在centos7上安装docker及使用kubeadm快速部署kubernetes集群

1. **系统配置更新（root）**

yum –y update

reboot

(查看内核版本： uname -a)

1. **Iptables设置（root）**

iptables -P FORWARD ACCEPT

iptables-save > /etc/sysconfig/iptables

1. **安装docker（root）**

安装所需要的包：

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

增加docker-ce 的repository：

yum-config-manager –add-repo \

<https://download.docker.com/linux/centos/docker-ce.repo>

yum makecache fast

安装docker-ce docker-ce-cli containerd.io

yum install docker-ce docker-ce-cli containerd.io

启动docker

systemctl enable docker

systemctl start docker

将非root用户加到docker用户组中

usermod –aG docker [username]

测试

docker run hello-world

1. **配置防火墙和开放必要的端口（root）**

systemctl enable firewalld

systemctl start firewalld

firewall-cmd –add-masquerade -–permanent

firewall-cmd –add-port=10250/tcp –permanent

firewall-cmd –add-port=8472/tcp --permanent

firewall-cmd –add-port=6443/tcp --permanent

firewall-cmd –reload

1. **配置netfilter（root）**

modprobe br\_netfilter

echo "br\_netfilter" > /etc/modules-load.d/br\_netfilter.conf

cat <<EOF > /etc/sysctl.d/k8s.conf

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

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

EOF

使之生效

sysctl –system

1. **配置setenforce（root）**

/usr/sbin/setenforce 0

sed -i 's/SELINUX=enforcing/SELINUX=permissive/g' /etc/selinux/config

1. **增加kubernetes的ali yum repository（root）**

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

[kubernetes]

name=Kubernetes

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

enabled=1

gpgcheck=0

repo\_gpgcheck=0

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

EOF

yum makecace fast

安装kubelet kubeadm kubectl（最好确认下最新版本号）

yum install –y kubelet kubeadm kubectl

systemctl enable kubelet

systemctl start kubelet

1. **配置swap，忽略使用虚拟内存的错误，官方不推荐使用swap（root）**

cat <<EOF > /etc/sysconfig/kubelet

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

EOF

**或者关闭系统的swap**

swapoff –a

yes | cp /etc/fstab /etc/fstab\_bak

cat /etc/fstab\_bak | grep –v swap > /etc/fstab

1. **kubeadm初始化及用户配置（root）**

拉取kubernetes必要的组件（1.14.1版本）

echo "Pull Kubernetes v1.14.1 Images from aliyuncs.com ......"

echo "Start"

MY\_REGISTRY=registry.cn-hangzhou.aliyuncs.com/openthings

*## 拉取镜像*

docker pull ${MY\_REGISTRY}/k8s-gcr-io-kube-apiserver:v1.14.1

docker pull ${MY\_REGISTRY}/k8s-gcr-io-kube-controller-manager:v1.14.1

docker pull ${MY\_REGISTRY}/k8s-gcr-io-kube-scheduler:v1.14.1

docker pull ${MY\_REGISTRY}/k8s-gcr-io-kube-proxy:v1.14.1

docker pull ${MY\_REGISTRY}/k8s-gcr-io-etcd:3.3.10

docker pull ${MY\_REGISTRY}/k8s-gcr-io-pause:3.1

docker pull ${MY\_REGISTRY}/k8s-gcr-io-coredns:1.3.1

*## 添加Tag*

docker tag ${MY\_REGISTRY}/k8s-gcr-io-kube-apiserver:v1.14.1 k8s.gcr.io/kube-apiserver:v1.14.1

docker tag ${MY\_REGISTRY}/k8s-gcr-io-kube-scheduler:v1.14.1 k8s.gcr.io/kube-scheduler:v1.14.1

docker tag ${MY\_REGISTRY}/k8s-gcr-io-kube-controller-manager:v1.14.1 k8s.gcr.io/kube-controller-manager:v1.14.1

docker tag ${MY\_REGISTRY}/k8s-gcr-io-kube-proxy:v1.14.1 k8s.gcr.io/kube-proxy:v1.14.1

docker tag ${MY\_REGISTRY}/k8s-gcr-io-etcd:3.3.10 k8s.gcr.io/etcd:3.3.10

docker tag ${MY\_REGISTRY}/k8s-gcr-io-pause:3.1 k8s.gcr.io/pause:3.1

docker tag ${MY\_REGISTRY}/k8s-gcr-io-coredns:1.3.1 k8s.gcr.io/coredns:1.3.1

echo "End"

通过kubeadm初始化整个kubernetes

kubeadm init --kubernetes-version=v1.14.1 \

--apiserver-advertise-address=192.168.91.129 \ #当前主机ip

--pod-network-cidr=192.168.0.0/16 \ #网段

--ignore-preflight-errors=Swap

设置用户配置，切换到非root用户

su – username

mkdir -p $HOME/.kube

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

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

echo export KUBECONFIG=~/.kube/config>> ~/.bashrc

source ~/.bashrc

设置kubectl自动完成

#在当前shell里设置自动完成

source <(kubectl completion bash)

# add autocomplete permanently to your bash shell

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

**安装calico网络（非root，coredns需要配置cni网络插件后才能正常启动）**

kubectl apply -f \

<https://docs.projectcalico.org/v3.6/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml>

1. **设置Master节点也可作为Node工作（非root），例如单节点操作**

kubectl taint nodes --all node-role.kubernetes.io/master-

1. **查看nodes、pods**

kubectl get nodes

kubectl get pods –all-namespaces (详细情况可加上 –o wide)

查看集群状态： kubectl get cs

多节点情况下，将某个节点加入某个集群，执行集群master初始化时生成的代码，如下

kubeadm join 192.168.91.128:6443 --token c2cc4y.6613cgw5it7kde0z \

--discovery-token-ca-cert-hash sha256:8c571f1a84be5edcb056a7fcbfc030076ecd93ee3ddaac66a79d4d5378f6fe48 \

--ignore-preflight-errors=Swap

**查询master节点token**

$ kubeadm token list

创建token： kubeadm token create

查看SHA256加密字符串：

openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin \

-outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.\* //'

master将某个节点移除：

kubectl drain [nodename] –delete-local-data –force –ignore-daemonsets

kubectl delete node [nodename]

节点node上重置kubernetes集群：

kubedam reset

node节点上使用kubectl命令需要配置KUBECONFIG

从master节点上复制/etc/kubernetes/admin.conf文件到node节点同目录下

scp /etc/kubernetes/admin.conf node@node:/etc/kubernetes/admin.conf

echo "export KUBECONFIG=/etc/kubernetes/admin.conf" >> ~/.bash\_profile

source ~/.bash\_profile

1. **发布一个nginx测试（单实例）**

kubectl create deployment nginx –imge=nginx

（通过这种方式创建的deployment在delete pods后会pods在删除后仍会重启！？）

curl [ip]

删除这个nginx

kubectl delete deployments/nginx

**ps. 实际运行一个容器时应该通过配置文件（yaml），如下**

kubectl create –f app-config.yaml

将对应pods删除： kubectl delete –f pod.yaml

**修改配置文件后生效（e.g.升级版本）:** kubectl replace –f app-config.yaml

或者： kubectl apply –f app-config.yaml

应用例子：开发完一个应用，并且制作好了容器镜像，可以在发布目录中附带一个Deployment的yaml文件，运维人员直接通过kubectl apply命令直接运行；当应用修改时，也需要修改这个yaml文件。

**使用kubectl exec 命令进入到Pod中，打开一个tty并执行/bin/bash**

$ kubectl exec –it [appname] -- /bin/bash

# ls /usr/share/nginx/html

1. **大概的开发部署流程**

* 本地开发，通过docker测试代码，制作镜像
* 选择合适的kubernetes API对象，编写对应的yaml文件 （通过kind指定Pod或 Deployment类型）
* 在Kubernetes上部署这个yaml文件， 剩余的操作通过kubectl执行或修改yaml文件，尽量不用docker的命令行

**PS. 实际编写yaml文件中，使用一个replicas=1的Deployment要比一个单独的Pod要好，因为pod所在的节点出故障的时候 pod可以调度到健康的节点上，单独的pod只能在节点健康的情况下由kubelet保证pod的健康状况。**

1. **End**

深入剖析Kubernetes笔记

**一个yaml文件的大概内容：**

****

kind: API对象类型

metadata: API对象的标识

labels: 实际运行后的Name

**Pod**

**是Kubernetes中最小的API对象，原子调度单元。**

**Pod中的所有容器，共享同一个Network Namespace，并且可以声明共享同一个Volume。**

**Pod中存在一个中间容器——Infra容器，它永远是第一个被创建的容器，而其它用户自定义的容器，则通过Network Namespace的方式，与Infra容器关联在一起。**

**Infra使用的镜像是k8s.gcr.io/pause**

**容器设计模式**

**Sidecar： an example**

****

**在Pod中，所有spec.initContainer中定义的容器，都会比spec.containers定义的用户容器先启动，并且initContainer容器会按顺序逐一启动，而直到它们都启动并且退出后，用户容器才会启动。**

**BTW：**

**Linux 支持7种namespace:  
1. cgroup用于隔离cgroup根目录;  
2. IPC用于隔离系统消息队列;  
3. Network隔离网络;  
4. Mount隔离挂载点;  
5. PID隔离进程;  
6. User隔离用户和用户组;  
7. UTS隔离主机名nis域名。**

**Volume**

**一个Volume对应宿主机的目录对于Pod来说只有一个，Pod中的容器只要声明挂载到这个Volume，就可以共享这个Volume对应的宿主机目录。For example:**

****

**Volume shared-data挂载在宿主机上的/data目录，nginx-container和debian-container再挂载到shared-data上。**

**Types of Volumes**

**emptyDir**

emptyDir类型的Volume在Pod分配到Node上时被创建，Kubernetes会在Node上自动分配一个目录，因此无需指定宿主机Node上对应的目录文件。 这个目录的初始内容为空，当Pod从Node上移除时，emptyDir中的数据会被永久删除。注：容器的crashing事件并不会导致emptyDir中的数据被删除。

使用场景：

* 临时空间，例如基于磁盘的合并排序
* 设置检查点以从崩溃事件中恢复未执行完毕的长计算
* 保存内容管理器容器从Web服务器容器提供数据时所获取的文件

**hostPath**

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

使用场景：

* 当运行的容器需要访问Docker内部结构时，如使用hostPath映射/var/lib/docker到容器；
* 当在容器中运行cAdvisor时，可以使用hostPath映射/dev/cgroups到容器中；

注意事项：

* 配置相同的pod（如通过podTemplate创建），可能在不同的Node上表现不同，因为不同节点上映射的文件内容不同
* 当Kubernetes增加了资源敏感的调度程序，hostPath使用的资源不会被计算在内
* 宿主机下创建的目录只有root有写权限。你需要让你的程序运行在privileged container上，或者修改宿主机上的文件权限。

**Pod对象的深入解析**

**Yaml文件中spec下的字段**

**nodeSelector：是一个供用户将Pod与Node进行绑定的字段。**

spec:

nodeSelector:

disktype: ssd

**nodeName：一旦Pod的这个字段被赋值，则意味这个Pod已经经过调度，因为调度的结果就是赋值节点的名字，因此，该字段一般由调度器负责设置。**

**hostAliases ：定义Pod的hosts文件（/etc/hosts）里的内容。**

****

**shareProcessNamespace：（true/false）Pod中的容器是否共享PID Namespace**

**hostNetwork：直接使用宿主机的网络**

**hostIPC：直接与宿主机进行IPC通信**

**hostPID：看到宿主机里正在运行的所有进程**

**container下的字段：**

**ImagePullPolicy：镜像拉取策略。Always表示每次创建Pod都重新拉取一次镜像；而当被定义为Never或IfNotPresent，则意味着Pod永远不会主动拉取这个镜像或者只在宿主机上不存在这个镜像时才拉取。新版本的默认值为IfNotPresent**

**Lifecycle：定义了Container Life Hooks，即在容器状态发生变化是触发一系列“钩子”。**

apiVersion: v1

kind: Pod

metadata:

name: lifecycle-demo

spec:

containers:

- name: lifecycle-demo-container

image: nginx

lifecycle:

postStart:

exec:

command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]

preStop:

exec:

command: ["/usr/sbin/nginx","-s","quit"]

**postStart指在容器启动后，立即执行一个指定的操作，实际是在Docker容器ENTRYPOINT执行之后操作，但并不严格保证顺序，即在postStart启动时ENTRYPOINT有可能还没结束。所以当postStart执行超时或错误时会在该Pod的Events中报出启动失败的错误信息，也会导致Pod处于失败的状态。**

**而preStop是在容器被杀死之前执行，是同步操作。因此它会阻塞当前的容器杀死流程，只到这个preStop定义的操作完成后才允许容器被杀死，与postStart不同。**

**Pod中与metadata、spec同级的status字段**

**Status下的phase：表示Pod的当前状态，具体有以下几种情况：**

****

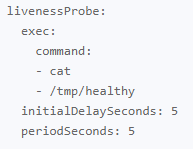
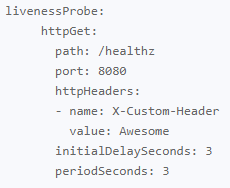
**Status下的conditions：主要包括PodScheduled、Ready、Initialized和Unschedulable，用于描述造成当前状态的具体原因是什么。**

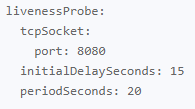
**四种特殊的Volume —— Projected Volume**

**分别为Secret、ConfigMap、Downward API和ServiceAccountToken，它们存在的意义并不是存放容器中的数据或用来进行容器和宿主机之间的数据交换，而是为容器提供预先定义好的数据**

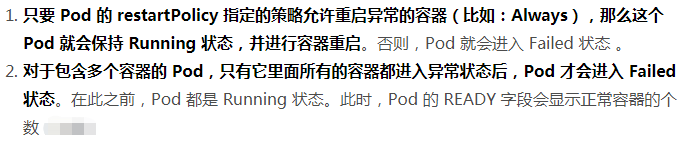
**Pod中容器健康检查和恢复机制**

**Spec.container.livenessProbe：容器健康检查，可以执行exec命令，也可以发起http或tcp请求。**

**** ****

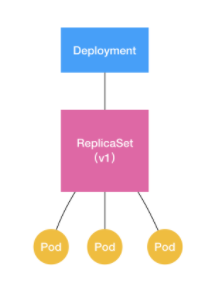
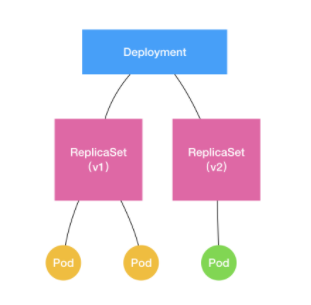
****

**Spec下restartPolicy：Pod的恢复机制，默认值为Always，即任何时候这个容器发生了异常，它就会被重新创建（旧的会被销毁）。另外，OnFailure表示只有在容器异常时才自动重启容器，Never表示从不重启容器。**

****

**注意：livenessProbe要优先于restartPolicy，当livenessProbe返回fail时不管container的状态如何都会重启**

**Deployment**

**** ****

**ReplicaSet是一个Deployment的子集，Deployment控制器实际控制ReplicaSet对象，而不是Pod对象。而ReplicaSet的个数描述应用的版本数，更新修改操作会增加**

****

replicas指定Pod的副本个数进行水平扩展

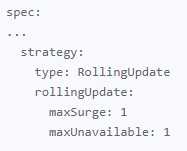
Pod的配置则写在template中

**滚动更新：将一个集群中正在运行的多个Pod版本，交替地逐一升级。**

**通过kubectl get deployments查看deployment的状态信息：**

****

**Deployment.spec.strategy中可以指定滚动更新中，可以创建多少个新Pod，删除掉多少个旧Pod。**

****

**Deployment的一些操作命令**

**#创建一个deployment，加上--record 记录每次操作所执行的命令**

$ kubectl create -f nginx-deployment.yaml --record

**#实时查看状态变化**

$ kubectl rollout status deployment/nginx-deployment

**#修改Deployment，编辑Etcd里这个API对象**

$ kubectl edit deployment/nginx-deployment

**#查看deployment状态详情**

$ kubectl describe deployment nginx-deployment

**#修改deployment所使用的镜像**

$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1

**#查看Deployment所控制的ReplicaSet**

$ kubectl get rs

**#将deployment回滚到上个版本， 加上--to-revision可以指定回滚版本**

$ kubectl rollout undo deployment/nginx-deployment

**#查看每次deployment变更版本的记录， 加上--revision可以指定版本**

$ kubectl rollout history deployment/nginx-deployment

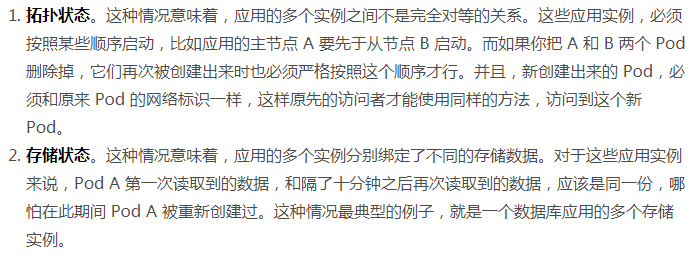
**Ps. Spec.revisionHistoryLimit设置保留历史版本的个数，若设置为0则无法做回滚操作**

**#通过kubectl rollout pause命令暂停deployment来对其进行多次更新操作，最后只生成一个ReplicaSet，在完成修改后再执行kubectl rollout resume触发一次滚动更新**

$ kubectl rollout pause deployment/nginx-deployment

$ kubectl rollout resume deployment/nginx-deployment

**有状态应用 Stateful Application**

****

**StatefulSet，通过某种方式记录状态，在Pod被重新创建时，为新的Pod恢复原来的状态。**

**Kubernetes中， Service作为将一组Pod暴露给外界访问的机制。比如一个deployment有3个Pod，在定义一个Service，用户只要能访问到这个Service，就能访问到某个Pod。**

**Service的访问方式**

****

**一个标准的Headless Service的yaml文件 svc.yaml**

****

clusterIP字段设为None表示这个Service被创建后并不会被分配一个VIP，而是会以DNS记录的方式暴露出它所代理的Pod

DNS的格式：

<pod-name>.<svc-name>.<namespace>.svc.cluster.local

**一个StatefulSet的yaml文件 statefulset.yaml**

****

serviceName=nginx告诉StatefulSet控制器在执行控制循环（control loop）时，使用nginx这个headless service来保证Pod的可解析身份

kubectl create –f svc.yaml

kubectl create –f statefulset.yaml

**# 查看两个启动的Pod的IP地址（1.28.4版本才能正确解析）**

$ kubectl run -i --tty --image busybox:1.28.4 dns-test --restart=Never --rm /bin/sh

**Pod的拓扑状态按照Pod的“名字+编号”的方式唯一固定，Pod对应的DNS**

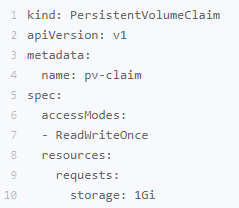
**存储状态（两种API对象），降低用户声明和使用持久化Volume的门槛**

**PVC（Persistent Volume Claim）**

**PV（Persistent Volume）**

**For example：**

1. **定义一个PVC**

****

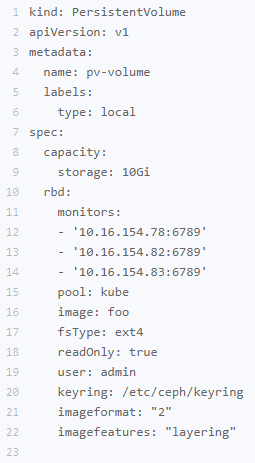
accessModes：

* ReadWriteOnce – the volume can be mounted as read-write by a single node
* ReadOnlyMany – the volume can be mounted read-only by many nodes
* ReadWriteMany – the volume can be mounted as read-write by many nodes

1. **在应用的Pod中声明使用这个PVC**

****

**创建的PVC对象，kubernetes会自动为它绑定一个符合配置要求的Volume，而实际的Volume来自定义的PV对象，如下**

****

****

VolumeClaimTemplates与template字段类似，会对每一个被这个StatefulSet管理的Pod声明一个对应配置条件的PVC。

自动创建的PVC，会与符合条件的PV绑定后，进入Bound状态

**#通过 kubectl get pvc 查看具体状态**

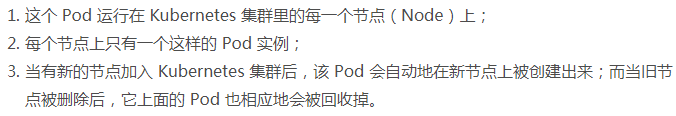
**$ kubectl get pvc –l app=nginx**

**StatefulSet的工作原理**

****

**DaemonSet**

**主要作用：在Kubernetes集群中运行一个Daemon Pod，而这个Pod有如下三个特征：**

****

**e.g. 各种网络插件的Agent组件、存储插件的Agent组件、监控组件和日志组件**

**注意，跟其他编排对象不同，DaemonSet开始运行的时机，通常比整个kubernetes集群出现的时机还要早。**

**# 查看集群中的DaemonSet对象**

**$ kubectl get ds –n kube-system [name]**

**DaemonSet控制器直接操作的就是Pod，并不像Deployment对象那样“一个版本对应一个ReplicaSet对象”，它的版本控制是通过ControllerRevision记录的。**

**（StatefulSet也是不通过ReplicaSet直接管Pod）**

**# 查看controllerrevision**

**$ kubectl get controllerrevision –n kube-system –l name=[…]**

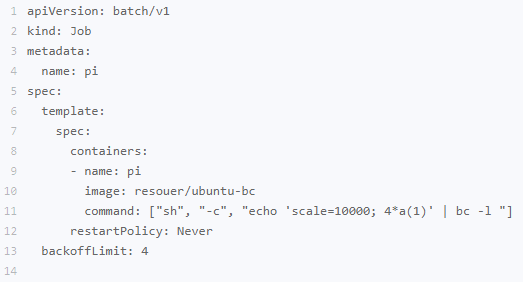
**ControllerRevision是一个通用的版本管理对象**

**spec.updateStrategy.rollingUpdate.partition**

**Canary Deploy或灰度发布时， 用于控制更新应用中的部分实例到新版本；指定一个数字，对于序号大于等于这个数字的Pod将会被更新；实际中一步步地减小partition的值，直到0，以完成更新发布。**

**Job与CronJob**

**An example:**

****

**在Job对象里restartPolicy只允许被设置为Never或OnFailure，而在deployment对象中，restartPolicy则只允许被设置为Always。**

**例子中，restartPolicy设为Never，则当Job失败时会不断尝试创建一个新的Pod，backoffLimit字段定义重试次数为4（实际默认值为6）**

**spec.activeDeadlineSeconds：设置最长运行时间（秒）**

**spec.parallelism：设置一个Job每次最多可以启动多少个Pod同时运行**

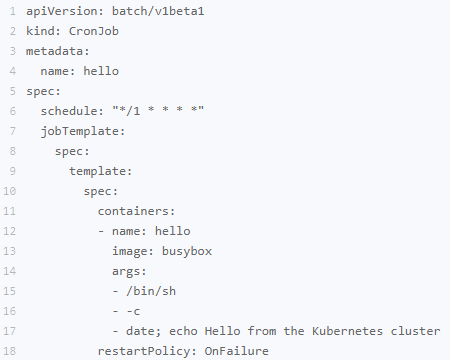
**spec.completions：设置Job至少要完成的Pod数量，即Job的最小完成数，对应DESIRED的值**

**#查看Pod的日志**

**$ kubectl logs [pod-name]**

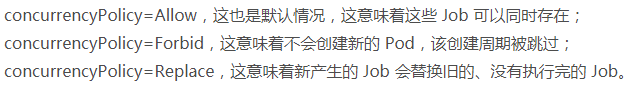
**实际上，Job Controller控制的对象直接就是Pod**

**定时任务API对象：CronJob**

****

**对于某个job还没执行完，而新的job就产生了的情况，**

**具体应对策略可通过spec.concurrencyPolicy字段处理**

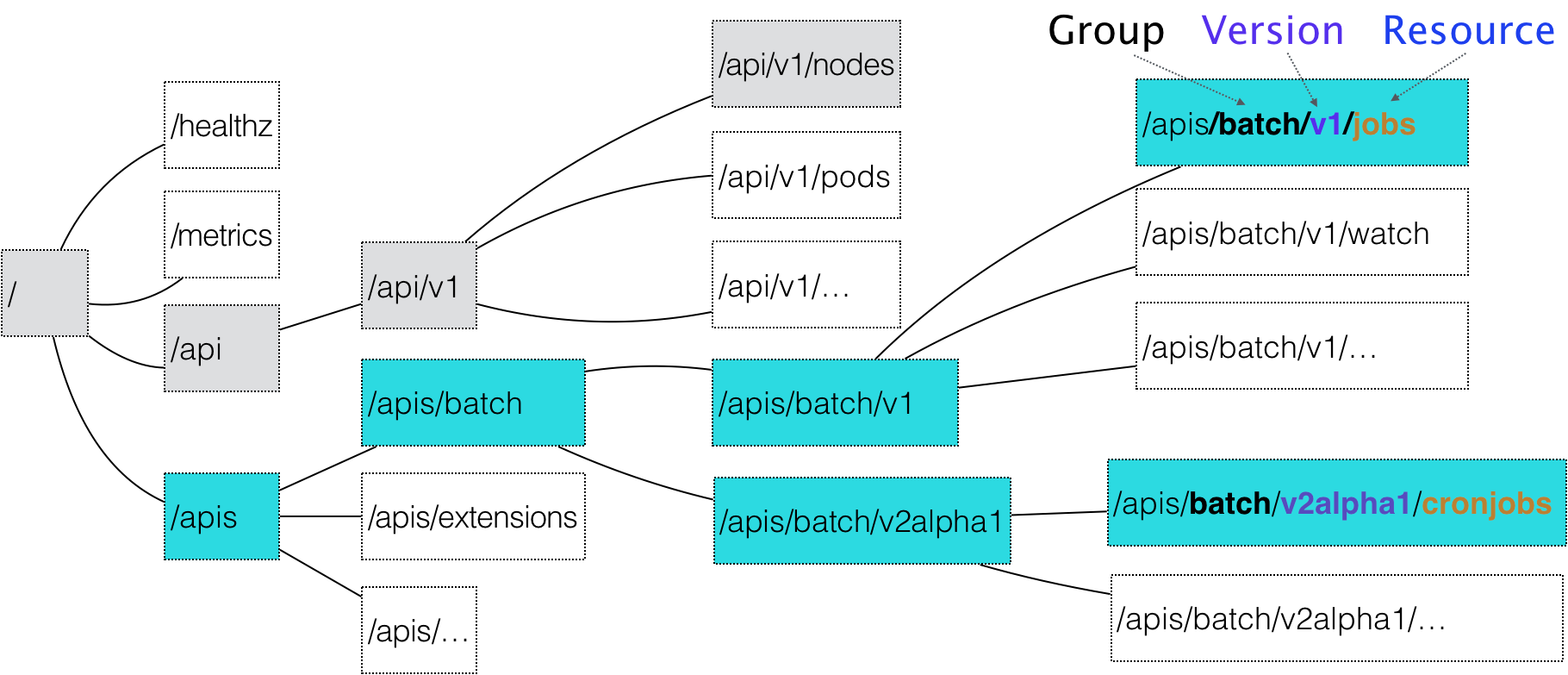
****

**如果某次Job创建失败，这次创建就会被标记为“miss”，在指定的时间窗口内，miss的数量达到100时，CronJob会停止再创建这个Job。**

**时间窗口由spec.startingDeadlineSeconds字段指定。**

**声明式API与Kubernetes编程范式**

**在Kubernetes项目中，一个API对象在Etcd里的完整资源路径，是由Group（API组） Version（API版本）和Resource（API资源类型）三个部分组成。**

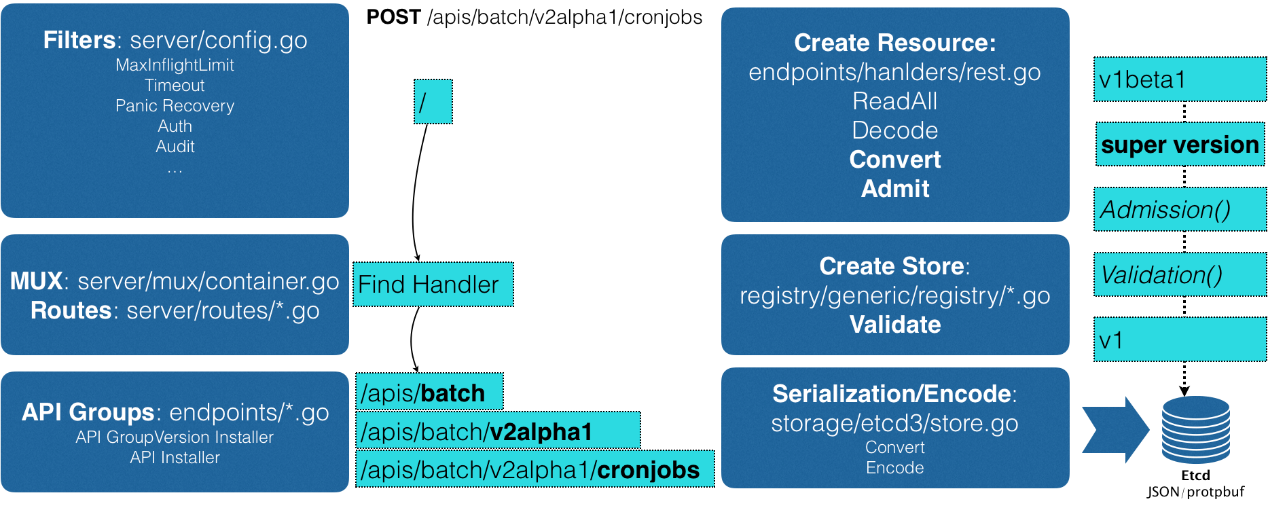


**For example，**

**D:\Users\sunxk\AppData\Local\Temp\1555638151(1).png**

**“CronJob”是这个API对象的Resource，batch是Group，v2alpha1是Version**

**Kubernetes的APIServer通过匹配API对象的组、版本号和资源类型来创建这个对象，创建过程如下：**

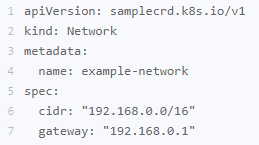


* 1. **当我们发起穿件CronJob的POST请求后，编写的YAML信息提交到APIServer，APIServer过滤这个请求，并完成一些前置性工作（授权、超时处理、审计等）；**
  2. **请求进入MUX和Routes流程，完成URL和Handler的绑定，机找到对应的CronJob的类型定义；**
  3. **APIServer根据CronJob的类型定义，使用用户提交的YAML文件里的字段创建一个CronJob对象，具体：把YAML文件转换成一个叫做Super Version的对象，即该API资源类型所有版本的字段全集（这样，用户提交不同版本的YAML文件都可以通过这个Super Version对象来处理）；**
  4. **APIServer先后进行Admission()和Validation()操作。Admission负责Dynamic Admission Controll （Initializer，为Pod注入一些基础容器和配置，如container、volume，进行TwoWayMergePatch操作，类似get merge），而Validation负责验证这个对象里的各个子弹是否合法，会创建一个Store（APIServer里叫作Registry的数据结构）保存验证通过的有效API对象；**
  5. **APIServer把验证过的API对象转换成用户最初提交的版本，进行序列化编码操作，并调用Etcd的API保存到Etcd。**

**CRD（Custom Resource Definition）**

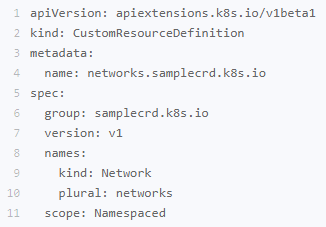
**在Kubernetes中添加自定义API资源要通过Go语言代码生成，使用k8s.io/code-generator的工具对Go编写的资源对象来生成。过程如下：**

**1.example-network.yaml使用samplecrd.k8s.io/v1中自定义的API对象NetWork**

****

使用NetWork对象中的cidr和gateway字段

**2.使用“自定义API资源”—CR（Custom resource）,则需要kubernetes了解对应的CRD（Custom resource Definition），编写对应的network.yaml文件**

****

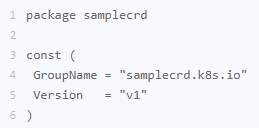
声明scope是Namespaced，即定义的这个Network是属于Namespace的对象，类似Pod

1. **使用Go语言对Network对象进行定义。在$GOPATH（go的workplace）创建一个结构如下的项目：**

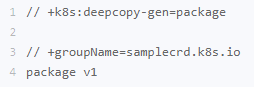
****

**关键在pkg/apis下，samplecrd就是API的组名（Group）,v1是版本（Version），types.go文件则定义Network对象(Resource)的完整描述。**

**register.go文件下定义全局变量，组名和版本，即yaml文件中使用到的apiVersion**

****

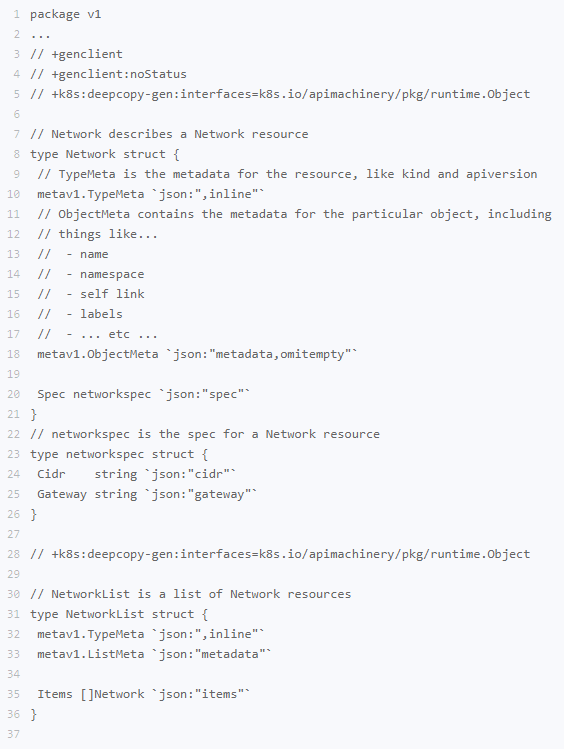
1. **doc.go文件（Golang的文档源文件），注释用于全局的代码生成控制**

****

为整个v1包里所有类型定义自动生成DeepCopy方法；

定义这个包对应的API组名

1. **types.go文件，定义Network类型中包含的字段，spec中的内容**

****

↑在生成DeepCopy时实现Kubernetes提供的runtime.Object接口（固定操作）

+genclient只需要写在Network类型上

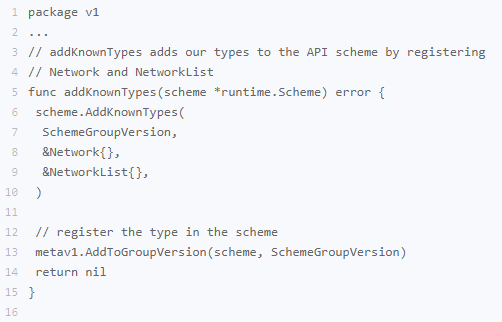
+genclient:noStatus表示定义的类型中没有Status字段，否则生成的Client会自带上UpdateStatus方法

**Network类型中定义TypeMeta（API元数据）、ObjectMeta（对象元数据）和Spec。Spec字段需要自定义一个类型networkspec， 再定义具体包含的字段 （Cidr和GateWay， 后面带的`json:“cidr”`表示这个字段被转换成JSON格式后的名字，即yaml文件中的字段名）**

**NetworkList类型表示一组Network对象应包括哪些字段，供Kubernetes中获取某个对象的List()方法，返回值为List类型**

1. **register.go文件**

**“registry”的作用是注册一个类型（Type）给APIServer。其中，Network资源类型在服务器端的注册工作由APIServer完成，而客户端“知道”对应资源类型的定义则需要通过register.go文件，其中最主要的方法addKnownTypes()，如下**

****

1. **通过Kubernetes提供的代码生成工具k8s.io/code-generator，为上面定义的Network资源类型自动生成clientset、informers和listers，自动生成的DeepCopy代码文件zz\_generated.deepcopy.go**

**代码生成后项目的目录结构，变为以下**

# 代码生成的工作目录，也就是我们的项目路径

$ ROOT\_PACKAGE="github.com/resouer/k8s-controller-custom-resource"

# API Group

$ CUSTOM\_RESOURCE\_NAME="samplecrd"

# API Version

$ CUSTOM\_RESOURCE\_VERSION="v1"

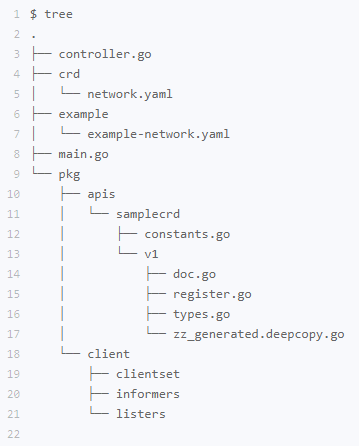
# 安装 k8s.io/code-generator

$ go get -u k8s.io/code-generator/...

$ cd $GOPATH/src/k8s.io/code-generator

# 执行代码自动生成，其中 pkg/client 是生成目标目录，pkg/apis 是类型定义目录

$ ./generate-groups.sh all "$ROOT\_PACKAGE/pkg/client" "$ROOT\_PACKAGE/pkg/apis" "$CUSTOM\_RESOURCE\_NAME:$CUSTOM\_RESOURCE\_VERSION"

****

1. **通过crd目录下network.yaml就可以在kubernetes中创建一个Network类型的API对象的CRD**

**$ kubectl apply –f crd/network.yaml**

**然后，就可以创建一个Network对象**

**$ kubectl apply –f example/example-network.yaml**

**声明式API=创建一个自定义API对象+为这个API对象编写一个自定义控制器**

**自定义控制器Custom Controller**

****

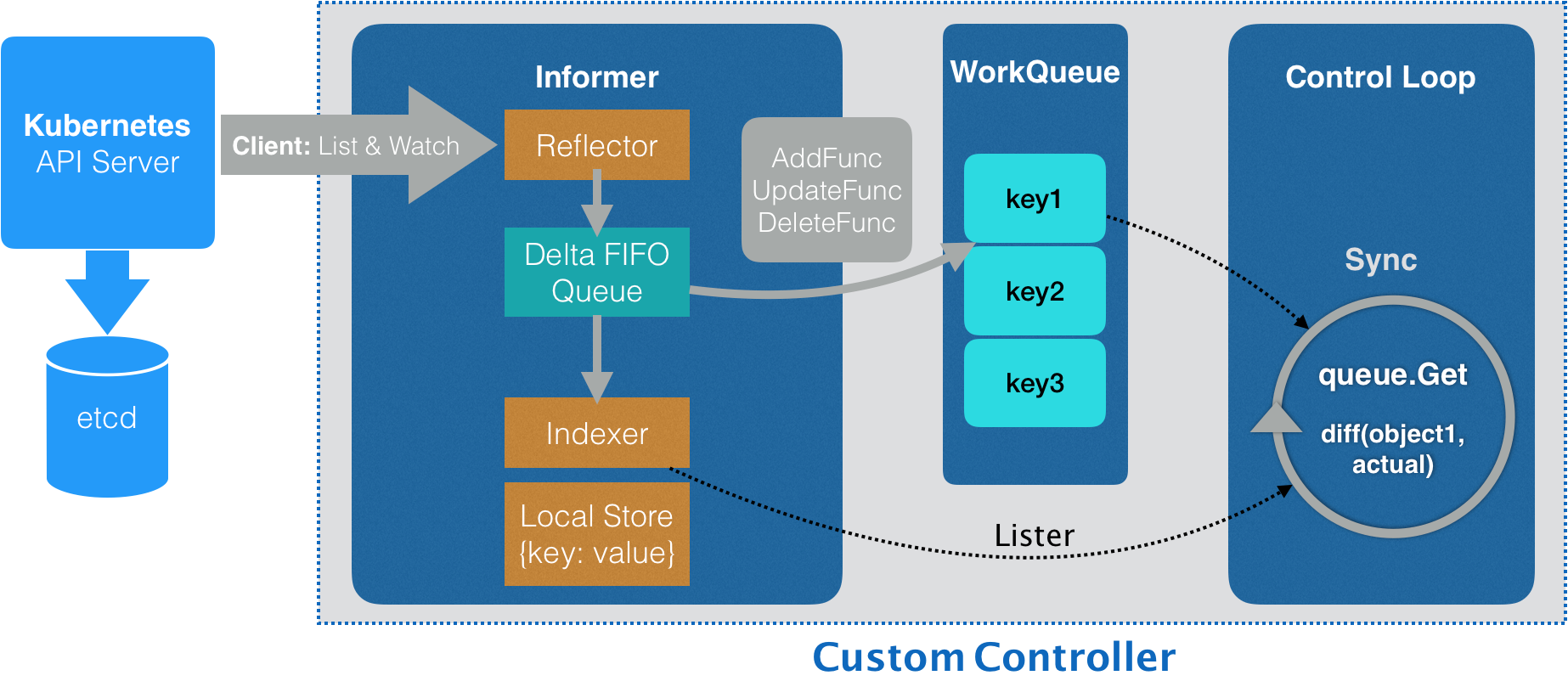
**基于之前定义的Network对象，通过main函数定义并初始化一个自定义控制器：**

1. **根据Master的配置（APIServer的地址端口和kubeconfig的路径），创建一个Kubernetes的client（kubeClient）和Network对象的client（networkClient）**

**Ps.若无master配置，main函数会直接使用一种名叫InClusterConfig的方式来创建这个client，并假设自定义控制器是以Pod的方式运行，并且会直接使用默认ServiceAccount数据卷里的授权信息来访问APIServer**

1. **Main函数为Network对象创建一个叫做InformerFactory（networkInformerFactory）的工厂，并使用它生成一个Network对象的Informer，传递给控制器；**
2. **启动Informer，然后执行controller.Run启动自定义控制器**

**自定义控制器的工作原理**

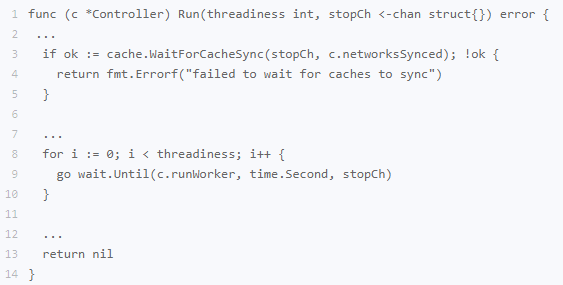


1. **Informer通过Client（networkClient）与APIServer建立连接，并通过Reflector来维护这个连接，使用ListAndWatch机制，对APIServer发来的实例对象被创建、删除或更新的事件消息进行处理。事件及其对应的API对象的组合被称为增量（Delta），被放入到Delta FIFO Queue中，Informer再从这个Queue中读取delta来创建或更新本地对象的索引（Indexer）和缓存（local Store）**
2. **Informer根据事件的类型，对应触发实现注册好的ResourceEventHandler（AddFunc、UpdateFunc和DeleteFunc）**
3. **工作队列WorkQueue负责同步Informer和控制循环（Control Loop）之间的数据，将具体处理事件对应的API对象的Key加入到工作队列**
4. **定时resync：每经过resyncPeriod指定的时间，Informer维护的本地缓存，都会使用最近一次LIST返回的结果强制更新一次，以保证缓存的有效性**
5. **控制循环的启动：等待Informer完成一次本地缓存的数据同步操作，然后通过goroutine启动一个（或并发多个）“无限循环”的任务**

**控制器的定义**

****

**controller对应的Run方法：**

****

**runWorker“无限循环”的具体逻辑：**

func (c \*Controller) runWorker() {

for c.processNextWorkItem() {

}

}

func (c \*Controller) processNextWorkItem() bool {

obj, shutdown := c.workqueue.Get()

...

err := func(obj interface{}) error {

...

if err := c.syncHandler(key); err != nil {

return fmt.Errorf("error syncing '%s': %s", key, err.Error())

}

c.workqueue.Forget(obj)

...

return nil

}(obj)

...

return true

}

func (c \*Controller) syncHandler(key string) error {

namespace, name, err := cache.SplitMetaNamespaceKey(key)

...

network, err := c.networksLister.Networks(namespace).Get(name)

if err != nil {

if errors.IsNotFound(err) {

glog.Warningf("Network does not exist in local cache: %s/%s, will delete it from Neutron ...",

namespace, name)

glog.Warningf("Network: %s/%s does not exist in local cache, will delete it from Neutron ...",

namespace, name)

// FIX ME: call Neutron API to delete this network by name.

// neutron.Delete(namespace, name)

return nil

}

...

return err

}

glog.Infof("[Neutron] Try to process network: %#v ...", network)

// FIX ME: Do diff().

//

// actualNetwork, exists := neutron.Get(namespace, name)

//

// if !exists {

// neutron.Create(namespace, name)

// } else if !reflect.DeepEqual(actualNetwork, network) {

// neutron.Update(namespace, name)

// }

return nil

}

**在执行周期（processNextWorkItem）中，首先从工作队列里出队（workqueue.Get）一个成员（Key：Network对象的namespace/name）,然后在syncHandler方法中通过这个Key尝试从Informer维护的缓存中获取对应的Network对象。**

**若控制循环从缓存中拿不到这个对象（networkLister返回了IsNotFound的错误），意味这个Network对象的Key是通过“删除”事件添加进工作队列的（对象已被删除，但Key还在队列中），此时调用Neutron的API把这个Key所对应的Neutron网络从真实的集群中删除。**

**此处若获取到了Network对象，则是APIServer里保存的“期望状态”，即用户通过YAML文件提交到APIServer里的信息。而“实际状态”则需要通过控制循环调用Neutron API来查询。通过对比二者的状态的差异，来创建或更新以完成一次调谐（reconcile）的过程。**

**基于角色的权限控制RBAC(Role-Based Access Control)**