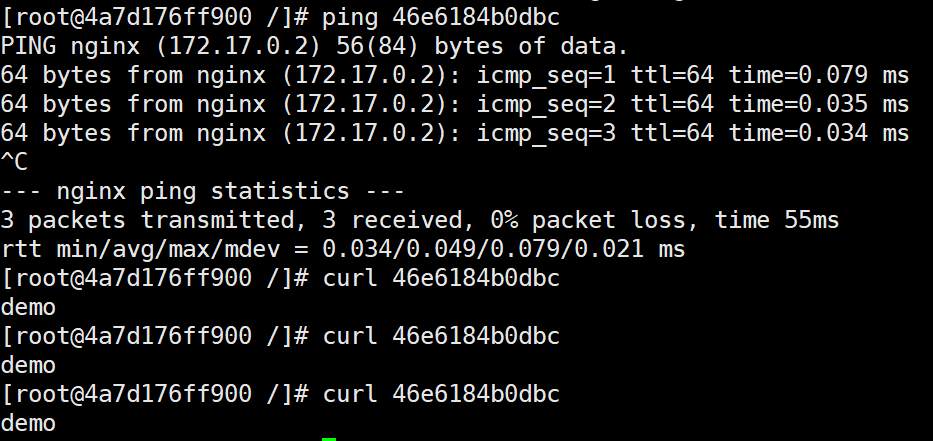
# 第一步：创建一个容器

docker run -d --name nginx nginx

# 第二步：启动一个测试容器

docker run -it --link nginx:nginx centos

# 第三步：在测试容器中测试访问目标容器，看是否可以正常访问



* 1. Network

| Command | Description |
| --- | --- |
| [docker network connect](https://docs.docker.com/engine/reference/commandline/network_connect/) | 将容器连接到网络 |
| [docker network create](https://docs.docker.com/engine/reference/commandline/network_create/) | 创建一个网络 |
| [docker network disconnect](https://docs.docker.com/engine/reference/commandline/network_disconnect/) | 断开容器与网络的连接 |
| [docker network inspect](https://docs.docker.com/engine/reference/commandline/network_inspect/) | 在一个或多个网络上显示详细信息 |
| [docker network ls](https://docs.docker.com/engine/reference/commandline/network_ls/) | 列出网络 |
| [docker network prune](https://docs.docker.com/engine/reference/commandline/network_prune/) | 删除所有未使用的网络 |
| [docker network rm](https://docs.docker.com/engine/reference/commandline/network_rm/) | 删除一个或多个网络 |

使用 docker network create 命令创建用户定义的网桥网络。

docker network create chenyang

使用network

docker run --name my-nginx \

--network chenyang \

-p 8080:80 \

nginx:latest

若要将正在运行的容器连接到现有用户定义的桥接网络，请使用docker network connect命令。

docker network connect chenyang my-nginx

案例：

# 第一步：创建一个名称为chenyang的network

Docker network create chenyang

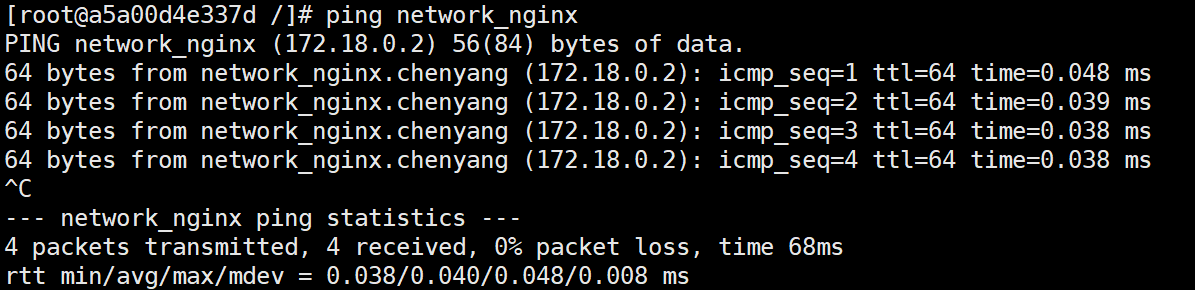
# 第二步：创建一个目标容器

Docker run -d --network chenyang --name network\_nginx nginx

# 第三步：创建一个测试容器

Docker run -it --network chenyang --name network\_centos centos

# 第四步：在测试容器之中测试网络连接情况



### 存储卷

一般情况下，我们可以这样理解：存储卷就是通过文件系统将容器之中的数据持久化保存。还有一个方面是将宿主主机的文件映射到容器之中。

案例

# 第一步：准备好本地的目录

Mkdir nginx

# 第二步：创建容器关联上我们目录

docker run -d --name v\_nginx -v /root/nginx/:/usr/share/nginx/html/ -p 8081:80 nginx

# 第三步：测试文件是否可以互通

[root@k8s-node nginx]# curl 127.0.0.1:8081

Index



### Dockerfile

Dockerfile 由一行行命令语句组成，并且支持以 # 开头的注释行。

一般的，Dockerfile 分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令。

例如:

FROM ubuntu

MAINTAINER Alvin alvincy@qq.com

# Commands to update the image

RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main universe" >> /etc/apt/sources.list

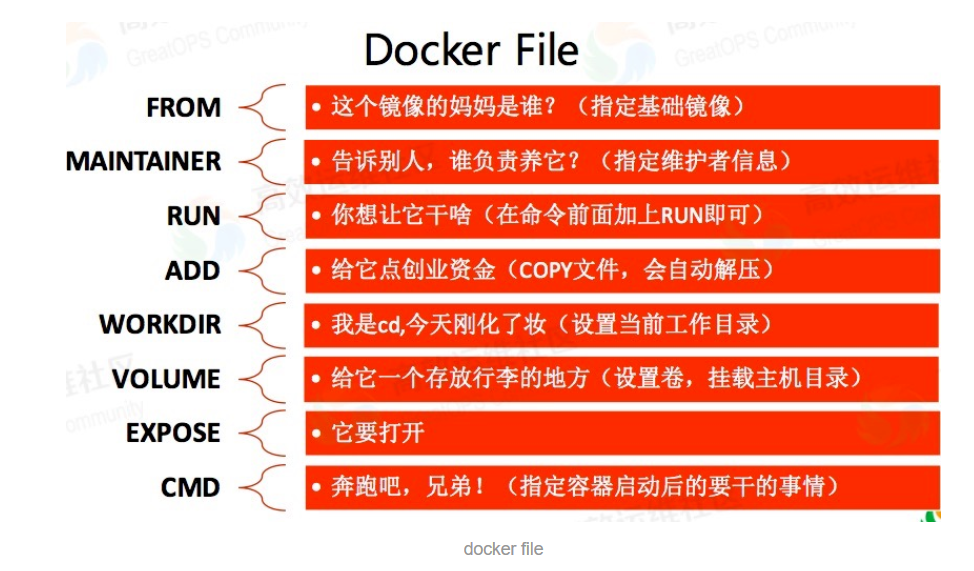
RUN apt-get update && apt-get install -y nginx

RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# Commands when creating a new container

CMD /usr/sbin/nginx

Dockerfile示意图：



#### 1. FROM

指明构建的新镜像是来自于哪个基础镜像，例如：

FROM centos

#### 2. MAINTAINER

指明镜像维护着及其联系方式（一般是邮箱地址），例如：

MAINTAINER Edison Zhou <edisonchou@hotmail.com>

#### RUN

构建镜像时运行的Shell命令，例如：

RUN ["yum", "install", "httpd"]

RUN yum install httpd

#### CMD

启动容器时执行的Shell命令，例如：

CMD ["-C", "/start.sh"]

CMD ["/usr/sbin/sshd", "-D"]

CMD /usr/sbin/sshd -D

#### ENTRYPOINT

启动容器时执行的Shell命令，同CMD类似，只是由ENTRYPOINT启动的程序不会被docker run命令行指定的参数所覆盖，而且，这些命令行参数会被当作参数传递给ENTRYPOINT指定指定的程序，例如：

ENTRYPOINT ["/bin/bash", "-C", "/start.sh"]

ENTRYPOINT /bin/bash -C '/start.sh'

#### EXPOSE

声明容器运行的服务端口，例如：

EXPOSE 80 443

#### ENV

设置环境内环境变量，例如：

ENV MYSQL\_ROOT\_PASSWORD 123456

ENV JAVA\_HOME /usr/local/jdk1.8.0\_45

#### ADD

拷贝文件或目录到镜像中，例如：

ADD <src>...<dest>

ADD html.tar.gz /var/www/html

ADD https://xxx.com/html.tar.gz /var/www/html

#### COPY

拷贝文件或目录到镜像中，用法同ADD，只是不支持自动下载和解压，例如：

COPY ./start.sh /start.sh

#### 10. VOLUME

指定容器挂载点到宿主机自动生成的目录或其他容器，例如：

VOLUME ["/var/lib/mysql"]

#### USER

为RUN、CMD和ENTRYPOINT执行Shell命令指定运行用户，例如：

USER <user>[:<usergroup>]

USER <UID>[:<UID>]

USER edisonzhou

#### WORKDIR

为RUN、CMD、ENTRYPOINT以及COPY和AND设置工作目录，例如：

WORKDIR /data

#### HEALTHCHECK

告诉Docker如何测试容器以检查它是否仍在工作，即健康检查，例如：

HEALTHCHECK --interval=5m --timeout=3s --retries=3 \

CMD curl -f http:/localhost/ || exit 1

其中，一些选项的说明：

* --interval=DURATION (default: 30s)：每隔多长时间探测一次，默认30
* -- timeout= DURATION (default: 30s)：服务响应超时时长，默认30
* --start-period= DURATION (default: 0s)：服务启动多久后开始探测，默认0秒
* --retries=N (default: 3)：认为检测失败几次为宕机，默认3次

一些返回值的说明：

* 0：容器成功是健康的，随时可以使用
* 1：不健康的容器无法正常工作
* 2：保留不使用此退出代码

FROM nginx

RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/\*

HEALTHCHECK --interval=5s --timeout=3s \

CMD curl -fs http://localhost/ || exit 1

这里我们设置了每 5 秒检查一次（这里为了试验所以间隔非常短，实际应该相对较长），如果健康检查命令超过 3 秒没响应就视为失败，并且使用 curl -fs http://localhost/ || exit 1 作为健康检查命令。

使用 docker build 来构建这个镜像：

docker build -t myweb:v1 .

构建好了后，我们启动一个容器：

docker run -d --name web -p 80:80 myweb:v1

当运行该镜像后，可以通过 docker container ls 看到最初的状态为 (health: starting)：

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

03e28eb00bd0 myweb:v1 "nginx -g 'daemon off" 3 seconds ago Up 2 seconds (health: starting) 80/tcp, 443/tcp web

在等待几秒钟后，再次 docker container ls，就会看到健康状态变化为了 (healthy)：

docker container ls

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

03e28eb00bd0 myweb:v1 "nginx -g 'daemon off" 18 seconds ago Up 16 seconds (healthy) 80/tcp, 443/tcp web

如果健康检查连续失败超过了重试次数，状态就会变为 (unhealthy)。

为了帮助排障，健康检查命令的输出（包括 stdout 以及 stderr）都会被存储于健康状态里，可以用 docker inspect 来查看。

docker inspect --format '{{json .State.Health}}' web | python -m json.tool

{

"FailingStreak": 0,

"Log": [

{

"End": "2016-11-25T14:35:37.940957051Z",

"ExitCode": 0,

"Output": "<!DOCTYPE html>\n<html>\n<head>\n<title>Welcome to nginx!</title>\n<style>\n body {\n width: 35em;\n margin: 0 auto;\n font-family: Tahoma, Verdana, Arial, sans-serif;\n }\n</style>\n</head>\n<body>\n<h1>Welcome to nginx!</h1>\n<p>If you see this page, the nginx web server is successfully installed and\nworking. Further configuration is required.</p>\n\n<p>For online documentation and support please refer to\n<a href=\"http://nginx.org/\">nginx.org</a>.<br/>\nCommercial support is available at\n<a href=\"http://nginx.com/\">nginx.com</a>.</p>\n\n<p><em>Thank you for using nginx.</em></p>\n</body>\n</html>\n",

"Start": "2016-11-25T14:35:37.780192565Z"

}

],

"Status": "healthy"

}

#### ARG

在构建镜像时，指定一些参数，例如：

FROM centos:6

ARG user # ARG user=root

USER $user

这时，我们在docker build时可以带上自定义参数user了，如下所示：

docker build --build-arg user=edisonzhou Dockerfile .

总结：

* Docker build -f参数的作用：指定我们Dockerfile的地址，默认是当前目录下的Dockerfile。

备注：docker build命令后面的点是必须的。

#### 案例

FROM centos7

MAINTAINER Alvin alvincy@qq.com

RUN mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup

RUN curl -o /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo

RUN yum makecache

RUN yum update -y

RUN yum install python3 -y

RUN pip3 install django

COPY docker /root/docker

WORKDIR /root/docker

EXPOSE 8080

CMD ["python3", "manage.py", "runserver", "0.0.0.0:8080", “&”]

结果：能够自动发布，自动部署。

作业：通过Dockerfile构建一个镜像，能够提供一个Django服务，然后通过webhook实现自动发布，自动部署。

自动部署的脚本：

#!/bin/bash

DATE=`date +%Y-%m-%d\_%H\_%M\_%s`

cd /www/test/oldboy

git pull

docker build -t django:$DATE .

docker stop $(docker ps | grep django | awk '{print $1}' | xargs)

docker run -d --name $DATE --rm -p 8089:80 django:$DATE

### Docker三剑客

#### docker-compose

**主要是解决本地docker容器编排问题。**一般是通过yaml配置文件来使用它，这个yaml文件里能记录多个容器启动的配置信息（镜像、启动命令、端口映射等），最后只需要执行docker-compose对应的命令就会像执行脚本一样地批量创建和销毁容器。

#### Docker Swarm

**解决多主机多个容器调度部署得问题。**swarm是基于docker平台实现的集群技术，他可以通过几条简单的指令快速的创建一个docker集群，接着在集群的共享网络上部署应用，最终实现分布式的服务。swarm技术相当不成熟，很多配置功能都无法实现，只能说是个半成品，目前更多的是使用Kubernetes来管理集群和调度容器。

#### Docker Machine

**解决docker运行环境问题。**docker技术是基于Linux内核的cgroup技术实现的，那么问题来了，如果在非Linux平台上使用docker技术需要依赖安装Linux系统的虚拟机。docker-machine就是docker公司官方提出的，用于在各种平台上快速创建具有docker服务的虚拟机的技术。

## 第三章：kubernetes基础

Kubernetes 是一个可移植的、可扩展的开源平台，用于管理容器化的工作负载和服务，可促进声明式配置和自动化。Kubernetes 拥有一个庞大且快速增长的生态系统。Kubernetes 的服务、支持和工具广泛可用。

### 简介

作为Google的竞争技术优势，Borg理所当然的被视为商业秘密隐藏起来，但当Tiwtter的工程师精心打造出属于自己的Borg系统（Mesos）时， Google也审时度势地推出了来源于自身技术理论的新的开源工具。

2014年6月，谷歌云计算专家埃里克·布鲁尔（Eric Brewer）在旧金山的发布会为这款新的开源工具揭牌，它的名字Kubernetes，意为“舵手”或“飞行员”，这也恰好与它在容器集群管理中的作用吻合，即作为装载了集装箱（Container）的众多货船的指挥者，负担着全局调度和运行监控的职责。

它是一个全新的基于容器技术的分布式架构领先方案。这个方案虽然还很新，但是它是谷歌十几年依赖大规模应用容器技术的经验积累和升华的一个重要成果。实现资源管理的自动化，以及跨多个数据中心的资源利用率的最大化。

其次，如果我们的系统设计遵循了Kubernetes的设计思想，那么传统系统架构中那些和业务没有多大关系的底层代码或功能模块，都可以立刻从我们的视线消失，我们不必再费心于负载均衡器和部署实施问题，不必再考虑引用或自己开发一个复杂的服务治理框架，不必再头疼于服务监控和故障处理模块的开发。使用Kubernets提供的解决方案，我们仅节省了不少于30%的开发成本，同时可以将精力更加集中于业务本身，而且由于Kubernetes提供了强大的自动化机制，所以系统后期的运维难度和运维成本大幅度降低。

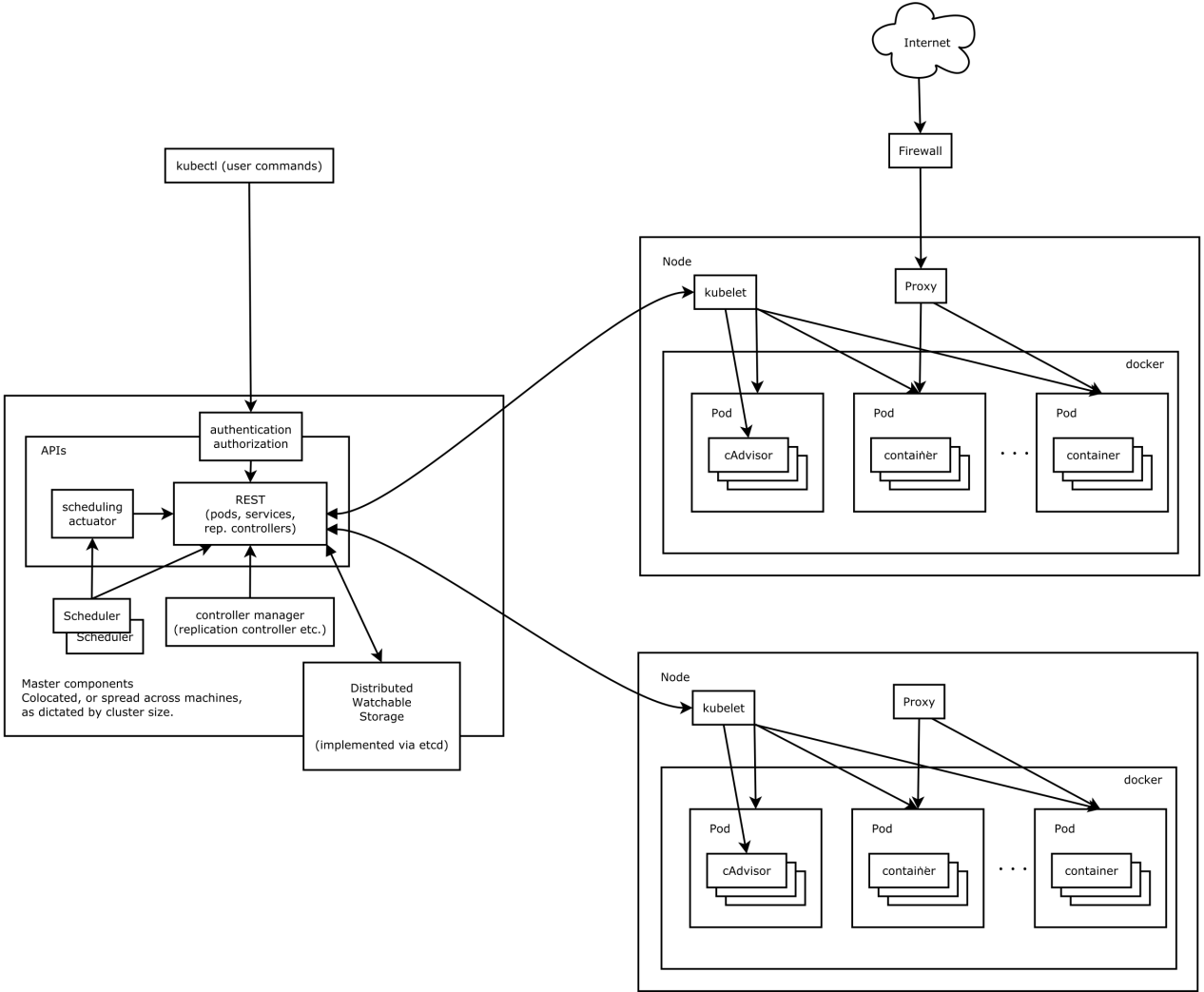
它是一个开发的开发平台。没有限定任何编程接口，所以不论是Java、Go、C++还是用Python编写的服务，都可以毫无困难地映射为Kubernetes的Service，并通过标准的TCP通信协议进行交互。此外，由于Kubernetes平台对现有的编程语言、编程框架、中间价没有任何侵入性，因此现有的系统很容器改造升级并迁移到Kubernetes平台上。

所以说它是一个完备的分布式系统支撑平台，Kubernetes具有完备的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力。同时，Kubernetes提供了完善的管理工具，这些工具涵盖了包括开发、部署测试、运维监控在内的各个环节。因此Kubernetes是一个全新的基于容器技术的分布式架构解决方案，并且是一个一站式的完备的分布式系统开发和支撑平台。

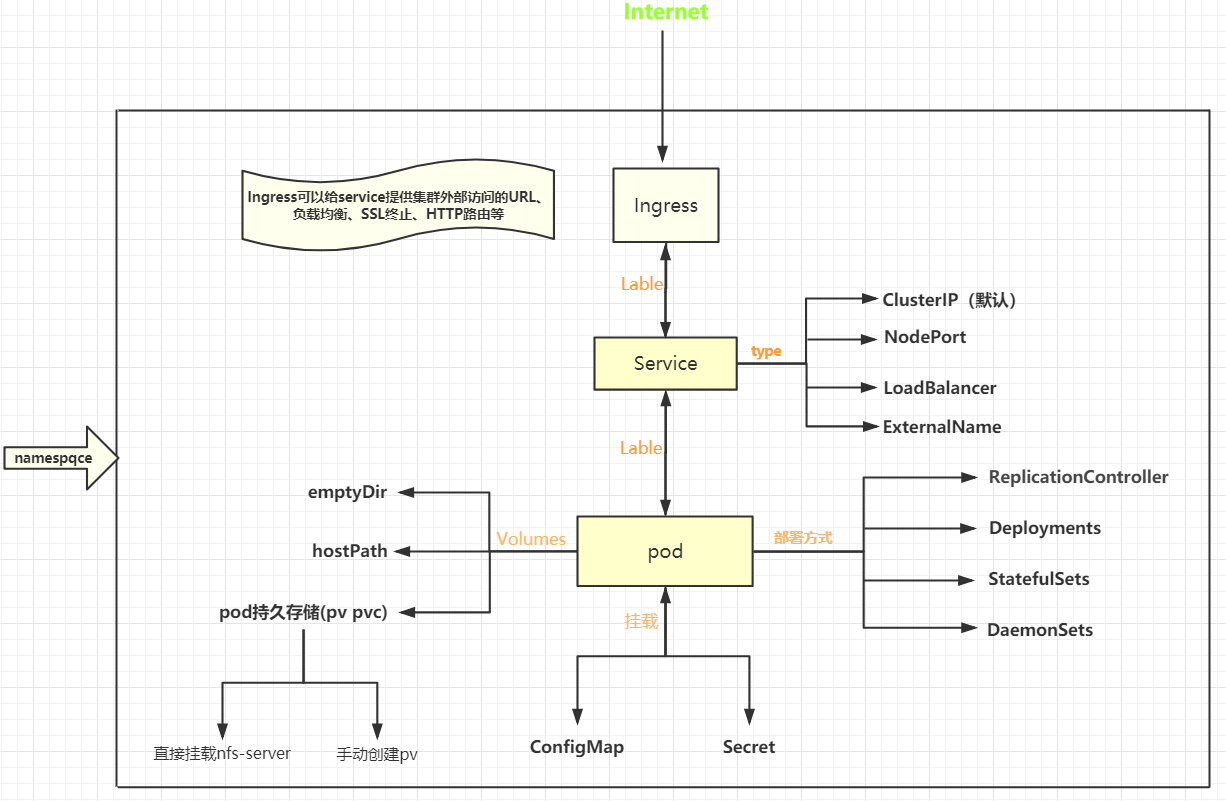
Kubernetes作为容器集群管理工具，于2015年7月22日迭代到 v 1.0并正式对外公布，这意味着这个开源容器编排系统可以正式在生产环境使用。与此同时，谷歌联合Linux基金会及其他合作伙伴共同成立了CNCF基金会( Cloud Native Computing Foundation)，并将Kuberentes 作为首个编入CNCF管理体系的开源项目，助力容器技术生态的发展进步。Kubernetes项目凝结了Google过去十年间在生产环境的经验和教训，从Borg的多任务Alloc资源块到Kubernetes的多副本Pod，从Borg的Cell集群管理，到Kubernetes设计理念中的联邦集群，在Docker等高级引擎带动容器技术兴起和大众化的同时，为容器集群管理提供独了到见解和新思路。

### 架构

Kubernetes集群包含有节点代理kubelet和Master组件(APIs, scheduler, etc)，一切都基于分布式的存储系统。



Kubernetes部署架构图



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（相当于mysql， etcd支持watch）
* 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
* Container runtime（上下文）负责镜像管理以及Pod和容器的真正运行（CRI）
* kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

除了核心组件，还有一些推荐的Add-ons：

* 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Web UI
* Federation提供跨可用区的集群
* Fluentd-elasticsearch提供集群日志采集、存储与查询

### 创建Kubernetes集群

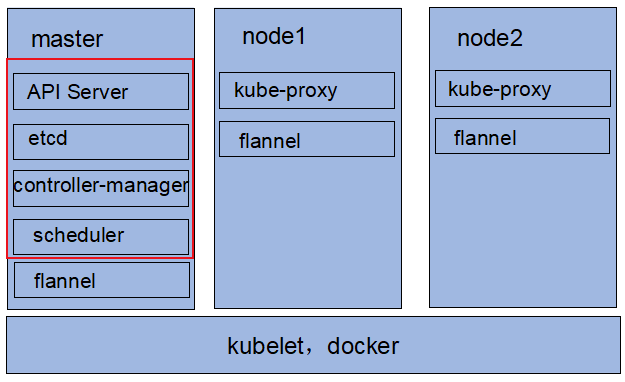
一般情况下，我们部署kubernetes集群有两种方式：

第一种方式：组件式安装

针对于master节点，将API Server、etcd、controller-manager、scheduler各组件进行yum install、编译安装或者展开安装的方式手动直接安装在master节点主机上，作为系统级守护进程运行。

第二种方式：采用kubeadm安装工具安装

每一个节点主机上包括master节点都要手动安装并运行docker，同时也都要手动安装并运行kubelet。如果将第一个节点初始化为master节点，在执行初始化这个步骤，其实就是通过kubeadm工具将API Server、etcd、controller-manager、scheduler各组件运行为Pod，也就是跑在docker上。而其他node节点，因已经运行了kubelet、docker组件，剩下的kube-proxy组件也是要运行在Pod上。



#### 基础环境配置

本次安装我们准备了三台CentOS 7服务器，其中一台Master节点，2台Node节点。其中Master节点上部署的组件有：Etcd、Kubelet、APIService Controller、Schedule和Docker等。Node节点上主要部署Kubelet和Docker。

|  |  |  |  |
| --- | --- | --- | --- |
| IP | 内网IP | 主机名 | 备注 |
| 10.0.0.50 | 172.16.1.50 | k8s-master | Master |
| 10.0.0.51 | 172.16.1.51 | k8s-node1 | Node |
| 10.0.0.52 | 172.16.1.52 | k8s-node2 | Node |

各节点采用主机名的方式，这种方式与IP地址相比具有较高的可扩展性。

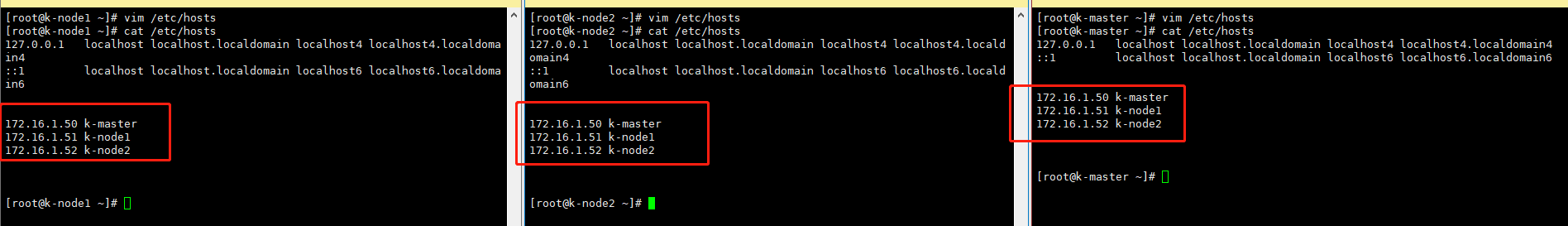
##### 修改主机名



* 为什么要修改主机名？

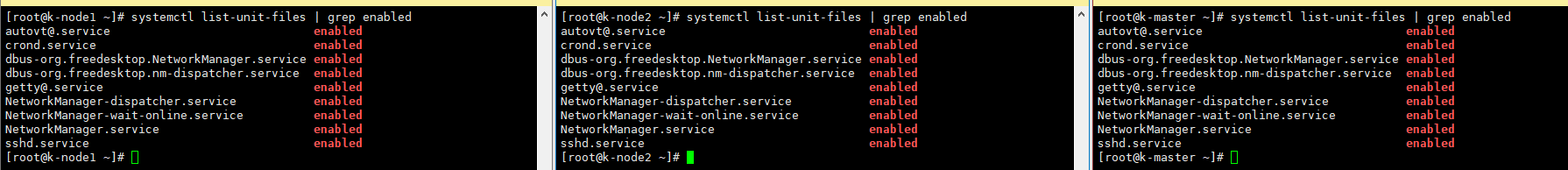
主要原因是kubernetes调度的时候会使用主机名进行调度。

##### 将所有节点配置Hosts



##### 关闭不必要的软件

$ systemctl list-unit-files | grep enabled | grep -v sshd | grep -v NetworkManager | grep -v ntp | grep -v @ | grep -v crond | awk '{print $1}' | xargs systemctl disable {} \;



##### 安装必要的包

$ mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.backup

$ curl -o /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.repo

$ mv /etc/yum.repos.d/epel.repo /etc/yum.repos.d/epel.repo.backup

$ mv /etc/yum.repos.d/epel-testing.repo /etc/yum.repos.d/epel-testing.repo.backup

$ curl -o /etc/yum.repos.d/epel.repo http://mirrors.aliyun.com/repo/epel-7.repo

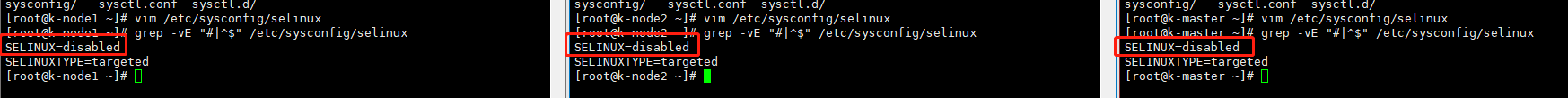
$ yum makecache

$ yum install wget expect vim net-tools ntp bash-completion ipvsadm ipset jq iptables conntrack sysstat libseccomp -y

##### 关闭Selinux

$ setenforce 0

将/etc/sysconfig/selinux文件中的SELINUX改为disabled。



##### 关闭swap分区

swapoff -a && sysctl -w

##### 同步所有机器上时间

$ ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime # 如果安装的时候已经设定了时区，这步可省略

$ echo "Asia/Shanghai" > /etc/timezone

$ ntpdate ntp.aliyun.com # 同步时间

$ crontab -e # 加入定时任务

\*/5 \* \* \* \* ntpdate ntp.aliyun.com 1>/dev/null

##### 使所有节点间免密登录

# 生成ssh key

$ ssh-keygen -t rsa

# 同步到其他节点

$ for i in k-master k-node1 k-node2;do

> expect -c "

> spawn ssh-copy-id -i /root/.ssh/id\_rsa.pub root@$i

> expect {

> \"\*yes/no\*\" {send \"yes\r\"; exp\_continue}

> \"\*password\*\" {send \"root\r\"; exp\_continue}

> \"\*Password\*\" {send \"root\r\";}

> } "

> done

[root@k8s-master ~]# for i in k8s-master k8s-node1 k8s-node2 ; do

> ssh-copy-id -i ~/.ssh/id\_rsa.pub root@$i

> done

# 测试

ssh root@k8s-node

##### 升级内核参数

Docker overlay2需要使用kernel 4.x版本，所以我们需要升级内核。我这里的内核使用4.18.9。CentOS 7.x 系统自带的 3.10.x 内核存在一些 Bugs，导致运行的 Docker、Kubernetes 不稳定。

$ wget http://mirror.rc.usf.edu/compute\_lock/elrepo/kernel/el7/x86\_64/RPMS/kernel-ml{,-devel}-4.18.9-1.el7.elrepo.x86\_64.rpm

# 同步到其他节点

$ for i in k-master k-node1 k-node2 ; do scp kernel-ml{,-devel}-4.18.16-1.el7.elrepo.x86\_64.rpm $i:/root; done

$ yum localinstall -y kernel-ml\*

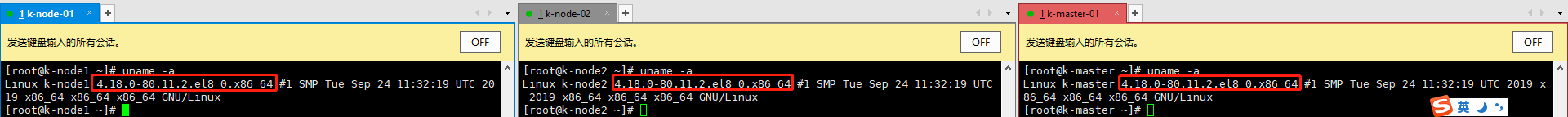
$ # 修改内核启动顺序

$ grub2-set-default 0 && grub2-mkconfig -o /etc/grub2.cfg

$ # 查看（可执行可不执行）

$ grubby --default-kernel

$ reboot



##### 修改内核参数

$ cat > /etc/sysctl.d/kubernetes.conf <<EOF

net.bridge.bridge-nf-call-iptables=1

net.bridge.bridge-nf-call-ip6tables=1

net.ipv4.ip\_forward=1

net.ipv4.tcp\_tw\_recycle=0

vm.swappiness=0

vm.overcommit\_memory=1

vm.panic\_on\_oom=0

fs.inotify.max\_user\_instances=8192

fs.inotify.max\_user\_watches=1048576

fs.file-max=52706963

fs.nr\_open=52706963

net.ipv6.conf.all.disable\_ipv6=1

net.netfilter.nf\_conntrack\_max=2310720

EOF

$ sysctl -p /etc/sysctl.d/kubernetes.conf

##### 安装IPVS

在本教程中，kube-proxy均采用ipvs模式，该模式也是新版本默认支持的代理模式，性能比iptables要高，如果服务器未装了ipvs，将会转换成iptables模式。这里我们设置开机自动加载：

$ cat > /etc/sysconfig/modules/k8s.modules <<EOF

modprobe -- ip\_vs

modprobe -- ip\_vs\_rr

modprobe -- ip\_vs\_wrr

modprobe -- ip\_vs\_sh

modprobe -- nf\_conntrack\_ipv4

modprobe -- ip\_tables

modprobe -- ip\_set

modprobe -- xt\_set

modprobe -- ipt\_set

modprobe -- ipt\_rpfilter

modprobe -- ipt\_REJECT

modprobe -- ipip

EOF

# 查看是否加载,如未看到信息可以重启试试。

$ chmod 755 /etc/sysconfig/modules/k8s.modules && bash /etc/sysconfig/modules/k8s.modules && lsmod | grep -e ip\_vs -e nf\_conntrack\_ipv4



#### 3.2 搭建集群

##### 3.2.1. 安装Docker

yum install -y yum-utils device-mapper-persistent-data lvm2

yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

yum install docker-ce -y

mkdir /etc/docker

cat > /etc/docker/daemon.json <<EOF

{

"exec-opts": ["native.cgroupdriver=systemd"],

"log-driver": "json-file",

"log-opts": {

"max-size": "100m"

},

"storage-driver": "overlay2",

"storage-opts": [

"overlay2.override\_kernel\_check=true"

]

}

EOF

mkdir -p /etc/systemd/system/docker.service.d

systemctl daemon-reload

systemctl restart docker

##### 3.2.1. Master节点安装Kubernetes

$ cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86\_64/

enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg

EOF

$ setenforce 0

$ yum install -y kubelet kubeadm kubectl

##### 3.2.3. Node节点上安装kubernetes

$ cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86\_64/

enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg

EOF

$ setenforce 0

$ yum install -y kubelet kubeadm

##### 3.2.4. 设置开机自启动

systemctl enable docker.service kubelet.service

##### 3.2.5. 初始化Kubernetes集群Master节点

###### 3.2.5.1. 设置忽略Swap

vim /etc/sysconf/kubelet # 添加如下内容

KUBELET\_EXTRA\_ARGS="--fail-swap-on=false"

###### 3.2.5.2. 安装

kubeadm init \

--apiserver-advertise-address=172.16.1.50 \ # 可不要（设置Master节点IP）

--image-repository=registry.aliyuncs.com/google\_containers \

--kubernetes-version=v1.17.0 \ # 可不要（设置安装的k8s版本）

--service-cidr=10.96.0.0/12 \

--pod-network-cidr=10.244.0.0/16 \

--ignore-preflight-errors=Swap # 如果没有Swap分区，可不要

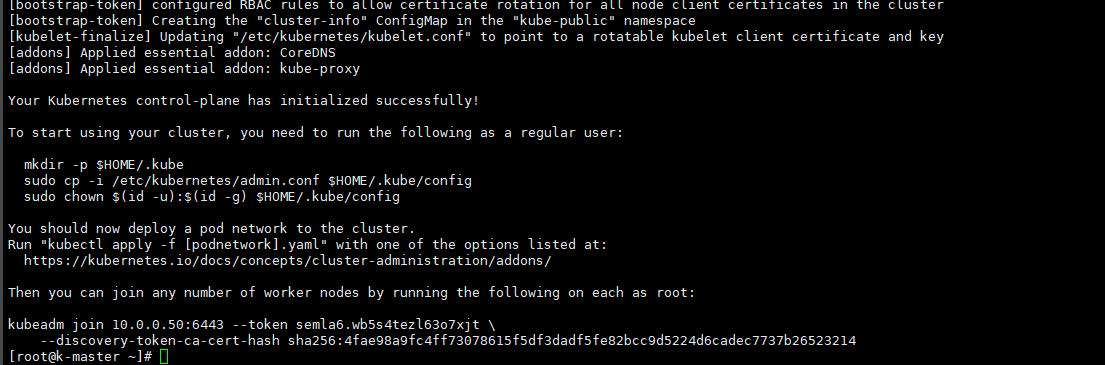
# 没特殊情况下

kubeadm init \

--image-repository=registry.aliyuncs.com/google\_containers \

--service-cidr=10.96.0.0/12 \

--pod-network-cidr=10.244.0.0/16



安装提示执行一下命令：

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

##### 3.2.5.2. Kubectl命令自动补全

yum install -y bash-completion

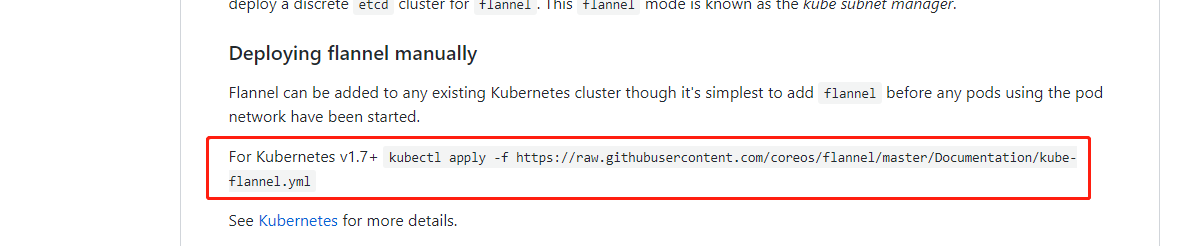
source /usr/share/bash-completion/bash\_completion

source <(kubectl completion bash)

echo "source <(kubectl completion bash)" >> ~/.bashrc

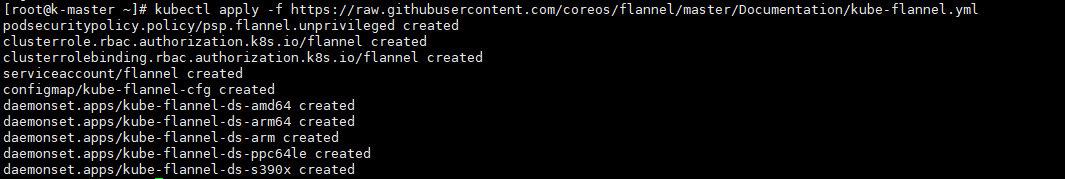
##### 3.2.5.3. 安装网络插件

#到GitHub上搜flannel网络插件。

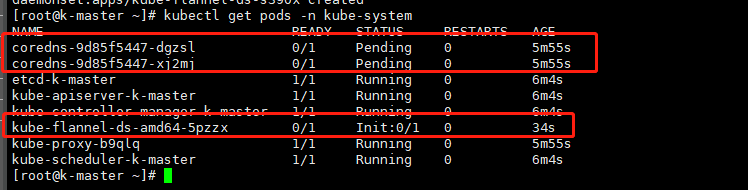


对于Kubernetes 1.7 以上可以执行上面命令安装网络插件：

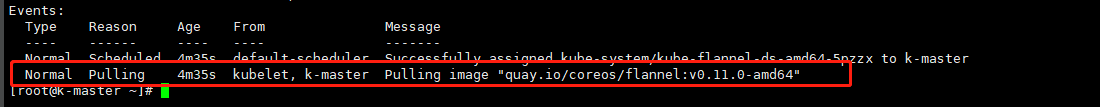
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml



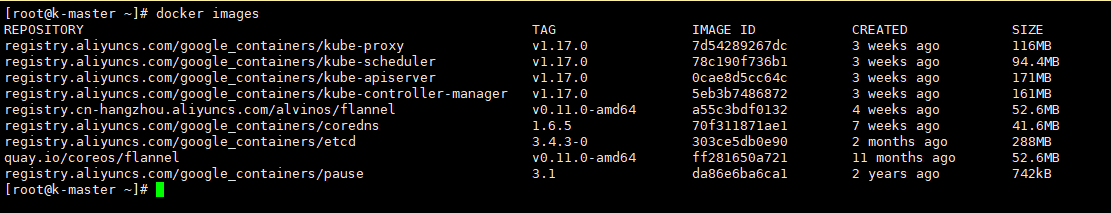
我们可以看到网络插件flannel正在初始化中：



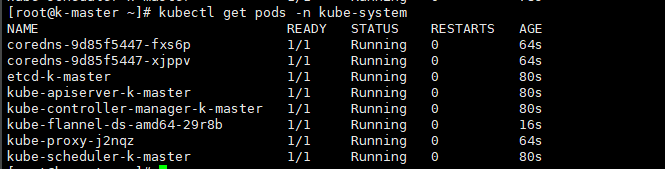
我们可以通过命令查看一下flannel详情：



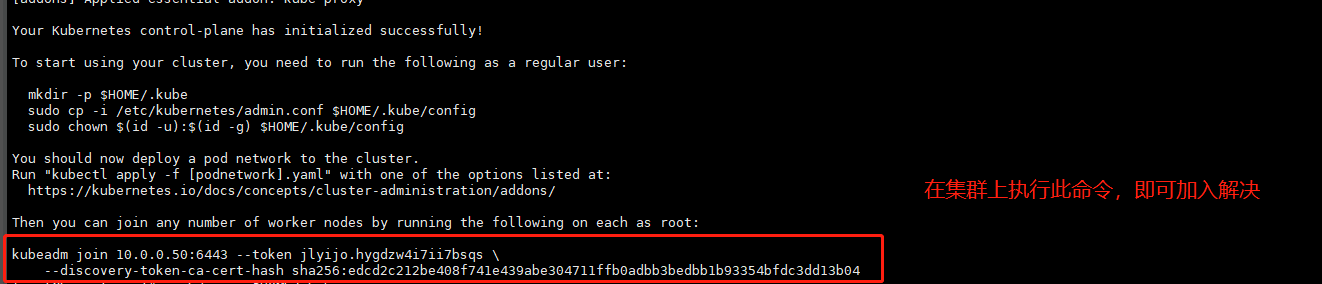
可以明显的看出，镜像拉去不下来；但是我们可以尝试以下方式。

docker pull quay-mirror.qiniu.com/coreos/flannel:v0.11.0-amd64; docker tag quay-mirror.qiniu.com/coreos/flannel:v0.11.0-amd64 quay.io/coreos/flannel:v0.11.0-amd64

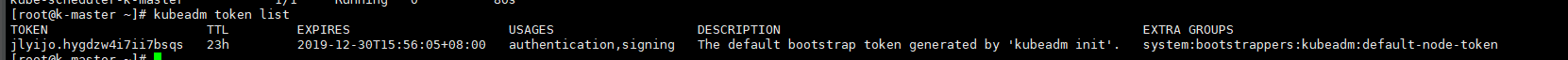
然后查看DNS：



##### 3.2.5.4. 将Node节点加入集群



有时候我们难免会忘记这个token，如果忘记，我们可以创建一个，首先我们看一下我们集群有多少个Token：



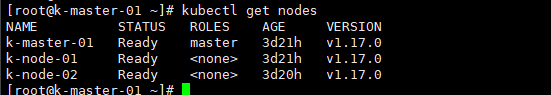
创建方式：

[root@k-master ~]# kubeadm token create --print-join-command

kubeadm join 10.0.0.50:6443 --token 038qwm.hpoxkc1f2fkgti3r --discovery-token-ca-cert-hash sha256:edcd2c212be408f741e439abe304711ffb0adbb3bedbb1b93354bfdc3dd13b04

# 注：上面的IP(10.0.0.50要改成172.16.1.50，因为Kubernetes在解析的时候回找主机名解析，如果使用10.0.0.50的话，很可能会报错。如果是本地虚拟机就修改不是就不需要。)

加入之后，我们可以查看一下是否成功：



### Pod

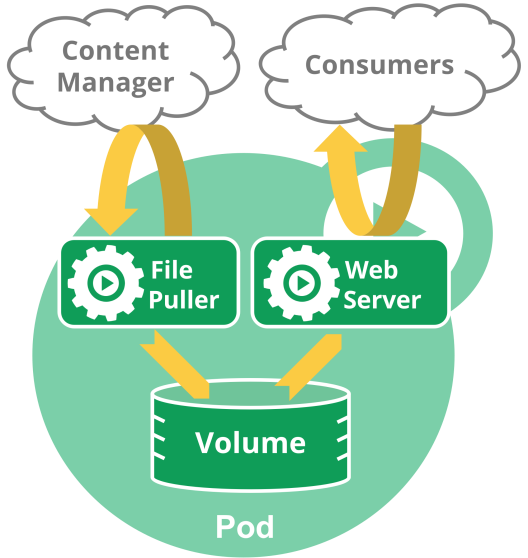
K8s有很多技术概念，同时对应很多API对象，最重要的也是最基础的是微服务Pod。Pod是在K8s集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8s最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod是K8s集群中所有业务类型的基础，可以看作运行在K8s集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前K8s中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和PetSet，本文后面会一一介绍。

#### 4.1. 什么是POD

Pod是Kubernetes中能够创建和部署的最小单元，是Kubernetes集群中的一个应用实例，总是部署在同一个节点Node上。Pod中包含了一个或多个容器，还包括了存储、网络等各个容器共享的资源。Pod支持多种容器环境，Docker则是最流行的容器环境。

* 单容器Pod，最常见的应用方式。
* Pod中不同容器之间的端口是不能冲突的
* 多容器Pod，对于多容器Pod，Kubernetes会保证所有的容器都在同一台物理主机或虚拟主机中运行。多容器Pod是相对高阶的使用方式，除非应用耦合特别严重，一般不推荐使用这种方式。一个Pod内的容器共享IP地址和端口范围，容器之间可以通过 localhost 互相访问。



Pod并不提供保证正常运行的能力，因为可能遭受Node节点的物理故障、网络分区等等的影响，整体的高可用是Kubernetes集群通过在集群内调度Node来实现的。通常情况下我们不要直接创建Pod，一般都是通过Controller来进行管理，但是了解Pod对于我们熟悉控制器非常有好处。

#### 4.2. Pod带来的好处

* Pod做为一个可以独立运行的服务单元，简化了应用部署的难度，以更高的抽象层次为应用部署管提供了极大的方便。
* Pod做为最小的应用实例可以独立运行，因此可以方便的进行部署、水平扩展和收缩、方便进行调度管理与资源的分配。
* Pod中的容器共享相同的数据和网络地址空间，Pod之间也进行了统一的资源管理与分配。

#### 4.3.Pod内如何管理多个容器

Pod中可以同时运行多个进程（作为容器运行）协同工作。同一个Pod中的容器会自动的分配到同一个 node 上。同一个Pod中的容器共享资源、网络环境和依赖，它们总是被同时调度。  
注意在一个Pod中同时运行多个容器是一种比较高级的用法。只有当你的容器需要紧密配合协作的时候才考虑用这种模式。

#### 4.4. Pod的持久性

Pod在设计⽀持就不是作为持久化实体的。在调度失败、节点故障、缺少资源或者节点维护的状态下都会死掉会被驱逐。  
通常，⽤户不需要⼿动直接创建Pod， ⽽是应该使⽤controller（例如Deployments） ， 即使是在创建单个Pod的情况下。 Controller可以提供集群级别的⾃愈功能、 复制和升级管理。  
Pod原语有利于：

* 调度程序和控制器可插拔性
* 支持pod级操作，无需通过控制器API代理他们
* 将pod生命周期与控制器生命周期分离
* 控制器与服务的分离，端点控制器只是监视pod
* 将集群集功能与kubelet及共鞥的清晰组合
* 高可用性应用程序，他们可以在终止前及在删除前更换pod

#### 4.5. Pod的终止

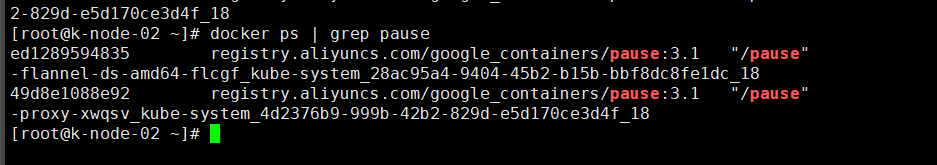
因为Pod作为在集群的节点上运⾏的进程， 所以在需要的时候能够优雅的终⽌掉是⼗分必要的（⽐起使⽤发送KILL信号这种暴⼒的⽅式）。⽤户需要能够放松删除请求， 并且知道它们何时会被终⽌， 是否被正确的删除。 ⽤户想终⽌程序时发送删除pod的请求， 在pod可以被强制删除前会有⼀个宽限期， 会发送⼀个TERM请求到每个容器的主进程。 ⼀旦超时， 将向主进程发送KILL信号并从API server中删除。 如果kubelet或者container manager在等待进程终⽌的过程中重启， 在重启后仍然会重试完整的宽限期。  
示例流程如下：

1. 用户发送删除pod的命令，默认宽限期30秒
2. 在Pod超过该宽限期后API server就会更新Pod的状态为dead
3. 在客户端命令行上显示的Pod的状态为为terminating；
4. 跟上一部同时，当kubelet发现pod被标记为terminating时，开始停止pod进程：
   1. 如果在pod中定义了preStop hook，在停止pod前会被调用。如果宽限期后 preStop hook仍然在运行，第二部会增加2秒宽限期
   2. 向Pod中的进程发送TERM信号
5. 跟第三部同时，该Pod将从该service的端点列表中删除，不再是replication controller中的一部分，关闭的慢的pod将继续处理load balancer转发的流量
6. 过了宽限期后，将向Pod中依然存在的进程发SIGKILL信号杀掉进程。

kubelet会在APIserver中完成Pod的删除，通过将优雅周期设置为0（立即删除）Pod在API中消失，并且客户端也不可见。

#### 4.6 Pause容器

Pause容器，又叫Infra容器。我们检查node节点的时候会发现每个node上都运行了很多的pause容器



kubernetes中的pause容器主要为每个业务容器提供以下功能：

* 在pod中担任Linux命名空间共享的基础；
* 启用pid命名空间，开启init进程。
* 共享网络。

#### 4.7 Pod的生命周期

像单独的容器应用一样，Pod并不是持久运行的。Pod创建后，Kubernetes为其分配一个UID，并且通过Controller调度到Node中运行，然后Pod一直保持运行状态直到运行正常结束或者被删除。在Node发生故障时，Controller负责将其调度到其他的Node中。Kubernetes为Pod定义了几种状态，分别如下：

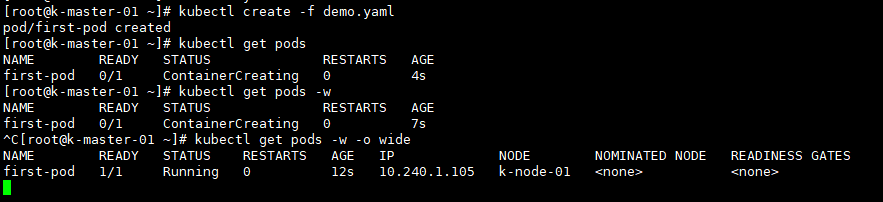
* Pending，Pod已创建，正在等待容器创建。经常是正在下载镜像，因为这一步骤最耗费时间。
* Running，Pod已经绑定到某个Node并且正在运行。或者可能正在进行意外中断后的重启。
* Succeeded，表示Pod中的容器已经正常结束并且不需要重启。
* Failed，表示Pod中的容器遇到了错误而终止。
* Unknown，因为网络或其他原因，无法获取Pod的状态。

#### 4.8 常用Pod管理命令

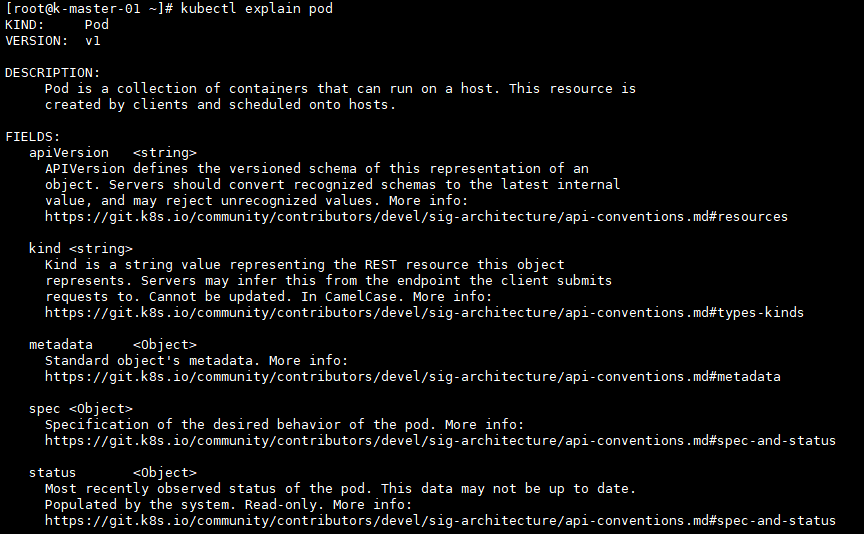
Pod的配置信息中有几个重要部分，apiVersion、kind、metadata(元数据)、spec以及status。其中apiVersion和kind是比较固定的，status是运行时的状态，所以最重要的就是metadata和spec两个部分。

先来看一个典型的配置文件，命名为 first-pod.yml

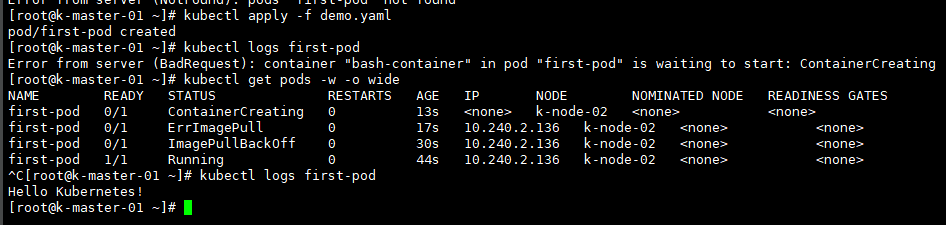
apiVersion: v1  
kind: Pod  
metadata:  
 name: first-pod  
 labels:  
 app: bash  
 tir: backend  
spec:  
 containers:  
 - name: bash-container  
 image: busybox  
 command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 10']



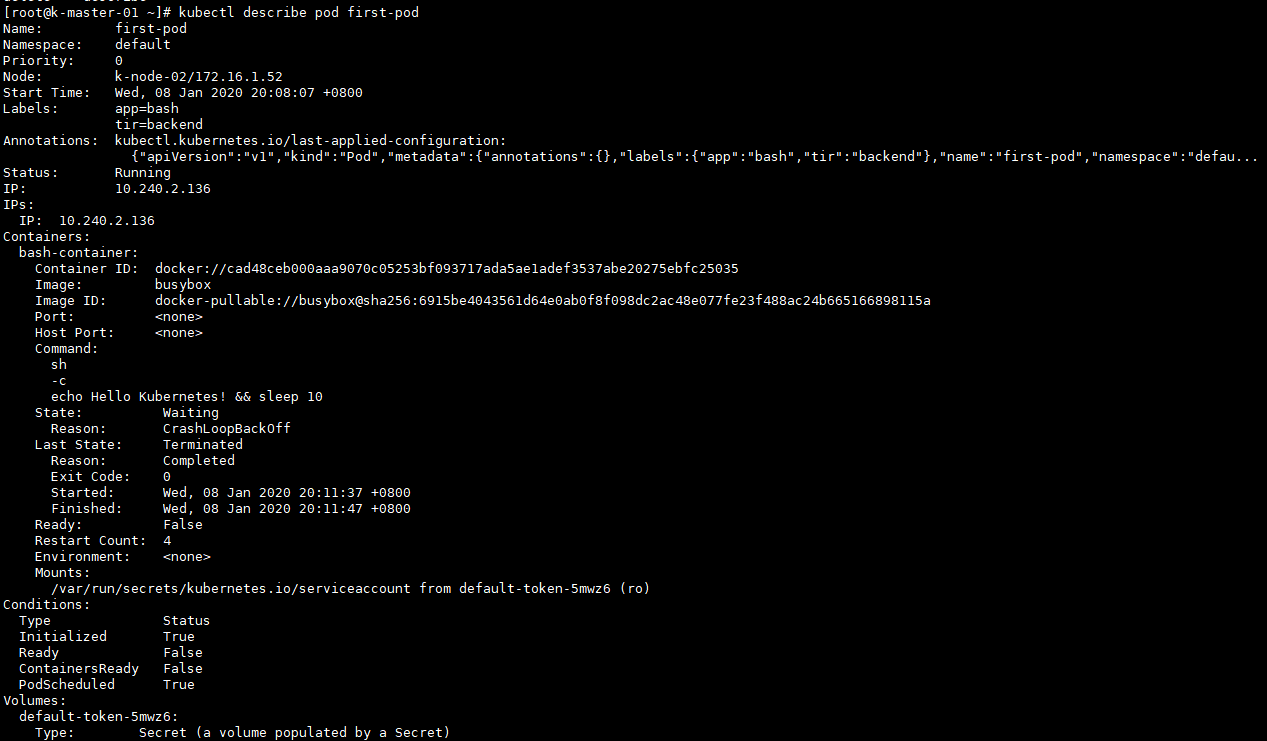
查看一下Pod的配置信息



查看一下Pod 的日志



查看运行中Pod详细信息



#### 4.9. 标签管理

Label是Kubernetes系统中另外一个核心概念。一个Label是一个key=value的键值对，其中key与vaue由用户自己指定。Label可以附加到各种资源对象上，例如Node、Pod、Service、RC等，一个资源对象可以定义任意数量的Label，同一个Label也可以被添加到任意数量的资源对象上去，Label通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除。

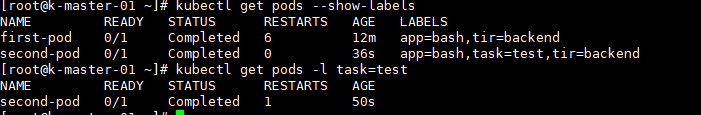
我们可以通过指定的资源对象捆绑一个或多个不同的Label来实现多维度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或者监控和分析应用（日志记录、监控、告警）等。一些常用等label示例如下。

* 版本标签："release" : "stable" , "release" : "canary"
* 环境标签："environment" : "dev" , "environment" : "production"
* 架构标签："tier" : "frontend" , "tier" : "backend" , "tier" : "middleware"
* 分区标签："partition" : "customerA" , "partition" : "customerB"
* 质量管控标签："track" : "daily" , "track" : "weekly"

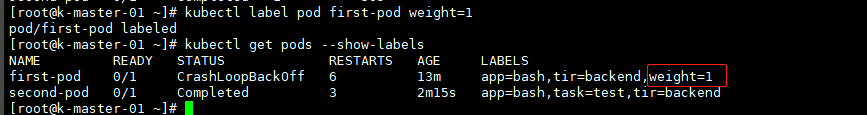
Label相当于我们熟悉的“标签”，給某个资源对象定义一个Label，就相当于給它打了一个标签，随后可以通过Label Selector（标签选择器）查询和筛选拥有某些Label的资源对象，Kubernetes通过这种方式实现了类似SQL的简单又通用的对象查询机制。



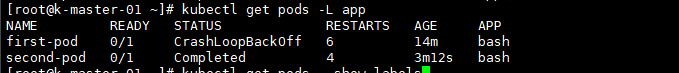
##### 4.9.1. 根据标签来查询Pod



##### 4.9.2. 增加标签



##### 4.9.3 可以将标签显示为列



#### 4.10. 删除Pod

删除Pod有两种方式：

1. 命令行删除

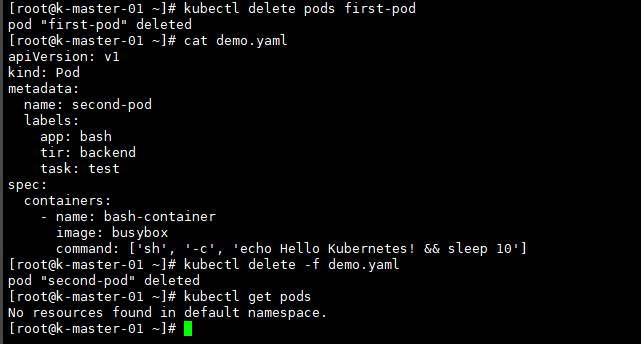
kubectl delete pods first-pod

1. 使用yaml文件删除

kubectl delete -f demo.yaml

1. 使用标签删除

kubectl delete pods -l tir=backend



#### 4.11. Pod进行健康检查

健康检查最简单的方式就是检查进程的状态。Kubelet 不断的询问 Docker daemon 这个容器进程是否还在运行，如果没有，这个容器就会被重启。目前在所有 Kubernetes 的案例中，这种健康检查是一直开启的。对与 Kubernetes 中所有运行的容器都是生效的。

在pod生命周期中可以做的一些事情。主容器启动前可以完成初始化容器，初始化容器可以有多个，他们是串行执行的，执行完成后就退出了，在主程序刚刚启动的时候可以指定一个post start 主程序启动开始后执行一些操作，在主程序结束前可以指定一个 pre stop 表示主程序结束前执行的一些操作。在程序启动后可以做两类检测 liveness probe（存活性探测） 和 readness probe（就绪性探测）

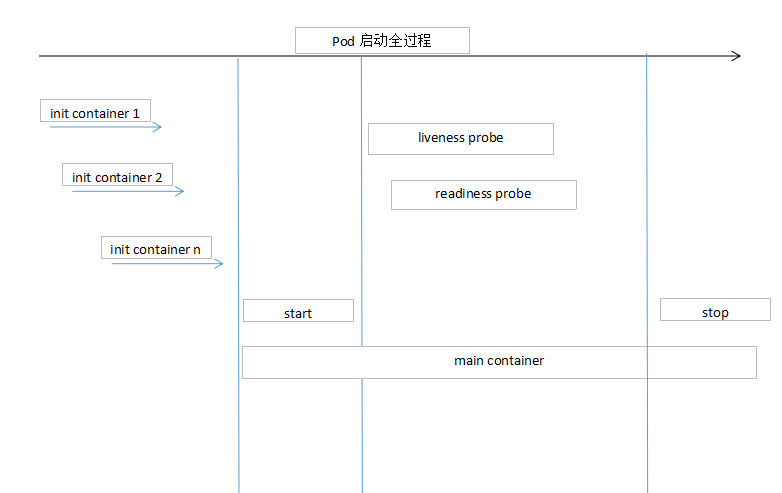
Kubernetes利用[Handler](https://godoc.org/k8s.io/kubernetes/pkg/api/v1" \l "Handler)功能，可以对容器的状况进行探测，有以下三种形式。

* ExecAction：在容器中执行特定的命令。
* TCPSocketAction：检查容器端口是否可以连接。
* HTTPGetAction：检查HTTP请求状态是否正常。

Kubelet 可以选择是否执行在容器上运行的两种探针执行和做出反应：

* LivenessProbe探针（存活性探测）：  
  用于判断容器是否存活，即Pod是否为running状态，如果LivenessProbe探针探测到容器不健康，则kubelet将kill掉容器，并根据容器的重启策略是否重启，如果一个容器不包含LivenessProbe探针，则Kubelet认为容器的LivenessProbe探针的返回值永远成功。
* ReadinessProbe探针（就绪性探测）：  
  用于判断容器是否正常提供服务，即容器的Ready是否为True，是否可以接收请求，如果ReadinessProbe探测失败，则容器的Ready将为False，控制器将此Pod的Endpoint从对应的service的Endpoint列表中移除，从此不再将任何请求调度此Pod上，直到下次探测成功。（剔除此pod不参与接收请求不会将流量转发给此Pod
* Lifecycle

定义容器启动后和终止前立即执行的动作



##### 4.11.1. LivenessProbe探针（存活性探测）



参数解析：

1. failureThreshold：最少连续几次探测失败的次数，满足该次数则认为fail
2. initialDelaySeconds：容器启动之后开始进行存活性探测的秒数。不填立即进行
3. periodSeconds：执行探测的频率（秒）。默认为10秒。最小值为1。
4. successThreshold：最少连续几次探测成功的次数，满足该次数则认为success。
5. timeoutSeconds：每次执行探测的超时时间

###### 4.11.1.1. ExecAction

apiVersion: v1

kind: Pod

metadata:

name: probe-exec

namespace: default

spec:

containers:

- name: nginx

image: nginx

livenessProbe:

exec:

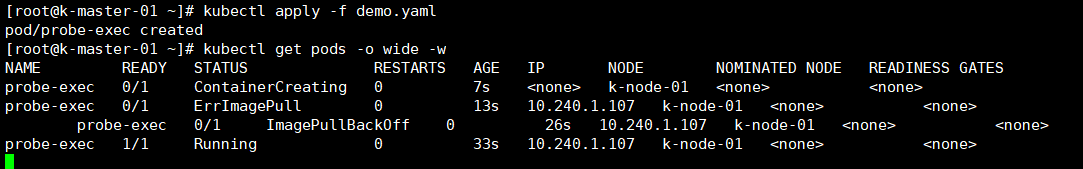
command:

- cat

- /tmp/health

initialDelaySeconds: 5

timeoutSeconds: 1



###### 4.11.1.2. TCPSocketAction

apiVersion: v1

kind: Pod

metadata:

name: probe-tcp

namespace: default

spec:

containers:

- name: nginx

image: nginx

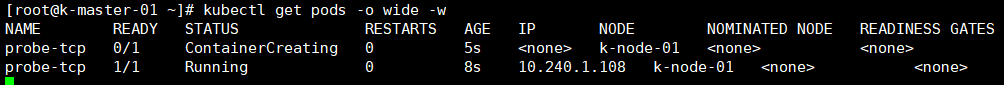
livenessProbe:

initialDelaySeconds: 5

timeoutSeconds: 1

tcpSocket:

port: 80



###### 4.11.1.3. HTTPGetAction

apiVersion: v1

kind: Pod

metadata:

name: probe-http

namespace: default

spec:

containers:

- name: nginx

image: nginx

livenessProbe:

httpGet:

path: /

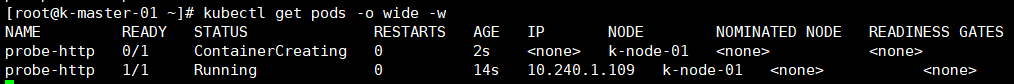
port: 80

host: 127.0.0.1

scheme: HTTP

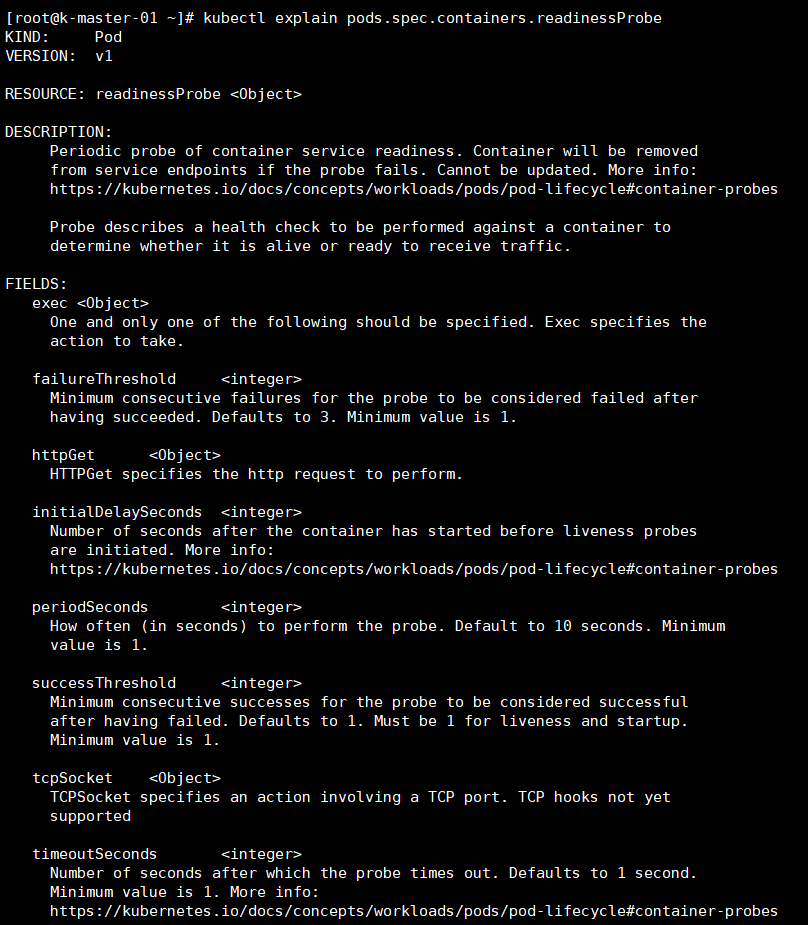
initialDelaySeconds: 5

timeoutSeconds: 1



* 补充：
  + 当使用kubectl exec时一定得保证对应的conteiner是存活且正常状态。
  + 如果一个POD中有多个容器，可使用-c参数来指定要进入的容器。

#### 4.11.2. ReadinessProbe探针（就绪性探测）



参数解析：

1. failureThreshold：最少连续几次探测失败的次数，满足该次数则认为fail
2. initialDelaySeconds：容器启动之后开始进行存活性探测的秒数。不填立即进行
3. periodSeconds：执行探测的频率（秒）。默认为10秒。最小值为1。
4. successThreshold：最少连续几次探测成功的次数，满足该次数则认为success。
5. timeoutSeconds：每次执行探测的超时时间

#### 4.11.3. 案例

kind: Pod

apiVersion: v1

metadata:

name: readinessprobe-nginx

namespace: default

labels:

provider: aliyun

business: pms

environmental: dev

spec:

containers:

- name: readinessprobe-nginx

image: nginx

imagePullPolicy: Always

ports:

- containerPort: 80

name: http

protocol: TCP

- containerPort: 443

name: https

protocol: TCP

readinessProbe:

httpGet:

port: 80

path: /demo.html

livenessProbe:

httpGet:

port: 80

path: /index.html

lifecycle:

postStart:

exec:

command: ["touch", "/usr/share/nginx/html/demo.html"]

### 5.ReplicaSet

Kubernetes 中的 ReplicaSet 主要的作用是维持一组 Pod 副本的运行，它的主要作用就是保证一定数量的 Pod 能够在集群中正常运行，它会持续监听这些 Pod 的运行状态，在 Pod 发生故障重启数量减少时重新运行新的 Pod 副本。

#### 5.1. 创建

ReplicaSert除了常见的 apiVersion、kind 和 metadata 属性之外，规格中总共包含三部分重要内容，也就是 Pod 副本数目 replicas、选择器 selector 和 Pod 模板 template，这三个部分共同定义了 ReplicaSet 的规格：

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: frontend

labels:

app: guestbook

tier: frontend

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

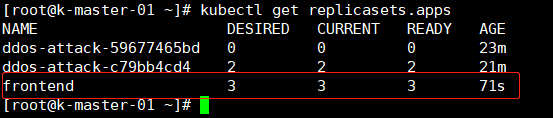
tier: frontend

spec:

containers:

- name: php-redis

image: redis



同一个 ReplicaSet 会使用选择器 selector 中的定义查找集群中查找自己持有的 Pod 对象，它们会根据标签的匹配获取能够获得的 Pod。所有 ReplicaSet 对象的增删改查都是由 ReplicationController 控制器完成的，该控制器会通过 Informer 监听 ReplicaSet 和 Pod 的变更事件并将其加入持有的待处理队列ReplicationController 中的 queue 其实就是一个存储待处理 ReplicaSet 的对象池，它运行的几个 Goroutine 会从队列中取出最新的数据进行处理。ReplicationController 启动的多个 Goroutine 会从队列中取出待处理的任务，然后调用 syncReplicaSet 进行同步，这个方法会按照传入的 key 从 etcd 中取出 ReplicaSet 对象，然后取出全部 Active 的 Pod。随后执行的 ClaimPods 方法会获取一系列 Pod 的所有权，如果当前的 Pod 与 ReplicaSet 的选择器匹配就会建立从属关系，否则就会释放持有的对象，或者直接忽视无关的 Pod，建立和释放关系的方法就是 AdoptPod 和 ReleasePod，AdoptPod 会设置目标对象的 metadata.OwnerReferences 字段。

* 镜像拉取规则的三种方式！
  + Always：总是到远程拉取最新镜像。
  + Never：不管本地有没有都不去远程拉取镜像
  + IfNotPresent：如果本地没有，就去远程拉取。

### 6.ReplicationController

ReplicationController 确保在任何时候都有特定数量的 pod 副本处于正常运行状态。 换句话说，ReplicationController 确保一个 pod 或一组同类的 pod 总是可用的。当 pods 数量过多时，ReplicationController 会终止多余的 pods。当 pods 数量太少时，ReplicationController 将会启动新的 pods。 与手动创建的 pod 不同，由 ReplicationController 创建的 pods 在失败、被删除或被终止时会被自动替换。 例如，在中断性维护（如内核升级）之后，您的 pod 会在节点上重新创建。 因此，即使您的应用程序只需要一个 pod，您也应该使用一个 ReplicationController。 ReplicationController 类似于进程管理器，但是 ReplicationController 不是监控单个节点上的单个进程，而是监控跨多个节点的多个 pods。

**示例：**

kind: ReplicationController

apiVersion: v1

metadata:

name: nginx

namespace: default

spec:

replicas: 2

selector:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

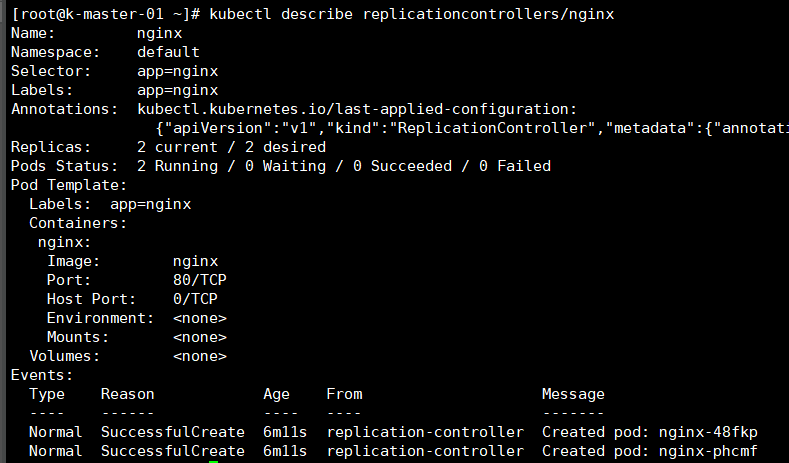
ports:

- containerPort: 80

name: http

检查ReplicationController的状态：

kubectl describe replicationcontrollers/nginx



要以机器可读的形式列出属于 ReplicationController 的所有 pod，可以使用如下命令

kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name}

ReplicaSet 是下一代 ReplicationController ，支持新的基于集合的标签选择器。 它主要被 Deployment 用来作为一种编排 pod 创建、删除及更新的机制。 请注意，我们推荐使用 Deployment 而不是直接使用 ReplicaSet，除非您需要自定义更新编排或根本不需要更新。

Deployment 是一种更高级别的 API 对象，它以类似于 kubectl rolling-update 的方式更新其底层 ReplicaSet 及其 Pod。 如果您想要这种滚动更新功能，那么推荐使用 Deployment，因为与 kubectl rolling-update 不同，它们是声明式的、服务端的，并且具有其它特性。

### Deployment

Deployment为Pod和Replica Set（升级版的 Replication Controller）提供声明式更新。您在Deployment对象中描述所需的状态，然后Deployment控制器将实际状态以受控的速率更改为所需的状态。您可以定义部署以创建新的副本集，或删除现有部署并在新部署中采用其所有资源。

* 注释：
  + K8S创建的资源名称都必须小写，只支持英文。
  + K8S资源名称中只支持中划线。

#### 7.1 创建

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

##### 7.1.1 命令行执行创建

[root@k-master-01 ~]# kubectl apply -f demo.yaml

deployment.apps/nginx-deployment created

[root@k-master-01 ~]# kubectl get deployments.apps

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 0/3 0 0 1s

将kubectl标志设置--record为true允许您将当前命令记录在正在创建或更新的资源的注释中。这对于将来的自省很有用。

##### 7.1.2 查看部署状态

[root@k-master-01 ~]# kubectl rollout status deployment nginx-deployment

deployment "nginx-deployment" successfully rolled out

隔几秒钟再次查看

[root@k-master-01 ~]# kubectl get deployments.apps

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 3/3 3 3 57s

这表明Deployment已创建了所有三个副本，并且所有副本都是最新的（包含最新的Pod模板）并且可用（Pod状态至少已为Deployment的状态准备就绪.spec.minReadySeconds）。运行 kubectl get rs并kubectl get pods显示创建的ReplicaSet（RS）和Pod。

[root@k-master-01 ~]# kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-85ff79dd56 3 3 3 7m9s

您可能会注意到ReplicaSet的名称始终为<the name of the Deployment>-<hash value of the pod template>

[root@k-master-01 ~]# kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

nginx-deployment-85ff79dd56-8946c 1/1 Running 0 9m25s app=nginx,pod-template-hash=85ff79dd56

nginx-deployment-85ff79dd56-dl4js 1/1 Running 0 9m25s app=nginx,pod-template-hash=85ff79dd56

nginx-deployment-85ff79dd56-zb26d 1/1 Running 0 9m25s app=nginx,pod-template-hash=85ff79dd56

创建的ReplicaSet确保始终有三个nginx Pod。

**Pod模板哈希标签**

注意上面的pod标签中示例输出中的pod-template-hash标签。此标签由Deployment控制器添加到Deployment创建或采用的每个ReplicaSet中。其目的是确保部署的子副本集不重叠。它是通过对ReplicaSet的PodTemplate进行哈希处理并将所得的哈希值用作标签值来计算的，该值将添加到ReplicaSet选择器，窗格模板标签以及ReplicaSet可能具有的任何现有Pod中。

* 1. 更新

假设我们现在要更新nginx Pods以使用nginx:latest镜像而不是nginx:1.16镜像。

[root@k-master-01 ~]# kubectl set image deployment/nginx-deployment nginx=nginx:1.16

deployment.apps/nginx-deployment image updated

要查看部署状态

[root@k-master-01 ~]# kubectl rollout status deployment nginx-deployment

deployment "nginx-deployment" successfully rolled out

部署成功后

[root@k-master-01 ~]# kubectl get deployments.apps

NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 3/3 3 3 36m

下次我们要更新这些Pod时，我们只需要再次更新Deployment的pod模板。

* 1. 回滚

有时可能想回滚部署；例如，当部署不稳定时（例如崩溃循环）。默认情况下，所有“部署”的推出历史记录都保留在系统中，以便可以随时回滚（可以通过修改修订历史记录限制来更改它）。

* + 1. 列出历史

[root@k-master-01 ~]# kubectl rollout history deployment nginx-deployment

deployment.apps/nginx-deployment

REVISION CHANGE-CAUSE

2 kubectl apply --filename=demo.yaml --record=true

3 kubectl apply --filename=demo.yaml --record=true

4 kubectl apply --filename=demo.yaml --record=true

* + 1. 回滚到版本3

[root@k-master-01 ~]# kubectl rollout undo deployment nginx-deployment --to-revision=3

deployment.apps/nginx-deployment rolled back

[root@k-master-01 ~]# kubectl rollout history deployment nginx-deployment

deployment.apps/nginx-deployment

REVISION CHANGE-CAUSE

2 kubectl apply --filename=demo.yaml --record=true

4 kubectl apply --filename=demo.yaml --record=true

5 kubectl apply --filename=demo.yaml --record=true

* 1. 扩容

命令行中的三种方式

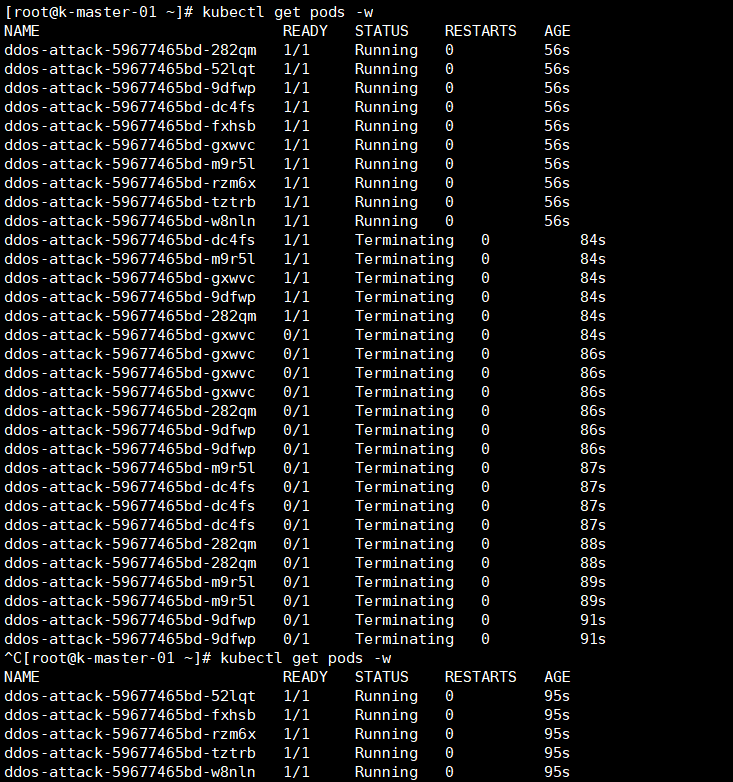
* + 1. Patch

[root@k-master-01 ~]# kubectl patch deploy ddos-attack -p '{"spec":{"replicas": 5}}'

deployment.apps/ddos-attack patched

# 打补丁的格式

Kubectl patch 需要修改的资源 需要修改的资源名称 -p ‘’



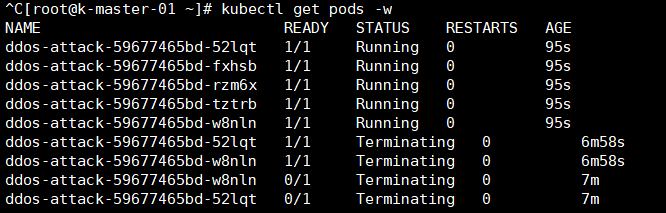
* + 1. Scale

[root@k-master-01 ~]# kubectl scale deployment ddos-attack --replicas=3

deployment.apps/ddos-attack scaled

# 步骤解释

Kubectl scale 资源类型 资源名称 修改项



* + 1. 修改配置文件中的replicas
  1. 暂停和恢复

您可以在触发一个或多个更新之前暂停部署，然后再恢复它。这样，您可以在暂停和恢复之间应用多个修复程序，而不会触发不必要的部署。

[root@k-master-01 ~]# kubectl rollout pause deployment/nginx-deployment

deployment.apps/nginx-deployment paused

然后更新部署的映像

[root@k-master-01 ~]# kubectl rollout history deployment nginx-deployment

deployment.apps/nginx-deployment

REVISION CHANGE-CAUSE

2 kubectl apply --filename=demo.yaml --record=true

5 kubectl apply --filename=demo.yaml --record=true

6 kubectl apply --filename=demo.yaml --record=true

[root@k-master-01 ~]# kubectl set image deploy/nginx-deployment nginx=nginx:1.15

deployment.apps/nginx-deployment image updated

[root@k-master-01 ~]# kubectl rollout history deployment nginx-deployment

deployment.apps/nginx-deployment

REVISION CHANGE-CAUSE

2 kubectl apply --filename=demo.yaml --record=true

5 kubectl apply --filename=demo.yaml --record=true

6 kubectl apply --filename=demo.yaml --record=true

[root@k-master-01 ~]# kubectl set resources deployment nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi

deployment.apps/nginx-deployment resource requirements updated

部署在暂停之前的初始状态将继续其功能，但是只要暂停部署，对部署的新更新将不会有任何效果。最后，恢复部署并观察新的ReplicaSet以及所有新更新。

[root@k-master-01 ~]# kubectl get pods -o wide -w

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-deployment-5549bd4b69-6h55t 0/1 ContainerCreating 0 2s <none> k-node-02 <none> <none>

nginx-deployment-5549bd4b69-dhjh5 1/1 Running 0 8s 10.240.1.182 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 1/1 Running 0 22m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-85ff79dd56-snfrf 1/1 Running 0 22m 10.240.1.180 k-node-01 <none> <none>

nginx-deployment-5549bd4b69-6h55t 1/1 Running 0 5s 10.240.2.196 k-node-02 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 1/1 Terminating 0 22m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-5549bd4b69-6qkc5 0/1 Pending 0 0s <none> <none> <none> <none>

nginx-deployment-5549bd4b69-6qkc5 0/1 Pending 0 0s <none> k-node-01 <none> <none>

nginx-deployment-5549bd4b69-6qkc5 0/1 ContainerCreating 0 0s <none> k-node-01 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 0/1 Terminating 0 22m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 0/1 Terminating 0 22m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-5549bd4b69-6qkc5 1/1 Running 0 6s 10.240.1.183 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-snfrf 1/1 Terminating 0 23m 10.240.1.180 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-snfrf 0/1 Terminating 0 23m 10.240.1.180 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-snfrf 0/1 Terminating 0 23m 10.240.1.180 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 0/1 Terminating 0 23m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-85ff79dd56-6k6hp 0/1 Terminating 0 23m 10.240.2.195 k-node-02 <none> <none>

nginx-deployment-85ff79dd56-snfrf 0/1 Terminating 0 23m 10.240.1.180 k-node-01 <none> <none>

nginx-deployment-85ff79dd56-snfrf 0/1 Terminating 0 23m 10.240.1.180 k-node-01 <none> <none>

最后观察RC

[root@k-master-01 ~]# kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-5489dd9d9 0 0 0 27m

nginx-deployment-5549bd4b69 3 3 3 2m54s

### ConfigMap

ConfigMap是用来存储配置文件的kubernetes资源对象，所有的配置内容都存储在etcd中。

* 挂载方式
  + 通过ENV环境变量的方式
  + 通过文件的形式挂载到POD中

#### 8.1 创建ConfigMap

创建ConfigMap的方式有4种：

##### 8.1.1 --from-literal

[root@k-master ~]# kubectl create configmap mysql --from-literal=db.host=localhost --from-literal=db.port=3306

configmap/mysql created

[root@k-master ~]# kubectl get configmaps

NAME DATA AGE

mysql 2 10s

[root@k-master ~]# kubectl describe configmaps mysql

Name: mysql

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

db.host:

----

localhost

db.port:

----

3306

Events: <none>

##### 8.1.2 通过指定文件创建，即将一个配置文件创建为一个ConfigMap --from-file=<文件>

[root@k-master ~]# vim nginx.conf

[root@k-master ~]# kubectl create configmap nginx --from-file=nginx.conf

configmap/nginx created

[root@k-master ~]# kubectl get configmaps

NAME DATA AGE

mysql 2 2m55s

nginx 1 8s

[root@k-master ~]# kubectl describe configmaps nginx

Name: nginx

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

nginx.conf:

----

#daemon off;

user www;

worker\_processes auto;

pid /var/run/nginx.pid;

worker\_rlimit\_nofile 51200;

events {

use epoll;

worker\_connections 51200;

multi\_accept on;

}

http {

}

Events: <none>

[root@k-master ~]#

* 通过指定目录创建，即将一个目录下的所有配置文件创建为一个ConfigMap，--from-file=<目录>

[root@k-master ~]# kubectl create configmap conf --from-file=conf/

configmap/conf created

[root@k-master ~]# kubectl get configmaps

NAME DATA AGE

conf 2 9s

mysql 2 5m16s

nginx 1 2m29s

[root@k-master ~]# kubectl describe configmaps config

Error from server (NotFound): configmaps "config" not found

[root@k-master ~]# kubectl describe configmaps conf

Name: conf

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

mysql.conf:

----

[mysqld]

nginx.conf:

----

#daemon off;

user www;

worker\_processes auto;

pid /var/run/nginx.pid;

worker\_rlimit\_nofile 51200;

events {

use epoll;

worker\_connections 51200;

multi\_accept on;

}

http {

include /application/nginx-1.16.1/conf/mime.types;

default\_type application/octet-stream;

}

Events: <none>

[root@k-master ~]#

* 事先写好标准的configmap的yaml文件，然后kubectl create -f 创建

[root@k-master ~]# vim config.yaml

kind: ConfigMap  
apiVersion: v1  
metadata:  
 name: configmap-yaml-test  
 namespace: default  
  
data:  
 db.host: "127.0.0.1"  
 db.port: "3306"  
 db.password: "123456"

[root@k-master ~]# kubectl apply -f config.yaml

configmap/yaml created

[root@k-master ~]# kubectl get configmaps

NAME DATA AGE

conf 2 23m

mysql 2 28m

nginx 1 25m

yaml 2 7s

[root@k-master ~]# kubectl describe configmaps yaml

Name: yaml

Namespace: default

Labels: app=yaml

Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"v1","data":{"nginx\_host":"127.0.0.1","nginx\_port":"80, 443"},"kind":"ConfigMap","metadata":{"annotations":{},"labels":{"app...

Data

====

nginx\_host:

----

127.0.0.1

nginx\_port:

----

80, 443

Events: <none>

#### 8.2 使用ConfigMap

##### 8.2.1 通过环境变量的方式，直接传递给pod

apiVersion: apps/v1

kind: Deployment

metadata:

name: test-config

spec:

selector:

matchLabels:

app: test-config

template:

metadata:

labels:

app: test-config

spec:

containers:

- name: test-config

image: busybox

resources:

limits:

memory: "128Mi"

cpu: "500m"

command:

- "/bin/sh"

- "-c"

- "env"

env:

- name: yaml

valueFrom:

configMapKeyRef:

name: NGINX-HOST

key: nginx\_host

##### 8.2.2 作为volume的方式挂载到pod内

kind: Deployment

apiVersion: apps/v1

metadata:

name: configmap-test-volumes

namespace: default

labels:

app: deployment-test

spec:

replicas: 2

selector:

matchLabels:

app: deployment-test-pod

template:

metadata:

name: deployment-pod-test

labels:

app: deployment-test-pod

spec:

containers:

- name: deployment-pod-nginx

image: nginx

imagePullPolicy: IfNotPresent

ports:

- containerPort: 80

name: http

volumeMounts:

- mountPath: "/etc/nginx/nginx.conf"

name: configmap

readOnly: true

subPath: nginx.conf

volumes:

- name: configmap

configMap:

name: nginx-baidu

items:

- key: nginx.conf

path: nginx.conf



注：

1. 删除configmap后原pod不受影响；然后再删除pod后，重启的pod的events会报找不到cofigmap的volume，除非设置为可为空。
2. pod起来后再通过kubectl edit configmap …修改configmap，pod内部的配置更新有延时。
3. 在容器内部修改挂进去的配置文件后，过一会内容会再次被刷新为原始configmap内容

#### 8.3 ConfigMap的热更新

更新 ConfigMap 后：

1. 使用该 ConfigMap 挂载的 Env 不会同步更新
2. 使用该 ConfigMap 挂载的 Volume 中的数据需要一段时间（实测大概10秒）才能同步更新

ENV 是在容器启动的时候注入的，启动之后 kubernetes 就不会再改变环境变量的值，且同一个 namespace 中的 pod 的环境变量是不断累加的。confgimap更新后，如果是以文件夹方式挂载的，会自动将挂载的Volume更新。如果是以文件形式挂载的，则不会自动更新。但是对多数情况的应用来说，配置文件更新后，最简单的办法就是重启Pod（杀掉再重新拉起）。如果是以文件夹形式挂载的，可以通过在容器内重启应用的方式实现配置文件更新生效。即便是重启容器内的应用，也要注意configmap的更新和容器内挂载文件的更新不是同步的，可能会有延时，因此一定要确保容器内的配置也已经更新为最新版本后再重新加载应用。

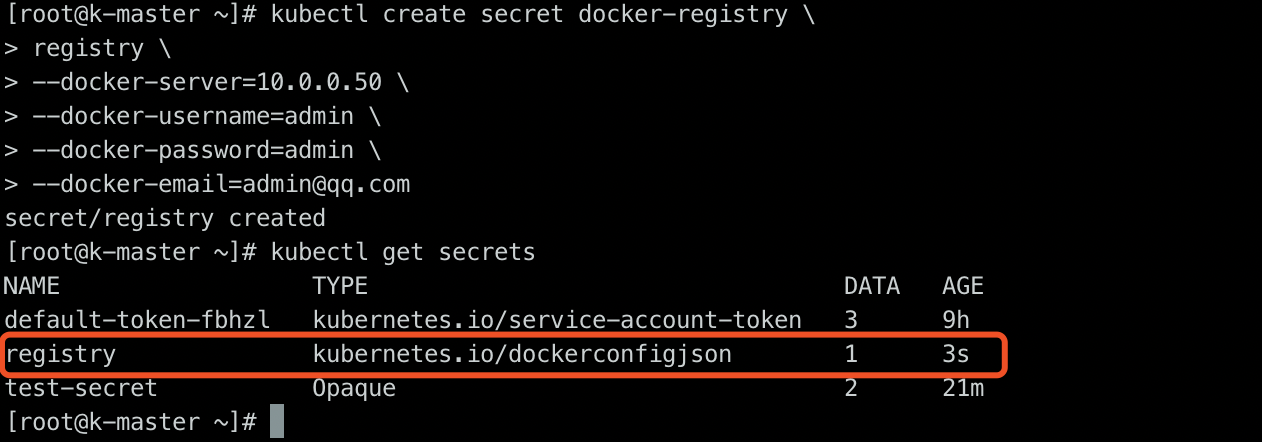
### Secret

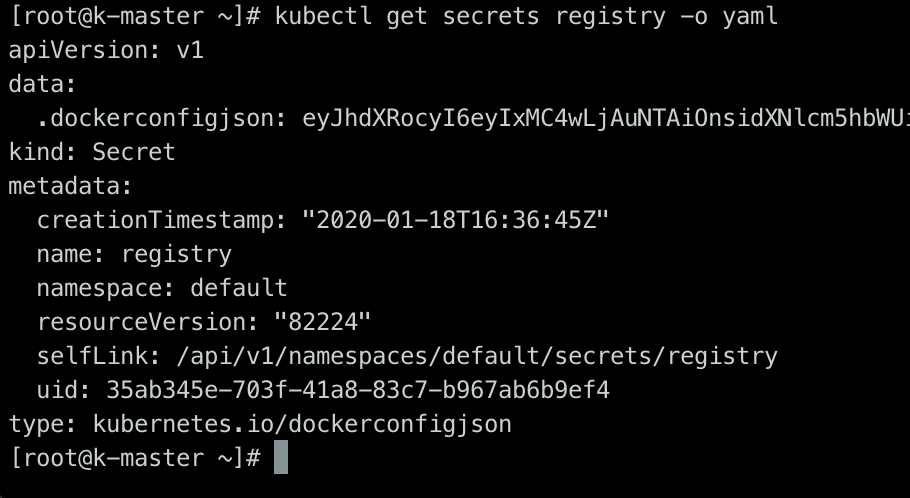
Secret与ConfigMap类似，但是用来存储敏感信息。它解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

#### 9.1. Secret类型和使用

Secret有三种类型。

9.1.1. docker-registry[kubernetes.io/dockerconfigjson]：创建一个给 Docker registry 使用的 secret





9.1.2. generic[Opaque]：从本地 file, directory 或者 literal value 创建一个 secret

**创建**

kind: Secret

apiVersion: v1

metadata:

name: test-secret

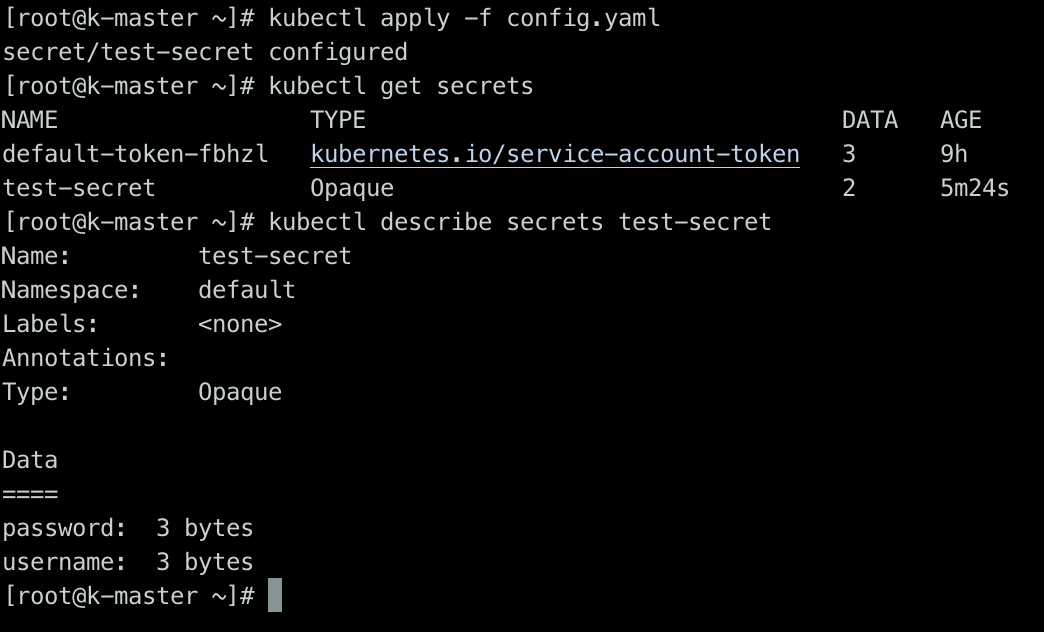
namespace: default

data:

username: root

password: root

type: Opaque



**使用**

kind: Deployment

apiVersion: apps/v1

metadata:

name: nginx

namespace: default

labels:

name: nginx

spec:

selector:

matchLabels:

name: nginx

template:

metadata:

labels:

name: nginx

spec:

containers:

- image: nginx

name: nginx

volumeMounts:

- name: secrets

mountPath: "/etc/secrets"

readOnly: true

ports:

- containerPort: 80

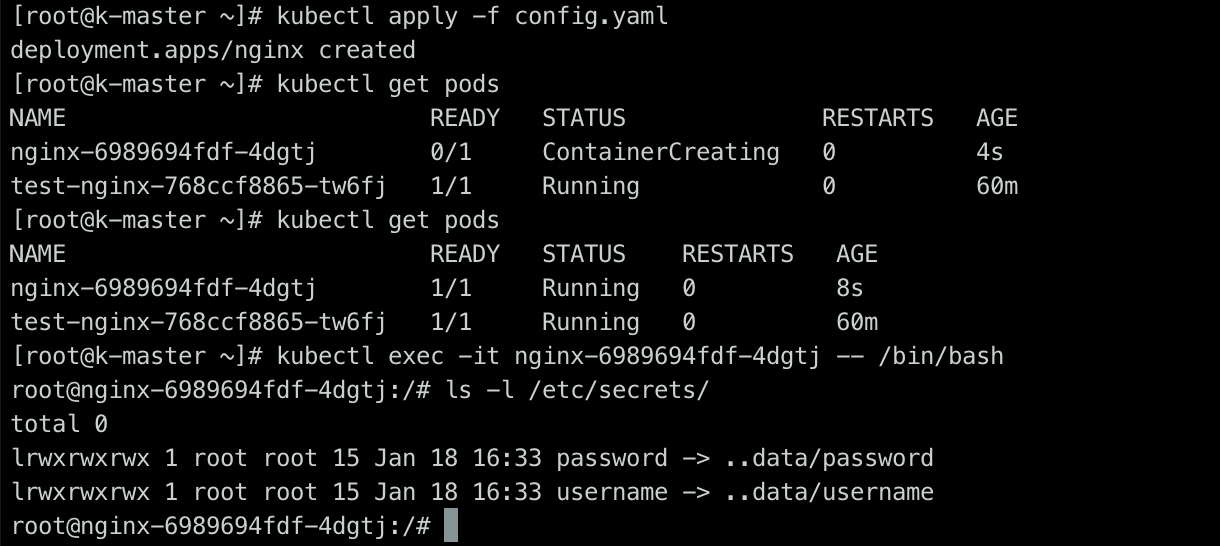
name: http

volumes:

- name: secrets

secret:

secretName: test-secret



9.1.3. tls[kubernetes.io/service-account-token]：创建一个 TLS secret。用于被serviceaccount引用。serviceaccout创建时Kubernetes会默认创建对应的secret。Pod如果使用了serviceaccount，对应的secret会自动挂载到Pod的/run/secrets/kubernetes.io/serviceaccount目录中。

[root@k-master-01 tls]# openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=alvin.com"

Generating a RSA private key

........................................+++++

...................................................................................+++++

writing new private key to 'tls.key'

-----

[root@k-master-01 tls]# kubectl create secret tls tls-secret --key tls.key --cert tls.crt

secret/tls-secret created

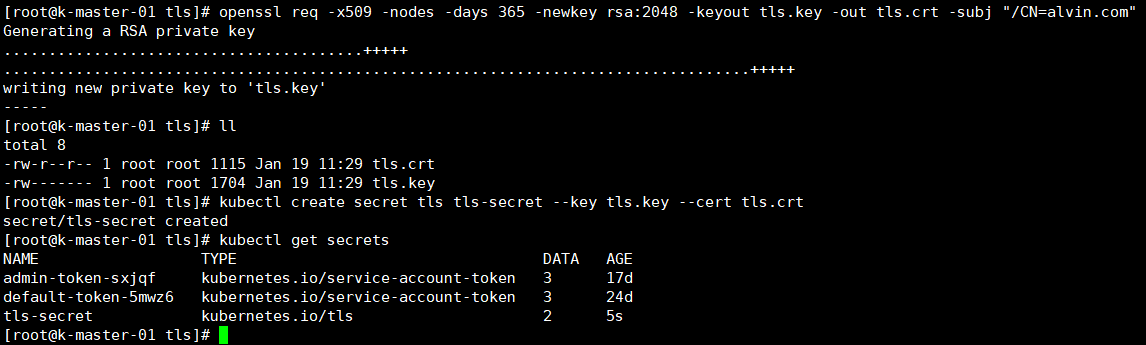
[root@k-master-01 tls]# kubectl get secrets

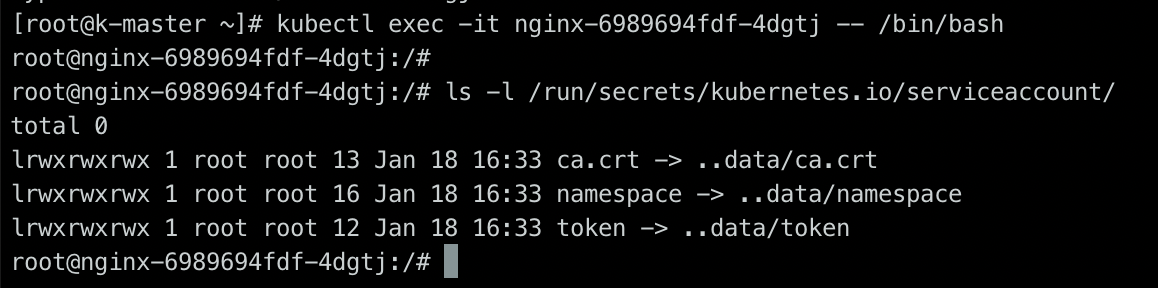
NAME TYPE DATA AGE

admin-token-sxjqf kubernetes.io/service-account-token 3 17d

default-token-5mwz6 kubernetes.io/service-account-token 3 24d

tls-secret kubernetes.io/tls 2 5s





#### 9.2. Secret与ConfigMap对比

在Kubernetes中，有集群级资源：pv, namespace。命名空间级的资源：pod, controller。

相同点：

1. key/value的形式
2. 属于某个特定的namespace
3. 可以导出到环境变量
4. 可以通过目录/文件形式挂载(支持挂载所有key和部分key)

不同点：

1. Secret可以被ServerAccount关联(使用)
2. Secret可以存储register的鉴权信息，用在ImagePullSecret参数中，用于拉取私有仓库的镜像
3. Secret支持Base64加密
4. Secret分为kubernetes.io/Service Account，kubernetes.io/dockerconfigjson，Opaque三种类型,Configmap不区分类型
5. Secret文件存储在tmpfs文件系统中，Pod删除后Secret文件也会对应的删除。

### DaemonSet

DaemonSet 确保全部（或者某些）节点上运行一个 Pod 的副本。当有节点加入集群时， 也会为他们新增一个 Pod 。当有节点从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod。

* 1. 创建 DaemonSet

kind: DaemonSet

apiVersion: apps/v1

metadata:

name: test-daemonset

namespace: default

labels:

app: filebeat

spec:

selector:

matchLabels:

app: filebeat

template:

metadata:

labels:

app: filebeat

spec:

containers:

- name: filebeat

image: docker.elastic.co/beats/filebeat:7.5.1

imagePullPolicy: Always

env:

- name: ELASTICSEARCH\_HOST

value: 106.13.81.75

- name: ELASTICSEARCH\_PORT

value: "9200"

ports:

- containerPort: 9200

name: filebeat-port

resources:

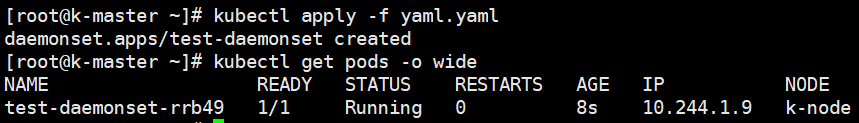
limits:

memory: 200Mi

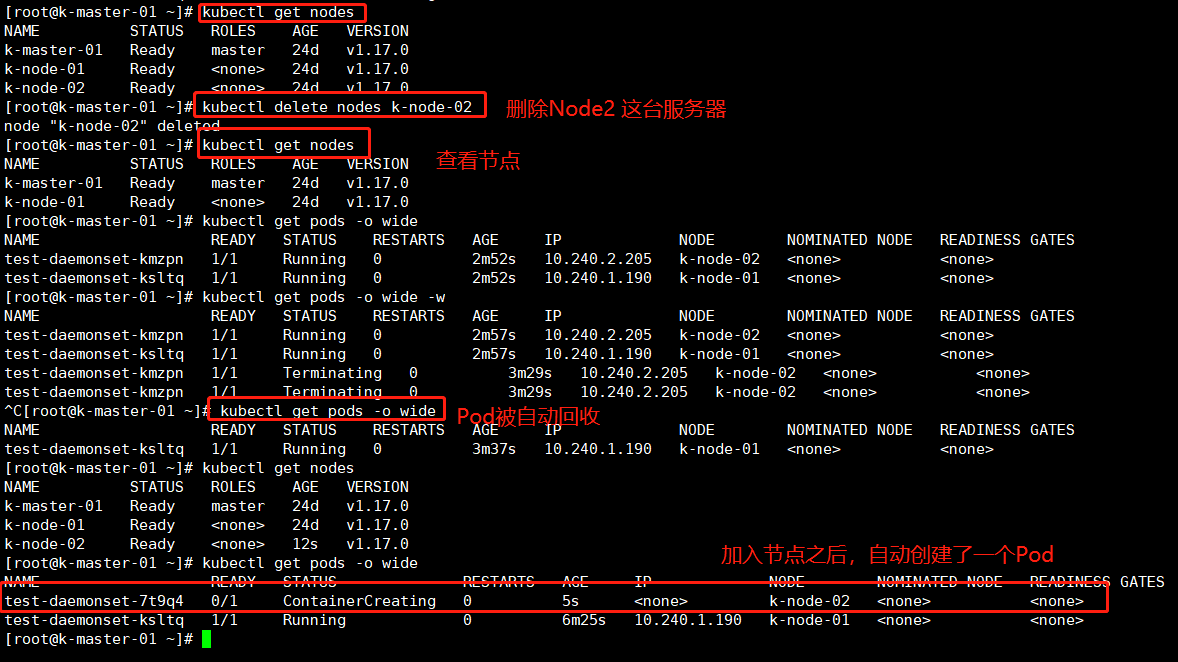
requests:

cpu: 100m

memory: 100Mi



我们可以清晰的看到分别在Node1和Node2上分别创建了一个Pod。那么我们现在删除一个Node查看结果：



### StatefulSet

我们知道，使用Kubernetes部署无状态的应用很方便，例如NGINX，Tomcat等，但是有时候我们需要部署像MySQL，Redis，ElasticSearch等这些有主从状态的应用的时候，如果我们单纯的去使用Deployment或DeamonSet的话，就不是那么的方便了。于是Kubernetes推出了StatefulSet控制器来完成这个需求。

#### 11.1 StatefulSet特点

StatefulSet适用于具有以下特点的应用

1. 具有固定的网络标记（主机名）
2. 具有持久化存储
3. 需要按顺序部署和扩展
4. 需要按顺序终止及删除
5. 需要按顺序滚动更新

#### 11.2创建StatefulSet

与Deployment一样，StatefulSet也是使用容器的Spec来创建Pod，与之不同StatefulSet创建的Pods在生命周期中会保持持久的标记（例如Pod Name）。

kind: Service

apiVersion: v1

metadata:

name: myapp-svc

namespace: default

labels:

app: myapp-svc

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: myapp-pod

---

kind: StatefulSet

apiVersion: apps/v1

metadata:

name: myapp

spec:

replicas: 2

selector:

matchLabels:

app: myapp-pod

serviceName: myapp-svc

template:

metadata:

labels:

app: myapp-pod

spec:

containers:

- name: myapp

image: nginx

ports:

- containerPort: 80

name: web

volumeMounts:

- mountPath: /usr/local/nginx/html

name: myappdata

volumeClaimTemplates:

- metadata:

name: myappdata

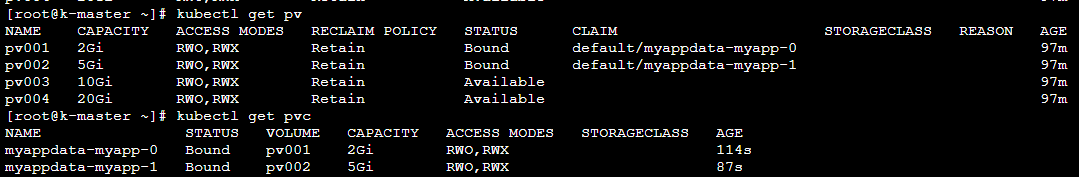
spec:

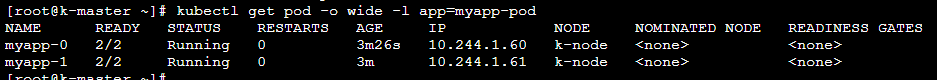
accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 2Gi



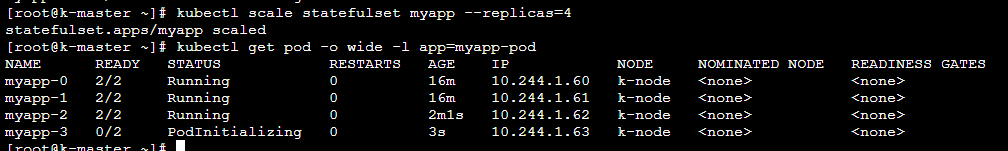


#### 11.3扩容/缩容

##### 11.3.1、scale

kubectl scale statefulset myapp --replicas=3

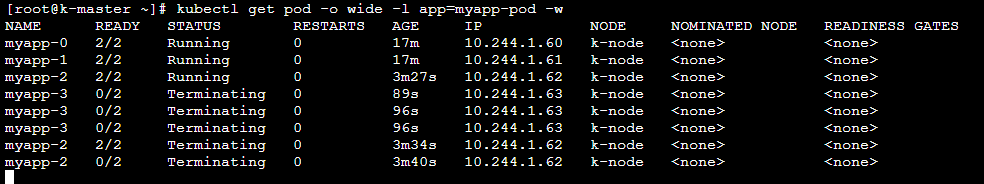
查看



##### 11.3.2、patch

kubectl patch statefulsets.apps myapp -p '{"spec":{"replicas":2}}'





##### 11.3.3、更新策略

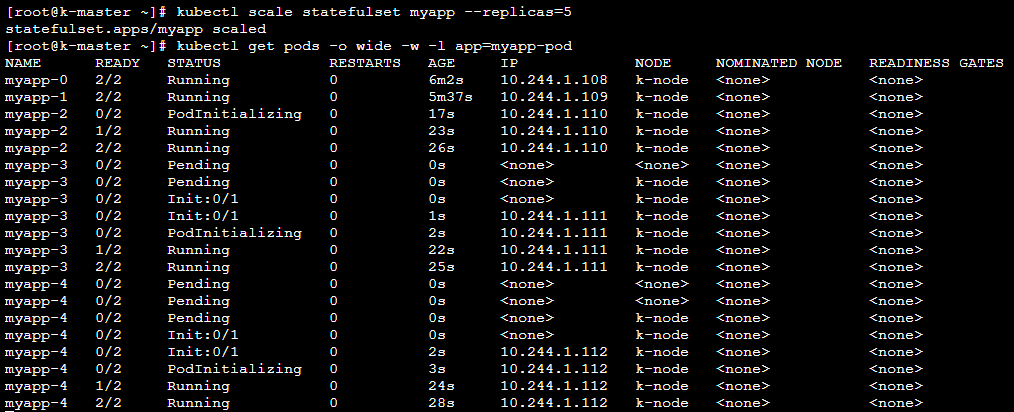
kubectl explain sts.spec.updateStrategy.rollingUpdate.partition

这种更新策略的含义是, 若当前statefulSet的副本数为5个，则Pod名为pod-0~pod-4，那么此时定义partition=4, 就意味着我要更新大于等于4的Pod，而只有pod-4的ID 4 是大于等于4的，所以只有pod-4会被更新，其它不会，这就是金丝雀更新。若后期发现pod-4更新后，工作一切正常，那么就可以调整partition=0，这样只要大于等于0的pod ID都将被更新。

首先我们将容器副本更新成5个。

kubectl scale statefulset myapp --replicas=5

kubectl get pods -o wide -w -l app=myapp-pod



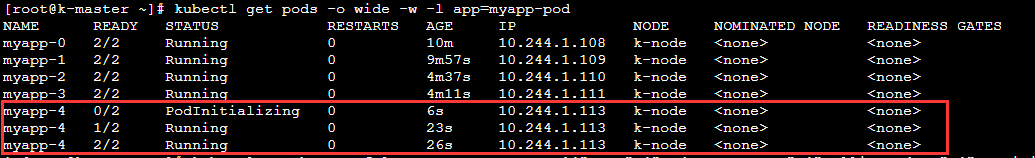
将更新策略设置成4

kubectl patch statefulsets.apps myapp -p '{"spec":{"updateStrategy":{"rollingUpdate":{"partition":4}}}}'



设置镜像为nginx:1.9.1

kubectl set image statefulset/myapp myapp=nginx:1.9.1



设置更新策略为0

kubectl patch statefulsets.apps myapp -p '{"spec":{"updateStrategy":{"rollingUpdate":{"partition":0}}}}'



### 12. Job/CronJob

Job负责处理任务，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。而CronJob则就是在Job上加上了时间调度。

#### 12.1 Job

首先用Yaml来创建一个Job：

kind: Job

apiVersion: batch/v1

metadata:

name: test-job

spec:

template:

metadata:

name: test-job

labels:

app: job

spec:

restartPolicy: Never

containers:

- name: busybox

imagePullPolicy: Always

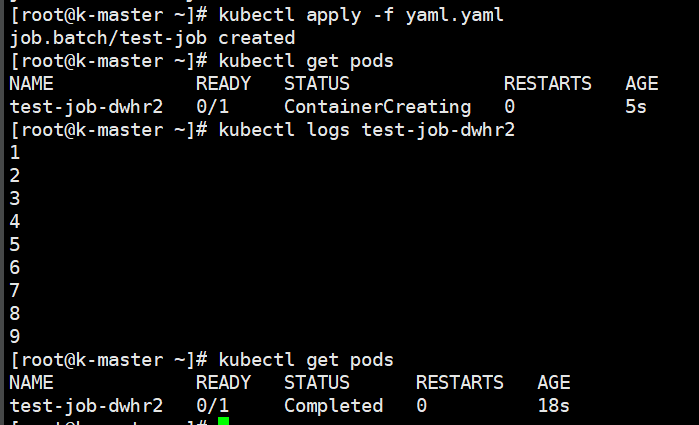
image: busybox

command:

- "/bin/sh"

- "-c"

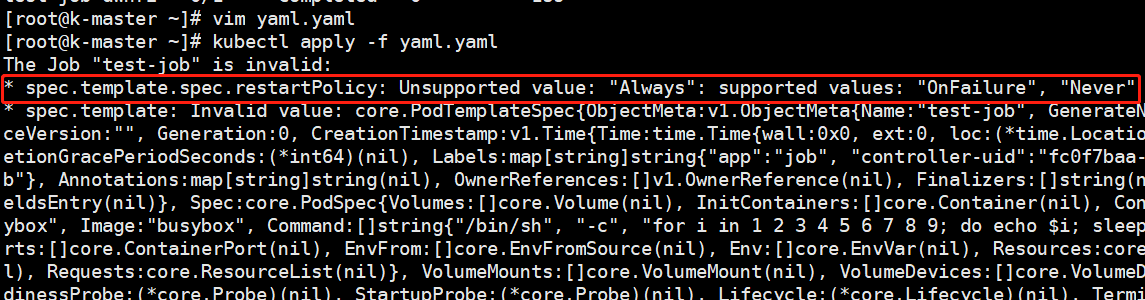
- "for i in 1 2 3 4 5 6 7 8 9; do echo $i; sleep 1; done"



注：Job的RestartPolicy仅支持Never和OnFailure两种，不支持Always。Job就相当于来执行一个批处理任务，执行完就结束了，如果使用Always的话就相当于陷入了死循环。

在Kubernetes中，pod的重启策略有：

* Always：当POD运行停止之后，总是重启
* Never：当POD运行停止之后，总是不重启
* OnFailure：当POD正常结束之后，不在重启，而在非正常情况下结束，则重启！



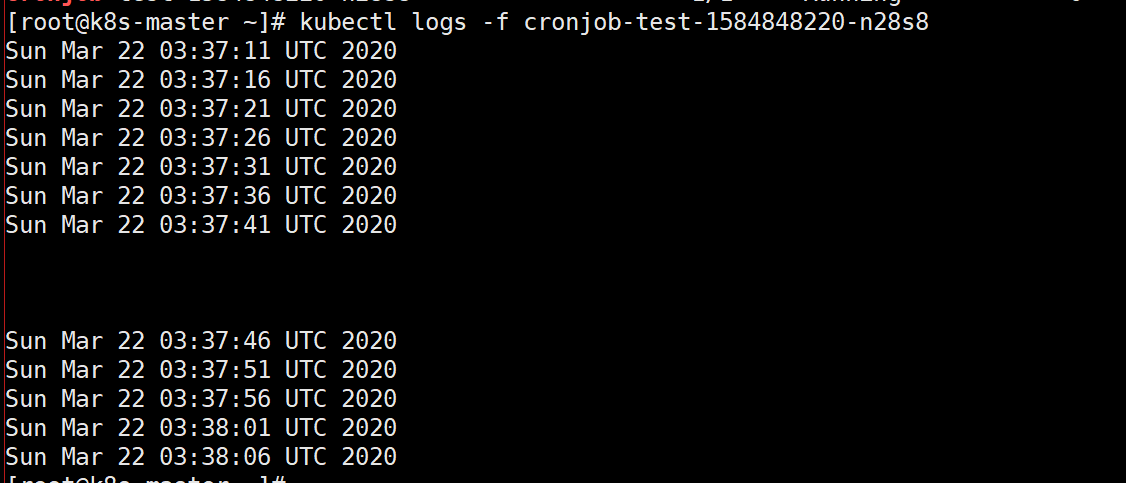
#### 12.2 CronJob

##### 12.2.1 创建CronJob

CronJob其实就是在Job的基础上加上了时间调度。这个实际上和我们Linux中的crontab就非常类似了。

kind: CronJob  
apiVersion: batch/v1beta1  
metadata:  
 name: cronjob-test  
  
spec:  
 schedule: "\* \* \* \* \*"  
 jobTemplate:  
 metadata:  
 name: cronjob-test  
  
 spec:  
 template:  
 metadata:  
 name: cronjob-test-pod  
  
 spec:  
 containers:  
 - name: busybox  
 image: busybox  
 command:  
 - "/bin/sh"  
 - "-c"  
 - "for i in 1 2 3 4 5 6 7 8 9 10 11 12; do echo $(date); sleep 5; done"  
 restartPolicy: OnFailure





##### 12.2.2 并发性规则

.spec.concurrencyPolicy也是可选的。它声明了 CronJob 创建的任务执行时发生重叠如何处理。spec 仅能声明下列规则中的一种：

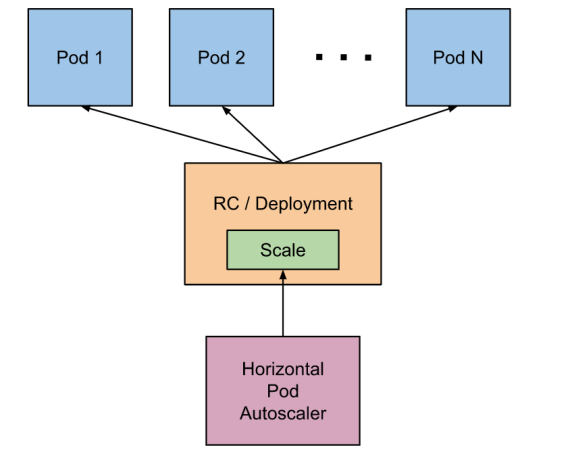
1. Allow (默认)：CronJob 允许并发任务执行。
2. Forbid： CronJob 不允许并发任务执行；如果新任务的执行时间到了而老任务没有执行完，CronJob 会忽略新任务的执行。
3. Replace：如果新任务的执行时间到了而老任务没有执行完，CronJob 会用新任务替换当前正在运行的任务。

请注意，并发性规则仅适用于相同 CronJob 创建的任务。如果有多个 CronJob，它们相应的任务总是允许并发执行的。

### 13.Horizontal Pod Autoscaling（HPA）

Horizontal Pod Autoscaling可以根据CPU使用率或应用自定义metrics自动扩展Pod数量（支持replication controller、deployment和replica set）。基于CPU利用率自动伸缩 replication controller、deployment和 replica set 中的 pod 数量，（除了 CPU 利用率）也可以 基于其他应程序提供的度量指标[custom metrics](https://git.k8s.io/community/contributors/design-proposals/instrumentation/custom-metrics-api.md" \t "https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale/_blank)。 pod 自动缩放不适用于无法缩放的对象，比如 DaemonSets。

#### 13.1. HPA工作机制



Pod 水平自动伸缩的实现是一个控制循环，由 controller manager 的 --horizontal-pod-autoscaler-sync-period 参数 指定周期（默认值为15秒）。

每个周期内，controller manager 根据每个 HorizontalPodAutoscaler 定义中指定的指标查询资源利用率。 controller manager 可以从 resource metrics API（每个pod 资源指标）和 custom metrics API（其他指标）获取指标。

### Service

将运行在一组 Pods 上的应用程序公开为网络服务的抽象方法。使用Kubernetes，您无需修改应用程序即可使用不熟悉的服务发现机制。 Kubernetes为Pods提供自己的IP地址和一组Pod的单个DNS名称，并且可以在它们之间进行负载均衡。

Kubernetes [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/" \t "https://kubernetes.io/zh/docs/concepts/services-networking/service/_blank) 是有生命周期的。他们可以被创建，而且销毁不会再启动。 如果您使用 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/" \t "https://kubernetes.io/zh/docs/concepts/services-networking/service/_blank) 来运行您的应用程序，则它可以动态创建和销毁 Pod。每个 Pod 都有自己的 IP 地址，但是在 Deployment 中，在同一时刻运行的 Pod 集合可能与稍后运行该应用程序的 Pod 集合不同。例如有一个Pod宕机，Deployment重新启动了一个Pod，这个Pod的名称和IP地址跟原来的IP完全不同。那么这导致了一个问题： 如果一个Deployment正在对外提供功能，那么客户端如何找出并跟踪要连接的IP地址，以便前端使用呢？

面对这样的问题，Kubernetes推出了Service，它通过筛选label来关联一组Pod，并对其提供负载均衡服务。

#### 14.1 定义Service

kind: Service

apiVersion: v1

metadata:

name: test-service

namespace: default

labels:

app: test-service

spec:

type: ClusterIP

selector:

app: test-service

ports:

- port: 80

targetPort: 80



#### 14.2 VIP 和 Service 代理

在 Kubernetes 集群中，每个 Node 运行一个 kube-proxy 进程。kube-proxy 负责为 Service 实现了一种 VIP（虚拟 IP）的形式。

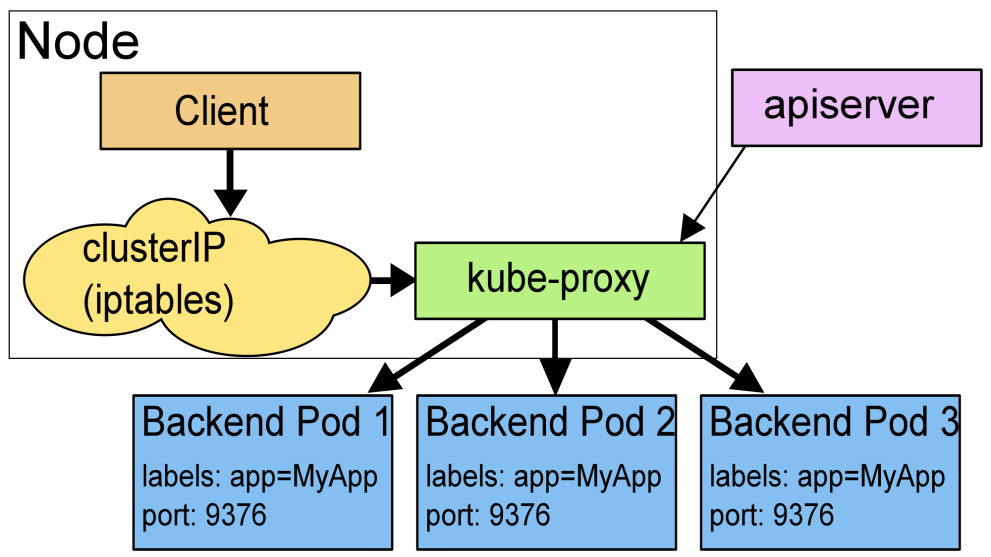
##### 14.2.1 userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会在本地 Node 上打开一个端口（随机选择）。 任何连接到“代理端口”的请求，都会被代理到 Service 的backend Pods 中的某个上面（如 Endpoints 所报告的一样）。 使用哪个 backend Pod，是 kube-proxy 基于 SessionAffinity 来确定的。

最后，它安装 iptables 规则，捕获到达该 Service 的 clusterIP（是虚拟 IP）和 Port 的请求，并重定向到代理端口，代理端口再代理请求到 backend Pod。

默认情况下，用户空间模式下的kube-proxy通过循环算法选择后端。

默认的策略是，通过 round-robin 算法来选择 backend Pod。



##### 14.2.2 iptables 代理模式

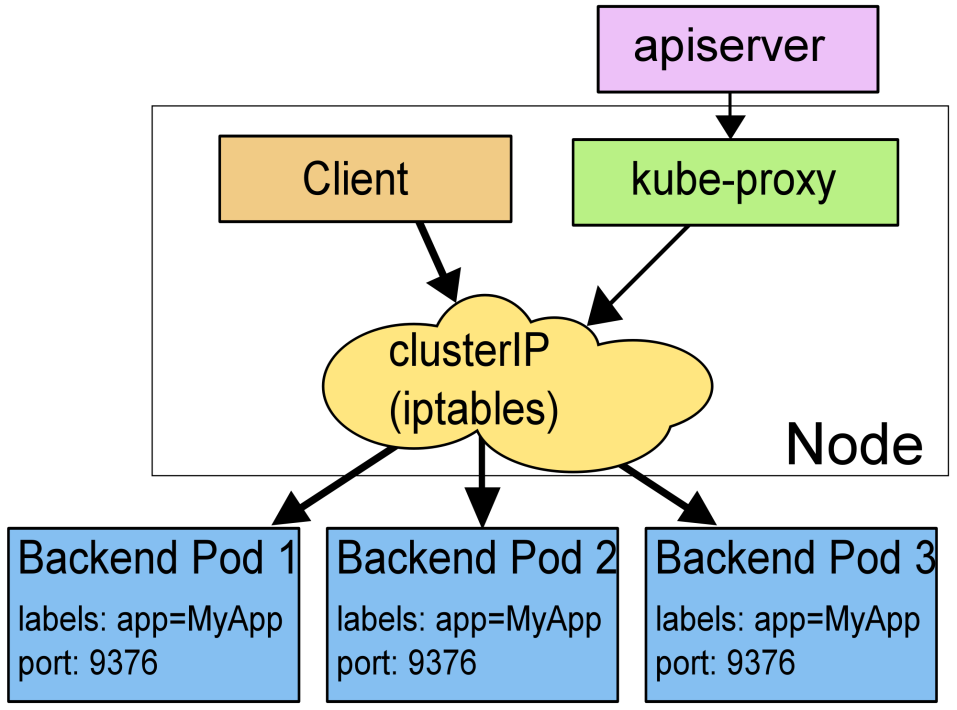
这种模式，kube-proxy 会监视 Kubernetes 控制节点对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会安装 iptables 规则，从而捕获到达该 Service 的 clusterIP 和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某个上面。 对于每个 Endpoints 对象，它也会安装 iptables 规则，这个规则会选择一个 backend 组合。

默认的策略是，kube-proxy 在 iptables 模式下随机选择一个 backend。

使用 iptables 处理流量具有较低的系统开销，因为流量由 Linux netfilter 处理，而无需在用户空间和内核空间之间切换。 这种方法也可能更可靠。

如果 kube-proxy 在 iptable s模式下运行，并且所选的第一个 Pod 没有响应，则连接失败。 这与用户空间模式不同：在这种情况下，kube-proxy 将检测到与第一个 Pod 的连接已失败，并会自动使用其他后端 Pod 重试。

您可以使用 Pod readiness 探测器验证后端 Pod 可以正常工作，以便 iptables 模式下的 kube-proxy 仅看到测试正常的后端。这样做意味着您避免将流量通过 kube-proxy 发送到已知已失败的Pod。



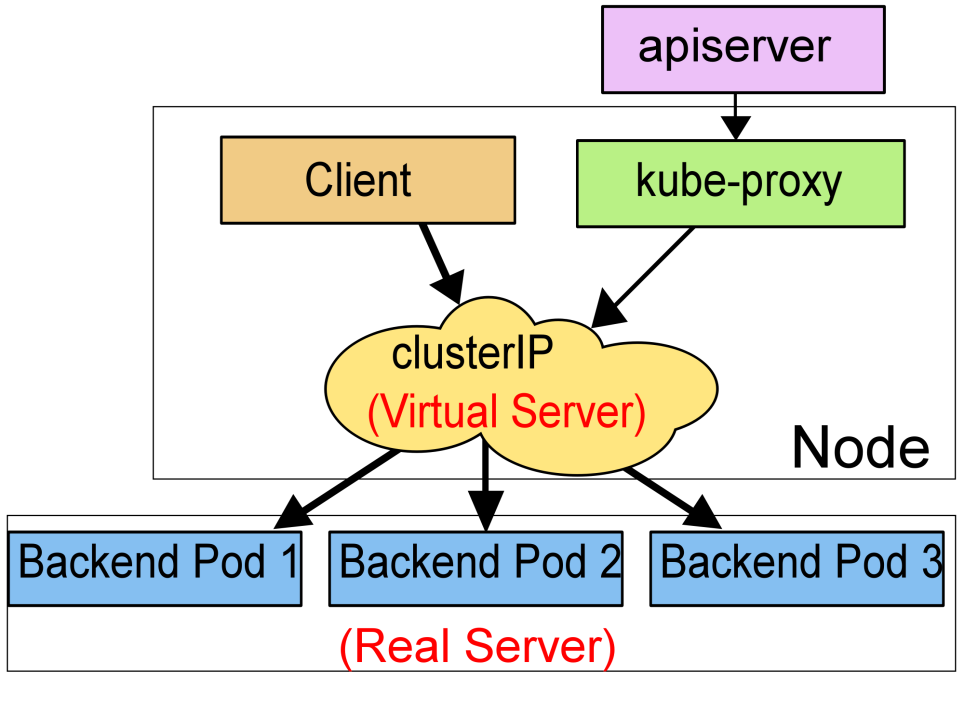
##### 14.2.3 IPVS 代理模式

在 ipvs 模式下，kube-proxy监视Kubernetes服务和端点，调用 netlink 接口相应地创建 IPVS 规则， 并定期将 IPVS 规则与 Kubernetes 服务和端点同步。 该控制循环可确保　IPVS　状态与所需状态匹配。 访问服务时，IPVS　将流量定向到后端Pod之一。

IPVS代理模式基于类似于 iptables 模式的 netfilter 挂钩函数，但是使用哈希表作为基础数据结构，并且在内核空间中工作。 这意味着，与 iptables 模式下的 kube-proxy 相比，IPVS 模式下的 kube-proxy 重定向通信的延迟要短，并且在同步代理规则时具有更好的性能。与其他代理模式相比，IPVS 模式还支持更高的网络流量吞吐量。

IPVS提供了更多选项来平衡后端Pod的流量。 这些是：

* rr：轮询调度
* lc：最小连接数
* dh：目标哈希
* sh：源哈希
* sed：最短期望延迟
* nq：不排队调度



#### 14.3 暴露内部服务

##### 14.3.1 Nodeport

创建这种方式的Service，内部可以通过ClusterIP进行访问，外部用户可以通过NodeIP:NodePort的方式单独访问每个Node上的实例。这种方式有很多问题，直接访问节点的地址和端口需要在客户端记录很多信息，Pod发生迁移后这些信息没办法动态更新，节点的防火墙及节点所在网络区域的防火墙策略配置会比较麻烦。

apiVersion: v1

kind: Service

metadata:

name: nginx

spec:

type: NodePort

ports:

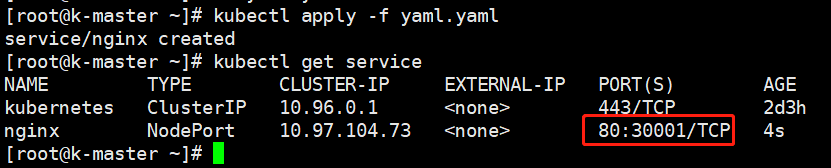
- port: 80

targetPort: 80

nodePort: 30001

selector:

app: nginx



##### 14.3.2 LoadBalancer

这种方式一般需要云供应商的支持。

##### 14.3.3 Ingress

具体请看Ingress小节。

#### 14.4 HeadLess Service

有时不需要或不想要service负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建 Headless Service。您可以使用headless Service 与其他服务发现机制进行接口，而不必与 Kubernetes 的实现捆绑在一起。典型应用就是Ingress。

##### 14.4.1. 正常Service案例

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

name: http

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

labels:

app: nginx

spec:

ports:

- port: 80

targetPort: 80

selector:

app: nginx

---

apiVersion: v1

kind: Service

metadata:

name: headless-service

spec:

selector:

app: nginx

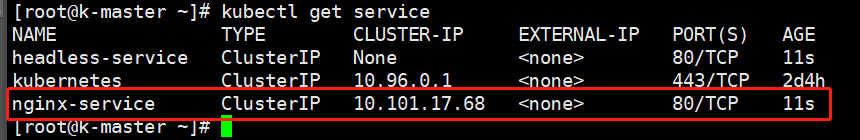
ports:

- protocol: TCP

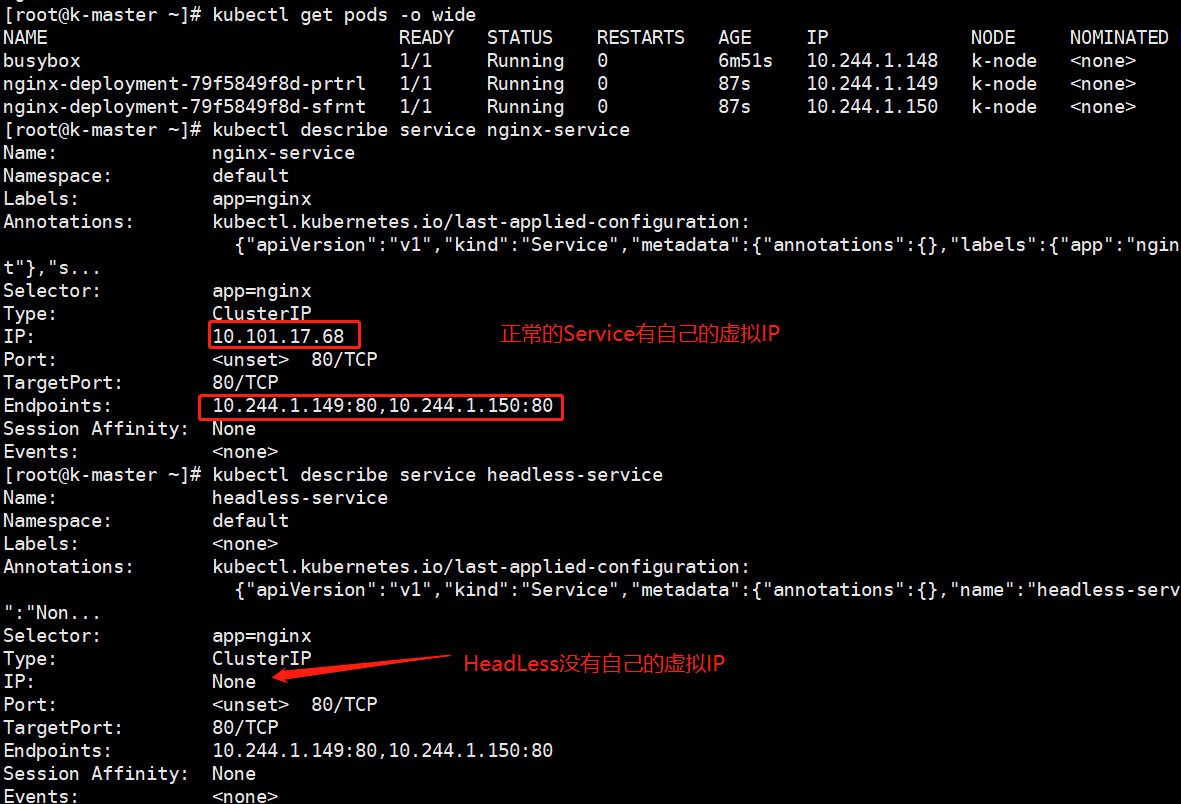
port: 80

targetPort: 80

clusterIP: None



查看一下Service的详情



测试服务，是由Service代理到Pod上。



##### 14.4.2 Headless Service案例

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

name: http

---

apiVersion: v1

kind: Service

metadata:

name: headless-service

spec:

selector:

app: nginx

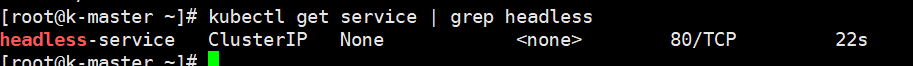
ports:

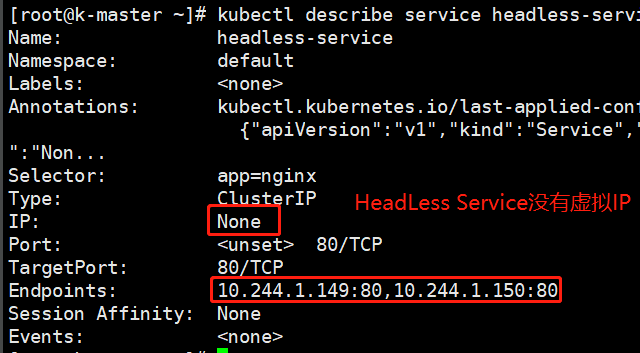
- protocol: TCP

port: 80

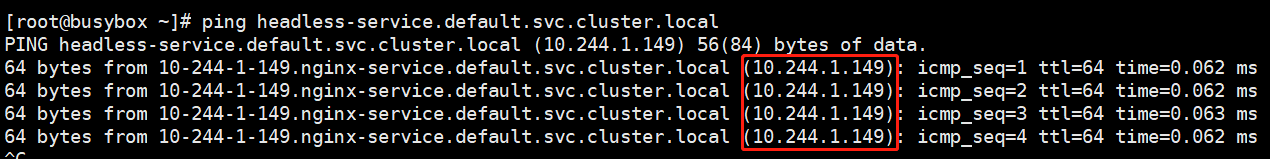
targetPort: 80

clusterIP: None





测试服务，可见HeadLess Service 直接访问的是Pod



### Ingress

在Kubernetes中，服务和Pod的IP地址仅可以在集群网络内部使用，对于集群外的应用是不可见的。为了使外部的应用能够访问集群内的服务。Kubernetes提供了NodePort, LoadBalancer和IngressNginx。

#### Ingress的组成

* Ingress Controller：将新加入的Ingress配置转化成nginx配置，并使其生效。
* Ingress服务：将Nginx配置抽象成Ingress配置，方便使用。

#### Ingress工作原理

* ingress controller通过和kubernetes api交互，动态的去感知集群中ingress规则变化。
* 然后读取它，按照自定义的规则，去集群当中寻找对应Service管理的Pod。
* 再将生成的Nginx配置写到nginx-ingress-control的pod里，这个Ingress controller的pod里运行着一个Nginx服务，控制器会把生成的nginx配置写入/etc/nginx.conf文件中。
* 然后reload一下使配置生效。以此达到域名分配置和动态更新的问题。

#### Ingress的优点

##### 动态配置服务

如果按照传统方式, 当新增加一个服务时, 我们可能需要在流量入口加一个反向代理指向我们新的k8s服务. 而如果用了Ingress, 只需要配置好这个服务, 当服务启动时, 会自动注册到Ingress的中, 不需要e外的操作.

##### 减少不必要的端口暴露

　　配置过k8s的都清楚, 第一步是要关闭防火墙的, 主要原因是k8s的很多服务会以NodePort方式映射出去, 这样就相当于给宿主机打了很多孔, 既不安全也不优雅. 而Ingress可以避免这个问题。

#### 创建Ingress

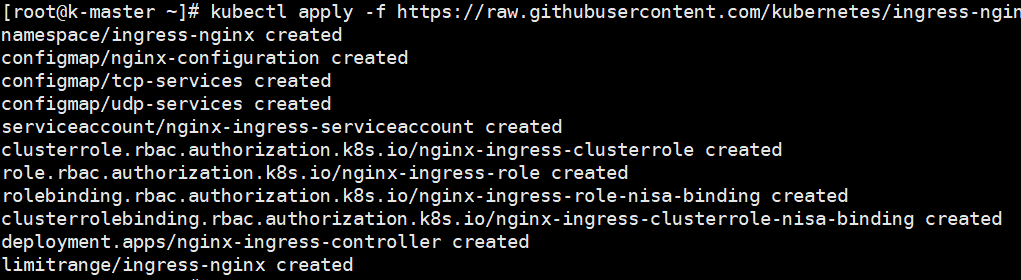
NginxIngress是托管在Github之上的：[https://github.com/kubernetes/ingress-nginx/](https://github.com/kubernetes/ingress-nginx/tree/nginx-0.20.0/deploy)，本章采用NodePort的方式来安装。

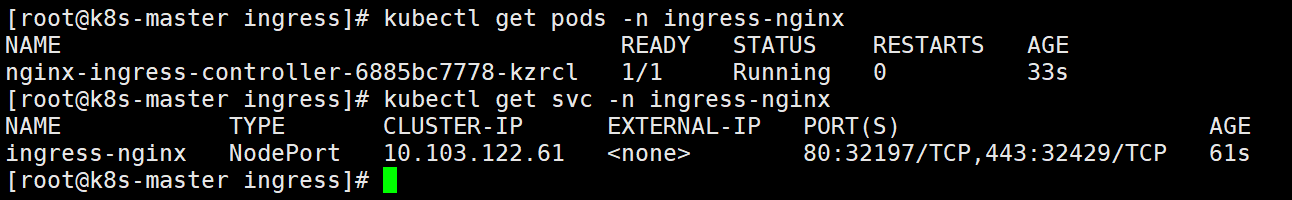
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.29.0/deploy/static/mandatory.yaml

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.29.0/deploy/static/provider/baremetal/service-nodeport.yaml

注：如果镜像拉取不下来可使用阿里云镜像海外构建的方式拉取，镜像连接：

registry.cn-hangzhou.aliyuncs.com/alvinos/nginx-ingress-controller:0.29.0



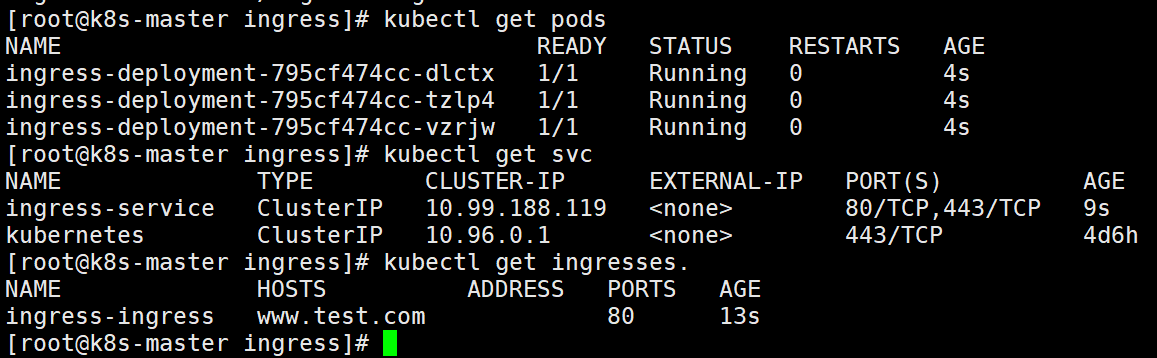


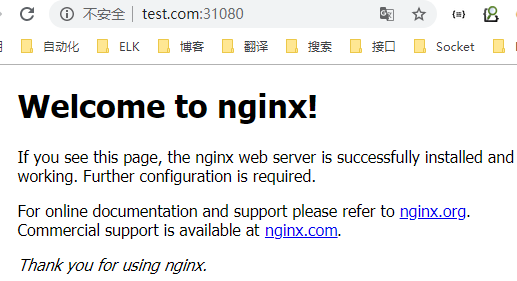


安装成功，之所以是404是因为还没有后端服务。

#### 使用Ingress-Nginx创建一个后端服务

kind: Deployment  
apiVersion: apps/v1  
metadata:  
 name: ingress-deployment  
 namespace: default  
 labels:  
 release: stable  
 environment: dev  
 tier: frontend  
 partition: customerA  
 track: daily  
 app: deployment  
  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 release: stable  
 environment: dev  
 tier: frontend  
 partition: customerA  
 track: daily  
 app: pod  
  
 template:  
 metadata:  
 labels:  
 release: stable  
 environment: dev  
 tier: frontend  
 partition: customerA  
 track: daily  
 app: pod  
 spec:  
 containers:  
 - name: ingress-pod  
 image: nginx  
 imagePullPolicy: IfNotPresent  
 ports:  
 - containerPort: 80  
 name: http  
  
 - containerPort: 443  
 name: https  
  
---  
kind: Service  
apiVersion: v1  
metadata:  
 name: ingress-service  
 namespace: default  
 labels:  
 release: stable  
 environment: dev  
 tier: frontend  
 partition: customerA  
 track: daily  
 app: svc  
  
spec:  
 type: ClusterIP  
 selector:  
 release: stable  
 environment: dev  
 tier: frontend  
 partition: customerA  
 track: daily  
 app: pod  
  
 ports:  
 - port: 80  
 targetPort: 80  
 name: http  
  
 - port: 443  
 targetPort: 443  
 name: https  
  
---  
kind: Ingress  
apiVersion: extensions/v1beta1  
metadata:  
 name: ingress-ingress  
 namespace: default  
 annotations:  
 kubernetes.io/ingress.class: "nginx"  
spec:  
 rules:  
 - host: www.test.com  
 http:  
 paths:  
 - path: /  
 backend:  
 serviceName: ingress-service  
 servicePort: 80





访问成功。

#### 基于TLS的Ingress-Nginx

如果要使用基于TLS的Ingress-Nginx的话，需要事先准备好Nginx的证书，也可以通过OpenSSL创建，创建方式如下

# 创建证书

[root@k8s-master ingress]# openssl genrsa -out tls.key 2048

Generating RSA private key, 2048 bit long modulus

.......................................+++

.............+++

e is 65537 (0x10001)

[root@k8s-master ingress]# openssl req -new -x509 -key tls.key -out tls.crt -subj /C=CN/ST=ShangHai/L=ShangHai/O=Ingress/CN=www.test.com

# 将证书保存到Secret中

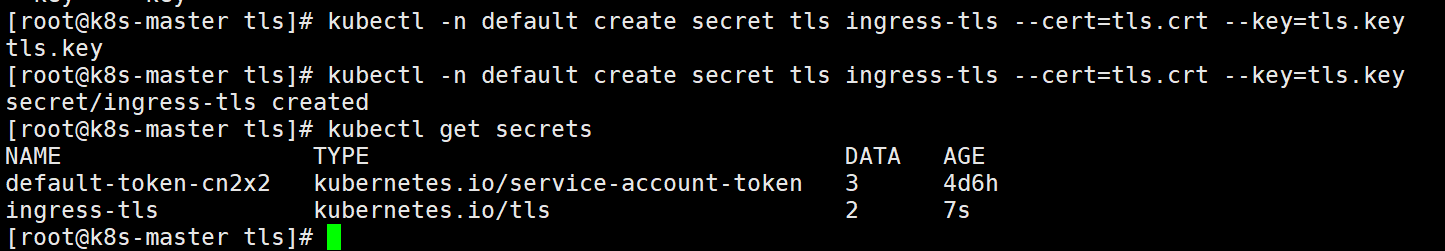
[root@k8s-master tls]# kubectl -n default create secret tls ingress-tls --cert=tls.crt --key=tls.key

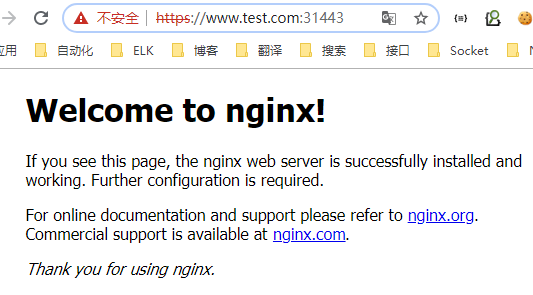
secret/ingress-tls created

kind: Ingress  
apiVersion: extensions/v1beta1  
metadata:  
 name: ingress-bak-test  
 namespace: default  
 labels:  
 app: ingress-test  
 annotations:  
 kubernetes.io/ingress.class: "nginx"  
spec:  
 tls:  
 - hosts:  
 - www.test.com  
  
 secretName: ingress-tls  
  
 rules:  
 - host: www.test.com  
 http:  
 paths:  
 - path: /  
 backend:  
 serviceName: ingress-service-bak-test  
 servicePort: 80  
---  
kind: Service  
apiVersion: v1  
metadata:  
 name: ingress-service-bak-test  
 namespace: default  
 labels:  
 app: ingress-service-test  
spec:  
 selector:  
 app: ingress-deployment-pod-bak-test  
 ports:  
 - port: 80  
 targetPort: 80  
 name: http  
  
 - port: 443  
 targetPort: 443  
 name: https  
  
---  
kind: Deployment  
apiVersion: apps/v1  
metadata:  
 name: ingress-deployment-bak-test  
 namespace: default  
 labels:  
 app: ingress-deployment-bak-test  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: ingress-deployment-pod-bak-test  
 template:  
 metadata:  
 labels:  
 app: ingress-deployment-pod-bak-test  
 spec:  
 containers:  
 - name: django  
 image: nginx  
 ports:  
 - name: http  
 containerPort: 80  
  
 - name: https  
 containerPort: 443

################# HTTP #################

kind: Deployment  
apiVersion: apps/v1  
metadata:  
 name: ingress-deployment-bak  
 namespace: default  
 labels:  
 app: ingress-deployment-bak  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: ingress-deployment-pod-bak  
 template:  
 metadata:  
 labels:  
 app: ingress-deployment-pod-bak  
 spec:  
 containers:  
 - name: django  
 image: alvinos/django:v1  
 ports:  
 - name: http  
 containerPort: 80  
---  
kind: Service  
apiVersion: v1  
metadata:  
 name: ingress-service-bak  
 namespace: default  
 labels:  
 app: ingress-service  
spec:  
 selector:  
 app: ingress-deployment-pod-bak  
 ports:  
 - port: 80  
 targetPort: 80  
 name: http  
  
---  
kind: Ingress  
apiVersion: extensions/v1beta1  
metadata:  
 name: ingress-bak  
 namespace: default  
 labels:  
 app: ingress  
 annotations:  
 kubernetes.io/ingress.class: "nginx"  
spec:  
 rules:  
 - host: www.bak.com  
 http:  
 paths:  
 - path: /  
 backend:  
 serviceName: ingress-service-bak  
 servicePort: 80





### Volume

我们知道，Pod是由容器组成的，而容器宕机或停止之后，数据就随之丢了，那么这也就意味着我们在做Kubernetes集群的时候就不得不考虑存储的问题，而存储卷就是为了Pod保存数据而生的。存储卷的类型有很多，我们常用到一般有四种：emptyDir，hostPath，NFS以及云存储等。

#### 16.2. emptyDir存储卷

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。

kind: Deployment

apiVersion: apps/v1

metadata:

name: test-volume-deployment

namespace: default

labels:

app: test-volume-deployment

spec:

selector:

matchLabels:

app: test-volume-pod

template:

metadata:

labels:

app: test-volume-pod

spec:

containers:

- name: nginx

image: busybox

imagePullPolicy: IfNotPresent

ports:

- containerPort: 80

name: http

- containerPort: 443

name: https

volumeMounts:

- mountPath: /data/

name: empty

command: ['/bin/sh','-c','while true;do echo $(date) >> /data/index.html;sleep 2;done']

- name: os

imagePullPolicy: IfNotPresent

image: busybox

volumeMounts:

- mountPath: /data/

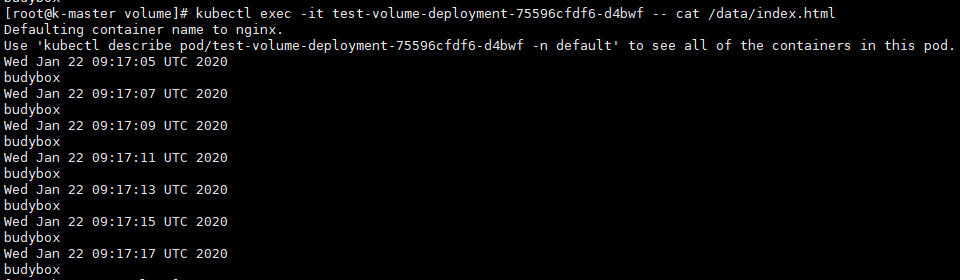
name: empty

command: ['/bin/sh','-c','while true;do echo 'budybox' >> /data/index.html;sleep 2;done']

volumes:

- name: empty

emptyDir: {}



#### 16.3. hostPath存储卷

hostPath类型则是映射node文件系统中的文件或者目录到pod里。在使用hostPath类型的存储卷时，也可以设置type字段，支持的类型有文件、目录、File、Socket、CharDevice和BlockDevice。

16.3.1. hostPath类型

| 取值 | 行为 |
| --- | --- |
|  | 空字符串（默认）用于向后兼容，这意味着在安装 hostPath 卷之前不会执行任何检查。 |
| DirectoryOrCreate | 如果在给定路径上什么都不存在，那么将根据需要创建空目录，权限设置为 0755，具有与 Kubelet 相同的组和所有权。 |
| Directory | 在给定路径上必须存在的目录。 |
| FileOrCreate | 如果在给定路径上什么都不存在，那么将在那里根据需要创建空文件，权限设置为 0644，具有与 Kubelet 相同的组和所有权。 |
| File | 在给定路径上必须存在的文件。 |
| Socket | 在给定路径上必须存在的 UNIX 套接字。 |
| CharDevice | 在给定路径上必须存在的字符设备。 |
| BlockDevice | 在给定路径上必须存在的块设备。 |

16.3.2 演示,创建存储卷，使用DirectoryOrCreate类型，node节点不存在会自动创建。

apiVersion: v1

kind: Pod

metadata:

name: vol-hostpath

namespace: default

spec:

volumes:

- name: html

hostPath:

path: /data/pod/volume1/

type: DirectoryOrCreate

containers:

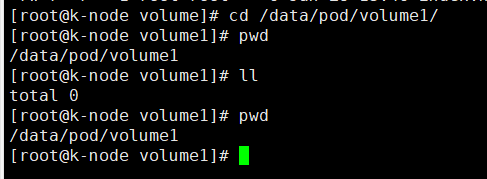
- name: myapp

image: nginx

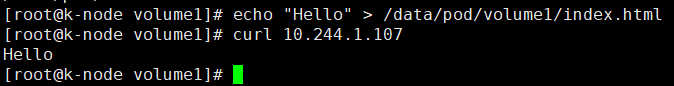
volumeMounts:

- name: html

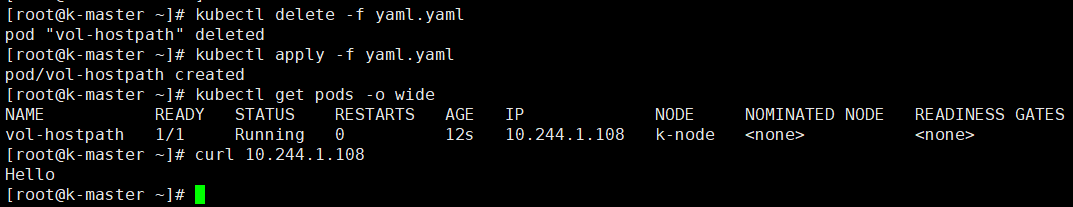
mountPath: /usr/share/nginx/html/



16.3.3 查询验证



16.3.4 就算pod被删除再重建，只要node还在，存储卷就还在。



#### 16.4. NFS存储卷

nfs使得我们可以挂载已经存在的共享到我们的Pod中，和emptyDir不同的是，当Pod被删除时，emptyDir也会被删除。但是nfs不会被删除，仅仅是解除挂在状态而已，这就意味着NFS能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间相互传递，并且nfs可以同时被多个pod挂在并进行读写。

16.4.1 安装nfs

yum install nfs-utils.x86\_64 -y

Vim /etc/exports

/root/volume/data1 172.26.192.0/20(rw,no\_root\_squash)

# 测试

mount -t nfs 172.19.0.3:/root/volume/data1 /data/pod/volume1

[root@k8s-node02 ~]# mount

......

172.19.0.3:/root/volume/data1 on /data/pod/volume1 type nfs4 (rw,relatime,vers=4.1,rsize=131072,wsize=131072,namlen=255,hard,proto=tcp,port=0,timeo=600,retrans=2,sec=sys,clientaddr=192.168.56.13,local\_lock=none,addr=192.168.56.14)

[root@stor01 ~]# showmount -e

Export list for 172.19.0.3:

/data/volumes 172.19.0.3/20

16.4.2创建

kind: Deployment

apiVersion: apps/v1

metadata:

name: nfs

namespace: default

spec:

replicas: 2

selector:

matchLabels:

app: nfs

template:

metadata:

labels:

app: nfs

spec:

containers:

- name: nginx

imagePullPolicy: IfNotPresent

image: nginx

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html/

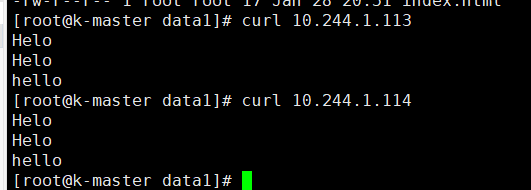
volumes:

- name: html

nfs:

path: /root/volume/data1

server: 172.19.0.3



### PV/PVC

PersistentVolume（PV）是集群中已由管理员配置的一段网络存储。 集群中的资源就像一个节点是一个集群资源。 PV是诸如卷之类的卷插件，但是具有独立于使用PV的任何单个pod的生命周期。 该API对象捕获存储的实现细节，即NFS，iSCSI或云提供商特定的存储系统。

PersistentVolumeClaim（PVC）是用户存储的请求。PVC的使用逻辑：在pod中定义一个存储卷（该存储卷类型为PVC），定义的时候直接指定大小，pvc必须与对应的pv建立关系，pvc会根据定义去pv申请，而pv是由存储空间创建出来的。pv和pvc是kubernetes抽象出来的一种存储资源。

#### 17.1生命周期

|  |  |
| --- | --- |
| **Volume 的生命周期 5 个阶段** | |
| Provisioning | 即 PV 的创建，可以直接创建 PV（静态方式），也可以使用 StorageClass 动态创建。 |
| Binding | 将 PV 分配给 PVC。 |
| Using | Pod通过PVC使用该Volume。 |
| Releasing | Pod释放Volume并删除PVC。 |
| Reclaiming | 回收PV，可以保留PV 以便下次使用，也可以直接从云存储中删除。 |
| Deleting | 删除 PV 并从云存储中删除后段存储。 |
| **Volume的4种状态** | |
| Available | 可用。 |
| Bound | 已经分配给 PVC。 |
| Released | PVC 解绑但还未执行回收策略。 |
| Failed | 发生错误。 |

#### 17.2 PV的访问模式

|  |  |
| --- | --- |
| ReadWriteOnce（RWO） | 可读可写，但只支持被单个节点挂载。 |
| ReadOnlyMany（ROX） | 只读，可以被多个节点挂载。 |
| ReadWriteMany（RWX） | 多路可读可写。这种存储可以以读写的方式被多个节点共享。不是每一种存储都支持这三种方式，像共享方式，目前支持的还比较少，比较常用的是 NFS。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。 |

#### 17.3 PV的回收策略

|  |  |
| --- | --- |
| Retain | 不清理, 保留 Volume（需要手动清理） |
| Recycle | 删除数据，即 rm -rf /thevolume/\*（只有 NFS 和 HostPath 支持） |
| Delete | 删除存储资源，比如删除 AWS EBS 卷（只有 AWS EBS, GCE PD, Azure Disk 和 Cinder 支持） |

#### 17.4 创建PV

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv001

labels:

app: pv001

spec:

nfs:

path: /root/datav1

server: 172.31.16.9

accessModes:

- "ReadWriteMany"

- "ReadWriteOnce"

capacity:

storage: 2Gi

---

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv002

labels:

app: pv002

spec:

nfs:

path: /root/datav2

server: 172.31.16.9

accessModes:

- "ReadWriteMany"

- "ReadWriteOnce"

capacity:

storage: 5Gi

---

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv003

labels:

app: pv003

spec:

nfs:

path: /root/datav3

server: 172.31.16.9

accessModes:

- "ReadWriteMany"

- "ReadWriteOnce"

capacity:

storage: 10Gi

---

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv004

labels:

app: pv004

spec:

nfs:

path: /root/datav4

server: 172.31.16.9

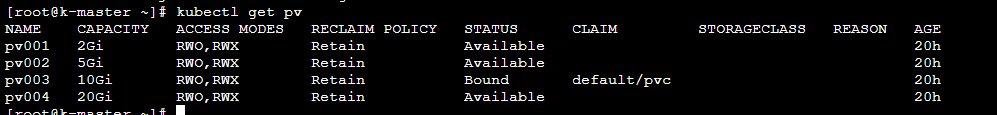
accessModes:

- "ReadWriteMany"

- "ReadWriteOnce"

capacity:

storage: 20Gi



#### 17.5 创建PVC

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc

namespace: default

spec:

accessModes:

- "ReadWriteMany"

resources:

requests:

storage: "6Gi"

---

kind: Deployment

apiVersion: apps/v1

metadata:

name: nfs

namespace: default

spec:

replicas: 2

selector:

matchLabels:

app: nfs

template:

metadata:

labels:

app: nfs

spec:

containers:

- name: nginx

imagePullPolicy: IfNotPresent

image: nginx

volumeMounts:

- name: html

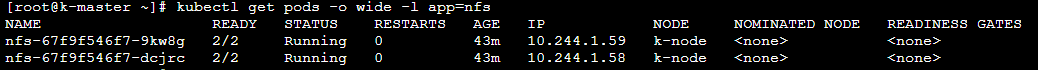
mountPath: /usr/share/nginx/html/

volumes:

- name: html

persistentVolumeClaim:

claimName: pvc



### StorageClass

StorageClass为我们提供了一种类似存储“类”的方法，集群管理员能够在一个集群中定义各种存储卷供应，用户不需要了解存储的细节和复杂性，就能够选择符合自己要求的存储。说白了，就是由kubernetes自动创建符合条件的存储。

#### 定义存储类

每一个存储类都包含provisioner、parameters和reclaimPolicy这三个参数域，当一个属于某个类的PersistentVolume需要被动态提供时，将会使用上述的参数域。

存储类对象的名称非常重要，用户通过名称类请求特定的存储类。管理员创建存储类对象时，会设置类的名称和其它的参数，存储类的对象一旦被创建，将不能被更新。管理员能够为PVC指定一个默认的存储类。

* + 1. 创建StorageClass

使用GlusterFS做自动存储的话需要安装Heketi服务，请参考《第五章：基于GlusterFS的Kubernetes》

# provisioner：表示存储分配器，需要根据后端存储的不同而变更；

# reclaimPolicy: 默认即”Delete”，删除pvc后，相应的pv及后端的volume，brick(lvm)等一起删除；设置为”Retain”时则保留数据，需要手工处理

# resturl：heketi API服务提供的url；

# restauthenabled：可选参数，默认值为”false”，heketi服务开启认证时必须设置为”true”；

# restuser：可选参数，开启认证时设置相应用户名；

# secretNamespace：可选参数，开启认证时可以设置为使用持久化存储的namespace；

# secretName：可选参数，开启认证时，需要将heketi服务的认证密码保存在secret资源中；

# clusterid：可选参数，指定集群id，也可以是1个clusterid列表，格式为”id1,id2”；

# volumetype：可选参数，设置卷类型及其参数，如果未分配卷类型，则有分配器决定卷类型；如”volumetype: replicate:3”表示3副本的replicate卷，”volumetype: disperse:4:2”表示disperse卷，其中‘4’是数据，’2’是冗余校验，”volumetype: none”表示distribute卷#

$ vim storageclass.yaml

kind: StorageClass  
apiVersion: storage.k8s.io/v1  
metadata:  
 name: storageclass-gluster  
provisioner: kubernetes.io/glusterfs  
parameters:  
 resturl: "http://k8s-master:18080"  
 clusterid: "76e679836682dbd0d023847af02f5dc5"  
 restauthenabled: "true"  
 restuser: "admin"  
 restuserkey: "admin"  
 gidMin: "40000"  
 gidMax: "50000"  
 volumetype: "replicate:2"

# 创建StorageClass

$ kubectl apply -f storageclass.yaml

# 创建PVC,看是否可以动态创建PV

kind: PersistentVolumeClaim  
apiVersion: v1  
metadata:  
 name: glusterfs-test-pvc  
 namespace: default  
 annotations:  
 volume.beta.kubernetes.io/storage-class: "glusterfs"  
spec:  
 storageClassName: glusterfs  
 accessModes:  
 - ReadWriteMany  
 resources:  
 requests:  
 storage: 2Gi

$ kubectl get pv

# 这里就可以看到动态创建的PV了。

