调度机制

调度框架

调度体系

学习目标

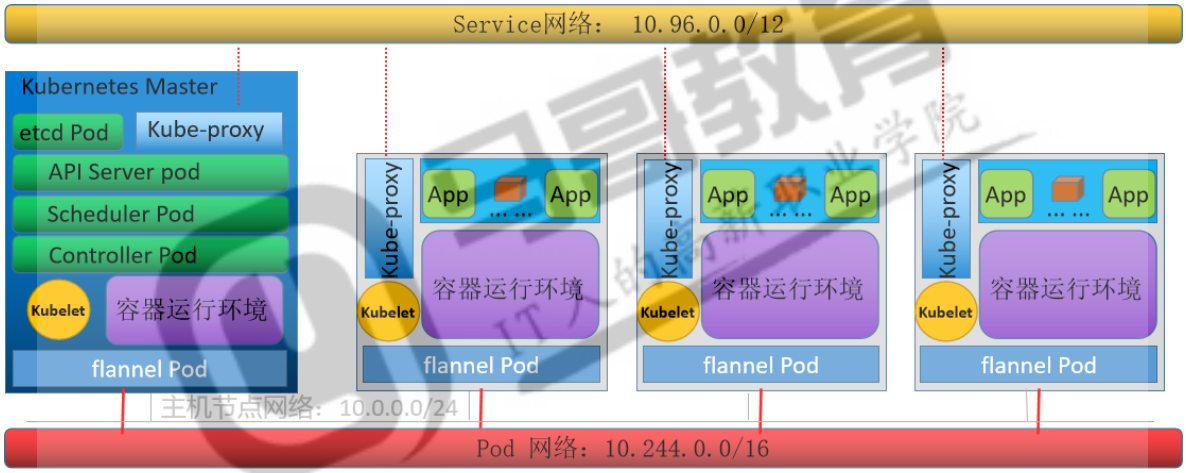
这一节，我们从 基础知识、细节梳理、小结 三个方面来学习。

基础知识

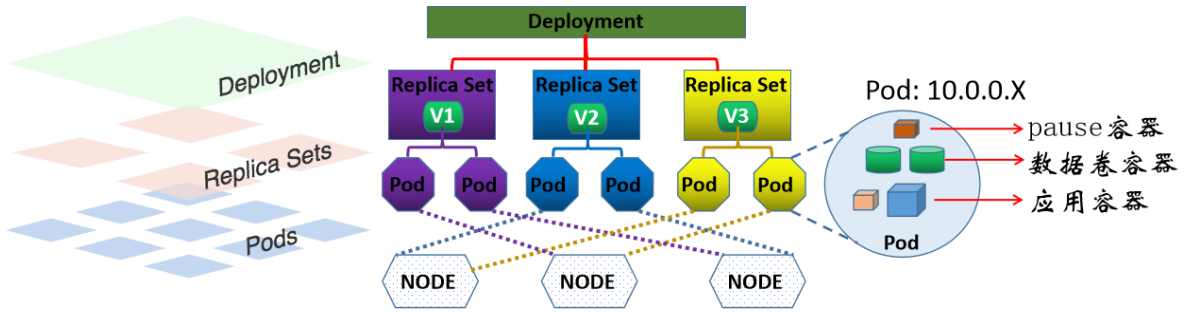
简介

> 到现在为止，我们对k8s已经有了一定程度的理解了。从最初我们第一次听说，Kubernetes是一个基于分布式架构实现的容器管理平台，尤其是对于资源的调度方面，分别从多个角度来实现相应的管控。

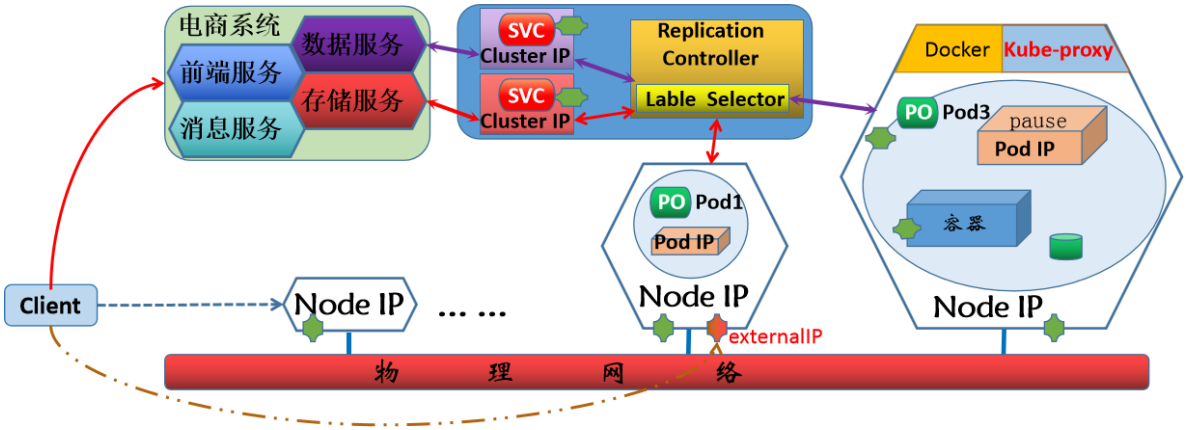
架构层面 - 集群的组成



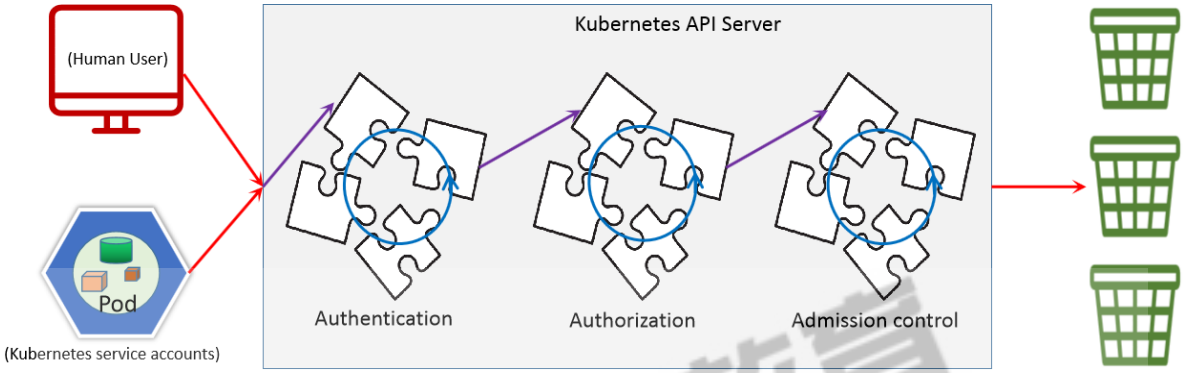
资源层面 - k8s集群资源的管控



网络层面 - k8s集群资源的访问

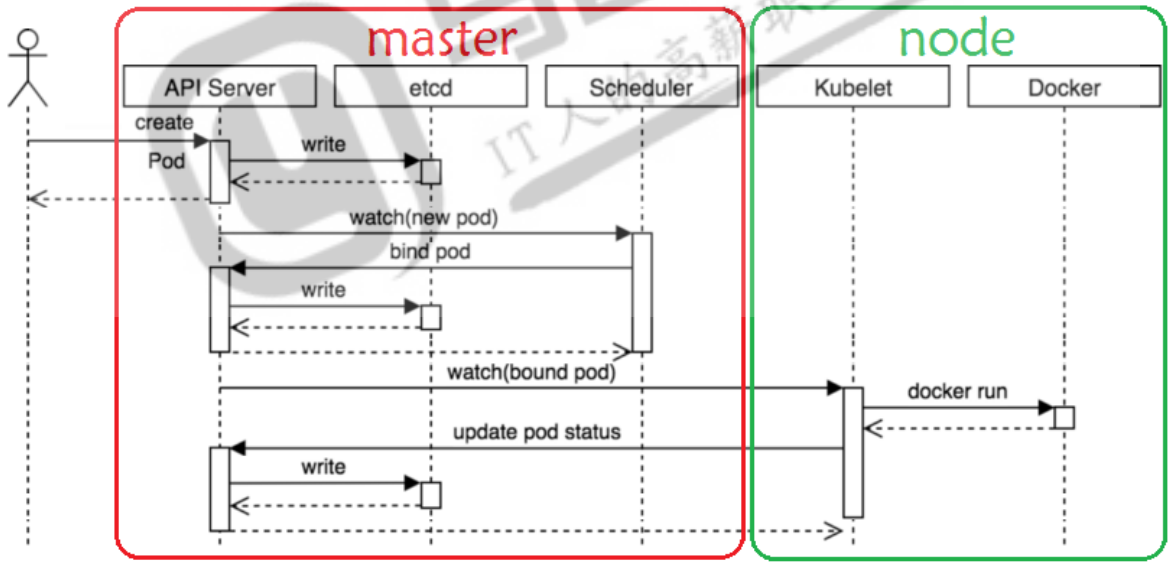


认证层面 - k8s集群资源的认证

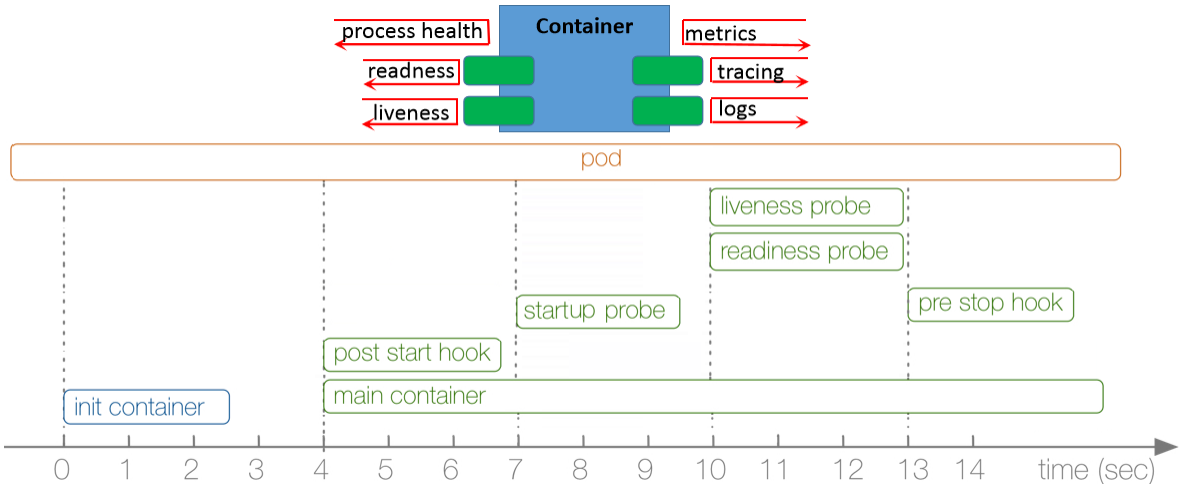


细节梳理

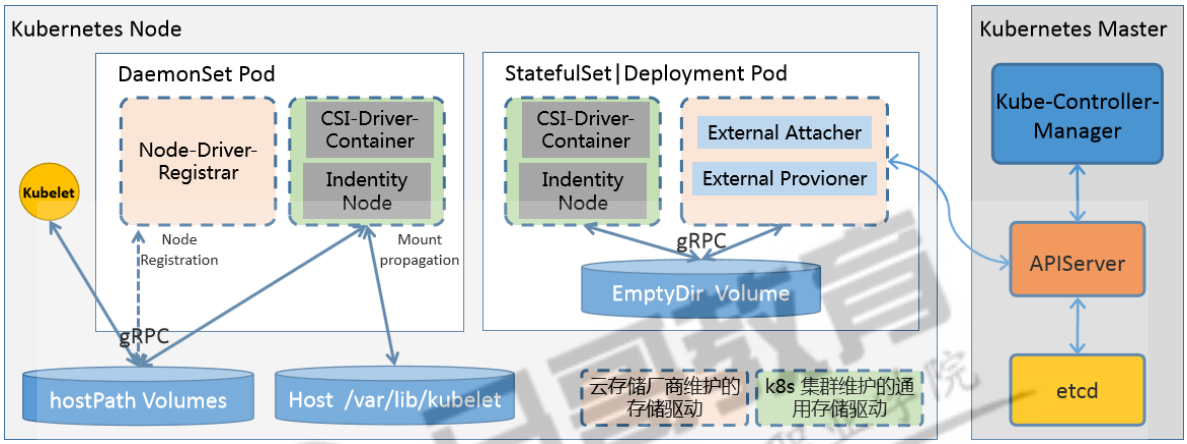
pod周期 - pod创建的完整流程



pod组成 - pod创建到成功，经历了什么



存储层面 - pod数据是如何整合的



创建层面 - 资源使用的数据同步

资源层面

角度	解析
资源配置项	Kubernetes技术已经对Pod做了相应的资源配额设置，目前主要包括两方面：CPU和内存
配额限制	Kubernetes中，对于每种资源的配额限定都需要两个参数：申请配额(Requests)和最大配额(Limits)
存活状态	pod层面主要是根据探针来实现的服务质量的检测：livenessProbe和readinessProbe。
对象分组	在kubernetes内部，尤其是pod层面，通过 标签和标签选择器 或者 nodeName 等机制实现资源的分组管理。
服务质量	在Kubernetes内部靠实时的监控各种资源对象的状态值，对比相应的"期望状态值"，然后由内部的组件做出自动的资源调整。

小结

调度框架

学习目标

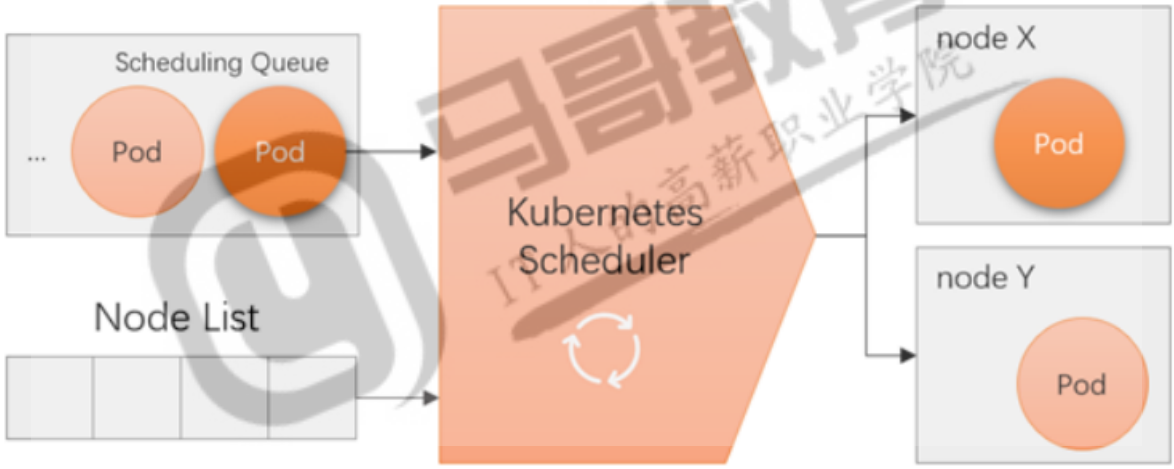
这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

前面的调度体系，全方位的梳理了一下，k8s资源调度相关的内容，但是最终的落脚点仍然要在资源上，而这些资源本质上都是节点上的资源。所以第一步就来了，如何把资源调度到对应的节点上呢？

kube-scheduler是 kubernetes 系统的核心组件之一，主要负责整个集群资源的调度功能，根据特定的调度算法和策略，将 Pod 调度到最优的工作节点上面去，从而更加合理、更加充分的利用集群的资源，这也是我们选择使用 kubernetes 一个非常重要的理由。如果一门新的技术不能帮助企业节约成本、提供效率，我相信是很难推进的。



队列产生的原因：任务优先级的确定。

单个对象：kubectl explain pod.spec.priority

优先级组：kubectl explain pod.spec.priorityClassName

调度过程

当Scheduler通过API server 的watch接口监听到新建Pod副本的信息后，它会检查所有符合该Pod要求的Node列表，开始执行Pod调度逻辑。调度成功后将Pod绑定到目标节点上。Scheduler在整个系统中承担了承上启下的作用，承上是负责接收创建的新Pod，为安排一个落脚的地（Node）,启下是安置工作完成后，目标Node上的kubelet服务进程接管后继工作，负责Pod生命周期的后半生。

具体来说，Scheduler的作用是将待调度的Pod安装特定的调度算法和调度策略绑定到集群中的某个合适的Node上，并将绑定信息传给API server 写入etcd中。整个调度过程中涉及三个对象，分别是：待调度的Pod列表，可以的Node列表，以及调度算法和策略。

调度功能

这个过程在我们看来好像比较简单，但在实际的生产环境中，需要考虑的问题就有很多了：

如何保证全部的节点调度的公平性？要知道并不是说有节点资源配置都是一样的

如何保证每个节点都能被分配资源？

集群资源如何能够被高效利用？

集群资源如何才能被最大化使用？

如何保证 Pod 调度的性能和效率？

用户是否可以根据自己的实际需求定制自己的调度策略？

考虑到实际环境中的各种复杂情况，kubernetes 的调度器采用插件化的形式实现，可以方便用户进行定制或者二次开发，我们可以自定义一个调度器并以插件形式和 kubernetes 进行集成。

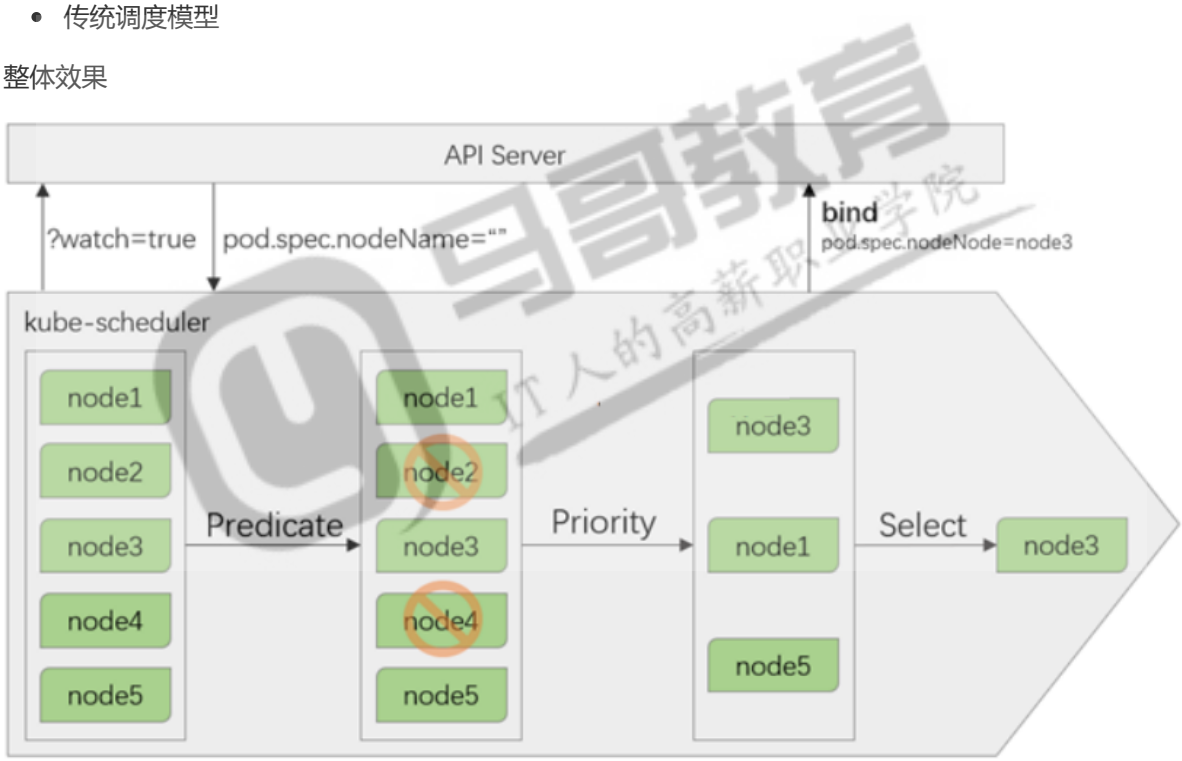
调度模型

k8s管理平台自动出现以来，其资源调度策略都是一个难啃的部分，而且一致在发展中。我们单从资源的调度框架的层面来分析的话，先后经历了两个调度的阶段：传统调度阶段、新型调度阶段。

调度模型

- 传统调度模型

整体效果



预选策略(predicate)

遍历nodelist，选择出符合要求的候选节点，过滤掉不满足条件的节点，k8s内置了多种预选规则供用户选择。

如没有Node符合Predicates策略规则，那该Pod就会被挂起，直到有Node能够满足。

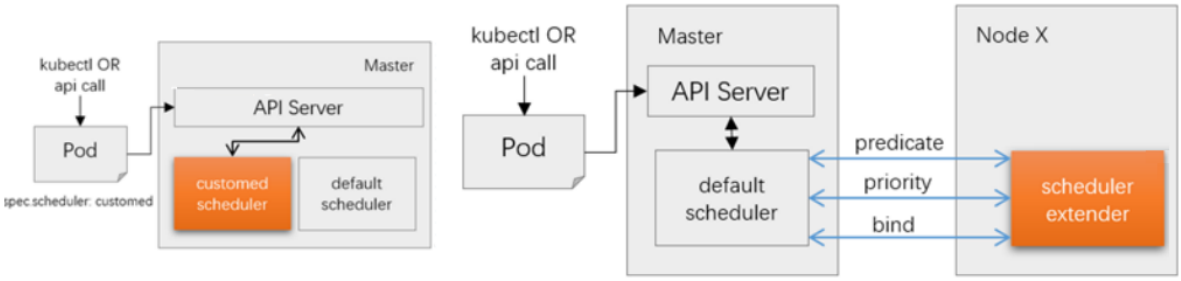
优选策略(priority)

在选择出符合要求的候选节点中，采用优选规则计算出每个节点的积分，最后选择得分最高的。

选定(select)

如果最高得分有好几个节点，select就会从中随机选择一个节点。

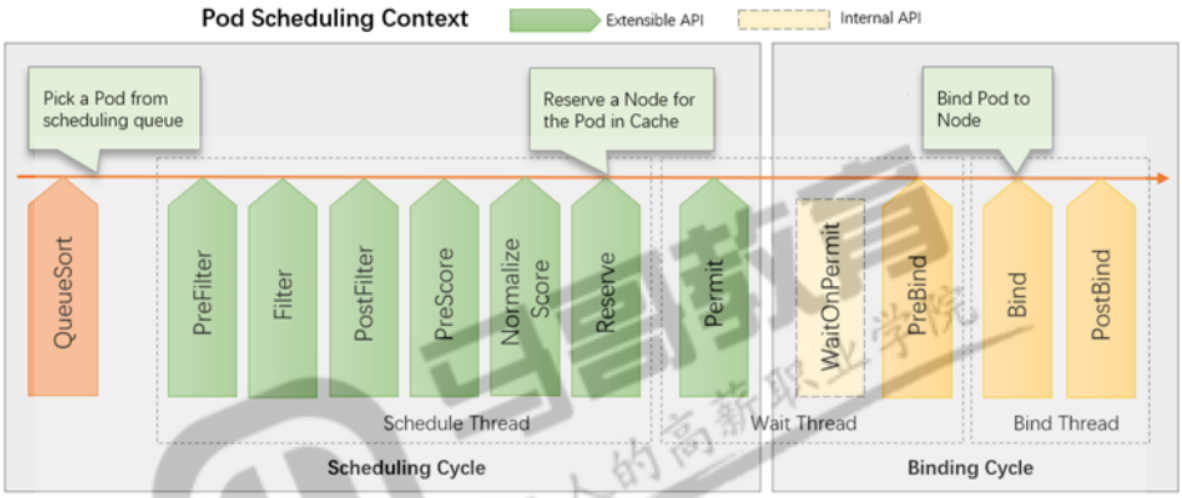
扩展支持



虽然k8s提供了非常多的预选策略函数和优选策略函数，但是其本身的特点也限制了更多灵活场景的功能扩充，所以就出现了两种扩展方案。
多调度器 - 通过自定义调度器的方式，让用户自己来选择使用哪一种调度策略(需要用户深度参与)
扩展接口 - 通过扩展的调度接口，将相关的扩展功能整合到当前的调度策略中(每次都要编译执行)

- 新型调度策略

调度框架



新型的调度框架是在传统调度逻辑的基础上，进行了整合再拆分。基本流程还是一样：队列-预选-评分-绑定，将这个基本流程划分为了三个部分：
任务队列 - 接收所有待分配的任务，任务本身会根据优先级合理调整队列的顺序。
调度循环 - 通过扩展接口的方式整合大量的预选机制和评分机制，同时为了方便扩展，预留了备用接口
绑定循环 - 先后通过预绑定队列(防止突发的高优先级任务出现)，然后进行正常的节点任务绑定。

vs 传统调度

1 所有的传统预选函数和优选函数，都变成了扩展插件的方式实现了动态的灵活绑定。
2 一切皆代码，所有的配置都可以通过代码的方式来进行实时调整，避免传统的再编译或用户强干预。

小结

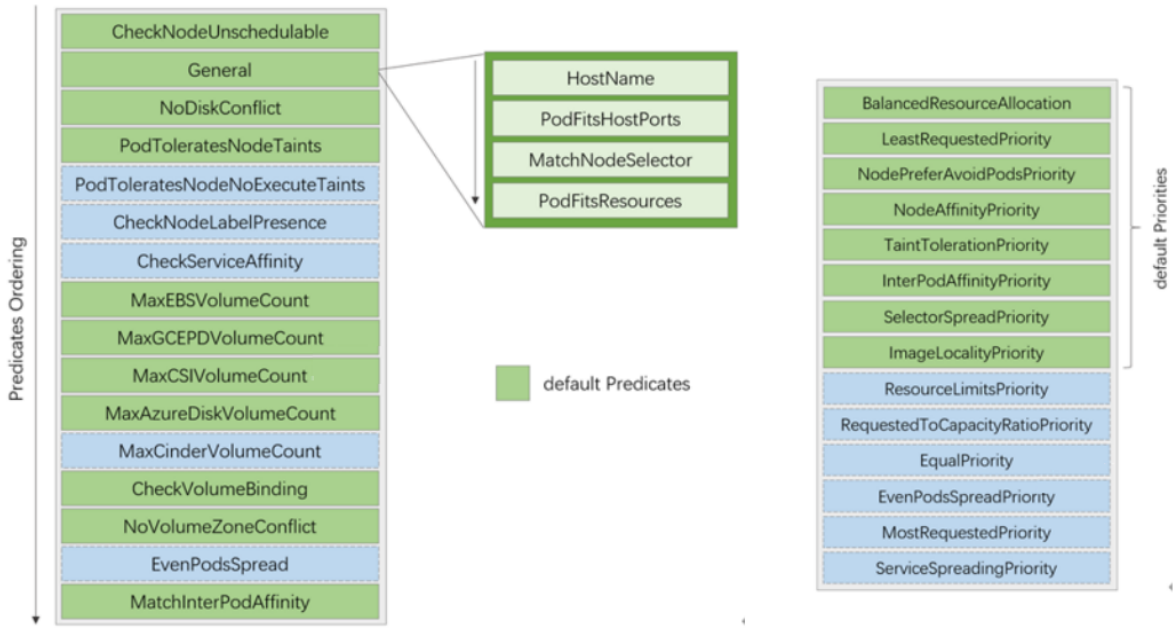
调度功能

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。

参数解析

常见的 预选参数和优选参数



无论是传统的调度功能还是新型的调度功能，对于各种功能函数来说，本质上的逻辑变化不大，只不过在实现细节上稍微有些不同。而且为了让调度的能力符合通用的使用，默认情况下，开启了一部分通用的策略，没有开启的策略可以通过配置参数的方式来进行扩展加载。

参数解析

参考资料

预选策略：
https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler/algorithm/predicates
优选函数：
https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler/algorithm/priorities

预选功能

在Kubernetes Scheduler上启用相应的预选函数才能实现相关调度机制的节点过滤需求，下面给出了这些于Kubernetes 各版本中支持的各预选函数虽然细节实现稍有不同，但是功能基本没有怎么变，其简要功能如下：
1）CheckNodeUnschedulable：检查节点是否被标识为Unschedulable，以及是否可将Pod调度于该类节点之上。
2）HostName：若Pod资源通过spec. nodeName明确指定了要绑定的目标节点，则节点名称与与该字段值相同的节点才会被保留。
3）PodFitsHostPorts：若Pod容器定义了ports.hostPort属性，该预选函数负责检查其值指定的端口是否已被节点上的其他容器或服务所占用，该端口已被占用的节点将被过滤掉。
4）MatchNodeSelector：若Pod资源规范上定义了spec.nodeSelector字段，则仅那些拥有匹配该标签选择器的标签的节点才会被保留。
5）NoDiskConflict：检查Pod对象请求的存储卷在此节点是否可用，不存在冲突则通过检查。

6）PodFitsResources：检查节点是否有足够资源（例如 CPU、内存和GPU等）满足Pod的运行需求；节点声明其资源可用容量，而Pod定义其资源需求（requests），于是调度器会判断节点是否有足够的可用资源运行Pod对象，无法满足则返回失败原因（例如，CPU或内存资源不足等）；调度器的评判资源消耗的标准是节点已分配资源量（各容器的requests值之和），而非其上的各Pod已用资源量，但那些在注解中标记为关键性（critical）的Pod资源则不受该预选函数控制。
7）PodToleratesNodeTaints：检查Pod的容忍度（spec.tolerations字段）是否能够容忍该节点上的污点（taints），不过，它仅关注具有NoSchedule和NoExecute两个效用标识的污点。
8）PodToleratesNodeNoExecuteTaints：检查Pod的容忍度是否能接纳节点上定义的NoExecute类型的污点。
9）CheckNodeLabelPresence：检查节点上某些标签的存在性，要检查的标签以及其可否存在则取决于用户的定义；在集群中的部署节点以regions/zones/racks类标签的拓扑方式编制，且基于该类标签对相应节点进行了位置标识时，预选函数可以根据位置标识将Pod调度至此类节点之上。
10）CheckServiceAffinity：根据调度的目标Pod对象所属的Service资源已关联的其他Pod对象的位置（所运行节点）来判断当前Pod可以运行的目标节点，其目的在于将同一Service对象的Pod放置在同一拓扑内（如同一个rack或zone）的节点上以提高效率。
11）MaxEBSVolumeCount：检查节点上已挂载的EBS存储卷数量是否超过了设置的最大值。
12）MaxGCEPDVolumeCount：检查节点上已挂载的GCE PD存储卷数量是否超过了设置的最大值，默认值为16。
13）MaxCSIVolumeCount：检查节点上已挂载的CSI存储卷数量是否超过了设置的最大值。
14）MaxAzureDiskVolumeCount：检查节点上已挂载的Azure Disk存储卷数量是否超过了设置的最大值，默认值为16。
15）MaxCinderVolumeCount：检查节点上已挂载的Cinder存储卷数量是否超过了设置的最大值。
16）CheckVolumeBinding：检查节点上已绑定和未绑定的PVC是否能满足Pod的存储卷需求，对于已绑定的PVC，此预选函数检查给定节点是否能兼容相应PV，而对于未绑定的PVC，预选函数搜索那些可满足PVC申请的可用PV，并确保它可与给定的节点兼容。
17）NoVolumeZoneConflict：在给定了存储故障域的前提下，检测节点上的存储卷是否可满足Pod定义的需求。
18）EvenPodsSpread：检查节点是否能满足Pod规范中topologySpreadConstraints字段中定义的约束以支持Pod的拓扑感知调度。
19）MatchInterPodAffinity：检查给定节点是否能满足Pod对象的亲和性或反亲和性条件，用于实现Pod亲和性调度或反亲和性调度。

优选功能

LeastRequestedPriority：
优先将Pod打散至集群中的各节点之上，以试图让各节点有着近似的计算资源消耗比例，适用于集群规模较少变动的场景；其分值由节点空闲资源与节点总容量的比值计算而来，即由CPU或内存资源的总容量减去节点上已有Pod对象需求的容量总和，再减去当前要创建的Pod对象的需求容量得到的结果除以总容量；CPU和内存具有相同权重，资源空闲比例越高的节点得分也就越高，其计算公式如为：(cpu((capacity - sum(requested)) * 10 / capacity) + memory((capacity - sum(requested)) * 10 / capacity))/ 2。

MostRequestedPriority：
与优选函数LeastRequestedPriority的评估节点得分的方法相似，但二者不同的是，当前函数将给予计算资源占用比例更大的节点以更高的得分，计算公式如为：(cpu((sum(requested)) * 10 / capacity) + memory((sum(requested)) * 10 / capacity))/ 2。该函数的目标在于优先让节点以满载的方式承载Pod资源，从而能够使用更少的节点数，因而较适用于节点规模可弹性伸缩的集群中以最大化地节约节点数量。

BalancedResourceAllocation：
以CPU和内存资源占用率的相近程度作为评估标准，二者越接近的节点权重越高。该优选函数不能单独使用，它需要和LeastRequestedPriority组合使用来平衡优化节点资源的使用状态，选择那些在部署当前Pod资源后系统资源更为均衡的节点。

ResourceLimitsPriority：
以是否能够满足Pod资源限制为评估标准，那些能够满足Pod对于CPU或（和）内存资源限制的节点将计入1分，节点未声明可分配资源或Pod未定义资源限制时不影响节点计分。

RequestedToCapacityRatio：
该函数允许用户自定义节点各类资源（例如CPU和内存等）的权重，以便提高大型集群中稀缺资源的利用率；该函数的行为可以通过名为requestedToCapacityRatioArguments的配置选项进行控制，它由shape和resources两个参数组成。

NodeAffinityPriority：
节点亲和调度机制，它根据Pod资源规范中的spec.nodeSelector来对给定节点进行匹配度检查，成功匹配到的条目越多则节点得分越高。不过，其评估过程使用PreferredDuringSchedulingIgnoredDuringExecution这一表示首选亲和的标签选择器。

ImageLocalityPriority：
镜像亲和调度机制，它根据给定节点上是否拥有运行当前Pod对象中的容器所依赖到的镜像文件来计算该节点的得分值。那些不具有该Pod对象所依赖到的任何镜像文件的节点得分为0，而那些存在相关镜像文件的各节点中，拥有被Pod所依赖到的镜像文件的体积之和越大的节点得分就会越高。

TaintTolerationPriority：
基于对Pod资源对节点的污点容忍调度偏好进行其优先级评估，它将Pod对象的tolerations列表与节点的污点进行匹配度检查，成功匹配的条目越多，则节点得分越低。

SelectorSpreadPriority：
尽可能分散Pod至不同节点上的调度机制，它首先查找标签选择器能够匹配到当前Pod标签的ReplicationController、ReplicaSet和StatefulSet等控制器对象，而后查找可由这类对象的标签选择器匹配到的现存各Pod对象及其所在的节点，而那些运行此类Pod对象越少的节点得分越高。简单来说，如其名称所示，此优选函数尽量把同一标签选择器匹配到的Pod资源打散到不同的节点上运行。

ServiceSpreadingPriority：
类似于SelectorSpreadPriority，它首先查找标签选择器能够匹配到当前Pod标签的Service对象，而后查找可由这类Service对象的标签选择器匹配到的现存各Pod对象及其所在的节点，而那些运行此类Pod对象越少的节点得分越高。

EvenPodsSpreadPriority：
用于将一组特定的Pod对象在指定的拓扑结构上进行均衡打散，打散条件定义在Pod对象的spec.topologySpreadConstraints字段上，它内嵌labelSelector指定标签选择器以匹配符合条件的Pod对象，使用topologyKey指定目标拓扑结构，使用maxSkew描述最大允许的不均衡数量，而无法满足指定的调度条件时的评估策略则由whenUnsatisfiable字段定义，它有两个可用取值，默认值DoNotSchedule表示不予调度，而ScheduleAnyway则表示以满足最小不均衡值的标准进行调度。

EqualPriority：
设定所有节点具有相同的权重1。

InterPodAffinityPriority：
遍历Pod对象的亲和性条目，并将那些能够匹配到给定节点的条目的权重相加，结果值越大的节点得分越高。

NodePreferAvoidPodsPriority：
此优选级函数权限默认为10000，它根据节点是否设置了注解信息scheduler.alpha.kubernetes.io/preferAvoidPods来计算其优选级。计算方式是，给定的节点无此注解信息时，其得分为10乘以权重10000，存在此注解信息时，对于那些由ReplicationController或ReplicaSet控制器管控的Pod对象的得分为0，其他Pod对象会被忽略（得最高分）。

调度梳理

根据上面对调度功能函数的了解，上面的大部分功能函数都是静态的函数，只管调用即可，只有个别的一些函数支持用户自定义的场景。

对于节点预选功能，主要是基于资源的各种限制来进行管控，我们可以通过不同的调度器配置来进行定制。而对于优选策略的配置，是我们比较关注的高阶调度。根据我们的梳理，这些高阶的调度主要包括如下几个方面

调度策略	解析
节点调度	为即将执行的任务，选择合理的节点
Pod调度	为即将执行的任务，分配合理的逻辑关联搭配
污点调度	在任务调度的时候，设定一些避免措施
拓扑调度	在任务已确定调度的前提下，合理的分配任务，实现资源的高效利用

调度思路

名称	解析
亲和 与 反亲和	满足条件后，就分配 或者 远离 指定的节点。
硬亲和 与 软亲和	不会遵循绝对的条件阈值，而是根据实际情况，做出倾向性策略 硬亲和会考量预选和优选策略，而软亲和仅考虑优选策略

小结

配置解析

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

我们知道，默认的调度策略有可能无法满足我们的需求，我们可以根据实际情况，定制自己的调度策略，然后整合到k8s的集群中。

配置格式

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
AlgorithmSource:                        # 指定调度算法配置源，v1alpha2版本起该配置进入废弃阶段
  Policy:                               # 基于调度策略的调度算法配置源
    File:                               # 文件格式的调度策略
      Path <string>:                    # 调度策略文件policy.cfg的位置
    ConfigMap:                          # configmap格式的调度策略
```

```
      Namespace <string>              # 调度策略configmap资源隶属的名称空间
      Name <string>                   # configmap资源的名称
  Provider <string>                   # 配置使用的调度算法的名称，例如DefaultProvider
LeaderElection: {}                    # 多kube-scheduler实例并在时使用的领导选举算法
ClientConnection: {}                  # 与API Server通信时提供给代理服务器的配置信息
HealthzBindAddress <string>           # 响应健康状态检测的服务器监听的地址和端口
MetricsBindAddress <string>           # 响应指标抓取请求的服务器监听地址和端口
DisablePreemption <bool>              # 是否禁用抢占模式，false表示不禁用
PercentageOfNodesToScore <int32>      # 需要过滤出的可用节点百分比
BindTimeoutSeconds  <int64>           # 绑定操作的超时时长，必须使用非负数
PodInitialBackoffSeconds  <int64>     # 不可调度Pod的初始补偿时长，默认值为1
PodMaxBackoffSeconds <int64>          # 不可调度Pod的最大补偿时长，默认为10
Profiles <[]string>                   # 加载的KubeSchedulerProfile配置列表，v1beta1支
持多个
Extenders <[]Extender>                # 加载的Extender列表
```

基本步骤

```
1 定制自定义的调度策略
2 绑定调度策略到集群
3 测试效果
```

简单实践

1 定制调度文件

```
准备文件目录
mkdir /etc/kubernetes/scheduler
vim /etc/kubernetes/scheduler/kubeschedulerconfiguration.yaml

定制资源调度策略
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
profiles:
- schedulerName: default-scheduler
- schedulerName: demo-scheduler
  plugins:
    filter:
      disabled:
      - name: NodeUnschedulable
    score:
      disabled:
      - name: NodeResourcesBalancedAllocation
        weight: 1
      - name: NodeResourcesLeastAllocated
        weight: 1
      enabled:
      - name: NodeResourcesMostAllocated
        weight: 5

配置解析：
    schedulerName: default-scheduler  表示，不影响正常的调度策略
    NodeResourcesBalancedAllocation   禁用了节点资源的平均分步
```

```
    NodeResourcesLeastAllocated         禁用了节点最少资源调度
    NodeResourcesMostAllocated          启用了所有资源最好都放在一个节点上。
```

2 应用配置文件

```
修改kube-scheduler的manifest文件
# vim /etc/kubernetes/manifests/kube-scheduler.yaml
...
spec:
  - command:
    - kube-scheduler
    - --authentication-kubeconfig=/etc/kubernetes/scheduler.conf
    - --authorization-kubeconfig=/etc/kubernetes/scheduler.conf
    - --config=/etc/kubernetes/scheduler/kubeschedulerconfiguration.yaml
    ...
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
      readOnly: true
    - mountPath: /etc/kubernetes/scheduler
      name: schedconf
      readOnly: true
  ...
  volumes:
  - hostPath:
      path: /etc/kubernetes/scheduler.conf
      type: FileOrCreate
    name: kubeconfig
  - hostPath:
      path: /etc/kubernetes/scheduler
      type: DirectoryOrCreate
    name: schedconf
status: {}

配置解析:
    --config 属性放在 kubeconfig 配置后面一行
    因为涉及到容器里面的目录，需要将宿主机里面的信息关联到容器里面
```

确认效果

```
保存manifest文件后，就可以直接来查看pod相关的信息了
kubectl get pod -n kube-system -w
```

3 基本测试

```
定制资源配置文件 01-scheduler-deployment-test.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pod-test
  template:
```

```
    metadata:
      labels:
        app: pod-test
    spec:
      schedulerName: demo-scheduler
      containers:
      - name: nginxpod-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
        imagePullPolicy: IfNotPresent
    属性解析：
        schedulerName 表示，我们采用哪种调度策略。

应用资源配置文件
kubectl apply -f 01-scheduler-deployment-test.yaml

查看效果
root@master1:~# kubectl get pod -o wide
NAME                    READY   STATUS          RESTARTS   AGE     IP       NODE
   ...
deployment-test-...5    0/1     ContainerCreating   0     19s     <none>
master1   ...
deployment-test-...q    0/1     ContainerCreating   0     19s     <none>
master1   ...
deployment-test-...c    0/1     ContainerCreating   0     19s     <none>
master1   ...
结果显示：
    所有的资源都被调度到了同一个节点上了。
```

小结

```

```

调度实践

节点调度

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

```
关于资源对于节点的调度来说，主要有两种场景：
    指定节点 - 根据节点的标签，直接将应用部署到指定的节点
    节点亲和 - 根据任务的配置倾向，选择合适的节点来进行资源的配置
```

对于节点的调度主要有以下四种场景：
节点亲和
    kubectl explain pod.spec.affinity.nodeAffinity
节点软亲和
    - preferredDuringSchedulingIgnoredDuringExecution
节点硬亲和
    - requiredDuringSchedulingIgnoredDuringExecution

注意：
    这些调度策略，仅在调度时候有用，一旦调度成功后，节点状态无法满足调度要求，也不会影响正常运行的pod

简单实践

实践1 - 调度应用到指定的节点上

定制资源配置文件
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodename
spec:
  nodeName: node2
  containers:
  - name: demoapp
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
```
配置解析：
    nodeName 将应用调度到node2节点主机上

应用资源配置文件
```
kubectl apply -f 02-scheduler-pod-nodename.yaml
```

查看效果
```
# kubectl get pod pod-nodename -o wide
```

```
root@master1:~/scheduler# kubectl get pod pod-nodename -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES
pod-nodename   1/1     Running   0          11s   10.244.2.3   node2   <none>           <none>
```

实践2 - 调度应用到特定的节点上

定制资源配置文件
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeselector
spec:
  containers:
  - name: demoapp
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  nodeSelector:
    node: ssd
```
配置解析：
    将应用调度到包含node标签，并且值为ssd的节点主机上

应用资源配置文件

```
kubectl apply -f 03-scheduler-pod-nodeselector.yaml

查看效果
# kubectl get node --show-labels   | grep ssd
# kubectl get pod pod-nodename -o wide
```

```
root@master1:~/scheduler# kubectl get pod pod-nodeselector  -o wide
NAME              READY   STATUS    RESTARTS   AGE   IP       NODE     NOMINATED NODE   READINESS GATES
pod-nodeselector  0/1     Pending   0          20s   <none>   <none>   <none>           <none>
root@master1:~/scheduler# kubectl describe pod pod-nodeselector
Name:         pod-nodeselector
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  45s   default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affini
ty/selector.
```

结果显示：
    由于没有适配的节点，所有当前的pod状态是 pending

为node2节点添加标签
kubectl label node node2 node=ssd

```
root@master1:~/scheduler# kubectl get pod pod-nodeselector  -o wide
NAME              READY   STATUS    RESTARTS   AGE     IP           NODE    NOMINATED NODE   READINESS GATES
pod-nodeselector  1/1     Running   0          3m52s   10.244.2.4   node2   <none>           <none>
root@master1:~/scheduler# kubectl describe pod pod-nodeselector
Name:         pod-nodeselector
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  3m56s  default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affin
ity/selector.
  Warning  FailedScheduling  2m30s  default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affin
ity/selector.
  Normal   Scheduled         12s    default-scheduler  Successfully assigned default/pod-nodeselector to node2
  Normal   Pulled            8s     kubelet            Container image "10.0.0.19:80/mykubernetes/pod_test:v0.1" alread
y present on machine
  Normal   Created           8s     kubelet            Created container demoapp
  Normal   Started           7s     kubelet            Started container demoapp
```

结果显示
    一旦有节点满足需求，pod自然就被调度到指定的node节点上了

收尾动作
kubectl delete -f pod-nodeselector.yaml
kubectl label node node2 node-

注意：

小结

节点调度进阶

学习目标

这一节，我们从 硬亲和、软亲和、进阶实践、小结 三个方面来学习。

硬亲和

- 属性解析

属性解析

```
kubectl explain pod.spec.affinity.nodeAffinity
    - requiredDuringSchedulingIgnoredDuringExecution 硬亲和性  必须满足亲和性。
        nodeSelectorTerms:  节点选择主题
            matchExpressions  匹配表达式,可用大量运算符:
                In                  label 的值在某个列表中
                NotIn               label 的值不在某个列表中
                Gt                  label 的值大于某个值
                Lt                  label 的值小于某个值
                Exists              某个 label 存在
                DoesNotExist        某个 label 不存在
            matchFields  匹配字段,可以不定义标签值
```

- 简单实践

需求

```
只要结点包含env标签值是dev 或者 test的时候，才允许部署pod
```

```
资源文件内容 04-scheduler-pod-node-required-affinity.yaml
apiVersion: v1
kind: Pod
metadata:
  name: node-required-affinity
spec:
  containers:
  - name: demoapp
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env
            operator: In
            values:
            - dev
            - test
应用资源定义文件
kubectl apply -f 04-scheduler-pod-node-required-affinity.yaml

查看效果
kubectl get pod node-required-affinity
```

```
root@master1:~/scheduler# kubectl get pod node-required-affinity
NAME                    READY   STATUS    RESTARTS   AGE
node-required-affinity   0/1     Pending   0          9s
root@master1:~/scheduler# kubectl describe pod node-required-affinity
Name:         node-required-affinity
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  28s   default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affini
ty/selector.
```

```
任一结点添加标签
kubectl label node node1 env=dev
```

```
root@master1:~/scheduler# kubectl get pod node-required-affinity -o wide
NAME                      READY   STATUS    RESTARTS   AGE     IP           NODE    NOMINATED NODE   READINESS GATES
node-required-affinity    1/1     Running   0          2m31s   10.244.1.2   node1   <none>           <none>
root@master1:~/scheduler# kubectl describe pod node-required-affinity
Name:         node-required-affinity
Events:
  Type     Reason            Age    From               Message
  ----     ------            ----   ----               -------
  Warning  FailedScheduling  2m36s  default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affin
ity/selector.
  Warning  FailedScheduling  78s    default-scheduler  0/3 nodes are available: 3 node(s) didn't match Pod's node affin
ity/selector.
  Normal   Scheduled         18s    default-scheduler  Successfully assigned default/node-required-affinity to node1
  Normal   Pulled            14s    kubelet            Container image "10.0.0.19:80/mykubernetes/pod_test:v0.1" alread
y present on machine
  Normal   Created           13s    kubelet            Created container demoapp
  Normal   Started           8s     kubelet            Started container demoapp
```

结果显示：

只有节点上存在符合标签后，该资源才会被创建到指定结点

收尾动作

```
kubectl delete -f 04-scheduler-pod-node-required-affinity.yaml
kubectl label node node1 env-
```

软亲和

- 属性解析

属性解析

```
kubectl explain pod.spec.affinity.nodeAffinity
    - preferredDuringSchedulingIgnoredDuringExecution 软亲和性 能满足最好，不满足也没关系。
        preference 优先级
            matchExpressions
            matchFields
        weight 权重值，1-100
            对于满足所有调度要求的每个节点，调度程序将通过迭代此字段的元素计算总和
            并在节点与对应的节点匹配时将“权重”添加到总和。
注意：
    与硬亲和属性不同，这里需要注意的是软亲和属性是一个列表对象，而preference不是一个列表项了。
```

实践1 - 只要结点包含env标签值是dev 或者 test的时候，才允许部署pod

```
资源文件内容 05-scheduler-pod-node-preferred-affinity.yaml
apiVersion: v1
kind: Pod
metadata:
  name: node-preferred-affinity
spec:
  containers:
  - name: demoapp
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 50
        preference:
          matchExpressions:
          - key: env
            operator: In
            values:
```

```
            - test
      - weight: 20
        preference:
          matchExpressions:
          - key: env
            operator: In
            values:
            - dev
```

注意：

软亲和的权重配置是与硬亲和的区别所在。

准备节点标签

```
kubectl label node node2 env=test
kubectl label node node1 env=dev
```

应用资源定义文件

```
kubectl apply -f 05-scheduler-pod-node-preferred-affinity.yaml
```

查看效果

```
kubectl get pod node-preferred-affinity
```

```
root@master1:~/scheduler# kubectl get pod node-preferred-affinity -o wide
NAME                       READY   STATUS    RESTARTS   AGE    IP            NODE    NOMINATED NODE   READINESS GATES
node-preferred-affinity    1/1     Running   0          113s   10.244.2.5    node2   <none>           <none>
root@master1:~/scheduler# kubectl describe  pod node-preferred-affinity
Name:         node-preferred-affinity
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  2m4s  default-scheduler  Successfully assigned default/node-preferred-affinity to node2
  Normal  Pulled     2m    kubelet            Container image "10.0.0.19:80/mykubernetes/pod_test:v0.1" already present
 on machine
  Normal  Created    119s  kubelet            Created container demoapp
  Normal  Started    116s  kubelet            Started container demoapp
```

结果显示：

由于 node2上的标签权重较高，所以就交给了node2

清除node2节点标签，后，重启资源对象

```
kubectl label node node2 env-
kubectl delete -f 05-scheduler-pod-node-preferred-affinity.yaml
kubectl apply -f 05-scheduler-pod-node-preferred-affinity.yaml
kubectl get pod -o wide
```

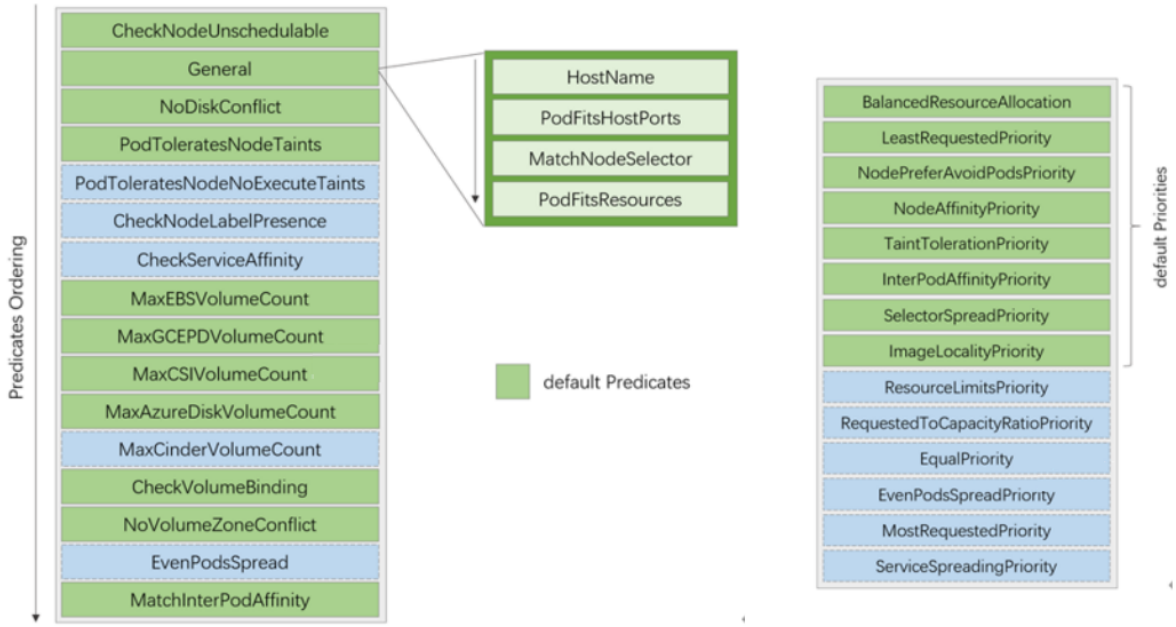
结果显示：

这个时候，资源调度到了node1上。

进阶实践

简介

根据我们之前对调度策略的了解，默认的的调度策略有一个优先级的筛选问题。
比如：即时我们的标签满足需求，如果资源需求不满足的话，仍然不会去调度资源。



对于我们之前实践的 预选和优选的问题，我们知道 预选是强制性的，一旦满足不了，就不再走下去了，而优选，仅仅是选择一个最好的，前提是 预选必须通过。

实践

资源文件内容 06-scheduler-pod-node-resourcefits-affinity.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-resourcefits-affinity
spec:
  replicas: 2
  selector:
    matchLabels:
      app: podtest
  template:
    metadata:
      labels:
        app: podtest
    spec:
      containers:
      - name: podtest
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            cpu: 2
            memory: 2Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: env
                operator: Exists
```

注意：

软亲和的权重配置是与硬亲和的区别所在。

准备节点标签

```
kubectl label node node2 env=test
kubectl label node node1 env=dev
```

应用资源定义文件

```
kubectl apply -f 06-scheduler-pod-node-resourcefits-affinity.yaml
```

查看效果

```
kubectl get pod node-resourcefits-affinity-67f5bff49c-zbqrx
```

```
root@master1:~/scheduler# kubectl apply -f pod-node-resourcefits-affinity.yaml
deployment.apps/node-resourcefits-affinity created
root@master1:~/scheduler# kubectl get pod
NAME                                        READY   STATUS    RESTARTS   AGE
node-resourcefits-affinity-576d8dcf5c-8j8zc   0/1     Pending   0          4s
node-resourcefits-affinity-576d8dcf5c-d6vnb   0/1     Pending   0          4s
root@master1:~/scheduler# kubectl describe pod node-resourcefits-affinity-576d8dcf5c-d6vnb
Name:           node-resourcefits-affinity-576d8dcf5c-d6vnb
Events:
  Type     Reason            Age   From              Message
  ----     ------            ----  ----              -------
  Warning  FailedScheduling  19s   default-scheduler  0/3 nodes are available: 1 Insufficient cpu, 2 node(s) didn't mat
ch Pod's node affinity/selector.
```

结果显示：

节点资源不足，所以无法正常的资源调度

查看主机资源

```
root@master1:~/scheduler# kubectl describe node node1
Name:               node1
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                650m (65%)   0 (0%)
  memory             400Mi (13%)  0 (0%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
```

调配资源限制

```
    ...
        resources:
          requests:
            cpu: 0.2
            memory: 200Mi
```

重新应用资源配置

```
kubectl apply -f 06-scheduler-pod-node-resourcefits-affinity.yaml
```

查看效果

```
kubectl get pod -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide
NAME                                        READY   STATUS    RESTARTS   AGE    IP           NODE    NOMINATED NODE
 READINESS GATES
node-resourcefits-affinity-67f5bff49c-qjtsj   1/1     Running   0          79s    10.244.1.6   node1   <none>
 <none>
node-resourcefits-affinity-67f5bff49c-zbqrx   1/1     Running   0          119s   10.244.1.5   node1   <none>
 <none>
```

小结

pod调度

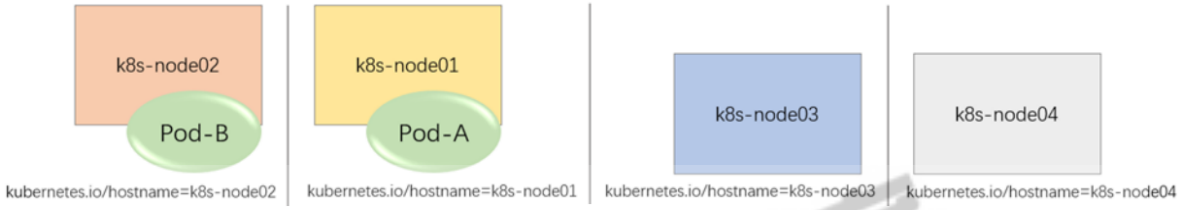
学习目标

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

场景需求

```
所谓的pod调度，主要说的是pod彼此之间的亲和性，也就是说，哪些pod应该在一起。
    比如：我们的k8s集群的节点分布在不同的区域或者不同的机房
        当服务A和服务B需要高效的交流数据的话，要求部署在同一个区域或者同一机房的时候。
        当服务A需要做冗余操作，那么多个服务A必须在不同的位置
```



属性解析

```
kubectl explain pod.spec.affinity.podAffinity
    requiredDuringSchedulingIgnoredDuringExecution -- 硬亲和性:
        labelSelector    选择跟那组Pod亲和，前提得知道如何判断
        namespaces       选择哪个命名空间进行条件匹配
        topologyKey          指定节点上的哪个键，这是一个必选项
        注意：这三个条件是一个逻辑与的关系
    preferredDuringSchedulingIgnoredDuringExecution -- 软亲和性:
        podAffinityTerm 与权重关联的亲和选项，这是一个必选项
            labelSelector
            namespaces       表示仅在指定的命名空间中查找
            topologyKey      在指定节点上的标签关键字
        weight               权重，这是一个必选项
```

简单实践

实践1 - 硬亲和，创建两个在不同主机上的pod，然后亲和pod和标签env=test的pod部署在一起

```
两个节点准备标签
kubectl label node node2 env=test
kubectl label node node1 env=dev

创建基础资源配置
for env in {dev,test}
do
cat >> 07-scheduler-pod-affinity-base.yaml <<-EOF
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-$env
  labels:
    env: $env
```

```
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  nodeSelector:
    env: $env
EOF
done
```

应用资源配置文件

```
kubectl apply -f 07-scheduler-pod-affinity-base.yaml
```

确认效果

```
kubectl get pod -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE    NOMINATED NODE   READINESS GATES
pod-dev    1/1     Running   0          10s   10.244.1.8    node1   <none>           <none>
pod-test   1/1     Running   0          10s   10.244.2.7    node2   <none>           <none>
```

创建资源配置文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-affinity
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - {key: env, operator: In, values: ["test"]}
        namespaces: ["default"]
        topologyKey: kubernetes.io/hostname
```

属性解析：

属性详解：

topologyKey的来源：

```
]# kubectl get nodes -o yaml | grep kubernetes.io/hostname
      kubernetes.io/hostname: master
      kubernetes.io/hostname: node1
      kubernetes.io/hostname: node2
```

其实指定 kubernetes.io/hostname 和 主机名 效果是一样的，只不过一个是自动获取，一个是手工指定

应用资源配置文件

```
kubectl apply -f 08-scheduler-pod-affinity-required.yaml
```

查看效果

```
kubectl get pod -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP            NODE    NOMINATED NODE   READINESS GATES
pod-affinity  1/1     Running   0          17s     10.244.2.8    node2   <none>           <none>
pod-dev       1/1     Running   0          4m15s   10.244.1.8    node1   <none>           <none>
pod-test      1/1     Running   0          4m15s   10.244.2.7    node2   <none>           <none>
```

结果显示：

由于是硬亲和状态，所以，直接去找对应的节点了

实践2 - 在多个满足条件的pod上，选择合适的节点

创建资源配置文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-affinity
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    podAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 60
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - {key: env, operator: In, values: ["dev"]}
          topologyKey: kubernetes.io/hostname
      - weight: 30
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - {key: env, operator: In, values: ["test"]}
          topologyKey: kubernetes.io/hostname
```

应用资源配置文件

```
kubectl apply -f 09-scheduler-pod-affinity-preferred.yaml
```

查看效果

```
kubectl get pod --show-labels -o wide
```

```
root@master1:~/scheduler# kubectl get pod --show-labels -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES   LABELS
pod-affinity   1/1     Running   0          78s   10.244.1.9   node1   <none>           <none>            <none>
pod-dev        1/1     Running   0          12m   10.244.1.8   node1   <none>           <none>            env=dev
pod-test       1/1     Running   0          12m   10.244.2.7   node2   <none>           <none>            env=test
```

结果显示：

优先选择的是 node1节点上的pod

删除pod-dev 后，重新执行pod-affinity对象

```
kubectl delete pod pod-dev
kubectl delete -f 09-scheduler-pod-affinity-preferred.yaml
kubectl apply -f 09-scheduler-pod-affinity-preferred.yaml
```

查看效果

```
kubectl get pod --show-labels -o wide
```

```
root@master1:~/scheduler# kubectl get pod --show-labels -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES   LABELS
pod-affinity   1/1     Running   0          22s   10.244.2.9   node2   <none>           <none>            <none>
pod-test       1/1     Running   0          20m   10.244.2.7   node2   <none>           <none>            env=test
```

进阶实践

实践3 - pod必须把数据存储到redis中，必须在一起

```
定义资源配置文件 10-scheduler-pod-affinity-redis-required.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: 10.0.0.19:80/mykubernetes/redis:6.2.5
        imagePullPolicy: IfNotPresent
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-affinity-required
spec:
  replicas: 4
  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:v0.1
        imagePullPolicy: IfNotPresent
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - {key: app, operator: In, values: ["redis"]}
            topologyKey: kubernetes.io/hostname
```

实践4 - pod必须把数据存储到redis中，原则上最好在一起，实在不行的话，划分到其他节点也可以

```
定义资源配置文件 11-scheduler-pod-affinity-redis-preferred.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-preferred
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      nodeName: node1
      containers:
      - name: redis
        image: 10.0.0.19:80/mykubernetes/redis:6.2.5
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            cpu: 300m
            memory: 512Mi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-affinity-preferred
spec:
  replicas: 4
  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:v0.1
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            cpu: 300m
            memory: 500Mi
      affinity:
        podAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - {key: app, operator: In, values: ["redis"]}
              topologyKey: kubernetes.io/hostname
          - weight: 50
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - {key: app, operator: In, values: ["redis"]}
              topologyKey: env
```

属性解析：

如果第一条件满足不了，那就用第二条件，第二条件不满足，就用其他条件

```
注意：
    这种方式，有可能会受到其他调度算法的影响。
```

小结

```
```

Pod调度进阶

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

```
所谓的反亲和，其实就是满足条件的话，就离这个pod远远的，从此不见面。与亲和正好相反
```

属性解析

```
kubectl explain pod.spec.affinity.podAntiAffinity
    requiredDuringSchedulingIgnoredDuringExecution -- 硬亲和性：
        labelSelector       选择跟那组Pod亲和，前提得知道如何判断
        namespaces          选择哪个命名空间进行条件匹配
        topologyKey         指定节点上的哪个键，这是一个必选项
        注意：这三个条件是一个逻辑与的关系
    preferredDuringSchedulingIgnoredDuringExecution -- 软亲和性：
        podAffinityTerm 与权重关联的亲和选项，这是一个必选项
            labelSelector
            namespaces
            topologyKey
        weight              权重，这是一个必选项

注意：Pod反亲和性场景，当应用服务A和数据库服务B要求尽量不要在同一台节点上的时候。
```

简单实践

实践1 - 硬性反亲和性，不要两个pod放在一起

```
准备工作
kubectl apply -f 07-scheduler-pod-affinity-base.yaml
检查效果
kubectl get pod
```

```
创建资源配置文件
apiVersion: v1
kind: Pod
metadata:
  name: pod-antiaffinity
spec:
  containers:
  - name: pod-test
```

```
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - {key: env, operator: In, values: ["dev"]}
        topologyKey: kubernetes.io/hostname
```

应用资源配置文件

```
kubectl apply -f 12-scheduler-pod-anaffinity-required.yaml
```

查看效果

```
kubectl get pod --show-labels -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide --show-labels
NAME              READY   STATUS    RESTARTS   AGE    IP             NODE    NOMINATED NODE   READINESS GATES   LABELS
pod-antiaffinity  1/1     Running   0          11s    10.244.2.11    node2   <none>           <none>            <none>
pod-dev           1/1     Running   0          3m8s   10.244.1.17    node1   <none>           <none>            env=dev
pod-test          1/1     Running   0          48m    10.244.2.7     node2   <none>           <none>            env=tes
t
```

实践2 - 软性反亲和性，最好不要哪些pod放在一起

创建资源配置文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-antiaffinity
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - {key: env, operator: In, values: ["dev"]}
            topologyKey: kubernetes.io/hostname
        - weight: 50
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - {key: env, operator: In, values: ["test"]}
            topologyKey: kubernetes.io/hostname
```

应用资源配置文件

```
kubectl apply -f 13-scheduler-pod-anaffinity-preferred.yaml
```

查看效果

```
kubectl get pod --show-labels -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide --show-labels
NAME              READY   STATUS              RESTARTS   AGE   IP            NODE      NOMINATED NODE   READINESS GATE
S   LABELS
pod-antiaffinity  0/1     ContainerCreating   0          11s   <none>        master1   <none>           <none>
    <none>
pod-dev           1/1     Running             0          10m   10.244.1.17   node1     <none>           <none>
    env=dev
pod-test          1/1     Running             0          56m   10.244.2.7    node2     <none>           <none>
    env=test
```

结果显示：

所有满足条件的都不能部署了。

小结

污点调度

学习目标

这一节，我们从 基础知识、属性解析、小结 三个方面来学习。

基础知识

需求

我们之前所学的所有调度策略，都是基于pod的属性来选择我们新的pod资源应该如何创建，而实际的生产角度上，往往会出现基于node节点的属性来选择是否让新的pod资源进行创建，但是对于节点来说，没有所谓的节点反亲和性，但是有一种类似的策略：污点和容忍度。

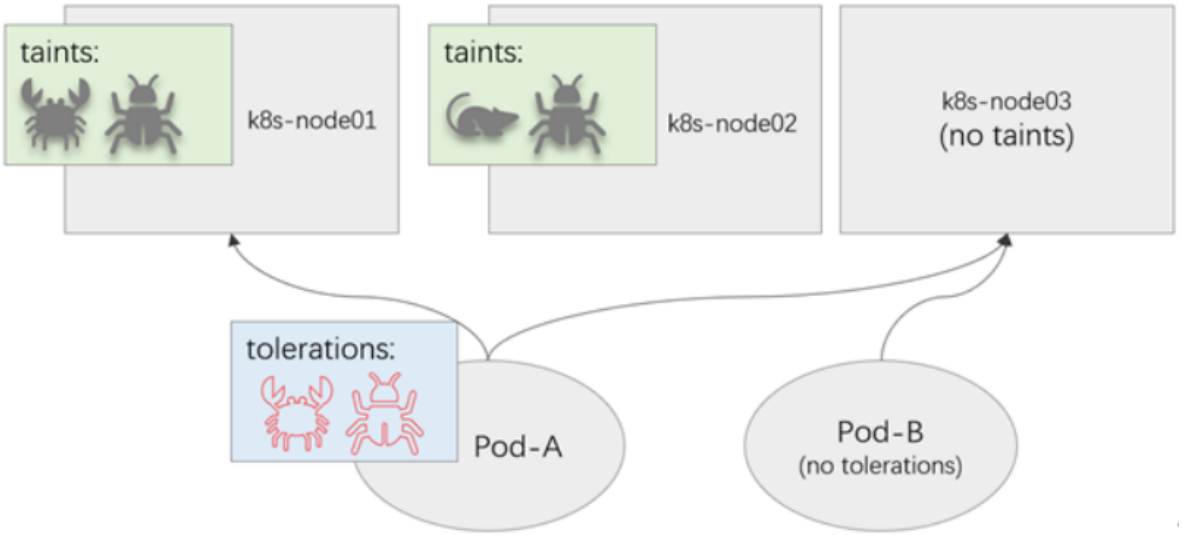
术语解析

术语	解析
污点(taints)	是定义在node节点上的键值属性数据。主要作用是让节点拒绝pod，尤其是不符合node规则的pod。
容忍度(tolerations)	是定义在Pod上的键值属性数据，用于配置可容忍的污点，调度器将其调度至容忍污点的节点上或无污点的节点

应用

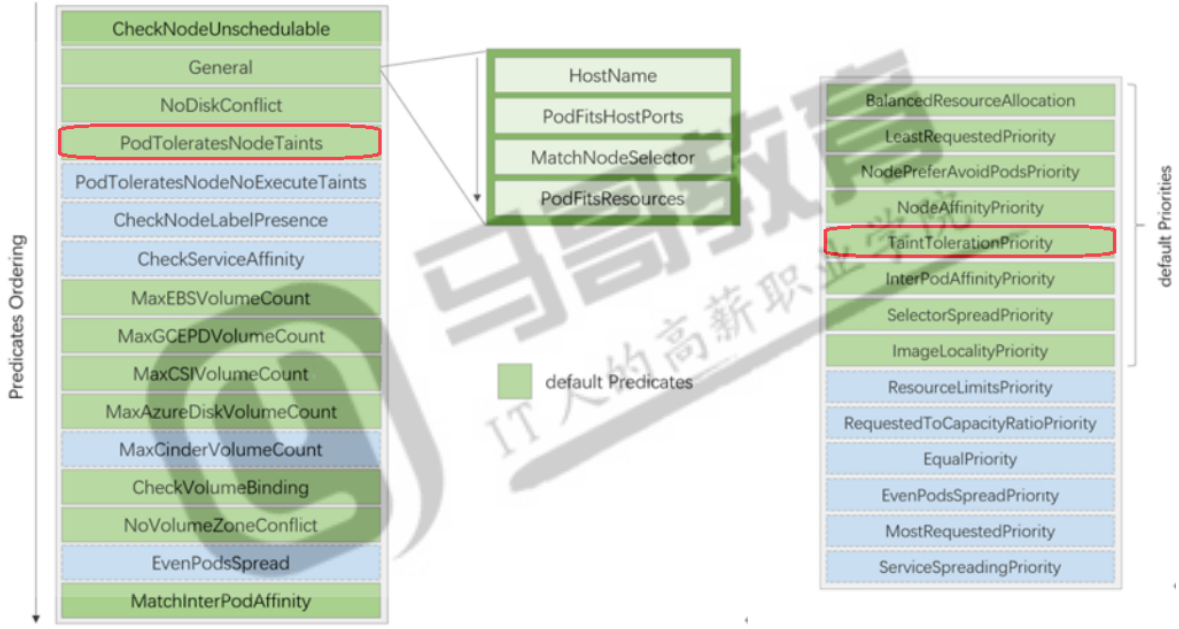
Taint(污点)和 Toleration(容忍)是相互配合的，可以用来避免 pod 被分配到不合适的节点上,每个节点上都可以应用一个或多个 taint ，这表示对于那些不能容忍这些 taint 的 pod，是不会被该节点接受的。

比如：A节点是集群中很重要的节点，不要将无关的pod来我这里运行，那么我们就可以在A上面添加一些污点标识。



作用来源

我们可以使用PodToleratesNodeTaints预选策略和TaintTolerationPriority优选函数完成该机制



污点功能

Kubernetes自1.6起支持使用污点自动标识问题节点，它通过节点控制器在特定条件下自动为节点添加污点信息实现。它们都使用NoExecute效用标识，因此非能容忍此类污点的现在Pod对象也会遭到驱逐。目前，内建使用的此类污点有如下几个。

node.kubernetes.io/not-ready：节点进入NotReady状态时被自动添加的污点。

node.alpha.kubernetes.io/unreachable：节点进入NotReachable状态时被自动添加的污点。

node.kubernetes.io/out-of-disk：节点进入OutOfDisk状态时被自动添加的污点。

node.kubernetes.io/memory-pressure：节点内存资源面临压力。

node.kubernetes.io/disk-pressure：节点磁盘资源面临压力。

node.kubernetes.io/network-unavailable：节点网络不可用。

node.cloudprovider.kubernetes.io/uninitialized：kubelet由外部的云环境程序启动时，它自动为节点添加此污点，待到云控制器管理器中的控制器初始化此节点时再将其删除。

不过，Kubernetes的核心组件通常都要容忍此类的污点，以确保其相应的DaemonSet控制器能够无视此类污点于节点上部署相应的关键Pod对象，例如kube-proxy或kube-flannel等。

属性解析

属性示例

```
查看污点和污点容忍度
]# kubectl describe pod kube-apiserver-master -n kube-system
...
Tolerations:       :NoExecute

]# kubectl describe pod kube-proxy -n kube-system
...
Node-Selectors:              kubernetes.io/os=linux
Tolerations:                 op=Exists              污点容忍度
                             node.kubernetes.io/disk-pressure:NoSchedule
op=Exists
                             node.kubernetes.io/memory-pressure:NoSchedule
op=Exists
                             node.kubernetes.io/network-unavailable:NoSchedule
op=Exists
                             node.kubernetes.io/not-ready:NoExecute op=Exists
                             node.kubernetes.io/pid-pressure:NoSchedule
op=Exists
                             node.kubernetes.io/unreachable:NoExecute op=Exists
                             node.kubernetes.io/unschedulable:NoSchedule
op=Exists

]# kubectl get node master -o yaml
...
  taints:   污点
  - effect: NoSchedule      效用标识,
    key: node-role.kubernetes.io/master
...
```

- Traint 污点

属性解析

```
Taint是节点上属性，我们看一下Taints如何定义

kubectl explain node.spec.taints（对象列表）
    effect  pod不能容忍这个污点时，他的行为是什么，
        他有三个值：NoSchedule、PreferNoSchedule、NoExecute
    key      定义一个key=value:effect
    value    定义一个值，这是一个必选项
    timeAdded
```

effect的属性值

属性	解析
NoSchedule	不能容忍此污点的Pod对象不可调度至当前节点，属于强制型约束关系，但添加污点对节点上现存的Pod对象不产生影响
PreferNoSchedule	NoSchedule的柔性约束版本，即调度器尽量确保不会将那些不能容忍此污点的Pod对象调度至当前节点，除非不存在其他任何能够容忍此污点的节点可用；添加该类效用的污点同样对节点上现存的Pod对象不产生影响。
NoExecute	不能容忍此污点的新Pod对象不可调度至当前节点，属于强制型约束关系，而且节点上现存的Pod对象因节点污点变动或Pod容忍度变动而不再满足匹配条件时，Pod对象将会被驱逐。

- Tolerations 容忍度

属性解析

```
Tolerations 是Pod上属性，我们看一下Tolerations如何定义

kubectl explain pod.spec.tolerations
    effect                节点调度后的操作
    key                   被容忍的key
    operator              Exists只要key在就可以调度，Equal（等值比较）必须是值要相同
    tolerationSeconds     被驱逐的宽限时间，默认是0 就是立即被驱逐
    value                 被容忍key的值
```

容忍度的判断方式

```
在Pod对象上定义容忍度时，它支持两种操作符，
    一种是等值比较，表示容忍度与污点必须在key、value和effect三者之上完全匹配，
    一种是存在性判断(Exists)，表示二者的key和effect必须完全匹配，而容忍度中的value字段要用空值。
```

容忍度的匹配原则

```
    一个节点可以配置使用多个污点，而一个Pod对象也可以有多个容忍度，将一个Pod对象的容忍度套用到特定节点的污点之上进行匹配度检测时，时将遵循如下逻辑：
    1）首先处理每个有着与之匹配的容忍度的污点；
    2）对于不能匹配到容忍度的所有污点，若存在一个污点使用了NoSchedule效用标识，则拒绝调度当前Pod至该节点；
    3）对于不能匹配到容忍度的所有污点，若都不具有NoSchedule效用标识，但至少有一个污点使用了PreferNoScheduler效用标准，则调度器会尽量避免将当前Pod对象调度至该节点。
    4）如果至少有一个不能匹配容忍度的污点使用了NoExecute效用标识，节点将立即驱逐当前Pod对象，或者不允许该Pod调度至给定的节点；而且，即便容忍度可以匹配到使用了NoExecute效用标识的污点，若在Pod上定义容忍度时同时使用tolerationSeconds属性定义了容忍时限，则在超出时限后当前脚Pod也将会被节点所驱逐。
```

小结

```

```

污点实践

学习目标

这一节，我们从 污点实践、容忍度实践、小结 三个方面来学习。

污点实践

命令格式

```
查看污点：
kubectl get nodes <nodename> -o go-template={{.spec.taints}}
kubectl describe nodes | grep -i taint

添加污点：
kubectl taint node <node-name> <key>=<value>:<effect> ...

删除污点：
kubectl taint node <node-name> <key>[:<effect>]-
kubectl patch nodes <node-name> -p '{"spec":{"taints":[]}}'
```

简单实践

```
查看污点
]# kubectl describe nodes | grep -i taint
Taints:             node-role.kubernetes.io/master:NoSchedule
Taints:             <none>
Taints:             <none>

]# kubectl get nodes master -o go-template={{.spec.taints}}
[map[effect:NoSchedule key:node-role.kubernetes.io/master]]

添加污点
]# kubectl taint node node1 node-type=production:NoSchedule
]# kubectl get nodes node1 -o go-template={{.spec.taints}}
[map[effect:NoSchedule key:node-type value:production]]
```

```
给node1增加标签
kubectl label node node1 env=dev

增加新资源
cat >> 14-scheduler-pod-taint-test.yaml <<-EOF
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
    imagePullPolicy: IfNotPresent
  nodeSelector:
    env: dev
EOF
```

应用资源配置文件

```
kubectl apply -f 14-scheduler-pod-taint-test.yaml
```

查看效果

```
]# kubectl get pod
```

```
root@master1:~/scheduler# kubectl get pod -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP       NODE     NOMINATED NODE   READINESS GATES
pod-test   0/1     Pending   0          20s   <none>   <none>   <none>           <none>
root@master1:~/scheduler# kubectl describe  pod pod-test
Name:         pod-test
Events:
  Type     Reason            Age   From              Message
  ----     ------            ----  ----              -------
  Warning  FailedScheduling  27s   default-scheduler  0/3 nodes are available: 1 node(s) had taint {node-type: producti
on}, that the pod didn't tolerate, 2 node(s) didn't match Pod's node affinity/selector.
```

结果显示：

由于污点原因，无法将pod创建到指定结点上

删除key为node-type，effect为NoSchedule的污点

方法一：

```
kubectl taint node node1 node-type-
```

方法二：

```
kubectl patch nodes node1 -p '{"spec":{"taints":[]}}'
```

查看效果

```
kubectl get pod
```

结果显示：

node1上立刻就有pod正常运行了

容忍度实践

容忍度实践

为node1添加污点

```
kubectl taint node node1 node-type=production:NoSchedule
```

查看效果

```
# kubectl describe node node1 | grep Taints
Taints:             node-type=production:NoSchedule

# kubectl get node node1 -o yaml
spec:
  ...
  taints:
  - effect: NoSchedule
    key: node-type
    value: production
```

创建资源定义文件

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-tol
spec:
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
```

```
    imagePullPolicy: IfNotPresent
  nodeSelector:
    env: dev
  tolerations:
  - key: "node-type"
    operator: "Equal"
    value: "production"
    effect: "NoSchedule"

应用资源定义文件
kubectl apply -f 15-scheduler-pod-toleration-test.yaml

查看效果
# kubectl get pod -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP            NODE
pod-tol   1/1     Running   0          11s   10.244.1.19   node1
结果显示：
    pod的容忍度配置成功了
```

容忍度驱离方法

```
为node1添加驱离污点
kubectl taint node node1 diskfull=true:NoExecute
注意： 一旦打上这个标签，当前节点上的所有pod都会被驱离

查看效果
kubectl get pod -w
结果显示：
    当前节点上的容器，自动就被移除了
```

清理措施

```
kubectl taint node node1 node-type-
kubectl taint node node1 diskfull-
```

小结

拓扑调度

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

我们知道，对于pod来说，其在定义pod亲和性和反亲和的时候，有一个 topologyKey的属性，但是默认情况下，pod的亲和性调度，仅仅针对单一的拓扑场景，也就是说，要么所有的pod都在这里，要么所有的pod都不要在这里，这样会导致，应用过于集中，反而导致物理资源的浪费。

那么我们希望在进行正常pod亲和性调度的时候，能够自动识别多个不同的调度节点，然后均匀的分散到多个节点中，从而实现资源的有效利用。而这就是我们接下来要学习的 拓扑调度。

注意：

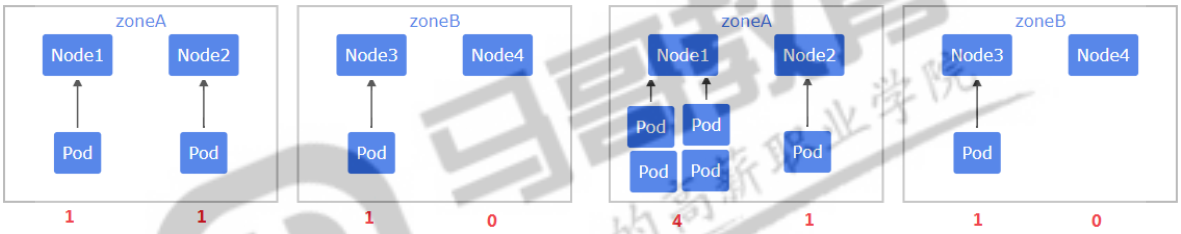
这个功能是 k8s 1.19版本才有的功能，

v1.18 之前版本中，要用 Pod 拓扑扩展，需要在 API 服务器 和调度器 中启用 EvenPodsSpread 特性。

属性解析

```
kubectl explain pod.spec.topologySpreadConstraints
   labelSelector        根据 LabelSelector来选择匹配的pod所在的位置
   maxSkew              描述了 pod 可能不均匀分布的程度，各节点pod数量比例的最大限制
   topologyKey          用于指定带有此标签的节点在同一拓扑中。
   whenUnsatisfiable    指示如果 Pod 不满足扩展约束，如何处理它。默认值是DoNotSchedule
```

maxSkew 默认值是1，也就是所有节点的分布式 1：1，如果这个值是大的话，则pod的分布可以是不均匀的。



简单实践

准备工作

为所有节点添加标识

```
kubectl label node master1 node=master1 zone=zoneA
kubectl label node node1 node=node1 zone=zoneA
kubectl label node node2 node=node2 zone=zoneB
```

实践1 - 均匀分散到多个节点中

定制资源配置文件

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-affinity-preferred
spec:
  replicas: 7
  selector:
    matchLabels:
      foo: bar
  template:
    metadata:
      labels:
        foo: bar
    spec:
```

```
      containers:
      - name: pod-test
        image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
        imagePullPolicy: IfNotPresent
      topologySpreadConstraints:
      - maxSkew: 3
        topologyKey: zone
        whenUnsatisfiable: DoNotSchedule
        labelSelector:
          matchLabels:
            zone: bar
```

应用资源配置文件

```
kubectl apply -f 16-scheduler-pod-topology-test.yaml
```

检查效果

```
kubectl get pod -o wide
```

```
root@master1:~/scheduler# kubectl get pod -o wide
NAME                                        READY   STATUS    RESTARTS   AGE   IP             NODE      NOMINATED NODE
pod-affinity-preferred-799d687758-2xp2g     1/1     Running   0          37s   10.244.1.28    node1     <none>
pod-affinity-preferred-799d687758-45d4b     1/1     Running   0          37s   10.244.0.17    master1   <none>
pod-affinity-preferred-799d687758-d5s87     1/1     Running   0          37s   10.244.0.16    master1   <none>
pod-affinity-preferred-799d687758-d6lw8     1/1     Running   0          37s   10.244.0.15    master1   <none>
pod-affinity-preferred-799d687758-jj4nd     1/1     Running   0          37s   10.244.1.27    node1     <none>
pod-affinity-preferred-799d687758-qxgwf     1/1     Running   0          37s   10.244.2.17    node2     <none>
pod-affinity-preferred-799d687758-s6jt9     1/1     Running   0          37s   10.244.1.29    node1     <none>
```

结果显示：

他们三个节点间是 3：3：1 的效果 3-1 =2 < 3 ,符合要求。

小结

定制资源

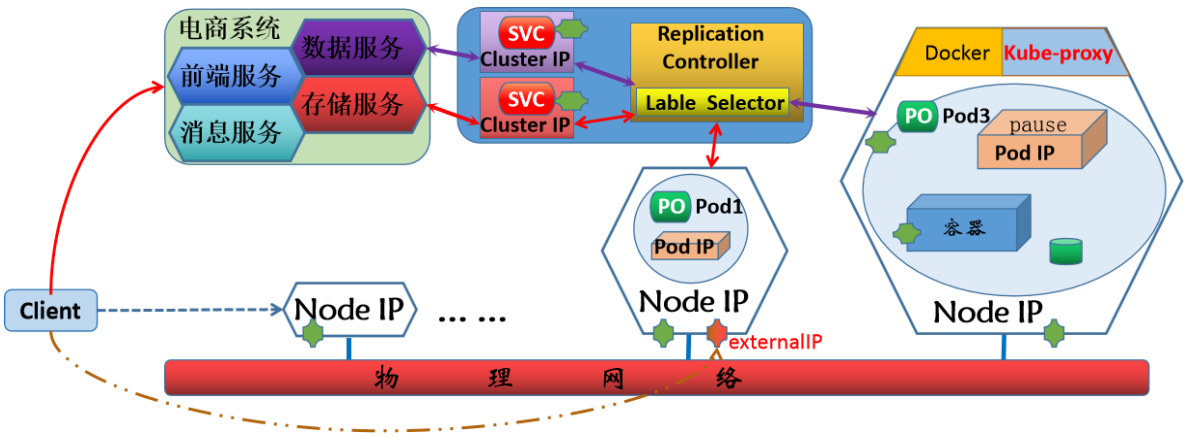
基础知识

学习目标

这一节，我们从 基础知识、CRD简介、小结 三个方面来学习。

基础知识

应用回顾



根据我们对k8s的学习，到目前位置，我们为了在k8s上能够正常的运行我们所需的服务，需要遵循以下方式来创建相关资源：

1 合理的分析业务需求
2 梳理业务需求的相关功能
3 定制不同功能的资源配置文件
4 应用资源配置文件，完善业务环境。

需求

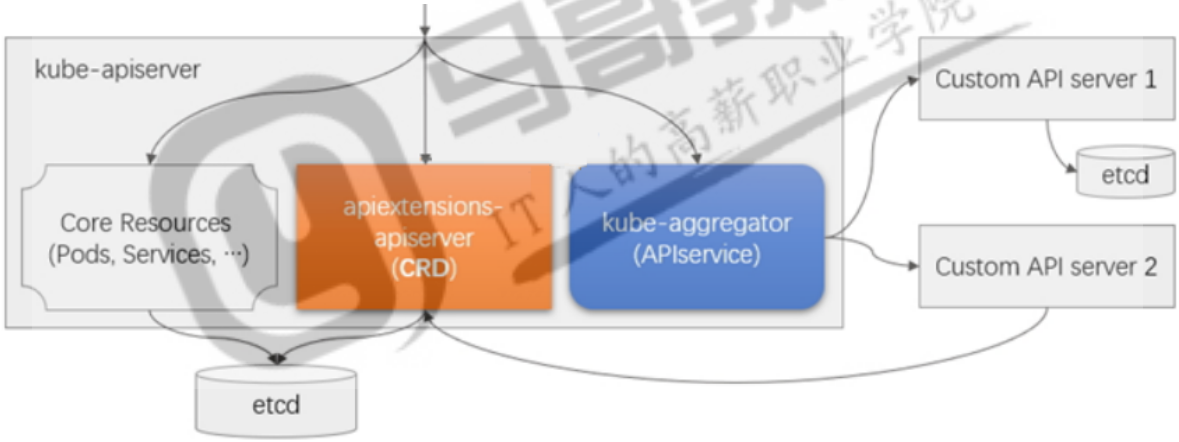
我们在操作k8s资源的时候，发现一个有趣的特点，到现在位置，我们所有的操作，基本上都是在k8s限制的资源对象中进行相关的操作，这些资源对象适用于通用的业务场景，而在我们的业务场景中，多多少少的会涉及到专用的资源对象。

比如：我们的监控场景需要监控的数据、日志场景需要收集的日志、流量场景需要传递的数据、等等。

为了高效的定制我们需要的环境，那么需要拥有一些专用的资源方便我们来使用，而在k8s之上却没有，提供了一个专用的接口，可以方便我们自己来定制需要的资源。

k8s扩展的方法

扩展Kubernetes API常用方式：
1、使用CRD自定义资源类型 （易用但限制颇多）
2、开发自定义API Server并聚合至主API Server （富于弹性但代码工作量大）
3、定制扩展K8s源码 （添加新的核心类型时采用）



示例

```
root@master1:~/crd# kubectl get pod -n kube-system
NAME                                       READY   STATUS    RESTARTS      AGE
calico-kube-controllers-75f8f6cc59-dvbfk   1/1     Running   0             114m
calico-node-4chgs                          1/1     Running   1 (9h ago)    12h
calico-node-phkcd                          1/1     Running   1 (11h ago)   12h
calico-node-rbrd9                          1/1     Running   1 (11h ago)   12h
```

calico环境创建的时候，就用到了很多CRD对象，而且我们为了让CRD能够生效，该软件还提供了一个controller的CRD控制器。

这个控制器就是将CRD对象转换为真正有意义的现实的代码。

CRD简介

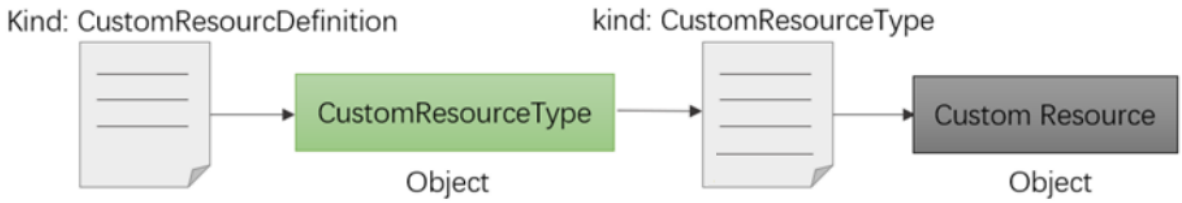
简介

资源（Resource） 是 Kubernetes API 中的一个端点， 其中存储的是某个类别的 API 对象 的一个集合。 例如内置的 pods 资源包含一组 Pod 对象。

定制资源（Custom Resource） 是对 Kubernetes API 的扩展，不一定在默认的 Kubernetes 安装中就可用。定制资源所代表的是对特定 Kubernetes 安装的一种定制。 不过，很多 Kubernetes 核心功能现在都用定制资源来实现，这使得 Kubernetes 更加模块化。

定制资源可以通过动态注册的方式在运行中的集群内或出现或消失，集群管理员可以独立于集群 更新定制资源。一旦某定制资源被安装，用户可以使用 kubectl 来创建和访问其中的对象，就像他们为 pods 这种内置资源所做的一样。

CRD 功能是在 Kubernetes 1.7 版本被引入的，用户可以根据自己的需求添加自定义的 Kubernetes 对象资源。



定制控制器

就定制资源本身而言，它只能用来存取结构化的数据。 当你将定制资源与 定制控制器（Custom Controller） 相结合时，定制资源就能够 提供真正的 声明式 API（Declarative API）。

使用声明式 API， 你可以 声明 或者设定你的资源的期望状态，并尝试让 Kubernetes 对象的当前状态 同步到其期望状态。控制器负责将结构化的数据解释为用户所期望状态的记录，并 持续地维护该状态。

你可以在一个运行中的集群上部署和更新定制控制器，这类操作与集群的生命周期无关。 定制控制器可以用于任何类别的资源，不过它们与定制资源结合起来时最为有效。 Operator 模式就是将定制资源 与定制控制器相结合的。你可以使用定制控制器来将特定于某应用的领域知识组织 起来，以编码的形式构造对 Kubernetes API 的扩展。

什么样的情况下用CRD

你希望使用 Kubernetes 客户端库和 CLI 来创建和更改新的资源。
你希望 kubectl 能够直接支持你的资源；例如，kubectl get my-object object-name。
你希望构造新的自动化机制，监测新对象上的更新事件，并对其他对象执行 CRUD 操作，或者监测后者更新前者。
你希望编写自动化组件来处理对对象的更新。
你希望使用 Kubernetes API 对诸如 .spec、.status 和 .metadata 等字段的约定。
你希望对象是对一组受控资源的抽象，或者对其他资源的归纳提炼。

小结

简单实践

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

配置解析

```
apiVersion: apiextensions.k8s.io/v1      # API群组和版本
kind: CustomResourceDefinition           # 资源类别
metadata:
  name <string>                          # 资源名称
spec:
    conversion <Object>                  # 定义不同版本间的格式转换方式
       trategy <string>                  # 不同版本间的自定义资源转换策略，有None和
Webhook两种取值
       webhook <Object>                  # 如何调用用于进行格式转换的webhook
    group <string>                       # 资源所属的API群组
    names <Object>                       # 自定义资源的类型，即该CRD创建资源规范时使用的
kind
      categories  <[]string>             # 资源所属的类别编目，例如”kubectl get
all”中的all
      kind <string>                      # kind名称，必选字段
      listKind <string>                  # 资源列表名称，默认为"`kind`List"
      plural <string>                    # 用于API路径
`/apis/<group>/<version>/.../<plural>`
      shortNames <[]string>              # 该资源的kind的缩写格式
      singular <string>                  # 资源kind的单数形式，必须使用全小写字母
    preserveUnknownFields <boolean>      # 预留的非知名字段，kind等都是知名的预留字段
    scope <string>                       # 作用域，可用值为Cluster和Namespaced
    versions <[]Object>                  # 版本号定义
      additionalPrinterColumns <[]Object>   # 需要返回的额外信息
      name <string>                      # 形如vM[alphaN|betaN]格式的版本名称，例如v1
或v1alpha2
      schema <Object>                    # 该资源的数据格式（schema）定义，必选字段
        openAPIV3Schema  <Object>        # 用于校验字段的schema对象，格式请参考相关手册
      served <boolean>                   # 是否允许通过RESTful API调度该版本，必选字段
      storage <boolean>                  # 将自定义资源存储于etcd中时是不是使用该版本
      subresources <Object>              # 子资源定义
        scale <Object>                   # 启用scale子资源，通过autoscaling/v1.Scale
发送负荷
        status <map[string]>             # 启用status子资源，为资源生成/status端点
```

配置示例

```
我们以 calico的自定义资源对象为例
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: networksets.crd.projectcalico.org
spec:
  group: crd.projectcalico.org
  names:
    kind: NetworkSet
    listKind: NetworkSetList
    plural: networksets
    singular: networkset
  scope: Namespaced
  versions:
```

```
  - name: v1
    schema:
      openAPIV3Schema:
        description: NetworkSet is the Namespaced-equivalent of the
GlobalNetworkSet.
        properties:
          apiVersion:
            description: '。。。'
            type: string
          kind:
            description: '。。。'
            type: string
          metadata:
            type: object
          spec:
            description: 。。。.
            properties:
              nets:
                description: 。。。.
                items:
                  type: string
                type: array
            type: object
        type: object
    served: true
    storage: true
status:
  acceptedNames:
    kind: ""
    plural: ""
  conditions: []
  storedVersions: []
```

```
root@master1:~/mykubernetes/calico# kubectl get CustomResourceDefinition | egrep 'NAME|networksets'
NAME                                          CREATED AT
globalnetworksets.crd.projectcalico.org       2021-10-10T00:51:40Z
networksets.crd.projectcalico.org             2021-10-10T00:51:43Z
root@master1:~/mykubernetes/calico# kubectl api-resources | egrep 'NAME|networksets'
NAME                              SHORTNAMES   APIVERSION                        NAMESPACED   KIND
globalnetworksets                              crd.projectcalico.org/v1          false        GlobalNetworkSet
networksets                                    crd.projectcalico.org/v1          true         NetworkSet
```

简单实践

定义资源

```
创建一个用户的CRD资源
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: users.auth.ilinux.io
spec:
  group: auth.ilinux.io
  names:
    kind: User
    plural: users
    singular: user
    shortNames:
    - u
  scope: Namespaced
  versions:
```

```
    - served: true
      storage: true
      name: v1alpha1
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                userID:
                  type: integer
                  minimum: 1
                  maximum: 65535
                groups:
                  type: array
                  items:
                    type: string
                email:
                  type: string
                password:
                  type: string
                  format: password
              required: ["userID","groups"]
```

应用资源配置文件

```
kubectl  apply -f 17-scheduler-crd-user-v1.yaml
```

查看效果

```
# kubectl get crd | egrep 'NA|users'
NAME                                          CREATED AT
users.auth.ilinux.io                          2021-10-10T13:42:30Z
```

```
root@master1:~/crd# kubectl explain users.auth.ilinux.io
KIND:     User
VERSION:  auth.ilinux.io/v1alpha1

DESCRIPTION:
     <empty>

FIELDS:
   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

   kind <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client submits
     requests to. Cannot be updated. In CamelCase. More info:
     https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

   metadata     <Object>
     Standard object's metadata. More info:
     https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata

   spec <Object>
```

简单使用自定义资源

创建自定义资源

```
apiVersion: auth.ilinux.io/v1alpha1
kind: User
metadata:
  name: admin
```

```
  namespace: default
spec:
  userID: 1
  email: sswang@example.com
  groups:
  - superusers
  - adminstrators
  password: www.example.com

应用资源配置文件
kubectl  apply -f 18-scheduler-crd-user-v1-test.yaml

查看效果
kubectl get users
```

注意

```
虽然我们创建好了CRD，但是CRD本身没有太大的意义，他的意义在于，我们如何在 控制器或者operator 中如何使用它
```

小结

```

```

流量调度

Ingress基础

学习目标

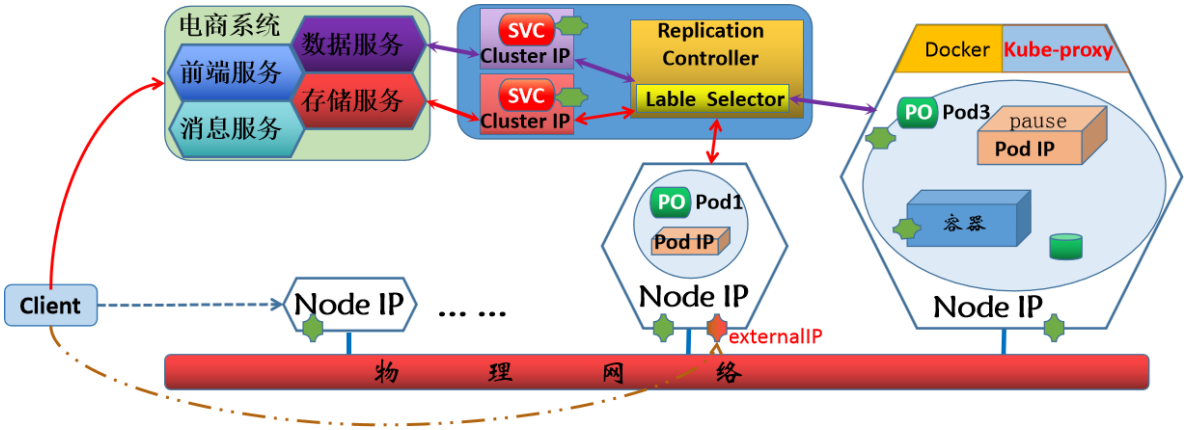
这一节，我们从 基础知识、原理解析、小结 三个方面来学习。

基础知识

简介

```
    在实际的应用中，kubenetes接受的不仅仅有内部的流量，还有外部流量，我们可以通过两种方式实现将集群外部的流量引入到集群的内部中来，从而实现外部客户的正常访问。
    service方式：
        nodePort、externalIP 等service对象方式，借助于 namespace、iptables、ipvs 等代理模式实现流量的转发。
    Host方式：
        通过node节点的 hostNetwork 与 hostPort 及配套的port映射，以间接的方式实现service的效果
```

k8s网络



在业内有一个不成文的说法，就是：

将集群内部 pod之类对象 之间的网络通信称为 东西向流量；

将集群外部应用 和 集群内部之间对象 之间的网络通信 称为 南北向流量。

南入↓ -- ingress

北出↑ -- egress

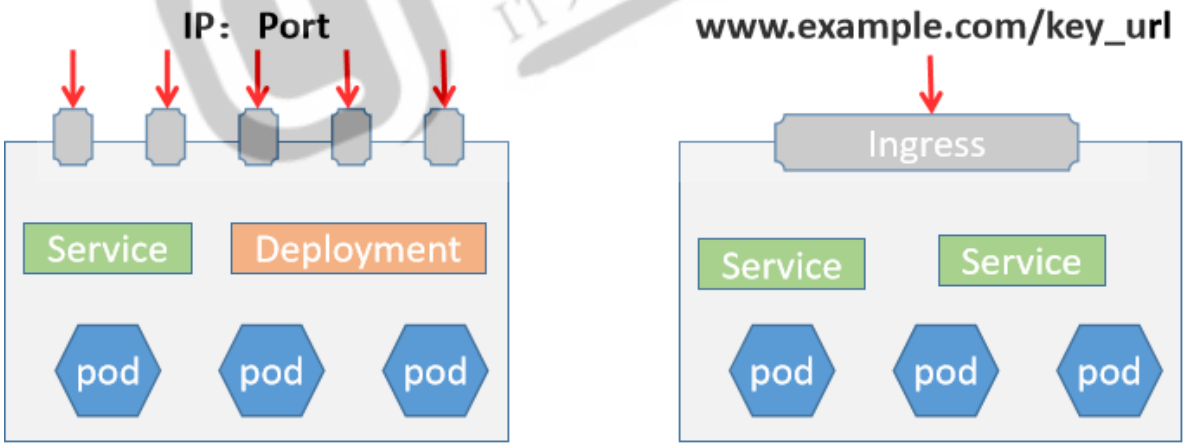
问题

1 在实际的应用中，虽然我们可以基于service或host等多种方式实现集群内外的网络通信，但是这些内容都是基于四层协议来进行调度的，而且服务级别的健康检查功能很难实现。

2 而对于外部流量的接入，一般都是http(s)协议的数据，四层协议是无法实现的。尤其是涉及到各种ssl会话的管理，由于其位于osi七层模型的会话层，要高于传输层的四层，service和host是无法正常实现的。

3 虽然service能够实现将外部流量引入到集群内部，但是其本质上，是通过网络规则方式来进行转发的，形象的说法就是在墙上打洞，将集群内部的服务暴露到外部。甚至这些洞和洞之间根本没有关联关系。

方案



为了解决上述的问题，就引入了一种新的解决方法ingress，简单来说，它可以解决上面提出的三个问题：

1 可以基于https的方式，将集群外部的流量统一的引入到集群内部

2 通过一个统一的流量入口，避免将集群内部大量的"洞"暴露给外部。

ingress这种利用应用层协议来进行流量的负载均衡效果，它可以实现让用户通过域名来访问相应的service就可以了，无需关心Node IP及Port是什么，避免了信息的泄露。

ingress 是k8s上的一种标准化资源格式，它为了剥离与特定负载均衡功能实现软件的相关性，而抽象出来的一种公共的统一的声明式配置格式。然后借助于特定的ingress controller功能负责将其加载并转换成k8s集群内部的配置信息。

ingress的特点符合我们之前对于CRD的学习 - ingress + controller，所以，其具备了动态更新并加载新配置的特性。而且ingress本身是不具备实现集群内外流量通信的功能的，这个功能是通过controller来实现的。

原理解析

实现

Ingress由任何具有反向代理功能的程序实现，如Nginx、Traefik、Envoy、HAProxy、Vulcand等，Ingress本身是运行于集群中的Pod资源对象。

ingress 主要包含两个组件Ingress Controller和Ingress。

组件	解析
Ingress	将Nginx的配置抽象成一个Ingress对象，每个服务对应一个ingress的yaml配置文件
Ingress Controller	将新加入的Ingress转化成Nginx的配置文件并使之生效

问题梳理

如果我们将ingress 以pod的方式部署到k8s集群内部，那么就会遇到多个问题：

1 ingress的pod如何引入外部流量
- 通过一个专用的service就可以实现了

2 ingress的pod如何把流量负载均衡到其他pod
- 关于pod负载均衡的流量，直接通过controller转发给后端pod即可。

3 后端应用的pod很多，如何知道谁是我们要转发的目标？
- 通过k8s的server对所有的pod进行分组管理，借助于controller内部的负载均衡配置，找到对应的目标。

原理解析

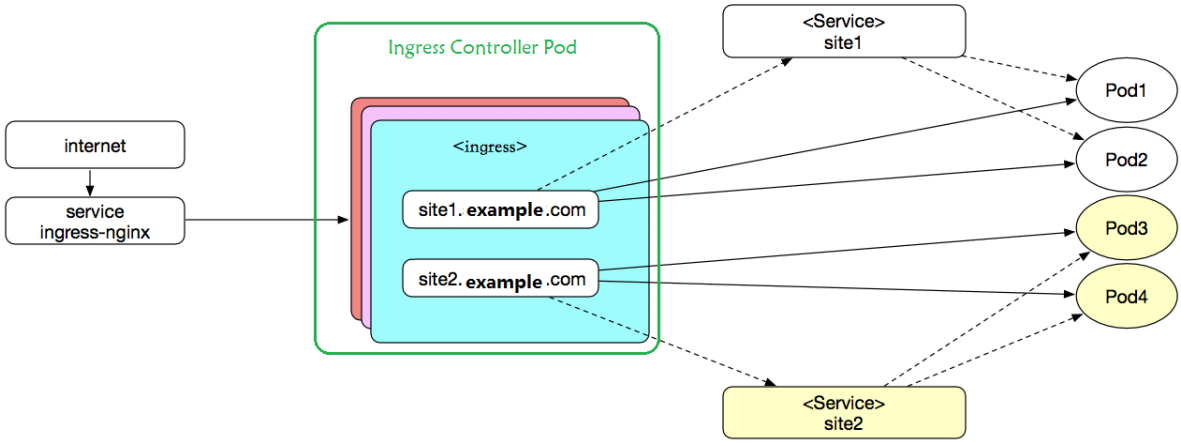
Ingress是授权入站连接到达集群服务的规则集合。

从外部流量调度到nodeport上的service

从service调度到ingress-controller

ingress-controller根据ingress[Pod]中的定义（虚拟主机或者后端的url）

根据虚拟主机名直接调度到后端的一组应用pod中



注意：
整个流程中涉及到了两处service内容
service ingress-nginx 是帮ingress controller接入外部流量的
虚线表示service对后端的应用进行分组，实现是ingress实际的访问流程

常见的解决方案

对于ingress controller的软件实现，其实没有特殊的要求，只要能够实现七层的负载均衡功能效果即可，各种组织对于ingress的controller虽然实现方式不同，但是功能基本上一样，常见的实现方式有：
Ingress-Nginx: Kong
HAProxy
性能相对于nginx来说，较稳定
Envoy:
基于C++语言开发，本身就为动态环境设计的，支持配置动态加载的XDS API
不能像ingress-nginx一样，直接加载配置文件并动态加载转换成对应的应用配置，所以需要借助于其他控制平面工具来实现。
常见解决方案：Contour, Gloo
Traefik

注意：ingress 自从 1.6版本引入后，在1.19 正式进入了一个稳定版本状态。

小结

nginx实践

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

参考资料：https://github.com/kubernetes/ingress-nginx
安装文档：https://kubernetes.github.io/ingress-nginx/deploy/

配置文件解析

```
apiVersion: networking.k8s.io/v1                # 资源所属的API群组和版本
kind: Ingress                                   # 资源类型标识符
metadata:                                       # 元数据
  name <string>                                 # 资源名称
  annotations:                                  # 资源注解，v1beta1使用下面的注解来指定要
解析该资源的控制器类型
    kubernetes.io/ingress.class: <string>       # 适配的Ingress控制器类别,便于多ingress
组件场景下，挑选针对的类型
  namespace <string>                            # 名称空间
spec:
  rules <[]Object>                              # Ingress规则列表，也就是http转发时候用到
的 url关键字
  - host <string>                               # 虚拟主机的FQDN，支持“*”前缀通配，不支持
IP，不支持指定端口
    http <Object>
      paths <[]Object>                          # 虚拟主机PATH定义的列表，由path和
backend组成
      - path <string>                           # 流量匹配的HTTP PATH，必须以/开头
        pathType <string>                       # 支持Exact、Prefix和
ImplementationSpecific，必选
        backend <Object>                        # 匹配到的流量转发到的目标后端
          resource <Object>                     # 引用的同一名称空间下的资源，与下面两个字
段互斥
          service <object>                      # 关联的后端Service对象
            name <string>                       # 后端Service的名称
            port <object>                       # 后端Service上的端口对象
              name <string>                     # 端口名称
              number <integer>                  # 端口号
  tls <[]Object>                                # TLS配置，用于指定上rules中定义的哪些
host需要工作HTTPS模式
  - hosts <[]string>                            # 使用同一组证书的主机名称列表
    secretName <string>                         # 保存于数字证书和私钥信息的secret资源名
称，用于主机认证
  backend <Object>                              # 默认backend的定义，可嵌套字段及使用格式
跟rules字段中的相同
  ingressClassName  <string>                    # ingress类名称，用于指定适配的控制器，类
似于注解的功能
```

环境部署

```
获取配置文件
wget  https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-
v1.0.3/deploy/static/provider/baremetal/deploy.yaml

修改基础镜像
# grep image: deploy.yaml
          image: 10.0.0.19:80/google_containers/ingress-nginx-controller:v1.0.0
          image: 10.0.0.19:80/google_containers/ingress-nginx-kube-webhook-
certgen:v1.0
          image: 10.0.0.19:80/google_containers/ingress-nginx-kube-webhook-
certgen:v1.0

开放访问入口地址
# vim deploy.yaml
261 apiVersion: v1
262 kind: Service
...
```

```
273   namespace: ingress-nginx
274 spec:
275   type: NodePort
276   externalIPs: ['10.0.0.12']            # 增加一个外网访问的入口ip
277   ports:
278     - name: http
279       port: 80
```

应用资源配置文件

```
kubectl apply -f deploy.yaml
```

确认效果

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

```
root@master1:~/ingress# kubectl get ns
NAME              STATUS   AGE
default           Active   70m
ingress-nginx     Active   41s
kube-node-lease   Active   70m
kube-public       Active   70m
kube-system       Active   70m
root@master1:~/ingress# kubectl get all -n ingress-nginx
NAME                                            READY   STATUS      RESTARTS   AGE
pod/ingress-nginx-admission-create--1-nmftn     0/1     Completed   0          48s
pod/ingress-nginx-admission-patch--1-rbljh      0/1     Completed   1          48s
pod/ingress-nginx-controller-6994db9574-lsnkj   1/1     Running     0          48s

NAME                                         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
service/ingress-nginx-controller             NodePort    10.103.246.59   10.0.0.12     80:30531/TCP,443:30040/TCP   48s
service/ingress-nginx-controller-admission   ClusterIP   10.103.0.158    <none>        443/TCP                      48s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ingress-nginx-controller   1/1     1            1           48s

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/ingress-nginx-controller-6994db9574   1         1         1       48s

NAME                                       COMPLETIONS   DURATION   AGE
job.batch/ingress-nginx-admission-create   1/1           2s         48s
job.batch/ingress-nginx-admission-patch    1/1           2s         48s
```

测试访问页面

```
root@master1:~/ingress# curl 10.0.0.12:30531
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

直接获取状态码

```
# curl -I -o /dev/null -s -w %{http_code}"\n" 10.0.0.12:30531
404
```

简单实践

创建deployment

定义资源文件

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
spec:
  replicas: 4
  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:v0.1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
          name: http
---
apiVersion: v1
kind: Service
metadata:
  name: deployment-service
spec:
  selector:
    app: pod-test
  ports:
    - name: http
      port: 80
      targetPort: 80
```

应用资源文件

```
kubectl apply -f 01-ingress-deployment-flask.yaml
```

查看效果

```
kubectl get deployments
kubectl get svc
kubectl get pod -o wide
```

```
root@master1:~/ingress# kubectl get deployments.apps
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   4/4     4            4           15s
root@master1:~/ingress# kubectl get svc deployment-service
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
deployment-service   ClusterIP   10.105.195.14   <none>        80/TCP    32s
root@master1:~/ingress# kubectl get pod -o wide
NAME                               READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES
deployment-test-95c58b447-87xsp    1/1     Running   0          39s   10.244.1.5   node1   <none>           <none>
deployment-test-95c58b447-jg6gz    1/1     Running   0          39s   10.244.2.6   node2   <none>           <none>
deployment-test-95c58b447-nshhn    1/1     Running   0          39s   10.244.1.4   node1   <none>           <none>
deployment-test-95c58b447-xvl8p    1/1     Running   0          39s   10.244.2.7   node2   <none>           <none>
```

创建ingress

定义资源文件

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: sswang.example.com
    http:
      paths:
      - path: /
```

```
        pathType: Prefix
        backend:
          service:
            name: deployment-service
            port:
              number: 80
```

属性解析：

注释信息，对于不同的ingress的内容是不一样的

对于ingress来说，最终要的就是rules了，这里的hosts是一个自定义的域名

应用资源文件

```
kubectl apply -f 02-ingress-http-test.yaml
```

查看效果

```
kubectl get ingress
kubectl describe ingress ingress-test
```

```
root@master1:~/ingress# kubectl get ingress
NAME           CLASS    HOSTS              ADDRESS     PORTS   AGE
ingress-test   <none>   sswang.example.com   10.0.0.16   80      5s
root@master1:~/ingress# kubectl describe ingress ingress-test
Name:             ingress-test
Namespace:        default
Address:          10.0.0.16
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host               Path  Backends
  ----               ----  --------
  sswang.example.com
                     /   deployment-service:80 (10.244.1.4:80,10.244.1.5:80,10.244.2.6:80 + 1 more...)
Annotations:         kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age                From                      Message
  ----    ------  ----               ----                      -------
  Normal  Sync    11s (x2 over 13s)  nginx-ingress-controller  Scheduled for sync
```

结果显式：

ingress 将 后端的deployment-service:80 的样式自动与 ingress 的 url(/) 关联在一起了

为了更好的演示域名的效果，我们为externalIPs的ip地址做一个域名绑定

```
echo '10.0.0.12 sswang.example.com' >> /etc/hosts
```

访问效果

```
curl sswang.example.com
```

```
root@master1:~/ingress# echo '10.0.0.12 sswang.example.com' >> /etc/hosts
root@master1:~/ingress# curl sswang.example.com
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-nshhn, ServerIP: 10.244.1.4!
root@master1:~/ingress# curl sswang.example.com
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-87xsp, ServerIP: 10.244.1.5!
root@master1:~/ingress# curl 10.0.0.12
<html>
<head><title>404 Not Found</title></hea
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
root@master1:~/ingress# curl http://10.0.0.12
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
root@master1:~/ingress# curl http://10.0.0.12/
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

结果显式：

通过域名可以正常的访问后端的服务，而非域名访问方式受到了限制

非域名方式进行访问

```
# curl http://10.0.0.12 -H "Host:sswang.example.com"
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-jg6gz, ServerIP: 10.244.2.6!
```

结果显示：

通过自定义携带域名的方式可以正常访问。

原理解析

```
进入到ingress的控制器端，查看配置效果
# kubectl -n ingress-nginx exec -it ingress-nginx-controller-6994db9574-lsnkj --
/bin/bash -c 'grep sswang nginx.conf'
    ## start server sswang.example.com
        server_name sswang.example.com ;
    ## end server sswang.example.com

结果显式：
    这里直接将我们提前定义的域名信息作为信息的入口了
```

小结

```

```

nginx进阶实践

学习目标

这一节，我们从 准备工作、多地址实践、小结 三个方面来学习。

需求

```
nginx controller的功能远不止我们所说的主机域名的管理，在实际的工作需求中，肯定会遇到，不同的
功能服务以子路径的方式来进行访问，以及与https相关的访问。
```

准备工作

```
单独再来一个nginx的服务
定义资源文件
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx-test
  template:
    metadata:
      labels:
        app: nginx-test
    spec:
      containers:
      - name: nginx-test
        image: 10.0.0.19:80/mykubernetes/nginx:1.21.3
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
```

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

配置解析

为了避免多个service使用同一个后端端口，我们这里一定要为service的端口命名

应用资源文件

```
kubectl apply -f 03-ingress-deployment-nginx.yaml
```

查看效果

```
kubectl get deployment
```

多地址实践

- 多主机访问实践

需求

```
访问 flask.example.com/的时候，返回flask的结果
访问 nginx.example.com/的时候，返回nginx的结果
```

资源配置文件

```
编辑资源定义文件
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-mul-url
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: flask.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: deployment-service
            port:
              name: http
              #number: 80
  - host: nginx.example.com
    http:
      paths:
      - path: /
```

```
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              name: nginx
```

配置解析：

这里面有两个主机访问的地址是同一个后端端口，如果还用number的话，就无法正常的识别导致同一个地址访问不同的后端效果

应用资源文件

```
kubectl apply -f 04-ingress-http-mul-host.yaml
```

查看效果

```
kubectl get ingress
kubectl describe ingress ingress-test
```

```
root@master1:~/ingress# kubectl get ingress
NAME              CLASS    HOSTS                                ADDRESS     PORTS   AGE
ingress-mul-url   <none>   flask.example.com,nginx.example.com   10.0.0.16   80      11m
root@master1:~/ingress# kubectl describe ingress ingress-mul-url
Name:             ingress-mul-url
Namespace:        default
Address:          10.0.0.16
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host               Path  Backends
  ----               ----  --------
  flask.example.com
                     /   deployment-service:http (10.244.1.4:80,10.244.1.5:80,10.244.2.6:80 + 1 more...)
  nginx.example.com
                     /   nginx-service:nginx (10.244.1.26:80,10.244.1.27:80,10.244.2.19:80 + 1 more...)
Annotations:         kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age                  From                      Message
  ----    ------  ----                 ----                      -------
  Normal  Sync    3m50s (x4 over 11m)  nginx-ingress-controller  Scheduled for sync
```

提前准备多个后端主机的域名

```
echo '10.0.0.12 flask.example.com nginx.example.com' >> /etc/hosts
```

访问效果

```
root@master1:~/ingress# curl flask.example.com
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-xvl8p, ServerIP: 10.244.2.7!
root@master1:~/ingress# curl flask.example.com
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-jg6gz, ServerIP: 10.244.2.6!
root@master1:~/ingress# curl nginx.example.com
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

- 单域名多url访问

需求

访问 sswang.example.com/flask的时候，返回flask的结果
访问 sswang.example.com/nginx的时候，返回nginx的结果

注意事项：

由于这里涉及到了域名的url转发，而后端服务有可能不存在，随意我们需要在后端url转发的时候，取消转发关键字。

方法就是，在annotation中添加一个重写的规则

```
nginx.ingress.kubernetes.io/rewrite-target: /
```

即，所有的请求，把ingress匹配到的url关键字清除掉

测试效果

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-mul-url
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: sswang.example.com
    http:
      paths:
      - path: /flask
        pathType: Prefix
        backend:
          service:
            name: deployment-service
            port:
              name: http
              #number: 80
      - path: /nginx
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              name: nginx
              #number: 80
```

应用资源文件

```
kubectl apply -f 05-ingress-http-mul-host.yaml
```

查看效果

```
kubectl get ingress
kubectl describe ingress
```

```
root@master1:~/ingress# kubectl get ingress
NAME             CLASS    HOSTS               ADDRESS     PORTS   AGE
ingress-mul-url  <none>   sswang.example.com  10.0.0.16   80      18m
root@master1:~/ingress# kubectl describe ingress ingress-mul-url
Name:             ingress-mul-url
Namespace:        default
Address:          10.0.0.16
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host               Path  Backends
  ----               ----  --------
  sswang.example.com
                     /flask   deployment-service:http (10.244.1.4:80,10.244.1.5:80,10.244.2.6:80 + 1 more...)
                     /nginx   nginx-service:nginx (10.244.1.26:80,10.244.1.27:80,10.244.2.19:80 + 1 more...)
Annotations:         kubernetes.io/ingress.class: nginx
                     nginx.ingress.kubernetes.io/rewrite-target: /
Events:
  Type    Reason  Age                  From                      Message
  ----    ------  ----                 ----                      -------
  Normal  Sync    2m20s (x12 over 19m)  nginx-ingress-controller  Scheduled for sync
```

浏览器访问效果

```
curl sswang.example.com/flask
curl sswang.example.com/nginx
```

```
root@master1:~/ingress# curl sswang.example.com/flask
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-xvl8p, ServerIP: 10.244.2.7!
root@master1:~/ingress# curl sswang.example.com/flask
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-xvl8p, ServerIP: 10.244.2.7!
root@master1:~/ingress# curl sswang.example.com/nginx
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

小结

nginx认证实践

学习目标

这一节，我们从 web认证、集群认证、小结 三个方面来学习。

web认证

需求

我们准备对基础的flask应用进行https方式来进行访问

方法：
    ingress的tls规则就可以帮助我们实现对http应用进行https升级

简单实践

定制secret文件

```
mkdir tls && cd tls
(umask 077; openssl genrsa -out tls.key 2048)
openssl req -new -x509 -key tls.key -out tls.crt -subj "/CN=sswang.example.com"
-days 365
kubectl create secret tls ingress-tls --cert=./tls.crt --key=./tls.key
```

定义资源配置文件

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: sswang.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: deployment-service
            port:
              number: 80
  tls:
```

```
  - hosts:
    - sswang.example.com
    secretName: ingress-tls
```

应用资源文件

```
kubectl apply -f 06-ingress-http-tls-test.yaml
```

查看效果

```
kubectl get ingress
kubectl describe ingress
```

```
root@master1:~/ingress# kubectl get ingress
NAME           CLASS    HOSTS               ADDRESS     PORTS     AGE
ingress-test   <none>   sswang.example.com   10.0.0.16   80, 443   106s
root@master1:~/ingress# kubectl describe ingress ingress-test
Name:             ingress-test
Namespace:        default
Address:          10.0.0.16
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
TLS:
  ingress-tls terminates sswang.example.com
Rules:
  Host               Path  Backends
  ----               ----  --------
  sswang.example.com
                     /   deployment-service:80 (10.244.1.4:80,10.244.1.5:80,10.244.2.6:80 + 1 more...)
Annotations:         kubernetes.io/ingress.class: nginx
Events:
  Type    Reason  Age                 From                      Message
  ----    ------  ----                ----                      -------
  Normal  Sync    68s (x2 over 119s)  nginx-ingress-controller  Scheduled for sync
```

结果显示：

所有访问 sswang.example.com的请求都需要进行tls的处理，否则失效。

测试效果

```
curl sswang.example.com
curl https://sswang.example.com
curl -k https://sswang.example.com
```

```
root@master1:~/ingress# curl sswang.example.com
<html>
<head><title>308 Permanent Redirect</title></head>
<body>
<center><h1>308 Permanent Redirect</h1></center>
<hr><center>nginx</center>
</body>
</html>
root@master1:~/ingress# curl https://sswang.example.com
curl: (60) SSL certificate problem: self signed certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
root@master1:~/ingress# curl -k https://sswang.example.com
kubernetes pod-test v0.1!! ClientIP: 10.244.2.5, ServerName: deployment-test-95c58b447-nshhn, ServerIP: 10.244.1.4!
```

集群认证

需求

我们知道，对于k8s的dashboard来说，它所实现的功能还是比较多的，而且原则上要求必须是https方式来进行访问。我们之前是通过service的方式实现正常的请求访问。有了ingress，我们也可以自由的访问dashboard了。比如我们访问k8s的dashboard的首页地址是

```
https://10.0.0.12:30443/#/login
```

方法：

除了我们之前所说的动态url的方式之外，就需要tls认证了。关于对集群内部的ingress的tls访问方式，可以借助于ingress controller的annotation的方式向nginx中传递一些属性，从而上nginx支持相应的功能。这与我们上一节的方式有所不同

1 通过两个annotation中的规则

ingress.kubernetes.io/ssl-passthrough: "true"

- 以为后面的dashboard本身就做了tls功能，所以这里进行tls代理即可

nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"

- 对后端直接启动https协议访问即可。

参考资料：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/

定制ingress资源配置文件

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-dashboard
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/ssl-passthrough: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
  - host: sswang.example.com
    http:
      paths:
      - path: /dashboard(/|$)(.*)
        pathType: Prefix
        backend:
          service:
            name: kubernetes-dashboard
            port:
              number: 443
```

应用资源文件

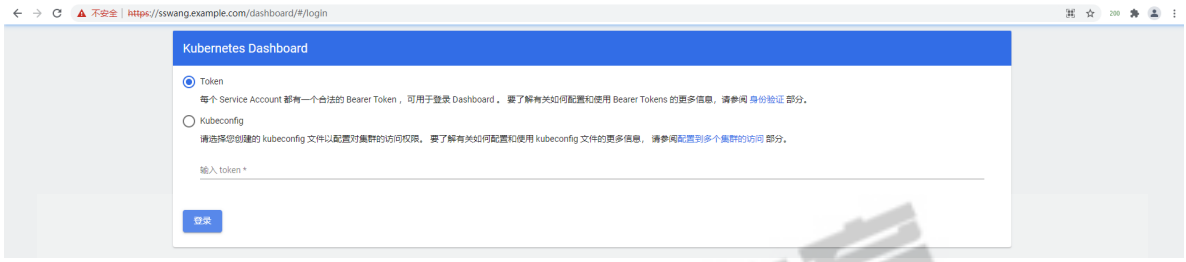
```
kubectl apply -f 07-ingress-https-dashboard.yaml
```

查看效果

```
kubectl get ingress -n kubernetes-dashboard
kubectl describe ingress -n kubernetes-dashboard
```

```
root@master1:~/ingress# kubectl get ingress -n kubernetes-dashboard
NAME                CLASS    HOSTS                ADDRESS   PORTS   AGE
ingress-dashboard   <none>   sswang.example.com             80      11s
root@master1:~/ingress# kubectl describe ingress -n kubernetes-dashboard
Name:             ingress-dashboard
Namespace:        kubernetes-dashboard
Address:
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host               Path  Backends
  ----               ----  --------
  sswang.example.com
                     /dashboard(/|$)(.*)   kubernetes-dashboard:443 (10.244.1.6:8443)
Annotations:         ingress.kubernetes.io/ssl-passthrough: true
                     kubernetes.io/ingress.class: nginx
                     nginx.ingress.kubernetes.io/backend-protocol: HTTPS
                     nginx.ingress.kubernetes.io/rewrite-target: /$2
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  Sync    17s   nginx-ingress-controller  Scheduled for sync
```

浏览器访问 https://sswang.example.com/dashboard/ 查看效果



结果显示：

借助于nginx-controller的tls功能可以正常的实现相关的https的效果。

小结