主讲老师：Fox

> 有道笔记地址：https://note.youdao.com/s/ZIp1XCs4

1.k8s重要概念

1.1 Pod

> https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/#working-with-pods

Pod 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。

简单理解，Pod是一组容器的集合，这里的容器我们可以理解为就是Docker，当然除了 Docker 之外，Kubernetes 支持 很多其他容器运行时。

在Pod中的容器中，共享网络，存储以及怎样运行容器的一些声明。比如：一个Pod中的容器可以使用localhost来进行互相访问。Pod中的内容总是搭配在一起来运行，统一调度，在`共享上下文`中运行。

Pod 的共享上下文包括一组 Linux 命名空间、控制组（cgroup）和可能一些其他的`隔离`方面， 即用来隔离 `Docker` 容器的技术。 在 Pod 的上下文中，每个独立的应用可能会进一步实施隔离。就 Docker 概念的术语而言，Pod 类似于共享命名空间和文件系统卷的`一组 Docker 容器`。

Pod示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: dev
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
```

由一个运行镜像 `nginx:1.19` 的容器组成

- apiVersion 记录 k8s 的 API Server 版本
- kind 记录该 yaml 的对象，比如这是一份 Pod 的 yaml 配置文件，那么值内容就是`Pod`
- metadata 记录了 Pod 自身的元数据，比如这个 Pod 的名字、这个 Pod 属于哪个 namespace
- spec 记录了 Pod 内部所有的资源的详细信息
    - containers 记录了 Pod 内的容器信息

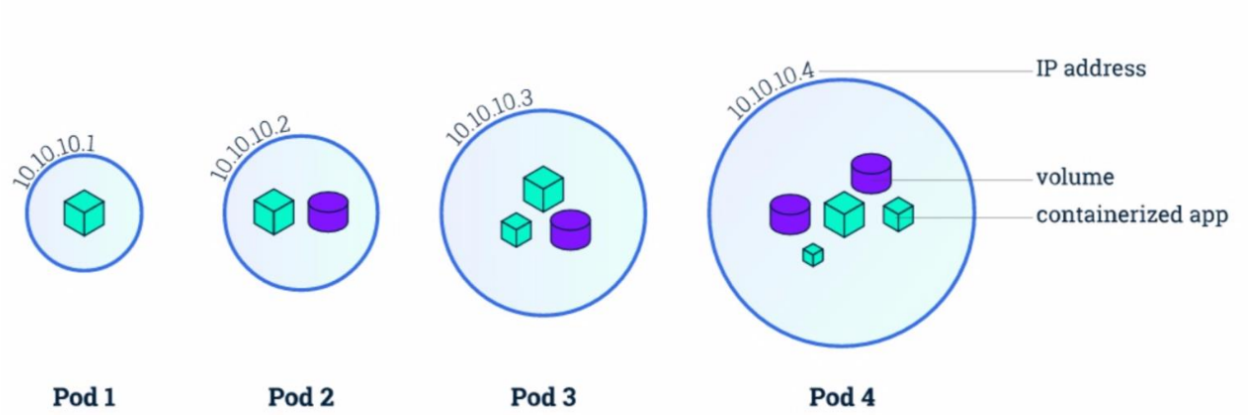
        - name 容器名
        - image 容器的镜像地址
        - resources 容器需要的 CPU、内存、GPU 等资源
        - command 容器的入口命令
        - args 容器的入口参数
        - volumeMounts 容器要挂载的 Pod 数据卷
        - ports 端口
        - env 环境变量
    - volumes 记录了 Pod 内的数据卷信息

Kubernetes 集群中的 Pod 主要有两种用法：

- **运行单个容器的 Pod。**"每个 Pod 一个容器" 模型是最常见的 Kubernetes 用例； 在这种情况下，可以将 Pod 看作单个容器的包装器，并且 Kubernetes 直接管理 Pod，而不是容器。
- **运行多个协同工作的容器的 Pod。** Pod 可能封装由多个紧密耦合且需要共享资源的共处容器组成的应用程序。

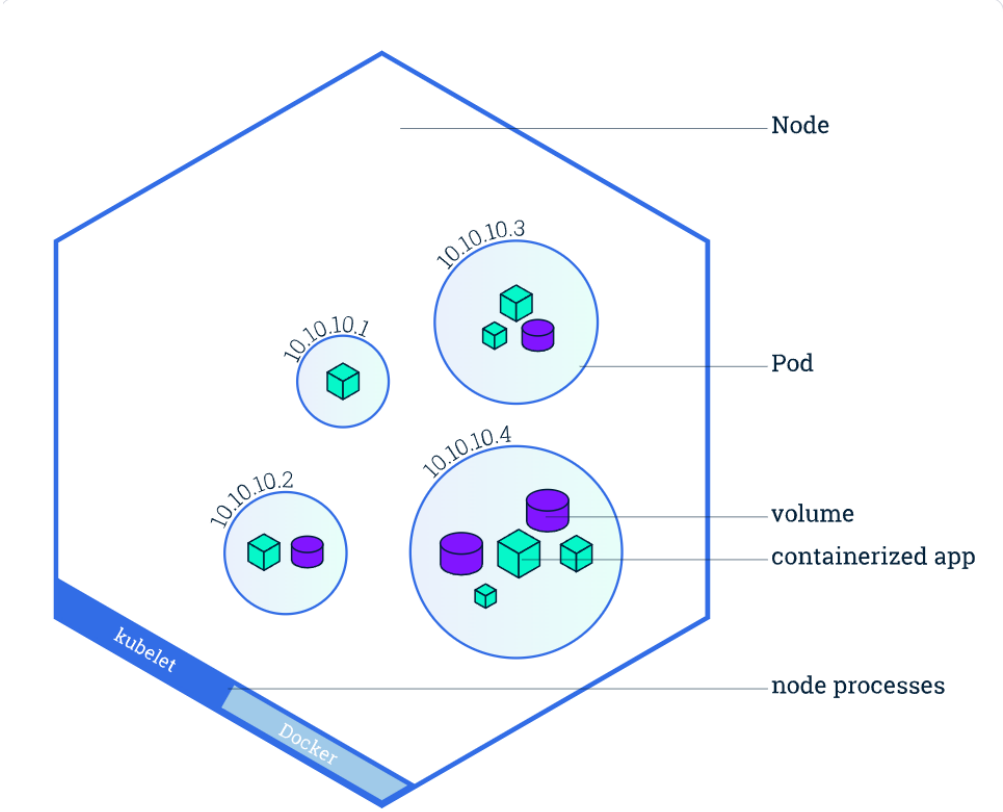
每个 Pod 都旨在运行给定应用程序的单个实例。如果希望横向扩展应用程序 （例如，运行多个实例以提供更多的资源），则应该使用多个 Pod，每个实例使用一个 Pod。 在 Kubernetes 中，这通常被称为**副本（Replication）**。

> 注意：Pod不是进程，只是提供了容器的运行环境，Pod一旦被创建，就会在其节点上运行，直到结束或者被销毁



在Pod中的容器也分为多种类型：

- 初始化容器（Init Container）：Init容器会在启动应用容器之前运行并完成
- 应用容器（App Container）：在初始化容器启动完毕后才开始启动，我们一般部署的容器
- 边车容器 （Sidecar Container）：与Pod中的应用容器一起运行的容器，Sidecar模式，在不更改主容器的基础上，增强其功能
- 临时容器（Ephemeral Container）：临时容器是使用 API 中的一种特殊的 `ephemeralcontainers` 处理器进行创建的，实现了调试容器附加到主进程的功能，然后你可以用于调试任何类型的问题，在做故障排查时很有用



标签管理

标签（Labels） 是附加到 Kubernetes 对象（比如 Pod）上的键值对。标签作用: **就是用来给 k8s 中对象起别名, 有了别名可以过滤和筛选**

标签由键值对组成，其有效标签值：

- 必须为 63 个字符或更少（可以为空）
- 除非标签值为空，必须以字母数字字符（[a-z0-9A-Z]）开头和结尾
- 包含破折号（-）、下划线（_）、点（.）和字母或数字

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  labels:
    name: myapp #创建时添加
spec:
  containers:
    - name: nginx
      image: nginx:1.21
      imagePullPolicy: IfNotPresent

```

```
    - name: redis
      image: redis:5.0.10
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

标签基本操作

```
# 查看标签
$ kubectl get pods --show-labels

# kubectl label pod pod名称 标签键值对
$ kubectl label pod myapp env=prod

# 覆盖标签 --overwrite
$ kubectl label --overwrite pod myapp env=test

# 删除标签 -号代表删除标签
$ kubectl label pod myapp env-

# 根据标签筛选 env=test/env  > = <
$ kubectl get po -l env=test
$ kubectl get po -l env
$ kubectl get po -l '!env' # 不包含的 pod
$ kubectl get po -l 'env in (test,prod)' #选择含有指定值的 pod
$ kubectl get po -l 'env notin (test,prod)' #选择含有指定值的 pod
```

自定义容器启动命令

和 Docker 容器一样,k8s中容器也可以通过command、args 用来修改容器启动默认执行命令以及传递相关参数。**但一般推荐使用 command 修改启动命令，使用 args 为启动命令传递参数。**

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
```

```
    app: redis
spec:
  containers:
    - name: redis
      image: redis:5.0.10
      command: ["redis-server"] #用来指定启动命令
      args: ["--appendonly yes"] # 用来为启动命令传递参数  配置aof持久化
      #args: ["redis-server","--appendonly yes"] # 单独使用修改启动命令并传递参数
      #args:                                     # 另一种语法格式
      #  - redis-server
      #  - "--appendonly yes"
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

测试效果

```
[root@k8s-master01 k8s-demo]# kubectl exec -it redis -- bash
root@redis:/data# redis-cli
127.0.0.1:6379> config get appendonly
1) "appendonly"
2) "yes"
127.0.0.1:6379>
```

资源分配

容器中的程序要运行，肯定是要占用一定资源的，比如cpu和内存等，如果不对某个容器的资源做限制，那么它就可能吃掉大量资源，导致其他容器无法运行。

> https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/assign-cpu-resource/

针对这种情况，k8s提供了对内存和cpu的资源进行配额的机制，这种机制主要通过resources选项实现，它有两个子选项：

- limits：用于限制运行时容器的最大占用资源，当容器占用资源超过limits时会被终止，并进行重启
- requests：用于设置容器需要的最小资源，如果环境资源不够，容器将无法启动

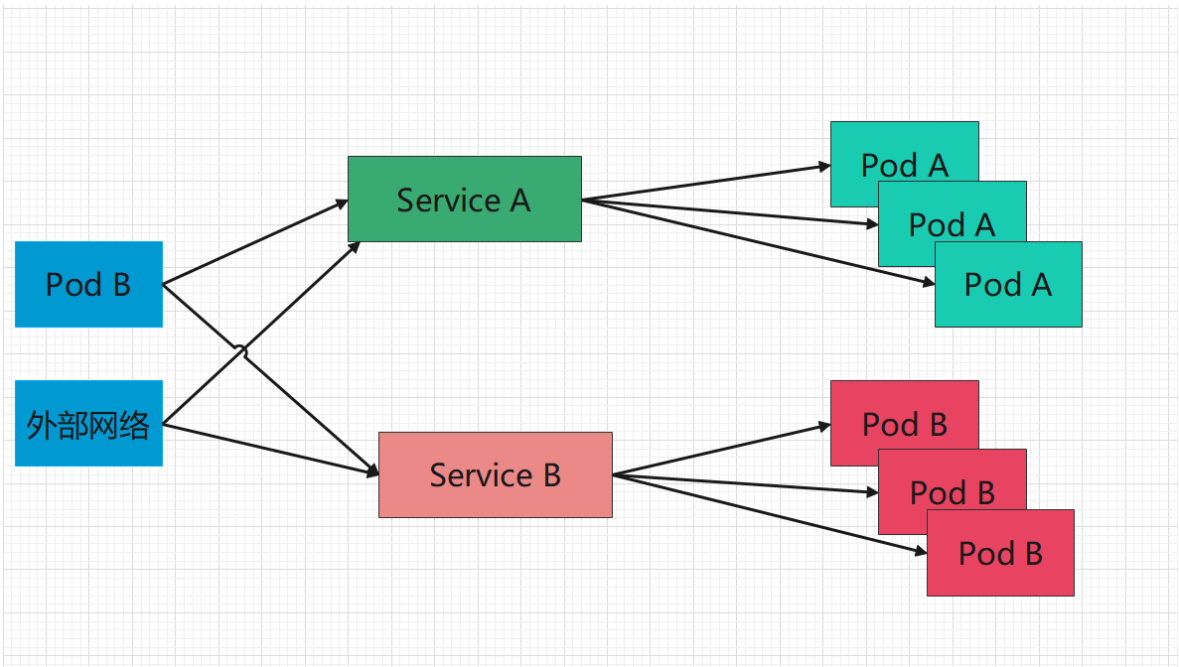
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-resources
  namespace: dev
spec:
  containers:
```

```
  - name: nginx
    image: nginx:1.17.1
    resources: #资源分配
      limits: #限制资源(上限)
        cpu: "2" #cpu限制
        memory: "10Gi"  #内存限制
      requests: #请求资源(下限)
        cpu: "1"
        memory: "10Mi"  #内存限制
```

1.2 Service

官方定义：

> 将运行在一组 Pods 上的应用程序公开为网络服务的抽象方法。使用 Kubernetes，您无需修改应用程序即可使用不熟悉的服务发现机制。Kubernetes 为 Pods 提供自己的 IP 地址，并为一组 Pod 提供相同的 DNS 名， 并且可以在它们之间进行负载均衡。



配置说明（资源文件清单）

```
kind: Service  #资源类型
apiVersion: v1 #资源版本
metadata:
  name: service
  namespace: dev
```

```
spec:
  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
  ports: #端口信息
    - protocol: TCP
      port: 3017  #service端口
      targetPort: 5003  #pod端口
      nodePort: 31122   #主机端口
```

Service类型

- ClusterIp：默认值，它是K8S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到集群内部直接使用

假定有一组 Pod，它们对外暴露了 9376 端口，同时还被打上 app=MyApp 标签

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: dev
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

上述配置创建一个名称为 "my-service" 的 Service 对象，它会将请求代理到使用 TCP 端口 9376，并且具有标签 "app=MyApp" 的 Pod 上。

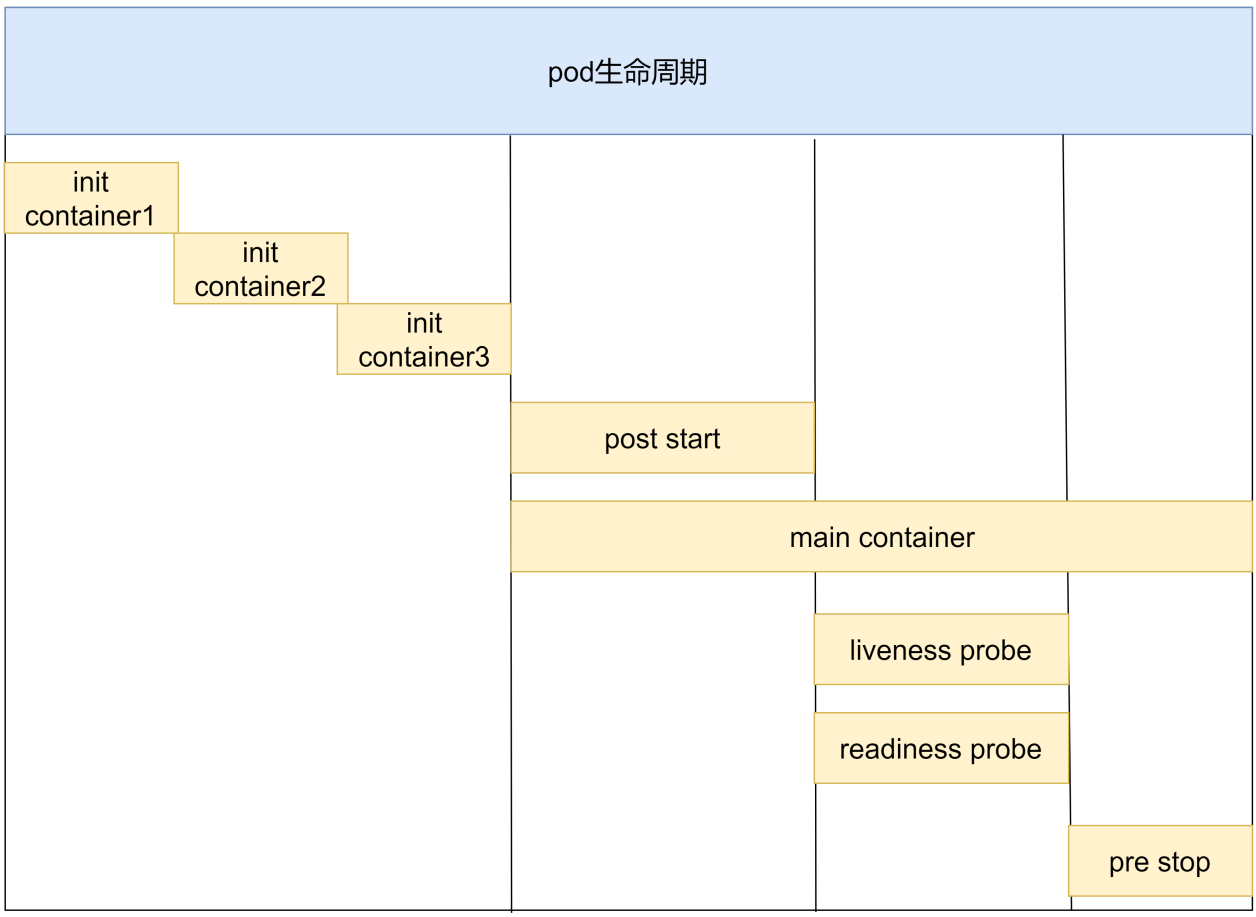
Kubernetes 为该服务分配一个 IP 地址（有时称为 “集群 IP”），该 IP 地址由服务代理使用。

2. Pod生命周期

> 官网: https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/pod-lifecycle/

我们一般将Pod对象从创建到终止的这段时间范围称为Pod的生命周期，它主要包含下面的过程：

- pod创建过程
- 运行初始化容器（init container）过程
- 运行主容器（main container）
  - 容器启动后钩子（post start）、容器终止前钩子（pre stop）
  - 容器的存活性检测（liveness probe）、就绪性检测（readiness probe）
- pod终止过程



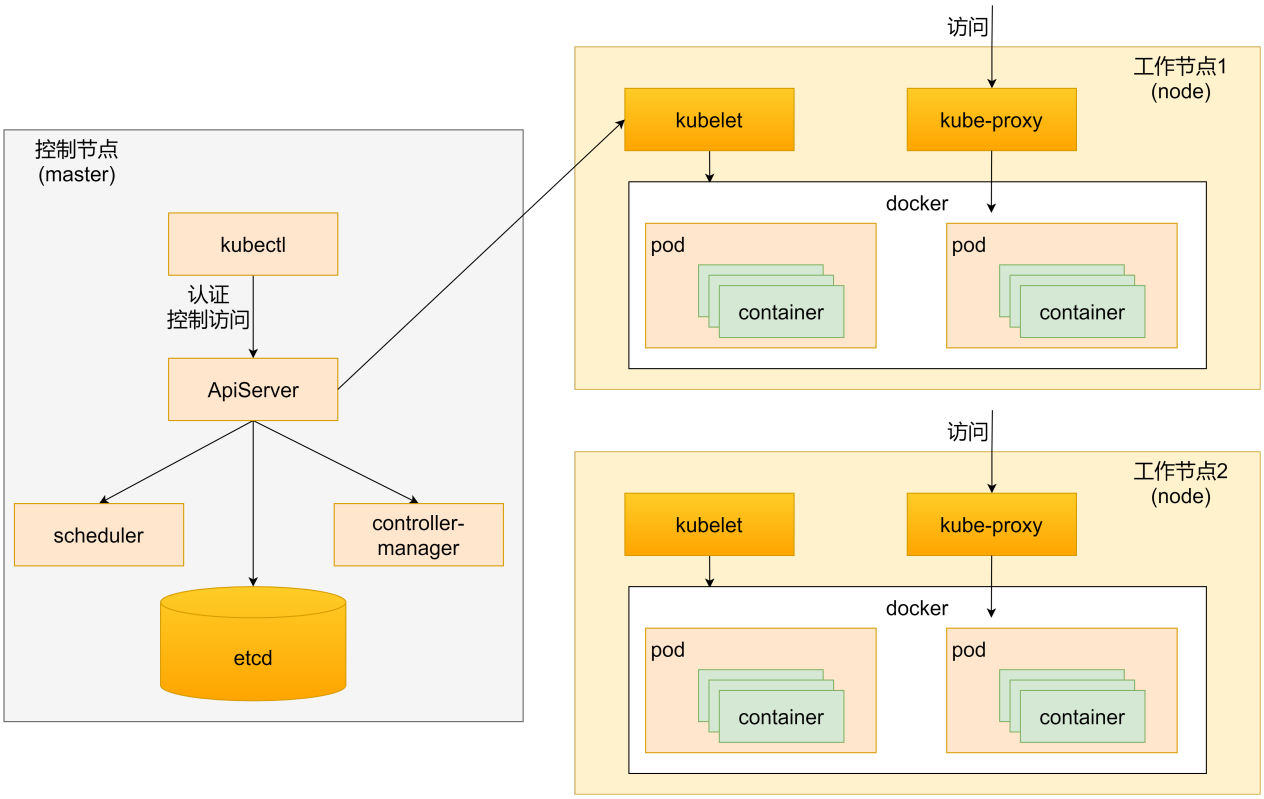
我们首先了解下Pod的状态值，我们可以通过`kubectl explain pod.status`命令来了解关于Pod的一些信息，Pod的状态定义在`PodStatus`对象中，其中有一个`phase`字段，下面是`phase`的可能取值：

取值	描述
`Pending`（悬决）	Pod 已被 Kubernetes 系统接受，但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间。
`Running`（运行中）	Pod 已经绑定到了某个节点，Pod 中所有的容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态。
`Succeeded`（成功）	Pod 中的所有容器都已成功终止，并且不会再重启。
`Failed`	Pod 中的所有容器都已终止，并且至少有一个容器是因为失败终止。也就是说，容

（失败）	器以非 0 状态退出或者被系统终止。
Unknown（未知）	因为某些原因无法取得 Pod 的状态。这种情况通常是因为与 Pod 所在主机通信失败。

> **说明：**
>
> 1. 当一个 Pod 被删除时，执行一些 kubectl 命令会展示这个 Pod 的状态为 Terminating（终止）。 这个 Terminating 状态并不是 Pod 阶段之一。 Pod 被赋予一个可以体面终止的期限，默认为 30 秒。 你可以使用 --force 参数来强制终止 Pod。
> 2. 如果某节点死掉或者与集群中其他节点失联，Kubernetes 会实施一种策略，将失去的节点上运行的所有 Pod 的 phase 设置为 Failed。

2.1 pod创建过程



1. 用户通过kubectl或其他api客户端提交需要创建的pod信息给apiserver
2. apiserver开始生成pod对象的信息，并将信息存入etcd，然后返回确认信息至客户端
3. apiserver开始反映etcd中pod对象的变化，其他组件使用watch机制来跟踪检查apiserver上的变动
4. scheduler发现有新的pod对象要创建，开始为pod分配主机并将结果信息更新至apiserver
5. node节点上的kubelet发现有pod调度过来，尝试启动容器，并将结果返回送至apiserver
6. apiserver将接收到的pod状态信息存入etcd中

2.2 终止过程

1. 用户向apiserver发送删除pod对象的命令
2. apiserver中的pod对象信息会随着时间的推移而更新，在宽限期内（默认30s），pod被视为dead

3. 将pod标记为terminating状态
4. kubelet在监控到pod对象转为terminating状态的同时启动pod关闭进程
5. 端点控制器监控到pod对象的关闭行为时将其从所有匹配到此端点的service资源的端点列表移除
6. 如果当前pod对象定义了prestop钩子处理器，则在其标记为terminating后即会以同步的方式启动执行
7. pod对象中的容器进程收到停止信号
8. 宽限期结束后，若pod中还存在仍在运行的进程，那么pod对象会收到立即终止的信号
9. kubelet请求apiserver将此pod资源的宽限期设置为0从而完成删除操作，此时pod对于用户已不可见

2.3 初始化容器

> 官网地址: https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/

初始化容器Init Container是在Pod的主容器启动之前要运行的容器，主要是做一些主容器的前置工作，可以是一个 它具有两大特征：

- 初始化容器必须运行完成直至结束，如果某个初始化容器运行失败，那么Kubernetes需要重启它直至成功完成。
- 初始化容器必须按照定义的顺序执行，当且仅当前一个成功之后，后面一个才能运行。

初始化容器有很多应用场景，下面列出的是最常见的几种：

- 提供主容器镜像中不具备的工具程序或自定义代码。
- 初始化容器要先于应用容器串行启动并运行完成；因此可用于延后应用容器的启动直至其依赖的条件得到满足。

使用示例

在 Pod 的规约中与用来描述应用容器的 `containers` 数组平行的位置指定 Init 容器。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'echo init-myservice is running! && sleep 5']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'echo init-mydb is running! && sleep 10']
```

查看pod详情

```
kubectl describe  pod/init-demo
```

2.4 容器钩子/回调

钩子函数能够感知自身生命周期中的事件，并在相应的时刻到来时运行用户指定的程序代码。
k8s在主容器的启动之后和停止之前提供了两个钩子函数:

- postStart：容器创建之后执行，如果失败了会重启容器，主要用于资源部署、环境准备等。不过需要注意的是如果钩子花费太长时间以至于不能运行或者挂起，容器将不能达到 running 状态。
- preStop：容器终止之前执行，执行完成之后容器将成功终止，在其完成之前会阻塞删除容器的操作，主要用于优雅关闭应用程序、通知其他系统等。如果钩子在执行期间挂起，Pod 阶段将停留在 running 状态并且永不会达到 failed 状态。

有三种方式来实现上面的钩子函数：

- Exec命令：在容器内执行一次命令，不过要注意的是该命令消耗的资源会被计入容器

```
......
    lifecycle:
        postStart:
            exec:
                command:
                    - cat
                    - /tmp/healthy
......
```

- TCPSocket：在当前容器尝试访问指定的socket

```
......
    lifecycle:
        postStart:
            tcpSocket:
                port:8080
......
```

```
```

- HttpGet：在当前容器中向某url发起http请求

```
......
    lifecycle:
        postStart:
            httpGet:
                path:/ # url地址
                port:80  # 端口号
                host:192.168.209.128  # 主机地址
                scheme: HTTP  # 支持的协议，http或https
......
```

使用示例

```
# nginx-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.19
      lifecycle:
        postStart: #容器创建过程中执行
          exec:
            command: ["/bin/sh","-c","echo postStart >> /start.txt"]
        preStop:  #容器终止之前执行
          exec:
            command: ["/bin/sh","-c","echo preStop >> /stop.txt && sleep 15"]
      ports:
        - containerPort: 80
```

部署pod，进入容器查看postStart钩子是否执行

删除pod,查看preStop钩子是否执行

2.5 容器探针

容器探针就是用来定期对容器进行健康检查的。

> https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/pod-lifecycle/#container-probes

探测类型

针对运行中的容器，`kubelet` 可以选择是否执行以下三种探针，以及如何针对探测结果作出反应：

- livenessProbe 存活探针，用于检测应用实例当前是否处于正常运行状态，如果不是，k8s会重启容器。如果容器不提供存活探针， 则默认状态为 Success。
- readinessProbe 就绪探针，用于检测应用实例当前是否可以接受请求，如果不能，k8s不会转发流量。如果就绪态探测失败， 端点控制器将从与 Pod 匹配的所有服务的端点列表中删除该 Pod 的 IP 地址。 初始延迟之前的就绪态的状态值默认为 Failure。 如果容器不提供就绪态探针，则默认状态为 Success。
- startupProbe 1.7+ 启动探针，用于容器中的应用是否已经启动。如果提供了启动探针，则所有其他探针都会被禁用，直到此探针成功为止。如果启动探测失败，kubelet 将杀死容器， 而容器依其重启策略进行重启。 如果容器没有提供启动探测，则默认状态为 Success。

探针机制

使用探针来检查容器有四种不同的方法。 每个探针都必须准确定义为这四种机制中的一种：

- exec

在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。

```
1 ......
2       livenessProbe:
3         postStart:
4           exec:
5             command:
6             - cat
7             - /tmp/healthy
8 ......
```

- grpc

使用gRPC 执行一个远程过程调用。 目标应该实现 gRPC健康检查。 如果响应的状态是 "SERVING"，则认为诊断成功。 gRPC 探针是一个 Alpha 特性，只有在你启用了 "GRPCContainerProbe" 特性门控时才能使用。

```
......
    livenessProbe:
      grpc:
        port: 2379
......
```

- httpGet

对容器的 IP 地址上指定端口和路径执行 HTTP GET 请求。如果响应的状态码大于等于 200 且小于 400，则诊断被认为是成功的。

```
......
  lifecycle:
    postStart:
      httpGet:
        path: #uri地址
        port:
        host:
        scheme: HTTP  #支持的协议，http或者https
......
```

- tcpSocket

对容器的 IP 地址上的指定端口执行 TCP 检查。如果端口打开，则诊断被认为是成功的。 如果远程系统（容器）在打开连接后立即将其关闭，这算作是健康的。

```
......
    livenessProbe:
      tcpSocket:
        port: 8080
......
```

探针使用示例

下面以liveness probes为例：

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  labels:
    exec: exec
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
    args:
    - /bin/sh
    - -c
    - sleep 7;nginx -g "daemon off;" #这一步会和初始化同时开始运行，也就是在初始化5s后和7秒之间，会检测出一次失败，7秒后启动后检测正常，所以pod不会重启
    imagePullPolicy: IfNotPresent
    livenessProbe:
      exec:    #这里使用 exec 执行 shell 命令检测容器状态
        command:
        - ls
        - /var/run/nginx.pid  #查看是否有pid文件
      initialDelaySeconds: 5   #初始化时间5s
      periodSeconds: 4    #检测间隔时间4s
      timeoutSeconds: 1   #默认检测超时时间为1s
      failureThreshold: 3   #默认失败次数为3次，达到3次后重启pod
      successThreshold: 1   #默认成功次数为1次，1 次代表成功
```

部署后测试pod是否重启

休眠时间改为30s,测试pod是否重启

测试结果：

1. 如果 sleep 7s，第一次检测发现失败，但是第二次检测发现成功后容器就一直处于健康状态不会重启。

2. 如果 sleep 30s，第一次检测失败，超过 3 次检测同样失败，k8s 就回杀死容器进行重启，反复循环这个过程。

2.6 重启策略

> https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

在容器探测中，一旦容器探测出现了问题，Kubernetes就会对容器所在的Pod进行重启，其实这是由Pod的重启策略决定的，Pod的重启策略有3种，分别如下：

- Always：容器失效时，自动重启该容器，默认值。
- OnFailure：容器终止运行且退出码不为0时重启。
- Never：不论状态如何，都不重启该容器。

重启策略适用于Pod对象中的所有容器，首次需要重启的容器，将在其需要的时候立即进行重启，随后再次重启的操作将由kubelet延迟一段时间后进行，且反复的重启操作的延迟时长以此为10s、20s、40s、80s、160s和300s，300s是最大的延迟时长。一旦某容器执行了 10 分钟并且没有出现问题，kubelet 对该容器的重启回退计时器执行重置操作。

使用示例

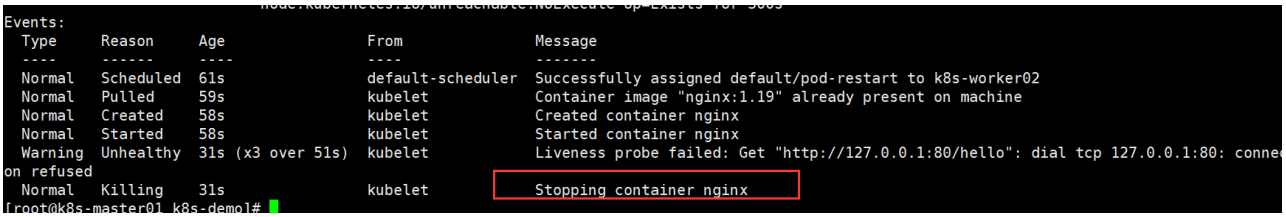
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-restart

  labels:
    user: fox
spec:
  containers:
    - name: nginx # 容器名称
      image: nginx:1.19  # 容器需要的镜像地址
      imagePullPolicy: IfNotPresent  # 设置镜像的拉取策略

      ports:
        - name: nginx-port  # 端口名称，如果执行，必须保证name在Pod中是唯一的
          containerPort: 80 # 容器要监听的端口
          protocol: TCP # 端口协议
      livenessProbe: # 声明周期配置
        httpGet:
          port: 80
          scheme: HTTP
          path: /hello
          host: 127.0.0.1
```

```
22    restartPolicy: Never # 重启策略
```

部署后测试，会发现容器探测失败后，直接停止容器，并没有选择重启。



3. Pod调度

在默认情况下，一个pod在哪个node节点上运行，是由scheduler组件采用相应的算法计算出来的，这个过程是不受人工控制的。

但是在实际过程中，这并不满足需求，因为很多情况下，我们想控制某些pod到达某些节点上，那么应该怎么做呢？

这就要求了解k8s对Pod的调度规则，k8s提供了四大类调度方式：

- 自动调度：运行在哪个节点上完全由Scheduler经过一系列的算法得出
- 定向调度：NodeName、NodeSelector
- 亲和性调度：NodeAffinity、PodAffinity、PodAntiAffinity
- 污点（容忍）调度：Taints、Toleration

3.1 定向调度

定向调度，指的是利用在Pod上声明的NodeName或NodeSelector，以此将Pod调度到期望的Node节点上。注意，这里的调度是强制的，这就意味着即使要调度的目标Node不存在，也会向上面进行调度，只不过Pod运行失败而已。

nodeName

nodeName用于强制约束将Pod调度到指定的name的Node节点上。这种方式，其实是直接跳过Scheduler的调度逻辑，直接将Pod调度到指定名称的节点。

示例

创建一个pod-nodename.yaml文件

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
```

```
spec:
  nodeName: k8s-worker04    # 调度 Pod 到特定的节点
  containers:
  - name: nginx
    image: nginx:1.19
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
```

pod被强制调度到k8s-worker04节点

修改nodeName为k8s-worker05，测试

虽然被指定在了k8s-worker05，但是由于k8s-worker05不存在，pod无法启动。

nodeSelector

nodeSelector用于将Pod调度到添加了指定标签的Node节点上，它是通过Kubernetes的label-selector机制实现的。换言之，在Pod创建之前，会由Scheduler使用MatchNodeSelector调度策略进行label匹配，找出目标node，然后将Pod调度到目标节点，该匹配规则是强制约束。

节点标签管理

```
#列出集群中的节点及其标签
kubectl get nodes --show-labels
#选择一个节点，给它添加一个标签
#kubectl label nodes nodeName(节点名称) labelName=value
kubectl label nodes k8s-worker02 nodeenv=pro
kubectl label nodes k8s-worker03 nodeenv=test

#删除节点标签
kubectl label nodes k8s-worker03 nodeenv-
# 覆盖标签
kubectl label --overwrite nodes k8s-worker02 nodeenv=test

# 查看具有 nodeenv=pro标签的节点
kubectl get nodes --show-labels | grep nodeenv=pro
```

使用示例

1）为node节点添加标签

```
kubectl label nodes k8s-worker02 nodeenv=pro
```

2）创建一个pod-nodeselector.yaml文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeselector

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  nodeSelector:
    nodeenv: pro  #指定调度到具有nodeenv=pro标签的节点上
```

3）验证

4) 如果调度到不存在的标签上，Pod无法启动

3.2 亲和性调度

虽然定向调度的两种方式，使用起来非常方便，但是也有一定的问题，那就是如果没有满足条件的Node，那么Pod将不会被运行，即使在集群中还有可用的Node列表也不行，这就限制了它的使用场景。

基于上面的问题，Kubernetes还提供了一种亲和性调度（Affinity）。它在nodeSelector的基础之上进行了扩展，可以通过配置的形式，实现优先选择满足条件的Node进行调度，如果没有，也可以调度到不满足条件的节点上，使得调度更加灵活。

Affinity主要分为三类：

- nodeAffinity（node亲和性）：以Node为目标，解决Pod可以调度到那些Node的问题。
- podAffinity（pod亲和性）：以Pod为目标，解决Pod可以和那些已存在的Pod部署在同一个拓扑域中的问题。
- podAntiAffinity（pod反亲和性）：以Pod为目标，解决Pod不能和那些已经存在的Pod部署在同一拓扑域中的问题。

> 亲和性：如果两个应用频繁交互，那么就有必要利用亲和性让两个应用尽可能的靠近，这样可以较少因网络通信而带来的性能损耗。

> 反亲和性：当应用采用多副本部署的时候，那么就有必要利用反亲和性让各个应用实例打散分布在各个Node上，这样可以提高服务的高可用性。

nodeAffinity

nodeAffinity的可选配置项：

```
requiredDuringSchedulingIgnoredDuringExecution  #Node节点必须满足指定的所有规则才可以，相当于硬限制
    nodeSelectorTerms  #节点选择列表
      matchFields   # 按节点字段列出的节点选择器要求列表
      matchExpressions   #按节点标签列出的节点选择器要求列表(推荐)
        key    #键
        values #值
        operator #关系符 支持Exists, DoesNotExist, In, NotIn, Gt, Lt

preferredDuringSchedulingIgnoredDuringExecution #优先调度到满足指定的规则的Node，相当于软限制 (倾向)
    preference   #一个节点选择器项，与相应的权重相关联
      matchFields #按节点字段列出的节点选择器要求列表
      matchExpressions   #按节点标签列出的节点选择器要求列表(推荐)
        key #键
        values #值
        operator #关系符 支持In, NotIn, Exists, DoesNotExist, Gt, Lt
    weight # 倾向权重，在范围1-100。
```

关系符的使用说明：

```
- matchExpressions:
    - key: env # 匹配存在标签的key为env的节点
      operator: Exists
    - key: env # 匹配标签的key为env,且value是"xxx"或"yyy"的节点
      operator: In
      values: ["xxx","yyy"]
    - key: env # 匹配标签的key为env,且value大于"xxx"的节点
      operator: Gt
      values: "xxx"

```

requiredDuringSchedulingIgnoredDuringExecution

硬限制示例

创建pod-nodeaffinity-required.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeaffinity-required

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    nodeAffinity:   #设置node亲和性
      requiredDuringSchedulingIgnoredDuringExecution:  #硬限制
        nodeSelectorTerms:
        - matchExpressions:
          - key: nodeenv
            operator: In
            values: ["xxx","yyy"]
```

部署后测试，pod调度失败

重新编辑配置文件的values

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeaffinity-required

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    nodeAffinity:   #设置node亲和性
      requiredDuringSchedulingIgnoredDuringExecution:  #硬限制
```

```
        nodeSelectorTerms:
        - matchExpressions:
          - key: nodeenv
            operator: In
            values: ["pro","yyy"]
```

> 注意，保证有节点有nodeenv=pro标签，没有可以执行kubectl label nodes k8s-worker02 nodeenv=pro

preferredDuringSchedulingIgnoredDuringExecution

软限制

创建pod-nodeaffinity-preferred.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeaffinity-preferred
spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    nodeAffinity:   #设置node亲和性
      preferredDuringSchedulingIgnoredDuringExecution:  # 优先调度到满足指定的规则的Node，相当于软限制 (倾向)
      - weight: 1
        preference: # 一个节点选择器项，与相应的权重相关联
          matchExpressions:
          - key: nodeenv
            operator: In
            values: ["xxx","yyy"]
```

测试结果

即使没有满足条件，pod也被正常调度了

注意：

> 如果同时定义了nodeSelector和nodeAffinity，那么必须两个条件都满足，Pod才能运行在指定的Node上。如果nodeAffinity指定了多个nodeSelectorTerms，那么只需要其中一个能够匹配成功即可。如果一个nodeSelectorTerms中有多个matchExpressions，则一个节点必须满足所有的才能匹配成功。如果一个Pod所在的Node在Pod运行期间其标签发生了改变，不再符合该Pod的nodeAffinity的要求，则系统将忽略此变化。

podAffinity

podAffinity主要实现以运行的Pod为参照，实现让新创建的Pod和参照的Pod在一个区域的功能。

查看PodAffinity的可选配置项

```
requiredDuringSchedulingIgnoredDuringExecution  # 硬限制
  namespaces # 指定参照pod的namespace
  topologyKey # 指定调度作用域
  labelSelector # 标签选择器
    matchExpressions  # 按节点标签列出的节点选择器要求列表(推荐)
      key    #  键
      values # 值
      operator # 关系符 支持In, NotIn, Exists, DoesNotExist.
    matchLabels    # 指多个matchExpressions映射的内容

preferredDuringSchedulingIgnoredDuringExecution # 软限制
  podAffinityTerm  # 选项
    namespaces
    topologyKey
    labelSelector
       matchExpressions
          key    #  键
          values # 值
          operator
       matchLabels
  weight 倾向权重，在范围1-100


```

> topologyKey用于指定调度的作用域，例如: 如果指定为kubernetes.io/hostname，那就是以Node节点为区分范围。 如果指定为beta.kubernetes.io/os，则以Node节点的操作系统类型来区分。

requireDuringSchedullingIgnoreDuringExecution

1) 创建一个参照pod，pod-podaffinity-target.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-podaffinity-target

  labels:
    podenv: pro   #设置标签
spec:
  containers:
  - name: nginx
    image: nginx:1.19
  nodeName: k8s-worker04  #将目标pod明确指定到k8s-worker04 上
```

```
[root@k8s-master01 k8s-demo]# vim pod-podaffinity-target.yaml
[root@k8s-master01 k8s-demo]# kubectl apply -f pod-podaffinity-target.yaml
pod/pod-podaffinity-target created
[root@k8s-master01 k8s-demo]# kubectl get pod/pod-podaffinity-target
NAME                     READY   STATUS    RESTARTS   AGE
pod-podaffinity-target   1/1     Running   0          12s
[root@k8s-master01 k8s-demo]# kubectl get pod/pod-podaffinity-target -owide
NAME                     READY   STATUS    RESTARTS   AGE   IP              NODE          NOMINATED NODE   READINESS GATES
pod-podaffinity-target   1/1     Running   0          16s   100.65.241.29   k8s-worker04   <none>           <none>
```

2) 创建pod-podaffinity-required.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-podaffinity-required

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    podAffinity:   #设置pod亲和性
      requiredDuringSchedulingIgnoredDuringExecution:  #硬限制
      -  labelSelector:
           matchExpressions: #匹配podenv的值在["xxx","yyy"]中的标签
           - key: podenv
             operator: In
```

```
            values: ["xxx","yyy"]
        topologyKey: kubernetes.io/hostname
```

调度失败，修改配置信息:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-podaffinity-required

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    podAffinity:   #设置pod亲和性
      requiredDuringSchedulingIgnoredDuringExecution:  #硬限制
      -  labelSelector:
          matchExpressions: #匹配podenv的值在["pro","yyy"]中的标签
          - key: podenv
            operator: In
            values: ["pro","yyy"]
        topologyKey: kubernetes.io/hostname
```

测试，pod被调度到k8s-worker04节点上

```
[root@k8s-master01 k8s-demo]# kubectl delete -f pod-podaffinity-required.yaml
pod "pod-podaffinity-required" deleted
[root@k8s-master01 k8s-demo]# kubectl apply -f pod-podaffinity-required.yaml
pod/pod-podaffinity-required created
[root@k8s-master01 k8s-demo]# kubectl get pod/pod-podaffinity-required  -owide
NAME                       READY   STATUS    RESTARTS   AGE   IP              NODE          NOMINATED NODE   READINESS GATES
pod-podaffinity-required   1/1     Running   0          7s    100.65.241.30   k8s-worker04   <none>           <none>
```

preferredDuringSchedulingIgnoredDuringExecution

创建pod-podaffinity-preferred.yaml，将pod调度到不满足调度条件的node上，看是否能成功

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-podaffinity-preferred
```

```
spec:
  containers: # 容器配置
    - name: nginx
      image: nginx:1.19
      imagePullPolicy: IfNotPresent
      ports:
        - name: nginx-port
          containerPort: 80
          protocol: TCP
  affinity: # 亲和性配置
    podAffinity: # Pod亲和性
      preferredDuringSchedulingIgnoredDuringExecution: #软限制
        - podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: podenv
                  operator: In
                  values:
                    - "xxx"
                    - "yyy"
            topologyKey: kubernetes.io/hostname
          weight: 1
```

pod在不满足给定调度条件情况下，任然可以调度。

修改values为pro后，重新部署，调度到k8s-worker04节点上

podAntiAffinity

podAntiAffinity主要实现以运行的Pod为参照，让新创建的Pod和参照的Pod不在一个区域的功能。

其配置方式和podAffinity一样。

创建pod-podantiaffinity-required.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-podantiaffinity-required
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.19
  affinity:   #亲和性设置
    podAntiAffinity:   #设置pod反亲和性
      requiredDuringSchedulingIgnoredDuringExecution:  #硬限制
      -  labelSelector:
           matchExpressions: #匹配env的值在["pro"]中的标签
           - key: podenv
             operator: In
             values: ["pro"]
         topologyKey: kubernetes.io/hostname
```

测试，不在一个节点上

3.3 污点和容忍调度

污点

前面的调度方式都是站在Pod的角度上，通过在Pod上添加属性，来确定Pod是否要调度到指定的Node上，其实我们也可以站在Node的角度上，通过在Node上添加污点属性，来决定是否运行Pod调度过来。

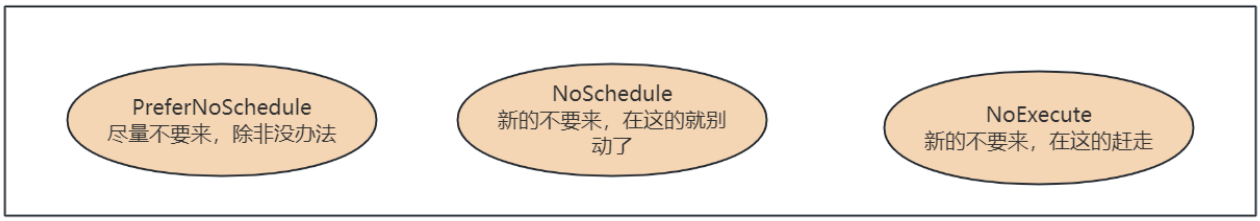
Node被设置了污点之后就和Pod之间存在了一种相斥的关系，进而拒绝Pod调度进来，甚至可以将已经存在的Pod驱逐出去。

污点的格式为：key=value:effect，key和value是污点的标签，effect描述污点的作用，支持如下三个选项：

1、PreferNoSchedule：Kubernetes将尽量避免把Pod调度到具有该污点的Node上，除非没有其他节点可以调度。

2、NoSchedule：Kubernetes将不会把Pod调度到具有该污点的Node上，但是不会影响当前Node上已经存在的Pod。

3、NoExecute：Kubernetes将不会把Pod调度到具有该污点的Node上，同时也会将Node上已经存在的Pod驱逐。



污点相关语法:

- 设置污点

```
kubectl taint node xxx key=value:effect
```

- 去除污点

```
kubectl taint node xxx key:effect-
```

- 去除所有污点

```
kubectl taint node xxx key-
```

- 查看指定节点上的污点

```
kubectl describe node 节点名称
```

k8s默认就会给Master节点添加一个污点标记，所以Pod就不会调度到Master节点上

使用示例

便于测试，停掉其他worker节点

1. 为k8s-worker01节点设置一个污点：tag=ms:PreferNoSchedule；然后创建pod1（pod1可以）
2. 修改为k8s-worker01节点设置一个污点：tag=ms:NoSchedule；然后创建pod2（pod1正常 pod2失败）
3. 修改为k8s-worker01节点设置一个污点：tag=ms:NoExecute；然后创建pod3（3个pod都失败）

1) 为k8s-worker01设置污点（PreferNoSchedule）

```
kubectl taint nodes k8s-worker01 tag=ms:PreferNoSchedule
```

2) 创建taint1(pod1)

```
kubectl run taint1 --image=nginx:1.19
```

3) 为k8s-worker01设置污点（取消PreferNoSchedule设置为NoSchedule）

```
kubectl taint nodes k8s-worker01 tag:PreferNoSchedule-
kubectl taint nodes k8s-worker01 tag=ms:NoSchedule
```

pod1无变化

4) 创建新的taint2（pod2）

```
kubectl run taint2 --image=nginx:1.19
```

发现新的pod无法running

kubectl describe pod taint2 查看详情

5）为k8s-worker01设置污点（取消NoSchedule，设置为NoExecute）

```
kubectl taint nodes k8s-worker01 tag:NoSchedule-
kubectl taint node k8s-worker01 tag=ms:NoExecute
```

6）pod1和pod2 都没了，创建一个taint3

```
kubectl run taint3 --image=nginx:1.19
```

新的也无法启动

容忍

上面介绍了污点的作用，我们可以在Node上添加污点用来拒绝Pod调度上来，但是如果就是想让一个Pod调度到一个有污点的Node上去，这时候应该怎么做？这就需要使用到容忍。

> 污点就是拒绝，容忍就是忽略，Node通过污点拒绝Pod调度上去，Pod通过容忍忽略拒绝。

容忍详细配置：

```
kubectl explain pod.spec.tolerations
......
FIELDS:
  key       # 对应着要容忍的污点的键，空意味着匹配所有的键
  value     # 对应着要容忍的污点的值
  operator  # key-value的运算符，支持Equal和Exists（默认）
  effect    # 对应污点的effect，空意味着匹配所有影响
  tolerationSeconds   # 容忍时间，当effect为NoExecute时生效，表示pod在Node上的停留时间
```

当operator为Equal的时候，如果Node节点有多个Taint，那么Pod每个Taint都需要容忍才能部署上去。

当operator为Exists的时候，有如下的三种写法：

- 容忍指定的污点，污点带有指定的effect：

```
  tolerations: # 容忍
    - key: "tag" # 要容忍的污点的key
      operator: Exists # 操作符
      effect: NoExecute # 添加容忍的规则，这里必须和标记的污点规则相同
```

- 容忍指定的污点，不考虑具体的effect：

```
  tolerations: # 容忍
    - key: "tag" # 要容忍的污点的key
      operator: Exists # 操作符
```

- 容忍一切污点（慎用）：

```
tolerations: # 容忍
    - operator: Exists # 操作符
```

在上面的污点示例中，已经给k8s-worker01打上了NoExecute的污点，此时任何Pod是调度不上去的，可以通过在Pod中添加容忍，将Pod调度上去。

创建pod-toleration.yaml文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-toleration

spec:
  containers:
  - name: nginx
    image: nginx:1.19
  tolerations:    #添加容忍
  - key: "tag"   #要容忍的污点的key
    operator: "Equal"   #操作符
    value: "ms"   #容忍的污点的value
    effect: "NoExecute"   #添加容忍的规则，这里必须和标记的污点规则相同
```

pod成功running