Kubernetes 升级策略解析

1 滚动更新简介

当 kubernetes 集群中的某个服务需要升级时，传统的做法是，先将要更新的服务下线，业务停止后再更新版本和配置，然后重新启动并提供服务。如果业务集群规模较大时，这个工作就变成了一个挑战，而且先全部了停止，再逐步升级的方式会导致服务较长时间不可用。kubernetes提供了滚动更新(rolling-update)的方式来解决上述问题。

简单来说，滚动更新就是针对多实例服务的一种**不中断服务**的更新升级方式。一般情况下，对于多实例服务，滚动更新采用对**各个实例逐个进行单独更新**而**非同一时刻对所有实例进行全部更新**的方式。

对于 k8s 集群部署的 service 来说，rolling update 就是指一次仅更新多个 pod，而不是在同一时刻更新该 service 下面的所有 pod，避免业务中断。

2 Deployment 升级策略

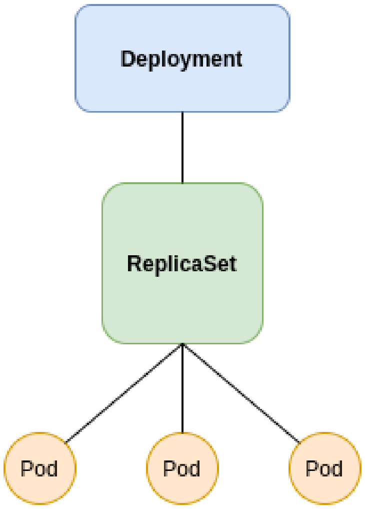
2.1 Deployment 简介

ReplicaSet 是 Kubernetes 系统中的核心核心概念之一。它能保证集群中它所控制的 Pod 的个数永远等于用户期望的个数。

Deployment 可以被认为是 ReplicaSet 的升级，同样保证了集群中的Pod数量，但他实现了“**滚动更新**”这个非常重要的功能。

Deployment 控制器**实际操纵的是 ReplicaSet 对象**，而不是 Pod 对象。

Deployment 实际上是一个两层控制器。首先，它通过 ReplicaSet 的个数来描述应用的版本（一个 ReplicaSet 对应 应用的一个版本）；然后，它再通过 ReplicaSet 的属性（比如 replicas 的值），来保证 Pod 的副本数量。



2.1 Recreate 策略

在该策略下进行更新，旧版本的所有 pods 会被终止，然后创建新版本的 pods，这将导致服务在一段时间内不可用。

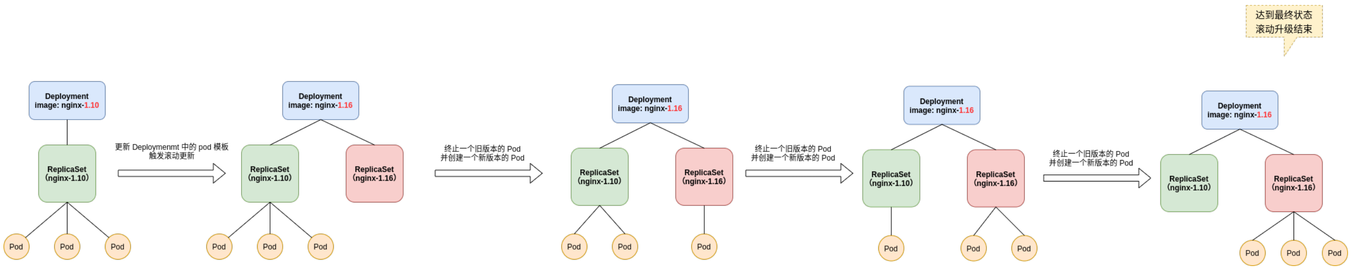
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: busybox
spec:
  strategy:
    //
    type: Recreate
……
```

2.2 RollingUpdate 策略

在该策略下进行更新，会逐个删除旧版本 pods，并创建新版本的 pods，保证服务一直可用。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: busybox
spec:
  strategy:
    #
    type: RollingUpdate
    #
    rollingUpdate:
      maxSurge: 2
      maxUnavaliable: 3
……
```

2.2.1 rolling-update 流程介绍



2.2.2 maxSurge 和 maxUnavailable

服务在滚动更新时，deployment控制器的目的是：给旧版本(old_rs)副本数减少至0、给新版本(new_rs)副本数量增至期望值(replicas)。大家在使用时，通常容易忽视控制速率的特性，在 2.2 节中，我们可以看见 rollingUpdate 下设有两个参数：

1. maxUnavailable：和期望ready的副本数比，不可用副本(可理解为非 ready 状态 的 Pods)数最大比例（或最大值），这个值越小，越能保证服务稳定，更新越平滑。**（当使用百分比时，deployment.spec.replicas * maxUnavailable 会向下取整）**
    - 例如，该值设置成30%，启动 rolling update 后旧的 ReplicatSet 将会立即缩容到期望的 Pod 数量的 70%。新的 Pod ready 后，随着新的 ReplicaSe t的扩容，旧的 ReplicaSet 会进一步缩容确保在升级的所有时刻可以用的 Pod 数量至少是期望 Pod 数量的 70%。
2. maxSurge：和期望ready的副本数比，超过期望副本数最大比例（或最大值），这个值调的越大，副本更新速度越快。**（当使用百分比时，deployment.spec.replicas *maxSurge 会向上取整）**
    - 例如，该值设置成30%，启动 rolling update 后新的 ReplicatSet 将会立即扩容，新老 Pod 的总数不能超过期望的 Pod 数量的 130%。旧的 Pod 被杀掉后，新的 ReplicaSet 将继续扩容，旧的 ReplicaSet 会进一步缩容，确保在升级的所有时刻所有的 Pod 数量和不会超过期望Pod 数量的 130%。

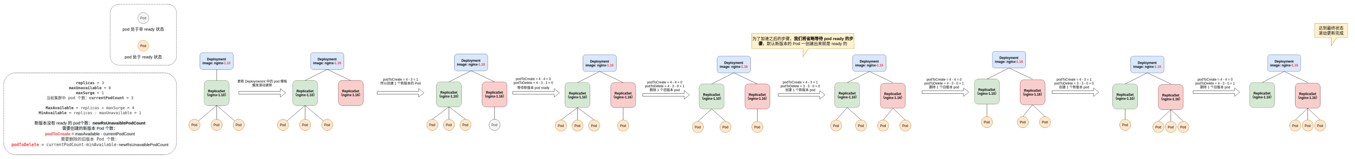
✓ 当只设置 maxSurge 时，滚动更新需要额外资源，可能会导致滚动更新卡住。详见 2.3.1 和 2.3.5 节

```
        replicas  deployment.spec.replicas  Pod
        newRSUnavailablePodCount  Ready  Pod
   MaxAvailable = replicas + maxSurge
       MinAvailable = replicas - maxUnavailable


replcas = 10, maxUnavailable = 2maxSurge = 3 .  MaxAvailable = 13 MinAvailable = 8
        10  pods CurrentPodCount = 10

pod
(1)  pod  = MaxAvailable - CurrentPodCount = 3  3   pods
(2)   3  pod  ready newRSUnavailablePodCount = 3 CurrentPodCount = 13
(3)  pod  = CurrentPodCount - MinAvailable - newRSUnavailablePodCount = 13 - 8 - 3 = 2 2  pods
(4)   3  pod  ready newRSUnavailablePodCount = 3 CurrentPodCount = 11

(5)  pod  = MaxAvailable - CurrentPodCount = 2 2   pods
(6)  1  pod ready 4  pod  ready newRSUnavailablePodCount = 4 CurrentPodCount = 13
(7)  pod  = CurrentPodCount - MinAvailable - newRSUnavailablePodCount = 13 - 8 - 4 = 1 1  pods
(8)  3  pod ready 2  pod  ready newRSUnavailablePodCount = 2 CurrentPodCount = 12

 …………
```

2.3RollingUpdate 策略实践（重点展开）

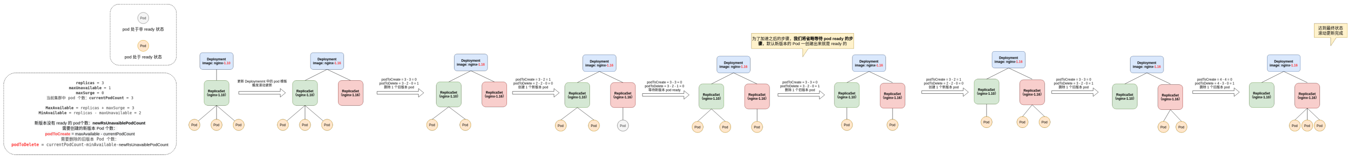
2.3.1maxUnavailable = 0 & maxSurge = n

这种情况下，新版本的 pod 会优先创建。假设 n = 1。


Pod
pod 处于非 ready 状态
Pod
pod 处于 ready 状态
replicas = 3
maxUnavailable = 0
maxSurge = 1
当前集群中 pod 个数: currentPodCount = 3
MaxAvailable = replicas + maxSurge = 4
MinAvailable = replicas - maxUnavailable = 1
新版本没有 ready 的 pod个数: newRsUnavaiblePodCount
需要创建的新版本 Pod 个数:
podToCreate = maxAvailable - currentPodCount
需要删除的旧版本 Pod 个数:
podToDelete = currentPodCount-minAvailable-newRsUnavaiblePodCount
Deployment
image: nginx-1.10
ReplicaSet
(nginx-1.10)
ReplicaSet
(nginx-1.16)
Pod
更新 Deploymenmt 中的 pod 模板
触发滚动更新
podToCreate = 4 - 3 = 1
所以创建 1 个新版本的 Pod
podToCreate = 4 - 4 = 0
podToDelete = 4 - 3 - 1 = 0
等待新版本 pod ready
podToCreate = 4 - 4 = 0
podToDelete = 4 - 3 - 0 = 1
删除 1 个旧版本 pod
为了加速之后的步骤，我们将省略等待 pod ready 的步骤，默认新版本的 Pod 一创建出来就是 ready 的
Deployment
image: nginx-1.16
podToCreate = 4 - 3 = 1
podToDelete = 3 - 3 - 0 = 0
创建 1 个新版本 pod
podToCreate = 4 - 4 = 0
podToDelete = 4 - 3 - 0 = 1
删除 1 个旧版本 pod
podToCreate = 4 - 3 = 1
podToDelete = 3 - 3 - 0 = 0
创建 1 个新版本 pod
podToCreate = 4 - 4 = 0
podToDelete = 4 - 3 - 0 = 1
删除 1 个旧版本 pod
达到最终状态
滚动更新完成

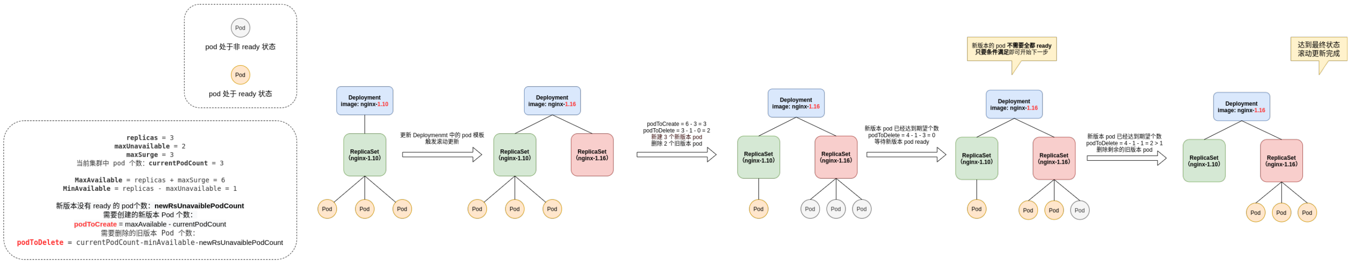
2.3.2 maxUnavailable = n & maxSurge = 0

这种情况下，旧版本的 pod 会优先删除。假设 n = 1。


Pod
pod 处于非 ready 状态
Pod
pod 处于 ready 状态
replicas = 3
maxUnavailable = 1
maxSurge = 0
当前集群中 pod 个数: currentPodCount = 3
MaxAvailable = replicas + maxSurge = 3
MinAvailable = replicas - maxUnavailable = 2
新版本没有 ready 的 pod个数: newRsUnavaiblePodCount
需要创建的新版本 Pod 个数:
podToCreate = maxAvailable - currentPodCount
需要删除的旧版本 Pod 个数:
podToDelete = currentPodCount-minAvailable-newRsUnavaiblePodCount
Deployment
image: nginx-1.10
ReplicaSet
(nginx-1.10)
Pod
更新 Deploymenmt 中的 pod 模板
触发滚动更新
Deployment
image: nginx-1.16
ReplicaSet
(nginx-1.16)
podToCreate = 3 - 3 = 0
podToDelete = 3 - 2 - 0 = 1
删除 1 个旧版本 pod
podToCreate = 3 - 2 = 1
podToDelete = 2 - 2 - 0 = 0
创建 1 个新版本 pod
podToCreate = 3 - 3 = 0
podToDelete = 3 - 2 - 1 = 0
等待新版本 pod ready
为了加速之后的步骤，我们将省略等待 pod ready 的步骤，默认新版本的 Pod 一创建出来就是 ready 的
podToCreate = 3 - 3 = 0
podToDelete = 3 - 2 - 0 = 1
删除 1 个旧版本 pod
podToCreate = 3 - 2 = 1
podToDelete = 2 - 2 - 0 = 0
创建 1 个新版本 pod
podToCreate = 3 - 3 = 0
podToDelete = 3 - 2 - 0 = 1
删除 1 个旧版本 pod
podToCreate = 4 - 4 = 0
podToDelete = 4 - 3 - 0 = 1
删除 1 个旧版本 pod
达到最终状态
滚动更新完成

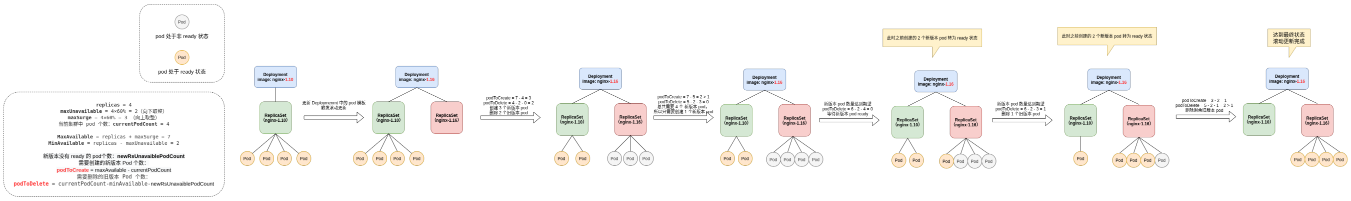
2.3.3 maxUnavailable = x & maxSurge = y

假设 x = 2, y = 3


Pod
pod 处于非 ready 状态
Pod
pod 处于 ready 状态
replicas = 3
maxUnavailable = 2
maxSurge = 3
当前集群中 pod 个数: currentPodCount = 3
MaxAvailable = replicas + maxSurge = 6
MinAvailable = replicas - maxUnavailable = 1
新版本没有 ready 的 pod个数: newRsUnavaiblePodCount
需要创建的新版本 Pod 个数:
podToCreate = maxAvailable - currentPodCount
需要删除的旧版本 Pod 个数:
podToDelete = currentPodCount-minAvailable-newRsUnavaiblePodCount
Deployment
image: nginx-1.10
ReplicaSet
(nginx-1.10)
Pod
更新 Deploymenmt 中的 pod 模板
触发滚动更新
Deployment
image: nginx-1.16
ReplicaSet
(nginx-1.16)
podToCreate = 6 - 3 = 3
podToDelete = 3 - 1 - 0 = 2
新建 3 个新版本 pod
删除 2 个旧版本 pod
新版本 pod 已经达到期望个数
podToDelete = 4 - 1 - 3 = 0
等待新版本 pod ready
新版本的 pod 不需要全都 ready
只要条件满足即可开始下一步
新版本 pod 已经达到期望个数
podToDelete = 4 - 1 - 1 = 2 > 1
删除剩余的旧版本 pod
达到最终状态
滚动更新完成

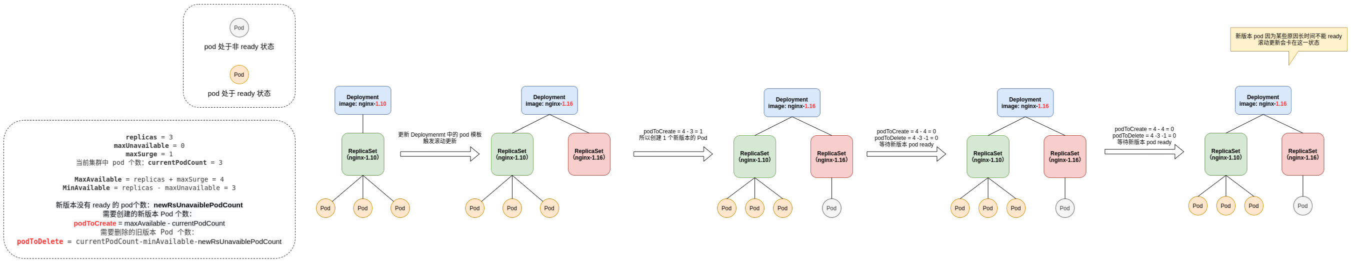
2.3.4 maxUnavailable = x % & maxSurge = y %

假设 x = 60, y = 60


Pod
pod 处于非 ready 状态
Pod
pod 处于 ready 状态
replicas = 4
maxUnavailable = 4×60% = 2 (向下取整)
maxSurge = 4×60% = 3 (向上取整)
当前集群中 pod 个数: currentPodCount = 4
MaxAvailable = replicas + maxSurge = 7
MinAvailable = replicas - maxUnavailable = 2
新版本没有 ready 的 pod个数: newRsUnavaiblePodCount
需要创建的新版本 Pod 个数:
podToCreate = maxAvailable - currentPodCount
需要删除的旧版本 Pod 个数:
podToDelete = currentPodCount-minAvailable-newRsUnavaiblePodCount
Deployment
image: nginx-1.10
ReplicaSet
(nginx-1.10)
Pod
更新 Deploymenmt 中的 pod 模板
触发滚动更新
Deployment
image: nginx-1.16
ReplicaSet
(nginx-1.16)
podToCreate = 7 - 4 = 3
podToDelete = 4 - 2 - 0 = 2
创建 3 个新版本 pod
删除 2 个旧版本 pod
podToCreate = 7 - 5 = 2 > 1
podToDelete = 5 - 2 - 3 = 0
总共需要 4 个 新版本 pod,
所以只需要创建 1 个新版本 pod
此时之前创建的 2 个新版本 pod 转为 ready 状态
新版本 pod 数量达到期望
podToDelete = 6 - 2 - 4 = 0
等待新版本 pod ready
此时之前创建的 2 个新版本 pod 转为 ready 状态
新版本 pod 数量达到期望
podToDelete = 6 - 2 - 3 = 1
删除 1 个旧版本 pod
podToCreate = 3 - 2 = 1
podToDelete = 5 - 2 - 1 = 2 > 1
删除剩余旧版本 pod
达到最终状态
滚动更新完成

2.3.5 可能出现的异常状况

1. 滚动更新卡住：这种情况应该优先排查新版本 pod 不能 ready 原因。可能因为**资源不足**导致 pod pending。


Pod
pod 处于非 ready 状态
Pod
pod 处于 ready 状态
replicas = 3
maxUnavailable = 0
maxSurge = 1
当前集群中 pod 个数: currentPodCount = 3
MaxAvailable = replicas + maxSurge = 4
MinAvailable = replicas - maxUnavailable = 3
新版本没有 ready 的 pod个数: newRsUnavaiblePodCount
需要创建的新版本 Pod 个数:
podToCreate = maxAvailable - currentPodCount
需要删除的旧版本 Pod 个数:
podToDelete = currentPodCount-minAvailable-newRsUnavaiblePodCount
Deployment
image: nginx-1.10
ReplicaSet
(nginx-1.10)
Pod
更新 Deploymenmt 中的 pod 模板
触发滚动更新
Deployment
image: nginx-1.16
ReplicaSet
(nginx-1.16)
podToCreate = 4 - 3 = 1
所以创建 1 个新版本的 Pod
podToCreate = 4 - 4 = 0
podToDelete = 4 -3 -1 = 0
等待新版本 pod ready
podToCreate = 4 - 4 = 0
podToDelete = 4 -3 -1 = 0
等待新版本 pod ready
新版本 pod 因为某些原因长时间不能 ready
滚动更新会卡在这一状态

3 StatefulSet 升级策略

3.1OnDelete 策略

在该策略下，**用户需要手动删除**旧版本的 pod，控制器会创建新版本的 pod

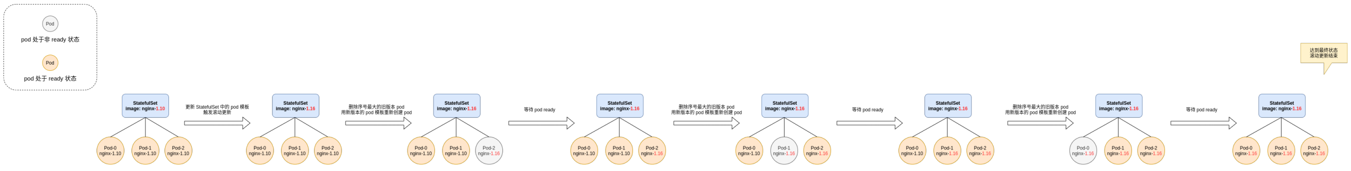
```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: busybox
spec:
  updateStrategy:
    type: OnDelete
……
```

3.2RollingUpdate 策略

StatefulSet 的滚动更新相对简单，它会从序号最大的旧版本 pod 开始重建更新 pod，**当新版本 pod ready 以后**，继续更新序号最大的旧版本 pod

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: busybox
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
……
```

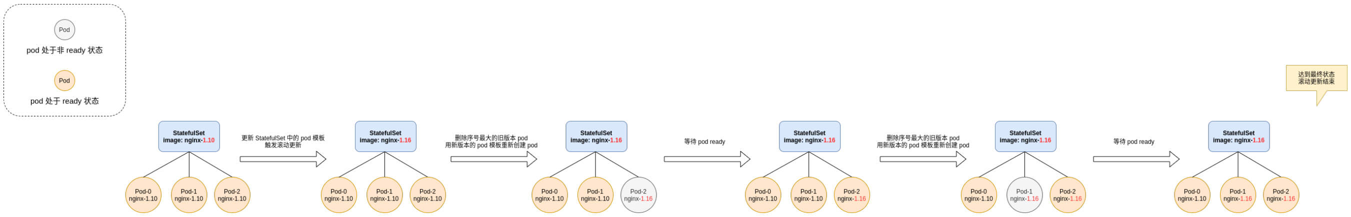
3.2.1 StatefulSet 滚动更新流程



3.2.2 partition

在 3.2 节中，我们看见 rollingUpdate 策略下有个 partition 参数。该参数作用为只更新序号大于等于 partition 的 pod。可以用于金丝雀

假设 partition = 1：



4 DaemonSet 升级策略

4.1OnDelete 策略

在该策略下，**用户需要手动删除**旧版本的 pod，控制器会创建新版本的 pod

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: busybox
spec:
  updateStrategy:
    type: OnDelete
……
```

4.2 RollingUpdate 策略

在该策略下进行更新，会逐个删除旧版本 pods，并创建新版本的 pods，保证服务一直可用。

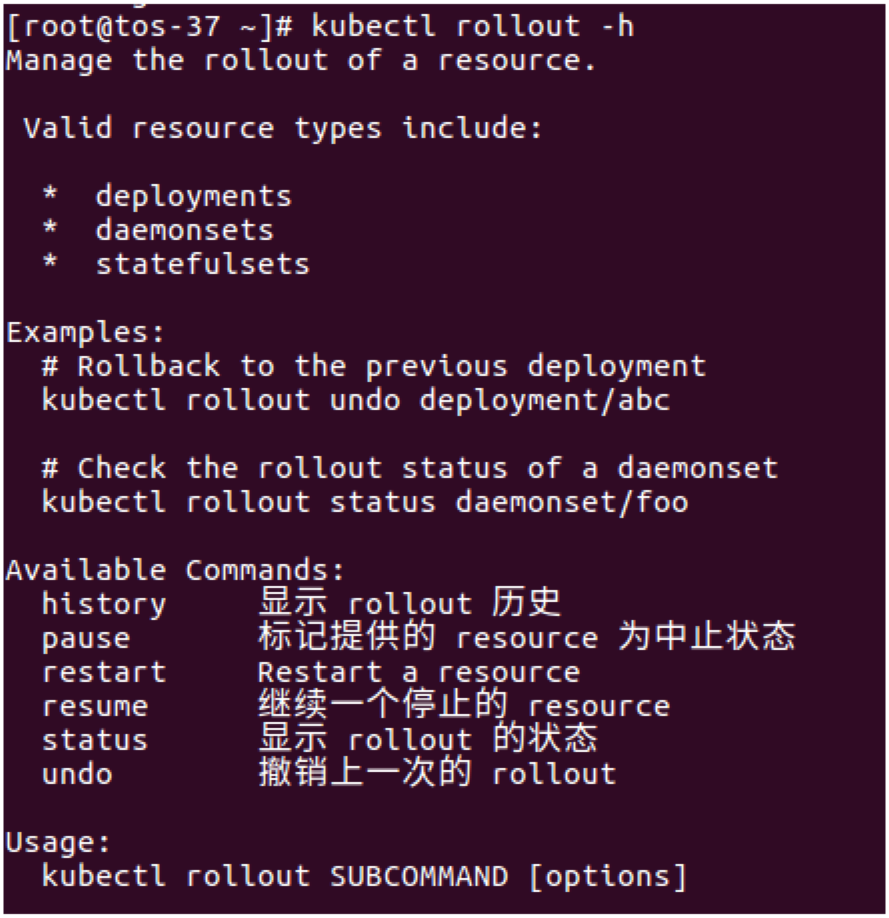
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: busybox
spec:
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
……
```

4.2.1 rolling-update 流程和maxUnavailable

daemonSet 的滚动更新流程与 deployment （maxUavailable = n，maxSurge = 0）时的更新流程一致，详情见 2.3.2 节

5 Kubectl rollout

K8S 针对滚动升级有 kubectl rollout 命令，主要作用为管理 deployment，statefulset 和 daemonset 的滚动升级



```
[root@tos-37 ~]# kubectl rollout -h
Manage the rollout of a resource.

 Valid resource types include:

  *  deployments
  *  daemonsets
  *  statefulsets

Examples:
  # Rollback to the previous deployment
  kubectl rollout undo deployment/abc

  # Check the rollout status of a daemonset
  kubectl rollout status daemonset/foo

Available Commands:
  history     显示 rollout 历史
  pause       标记提供的 resource 为中止状态
  restart     Restart a resource
  resume      继续一个停止的 resource
  status      显示 rollout 的状态
  undo        撤销上一次的 rollout

Usage:
  kubectl rollout SUBCOMMAND [options]
```

5.1 history

我们可以通过 history 子命令获取历史版本信息

```
#revison
[root@tos-37 ~]# kubectl rollout history deployment nginx-deployment

deployment.extensions/nginx-deployment
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=abcdocker-test.yaml --record=true
2         kubectl apply --filename=abcdocker-test.yaml --record=true
3         kubectl apply --filename=abcdocker-test.yaml --record=true
4         kubectl apply --filename=abcdocker-test.yaml --record=true

#nginx-deploymentdeploymentkubectl get deployment
```

5.2 undo

undo子命令可以配合history子命令把工作负载回滚到之前的版本

```
[root@tos-37 ~]# kubectl rollout undo deployment nginx-deployment --to-revision=1
deployment.extensions/nginx-deployment rolled back

# --to-revision=
```

5.3 pause

将提供的资源标记为暂停

目前仅支持的资源：deployments。

只要deployment在暂停中，使用deployment更新将不会生效。

```
[root@tos-37 ~]# kubectl rollout pause deployment demo-deployment
deployment.apps/demo-deployment paused
```

5.4 resume

恢复已暂停的资源

目前仅支持的资源：deployments。

```
[root@tos-37 ~]# kubectl rollout resume deployment demo-deployment
deployment.apps/demo-deployment resumed
```

5.5 status

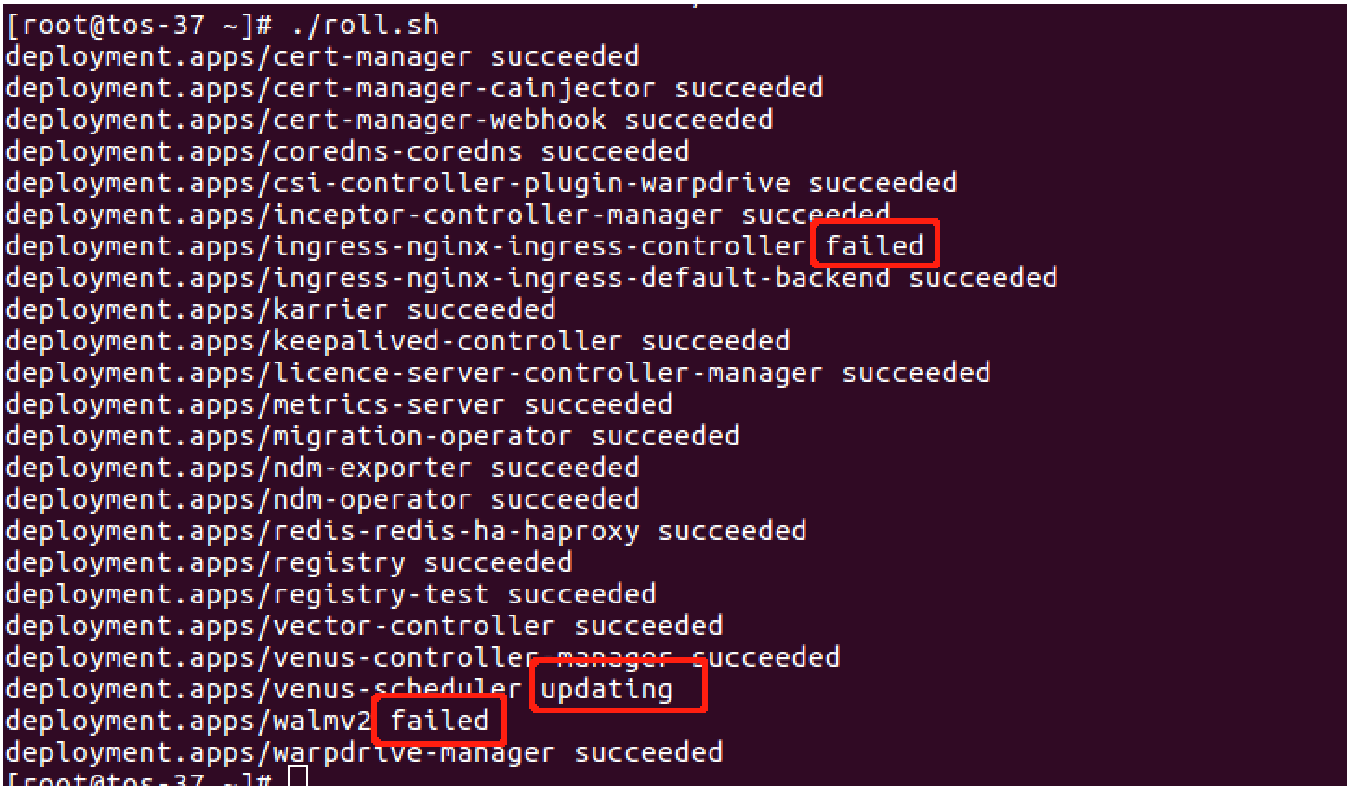
查看资源的状态

```
[root@tos-37 ~]# kubectl rollout status --watch=true  deployment demo-deployment
Waiting for deployment "demo-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "demo-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "demo-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "demo-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "demo-deployment" rollout to finish: 4 of 5 updated replicas are available...
deployment "demo-deployment" successfully rolled out

#  --watch=true
```

5.5.1 批量获取滚动更新状态

```
#!/bin/bash

# <your namespace> namespace
ns= <your namespace>
# <your workload> deploysts  ds  wsts
name=$(kubectl get <your workload>  -n$ns -o name)
for i in $name; do
  #
  kubectl rollout status $i --watch=false -n$ns  > /dev/null 2>&1
  if [ $? -ne 0 ]; then
    echo $i failed
    continue
  fi
  #
  kubectl rollout status $i --watch=true -n$ns --timeout=0.5s > /dev/null 2>&1
  if [ $? -ne 0 ]; then
    echo $i updating
  else
    echo $i succeeded
  fi
done
```

设置 <your namespace> 为 kube-system， <your workload> 为 deploy，获取结果如下：



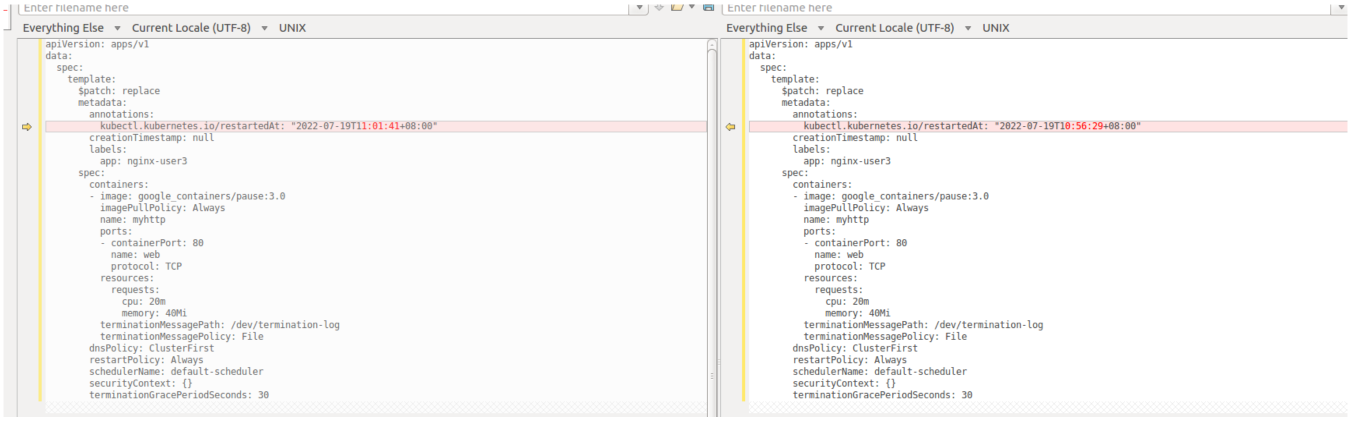
```
[root@tos-37 ~]# ./roll.sh
deployment.apps/cert-manager succeeded
deployment.apps/cert-manager-cainjector succeeded
deployment.apps/cert-manager-webhook succeeded
deployment.apps/coredns-coredns succeeded
deployment.apps/csi-controller-plugin-warpdrive succeeded
deployment.apps/inceptor-controller-manager succeeded
deployment.apps/ingress-nginx-ingress-controller failed
deployment.apps/ingress-nginx-ingress-default-backend succeeded
deployment.apps/karrier succeeded
deployment.apps/keepalived-controller succeeded
deployment.apps/licence-server-controller-manager succeeded
deployment.apps/metrics-server succeeded
deployment.apps/migration-operator succeeded
deployment.apps/ndm-exporter succeeded
deployment.apps/ndm-operator succeeded
deployment.apps/redis-redis-ha-haproxy succeeded
deployment.apps/registry succeeded
deployment.apps/registry-test succeeded
deployment.apps/vector-controller succeeded
deployment.apps/venus-controller-manager succeeded
deployment.apps/venus-scheduler updating
deployment.apps/walmv2 failed
deployment.apps/warpdrive-manager succeeded
[root@tos-37 ~]#
```

在实际使用中，除了关注滚动升级 failed 的 workload，还应该多关注长时间处于 updating 的 workload，防止出现 2.3.5 节中的情况

5.6 restart

restart 子命令将滚动重启指定的 workload 下的所有 pod。

实现方式：修改 workload 的 template.metadata.annotation，使得 workload 触发滚动更新



```
apiVersion: apps/v1
data:
  spec:
    template:
      $patch: replace
      metadata:
        annotations:
          kubectl.kubernetes.io/restartedAt: "2022-07-19T11:01:41+08:00"
        creationTimestamp: null
        labels:
          app: nginx-user3
      spec:
        containers:
        - image: google_containers/pause:3.0
          imagePullPolicy: Always
          name: myhttp
          ports:
          - containerPort: 80
            name: web
            protocol: TCP
          resources:
            requests:
              cpu: 20m
              memory: 40Mi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
        dnsPolicy: ClusterFirst
        restartPolicy: Always
        schedulerName: default-scheduler
        securityContext: {}
        terminationGracePeriodSeconds: 30
```

```
apiVersion: apps/v1
data:
  spec:
    template:
      $patch: replace
      metadata:
        annotations:
          kubectl.kubernetes.io/restartedAt: "2022-07-19T10:56:29+08:00"
        creationTimestamp: null
        labels:
          app: nginx-user3
      spec:
        containers:
        - image: google_containers/pause:3.0
          imagePullPolicy: Always
          name: myhttp
          ports:
          - containerPort: 80
            name: web
            protocol: TCP
          resources:
            requests:
              cpu: 20m
              memory: 40Mi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
        dnsPolicy: ClusterFirst
        restartPolicy: Always
        schedulerName: default-scheduler
        securityContext: {}
        terminationGracePeriodSeconds: 30
```

6 滚动升级流程源码分析

Statefulset 和 DaemonSet 的滚动更新逻辑都非常简单，在这里我们主要分析 deployment 的滚动更新逻辑。

6.1 syncDeployment()

```
// kubernetes/pkg/controller/deployment/deployment_controller.go
// deployment  syncDeployment()
func (dc *DeploymentController) syncDeployment(key string) error {
    ......
    switch d.Spec.Strategy.Type {
    case apps.RecreateDeploymentStrategyType:
        return dc.rolloutRecreate(d, rsList, podMap)
    case apps.RollingUpdateDeploymentStrategyType:
        //  rolloutRolling
        return dc.rolloutRolling(d, rsList)
    }
    ......
}
```

6.2 rolloutRolling()

滚动更新的逻辑位于rolloutRolling() 中

```
func (dc *DeploymentController) rolloutRolling(......) error {
    // 1 rs rs
    newRS, oldRSs, err := dc.getAllReplicaSetsAndSyncRevision(d, rsList, true)
    if err != nil {
        return err
    }
    allRSs := append(oldRSs, newRS)
    // 2 scale up  pod
    scaledUp, err := dc.reconcileNewReplicaSet(allRSs, newRS, d)
    if err != nil {
        return err
    }
    if scaledUp {
        return dc.syncRolloutStatus(allRSs, newRS, d)
    }
    // 3 scale down  pod
    scaledDown, err := dc.reconcileOldReplicaSets(allRSs, controller.FilterActiveReplicaSets(oldRSs), newRS, d)
    if err != nil {
        return err
    }
    if scaledDown {
        return dc.syncRolloutStatus(allRSs, newRS, d)
    }
    // 4 rs
    if deploymentutil.DeploymentComplete(d, &d.Status) {
        if err := dc.cleanupDeployment(oldRSs, d); err != nil {
            return err
        }
    }
    // 5 deployment status
    return dc.syncRolloutStatus(allRSs, newRS, d)
}
```

6.3 scale up

在滚动更新时，最重要的就是 scale up 新版本 pod 和 scale down 旧版本 pod。而 scale up 新版本 pod 的逻辑就在reconcileNewReplicaSet() 中。

reconcileNewReplicaSet主要逻辑如下：

- 1、判断newRS.Spec.Replicas和deployment.Spec.Replicas是否相等，如果相等则直接返回，说明已经达到期望状态；
- 2、若newRS.Spec.Replicas>deployment.Spec.Replicas，则说明 newRS 副本数已经超过期望值，调用dc.scaleReplicaSetAndRecordEvent进行 scale down；
- 3、此时newRS.Spec.Replicas<deployment.Spec.Replicas，调用NewRSNewReplicas为 newRS 计算所需要的副本数，计算原则遵守maxSurge和maxUnavailable的约束；
- 4、调用scaleReplicaSetAndRecordEvent更新 newRS 对象，设置 rs.Spec.Replicas、rs.Annotations[DesiredReplicasAnnotation] 以及 rs.Annotations[MaxReplicasAnnotation] ；

```
func (dc *DeploymentController) reconcileNewReplicaSet(......) (bool, error) {
    // 1
    if *(newRS.Spec.Replicas) == *(deployment.Spec.Replicas) {
        return false, nil
    }
    // 2 scale down
    if *(newRS.Spec.Replicas) > *(deployment.Spec.Replicas) {
        scaled, _, err := dc.scaleReplicaSetAndRecordEvent(newRS, *(deployment.Spec.Replicas), deployment)
        return scaled, err
    }
    // 3 newRS
    newReplicasCount, err := deploymentutil.NewRSNewReplicas(deployment, allRSs, newRS)
    if err != nil {
        return false, err
    }
    // 4 scale  rs  annotation  rs.Spec.Replicas
    scaled, _, err := dc.scaleReplicaSetAndRecordEvent(newRS, newReplicasCount, deployment)
    return scaled, err
}
```

NewRSNewReplicas是为 newRS 计算所需要的副本数，该方法主要逻辑为：

- 1、判断更新策略；
- 2、计算 maxSurge 值；
- 3、通过 allRSs 计算 currentPodCount 的值；
- 4、最后计算 scaleUpCount 值；

```
func NewRSNewReplicas(......) (int32, error) {
    switch deployment.Spec.Strategy.Type {
    case apps.RollingUpdateDeploymentStrategyType:
        // 1 maxSurge
        maxSurge, err := intstrutil.GetValueFromIntOrPercent(deployment.Spec.Strategy.RollingUpdate.MaxSurge,
int(*(deployment.Spec.Replicas)), true)
        if err != nil {
            return 0, err
        }
        // 2 rs.Spec.Replicas  currentPodCount
        currentPodCount := GetReplicaCountForReplicaSets(allRSs)
        maxTotalPods := *(deployment.Spec.Replicas) + int32(maxSurge)
        if currentPodCount >= maxTotalPods {
            return *(newRS.Spec.Replicas), nil
        }
        // 3 scaleUpCount
        scaleUpCount := maxTotalPods - currentPodCount
        scaleUpCount = int32(integer.IntMin(int(scaleUpCount), int(*(deployment.Spec.Replicas)-*(newRS.Spec.
Replicas))))
        return *(newRS.Spec.Replicas) + scaleUpCount, nil
    case apps.RecreateDeploymentStrategyType:
        return *(deployment.Spec.Replicas), nil
    default:
        return 0, fmt.Errorf("deployment type %v isn't supported", deployment.Spec.Strategy.Type)
    }
}
```

6.4 scale down

在滚动更新时，最重要的就是 scale up 新版本 pod 和 scale down 旧版本 pod。而 scale down 旧版本 pod 的逻辑就在 reconcileOldReplicaSets() 中。

reconcileOldReplicaSets的主要逻辑如下：

- 1、通过 oldRSs 和 allRSs 获取 oldPodsCount 和 allPodsCount；
- 2、计算 deployment 的 maxUnavailable、minAvailable、newRSUnavailablePodCount、maxScaledDown 值，当 deployment 的 maxSurge 和 maxUnavailable 值为百分数时，计算 maxSurge 向上取整而 maxUnavailable 则向下取整；
- 3、清理异常的 rs；
- 4、计算 oldRS 的 scaleDownCount；

```
func (dc *DeploymentController) reconcileOldReplicaSets(......)   (bool, error) {
    // 1 oldPodsCount
    oldPodsCount := deploymentutil.GetReplicaCountForReplicaSets(oldRSs)
    if oldPodsCount == 0 {
        return false, nil
    }
    // 2 allPodsCount
    allPodsCount := deploymentutil.GetReplicaCountForReplicaSets(allRSs)
    // 3 maxScaledDown
    maxUnavailable := deploymentutil.MaxUnavailable(*deployment)
    minAvailable := *(deployment.Spec.Replicas) - maxUnavailable
    newRSUnavailablePodCount := *(newRS.Spec.Replicas) - newRS.Status.AvailableReplicas
    maxScaledDown := allPodsCount - minAvailable - newRSUnavailablePodCount
    if maxScaledDown <= 0 {
        return false, nil
    }
    // 4 rs
    oldRSs, cleanupCount, err := dc.cleanupUnhealthyReplicas(oldRSs, deployment, maxScaledDown)
    if err != nil {
        return false, nil
    }
    allRSs = append(oldRSs, newRS)
    // 5 old rs
    scaledDownCount, err := dc.scaleDownOldReplicaSetsForRollingUpdate(allRSs, oldRSs, deployment)
    if err != nil {
        return false, nil
    }
    totalScaledDown := cleanupCount + scaledDownCount
    return totalScaledDown > 0, nil
}
```

通过上面的代码可以看出，滚动更新过程中主要是通过调用reconcileNewReplicaSet对 newRS 不断扩容，调用reconcileOldReplicaSets对 oldRS 不断缩容，最终达到期望状态，并且在整个升级过程中，都严格遵守maxSurge和maxUnavailable的约束。

7 总结

从以上的文档中，我们可以看见不同参数下的滚动升级的不同表现，比如优先删除旧版本，优先创建新版本，升级的速率快慢等等。

具体参数设置还需要结合具体场景。

8 参考

第9章 基于云原生技术的 CICD 设计与开发-Grissom-202010.pdf

https://kubernetes.io/zh-cn/docs/concepts/workloads/controllers/deployment/

http://docs.kubernetes.org.cn/643.html

https://jamesdefabia.github.io/docs/user-guide/kubectl/kubectl_rolling-update/

https://cloud.tencent.com/developer/article/1650197

https://zhuanlan.zhihu.com/p/90282741

https://draveness.me/kubernetes-deployment/

https://mp.weixin.qq.com/s?__biz=Mzg5NDYxNTYyMw==&mid=2247487585&idx=1&sn=bcc0b669d0680a6ca6269f66529dae1e&source=41#wechat_redirect

https://www.cnblogs.com/wtzbk/p/15335825.html

https://www.bookstack.cn/read/source-code-reading-notes/kubernetes-deployment_controller.md