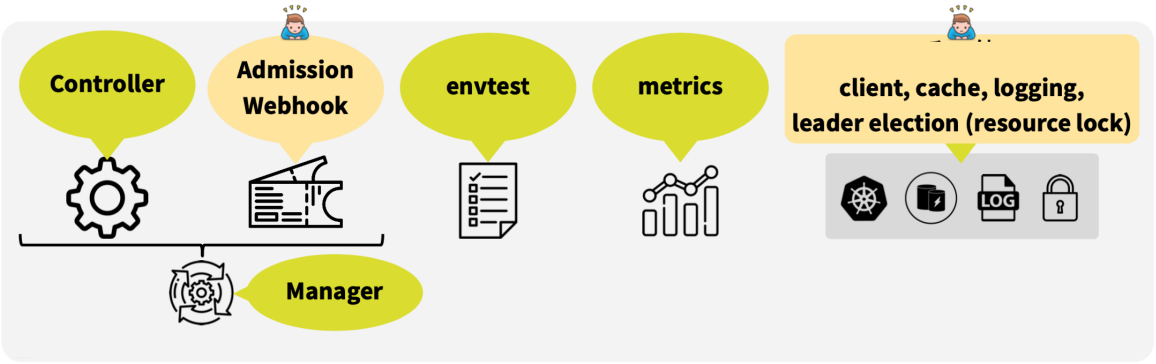
controller-runtime 原理之 manager

本节主要介绍 controller-runtime 框架如何将 Manager 与 Controller 进行关联以及如何启动控制器的。

上文我们介绍了 controller-runtime 中的 Controller 的实现，这个控制器的实现和我们自定义控制器的流程基本一致的，那么 controller-runtime 是如何来使用这个 Controller 的呢，本节我们就来详细介绍下。

在 controller-runtime 中使用了一个 Manager 的接口来管理 Controller，除了控制器其实还可以管理 Admission Webhook，也包括访问资源对象的 client、cache、scheme 等，如下图所示：



Manager 如何使用

首先我们先来查看下 controller-runtime 中的 Manager 是如何使用的，查看 controller-runtime 代码仓库中的示例，位于 `https://github.com/kubernetes-sigs/controller-runtime/tree/master/examples/crd`，示例中关于 Manager 的使用步骤为：

1. 实例化 manager，参数 config
2. 向 manager 添加 scheme
3. 向 manager 添加 controller，该 controller 包含一个 reconciler 结构体，我们需要在 reconciler 结构体实现逻辑处理
4. 向 manager 添加 webhook，同样需要实现逻辑处理
5. 启动 manager.start()

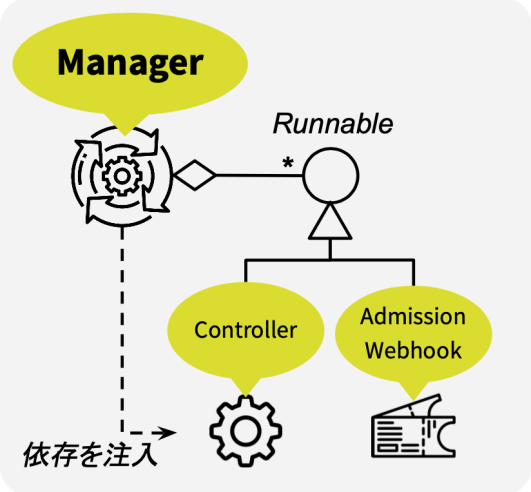
代码如下所示：

```
// 根据 config 实例化 Manager
// config.GetConfigOrDie() 使用默认的配置~/.kube/config
manager.New(config.GetConfigOrDie(), manager.Options{})

// 将 api 注册到 Scheme，Scheme 提供了 GVK 到 go type 的映射。
// 如果多个 crd，需要多次调用 AddToScheme
api.AddToScheme(mgr.GetScheme())
```

```
// 注册 Controller 到 Manager
// For：监控的资源，相当于调用 Watches(&source.Kind{Type: apiType},&handler.EnqueueRequestForObject{})
// Owns：拥有的下属资源，如果 corev1.Pod{} 资源属于 api.ChaosPod{}，也将会被监控，相当于调用 Watches(&source.Kind{Type: <ForType-apiType>}, &har
// reconciler 结构体：继承 Reconciler，需要实现该结构体和 Reconcile 方法
// mgr.GetClient()·mgr.GetScheme() 是 Client 和 Scheme，前面的 manager.New 初始化了
err = builder.ControllerManagedBy(mgr).
    For(&api.ChaosPod{}).
    Owns(&corev1.Pod{}).
    Complete(&reconciler{
      Client: mgr.GetClient(),
      scheme: mgr.GetScheme(),
    })
// 构建webhook
err = builder.WebhookManagedBy(mgr).For(&api.ChaosPod{}).Complete()
// 启动manager,实际上是启动controller
mgr.Start(ctrl.SetupSignalHandler())
```

Manager 是一个用于初始化共享依赖关系的接口，接口定义如下所示（只显示了核心的几个方法）：

```
// pkg/manager/manager.go

// Manager 初始化共享的依赖关系，比如 Caches 和 Client，并将他们提供给 Runnables
type Manager interface {
  // Add 将在组件上设置所需的依赖关系，并在调用 Start 时启动组件
  // Add 将注入接口的依赖关系 - 比如 注入 inject.Client
  // 根据 Runnable 是否实现了 LeaderElectionRunnable 接口判断
  // Runnable 可以在非 LeaderElection 模式（始终运行）或 LeaderElection 模式（如果启用了 LeaderElection，则由 LeaderElection 管理）下运行
  Add(Runnable) error

  // SetFields 设置对象上的所有依赖关系，而该对象已经实现了 inject 接口
  // 比如 inject.Client
  SetFields(interface{}) error


  // Start 启动所有已注册的控制器，并一直运行，直到停止通道关闭
  // 如果启动任何控制器都出错，则返回错误。
  // 如果使用了 LeaderElection，则必须在此返回后立即退出二进制，否则需要 Leader 选举的组件可能会在 Leader 锁丢失后继续运行
  Start(<-chan struct{}) error

  ......
}
```



Manager 可以管理 Runnable 的生命周期（添加/启动），如果您不通过 Manager 启动（需要处理各种常见的依赖关系）。

Manager 还保持共同的依赖性：client、cache、scheme 等。

- 提供了getter(例如GetClient())
- 还有一个简单的依赖注入机制(runtime/inject)

此外还支持领导人选举，只需用选项指定即可，还提供了一个用于优雅关闭的信号处理程序。

Manager 实例化

然后查看下 Manager 的实例化 New 函数的实现：

```
// 返回一个新的 Manager，用于创建 Controllers
func New(config *rest.Config, options Options) (Manager, error) {
  if config == nil {
    return nil, fmt.Errorf("must specify Config")
  }
```

```
  // 设置 options 属性的默认值
  options = setOptionsDefaults(options)
  ......
  return &controllerManager{
    ......
  }, nil
}
```

New 函数中就是为 Manager 执行初始化工作，其中 `setOptionsDefaults` 函数为 Options 属性设置了默认的一些参数值，最后返回的是一个 controllerManager 的实例，这是因为该结构体是 Manager 接口的一个实现，所以 Manager 的真正操作都是这个结构体去实现的。

接下来最重要的就是注册 Controller 到 Manager 的过程：

```
err = builder.ControllerManagedBy(mgr).
    For(&api.ChaosPod{}).
    Owns(&corev1.Pod{}).
    Complete(&reconciler{
      Client: mgr.GetClient(),
      scheme: mgr.GetScheme(),
    })
```

`builder.ControllerManagedBy` 函数返回一个新的控制器构造器 Builder 对象，生成的控制器将由所提供的管理器 Manager 启动，函数实现很简单：

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

// 控制器构造器
type Builder struct {
  forInput         ForInput
  ownsInput        []OwnsInput
  watchesInput     []WatchesInput
  mgr              manager.Manager
  globalPredicates []predicate.Predicate
  config           *rest.Config
  ctrl             controller.Controller
  ctrlOptions      controller.Options
  log              logr.Logger
  name             string
}

// ControllerManagedBy 返回一个新的控制器构造器
// 它将由提供的 Manager 启动
func ControllerManagedBy(m manager.Manager) *Builder {
  return &Builder{mgr: m}
}
```

可以看到 controller-runtime 封装了一个 Builder 的结构体用来生成 Controller，将 Manager 传递给这个构造器，然后是调用构造器的 `For` 函数：

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

// ForInput 表示 For 方法设置的信息
type ForInput struct {
  object     runtime.Object
  predicates []predicate.Predicate
}

// For 函数定义了被调谐的对象类型
// 并配置 ControllerManagedBy 通过调谐对象来响应 create/delete/update 事件
// 调用 For 函数相当于调用：
// Watches(&source.Kind{Type: apiType}, &handler.EnqueueRequestForObject{})
func (blder *Builder) For(object runtime.Object, opts ...ForOption) *Builder {
  input := ForInput{object: object}
  for _, opt := range opts {
    opt.ApplyToFor(&input)
  }
  blder.forInput = input
  return blder
}
```

`For` 函数就是用来定义我们要处理的对象类型的，接着调用了 `Owns` 函数：

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

// OwnsInput 表示 Owns 方法设置的信息
type OwnsInput struct {
  object     runtime.Object
  predicates []predicate.Predicate
}

// Owns 定义了 ControllerManagedBy 生成的对象类型
// 并配置 ControllerManagedBy 通过调谐所有者对象来响应 create/delete/update 事件
// 这相当于调用：
// Watches(&source.Kind{Type: <ForType-forInput>}, &handler.EnqueueRequestForOwner{OwnerType: apiType, IsController: true})
func (blder *Builder) Owns(object runtime.Object, opts ...OwnsOption) *Builder {
  input := OwnsInput{object: object}
  for _, opt := range opts {
    opt.ApplyToOwns(&input)
  }

  blder.ownsInput = append(blder.ownsInput, input)
  return blder
}
```

`Owns` 函数就是来配置我们监听的资源对象的子资源，如果想要协调资源则需要调用 `Owns` 函数进行配置，然后就是最重要的 `Complete` 函数了：

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

func (blder *Builder) Complete(r reconcile.Reconciler) error {
  // 调用 Build 函数构建 Controller
  _, err := blder.Build(r)
  return err
}

// Build 构建应用程序 ControllerManagedBy 并返回它创建的 Controller
func (blder *Builder) Build(r reconcile.Reconciler) (controller.Controller, error) {
  if r == nil {
    return nil, fmt.Errorf("must provide a non-nil Reconciler")
  }
  if blder.mgr == nil {
    return nil, fmt.Errorf("must provide a non-nil Manager")
  }

  // 配置 Rest Config
  blder.loadRestConfig()

  // 配置 ControllerManagedBy
  if err := blder.doController(r); err != nil {
    return nil, err
  }

  // 配置 Watch
  if err := blder.doWatch(); err != nil {
    return nil, err
  }

  return blder.ctrl, nil
}
```

`Complete` 函数通过调用 Build 函数来构建 Controller，其中比较重要的就是 `doController` 和 `doWatch` 两个函数，`doController` 就是去真正实例化 Controller 的函数：

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

// 根据 GVK 获取控制器名称
func (blder *Builder) getControllerName(gvk schema.GroupVersionKind) string {
  if blder.name != "" {
    return blder.name
  }
  return strings.ToLower(gvk.Kind)
}

func (blder *Builder) doController(r reconcile.Reconciler) error {
  ctrlOptions := blder.ctrlOptions
  if ctrlOptions.Reconciler == nil {
    ctrlOptions.Reconciler = r
  }

  // 从我们正在调谐的对象中检索 GVK
  gvk, err := getGvk(blder.forInput.object, blder.mgr.GetScheme())
```

```
  if err != nil {
    return err
  }

  // 配置日志 Logger
  if ctrlOptions.Log == nil {
    ctrlOptions.Log = blder.mgr.GetLogger()
  }
  ctrlOptions.Log = ctrlOptions.Log.WithValues("reconcilerGroup", gvk.Group, "reconcilerKind", gvk.Kind)

  // 构造 Controller
  // var newController = controller.New
  blder.ctrl, err = newController(blder.getControllerName(gvk), blder.mgr, ctrlOptions)
  return err
}
```

上面的函数通过获取资源对象的 GVK 来获取 Controller 的名称，最后通过一个 newController 函数（controller.New 的别名）来实例化一个真正的 Controller：

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

// New 返回一个 Manager 处注册的 Controller
// Manager 将确保共享缓存在控制器启动前已经同步
func New(name string, mgr manager.Manager, options Options) (Controller, error) {
  c, err := NewUnmanaged(name, mgr, options)
  if err != nil {
    return nil, err
  }

  // 将 controller 作为 manager 的组件
  return c, mgr.Add(c)
}

// NewUnmanaged 返回一个新的控制器，而不将其添加到 manager 中
// 调用者负责启动返回的控制器
func NewUnmanaged(name string, mgr manager.Manager, options Options) (Controller, error) {
  if options.Reconciler == nil {
    return nil, fmt.Errorf("must specify Reconciler")
  }

  if len(name) == 0 {
    return nil, fmt.Errorf("must specify Name for Controller")
  }

  if options.MaxConcurrentReconciles <= 0 {
    options.MaxConcurrentReconciles = 1
  }

  if options.RateLimiter == nil {
    options.RateLimiter = workqueue.DefaultControllerRateLimiter()
  }

  if options.Log == nil {
    options.Log = mgr.GetLogger()
  }

  // 在 Reconciler 中注入依赖关系
  if err := mgr.SetFields(options.Reconciler); err != nil {
    return nil, err
  }

  // 创建 Controller 并配置依赖关系
  return &controller.Controller{
    Do: options.Reconciler,
    MakeQueue: func() workqueue.RateLimitingInterface {
      return workqueue.NewNamedRateLimitingQueue(options.RateLimiter, name)
    },
    MaxConcurrentReconciles: options.MaxConcurrentReconciles,
    SetFields:               mgr.SetFields,
    Name:                    name,
    Log:                     options.Log.WithName("controller").WithValues("controller", name),
  }, nil
}
```

可以看到 `NewUnmanaged` 函数才是真正实例化 Controller 的地方，终于和前文的 Controller 联系起来来，Controller 实例化完成后，又通过 `mgr.Add(c)` 函数将控制器添加到 Manager 中去进行管理，所以我们还需要去查看下 Manager 的 Add 函数的实现，当然是看 controllerManager 中的具体实现：

```
// pkg/manager/manager.go
```

```
// Runnable 允许一个组件被启动
type Runnable interface {
  Start(<-chan struct{}) error
}

// pkg/manager/internal.go

// Add 设置i的依赖，并将其他添加到 Runnables 列表中启动
func (cm *controllerManager) Add(r Runnable) error {
  cm.mu.Lock()
  defer cm.mu.Unlock()
  if cm.stopProcedureEngaged {
    return errors.New("can't accept new runnable as stop procedure is already engaged")
  }

  // 设置对象的依赖
  if err := cm.SetFields(r); err != nil {
    return err
  }

  var shouldStart bool

  // 添加 runnable 到 leader election 或者非 leaderelection 列表
  if leRunnable, ok := r.(LeaderElectionRunnable); ok && !leRunnable.NeedLeaderElection() {
    shouldStart = cm.started
    cm.nonLeaderElectionRunnables = append(cm.nonLeaderElectionRunnables, r)
  } else {
    shouldStart = cm.startedLeader
    cm.leaderElectionRunnables = append(cm.leaderElectionRunnables, r)
  }

  if shouldStart {
    // 如果已经启动，启动控制器
    cm.startRunnable(r)
  }

  return nil
}

func (cm *controllerManager) startRunnable(r Runnable) {
  cm.waitForRunnable.Add(1)
  go func() {
    defer cm.waitForRunnable.Done()
    if err := r.Start(cm.internalStop); err != nil {
      cm.errChan <- err
    }
  }()
}
```

controllerManager 的 Add 函数传递的是一个 Runnable 参数，Runnable 是一个接口，用来表示可以启动的一个组件，而恰好 Controller 实际上就实现了这个接口的 Start 函数，所以可以通过 Add 函数来添加 Controller 实例，在 Add 函数中除了依赖注入之外，还根据 Runnable 来判断组件是否支持选举功能，支持则将组件加入到 `leaderElectionRunnables` 列表中，否则加入到 `nonLeaderElectionRunnables` 列表中，这点非常重要，涉及到后面控制器的启动方式。

启动 Manager

如果 Manager 已经启动了，现在调用 Add 函数来添加 Runnable，则需要立即调用 `startRunnable` 函数启动控制器，startRunnable 函数就是在一个 goroutine 中去调用 Runnable 的 Start 函数，这里就相当于调用 Controller 的 Start 函数来启动控制器了。

到这里就实例化 Controller 完成了，回到前面 Builder 的 build 函数中，`doController` 函数调用完成，接着是 `doWatch` 函数的实现：

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

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

  // Watches 管理的类型（子类型）
  for _, own := range blder.ownsInput {
    src := &source.Kind{Type: own.object}
    hdler := &handler.EnqueueRequestForOwner{
```

```
      OwnerType:    blder.forInput.object,
      IsController: true,
    }
    allPredicates := append([]predicate.Predicate(nil), blder.globalPredicates...)
    allPredicates = append(allPredicates, own.predicates...)
    if err := blder.ctrl.Watch(src, hdler, allPredicates...); err != nil {
      return err
    }
  }

  // 执行 watch 请求
  for _, w := range blder.watchesInput {
    allPredicates := append([]predicate.Predicate(nil), blder.globalPredicates...)
    allPredicates = append(allPredicates, w.predicates...)
    if err := blder.ctrl.Watch(w.src, w.eventhandler, allPredicates...); err != nil {
      return err
    }

  }
  return nil
}
```

上面的 doWatch 函数就是去将我们需要调谐的资源对象放到 Controller 中进行 Watch 操作，包括资源对象管理的子类型，都需要去执行 Watch 操作，这就又回到了前面 Controller 的 Watch 操作了，其实就是去注册 Informer 的事件监听器，将数据添加到工作队列中去。这样到这里我们就将 Controller 初始化完成，并为我们调谐的资源对象执行了 Watch 操作。

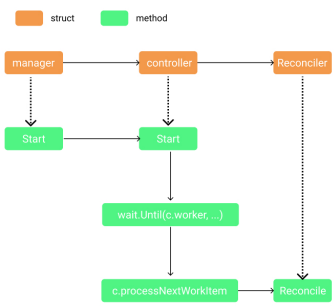
最后是调用 Manager 的 Start 函数来启动 Manager，由于上面我们已经把 Controller 添加到了 Manager 中，所以这里启动其实是启动关联的 Controller，启动函数实现如下所示：

```
// pkg/manager/internal.go

func (cm *controllerManager) Start(stop <-chan struct{}) (err error) {
  stopComplete := make(chan struct{})
  defer close(stopComplete)
  // stopComplete 关闭后必须在 deferer 执行下面的操作，否则会出现死锁
  defer func() {
    // https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/gettyimages-459889618-1533579787.jpg
    stopErr := cm.engageStopProcedure(stopComplete)
    if stopErr != nil {
      if err != nil {
        err = utilerrors.NewAggregate([]error{err, stopErr})
      } else {
        err = stopErr
      }
    }
  }()

  cm.errChan = make(chan error)

  // Metrics 服务
  if cm.metricsListener != nil {
    go cm.serveMetrics(cm.internalStop)
  }

  // 健康检测的服务
  if cm.healthProbeListener != nil {
    go cm.serveHealthProbes(cm.internalStop)
  }

  // 启动非 LeaderElection 的 Runnables
  go cm.startNonLeaderElectionRunnables()

  go func() {
    if cm.resourceLock != nil {
      // 启动 LeaderElection 选举
      err := cm.startLeaderElection()
      if err != nil {
        cm.errChan <- err
      }
    } else {
      close(cm.elected)
      // 启动 LeaderElection 的 Runnables
      go cm.startLeaderElectionRunnables()
    }
  }()

  select {
  case <-stop:
    // We are done
    return nil
  case err := <-cm.errChan:
    // Error starting or running a runnable
```

```
    return err
  }
}
```

上面的启动函数其实就是去启动前面我们加入到 Manager 中的 Runnable（Controller），非 LeaderElection 的列表与 LeaderElection 的列表都分别在一个 goroutine 中启动：

```
// pkg/manager/internal.go

func (cm *controllerManager) waitForCache() {
  if cm.started {
    return
  }

  // Start the Cache. Allow the function to start the cache to be mocked out for testing
  if cm.startCache == nil {
    cm.startCache = cm.cache.Start
  }
  cm.startRunnable(RunnableFunc(func(stop <-chan struct{}) error {
    return cm.startCache(stop)
  }))

  cm.cache.WaitForCacheSync(cm.internalStop)
  cm.started = true
}

// 启动非 LeaderElection Runnables
func (cm *controllerManager) startNonLeaderElectionRunnables() {
  cm.mu.Lock()
  defer cm.mu.Unlock()
  // 等待缓存同步完成
  cm.waitForCache()

  // 开始启动所有的非 leaderelection 的 Runnables
  for _, c := range cm.nonLeaderElectionRunnables {
    cm.startRunnable(c)
  }
}

func (cm *controllerManager) startLeaderElectionRunnables() {
  cm.mu.Lock()
  defer cm.mu.Unlock()
  // 等待缓存同步完成
  cm.waitForCache()

  for _, c := range cm.leaderElectionRunnables {
    cm.startRunnable(c)
  }

  cm.startedLeader = true
}

// 真正的启动一个 Runnable
func (cm *controllerManager) startRunnable(r Runnable) {
  cm.waitForRunnable.Add(1)
  go func() {
    defer cm.waitForRunnable.Done()
    if err := r.Start(cm.internalStop); err != nil {
      cm.errChan <- err
    }
  }()
}
```



可以看到最终还是去调用的 Runnable 的 Start 函数来启动，这里其实也就是 Controller 的 Start 函数，前文我们已经详细介绍过，这个函数相当于启动一个控制循环不断从工作队列中消费数据，然后给到一个 Reconciler 接口进行处理，也就是我们要去实现的 `Reconcile(Request) (Result, error)` 这个业务逻辑函数。

到这里我们就完成了 Manager 的整个启动过程，包括 Manager 是如何初始化，如何和 Controller 进行关联以及如何启动 Controller 的，了解了整个 controller-runtime 的原理过后，我们再去使用 kubebuilder 来编写 Operator 就更加容易了。