koordinator Spark 作业托管最佳实践

参考自koordinator官网

```
https://koordinator.sh/zh-Hans/docs/best-practices/colocation-of-spark-jobs/
```

下载koordinator源码

```
wget https://github.com/koordinator-sh/koordinator/archive/refs/tags/v1.4.0.tar.gz
```

使用helm安装Apache Spark Operator

进入源码目录

```
cd koordinator-1.4.0/examples
```

创建命名空间

```
kubectl create namespace spark-operator
```

使用helm 创建spark operator

```
helm install koord-spark-operator ./spark-operator-chart/ --namespace spark-operator
```

验证spark 是否成功运行

```
helm status --namespace spark-operator koord-spark-operator
```

创建命名空间spark-demo和服务帐户spark

```
kubectl apply -f examples/spark-jobs/service-account.yaml
```

创建 Colocation Profile,以便在命名空间 Spark-demo 中创建的所有 Pod 都将以 Colocation 模式运行

```
kubectl apply -f examples/spark-jobs/cluster-colocation-profile.yaml
```

cluster-colocation-profile.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: spark-demo
  labels:
    koordinator.sh/enable-colocation: "true"
---
apiVersion: config.koordinator.sh/v1alpha1
kind: ClusterColocationProfile
metadata:
  name: spark-demo
spec:
  namespaceSelector:
    matchLabels:
      koordinator.sh/enable-colocation: "true" //
  selector:
    matchLabels:
      sparkoperator.k8s.io/launched-by-spark-operator: "true"
  qosClass: BE
  priorityClassName: koord-batch
  koordinatorPriority: 1000
  schedulerName: koord-scheduler
```

- qosClass是服务质量，分别有枚举类型LSE、LSR、LS、BE 和 SYSTEM

SYSTEM	**系统进程，资源受限 对于 DaemonSets 等系统服务，虽然需要保证系统服务的延迟，但也需要限制节点上这些系统服务容器的资源使用，以确保其不占用过多的资源**
LSE(Latency Sensitive Exclusive)	保留资源并组织同 QoS 的 pod 共享资源 很少使用，常见于中间件类应用，一般在独立的资源池中使用
LSR(Latency Sensitive Reserved)	预留资源以获得更好的确定性 类似于社区的 Guaranteed，CPU 核被绑定
LS(Latency Sensitive)	共享资源，对突发流量有更好的弹性 微服务工作负载的典型QoS级别，实现更好的资源弹性和更灵活的资源调整能力
BE(Best Effort)	共享不包括 LSE 的资源，资源运行质量有限，甚至在极端情况下被杀死 批量作业的典型 QoS 水平，在一定时期内稳定的计算吞吐量，低成本资源

- priorityClassName

指定要写入到 Pod.Spec.PriorityClassName 中的 Kubenretes PriorityClass. 选项为 koord-prod、koord-mid、koord-batch 和 koord-free。

优先级:

PriorityClass	优先级范围	描述
koord-prod	[9000, 9999] 需要提前规划资源配额，并且保证在配额内成功。	
koord-mid	[7000, 7999] 需要提前规划资源配额，并且保证在配额内成功。	
koord-batch	[5000, 5999] 需要提前规划资源配额，一般允许借用配额。	
koord-free	[3000, 3999] 不保证资源配额，可分配的资源总量取决于集群的总闲置资源。	

koordinatorPriority要和 priorityClassName 搭配使用，比如

```
koord-prod => 9911
```

schedulerName: 如果指定，则 Pod 将由指定的调度器调度。 Koordinator 在 Kubernetes 集群中部署时会初始化这四个 PriorityClass。

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: koord-prod
value: 9000
description: "This priority class should be used for prod service pods
only."
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: koord-mid
value: 7000
description: "This priority class should be used for mid service pods
only."
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: koord-batch
value: 5000
description: "This priority class should be used for batch service pods
only."
---
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: koord-free
value: 3000
```

```
description: "This priority class should be used for free service pods
only."
```

schedulerName: koord-scheduler,配置在configMap里

```
Koord-Scheduler
Koord-Scheduler 以 Deployment 的形式部署在集群中，用于增强 Kubernetes 在 QoS-
aware，差异化 SLO 以及任务调度场景的资源调度能力，具体包括:

QoS-aware 调度，包括负载感知调度让节点间负载更佳平衡，资源超卖的方式支持运行更多的低优先
级工作负载。
差异化 SLO，包括 CPU 精细化编排，为不同的工作负载提供不同的 QoS 隔离策略（cfs，LLC，
memory 带宽，网络带宽，磁盘io）。
任务调度，包括弹性额度管理，Gang 调度，异构资源调度等，以支持更好的运行大数据和 AI 工作
负载。
为了更好的支持不同类型的工作负载，Koord-scheduler 还包括了一些通用性的能力增强：

Reservation，支持为特定的 Pod 或者工作负载预留节点资源。资源预留特性广泛应用于重调度，
资源抢占以及节点碎片整理等相关优化过程。
Node Reservation，支持为 kubernetes 之外的工作负载预留节点资源，一般应用于节点上运行
着非容器化的负载场景。
```

configMap 定义：

```
apiVersion: v1
data:
  koord-scheduler-config: |
    apiVersion: kubescheduler.config.k8s.io/v1beta2
    kind: KubeSchedulerConfiguration
    leaderElection:
      leaderElect: true
      resourceLock: leases
      resourceName: koord-scheduler
      resourceNamespace: koordinator-system
    profiles:
      - pluginConfig:
        - name: NodeResourcesFit
          args:
            apiVersion: kubescheduler.config.k8s.io/v1beta2
            kind: NodeResourcesFitArgs
            scoringStrategy:
              type: LeastAllocated
              resources:
                - name: cpu
                  weight: 1
                - name: memory
                  weight: 1
```

```
                - name: "kubernetes.io/batch-cpu"
                  weight: 1
                - name: "kubernetes.io/batch-memory"
                  weight: 1
        - name: LoadAwareScheduling
          args:
            apiVersion: kubescheduler.config.k8s.io/v1beta2
            kind: LoadAwareSchedulingArgs
            filterExpiredNodeMetrics: false
            nodeMetricExpirationSeconds: 300
            resourceWeights:
              cpu: 1
              memory: 1
            usageThresholds:
              cpu: 65
              memory: 95
            # disable by default
            # prodUsageThresholds indicates the resource utilization
threshold of Prod Pods compared to the whole machine.
            # prodUsageThresholds:
            #   cpu: 55
            #   memory: 75
            # scoreAccordingProdUsage controls whether to score according
to the utilization of Prod Pod
            # scoreAccordingProdUsage: true
            # aggregated supports resource utilization filtering and
scoring based on percentile statistics
            # aggregated:
            #   usageThresholds:
            #     cpu: 65
            #     memory: 95
            #   usageAggregationType: "p95"
            #   scoreAggregationType: "p95"
            estimatedScalingFactors:
              cpu: 85
              memory: 70
        - name: ElasticQuota
          args:
            apiVersion: kubescheduler.config.k8s.io/v1beta2
            kind: ElasticQuotaArgs
            quotaGroupNamespace: koordinator-system
        plugins:
          queueSort:
            disabled:
              - name: "*"
            enabled:
              - name: Coscheduling
          preFilter:
            enabled:
              - name: Reservation
              - name: NodeNUMAResource
              - name: DeviceShare
              - name: Coscheduling
              - name: ElasticQuota
```

```
      filter:
        enabled:
          - name: LoadAwareScheduling
          - name: NodeNUMAResource
          - name: DeviceShare
          - name: Reservation
      postFilter:
        disabled:
          - name: "*"
        enabled:
          - name: Reservation
          - name: Coscheduling
          - name: ElasticQuota
          - name: DefaultPreemption
      preScore:
        enabled:
          - name: Reservation # The Reservation plugin must come first
      score:
        enabled:
          - name: LoadAwareScheduling
            weight: 1
          - name: NodeNUMAResource
            weight: 1
          - name: DeviceShare
            weight: 1
          - name: Reservation
            weight: 5000
      reserve:
        enabled:
          - name: Reservation # The Reservation plugin must come first
          - name: LoadAwareScheduling
          - name: NodeNUMAResource
          - name: DeviceShare
          - name: Coscheduling
          - name: ElasticQuota
      permit:
        enabled:
          - name: Coscheduling
      preBind:
        enabled:
          - name: NodeNUMAResource
          - name: DeviceShare
          - name: Reservation
          - name: DefaultPreBind
      bind:
        disabled:
          - name: "*"
        enabled:
          - name: Reservation
          - name: DefaultBinder
      postBind:
        enabled:
          - name: Coscheduling
    schedulerName: koord-scheduler
```

```
kind: ConfigMap
metadata:
  annotations:
    meta.helm.sh/release-name: koordinator
    meta.helm.sh/release-namespace: default
  creationTimestamp: "2024-01-27T19:14:20Z"
  labels:
    app.kubernetes.io/managed-by: Helm
  name: koord-scheduler-config
  namespace: koordinator-system
  resourceVersion: "125368"
  uid: 547aadc7-0b9e-4a52-b4ab-f5954bd6d78d
```

使用以下命令将 Spark TC 示例作业提交到命名空间 Spark-demo：

```
kubectl apply -f examples/spark-jobs/spark-tc-complex.yaml
```

提交spark tc 作业

```
kubectl apply -f examples/spark-jobs/spark-tc-complex.yaml
```

检查spark 应用的状态：

```
kubectl get sparkapplication -n spark-demo spark-tc-complex
```

实际服务器只有4C 所以需要下降spark 内存的限制，修改spark-tc-complex.yaml.

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  namespace: spark-demo
  name: spark-tc-complex
spec:
  type: Scala
  mode: cluster
  image: "docker.io/koordinatorsh/spark:v3.2.1-koord-examples"
  imagePullPolicy: IfNotPresent
  mainClass: org.apache.spark.examples.SparkTC
  mainApplicationFile: "local:///opt/spark/examples/jars/spark-
examples_2.12-3.2.1-tc1.3.jar"
  sparkVersion: "3.2.1"
  restartPolicy:
    type: Never
  volumes:
```

```
    - name: "test-volume"
      hostPath:
        path: "/tmp"
        type: Directory
  driver:
    cores: 1
    coreLimit: "1"
    memory: "1g"
    labels:
      version: 3.2.1
    serviceAccount: spark
    volumeMounts:
      - name: "test-volume"
        mountPath: "/tmp"
  executor:
    cores: 1
    coreLimit: "1"
    instances: 1
    memory: "1g"
    labels:
      version: 3.2.1
    volumeMounts:
      - name: "test-volume"
        mountPath: "/tmp"
```

发现之前的cpu 内存使用并没有使用那么多

```
$ kubectl describe node
    Allocated resources:
    Resource                 Requests
    cpu                      7620m (95.25%)

$ kubectl top node
    NAME                            CPU(cores)             CPU%
    cn-hangzhou.your-node-1         1190m                  14.8%
    cn-hangzhou.your-node-2         1620m                  20.25%
```

调度后:

```
root@iZbp118td5g42g53vdnvgrZ:~/koordinator-1.4.0# kubectl top node
NAME                      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
izbp118td5g42g53vdnvgqz   2463m        61%    3903Mi          54%
izbp118td5g42g53vdnvgrz   163m         4%     1524Mi          21%
```

总结:

使用 cluster-colocation-profile.yaml 托管运行后主机的cpu 使用了明显提高了，减少了cpu调度的浪费