自定义控制器

 本文主要介绍如何为 CRD 资源编写自定义的控制器

介绍

上节课我们已经学习了如何使用 code-generator 来进行代码自动生成，通过代码自动生成可以帮我们自动生成 CRD 资源对象客户端访问的 ClientSet、Informer、Lister 等工具包，接下来我们就来了解下如何编写一个自定义的控制器。

CRD 定义

这里我们来针对前面课程中介绍的 CronTab 自定义资源对象编写一个自定义的控制器，对应的资源清单文件如下所示：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
  - name: v1beta1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        description: Define CronTab YAML Spec
        type: object
        properties:
          spec:
            type: object
            properties:
              cronSpec:
                type: string
              image:
                type: string
              replicas:
                type: integer
  scope: Namespaced
  names:
    kind: CronTab
    plural: crontabs
    singular: crontab
    shortNames:
    - ct
```

我们需要针对上面的规范来定义资源结构，首先初始化项目：

```
$ mkdir -p github.com/cnych/controller-demo && cd github.com/cnych/controller-demo
# 初始化项目
$ go mod init github.com/cnych/controller-demo
go: creating new go.mod: module github.com/cnych/controller-demo
# 获取依赖
$ go get k8s.io/apimachinery@v0.17.9
$ go get -d k8s.io/code-generator@v0.17.9
$ go get k8s.io/client-go@v0.17.9
```

然后初始化 CRD 资源类型，建立好自己的 CRD 结构体，然后使用code-generator 生成客户端代码：

```
$ mkdir -p pkg/apis/stable/v1beta1
```

在该文件夹中新建 `doc.go` 文件，内容如下所示：

```
// +k8s:deepcopy-gen=package
// +groupName=stable.example.com

package v1beta1
```

根据 CRD 的规范定义，这里我们定义的 group 为 `stable.example.com`，版本为 `v1beta1` ，在顶部添加了一个代码自动生成的 `deepcopy-gen` 的 tag，为整个包中的类型生成深拷贝方法。

然后就是非常重要的资源对象的结构体定义，新建 `type.go` 文件，内容如下所示：

```
package v1beta1

import (
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// +genclient
// +genclient:noStatus
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// 根据 CRD 定义 CronTab 结构体
type CronTab struct {
  metav1.TypeMeta `json:",inline"`
  // +optional
  metav1.ObjectMeta `json:"metadata,omitempty"`
  Spec              CronTabSpec `json:"spec"`
}

// +k8s:deepcopy-gen=false

type CronTabSpec struct {
  CronSpec string `json:"cronSpec"`
  Image    string `json:"image"`
  Replicas int    `json:"replicas"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// CronTab 资源列表
type CronTabList struct {
  metav1.TypeMeta `json:",inline"`

  // 标准的 list metadata
  // +optional
  metav1.ListMeta `json:"metadata,omitempty"`

  Items []CronTab `json:"items"`
}
```

然后可以参考系统内置的资源对象，还需要提供 AddToScheme 与 Resource 两个变量供 client 注册，新建 `register.go` 文件，内容如下所示：

```
package v1beta1

import (
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/apimachinery/pkg/runtime"
  "k8s.io/apimachinery/pkg/runtime/schema"
)

// GroupName is the group name use in this package
const GroupName = "stable.example.com"
// 注册自己的自定义资源
var SchemeGroupVersion = schema.GroupVersion{Group: GroupName, Version: "v1beta1"}

// Resource takes an unqualified resource and returns a Group qualified GroupResource
func Resource(resource string) schema.GroupResource {
  return SchemeGroupVersion.WithResource(resource).GroupResource()
}

var (
  // SchemeBuilder initializes a scheme builder
  SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
  // AddToScheme is a global function that registers this API group & version to a scheme
```

```
  AddToScheme = SchemeBuilder.AddToScheme
)

// Adds the list of known types to Scheme.
func addKnownTypes(scheme *runtime.Scheme) error {
  // 添加 CronTab 与 CronTabList 这两个资源到 scheme
  scheme.AddKnownTypes(SchemeGroupVersion,
    &CronTab{},
    &CronTabList{},
  )
  metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
  return nil
}
```

最终的项目结构如下所示：

```
$ tree .
.
├── go.mod
├── go.sum
└── pkg
    └── apis
        └── stable
            └── v1beta1
                ├── doc.go
                ├── register.go
                └── types.go

4 directories, 5 files
```

生成代码

上面我们准备好资源的 API 资源类型后，就可以使用开始生成 CRD 资源的客户端使用的相关代码了。

首先创建生成代码的脚本，下面这些脚本均来源于 sample-controller 提供的示例：

```
$ mkdir hack && cd hack
```

在该目录下面新建 `tools.go` 文件，添加 code-generator 依赖，因为在没有代码使用 code-generator 时，`go module` 默认不会为我们依赖此包。文件内容如下所示：

```
// +build tools

// 建立 tools.go 来依赖 code-generator
// 因为在没有代码使用 code-generator 时，go module 默认不会为我们依赖此包.
package tools

import _ "k8s.io/code-generator"
```

然后新建 `update-codegen.sh` 脚本，用来配置代码生成的脚本：

```
#!/usr/bin/env bash

set -o errexit
set -o nounset
set -o pipefail

SCRIPT_ROOT=$(dirname "${BASH_SOURCE[0]}")/..
CODEGEN_PKG=${CODEGEN_PKG:-$(cd "${SCRIPT_ROOT}"; ls -d -1 ./vendor/k8s.io/code-generator 2>/dev/null || echo ../code-generator)}

bash "${CODEGEN_PKG}"/generate-groups.sh "deepcopy,client,informer,lister" \
  github.com/cnych/controller-demo/pkg/client github.com/cnych/controller-demo/pkg/apis \
  stable:v1beta1 \
  --output-base "${SCRIPT_ROOT}"/../../.. \
  --go-header-file "${SCRIPT_ROOT}"/hack/boilerplate.go.txt

# To use your own boilerplate text append:
#   --go-header-file "${SCRIPT_ROOT}"/hack/custom-boilerplate.go.txt
```

同样还有上节课提到的 `verify-codegen.sh` 脚本，用来校验生成的代码是否是最新的：

```
#!/usr/bin/env bash

set -o errexit
set -o nounset
set -o pipefail

SCRIPT_ROOT=$(dirname "${BASH_SOURCE[0]}")/..

DIFFROOT="${SCRIPT_ROOT}/pkg"
TMP_DIFFROOT="${SCRIPT_ROOT}/_tmp/pkg"
_tmp="${SCRIPT_ROOT}/_tmp"

cleanup() {
  rm -rf "${_tmp}"
}
trap "cleanup" EXIT SIGINT

cleanup

mkdir -p "${TMP_DIFFROOT}"
cp -a "${DIFFROOT}"/* "${TMP_DIFFROOT}"

"${SCRIPT_ROOT}/hack/update-codegen.sh"
echo "diffing ${DIFFROOT} against freshly generated codegen"
ret=0
diff -Naupr "${DIFFROOT}" "${TMP_DIFFROOT}" || ret=$?
cp -a "${TMP_DIFFROOT}"/* "${DIFFROOT}"
if [[ $ret -eq 0 ]]
then
  echo "${DIFFROOT} up to date."
else
  echo "${DIFFROOT} is out of date. Please run hack/update-codegen.sh"
  exit 1
fi
```

还有一个为生成的代码文件添加头部内容的 `boilerplate.go.txt` 文件，内容如下所示，其实就是为每个生成的代码文件头部添加上下面的开源协议声明：

```
/*
Copyright The Kubernetes Authors.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/
```

接下来我们就可以来执行代码生成的脚本了，首先将依赖包放置到 vendor 目录中去：

```
$ go mod vendor
```

然后执行脚本生成代码：

```
$ chmod +x ./hack/update-codegen.sh
$ ./hack/update-codegen.sh
Generating deepcopy funcs
Generating clientset for stable:v1beta1 at github.com/cnych/controller-demo/pkg/client/clientset
Generating listers for stable:v1beta1 at github.com/cnych/controller-demo/pkg/client/listers
Generating informers for stable:v1beta1 at github.com/cnych/controller-demo/pkg/client/informers
```

代码生成后，整个项目的 pkg 包变成了下面的样子：

```
$ tree pkg
pkg
├── apis
│   └── stable
│       └── v1beta1
```

```
│           ├── doc.go
│           ├── register.go
│           ├── types.go
│           └── zz_generated.deepcopy.go
└── client
    ├── clientset
    │   └── versioned
    │       ├── clientset.go
    │       ├── doc.go
    │       ├── fake
    │       │   ├── clientset_generated.go
    │       │   ├── doc.go
    │       │   └── register.go
    │       ├── scheme
    │       │   ├── doc.go
    │       │   └── register.go
    │       └── typed
    │           └── stable
    │               └── v1beta1
    │                   ├── crontab.go
    │                   ├── doc.go
    │                   ├── fake
    │                   │   ├── doc.go
    │                   │   ├── fake_crontab.go
    │                   │   └── fake_stable_client.go
    │                   ├── generated_expansion.go
    │                   └── stable_client.go
    ├── informers
    │   └── externalversions
    │       ├── factory.go
    │       ├── generic.go
    │       ├── internalinterfaces
    │       │   └── factory_interfaces.go
    │       └── stable
    │           ├── interface.go
    │           └── v1beta1
    │               ├── crontab.go
    │               └── interface.go
    └── listers
        └── stable
            └── v1beta1
                ├── crontab.go
                └── expansion_generated.go

20 directories, 26 files
```

仔细观察可以发现 `pkg/apis/stable/v1beta1` 目录下面多了一个 `zz_generated.deepcopy.go` 文件,在 `pkg/client` 文件夹下生成了 clientset和 informers 和 listers 三个目录，有了这几个自动生成的客户端相关操作包，我们就可以去访问 CRD 资源了，可以和使用内置的资源对象一样去对 CronTab 进行 List 和 Watch 操作了。

编写控制器

可以参考前面编写控制器的方式，首先要先获取访问资源对象的 ClientSet，在项目根目录下面新建 main.go 文件

```go
package main

import (
  "flag"
  "os"
  "os/signal"
  "path/filepath"
  "syscall"
  "time"

  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/rest"
  "k8s.io/client-go/tools/clientcmd"
  "k8s.io/client-go/util/homedir"
  "k8s.io/klog"

  clientset "github.com/cnych/controller-demo/pkg/client/clientset/versioned"
  informers "github.com/cnych/controller-demo/pkg/client/informers/externalversions"
)

var (
  onlyOneSignalHandler = make(chan struct{})
  shutdownSignals      = []os.Signal{os.Interrupt, syscall.SIGTERM}
)

// SetupSignalHandler 注册 SIGTERM 和 SIGINT 信号
// 返回一个 stop channel，该通道在捕获到第一个信号时被关闭
// 如果捕捉到第二个信号，程序将直接退出
```

```
func setupSignalHandler() (stopCh <-chan struct{}) {
  // 当调用两次的时候 panics
  close(onlyOneSignalHandler)

  stop := make(chan struct{})
  c := make(chan os.Signal, 2)
  // Notify 函数让 signal 包将输入信号转发到 c
  // 如果没有列出要传递的信号，会将所有输入信号传递到 c；否则只传递列出的输入信号
  // 参考文档：https://cloud.tencent.com/developer/article/1645996
  signal.Notify(c, shutdownSignals...)
  go func() {
    <-c
    close(stop)
    <-c
    os.Exit(1) // 第二个信号，直接退出
  }()

  return stop
}

func initClient() (*kubernetes.Clientset, *rest.Config, error) {
  var err error
  var config *rest.Config
  // inCluster（Pod）、KubeConfig（kubectl）
  var kubeconfig *string

  if home := homedir.HomeDir(); home != "" {
    kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"), "(可选) kubeconfig 文件的绝对路径")
  } else {
    kubeconfig = flag.String("kubeconfig", "", "kubeconfig 文件的绝对路径")
  }
  flag.Parse()

  // 首先使用 inCluster 模式(需要去配置对应的 RBAC 权限，默认的sa是default->是没有获取deployments的List权限)
  if config, err = rest.InClusterConfig(); err != nil {
    // 使用 KubeConfig 文件创建集群配置 Config 对象
    if config, err = clientcmd.BuildConfigFromFlags("", *kubeconfig); err != nil {
      panic(err.Error())
    }
  }

  // 已经获得了 rest.Config 对象
  // 创建 Clientset 对象
  kubeclient, err := kubernetes.NewForConfig(config)
  if err != nil {
    return nil, config, err
  }
  return kubeclient, config, nil
}

func main() {
  flag.Parse()

  //设置一个信号处理，应用于优雅关闭
  stopCh := setupSignalHandler()

  _, cfg, err := initClient()
  if err != nil {
    klog.Fatalf("Error building kubernetes clientset: %s", err.Error())
  }

  // 实例化一个 CronTab 的 ClientSet
  crontabClient, err := clientset.NewForConfig(cfg)
  if err != nil {
    klog.Fatalf("Error building crontab clientset: %s", err.Error())
  }

  // informerFactory 工厂类， 这里注入我们通过代码生成的 client
  // clent 主要用于和 API Server 进行通信，实现 ListAndWatch
  crontabInformerFactory := informers.NewSharedInformerFactory(crontabClient, time.Second*30)

  // 实例化自定义控制器
  controller := NewController(crontabInformerFactory.Stable().V1beta1().CronTabs())

  // 启动 informer，开始List & Watch
  go crontabInformerFactory.Start(stopCh)

  if err = controller.Run(2, stopCh); err != nil {
    klog.Fatalf("Error running controller: %s", err.Error())
  }
}
```

首先初始化一个用于访问 CronTab 资源的 ClientSet 对象，然后同样新建一个 CronTab 的 InformerFactory 实例，通过这个工厂实例可以去启动 Informer 开始对 CronTab 的 List 和 Watch 操作，然后同样我们要自己去封装一个自定义的控制器，在这个

控制器里面去实现一个控制循环，不断对 CronTab 的状态进行调谐。

在项目根目录下新建 `controller.go` 文件，内容如下所示：

```
package main

import (
  "fmt"
  "time"

  "k8s.io/apimachinery/pkg/api/errors"
  "k8s.io/apimachinery/pkg/util/runtime"
  "k8s.io/apimachinery/pkg/util/wait"
  "k8s.io/client-go/tools/cache"
  "k8s.io/client-go/util/workqueue"
  "k8s.io/klog"

  crdv1beta1 "github.com/cnych/controller-demo/pkg/apis/stable/v1beta1"
  informers "github.com/cnych/controller-demo/pkg/client/informers/externalversions/stable/v1beta1"
)

type Controller struct {
  informer  informers.CronTabInformer
  workqueue workqueue.RateLimitingInterface
}

func NewController(informer informers.CronTabInformer) *Controller {
  //使用client 和前面创建的 Informer，初始化了自定义控制器
  controller := &Controller{
    informer: informer,
    // WorkQueue 的实现，负责同步 Informer 和控制循环之间的数据
    workqueue: workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), "CronTab"),
  }

  klog.Info("Setting up crontab event handlers")

  // informer 注册了三个 Handler（AddFunc、UpdateFunc 和 DeleteFunc）
  // 分别对应 API 对象的“添加”“更新”和“删除”事件。
  // 而具体的处理操作，都是将该事件对应的 API 对象加入到工作队列中
  informer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: controller.enqueueCronTab,
    UpdateFunc: func(old, new interface{}) {
      oldObj := old.(*crdv1beta1.CronTab)
      newObj := new.(*crdv1beta1.CronTab)
      // 如果资源版本相同则不处理
      if oldObj.ResourceVersion == newObj.ResourceVersion {
        return
      }
      controller.enqueueCronTab(new)
    },
    DeleteFunc: controller.enqueueCronTabForDelete,
  })
  return controller
}

func (c *Controller) Run(threadiness int, stopCh <-chan struct{}) error {
  defer runtime.HandleCrash()
  defer c.workqueue.ShutDown()

  // 记录开始日志
  klog.Info("Starting CronTab control loop")
  klog.Info("Waiting for informer caches to sync")
  if ok := cache.WaitForCacheSync(stopCh, c.informer.Informer().HasSynced); !ok {
    return fmt.Errorf("failed to wait for caches to sync")
  }

  klog.Info("Starting workers")
  for i := 0; i < threadiness; i++ {
    go wait.Until(c.runWorker, time.Second, stopCh)
  }

  klog.Info("Started workers")
  <-stopCh
  klog.Info("Shutting down workers")
  return nil
}

// runWorker 是一个不断运行的方法，并且一直会调用 c.processNextWorkItem 从workqueue读取和读取消息
func (c *Controller) runWorker() {
  for c.processNextWorkItem() {
  }
}

// 从workqueue读取和读取消息
func (c *Controller) processNextWorkItem() bool {
```

```
  obj, shutdown := c.workqueue.Get()
  if shutdown {
    return false
  }
  err := func(obj interface{}) error {
    defer c.workqueue.Done(obj)
    var key string
    var ok bool
    if key, ok = obj.(string); !ok {
      c.workqueue.Forget(obj)
      runtime.HandleError(fmt.Errorf("expected string in workqueue but got %#v", obj))
      return nil
    }
    if err := c.syncHandler(key); err != nil {
      return fmt.Errorf("error syncing '%s': %s", key, err.Error())
    }
    c.workqueue.Forget(obj)
    klog.Infof("Successfully synced '%s'", key)
    return nil
  }(obj)

  if err != nil {
    runtime.HandleError(err)
    return true
  }
  return true
}

// 尝试从 Informer 维护的缓存中拿到了它所对应的 CronTab 对象
func (c *Controller) syncHandler(key string) error {
  namespace, name, err := cache.SplitMetaNamespaceKey(key)
  if err != nil {
    runtime.HandleError(fmt.Errorf("invalid resource key: %s", key))
    return nil
  }

  crontab, err := c.informer.Lister().CronTabs(namespace).Get(name)

  //从缓存中拿不到这个对象,那就意味着这个 CronTab 对象的 Key 是通过前面的“删除”事件添加进工作队列的。
  if err != nil {
    if errors.IsNotFound(err) {
      // 对应的 crontab 对象已经被删除了
      klog.Warningf("[CronTabCRD] %s/%s does not exist in local cache, will delete it from CronTab ...",
        namespace, name)
      klog.Infof("[CronTabCRD] deleting crontab: %s/%s ...", namespace, name)
      return nil
    }
    runtime.HandleError(fmt.Errorf("failed to get crontab by: %s/%s", namespace, name))
    return err
  }
  klog.Infof("[CronTabCRD] try to process crontab: %#v ...", crontab)
  return nil
}

func (c *Controller) enqueueCronTab(obj interface{}) {
  var key string
  var err error
  if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
    runtime.HandleError(err)
    return
  }
  c.workqueue.AddRateLimited(key)
}

func (c *Controller) enqueueCronTabForDelete(obj interface{}) {
  var key string
  var err error
  key, err = cache.DeletionHandlingMetaNamespaceKeyFunc(obj)
  if err != nil {
    runtime.HandleError(err)
    return
  }
  c.workqueue.AddRateLimited(key)
}
```

我们这里自定义的控制器只封装了一个 Informer 和一个限速队列，我们当然也可以在里面添加一个用于访问本地缓存的 Indexer，但实际上 Informer 中已经包含了 Lister，对于 List 和 Get 操作都会去通过 Indexer 从本地缓存中获取数据，所以只用一个 Informer 也是完全可行的。

同样在 Informer 中注册了3个事件处理器，将监听的事件获取到后送入 workqueue 队列，然后通过控制器的控制循环不断从队列中消费数据，根据获取的 key 来获取数据判断对象是需要删除还是需要进行其他业务处理，这里我们同样也只是打印出了对应的操作日志，对于实际的项目则进行相应的业务逻辑处理即可。

到这里一个完整的自定义 API 对象和它所对应的自定义控制器就编写完毕了。

测试

接下来我们可以直接编译代码：

```
$ go build -ldflags "-s -w" -v -o crontab-controller .
```

编译完成后，会生成 `crontab-controller` 的二进制文件，由于我们代码中支持 inCluster 和 kubeconfig 两种模式，所以我们可以在 Kubernetes 集群中或者在本地（前提是可以访问 Kubernetes 集群）都可以运行该控制器。

比如我们在本地运行，由于 `~/.kube/config` 配置了访问 Kubernetes 集群的文件，所以我们直接运行该控制器即可：

```
$ ./crontab-controller
I1010 15:18:06.402954   82070 controller.go:31] Setting up crontab event handlers
I1010 15:18:06.403365   82070 controller.go:57] Starting CronTab control loop
I1010 15:18:06.403377   82070 controller.go:58] Waiting for informer caches to sync
I1010 15:18:06.605476   82070 controller.go:63] Starting workers
I1010 15:18:06.605582   82070 controller.go:68] Started workers
```

现在我们来创建一个 CronTab 资源对象：

```
# crontab-demo.yaml
apiVersion: stable.example.com/v1beta1
kind: CronTab
metadata:
  name: crontab-demo
  namespace: default
spec:
  image: "nginx:1.7.9"
  cronSpec: "* * * * */5"
  replicas: 2
```

直接创建上面的对象，注意观察控制器的日志：

```
$ kubectl apply -f crontab-demo.yaml
crontab.stable.example.com/crontab-demo created
$ kubectl get crontab
NAME           AGE
crontab-demo   70s
```

在上面的资源对象创建成功后在控制器的日志中则会出现如下所示的信息：

```
......
I1010 15:18:06.605476   82070 controller.go:63] Starting workers
I1010 15:18:06.605582   82070 controller.go:68] Started workers
I1010 15:19:55.115241   82070 controller.go:132] [CronTabCRD] try to process crontab: &v1beta1.CronTab{TypeMeta:v1.TypeMeta{Kind:"", A
I1010 15:19:55.116099   82070 controller.go:99] Successfully synced 'default/crontab-demo'
```

可以看到，我们上面创建 crontab-demo.yaml 的操作，触发了 EventHandler 的添加事件，从而被放进了工作队列。然后控制器的控制循环从队列里拿到这个对象，并且打印出了正在处理这个 CronTab 对象的日志信息。

同样我们删除这个资源的时候，也会有对应的提示：

```
$ kubectl delete crontab crontab-demo
crontab.stable.example.com "crontab-demo" deleted
```

当删除成功后对应的控制器出现了如下所示的日志信息：

```
W1010 15:23:11.738930   82070 controller.go:124] [CronTabCRD] default/crontab-demo does not exist in local cache, will delete it from
I1010 15:23:11.738952   82070 controller.go:126] [CronTabCRD] deleting crontab: default/crontab-demo ...
I1010 15:23:11.738961   82070 controller.go:99] Successfully synced 'default/crontab-demo'
```

这就是开发自定义 CRD 控制器的基本流程，当然我们还可以在事件处理的业务逻辑中去记录一些 Events 信息，这样我们就可以通过 Event 去了解我们资源的状态了。