

# 第六章 Pod控制器详解

本章节主要介绍各种Pod控制器的详细使用。

## Pod控制器介绍

Pod是kubernetes的最小管理单元，在kubernetes中，按照pod的创建方式可以将其分为两类：

- 自主式pod：kubernetes直接创建出来的Pod，这种pod删除后就没有了，也不会重建
- 控制器创建的pod：kubernetes通过控制器创建的pod，这种pod删除了之后还会自动重建

### 什么是Pod控制器

Pod控制器是管理pod的中间层，使用Pod控制器之后，只需要告诉Pod控制器，想要多少个什么样的Pod就可以了，它会创建出满足条件的Pod并确保每一个Pod资源处于用户期望的目标状态。如果Pod资源在运行中出现故障，它会基于指定策略重新编排Pod。

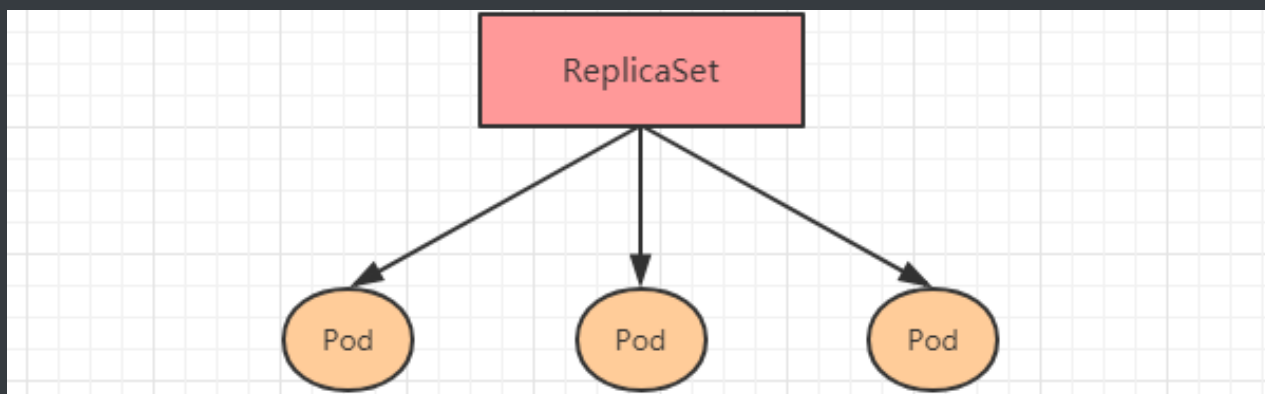
在kubernetes中，有很多类型的pod控制器，每种都有自己的适合的场景，常见的有下面这些：

- ReplicationController：比较原始的pod控制器，已经被废弃，由ReplicaSet替代
- ReplicaSet：保证副本数量一直维持在期望值，并支持pod数量扩缩容，镜像版本升级
- Deployment：通过控制ReplicaSet来控制Pod，并支持滚动升级、回退版本
- Horizontal Pod Autoscaler：可以根据集群负载自动水平调整Pod的数量，实现削峰填谷
- DaemonSet：在集群中的指定Node上运行且仅运行一个副本，一般用于守护进程类的任务
- Job：它创建出来的pod只要完成任务就立即退出，不需要重启或重建，用于执行一次性任务
- Cronjob：它创建的Pod负责周期性任务控制，不需要持续后台运行

- StatefulSet: 管理有状态应用

## ReplicaSet(RS)

ReplicaSet的主要作用是**保证一定数量的pod正常运行**，它会持续监听这些Pod的运行状态，一旦Pod发生故障，就会重启或重建。同时它还支持对pod数量的扩缩容和镜像版本的升降级。



ReplicaSet的资源清单文件:

```
apiVersion: apps/v1 # 版本号
kind: ReplicaSet # 类型
metadata: # 元数据
  name: # rs名称
  namespace: # 所属命名空间
  labels: # 标签
  controller: rs
spec: # 详情描述
  replicas: 3 # 副本数量
  selector: # 选择器, 通过它指定该控制器管理哪些pod
    matchLabels: # Labels匹配规则
      app: nginx-pod
    matchExpressions: # Expressions匹配规则
      - {key: app, operator: In, values: [nginx-pod]}
  template: # 模板, 当副本数量不足时, 会根据下面的模板创建pod副本
    metadata:
      labels:
        app: nginx-pod
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```

在这里面，需要新了解的配置项就是 spec 下面几个选项：

- replicas：指定副本数量，其实就是当前rs创建出来的pod的数量，默认为1
- selector：选择器，它的作用是建立pod控制器和pod之间的关联关系，采用的Label Selector机制  
在pod模板上定义label，在控制器上定义选择器，就可以表明当前控制器能管理哪些pod了
- template：模板，就是当前控制器创建pod所使用的模板板，里面其实就是前一章学过的pod的定义

## 创建ReplicaSet

创建pc-replicaset.yaml文件，内容如下：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: pc-replicaset
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      labels:
        app: nginx-pod
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.17.1
```

# 创建rs

```
[root@master ~]# kubectl create -f pc-replicaset.yaml
replicaset.apps/pc-replicaset created
```

# 查看rs

# DESIRED:期望副本数量

# CURRENT:当前副本数量

# READY:已经准备好提供服务的副本数量

```
[root@master ~]# kubectl get rs pc-replicaset -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES
pc-replicaset	3	3	3	22s	nginx	nginx:1.17.1
app=nginx-pod						

# 查看当前控制器创建出来的pod

# 这里发现控制器创建出来的pod的名称是在控制器名称后面拼接了-xxxxx随机码

```
[root@master ~]# kubectl get pod -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-replicaset-6vmvt	1/1	Running	0	54s
pc-replicaset-fmb8f	1/1	Running	0	54s
pc-replicaset-snrk2	1/1	Running	0	54s

## 扩缩容

# 编辑rs的副本数量, 修改spec:replicas: 6即可

```
[root@master ~]# kubectl edit rs pc-replicaset -n dev
replicaset.apps/pc-replicaset edited
```

# 查看pod

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-replicaset-6vmvt	1/1	Running	0	114m
pc-replicaset-cftnp	1/1	Running	0	10s
pc-replicaset-fjlm6	1/1	Running	0	10s
pc-replicaset-fmb8f	1/1	Running	0	114m
pc-replicaset-s2whj	1/1	Running	0	10s
pc-replicaset-snrk2	1/1	Running	0	114m

# 当然也可以直接使用命令实现

# 使用scale命令实现扩缩容，后面--replicas=n直接指定目标数量即可

```
[root@master ~]# kubectl scale rs pc-replicaset --replicas=2 -n dev
replicaset.apps/pc-replicaset scaled
```

# 命令运行完毕，立即查看，发现已经有4个开始准备退出了

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-replicaset-6vmvt	0/1	Terminating	0	118m
pc-replicaset-cftnp	0/1	Terminating	0	4m17s
pc-replicaset-fjlm6	0/1	Terminating	0	4m17s
pc-replicaset-fmb8f	1/1	Running	0	118m
pc-replicaset-s2whj	0/1	Terminating	0	4m17s
pc-replicaset-snrk2	1/1	Running	0	118m

#稍等片刻，就只剩下2个了

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-replicaset-fmb8f	1/1	Running	0	119m
pc-replicaset-snrk2	1/1	Running	0	119m

## 镜像升级

# 编辑rs的容器镜像 - image: nginx:1.17.2

```
[root@master ~]# kubectl edit rs pc-replicaset -n dev
replicaset.apps/pc-replicaset edited
```

# 再次查看，发现镜像版本已经变更了

```
[root@master ~]# kubectl get rs -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS
pc-replicaset	2	2	2	140m	nginx

```

nginx:1.17.2 ...

# 同样的道理，也可以使用命令完成这个工作
# kubectl set image rs rs名称 容器=镜像版本 -n namespace
[root@master ~]# kubectl set image rs pc-replicaset nginx=nginx:1.17.1 -n dev
replicaset.apps/pc-replicaset image updated

# 再次查看，发现镜像版本已经变更了
[root@master ~]# kubectl get rs -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS
pc-replicaset	2	2	2	145m	nginx

```

nginx:1.17.1 ...

```

## 删除ReplicaSet

```
# 使用kubectl delete命令会删除此RS以及它管理的Pod
# 在kubernetes删除RS前，会将RS的replicasclear调整为0，等待所有的Pod被删除后，在执行RS对象的删除
[root@master ~]# kubectl delete rs pc-replicaset -n dev
replicaset.apps "pc-replicaset" deleted
[root@master ~]# kubectl get pod -n dev -o wide
No resources found in dev namespace.

# 如果希望仅仅删除RS对象（保留Pod），可以使用kubectl delete命令时添加--cascade=false选项（不推荐）。
[root@master ~]# kubectl delete rs pc-replicaset -n dev --cascade=false
replicaset.apps "pc-replicaset" deleted
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-replicaset-cl82j	1/1	Running	0	75s

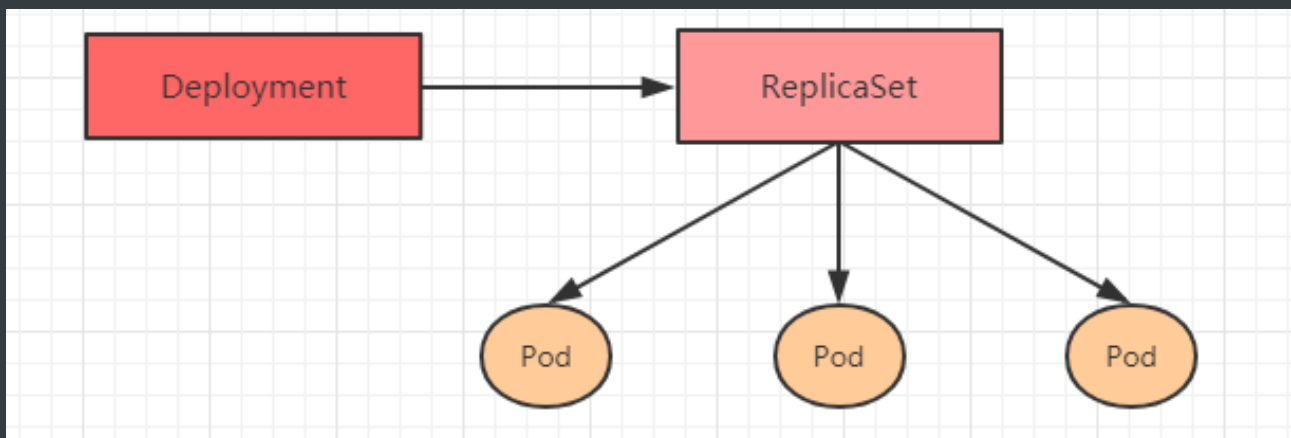
```
pc-replicaset-dslhb 1/1 Running 0 75s
```

# 也可以使用yaml直接删除(推荐)

```
[root@master ~]# kubectl delete -f pc-replicaset.yaml  
replicaset.apps "pc-replicaset" deleted
```

## Deployment(Deploy)

为了更好的解决服务编排的问题，kubernetes在V1.2版本开始，引入了Deployment控制器。值得一提的是，这种控制器并不直接管理pod，而是通过管理ReplicaSet来简介管理Pod，即：Deployment管理ReplicaSet，ReplicaSet管理Pod。所以Deployment比ReplicaSet功能更加强大。



Deployment主要功能有下面几个：

- 支持ReplicaSet的所有功能
- 支持发布的停止、继续
- 支持滚动升级和回滚版本

Deployment的资源清单文件：

```
apiVersion: apps/v1 # 版本号  
kind: Deployment # 类型  
metadata: # 元数据  
  name: # rs名称  
  namespace: # 所属命名空间  
  labels: #标签
```

```

    controller: deploy
spec: # 详情描述
  replicas: 3 # 副本数量
  revisionHistoryLimit: 3 # 保留历史版本
  paused: false # 暂停部署, 默认是false
  progressDeadlineSeconds: 600 # 部署超时时间 (s) , 默认是600
  strategy: # 策略
    type: RollingUpdate # 滚动更新策略
    rollingUpdate: # 滚动更新
      maxSurge: 30% # 最大额外可以存在的副本数, 可以为百分比, 也可以为整数
      maxUnavailable: 30% # 最大不可用状态的 Pod 的最大值, 可以为百分比, 也可以
为整数
  selector: # 选择器, 通过它指定该控制器管理哪些pod
    matchLabels: # Labels匹配规则
      app: nginx-pod
    matchExpressions: # Expressions匹配规则
      - {key: app, operator: In, values: [nginx-pod]}
  template: # 模板, 当副本数量不足时, 会根据下面的模板创建pod副本
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.17.1
          ports:
            - containerPort: 80

```

## 创建deployment

创建pc-deployment.yaml, 内容如下:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pc-deployment

```



```

namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.1

```

#### # 创建deployment

```

[root@master ~]# kubectl create -f pc-deployment.yaml --record=true
deployment.apps/pc-deployment created

```

#### # 查看deployment

# UP-TO-DATE 最新版本的pod的数量

# AVAILABLE 当前可用的pod的数量

```

[root@master ~]# kubectl get deploy pc-deployment -n dev

```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
pc-deployment	3/3	3	3	15s

#### # 查看rs

# 发现rs的名称是在原来deployment的名字后面添加了一个10位数的随机串

```

[root@master ~]# kubectl get rs -n dev

```

NAME	DESIRED	CURRENT	READY	AGE
pc-deployment-6696798b78	3	3	3	23s

#### # 查看pod

```

[root@master ~]# kubectl get pods -n dev

```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-6696798b78-d2c8n	1/1	Running	0	107s

pc-deployment-6696798b78-smpvp	1/1	Running	0	107s
pc-deployment-6696798b78-wvjd8	1/1	Running	0	107s

## 扩缩容

# 变更副本数量为5个

```
[root@master ~]# kubectl scale deploy pc-deployment --replicas=5 -n dev
deployment.apps/pc-deployment scaled
```

# 查看deployment

```
[root@master ~]# kubectl get deploy pc-deployment -n dev
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
pc-deployment	5/5	5	5	2m

# 查看pod

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-6696798b78-d2c8n	1/1	Running	0	4m19s
pc-deployment-6696798b78-jxmdq	1/1	Running	0	94s
pc-deployment-6696798b78-mktqv	1/1	Running	0	93s
pc-deployment-6696798b78-smpvp	1/1	Running	0	4m19s
pc-deployment-6696798b78-wvjd8	1/1	Running	0	4m19s

# 编辑deployment的副本数量, 修改spec:replicas: 4即可

```
[root@master ~]# kubectl edit deploy pc-deployment -n dev
deployment.apps/pc-deployment edited
```

# 查看pod

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-6696798b78-d2c8n	1/1	Running	0	5m23s
pc-deployment-6696798b78-jxmdq	1/1	Running	0	2m38s
pc-deployment-6696798b78-smpvp	1/1	Running	0	5m23s
pc-deployment-6696798b78-wvjd8	1/1	Running	0	5m23s

## 镜像更新

deployment支持两种更新策略: **重建更新** 和 **滚动更新**, 可以通过 `strategy` 指定策略类型, 支持两个属性:

`strategy`: 指定新的Pod替换旧的Pod的策略, 支持两个属性:

`type`: 指定策略类型, 支持两种策略

**Recreate**: 在创建出新的Pod之前会先杀掉所有已存在的Pod

**RollingUpdate**: 滚动更新, 就是杀死一部分, 就启动一部分, 在更新过程中, 存在两个版本Pod

`rollingUpdate`: 当type为RollingUpdate时生效, 用于为RollingUpdate设置参数, 支持两个属性:

`maxUnavailable`: 用来指定在升级过程中不可用Pod的最大数量, 默认为25%。

`maxSurge`: 用来指定在升级过程中可以超过期望的Pod的最大数量, 默认为25%。

## 重建更新

1) 编辑pc-deployment.yaml, 在spec节点下添加更新策略

```
spec:
  strategy: # 策略
  type: Recreate # 重建更新
```

2) 创建deploy进行验证

# 变更镜像

```
[root@master ~]# kubectl set image deployment pc-deployment
nginx=nginx:1.17.2 -n dev
deployment.apps/pc-deployment image updated
```

# 观察升级过程

```
[root@master ~]# kubectl get pods -n dev -w
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-5d89bdfbf9-65qcw	1/1	Running	0	31s
pc-deployment-5d89bdfbf9-w5nzv	1/1	Running	0	31s

pc-deployment-5d89bdfbf9-xpt7w	1/1	Running	0	31s
pc-deployment-5d89bdfbf9-xpt7w	1/1	Terminating	0	41s
pc-deployment-5d89bdfbf9-65qcw	1/1	Terminating	0	41s
pc-deployment-5d89bdfbf9-w5nzv	1/1	Terminating	0	41s
pc-deployment-675d469f8b-grn8z	0/1	Pending	0	0s
pc-deployment-675d469f8b-hbl4v	0/1	Pending	0	0s
pc-deployment-675d469f8b-67nz2	0/1	Pending	0	0s
pc-deployment-675d469f8b-grn8z	0/1	ContainerCreating	0	0s
pc-deployment-675d469f8b-hbl4v	0/1	ContainerCreating	0	0s
pc-deployment-675d469f8b-67nz2	0/1	ContainerCreating	0	0s
pc-deployment-675d469f8b-grn8z	1/1	Running	0	1s
pc-deployment-675d469f8b-67nz2	1/1	Running	0	1s
pc-deployment-675d469f8b-hbl4v	1/1	Running	0	2s

## 滚动更新

1) 编辑pc-deployment.yaml,在spec节点下添加更新策略

```
spec:
  strategy: # 策略
    type: RollingUpdate # 滚动更新策略
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
```

2) 创建deploy进行验证

# 变更镜像

```
[root@master ~]# kubectl set image deployment pc-deployment  
nginx=nginx:1.17.3 -n dev
```

deployment.apps/pc-deployment image updated

# 观察升级过程

```
[root@master ~]# kubectl get pods -n dev -w
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-c848d767-8rbzt	1/1	Running	0	31m
pc-deployment-c848d767-h4p68	1/1	Running	0	31m
pc-deployment-c848d767-hlmz4	1/1	Running	0	31m
pc-deployment-c848d767-rrqcn	1/1	Running	0	31m
pc-deployment-966bf7f44-226rx	0/1	Pending		0s
pc-deployment-966bf7f44-226rx	0/1	ContainerCreating		0s
pc-deployment-966bf7f44-226rx	1/1	Running		1s
pc-deployment-c848d767-h4p68	0/1	Terminating		34m
pc-deployment-966bf7f44-cnd44	0/1	Pending		0s
pc-deployment-966bf7f44-cnd44	0/1	ContainerCreating		0s
pc-deployment-966bf7f44-cnd44	1/1	Running		2s
pc-deployment-c848d767-hlmz4	0/1	Terminating		34m
pc-deployment-966bf7f44-px48p	0/1	Pending		0s
pc-deployment-966bf7f44-px48p	0/1	ContainerCreating		0s

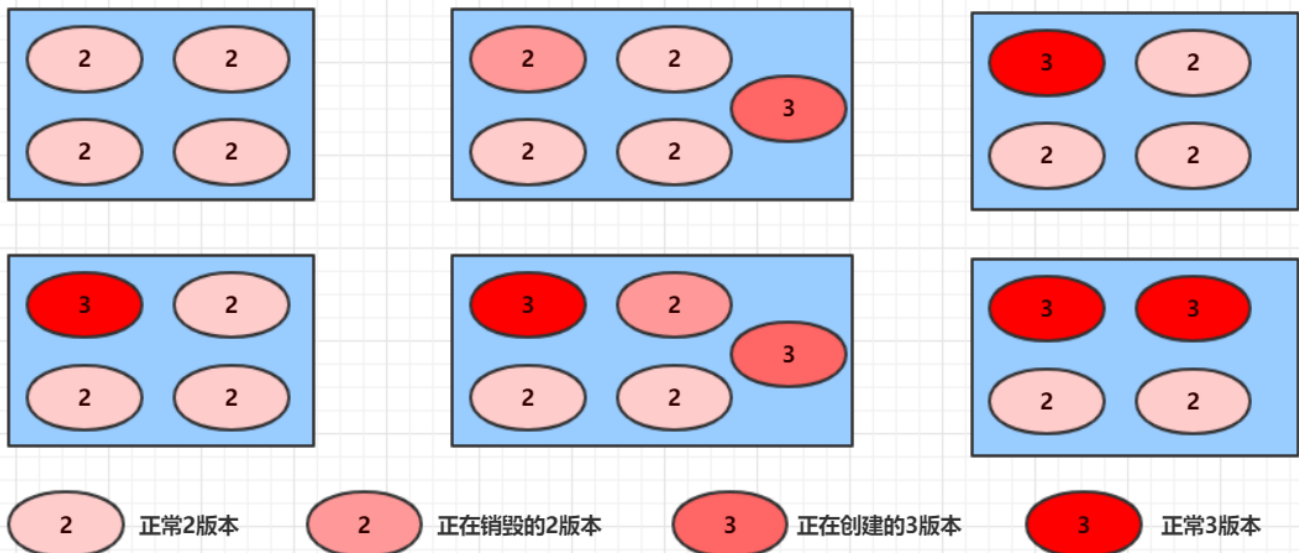
pc-deployment-966bf7f44-px48p 0s	1/1	Running	0
pc-deployment-c848d767-8rbzt 34m	0/1	Terminating	0
pc-deployment-966bf7f44-dkmqp 0s	0/1	Pending	0
pc-deployment-966bf7f44-dkmqp 0s	0/1	ContainerCreating	0
pc-deployment-966bf7f44-dkmqp 2s	1/1	Running	0
pc-deployment-c848d767-rrqcn 34m	0/1	Terminating	0

# 至此，新版本的pod创建完毕，就版本的pod销毁完毕

# 中间过程是滚动进行的，也就是边销毁边创建

滚动更新的过程：

滚动更新：每一轮,销毁一个2版本的Pod，创建一个3版本的Pod



镜像更新中rs的变化:

```
# 查看rs,发现原来的rs的依旧存在, 只是pod数量变为了0, 而后又新产生了一个rs, pod数量为4
```

```
# 其实这就是deployment能够进行版本回退的奥妙所在, 后面会详细解释
```

```
[root@master ~]# kubectl get rs -n dev
```

NAME	DESIRED	CURRENT	READY	AGE
pc-deployment-6696798b78	0	0	0	7m37s
pc-deployment-6696798b11	0	0	0	5m37s
pc-deployment-c848d76789	4	4	4	72s

## 版本回退

deployment支持版本升级过程中的暂停、继续功能以及版本回退等诸多功能, 下面具体来看.

kubectl rollout: 版本升级相关功能, 支持下面的选项:

- status 显示当前升级状态
- history 显示 升级历史记录
- pause 暂停版本升级过程
- resume 继续已经暂停的版本升级过程
- restart 重启版本升级过程
- undo 回滚到上一级版本 (可以使用--to-revision回滚到指定版本)

```
# 查看当前升级版本的状态
```

```
[root@master ~]# kubectl rollout status deploy pc-deployment -n dev  
deployment "pc-deployment" successfully rolled out
```

```
# 查看升级历史记录
```

```
[root@master ~]# kubectl rollout history deploy pc-deployment -n dev  
deployment.apps/pc-deployment
```

```
REVISION  CHANGE-CAUSE
```

```
1          kubectl create --filename=pc-deployment.yaml --record=true  
2          kubectl create --filename=pc-deployment.yaml --record=true  
3          kubectl create --filename=pc-deployment.yaml --record=true
```

```
# 可以发现有三版记录, 说明完成过两次升级
```

# 版本回滚

# 这里直接使用--to-revision=1回滚到了1版本， 如果省略这个选项，就是回退到上个版本，就是2版本

```
[root@master ~]# kubectl rollout undo deployment pc-deployment --to-revision=1 -n dev
```

deployment.apps/pc-deployment rolled back

# 查看发现，通过nginx镜像版本可以发现到了第一版

```
[root@master ~]# kubectl get deploy -n dev -o wide
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
pc-deployment	4/4	4	4	74m	nginx
nginx:1.17.1					

# 查看rs，发现第一个rs中有4个pod运行，后面两个版本的rs中pod为运行

# 其实deployment之所以可以实现版本的回滚，就是通过记录下历史rs来实现的，

# 一旦想回滚到哪个版本，只需要将当前版本pod数量降为0，然后将回滚版本的pod提升为目标数量就可以了

```
[root@master ~]# kubectl get rs -n dev
```

NAME	DESIRED	CURRENT	READY	AGE
pc-deployment-6696798b78	4	4	4	78m
pc-deployment-966bf7f44	0	0	0	37m
pc-deployment-c848d767	0	0	0	71m

## 金丝雀发布

Deployment控制器支持控制更新过程中的控制，如“暂停(pause)”或“继续(resume)”更新操作。

比如有一批新的Pod资源创建完成后立即暂停更新过程，此时，仅存在一部分新版本的应用，主体部分还是旧的版本。然后，再筛选一小部分的用户请求路由到新版本的Pod应用，继续观察能否稳定地按期望的方式运行。确定没问题之后再继续完成余下的Pod资源滚动更新，否则立即回滚更新操作。这就是所谓的金丝雀发布。

# 更新deployment的版本，并配置暂停deployment



```
[root@master ~]# kubectl set image deploy pc-deployment
nginx=nginx:1.17.4 -n dev && kubectl rollout pause deployment pc-
deployment -n dev
deployment.apps/pc-deployment image updated
deployment.apps/pc-deployment paused
```

#观察更新状态

```
[root@master ~]# kubectl rollout status deploy pc-deployment -n dev
Waiting for deployment "pc-deployment" rollout to finish: 2 out of 4 new
replicas have been updated...
```

# 监控更新的过程，可以看到已经新增了一个资源，但是并未按照预期的状态去删除一个旧的资源，就是因为使用了pause暂停命令

```
[root@master ~]# kubectl get rs -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	
pc-deployment-5d89bdfbf9	3	3	3	19m	nginx
CONTAINERS IMAGES					
nginx:1.17.1					
pc-deployment-675d469f8b	0	0	0	14m	nginx
nginx:1.17.2					
pc-deployment-6c9f56fcfb	2	2	2	3m16s	nginx
nginx:1.17.4					

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-5d89bdfbf9-rj8sq	1/1	Running	0	7m33s
pc-deployment-5d89bdfbf9-ttwgg	1/1	Running	0	7m35s
pc-deployment-5d89bdfbf9-v4wvc	1/1	Running	0	7m34s
pc-deployment-6c9f56fcfb-996rt	1/1	Running	0	3m31s
pc-deployment-6c9f56fcfb-j2gtj	1/1	Running	0	3m31s

# 确保更新的pod没问题了，继续更新

```
[root@master ~]# kubectl rollout resume deploy pc-deployment -n dev
deployment.apps/pc-deployment resumed
```

# 查看最后的更新情况

```
[root@master ~]# kubectl get rs -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	
pc-deployment-5d89bdfbf9	0	0	0	21m	nginx
nginx:1.17.1					
pc-deployment-675d469f8b	0	0	0	16m	nginx
nginx:1.17.2					
pc-deployment-6c9f56fcfb	4	4	4	5m11s	nginx
nginx:1.17.4					

```
[root@master ~]# kubectl get pods -n dev
```

NAME	READY	STATUS	RESTARTS	AGE
pc-deployment-6c9f56fcfb-7bfwh	1/1	Running	0	37s
pc-deployment-6c9f56fcfb-996rt	1/1	Running	0	5m27s
pc-deployment-6c9f56fcfb-j2gtj	1/1	Running	0	5m27s
pc-deployment-6c9f56fcfb-rf84v	1/1	Running	0	37s

## 删除Deployment

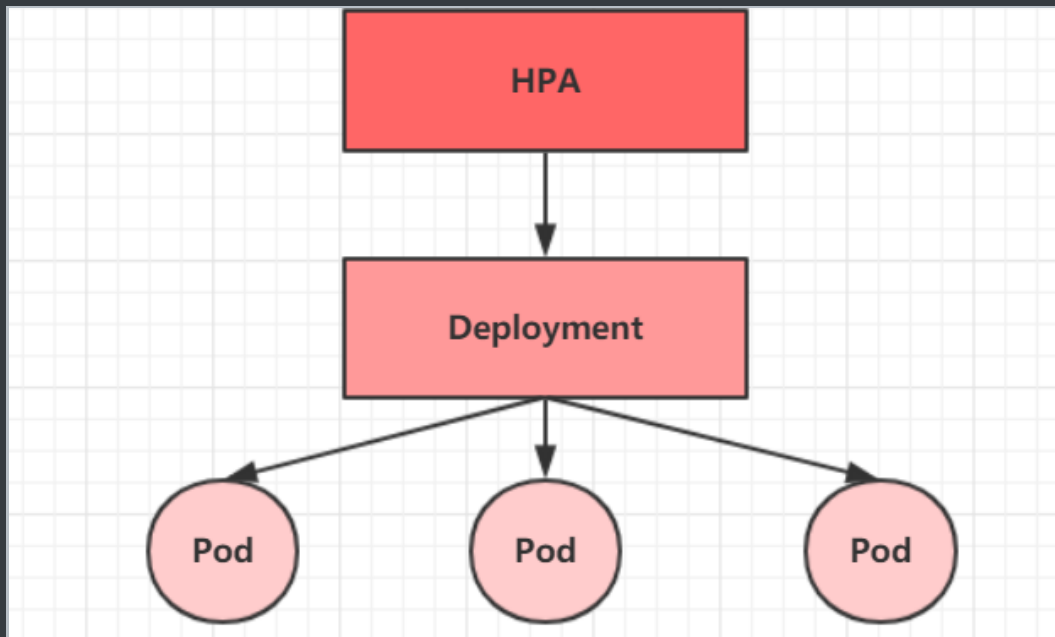
```
# 删除deployment，其下的rs和pod也将被删除
```

```
[root@master ~]# kubectl delete -f pc-deployment.yaml  
deployment.apps "pc-deployment" deleted
```

## Horizontal Pod Autoscaler(HPA)

在前面的课程中，我们已经可以实现通过手工执行 `kubectl scale` 命令实现Pod扩容或缩容，但是这显然不符合Kubernetes的定位目标--自动化、智能化。Kubernetes期望可以实现通过监测Pod的使用情况，实现pod数量的自动调整，于是就产生了Horizontal Pod Autoscaler（HPA）这种控制器。

HPA可以获取每个Pod利用率，然后和HPA中定义的指标进行对比，同时计算出需要伸缩的具体值，最后实现Pod的数量的调整。其实HPA与之前的Deployment一样，也属于一种Kubernetes资源对象，它通过追踪分析RC控制的所有目标Pod的负载变化情况，来确定是否需要针对性地调整目标Pod的副本数，这是HPA的实现原理。



接下来，我们来做一个实验

## 1 安装metrics-server

metrics-server可以用来收集集群中的资源使用情况

```
# 安装git
[root@master ~]# yum install git -y
# 获取metrics-server，注意使用的版本
[root@master ~]# git clone -b v0.3.6 https://github.com/kubernetes-incubator/metrics-server
# 修改deployment，注意修改的是镜像和初始化参数
[root@master ~]# cd /root/metrics-server/deploy/1.8+/
[root@master 1.8+]# vim metrics-server-deployment.yaml
按图中添加下面选项
hostNetwork: true
image: registry.cn-hangzhou.aliyuncs.com/google_containers/metrics-server-amd64:v0.3.6
args:
- --kubelet-insecure-tls
- --kubelet-preferred-address-types=InternalIP,Hostname,InternalDNS,ExternalDNS,ExternalIP
```

```

hostNetwork: true
serviceAccountName: metrics-server
volumes:
# mount in tmp so we can safely use from-scratch images and/or read-only containers
- name: tmp-dir
  emptyDir: {}
containers:
- name: metrics-server
  image: registry.cn-hangzhou.aliyuncs.com/google_containers/metrics-server-amd64:v0.3.6
  imagePullPolicy: Always
  args:
- --kubelet-insecure-tls
- --kubelet-preferred-address-types=InternalIP,Hostname,InternalDNS,ExternalDNS,ExternalIP
volumeMounts:

```

# 安装metrics-server

```
[root@master 1.8+]# kubectl apply -f ./
```

# 查看pod运行情况

```
[root@master 1.8+]# kubectl get pod -n kube-system
```

```
metrics-server-6b976979db-2xwbj    1/1    Running    0    90s
```

# 使用kubectl top node 查看资源使用情况

```
[root@master 1.8+]# kubectl top node
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
master	98m	4%	1067Mi	62%
node1	27m	1%	727Mi	42%
node2	34m	1%	800Mi	46%

```
[root@master 1.8+]# kubectl top pod -n kube-system
```

NAME	CPU(cores)	MEMORY(bytes)
coredns-6955765f44-7ptsb	3m	9Mi
coredns-6955765f44-vcwr5	3m	8Mi
etcd-master	14m	145Mi
...		

# 至此,metrics-server安装完成

## 2 准备deployment和service

为了操作简单,直接使用命令

# 创建deployment

```
[root@master 1.8+]# kubectl run nginx --image=nginx:latest --
requests=cpu=100m -n dev
# 创建service
[root@master 1.8+]# kubectl expose deployment nginx --type=NodePort --
port=80 -n dev

# 查看
[root@master 1.8+]# kubectl get deployment,pod,svc -n dev
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/nginx	1/1	1	1	47s

NAME	READY	STATUS	RESTARTS	AGE
pod/nginx-7df9756ccc-bh8dr	1/1	Running	0	47s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
service/nginx	NodePort	10.109.57.248	<none>	80:31136/TCP
AGE				35s

### 3 部署HPA

创建pc-hpa.yaml

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: pc-hpa
  namespace: dev
spec:
  minReplicas: 1 #最小pod数量
  maxReplicas: 10 #最大pod数量
  targetCPUUtilizationPercentage: 3 # CPU使用率指标
  scaleTargetRef: # 指定要控制的nginx信息
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
```

#### # 创建hpa

```
[root@master 1.8+]# kubectl create -f pc-hpa.yaml
horizontalpodautoscaler.autoscaling/pc-hpa created
```

#### # 查看hpa

```
[root@master 1.8+]# kubectl get hpa -n dev
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
pc-hpa	Deployment/nginx	0%/3%	1	10	1	62s

## 4 测试

使用压测工具对service地址 192.168.109.100:31136 进行压测，然后通过控制台查看hpa和pod的变化

hpa变化

```
[root@master ~]# kubectl get hpa -n dev -w
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
pc-hpa 4m11s	Deployment/nginx	0%/3%	1	10	1	
pc-hpa 5m19s	Deployment/nginx	0%/3%	1	10	1	
pc-hpa 6m50s	Deployment/nginx	22%/3%	1	10	1	
pc-hpa 7m5s	Deployment/nginx	22%/3%	1	10	4	
pc-hpa 7m21s	Deployment/nginx	22%/3%	1	10	8	
pc-hpa 7m51s	Deployment/nginx	6%/3%	1	10	8	
pc-hpa 9m6s	Deployment/nginx	0%/3%	1	10	8	
pc-hpa	Deployment/nginx	0%/3%	1	10	8	13m
pc-hpa	Deployment/nginx	0%/3%	1	10	1	14m

deployment变化

```
[root@master ~]# kubectl get deployment -n dev -w
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	11m
nginx	1/4	1	1	13m
nginx	1/4	1	1	13m
nginx	1/4	1	1	13m
nginx	1/4	4	1	13m
nginx	1/8	4	1	14m
nginx	1/8	4	1	14m
nginx	1/8	4	1	14m
nginx	1/8	8	1	14m
nginx	2/8	8	2	14m
nginx	3/8	8	3	14m
nginx	4/8	8	4	14m

nginx	5/8	8	5	14m
nginx	6/8	8	6	14m
nginx	7/8	8	7	14m
nginx	8/8	8	8	15m
nginx	8/1	8	8	20m
nginx	8/1	8	8	20m
nginx	1/1	1	1	20m

pod变化

```
[root@master ~]# kubectl get pods -n dev -w
```

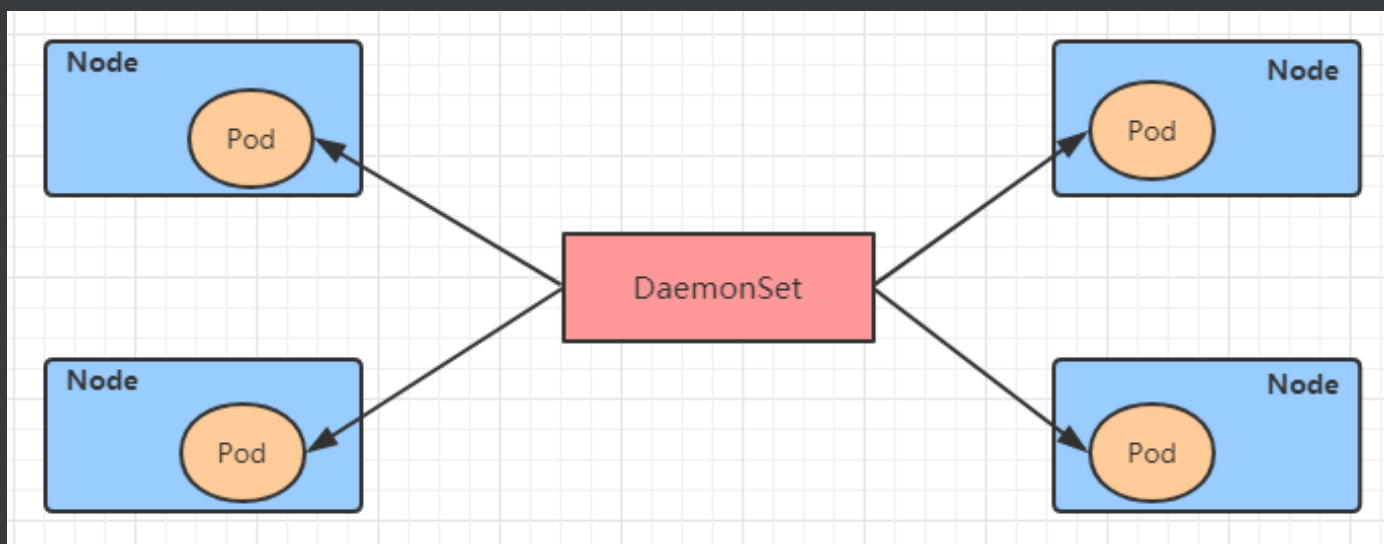
NAME	READY	STATUS	RESTARTS	AGE
nginx-7df9756ccc-bh8dr	1/1	Running	0	11m
nginx-7df9756ccc-cpgrv	0/1	Pending	0	0s
nginx-7df9756ccc-8zhwk	0/1	Pending	0	0s
nginx-7df9756ccc-rr9bn	0/1	Pending	0	0s
nginx-7df9756ccc-cpgrv	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-8zhwk	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-rr9bn	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-m9gsj	0/1	Pending	0	0s
nginx-7df9756ccc-g56qb	0/1	Pending	0	0s
nginx-7df9756ccc-sl9c6	0/1	Pending	0	0s
nginx-7df9756ccc-fgst7	0/1	Pending	0	0s
nginx-7df9756ccc-g56qb	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-m9gsj	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-sl9c6	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-fgst7	0/1	ContainerCreating	0	0s
nginx-7df9756ccc-8zhwk	1/1	Running	0	19s
nginx-7df9756ccc-rr9bn	1/1	Running	0	30s
nginx-7df9756ccc-m9gsj	1/1	Running	0	21s
nginx-7df9756ccc-cpgrv	1/1	Running	0	47s
nginx-7df9756ccc-sl9c6	1/1	Running	0	33s
nginx-7df9756ccc-g56qb	1/1	Running	0	48s
nginx-7df9756ccc-fgst7	1/1	Running	0	66s
nginx-7df9756ccc-fgst7	1/1	Terminating	0	6m50s



nginx-7df9756ccc-8zhwk	1/1	Terminating	0	7m5s
nginx-7df9756ccc-cpgrv	1/1	Terminating	0	7m5s
nginx-7df9756ccc-g56qb	1/1	Terminating	0	6m50s
nginx-7df9756ccc-rr9bn	1/1	Terminating	0	7m5s
nginx-7df9756ccc-m9gsj	1/1	Terminating	0	6m50s
nginx-7df9756ccc-sl9c6	1/1	Terminating	0	6m50s

## DaemonSet(DS)

DaemonSet类型的控制器可以保证在集群中的每一台（或指定）节点上都运行一个副本。一般适用于日志收集、节点监控等场景。也就是说，如果一个Pod提供的功能是节点级别的（每个节点都需要且只需要一个），那么这类Pod就适合使用DaemonSet类型的控制器创建。



DaemonSet控制器的特点：

- 每当向集群中添加一个节点时，指定的 Pod 副本也将添加到该节点上
- 当节点从集群中移除时，Pod 也就被垃圾回收了

下面先来看下DaemonSet的资源清单文件

```
apiVersion: apps/v1 # 版本号
kind: DaemonSet # 类型
metadata: # 元数据
  name: # rs名称
```

```

namespace: # 所属命名空间
labels: # 标签
  controller: daemonset
spec: # 详情描述
  revisionHistoryLimit: 3 # 保留历史版本
  updateStrategy: # 更新策略
    type: RollingUpdate # 滚动更新策略
    rollingUpdate: # 滚动更新
      maxUnavailable: 1 # 最大不可用状态的 Pod 的最大值，可以为百分比，也可以为
      整数
  selector: # 选择器，通过它指定该控制器管理哪些pod
    matchLabels: # Labels匹配规则
      app: nginx-pod
    matchExpressions: # Expressions匹配规则
      - {key: app, operator: In, values: [nginx-pod]}
  template: # 模板，当副本数量不足时，会根据下面的模板创建pod副本
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.17.1
          ports:
            - containerPort: 80

```

创建pc-daemonset.yaml，内容如下：

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: pc-daemonset
  namespace: dev
spec:
  selector:
    matchLabels:

```

```
    app: nginx-pod
template:
  metadata:
    labels:
      app: nginx-pod
  spec:
    containers:
      - name: nginx
        image: nginx:1.17.1
```

#### # 创建daemonset

```
[root@master ~]# kubectl create -f pc-daemonset.yaml
daemonset.apps/pc-daemonset created
```

#### # 查看daemonset

```
[root@master ~]# kubectl get ds -n dev -o wide
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	AGE	
pc-daemonset	2	2	2	2	2	24s	nginx
							nginx:1.17.1

#### # 查看pod,发现在每个Node上都运行一个pod

```
[root@master ~]# kubectl get pods -n dev -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
pc-daemonset-9bck8	1/1	Running	0	37s	10.244.1.43
node1					
pc-daemonset-k224w	1/1	Running	0	37s	10.244.2.74
node2					

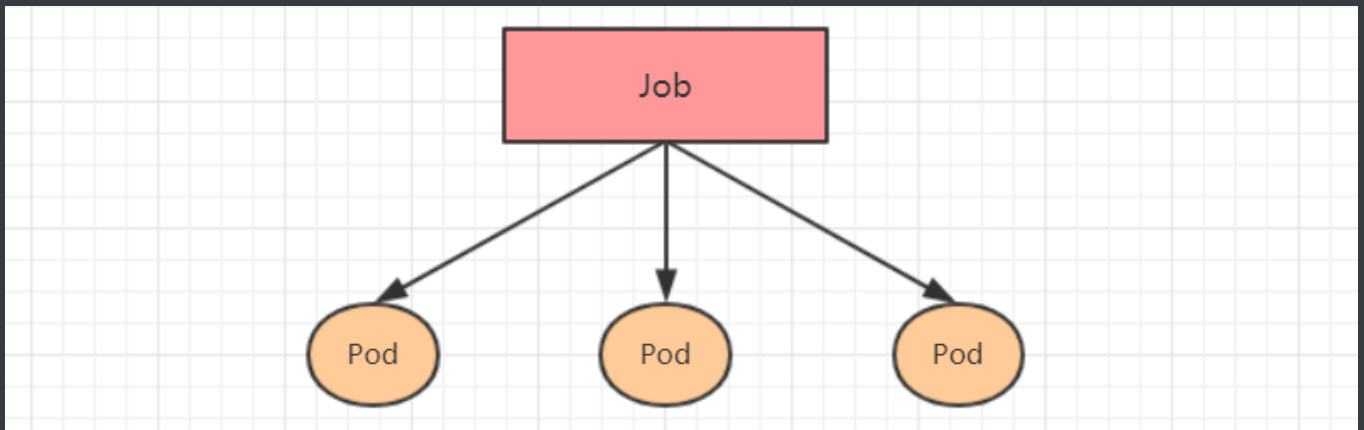
#### # 删除daemonset

```
[root@master ~]# kubectl delete -f pc-daemonset.yaml
daemonset.apps "pc-daemonset" deleted
```

# Job

Job，主要用于负责批量处理(一次要处理指定数量任务)短暂的一次性(每个任务仅运行一次就结束)任务。Job特点如下：

- 当Job创建的pod执行成功结束时，Job将记录成功结束的pod数量
- 当成功结束的pod达到指定的数量时，Job将完成执行



Job的资源清单文件：

```
apiVersion: batch/v1 # 版本号
kind: Job # 类型
metadata: # 元数据
  name: # rs名称
  namespace: # 所属命名空间
  labels: #标签
    controller: job
spec: # 详情描述
  completions: 1 # 指定job需要成功运行Pods的次数。默认值：1
  parallelism: 1 # 指定job在任一时刻应该并发运行Pods的数量。默认值：1
  activeDeadlineSeconds: 30 # 指定job可运行的时间期限，超过时间还未结束，系统将会尝试进行终止。
  backoffLimit: 6 # 指定job失败后进行重试的次数。默认是6
  manualSelector: true # 是否可以使用selector选择器选择pod，默认是false
  selector: # 选择器，通过它指定该控制器管理哪些pod
    matchLabels: # Labels匹配规则
      app: counter-pod
```

```

matchExpressions: # Expressions匹配规则
  - {key: app, operator: In, values: [counter-pod]}
template: # 模板, 当副本数量不足时, 会根据下面的模板创建pod副本
  metadata:
    labels:
      app: counter-pod
  spec:
    restartPolicy: Never # 重启策略只能设置为Never或者OnFailure
    containers:
      - name: counter
        image: busybox:1.30
        command: ["bin/sh", "-c", "for i in 9 8 7 6 5 4 3 2 1; do echo $i;sleep 2;done"]

```

关于重启策略设置的说明:

如果指定为OnFailure, 则job会在pod出现故障时重启容器, 而不是创建pod, failed次数不变

如果指定为Never, 则job会在pod出现故障时创建新的pod, 并且故障pod不会消失, 也不会重启, failed次数加1

如果指定为Always的话, 就意味着一直重启, 意味着job任务会重复去执行了, 当然不对, 所以不能设置为Always

创建pc-job.yaml, 内容如下:

```

apiVersion: batch/v1
kind: Job
metadata:
  name: pc-job
  namespace: dev
spec:
  manualSelector: true
  selector:
    matchLabels:
      app: counter-pod
  template:

```

```

metadata:
  labels:
    app: counter-pod
spec:
  restartPolicy: Never
  containers:
  - name: counter
    image: busybox:1.30
    command: ["bin/sh", "-c", "for i in 9 8 7 6 5 4 3 2 1; do echo
$i;sleep 3;done"]

```

### # 创建job

```

[root@master ~]# kubectl create -f pc-job.yaml
job.batch/pc-job created

```

### # 查看job

```

[root@master ~]# kubectl get job -n dev -o wide -w
NAME          COMPLETIONS  DURATION  AGE  CONTAINERS  IMAGES
SELECTOR
pc-job        0/1          21s       21s  counter     busybox:1.30
app=counter-pod
pc-job        1/1          31s       79s  counter     busybox:1.30
app=counter-pod

```

# 通过观察pod状态可以看到，pod在运行完毕任务后，就会变成Completed状态

```

[root@master ~]# kubectl get pods -n dev -w
NAME          READY  STATUS    RESTARTS  AGE
pc-job-rxg96  1/1    Running   0          29s
pc-job-rxg96  0/1    Completed 0          33s

```

# 接下来，调整下pod运行的总数量和并行数量 即：在spec下设置下面两个选项

# completions: 6 # 指定job需要成功运行Pods的次数为6

# parallelism: 3 # 指定job并发运行Pods的数量为3

# 然后重新运行job，观察效果，此时会发现，job会每次运行3个pod，总共执行了6个pod

```

[root@master ~]# kubectl get pods -n dev -w
NAME          READY  STATUS    RESTARTS  AGE

```

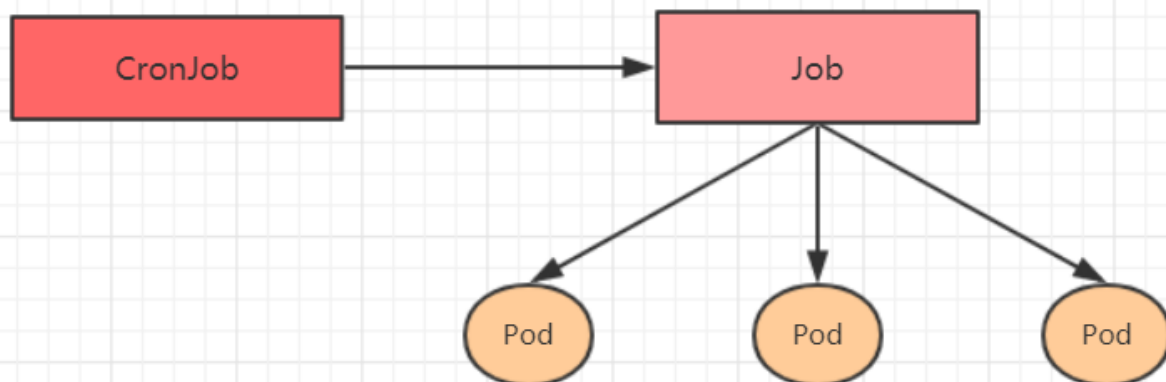
pc-job-684ft	1/1	Running	0	5s
pc-job-jhj49	1/1	Running	0	5s
pc-job-pfcvh	1/1	Running	0	5s
pc-job-684ft	0/1	Completed	0	11s
pc-job-v7rhr	0/1	Pending	0	0s
pc-job-v7rhr	0/1	Pending	0	0s
pc-job-v7rhr	0/1	ContainerCreating	0	0s
pc-job-jhj49	0/1	Completed	0	11s
pc-job-fhwf7	0/1	Pending	0	0s
pc-job-fhwf7	0/1	Pending	0	0s
pc-job-pfcvh	0/1	Completed	0	11s
pc-job-5vg2j	0/1	Pending	0	0s
pc-job-fhwf7	0/1	ContainerCreating	0	0s
pc-job-5vg2j	0/1	Pending	0	0s
pc-job-5vg2j	0/1	ContainerCreating	0	0s
pc-job-fhwf7	1/1	Running	0	2s
pc-job-v7rhr	1/1	Running	0	2s
pc-job-5vg2j	1/1	Running	0	3s
pc-job-fhwf7	0/1	Completed	0	12s
pc-job-v7rhr	0/1	Completed	0	12s
pc-job-5vg2j	0/1	Completed	0	12s

# 删除job

```
[root@master ~]# kubectl delete -f pc-job.yaml
job.batch "pc-job" deleted
```

## CronJob(CJ)

CronJob控制器以Job控制器资源为其管控对象，并借助它管理pod资源对象，Job控制器定义的作业任务在其控制器资源创建之后便会立即执行，但CronJob可以以类似于Linux操作系统的周期性任务作业计划的方式控制其运行**时间点**及**重复运行**的方式。也就是说，**CronJob可以在特定的时间点(反复的)去运行job任务。**



CronJob的资源清单文件:

```
apiVersion: batch/v1beta1 # 版本号
kind: CronJob # 类型
metadata: # 元数据
  name: # rs名称
  namespace: # 所属命名空间
  labels: # 标签
  controller: cronjob
spec: # 详情描述
  schedule: # cron格式的作业调度运行时间点,用于控制任务在什么时间执行
  concurrencyPolicy: # 并发执行策略,用于定义前一次作业运行尚未完成时是否以及如何
    运行后一次的作业
  failedJobHistoryLimit: # 为失败的任务执行保留的历史记录数,默认为1
  successfulJobHistoryLimit: # 为成功的任务执行保留的历史记录数,默认为3
  startingDeadlineSeconds: # 启动作业错误的超时时长
  jobTemplate: # job控制器模板,用于为cronjob控制器生成job对象;下面其实就是job的
    定义
    metadata:
    spec:
      completions: 1
      parallelism: 1
      activeDeadlineSeconds: 30
      backoffLimit: 6
      manualSelector: true
      selector:
```



```

matchLabels:
  app: counter-pod
matchExpressions: 规则
  - {key: app, operator: In, values: [counter-pod]}
template:
  metadata:
    labels:
      app: counter-pod
  spec:
    restartPolicy: Never
    containers:
      - name: counter
        image: busybox:1.30
        command: ["bin/sh","-c","for i in 9 8 7 6 5 4 3 2 1; do echo
$i;sleep 20;done"]

```

需要重点解释的几个选项:

schedule: cron表达式, 用于指定任务的执行时间

```

*/1 * * * *
<分钟> <小时> <日> <月份> <星期>

```

分钟 值从 0 到 59.

小时 值从 0 到 23.

日 值从 1 到 31.

月 值从 1 到 12.

星期 值从 0 到 6, 0 代表星期日

多个时间可以用逗号隔开; 范围可以用连字符给出; \*可以作为通配符; /表示每...

concurrencyPolicy:

Allow: 允许Jobs并发运行(默认)

Forbid: 禁止并发运行, 如果上一次运行尚未完成, 则跳过下一次运行

Replace: 替换, 取消当前正在运行的作业并用新作业替换它

创建pc-cronjob.yaml, 内容如下:

```

apiVersion: batch/v1beta1

```

```

kind: CronJob
metadata:
  name: pc-cronjob
  namespace: dev
  labels:
    controller: cronjob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    metadata:
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: counter
              image: busybox:1.30
              command: ["bin/sh", "-c", "for i in 9 8 7 6 5 4 3 2 1; do echo
$i;sleep 3;done"]

```

#### # 创建cronjob

```

[root@master ~]# kubectl create -f pc-cronjob.yaml
cronjob.batch/pc-cronjob created

```

#### # 查看cronjob

```

[root@master ~]# kubectl get cronjobs -n dev

```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
pc-cronjob	*/1 * * * *	False	0	<none>	6s

#### # 查看job

```

[root@master ~]# kubectl get jobs -n dev

```

NAME	COMPLETIONS	DURATION	AGE
pc-cronjob-1592587800	1/1	28s	3m26s
pc-cronjob-1592587860	1/1	28s	2m26s
pc-cronjob-1592587920	1/1	28s	86s

```
# 查看pod
```

```
[root@master ~]# kubectl get pods -n dev
```

pc-cronjob-1592587800-x4tsm	0/1	Completed	0	2m24s
pc-cronjob-1592587860-r5gv4	0/1	Completed	0	84s
pc-cronjob-1592587920-9dxxq	1/1	Running	0	24s

```
# 删除cronjob
```

```
[root@master ~]# kubectl delete -f pc-cronjob.yaml  
cronjob.batch "pc-cronjob" deleted
```

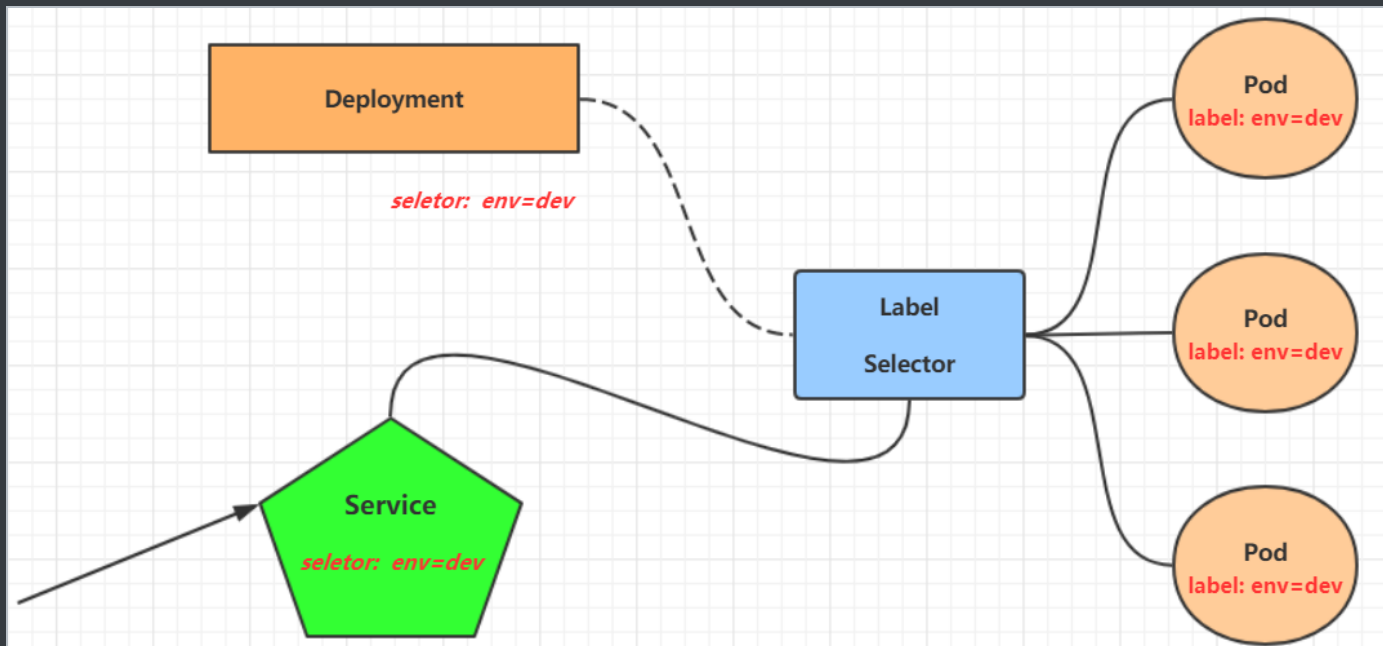
## 第七章 Service详解

本章节主要介绍kubernetes的流量负载组件：Service和Ingress。

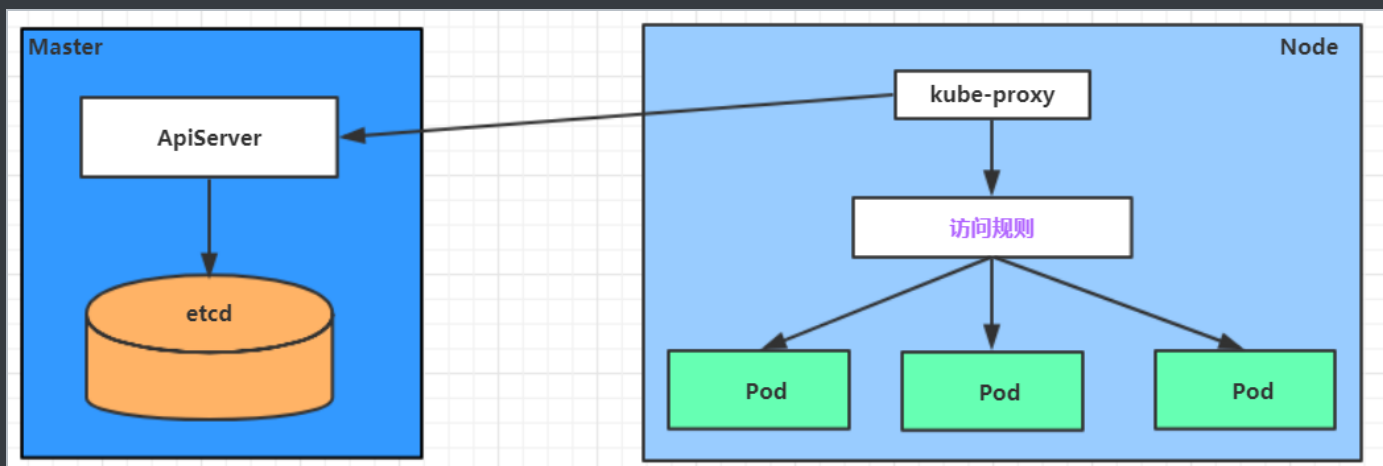
### Service介绍

在kubernetes中，pod是应用程序的载体，我们可以通过pod的ip来访问应用程序，但是pod的ip地址不是固定的，这也就意味着不方便直接采用pod的ip对服务进行访问。

为了解决这个问题，kubernetes提供了Service资源，**Service会对提供同一个服务的多个pod进行聚合，并且提供一个统一的入口地址**。通过访问Service的入口地址就能访问到后面的pod服务。



Service在很多情况下只是一个概念，真正起作用的其实是kube-proxy服务进程，每个Node节点上都运行着一个kube-proxy服务进程。当创建Service的时候会通过api-server向etcd写入创建的service的信息，而kube-proxy会基于监听的机制发现这种Service的变动，然后它会将最新的Service信息转换成对应的访问规则。



```
# 10.97.97.97:80 是service提供的访问入口
# 当访问这个入口的时候，可以发现后面有三个pod的服务在等待调用，
# kube-proxy会基于rr（轮询）的策略，将请求分发到其中一个pod上去
# 这个规则会同时在集群内的所有节点上都生成，所以在任何一个节点上访问都可以。

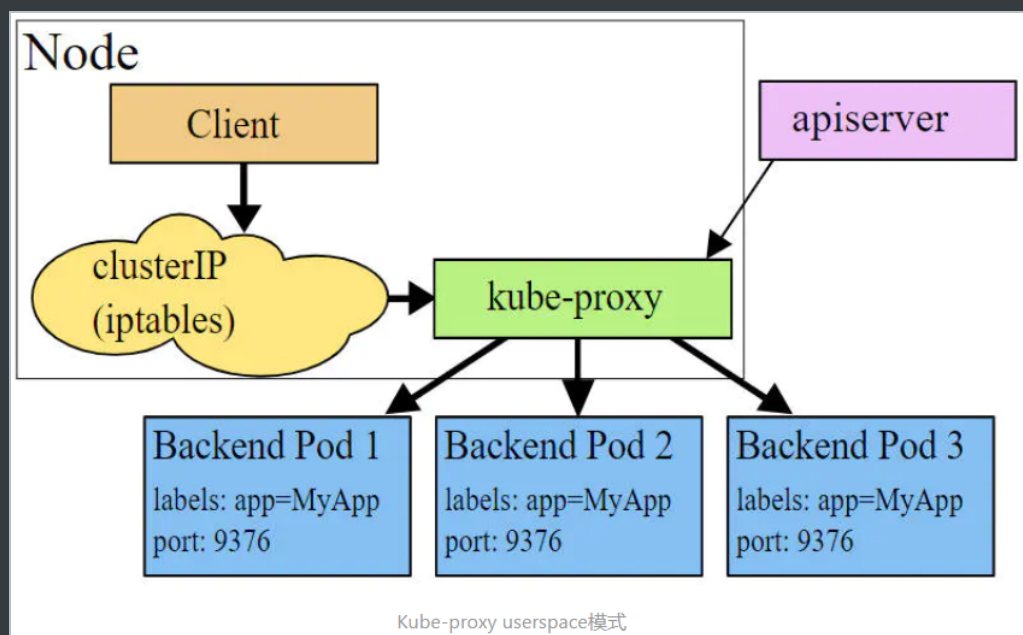
[root@node1 ~]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.97.97.97:80 rr
  -> 10.244.1.39:80               Masq    1      0      0
  -> 10.244.1.40:80               Masq    1      0      0
  -> 10.244.2.33:80               Masq    1      0      0
```

kube-proxy目前支持三种工作模式:

### userspace 模式

userspace模式下，kube-proxy会为每一个Service创建一个监听端口，发向Cluster IP的请求被Iptables规则重定向到kube-proxy监听的端口上，kube-proxy根据LB算法选择一个提供服务的Pod并和其建立链接，以将请求转发到Pod上。

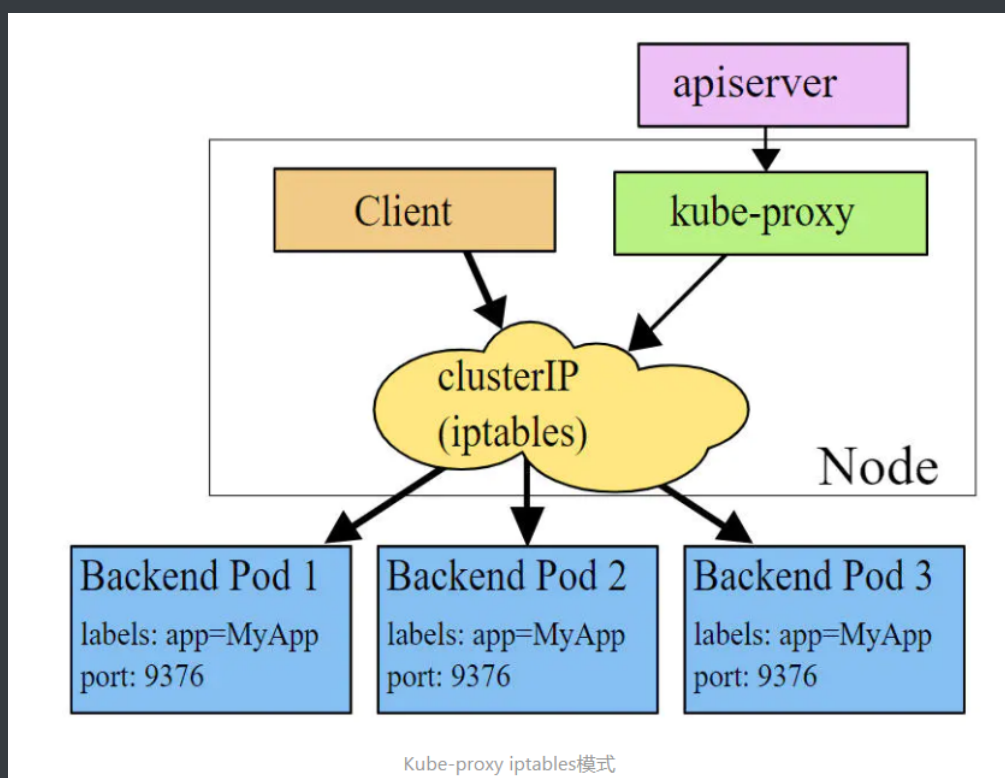
该模式下，**kube-proxy充当了一个四层负责均衡器的角色**。由于kube-proxy运行在userspace中，在进行转发处理时会增加内核和用户空间之间的数据拷贝，虽然比较稳定，但是效率比较低。



## iptables 模式

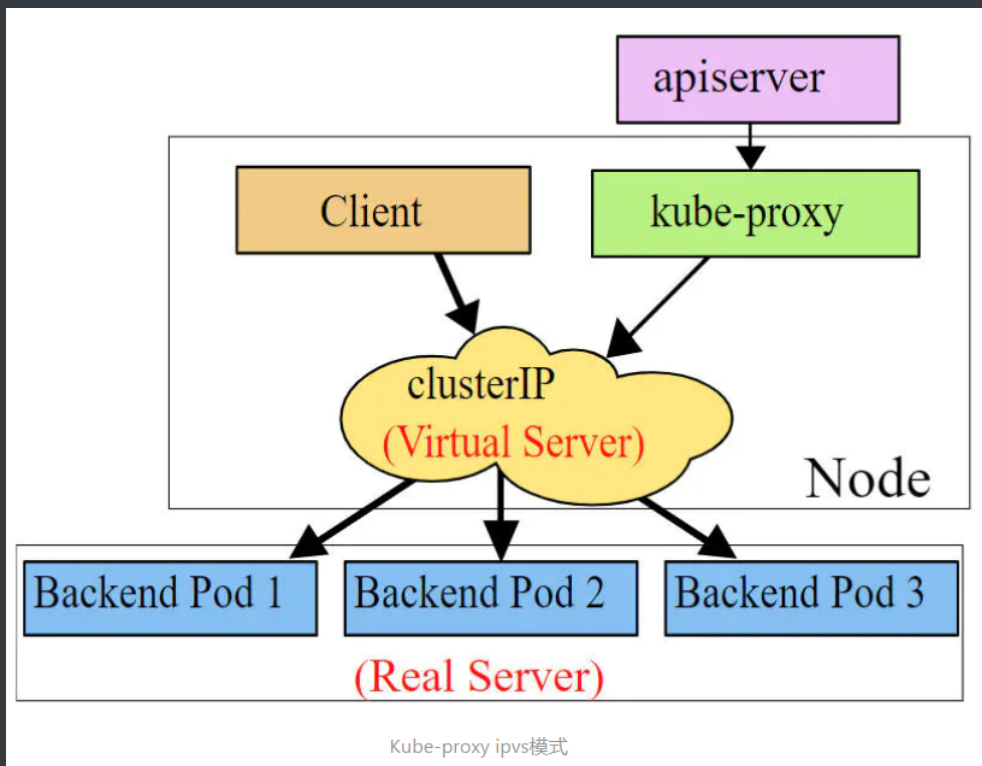
iptables模式下，kube-proxy为service后端的每个Pod创建对应的iptables规则，直接将发向Cluster IP的请求重定向到一个Pod IP。

该模式下kube-proxy不承担四层负责均衡器的角色，只负责创建iptables规则。该模式的优点是较userspace模式效率更高，但不能提供灵活的LB策略，当后端Pod不可用时也无法进行重试。



## ipvs 模式

ipvs模式和iptables类似，kube-proxy监控Pod的变化并创建相应的ipvs规则。ipvs相对iptables转发效率更高。除此以外，ipvs支持更多的LB算法。



```
# 此模式必须安装ipvs内核模块，否则会降级为iptables
# 开启ipvs
[root@master ~]# kubectl edit cm kube-proxy -n kube-system
[root@master ~]# kubectl delete pod -l k8s-app=kube-proxy -n kube-system
[root@node1 ~]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port          Forward Weight ActiveConn InActConn
TCP  10.97.97.97:80 rr
  -> 10.244.1.39:80               Masq    1      0      0
  -> 10.244.1.40:80               Masq    1      0      0
  -> 10.244.2.33:80               Masq    1      0      0
```

## Service类型

Service的资源清单文件：

```
kind: Service # 资源类型
apiVersion: v1 # 资源版本
metadata: # 元数据
```

```
name: service # 资源名称
namespace: dev # 命名空间
spec: # 描述
  selector: # 标签选择器，用于确定当前service代理哪些pod
    app: nginx
  type: # Service类型，指定service的访问方式
  clusterIP: # 虚拟服务的ip地址
  sessionAffinity: # session亲和性，支持ClientIP、None两个选项
  ports: # 端口信息
    - protocol: TCP
      port: 3017 # service端口
      targetPort: 5003 # pod端口
      nodePort: 31122 # 主机端口
```

- ClusterIP: 默认值，它是Kubernetes系统自动分配的虚拟IP，只能在集群内部访问
- NodePort: 将Service通过指定的Node上的端口暴露给外部，通过此方法，就可以在集群外部访问服务
- LoadBalancer: 使用外接负载均衡器完成到服务的负载分发，注意此模式需要外部云环境支持
- ExternalName: 把集群外部的服务引入集群内部，直接使用

## Service使用

### 实验环境准备

在使用service之前，首先利用Deployment创建出3个pod，注意要为pod设置 `app=nginx-pod` 的标签

创建deployment.yaml，内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pc-deployment
  namespace: dev
spec:
```



```
replicas: 3
selector:
  matchLabels:
    app: nginx-pod
template:
  metadata:
    labels:
      app: nginx-pod
  spec:
    containers:
      - name: nginx
        image: nginx:1.17.1
        ports:
          - containerPort: 80
```

```
[root@master ~]# kubectl create -f deployment.yaml
deployment.apps/pc-deployment created
```

# 查看pod详情

```
[root@master ~]# kubectl get pods -n dev -o wide --show-labels
```

NAME	READY	STATUS	IP	NODE
pc-deployment-66cb59b984-8p84h	1/1	Running	10.244.1.40	node1
app=nginx-pod				
pc-deployment-66cb59b984-vx8vx	1/1	Running	10.244.2.33	node2
app=nginx-pod				
pc-deployment-66cb59b984-wnncx	1/1	Running	10.244.1.39	node1
app=nginx-pod				

# 为了方便后面的测试，修改下三台nginx的index.html页面（三台修改的IP地址不一致）

```
# kubectl exec -it pc-deployment-66cb59b984-8p84h -n dev /bin/sh
```

```
# echo "10.244.1.40" > /usr/share/nginx/html/index.html
```

#修改完毕之后，访问测试

```
[root@master ~]# curl 10.244.1.40
```

```
10.244.1.40
```

```
[root@master ~]# curl 10.244.2.33
10.244.2.33
[root@master ~]# curl 10.244.1.39
10.244.1.39
```

## ClusterIP类型的Service

创建service-clusterip.yaml文件

```
apiVersion: v1
kind: Service
metadata:
  name: service-clusterip
  namespace: dev
spec:
  selector:
    app: nginx-pod
  clusterIP: 10.97.97.97 # service的ip地址, 如果不写, 默认会生成一个
  type: ClusterIP
  ports:
    - port: 80 # Service端口
      targetPort: 80 # pod端口
```

# 创建service

```
[root@master ~]# kubectl create -f service-clusterip.yaml
service/service-clusterip created
```

# 查看service

```
[root@master ~]# kubectl get svc -n dev -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE	SELECTOR			
service-clusterip	ClusterIP	10.97.97.97	<none>	80/TCP
13s	app=nginx-pod			

# 查看service的详细信息

```
# 在这里有一个Endpoints列表，里面就是当前service可以负载到的服务入口
[root@master ~]# kubectl describe svc service-clusterip -n dev
Name:                service-clusterip
Namespace:           dev
Labels:              <none>
Annotations:         <none>
Selector:            app=nginx-pod
Type:                ClusterIP
IP:                  10.97.97.97
Port:                <unset> 80/TCP
TargetPort:          80/TCP
Endpoints:           10.244.1.39:80,10.244.1.40:80,10.244.2.33:80
Session Affinity:    None
Events:              <none>
```

```
# 查看ipvs的映射规则
```

```
[root@master ~]# ipvsadm -Ln
```

```
TCP 10.97.97.97:80 rr
```

-> 10.244.1.39:80	Masq	1	0	0
-> 10.244.1.40:80	Masq	1	0	0
-> 10.244.2.33:80	Masq	1	0	0

```
# 访问10.97.97.97:80观察效果
```

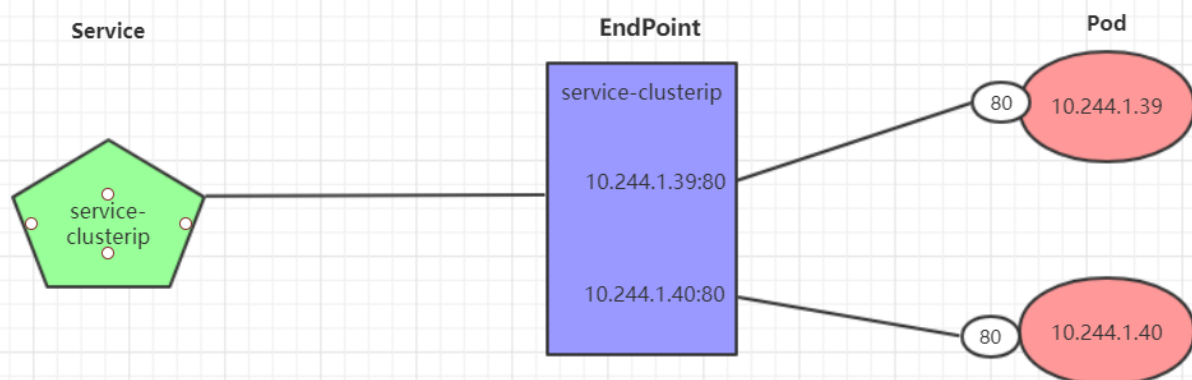
```
[root@master ~]# curl 10.97.97.97:80
```

```
10.244.2.33
```

## Endpoint

Endpoint是kubernetes中的一个资源对象，存储在etcd中，用来记录一个service对应的所有pod的访问地址，它是根据service配置文件中selector描述产生的。

一个Service由一组Pod组成，这些Pod通过Endpoints暴露出来，**Endpoints是实现实际服务的端点集合**。换句话说，service和pod之间的联系是通过endpoints实现的。



```
[root@master ~]# kubectl get endpoints -n dev -o wide
```

NAME	ENDPOINTS	AGE
service-clusterip	10.244.1.39:80,10.244.1.40:80,10.244.2.33:80	50m

## 负载分发策略

对Service的访问被分发到了后端的Pod上去，目前kubernetes提供了两种负载分发策略：

- 如果不定义，默认使用kube-proxy的策略，比如随机、轮询
- 基于客户端地址的会话保持模式，即来自同一个客户端发起的所有请求都会转发到固定的一个Pod上

此模式可以在spec中添加 `sessionAffinity:ClientIP` 选项

# 查看ipvs的映射规则【rr 轮询】

```
[root@master ~]# ipvsadm -Ln
```

```
TCP 10.97.97.97:80 rr
```

```
-> 10.244.1.39:80      Masq    1      0      0
-> 10.244.1.40:80      Masq    1      0      0
-> 10.244.2.33:80      Masq    1      0      0
```

# 循环访问测试

```
[root@master ~]# while true;do curl 10.97.97.97:80; sleep 5; done;
```

```
10.244.1.40
10.244.1.39
10.244.2.33
10.244.1.40
10.244.1.39
10.244.2.33
```

```
# 修改分发策略----sessionAffinity:ClientIP
```

```
# 查看ipvs规则【persistent 代表持久】
```

```
[root@master ~]# ipvsadm -Ln
```

```
TCP 10.97.97.97:80 rr persistent 10800
```

```
-> 10.244.1.39:80 Masq 1 0 0
```

```
-> 10.244.1.40:80 Masq 1 0 0
```

```
-> 10.244.2.33:80 Masq 1 0 0
```

```
# 循环访问测试
```

```
[root@master ~]# while true;do curl 10.97.97.97; sleep 5; done;
```

```
10.244.2.33
```

```
10.244.2.33
```

```
10.244.2.33
```

```
# 删除service
```

```
[root@master ~]# kubectl delete -f service-clusterip.yaml
```

```
service "service-clusterip" deleted
```

## HeadLiness类型的Service

在某些场景中，开发人员可能不想使用Service提供的负载均衡功能，而希望自己来控制负载均衡策略，针对这种情况，kubernetes提供了HeadLiness Service，这类Service不会分配Cluster IP，如果想要访问service，只能通过service的域名进行查询。

创建service-headliness.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: service-headliness
  namespace: dev
spec:
  selector:
    app: nginx-pod
  clusterIP: None # 将clusterIP设置为None, 即可创建headliness Service
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
```

#### # 创建service

```
[root@master ~]# kubectl create -f service-headliness.yaml
service/service-headliness created
```

#### # 获取service, 发现CLUSTER-IP未分配

```
[root@master ~]# kubectl get svc service-headliness -n dev -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE	SELECTOR			
service-headliness	ClusterIP	None	<none>	80/TCP
11s	app=nginx-pod			

#### # 查看service详情

```
[root@master ~]# kubectl describe svc service-headliness -n dev
```

Name:	service-headliness
Namespace:	dev
Labels:	<none>
Annotations:	<none>
Selector:	app=nginx-pod
Type:	ClusterIP
IP:	None
Port:	<unset> 80/TCP

```
TargetPort:      80/TCP
Endpoints:       10.244.1.39:80,10.244.1.40:80,10.244.2.33:80
Session Affinity: None
Events:          <none>
```

# 查看域名的解析情况

```
[root@master ~]# kubectl exec -it pc-deployment-66cb59b984-8p84h -n dev
/bin/sh
```

```
/ # cat /etc/resolv.conf
```

```
nameserver 10.96.0.10
```

```
search dev.svc.cluster.local svc.cluster.local cluster.local
```

```
[root@master ~]# dig @10.96.0.10 service-
headliness.dev.svc.cluster.local
```

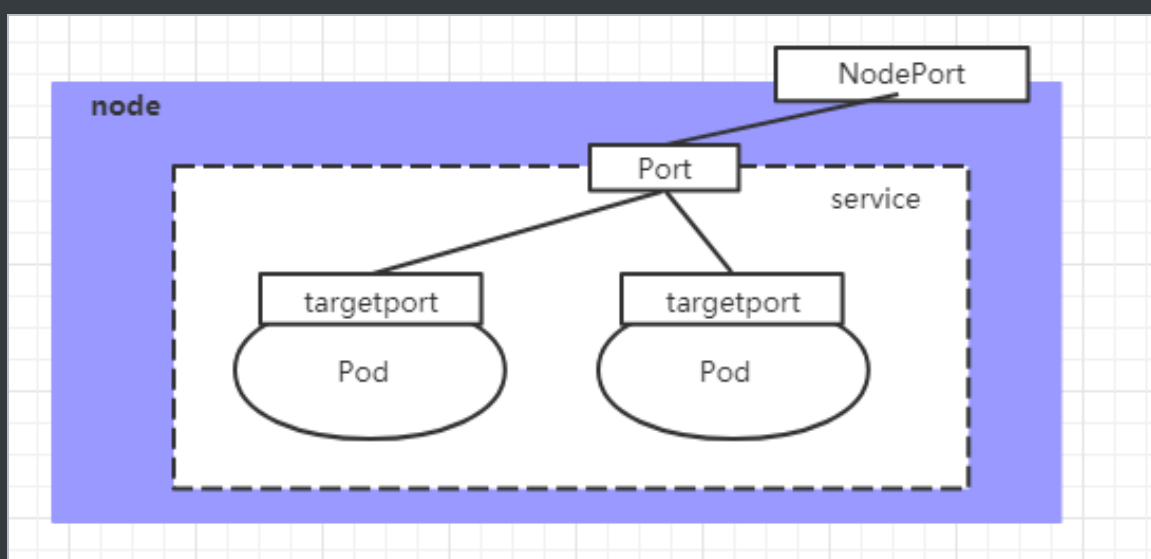
```
service-headliness.dev.svc.cluster.local. 30 IN A 10.244.1.40
```

```
service-headliness.dev.svc.cluster.local. 30 IN A 10.244.1.39
```

```
service-headliness.dev.svc.cluster.local. 30 IN A 10.244.2.33
```

## NodePort类型的Service

在之前的样例中，创建的Service的ip地址只有集群内部才可以访问，如果希望将Service暴露给集群外部使用，那么就要使用到另外一种类型的Service，称为NodePort类型。**NodePort的工作原理其实就是将service的端口映射到Node的一个端口上**，然后就可以通过NodeIp:NodePort 来访问service了。



## 创建service-nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: service-nodeport
  namespace: dev
spec:
  selector:
    app: nginx-pod
  type: NodePort # service类型
  ports:
    - port: 80
      nodePort: 30002 # 指定绑定的node的端口(默认的取值范围是: 30000-32767), 如果不指定, 会默认分配
      targetPort: 80
```

### # 创建service

```
[root@master ~]# kubectl create -f service-nodeport.yaml
service/service-nodeport created
```

### # 查看service

```
[root@master ~]# kubectl get svc -n dev -o wide
```

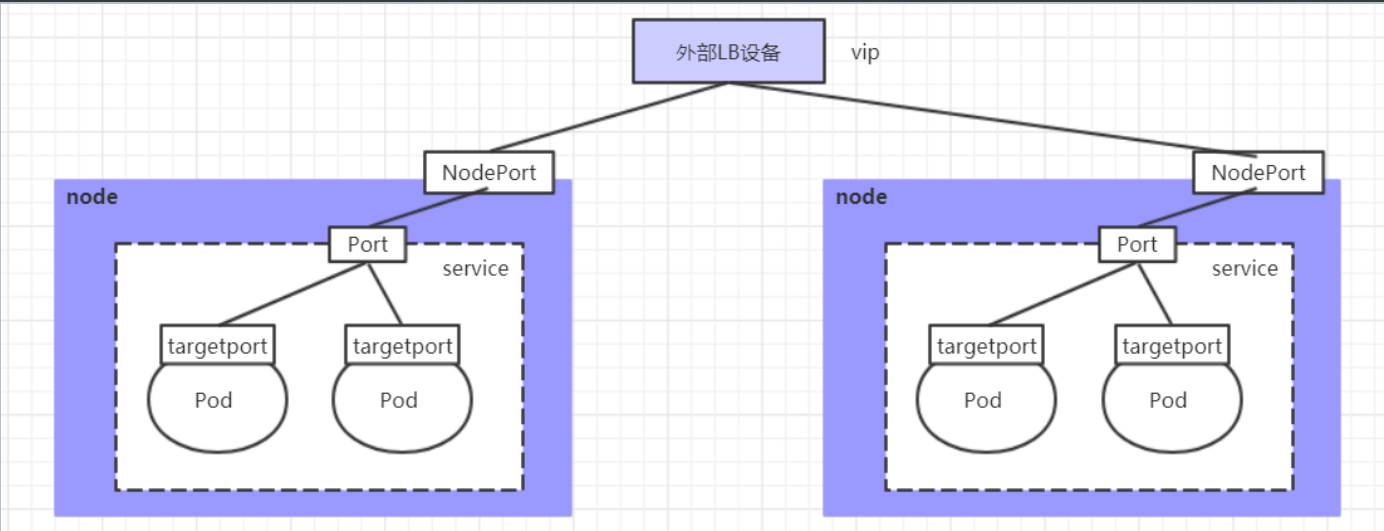
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
service-nodeport	NodePort	10.105.64.191	<none>	80:30002/TCP
selector: app=nginx-pod				

# 接下来可以通过电脑主机的浏览器去访问集群中任意一个nodeip的30002端口, 即可访问到 pod



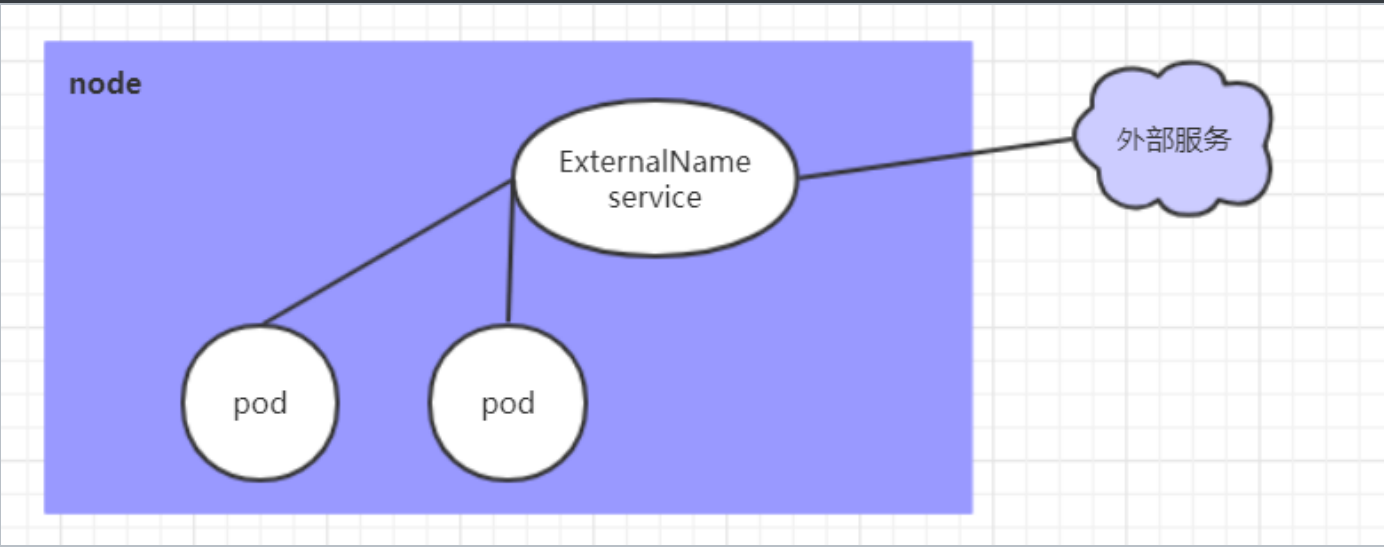
## LoadBalancer类型的Service

LoadBalancer和NodePort很相似，目的都是向外部暴露一个端口，区别在于LoadBalancer会在集群的外部再来做一个负载均衡设备，而这个设备需要外部环境支持的，外部服务发送到这个设备上的请求，会被设备负载之后转发到集群中。



## ExternalName类型的Service

ExternalName类型的Service用于引入集群外部的服务，它通过 externalName 属性指定外部一个服务的地址，然后在集群内部访问此service就可以访问到外部的服务了。



```
apiVersion: v1
kind: Service
metadata:
  name: service-externalname
  namespace: dev
spec:
  type: ExternalName # service类型
  externalName: www.baidu.com #改成ip地址也可以
```

# 创建service

```
[root@master ~]# kubectl create -f service-externalname.yaml
service/service-externalname created
```

# 域名解析

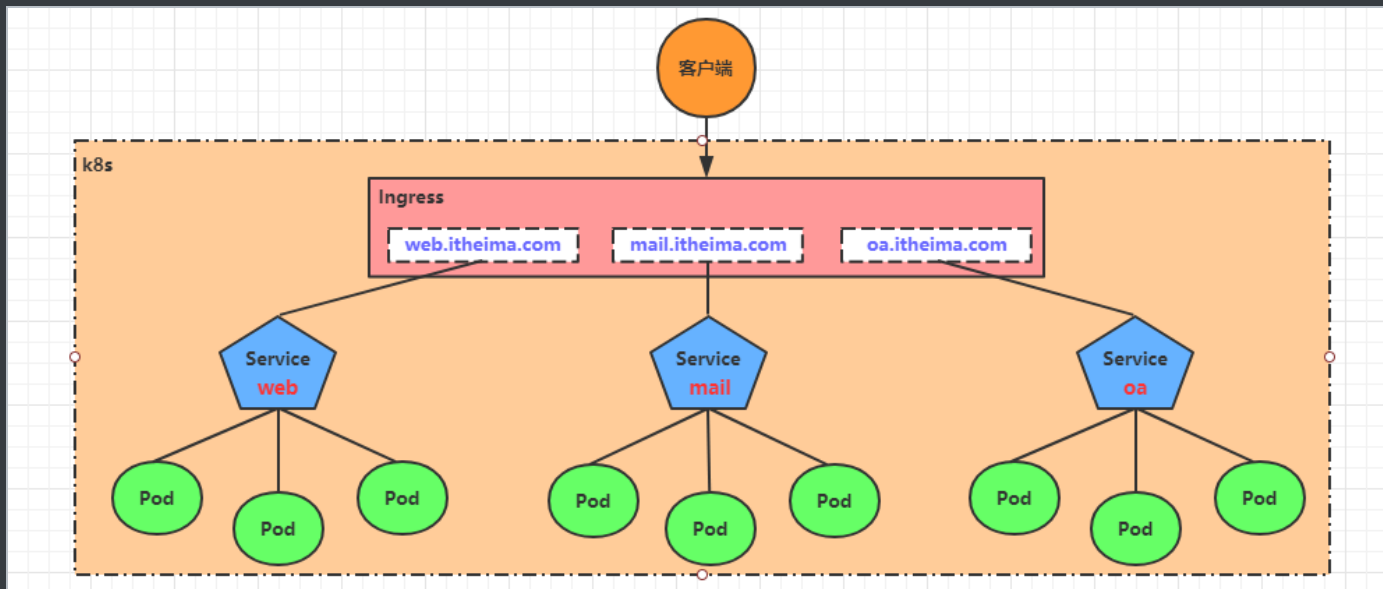
```
[root@master ~]# dig @10.96.0.10 service-externalname.dev.svc.cluster.local
service-externalname.dev.svc.cluster.local. 30 IN CNAME www.baidu.com.
www.baidu.com. 30 IN CNAME www.a.shifen.com.
www.a.shifen.com. 30 IN A 39.156.66.18
www.a.shifen.com. 30 IN A 39.156.66.14
```

## Ingress介绍

在前面课程中已经提到，Service对集群之外暴露服务的主要方式有两种：NodePort和LoadBalancer，但是这两种方式，都有一定的缺点：

- NodePort方式的缺点是会占用很多集群机器的端口，那么当集群服务变多的时候，这个缺点就愈发明显
- LB方式的缺点是每个service需要一个LB，浪费、麻烦，并且需要kubernetes之外设备的支持

基于这种现状，kubernetes提供了Ingress资源对象，Ingress只需要一个NodePort或者一个LB就可以满足暴露多个Service的需求。工作机制大致如下图所示：

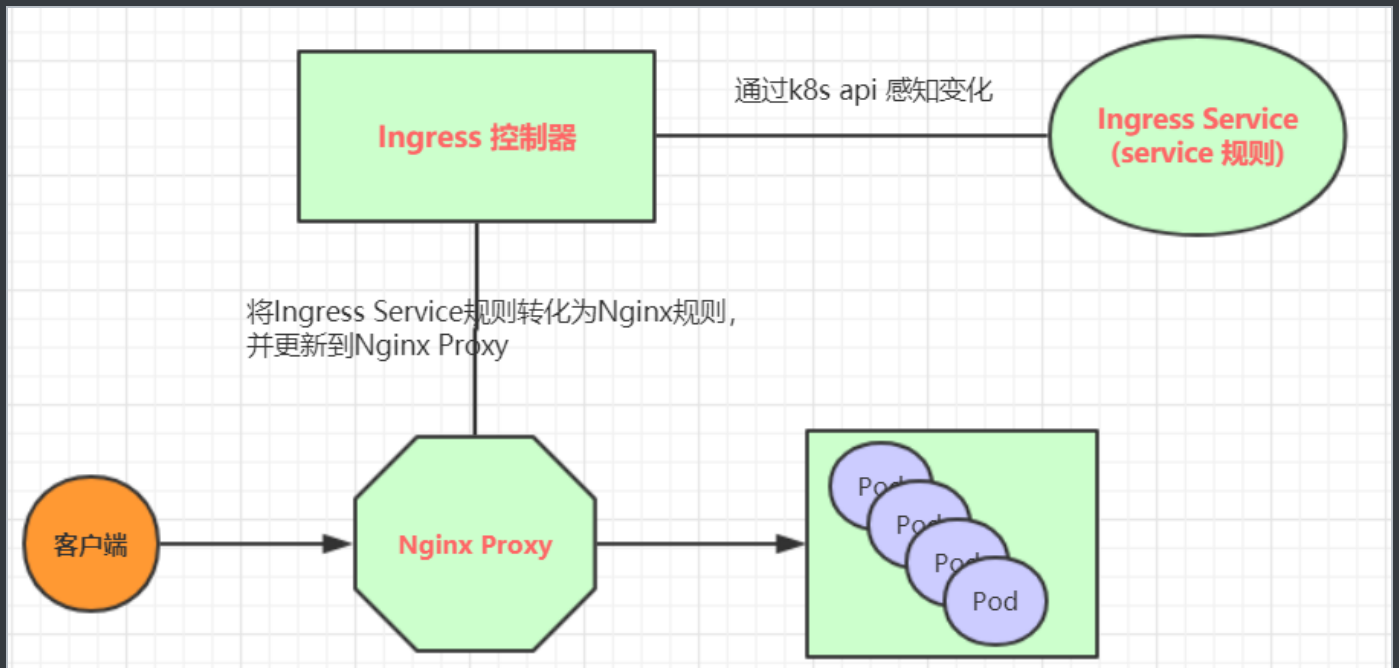


实际上，**Ingress**相当于一个7层的负载均衡器，是kubernetes对反向代理的一个抽象，它的工作原理类似于Nginx，可以理解成在**Ingress**里建立诸多映射规则，**Ingress Controller**通过监听这些配置规则并转化成**Nginx**的反向代理配置，然后对外部提供服务。在这里有两个核心概念：

- ingress：kubernetes中的一个对象，作用是定义请求如何转发到service的规则
- ingress controller：具体实现反向代理及负载均衡的程序，对ingress定义的规则进行解析，根据配置的规则来实现请求转发，实现方式有很多，比如Nginx, Contour, Haproxy等等

Ingress（以Nginx为例）的工作原理如下：

1. 用户编写Ingress规则，说明哪个域名对应kubernetes集群中的哪个Service
2. Ingress控制器动态感知Ingress服务规则的变化，然后生成一段对应的Nginx反向代理配置
3. Ingress控制器会将生成的Nginx配置写入到一个运行着的Nginx服务中，并动态更新
4. 到此为止，其实真正在工作的就是一个Nginx了，内部配置了用户定义的请求转发规则



# Ingress使用

## 环境准备

### 搭建ingress环境

```
# 创建文件夹
[root@master ~]# mkdir ingress-controller
[root@master ~]# cd ingress-controller/

# 获取ingress-nginx, 本次案例使用的是0.30版本
[root@master ingress-controller]# wget
https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-
0.30.0/deploy/static/mandatory.yaml
[root@master ingress-controller]# wget
https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-
0.30.0/deploy/static/provider/baremetal/service-nodeport.yaml

# 修改mandatory.yaml文件中的仓库
# 修改quay.io/kubernetes-ingress-controller/nginx-ingress-
controller:0.30.0
# 为quay-mirror.qiniu.com/kubernetes-ingress-controller/nginx-ingress-
controller:0.30.0
```

```
# 创建ingress-nginx
```

```
[root@master ingress-controller]# kubectl apply -f ./
```

```
# 查看ingress-nginx
```

```
[root@master ingress-controller]# kubectl get pod -n ingress-nginx
```

NAME	READY	STATUS
pod/nginx-ingress-controller-fbf967dd5-4qbp	1/1	Running
12h		0

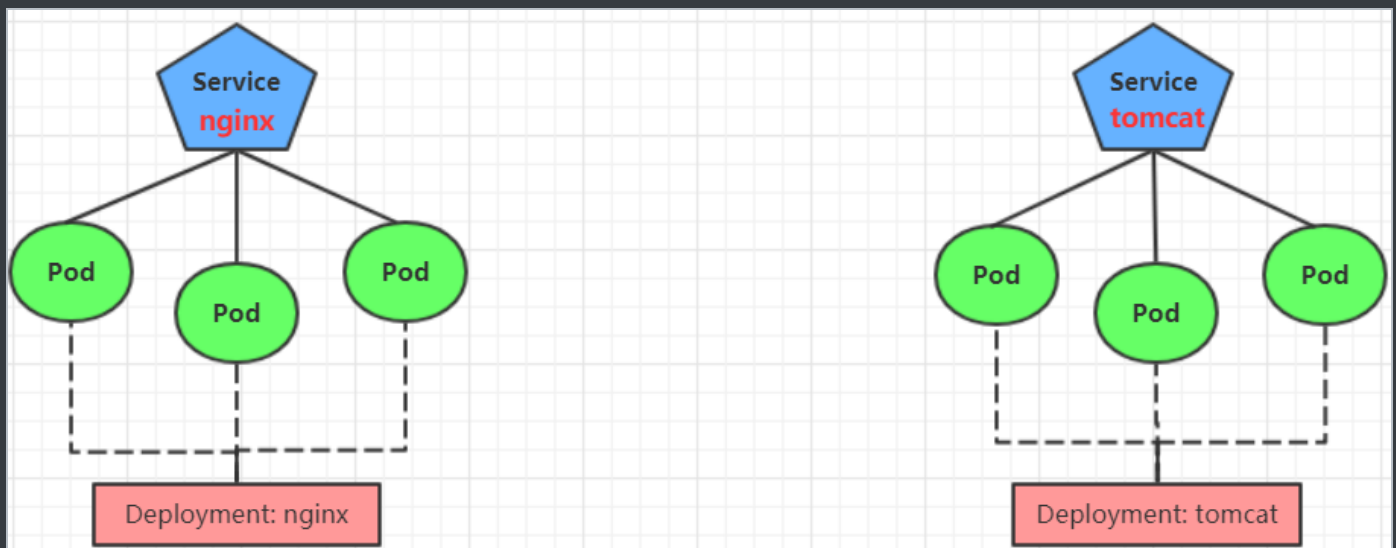
```
# 查看service
```

```
[root@master ingress-controller]# kubectl get svc -n ingress-nginx
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
ingress-nginx	NodePort	10.98.75.163	<none>	80:32240/TCP,443:31335/TCP
11h				

## 准备service和pod

为了后面的实验比较方便，创建如下图所示的模型



创建tomcat-nginx.yaml

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: nginx-deployment
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.17.1
          ports:
            - containerPort: 80
```

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deployment
  namespace: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: tomcat-pod
  template:
    metadata:
      labels:
        app: tomcat-pod
    spec:
```

```
containers:
  - name: tomcat
    image: tomcat:8.5-jre10-slim
    ports:
      - containerPort: 8080
```

---

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: dev
spec:
  selector:
    app: nginx-pod
  clusterIP: None
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
```

---

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
  namespace: dev
spec:
  selector:
    app: tomcat-pod
  clusterIP: None
  type: ClusterIP
  ports:
    - port: 8080
```

```
targetPort: 8080
```

# 创建

```
[root@master ~]# kubectl create -f tomcat-nginx.yaml
```

# 查看

```
[root@master ~]# kubectl get svc -n dev
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx-service	ClusterIP	None	<none>	80/TCP	48s
tomcat-service	ClusterIP	None	<none>	8080/TCP	48s

## Http代理

创建ingress-http.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-http
  namespace: dev
spec:
  rules:
    - host: nginx.itheima.com
      http:
        paths:
          - path: /
            backend:
              serviceName: nginx-service
              servicePort: 80
    - host: tomcat.itheima.com
      http:
        paths:
          - path: /
            backend:
              serviceName: tomcat-service
```



```
servicePort: 8080
```

```
# 创建
```

```
[root@master ~]# kubectl create -f ingress-http.yaml
ingress.extensions/ingress-http created
```

```
# 查看
```

```
[root@master ~]# kubectl get ing ingress-http -n dev
```

NAME	HOSTS	ADDRESS	PORTS
ingress-http	nginx.itheima.com,tomcat.itheima.com		80
AGE	22s		

```
# 查看详情
```

```
[root@master ~]# kubectl describe ing ingress-http -n dev
```

```
...
```

```
Rules:
```

Host	Path	Backends
-----	----	-----
nginx.itheima.com	/	nginx-service:80 (10.244.1.96:80,10.244.1.97:80,10.244.2.112:80)
tomcat.itheima.com	/	tomcat- service:8080(10.244.1.94:8080,10.244.1.95:8080,10.244.2.111:8080)

```
...
```

```
# 接下来,在本地电脑上配置host文件,解析上面的两个域名到192.168.109.100(master)上
# 然后,就可以分别访问tomcat.itheima.com:32240 和 nginx.itheima.com:32240
查看效果了
```

## Https代理

### 创建证书

# 生成证书

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout  
tls.key -out tls.crt -subj "/C=CN/ST=BJ/L=BJ/O=nginx/CN=itheima.com"
```

# 创建密钥

```
kubectl create secret tls tls-secret --key tls.key --cert tls.crt
```

创建ingress-https.yaml

```
apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
  name: ingress-https  
  namespace: dev  
spec:  
  tls:  
    - hosts:  
      - nginx.itheima.com  
      - tomcat.itheima.com  
      secretName: tls-secret # 指定密钥  
  rules:  
    - host: nginx.itheima.com  
      http:  
        paths:  
          - path: /  
            backend:  
              serviceName: nginx-service  
              servicePort: 80  
    - host: tomcat.itheima.com  
      http:  
        paths:  
          - path: /  
            backend:  
              serviceName: tomcat-service  
              servicePort: 8080
```

# 创建

```
[root@master ~]# kubectl create -f ingress-https.yaml
ingress.extensions/ingress-https created
```

# 查看

```
[root@master ~]# kubectl get ing ingress-https -n dev
```

NAME	HOSTS	ADDRESS
ingress-https	nginx.itheima.com,tomcat.itheima.com	10.104.184.38
80, 443	2m42s	

# 查看详情

```
[root@master ~]# kubectl describe ing ingress-https -n dev
```

...

TLS:

tls-secret terminates nginx.itheima.com,tomcat.itheima.com

Rules:

Host	Path	Backends
------	------	----------

----

-----

nginx.itheima.com	/	nginx-service:80
-------------------	---	------------------

(10.244.1.97:80,10.244.1.98:80,10.244.2.119:80)

tomcat.itheima.com	/	tomcat-
--------------------	---	---------

service:8080(10.244.1.99:8080,10.244.2.117:8080,10.244.2.120:8080)

...

# 下面可以通过浏览器访问<https://nginx.itheima.com:31335> 和  
<https://tomcat.itheima.com:31335>来查看了