Reflector 源码分析

本文主要对 Informer 中的 Reflector 组件进行分析说明。

前面我们说了 Informer 通过对 APIServer 的资源对象执行 List 和 Watch 操作，把获取到的数据存储在本地的缓存中，其中实现这个的核心功能就是 Reflector，我们可以称其为反射器，从名字我们可以看出来它的主要功能就是反射，就是将 Etcd 里面的数据反射到本地存储（DeltaFIFO）中。Reflector 首先通过 List 操作获取所有的资源对象数据，保存到本地存储，然后通过 Watch 操作监控资源的变化，触发相应的事件处理，比如前面示例中的 Add 事件、Update 事件、Delete 事件。

Reflector 结构体的定义位于 `staging/src/k8s.io/client-go/tools/cache/reflector.go` 下面：

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

// Reflector(反射器) 监听指定的资源，将所有的变化都反射到给定的存储中去
type Reflector struct {
  // name 标识这个反射器的名称，默认为 文件:行数（比如reflector.go:125）
  // 默认名字通过 k8s.io/apimachinery/pkg/util/naming/from_stack.go 下面的 GetNameFromCallsite 函数生成
  name string

  // 期望放到 Store 中的类型名称，如果提供，则是 expectedGVK 的字符串形式
  // 否则就是 expectedType 的字符串，它仅仅用于显示，不用于解析或者比较。
  expectedTypeName string
  // An example object of the type we expect to place in the store.
  // Only the type needs to be right, except that when that is
  // `unstructured.Unstructured` the object's `"apiVersion"` and
  // `"kind"` must also be right.
  // 我们放到 Store 中的对象类型
  expectedType reflect.Type
  // 如果是非结构化的，我们期望放在 Sotre 中的对象的 GVK
  expectedGVK *schema.GroupVersionKind
  // 与 watch 源同步的目标 Store
  store Store
  // 用来执行 lists 和 watches 操作的 listerWatcher 接口（最重要的）
  listerWatcher ListerWatcher

  // backoff manages backoff of ListWatch
  backoffManager wait.BackoffManager

  resyncPeriod time.Duration
  // ShouldResync 会周期性的被调用，当返回 true 的时候，就会调用 Store 的 Resync 操作
  ShouldResync func() bool
  // clock allows tests to manipulate time
  clock clock.Clock
  // paginatedResult defines whether pagination should be forced for list calls.
  // It is set based on the result of the initial list call.
  paginatedResult bool
  // Kubernetes 资源在 APIServer 中都是有版本的，对象的任何修改(添加、删除、更新)都会造成资源版本更新，lastSyncResourceVersion 就是指的这个版本
  lastSyncResourceVersion string
  // 如果之前的 list 或 watch 带有 lastSyncResourceVersion 的请求中是一个 HTTP 410（Gone）的失败请求，则 isLastSyncResourceVersionGone 为 true
  isLastSyncResourceVersionGone bool
  // lastSyncResourceVersionMutex 用于保证对 lastSyncResourceVersion 的读/写访问。
  lastSyncResourceVersionMutex sync.RWMutex
  // WatchListPageSize is the requested chunk size of initial and resync watch lists.
  // If unset, for consistent reads (RV="") or reads that opt-into arbitrarily old data
  // (RV="0") it will default to pager.PageSize, for the rest (RV != "" && RV != "0")
  // it will turn off pagination to allow serving them from watch cache.
  // NOTE: It should be used carefully as paginated lists are always served directly from
  // etcd, which is significantly less efficient and may lead to serious performance and
  // scalability problems.
  WatchListPageSize int64
}

// NewReflector 创建一个新的反射器对象，将使给定的 Store 保持与服务器中指定的资源对象的内容同步。
// 反射器只把具有 expectedType 类型的对象放到 Store 中，除非 expectedType 是 nil。
// 如果 resyncPeriod 是非0，那么反射器会周期性地检查 ShouldResync 函数来决定是否调用 Store 的 Resync 操作
// `ShouldResync==nil` 意味着总是要执行 Resync 操作。
// 这使得你可以使用反射器周期性地处理所有的全量和增量的对象。
func NewReflector(lw ListerWatcher, expectedType interface{}, store Store, resyncPeriod time.Duration) *Reflector {
  // 默认的反射器名称为 file:line
  return NewNamedReflector(naming.GetNameFromCallsite(internalPackages...), lw, expectedType, store, resyncPeriod)
}

// NewNamedReflector 与 NewReflector 一样，只是指定了一个 name 用于日志记录
```

```
func NewNamedReflector(name string, lw ListerWatcher, expectedType interface{}, store Store, resyncPeriod time.Duration) *Reflector {
  realClock := &clock.RealClock{}
  r := &Reflector{
    name:          name,
    listerWatcher: lw,
    store:         store,
    backoffManager: wait.NewExponentialBackoffManager(800*time.Millisecond, 30*time.Second, 2*time.Minute, 2.0, 1.0, realClock),
    resyncPeriod:   resyncPeriod,
    clock:          realClock,
  }
  r.setExpectedType(expectedType)
  return r
}
```

从源码中我们可以看出来通过 `NewReflector` 实例化反射器的时候，必须传入一个 `ListerWatcher` 接口对象，这个也是反射器最核心的功能，该接口拥有 List 和 Watch 方法，用于获取和监控资源对象。

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

// Lister 是知道如何执行初始化List列表的对象
type Lister interface {
  // List 应该返回一个列表类型的对象；
  // Items 字段将被解析，ResourceVersion 字段将被用于正确启动 watch 的地方
  List(options metav1.ListOptions) (runtime.Object, error)
}

// Watcher 是知道如何执行 watch 操作的任何对象
type Watcher interface {
  // Watch 在指定的版本开始执行 watch 操作
  Watch(options metav1.ListOptions) (watch.Interface, error)
}

// ListerWatcher 是任何知道如何对一个资源执行初始化List列表和启动Watch监控操作的对象
type ListerWatcher interface {
  Lister
  Watcher
}
```

而 Reflector 对象通过 Run 函数来启动监控并处理监控事件的：

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

// Run 函数反复使用反射器的 ListAndWatch 函数来获取所有对象和后续的 deltas。
// 当 stopCh 被关闭的时候，Run函数才会退出。
func (r *Reflector) Run(stopCh <-chan struct{}) {
  klog.V(2).Infof("Starting reflector %s (%s) from %s", r.expectedTypeName, r.resyncPeriod, r.name)
  wait.BackoffUntil(func() {
    if err := r.ListAndWatch(stopCh); err != nil {
      utilruntime.HandleError(err)
    }
  }, r.backoffManager, true, stopCh)
  klog.V(2).Infof("Stopping reflector %s (%s) from %s", r.expectedTypeName, r.resyncPeriod, r.name)
}
```

所以不管我们传入的 ListWatcher 对象是如何实现的 List 和 Watch 操作，只要实现了就可以，最主要的还是看 ListAndWatch 函数是如何去实现的，如何去调用 List 和 Watch 的：

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

// ListAndWatch 函数首先列出所有的对象，并在调用的时候获得资源版本，然后使用该资源版本来进行 watch 操作。
// 如果 ListAndWatch 没有初始化 watch 成功就会返回错误。
func (r *Reflector) ListAndWatch(stopCh <-chan struct{}) error {
  klog.V(3).Infof("Listing and watching %v from %s", r.expectedTypeName, r.name)
  var resourceVersion string

  options := metav1.ListOptions{ResourceVersion: r.relistResourceVersion()}

  if err := func() error {
    initTrace := trace.New("Reflector ListAndWatch", trace.Field{"name", r.name})
    defer initTrace.LogIfLong(10 * time.Second)
    var list runtime.Object
    var paginatedResult bool
    var err error
    listCh := make(chan struct{}, 1)
    panicCh := make(chan interface{}, 1)
    go func() {
      defer func() {
```

```
        if r := recover(); r != nil {
          panicCh <- r
        }
      }()
      // 如果 listWatcher 支持，会尝试 chunks（分块）收集 List 列表数据
      // 如果不支持，第一个 List 列表请求将返回完整的响应数据。
      pager := pager.New(pager.SimplePageFunc(func(opts metav1.ListOptions) (runtime.Object, error) {
        return r.listerWatcher.List(opts)
      }))
      switch {
      case r.WatchListPageSize != 0:
        pager.PageSize = r.WatchListPageSize
      case r.paginatedResult:
        // 获得一个初始的分页结果。假定此资源和服务器符合分页请求，并保留默认的分页器大小设置
      case options.ResourceVersion != "" && options.ResourceVersion != "0":
        pager.PageSize = 0
      }

      list, paginatedResult, err = pager.List(context.Background(), options)
      if isExpiredError(err) {
        r.setIsLastSyncResourceVersionExpired(true)
        list, paginatedResult, err = pager.List(context.Background(), metav1.ListOptions{ResourceVersion: r.relistResourceVersion()})
      }
      close(listCh)
    }()
    select {
    case <-stopCh:
      return nil
    case r := <-panicCh:
      panic(r)
    case <-listCh:
    }
    if err != nil {
      return fmt.Errorf("%s: Failed to list %v: %v", r.name, r.expectedTypeName, err)
    }

    if options.ResourceVersion == "0" && paginatedResult {
      r.paginatedResult = true
    }

    r.setIsLastSyncResourceVersionExpired(false) // list 成功
    initTrace.Step("Objects listed")
    listMetaInterface, err := meta.ListAccessor(list)
    if err != nil {
      return fmt.Errorf("%s: Unable to understand list result %#v: %v", r.name, list, err)
    }
    // 获取资源版本号
    resourceVersion = listMetaInterface.GetResourceVersion()
    initTrace.Step("Resource version extracted")
    // 将资源数据转换成资源对象列表，将 runtime.Object 对象转换成 []runtime.Object 对象
    items, err := meta.ExtractList(list)
    if err != nil {
      return fmt.Errorf("%s: Unable to understand list result %#v (%v)", r.name, list, err)
    }
    initTrace.Step("Objects extracted")
    // 将资源对象列表中的资源对象和资源版本号存储在 Store 中
    if err := r.syncWith(items, resourceVersion); err != nil {
      return fmt.Errorf("%s: Unable to sync list result: %v", r.name, err)
    }
    initTrace.Step("SyncWith done")
    r.setLastSyncResourceVersion(resourceVersion)
    initTrace.Step("Resource version updated")
    return nil
  }(); err != nil {
    return err
  }

  resyncerrc := make(chan error, 1)
  cancelCh := make(chan struct{})
  defer close(cancelCh)
  go func() {
    resyncCh, cleanup := r.resyncChan()
    defer func() {
      cleanup()
    }()
    for {
      select {
      case <-resyncCh:
      case <-stopCh:
        return
      case <-cancelCh:
        return
      }
      // 如果 ShouldResync 为 nil 或者调用返回true，则执行 Store 的 Resync 操作
      if r.ShouldResync == nil || r.ShouldResync() {
        klog.V(4).Infof("%s: forcing resync", r.name)
        if err := r.store.Resync(); err != nil {
```

```
          resyncerrc <- err
          return
        }
      }
      cleanup()
      resyncCh, cleanup = r.resyncChan()
    }
  }()

  for {
    // stopCh 一个停止循环的机会
    select {
    case <-stopCh:
      return nil
    default:
    }

    timeoutSeconds := int64(minWatchTimeout.Seconds() * (rand.Float64() + 1.0))
    // 设置watch的选项，因为前期列举了全量对象，从这里只要监听最新版本以后的资源就可以了
    // 如果没有资源变化总不能一直挂着吧？也不知道是卡死了还是怎么了，所以设置一个超时会好一点
    options = metav1.ListOptions{
      ResourceVersion: resourceVersion,
      TimeoutSeconds: &timeoutSeconds,
      AllowWatchBookmarks: true,
    }

    start := r.clock.Now()
    // 执行 Watch 操作
    w, err := r.listerWatcher.Watch(options)
    if err != nil {
      switch {
      case isExpiredError(err):
        klog.V(4).Infof("%s: watch of %v closed with: %v", r.name, r.expectedTypeName, err)
      case err == io.EOF:
        // watch closed normally
      case err == io.ErrUnexpectedEOF:
        klog.V(1).Infof("%s: Watch for %v closed with unexpected EOF: %v", r.name, r.expectedTypeName, err)
      default:
        utilruntime.HandleError(fmt.Errorf("%s: Failed to watch %v: %v", r.name, r.expectedTypeName, err))
      }
      if utilnet.IsConnectionRefused(err) {
        time.Sleep(time.Second)
        continue
      }
      return nil
    }
    // 调用 watchHandler 来处理分发 watch 到的事件对象
    if err := r.watchHandler(start, w, &resourceVersion, resyncerrc, stopCh); err != nil {
      if err != errorStopRequested {
        switch {
        case isExpiredError(err):
          klog.V(4).Infof("%s: watch of %v closed with: %v", r.name, r.expectedTypeName, err)
        default:
          klog.Warningf("%s: watch of %v ended with: %v", r.name, r.expectedTypeName, err)
        }
      }
      return nil
    }
  }
}
```

首先通过反射器的 `relistResourceVersion` 函数获得反射器 relist 的资源版本，如果资源版本非 0，则表示根据资源版本号继续获取，当传输过程中遇到网络故障或者其他原因导致中断，下次再连接时，会根据资源版本号继续传输未完成的部分。可以使本地缓存中的数据与Etcd集群中的数据保持一致，该函数实现如下所示：

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

// relistResourceVersion 决定了反射器应该list或者relist的资源版本。
// 如果最后一次relist的结果是HTTP 410 (Gone) 状态码，则返回""，这样relist将通过quorum读取etcd中可用的最新资源版本。
// 返回使用 lastSyncResourceVersion，这样反射器就不会使用在relist结果或watch事件中watch到的资源版本更老的资源版本进行relist了
func (r *Reflector) relistResourceVersion() string {
  r.lastSyncResourceVersionMutex.RLock()
  defer r.lastSyncResourceVersionMutex.RUnlock()

  if r.isLastSyncResourceVersionGone {
    // 因为反射器会进行分页List请求，如果 lastSyncResourceVersion 过期了，所有的分页列表请求就都会跳过 watch 缓存
    // 所以设置 ResourceVersion="", 然后再次 List，重新建立反射器到最新的可用资源版本，从 etcd 中读取，保持一致性。
    return ""
  }
  if r.lastSyncResourceVersion == "" {
    // 反射器执行的初始 List 操作的时候使用0作为资源版本。
    return "0"
  }
```

```
  return r.lastSyncResourceVersion
}
```

上面的 ListAndWatch 函数实现看上去虽然非常复杂，但其实大部分是对分页的各种情况进行处理，最核心的还是调用 `r.listerWatcher.List(opts)` 获取全量的资源对象，而这个 List 其实 ListerWatcher 实现的 List 方法，这个 ListerWatcher 接口实际上在该接口定义的同一个文件中就有一个 ListWatch 结构体实现了：

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

// ListFunc 知道如何 List 资源
type ListFunc func(options metav1.ListOptions) (runtime.Object, error)

// WatchFunc 知道如何 watch 资源
type WatchFunc func(options metav1.ListOptions) (watch.Interface, error)

// ListWatch 结构体知道如何 list 和 watch 资源对象，它实现了 ListerWatcher 接口。
// 它为 NewReflector 使用者提供了方便的函数。其中 ListFunc 和 WatchFunc 不能为 nil。
type ListWatch struct {
  ListFunc  ListFunc
  WatchFunc WatchFunc
  // DisableChunking 对 list watcher 请求不分块。
  DisableChunking bool
}

// 列出一组 APIServer 资源
func (lw *ListWatch) List(options metav1.ListOptions) (runtime.Object, error) {
  return lw.ListFunc(options)
}

// Watch 一组 APIServer 资源
func (lw *ListWatch) Watch(options metav1.ListOptions) (watch.Interface, error) {
  return lw.WatchFunc(options)
}
```

当我们真正使用一个 Informer 对象的时候，实例化的时候就会调用这里的 ListWatch 来进行初始化，比如前面我们实例中使用的 Deployment Informer，

```
// k8s.io/client-go/informers/apps/v1/deployment.go

// NewFilteredDeploymentInformer 为 Deployment 构造一个新的 Informer。
// 总是倾向于使用一个 informer 工厂来获取一个 shared informer，而不是获取一个独立的 informer，这样可以减少内存占用和服务器的连接数。
func NewFilteredDeploymentInformer(client kubernetes.Interface, namespace string, resyncPeriod time.Duration, indexers cache.Indexers,
  return cache.NewSharedIndexInformer(
    &cache.ListWatch{
      ListFunc: func(options metav1.ListOptions) (runtime.Object, error) {
        if tweakListOptions != nil {
          tweakListOptions(&options)
        }
        return client.AppsV1().Deployments(namespace).List(context.TODO(), options)
      },
      WatchFunc: func(options metav1.ListOptions) (watch.Interface, error) {
        if tweakListOptions != nil {
          tweakListOptions(&options)
        }
        return client.AppsV1().Deployments(namespace).Watch(context.TODO(), options)
      },
    },
    &appsv1.Deployment{},
    resyncPeriod,
    indexers,
  )
}

func (f *deploymentInformer) defaultInformer(client kubernetes.Interface, resyncPeriod time.Duration) cache.SharedIndexInformer {
  return NewFilteredDeploymentInformer(client, f.namespace, resyncPeriod, cache.Indexers{cache.NamespaceIndex: cache.MetaNamespaceInde
}

func (f *deploymentInformer) Informer() cache.SharedIndexInformer {
  return f.factory.InformerFor(&appsv1.Deployment{}, f.defaultInformer)
}
```

从上面代码我们就可以看出来当我们去调用一个资源对象的 Informer() 的时候，就会去调用上面的 `NewFilteredDeploymentInformer` 函数进行初始化，而在初始化的使用就传入了 `cache.ListWatch` 对象，其中就有 List 和 Watch 的实现操作，也就是说前面反射器在 ListAndWatch 里面调用的 ListWatcher 的 List 操作是在一个具体的资源对象的 Informer 中实现的，比如我们这里就是通过的 ClientSet 客户端与 APIServer 交互获取到 Deployment 的资源列表数据的，通过在 ListFunc 中的 `client.AppsV1().Deployments(namespace).List(context.TODO(), options)` 实现的，这下应该好理解了吧。

获取到了全量的 List 数据过后，通过 `listMetaInterface.GetResourceVersion()` 来获取资源的版本号，ResourceVersion（资源版本号）非常重要，Kubernetes 中所有的资源都拥有该字段，它标识当前资源对象的版本号，每次修改（CUD）当前资源对象时，Kubernetes API Server 都会更改 ResourceVersion，这样 client-go 执行 Watch 操作时可以根据ResourceVersion 来确定当前资源对象是否发生了变化。

然后通过 meta.ExtractList 函数将资源数据转换成资源对象列表，将 `runtime.Object` 对象转换成 `[]runtime.Object` 对象，因为全量获取的是一个资源列表。

接下来是通过反射器的 `syncWith` 函数将资源对象列表中的资源对象和资源版本号存储在 Store 中，这个会在后面的章节中详细说明。

最后处理完成后通过 `r.setLastSyncResourceVersion(resourceVersion)` 操作来设置最新的资源版本号，其他的就是启动一个 goroutine 去定期检查是否需要执行 Resync 操作，调用存储中的 `r.store.Resync()` 来执行，关于存储的实现在后面和大家进行讲解。

紧接着就是 Watch 操作了，Watch 操作通过 HTTP 协议与 APIServer 建立长连接，接收Kubernetes API Server 发来的资源变更事件，和 List 操作一样，Watch 的真正实现也是具体的 Informer 初始化的时候传入的，比如上面的 Deployment Informer 中初始化的时候传入的 WatchFunc，底层也是通过 ClientSet 客户端对 Deployment 执行 Watch 操作 `client.AppsV1().Deployments(namespace).Watch(context.TODO(), options)` 实现的。

获得 watch 的资源数据后，通过调用 `r.watchHandler` 来处理资源的变更事件，当触发Add 事件、Update 事件、Delete 事件时，将对应的资源对象更新到本地缓存（DeltaFIFO）中并更新 ResourceVersion 资源版本号。

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

// watchHandler 监听 w 保持资源版本最新
func (r *Reflector) watchHandler(start time.Time, w watch.Interface, resourceVersion *string, errc chan error, stopCh <-chan struct{})
  eventCount := 0

  defer w.Stop()

loop:
  for {
    select {
    case <-stopCh:
      return errorStopRequested
    case err := <-errc:
      return err
    case event, ok := <-w.ResultChan():  // 从 watch 中获取事件对象
      if !ok {
        break loop
      }
      if event.Type == watch.Error {
        return apierrors.FromObject(event.Object)
      }
      if r.expectedType != nil {
        if e, a := r.expectedType, reflect.TypeOf(event.Object); e != a {
          utilruntime.HandleError(fmt.Errorf("%s: expected type %v, but watch event object had type %v", r.name, e, a))
          continue
        }
      }
      if r.expectedGVK != nil {
        if e, a := *r.expectedGVK, event.Object.GetObjectKind().GroupVersionKind(); e != a {
          utilruntime.HandleError(fmt.Errorf("%s: expected gvk %v, but watch event object had gvk %v", r.name, e, a))
          continue
        }
      }
      meta, err := meta.Accessor(event.Object)
      if err != nil {
        utilruntime.HandleError(fmt.Errorf("%s: unable to understand watch event %#v", r.name, event))
        continue
      }
      // 获得当前 watch 到资源的资源版本号
      newResourceVersion := meta.GetResourceVersion()
      switch event.Type {  // 分发事件
      case watch.Added:  //  Add 事件
        err := r.store.Add(event.Object)
        if err != nil {
          utilruntime.HandleError(fmt.Errorf("%s: unable to add watch event object (%#v) to store: %v", r.name, event.Object, err))
        }
      case watch.Modified:  // Update 事件
        err := r.store.Update(event.Object)
        if err != nil {
          utilruntime.HandleError(fmt.Errorf("%s: unable to update watch event object (%#v) to store: %v", r.name, event.Object, err))
        }
      case watch.Deleted:  // Delete 事件
        err := r.store.Delete(event.Object)
        if err != nil {
          utilruntime.HandleError(fmt.Errorf("%s: unable to delete watch event object (%#v) from store: %v", r.name, event.Object, err
        }
```

```
      case watch.Bookmark:
        // `Bookmark` 意味着 watch 已经同步到这里了，只要更新资源版本即可。
      default:
        utilruntime.HandleError(fmt.Errorf("%s: unable to understand watch event %#v", r.name, event))
      }
      // 更新资源版本号
      *resourceVersion = newResourceVersion
      r.setLastSyncResourceVersion(newResourceVersion)
      eventCount++
    }
  }

  watchDuration := r.clock.Since(start)
  if watchDuration < 1*time.Second && eventCount == 0 {
    return fmt.Errorf("very short watch: %s: Unexpected watch close - watch lasted less than a second and no items received", r.name)
  }
  klog.V(4).Infof("%s: Watch close - %v total %v items received", r.name, r.expectedTypeName, eventCount)
  return nil
}
```

这就是 Reflector 反射器中最核心的 `ListAndWatch` 实现，从上面的实现我们可以看出获取到的数据最终都流向了本地的 Store，也就是 DeltaFIFO，所以接下来我们需要来分析 DeltaFIFO 的实现。