Shared Informer 源码分析

本文主要对 SharedInformer 组件进行分析说明。

介绍
SharedInformer
Controller
sharedProcessor
SharedInformer 的实现

介绍

上节课我们分析了 Indexer 组件的实现，实际上最开始的时候我们在 Informer 示例中通过 Informer 的 Lister 获取的资源对象数据就来自于 Indexer，当然除了 Lister 之外最重要的就是资源对象事件监听的操作，这些都是在 SharedInformer 中去实现的，所以我们需要去分析下 SharedInformer 的实现，这样就可以完整的将前面的内容串联起来了。

SharedInformer

我们平时说的 Informer 其实就是 SharedInformer，它是可以共享使用的。如果同一个资源的 Informer 被实例化多次，那么就会运行多个 ListAndWatch 操作，这会加大 APIServer 的压力。而 SharedInformer 通过一个 map 来让同一类资源的 Informer 实现共享一个 Refelctor，这样就不会出现上面这个问题了。接下来我们先来查看 SharedInformer 的具体实现：

```
// k8s.io/client-go/tools/cache/shared_informer.go

type SharedInformer interface {
    // 添加资源事件处理器，当有资源变化时就会通过回调通知使用者
    AddEventHandler(handler ResourceEventHandler)
    // 需要周期同步的资源事件处理器
    AddEventHandlerWithResyncPeriod(handler ResourceEventHandler, resyncPeriod time.Duration)

    // 获取一个 Store 对象，前面我们讲解了很多实现 Store 的结构
    GetStore() Store
    // 获取一个 Controller，下面会详细介绍，主要是用来将 Reflector 和 DeltaFIFO 组合到一起工作
    GetController() Controller

    // SharedInformer 的核心实现，启动并运行这个 SharedInformer
    // 当 stopCh 关闭时候，informer 才会退出
    Run(stopCh <-chan struct{})

    // 告诉使用者全量的对象是否已经同步到了本地存储中
    HasSynced() bool
    // 最新同步资源的版本
    LastSyncResourceVersion() string
}

// 在 SharedInformer 基础上扩展了添加和获取 Indexers 的能力
type SharedIndexInformer interface {
    SharedInformer
    // 在启动之前添加 indexers 到 informer 中
    AddIndexers(indexers Indexers) error
    GetIndexer() Indexer
}
```

如果我们要处理资源的事件的话，就需要添加一个事件处理器，传入一个 `ResourceEventHandler` 接口，其定义如下所示：

```
// k8s.io/client-go/tools/cache/controller.go

type ResourceEventHandler interface {
    // 添加对象回调函数
    OnAdd(obj interface{})
    // 更新对象回调函数
    OnUpdate(oldObj, newObj interface{})
    // 删除对象回调函数
    OnDelete(obj interface{})
}
```

然后接下来我们来看看 `SharedIndexInformer` 的具体实现类的定义：

```
// k8s.io/client-go/tools/cache/shared_informer.go

type sharedIndexInformer struct {
    // Indexer也是一种Store，这个我们知道的，Controller负责把Reflector和FIFO逻辑串联起来
    // 所以这两个变量就涵盖了开篇那张图里面的Reflector、DeltaFIFO和LocalStore(cache)
    indexer    Indexer
    // 在 Controller 中将 Reflector 和 DeltaFIFO 关联了起来
    controller Controller

    // 对 ResourceEventHandler 进行了一层层封装，统一由 sharedProcessor 管理
    processor             *sharedProcessor

    // 监控对象在一个时间窗口内是否发生了变化
    cacheMutationDetector MutationDetector

    // 用于 Reflector 中真正执行 ListAndWatch 的操作
    listerWatcher ListerWatcher

    // informer 中要处理的对象
    objectType    runtime.Object

    // 定期同步周期
    resyncCheckPeriod time.Duration
    // 任何通过 AddEventHandler 添加的处理程序的默认重新同步的周期
    defaultEventHandlerResyncPeriod time.Duration
    clock clock.Clock

    // 启动、停止标记
    started, stopped bool
    startedLock      sync.Mutex

    blockDeltas sync.Mutex
}
```

Controller

上面我们看到在 sharedIndexInformer 中定义了一个 Controller，这里的 Controller 并不是我们比较熟悉的 kube-controller-manager 管理的各种控制器，这里的 Controller 定义在 `client-go/tools/cache/controller.go` 中，目的是用来把 Reflector、DeltaFIFO 这些组件组合起来形成一个相对固定的、标准的处理流程。我们先来看下 Controller 的定义：

```
// k8s.io/client-go/tools/cache/controller.go

// Controller 的抽象接口
type Controller interface {
    // Run 函数主要做两件事，一件是构造并运行一个 Reflector 反射器，将对象/通知从 Config 的
    // ListerWatcher 送到 Config 的 Queue 队列，并在该队列上调用 Resync 操作
    // 另外一件事就是不断从队列中弹出对象，并使用 Config 的 ProcessFunc 进行处理
    Run(stopCh <-chan struct{})
    HasSynced() bool                  // APIServer 中的资源对象是否同步到了 Store 中
    LastSyncResourceVersion() string // 最新的资源版本号
}
```

因为 Controller 把多个模块整合起来实现了一套业务逻辑，所以在创建Controller 的时候需要提供一些配置：

```
// k8s.io/client-go/tools/cache/controller.go

type Config struct {
    Queue                         // 资源对象的队列，其实就是一个 DeltaFIFO
    ListerWatcher                 // 用来构造 Reflector 的
    Process ProcessFunc           // DeltaFIFO 队列 Pop 的时候回调函数，用于处理弹出的对象
    ObjectType runtime.Object     // 对象类型，也就是 Reflector 中使用的
    FullResyncPeriod time.Duration // 全量同步周期，在 Reflector 中使用
    ShouldResync ShouldResyncFunc  // Reflector 中是否需要 Resync 操作
    RetryOnError bool             // 出现错误是否需要重试
}
```

Controller 自己构造 Reflector 获取对象，Reflector 作为 DeltaFIFO 生产者持续监控 APIServer 的资源变化并推送到队列中。Controller 的 Run() 就是是队列的消费者，从队列中弹出对象并调用 Process() 进行处理。接下来我们来看 Controller 的一个具体实现 controller：

```
// k8s.io/client-go/tools/cache/controller.go
```

```
// controller是 Controller 的一个具体实现
type controller struct {
    config         Config       // 配置
    reflector      *Reflector   // 反射器
    reflectorMutex sync.RWMutex // 反射器的锁
    clock          clock.Clock  // 时钟
}

// 控制器核心实现
func (c *controller) Run(stopCh <-chan struct{}) {
    defer utilruntime.HandleCrash()
    // 新建一个协程，如果收到系统退出的信号就关闭队列
    go func() {
        <-stopCh
        c.config.Queue.Close()
    }()
    // 实例化一个 Reflector，传入的参数都是从 Config 中获取的
    r := NewReflector(
        c.config.ListerWatcher,
        c.config.ObjectType,
        c.config.Queue,
        c.config.FullResyncPeriod,
    )
    r.ShouldResync = c.config.ShouldResync
    r.clock = c.clock

    // 将反射器给到controller
    c.reflectorMutex.Lock()
    c.reflector = r
    c.reflectorMutex.Unlock()

    // 等待所有协程执行完毕
    var wg wait.Group
    defer wg.Wait()

    // StartWithChannel 会启动协程执行 Reflector.Run()，接收到 stopCh 信号才会退出协程
    wg.StartWithChannel(stopCh, r.Run)

    // wait.Unitl() 就是周期性的调用 c.processLoop() 操作处理弹出的对象
    wait.Until(c.processLoop, time.Second, stopCh)
}
```

从上面的核心函数 Run 的实现方式来看，该函数中主要就是实例化一个 Reflector，然后启动一个协程去执行这个反射器的 Run 函数，这个 Run 函数前面我们已经讲解过就是去调用 `ListAndWatch` 函数进行 List 和 Watch 操作，这个操作中具体的实现就是 Config 中的 `ListerWatcher`。然后的一个核心就是 processLoop() 函数的实现：

```
// k8s.io/client-go/tools/cache/controller.go

// 处理队列弹出的对象
func (c *controller) processLoop() {
  // 死循环，不断从队列中弹出对象来处理
  for {
    // 从队列中弹出一个对象，然后处理这个对象
    // 真正处理的是通过 Config 传递进来的 Process 函数
    obj, err := c.config.Queue.Pop(PopProcessFunc(c.config.Process))
    if err != nil {
      // 如果队列关闭了那就直接退出了
      if err == ErrFIFOClosed {
        return
      }
      // 如果配置的是错误后允许重试
      if c.config.RetryOnError {
        // 如果错误可以再重试那么将弹出的对象重新入队列进行处理
        c.config.Queue.AddIfNotPresent(obj)
      }
    }
  }
}
```

上面的代码其实核心的处理就是从 DeltaFIFO 中不断 Pop 出一个对象，然后交给 Config 传递进来的 Process 函数去处理，这个函数是在 SharedInformer 中初始化的时候传递进来的。

sharedProcessor

然后上面 `SharedIndexInformer` 的实现中还有一个比较重要的属性就是 `sharedProcessor`，就是专门来处理事件的，通过 AddEventHandler 函数添加的处理器就会被封装成 processorListener，然后通过 sharedProcessor 管理起来，其定义如下所示：

```
// k8s.io/client-go/tools/cache/shared_informer.go

// sharedProcessor 有一个 processorListener 的集合，可以向它的监听器分发事件通知对象。
type sharedProcessor struct {
  listenersStarted bool  // 所有处理器是否已经启动
  listenersLock    sync.RWMutex  // 读写锁
  listeners        []*processorListener  // 通用的处理器列表
  syncingListeners []*processorListener  // 需要定时同步的处理器列表
  clock            clock.Clock
  wg               wait.Group
}

type processorListener struct {
  nextCh chan interface{}
  addCh  chan interface{}  // 添加事件的通道

  handler ResourceEventHandler

  // pendingNotifications 是一个无边界的环形缓冲区，用于保存所有尚未分发的通知。
  pendingNotifications buffer.RingGrowing

  requestedResyncPeriod time.Duration

  resyncPeriod time.Duration

  nextResync time.Time

  resyncLock sync.Mutex
}
```

processorListener 中就包含一个资源事件处理器，那么我们是如何传递事件进来的呢？首先我们来看看添加一个处理器是如何实现的：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (p *processorListener) add(notification interface{}) {
  p.addCh <- notification
}
```

可以看到添加事件很简单，直接通过 addCh 这个通道接收，notification 就是我们所说的事件，也就是前面我们常说的 DeltaFIFO 输出的 Deltas。上面我们可以看到 addCh 是定义成的一个无缓冲通道，所以这个 add() 函数就是一个事件分发器，从 DeltaFIFO 中弹出的对象要逐一送到多个处理器，如果处理器没有及时处理 addCh 则会阻塞住：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (p *processorListener) pop() {
  defer utilruntime.HandleCrash()
  defer close(p.nextCh) // 通知 run() 函数停止

  var nextCh chan<- interface{}
  var notification interface{}
  // 死循环
  for {
    select {
    // nextCh 还没初始化时，会被阻塞
    case nextCh <- notification:
      // 如果发送成功了（下面的 run 函数中消耗了数据后），从缓冲中再取一个事件出来
      var ok bool
      notification, ok = p.pendingNotifications.ReadOne()
      if !ok { // 没有事件被 Pop，设置 nextCh 为 nil
        nextCh = nil // Disable 这个 select 的 case
      }
    // 从 p.addCh 通道中读取一个事件，消费 addCh 通道
    case notificationToAdd, ok := <-p.addCh:
      if !ok {  // 如果关闭了，则退出
        return
      }
      // notification 为空说明还没发送任何事件给处理器（pendingNotifications 为空）
      if notification == nil {
        // 把刚刚获取的事件通过 p.nextCh 发送给处理器
        notification = notificationToAdd
        nextCh = p.nextCh
      } else {
        // 上一个事件还没发送完成（已经有一个通知等待发送），就先放到缓冲通道中
        p.pendingNotifications.WriteOne(notificationToAdd)
      }
    }
```

```
  }
}
```

pop() 函数的实现的利用了 golang 的 select 来同时操作多个 channel ，select 的 case 表达式都没有满足求值条件，那么 select 语句就会被阻塞，直到至少有一个 case 表达式满足条件为止，如果多个 case 语句同时满足则随机选择一个 case 执行。接下来，我们看看从 nextCh 读取事件后是如何处理的：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (p *processorListener) run() {
  // 当关闭 stopCh 后才会退出
  stopCh := make(chan struct{})
  wait.Until(func() {
    // 不断从 nextCh 通道中取数据
    for next := range p.nextCh {
      // 判断事件类型
      switch notification := next.(type) {
      case updateNotification:
        p.handler.OnUpdate(notification.oldObj, notification.newObj)
      case addNotification:
        p.handler.OnAdd(notification.newObj)
      case deleteNotification:
        p.handler.OnDelete(notification.oldObj)
      default:
        utilruntime.HandleError(fmt.Errorf("unrecognized notification: %T", next))
      }
    }
    // 当 p.nextCh 是空的且是关闭的时候才能到达这里，关闭 stopCh
    close(stopCh)
  }, 1*time.Second, stopCh)
}
```

run() 和 pop() 是 processorListener 的两个最核心的函数，processorListener 就是实现了事件的缓冲和处理，在没有事件的时候可以阻塞处理器，当事件较多是可以把事件缓冲起来，实现了事件分发器与处理器的异步处理。processorListener 的 run() 和 pop() 函数其实都是通过 sharedProcessor 启动的协程来调用的，所以下面我们再来对 sharedProcessor 进行分析了。首先看下如何添加一个 processorListener：

```
// k8s.io/client-go/tools/cache/shared_informer.go

// 添加处理器
func (p *sharedProcessor) addListener(listener *processorListener) {
  p.listenersLock.Lock()  // 加锁
  defer p.listenersLock.Unlock()
  // 调用 addListenerLocked 函数
  p.addListenerLocked(listener)
  // 如果事件处理列表中的处理器已经启动了，则手动启动下面的两个协程
  // 相当于启动后了
  if p.listenersStarted {
    // 通过 wait.Group 启动两个协程，就是上面我们提到的 run 和 pop 函数
    p.wg.Start(listener.run)
    p.wg.Start(listener.pop)
  }
}

// 将处理器添加到处理器的列表中
func (p *sharedProcessor) addListenerLocked(listener *processorListener) {
  // 添加到通用处理器列表中
  p.listeners = append(p.listeners, listener)
  // 添加到需要定时同步的处理器列表中
  p.syncingListeners = append(p.syncingListeners, listener)
}
```

这里添加处理器的函数 addListener 其实在 sharedIndexInformer 中的 AddEventHandler 函数中就会调用这个函数来添加处理器。然后就是事件分发的函数实现：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (p *sharedProcessor) distribute(obj interface{}, sync bool) {
  p.listenersLock.RLock()
  defer p.listenersLock.RUnlock()
  // sync 表示 obj 对象是否是同步事件对象
  // 将对象分发给每一个事件处理器列表中的处理器
  if sync {
    for _, listener := range p.syncingListeners {
      listener.add(obj)
```

```
    }
  } else {
    for _, listener := range p.listeners {
      listener.add(obj)
    }
  }
}
```

然后就是将 sharedProcessor 运行起来：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (p *sharedProcessor) run(stopCh <-chan struct{}) {
  func() {
    p.listenersLock.RLock()
    defer p.listenersLock.RUnlock()
    // 遍历所有的处理器，为处理器启动两个后台协程：run 和 pop 操作
    // 后续添加的处理器就是在上面的 addListener 中去启动的
    for _, listener := range p.listeners {
      p.wg.Start(listener.run)
      p.wg.Start(listener.pop)
    }
    // 标记为所有处理器都已启动
    p.listenersStarted = true
  }()
  // 等待退出信号
  <-stopCh
  // 接收到退出信号后，关闭所有的处理器
  p.listenersLock.RLock()
  defer p.listenersLock.RUnlock()
  // 遍历所有处理器
  for _, listener := range p.listeners {
    // 关闭 addCh，pop 会停止，pop 会通知 run 停止
    close(listener.addCh)
  }
  // 等待所有协程退出，就是上面所有处理器中启动的两个协程 pop 与 run
  p.wg.Wait()
}
```

到这里 sharedProcessor 就完成了对 ResourceEventHandler 的封装处理，当然最终 sharedProcessor 还是在 SharedInformer 中去调用的。

SharedInformer 的实现

接下来我们就来看下 SharedInformer 的具体实现：

```
// k8s.io/client-go/tools/cache/shared_informer.go

// 为 listwatcher 创建一个新的实例，用于 Reflector 从 apiserver 获取资源
// 所以需要外部提供一个资源类型
func NewSharedInformer(lw ListerWatcher, exampleObject runtime.Object, defaultEventHandlerResyncPeriod time.Duration) SharedInformer {
  // 调用 NewSharedIndexInformer 实现
  return NewSharedIndexInformer(lw, exampleObject, defaultEventHandlerResyncPeriod, Indexers{})
}

func NewSharedIndexInformer(lw ListerWatcher, exampleObject runtime.Object, defaultEventHandlerResyncPeriod time.Duration, indexers In
  realClock := &clock.RealClock{}
  sharedIndexInformer := &sharedIndexInformer{
    processor:                       &sharedProcessor{clock: realClock},
    indexer:                         NewIndexer(DeletionHandlingMetaNamespaceKeyFunc, indexers),
    listerWatcher:                   lw,
    objectType:                      exampleObject,
    resyncCheckPeriod:               defaultEventHandlerResyncPeriod,
    defaultEventHandlerResyncPeriod: defaultEventHandlerResyncPeriod,
    cacheMutationDetector:           NewCacheMutationDetector(fmt.Sprintf("%T", exampleObject)),
    clock:                           realClock,
  }
  return sharedIndexInformer
}
```

实例化 SharedInformer 比较简单，实例化完成后就可以添加事件处理器了：

```
// k8s.io/client-go/tools/cache/shared_informer.go

// 使用默认的同步周期添加事件处理器
func (s *sharedIndexInformer) AddEventHandler(handler ResourceEventHandler) {
  s.AddEventHandlerWithResyncPeriod(handler, s.defaultEventHandlerResyncPeriod)
```

```
}

// 真正的添加事件处理器的实现
func (s *sharedIndexInformer) AddEventHandlerWithResyncPeriod(handler ResourceEventHandler, resyncPeriod time.Duration) {
  s.startedLock.Lock()
  defer s.startedLock.Unlock()
  // 如果已经结束了，那就直接返回了
  if s.stopped {
    klog.V(2).Infof("Handler %v was not added to shared informer because it has stopped already", handler)
    return
  }
  // 如果同步周期>0
  if resyncPeriod > 0 {
    // 同步周期不能小于最小的时间
    if resyncPeriod < minimumResyncPeriod {
      klog.Warningf("resyncPeriod %d is too small. Changing it to the minimum allowed value of %d", resyncPeriod, minimumResyncPeriod)
      resyncPeriod = minimumResyncPeriod
    }
    //
    if resyncPeriod < s.resyncCheckPeriod {
      // 如果已经启动了，那就用 resyncCheckPeriod 这个周期
      if s.started {
        klog.Warningf("resyncPeriod %d is smaller than resyncCheckPeriod %d and the informer has already started. Changing it to %d",
        resyncPeriod = s.resyncCheckPeriod
      } else {
        // 如果事件处理器的同步周期小于当前的 resyncCheckPeriod 且还没启动
        // 则更新 resyncCheckPeriod 为 resyncPeriod
        // 并相应调整所有监听器的同步周期
        s.resyncCheckPeriod = resyncPeriod
        s.processor.resyncCheckPeriodChanged(resyncPeriod)
      }
    }
  }

  // 新建事件处理器
  listener := newProcessListener(handler, resyncPeriod, determineResyncPeriod(resyncPeriod, s.resyncCheckPeriod), s.clock.Now(), initi

  // 如果没有启动，那么就直接添加处理器就可以了
  if !s.started {
    // 上面我们分析过添加事件处理器
    s.processor.addListener(listener)
    return
  }

  // blockDeltas 提供了一种方法来停止所有的事件分发，以便后面的事件处理器可以安全地加入 SharedInformer。
  s.blockDeltas.Lock()
  defer s.blockDeltas.Unlock()
  // 添加处理器
  s.processor.addListener(listener)

  // 因为到这里证明 SharedInformer 已经启动了，可能很多对象已经让其他处理器处理过了
  // 所以这些对象就不会再通知新添加的处理器了，所以这里遍历 indexer 中的所有对象去通知新添加的处理器
  for _, item := range s.indexer.List() {
    listener.add(addNotification{newObj: item})
  }
}
```

事件处理器添加完过后就要看下 SharedInformer 是如何把事件分发给每个处理器的了：

```
// k8s.io/client-go/tools/cache/shared_informer.go

func (s *sharedIndexInformer) Run(stopCh <-chan struct{}) {
  defer utilruntime.HandleCrash()
  // 新建一个 DeltaFIFO
  fifo := NewDeltaFIFOWithOptions(DeltaFIFOOptions{
    KnownObjects:          s.indexer,
    EmitDeltaTypeReplaced: true,
  })
  // 用于构造 Controller 的配置
  cfg := &Config{
    Queue:            fifo,
    ListerWatcher:    s.listerWatcher,
    ObjectType:       s.objectType,
    FullResyncPeriod: s.resyncCheckPeriod,
    RetryOnError:     false,
    ShouldResync:     s.processor.shouldResync,
    // Controller 调用 DeltaFIFO 的 Pop 函数传入这个回调函数来处理弹出的对象
    Process: s.HandleDeltas,
  }

  func() {
    s.startedLock.Lock()
    defer s.startedLock.Unlock()
    // 新建一个 Controller 并标记为已经启动
```

```
    s.controller = New(cfg)
    s.controller.(*controller).clock = s.clock
    s.started = true
  }()

  processorStopCh := make(chan struct{})
  var wg wait.Group
  defer wg.Wait()              // 等待处理器停止
  defer close(processorStopCh) // 通知处理器停止
  // 启动两个协程
  wg.StartWithChannel(processorStopCh, s.cacheMutationDetector.Run)
  wg.StartWithChannel(processorStopCh, s.processor.run)

  defer func() {
    s.startedLock.Lock()
    defer s.startedLock.Unlock()
    // 标记为已停止
    s.stopped = true
  }()
  // 启动 Controller
  s.controller.Run(stopCh)
}
```

sharedIndexInformer 通过 Run() 函数启动了 Controller 和 sharedProcess，Controller 通过 DeltaFIFO 的 Pop 函数弹出 Deltas 对象，并使用 HandleDeltas 函数来处理这个对象，前面其实我们就讲解过。这个函数把 Deltas 转换为 sharedProcess 需要的各种Notification 类型。

```
// k8s.io/client-go/tools/cache/shared_informer.go

// DeltaFIFO 的对象被 Pop 后的处理函数
func (s *sharedIndexInformer) HandleDeltas(obj interface{}) error {
  s.blockDeltas.Lock()
  defer s.blockDeltas.Unlock()

  // 因为 Deltas 是 Delta 列表，里面包含一个对象的多个操作
  // 所以要从最老的 Delta 到最新的 Delta 遍历处理
  for _, d := range obj.(Deltas) {
    switch d.Type { // 根据对象操作类型进行处理
    // 同步、替换、添加、更新类型
    case Sync, Replaced, Added, Updated:
      s.cacheMutationDetector.AddObject(d.Object)
      // 如果 indexer 中有这个对象，则当成更新事件进行处理
      if old, exists, err := s.indexer.Get(d.Object); err == nil && exists {
        if err := s.indexer.Update(d.Object); err != nil {
          return err
        }

        isSync := false
        switch {
        case d.Type == Sync:
          isSync = true
        case d.Type == Replaced:
          if accessor, err := meta.Accessor(d.Object); err == nil {
            if oldAccessor, err := meta.Accessor(old); err == nil {
              isSync = accessor.GetResourceVersion() == oldAccessor.GetResourceVersion()
            }
          }
        }
        // 通知处理器处理事件
        s.processor.distribute(updateNotification{oldObj: old, newObj: d.Object}, isSync)
      } else {
        // 将对象添加到 indexer 存储中
        if err := s.indexer.Add(d.Object); err != nil {
          return err
        }
        // 然后通知处理器处理事件
        s.processor.distribute(addNotification{newObj: d.Object}, false)
      }
    // 删除类型
    case Deleted:
      // 从 indexer 中删除对象
      if err := s.indexer.Delete(d.Object); err != nil {
        return err
      }
      // 通知所有的处理器对象被删除了
      s.processor.distribute(deleteNotification{oldObj: d.Object}, false)
    }
  }
  return nil
}
```

到这里我们就将整个 SharedInformer 的流程就梳理清楚了，最后我们再来总结下 SharedInformer 的整个流程：

1. 通过 Reflector 实现资源对象的 List 和 Watch 操作
2. 将通过 List 全量列举的对象存储在 Indexer 中，然后再 Watch 资源，一旦有变化就更新 Indexer，并在 Indexer 中采用 Namespace 做对象索引
3. 更新到 Indexer 的过程通过 DeltaFIFO 实现有顺序的更新，因为资源状态是通过全量+增量的方式实现同步的，所以顺序错误会造成状态不一致
4. 使用者可以注册回调函数，在更新到 Indexer 的同时通知使用者去处理，为了保证回调处理不被某一个处理器阻塞，SharedInformer 实现了processorListener 异步缓冲处理
5. 整个过程是通过 Controller 来驱动的