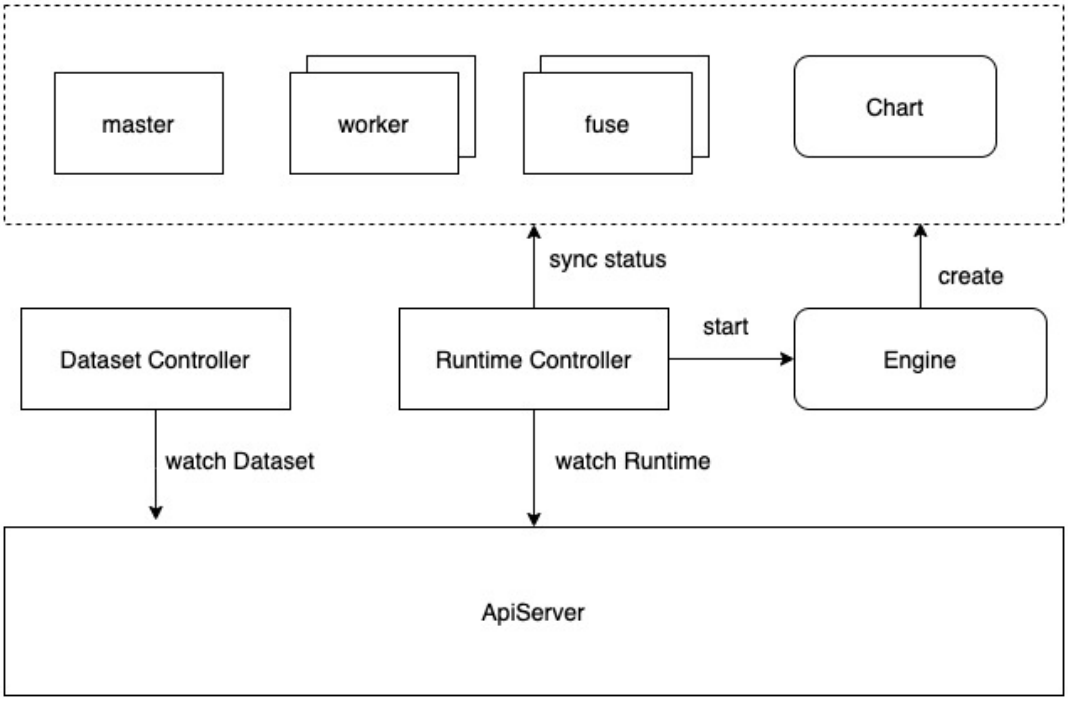
如何实现 Fluid 定制化 runtime

作者：朱唯唯 from Juicedata

Fluid 工作原理

基于 CRD + Controller 的模型

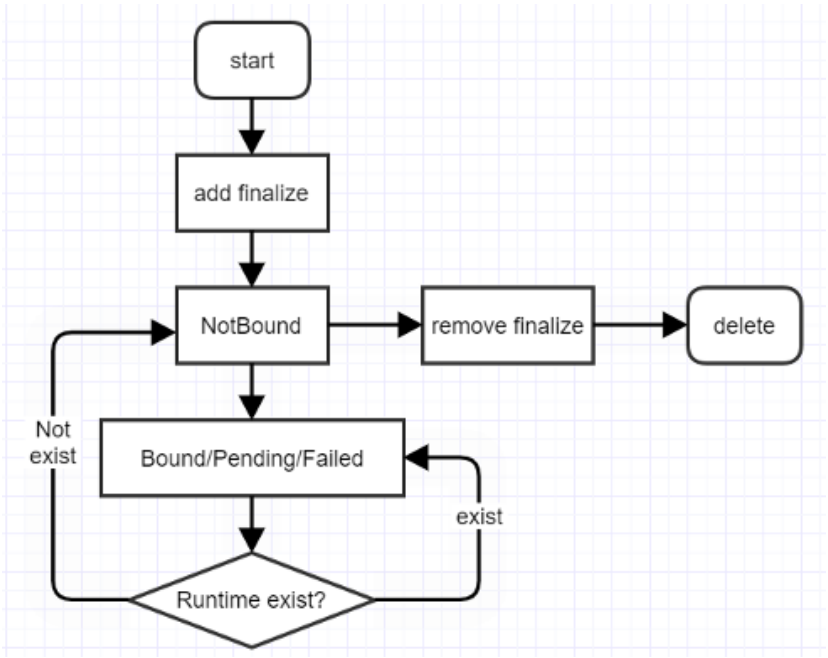


Dataset Controller：管理 Dataset 的生命周期；

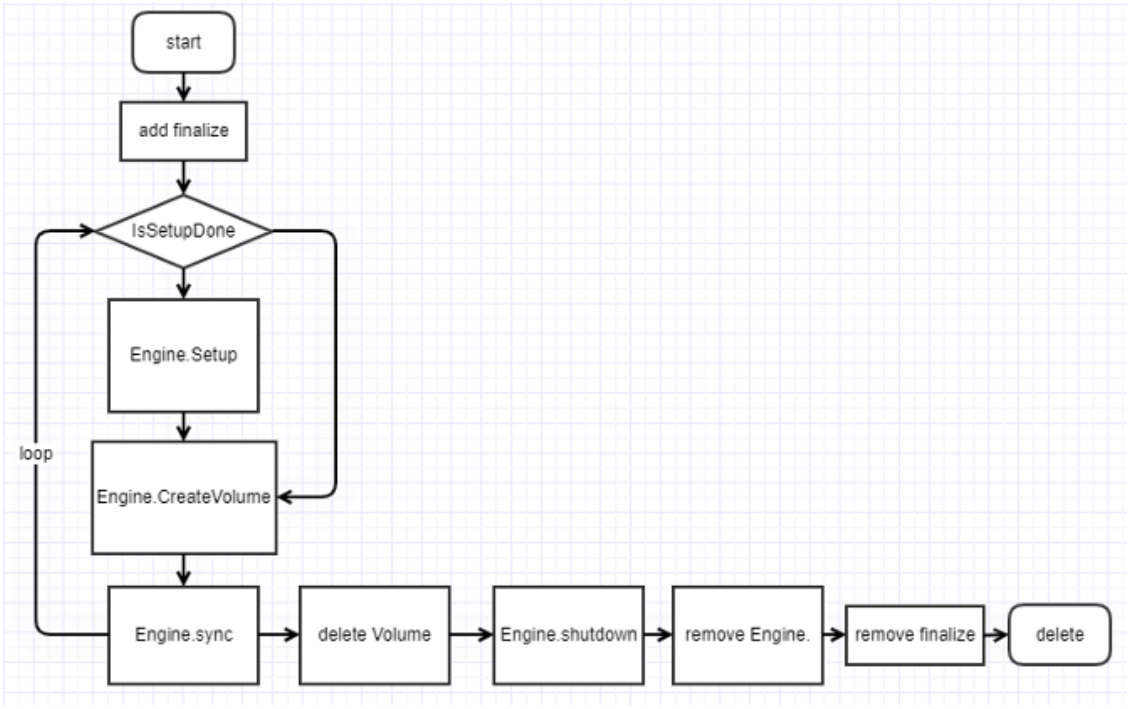
Runtime Controller：管理相对应 Runtime 的生命周期；

Engine：Runtime Controller 内部的组件，负责维护整个 runtime 底层；

dataset 的生命周期



runtime 的生命周期



我们怎么接入？

第一步：创建 CRD xxxRuntime

```
kubebuilder create api --group data --version v1alpha1 --kind xxxRuntime --namespaced true
```

在 `api/v1alpha1/xxxruntime_types.go` 文件中，填充 xxxRuntimeSpec 添加自定义的字段，比如 JuiceFSRuntimeSpec：

```
type JuiceFSRuntimeSpec struct {
  // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
  // Important: Run "make" to regenerate code after modifying this file

  // The version information that instructs fluid to orchestrate a particular version of JuiceFS.
  JuiceFSVersion VersionSpec `json:"juicefsVersion,omitempty"`

  // The spec of init users
  InitUsers InitUsersSpec `json:"initUsers,omitempty"`

  // The component spec of JuiceFS master
  Master JuiceFSCompTemplateSpec `json:"master,omitempty"`

  // The component spec of JuiceFS worker
  Worker JuiceFSCompTemplateSpec `json:"worker,omitempty"`

  // The component spec of JuiceFS job Worker
  JobWorker JuiceFSCompTemplateSpec `json:"jobWorker,omitempty"`

  // Desired state for JuiceFS Fuse
  Fuse JuiceFSFuseSpec `json:"fuse,omitempty"`

  // Tiered storage used by JuiceFS
  TieredStore TieredStore `json:"tieredstore,omitempty"`

  // Configs of JuiceFS
  Configs *[]string `json:"configs,omitempty"`

  // The replicas of the worker, need to be specified
  Replicas int32 `json:"replicas,omitempty"`

  // Manage the user to run Juicefs Runtime
  RunAs *User `json:"runAs,omitempty"`

  // Disable monitoring for JuiceFS Runtime
  // Prometheus is enabled by default
  // +optional
  DisablePrometheus bool `json:"disablePrometheus,omitempty"`
}
```

第二步：创建 Controller

在上一步创建出来的文件 `xxx_controller.go` 中，定义 xxxRuntimeReconciler 结构体，并包含 `controllers.RuntimeReconciler`

```
type JuiceFSRuntimeReconciler struct {
  Scheme  *runtime.Scheme
  engines map[string]base.Engine
  mutex   *sync.Mutex
  *controllers.RuntimeReconciler
}
```

`RuntimeReconciler` 实现了一些通用的方法，我们可以在自己的 reconciler 中调用GetRuntime方法获取 xxxruntime，塞入 ctx；调用 ReconcileInternal 实现具体逻辑。

```
func (r *JuiceFSRuntimeReconciler) Reconcile(context context.Context, req ctrl.Request) (ctrl.Result, error) {
  ...
  ctx := cruntime.ReconcileRequestContext{
    Context:        context,
    Log:            r.Log.WithValues("juicefsruntime", req.NamespacedName),
    NamespacedName: req.NamespacedName,
    Recorder:       r.Recorder,
    Category:       common.AccelerateCategory,
    RuntimeType:    runtimeType,
    Client:         r.Client,
    FinalizerName:  runtimeResourceFinalizerName,
  }

  ctx.Log.V(1).Info("process the request", "request", req)
  //  1.Load the Runtime
  runtime, err := r.getRuntime(ctx)
  ...
  ctx.Runtime = runtime

  // reconcile the implement
  return r.ReconcileInternal(ctx)
}
```

我们在 xxxRuntimeReconciler 的 `GetOrCreateEngine` 方法中构建自己的 engine

第三步：Engine 的实现

Engine 所有的实现都在 pkg/ddc/xxx 文件夹中。

初始化 Engine，提供一个 `Build` 方法，返回我们的自定义 Engine：

```
func Build(id string, ctx cruntime.ReconcileRequestContext) (base.Engine, error) {
  engine := &JuiceFSEngine{
    name:                  ctx.Name,
    namespace:             ctx.Namespace,
    Client:                ctx.Client,
```

```
    Log:                    ctx.Log,
    runtimeType:            ctx.RuntimeType,
    gracefulShutdownLimits: 5,
    retryShutdown:          0,
    MetadataSyncDoneCh:     nil,
  }
  ...

  engine.Helper = ctrl.BuildHelper(runtimeInfo, ctx.Client, engine.Log)
  template := base.NewTemplateEngine(engine, id, ctx)
  ...
  return template, err
}
```

并将 Engine 注册到 pkg/ddc/factory.go 的 init 函数中：

```
func init() {
  buildFuncMap = map[string]buildFunc{
    "alluxio":  alluxio.Build,
    "jindo":    jindo.Build,
    "jindofsx": jindofsx.Build,
    "goosefs":  goosefs.Build,
    "juicefs":  juicefs.Build,
  }
}
```

Engine 需要实现的接口：

engine.Setup

准备自己的 runtime 的 chart 安装包，放在 charts/xxx 中。一般包含 master（statefulset）、worker（statefulset）、fuse（daemonset）。

master 相关：

- ShouldSetupMaster：检查 runtime 的 status，判断是否需要安装 master
- SetupMaster：根据 runtime 和 dataset 的参数，渲染 values ，创建 chart
- CheckMasterReady：检查 master 是否 ready，并更新 runtime 的状态

Worker 相关：

- ShouldSetupWorkers：检查 runtime 的 status，判断是否需要安装 worker
- SetupWorkers：根据 runtime 的 .Spec.Replicas，调用AssignNodesToCache方法，选择对应数量的主机。

- CheckWorkersReady：检查 worker 是否 ready，并更新 runtime 的状态

UFS 相关：

- ShouldCheckUFS
- ShouldUpdateUFS
- PrepareUFS
- UpdateOnUFSChange

engine.CreateVolume & engine.DeleteVolume

- CreateVolume：创建与 Dataset 同名的 PV 和 PVC。并将挂载点和 mountType 传入 PV 中。
- DeleteVolume：删除对应的 PV 和 PVC

engine.Sync

- SyncRuntime
- SyncMetadata
- UpdateCacheOfDataset
- CheckRuntimeHealthy
- SyncReplicas
- CheckAndUpdateRuntimeStatus
- UpdateDatasetStatus

第四步：准备 Dockerfile

Dockerfile 放置于 docker/Dockerfile.xxx 中，需要安装：helm、kubectl