Informer 使用

 本文主要介绍 client-go 的 Informer 和 Cache

介绍

前面我们在使用 Clientset 的时候了解到我们可以使用 Clientset 来获取所有的原生资源对象，那么如果我们想要去一直获取集群的资源对象数据呢？岂不是需要用一个轮询去不断执行 List() 操作？这显然是不合理的，实际上除了常用的 CRUD 操作之外，我们还可以进行 Watch 操作，可以监听资源对象的增、删、改、查操作，这样我们就可以根据自己的业务逻辑去处理这些数据了。

Watch 通过一个 event 接口监听对象的所有变化（添加、删除、更新）：

```
// staging/src/k8s.io/apimachinery/pkg/watch/watch.go
// Interface 可以被任何知道如何 watch 和通知变化的对象实现
type Interface interface {
  // Stops watching. Will close the channel returned by ResultChan(). Releases
  // any resources used by the watch.
  Stop()

  // Returns a chan which will receive all the events. If an error occurs
  // or Stop() is called, this channel will be closed, in which case the
  // watch should be completely cleaned up.
  ResultChan() <-chan Event
}
```

watch 接口的 ResultChan 方法会返回如下几种事件：

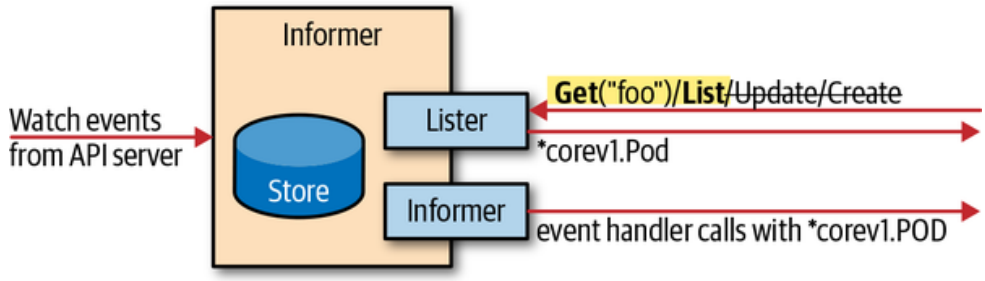
```
// staging/src/k8s.io/apimachinery/pkg/watch/watch.go
// EventType 定义可能的事件类型
type EventType string

const (
  Added    EventType = "ADDED"
  Modified EventType = "MODIFIED"
  Deleted  EventType = "DELETED"
  Bookmark EventType = "BOOKMARK"
  Error    EventType = "ERROR"

  DefaultChanSize int32 = 100
)

// Event represents a single event to a watched resource.
// +k8s:deepcopy-gen=true
type Event struct {
  Type EventType

  // Object is:
  //  * If Type is Added or Modified: the new state of the object.
```

```
  //  * If Type is Deleted: the state of the object immediately before deletion.
  //  * If Type is Bookmark: the object (instance of a type being watched) where
  //    only ResourceVersion field is set. On successful restart of watch from a
  //    bookmark resourceVersion, client is guaranteed to not get repeat event
  //    nor miss any events.
  //  * If Type is Error: *api.Status is recommended; other types may make sense
  //    depending on context.
  Object runtime.Object
}
```

这个接口虽然我们可以直接去使用，但是实际上并不建议这样使用，因为往往由于集群中的资源较多，我们需要自己在客户端去维护一套缓存，而这个维护成本也是非常大的，为此 client-go 也提供了自己的实现机制，那就是 `Informers`。Informers 是这个事件接口和带索引查找功能的内存缓存的组合，这样也是目前最常用的用法。Informers 第一次被调用的时候会首先在客户端调用 List 来获取全量的对象集合，然后通过 Watch 来获取增量的对象更新缓存。

运行原理

一个控制器每次需要获取对象的时候都要访问 APIServer，这会给系统带来很高的负载，Informers 的内存缓存就是来解决这个问题的，此外 Informers 还可以几乎实时的监控对象的变化，而不需要轮询请求，这样就可以保证客户端的缓存数据和服务端的数据一致，就可以大大降低 APIServer 的压力了。



Informers

如上图展示了 Informer 的基本处理流程：

- 以 events 事件的方式从 APIServer 获取数据
- 提供一个类似客户端的 Lister 接口，从内存缓存中 get 和 list 对象
- 为添加、删除、更新注册事件处理程序

此外 Informers 也有错误处理方式，当长期运行的 watch 连接中断时，它们会尝试使用另一个 watch 请求来恢复连接，在不丢失任何事件的情况下恢复事件流。如果中断的时间较长，而且 APIServer 丢失了事件（etcd 在新的 watch 请求成功之前从数据库中清除了这些事件），那么 Informers 就会重新 List 全量数据。

而且在重新 List 全量操作的时候还可以配置一个重新同步的周期参数，用于协调内存缓存数据和业务逻辑的数据一致性，每次过了该周期后，注册的事件处理程序就将被所有的对象调用，通常这个周期参数以分为单位，比如10分钟或者30分钟。

> 注意：重新同步是纯内存操作，不会触发对服务器的调用。

Informers 的这些高级特性以及超强的鲁棒性，都足以让我们不去直接使用客户端的 Watch() 方法来处理自己的业务逻辑，而且在 Kubernetes 中也有很多地方都有使用到 Informers。但是在使用 Informers 的时候，通常每个 GroupVersionResource（GVR）只实例化一个 Informers，但是有时候我们在一个应用

中往往有使用多种资源对象的需求，这个时候为了方便共享 Informers，我们可以通过使用共享 Informer 工厂来实例化一个 Informer。

共享 Informer 工厂允许我们在应用中为同一个资源共享 Informer，也就是说不同的控制器循环可以使用相同的 watch 连接到后台的 APIServer，例如，kube-controller-manager 中的控制器数据量就非常多，但是对于每个资源（比如 Pod），在这个进程中只有一个 Informer。

示例

首先我们创建一个 Clientset 对象，然后使用 Clientset 来创建一个共享的 Informer 工厂，Informer 是通过 `informer-gen` 这个代码生成器工具自动生成的，位于 `k8s.io/client-go/informers` 中。

这里我们来创建一个用于获取 Deployment 的共享 Informer，代码如下所示：

```
package main

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

  v1 "k8s.io/api/apps/v1"
  "k8s.io/apimachinery/pkg/labels"
  "k8s.io/client-go/informers"
  "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"
)

func main() {
  var err error
  var config *rest.Config

  var kubeconfig *string

  if home := homedir.HomeDir(); home != "" {
    kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"), "[可选] kubeconfig 绝对路径")
  } else {
    kubeconfig = flag.String("kubeconfig", "", "kubeconfig 绝对路径")
  }
  // 初始化 rest.Config 对象
  if config, err = rest.InClusterConfig(); err != nil {
    if config, err = clientcmd.BuildConfigFromFlags("", *kubeconfig); err != nil {
      panic(err.Error())
    }
  }
  // 创建 Clientset 对象
  clientset, err := kubernetes.NewForConfig(config)
  if err != nil {
    panic(err.Error())
  }

  // 初始化 informer factory（为了测试方便这里设置每30s重新 List 一次）
  informerFactory := informers.NewSharedInformerFactory(clientset, time.Second*30)
  // 对 Deployment 监听
  deployInformer := informerFactory.Apps().V1().Deployments()
  // 创建 Informer（相当于注册到工厂中去，这样下面启动的时候就会去 List & Watch 对应的资源）
  informer := deployInformer.Informer()
  // 创建 Lister
  deployLister := deployInformer.Lister()
  // 注册事件处理程序
  informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc:    onAdd,
```

```
    UpdateFunc: onUpdate,
    DeleteFunc: onDelete,
  })

  stopper := make(chan struct{})
  defer close(stopper)

  // 启动 informer，List & Watch
  informerFactory.Start(stopper)
  // 等待所有启动的 Informer 的缓存被同步
  informerFactory.WaitForCacheSync(stopper)

  // 从本地缓存中获取 default 中的所有 deployment 列表
  deployments, err := deployLister.Deployments("default").List(labels.Everything())
  if err != nil {
    panic(err)
  }
  for idx, deploy := range deployments {
    fmt.Printf("%d -> %s\n", idx+1, deploy.Name)
  }
  <-stopper
}

func onAdd(obj interface{}) {
  deploy := obj.(*v1.Deployment)
  fmt.Println("add a deployment:", deploy.Name)
}

func onUpdate(old, new interface{}) {
  oldDeploy := old.(*v1.Deployment)
  newDeploy := new.(*v1.Deployment)
  fmt.Println("update deployment:", oldDeploy.Name, newDeploy.Name)
}

func onDelete(obj interface{}) {
  deploy := obj.(*v1.Deployment)
  fmt.Println("delete a deployment:", deploy.Name)
}
```

上面的代码运行可以获得 default 命名空间之下的所有 Deployment 信息以及整个集群的 Deployment 数据：

```
$ go run main.go
add a deployment: dingtalk-hook
add a deployment: spin-orca
add a deployment: argocd-server
add a deployment: istio-egressgateway
add a deployment: vault-agent-injector
add a deployment: rook-ceph-osd-0
add a deployment: rook-ceph-osd-2
add a deployment: code-server
......
1 -> nginx
2 -> helloworld-v1
3 -> productpage-v1
4 -> details-v1
......
```

这是因为我们首先通过 Informer 注册了事件处理程序，这样当我们启动 Informer 的时候首先会将集群的全量 Deployment 数据同步到本地的缓存中，会触发 `AddFunc` 这个回调函数，然后我们又在下面使用 `Lister()` 来获取 default 命名空间下面的所有 Deployment 数据，这个时候的数据是从本地的缓存中获取的，所以就看到了上面的结果，由于我们还配置了每30s重新全量 List 一次，所以正常每30s我们也可以看到所有的 Deployment 数据出现在 `UpdateFunc` 回调函数下面，我们也可以尝试去删除一个 Deployment，同样也会出现对应的 `DeleteFunc` 下面的事件。