简介

Kubernetes提供的集群控制平面（master节点）与Kubernetes APIServer通信的命令行工具——kubectl。kubectl默认配置文件目录$HOME/.kube/config。可以通过 --kubeconfig 参数来指定kubectl的配置文件。

素材准备

Replication Controller

```
apiVersion: v1
kind: ReplicationController     # 副本控制器 RC
metadata:
  name: myhello-rc              # RC名称，全局唯一
  labels:
    name: myhello-rc
spec:
  replicas: 5                   # Pod副本期待数量
  selector:
    name: myhello-rc-pod
  template:
    metadata:
      labels:
        name: myhello-rc-pod
    spec:
      containers:                          # Pod 内容的定义部分
      - name: myhello                     #容器的名称
        image: xlhmzch/hello:1.0.0         #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
```

```
apiVersion: v1
kind: Service
metadata:
  name: myhello-svc
  labels:
    name: myhello-svc
spec:
  type: NodePort # 对外提供端口
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    name: http
```

```
    nodePort: 30000
  selector:
    name: myhello-rc-pod
```

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    name: myapp-deploy
spec:
  replicas: 5
  selector:
    matchLabels:
      name: myapp-deploy-pod
  template:
    metadata:
      labels:
        name: myapp-deploy-pod
    spec:
     #nodeSelector:
       #nodetype: worker
      containers:                          # Pod 内容的定义部分
      - name: myhello                     #容器的名称
        image: xlhmzch/hello:1.0.0        #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        resources:
          requests:
            cpu: 100m
      - name: myredis                     #容器的名称
        image: redis       #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 6379
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        resources:
          requests:
            cpu: 100m
```

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

```
metadata:
  name: myapp-svc
  labels:
    name: myapp-svc
spec:
  type: NodePort # 对外提供端口
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    name: http
    nodePort: 30001
  selector:
    name: myapp-deploy-pod
```

DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: myapp-ds
  namespace: default
  labels:
    app: myapp-ds
spec:
  selector:
    matchLabels:
      app: myapp-ds
  template:
    metadata:
      labels:
        app: myapp-ds
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:                          # Pod 内容的定义部分
      - name: myhello                     #容器的名称
        image: xlhmzch/hello:1.0.0        #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
```

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-ds-svc
  labels:
```

```
    name: myapp-ds-svc
spec:
  type: NodePort # 对外提供端口
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 80
    name: http
    nodePort: 30002
  selector:
    app: myapp-ds
```

kubectl 命令自动补全

```
# 安装自动补全插件
sudo apt-get install -y bash-completion
# 添加.bashrc文件内容
echo "source <(kubectl completion bash)" >> ~/.bashrc
# 加载最新的.bashrc
source ~/.bashrc
```

kubectl语法

```
kubectl [command] [TYPE] [NAME] [flags]
```

1. command：指定要对一个或多个资源执行的操作，例如：create、get、describe、delete等。
2. TYPE：指定 资源类型。资源类型**不区分大小写**，可以指定单数、复数或缩写形式。例如：以下命令输出相同的结果：

```
kubectl get pod pod1
kubectl get pods pod1
kubectl get po pod1
```

3. NAME：指定资源名称。**名称区分大小写。**如果省略名称，这显示所有资源的详细信息。例如：kubectl get pods 。
4. 在对多个资源执行操作时，可以按类型和名称指定每个资源，或指定一个或多个文件：

a. 按类型和名称指定资源：

- 对所有类型相同的资源进行分组，TYPE1 name1 name2 name<#>。

例如：

```
kubectl get pod pod1 pod2 pod3
```

- 分别指定多个资源类型：TYPE1/name1 TYPE2/name2 TYPE3/name3 TYPE<#>/name<#>。

例如：

```
kubectl get pod/pod1 pod/pod2 rc/rc1 svc/svc1
```

b. 用一个或多个文件指定资源：-f file1 -f file2 -f file<#>
例如：

```
 kubectl get -f myhello-rc.yaml  -f myhello-svc.yaml
```

5. flags：指定可选的参数。例如：可以使用 -o 或 --output 参数来指定 输出格式。

例如：

```
kubectl get -f myhello-rc.yaml  -f myhello-svc.yaml -o wide
kubectl get -f myhello-rc.yaml  -f myhello-svc.yaml --output json
```

基础操作命令

api-resources

打印服务器上所支持的 API 资源

1. 用法

```
# 打印支持的API Resource
kubectl api-resources

# 打印更多信息
kubectl api-resources -o wide

# 根据name 排序
kubectl api-resources --sort-by=name

# 仅打印支持命名空间的资源
kubectl api-resources --namespaced=true

# 打印不支持命名空间的资源
kubectl api-resources --namespaced=false

# 根据分组打印该分组的资源，例如：apps、authorization.k8s.io 等
kubectl api-resources --api-group=apps
```

api-versions

打印支持的API Versions

```
kubectl api-versions
```

create

通过文件或标准输入来创建资源

1. 用法

```
kubectl create -f FILENAME
```

2. 案例

```
# 基于文件创建资源
kubectl create -f myhello-rc.yaml -f myhello-svc.yaml -f myapp-deployment.yaml -f
myapp-svc.yaml -f myapp-ds.yaml

# 将文件内容以标准输入，传入kubectl create
cat myhello-rc.yaml | kubectl create -f -

# 编辑文件，以编辑结果为输入参数
kubectl create -f myhello-rc.yaml --edit -o json
```

expose

给定副本控制器、服务、Deployment 或 Pod，将其暴露为新的 kubernetes Service，其本质是通过现有资源对象的配置信息将新的Service与原有资源背后的Pod做关联

1. 用法

```
 kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol=TCP|UDP|SCTP]
[--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-
service] [--type=type]
```

2. 案例

```
# 为副本控制器myhello-rc 创建service，端口为8000，容器端口为80
kubectl expose rc myhello-rc --port=8000 --target-port=80 --name=myhello-svc-
8000

# 通过replication controller 定义文件来创建service
kubectl expose -f myhello-rc.yaml --port=8000 --target-port=80 --name=myhello-
svc-8000-file

# 根据指定的pod 创建service，并指定service名称
kubectl expose pod <podname> --port=444 --name=myhello-svc-444

# 通过service 创建新的service
kubectl expose service myhello-svc --port=8080 --target-port=80 --name=myhello-
svc-8080
```

```
# 查看所有service
kubectl get svc

# 查看某个service详情
kubectl describe svc <svcname>
```

run

在集群中使用指定镜像启动容器

1. 用法

```
kubectl run NAME --image=image [--env="key=value"] [--port=port] [--dry-
run=server|client] [--overrides=inline-json] [--command] -- [COMMAND] [args...]
```

2. 案例

```
# 通过镜像运行一个名为 nginx 的pod
kubectl run nginx --image=nginx

# 通过镜像运行一个名为myhello 的pod
kubectl run myhello --image=xlhmzch/hello:1.0.0

# 指定端口号、环境变量、labels
kubectl run redis --image=redis --port=6379 \
--env="DNS_DOMAIN=cluster" --env="POD_NAMESPACE=default" \
--labels="app=redis,env=prod"

# 只打印命令执行的结果，而不做实际操作
kubectl run nginx --image=nginx --dry-run=client

# 运行一个busybox容器，并进入交互，同时设置其重启策略为Never
kubectl run -i -t busybox --image=busybox --restart=Never

# 启动nginx pod 采用自定义启动命令和启动参数
# kubectl run nginx --image=nginx --command -- <cmd> <arg1> ... <argN>
kubectl run nginx2 --image=nginx --command -- echo 123456
```

```
# 查看默认命名空间下所有pod
kubectl get pod

# 获取单个pod并以yaml格式输出
kubectl get pod <podname> -o yaml

# 输出某个pod内所有容器的日志
kubectl logs pod/<podname> --all-containers=true

# 输出 pod/myhello-rc-dtshm 内 myhello容器的日志
kubectl logs pod/myhello-rc-dtshm  -c myhello
```

set

为对象设置功能特性（环境变量、镜像等）

1. 用法

```
kubectl set SUBCOMMAND
```

env

更新资源环境变量，支持pod（po）、replicationcontroller（rc）、部署（deploy）、守护程序集（ds）、状态集（sts）、cronjob（cj）、ReplicateSet（rs）等资源对象的更新

1. 用法

```
kubectl set env RESOURCE/NAME KEY_1=VAL_1 ... KEY_N=VAL_N
```

2. 案例

```
# 为rc/myhello-rc 添加环境变量 STORAGE_DIR=/local
kubectl set env rc/myhello-rc STORAGE_DIR=/local

# 查看 rc/myhello-rc 环境变量列表
kubectl set env rc/myhello-rc --list

# 列出所有pod的环境变量列表
kubectl set env pods --all --list

# 设置环境变量，并以yaml格式打印出来
kubectl set env rc/myhello-rc STORAGE_DIR=/data1 -o yaml

# 为所有rc设置环境变量为 EVN=prod
kubectl set env rc --all ENV=prod

# 将所有rc上环境变量 ENV移除
kubectl set env rc --all ENV-

# 移除 deployments/myapp-deployment 对象中，容器名为 myhello 的环境变量 env1
kubectl set env deployments/myapp-deployment --containers="myhello" env1-

# 根据文件，移除资源上的环境变量
kubectl set env -f myhello-rc.yaml env1-
```

注意：rc上修改环境变量，并不会及时反映到Pod；而deployment修改环境变量则会及时反映到Pod

image

更新现有资源容器镜像，支持pod（po）、replicationcontroller（rc）、部署（deploy）、守护程序集（ds）、状态集（sts）、cronjob（cj）、ReplicatSet（rs）等资源

1. 用法

```
kubectl set image (-f FILENAME | TYPE NAME) CONTAINER_NAME_1=CONTAINER_IMAGE_1
... CONTAINER_NAME_N=CONTAINER_IMAGE_N
```

2. 案例

```
# 将deployment/myapp-deployment 中myhello 容器的镜像更改为 xlhmzch/hello:1.0.1
kubectl set image deployment/myapp-deployment myhello=xlhmzch/hello:1.0.1

# 根据文件修改镜像，--local表示不向apiserver发送请求，仅本地修改并输出yaml格式内容
kubectl set image -f myapp-deployment.yaml myhello=xlhmzch/hello:1.0.1 --local -o
yaml

# 修改 deployment/myapp-deployment myhello 容器和myredis容器的镜像
kubectl set image deployment/myapp-deployment myhello=xlhmzch/hello:1.0.1
myredis=redis:alpine

# 修改ds/myapp-ds myhello 容器镜像
kubectl set image ds/myapp-ds myhello=xlhmzch/hello:1.0.2
```

注意：修改RC的image不会及时反映到Pod；而修改deployment的image则会及时反映到Pod

resources

为Pod模板资源对象指定计算资源需求（CPU，内存等）支持pod（po）、replicationcontroller（rc）、部署（deploy）、守护程序集（ds）、状态集（sts）、cronjob（cj）、ReplicatSet（rs）等资源

1. 用法

```
kubectl set resources (-f FILENAME | TYPE NAME) ([--limits=LIMITS & --
requests=REQUESTS]
```

2. 案例

```
# 设置单个容器的资源
kubectl set resources deployment myapp-deployment -c=myhello --
limits=cpu=200m,memory=512Mi

# 设置myapp-deployment部署下所有容器资源
kubectl set resources deployment myapp-deployment --limits=cpu=200m,memory=512Mi
--requests=cpu=100m,memory=256Mi

# 移除资源限制，设置为0即不限制
kubectl set resources deployment myapp-deployment --limits=cpu=0,memory=0 --
requests=cpu=0,memory=0

# 根据文件设置资源，打印出结果，且不向apiserver发送请求
kubectl set resources -f myapp-deployment.yaml --limits=cpu=200m,memory=512Mi --
local -o yaml
```

selector

在资源上设置选择器。如果资源在调用“set selector”之前有一个选择器，则新选择器将覆盖旧选择器。如果指定了--resource version，则更新将使用此资源版本，否则将使用现有资源版本。目前只支持Service资源对象

1. 用法

```
kubectl set selector (-f FILENAME | TYPE NAME) EXPRESSIONS [--resource-
version=version]
```

2. 案例

```
# 将service  myhello-svc 的label selector 修改为 env=proc
kubectl set selector svc myapp-svc env=proc

kubectl set selector svc myapp-svc name=myapp-deploy-pod
```

```
# 查看service 对象明细
kubectl describe svc myapp-svc
```

explain

显示资源文档说明。

1. 用法

```
kubectl explain RESOURCE
```

2. 案例

```
# 获取资源及其字段的文档
kubectl explain pods

获取资源特定字段的文档
kubectl explain pods.spec.containers
```

get

显示一个或者多个资源信息

1. 用法

```
 kubectl get [(-o|--output=)json|yaml|name|go-template|go-template-
file|template|templatefile|jsonpath|jsonpath-as-json|jsonpath-file|custom-
columns|custom-columns-file|wide] (TYPE[.VERSION][.GROUP] [NAME | -l label] |
TYPE[.VERSION][.GROUP]/NAME ...) [flags]
```

2. 案例

```
# 获取默认命名空间下所有Pod
kubectl get pods

# 获取更多Pod信息
kubectl get pods -o wide

# 获取副本控制器 myhello-rc
kubectl get replicationcontroller myhello-rc

# 获取处于apps.v1 api组下所有 deployment，并以json格式答应
```

```
kubectl get deployments.v1.apps -o json

# 获取单个Pod，以json格式数据
kubectl get -o json pod myhello-rc-278jg

# 根据文件获取对象
kubectl get -f myapp-deployment.yaml -o json

# 返回对象指定的值
kubectl get -o template deployment/myapp-deployment --template=
{{.status.readyReplicas}}

# 自定义返回字段，分别指定了列名为：CONTAINER 和 IMAGE
kubectl get pod  -o custom-
columns=CONTAINER:.spec.containers[0].name,IMAGE:.spec.containers[0].image

# 获取 所有rc 和 service
kubectl get rc,services

# 获取一个或多个资源
kubectl get rc/myhello-rc service/myhello-svc deployment/myapp-deployment
```

edit

修改服务器上的某资源

1. 用法

```
kubectl edit (RESOURCE/NAME | -f FILENAME)
```

2. 案例

```
# 修改myhello-svc service
kubectl edit svc/myhello-svc

# 以打开为json文件的方式修改
kubectl edit svc/myhello-svc -o json

# 修改资源配置，并将修改后的内容添加到注解
kubectl edit deployment/myapp-deployment -o yaml --save-config
```

delete

通过文件名、标准输入、资源和名字删除资源，或者通过资源和标签选择器来删除资源

1. 案例

```
# 通过定义文件删除资源
kubectl delete -f myhello-rc.yaml

# 指定所有后缀名为yaml的文件，删除这些文件定义的资源
kubectl delete -f '*.yaml'

# 以文件内容为参数，删除资源
```

```
cat myhello-rc.yaml | kubectl delete -f -

# 删除名称为 myhello-pod 或 myhello-svc 的pod 和 service
kubectl delete pod,service myhello-pod myhello-svc

# 删除标签为 myhello-pod的pod和service
kubectl delete pods,services -l name=myhello-rc-pod

# 最小延迟删除Pod
kubectl delete pod <podname> --now

# 强制删除 Pod
kubectl delete pod <podname> --force

# 删除所有Pod
kubectl delete pods --all
```

label

更新资源的标签

1. 用法

```
 kubectl label [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ...
KEY_N=VAL_N [--resource-version=version]
```

2. 案例

```
# 为pod/nginx添加标签
kubectl label pods nginx status=unhealthy

# 修改已存在的label
kubectl label --overwrite pods nginx status=healthy

# 为当前命名空间下所有pod添加标签
kubectl label pods --all status=unhealthy

# 根据资源定义文件添加标签
kubectl label -f myapp-deployment.yaml status=unhealthy

# 删除当前命名空间下所有Pod的status标签
kubectl label pods --all status-

# 删除指定label
kubectl label pods nginx status-

# 根据资源定义文件删除资源标签
kubectl label -f myapp-deployment.yaml status-
```

3. 案例

```
# 为节点添加标签
kubectl label node k8s-master1 nodetype=master
kubectl label node k8s-node1 nodetype=worker
kubectl label node k8s-node2 nodetype=worker
kubectl label nodes k8s-node2 nodetype-
```

```
# 为pod指定部署的节点
nodeSelector:
  nodetype: worker
```

annotate

更新资源所关联的注解

1. 用法

```
kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ...
KEY_N=VAL_N [--resource-version=version]
```

2. 案例

```
# 为资源类型为 rc 的 myhello-rc 添加注解 description='my hello rc'
kubectl annotate rc myhello-rc description='my hello rc'

# 为文件myhello-rc.yaml所定义的资源 添加注解 description1='my hello rc1'
kubectl annotate -f myhello-rc.yaml description1='my hello rc1'

# 重写 myhello-rc 的description 注解
kubectl annotate --overwrite rc myhello-rc description='this a replication
controller'

# 为当前命名空间下所有pod 添加注解
kubectl annotate pods --all description='myhello running golang program'

# 更新指定resourceVersion 的单一资源对象
kubectl annotate rc myhello-rc description3='my hello rc3' --resource-
version=10564

# 删除注解
kubectl annotate pods --all description-
```

应用部署命令

diff

显示目前版本与将要应用的版本之间的差异，仅对比yaml文件所定义的项目

1. 用法

```
kubectl diff -f FILENAME
```

2. 案例

```
# 通过文件对比
kubectl diff -f  myapp-deployment.yaml

# 通过输入对比
cat myapp-deployment.yaml | kubectl diff -f -

# 对比当前目录yaml后缀的文件
kubectl diff -f '*.yaml'
```

apply

基于文件或标准输入，将新的配置应用到资源上

1. 用法

```
 kubectl apply -f FILENAME
```

2. 案例

```
# 将配置应用到资源
kubectl apply -f myapp-deployment.yaml

# 通过输入的方式讲配置应用到资源
cat myapp-deployment.yaml | kubectl apply -f -

# 将当前目录yaml后缀的文件应用到资源
kubectl apply -f '*.yaml'
```

replace

基于文件或标准输入，将新的配置已替换的方式应用到资源上

1. 用法

```
 kubectl replace -f FILENAME
```

2. 案例

```
# 将配置应用到资源
kubectl replace -f myapp-deployment.yaml

# 通过输入的方式讲配置应用到资源
cat myapp-deployment.yaml | kubectl replace -f -
```

rollout

管理资源的上线，支持 deployments、daemonsets、statefulsets等资源对象

1. 用法

```
kubectl rollout SUBCOMMAND
```

history

查看历史修订版本和配置

1. 用法

```
kubectl rollout history (TYPE NAME | TYPE/NAME) [flags]
```

2. 案例

```
# 查看DaemonSet/cadvisor 的发布历史
kubectl rollout history ds/myapp-ds

# 查看修订版本号为3的历史记录详细信息
kubectl rollout history daemonset/myapp-ds --revision=3
```

pause

将提供的资源标记为已暂停。控制器不会协调暂停的资源。使用“kubectl rollout resume”恢复暂停的资源。
当前仅支持 deployment 资源对象，由于deployment的滚动更新机制，如果在部署过程中使用了pause，将会导致一个部署中的pod版本不一致
暂停 Deployment，然后再触发一个或多个更新，最后再继续（resume）该 Deployment。这种做法可以在暂停和继续中间对 Deployment 做多次更新，而无需触发不必要的滚动更新。简而言之：多次修改之后，在执行resume命令之后，对之前的修改一起反映到Pod。但是对服务的扩容和缩容不受暂停约束。

1. 用法

```
 kubectl rollout pause RESOURCE
```

2. 案例

```
# 暂停部署
kubectl rollout pause deployment myapp-deployment
```

resume

恢复暂停的资源。
控制器不会协调暂停的资源。通过恢复资源，我们可以再次协调资源。当前仅支持恢复deployment。

1. 用法

```
 kubectl rollout resume RESOURCE
```

2. 案例

```
kubectl rollout resume deployment myapp-deployment
```

restart

重启资源对象

1. 用法

```
kubectl rollout restart RESOURCE
```

2. 案例

```
# 重启部署
kubectl rollout restart deployment/myapp-deployment

# 重启守护进程
kubectl rollout restart daemonset/myapp-ds

# 根据selector 重启部署
kubectl rollout restart deployment --selector=name=myapp-deploy
```

status

1. 用法

```
 kubectl rollout status (TYPE NAME | TYPE/NAME) [flags]
```

2. 案例

```
# 查看发布状态
kubectl rollout status deployment/myapp-deployment
```

```
nick@k8s-master1:~/work$ kubectl edit deployments.apps myapp-deployment
deployment.apps/myapp-deployment edited
nick@k8s-master1:~/work$ kubectl rollout status deployment/myapp-deployment
Waiting for deployment "myapp-deployment" rollout to finish: 11 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 12 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 13 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 14 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 15 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 16 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 17 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 18 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 19 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 20 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 21 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 22 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 23 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 24 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 25 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 26 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 27 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 28 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 29 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 30 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 31 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 32 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 33 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 34 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 35 of 100 updated replicas are available...
Waiting for deployment "myapp-deployment" rollout to finish: 36 of 100 updated replicas are available...
```

undo

回滚到之前版本

1. 用法

```
kubectl rollout undo (TYPE NAME | TYPE/NAME) [flags]
```

2. 案例

```
# 回滚deployment/myapp-deployment 到上一个版本
kubectl rollout undo deployment/myapp-deployment

# 回滚到指定版本
kubectl rollout undo daemonset/myapp-ds --to-revision=2

# 演习回滚，查看结果。并未做真正的操作
kubectl rollout undo --dry-run=server deployment/myapp-deployment
```

3. 连续的undo，并不会一直往前回滚到很老的版本，而会在最近两个版本间来回切换

```
# 分三次修改镜像版本，分别改为：1.0.0 1.0.1 1.0.2
kubectl edit ds/myapp-ds

# 回滚到上一个版本，查看详情镜像版本为：1.0.1
kubectl rollout undo ds/myapp-ds

# 回滚到上一个版本，查看详情镜像版本为：1.0.2
kubectl rollout undo ds/myapp-ds
```

scale

为deployment、replica set、 replication controller、statefulset 设置pod的副本数。

1. 用法

```
kubectl scale [--resource-version=version] [--current-replicas=count] --
replicas=COUNT (-f FILENAME | TYPE NAME)
```

2. 案例

```
# 修改副本数量为3
kubectl scale --replicas 3 deployment myapp-deployment

# 修改文件定义资源的副本数量为30
kubectl scale --replicas=30 -f myapp-deployment.yaml

# 如果当前副本数为30，则将副本数改为10
kubectl scale --current-replicas=30 --replicas=10 deployment/myapp-deployment

# 将指定 rc 和 deployment的副本数改为6
kubectl scale --replicas=6 rc/myhello-rc deployment/myapp-deployment
```

autoscale

创建自动缩放器，自动选择和设置在Kubernetes群集中运行的POD数。支持 deployment、replica set、stateful set、replication controller等资源对象。当CPU或内存的使用率超过设定值之后，会开始自动扩容。当指标恢复之后，大约5分钟后，会开始缩容。自动伸缩的支持，必须为pod中每个容器设置所需最小资源

1. 用法

```
kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=MINPODS] --
max=MAXPODS [--cpu-percent=CPU]
```

2. 案例

```
# 最少2个pod ,最多10个pod，采用默认缩放策略
kubectl autoscale deployment myapp-deployment --min=2 --max=10

# 最多15个pod，目标pod cpu利用率40%
kubectl autoscale deployment myapp-deployment --min=2 --max=15 --cpu-percent=40
```

```
# 查看自动扩展器
kubectl get horizontalpodautoscalers
```

3. metrics server

自动伸缩，必须安装metrics server。metrics server 用于获取节点指标。metrics server安装条件，k8s集群必须开启聚合层（默认已配置）；节点kubelet 服务启用webhook鉴权（默认已启用）；metrics server 启动项添加 --kubelet-insecure-tls 选项。

文档:

metrics server: https://github.com/kubernetes-sigs/metrics-server#readme

k8s 聚合层：https://kubernetes.io/zh-cn/docs/tasks/extend-kubernetes/configure-aggregation-layer/

k8s扩展服务：https://kubernetes.io/zh-cn/docs/tasks/extend-kubernetes/setup-extension-api-server/

k8s webhook鉴权：https://kubernetes.io/zh-cn/docs/reference/access-authn-authz/kubelet-authn-authz/

扩缩策略：https://kubernetes.io/zh-cn/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-policies

4. metrics server 安装

components.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
    rbac.authorization.k8s.io/aggregate-to-view: "true"
  name: system:aggregated-metrics-reader
```

```
rules:
- apiGroups:
  - metrics.k8s.io
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
rules:
- apiGroups:
  - ""
  resources:
  - nodes/metrics
  verbs:
  - get
- apiGroups:
  - ""
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
```

```
  name: metrics-server:system:auth-delegator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    k8s-app: metrics-server
  name: system:metrics-server
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-server
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: https
  selector:
    k8s-app: metrics-server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  strategy:
    rollingUpdate:
      maxUnavailable: 0
```

```
  template:
    metadata:
      labels:
        k8s-app: metrics-server
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --kubelet-insecure-tls
        - --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        image: registry.aliyuncs.com/google_containers/metrics-server:v0.6.2
        imagePullPolicy: IfNotPresent
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /livez
            port: https
            scheme: HTTPS
          periodSeconds: 10
        name: metrics-server
        ports:
        - containerPort: 4443
          name: https
          protocol: TCP
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /readyz
            port: https
            scheme: HTTPS
          initialDelaySeconds: 20
          periodSeconds: 10
        resources:
          requests:
            cpu: 100m
            memory: 200Mi
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true
          runAsNonRoot: true
          runAsUser: 1000
        volumeMounts:
        - mountPath: /tmp
          name: tmp-dir
      nodeSelector:
        kubernetes.io/os: linux
      priorityClassName: system-cluster-critical
      serviceAccountName: metrics-server
      volumes:
      - emptyDir: {}
        name: tmp-dir
---
```

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  labels:
    k8s-app: metrics-server
  name: v1beta1.metrics.k8s.io
spec:
  group: metrics.k8s.io
  groupPriorityMinimum: 100
  insecureSkipTLSVerify: true
  service:
    name: metrics-server
    namespace: kube-system
  version: v1beta1
  versionPriority: 100
```

```
kubectl apply -f components.yaml
```

集群管理

top

显示节点或pod的资源（cpu/memory）使用情况，必须安装metrics server

```
# 显示所有node资源使用情况
kubectl top node
# 显示某个node（k8s-master1）资源显示情况
kubectl top node k8s-master1
# 显示label 为nodetype=master 的节点资源使用情况
kubectl top node -l nodetype=master
```

```
# 显示所有pod 资源使用情况
kubectl top pod
# 显示所有容器资源使用情况
kubectl top pod --containers
# 显示label 为name=myapp-deploy-pod 的pod资源使用情况
kubectl top pod -l name=myapp-deploy-pod
```

cordon

标记节点为不可调度

```
# 将节点设置为不可调度
kubectl cordon <nodename>
```

uncordon

标记节点为可调度的

```
# 将节点设置为可调度
kubectl uncordon <nodename>
```

drain

腾空节点，准备维护。该操作会自动将节点设置为不可调度状态

```
# 腾空指定节点，包括未声明控制器的POD,忽略守护守护进程POD
kubectl drain <nodename> --force --ignore-daemonsets
# 指定宽限期，即pod正常结束的实践，若超过该时间将会被强制杀死
kubectl drain <nodename> --force --ignore-daemonsets --grace-period=60
```

taint

更新一个或多个节点污点。污点（taint）和 容忍度（toleration）相互配合，可以用来避免pod被分配到不合适的节点。污点格式：key=value:effect，key和value用户自定义，effect取值范围为：NoSchedule, PreferNoSchedule 或 NoExecute。
NoSchedule 表示Kubernetes不会将pod调度到该节点，但是已经在该节点上运行的pod不受影响。
PreferNoSchedule 表示Kubernetes会尽量避免将pod调度到该节点，但不是强制的
NoExecute 表示Kubernetes不会将pod调度到该节点并且会将该节点上的pod驱逐

污点设置

```
# 添加污点key=key1,effect=NoSchedule
kubectl taint node k8s-node1 key1:NoSchedule

# 添加污点 key=key1,value=value1,effect=PreferNoSchedule
kubectl taint node k8s-node1 key1=value1:PreferNoSchedule

# 添加污点 key=key1,value=value1,effect=NoExecute
kubectl taint node k8s-node1 key1=value1:NoExecute

# 修改已存在的污点
kubectl taint node k8s-node1 --overwrite key1=v1:NoSchedule

# 删除污点
kubectl taint node k8s-node1 key1=v1:NoSchedule-
kubectl taint node k8s-node1 key1-
```

容忍度使用

节点设置污点后，根据情况会出现pod不被调度到该节点或者从该节点上驱逐等情况。若pod想继续在有污点的节点上运行，则必须设置容忍度，容忍节点所有污点。tolerations 字段设置pod容忍度。

```
tolerations:
- key: k1
  operator: "Equal"
  value: v1
  effect: NoExecute
  tolerationSeconds: 3600
- key: k2
  operator: "Exists"
  effect: NoSchedule
- effect: NoSchedule
  operator: "Exists"
```

容忍度与污点通过key、value、effect三个字段来匹配。operator 表示匹配规则，Equal 表示 容忍度与污点key、value、effect必须相等，才能容忍该污点；Exists 表示容忍度与任意的key、value、effect都匹配。
以上配置解读：
第一条：容忍key值为k1且value值为v1且effect值为NoExecute的污点，容忍时间为3600秒，超时后将被驱逐出该节点；
第二条：容忍key值为k2且effect值为NoExecute的污点；
第三条：容忍effect值为NoExecute的污点。

1. 污点设置

```
kubectl taint node k8s-node1 k1=v1:NoSchedule
kubectl taint node k8s-node1 k2=v2:NoExecute
```

2. pod 容忍度设置

```
# pod 容忍度设置
tolerations:
- key: k1
  operator: "Equal"
  value: v1
  effect: NoSchedule
- key: k2
  operator: "Exists"
  effect: NoExecute
```

故障排除和调试

describe

显示某个资源或某组资源的详细信息

1. 用法

```
kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | -l label] | TYPE/NAME)
```

2. 案例

```
# 显示单个node节点详细信息
kubectl describe nodes k8s-node1

# 显示单个pod详细信息
kubectl describe pods/nginx

# 显示文件描述的资源的详细信息
kubectl describe -f myapp-deployment.yaml

# 显示以k8s开头的节点的详细信息
kubectl describe node k8s

# 显示以myapp-deployment开头的pod的详细信息，pod命名通常与其控制器有关
kubectl describe pods myapp-deployment
```

```
# 显示指定label的pod详细信息
kubectl describe po -l name=myapp
```

logs

输出 pod 中某容器的日志

1. 用法

```
kubectl logs [-f] [-p] (POD | TYPE/NAME) [-c CONTAINER]
```

2. 案例

```
# 运行一个nginx Pod
kubectl run nginx --image=nginx
```

```
# 获取pod第一个容器的日志
kubectl logs nginx

# 获取pod中所有容器的日志
kubectl logs <podname> --all-containers=true

# 获取labels包含 name=myapp的所有pod下的所有容器日志
kubectl logs -l name=myapp --all-containers=true

# 持续输出pod中某个容器产生的日志，容器名为 myhello
kubectl logs -f -c myhello <podname>

# 持续输出labels包含 name=myapp的所有pod下的所有容器日志，最大并发日志请求数为10
kubectl logs -f -l name=myapp --all-containers=true --max-log-requests=10

# 获取最近几行日志
kubectl logs --tail=5 nginx

# 获取最近一个小时的日志
kubectl logs --since=1h nginx

# 获取pod中第一个容器的日志
kubectl logs pod/<podname>

# 获取指定deployment中，第一个pod的指定容器的日志。默认日志请求并发数为5
kubectl logs deployment/myapp-deployment -c myhello
```

3. 其他案例

```
# 输出pod web-1中曾经运行过的，但目前已终止的容器ruby的日志
kubectl logs -p -c ruby web-1
```

attach

连接到一个正在运行的容器

1. 用法

```
kubectl attach (POD | TYPE/NAME) -c CONTAINER
```

2. 案例

```
# 连接到指定pod中正在运行的第一个容器
kubectl attach <podname>

# 连接到指定pod中正在运行容器名为 myhello的容器
kubectl attach <podname> -c myhello

# 连接到指定deployments正在运行的第一个容器
kubectl attach deployments/myapp-deployment
```

exec

在容器中执行相关命令

1. 用法

```
kubectl exec (POD | TYPE/NAME) [-c CONTAINER] [flags] -- COMMAND [args...]
```

2. 案例

```
# 在pod nginx 第一个容器中执行date命令
kubectl exec nginx -- date

# 通过-c 指定容器
kubectl exec <podname> -c myhello -- date

# 传入 ls命令和相关参数
kubectl exec <podname> -c myhello -- ls -al ./

# 通过 -it 开启一个虚拟终端
kubectl exec <podname> -c myhello -i -t -- /bin/sh

# deployment/myapp-deployment第一个容器中执行命令
kubectl exec deployments/myapp-deployment -- date

# svc/myapp-svc 第一个容器中执行命令
kubectl exec svc/myapp-svc -- date
```

port-forward

将一个或者多个本地端口转发到 pod。如果有多个pod符合条件，将自动选择一个pod。当所选pod终止时，转发会话结束，需要重新运行该命令才能恢复转发。

1. 用法

```
kubectl port-forward TYPE/NAME [options] [LOCAL_PORT:]REMOTE_PORT [...
[LOCAL_PORT_N:]REMOTE_PORT_N]
```

2. 案例

```
# 转发本机5000 6000 端口到pod对应端口
kubectl port-forward pod/nginx 5000 6000

# 转发本机 5000 6000 分别到80和81端口
kubectl port-forward pod/nginx 5000:80 6000:81

# 从deployments/myapp-deployment 选择一个pod进行转发操作
kubectl port-forward deployments/myapp-deployment 5000 6000

# 从service中选择第一个pod，并将端口转发到端口名为http的端口上
kubectl port-forward service/myapp-svc 8080:http

# 通过--address指定监听地址
kubectl port-forward --address localhost,192.168.239.142 pod/nginx 8888:80

# 随机一个本地端口，转发到pod的指定端口
kubectl port-forward pod/nginx :80
```

proxy

在本地主机和Kubernetes API服务器之间创建代理服务器或应用程序级网关。它还允许通过指定的HTTP路径提供静态内容。所有传入数据都通过一个端口进入并转发到远程Kubernetes API服务器端口，但与静态内容路径匹配的路径除外

1. 用法

```
kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-
prefix=prefix]
```

2. 案例

```
kubectl proxy --api-prefix=/custom/ --port=8011 --www=$HOME/ --www-
prefix=/static/
```

```
# 通过代理访问apiserver接口
curl http://127.0.0.1:8011/custom/api/v1/pods
# 通过代理访问静态内容
curl http://127.0.0.1:8011/static/
```

cp

将文件和目录拷入/拷出容器

1. 用法

```
 kubectl cp <file-spec-src> <file-spec-dest>
```

2. 案例

```
# 把pod第一个容器中当前工作目录下的app 复制到宿主机的/tmp/app
kubectl cp  <podname>:app /tmp/app

# 把/tmp/app 文件复制到 pod第一个容器当前工作目录下，命名为app1
kubectl cp /tmp/app  <podname>:app1

# 把default命名空间下pod中容器myhello中当前工作目录下的app 复制到宿主机的/tmp/app
kubectl cp  default/<podname>:app /tmp/app -c myhello
```

```
# 查看容器当前工作目录内容
kubectl exec <podname> ls
```

debug

创建用于排查工作负载和节点故障的调试会话

1. 用法

```
kubectl debug (POD | TYPE[[.VERSION].GROUP]/NAME) [ -- COMMAND [args...] ]
```

2. 案例1：共享进程空间

```
# 运行一个nginx pod
kubectl run nginx --image=nginx

# 创建一个新的pod my-debugger 用来调试，
#将原有pod内的容器拷贝到新的pod，并增加一个镜像为ubuntu的容器
# 并且通过进程共享
kubectl debug nginx -it --image=ubuntu --copy-to=my-debugger --share-processes
# 开启另一个终端，可以查看命名空间共享配置
kubectl get pod my-debugger -o yaml | grep shareProcessNamespace
```

调试容器内，执行ps ax可以看到包括被调试容器在内的所有进程

```
nick@k8s-master1:~$ kubectl debug nginx -it --image=ubuntu --copy-to=my-debugger2 --share-processes
Defaulting debug container name to debugger-q6qcj.
If you don't see a command prompt, try pressing enter.
root@my-debugger2:/# ps
    PID TTY          TIME CMD
     41 pts/0    00:00:00 bash
     49 pts/0    00:00:00 ps
root@my-debugger2:/# ps ax
    PID TTY      STAT   TIME COMMAND
      1 ?        Ss     0:00 /pause
      7 ?        Ss     0:00 nginx: master process nginx -g daemon off;
     37 ?        S      0:00 nginx: worker process
     38 ?        S      0:00 nginx: worker process
     39 ?        S      0:00 nginx: worker process
     40 ?        S      0:00 nginx: worker process
     41 pts/0    Ss     0:00 bash
     50 pts/0    R+     0:00 ps ax
```

通过访问/proc/{PID}/root 可访问到其他容器的文件系统，PID为主进程ID

```
root@my-debugger2:/# ls /proc/7/root
bin  boot  dev  docker-entrypoint.d  docker-entrypoint.sh  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

3. 案例2：更改启动命令、容器镜像

```
# 运行一个myhello的pod
kubectl run myhello --image=xlhmzch/hello:1.0.0

# 更改容器启动命令
kubectl debug myhello -it --copy-to=my-debugger1 --container=myhello -- /bin/sh

# 更改容器镜像
kubectl debug myhello -it --copy-to=my-debugger2 --set-
image=myhello=xlhmzch/hello:1.0.1
kubectl debug <podname> -it --copy-to=my-debugger3 --set-
image=myhello=xlhmzch/hello:1.0.1,myredis=redis:alpine

# 复制并注入临时容器，共享进程空间同时修改myhello容器的镜像
kubectl debug myhello -it --copy-to=my-debugger4 --image=busybox --set-
image=myhello=xlhmzch/hello:1.0.1 --share-processes
```

4. 案例3：调试节点

```
# 通过创建Pod来调试节点，Pod将运行在指定的节点上，节点的根文件系统挂载在/host目录下
kubectl debug node/k8s-node1 -it --image=busybox
```

5. 其他，需要为集群开启临时容器等特性功能，否则无法使用以下操作

```
# 直接在指定的pod中创建一个基于busybox的临时容器
kubectl debug <podname> -it --image=busybox

# 直接在指定的pod中创建一个基于自定义镜像的容器，并且为容器指定容器名为debugger
kubectl debug <podname> --image=<cusimage> -c debugger
```