**K8S中一些基本的概念**

**控制器模式与声明式 API**

**控制器模式**

K8s 作为一个“容器编排”平台，其核心的功能是编排，Pod 作为 K8s 调度的最小单位,具备很多属性和字段，K8s 的编排正是通过一个个控制器根据被控制对象的属性和字段来实现。

下边是一个简单的例子

apiVersion: apps/v1

kind: Deployment

metadata:

name: test

spec:

selector:

matchLabels:

app: test

replicas: 2

template:

metadata:

labels:

app: test

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

K8s 集群在部署时包含了 Controllers 组件，里面对于每个 build-in 的资源类型（比如 Deployments, Statefulset, CronJob, ...）都有对应的 Controller，基本是 1:1 的关系。上面的例子中，Deployment 资源创建之后，对应的 Deployment Controller 编排动作很简单，确保携带了 app=test 的 Pod 个数永远等于 2，Pod 由 template 部分定义，具体来说，K8s 里面是 kube-controller-manager 这个组件在做这件事，可以看下 K8s 项目的 pkg/controller 目录，里面包含了所有控制器，都以独有的方式负责某种编排功能，但是它们都遵循一个通用编排模式，即：调谐循环（Reconcile loop），就是一个无限循环（实际是事件驱动+定时同步来实现，不是无脑循环）不断地对比期望状态和实际状态，如果有出入则进行 Reconcile（调谐）逻辑将实际状态调整为期望状态。期望状态就是我们的对象定义（通常是 YAML 文件），实际状态是集群里面当前的运行状态（通常来自于 K8s 集群内外相关资源的状态汇总），控制器的编排逻辑主要是第三步做的，这个操作被称为调谐（Reconcile），整个控制器调谐的过程称为“Reconcile Loop”，调谐的最终结果一般是对被控制对象的某种写操作，比如增/删/改 Pod。

在控制器中定义被控制对象是通过“模板”完成的，比如 Deployment 里面的 template 字段里的内容跟一个标准的 Pod 对象的 API 定义一样，所有被这个 Deployment 管理的 Pod 实例，都是根据这个 template 字段的创建的，这就是 PodTemplate，一个控制对象的定义一般是由上半部分的控制定义（期望状态），加上下半部分的被控制对象的模板组成。

**声明式 API**

所谓声明式就是“告诉 K8s 你要什么，而不是告诉它怎么做的命令”，一个很熟悉的例子就是 SQL，你“告诉 DB 根据条件和各类算子返回数据，而不是告诉它怎么遍历，过滤，聚合”。在 K8s 里面，声明式的体现就是 kubectl apply 命令，在对象创建和后续更新中一直使用相同的 apply 命令，告诉 K8s 对象的终态即可，底层是通过执行了一个对原有 API 对象的 PATCH 操作来实现的，可以一次性处理多个写操作，具备 Merge 能力 diff 出最终的 PATCH，而命令式一次只能处理一个写请求。

**CustomResourceDefinition**

声明式 API 让 K8s 的“容器编排”简单明了，而控制器（以及容器运行时，存储，网络模型等）是这个过程的核心。说到这里，就会有人希望也能像 build-in 资源一样构建自己的自定义资源（CRD-Customize Resource Definition），然后为自定义资源写一个对应的控制器，推出自己的声明式 API。K8s 提供了 CRD 的扩展方式来满足用户这一需求，而且由于这种扩展方式十分灵活，对于用户来说，实现 CRD 扩展主要做两件事：

1. 编写 CRD 并将其部署到 K8s 集群里；

这一步的作用就是让 K8s 知道有这个资源及其结构属性，在用户提交该自定义资源的定义时（通常是 YAML 文件定义），K8s 能够成功校验该资源并创建出对应的 Go struct 进行持久化，同时触发控制器的调谐逻辑。

1. 编写 Controller 并将其部署到 K8s 集群里。

这一步的作用就是实现调谐逻辑。

**二、基于sample-controller实现自定义CRD控制器（Operator）**

sample-controller 是 kubernetes 官方提供的 CRD Controller 样例实现：

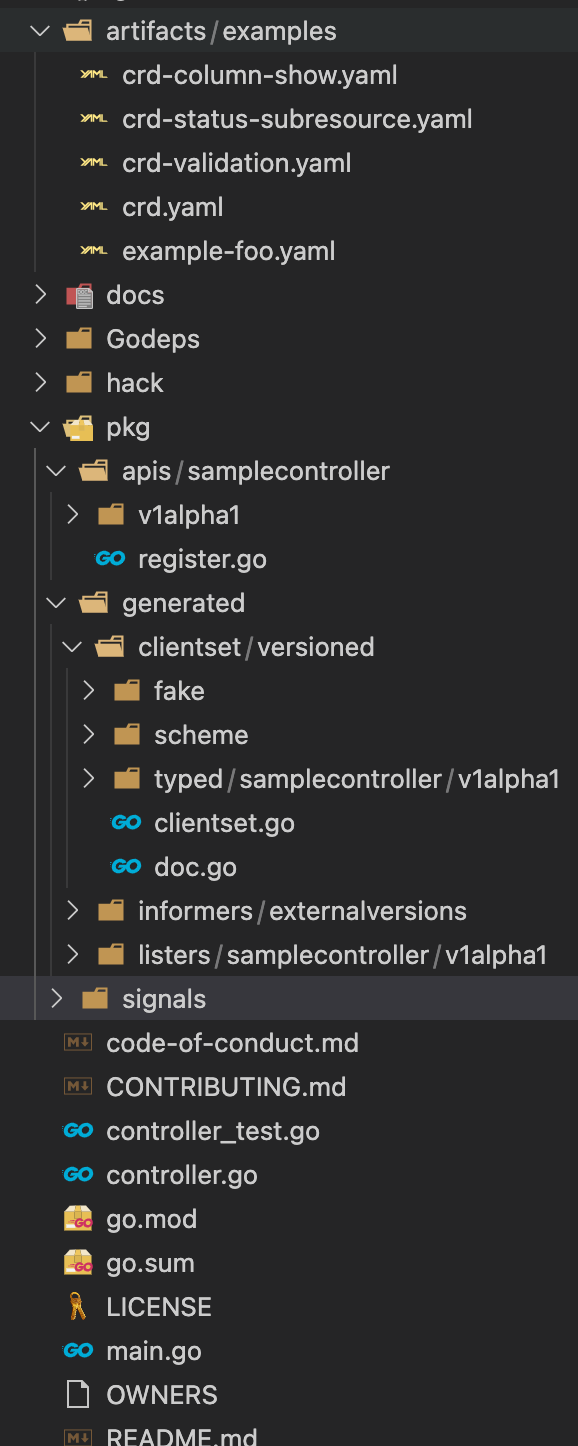
Git 地址：<https://github.com/kubernetes/sample-controller>

下面，我们就以该样例程序为例，研究如何实现自定义 CRD 控制器

1 下载 sample-controller 源码到 ~/Project/gowork/sample-controller 路径下

git clone <https://github.com/kubernetes/sample-controller.git>

然后我们看下项目的目录结构



2 编译  sample-controller

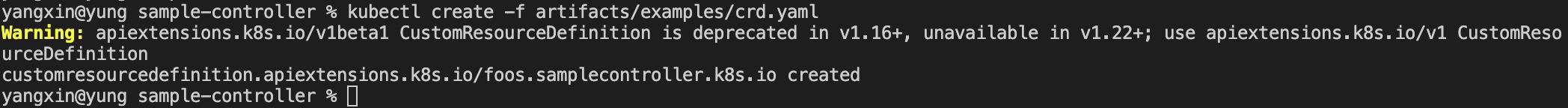
go build -o sample-controller .

3 运行  sample-controller

./sample-controller -kubeconfig=$HOME/.kube/config

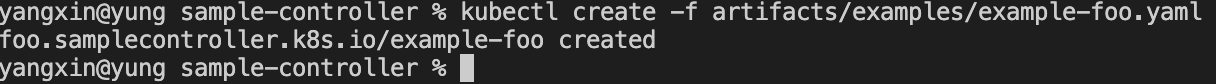
4 创建/注册 CRD 资源

kubectl create -f artifacts/examples/crd.yaml



5 创建 Foo 对象

kubectl create -f artifacts/examples/example-foo.yaml



6 查看 Foo 的部署情况

图形用户界面, 文本, 应用程序

描述已自动生成

7 查看 crds 的创建情况

文本

描述已自动生成

8 查看 pods 的运行情况

文本

描述已自动生成

9现在基本的构建流程已经了解，那么接下来看下分析下源码

9.1首先分析下 CRD 资源定义

文件位置 artifacts/examples/crd.yaml

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

# 名称必须与下面的spec字段匹配，格式为: <plural>.<group> 可通过 kubectl get crds 查看

name: foos.samplecontroller.k8s.io

spec:

# 用于REST API的组名称: /apis/<group>/<version>

group: samplecontroller.k8s.io

# 此CustomResourceDefinition支持的版本列表

version: v1alpha1

names:

# kind字段使用驼峰命名规则. 资源清单使用如此

kind: Foo

# 在CLI(shell界面输入的参数)上用作别名并用于显示的单数名称

singular: foo

# URL中使用的复数名称: /apis/<group>/<version>/<plural>

plural: foos

# 短名称允许短字符串匹配CLI上的资源，意识就是能通过kubectl 在查看资源的时候使用该资源的简名称来获取。

shortNames:

- fo

# categories字段指定自定义资源所属的组

categories:

- all

# 指定crd资源作用范围在命名空间或集群

scope: Namespaced

该定义文件，声明了一种名为 Foo 的资源，告诉 API Server，有一种资源叫做 Foo。该 Foo 资源将被 sample-controller 所监听，并对其相关事件进行处理

9.2 然后，看下创建 Foo 资源的 YAML 定义

     文件位置 artifacts/examples/example-foo.yaml

apiVersion: samplecontroller.k8s.io/v1alpha1

kind: Foo

metadata:

name: example-foo

spec:

deploymentName: example-foo

replicas: 1

该文件声明要创建的资源为 Foo 类型，副本数为 1，这个创建事件会被 sample-controller 拦截和处理

9.3 最后，重点分析 sample-controller 的实现逻辑

a.  首先，找到入口函数 sample-controller/main.go#main

func main() {

// 初始化日志包

klog.InitFlags(nil)

// 解析命令行参数

flag.Parse()

// 设置信号，以便我们更好地处理第一个停止信号

stopCh := signals.SetupSignalHandler()

// 从master和kubeconfig文件中生成k8s的配置信息

cfg, err := clientcmd.BuildConfigFromFlags(masterURL, kubeconfig)

if err != nil {

klog.Fatalf("Error building kubeconfig: %s", err.Error())

}

// 通过传递的k8s配置信息生成一个k8s集群资源控制对象（包含默认的资源对象）

kubeClient, err := kubernetes.NewForConfig(cfg)

if err != nil {

klog.Fatalf("Error building kubernetes clientset: %s", err.Error())

}

// 通过传递的k8s配置信息生成一个自定义的资源控制对象（只包含自定义的资源）

exampleClient, err := clientset.NewForConfig(cfg)

if err != nil {

klog.Fatalf("Error building example clientset: %s", err.Error())

}

// 基于 Client、Informer 初始化自定义 Controller，监听 Deployment 以及 Foos 资源变化

kubeInformerFactory := kubeinformers.NewSharedInformerFactory(kubeClient, time.Second\*30)

exampleInformerFactory := informers.NewSharedInformerFactory(exampleClient, time.Second\*30)

// 开启控制器

controller := NewController(kubeClient, exampleClient,

kubeInformerFactory.Apps().V1().Deployments(),

exampleInformerFactory.Samplecontroller().V1alpha1().Foos(),

)

// notice that there is no need to run Start methods in a separate goroutine. (i.e. go kubeInformerFactory.Start(stopCh)

// Start method is non-blocking and runs all registered informers in a dedicated goroutine.

kubeInformerFactory.Start(stopCh)

exampleInformerFactory.Start(stopCh)

if err = controller.Run(2, stopCh); err != nil {

klog.Fatalf("Error running controller: %s", err.Error())

}

}

大体流程就是：

-  读取 kubeconfig 配置，构造用于事件监听的 Kubernetes Client这里创建了两个，一个监听普通事件，一个监听 Foo 事件

-  基于 Client 构造监听相关的 informer

-  基于 Client、Informer 初始化自定义 Controller，监听 Deployment 以及 Foos 资源变化

-  开启 Controller

b.紧接着，我们看下 Controller 是如何处理事件的

代码位置 sample-controller/controller.go

**首先**，看下 Controller 的结构定义

// Controller is the controller implementation for Foo resources

type Controller struct {

// kubeclientset is a standard kubernetes clientset

kubeclientset kubernetes.Interface

// sampleclientset is a clientset for our own API group

sampleclientset clientset.Interface

deploymentsLister appslisters.DeploymentLister

deploymentsSynced cache.InformerSynced

foosLister listers.FooLister

foosSynced cache.InformerSynced

// workqueue is a rate limited work queue. This is used to queue work to be

// processed instead of performing it as soon as a change happens. This

// means we can ensure we only process a fixed amount of resources at a

// time, and makes it easy to ensure we are never processing the same item

// simultaneously in two different workers.

workqueue workqueue.RateLimitingInterface

// recorder is an event recorder for recording Event resources to the

// Kubernetes API.

recorder record.EventRecorder

}

Controller 的关键成员即两个事件的 Listener（appslisters.DeploymentLister、listers.FooLister），这两个成员将由 main 函数传入参数进行初始化。此外，为了缓冲事件处理，这里使用队列暂存事件，相关成员即为 workqueue.RateLimitingInterface record.EventRecorder 用于记录事件

**接着**，分析 Controller 的构造过程，代码如下：

// NewController returns a new sample controller

func NewController(

kubeclientset kubernetes.Interface,

sampleclientset clientset.Interface,

deploymentInformer appsinformers.DeploymentInformer,

fooInformer informers.FooInformer) \*Controller {

// Create event broadcaster

// Add sample-controller types to the default Kubernetes Scheme so Events can be

// logged for sample-controller types.

// 将 sample-controller 的类型信息（Foo）添加到默认 Kubernetes Scheme，以便能够记录到其事件

utilruntime.Must(samplescheme.AddToScheme(scheme.Scheme))

klog.V(4).Info("Creating event broadcaster")

eventBroadcaster := record.NewBroadcaster()

eventBroadcaster.StartStructuredLogging(0)

eventBroadcaster.StartRecordingToSink(&typedcorev1.EventSinkImpl{Interface: kubeclientset.CoreV1().Events("")})

recorder := eventBroadcaster.NewRecorder(scheme.Scheme, corev1.EventSource{Component: controllerAgentName})

controller := &Controller{

kubeclientset: kubeclientset,

sampleclientset: sampleclientset,

deploymentsLister: deploymentInformer.Lister(),

deploymentsSynced: deploymentInformer.Informer().HasSynced,

foosLister: fooInformer.Lister(),

foosSynced: fooInformer.Informer().HasSynced,

workqueue: workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), "Foos"),

recorder: recorder,

}

klog.Info("Setting up event handlers")

// Set up an event handler for when Foo resources change

fooInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{

AddFunc: controller.enqueueFoo,

UpdateFunc: func(old, new interface{}) {

controller.enqueueFoo(new)

},

})

// Set up an event handler for when Deployment resources change. This

// handler will lookup the owner of the given Deployment, and if it is

// owned by a Foo resource will enqueue that Foo resource for

// processing. This way, we don't need to implement custom logic for

// handling Deployment resources. More info on this pattern:

// https://github.com/kubernetes/community/blob/8cafef897a22026d42f5e5bb3f104febe7e29830/contributors/devel/controllers.md

deploymentInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{

AddFunc: controller.handleObject,

UpdateFunc: func(old, new interface{}) {

newDepl := new.(\*appsv1.Deployment)

oldDepl := old.(\*appsv1.Deployment)

if newDepl.ResourceVersion == oldDepl.ResourceVersion {

// Periodic resync will send update events for all known Deployments.

// Two different versions of the same Deployment will always have different RVs.

return

}

controller.handleObject(new)

},

DeleteFunc: controller.handleObject,

})

return controller

}

初始化 Controller 大体过程如下：

-  将 sample-controller 的类型信息（Foo）添加到默认 Kubernetes Scheme，以便能够记录到其事件

-  基于新 Scheme 创建一个事件记录 recorder ，用于记录来自 “sample-controller” 的事件

-  基于函数入参及刚刚构造的 recorder，初始化 Controller

-  设置对 Foo 资源变化的事件处理函数（Add、Update 均通过 enqueueFoo 处理）

-  设置对 Deployment 资源变化的事件处理函数（Add、Update、Delete 均通过 handleObject 处理）

-  返回初始化的 Controller

**三、基于 Kubebuidler 实现自定义 CRD 控制器（Operator）**

[Kubebuilder](https://github.com/kubernetes-sigs/kubebuilder) 是一个使用 CRDs 构建 K8s API 的 SDK，主要是：

* 提供脚手架工具初始化 CRDs 工程，自动生成 boilerplate 代码和配置；
* 提供代码库封装底层的 K8s go-client；

方便用户从零开始开发 CRDs，Controllers 和 Admission Webhooks 来扩展 K8s。

**Kubebuilder 核心概念**

**GVKs&GVRs**

GVK = GroupVersionKind，GVR = GroupVersionResource。

**API Group & Versions（GV）**

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。

**Kinds & Resources**

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](https://kubernetes.io/docs/reference/kubectl/overview/#resource-types)），一般来说 Kinds 和 Resources 是 1:1 的，比如 pods Resource 对应 Pod Kind，但是有时候相同的 Kind 可能对应多个 Resources，比如 Scale Kind 可能对应很多 Resources：deployments/scale，replicasets/scale，对于 CRD 来说，只会是 1:1 的关系。  
 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。

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。GVK/GVR 就是 K8s 资源的坐标，是我们创建/删除/修改/读取资源的基础。

**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，比如说我们给定一个 Scheme: "tutotrial.kubebuilder.io/api/v1".CronJob{} 这个 Go type 映射到 batch.tutotrial.kubebuilder.io/v1 的 CronJob GVK，那么从 Api Server 获取到下面的 JSON:

{

"kind": "CronJob",

"apiVersion": "batch.tutorial.kubebuilder.io/v1",

...

}

就能构造出对应的 Go type了，通过这个 Go type 也能正确地获取 GVR 的一些信息，控制器可以通过该 Go type 获取到期望状态以及其他辅助信息进行调谐逻辑。

**Manager**

Kubebuilder 的核心组件，具有 3 个职责：

* 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。

**Cache**

Kubebuilder 的核心组件，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，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。

**Controller**

Kubebuidler 为我们生成的脚手架文件，我们只需要实现 Reconcile 方法即可。

**Clients**

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，其中查询功能实际查询是本地的 Cache，写操作直接访问 Api Server。

**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。

**Finalizer**

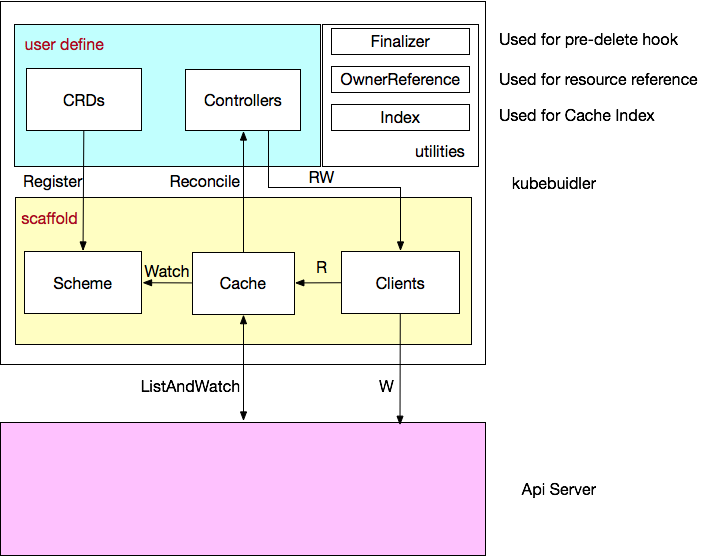
在一般情况下，如果资源被删除之后，我们虽然能够被触发删除事件，但是这个时候从 Cache 里面无法读取任何被删除对象的信息，这样一来，导致很多垃圾清理工作因为信息不足无法进行，K8s 的 Finalizer 字段用于处理这种情况。在 K8s 中，只要对象 ObjectMeta 里面的 Finalizers 不为空，对该对象的 delete 操作就会转变为 update 操作，具体说就是 update  deletionTimestamp 字段，其意义就是告诉 K8s 的 GC“在deletionTimestamp 这个时刻之后，只要 Finalizers 为空，就立马删除掉该对象”。

所以一般的使用姿势就是在创建对象时把 Finalizers 设置好（任意 string），然后处理 DeletionTimestamp 不为空的 update 操作（实际是 delete），根据 Finalizers 的值执行完所有的 pre-delete hook（此时可以在 Cache 里面读取到被删除对象的任何信息）之后将 Finalizers 置为空即可。

**OwnerReference**

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。

所有概念集合在一起如图 1 所示：



1 创建一个新的目录 operator-kubebuilder

mkdir -p operator-kubebuilder

文本

描述已自动生成

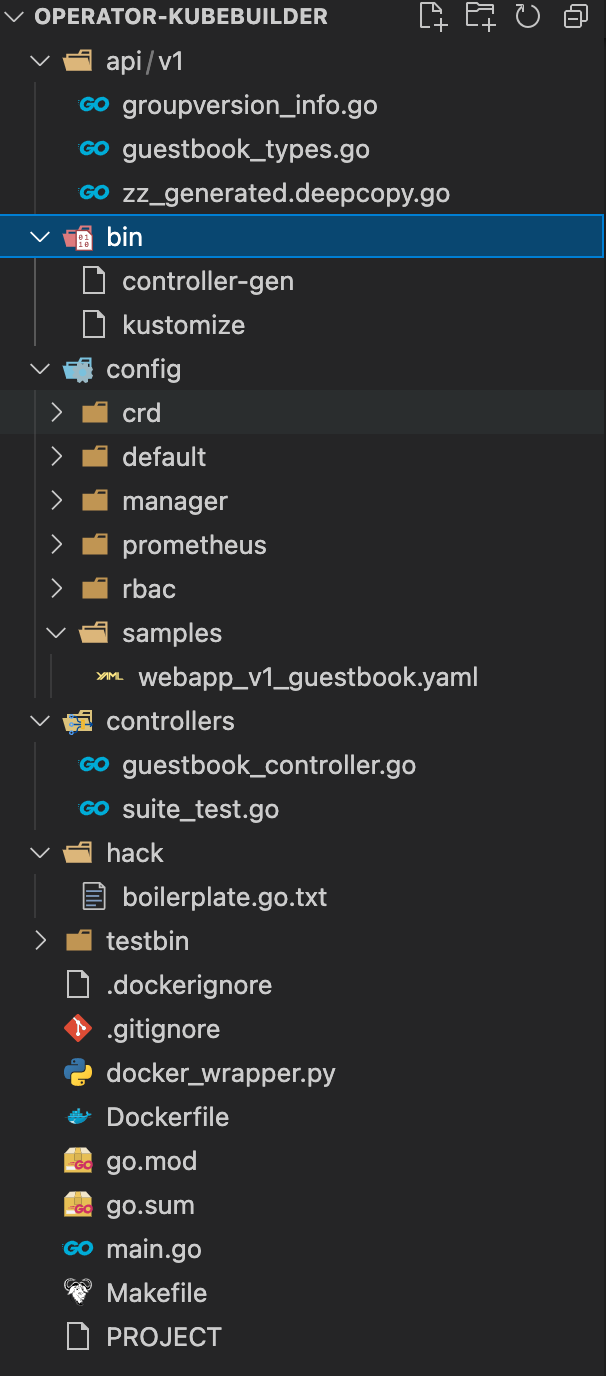
2 使用 go mod init yung.com/operator 初始化一个go工作目录

文本

描述已自动生成

3 使用 kubebuilder init --domain yung.com --repo yung.com/operator --skip-go-version-check 初始化项目

项目结构如下：



4 使用 kubebuilder create api --group webapp --version v1 --kind Guestbook 创建 API

5 向集群中安装 crds

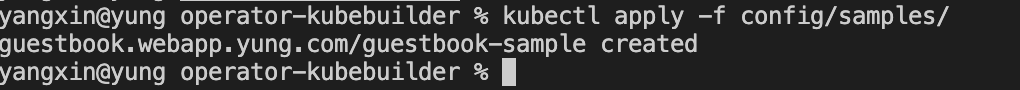
make install

文本

描述已自动生成

6 安装资源实例

kubectl apply -f config/samples/



7 在集群上运行自定义的资源类型

make docker-build docker-push IMG=<some-registry>/<project-name>:tag

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

**文本

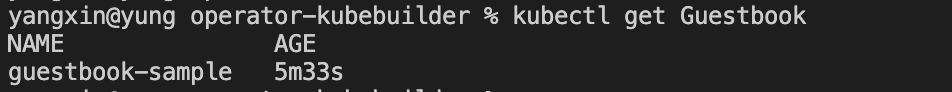
描述已自动生成**

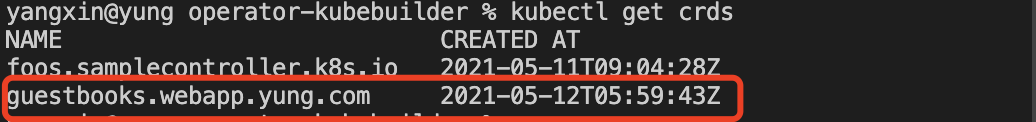
**文本

中度可信度描述已自动生成**

**文本

描述已自动生成**

****

****

8 从集群中卸载自定义资源类型

make uninstall

文本

描述已自动生成

9 从集群中卸载自定义控制器

make undeploy

文本

描述已自动生成

10 kubebuilder 官方文档 <https://book.kubebuilder.io>

源码分析：

**从 main.go 开始**

Kubebuilder 创建的 main.go 是整个项目的入口，逻辑十分简单：

var (

scheme = runtime.NewScheme()

setupLog = ctrl.Log.WithName("setup")

)

func init() {

// 注册 scheme 到 k8s

utilruntime.Must(clientgoscheme.AddToScheme(scheme))

// 注册 scheme 到 operator

utilruntime.Must(webappv1.AddToScheme(scheme))

//+kubebuilder:scaffold:scheme

}

func main() {

var metricsAddr string

var enableLeaderElection bool

var probeAddr string

flag.StringVar(&metricsAddr, "metrics-bind-address", ":8080", "The address the metric endpoint binds to.")

flag.StringVar(&probeAddr, "health-probe-bind-address", ":8081", "The address the probe endpoint binds to.")

flag.BoolVar(&enableLeaderElection, "leader-elect", false,

"Enable leader election for controller manager. "+

"Enabling this will ensure there is only one active controller manager.")

opts := zap.Options{

Development: true,

}

opts.BindFlags(flag.CommandLine)

flag.Parse()

ctrl.SetLogger(zap.New(zap.UseFlagOptions(&opts)))

// 初始化了一个 Manager

mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{

Scheme: scheme,

MetricsBindAddress: metricsAddr,

Port: 9443,

HealthProbeBindAddress: probeAddr,

LeaderElection: enableLeaderElection,

LeaderElectionID: "8482ba04.yung.com",

})

if err != nil {

setupLog.Error(err, "unable to start manager")

os.Exit(1)

}

// 初始化一个调协器

guestBookReconciler := &controllers.GuestbookReconciler{

Client: mgr.GetClient(),

Log: ctrl.Log.WithName("controllers").WithName("Guestbook"),

Scheme: mgr.GetScheme(),

}

// 给调协器设置一个Manager

if err = guestBookReconciler.SetupWithManager(mgr); err != nil {

setupLog.Error(err, "unable to create controller", "controller", "Guestbook")

os.Exit(1)

}

//+kubebuilder:scaffold:builder

// Manager的存活检查

if err := mgr.AddHealthzCheck("healthz", healthz.Ping); err != nil {

setupLog.Error(err, "unable to set up health check")

os.Exit(1)

}

// Manager的就绪检查

if err := mgr.AddReadyzCheck("readyz", healthz.Ping); err != nil {

setupLog.Error(err, "unable to set up ready check")

os.Exit(1)

}

// 启动Manager

setupLog.Info("starting manager")

if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {

setupLog.Error(err, "problem running manager")

os.Exit(1)

}

}

可以看到在 init 方法里面我们将 appsv1alpha1 注册到 Scheme 里面去了，这样一来 Cache 就知道 watch 谁了，main 方法里面的逻辑基本都是 Manager 的：

1. 初始化了一个 Manager；
2. 将 Manager 的 Client 传给 Controller，并且调用 SetupWithManager 方法传入 Manager 进行 Controller 的初始化；
3. 启动 Manager。

我们的核心就是看这 3 个流程。

### Manager 初始化

Manager 初始化代码如下：

// New returns a new Manager for creating Controllers.

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

}

clientOptions := client.Options{Scheme: options.Scheme, Mapper: mapper}

apiReader, err := client.New(config, clientOptions)

if err != nil {

return nil, err

}

writeObj, err := options.ClientBuilder.

WithUncached(options.ClientDisableCacheFor...).

Build(cache, config, clientOptions)

if err != nil {

return nil, err

}

.. .. .. .. ..

return &controllerManager{

config: config,

scheme: options.Scheme,

cache: cache,

fieldIndexes: cache,

client: writeObj,

apiReader: apiReader,

recorderProvider: recorderProvider,

resourceLock: resourceLock,

mapper: mapper,

metricsListener: metricsListener,

metricsExtraHandlers: metricsExtraHandlers,

logger: options.Logger,

elected: make(chan struct{}),

port: options.Port,

host: options.Host,

certDir: options.CertDir,

leaseDuration: \*options.LeaseDuration,

renewDeadline: \*options.RenewDeadline,

retryPeriod: \*options.RetryPeriod,

healthProbeListener: healthProbeListener,

readinessEndpointName: options.ReadinessEndpointName,

livenessEndpointName: options.LivenessEndpointName,

gracefulShutdownTimeout: \*options.GracefulShutdownTimeout,

internalProceduresStop: make(chan struct{}),

}, nil

}

可以看到主要是创建 Cache 与 Clients：

**创建 Cache**

Cache 初始化代码如下：

// New initializes and returns a new Cache.

func New(config \*rest.Config, opts Options) (Cache, error) {

opts, err := defaultOpts(config, opts)

if err != nil {

return nil, err

}

im := internal.NewInformersMap(config, opts.Scheme, opts.Mapper, \*opts.Resync, opts.Namespace)

return &informerCache{InformersMap: im}, nil

}

// NewInformersMap creates a new InformersMap that can create informers for

// both structured and unstructured objects.

func NewInformersMap(config \*rest.Config,

scheme \*runtime.Scheme,

mapper meta.RESTMapper,

resync time.Duration,

namespace string) \*InformersMap {

return &InformersMap{

structured: newStructuredInformersMap(config, scheme, mapper, resync, namespace),

unstructured: newUnstructuredInformersMap(config, scheme, mapper, resync, namespace),

metadata: newMetadataInformersMap(config, scheme, mapper, resync, namespace),

Scheme: scheme,

}

}

可以看到 Cache 主要就是创建了 InformersMap，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。

**创建 Clients**

func New(config \*rest.Config, options Options) (Client, error) {

if config == nil {

return nil, fmt.Errorf("must provide non-nil rest.Config to client.New")

}

// Init a scheme if none provided

if options.Scheme == nil {

options.Scheme = scheme.Scheme

}

// Init a Mapper if none provided

if options.Mapper == nil {

var err error

options.Mapper, err = apiutil.NewDynamicRESTMapper(config)

if err != nil {

return nil, err

}

}

clientcache := &clientCache{

config: config,

scheme: options.Scheme,

mapper: options.Mapper,

codecs: serializer.NewCodecFactory(options.Scheme),

structuredResourceByType: make(map[schema.GroupVersionKind]\*resourceMeta),

unstructuredResourceByType: make(map[schema.GroupVersionKind]\*resourceMeta),

}

rawMetaClient, err := metadata.NewForConfig(config)

if err != nil {

return nil, fmt.Errorf("unable to construct metadata-only client for use as part of client: %w", err)

}

c := &client{

typedClient: typedClient{

cache: clientcache,

paramCodec: runtime.NewParameterCodec(options.Scheme),

},

unstructuredClient: unstructuredClient{

cache: clientcache,

paramCodec: noConversionParamCodec{},

},

metadataClient: metadataClient{

client: rawMetaClient,

restMapper: options.Mapper,

},

scheme: options.Scheme,

mapper: options.Mapper,

}

return c, nil

}

**Controller 初始化**

下面看看 Controller 的启动：

// SetupWithManager sets up the controller with the Manager.

func (r \*GuestbookReconciler) SetupWithManager(mgr ctrl.Manager) error {

return ctrl.NewControllerManagedBy(mgr).

For(&webappv1.Guestbook{}).

Complete(r)

}

使用的是 Builder 模式，NewControllerManagerBy 和 For 方法都是给 Builder 传参，最重要的是最后一个方法 Complete，其逻辑是：

// Complete builds the Application ControllerManagedBy.

func (blder \*Builder) Complete(r reconcile.Reconciler) error {

\_, err := blder.Build(r)

return err

}

// Build builds the Application ControllerManagedBy and returns the Controller it created.

func (blder \*Builder) Build(r reconcile.Reconciler) (controller.Controller, error) {

if r == nil {

return nil, fmt.Errorf("must provide a non-nil Reconciler")

}

if blder.mgr == nil {

return nil, fmt.Errorf("must provide a non-nil Manager")

}

if blder.forInput.err != nil {

return nil, blder.forInput.err

}

// Checking the reconcile type exist or not

if blder.forInput.object == nil {

return nil, fmt.Errorf("must provide an object for reconciliation")

}

// Set the Config

blder.loadRestConfig()

// 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.ctrl, nil

}

主要是看看 doController 和 doWatch 方法：

func (blder \*Builder) doController(r reconcile.Reconciler) error {

ctrlOptions := blder.ctrlOptions

if ctrlOptions.Reconciler == nil {

ctrlOptions.Reconciler = r

}

// Retrieve the GVK from the object we're reconciling

// to prepopulate logger information, and to optionally generate a default name.

gvk, err := getGvk(blder.forInput.object, blder.mgr.GetScheme())

if err != nil {

return err

}

// Setup the logger.

if ctrlOptions.Log == nil {

ctrlOptions.Log = blder.mgr.GetLogger()

}

ctrlOptions.Log = ctrlOptions.Log.WithValues("reconciler group", gvk.Group, "reconciler kind", gvk.Kind)

// Build the controller and return.

blder.ctrl, err = newController(blder.getControllerName(gvk), blder.mgr, ctrlOptions)

return err

}

// New returns a new Controller registered with the Manager. The Manager will ensure that shared Caches have

// been synced before the Controller is Started.

func New(name string, mgr manager.Manager, options Options) (Controller, error) {

c, err := NewUnmanaged(name, mgr, options)

if err != nil {

return nil, err

}

// Add the controller as a Manager components

return c, mgr.Add(c)

}

// NewUnmanaged returns a new controller without adding it to the manager. The

// caller is responsible for starting the returned controller.

func NewUnmanaged(name string, mgr manager.Manager, options Options) (Controller, error) {

if options.Reconciler == nil {

return nil, fmt.Errorf("must specify Reconciler")

}

if len(name) == 0 {

return nil, fmt.Errorf("must specify Name for Controller")

}

if options.Log == nil {

options.Log = mgr.GetLogger()

}

if options.MaxConcurrentReconciles <= 0 {

options.MaxConcurrentReconciles = 1

}

if options.RateLimiter == nil {

options.RateLimiter = workqueue.DefaultControllerRateLimiter()

}

// Inject dependencies into Reconciler

if err := mgr.SetFields(options.Reconciler); err != nil {

return nil, err

}

// Create controller with dependencies set

return &controller.Controller{

Do: options.Reconciler,

MakeQueue: func() workqueue.RateLimitingInterface {

return workqueue.NewNamedRateLimitingQueue(options.RateLimiter, name)

},

MaxConcurrentReconciles: options.MaxConcurrentReconciles,

SetFields: mgr.SetFields,

Name: name,

Log: options.Log.WithName("controller").WithName(name),

}, nil

}

该方法初始化了一个 Controller，传入了一些很重要的参数：

* Do：Reconcile 逻辑；
* MakeQueue：Watch 资源的 CUD 事件缓存；

**doWatch 方法**

unc (blder \*Builder) doWatch() error {

// Reconcile type

typeForSrc, err := blder.project(blder.forInput.object, blder.forInput.objectProjection)

if err != nil {

return err

}

src := &source.Kind{Type: typeForSrc}

hdler := &handler.EnqueueRequestForObject{}

allPredicates := append(blder.globalPredicates, blder.forInput.predicates...)

if err := blder.ctrl.Watch(src, hdler, allPredicates...); err != nil {

return err

}

// Watches the managed types

for \_, own := range blder.ownsInput {

typeForSrc, err := blder.project(own.object, own.objectProjection)

if err != nil {

return err

}

src := &source.Kind{Type: typeForSrc}

hdler := &handler.EnqueueRequestForOwner{

OwnerType: blder.forInput.object,

IsController: true,

}

allPredicates := append([]predicate.Predicate(nil), blder.globalPredicates...)

allPredicates = append(allPredicates, own.predicates...)

if err := blder.ctrl.Watch(src, hdler, allPredicates...); err != nil {

return err

}

}

// Do the watch requests

for \_, w := range blder.watchesInput {

allPredicates := append([]predicate.Predicate(nil), blder.globalPredicates...)

allPredicates = append(allPredicates, w.predicates...)

// If the source of this watch is of type \*source.Kind, project it.

if srckind, ok := w.src.(\*source.Kind); ok {

typeForSrc, err := blder.project(srckind.Type, w.objectProjection)

if err != nil {

return err

}

srckind.Type = typeForSrc

}

if err := blder.ctrl.Watch(w.src, w.eventhandler, allPredicates...); err != nil {

return err

}

}

return nil

}

可以看到该方法对本 Controller 负责的 CRD 进行了 watch，同时底下还会 watch 本 CRD 管理的其他资源，这个 blder.ownsInput可以通过 Controller 初始化 Buidler 的 Owns 方法传入，说到 Watch 我们关心两个逻辑：

1. 注册的 handler

ype EnqueueRequestForObject struct{}

// Create implements EventHandler

func (e \*EnqueueRequestForObject) Create(evt event.CreateEvent, q workqueue.RateLimitingInterface) {

if evt.Object == nil {

enqueueLog.Error(nil, "CreateEvent received with no metadata", "event", evt)

return

}

q.Add(reconcile.Request{NamespacedName: types.NamespacedName{

Name: evt.Object.GetName(),

Namespace: evt.Object.GetNamespace(),

}})

}

// Update implements EventHandler

func (e \*EnqueueRequestForObject) Update(evt event.UpdateEvent, q workqueue.RateLimitingInterface) {

if evt.ObjectOld != nil {

q.Add(reconcile.Request{NamespacedName: types.NamespacedName{

Name: evt.ObjectOld.GetName(),

Namespace: evt.ObjectOld.GetNamespace(),

}})

} else {

enqueueLog.Error(nil, "UpdateEvent received with no old metadata", "event", evt)

}

if evt.ObjectNew != nil {

q.Add(reconcile.Request{NamespacedName: types.NamespacedName{

Name: evt.ObjectNew.GetName(),

Namespace: evt.ObjectNew.GetNamespace(),

}})

} else {

enqueueLog.Error(nil, "UpdateEvent received with no new metadata", "event", evt)

}

}

// Delete implements EventHandler

func (e \*EnqueueRequestForObject) Delete(evt event.DeleteEvent, q workqueue.RateLimitingInterface) {

if evt.Object == nil {

enqueueLog.Error(nil, "DeleteEvent received with no metadata", "event", evt)

return

}

q.Add(reconcile.Request{NamespacedName: types.NamespacedName{

Name: evt.Object.GetName(),

Namespace: evt.Object.GetNamespace(),

}})

}

可以看到 Kubebuidler 为我们注册的 Handler 就是将发生变更的对象的 NamespacedName 入队列，如果在 Reconcile 逻辑中需要判断创建/更新/删除，需要有自己的判断逻辑。

2. 注册的流程

// Watch implements controller.Controller

func (c \*Controller) Watch(src source.Source, evthdler handler.EventHandler, prct ...predicate.Predicate) error {

c.mu.Lock()

defer c.mu.Unlock()

// Inject Cache into arguments

if err := c.SetFields(src); err != nil {

return err

}

if err := c.SetFields(evthdler); err != nil {

return err

}

for \_, pr := range prct {

if err := c.SetFields(pr); err != nil {

return err

}

}

// Controller hasn't started yet, store the watches locally and return.

//

// These watches are going to be held on the controller struct until the manager or user calls Start(...).

if !c.Started {

c.startWatches = append(c.startWatches, watchDescription{src: src, handler: evthdler, predicates: prct})

return nil

}

c.Log.Info("Starting EventSource", "source", src)

return src.Start(c.ctx, evthdler, c.Queue, prct...)

}

// Start implements controller.Controller

func (c \*Controller) Start(ctx context.Context) error {

// use an IIFE to get proper lock handling

// but lock outside to get proper handling of the queue shutdown

c.mu.Lock()

if c.Started {

return errors.New("controller was started more than once. This is likely to be caused by being added to a manager multiple times")

}

// Set the internal context.

c.ctx = ctx

c.Queue = c.MakeQueue()

defer c.Queue.ShutDown() // needs to be outside the iife so that we shutdown after the stop channel is closed

err := func() error {

defer c.mu.Unlock()

// TODO(pwittrock): Reconsider HandleCrash

defer utilruntime.HandleCrash()

// NB(directxman12): launch the sources \*before\* trying to wait for the

// caches to sync so that they have a chance to register their intendeded

// caches.

for \_, watch := range c.startWatches {

c.Log.Info("Starting EventSource", "source", watch.src)

if err := watch.src.Start(ctx, watch.handler, c.Queue, watch.predicates...); err != nil {

return err

}

}

// Start the SharedIndexInformer factories to begin populating the SharedIndexInformer caches

c.Log.Info("Starting Controller")

for \_, watch := range c.startWatches {

syncingSource, ok := watch.src.(source.SyncingSource)

if !ok {

continue

}

if err := syncingSource.WaitForSync(ctx); err != nil {

// This code is unreachable in case of kube watches since WaitForCacheSync will never return an error

// Leaving it here because that could happen in the future

err := fmt.Errorf("failed to wait for %s caches to sync: %w", c.Name, err)

c.Log.Error(err, "Could not wait for Cache to sync")

return err

}

}

// All the watches have been started, we can reset the local slice.

//

// We should never hold watches more than necessary, each watch source can hold a backing cache,

// which won't be garbage collected if we hold a reference to it.

c.startWatches = nil

if c.JitterPeriod == 0 {

c.JitterPeriod = 1 \* time.Second

}

// Launch workers to process resources

c.Log.Info("Starting workers", "worker count", c.MaxConcurrentReconciles)

ctrlmetrics.WorkerCount.WithLabelValues(c.Name).Set(float64(c.MaxConcurrentReconciles))

for i := 0; i < c.MaxConcurrentReconciles; i++ {

go wait.UntilWithContext(ctx, func(ctx context.Context) {

// Run a worker thread that just dequeues items, processes them, and marks them done.

// It enforces that the reconcileHandler is never invoked concurrently with the same object.

for c.processNextWorkItem(ctx) {

}

}, c.JitterPeriod)

}

c.Started = true

return nil

}()

if err != nil {

return err

}

<-ctx.Done()

c.Log.Info("Stopping workers")

return nil

}

我们的 Handler 实际注册到 Informer 上面，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.Manager 启动**

func (cm \*controllerManager) Start(ctx context.Context) (err error) {

.. .. ..

go cm.startNonLeaderElectionRunnables()

.. .. ..

}

func (cm \*controllerManager) startNonLeaderElectionRunnables() {

cm.mu.Lock()

defer cm.mu.Unlock()

cm.waitForCache(cm.internalCtx)

// Start the non-leaderelection Runnables after the cache has synced

for \_, c := range cm.nonLeaderElectionRunnables {

// Controllers block, but we want to return an error if any have an error starting.

// Write any Start errors to a channel so we can return them

cm.startRunnable(c)

}

}

func (cm \*controllerManager) startLeaderElectionRunnables() {

cm.mu.Lock()

defer cm.mu.Unlock()

cm.waitForCache(cm.internalCtx)

// Start the leader election Runnables after the cache has synced

for \_, c := range cm.leaderElectionRunnables {

// Controllers block, but we want to return an error if any have an error starting.

// Write any Start errors to a channel so we can return them

cm.startRunnable(c)

}

cm.startedLeader = true

}

func (cm \*controllerManager) waitForCache(ctx context.Context) {

if cm.started {

return

}

// Start the Cache. Allow the function to start the cache to be mocked out for testing

if cm.startCache == nil {

cm.startCache = cm.cache.Start

}

cm.startRunnable(RunnableFunc(func(ctx context.Context) error {

return cm.startCache(ctx)

}))

// Wait for the caches to sync.

// TODO(community): Check the return value and write a test

cm.cache.WaitForCacheSync(ctx)

// TODO: This should be the return value of cm.cache.WaitForCacheSync but we abuse

// cm.started as check if we already started the cache so it must always become true.

// Making sure that the cache doesn't get started twice is needed to not get a "close

// of closed channel" panic

cm.started = true

}

主要就是启动 Cache，Controller，将整个事件流运转起来，我们下面来看看启动逻辑。

**Cache 启动**

// Start calls Run on each of the informers and sets started to true. Blocks on the context.

func (m \*InformersMap) Start(ctx context.Context) error {

go m.structured.Start(ctx)

go m.unstructured.Start(ctx)

go m.metadata.Start(ctx)

<-ctx.Done()

return nil

}

// Start calls Run on each of the informers and sets started to true. Blocks on the context.

// It doesn't return start because it can't return an error, and it's not a runnable directly.

func (ip \*specificInformersMap) Start(ctx context.Context) {

func() {

ip.mu.Lock()

defer ip.mu.Unlock()

// Set the stop channel so it can be passed to informers that are added later

ip.stop = ctx.Done()

// Start each informer

for \_, informer := range ip.informersByGVK {

go informer.Informer.Run(ctx.Done())

}

// Set started to true so we immediately start any informers added later.

ip.started = true

close(ip.startWait)

}()

<-ctx.Done()

}

func (s \*sharedIndexInformer) Run(stopCh <-chan struct{}) {

defer utilruntime.HandleCrash()

fifo := NewDeltaFIFOWithOptions(DeltaFIFOOptions{

KnownObjects: s.indexer,

EmitDeltaTypeReplaced: true,

})

cfg := &Config{

Queue: fifo,

ListerWatcher: s.listerWatcher,

ObjectType: s.objectType,

FullResyncPeriod: s.resyncCheckPeriod,

RetryOnError: false,

ShouldResync: s.processor.shouldResync,

Process: s.HandleDeltas,

WatchErrorHandler: s.watchErrorHandler,

}

func() {

s.startedLock.Lock()

defer s.startedLock.Unlock()

s.controller = New(cfg)

s.controller.(\*controller).clock = s.clock

s.started = true

}()

// Separate stop channel because Processor should be stopped strictly after controller

processorStopCh := make(chan struct{})

var wg wait.Group

defer wg.Wait() // Wait for Processor to stop

defer close(processorStopCh) // Tell Processor to stop

wg.StartWithChannel(processorStopCh, s.cacheMutationDetector.Run)

wg.StartWithChannel(processorStopCh, s.processor.run)

defer func() {

s.startedLock.Lock()

defer s.startedLock.Unlock()

s.stopped = true // Don't want any new listeners

}()

s.controller.Run(stopCh)

}

// Run begins processing items, and will continue until a value is sent down stopCh or it is closed.

// It's an error to call Run more than once.

// Run blocks; call via go.

func (c \*controller) Run(stopCh <-chan struct{}) {

defer utilruntime.HandleCrash()

go func() {

<-stopCh

c.config.Queue.Close()

}()

r := NewReflector(

c.config.ListerWatcher,

c.config.ObjectType,

c.config.Queue,

c.config.FullResyncPeriod,

)

r.ShouldResync = c.config.ShouldResync

r.clock = c.clock

if c.config.WatchErrorHandler != nil {

r.watchErrorHandler = c.config.WatchErrorHandler

}

c.reflectorMutex.Lock()

c.reflector = r

c.reflectorMutex.Unlock()

var wg wait.Group

wg.StartWithChannel(stopCh, r.Run)

wait.Until(c.processLoop, time.Second, stopCh)

wg.Wait()

}

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。

四、**基于 Operator-SDK实现自定义 CRD 控制器（Operator）**

1 创建一个新的目录 operator-sdk

文本

描述已自动生成

2 使用 go mod init github.com/yx\_yufeiyu/memcached.k8s.com

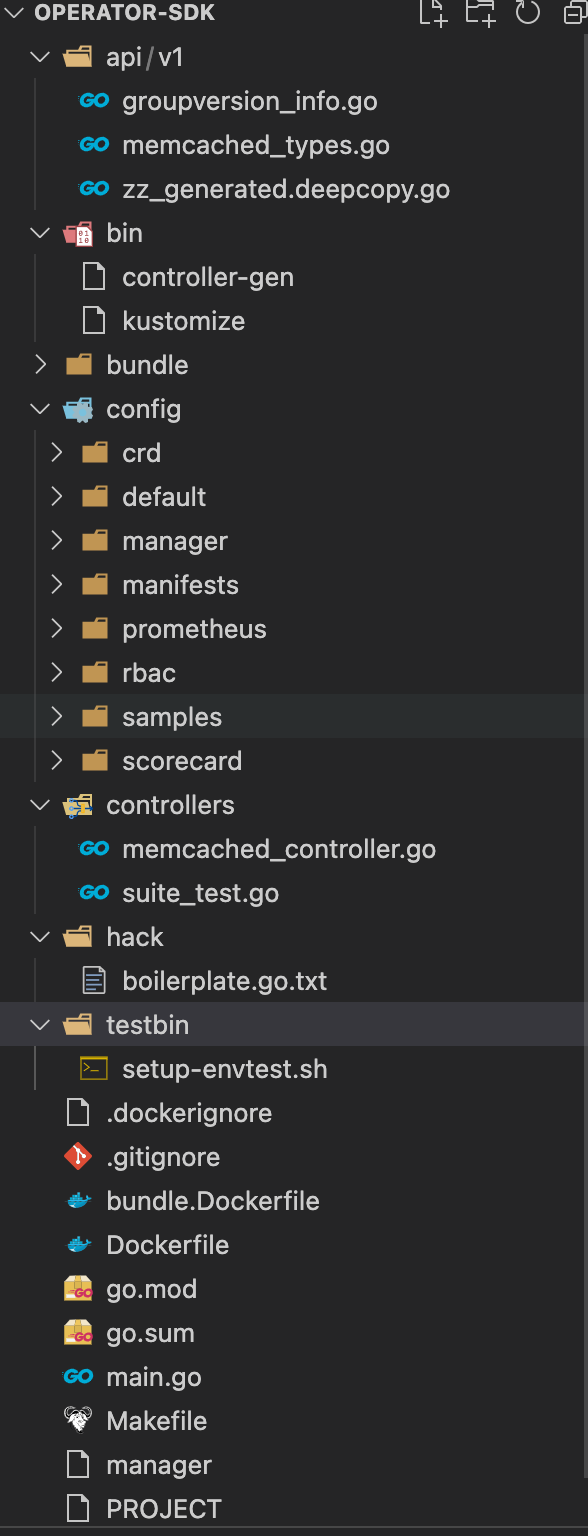
初始化一个go工作目录

3 使用 operator-sdk init --domain memcached.k8s.com --owner yung --skip-go-version-check 初始化项目

文本

描述已自动生成

项目目录如下：



4 使用 operator-sdk create api --group cache --version v1 --kind Memcached 创建 API

文本

描述已自动生成

5 构建和并推送镜像 make docker-build docker-push IMG="example.com/memcached-operator:v0.0.1"

docker build -t 646222472/memcached-operator:v0.0.1 .

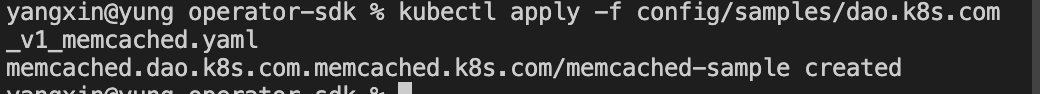
6 部署 deployment

文本

描述已自动生成

7 添加自定义资源

kubectl apply -f config/samples/dao.k8s.com\_v1\_memcached.yaml



8 查看相关资源

图形用户界面, 文本

描述已自动生成

文本

中度可信度描述已自动生成

图片包含 表格

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

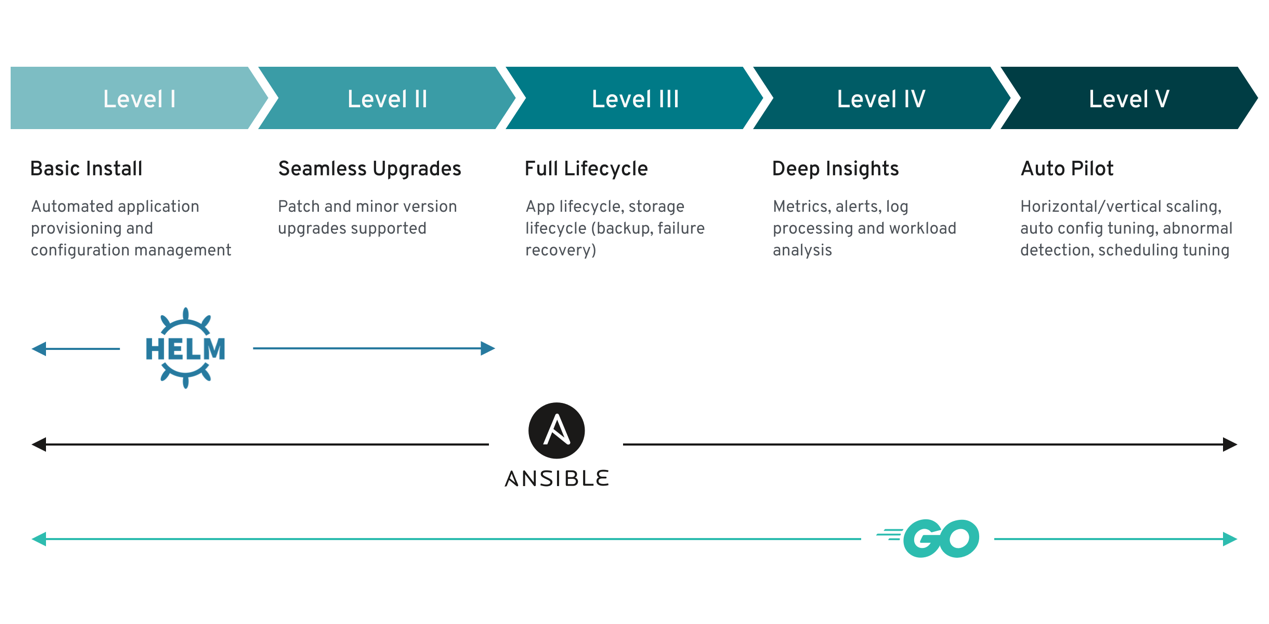
9 删除相关控制器和自定义资源

文本

描述已自动生成

10 官方资料 <https://sdk.operatorframework.io/docs/building-operators/golang/quickstart/>

11 Operator的功能等级



详细说明：<https://sdk.operatorframework.io/docs/advanced-topics/operator-capabilities/operator-capabilities/>

五、结束语

主流的 operator framework 主要有两个：kubebuilder 和 operator-sdk

两者实际上并没有本质的区别，它们都是让开发人员专注于 CRD 的业务代码开发，

它们的核心都是使用官方的 controller-tools 和 controller-runtime。不过细节上稍有不同，比如 kubebuilder 有着更为完善的测试与部署以及代码生成的脚手架等；而 operator-sdk 对 ansible operator 这类上层操作的支持更好一些。

* operator-sdk 由 CoreOS 开发和维护
* kubebuilder 由 k8s SIG 开发和维护

一个是在 k8s 领域中表现非常积极的 CoreOS，一个是 k8s SIG(k8s 亲儿子)，operator-sdk 和 kubebuilder 该如何选型呢？

那就是：选啥都行！原因是 operator-sdk 和 kubebuilder 正在进行深度的整合。

* **整合目标**

将 kubebuilder 和 operator-sdk 围绕 kubebuilder 项目框架进行整合，将 operator-sdk 的 Go operator 功能酌情移至上游，并联手维护 kube-builder，以便 kubebuilder 和 operator-sdk 都能为基于 Go 的 operator 提供相同的项目结构和命令行界面。

* **背景**

kubebuilder 和 operator-sdk 是类似的项目，旨在简化开发构建 kubernetes operator(或 controller)。这两个项目都广泛使用了 controller-runtime 和 controller-tools 项目，因此构建了类似的 Go 源文件和包结构。

* **动机**

operator-sdk 和 kubebuilder 贡献者合作改进了它们共享的上游依赖，但是与脚手架 Go operator 相关的 operator-sdk 和 kubebuilder 之间依然存在着很大的重叠。这两个项目都有用于初始化新项目并添加新 APIs 和 controllers 的样板实现的命令。

整合 kubebuilder 和 operator-sdk 动机是，与其在 Go operator 的项目脚手架上进行的重复的工作，不如在基于一个实现一起开发，这可以加快两个框架的迭代进度，并可能导致更通用的解决方案。

* **整合计划**

kubebuilder 和 operator-sdk 贡献者创建了一个[GitHub项目](https://github.com/kubernetes-sigs/kubebuilder/projects/7)，以跟踪对齐项目所需的工作。有三个主要主题。

* kubebuilder 整合 operator-sdk 代码

operator-sdk 项目包含了 Go operator 开发人员可以使用的各种特性，无论特性是基于 kubebuilder 还是 operator-sdk。这些特性将在适当的时候整合到 kubebuilder、 controller-runtime 和 controller-tools 中。其中包括:

* DynamicRESTMapper 可以使 operator 启动后动态自动地发现添加到集群的新 CRD
* 当资源的 metas.generation 字段已更改时，GenerationChangedPredicate 可以触发 recocile 事件
* 当构造默认基于 zap 的日志时，可以使用 flags helpers 提供更细粒度的配置

operator-sdk 贡献者计划开始在 kubebuilder（及相关项目）中进行与 Go operator 相关的代码的所有开发，并花费更多时间帮助 kubebuilder 贡献者维护这些项目。

* 远景

为了使 kubebuilder 更具可扩展性，社区一直在讨论一项建议，向 kubebuilder 添加扩展点以支持不同的 operator 模式。operator 模式的一个示例是使用现有库实例化 API 和控制器的插件模式。

更广泛地说，该想法是增加对基于可执行插件的扩展的支持，该扩展可以在将文件写入磁盘之前修改 kubebuilder 的脚手架，以便该项目（例如 Go 代码，kustomize 模板，项目 Makefile 和 Dockerfile）可以通过扩展自定义内容。

* 文档

即使其中很大一部分重叠，operator-sdk 和 kubebuilder 当前仍维护独立的文档。通过共同努力，operator-sdk 贡献者将基于 Go 的 operator 文档集成到 kubebuilder 文档中，并与 kubebuilder 贡献者一起保持文档的及时更新。