# 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数组 | 单个或多个pod |

Pod：

|  |  |  |  |
| --- | --- | --- | --- |
| key | 必选 | value类型 | 说明 |
| operation | true | int | 1增加，2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 |
| clusterMasterIP | True | string | 集群master节点IP |

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 |

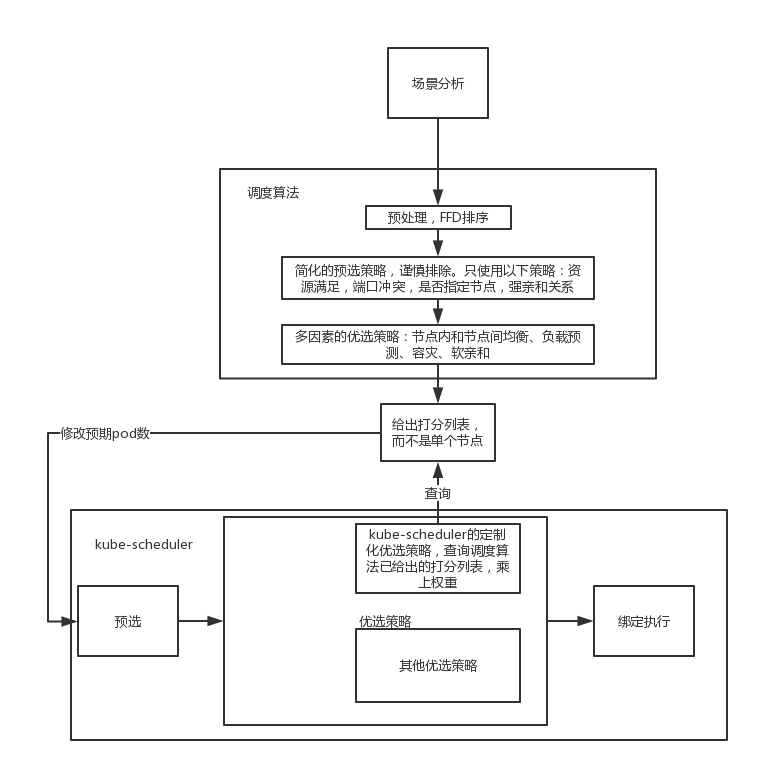
**面向的问题**

调度算法模块主要工作是接收调控模块的容器调度请求，即增加或减少指定服务下的指定数量的pod。模块的输出是这些pod的增加或减少要分别在那些节点上进行，输出交给调度执行器执行。调度算法模块的工作类似于K8s的Kube-scheduler。

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，排除可能会调度失败的节点。

* 增加pod时的筛选依据有：

1. CheckNodeConditionPredicate节点状态检查，排除未就绪、网络不可用或被标记为不可调度的节点。
2. CheckNodeDiskPressurePredicate硬盘压力检查，排除处于硬盘压力状态的节点，否则新容器会被节点上的Kubelet进程驱逐。
3. CheckNodeMemoryPressurePredicate内存压力检查，如果新容器的服务质量等级较高，则排除处于内存压力状态的节点。
4. CheckNodePIDPressurePredicate进程分配压力检查，排除处于进程分配压力状态的节点。
5. CheckVolumeBindingPredicate卷绑定检查，检查容器需要的云服务供应商的存储卷的拓扑要求，由于落地的生产环境没有使用外部存储卷供应商，也没有拓扑要求，而且也没有相关的数据可做测试，最后没有采用这个规则。
6. MatchInterPodAffinityPredicate容器间硬约束亲和性检查，容器本身可以设置与其他容器有哪些偏好或冲突，新容器需要满足节点上已有容器的硬约束反亲和性，同时也要满足新容器本身的硬约束亲和性与反亲和性。
7. PodFitsHostPortsPredicate容器暴露主机端口检查，排除容器需要暴露的端口已经被占用的节点。
8. PodFitsResourcesPredicate最低资源满足检查，节点余下可分配的资源需要满足容器需要的最低cpu和内存请求。
9. PodMatchNodeSelectorPredicate节点选择器检查，如果容器设置了对节点的选择器或对节点的硬约束亲和性，则需要节点满足对应的域描述的要求。
10. PodToleratesNodeTaintsPredicate污点容忍检查，节点由于种种原因可能被标记上污点，只有新容器明确可以容忍节点上所有类型的污点的时候，此节点才不会被排除。

* 减少pod时的筛选依据有：

1. PodExistingOnNodePredicate机器上有此类型的pod（需要动态数据）

* 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，也没有第三方云服务存储提供商。

**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. BalancedResourceAllocationPriority节点内资源均衡策略，节点分配此容器后，如果各项资源分配率相差很大，节点得分会变低。
2. ImageLocalityPriority镜像本地化策略，节点上已有的镜像越多，下载镜像的网络成本和存储成本越低，节点得分越高，但是最终没有拿到足够的监控数据，最后没有采用这个规则。
3. InterPodAffinityPriority容器间软亲和约束策略，与预选约束的容器间硬约束亲和性检查类似，但是软亲和不要求一定满足，只是以一定权重影响最终得分。
4. NodeAffinityPriority节点软约束亲和性策略，与节点的硬约束亲和性类似。
5. NodeLoadForecastPriority节点负载预测策略，根据下个时间片里节点的负载趋势，提前调整节点的优选得分，但由于依赖的预测数据不准确，最后没有采用。
6. NodePreferAvoidPodsPriority节点避免策略，有些节点可能标记了要特意尽量避开一些容器，但是这是一个软约束，实在无法避开时也可以接受。由于没有拿到相应的标记数据，这个策略最后没有采用。
7. RequestedPriority节点间资源均衡策略，节点分配此容器后，剩余资源比例约高，节点得分越高。
8. SelectorSpreadPriority服务容灾策略，如果同一个服务大部分容器都在少数几个节点上，那么节点宕机、故障造成的后果会较为严重，因此需要分散服务的容器到不同的节点上，该节点拥有此服务下容器越多，得分越低。
9. TaintTolerationPriority软约束污点容忍策略，与预选策略的污点容忍检查类似，但只是以一定权重影响最终得分。

score = Σ weighti \* scorei 选择最大得分的机器增加这个pod，同时修改临时缓存，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. BalancedResourceAllocationPriority节点内资源均衡策略，节点分配此容器后，如果各项资源分配率相差很大，节点得分会变低。
2. RequestedPriority节点间资源均衡策略，节点分配此容器后，剩余资源比例约高，节点得分越高。
3. SelectorSpreadPriority服务容灾策略，如果同一个服务大部分容器都在少数几个节点上，那么节点宕机、故障造成的后果会较为严重，因此需要分散服务的容器到不同的节点上，该节点拥有此服务下容器越多，得分越低。

score = Σ weighti \* scorei 选择最大得分的机器删除这个pod，同时修改临时缓存，进行下个一个pod的处理。

**模拟退火调度器**

主要逻辑在AnnealingScheduler.annealing()，伪代码如下：

/\*

\* J(y)：在状态y时的评价函数值

\* Y(i)：表示当前状态

\* Y(i+1)：表示新的状态

\* r： 用于控制降温的快慢

\* T： 系统的温度，系统初始应该要处于一个高温的状态

\* T\_min ：温度的下限，若温度T达到T\_min，则停止搜索

\*/

while( T > T\_min )

{

　　dE = J( Y(i+1) ) - J( Y(i) ) ;

　　if ( dE >=0 ) //表达移动后得到更优解，则总是接受移动

Y(i+1) = Y(i) ; //接受从Y(i)到Y(i+1)的移动

　　else {

// 函数exp( dE/T )的取值范围是(0,1) ，dE/T越大，则exp( dE/T )也

if ( exp( dE/T ) > random( 0 , 1 ) )

Y(i+1) = Y(i) ; //接受从Y(i)到Y(i+1)的移动

　　}

　　T = r \* T ; //降温退火 ，0<r<1 。r越大，降温越慢；r越小，降温越快

　　/\*

　　\* 若r过大，则搜索到全局最优解的可能会较高，但搜索的过程也就较长。若r过小，则搜索的过程会很快，但最终可能会达到一个局部最优值

　　\*/

　　i ++ ;

}

具体实现方法：

1. 生成初始解：生成一个可行的解作为当前解输入迭代过程，并定义一个大数作为初始温度。对于一个容器调度请求的输入序列，给其中每一个容器，在满足该容器所有预选条件的节点中，随机分配一个节点，每个可行的分配方案称为一个可行解。每个可行解表达为 , 表示一个容器操作：从第1个到第i-1个容器操作完成后，容器第i个容器操作选择的节点是, 表示容器调度请求总数。初始温度表示为。
2. 迭代过程：首先，由一个产生函数从当前解产生一个位于解空间的新解，产生新解的方法是，对于当前解，随机选取一个容器调度操作，假定尚未执行, 和都已执行，计算此时的集群状态缓存并保存为，k表示迭代次数，在这个集群状态下对第j个容器调度请求进行节点的预选阶段，在预选通过的节点中随机选取一个，生成, 替换当前解的，得到新解。其次，计算与新解所对应的目标函数差，计算的方法是，读取之前提到的集群状态缓存，在下只计算新旧两解中与的优选的分差作为最后的目标函数差，忽略其他容器操作（为了加速计算）。如果，接受新解；否则，以概率接受新解。然后，温度自乘以退火系数，循环进行下一轮（k+1轮）迭代。
3. 停止准则：每次迭代完成后检查温度是否到达最低温度，或者解的实际得分满足，已经足够高，此时停止迭代。

**遗传算法调度器**

主要逻辑在GeneticScheduler. genetic()，具体实现方法：

1. 生成初始种群：多次重复权利要求3所述的生成初始解过程，形成多个可行解，构成一个初始种群，表示为, 表示权利要求3所述的可行解，表示种群的大小。
2. 自然选择：利用优选策略，求出种群每个解的优选策略得分，定义为适应度函数值。然后生成新的一代种群，生成的方法是，以概率抽取当前种群中解，解被抽取到的概率为，每次抽取2个，选取其中得分高的那一个加入下一代种群，重复抽取过程，直至下一代种群的大小达到。
3. 染色体交叉：在当前种群中，将解两两配对，每一对解以交叉概率进行染色体交叉，交叉的方法如下：当两个解和中的容器操作相等的比例超过近亲比率，即时，放弃交叉，否则，从1~n中随机选择一个位置k，两个解在这个位置之后的pod分配的节点对调，变成和。
4. 变异：在当前种群中，每一个解以变异概率进行变异，变异的方法与权利要求2中模拟退火生成新解的方法相同。
5. 终止：重复上述(2),(3),(4)步骤，直至满足设定的循环次数为止，最后选择种群里得分最高的最为最终选择调度方案。