# 实验三：容器管理-k8s搭建与调度器定制



## 实验目的：

1、学习搭建Kubernetes集群，并了解集群核心组件、插件
2、学习并了解Kubernetes调度器
3、了解Kubernetes调度器代码
4、学习拓展Kubernetes调度器的方法

## 实验内容：

0、务必先阅读参考资料
1、搭建Kubernetes集群，介绍核心组件功能
2、介绍Kubernetes调度器工作基本原理
3、介绍Kubernetes调度器拓展方法
4、实践Kubernetes调度器的拓展

# 实验前的准备：

1、将参考资料阅读完毕
2、了解本次实验的环境：play with k8s平台https://labs.play-with-k8s.com/

# 实验操作步骤：

**【注】**
**1、本实验无法在超算习堂上进行。目前采用play with k8s平台进行。因此，搭建kubernetes集群的步骤非常简单（跳过了许多准备工作），只需要执行kubeadm init 和 kubeadm join 和一些必要的简单配置。**
**2、以下介绍的是在一般虚拟机上安装k8s的完整步骤，即使使用play with k8s，也请务必学习。**
**3、条件允许的同学，可以不使用play with k8s，尽量使用自己的虚拟机（多台虚拟机）完成。**

## 1、搭建Kubernetes集群

### 1.1 安装准备工作：

- 操作系统：Ubuntu Desktop 16.04 LTS
- 保证系统内存不小于2GB，CPU核心数不小于2个。
- 关闭内存交换，打开Linux终端以root用户输入命令：swapoff －a。
- 全程建议以root用户身份运行操作。

### 1.2 安装Docker

```
curl -fsSL get.docker.com -o get-docker.sh
sudo sh get-docker.sh --mirror Aliyun

sudo systemctl enable docker
sudo systemctl start docker
```

```
6
7 sudo groupadd docker
8 sudo usermod -aG docker $USER
9
10 # 自行配置镜像加速
11
12 # 配置之后，重启
13
14 sudo systemctl daemon-reload
15 sudo systemctl restart docker
```

## 1.3安装 kubeadm等

```
1 # 这步可能不成功，没关系
2 $ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | ap
  t-key add -
3
4
5 $ cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
6 deb http://mirrors.ustc.edu.cn/kubernetes/apt kubernetes-xenial main
7 EOF
8
9 $ apt-get update
10 $ apt-get install -y --allow-unauthenticated kubelet=1.16.7-00 kubea
  dm=1.16.7-00
```

查看所需镜像版本

```
root@tjrone:~# kubeadm config images list
I0606 21:04:44.637572   29742 version.go:251] remote version is much newer: v1.18.3; falling back to: stable-1.16
k8s.gcr.io/kube-apiserver:v1.16.10
k8s.gcr.io/kube-controller-manager:v1.16.10
k8s.gcr.io/kube-scheduler:v1.16.10
k8s.gcr.io/kube-proxy:v1.16.10
k8s.gcr.io/pause:3.1
k8s.gcr.io/etcd:3.3.15-0
k8s.gcr.io/coredns:1.6.2
```

编写一个脚本，用来从国内镜像源拉取以上镜像，再改名，骗过kubeadm

```
1 #!/bin/bash
```

```
images=(
  kube-apiserver:v1.16.10
  kube-controller-manager:v1.16.10
  kube-scheduler:v1.16.10
  kube-proxy:v1.16.10
  pause:3.1
  etcd:3.3.15-0
  coredns:1.6.2
)

for imageName in ${images[@]} ; do
    docker pull registry.cn-hangzhou.aliyuncs.com/google_containers/$imageName
    docker tag registry.cn-hangzhou.aliyuncs.com/google_containers/$imageName k8s.gcr.io/$imageName
    docker rmi registry.cn-hangzhou.aliyuncs.com/google_containers/$imageName
done
```

## 1.3 Master结点部署

接下来输入以下命令即可使用kubeadm初始化Kubernetes集群Master结点。
这里最好指定pod网络，不然后面部署网络插件的时候会有点bug

```
$ kubeadm init --pod-network-cidr 172.16.0.0/16
```

几分钟后会生成类型以下结果：

```
kubeadm join 10.168.0.2:6443 --token 00bwbx.uvnaa2ewjflwu1ry --discovery-token-ca-cert-hash sha256:00eb62a2a6020f94132e3fe1ab721349bbcd3e9b94da9654cfe15f2985ebd711
```

这个生成的 kubeadm join 命令，就是用来给这个 Master 节点添加更多工作节点（Worker）的命令。我们在后面部署 Worker 节点的时候马上会用到它，所以找一个地方把这条命令记录下来。

kubeadm 还会提示我们第一次使用 Kubernetes 集群所需要的配置命令：

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

需要这些配置命令的原因是：Kubernetes 集群默认需要加密方式访问。所以，这几条命令，就是将刚刚部署生成的 Kubernetes 集群的安全配置文件，保存到当前用户的.kube 目录下，kubectl 默认会使用这个目录下的授权信息访问 Kubernetes 集群。如果不这么做的话，我们每次都需要通过 export KUBECONFIG 环境变量告诉 kubectl 这个安全配置文件的位置。

现在，我们就可以使用 kubectl get 命令来查看当前唯一一个节点的状态了：

```
$ kubectl get nodes


NAME      STATUS     ROLES     AGE       VERSION
master    NotReady   master    1d        v1.11.1
```

可以看到，这个 get 指令输出的结果里，Master 节点的状态是 NotReady。

在调试 Kubernetes 集群时，最重要的手段就是用 kubectl describe 来查看这个节点（Node）对象的详细信息、状态和事件（Event）：

```
$ kubectl describe node master

...
Conditions:
...

Ready   False ... KubeletNotReady  runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni config uninitialized
```

通过 kubectl describe 指令的输出，我们可以看到 NodeNotReady 的原因在于，我们尚未部署任何网络插件。

另外，我们还可以通过 kubectl 检查这个节点上各个系统 Pod 的状态，其中，kube-system 是 Kubernetes 项目预留的系统 Pod 的工作空间（Namepsace，注意它并不是 Linux Namespace，它只是 Kubernetes 划分不同工作空间的单位）：

```
$ kubectl get pods -n kube-system

NAME                              READY   STATUS    RESTARTS   AGE
coredns-78fcdf6894-j9s52          0/1     Pending   0          1h
```

```
coredns-78fcdf6894-jm4wf          0/1     Pending     0     1h
etcd-master                       1/1     Running     0     2s
kube-apiserver-master             1/1     Running     0     1s
kube-controller-manager-master    0/1     Pending     0     1s
kube-proxy-xbd47                  1/1     NodeLost    0     1h
kube-scheduler-master             1/1     Running     0     1s
```

可以看到，CoreDNS、kube-controller-manager 等依赖于网络的 Pod 都处于 Pending 状态，即调度失败。这当然是符合预期的：因为这个 Master 节点的网络尚未就绪。

在 Kubernetes 项目“一切皆容器”的设计理念指导下，部署网络插件非常简单，只需要执行一句 kubectl apply 指令，以 Weave 为例：

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/2140ac876ef134e0ed5af15c65e414cf26827915/Documentation/kube-flannel.yml
```

最后测试集群是否正常工作。输入命令。如输出如图所示则集群已正常运行。Pod状态为running时表示已正常部署，节点状态为ready时表示正常运行。

```
kubectl get nodes

kubectl get pods –all-namespaces
```

```
root@ljl-virtual-machine:/home/ljl# kubectl get nodes
NAME                  STATUS   ROLES    AGE    VERSION
ljl-virtual-machine   Ready    master   101m   v1.17.3
root@ljl-virtual-machine:/home/ljl# kubectl get pods --all-namespaces
NAMESPACE     NAME                                          READY   STATUS    RESTARTS   AGE
kube-system   coredns-6955765f44-4kk46                      1/1     Running   0          101m
kube-system   coredns-6955765f44-z2mts                      1/1     Running   0          101m
kube-system   etcd-ljl-virtual-machine                      1/1     Running   0          101m
kube-system   kube-apiserver-ljl-virtual-machine            1/1     Running   0          101m
kube-system   kube-controller-manager-ljl-virtual-machine   1/1     Running   0          101m
kube-system   kube-flannel-ds-amd64-vg4xl                   1/1     Running   0          99m
kube-system   kube-proxy-2swb8                              1/1     Running   0          101m
kube-system   kube-scheduler-ljl-virtual-machine            1/1     Running   0          101m
```

至此，Kubernetes 的 Master 节点就部署完成了。如果你只需要一个单节点的 Kubernetes，现在你就可以使用了。不过，在默认情况下，Kubernetes 的 Master 节点是不能运行用户 Pod 的。

## 1.4 Worker 节点部署

Kubernetes 的 Worker 节点跟 Master 节点几乎是相同的，它们运行着的都是一个 kubelet 组件。

唯一的区别在于，在 kubeadm init 的过程中，kubelet 启动后，Master 节点上还会自动运行 kube-apiserver、kube-scheduler、kube-controller-manger 这三个系统 Pod。

所以，相比之下，部署 Worker 节点反而是最简单的，只需要两步即可完成。

第一步，在所有 Worker 节点上执行“安装 kubeadm 和 Docker”一节的所有步骤。

第二步，执行部署 Master 节点时生成的 kubeadm join 指令：

```
$ kubeadm join 10.168.0.2:6443 --token 00bwbx.uvnaa2ewjflwu1ry --discovery-token-ca-cert-hash sha256:00eb62a2a6020f94132e3fe1ab721349bbcd3e9b94da9654cfe15f2985ebd711
```

## 1.5 通过Taint/Toleration调整Master执行Pod的策略

在前面提到过，默认情况下 Master 节点是不允许运行用户 Pod 的。而 Kubernetes 做到这一点，依靠的是 Kubernetes 的 Taint/Toleration 机制。

简单介绍原理：

> 一旦某个节点被加上了一个 Taint，即被“打上了污点”，那么所有 Pod 就都不能在这个节点上运行，因为 Kubernetes 的 Pod 都有“洁癖”。除非，有个别的 Pod 声明自己能“容忍”这个“污点”，即声明了 Toleration，它才可以在这个节点上运行。

为节点打上“污点”（Taint）的命令是：

```
$ kubectl taint nodes node1 foo=bar:NoSchedule
```

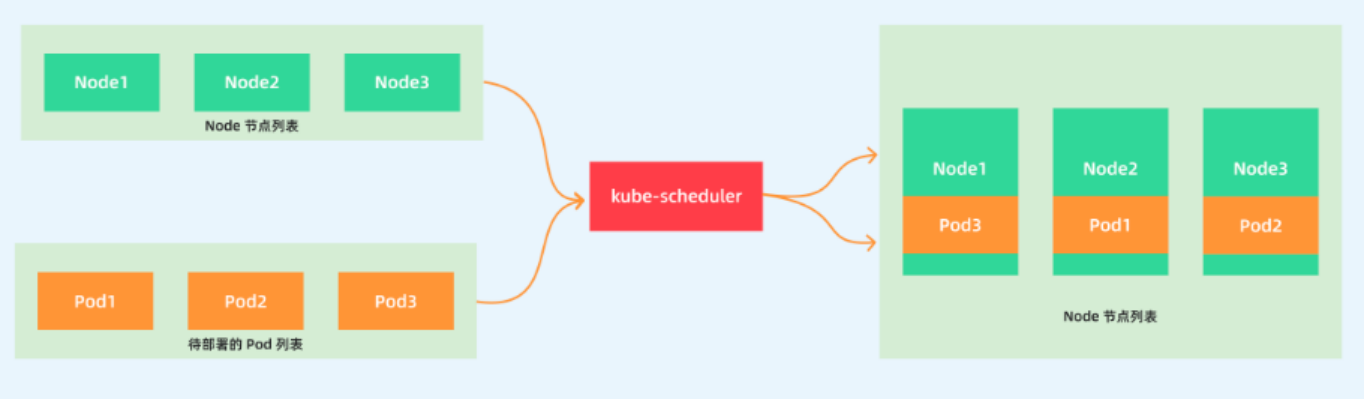
通过 kubectl describe 检查一下 Master 节点的 Taint 字段：

```
$ kubectl describe node master
...
Name:               master
Roles:              master
Taints:             node-role.kubernetes.io/master:NoSchedule
```

如果你就是想要一个单节点的 Kubernetes，删除这个 Taint 才是正确的选择：

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
```

# 2、自定义并拓展k8s调度器




Kubernetes调度示意图

## 2.1、背景知识：

一、Kubernetes 调度程序是如何工作的:

1. 默认调度器根据指定的参数启动（我们使用 kubeadm 搭建的集群，启动配置文件位于 `/etc/kubernetes/manifests/kube-schdueler.yaml`）
2. watch apiserver，将 `spec.nodeName` 为空的 Pod 放入调度器内部的调度队列中
3. 从调度队列中 Pop 出一个 Pod，开始一个标准的调度周期
4. 从 Pod 属性中检索“硬性要求”（比如 CPU/内存请求值，nodeSelector/nodeAffinity），然后过滤阶段发生，在该阶段计算出满足要求的节点候选列表
5. 从 Pod 属性中检索“软需求”，并应用一些默认的“软策略”（比如 Pod 倾向于在节点上更加聚拢或分散），最后，它为每个候选节点给出一个分数，并挑选出得分最高的最终获胜者
6. 和 apiserver 通信（发送绑定调用），然后设置 Pod 的 `spec.nodeName` 属性以表示将该 Pod 调度到的节点。

二、有4种扩展并自定义 Kubernetes 调度器的方法：

1、一种方法就是直接 clone 开源的 kube-scheduler 源代码，在合适的位置直接修改代码，然后重新编译运行修改后的程序，本实验不建议使用这种方法，因为需要花费大量额外的精力来和上游的调度程序更改保持一致。

2、第二种方法就是开发并运行一个和默认的调度程序独立的调度程序scheduler，默认的调度器和我们自定义的调度器可以通过 Pod 的 spec.schedulerName 来覆盖各自的 Pod，默认是使用 default 默认的调度器，但是多个调度程序共存的情况下也比较麻烦。比如当多个调度器将 Pod 调度到同一个节点的时候，可能会遇到一些问题。因为很有可能两个调度器都同时将两个 Pod 调度到同一个节点上去，但是很有可能其中一个 Pod 运行后其实资源就消耗完了。

3、第三种方法是scheduler extender，这个方案目前是一个可行的方案，可以和上游调度程序兼容，所谓的scheduler extender其实就是一个可配置的 Webhook 而已，里面包含 filter 和 priority 两个端点，分别对应调度周期中的两个主要阶段（过滤和打分）。

> 4、第四种方法是通过调度框架（Scheduling Framework），Kubernetes v1.15 版本中引入了可插拔架构的调度框架，使得定制调度器这个任务变得更加的容易。调库框架向现有的调度器中添加了一组插件化的 API，该 API 在保持调度程序“核心”简单且易于维护的同时，使得大部分的调度功能以插件的形式存在，而且在我们现在的 v1.16 版本中上面的 调度器扩展程序 也已经被废弃了，所以以后调度框架才是自定义调度器的核心方式。

本实验使用第三种方法，实现一个自定义的K8s Scheduler：

## 2.2、查看集群中的scheduler的YAML配置文件

```
root@master:/etc/kubernetes# kubectl get pod -n kube-system kube-scheduler-master -o yaml
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    。。。
  creationTimestamp: "2019-11-05T10:03:15Z"
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler-master
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    image: k8s.gcr.io/kube-scheduler:v1.13.4
    imagePullPolicy: IfNotPresent
    livenessProbe:
      ...
    name: kube-scheduler
    resources:
      requests:
        cpu: 100m
```

```
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  hostNetwork: true
  nodeName: master
  priority: 2000000000
  priorityClassName: system-cluster-critical
  restartPolicy: Always
  schedulerName: default-scheduler
  tolerations:
  - effect: NoExecute
    operator: Exists
  volumes:
  - hostPath:
      path: /etc/kubernetes/scheduler.conf
      type: FileOrCreate
    name: kubeconfig
```

通过kube-scheduler的启动命令，以及volumes字段的内容，可以看出目前的kube-scheduler并未配置extender。

## 2.3、编写extender代码

我们直接用 golang 来实现一个简单的调度器扩展程序，当然你可以使用其他任何编程语言，示例如下所示：**请同学们自己写这部分代码，作业要交。**示例仓库：https://github.com/tjrone/sample-scheduler-extender

```
func main() {
    router := httprouter.New()
    router.GET("/", Index)
    router.POST("/filter", Filter)
    router.POST("/prioritize", Prioritize)

    log.Fatal(http.ListenAndServe(":8888", router))
```

```
}
```

我们实现 /filter 和 /prioritize 两个功能的handler function处理程序。Filter 这个扩展函数接收一个输入类型为 schedulerapi.ExtenderArgs 的参数，然后返回一个类型为 *schedulerapi.ExtenderFilterResult 的值。在函数中，我们可以进一步过滤输入的节点：

```
// filter 根据扩展程序定义的预选规则来过滤节点
func filter(args schedulerapi.ExtenderArgs) *schedulerapi.ExtenderFilterResult {
    var filteredNodes []v1.Node
    failedNodes := make(schedulerapi.FailedNodesMap)
    pod := args.Pod

    for _, node := range args.Nodes.Items {
        fits, failReasons, _ := podFitsOnNode(pod, node)
        if fits {
            filteredNodes = append(filteredNodes, node)
        } else {
            failedNodes[node.Name] = strings.Join(failReasons, ",")
        }
    }

    result := schedulerapi.ExtenderFilterResult{
        Nodes: &v1.NodeList{
            Items: filteredNodes,
        },
        FailedNodes: failedNodes,
        Error:       "",
    }

    return &result
}
```

在过滤函数中，我们循环每个节点然后用我们**自己实现的业务逻辑来判断**是否应该批准该节点，这里示例实现比较简单，在 `podFitsOnNode()` 函数中我们只是简单的检查随机数是否为偶数来判断即可，如果是的话我们就认为这是一个“被批准”的节点，否则拒绝批准该节点。

```
var predicatesSorted = []string{LuckyPred}
```

```
2
3  var predicatesFuncs = map[string]FitPredicate{
4      LuckyPred: LuckyPredicate,
5  }
6
7  type FitPredicate func(pod *v1.Pod, node v1.Node) (bool, []string, e
   rror)
8
9  func podFitsOnNode(pod *v1.Pod, node v1.Node) (bool, []string, error
   ) {
10     fits := true
11     var failReasons []string
12     for _, predicateKey := range predicatesSorted {
13         fit, failures, err := predicatesFuncs[predicateKey](pod, nod
   e)
14         if err != nil {
15             return false, nil, err
16         }
17         fits = fits && fit
18         failReasons = append(failReasons, failures...)
19     }
20     return fits, failReasons, nil
21 }
22
23 func LuckyPredicate(pod *v1.Pod, node v1.Node) (bool, []string, erro
   r) {
24     lucky := rand.Intn(2) == 0
25     if lucky {
26         log.Printf("pod %v/%v is lucky to fit on node %v\n", pod.Nam
   e, pod.Namespace, node.Name)
27         return true, nil, nil
28     }
29     log.Printf("pod %v/%v is unlucky to fit on node %v\n", pod.Name,
   pod.Namespace, node.Name)
30     return false, []string{LuckyPredFailMsg}, nil
31 }
```

同样的打分功能用同样的方式来实现，我们在每个节点上随机给出一个分数：

```go
// it's webhooked to pkg/scheduler/core/generic_scheduler.go#PrioritizeNodes()
// 这个函数输出的分数会被添加会默认的调度器
func prioritize(args schedulerapi.ExtenderArgs) *schedulerapi.HostPriorityList {
    pod := args.Pod
    nodes := args.Nodes.Items

    hostPriorityList := make(schedulerapi.HostPriorityList, len(nodes))
    for i, node := range nodes {
        score := rand.Intn(schedulerapi.MaxPriority + 1)  // 在最大优先级内随机取一个值
        log.Printf(luckyPrioMsg, pod.Name, pod.Namespace, score)
        hostPriorityList[i] = schedulerapi.HostPriority{
            Host:  node.Name,
            Score: score,
        }
    }

    return &hostPriorityList
}
```

## 4、编译、运行extender

使用下面的命令来编译打包我们的应用：

```
$ GOOS=linux GOARCH=amd64 go build -o extender
```

构建完成后，将应用 `extender` 拷贝到 `kube-scheduler` 所在的节点（master结点）直接运行即可：

```
$ ./extender
```

## 5、配置策略文件

现在我们就需要编辑一个策略文件，并将策略文件配置到 kube-scheduler 组件中。

### 5.1 首先编辑KubeSchedulerConfiguration对象

调度器kube-scheduler 的启动参数 --config会指向包含KubeSchedulerConfiguration对象的文件地址，例如将以下YAML文件存放在 /etc/kubernetes/scheduler-extender.yaml

```yaml
# 通过"--config" 传递文件内容
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
algorithmSource:
  policy:
    file:
      path: "/etc/kubernetes/scheduler-extender-policy.json" # 指定自定义调度策略文件
```

以上文件的关键参数是 algorithmSource.policy，这个策略文件，这是我们重点关注的。

### 5.2 编辑策略文件

该策略文件 /etc/kubernetes/scheduler-extender-policy.yaml编辑如下：

```json
{
    "kind" : "Policy",
    "apiVersion" : "v1",
    "extenders" : [{
        "urlPrefix": "http://localhost:8888/",
        "filterVerb": "filter",
        "prioritizeVerb": "prioritize",
        "weight": 1,
        "enableHttps": false
    }]
}
```

该策略文件定义了一个 HTTP 的扩展程序服务，该服务运行在 127.0.0.1:8888 下面，并且已经将该策略注册到了默认的调度器中，这样在过滤和打分阶段结束后，可以将结果分别传递给该扩展程序的端点 <urlPrefix>/<filterVerb> 和 <urlPrefix>/<prioritizeVerb>

### 5.3 配置该策略文件

我们这里集群是 kubeadm 搭建的，因此，kube-scheduler是以static pod方式启动的（需了解static pod的概念及功能，请参考资料），因此直接修改文件

/etc/kubernetes/manifests/kube-schduler.yaml 文件即可，内容如下所示：（重点关注：1、 --config参数配置。2、volume挂载的变化）

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --authentication-kubeconfig=/etc/kubernetes/scheduler.conf
    - --authorization-kubeconfig=/etc/kubernetes/scheduler.conf
    - --bind-address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    - --config=/etc/kubernetes/scheduler-extender.yaml
    - --v=9
    image: gcr.azk8s.cn/google_containers/kube-scheduler:v1.16.2
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 8
      httpGet:
        host: 127.0.0.1
        path: /healthz
        port: 10251
        scheme: HTTP
      initialDelaySeconds: 15
      timeoutSeconds: 15
    name: kube-scheduler
    resources:
      requests:
        cpu: 100m
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
```

```
      name: kubeconfig
      readOnly: true
    - mountPath: /etc/kubernetes/scheduler-extender.yaml
      name: extender
      readOnly: true
    - mountPath: /etc/kubernetes/scheduler-extender-policy.json
      name: extender-policy
      readOnly: true
  hostNetwork: true
  priorityClassName: system-cluster-critical
  volumes:
  - hostPath:
      path: /etc/kubernetes/scheduler.conf
      type: FileOrCreate
    name: kubeconfig
  - hostPath:
      path: /etc/kubernetes/scheduler-extender.yaml
      type: FileOrCreate
    name: extender
  - hostPath:
      path: /etc/kubernetes/scheduler-extender-policy.json
      type: FileOrCreate
    name: extender-policy
status: {}
```

## 6、检查scheduler重启

kube-scheduler 重新配置后可以查看日志来验证是否重启成功：

```
$ kubectl logs -f kube-scheduler-ydzs-master -n kube-system
I0102 15:17:38.824657       1 serving.go:319] Generated self-signed cert in-memory
I0102 15:17:39.472276       1 server.go:143] Version: v1.16.2
I0102 15:17:39.472674       1 defaults.go:91] TaintNodesByCondition is enabled, PodToleratesNodeTaints predicate is mandatory
W0102 15:17:39.479704       1 authorization.go:47] Authorization is disabled
W0102 15:17:39.479733       1 authentication.go:79] Authentication is disabled
```

```
I0102 15:17:39.479777       1 deprecated_insecure_serving.go:51] Serving healthz insecurely on [::]:10251
I0102 15:17:39.480559       1 secure_serving.go:123] Serving securely on 127.0.0.1:10259
I0102 15:17:39.682180       1 leaderelection.go:241] attempting to acquire leader lease  kube-system/kube-scheduler...
I0102 15:17:56.500505       1 leaderelection.go:251] successfully acquired lease kube-system/kube-scheduler
```

## 7、测试拓展后的调度器

对于示例extender，这里运行一个 Deployment ，来测试我们的调度器的工作情况，我们准备一个包含**20个副本**的deployment Yaml：(test-scheduler.yaml)

该pause镜像没有什么实际功能。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause
spec:
  replicas: 20
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: gcr.azk8s.cn/google_containers/pause:3.1
```

运行以上的对象：

```
$ kuectl apply -f test-scheduler.yaml
deployment.apps/pause created
```

查看extender打印的日志信息，会输出类似以下的运行结果：

```
$ ./extender
......
2020/01/03 12:27:29 pod pause-58584fbc95-bwn7t/default is unlucky to fit on node ydzs-node1
2020/01/03 12:27:29 pod pause-58584fbc95-bwn7t/default is lucky to get score 7
2020/01/03 12:27:29 pod pause-58584fbc95-bwn7t/default is lucky to get score 9
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is unlucky to fit on node ydzs-node3
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is unlucky to fit on node ydzs-node4
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is lucky to fit on node ydzs-node1
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is lucky to fit on node ydzs-node2
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is lucky to get score 4
2020/01/03 12:27:29 pod pause-58584fbc95-86w92/default is lucky to get score 8
......
```

我们可以看到 Pod 调度的过程，另外默认调度程序会定期重试失败的 Pod，因此它们将一次又一次地重新传递到我们的调度扩展程序上，我们的逻辑是检查随机数是否为偶数，所以最终所有 Pod 都将处于运行状态。

**注：测试部分，请同学们设计自己的测试方法。作业说明自己的测试方法和测试结果。**

# 参考资料：

📎 10 Kubernetes一键部署利器：kubeadm.html📎 11 从0到1：搭建一个完整的Kubernetes集群.html
📎 41 十字路口上的Kubernetes默认调度器.html