编写控制器

本文主要介绍如何使用 WorkQueue 来编写控制器

介绍

Kubernetes 控制器是一个主动调谐的过程，它会 watch 一些对象的期望状态，也会 watch 实际的状态，然后，控制器发送一些指令尝试让对象的当前状态往期望状态迁移。

控制器最简单的实现就是一个循环：

```
for {
  desired := getDesiredState()
  current := getCurrentState()
  makeChanges(desired, current)
}
```

Watches 这些都只是这个逻辑的优化。

https://github.com/kubernetes/community/blob/master/contributors/devel/sig-api-machinery/controllers.md

指南

当你在编写控制器时，有一些准则将有助于确保你得到你需要的结果和。

- 一次只操作一个元素。如果你使用了 `workqueue.Interface`，你可以将某个资源的变化排成队列，随后将它们弹到多个 "worker " gofuncs 中，并保证没有两个 gofuncs 会同时对同一个元素进行操作。
- 很多控制器必须触发掉多个资源（我需要 "如果Y发生变化就检查X"），但几乎所有的控制器都可以根据关系将这些资源折叠成一个 "检查这个X "的队列。例如，ReplicaSet 控制器需要对一个 Pod 被删除做出反应，但它通过找到相关的 ReplicaSets 并对这些进行排队来实现。
- 资源之间的随机排序。当控制器排队关闭多种类型的资源时，无法保证这些资源之间的排序。
- 不同的 watches 是独立更新的。即使有 "创建的资源A/X "和 "创建的资源B/Y "的客观排序，你的控制器也可以观察到 "创建的资源B/Y "和 "创建的资源A/X"。
- 级别驱动，而不是边缘驱动。就像有一个 shell 脚本不是一直在运行一样，你的控制器可能会在再次运行之前关闭不确定的时间。
- 如果一个 API 对象出现的标记值为 true，你不能指望已经看到它从false 变成 true，只能说你现在观察到它是 true。即使是 API watch 也会受到这个问题的影响，所以要确保你不指望看到变化，除非你的控制器也在对象的状态中标记它最后做出决定的信息。
- 使用 SharedInformers。SharedInformers 提供了回调函数来接收特定资源的添加、更新和删除的通知，它们还提供了访问共享缓存和确定缓存何时启动的便利功能。

  使用 `https://git.k8s.io/kubernetes/staging/src/k8s.io/client-go/informers/factory.go` 中的工厂方法来确保你和其他人共享同一个缓存实例。

  这样我们就大大减少了 APIServer 的连接以及重复的序列化、重复的反序列化、重复的缓存等成本。

  你可能会看到其他机制，比如反射器和 DeltaFIFO 驱动控制器。这些都是旧的机制，我们后来用它们来构建 SharedInformers，你应该避免在新控制器中使用它们。
- 永远不要 Mutate 原始对象!! 缓存是跨控制器共享的，这意味着如果你修改了你的对象的 "副本"（实际上是引用或浅层副本），你会搞乱其他控制器（不仅仅是你自己的）。

最常见的失败方式是做一个浅层拷贝，然后 Mutate 一个映射，比如Annotations。使用 `api.Scheme.Copy` 进行深层复制。

```
package main

import (
  "flag"
  "fmt"
  "path/filepath"
  "time"

  v1 "k8s.io/api/core/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/apimachinery/pkg/fields"
  "k8s.io/apimachinery/pkg/util/runtime"
  "k8s.io/apimachinery/pkg/util/wait"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/rest"
  "k8s.io/client-go/tools/cache"
  "k8s.io/client-go/tools/clientcmd"
  "k8s.io/client-go/util/homedir"
  "k8s.io/client-go/util/workqueue"
  "k8s.io/klog"
)

type Controller struct {
  indexer  cache.Indexer
  queue    workqueue.RateLimitingInterface
  informer cache.Controller
}

func NewController(queue workqueue.RateLimitingInterface, indexer cache.Indexer, informer cache.Controller) *Controller {
  return &Controller{
    informer: informer,
    indexer:  indexer,
    queue:    queue,
  }
}

func (c *Controller) processNextItem() bool {
  // 等到工作队列中有一个新元素
  key, quit := c.queue.Get()
  if quit {
    return false
  }
  // 告诉队列我们已经完成了处理此 key 的操作
  // 这将为其他 worker 解锁该 key
  // 这将确保安全的并行处理，因为永远不会并行处理具有相同 key 的两个Pod
  defer c.queue.Done(key)

  // 调用包含业务逻辑的方法
  err := c.syncToStdout(key.(string))
  // 如果在执行业务逻辑期间出现错误，则处理错误
  c.handleErr(err, key)
  return true
}

// syncToStdout 是控制器的业务逻辑实现
// 在此控制器中，它只是将有关 Pod 的信息打印到 stdout
// 如果发生错误，则简单地返回错误
// 此外重试逻辑不应成为业务逻辑的一部分。
func (c *Controller) syncToStdout(key string) error {
  // 从本地存储中获取 key 对应的对象
  obj, exists, err := c.indexer.GetByKey(key)
  if err != nil {
    klog.Errorf("Fetching object with key %s from store failed with %v", key, err)
    return err
  }
  if !exists {
    fmt.Printf("Pod %s does not exists anymore\n", key)
  } else {
    fmt.Printf("Sync/Add/Update for Pod %s\n", obj.(*v1.Pod).GetName())
  }
  return nil
}

// 检查是否发生错误，并确保我们稍后重试
func (c *Controller) handleErr(err error, key interface{}) {
  if err == nil {
    // 忘记每次成功同步时 key 的#AddRateLimited历史记录。
    // 这样可以确保不会因过时的错误历史记录而延迟此 key 更新的以后处理。
    c.queue.Forget(key)
```

```
    return
  }
  //如果出现问题，此控制器将重试5次
  if c.queue.NumRequeues(key) < 5 {
    klog.Infof("Error syncing pod %v: %v", key, err)
    // 重新加入 key 到限速队列
    // 根据队列上的速率限制器和重新入队历史记录，稍后将再次处理该 key
    c.queue.AddRateLimited(key)
    return
  }
  c.queue.Forget(key)
  // 多次重试，我们也无法成功处理该key
  runtime.HandleError(err)
  klog.Infof("Dropping pod %q out of the queue: %v", key, err)
}

// Run 开始 watch 和同步
func (c *Controller) Run(threadiness int, stopCh chan struct{}) {
  defer runtime.HandleCrash()

  // 停止控制器后关掉队列
  defer c.queue.ShutDown()
  klog.Info("Starting Pod controller")

  // 启动
  go c.informer.Run(stopCh)

  // 等待所有相关的缓存同步，然后再开始处理队列中的项目
  if !cache.WaitForCacheSync(stopCh, c.informer.HasSynced) {
    runtime.HandleError(fmt.Errorf("Timed out waiting for caches to sync"))
    return
  }

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

  <-stopCh
  klog.Info("Stopping Pod controller")
}

func (c *Controller) runWorker() {
  for c.processNextItem() {
  }
}

func initClient() (*kubernetes.Clientset, 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
  return kubernetes.NewForConfig(config)
}

func main() {
  clientset, err := initClient()
  if err != nil {
    klog.Fatal(err)
  }

  // 创建 Pod ListWatcher
  podListWatcher := cache.NewListWatchFromClient(clientset.CoreV1().RESTClient(), "pods", v1.NamespaceDefault, fields.Everything())

  // 创建队列
  queue := workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())

  // 在 informer 的帮助下，将工作队列绑定到缓存
  // 这样，我们确保无论何时更新缓存，都将 pod key 添加到工作队列中
  // 注意，当我们最终从工作队列中处理元素时，我们可能会看到 Pod 的版本比响应触发更新的版本新
```

```
  indexer, informer := cache.NewIndexerInformer(podListWatcher, &v1.Pod{}, 0, cache.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
      key, err := cache.MetaNamespaceKeyFunc(obj)
      if err == nil {
        queue.Add(key)
      }
    },
    UpdateFunc: func(old interface{}, new interface{}) {
      key, err := cache.MetaNamespaceKeyFunc(new)
      if err == nil {
        queue.Add(key)
      }
    },
    DeleteFunc: func(obj interface{}) {
      key, err := cache.DeletionHandlingMetaNamespaceKeyFunc(obj)
      if err == nil {
        queue.Add(key)
      }
    },
  }, cache.Indexers{})

  controller := NewController(queue, indexer, informer)

  indexer.Add(&v1.Pod{
    ObjectMeta: metav1.ObjectMeta{
      Name:      "mypod",
      Namespace: v1.NamespaceDefault,
    },
  })

  // start controller
  stopCh := make(chan struct{})
  defer close(stopCh)
  go controller.Run(1, stopCh)

  select {}
}
```