Artigo
Mikhail Khomenko · Mar. 2, 2021 17min de leitura

Mergulho profundo no InterSystems Kubernetes Operator: introdução aos Operadores Kubernetes

Introdução
Vários recursos nos dizem como executar o IRIS em um cluster Kubernetes, como Implantar uma solução InterSystems IRIS no EKS usando GitHub Actions e Implantar a solução InterSystems IRIS no GKE usando GitHub Actions. Esses métodos funcionam, mas exigem que você crie manifestos do Kubernetes e gráficos do Helm, o que pode ser bastante demorado.
Para simplificar a implantação do IRIS, a InterSystems desenvolveu uma ferramenta incrível chamada InterSystems Kubernetes Operator (IKO). Vários recursos oficiais explicam o uso de IKO em detalhes, como Novo vídeo: Intersystems IRIS Kubernetes Operator e InterSystems Kubernetes Operator.

A documentação do Kubernetes diz que os operadores substituem um operador humano que sabe como lidar com sistemas complexos no Kubernetes. Eles fornecem configurações do sistema na forma de *recursos personalizados*. Um operador inclui um *controlador personalizado* que lê essas configurações e executa as etapas definidas pelas configurações para configurar e manter corretamente sua aplicação. O controlador personalizado é um pod simples implantado no Kubernetes. Portanto, de modo geral, tudo que você precisa fazer para fazer um operador trabalhar é implantar um pod de controlador e definir suas configurações em recursos personalizados.
Você pode encontrar uma explicação de alto nível sobre os operadores em: Como explicar os operadores do Kubernetes em inglês simples. Além disso, um e-book gratuito da O ' Reilly está disponível para download.
Neste artigo, veremos mais de perto o que são os operadores e o que os faz funcionar. Também escreveremos nosso próprio operador.

Pré-requisitos e configuração

Para acompanhar, você precisará instalar as seguintes ferramentas:

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

Os recursos da API são um conceito importante no Kubernetes. Esses recursos permitem que você interaja com o Kubernetes por meio de endpoints HTTP que podem ser agrupados e versionados. A API padrão pode ser estendida com recursos personalizados, o que exige que você forneça uma Definição de Recurso Personalizado (CRD). Dê uma olhada na páginaEstender a API Kubernetes com CustomResourceDefinitions para informações detalhadas.
Aqui está um exemplo de um 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
```

No exemplo acima, definimos o recurso API GVK (Group/Version/Kind) como example.com/v1alpha1/Iris, com réplicas como o único campo obrigatório.
Agora vamos definir um recurso personalizado com base em nosso CRD:

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

Em nosso recurso personalizado, podemos definir quaisquer campos além de réplicas, o que é exigido pelo CRD. Depois de implantar os dois arquivos acima, nosso recurso personalizado deve se tornar visível para o kubectl padrão.
Vamos iniciar o Kubernetes localmente usando kind e, em seguida, executar os seguintes comandos 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
```

Embora tenhamos definido uma quantidade de réplica para o nosso IRIS, nada realmente acontece no momento. Isso é esperado. Precisamos implantar um controlador - a entidade que pode ler nosso recurso personalizado e realizar algumas ações baseadas em configurações.
Por enquanto, vamos limpar o que criamos:

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

Controlador

Um controlador pode ser escrito em qualquer idioma. Usaremos Golang como linguagem "nativa" do Kubernetes. Poderíamos escrever a lógica de um controlador do zero, mas o pessoal do Google e da RedHat nos deu uma vantagem. Eles criaram dois projetos que podem gerar o código do operador que exigirá apenas alterações mínimas –kubebuilder e operator-sdk. Esses dois são comparados na página kubebuilder vs operator-sdk, bem como aqui: Qual é a diferença entre kubebuilder e operator-sdk #1758.

Kubebuilder

É conveniente começar nosso contato com o Kubebuilder na página do livro do Kubebuilder. O vídeo Tutorial: Zero ao Operador em 90 minutos do mantenedor do Kubebuilder também pode ajudar.

Implementações de exemplo do projeto Kubebuilder podem ser encontradas nos repositórios sample-controller-kubebuilder e kubebuilder-sample-controller.

Vamos construir um novo projeto de operador:

```
$ mkdir iris
$ cd iris
$ go mod init iris # Cria um novo módulo, chame-o de iris
$ kubebuilder init --domain myardyas.club # Um domínio arbitrário, usado abaixo como um sufixo no grupo da API
```

Fazer scaffolding inclui muitos arquivos e manifestos. O arquivo main.go, por exemplo, é o ponto de entrada do código. Ele importa a biblioteca controller-runtime, e instancia e executa um gerenciador especial que mantém o controle da execução do controlador. Nada há mudar em nenhum desses arquivos.

Vamos criar o CRD:

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

Novamente, muitos arquivos são gerados. Eles são descritos em detalhes na página Adicionando uma nova API. Por exemplo, você pode ver que um arquivo do tipo Iris foi adicionado em api/v1alpha1/iristypes.go. Em nosso primeiro CRD de exemplo, definimos o campo de réplicasnecessário. Vamos criar um campo idêntico aqui, desta vez na estrutura IrisSpec. Também adicionaremos o campo DeploymentName. A contagem de réplicas também deve estar visível na seção Status , portanto, precisamos fazer as seguintes alterações:

```
$ 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"`
}
…
```

Depois de editar a API, passaremos para a edição do boilerplate do controlador. Toda a lógica deve ser definida no método Reconcile (este exemplo é retirado principalmente do mykindcontroller.go). Também adicionamos alguns métodos auxiliares e reescrevemos o método SetupWithManager.

```
$ vim controllers/iris_controller.go
…
import (
...
// Deixe as importações existentes e adicione esses pacotes
        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"
)
// Adicione o campo Recorder para habilitar eventos Kubernetes
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)
    // Buscar por objetos Iris por nome
    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
}

// Exclui os recursos de implantação que não correspondem mais ao campo iris.spec.dep
loymentName
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 {
            // Sai da implantação se o nome corresponder ao do recurso Iris
            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"
)

// Especifica como o controlador é construído para assistir um CR e outros recursos
// que pertencem e são gerenciados por esse controlador
func (r *IrisReconciler) SetupWithManager(mgr ctrl.Manager) error {
    if err := mgr.GetFieldIndexer().IndexField(&apps.Deployment{}, deploymentOwnerKey
, func(rawObj runtime.Object) []string {
        // pega o objeto Deployment, extraia o proprietário...
        depl := rawObj.(*apps.Deployment)
        owner := metav1.GetControllerOf(depl)
        if owner == nil {
            return nil
        }
        // ...certifica-se de que é um Iris...
```

```
        if owner.APIVersion != testv1alpha1.GroupVersion.String() || owner.Kind != "I
ris" {
            return nil
        }

        // ...e se for assim, o retorna
        return []string{owner.Name}
    }); err != nil {
        return err
    }

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

Para fazer o registro de eventos funcionar, precisamos adicionar mais uma linha ao arquivo 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 {
```

Agora tudo está pronto para configurar um operador.
Vamos instalar o CRD primeiro usando o Makefile de destino install:

```
$ cat Makefile
…
# Instala CRDs em um cluster
install: manifests
        kustomize build config/crd | kubectl apply -f -
...
$ make install
```

Você pode dar uma olhada no arquivo CRD YAML resultante no diretório config/crd/bases/.
Agora verifique a existência do CRD no cluster:

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

Vamos executar nosso controlador em outro terminal, localmente (não no Kubernetes) – apenas para ver se ele 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}
…
```

Agora que temos o CRD e o controlador instalados, tudo o que precisamos fazer é criar uma instância do nosso recurso personalizado. Um modelo pode ser encontrado no arquivo config/samples/example.comv1alpha1iris.yaml. Neste arquivo, precisamos fazer alterações semelhantes àquelas em crd-object.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
```

Após um breve atraso causado pela necessidade de extrair uma imagem IRIS, você deverá ver o pod IRIS em execução:

```
$ 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
```

Você pode abrir o portal IRIS usando o comando kubectl port-forward:

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

Vá em http://localhost:52773/csp/sys/UtilHome.csp em seu navegador.
E se mudarmos a contagem das réplicas no CRD? Vamos fazer e aplicar esta mudança:

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

Agora você deve ver outro pod Iris aparecer.

```
$ 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
```

Veja as mensagens de log no terminal onde o controlador executa o relatório de reconciliação bem-sucedida:

```
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, nossos controladores parecem estar funcionando. Agora estamos prontos para implantar esse controlador dentro do Kubernetes como um pod. Para isso, precisamos criar o contêiner docker do controlador e enviá-lo para o registro. Pode ser qualquer registro que funcione com Kubernetes – DockerHub, ECR, GCR e assim por diante. Usaremos o Kubernetes local (kind), então vamos implantar o controlador no registro local usando o script kind-with-registry.sh disponível na página de Registro Local. Podemos simplesmente remover o cluster atual e recriá-lo:

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

O controlador será implantado no namespace do sistema IRIS. Como alternativa, você pode verificar todos os pods para encontrar um 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
```

Vamos verificar os logs:

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

Você pode experimentar alterar a contagem de réplicas no CRD e observar como essas mudanças são refletidas na contagem de instâncias IRIS.

Operator-SDK

Outra ferramenta útil para gerar o código do operador é o Operator SDK. Para ter uma ideia inicial dessa ferramenta, dê uma olhada neste tutorial. Você deve instalar o operador-sdk primeiro.
Para nosso caso de uso simples, o processo será semelhante ao que trabalhamos com o kubebuilder (você pode excluir/criar o kind cluster com o registro Docker antes de continuar). Execute em outro diretório:

```
$ mkdir iris
$ cd iris
$ go mod init iris
$ operator-sdk init --domain=myardyas.club
$ operator-sdk create api --group=test --version=v1alpha1 --kind=Iris
# Responda dois 'yes'
```

Agora altere as estruturas IrisSpec e IrisStatus no mesmo arquivo – api/v1alpha1/iristypes.go.
Usaremos o mesmo arquivo iriscontroller.go que usamos no kubebuilder. Não se esqueça de adicionar o campo Recorder no arquivo main.go.
Como o kubebuilder e o operator-sdk usam versões diferentes dos pacotes Golang, você deve adicionar um contexto na função SetupWithManager em controllers/iriscontroller.go:

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

Em seguida, instale o CRD e o operador (certifique-se de que o kind cluster está em execução):

```
$ 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
```

Agora você deve ver o CRD, o pod do operador e o(s) pod(s) IRIS semelhantes aos que vimos quando trabalhamos com o kubebuilder.

Conclusão

Embora um controlador inclua muito código, você viu que alterar as réplicas IRIS é apenas uma questão de alterar uma linha em um recurso personalizado. Toda a complexidade está oculta na implementação do controlador.
Vimos como um operador simples pode ser criado usando ferramentas de scaffolding úteis.
Nosso operador se preocupava apenas com as réplicas IRIS. Agora imagine que realmente precisamos ter os dados IRIS persistentes no disco. Isso exigiria StatefulSet e Volumes Persistentes. Além disso, precisaríamos de um Service e, talvez, Ingress para acesso externo. Devemos ser capazes de definir a versão do IRIS e a senha do sistema, espelhamento e/ou ECP e assim por diante. Você pode imaginar a quantidade de trabalho que a InterSystems teve que realizar para simplificar a implantação do IRIS, ocultando toda a lógica específica do IRIS dentro do código do operador.
No próximo artigo, veremos o IRIS Operator (IKO) em mais detalhes, e investigaremos suas possibilidades em cenários mais complexos.

#DevOps #Kubernetes #InterSystems IRIS