Dockerfile

Dockerfile 是一个用来构建镜像的文本文件，文本内容包含了一条条构建镜像所需的指令和说明。

语法规则

- **FROM**

  `FROM nginx`定制的镜像都是基于 FROM 的镜像，后续的操作都是基于这个nginx

- **RUN**

  `RUN`用于执行后面跟着的命令行命令。每执行一次都会在 **docker** 上新建一层。所以过多无意义的层，会造成镜像膨胀过大。以 **&&** 符号连接命令，这样执行后，只会创建 1 层镜像。

- **CMD**
- **WORKDIR**

  设置后续指令的工作目录。

-

Dockerfile 指令	说明
FROM	指定基础镜像，用于后续的指令构建。
MAINTAINER	指定Dockerfile的作者/维护者。（已弃用，推荐使用LABEL指令）
LABEL	添加镜像的元数据，使用键值对的形式。
RUN	在构建过程中在镜像中执行命令。
CMD	指定容器创建时的默认命令。（可以被覆盖）
ENTRYPOINT	设置容器创建时的主要命令。（不可被覆盖）
EXPOSE	声明容器运行时监听的特定网络端口。
ENV	在容器内部设置环境变量。
ADD	将文件、目录或远程URL复制到镜像中。
COPY	将文件或目录复制到镜像中。
VOLUME	为容器创建挂载点或声明卷。
WORKDIR	设置后续指令的工作目录。
USER	指定后续指令的用户上下文。
ARG	定义在构建过程中传递给构建器的变量，可使用 "docker build" 命令设置。
ONBUILD	当该镜像被用作另一个构建过程的基础时，添加触发器。
STOPSIGNAL	设置发送给容器以退出的系统调用信号。
HEALTHCHECK	定义周期性检查容器健康状态的命令。
SHELL	覆盖Docker中默认的shell，用于RUN、CMD和ENTRYPOINT指令。

多阶段构建

多个阶段逐步构建 Docker 镜像的方法，每个阶段都有自己的基础镜像和构建上下文。这种方法有助于减小最终镜像的大小，移除构建阶段中的不必要文件和依赖，使得最终的镜像更轻量化。

```
#第一阶段
FROM golang:1.17 as builder

WORKDIR /app

COPY . .

ENV GOPROXY=https://goproxy.cn,direct # 当出现timeout的时候可以加上代理

RUN CGO_ENABLED=0 go build -o ingress-manager main.go
```

```
#第二阶段
FROM alpine:3.15.3

WORKDIR /app

COPY --from=builder /app/ingress-manager .

CMD ["./ingress-manager"]
```

> */app/ingress-manager* 这是构建过程中唯一要保留的文件。

构建过程

1. 写dockefile文件

```
FROM nginx
RUN echo '这是一个本地构建的nginx镜像' > /usr/share/nginx/html/index.html
```

2. 本地构建

   在 Dockerfile 文件的存放目录下，执行构建动作。

   ```
   #通过目录下的 Dockerfile 构建一个 nginx:v3（镜像名称:镜像标签）
   $docker build -t nginx:v3 .
   ```

   ```
   #xiaox1958141/ingress-manager：镜像的仓库或用户命名空间。
   $ docker build -t xiaox1958141/ingress-manager:v1.0.0 .
   ```

   > . 是上下文路径。***docker build*** 命令得知这个路径后，会将路径下的所有内容打包。上下文路径下不要放无用的文件，因为会一起打包发送给 *docker* 引擎，如果文件过多会造成过程缓慢。

3. 上传仓库

   在上传仓库之前，先确保已经登录到 Docker Hub。

   ```
   docker login

   docker push xiaox1958141/ingress-manager:v1.0.0
   ```

4. 拉取

```
kubectl create deployment ingress-manager --image
xiaox1958141/ingress-manager:v1.0.0
```

5.

身份验证和授权

API Server授权管理

当API Server被调用时，需要先进行用户认证，然后通过授权策略执行用户授权。授权策略通过API Server启动参数`--authorization-mode`设置：

（1）AlwaysDeny：拒绝所有请求

（2）AlwaysAllow：允许接收所有请求

（3）**ABAC**（Attributed-Based Access Control）：基于属性的访问控制，表示使用用户配置的授权规则对用户请求进行匹配和控制

（4）**Webhook**：通过调用外部REST服务对用户进行授权

（5）**RBAC**：Role-Based Access Control，基于角色的访问控制

（6）**Node**：一种专用模式，用于对kubelet发起的请求进行访问控制

`sudo cat /etc/kubernetes/manifests/kube-apiserver.yaml`查看授权策略

如`--authorization-mode=RBAC,node`

RBAC授权模式

基于角色的访问控制

RBAC资源对象

RBAC有四个资源对象，分别是Role、ClusterRole、RoleBinding、ClusterRoleBinding

- **Role**

  一组权限的集合，在一个命名空间中，可以用其来定义一个角色，只能对命名空间内的资源进行授权。

```yaml
#定义一个角色用来读取Pod的权限,允许读取核心API组的Pod资源
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: pod-read
rules:
- apiGroups: [""]       #支持的API组列表
  resources: ["pods"]   #支持的资源对象列表
  resourceNames: []     #指定resource的名称
  verbs: ["get","watch","list"] #对资源对象的操作方法列表
```

查看 `kubectl get roles -n rbac`

- **ClusterRole**

  具有和角色一致的命名空间资源的管理能力，还可用于授权 集群级别的资源、非资源型的路径、包含全部命名空间的资源

```yaml
#定义一个集群角色可让用户访问任意secrets
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secrets-clusterrole
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get","watch","list"]
```

- **RoleBinding**

  把一个角色绑定在一个目标subjects上，可以是User，Group，Service Account，使用RoleBinding为某个命名空间授权

```yaml
#将在rbac命名空间中把pod-read角色授予用户es
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-bind
  namespace: rbac
subjects:
- kind: User
  name: es
  apiGroup: rbac.authorization.k8s.io
roleRef:
- kind: Role
  name: pod-read
  apiGroup: rbac.authorizatioin.k8s.io
```

```
# kube-system命名空间中名为default的Service Account
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

- **ClusterRoleBinding**

  把一个角色绑定在一个目标上，集群角色绑定的角色只能是集群角色，使用ClusterRoleBinding为集群范围内授权。

```
#允许manager组的用户读取所有namaspace的secrets
apiVersion: rabc.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secret-global
subjects:
- kind: Group
  name: manager
  apiGroup: rabc.authorization.k8s.io
ruleRef:
- kind: ClusterRole
  name: secret-read
  apiGroup: rabc.authorization.k8s.io
```

Service Account

> 用户账户: 除了 *Service Account*，*Kubernetes* 还支持用户账户，允许用户和其他实体进行身份认证。

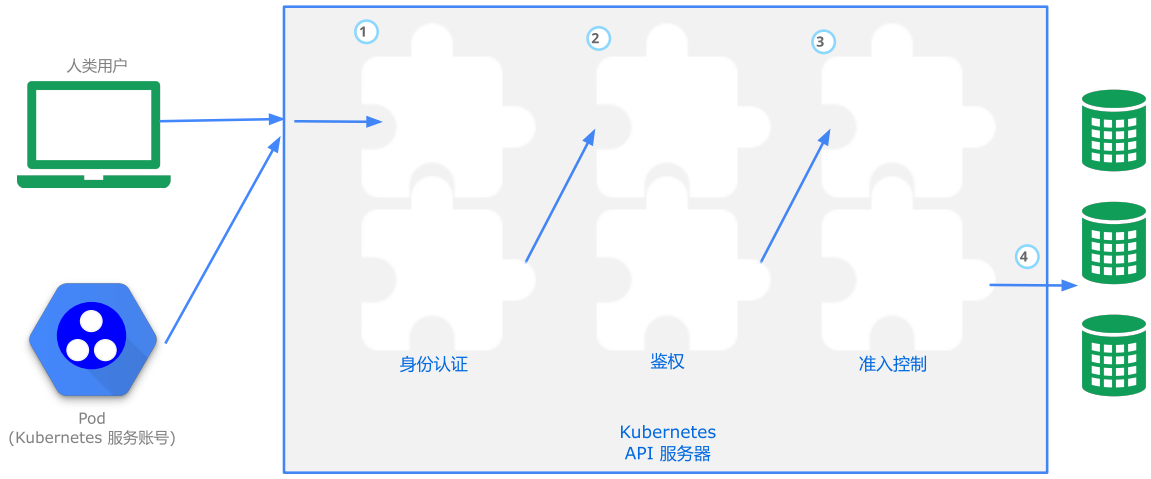
服务账号，也是一种账号。给运行在**Pod**里面的进程提供必要的身份证明。这个ServiceAccount就相当于是拥有某个角色的账号，也就拥有了某些权限。

在每个**Namespace**下都有一个名为**default**的默认**Service Account**对象，在这个ServiceAccount里面有一个名为Tokens的可以当做Volume被挂载到Pod里的Secret，当Pod启动时，这个Secret会自动被挂载到Pod的指定目录下，用来协助完成**Pod**中的进程访问**API Server**时的身份鉴权。

> ***pod***访问***API Server***时如何进行身份认证的？
>
> *https://kubernetes.io/zh-cn/docs/tasks/run-application/access-api-from-pod/*
>
> *https://kubernetes.io/zh-cn/docs/concepts/security/controlling-access/*

pod访问API Server服务时，在pod中是以service的方式访问名为kubenetes(https443端口)这个服务的，使用 TLS 进行加密通信。

```
aiedge@xx-test-master235:~$ kubectl get svc
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP
147d
```



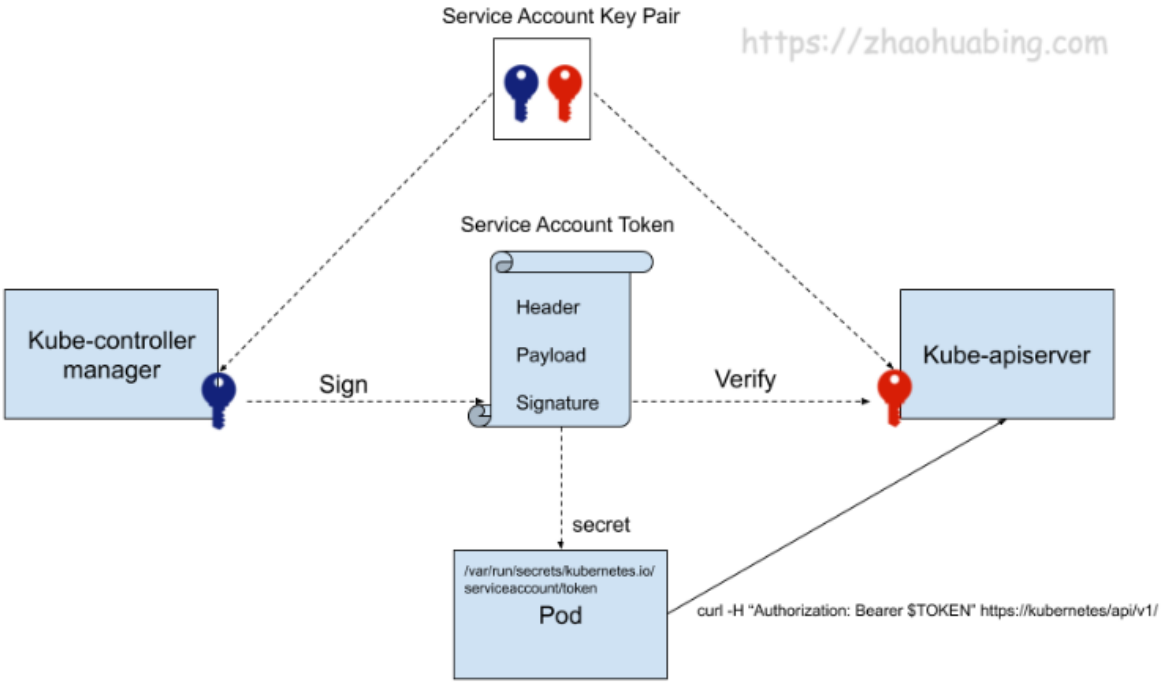
在pod中，cd /run/secrets/kubernates.io/serviceaccount/有三个文件：

- 在为一个 pod 指定了 service account 后，kubernetes 会为该 service account 生成一个 JWT token，并使用 secret 将该 service account token 加载到 pod 上（容器中`/var/run/secrets/kubernetes.io/serviceaccount/token`文件）。pod 中的应用可以使用 service account token 来访问 api server。

  token的生成：

  由Kubernetes Controller进程kube-controller-manager 用**API Server**的私钥（--service-account-private-zkey-file指定的私钥）签名指定生成的一个**JWT Secret**。
- 通过HTTPS方式与API Server建立连接后，会用Pod里指定路径下的一个**CA**证书（容器中`/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`验证**API Server**发来的证书，验证是否为CA证书签名的合法证书。
- **Pod**在调用**API Server**时，在**Http Header**中传递了一个**Token**字符串。API Server收到Token后，采用自身私钥（service-accountkey-file指定，如果没有指定，则默认采用tls-private-key-file指定的参数）对**Token**进行合法性验证

- 命名空间域 *API* 操作的默认命名空间放置每个容器中的 */var/run/secrets/kubernetes.io/serviceaccount/namespace* 文件
- 鉴权：一旦 *Token* 被验证为有效，*API Server* 将使用其中的用户信息进行 *RBAC* 鉴权，以确定用户是否有执行请求操作的权限



公钥加密，私钥解密。
私钥数字签名，公钥验证。

密钥对：*sa.key sa.pub* 根证书：*ca.crt etcd/ca.crt* 私钥:*ca.key* 等

*service Account*密钥对 *sa.key sa.pub*
提供给 *kube-controller-manager*使用，*kube-controller-manager*通过 *sa.key* 对 *token* 进行签名，
master 节点通过公钥 *sa.pub* 进行签名的验证 如 *kube-proxy* 是以 *pod* 形式运行的，在 *pod* 中，
直接使用 *service account* 与 *kube-apiserver* 进行认证，此时就不需要再单独为 *kube-proxy* 创建证书了，
会直接使用*token*校验。

使用

1. 创建ServiceAccount对象

```
kubectl create serviceaccount default:ingress-manager-sa
```

2. 使用鉴权机制（如RBAC）为ServiceAccount对象授权

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  creationTimestamp: null
  name: ingress-manager-role
rules:
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - watch
- apiGroups:
  - networking.k8s.io
  resources:
  - ingresses
  verbs:
  - list
  - watch
  - create
  - update
  - delete
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: null
  name: ingress-manager-rb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: ingress-manager-role
subjects:
- kind: ServiceAccount
  name: ingress-manager-sa
  namespace: default
```

3. 在创建 Pod 期间将 ServiceAccount 对象指派给 Pod。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: ingress-manager
  name: ingress-manager
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: ingress-manager
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: ingress-manager
    spec:
      serviceAccountName: ingress-manager-sa
      containers:
      - image: xiaox1958141/ingress-manager:v1.0.0
        name: ingress-manager
        resources: {}
status: {}
```

4.

核心概念

GVK & GVR

GVK = GroupVersionKind

GVR = GroupVersionResource

API Group 是相关 API 功能的集合，每个 Group 拥有一或多个 Versions，用于接口的演进。

每个 GV 都包含多个 API 类型，称为 **Kinds**，在不同的 Versions 之间同一个 Kind 定义可能不同， **Resource** 是 Kind 的对象标识（resource type）

根据 **GVK** K8s 就能找到你到底要创建什么类型的资源，根据你定义的 Spec 创建好资源之后就成为了 **Resource**，也就是 **GVR**。

每一个 GVK 都关联着一个 package 中给定的 root Go type，比如 apps/v1/Deployment 就关联着 K8s 源码里面 k8s.io/api/apps/v1 package 中的 Deployment struct，我们提交的各类资源定义 YAML 文件都需要写：

- apiVersion：这个就是 GV 。

- kind：这个就是 K。

```
常用资源 GVK：
core/v1/Pod
core/v1/Node
core/v1/Service
apps/v1/Deployment
```

通过 https://pkg.go.dev/k8s.io/api 可以搜索到资源详细信息（方法、字段等）

https://kubernetes.io/docs/reference/kubernetes-api/

HTTP Request

GET /api/v1/namespaces/{namespace}/pods/{name}

Scheme

每一组 Controllers 都需要一个 Scheme，提供了 Kinds 与对应 Go types 的映射，（也就是说给定 **Go type** 就知道他的 **GVK**，给定 **GVK** 就知道他的 **Go type**），APIServer 根据 Scheme来进行资源的序列化和反序列化。

api和apimachinery

api

https://github.com/kubernetes/api

主要功能：

- 内建资源对象定义
- 内建资源对象注册

apimachinery

https://github.com/kubernetes/apimachinery

主要存放服务端和客户端公用库，包含：

- ObjectMeta与TypeMeta
- **Scheme**：type对象注册、type对象与GVK的转换、版本转换方法注册
- RESTMapper：GVK与GVR转换
- 编码与解码
-
- 版本转换
- ...

Owners and Dependents

在 Kubernetes 中，一些对象是其他对象的“属主（Owner）”。 例如，ReplicaSet 是一组 Pod 的属主。 具有属主的对象是属主的“附属（Dependent）”。

附属对象有一个 `metadata.ownerReferences` 字段，用于引用其属主对象。一个有效的属主引用， 包含与附属对象同在一个命名空间下的对象名称和一个 UID。

附属对象还有一个 `ownerReferences.blockOwnerDeletion` 字段，该字段使用布尔值，用于控制特定的附属对象是否可以阻止垃圾收集删除其属主对象。（即在删除引用之前无法删除所有者）

```
aiedge@xx-test-master235:~/project/distributed-inference/pipeline-operator$ kubectl get deployment -n pipeline facedetection-1
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2024-01-12T08:36:28Z"
  generation: 1
  labels:
    pipeline: facedetection-pipeline
    step: "1"
  name: facedetection-1
  namespace: pipeline
  ownerReferences:
  - apiVersion: distri-infer.ndsl.cn/v1beta1
    blockOwnerDeletion: true
    controller: true
    kind: Pipeline
    name: facedetection-pipeline
    uid: 4b132047-1a4a-40f0-9532-45e5b696919e
  resourceVersion: "11945004"
  uid: 2d4e6c64-6196-4596-ad55-87a065ba3bab
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      pipeline: facedetection-pipeline
      step: "1"
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
```

根据设计，kubernetes 不允许跨名字空间指定属主。 名字空间范围的附属可以指定集群范围的或者名字空间范围的属主。 名字空间范围的属主必须和该附属处于相同的名字空间。

> 上述即
>
> *// 验证owner和object的namespace，有两种情况会产生异常：*
>
> *// 1.owner的namespace不为空，object的namespace为空*
>
> *// 2.owner的namespace不为空，object的namespace不等于owner的namespace*

> 如果垃圾收集器检测到无效的跨名字空间的属主引用， 或者一个集群范围的附属指定了一个名字空间范围类型的属主， 那么它就会报告一个警告事件。该事件的原因是 *`OwnerRefInvalidNamespace`*， *`involvedObject`* 属性中包含无效的附属。 你可以运行 *`kubectl get events -A --field-selector=reason=OwnerRefInvalidNamespace`* 来获取该类型的事件。

资源回收

Deployment, Job, DaemonSet等资源, 在删除时候, 其相关的Pod都会被删除, 这种机制就是kubernetes的垃圾收集器.

但是Kubernetes的垃圾收集器仅能删除Kubernetes API的资源。Kubernetes对于删除级联资源提供了2种模式：

- Background:在这模式下，Kubernetes会直接删除属主资源，然后再由垃圾收集器在后台删除相关的API资源
- Foreground:在这模式下，Owner资源会透过设定`metadta.deletionTimestamp`字段来表示"正在删除中"。这时Owner资源依然存在于集群中，并且能透过REST API查看到相关信息。该资源被删除条件是当移除了`metadata.finalizers`字段后，才会真正的从集群中移除。这样机制形成了预删除挂钩（Pre-delete hook），因此我们能在正在删除的期间，开始回收相关的资源（如虚拟机或其他Kubernetes API资源等等），当回收完后，再将该资源删除。

Finalizer

可以在`controller.go`里面加入Finalizers机制来确保资源被正确删除, 做法也比较简单, 只需要在对应的创建函数中对其对应的资源设置`metadata.finalizers`即可

Client-go

调用kubernetes集群资源对象API的客户端。 client-go版本和k8s版本最好对应

类型

- **RESTClient**：最基础客户端
- **Clientset**：包含所有**k8s**内置资源的client。在RESTClient的基础上封装了对Resource和Version的管理方法
- **dynamicClient**：可操作任意k8s资源，包括**CRD**自定义资源。返回的对象是一个`map[string]interface{}`
- **DiscoveryClient**：用于发现k8s提供的资源组、资源版本、资源信息

使用

RESTClient

1. 读取集群配置文件**kubeconfig**文件并实例化**config**对象。 读取时是调用tools/clientcmd/api/loader.go中的Load函数读取配置文件。

```
config, err := clientcmd.BuildConfigFromFlags("",
"/root/.kube/config")  //第一个参数是apiserver地址，可省略
//第二个参数可以是 clientcmd.RecommendedHomeFile
//第二个参数可以是""空，两个参数都为空时从默认的集群配置文件kubeconfig的路径获取

// 因为pod的group为空，version为v1
config.GroupVersion = &v1.SchemeGroupVersion
// 设置反序列化
config.NegotiatedSerializer = scheme.Codecs
// 指定ApiPath,参考/api/v1/namespaces/{namespace}/pods
config.APIPath = "/api"
```

> 若在集群内（*eg*集群内的*pod*）运行时，*client-go*在调用*“rest.InClusterConfig() ”*将使用*pod*内 *“/var/run/secrets/kubernetes.io/serviceaccount”*目录下的*token*文件做集群内的访问认证
>
> *config, err :=rest.InClusterConfig()*

2. 获取client

```
restClient, er := rest.RESTClientFor(config)
```

3. 执行操作

```
pod:= v1.Pod{}
restClient.Get().Namespace("default").Resource("pods").Name("mynginx
").Do(context.TODO()).Into(&pod)
```

> 打断点发现请求的*url*：
> *192.168.20.235:6443/api/v1/namespaces/default/pods/mynginx-7968d6dcd5-gsjwk*

clientset

```
//config
config, err := clientcmd.BuildConfigFromFlags("",
clientcmd.RecommendedHomeFile)

//client
clientset, err := kubernetes.NewForConfig(config) //config

//get data
pod, err := clientset.CoreV1().Pods("default").Get(context.TODO(),
"mynginx-7968d6dcd5-gsjwk", v1.GetOptions{})
```

常用方法函数

1. **获取key的方法**

   `cache.MetaNamespaceKeyFunc(obj)`

2. **获取OwnerReference**

   调用`metav1.GetControllerOf`获取该pod对象的OwnerReference（对象之间的所有权关系），并判断该pod是否有上层controller。

   比如：

   ```
   // 获取 Pod 的 OwnerReference
   ownerReference := metav1.GetControllerOf(pod)
   if ownerReference != nil {
       fmt.Printf("Pod %s has owner: %s\n", pod.Name,
   ownerReference.Name)
       // 在这里可以判断上层是否是 Controller，比如 Deployment、StatefulSet
   等
       if isController(ownerReference.Kind) {
           fmt.Printf("Pod %s is controlled by %s\n", pod.Name,
   ownerReference.Kind)
       }
   } else {
       fmt.Printf("Pod %s has no owner\n", pod.Name)
   }
   ```

   头文件`metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"`

3. **从缓存中删除指定对象**

   `cache.DeletionHandlingMetaNamespaceKeyFunc()`

4. 分解key的方法，通过key获取namespace，name

cache.SplitMetaNamespaceKey(key)

```
//与 MetaNamespaceKeyFunc() 功能相反的是 SplitMetaNamespaceKey() 函数，
它将传入的 Key 分解，返回对象所在的命名空间和对象名称。
 namespace, name, err := cache.SplitMetaNamespaceKey(key)
 if err != nil {
  return true
 }
//使用namespace进行查找
service,err :=c.serviceLister.Services(namwspacekey).Get(name)
if err !=nil{
    return err
}
```

5. 周期性执行一个函数

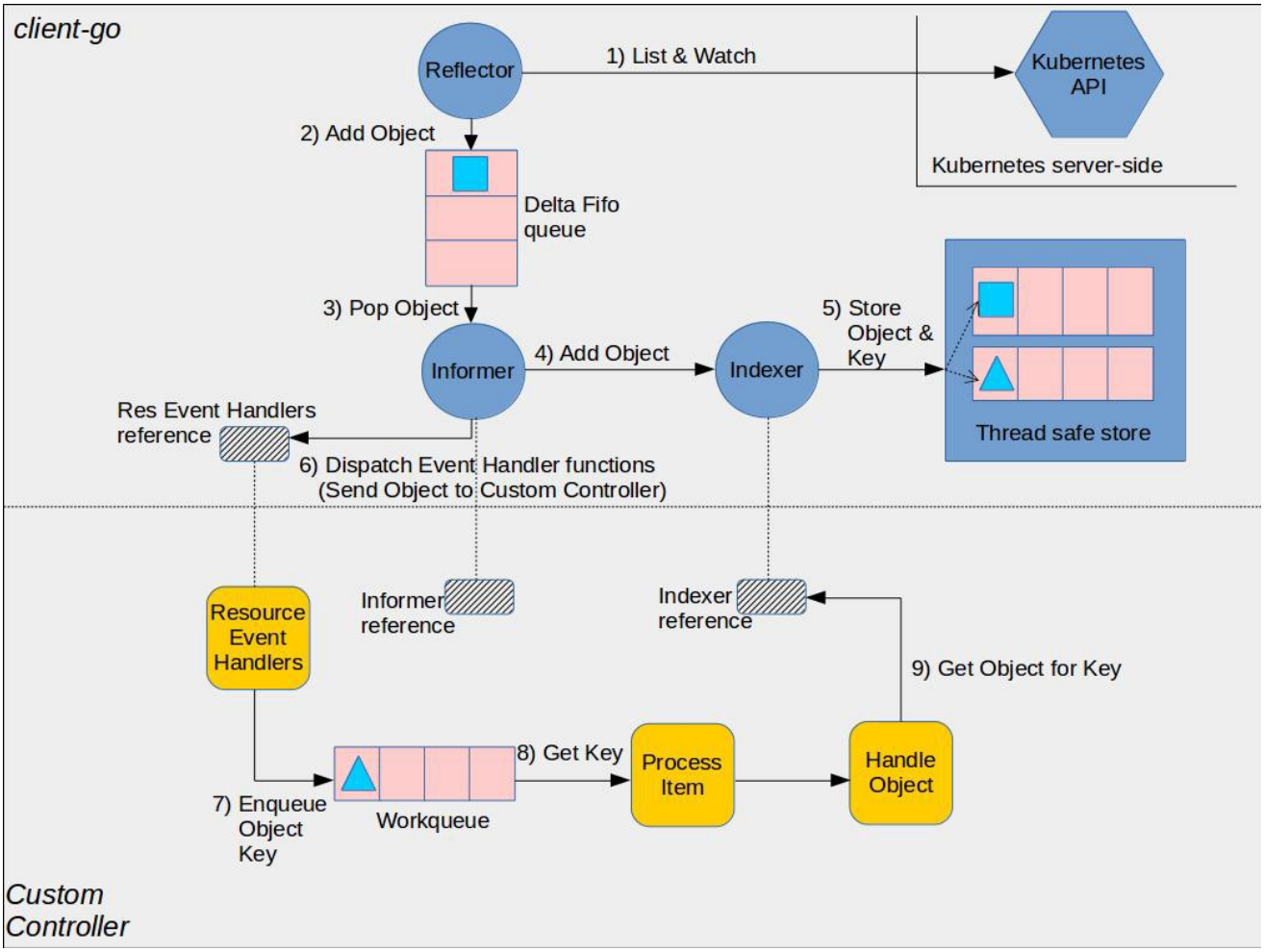
```
import "k8s.io/apimachinery/pkg/util/wait"
//每隔一秒钟输出当前的时间
wait.Forever(func() {
fmt.Println(time.Now().String())
}, time.Second)
```

```
//带StopSignal的周期性执行函数
var stopCh = make(chan struct{})
for i := 0; i < workNum; i++ { //至少有workNum个goroutine在执行
    go wait.Until(c.worker, time.Minute, stopCh) //每隔1分钟调用一次
worker

}
```

informer机制

K8s 在 client go 中提供了一个 informer 客户端库，用于监视 Kubernetes API 服务器中的资源并将它们的当前状态缓存到本地（而无需不断地向 **API** 服务器发出请求）。

HTTP 链接出问题后的重连，只需要list 变化的部分，而不是所有资源。



> 图中间的虚线将图分为上下两部分，其中上半部分是 ***Informer*** 库中的组件，下半部分则是使用 ***Informer*** 库编写的自定义 ***Controller*** 中的组件，这两部分一起组成了一个完整的 *Controller*。

Reflector：

使用**client-go(**或者 **K8s HTTP API) List/Watch API Server** 中指定的资源，然后使用 List 接口返回的 resourceVersion 来 watch 后续的资源变化

- 使用 List 的结果刷新 FIFO 队列
- 将 Watch 收到的事件加入到 FIFO 队列

Informer：

Informer 在一个循环中从 **FIFO** 队列中拿出资源对象进行处理。放到 Indexer 中、添加事件处理函数 ResourceEventHandler

Indexer：

将全量数据进行缓存并建立索引。

```
//map中的key是IndexFunc计算出来的结果，比如default；
//map中的value是所有obj的key的集合
type Index map[string]sets.String

//key是索引的分类名，比如namespace；
//value是一个方法，通过该方法可以获取obj的namespace，比如default
type Indexers map[string]IndexFunc

//key是索引的分类名，比如namespace；
type Indices map[string]Index
```

updateIndices：

缓存**cache**

Lister：

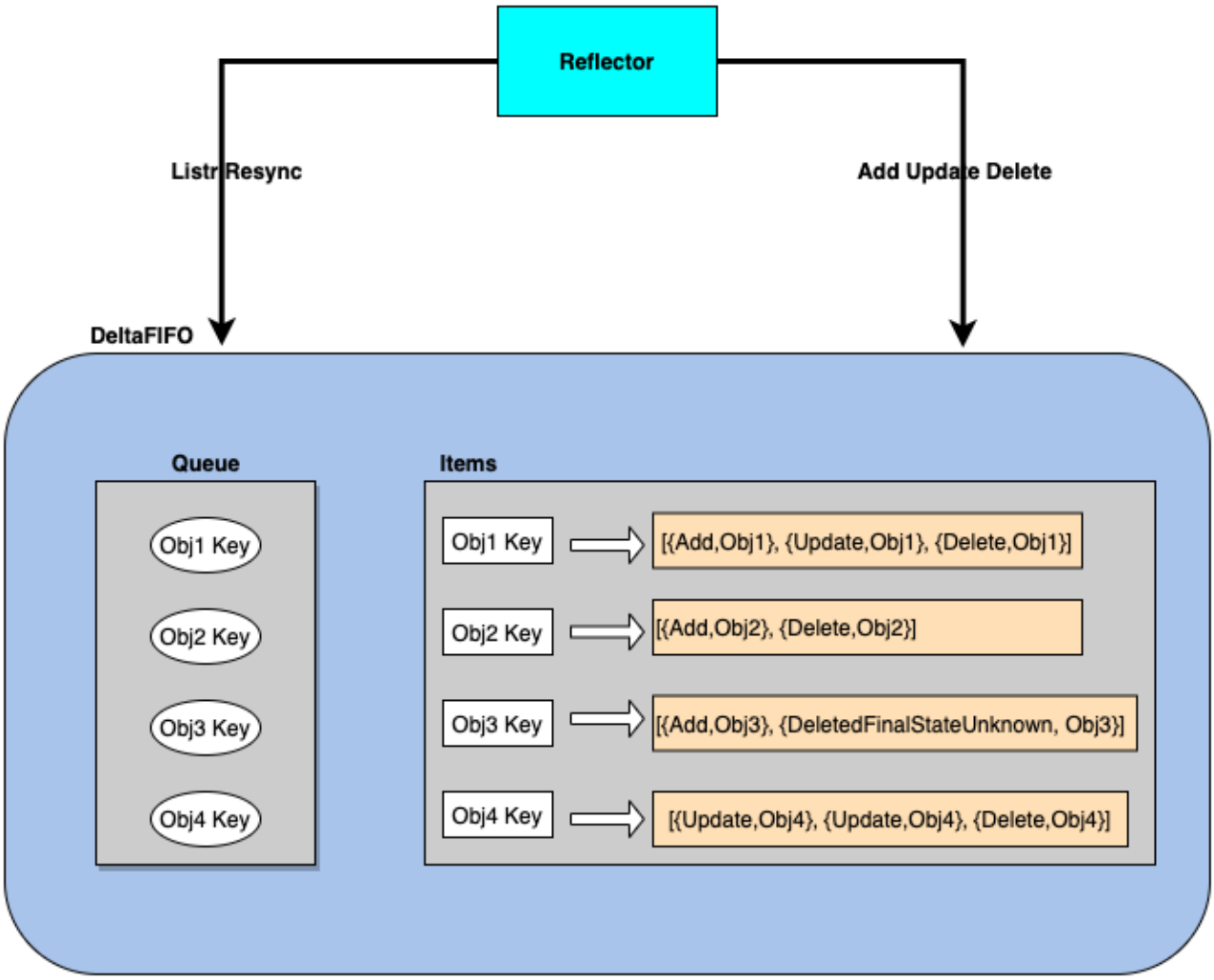
让应用程序从本地缓存中直接获取资源对象的列表

serviceInformer.Lister()

DeltaFIFO：

Reflector得到的数据放入DeltaFIFO，将数据

```
type DeltaFIFO struct {
  //...
  //存放Delta
  //与queue中存放的key是同样的key
    items map[string]Deltas
  //可以确保顺序性
    queue []string
  //...
  //默认使用MetaNamespaceKeyFunc，默认使用<namespace>/<name>的格式，不指定
namespace时用<name>
  //那么我们从队列的key里面也可以获取到重要的信息了
    keyFunc KeyFunc
  //其实就是Indexer
    knownObjects KeyListerGetter
  //...
}
```



SharedInformer （使用——ResourceEventHandler)

前面所说的组件都可以使用client-go中的SharedInformer来使用DeltaFIFO产生的数据，添加事件处理函数，把资源事件放入队列

> *K8s* 在 *client go* 中基于 *Informer* 之上再做了一层封装，提供了 *SharedInformer* 机制。采用 *SharedInformer* 后，客户端对同一种资源对象只会有一个对 *API Server* 的 *ListAndWatch* 调用，多个 *Informer* 也会共用同一份缓存，减少了对 *API Server* 的请求，提高了性能。

数据结构：

每个**`listener`**对应用户的**`EventHandler`**

```
type sharedIndexInformer struct {
   indexer    Indexer //informer中的底层缓存cache
   controller Controller //持有reflector和deltaFIFO对象，reflector对象将会
listWatch对象添加到deltaFIFO，同时更新indexer cahce，更新成功则通过
sharedProcessor触发用户配置的Eventhandler

   processor             *sharedProcessor //持有一系列的listener，每个
listener对应用户的EventHandler
```

```
    cacheMutationDetector MutationDetector //可以先忽略，这个对象可以用来监测
local cache是否被外部直接修改

    // This block is tracked to handle late initialization of the
controller
    listerWatcher ListerWatcher //deployment的listWatch方法
    objectType    runtime.Object

   ...

    started, stopped bool
    startedLock      sync.Mutex

    // blockDeltas gives a way to stop all event distribution so that a
late event handler can safely join the shared informer.
    blockDeltas sync.Mutex
}
```

SharedInformerFactory

SharedInformerFactory封装了NewSharedIndexInformer方法，构造informer的入口，里面包含了一堆Informer，指定了ListWatch函数。工厂模式来生成各类的Informer。同一种资源类型共用一个Infomer

以podInformer为例，看看一个具体的资源Informer需要实现哪些功能。

podInformer

其实就是 cache.SharedIndexInformer

listFunc: client.CoreV1().Pods(namespace).List(options)

WatchFunc: client.CoreV1().Pods(namespace).Watch(options)

```
// PodInformer provides access to a shared informer and lister for
// Pods.
// 只需要实现Informer，Lister函数
type PodInformer interface {
  Informer() cache.SharedIndexInformer
  Lister() v1.PodLister
}

type podInformer struct {
  factory          internalinterfaces.SharedInformerFactory    //  是哪一个
factory生成的informer
  tweakListOptions internalinterfaces.TweakListOptionsFunc    //  有哪些
filter
  namespace        string                                     //  命名空间
}
```

启动：

```
// 第一种方式
 go sharedInformer.Run(stopCh)
 time.Sleep(2 * time.Second)

// 第二种方式，这种方式是启动factory下面所有的informer
factory.Start(stopCh)
factory.WaitForCacheSync(stopCh)

//实际上，实现了 启动 Informers 开始监听资源变化 同步资源的当前状态 创建缓存
```

使用：

```
//get informer
factory := informers.NewSharedInformerFactory(clientset, 0)
//带有参数，限定namespace
//informers.NewSharedInformerFactoryWithOptions(clientset, 0,
informers.WithNamespace("default"))
informer := factory.Core().V1().Pods().Informer()

//add event handler
informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        fmt.Println("Add Event")
    },
    UpdateFunc: func(oldObj, newObj interface{}) {
        fmt.Println("Update Event")
    },
    DeleteFunc: func(obj interface{}) {
        fmt.Println("Delete Event")
    },
})

//start informer
stopCh := make(chan struct{})//用于向Informer发出停止信号
factory.Start(stopCh)
factory.WaitForCacheSync(stopCh)//等待Informer的缓存与K8s API Server 同步，
数据一致。
<-stopCh//阻塞主gorountine，直到在 stopCh 通道上接收到某个值。
```

结果：初始当前集群有多少个pod就输出多少个"Add Event"；当有其他事件发生对应触发，如添加pod "Add Event"，.....

workqueue：

对队列中的资源事件进行处理：读取**cache**里的**obj**，获取对应的**key**，做对应处理。是写Controller的主要部分

队列类型：

- 通用队列
- 延迟队列：处理需要在一定时间后执行的任务
- 限速队列：根据相应的算法获取元素的延迟时间，然后利用延迟队列来控制队列的速度。通常使用限速队列，其包含通用队列和延迟队列

```
rateLimitingQue :=
workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())
informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        fmt.Println("Add Event")
        key, err := cache.MetaNamespaceKeyFunc(obj) //获取对应的key
        if err != nil {
            fmt.Println(err)

        }
        rateLimitingQue.AddRateLimited(key) //把key放入限速队列
    },
    ...
}
```

> 当发生 *"Add Event"* 事件时，首先通过 *`cache.MetaNamespaceKeyFunc`* 获取对应的资源对象的 *key*。然后，将这个 *key* 放入限速队列 *`rateLimitingQue`* 中。通过将 *key* 放入限速队列，可以限制事件处理的速率，确保系统在单位时间内不会处理过多的事件，防止过载。

worker

不停从**workqueue**里获取**key**来处理。

处理过程：期望状态**spec**与当前状态**status**的对比。做调谐。

Kubernetes Controller 机制

Controller 原理——“是什么”

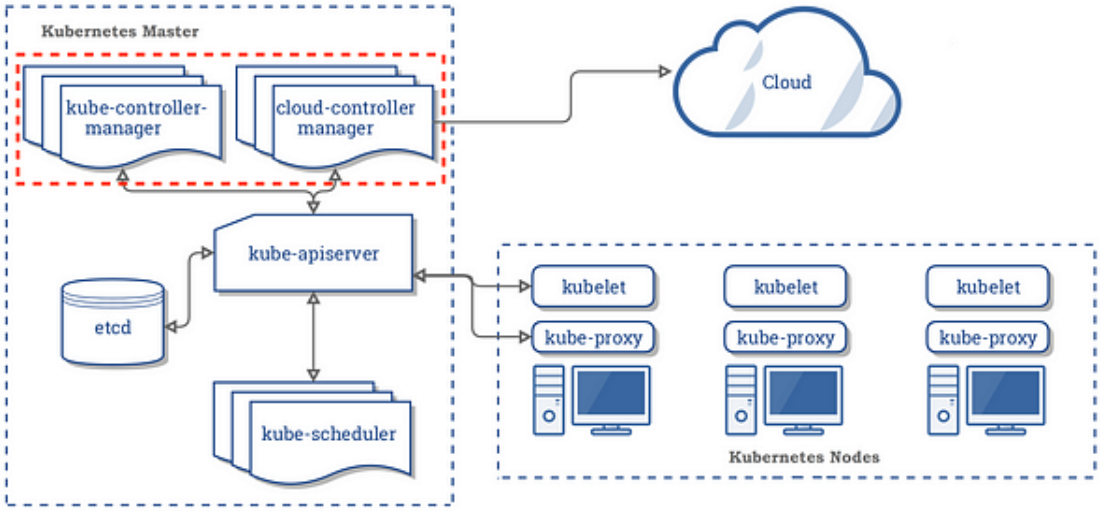
> *A Kubernetes controller is **a control loop** that **watches** the state of your cluster, then makes changes to move the **current cluster state** closer to **the desired state.***

在 K8s 中，用户通过声明式 **API** 定义资源的“预期状态”，Controller 则负责监视资源的实际状态，当资源的实际状态和“预期状态”不一致时，Controller 则对系统进行必要的更改，以确保两者一致，这个过程被称之为调谐（**Reconcile**）。

> 凡是遵循“*Watch K8s* 资源并根据资源变化进行调谐”模式的控制程序都可以叫做 *Controller*。而 ***Operator*** 是一种专用的 *Controller*，用于在 *Kubernetes* 中管理一些复杂的，有状态的应用程序。例如在 *Kubernetes* 中管理 *MySQL* 数据库的 *MySQL Operator*。
>
> 调谐循环（*Reconcile loop*）：实际是事件驱动+定时同步来实现，不是无脑循环

Kubernetes 内置 Controllers

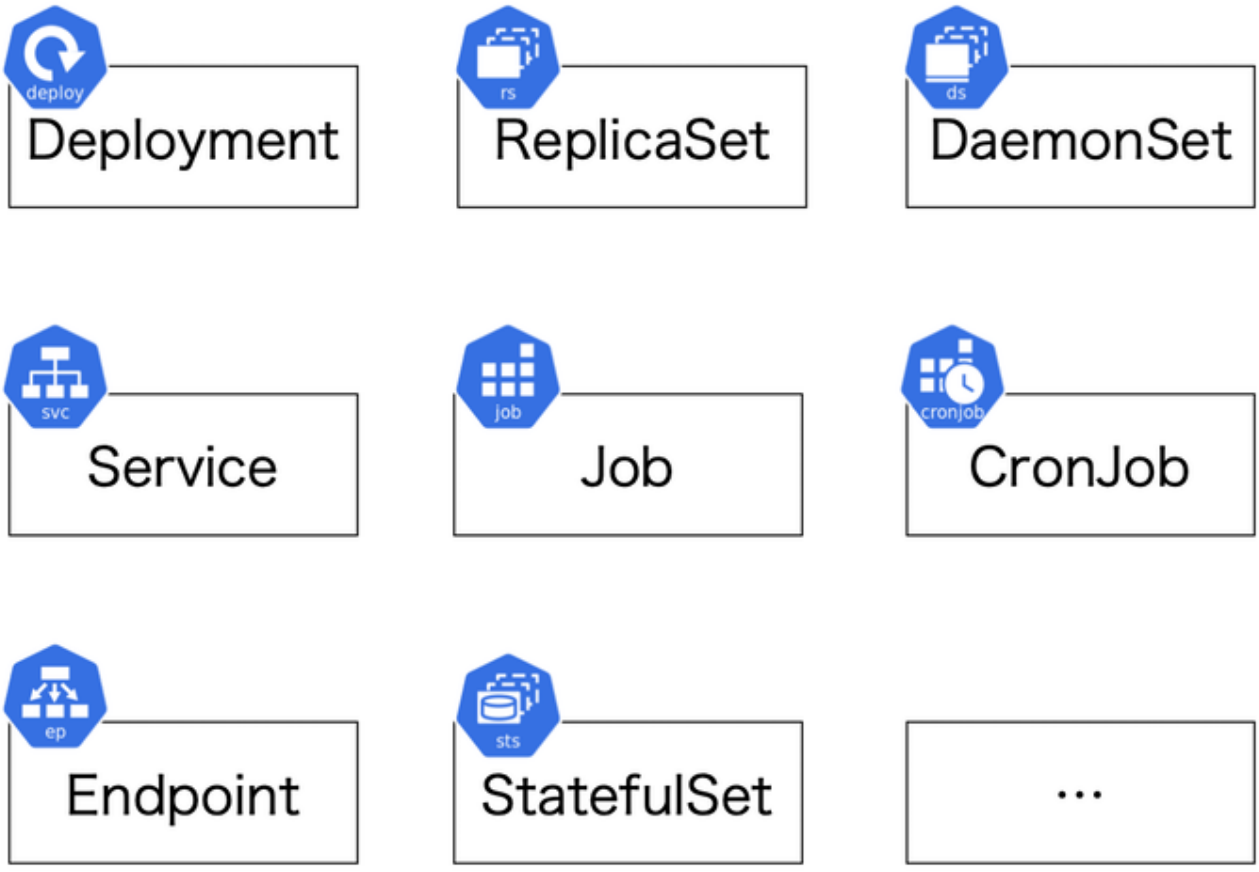
大量 Kubernetes Controllers 都存在于 `kube-controller-manager`和 `cloud-controller-manager`两个组件中，它们构成 Kubernetes controller manager，是 Kubernetes 的大脑，它通过 **API server** 监控整个集群的状态，并确保集群处于预期的工作状态。



kube-controller-manager

kube-controller-manager 中包含20多个 Controller，为降低复杂性，它们被编译到同一个二进制文件中。

每个 Controller 通过 **API server** 提供的接口，实时监控集群中每个资源对象的当前状态，并尝试将当前状态修复到“期望状态”。



cloud-controller-manager

cloud-controller-manager 用来配合云服务提供商的控制，也包括一系列的控制器，如

- Route Controller: 用于配置云平台中的路由
- Node Controller: 用于更新、维护、管理节点

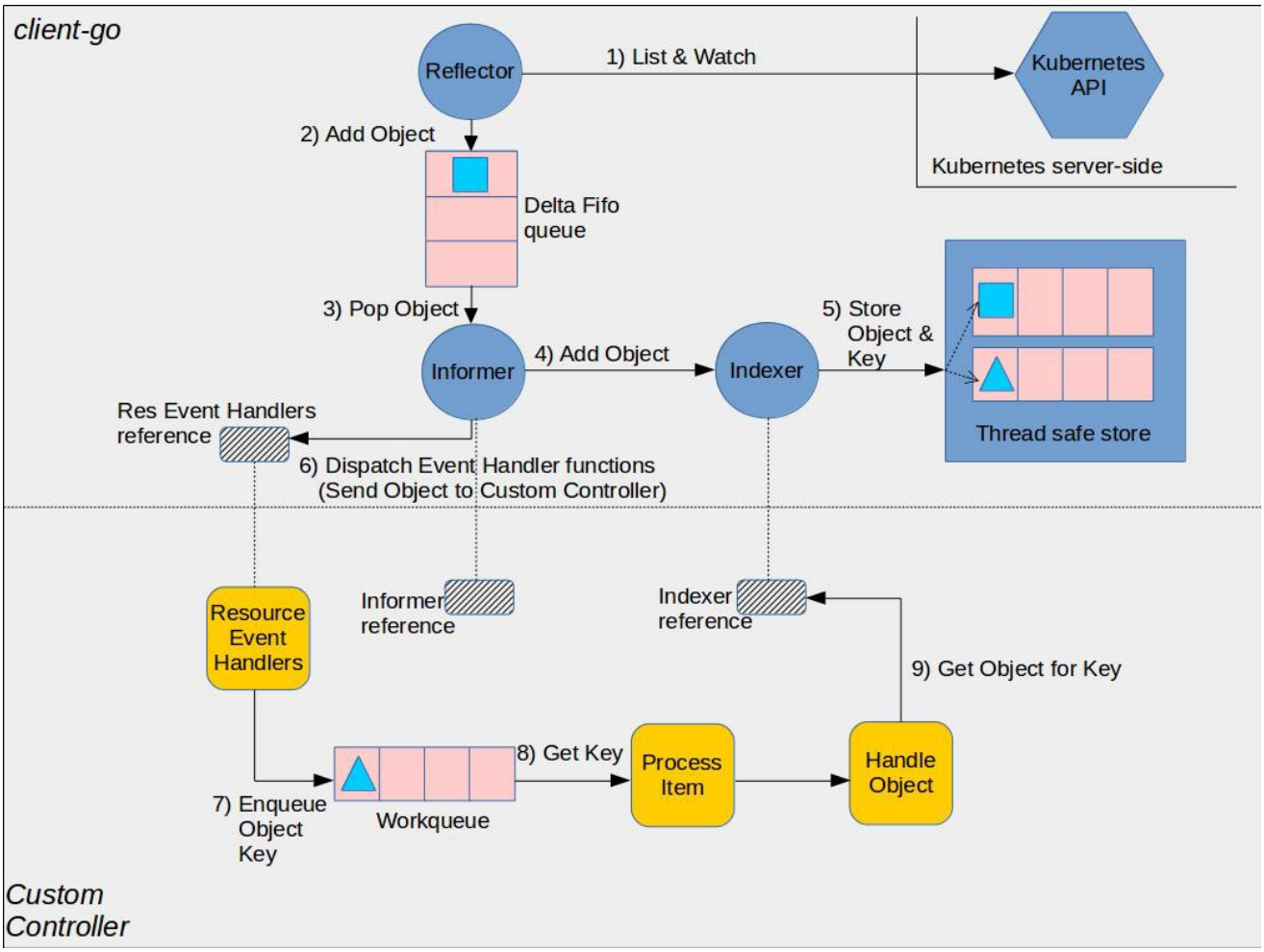
组件架构——Informer 机制

https://github.com/kevinlisr/baidingtech/blob/main/slide/

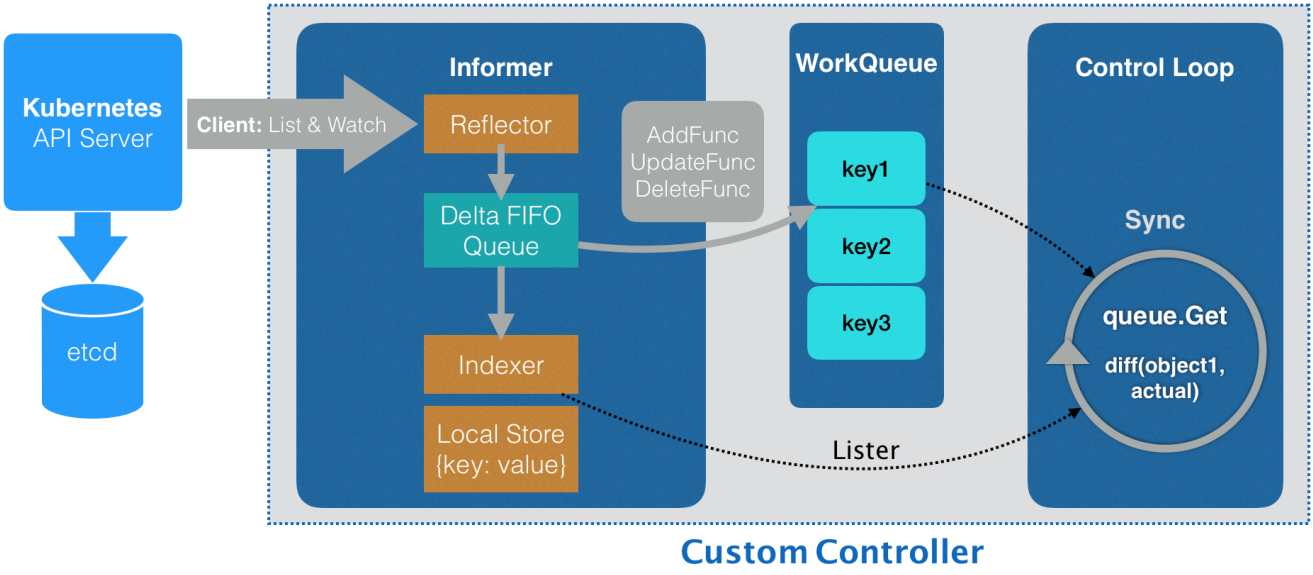
https://jimmysong.io/kubernetes-handbook/develop/client-go-informer-sourcecode-analyse.html

K8s 在 client go 中提供了一个 informer 客户端库，用于监视 Kubernetes API 服务器中的资源并将它们的当前状态缓存到本地（而无需不断地向 **API** 服务器发出请求）。

HTTP 链接出问题后的重连，只需要list 变化的部分，而不是所有资源。



> 图中间的虚线将图分为上下两部分，其中上半部分是 ***Informer*** 库中的组件，下半部分则是使用 ***Informer*** 库编写的自定义 ***Controller*** 中的组件，这两部分一起组成了一个完整的 *Controller*。



全手动实现Controller

只监听k8s内置资源：

```
type controller struct {
    client        kubernetes.Interface
    ingressLister netLister.IngressLister  //ingressInformer.Lister()
,List lists all Services in the indexer.
    serviceLister coreLister.ServiceLister
    queue         workqueue.RateLimitingInterface
}
```

通过EventHandler获取对应的事件obj，workqueue去获取key（`cache.MetaNamespacekeyFunc(obj)`），把key放入workqueue。然后work可以去inderx(cache)中读取对应资源

demo

```
235虚拟机上
aiedge@xx-test-master235:~/kubebuilder-demo/client-go-test$
```

自定义 CRD

Custom Resource (CR)：自定义资源，Kubernetes API 的扩展。

CRD：自定义资源定义。可以扩展 Kubernetes API

Custom Controller：自定义控制器，与 CR 共同使用实现声明式 API

Operator ：Operator =Custom Resource + Custom Controller Operator 名词来源于操作员，在 Kubernetes 中自动化完成一系列工作。例如：按需部署应用获取/还原应用状态的备份

> 链接
>
> *CRD*：*https://kubernetes.io/zh/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/*
>
> 更多 *Operator* 示例可参考： *https://operatorhub.io/*

注册自定义资源：

开发者需要通过K8S提供的方式注册自定义资源，即通过CRD进行注册，注册之后，K8S就知道我们自定义资源的存在了，然后我们就可以像使用K8S内置资源一样使用自定义资源（CR）

yaml定义

CRD 的结构中主要包含下列的内容：

- TypeMeta - CRD 的 Group，Version 和 Kind
- ObjectMeta - 标准的 k8s metadata 字段，包括 name 和 namespace
- Spec - CRD 中的自定义字段
- Status - Spec 对应的状态

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # 名字必需与下面的 spec 字段匹配，并且格式为 '<名称的复数形式>.<组名>'
  name: demos.example.com
spec:
  # 组名称，用于 REST API: /apis/<组>/<版本>
  group: example.com
  names:
    # 名称的复数形式，用于 URL：/apis/<组>/<版本>/<名称的复数形式>
    plural: demos
    # 名称的单数形式，作为命令行使用时和显示时的别名
    singular: demo
```

```
    # kind 通常是单数形式的帕斯卡编码（PascalCased）形式。你的资源清单会使用这一形式。
    kind: Demo
    # shortNames 允许你在命令行使用较短的字符串来匹配资源
    shortNames:
    - dm
    categories: #加入一个已有类别 kubectl get all可获取到
    - all
  # 可以是 Namespaced 或 Cluster
  scope: Namespaced
  # 列举此 CustomResourceDefinition 所支持的版本
  versions:
    - name: v1
      # 每个版本都可以通过 served 标志来独立启用或禁止
      served: true
      # 其中一个且只有一个版本必需被标记为存储版本
      storage: true
      schema:  # openAPIV3Schema用于合法性验证
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties: # 验证cr的name是否以test开头，否则在创建时会提示非法
                name:
                  type: string
                  pattern: '^test$'
      additionalPrinterColumns: # 附加字段，在get获取信息的时候可以打印出这些额外信息
      - name: CR-Name
        type: string
        jsonPath: .spec.name
```

demo——pipeline

```
---
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.12.0
   # 名字必需与下面的 spec 字段匹配，并且格式为 '<名称的复数形式>.<组名>'
  name: pipelines.distri-infer.ndsl.cn
spec:
  # 组名称，用于 REST API: /apis/<组>/<版本>
  group: distri-infer.ndsl.cn
  names:
```

```
    # kind 通常是单数形式的驼峰命名（CamelCased）形式。你的资源清单会使用这一形式。
    kind: Pipeline
    listKind: PipelineList
    # 名称的复数形式，用于 URL：/apis/<组>/<版本>/<名称的复数形式>
    plural: pipelines
     # 名称的单数形式，作为命令行使用时和显示时的别名
    singular: pipeline
  # 可以是 Namespaced 名字空间作用域的，或 Cluster集群作用域
  scope: Cluster
   # 列举此 CustomResourceDefinition 所支持的版本
  versions:
  # kubectl get打印的结果中 额外打印的列
  - additionalPrinterColumns:
    - jsonPath: .spec.steps[*].model
      name: Steps.model
      type: string
    - jsonPath: .status.phase
      name: Status.phase
      type: string
    - jsonPath: .status.detailPhase
      name: Status.DetailPhase
      type: string
    name: v1beta1
    schema: # openAPIV3Schema用于cr的合法性验证
      openAPIV3Schema:
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this representation
              of an object. Servers should convert recognized schemas to the latest
              internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources'
            type: string
          kind:
            description: 'Kind is a string value representing the REST resource this
              object represents. Servers may infer this from the endpoint the client
              submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
            type: string
          metadata:
            type: object
          spec:
```

```
            description: PipelineSpec defines the desired state of
Pipeline
            properties:
              modelStorage:
                description: storage of model documents
                properties:
                  csiParameter:
                    additionalProperties:
                      type: string
                    type: object
                  type:
                    type: string
                required: # 验证必需字段
                - csiParameter
                - type
                type: object
              steps:
                description: all steps of a pipeline
                items:
                  properties:
                    args:
                      additionalProperties:
                        type: string
                      type: object
                    image:
                      type: string
                    location:
                      type: string
                    model:
                      type: string
                  required:
                  - image
                  - location
                  - model
                  type: object
                type: array
            required:
            - modelStorage
            - steps
            type: object
          status:
            description: PipelineStatus defines the observed state of
Pipeline
            properties:
              detailPhase:
                properties:
                  pvPhase:
                    type: string
                  pvcPhase:
```

```
                    type: string
                  stepsPhase:
                    items:
                      properties:
                        deploymentPhase:
                          type: string
                      required:
                      - deploymentPhase
                      type: object
                    type: array
                type: object
              phase:
                description: 'INSERT ADDITIONAL STATUS FIELD - define
observed state
                  of cluster Important: Run "make" to regenerate code
after modifying
                  this file'
                type: string
            type: object
        type: object
    # 每个版本都可以通过 served 标志来独立启用或禁止
    served: true
    # 其中一个且只有一个版本必需被标记为存储版本 存到etcd
    storage: true
    subresources:
      status: {}
```

之后创建它：

```
kubectl apply -f resourcedefinition.yaml
```

这样一个新的集群空间约束的 RESTful API 端点会被创建在：

```
/apis/distri-infer.ndsl.cn/v1/pipelines/...
```

> 如果是集群空间约束的 *RESTful API* 端点会被创建在：
>
> *`/apis/<group>/<version>/<crd name>/...`*
>
> 如果是命名空间约束的 *RESTful API* 端点会被创建在：
>
> *`/apis/<group>/<version>/namespaces/<namespace>/<crd name>`*

特殊字段说明

Schema

模板yaml中的Schema字段：模式合法性验证

schema的生成可以通过代码生成器kubebuilder中的crd-gen工具中的crd-schema-gen。

子资源

CRD仅支持status和scale子资源

```
...
schema:
...
subresources:
    # status 启用 status 子资源
    status: {}
    # scale 启用 scale 子资源
    scale:
        # specReplicasPath 定义定制资源中对应 scale.spec.replicas 的 JSON 路
径
        specReplicasPath: .spec.replicas
     ...
```

多版本

```
...
  versions:
  ...
  conversion:
    strategy: Webhook
    webhook:
      conversionReviewVersions: ["v1","v1beta1"]
      clientConfig:
        service:
          namespace: default
          name: example-conversion-webhook-server
          path: /crdconvert
        caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
```

创建cr

像内置资源比如Pod一样声明资源，使用CR声明我们的资源信息

```
apiVersion: distri-infer.ndsl.cn/v1beta1
kind: Pipeline
```

```yaml
metadata:
  name: facedetection-pipeline
spec:
  # TODO(user): Add fields here
  steps:
    - image: nginx:latest
      location: xx-test-node236
      model: facedetection-1
      args:
        listenPort: "9080"
    - image: nginx:latest
      location: xx-test-node239
      model: facedetection-2
      args: {}
  modelStorage:
    csiParameter:
      server: 192.168.20.235
      share: /home/aiedge/csiTest
    type: nfs
status:
  phase: failed
```

Finalizers

Finalizer 能够让控制器实现异步的删除前（**Pre-delete**）回调 （做一些删除相关的工作）。 与内置对象类似，定制对象也支持 Finalizer

```yaml
apiVersion: "example.com/v1"
kind: Demo
metadata:
  name: crd-demo
  finalizers:
  - example.com/finalizer
spec:
  name: test
```

一些代码生成工具

自动代码生成工具将controller之外的事情都做好了，我们只要专注于controller的开发就好

code-generator

K8S官方提供的一组代码生成工具，主要用于产生 kubernetes-style API types的Client, Deep-copy, Informer, Lister等功能的程序.

它主要有两个应用场景：

- 为CRD编写自定义controller时，可以使用它来生成我们需要的`versioned client`、`informer`、`lister`以及其他工具方法
- 编写自定义API Server时，可以用它来 `internal` 和 `versioned`类型的转换 `defaulters`、`internal` 和 `versioned`的`clients`和`informers`

使用

- git拉取`https://github.com/kubernetes/code-generator.git`，切换到`git checkout v0.23.3`
- 安装

  `go install code-generator/cmd/{client-gen,lister-gen,informer-gen,deepcopy-gen}`
- `go env GOPATH`查看路径，在此路径下有安装的工具
- 使用generate-groups.sh

  `code-generator/generate-groups.sh deepcopy,client,informer MOD_NAME/pkg/generated MOD_NAME/pkg/apis foo.example.com:v1 --output-base MOD_DIR/..  --go-header-file "code-generator/hack/boilerplate.go.txt"`

标记

```
 // +genclient
 // +genclient:noStatus
 //cluster级别的
 // +genclient:nonNamespaced
 // +genclient:noVerbs
 // +genclient:onlyVerbs=create,delete
 //
+genclient:skipVerbs=get,list,create,update,patch,delete,deleteCollectio
n,watch
 //
+genclient:method=Create,verb=create,result=k8s.io/apimachinery/pkg/apis
/meta/v1.Status
```

包级别

```
// +k8s:deepcopy-gen=package
// +groupName=foo.example.com
package v1
```

controller-tools

安装

1. 获取controller-tools的代码，并切换到v0.8.0 的tag上

   ```
   git checkout v0.8.0
   ```

2. 编译项目，安装代码生成工具，这里我们只安装我们接下来会用到的工具

   ```
   go install ./cmd/{controller-gen,type-scaffold}
   ```

使用type-scaffold

用于快速生成 **Kubernetes** 资源类型**CRD**的结构，包括根类型、Spec 和 Status 类型以及列表类型。

```
type-scaffold --kind Foo
type-scaffold --help
```

在生成types.go后，可以在里面的结构中写想要的字段，然后重新使用`controller-gen crd paths=./... output:crd:dir=config/crds`生成crd.yaml

使用controller-gen

用于生成 **Kubernetes** 控制器代码的工具，

```
#生成 CustomResourceDefinition（CRD）的 YAML 文件
controller-gen crd paths=./... output:crd:dir=config/crds
#生成针对 Kubernetes 对象的深拷贝（deepcopy）函数和客户端代码
controller-gen object paths=./...
#生成 RBAC 规则
controller-gen rbac paths=./...
```

使用代码生成工具来编写自定义的CRD的controller

1.编写CRD.yaml

2.项目框架

```
pkg
├── apis
│   └── appcontroller # 提供该 Package 的 API Group Name。
│       ├── register.go
│       └── v1alpha1 # API 各版本结构定义。Kubernetes API 是支援多版本的。
│           ├── doc.go
│           ├── register.go
│           ├── types.go
```

3. apis/group/version文件夹下文件

v1alpha1/doc.go 文件

使用一个全局 tag `+k8s:deepcopy-gen=package`，来为整个 package 中的所有数据结构生成 DeepCopy 方法。

```
// +k8s:deepcopy-gen=package
// +groupName=samplecontroller.k8s.io

// Package v1alpha1 is the v1alpha1 version of the API.
package v1alpha1
```

v1alpha1/types.go 文件

定义CRD中资源的结构, 以及定义code-generator的local tags.

```
/* source code from https://github.com/kubernetes/sample-
controller/blob/master/pkg/apis/samplecontroller/v1alpha1/types.go */
package v1alpha1

import (
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// +genclient
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type Foo struct {
  metav1.TypeMeta   `json:",inline"`
  metav1.ObjectMeta `json:"metadata,omitempty"`

  Spec   FooSpec   `json:"spec"`
```

```
    Status FooStatus `json:"status"`
}

type FooSpec struct {
    DeploymentName string `json:"deploymentName"`
    Replicas       *int32 `json:"replicas"`
}

type FooStatus struct {
    AvailableReplicas int32 `json:"availableReplicas"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// FooList is a list of Foo resources
type FooList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata"`

    Items []Foo `json:"items"`
}
```

v1alpha1/register.go 文件

用于将刚建立的新API版本与新资源类型注册到API Group Schema中, 以便API Server能识别

> *Scheme:* 用于*API* 资源群组之间的的序列化, 反序列化与版本转换

```
// SchemeGroupVersion is group version used to register these objects
var SchemeGroupVersion = schema.GroupVersion{Group:
appcontroller.GroupName, Version: "v1alpha1"}

// Kind takes an unqualified kind and returns back a Group qualified
GroupKind
func Kind(kind string) schema.GroupKind {
    return SchemeGroupVersion.WithKind(kind).GroupKind()
}

// Resource takes an unqualified resource and returns a Group qualified
GroupResource
func Resource(resource string) schema.GroupResource {
    return SchemeGroupVersion.WithResource(resource).GroupResource()
}

var (
    // SchemeBuilder initializes a scheme builder
```

```
    SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
    // AddToScheme is a global function that registers this API group &
version to a scheme
    AddToScheme = SchemeBuilder.AddToScheme
)

// Adds the list of known types to Scheme.
func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &App{},
        &AppList{},
    )
    metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
    return nil
}
```

register.go

这个文件很简单, 主要是给CRD取个groupname

```
package appcontroller

// GroupName is the group name used in this package
const (
    GroupName = "appcontroller.example.com"
)
```

4. 生成代码

```
/home/aiedge/kubebuilder-demo/code-generator/generate-groups.sh \
# 期望生成的函数列表 all is "deepcopy,client,informer,lister"
all \
# 生成代码的目标目录
pkg/generated  \
# CRD所在目录
pkg/apis   \
# CRD的group name和version
crd.example.com:v1   \
# 这个文件里面其实是开源授权说明，但如果没有这个入参，该命令无法执行
--go-header-file "/home/aiedge/kubebuilder-demo/code-
generator/hack/boilerplate.go.txt"  \
# 指定输出文件夹, 默认是GOPATH/Src
--output-base ../../ -v 10
```

```
└── pkg
    ├── apis
    │   └── crd.example.com
```

```
│       └── v1
│           ├── doc.go
│           ├── register.go
│           ├── types.go
│           └── zz_generated.deepcopy.go
└── generated
    ├── clientset
    │   └── versioned
    │       ├── clientset.go
    │       ├── doc.go
    │       ├── fake
    │       │   ├── clientset_generated.go
    │       │   ├── doc.go
    │       │   └── register.go
    │       ├── scheme
    │       │   ├── doc.go
    │       │   └── register.go
    │       └── typed
    │           └── crd.example.com
    │               └── v1
    │                   ├── crd.example.com_client.go
    │                   ├── doc.go
    │                   ├── fake
    │                   │   ├── doc.go
    │                   │   ├── fake_crd.example.com_client.go
    │                   │   └── fake_foo.go
    │                   ├── foo.go
    │                   └── generated_expansion.go
    ├── informers
    │   └── externalversions
    │       ├── crd.example.com
    │       │   ├── interface.go
    │       │   └── v1
    │       │       ├── foo.go
    │       │       └── interface.go
    │       ├── factory.go
    │       ├── generic.go
    │       └── internalinterfaces
    │           └── factory_interfaces.go
    └── listers
        └── crd.example.com
            └── v1
                ├── expansion_generated.go
                └── foo.go
```

5. 编写该自定义 CRD 的 Controller

在上面生成了代码之后, 下面只需要编写controller的逻辑了。

利用client-go和code-generator生成的代码来完成控制器的核心功能, 通常在写一个控制器的时候, 会建一个controller struct, 并包含下面元素

- **Clientset**: 控制器与Kubernetes API Server进行互动，以操作资源。
- **Informer**:控制器的SharedInformer，用于接收API事件，并呼叫回调函数。
- **InformerSynced**: 确认SharedInformer的储存是否以获得至少一次完整list通知。
- **Lister**: 用于列出或获取缓存中的VirtualMachine资源。
- **Workqueue**:控制器的资源处理队列，都Informer收到事件时，会将物件推到这个队列，并在协调程序取出处理。当发生错误时，可以用于Requeue当前物件。

```
    samplev1alpha1 "app-controller/pkg/apis/appcontroller/v1alpha1"
    clientset "app-controller/pkg/generated/clientset/versioned"
    samplescheme "app-
controller/pkg/generated/clientset/versioned/scheme"
    informers "app-
controller/pkg/generated/informers/externalversions/appcontroller/v1alph
a1"
    listers "app-
controller/pkg/generated/listers/appcontroller/v1alpha1"
```

demo

235上operator-crd

Kubebuilder

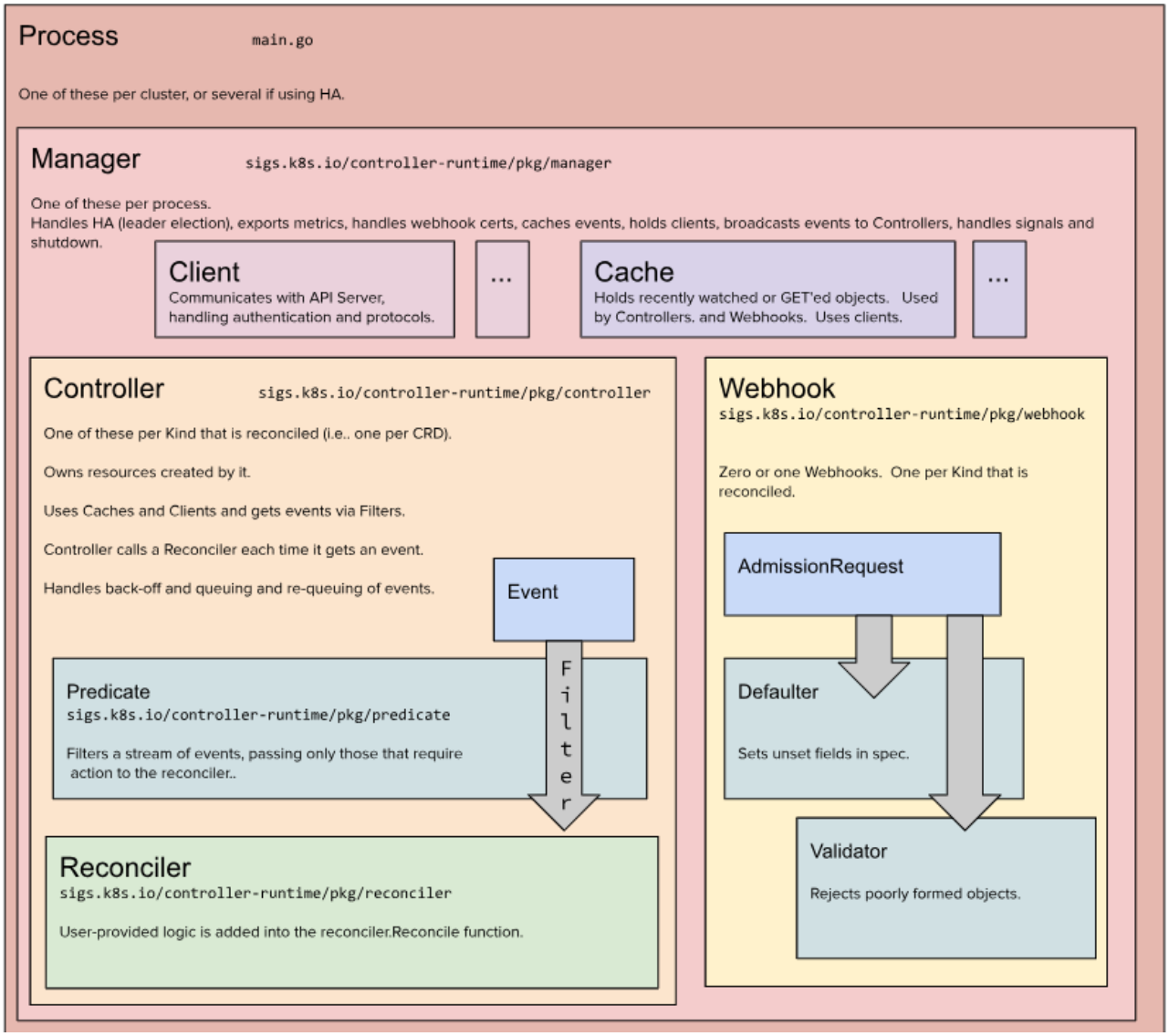
Kubebuilder 是基于 Custom Resource Definition (CRD) 构建 Kubernetes API 的框架由 Kubernetes Special Interest Group (SIG) API Machinery 所有及维护

官方文档 **https://book.kubebuilder.io/introduction.html**

架构

> *Architecture - The Kubebuilder Book*
>
> *kubebuild背后原理分析 - 知乎 (zhihu.com)*

kubebuilder底层使用的就是**Controller-runtime**



> 架构图理解：
>
> 从外向内看这张图，最外层是一个*main.go*的*process*进程。往里看是*manager*，用来管理*kubebuilder*里的模块，接下来往里看有哪些模块：*client*、*cache*、*controller*、*webhook*。其中，*webhook*是一个用于准入控制验证的，对*cr*格式的验证，是可选的。
>
> *kubebuilder*是基于*client-go*进行封装，用*controller-runtime*这个库实现*controller*的调谐过程。开发者只需要填资源的定义、监控的资源事件、*reconciler*逻辑。这些都是基于*client*、*cache*、*controllerm*模块实现的

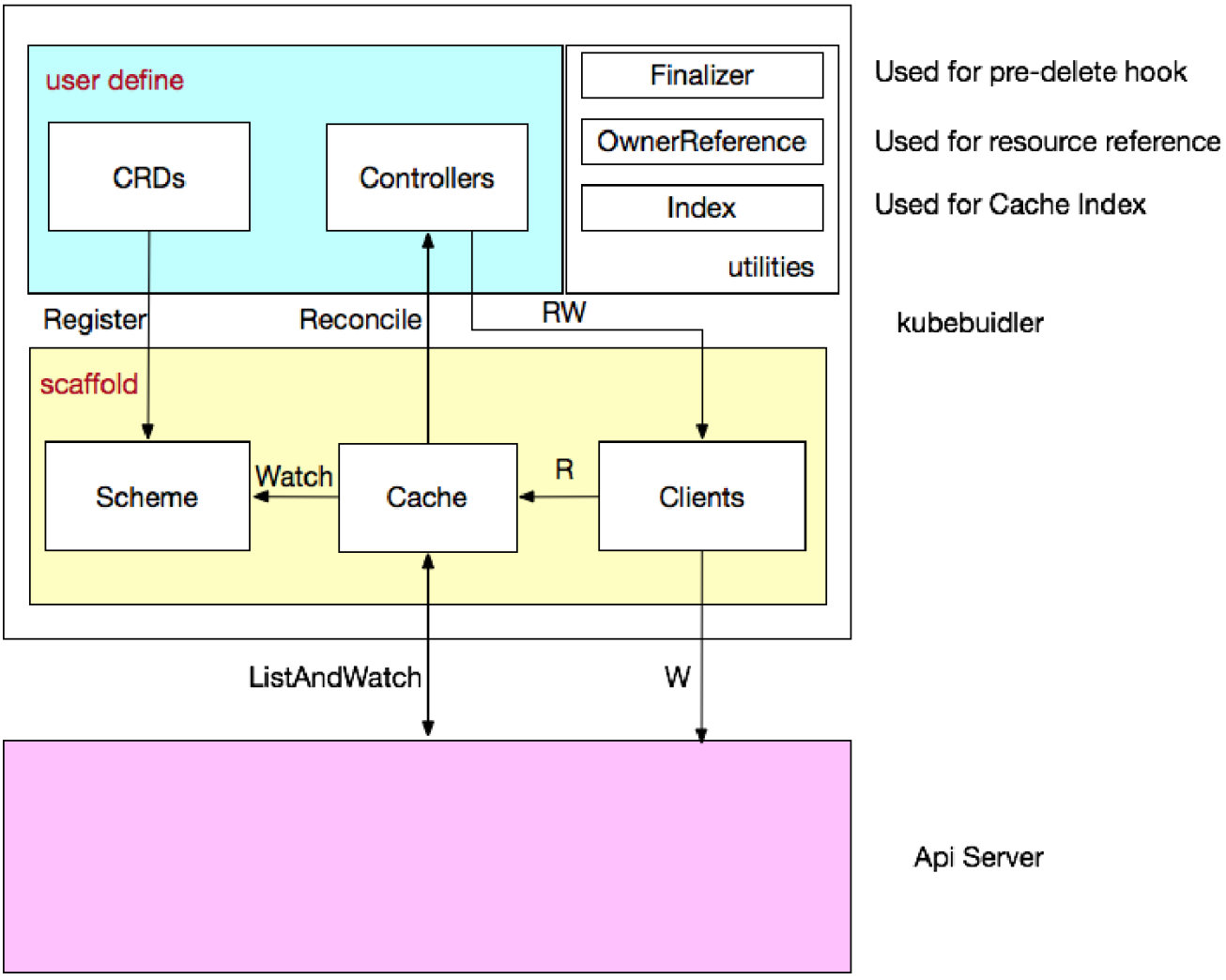
- （1） **Client**：用于读写 Kubernetes 资源对象的客户端。和apiserver交互的
- （2） **Cache**：本地缓存，用于保存需要监听的 Kubernetes 资源。缓存提供了只读客户端， 用于从缓存中读取对象。缓存还可以注册处理方法（EventHandler），以响应更新的事件。
- （3） **Manager**：用于控制多个 Controller，提供 Controller 共用的依赖项，如 **Client**、 **Cache**、**Schemes** 等。通过调用 **Manager.Start** 方法， 可以启动 **Controller**。

- （4）**Controller**：控制器，响应事件（Kubernetes 资源对象的创建、更新、删除）并 确保对象规范（Spec 字段）中指定的状态与系统状态匹配，如果不匹配，则控制器需要根 据事件的对象，通过协调器（Reconciler）进行同步。一方面会向**informer**注册**eventHandler**，另一方面会从**queue**中拿出数据并执行**reconciler**函数
    - **reconciler**：自定reconciler义业务逻辑。获取当前资源状态与预期状态做对比
    - **predicate**：使用它来对event进行过滤，筛选出所需要监听的资源的某些事件
- （5）**WebHook**：准 入 WebHook（Admission WebHook） 是 扩 展 Kubernetes API 的 一种机制，WebHook 可以根据事件类型进行配置，比如资源对象的创建、删除、更改等 事件，当配置的事件发生时，Kubernetes 的 APIServer 会向 WebHook 发送准入请求 （AdmissionRequests），WebHook 可以对请求中的资源对象进行更改或准入验证，然后将 处理结果响应给 APIServer。

Controller-runtime

在K8s中，实现一个controller是通过controller-runtime(https://github.com/kubernetes-sigs/controller-runtime) 框架来实现的，包括Kubebuilder、operator-sdk等工具也只是在controller-runtime上做了封装，以便开发者快速生成项目的脚手架而已。

kubebuilder与kubernetes控制器的关系



> 这张图主要分两个部分：*Api server* 与 *kubebuilder*
>
> *kubebuilder* 中也可以分成两部分 ： 上面是用户自定义的 *crd* 以及 *controller* 中要监控的资源事件及*reconciler* 的逻辑；
>
> - *Controller Watch* 自定义资源，此时 *controller* 会从 *Cache* 里面去获取 *Share Informer*,如果没有则创建,然后对该 *Share Informer* 进行 *Watch*,将得到的资源的名字和 *Namespace*存入到*Queue*中；
>
> 然后是黄色部分的脚手架（基于*controller-runtime*的封装），分为三个模块：
>
> - *cache*：*cache* 内部有很多*Informer*,注册到*scheme*中的资源就会生成对应的 *informer*，*informer*去监听对应的事件，监听到之后放入*workqueue*, *controller* 中的*reconciler*会取出*workqueue*中的*items*进行相应的处理
> - *scheme*：*crds*定义的*go types*映射 *GVK*，资源注册到*scheme*中，
> - *client*：在*reconcile*逻辑中会对资源进行读写，写是直接与*api-server*交互，读是在*cache*中读取
>
> 右上角就是一些辅助工具
>
> - *finalizer*：预删除，删除前进行一些操作。在 *K8s* 中，只要对象 *ObjectMeta* 里面的 *Finalizers* 不为空，对该对象的 *delete* 操作就会转变为 *update* 操作，具体说就是 *update deletionTimestamp* 字段，其意义就是告诉 *K8s* 的 *GC*“在 *deletionTimestamp* 这个时刻之后，只要 *Finalizers* 为空，就立马删除掉该对象”。
> - *ownerReference*：建立资源所属关系。*K8s GC* 在删除一个对象时，任何 *ownerReference* 是该对象的对象都会被清除，与此同时，*Kubebuidler* 支持所有对象的变更都会触发 *Owner* 对象 *controller* 的 *Reconcile* 方法。
> - *index*：由于 *Controller* 经常要对 *Cache* 进行查询，*Kubebuilder* 提供 *Index utility* 给 *Cache* 加索引提升查询效率。
>
> 流程：
>
> - 首先，注册到*scheme*中的资源就会生成对应的*informer*。*controller*会向*cache*中的*informer*注册特定资源的*eventHandler*
> - 然后，*cache*会启动*informer*，*informer*向*apiserver*发出请求，建立连接
> - 当*informer*监听对应的事件后，使用*controller*注册来的*eventHandler*判断是否推入*workerqueue*
> - 当*workqueue*队列中有元素被推入，*controller*会将元素取出，并执行用户侧的 *reconciler*

辅助工具

ownerReference：

- SetControllerReference 函数
- SetOwnerReference 函数：该方法使您可以声明owner对object具有依赖关系，而无需将其指定为控制器（Controller=true）

使用

安装

- https://github.com/kubernetes-sigs/kubebuilder/releases/download/v3.11.1/kubebuilder_linux_amd64下载对应版本
- kubebuilder version

常用命令

- kubebuilder init [--plugins= [--project-version=]] 初始化项目目录
- kubebuilder create [command] ，[command]为api时创建资源创建controller，eg. kubebuilder create api --group ship --version v1beta1 --kind Frigate
- 生产CRD：make manifests
- 编译工程：make
- 部署CRD：make install
- 运行CR：make run
- 制作镜像并上传：make docker-build docker-push IMG=/:tag
- 部署operator：make deploy IMG=/:tag

实际使用

- 创建项目文件夹，初始化go mod
- 创建脚手架工程，`kubebuilder init --domain example.com`，其中`example.com`是创建的CRD的组名的一部分。此时会生成一些文件夹

```
aiedge@xx-test-master235:~/kubebuilder-demo/kubebuilder-demo$ ls
Dockerfile  Makefile  PROJECT  README.md  cmd  config  go.mod
go.sum  hack
```

> 其中，
>
> *`Dockerfile`*构建 *Docker* 镜像的文件
>
> *`Makefile`*项目的构建运行
>
> *`PROJECT`*：用于搭建新组建的*Kubebuilder*元数据
>
> *`cmd`*里面是***main***文件，控制器入口
>
> *`config`*是一些*yaml*文件。*config/manager*：集群*pod*启动相关的控制器*yaml*。*config/rbac*：运行控制器所需的权限定义。
>
> *`hack`*是头文件*boilerplate.go.txt*，辅助脚本和工具

- 创建**API**，`kubebuilder create api --group ingress --version v1beta1 --kind App`创建资源创建controller。此时会生成一些文件

```
aiedge@xx-test-master235:~/kubebuilder-demo/kubebuilder-demo$ ls
Dockerfile  Makefile  PROJECT  README.md  api  bin  cmd  config
go.mod  go.sum  hack  internal
```

> *api/vebeta1*：其中有*types*、 *groupversion*（有关*group*的定义变量）、*zz_deepcopy*文件
>
> *bin*里面是*controller-gen*
>
> *internal/controller*是*controller*相关的。在*app_controller.go*中*Reconcile()*函数是我们需要自己填充的，类似之前的*synchandler()*。

```
# 此时目录结构如下：
.
├── Dockerfile  #构建镜像的Dockerfile文件
├── Makefile#构建和部署文件
├── PROJECT#工程元数据
├── README.md
├── api
│   └── v1beta1
│       ├── app_types.go #自定义CRD类型文件,里面为自定义参数封装
│       ├── groupversion_info.go
│       └── zz_generated.deepcopy.go
├── bin
│   └── controller-gen
├── cmd
│   └── main.go#程序入口
├── config
│   ├── crd
│   │   ├── kustomization.yaml
│   │   ├── kustomizeconfig.yaml
│   │   └── patches
│   │       ├── cainjection_in_apps.yaml
│   │       └── webhook_in_apps.yaml#部署到K8S的CRD yaml
│   ├── default
│   │   ├── kustomization.yaml
│   │   ├── manager_auth_proxy_patch.yaml
│   │   └── manager_config_patch.yaml
│   ├── manager  #部署到K8S的控制器 yaml
│   │   ├── kustomization.yaml
│   │   └── manager.yaml
│   ├── prometheus
│   │   ├── kustomization.yaml
│   │   └── monitor.yaml
│   ├── rbac #部署到K8S的权限控制 yaml
```

```
│   │   ├── app_editor_role.yaml
│   │   ├── app_viewer_role.yaml
│   │   ├── auth_proxy_client_clusterrole.yaml
│   │   ├── auth_proxy_role.yaml
│   │   ├── auth_proxy_role_binding.yaml
│   │   ├── auth_proxy_service.yaml
│   │   ├── kustomization.yaml
│   │   ├── leader_election_role.yaml
│   │   ├── leader_election_role_binding.yaml
│   │   ├── role_binding.yaml
│   │   └── service_account.yaml
│   └── samples
│       ├── ingress_v1beta1_app.yaml#部署到K8S的CR yaml说明
│       └── kustomization.yaml
├── go.mod
├── go.sum
├── hack
│   └── boilerplate.go.txt
└── internal
    └── controller # 自定义资源控制逻辑
        ├── app_controller.go
        └── suite_test.go

15 directories, 38 files
```

- 定义**CRD**，修改crd字段(可选)

  改该文件 api/v1/guestbook_types.go

```
type AppSpec struct {
    //+kubebuilder:default:enable_ingress=false
    EnableIngress bool   `json:"enable_ingress,omitempty"`
    EnableService bool   `json:"enable_service"`
    Replicas      int32  `json:"replicas"`
    Image         string `json:"image"`
}
//其中Image、Replicas、EnableService为必须设置的属性，EnableIngress可以为空.
```

- 更新crd和cr

```
make manifests
```

  生成了config/crd/bases/ingress.example.com_apps是**CRD**的**yaml**。

修改config/crd/samples/ingress_v1beta1_app.yaml(一个CR的例子)

```
apiVersion: ingress.example.com/v1beta1
kind: App
metadata:
  labels:
    app.kubernetes.io/name: app
    app.kubernetes.io/instance: app-sample
  name: app-sample
spec:
  # TODO(user): Add fields here
  image: nginx:latest
  replicas: 3
  enable_ingress: false #会被修改为true
  enable_service: true #成功
```

- 编写**controller**逻辑，在控制器中实现协调循环（reconcile loop），watch 额外的资源
- 测试

本地测试

- 安装CRDS到集群

```
make install
go run cmd/main.go //本地测试的时候可以直接启动程序测试
```

- 部署cr的例子

```
kubectl apply -f config/samples
```

到此时，crd和cr成功部署

- 修改cr文件，看controller是否在生效

```
kubectl edit -f config/samples/ingress_v1beta1_app.yaml
```

> 如果要将控制器部署到集群中去，首先需要构建并推送镜像到镜像仓库：
>
> ```
> $ make docker-build docker-push IMG=<some-registry>/<project-name>:tag
> ```
>
> 根据 *IMG* 指定的镜像将控制器部署到集群中:

```
$ make deploy IMG=<some-registry>/<project-name>:tag
```

- 安装CRDS到集群

```
make install
go run cmd/main.go //本地测试的时候可以直接启动程序测试
```

- 部署cr的例子

```
kubectl apply -f config/samples
```

到此时，crd和cr成功部署

- 运行实例

```
make run
```

要从你的集群中删除 CRD 也很简单：

```
$ make uninstall
```

代码部署到集群内部

- 写Dockerfile文件
- 本地构建`docker build -t xiaox1958141/ingress-manager:v1.0.0 .`
- 推镜像到仓库`docker push xiaox1958141/ingress-manager:v1.0.0`。（记得sudo docker login）
- 部署 `kubectl create deployment ingress-manager --image xiaox1958141/ingress-manager:v1.0.0 --dry-run=client -o yaml`。其中，通过 --dry-run=client 参数告诉 Kubernetes 只执行客户端验证而不实际创建资源。根据输出判断是否符合要求，是的话保存至manifests目录下。`> manifests/ingress-manager.yaml`

```
kubectl create deployment ingress-manager --image
xiaox1958141/ingress-manager:v1.0.0 --dry-run=client -o yaml

kubectl create ingress ingress-manager --namespace=default --dry-
run=client -o yaml \
>   --rule="your-app.example.com/*=default-backend-service:80"
```

- 创建serviceAccount：`kubectl create serviceaccount ingress-manager-sa --dry-run=client -o yaml`没报错没问题保存至manifests目录下。`> manifests/ingress-manager-sa.yaml`
- 在ingress-manager.yaml中添加`serviceAccountName: ingress-manager-sa`
- 创建role进行角色绑定`kubectl create clusterrole ingress-manager-role --resource=ingress,service --verb list,watch,create,update,delete --dry-run=client -o yaml > manifests/ingress-manager-role.yaml`，绑定`kubectl create clusterrolebinding ingress-manager-rb --clusterrole ingress-manager-role --serviceaccount default:ingress-manager-sa --dry-run=client -o yaml > manifests/ingress-manager-rb.yaml`
- 应用`kubectl apply -f manifests`，查看对应资源是否创建好 `kubectl get clusterrole ingress-manager-role`
- 测试 `kubectl run nignx-demo --image=nginx:latest` `kubectl expose pods/nignx-demo --port 80` `kubectl get ingress` 修改svc`kubectl edit service nignx-demo`加上annotation.`kubectl get ingress`

debug

```
kubectl get pods -n kube-system -l component=kube-controller-manager
```

得到pod 查看log

```
// +kubebuilder:resource:path=pipelines,scope=Namespaced
```

```
kubectl logs kube-controller-manager-xx-test-master235 -n kube-system
|grep
```

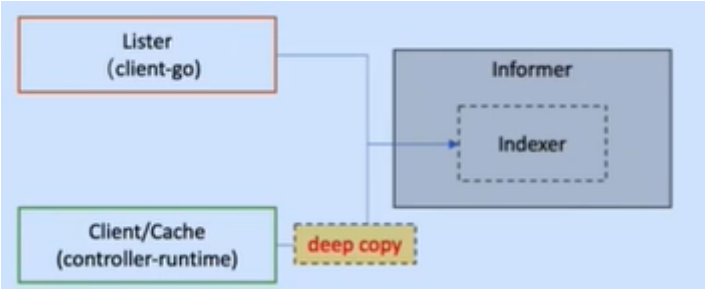
```
type StepPhase struct {
    PVPhase         string `json:"pvPhase"`
    DeploymentPhase string `json:"deploymentPhase"`
    ServicePhase    string `json:"servicePhase"`
}

type PipelinePhase struct {
    StepsPhase []StepPhase `json:"steps"`
}

// +kubebuilder:default:phase=creating
// PipelineStatus defines the observed state of Pipeline
type PipelineStatus struct {
```

```
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this
file
    Phase PipelinePhase `json:"phase"`
}
```

demo

https://github.com/esparig/kubebuilder-cronjob/blob/main/api/v1/cronjob_types.go

deep copy



- 之前的client-go：不能对取到的cache数据进行修改
- 现在的controller-runtime：全局深拷贝

新的版本允许 针对某些资源disable deep copy，用到时手动deep copy

源码分析

项目源码，manager中skip至每个模块定义

main.go

去看源码，main.go 主要是做了一些启动的工作包括：

- 初始化一个 **Manager**
- 初始化 **Controller**（**Reconciler**）：将 Manager 的 Client 传给 Controller，并且调用 SetupWithManager 方法传入 Manager 进行 Controller 的初始化；
- （启动 WebHook）
- 添加健康检查
- 启动 **Manager**

> 在 *init* 方法里面我们初始化*Scheme*, 注册原生资源以及自定义资源;，这样一来 *Cache* 就知道 *watch* 谁了

具体流程：

1. 初始化一个 **Manager**，mgr, err := ctrl.**NewManager**(...)，
   - 创建并注册scheme ，其中`scheme  = runtime.NewScheme()`
   - 创建cluter，to create the **Client and Cache**.

     “记录`schema`中每个`GVK`对象与`informer`的对应关系，使用时可根据`GVK`得到`informer`再去`List`/`Get`。”
   - **`Runnable`** 接口提供了一个抽象层，将 manager 启动、对象添加和指标添加等操作进行了统一。添加manager所需组件（Webhooks Caches LeaderElection）放入**runable**，负责控制器的启动和运行。再把runable注入manager

     runnable 该对象用来管理controller-runtime中可执行组件的集合，当前一共分为了Webhooks、Caches、LeaderElection和others四类，之后还会把controller注入。
2. 初始化**controller**：将**controller**绑定到**manager**，**SetupWithManager**方法(每个controller自己实现的)

```
func (r *PipelineReconciler) SetupWithManager(mgr ctrl.Manager)
error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&distriinferv1beta1.Pipeline{}).
        Owns(&corev1.PersistentVolume{}).
        Owns(&corev1.PersistentVolumeClaim{}).
        Owns(&appv1.Deployment{}).
        Owns(&corev1.Service{}).
        Complete(r)
}
```

   - 通过 **NewControllerManagedBy**创建**controller**， **Builder** 模式实例化一个 **Builder** 对象（用于构建controller），其中传入的 Manager（第1步生成的）的一些配置注入到controller， 提供创建 **Controller** 所需的依赖。
   - 挑选要监听的资源及关心的事件
     - 使用 **For**（object client.Object，opts ...ForOption）方法设置需要监听的资源 类型。实际就是完善Builder的forInput结构体。
     - **Owns** 指定的是我所关心的资源的从属资源。当监听到从属资源变更，其**event**也会进入到**controller**的**queue**中。会触发**Reconcile**逻辑
     - **watch**。
   - **Complete** 用于生成controller，将用户的reconciler注册到controller，并生成watch资源的默认eventHandler，同时执行controller的watch函数

```
type Builder struct {
    forInput         ForInput//主资源
    ownsInput        []OwnsInput//主资源的从属资源
    watchesInput     []WatchesInput
    mgr              manager.Manager
    globalPredicates []predicate.Predicate
    ctrl             controller.Controller
    ctrlOptions      controller.Options
    name             string
}
finalizers:
  - kubernetes.io/pv-protection
```

Complete 最后其实是调用了 **Build** 方法。Build 中最主要**doController**以及**doWatch**

```
func (blder *Builder) Build(r reconcile.Reconciler)
(manager.Manager, error) {
...
    // Set the Manager
    if err := blder.doManager(); err != nil {
        return nil, err
    }
    // Set the ControllerManagedBy
    if err := blder.doController(r); err != nil {
        return nil, err
    }
    // Set the Watch
    if err := blder.doWatch(); err != nil {
        return nil, err
    }
...
    return blder.mgr, nil
}
```

主要是看看 doController 和 doWatch 方法：

- **doController**：初始化了一个 Controller，这里面传入了我们实现 的 Reconciler以及获取到我们的 GVK 的名称。调用`controller.New`来创建`controller`并添加到`manager`的`runnable`
    - Do：Reconcile 逻辑；
    - Cache：找 Informer 注册 Watch；
    - Client：对 K8s 资源进行 CRUD；
    - Queue：Watch 资源的 CUD 事件缓存；
    - Recorder：事件收集。

```
//设置MaxConcurrentReconciles并发量为1,cache同步时间间隔2分钟
//...
return &controller.Controller{
    Do: options.Reconciler,
    MakeQueue: func() workqueue.RateLimitingInterface {
            return
workqueue.NewNamedRateLimitingQueue(options.RateLimiter,
name)
        },
        MaxConcurrentReconciles:
options.MaxConcurrentReconciles,
        CacheSyncTimeout:        options.CacheSyncTimeout,
        Name:                    name,
        LogConstructor:          options.LogConstructor,
        RecoverPanic:            options.RecoverPanic,
        LeaderElected:
options.NeedLeaderElection,
    }, nil
}
```

- **doWatch**：watch CRD 资源的变更，以及watch CRD 资源的own resouces的变更。通过过滤源事件的变化，allPredicates是过滤器，只有所有的过滤器都返回 **true** 时，才会将事件传递给 **EventHandler hdler**，这里会将 Handler 注册到 Informer 上

```
func (blder *Builder) doWatch() error {
    if blder.forInput.object != nil {
        //将主资源进行投影
        obj, err := blder.project(blder.forInput.object,
blder.forInput.objectProjection)
            if err != nil {
                return err
            }
        //创建主资源事件源
            src := source.Kind(blder.mgr.GetCache(), obj)
            hdler := &handler.EnqueueRequestForObject{}
            allPredicates :=
append(blder.globalPredicates,
blder.forInput.predicates...)
        //设置对主资源的监视
        if err := blder.ctrl.Watch(src, hdler,
allPredicates...); err != nil {
                return err
            }
        }
    //监视子资源：
    //...
```

```
}
```

（在SetupWithManager的步骤中，controller-runtime会将创建的controller放到runnables中）

```
// Watch implements controller.Controller
func (c *Controller) Watch(src source.Source, evthdler
handler.EventHandler, prct ...predicate.Predicate) error {
    ...
    log.Info("Starting EventSource", "controller", c.Name,
"source", src)
    return src.Start(evthdler, c.Queue, prct...)
}
//Start
func (is *Informer) Start(handler handler.EventHandler, queue
workqueue.RateLimitingInterface,
    ...

 //我们的 Handler 实际注册到 Informer 上面

is.Informer.AddEventHandler(internal.EventHandler{Queue: queue,
EventHandler: handler, Predicates: prct})
    return nil
}
```

这样整个逻辑就串起来了，通过 Cache 我们创建了所有 Scheme 里面 GVKs 的 Informers，然后对应 GVK 的 Controller 注册了 Watch Handler 到对应的 Informer，这样一来对应的 GVK 里面的资源有变更都会触发 Handler，将变更事件写到 Controller 的事件队列中，之后触发我们的 Reconcile 方法。

3. 启动**Manager**，mgr.Start(ctrl.SetupSignalHandler())。所有的Start实现了Runnable接口，这里调用Start将启动**Runnable**中所有组件。调用了 对应组件的Start方法

   主要就是启动 Cache，Controller，将整个事件流运转起来

```
https://github.com/kubernetes-sigs/controller-
runtime/blob/main/pkg/manager/internal.go
//启动监控服务
//...
//启动cache，启动weebhook等其他组件服务
controllermanager.runnables.Webhooks.Start
//启动Controller，  Controller.start
{
    //等待 cache 同步
    //根据MaxConcurrentReconciles，启动多个processNextWorkItem
    //每个 Worker 调用reconcileHandler 来进行数据处理
    result, err := c.Reconcile(ctx, req)
```

```
    ...
}
```

```
//上面的Reconcile方法 最终return c.Do.Reconcile(ctx, req)
//c.Do.Reconcile与sample-controller工作流程一致，不断获取工作队列中的数据调用Reconcile进行调谐。
//c.Do.Reconcile最终就是app-controller.go中需要自己实现的的Reconcile方法
```

Manager

Controller runtime 中引入了 Manager 组件。在一个进程中可以有多个 **Controller**，每个 **Controller** 负责对一种资源进行调谐。Manager 则用来统一管理这些 Controller 的生命周期。职责：

- 负责运行所有的 Controllers；
- 初始化共享 caches，包含 listAndWatch 功能；
- 初始化 clients 用于与 Api Server 通信。

interface结构体

```
type Manager interface {
  cluster.Cluster                    //cluster.Cluster  提供了一系列方法，以获取与集群相关的对象。
  Add(Runnable) error                //添加controller
  Elected() <-chan struct{}          // 选举相关，返回一个 Channel 结构，用于判断选举状态。当未配置选举或当选 Leader 时，Channel 将被关闭。
  AddMetricsExtraHandler(path string, handler http.Handler) error  //metrics相关
  AddHealthzCheck(name string, check healthz.Checker) error        // 健康检查相关
  AddReadyzCheck(name string, check healthz.Checker) error         // 是否就绪
  Start(ctx context.Context) error                          // 启动所有的controller
  GetWebhookServer() *webhook.Server
  GetLogger() logr.Logger
  GetControllerOptions() v1alpha1.ControllerConfigurationSpec

}
```

Manager 初始化

```
func New(config *rest.Config, options Options) (Manager, error) {
    ...
    // Create the cache for the cached read client and registering informers
```

```
    cache, err := options.NewCache(config, cache.Options{Scheme:
options.Scheme, Mapper: mapper, Resync: options.SyncPeriod, Namespace:
options.Namespace})
    if err != nil {
        return nil, err
    }
    apiReader, err := client.New(config, client.Options{Scheme:
options.Scheme, Mapper: mapper})
    if err != nil {
        return nil, err
    }
    writeObj, err := options.NewClient(cache, config,
client.Options{Scheme: options.Scheme, Mapper: mapper})
    if err != nil {
        return nil, err
    }
    ...
```

可以看到主要是创建 Cache 与 Clients

Cache

是对client-go中的informer实现了一层包装.

负责在 **Controller** 进程里面根据 **Scheme** 同步 **Api Server** 中所有该 **Controller** 关心 **GVKs** 的 **GVRs**，其核心是 **GVK -> Informer** 的映射，创建了一些 **Informer**。每个 **Informer** 会负责监听对应 **GVK** 的 **GVRs** 的创建/删除/更新操作，以触发 **Controller** 的 **Reconcile** 逻辑。

interface结构体

```
type Cache interface {
    // Cache acts as a client to objects stored in the cache.
    client.Reader

    // Cache loads informers and adds field indices.
    Informers
}
```

创建cache

可以看到 Cache 主要就是创建了 Informers，Scheme 里面的每个 GVK 都创建了对应的 Informer，通过 informersByGVK 这个 map 做 GVK 到 Informer 的映射，每个 Informer 会根据 ListWatch 函数对对应的 GVK 进行 List 和 Watch。

启动cache

```
func (ip *specificInformersMap) Start(stop <-chan struct{}) {
    func() {
        ...
        // Start each informer
        for _, informer := range ip.informersByGVK {
            go informer.Informer.Run(stop)
        }
    }()
}
func (s *sharedIndexInformer) Run(stopCh <-chan struct{}) {
        ...
        // informer push resource obj CUD delta to this fifo queue
    fifo := NewDeltaFIFO(MetaNamespaceKeyFunc, s.indexer)
    cfg := &Config{
        Queue:            fifo,
        ListerWatcher:    s.listerWatcher,
        ObjectType:       s.objectType,
        FullResyncPeriod: s.resyncCheckPeriod,
        RetryOnError:     false,
        ShouldResync:     s.processor.shouldResync,
                // handler to process delta
        Process: s.HandleDeltas,
    }
    func() {
        s.startedLock.Lock()
        defer s.startedLock.Unlock()
                // this is internal controller process delta generate by
reflector
        s.controller = New(cfg)
        s.controller.(*controller).clock = s.clock
        s.started = true
    }()
        ...
    wg.StartWithChannel(processorStopCh, s.processor.run)
    s.controller.Run(stopCh)
}
func (c *controller) Run(stopCh <-chan struct{}) {
    ...
    r := NewReflector(
        c.config.ListerWatcher,
        c.config.ObjectType,
        c.config.Queue,
        c.config.FullResyncPeriod,
    )
    ...
        // reflector is delta producer
    wg.StartWithChannel(stopCh, r.Run)
```

```
        // internal controller's processLoop is comsume logic
    wait.Until(c.processLoop, time.Second, stopCh)
}
```

Cache 的初始化核心是初始化所有的 **Informer**，Informer 的初始化核心是创建了 reflector 和内部 controller，reflector 负责监听 Api Server 上指定的 GVK，将变更写入 delta 队列中，可以理解为变更事件的生产者，内部 controller 是变更事件的消费者，他会负责更新本地 indexer，以及计算出 CUD 事件推给我们之前注册的 Watch Handler。

Client

在实现 Controller 的时候不可避免地需要对某些资源类型进行创建/删除/更新，就是通过该 Clients 实现的。实际上也是对client-go的一层包装。client 创建了两个。一个用于读，一个用于写。其中查询功能实际查询是本地的 **Cache**，写操作直接访问 **Api Server**。

Cluster组件中包含了**Cache**和**Client**组件

```
// Client knows how to perform CRUD operations on Kubernetes objects.
type Client interface {
    Reader
    Writer
    StatusClient
    SubResourceClientConstructor

    // Scheme returns the scheme this client is using.
    Scheme() *runtime.Scheme
    // RESTMapper returns the rest this client is using.
    RESTMapper() meta.RESTMapper
    // GroupVersionKindFor returns the GroupVersionKind for the given
object.
    GroupVersionKindFor(obj runtime.Object) (schema.GroupVersionKind,
error)
    // IsObjectNamespaced returns true if the GroupVersionKind of the
object is namespaced.
    IsObjectNamespaced(obj runtime.Object) (bool, error)
}
```

```
//读操作client
type client struct {
    typedClient        typedClient
    unstructuredClient unstructuredClient
    metadataClient     metadataClient
    scheme             *runtime.Scheme
    mapper             meta.RESTMapper

    cache              Reader
    uncachedGVKs       map[schema.GroupVersionKind]struct{}
    cacheUnstructured bool
}
```

读操作使用上面创建的 Cache，写操作使用 K8s go-client 直连。

controller

在生成的文件中：**internal/controller/app_controller.go**有两个方法

启动

controller跟随manager.start而启动。

- controller.start()中定义了：启动 goroutine 不断地查询队列，如果有变更消息则触发到我们自定义的 Reconcile 逻辑。

  ```
  func (c *Controller) Start(stop <-chan struct{}) error {
  ...

        // Launch workers to process resources
        log.Info("Starting workers", "controller", c.Name, "worker
  count", c.MaxConcurrentReconciles)
        for i := 0; i < c.MaxConcurrentReconciles; i++ {
          // 多个workder，间隔一定时间(1s)工作一次
           go wait.Until(c.worker, c.JitterPeriod, stop)
        }

  ...
  }
  ```

  调用c.worker

- c.worker()，开启循环不断处理下一个worker

```
func (c *Controller) worker() {
  for c.processNextWorkItem() {
  }
}
```

调用processNextWorkItem

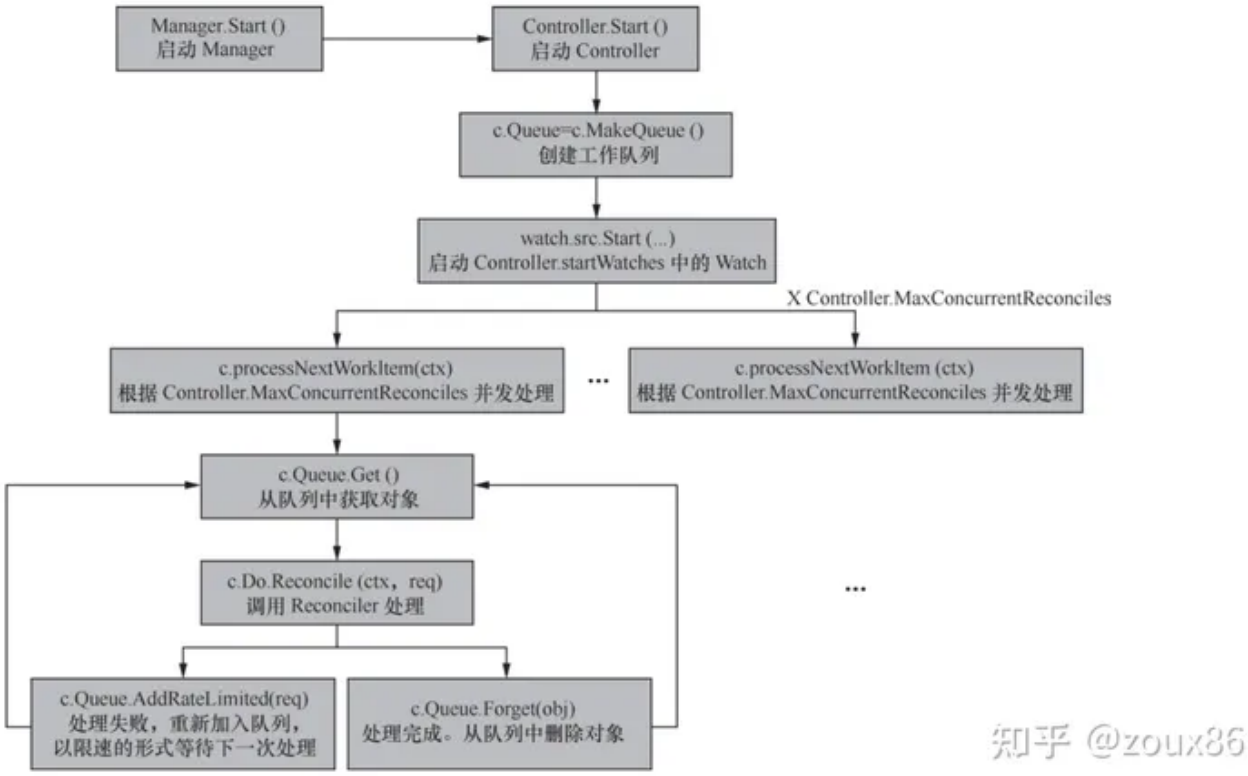
- processNextWorkItem

```
func (c *Controller) processNextWorkItem() bool {
  obj, shutdown := c.Queue.Get()
  if shutdown {
    // Stop working
    return false
  }

  defer c.Queue.Done(obj)
  return c.reconcileHandler(obj)
}
```

调用reconcileHandler

- reconcileHandler

```
func (c *Controller) reconcileHandler(obj interface{}) bool {
  ...

  // RunInformersAndControllers the syncHandler, passing it the
namespace/Name string of the
  // resource to be synced.
  if result, err := c.Do.Reconcile(req); err != nil {
    ...
  }

  ...
}
```



添加事件处理方法

builder中dowatch方法会根据 创建对应的handler、kind类型的source，当启动controller时，启动source，在kind.start方法中将handler注册到cache的informer中

```
i.AddEventHandler(NewEventHandler(ctx, queue, handler,
prct).HandlerFuncs())
```

```
// Kind is used to provide a source of events originating inside the
cluster from Watches (e.g. Pod Create).
type Kind struct {
    // Type is the type of object to watch.  e.g. &v1.Pod{}
    Type client.Object//触发事件的资源类型
    // Cache used to watch APIs
    Cache cache.Cache
    // started may contain an error if one was encountered during
startup. If its closed and does not
    // contain an error, startup and syncing finished.
    started     chan error
    startCancel func()
}
```

使用：

```
// 创建一个 source.Kind 事件源，用于监视 MyResource 类型的资源变化
src := &source.Kind{
    Type:   &myv1alpha1.MyResource{}, // MyResource 是一个自定义资源对象
    Scheme: mgr.GetScheme(),
}
```

```
// 创建一个 controller，使用上述事件源
if err := ctrl.NewControllerManagedBy(mgr).
    For(&myv1alpha1.MyResource{}).
    Owns(&corev1.Pod{}).
    Watches(src, &handler.EnqueueRequestForObject{}).
    Complete(&ReconcileMyResource{}); err != nil {
    log.Error(err, "unable to create controller")
    os.Exit(1)
}
```

过滤事件

在builder中，通过WithEventFilter添加**predicates**来决定create、update、delete事件是否处理

```
    src := source.Kind(blder.mgr.GetCache(), obj)
        hdler := &handler.EnqueueRequestForObject{}
        allPredicates := append(blder.globalPredicates,
blder.forInput.predicates...)
        if err := blder.ctrl.Watch(src, hdler, allPredicates...);
//...
```

使用：

```
import ctrl "sigs.k8s.io/controller-runtime"
func (r *AppReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    _ = log.FromContext(ctx)

    // TODO(user): your logic here

    return ctrl.Result{}, nil
}

// SetupWithManager sets up the controller with the Manager.将controller
添加到MAnager中
func (r *AppReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&ingressv1beta1.App{}).
        Complete(r)
}
```

Reconcile

- `r *AppReconciler`

```
// AppReconciler reconciles a App object
type AppReconciler struct {
    client.Client
    Scheme *runtime.Scheme
}
```

- 函数请求参数中**`req ctrl.Request`**，实际上是`namespace + / + name`，就是之前手写controller中syncHandler的`key`
- 函数返回参数中`ctrl.Result{}`是**Reconciler**处理的结果。

```
type Result struct {
    // true: 重新入队这个key
    Requeue bool

    // 某个时间段后重新入队
    RequeueAfter time.Duration
}
```

- 函数主体需要自己完善。

返回

以下是重新启动协调的一些可能的返回选项：

- 出现错误：

```
return ctrl.Result{}, err
```

- 没有错误：

```
return ctrl.Result{Requeue: true}, nil
```

- 因此，要停止协调，请使用：

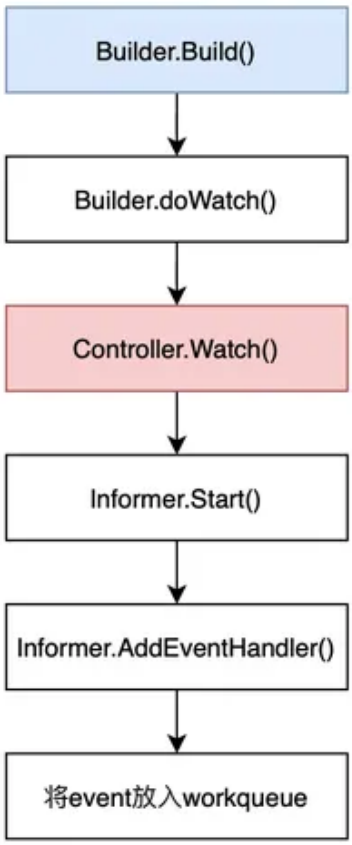
```
return ctrl.Result{}, nil
```

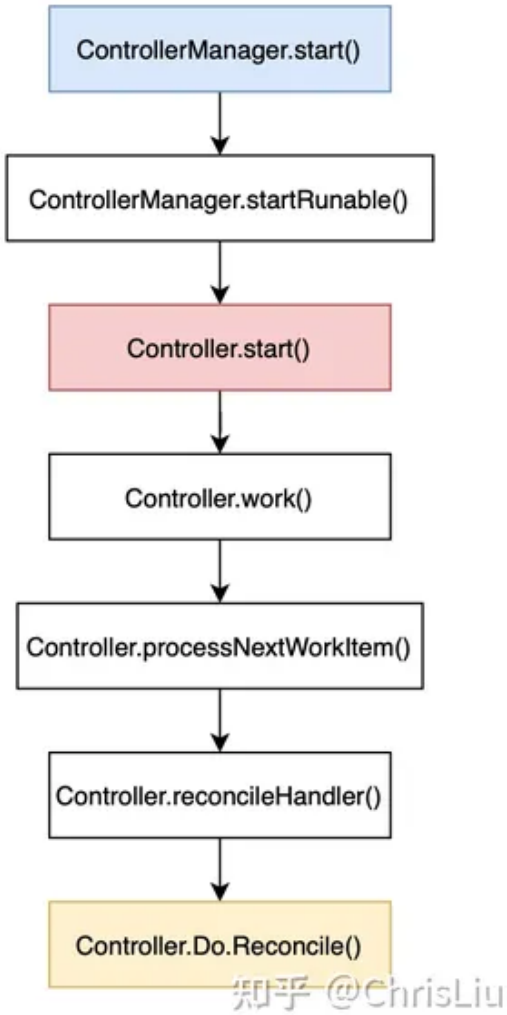
- X时间后再次协调：

```
return ctrl.Result{RequeueAfter: nextRun.Sub(r.Now())}, nil
```

总结

在这里以controller两个核心func——Watch\Start为中心，梳理出两条线表示 controller-runtime 的调用链。





kustomize

https://zhuanlan.zhihu.com/p/92153378

使用kubebuilder之后生成了kustomize相关的文件。

kustomize 是一个通过 **kustomization** 文件定制 **kubernetes** 对象的工具，它可以通过一些资源生成一些新的资源，也可以定制不同的资源的集合。

适用场景：

- 从其他来源生成资源
- 为资源设置贯穿性（Cross-Cutting）字段
- 组织和定制资源集合

> 官方定义：*kustomize* 使用 *k8s* 原生概念帮助创建并复用资源配置*(YAML)*，允许用户以一个应用描述文件 （***YAML*** 文件）为基础（***Base YAML***），然后通过 ***Overlay*** 的方式生成最终部署应用所需的描述文件。

一些术语

- **kustomization**

术语 kustomization 指的是 kustomization.yaml 文件，或者指的是包含 kustomization.yaml 文件的目录以及它里面引用的所有相关文件路径

- **base**

base 指的是一个 kustomization , 任何的 kustomization 包括 overlay (后面提到)，都可以作为另一个 kustomization 的 base (简单理解为基础目录)。base 中描述了共享的内容，如资源和常见的资源配置

- **overlay**

overlay 是一个 kustomization, 它修改(并因此依赖于)另外一个 kustomization. overlay 中的 kustomization指的是一些其它的 kustomization, 称为其 base. 没有 base, overlay 无法使用，并且一个 overlay 可以用作 另一个 overlay 的 base(基础)。简而言之，overlay 声明了与 base 之间的差异。通过 overlay 来维护基于 base 的不同 variants(变体)，例如开发、QA 和生产环境的不同 variants

- **variant**

variant 是在集群中将 overlay 应用于 base 的结果。例如开发和生产环境都修改了一些共同 base 以创建不同的 variant。这些 variant 使用相同的总体资源，并与简单的方式变化，例如 deployment 的副本数、ConfigMap使用的数据源等。简而言之，variant 是含有同一组 base 的不同 kustomization

- **resource**

在 kustomize 的上下文中，resource 是描述 k8s API 对象的 YAML 或 JSON 文件的相对路径。即是指向一个声明了 kubernetes API 对象的 YAML 文件

- **patch**

修改文件的一般说明。文件路径，指向一个声明了 kubernetes API patch 的 YAML 文件

示例

```
# pod.yaml contents
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

```
# deployments.yaml contents
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy
spec:
  template:
    metadata:
      labels:
        foo: bar
    spec:
      containers:
        - name: nginx
          image: nginx
          args:
          - arg1
          - arg2
```

WebHook

https://kubernetes.io/zh-cn/docs/reference/access-authn-authz/extensible-admission-controllers/

准入控制器 是一段代码，它会在请求通过认证和鉴权之后、对象被持久化之前拦截到达 **API** 服务器的请求。

准入**Webhook** 是一种用于接收准入请求并对其进行处理的 HTTP 回调机制。(将自定义的webhook 的server注册到k8s，k8s拦截到请求 通过配置的回调将请求转发到webhook server)

可以定义两种类型的准入 **Webhook**， 即验证性质的准入 **validating admission webhook** 和变更性质**mutating admission webhook**。 变更性质的准入 Webhook 会先被调用。它们可以修改发送到 API 服务器的对象以执行自定义的设置默认值操作。

> 简单的校验我们可以直接使用***CRD***的***scheme***校验，但是复杂一点的需求 就需要 *webhook*

使用

https://kubernetes.io/zh-cn/docs/reference/access-authn-authz/extensible-admission-controllers/#write-an-admission-webhook-server

- 编写一个准入 Webhook 服务器

- 部署准入 Webhook 服务

  Webhook 服务器通过 deployment API 部署在 Kubernetes 集群中。该测试还将创建一个 Service 作为 Webhook 服务器的前端

- 即时配置准入 Webhook

  通过 ValidatingWebhookConfiguration 或者 MutatingWebhookConfiguration 动态配置哪些资源要被哪些准入 Webhook 处理。

```yaml
#  Pods 的创建操作定义一个 Validating Webhook
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  rules: # Webhook 将会应用的操作和资源
  - apiGroups: [""]
    apiVersions: ["v1"]
    operations: ["CREATE"]
    resources: ["pods"]
    scope: "Namespaced"
  clientConfig: # Webhook 客户端的配置信息
    service:
      namespace: "example-namespace"
      name: "example-service"
    caBundle: <CA_BUNDLE> #用于验证 Webhook 服务器证书的 CA Bundle（Base64 编码的 X.509 证书）
  admissionReviewVersions: ["v1"] # 指定 AdmissionReview 版本
  sideEffects: None #Webhook 的副作用
  timeoutSeconds: 5 #Webhook 调用超时，则根据 Webhook 的失败策略处理请求。
```

  部署到外部的**webhook server**：使用url

```
webhooks:
- name: my-webhook.example.com
  clientConfig:
    url: "https://my-webhook.example.com:9443/my-webhook-path"
```

- 对 API 服务器进行身份认证

用kubebuilder创建 （案例）

我们希望在用户创建App资源时，做一些更细的控制。

```
apiVersion: ingress.baiding.tech/v1beta1
kind: App
metadata:
  name: app-sample
spec:
  image: nginx:latest
  replicas: 3
  enable_ingress: false #默认值为false，需求为：设置为反向值;为true时，
enable_service必须为true
  enable_service: false
```

简单的校验我们可以直接使用CRD的scheme校验，但是复杂一点的需求我们使用webhook

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /home/aiedge/csiTest
    server: 192.168.20.235



 ---
 apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 192.168.20.235
```

```
  share: /home/aiedge/csiTest
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

1. 创建

```
kubebuilder create webhook --group ingress --version v1beta1 --kind App
--defaulting --programmatic-validation --conversion
#--programmatic-validation: 启用程序化验证，这允许您编写自定义验证逻辑。
#--conversion: 启用资源版本之间的自动转换。

#运行后生成api/v1beta1/app_webhook.go 及一些webhook相关的yaml
#在`api/v1beta1/app_webhook.go`中是需要自己补充实现的
```

创建之后，在main.go中会添加以下代码:

```
    if err = (&ingressv1beta1.App{}).SetupWebhookWithManager(mgr); err
!= nil {
        setupLog.Error(err, "unable to create webhook", "webhook",
"App")
        os.Exit(1)
    }
```

同时会生成下列文件，主要有：

- `api/v1beta1/app_webhook.go` webhook对应的handler，我们添加业务逻辑的地方
- `api/v1beta1/webhook_suite_test.go` 测试
- `config/certmanager` 自动生成自签名的证书，用于webhook server提供https服务
- `config/webhook` 用于注册**webhook**到**k8s**中
- `config/crd/patches`为conversion自动注入caBoundle
- `config/default/manager_webhook_patch.yaml` 让manager的deployment支持webhook请求，把**`config/certmanager`** 证书挂载进去
- `config/default/webhookcainjection_patch.yaml`为webhook server拿到证书注入caBoundle

注入caBoundle由cert-manager的ca-injector 组件实现

2. 修改配置

   为了支持webhook，我们需要修改config/default/kustomization.yaml将相应的配置打开，具体可参考注释。

3. **webhook业务逻辑**

   设置enable_ingress的默认值，校验enable_service的值

4. 安装cert-manager

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.8.0/cert-manager.yaml
```

5. 部署
6. 验证

案例——kubebuilder

https://github.com/baidingtech/operator-lesson-demo/blob/main/kubebuilder-demo/docs/%E5%AE%9E%E7%8E%B0controller%E9%80%BB%E8%BE%91.md

235:/home/aiedge/kubebuilder-demo/kubebuilder-demo

版本适配

Kubebuilder	controller-runtime	Kubernetes
v1.x	v0.1.x	v1.13
v2.x	v0.2.x - v0.6.x	v1.14 - v1.18
v3.x	v0.7.x - ?	v1.19 - ?

Latest version matched:

- controller-runtime v0.X <-> Kubernetes v0.{X+12}

Kubernetes	client-go	controller-runtime
v1.4	v1.4	--
v1.5	v1.5, v2.0	--
v1.6 – v1.13	v3.0 – v10.0	--
v1.14	v11.0	v0.2
v1.15	v12.0, v0.15	v0.3
v1.16-v1.28	v0.16 – v0.28	v0.4 – v0.16

参考

https://able8.medium.com/how-to-write-a-kubernetes-custom-controller-622841d1d3f6

https://github.com/chenzongshu/Kubernetes/blob/master/%E8%87%AA%E5%B7%B1%E5%8A%A8%E6%89%8B%E5%86%99controller.md

※ https://www.cnblogs.com/alisystemsoftware/p/11580202.html