

基于k8s operator设计实现通过k8s集群下发操作系统配置

西安电子科技大学 XenoWYC121

1. 前期准备

本项目使用的操作系统为RockyLinux9，因为需要搭建k8s集群，至少需要三台虚拟机，一台作为master，两台作为node。虚拟机操作系统安装过程省略。

1.1 K8S安装

1.1.1 虚拟机操作系统安装

略

1.1.2 主机名配置

使用以下命令配置主机名：

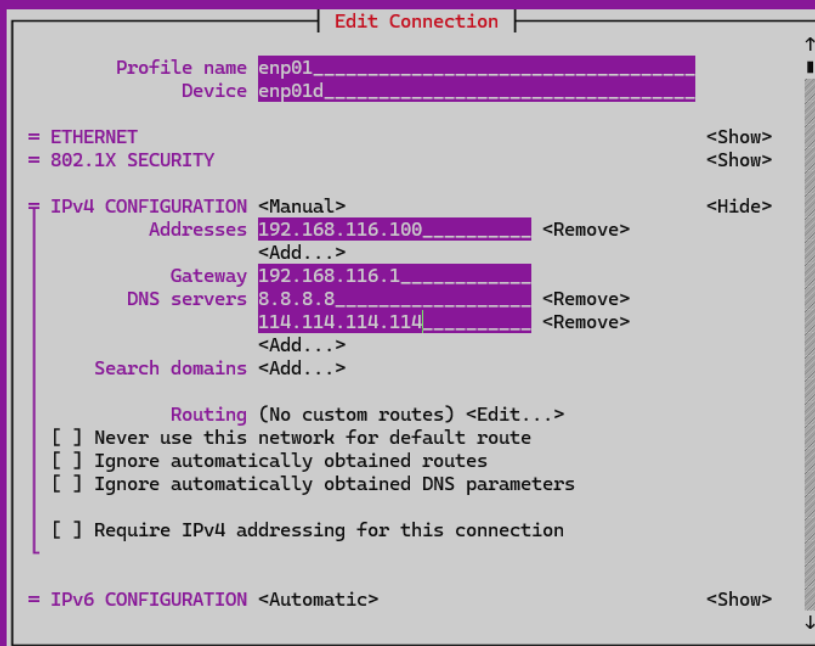
```
hostnamectl set-hostname ${NodeName}
```

三台主机名分别为：master、node1、node2

1.1.3 虚拟机网络配置

1.1.3.1 IP地址配置

使用nmtui命令打开NetworkManager的tui管理界面，配置静态IP地址：



master IP: 192.168.116.100

node1 IP: 192.168.116.101

node2 IP: 192.168.116.102

使用以下命令重启网络服务，使配置文件生效。

```
sudo systemctl restart NetworkManager
```

1.1.3.2 关闭防火墙

使用以下命令关闭防火墙：

```
sudo systemctl disable firewalld --now
```

1.1.4 关闭selinux与swap

```
free -h //查看是否关闭
sudo swapoff -a //暂时关闭
sudo sed -i 's/.*swap.*/#&/' /etc/fstab //永久关闭
free -h
getenforce
cat /etc/selinux/config
sudo setenforce 0
sudo sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config
cat /etc/selinux/config
```

1.1.5 安装docker与containerd

1.24.0版本开始，k8s使用containerd代替docker作为默认容器引擎。

```
# https://docs.docker.com/engine/install/centos/
sudo yum remove docker docker-client docker-client-latest docker-common
docker-
latest docker-latest-logrotate docker-logrotate docker-engine
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
# yum --showduplicates list docker-ce
sudo yum install -y docker-ce docker-ce-cli containerd.io docker-compose-
plugin
sudo yum install -y containerd
# 启动 docker 时，会启动 containerd
# sudo systemctl status containerd.service
sudo systemctl stop containerd.service
# 设置containerd 的默认配置文件
sudo cp /etc/containerd/config.toml /etc/containerd/config.toml.bak
sudo containerd config default > $HOME/config.toml
sudo cp $HOME/config.toml /etc/containerd/config.toml
# 对containerd 换源和设置cgroup的设置
sudo sed -i "s#registry.k8s.io/pause#registry.aliyuncs.com/k8sxio/pause#g"
/etc/containerd/config.toml
# https://kubernetes.io/zh-cn/docs/setup/production-environment/container-
runtimes/#containerd-systemd
# 确保 /etc/containerd/config.toml 中的 disabled_plugins 内不存在 cri
sudo sed -i "s#SystemdCgroup = false#SystemdCgroup = true#g"
/etc/containerd/config.toml
# containerd 忽略证书验证的配置
# [plugins."io.containerd.grpc.v1.cri".registry.configs]
#
[plugins."io.containerd.grpc.v1.cri".registry.configs."192.168.0.12:8001".tl
s]
# insecure_skip_verify = true
sudo systemctl enable --now containerd.service
sudo systemctl start docker.service

sudo systemctl enable docker.service
sudo systemctl enable docker.socket
sudo systemctl list-unit-files | grep docker
sudo mkdir -p /etc/docker
# 对docker换源和cgroup设置
cat<<EOF | sudo tee /etc/docker/daemon.json
{
"registry-mirrors": ["https://mirror.aliyuncs.com"],
"exec-opts": ["native.cgroupdriver=systemd"]
}
```

EOF

```

sudo systemctl daemon-reload
sudo systemctl restart docker
sudo docker info
sudo systemctl status docker.service
sudo systemctl status containerd.service

```

1.1.6 添加k8s仓库

```

cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-
x86_64/
# 是否开启本仓库
enabled=1
# 是否检查 gpg 签名文件
gpgcheck=0
# 是否检查 gpg 签名文件
repo_gpgcheck=0
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF

```

注意：在本项目平台搭建时，国内容器镜像源可以正常使用。截至本报告编写时，大部分国内镜像源已无法使用

1.1.7 内核参数设置

```

# 设置所需的 sysctl 参数，参数在重新启动后保持不变
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1 //这两条让网络连接走iptables（禁用可以不设置）
net.ipv4.ip_forward = 1 //是否允许将一个网络接口收到的数据包转发到另一个网络接口
EOF
# 应用 sysctl 参数而不重新启动
sudo sysctl --system

```

1.1.8 安装1.26.2版本

```

sudo yum install -y kubelet-1.26.2-0 kubeadm-1.26.2-0 kubectl-1.26.2-0 --
disableexcludes=kubernetes --nogpgcheck
systemctl daemon-reload
sudo systemctl restart kubelet
sudo systemctl enable kubelet

```

以上内容需要在node和master机器全部运行一次，下面的操作Master与Node将存在差异

Master上执行

```

kubeadm init --image-repository=registry.aliyuncs.com/google_containers
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
# 或者在环境变量中添加: export KUBECONFIG=/etc/kubernetes/admin.conf
# 添加完环境变量后, 刷新环境变量: source /etc/profile
kubectl cluster-info
# 初始化失败后, 可进行重置, 重置命令: kubeadm reset -
# 执行成功后, 会出现类似下列内容:
# kubeadm join 192.168.116.100:6443 --token f9lv rz.59mykzssqw6vjh32 \
# --discovery-token-ca-cert-hash
# sha256:4e23156e2f71c5df52dfd2b9b198cce5db27c47707564684ea74986836900107
# 查看join命令
# kubeadm token create --print-join-command

```

如果init失败, 查看kubelet的运行日志

```
journalctl -xefu kubelet
```

Node上执行

将Node节点加入到K8s集群中

```

# 运行的内容来自上方执行结果
kubeadm join 192.168.116.100:6443 --token f9lv rz.59mykzssqw6vjh32 \
--discovery-token-ca-cert-hash
sha256:4e23156e2f71c5df52dfd2b9b198cce5db27c47707564684ea74986836900107

```

此时node还是not ready, pod 也没有正常运行, 需要在master配置网络

Master 网络配置

下载calico.yaml

```
wget
https://projectcalico.docs.tigera.io/archive/v3.25/manifests/calico.yaml
```

下载的calico.yaml无法直接使用，需要修改：
使用vim打开calico.yaml：

```
# 在 - name: CLUSTER_TYPE 下方添加如下内容
- name: CLUSTER_TYPE
value: "k8s,bgp"
# 下方为新增内容
- name: IP_AUTODETECTION_METHOD
value: "interface=网卡名称"
```

运行 `kubectl apply -f calico.yaml` 命令，等待网络配置完成。
k8s搭建成功

1.2 operator-sdk

1.2.1 operator-sdk简介

operator-sdk 是一个快速实现 Operator 的工具包，可快速生成 k8s 的 CRD、Controller、Webhook，用户只需要实现业务逻辑。一般创建 Operator 流程如下：

1. 创建工作目录，初始化项目
2. 创建 API，填充字段
3. 创建 Controller，编写核心协调逻辑(Reconcile)
4. 创建 Webhook，实现接口，可选
5. 验证测试
6. 发布到集群中

1.2.2 安装operator-sdk

依照官网提供的命令进行安装

1.2.2.1 设置平台信息

```
export ARCH=$(case $(uname -m) in x86_64) echo -n amd64 ;; aarch64) echo -n
arm64 ;; *) echo -n $(uname -m) ;; esac)
export OS=$(uname | awk '{print tolower($0)}')
```

1.2.2.2 下载对应平台的可执行文件

```
export OPERATOR_SDK_DL_URL=https://github.com/operator-framework/operator-
sdk/releases/download/v1.35.0
curl -LO ${OPERATOR_SDK_DL_URL}/operator-sdk_${OS}_${ARCH}
```

将可执行文件移入/usr/bin路径下，并赋予执行权限。

1.2.3 初始化k8s operator项目

```
operator-sdk init --domain=mydomain.com --repo=github.com/myusername/my-
operator
```

这条命令会创建一个名为 `my-operator` 的项目，并且设置项目域为 `mydomain.com`，指定代码仓库的路径。

运行这条命令后，会自动下载相关依赖。

1.2.4 创建API

```
operator-sdk create api --group=apps --version=v1alpha1 --kind=mykind --
resource --controller
```

这个命令会创建一个 API 组为 `apps`，版本为 `v1alpha1`，资源类型为 `mykind` 的 CRD，并且会生成相应的控制器代码。

至此，前期准备工作完成。

2. 定义CRD

打开 `api/v1alpha1/mykind_types.go` 文件，定义资源规范（Spec）和状态（Status）

MyKind struct

go 语言结构体，定义了CRD所拥有的全部字段与信息，内部已经预定义了如下属性：

```
type MyKind struct {
    metav1.TypeMeta   `json:",inline"` // 类型元信息
    metav1.ObjectMeta `json:"metadata,omitempty"` // 对象元信息
    Spec              MyKindSpec `json:"spec,omitempty"` // 资源规范，由用户定义在
    YAML中
```

```
Status MyKindStatus `json:"status,omitempty"` // 状态，由controller维护
}
```

这些属性是组成CRD的最小内容，可以添加新属性，但是不要删除上述属性。

2.1 MyKindSpec struct

mykind中的 Spec属性的类型，保存有用户可以使用的资源规范，由YAML或Json配置。

```
type MyKindSpec struct {
    Workers []Worker `json:"Workers"` // Workers: 目标主机列表，为空时为集群中
    所有节点
    Http    Http    `json:"Http"` // Http: Http请求
}
```

2.1.1 Worker struct

目标主机，保存有目标主机名称。Controller将解析名称，获取到主机的信息，特别是主机的IP地址。

```
type Worker struct {
    Name string `json:"Name"` // Name : 节点名称
}
```

2.1.2 Http struct

用于构造Http请求，保存有Http请求的路径与参数。

```
type Http struct {
    Port      uint32    `json:"Port"` // Port : 目标主机端口
    Api       string    `json:"Api"`   // Api : 目的接口
    Method    string    `json:"Method"` // Method: 请求方法
    TargetType string    `json:"TargetType"` // TargetType: 下发请求的主机类
    型
    Parameters []Parameter `json:"Parameters"` // Parameters: 参数列表
}
```

2.1.2.1 Parameter struct

Http请求中的参数列表类型，以键值对的方式呈现。


```
// Parameter : HTTP请求参数, 键值对
type Parameter struct {
    Key    string `json:"Key"`
    Value  string `json:"Value"`
}
```

2.2 MyKindStatus struct

用于保存当前资源的状态信息, 供用户查看。由Controller负责维护, 用户无法修改。

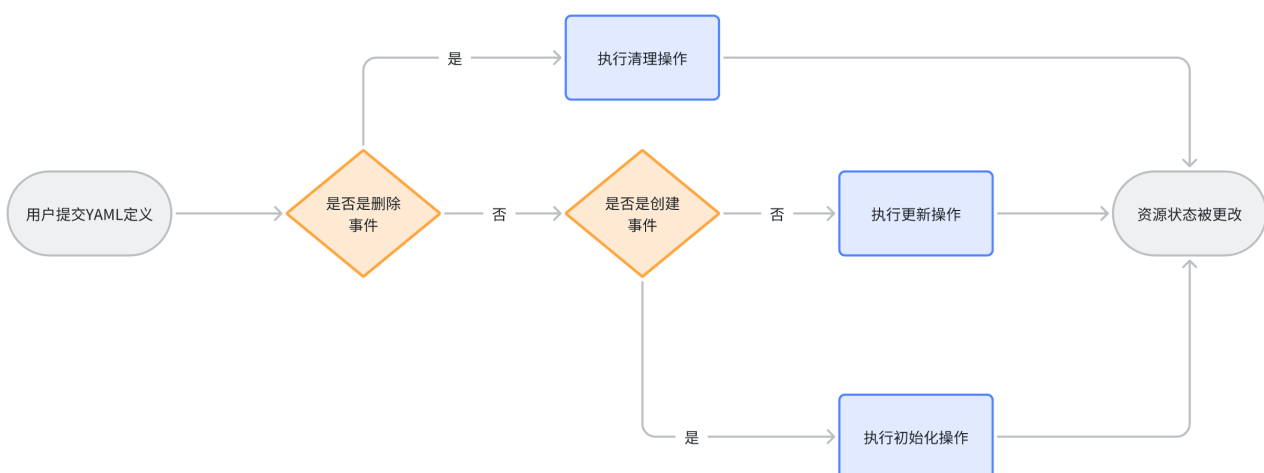
```
// MyKindStatus 定义了资源当前的状态信息
type MyKindStatus struct {
    State string `json:"State"` // State: 上一个请求的执行结果, 成功为SUCCESS,
    // 失败为ERROR
    Msg    string `json:"Msg"` // Msg: 状态详情, 用户失败是显示失败原因。
}
```

State用于保存上一条请求的执行结果, Msg保存执行失败的具体原因。

3. 编写 Controller

3.1 Controller主要流程

在阶段二中, 我们对Controller功能进行了增强, 导致其执行流程与阶段一相比有较大的改变。阶段二中的整体执行流程如下所示:

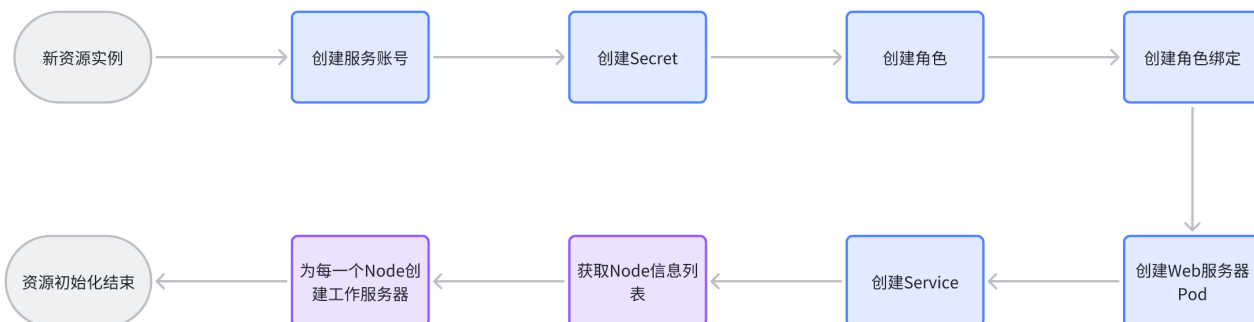


首先Controller将判断是否是删除事件触发了Controller执行, 通过在Controller内部读取上下文的删除关键字即可判断。如果为删除事件, 将会执行清理操作。

如果不是删除事件, Controller将会继续判断是否为创建时间, 此时可以通过读取对象的Status状态, 状态为空时可以认为是在创建资源, 将会执行初始化操作。

最后在既不是删除也不是创建事件时，则可以认为是更新事件，此时只需完成更新操作，完成配置的下发任务即可。

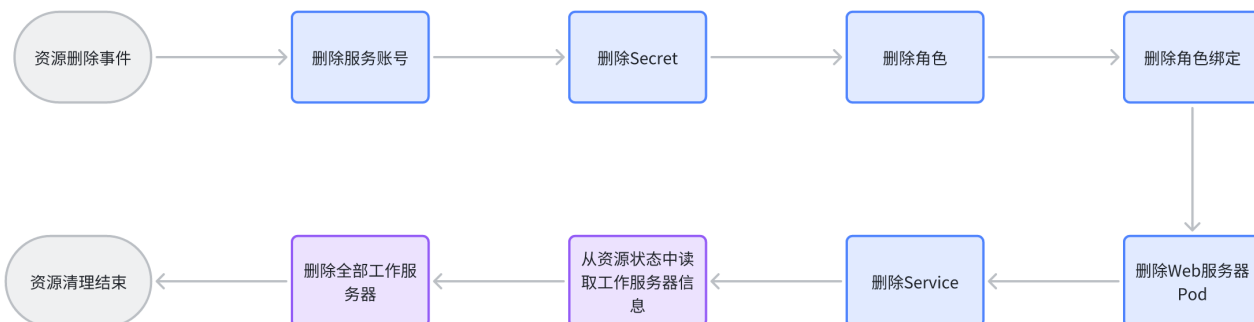
3.2 初始化操作



以上所有的操作完成后都存在一步结果判断过程，在此处省略。关于创建的这些资源有何种作用，见章节4。

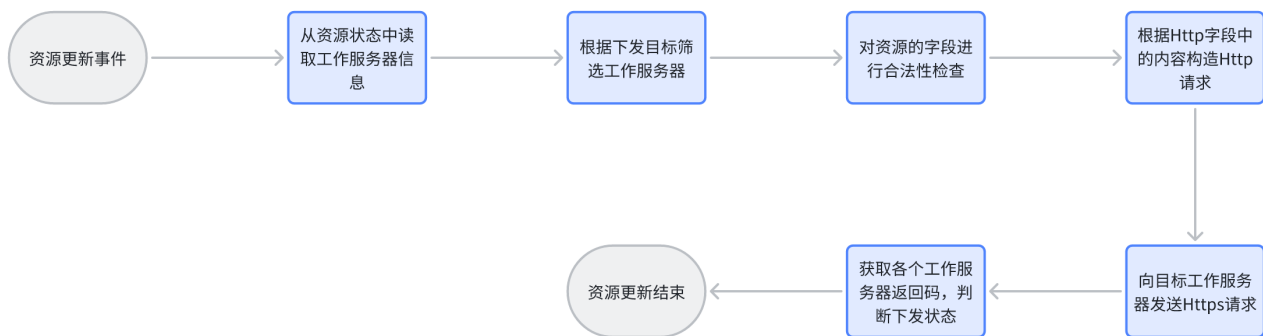
在初始化阶段的主要目标就是创建全部组件以及相关依赖项，蓝色方框代表了创建Web服务器以及相关依赖的操作，紫色方框代表了创建各个Node上的工作服务器的相关操作。

3.3 清理操作



以上所有的操作完成后都存在一步结果判断过程，在此处省略。清理操作的流程基本上就是初始化操作的逆操作，删除全部组件及其依赖的过程。

3.4 更新操作



以上所有的操作完成后都存在一步结果判断过程，在此处省略。更新操作就是下发配置的主要流程。在获取到工作服务信息后，首先根据下发配置的节点类型筛选工作服务器，之后进行合法性检查、构造http请求，最后发送给目标并获取执行结果。

4. 实现web界面

4.1 用户认证

所有 Kubernetes 集群都有两类用户：由 Kubernetes 管理的服务账号和普通用户。Kubernetes 假定普通用户是由一个与集群无关的服务通过以下方式之一进行管理的：

- 负责分发私钥的管理员（证书）
- 类似 Keystone 或者 Google Accounts 这类用户数据库（OIDC、webhook）
- 包含用户名和密码列表的文件（静态令牌）

有鉴于此，Kubernetes 并不包含用来代表普通用户账号的对象。普通用户的信息无法通过 API 调用添加到集群中。尽管无法通过 API 调用来添加普通用户，Kubernetes 仍然认为能够提供由集群的证书机构签名的合法证书的用户是通过身份认证的用户。

与此不同，服务账号是 Kubernetes API 所管理的用户。它们被绑定到特定的名字空间，或者由 API 服务器自动创建，或者通过 API 调用创建。服务账号与一组以 Secret 保存的凭据相关，这些凭据会被挂载到 Pod 中，从而允许集群内的进程访问 Kubernetes API。

API 请求则或者与某普通用户相关联，或者与某服务账号相关联，亦或者被视作匿名请求。这意味着集群内外的每个进程在向 API 服务器发起请求时都必须通过身份认证，否则会被视作匿名用户。这里的进程可以是在某工作站上输入 kubectl 命令的操作人员，也可以是节点上的 kubelet 组件，还可以是调用API的外部进程。

Kubernetes API Server可以同时启用多种身份认证方法，并且通常会至少使用两种方法：

- 针对服务账号使用服务账号令牌认证
- 对用户的身份进行认证的方法

当集群中启用了多个身份认证模块时，第一个成功地对请求完成身份认证的模块会直接做出评估决定。API 服务器并不保证身份认证模块的运行顺序。

k8s提供了多种用户认证方式（详情见文档：《k8s用户认证与鉴权》），因为web服务器需要运行于POD中，使用**服务账号**方式最为方便并且可以保证一定的安全性。web服务器向k8s发送请求时将通过服务账号保存的token进行用户认证。

4.2 用户授权

4.2.1 权限裁定

Kubernetes 对 API 请求的鉴权在 API 服务器内进行。API 服务器根据所有策略评估所有请求属性，可能还会咨询外部服务，然后允许或拒绝该请求。

API 请求的所有部分都必须通过某种鉴权机制才能继续，换句话说：默认情况下拒绝访问。

当系统配置了多个鉴权模块时，Kubernetes 将按顺序使用每个模块。如果任何鉴权模块批准或拒绝请求，则立即返回该决定，并且不会与其他鉴权模块协商。如果所有模块对请求没有意见（该模块不负责处理此类请求），则拒绝该请求。总体拒绝裁决意味着 API 服务器拒绝请求并以 HTTP 403（禁止）状态进行响应。

4.2.2 资源请求

为了确定资源 API 端点的请求动词，Kubernetes 会映射所使用的 HTTP 动词，并考虑该请求是否作用于单个资源或资源集合：

HTTP 动词	请求动词
POST	create
GET 、 HEAD	get （针对单个资源）、 list （针对集合，包括完整的对象内容）、 watch （用于查看单个资源或资源集合）
PUT	update
PATCH	patch
DELETE	delete （针对单个资源）、 deletecollection （针对集合）

请求动词与 Http 动词的映射关系，K8S 内部权限使用请求动词，而 Restful 只支持 Http 方法，需要进行转换。

4.2.3 授权模式

详情见文档《k8s用户认证与鉴权》

结合授权的灵活性与安全性，最终选择RBAC方式进行权限管理。

4.3 创建用户并授权

4.3.1 创建服务账号

创建 service account 有两种方式，可以使用 kubectl 命令，也可以声明 yaml 文件。这里创建的

service account 不需要额外配置，可以直接使用命令：

```
kubectl create sa user1
```

4.3.2 创建角色

同样拥有两种创建方式，此处使用命令。通过以下命令创建 operator-admin 角色，并授予对于 operator 资源的get,list,watch,create,update,patch,delete 权限：

```
kubectl create role operator-admin -
verb=get,list,watch,create,update,patch,delete -resource=operator
```

4.3.3 角色绑定

通过以下命令将 pod-admin 角色与 user1 服务账号绑定，实现用户授权。

```
kubectl create rolebinding user-rolebinding --role=operator-admin -
serviceaccount=default:user1
```

4.3.4 token(secret)生成

注意：如果 k8s 版本小于 1.24，则不需要此步骤。1.24.0 更新之后创建服务账号不会自动生成 Secret 需要对其手动创建。

使用yaml生成对应用户的token，yaml定义如下：

```
apiVersion: v1
kind: Secret
metadata:
  name: user-secret
  namespace: default
  annotations:
    kubernetes.io/service-account.name: "user1"
type: kubernetes.io/service-account-token
```

使用以下命令即可查看生成的 secret:

```
kubectl get secret user-secret -o jsonpath='{.data}'
```

4.4 RESTful API 调用

通过查看k8s官方API文档，K8S官方内置资源资源的API一般为:

```
/api/v1/namespaces/{namespace}/{resource}/{name}
```

K8s第三方资源（如operator）的API一般为:

```
/apis/{domain}/{version}/namespaces/{namespace}/{resource}/{name}
```

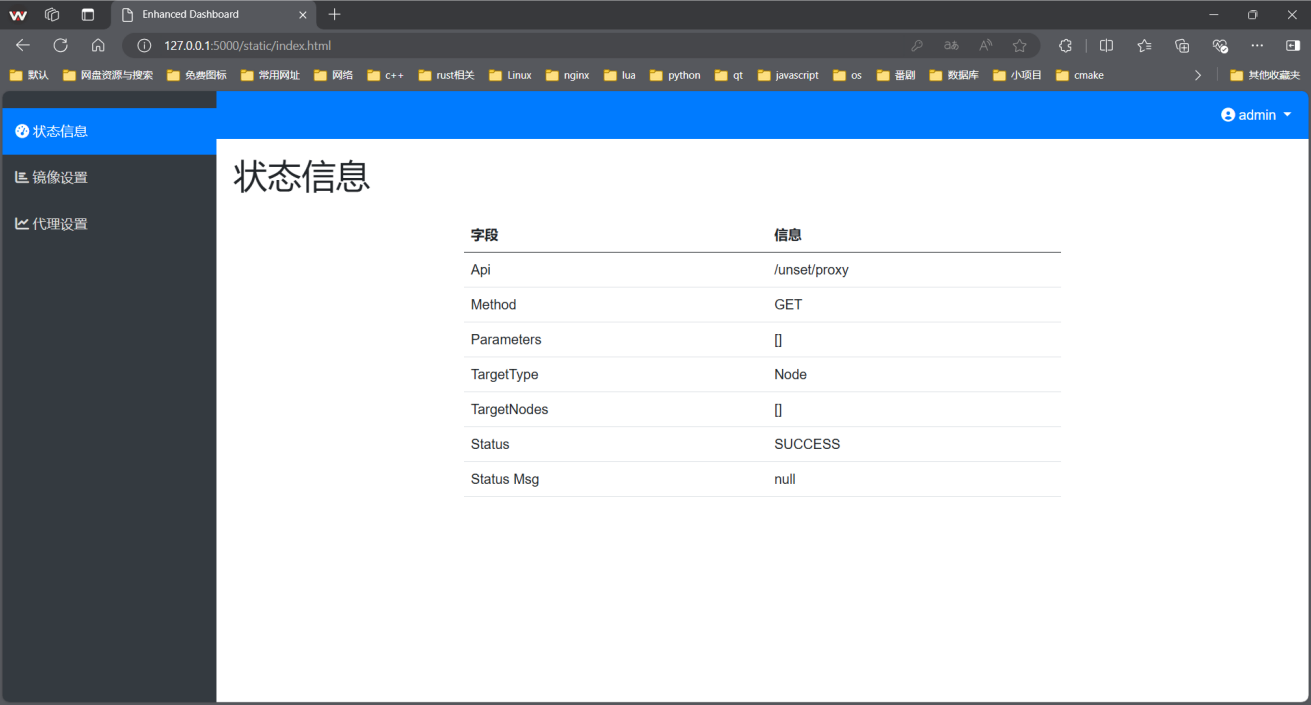
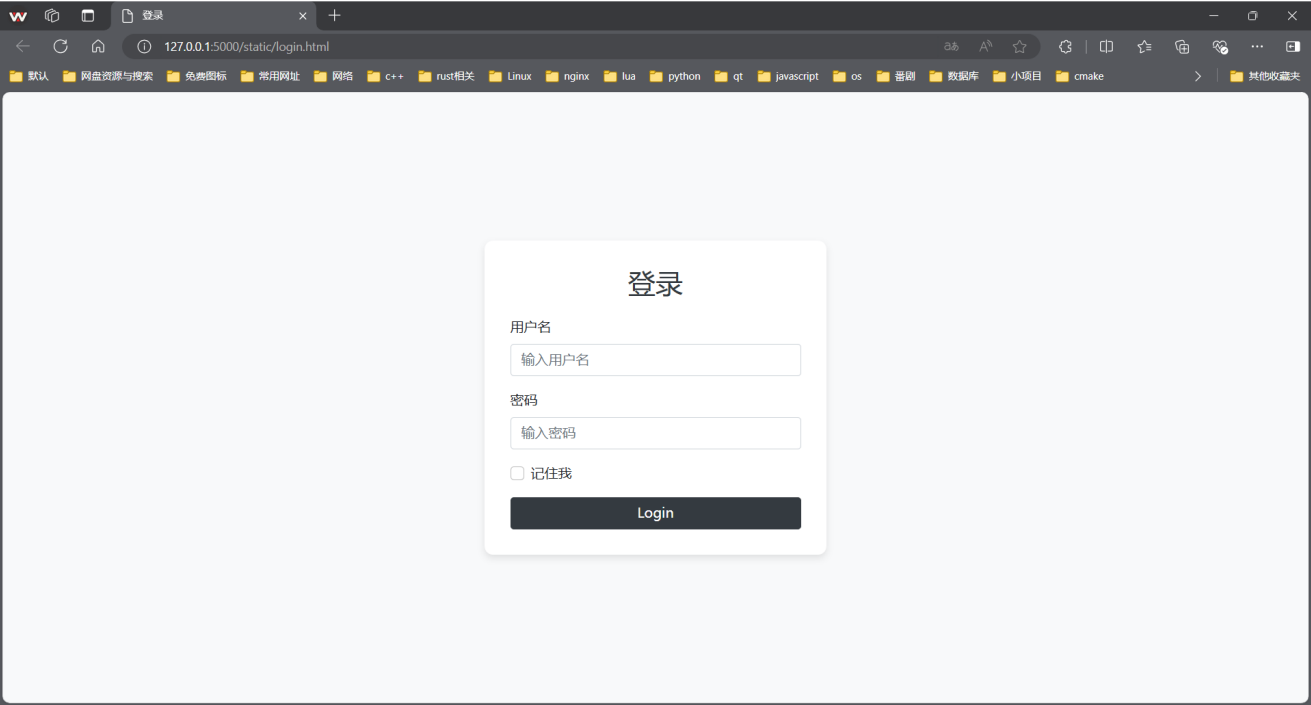
通过以上API在拥有权限的情况下就可以完成对k8s内部资源的操作。

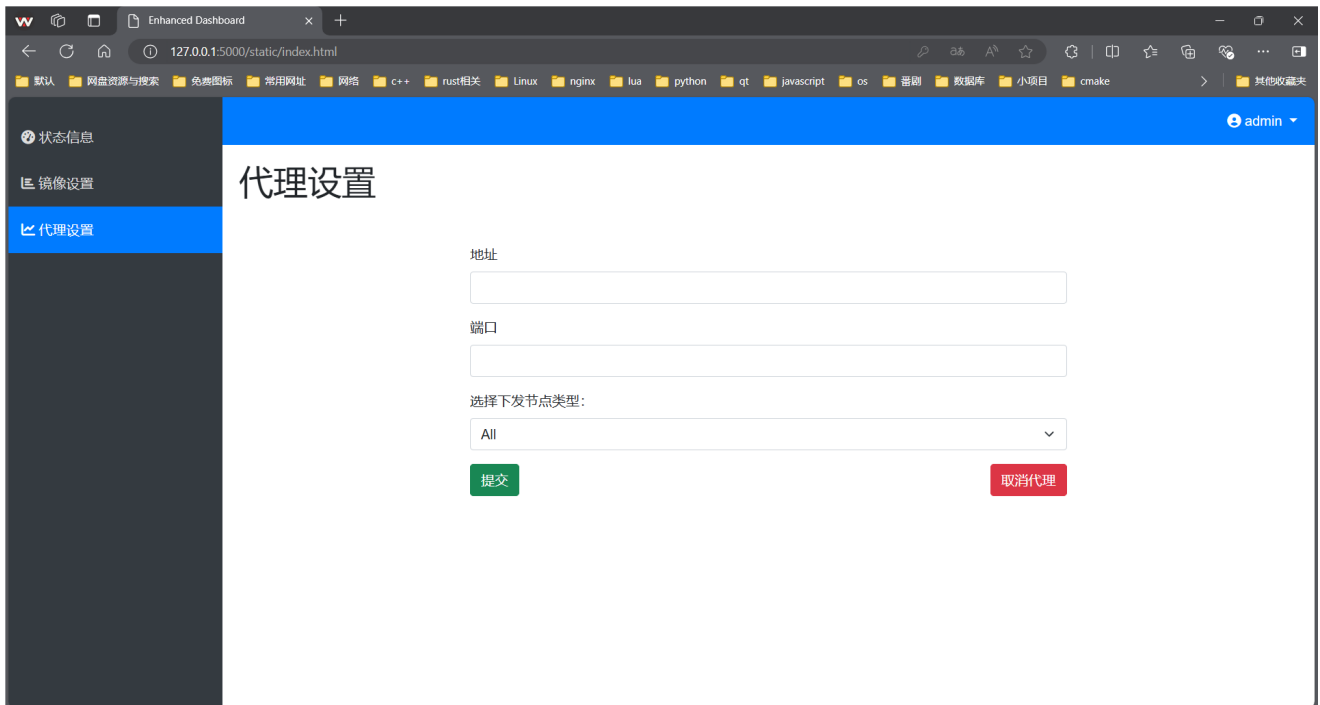
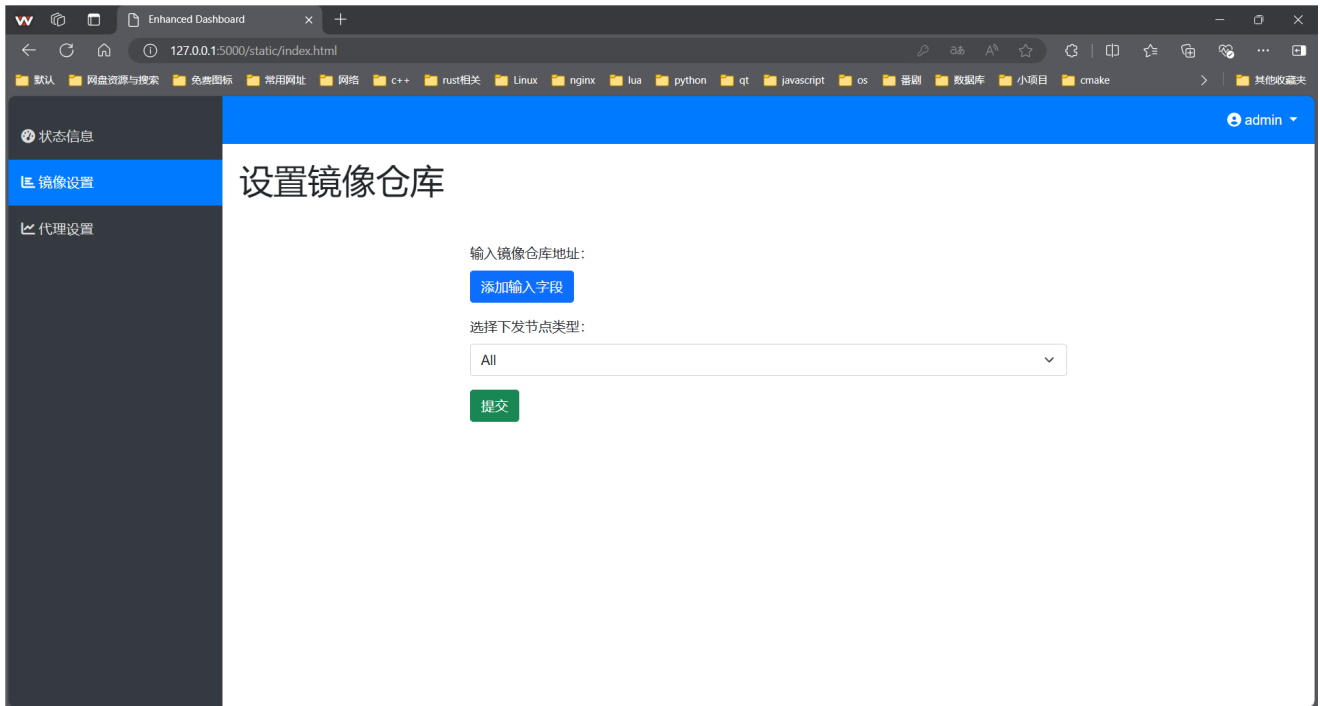
官方文档对于接口的描述过于简单，并且可能返回的状态码有遗漏，通过测试我们自行测试并找到了一些情况写入了文档《K8S RESTful API 使用》与《K8S Restful API返回状态码》中。

4.5 Web实现

4.5.1 前端实现

web前端使用Bootstrap5框架实现，界面简洁大方：





4.5.2 后端实现

后端使用flask框架实现，MVC架构风格，简单且功能强大，分为以下模块：

4.5.2.1 model

使用 `sqlalchemy` 实现了对象关系映射，主要存储了user用户实例，作为登录的凭证。

4.5.2.2 service

负责实现web服务器的业务功能，主要完成与k8s的交互。

4.5.2.3 controller

负责接收请求、路由然后调用相应的service处理请求，返回最后的结果。

4.5.2.4 config

负责存储web服务器的配置信息，比如绑定地址、数据库配置、session密钥等。

5. 实现工作服务器

工作服务器依然使用python flask框架编写，本身仍然是http服务器。基本功能为接收controller发送的配置修改请求，然后修改对应的配置，返回配置修改结果，实现如下：

5.1 controller

负责接收请求、路由然后调用相应的service处理请求，返回最后的结果。

5.2 service

负责实现配置修改功能，读取、修改并写入配置文件。

5.3 Network Policy

使用K8S内置的Network Policy资源，限制外界对工作服务器的访问。

Network Policy是一种用于控制Pod之间以及Pod与其他网络端点之间通信的策略。它允许用户定义规则来指定哪些Pod可以互相通信，哪些不能。Network Policy的工作机制类似于防火墙规则，能够基于标签选择Pod，并设置允许或拒绝的通信流量。

为各个工作服务器pod设置network policy，使其只能被Controller pod访问，外界无法访问。

6. 实现组件容器化

容器OS在不具有SSH服务器情况下，环境的安装与非容器组件配置较为麻烦。在组件容器化之后，运行环境打包到容器中，再无需额外配置。将组件运行于Pod的容器内就可以使用K8S的调度机制，直接使组件运行于相应容器OS主机上。

使用docker工具完成组件容器化。

6.1 工作服务器容器化

6.1.1 Docker镜像构建

工作服务器的镜像打包没什么特别的地方，就是下载基础镜像，将源代码复制到镜像中，再用run关键字安装好相关依赖。

```
FROM hub.atomgit.com/amd64/python:3.11.5-slim-bullseye

WORKDIR /usr/src/app

COPY . .
RUN pip install -i https://mirrors.usc.edu.cn/pypi/web/simple pip -U && pip
config set global.index-url https://mirrors.usc.edu.cn/pypi/web/simple &&
pip install flask

CMD [ "python", "./webserver.py" ]
```

6.1.2 Pod定义

工作服务器需要直接访问主机的/etc目录以达到修改配置的目的，所以需要对Pod定义进行额外修改。需要用到K8S中的HostPath机制，将主机的/etc目录挂载到Pod内运行的容器中。示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: "operator-pod"
  namespace: "default"
spec:
  nodeName: "nodename"
  volumes:
    - name: "host-volume"
      hostPath:
        path: "/etc"
  containers:
    - name: "comm-server"
      volumeMounts:
        - name: "host-volume"
          mountPath: "/host/etc"
      image: "comm_server:v1"
      ports:
        - containerPort: 8000
```

6.2 Web服务器容器化

为了保证可移植性，需要对Web服务器的代码进行一些修改，使其可以通过环境变量获取到k8s api server 访问Token与k8s api server地址（k8s pod中的容器会以环境变量的形式被k8s集群注入一部分配置信息）。

6.2.1 Docker镜像构建

Dockerfile定义如下：

```
FROM hub.atomgit.com/amd64/python:3.11.5-slim-bullseye

WORKDIR /usr/src/app

COPY . .
RUN pip install -i https://mirrors.ustc.edu.cn/pypi/web/simple pip -U && pip
config set global.index-url https://mirrors.ustc.edu.cn/pypi/web/simple &&
pip install flask && pip install flask_sqlalchemy && pip install requests

CMD [ "python", "./app.py" ]
```

下载基础镜像，将源代码复制到镜像中，再用run关键字安装好相关依赖。

6.2.2 Pod定义

需要以设置环境变量，将配置信息注入，好让web服务器可以读取到配置信息。示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server-pod
  labels:
    app: operator-web-server
spec:
  containers:
    - name: web-server-container
      image: web_server:v1
      env:
        - name: K8S_TOKEN
          value: "token value...."
        - name: HOST_ADDRESS
          value: "address value...."
      ports:
        - containerPort: 5000
```

6.3 Controller 容器化

6.3.1 Docker镜像构建

Dockerfile定义如下：

```
# Build the manager binary
FROM hub.atomgit.com/amd64/golang:1.20 AS builder
ARG TARGETOS
ARG TARGETARCH

WORKDIR /workspace

COPY . .

RUN go env -w GOPROXY=https://goproxy.cn,direct

RUN go build -a -o manager cmd/main.go

ENTRYPOINT ["/workspace/manager"]
```

6.3.2 Pod相关定义

除去Pod本身的定义以外，Controller还需要具有内部操作资源的相关权限才能正常运行。

首先定义绑定给Pod的服务账号：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: operator-admin-sa
  namespace: default
```

在拥有服务账号后，给其绑定相对应的权限，创建集群角色与集群角色绑定：

```
# 集群角色
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: operator-admin
rules:
- apiGroups: [""]
  resources: ["serviceaccounts"]
  verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: ["rbac.authorization.k8s.io"]
```

```

resources: ["roles"]
verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["rolebindings"]
  verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["create", "delete", "get", "list", "watch"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["mygroup.my.domain"]
  resources: ["mykinds", "mykinds/status"]
  verbs: ["get", "list", "create", "update", "delete", "patch"]

--- # 分隔符
# 角色绑定
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: operator-admin-binding
subjects:
- kind: ServiceAccount
  name: operator-admin-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: operator-admin
  apiGroup: rbac.authorization.k8s.io

```

集群角色 `operator-admin` 除了拥有Operator类型（mykind）的权限以外，还拥有所要操作的所有资源的相关权限。

在完成上述资源的创建以后，Controller Pod才能正常运行，以下是Controller Pod的定义：

```

apiVersion: v1
kind: Pod
metadata:
  name: operator-controller-pod
  labels:
    app: operator-controller-pod
spec:
  nodeName: master
  serviceName: operator-admin-sa
  containers:

```

```
- name: operator-controller  
  image: 192.168.116.100:5000/mycontroller:latest
```

K8S在接受Pod定义之后将拉取controller镜像，创建Pod并运行。

7. 防火墙配置

关于防火墙的配置与实现，见文档《防火墙配置脚本》。