Artículo
Eduardo Anglada · 22 ene, 2021 Lectura de 19 min

Análisis en detalle del operador InterSystems Kubernetes: Introducción a los operadores Kubernetes

Introducción

Varios recursos nos enseñan cómo ejecutar IRIS en un clúster de Kubernetes, como Deploying an InterSystems IRIS Solution on EKS using GitHub Actions y Deploying InterSystems IRIS solution on GKE Using GitHub Actions. Estos métodos funcionan, pero requieren la creación de manifiestos para Kubernetes y Helm charts, lo que puede requerir mucho tiempo.
Para simplificar la implementación en IRIS, InterSystems desarrolló una asombrosa herramienta llamada InterSystems Kubernetes Operator (IKO). Varios recursos explican el uso de IKO, como New Video: Intersystems IRIS Kubernetes Operator e InterSystems Kubernetes Operator.

En la documentación de Kubernetes se indica que los operadores sustituyen a un operador humano ya que saben cómo lidiar con los sistemas complejos en Kubernetes. Proporcionan la configuración del sistema por medio de *recursos personalizados*. Un operador incluye un *controlador personalizado* que interpreta esta configuración y lleva a cabo los pasos establecidos para configurar y mantener funcionando correctamente tu aplicación. El controlador personalizado es un contenedor sencillo que se implementó en Kubernetes. Por lo tanto, en términos generales, todo lo que necesitas hacer para que un operador funcione es implementar un contenedor para el controlador y definir su configuración en los recursos personalizados.

Puede encontrar explicaciones más especializadas sobre los operadores en el artículo How to explain Kubernetes Operators in plain English. Además, hay un libro electrónico gratuito de O'Reilly, disponible para descarga.

En este artículo, veremos más de cerca qué son los operadores y qué los hace funcionar. También escribiremos nuestro propio operador.

Configuración y requisitos previos

Para continuar, necesitarás instalar las siguientes herramientas:

kind

```
$ kind --version
kind version 0.9.0
```

golang

```
$ go version
go version go1.13.3 linux/amd64
```

kubebuilder

```
$ kubebuilder version
Version: version.Version{KubeBuilderVersion:"2.3.1"…
```

kubectl

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.11"...
```

operator-sdk

```
$ operator-sdk version
operator-sdk version: "v1.2.0"…
```

Recursos personalizados

Los recursos de la API son un concepto importante en Kubernetes. Estos recursos te permiten interactuar con Kubernetes mediante endpoints HTTP que pueden agruparse y versionarse. La API estándar puede ampliarse con los recursos personalizados, los cuales requieren que proporciones una Definición de recursos personalizados (CRD). Para más información, echa un vistazo a la página Extend the Kubernetes API with Custom Resource Definitions.

Este es un ejemplo de una CRD:

```
$ cat crd.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: irises.example.com
spec:
  group: example.com
  version: v1alpha1
  scope: Namespaced
  names:
    plural: irises
    singular: iris
    kind: Iris
    shortNames:
    - ir
  validation:
    openAPIV3Schema:
      required: ["spec"]
      properties:
        spec:
```

```
          required: ["replicas"]
          properties:
            replicas:
              type: "integer"
              minimum: 0
```

En el ejemplo anterior, definimos el recurso API GVK (Group/Version/Kind) como example.com/v1alpha1/Iris, con replicas como el único campo requerido.
Ahora definiremos un recurso personalizado basado en nuestra CRD:

```
$ cat crd-object.yaml
apiVersion: example.com/v1alpha1
kind: Iris
metadata:
  name: iris
spec:
  test: 42
  replicas: 1
```

En nuestro recurso personalizado, podemos definir cualquier campo además del número de replicas, lo cual es necesario para la CRD.
Después de implementar los dos archivos anteriores, nuestro recurso personalizado debería ser visible para el kubectl estándar.
A continuación, iniciaremos Kubernetes localmente utilizando kind, y después ejecutaremos los siguientes comandos de kubectl :

```
$ kind create cluster
$ kubectl apply -f crd.yaml
$ kubectl get crd irises.example.com
NAME                  CREATED AT
irises.example.com    2020-11-14T11:48:56Z

$ kubectl apply -f crd-object.yaml
$ kubectl get iris
NAME   AGE
iris   84s
```

Aunque hemos establecido un cierto número de réplicas para nuestro IRIS, realmente no sucedió nada hasta el momento. Como se esperaba. Necesitamos implementar un controlador - la entidad que pueda leer nuestro recurso personalizado y realice algunas acciones basadas en la configuración.
Por ahora, eliminaremos lo que creamos:

```
$ kubectl delete -f crd-object.yaml
$ kubectl delete -f crd.yaml
```

Controlador

Un controlador puede escribirse en cualquier lenguaje. Utilizaremos Golang por ser el lenguaje “ nativo ” de Kubernetes. Podríamos escribir la lógica de un controlador desde cero, pero

nuestros buenos amigos de Google y RedHat se nos adelantaron. Ellos han creado dos proyectos que pueden generar el código del operador y que solo requerirán cambios mínimos - kubebuilder y operator-sdk. Ambos pueden compararse en la página kubebuilder vs operator-sdk, y también en el artículo: What is the difference between kubebuilder and operator-sdk #1758.

Kubebuilder

Es útil que comencemos a familiarizarnos con Kubebuilder desde la página Kubebuilder book. El video Tutorial: Zero to Operator in 90 minutes también podría ayudar.

En los repositorios de sample-controller-kubebuilder y en kubebuilder-sample-controller se pueden encontrar ejemplos sobre las implementaciones del proyecto Kubebuilder.

Configuremos un nuevo proyecto para el operador:

```
$ mkdir iris
$ cd iris
$ go mod init iris # Creates a new module, name it iris
$ kubebuilder init --domain myardyas.club # An arbitrary domain, used below as a suff
ix in the API group
```

La configuración incluye muchos archivos y manifiestos. Por ejemplo, el archivo main.go es el punto de entrada del código. Importa la biblioteca controller-runtime, crea una instancia y ejecuta un administrador especial que registra la ejecución del controlador. No es necesario cambiar nada en ninguno de estos archivos.

Vamos a crear la CRD:

```
$ kubebuilder create api --group test --version v1alpha1 --kind Iris
Create Resource [y/n]
y
Create Controller [y/n]
y
…
```

De nuevo, se generan muchos archivos. Estos se describen con detallade en la página Adding a new API. Por ejemplo, puedes ver que se añade un archivo para el kind Iris en api/v1alpha1/iristypes.go. En nuestro primer ejemplo de la CRD, definimos el campo requerido replicas. Vamos a crear un campo idéntico aquí, esta vez en la estructura IrisSpec. También agregaremos el campo DeploymentName. El número de réplicas también debería ser visible en la sección Status, por lo que necesitamos hacer los siguientes cambios:

```
$ vim api/v1alpha1/iris_types.go
…
type IrisSpec struct {
        // +kubebuilder:validation:MaxLength=64
        DeploymentName string `json:"deploymentName"`
        // +kubebuilder:validation:Minimum=0
        Replicas *int32 `json:"replicas"`
```

```
}
…
type IrisStatus struct {
        ReadyReplicas int32 `json:"readyReplicas"`
}
…
```

Después de editar la API, editaremos el código repetitivo del controlador. Toda la lógica debe definirse en el método Reconcile (la mayor parte de este ejemplo proviene de mykindcontroller.go). También agregaremos un par de métodos auxiliares y reescribimos el método SetupWithManager.

```
$ vim controllers/iris_controller.go
…
import (
...
// Leave the existing imports and add these packages
        apps "k8s.io/api/apps/v1"
        core "k8s.io/api/core/v1"
        apierrors "k8s.io/apimachinery/pkg/api/errors"
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
        "k8s.io/client-go/tools/record"
)
// Add the Recorder field to enable Kubernetes events
type IrisReconciler struct {
        client.Client
        Log    logr.Logger
        Scheme *runtime.Scheme
        Recorder record.EventRecorder
}
…
// +kubebuilder:rbac:groups=test.myardyas.club,resources=iris,verbs=get;list;watch;cr
eate;update;patch;delete
// +kubebuilder:rbac:groups=test.myardyas.club,resources=iris/status,verbs=get;update
;patch
// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;up
date;delete
// +kubebuilder:rbac:groups="",resources=events,verbs=create;patch

func (r *IrisReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("iris", req.NamespacedName)
    // Fetch Iris objects by name
    log.Info("fetching Iris resource")
    iris := testv1alpha1.Iris{}
    if err := r.Get(ctx, req.NamespacedName, &iris); err != nil {
        log.Error(err, "unable to fetch Iris resource")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }
    if err := r.cleanupOwnedResources(ctx, log, &iris); err != nil {
        log.Error(err, "failed to clean up old Deployment resources for Iris")
        return ctrl.Result{}, err
    }

    log = log.WithValues("deployment_name", iris.Spec.DeploymentName)
```

```
    log.Info("checking if an existing Deployment exists for this resource")
    deployment := apps.Deployment{}
    err := r.Get(ctx, client.ObjectKey{Namespace: iris.Namespace, Name: iris.Spec.Dep
loymentName}, &deployment)
    if apierrors.IsNotFound(err) {
        log.Info("could not find existing Deployment for Iris, creating one...")

        deployment = *buildDeployment(iris)
        if err := r.Client.Create(ctx, &deployment); err != nil {
            log.Error(err, "failed to create Deployment resource")
            return ctrl.Result{}, err
        }

        r.Recorder.Eventf(&iris, core.EventTypeNormal, "Created", "Created deployment
 %q", deployment.Name)
        log.Info("created Deployment resource for Iris")
        return ctrl.Result{}, nil
    }
    if err != nil {
        log.Error(err, "failed to get Deployment for Iris resource")
        return ctrl.Result{}, err
    }

    log.Info("existing Deployment resource already exists for Iris, checking replica
count")

    expectedReplicas := int32(1)
    if iris.Spec.Replicas != nil {
        expectedReplicas = *iris.Spec.Replicas
    }

    if *deployment.Spec.Replicas != expectedReplicas {
        log.Info("updating replica count", "old_count", *deployment.Spec.Replicas, "n
ew_count", expectedReplicas)
        deployment.Spec.Replicas = &expectedReplicas
        if err := r.Client.Update(ctx, &deployment); err != nil {
            log.Error(err, "failed to Deployment update replica count")
            return ctrl.Result{}, err
        }

        r.Recorder.Eventf(&iris, core.EventTypeNormal, "Scaled", "Scaled deployment %
q to %d replicas", deployment.Name, expectedReplicas)

        return ctrl.Result{}, nil
    }

    log.Info("replica count up to date", "replica_count", *deployment.Spec.Replicas)
    log.Info("updating Iris resource status")

    iris.Status.ReadyReplicas = deployment.Status.ReadyReplicas
    if r.Client.Status().Update(ctx, &iris); err != nil {
        log.Error(err, "failed to update Iris status")
        return ctrl.Result{}, err
    }

    log.Info("resource status synced")
```

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

// Delete the deployment resources that no longer match the iris.spec.deploymentName
field
func (r *IrisReconciler) cleanupOwnedResources(ctx context.Context, log logr.Logger,
iris *testv1alpha1.Iris) error {
    log.Info("looking for existing Deployments for Iris resource")

    var deployments apps.DeploymentList
    if err := r.List(ctx, &deployments, client.InNamespace(iris.Namespace), client.Ma
tchingField(deploymentOwnerKey, iris.Name)); err != nil {
        return err
    }

    deleted := 0
    for _, depl := range deployments.Items {
        if depl.Name == iris.Spec.DeploymentName {
            // Leave Deployment if its name matches the one in the Iris resource
            continue
        }

        if err := r.Client.Delete(ctx, &depl); err != nil {
            log.Error(err, "failed to delete Deployment resource")
            return err
        }

        r.Recorder.Eventf(iris, core.EventTypeNormal, "Deleted", "Deleted deployment
%q", depl.Name)
        deleted++
    }

    log.Info("finished cleaning up old Deployment resources", "number_deleted", delet
ed)
    return nil
}

func buildDeployment(iris testv1alpha1.Iris) *apps.Deployment {
    deployment := apps.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:            iris.Spec.DeploymentName,
            Namespace:       iris.Namespace,
            OwnerReferences: []metav1.OwnerReference{*metav1.NewControllerRef(&iris,
testv1alpha1.GroupVersion.WithKind("Iris"))},
        },
        Spec: apps.DeploymentSpec{
            Replicas: iris.Spec.Replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: map[string]string{
                    "iris/deployment-name": iris.Spec.DeploymentName,
                },
            },
            Template: core.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: map[string]string{
                        "iris/deployment-name": iris.Spec.DeploymentName,
```

```
                    },
                },
                Spec: core.PodSpec{
                    Containers: []core.Container{
                        {
                            Name:  "iris",
                            Image: "
store/intersystems/iris-community:2020.4.0.524.0",
                        },
                    },
                },
            },
        },
    }
    return &deployment
}

var (
    deploymentOwnerKey = ".metadata.controller"
)

// Specifies how the controller is built to watch a CR and other resources
// that are owned and managed by that controller
func (r *IrisReconciler) SetupWithManager(mgr ctrl.Manager) error {
    if err := mgr.GetFieldIndexer().IndexField(&apps.Deployment{}, deploymentOwnerKey
, func(rawObj runtime.Object) []string {
        // grab the Deployment object, extract the owner...
        depl := rawObj.(*apps.Deployment)
        owner := metav1.GetControllerOf(depl)
        if owner == nil {
            return nil
        }
        // ...make sure it's an Iris...
        if owner.APIVersion != testv1alpha1.GroupVersion.String() || owner.Kind != "I
ris" {
            return nil
        }

        // ...and if so, return it
        return []string{owner.Name}
    }); err != nil {
        return err
    }

    return ctrl.NewControllerManagedBy(mgr).
        For(&testv1alpha1.Iris{}).
        Owns(&apps.Deployment{}).
        Complete(r)
}
```

Para que el registro de eventos funcione, necesitamos agregar otra línea al archivo main.go:

```
if err = (&controllers.IrisReconciler{
                Client: mgr.GetClient(),
                Log:    ctrl.Log.WithName("controllers").WithName("Iris"),
```

```
                Scheme: mgr.GetScheme(),
                Recorder: mgr.GetEventRecorderFor("iris-controller"),
        }).SetupWithManager(mgr); err != nil {
```

Ahora todo está listo para establecer un operador.
Primero vamos a instalar la CRD utilizando el destino de instalación Makefile:

```
$ cat Makefile
…
# Install CRDs into a cluster
install: manifests
        kustomize build config/crd | kubectl apply -f -
...
$ make install
```

Puedes echar un vistazo al archivo CRD YAML resultante, en el directorio config/crd/bases/.
Ahora, comprueba la existencia de la CRD en el clúster:

```
$ kubectl get crd
NAME                        CREATED AT
iris.test.myardyas.club   2020-11-17T11:02:02Z
```

Vamos a ejecutar nuestro controlador en otro terminal, a nivel local (no en Kubernetes), solo para ver si realmente funciona:

```
$ make run
...
2020-11-17T13:02:35.649+0200 INFO controller-
runtime.metrics metrics server is starting to listen {"addr": ":8080"}
2020-11-17T13:02:35.650+0200 INFO setup starting manager
2020-11-17T13:02:35.651+0200 INFO controller-
runtime.manager starting metrics server {"path": "/metrics"}
2020-11-17T13:02:35.752+0200 INFO controller-
runtime.controller Starting EventSource
{"controller": "iris", "source": "kind source: /, Kind="}
2020-11-17T13:02:35.852+0200 INFO controller-
runtime.controller Starting EventSource
{"controller": "iris", "source": "kind source: /, Kind="}
2020-11-17T13:02:35.853+0200 INFO controller-
runtime.controller Starting Controller {"controller": "iris"}
2020-11-17T13:02:35.853+0200 INFO controller-
runtime.controller Starting workers {"controller": "iris", "worker count": 1}
…
```

Ahora que ya tenemos la CRD y el controlador instalados, lo único que debemos hacer es crear una instancia de nuestro recurso personalizado. Es posible encontrar una plantilla en el archivo config/samples/example.comv1alpha1iris.yaml. En este archivo, necesitamos hacer modificaciones similares a las que se realizaron en crd-objeto.yaml:

```
$ cat config/samples/test_v1alpha1_iris.yaml
apiVersion: test.myardyas.club/v1alpha1
```

```
kind: Iris
metadata:
  name: iris
spec:
  deploymentName: iris
  replicas: 1
$ kubectl apply -f config/samples/test_v1alpha1_iris.yaml
```

Tras un ligero retraso debido a la necesidad de extraer una imagen de IRIS, deberías ver el contenedor de IRIS en ejecución:

```
$ kubectl get deploy
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
iris   1/1     1            1           119s
$ kubectl get pod
NAME                    READY   STATUS    RESTARTS   AGE
iris-6b78cbb67-vk2gq    1/1     Running   0          2m42s
$ kubectl logs -f -l iris/deployment-name=iris
```

Puedes abrir el portal de IRIS usando el comando port-forward de kubectl:

```
$ kubectl port-forward deploy/iris 52773
```

Ve a http://localhost:52773/csp/sys/UtilHome.csp en tu navegador.
¿Qué sucedería si cambiamos el número de replicas en la CRD? Vamos a hacerlo y aplicar este cambio:

```
$ vi config/samples/test_v1alpha1_iris.yaml
…
  replicas: 2
$ kubectl apply -f config/samples/test_v1alpha1_iris.yaml
```

Ahora ser debería ver que aparece otro contenedor de Iris.

```
$ kubectl get events
…
54s         Normal   Scaled                  iris/iris                  Scaled dep
loyment "iris" to 2 replicas
54s         Normal   ScalingReplicaSet       deployment/iris            Scaled up
replica set iris-6b78cbb67 to 2
```

Esto son los mensajes, en el terminal donde el controlador está en ejecución, que informan sobre la reconciliación exitosa:

```
2020-11-17T13:09:04.102+0200 INFO controllers.Iris
replica count up to date
{"iris": "default/iris", "deployment_name": "iris", "replica_count": 2}
2020-11-17T13:09:04.102+0200 INFO controllers.Iris
updating Iris resource status {"iris": "default/iris", "deployment_name": "iris"}
2020-11-17T13:09:04.104+0200 INFO controllers.Iris
```

```
resource status synced {"iris": "default/iris", "deployment_name": "iris"}
2020-11-17T13:09:04.104+0200 DEBUG controller-
runtime.controller Successfully Reconciled
{"controller": "iris", "request": "default/iris"}
```

Ok, parece que nuestros controladores están funcionando. Ahora, estamos listos para implementar ese controlador dentro de Kubernetes como un contenedor. Para hacerlo, necesitamos crear el controlador del contenedor Docker y llevarlo al registro. Este puede ser cualquier registro que funcione con Kubernetes: DockerHub, ECR, GCR, etc.
Utilizaremos los Kubernetes locales (kind), así que vamos a implementar el controlador en el registro local mediante el script kind-with-registry.sh, disponible en la página Local Registry. Para ello, podemos simplemente eliminar el clúster actual y recrearlo:

```
$ kind delete cluster
$ ./kind_with_registry.sh
$ make install
$ docker build . -t localhost:5000/iris-
operator:v0.1 # Dockerfile is autogenerated by kubebuilder
$ docker push localhost:5000/iris-operator:v0.1
$ make deploy IMG=localhost:5000/iris-operator:v0.1
```

El controlador se implementará en el namespace IRIS-system. O puedes escanear todos los contenedores para encontrar un namespace como kubectl get pod -A):

```
$ kubectl -n iris-system get po
NAME                                    READY   STATUS    RESTARTS   AGE
iris-controller-manager-bf9fd5855-kbklt   2/2     Running   0          54s
```

Revisemos los registros:

```
$ kubectl -n iris-system logs -f -l control-plane=controller-manager -c manager
```

Puedes experimentar modificando el número de replicas en la CRD y observar cómo estos cambios se reflejan en el número de instancias en IRIS.

Operator-SDK

Otra herramienta útil para generar el código del operador es Operator SDK. Para tener una idea inicial sobre cómo funciona esta herramienta, echa un vistazo a este tutorial. Primero deberías instalar operator-sdk.
Para nuestro sencillo caso de uso, el proceso será similar al que realizamos con kubebuilder (puedes eliminar/crear el clúster kind con el registro Docker antes de continuar). Ejecuta el siguiente script en otro directorio:

```
$ mkdir iris
$ cd iris
$ go mod init iris
$ operator-sdk init --domain=myardyas.club
```

```
$ operator-sdk create api --group=test --version=v1alpha1 --kind=Iris
# Answer two 'yes'
```

Ahora modifica las estructuras IrisSpec e IrisStatus en el mismo archivo, api/v1alpha1/iristypes.go.
Utilizaremos el mismo archivo iriscontroller.go, como hicimos en kubebuilder. No olvides agregar el campo Recorder en el archivo main.go.
Dado que kubebuilder y operator-sdk utilizan versiones diferentes de los paquetes Golang, debes agregar contexto en la función SetupWithManager que se encuentra en controllers/iriscontroller.go:

```
ctx := context.Background()
if err := mgr.GetFieldIndexer().IndexField
(ctx, &apps.Deployment{}, deploymentOwnerKey, func(rawObj runtime.Object) []string {
```

Luego, instala la CRD y el operador (asegúrate de que el kind del clúster se está ejecutando):

```
$ make install
$ docker build . -t localhost:5000/iris-operator:v0.2
$ docker push localhost:5000/iris-operator:v0.2
$ make deploy IMG=localhost:5000/iris-operator:v0.2
```

Ahora debería ver la CRD, el contenedor del operador y los contenedores de IRIS de forma similar a los que vimos cuando trabajamos con kubebuilder.

Conclusión

Aunque un controlador incluye una gran cantidad de código, se puede ver que modificar las replicas de IRIS simplemente es cuestión de modificar una línea en un recurso personalizado. Toda la complejidad está oculta en la implementación del controlador. Hemos visto cómo se puede crear un operador simple mediante el uso de herramientas útiles para configurarlo. Nuestro operador solo necesitaba las replicas de IRIS. Ahora imagina que en realidad necesitamos que los datos de IRIS persistan en el disco - esto requeriría que utilizáramos StatefulSet y Persistent Volumes. Además, necesitaríamos un Service y, tal vez, un Ingress para acceder externamente. Deberíamos ser capaces de establecer la versión de IRIS y la contraseña del sistema, Mirroring y/o ECP, entre otras cosas. Se puede imaginar la cantidad de trabajo que InterSystems tuvo que hacer para simplificar la implementación de IRIS, ocultando toda la lógica específica de IRIS dentro del código del operador.
En el próximo artículo, analizaremos el Operador de IRIS (IKO) más en detalle e investigaremos todo lo que puede hacer en escenarios más complejos.

#DevOps #Kubernetes #InterSystems IRIS

URL de fuente:https://es.community.intersystems.com/post/an%C3%A1lisis-en-detalle-del-operador-intersystems-kubernetes-introducci%C3%B3n-los-operadores-kubernetes