# controller-runtime 原理之控制器

本节主要介绍 controller-runtime 框架的基本使用与原理。



controller-runtime(https://github.com/kubernetes-sigs/controller-runtime) 框架实际上是社区帮我们封装的一个控制器处理的框架，底层核心实现原理和我们前面去自定义一个 controller 控制器逻辑是一样的，只是在这个基础上新增了一些概念，开发者直接使用这个框架去开发控制器会更加简单方便。包括 kubebuilder、operator-sdk 这些框架其实都是在 controller-runtime 基础上做了一层封装，方便开发者快速生成项目的脚手架而已。下面我们就来分析下 controller-runtime 是如何实现的控制器处理。

## Controller 的实现

首先我们还是去查看下控制器的定义以及控制器是如何启动的。控制器的定义结构体如下所示：

```
// pkg/internal/controller/controller.go

type Controller struct {
  // Name 用于跟踪、记录和监控中控制器的唯一标识，必填字段
  Name string

  // 可以运行的最大并发 Reconciles 数量，默认值为1
  MaxConcurrentReconciles int

  // Reconciler 是一个可以随时调用对象的 Name/Namespace 的函数
  // 确保系统的状态与对象中指定的状态一致，默认为 DefaultReconcileFunc 函数
  Do reconcile.Reconciler

  // 一旦控制器准备好启动，MakeQueue 就会为这个控制器构造工作队列
  MakeQueue func() workqueue.RateLimitingInterface

  // 队列通过监听来自 Infomer 的事件，添加对象键到队列中进行处理
  // MakeQueue 属性就是来构造这个工作队列的
  // 也就是前面我们讲解的工作队列，我们将通过这个工作队列来进行消费
  Queue workqueue.RateLimitingInterface

  // SetFields 用来将依赖关系注入到其他对象，比如 Sources、EventHandlers 以及 Predicates
  SetFields func(i interface{}) error

  // 控制器同步信号量
```

```
  mu sync.Mutex

  // 允许测试减少 JitterPeriod，使其更快完成
  JitterPeriod time.Duration

  // 控制器是否已经启动
  Started bool

  // TODO(community): Consider initializing a logger with the Controller Name as the tag

  // startWatches 维护了一个 sources-handlers 以及 predicates 列表以方便在控制器启动的时候启动
  startWatches []watchDescription

  // 日志记录
  Log logr.Logger
}
```

上面的结构体就是 controller-runtime 中定义的控制器结构体，我们可以看到结构体中仍然有一个限速的工作队列，但是看上去没有资源对象的 Informer 或者 Indexer 的数据，实际上这里是通过下面的 startWatches 属性做了一层封装，该属性是一个 watchDescription 队列，一个 watchDescription 包含了所有需要 watch 的信息：

```
// pkg/internal/controller/controller.go

// watchDescription 包含所有启动 watch 操作所需的信息
type watchDescription struct {
  src        source.Source
  handler    handler.EventHandler
  predicates []predicate.Predicate
}
```

整个控制器中最重要的两个函数是 Watch 与 Start，下面我们就来分析下他们是如何实现的。

## Watch 函数实现

```
// pkg/internal/controller/controller.go

func (c *Controller) Watch(src source.Source, evthdler handler.EventHandler, prct ...predicate.Predicate) error {
  c.mu.Lock()
  defer c.mu.Unlock()

  // 注入 Cache 到参数中
  if err := c.SetFields(src); err != nil {
    return err
  }
  if err := c.SetFields(evthdler); err != nil {
    return err
  }
  for _, pr := range prct {
    if err := c.SetFields(pr); err != nil {
      return err
    }
  }

  // Controller 还没启动，把 watches 存放到本地然后返回
  //
  // 这些 watches 会被保存到控制器结构体中，直到调用 Start(...) 函数
  if !c.Started {
    c.startWatches = append(c.startWatches, watchDescription{src: src, handler: evthdler, predicates: prct})
    return nil
  }

  c.Log.Info("Starting EventSource", "source", src)
  // 调用 src 的 Start 函数
```

```
  return src.Start(evthdler, c.Queue, prct...)
}
```

上面的 Watch 函数可以看到最终是去调用的 Source 这个参数的 Start 函数，Source 是事件的源，如对资源对象的 Create、Update、Delete 操作，需要由 `event.EventHandlers` 将 `reconcile.Requests` 入队列进行处理。

- 使用 Kind 来处理来自集群的事件（如 Pod 创建、Pod 更新、Deployment 更新）。
- 使用 Channel 来处理来自集群外部的事件（如 GitHub Webhook 回调、轮询外部 URL）。

```
// pkg/source/source.go

type Source interface {
  // Start 是一个内部函数
  // 只应该由 Controller 调用，向 Informer 注册一个 EventHandler
  // 将 reconcile.Request 放入队列
  Start(handler.EventHandler, workqueue.RateLimitingInterface, ...predicate.Predicate) error
}
```

我们可以看到 `source.Source` 是一个接口，它是 `Controller.Watch` 的一个参数，所以要看具体的如何实现的 Source.Start 函数，我们需要去看传入 `Controller.Watch` 的参数，在 controller-runtime 中调用控制器的 Watch 函数的入口实际上位于 `pkg/builder/controller.go` 文件中的 `doWatch()` 函数：

```
// pkg/builder/controller.go

func (blder *Builder) doWatch() error {
  // Reconcile type
  src := &source.Kind{Type: blder.forInput.object}
  hdler := &handler.EnqueueRequestForObject{}
  allPredicates := append(blder.globalPredicates, blder.forInput.predicates...)
  err := blder.ctrl.Watch(src, hdler, allPredicates...)
  if err != nil {
    return err
  }
  ......
  return nil
}
```

可以看到 Watch 的第一个参数是一个 `source.Kind` 的类型，该结构体就实现了上面的 `source.Source` 接口：

```
// pkg/source/source.go

// Kind 用于提供来自集群内部的事件源，这些事件来自于 Watches（例如 Pod Create 事件）
type Kind struct {
  // Type 是 watch 对象的类型，比如 &v1.Pod{}
  Type runtime.Object

  // cache 用于 watch 的 APIs 接口
  cache cache.Cache
}

// 真正的 Start 函数实现
func (ks *Kind) Start(handler handler.EventHandler, queue workqueue.RateLimitingInterface,
  prct ...predicate.Predicate) error {

  // Type 在使用之前必须提前指定
  if ks.Type == nil {
    return fmt.Errorf("must specify Kind.Type")
  }

  // cache 也应该在调用 Start 之前被注入了
  if ks.cache == nil {
```

```
    return fmt.Errorf("must call CacheInto on Kind before calling Start")
  }

  // 从 Cache 中获取 Informer
  // 并添加一个事件处理程序来添加队列
  i, err := ks.cache.GetInformer(context.TODO(), ks.Type)
  if err != nil {
    if kindMatchErr, ok := err.(*meta.NoKindMatchError); ok {
      log.Error(err, "if kind is a CRD, it should be installed before calling Start",
        "kind", kindMatchErr.GroupKind)
    }
    return err
  }
  i.AddEventHandler(internal.EventHandler{Queue: queue, EventHandler: handler, Predicates: prct})
  return nil
}
```

从上面的具体实现我们就可以看出来 `Controller.Watch` 函数就是实现的获取资源对象的 Informer 以及注册事件监听函数。Informer 是通过 cache 获取的，cache 是在调用 Start 函数之前注入进来的，这里其实我们不用太关心；下面的 AddEventHandler 函数中是一个 `internal.EventHandler` 结构体，那这个结构体比如会实现 client-go 中提供的 `ResourceEventHandler` 接口，也就是我们熟悉的 OnAdd、OnUpdate、OnDelete 几个函数：

```
// pkg/source/internal/eventsource.go

// EventHandler 实现了 cache.ResourceEventHandler 接口
type EventHandler struct {
  EventHandler handler.EventHandler
  Queue        workqueue.RateLimitingInterface
  Predicates   []predicate.Predicate
}

func (e EventHandler) OnAdd(obj interface{}) {
  // kubernetes 对象被创建的事件
  c := event.CreateEvent{}

  // 获取对象 metav1.Object
  if o, err := meta.Accessor(obj); err == nil {
    c.Meta = o
  } else {
    log.Error(err, "OnAdd missing Meta",
      "object", obj, "type", fmt.Sprintf("%T", obj))
    return
  }

  // 断言 runtime.Object
  if o, ok := obj.(runtime.Object); ok {
    c.Object = o
  } else {
    log.Error(nil, "OnAdd missing runtime.Object",
      "object", obj, "type", fmt.Sprintf("%T", obj))
    return
  }

  // Predicates 用于事件过滤，循环调用 Predicates 的 Create 函数
  for _, p := range e.Predicates {
    if !p.Create(c) {
      return
    }
  }

  // 调用 EventHander 的 Create 函数
  e.EventHandler.Create(c, e.Queue)
}

func (e EventHandler) OnUpdate(oldObj, newObj interface{}) {
  u := event.UpdateEvent{}

  // Pull metav1.Object out of the object
```

```
  if o, err := meta.Accessor(oldObj); err == nil {
    u.MetaOld = o
  } else {
    log.Error(err, "OnUpdate missing MetaOld",
      "object", oldObj, "type", fmt.Sprintf("%T", oldObj))
    return
  }

  // Pull the runtime.Object out of the object
  if o, ok := oldObj.(runtime.Object); ok {
    u.ObjectOld = o
  } else {
    log.Error(nil, "OnUpdate missing ObjectOld",
      "object", oldObj, "type", fmt.Sprintf("%T", oldObj))
    return
  }

  // Pull metav1.Object out of the object
  if o, err := meta.Accessor(newObj); err == nil {
    u.MetaNew = o
  } else {
    log.Error(err, "OnUpdate missing MetaNew",
      "object", newObj, "type", fmt.Sprintf("%T", newObj))
    return
  }

  // Pull the runtime.Object out of the object
  if o, ok := newObj.(runtime.Object); ok {
    u.ObjectNew = o
  } else {
    log.Error(nil, "OnUpdate missing ObjectNew",
      "object", oldObj, "type", fmt.Sprintf("%T", oldObj))
    return
  }

  for _, p := range e.Predicates {
    if !p.Update(u) {
      return
    }
  }

  // 调用 EventHandler 的 Update 函数
  e.EventHandler.Update(u, e.Queue)
}

func (e EventHandler) OnDelete(obj interface{}) {
  d := event.DeleteEvent{}

  // Deal with tombstone events by pulling the object out.  Tombstone events wrap the object in a
  // DeleteFinalStateUnknown struct, so the object needs to be pulled out.
  // Copied from sample-controller
  // This should never happen if we aren't missing events, which we have concluded that we are not
  // and made decisions off of this belief.  Maybe this shouldn't be here?
  var ok bool
  if _, ok = obj.(metav1.Object); !ok {
    // 假设对象没有 Metadata，假设是一个 DeletedFinalStateUnknown 类型的对象
    tombstone, ok := obj.(cache.DeletedFinalStateUnknown)
    if !ok {
      log.Error(nil, "Error decoding objects.  Expected cache.DeletedFinalStateUnknown",
        "type", fmt.Sprintf("%T", obj),
        "object", obj)
      return
    }

    // Set obj to the tombstone obj
    obj = tombstone.Obj
  }

  // Pull metav1.Object out of the object
  if o, err := meta.Accessor(obj); err == nil {
    d.Meta = o
  } else {
    log.Error(err, "OnDelete missing Meta",
```

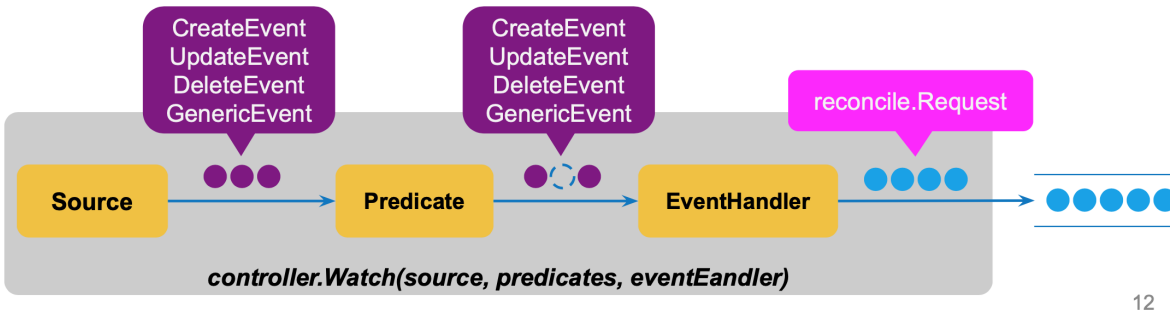
```
      "object", obj, "type", fmt.Sprintf("%T", obj))
    return
  }

  // Pull the runtime.Object out of the object
  if o, ok := obj.(runtime.Object); ok {
    d.Object = o
  } else {
    log.Error(nil, "OnDelete missing runtime.Object",
      "object", obj, "type", fmt.Sprintf("%T", obj))
    return
  }

  for _, p := range e.Predicates {
    if !p.Delete(d) {
      return
    }
  }

  // 调用 EventHandler 的 delete 函数
  e.EventHandler.Delete(d, e.Queue)
}
```

上面的 EventHandler 结构体实现了 client-go 中的 `ResourceEventHandler` 接口，实现过程中我们可以看到调用了 Predicates 进行事件过滤，过滤后才是真正的事件处理，不过其实真正的事件处理也不是在这里去实现的，而是通过 `Controller.Watch` 函数传递进来的 `handler.EventHandler` 处理的，这个函数通过前面的 doWatch() 函数可以看出来它是一个 `&handler.EnqueueRequestForObject{}` 对象，所以真正的事件处理逻辑是这个函数去实现的：

```
// pkg/handler/enqueue.go

// EnqueueRequestForObject 是一个包含了作为事件源的对象的 Name 和 Namespace 的入队列的 Request
// (例如，created/deleted/updated 对象的 Name 和 Namespace)
// handler.EnqueueRequestForObject 几乎被所有关联资源（如 CRD）的控制器使用，以协调关联的资源
type EnqueueRequestForObject struct{}

// Create 函数实现
func (e *EnqueueRequestForObject) Create(evt event.CreateEvent, q workqueue.RateLimitingInterface) {
  if evt.Meta == nil {
    enqueueLog.Error(nil, "CreateEvent received with no metadata", "event", evt)
    return
  }
  // 添加一个 Request 对象到工作队列
  q.Add(reconcile.Request{NamespacedName: types.NamespacedName{
    Name:      evt.Meta.GetName(),
    Namespace: evt.Meta.GetNamespace(),
  }})
}

// Update 函数实现
func (e *EnqueueRequestForObject) Update(evt event.UpdateEvent, q workqueue.RateLimitingInterface) {
  if evt.MetaOld != nil {
    // 如果旧的meta对象不为空，添加到工作队列中
    q.Add(reconcile.Request{NamespacedName: types.NamespacedName{
      Name:      evt.MetaOld.GetName(),
      Namespace: evt.MetaOld.GetNamespace(),
    }})
  } else {
    enqueueLog.Error(nil, "UpdateEvent received with no old metadata", "event", evt)
  }

  if evt.MetaNew != nil {
    // 如果新的meta对象不为空，添加到工作队列中
    q.Add(reconcile.Request{NamespacedName: types.NamespacedName{
      Name:      evt.MetaNew.GetName(),
      Namespace: evt.MetaNew.GetNamespace(),
    }})
  } else {
    enqueueLog.Error(nil, "UpdateEvent received with no new metadata", "event", evt)
```

```
  }
}

// Delete 函数实现
func (e *EnqueueRequestForObject) Delete(evt event.DeleteEvent, q workqueue.RateLimitingInterface) {
  if evt.Meta == nil {
    enqueueLog.Error(nil, "DeleteEvent received with no metadata", "event", evt)
    return
  }
  // 因为前面关于对象的删除状态已经处理了，所以这里直接放入队列中即可
  q.Add(reconcile.Request{NamespacedName: types.NamespacedName{
    Name:      evt.Meta.GetName(),
    Namespace: evt.Meta.GetNamespace(),
  }})
}
```

通过 `EnqueueRequestForObject` 的 Create/Update/Delete 实现可以看出我们放入到工作队列中的元素不是以前默认的元素唯一的 KEY，而是经过封装的 `reconcile.Request` 对象，当然通过这个对象也可以很方便获取对象的唯一标识 KEY。




12

总结起来就是 `Controller.Watch` 函数就是来实现之前自定义控制器中的 Informer 初始化以及事件监听函数的注册。

## Start 函数实现

上面我们分析了控制器的 Watch 函数的实现，下面我们来分析另外一个重要的函数 `Controller.Start` 函数的实现。

```
// pkg/internal/controller/controller.go

func (c *Controller) Start(stop <-chan struct{}) error {
  c.mu.Lock()

  // 调用 MakeQueue() 函数生成工作队列
  c.Queue = c.MakeQueue()
  // 函数退出后关闭队列
  defer c.Queue.ShutDown()

  err := func() error {
    defer c.mu.Unlock()

    // TODO(pwittrock): Reconsider HandleCrash
    defer utilruntime.HandleCrash()

    // NB(directxman12): 在试图等待缓存同步之前启动 sources
    // 这样他们有机会注册他们的目标缓存
    for _, watch := range c.startWatches {
      c.Log.Info("Starting EventSource", "source", watch.src)
      if err := watch.src.Start(watch.handler, c.Queue, watch.predicates...); err != nil {
        return err
      }
    }
```

```
    // 启动 SharedIndexInformer 工厂，开始填充 SharedIndexInformer 缓存
    c.Log.Info("Starting Controller")

    for _, watch := range c.startWatches {
      syncingSource, ok := watch.src.(source.SyncingSource)
      if !ok {
        continue
      }
      // 等待 Informer 同步完成
      if err := syncingSource.WaitForSync(stop); err != nil {
        err := fmt.Errorf("failed to wait for %s caches to sync: %w", c.Name, err)
        c.Log.Error(err, "Could not wait for Cache to sync")
        return err
      }
    }

    // 所有的 watches 已经启动，重置
    c.startWatches = nil

    if c.JitterPeriod == 0 {
      c.JitterPeriod = 1 * time.Second
    }

    // 启动 workers 来处理资源
    c.Log.Info("Starting workers", "worker count", c.MaxConcurrentReconciles)
    for i := 0; i < c.MaxConcurrentReconciles; i++ {
      go wait.Until(c.worker, c.JitterPeriod, stop)
    }
    c.Started = true
    return nil
  }()
  if err != nil {
    return err
  }

  <-stop
  c.Log.Info("Stopping workers")
  return nil
}
```

上面的 Start 函数很简单，和我们之前自定义控制器中启动控制循环比较类似，都是先等待资源对象的 Informer 同步完成，然后启动 workers 来处理资源对象，而且 worker 函数都是一样的实现方式：

```
// pkg/internal/controller/controller.go

// worker 运行一个工作线程，从队列中弹出元素处理，并标记为完成
// 强制要求永远不和同一个对象同时调用 reconcileHandler
func (c *Controller) worker() {
  for c.processNextWorkItem() {
  }
}

// processNextWorkItem 将从工作队列中弹出一个元素，并尝试通过调用 reconcileHandler 来处理它
func (c *Controller) processNextWorkItem() bool {
  // 从队列中弹出元素
  obj, shutdown := c.Queue.Get()
  if shutdown {
    // 队列关闭了，直接返回 false
    return false
  }

  // 标记为处理完成
  defer c.Queue.Done(obj)
  // 调用 reconcileHandler 进行元素处理
  return c.reconcileHandler(obj)
}

func (c *Controller) reconcileHandler(obj interface{}) bool {
  // 处理完每个元素后更新指标
```

```
  reconcileStartTS := time.Now()
  defer func() {
    c.updateMetrics(time.Since(reconcileStartTS))
  }()

  // 确保对象是一个有效的 request 对象
  req, ok := obj.(reconcile.Request)
  if !ok {
    // 工作队列中的元素无效，所以调用 Forget 函数
    // 否则会进入一个循环尝试处理一个无效的元素
    c.Queue.Forget(obj)
    c.Log.Error(nil, "Queue item was not a Request", "type", fmt.Sprintf("%T", obj), "value", obj)
    // 直接返回 true
    return true
  }

  log := c.Log.WithValues("name", req.Name, "namespace", req.Namespace)

  // RunInformersAndControllers 的 syncHandler，传递给它要同步的资源的 namespace/Name 字符串
  // 调用 Reconciler 函数来处理这个元素，也就是我们真正去编写业务逻辑的地方
  if result, err := c.Do.Reconcile(req); err != nil {
    // 如果业务逻辑处理出错，重新添加到限速队列中去
    c.Queue.AddRateLimited(req)
    log.Error(err, "Reconciler error")
    // Metrics 指标记录
    ctrlmetrics.ReconcileErrors.WithLabelValues(c.Name).Inc()
    ctrlmetrics.ReconcileTotal.WithLabelValues(c.Name, "error").Inc()
    return false
  } else if result.RequeueAfter > 0 {
    // 如果调谐函数 Reconcile 处理结果中包含大于0的 RequeueAfter
    //
    // 需要注意如果 result.RequeuAfter 与一个非 nil 的错误一起返回，则 result.RequeuAfter 会丢失。
    // 忘记元素
    c.Queue.Forget(obj)
    // 加入队列
    c.Queue.AddAfter(req, result.RequeueAfter)
    ctrlmetrics.ReconcileTotal.WithLabelValues(c.Name, "requeue_after").Inc()
    return true
  } else if result.Requeue {
    // 重新加入队列
    c.Queue.AddRateLimited(req)
    ctrlmetrics.ReconcileTotal.WithLabelValues(c.Name, "requeue").Inc()
    return true
  }

  // 最后如果没有发生错误，我们就会 Forget 这个元素
  // 这样直到发送另一个变化它就不会再被排队了
  c.Queue.Forget(obj)

  // TODO(directxman12): What does 1 mean?  Do we want level constants?  Do we want levels at all?
  log.V(1).Info("Successfully Reconciled")

  // metrics 指标记录
  ctrlmetrics.ReconcileTotal.WithLabelValues(c.Name, "success").Inc()
  // 直接返回true
  return true
}
```

上面的 reconcileHandler 函数就是我们真正执行元素业务处理的地方，函数中包含了事件处理以及错误处理，真正的事件处理是通过 `c.Do.Reconcile(req)` 暴露给开发者的，所以对于开发者来说，只需要在 Reconcile 函数中去处理业务逻辑就可以了。

根据 `c.Do.Reconcile(req)` 函数的返回值来判断是否需要将元素重新加入队列进行处理：

- 如果返回 error 错误，则将元素重新添加到限速队列中
- 如果返回的 `result.RequeueAfter > 0`，则先将元素忘记，然后在 `result.RequeueAfter` 时间后加入到队列中
- 如果返回 `result.Requeue`，则直接将元素重新加入到限速队列中

- 如果正常返回，则直接忘记这个元素

到这里其实基本上就实现了和我们自定义控制器一样的逻辑，只是将业务处理的逻辑暴露给了开发者去自己实现。接下来我们就需要了解下 controller-runtime 是如何去控制器控制器的初始化以及启动的。