25 | 深入解析声明式API（二）：编写自定义控制器

25 | 深入解析声明式API（二）：编写自定义控制器

张磊 2018-10-19

17:50

讲述：张磊 大小：8.17M

你好，我是张磊。今天我和你分享的主题是：深入解析声明式 API 之编写自定义控制器。

在上一篇文章中，我和你详细分享了 Kubernetes 中声明式 API 的实现原理，并且通过一个添加 Network 对象的实例，为你讲述了在 Kubernetes 里添加 API 资源的过程。

在今天的这篇文章中，我就继续和你一起完成剩下一半的工作，即：为 Network 这个自定义 API 对象编写一个自定义控制器（Custom Controller）。

正如我在上一篇文章结尾处提到的，“声明式 API”并不像“命令式 API”那样有着明显的执行逻辑。这就使得**基于声明式 API 的业务功能实现，往往需要通过控**

制器模式来“监视”API 对象的变化（比如，创建或者删除 Network），然后以此来决定实际要执行的具体工作。

接下来，我就和你一起通过编写代码来实现这个过程。这个项目和上一篇文章里的代码是同一个项目，你可以从这个 GitHub 库里找到它们。我在代码里还加上了丰富的注释，你可以随时参考。

总得来说，编写自定义控制器代码的过程包括：编写 main 函数、编写自定义控制器的定义，以及编写控制器里的业务逻辑三个部分。

首先，我们来编写这 * 个自定义控制器的 main 函数。

main 函数的主要工作就是，定义并初始化一个自定义控制器（Custom Controller），然后启动它。这部分代码的主要内容如下所示：

```
func main() {
...
cfg, err := clientcmd.BuildConfigFromFlags(masterURL, kubeconfig)
...
kubeClient, err := kubernetes.NewForConfig(cfg)
...
networkClient, err := clientset.NewForConfig(cfg)
...
networkInformerFactory :=
informers.NewSharedInformerFactory(networkClient, ...)
controller := NewController(kubeClient, networkClient,
networkInformerFactory.Samplecrd().V1().Networks())
go networkInformerFactory.Start(stopCh)
if err = controller.Run(2, stopCh); err != nil {
glog.Fatalf("Error running controller: %s", err.Error())
}
}
```

□复制代码

可以看到，这个 main 函数主要通过三步完成了初始化并启动一个自定义控制器的工作。

第一步：main 函数根据我提供的 Master 配置（APIServer 的地址端口和 kubeconfig 的路径），创建一个 Kubernetes 的 client（kubeClient）和

Network 对象的 client（networkClient）。

但是，如果我没有提供 Master 配置呢？

这时，main 函数会直接使用一种名叫**InClusterConfig**的方式来创建这个 client。这个方式，会假设你的自定义控制器是以 Pod 的方式运行在 Kubernetes 集群里的。

而我在第 15 篇文章《深入解析 Pod 对象（二）：使用进阶》中曾经提到过，Kubernetes 里所有的 Pod 都会以 Volume 的方式自动挂载 Kubernetes 的默认 ServiceAccount。所以，这个控制器就会直接使用默认 ServiceAccount 数据卷里的授权信息，来访问 APIServer。

第二步：main 函数为 Network 对象创建一个叫作 InformerFactory（即：networkInformerFactory）的工厂，并使用它生成一个 Network 对象的 Informer，传递给控制器。

第三步：main 函数启动上述的 Informer，然后执行 controller.Run，启动自定义控制器。

至此，main 函数就结束了。

看到这，你可能会感到非常困惑：编写自定义控制器的过程难道就这么简单吗？这个 Informer 又是个什么东西呢？

别着急。

接下来，我就为你**详细解释一下这个自定义控制器的工作原理。**

在 Kubernetes 项目中，一个自定义控制器的工作原理，可以用下面这样一幅流程图来表示（在后面的叙述中，我会用“示意图”来指代它）：

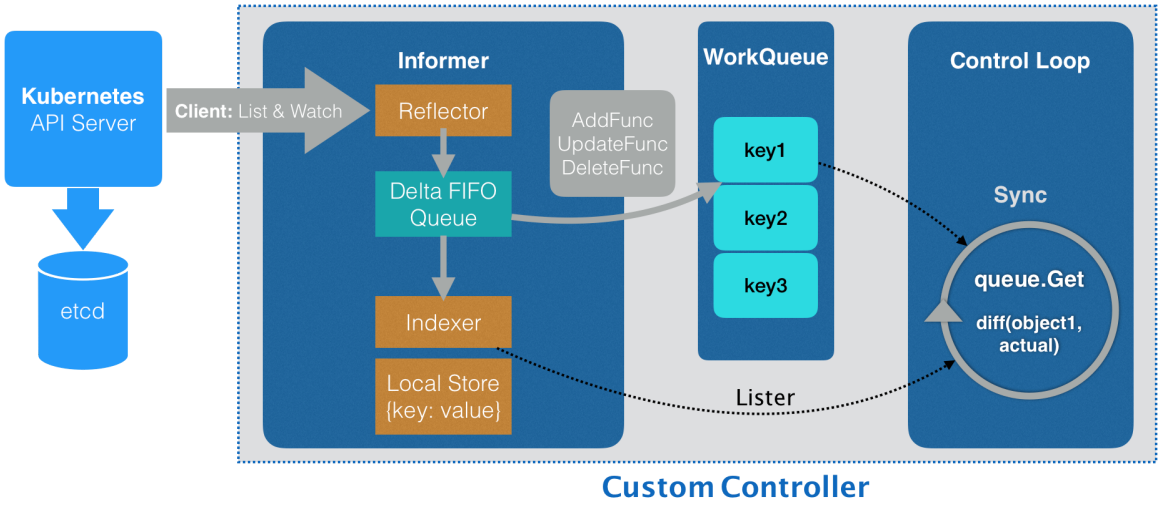

图 1 自定义控制器的工作流程示意图

我们先从这幅示意图的最左边看起。

这个控制器要做的第一件事，是从 Kubernetes 的 APIServer 里获取它所关心的对象，也就是我定义的 Network 对象。

这个操作，依靠的是一个叫作 Informer（可以翻译为：通知器）的代码库完成的。Informer 与 API 对象是一一对应的，所以我传递给自定义控制器的，正是一个 Network 对象的 Informer（Network Informer）。

不知你是否已经注意到，我在创建这个 Informer 工厂的时候，需要给它传递一个 networkClient。

事实上，Network Informer 正是使用这个 networkClient，跟 APIServer 建立了连接。不过，真正负责维护这个连接的，则是 Informer 所使用的 Reflector 包。

更具体地说，Reflector 使用的是一种叫作**ListAndWatch**的方法，来“获取”并“监听”这些 Network 对象实例的变化。

在 ListAndWatch 机制下，一旦 APIServer 端有新的 Network 实例被创建、删除或者更新，Reflector 都会收到“事件通知”。这时，该事件及它对应的 API 对象这个组合，就被称为增量（Delta），它会被放进一个 Delta FIFO Queue（即：增量先进先出队列）中。

而另一方面，Informe 会不断地从这个 Delta FIFO Queue 里读取（Pop）增量。每拿到一个增量，Informer 就会判断这个增量里的事件类型，然后创建或者更新本地对象的缓存。这个缓存，在 Kubernetes 里一般被叫作 Store。

比如，如果事件类型是 Added（添加对象），那么 Informer 就会通过一个叫作 Indexer 的库把这个增量里的 API 对象保存在本地缓存中，并为它创建索引。相反地，如果增量的事件类型是 Deleted（删除对象），那么 Informer 就会从本地缓存中删除这个对象。

这个**同步本地缓存的工作，是 Informer 的第一个职责，也是它最重要的职责。**

而**Informer 的第二个职责，则是根据这些事件的类型，触发事先注册好的 ResourceEventHandler**。这些 Handler，需要在创建控制器的时候注册给它对应的 Informer。

接下来，我们就来编写这个控制器的定义，它的主要内容如下所示：

```
func NewController(
kubeclientset kubernetes.Interface,
networkclientset clientset.Interface,
networkInformer informers.NetworkInformer) *Controller {
...
controller := &Controller{
kubeclientset: kubeclientset,
networkclientset: networkclientset,
networksLister: networkInformer.Lister(),
networksSynced: networkInformer.Informer().HasSynced,
workqueue: workqueue.NewNamedRateLimitingQueue(..., "Networks"),
...
}
networkInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
AddFunc: controller.enqueueNetwork,
UpdateFunc: func(old, new interface{}) {
oldNetwork := old.(*samplecrdv1.Network)
newNetwork := new.(*samplecrdv1.Network)
if oldNetwork.ResourceVersion == newNetwork.ResourceVersion {
return
}
controller.enqueueNetwork(new)
},
```

```
    DeleteFunc: controller.enqueueNetworkForDelete,
    return controller
    }
```

复制代码

我前面在 main 函数里创建了两个 client（kubeclientset 和 networkclientset），然后在这段代码里，使用这两个 client 和前面创建的 Informer，初始化了自定义控制器。

值得注意的是，在这个自定义控制器里，我还设置了一个工作队列（work queue），它正是处于示意图中间位置的 WorkQueue。这个工作队列的作用是，负责同步 Informer 和控制循环之间的数据。

> *实际上，Kubernetes 项目为我们提供了很多个工作队列的实现，你可以根据需要选择合适的库直接使用。*

然后，我为 networkInformer 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。

需要注意的是，实际入队的并不是 API 对象本身，而是它们的 Key，即：该 API 对象的<namespace>/<name>。

而我们后面即将编写的控制循环，则会不断地从这个工作队列里拿到这些 Key，然后开始执行真正的控制逻辑。

综合上面的讲述，你现在应该就能明白，**所谓 Informer，其实就是一个带有本地缓存和索引机制的、可以注册 EventHandler 的 client。**它是自定义控制器跟 APIServer 进行数据同步的重要组件。

更具体地说，Informer 通过一种叫作 ListAndWatch 的方法，把 APIServer 中的 API 对象缓存在了本地，并负责更新和维护这个缓存。

其中，ListAndWatch 方法的含义是：首先，通过 APIServer 的 LIST API“获取”所有最新版本的 API 对象；然后，再通过 WATCH API 来“监听”所有这些 API 对象的变化。

而通过监听到的事件变化，Informer 就可以实时地更新本地缓存，并且调用这些事件对应的 EventHandler 了。

此外，在这个过程中，每经过 resyncPeriod 指定的时间，Informer 维护的本地缓存，都会使用最近一次 LIST 返回的结果强制更新一次，从而保证缓存的有效

性。在 Kubernetes 中，这个缓存强制更新的操作就叫作：resync。

需要注意的是，这个定时 resync 操作，也会触发 Informer 注册的“更新”事件。但此时，这个“更新”事件对应的 Network 对象实际上并没有发生变化，即：新、旧两个 Network 对象的 ResourceVersion 是一样的。在这种情况下，Informer 就不需要对这个更新事件再做进一步的处理了。

这也是为什么我在上面的 UpdateFunc 方法里，先判断了一下新、旧两个 Network 对象的版本（ResourceVersion）是否发生了变化，然后才开始进行的入队操作。

以上，就是 Kubernetes 中的 Informer 库的工作原理了。

接下来，我们就来到了示意图中最后面的控制循环（Control Loop）部分，也正是我在 main 函数最后调用 controller.Run() 启动的“控制循环”。它的主要内容如下所示：

```
func (c *Controller) Run(threadiness int, stopCh <-chan struct{}) error {
...
if ok := cache.WaitForCacheSync(stopCh, c.networksSynced); !ok {
return fmt.Errorf("failed to wait for caches to sync")
}
...
for i := 0; i < threadiness; i++ {
go wait.Until(c.runWorker, time.Second, stopCh)
}
...
return nil
}
```

复制代码

可以看到，启动控制循环的逻辑非常简单：

首先，等待 Informer 完成一次本地缓存的数据同步操作；
然后，直接通过 goroutine 启动一个（或者并发启动多个）“无限循环”的任务。

而这个“无限循环”任务的每一个循环周期，执行的正是我们真正关心的业务逻辑。

所以接下来，我们就来编写这个自定义控制器的业务逻辑，它的主要内容如下所示：

```
func (c *Controller) runWorker() {
for c.processNextWorkItem() {
}
}
func (c *Controller) processNextWorkItem() bool {
obj, shutdown := c.workqueue.Get()
...
err := func(obj interface{}) error {
...
if err := c.syncHandler(key); err != nil {
return fmt.Errorf("error syncing '%s': %s", key, err.Error())
}
c.workqueue.Forget(obj)
...
return nil
}(obj)
...
return true
}
func (c *Controller) syncHandler(key string) error {
namespace, name, err := cache.SplitMetaNamespaceKey(key)
...
network, err := c.networksLister.Networks(namespace).Get(name)
if err != nil {
if errors.IsNotFound(err) {
glog.Warningf("Network does not exist in local cache: %s/%s, will
delete it from Neutron ...",
namespace, name)
glog.Warningf("Network: %s/%s does not exist in local cache, will
delete it from Neutron ...",
namespace, name)
```

```
    // FIX ME: call Neutron API to delete this network by name.
    //
    // neutron.Delete(namespace, name)
    return nil
    }
    ...
    return err
    }
    glog.Infof("[Neutron] Try to process network: %#v ...", network)
    // FIX ME: Do diff().
    //
    // actualNetwork, exists := neutron.Get(namespace, name)
    //
    // if !exists {
    // neutron.Create(namespace, name)
    // } else if !reflect.DeepEqual(actualNetwork, network) {
    // neutron.Update(namespace, name)
    // }
    return nil
    }
```

复制代码

可以看到，在这个执行周期里（processNextWorkItem），我们**首先**从工作队列里出队（workqueue.Get）了一个成员，也就是一个 Key（Network 对象的：namespace/name）。

然后，在 syncHandler 方法中，我使用这个 Key，尝试从 Informer 维护的缓存中拿到了它所对应的 Network 对象。

可以看到，在这里，我使用了 networksLister 来尝试获取这个 Key 对应的 Network 对象。这个操作，其实就是在访问本地缓存的索引。实际上，在 Kubernetes 的源码中，你会经常看到控制器从各种 Lister 里获取对象，比如：podLister、nodeLister 等等，它们使用的都是 Informer 和缓存机制。

而如果控制循环从缓存中拿不到这个对象（即：networkLister 返回了 IsNotFound 错误），那就意味着这个 Network 对象的 Key 是通过前面的“删

除”事件添加进工作队列的。所以，尽管队列里有这个 Key，但是对应的 Network 对象已经被删除了。

这时候，我就需要调用 Neutron 的 API，把这个 Key 对应的 Neutron 网络从真实的集群里删除掉。

而如果能够获取到对应的 Network 对象，我就可以执行控制器模式里的对比“期望状态”和“实际状态”的逻辑了。

其中，自定义控制器“千辛万苦”拿到的这个 Network 对象，**正是 APIServer 里保存的“期望状态”**，即：用户通过 YAML 文件提交到 APIServer 里的信息。当然，在我们的例子里，它已经被 Informer 缓存在了本地。

那么，“实际状态”又从哪里来呢?

当然是来自于实际的集群了。

所以，我们的控制循环需要通过 Neutron API 来查询实际的网络情况。

比如，我可以先通过 Neutron 来查询这个 Network 对象对应的真实网络是否存在。

> 如果不存在，这就是一个典型的“期望状态”与“实际状态”不一致的情形。这时，我就需要使用这个 Network 对象里的信息（比如：CIDR 和 Gateway），调用 Neutron API 来创建真实的网络。
> 如果存在，那么，我就要读取这个真实网络的信息，判断它是否跟 Network 对象里的信息一致，从而决定我是否要通过 Neutron 来更新这个已经存在的真实网络。

这样，我就通过对比“期望状态”和“实际状态”的差异，完成了一次调协（Reconcile）的过程。

至此，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。

> *备注：与 Neutron 相关的业务代码并不是本篇文章的重点，所以我仅仅通过注释里的伪代码为你表述了这部分内容。如果你对这些代码感兴趣的话，可以自行完成。最简单的情况，你可以自己编写一个 Neutron Mock，然后输出对应的操作日志。*

接下来，我们就一起来把这个项目运行起来，查看一下它的工作情况。

你可以自己编译这个项目，也可以直接使用我编译好的二进制文件（samplecrd-controller）。编译并启动这个项目的具体流程如下所示：

```
# Clone repo

$ git clone https://github.com/resouer/k8s-controller-custom-resource$ cd k8s-controller-custom-resource

### Skip this part if you don't want to build

# Install dependency

$ go get github.com/tools/godep

$ godep restore

# Build

$ go build -o samplecrd-controller .

$ ./samplecrd-controller -kubeconfig=$HOME/.kube/config -alsologtostderr=true

I0915 12:50:29.051349 27159 controller.go:84] Setting up event handlers

I0915 12:50:29.051615 27159 controller.go:113] Starting Network control loop

I0915 12:50:29.051630 27159 controller.go:116] Waiting for informer caches to sync

E0915 12:50:29.066745 27159 reflector.go:134] github.com/resouer/k8s-controller-custom-resource/pkg/client/informers/externalversions/factory.go:117: Failed to list *v1.Network: the server could not find the requested resource (get networks.samplecrd.k8s.io)

...
```

□复制代码

你可以看到，自定义控制器被启动后，一开始会报错。

这是因为，此时 Network 对象的 CRD 还没有被创建出来，所以 Informer 去 APIServer 里“获取”（List）Network 对象时，并不能找到 Network 这个 API 资源类型的定义，即：

```
Failed to list *v1.Network: the server could not find the requested resource (get networks.samplecrd.k8s.io)
```

□复制代码

所以，接下来我就需要创建 Network 对象的 CRD，这个操作在上一篇文章里已经介绍过了。

在另一个 shell 窗口里执行：

```
$ kubectl apply -f crd/network.yaml
```

□复制代码

这时候，你就会看到控制器的日志恢复了正常，控制循环启动成功：

```
...
I0915 12:50:29.051630 27159 controller.go:116] Waiting for informer caches to sync
...
I0915 12:52:54.346854 25245 controller.go:121] Starting workers
I0915 12:52:54.346914 25245 controller.go:127] Started workers
```

□复制代码

接下来，我就可以进行 Network 对象的增删改查操作了。

首先，创建一个 Network 对象：

```
$ cat example/example-network.yaml
apiVersion: samplecrd.k8s.io/v1
kind: Network
metadata:
name: example-network
spec:
cidr: "192.168.0.0/16"
gateway: "192.168.0.1"
$ kubectl apply -f example/example-network.yaml
network.samplecrd.k8s.io/example-network created
```

□复制代码

这时候，查看一下控制器的输出：

```
...
I0915 12:50:29.051349 27159 controller.go:84] Setting up event handlers
I0915 12:50:29.051615 27159 controller.go:113] Starting Network control loop
I0915 12:50:29.051630 27159 controller.go:116] Waiting for informer caches to sync
...
```

```
I0915 12:52:54.346854 25245 controller.go:121] Starting workers
I0915 12:52:54.346914 25245 controller.go:127] Started workers
I0915 12:53:18.064409 25245 controller.go:229] [Neutron] Try to process network: &v1.Network{TypeMeta:v1.TypeMeta{Kind:"", APIVersion:""}, ObjectMeta:v1.ObjectMeta{Name:"example-network", GenerateName:"", Namespace:"default", ... ResourceVersion:"479015", ... Spec:v1.NetworkSpec{Cidr:"192.168.0.0/16", Gateway:"192.168.0.1"}}
...
I0915 12:53:18.064650 25245 controller.go:183] Successfully synced 'default/example-network'
...
```

□复制代码

可以看到，我们上面创建 example-network 的操作，触发了 EventHandler 的“添加”事件，从而被放进了工作队列。

紧接着，控制循环就从队列里拿到了这个对象，并且打印出了正在“处理”这个 Network 对象的日志。

可以看到，这个 Network 的 ResourceVersion，也就是 API 对象的版本号，是 479015，而它的 Spec 字段的内容，跟我提交的 YAML 文件一摸一样，比如，它的 CIDR 网段是：192.168.0.0/16。

这时候，我来修改一下这个 YAML 文件的内容，如下所示：

```
$ cat example/example-network.yaml
apiVersion: samplecrd.k8s.io/v1
kind: Network
metadata:
name: example-network
spec:
cidr: "192.168.1.0/16"
gateway: "192.168.1.1"
```

□复制代码

可以看到，我把这个 YAML 文件里的 CIDR 和 Gateway 字段的修改成了 192.168.1.0/16 网段。

然后，我们执行了 kubectl apply 命令来提交这次更新，如下所示：

```
$ kubectl apply -f example/example-network.yaml
```

```
network.samplecrd.k8s.io/example-network configured
```

□复制代码

这时候，我们就可以观察一下控制器的输出：

```
...
I0915 12:53:51.126029 25245 controller.go:229] [Neutron] Try to process network: &v1.Network{TypeMeta:v1.TypeMeta{Kind:"", APIVersion:""}, ObjectMeta:v1.ObjectMeta{Name:"example-network", GenerateName:"", Namespace:"default", ... ResourceVersion:"479062", ... Spec:v1.NetworkSpec{Cidr:"192.168.1.0/16", Gateway:"192.168.1.1"}}
...
I0915 12:53:51.126348 25245 controller.go:183] Successfully synced 'default/example-network'
```

□复制代码

可以看到，这一次，Informer 注册的“更新”事件被触发，更新后的 Network 对象的 Key 被添加到了工作队列之中。

所以，接下来控制循环从工作队列里拿到的 Network 对象，与前一个对象是不同的：它的 ResourceVersion 的值变成了 479062；而 Spec 里的字段，则变成了 192.168.1.0/16 网段。

最后，我再把这个对象删除掉：

```
$ kubectl delete -f example/example-network.yaml
```

□复制代码

这一次，在控制器的输出里，我们就可以看到，Informer 注册的“删除”事件被触发，并且控制循环“调用”Neutron API“删除”了真实环境里的网络。这个输出如下所示：

```
W0915 12:54:09.738464 25245 controller.go:212] Network: default/example-network does not exist in local cache, will delete it from Neutron ...
I0915 12:54:09.738832 25245 controller.go:215] [Neutron] Deleting network: default/example-network ...
I0915 12:54:09.738854 25245 controller.go:183] Successfully synced 'default/example-network'
```

□复制代码

以上，就是编写和使用自定义控制器的全部流程了。

实际上，这套流程不仅可以用在自定义 API 资源上，也完全可以用在 Kubernetes 原生的默认 API 对象上。

比如，我们在 main 函数里，除了创建一个 Network Informer 外，还可以初始化一个 Kubernetes 默认 API 对象的 Informer 工厂，比如 Deployment 对象的 Informer。这个具体做法如下所示：

```
func main() {
...
kubeInformerFactory :=
kubeinformers.NewSharedInformerFactory(kubeClient, time.Second*30)
controller := NewController(kubeClient, exampleClient,
kubeInformerFactory.Apps().V1().Deployments(),
networkInformerFactory.Samplecrd().V1().Networks())
go kubeInformerFactory.Start(stopCh)
...
}
```

复制代码

在这段代码中，我们**首先**使用 Kubernetes 的 client（kubeClient）创建了一个工厂；

然后，我用跟 Network 类似的处理方法，生成了一个 Deployment Informer；

接着，我把 Deployment Informer 传递给了自定义控制器；当然，我也要调用 Start 方法来启动这个 Deployment Informer。

而有了这个 Deployment Informer 后，这个控制器也就持有了所有 Deployment 对象的信息。接下来，它既可以通过 deploymentInformer.Lister() 来获取 Etcd 里的所有 Deployment 对象，也可以为这个 Deployment Informer 注册具体的 Handler 来。

更重要的是，**这就使得在这个自定义控制器里面，我可以通过对自定义 API 对象和默认 API 对象进行协同，从而实现更加复杂的编排功能。**

比如：用户每创建一个新的 Deployment，这个自定义控制器，就可以为它创建一个对应的 Network 供它使用。

这些对 Kubernetes API 编程范式的更高级应用，我就留给你在实际的场景中去探索和实践了。

总结

在今天这篇文章中，我为你剖析了 Kubernetes API 编程范式的具体原理，并编写了一个自定义控制器。

这其中，有如下几个概念和机制，是你一定要理解清楚的：

所谓的 Informer，就是一个自带缓存和索引机制，可以触发 Handler 的客户端库。这个本地缓存在 Kubernetes 中一般被称为 Store，索引一般被称为 Index。

Informer 使用了 Reflector 包，它是一个可以通过 ListAndWatch 机制获取并监视 API 对象变化的客户端封装。

Reflector 和 Informer 之间，用到了一个“增量先进先出队列”进行协同。而 Informer 与你要编写的控制循环之间，则使用了一个工作队列来进行协同。

在实际应用中，除了控制循环之外的所有代码，实际上都是 Kubernetes 为你自动生成的，即：pkg/client/{informers, listers, clientset}里的内容。

而这些自动生成的代码，就为我们提供了一个可靠而高效地获取 API 对象“期望状态”的编程库。

所以，接下来，作为开发者，你就只需要关注如何拿到“实际状态”，然后如何拿它去跟“期望状态”做对比，从而决定接下来要做的业务逻辑即可。

以上内容，就是 Kubernetes API 编程范式的核心思想。

思考题

请思考一下，为什么 Informer 和你编写的控制循环之间，一定要使用一个工作队列来进行协作呢？