核心知识

控制基础

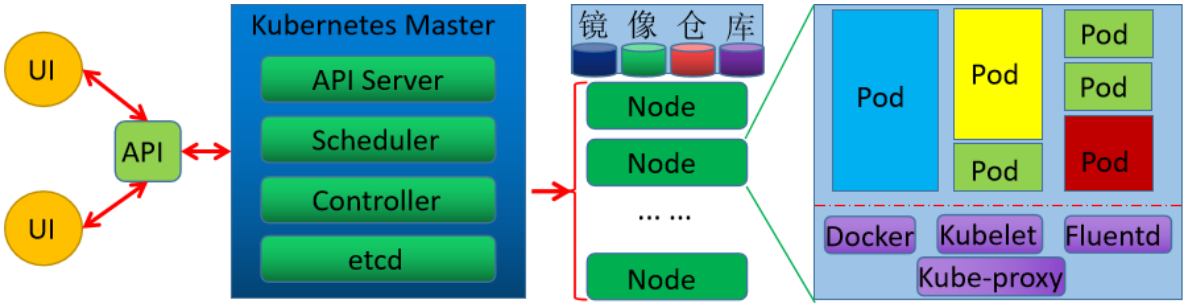
集群回顾

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

集群基本结构



k8s集群通过 master角色主机 和 node角色主机 实现集群的环境搭建，所有的资源对象都是以pod为单元进行管理的。pod内部都是一个个相互关联的 容器对象。这些容器对象是来源于镜像仓库的镜像文件创建出来的。

也就是说，k8s主要有 控制层面和数据层面来组成：

控制层面

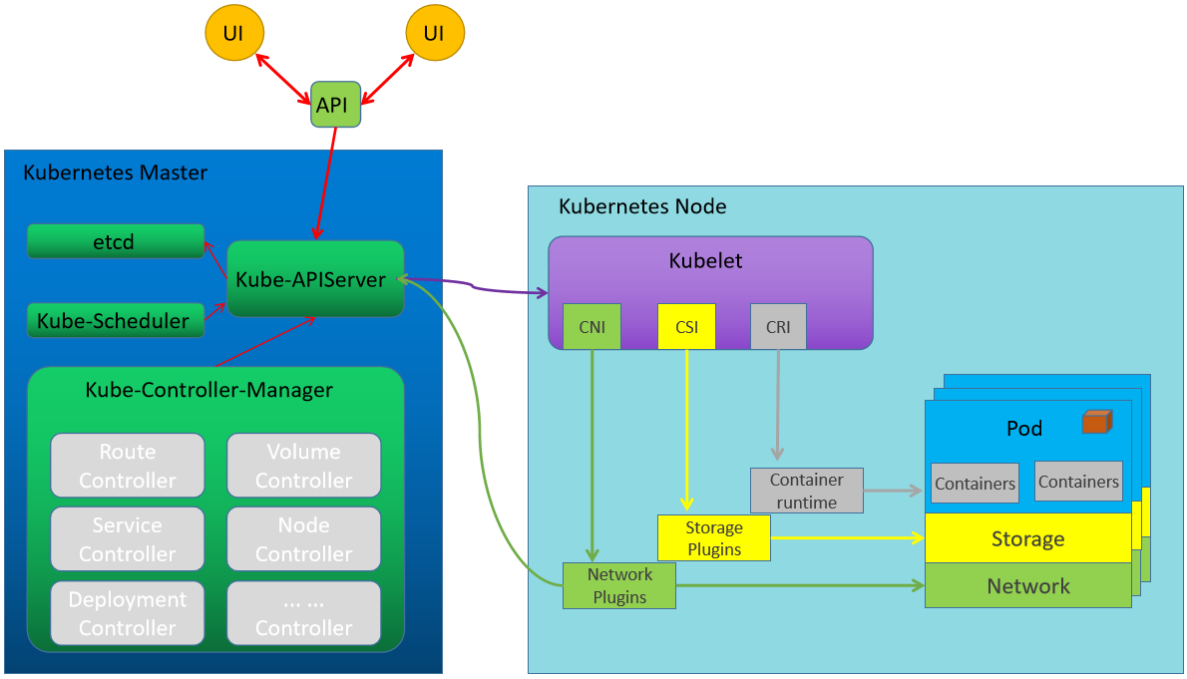
API Server - 提供数据的注册和监视各种资源对象的功能

Scheduler - 将资源对象调度到合适的节点中。

Controller - 大量控制功能的集合,他是与API Server结合起来完成控制层面操作的最重要的组成部分。

数据层面

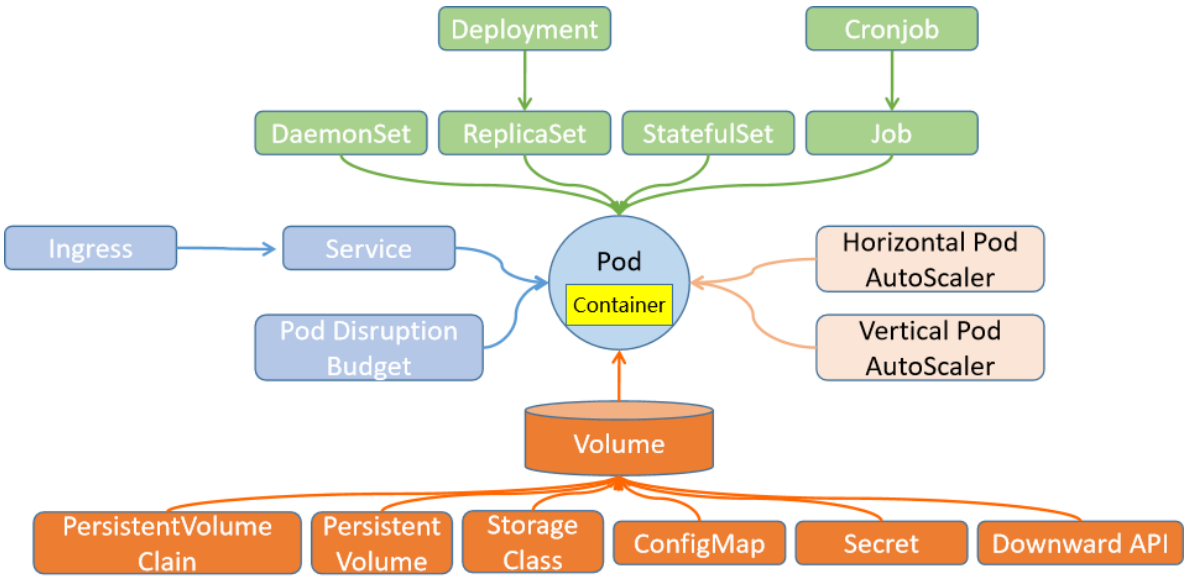
组件基本流程图



k8s集群中的所有资源对象都是

- 通过 master角色主机上的各种组件进行统一管理，
- 基于node角色上的kubelet组件实现信息的交流。
- pod内部的应用对象
  - 基于CRI让应用本身是运行在容器内部。
  - 基于CSI实现各种持久化数据的保存。
  - 基于CNI实现多个pod应用程序之间的通信交流。

资源对象



对于k8s集群应用来说，所有的程序应用都是

- 运行在 Pod资源对象里面，
- 借助于service资源对象向外提供服务访问。
- 借助于各种存储资源对象实现数据的可持久化
- 借助于各种配置资源对象实现配置属性、敏感信息的管理操作

应用编排

简介

我们在工作中为了完成大量的业务目标，首先会根据业务应用内部的关联关系，把业务拆分成多个子任务，然后对这些子任务进行顺序组合，当子任务按照方案执行完毕后，就完成了业务目标。任务编排，就是对多个子任务执行顺序进行确定的过程。

对于k8s来说，
- 对于紧密相关的多个子任务，我们把它们放到同一个pod内部，
- 对于非紧密关联的多个任务，分别放到不同的pod中，
- 然后借助于 endpoint+service的方式实现彼此之间的 相互调用。
- 为了让这些纷乱繁杂的任务能够互相发现自己，我们通过集群的 CornDNS组件实现服务注册发现功能。

k8s编排

对于k8s场景中的应用任务来说，这些任务彼此之间主要存在 部署、扩容、缩容、更新、回滚等。虽然我基于pod的方式实现了应用任务的部署功能操作，但是对于我们自主式的pod来说，它并不能实现其他的几种任务。

而这显然 距 k8s的核心功能 是有一定的距离的，因此，在k8s集群的核心功能之上，有一群非常重要的组件，这些组件就是用于对pod实现所谓的任务编排功能，这些组件我们统统将其称为 -- 控制器。

控制器

对于控制器来说，他有很多中种类:
节点控制器(Node Controller): 负责在节点出现故障时进行通知和响应
任务控制器(Job controller): 监测代表一次性任务的 Job 对象，然后创建 Pods 来运行这些任务直至完成
端点控制器(Endpoints Controller): 填充端点(Endpoints)对象(即加入 Service 与 Pod)
服务帐户和令牌控制器(Service Account & Token Controllers): 为新的命名空间创建默认帐户和 API 访问令牌

小结

控制原理

学习目标

这一节，我们从 原理解析、流程梳理、小结 三个方面来学习。

原理解析

目标

上一节，我们已经说过了，Kube-controller-manager 是与API server 结合起来实现控制层面核心功能中最重要的功能。那么我们所谓的控制循环是如何工作的呢？

master节点组件解析

根据我们之前对各种应用程序对象的操作，比如pod、service、token、configmap等等。我们知道，这些应用程序主要有两种在形式：

数据形态(应用程序的条目) - 存储在API Server中的各种数据条目

实体形态(真正运行的对象) - 通过kube-controller-manager到各节点上创建的具体对象。

API Server 从某种层面上来说，它仅仅是一个DB，只不过

- 它支持对资源数据条目的 watch/notify 的功能，
- 它对于数据存储的格式进行了限制定制，也就是说，只能接收固定格式的数据规范。

如果我们仅仅将相关对象的属性信息插入到 APIServer上，就相当于我们做企业规划的时候，准备为某个项目配置多少资源，而这个时候，这些资源是没有到位的，无法进行项目运行的。而kube-controller-manager就相当于企业的创始人，根据APIServer上的规划，通过招人、拉投资等方式，将一个个的具体对象落地。

示例：

比如每一个Service，就是一个Service对象的定义，同时它还代表了每个节点上iptables或ipvs规则，而这些节点上的规则，是由kube-controller-manager结合 kube-proxy来负责进行落地。

pod的调度

每一个Pod，就是一个应用程序的定义，同时它还代表了每个节点上资源的限制，而这些节点上的应用程序，是由kube-controller-manager 结合 kubelet来负责进行落地。

资源在创建的时候，一般会由Scheduler调度到合适的Node节点上。这个时候，kubeServer在各个节点上的客户端kubelet组件，如果发现调度的节点是本地，那么会在本地节点将对应的资源进行落地，同时负责各个节点上的与APIServer相关的资源对象的监视功能。

注意，kubelet只负责单个节点上的Pod管理和调度。一旦我们需要对pod进行扩容缩容，涉及到了跨节点的资源调度，kubelet是做不到的。对于这些更高层级的应用资源调度，我们只能通过构建在 k8s 核心基础资源对象之上的控制资源来实现。

流程梳理

检查实践

定义资源配置文件

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-test
spec:
  containers:
  - name: nginx
    image: 10.0.0.19:80/mykubernetes/nginx:1.21.3
    env:
    - name: HELLO
      value: "Hello kubernetes nginx"
```

应用资源定义文件

```
kubectl apply -f 01-pod-nginx.yaml
```

查看效果

```
kubectl get pod -o yaml
```

```
root@master1:~# kubectl get pod nginx-test -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"nginx-test","namespace":"default"},"spec":{"co
e":"HELLO","value":"Hello kubernetes nginx"}],"image":"10.0.0.19:80/mykubernetes/nginx:1.21.3","name":"nginx"}]}}
  creationTimestamp: "2021-09-30T03:54:45Z"
  name: nginx-test
  namespace: default
  resourceVersion: "760757"
  uid: 13463e83-785a-4316-8751-4756a83ac7c1
spec:
  containers:
  - env:
    - name: HELLO
      value: Hello kubernetes nginx
    image: 10.0.0.19:80/mykubernetes/nginx:1.21.3
    imagePullPolicy: IfNotPresent
    name: nginx
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-ntwg9
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2021-09-30T03:54:45Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2021-09-30T03:54:51Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2021-09-30T03:54:51Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2021-09-30T03:54:45Z"
    status: "True"
    type: PodScheduled
```

流程梳理

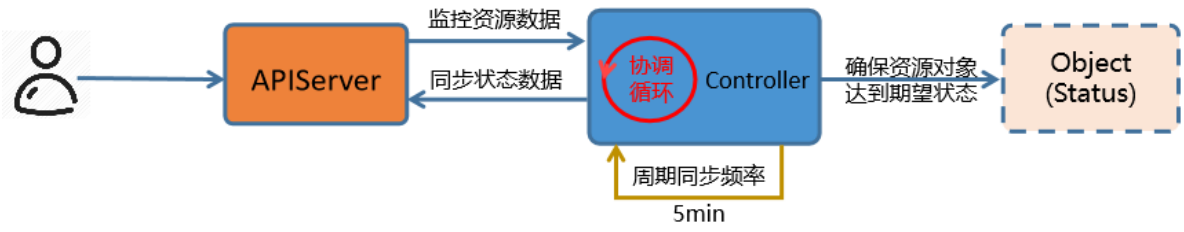
1 用户向 APIserver中插入一个应用资源的数据形态
   - 这个数据形态中定义了该资源对象的 "期望"状态，
   - 数据经由 APIserver 保存到 ETCD 中。

2 kube-controller-manager 中的各种控制器会监视 Apiserver上与自己相关的资源对象的变动
   比如 Service Controller只负责Service资源的控制，Pod Controller只负责Pod资源的控制等。

3 一旦APIServer中的资源对象发生变动，对应的Controller执行相关的配置代码，到对应的node节点上运行
   - 该资源对象会在当前节点上，按照用户的"期望"进行运行
   - 这些实体对象的运行状态我们称为 "实际状态"
   - 即，控制器的作用就是确保 "期望状态" 与 "实际状态" 相一致

4 Controller将这些实际的资源对象状态，通过APIServer存储到ETCD的同一个数据条目的status的字段中

5 资源对象在运行过程中，Controller 会循环的方式向 APIServer 监控 spec 和 status 的值是否一致
   - 如果两个状态不一致，那么就指挥node节点的资源进行修改，保证 两个状态一致
   - 状态一致后，通过APIServer同步更新当前资源对象在ETCD上的数据



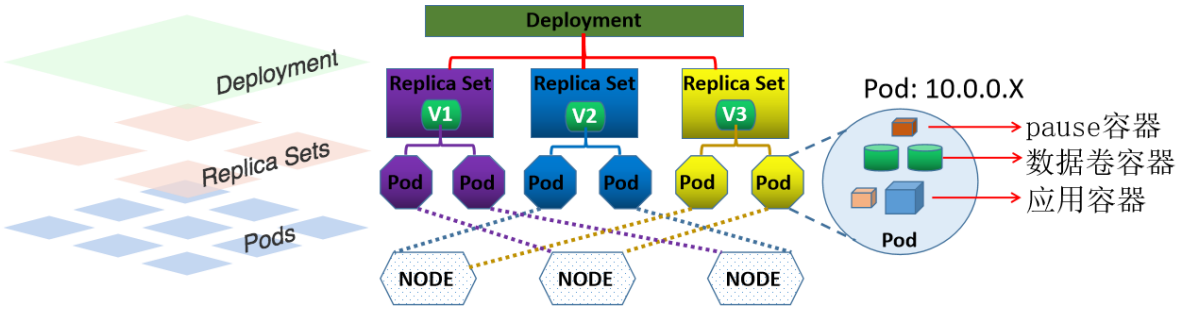
控制器分类

控制器	解析
ReplicationController	最早期的Pod控制器，目前已被废弃。
RelicaSet	副本集，负责管理一个应用(Pod)的多个副本状态
Deployment	它不直接管理Pod，而是借助于ReplicaSet来管理Pod；最常用的无状态应用控制器；
DaemonSet	守护进程集，用于确保在每个节点仅运行某个应用的一个Pod副本。用于完成系统级任务。
StatefulSet	功能类似于Deployment，但StatefulSet专用于编排有状态应用
Job	有终止期限的一次性作业式任务，而非一直处于运行状态的服务进程；
CronJob	有终止期限的周期性作业式任务

注意：
早期的大多数控制器都位于`extensions/v1beta1, v1beta2, ...`，而这些版本都已经被新的替换了
我们主要来说 `apps/v1` 版本下的控制器
每个控制器对象 也需要 对应的`controller`来进行管理

控制器 vs pod

控制器主要是通过管理pod来实现任务的编排效果，那么控制器是通过什么机制找到pod的呢？
- 标签 或者 标签选择器



小结

控制资源

标签

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

Kubernetes通过标签来管理对应或者相关联的各种资源对象，Lable是kubernetes中的核心概念之一。

特点

1、创建：
    Lable通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除
2、作用：
    一个资源对象可以定义多个Lable，同一个Lable也可以关联多个资源对象上去
3、本质：
    lable本质上是一个key/value键值对，其中key与value由用户自己指定，Lable可以附加到各种资源对象上(比如我们学过的Node和Pod等)。
注意：
    key的命名：由"字母、数字、_、.、-"这五类组成,只能以字符或数字作为开头和结尾。
    标签名称不能多于63个字符

应用目的

Label关联到各种资源对象上，通过对Lable的管理从而达到对同Lable的资源进行分组管理(分配、调度、配置、部署等)。
常用标签使用场景：
- 版本标签："release" : "stable"，"release" : "canary"，"release" : "beta"。
- 环境标签："environment" : "dev"，"environment" : "qa"，"environment" : "prod"。
- 应用标签："app" : "ui"，"app" : "as"，"app" : "pc"，"app" : "sc"。
- 架构层级标签："tier" : "frontend"，"tier" : "backend", "tier" : "cache"。
- 分区标签："partition" : "customerA"，"partition" : "customerB"。
- 品控级别标签："track" : "daily"，"track" : "weekly"。

操作方法

关于label的操作主要有两种：yaml方法、命令行方法

yaml方法：
    在特定的属性后面按照指定方式增加内容即可，格式如下：

```
labels:
  key: value
```

    注意：
        labels 是 一个复数格式。

命令行方法：
    查看标签：kubectl get pods -l label_name=label_value
        参数：
            -l 就是指定标签条件，获取指定资源对象，=表示匹配，!= 表示不匹配
            如果后面的选择标签有多个的话，使用逗号隔开
            如果针对标签的值进行范围过滤的话，可以使用如下格式：
            -l "label_name in|notin (value1, value2, value3, ...)"
    增加标签：kubectl label 资源类型 资源名称 label_name=label_value
        参数：
            同时增加多个标签，只需要在后面多写几个就可以了，使用空格隔开
            默认情况下，已存在的标签是不能修改的，使用 --overwrite=true 表示强制覆盖
            label_name=label_value样式写成 label_name- ，表示删除label

简单实践

资源文件方式

资源对象Lablel不是单独定义的，是需要依附在某些资源对象上才可以，常见的就是依附在Pod对象上。
Label在之前的Pod的资源定义文件pod-test.yaml中的实践样式如下：
初始化Pod资源对象应用时候，在资源对象的元数据metadata属性下添加一条Labels配置：

```
标签的内容是：app: nginx
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
  labels:
    app: nginx
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1

添加label之前的效果
]# kubectl describe pod pod-test
Name:               pod-test
...
Labels:             <none>
...
添加label后，再次创建一个Pod
kubectl create -f 02-pod-label.yaml
查看pod的基本信息
]# kubectl get pod pod-test --show-labels
NAME         READY   STATUS    RESTARTS   AGE   LABELS
pod-test   1/1     Running   0          13m   app=nginx

列出指定标签的pod
]# kubectl get pods -l app=nginx
NAME         READY     STATUS    RESTARTS   AGE
pod-test   1/1       Running   0          20s
```

命令行方式

```
做一个新的pod
kubectl run nginx-test --image=10.0.0.19:80/mykubernetes/nginx:1.21.3 --
labels="app=nginx-test,env=prod"

给pod打标签
]# kubectl label pod nginx-test release=1 pro=dev
pod/nginx-test labeled
]# kubectl get pod nginx-test  --show-labels
NAME         READY   STATUS    RESTARTS   AGE   LABELS
nginx-test   1/1     Running   0          14m   app=nginx-
test,env=prod,pro=dev,release=1

强制覆盖
]# kubectl label pod nginx-test release=2
```

```
error: 'release' already has a value (1), and --overwrite is false
]# kubectl label pod nginx-test release=2 --overwrite=true
pod/nginx-test labeled
]# kubectl get pod nginx-test  --show-labels
NAME         READY   STATUS    RESTARTS   AGE   LABELS
nginx-test   1/1     Running   0          16m   app=nginx-
test,env=prod,pro=dev,release=2
```

小结

标签选择器

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

```
    Lablel附加到Kubernetes集群中的各种资源对象上，目的就是对这些资源对象进行分组管理，而分组管理的核心就是：Lablel Selector。

分组管理的原理：
    我们可以通过Label Selector(标签选择器)查询和筛选某些特定Label的资源对象，进而可以对他们进行相应的操作管理，类似于我们的sql语句中where的条件：
    select * from where ...

Lablel Selector跟Label一样，不能单独定义，必须附加在一些资源对象的定义文件上。一般附加在RC和Service的资源定义文件中。
```

表达式

```
Label Selector使用时候有两种常见的表达式：等式和集合

等式：
    name = nginx                    匹配所有具有标签 name = nginx 的资源对象
    name != nginx                   匹配所有不具有标签 name = nginx 的资源对象
集合：
    env in (dev, test)              匹配所有具有标签 env = dev 或者 env = test 的资源对象
    name not in (frontend)          匹配所有不具有标签 name = frontend 的资源对象
```

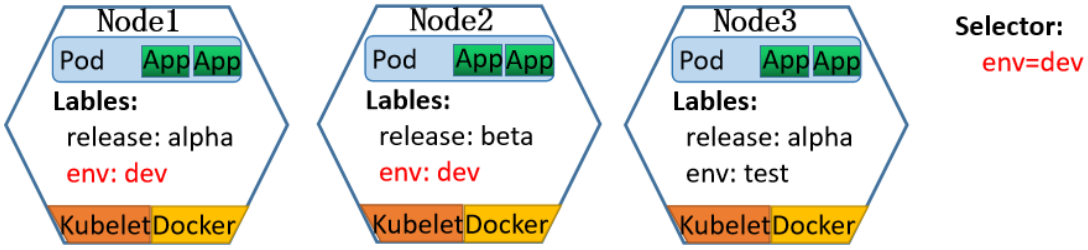
随着Kubernetes功能的不断完善，集合表达式逐渐有了两种规范写法：匹配标签、匹配表达式

```
匹配标签：
    matchLabels:
      name: nginx
匹配表达式：
    matchExpressions:
      - {key: name, operator: NotIn, values: [frontend]}

常见的operator操作属性值有：
    In、NotIn、Exists、NotExists等
    Exists和DostNotExist时，values必须为空，即
    { key: environment, opetator: Exists,values:}
注意：
    这些表达式，一般应用在RS、RC、Deployment等其它管理对象中。
```

应用场景

```
标签选择器的主要应用场景：
    监控具体的Pod、负载均衡调度、定向调度，常用于 Pod、Node等资源对象
```




```
当设置 env=dev的Label Selector,则会匹配到Node1和Node2上的Pod
```

简单实践

- 命令方式

命令行简单实践

```
等值过滤
# kubectl get pods -l env=prod
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          8m33s
# kubectl get pods -l env==prod
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          8m43s

# kubectl get pods -l app!=nginx
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          9m15s

# kubectl get pods -l app
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          9m38s
# kubectl get pods -l '!app'        注意：为了避免shell解析，这里需要添加单引号
No resources found in default namespace.
```

集合过滤

```
# kubectl get pods -l "app in (nginx-test, hah, heh)"       多条件取交集
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          10m

# kubectl get pods -l "app notin (nginx, hah, heh)"
NAME          READY   STATUS    RESTARTS   AGE
nginx-test2   1/1     Running   0          11m
```

删除标签

```
]# kubectl label pod nginx-test pro- release-
pod/nginx-test labeled
]# kubectl get pod nginx-test  --show-labels
NAME         READY   STATUS    RESTARTS   AGE   LABELS
nginx-test   1/1     Running   0          17m   app=nginx
```

- 配置文件

我们这里以service资源对象的创建为例。

准备多个后端pod对象

```
kubectl run nginx-test1 --image=10.0.0.19:80/mykubernetes/pod_test:v0.1 --
labels="app=nginx"
kubectl run nginx-test2 --image=10.0.0.19:80/mykubernetes/pod_test:v0.1 --
labels="app=nginx"
kubectl run nginx-test3 --image=10.0.0.19:80/mykubernetes/pod_test:v0.1 --
labels="app=nginx1"
```

检查效果

```
# kubectl get pod --show-labels -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP            NODE    ...
LABELS
nginx-test1   1/1     Running   0          12s   10.244.6.71   node2   ...
app=nginx
nginx-test2   1/1     Running   0          11s   10.244.6.72   node2   ...
app=nginx
nginx-test3   1/1     Running   0          10s   10.244.5.50   node1   ...
app=nginx1
```

创建service对象

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  selector:
    app: nginx
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 80
```

创建service对象

```
kubectl apply -f 03-pod-service.yml
```

检查效果

```
# kubectl get svc -o wide
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
SELECTOR
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP   7d21h
<none>
service-test   ClusterIP   10.107.78.208   <none>        80/TCP    10s
app=nginx
# kubectl describe svc service-test
Name:              service-test
...
TargetPort:        80/TCP
Endpoints:         10.244.6.71:80,10.244.6.72:80
...
```

结果显式：

service 自动关联了两个Endpoints

随机访问效果：

```
# curl 10.107.78.208
kubernetes pod-test !! ClientIP: 10.244.0.0, ServerName: nginx-test2, ServerIP:
10.244.6.76!
# curl 10.107.78.208
kubernetes pod-test !! ClientIP: 10.244.0.0, ServerName: nginx-test1, ServerIP:
10.244.6.75!
# curl 10.107.78.208
kubernetes pod-test !! ClientIP: 10.244.0.0, ServerName: nginx-test1, ServerIP:
10.244.6.75!
# curl 10.107.78.208
kubernetes pod-test !! ClientIP: 10.244.0.0, ServerName: nginx-test1, ServerIP:
10.244.6.75!
# curl 10.107.78.208
kubernetes pod-test !! ClientIP: 10.244.0.0, ServerName: nginx-test2, ServerIP:
10.244.6.76!
```

结果显式：

后端随机代理到不同的pod应用了

RC & RS

学习目标

这一节，我们从 基础知识、RS基础、小结 三个方面来学习。

RC基础

RC

```
Replication Controller（RC），是kubernetes系统中的核心概念之一。RC是Kubernetes集群实现
Pod资源对象自动化管理的基础。

简单来说，RC其实是定义了一个期望的场景，RC有以下特点：
1、组成：定义了Pod副本的期望状态：包括数量，筛选标签和模板
    Pod期待的副本数量(replicas).
    永远筛选目标Pod的标签选择器(Label Selector)
    Pod数量不满足预期值，自动创建Pod时候用到的模板(template)
2、意义：自动监控Pod运行的副本数目符合预期，保证Pod高可用的核心组件，常用于Pod的生命周期管理

拓展：自动监控的功能是基于哪些资源对象？
```

文件示例

RC资源对象定义文件，遵循资源对象定义文件的格式，只不过spec期望的部分是RC的主要内容
编辑一个04-controller-rc.yaml文件，由RC自动控制Pod资源对象的预期状态效果：
运行2个nginx容器，运行的容器携带两个标签，运行容器的模板文件在sepc.template.spec部分

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: 10.0.0.19:80/mykubernetes/nginx:1.21.3
```

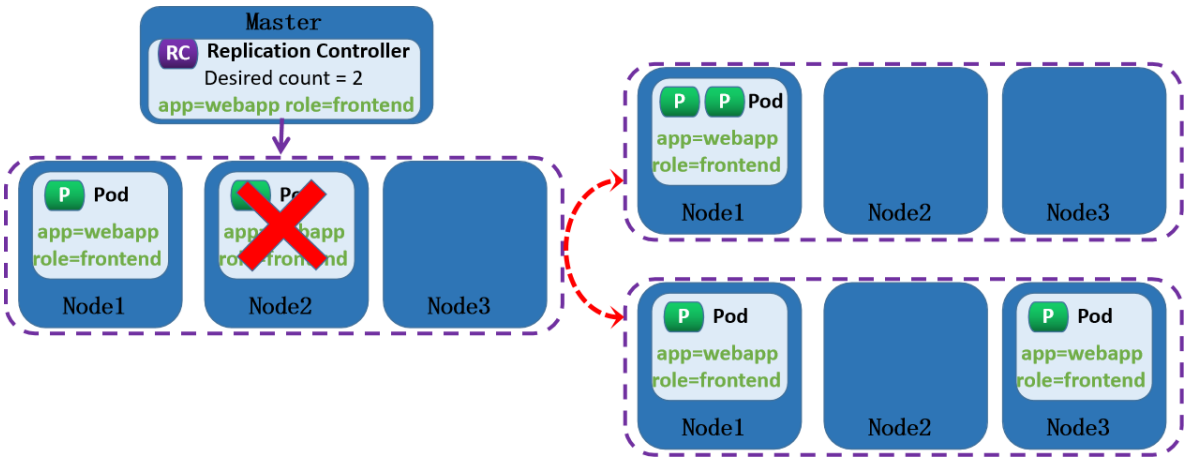
关键点：
RC的期望状态三结构：
replicas-副本数量　　selector-标签选择器　　template-容器模板文件
spec.template.metadata.labels 的属性必须跟Pod资源对象的metadata.labels属性一致

RC 作用

当我们通过"资源定义文件"定义好了一个RC资源对象，把它提交到Kubernetes集群中以后，Master节点上的Controller Manager组件就得到通知(问：为什么？因为什么？)，定期巡检系统中当前存活的Pod，并确保Pod实例数量刚到满足RC的期望值。
如果Pod数量大于RC定义的期望值，那么就杀死一些Pod
如果Pod数量小于RC定义的期望值，那么就创建一些Pod
所以：通过RC资源对象，Kubernetes实现了业务应用集群的高可用性，大大减少了人工干预，提高了管理的自动化。
拓展：
想要扩充Pod副本的数量，可以直接修改replicas的值即可

实现机制

当其中一个Node的Pod意外终止，根据RC的定义，Pod的期望值是2，所以会随机找一个Node结点重新再创建一个新的Pod，来保证整个集群中始终存在两个Pod运行



注意：

删除RC并不会影响通过该RC资源对象创建好的Pod。如果要删除所有的Pod那么可以设置RC的replicas的值为0，然后更新该RC。

另外kubectl提供了stop和delete命令来一次性删除RC和RC控制的Pod。

Pod提供的是无状态服务，不会影响到客户的访问效果。

RS基础

- 基础

简介

由于Replication Controller与Kubernetes代码中的模块Replication Controller同名，而且这个名称无法准确表达它的本意，即Pod副本的控制，所以从kubernetes v1.2开始，它就升级成了一个新的概念：Replica Set（RS）。

RS和RC两者功能上没有太大的区别，只不过是表现形式上不一样：

RC中的Label Selector是基于等式的

RS中的Label Selector是基于集合的

因为集合的特点，这就使得Replica Set的功能更强大。

文件示例

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: …
  namespace: …
spec:
  minReadySeconds <integer>          # Pod就绪后多少秒内，Pod任一容器无crash方可视为“就绪”
  replicas <integer>                 # 期望的Pod副本数，默认为1
  selector:                          # 标签选择器，必须匹配template字段中Pod模板中的标签;
    matchExpressions <[]Object>      # 标签选择器表达式列表，多个列表项之间为“与”关系
    matchLabels <map[string]string> # map格式的标签选择器
  template:                          # Pod模板对象
    metadata:                        # Pod对象元数据
      labels:                        # 由模板创建出的Pod对象所拥有的标签，必须要能够匹配前面定义的标签选择器
    spec:                            # Pod规范，格式同自主式Pod
      ……
```

注意：
    RS和RC之间selector的格式区别

应用

RS很少单独使用，它主要是被Deployment这个更高层的资源对象所使用，从而形成了一整套Pod的创建、删除、更新的编排机制。
Replica Set 与Deployment这两个重要资源对象逐步替换了RC的作用

实现逻辑

Controller Manager 根据 ReplicaSet Control Loop 管理 ReplicaSet Object，由该对象向API Server请求管理Pod对象(标签选择器选定的)
    如果没有pod：
        以Pod模板向API Server请求创建Pod对象，由Scheduler调度并绑定至某节点，由相应节点kubelet负责运行。

- 简单实践

创建资源对象

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-test
spec:
  minReadySeconds: 0
  replicas: 3
  selector:
    matchLabels:
      app: rs-test
      release: stable
      version: v1.0
  template:
    metadata:
      labels:
        app: rs-test
        release: stable
        version: v1.0
    spec:
      containers:
      - name: rs-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1

应用资源对象
kubectl apply -f 05-controller-replicaset.yaml

查看效果
root@master1:~/controller# kubectl get rs
NAME              DESIRED   CURRENT   READY   AGE
replicaset-test   3         3         3       23s
root@master1:~/controller# kubectl get pod
NAME                    READY   STATUS    RESTARTS   AGE
replicaset-test-gkvbk   1/1     Running   0          26s
```

```
replicaset-test-nrmzs   1/1     Running   0          26s
replicaset-test-nxl9g   1/1     Running   0          26s
```

简单操作

```
删除一个Pod
root@master1:~/controller# kubectl delete pod replicaset-test-gkvbk
pod "replicaset-test-gkvbk" deleted
root@master1:~/controller# kubectl get pod
NAME                    READY   STATUS    RESTARTS   AGE
replicaset-test-gspxd   1/1     Running   0          34s
replicaset-test-nrmzs   1/1     Running   0          2m49s
replicaset-test-nxl9g   1/1     Running   0          2m49s
可以看到：
    删除一个Pod，RC又创建了一个Pod，证明上面我们对Replication Controller的作用分析没有任何问题。

RC调整Pod数量
基于对象调整副本数：kubectl scale --replicas=5 rc/rc_name
基于文件调整副本数：kubectl scale --replicas=3 -f rc_name.yaml

调整rc/nginx资源的pod副本数量
root@master1:~/controller# kubectl scale --replicas=4 rs/replicaset-test
replicaset.apps/replicaset-test scaled
root@master1:~/controller# kubectl get pod
NAME                    READY   STATUS    RESTARTS   AGE
replicaset-test-4kp5x   1/1     Running   0          3s         # 这个是新增加的
replicaset-test-gspxd   1/1     Running   0          109s
replicaset-test-nrmzs   1/1     Running   0          4m4s
replicaset-test-nxl9g   1/1     Running   0          4m4s

指定资源定义文件，缩小Pod数量
root@master1:~/controller# kubectl scale --replicas=1 -f 05-controller-
replicaset.yaml
replicaset.apps/replicaset-test scaled
root@master1:~/controller# kubectl get pod
NAME                    READY   STATUS        RESTARTS   AGE
replicaset-test-nxl9g   1/1     Running       0          4m55s
```

更新版本

```
更新命令
kubectl set image 资源类型/资源名称 pod名称=镜像版本

更新pod版本
kubectl set image replicaset/replicaset-test rs-
test=10.0.0.19:80/mykubernetes/pod_test:v0.2

查看效果
Croot@master1:~/controller# kubectl get pod -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP              NODE
...
replicaset-test-nxl9g   1/1     Running   0          27m   10.244.6.234    node2
...

root@master1:~/controller# kubectl get rs -o wide
```

```
NAME              DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES ...
replicaset-test   1         1         1       27m   rs-test
10.0.0.19:80/mykubernetes/pod_test:v0.2

root@master1:~/controller# curl 10.244.6.234
kubernetes pod-test v0.1 !! ClientIP: 10.244.0.0, ServerName: replicaset-test-
nxl9g, ServerIP: 10.244.6.234!
```

结果显式

虽然镜像的模板信息更新了，但是pod的访问效果没有变？

原因是 RS遵循的是 - 删除式更新，也就是说，只有删除老的replicaset，新生成的pod才会使用新的模板信息

删除就容器

```
root@master1:~/controller# kubectl delete pod replicaset-test-8w4gs

root@master1:~/controller# kubectl get pod -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP             NODE
...
replicaset-test-4xw4n   1/1     Running   0          37s   10.244.6.237   node2
  ...
root@master1:~/controller# curl 10.244.6.237
kubernetes pod-test v0.2!! ClientIP: 10.244.0.0, ServerName: replicaset-test-
4xw4n, ServerIP: 10.244.6.237!
```

结果显式：

RS的pod效果已经更新完毕了

小结

RS更新

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

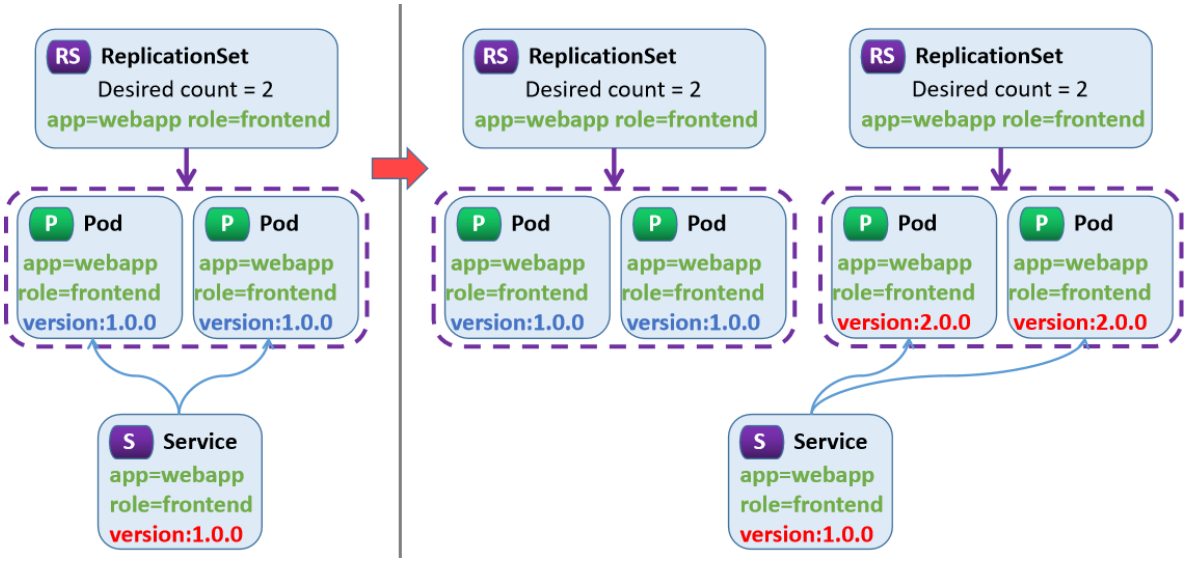
蓝绿部署

蓝绿部署是一种应用发布模式，可将用户流量从先前版本的应用或微服务逐渐转移到几乎相同的新版本中（两者均保持在生产环境中运行）。

旧版本可以称为蓝色环境，而新版本则可称为绿色环境。一旦生产流量从蓝色完全转移到绿色，蓝色就可以在回滚或退出生产的情况下保持待机，也可以更新成为下次更新的模板。

这种持续部署模式原本存在不足之处

- 并非所有环境都具有相同的正常运行时间要求或正确执行 CI/CD 流程（如蓝绿部署）所需的资源。
- 在部署过程中，可能导致大量资源占用

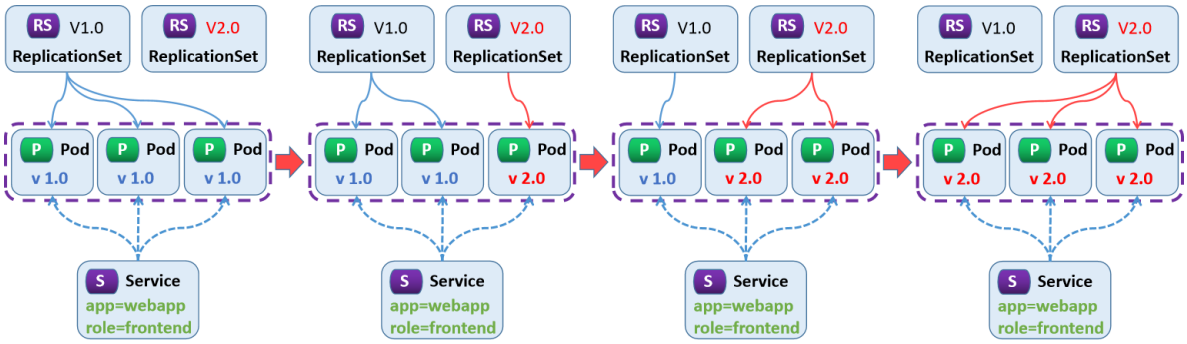


滚动更新

所谓的滚动更新是在蓝绿部署的基础上，加快了更新的效率，其特点在于：

- 一次只更新一小部分副本
- 上次更新成功后，再更新更多的副本
- 最终完成所有副本的更新

这个过程中，新旧版本同时存在，并不会导致大量的资源浪费。



关联关系

类型	蓝绿	滚动
配置文件	两个	两个
更新时间	长	短
资源需求	多	少
访问特点	一个版本	多个版本
部署影响	影响范围大	影响范围小

准备工作

清空pod

```
kubectl delete -f 05-controller-replicaset.yaml
```

准备service资源对象

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: replicaset-svc
spec:
  type: ClusterIP
  selector:
    app: rs-test
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80

应用资源对象
kubectl apply -f 06-controller-replicaset-svc.yaml

准备测试界面
kubectl run pod-$RANDOM --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 -it --rm --command -- /bin/bash
/ # while true; do curl --connect-timeout 1 replicaset-svc.default.svc; sleep .2;done
```

- 滚动更新

更新版本

```
准备两个版本的RS
cp 05-controller-replicaset.yaml 07-controller-replicaset-1.yaml
cp 05-controller-replicaset.yaml 07-controller-replicaset-2.yaml
更改 07-controller-replicaset-2.yaml 的版本信息和镜像版本
```

```
root@master1:~/controller# cat replicaset-deploy-1.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-test
spec:
  minReadySeconds: 0
  replicas: 3
  selector:
    matchLabels:
      app: rs-test
      release: stable
      version: v1.0
  template:
    metadata:
      labels:
        app: rs-test
        release: stable
        version: v1.0
    spec:
      containers:
      - name: rs-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
```

```
root@master1:~/controller# cat replicaset-deploy-2.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-test-2
spec:
  minReadySeconds: 0
  replicas: 0
  selector:
    matchLabels:
      app: rs-test
      release: stable
      version: v2.0
  template:
    metadata:
      labels:
        app: rs-test
        release: stable
        version: v2.0
    spec:
      containers:
      - name: rs-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.2
```

```
由于两个RS的标签不一样，所以可以同时启动，只不过一个数量为1，一个数量为0
kubectl apply -f 07-controller-replicaset-1.yaml
kubectl apply -f 07-controller-replicaset-2.yaml

查看效果
root@master1:~/controller# kubectl get rs
NAME                DESIRED   CURRENT   READY   AGE
replicaset-test     3         3         3       98s
replicaset-test-2   0         0         0       94s
```

轮询将 1版本减1，2版本加1

```
第1次更新
kubectl edit replicaset-test
    副本数量 - 1
kubectl edit replicaset-test-2
    副本数量 + 1

第2次更新
kubectl edit replicaset-test
    副本数量 - 1
kubectl edit replicaset-test-2
    副本数量 + 1

第3次更新
kubectl edit replicaset-test
    副本数量 - 1
kubectl edit replicaset-test-2
    副本数量 + 1

结果显式：
    更新的过程中，新旧版本是共存的效果

注意：
    版本的更换数量，可以手工自己定义
```

```
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-pxnfr, ServerIP: 10.244.5.152!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: replicaset-test-2-59f7w, ServerIP: 10.244.6.246!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-d8lvf, ServerIP: 10.244.5.153!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-pxnfr, ServerIP: 10.244.5.152!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: replicaset-test-2-59f7w, ServerIP: 10.244.6.246!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-d8lvf, ServerIP: 10.244.5.153!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-pxnfr, ServerIP: 10.244.5.152!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: replicaset-test-2-59f7w, ServerIP: 10.244.6.246!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-d8lvf, ServerIP: 10.244.5.153!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: replicaset-test-pxnfr, ServerIP: 10.244.5.152!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: replicaset-test-2-59f7w, ServerIP: 10.244.6.246!
```

```
整体来说，这种方式的滚动更新，太麻烦了
```

- 蓝绿更新

准备工作

```
清除之前的所有资源
kubectl delete -f 07-controller-replicaset-1.yaml
kubectl delete -f 07-controller-replicaset-2.yaml
kubectl delete -f 05-controller-replicaset.yaml
```

准备资料

```
由于之前两个 配置文件来回更新的话，导致操作繁琐，这次我们采用环境变量的方式来进行操作。
apiVersion: v1
kind: Service
metadata:
  name: replicaset-blue-green
spec:
  type: ClusterIP
  selector:
```

```
    app: rs-test
    ctr: rs-${DEPLOY}
    version: ${VERSION}
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-${DEPLOY}
spec:
  minReadySeconds: 3
  replicas: 2
  selector:
    matchLabels:
      app: rs-test
      ctr: rs-${DEPLOY}
      version: ${VERSION}
  template:
    metadata:
      labels:
        app: rs-test
        ctr: rs-${DEPLOY}
        version: ${VERSION}
    spec:
      containers:
      - name: pod-test
        image: 10.0.0.19:80/mykubernetes/pod_test:${VERSION}
```

基于envsubst的方式生成对应的文件，效果如下

```
DEPLOY=blue VERSION=v0.1 envsubst < 08-controller-replicaset-blue-green.yaml
```

```
root@master1:~/controller# DEPLOY=blue VERSION=v0.1 envsubst < replicaset-blue-green.yaml
apiVersion: v1
kind: Service
metadata:
  name: replicaset-blue-green
spec:
  type: ClusterIP
  selector:
    app: rs-test
    ctr: rs-blue
    version: v0.1
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-blue
spec:
  minReadySeconds: 3
  replicas: 2
  selector:
    matchLabels:
      app: rs-test
      ctr: rs-blue
      version: v0.1
  template:
    metadata:
      labels:
        app: rs-test
        ctr: rs-blue
        version: v0.1
    spec:
      containers:
      - name: pod_test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
```

执行配置文件

```
DEPLOY=blue VERSION=v0.1 envsubst < 08-controller-replicaset-blue-green |
kubectl apply -f -
```

注意：

最后面有一个短横线 "-"

发布绿版本

```
DEPLOY=green VERSION=v0.2 envsubst < 08-controller-replicaset-blue-green |
kubectl apply -f -
```

```
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: rs-blue-ddmq8, ServerIP: 10.244.6.249!
kubernetes pod-test v0.1!! ClientIP: 10.244.6.241, ServerName: rs-blue-gtg65, ServerIP: 10.244.5.154!
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (7) Failed to connect to replicaset-blue-green.default.svc port 80: Connection refused
curl: (28) Connection timed out after 1000 milliseconds
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: rs-green-m75mz, ServerIP: 10.244.5.155!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: rs-green-7rhqs, ServerIP: 10.244.6.250!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: rs-green-m75mz, ServerIP: 10.244.5.155!
kubernetes pod-test v0.2!! ClientIP: 10.244.6.241, ServerName: rs-green-7rhqs, ServerIP: 10.244.6.250!
```

结果显式：

进行蓝绿部署的时候，必须等到所有新的版本更新完毕后，再开放新的service，否则就会导致服务中断的效果

小结

Deployment基础

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

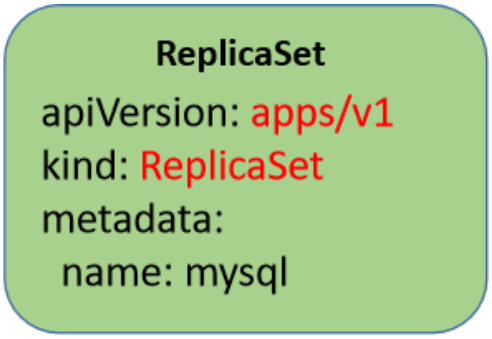
Deployment资源对象在内部使用Replica Set来实现Pod的自动化编排。Deployment资源对象不管是在作用、文件定义格式、具体操作等方面都可以看做RC的一次升级，两者的相似度达90%。
相对于RC的一个最大的升级是：
我们可以随时知道当前Pod的"部署"进度，即Pod创建--调度--绑定Node--在目标Node上启动容器。

场景

Deployment的使用场景非常多,一句话：只要是涉及到Pod资源对象的自动化管理，都是他的场景，
创建Deployment资源对象，生成对应的Replicas Set并完成Pod的自动管理
检查Deployment对象状态，检查Pod自动管理效果
扩展Deployment资源对象，以应对应用业务的高可用
...

资源定义

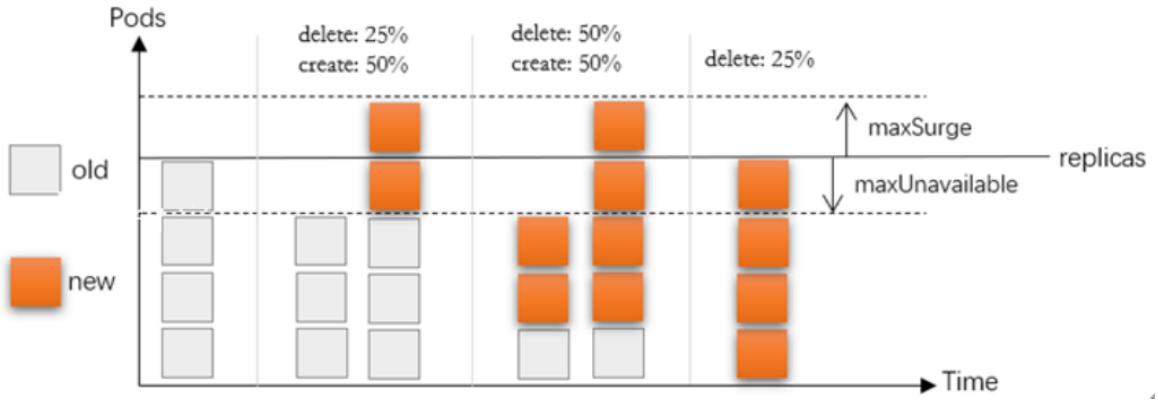
Deployment的定义与Replica Set的定义类似，除了API声明与Kind类型有所区别，其他基本上都一样。



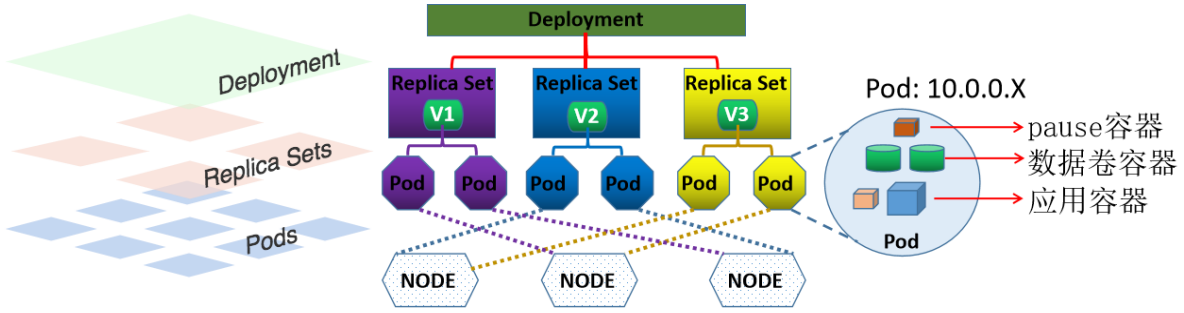
不过 Deployment对滚动更新多了一些更新策略的内容

```
apiVersion: apps/v1              # API群组及版本
kind: Deployment                 # 资源类型特有标识
metadata:
  name <string>                  # 资源名称，在作用域中要唯一
  namespace <string>             # 名称空间；Deployment隶属名称空间级别
spec:
  minReadySeconds <integer>      # Pod就绪后多少秒内任一容器无crash方可视为“就绪”
  replicas <integer>             # 期望的Pod副本数，默认为1
  selector <object>              # 标签选择器，必须匹配template字段中Pod模板中的标签
  template <object>              # Pod模板对象

  revisionHistoryLimit <integer> # 滚动更新历史记录数量，默认为10
  strategy <Object>              # 滚动更新策略
    type <string>                # 滚动更新类型，可用值有Recreate和RollingUpdate；
    rollingUpdate <Object>       # 滚动更新参数，专用于RollingUpdate类型
      maxSurge <string>          # 更新期间可比期望的Pod数量多出的数量或比例；
      maxUnavailable <string>    # 更新期间可比期望的Pod数量缺少的数量或比例，10，
  progressDeadlineSeconds <integer> # 滚动更新故障超时时长，默认为600秒
  paused <boolean>               # 是否暂停部署过程
```



部署相关的资源对象关系



简单实践

命令行创建对象

创建命令

```
kubectl create deployment pod-test --
image=10.0.0.19:80/mykubernetes/pod_test:v0.1 --replicas=3
```

查看效果

```
root@master1:~/controller# kubectl get deployments.apps pod-test
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
pod-test   3/3     3            3           8s
root@master1:~/controller# kubectl get pod
NAME                        READY   STATUS    RESTARTS   AGE
pod-3155                    1/1     Running   0          62m
pod-test-f79df5cfb-75g8m    1/1     Running   0          36s
pod-test-f79df5cfb-8bzck    1/1     Running   0          36s
pod-test-f79df5cfb-kmsnl    1/1     Running   0          36s
```

资源定义文件创建对象

创建资源定义文件

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginxpod-test
  template:
    metadata:
      labels:
        app: nginxpod-test
    spec:
      containers:
      - name: nginxpod-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
```

应用资源定义文件

```
kubectl apply -f 09-controller-deployment-test.yaml
```

检查关系

获取deployment对象信息

```
kubectl get deployments.apps
```

获取Replica Set对象信息

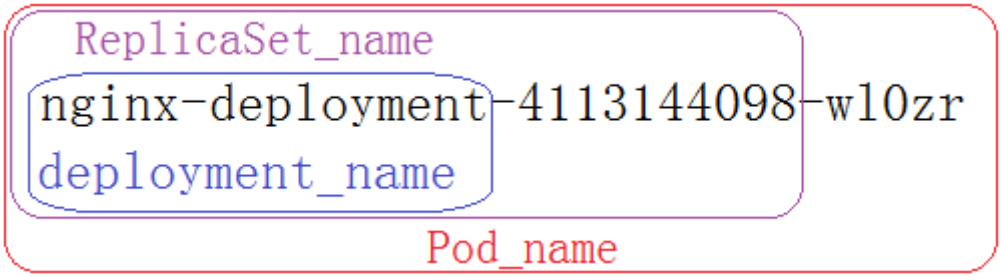
```
kubectl get rs deployment-test-97c5f9798
```

获取Pod对象信息

```
kubectl get pod
```

```
root@master1:~/controller# kubectl get deployments.apps
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   3/3     3            3           34s
root@master1:~/controller# kubectl get rs deployment-test-97c5f9798
NAME                         DESIRED   CURRENT   READY   AGE
deployment-test-97c5f9798    3         3         3       42s
root@master1:~/controller# kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
deployment-test-97c5f9798-jgbxr    1/1     Running   0          48s
deployment-test-97c5f9798-lp5lx    1/1     Running   0          48s
deployment-test-97c5f9798-ttvhr    1/1     Running   0          48s
```

结果显示：Pod的命名结构




```
查看 deployment的详细过程
kubectl describe deployment deployment-test
```

Deployment实践

学习目标

这一节，我们从 扩容缩容、动态更新、小结 三个方面来学习。

扩容缩容

基本语法

```
基于deployment调整Pod有两种方法：
基于资源对象调整：
kubectl scale <--current-replicas=3> --replicas=5 deployment/deploy_name
基于资源文件调整：
kubectl scale --replicas=4 -f deploy_name.yaml
```

简单实践

```
查看当前deployment对象信息
root@master1:~/deployment# kubectl get deployment
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   3/3     3            3           26s

基于资源对象扩充Pod数量，携带当前副本数量信息--current-replicas
]# kubectl scale --current-replicas=3 --replicas=5 deployment/deployment-test
deployment "deployment-test" scaled
root@master1:~/deployment# kubectl get deployment deployment-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   5/5     5            5           102s

基于资源对象扩充Pod数量，不携带当前副本数量信息
```

```
]# kubectl scale --replicas=2 deployment/deployment-test
deployment "nginx-deployment" scaled
root@master1:~/deployment# kubectl get deployment deployment-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   2/2     2            2           2m46s

基于资源文件调整Pod数量
]# kubectl scale --replicas=4 -f deployment-test.yaml
deployment "deployment-test" scaled
root@master1:~/deployment# kubectl get deployment deployment-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   4/4     4            4           3m13s
```

动态更新

命令简介

```
更新命令1：kubectl set SUBCOMMAND [options] 资源类型 资源名称
    子命令：常用的子命令就是image
    参数详解：
        --record=true        更改的时候，将信息增加到历史记录中

更新命令2：kubectl patch (-f FILENAME | TYPE NAME) -p PATCH [options]
    参数详解：
    --patch=''      设定对象属性内容

回滚命令： kubectl rollout SUBCOMMAND [options] 资源类型 资源名称
    子命令：
    history     显示 rollout 历史
    pause       标记提供的 resource 为中止状态，而pause目前仅仅支持 deployment对象
    resume      继续一个停止的 resource
    status      显示 rollout 的状态
    undo        撤销上一次的 rollout
```

简单实践

```
更改对象
kubectl set image deployment/deployment-test nginxpod-
test='10.0.0.19:80/mykubernetes/pod_test:v0.1' --record=true
kubectl set image deployment/deployment-test nginxpod-
test='10.0.0.19:80/mykubernetes/pod_test:v0.2' --record=true
注意：
    nginxpod-test 是容器名称，10.0.0.19:80/mykubernetes/pod_test:v0.1 是 image的属
性值。

查看更新状态
]# kubectl rollout status deployment deployment-test

查看历史
root@master1:~/deployment# kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
2         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.2 --record=true
```

```
3         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.1 --record=true
```

撤销更改

```
]# kubectl rollout undo deployment deployment-test
]# kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
3         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.1 --record=true
4         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.2 --record=true
```

更改1个后，就暂停，查看效果

```
]# kubectl set image deployment/deployment-test nginxpod-
test='10.0.0.19:80/mykubernetes/nginx:1.21.3' --record=true && kubectl rollout
pause deployment deployment-test
]# kubectl rollout status deployment deployment-test
Waiting for deployment "deployment-test" rollout to finish: 1 out of 4 new
replicas have been updated....
```

继续更新

```
]# kubectl rollout resume deployment deployment-test
]# kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
3         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.1 --record=true
4         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.2 --record=true
5         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/nginx:1.21.3 --record=true
```

回到版本3

```
]# kubectl rollout undo --to-revision=3 deployment deployment-test
deployment.apps/deployment-test rolled back
]# kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
4         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.2 --record=true
5         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/nginx:1.21.3 --record=true
6         kubectl set image deployment/deployment-test nginxpod-
test=10.0.0.19:80/mykubernetes/pod_test:v0.1 --record=true
```

结果显示：

将原来的3号，拿到了6号位置，重新执行了一遍，原来的3号就没有了

打补丁实践

修改副本数量

```
]# kubectl patch deployment deployment-test -p '{"spec":{"replicas":2}}'
root@master1:~/deployment# kubectl get deployment
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   2/2     2            2           15m
```

修改期望数据

```
]# kubectl  patch deployment deployment-test -p '{"spec":{"strategy":
{"rollingUpdate":{"maxSurge":1,"maxUnavailable":0}}}}'
```

```
]# kubectl describe deployments deployment-test
Name:                   nginx-deployment
...
RollingUpdateStrategy:  0 max unavailable, 1 max surge
...
```

小结

滚动更新

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

需求

> 对于应用的动态扩展来说，还有一种叫滚动更新，其实我们刚才rollout的效果就是一种特殊的滚动更新，只不过是一个一个的变动，而真正的滚动更新远远要比默认的效果灵活.

属性解析

```
kubectl explain deployment.spec.strategy
rollingUpdate        <Object>
  maxSurge              <string>     更新时候，容量允许扩大的值
  maxUnavailable        <string>     多少个pod不能用时候，再来更新
type <string>        主要有两种类型："Recreate"、"RollingUpdate-默认"
```

属性	解析
minReadySeconds	Kubernetes在等待设置的时间后才进行升级 如果没有设置该值，Kubernetes会假设该容器启动起来后就提供服务了 如果没有设置该值，在某些极端情况下可能会造成服务不正常运行
maxSurge	升级过程中最多可以比原先设置多出的POD数量，默认是25% 例如：maxSurage=1，replicas=5,则表示Kubernetes会先启动1一个新的Pod后才删掉一个旧的POD，整个升级过程中最多会有5+1个POD。
maxUnavaible	升级过程中最多有多少个POD处于无法提供服务的状态，默认是25% 当maxSurge不为0时，该值也不能为0 例如：maxUnavaible=1，则表示Kubernetes整个升级过程中最多会有1个POD处于无法服务的状态。

基本样式

```
这里有一个他们的基本属性样式：
minReadySeconds: 5
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

简单实践

定制配置文件

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-update
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginxpod-test
  template:
    metadata:
      labels:
        app: nginxpod-test
    spec:
      containers:
      - name: pod-roll
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 2
  属性解析：
    minReadySeconds: 5 表示我们在更新的时候，需要先等待5秒，而不是一旦发生变化就滚动更新
    maxSurge: 2 表示我们在更新的时候，可以先增加两个新的，然后再删除多出来的两个旧的
    maxUnavailable: 2 表示当我们的资源数量有两个故障的时候，自动进行增加
```

简单实践

```
应用资源文件
kubectl apply -f 10-controller-deployment-rollupdate.yaml
查看效果
kubectl rollout status deployment rolling-update
调整后继续检查
kubectl set image deployment/rolling-update pod-
roll='10.0.0.19:80/mykubernetes/pod_test:v0.2' --record=true
kubectl rollout status deployment rolling-update
kubectl get pod -w
```

问题：

更新后的pod是否还是在原来的node结点上呢？
原生的kubernetes是没有这个功能，需要你根据实际的调度策略来做，或者找相关的云平台来实现

小结

DaemonSet

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

DaemonSet能够让所有（或者特定）的节点"精确的"运行同一个pod，它一般应用在集群环境中所有节点都必须运行的守护进程的场景。
我们在部署k8s环境的时候，网络的部署样式就是基于这种DaemonSet的方式，因为对于网络来说，是所有节点都必须具备的基本能力，而且不能随意中断，否则的话，节点上的容器通信就会出现问题，样式如下。

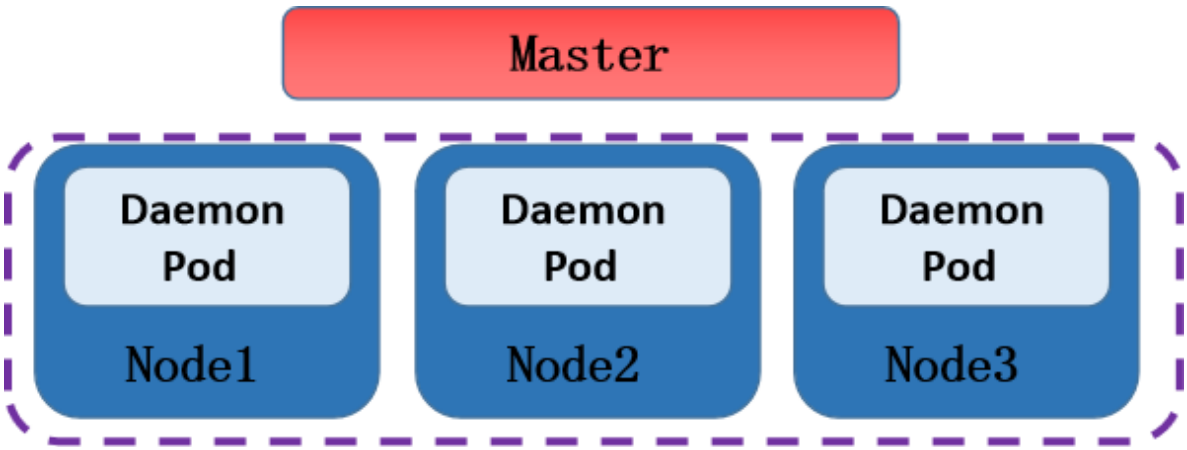
```
root@master1:~/mykubernetes/flannel# grep kind kube-flannel.yml
...
kind: DaemonSet
结果显示：
    除了权限和认证、配置之外的资源对象都是以DaemonSet的方式在运行。

root@master1:~/daemonset# kubectl get ds -n kube-system
NAME               DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE
SELECTOR             AGE
kube-flannel-ds    3         3         3       3            3           <none>
                 4h57m
kube-proxy         3         3         3       3            3
kubernetes.io/os=linux   5h2m
```

当节点加入到K8S集群中，pod会被（DaemonSet）调度到该节点上运行，当节点从K8S集群中被移除，被DaemonSet调度的pod会被移除，如果删除DaemonSet，所有跟这个DaemonSet相关的pods都会被删除。

在某种程度上，DaemonSet承担了RC的部分功能，它也能保证相关pods持续运行，如果一个DaemonSet的Pod被杀死、停止、或者崩溃，那么DaemonSet将会重新创建一个新的副本在这台计算节点上。

常用于后台支撑服务
    集群存储守护进程，如：glusterd，ceph
    日志收集服务，如：fluentd，logstash
    监控服务，如：Prometheus，collectd



属性解析

```
apiVersion: apps/v1                    # API群组及版本
kind: DaemonSet                        # 资源类型特有标识
metadata:
  name <string>                        # 资源名称，在作用域中要唯一
  namespace <string>                   # 名称空间；DaemonSet资源隶属名称空间级别
spec:
  minReadySeconds <integer>            # Pod就绪后多少秒内任一容器无crash方可视为“就绪”
  selector <object>                    # 标签选择器，必须匹配template字段中Pod模板中的标
签
  template <object>                    # Pod模板对象；

  revisionHistoryLimit <integer>       # 滚动更新历史记录数量，默认为10；
  updateStrategy <Object>              # 滚动更新策略
    type <string>                      # 滚动更新类型，可用值有OnDelete和
RollingUpdate；
    rollingUpdate <Object>             # 滚动更新参数，专用于RollingUpdate类型
      maxUnavailable <string>          # 更新期间可比期望的Pod数量缺少的数量或比例
```

简单实践

需求

之前我们在Node上启动Pod需要在RC中指定replicas的副本数的值，有些情况下，我们需要在所有节点都运行一个Pod，因为Node数量会变化，所以Pod的副本数使用RC来指定就不合适了，这个时候Daemon Sets就派上了用场。简单来说，Daemon Sets就是让一个pod在所有的k8s集群节点上都运行一个。

定制资源定义文件

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset-test
spec:
  selector:
    matchLabels:
      app: pod-test
  template:
    metadata:
      labels:
        app: pod-test
```

```
    spec:
      containers:
      - name: pod-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.2

创建资源对象
]# kubectl create -f 11-controller-daemonset-test.yaml
```

效果演示

```
检查效果
root@master1:~/daemonset# kubectl get daemonset
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE
SELECTOR   AGE
daemonset-test   2         2         2       2            2           <none>
     39s
root@master1:~/daemonset# kubectl get pod -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP             NODE
。。。
daemonset-test-fk27t   1/1     Running   0          39s   10.244.2.26   node2
。。。
daemonset-test-ntmcd   1/1     Running   0          39s   10.244.1.16   node1
。。。

daemonset对象也支持滚动更新
kubectl set image  daemonsets daemonset-test pod-
test='10.0.0.19:80/mykubernetes/pod_test:v0.1' --record=true  && kubectl rollout
status daemonset daemonset-test
kubectl describe daemonsets daemonset-test

注意：
    daemonsets对象不支持pause动作
```

实践2 - 监控软件在所有节点上都部署采集数据的功能

```
定制基本配置文件
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset-demo
  namespace: default
  labels:
    app: prometheus
    component: node-exporter
spec:
  selector:
    matchLabels:
      app: prometheus
      component: node-exporter
  template:
    metadata:
      name: prometheus-node-exporter
      labels:
        app: prometheus
        component: node-exporter
    spec:
```

```
      containers:
      - image: 10.0.0.19:80/mykubernetes/node-exporter:v1.2.2
        name: prometheus-node-exporter
        ports:
        - name: prom-node-exp
          containerPort: 9100
          hostPort: 9100
        livenessProbe:
          tcpSocket:
            port: prom-node-exp
          initialDelaySeconds: 3
        readinessProbe:
          httpGet:
            path: '/metrics'
            port: prom-node-exp
            scheme: HTTP
          initialDelaySeconds: 5
      hostNetwork: true
      hostPID: true
```

属性解析：

hostNetwork 和 hostPID 容器共享宿主机的 网络和PID

应用配置文件

```
kubectl apply -f 12-controller-daemonset-prometheus.yaml
```

确认效果

```
root@master1:~/daemonset# kubectl get pod -o wide
NAME                    READY   STATUS    RESTARTS   AGE     IP          NODE
...
daemonset-demo-mnxd5    1/1     Running   0          8m16s   10.0.0.15   node1
...
daemonset-demo-tcm2n    1/1     Running   0          8m16s   10.0.0.16   node2
...
```

获取相关的数据

```
curl 10.0.0.15:9100/metrics
```

小结

Job

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

场景

> 在我们日常的工作中，经常会遇到临时执行一个动作，但是这个动作必须在某个时间点执行才可以，而我们又不想一直这么傻傻的等待，即使等待到了，由于特殊原因，我们执行的时候，已经不是准确的时间点了。
> 针对于这种场景，我们一般使用job的方式来帮助我们定制化的完成任务。

分类

在k8s场景中，关于job的执行，主要有两种类型：串行job、并行job。
串行job：即所有的job任务都在上一个job执行完毕后，再开始执行
并行job：如果存在多个job，我们可以设定并行执行的job数量。

属性解析

```
apiVersion: batch/v1                        # API群组及版本
kind: Job                                   # 资源类型特有标识
metadata:
  name <string>                             # 资源名称，在作用域中要唯一
  namespace <string>                        # 名称空间；Job资源隶属名称空间级别
spec:
  selector <object>                         # 标签选择器，必须匹配template字段中Pod模板中
的标签
  template <object>                         # Pod模板对象
  completions <integer>                     # 期望的成功完成的作业次数，成功运行结束的Pod数
量
  ttlSecondsAfterFinished  <integer>    # 终止状态作业的生存时长，超期将被删除
  parallelism  <integer>                    # 作业的最大并行度，默认为1
  backoffLimit <integer>                    # 将作业标记为Failed之前的重试次数，默认为6
  activeDeadlineSeconds  <integer>      # 作业启动后可处于活动状态的时长
```

配置示例

```
串行运行共5次任务：
spec
  parallelism: 1
  completion: 5

并行2队列共运行5次任务：
spec
  parallelism: 2
  completion: 5
```

简单实践

实践1 - 单个任务

```
定制资源配置文件
apiVersion: batch/v1
kind: Job
metadata:
  name: job-single
spec:
  template:
    metadata:
      name: job-single
    spec:
      restartPolicy: Never
      containers:
      - name: job-single
        image: 10.0.0.19:80/mykubernetes/admin-box:v0.1
        command: [ "/bin/sh", "-c", "for i in 9 8 7 6 5 4 3 2 1; do echo $i;
sleep 2; done" ]
```

```
    属性解析：
        对于job来说，他的重启策略只有两种：仅支持Never和OnFailure两种，
        不支持Always，否则的话就成死循环了。

应用资源文件
kubectl apply -f 13-controller-job-single.yaml

检查效果
root@master1:~/job# kubectl get pod
NAME                   READY   STATUS      RESTARTS   AGE
job-single--1-jz4rk    0/1     Completed   0          30s
    结果显示：
        job任务执行完毕后，状态是 Completed

]# kubectl logs job-single-q6jz4 --timestamps=true
... 09:15:48 ... 9
... 09:15:50 ... 8
... 09:15:52 ... 7
... 09:15:54 ... 6
... 09:15:56 ... 5
... 09:15:58 ... 4
... 09:16:00 ... 3
... 09:16:02 ... 2
... 09:16:04 ... 1
]# kubectl describe jobs.batch
Name:           job-single
...
```

实践2 - 多个串行任务

```
资源定义文件 14-controller-job-multi-chuan.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-multi-chuan
spec:
  completions: 5
  parallelism: 1
  template:
    spec:
      containers:
      - name: job-multi
        image: 10.0.0.19:80/mykubernetes/admin-box:v0.1
        command: ["/bin/sh","-c","echo job; sleep 3"]
      restartPolicy: OnFailure

执行资源定义文件
kubectl apply -f 14-controller-job-multi-chuan.yaml
查看效果
]# kubectl get job
NAME              COMPLETIONS   DURATION   AGE
job-multi-chuan   5/5           29s        29s

]# job_list=$(kubectl get pod | grep job | sort -k 5 | awk '{print $1}')
]# for i in $job_list; do kubectl logs $i --timestamps=true; done
... 09:55:01 ... job
... 09:54:56 ... job
```

```
... 09:54:50 ... job
... 09:54:41 ... job
... 09:54:35 ... job
```

结果显示：

这些任务，确实是串行的方式来执行，由于涉及到任务本身是启动和删除，所以时间间隔要大于3s

实践3 - 并行任务

资源定义文件 15-controller-job-multi-bing.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-multi-bing
spec:
  completions: 6
  parallelism: 2
  template:
    spec:
      containers:
      - name: job-multi-bing
        image: 10.0.0.19:80/mykubernetes/admin-box:v0.1
        command: ["/bin/sh","-c","echo job; sleep 3"]
      restartPolicy: OnFailure
```

执行资源定义文件

```
kubectl apply -f 15-controller-job-multi-bing.yaml
```

查看效果

```
]# kubectl get job -w
NAME             COMPLETIONS   DURATION   AGE
job-multi-bing   2/6           11s        11s
job-multi-bing   3/6           12s        12s
job-multi-bing   4/6           12s        12s
job-multi-bing   5/6           18s        18s
job-multi-bing   6/6           18s        18s

]# job_list=$(kubectl get pod | grep job | sort -k 5 | awk '{print $1}')
]# for i in $job_list; do kubectl logs $i --timestamps=true; done
... 09:51:13 ... job
... 09:51:13 ... job
... 09:51:06 ... job
... 09:51:05 ... job
... 09:50:58 ... job
... 09:50:58 ... job
```

结果显示：

这6条任务确实是两两并行执行的，

小结

CronJob

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

需求

job的功能是对于某些临时任务来说是非常好的一个功能，但是我们还会遇到更多更常见的定时任务，而对于定时任务，k8s给我们提供了 Cronjob的功能。

简介

CronJob其实就是在Job的基础上加上了时间调度，我们可以：在给定的时间点运行一个任务，也可以周期性地在给定时间点运行。其效果与linux中的crontab效果非常类似，一个CronJob对象其实就对应中crontab文件中的一行，它根据配置的时间格式周期性地运行一个Job，格式和crontab也是一样的。

crontab的格式如下：

```
   分       时        日        月       周      命令
 (0～59)  (0～23)  (1～31)  (1～12)  (0～7)
```

属性解析

```
apiVersion: batch/v1                      # API群组及版本
kind: CronJob                             # 资源类型特有标识
metadata:
  name <string>                           # 资源名称，在作用域中要唯一
  namespace <string>                      # 名称空间；CronJob资源隶属名称空间级别
spec:
  jobTemplate  <Object>                   # job作业模板，必选字段
    metadata <object>                     # 模板元数据
    spec <object>                         # 作业的期望状态
  schedule <string>                       # 调度时间设定，必选字段
  concurrencyPolicy  <string>             # 并发策略，可用值有Allow、Forbid和Replace
  failedJobsHistoryLimit <integer>        # 失败作业的历史记录数，默认为1
  successfulJobsHistoryLimit  <integer> # 成功作业的历史记录数，默认为3
  startingDeadlineSeconds  <integer>    # 因错过时间点而未执行的作业的可超期时长
  suspend  <boolean>                      # 是否挂起后续的作业，不影响当前作业，默认为
false
```

简单实践

实践1 - 简单周期任务

资源定义文件 16-controller-cronjob-simple.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: cronjob
```

```
            image: 10.0.0.19:80/mykubernetes/admin-box:v0.1
            command: ["/bin/sh","-c","echo job"]
```

执行资源定义文件

```
kubectl apply -f 16-controller-cronjob-simple.yaml
```

查看效果

```
]# kubectl get cronjobs.batch
NAME      SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob   */2 * * * *   False     0        44s             53s

]# kubectl get jobs.batch
NAME                  COMPLETIONS   DURATION   AGE
cronjob-1562754000    1/1           3s         5m29s
cronjob-1562754120    1/1           3s         3m29s
cronjob-1562754240    1/1           4s         89s
```

注意：

这是6分钟过后才进行查看的效果

```
# cornjob_list=$(kubectl get pod | grep cronjob | awk '{print $1}')
]# for i in $cornjob_list ;do kubectl logs $i --timestamps=true; done
... 10:18:02 ... job
... 10:20:02 ... job
... 10:22:02 ... job
```

实践2 - 秒级周期任务

资源定义文件 17-controller-cronjob-second.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cronjob-second
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: cronjob
            image: 10.0.0.19:80/mykubernetes/admin-box:v0.1
            command: ["/bin/sh","-c","i=0; until [ $i -eq 60 ]; do sleep 10; let
i=i+10; echo $i job; done"]
```

属性解析：

我们这里的调度是每秒都执行后面的命令

命令是每执行一下休息10s

注意：这里的10s应该符合原则上的时间切分规则 -- 整除

执行资源定义文件

```
kubectl apply -f cronjob-second.yaml
```

查看效果

```
]# kubectl get cronjobs.batch
NAME             SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob-second   * * * * *   False     1        17s             30s

root@master1:~/job# kubectl get jobs.batch
```

```
NAME                        COMPLETIONS   DURATION   AGE
cronjob-second-27220909   1/1           61s        108s
cronjob-second-27220910   0/1           48s        48s
root@master1:~/job# kubectl get pod
NAME                                 READY   STATUS      RESTARTS   AGE
cronjob-second-27220909--1-gn64w   0/1     Completed   0          111s
cronjob-second-27220910--1-r524z   1/1     Running     0          51s
```

结果显示：

周期性任务每分钟执行一次

```
root@master1:~/job# kubectl logs cronjob-second-27220909--1-gn64w -f
10 job
20 job
30 job
40 job
50 job
60 job
```

结果显示：

每个任务中，周期性的执行一个动作

小结