

容器云平台调度器方案简介

李俊茂

本次分享主要是对一些常见的云平台调度器及其设计思想做介绍，主要为后续研究与开发打下基础，请大家积极讨论指正

目录

1. 调度器设计目标
2. k8s原生调度器kube-scheduler
3. 华为火山调度器Volcano
4. 微软openPAI调度器HivedScheduler
5. Fluid调度器
6. KubeFed多集群调度器方案介绍
7. 腾讯tensile-kube多集群调度器方案介绍

调度器设计目标

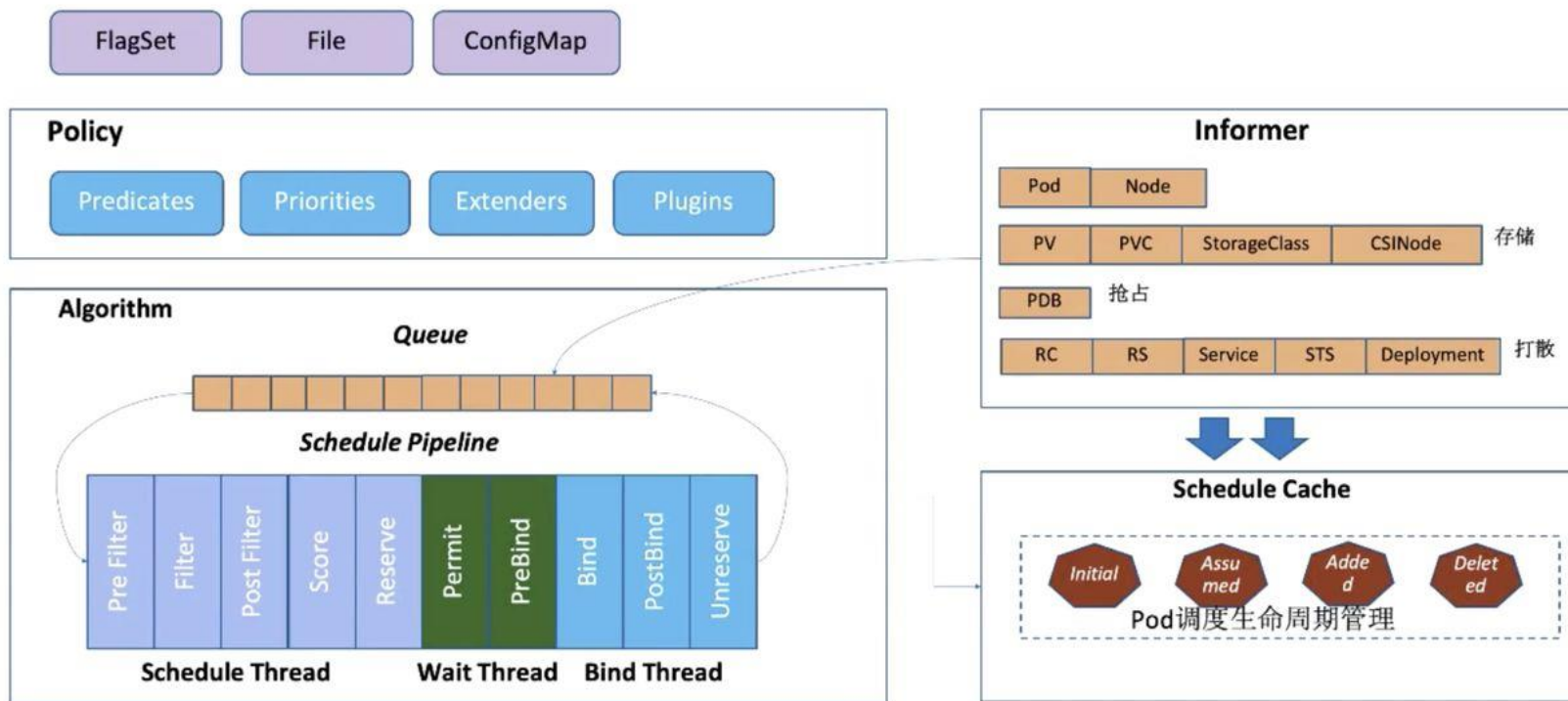
调度器设计目标

- 公平性：在调度任务时需要公平的进行决策，每个节点都有被分配资源的机会，调度器需要对不同节点的使用作出平衡决策。
- 资源高效利用：最大化群集所有资源的利用率，使有限的CPU、内存等资源服务尽可能更多的任务。
- 效率问题：能快速的完成对大批量Pod的调度工作，在集群规模扩增的情况下，依然保证调度过程的性能。
- 灵活性：在实际运作中，用户往往希望Pod的调度策略是可控的，从而处理大量复杂的实际问题。因此平台要允许多个调度器并行工作，同时支持自定义调度器

为达到上述目标，云平台调度器通过结合Node资源、负载情况、数据位置等各种因素进行调度判断，确保在满足场景需求的同时将任务分配到最优节点。

k8s原生调度器kube-scheduler

调度器框架



Policy: 指定使用哪些Predicates(过滤器)、Priorities(打分器)、Extenders（扩展调度器）、Plugins(插件)

Informer: 从kube-apiserver获取Pods、Nodes、PV, PVC等数据，并存为Cache

调度流水线: 通过Informer将pod信息插入Queue

循环从Queue取出执行: Scheduler Thread, Wait Thread, Bind Thread

调度器框架

SchedulingQueue阶段: 包含**activeQ**、**backoffQ**、**unschedulableQ**

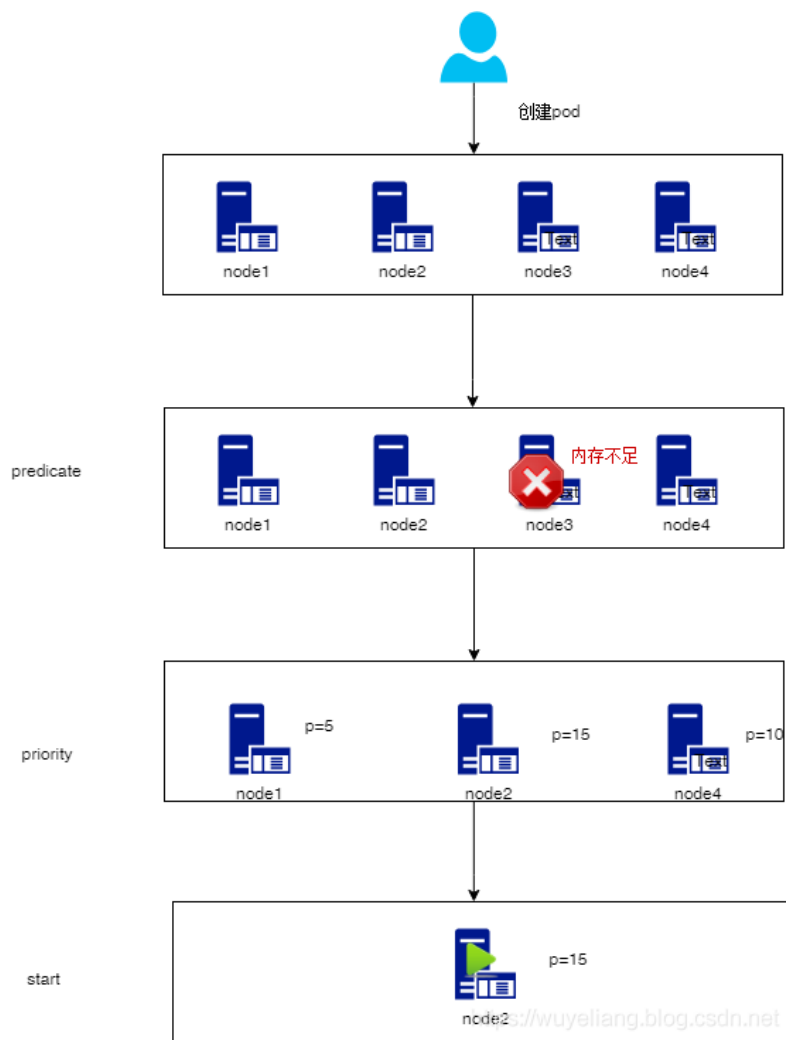
Scheduler Thread 阶段: **Pipeline**拿到一个等待调度的**Pod**，会从**NodeCache**里面拿到相关的**Node**执行 **Filter**逻辑匹配

Score阶段: 依据**Policy**配置的算分插件进行排序，分数最高的节点作为 **SelectHost**

Reserver阶段: 预占的过程，修改**Pod**在**PodCache**的状态为 **Assumed** 的状态(处于内存态)

Pod状态机生命周期: **Initial->Assumed->Added->Deleted**， 当通过 **Informer watch**到**Pod**已经确定分配到节点时，才会把**Pod**的状态变成 **Added**；选中完节点在**Bind**的时候，有可能会 **Bind**失败，在 **Bind** 失败的时候会做回退，就把**Assumed**退回**Initial**，从**NodeCache**中把**Pod**信息清除

调度器框架



预选 (Predicates): 输入是所有节点，输出是满足预选条件的节点。kube-scheduler根据预选策略过滤掉不满足策略的Nodes

优选 (Priorities): 输入是预选阶段筛选出的节点，优选会根据优先策略为通过预选的Nodes进行打分排名，选择得分最高的Node

调度算法实现

Predicates（预选阶段）

1. 基于存储卷数量的判断

MaxEBSVolumeCount: 确保已挂载的EBS存储卷数量不超过设置的最大值（默认39）

MaxGCEPDVolumeCount: 确保已挂载的GCE存储卷数量不超过预设的最大值（默认16）

MaxAzureDiskVolumeCount: 确保已挂载的Azure存储卷不超过设置的最大值（默认16）

2. 基于资源压力状态的判断

CheckNodeMemoryPressure: 节点是否进入到内存压力状态，如果是则只允许调度内存为0标记的Pod

CheckNodeDiskPressure: 节点是否进入到磁盘压力状态，如果是，则不能调度新的Pod

3. 基于卷冲突的判断

NoDiskConflict: 检查是否存在Volume冲突，仅限于 GCE PD、AWS EBS、Ceph RBD以及ISCSI

NoVolumeNodeConflict: 检查节点是否满足Pod所引用的Volume的条件

NoVolumeZoneConflict: 检查volume zone是否冲突

4. 基于约束关系的判断

MatchNodeSelector: 检查节点label是否匹配Pod指定的nodeSelector

MatchInterPodAffinity: 根据Pod之间的亲和性做判断

PodToleratesNodeTaints: 判断Pod不允许被调度到哪些节点，涉及到Taints（污点）和Toleration（容忍）

5. 基于适合性的判断

PodFitsResources: 检查节点是否有足够资源（如CPU、内存、GPU等）满足Pod的运行需求

PodFitsHostPorts: 检查Pod容器所需的HostPort是否已被节点上其它容器或服务占用

PodFitsHost: 检查Pod指定的NodeName是否匹配当前节点

调度算法实现

Priorities（优选阶段）

用一组优先级函数处理每个通过预选的节点，每个函数返回0-10的分数，各个函数有不同权重，最终得分是所有优先级函数的加权和

1. Node资源水位

LeastRequestedPriority: 优先打散，把Pod分到资源空闲率最高的节点上，而非空闲资源最大的节点

MaxRequestedPriority: 优先堆叠，把 Pod 分配到资源使用率最高的节点上

BalanceResourceAllocation: 碎片率，指Node上的多种资源之间的资源使用率的差值

RequestToCapacityRatioPriority: 指定比率，可以在Scheduler启动的时候，为每一个资源使用率设置得分，从而实现控制集群上node资源分配分布曲线

2. Pod打散

ServiceSpreadingPriority/SelectorSpreadPriority: 用于实现Pod所属的Controller下所有的Pod在Node上打散的要求

EvenPodsSpreadPriority: 用来指定一组符合条件的Pod在某个拓扑结构上的打散需求(结合TopologySpreadConstraints特性)

调度算法实现

Priorities（优选阶段）

3.Node 亲和&反亲和

NodeAffinityPriority: 满足Pod和Node的亲和&反亲和

ServiceAntiAffinity: Service下Pod的分布要按照Node的某个label进行均衡

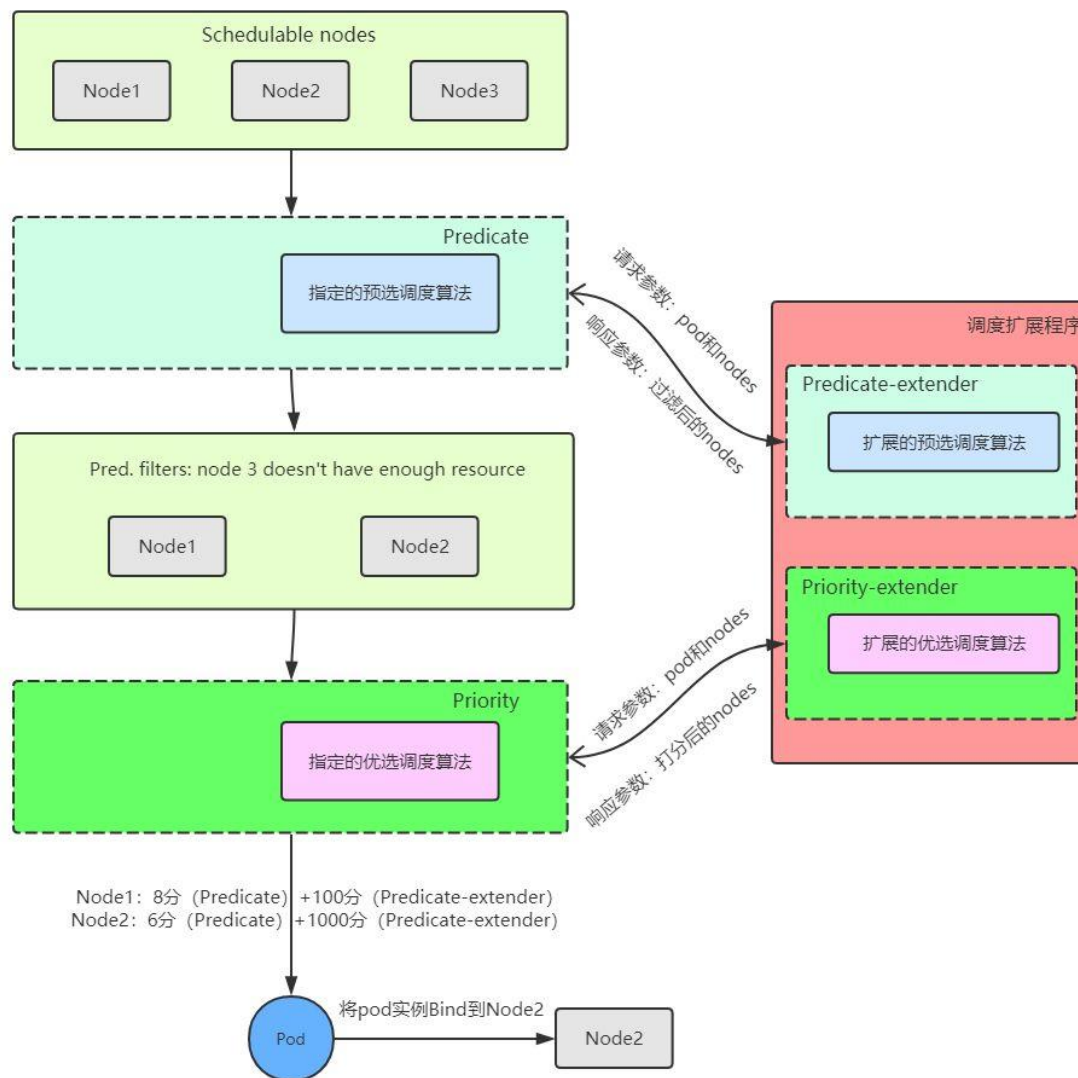
ImageLocalityPriority: 如果节点里面存在镜像的话，优先把Pod调度到这个节点上

4. Pod亲和&反亲和

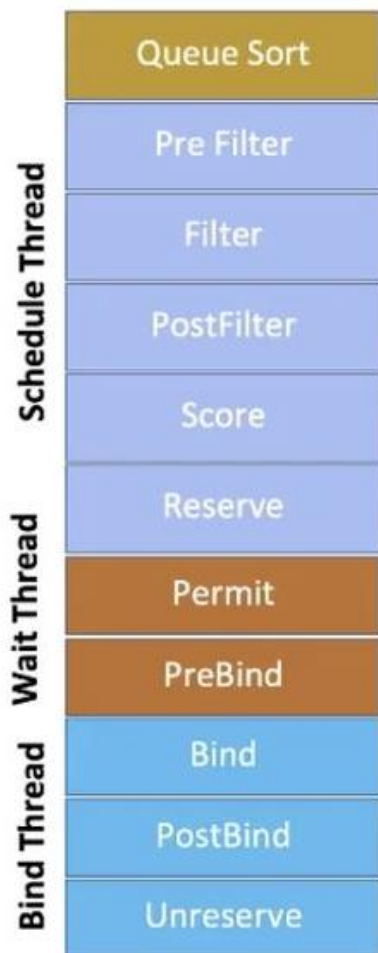
InterPodAffinityPriority: Pod之间亲和&反亲和，比如互相提供服务的Pod在相同节点更快；CPU密集型可能互相干扰的Pod避免在同个节点

NodePreferAvoidPodsPriority: 用于实现某些controller尽量不分配到某些节点上的能力；通过在node上加annotation声明哪些controller不要分配到 Node上

扩展调度器



扩展调度器



- QueueSort : 支持自定义的Pod的排序
- PreFilter: 对Pod的请求做预处理
- Filter: 自定义filter逻辑
- PostFilter: 可以用于logs/metrics, 或者对Score之前做数据预处理
- Score: 自定义的Score逻辑
- Reserve: 有状态的plugin可以对资源做内存记账
- Permit: wait, deny, approve, 可以作为gang的插入点
- PreBind: 在真正bind node之前, 执行一些操作, 例如: 云盘挂载盘到Node上
- Bind: 一个Pod只会被一个BindPlugin处理
- PostBind: bind成功之后执行的逻辑
- Unreserve: 在permit到Bind这几个阶段只要报错就回退

社区插件项目 Scheduler-Plugins

GangScheduling

ElasticQuota

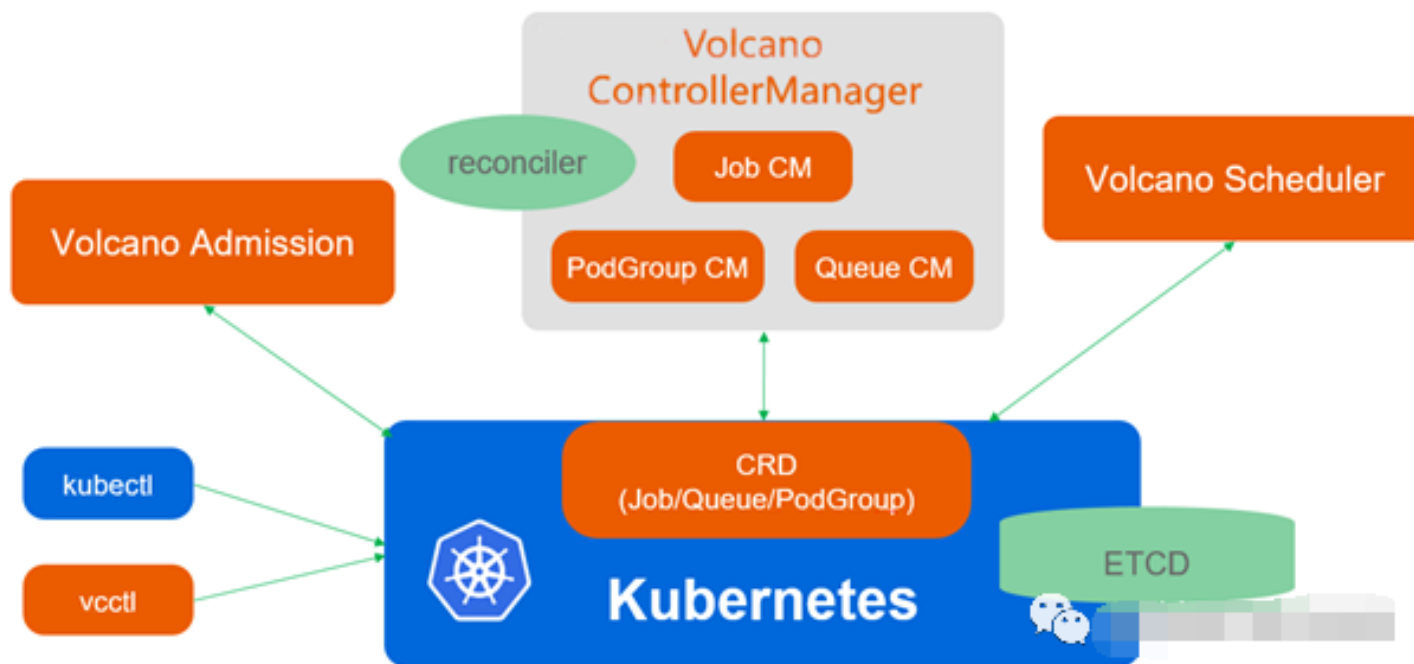
CapacityScheduling

LoadAwareScheduling

.....

华为开源Volcano调度器

架构体系



- 增加队列Queue: 实现资源软隔离
- 增加PodGroup: 任务组, 与queue绑定, 实现GangScheduling
- 增加Job: 作业定义, 生命周期策略、任务模板等

任务模板

```
apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
  name: tf-job
spec:
  tasks:
    - name: "ps"
      replicas: 2
      template:
        spec:
          containers:
            - name: ps
              image: ps-img
    - name: "worker"
      replicas: 5
      template:
        spec:
          containers:
            - name: worker
              image: worker-img
```

任务模板

```
apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
  name: tf_distributed
spec:
  minAvailable: 6
  schedulerName: volcano
  plugins:
    env: []
    svc: []
  policies:
    - event: PodEvicted
      action: RestartJob
  tasks:
    - replicas: 4
      name: worker
      policies:
        - event: TaskCompleted
          action: CompleteJob
      template:
        spec:
          containers:
            - command:
                - sh
                - -c
                - |
                  PS_HOST=`cat /etc/volcano/ps.host | sed 's/$/&:2222/g' | tr "\n" ",";
                  WORKER_HOST=`cat /etc/volcano/worker.host | sed 's/$/&:2222/g' | tr "\n" ",";
                  python tf_distributed.py --job_name=worker --task_index=${VK_TASK_INDEX}
                    --ps_hosts=${PS_HOST} --worker_hosts=${WORKER_HOST}
              image: volcanosh/example-tf:0.8
              name: tensorflow
            restartPolicy: OnFailure
    - replicas: 2
      name: ps
      template:
```

错误处理

Worker结束作业结束

作业插件机制

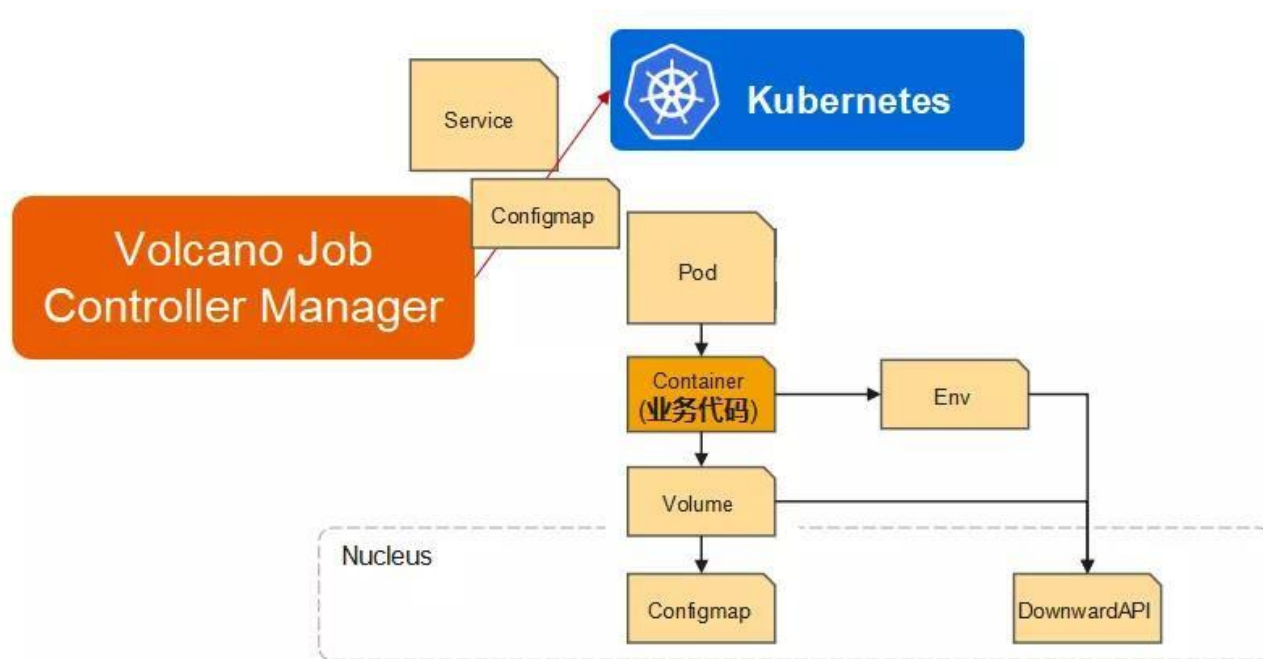
定义接口**PluginInterface**

当一个作业在增删改查，或者**Job**在创建的时候，用户可以实现这个函数，去做定制化的需求

已提供的插件

- svc**: 不同类型的任务之间互访
- env**: 任务索引等环境变量
- ssh**: ssh秘钥对创建及挂载，免密登录

作业插件机制

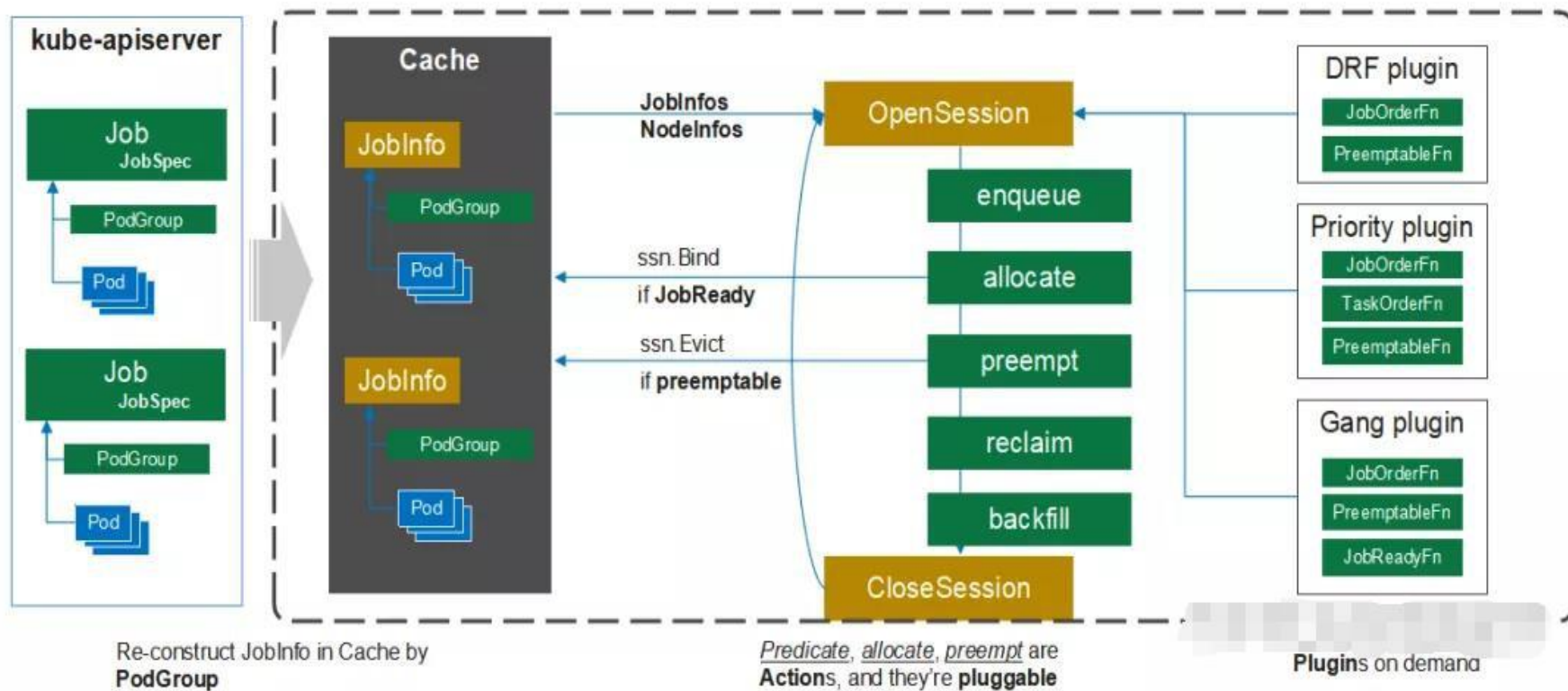


svc plugin

Mount Path:

/etc/volcano/VC_{taskName}_HOSTS

scheduler框架



scheduler框架

```
// Action is the interface of scheduler action.
type Action interface {
    // The unique name of Action.
    Name() string

    // Initialize initializes the allocator plugins.
    Initialize()

    // Execute allocates the cluster's resources into each queue.
    Execute(ssn *Session)

    // UnInitialize un-initializes the allocator plugins.
    UnInitialize()
}

// Plugin is the interface of scheduler plugin
type Plugin interface {
    // The unique name of Plugin.
    Name() string

    OnSessionOpen(ssn *Session)
    OnSessionClose(ssn *Session)
}
```

Action和Plugin都通过实现接口自定义实现，Action使用Plugin提供的功能

scheduler主要特性

Gang Scheduling: 组调度;

Fair Share: 队列间公平调度;

Preempt: 抢占;

Backfill: 两队列各有资源不满足的作业时, 将一队列资源临时分配给另一队列;

Reclaim: 创建新队列时, 对旧队列所占资源重新分配

Topology Aware Scheduling: 拓扑调度

Service Level Agreement: 可配置最大等待时间, 超时则优先调度

Time Division Multiplexing: 配置复用节点 (同时属于多个集群的节点) 的分用时段

微软openPAI调度器HivedScheduler

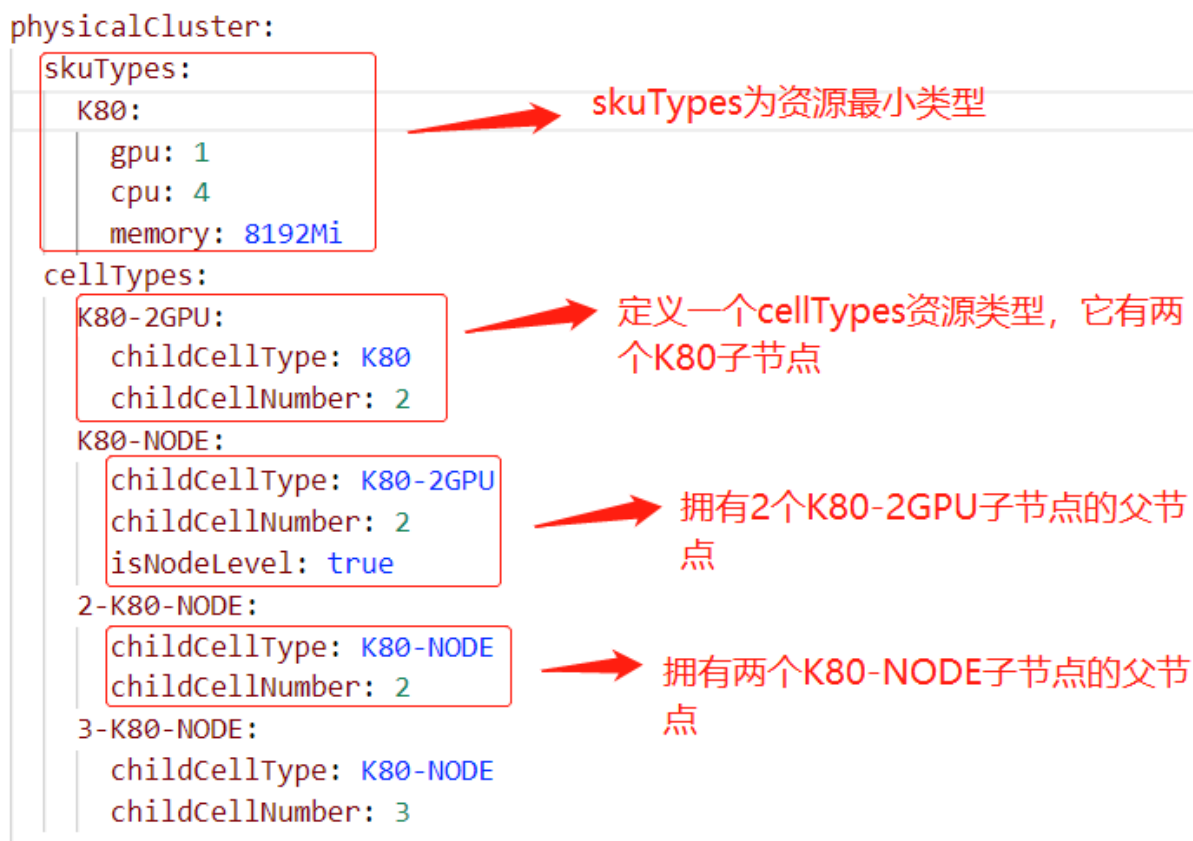
Hived调度器特性

Hived Scheduler为基于kube-scheduler的extender

1. 多租户：Virtual Cluster (VC)
2. 细粒度VC资源保证：数量、类型、固定资源、拓扑等
3. 灵活的VC内调度：拓扑感知、灵活的硬件类型、固定资源、调度策略定制化等
4. 优化的资源碎片管理和饥饿避免
5. 优先级、VC内与VC间任务抢占
6. Gang-scheduling
7. 错误容忍、硬件错误感知

拓扑定义

1. 定义physicalCluster资源



拓扑定义

2. 定义physicalCells资源，为physicalCluster的实例化

```
physicalCells:
- cellType: 3-K80-NODE
  cellChildren:
  - cellAddress: 10.151.41.19
  - cellAddress: 10.151.41.25
    pinnedCellId: VC2-K80
  - cellAddress: 10.151.41.26
- cellType: 2-K80-NODE
  cellChildren:
  - cellAddress: 10.151.41.23
  - cellAddress: 10.151.41.24
```

拓扑定义

3. 定义virtualClusters资源

```
virtualClusters:
  vc1:
    virtualCells:
      - cellType: 3-K80-NODE.K80-NODE
      - cellNumber: 1
  vc2:
    virtualCells:
      - cellType: 3-K80-NODE.K80-NODE
      - cellNumber: 1
    pinnedCells:
      - pinnedCellId: VC2-K80
  default:
    virtualCells:
      - cellType: 2-K80-NODE
      - cellNumber: 1
```

cellType为k80-NODE

cellType为2-K80-NODE

作业模板

```
protocolVersion: 2
name: itc-buddy
type: job
prerequisites:
  - protocolVersion: 2
    name: keras_tensorflow_example
    type: dockerimage
    uri: openpai/pai.example.keras.tensorflow
taskRoles:
  train:
    instances: 2
    completion:
      minFailedInstances: 1
      minSucceededInstances: 6
    dockerImage: keras_tensorflow_example
    resourcePerInstance:
      cpu: 4
      memoryMB: 8192
      gpu: 1
    commands:
      - nvidia-smi -L
      - printenv
      - sleep 10m
defaults:
  virtualCluster: vc1
extras:
  gangAllocation: true
  hivedScheduler:
    jobPriorityClass: prod
    taskRoles:
      train:
        skuType: K80
```

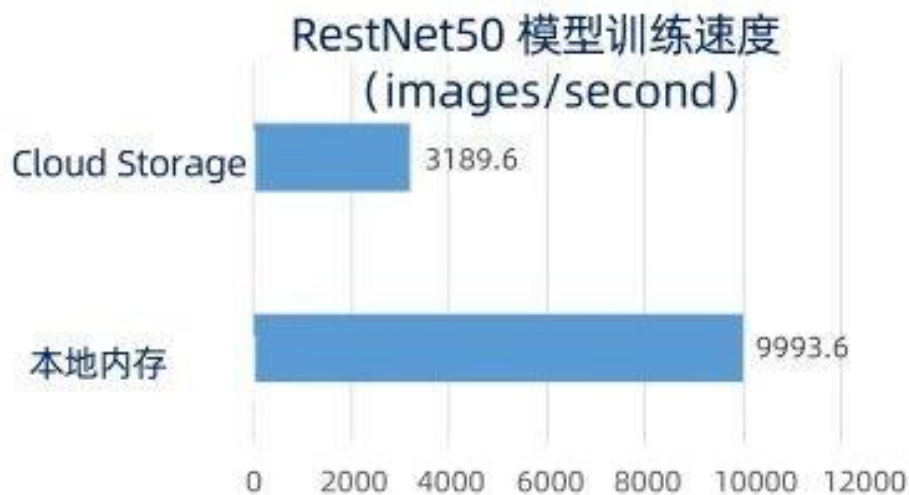
2个副本

调度到vc1集群

由于两个副本在同一个AffinityGroup中，会被调度到VC1中两个相邻节点。

Fluid调度器

当前面临的问题



1. 云平台计算存储分离架构导致数据访问延时高
2. 混合云场景下跨存储系统的联合分析困难

Fluid特性

1. 提供云平台数据集抽象的原生支持:

数据密集型应用所需基础支撑能力功能化，实现数据高效访问并降低多维成本

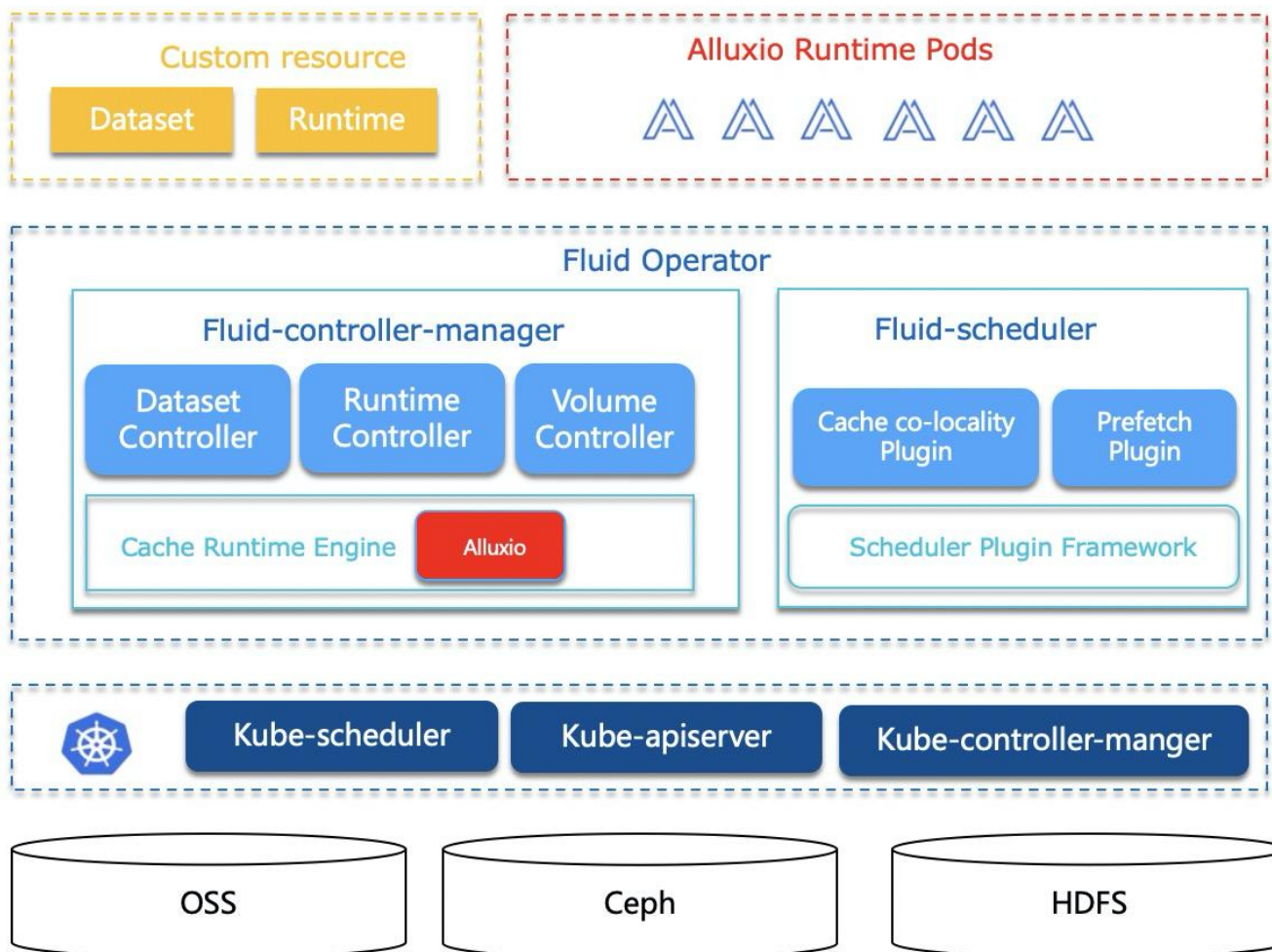
2. 混合云场景下跨存储系统的联合分析困难:

通过数据集缓存引擎与 Kubernetes 容器调度和扩缩容能力的相互配合，实现数据集可迁移性

3. 面向云上数据本地化的应用调度:

Kubernetes 调度器通过与缓存引擎交互获得节点的数据缓存信息，将使用该数据的应用以透明的方式调度到包含数据缓存的节点，最大化缓存本地性的优势。

Fluid架构



Fluid架构

Cache Runtime Engine: 是真正完成缓存数据具体工作的地方

Dataset Controller: 负责整个数据集的生命周期管理，包括数据集的创建，以及要和哪个 Runtime 进行绑定

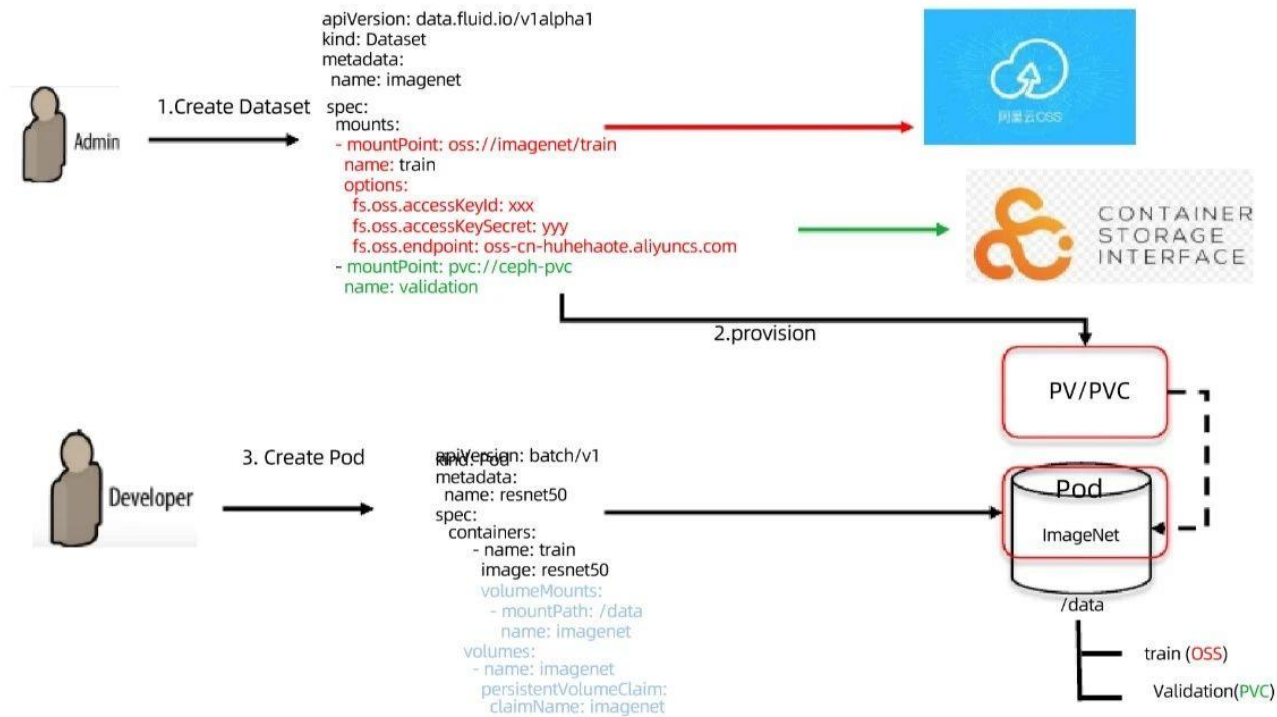
Runtime Controller: 负责数据集如何在云原生平台上被调度与缓存，应该放在哪些节点上，要有多少副本

Volume Controller: 由于 Fluid 是基于 K8s 运行，因此需要和 K8s 进行对接，使用 PVC（数据持久卷）

Cache co-locality Plugin: 结合前面数据编排的信息，把应用调度到最合适的节点上，然后尽量能够让用户去读到缓存节点里面的信息

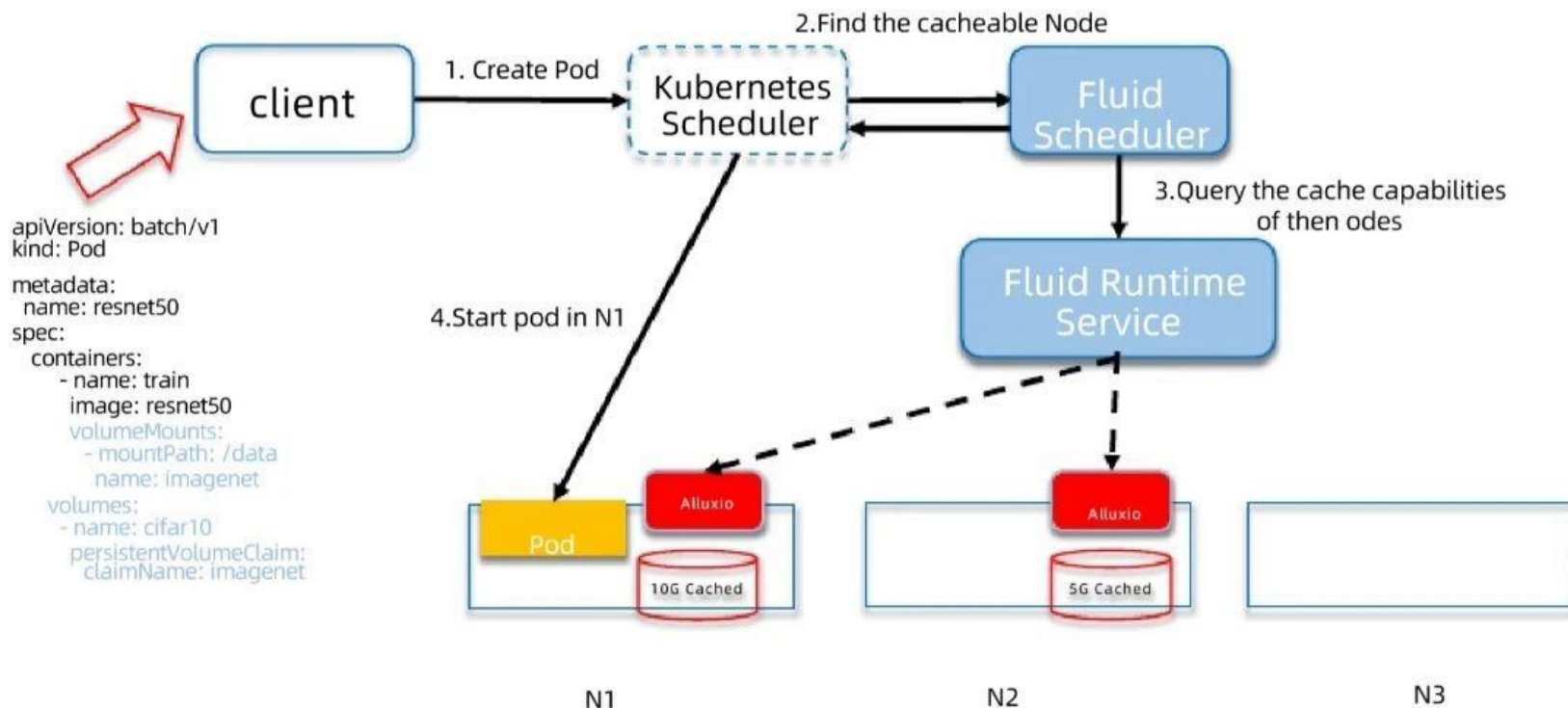
Prefetch Plugin: 当用户集群还没有缓存流入数据情况之下，且知道应用肯定是要读哪一类数据时，尤其在应用调度和编排运行之前，可以做 Prefetch 的调度，将这个数据从最底下的存储卷当中缓存到数据缓存里面，可以手动触发

Fluid用例



1. 用户任务所需数据分别来自阿里云和本地存储 **Ceph**。先创建一个自定义 **K8s** 资源对象 **Dataset**，对应的 **mountPoint** 分别是阿里云和 **Ceph**。
2. 创建过程中 **Fluid** 会创建一个**Dataset**，并同时创建一个**PVC**
3. 当用户需要用这个数据时创建一个 **Pod**，以 **PVC** 挂载的方式，将 **Dataset** 关联到运行中的 **Pod** 中对数据进行访问

Fluid用例



1. 调度Pod时，K8s 调度器和 Fluid 调度器交互时会看见三个节点，其中有两个有 Alluxio 缓存节点。K8s 调度包括两个阶段，一个是过滤阶段，另一个是优选阶段。在过滤阶段就会将第三个节点直接过滤掉，而在优选阶段可以利用一些优选的策略来选择更合适的节点，例如缓存空间量大小。

数据缓存调度

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
  name: hbase
spec:
  mounts:
    - mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
      name: hbase
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: hbase-cache
              operator: In
              values:
                - "true"
```

1. Dataset数据集CRD通过NodeSelector，被调度到拥有该Label的节点上
2. Dataset与节点上AlluxioRuntime服务绑定，并创建PVC和PV
3. Pod调度到节点上使用PVC读取数据集

数据预加载

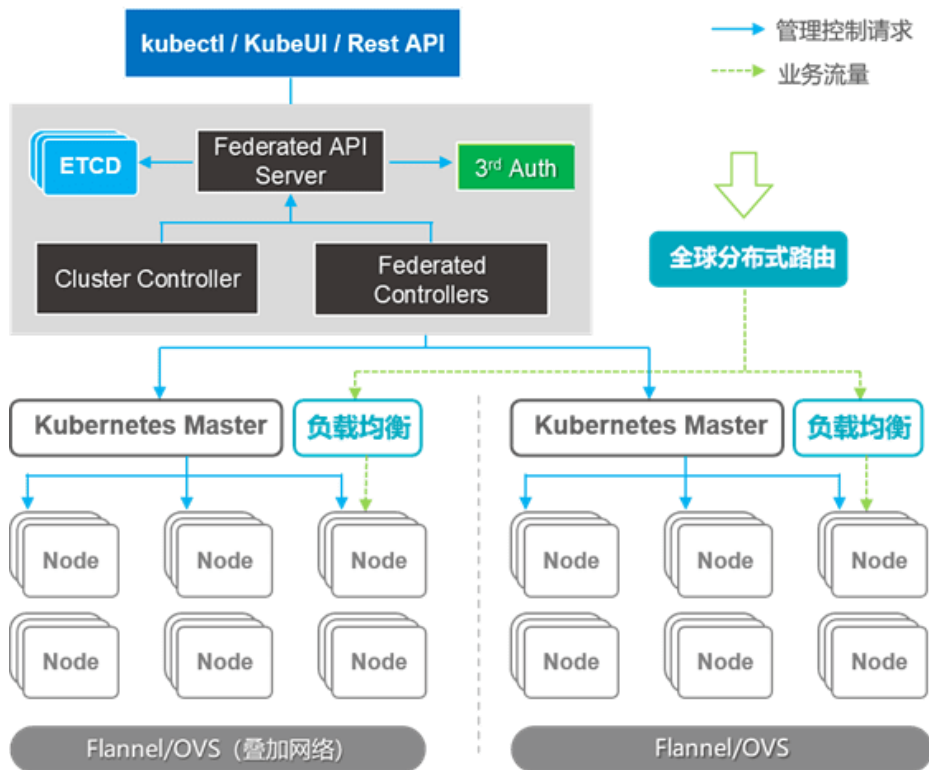
```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
  name: spark-dataload
spec:
  dataset:
    name: spark
    namespace: default
  target:
    - path: /spark/spark-2.4.7
    - path: /spark/spark-3.0.1/pyspark-3.0.1.tar.gz
```

1.通过创建DataLoad CRD，可提前将数据集加载进缓存引擎，加快首次读取速度

KubeFed多集群调度器方案介绍

V1架构

V1架构：引入了**Federated API Server**，用于增加集群相关**API**，屏蔽集群差异，统一请求入口，同时提供**Cluster Controller**用于管理多个集群状态、集群级别对象创建；**Service Controller**用来实现跨集群服务发现



优点：

V1架构兼容K8S原生API，从单集群到多集群演进可以变得很自然

缺点：

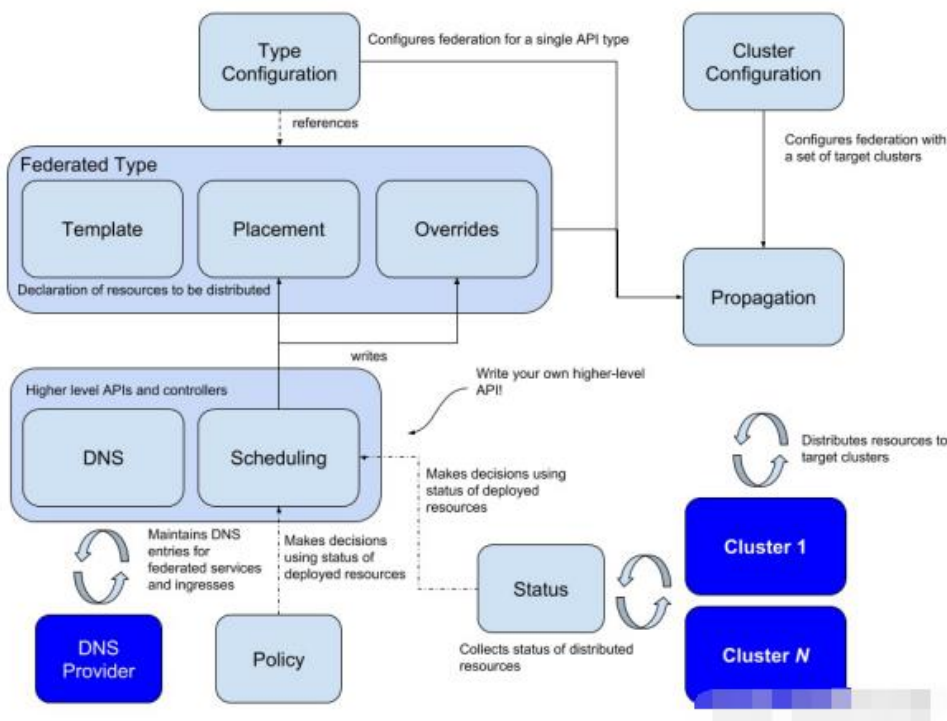
1. 集群信息嵌入原生API的Annotation中，会导致原生API体积膨胀而丑陋；

2. 没有集群生命周期管理特有API，导致其生命周期管理能力无法扩展；

3. 无法提供API对象版本控制，比如Deployment在K8S为GA，但在Federation中可能仍是Beta；

V2架构

V2架构：利用CRD来提供独立的API对象，新的API来封装K8S原生对象，同时也可以方便的对新增API提供版本管理



Type configuration: 定义Kubefed接管的K8S的资源类型

Cluster configuration: 定义Kubefed接管的K8S集群

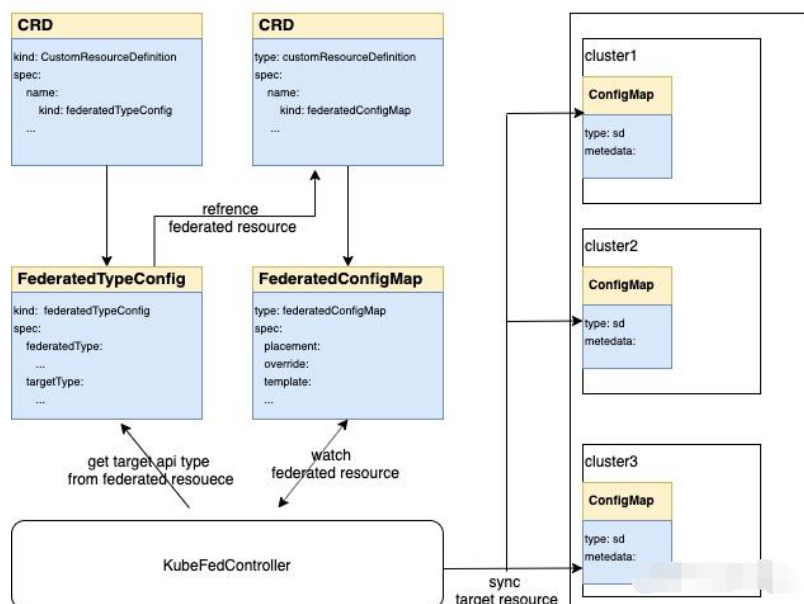
在类型配置中的三个关键的概念，用于控制资源向拖管集群分发策略：

Templates: 定义一个原生的K8S资源类型；

Placement: 定义资源将分发的集群；

Overrides: 针对集群自由修正资源；

V2架构联邦化资源



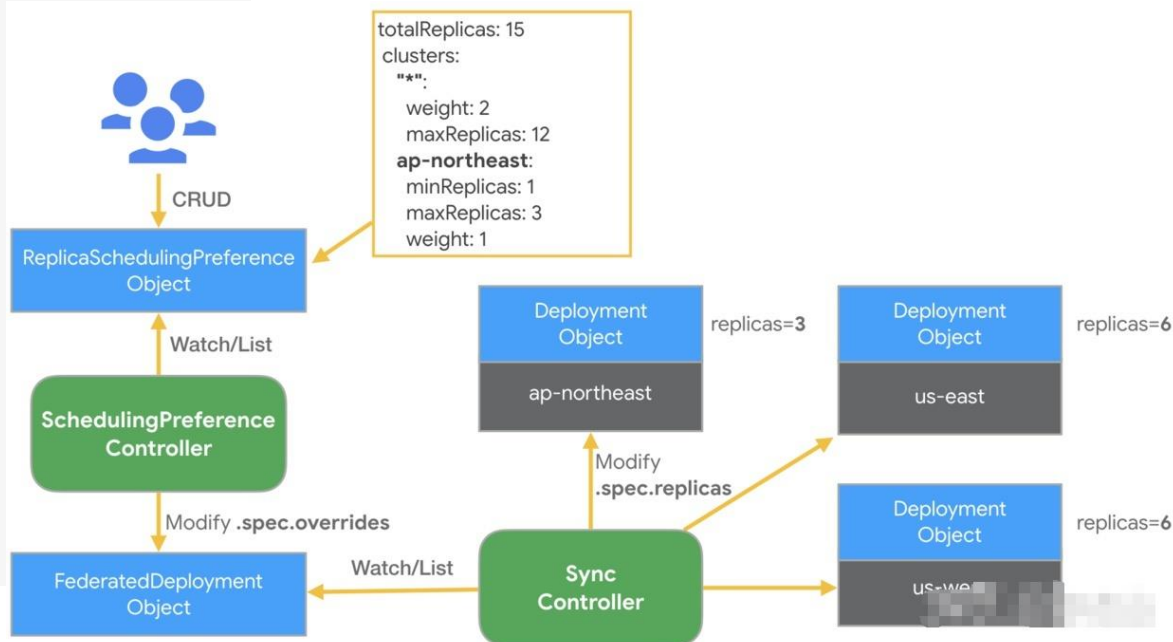
```
apiVersion: core.kubefed.k8s.io/v1beta1
kind: FederatedTypeConfig
metadata:
  name: configmaps
  namespace: kube-federation-system
spec:
  federatedType:
    group: types.kubefed.k8s.io
    kind: FederatedConfigMap
    pluralName: federatedconfigmaps
    scope: Namespaced
    version: v1beta1
  propagation: Enabled
  targetType:
    kind: ConfigMap
    pluralName: configmaps
    scope: Namespaced
    version: v1
```

假如想将 configmap 通过联邦机制在多个集群中创建:

1. 在 Federation Host 集群中创建 FederatedConfigMap CRD 资源
2. 创建 FederatedTypeConfig 资源将 FederatedConfigMap 和 Configmap 建立关联。

V2架构跨集群调度

```
apiVersion: scheduling.kubefed.k8s.io/v1alpha1
kind: ReplicaSchedulingPreference
metadata:
  name: test-deployment
  namespace: test-namespace
spec:
  targetKind: FederatedDeployment
  totalReplicas: 15
  clusters:
    ***:
      weight: 2
      maxReplicas: 12
    cluster3:
      minReplicas: 1
      maxReplicas: 3
      weight: 1
```



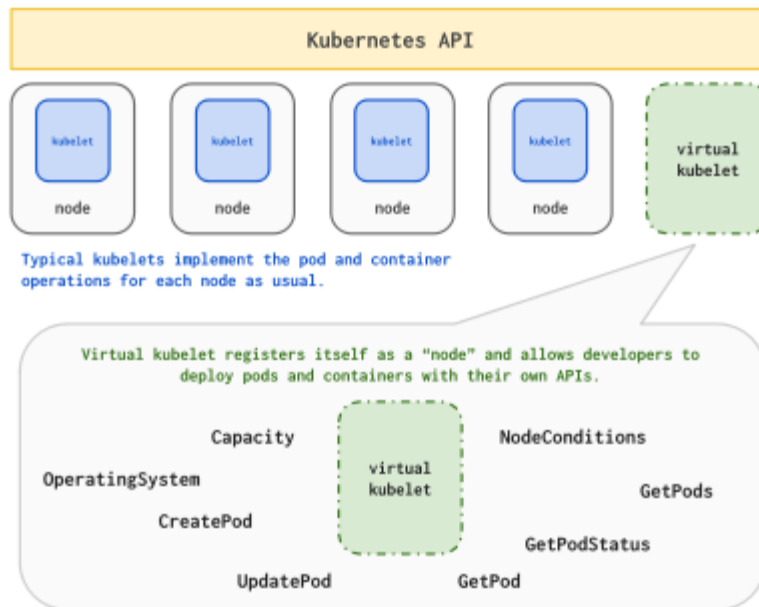
1. 创建ReplicaSchedulingPreference(RSP)
2. 当rsp创建后, kubefed rsp controller 会监听到变化, 并获取rsp内容, 根据rsp定义的策略计算出每个集群的副本数量, 之后修改 federatedDeployment 资源的spec.overrides内容
3. 最后kubefed sync controller 会监听到 federatedDeployment 的变化, 并将新的副本同步到联邦集群中

腾讯tensile-kube多集群调度器方案介绍

业务需求

1. 腾讯游戏容器计算平台线上有数十个Kubernetes集群，这些集群都存在一些碎片资源，无法得到有效利用。常见的场景是：一个作业需要N个资源，但是现有集群A、B、C等所剩资源都不满足N，而集群A、B、C资源总和又能满足N
2. 业务有一种类型的服务需要同时发布到多个集群
3. 已有的kubeFed方案不够轻量化，且对于现有的Kubernetes会有较大的改造成本，同时增加了很多繁杂的CRD

virtual kubelet简介



kubelets通常如何工作

Kubernetes kubelet为每个Kubernetes节点（Node）实现Pod和容器操作。它们作为每个节点上的代理运行，无论该节点是物理服务器还是虚拟机，并在该节点上处理Pod/容器操作。kubelets将名为PodSpec的配置作为输入，并确保PodSpec中指定的容器正在运行且运行正常

Virtual Kubelet的工作原理

从Kubernetes API服务器的角度来看，Virtual Kubelet看起来像普通的kubelet，但其关键区别在于它们在其他地方调度容器，例如在云无服务器API中，而不是在节点上。

tenstile-kube架构

➤ virtual-node

基于virtual-kubelet实现的k8s provider;上层集群创建的Pod将被同步到下层集群中;若Pod依赖于configmaps或secrets, 依赖也将在下层集群中创建

➤ multi-cluster scheduler

基于K8S调度框架实现的调度器; 将监听所有低层级集群的资源容量, 并在调度Pod时调用过滤器; 如果可用节点数量大于等于1, pods将会被调度; 这将增加资源消耗, 所以我们增加了一个descheduler

➤ descheduler

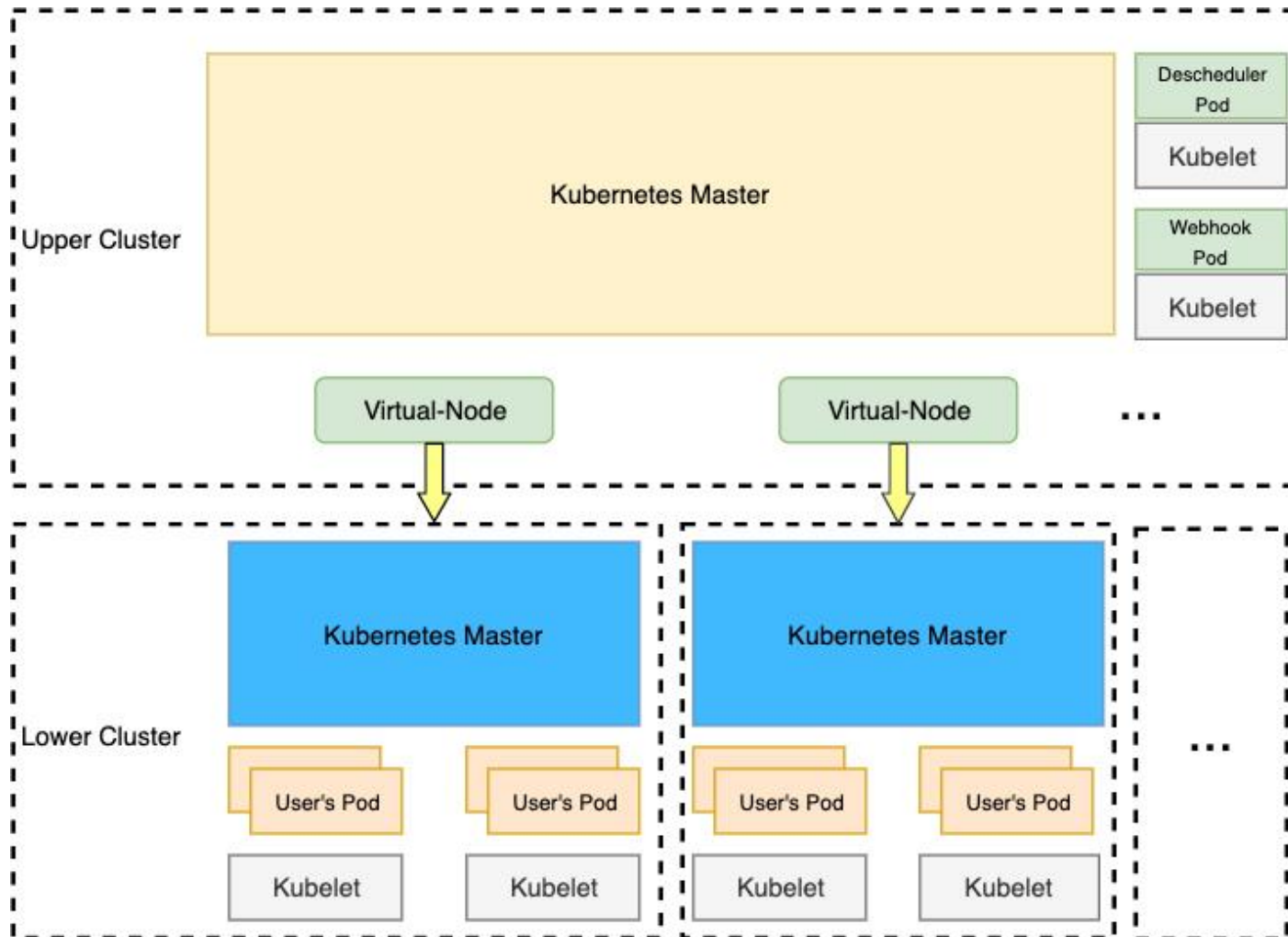
用于避免资源碎片带来的影响, 这个组件基于社区的descheduler进行了二次开发, 使之更适用于这个场景;

multi-cluster scheduler 和 descheduler 在上层集群中可以二选一或者同时使用: 当下层集群总节点数大于10000时, descheduler 消耗资源更少; 反之选Multi-scheduler

➤ webhook

对Pod中可能对上层集群调度产生干扰的字段, 如nodeSelector, nodeAffinity进行转换, 将其写入annotation中, virtual-node在创建Pod时再将其恢复, 避免影响用户期望的调度结果。

tensile-kube架构



Thank you.