第 2 章

Docker 及 Kubernetes 基础

上一章主要讲解了 Kubernetes 各种版本的安装方式，相信读者已经有了一套高可用 Kubernetes 集群了，并且也对 Kubernetes 的架构和各种组件有了一些认识。本章主要讲解 Docker 和 Kubernetes 的一些基本概念和简单操作，基于上一章搭建的集群来学习本章内容会让自己印象更加深刻。

2.1 Docker 基础

2.1.1 Docker 介绍

Docker 是一个开源的软件项目，在 Linux 操作系统上，Docker 提供了一个额外的软件抽象层及操作系统层虚拟化的自动管理机制。Docker 运行名为“Container（容器）”的软件包，容器之间彼此隔离，并捆绑了自己的应用程序、工具、库和配置文件。所有容器都由单个操作系统内核运行，因此比虚拟机更轻量级。

Docker 利用 Linux 资源分离机制，例如 cgroups 及 Linux Namespace 来创建相互独立的容器（Container），可以在单个 Linux 实体下运行，避免了启动一个虚拟机造成的额外负担。Linux 核心对 Namespace（命名空间）的支持完全隔离了不同 Namespace 下的应用程序的“视野”（即作用范围），包括进程树、网络、用户 ID 与挂载的文件系统等，而核心 cgroups 则提供了资源隔离，包括 CPU、存储器、Block I/O 与网络。

2.1.2 Docker 基本命令

本节介绍 Docker 的一些常用命令，这些命令有助于读者排查和解决集群中的问题。

查看 Docker 版本。包括 Docker 版本号、API 版本号、Git Commit、Go 版本号等。

```
[root@K8S-master01 ~]# docker version
```

```
Client:
 Version:     17.09.1-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:       Thu Dec  7 22:23:40 2017
 OS/Arch:     linux/amd64

Server:
 Version:     17.09.1-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:       Thu Dec  7 22:25:03 2017
 OS/Arch:     linux/amd64
 Experimental: false
```

显示 Docker 信息：

```
Containers: 22
 Running: 21
 Paused: 0
 Stopped: 1
Images: 18
Server Version: 17.09.1-ce
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 06b9cb35161009dcb7123345749fef02f7cea8e0
runc version: 3f2f8b84a77f73d38244dd690525642a72156c64
init version: 949e6fa
Security Options:
 seccomp
  Profile: default
Kernel Version: 4.18.9-1.el7.elrepo.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 3.848GiB
Name: K8S-master01
ID: HM66:LH4K:PNES:GFJX:TKNX:TLOH:WONE:KLHT:YRB3:3KAR:3WZJ:HYOX
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
```

```
    Debug Mode (server): false
    Username: dotbalo
    Registry: https://index.docker.io/v1/
    Experimental: false
    Insecure Registries:
     127.0.0.0/8
    Live Restore Enabled: false
```

查询镜像。OFFICIAL 为 OK 的是官方镜像，默认搜索的是 hub.docker.com。

```
    [root@K8S-master01 ~]# docker search nginx
    NAME                                              DESCRIPTION
STARS             OFFICIAL          AUTOMATED
    nginx                                             Official build of Nginx.
10749             [OK]
    jwilder/nginx-proxy                               Automated Nginx reverse
proxy for docker c...   1507                                [OK]
    richarvey/nginx-php-fpm                           Container running Nginx +
PHP-FPM capable ...   675                                 [OK]
    jrcs/letsencrypt-nginx-proxy-companion            LetsEncrypt container
to use with nginx as...   469                                 [OK]
    webdevops/php-nginx                               Nginx with PHP-FPM
120                                 [OK]
    kitematic/hello-world-nginx                       A light-weight nginx
container that demons...   119
    zabbix/zabbix-web-nginx-mysql                     Zabbix frontend based on
Nginx web-server ...   86                                  [OK]
    bitnami/nginx                                     Bitnami nginx Docker Image
60                                  [OK]
    linuxserver/nginx                                 An Nginx container, brought
to you by Linu...   51
    1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5
ubuntu-16-nginx-php-phpmyadmin-mysql-5        48
[OK]
    tobi312/rpi-nginx                                 NGINX on Raspberry Pi / armhf
23                                  [OK]
    nginx/nginx-ingress                               NGINX Ingress Controller
for Kubernetes        15
    blacklabelops/nginx                               Dockerized Nginx Reverse
Proxy Server.         12                                  [OK]
    wodby/drupal-nginx                                Nginx for Drupal container
image              11                                 [OK]
    centos/nginx-18-centos7                           Platform for running nginx
1.8 or building...   10
    nginxdemos/hello                                  NGINX webserver that serves
a simple page ...   9                                   [OK]
    webdevops/nginx                                   Nginx container
8                                   [OK]
    centos/nginx-112-centos7                          Platform for running nginx
1.12 or buildin...   6
    1science/nginx                                    Nginx Docker images that
include Consul Te...   4                                   [OK]
    travix/nginx                                      NGinx reverse proxy
2                                   [OK]
    mailu/nginx                                       Mailu nginx frontend
```

```
2                                  [OK]
    pebbletech/nginx-proxy                                nginx-proxy sets up a
container running ng...   2                                    [OK]
    toccoag/openshift-nginx                               Nginx reverse proxy for
Nice running on sa...   1                                   [OK]
    ansibleplaybookbundle/nginx-apb                        An APB to deploy NGINX
0                                  [OK]
    wodby/nginx                                         Generic nginx
```

拉取/下载镜像。默认是 hub.docker.com（docker.io）上面的镜像，如果拉取公司内部的镜像或者其他仓库上的镜像，需要在镜像前面加上仓库的 URL，如：

```
    docker pull harbor.xxx.net/frontend:v1
```

拉取公网上的 Nginx 镜像：

```
    # 把公网上的镜像拉取到本地服务器，不指定版本号为 latest
    [root@K8S-master01 ~]# docker pull nginx
    Using default tag: latest
    latest: Pulling from library/nginx
    Digest:
sha256:b543f6d0983fbc25b9874e22f4fe257a567111da96fd1d8f1b44315f1236398c
    Status: Image is up to date for nginx:latest

    #拉取指定版本
    [root@K8S-master01 ~]# docker pull nginx:1.15
    1.15: Pulling from library/nginx
    Digest:
sha256:b543f6d0983fbc25b9874e22f4fe257a567111da96fd1d8f1b44315f1236398c
    Status: Downloaded newer image for nginx:1.15
```

推送镜像。把本地的镜像推送到公网仓库中，或者公司内部的仓库中。

默认登录和推送的是公网的镜像，如果需要推送到公司仓库或者其他仓库，只需要在镜像前面使用 tag 并加上 URL 即可：

```
    [root@K8S-master01 ~]# docker images | grep nginx-v2
    nginx-v2                                                              latest
3d9c6e44d3db       3 hours ago        109MB
    [root@K8S-master01 ~]# docker tag nginx-v2 dotbalo/nginx-v2:test
    [root@K8S-master01 ~]# docker images | grep nginx-v2
    dotbalo/nginx-v2                                                       test
3d9c6e44d3db       3 hours ago        109MB
    nginx-v2                                                              latest
3d9c6e44d3db       3 hours ago        109MB
    [root@K8S-master01 ~]# docker login
    Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
    Username (dotbalo): dotbalo
    Password:
    Login Succeeded
    [root@K8S-master01 ~]# docker push dotbalo/nginx-v2:test
    The push refers to a repository [docker.io/dotbalo/nginx-v2]
    2eaa7b5717a2: Mounted from dotbalo/nginx
    a674e06ede38: Mounted from dotbalo/nginx
    b7efe781401d: Mounted from dotbalo/nginx
```

```
    c9c2a3696080: Mounted from dotbalo/nginx
    7b4e562e58dc: Mounted from dotbalo/nginx
    test: digest:
sha256:5d749d2b10150426b510d2c3a05a99cf547c2ca1be382e1dbb2f90b68b6bea96 size:
1362
```

前台启动一个容器：

```
    [root@DockerTestServer ~]# docker run -ti nginx bash
    root@23bc7ccabb09:/#
```

后台启动：

```
    [root@DockerTestServer ~]# docker run -tid nginx bash
    1bcf5154d5c3a57d92a6796f526eac2cefd962aaca9cf4098689bfe830bb9e5e
```

端口映射。可以将本机的端口映射到容器的端口，比如将本机的 1111 端口映射到容器的 80 端口：

```
    [root@DockerTestServer ~]# docker run -ti -p 1111:80 nginx bash
    root@cd676d572188:/#
```

挂载卷。可以将本机的目录挂载到容器的指定目录，比如将 hosts 文件挂载到容器的 hosts：

```
    [root@DockerTestServer ~]# docker run -ti -p 1111:80 -v /etc/hosts:/etc/hosts
nginx bash
    root@cd676d572188:/#
```

查看当前正在运行的容器：

```
    [root@K8S-master01 K8S-ha-install]# docker ps
    CONTAINER ID       IMAGE
COMMAND                CREATED           STATUS            PORTS
NAMES
    862e82066496       94ec7e53edfc
"nginx -g 'daemon ..."   21 hours ago       Up 21 hours
K8S_nginx_nginx-deployment-57895845b8-vb7bs_default_d0d254f8-1fb3-11e9-a9f2-00
0c293ad492_1
    10bf838e18d0
registry.cn-hangzhou.aliyuncs.com/google_containers/pause-amd64:3.1   "/pause"
21 hours ago       Up 21 hours
K8S_POD_nginx-deployment-57895845b8-vb7bs_default_d0d254f8-1fb3-11e9-a9f2-000c
293ad492_1
```

查看所有容器，包括已经退出的：

```
    [root@K8S-master01 K8S-ha-install]# docker ps -a
```

查看正在运行的容器（即显示出容器的 ID）：

```
    [root@K8S-master01 K8S-ha-install]# docker ps -q
    ……
    0d1a98b3c402
    c1fd8ff1f7f2
    86b1c069024b
    ……
```

查看所有容器的 ID，包括已经退出的：

```
    [root@K8S-master01 K8S-ha-install]# docker ps -aq
    ……
    17019738d93d
    b3bb2a592dfb
    e0637b76afe3
    0b74e028d0ae
    65a1b5e1e501
    ……
```

进入到一个后台运行的容器（即之前用-d 命令参数来指定后台运行方式的容器）：

```
    [root@K8S-master01 K8S-ha-install]# docker ps  | tail -1
    86b1c069024b       nginx:latest
"nginx -g 'daemon ..."   4 days ago         Up 21 hours        80/tcp,
0.0.0.0:16443->16443/tcp   nginx-lb
    [root@K8S-master01 K8S-ha-install]# docker exec -ti 86b1c069024b bash
    root@nginx-lb:/#
```

拷贝文件。双向拷贝，可以将本机的文件拷贝到容器，反之亦然：

```
    [root@K8S-master01 K8S-ha-install]# docker cp README.md
92aceec0dcdd327a709bf0ec83:/tmp
    #exec 也可直接执行容器命令
    [root@K8S-master01 K8S-ha-install]# docker exec 92aceec0dcdd327a709bf0ec83 ls
/tmp/
    README.md
```

删除已经退出的容器：

```
    [root@K8S-master01 K8S-ha-install]# docker ps -a |grep Exited | tail -3
    600e5da5c196       3cab8e1b9802
"etcd --advertise-..."   4 days ago         Exited (137) 21 hours ago
K8S_etcd_etcd-K8S-master01_kube-system_c94bb8ceba1b924e6e3175228b168fe0_0
    5a1848d923a1
registry.cn-hangzhou.aliyuncs.com/google_containers/pause-amd64:3.1   "/pause"
4 days ago         Exited (0) 21 hours ago
K8S_POD_kube-scheduler-K8S-master01_kube-system_9c27268d8e3e5c14fa0160192a2c79
88_0
    280fc86494f1
registry.cn-hangzhou.aliyuncs.com/google_containers/pause-amd64:3.1   "/pause"
4 days ago         Exited (0) 21 hours ago
K8S_POD_etcd-K8S-master01_kube-system_c94bb8ceba1b924e6e3175228b168fe0_0
    [root@K8S-master01 K8S-ha-install]# docker rm 600e5da5c196 5a1848d923a1
280fc86494f1
    600e5da5c196
    5a1848d923a1
    280fc86494f1
    [root@K8S-master01 K8S-ha-install]# docker ps -a |grep Exited | grep -E
"600e5da5c196|5a1848d923a1|280fc86494f1"
```

删除本机镜像。比如删除 REPOSITORY 为 none 的镜像：

```
    [root@K8S-master01 K8S-ha-install]# docker images | grep none
    <none><none>                   7ad745acca31       2 days ago         5.83MB
    dotbalo/canary                                                       <none>
00f40cc9b7f6       2 days ago         5.83MB
    dotbalo/canary                                                       <none>
```

```
9b0f2f308931       2 days ago         5.83MB
    <none><none>                    c3d2357e9cbd       2 days ago         4.41MB
    dotbalo/nginx                                                         <none>
97c97cee03f9       3 days ago         109MB
    [root@K8S-master01 K8S-ha-install]# docker rmi 7ad745acca31 00f40cc9b7f6
9b0f2f308931 c3d2357e9cbd 97c97cee03f9
    Deleted:
sha256:7ad745acca31e3f753a3d50e45b7868e9a1aa177369757a9724bccf0654abcb2
    Deleted:
sha256:0546dcf8a97e167875d6563ef7f02ddd8ad3fc0d5f5c064b41e1ce67369b7e06
    Untagged:
dotbalo/canary@sha256:cdd99e578cb2cb8e84eaf2e077c2195a40948c9621d32004a9b5f4e8
2a408f4d
    Deleted:
sha256:00f40cc9b7f6946f17a0eb4fef859aa4e898d3170f023171d0502f8b447353a6
    Deleted:
sha256:7306c50196b5adc635e59152851dbb7fb2dc8782ecb217702849be26e3b1f2a5
    Deleted:
sha256:6b4fe6af6a9cd0d567326e718b91fdd5aca3d39d32bd40bbdd372430be286e3f
    Deleted:
sha256:b864518ff0e99c77046a58f6d82311c8eb64a88ed60bc28d8bd330137eddc024
    Untagged:
dotbalo/canary@sha256:8edea17bdeb346d20f1e93d0d4bf340f42ee8c8373885aa388c536e1
a718c7e7
    Deleted:
sha256:9b0f2f308931a88a5731955d58ae1226b5c147d8f372dae7c2250c0ff9854bf4
    Deleted:
sha256:659bcc00764060582794181890c8b63d6bbf60e8d3da035f76aa2f4d261742d7
    Deleted:
sha256:96163717e76d4b869461e39ed33c4e4066e7de44974557c6206d0f855fb58eb2
    Deleted:
sha256:4a04bebd433278ce549d9e941c2fc3f14021a450ed8ecf58f79c1668b6b9e72e
    Deleted:
sha256:5d65989598faa4ab6361db2655ab43866df88d850621d607474c165eefd6c73e
    Deleted:
sha256:ea8d75cec9b5fc0baf635c584fe818ba9fb2264a30f7210da2c58cfd71cc53b8
    Deleted:
sha256:c3d2357e9cbda84bc7feb1bbebbd3bd9bf6cd37b4415f8746f9cd95b8c11eb83
    Deleted:
sha256:6dc63f7195aae9d4b6764094fe786e32d590851f9646434d29d4fd40acf1c8ef
    Deleted:
sha256:03deec3a1538718aca4021e3f11293c55937fe1191e63a2c59948032e8ded166
    Deleted:
sha256:de87bb7eb02235ddc48979aee72779582ccf07e6b68685d905f8444e3cb5ed94
    Deleted:
sha256:7446f95fc910f657a872f3b99f4b0a91c8aa5471b90aa469a289d3b06f4be22c
    Deleted:
sha256:e901da9c8b00dd031d9ab42623f149a5247b92d78bee375e22ecb67f3b5911c3
    Deleted:
sha256:c9014ca736145dc855ed49b2d11e10fc68b1bcd94b2bf7fa43066d490ae0a7e3
    Deleted:
sha256:f8a9cd62cbd033d5f0cc292698c2ace8f9ca2e322dced364a8b6cbc67dd5d279
    Untagged:
dotbalo/nginx@sha256:deb5bcbfcedf451ddba3422a95b213613dc23c42ee6a63e746d09e040
e0cc7f8
```

```
    Deleted:
sha256:97c97cee03f9a552e4edf34766af09b7f6a74782776a199c5e7492971309158a
    Deleted:
sha256:697f26740b36e9a5aee72a4ca01cc6f644b59092d49ae043de9857e09ca9637e
```

镜像打标签（tag）。用于区分不同版本的镜像：

```
    [root@K8S-master01 K8S-ha-install]# docker images | grep nginx | tail -1
    nginx
1.7.9                    84581e99d807       3 years ago        91.7MB
    #不加 URL 一般为公网仓库中自己的仓库
    [root@K8S-master01 K8S-ha-install]# docker tag nginx dotbalo/nginx:v1
    #加 URL 一般为公司内部仓库或者其他仓库
    [root@K8S-master01 K8S-ha-install]# docker tag nginx
harbor.xxx.net/stage/nginx:v1
```

使用 dockerbuild 通过 Dockerfile 制作镜像。注意最后的一个点（.），表示使用当前目录的 Dockerfile：

```
    dockerbuild-t image_name:image_tag .
```

上述演示的都是 Docker 常用的基本命令，已可以满足日常需求，如果读者想要深入了解，可以参考 Docker 的相关资料。

2.1.3　Dockerfile 的编写

Dockerfile 是用来快速创建自定义镜像的一种文本格式的配置文件，在持续集成和持续部署时，需要使用 Dockerfile 生成相关应用程序的镜像，然后推送到公司内部仓库中，再通过部署策略把镜像部署到 Kubernetes 中。

通过 Dockerfile 提供的命令可以构建 Dockerfile 文件，Dockerfile 的常用命令如下：

```
    FROM：继承基础镜像
    MAINTAINER：镜像制作作者的信息
    RUN：用来执行 shell 命令
    EXPOSE：暴露端口号
    CMD：启动容器默认执行的命令，会被覆盖
    ENTRYPOINT：启动容器真正执行的命令，不会被覆盖
    VOLUME：创建挂载点
    ENV：配置环境变量
    ADD：复制文件到容器，一般拷贝文件，压缩包自动解压
    COPY：复制文件到容器，一般拷贝目录
    WORKDIR：设置容器的工作目录
    USER：容器使用的用户
```

以下简单演示每个命令的使用方法。

使用 RUN 创建一个用户：

```
    [root@DockerTestServer test]# cat Dockerfile
    # base image
    FROM centos:6
    MAINTAINER dot
    RUN useradd dot
```

```
执行构建
docker build -t centos:user .
```

使用 ENV 定义环境变量并用 CMD 执行命令：

```
[root@DockerTestServer test]# cat Dockerfile
# base image
FROM centos:6
MAINTAINER dot
RUN useradd dot
RUN mkdir dot
ENV envir=test version=1.0
CMD echo "envir:$envir version:$version"
```

执行构建并启动测试：

```
#执行构建
docker build -t centos:env-cmd .
#启动镜像验证 ENV 和 CMD
[root@DockerTestServer test]# docker run centos:env-cmd
envir:test version:1.0
```

使用 ADD 添加一个压缩包，使用 WORKDIR 改变工作目录：

```
# base image
FROM nginx
MAINTAINER dot
ADD ./index.tar.gz /usr/share/nginx/html/
WORKDIR /usr/share/nginx/html
```

使用 COPY 拷贝指定目录下的所有文件到容器，不包括本级目录。

此时只会拷贝 webroot 下的所有文件，不会将 webroot 拷贝过去：

```
# base image
FROM nginx
MAINTAINER dot
ADD ./index.tar.gz /usr/share/nginx/html/
WORKDIR /usr/share/nginx/html
COPY webroot/ .
```

设置启动容器的用户，在生产环境中一般不建议使用 root 启动容器，所以可以根据公司业务场景自定义启动容器的用户：

```
# base image
FROM centos:6
MAINTAINER dot

ADD ./index.tar.gz /usr/share/nginx/html/
WORKDIR /usr/share/nginx/html
COPY webroot/ .
RUN useradd -m tomcat -u 1001
USER 1001
```

使用 Volume 创建容器可挂载点：

```
# base image
FROM centos:6
```

```
MAINTAINER dot

VOLUME /data
```

挂载 Web 目录到/data，注意，对于宿主机路径，要写绝对路径：

```
docker run -ti --rm -v `pwd`/web:/data centos:volume bash
```

2.2　Kubernetes 基础

Kubernetes 致力于提供跨主机集群的自动部署、扩展、高可用以及运行应用程序容器的平台，其遵循主从式架构设计，其组件可以分为管理单个节点（Node）组件和控制平面组件。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上运行，也可以在支持高可用集群的多个节点上运行。本节主要介绍 Kubernetes 的重要概念和相关组件。

2.2.1　Master 节点

Master 节点是 Kubernetes 集群的控制节点，在生产环境中不建议部署集群核心组件外的任何 Pod，公司业务的 Pod 更是不建议部署到 Master 节点上，以免升级或者维护时对业务造成影响。

Master 节点的组件包括：

- APIServer。APIServer 是整个集群的控制中枢，提供集群中各个模块之间的数据交换，并将集群状态和信息存储到分布式键-值（key-value）存储系统 Etcd 集群中。同时它也是集群管理、资源配额、提供完备的集群安全机制的入口，为集群各类资源对象提供增删改查以及 watch 的 REST API 接口。APIServer 作为 Kubernetes 的关键组件，使用 Kubernetes API 和 JSON over HTTP 提供 Kubernetes 的内部和外部接口。
- Scheduler。Scheduler 是集群 Pod 的调度中心，主要是通过调度算法将 Pod 分配到最佳的节点（Node），它通过 APIServer 监听所有 Pod 的状态，一旦发现新的未被调度到任何 Node 节点的 Pod（PodSpec.NodeName 为空），就会根据一系列策略选择最佳节点进行调度，对每一个 Pod 创建一个绑定（binding），然后被调度的节点上的 Kubelet 负责启动该 Pod。Scheduler 是集群可插拔式组件，它跟踪每个节点上的资源利用率以确保工作负载不会超过可用资源。因此 Scheduler 必须知道资源需求、资源可用性以及其他约束和策略，例如服务质量、亲和力/反关联性要求、数据位置等。Scheduler 将资源供应与工作负载需求相匹配以维持系统的稳定和可靠，因此 Scheduler 在调度的过程中需要考虑公平、资源高效利用、效率等方面的问题。
- Controller Manager。Controller Manager 是集群状态管理器（它的英文直译名为控制器管理器），以保证 Pod 或其他资源达到期望值。当集群中某个 Pod 的副本数或其他资源因故障和错误导致无法正常运行，没有达到设定的值时，Controller Manager 会尝试自动修复并使其达到期望状态。Controller Manager 包含 NodeController、ReplicationController、

EndpointController、NamespaceController、ServiceAccountController、ResourceQuotaController、ServiceController和TokenController，该控制器管理器可与API服务器进行通信以在需要时创建、更新或删除它所管理的资源，如 Pod、服务断点等。

- Etcd。Etcd 由 CoreOS 开发，用于可靠地存储集群的配置数据，是一种持久性、轻量型、分布式的键-值（key-value）数据存储组件。Etcd 作为 Kubernetes 集群的持久化存储系统，集群的灾难恢复和状态信息存储都与其密不可分，所以在 Kubernetes 高可用集群中，Etcd 的高可用是至关重要的一部分，在生产环境中建议部署为大于 3 的奇数个数的 Etcd，以保证数据的安全性和可恢复性。Etcd 可与 Master 组件部署在同一个节点上，大规模集群环境下建议部署在集群外，并且使用高性能服务器来提高 Etcd 的性能和降低 Etcd 同步数据的延迟。

2.2.2 Node 节点

Node 节点也被称为 Worker 或 Minion，是主要负责部署容器（工作负载）的单机（或虚拟机），集群中的每个节点都必须具备容器的运行环境（runtime），比如 Docker 及其他组件等。

Kubelet 作为守护进程运行在 Node 节点上，负责监听该节点上所有的 Pod，同时负责上报该节点上所有 Pod 的运行状态，确保节点上的所有容器都能正常运行。当 Node 节点宕机（NotReady 状态）时，该节点上运行的 Pod 会被自动地转移到其他节点上。

Node 节点包括：

- Kubelet，负责与 Master 通信协作，管理该节点上的 Pod。
- Kube-Proxy，负责各 Pod 之间的通信和负载均衡。
- Docker Engine，Docker 引擎，负载对容器的管理。

2.2.3 Pod

1. 什么是 Pod

Pod 可简单地理解为是一组、一个或多个容器，具有共享存储/网络及如何运行容器的规范。Pad 包含一个或多个相对紧密耦合的应用程序容器，处于同一个 Pod 中的容器共享同样的存储空间（Volume，卷或存储卷）、IP 地址和 Port 端口，容器之间使用 localhost:port 相互访问。根据 Docker 的构造，Pod 可被建模为一组具有共享命令空间、卷、IP 地址和 Port 端口的 Docker 容器。

Pod 包含的容器最好是一个容器只运行一个进程。每个 Pod 包含一个 pause 容器，pause 容器是 Pod 的父容器，它主要负责僵尸进程的回收管理。

Kubernetes 为每个 Pod 都分配一个唯一的 IP 地址，这样就可以保证应用程序使用同一端口，避免了发生冲突的问题。

一个 Pod 的状态信息保存在 PodStatus 对象中，在 PodStatus 中有一个 Phase 字段，用于描述 Pod 在其生命周期中的不同状态，参考表 2-1。

表 2-1　Pod 状态字段 Phase 的不同取值

状态	说明
Pending（挂起）	Pod 已被 Kubernetes 系统接收，但仍有一个或多个容器未被创建。可以通过 describe 查看处于 Pending 状态的原因
Running（运行中）	Pod 已经被绑定到一个节点上，并且所有的容器都已经被创建。而且至少有一个是运行状态，或者是正在启动或者重启。可以通过 logs 查看 Pod 的日志
Succeeded（成功）	所有容器执行成功并终止，并且不会再次重启
Failed（失败）	所有容器都已终止，并且至少有一个容器以失败的方式终止，也就是说这个容器要么以非零状态退出，要么被系统终止
Unknown（未知）	通常是由于通信问题造成的无法获得 Pod 的状态

2. Pod 探针

Pod 探针用来检测容器内的应用是否正常，目前有三种实现方式，参考表 2-2。

表 2-2　Pod 探针的实现方式

实现方式	说明
ExecAction	在容器内执行一个指定的命令，如果命令返回值为 0，则认为容器健康
TCPSocketAction	通过 TCP 连接检查容器指定的端口，如果端口开放，则认为容器健康
HTTPGetAction	对指定的 URL 进行 Get 请求，如果状态码在 200~400 之间，则认为容器健康

Pod 探针每次检查容器后可能得到的容器状态，如表 2-3 所示。

表 2-3　Pod 探针检查容器后可能得到的状态

状态	说明
Success（成功）	容器通过检测
Failure（失败）	容器检测失败
Unknown（未知）	诊断失败，因此不采取任何措施

Kubelet 有两种探针（即探测器）可以选择性地对容器进行检测，参考表 2-4。

表 2-4　探针的种类

种类	说明
livenessProbe	用于探测容器是否在运行，如果探测失败，kubelet 会“杀死”容器并根据重启策略进行相应的处理。如果未指定该探针，将默认为 Success
readinessProbe	一般用于探测容器内的程序是否健康，即判断容器是否为就绪（Ready）状态。如果是，则可以处理请求，反之 Endpoints Controller 将从所有的 Services 的 Endpoints 中删除此容器所在 Pod 的 IP 地址。如果未指定，将默认为 Success

3. Pod 镜像拉取策略和重启策略

Pod 镜像拉取策略。用于配置当节点部署 Pod 时，对镜像的操作方式，参考表 2-5。

表 2-5　镜像拉取策略

操作方式	说明
Always	总是拉取，当镜像 tag 为 latest 时，默认为 Always

（续表）

操作方式	说明
Never	不管是否存在都不会拉取
IfNotPresent	镜像不存在时拉取镜像，默认，排除 latest

Pod 重启策略。在 Pod 发生故障时对 Pod 的处理方式参考表 2-6。

表 2-6 Pod 重启策略

操作方式	说明
Always	默认策略。容器失效时，自动重启该容器
OnFailure	容器以不为 0 的状态码终止，自动重启该容器
Never	无论何种状态，都不会重启

4. 创建一个 Pod

在生产环境中，很少会单独启动一个 Pod 直接使用，经常会用 Deployment、DaemonSet、StatefulSet 等方式调度并管理 Pod，定义 Pod 的参数同时适应于 Deployment、DaemonSet、StatefulSet 等方式。

在 Kubeadm 安装方式下，kubernetes 系统组件都是用单独的 Pod 启动的，当然有时候也会单独启动一个 Pod 用于测试业务等，此时可以单独创建一个 Pod。

创建一个 Pod 的标准格式如下：

```
apiVersion: v1 # 必选，API 的版本号
kind: Pod   # 必选，类型 Pod
metadata:   # 必选，元数据
  name: nginx   # 必选，符合 RFC 1035 规范的 Pod 名称
  namespace: web-testing # 可选，不指定默认为 default，Pod 所在的命名空间
  labels:   # 可选，标签选择器，一般用于 Selector
    - app: nginx
  annotations:  # 可选，注释列表
    - app: nginx
spec:   # 必选，用于定义容器的详细信息
  containers:   # 必选，容器列表
  - name: nginx # 必选，符合 RFC 1035 规范的容器名称
    image: nginx: v1# 必选，容器所用的镜像的地址
    imagePullPolicy: Always # 可选，镜像拉取策略
     command:
     - nginx # 可选，容器启动执行的命令
     - -g
     - "daemon off;"
         workingDir: /usr/share/nginx/html  # 可选，容器的工作目录
         volumeMounts:   # 可选，存储卷配置
         - name: webroot # 存储卷名称
           mountPath: /usr/share/nginx/html # 挂载目录
           readOnly: true    # 只读
         ports:  # 可选，容器需要暴露的端口号列表
         - name: http    # 端口名称
           containerPort: 80 # 端口号
           protocol: TCP # 端口协议，默认 TCP
```

```
    env:    # 可选，环境变量配置
    - name: TZ  # 变量名
      value: Asia/Shanghai
    - name: LANG
      value: en_US.utf8
    resources:  # 可选，资源限制和资源请求限制
      limits:   # 最大限制设置
        cpu: 1000m
        memory: 1024MiB
      requests: # 启动所需的资源
        cpu: 100m
        memory: 512MiB
    readinessProbe: # 可选，容器状态检查
    httpGet:    # 检测方式
        path: / # 检查路径
        port: 80    # 监控端口
      timeoutSeconds: 2 # 超时时间
      initialDelaySeconds: 60   # 初始化时间
    livenessProbe:  # 可选，监控状态检查
      exec: # 检测方式
        command:
        - cat
        - /health
      httpGet:  # 检测方式
        path: /_health
        port: 8080
        httpHeaders:
        - name: end-user
          value: jason
      tcpSocket:    # 检测方式
        port: 80
      initialDelaySeconds: 60   # 初始化时间
      timeoutSeconds: 2 # 超时时间
      periodSeconds: 5  # 检测间隔
      successThreshold: 2 # 检查成功为 2 次表示就绪
      failureThreshold: 1 # 检测失败 1 次表示未就绪
    securityContext:    # 可选，限制容器不可信的行为
      provoleged: false
  restartPolicy: Always # 可选，默认为 Always
  nodeSelector: # 可选，指定 Node 节点
    region: subnet7
  imagePullSecrets: # 可选，拉取镜像使用的 secret
  - name: default-dockercfg-86258
  hostNetwork: false    # 可选，是否为主机模式，如是，会占用主机端口
  volumes:  # 共享存储卷列表
  - name: webroot # 名称，与上述对应
    emptyDir: {}    # 共享卷类型，空
    hostPath:       # 共享卷类型，本机目录
      path: /etc/hosts
    secret: # 共享卷类型，secret 模式，一般用于密码
      secretName: default-token-tf2jp # 名称
      defaultMode: 420 # 权限
      configMap:    # 一般用于配置文件
```

```
            name: nginx-conf
            defaultMode: 420
```

2.2.4 Label 和 Selector

当 Kubernetes 对系统的任何 API 对象如 Pod 和节点进行“分组”时，会对其添加 Label（key=value 形式的“键-值对”）用以精准地选择对应的 API 对象。而 Selector（标签选择器）则是针对匹配对象的查询方法。注：键-值对就是 key-value pair。

例如，常用的标签 tier 可用于区分容器的属性，如 frontend、backend；或者一个 release_track 用于区分容器的环境，如 canary、production 等。

1. 定义 Label

应用案例：

公司与 xx 银行有一条专属的高速光纤通道，此通道只能与 192.168.7.0 网段进行通信，因此只能将与 xx 银行通信的应用部署到 192.168.7.0 网段所在的节点上，此时可以对节点进行 Label（即加标签）：

```
[root@K8S-master01 ~]# kubectl label node K8S-node02 region=subnet7
node/K8S-node02 labeled
```

然后，可以通过 Selector 对其筛选：

```
[root@K8S-master01 ~]# kubectl get no -l region=subnet7
NAME        STATUS   ROLES    AGE    VERSION
K8S-node02   Ready    <none>   3d17h   v1.12.3
```

最后，在 Deployment 或其他控制器中指定将 Pod 部署到该节点：

```
containers:
  ......
dnsPolicy: ClusterFirst
nodeSelector:
  region: subnet7
restartPolicy: Always
......
```

也可以用同样的方式对 Service 进行 Label：

```
[root@K8S-master01 ~]# kubectl label svc canary-v1 -n canary-production
env=canary version=v1
service/canary-v1 labeled
```

查看 Labels：

```
[root@K8S-master01 ~]# kubectl get svc -n canary-production --show-labels
NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   LABELS
canary-v1   ClusterIP   10.110.253.62   <none>       8080/TCP   24h
env=canary,version=v1
```

还可以查看所有 Version 为 v1 的 svc：

```
[root@K8S-master01 canary]# kubectl get svc --all-namespaces -l version=v1
```

```
    NAMESPACE          NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
    canary-production   canary-v1   ClusterIP   10.110.253.62   <none>
8080/TCP   25h
```

其他资源的 Label 方式相同。

2. Selector 条件匹配

Selector 主要用于资源的匹配，只有符合条件的资源才会被调用或使用，可以使用该方式对集群中的各类资源进行分配。

假如对 Selector 进行条件匹配，目前已有的 Label 如下：

```
    [root@K8S-master01 ~]# kubectl get svc --show-labels
    NAME         TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
LABELS
    details      ClusterIP   10.99.9.178     <none>       9080/TCP   45h
app=details
    kubernetes    ClusterIP   10.96.0.1       <none>       443/TCP    3d19h
component=apiserver,provider=kubernetes
    nginx        ClusterIP   10.106.194.137   <none>       80/TCP     2d21h
app=productpage,version=v1
    nginx-v2     ClusterIP   10.108.176.132   <none>       80/TCP     2d20h
<none>
    productpage   ClusterIP   10.105.229.52    <none>       9080/TCP   45h
app=productpage,tier=frontend
    ratings      ClusterIP   10.96.104.95     <none>       9080/TCP   45h
app=ratings
    reviews      ClusterIP   10.102.188.143   <none>       9080/TCP   45h
app=reviews
```

选择 app 为 reviews 或者 productpage 的 svc：

```
    [root@K8S-master01 ~]# kubectl get svc -l  'app in (details, productpage)'
--show-labels
    NAME         TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
LABELS
    details      ClusterIP   10.99.9.178     <none>       9080/TCP   45h
app=details
    nginx        ClusterIP   10.106.194.137   <none>       80/TCP     2d21h
app=productpage,version=v1
    productpage   ClusterIP   10.105.229.52    <none>       9080/TCP   45h
app=productpage,tier=frontend
```

选择 app 为 productpage 或 reviews 但不包括 version=v1 的 svc：

```
    [root@K8S-master01 ~]# kubectl get svc -l  version!=v1,'app in (details,
productpage)' --show-labels
    NAME         TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   LABELS
    details      ClusterIP   10.99.9.178     <none>       9080/TCP   45h
app=details
    productpage   ClusterIP   10.105.229.52   <none>       9080/TCP   45h
app=productpage,tier=frontend
```

选择 labelkey 名为 app 的 svc：

```
    [root@K8S-master01 ~]# kubectl get svc -l app --show-labels
    NAME         TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
LABELS
    details      ClusterIP   10.99.9.178     <none>       9080/TCP   45h
app=details
    nginx        ClusterIP   10.106.194.137   <none>       80/TCP    2d21h
app=productpage,version=v1
    productpage   ClusterIP   10.105.229.52    <none>       9080/TCP   45h
app=productpage,tier=frontend
    ratings      ClusterIP   10.96.104.95    <none>       9080/TCP   45h
app=ratings
    reviews      ClusterIP   10.102.188.143   <none>       9080/TCP   45h
app=reviews
```

3. 修改标签（Label）

在实际使用中，Label 的更改是经常发生的事情，可以使用 overwrite 参数修改标签。

修改标签，比如将 version=v1 改为 version=v2：

```
    [root@K8S-master01 canary]# kubectl get svc -n canary-production --show-labels
    NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   LABELS
    canary-v1   ClusterIP   10.110.253.62   <none>       8080/TCP   26h
env=canary,version=v1
    [root@K8S-master01 canary]# kubectl label svc canary-v1 -n canary-production
version=v2 --overwrite
    service/canary-v1 labeled
    [root@K8S-master01 canary]# kubectl get svc -n canary-production --show-labels
    NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   LABELS
    canary-v1   ClusterIP   10.110.253.62   <none>       8080/TCP   26h
env=canary,version=v2
```

4. 删除标签（Label）

删除标签，比如删除 version：

```
    [root@K8S-master01 canary]# kubectl label svc canary-v1 -n canary-production
version-
    service/canary-v1 labeled
    [root@K8S-master01 canary]# kubectl get svc -n canary-production --show-labels
    NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   LABELS
    canary-v1   ClusterIP   10.110.253.62   <none>       8080/TCP   26h
env=canary
```

2.2.5 Replication Controller 和 ReplicaSet

Replication Controller（复制控制器，RC）和 ReplicaSet（复制集，RS）是两种部署 Pod 的方式。因为在生产环境中，主要使用更高级的 Deployment 等方式进行 Pod 的管理和部署，所以本节只对 Replication Controller 和 Replica Set 的部署方式进行简单介绍。

1. Replication Controller

Replication Controller 可确保 Pod 副本数达到期望值，也就是 RC 定义的数量。换句话说，Replication Controller 可确保一个 Pod 或一组同类 Pod 总是可用。

如果存在的 Pod 大于设定的值，则 Replication Controller 将终止额外的 Pod。如果太小，Replication Controller 将启动更多的 Pod 用于保证达到期望值。与手动创建 Pod 不同的是，用 Replication Controller 维护的 Pod 在失败、删除或终止时会自动替换。因此即使应用程序只需要一个 Pod，也应该使用 Replication Controller。Replication Controller 类似于进程管理程序，但是 Replication Controller 不是监视单个节点上的各个进程，而是监视多个节点上的多个 Pod。

定义一个 Replication Controller 的示例如下。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

2. ReplicaSet

ReplicaSet 是支持基于集合的标签选择器的下一代 Replication Controller，它主要用作 Deployment 协调创建、删除和更新 Pod，和 Replication Controller 唯一的区别是，ReplicaSet 支持标签选择器。在实际应用中，虽然 ReplicaSet 可以单独使用，但是一般建议使用 Deployment（部署）来自动管理 ReplicaSet，除非自定义的 Pod 不需要更新或有其他编排等。

定义一个 ReplicaSet 的示例如下：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
```

```
        labels:
          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
            # If your cluster config does not include a dns service, then to
            # instead access environment variables to find service host
            # info, comment out the 'value: dns' line above, and uncomment the
            # line below.
            # value: env
          ports:
          - containerPort: 80
```

2.2.6 Deployment

虽然 ReplicaSet 可以确保在任何给定时间运行的 Pod 副本达到指定的数量，但是 Deployment（部署）是一个更高级的概念，它管理 ReplicaSet 并为 Pod 和 ReplicaSet 提供声明性更新以及许多其他有用的功能，所以建议在实际使用中，使用 Deployment 代替 ReplicaSet。

如果在 Deployment 对象中描述了所需的状态，Deployment 控制器就会以可控制的速率将实际状态更改为期望状态。也可以在 Deployment 中创建新的 ReplicaSet，或者删除现有的 Deployment 并使用新的 Deployment 部署所用的资源。

1. 创建 Deployment

创建一个 Deployment 文件，并命名为 dc-nginx.yaml，用于部署三个 Nginx Pod：

```
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:1.7.9
          ports:
          - containerPort: 80
```

示例解析

- nginx-deployment：Deployment 的名称。
- replicas：创建 Pod 的副本数。
- selector：定义 Deployment 如何找到要管理的 Pod，与 template 的 label（标签）对应。
- template 字段包含以下字段：
  - app　nginx 使用 label（标签）标记 Pod。
  - spec　表示 Pod 运行一个名字为 nginx 的容器。
  - image　运行此 Pod 使用的镜像。
  - Port　容器用于发送和接收流量的端口。

使用 kubectlcreate 创建此 Deployment：

```
[root@K8S-master01 2.2.8.1]# kubectl create -f dc-nginx.yaml
deployment.apps/nginx-deployment created
```

使用 kubectlget 或者 kubectldescribe 查看此 Deployment：

```
[root@K8S-master01 2.2.8.1]# kubectl get deploy
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3          1          60s
```

其中，

- NAME：集群中 Deployment 的名称。
- DESIRED：应用程序副本数。
- CURRENT：当前正在运行的副本数。
- UP-TO-DATE：显示已达到期望状态的被更新的副本数。
- AVAILABLE：显示用户可以使用的应用程序副本数，当前为 1，因为部分 Pod 仍在创建过程中。
- AGE：显示应用程序运行的时间。

查看此时 Deployment rollout 的状态：

```
[root@K8S-master01 2.2.8.1]# kubectl rollout status
deployment/nginx-deployment
deployment "nginx-deployment" successfully rolled out
```

再次查看此 Deployment：

```
[root@K8S-master01 2.2.8.1]# kubectl get deploy
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3          3          11m
```

查看此 Deployment 创建的 ReplicaSet：

```
[root@K8S-master01 2.2.8.1]# kubectl get rs
```

```
    NAME                          DESIRED   CURRENT   READY   AGE
    nginx-deployment-5c689d88bb   3         3         3       12m
```

注 意

ReplicaSet（复制集，RS）的命名格式为[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]POD- TEMPLATE-HASH-VALUE，是自动生成的，不要手动指定。

查看此 Deployment 创建的 Pod：

```
    [root@K8S-master01 2.2.8.1]# kubectl get pods --show-labels
    NAME                               READY   STATUS    RESTARTS   AGE   LABELS
    nginx-deployment-5c689d88bb-6b95k   1/1     Running   0          13m
app=nginx,pod-template-hash=5c689d88bb
    nginx-deployment-5c689d88bb-9z5z2   1/1     Running   0          13m
app=nginx,pod-template-hash=5c689d88bb
    nginx-deployment-5c689d88bb-jc8hr   1/1     Running   0          13m
app=nginx,pod-template-hash=5c689d88bb
```

2. 更新 Deployment

一般对应用程序升级或者版本迭代时，会通过 Deployment 对 Pod 进行滚动更新。

注 意

当且仅当 Deployment 的 Pod 模板（即.spec.template）更改时，才会触发 Deployment 更新，例如更新 label（标签）或者容器的 image（镜像）。

假如更新 Nginx Pod 的 image 使用 nginx:1.9.1：

```
    [root@K8S-master01 2.2.8.1]# kubectl set image deployment nginx-deployment
nginx=nginx:1.9.1 --record
    deployment.extensions/nginx-deployment image updated
```

当然也可以直接编辑 Deployment，效果相同：

```
    [root@K8S-master01 2.2.8.1]#   kubectl edit
deployment.v1.apps/nginx-deployment
    deployment.apps/nginx-deployment edited
```

使用 kubectl rollout status 查看更新状态：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout status
deployment.v1.apps/nginx-deployment
    Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new
replicas have been updated...
    Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new
replicas have been updated...
    Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new
replicas have been updated...
    Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new
replicas have been updated...
    Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas
are pending termination...
    Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas
```

```
are pending termination...
    deployment "nginx-deployment" successfully rolled out
```

查看 ReplicaSet：

```
    [root@K8S-master01 2.2.8.1]# kubectl get rs
    NAME                     DESIRED   CURRENT   READY   AGE
    nginx-deployment-5c689d88bb   0         0         0      34m
    nginx-deployment-6987cdb55b   3         3         3      5m14s
```

通过 describe 查看 Deployment 的详细信息：

```
    [root@K8S-master01 2.2.8.1]# kubectl describe deploy nginx-deployment
    Name:               nginx-deployment
    Namespace:          default
    CreationTimestamp:     Thu, 24 Jan 2019 15:15:15 +0800
    Labels:             app=nginx
    Annotations:          deployment.kubernetes.io/revision: 2
                        kubernetes.io/change-cause: kubectl set image deployment
nginx-deployment nginx=nginx:1.9.1 --record=true
    Selector:           app=nginx
    Replicas:           3 desired | 3 updated | 3 total | 3 available | 0
unavailable
    StrategyType:          RollingUpdate
    MinReadySeconds:       0
    RollingUpdateStrategy:  25% max unavailable, 25% max surge
    Pod Template:
      Labels:  app=nginx
      Containers:
       nginx:
        Image:       nginx:1.9.1
        Port:        80/TCP
        Host Port:    0/TCP
        Environment:  <none>
        Mounts:      <none>
      Volumes:       <none>
    Conditions:
      Type          Status  Reason
      ----          ------  ------
      Available     True    MinimumReplicasAvailable
      Progressing   True    NewReplicaSetAvailable
    OldReplicaSets:  <none>
    NewReplicaSet:   nginx-deployment-6987cdb55b (3/3 replicas created)
    Events:
      Type    Reason           Age    From                   Message
      ----    ------           ----   ----                   -------
      Normal  ScalingReplicaSet  36m   deployment-controller  Scaled up replica
set nginx-deployment-5c689d88bb to 3
      Normal  ScalingReplicaSet  7m16s  deployment-controller  Scaled up replica
set nginx-deployment-6987cdb55b to 1
      Normal  ScalingReplicaSet  5m18s  deployment-controller  Scaled down
replica set nginx-deployment-5c689d88bb to 2
      Normal  ScalingReplicaSet  5m18s  deployment-controller  Scaled up replica
set nginx-deployment-6987cdb55b to 2
      Normal  ScalingReplicaSet  4m35s  deployment-controller  Scaled down
replica set nginx-deployment-5c689d88bb to 1
```

```
    Normal  ScalingReplicaSet  4m34s  deployment-controller  Scaled up replica
set nginx-deployment-6987cdb55b to 3
    Normal  ScalingReplicaSet  3m30s  deployment-controller  Scaled down
replica set nginx-deployment-5c689d88bb to 0
```

在 describe 中可以看出，第一次创建时，它创建了一个名为 nginx-deployment-5c689d88bb 的 ReplicaSet，并直接将其扩展为 3 个副本。更新部署时，它创建了一个新的 ReplicaSet，命名为 nginx-deployment-6987cdb55b，并将其副本数扩展为 1，然后将旧的 ReplicaSet 缩小为 2，这样至少可以有 2 个 Pod 可用，最多创建了 4 个 Pod。以此类推，使用相同的滚动更新策略向上和向下扩展新旧 ReplicaSet，最终新的 ReplicaSet 可以拥有 3 个副本，并将旧的 ReplicaSet 缩小为 0。

3. 回滚 Deployment

当新版本不稳定时，可以对其进行回滚操作，默认情况下，所有 Deployment 的 rollout 历史都保留在系统中，可以随时回滚。

假设我们又进行了几次更新：

```
    [root@K8S-master01 2.2.8.1]# kubectl set image deployment nginx-deployment
nginx=dotbalo/canary:v1 --record
    [root@K8S-master01 2.2.8.1]# kubectl set image deployment nginx-deployment
nginx=dotbalo/canary:v2 --record
```

使用 kubectl rollout history 查看部署历史：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout history
deployment/nginx-deployment
    deployment.extensions/nginx-deployment
    REVISION  CHANGE-CAUSE
    1         <none>
    2         kubectl set image deployment nginx-deployment nginx=nginx:1.9.1
--record=true
    3         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v1
--record=true
    4         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v2
--record=true
```

查看 Deployment 某次更新的详细信息，使用--revision 指定版本号：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout history
deployment.v1.apps/nginx-deployment --revision=3
    deployment.apps/nginx-deployment with revision #3
    Pod Template:
      Labels:   app=nginx
        pod-template-hash=645959bf6b
      Annotations:  kubernetes.io/change-cause: kubectl set image deployment
nginx-deployment nginx=dotbalo/canary:v1 --record=true
      Containers:
       nginx:
        Image:  dotbalo/canary:v1
        Port:   80/TCP
        Host Port:  0/TCP
        Environment:<none>
        Mounts: <none>
      Volumes:  <none>
```

使用 kubectl rollout undo 回滚到上一个版本：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout undo
deployment.v1.apps/nginx-deployment
    deployment.apps/nginx-deployment
```

再次查看更新历史，发现 REVISION5 回到了 canary:v1：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout history
deployment/nginx-deployment
    deployment.extensions/nginx-deployment
    REVISION  CHANGE-CAUSE
    1         <none>
    2         kubectl set image deployment nginx-deployment nginx=nginx:1.9.1
--record=true
    4         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v2
--record=true
    5         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v1
--record=true
```

使用--to-revision 参数回到指定版本：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout undo deployment/nginx-deployment
--to-revision=2
    deployment.extensions/nginx-deployment
```

4. 扩展 Deployment

当公司访问量变大，三个 Pod 已无法支撑业务时，可以对其进行扩展。

使用 kubectl scale 动态调整 Pod 的副本数，比如增加 Pod 为 5 个：

```
    [root@K8S-master01 2.2.8.1]# kubectl scale
deployment.v1.apps/nginx-deployment --replicas=5
    deployment.apps/nginx-deployment scaled
```

查看 Pod，此时 Pod 已经变成了 5 个：

```
    [root@K8S-master01 2.2.8.1]# kubectl get po
    NAME                              READY   STATUS    RESTARTS   AGE
    nginx-deployment-5f89547d9c-5r56b   1/1     Running   0          90s
    nginx-deployment-5f89547d9c-htmn7   1/1     Running   0          25s
    nginx-deployment-5f89547d9c-nwxs2   1/1     Running   0          99s
    nginx-deployment-5f89547d9c-rpwlg   1/1     Running   0          25s
    nginx-deployment-5f89547d9c-vlr5p   1/1     Running   0          95s
```

5. 暂停和恢复 Deployment 更新

Deployment 支持暂停更新，用于对 Deployment 进行多次修改操作。

使用 kubectl rollout pause 暂停 Deployment 更新：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout pause
deployment/nginx-deployment
    deployment.extensions/nginx-deployment paused
```

然后对 Deployment 进行相关更新操作，比如更新镜像，然后对其资源进行限制：

```
    [root@K8S-master01 2.2.8.1]# kubectl set image
deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1
```

```
    deployment.apps/nginx-deployment image updated
    [root@K8S-master01 2.2.8.1]# kubectl set resources
deployment.v1.apps/nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi
    deployment.apps/nginx-deployment resource requirements updated
```

通过 rollout history 可以看到没有新的更新：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout history
deployment.v1.apps/nginx-deployment
    deployment.apps/nginx-deployment
    REVISION  CHANGE-CAUSE
    1         <none>
    5         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v1
--record=true
    7         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v2
--record=true
    8         kubectl set image deployment nginx-deployment nginx=dotbalo/canary:v2
--record=true
```

使用 kubectl rollout resume 恢复 Deployment 更新：

```
    [root@K8S-master01 2.2.8.1]# kubectl rollout resume
deployment.v1.apps/nginx-deployment
    deployment.apps/nginx-deployment resumed
```

可以查看到恢复更新的 Deployment 创建了一个新的 RS（复制集）：

```
    [root@K8S-master01 2.2.8.1]# kubectl get rs
    NAME                          DESIRED   CURRENT   READY   AGE
    nginx-deployment-57895845b8   5         5         4       11s
```

可以查看 Deployment 的 image（镜像）已经变为 nginx:1.9.1

```
    [root@K8S-master01 2.2.8.1]# kubectl describe deploy nginx-deployment
    Name:                   nginx-deployment
    Namespace:              default
    CreationTimestamp:      Thu, 24 Jan 2019 15:15:15 +0800
    Labels:                 app=nginx
    Annotations:            deployment.kubernetes.io/revision: 9
                            kubernetes.io/change-cause: kubectl set image deployment
nginx-deployment nginx=dotbalo/canary:v2 --record=true
    Selector:               app=nginx
    Replicas:               5 desired | 5 updated | 5 total | 5 available | 0
unavailable
    StrategyType:           RollingUpdate
    MinReadySeconds:        0
    RollingUpdateStrategy:  25% max unavailable, 25% max surge
    Pod Template:
      Labels:  app=nginx
      Containers:
       nginx:
        Image:      nginx:1.9.1
        Port:       80/TCP
        Host Port:  0/TCP
```

6. 更新 Deployment 的注意事项

（1）清理策略

在默认情况下，revision 保留 10 个旧的 ReplicaSet，其余的将在后台进行垃圾回收，可以在.spec.revisionHistoryLimit 设置保留 ReplicaSet 的个数。当设置为 0 时，不保留历史记录。

（2）更新策略

- .spec.strategy.type==Recreate，表示重建，先删掉旧的 Pod 再创建新的 Pod。
- .spec.strategy.type==RollingUpdate，表示滚动更新，可以指定 maxUnavailable 和 maxSurge 来控制滚动更新过程。
  - .spec.strategy.rollingUpdate.maxUnavailable，指定在回滚更新时最大不可用的 Pod 数量，可选字段，默认为 25%，可以设置为数字或百分比，如果 maxSurge 为 0，则该值不能为 0。
  - .spec.strategy.rollingUpdate.maxSurge 可以超过期望值的最大 Pod 数，可选字段，默认为 25%，可以设置成数字或百分比，如果 maxUnavailable 为 0，则该值不能为 0。

（3）Ready 策略

.spec.minReadySeconds 是可选参数，指定新创建的 Pod 应该在没有任何容器崩溃的情况下视为 Ready（就绪）状态的最小秒数，默认为 0，即一旦被创建就视为可用，通常和容器探针连用。

2.2.7　StatefulSet

StatefulSet（有状态集）常用于部署有状态的且需要有序启动的应用程序。

1. StatefulSet 的基本概念

StatefulSet 主要用于管理有状态应用程序的工作负载 API 对象。比如在生产环境中，可以部署 ElasticSearch 集群、MongoDB 集群或者需要持久化的 RabbitMQ 集群、Redis 集群、Kafka 集群和 ZooKeeper 集群等。

和 Deployment 类似，一个 StatefulSet 也同样管理着基于相同容器规范的 Pod。不同的是，StatefulSet 为每个 Pod 维护了一个粘性标识。这些 Pod 是根据相同的规范创建的，但是不可互换，每个 Pod 都有一个持久的标识符，在重新调度时也会保留，一般格式为 StatefulSetName-Number。比如定义一个名字是 Redis-Sentinel 的 StatefulSet，指定创建三个 Pod，那么创建出来的 Pod 名字就为 Redis-Sentinel-0、Redis-Sentinel-1、Redis-Sentinel-2。而 StatefulSet 创建的 Pod 一般使用 Headless Service（无头服务）进行通信，和普通的 Service 的区别在于 Headless Service 没有 ClusterIP，它使用的是 Endpoint 进行互相通信，Headless 一般的格式为：

statefulSetName-{0..N-1}.serviceName.namespace.svc.cluster. local。

说明：

- serviceName 为 Headless Service 的名字。
- 0..N-1 为 Pod 所在的序号，从 0 开始到 N-1。

- statefulSetName 为 StatefulSet 的名字。
- namespace 为服务所在的命名空间。
- .cluster.local 为 Cluster Domain（集群域）。

比如，一个 Redis 主从架构，Slave 连接 Master 主机配置就可以使用不会更改的 Master 的 Headless Service，例如 Redis 从节点（Slave）配置文件如下：

```
    port 6379
    slaveofredis-sentinel-master-ss-0.redis-sentinel-master-ss.public-service.
svc.cluster.local 6379
    tcp-backlog 511
    timeout 0
    tcp-keepalive 0
    ……
```

其中，redis-sentinel-master-ss-0.redis-sentinel-master-ss.public-service.svc.cluster.local 是 Redis Master 的 Headless Service。具体 Headless 可以参考 2.2.13 节。

2. 使用 StatefulSet

一般 StatefulSet 用于有以下一个或者多个需求的应用程序：

- 需要稳定的独一无二的网络标识符。
- 需要持久化数据。
- 需要有序的、优雅的部署和扩展。
- 需要有序的、自动滚动更新。

如果应用程序不需要任何稳定的标识符或者有序的部署、删除或者扩展，应该使用无状态的控制器部署应用程序，比如 Deployment 或者 ReplicaSet。

3. StatefulSet 的限制

StatefulSet 是 Kubernetes 1.9 版本之前的 beta 资源，在 1.5 版本之前的任何 Kubernetes 版本都没有。

Pod 所用的存储必须由 PersistentVolume Provisioner（持久化卷配置器）根据请求配置 StorageClass，或者由管理员预先配置。

为了确保数据安全，删除和缩放 StatefulSet 不会删除与 StatefulSet 关联的卷，可以手动选择性地删除 PVC 和 PV（关于 PV 和 PVC 请参考 2.2.12 节）。

StatefulSet 目前使用 Headless Service（无头服务）负责 Pod 的网络身份和通信，但需要创建此服务。

删除一个 StatefulSet 时，不保证对 Pod 的终止，要在 StatefulSet 中实现 Pod 的有序和正常终止，可以在删除之前将 StatefulSet 的副本缩减为 0。

4. StatefulSet 组件

定义一个简单的 StatefulSet 的示例如下：

```
    apiVersion: v1
    kind: Service
```

```
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "nginx-storage-class"
      resources:
        requests:
          storage: 1Gi
```

其中，

- kind: Service 定义了一个名字为 Nginx 的 Headless Service，创建的 Service 格式为 nginx-0.nginx.default.svc.cluster.local，其他的类似，因为没有指定 Namespace（命名空间），所以默认部署在 default。
- kind: StatefulSet 定义了一个名字为 web 的 StatefulSet，replicas 表示部署 Pod 的副本数，本实例为 2。
- volumeClaimTemplates 表示将提供稳定的存储 PV（持久化卷）作持久化，PV 可以是手动创建或者自动创建。在上述示例中，每个 Pod 将配置一个 PV，当 Pod 重新调度到某个节

点上时，Pod 会重新挂载 volumeMounts 指定的目录（当前 StatefulSet 挂载到 /usr/share/nginx/html），当删除 Pod 或者 StatefulSet 时，不会删除 PV。

在 StatefulSet 中必须设置 Pod 选择器（.spec.selector）用来匹配其标签（.spec.template.metadata.labels）。在 1.8 版本之前，如果未配置该字段（.spec.selector），将被设置为默认值，在 1.8 版本之后，如果未指定匹配 Pod Selector，则会导致 StatefulSet 创建错误。

当 StatefulSet 控制器创建 Pod 时，它会添加一个标签 statefulset.kubernetes.io/pod-name，该标签的值为 Pod 的名称，用于匹配 Service。

5. 创建 StatefulSet

创建 StatefulSet 之前，需要提前创建 StatefulSet 持久化所用的 PersistentVolumes（持久化卷，以下简称 PV，也可以使用 emptyDir 不对数据进行保留），当然也可以使用动态方式自动创建 PV，关于 PV 将在 2.2.12 节进行详解，本节只作为演示使用，也可以先阅读 2.2.12 节进行了解。

本例使用 NFS 提供静态 PV，假如已有一台 NFS 服务器，IP 地址为 192.168.2.2，配置的共享目录如下：

```
[root@nfs web]# cat /etc/exports | tail -1
/nfs/web/ *(rw,sync,no_subtree_check,no_root_squash)
[root@nfs web]# exportfs -r
[root@nfs web]# systemctl reload nfs-server
[root@nfs web]# ls -l /nfs/web/
total 0
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx0
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx1
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx2
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx3
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx4
drwxr-xr-x 2 root root 6 Jan 31 17:22 nginx5
```

Nginx0-5 作为 StatefulSet Pod 的 PV 的数据存储目录，使用 PersistentVolume 创建 PV，文件如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nginx-5
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: "nginx-storage-class"
  nfs:
    # real share directory
    path: /nfs/web/nginx5
    # nfs real ip
    server: 192.168.2.2
```

具体参数的配置及其含义，可参考 2.2.12 节。

创建 PV：

```
[root@K8S-master01 2.2.7]# kubectl create -f web-pv.yaml
persistentvolume/pv-nginx-0 created
persistentvolume/pv-nginx-1 created
persistentvolume/pv-nginx-2 created
persistentvolume/pv-nginx-3 created
persistentvolume/pv-nginx-4 created
persistentvolume/pv-nginx-5 created
```

查看 PV：

```
[root@K8S-master01 2.2.7]# kubectl get pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
STORAGECLASS         REASON   AGE
pv-nginx-0   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
pv-nginx-1   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
pv-nginx-2   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
pv-nginx-3   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
pv-nginx-4   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
pv-nginx-5   1Gi       RWO          Recycle         Available
nginx-storage-class          26s
```

创建 StatefulSet：

```
[root@K8S-master01 2.2.7]# kubectl create -f sts-web.yaml
service/nginx created
statefulset.apps/web created
[root@K8S-master01 2.2.7]# kubectl get sts
NAME   DESIRED   CURRENT   AGE
web    2         2         12s
[root@K8S-master01 2.2.7]# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   7d2h
nginx        ClusterIP   None         <none>        80/TCP    16s
[root@K8S-master01 2.2.7]# kubectl get po -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          2m5s
web-1   1/1     Running   0          115s
```

查看 PVC 和 PV，可以看到 StatefulSet 创建的两个 Pod 的 PVC 已经和 PV 绑定成功：

```
[root@K8S-master01 2.2.7]# kubectl get pvc
NAME        STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS
AGE
www-web-0   Bound    pv-nginx-5   1Gi        RWO           nginx-storage-class
2m31s
www-web-1   Bound    pv-nginx-0   1Gi        RWO           nginx-storage-class
2m21s
[root@K8S-master01 2.2.7]# kubectl get pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
```

```
STORAGECLASS         REASON   AGE
    pv-nginx-0   1Gi       RWO          Recycle         Bound
default/www-web-1   nginx-storage-class          4m8s
    pv-nginx-1   1Gi       RWO          Recycle         Available
nginx-storage-class          4m8s
    pv-nginx-2   1Gi       RWO          Recycle         Available
nginx-storage-class          4m8s
    pv-nginx-3   1Gi       RWO          Recycle         Available
nginx-storage-class          4m8s
    pv-nginx-4   1Gi       RWO          Recycle         Available
nginx-storage-class          4m8s
    pv-nginx-5   1Gi       RWO          Recycle         Bound
default/www-web-0   nginx-storage-class          4m8s
```

6. 部署和扩展保障

Pod 的部署和扩展规则如下：

- 对于具有 N 个副本的 StatefulSet，将按顺序从 0 到 N-1 开始创建 Pod。
- 当删除 Pod 时，将按照 N-1 到 0 的反顺序终止。
- 在缩放 Pod 之前，必须保证当前的 Pod 是 Running（运行中）或者 Ready（就绪）。
- 在终止 Pod 之前，它所有的继任者必须是完全关闭状态。

StatefulSet 的 pod.Spec.TerminationGracePeriodSeconds 不应该指定为 0，设置为 0 对 StatefulSet 的 Pod 是极其不安全的做法，优雅地删除 StatefulSet 的 Pod 是非常有必要的，而且是安全的，因为它可以确保在 Kubelet 从 APIServer 删除之前，让 Pod 正常关闭。

当创建上面的 Nginx 实例时，Pod 将按 web-0、web-1、web-2 的顺序部署 3 个 Pod。在 web-0 处于 Running 或者 Ready 之前，web-1 不会被部署，相同的，web-2 在 web-1 未处于 Running 和 Ready 之前也不会被部署。如果在 web-1 处于 Running 和 Ready 状态时，web-0 变成 Failed（失败）状态，那么 web-2 将不会被启动，直到 web-0 恢复为 Running 和 Ready 状态。

如果用户将 StatefulSet 的 replicas 设置为 1，那么 web-2 将首先被终止，在完全关闭并删除 web-2 之前，不会删除 web-1。如果 web-2 终止并且完全关闭后，web-0 突然失败，那么在 web-0 未恢复成 Running 或者 Ready 时，web-1 不会被删除。

7. StatefulSet 扩容和缩容

和 Deployment 类似，可以通过更新 replicas 字段扩容/缩容 StatefulSet，也可以使用 kubectlscale 或者 kubectlpatch 来扩容/缩容一个 StatefulSet。

（1）扩容

将上述创建的 sts 副本增加到 5 个（扩容之前必须保证有创建完成的静态 PV，动态 PV 和 emptyDir）：

```
[root@K8S-master01 2.2.7]# kubectl scale sts web --replicas=5
statefulset.apps/web scaled
```

查看 Pod 及 PVC 的状态：

```
[root@K8S-master01 2.2.7]# kubectl get pvc
NAME       STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS
```

```
AGE
    www-web-0   Bound    pv-nginx-0   1Gi       RWO          nginx-storage-class
2m54s
    www-web-1   Bound    pv-nginx-2   1Gi       RWO          nginx-storage-class
2m44s
    www-web-2   Bound    pv-nginx-5   1Gi       RWO          nginx-storage-class
112s
    www-web-3   Bound    pv-nginx-1   1Gi       RWO          nginx-storage-class
75s
    www-web-4   Bound    pv-nginx-3   1Gi       RWO          nginx-storage-class
49s
    [root@K8S-master01 2.2.7]# kubectl get po
    NAME    READY   STATUS    RESTARTS   AGE
    web-0   1/1     Running   0          2m58s
    web-1   1/1     Running   0          2m48s
    web-2   1/1     Running   0          116s
    web-3   1/1     Running   0          79s
    web-4   1/1     Running   0          53s
```

也可使用以下命令动态查看：

```
    kubectl get pods -w -l app=nginx
```

（2）缩容

在一个终端动态查看：

```
    [root@K8S-master01 2.2.7]# kubectl get pods -w -l app=nginx
    NAME    READY   STATUS    RESTARTS   AGE
    web-0   1/1     Running   0          4m37s
    web-1   1/1     Running   0          4m27s
    web-2   1/1     Running   0          3m35s
    web-3   1/1     Running   0          2m58s
    web-4   1/1     Running   0          2m32s
```

在另一个终端将副本数改为 3：

```
    [root@K8S-master01 ~]# kubectl patch sts web -p '{"spec":{"replicas":3}}'
    statefulset.apps/web patched
```

此时可以看到第一个终端显示 web-4 和 web-3 的 Pod 正在被删除（或终止）：

```
    [root@K8S-master01 2.2.7]# kubectl get pods -w -l app=nginx
    NAME    READY   STATUS    RESTARTS   AGE
    web-0   1/1     Running   0          4m37s
    web-1   1/1     Running   0          4m27s
    web-2   1/1     Running   0          3m35s
    web-3   1/1     Running   0          2m58s
    web-4   1/1     Running   0          2m32s
    web-0   1/1   Running   0    5m8s
    web-0   1/1   Running   0    5m11s
    web-4   1/1   Terminating   0    3m36s
    web-4   0/1   Terminating   0    3m38s
    web-4   0/1   Terminating   0    3m47s
    web-4   0/1   Terminating   0    3m47s
    web-3   1/1   Terminating   0    4m13s
    web-3   0/1   Terminating   0    4m14s
```

```
    web-3   0/1   Terminating   0     4m22s
    web-3   0/1   Terminating   0     4m22s
```

查看状态，此时 PV 和 PVC 不会被删除：

```
    [root@K8S-master01 2.2.7]# kubectl get po
    NAME    READY   STATUS    RESTARTS   AGE
    web-0   1/1     Running   0          7m11s
    web-1   1/1     Running   0          7m1s
    web-2   1/1     Running   0          6m9s
    [root@K8S-master01 2.2.7]# kubectl get pvc
    NAME        STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS
AGE
    www-web-0   Bound    pv-nginx-0   1Gi        RWO           nginx-storage-class
7m15s
    www-web-1   Bound    pv-nginx-2   1Gi        RWO           nginx-storage-class
7m5s
    www-web-2   Bound    pv-nginx-5   1Gi        RWO           nginx-storage-class
6m13s
    www-web-3   Bound    pv-nginx-1   1Gi        RWO           nginx-storage-class
5m36s
    www-web-4   Bound    pv-nginx-3   1Gi        RWO           nginx-storage-class
5m10s
    [root@K8S-master01 2.2.7]# kubectl get pv
    NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
STORAGECLASS         REASON   AGE
    pv-nginx-0   1Gi        RWO           Recycle          Bound
default/www-web-0   nginx-storage-class           78m
    pv-nginx-1   1Gi        RWO           Recycle          Bound
default/www-web-3   nginx-storage-class           78m
    pv-nginx-2   1Gi        RWO           Recycle          Bound
default/www-web-1   nginx-storage-class           78m
    pv-nginx-3   1Gi        RWO           Recycle          Bound
default/www-web-4   nginx-storage-class           78m
    pv-nginx-4   1Gi        RWO           Recycle          Available
nginx-storage-class           78m
    pv-nginx-5   1Gi        RWO           Recycle          Bound
default/www-web-2   nginx-storage-class           78m
```

8. 更新策略

在 Kubernetes 1.7 以上的版本中，StatefulSet 的.spec.updateStrategy 字段允许配置和禁用容器的自动滚动更新、标签、资源限制以及 StatefulSet 中 Pod 的注释等。

（1）On Delete 策略

OnDelete 更新策略实现了传统（1.7 版本之前）的行为，它也是默认的更新策略。当我们选择这个更新策略并修改 StatefulSet 的.spec.template 字段时，StatefulSet 控制器不会自动更新 Pod，我们必须手动删除 Pod 才能使控制器创建新的 Pod。

（2）RollingUpdate 策略

RollingUpdate（滚动更新）更新策略会更新一个 StatefulSet 中所有的 Pod，采用与序号索引相反的顺序进行滚动更新。

比如 Patch 一个名称为 web 的 StatefulSet 来执行 RollingUpdate 更新：

```
    [root@K8S-master01 2.2.7]# kubectl patch statefulset web -p
'{"spec":{"updateStrategy":{"type":"RollingUpdate"}}}'
    statefulset.apps/web patched
```

查看更改后的 StatefulSet：

```
    [root@K8S-master01 2.2.7]# kubectl get sts web -o yaml | grep -A 1
"updateStrategy"
     updateStrategy:
       type: RollingUpdate
```

然后改变容器的镜像进行滚动更新：

```
    [root@K8S-master01 2.2.7]# kubectl patch statefulset web --type='json'
-p='[{"op": "replace", "path": "/spec/template/spec/containers/0/image",
"value":"dotbalo/canary:v1"}]'
    statefulset.apps/web patched
```

如上所述，StatefulSet 里的 Pod 采用和序号相反的顺序更新。在更新下一个 Pod 前，StatefulSet 控制器会终止每一个 Pod 并等待它们变成 Running 和 Ready 状态。在当前顺序变成 Running 和 Ready 状态之前，StatefulSet 控制器不会更新下一个 Pod，但它仍然会重建任何在更新过程中发生故障的 Pod，使用它们当前的版本。已经接收到请求的 Pod 将会被恢复为更新的版本，没有收到请求的 Pod 则会被恢复为之前的版本。

在更新过程中可以使用 kubectl rollout status sts/<name> 来查看滚动更新的状态：

```
    [root@K8S-master01 2.2.7]# kubectl rollout status sts/web
    Waiting for 1 pods to be ready...
    waiting for statefulset rolling update to complete 1 pods at revision
web-56b5798f76...
    Waiting for 1 pods to be ready...
    Waiting for 1 pods to be ready...
    waiting for statefulset rolling update to complete 2 pods at revision
web-56b5798f76...
    Waiting for 1 pods to be ready...
    Waiting for 1 pods to be ready...
    statefulset rolling update complete 3 pods at revision web-56b5798f76...
```

查看更新后的镜像：

```
    [root@K8S-master01 2.2.7]# for p in 0 1 2; do kubectl get po web-$p --template
'{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
    dotbalo/canary:v1
    dotbalo/canary:v1
    dotbalo/canary:v1
```

（3）分段更新

StatefulSet 可以使用 RollingUpdate 更新策略的 partition 参数来分段更新一个 StatefulSet。分段更新将会使 StatefulSet 中其余的所有 Pod（序号小于分区）保持当前版本，只更新序号大于等于分区的 Pod，利用此特性可以简单实现金丝雀发布（灰度发布）或者分阶段推出新功能等。注：金丝雀发布是指在黑与白之间能够平滑过渡的一种发布方式。

比如我们定义一个分区"partition":3，可以使用 patch 直接对 StatefulSet 进行设置：

```
    # kubectl patch statefulset web -p
```

```
'{"spec":{"updateStrategy":{"type":"RollingUpdate","rollingUpdate":{"partition
":3}}}}'
    statefulset "web" patched
```

然后再次 patch 改变容器的镜像：

```
    # kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path":
"/spec/template/spec/containers/0/image",
"value":"K8S.gcr.io/nginx-slim:0.7"}]'
    statefulset "web" patched
```

删除 Pod 触发更新：

```
    kubectl delete po web-2
    pod "web-2" deleted
```

此时，因为 Podweb-2 的序号小于分区 3，所以 Pod 不会被更新，还是会使用以前的容器恢复 Pod。

将分区改为 2，此时会自动更新 web-2（因为之前更改了更新策略），但是不会更新 web-0 和 web-1：

```
    # kubectl patch statefulset web -p
'{"spec":{"updateStrategy":{"type":"RollingUpdate","rollingUpdate":{"partition
":2}}}}'
    statefulset "web" patched
```

按照上述方式，可以实现分阶段更新，类似于灰度/金丝雀发布。查看最终的结果如下：

```
    [root@K8S-master01 2.2.7]# for p in 0 1 2; do kubectl get po web-$p --template
'{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
    dotbalo/canary:v1
    dotbalo/canary:v1
    dotbalo/canary:v2
```

9. 删除 StatefulSet

删除 StatefulSet 有两种方式，即级联删除和非级联删除。使用非级联方式删除 StatefulSet 时，StatefulSet 的 Pod 不会被删除；使用级联删除时，StatefulSet 和它的 Pod 都会被删除。

（1）非级联删除

使用 kubectldeletestsxxx 删除 StatefulSet 时，只需提供--cascade=false 参数，就会采用非级联删除，此时删除 StatefulSet 不会删除它的 Pod：

```
    [root@K8S-master01 2.2.7]# kubectl get po
    NAME    READY   STATUS    RESTARTS   AGE
    web-0   1/1     Running   0          16m
    web-1   1/1     Running   0          16m
    web-2   1/1     Running   0          11m
    You have new mail in /var/spool/mail/root
    [root@K8S-master01 2.2.7]# kubectl delete statefulset web --cascade=false
    statefulset.apps "web" deleted
    [root@K8S-master01 2.2.7]# kubectl get sts
    No resources found.
    [root@K8S-master01 2.2.7]# kubectl get po
    NAME    READY   STATUS    RESTARTS   AGE
```

```
web-0   1/1     Running   0          16m
web-1   1/1     Running   0          16m
web-2   1/1     Running   0          11m
```

由于此时删除了 StatefulSet，因此单独删除 Pod 时，不会被重建：

```
[root@K8S-master01 2.2.7]# kubectl get po
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          16m
web-1   1/1     Running   0          16m
web-2   1/1     Running   0          11m
[root@K8S-master01 2.2.7]# kubectl delete po web-0
pod "web-0" deleted
[root@K8S-master01 2.2.7]# kubectl get po
NAME    READY   STATUS    RESTARTS   AGE
web-1   1/1     Running   0          18m
web-2   1/1     Running   0          12m
```

当再次创建此 StatefulSet 时，web-0 会被重新创建，web-1 由于已经存在而不会被再次创建，因为最初此 StatefulSet 的 replicas 是 2，所以 web-2 会被删除，如下（忽略 AlreadyExists 错误）：

```
[root@K8S-master01 2.2.7]# kubectl create -f sts-web.yaml
statefulset.apps/web created
Error from server (AlreadyExists): error when creating "sts-web.yaml": services
"nginx" already exists
[root@K8S-master01 2.2.7]# kubectl get po
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          32s
web-1   1/1     Running   0          19m
```

（2）级联删除

省略--cascade=false 参数即为级联删除：

```
[root@K8S-master01 2.2.7]# kubectl delete statefulset web
statefulset.apps "web" deleted
[root@K8S-master01 2.2.7]# kubectl get po
No resources found.
```

也可以使用-f 参数直接删除 StatefulSet 和 Service（此文件将 sts 和 svc 写在了一起）：

```
[root@K8S-master01 2.2.7]# kubectl delete -f sts-web.yaml
service "nginx" deleted
Error from server (NotFound): error when deleting "sts-web.yaml":
statefulsets.apps "web" not found
[root@K8S-master01 2.2.7]#
```

2.2.8　DaemonSet

DaemonSet（守护进程集）和守护进程类似，它在符合匹配条件的节点上均部署一个 Pod。

1. 什么是 DaemonSet

DaemonSet 确保全部（或者某些）节点上运行一个 Pod 副本。当有新节点加入集群时，也会为它们新增一个 Pod。当节点从集群中移除时，这些 Pod 也会被回收，删除 DaemonSet 将会删除

它创建的所有 Pod。

使用 DaemonSet 的一些典型用法：

- 运行集群存储 daemon（守护进程），例如在每个节点上运行 Glusterd、Ceph 等。
- 在每个节点运行日志收集 daemon，例如 Fluentd、Logstash。
- 在每个节点运行监控 daemon，比如 Prometheus Node Exporter、Collectd、Datadog 代理、New Relic 代理或 Ganglia gmond。

2. 编写 DaemonSet 规范

创建一个 DaemonSet 的内容大致如下，比如创建一个 fluentd 的 DaemonSet：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-es-v2.0.4
  namespace: logging
  labels:
    K8S-app: fluentd-es
    version: v2.0.4
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
spec:
  selector:
    matchLabels:
      K8S-app: fluentd-es
      version: v2.0.4
  template:
    metadata:
      labels:
        K8S-app: fluentd-es
        kubernetes.io/cluster-service: "true"
        version: v2.0.4
      # This annotation ensures that fluentd does not get evicted if the node
      # supports critical pod annotation based priority scheme.
      # Note that this does not guarantee admission on the nodes (#40573).
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ''
        seccomp.security.alpha.kubernetes.io/pod: 'docker/default'
    spec:
      serviceAccountName: fluentd-es
      containers:
      - name: fluentd-es
        image: K8S.gcr.io/fluentd-elasticsearch:v2.0.4
        env:
        - name: FLUENTD_ARGS
          value: --no-supervisor -q
        resources:
          limits:
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
```

```
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
        - name: config-volume
          mountPath: /etc/fluent/config.d
      nodeSelector:
        beta.kubernetes.io/fluentd-ds-ready: "true"
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
      - name: config-volume
        configMap:
          name: fluentd-es-config-v0.1.4
```

（1）必需字段

和其他所有 Kubernetes 配置一样，DaemonSet 需要 apiVersion、kind 和 metadata 字段，同时也需要一个.spec 配置段。

（2）Pod 模板

.spec 唯一需要的字段是.spec.template。.spec.template 是一个 Pod 模板，它与 Pod 具有相同的配置方式，但它不具有 apiVersion 和 kind 字段。

除了 Pod 必需的字段外，在 DaemonSet 中的 Pod 模板必须指定合理的标签。

在 DaemonSet 中的 Pod 模板必须具有一个 RestartPolicy，默认为 Always。

（3）Pod Selector

.spec.selector 字段表示 Pod Selector，它与其他资源的.spec.selector 的作用相同。

.spec.selector 表示一个对象，它由如下两个字段组成：

- matchLabels，与 ReplicationController 的.spec.selector 的作用相同，用于匹配符合条件的 Pod。
- matchExpressions，允许构建更加复杂的 Selector，可以通过指定 key、value 列表以及与 key 和 value 列表相关的操作符。

如果上述两个字段都指定时，结果表示的是 AND 关系（逻辑与的关系）。

.spec.selector 必须与.spec.template.metadata.labels 相匹配。如果没有指定，默认是等价的，如果它们的配置不匹配，则会被 API 拒绝。

（4）指定节点部署 Pod

如果指定了.spec.template.spec.nodeSelector，DaemonSet Controller 将在与 Node Selector（节点选择器）匹配的节点上创建 Pod，比如部署在磁盘类型为 ssd 的节点上（需要提前给节点定义标签 Label）：

```
    containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
      nodeSelector:
        disktype: ssd
```

提 示

Node Selector 同样适用于其他 Controller。

3. 创建 DaemonSet

在生产环境中，公司业务的应用程序一般无须使用 DaemonSet 部署，一般情况下只有像 Fluentd（日志收集）、Ingress（集群服务入口）、Calico（集群网络组件）、Node-Exporter（监控数据采集）等才需要使用 DaemonSet 部署到每个节点。本节只演示 DaemonSet 的使用。

比如创建一个 nginxingress：

```
    [root@K8S-master01 2.2.8]# pwd
    /root/chap02/2.2.8
    [root@K8S-master01 2.2.8]# kubectl create -f nginx-ds.yaml
    namespace/ingress-nginx created
    configmap/nginx-configuration created
    configmap/tcp-services created
    configmap/udp-services created
    serviceaccount/nginx-ingress-serviceaccount created
    clusterrole.rbac.authorization.K8S.io/nginx-ingress-clusterrole created
    role.rbac.authorization.K8S.io/nginx-ingress-role created
    rolebinding.rbac.authorization.K8S.io/nginx-ingress-role-nisa-binding
created
    clusterrolebinding.rbac.authorization.K8S.io/nginx-ingress-clusterrole-nis
a-binding created
    daemonset.extensions/nginx-ingress-controller created
```

此时会在每个节点创建一个 Pod：

```
    [root@K8S-master01 2.2.8]# kubectl get po -n ingress-nginx
    NAME                             READY   STATUS    RESTARTS   AGE
    nginx-ingress-controller-fjkf2   1/1     Running   0          44s
    nginx-ingress-controller-gfmcv   1/1     Running   0          44s
    nginx-ingress-controller-j89qc   1/1     Running   0          44s
    nginx-ingress-controller-sqsk2   1/1     Running   0          44s
    nginx-ingress-controller-tgdt6   1/1     Running   0          44s
    [root@K8S-master01 2.2.8]# kubectl get po -n ingress-nginx -o wide
    NAME                             READY   STATUS    RESTARTS   AGE
    IP            NODE          NOMINATED NODE
    nginx-ingress-controller-fjkf2   1/1     Running   0          50s
192.168.20.30   K8S-node01    <none>
    nginx-ingress-controller-gfmcv   1/1     Running   0          50s
192.168.20.21   K8S-master02   <none>
    nginx-ingress-controller-j89qc   1/1     Running   0          50s
192.168.20.22   K8S-master03   <none>
    nginx-ingress-controller-sqsk2   1/1     Running   0          50s
192.168.20.31   K8S-node02    <none>
    nginx-ingress-controller-tgdt6   1/1     Running   0          50s
```

```
192.168.20.20   K8S-master01   <none>
```

注　意

因为笔者的 Master 节点删除了 Taint（Taint 和 Toleration 见 2.2.18），所以也能部署 Ingress 或者其他 Pod，在生产环境下，在 Master 节点最好除了系统组件外不要部署其他 Pod。

4. 更新和回滚 DaemonSet

如果修改了节点标签（Label），DaemonSet 将立刻向新匹配上的节点添加 Pod，同时删除不能匹配的节点上的 Pod。

在 Kubernetes 1.6 以后的版本中，可以在 DaemonSet 上执行滚动更新，未来的 Kubernetes 版本将支持节点的可控更新。

DaemonSet 滚动更新可参考：https://kubernetes.io/docs/tasks/manage-daemon/update-daemon-set/。

DaemonSet 更新策略和 StatefulSet 类似，也有 OnDelete 和 RollingUpdate 两种方式。

查看上一节创建的 DaemonSet 更新方式：

```
    [root@K8S-master01 2.2.8]# kubectl get ds/nginx-ds -o
go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
    RollingUpdate
```

提　示

如果是其他 DaemonSet，请确保更新策略是 RollingUpdate（滚动更新）。

（1）命令式更新

```
    kubectl edit ds/<daemonset-name>
    kubectl patch ds/<daemonset-name> -p=<strategic-merge-patch>
```

（2）更新镜像

```
    kubectl set image
ds/<daemonset-name><container-name>=<container-new-image>--record=true
```

（3）查看更新状态

```
    kubectl rollout status ds/<daemonset-name>
```

（4）列出所有修订版本

```
    kubectl rollout history daemonset <daemonset-name>
```

（5）回滚到指定 revision

```
    kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>
```

DaemonSet 的更新和回滚与 Deployment 类似，此处不再演示。

2.2.9　ConfigMap

一般用 ConfigMap 管理一些程序的配置文件或者 Pod 变量，比如 Nginx 配置、MavenSetting

配置文件等。

1. 什么是 ConfigMap

ConfigMap 是一个将配置文件、命令行参数、环境变量、端口号和其他配置绑定到 Pod 的容器和系统组件。ConfigMaps 允许将配置与 Pod 和组件分开，这有助于保持工作负载的可移植性，使配置更易于更改和管理。比如在生产环境中，可以将 Nginx、Redis 等应用的配置文件存储在 ConfigMap 上，然后将其挂载即可使用。

相对于 Secret，ConfigMap 更倾向于存储和共享非敏感、未加密的配置信息，如果要在集群中使用敏感信息，最好使用 Secret。

2. 创建 ConfigMap

可以使用 kubectl create configmap 命令从目录、文件或字符值创建 ConfigMap：

```
kubectl create configmap <map-name><data-source>
```

说明：

- map-name，ConfigMap 的名称。
- data-source，数据源，数据的目录、文件或字符值。

数据源对应于 ConfigMap 中的键-值对（key-value pair），其中，

- key：文件名或密钥。
- value：文件内容或字符值。

（1）从目录创建 ConfigMap

可以使用 kubectl create configmap 命令从同一个目录中的多个文件创建 ConfigMap。

创建一个配置文件目录并且下载两个文件作为测试配置文件：

```
mkdir -p configure-pod-container/configmap/kubectl/

wget
https://K8S.io/docs/tasks/configure-pod-container/configmap/kubectl/game.prope
rties -O configure-pod-container/configmap/kubectl/game.properties

wget
https://K8S.io/docs/tasks/configure-pod-container/configmap/kubectl/ui.propert
ies -O configure-pod-container/configmap/kubectl/ui.properties
```

创建 ConfigMap，默认创建在 default 命名空间下，可以使用-n 更改 NameSpace（命名空间）：

```
[root@K8S-master01 ~]# kubectl create configmap game-config
--from-file=configure-pod-container/configmap/kubectl/
configmap/game-config created
```

查看当前的 ConfigMap：

```
[root@K8S-master01 ~]# kubectl describe configmaps game-config
Name:         game-config
Namespace:    default
Labels:       <none>
```

```
    Annotations:  <none>

    Data
    ====
    game.properties:
    ----
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
    ui.properties:
    ----
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice

    Events:  <none>
```

可以看到，ConfigMap 的内容与测试的配置文件内容一致：

```
    [root@K8S-master01 ~]# cat
configure-pod-container/configmap/kubectl/game.properties
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30[root@K8S-master01 ~]# cat
configure-pod-container/configmap/kubectl/ui.properties
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
```

（2）从文件创建 ConfigMap

可以使用 kubectl create configmap 命令从单个文件或多个文件创建 ConfigMap。

例如以 configure-pod-container/configmap/kubectl/game.properties 文件建立 ConfigMap：

```
    [root@K8S-master01 ~]# kubectl create configmap game-config-2
--from-file=configure-pod-container/configmap/kubectl/game.properties
    configmap/game-config-2 created
```

查看当前的 ConfigMap：

```
    [root@K8S-master01 ~]# kubectl get cm game-config-2
    NAME          DATA   AGE
    game-config-2   1     38s
```

也可以使用--from-file 多次传入参数以从多个数据源创建 ConfigMap：

```
    [root@K8S-master01 ~]# kubectl create configmap game-config-3
```

```
--from-file=configure-pod-container/configmap/kubectl/game.properties
--from-file=configure-pod-container/configmap/kubectl/ui.properties
    configmap/game-config-3 created
```

查看当前的 ConfigMap：

```
    [root@K8S-master01 ~]# kubectl get cm game-config-3 -oyaml
    apiVersion: v1
    data:
      game.properties: |-
        enemies=aliens
        lives=3
        enemies.cheat=true
        enemies.cheat.level=noGoodRotten
        secret.code.passphrase=UUDDLRLRBABAS
        secret.code.allowed=true
        secret.code.lives=30
      ui.properties: |
        color.good=purple
        color.bad=yellow
        allow.textmode=true
        how.nice.to.look=fairlyNice
    kind: ConfigMap
    metadata:
      creationTimestamp: 2019-02-11T08:33:34Z
      name: game-config-3
      namespace: default
      resourceVersion: "4266928"
      selfLink: /api/v1/namespaces/default/configmaps/game-config-3
      uid: b88eea8b-2dd7-11e9-9180-000c293ad492
```

（3）从 ENV 文件创建 ConfigMap

可以使用--from-env-file 从 ENV 文件创建 ConfigMap。

首先创建/下载一个测试文件，文件内容为 key=value 的格式：

```
    [root@K8S-master01 ~]# wget
https://K8S.io/docs/tasks/configure-pod-container/configmap/kubectl/game-env-f
ile.properties -O
configure-pod-container/configmap/kubectl/game-env-file.properties

    [root@K8S-master01 ~]# cat
configure-pod-container/configmap/kubectl/game-env-file.properties
    enemies=aliens
    lives=3
    allowed="true"

    # This comment and the empty line above it are ignored
```

创建 ConfigMap：

```
    [root@K8S-master01 ~]# kubectl create configmap game-config-env-file \

--from-env-file=configure-pod-container/configmap/kubectl/game-env-file.proper
ties
    configmap/game-config-env-file created
```

查看当前的 ConfigMap：

```
[root@K8S-master01 ~]# kubectl get configmap game-config-env-file -o yaml
apiVersion: v1
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
kind: ConfigMap
metadata:
  creationTimestamp: 2019-02-11T08:40:17Z
  name: game-config-env-file
  namespace: default
  resourceVersion: "4267912"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file
  uid: a84ccd32-2dd8-11e9-90e9-000c293bfe27
```

注　意

如果使用--from-env-file 多次传递参数以从多个数据源创建 ConfigMap 时，仅最后一个 ENV 生效。

（4）自定义 data 文件名创建 ConfigMap

可以使用以下命令自定义文件名：

```
kubectl create configmap game-config-3
--from-file=<my-key-name>=<path-to-file>
```

比如将 game.properties 文件定义为 game-special-key：

```
[root@K8S-master01 ~]# kubectl create configmap game-config-4
--from-file=game-special-key=configure-pod-container/configmap/kubectl/game.pr
operties
configmap/game-config-4 created
[root@K8S-master01 ~]# kubectl get configmaps game-config-4 -o yaml
apiVersion: v1
data:
  game-special-key: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2019-02-11T08:46:08Z
  name: game-config-4
  namespace: default
  resourceVersion: "4268642"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-4
  uid: 797d269a-2dd9-11e9-90e9-000c293bfe27
```

（5）从字符值创建 ConfigMaps

可以使用 kubectl create configmap 与--from-literal 参数来定义命令行的字符值：

```
    [root@K8S-master01 ~]# kubectl create configmap special-config
--from-literal=special.how=very --from-literal=special.type=charm
    configmap/special-config created
    [root@K8S-master01 ~]# kubectl get cm special-config -o yaml
    apiVersion: v1
    data:
      special.how: very
      special.type: charm
    kind: ConfigMap
    metadata:
      creationTimestamp: 2019-02-11T08:49:28Z
      name: special-config
      namespace: default
      resourceVersion: "4269314"
      selfLink: /api/v1/namespaces/default/configmaps/special-config
      uid: f0dbb926-2dd9-11e9-8f6f-000c298bf023
```

3. ConfigMap 实践

本节主要讲解 ConfigMap 的一些常见使用方法，比如通过单个 ConfigMap 定义环境变量、通过多个 ConfigMap 定义环境变量和将 ConfigMap 作为卷使用等。

（1）使用单个 ConfigMap 定义容器环境变量

首先在 ConfigMap 中将环境变量定义为键-值对（key-value pair）：

```
    kubectl create configmap special-config --from-literal=special.how=very
```

然后，将 ConfigMap 中定义的值 special.how 分配给 Pod 的环境变量 SPECIAL_LEVEL_KEY：

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: dapi-test-pod
    spec:
      containers:
        - name: test-container
          image: busybox
          command: [ "/bin/sh", "-c", "env" ]
          env:
            # Define the environment variable
    - name: SPECIAL_LEVEL_KEY
            valueFrom:
              configMapKeyRef:
                # The ConfigMap containing the value you want to assign to
SPECIAL_LEVEL_KEY
    name: special-config
                # Specify the key associated with the value
    key: special.how
      restartPolicy: Never
```

（2）使用多个 ConfigMap 定义容器环境变量

首先定义两个或多个 ConfigMap：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

然后，在 Pod 中引用 ConfigMap：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
- name: SPECIAL_LEVEL_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
       - name: LOG_LEVEL
        valueFrom:
          configMapKeyRef:
            name: env-config
            key: log_level
  restartPolicy: Never
```

（3）将 ConfigMap 中所有的键-值对配置为容器的环境变量

创建含有多个键-值对的 ConfigMap：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

使用 envFrom 将 ConfigMap 所有的键-值对作为容器的环境变量，其中 ConfigMap 中的键作为 Pod 中的环境变量的名称：

```
apiVersion: v1
kind: Pod
```

```
  metadata:
    name: dapi-test-pod
  spec:
    containers:
      - name: test-container
        image: busybox
        command: [ "/bin/sh", "-c", "env" ]
   envFrom:
       - configMapRef:
           name: special-config
    restartPolicy: Never
```

（4）将 ConfigMap 添加到卷

大部分情况下，ConfigMap 定义的都是配置文件，不是环境变量，因此需要将 ConfigMap 中的文件（一般为--from-file 创建）挂载到 Pod 中，然后 Pod 中的容器就可引用，此时可以通过 volume 进行挂载。

例如，将名称为 special-config 的 ConfigMap，挂载到容器的/etc/config/目录下：

```
  apiVersion: v1
  kind: Pod
  metadata:
    name: dapi-test-pod
  spec:
    containers:
      - name: test-container
        image: busybox
        command: [ "/bin/sh", "-c", "ls /etc/config/" ]
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
    volumes:
      - name: config-volume
        configMap:
          # Provide the name of the ConfigMap containing the files you want
          # to add to the container
          name: special-config
    restartPolicy: Never
```

此时 Pod 运行，会执行 command 的命令，即执行 ls /etc/config/

```
  special.level
  special.type
```

注　意
/etc/config/会被覆盖。

（5）将 ConfigMap 添加到卷并指定文件名

使用 path 字段可以指定 ConfigMap 挂载的文件名，比如将 special.level 挂载到/etc/config，并指定名称为 keys：

```
  apiVersion: v1
  kind: Pod
```

```
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh","-c","cat /etc/config/keys" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.level
          path: keys
  restartPolicy: Never
```

此时启动 Pod 时会打印：very

（6）指定特定路径和文件权限

方式和 Secret 类似，可参考 2.2.10.4 节的内容。

4. ConfigMap 限制

（1）必须先创建 ConfigMap 才能在 Pod 中引用它，如果 Pod 引用的 ConfigMap 不存在，Pod 将无法启动。

（2）Pod 引用的键必须存在于 ConfigMap 中，否则 Pod 无法启动。

（3）使用 envFrom 配置容器环境变量时，默认会跳过被视为无效的键，但是不影响 Pod 启动，无效的变量会记录在事件日志中，如下：

```
    kubectl get events
      LASTSEEN FIRSTSEEN COUNT NAME         KIND  SUBOBJECT  TYPE
REASON                        SOURCE              MESSAGE
      0s      0s       1    dapi-test-pod Pod            Warning
InvalidEnvironmentVariableNames   {kubelet, 127.0.0.1}  Keys [1badkey,
2alsobad] from the EnvFrom configMap default/myconfig were skipped since they
are considered invalid environment variable names.
```

（4）ConfigMap 和引用它的 Pod 需要在同一个命名空间。

2.2.10　Secret

Secret 对象类型用来保存敏感信息，例如密码、令牌和 SSH Key，将这些信息放在 Secret 中比较安全和灵活。用户可以创建 Secret 并且引用到 Pod 中，比如使用 Secret 初始化 Redis、MySQL 等密码。

1. 创建 Secret

创建 Secret 的方式有很多，比如使用命令行 Kubelet 或者使用 Yaml/Json 文件创建等。

（1）使用 Kubectl 创建 Secret

假设有些 Pod 需要访问数据库，可以将账户密码存储在 username.txt 和 password.txt 文件里，然后以文件的形式创建 Secret 供 Pod 使用。

创建账户信息文件：

```
[root@K8S-master01 ~]# echo -n "admin" > ./username.txt
[root@K8S-master01 ~]# echo -n "1f2d1e2e67df" > ./password.txt
```

以文件 username.txt 和 password.txt 创建 Secret：

```
[root@K8S-master01 ~]# kubectl create secret generic db-user-pass
--from-file=./username.txt --from-file=./password.txt
secret/db-user-pass created
```

查看 Secret：

```
[root@K8S-master01 ~]# kubectl get secrets db-user-pass
NAME          TYPE    DATA   AGE
db-user-pass   Opaque   2     33s
[root@K8S-master01 ~]# kubectl describe secrets/db-user-pass
Name:        db-user-pass
Namespace:    default
Labels:      <none>
Annotations:  <none>

Type:  Opaque

Data
====
password.txt:  12 bytes
username.txt:  0 bytes
```

默认情况下，get 和 describe 命令都不会显示文件的内容，这是为了防止 Secret 中的内容被意外暴露。可以参考 2.2.10.2 一节的方式解码 Secret。

（2）手动创建 Secret

手动创建 Secret，因为每一项内容必须是 base64 编码，所以要先对其进行编码：

```
[root@K8S-master01 ~]# echo -n "admin" | base64
YWRtaW4=
[root@K8S-master01 ~]# echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

然后，创建一个文件，内容如下：

```
[root@K8S-master01 ~]# cat db-user-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

最后，使用该文件创建一个 Secret：

```
[root@K8S-master01 ~]# kubectl create -f db-user-secret.yaml
secret/mysecret created
```

2. 解码 Secret

Secret 被创建后，会以加密的方式存储于 Kubernetes 集群中，可以对其进行解码获取内容。首先以 yaml 的形式获取刚才创建的 Secret：

```
[root@K8S-master01 ~]# kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2019-02-09T03:16:19Z
  name: mysecret
  namespace: default
  resourceVersion: "3811354"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 11e49e9f-2c19-11e9-8f6f-000c298bf023
type: Opaque
```

然后通过--decode 解码 Secret：

```
[root@K8S-master01 ~]# echo "MWYyZDFlMmU2N2Rm" | base64 --decode
1f2d1e2e67df
```

3. 使用 Secret

Secret 可以作为数据卷被挂载，或作为环境变量以供 Pod 的容器使用。

（1）在 Pod 中使用 Secret

在 Pod 中的 volume 里使用 Secret：

①首先创建一个 Secret 或者使用已有的 Secret，多个 Pod 可以引用同一个 Secret。

②在 spec.volumes 下增加一个 volume，命名随意，spec.volumes.secret.secretName 必须和 Secret 对象的名字相同，并且在同一个 Namespace 中。

③将 spec.containers.volumeMounts 加到需要用到该 Secret 的容器中，并且设置 spec.containers.volumeMounts.readOnly = true。

④使用 spec.containers.volumeMounts.mountPath 指定 Secret 挂载目录。

例如，将名字为 mysecret 的 Secret 挂载到 Pod 中的/etc/foo：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
```

```
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

用到的每个 Secret 都需要在 spec.volumes 中指明，如果 Pod 中有多个容器，每个容器都需要自己的 volumeMounts 配置块，但是每个 Secret 只需要一个 spec.volumes，可以根据自己的应用场景将多个文件打包到一个 Secret 中，或者使用多个 Secret。

（2）自定义文件名挂载

挂载 Secret 时，可以使用 spec.volumes.secret.items 字段修改每个 key 的目标路径，即控制 Secret Key 在容器中的映射路径。

比如：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

上述挂载方式，将 mysecret 中的 username 存储到了/etc/foo/my-group/my-username 文件中，而不是/etc/foo/username（不指定 items），由于 items 没有指定 password，因此 password 不会被挂载。如果使用了 spec.volumes.secret.items，只有在 items 中指定的 key 才会被挂载。

挂载的 Secret 在容器中作为文件，我们可以在 Pod 中查看挂载的文件内容：

```
$ ls /etc/foo/
username
password
$ cat /etc/foo/username
admin
$ cat /etc/foo/password
1f2d1e2e67df
```

（3）Secret 作为环境变量

Secret 可以作为环境变量使用，步骤如下：

①创建一个 Secret 或者使用一个已存在的 Secret，多个 Pod 可以引用同一个 Secret。

②为每个容器添加对应的 Secret Key 环境变量 env.valueFrom.secretKeyRef。

比如，定义 SECRET_USERNAME 和 SECRET_PASSWORD 两个环境变量，其值来自于名字为 mysecret 的 Secret：

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

挂载成功后，可以在容器中使用此变量：

```
$ echo $SECRET_USERNAME
admin
$ echo $SECRET_PASSWORD
1f2d1e2e67df
```

4. Secret 文件权限

Secret 默认挂载的文件的权限为 0644，可以通过 defaultMode 方式更改权限：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
```

```
        secretName: mysecret
        defaultMode: 256
```

更改的 Secret 挂载到/etc/foo 目录的文件权限为 0400。新版本可以直接指定 400。

5. imagePullSecret

在拉取私有镜像库中的镜像时，可能需要认证后才可拉取，此时可以使用 imagePullSecret 将包含 Docker 镜像注册表密码的 Secret 传递给 Kubelet，然后即可拉取私有镜像。

Kubernetes 支持在 Pod 中指定 Registry Key，用于拉取私有镜像仓库中的镜像。

首先创建一个镜像仓库账户信息的 Secret：

```
    $kubectl create secret docker-registry myregistrykey
--docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER
--docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

如果需要访问多个 Registry，则可以为每个注册表创建一个 Secret，在 Pods 拉取镜像时，Kubelet 会合并 imagePullSecrets 到.docker/config.json。注意 Secret 需要和 Pod 在同一个命名空间中。

创建完 imagePullSecrets 后，可以使用 imagePullSecrets 的方式引用该 Secret：

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: foo
      namespace: awesomeapps
    spec:
      containers:
        - name: foo
          image: janedoe/awesomeapp:v1
      imagePullSecrets:
        - name: myregistrykey
```

6. 使用案例

本节演示的是 Secret 的一些常用配置，比如配置 SSH 密钥、创建隐藏文件等。

（1）定义包含 SSH 密钥的 Pod

首先，创建一个包含 SSH Key 的 Secret：

```
    $kubectl create secret generic ssh-key-secret
--from-file=ssh-privatekey=/path/to/.ssh/id_rsa
--from-file=ssh-publickey=/path/to/.ssh/id_rsa.pub
```

然后将其挂载使用：

```
    kind: Pod
    apiVersion: v1
    metadata:
      name: secret-test-pod
      labels:
        name: secret-test
    spec:
      volumes:
      - name: secret-volume
        secret:
```

```
      secretName: ssh-key-secret
  containers:
  - name: ssh-test-container
    image: mySshImage
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

上述密钥会被挂载到/etc/secret-volume。注意，挂载 SSH Key 需要考虑安全性的问题。

（2）创建隐藏文件

为了将数据“隐藏”起来（即文件名以句点符号开头的文件），可以让 Key 以一个句点符号开始，比如定义一个以句点符号开头的 Secret：

```
kind: Secret
apiVersion: v1
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
```

挂载使用：

```
kind: Pod
apiVersion: v1
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
  - name: dotfile-test-container
    image: K8S.gcr.io/busybox
    command:
    - ls
    - "-l"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

此时会在/etc/secret-volume 下创建一个.secret-file 的文件。

2.2.11　HPA

1. 什么是 HPA

HPA（Horizontal Pod Autoscaler，水平 Pod 自动伸缩器）可根据观察到的 CPU、内存使用率或自定义度量标准来自动扩展或缩容 Pod 的数量。HPA 不适用于无法缩放的对象，比如 DaemonSet。

HPA 控制器会定期调整 RC 或 Deployment 的副本数，以使观察到的平均 CPU 利用率与用户指定的目标相匹配。

HPA 需要 metrics-server（项目地址：https://github.com/kubernetes-incubator/metrics-server）获取度量指标，由于在高可用集群安装中已经安装了 metrics-server，所以本节的实践部分无须再次安装。

2. HPA 实践

在生产环境中，总会有一些意想不到的事情发生，比如公司网站流量突然升高，此时之前创建的 Pod 已不足以撑住所有的访问，而运维人员也不可能 24 小时守着业务服务，这时就可以通过配置 HPA，实现负载过高的情况下自动扩容 Pod 副本数以分摊高并发的流量，当流量恢复正常后，HPA 会自动缩减 Pod 的数量。

本节将测试实现一个 Web 服务器的自动伸缩特性，具体步骤如下：

首先启动一个 Nginx 服务：

```
    [root@K8S-master01 ~]#kubectl run nginx-server --requests=cpu=10m
--image=nginx --port=80
    service/php-apache created
    deployment.apps/php-apache created
```

临时开启 nginx-server 的端口，实际使用时需要定义 service：

```
    kubectl expose deployment nginx-server --port=80
```

使用 kubectl autoscale 创建 HPA：

```
    [root@K8S-master01 ~]# kubectl autoscale deployment nginx-server
--cpu-percent=10 --min=1 --max=10
```

此 HPA 将根据 CPU 的使用率自动增加和减少副本数量，上述设置的是 CPU 使用率超过 10%（--cpu-percent 参数指定）即会增加 Pod 的数量，以保持所有 Pod 的平均 CPU 利用率为 10%，允许最大的 Pod 数量为 10（--max），最少的 Pod 数为 1（--min）。

查看当前 HPA 状态，因为未对其发送任何请求，所以当前 CPU 使用率为 0%：

```
    [root@K8S-master01 metric-server]# kubectl get hpa
    NAME          REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS
AGE
    nginx-server   Deployment/nginx-server   0%/10%    1         10        1
5m
```

查看当前 Nginx 的 Service 地址：

```
    [root@K8S-master01 ~]# kubectl get service -n default
    NAME          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
    kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP   1d
    nginx-server   ClusterIP   10.108.160.23   <none>        80/TCP    5m
```

增加负载：

```
    [root@K8S-master01 ~]# while true; do wget -q -O- http://10.108.160.23 >
/dev/null; done
```

1 分钟左右再次查看 HPA：

```
    [root@K8S-master01 metric-server]# kubectl get hpa
    NAME          REFERENCE               TARGETS    MINPODS   MAXPODS   REPLICAS
AGE
    nginx-server   Deployment/nginx-server   540%/10%   1         10        1
15m
```

再次查看 Pod，可以看到 nginx-server 的 Pod 已经在扩容阶段：

```
    [root@K8S-master01 metric-server]# kubectl get po
    NAME                          READY   STATUS              RESTARTS   AGE
    nginx-server-589c8db585-5cbxl   0/1     ContainerCreating   0
<invalid>
    nginx-server-589c8db585-7whl8   1/1     Running             0
<invalid>
    nginx-server-589c8db585-cv4hs   1/1     Running             0
<invalid>
    nginx-server-589c8db585-m5dn6   0/1     ContainerCreating   0
<invalid>
    nginx-server-589c8db585-sxbfm   1/1     Running             0          19m
    nginx-server-589c8db585-xbctd   0/1     ContainerCreating   0
<invalid>
    nginx-server-589c8db585-xffs9   1/1     Running             0
<invalid>
    nginx-server-589c8db585-xlb8s   0/1     ContainerCreating   0
<invalid>
```

在增加负荷的终端，按 Ctrl+C 键终止访问。

停止 1 分钟后再次查看 HPA 和 deployment，此时副本已经恢复为 1：

```
    [root@K8S-master01 metric-server]# kubectl get hpa
    NAME          REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS
AGE
    nginx-server   Deployment/nginx-server   0%/10%    1         10        10
20m
```

2.2.12　Storage

本节介绍 Kubernetes Storage 的相关概念与使用，一般做持久化或者有状态的应用程序才会用到 Storage。

1. Volumes

Container（容器）中的磁盘文件是短暂的，当容器崩溃时，kubelet 会重新启动容器，但最初的文件将丢失，Container 会以最干净的状态启动。另外，当一个 Pod 运行多个 Container 时，各个容器可能需要共享一些文件。Kubernetes Volume 可以解决这两个问题。

（1）背景

Docker 也有卷的概念，但是在 Docker 中卷只是磁盘上或另一个 Container 中的目录，其生命周期不受管理。虽然目前 Docker 已经提供了卷驱动程序，但是功能非常有限，例如从 Docker 1.7 版本开始，每个 Container 只允许一个卷驱动程序，并且无法将参数传递给卷。

另一方面，Kubernetes 卷具有明确的生命周期，与使用它的 Pod 相同。因此，在 Kubernetes

中的卷可以比 Pod 中运行的任何 Container 都长，并且可以在 Container 重启或者销毁之后保留数据。Kubernetes 支持多种类型的卷，Pod 可以同时使用任意数量的卷。

从本质上讲，卷只是一个目录，可能包含一些数据，Pod 中的容器可以访问它。要使用卷 Pod 需要通过.spec.volumes 字段指定为 Pod 提供的卷，以及使用.spec.containers.volumeMounts 字段指定卷挂载的目录。从容器中的进程可以看到由 Docker 镜像和卷组成的文件系统视图，卷无法挂载其他卷或具有到其他卷的硬链接，Pod 中的每个 Container 必须独立指定每个卷的挂载位置。

（2）卷的类型

Kubernetes 支持的卷的类型有很多，以下为常用的卷。

①awsElasticBlockStore（EBS）

awsElasticBlockStore 卷挂载一个 AWS EBS Volume 到 Pod 中，与 emptyDir 卷不同的是，当移除 Pod 时 EBS 卷的内容不会被删除，这意味着可以将数据预先放置在 EBS 卷中，并且可以在 Pod 之间切换该数据。

使用 awsElasticBlockStore 卷的限制：

- 运行 Pod 的节点必须是 AWS EC2 实例。
- AWS EC2 实例需要和 EBS 卷位于同一区域和可用区域。
- EBS 仅支持挂载卷的单个 EC2 实例。

在将 Pod 与 EBS 卷一起使用之前，需要先创建 EBS 卷，确保该卷的区域与集群的区域匹配，并检查 size 和 EBS 卷类型是否合理：

```
    aws ec2 create-volume --availability-zone=eu-west-1a --size=10
--volume-type=gp2
```

AWS EBS 示例配置：

```
    apiVersion: v1
    kind: Pod
    metadata:
      name: test-ebs
    spec:
      containers:
      - image: K8S.gcr.io/test-webserver
        name: test-container
        volumeMounts:
        - mountPath: /test-ebs
          name: test-volume
      volumes:
      - name: test-volume
        # This AWS EBS volume must already exist.
        awsElasticBlockStore:
          volumeID: <volume-id>
          fsType: ext4
```

②CephFS

CephFS 卷允许将一个已经存在的卷挂载到 Pod 中，和 emptyDir 卷不同的是，当移除 Pod 时，CephFS 卷的内容不会被删除，这意味着可以将数据预先放置在 CephFS 卷中，并且可以在 Pod 之

间切换该数据。CephFS 卷可以被多个写设备同时挂载。

和 AWS EBS 一样，需要先创建 CephFS 卷后才能使用它。

关于 CephFS 的更多内容，可以参考以下文档：

https://github.com/kubernetes/examples/tree/master/staging/volumes/cephfs/

③ConfigMap

ConfigMap 卷也可以作为 volume 使用，存储在 ConfigMap 中的数据可以通过 ConfigMap 类型的卷挂载到 Pod 中，然后使用该 ConfigMap 中的数据。引用 ConfigMap 对象时，只需要在 volume 中引用 ConfigMap 的名称即可，同时也可以自定义 ConfigMap 的挂载路径。

例如，将名称为 log-config 的 ConfigMap 挂载到 Pod 的/etc/config 目录下，挂载的文件名称为 path 指定的值，当前为 log_level：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level
```

注　意

ConfigMap 需要提前创建。

④emptyDir

和上述 volume 不同的是，如果删除 Pod，emptyDir 卷中的数据也将被删除，一般 emptyDir 卷用于 Pod 中的不同 Container 共享数据。它可以被挂载到相同或不同的路径上。

默认情况下，emptyDir 卷支持节点上的任何介质，可能是 SSD、磁盘或网络存储，具体取决于自身的环境。可以将 emptyDir.medium 字段设置为 Memory，让 Kubernetes 使用 tmpfs（内存支持的文件系统），虽然 tmpfs 非常快，但是 tmpfs 在节点重启时，数据同样会被清除，并且设置的大小会被计入到 Container 的内存限制当中。

使用 emptyDir 卷的示例，直接指定 emptyDir 为{}即可：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
```

```
  containers:
  - image: K8S.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

⑤GlusterFS

GlusterFS （以下简称为 GFS）是一个开源的网络文件系统，常被用于为 Kubernetes 提供动态存储，和 emptyDir 不同的是，删除 Pod 时 GFS 卷中的数据会被保留。

关于 GFS 的使用示例请参看 3.1 节。

⑥hostPath

hostPath 卷可将节点上的文件或目录挂载到 Pod 上，用于 Pod 自定义日志输出或访问 Docker 内部的容器等。

使用 hostPath 卷的示例。将主机的/data 目录挂载到 Pod 的/test-pd 目录：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: K8S.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

hostPath 卷常用的 type（类型）如下。

- type 为空字符串：默认选项，意味着挂载 hostPath 卷之前不会执行任何检查。
- DirectoryOrCreate：如果给定的 path 不存在任何东西，那么将根据需要创建一个权限为 0755 的空目录，和 Kubelet 具有相同的组和权限。
- Directory：目录必须存在于给定的路径下。
- FileOrCreate：如果给定的路径不存储任何内容，则会根据需要创建一个空文件，权限设置为 0644，和 Kubelet 具有相同的组和所有权。
- File：文件，必须存在于给定路径中。
- Socket：UNIX 套接字，必须存在于给定路径中。
- CharDevice：字符设备，必须存在于给定路径中。

- BlockDevice：块设备，必须存在于给定路径中。

⑦NFS

NFS 卷也是一种网络文件系统，同时也可以作为动态存储，和 GFS 类似，删除 Pod 时，NFS 中的数据不会被删除。NFS 可以被多个写入同时挂载。

关于 NFS 的使用，请参考第 3 章。

⑧persistentVolumeClaim

persistentVolumeClaim 卷用于将 PersistentVolume（持久化卷）挂载到容器中，PersistentVolume 分为动态存储和静态存储，静态存储的 PersistentVolume 需要手动提前创建 PV，动态存储无需手动创建 PV。

⑨Secret

Secret 卷和 ConfigMap 卷类似，详情见 2.2.10 节。

⑩SubPath

有时可能需要将一个卷挂载到不同的子目录，此时使用 volumeMounts.subPath 可以实现不同子目录的挂载。

本示例为一个 LAMP 共享一个卷，使用 subPath 卷挂载不同的目录：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "rootpasswd"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

更多 volume 可参考：

```
https://kubernetes.io/docs/concepts/storage/volumes/
```

2. PersistentVolume

管理计算资源需要关注的另一个问题是管理存储，PersistentVolume 子系统为用户和管理提供

了一个 API，用于抽象如何根据使用类型提供存储的详细信息。为此，Kubernetes 引入了两个新的 API 资源：PersistentVolume 和 PersistentVolumeClaim。

PersistentVolume（简称 PV）是由管理员设置的存储，它同样是集群中的一类资源，PV 是容量插件，如 Volumes（卷），但其生命周期独立使用 PV 的任何 Pod，PV 的创建可使用 NFS、iSCSI、GFS、CEPH 等。

PersistentVolumeClaim（简称 PVC）是用户对存储的请求，类似于 Pod，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），PVC 可以请求特定的大小和访问模式。例如，可以以一次读/写或只读多次的模式挂载。

虽然 PVC 允许用户使用抽象存储资源，但是用户可能需要具有不同性质的 PV 来解决不同的问题，比如使用 SSD 硬盘来提高性能。所以集群管理员需要能够提供各种 PV，而不仅是大小和访问模式，并且无须让用户了解这些卷的实现方式，对于这些需求可以使用 StorageClass 资源实现。

目前 PV 的提供方式有两种：静态或动态。

静态 PV 由管理员提前创建，动态 PV 无需提前创建，只需指定 PVC 的 StorageClasse 即可。

（1）回收策略

当用户使用完卷时，可以从 API 中删除 PVC 对象，从而允许回收资源。回收策略会告诉 PV 如何处理该卷，目前卷可以保留、回收或删除。

- Retain：保留，该策略允许手动回收资源，当删除 PVC 时，PV 仍然存在，volume 被视为已释放，管理员可以手动回收卷。
- Recycle：回收，如果 volume 插件支持，Recycle 策略会对卷执行 rm -rf 清理该 PV，并使其可用于下一个新的 PVC，但是本策略已弃用，建议使用动态配置。
- Delete：删除，如果 volume 插件支持，删除 PVC 时会同时删除 PV，动态卷默认为 Delete。

（2）创建 PV

在使用持久化时，需要先创建 PV，然后再创建 PVC，PVC 会和匹配的 PV 进行绑定，然后 Pod 即可使用该存储。

创建一个基于 NFS 的 PV：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
```

```
    path: /tmp
    server: 172.17.0.2
```

说明

- capacity：容量。
- accessModes：访问模式。包括以下 3 种：
  - ReadWriteOnce：可以被单节点以读写模式挂载，命令行中可以被缩写为 RWO。
  - ReadOnlyMany：可以被多个节点以只读模式挂载，命令行中可以被缩写为 ROX。
  - ReadWriteMany：可以被多个节点以读写模式挂载，命令行中可以被缩写为 RWX。
- storageClassName：PV 的类，一个特定类型的 PV 只能绑定到特定类别的 PVC。
- persistentVolumeReclaimPolicy：回收策略。
- mountOptions：非必须，新版本中已弃用。
- nfs：NFS 服务配置。包括以下两个选项：
  - path：NFS 上的目录
  - server：NFS 的 IP 地址

创建的 PV 会有以下几种状态：

- Available（可用），没有被 PVC 绑定的空间资源。
- Bound（已绑定），已经被 PVC 绑定。
- Released（已释放），PVC 被删除，但是资源还未被重新使用。
- Failed（失败），自动回收失败。

可以创建一个基于 hostPath 的 PV：

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

（3）创建 PVC

创建 PVC 需要注意的是，各个方面都符合要求 PVC 才能和 PV 进行绑定，比如 accessModes、storageClassName、volumeMode 都需要相同才能进行绑定。

创建 PVC 的示例如下：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
```

```
    name: myclaim
  spec:
    accessModes:
      - ReadWriteOnce
    volumeMode: Filesystem
    resources:
      requests:
        storage: 8Gi
    storageClassName: slow
    selector:
      matchLabels:
        release: "stable"
      matchExpressions:
        - {key: environment, operator: In, values: [dev]}
```

比如上述基于 hostPath 的 PV 可以使用以下 PVC 进行绑定，storage 可以比 PV 小：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

然后创建一个 Pod 指定 volumes 即可使用这个 PV：

```
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

注 意

claimName 需要和上述定义的 PVC 名称 task-pv-claim 一致。

3. StorageClass

StorageClass 为管理员提供了一种描述存储“类”的方法，可以满足用户不同的服务质量级别、备份策略和任意策略要求的存储需求，一般动态 PV 都会通过 StorageClass 来定义。

每个 StorageClass 包含字段 provisioner、parameters 和 reclaimPolicy，StorageClass 对象的名称很重要，管理员在首次创建 StorageClass 对象时设置的类的名称和其他参数，在被创建对象后无法再更新这些对象。

定义一个 StorageClass 的示例如下：

```
kind: StorageClass
apiVersion: storage.K8S.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
mountOptions:
  - debug
volumeBindingMode: Immediate
```

（1）Provisioner

StorageClass 有一个 provisioner 字段，用于指定配置 PV 的卷的类型，必须指定此字段，目前支持的卷插件如表 2-7 所示。

表 2-7　卷插件的类型

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
Flexvolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

注 意

provisioner 不仅限于此处列出的内部 provisioner，还可以运行和指定外部供应商。例如，NFS 不提供内部配置程序，但是可以使用外部配置程序，外部配置方式参见以下网址：https://github.com/kubernetes-incubator/external-storage

（2）ReclaimPolicy

回收策略，可以是 Delete、Retain，默认为 Delete。

（3）MountOptions

通过 StorageClass 动态创建的 PV 可以使用 MountOptions 指定挂载参数。如果指定的卷插件不支持指定的挂载选项，就不会被创建成功，因此在设置时需要进行确认。

（4）Parameters

PVC 具有描述属于 StorageClass 卷的参数，根据具体情况，取决于 provisioner，可以接受不同类型的参数。比如，type 为 io1 和特定参数 iopsPerGB 是 EBS 所具有的。如果省略配置参数，将采用默认值。

4. 定义 StorageClass

StorageClass 一般用于定义动态存储卷，只需要在 Pod 上指定 StorageClass 的名字即可自动创建对应的 PV，无须再手工创建。

以下为常用的 StorageClass 定义方式。

（1）AWS EBS

```
kind: StorageClass
apiVersion: storage.K8S.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

说明

- type：io1、gp2、sc1、st1，默认为 gp2。详情可查看以下网址的内容：

```
    https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html
```

- iopsPerGB：仅适用于 io1 卷，即每 GiB 每秒的 I/O 操作。
- fsType：Kubernetes 支持的 fsType，默认值为：ext4。

（2）GCE PD

```
kind: StorageClass
apiVersion: storage.K8S.io/v1
metadata:
  name: slow
```

```
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
```

说明

- type：pd-standard 或 pd-ssd，默认为 pd-standard。
- replication-type：none 或 regional-pd，默认值为 none。

（3）GFS

```
apiVersion: storage.K8S.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

说明

- resturl：Gluster REST 服务/Heketi 服务的 URL，这是 GFS 动态存储必需的参数。
- restauthenabled：用于对 REST 服务器进行身份验证，此选项已被启用。如果需要启用身份验证，只需指定 restuser、restuserkey、secretName 或 secretNamespace 其中一个即可。
- restuser：访问 Gluster REST 服务的用户。
- secretNamespace，secretName：与 Gluster REST 服务交互时使用的 Secret。这些参数是可选的，如果没有身份认证不用配置此参数。该 Secret 使用 type 为 kubernetes.io/glusterfs 的 Secret 进行创建，例如：

```
kubectl create secret generic heketi-secret \
  --type="kubernetes.io/glusterfs" --from-literal=key='opensesame' \
  --namespace=default
```

- clusterid：Heketi 创建集群的 ID，可以是一个列表，用逗号分隔。
- gidMin，gidMax：StorageClass 的 GID 范围，可选，默认为 2000-2147483647。
- volumetype：创建的 GFS 卷的类型，主要分为以下 3 种：
  - Replica 卷：volumetype: replicate:3，表示每个 PV 会创建 3 个副本。
  - Disperse/EC 卷：volumetype: disperse:4:2，其中 4 是数据，2 是冗余。
  - Distribute 卷：volumetype: none。

当使用 GFS 作为动态配置 PV 时，会自动创建一个格式为 gluster-dynamic-<claimname>的 Endpoint 和 Headless Service，删除 PVC 会自动删除 PV、Endpoint 和 Headless Service。

（4）Ceph RBD

```
kind: StorageClass
apiVersion: storage.K8S.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  userSecretNamespace: default
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

说明

- monitors：Ceph 的 monitor，用逗号分隔，此参数是必需的。
- adminId：Ceph 客户端的 ID，默认为 admin。
- adminSecretName：adminId 的 Secret 名称，此参数是必需的，该 Secret 必须是 kubernetes.io/rbd 类型。
- adminSecretNamespace：Secret 所在的 NameSpace（命名空间），默认为 default。
- pool：Ceph RBD 池，默认为 rbd。
- userId：Ceph 客户端 ID，默认值与 adminId 相同。
- userSecretName：和 adminSecretName 类似，必须与 PVC 存在于同一个命名空间，创建方式如下：

```
    kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \

--from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9
PQ==' \
     --namespace=kube-system
```

- imageFormat：Ceph RBD 镜像格式，默认值为 2，旧一些的为 1。
- imagefeatures：可选参数，只有设置 imageFormat 为 2 时才能使用，目前仅支持 layering。

更多详情请参考以下网址：

https://kubernetes.io/docs/concepts/storage/storage-classes/

4. 动态存储卷

动态卷的配置允许按需自动创建 PV，如果没有动态配置，集群管理员必须手动创建 PV。

动态卷的配置基于 StorageClass API 组中的 API 对象 storage.K8S.io。

（1）定义 GCE 动态预配置

要启用动态配置，集群管理员需要为用户预先创建一个或多个 StorageClass 对象，比如创建一

个名字为 slow 且使用 gce 提供存储卷的 StorageClass：

```
apiVersion: storage.K8S.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

再例如创建一个能提供 SSD 磁盘的 StorageClass：

```
apiVersion: storage.K8S.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

用户通过定义包含 StorageClass 的 PVC 来请求动态调配的存储。在 Kubernetesv 1.6 之前，是通过 volume.beta.kubernetes.io/storage-class 注解来完成的。在 1.6 版本之后，此注解已弃用。

例如，创建一个快速存储类，定义的 PersistentVolumeClaim 如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
  resources:
    requests:
      storage: 30Gi
```

注　意

storageClassName 要与上述创建的 StorageClass 名字相同。

之后会自动创建一个 PV 与该 PVC 进行绑定，然后 Pod 即可挂载使用。

（2）定义 GFS 动态预配置

可以参考 3.1 节定义一个 GFS 的 StorageClass：

```
apiVersion: storage.K8S.io/v1
kind: StorageClass
metadata:
  name: gluster-heketi
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://10.111.95.240:8080"
  restauthenabled: "false"
```

之后定义一个 PVC：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-gluster-heketi
spec:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "gluster-heketi"
  resources:
    requests:
      storage: 1Gi
```

PVC 一旦被定义，系统便发出 Heketi 进行相应的操作，在 GFS 集群上创建 brick，再创建并启动一个 volume。

然后定义一个 Pod 使用该存储卷：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-use-pvc
spec:
  containers:
  - name: pod-use-pvc
    image: busybox
    command:
    - sleep
    - "3600"
    volumeMounts:
    - name: gluster-volume
      mountPath: "/pv-data"
      readOnly: false
  volumes:
  - name: gluster-volume
    persistentVolumeClaim:
      claimName: pvc-gluster-heketi
```

claimName 为上述创建的 PVC 的名称。

2.2.13 Service

Service 主要用于 Pod 之间的通信，对于 Pod 的 IP 地址而言，Service 是提前定义好并且是不变的资源类型。

1. 基本概念

Kubernetes Pod 具有生命周期的概念，它可以被创建、删除、销毁，一旦被销毁就意味着生命周期的结束。通过 ReplicaSet 能够动态地创建和销毁 Pod，例如进行扩缩容和执行滚动升级。每个 Pod 都会获取到它自己的 IP 地址，但是这些 IP 地址不总是稳定和可依赖的，这样就会导致一个问题：在 Kubernetes 集群中，如果一组 Pod（比如后端的 Pod）为其他 Pod（比如前端的 Pod）提供服务，那么如果它们之间使用 Pod 的 IP 地址进行通信，在 Pod 重建后，将无法再进行连接。

为了解决上述问题，Kubernetes 引用了 Service 这样一种抽象概念：逻辑上的一组 Pod，即一

种可以访问 Pod 的策略——通常称为微服务。这一组 Pod 能够被 Service 访问到，通常是通过 Label Selector（标签选择器）实现的。

举个例子，有一个用作图片处理的 backend（后端），运行了 3 个副本，这些副本是可互换的，所以 frontend（前端）不需要关心它们调用了哪个 backend 副本，然而组成这一组 backend 程序的 Pod 实际上可能会发生变化，即便这样 frontend 也没有必要知道，而且也不需要跟踪这一组 backend 的状态，因为 Service 能够解耦这种关联。

对于 Kubernetes 集群中的应用，Kubernetes 提供了简单的 Endpoints API，只要 Service 中的一组 Pod 发生变更，应用程序就会被更新。对非 Kubernetes 集群中的应用，Kubernetes 提供了基于 VIP 的网桥的方式访问 Service，再由 Service 重定向到 backend Pod。

2. 定义 Service

一个 Service 在 Kubernetes 中是一个 REST 对象，和 Pod 类似。像所有 REST 对象一样，Service 的定义可以基于 POST 方式，请求 APIServer 创建新的实例。例如，假定有一组 Pod，它们暴露了 9376 端口，同时具有 app=MyApp 标签。此时可以定义 Service 如下：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

上述配置创建一个名为 my-service 的 Service 对象，它会将请求代理到 TCP 端口为 9376 并且具有标签 app=MyApp 的 Pod 上。这个 Service 会被分配一个 IP 地址，通常称为 ClusterIP，它会被服务的代理使用。

需要注意的是，Service 能够将一个接收端口映射到任意的 targetPort。默认情况下，targetPort 将被设置为与 Port 字段相同的值。targetPort 可以设置为一个字符串，引用 backend Pod 的一个端口的名称。

Kubernetes Service 能够支持 TCP 和 UDP 协议，默认为 TCP 协议。

3. 定义没有 Selector 的 Service

Service 抽象了该如何访问 Kubernetes Pod，但也能够抽象其他类型的 backend，例如：

- 希望在生产环境中访问外部的数据库集群。
- 希望 Service 指向另一个 NameSpace 中或其他集群中的服务。
- 正在将工作负载转移到 Kubernetes 集群，和运行在 Kubernetes 集群之外的 backend。

在任何这些场景中，都能定义没有 Selector 的 Service：

```
kind: Service
apiVersion: v1
metadata:
```

```
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

由于这个 Service 没有 Selector，就不会创建相关的 Endpoints 对象，可以手动将 Service 映射到指定的 Endpoints：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

注 意

Endpoint IP 地址不能是 loopback（127.0.0.0/8）、link-local（169.254.0.0/16）或者 link-local 多播地址（224.0.0.0/24）。

访问没有 Selector 的 Service 与有 Selector 的 Service 的原理相同。请求将被路由到用户定义的 Endpoint，该示例为 1.2.3.4:9376。

ExternalName Service 是 Service 的特例，它没有 Selector，也没有定义任何端口和 Endpoint，它通过返回该外部服务的别名来提供服务。

比如当查询主机 my-service.prod.svc 时，集群的 DNS 服务将返回一个值为 my.database.example.com 的 CNAME 记录：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

4. VIP 和 Service 代理

在 Kubernetes 集群中，每个节点运行一个 kube-proxy 进程。kube-proxy 负责为 Service 实现了一种 VIP（虚拟 IP）的形式，而不是 ExternalName 的形式。在 Kubernetesv 1.0 版本中，代理完全是 userspace。在 Kubernetesv 1.1 版中新增了 iptables 代理，从 Kubernetesv 1.2 版起，默认是 iptables 代理。从 Kubernetesv 1.8 版开始新增了 ipvs 代理，生产环境建议使用 ipvs 模式。

在 Kubernetesv 1.0 版中 Service 是 4 层（TCP/UDP over IP）概念，在 Kubernetesv 1.1 版中新增了 Ingress API（beta 版），用来表示 7 层（HTTP）服务。

（1）iptables 代理模式

这种模式下 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 它会创建 iptables 规则，从而捕获到该 Service 的 ClusterIP（虚拟 IP）和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某个 Pod 上面。对于每个 Endpoints 对象，它也会创建 iptables 规则，这个规则会选择一个 backend Pod。

默认的策略是随机选择一个 backend，如果要实现基于客户端 IP 的会话亲和性，可以将 service.spec.sessionAffinity 的值设置为 ClusterIP（默认为 None）。

和 userspace 代理类似，网络返回的结果都是到达 Service 的 IP:Port 请求，这些请求会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、Service 或 Pod 的任何信息。这比 userspace 代理更快、更可靠，并且当初始选择的 Pod 没有响应时，iptables 代理能够自动重试另一个 Pod。

（2）ipvs 代理模式

在此模式下，kube-proxy 监视 Kubernetes Service 和 Endpoint，调用 netlink 接口以相应地创建 ipvs 规则，并定期与 Kubernetes Service 和 Endpoint 同步 ipvs 规则，以确保 ipvs 状态与期望保持一致。访问服务时，流量将被重定向到其中一个后端 Pod。

与 iptables 类似，ipvs 基于 netfilter 钩子函数，但是 ipvs 使用哈希表作为底层数据结构并在内核空间中工作，这意味着 ipvs 可以更快地重定向流量，并且在同步代理规则时具有更好的性能，此外，ipvs 为负载均衡算法提供了更多的选项，例如：

- rr　轮询
- lc　最少连接
- dh　目标哈希
- sh　源哈希
- sed　预计延迟最短
- nq　从不排队

5. 多端口 Service

在许多情况下，Service 可能需要暴露多个端口，对于这种情况 Kubernetes 支持 Service 定义多个端口，但使用多个端口时，必须提供所有端口的名称，例如：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
```

```
    port: 443
    targetPort: 9377
```

6. 发布服务/服务类型

对于应用程序的某些部分（例如前端），一般要将服务公开到集群外部供用户访问。这种情况下都是用 Ingress 通过域名进行访问。

Kubernetes ServiceType（服务类型）主要包括以下几种：

- ClusterIP　在集群内部使用，默认值，只能从集群中访问。
- NodePort　在所有节点上打开一个端口，此端口可以代理至后端 Pod，可以通过 NodePort 从集群外部访问集群内的服务，格式为 NodeIP:NodePort。
- LoadBalancer　使用云提供商的负载均衡器公开服务，成本较高。
- ExternalName　通过返回定义的 CNAME 别名，没有设置任何类型的代理，需要 1.7 或更高版本 kube-dns 支持。

以 NodePort 为例。如果将 type 字段设置为 NodePort，则 Kubernetes 将从--service-node-port-range 参数指定的范围（默认为 30000-32767）中自动分配端口，也可以手动指定 NodePort，并且每个节点将代理该端口到 Service。

一般格式如下：

```
kind: Service
apiVersion: v1
metadata:
  labels:
    K8S-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - port: 443
      targetPort: 8443
      nodePort: 30000
  selector:
    K8S-app: kubernetes-dashboard
```

常用的服务访问是 NodePort 和 Ingress（关于 Ingress 参看 2.2.14 节），其他服务访问方式详情参看以下网址：

https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types

2.2.14 Ingress

Ingress 为 Kubernetes 集群中的服务提供了入口，可以提供负载均衡、SSL 终止和基于名称的虚拟主机，在生产环境中常用的 Ingress 有 Treafik、Nginx、HAProxy、Istio 等。

1. 基本概念

在 Kubernetesv 1.1 版中添加的 Ingress 用于从集群外部到集群内部 Service 的 HTTP 和 HTTPS

路由，流量从 Internet 到 Ingress 再到 Services 最后到 Pod 上，通常情况下，Ingress 部署在所有的 Node 节点上。

Ingress 可以配置提供服务外部访问的 URL、负载均衡、终止 SSL，并提供基于域名的虚拟主机。但 Ingress 不会暴露任意端口或协议。

2. 创建一个 Ingress

创建一个简单的 Ingress 如下：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

上述 host 定义该 Ingress 的域名，将其解析至任意 Node 上即可访问。

如果访问的是 foo.bar.com/foo，则被转发到 service1 的 4200 端口。

如果访问的是 foo.bar.com/bar，则被转发到 service2 的 8080 端口。

（1）Ingress Rules

- host：可选，一般都会配置对应的域名。
- path：每个路径都有一个对应的 serviceName 和 servicePort，在流量到达服务之前，主机和路径都会与传入请求的内容匹配。
- backend：描述 Service 和 Port 的组合。对 Ingress 匹配主机和路径的 HTTP 与 HTTPS 请求将被发送到对应的后端。

（2）默认后端

没有匹配到任何规则的流量将被发送到默认后端。默认后端通常是 Ingress Controller 的配置选项，并未在 Ingress 资源中指定。

3. Ingress 类型

（1）单域名

单个域名匹配多个 path 到不同的 service：

```
apiVersion: extensions/v1beta1
kind: Ingress
```

```
    metadata:
      name: simple-fanout-example
      annotations:
        nginx.ingress.kubernetes.io/rewrite-target: /
    spec:
      rules:
      - host: foo.bar.com
        http:
          paths:
          - path: /foo
            backend:
              serviceName: service1
              servicePort: 4200
          - path: /bar
            backend:
              serviceName: service2
              servicePort: 8080
```

此时，访问 foo.bar.com/foo 到 service1 的 4200。访问 foo.bar.com/bar 到 service2 的 8080。

（2）多域名

基于域名的虚拟主机支持将 HTTP 流量路由到同一 IP 地址的多个主机名：

```
    apiVersion: extensions/v1beta1
    kind: Ingress
    metadata:
      name: name-virtual-host-ingress
    spec:
      rules:
      - host: foo.bar.com
        http:
          paths:
          - backend:
              serviceName: service1
              servicePort: 80
      - host: bar.foo.com
        http:
          paths:
          - backend:
              serviceName: service2
              servicePort: 80
```

此时，访问 foo.bar.com 到 service1，访问 bar.foo.com 到 service2。

（3）基于 TLS 的 Ingress

首先创建证书，生产环境的证书为公司购买的证书：

```
    kubectl -n default create secret tls nginx-test-tls --key=tls.key
--cert=tls.crt
```

定义 Ingress（此示例为 Traefik，nginx-ingress 将 traefik 改为 nginx 即可）：

```
    apiVersion: extensions/v1beta1
    kind: Ingress
    metadata:
      name: nginx-https-test
```

```
    namespace: default
    annotations:
      kubernetes.io/ingress.class: traefik
  spec:
    rules:
    - host: traefix-test.com
      http:
        paths:
        - backend:
            serviceName: nginx-svc
            servicePort: 80
    tls:
     - secretName: nginx-test-tls
```

4. 更新 Ingress

更新 Ingress 可以直接使用 kubectl edit ingress INGRESS-NAME 进行更改，也可以通过 kubectlapply-f NEW-INGRESS-YAML.yaml 进行更改。

更多 Ingress 配置请参考第 5 章 Nginx Ingress 的内容。

2.2.15　Taint 和 Toleration

Taint 能够使节点排斥一类特定的 Pod，Taint 和 Toleration 相互配合可以用来避免 Pod 被分配到不合适的节点，比如 Master 节点不允许部署系统组件之外的其他 Pod。每个节点上都可以应用一个或多个 Taint，这表示对于那些不能容忍这些 Taint 的 Pod 是不会被该节点接受的。如果将 Toleration 应用于 Pod 上，则表示这些 Pod 可以（但不要求）被调度到具有匹配 Taint 的节点上。

1. 概念

给节点增加一个 Taint：

```
[root@K8S-master01 2.2.8]# kubectl taint nodes K8S-node01 key=value:NoSchedule
node/K8S-node01 tainted
```

上述命令给 K8S-node01 增加一个 Taint，它的 key 对应的就是键，value 对应就是值，effect 对应的就是 NoSchedule。这表明只有和这个 Taint 相匹配的 Toleration 的 Pod 才能够被分配到 K8S-node01 节点上。按如下方式在 PodSpec 中定义 Pod 的 Toleration，就可以将 Pod 部署到该节点上。

方式一：

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

方式二：

```
tolerations:
- key: "key"
  operator: "Exists"
```

```
    effect: "NoSchedule"
```

一个 Toleration 和一个 Taint 相匹配是指它们有一样的 key 和 effect，并且如果 operator 是 Exists（此时 toleration 不指定 value）或者 operator 是 Equal，则它们的 value 应该相等。

注意两种情况：

- 如果一个 Toleration 的 key 为空且 operator 为 Exists，表示这个 Toleration 与任意的 key、value 和 effect 都匹配，即这个 Toleration 能容忍任意的 Taint:

```
    tolerations:
    - operator: "Exists"
```

- 如果一个 Toleration 的 effect 为空，则 key 与之相同的相匹配的 Taint 的 effect 可以是任意值:

```
    tolerations:
    - key: "key"
      operator: "Exists"
```

上述例子使用到 effect 的一个值 NoSchedule，也可以使用 PreferNoSchedule，该值定义尽量避免将 Pod 调度到存在其不能容忍的 Taint 的节点上，但并不是强制的。effect 的值还可以设置为 NoExecute。

一个节点可以设置多个 Taint，也可以给一个 Pod 添加多个 Toleration。Kubernetes 处理多个 Taint 和 Toleration 的过程就像一个过滤器：从一个节点的所有 Taint 开始遍历，过滤掉那些 Pod 中存在与之相匹配的 Toleration 的 Taint。余下未被过滤的 Taint 的 effect 值决定了 Pod 是否会被分配到该节点，特别是以下情况：

- 如果未被过滤的 Taint 中存在一个以上 effect 值为 NoSchedule 的 Taint，则 Kubernetes 不会将 Pod 分配到该节点。
- 如果未被过滤的 Taint 中不存在 effect 值为 NoExecute 的 Taint，但是存在 effect 值为 PreferNoSchedule 的 Taint，则 Kubernetes 会尝试将 Pod 分配到该节点。
- 如果未被过滤的 Taint 中存在一个以上 effect 值为 NoExecute 的 Taint，则 Kubernetes 不会将 Pod 分配到该节点（如果 Pod 还未在节点上运行），或者将 Pod 从该节点驱逐（如果 Pod 已经在节点上运行）。

例如，假设给一个节点添加了以下的 Taint：

```
kubectl taint nodes K8S-node01 key1=value1:NoSchedule
kubectl taint nodes K8S-node01 key1=value1:NoExecute
kubectl taint nodes K8S-node01 key2=value2:NoSchedule
```

然后存在一个 Pod，它有两个 Toleration：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
```

```
    value: "value1"
    effect: "NoExecute"
```

在上述例子中，该 Pod 不会被分配到上述节点，因为没有匹配第三个 Taint。但是如果给节点添加上述 3 个 Taint 之前，该 Pod 已经在上述节点中运行，那么它不会被驱逐，还会继续运行在这个节点上，因为第 3 个 Taint 是唯一不能被这个 Pod 容忍的。

通常情况下，如果给一个节点添加了一个 effect 值为 NoExecute 的 Taint，则任何不能容忍这个 Taint 的 Pod 都会马上被驱逐，任何可以容忍这个 Taint 的 Pod 都不会被驱逐。但是，如果 Pod 存在一个 effect 值为 NoExecute 的 Toleration 指定了可选属性 tolerationSeconds 的值，则该值表示是在给节点添加了上述 Taint 之后 Pod 还能继续在该节点上运行的时间，例如：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

表示如果这个 Pod 正在运行，然后一个匹配的 Taint 被添加到其所在的节点，那么 Pod 还将继续在节点上运行 3600 秒，然后被驱逐。如果在此之前上述 Taint 被删除了，则 Pod 不会被驱逐。

删除一个 Taint：

```
kubectl taint nodes K8S-node01 key1:NoExecute-
```

查看 Taint：

```
[root@K8S-master01 2.2.8]# kubectl describe node K8S-node01 | grep Taint
Taints:             key=value:NoSchedule
```

2. 用例

通过 Taint 和 Toleration 可以灵活地让 Pod 避开某些节点或者将 Pod 从某些节点被驱逐。下面是几种情况。

（1）专用节点

如果想将某些节点专门分配给特定的一组用户使用，可以给这些节点添加一个 Taint（kubectl taint nodes nodename dedicated=groupName:NoSchedule），然后给这组用户的 Pod 添加一个相对应的 Toleration。拥有上述 Toleration 的 Pod 就能够被分配到上述专用节点，同时也能够被分配到集群中的其他节点。如果只希望这些 Pod 只能分配到上述专用节点中，那么还需要给这些专用节点另外添加一个和上述 Taint 类似的 Label（例如：dedicated=groupName），然后给 Pod 增加节点亲和性要求或者使用 NodeSelector，就能将 Pod 只分配到添加了 dedicated=groupName 标签的节点上。

（2）特殊硬件的节点

在部分节点上配备了特殊硬件（比如 GPU）的集群中，我们只允许特定的 Pod 才能部署在这些节点上。这时可以使用 Taint 进行控制，添加 Taint 如 kubectl taint nodes nodename special=true:NoSchedule 或者 kubectl taint nodes nodename special=true:PreferNoSchedule，然后给需要部署在这些节点上的 Pod 添加相匹配的 Toleration 即可。

（3）基于 Taint 的驱逐

属于 alpha 特性，在每个 Pod 中配置在节点出现问题时的驱逐行为。

3. 基于 Taint 的驱逐

之前提到过 Taint 的 effect 值 NoExecute，它会影响已经在节点上运行的 Pod。如果 Pod 不能忍受 effect 值为 NoExecute 的 Taint，那么 Pod 将会被马上驱逐。如果能够忍受 effect 值为 NoExecute 的 Taint，但是在 Toleration 定义中没有指定 tolerationSeconds，则 Pod 还会一直在这个节点上运行。

在 Kubernetes 1.6 版以后已经支持（alpha）当某种条件为真时，Node Controller 会自动给节点添加一个 Taint，用以表示节点的问题。当前内置的 Taint 包括：

- node.kubernetes.io/not-ready　节点未准备好，相当于节点状态 Ready 的值为 False。
- node.kubernetes.io/unreachable　Node Controller 访问不到节点，相当于节点状态 Ready 的值为 Unknown。
- node.kubernetes.io/out-of-disk　节点磁盘耗尽。
- node.kubernetes.io/memory-pressure　节点存在内存压力。
- node.kubernetes.io/disk-pressure　节点存在磁盘压力。
- node.kubernetes.io/network-unavailable　节点网络不可达。
- node.kubernetes.io/unschedulable　节点不可调度。
- node.cloudprovider.kubernetes.io/uninitialized　如果 Kubelet 启动时指定了一个外部的 cloudprovider，它将给当前节点添加一个 Taint 将其标记为不可用。在 cloud-controller-manager 的一个 controller 初始化这个节点后，Kubelet 将删除这个 Taint。

使用这个 alpha 功能特性，结合 tolerationSeconds，Pod 就可以指定当节点出现一个或全部上述问题时，Pod 还能在这个节点上运行多长时间。

比如，一个使用了很多本地状态的应用程序在网络断开时，仍然希望停留在当前节点上运行一段时间，愿意等待网络恢复以避免被驱逐。在这种情况下，Pod 的 Toleration 可以这样配置：

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

注　意

Kubernetes 会自动给 Pod 添加一个 key 为 node.kubernetes.io/not-ready 的 Toleration 并配置 tolerationSeconds=300，同样也会给 Pod 添加一个 key 为 node.kubernetes.io/unreachable 的 Toleration 并配置 tolerationSeconds=300，除非用户自定义了上述 key，否则会采用这个默认设置。

这种自动添加 Toleration 的机制保证了在其中一种问题被检测到时，Pod 默认能够继续停留在当前节点运行 5 分钟。这两个默认 Toleration 是由 DefaultTolerationSeconds admission controller 添加的。

DaemonSet 中的 Pod 被创建时，针对以下 Taint 自动添加的 NoExecute 的 Toleration 将不会指

定 tolerationSeconds：

- node.alpha.kubernetes.io/unreachable
- node.kubernetes.io/not-ready

这保证了出现上述问题时 DaemonSet 中的 Pod 永远不会被驱逐。

2.2.16　RBAC

1. RBAC 基本概念

RBAC（Role-Based Access Control，基于角色的访问控制）是一种基于企业内个人用户的角色来管理对计算机或网络资源的访问方法，其在 Kubernetes 1.5 版本中引入，在 1.6 时升级为 Beta 版本，并成为 Kubeadm 安装方式下的默认选项。启用 RBAC 需要在启动 APIServer 时指定 --authorization-mode=RBAC。

RBAC 使用 rbac.authorization.K8S.io API 组来推动授权决策，允许管理员通过 Kubernetes API 动态配置策略。

RBAC API 声明了 4 种顶级资源对象，即 Role、ClusterRole、RoleBinding、ClusterRoleBinding，管理员可以像使用其他 API 资源一样使用 kubectl API 调用这些资源对象。例如：kubectl create -f (resource).yml。

2. Role 和 ClusterRole

Role 和 ClusterRole 的关键区别是，Role 是作用于命名空间内的角色，ClusterRole 作用于整个集群的角色。

在 RBAC API 中，Role 包含表示一组权限的规则。权限纯粹是附加允许的，没有拒绝规则。

Role 只能授权对单个命名空间内的资源的访问权限，比如授权对 default 命名空间的读取权限：

```
kind: Role
apiVersion: rbac.authorization.K8S.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole 也可将上述权限授予作用于整个集群的 Role，主要区别是，ClusterRole 是集群范围的，因此它们还可以授予对以下内容的访问权限：

- 集群范围的资源（如 Node）。
- 非资源端点（如/healthz）。
- 跨所有命名空间的命名空间资源（如 Pod）。

比如，授予对任何特定命名空间或所有命名空间中的 secret 的读权限（取决于它的绑定方式）：

```
kind: ClusterRole
```

```
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      # "namespace" omitted since ClusterRoles are not namespaced
      name: secret-reader
    rules:
    - apiGroups: [""]
      resources: ["secrets"]
      verbs: ["get", "watch", "list"]
```

3. RoleBinding 和 ClusterRoleBinding

RoleBinding 将 Role 中定义的权限授予 User、Group 或 Service Account。RoleBinding 和 ClusterRoleBinding 最大的区别与 Role 和 ClusterRole 的区别类似，即 RoleBinding 作用于命名空间，ClusterRoleBinding 作用于集群。

RoleBinding 可以引用同一命名空间的 Role 进行授权，比如将上述创建的 pod-reader 的 Role 授予 default 命名空间的用户 jane，这将允许 jane 读取 default 命名空间中的 Pod：

```
    # This role binding allows "jane" to read pods in the "default" namespace.
    kind: RoleBinding
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      name: read-pods
      namespace: default
    subjects:
    - kind: User
      name: jane # Name is case sensitive
      apiGroup: rbac.authorization.K8S.io
    roleRef:
      kind: Role #this must be Role or ClusterRole
      name: pod-reader # this must match the name of the Role or ClusterRole you
wish to bind to
      apiGroup: rbac.authorization.K8S.io
```

说明

- roleRef：绑定的类别，可以是 Role 或 ClusterRole。

RoleBinding 也可以引用 ClusterRole 来授予对命名空间资源的某些权限。管理员可以为整个集群定义一组公用的 ClusterRole，然后在多个命名空间中重复使用。

比如，创建一个 RoleBinding 引用 ClusterRole，授予 dave 用户读取 development 命名空间的 Secret：

```
    # This role binding allows "dave" to read secrets in the "development" namespace.
    kind: RoleBinding
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      name: read-secrets
      namespace: development # This only grants permissions within the "development"
namespace.
    subjects:
    - kind: User
      name: dave # Name is case sensitive
      apiGroup: rbac.authorization.K8S.io
```

```
    roleRef:
      kind: ClusterRole
      name: secret-reader
      apiGroup: rbac.authorization.K8S.io
```

ClusterRoleBinding 可用于在集群级别和所有命名空间中授予权限，比如允许组 manager 中的所有用户都能读取任何命名空间的 Secret：

```
    # This cluster role binding allows anyone in the "manager" group to read secrets
in any namespace.
    kind: ClusterRoleBinding
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      name: read-secrets-global
    subjects:
    - kind: Group
      name: manager # Name is case sensitive
      apiGroup: rbac.authorization.K8S.io
    roleRef:
      kind: ClusterRole
      name: secret-reader
      apiGroup: rbac.authorization.K8S.io
```

4. 对集群资源的权限控制

在 Kubernetes 中，大多数资源都由其名称的字符串表示，例如 pods。但是一些 Kubernetes API 涉及的子资源（下级资源），例如 Pod 的日志，对应的 Endpoint 的 URL 是：

```
    GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

在这种情况下，pods 是命名空间资源，log 是 Pod 的下级资源，如果对其进行访问控制，要使用斜杠来分隔资源和子资源，比如定义一个 Role 允许读取 Pod 和 Pod 日志：

```
    kind: Role
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      namespace: default
      name: pod-and-pod-logs-reader
    rules:
    - apiGroups: [""]
      resources: ["pods", "pods/log"]
      verbs: ["get", "list"]
```

针对具体资源（使用 resourceNames 指定单个具体资源）的某些请求，也可以通过使用 get、delete、update、patch 等进行授权，比如，只能对一个叫 my-configmap 的 configmap 进行 get 和 update 操作：

```
    kind: Role
    apiVersion: rbac.authorization.K8S.io/v1
    metadata:
      namespace: default
      name: configmap-updater
    rules:
    - apiGroups: [""]
      resources: ["configmaps"]
```

```
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

注 意

如果使用了 resourceNames，则 verbs 不能是 list、watch、create、deletecollection 等。

5. 聚合 ClusterRole

从 Kubernetes 1.9 版本开始，Kubernetes 可以通过一组 ClusterRole 创建聚合 ClusterRoles，聚合 ClusterRoles 的权限由控制器管理，并通过匹配 ClusterRole 的标签自动填充相对应的权限。

比如，匹配 rbac.example.com/aggregate-to-monitoring: "true"标签来创建聚合 ClusterRole：

```
kind: ClusterRole
apiVersion: rbac.authorization.K8S.io/v1
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # Rules are automatically filled in by the controller manager.
```

然后创建与标签选择器匹配的 ClusterRole 向聚合 ClusterRole 添加规则：

```
kind: ClusterRole
apiVersion: rbac.authorization.K8S.io/v1
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# These rules will be added to the "monitoring" role.
rules:
- apiGroups: [""]
  resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

6. Role 示例

以下示例允许读取核心 API 组中的资源 Pods（只写了规则 rules 部分）：

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

允许在 extensions 和 apps API 组中读写 deployments：

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

允许对 Pods 的读和 Job 的读写：

```
rules:
- apiGroups: [""]
```

```
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["batch", "extensions"]
    resources: ["jobs"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

允许读取一个名为 my-config 的 ConfigMap（必须绑定到一个 RoleBinding 来限制到一个命名空间下的 ConfigMap）：

```
  rules:
  - apiGroups: [""]
    resources: ["configmaps"]
    resourceNames: ["my-config"]
    verbs: ["get"]
```

允许读取核心组 Node 资源（Node 属于集群级别的资源，必须放在 ClusterRole 中，并使用 ClusterRoleBinding 进行绑定）：

```
  rules:
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
```

允许对非资源端点/healthz 和所有其子资源路径的 Get 和 Post 请求（必须放在 ClusterRole 并与 ClusterRoleBinding 进行绑定）：

```
  rules:
  - nonResourceURLs: ["/healthz", "/healthz/*"] # '*' in a nonResourceURL is a
suffix glob match
    verbs: ["get", "post"]
```

7. RoleBinding 示例

以下示例绑定为名为“alice@example.com”的用户（只显示 subjects 部分）：

```
  subjects:
  - kind: User
    name: "alice@example.com"
    apiGroup: rbac.authorization.K8S.io
```

绑定为名为“frontend-admins”的组：

```
  subjects:
  - kind: Group
    name: "frontend-admins"
    apiGroup: rbac.authorization.K8S.io
```

绑定为 kube-system 命名空间中的默认 Service Account：

```
  subjects:
  - kind: ServiceAccount
    name: default
    namespace: kube-system
```

绑定为 qa 命名空间中的所有 Service Account：

```
  subjects:
  - kind: Group
```

```
  name: system:serviceaccounts:qa
  apiGroup: rbac.authorization.K8S.io
```

绑定所有 Service Account：

```
subjects:
- kind: Group
  name: system:serviceaccounts
  apiGroup: rbac.authorization.K8S.io
```

绑定所有经过身份验证的用户（v1.5+）：

```
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.K8S.io
```

绑定所有未经过身份验证的用户（v1.5+）：

```
subjects:
- kind: Group
  name: system:unauthenticated
  apiGroup: rbac.authorization.K8S.io
```

对于所有用户：

```
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.K8S.io
- kind: Group
  name: system:unauthenticated
  apiGroup: rbac.authorization.K8S.io
```

8. 命令行的使用

权限的创建可以使用命令行直接创建，较上述方式更加简单、快捷，下面我们逐一介绍常用命令的使用。

（1）kubectl create role

创建一个 Role，命名为 pod-reader，允许用户在 Pod 上执行 get、watch 和 list：

```
kubectl create role pod-reader --verb=get --verb=list --verb=watch
--resource=pods
```

创建一个指定了 resourceNames 的 Role，命名为 pod-reader：

```
kubectl create role pod-reader --verb=get --resource=pods
--resource-name=readablepod --resource-name=anotherpod
```

创建一个命名为 foo，并指定 APIGroups 的 Role：

```
kubectl create role foo --verb=get,list,watch --resource=replicasets.apps
```

针对子资源创建一个名为 foo 的 Role：

```
kubectl create role foo --verb=get,list,watch --resource=pods,pods/status
```

针对特定/具体资源创建一个名为 my-component-lease-holder 的 Role：

```
    kubectl create role my-component-lease-holder --verb=get,list,watch,update
--resource=lease --resource-name=my-component
```

（2）kubectl create clusterrole

创建一个名为 pod-reader 的 ClusterRole，允许用户在 Pod 上执行 get、watch 和 list：

```
    kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods
```

创建一个名为 pod-reader 的 ClusterRole，并指定 resourceName：

```
    kubectl create clusterrole pod-reader --verb=get --resource=pods
--resource-name=readablepod --resource-name=anotherpod
```

使用指定的 apiGroup 创建一个名为 foo 的 ClusterRole：

```
    kubectl create clusterrole foo --verb=get,list,watch
--resource=replicasets.apps
```

使用子资源创建一个名为 foo 的 ClusterRole：

```
    kubectl create clusterrole foo --verb=get,list,watch
--resource=pods,pods/status
```

使用 non-ResourceURL 创建一个名为 foo 的 ClusterRole：

```
    kubectl create clusterrole "foo" --verb=get --non-resource-url=/logs/*
```

使用指定标签创建名为 monitoring 的聚合 ClusterRole：

```
    kubectl create clusterrole monitoring
--aggregation-rule="rbac.example.com/aggregate-to-monitoring=true"
```

（3）kubectl create rolebinding

创建一个名为 bob-admin-binding 的 RoleBinding，将名为 admin 的 ClusterRole 绑定到名为 acme 的命名空间中一个名为 bob 的 user：

```
    kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob
--namespace=acme
```

创建一个名为 myapp-view-binding 的 RoleBinding，将名为 view 的 ClusterRole，绑定到 acme 命名空间中名为 myapp 的 ServiceAccount：

```
    kubectl create rolebinding myapp-view-binding --clusterrole=view
--serviceaccount=acme:myapp --namespace=acme
```

（4）kubectl create clusterrolebinding

创建一个名为 root-cluster-admin-binding 的 clusterrolebinding，将名为 cluster-admin 的 ClusterRole 绑定到名为 root 的 user：

```
    kubectl create clusterrolebinding root-cluster-admin-binding
--clusterrole=cluster-admin --user=root
```

创建一个名为 myapp-view-binding 的 clusterrolebinding，将名为 view 的 ClusterRole 绑定到 acme 命名空间中名为 myapp 的 ServiceAccount：

```
    kubectl create clusterrolebinding myapp-view-binding --clusterrole=view
--serviceaccount=acme:myapp
```

2.2.17 CronJob

CronJob 用于以时间为基准周期性地执行任务，这些自动化任务和运行在 Linux 或 UNIX 系统上的 CronJob 一样。CronJob 对于创建定期和重复任务非常有用，例如执行备份任务、周期性调度程序接口、发送电子邮件等。

对于 Kubernetes 1.8 以前的版本，需要添加--runtime-config=batch/v2alpha1=true 参数至 APIServer 中，然后重启 APIServer 和 Controller Manager 用于启用 API，对于 1.8 以后的版本无须修改任何参数，可以直接使用，本节的示例基于 1.8 以上的版本。

1. 创建 CronJob

创建 CronJob 有两种方式，一种是直接使用 kubectl 创建，一种是使用 yaml 文件创建。

使用 kubectl 创建 CronJob 的命令如下：

```
    kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox
-- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
```

对应的 yaml 文件如下：

```
    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
```

说　明
本例创建一个每分钟执行一次、打印当前时间和 Hello from the Kubernetes cluster 的计划任务。

查看创建的 CronJob：

```
    $ kubectl get cj
    NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
    hello   */1 * * * *   False     0        <none>          5s
```

等待 1 分钟可以查看执行的任务（Jobs）：

```
$ kubectl get jobs
NAME             COMPLETIONS   DURATION   AGE
hello-1558779360   1/1           23s        32s
```

CronJob 每次调用任务的时候会创建一个 Pod 执行命令，执行完任务后，Pod 状态就会变成 Completed，如下所示：

```
$ kubectl get po
NAME                       READY   STATUS      RESTARTS   AGE
hello-1558779360-jcp4r       0/1     Completed   0          37s
```

可以通过 logs 查看 Pod 的执行日志：

```
$ kubectl logs -f hello-1558779360-jcp4r
Sat May 25 10:16:23 UTC 2019
Hello from the Kubernetes cluster
```

如果要删除 CronJob，直接使用 delete 即可：

```
kubectl delete cronjob hello
```

2. 可用参数的配置

定义一个 CronJob 的 yaml 文件如下：

```
apiVersion: v1
items:
- apiVersion: batch/v1beta1
  kind: CronJob
  metadata:
    labels:
      run: hello
    name: hello
    namespace: default
  spec:
    concurrencyPolicy: Allow
    failedJobsHistoryLimit: 1
    jobTemplate:
      metadata:
        creationTimestamp: null
      spec:
        template:
          metadata:
            creationTimestamp: null
            labels:
              run: hello
          spec:
            containers:
            - args:
              - /bin/sh
              - -c
              - date; echo Hello from the Kubernetes cluster
              image: busybox
              imagePullPolicy: Always
              name: hello
```

```
            resources: {}
            terminationMessagePath: /dev/termination-log
            terminationMessagePolicy: File
          dnsPolicy: ClusterFirst
          restartPolicy: OnFailure
          schedulerName: default-scheduler
          securityContext: {}
          terminationGracePeriodSeconds: 30
    schedule: '*/1 * * * *'
    successfulJobsHistoryLimit: 3
    suspend: false
```

其中各参数的说明如下（可以按需修改）：

- schedule　调度周期，和 Linux 一致，分别是分时日月周。
- restartPolicy　重启策略，和 Pod 一致。
- concurrencyPolicy　并发调度策略。可选参数如下:
  - Allow　允许同时运行多个任务。
  - Forbid　不允许并发运行，如果之前的任务尚未完成，新的任务不会被创建。
  - Replace　如果之前的任务尚未完成，新的任务会替换的之前的任务。
- Suspend　如果设置为 true，则暂停后续的任务，默认为 false。
- successfulJobsHistoryLimit　保留多少已完成的任务，按需配置。
- failedJobsHistoryLimit　保留多少失败的任务。

相对于 Linux 上的计划任务，Kubernetes 的 CronJob 更具有可配置性，并且对于执行计划任务的环境只需启动相对应的镜像即可。比如，如果需要 Go 或者 PHP 环境执行任务，就只需要更改任务的镜像为 Go 或者 PHP 即可，而对于 Linux 上的计划任务，则需要安装相对应的执行环境。此外，Kubernetes 的 CronJob 是创建 Pod 来执行，更加清晰明了，查看日志也比较方便。可见，Kubernetes 的 CronJob 更加方便和简单。

更多 CronJob 的内容，可以参考 Kubernetes 的官方文档：https://kubernetes.io/docs/home/。

2.3 小　结

本章讲解了 Docker 和 Kubernetes 在生产环境中常用的基础知识，同时也举例说明了使用场景，希望读者务必深入理解本章内容，因为概念及原理在使用过程中尤为重要，对业务架构的设计和集群排错也都有很大的帮助。有关 Docker 和 Kubernetes 的更多概念可以参考官方文档：

https://kubernetes.io/docs/concepts/

https://docs.docker.com/