link: null
title: 珠峰架构师成长计划
description: service-user-v2.yaml
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=199 sentences=303, words=2359

1.灰度发布

- 灰度发布是一种发布方式，也叫金丝雀发布
- 起源是矿工在下井之前会先放一只金丝雀到井里，如果金丝雀不叫了，就代表瓦斯浓度高。原因是金丝雀对瓦斯气体很敏感
- 灰度发布的做法是：会在现存旧应用的基础上，启动一个新版应用
- 但是新版应用并不会直接让用户访问。而是先让测试同学去进行测试。如果没有问题，则可以将真正的用户流量慢慢导入到新版
- 在这中间，持续对新版本运行状态做观察，直到慢慢切换过去，这就是所谓的A/B测试。当然，你也可以招募一些灰度用户，给他们设置独有的灰度标示（Cookie，Header），来让他们可以访问到新版应用
- 当然，如果中间切换出现问题，也应该将流量迅速地切换到老应用上

1.1 准备新版本Service

```
cp deployment-user-v1.yaml deployment-user-v2.yaml
```

```
apiVersion: apps/v1  #API 配置版本
kind: Deployment     #资源类型
metadata:
+  name: user-v2     #资源名称
spec:
  selector:
    matchLabels:
+      app: user-v2 #告诉deployment根据规则匹配相应的Pod进行控制和管理，matchLabels字段匹配Pod的label值
  replicas: 3 #声明一个 Pod,副本的数量
  template:
    metadata:
      labels:
+        app: user-v2 #Pod的名称
    spec:   #组内创建的 Pod 信息
      containers:
      - name: nginx #容器的名称
+        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v2
        ports:
        - containerPort: 80 #容器内映射的端口
```

service-user-v2.yaml

```
apiVersion: v1
kind: Service
metadata:
+  name: service-user-v2
spec:
  selector:
+    app: user-v2
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: NodePort
```

```
kubectl apply -f deployment-user-v2.yaml service-user-v2.yaml
```

1.2 根据 Cookie 切分流量

- 基于 Cookie 切分流量。这种实现原理主要根据用户请求中的 Cookie 是否存在灰度标示 Cookie去判断是否为灰度用户，再决定是否返回灰度版本服务
- nginx.ingress.kubernetes.io/canary：可选值为 true / false 。代表是否开启灰度功能
- nginx.ingress.kubernetes.io/canary-by-cookie：灰度发布 cookie 的 key。当 key 值等于 always 时，灰度触发生效。等于其他值时，则不会走灰度环境 ingress-gray.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: user-canary
annotations:
  kubernetes.io/ingress.class: nginx
  nginx.ingress.kubernetes.io/rewrite-target: /
  nginx.ingress.kubernetes.io/canary: "true"
  nginx.ingress.kubernetes.io/canary-by-cookie: "vip_user"
spec:
rules:
- http:
    paths:
     - backend:
        serviceName: service-user-v2
        servicePort: 80
backend:
   serviceName: service-user-v2
   servicePort: 80
```

- 生效配置文件

```
kubectl apply -f ./ingress-gray.yaml
```

- 来获取 ingress 的外部端口
- -n: 根据资源名称进行模糊查询

```
kubectl -n ingress-nginx get svc
```

```
curl http:
curl http:
curl --cookie "vip_user=always"  http:
```

- 基于 Header 切分流量，这种实现原理主要根据用户请求中的 header 是否存在灰度标示 header去判断是否为灰度用户，再决定是否返回灰度版本服务

```
vi ingress-gray.yaml
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: user-canary
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/canary: "true"
+    nginx.ingress.kubernetes.io/canary-by-header: "name"
+    nginx.ingress.kubernetes.io/canary-by-header-value: "vip"
spec:
  rules:
  - http:
      paths:
       - backend:
          serviceName: service-user-v2
          servicePort: 80
  backend:
     serviceName: service-user-v2
     servicePort: 80
```

```
kubectl apply -f ingress-gray.yaml
curl --header "name:vip"  http:
```

1.4 基于权重切分流量

- 这种实现原理主要是根据用户请求，通过根据灰度百分比决定是否转发到灰度服务环境中
- nginx.ingress.kubernetes.io/canary-weight：值是字符串，为 0-100 的数字，代表灰度环境命中概率。如果值为 0，则表示不会走灰度。值越大命中概率越大。当值 = 100 时，代表全走灰度

```
vi ingress-gray.yaml
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: user-canary
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/canary: "true"
+   nginx.ingress.kubernetes.io/canary-weight: "50"
spec:
  rules:
  - http:
      paths:
       - backend:
          serviceName: service-user-v2
          servicePort: 80
  backend:
     serviceName: service-user-v2
     servicePort: 80
```

```
kubectl apply -f ingress-gray.yaml
for ((i=1; i10; i++)); do curl http:
```

1.5 优先级

- canary-by-header -> canary-by-cookie -> canary-weight
- k8s 会优先去匹配 header，如果未匹配则去匹配 cookie，最后是 weight

2.滚动发布

- 滚动发布，则是我们一般所说的无宕机发布。其发布方式如同名称一样，一次取出一台/多台服务器（看策略配置）进行新版本更新。当取出的服务器新版确保无问题后，接着采用同等方式更新后面的服务器
- k8s创建副本应用程序的最佳方法就是部署(Deployment)，部署自动创建副本集(ReplicaSet)，副本集可以精确地控制每次替换的Pod数量，从而可以很好的实现滚动更新
- k8s每次使用一个新的副本控制器(replication controller)来替换已存在的副本控制器，从而始终使用一个新的Pod模板来替换旧的pod模板
    - 创建一个新的replication controller
    - 增加或减少pod副本数量，直到满足当前批次期望的数量
    - 删除旧的replication controller

2.1 发布流程和策略

- 优点
    - 不需要停机更新，无感知平滑更新。
    - 版本更新成本小,不需要新旧版本共存
- 缺点
    - 更新时间长：每次只更新一个/多个镜像，需要频繁连续等待服务启动缓冲
    - 旧版本环境无法得到备份：始终只有一个环境存在
    - 回滚版本异常痛苦：如果滚动发布到一半出了问题，回滚时需要使用同样的滚动策略回滚旧版本

2.2 配置文件

- 先扩容为10个副本

```
kubectl get deploy
kubectl scale deployment user-v1  --replicas=10
```

deployment-user-v1.yaml

```
apiVersion: apps/v1  #API 配置版本
kind: Deployment     #资源类型
metadata:
  name: user-v1     #资源名称
spec:
  minReadySeconds: 1
+ strategy:
+   type: RollingUpdate
+   rollingUpdate:
+     maxSurge: 1
+     maxUnavailable: 0
+ selector:
+   matchLabels:
+     app: user-v1 #告诉deployment根据规则匹配相应的Pod进行控制和管理，matchLabels字段匹配Pod的label值
  replicas: 10 #声明一个 Pod,副本的数量
  template:
    metadata:
      labels:
        app: user-v1 #Pod的名称
    spec:   #组内创建的 Pod 信息
      containers:
      - name: nginx #容器的名称
+       image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3 #使用哪个镜像
        ports:
        - containerPort: 80 #容器内映射的端口
```

参数 含义 minReadySeconds 容器接受流量延缓时间：单位为秒，默认为0。如果没有设置的话，k8s会认为容器启动成功后就可以用了。设置该值可以延缓容器流量切分 strategy.type = RollingUpdate ReplicaSet 发布类型，声明为滚动发布，默认也为滚动发布 strategy.rollingUpdate.maxSurge 最多Pod数量：为数字类型/百分比。如果 maxSurge 设置为1，replicas 设置为10，则在发布过程中pod数量最多为10 + 1个（多出来的为旧版本pod，平滑期不可用状态）。maxUnavailable 为 0 时，该值也不能设置为0 strategy.rollingUpdate.maxUnavailable 升级中最多不可用pod的数量：为数字类型/百分比。当 maxSurge 为 0 时，该值也不能设置为0

```
kubectl apply -f ./deployment-user-v1.yaml
deployment.apps/user-v1 configured
```

```
kubectl rollout status deployment/user-v1
Waiting for deployment "user-v1" rollout to finish: 3 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 3 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

Waiting for deployment "user-v1" rollout to finish: 4 of 10 updated replicas are available...

deployment "user-v1" successfully rolled out
```

3.服务可用性探针

3.1 什么是健康度检查？

- 当 Pod 的状态为 Running 时，该 Pod 就可以被分配流量(可以访问到)了
- 一个后端容器启动成功，不一定不代表服务启动成功

3.2 什么是服务探针？

- define-a-TCP-liveness-probe (https://kubernetes.io/zh/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-a-TCP-liveness-probe)
- **3.2.1 存活探针 LivenessProbe #**
- 第一种是存活探针。存活探针是对运行中的容器检测的。如果想检测你的服务在运行中有没有发生崩溃，服务有没有中途退出或无响应，可以使用这个探针
- 如果探针探测到错误， Kubernetes 就会杀掉这个 Pod；否则就不会进行处理。如果默认没有配置这个探针， Pod 不会被杀死

** 3.2.2 可用探针 ReadinessProbe #**

- 第二种是可用探针。作用是用来检测 Pod 是否允许被访问到（是否准备好接受流量）。如果你的服务加载很多数据，或者有其他需求要求在特定情况下不被分配到流量，那么可以用这个探针
- 如果探针检测失败，流量就不会分配给该 Pod。在没有配置该探针的情况下，会一直将流量分配给 Pod。当然，探针检测失败，Pod 不会被杀死

** 3.2.3 启动探针 StartupProbe #**

- 第三种是启动探针。作用是用来检测 Pod 是否已经启动成功。如果你的服务启动需要一些加载时长（例如初始化日志，等待其他调用的服务启动成功）才代表服务启动成功，则可以用这个探针。
- 如果探针检测失败，该 Pod 就会被杀死重启。在没有配置该探针的情况下，默认不会杀死 Pod 。在启动探针运行时，其他所有的探针检测都会失效

探针名称 在哪个环节触发 作用 检测失败对Pod的反应 启动探针 Pod 运行时 检测服务是否启动成功 杀死 Pod 并重启 存活探针 Pod 运行时 检测服务是否崩溃，是否需要重启服务 杀死 Pod 并重启 可用探针 Pod 运行时 检测服务是不是允许被访问到 停止Pod的访问调度，不会被杀死重启

3.3 探测方式

** 3.3.1 ExecAction #**

- 通过在 Pod 的容器内执行预定的 Shell 脚本命令。如果执行的命令没有报错退出（返回值为0），代表容器状态健康。否则就是有问题的

vi shell-probe.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: shell-probe
  name: shell-probe
spec:
  containers:
  - name: shell-probe
    image: registry.aliyuncs.com/google_containers/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

```
kubectl apply -f liveness.yaml
kubectl get pods | grep liveness-exec
kubectl describe pods liveness-exec

Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  2m44s                default-scheduler  Successfully assigned default/liveness-exec to node1
  Normal   Pulled     2m41s                kubelet            Successfully pulled image "registry.aliyuncs.com/google_containers/busybox" in 1.669600584s
  Normal   Pulled     86s                  kubelet            Successfully pulled image "registry.aliyuncs.com/google_containers/busybox" in 605.008964ms
  Warning  Unhealthy  41s (x6 over 2m6s)   kubelet            Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory
  Normal   Killing    41s (x2 over 116s)   kubelet            Container liveness failed liveness probe, will be restarted
  Normal   Created    11s (x3 over 2m41s)  kubelet            Created container liveness
  Normal   Started    11s (x3 over 2m41s)  kubelet            Started container liveness
  Normal   Pulling    11s (x3 over 2m43s)  kubelet            Pulling image "registry.aliyuncs.com/google_containers/busybox"
  Normal   Pulled     11s                  kubelet            Successfully pulled image "registry.aliyuncs.com/google_containers/busybox" in 521.70892ms
```

** 3.3.2 TCPSocketAction #**

- 这种方式是使用 TCP 套接字检测。 Kubernetes 会尝试在 Pod 内与指定的端口进行连接。如果能建立连接（Pod的端口打开了），这个容器就代表是健康的，如果不能，则代表这个 Pod 就是有问题的

tcp-probe.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: tcp-probe
  labels:
    app: tcp-probe
spec:
  containers:
  - name: tcp-probe
    image: nginx
    ports:
    - containerPort: 80
    readinessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
```

```
kubectl apply -f tcp-probe.yaml
kubectl get pods | grep tcp-probe
kubectl describe pods tcp-probe
```

```
kubectl exec -it tcp-probe  -- /bin/sh
apt-get update
apt-get install vim -y
vi  /etc/nginx/conf.d/default.conf
80=>8080
nginx -s reload
kubectl describe pod tcp-probe
Warning  Unhealthy  6s    kubelet            Readiness probe failed: dial tcp 10.244.1.47:80: connect: connection
```

** 3.3.3 HTTPGetAction #**

- 这种方式是使用 HTTP GET 请求。Kubernetes 会尝试访问 Pod 内指定的API路径。如果返回200，代表容器就是健康的。如果不能，代表这个 Pod 是有问题的

vi http-probe.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: http-probe
  name: http-probe
spec:
  containers:
  - name: http-probe
    image: registry.cn-beijing.aliyuncs.com/zhangrenyang/http-probe:1.0.0
    livenessProbe:
      httpGet:
        path: /liveness
        port: 80
        httpHeaders:
        - name: source
          value: probe
      initialDelaySeconds: 3
      periodSeconds: 3
```

```
vim ./http-probe.yaml
kubectl apply -f ./http-probe.yaml
kubectl describe pods http-probe
Normal   Killing    5s                kubelet            Container http-probe failed liveness probe, will be restarted
docker pull registry.cn-beijing.aliyuncs.com/zhangrenyang/http-probe:1.0.0
kubectl replace --force -f http-probe.yaml
```

Dockerfile

```
FROM node
COPY ./app /app
WORKDIR /app
EXPOSE 3000
CMD node index.js
```

```
let http = require('http');
let start = Date.now();
http.createServer(function(req,res){
  if(req.url === '/liveness'){
    let value = req.headers['source'];
    if(value === 'probe'){
     let duration = Date.now()-start;
      if(duration>10*1000){
          res.statusCode=500;
          res.end('error');
      }else{
          res.statusCode=200;
          res.end('success');
      }
    }else{
     res.statusCode=200;
     res.end('liveness');
    }
  }else{
     res.statusCode=200;
     res.end('liveness');
  }
}).listen(3000,function(){console.log("http server started on 3000")});
```

4.储存机密信息

4.1 什么是 Secret

- Secret 是 Kubernetes 内的一种资源类型，可以用它来存放一些机密信息（密码，token，密钥等）
- 信息被存入后，我们可以使用挂载卷的方式挂载进我们的 Pod 内。当然也可以存放docker私有镜像库的登录名和密码，用于拉取私有镜像。

4.2 Opaque 类型

- 一般拿来存放密码，密钥等信息，存储格式为 base64

** 4.2.1 命令行创建 #**

- account为 自定义的名称
- --from-literal 的后面则跟随一组 key=value

```
kubectl create secret generic mysql-account --from-literal=username=zhufeng --from-literal=password=123456
```

```
kubectl get secret
```

字段 含义 NAME Secret的名称 TYPE Secret的类型 DATA 存储内容的数量 AGE 创建到现在的时间

```

kubectl edit secret account

kubectl get secret account -o yaml
//输出json格式
kubectl get secret account -o json
//对Base64进行解码
echo MTIzNDU2 | base64 -d
```

** 4.2.2 配置文件创建 #**

mysql-account.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-account
stringData:
  username: root
  password: root
type: Opaque
```

```
kubectl apply -f mysql-account.yaml
secret/mysql-account created
kubectl get secret mysql-account -o yaml
```

4.3 私有镜像库认证

- 第二种是私有镜像库认证类型，这种类型也比较常用，一般在拉取私有库的镜像时使用

** 4.3.1 命令行创建 #**

```
kubectl create secret docker-registry private-registry \
--docker-username=[用户名] \
--docker-password=[密码] \
--docker-email=[邮箱] \
--docker-server=[私有镜像库地址]
```

```

kubectl get secret private-registry -o yaml
```

```
echo [value] | base64 -d
```

** 4.3.2 通过文件创建 #**

vi private-registry-file.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-file
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodHRwczo
type: kubernetes.io/dockerconfigjson
```

```
kubectl apply -f ./private-registry-file.yaml
kubectl get secret private-registry-file -o yaml
```

4.4 使用

** 4.4.1 Volume 挂载 #**

- 通过存储卷的方式挂载进去

```
apiVersion: apps/v1  #API 配置版本
kind: Deployment     #资源类型
metadata:
  name: user-v1     #资源名称
spec:
  minReadySeconds: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: user-v1 #告诉deployment根据规则匹配相应的Pod进行控制和管理，matchLabels字段匹配Pod的label值
+ replicas: 1 #声明一个 Pod,副本的数量
  template:
    metadata:
      labels:
        app: user-v1 #Pod的名称
    spec:   #组内创建的 Pod 信息
+     volumes:
+       - name: mysql-account
+         secret:
+           secretName: mysql-account
      containers:
      - name: nginx #容器的名称
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3 #使用哪个镜像
+       volumeMounts:
+       - name: mysql-account
+         mountPath: /mysql-account
+         readOnly: true
        ports:
        - containerPort: 80 #容器内映射的端口
```

```
kubectl describe pods  user-v1-b88799944-tjgrs
kubectl exec -it user-v1-b88799944-tjgrs  -- ls /root
```

** 4.4.2 环境变量注入 #**

- 第二种是将 Secret 注入进容器的环境变量

deployment-user-v1.yaml

```
apiVersion: apps/v1  #API 配置版本
kind: Deployment     #资源类型
metadata:
  name: user-v1     #资源名称
spec:
  minReadySeconds: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: user-v1 #告诉deployment根据规则匹配相应的Pod进行控制和管理，matchLabels字段匹配Pod的label值
  replicas: 1 #声明一个 Pod,副本的数量
  template:
    metadata:
      labels:
        app: user-v1 #Pod的名称
    spec:   #组内创建的 Pod 信息
      volumes:
        - name: mysql-account
          secret:
            secretName: mysql-account
      containers:
      - name: nginx #容器的名称
+       env:
+       - name: USERNAME
+         valueFrom:
+           secretKeyRef:
+             name: mysql-account
+             key: username
+       - name: PASSWORD
+         valueFrom:
+           secretKeyRef:
+             name: mysql-account
+             key: password
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3 #使用哪个镜像
        volumeMounts:
        - name: mysql-account
          mountPath: /mysql-account
          readOnly: true
        ports:
        - containerPort: 80 #容器内映射的端口
```

```
kubectl apply -f deployment-user-v1.yaml
kubectl get pods
kubectl describe pod  user-v1-5f48f78d86-hjkcl
kubectl exec -it user-v1-688486759f-9snpx -- env | grep USERNAME
```

** 4.4.3 Docker 私有库认证 #**

- 第三种是 Docker 私有库类型，这种方法只能用来配置 私有镜像库认证。

vi v4.yaml

```
image: [仅有镜像库地址]/[镜像名称]:[镜像标签]
```

```
kubectl apply -f v4.yaml
kubectl get pods
kubectl describe pods [POD_NAME]
```

vi v4.yaml

```
+imagePullSecrets:
+ - name: private-registry-file
containers:
  - name: nginx
```

```
kubectl apply -f v4.yaml
```

5.服务发现

- 当A服务依赖另一个 B服务,而我们常常不知道 B服务 的端口和IP，且端口和IP也相对不固定有可能经常更改
- 这时候，我们就需要 服务发现

5.1 服务发现

- 服务发现是指使用一个注册中心来记录分布式系统中的全部服务的信息，以便其他服务能够快速的找到这些已注册的服务

5.2 CoreDNS

- Pod 的 IP 常常是漂移且不固定的，所以我们要使用 Service 这个神器来将它的访问入口固定住
- 可以利用 DNS 的机制给每个 Service 加一个内部的域名，指向其真实的IP
- 在Kubernetes中，对 Service 的服务发现，是通过一种叫做 CoreDNS 的组件去实现的
- coreDNS 是使用 Go 语言实现的一个DNS服务器
    - -n 按命名空间过滤
    - -l 按标签过滤
    - -o wide 输出额外信息。对于Pod，将输出Pod所在的Node名

```
kubectl -n kube-system get all  -l k8s-app=kube-dns -o wide
```

5.3 服务发现规则

- kubectl exec 的作用是可以直接在容器内执行Shell脚本
    - 命令格式： kubectl exec -it [PodName] -- [Command]
    - -i：即使没有连接，也要保持标准输入保持打开状态。一般与 -t 连用。
    - -t：分配一个伪TTY（终端设备终端窗口），一般与 -i 连用。可以分配给我们一个Shell终端

```
kubectl get pods
kubectl get svc
kubectl exec -it user-v1-688486759f-9snpx -- /bin/sh
curl http://service-user-v2
```

- namespace(命名空间)
    - kubernetes namespace（命名空间）是 kubernetes 里比较重要的一个概念
    - 在启动集群后，kubernetes 会分配一个默认命名空间，叫default。不同的命名空间可以实现资源隔离，服务隔离，甚至权限隔离
    - 因为我们在之前创建的服务，都没有指定 namespace ，所以我们的服务都是在同一个 namespace default下
    - 在同 namespace 下的规则，我们只需要直接访问http://ServiceName:Port (http://ServiceName:Port) 就可以访问到相应的 Service
    - 不同 namespace 下的规则是 [ServiceName].[NameSpace].svc.cluster.local
- ServiceName 就是我们创建的 Service 名称
- NameSpace 则是命名空间。如果你没有命名空间，则这个值为 default。

```
curl http:
```

6.统一管理服务环境变量

- Kubernetes Secret 的主要作用是来存放密码，密钥等机密信息
- 对于环境变量的配置：例如你的数据库地址，负载均衡要转发的服务地址等信息。这部分内容使用 Secret 显然不合适，打包在镜像内耦合又太严重。这里，我们可以借助 Kubernetes ConfigMap 来配置这项事情

6.1 什么是 ConfigMap

- ConfigMap 是 Kubernetes 的一种资源类型，我们可以使用它存放一些环境变量和配置文件
- 信息存入后，我们可以使用挂载卷的方式挂载进我们的 Pod 内，也可以通过环境变量注入
- 和 Secret 类型最大的不同是，存在 ConfigMap 内的内容不会加密

6.2 创建

** 6.2.1 命令行创建 #**

```
kubectl create configmap [config_name] --from-literal=[key]=[value]
```

```
kubectl create configmap mysql-config --from-literal=MYSQL_HOST=192.168.1.172 --from-literal=MYSQL_PORT=3306
```

- 需要注意，configmap 的名称必须是全小写，特殊符号只能包含 '-' 和 '.'。可以用下面的这个正则表达式校验下看看符不符合规则： [a-z0-9]([-a-z0-9]*[a-z0-9])?(\.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*')

```
kubectl get cm
kubectl describe cm mysql-config
```

** 6.2.2 配置清单创建 #**

- kind 的值为 ConfigMap,代表声明一个 ConfigMap 类型的资源
- metadata.name 代表是该 configmap 的名称
- data 是存放数据的地方，数据格式为 key:value

mysql-config-file.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config-file
data:
  MYSQL_HOST: "192.168.1.172"
  MYSQL_PORT: "3306"
```

```
kubectl apply -f ./mysql-config-file.yaml
kubectl describe cm mysql-config-file
```

** 6.2.3 文件创建 #**

- --from-file代表一个文件
- key是文件在 configmap 内的 key
- file_path 是文件的路径

```
kubectl create configmap [configname] --from-file=[key]=[file_path]
```

env.config

```
HOST: 192.168.0.1
PORT: 8080
```

```
kubectl create configmap env-from-file --from-file=env=./env.config
configmap/env-from-file created
```

```
kubectl get cm env-from-file -o yaml
```

** 6.2.4 目录创建 #**

- 也可以直接将一个目录下的文件整个存入进去

```
kubectl create configmap [configname] --from-file=[dir_path]
```

```
mkdir env && cd ./env
echo 'local' > env.local
echo 'test' > env.test
echo 'prod' > env.prod
```

```
kubectl create configmap env-from-dir --from-file=./
```

```
kubectl get cm env-from-dir -o yaml
```

6.3 使用方式

** 6.3.1 环境变量注入 #**

- name 为要选择注入的 configmap 名称
- key 则为 configmap 内的其中一个 key

```
containers:
  - name: nginx #容器的名称
+   env:
+     - name: MYSQL_HOST
+       valueFrom:
+         configMapKeyRef:
+           name: mysql-config
+           key: MYSQL_HOST
```

```
kubectl apply -f ./v1.yaml
```

```

kubectl exec  -it user-v1-744f48d6bd-9klqr -- env | grep MYSQL_HOST
kubectl exec  -it user-v1-744f48d6bd-9klqr  -- env | grep MYSQL_PORT
```

```
      containers:
      - name: nginx #容器的名称
        env:
+       envFrom:
+       - configMapRef:
+           name: mysql-config
+           optional: true
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3 #使用哪个镜像
        volumeMounts:
        - name: mysql-account
          mountPath: /mysql-account
          readOnly: true
        ports:
        - containerPort: 80 #容器内映射的端口
```

** 6.3.2 存储卷挂载 #**

- 存储卷挂载会将 configmap 里内容中的每个 key 和 value，以独立文件方式以外部挂载卷方式挂载进去（ key 是文件名，value 是文件内容）
- 在 Pod 层面声明一个外部存储卷
    - name 为存储卷名称
    - configMap 代表存储卷的文件来源
    - configMap.name 要填入要加载的 configMap 名称
- 在容器镜像层面配置存储卷
    - name 的值来源于第一步配置的 name 值
    - mountPath 为要挂载的目录
    - readonly 则代表文件是不是只读

```
  template:
    metadata:
      labels:
        app: user-v1 #Pod的名称
    spec:   #组内创建的 Pod 信息
      volumes:
        - name: mysql-account
          secret:
            secretName: mysql-account
+       - name: envfiles
+         configMap:
+           name: env-from-dir
      containers:
      - name: nginx #容器的名称
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: mysql-account
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-account
              key: password
        envFrom:
        - configMapRef:
            name: mysql-config
            optional: true
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3 #使用哪个镜像
        volumeMounts:
        - name: mysql-account
          mountPath: /mysql-account
          readOnly: true
+       - name: envfiles
+         mountPath: /envfiles
+         readOnly: true
        ports:
        - containerPort: 80 #容器内映射的端口
```

```
kubectl apply -f deployment-user-v1.yaml
kubectl get pods
kubectl describe pod user-v1-79b8768f54-r56kd
kubectl exec -it user-v1-744f48d6bd-9klqr -- ls /envfiles
```

- 可以借助 volumes.configMap.items[] 字段来配置多个 item 项

```
 spec:   #组内创建的 Pod 信息
      volumes:
        - name: mysql-account
          secret:
            secretName: mysql-account
        - name: envfiles
          configMap:
            name: env-from-dir
+           items:
+           - key: env.local
+             path: env.local
```

7.污点与容忍

- 如何干预Pod 部署到特定的节点上可以通过污点与容忍度去实现

7.1 么是污点和容忍度？

- 在 Kubernetes 中， Pod 被部署到 Node 上面去的规则和逻辑是由 Kubernetes 的调度组件根据 Node 的剩余资源，地位，以及其他规则自动选择调度的
- 但前端和后端往往服务器资源的分配都是不均衡的，甚至有的服务只能让特定的服务器来跑
- 在这种情况下，我们选择自动调度是不均衡的，就需要人工去干预匹配选择规则了
- 这时候，就需要在给 Node 添加一个叫做污点的东西，以确保 Node 不被 Pod 调度到
- 当你给 Node 设置一个污点后，除非给 Pod 设置一个相对应的容忍度，否则 Pod 才能被调度上去。这也就是污点和容忍的来源
- 污点的格式是 key=value，可以自定义自己的内容，就像是一组 Tag 一样
- Node_Name 为要添加污点的 node 名称
- key 和 value 为一组键值对，代表一组标示标签
- NoSchedule 则为不被调度的意思，和它同级别的还有其他的值：PreferNoSchedule 和 NoExecute

```
kubectl taint nodes [Node_Name] [key]=[value]:NoSchedule

kubectl taint nodes node1 user-v4=true:NoSchedule

kubectl describe node node1
kubectl describe node master
Taints: node-role.kubernetes.io/master:NoSchedule
```

vi deployment-user-v4.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-v4
spec:
  minReadySeconds: 1
  selector:
    matchLabels:
      app: user-v4
  replicas: 1
  template:
    metadata:
      labels:
        app: user-v4
    spec:
      containers:
      - name: nginx
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3
        ports:
        - containerPort: 80
```

```
kubectl apply -f deployment-user-v4.yaml
```

- 给 Pod 设置容忍度
    - 想让 Pod 被调度过去，需要在 Pod 一侧添加相同的容忍度才能被调度到
    - 给 Pod 设置一组容忍度，以匹配对应的 Node 的污点
    - key 和 value 是你配置 Node 污点的 key 和 value
    - effect 是 Node 污点的调度效果，和 Node 的设置项也是匹配的
    - operator 是运算符，equal 代表只有 key 和 value 相等才算数。当然也可以配置 exists，代表只要 key 存在就匹配，不需要校验 value 的值

vi deployment-user-v4.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-v4
spec:
  minReadySeconds: 1
  selector:
    matchLabels:
      app: user-v4
  replicas: 1
  template:
    metadata:
      labels:
        app: user-v4
    spec:
+     tolerations:
+     - key: "user-v4"
+       operator: "Equal"
+       value: "true"
+       effect: "NoSchedule"
      containers:
      - name: nginx
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3
        ports:
        - containerPort: 80
```

- 修改Node的污点

```
kubectl taint nodes node1 user-v4=1:NoSchedule --overwrite
```

- 删除 Node 的污点

```
kubectl taint nodes node1 user-v4-
```

- 在master上布署pod

```
kubectl taint nodes node1 user-v4=true:NoSchedule
kubectl describe node node1
kubectl describe node master
```

vi deployment-user-v4.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-v4
spec:
  minReadySeconds: 1
  selector:
    matchLabels:
      app: user-v4
  replicas: 1
  template:
    metadata:
      labels:
        app: user-v4
    spec:
+     tolerations:
+     - key: "node-role.kubernetes.io/master"
+       operator: "Exists"
+       effect: "NoSchedule"
      containers:
      - name: nginx
        image: registry.cn-beijing.aliyuncs.com/zhangrenyang/nginx:user-v3
        ports:
        - containerPort: 80
```

```
kubectl apply -f deployment-user-v4.yaml
```

apiVersion: v1kind: Podmetadata: name: private-regspec: containers: - name: private-reg-container image: imagePullSecrets: - name: har

- Nexus3密码找回 (https://www.jianshu.com/p/02d269c86bbe)
- nexus配置https支持 (https://cloud.tencent.com/developer/article/1671701)