91Kubernetes 系列（八十四）如何创建 Kubernetes Operator?

在这个关于 Operator 模式的系列的第一部分，我们了解了它是什么以及它在哪些情况下非常有用，尤其是对于自动化。

那么今天我们将了解如何创建 Kubernetes Operator！

工具

与每个开源解决方案一样，如果您需要做某事，就会有一堆具有各自特性的工具。如果您想查看该列表，请查看 Kubernetes 文档（https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#writing-operator）。

在本系列中，我们将使用 Operator Framework(https://operatorframework.io/) 和 KubeBuilder(https://book.kubebuilder.io/)。

Operator Framework

关于 Operator Framework 的几句话，我们将使用 Go SDK，但你需要知道你也可以将它与 Ansible 和 Helm 一起使用。

设置

Homebrew

如果您使用的是 Homebrew，则可以使用以下命令安装 Operator Framework SDK：

```
brew install operator-sdk
```

从 Github Release 获取

```
# Define informations about your platform
export ARCH=$(case $(uname -m) in x86_64) echo -n amd64 ;; aarch64) echo -n arm64 ;; *) echo -n $(uname -m) ;; esac)
export OS=$(uname | awk '{print tolower($0)}')

# Download the binary for your platform
export OPERATOR_SDK_DL_URL=https://github.com/operator-framework/operator-sdk/releases/download/v1.28.0
curl -LO ${OPERATOR_SDK_DL_URL}/operator-sdk_${OS}_${ARCH}
# Install the binary
chmod +x operator-sdk_${OS}_${ARCH} && sudo mv operator-sdk_${OS}_${ARCH} /usr/local/bin/operator-sdk
```

创建第一个 Operator

初始化项目

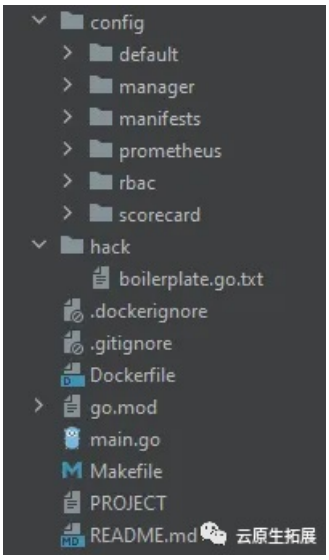
首先要做的是使用以下命令初始化项目：

```
operator-sdk init — domain [YOUR DOMAIN] — repo [YOUR CODE REPOSITORY]
```

例如：

```
operator-sdk init --domain adaendra.org --repo github.com/adaendra/test-operator
```

它将生成这样的文件夹结构



您将能够找到一些通用文件、许多通用文件（如 Makefile 或 Dockerfile）以及以 `main.go` 开头的 Golang 项目。

> 注意：默认情况下，您的命名空间能够监视集群中任何位置的资源。

> 所以如果你想限制它的视野，你可以更新manager的定义来添加 Namespace 选项。 `mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{Namespace: "dummy_ns"})`

> 有关 Operator 范围的更多信息，请查看 SDK 文档(https://sdk.operatorframework.io/docs/building-operators/golang/operator-scope/)

创建 API、控制器和 CRD

在很多情况下，当我们使用 Operator 时，我们希望创建一个自定义资源定义，它将用作我们任务的参考。

在本教程中，我们将在网关组中创建一个自定义资源 MyProxy，该资源将为每个实例部署一个 Nginx 部署。

生成代码的命令

```
operator-sdk create api --group gateway --version v1alpha1 --kind MyProxy --resource --controller
```

执行后，您可以看到两个新文件夹： `api` 和 `controllers` 。

API

在这个文件夹中，我们唯一感兴趣的文件是 `myproxy_types.go` 。在此文件中，我们将定义规范中所需的所有字段，但我们也将在这里定义 Status 结构！

对于我们的示例，我们将在 MyProxy Spec 中定义一个 `Name` 字段。

```
type MyProxySpec struct {
    Name string `json:"name,omitempty"`
}
```

> 重要的！！该文件用作为您的 Operator 构建大量 yaml 文件的基础。因此，此文件中的每次修改都执行这两个命令： `make manifests & make generate`

Controller

在此文件夹中，您将找到与您的自定义资源相关的每个控制器，例如我们之前生成的 `myproxy_controller.go` 。此文件夹是您的Operator可以执行的操作的中心位置。

在每个控制器文件中，您会发现我们必须更新的两个方法： `Reconcile` 和 `SetupWithManager` 。

SetupWithManager

```
// SetupWithManager sets up the controller with the Manager.
func (r *MyProxyReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&gatewayv1alpha1.MyProxy{}).
        Owns(&appsv1.Deployment{}).
        Complete(r)
}
```

在这个例子中（这也是我们的实现），我们可以看到：

`ctrl.NewControllerManagedBy(mgr)` 创建一个带有基本选项的新控制器。（通过这种方法，您可以个性化控制器选项，例如您想要并行的最大协调数量）

`For(&gatewayv1alpha1.MyProxy{})` 将声明我们希望在特定类型的资源上发生添加/更新/删除事件时触发协调。（这里是MyProxy）您可以将它用于您想要观看的每种资源。（例如，如果您想通过 Nginx 动态公开所有部署，则很有用）

`Owns(&appsv1.Deployment{})` 与 For 非常相似，因此它会声明我们希望在发生添加/更新/删除事件时触发协调。但它也会添加一个过滤器，因为只有当 Operator 拥有事件资源时才会触发对账。（因此，如果您更新另一个部署，您的 Operator 中不会发生任何事情）

Reconcile

此方法是 Operator 的核心，并且是每次触发对帐时都会执行的方法。

但在深入了解该功能之前，有一件重要的事情需要先了解。在方法上方，您可以看到一些以 //+kubebuilder 开头的注释。此评论定义了您的 Operator 的权利！

所以正确定义它们非常重要！在我们的例子中，我们需要为 Operaator 添加一些权限，以便能够读取/创建和更新部署。

每个注释的定义如下：

```
// +kubebuiler:rbac:groups=[资源组],resources=[资源名称],verbs=[动词]
```

资源的组只能是一个值，但是对于资源名称和动词，你可以一次定义多个值，所有的值都用 ; 连接起来。

```
// +kubebuilder:rbac:groups=gateway.adaendra.org,resources=myproxies,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=gateway.adaendra.org,resources=myproxies/status,verbs=get;update;patch
// +kubebuilder:rbac:groups=gateway.adaendra.org,resources=myproxies/finalizers,verbs=update
// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete
```

现在我们可以深入研究函数代码。如前所述，每次触发对帐时都会调用此函数。因此，我们必须小心我们在这里所做的事情！

例如，如果我们要创建资源，我们必须确保它们不存在于集群中！如果它已经存在，我们必须检查它并在需要时进行一些更新！

1. 检索我们的自定义资源

所以第一步是尝试检索我们的自定义资源的实例（这里是 MyProxy 的实例）。我们需要它来获取它的规范，并能够更新它的状态。

```
// Retrieve the resource
myProxy := &gatewayv1alpha1.MyProxy{}
err := r.Get(ctx, req.NamespacedName, myProxy)

if err != nil {
    // If we have an error and this error said "not found", we ignore the error
    if errors.IsNotFound(err) {
        log.Info("Resource not found. Error ignored as the resource must have been deleted.")
        return ctrl.Result{}, nil
    }
    // If it's another error, we return it
    log.Error(err, "Error while retrieving MyProxy instance")
    return ctrl.Result{}, err
}
```

2. 获取operator管理的资源

现在我们有了我们的“父”资源，我们要检索我们的“子”资源。在我们的例子中，它是一个 Deployment ，它的名称由 MyProxy 中的字段 Name 定义，并且必须位于命名空间 test_ns 中。

```
found := &appsv1.Deployment{}
err = r.Get(ctx, types.NamespacedName{Name: myProxy.Spec.Name, Namespace: "test_ns"}, found)
```

3. 检查资源是否存在

接下来的步骤包括检查我们在上一步中得到了什么。如果变量 err 是未找到错误，我们知道该资源不存在，因此我们可以创建它！如果它包含另一个错误，我们将其返回。

我们的实现看起来像这样:

```
if err != nil && errors.IsNotFound(err) {
   // Define a new deployment
   dep := r.deploymentForExample(myProxy)
   log.Info("Creating a new Deployment", "Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Name)
   err = r.Create(ctx, dep)
   if err != nil {
      log.Error(err, "Failed to create new Deployment", "Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Nam
      return ctrl.Result{}, err
   }
   // Deployment created successfully - return and requeue
   return ctrl.Result{Requeue: true}, nil
} else if err != nil {
   log.Error(err, "Failed to get Deployment")
   return ctrl.Result{}, err
}
```

这是 deploymentForExample 的一个非常简单的例子:

```
func (r *MyProxyReconciler) deploymentForExample(myproxy *gatewayv1alpha1.MyProxy) *appsv1.Deployment {
   dep := &appsv1.Deployment{}
   dep.Namespace = "test_ns"
   dep.Name = myproxy.Spec.Name
   var replicas int32 = 2
   labels := map[string]string{
      "test_label": myproxy.Spec.Name,
   }
   dep.Spec = appsv1.DeploymentSpec{
      Replicas: &replicas,
      Template: corev1.PodTemplateSpec{
         Spec: corev1.PodSpec{
            Containers: []corev1.Container{
               {
                  Name:  "nginx",
                  Image: "nginx",
               },
            },
         },
      },
   }
   dep.Labels = labels
   dep.Spec.Template.Labels = labels
   return dep
}
```

4. 更新资源

如果我们在尝试检索资源时没有遇到错误，则意味着我们能够正确获取资源。所以我们可以检查它的参数并在某些值已更改时更新它。

```
var size int32 = 2
if *found.Spec.Replicas != size {
   found.Spec.Replicas = &size
   err = r.Update(ctx, found)
   if err != nil {
      log.Error(err, "Failed to update Deployment", "Deployment.Namespace", found.Namespace, "Deployment.Name", found.Nam
      return ctrl.Result{}, err
   }
   // Spec updated - return and requeue
   return ctrl.Result{Requeue: true}, nil
}
```

在我们的示例中，我们将检查 pod 的数量是否仍然等于 2 。如果不是这种情况，我们将尝试更新资源，并在出现错误时管理错误。

更新生成的文件

一旦我们完成了控制器的更新，执行以下命令非常重要：

```
make manifests
```

我们之前看到，我们可以找到一些定义控制器 RBAC 权限的注释。因此我们需要执行此命令（至少）生成 RBAC 定义文件。

Operator 构建

现在我们的 Operator 已经可以使用了，我们可以在部署之前构建它。

构建前

默认情况下，构建的镜像将命名为 `controller:latest` ，可以推送到 `example.com/tmpoperator` 。可以想象，它会产生一些问题。

所以，如果你想更新这些信息，你必须：

- 更新 Makefile 中的变量 IMG 和 IMAGE_TAG_BASE
- 更新 config/manager/manager.yaml 中的镜像名称

构建

要执行构建，请使用此命令

```
make docker-build
```

如果你想将推送到远程 docker registry，这一个

```
make docker-push
```

部署

要部署您的 Operator，您必须执行 2 个命令：

在集群上部署所有自定义资源定义

```
make install
```

部署您的 Operator

```
```

```
make deploy
```

测试

完成上述所有操作后，您可以尝试部署一个 `MyProxy` 实例，您应该会看到一个 nginx 部署出现！

MyProxy 实例定义示例

```
apiVersion: gateway.example.com/v1alpha1
kind: MyProxy
metadata:
  labels:
    app.kubernetes.io/name: myproxy
    app.kubernetes.io/instance: myproxy-sample
    app.kubernetes.io/part-of: tmpoperator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: tmpoperator
  name: myproxy-sample
spec:
  name: toto
```

这部分很长，但有必要向您展示如何创建一个 Operator 并看看我们可以做什么。

在本系列的下一部分中，我们将看到高级配置和功能，以帮助您的 Operator 提高效率！

我希望它能对你有所帮助，如果你有任何问题（没有愚蠢的问题）或某些点你不清楚，请不要犹豫，在评论中添加你的问题。