Operator 简单示例

本节主要介绍如何使用 Operator Framework 编写一个简单的 Operator 应用。

介绍

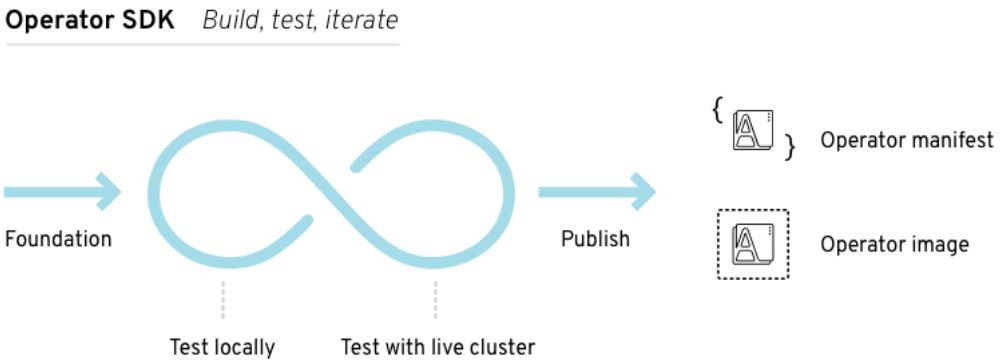
`Operator` 就可以看成是 CRD 和 Controller 的一种组合特例，Operator 是一种思想，它结合了特定领域知识并通过 CRD 机制扩展了 Kubernetes API 资源，使用户管理 Kubernetes 的内置资源（Pod、Deployment等）一样创建、配置和管理应用程序，Operator 是一个特定的应用程序的控制器，通过扩展 Kubernetes API 资源以代表 Kubernetes 用户创建、配置和管理复杂应用程序的实例，通常包含资源模型定义和控制器，通过 `Operator` 通常是为了实现某种特定软件（通常是有状态服务）的自动化运维。

我们完全可以通过上面的方式编写一个 CRD 对象，然后去手动实现一个对应的 Controller 就可以实现一个 Operator，但是我们也发现从头开始去构建一个 CRD 控制器并不容易，需要对 Kubernetes 的 API 有深入了解，并且 RBAC 集成、镜像构建、持续集成和部署等都需要很大工作量。为了解决这个问题，社区就推出了对应的简单易用的 Operator 框架，比较主流的是 kubebuilder 和 Operator Framework，这两个框架的使用基本上差别不大，我们可以根据自己习惯选择一个即可，我们这里先使用 `Operator Framework` 来给大家简要说明下 Operator 的开发。

Operator Framework

Operator Framework 是 CoreOS 开源的一个用于快速开发 Operator 的工具包，该框架包含两个主要的部分：

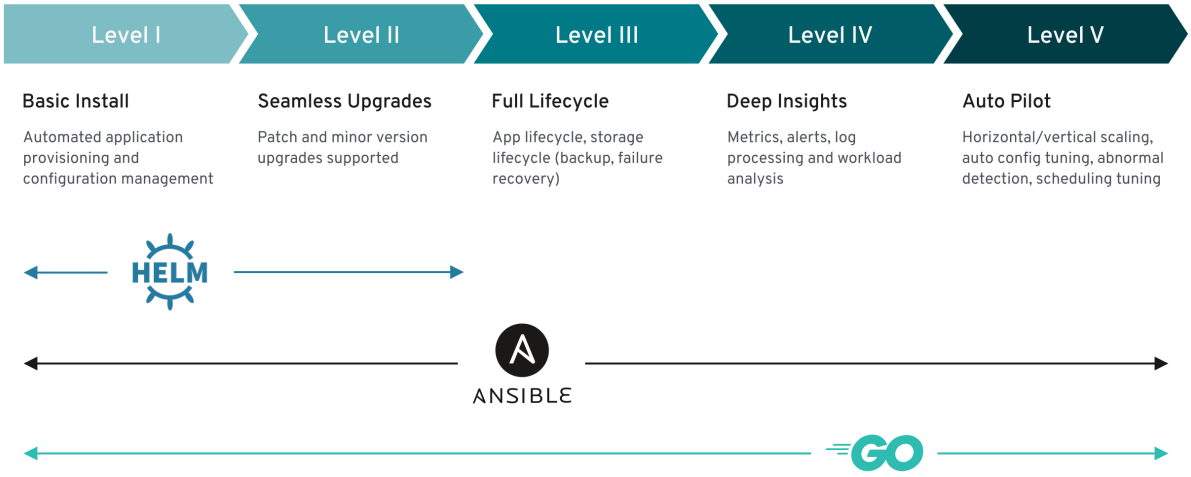
- Operator SDK: 无需了解复杂的 Kubernetes API 特性，即可让你根据你自己的专业知识构建一个 Operator 应用。
- Operator Lifecycle Manager（OLM）: 帮助你安装、更新和管理跨集群的运行中的所有 Operator（以及他们的相关服务）



Operator SDK 提供了用于开发 Go、Ansible 以及 Helm 中的 Operator 的工作流，下面的工作流适用于 Golang 的 Operator：

1. 使用 SDK 创建一个新的 Operator 项目
2. 通过添加自定义资源（CRD）定义新的资源 API
3. 指定使用 SDK API 来 watch 的资源
4. 定义 Operator 的协调（reconcile）逻辑
5. 使用 Operator SDK 构建并生成 Operator 部署清单文件

每种 Operator 类型都有不同的功能集，在选择项目的类型时，重要的是要了解每种项目类型的功能和局限性以及 Operator 的用例。



示例

我们平时在部署一个简单的 Webserver 到 Kubernetes 集群中的时候，都需要先编写一个 Deployment 的控制器，然后创建一个 Service 对象，通过 Pod 的 label 标签进行关联，最后通过 Ingress 或者 type=NodePort 类型的 Service 来暴露服务，每次都需要这样操作，是不是略显麻烦，我们就可以创建一个自定义的资源对象，通过我们的 CRD 来描述我们要部署的应用信息，比如镜像、服务端口、环境变量等等，然后创建我们的自定义类型的资源对象的时候，通过控制器去创建对应的 Deployment 和 Service，是不是就方便很多了，相当于我们用一个资源清单去描述了 Deployment 和 Service 要做的两件事情。

这里我们将创建一个名为 AppService 的 CRD 资源对象，然后定义如下的资源清单进行应用部署：

```
apiVersion: app.example.com/v1
kind: AppService
```

```
metadata:
  name: nginx-app
spec:
  size: 2
  image: nginx:1.7.9
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30002
```

通过这里的自定义的 AppService 资源对象去创建副本数为2的 Pod，然后通过 `nodePort=30002` 的端口去暴露服务，接下来我们就来一步一步的实现我们这里的这个简单的 Operator 应用。

开发环境

要开发 Operator 自然 Kubernetes 集群是少不了的，还需要 Golang 的环境，这里的安装就不多说了。

> Docker 版本需要 17.03+，Kubectl 版本为 v1.11.3+，如果使用`apiextensions.k8s.io/v1` 版本的 CRD，则需要 v1.16.0+ 版本。

然后需要安装 `operator-sdk`，operator sdk 安装方法非常多，我们可以直接在 github 上面下载需要使用的版本，然后放置到 PATH 环境下面即可，当然也可以将源码 clone 到本地手动编译安装即可，如果你是 Mac，当然还可以使用常用的 brew 工具进行安装：

```
$ brew install operator-sdk
......
$ operator-sdk version
operator-sdk version: "v1.1.0", commit: "9d27e224efac78fcc9354ece4e43a50eb30ea968", kubernetes version: "v1.18.2", go version: "go1.15
$ go version
go version go1.15.3 darwin/amd64
```

我们这里使用的 sdk 版本是 `v1.1.0`，其他安装方法可以参考文档：https://sdk.operatorframework.io/docs/installation/install-operator-sdk/

创建项目

环境准备好了，接下来就可以使用 `operator-sdk` 直接创建一个新的项目了，命令格式为：`operator-sdk` init。

按照上面我们预先定义的 CRD 资源清单，我们这里可以这样创建：

```
# 创建项目目录
$ mkdir -p opdemo && cd opdemo
$ export GO111MODULE=on  # 使用gomodules包管理工具
$ export GOPROXY="https://goproxy.cn"
# 使用包代理，加速# 使用 sdk 创建一个名为 opdemo 的 operator 项目，如果在 GOPATH 之外需要指定 repo 参数
$ go mod init github.com/cnych/opdemo/v2
# 使用下面的命令初始化项目
$ operator-sdk init --domain ydzs.io --license apache2 --owner "cnych"
Writing scaffold for you to edit...
Get controller runtime:
$ go get sigs.k8s.io/controller-runtime@v0.6.2
go: downloading sigs.k8s.io/controller-runtime v0.6.2
go: downloading k8s.io/client-go v0.18.6
go: downloading k8s.io/utils v0.0.0-20200603063816-c1c6865ac451
go: downloading github.com/prometheus/procfs v0.0.11
go: downloading golang.org/x/net v0.0.0-20200520004742-59133d7f0dd7
go: downloading github.com/golang/groupcache v0.0.0-20190129154638-5b532d6fd5ef
go: downloading k8s.io/apiextensions-apiserver v0.18.6
Update go.mod:
$ go mod tidy
Running make:
$ make
/Users/ych/devs/projects/go/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
go build -o bin/manager main.go
Next: define a resource with:
$ operator-sdk create api
```

初始化完成后的项目结构如下所示：

```
$ tree -L 2
.
├── Dockerfile
├── Makefile
├── PROJECT
├── bin
│   └── manager
├── config
│   ├── certmanager
│   ├── default
│   ├── manager
│   ├── prometheus
│   ├── rbac
│   ├── scorecard
│   └── webhook
├── go.mod
├── go.sum
├── hack
│   └── boilerplate.go.txt
└── main.go

10 directories, 8 files
```

到这里一个全新的 Operator 项目就新建完成了。

项目结构

使用 `operator-sdk init` 命令创建新的 Operator 项目后，项目目录就包含了很多生成的文件夹和文件。

- go.mod/go.sum - Go Modules 包管理清单，用来描述当前 Operator 的依赖包。
- main.go 文件，使用 operator-sdk API 初始化和启动当前 Operator 的入口。
- deploy - 包含一组用于在 Kubernetes 集群上进行部署的通用的 Kubernetes 资源清单文件。
- pkg/apis - 包含定义的 API 和自定义资源（CRD）的目录树，这些文件允许 sdk 为 CRD 生成代码并注册对应的类型，以便正确解码自定义资源对象。
- pkg/controller - 用于编写所有的操作业务逻辑的地方
- version - 版本定义
- build - Dockerfile 定义目录

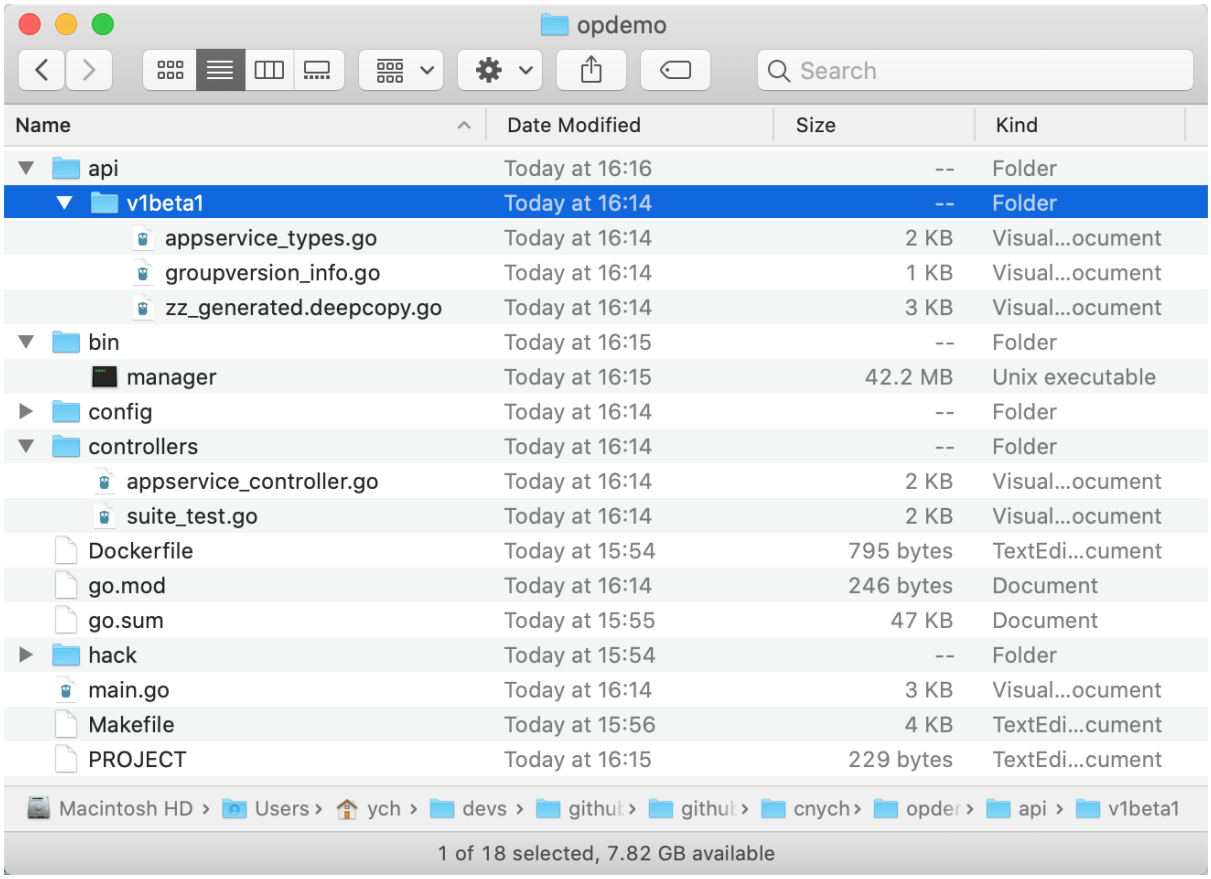
我们主要需要编写的是 `pkg` 目录下面的 api 定义以及对应的 controller 实现。

添加 API

接下来为我们的自定义资源添加一个新的 API，按照上面我们预定义的资源清单文件，在 Operator 相关根目录下面执行如下命令：

```
$ operator-sdk create api --group app --version v1beta1 --kind AppService
Create Resource [y/n]
y
Create Controller [y/n]
y
Writing scaffold for you to edit...
api/v1beta1/appservice_types.go
controllers/appservice_controller.go
Running make:
$ make
/Users/ych/devs/projects/go/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
go build -o bin/manager main.go
```

这里我们添加了一个 group 为 app，版本为 v1beta1 的 AppService 的资源对象，添加完成后，我们可以看到类似于下面的这样项目结构，我们可以看到生成了对应的 api 和 controllers 包：



自定义 API

打开源文件 `api/v1beta1/appservice_types.go`，需要我们根据我们的需求去自定义结构体 `AppServiceSpec`，我们最上面预定义的资源清单中就有 `size`、`image`、`ports` 这些属性，所有我们需要用到的属性都需要在这个结构体中进行定义：

```
// AppServiceSpec defines the desired state of AppService
type AppServiceSpec struct {
  // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
  // Important: Run "make" to regenerate code after modifying this file

  Size      *int32                        `json:"size"`
  Image     string                        `json:"image"`
  Resources corev1.ResourceRequirements   `json:"resources,omitempty"`
  Envs      []corev1.EnvVar               `json:"envs,omitempty"`
  Ports     []corev1.ServicePort          `json:"ports,omitempty"`
```

代码中会涉及到一些包名的导入，由于包名较多，所以我们会使用一些别名进行区分，主要的包含下面几个：

```
import (
    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)
```

这里的 resources、envs、ports 的定义都是直接引用的 `"k8s.io/api/core/v1"` 中定义的结构体，而且需要注意的是我们这里使用的是 `ServicePort`，而不是像传统的 Pod 中定义的 ContanerPort，这是因为我们的资源清单中不仅要描述容器的 Port，还要描述 Service 的 Port。

然后一个比较重要的结构体 `AppServiceStatus` 用来描述资源的状态，当然我们可以根据需要去自定义状态的描述，我这里就偷懒直接使用 Deployment 的状态了：

```
// AppServiceStatus defines the observed state of AppService
type AppServiceStatus struct {
  // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
  // Important: Run "make" to regenerate code after modifying this file
```

```
  appsv1.DeploymentStatus `json:",inline"`
}
```

定义完成后，在项目根目录下面执行如下命令：

```
$ make
/Users/ych/devs/projects/go/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
go build -o bin/manager main.go
```

该命令会使用我们更新后的资源对象结构重新自动生成一些代码，这样我们就算完成了对自定义资源对象的 API 的声明。

实现业务逻辑

上面 API 描述声明完成了，接下来就需要我们来进行具体的业务逻辑实现了，编写具体的 controller 实现，打开源文件 `controllers/appservice_controller.go`，需要我们去更改的地方也不是很多，核心的就是 `Reconcile` 方法，该方法就是去不断的 watch 资源的状态，然后根据状态的不同去实现各种操作逻辑。

首先 sdk 为我们搭建了一个基本的 reconciler 结构，几乎每一个调谐器器都需要记录日志，并且能够获取对象，所以可以直接使用。

```
// AppServiceReconciler reconciles a AppService object
type AppServiceReconciler struct {
  client.Client
  Log    logr.Logger
  Scheme *runtime.Scheme
}
```

`Reconcile` 实际上是对单个对象进行调谐，我们的 Request 只是有一个名字，但我们可以使用 client 从缓存中获取这个对象。我们**返回一个空的结果，没有错误**，这就向 controller-runtime 表明我们**已经成功地对这个对象进行了调谐**，在有一些变化之前不需要再尝试调谐。

大多数控制器需要**一个日志句柄和一个上下文**，所以我们在 Reconcile 中将他们初始化。上下文是用来允许取消请求的，它是所有 client 方法的第一个参数。

controller-runtime 通过一个名为 logr 的库使用结构化的日志记录。日志记录的工作原理是将键值对附加到静态消息中，我们可以在我们的调谐方法的顶部预先分配一些键值对，让这些数据附加到这个调谐器的所有日志行。

```
// +kubebuilder:rbac:groups=app.ydzs.io,resources=appservices,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=app.ydzs.io,resources=appservices/status,verbs=get;update;patch

func (r *AppServiceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
   _ = context.Background()
   _ = r.Log.WithValues("appservice", req.NamespacedName)

   // Reconcile successful - don't requeue
   // return ctrl.Result{}, nil
   // Reconcile failed due to error - requeue
   // return ctrl.Result{}, err
   // Requeue for any reason other than an error
   // return ctrl.Result{Requeue: true}, nil

   // your logic here

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

最后，我们将 Reconcile 添加到 manager 中，这样当 manager 启动时它就会被启动。现在，我们只是注意到这个 Reconcile 是在 `AppService` 上运行的，以后，我们也会用这个来标记其他的对象。

```
func (r *AppServiceReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
    For(&appv1beta1.AppService{}).
    Complete(r)
}
```

现在我们已经了解了 Reconcile 的基本结构，我们来补充一下 AppService 的调谐逻辑。核心代码如下：

```
// +kubebuilder:rbac:groups=app.ydzs.io,resources=appservices,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=app.ydzs.io,resources=appservices/status,verbs=get;update;patch

func (r *AppServiceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
  ctx := context.Background()
  log := r.Log.WithValues("appservice", req.NamespacedName)

  // 业务逻辑实现
  // 获取 AppService 实例
  var appService appv1beta1.AppService
  err := r.Get(ctx, req.NamespacedName, &appService)
  if err != nil {
    // MyApp 被删除的时候，忽略
    if client.IgnoreNotFound(err) != nil {
      return ctrl.Result{}, err
    }
    return ctrl.Result{}, nil
  }

  log.Info("fetch appservice objects", "appservice", appService)

  // 如果不存在，则创建关联资源
  // 如果存在，判断是否需要更新
  //   如果需要更新，则直接更新
  //   如果不需要更新，则正常返回
  deploy := &appsv1.Deployment{}
  if err := r.Get(ctx, req.NamespacedName, deploy); err != nil && errors.IsNotFound(err) {
    // 1. 关联 Annotations
    data, _ := json.Marshal(appService.Spec)
    if appService.Annotations != nil {
      appService.Annotations[oldSpecAnnotation] = string(data)
    } else {
      appService.Annotations = map[string]string{oldSpecAnnotation: string(data)}
    }
    if err := r.Client.Update(ctx, &appService); err != nil {
      return ctrl.Result{}, err
    }
    // 创建关联资源
    // 2. 创建 Deployment
    deploy := resources.NewDeploy(&appService)
    if err := r.Client.Create(ctx, deploy); err != nil {
      return ctrl.Result{}, err
    }
    // 3. 创建 Service
    service := resources.NewService(&appService)
    if err := r.Create(ctx, service); err != nil {
      return ctrl.Result{}, err
    }
    return ctrl.Result{}, nil
  }
  oldspec := appv1beta1.AppServiceSpec{}
  if err := json.Unmarshal([]byte(appService.Annotations[oldSpecAnnotation]), &oldspec); err != nil {
    return ctrl.Result{}, err
  }
  // 当前规范与旧的对象不一致，则需要更新
  if !reflect.DeepEqual(appService.Spec, oldspec) {
    // 更新关联资源
    newDeploy := resources.NewDeploy(&appService)
    oldDeploy := &appsv1.Deployment{}
    if err := r.Get(ctx, req.NamespacedName, oldDeploy); err != nil {
      return ctrl.Result{}, err
    }
    oldDeploy.Spec = newDeploy.Spec
    if err := r.Client.Update(ctx, oldDeploy); err != nil {
      return ctrl.Result{}, err
    }

    newService := resources.NewService(&appService)
    oldService := &corev1.Service{}
    if err := r.Get(ctx, req.NamespacedName, oldService); err != nil {
      return ctrl.Result{}, err
    }
    // 需要指定 ClusterIP 为之前的，不然更新会报错
    newService.Spec.ClusterIP = oldService.Spec.ClusterIP
    oldService.Spec = newService.Spec
    if err := r.Client.Update(ctx, oldService); err != nil {
      return ctrl.Result{}, err
    }
    return ctrl.Result{}, nil
  }
  return ctrl.Result{}, nil
}
```

上面就是业务逻辑实现的核心代码，逻辑很简单，就是去判断资源是否存在，不存在，则直接创建新的资源，创建新的资源除了需要创建 Deployment 资源外，还需要创建 Service 资源对象，因为这就是我们的需求，当然你还可以自己去扩展，比如在创建一个 Ingress 对象。更新也是一样的，去对比新旧对象的声明是否一致，不一致则需要更新，同样的，两种资源都需要更新的。

另外两个核心的方法就是上面的 `resources.NewDeploy(instance)` 和 `resources.NewService(instance)` 方法，这两个方法实现逻辑也很简单，就是根据 CRD 中的声明去填充 Deployment 和 Service 资源对象的 Spec 对象即可。

NewDeploy 方法实现如下：

```
package resources

import (
  appv1beta1 "github.com/cnych/opdemo/v2/api/v1beta1"
  appsv1 "k8s.io/api/apps/v1"
  corev1 "k8s.io/api/core/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/apimachinery/pkg/runtime/schema"
)

func NewDeploy(app *appv1beta1.AppService) *appsv1.Deployment {
  labels := map[string]string{"app": app.Name}
  selector := &metav1.LabelSelector{MatchLabels: labels}
  return &appsv1.Deployment{
    TypeMeta: metav1.TypeMeta{
      APIVersion: "apps/v1",
      Kind:       "Deployment",
    },
    ObjectMeta: metav1.ObjectMeta{
      Name:      app.Name,
      Namespace: app.Namespace,

      OwnerReferences: []metav1.OwnerReference{
        *metav1.NewControllerRef(app, schema.GroupVersionKind{
          Group:  appv1beta1.GroupVersion.Group,
          Version: appv1beta1.GroupVersion.Version,
          Kind: appv1beta1.Kind,
        }),
      },
    },
    Spec: appsv1.DeploymentSpec{
      Replicas: app.Spec.Size,
      Template: corev1.PodTemplateSpec{
        ObjectMeta: metav1.ObjectMeta{
          Labels: labels,
        },
        Spec: corev1.PodSpec{
          Containers: newContainers(app),
        },
      },
      Selector: selector,
    },
  }
}

func newContainers(app *appv1beta1.AppService) []corev1.Container {
  containerPorts := []corev1.ContainerPort{}
  for _, svcPort := range app.Spec.Ports {
    cport := corev1.ContainerPort{}
    cport.ContainerPort = svcPort.TargetPort.IntVal
    containerPorts = append(containerPorts, cport)
  }
  return []corev1.Container{
    {
      Name: app.Name,
      Image: app.Spec.Image,
      Resources: app.Spec.Resources,
      Ports: containerPorts,
      ImagePullPolicy: corev1.PullIfNotPresent,
      Env: app.Spec.Envs,
    },
  }
}
```

newService 对应的方法实现如下：

```
package resources

import (
  "k8s.io/apimachinery/pkg/runtime/schema"
```

```
  corev1 "k8s.io/api/core/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  appv1beta1 "github.com/cnych/opdemo/v2/api/v1beta1"
)

func NewService(app *appv1beta1.AppService) *corev1.Service {
  return &corev1.Service {
    TypeMeta: metav1.TypeMeta {
      Kind: "Service",
      APIVersion: "v1",
    },
    ObjectMeta: metav1.ObjectMeta{
      Name: app.Name,
      Namespace: app.Namespace,
      OwnerReferences: []metav1.OwnerReference{
        *metav1.NewControllerRef(app, schema.GroupVersionKind{
          Group: appv1beta1.GroupVersion.Group,
          Version: appv1beta1.GroupVersion.Version,
          Kind: appv1beta1.Kind,
        }),
      },
    },
    Spec: corev1.ServiceSpec{
      Type: corev1.ServiceTypeNodePort,
      Ports: app.Spec.Ports,
      Selector: map[string]string{
        "app": app.Name,
      },
    },
  }
}
```

这样我们就实现了 AppService 这种资源对象的业务逻辑。

调试

如果我们本地有一个可以访问的 Kubernetes 集群，我们也可以直接进行调试，在本地用户 `~/.kube/config` 文件中配置集群访问信息，下面的信息表明可以访问 Kubernetes 集群：

```
$ kubectl cluster-info
Kubernetes master is running at https://ydzs-master:6443
KubeDNS is running at https://ydzs-master:6443/api/v1/namespaces/kube-system/services/kube-dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

首先，需要在集群中安装 CRD 对象：

```
$ make install
/Users/ych/devs/projects/go/bin/controller-gen "crd:trivialVersions=true" rbac:roleName=manager-role webhook paths="./..." output:crd:
/usr/local/bin/kustomize build config/crd | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/appservices.app.ydzs.io created
$ kubectl get crd |grep appservice
appservices.app.ydzs.io                                 2020-10-16T09:26:05Z
```

当我们通过 `kubectl get crd` 命令获取到我们定义的 CRD 资源对象，就证明我们定义的 CRD 安装成功了。其实现在只是 CRD 的这个声明安装成功了，但是我们这个 CRD 的具体业务逻辑实现方式还在我们本地，并没有部署到集群之中，我们可以通过下面的命令来在本地项目中启动 Operator 的调试：

```
$ make run
/Users/ych/devs/projects/go/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
/Users/ych/devs/projects/go/bin/controller-gen "crd:trivialVersions=true" rbac:roleName=manager-role webhook paths="./..." output:crd:
go run ./main.go
I1016 17:29:44.477805   51603 request.go:621] Throttling request took 1.048368705s, request: GET:https://ydzs-master:6443/apis/snapsho
2020-10-16T17:29:45.487+0800    INFO    controller-runtime.metrics      metrics server is starting to listen    {"addr": ":8080"}
2020-10-16T17:29:45.487+0800    INFO    setup   starting manager
2020-10-16T17:29:45.488+0800    INFO    controller      Starting EventSource    {"reconcilerGroup": "app.ydzs.io", "reconcilerKind": "
2020-10-16T17:29:45.488+0800    INFO    controller-runtime.manager      starting metrics server {"path": "/metrics"}
2020-10-16T17:29:45.690+0800    INFO    controller      Starting Controller     {"reconcilerGroup": "app.ydzs.io", "reconcilerKind": "
2020-10-16T17:29:45.690+0800    INFO    controller      Starting workers        {"reconcilerGroup": "app.ydzs.io", "reconcilerKind": "
......
```

上面的命令会在本地运行 Operator 应用，通过 `~/.kube/config` 去关联集群信息，现在我们去添加一个 AppService 类型的资源然后观察本地 Operator 的变化情况，资源清单文件就是我们上面预定义的（config/samples/app_v1beta1_appservice.yaml）：

```
apiVersion: app.ydzs.io/v1beta1
kind: AppService
metadata:
  name: nginx
spec:
  size: 2
  image: nginx:1.7.9
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30002
```

直接创建这个资源对象：

```
$ kubectl apply -f config/samples/app_v1beta1_appservice.yaml
appservice.app.ydzs.io/nginx-app created
```

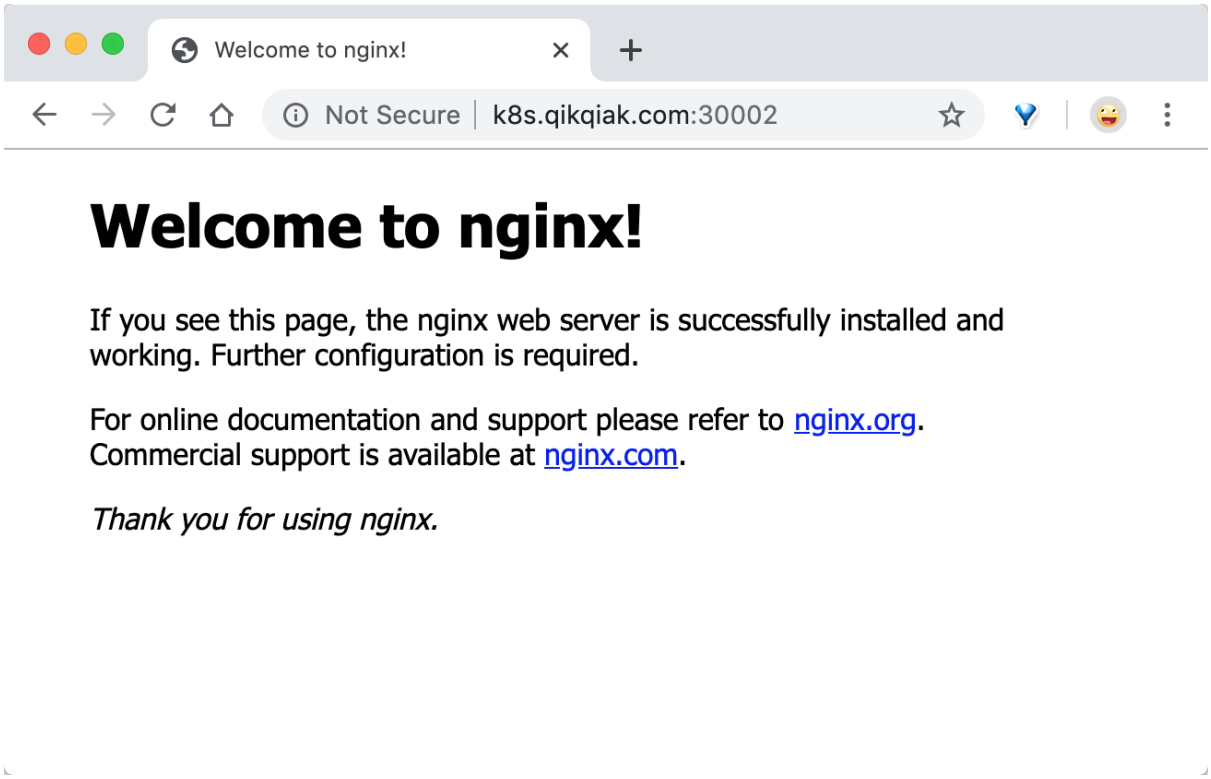
我们可以看到我们的应用创建成功了，这个时候查看 Operator 的调试窗口会有如下的信息出现：

```
......
{"level":"info","ts":1559207416.670523,"logger":"controller_appservice","msg":"Reconciling AppService","Request.Namespace":"default","
{"level":"info","ts":1559207417.004226,"logger":"controller_appservice","msg":"Reconciling AppService","Request.Namespace":"default","
{"level":"info","ts":1559207417.004331,"logger":"controller_appservice","msg":"Reconciling AppService","Request.Namespace":"default","
{"level":"info","ts":1559207418.33779,"logger":"controller_appservice","msg":"Reconciling AppService","Request.Namespace":"default","R
{"level":"info","ts":1559207418.951193,"logger":"controller_appservice","msg":"Reconciling AppService","Request.Namespace":"default","
......
```

然后我们可以去查看集群中是否有符合我们预期的资源出现：

```
$ kubectl get AppService
NAME        AGE
nginx       2m8s
$ kubectl get deploy
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   p               2/2     2            2           2m20s
$ kubectl get svc
NAME             TYPE           CLUSTER-IP       EXTERNAL-IP             PORT(S)          AGE
nginx            NodePort       10.111.179.0     <none>                  80:30002/TCP     2m23s
```

看到了吧，我们定义了两个副本（size=2），这里就出现了两个 Pod，还有一个 NodePort=30002 的 Service 对象，我们可以通过该端口去访问下应用：



如果应用在安装过程中出现了任何问题，我们都可以通过本地的 Operator 调试窗口找到有用的信息，然后调试修改即可。

清理：

```
$ kubectl delete -f config/samples/app_v1beta1_appservice.yaml
$ make uninstall
```

部署

自定义的资源对象现在测试通过了，但是如果我们将本地的调试控制器终止掉，我们可以猜想到就没办法处理 AppService 资源对象的一些操作了，所以我们需要将我们的业务逻辑实现部署到集群中去。

执行下面的命令构建 Operator 应用打包成 Docker 镜像：

```
$ export USERNAME=<dockerbub-username>
$ make docker-build IMG=$USERNAME/opdemo:v1.0.0
......
Successfully built 29cd605c4ad2
Successfully tagged cnych/opdemo:v1.0.0
INFO[0041] Operator build complete.
```

镜像构建成功后，推送到 docker hub：

```
$ make docker-push IMG=$USERNAME/opdemo:v1.0.0
```

镜像推送成功后，使用下面的命令直接部署控制器：

```
$ make deploy IMG=$USERNAME/opdemo:v1.0.0
```

现在 Operator 的资源清单文件准备好了，然后就可以使用下面的命令来部署 CRD 资源对象了：

```
$ kubectl apply -f config/samples/app_v1beta1_appservice.yaml
$ kubectl get crd |grep myapp
myapps.app.ydzs.io   2020-11-06T07:06:54Z
```

到这里我们的 CRD 和 Operator 实现都已经安装成功了。