CRD自定义资源&实现controller调度器

liangdong@smzdm.com

[创建CRD 1](#_Toc535248291)

[使用CRD 3](#_Toc535248292)

[生成CRD类库 4](#_Toc535248293)

[下载code generation工具 6](#_Toc535248294)

[创建pkg/apis目录结构 6](#_Toc535248295)

[创建group registers 7](#_Toc535248296)

[创建doc.go 7](#_Toc535248297)

[创建types.go 8](#_Toc535248298)

[创建register.go 9](#_Toc535248299)

[生成代码 10](#_Toc535248300)

[访问CRD 13](#_Toc535248301)

K8S中一切都是resource，比如Deployment，Service等等。

我们可以基于CRD（CustomResourceDefinitions）功能新增resource，比如我想自定义一种Deployment资源，提供不同的部署策略。

我们知道resource可以通过k8s的RESTFUL API进行CURD操作，对于CRD创建的resource也是一样的。

CRD仅仅是定义一种resource，我们还需要实现controller，类似于deployment controller等等，监听对应资源的CURD事件，做出对应的处理，比如部署POD。

CRD官方文档：<https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/>

接下来，我们先创建一个CRD。

# 创建CRD

我打算创建一种CRD，就是部署nginx用的。 只需要配置replicas数量，然后我编写的controller就会拉起对应数量的nginx pod。

下面是CRD的定义：

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

 # name must match the spec fields below, and be in the form: <plural>.<group>

 name: nginxes.mycompany.com

spec:

 # group name to use for REST API: /apis/<group>/<version>

 group: mycompany.com

 # list of versions supported by this CustomResourceDefinition

 versions:

 - name: v1

 # Each version can be enabled/disabled by Served flag.

 served: true

 # One and only one version must be marked as the storage version.

 storage: true

 # either Namespaced or Cluster

 scope: Cluster

 names:

 # plural name to be used in the URL: /apis/<group>/<version>/<plural>

 plural: nginxes

 # singular name to be used as an alias on the CLI and for display

 singular: nginx

 # kind is normally the CamelCased singular type. Your resource manifests use this.

 kind: Nginx

 # shortNames allow shorter string to match your resource on the CLI

 shortNames:

 - ngx

kubectl apply -f 执行上述yaml即可。

首先我们要定义一组新的api group，这里叫做mycompany.com。

接着得定义api的版本，我起名叫做v1。

在group=mycompany.com下，我们有一种资源叫做Nginx，这就是Kind，我们经常用各种Kind，比如Deployment，CRD就是自定义了一种资源。

scope是指CRD自身的定义是否放在某个namespace下面，如果放在某个namespace下面，那么namespace被删除的时候CRD定义也就被删了，所以我放在cluster下面。

上面英文单词plural是英文复数的意思，singular是英文单数的意思，shortNames就是缩写（kubectl可以用短名称）。

# 使用CRD

接下来，我们可以用CRD创建一个resource object了。

下面的yaml指定部署一套nginx：

apiVersion: "mycompany.com/v1"

kind: Nginx

metadata:

 name: my-deployment

spec:

 replicas: 2

apiVersion指定了使用我们的CRD，kind是Nginx资源，metadata是k8s通用的资源描述元信息，spec部分每一种资源各不相同，我定义的这个CRD为了简化测试，只需要传一个replicas数量，controller会启动对应数量的Nginx。

我们把这个yaml提交一下，然后可以查看相关资源：



查看详细信息：



这就完成了object的创建，和我们熟悉的deployment,service等没有什么区别。

CRD和内建的资源一样，可以CURD，上述资源对应apiserver的REST url是：

/apis/mycompany.com/v1/nginxes/my-nginx。

如果创建my-nginx的时候指定namespace的话，那么URL将变成：

/apis/mycompany.com/v1/namespaces/xxx/nginxes/my-nginx。

（在v1.13之后支持创建CRD的时候配置validation，指定spec里可以放什么东西，不是很重要就不展开了）

# 生成CRD类库

接下来，我希望编写一个nginx\_controller，它监听apiserver中Nginx资源的变化，做出对应的部署动作。

实现controller需要用client-go库 。

官方提供了一个示例：<https://github.com/kubernetes/sample-controller> ，但是只能监听K8s内建的resource>

实现controller时的程序设计流程如下：

 <https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md>：



为了实现对CRD资源的监听与操作，可以使用code-generator项目来生成脚手架代码：<https://github.com/kubernetes/code-generator> 。

这篇文章，介绍了如何step by step的使用code-generator搭建骨架：<https://blog.openshift.com/kubernetes-deep-dive-code-generation-customresources/> 。

另外一篇文章，介绍了如何不用code-generator搭建骨架：

<https://thenewstack.io/extend-kubernetes-1-7-custom-resources/>

最终开发controller时，社区给出的设计思想：<https://github.com/kubernetes/community/blob/8decfe4/contributors/devel/controllers.md>

另外一篇controller开发思想：

<https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html>

## 下载code generation工具

先执行go get github.com/kubernetes/code-generator。

再创建目录: mkdir -p $GOPATH/src/k8s.io

最后mv $GOPATH/src/github.com/kubernetes/code-generator到$GOPATH/src/k8s.io

之所以这么麻烦，因为k8s.io被墙了，我们先从github下载一份镜像代码，再放到k8s.io目录下即可。

我们要用code generation来生成访问自定义CRD的client代码。

## 创建pkg/apis目录结构

我的示例代码在这里： <https://github.com/owenliang/k8s-client-go/tree/master/demo9>

我们要做的就是在自己的项目下，创建一个这样的目录结构：



目录pkg/apis/{controller起个名字就行}/{CRD的版本号}。

## 创建group registers

在pkg/apis/nginx\_contronller这一级，需要放一个register：

。

里面写上这样的东西：

**package** nginx\_controller

**const** (
 ***GroupName*** = **"mycompany.com"**)

把api group放上去。

## 创建doc.go

放doc.go，

里面写上这样的注释内容：

*// +k8s:deepcopy-gen=package
// +groupName=mycompany.com***package** v1

一会代码生成器会识别注释，扫描这个v1包下面的文件。

## 创建types.go

在这个文件里，我们要定义CRD的结构体：

**package** v1

**import** (
 meta\_v1 **"k8s.io/apimachinery/pkg/apis/meta/v1"**)

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

// 单个object, 例如：kubectl get nginxs.mycompany.com {name} -o yaml***type** Nginx **struct** {
 meta\_v1.TypeMeta **`json:",inline"`** *// Kind, ApiVersion* meta\_v1.ObjectMeta **`json:"metadata"`** *// metadata.name, metadata.namespace等...* Spec Spec **`json:"spec"`** Status Status **`json:"status, omitempty"`**}

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

// object列表, 例如：kubectl get nginexs.mycompany.com -o yaml***type** NginxList **struct** {
 meta\_v1.TypeMeta **`json:",inline"`** *// Kind总是List* meta\_v1.ListMeta **`json:"metadata"`** Items []Nginx **`json:"items"`**}

*// 自定义***type** Spec **struct** {
 Replicas int **`json:"replicas"`**}

*// 自定义***type** Status **struct** {
 Message string **`json:"message,omitempty"`**}

注意，我们定义的CRD叫做Nginx，那么就定义一个struct Nginx，里面能包含了任何Resource都有的元信息。Spec和Status可以自定义。我们需要在这个CRD上方写注释：

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

这样code generation会识别到注释，+genclient意味着生成访问Nginx资源的client代码；deepcopy-gen用来生成一种特殊的deepCopy方法，大家就不用关心了。

NginxList是复数形式的结构体，里面放了多个Nginx，当我们执行kubectl get ngx的时候，返回的就是一个NginxList列表，它的Kind是List。 XXX和XXXList总是要同时定义的，记得在List上方写生成deepCopy方法的注释：

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

对于其他内嵌的结构体，不用写注释，照做即可。

## 创建register.go

未来code generation会根据types.go和doc.go自动生成register.go，但是目前还没发布这个特性。

手动编写这个Register.go吧：

**package** v1

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

 "github.com/owenliang/k8s-client-go/demo9/pkg/apis/nginx\_controller"**)

*// SchemeGroupVersion is group version used to register these objects***var** SchemeGroupVersion = schema.GroupVersion{Group: nginx\_controller.***GroupName***, Version: **"v1"**}

*// Kind takes an unqualified kind and returns back a Group qualified GroupKind***func** Kind(kind string) schema.GroupKind {
 **return** SchemeGroupVersion.WithKind(kind).GroupKind()
}

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

**var** (
 SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
 AddToScheme = SchemeBuilder.AddToScheme
)

*// Adds the list of known types to Scheme.***func** addKnownTypes(scheme \*runtime.Scheme) error {
 scheme.AddKnownTypes(SchemeGroupVersion,
 &Nginx{},
 &NginxList{},
 )
 metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
 **return** nil
}

实现自己的CRD时候，只需要替换红色字体部分。

## 生成代码

确认你准备好了：



那么就可以生成CRD相关的代码了，在任意目录执行如下命令：

$GOPATH/src/k8s.io/code-generator/generate-groups.sh all github.com/owenliang/k8s-client-go/demo9/pkg/client github.com/owenliang/k8s-client-go/demo9/pkg/apis nginx\_controller:v1

一共4个参数，后面2个红色的参数，大家需要根据自己创建的pkg目录进行替换。 最后一个参数，是要扫描的group和version，这里就是nginx\_controller目录下的v1版本。



执行后，生成了zz\_generated.deepcopy.go文件，里面的代码就是给types.go里的各种类型支持了deepCopy深拷贝函数:

*// DeepCopyInto is an autogenerated deepcopy function, copying the receiver, writing into out. in must be non-nil.***func** (in \*Nginx) DeepCopyInto(out \*Nginx) {
 \*out = \*in
 out.TypeMeta = in.TypeMeta
 in.ObjectMeta.DeepCopyInto(&out.ObjectMeta)
 out.Spec = in.Spec
 out.Status = in.Status
 **return**}

*// DeepCopyInto is an autogenerated deepcopy function, copying the receiver, writing into out. in must be non-nil.***func** (in \*Status) DeepCopyInto(out \*Status) {
 \*out = \*in
 **return**}

但是发现，其实Spec和Status字段并没有深拷贝，还是浅拷贝，大家注意别踩坑即可，届时可以根据需要来完善这些函数，代码生成器只是搭了个通用的骨架。

Client目录也是生成的，里面包含了编写controller需要的各种组件，clientset是访问CRD的k8s客户端，informers之前的图片里提到过。

## 访问CRD

我们现在可以编写一个main.go：

**package** main

**import** (
 **"github.com/owenliang/k8s-client-go/common"
 "k8s.io/client-go/rest"
 "fmt"
 "github.com/owenliang/k8s-client-go/demo9/pkg/client/clientset/versioned"
 "k8s.io/apimachinery/pkg/apis/meta/v1"** crd\_v1 **"github.com/owenliang/k8s-client-go/demo9/pkg/apis/nginx\_controller/v1"**)

**func** main() {
 **var** (
 restConf \*rest.Config
 crdClientset \*versioned.Clientset
 nginx \*crd\_v1.Nginx
 err error
 )

 *// 读取admin.conf, 生成客户端基本配置* **if** restConf, err = common.GetRestConf(); err != nil {
 **goto FAIL** }

 *// 创建CRD的client* **if** crdClientset, err = versioned.NewForConfig(restConf); err != nil {
 **goto FAIL** }

 *// 获取CRD的nginx对象* **if** nginx, err = crdClientset.MycompanyV1().Nginxes(**"default"**).Get(**"my-nginx"**, v1.GetOptions{}); err != nil {
 **goto FAIL** }

 fmt.Println(nginx)

 **return

FAIL**:
 fmt.Println(err)
 **return**}

首先照常读取admin.conf为rest.Config，它包含了k8s apiserver访问地址和密钥。 然后调用生成的NewForConfig方法创建访问CRD的clientset。

接下来就和访问内置的K8S resource一样，通过clientset.{apiVersion}.{resource}就可以访问Get/Delete等方法了。

这里就是获取了一下my-nginx那个Nginx对象，打印了一下：

&{{ } {my-nginx /apis/mycompany.com/v1/nginxes/my-nginx 32f36eec-1553-11e9-849d-08002754aaa3 1267022 1 2019-01-11 11:44:29 +0800 CST <nil> <nil> map[] map[kubectl.kubernetes.io/last-applied-configuration:{"apiVersion":"mycompany.com/v1","kind":"Nginx","metadata":{"annotations":{},"name":"my-nginx"},"spec":{"replicas":2}}

至此，我们就准备好了controller开发的基本类库了，接下来就要研究如何实现controller部分了。

# 实现CRD controller

完整代码见：<https://github.com/owenliang/k8s-client-go/tree/master/demo10>

## 实现的效果

提交这样的yaml：



我的controller调度器，可以自动生成对应数量的POD，都运行着nginx：



更新replicas配置，controller会实时的调整POD数量。

运行中POD失联或者宕机，controller会补充POD数量，并且删除异常POD。

可以理解成一个简化的replicaset实现。

## 原理



回到这张图。

我们编程主要涉及这么几个概念：

1. informer：生成的代码，监听apiserver中特定资源变化，然后会存储到一个线程安全的local cache中，最后回调我们自己实现的event handler。
2. local cache：生成的代码，informer实时同步apiserver（也就是etcd）中的数据到内存中存储，可以有效降低apiserver的查询压力，但缺点就是实时性不好，本地会比远程的数据落后一点点但会最终与etcd一致，所以需要根据情况具体分析是走Local cache还是apiserver实时获取数据。
3. lister：生成的代码，提供了CURD操作访问local cache。
4. controller：一个逻辑概念，就是指调度某种资源的实现而已，需要我们自己开发。Controller做的事情主要包括：
	1. 实现event handler处理资源的CURD操作
	2. 在event handler，可以使用workqueue类库实现相同资源对象的连续event的去重，以及event处理异常后的失败重试，通常是建议使用的。
5. Workqueue：一个单独的类库，是可选使用的，但通常都会使用，原因上面说了。我们需要在实现event handler的时候把发生变化的资源标识放入workqueue，供下面的processor消费。
6. Clientset：默认clientset只能CRUD k8s提供的资源类型，比如deployments，daemonset等；生成的代码为我们自定义的资源（CRD）生成了单独的clientset，从而让我们使用结构化的代码CURD自定义资源。也就是说，想操作内建资源就用k8s自带的clientset，想操作CRD就用生成代码里的clientset。
7. Processor：我们实现的go协程，消费workqueue中的事件，workqueue提供了按资源标识的去重。

## 先修复lister代码

生成代码后，我在实践中遇到了一个报错，发现生成的lister代码在namespace为空的情况下实现有点问题，需要进行调整。

修改nginx.go中的Get方法：



即namespace为空的情况下，进行兼容，后面会提到背后的原因。

Lister的作用是访问local cache中的k/v数据，我们说过这个问题。

## Main.go

代码地址：<https://github.com/owenliang/k8s-client-go/blob/master/demo10/main.go>

先初始化日志，打印到屏幕上：

*// 日志参数*klog.InitFlags(nil)
flag.Set(**"logtostderr"**, **"1"**) *// 输出日志到stderr*flag.Parse()

再初始化内建资源的clientset和CRD资源的clientset，这里内建资源就是操作POD（我得维护POD数量啊），CRD资源就是Kind: Nginx：

*// 读取admin.conf, 生成客户端基本配置***if** restConf, err = common.GetRestConf(); err != nil {
 **goto FAIL**}

*// 创建CRD的client***if** crdClientset, err = versioned.NewForConfig(restConf); err != nil {
 **goto FAIL**}

*// 创建K8S内置的client***if** clientset, err = kubernetes.NewForConfig(restConf); err != nil {
 **goto FAIL**}

创建informer工厂（就这么个东西，不用关心内部是啥），生成监听POD资源的informer监听器，以及监听Nginx资源的informer监听器。我们说过，informer监听资源变化，并且存到Local cache里，最后还会回调我们的event handelr，这些工作都是podInformer和nginxInformer内部实现的。（注意到第2个参数的时间了吗，这是resync机制，k8s会定期的把local cache中的全量数据回调给event handler，给我们的代码定时修正调度的机会，这样我们的代码实现即便异常覆盖不全面，也有机会定时的修正）

*// 内建informer工厂*informerFactory = informers.NewSharedInformerFactory(clientset, time.***Second*** \* 120)
*// crd Informer工厂*crdInformerFactory = externalversions.NewSharedInformerFactory(crdClientset, time.***Second*** \* 120)

*// POD informer*podInformer = informerFactory.Core().V1().Pods()
*// nginx informer*nginxInformer = crdInformerFactory.Mycompany().V1().Nginxes()

最后，我们创建了controller，这是自定义的结构体，不是K8S生成的。做的事情就是把informer和clientset传进去备用：

*// 创建调度controller*nginxController = &controller.NginxController{Clientset: clientset, CrdClientset: crdClientset, PodInformer:podInformer, NginxInformer: nginxInformer}
nginxController.Start()

*// 等待***for** {
 time.Sleep(1 \* time.***Second***)
}

所以接下来看controller逻辑了。

## Nginx\_controller.go

代码地址：<https://github.com/owenliang/k8s-client-go/blob/master/demo10/controller/nginx_controller.go>

controller就是核心调度器了，目标就是监听Nginx和对应POD资源的变化，做出对应的动作，实现最终一致的调度。

结构体如下，没见过的就是workqueue两个。 我们可见，nginx和pod资源的东西都是两套，各自的Informer，各自的clientset，各自的workqueue。

**type** NginxController **struct** {
 Clientset \*kubernetes.Clientset
 CrdClientset \*versioned.Clientset
 PodInformer core\_v1.PodInformer
 NginxInformer v1.NginxInformer

 NginxWorkqueue workqueue.RateLimitingInterface
 PodWorkqueue workqueue.RateLimitingInterface
}

启动controler需要完成一些初始化工作。

首先就是注册informer的event handler，2种资源的事件处理逻辑各不相同，所以是分开的。资源的事件分3种，Add,Update,Delete，想想kubectl apply/delete -f xxx.yaml就知道了。 Event handler的实现我拆分在别的文件里了，一会再说。

**func** (nginxController \*NginxController) Start() (err error) {
 **var** (
 stopCh = make(**chan struct**{})
 i int
 syncOk bool
 )

 *// pod informer的event handler* nginxController.PodInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
 AddFunc: **func**(obj **interface**{}) {
 nginxController.OnAddPod(obj)
 },
 UpdateFunc: **func**(oldObj, newObj **interface**{}) {
 nginxController.OnUpdatePod(oldObj, newObj)
 },
 DeleteFunc: **func**(obj **interface**{}) {
 nginxController.OnDeletePod(obj)
 },
 })
 *// nginx informer的event handler* nginxController.NginxInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
 AddFunc: **func**(obj **interface**{}) {
 nginxController.OnAddNginx(obj)
 },
 UpdateFunc: **func**(oldObj, newObj **interface**{}) {
 nginxController.OnUpdateNginx(oldObj, newObj)
 },
 DeleteFunc: **func**(obj **interface**{}) {
 nginxController.OnDeleteNginx(obj)
 },
 })

通常我们只需要知道哪个resoure object变了，而不需要知道是add还是delete还是update，所以一般会用到workqueue来做事件的缓冲和去重。

这里为2种资源分别创建了workqueue。

*// event handler会把event丢到workqueue里, 被processor消费*nginxController.NginxWorkqueue = workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), **"Nginx"**)
nginxController.PodWorkqueue = workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), **"Pod"**)

接下来，我们有informer了，informer注册好event handler了，并且event handler要使用的workqueue也创建好了（event handler向workqueue投递事件），就可以启动informer开始监听etcd数据了：

*// nginx informer开始拉取事件，存到local cache，并回调event handler***go** nginxController.NginxInformer.Informer().Run(stopCh)
*// pod informer开始拉取事件，存到local cache，并回调event handler***go** nginxController.PodInformer.Informer().Run(stopCh)

因为我们大部分调度逻辑是基于local cache数据做决策的（频繁访问apiserver获取实时数据比较浪费性能），所以在正式开始处理workqueue中的数据之前，我们得确保etcd的数据全量同步到local cache里，否则就会影响我们调度策略的准确性：比如apiserver里明明有5个POD了，结果local cache里没有，导致我们又请求创建更多POD。

*// 等待etcd已有数据都下载回来, 再启动事件处理线程, 这样local cache可以反馈出贴近准实时的etcd数据，供逻辑决策准确***if** syncOk = cache.WaitForCacheSync(stopCh, nginxController.NginxInformer.Informer().HasSynced, nginxController.PodInformer.Informer().HasSynced); !syncOk {
 err = fmt.Errorf(**"sync失败"**)
 **return**}

等待2个informer首次同步全量数据完成后，我们就可以启动processor消费workqueue中的event了。

*// 启动nginx event processor***for** i = 0; i < 2; i++ {
 **go** nginxController.runNginxWorker()
 **go** nginxController.runPodWorker()
}

Workqueue是线程安全的，随便启几个消费协程。另外呢，workqueue提供独占机制，如果某个event被其他协程取走并且处理中，那么即便这期间相同的event到来，workqueue仍旧可以保证其他协程是取不到这个event的，直到之前那个协程处理完成通知workqueue。 这保证了同一个资源对象，不会被并发消费。

## Nginx\_worker.go

代码地址： <https://github.com/owenliang/k8s-client-go/blob/master/demo10/controller/nginx_worker.go>

Nginx资源的事件，会经过nginx informer到达nginx event handler。做的事情就是把发生变化的资源放入workqueue去重，供Processor消费：

**func** (nginxController \*NginxController)OnAddNginx(obj **interface**{}) {
 *// 把event存到workqueue* nginxController.EnqueueNginx(obj)
}

**func** (nginxController \*NginxController) OnUpdateNginx(oldObj, newObj **interface**{}) {
 *// 把event存到workqueue* nginxController.EnqueueNginx(newObj)
}

**func** (nginxController \*NginxController) OnDeleteNginx(obj **interface**{}) {
 *// 把event存到workqueue* nginxController.EnqueueNginx(obj)
}

workqueue只能add一个字符串，默认是发生变化的资源对象的唯一标识。 其实就是namespace/name作为key，放到workqueue里排队（以及去重）。

一会processor取出key反解到namespace和name，可以去local cache里得到资源对象，所以workqueue并不需要放入真正的资源object，一个标识符就够了。至于addratelimited方法，它会统计同一个key的加入次数，如果加入太频繁就会延迟它加入队列的时间，表现就是消费者短时间取不到这个key，直到失败回避时间过期：

*// 向workqueue设置变化的resource***func** (nginxController \*NginxController) EnqueueNginx(obj **interface**{}) {
 **var** (
 key string
 err error
 )

 **if** key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
 **return** }

 *// 把workqueue里放的是字符串的key, 会针对这个key做限速和去重* nginxController.NginxWorkqueue.AddRateLimited(key)
}

我们在controller代码中拉起的协程，就是消费workqueue：

*// 消费workqueue***func** (nginxController \*NginxController) runNginxWorker() {
 **for** {
 nginxController.processNginxEvent()
 }
}

继续看ProcessNginxEvent的逻辑。

首先从workqueue取出一个key，这里obj其实就是namespace/name的key。无论处理成功还是异常，我们一定要最后通知workqueue说我们处理完成了，不再独占，这样后续相同key进入workqueue才能被其他协程Get到。

*// 处理event***func** (nginxController \*NginxController) processNginxEvent() {
 **var** (
 obj **interface**{}
 key string
 ok bool
 shutdown bool
 err error
 )

 **if** obj, shutdown = nginxController.NginxWorkqueue.Get(); shutdown {
 **return** }

 *// 处理结束, 从队列删除* **defer** nginxController.NginxWorkqueue.Done(obj)

 *// workqueue的key* **if** key, ok = obj.(string); !ok {
 nginxController.NginxWorkqueue.Forget(obj)
 **return** }

接下来进入核心调度逻辑：
*////////// 核心逻辑 ////////////***if** err = nginxController.handleNginxEvent(key); err != nil && !errors.IsNotFound(err) {
 **goto FAIL**}

分析它的实现，首先将变化的nginx对象的namespace和name从key反解出来，然后调用lister()从local cache取出它的信息:

*// 反解workqueue的key, 得到namespace/name***if** namespace, name, err = cache.SplitMetaNamespaceKey(key); err != nil {
 **return**}

*// 获取local cache里的对应Nginx object***if** nginx, err = nginxController.NginxInformer.Lister().Nginxes(namespace).Get(name); err != nil && !errors.IsNotFound(err) {
 **return**}

如果local cache里没有，说明这个nginx object真的被删除了，因为informer是先更新local cache再回调event handler。 我们需要当前nginx对象是不是被删除状态：

**if** errors.IsNotFound(err) {
 nginxNotFound = ***true***}

接下来，我们调用apiserver获取nginx对象关联的POD有哪些。 这是通过筛选label实现的，因为我们后续在创建POD的时候会给POD打上nginxName标签，value就是所属nginx对象的namespace/pod，这样就给POD以及nginx对象产生了逻辑上的关联性，可以互相找得到。 另外，这里我没有使用local cache中的POD信息，而是直接调用apiserver获取了最新的pod列表，这是出于调度对时效性的要求，否则会导致我们的决策出现重要失误，看完完整代码就更理解我所说的了。

*// 筛选出关联的POD*selector = labels.NewSelector()
**if** requirement, err = labels.NewRequirement(**"nginxKey"**, selection.***Equals***, []string{ key }); err != nil {
 **return**}
selector = selector.Add(\*requirement) *// 注意返回值覆盖

// 出于调度实时性的需要, POD列表取apiserver最新的状态, 不走local cache***if** podList, err = nginxController.Clientset.CoreV1().Pods(namespace).List(core\_v1.ListOptions{LabelSelector: selector.String()}); err != nil {
 **return**}
pods = podList.Items

接下来，就是处理nginx对象删除的情况，需要把关联的POD都删除。 就是调用apiserver一个一个Delete掉就可以了，中途出现异常就终止处理，会有上层的workqueue重试机制作用，过会再说。

*// 现有POD数量*podCount = len(pods)

*// nginx已删除, 清理所有关联PODS***if** nginxNotFound {
 **for** i := 0; i < podCount; i++ {
 **if** err = nginxController.Clientset.CoreV1().Pods(pods[i].Namespace).Delete(pods[i].Name, nil); err != nil && !errors.IsNotFound(err) {
 **return** }
 klog.Infoln(**"[Nginx - 清理POD]"**, key, pods[i].Name)
 }
 **return**}

如果nginx还在，那么我们就需要根据现在的POD情况，向nginx期望的POD情况发展，也就是调度。 这里统计一下运行中的POD有几个（running），以及启动中的POD有几个（pending），其他的状态都表示POD挂了/退了/失联了，直接删掉就行。

*// Nginx部署策略: 确保足够数量的POD运行即可
// 1, running+pending<replicas, 那么创建
// 2, running+pending>replicas, 那么删除
// 3, 其他状态的删除

// 统计一下running和pending的POD个数***for** i := 0; i < podCount; i++ {
 **if** pods[i].Status.Phase == v1.***PodRunning*** {
 running++
 } **else if** pods[i].Status.Phase == v1.***PodPending*** {
 pending++
 } **else** { *// 其他状态的删除* **if** err = nginxController.Clientset.CoreV1().Pods(pods[i].Namespace).Delete(pods[i].Name, nil); err != nil && !errors.IsNotFound(err) {
 **return** }
 }
}

接下来，我们就可以进行扩容 or 缩容的处理了。 同一时刻，要么就是POD太少需要扩容，要么就是POD太多需要缩容，没有中间状态，所以逻辑其实就是2选1，很简单。 如果running+pending的比replicas预期少，那么就一个一个Create出来，异常就返回，上层workqueue会做重试逻辑。

*// 扩容***if** running + pending < nginx.Spec.Replicas { *// 不足就补充* toScale := nginx.Spec.Replicas - running - pending
 **for** i := 0; i < toScale; i++ {
 pod = v1.Pod{
 ObjectMeta: core\_v1.ObjectMeta{
 Name: **"nginx-pod-"** + strconv.Itoa(podId),
 Labels: **map**[string]string{**"nginxKey"**: key},
 },
 Spec: v1.PodSpec{
 Containers: []v1.Container{
 {Name: **"nginx"**, Image: **"nginx:latest"**},
 },
 },
 }
 **if** namespace = nginx.Namespace; namespace == **""** {
 namespace = **"default"** }
 **if** created, err = nginxController.Clientset.CoreV1().Pods(namespace).Create(&pod); err != nil {
 **return** }
 pods = append(pods, \*created)
 podId++
 podCount++

 klog.Infoln(**"[Nginx - 扩容POD]"**, key, created.Name)
 }
}

否则就是缩容，优先删掉启动中的pod，其次删掉运行中的pod，这样就快速的缩小了POD规模：

**else if** running + pending > nginx.Spec.Replicas { *// 缩容* toDelete := running + pending - nginx.Spec.Replicas
 *// 先删pending的* **for** i := 0; i < len(pods); i++ {
 **if** toDelete == 0 {
 **break** }
 **if** pods[i].Status.Phase != v1.***PodPending*** {
 **continue** }
 **if** err = nginxController.Clientset.CoreV1().Pods(pods[i].Namespace).Delete(pods[i].Name, nil); err != nil && !errors.IsNotFound(err) {
 **return** }
 toDelete--
 klog.Infoln(**"[Nginx - 缩容POD]"**, key, pods[i].Name)
 }
 *// 再删running的* **for** i := 0; i < len(pods); i++ {
 **if** toDelete == 0 {
 **break** }
 **if** pods[i].Status.Phase != v1.***PodRunning*** {
 **continue** }
 **if** err = nginxController.Clientset.CoreV1().Pods(pods[i].Namespace).Delete(pods[i].Name, nil); err != nil && !errors.IsNotFound(err){
 **return** }
 toDelete--
 klog.Infoln(**"[Nginx - 缩容POD]"**, key, pods[i].Name)
 }
}

最后就说一下workqueue的统一重试逻辑是什么意思，回到之前的代码。 如果处理事件过程中遇到任何异常，那么跳到FAIL标签，重新把key添加到workqueue，等待重试即可，workqueue会在合适的延迟后重新允许我们Get到这个key。 如果处理正常，那么就Forget掉这个key，其实就是重置它的重试次数，这样下次再来同样的event就可以理解消费得到：

 *////////// 核心逻辑 ////////////* **if** err = nginxController.handleNginxEvent(key); err != nil && !errors.IsNotFound(err) {
 **goto FAIL** }

 *// 处理成功，重置失败计数* nginxController.NginxWorkqueue.Forget(obj)
 **return

FAIL**:
 klog.Errorln(**"[Nginx - 处理异常]"**, key, err)
 *// 处理失败, 重新放回队列, 累加限速计数* nginxController.NginxWorkqueue.AddRateLimited(key)

同时也需要，defer Done()总是调用的，它不影响我们重新addRateLimited，底层是双buffer机制，即add之后再done，接下来还是可以取到这次add的event。

## Pod\_worker.go

代码地址： <https://github.com/owenliang/k8s-client-go/blob/master/demo10/controller/pod_worker.go>

任意POD变化，都会通知pod informer，进而回调pod event handler，我们要做的就是找出包含nginxKey标签的POD，这种属于我们关心的POD。POD的Cevent通常是我们对nginx调度导致的，但也可能是POD自身状态变化（比如运行失败）或者人工操作了POD导致的事件。 无论哪种事件，我们只需要找出POD属于哪个nginx对象，然后向nginx的workqueue投递一个event，让nginx的现有调度算法去check最新的情况即可，即POD这边并不需要什么调度逻辑，全部收敛在nginx资源的调度中即可。

所以，我们的实现很简单，不需要用workqueue，不需要processor，直接在event handler里做事件转发即可。

**func** (nginxController \*NginxController)OnAddPod(obj **interface**{}) {
 nginxController.handlePodObject(obj)
}

**func** (nginxController \*NginxController) OnUpdatePod(oldObj, newObj **interface**{}) {
 nginxController.handlePodObject(newObj)
}

**func** (nginxController \*NginxController) OnDeletePod(obj **interface**{}) {
 nginxController.handlePodObject(obj)
}

*// 消费workqueue***func** (nginxController \*NginxController) runPodWorker() {
 *// 什么也不做*}

具体的转发逻辑就是取出nginxKey标签，作为key投递给nginx的workqueue：

**func** (nginxController \*NginxController) handlePodObject(obj **interface**{}) (err error) {
 **var** (
 pod \*v1.Pod
 ok bool
 nginxKey string
 hasLabel bool
 )

 *// 反解出Pod* **if** pod, ok = obj.(\*v1.Pod); !ok {
 **return** }

 *// 确认属于nginx部署的POD* **if** nginxKey, hasLabel = pod.Labels[**"nginxKey"**]; !hasLabel {
 **return** *// 不属于nginx部署的POD， 忽略* }

 *// 投递给nginx的workqueue* nginxController.NginxWorkqueue.AddRateLimited(nginxKey)
 klog.Infoln(**"[POD - 更新]"**, nginxKey, pod.Name)
 **return**}