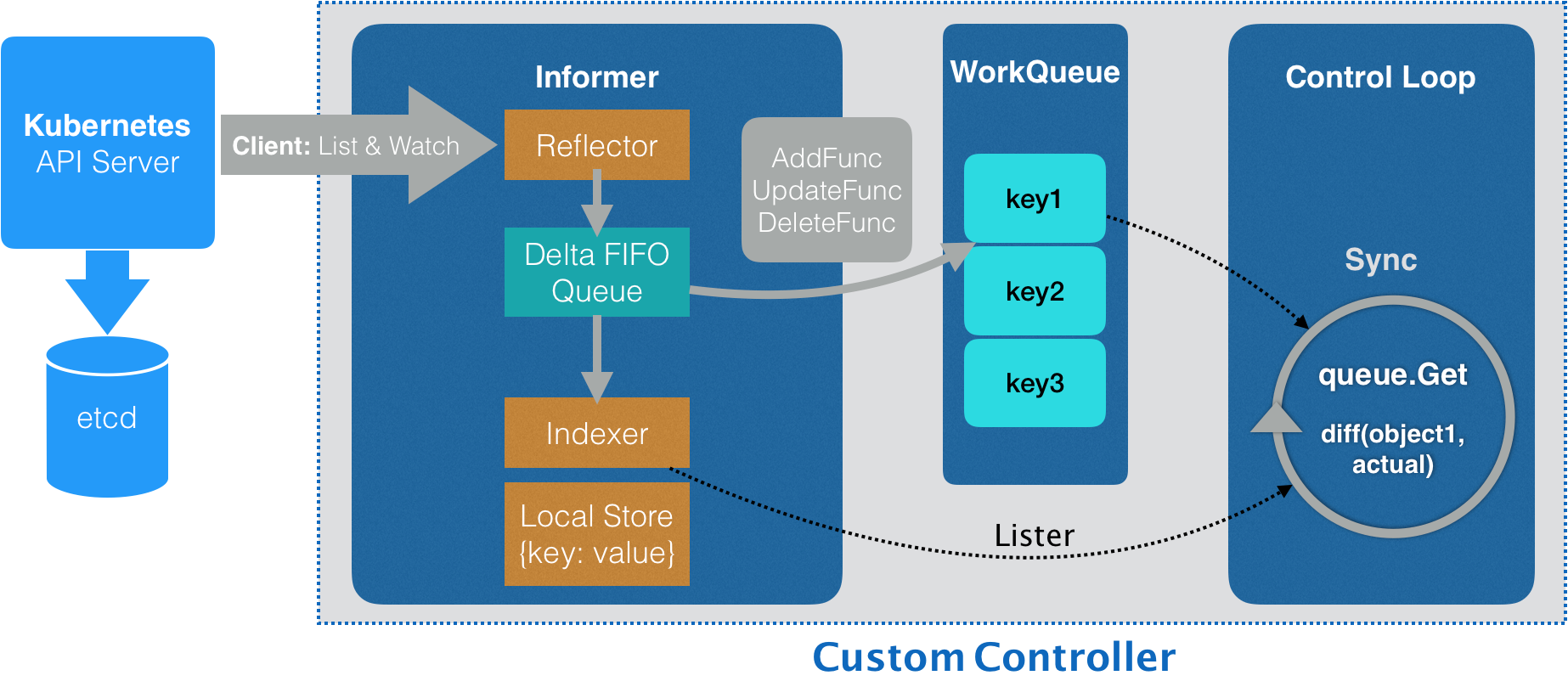
## Custom Controller



一．一定要理解清楚的概念和机制

1. 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。
2. 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。
3. 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。
4. 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。

1. 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。
2. 控制器开发的主要三步

假设定义了一个CRD:

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

  name: networks.samplecrd.k8s.io

spec:

  group: samplecrd.k8s.io

  version: v1

  names:

    kind: Network

    plural: networks

  scope: Namespaced

CR为：

apiVersion: samplecrd.k8s.io/v1

kind: Network

metadata:

  name: example-network

spec:

  cidr: "192.168.0.0/16"

  gateway: "192.168.0.1"

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())

}

}

第一步：main 函数根据提供的 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）。

如果没有提供 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。

1. informer原理

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）。

Network Informer 正是使用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。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。

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，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。

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。

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。比如：

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

}

前面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。

在这个自定义控制器里，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。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。

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。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>。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，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。

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。

1. controller.Run() 启动的“控制循环”

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

}

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。

自定义控制器业务逻辑里，则回去拿到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，它已经被 Informer 缓存在了本地。

实际状态”又从哪里来呢？当然是来自于实际的集群了。

比如，上面例子中，控制循环需要通过 Neutron API 来查询实际的网络情况（了解即可）。最终通过对比“期望状态”和“实际状态”的差异，完成了一次调协（Reconcile）的过程。

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

}