How Longhorn design the backup CRDs and the controllers

Date: Oct 07, 2021

Author: JenTing Hsiao

History

When the user clicks the Backup page, Longhorn gathers all the backups information from the remote S3/NFS in synchronous way.

- Pros: easy to implement and can get the real time information.
- Cons:
  - timeout somtimes if there're lots of backups *or* network latency is high.
  - impossible to implement the pagination by user specified criteria (because we have our own filed in remote metadata which AWS S3 SDK doesn't know the custom field).

So

We decided to async pull the remote backups and stores into the cluster CRs (backup download).

Besides, we moves the backup creation to use Backup CR also (backup upload).

What do we have

- Backup Target CRD and it's controller.
- Backup Volume CRD and it's controller.
- Backup CRD and it's controller.

Note: a controller is to keep matching the current state of a resource with its desired state.

1st design of the CRDs

- Backup Target

```
metadata:
  name: the backup target name.
spec:
  backupTargetURL: the backup target URL. (string)
  credentialSecret: the backup target credential secret. (string)
  pollInterval: the backup target poll interval. (metav1.Duration)
status:
  ownerID: the node ID which is responsible for running operations of the backup target controller. (string)
  available: records if the remote backup target is available or not. (bool)
  lastSyncedAt: records the last time the backup target was running the reconcile process. (metav1.Time)
```

- Backup Volume

```
skip...
```

- Backup

```
skip...
```

Discussion of 1st design of CRDs

By default, Longhorn controller runs reconcile when a create/update/delete events comes in *or* reconcile all CRs every 30 seconds.

```
# resync time is 30 secs
lhInformerFactory := lhinformers.NewSharedInformerFactory(lhClient, time.Second*30)
```

- Pros: easy to understand, the backup_target_controller reconcile the BackupTarget CR spec, and updates the status by check

  ```
  if (time.Now() - status.LastSyncedAt < spec.PollInterval) {
      return nil
  }
  // run reconcile loop
  ```

- Cons:
  - if the user configures the poll interval < 30 seconds, the timer is inaccurate.
  - unable to design a force sync *now* mechanism.

2nd design of CRDs

- Backup Target

```
metadata:
  name: the backup target name.
spec:
  backupTargetURL: the backup target URL. (string)
  credentialSecret: the backup target credential secret. (string)
  pollInterval: the backup target poll interval. (metav1.Duration)
  syncRequestAt: the time to request run sync the remote backup target. (metav1.Time)
status:
  ownerID: the node ID which is responsible for running operations of the backup target controller. (string)
  available: records if the remote backup target is available or not. (bool)
  lastSyncedAt: records the last time the backup target was running the reconcile process. (metav1.Time)
```

- Backup Volume

```
skip...
```

- Backup

```
skip...
```

Discussion of 2nd design of CRDs

- Pros:
  - the timer is accurate.
  - have a force sync *now* mechanism.
- Cons:
  - have to create a timer to periodically updates the `spec.syncRequestAt` when the poll interval reaches.

```
wait.PollUntil(pollInterval, func() (done bool, err error) {
    backupTarget.Spec.SyncRequestedAt = time.Now()
    UpdateBackupTarget(backupTarget)
    return false, nil
}, stopCh)
```

  - now, we create a timer in setting_controller. But it's able to leverage the Kubernetes Job.

Let's talk about the ownerID

The Longhorn-manager is a DaemonSet (every node have a Pod).
So, every node runs the same controllers.

- For per-node operation: only ownerID (owner node) is responsible to reconcile the CR, for example, only the volume belongs to this node should reconcile the volume CR in volume_controller.
- For per-cluster operation:
  - Kubernetes contains the leader-election mechanism that only 1 controller will runs across all nodes.
  - However, Longhorn does not go with leader election mechanism way but leverage the ownerID (owner node).

Let's talk about the ownerID (cont.)

Each controller has it's way to decide the ownerID.

Generally speaking are:

- node is ready
- engine image is ready
- ...

```
func isControllerResponsibleFor(controllerID string, ds *datastore.DataStore, name, preferredOwnerID, currentOwnerID string) bool {
    // we use this approach so that if there is an issue with the data store
    // we don't accidentally transfer ownership
    isOwnerUnavailable := func(node string) bool {
        isUnavailable, err := ds.IsNodeDownOrDeletedOrMissingManager(node)
        if node != "" && err != nil {
            logrus.Errorf("Error while checking IsNodeDownOrDeletedOrMissingManager for object %v, node %v: %v", name, node, err)
        }
        return node == "" || isUnavailable
    }

    isPreferredOwner := controllerID == preferredOwnerID
    continueToBeOwner := currentOwnerID == controllerID && isOwnerUnavailable(preferredOwnerID)
    requiresNewOwner := isOwnerUnavailable(currentOwnerID) && isOwnerUnavailable(preferredOwnerID)
    return isPreferredOwner || continueToBeOwner || requiresNewOwner
}
```

You could check the function `isResponsibleFor` inside each controller.

Register Multiple Informers Within A Controller

For example, we want to notify the backup_target_controller once the default engine image is ready so the backup_target_controller is able to list/read remote backup target by the default engine image.

```
backupTargetInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc:    btc.enqueueBackupTarget,
    UpdateFunc: func(old, cur interface{}) { btc.enqueueBackupTarget(cur) },
})

engineImageInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    UpdateFunc: func(old, cur interface{}) {
        btc.enqueueEngineImage(cur)
    },
})

func (btc *BackupTargetController) enqueueEngineImage(obj interface{}) {
        ei, ok := obj.(*longhorn.EngineImage)
        if !ok {
                return
        }

        defaultEngineImage, err := btc.ds.GetSettingValueExisted(types.SettingNameDefaultEngineImage)
        // Enqueue the backup target only when the default engine image becomes ready
        if err != nil || ei.Spec.Image != defaultEngineImage || ei.Status.State != types.EngineImageStateDeployed {
                return
        }
        btc.queue.AddRateLimited(ei.Namespace + "/" + types.DefaultBackupTargetName)
}
```

How Controller Prevent Handles The Unchanged Resource

Since the controller resync perioid is 30 seconds by default. So every 30 seconds, the controller reconciles all the resources again.

How do we prevent the unchanged resource won't enqueue?

> Use resourceVersion

```
engineImageInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    UpdateFunc: func(old, cur interface{}) {
        oldEI := old.(*longhorn.EngineImage)
        curEI := cur.(*longhorn.EngineImage)
        if curEI.ResourceVersion == oldEI.ResourceVersion {
            return
        }
        btc.enqueueEngineImage(cur)
    },
})
```

Code Convention - Defer Function

In Longhorn controller, the code convention to updating the status field is to use defer function.

```
existingBackupTarget := backupTarget.DeepCopy()
defer func() {
  if err != nil {
    // we don't want to update status field if there is anything wrong.
    return
  }
  if reflect.DeepEqual(existingBackupTarget.Status, backupTarget.Status) {
    // make sure the one of the status field is updated.
    return
  }
  if _, err := btc.ds.UpdateBackupTargetStatus(backupTarget); err != nil && apierrors.IsConflict(errors.Cause(err)) {
    // update the status field.
    // re-enqueue when update conflicts.
    btc.enqueueBackupTarget(backupTarget)
  }
}()
```

Future Of Longhorn Controller

- CRD Schema
- CR Data Validation
- Longhorn operation could be interact by CR resource directly.
  - leverage by kubectl *or* Longhorn CLI.
  - right now, Longhorn verified operation is HTTP endpoint (longhorn-ui *or* python client).