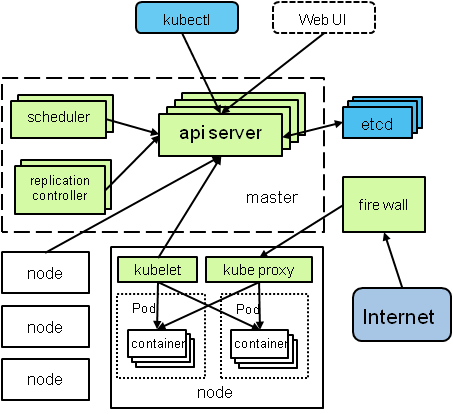
# 概述

基于K8S V1.16

## 目录结构

|  |
| --- |
| api: 输出接口文档用，基本是json源码  build：构建脚本  cmd：所有的二进制可执行文件入口代码，也就是各种命令的接口代码。  pkg：项目代码主目录，cmd只是接口，这里是具体实现。cmd类似业务代码，pkg类似核心  plugin：插件  test：测试相关的工具  third\_party：第三方工具  docs：文档  example：使用例子  Godeps：项目依赖的Go的第三方包，比如docker客户端sdk，rest等  hack：工具箱，各种编译，构建，校验的脚本 |

## 调用结构



管理围绕着Api Server，runtime围绕着kubeproxy。

K8S Api Server主要有以下三类通信交互的场景：

**1）kubelet与API Server交互**

每个Node节点上的kubelet定期就会调用API Server的REST接口报告自身状态，API Server接收这些信息后，将节点状态信息更新到etcd中。kubelet也通过API Server的Watch接口监听Pod信息，如果监听到新的Pod副本被调度绑定到本节点，则执行Pod对应的容器的创建和启动逻辑；如果监听到Pod对象被删除，则删除本节点上的相应的Pod容器；如果监听到修改Pod信息，则kubelet监听到变化后，会相应的修改本节点的Pod容器。

**2）kube-controller-manager与API Server交互**

kube-controller-manager中的Node Controller模块通过API Server提供的Watch接口，实时监控Node的信息，并做相应处理。

**3）kube-scheduler与API Server交互**

Scheduler通过API Server的Watch接口监听到新建Pod副本的信息后，它会检索所有符合该Pod要求的Node列表，开始执行Pod调度逻辑。调度成功后将Pod绑定到目标节点上。

为了缓解各模块对API Server的访问压力，各功能模块都采用缓存机制来缓存数据，各功能模块定时从API Server获取指定的资源对象信息（LIST/WATCH方法），然后将信息保存到本地缓存，功能模块在某些情况下不直接访问API Server，而是通过访问缓存数据来间接访问API Server。

# 模块说明

## Scheduler

K8S Scheduler的作用是：

1.将待调度的Pod（API新创建的Pod、Controller Manager为补足副本而创建的Pod等）按照特定的调度算法和调度策略绑定到集群中的某个合适的Node上

2.并将绑定信息写入etcd中。

在整个调度过程中涉及三个对象，分别是：

1.待调度Pod列表

2.可用Node列表

3.调度算法和策略

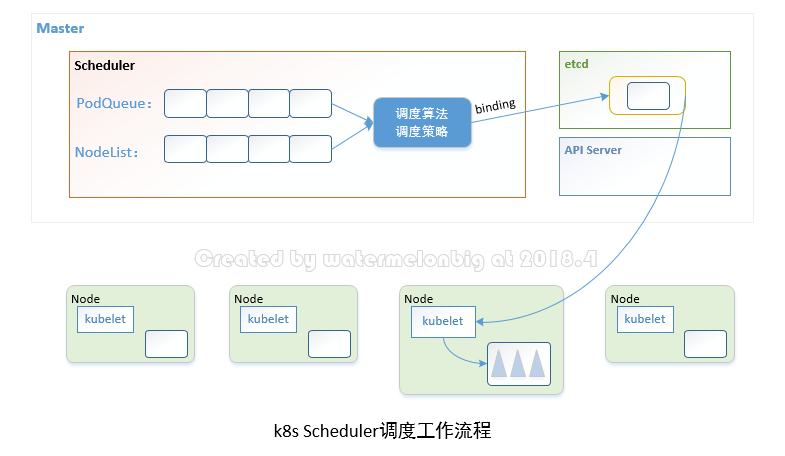
### 执行流程

#### Predicate

预选调度过程，即遍历所有目标Node，筛选出符合要求的候选节点。

kubernetes内置了多种预选策略（xxx Predicates）供用户选择。

#### Priority



### 调度算法

#### Predicate算法

|  |  |  |
| --- | --- | --- |
| **算法** | **说明** | **[默认] 注册不加载** |
| MatchNodeSelector | 检查Node节点的label定义是否满足Pod的NodeSelector属性需求 | 是 |
| PodFitsResources | 检查主机的资源是否满足Pod的需求：  根据实际已经分配（Limit）资源量做调度，而不是使用已实际使用的资源量做调度 | 是 |
| PodFitsHostPorts | 检查Pod内每一个容器所需的HostPort是否已被其它容器占用：  如果有所需的HostPort不满足需求，那么Pod不能调度到这个主机上 | 是 |
| HostName | 检查主机名称是不是Pod指定的NodeName | 是 |
| NoDiskConflict | 检查在此主机上是否存在卷冲突：  如果这个主机已经挂载了卷，其它同样使用这个卷的Pod不能调度到这个主机上  不同的存储后端具体规则不同 |  |
| 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 |  |
| PodToleratesNodeTaints | 确保Pod定义的tolerates能接纳node定义的taints |  |
| CheckNodeMemoryPressure | 检查Pod是否可以调度到已经报告了主机内存压力过大的节点 |  |
| CheckNodeDiskPressure | 检查Pod是否可以调度到已经报告了主机的存储压力过大的节点 |  |
| MaxEBSVolumeCount | 确保已挂载的EBS存储卷不超过设置的最大值，默认39 |  |
| MaxGCEPDVolumeCount | 确保已挂载的GCE存储卷不超过设置的最大值，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 |  |
| MatchInterPodAffinity | 检查Pod和其他Pod是否符合亲和性规则 |  |
| GeneralPredicates | 检查Pod与主机上kubernetes相关组件是否匹配 |  |
| NoVolumeNodeConflict | 检查给定的Node限制前提下，检查如果在此主机上部署Pod是否存在卷冲突 |  |

##### NoDiskConflict

判断备选Pod的gcePersistentDisk或AWSElasticBlockStore和备选的节点中**已存在的Pod是否存在冲突**，过程如下：

1、读取备选Pod的所有Volume的信息（即pod.Spec.Volumes），对每个Volume执行以下步骤进行冲突检测。

1-1、如果该Volume是gcePersistentDisk，则将Volume和备选节点上的所有Pod的每个Volume进行比较：

1-1-1、如果发现相同的gcePersistentDisk，则返回false，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合作为备选Pod；

1-2、如果该Volume是AWSElasticBlockStore，则将Volume和备选节点上的所有Pod的每个Volume进行比较：

1-2-1、如果发现相同的AWSElasticBlockStore，则返回false，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合备选Pod。

2、如果检查完备选Pod的所有Volume均未发现冲突，则返回true，表明不存在磁盘冲突，反馈给调度器该备选节点适合备选Pod。

##### PodFitsResources

判断备选节点的资源是否满足备选Pod的需求，过程如下：

1、计算备选Pod和节点中已存在Pod的所有容器的需求资源（内存和CPU）的总和。

2、获得备选节点的状态信息，其中包含节点的资源信息。

3、如果备选Pod和节点中已存在Pod的所有容器的需求资源（内存和CPU）的总和，超出了备选节点拥有的资源，则返回false，表明备选节点不适合备选Pod，否则返回true，表明备选节点适合备选Pod。

#### Priority算法

|  |  |  |
| --- | --- | --- |
| **算法** | **说明** | **[默认] 注册不加载** |
| EqualPriority | 所有节点同样优先级，无实际效果 | 是 |
| ImageLocalityPriority | 根据主机上是否已具备Pod运行的环境来打分，得分计算：  不存在所需镜像，返回0分，  存在镜像，镜像越大得分越高 | 是 |
| LeastRequestedPriority | 计算Pods需要的CPU和内存在当前节点可用资源的百分比，最小百分比节点为最优 得分计算公式：  cpu((capacity – sum(requested)) \* 10 / capacity)   + memory((capacity – sum(requested)) \* 10 / capacity) / 2 |  |
| BalancedResourceAllocation | 节点上各项资源（CPU、内存）使用率最均衡的为最优 得分计算公式：  10 – abs(totalCpu/cpuNodeCapacity-totalMemory/memoryNodeCapacity)\*10 |  |
| SelectorSpreadPriority | 按Service和Replicaset归属计算Node上分布最少的同类Pod数量 得分计算：数量越少得分越高 |  |
| NodePreferAvoidPodsPriority | 判断alpha.kubernetes.io/preferAvoidPods属性，设置权重为10000，覆盖其他策略 |  |
| NodeAffinityPriority | 节点亲和性选择策略，提供两种选择器支持：  requiredDuringSchedulingIgnoredDuringExecution  保证所选的主机必须满足所有Pod对主机的规则要求  preferresDuringSchedulingIgnoredDuringExecution  调度器会尽量但不保证满足NodeSelector的所有要求 |  |
| TaintTolerationPriority | 类似于Predicates策略中的PodToleratesNodeTaints，优先调度到标记了Taint的节点 |  |
| InterPodAffinityPriority | Pod亲和性选择策略，类似NodeAffinityPriority，提供两种选择器支持：  requiredDuringSchedulingIgnoredDuringExecution  保证所选的主机必须满足所有Pod对主机的规则要求  preferresDuringSchedulingIgnoredDuringExecution  调度器会尽量但不保证满足NodeSelector的所有要求  （两个子策略：podAffinity和podAntiAffinity） |  |
| MostRequestedPriority | 动态伸缩集群环境比较适用，会优先调度pod到使用率最高的主机节点。  在伸缩集群时，就会腾出空闲机器，从而进行停机处理。 | 是 |

## Controller Manager

作为集群内部的管理控制中心，负责集群内的Node、Pod副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）的管理

每个Controller通过API Server提供的接口实时监控整个集群的每个资源对象的当前状态，当发生各种故障导致系统状态发生变化时，会自动尝试着将系统状态从"现有状态"修正到"期望状态"。

### Replication Controller(RC)

核心作用是确保在任何时候集群中一个RC所关联的Pod副本数始终保持预设值。

只有通过RC创建的Pod，才能纳入RC管理。绕过RC的创建，将脱离RC的管理视线。

#### 典型场景

##### 1、重新调度

当发生节点故障或Pod被意外终止运行时，可以重新调度保证集群中仍然运行指定的副本数。

##### 2、弹性伸缩

通过手动或自动扩容代理修复副本控制器的spec.replicas属性，可以实现弹性伸缩。

##### 3、滚动升级

推荐的方式是创建一个新的只有一个副本的RC，若新的RC副本数量加1，则旧的RC的副本数量减1，直到这个旧的RC的副本数量为零，然后删除该旧的RC。

内部机制是，通过改变RC中的Pod模板（主要是镜像版本）来实现系统的滚动升级。

#### 对Pod的管理特性

##### 1、回收

在通常情况下，Pod对象被成功创建后不会消失。

唯一的例外是当Pod处于succeeded或failed状态的时间过长（超时参数由系统设定）时，该Pod会被系统自动回收。

管理该Pod的副本控制器将在其他工作节点上重新创建、运行该Pod副本。

##### 2、Pod模板和Pod实例

RC中的Pod模板就像一个模具，模具制造出来的东西一旦离开模具，它们之间就再没关系了。一旦Pod被创建，无论模板如何变化，也不会影响到已经创建的Pod。

##### 3、Pod脱离

Pod可以通过修改标签来脱离RC的管控，该方法可以用于将Pod从集群中迁移，数据修复等调试。

##### 4、RC删除

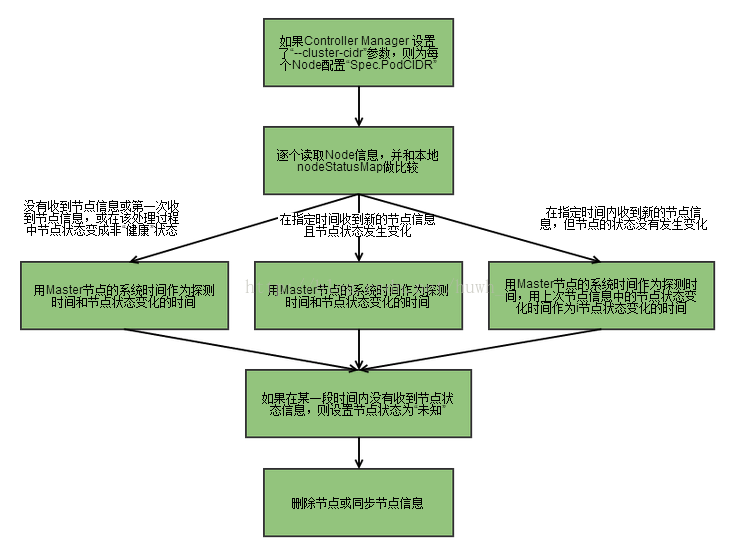
删除一个RC不会影响它所创建的Pod，如果要删除Pod需要将RC的副本数属性设置为0。

**Note**：

只有当Pod的重启策略是Always的时候（RestartPolicy=Always），副本控制器才会管理该Pod的操作（例如创建、销毁、重启等）。

### Node Controller

通过API Server实时获取Node的相关信息，实现管理和监控集群中的各个Node节点的相关控制功能。



### ResourceQuota Controller

确保指定的资源对象在任何时候都不会超量占用系统物理资源。

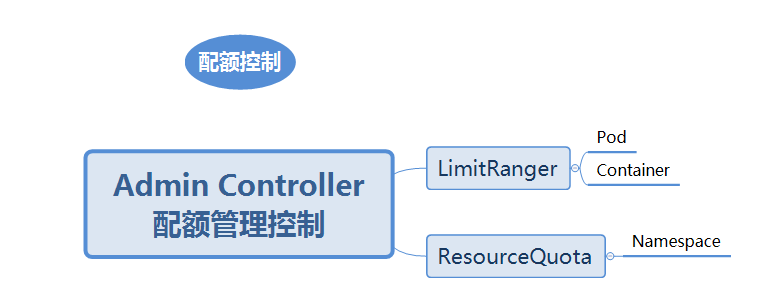
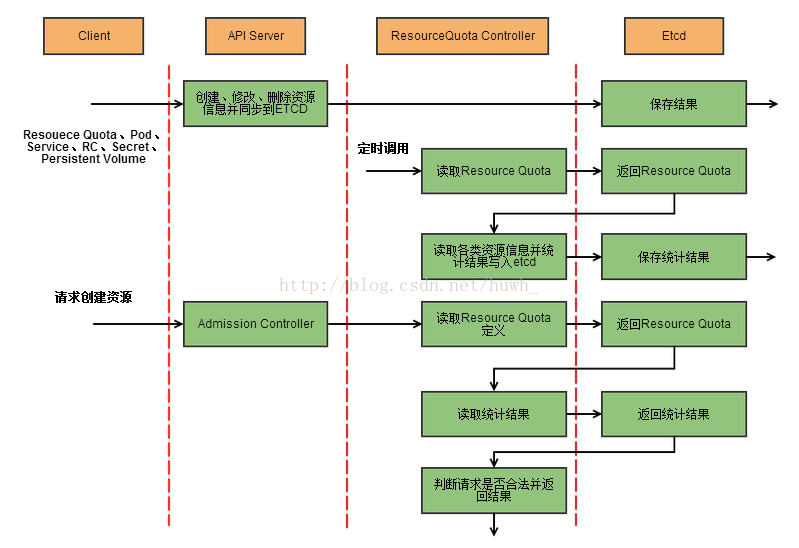


图 资源划分

资源分配的流程如图：



Note：

目前k8s支持如下三个层次的资源配额管理：

1）容器级别：对CPU和Memory进行限制

2）Pod级别：对一个Pod内所有容器的可用资源进行限制

3）Namespace级别：是Namespace级别（多租户）的限制，包括：

Pod数量

Replication Controller数量

Service数量

ResourceQuota数量

Secret数量

可持有的PV（Persistent Volume）数量

### Namespace Controller

创建的Namespace保存在etcd中，Namespace Controller定时通过API Server读取这些Namespace信息。

#### 典型场景

##### 优雅删除

Namespace被记为"优雅删除"（即设置删除期限，DeletionTimestamp）

* **对应动作**

该Namespace状态“Terminating”被保存到etcd中。

Namespace Controller同时删除该Namespace下的ServiceAccount、RC、Pod、Secret、PersistentVolume、ListRange、ResourceQuota和Event等资源对象。

* **同期其他行为-创建**

Admission Controller的NamespaceLifecycle插件来阻止为该Namespace创建新的资源。

* **收尾动作**

在Namespace Controller删除完该Namespace中的所有资源对象后，Namespace Controller对该Namespace执行finalize操作，删除Namespace的spec.finalizers域中的信息。

### ServiceAccount Controller

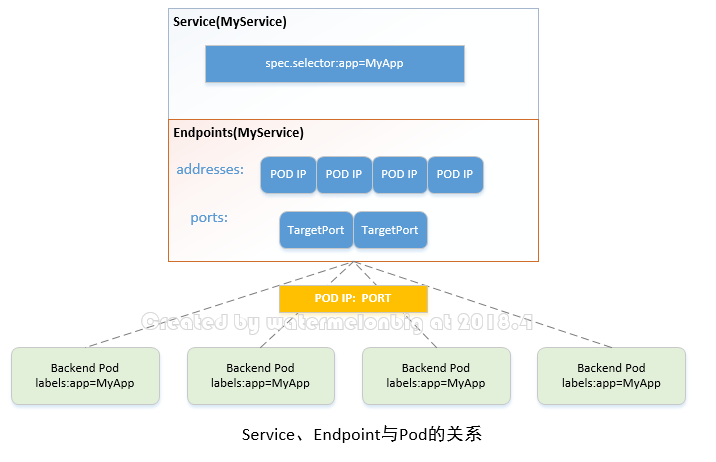
略

### Token Controller

略

### Service Controller

kube-proxy进程获取每个Service的Endpoints，实现Service的负载均衡功能。



### Endpoint Controller

Endpoints表示了一个Service对应的所有Pod副本的访问地址。

Endpoints Controller负责生成和维护所有Endpoints对象的控制器。它负责监听Service和对应的Pod副本的变化。

#### 典型场景-事件监听

##### 1、Service被删除

删除和该Service同名的Endpoints对象

##### 2、Service被创建或修改

根据该Service信息获得相关的Pod列表，然后创建或更新Service对应的Endpoints对象

##### 3、Pod事件

更新它对应的Service的Endpoints对象

## Kubelet

每个Node节点（又称Minion）上都会启动一个kubelet服务进程。

* kubelet用于处理Master节点下发到本Node的任务，管理Pod和其中的容器。
* 每个kubelet会在API Server上注册节点自身信息，定期向Master节点汇报节点资源的使用情况，并通过cAdvisor监控容器和节点资源

### 节点管理

节点通过设置kubelet的启动参数“--register-node”，来决定是否自动主动地向API Server注册自己，默认为true。

在自注册时，kubelet启动时还包含以下参数：

* --api-servers: API Server的位置
* --kubeconfig：kubeconfig文件，用于访问API Server的配置文件
* --cloud-provider：云服务商地址，仅用于公有云环境

在集群运行过程中，如果遇到资源不足的情况，则用户很容易通过添加机器及利用kubelet的自注册模式来实现扩容

其他参数

* --node-status-update-frequency参数，可以配置kubelet向API Server报告节点状态的时间频率，默认为间隔10s

### Pod管理

kubelet通过以下几种方式获取自身Node上所需要运行的Pod清单。

* 文件

kubelet启动参数"--config"指定的配置文件目录下的文件，通过--file-check-frequency设置检查该文件目录的时间间隔，默认为20s。

* HTTP端点（URL）

通过"--manifest-url"参数设置，通过--http-check-frequency设置检查时间间隔，默认为20s。

* API Server

kubelet通过API Server监听etcd目录，同步Pod列表。

以非API Server方式创建的Pod都叫做Static Pod。kubelet将Static Pod状态汇报给API Server，API Server为该Static Pod创建一个Mirror Pod和其相匹配。

* Mirror Pod的状态将真实反映Static Pod的状态。
* 当Static Pod被删除时，与之相对应的Mirror Pod也会被删除。

#### 操作流程

kubelet监听etcd，所以针对Pod的操作将会被kubelet监听到。

如果是创建和修改Pod任务，则做如下处理：

* 为该Pod创建一个数据目录。
* 从API Server读取该Pod清单。
* 为该Pod挂载外部卷（External Volume）
* 下载Pod用到的Secret。
* 检查已运行在节点中的Pod，如果该Pod没有容器或Pause容器没有启动，则先停止Pod里所有容器的进程。如果在Pod中有需要删除的容器，则删除这些容器。
* 用"kubernetes/pause"镜像为每个Pod创建一个容器。该Pause容器用于接管Pod中所有其它容器的网络。
* 为Pod中的每个容器做如下处理：

为容器计算一个hash值，然后用容器的名字去查询对应Docker容器的hash值。这里的hash值可以理解为是依据容器的定义文件所生成的。对比两个hash值，发现有不同则会停止Docker中容器的进程，并停止与之关联的Pause容器的进程。若二者相同，则不做任何处理。

如果容器被终止了，且没有指定restartPolicy重启策略，则不做任何处理。

调用Docker Client下载容器镜像，调用Docker Client运行容器。

### 容器管理

#### 健康检查

通过两类探针来检查：LivenessProbe和ReadinessProbe

##### LivenessProbe探针

功能

* 用于判断容器是否存活（running状态）。
* 如果LivenessProbe探针探测到容器非健康，则kubelet将杀掉该容器，并根据容器的重启策略做相应处理。
* 如果容器不包含LivenessProbe探针，则kubelet认为该探针的返回值永远为“success”。

主要参数

* initialDelaySeconds：启动容器后首次进行健康检查的等待时间，单位为秒。
* timeoutSeconds：健康检查发送请求后等待响应的时间，如果超时响应kubelet则认为容器非健康，重启该容器，单位为秒

实现方式

|  |  |
| --- | --- |
|  | **机制** |
| ExecAction | 在一个容器内部执行一个命令  如果该命令状态返回值为0，则表明容器健康。 |
| TCPSocketAction | 通过容器IP地址和端口号执行TCP检查  如果能够建立TCP连接，则表明容器健康。 |
| HTTPGetAction | 通过容器的IP地址、端口号及路径调用HTTP Get方法  如果响应的状态码大于等于200且小于等于400，则认为容器健康。 |

##### ReadinessProbe探针

用于判断容器是否启动完成（read状态），可以接受请求。

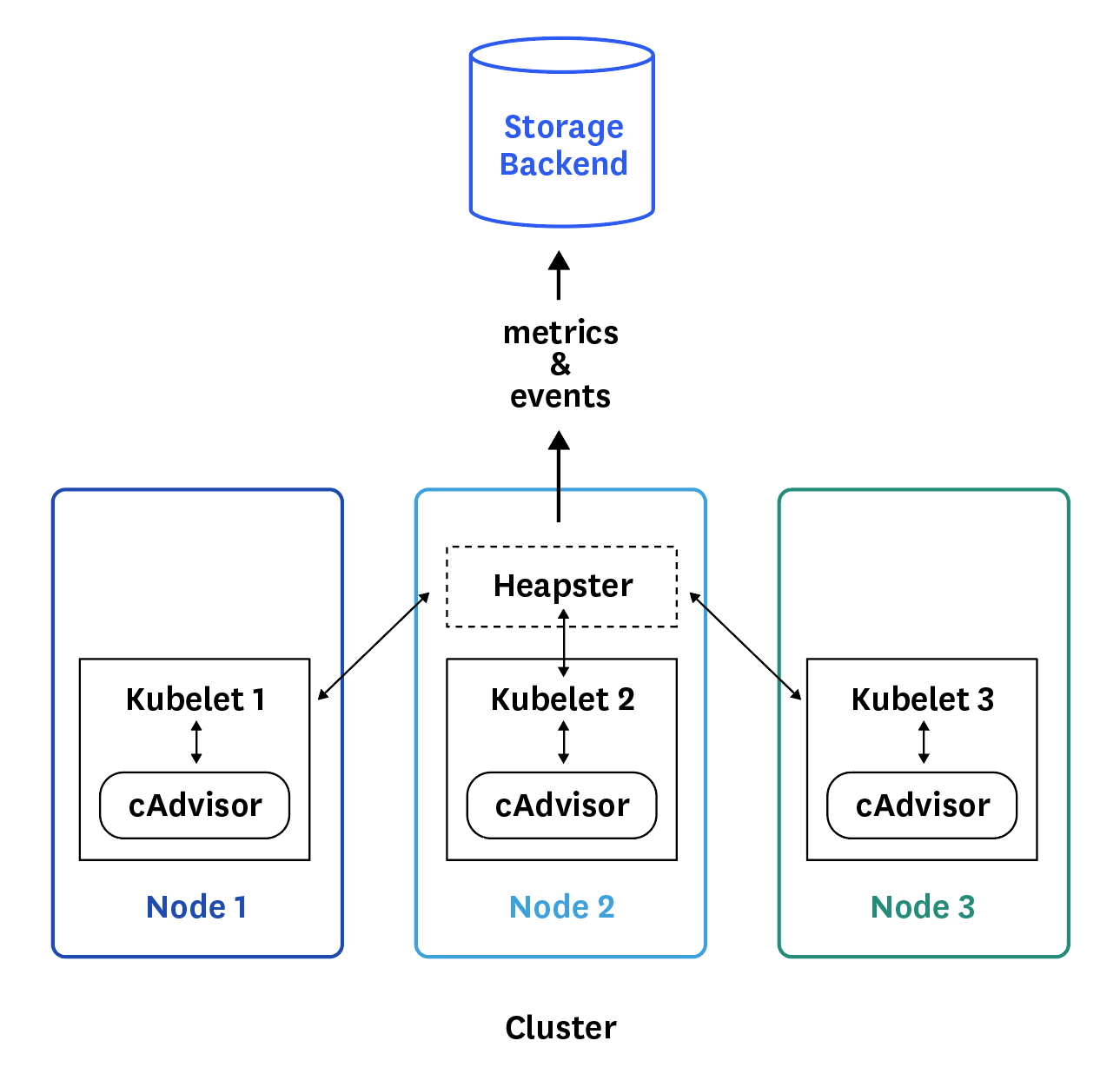
如果ReadnessProbe探针检测失败，则Pod的状态将被修改。Endpoint Controller将从Service的Endpoint中删除包含该容器所在Pod的Endpoint。

### cAdvisor

cAdvisor是一个开源的分析容器资源使用率和性能特性的代理工具。

在k8s项目中，cAdvisor被集成到k8s代码中，cAdvisor自动查找所有在其所在节点上的容器，自动采集CPU、内存、文件系统和网络使用的统计信息。

cAdvisor通过它所在节点机的Root容器，采集并分析该节点机的全面使用情况。



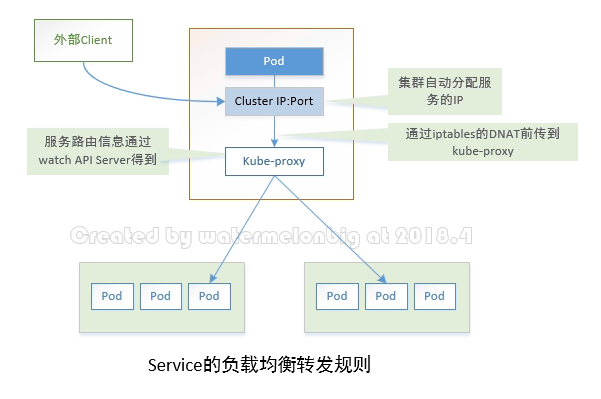
## Kube-proxy

### 工作模式

|  |  |  |
| --- | --- | --- |
| **模式** | **说明** | **补充说明** |
| userspace | 最早的负载均衡方案  它在用户空间监听一个端口，所有服务通过 iptables 转发到这个端口，然后在其内部负载均衡到实际的Pod  主要问题是效率低，有明显的性能瓶颈 |  |
| iptables | 目前推荐的方案  完全以 iptables 规则的方式来实现 service 负载均衡。  主要问题是在服务多的时候产生太多的 iptables 规则，非增量式更新会引入一定的时延，大规模时有明显的性能问题 |  |
| ipvs | 为解决 iptables 模式的性能问题  v1.11 新增 ipvs 模式(v1.8 开始支持测试版，并在 v1.11 GA)  采用增量式更新，并可以保证 service 更新期间连接保持不断开 | Node 上加载内核模块:  nf\_conntrack\_ipv4  ip\_vs  ip\_vs\_rr  ip\_vs\_wrr  ip\_vs\_sh |
| winuserspace | 同 userspace，但仅工作在 windows 节点上 |  |

### 工作机制

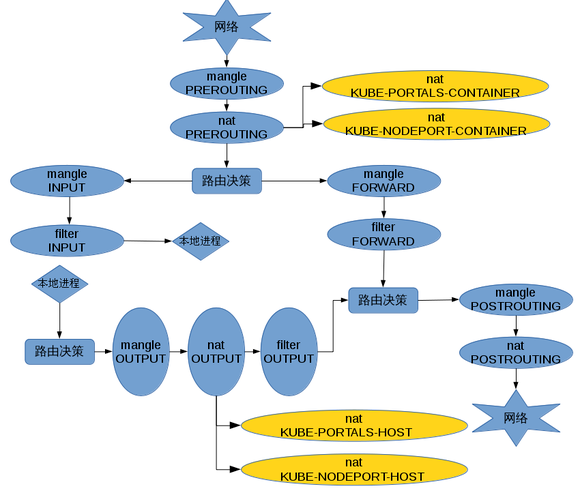
支撑Service概念落地的服务。



Kube-proxy基于iptables实现，并拓展了所需的链：

|  |  |
| --- | --- |
| **名称** | **作用** |
| KUBE-SERVICES | 操作跳转规则的主要chain |
| KUBE-NODEPORTS | 针对通过nodeport访问的package做的操作 |
| KUBE-POSTROUTING |  |
| KUBE-MARK-MASQ | 对于符合条件的包 set mark 0x4000,  有此标记的数据包会在KUBE-POSTROUTING chain中统一做MASQUERADE |
| KUBE-MARK-DROP | 对于未能匹配到跳转规则的traffic set mark 0x8000  有此标记的数据包会在filter表drop掉 |

Kube-proxy通过为默认的PREROUTING，Output和POSTROUTING chain增加规则，跳转至Kubernetes自定义chain。



Kube-proxy内部也创建了一个负载均衡器LoadBalancer，保存了Service到对应的后端Endpoint列表的动态路由转发路由表，而具体的路由选择则取决于RoundRobin负载均衡算法及Service的Session会话保持（SessionAffinity）这两个特性。

### 执行流程

* 针对发生变化的Service列表，kube-proxy的具体处理流程如下：
  + 如果该Service没有设置集群IP（ClusterIP），则不做任何处理，否则，获取该Service的所有端口定义列表（spec.ports域）。
  + 逐个读取服务端口定义列表中的端口信息，根据端口名称、Service名称和Namespace判断本地是否已经存在对应的服务代理对象，如果不存在则新建；如果存在并且Service端口被修改过，则先删除iptables中和该Service端口相关的规则，关闭服务代理对象，然后走新建流程，即为该Service端口分配服务代理对象并为该Service创建相关的iptables规则。
  + 更新负载均衡器组件中对应Service的转发地址列表，对于新建的Service，确定转发时的会话保持策略。
  + 对于已删除的Service则进行清理。
* 针对Endpoint的变化，kube-proxy会自动更新负载均衡器中对应Service的转发地址列表。

以上两个步骤，kube-proxy针对iptables所做的一些具体操作如下：

kube-proxy在启动时和监听到Service或Endpoint的变化后，会在本机iptables的NAT表中添加4条规则链。

* KUBE-PORTALS-CONTAINER：从容器中通过Service Cluster IP和端口号访问Service的请求。
* KUBE-PORTALS-HOST：从主机中通过Service Cluster IP和端口号访问Service的请求。
* KUBE-NODEPORT-CONTAINER：从容器中通过Service的NodePort端口号访问Service的请求。
* KUBE-NODEPORT-HOST：从主机中通过Service的NodePort端口号访问Service的请求。

# 常用操作

## Kubectl

入口代码 cmd/kubectl/kubectl.go

### kubectl create命令

调用kubernetes API Server提供的REST API来创建kubernetes资源对象。

例如Pod、Service、RC等。资源的描述对象来自-f指定的文件或者来自命令行的输入流。

|  |
| --- |
| NewCmdCreate() [pkg/kubectl/cmd/create/create.go]  -> RunCreate()：通过 f.NewBuilder() 调用 f.builder()  -> f.builder():根据命令行参数，创建对应visitor对象获取相关资源，遍历相关的所有visitor对象，触发用户指定的visitorFun回调函数处理具体资源，最终完成资源对象业务逻辑  -> FilenameParam()：解析传入的filenames资源  -> Flatten()：表示资源对象为一个数组，使用 FlattenListVisitor 来遍历  -> Do()：返回一个Rest对象，包括资源相关的visitor对象 |

### kubectl rolling update命令

负责滚动更新升级RC(ReplicationController)

**场景：执行中断，继续执行**

执行过程中会在当前rolling update上增加一个Annotation标签-kubectl.kubernetes.io/next-controller-id，标签的值就是下一个要执行的新RC的名字。

**场景：Image升级**

还会在RC的Selector上贴一个名为deploymentKey的Label，Label的值时RC的内容进行Hash计算后的值(类似签名)，可方便的比较RC里的Image名字是否发生了变化。

|  |
| --- |
| NewCmdRollingUpdate() [pkg/kubectl/cmd/rollingupdate.go]  -> Run() [pkg/kubectl/cmd/rollingupdate.go]  -> findNewName()：查询新RC名；若未提供，则从Old RC 中根据 kubectl.kubernetes.io/next-controller-id 标签找新RC的名字并返回  -> 新RC存在则继续使用，否则调用CreateNewControllerFromCurrentController方法创建一个新RC  - 在新RC的创建过程中设定deploymentKey为自己的Hash签名  -> NewRollingUpdater()：生成RollingUpdater对象  RollingUpdater的方法是rolling update的核心  - 核心流程是每次让新RC的Pod副本数量加1，同时旧RC的Pod副本数量减1，直到新RC的Pod副本数量达到预期值同时旧RC的Pod副本数量变为零为止  - 在这个过程中由于新旧RC的Pod副本一直在变动  - 因此需要记录最初不变的Pod副本数量，这就是RC的Annotation标签-kubectl.kubernetes.io/desired-replicas |

# 源码笔记

## pkg/scheduler

### nodeinfo/node\_info.go

节点信息、主要数据结构、基本操作

|  |
| --- |
| type ImageStateSummary struct {//镜像信息摘要  Size int64 //镜像大小  NumNodes int //使用该镜像的node数量  } |

|  |
| --- |
| type NodeInfo struct { // Node信息结构体  node \*v1.Node  pods []\*v1.Pod  podsWithAffinity []\*v1.Pod  usedPorts HostPortInfo  requestedResource \*Resource // 当前node上所有pod的request总和  nonzeroRequest \*Resource // 当前node上所有非0(资源大于最小可用量)的pod数  allocatableResource \*Resource // node的实际所有的可分配资源(对应于Node.Status.Allocatable.\*)，可理解为node的资源总量  taints []v1.Taint // 快速查询用  taintsErr error  imageStates map[string]\*ImageStateSummary  TransientInfo \*TransientSchedulerInfo //  memoryPressureCondition v1.ConditionStatus // 快速查询用  diskPressureCondition v1.ConditionStatus // 快速查询用  pidPressureCondition v1.ConditionStatus // 快速查询用  generation int64 // (类似版本号)NodeInfo变化时，递增  } |
|  |

### algorithm/predicates/predicates.go

分配算法中的predicates阶段实现，用于选择Pod应该运行的Node节点

#### 主数据结构

|  |
| --- |
| type VolumeFilter struct { // VolumeFilter：通过检查PD容量来过滤/选择 volume  FilterVolume func(vol \*v1.Volume) (id string, relevant bool) //过滤普通volume  FilterPersistentVolume func(pv \*v1.PersistentVolume) (id string, relevant bool) //过滤持久化volume  MatchProvisioner func(sc \*storagev1.StorageClass) (relevant bool) //通过StorageClass的存储策略，进行过滤  IsMigrated func(csiNode \*storagev1beta1.CSINode) bool // plugin是否迁移到CSI驱动  } |

#### 主函数

Predicate()，初筛合适node

|  |
| --- |
| func (c \*VolumeZoneChecker) predicate  (pod \*v1.Pod, meta PredicateMetadata, nodeInfo \*schedulernodeinfo.NodeInfo)  (bool, []PredicateFailureReason, error) {  1、len(pod.Spec.Volumes) == 0 ，为空白pod，直接选用    2、通过 v1的 LabelZoneFailureDomain 和 LabelZoneRegion，去匹配 node 的标签  2.1、合适的放入 候选集nodeConstraints  2.2、若 候选集nodeConstraints > 0，则认为该 node 合适    3、PV的分配  3.1、不指定PV，直接成功  3.2、按驱动类型过滤  3.3、按标签过滤，得到 候选集volumeVSet  3.4、volumeVSet 和 nodeConstraints 无法对应上，则认为失败  4、经过以上过滤，都未失败，则认为成功  } |

### proxy/service.go

|  |
| --- |
| type BaseServiceInfo struct { // Service的基本信息  clusterIP net.IP  port int  protocol v1.Protocol  nodePort int  loadBalancerStatus v1.LoadBalancerStatus  sessionAffinityType v1.ServiceAffinity  stickyMaxAgeSeconds int  externalIPs []string  loadBalancerSourceRanges []string  healthCheckNodePort int  onlyNodeLocalEndpoints bool  } |

|  |
| --- |
| type ServiceChangeTracker struct { //记录未提交的service变更记录，key为namespace和名称  lock sync.Mutex  items map[types.NamespacedName]\*serviceChange // 类型为ServiceMap的 {previous, current}  makeServiceInfo makeServicePortFunc // 让 proxier 注入相关信息  isIPv6Mode \*bool  recorder record.EventRecorder  } |

### proxy/endpoint.go

类同 proxy/service.go，如

EndpointChangeTracker 类似 ServiceChangeTracker

## pkg/kubelet

### server/stats/handler.go

接收外部统计请求，转而向 cAdvisor 获取实际数据，然后返回给请求方

### server/streaming/server.go

|  |
| --- |
| type Server interface { // 处理流式请求的Server  http.Handler  // 处理方法，实际指向 k8s.io/cri-api/pkg/apis/runtime/v1alpha2 的实现方法  GetExec(\*runtimeapi.ExecRequest) (\*runtimeapi.ExecResponse, error)  GetAttach(req \*runtimeapi.AttachRequest) (\*runtimeapi.AttachResponse, error)  GetPortForward(\*runtimeapi.PortForwardRequest) (\*runtimeapi.PortForwardResponse, error)  Start(stayUp bool) error // stayUp：true表示由Stop()方法关闭；否则在所有期望链接关闭后，server自动结束。  // 调用 Get{Exec,Attach,PortForward} 会计数  Stop() error  } |

### container/os.go

OS操作的封装，会将抽象的os操作，转为实际os对象的操作

|  |
| --- |
| type RealOS struct{} // RealOS 声明一个类型，后面用作类似void指针，指向具体的实现类  func (RealOS) MkdirAll(path string, perm os.FileMode) error {  return os.MkdirAll(path, perm)  } |

### container/runtime.go

包含很多runtime环境中重要的数据结构

|  |
| --- |
| type Runtime interface { // 要求实现是线程安全的  Type() string  Version() (Version, error)  APIVersion() (Version, error) // 缓存container runtime中的API version信息(要求实现定期更新该缓存)  Status() (\*RuntimeStatus, error) // 返回runtime状态，当runtime为出错/nil时返回error  GetPods(all bool) ([]\*Pod, error)  GarbageCollect(gcPolicy ContainerGCPolicy, allSourcesReady bool, evictNonDeletedPods bool) error // 指定gc以清除"dead"状态container  SyncPod(pod \*v1.Pod, podStatus \*PodStatus, pullSecrets []v1.Secret, backOff \*flowcontrol.Backoff) PodSyncResult // 将pod同步到本对象  KillPod(pod \*v1.Pod, runningPod Pod, gracePeriodOverride \*int64) error // kill掉Pod中的所有container  GetPodStatus(uid types.UID, name, namespace string) (\*PodStatus, error)  GetContainerLogs(ctx context.Context, pod \*v1.Pod, containerID ContainerID, logOptions \*v1.PodLogOptions, stdout, stderr io.Writer) (err error)  DeleteContainer(containerID ContainerID) error // 删除指定container。当该container处于"running"状态时，返回error  ImageService // 提供image相关方法  UpdatePodCIDR(podCIDR string) error // 向runtime发送一个新的podCIDR。本方法只是转发给runtime shim  } |

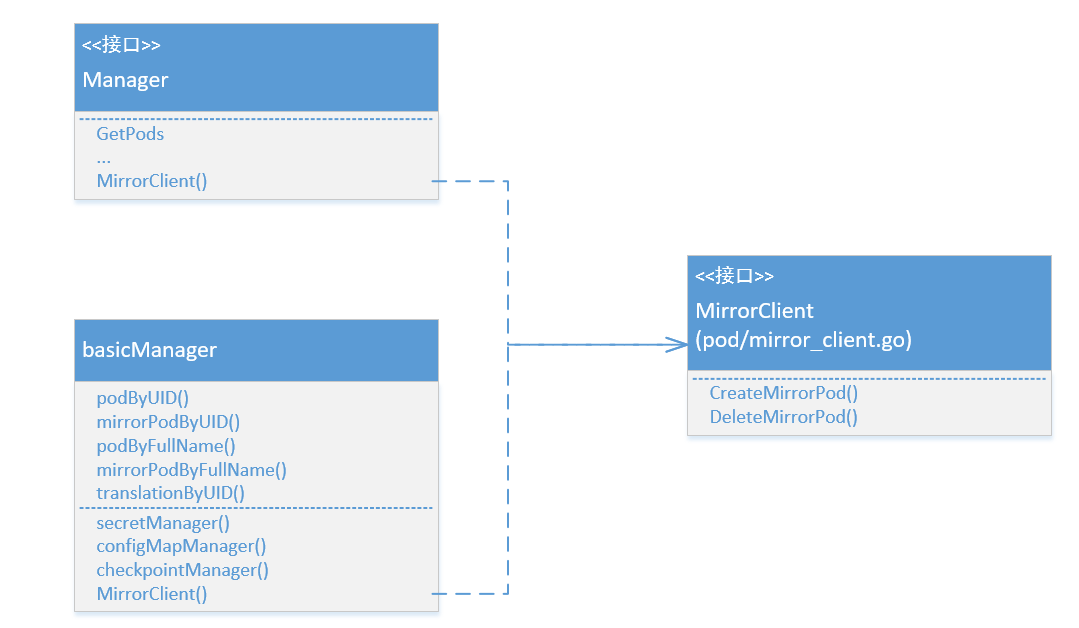
|  |
| --- |
| type ImageService interface {  PullImage(image ImageSpec, pullSecrets []v1.Secret, podSandboxConfig \*runtimeapi.PodSandboxConfig) (string, error) // 通过网络拉取image到本地  GetImageRef(image ImageSpec) (string, error) //返回本地的镜像引用，如果本地没有该镜像，返回("", nil)  ListImages() ([]Image, error) // 获取本地所有images列表  RemoveImage(image ImageSpec) error // 移除指定image  ImageStats() (\*ImageStats, error) // 获取image的统计数据  } |

|  |
| --- |
| type PodPair struct { // 包括，runtime#Pod 和 api#Pod (?但不知道用途)  APIPod \*v1.Pod // runtime#Pod  RunningPod \*Pod //api#Pod(定义在 pkg/kubelet/container/runtime#Pod)  }  func (p \*Pod) ToAPIPod() \*v1.Pod {} //Pod到v1.Pod的转换方法 |



### pod/pod\_manager.go

Pod对象管理，包括Pod信息存储、静态Pod和镜像Pod之间映射关系管理等。



## pkg/quota

### v1/evaluator/core/pods.go

|  |
| --- |
| func (p \*podEvaluator) Constraints(required []corev1.ResourceName, item runtime.Object) error { // 根据可供资源的类型，筛选是否满足  // 1.将Pod数据统一转成 corev1.Pod 格式，失败则返回错误  pod, err := toExternalPodOrError(item)  // BACKWARD COMPATIBILITY REQUIREMENT: if we quota cpu or memory, then each container  // must make an explicit request for the resource. this was a mistake. it coupled  // validation with resource counting, but we did this before QoS was even defined.  // let's not make that mistake again with other resources now that QoS is defined.    // 2.在 validationSet 资源类型范围内，根据资源充足情况(i.e. requests < limits)，筛选  // - 根据 Containers 进行筛选  // - 根据 InitContainers 进行筛选  } |

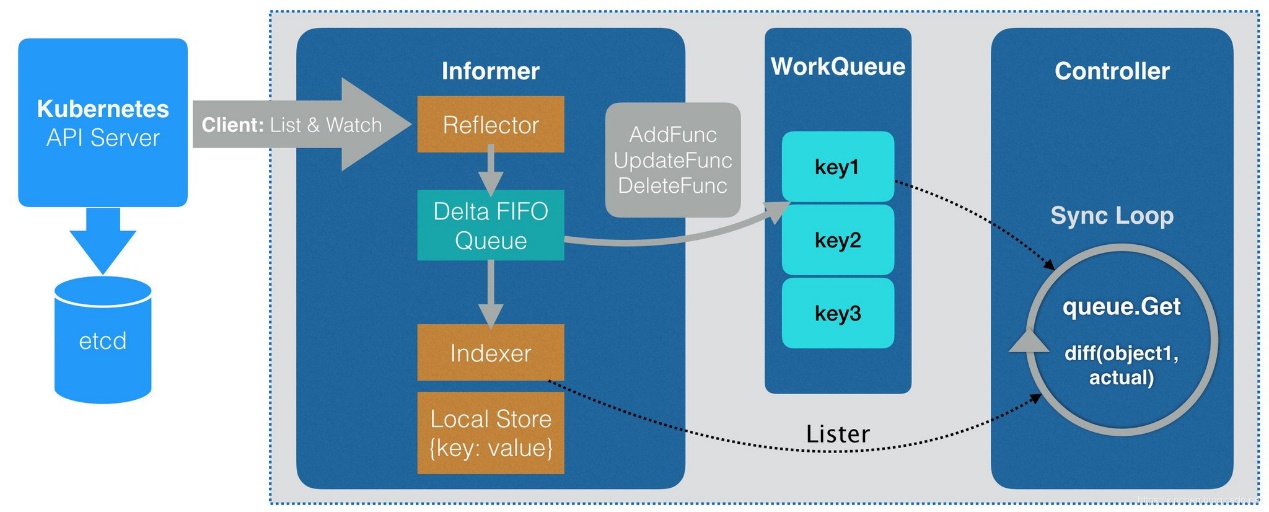
部分主要实体

# 开发/扩展

## 自定义controller：Student

controller的作用就是监听指定对象的新增、删除、修改等变化，针对这些变化做出相应的响应（例如新增pod的响应为创建docker容器）。

如下图，API对象的变化会通过Informer存入队列（WorkQueue），在Controller中消费队列的数据做出响应，响应相关的具体代码就是我们要做的真正业务逻辑；



假设要定义对一个Student实体操作的controller，参考以下步骤：

### 创建CRD：使K8S能识别Student类型

CRD(Custom Resource Definition)，可以参考[官方文档](https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/#create-a-customresourcedefinition)。

1. **登录可以执行kubectl命令的机器，创建student.yaml文件**

内容如下：

|  |
| --- |
| apiVersion: apiextensions.k8s.io/v1beta1  kind: CustomResourceDefinition  metadata:  # metadata.name的内容是由"复数名.分组名"构成，如下，students是复数名，bolingcavalry.k8s.io是分组名  name: students.bolingcavalry.k8s.io  spec:  # 分组名，在REST API中也会用到的，格式是: /apis/分组名/CRD版本  group: bolingcavalry.k8s.io  # list of versions supported by this CustomResourceDefinition  versions:  - name: v1  # 是否有效的开关.  served: true  # 只有一个版本能被标注为storage  storage: true  # 范围是属于namespace的  scope: Namespaced  names:  # 复数名  plural: students  # 单数名  singular: student  # 类型名  kind: Student  # 简称，就像service的简称是svc  shortNames:  - stu |

1. **在student.yaml所在目录执行命令kubectl apply -f student.yaml**

即可在k8s环境创建Student的定义。

今后如果发起对类型为Student的对象的处理，k8s的api server就能识别到该对象类型。

用命令查看更多细节(stu是在student.yaml中定义的简称)：

kubectl get crd

kubectl describe crd stu

1. **在ectd中查看CRD**

如果配置了etcdctl，可用以下命令

|  |
| --- |
| ETCDCTL\_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/healthcheck-client.crt --key=/etc/kubernetes/pki/etcd/healthcheck-client.key get /registry/apiextensions.k8s.io/customresourcedefinitions/ --prefix |

### 创建对象：使K8S能识别Student对象身份

1. **创建object-student.yaml文件**

内容如下：

|  |
| --- |
| apiVersion: bolingcavalry.k8s.io/v1  kind: Student  metadata:  name: object-student  spec:  name: "张三"  school: "深圳中学" |

1. **创建对象**

在object-student.yaml文件所在目录执行命令kubectl apply -f object-student.yaml，会看到提示创建成功

查看已创建成功的Student对象：

kubectl get stu

1. **在ectd中查看对象**

如果配置了etcdctl，可用以下命令

|  |
| --- |
| ETCDCTL\_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/healthcheck-client.crt --key=/etc/kubernetes/pki/etcd/healthcheck-client.key get /registry/bolingcavalry.k8s.io/students/default/object-student --print-value-only |

### 自动生成代码

编写API对象Student相关的声明的定义代码，然后用代码生成工具结合这些代码，自动生成Client、Informet、WorkQueue相关的代码。

### 编写controller代码

### 总结

1、创建CRD（Custom Resource Definition），令k8s明白我们自定义的API对象；

2、编写代码，将CRD的情况写入对应的代码中，然后通过自动代码生成工具，将controller之外的informer，client等内容较为固定的代码通过工具生成；

3、编写controller，在里面判断实际情况是否达到了API对象的声明情况，如果未达到，就要进行实际业务处理，而这也是controller的通用做法；

实际编码过程并不复杂，动手编写的文件如下：

|  |
| --- |
| ├── controller.go  ├── main.go  └── pkg  ├── apis  │ └── bolingcavalry  │ ├── register.go  │ └── v1  │ ├── doc.go  │ ├── register.go  │ └── types.go  └── signals  ├── signal.go  ├── signal\_posix.go  └── signal\_windows.go |

# 编译/安装

在一台1C2GB的Centos7.6云服务器上尝试源码安装，还未成功2333333

一直失败在编译阶段，具体是deepcopy-gen编译失败，还未找到原因。

# 常见问题

## Pod相关问题

### Pod始终处于Pending状态

      表示Pod无法被正常的调度到节点上，通常由于某种系统资源无法满足Pod运行的需求，具体可能原因：

1. **系统没有足够的资源**

已经用尽了集群中所有的CPU或内存资源。需要清理一些不在需要的Pod，调整它们所需的资源量，或者向集群中增加新的节点。

1. **用户指定了hostPort**

通过hostPort用户能够将服务暴露到指定的主机端口上，会限制Pod能够被调度运行的节点

### Pod始终处于Waiting状态

表示已经被调度到了一个工作节点，却无法在那个节点上运行。

可以使用kubectl describe 含有更详细的错误信息。

具体可能原因：

**1、无法下载镜像**

最经常导致Pod始终Waiting的原因。

### Pod处于CrashLoopBackOff 状态

CrashLoopBackOff状态说明容器曾经启动，但又异常退出了。此时 Pod 的 RestartCounts 通常是大于 0，具体可能原因：

1. **容器进程退出**

**2、健康检查失败退出**

**3、OOM Killed**

### Pod一直崩溃或运行不正常

可以使用kubectl describe以及kubectl logs排查问题，但是这个一般也不确定

可能原因：

1. **健康检测失败**
2. **OOM情况**
3. **容器运行生命周期结束**

## 集群相关问题

### 集群雪崩

集群中节点的连锁反应

可能原因：

1. **Kubelet预留资源被占用**

业务占用资源过多，影响到Kubelet收集、上报节点状态信息

kublet加上如下配置，以保证在Node高负荷时，也能保证当kubelet需要cpu时至少能有--kube-reserved设置的cpu cores可用：

|  |
| --- |
| --enforce-node-allocatable=pods,kube-reserved,system-reserved  --kube-reserved-cgroup=/kubelet.service  --system-reserved-cgroup=/system.slice  或  --eviction-hard=memory.available<1024Mi,nodefs.available<10%,nodefs.inodesFree<5%  --system-reserved=cpu=0.5,memory=1G  --kube-reserved=cpu=0.5,memory=1G  --kube-reserved-cgroup=/system.slice/kubelet.service  --system-reserved-cgroup=/system.slice  --enforce-node-allocatable=pods,kube-reserved,system-reserved |

## 存储/磁盘相关问题

### NFS挂载错误

**1、缺少必要工具包**

提示：nfs挂载错误wrong fs type, bad option, bad superblock

查看/sbin/mount.<type>文件，果然发现没有/sbin/mount.nfs的文件，安装nfs-utils即可

### PV无法删除

**1、PV始终处于“Terminating”状态，且delete不掉**

删除k8s中的记录：

kubectl patch pv xxx -p '{"metadata":{"finalizers":null}}'

# 附录

## 资料

<https://github.com/kubernetes/kubernetes> 源码

<https://www.cnblogs.com/DaweiJ/articles/8865762.html> K8S各模块简述&机制

<https://blog.csdn.net/zhonglinzhang/article/details/85257134> K8S坑问题汇总

<https://github.com/kubernetes/kubernetes/blob/master/pkg/proxy/ipvs/README.md> IPvs在各种模式下的工作原理

<https://blog.csdn.net/boling_cavalry/article/details/88917818> 用户自定义controller