95Kubernetes 系列（八十九）K8s Operator — Conditions

了解如何管理资源状态后，今天我们将更进一步定义条件。

这些条件是什么？

如果您已经使用过 Kubernetes 并描述过资源，那么您已经看到了一些情况，但没有意识到它们的存在。但这个元素确实很重要：

> 条件是对资源进行的一系列检查，以确保资源处于就绪状态。

为什么要实施它们？

如果实现它们真的很重要，那是因为它们将为我们提供大量用于调试的信息。

在我们的系列示例中，MyProxy 可以有一个条件“Pod 已正确启动”。这样我们就可以直接看到资源状态是否一切顺利。

另一种真正有用的方法是使用它们作为创建另一个资源或执行特定操作的要求。

如何实施？

与状态一样，我们必须更新 `api/../xxx_types.go` 中的资源状态定义，并在控制器的 Reconcile 方法中修改状态更新。

更新状态定义

在状态中，我们将添加一个新变量 `Conditions` ：

```
type MyProxyStatus struct {
   Conditions []metav1.Condition `json:"conditions"`
   PodNames   []string           `json:"pod_names"`
}
```

更新协调 Reconcile 功能

与任何其他资源变量一样，我们可以更新 `Conditions` 参数的值。

```
myProxy.Status.Conditions = conditions
err = r.Status().Update(ctx, myProxy)
if err != nil {
   log.Error(err, "Failed to update MyProxy status")
   return ctrl.Result{}, err
}
```

但我们如何创建它们呢？

创建 Conditions

Conditions 对象非常简单，只有 4 个字段：

- Status：其中包含条件的状态。它只接受 3 个值
  - metav1.ConditionTrue : 如果情况得到解决
  - metav1.ConditionFalse : 如果情况没有解决
  - metav1.ConditionUnknown ： 如果没有足够的信息来解决状况的状态
- Type：条件名称（例如：检查 Pod 运行状况）
- Reason：状态摘要（例如：健康的 Pod）
- Message：较长的消息，添加有关条件状态的详细信息（通常用于详细说明错误）

从这里，我们现在可以创建方法来创建 `Conditions` 来填充之前定义的条件变量。

例如：

```
func checkPodExistance(podNames []string) metav1.Condition {
   if len(podNames) == 2 {
      return metav1.Condition{
         Status:  metav1.ConditionTrue,
         Reason:  "Pods found",
         Message: "Both pods were found",
         Type:    "Check existance of pods",
      }
   } else {
      return metav1.Condition{
         Status:  metav1.ConditionFalse,
         Reason:  "Pods not found",
         Message: "The list of pod names doesn't contains the correct number of pods",
         Type:    "Check existance of pods",
      }
   }
}
```

条件是一个强大的工具，一旦你测试过它，你就离不开它！

在本系列的下一集中，我们将重点关注可以在 Operator 中添加的注释！

我希望它能对你有所帮助，如果你有任何问题（没有愚蠢的问题）或某些点你不清楚，请不要犹豫，在评论中添加你的问题。