ecnf项目分享

背景

本篇内容主要面向容器相关开发时遇到的和 golang 有关内容，假定阅读者已经有语言基础，对容器和其相关业务不熟悉

go语言

一种静态类型、编译型语言，旨在提供简洁、高效的编程体验。Go 语言结合了传统编译型语言的性能优势和脚本语言的开发效率，特别适合构建可扩展的网络服务和分布式系统

- go.dev 是 Go 语言的官方网站，提供了丰富的资源和信息，包括：
  - 文档 介绍语言的各方面，包括包管理，内存模型，子命令等
    - https://pkg.go.dev/cmd/compile # 编译优化
    - https://go.dev/ref/mem # 内存模型
    - https://go.dev/doc/cmd # 子命令
  - 教程和指南：提供从入门到高级的教程，帮助开发者学习 Go 语言。
    - https://go.dev/doc/effective_go
  - 发布：展示 Go 发布信息
    - https://go.dev/doc/devel/release
  - 工具和库：列出了社区开发的工具和库，方便开发者使用
    - https://pkg.go.dev/std # 标准库, 也可搜索

安装

- https://studygolang.com/dl

项目布局

- 参考 https://github.com/golang-standards/project-layout/blob/master/README_zh.md

lint检查

- golangci-lint 语言静态代码分析工具，它集成了多种 Go 语言的 linter，用于帮助开发者发现代码中的潜在问题、风格不一致、性能问题
- CI 必不可少的工具，配置在项目的根目录文件上 .golangci.yml
- 更多细节：https://golangci-lint.run

profile

- 工具：https://github.com/google/pprof
  - go install github.com/google/pprof@latest
- 采集后的数据默认保存到目录 $HOME/pprof/，常用的类型
  - cpu: /debug/pprof/profile 确定应用程序消耗 CPU 周期时花费时间的位置
  - heap: /debug/pprof/heap 进行堆分配时记录堆栈跟踪，用于监视当前和历史内存使用情况，以及检查内存泄漏
  - goroutine： 报告所有当前goroutine的堆栈信息
  - block: /debug/pprof/block, 查看导致阻塞同步的堆栈跟踪, 默认情况下不启用块配置文件
- 以 heap 为例
  - pprof -tls_key key.pem -tle_cert cert.pem -tls_ca ca.pem -seconds 30 https://127.0.0.1:10250/debug/pprof/heap # 采集 heap 信息
  - pprof -http :8089 cpu.profile # 启动 web 分析
- 更多细节见： https://go.dev/blog/pprof

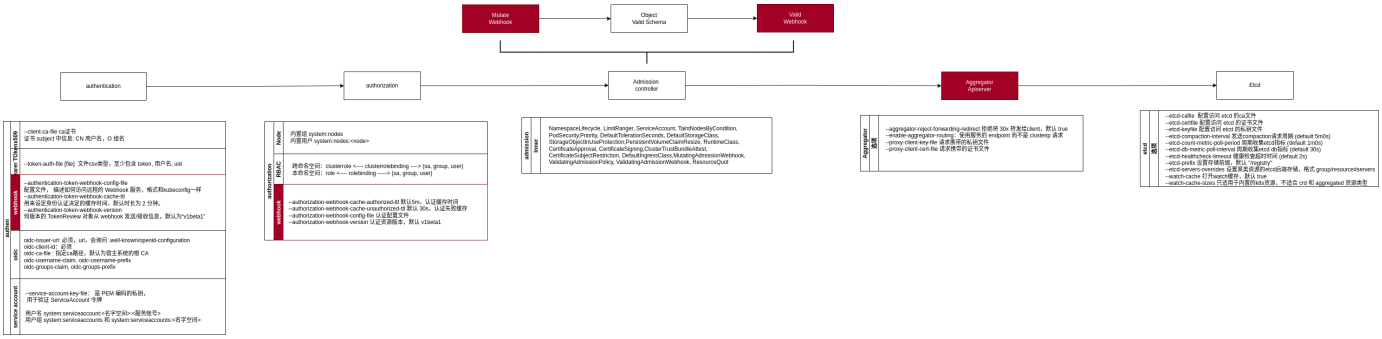
```yaml
# 常用配置 .golangci.yml
run:
  timeout: 5m
  allow-parallel-runners: true
  allow-serial-runners: true
linters:
  presets:
    - bugs
    - error
    - performance
    - unused
  fast: true
issues:
  exclude-dirs:
    - vendor/
```

kubernetes

- 由多个组件组成，包括控制平面组件（如 kube-apiserver、etcd、kube-controller-manager 和 kube-scheduler）和节点组件（如 kubelet 和 kube-proxy）

apiserver

- 架构及配置



- 上图中介绍 apiserver 内部主要组成内容，其中 红色部分是可以外部扩展的地方
  - authen 认证可以使用外部 webhook
  - author 授权可以使用外部 webhook
  - admission 资源注入和校验
    - mutateWebhook 资源注入
    - ValidWebhook 资源校验
  - aggregetor：使用 apiservers 资源定义
- 有关kubectl 命令
  - kubectl api-resources # 查看注册的资源类型，版本
  - kubectl get apiservices # 查看apiservice，包含 aggregate 扩展
  - kubectl explain pod.spec # 查看pod.spec 中有哪些配置字段及字段含义
  - kubectl get sc # 获取主要组件状态，包括 scheduler，etcd，controller

资源路径格式

- cluster scope
  - /apis/G/V/R/NAME/SUBRESOURCE
- namespace scope
  - /apis/G/V/namespaces/[ns]/R/NAME/SUBRESOURCE

```
/api/v1/namespaces/a/secrets/a
/apis/networking.k8s.io/v1/namespaces/a/ingresses/a
/apis/apiregistration.k8s.io/v1/apiservices/a
```

admissionWebhook

- 使用 Admission 结构
- Request 数据结构
  - UID types.UID // UID是单个请求/响应的标识符，区分那些在其他方面相同的请求实例
  - Kind metav1.GroupVersionKind // 标识资源类型，比如 v1.Pod 或 autoscaling.v1.Scale等
  - Resource metav1.GroupVersionResource // 请求的完全限定的资源 (如 v1.pods)

    - SubResource string // 可选，子资源名称，如 status，scale
    - Name string // 可选，如果是创建操作，名称可能是服务端生成
    - Namespace string // 可选，命名空间
    - Operation Operation // 操作类型，在 CREATE，UPDATE，DELETE，CONNECT 之一
    - UserInfo authenticationv1.UserInfo // 用户信息，包括名称，id，所在组列表和额外属性
- Response 数据结构
    - UID types.UID // UID是单个请求/响应的标识符,同Request中内容
    - Allowed bool // 表明请求是否被允许
    - Patch []byte // 可选，补丁数据，当前只支持JSONPatch格式
    - PatchType // 可选，当前只有JSONPatch
    - Warnings []string // 可选，在256B大小以内

主要代码

- stage/src/k8s.io：该目录下是 api 核心，有以下主要子目录
    - api: 定义内置资源的struct，以及scheme注册
    - apiserver: 通用 http server 初始化，虽是通用，但包括 admission 准入控制器，audit 审计控制器，authen认证控制器，author 授权控制器，quota 配置控制器，registry/storage 存储控制器，utils(流控,ws, shuffleshareds等),以及 apis 用于集成以上所有
    - apiextensions-apiserver：在apiserver基础上建立 CRD 控制器
    - apimachinery：资源通用结构以及资源编解码

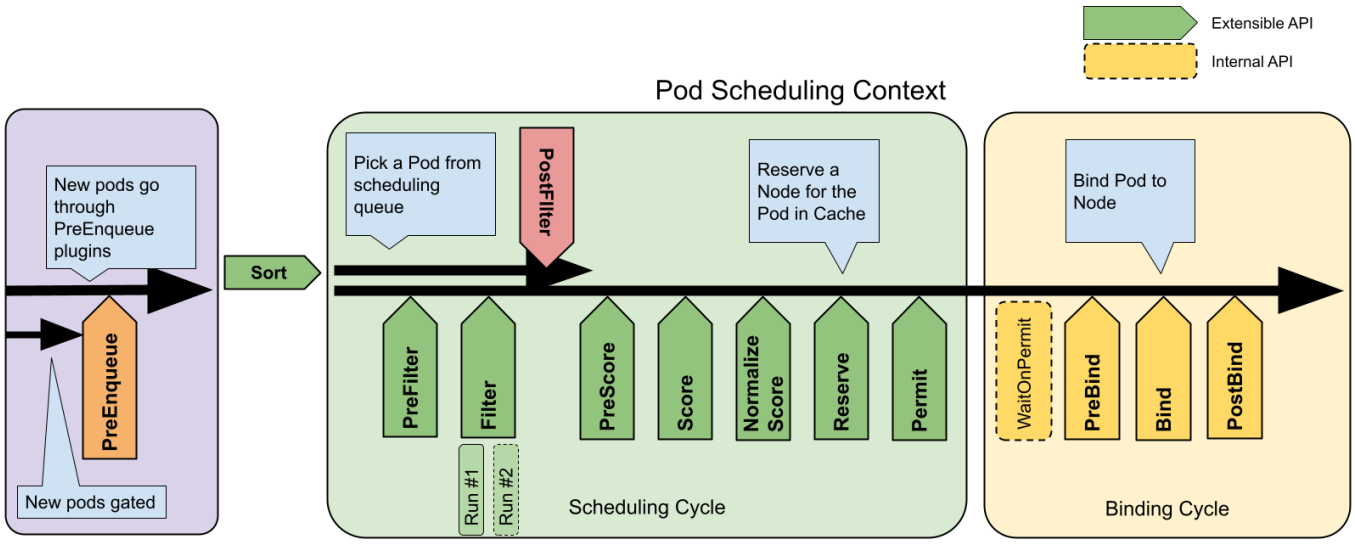
controller

- 集合众多 opeartor 于一体，主要控制器类型以下
    - Service-lb
    - Persistentvolume-attach-detach
    - Certificatesigningrequest-signing
    - Deployment
    - Statefulset
    - Endpoints
    - Endpointslice
    - Endpointslice-mirroring
    - Ephemeral-volume
    - Garbage-collector
    - Horizontal-pod-autoscaler
    - Job

- Cronjob
- Legacy-serviceaccount-token-cleaner
- Namespace
- Node-ipam
- Node-lifecycle
- Persistentvolume-binder
- Pod-garbage-collector
- Replicaset
- Replicationcontroller
- Resourcequota
- Serviceaccount
- Ttl-after-finished
- Validatingadmissionpolicy-status

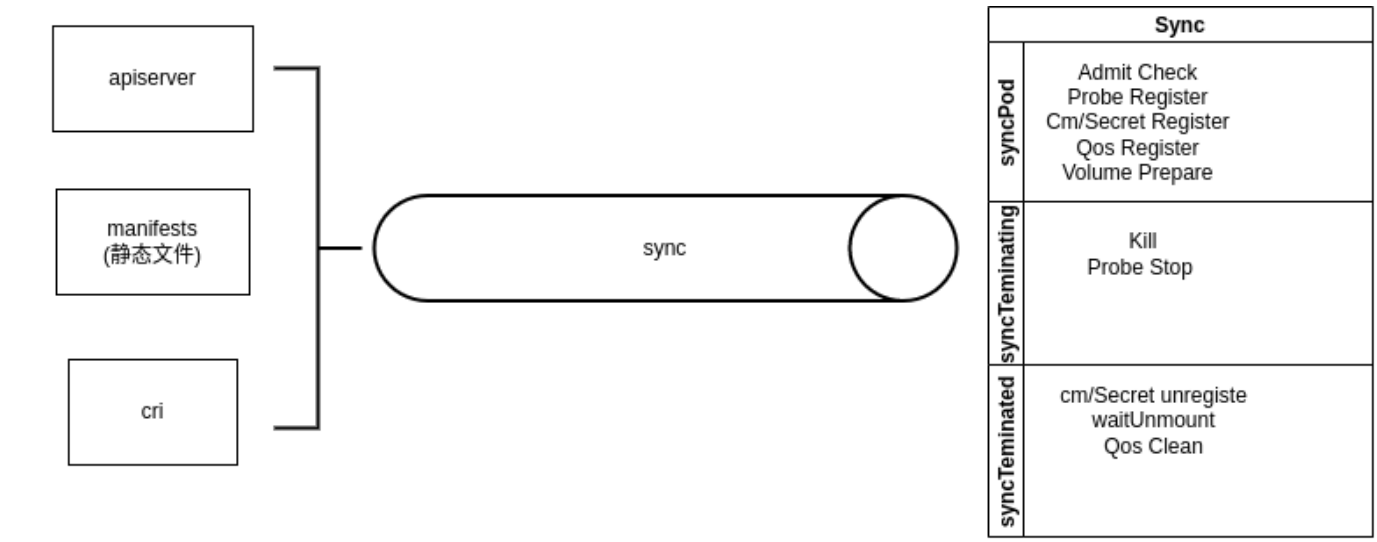
scheduler

- pod 一个个调度，架构



kubelete

- 管理容器的资源以及其生命周期，当前主要涉及部分和其架构



controller	evict	--system-reserved 一组Resource(如cpu=200m,memory=500Mi)，描述为非Kubernetes组件保留的资源 --kube-reserved 一组Resource(如cpu=200m,memory=500Mi)，描述为Kubernetes系统组件保留的资源 --eviction-hard 一组驱逐阈值（例如memory.available<1Gi），如果满足这些阈值，将触发Pod驱逐 --eviction-soft 一组驱逐阈值（如memory.available<1.5Gi），如果在相应的宽限期内满足这些阈值，将触发Pod驱逐 --eviction-soft-grace-period 一组驱逐宽限期（如memory.available=1m30s），对应于软驱逐阈值必须保持多长时间才能触发Pod驱逐 --image-gc-high-threshold int32 磁盘使用百分比，达到此阈值后将始终运行镜像垃圾回收，在[0, 100]范围内 --eviction-pressure-transition-period 在退出驱逐压力条件之前必须等待的持续时间，默认5m
	plugin	**原理:** 1 plugin manager 监听 /var/lib/kubelet/plugins_registry 目录文件创建，并发起 grpc 连接和请求 GetInfo 2 上一步得到的info：类型 + socket 路径 + 版本，一般 socket路径在 /var/lib/kubelet/plugins/ 3 从 plugins 选取合适的 controller，使用上一步 socket路径，之后是各自的 plugin 业务问题 **volume plugin (csi)** 1 kubelet 侧主要使用是 node plugin，其主要方法 publishVolume，用于将卷挂载到 /var/lib/kubelet/volumes 2 kubelet 将上一步的卷使用 mount -t bind {src} {dest} 挂载到容器的 rootfs 中 **device plugin** 1 kubelet 请求 ListAndWatcher 接口，业务需有任何更新上报健康设备信息(包括已经分配) 2 容器内 spec.resources.{vendor}/{device}: {num} 时，kubelet 向 socket 请求 Allocate,返回 env, mount, devices信息 dynamic resource allocation: 新一代 device plugin，主要解决分配粒度问题，调度问题，hook 执行问题
	advisor	使用 cAdvisor 统计容器使用情况，包括 cpu，memory，io 用于驱逐管理

pod spec 和 status 中解读

- 调度有关
  - spec.affinity：根据节点/po的标签，进行亲和和反亲和，同时支持强制和优先两类
  - spec.nodeSelector: 强制调度到有标签的节点
  - spec.nodeName: 跳过调度器，直接在指定节点运行
  - spec.schedulerName: 选择的调度器
  - spec.schedulingGates: 当满足条件时才触发调度，
  - spec.readinessGates：当满足条件时，pod才会被设置为 ready
  - spec.priorityClassName: 优先级有关
  - spec.tolerations: 容忍，和节点的污点对应，白名单关系
- 图中已经介绍 device plugin 原理，需要注意的是上报资源名称时遵循格式 {vendor}/{name}

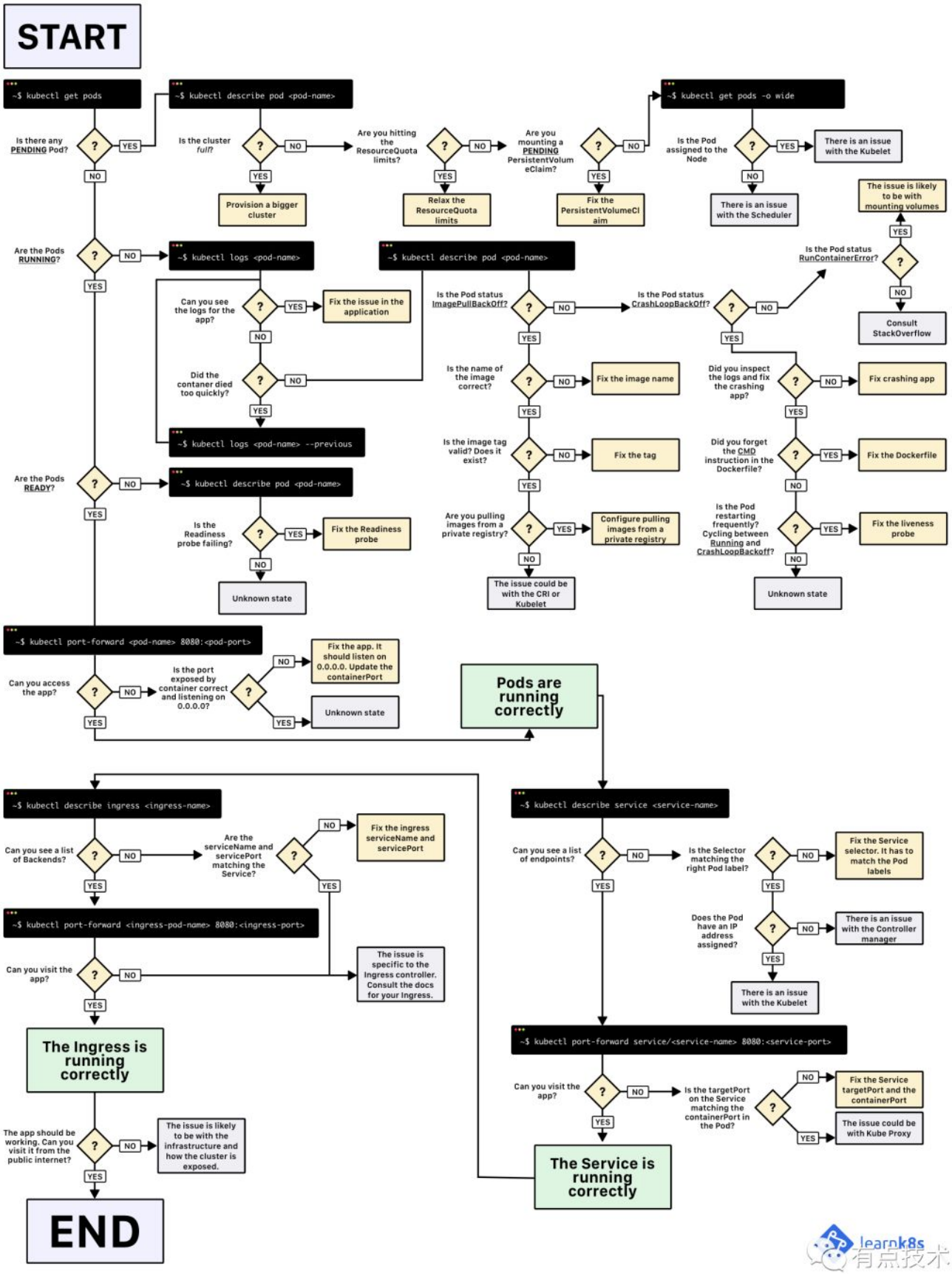
kube-proxy

- 网络模型，该模型确定 cni 具体要做哪些事
  - 每个 pod 都有自己的 IP 地址
  - Pod 内的容器共享 IP 地址，且相互自由通信
  - Pod 可以使用 Pod IP 地址与集群中的其他 Pod 通信
- 域名解析
  - clusterip: {svcname}.{ns}.svc.cluster.local
  - 指定 spec.hostname/subdomain 时，pod 域名 {hostname}.{subdomain}.{ns}.svc.cluster.local
  - headless svc域名：{pod-name}.{svcname}.{ns}.svc.cluster.local
- kube-proxy 主要添加的是 nat/filtre 表，用于 L4 负载均衡，即业务请求 svc 后，具体选择哪个 endpoint 过程，题外话 netfilter提供5链4表，某表可以加入不同链，但不是所有都可以加入；不同表执行主要逻辑都是 match + target，但是 target 能支持的范围不同
- nat 表
  - prerouting KUBE-SERVICES
  - output KUBE-SERVICES
  - postrouting KUBE-POSTROUTING
- filter 表
  - input KUBE-FIREWALL-> KUBE-PROXY-FIREWALL -> KUBE-NODEPORTS -> KUBE-EXTERNAL-SERVICES
  - forward KUBE-PROXY-FIREWALL -> KUBE-FORWARD -> KUBE-SERVICES -> KUBE-EXTERNAL-SERVICES
  - output KUBE-FIREWALL -> KUBE-PROXY-FIREWALL -> KUBE-SERVICES
- KUBE-SERVICES
  - ip 匹配，调到到 KUBE-SVC-xxx 目标，其内容如下
    - port 匹配则 MARK-MASQ (mark 0x4000/0x4000)
    - 跳到 KUBE-SEP-xxx，具体内容如下
      - 源地址是 pod ip，则 MARK-MASQ (mark 0x4000/0x4000)
      - dnat 到 pod+port
  - KUBE-NODEPORTS 最后一条
- KUBE-FIREWALL
  - 源不是127，目的是127 则丢弃
- KUBE-FORWARD
  - accept 0x4000/0x4000 和 已建立
- KUBE-NODEPORTS 必须在最后一条

    - nodePort 匹配，跳到 KUBE-EXT-xx ，内容如下
        - 跳到 MARK-MASQ (标记 mark 0x4000/0x4000)
        - 跳到 KUBE-SVC-xxx , 参考上面记录

定位

- 官方知识文档
    - https://kubernetes.io/zh-cn/docs/home/
    - https://kubernetes.io/docs/
- 问题定位时
    - 主要看 event, node, pod.status 信息
    - 看有关 webhook 通过
        - kubectl get mutatingwebhookconfigurations
        - kubectl get validatingwebhookconfigurations



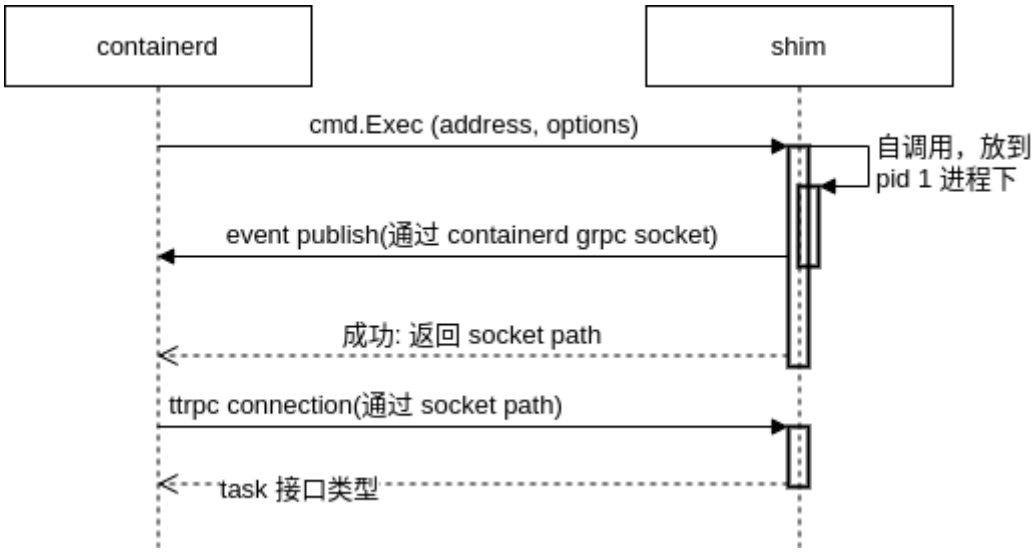
container

- 围绕 oci 标准展开，三种spec
  - runtime：运行时标准，环境中可以看到

        - /run/containerd/io.containerd.runtime.v2.task/k8s.io/$[id]/config.json
    - image：镜像标准
    - distribution：镜像仓库，api 标准, 代码

runtime

- runtime 主要是 cri 实现，当前主要实现有 containerd，crio
- cri 接口： https://github.com/kubernetes/cri-api
- containerd 中 cri 主要配置项目
    - cni：配置 bin 和 etc 目录
    - snapshotter: 镜像挂载方式，当前使用 overlayfs
    - registry：配置仓库及其镜像url 等信息
    - runtimes：配置runtime 信息
        - runtime_type: 运行时名称，拼接为 containerd-shim-[name]-v2
        - cni：覆盖上层配置
        - snapshotter：覆盖上层配置
        - sandboxmode: 在 2.0 版本中添加，设置工作模式, sandbox api 开关
        - options：各自的启动项
- 下图介绍 containerd 和 runtime(shim) 调用过程



image

- 规格
    - docker.v1.1
    - docker.v1.2
    - oci.v1
- 特点：平台独立，内容可寻址
- 什么是内容可寻址，文件以 sha256 命名，并且文件内容具有类型

- 镜像解析过程，对比 oci v1 和 docker v2 版本，另外 containrd 在 2.0 放弃 docker v1 解析，但是发现多架构镜像是 docker v1 版本
  - oci.image.index 多架构元文件
    - oci.image.manifest 单架构元文件，记录 config 和 data layers
      - oci.image.config 镜像配置文件，包含runtime的数据格式(env, cmd, volumes,tty),属性history, os, arch, created),文件层 rootfs
      - oci.image.layer 镜像数据文件
  - distribution.manifest.list 多架构文件
    - distribution.manifest 单架构元文件
      - container.image 镜像配置文件
      - image.rootfs.diff.tar.gzip 镜像数据文件
- 以 busybox 为例，一层 layer 数据，解压出来有如下文件
  - blobs/sha256/50aa4698fa6262977cff...
  - blobs/sha256/65ad0d468eb1c558bf7...
  - blobs/sha256/9ae97d36d26566ff84e...
  - blobs/sha256/ec562eabd705d25bfea...
  - index.json
  - manifest.json
  - oci-layout

```
# index.json 内容
{"schemaVersion":2,"manifests":
[{"mediaType":"application/vnd.oci.image.index.v1+json","digest":"sha256:9ae
97d36d26566ff84e8893c64a6dc4fe8ca6d1144bf5b87b2b85a32def253c7","size":6761,"
annotations":
{"io.containerd.image.name":"docker.yylt.gq/library/busybox:latest","org.ope
ncontainers.image.ref.name":"latest"}}]}

# manifest.json 内容
[{"Config":"blobs/sha256/65ad0d468eb1c558bf7f4e64e790f586e9eda649ee9f130cd0e
835b292bbc5ac","RepoTags":
["docker.yylt.gq/library/busybox:latest"],"Layers":
["blobs/sha256/ec562eabd705d25bfea8c8d79e4610775e375524af00552fe871d33382615
63c"]}]
```

- 在使用命令行工具，如 docker images，ctr img 等会看到id 列，同时在看 manifest 文件，有rootfs 下的 id，这些区别如下
  - image id: config文件的sha256值，是 docker/ctr/critl img 显示的id

    - image digest: manifest文件的sha256值，在 registry 显示
    - diff_ids: 在config文件有 rootfs 字段记录，压缩文件解压后计算 sha256，具体命令
        - cat [file] | gunzip - | sha256sum -
    - chainid: 和parent diffID相加后计算哈希得到，确定上下层关系，在镜像挂载时使用
        - [A, B] => [A, Sha256(A+B)] // Sha256(A+B) 即 chainID

cni

- 标准定义在 https://github.com/containernetworking/cni
- 通常 etc 路径为 /etc/cni/net.d；bin 路径为 /opt/cni/bin
- 版本迭代的主要修改
    - v1.0.0 删除非list配置文件，移除 version 在 plugins中
    - v0.4.0 新增check命令，并且在删除时把preResult传入
    - v0.3.1 常用的版本
- 常用 cni org
    - https://github.com/containernetworking
        - 包含简单单机 ipam， 流量控制等
    - https://github.com/k8snetworkplumbingwg
        - 包含网卡设备 cni，如 bound，ovs，sriov，ib 等
- 配置文件有两种格式 conf 和 conflist

```
# conflist 格式
{
  "cniVersion": "0.3.1",
  "name": "cni0",
  "plugins": [
    {"type": "flannel"}
    ]
}

# conf 格式
{
  "type": "multus"
  "cniVersion": "0.3.1",
  "delegates": [{
      "plugins": [
        { "type": "flannel"},
        { "type": "portmap"}
          ]
```

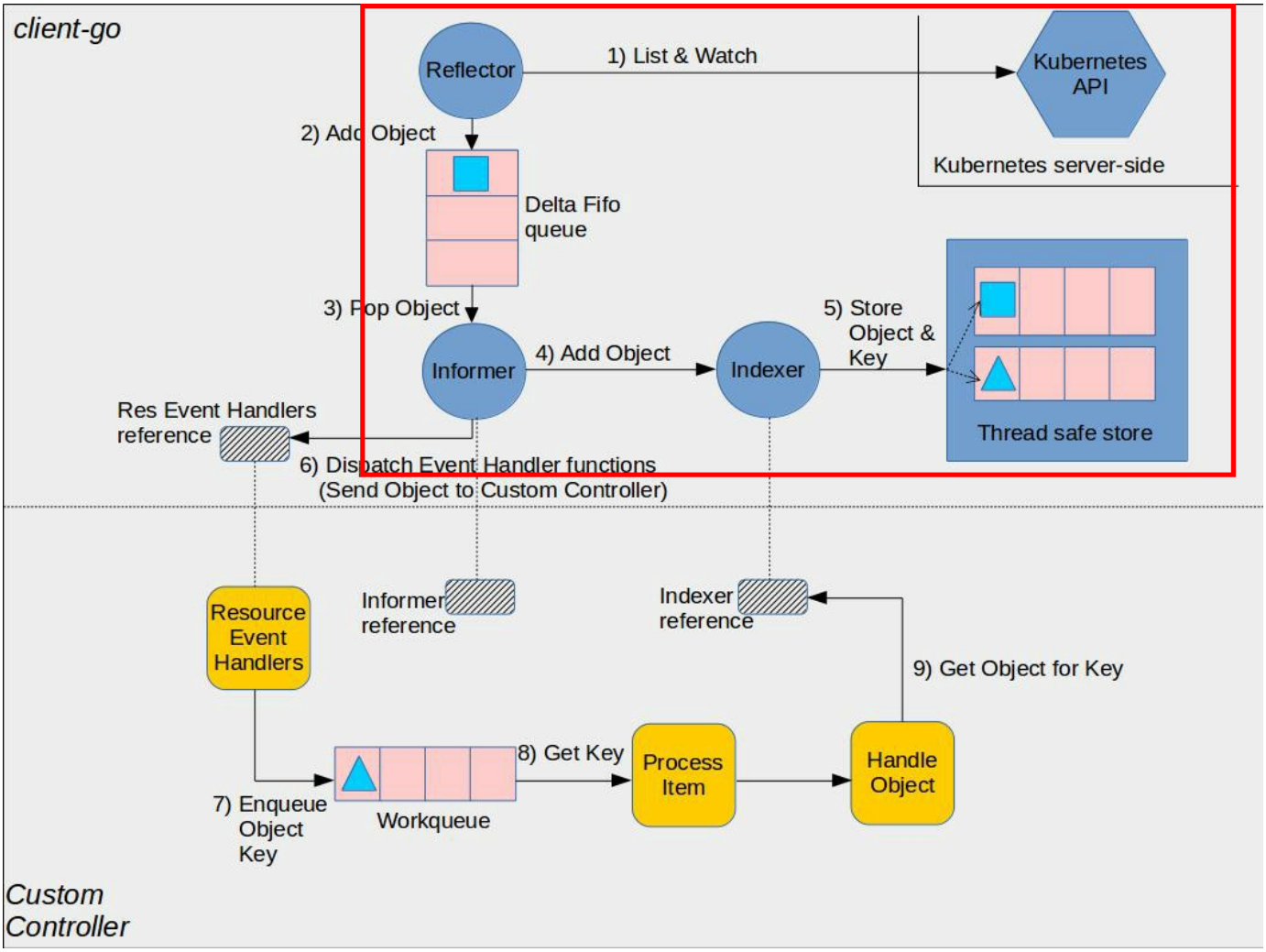
```
        }]
}
```

定位

- containerd 不同命名空间有不同镜像，挂载，运行时
- ctr -n k8s.io i ls #列出k8s.io 空间下的镜像，当镜像是 cri 拉取时，在展示列表有三种形式，repo:tag, repo@sha256, repo:tag@sha256
- ctr -n k8s.io c ls # 列出当前已经创建的容器，包括 initContainer
- ctr -n k8s.io t ls # 列出当前正在运行的容器，是 workerContainers

operator

- 本质上和 kube-controller 是一样的，不同是 controller 是通用， crd 是私用控制
- Operator 是一种 Kubernetes 扩展，它利用自定义资源（Custom Resources）来管理应用程序及其组件，用于处理应用的生命周期，如部署、升级、备份和故障恢复。
- 对于容器开发，都绕不过 operator 的有关内容，相信大家比较熟悉下图，在operator开发过程中，只需要一个函数 Reconcile，本质是 Resource Event Handlers



- 使用的工具有
  - kubebuilder

- controller-gen

开发简单的 Operator 流程

- 创建 Operator 项目：使用 kubebuilder 创建一个新的 Operator 项目。
- 定义自定义资源：使用 Kubernetes API 定义你的自定义资源。
- 实现业务逻辑：在 Golang 中编写处理自定义资源事件的逻辑。
- 测试和部署：在本地或云环境中测试你的 Operator，并将其部署到 Kubernetes 集群。

kubebuilder

- 用于快速开发 Kubernetes 自定义控制器和 Operator 的框架，基于 controller-runtime 库，提供了一套工具和库，帮助开发者简化 Operator 的创建和管理过程，使得开发者能够专注于业务逻辑的实现，而不是底层的基础设施细节
- 安装
  - go install sigs.k8s.io/kubebuilder/cmd/kubebuilder@latest
- 初始化，确保执行所在目录空
  - kubebuilder init --domain easystack.io --repo repo.easystack.io/ecnf/hello
- crd 初始化，会得到文件 api/v1alpha1/hello_types.go
  - kubebuilder create api --group ecnf --version v1alpha1 --kind Hello
- web 初始化，会得到文件 api/v1alpha1/hello_webhook.go，需指定 mutate 还是 valid 类型
  - kubebuilder create api --group ecnf --version v1alpha1 --kind Hello
- 项目初始化之后基本已经完成得到如下信息，遵循 go 项目排版格式

```
├── Dockerfile
├── Makefile
├── PROJECT
├── README.md
├── cmd
│   └── main.go
├── config
│   ├── default/
│   ├── manager/
│   ├── prometheus/
│   └── rbac/
├── go.mod
├── go.sum
├── hack
│   └── boilerplate.go.txt
└── test
    ├── e2e/
    └── utils/
```

- 完成以上可以得到一个初始化的工程项目，当需要更新 CRD 的 spec 和 status 时，会通过向文件 api/v1alpha1/hello_types.go 做修改，但是修改后还需要同步到 config/default 目录的 crd.yaml 文件定义，以及修改还需要更新 zz_generated.deepcopy.go 文件
- 此时就需要另一个工具 controller-gen

controller-gen

- 工具，主要用于在 crd 数据结构发生变化时同步用
- 可以生成类型 webhook, rbac, object, crd
  - object: 生成DeepCopy接口，更新zz_generated.deepcopy.go
  - 其他均是更新 config/ 下的文件
- 通用配置
  - path 指定扫描路径，通常是 ./... 代表所有子目录
- 输出配置
  - `output:dir=<string>` 输出到指定目录
  - `output:none` 无输出
  - `output:stdout` 标准输出
- 常用方式
  - controller-gen rbac:roleName=manage crd output:crd:artifacts:config=config/crd/bases paths="./..."
  - controller-gen object path=./...
- go文件中支持写注解，并将注解转换为文件内容

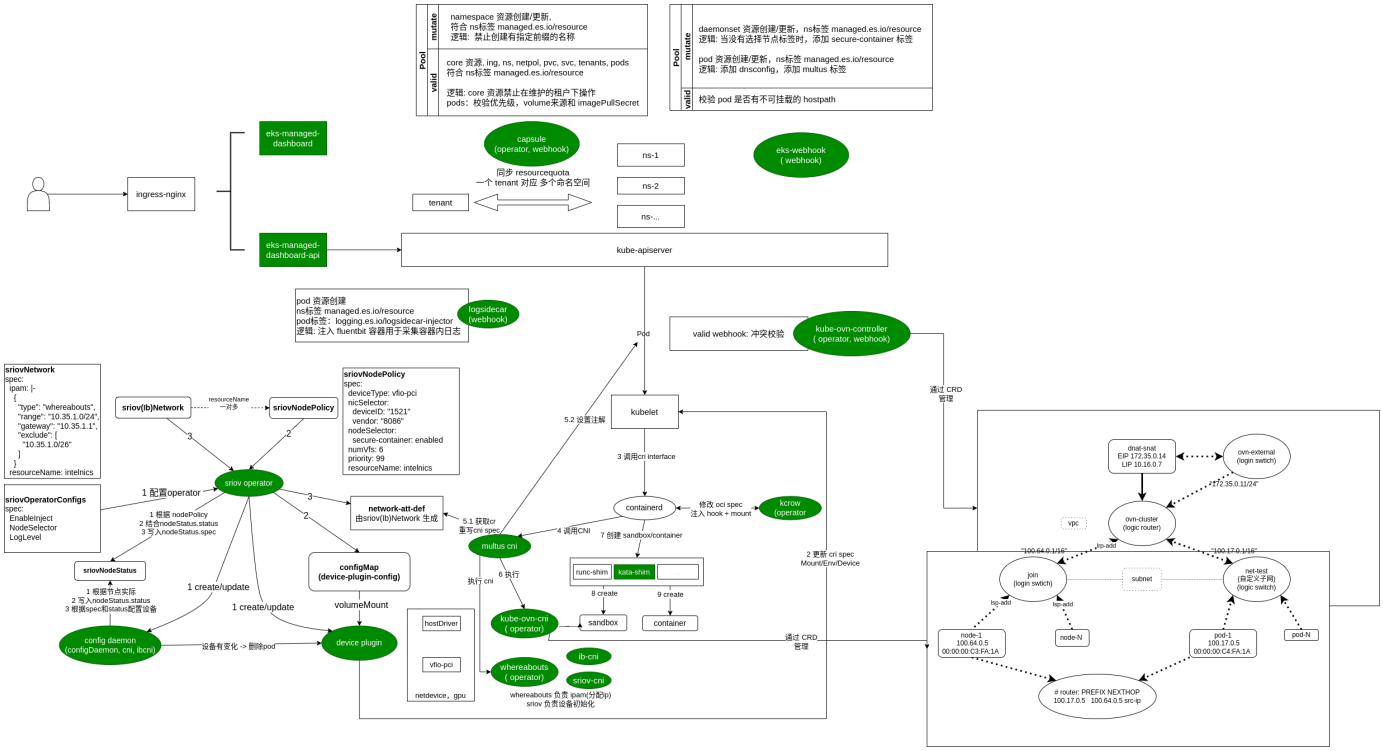
```
# api/v1alpha1/hello_types.go 文件
//+kubebuilder:object:root=true
//+kubebuilder:printcolumn:name="Type",type=string,JSONPath=`.spec.type`,des
cription="The type of cluster"  // kubectl get 时会将 spec.type 字段打印一列
type Hello struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   HelloSpec   `json:"spec,omitempty"`
    Status HelloStatus `json:"status,omitempty"`
}
```

安全容器服务

- chart：
  - ark-eks-managed
- 包含以下控制器

    - capsule-controller-manager-bb559b94f-l9x2c
    - cluster-manager-5dc47976f7-5cgnw
    - eks-managed-dashboard-6586c8bbcf-59p2v
    - eks-managed-dashboard-api-7d85dfd79d-blmgb
    - eks-webhook-66d68d8fbd-6mx9q
    - kcrow-4wzjf
    - kube-multus-ds-7p885
    - kube-ovn-cni-5wsvm
    - kube-ovn-controller-b8df6cf-bfzg4
    - logsidecar-injector-deploy-6c875d7b4c-jkmhp
    - ovn-webhook-647bdf9ddf-fbmxh
    - secure-container-config-containerd-rqj4v
    - secure-container-ecr-deploy-8bwk6
    - sriov-network-config-daemon-fpwmx
    - sriov-network-operator-cd8845d75-lpgk8
    - whereabouts-cncd4
- 包含以下 mutate webhook
    - capsule-mutating-webhook-configuration
    - eks-ds-mutator
    - eks-pod-mutator
    - logsidecar-injector-admission-mutate
- 包含以下 valid webhook
    - capsule-validating-webhook-configuration
    - eks-pod-validator
    - kube-ovn-webhook
- 架构



特点

- 使用较多 admission webhook 扩展，device plugin 扩展，nri 扩展
- operator 多，主要使用以下 crd
  - tenants.capsule.clastix.io/v1beta1
  - clusters.ecns.easystack.io/v1
  - network-attachment-definitions.k8s.cni.cncf.io/v1

sriov

- 网卡管理涉及的资源类型毕竟多，在图中列出主要资源有
- sriov(ib)Network: 本质上是 cni 配置文件，和 flannel.conflist 区别在于，配置是属于某类资源
- sriovNodePolicy：节点设备管理的信息，需要人工创建，指定如何切分设备，驱动等
- sriovOperatorConfigs：operator启动配置项，重要的是节点选择和资源前缀
  - NodeSelector：节点选择器，用于创建 config-daemon 和 device-plugin
  - Prefix：设置上报给 kubelet 的 vendor 信息
- sriovNodeStatus：节点信息，该资源被两种控制器读写
  - spec： 由 operator 写入，config-daemon 读取。主要同步机制是 config-daemon 将 status 信息向 spec 信息靠近，即操作节点设备，用于切分和设置驱动等
  - status：由 config-daemon 写入，operator 读取。config-daemon将当前设备信息写入，根据当前设备和预期会设置状态正在同步/成功/失败
- device-plugin-config：该资源是 configmap 类型，其中 key 是节点名称，value 是上报的资源信息。整体来说该资源是 config-daemon 创建和更新，一旦发生变化会重建 device-plugin 的pod，重建的目的是将最新的 device-plugin-config 挂载到 device-plugin 中，由其上报给kubelet

```
# sriovNetwork 例子
spec:
  ipam: |-
    {
      "type": "whereabouts",
      "range": "10.35.1.0/24",
      "gateway": "10.35.1.1",
      "exclude": [
        "10.35.1.0/26"
      ]
    }
  resourceName: intelnics #当pod请求这类资源时，会使用此 cni 配置

# sriovNodePolicy 例子
spec:
  deviceType: vfio-pci #设置切分后的驱动，默认 eth
  nicSelector:  # 设备选择器，指定厂商/设备ID
    deviceID: "1521"
    vendor: "8086"
  nodeSelector: # 节点选择器，先选节点再选设备
    secure-container: enabled
  numVfs: 6  # 切分后的 vf 数量，默认 0
  priority: 99 # 此策略的优先级，数字越小优先级越高
  resourceName: intelnics  #切分后上报给 kubelet 的资源名称，至于 vendorID 是
operaotr 的配置项

# device-plugin-config 例子
apiVersion: v1
kind: ConfigMap
metadata:
  name: device-plugin-config
data:
  node-1: '{"resourceList":[{"resourceName":"intelnics","selectors":
{"vendors":["8086"],"devices":["1520"],"drivers":["vfio-pci"],"rootDevices":
["0000:18:00.1"],"IsRdma":false,"NeedVhostNet":false}}]}'
```

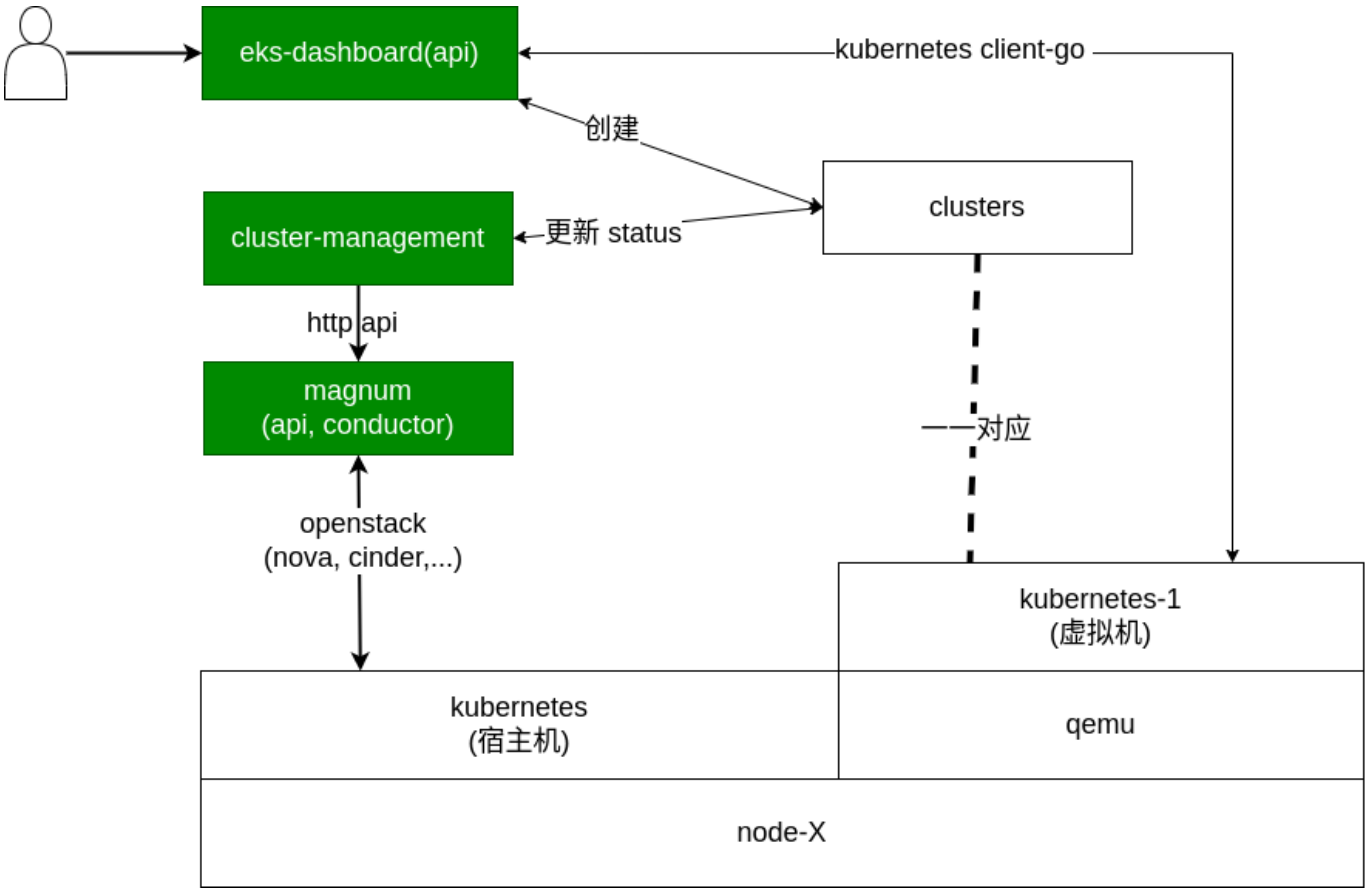
cni

- 环境中使用比较多的 cni，介绍部分 cni 流程
- multus 执行Add命名
  - pod是否存在注解`v1.multus-cni.io/default-network`，通过注解值查找 cr，并设置 Delegates[0] 为cr内容，否则使用主机的 cni 配置作为 Delegates[0]

    - 查询pod是否存在注解`k8s.v1.cni.cncf.io/networks`，通过注解值查找cr，追加到 Delegates 中，多网卡时需配置
    - cr 存在注解 `k8s.v1.cni.cncf.io/resourceName` 时，向 cni conf 注入该信息，主要用于 sriov 场景，sriov 使用此值确定设备地址
- sriov-cni 执行Add命令
    - 通过pci信息获取 vfid， pfname，ifname
    - 若ifname为找到，检查是否为dpdk方式
    - 设置 网卡 参数 vlan,mac,spoof,trust等
    - 非dpdk方式时，down网卡，切换netns，up网卡
    - 执行ipam，如果有的话
    - 信息写入文件 /var/lib/cni/sriov/{cid}-{ifname}

k8s容器服务

- chart：
    - ark-eks
- 包含以下控制器
    - cluster-management-bb559b94f-l9x2c
    - eks-dashboard-api-5dc47976f7-5cgnw
    - magnum-api-6586c8bbcf-59p2v
    - magnum-conductor-7d85dfd79d-blmgb
    - eks-dashboard-66d68d8fbd-6mx9q



- api 用于创建 clusters 资源，该资源被 cluster-manage 监听
- cluster-manage 调用 magnum api 管理集群生命周期

容器镜像服务

- chart
  - ark-container-registry
- 包含以下控制器
  - container-registry-dashboard-55c5f7dc48-zsdq9
  - container-registry-dashboard-api-7d9c9b6567-wzdtb
  - harbor-chartmuseum-0
  - harbor-core-689c9f76d4-ddcwp
  - harbor-database-0
  - harbor-jobservice-0
  - harbor-redis-0
  - harbor-registry-0

- 容器镜像服务围绕 harbor 进行，本身 harbor 是多租户模式

- dashboard-api 根据 token 中携带的用户信息，项目向 harbor 创建相应租户和密钥

容器应用中心

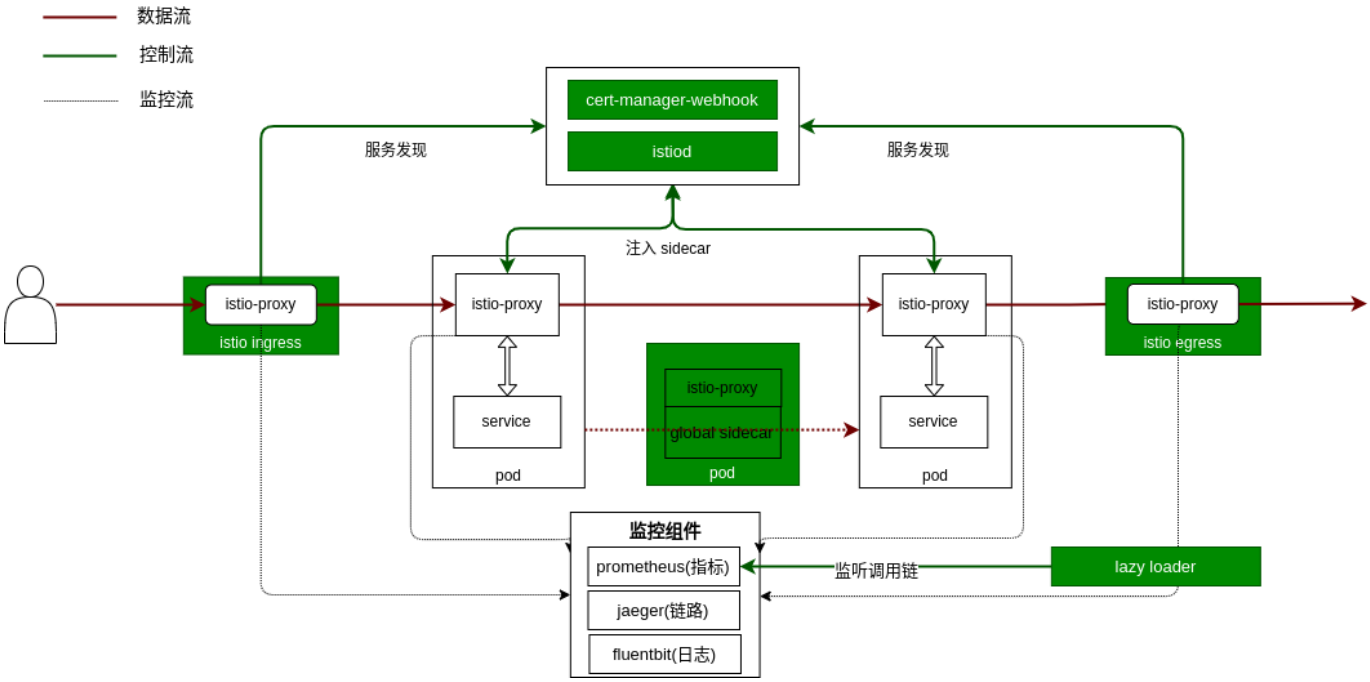
- chart
  - ark-ecns-appstore
- 包含以下控制器
  - appstore-dashboard-55c5f7dc48-zsdq9
  - appstore-dashboard-api-7d9c9b6567-wzdtb
- dashboard-api 是封装 helm v3 sdk ，支持以下功能
  - 使用容器镜像服务 chartmuseum 用于存储 chart 信息
  - 使用 helm v3 管理应用生命周期

服务网格

- chart
  - ark-servicemesh
- 包含以下控制器
  - cert-manager-b76ddbd87-vpl7w
  - cert-manager-cainjector-c79ff9b78-gqxgb
  - cert-manager-istio-csr-65df75cd74-42wl6
  - cert-manager-webhook-6d7b8b5b7c-mb6bb
  - dubbo-controller-556f9dc495-cxt2s
  - gateway-api-admission-server-5c5776568b-lmj7p
  - global-sidecar-7558dbf4ff-d6dv7
  - grafana-749d7948bd-wpdqk
  - istio-eastwestgateway-d98b79bbb-mhvp2
  - istio-egressgateway-5f64f99dbb-qdlbj
  - istio-ingressgateway-585b879444-r6n7f
  - istiod-6844555b4-n79lz
  - jaeger-0
  - kiali-675c7f84c7-97twc
  - lazyload-5f9d8fcd9c-chs6d
  - limiter-6c8995df8c-vqqfh
  - registryhub-6b44cdcf56-5hh2v
  - servicemesh-dashboard-66ccd6c9b8-798kz

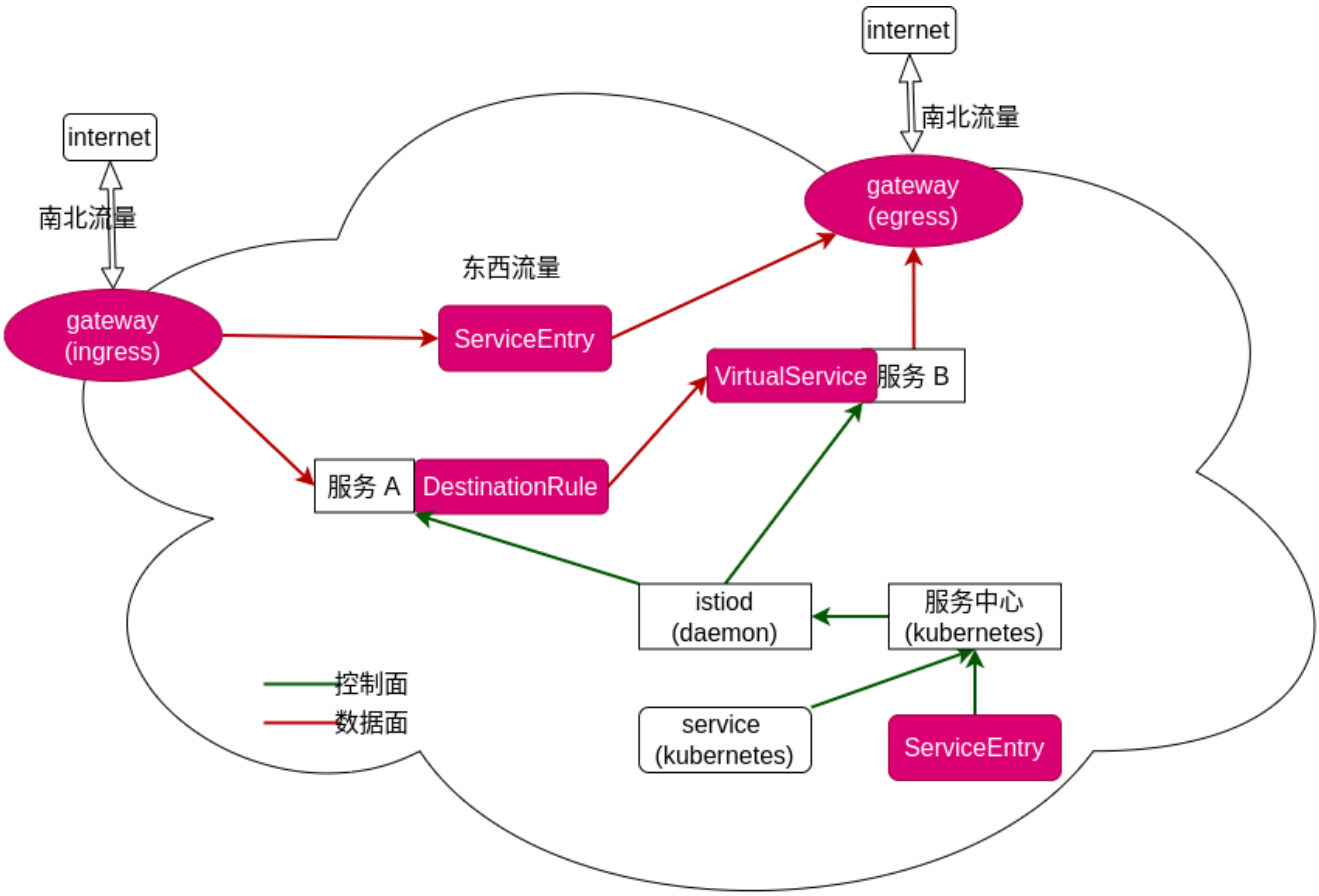
- servicemesh-dashboard-api-7875794dcf-btfmd
- slime-boot-69cddd58f-vxdsw

- 整体架构图， 主要流量包括数据面，控制面和监控面。其中数据面和监控面比较清晰，控制面流量比较复杂



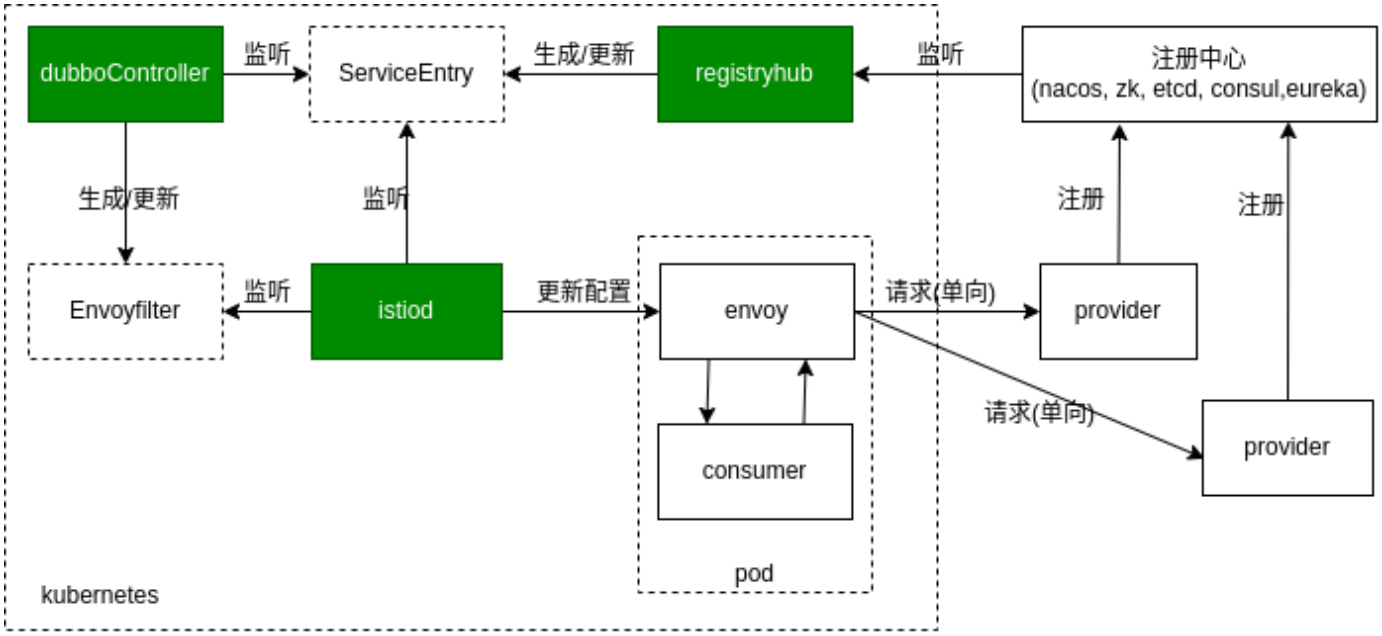
资源

- 主要涉及的 crd 资源
  - destinationrules.networking.istio.io/v1beta1 用于客户端，配置流量池和流量策略
  - gateways.networking.istio.io/v1beta1 选择某类pod作为边缘网关
  - serviceentries.networking.istio.io/v1beta1 配置和 kubernetse service 等价,可用于指定外部服务
  - sidecars.networking.istio.io/v1beta1 配置sidecar 可以使用的服务，如不能使用则是 L4
  - virtualservices.networking.istio.io/v1beta1 和ingress资源等价，用于配置路由策略
  - registryhubs.ecns.easystack.cn/v1alpha1 自研的crd，用于多注册中心
  - servicefences.microservice.slime.io/v1alpha1 懒加载的服务条目
  - smartlimiters.microservice.slime.io/v1alpha2 全局限流配置



多注册中心

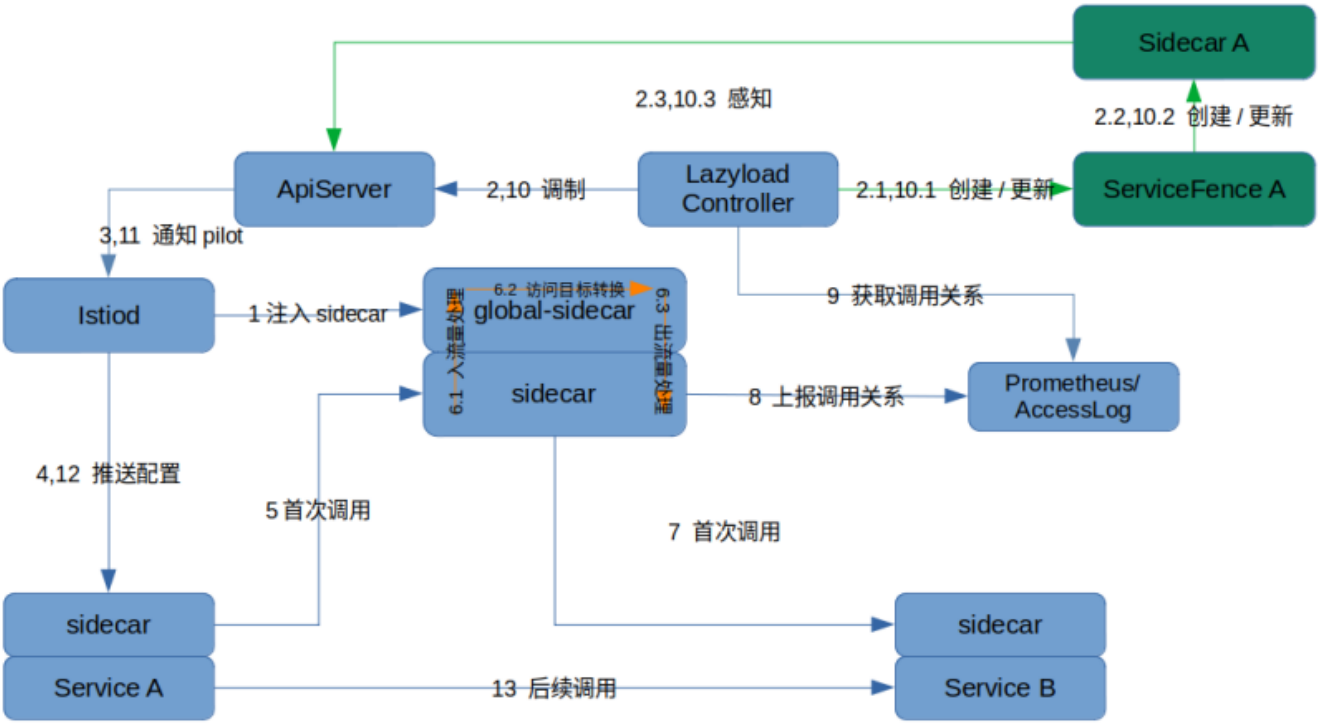
- istio 默认是从 k8s 读取 service，知道服务地址，但不止于 k8s，istio同样支持第三方的服务注册，如 zookpper，nacos，etcd 等



懒加载

- 懒加载目的是更新 sidecar 的服务条目，进而加快sidecar加载的速度
- 但懒加载还存在以下问题

- 使用 service 资源，并解析端口协议，这两项导致无法使用 serviceentry ，并且对k8s service 有一定要求
- 增加复杂度



ai基础设施

- chart
  - ark-easy-ai-infra

云原生云主机

- chart
  - ark-kubevirt