一、Service详解

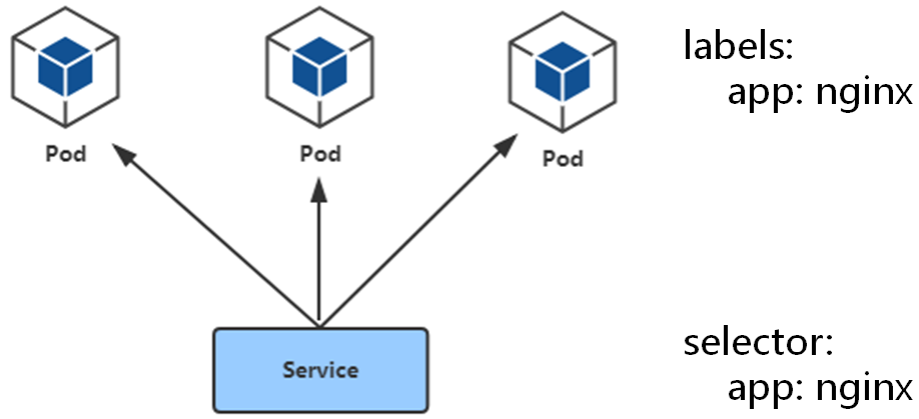
1.Service存在的意义

Pod是短暂的，每次创建/删除都会重新生成新的IP。这就导致了一个问题： 如果一组 Pod他 Pod提供功能或服务， 那要怎么连接固定的IP呢?

- 防止Pod失联（服务发现）
- 定义一组Pod访问方策略（负载均衡）

2.Pod与Service关系

- 通过label-selector相关联
- 通过Service实现Pod的负载均衡（ TCP/UDP 4层）



示例：

查看service信息：

```
kubectl  get svc
```

```
[root@k8s-master2 ~]# kubectl  get svc
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)           AGE
kubernetes   ClusterIP   10.0.0.1      <none>        443/TCP           20d
nginx-web    NodePort    10.0.0.63     <none>        10090:32377/TCP   11d
web-nginx    NodePort    10.0.0.160    <none>        80:30008/TCP      5d23h
[root@k8s-master2 ~]#
```

查看service关联的pod：

在k8s集群中创建一个service，它就自动创建一个对应的endpoint资源。它用来记录一个service对应的所有pod的访问地址，所以查看service关联的pod就可以通过endpoint查看。

```
kubectl  get ep
```

```
[root@k8s-master2 ~]# kubectl  get ep
NAME         ENDPOINTS                                              AGE
kubernetes   192.168.17.100:6443,192.168.17.101:6443                20d
nginx-web    10.244.0.11:80,10.244.1.34:80,10.244.2.16:80 + 1 more...   11d
web-nginx    10.244.0.10:80,10.244.1.35:80,10.244.2.14:80 + 1 more...   5d23h
[root@k8s-master2 ~]#
```

查看service与pod关联关系：

pod与service是通过labe-selector关联的

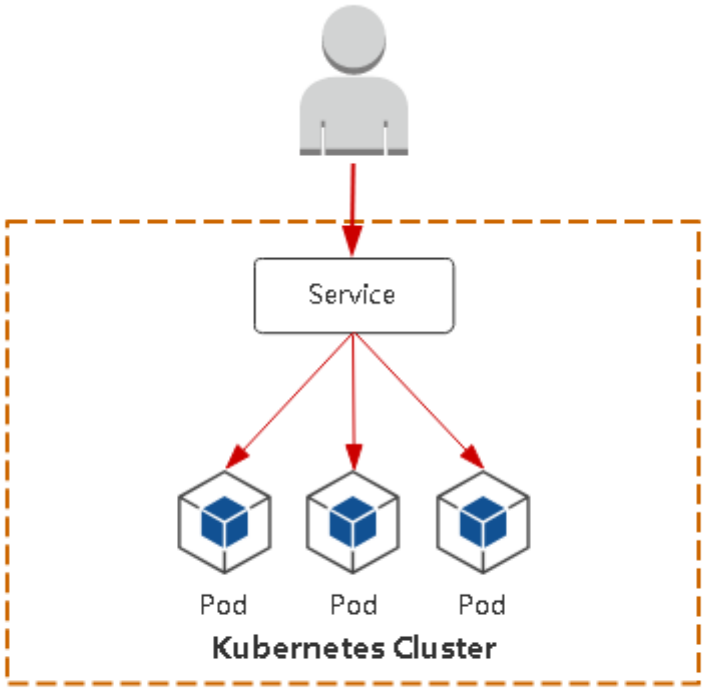
```
# 查看service的标签选择器
kubectl  get svc -o wide

# 查看pod的标签
kubectl  get pod --show-labels
```

```
[root@k8s-master2 ~]# kubectl  get svc -o wide
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)           AGE   SELECTOR
kubernetes   ClusterIP   10.0.0.1      <none>        443/TCP           20d   <none>
nginx-web    NodePort    10.0.0.63     <none>        10090:32377/TCP   11d   app=nginx1.18,project=test
web-nginx    NodePort    10.0.0.160    <none>        80:30008/TCP      6d    app=web-nginx
[root@k8s-master2 ~]#
```

```
[root@k8s-master2 ~]# kubectl  get pod --show-labels
NAME                         READY   STATUS    RESTARTS   AGE     LABELS
nginx1.18-5b4f854b84-224qc   1/1     Running   0          20m     app=nginx1.18,pod-template-hash=5b4f854b84,project=test
nginx1.18-5b4f854b84-7fcgs   1/1     Running   4          11d     app=nginx1.18,pod-template-hash=5b4f854b84,project=test
nginx1.18-5b4f854b84-8xvcg   1/1     Running   2          5d23h   app=nginx1.18,pod-template-hash=5b4f854b84,project=test
nginx1.18-5b4f854b84-fkpt5   1/1     Running   2          3d8h    app=nginx1.18,pod-template-hash=5b4f854b84,project=test
web-5dcb957ccc-w84f4         1/1     Running   4          11d     app=web,pod-template-hash=5dcb957ccc
web-nginx-78fb8f8f9d-4dvm6   1/1     Running   2          5d23h   app=web-nginx,pod-template-hash=78fb8f8f9d
web-nginx-78fb8f8f9d-h2hvd   1/1     Running   2          5d23h   app=web-nginx,pod-template-hash=78fb8f8f9d
web-nginx-78fb8f8f9d-jds5s   1/1     Running   2          5d23h   app=web-nginx,pod-template-hash=78fb8f8f9d
web-nginx-78fb8f8f9d-r4hdw   1/1     Running   2          5d23h   app=web-nginx,pod-template-hash=78fb8f8f9d
[root@k8s-master2 ~]#
```

3.Service三种类型

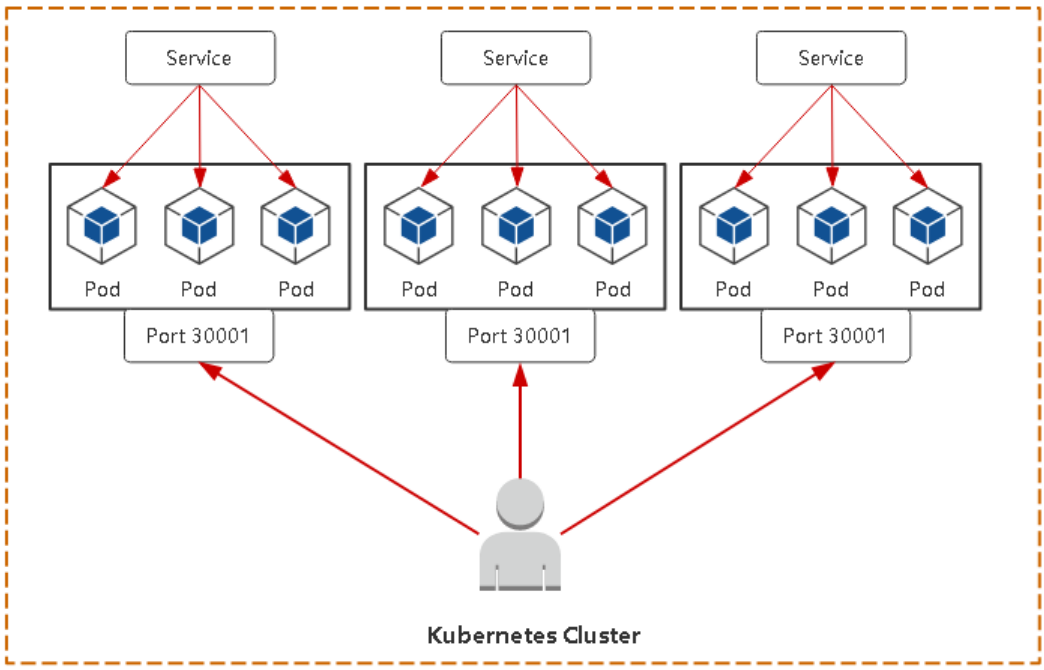
- Cluster IP：默认类型，集群内容使用，分配一个稳定的ip地址（即VIP），只能在集群内部使用（同Namespace内的Pod）。



ClusterIP

```yaml
# clusterip yaml文件
apiVersion: v1
kind: Service
metadata:
  name: web2
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: web-nginx
  type: ClusterIP
```

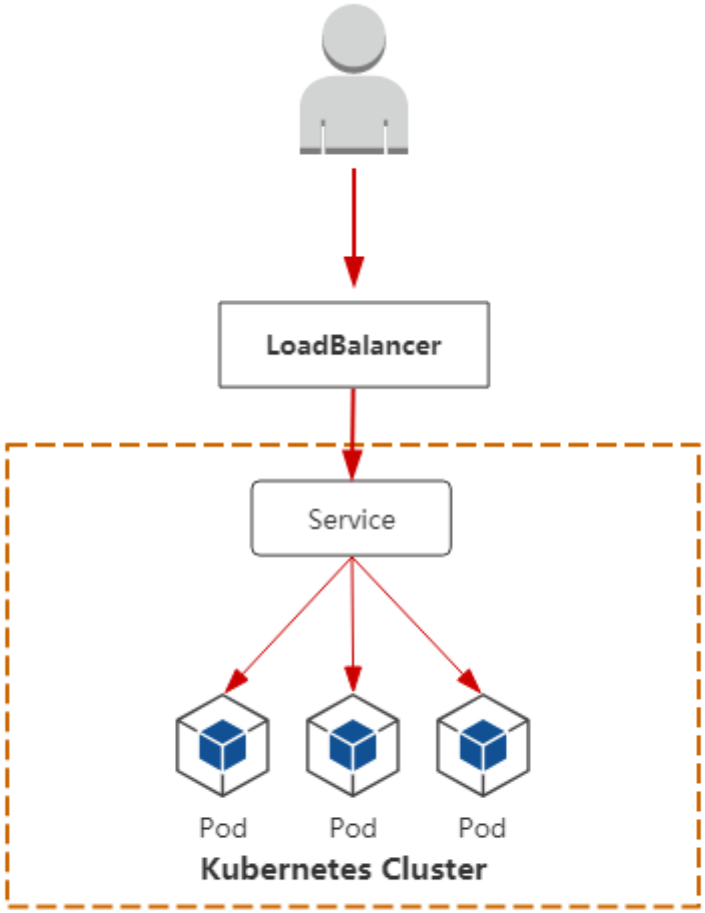
- NodePort: 对外暴露应用。在每个节点上启用一个端口来暴露服务，可以在集群外部访问。也会分配一个稳定内部集群IP地址。访问地址：：端口范围30000-32067



NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web3
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30080
  selector:
    app: web-nginx
  type: NodePort
```

- LoadBalancer: 对外暴露应用，适用公有云、与NodePort类似，在每个节点上启用一个端口来暴露服务。除此之外，Kubernetes会请求底层云平台上的负载均衡器，将每个Node（[NodeIP]:[NodePort]）作为后端添加进去。



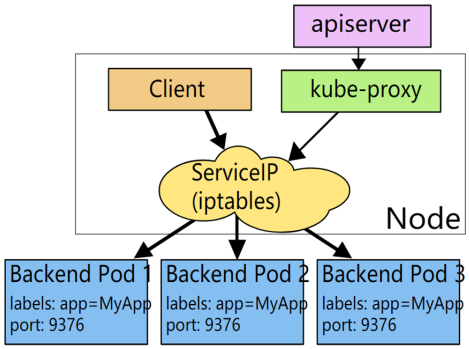
LoadBalancer

4. Service代理模式

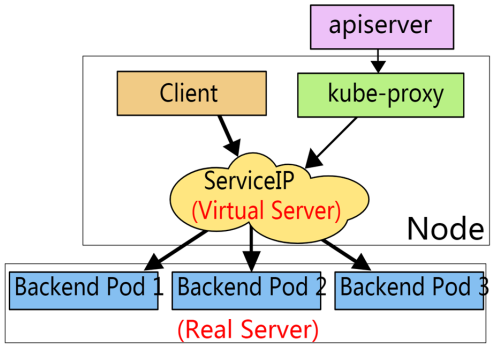
目前k8s中使用的两种代理模式(最早期userspace已经被淘汰)：

- **Iptables**
  - 灵活，功能强大
  - 规则遍历匹配和更新，呈线性时延
  - 可扩展性
- **IPVS**
  - 工作在内核态，有更好的性能
  - 调度算法丰富：rr，wrr，lc，wlc，ip hash...

  默认使用的是iptables，大规模集群推荐使用IPVS



Iptables



IPVS

更改service代理模式：

```
# 安装管理工具
yum -y install ipvsadm ipvsset
# 查看内核模块
lsmod | grep ip_vs
# kube-proxy开启ipvs并重启kube-proxy
修改内容：mode: "ipvs"
# 删除pod重建，然后可以查看日志是否变为ipvs模式
kubectl delete pod xxx -n kube-system
# 也可以查看是否有规则，如果有规则的话说明ipvs生效了
ipvsadm -ln
```

```
[root@k8s-master1 ~]# lsmod | grep ip_vs
ip_vs_sh               12688  0
ip_vs_wrr              12697  0
ip_vs_rr               12600  0
ip_vs                 145458  6 ip_vs_rr,ip_vs_sh,ip_vs_wrr
nf_conntrack          139264  9 ip_vs,nf_nat,nf_nat_ipv4,nf_nat_ipv6,xt_conntrack,nf_nat_masquerade_ipv4,nf_conntrack_netlink,nf
_conntrack_ipv4,nf_conntrack_ipv6
libcrc32c              12644  4 xfs,ip_vs,nf_nat,nf_conntrack
[root@k8s-master1 ~]#
```

二、Ingress详解

Ingress为弥补nodePort不足而生

NodePort存在的不足：

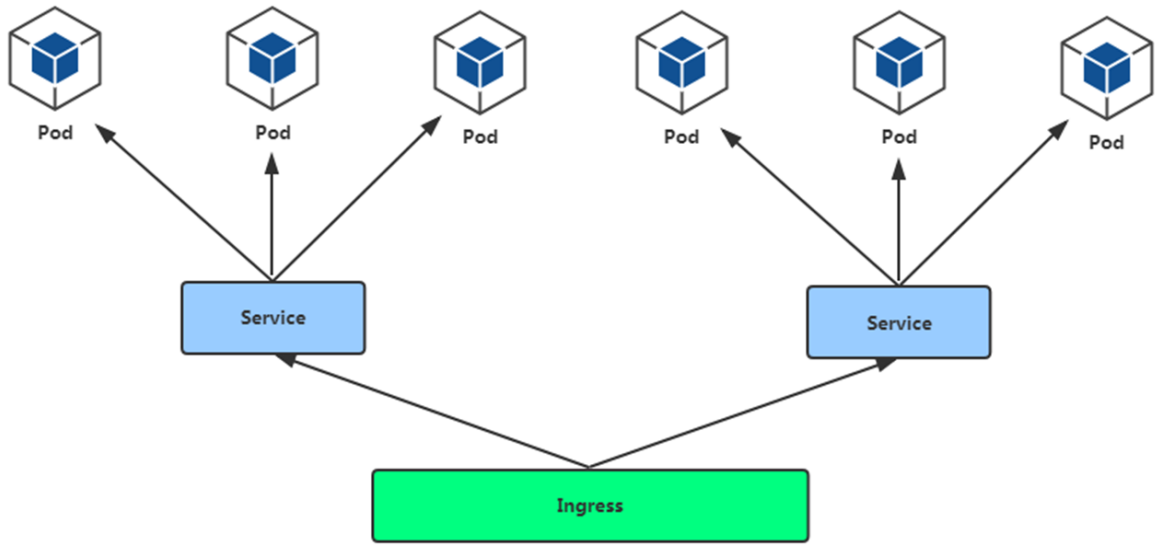
- 一个端口只能一个服务使用，端口需提前规划
- 只支持4层负载均衡。（不可以使用域名访问）

官网说明：

https://kubernetes.io/docs/concepts/services-networking/ingress/

1.Pod与Ingress关系

- 通过Service关联
- 通过IngressController实现Pod负载均衡
    - 支持TCP/UDP 4层和HTTP 7层



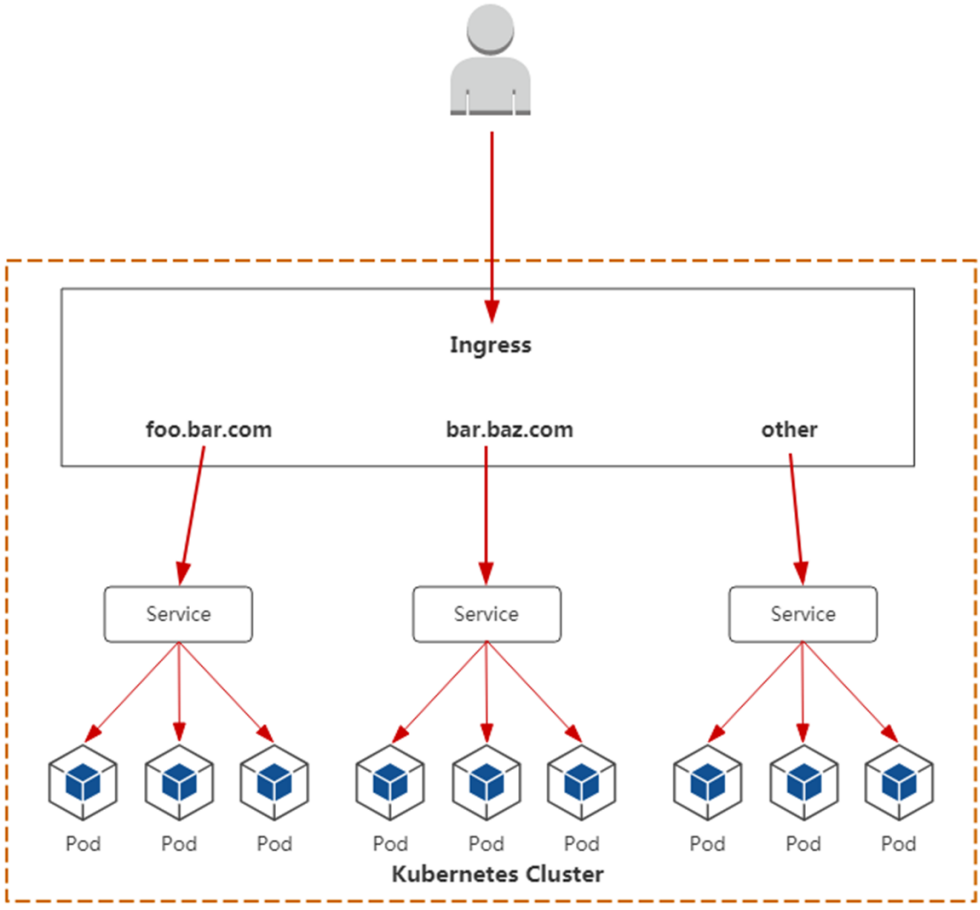
2.Ingress Controller

为了使Ingress资源正常工作，集群必须运行一个Ingress Controller（负载均衡实现）。

所以要想通过ingress暴露你的应用，大致分为两步：

1. 部署Ingress Controller
2. 创建Ingress规则

   整体流程如下：



Ingress Controller有很多实现，我们这里采用官方维护的Nginx控制器。

部署文档：https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md

Traefik： HTTP反向代理、负载均衡工具

Istio：服务治理，控制入口流量

注意事项：

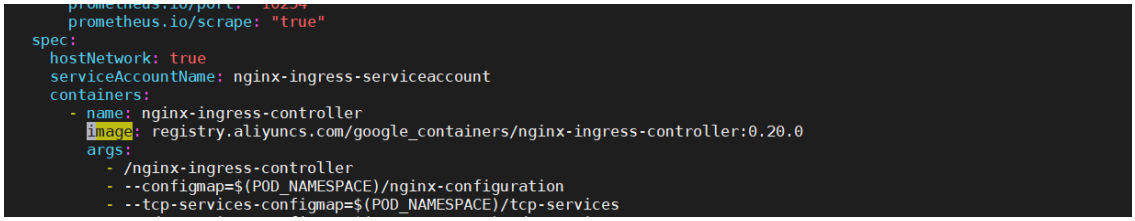
- 镜像下载地址需要修改为国内地址：

  image: registry.aliyuncs.com/google_containers/nginx-ingress-controller:0.20.0

  30版本地址：

  image: registry.aliyuncs.com/google_containers/nginx-ingress-controller:0.30.0

- 使用宿主机网络：hostNetwork: true



操作步骤：

```
# 通过yaml创建ingress-controller
kubectl  apply -f ingress-controller.yaml

# 查看是否成功
kubectl get pod -n ingress-nginx
```

```
[root@k8s-master1 ingress]# kubectl get pod -n ingress-nginx
NAME                             READY   STATUS    RESTARTS   AGE
nginx-ingress-controller-gntmt   1/1     Running   0          3m56s
nginx-ingress-controller-w89k8   1/1     Running   0          4m18s
nginx-ingress-controller-xl26m   1/1     Running   0          4m10s
nginx-ingress-controller-zm59g   1/1     Running   0          4m14s
[root@k8s-master1 ingress]#
```

此时在任意Node上就可以看到该控制监听的80和443端口：

```
netstat -natp |egrep ":80|:443"
```

```
[root@k8s-node1 ~]# netstat -natp |egrep ":80|:443"
tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTEN      37991/nginx: master
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      37991/nginx: master
tcp        0      0 192.168.17.102:2379     192.168.17.101:44326    ESTABLISHED 832/etcd
```

80和443端口就是接收来自外部访问集群中应用流量，转发对应的Pod上。

3. Ingress规则

在ingress里有三个必要字段：

- host：访问该应用的域名，也就是域名解析
- serverName：应用的service名称
- serverPort：service端口

3.1 HTTP访问

使用yaml创建一个ingress

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  - host: test-ingress.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web-nginx
          servicePort: 80
```

```
# 应用
kubectl  apply -f ingress.yaml

# 查看
kubectl  get ingress
```

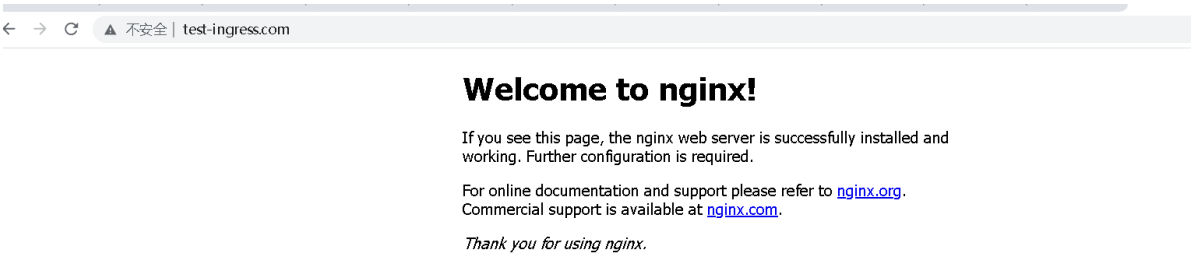
```
[root@k8s-master1 ingress]# kubectl  get ingress
NAME           CLASS    HOSTS              ADDRESS   PORTS   AGE
test-ingress   <none>   test-ingress.com             80      2m55s
[root@k8s-master1 ingress]#
```

生产环境：test-ingress.com域名是在你购买域名的运营商上进行解析，A记录值为K8S Node的公网IP（该Node必须运行了Ingress controller）。

测试环境：可以绑定hosts模拟域名解析（"C:\Windows\System32\drivers\etc\hosts"），对应IP是K8S Node的内网IP。

如：192.168.17.102 test-ingress.com

用域名测试访问：



3.2 HTTPS访问

https的yaml如下：

注意修改参数

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
```

```
metadata:
  name: tls-test-ingress.com
spec:
  tls:
  - hosts:
    - ssltest.ingress.com
    secretName: test-ingress-com
  rules:
    - host: ssltest.ingress.com
      http:
        paths:
        - path: /
          backend:
            serviceName: web
            servicePort: 80
```

secretName用于保存https证书。

使用cfssl工具自签证书用于测试，先下载cfssl工具：

```
curl -s -L -o /usr/local/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
curl -s -L -o /usr/local/bin/cfssljson
https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
curl -s -L -o /usr/local/bin/cfssl-certinfo https://pkg.cfssl.org/R1.2/cfssl-
certinfo_linux-amd64
# 添加执行权限
chmod +x /usr/local/bin/cfssl*
```

生成证书：

生成ca-config.json

```
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "87600h"
    },
    "profiles": {
      "kubernetes": {
         "expiry": "87600h",
         "usages": [
            "signing",
            "key encipherment",
            "server auth",
            "client auth"
        ]
      }
    }
  }
}
EOF
```

生成ca-csr.json

```
cat > ca-csr.json <<EOF
```

```
{
    "CN": "kubernetes",
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "L": "Beijing",
            "ST": "Beijing"
        }
    ]
}
EOF
```

```
# 执行
cfssl gencert -initca ca-csr.json | cfssljson -bare ca -
```

生成test.ingress.com-csr.json文件

```
cat > test.ingress.com-csr.json <<EOF
{
  "CN": "blog.ctnrs.com",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "L": "BeiJing",
      "ST": "BeiJing"
    }
  ]
}
EOF
```

```
# 注意根据自己需求修改参数
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes test.ingress.com-csr.json | cfssljson -bare test.ingress.com

# 注意修改参数
kubectl create secret tls test-ingress-com --cert=test.ingress.com.pem --
key=test.ingress.com-key.pem
```

生成后的结果：

```
[root@k8s-master2 ssl]# ll *pem
-rw------- 1 root root 1679 Mar 20 17:57 ca-key.pem
-rw-r--r-- 1 root root 1273 Mar 20 17:57 ca.pem
-rw------- 1 root root 1679 Mar 20 18:08 test.ingress.com-key.pem
-rw-r--r-- 1 root root 1314 Mar 20 18:08 test.ingress.com.pem
[root@k8s-master2 ssl]#
```

将证书保存在secret里：

```
kubectl create secret tls test-ingress-com --cert=test.ingress.com.pem --
key=test.ingress.com-key.pem
```

这样，ingress就能通过secret名称拿到要用的证书了

绑定本地hosts：

192.168.17.102 ssltest.ingress.com

https访问：

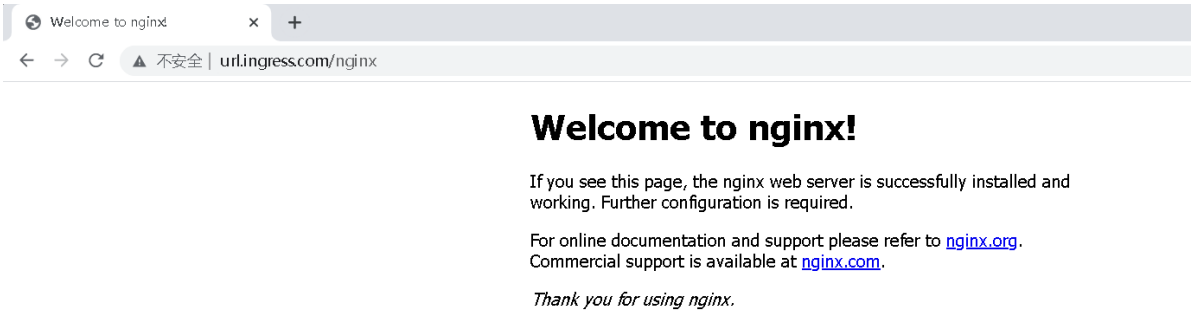
3.3 根据URL路由访问

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: url-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: url.ingress.com
    http:
      paths:
      - path: /nginx
        backend:
          serviceName: web-nginx
          servicePort: 80
  - host: url.ingress.com
    http:
      paths:
      - path: /tomcat
        backend:
          serviceName: tomcat-web
          servicePort: 8080
```

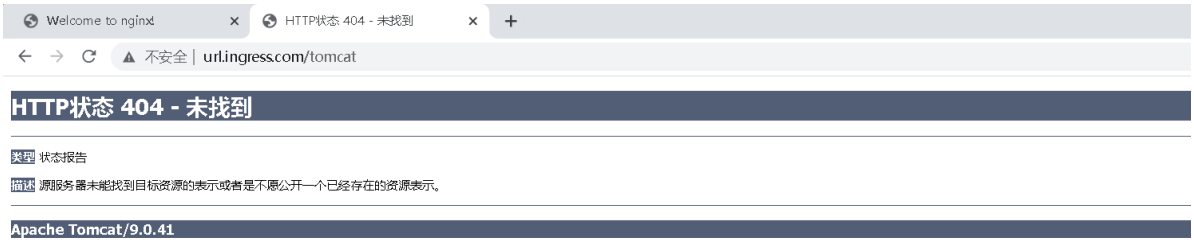
上面配置了相同的hosts，path分别对应nginx和tomcat

把url.ingress.com绑定到本地hots

访问：http://url.ingress.com/nginx



访问：http://url.ingress.com/tomcat



tomcat中没有数据所以404了，但是能正常访问到tomcat

工作流程:

```
                       |           |  >>  serviceName: web-nginx   80
nginx.ingress.com >>   |  node ip  |
                       |           |  >>   serviceName: tomcat-web  8080
```

3.4 虚拟主机

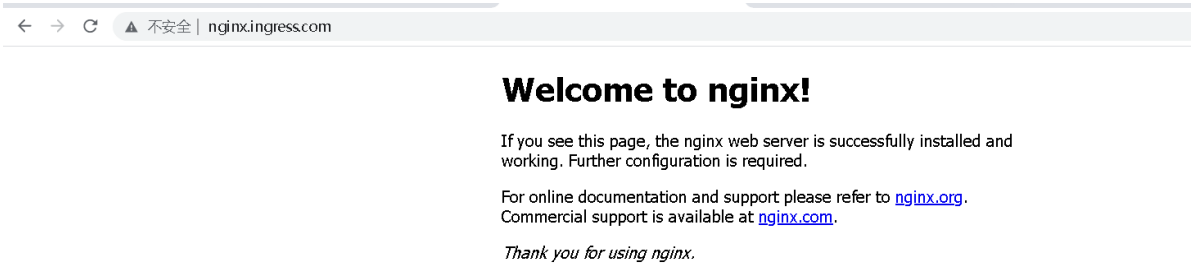
```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: virtual-host-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: nginx.ingress.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web-nginx
          servicePort: 80
  - host: tomcat.ingress.com
    http:
      paths:
      - path: /
        backend:
```

```
          serviceName: tomcat-web
          servicePort: 8080
```

上面配置2个hosts，分别对应nginx和tomcat

把nginx.ingress.com 和 tomcat.ingress.com绑定到本地hots

访问:http://nginx.ingress.com/



访问tomcat：http://tomcat.ingress.com/tomcat

工作流程：

```
nginx.ingress.com >>  |           |  >>  serviceName: web-nginx   80
                      |  node ip  |
tomcat.ingress.com >> |           |  >>   serviceName: tomcat-web  8080
```

4. annotation对Ingress个性化配置

参考文档： https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-configuration/annotations.md

HTTP：配置Nginx常用参数

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
     kubernetes.io/ingress.class: "nginx"
     nginx.ingress.kubernetes.io/proxy-connect-timeout: "600" # 连接超时 默认为60秒
     nginx.ingress.kubernetes.io/proxy-send-timeout: "600"  # 发送超时 默认为60秒
     nginx.ingress.kubernetes.io/proxy-read-timeout: "600" # 读取超时 默认为60秒
     nginx.ingress.kubernetes.io/proxy-body-size: "10m"  # 上传文件大小限制
spec:
```

```
  rules:
  - host: ingress-test.com
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx-test
          servicePort: 18080
```

进入控制器中，查看参数是否已经修改

```
# 进入容器
kubectl exec -it nginx-ingress-controller-5dj8q bash  -n ingress-nginx
# 在容器中查看nginx配置文件
/etc/nginx/nginx.conf |grep 600
```

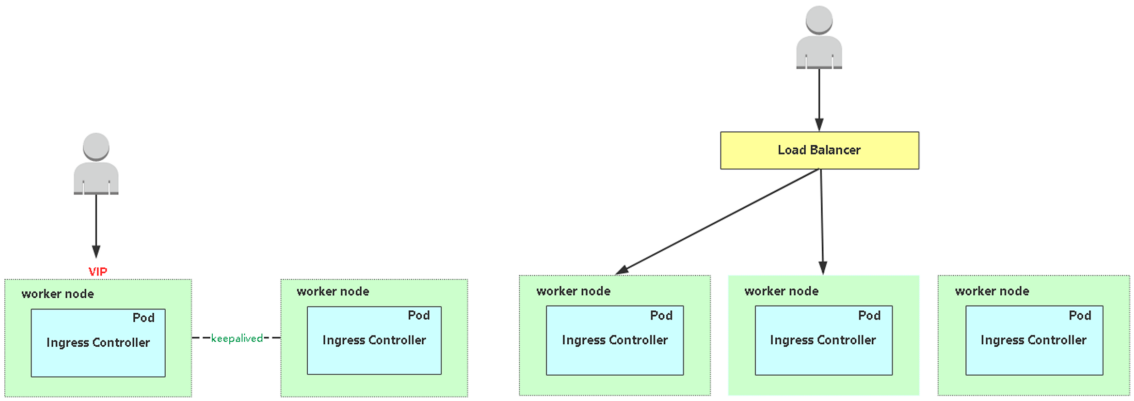
```
bash-5.0$ cat /etc/nginx/nginx.conf |grep 600
                        proxy_connect_timeout                   600s;
                        proxy_send_timeout                      600s;
                        proxy_read_timeout                      600s;
bash-5.0$
```

HTTPS：禁止访问HTTP强制跳转到HTTPS（默认开启）

```
# ture改为fales
 nginx.ingress.kubernetes.io/ssl-redirect: 'false'
```

5. Ingress Controller高可用方案

如果域名只解析到一台Ingress controller，是存在单点的，挂了就不能提供服务了。这就需要具备高可用，有两种常见方案：



左边：双机热备，选择两台Node专门跑Ingress controller，然后通过keepalived对其做主备。用户通过VIP访问。

右边：高可用集群（推荐），前面加一个负载均衡器，转发请求到后端多台Ingress controller。

实现高可用

高可用集群实现步骤，也就是上面右边的图

- **把ingress-controller固定到指定节点**

如果k8s集群比较少，可以使用DamonSet方式部署，集群规模比较大的话可以指定几台node部署。这里使用DaemonSet方式部署

修改ingress-controller.yaml

修改内容：

1.把Deployment改为DamonSet方式部署，并删除replicas（副本数）

2.因为是DaemonSet方式部署，所以删除replicas（副本数）

```
kind: DaemonSet
```

修改前：

```
---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  replicas: 1
  selector:
```

修改后：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:                          删除replicas
  selector:
    matchLabels:
      app.kubernetes.io/name: ingress-nginx
      app.kubernetes.io/part-of: ingress-nginx
  template:
```

修改完成后，重新部署ingress-controller

```
kubectl apply -f ingress-controller.yaml
# 应用完成后，过段时间查看pod是否运行
kubectl get pod --all-namespaces |grep ingres
```

```
[root@k8s-master1 ~]# kubectl get pod --all-namespaces |grep ingres
ingress-nginx          nginx-ingress-controller-256zz              1/1     Running
0          6m29s
ingress-nginx          nginx-ingress-controller-gd6k6              1/1     Running
0          6m24s
ingress-nginx          nginx-ingress-controller-gvgh2              1/1     Running
0          6m19s
ingress-nginx          nginx-ingress-controller-k2rvk              1/1     Running
0          6m26s
[root@k8s-master1 ~]#
```

- **安装配置nginx**

  这里直接使用yum方式安装

  ```
  # 安装epel源
  yum install epel-release -y
  # 安装nginx
  yum install nginx -y
  ```

  修改配置文件：

  ```
  vim /etc/nginx/nginx.conf
  ```

  修改内容：

  ```
  # 添加upstream，根据k8s节点ip填写
      upstream ingress{
          server 10.0.33.190;
          server 10.0.33.191;
          server 10.0.33.192;
          server 10.0.33.193;
      }
  # 设置反向代理
      server {
          listen       80;
          listen       [::]:80;
          location / {
              proxy_pass http://ingress;
              proxy_set_header Host $host; #传输域名
          }
      }
  ```

  修改完成后reload一下

- **测试**

  修改C:\Windows\System32\drivers\etc\hosts文件，修改对应nginx的ip再进行访问。

  访问域名，查看nginx日志看有没有跳转后端ingress的地址

三、应用程序配置

1.secret

secret加密数据并存放Etcd中，让Pod的容器以挂载Volume方式访问。

应用场景：凭据

Pod使用secret两种方式：

- 变量注入
- 挂载

例如：创建一个secret用于保存应用程序用到的账号密码

```
# 创建加密用户密码
[root@k8s-master1 ~]# echo -n "admin" |base64
YWRtaW4=
[root@k8s-master1 ~]# echo -n "passwd" |base64
cGFzc3dk
```

创建secret yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dk
```

```
# 查看创建secret
kubectl  get secret
```

变量注入方式在Pod中使用secret：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: nginx
    image: nginx
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
```

创建完pod后进入容器查看是否传入变量：

```
# 进入pod
kubectl  exec -it mypod sh
# 查看变量
echo $SECRET_USERNAME
echo $SECRET_PASSWORD
```

```
[root@k8s-master1 ~]# kubectl  exec -it mypod sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl kubectl exe
# echo $SECRET_USERNAME
admin
# echo $SECRET_PASSWORD
passwd
#
```

数据挂载方式在Pod中使用secret：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

```
# 进入容器中查看是否挂载成功
kubectl  exec -it mypod sh
cat /etc/foo/username
cat /etc/foo/password
```

```
[root@k8s-master1 ~]# kubectl  exec -it mypod sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl kubectl
# cat /etc/foo/username
admin#
# cat /etc/foo/password
passwd#
```

2.configmap

与Secret类似，区别在于ConfigMap保存的是不需要加密配置信息。

应用场景：应用配置

例如：创建一个configmap用于保存应用程序用到的字段值

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfig
  namespace: default
data:
  special.level: info
  special.type: hello
```

变量注入方式在Pod中使用configmap：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: busybox
      image: busybox
      command: [ "/bin/sh", "-c", "echo $(LEVEL) $(TYPE)" ]
      env:
        - name: LEVEL
          valueFrom:
            configMapKeyRef:
              name: myconfig
              key: special.level
        - name: TYPE
          valueFrom:
            configMapKeyRef:
              name: myconfig
              key: special.type
  restartPolicy: Never
```

查看Pod日志就可以看到容器里打印的键值了：

```
kubectl logs mypod
```

```
[root@k8s-master1 ~]# kubectl  logs mypod
info hello
[root@k8s-master1 ~]#
```

.

举一个常见的用法，例如将应用程序的配置文件保存到configmap中，这里以redis为例：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  redis.properties: |
    redis.host=127.0.0.1
    redis.port=6379
    redis.password=123456
---
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: busybox
      image: busybox
      command: [ "/bin/sh","-c","cat /etc/config/redis.properties" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
```

```
  volumes:
    - name: config-volume
      configMap:
        name: redis-config
  restartPolicy: Never
```

四、Pod数据持久化

参考文档：https://kubernetes.io/docs/concepts/storage/volumes/

- Kubernetes中的Volume提供了在容器中挂载外部存储的能力
- Pod需要设置卷来源（spec.volume）和挂载点（spec.containers.volumeMounts）两个信息后才可以使用相应的Volume

挂载分类：

1、本地：hostPath，emptyDir
2、网络：nfs，rbd，cephfs，glusterfs
3、公有云：awsElasticBlockStore，azureDisk
4、k8s资源：secret，configMap

1. emptyDir

创建一个空卷，挂载到Pod中的容器。Pod删除该卷也会被删除。

应用场景：Pod中容器之间数据共享

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: write
    image: centos
    command: ["bash","-c","for i in {1..100};do echo $i >> /data/hello;sleep
1;done"]
    volumeMounts:
      - name: data
        mountPath: /data

  - name: read
    image: centos
    command: ["bash","-c","tail -f /data/hello"]
    volumeMounts:
      - name: data
        mountPath: /data

  volumes:
  - name: data
    emptyDir: {}
```

以上yaml创建了2个pod：write和read，一个往/data/目录中的hello文件写数据，一个读取/data/hello中的数据

查看read 日志，看是否在读取数据

```
kubectl  logs -f my-pod -c read
```

```
[root@k8s-master1 volume]# kubectl  logs -f my-pod -c read
1
2
3
4
5
6
7
8
9
10
11
12
```

2. hostPath

挂载Node文件系统上文件或者目录到Pod中的容器。

应用场景：Pod中容器需要访问宿主机文件

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: busybox
    image: busybox
    args:
    - /bin/sh
    - -c
    - sleep 36000
    volumeMounts:
    - name: data
      mountPath: /data
  volumes:
  - name: data
    hostPath:
      path: /tmp
      type: Directory
```

volumes:中定义宿主机节点的/tmp目录

volumeMounts:定义容器中挂载的路径

进入容器查看/data目录下与对应的节点下的/tmp目录数据相同

```
kubectl  exec -it my-pod sh
```

```
[root@k8s-master1 volume]# kubectl  exec -it my-pod sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl kubectl exec [POD] -- [C
/ # ls /data
systemd-private-e8d6c8af1e1746919a948031da297ea1-chronyd.service-jUyBWh  vmware-root_537-4257134911
vmware-root_535-4290690870                                               vmware-root_548-2999460674
/ #
```

3. 网络共享存储

目前主流的网络共享存储：nfs，rbd，cephfs，glusterfs等，这里使用nfs演示

安装nfs

```
# yum 方式安装（k8s所有节点都装）
yum install nfs-utils
# 设置开机启动
systemctl enable nfs
# 创建一个需要挂载的目录
mkdir -p /data/nfstest
# 找一台机器进行挂载
# 修改配置文件，将刚创建的目录共享出去
vi /etc/exports
# 添加以下内容
/data/nfstest    192.168.17.0/24(rw,no_root_squash)

# 重启nfs
systemctl  restart nfs
#查看是否共享成功
showmount -e
```

```
[root@k8s-node1 ~]# showmount -e
Export list for k8s-node1:
/data/nfstest 192.168.17.0/24
[root@k8s-node1 ~]#
```

使用yaml创建一个deployment

volumes:中定义来源信息。nfs为类型，server：192.168.17.102 为nfs的ip，path: /data/nfstest 挂载路径

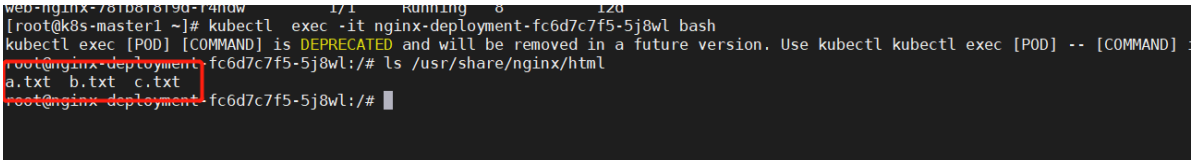
volumeMounts：中定义容器的挂载信息。mountPath: /usr/share/nginx/html 为容器中挂载地址。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: wwwroot
          mountPath: /usr/share/nginx/html
        ports:
        - containerPort: 80
```

```
      volumes:
      - name: wwwroot
        nfs:
          server: 192.168.17.102
          path: /data/nfstest
```

测试是否挂载成功

```
# 在nfs节点中创建几个测试文件
cd /data/nfstest
touch {a,b,c}.txt
# 进入pod中查看挂载点是否有这个3个文件
kubectl  exec -it nginx-deployment-fc6d7c7f5-5j8wl bash
ls /usr/share/nginx/html
```



4.PV/PVC

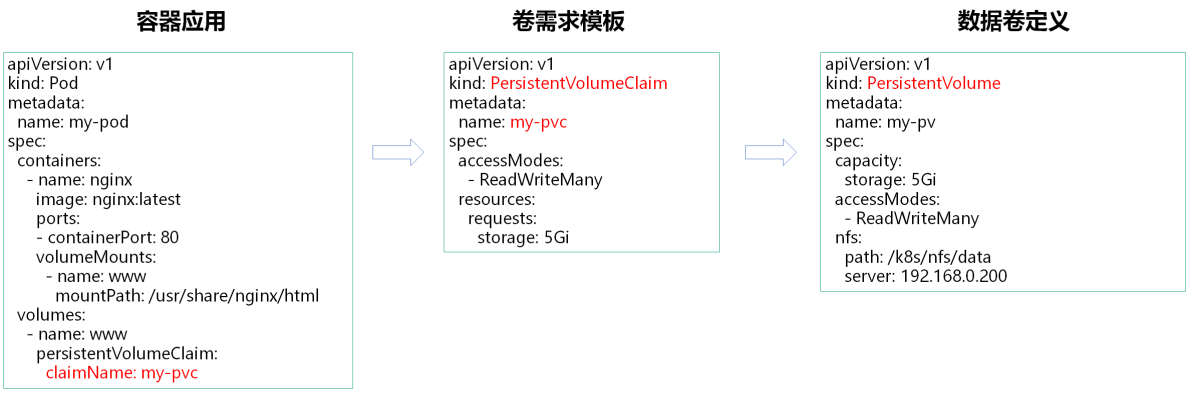
PersistentVolume（PV）：是集群中由管理员配置的**一段网络存储。** 它是集群中的资源，就像节点是集群资源一样。 PV是容量插件，如Volumes，但其生命周期独立于使用PV的任何单个pod。 此API对象捕获存储实现的详细信息，包括NFS，iSCSI或特定于云提供程序的存储系统。

PV供给分为静态/动态供给：

- 静态供给Static：集群管理员创建多个PV。它们携带可供集群用户使用的真实存储的详细信息。 它们存在于Kubernetes API中，可用于消费
- 动态提供Dynamic：当管理员创建的静态PV都不匹配用户的PersistentVolumeClaim时，集群可能会尝试为PVC动态配置卷。 此配置基于StorageClasses：PVC必须请求一个类，并且管理员必须已创建并配置该类才能进行动态配置。 要求该类的声明有效地为自己禁用动态配置。

PersistentVolumeClaim（PVC）：是由**用户进行存储的请求。** 它类似于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）。声明可以请求特定的大小和访问模式（例如，可以一次读/写或多次只读）。

PVC和PV是一一对应的。



1.生命周期

PV是集群中的资源。PVC是对这些资源的请求，并且还充当对资源的检查。PV和PVC之间的相互作用遵循以下生命的周期：

Provisioning ----> Binding ----> Using ----> Releasing ----> Recycling

- **供应配置Provisioning**: 通过集群外的存储系统或者云平台来提供存储持久化支持。
- **绑定Binding**：用户创建PVC并指定需要的资源和访问模式。在找到可用的PV之前，PVC一直会处于未绑定状态。
- **使用Using**： 用户可以在容器中想volume一样使用PVC
- **释放Releasing**---用户删除pvc来回收存储资源，pv将变成“released”状态。由于还保留着之前的数据，这些数据需要根据不同的策略来处理，否则这些存储资源无法被其他pvc使用。
- **回收Recycling**

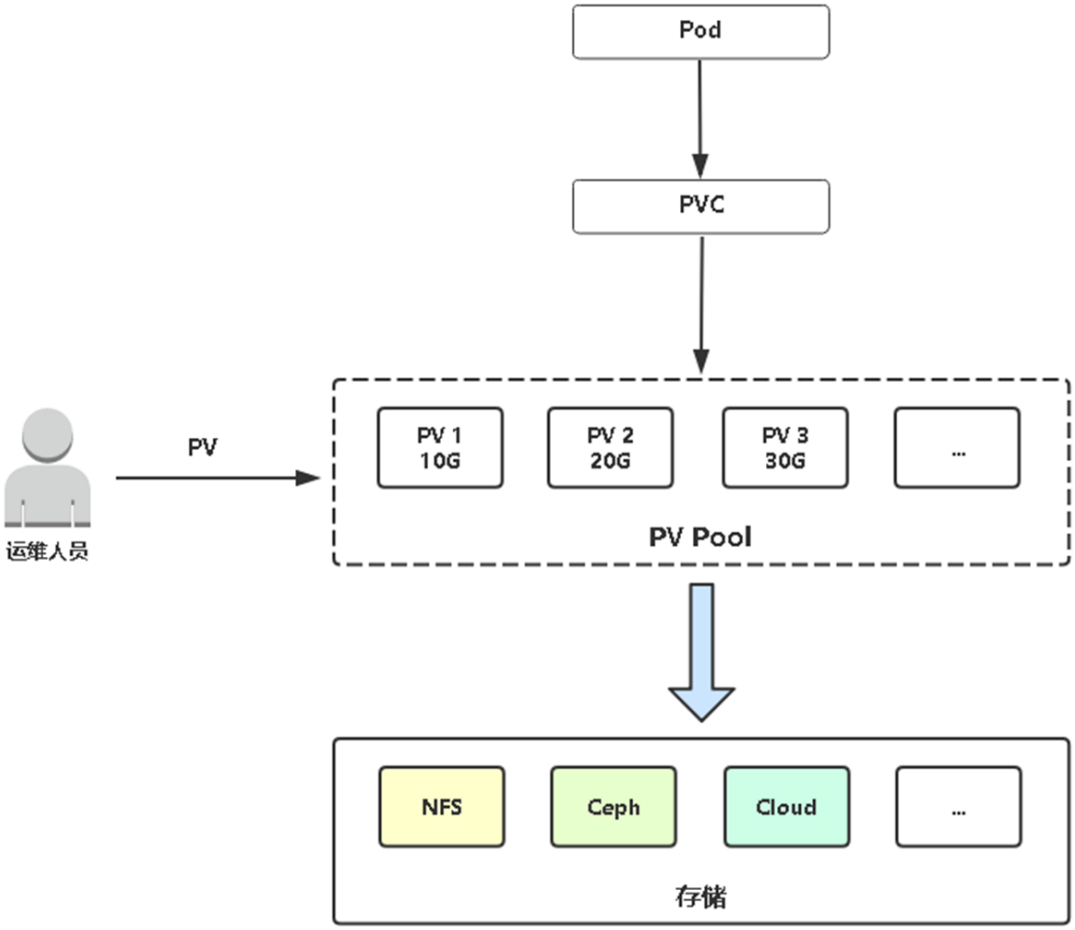
  PV可以设置三种回收策略：保留（Retain），回收（Recycle）和删除（Delete）。

  - 保留策略：保留PV&数据 [推荐]
  - 删除策略：直接删除PV+数据 【不推荐】
  - 回收策略：清除数据，保留PV 【被废弃】

访问模式：
ReadWriteOnce(RWO) 单个pod读写 ，应用场景：数据独立，一般块设备
ReadOnlyMany(ROX) 所有pod只读
ReadWriteMany(RWX) 所有pod读写，应用场景：数据共享，文件系统

2.静态供给

静态供给是指提前创建多个PV，以供使用。



提前准备一个NFS服务器（上面已经准备好了）

使用yaml创建3个PV：分别为5G/10G/20G

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001        # 修改PV名称
spec:
  capacity:
    storage: 5Gi    # 修改大小
  accessModes:
    - ReadWriteMany
  nfs:
    path: /data/nfstest/pv001    # 修改目录名
    server: 192.168.17.102

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv002        # 修改PV名称
spec:
  capacity:
    storage: 10Gi    # 修改大小
  accessModes:
    - ReadWriteMany
```

```
  nfs:
    path: /data/nfstest/pv002   # 修改目录名
    server: 192.168.17.102

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv003       # 修改PV名称
spec:
  capacity:
    storage: 20Gi   # 修改大小
  accessModes:
    - ReadWriteMany
  nfs:
    path: /data/nfstest/pv003   # 修改目录名
    server: 192.168.17.102
```

在nfs也创建对应的目录

```
mkdir -p /data/nfstest/pv00{1,2,3}
```

查看创建的PV

```
[root@k8s-master1 ~]# kubectl  get pv
NAME    CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
pv001   5Gi        RWX            Retain           Available                                   30m
pv002   10Gi       RWX            Retain           Available                                   30m
pv003   20Gi       RWX            Retain           Available                                   30m
[root@k8s-master1 ~]#
```

创建Pod使用PV

以下yaml创建了Pod和PVC：

PVC中：命名my-pvc，申请了8G磁盘

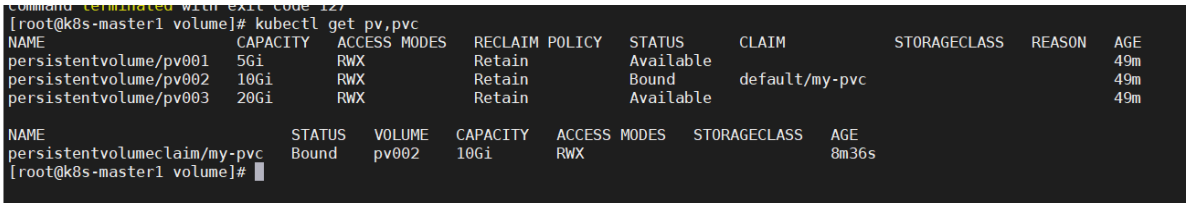
Pod中：使用了名为my-pvc的PVC，并挂载到:/usr/share/nginx/html

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
      - containerPort: 80
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumes:
    - name: www
      persistentVolumeClaim:
        claimName: my-pvc

---
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 8Gi
```

创建完查看PV，Pvc，会发现它们pv002绑定成功。

```
kubectl get pv,pvc
```



上面Pvc申请的是8G磁盘，PV里面是5G/10G/20G，没有与8G匹配的PV。这里其实它会自动匹配合符的PV。

测试验证：

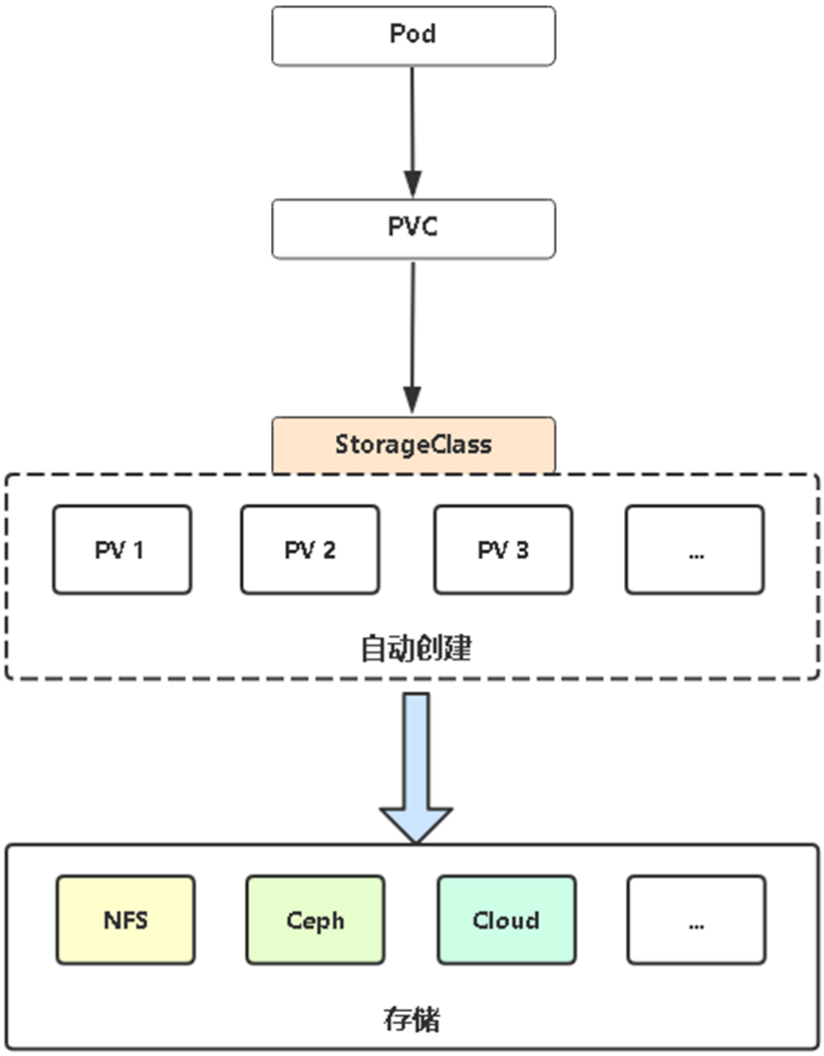
进入到容器中/usr/share/nginx/html（PV挂载目录）目录下创建一个文件测试

```
kubectl exec -it my-pod bash
cd /usr/share/nginx/html
echo "123" > index.html
```

在nfs挂载的pv002目录查看也生成了此文件，说明正常：

3. 动态供给

动态供给（Dynamical Provision），即如果没有满足 PVC 条件的 PV，会动态创建 PV。相比静态供给，动态供给有明显的优势：不需要提前创建 PV，减少了工作量，提高了效率。



动态供给是通过 StorageClass 实现的，StorageClass 定义了如何创建 PV。

StorageClass声明存储插件，用于自动创建PV：

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

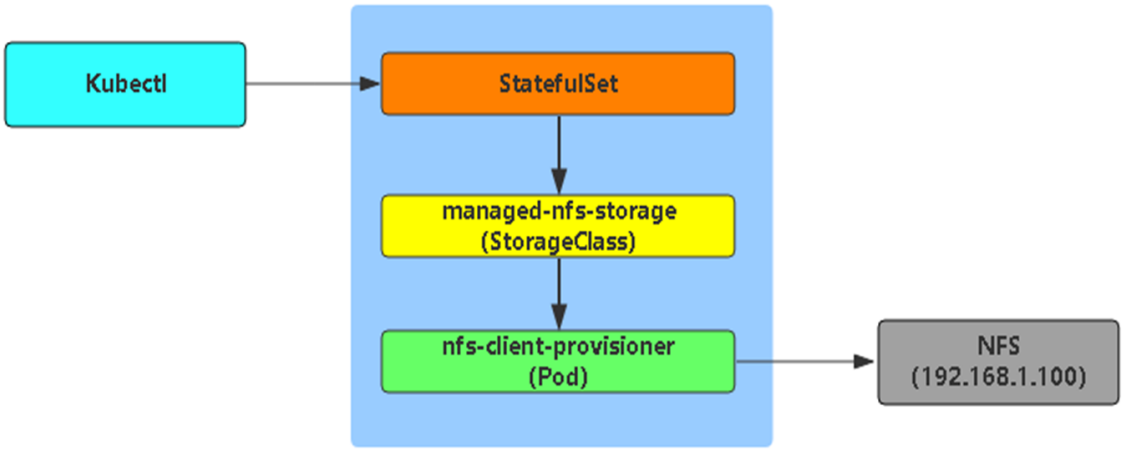
下图这个表中，√ 表示支持动态供给。我们用的NFS其实是不支持动态供给的，但是可以使用其他方式实现。

存储制备器

每个 StorageClass 都有一个制备器（Provisioner），用来决定使用哪个卷插件制备 PV。

卷插件	内置制备器	配置例子
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-

4. PV动态供给示例（NFS）



由于K8S不支持NFS动态供给，还需要先安装上图中的nfs-client-provisioner插件：

官方插件下载：https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client/deploy

如果无法下载把整个目录下载，并解压

```
# 进入目录开始安装
cd nfs-client
# 修改里面NFS地址和共享目录
vi deployment.yaml
```

```
        env:
          - name: PROVISIONER_NAME
            value: fuseim.pri/ifs
          - name: NFS_SERVER
            value: 192.168.17.102
          - name: NFS_PATH
            value: /data/nfstest
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.17.102
            path: /data/nfstest
```

```
# 应用所有yaml
kubectl apply -f .
# 查看pod
kubectl  get pod |grep nfs
```

```
[root@k8s-master1 nfs-clinet]# kubectl  get pod |grep nfs
nfs-client-provisioner-bb77dd455-m6x6b   1/1     Running   0          117s
[root@k8s-master1 nfs-clinet]#
```

测试:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
      - containerPort: 80
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumes:
    - name: www
      persistentVolumeClaim:
        claimName: my-pvc

---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: "managed-nfs-storage"
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 12Gi
```

这次会自动创建12G PV并与PVC绑定。

查看创建结果：

```
kubectl  get pv
kubectl  get pvc
```

```
[root@k8s-master1 nfs-clinet]# kubectl  get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM           STORAGECLASS          REASON   AGE
pvc-39994f59-a85e-4210-a605-6ad67283dd9c   12Gi       RWX            Delete           Bound    default/my-pvc  managed-nfs-storage            2m37s
[root@k8s-master1 nfs-clinet]#
[root@k8s-master1 nfs-clinet]#
[root@k8s-master1 nfs-clinet]# kubectl  get pvc
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS          AGE
my-pvc   Bound    pvc-39994f59-a85e-4210-a605-6ad67283dd9c   12Gi       RWX            managed-nfs-storage   2m42s
[root@k8s-master1 nfs-clinet]#
```

我们在nfs服务器上查看，也会生成一个对应的目录：

```
2. 192.168.17.100 (root)   6. 192.168.17.102 (root)
[root@k8s-node1 nfstest]# ll
total 0
drwxrwxrwx 2 root root  6 Mar 23 00:45 default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c
drwxr-xr-x 2 root root 24 Mar 22 23:59 pv002
[root@k8s-node1 nfstest]#
```

注意这里回收策略变为了：Delete，为了以防误删数据，修改一下class.yaml中的参数：archiveOnDelete: "false"改为“true”

修改完成后，删掉重新apply -f 一下。

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

测试：

我们挂载目录下创建一些数据

```
# 在nfs服务器，进入目录并创建3个文本
cd default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c/
touch {1,2,3}.txt
```

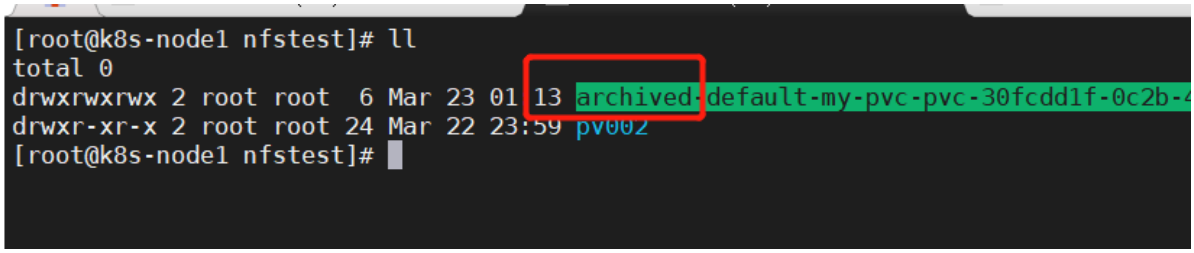
```
2. 192.168.17.100 (root)   6. 192.168.17.102 (root)
[root@k8s-node1 nfstest]# cd default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c/
[root@k8s-node1 default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c]# touch {1,2,3}.txt
[root@k8s-node1 default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c]# ll
total 0
-rw-r--r-- 1 root root 0 Mar 23 00:52 1.txt
-rw-r--r-- 1 root root 0 Mar 23 00:52 2.txt
-rw-r--r-- 1 root root 0 Mar 23 00:52 3.txt
[root@k8s-node1 default-my-pvc-pvc-39994f59-a85e-4210-a605-6ad67283dd9c]#
```

再进入容器中查看是否有数据：

```
[root@k8s-master1 nfs-clinet]# kubectl  exec -it my-pod bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl kubectl exec [POD] -- [COMMAND] instead.
root@my-pod:/# ls /usr/share/nginx/html
1.txt  2.txt  3.txt
root@my-pod:/#
```

删除pod，pvc后，在nfs服务中查看数据是否还在。

进入nfs服务器，查看刚刚自动创建的目录不在了，但是出现了一个新目录，说明已经开启自动备份：



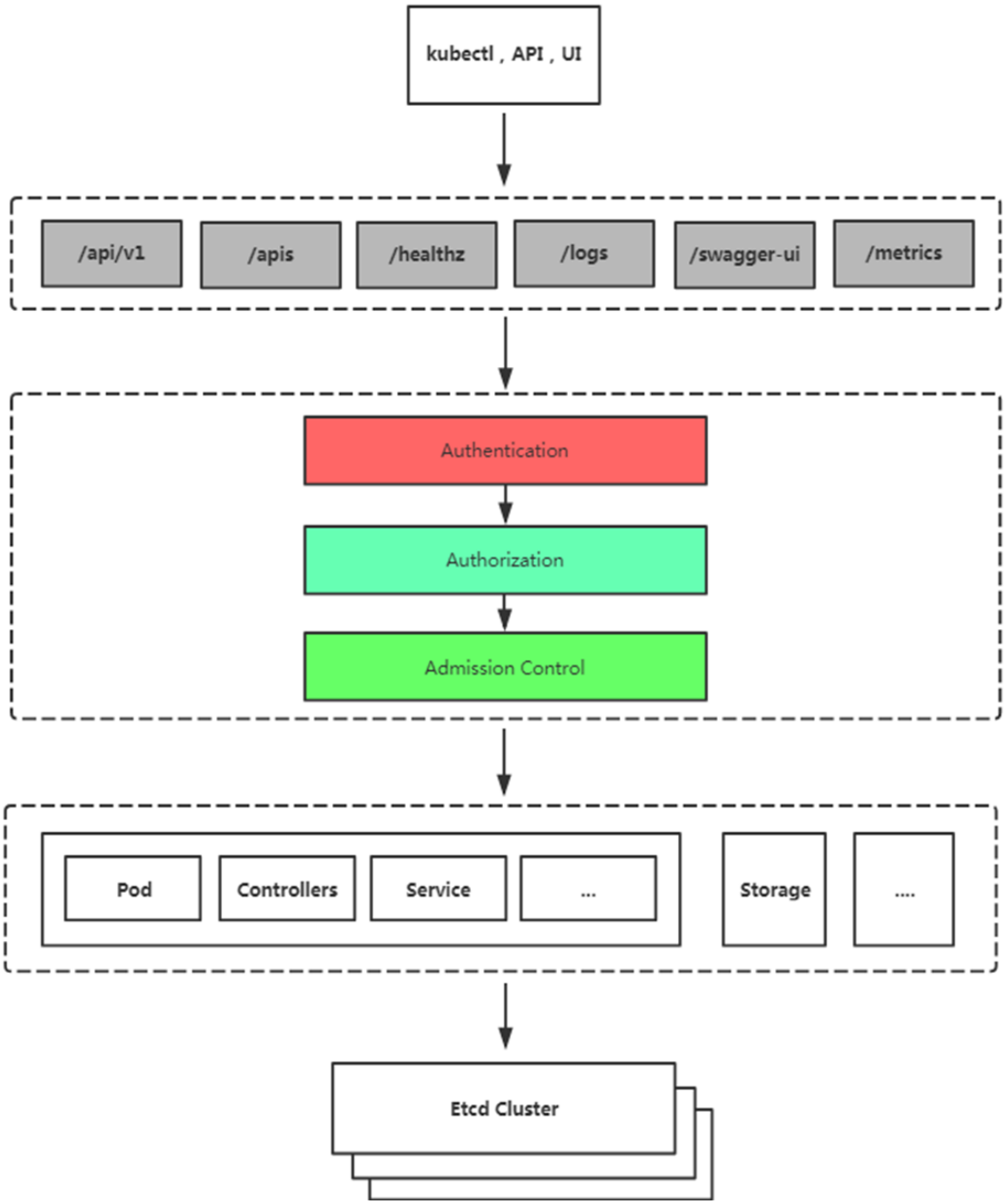
五、鉴权框架与用户权限分配

1.Kubernetes的安全框架

- 访问K8S集群的资源需要过三关：认证、鉴权、准入控制
- 普通用户若要安全访问集群API Server，往往需要证书、Token或者用户名+密码；Pod访问，需要ServiceAccount
- K8S安全控制框架主要由下面3个阶段进行控制，每一个阶段都支持插件方式，通过API Server配置来启用插件。

访问API资源要经过以下三关才可以：

1. Authentication（鉴权）
2. Authorization（授权）
3. Admission Control（准入控制）



2.传输安全，认证，授权，准入控制

传输安全：

- 放弃8080，使用6443
- 全面基于HTTPS通信

鉴权：三种客户端身份认证：

- HTTPS 证书认证：基于CA证书签名的数字证书认证
- HTTP Token认证：通过一个Token来识别用户
- HTTP Base认证：用户名+密码的方式认证（安全性低，基本弃用）

授权：

RBAC（Role-Based Access Control，基于角色的访问控制）：负责完成授权（Authorization）工作。

根据API请求属性，决定允许还是拒绝。

准入控制：

Adminssion Control实际上是一个准入控制器插件列表，发送到API Server的请求都需要经过这个列表中的每个准入控制器插件的检查，检查不通过，则拒绝请求。

3. 使用RBAC授权

RBAC（Role-Based Access Control，基于角色的访问控制），允许通过Kubernetes API动态配置策略。

角色

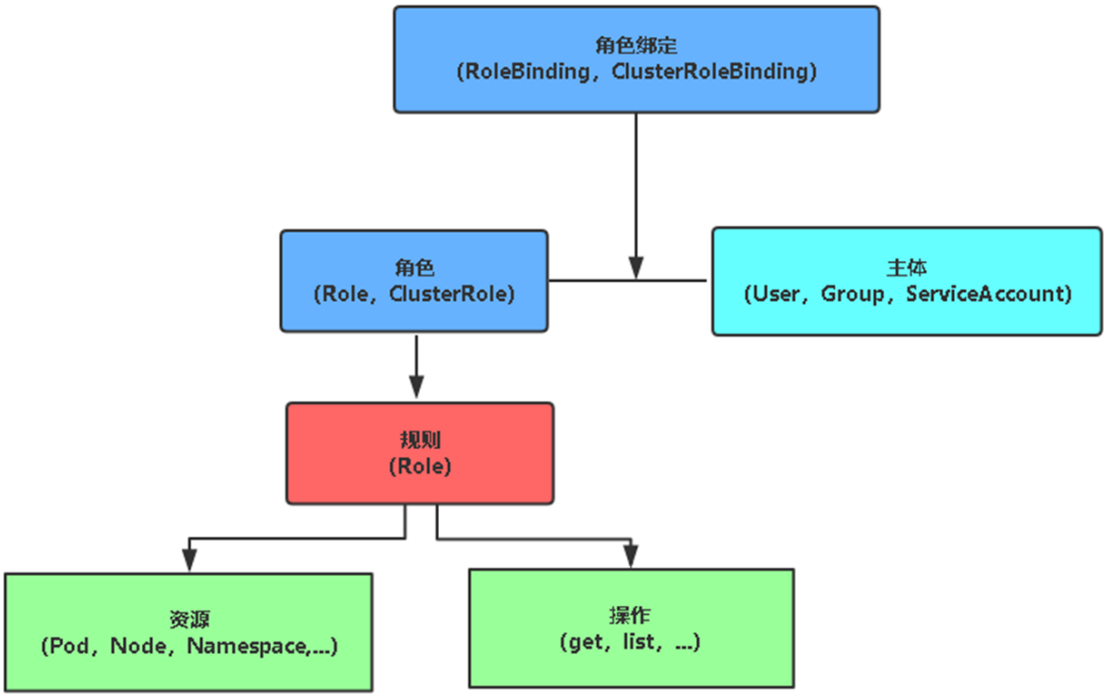
- Role：授权特定命名空间的访问权限
- ClusterRole：授权所有命名空间的访问权限

角色绑定

- RoleBinding：将角色绑定到主体（即subject）
- ClusterRoleBinding：将集群角色绑定到主体

主体（subject）

- User：用户
- Group：用户组
- ServiceAccount：服务账号



4.RBAC示例

创建一个test用户授权default命名空间Pod读取权限

实现步骤：

1. 用K8S CA签发客户端证书
2. 生成kubeconfig授权文件
3. 创建RBAC权限策略

1.用K8S CA签发客户端证书

- **cfssl工具生成**

  使用cfssl工具自签证书，先下载cfssl工具

```
curl -s -L -o /usr/local/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
curl -s -L -o /usr/local/bin/cfssljson https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
curl -s -L -o /usr/local/bin/cfssl-certinfo https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64
# 添加执行权限
chmod +x /usr/local/bin/cfssl*
```

- **签发客户端证书**

```
# 创建一个rbac目录
mkdir rbac
cd rbac/
```

  执行以下步骤

```
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "87600h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
            "signing",
            "key encipherment",
            "server auth",
            "client auth"
        ],
        "expiry": "87600h"
      }
    }
  }
}
EOF

cat > test-csr.json <<EOF
{
  "CN": "test",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
```

```
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
}
EOF


cfssl gencert -ca=/opt/kubernetes/ssl/ca.pem -ca-key=/opt/kubernetes/ssl/ca-key.pem -config=ca-config.json -profile=kubernetes test-csr.json | cfssljson -bare test
```

注意参数说明：

- test-csr.json

  CN: 用户名，这里使用test用户，就是："CN": "test"

  O: 用户组，这里是用的k8s，就是"O": "k8s"

- cfssl gencert命令参数

  -ca=：证书路径，注意根据实际路径填写。后缀也可能不一样，如：.crt结尾的

  -ca-key=：证书对应的私钥文件

执行完成后，会生成：test.pem：证书文件，test-key.pem：私钥文件

```
[root@k8s-master1 rbac]# ll -lrt
总用量 20
-rw-r--r-- 1 root root  292 10月  6 20:27 ca-config.json
-rw-r--r-- 1 root root  217 10月  6 20:27 test-csr.json
-rw-r--r-- 1 root root 1383 10月  6 20:33 test.pem
-rw------- 1 root root 1675 10月  6 20:33 test-key.pem
-rw-r--r-- 1 root root  993 10月  6 20:33 test.csr
[root@k8s-master1 rbac]#
```

2.生成kubeconfig授权文件

依次执行以下命令，或者改为脚本执行，注意参数修改

```
# 生成kubeconfig授权文件
# 参数：  --certificate-authority=: k8s的证书，注意不是刚生成的。
#       --server : k8s apiserver的地址
#       --kubeconfig : 生成后的授权文件名
kubectl config set-cluster kubernetes \
  --certificate-authority=/opt/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=https://10.0.33.190:6443 \
  --kubeconfig=test.kubeconfig

# 设置客户端认证
kubectl config set-credentials test \
  --client-key=test-key.pem \
  --client-certificate=test.pem \
  --embed-certs=true \
  --kubeconfig=test.kubeconfig

# 设置默认上下文
```

```
kubectl config set-context kubernetes \
  --cluster=kubernetes \
  --user=test \
  --kubeconfig=test.kubeconfig

# 设置当前使用配置
kubectl config use-context kubernetes --kubeconfig=test.kubeconfig
```

执行完以上步骤后，可以测试一下：

```
# --kubeconfig指定刚生成的test.kubeconfig
kubectl --kubeconfig=test.kubeconfig get pods
```

提示没有权限访问：

```
[root@k8s-master1 rbac]# kubectl --kubeconfig=test.kubeconfig get pods
Error from server (Forbidden): pods is forbidden: User "test" cannot list resource "pods" in AP
I group "" in the namespace "default"
[root@k8s-master1 rbac]#
```

3.创建RBAC权限策略

- **创建角色（权限集合）**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

参数说明：

kind: Role ： 角色资源类型。如果是全部命名空间需要使用：ClusterRole，并且metadata中不指定命名空间。

namespace: default：指定命名空间

apiGroups: [""]：api组

resources: ["pods"]：资源，如pod，deployment，svc等。

verbs: ["get", "watch", "list"]：查看资源方法

- **将test用户绑定到角色**

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: test
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

参数说明:

- kind: User：指定类型，如User，Group，ServiceAccount
- name: test：上面指定的是User，这里就填写用户名

- **执行并测试**

把以上两个yaml中内容合在一起执行。

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: test
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

执行以上yaml后测试：

- **测试查看defaut命名空间下pod**

```
kubectl --kubeconfig=test.kubeconfig get pods
```

结果：可以正常查看

```
[root@k8s-master1 rbac]# kubectl --kubeconfig=test.kubeconfig get pods
NAME                                     READY   STATUS    RESTARTS   AGE
dns-test                                 1/1     Running   0          8h
gateway-679dbc667-4zpl7                  1/1     Running   0          23d
gateway-679dbc667-fh7sk                  1/1     Running   0          23d
nfs-client-provisioner-78f5fc467-hq4w9   1/1     Running   3          89d
nginx-deployment-5c685c9bf-d8thd         1/1     Running   4          89d
nginx-deployment-5c685c9bf-hkwx5         1/1     Running   3          89d
nginx-deployment-5c685c9bf-nijzt         1/1     Running   3          89d
```

- **测试查看default命名空间下其它资源**

```
# 查看service
kubectl --kubeconfig=test.kubeconfig get svc
```

结果：无权限查看其他资源，如service

- **测试查看其他命名空间下pod**

```
# 查看ingress-nginx命令空间下的pod
kubectl --kubeconfig=test.kubeconfig -n ingress-nginx get pod
```

结果:无权限查看

- **测试删除default命名空间下pod**

```
kubectl --kubeconfig=test.kubeconfig delete  pod  web-nginx-5569b9c774-g4fd7
```

结果：无权限

- **修改用户权限**

上面测试只有pod的查看权限，这里再添加`services`/`deployments`两个查看权限

直接修改上面的yaml内容并执行即可，修改内容：

`resources: ["pods","services","deployments"]`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: ["*"]
  resources: ["pods","services","deployments"]
  verbs: ["get", "watch", "list"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: test
  apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

再次测试查default命名空间下service资源

```
kubectl --kubeconfig=test.kubeconfig get svc
```

再次查看可以看到svc

```
[root@k8s-master1 rbac]# kubectl --kubeconfig=test.kubeconfig get svc
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)           AGE
gateway        NodePort    10.0.0.144    <none>        8080:30880/TCP    25d
headless-svc   ClusterIP   None          <none>        18099/TCP         22h
kubernetes     ClusterIP   10.0.0.1      <none>        443/TCP           111d
nginx-test     NodePort    10.0.0.125    <none>        18080:30909/TCP   4d22h
nginx-zj       NodePort    10.0.0.63     <none>        10090:30823/TCP   4d22h
oam            NodePort    10.0.0.107    <none>        8080:30888/TCP    26d
oam-web        NodePort    10.0.0.232    <none>        9000:30900/TCP    91d
test22         ClusterIP   10.0.0.103    <none>        18099/TCP         24h
testheadless   ClusterIP   None          <none>        19999/TCP         22h
utry-db        NodePort    10.0.0.165    <none>        3306:30306/TCP    27d
```

- **其他机器使用kubeconfig访问**

  如果机器上有kubectl，直接把test.kubeconfig拷贝到其他机器使用即可

  如果没有kubetcl命令行，则把其他机器人的复制过来即可：

  ```
  scp /usr/bin/kubectl root@10.0.33.194:/usr/bin/kubectl
  ```

  如果不想指定kubeconfig，可以执行以下操作：

  ```
  # 当前用户的根目录下创建.kube目录
  mkdir ~/.kube
  mv test.kubeconfig ~/.kube/config
  ```

- **扩展:**

  `kubectl api-resources`可以查看apiGroups

  `kubectl get clusterrole` 可以查看系统自带角色

六、Helm应用包管理

1. Helm简介

Helm是一个Kubernetes的包管理工具，就像Linux下的包管理器，如yum/apt等，可以很方便的将之前打包好的yaml文件部署到kubernetes上。

Helm有3个重要概念：

- **helm**：一个命令行客户端工具，主要用于Kubernetes应用chart的创建、打包、发布和管理。
- **Chart**：应用描述，一系列用于描述 k8s 资源相关文件的集合。
- **Release**：基于Chart的部署实体，一个 chart 被 Helm 运行后将会生成对应的一个 release；将在k8s中创建出真实运行的资源对象。

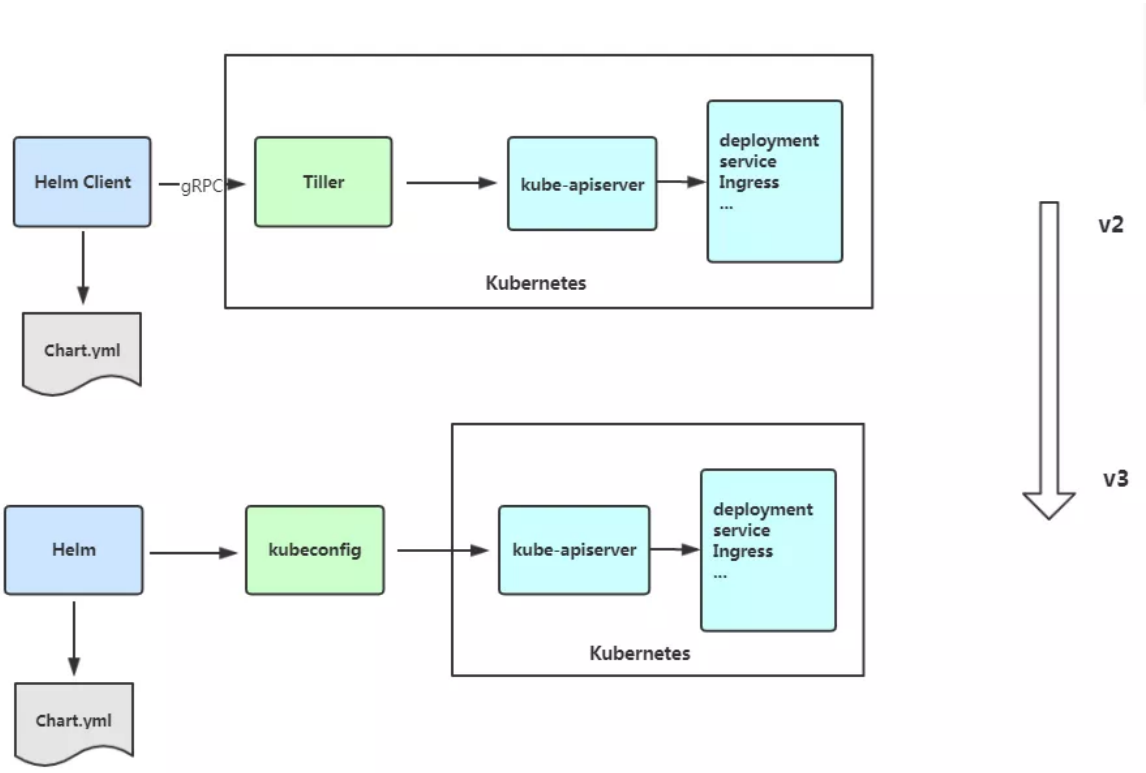
2. Helm v3 变化

2019年11月13日， Helm团队发布 `Helm v3` 的第一个稳定版本。

该版本主要变化如下：

1、 架构变化

最明显的变化是 `Tiller` 的删除



2、 `Release` 名称可以在不同命名空间重用

3、支持将 Chart 推送至 Docker 镜像仓库中

4、使用JSONSchema验证chart values

5、其他

1）为了更好地协调其他包管理者的措辞 `Helm CLI` 个别更名

```
helm delete` 更名为 `helm uninstall
helm inspect` 更名为 `helm show
helm fetch` 更名为 `helm pull
```

但以上旧的命令当前仍能使用。

2）移除了用于本地临时搭建 `Chart Repository` 的 `helm serve` 命令。

3）自动创建名称空间

在不存在的命名空间中创建发行版时，Helm 2创建了命名空间。Helm 3遵循其他Kubernetes对象的行为，如果命名空间不存在则返回错误。

4） 不再需要 `requirements.yaml` , 依赖关系是直接在 `chart.yaml` 中定义。

3.为什么需要helm

K8S上的应用对象，都是由特定的资源描述组成，包括deployment、service等。都保存各自文件中或者集中写到一个配置文件。然后kubectl apply –f 部署。

Deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15
        ports:
        - containerPort: 80
```

Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: web
  name: web
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: web
```

ConfigMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  redis.properties: |
    redis.host=127.0.0.1
    redis.port=6379
    redis.password=123456
```

Ingress.yaml

```
apiVersion:
networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: web
spec:
  rules:
  - host: www.ctnrs.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web
          servicePort: 80
```

如果应用只由一个或几个这样的服务组成，上面部署方式足够了。

而对于一个复杂的应用，会有很多类似上面的资源描述文件，例如微服务架构应用，组成应用的服务可能多达十个，几十个。如果有更新或回滚应用的需求，可能要修改和维护所涉及的大量资源文件，而这种组织和管理应用的方式就显得力不从心了。

且由于缺少对发布过的应用版本管理和控制，使Kubernetes上的应用维护和更新等面临诸多的挑战，主要面临以下问题：

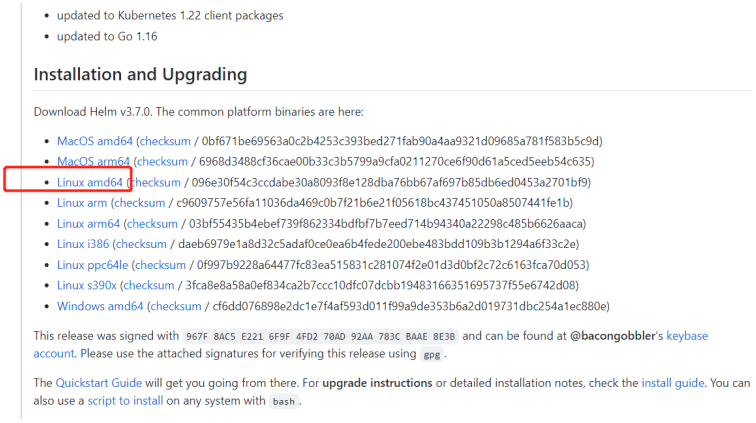
1. **如何将这些服务作为一个整体管理**
2. **这些资源文件如何高效复用**
3. **不支持应用级别的版本管理**

5. helm客户端使用

客户端安装

helm安装非常简单，下载解压即可

Helm客户端下载地址：https://github.com/helm/helm/releases



- updated to Kubernetes 1.22 client packages
- updated to Go 1.16

Installation and Upgrading

Download Helm v3.7.0. The common platform binaries are here:

- MacOS amd64 (checksum / 0bf671be69563a0c2b4253c393bed271fab90a4aa9321d09685a781f583b5c9d)
- MacOS arm64 (checksum / 6968d3488cf36cae00b33c3b5799a9cfa0211270ce6f90d61a5ced5eeb54c635)
- Linux amd64 (checksum / 096e30f54c3ccdabe30a8093f8e128dba76bb67af697b85db6ed0453a2701bf9)
- Linux arm (checksum / c9609757e56fa11036da469c0b7f21b6e21f05618bc437451050a8507441fe1b)
- Linux arm64 (checksum / 03bf55435b4ebef739f862334bdfbf7b7eed714b94340a22298c485b6626aaca)
- Linux i386 (checksum / daeb6979e1a8d32c5adaf0ce0ea6b4fede200ebe483bdd109b3b1294a6f33c2e)
- Linux ppc64le (checksum / 0f997b9228a64477fc83ea515831c281074f2e01d3d0bf2c72c6163fca70d053)
- Linux s390x (checksum / 3fca8e8a58a0ef834ca2b7ccc10dfc07dcbb19483166351695737f55e6742d08)
- Windows amd64 (checksum / cf6dd076898e2dc1e7f4af593d011f99a9de353b6a2d019731dbc254a1ec880e)

This release was signed with `967F 8AC5 E221 6F9F 4FD2 70AD 92AA 783C BAAE 8E3B` and can be found at **@bacongobbler**'s keybase account. Please use the attached signatures for verifying this release using `gpg`.

The Quickstart Guide will get you going from there. For **upgrade instructions** or detailed installation notes, check the install guide. You can also use a script to install on any system with `bash`.

解压移动到/usr/bin/目录即可。

```
# 下载
wget https://get.helm.sh/helm-v3.2.4-linux-amd64.tar.gz
# 解压
tar zxvf helm-v3.2.4-linux-amd64.tar.gz
mv linux-amd64/helm /usr/bin/
```

安装完成后，测试一下

```
helm  version
```

```
[root@k8s-master1 ~]# helm  version
version.BuildInfo{Version:"v3.2.4", GitCommit:"0ad800ef43d3b826f31a5ad8dfbb4fe05d143688", GitTr
eeState:"clean", GoVersion:"go1.13.12"}
[root@k8s-master1 ~]#
```

Helm常用命令

命令	描述
completion	命令补全，source <(helm completion bash)
create	创建一个chart并指定名字
dependency	管理chart依赖
get	下载一个release。可用子命令：all、hooks、manifest、notes、values
history	获取release历史
install	安装一个chart
list	列出release
package	将chart目录打包到chart存档文件中
pull	从远程仓库中下载chart并解压到本地 # helm pull stable/mysql --untar
repo	添加，列出，移除，更新和索引chart仓库。可用子命令：add、index、list、remove、update
rollback	从之前版本回滚
search	根据关键字搜索chart。可用子命令：hub、repo
show	查看chart详细信息。可用子命令：all、chart、readme、values
status	显示已命名版本的状态
template	本地呈现模板
uninstall	卸载一个release
upgrade	更新一个release
version	查看helm客户端版本

配置国内Chart仓库

- 微软仓库（http://mirror.azure.cn/kubernetes/charts/）这个仓库推荐，基本上官网有的chart这里都有。
- 阿里云仓库（https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts ）

  https://apphub.aliyuncs.com/
- 官方仓库（https://hub.kubeapps.com/charts/incubator）官方chart仓库，国内不好使。
- 国内开源社区：http://mirror.kaiyuanshe.cn/kubernetes/charts/

添加存储库：

```
helm repo add stable http://mirror.azure.cn/kubernetes/charts
helm repo add aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
helm repo update
```

查看配置的存储库：

```
helm repo list
```

查找配置库：

```
# 在配置库中查找mysql chart包
helm search repo stable |grep mysql
```

删除配置库：

```
helm repo remove aliyun
```

6. Helm基本使用

主要介绍三个命令：

- helm install
- helm upgrade
- helm rollback

6.1 使用helm安装一个应用

```
# 查找chart,这里使用mysql示例
helm search repo mysql

# 查看chart信息
helm  show chart stable/mysql
```

```
# 安装，格式 helm install 名称 配置库/应用名
helm install db stable/mysql

# 这一步如果报错：Kubernetes cluster unreachable
# 则需要指定下api-server的地址
export KUBERNETES_MASTER=http://127.0.0.1:8080
```

```
# 查看发布状态
 helm status db
```

6.2 自定义chart配置选项

helm install自定义chart配置选项：

上面部署的mysql并没有成功，这是因为并不是所有的chart都能按照默认配置运行成功，可能会需要一些环境依赖，例如PV。

所以我们需要自定义chart配置选项，安装过程中有两种方法可以传递配置数据：

- --values（或-f）：指定带有覆盖的YAML文件。这可以多次指定，最右边的文件优先
- --set：在命令行上指定替代。如果两者都用，--set优先级高

values使用

先将修改的变量写到一个yaml文件中

```
helm  show values stable/mysql > config.yaml
```

修改yaml内容,把数据卷改为自动扩容：

```
  storageClass: "managed-nfs-storage"
  accessMode: ReadWriteOnce
  size: 8Gi
```

```
persistence:
  enabled: true
  ## database data Persistent Volume Storage Class
  ## If defined, storageClassName: <storageClass>
  ## If set to "-", storageClassName: "", which disables dynamic provisioning
  ## If undefined (the default) or set to null, no storageClassName spec is
  ##   set, choosing the default provisioner.  (gp2 on AWS, standard on
  ##   GKE, AWS & OpenStack)
  ##
  # storageClass: "-"
  storageClass: "managed-nfs-storage"
  accessMode: ReadWriteOnce
  size: 8Gi
  annotations: {}

## Use an alternate scheduler, e.g. "stork".
## ref: https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/
##
# schedulerName:
```

添加用户和密码：

```
mysqlUser: test
## Default: random 10 character string
mysqlPassword: test123456
```

修改svc类型为NodePort

```
  type: NodePort
  port: 3306
  nodePort: 32000
```

再次安装：

```
helm  install db  -f config.yaml  stable/mysql
```

查看pod：

```
[root@k8s-master1 ~]# kubectl  get pod
NAME                                     READY   STATUS    RESTARTS   AGE
db-mysql-5fcf65b44d-m897x                1/1     Running   0          94s
nfs-client-provisioner-bb77dd455-nbtmr   1/1     Running   4          7d13h
nginx-deployment-fc6d7c7f5-m4wc9         1/1     Running   4          7d18h
nginx-deployment-fc6d7c7f5-p9wbq         1/1     Running   4          7d18h
nginx-deployment-fc6d7c7f5-rk56h         1/1     Running   2          2d20h
```

查看svc：

```
[root@k8s-master1 ~]# kubectl  get svc
NAME          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
db-mysql      NodePort    10.0.0.99      <none>        3306:32000/TCP   42s
kubernetes    ClusterIP   10.0.0.1       <none>        443/TCP          34d
nginx-web     NodePort    10.0.0.63      <none>        10090:32377/TCP  25d
tomcat-web    NodePort    10.0.0.96      <none>        8080:30990/TCP   9d
web-nginx     NodePort    10.0.0.160     <none>        80:30008/TCP     20d
web3          NodePort    10.0.0.227     <none>        80:30080/TCP     14d
[root@k8s-master1 ~]#
```

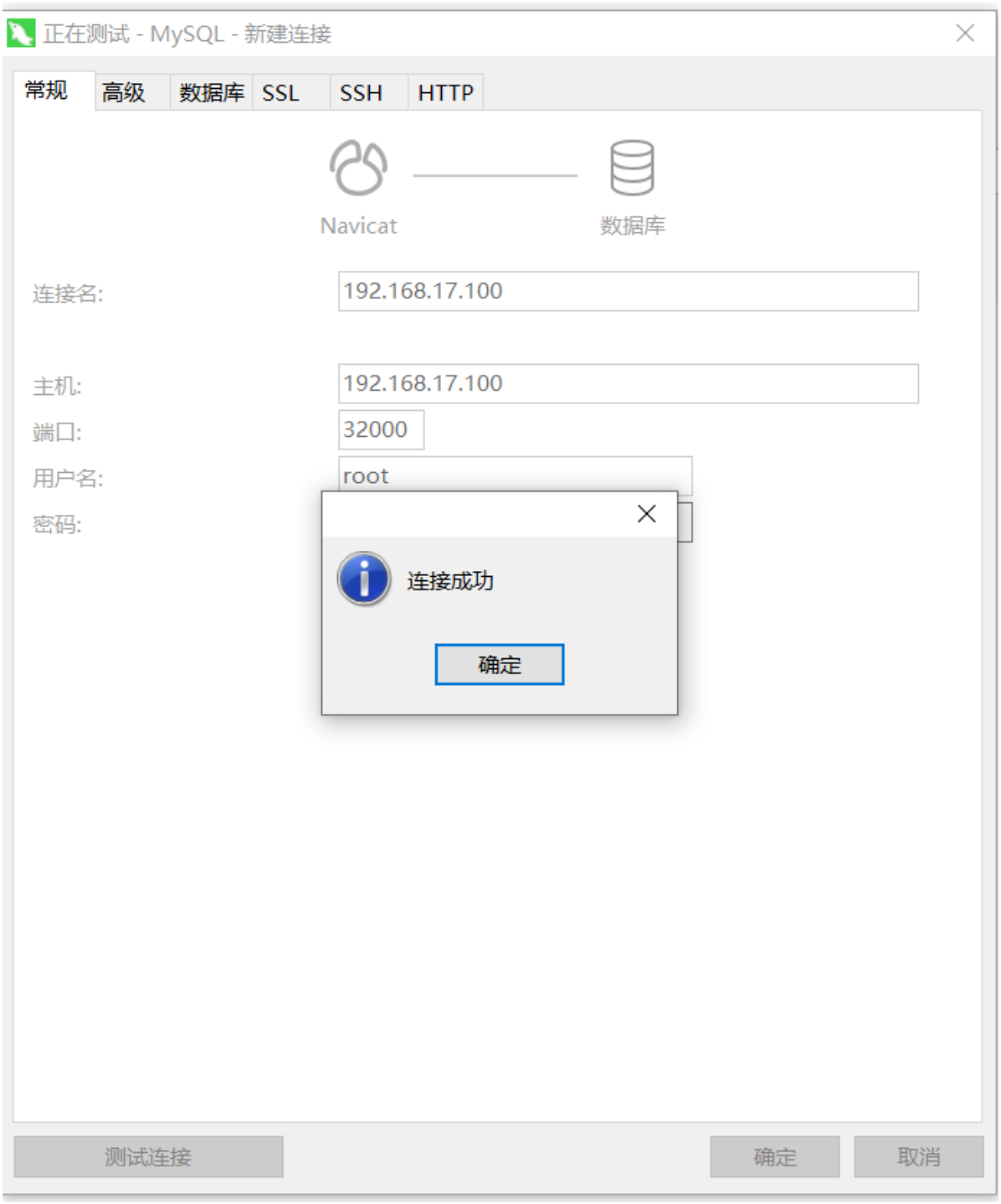
测试：

```
# 查看root默认密码
MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default db-mysql -o
jsonpath="{.data.mysql-root-password}" | base64 --decode; echo)

echo $MYSQL_ROOT_PASSWORD

# 登陆mysql pod中
kubectl  exec -it db-mysql-5fcf65b44d-kdt57 bash
# 在容器中连接mysql
mysql -u root -p6SXU3yvEfI

# 赋予远程连接权限
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'test123456';
flush privileges;
```

```
root@db-mysql-5fcf65b44d-kdt57:/# mysql -u root -p6SXU3yvEfI
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 130
Server version: 5.7.30 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

使用远程工具连接mysql：

192.168.17.100:32000 任意node节点ip+svc端口



set使用

与values不同的是，set直接使用命令行替代变量：

设置动态pv

```
helm install db --set persistence.storageClass="managed-nfs-storage"
stable/mysql
```

也可以把chart包下载下来查看详情：

```
helm pull stable/mysql --untar
```

```
[root@k8s-master1 mysql]# ll
total 36
-rw-r--r-- 1 root root   406 Mar 30 15:44 Chart.yaml
-rw-r--r-- 1 root root 23661 Mar 30 15:44 README.md
drwxr-xr-x 3 root root   268 Mar 30 15:44 templates
-rw-r--r-- 1 root root  6164 Mar 30 15:44 values.yaml
[root@k8s-master1 mysql]#
```

values yaml与set使用：

yaml	set
name: value	--set name=value
a: b c: d	--set a=b,c=d
outer: inner: value	--set outer.inner=value

name:
  - a
  - b
  - c
	--set name={a, b, c}

servers: - port: 80	--set servers[0].port=80

servers:
  - port: 80
  host: example
	--set servers[0].port=80,servers[0].host=example

name: "value1,value2"	--set name=value1\,value2
nodeSelector: kubernetes.io/role: master	--set nodeSelector."kubernetes\.io/role"=master

6.3 install 安装来源

helm install命令可以从多个来源安装

- chart存储库
- 本地chart存档（helm install foo-0.1.1.tgz）
- chart目录（helm install path/to/foo）
- 完整的URL（helm install https://example.com/charts/foo-1.2.3.tgz）

6.5 构建一个Helm Chart

```
# 创建一个chart
helm create mychart

# 查看创建的chart目录结构
```

```
tree mychart/

mychart/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

```
[root@k8s-master1 ~]# helm  create mychart
Creating mychart
[root@k8s-master1 ~]# tree mychart/
mychart/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml

3 directories, 10 files
[root@k8s-master1 ~]#
```

目录结构说明：

- Chart.yaml：用于描述这个 Chart的基本信息，包括名字、描述信息以及版本等。
- values.yaml ： 用于存储 templates 目录中模板文件中用到变量的值。
- Templates： 目录里面存放所有yaml模板文件。
- charts：目录里存放这个chart依赖的所有子chart。
- NOTES.txt ： 用于介绍Chart帮助信息， helm install 部署后展示给用户。例如：如何使用这个 Chart、列出缺省的设置等。
- _helpers.tpl：放置模板助手的地方，可以在整个 chart 中重复使用

创建Chart后，接下来就可以进行部署：

```
# 默认是一个nginx镜像
helm install web mychart/
```

```
[root@k8s-master1 ~]# helm install web mychart/
NAME: web
LAST DEPLOYED: Tue Mar 30 15:58:11 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=mychart,app.kubernetes.io/instance=web" -o jsonpath="{
.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
[root@k8s-master1 ~]#
```

也可以打包推送的charts仓库共享别人使用。

```
helm package mychart/
```

6.6 升级、回滚和删除

发布新版本的chart时，或者当您要更改发布的配置时，可以使用该`helm upgrade`命令。

```
# 设置nginx版本
helm  upgrade --set image.tag=1.17 web mychart

# 或者使用yaml文件方式
# 先把values.yaml中service类型改为NodePort，然后更新
helm upgrade -f mychart/values.yaml web  mychart
```

```
service:
  type: NodePort
  port: 80
```

如果在发布后没有达到预期的效果，则可以使用`helm rollback`回滚到之前的版本。

```
 helm rollback web 2
```

```
[root@k8s-master1 ~]#  helm rollback web 2
Rollback was a success! Happy Helming!
[root@k8s-master1 ~]#
```

查看历史版本配置信息

```
helm get all web
helm  get all --revision 2 web
```

7.Chart模板

Helm最核心的就是模板，即模板化的K8S manifests文件。

它本质上就是一个Go的template模板。Helm在Go template模板的基础上，还会增加很多东西。如一些自定义的元数据信息、扩展的库以及一些类似于编程形式的工作流，例如条件语句、管道等等。这些东西都会使得我们的模板变得更加丰富。

1、模板

有了模板，我们怎么把我们的配置融入进去呢？用的就是这个values文件。这两部分内容其实就是chart的核心功能。

这里使用部署nginx应用，熟悉模板使用，先把templates 目录下面所有文件全部删除掉，这里我们自己来创建模板文件：

```
# 删除template下面文件
 rm -rf mychart/templates/*
# 新建模板文件
vi mychart/templates/deployment.yaml
```

deployment.yaml内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
```

实际上，这已经是一个可安装的Chart包了，通过 `helm install` 命令来进行安装：

```
helm install web mychart
```

```
[root@k8s-master1 ~]# ll mychart
total 8
drwxr-xr-x 2 root root    6 Mar 30 15:54 charts
-rw-r--r-- 1 root root 1098 Mar 30 15:54 Chart.yaml
drwxr-xr-x 2 root root   29 Mar 31 14:47 templates
-rw-r--r-- 1 root root 1799 Mar 30 19:48 values.yaml
[root@k8s-master1 ~]# helm  install web mychart
NAME: web
LAST DEPLOYED: Wed Mar 31 14:55:42 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@k8s-master1 ~]#
```

这样部署，其实与直接apply没什么两样。

然后使用如下命令可以看到实际的模板被渲染过后的资源文件：

```
helm manifest web
```

可以看到，这与刚开始写的内容是一样的，包括名字、镜像等，我们希望能在一个地方统一定义这些会经常变换的字段，这就需要用到Chart的模板了。

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.16
```

```
        name: nginx
```

这个deployment就是一个Go template的模板，这里定义的Release模板对象属于Helm内置的一种对象，是从values文件中读取出来的。这样一来，我们可以将需要变化的地方都定义变量。

再执行helm install chart 可以看到现在生成的名称变成了**web-deployment**，证明已经生效了。也可以使用命令helm get manifest查看最终生成的文件内容。

2、调试

Helm也提供了`--dry-run --debug`调试参数，帮助你验证模板正确性。在执行`helm install`时候带上这两个参数就可以把对应的values值和渲染的资源清单打印出来，而不会真正的去部署一个release。

比如我们来调试上面创建的 chart 包：

```
# helm install web2 --dry-run /root/mychart
```

3、内置对象

刚刚我们使用`{{.Release.Name}}`将 release 的名称插入到模板中。这里的 Release 就是 Helm 的内置对象，下面是一些常用的内置对象：

Release.Name	**release 名称**
Release.Name	release 名字
Release.Namespace	release 命名空间
Release.Service	release 服务的名称
Release.Revision	release 修订版本号，从1开始累加

4、Values

Values对象是为Chart模板提供值，这个对象的值有4个来源：

- chart 包中的 values.yaml 文件
- 父 chart 包的 values.yaml 文件
- 通过 helm install 或者 helm upgrade 的`-f`或者`--values`参数传入的自定义的 yaml 文件
- 通过`--set`参数传入的值

chart 的 values.yaml 提供的值可以被用户提供的 values 文件覆盖，而该文件同样可以被`--set`提供的参数所覆盖。

这里我们来重新编辑 mychart/values.yaml 文件，将默认的值全部清空，然后添加一个副本数：

```
# cat values.yaml
replicas: 3
image: "nginx"
imageTag: "1.17"

# cat templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: {{ .Release.Name }}-deployment
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: {{ .Values.image }}:{{ .Values.imageTag }}
        name: nginx
```

查看渲染结果：

```
# helm install --dry-run web ../mychart/
```

values 文件也可以包含结构化内容，例如：

```
# cat values.yaml
...
label:
  project: ms
  app: nginx

# cat templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-deployment
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      project: {{ .Values.label.project }}
      app: {{ .Values.label.app }}
  template:
    metadata:
      labels:
        project: {{ .Values.label.project }}
        app: {{ .Values.label.app }}
    spec:
      containers:
      - image: {{ .Values.image }}:{{ .Values.imageTag }}
        name: nginx
```

查看渲染结果：

```
# helm install --dry-run web ../mychart/
```

小结

开发Chart大致流程：

1. 先创建模板 helm create demo
2. 修改Chart.yaml，Values.yaml，添加常用的变量
3. 在templates目录下创建部署镜像所需要的yaml文件，并变量引用yaml里经常变动的字段