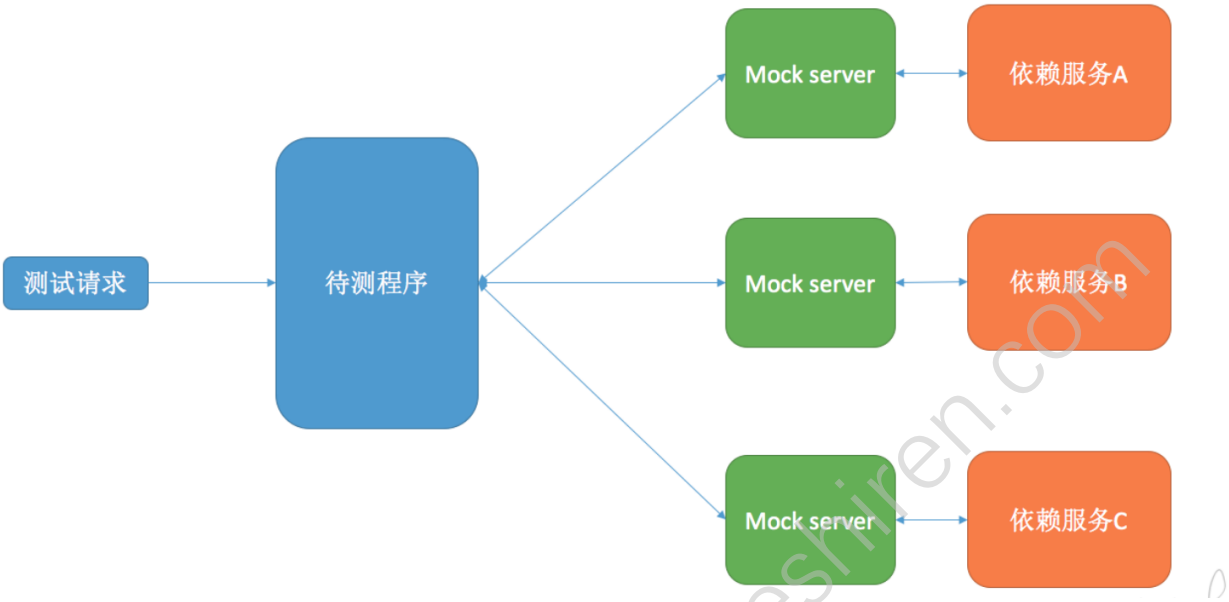
mock server on k8s

mock server的一般玩法



- mock server的目的是一定程度上隔离待测系统，将依赖的其他服务mock掉后达到只测试待测系统的目的。
- mock server拦在待测程序和依赖服务中间， 接收来自待测程序法来的请求。 用户会在mock server中编写规则匹配模式，当发来的请求匹配到规则后(比如header中带有特殊的字段)会返回事前mock好的response。如果没有命中规则，则会把请求转发给真实的服务进行处理。 这样做的目的有二： 1. 系统过于复杂，要完全mock所有接口成本过高，所以这种模式可以只mock核心的，耗时长的接口。 2. 普通用户访问系统不受影响，因为普通用户的header里没带那些特殊字段， 所以正常使用。 而需要mock server的特定程序在测试请求中加了特殊字段所以可以利用mock server的能力，这样使得一个环境可以对双方开放， 互不影响。
- 测试程序可以实现一套case，在连接mock server的时候可以测试，而当把mock server 下掉后依然可以运行的目的。当然这需要对case和mock都有相应的设计。 这样的目的实现该模块自己开发的时候先用mock server进行测试，但是后面集成时完成相应的集成测试。 当然也可以直接维护两套case， 一套对mock server的一套对真实环境的。看每个人的选择

mock server的开源工具

介绍一个我喜欢用的开源mock server： https://www.mock-server.com/mock_server/running_mock_server.html
mock server的规则可支持java， json，js 3种格式。比如java风格的：

```
public class MockServer {
public static void main(String[] args) {

ClientAndServer server = new ClientAndServer("localhost", 8080, 1080);

server.when(
request()
.withMethod("GET")
.withPath("/hello/say")
).respond(
response()
.withBody("mock successfully")
);
server.when(
request()
.withMethod("GET")
.withPath("/test")
.withQueryStringParameters(
param("p", "1")
)
).respond(
response()
.withCookie(new Cookie("cKey", "cValue"))
.withBody("test1")
);

server.when(
request()
.withMethod("GET")
.withPath("/test")
.withQueryStringParameters(
param("p", "2")
)
).respond(
response()
.withBody("test2")
);
}

}
```

- 上面new一个ClientAndServer("localhost", 8080, 1080) 就是在启动一个mock server。 前两个参数是真实服务的ip地址和端口号，mock server一旦发现有请求没有命中事前定义的规则， 那么就会转发给真实的服务处理。 而第三个参数就是mock server自己的端口号了。
- 之后的程序都是规则匹配了， 可以编写规则拦截特定的cookie，header，path，参数。 命中规则后就可以返回特定的response，并且可以设定延迟时间， 也就是可以设定延迟个几秒再返回请求， 这个对于需要测试网络延迟场景的case比较有用，比如我们在混沌工程中注入特定接口的延迟故障，就是这么来的。

也可以是json格式的，比如启动mock server的时候直接指定读取哪个json文件中的规则。 例如：

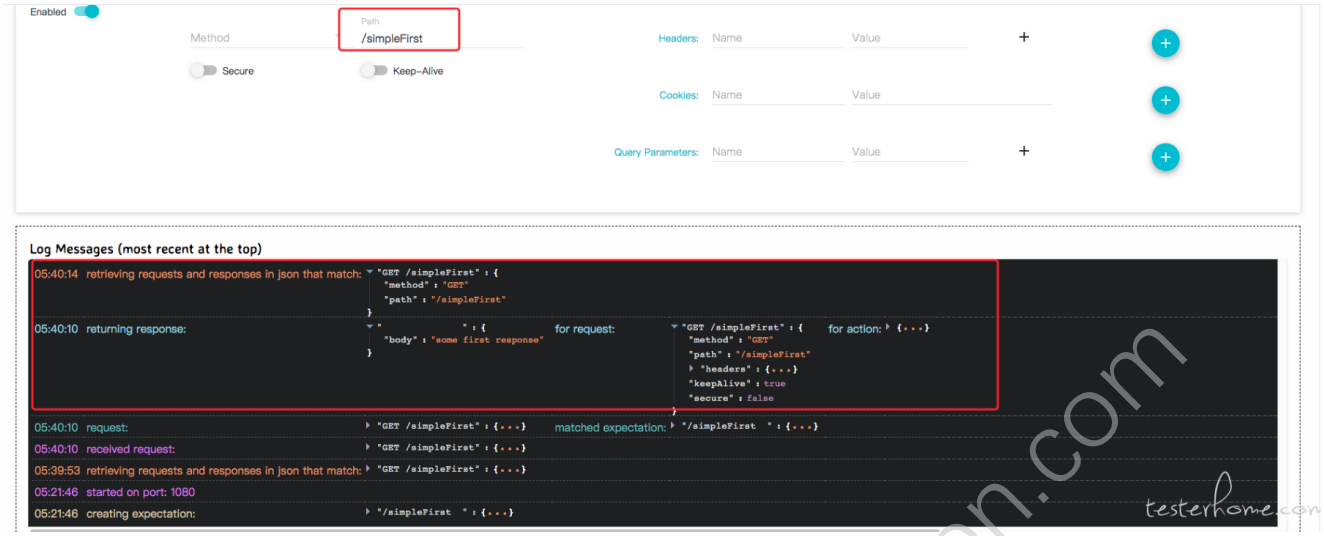
```
java -Dmockserver.initializationJsonPath=${mock_file_path} -jar mockserver-netty-5.8.1-jar-with-dependen
cies.jar -serverPort 1080 -proxyRemotePort ${dport} -logLevel INFO

[
{
"httpRequest": {
"path": "/simpleFirst"
},
"httpResponse": {
"body": "some first response"
}
},
{
"httpRequest": {
"path": "/simpleSecond"
},
"httpResponse": {
"body": "some second response"
}
}
]
```

- 上面代码第一行是启动命令， 在官网上下载mock server的netty包后， 可以指定规则文件的路径
- 上面代码中的json就是规则文件。

具体的语法和规则请大家移步官网

图形界面

mock server本身提供了一个图形界面来实时查看mock server接受到请求和规则命中情况。 如下：

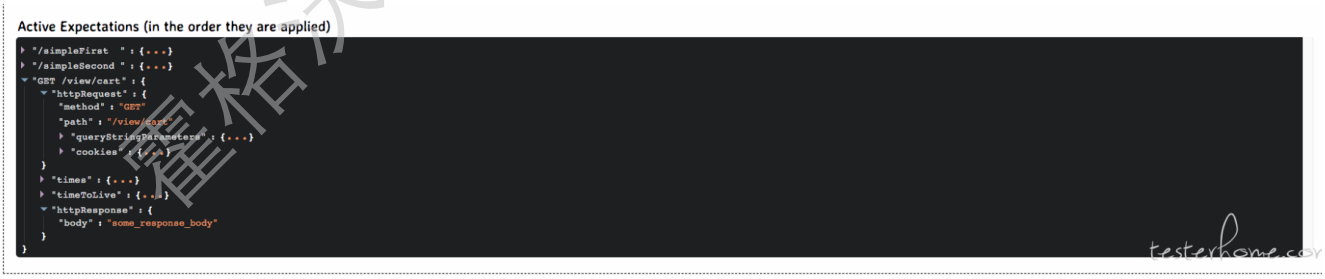


动态加入规则

当mock server启动后，依然是可以通过rest API 动态往里面加入规则的。 比如：

```
curl -v -X PUT "http://172.27.128.8:31234/mockserver/expectation" -d '{
"httpRequest" : {
"method" : "GET",
"path" : "/view/cart",
"queryStringParameters" : {
"cartId" : [ "055CA455-1DF7-45BB-8535-4F83E7266092" ]
},
"cookies" : {
"session" : "4930456C-C718-476F-971F-CB8E047AB349"
}
},
"httpResponse" : {
"body" : "some_response_body"
}
}'
```

这就会给mock server中加入了一个新的规则， 可以在mock server的UI上看到



动态加入规则的目的是在一些比较复杂的情况下， 比如mock server跟部署工具或者环境绑定在一起，
手动起停比较困难的情况下，动态加入有利于作为workround和调式mock 规则时使用。 当然mock server同样提供一个好用的功能，
它会把你动态加入的这些规则保存到一个文件中。 只需要你启动的时候加入两个参数就可以。 比如：

```
java -Dmockserver.persistExpectations=true -Dmockserver.persistedExpectationsPath=mockserverInitializati
on.json  -Dmockserver.initializationJsonPath=${mock_file_path} -jar mockserver-netty-5.8.1-jar-with-depe
ndencies.jar -serverPort 1080 -proxyRemotePort ${dport} -logLevel INFO
```

上面的命令比之前多了两个参数分别是-Dmockserver.persistExpectations=true
-Dmockserver.persistedExpectationsPath=mockserverInitialization.json 。 这两个参数保证了当你在mocker server上动态加入规则后，
这个规则能保存在这个文件里。 比如：

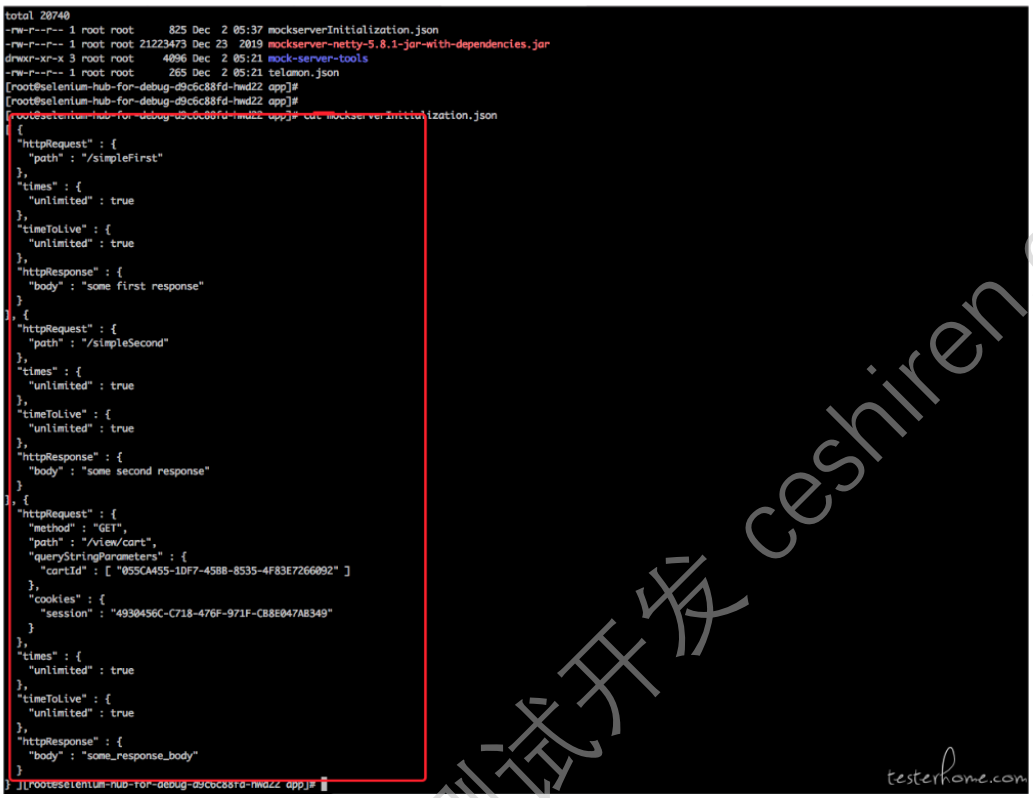
这种模式比较方便你调试mock 规则。

抓取response

有些时候研发的接口文档规范很差， 甚至研发自己都不知道要mock的response长什么样子。 所以我们需要能抓取到实际返回的接口请求。 那么mock server也提供了这样一个功能。 我们可以动态的调用一个api。 如下：

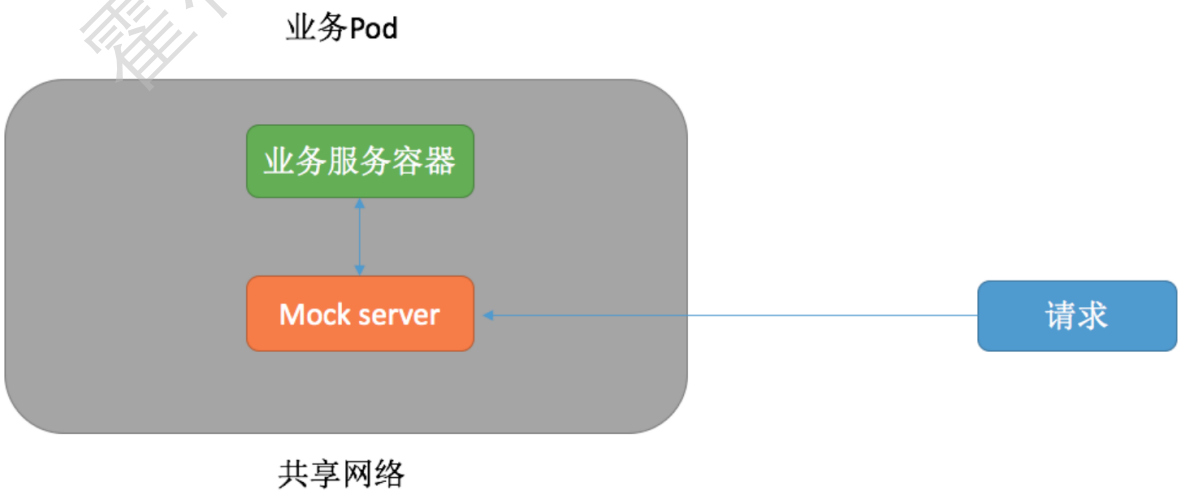
```
curl -v -X PUT "http://172.27.128.8:31234/mockserver/retrieve?type=REQUEST_RESPONSES" -d '{
"path": "/grid/console",
"method": "GET"
}'
```

上面的代码是在从mock server中所有method为GET 路径为/grid/console的请求和相应详情。 效果如下：



在k8s中的玩法

在微服务架构中， 一次测试中可能会需要很多个mock server， 以为一个mock server只能mock一个真实的服务。 那么如何部署这些mock server就是我这几天在研究的。 我们的产品是部署在k8s中的， 借鉴我们在混沌工程中进行故障注入的实践方式，这一次我同样选择使用side car模式向服务所在POD中注入mock server 容器。 如下：



- 第一步先注入一个init container， init container就是pod的初始化容器， 我们注入这个初始化容器是为了使用iptables 来修改网络规则， 把原本应该发送给真实服务的请求转发给mock server
- 第二部注入mock server 容器， 这个容器启动时接管了所有的流量， 命中规则返回mock response， 没有命中规则的话转发给真实服务。

由于在k8s pod中的所有容器都是共享网络名称空间的， 所以这些容器都是天然的网络互通的(用localhost就可以访问了).
通过编写这样的工具，就可以做到对产品的部署方式上进行无入侵的解决方案。

实现方式

- 语言：golang
- 包： k8s开源的client-go

```
package main

import (
"encoding/json"
"flag"
"fmt"
"github.com/pkg/errors"
log "github.com/sirupsen/logrus"
"io/ioutil"
corev1 "k8s.io/api/core/v1"
metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
yamlUtil "k8s.io/apimachinery/pkg/util/yaml"
"k8s.io/client-go/kubernetes"
"k8s.io/client-go/tools/clientcmd"
"os"
"strings"
"time"
)

var (
isRecover bool
kubeConfigPath string
configPath string
)

const (
secretName = "mock-server-secret"
secretFilePath = "pull-secret.yaml"
timeout = 10
initContainerName = "mock-init"
mockServerContainerName = "mock-server"
mockServerContainerImage = "reg.4paradigm.com/qa/mock-server"
)

func init() {
log.SetOutput(os.Stdout)
log.SetLevel(log.DebugLevel)
}

func main() {
flag.BoolVar(&isRecover, "r", false, "podmock server")
flag.StringVar(&kubeConfigPath, "-kubeconfig", "kubeconfig", "k8skubeconfig, k8s")
flag.StringVar(&configPath, "-config", "config.json", "podmock server")
flag.Parse()

log.Info("init the kubeconfig")
kubeConfig, err := clientcmd.BuildConfigFromFlags("", kubeConfigPath)
if err != nil {
log.Error("cannot init the kubeconfig")
panic(err.Error())
}
log.Info("init the k8sclient")
k8s, err := kubernetes.NewForConfig(kubeConfig)
if err != nil {
log.Error("cannot init the k8s client")
panic(err.Error())
}

configs, err := readConfig(configPath)
if err != nil {
handleErr(err, "Failed to load config %s ", "config.json")
}
```

```
deploys := make(map[string]string)
for _, c := range configs {
if isRecover {
log.Debugf("start to reset mock server ns[%s] deployment[%s]", c.Namespace, c.DeploymentName)
err = reset(k8s, c.Namespace, c.DeploymentName)
if err != nil {
handleErr(err, "Failed to reset the mock server. deployment=%s namespace=%s", c.Namespace, c.DeploymentN
ame)
}
deploys[c.DeploymentName] = c.Namespace
} else {
err = addSecrets(k8s,c.Namespace)
if err != nil {
handleErr(err, "Failed to add Secrets to the namespace %s", c.Namespace)
}
log.Debugf("start to inject mock server ns[%s] deployment[%s] dport[%s] mockfile[%s]", c.Namespace, c.De
ploymentName, c.Dport, c.MockFilePath)
err = injectMockServer(k8s, c.Namespace, c.DeploymentName, c.Dport, c.MockFilePath)
if err != nil {
handleErr(err, "Failed to setup the mock server. deployment=%s namespace=%s", c.Namespace, c.DeploymentN
ame)
}
deploys[c.DeploymentName] = c.Namespace
}
}

err = waitDeploymentReady(k8s, deploys)
if err != nil {
fmt.Printf("err: %+v\n", err)
os.Exit(1)
}

log.Info("Done")
}

func handleErr(err error, message string, v ...interface{}) {
err = errors.WithMessagef(err, message, v)
fmt.Printf("err: %+v\n", err)
os.Exit(1)
}

func reset(k8s *kubernetes.Clientset, ns string, deploymentName string) error {
deployment, err := k8s.AppsV1().Deployments(ns).Get(deploymentName, metav1.GetOptions{})
if err != nil {
return errors.Wrap(err, "Failed to get deployment")
}

initContainers := deployment.Spec.Template.Spec.InitContainers
for index, i := range initContainers {
if i.Name == "mock-init" {
initContainers = append(initContainers[:index], initContainers[index+1:]...)
}
}
deployment.Spec.Template.Spec.InitContainers = initContainers

Containers := deployment.Spec.Template.Spec.Containers
for index, i := range Containers {
if i.Name == "mock-server" {
Containers = append(Containers[:index], Containers[index+1:]...)
}
}
deployment.Spec.Template.Spec.Containers = Containers

_, err = k8s.AppsV1().Deployments(ns).Update(deployment)
if err != nil {
return errors.Wrap(err, "Failed to update deployment")
}

return nil
}

func injectMockServer(k8s *kubernetes.Clientset, ns string, deploymentName string, dport string, mockFil
ePath string) error {
deployment, err := k8s.AppsV1().Deployments(ns).Get(deploymentName, metav1.GetOptions{})
if err != nil {
return errors.Wrap(err, "Failed to get deployment")
}
```

```
initContainers := deployment.Spec.Template.Spec.InitContainers
isExist := false
for _, i := range initContainers {
if i.Name == initContainerName {
isExist = true
}
}
//iptables -t nat -A PREROUTING -p tcp --dport 7777 -j REDIRECT --to-port 6666
if !isExist {
s := &corev1.SecurityContext{
Capabilities: &corev1.Capabilities{
Add: []corev1.Capability{"NET_ADMIN"},
},
}

mockServerInitContainer := corev1.Container{
Image:           "biarca/iptables",
ImagePullPolicy: corev1.PullIfNotPresent,
Name:            initContainerName,
Command: []string{
"iptables",
"-t",
"nat",
"-A",
"PREROUTING",
"-p",
"tcp",
"--dport",
dport,
"-j",
"REDIRECT",
"--to-port",
"1080",
},
SecurityContext: s,
}
initContainers = append(initContainers, mockServerInitContainer)
deployment.Spec.Template.Spec.InitContainers = initContainers
}

containers := deployment.Spec.Template.Spec.Containers
isExist = false
for _, i := range containers {
if i.Name == mockServerContainerName {
isExist = true
}
}
if !isExist {
c := corev1.Container{
Image:           mockServerContainerImage,
ImagePullPolicy: corev1.PullAlways,
Name:            mockServerContainerName,
Env: []corev1.EnvVar{
{
Name:  "mock_file_path",
Value: mockFilePath,
},
{
Name:  "dport",
Value: dport,
},
},
}
containers = append(containers, c)
deployment.Spec.Template.Spec.Containers = containers
deployment.Spec.Template.Spec.ImagePullSecrets = append(deployment.Spec.Template.Spec.ImagePullSecrets,
corev1.LocalObjectReference{Name: secretName})

_, err = k8s.AppsV1().Deployments(ns).Update(deployment)
if err != nil {
return errors.Wrap(err, "Failed to update deployment")
}
}
return nil
}
```

```go
type Config struct {
Namespace string `json:"namespace"`
//KubeConfigPath string `json:"kubeconfig_path"`
DeploymentName string `json:"deployment_name"`
Dport          string `json:"dport"`
MockFilePath   string `json:"mock_file_path"`
}

func readConfig(path string) ([]Config, error) {
var config []Config

data, err := ioutil.ReadFile(path)
if err != nil {
return nil, errors.Wrap(err, "ready file failed")
}

err = json.Unmarshal(data, &config)
if err != nil {
return nil, errors.Wrap(err, "unmarshal json failed")
}
return config, nil
}

func addSecrets(k8s *kubernetes.Clientset, ns string) error {
_, err := k8s.CoreV1().Secrets(ns).Get(secretName, metav1.GetOptions{})
if err != nil {
if !strings.Contains(err.Error(), "not found") {
return errors.Wrapf(err, "get secret %s", secretName)
}else {
log.Debugf("namespace[%s] has no secret, now create one", ns)
var (
err    error
data   []byte
)
if data, err = ioutil.ReadFile(secretFilePath); err != nil {
return errors.Wrapf(err, "read %s", secretFilePath)
}
if data, err = yamlUtil.ToJSON(data); err != nil {
return errors.Wrap(err, "covert yaml to json")
}
se := &corev1.Secret{}
if err = json.Unmarshal(data, se); err != nil {
return errors.Wrap(err, "json unmarshal to secret")
}
//cluster := se.ObjectMeta.ClusterName
//namespace := se.ObjectMeta.Namespace
//seName := se.ObjectMeta.Name
se.Namespace = ns
if _, err = k8s.CoreV1().Secrets(ns).Create(se); err != nil {
return errors.Wrap(err, "create secret failed")
}
return nil
}
}
return nil
}


func waitDeploymentReady(k8s *kubernetes.Clientset, deploys map[string]string) error {
now := time.Now()
time.Sleep(time.Second * 1)
for deploymentName, ns := range deploys {
deploy, err := k8s.AppsV1().Deployments(ns).Get(deploymentName, metav1.GetOptions{})
if err != nil {
return errors.Wrapf(err, "Failed to get deployment[%s] ns[%s]", deploymentName, ns)
}

sumReplica := deploy.Status.UnavailableReplicas + deploy.Status.AvailableReplicas
for deploy.Status.ReadyReplicas != *deploy.Spec.Replicas || sumReplica != *deploy.Spec.Replicas {
if time.Now().Sub(now) > time.Minute*timeout {
return errors.Wrapf(err, "deployment is not ready name:%s ns:%s", deploymentName, ns)
}

deploy, err = k8s.AppsV1().Deployments(ns).Get(deploymentName, metav1.GetOptions{})
if err != nil {
return errors.Wrapf(err, "Failed to get deployment[%s] ns[%s]", deploymentName, ns)
}
```

```
time.Sleep(time.Second * 5)
sumReplica = deploy.Status.UnavailableReplicas + deploy.Status.AvailableReplicas
log.Debugf("Waiting: the deploy[%s] the spec replica is %d, readyRelicas is %d, unavail replica is %d,
avail replica is %d",
deploy.Name, *deploy.Spec.Replicas, deploy.Status.ReadyReplicas, deploy.Status.UnavailableReplicas, depl
oy.Status.AvailableReplicas)
}
}
return nil
}
```

dockerfile:

```
FROM docker.4pd.io/base-image-openjdk8:1.0.1

RUN yum install -y git
RUN wget -O "mockserver-netty-5.8.1-jar-with-dependencies.jar" "http://search.maven.org/remotecontent?fi
lepath=org/mock-server/mockserver-netty/5.8.1/mockserver-netty-5.8.1-jar-with-dependencies.jar"

EXPOSE 1080

ENTRYPOINT git clone https://xxxx:xxxx@xxxxxxxxx.git && cp mock-server-tools/${mock_file_path} . && bash
-x java -Dmockserver.persistExpectations=true
-Dmockserver.persistedExpectationsPath=mockserverInitialization.json
-Dmockserver.initializationJsonPath=${mock_file_path} -jar
mockserver-netty-5.8.1-jar-with-dependencies.jar -serverPort 1080 -proxyRemotePort ${dport} -logLevel
INFO
```