# 调度算法模块接口

1. 增加或减少pod的接口

接口功能：请求增加/减少某服务的pod，请求会被加进调度请求队列，等待下一轮调度时被读取.

URL:http://hostname:port/schedulePod

支持格式:json

HTTP请求方式：POST

请求参数：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 必选 | 类型 | 说明 |
| podList | true | Json字符串 | 单个或多个pod |

Pod：

|  |  |  |  |
| --- | --- | --- | --- |
| key | 必选 | value类型 | 说明 |
| operation | true | int | 0增加，1减少 |
| namespace | true | string | 服务所在的命名空间 |
| serviceName | true | string | 服务名 |
| number | true | string | 要增加/减少该类型的pod的数量 |

返回字段：

|  |  |  |
| --- | --- | --- |
| 返回字段 | 字段类型 | 说明 |
| isSucceed | boolean | 是否成功加进调度请求队列 |

# 开发任务

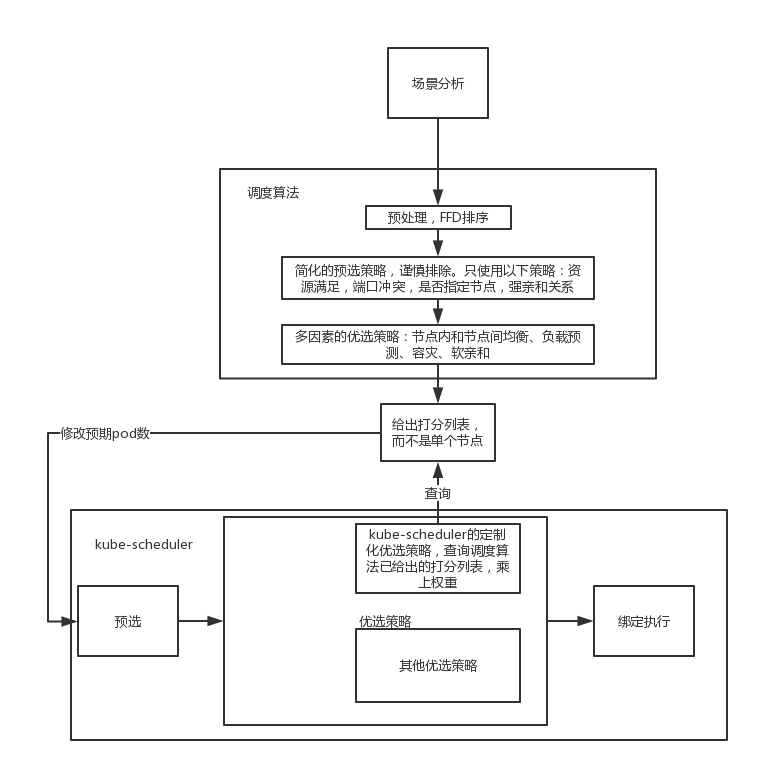
1. 预处理策略
2. 预选策略
   1. PodFitsResources
   2. PodFitsHostPorts
   3. HostName
   4. MatchNodeSelector
   5. PodToleratesNodeTaints
   6. CheckNodeMemoryPressure, CheckNodeDiskPressure
   7. MatchInterPodAffinity
3. 优选策略
   1. LeastRequestedPriority
   2. BalancedResourceAllocation
   3. InterPodAffinityPriority
   4. SelectorSpreadPriority
   5. NodeAffinityPriority
   6. NodePreferAvoidPodsPriority
   7. TaintTolerationPriority
   8. ImageLocalityPriority optional
   9. NodeLoadForecastPriority 负载预测优选
4. 并行优化
   1. 预选node->rule，node并行
   2. 优选rule->node，rule并行
5. 短期缓存维护
6. 消息队列维护

**面向的问题**

调度算法主要工作是在场景分析出的结果上再加工。确定调查的调度顺序，用最小的步骤和资源完成需要调度的pod，调度算法主要处理下面几种调度情况：

1. 添加pod：确定哪些pod需要被添加到哪些机器上；
2. 删除pod：确定哪些机器上的哪些pod需要删除；

**流程**



**预处理presort**

做完上一轮调度后，本轮调度首先从消息队列中取出所有要增加/减少的pod。 对要增加/减少的pod排序，排序优先依据：

1. 删除 优先于 增加
2. 相对占用的资源大小，通过ServiceName查询服务画像里的[资源密集类型, 当前负载资源占用, CPU Mem request值]。如果是cpu密集型的，优先级=此pod要占用的cpu / cpu密集型pods要占用的总cpu。

**预选predicates**

按预处理产生的顺序选取要增加/减少的pod，为每一个pod评估各个节点。筛选出此pod可选的机器节点，

* 增加pod时的筛选依据(GeneralPredicates)有：

1. 检查节点是否有足够资源 PodFitsResources （需要动态数据）
2. HostPort冲突 PodFitsHostPorts （需要动态数据）
3. PodSpec的NodeName字段中指定节点 PodSpec的NodeName （需要静态数据）
4. Pod的nodeSelector和NodeAffinity和PodAffinity定义的强约束标签 PodSelectorMatches MatchInterPodAffinity（需要静态数据）
5. PodToleratesNodeTaints （需要静态数据）
6. CheckNodeMemoryPressure, CheckNodeDiskPressure（需要动态数据）

* 减少pod时的筛选依据有：

1. 机器上有此类型的pod（需要动态数据）
2. 删除后不能违反应用应用强亲和关系，比如PodAffinity定义（需要静态数据）

* 已经明确不考虑的预选策略：

1. NoDiskConflict：pod所需的卷是否和节点已存在的卷冲突。如果节点已经挂载了某个卷，其它同样使用这个卷的pod不能再调度到这个主机上。我们没有这个限制
2. NoVolumeZoneConflict：检查给定的zone限制前提下，检查如果在此主机上部署Pod是否存在卷冲突。我们没有设置zone-labels。
3. MaxEBSVolumeCount, MaxGCEPDVolumeCount, MaxAzureDiskVolumeCount：我们没有存储卷数量上限的设置。

**优选priorities**

* 增加pod时，对预选阶段得到的可选节点逐个打分，打分策略考虑：

1. 优先使用空闲的节点。weight1 \* 增加pod后节点空闲资源与节点总容量的比值。 LeastRequestedPriority
2. 尽量使CPU内存等利用率一致，避免出现类似CPU很高但内存很低这种情况。weight2 \* 增加pod后各个类型的资源占用最小与最大之比。BalancedResourceAllocation
3. 为了容灾考虑，尽量不把同一服务下的pod放在一起。 weight3\*其他node部署此服务的pod数与此服务总pod数之比。 SelectorSpreadPriority
4. 还可以额外定义规定某些应用倾向于放在一起、某些应用和某些node不倾向于放在一起等等软亲和约束。weight4 \* ( Σ 命中的规则对应的程度分数 )。可以包括NodeAffinity和PodAffinity中定义的软约束标签。 NodeAffinityPriority TaintTolerationPriority
5. 另外从服务负载预测可以得知接下来的一段时间哪些节点上的负载会变高或低，因此调整该node的得分。weight5 \* ( ± 即将减少/增加的资源占机器资源的比例 )

score = Σ weighti \* score~i ~选择最大得分的机器增加这个pod，同时修改这个机器的信息上的资源信息，进行下个一个pod的处理。

* 减少pod时，对预选阶段得到的可选节点逐个打分，打分策略考虑：

1. 优先考虑繁忙的节点。weight1 \* 减少pod后节点占用资源与节点总容量的比值。
2. 尽量使CPU内存等利用率一致，避免出现类似CPU很高但内存很低这种情况。weight2 \* 减少pod后各个类型的资源占用最小与最大之比。
3. 为了容灾考虑，尽量减少同一node下的相同服务的pod。 weight3 \* 此node部署此服务的pod数与此服务总pod数之比。
4. 还可以额外定义规定某些应用倾向于放在一起，尽量不违反这个软约束。weight4 \* ( Σ 命中的规则对应的程度分数 )。可以包括NodeAffinity和PodAffinity中定义的软约束标签。
5. 另外从服务负载预测可以得知接下来的一段时间哪些节点上的pod的负载会变高或低，从而影响node的负载，因此调整该node的得分。weight5 \* ( ± 即将减少/增加的资源占机器资源的比例 )

score = Σ weighti \* score~i ~选择最大得分的机器增加这个pod，同时修改这个机器的信息上的资源信息，进行下个一个pod的处理。