云原生服务中心

开发者指南

文档版本 01

发布日期 2024-09-04

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 概述.....1
2 OSC 服务规范.....4
2.1 OSC 服务规范介绍.....4
2.2 服务包规范.....7
2.2.1 OSC 服务包介绍.....7
2.2.2 元数据 Metadata.....8
2.2.3 资源集合 Manifests.....10
2.2.3.1 资源集合.....11
2.2.3.2 自定义资源 CRD.....11
2.2.3.3 自定义服务 CSD.....11
2.2.3.4 vendor 目录.....18
2.2.4 第三方目录 Raw.....19
2.2.5 生命周期 Lifecycle.....20
2.2.6 README.....22
2.3 开源服务包规范.....22
2.3.1 Operator 规范.....22
2.3.1.1 接入规范说明.....23
2.3.2 Helm 规范.....23
2.3.2.1 接入规范说明.....24
3 服务接入.....26
3.1 服务能力介绍.....26
3.2 Helm 服务接入 OSC 平台.....28
3.3 Operator 服务接入 OSC 平台.....29
3.4 对接运维能力(可选).....29
3.4.1 服务包扩展目录.....29
3.4.2 如何配置监控.....31
3.4.3 对接配置日志.....34
3.4.4 对接实例升级能力.....36
3.5 配置表单控件.....36
3.5.1 控件类型.....37
3.5.2 表单控件.....37
3.5.2.1 基础控件.....37

3.5.2.2 自定义控件 39
3.5.2.2.1 specDescriptors 40
3.5.2.2.2 statusDescriptors 44
3.5.3 表单控件配置组合示例 44
4 附录 46
4.1 如何从零开始开发 Operator 46
4.1.1 开发 Operator 46
4.1.1.1 安装 Kubebuilder 47
4.1.1.2 构建 Operator 48
4.1.1.2.1 CRD 介绍 48
4.1.1.2.2 创建 Operator 项目 50
4.1.1.2.3 创建 API 和 Controller 51
4.1.1.3 实现 Operator 52
4.1.1.3.1 定义 API 52
4.1.1.3.2 实现 Controller 53
4.1.1.3.3 生成代码和资源描述文件 56
4.1.2 制作镜像 56
4.1.2.1 制作实例镜像 56
4.1.2.2 制作 Operator 镜像 60
4.1.3 制作服务包 60
4.1.3.1 创建服务包模板 60
4.1.3.2 修改服务包内容 60
4.1.3.3 生成服务包 63
4.2 OSC 扩展说明 63
4.2.1 oscctl 工具使用 63
4.3 Operator 代码示例 65
4.3.1 CRD 典型格式 65
4.3.2 Controller 实现 66
4.3.3 挂载存储 73
4.3.4 创建存储 74
4.4 安装 Controller 75

1 概述

OSC服务开发者指南面向OSC服务开发者，帮助开发者能够快速开发一个新服务或者将已有的服务转换成可以部署在OSC平台使用的服务。OSC为客户提供开箱即用的云原生服务，可以通过OSC部署到任意基础设施，包括云容器引擎、华为云UCS等场景。开发者可以遵从OSC自身的服务规范，或者直接使用开源Helm、Operator-Framework规范进行开发部署使用。

图 1-1 OSC 服务规范场景下开发整体流程图

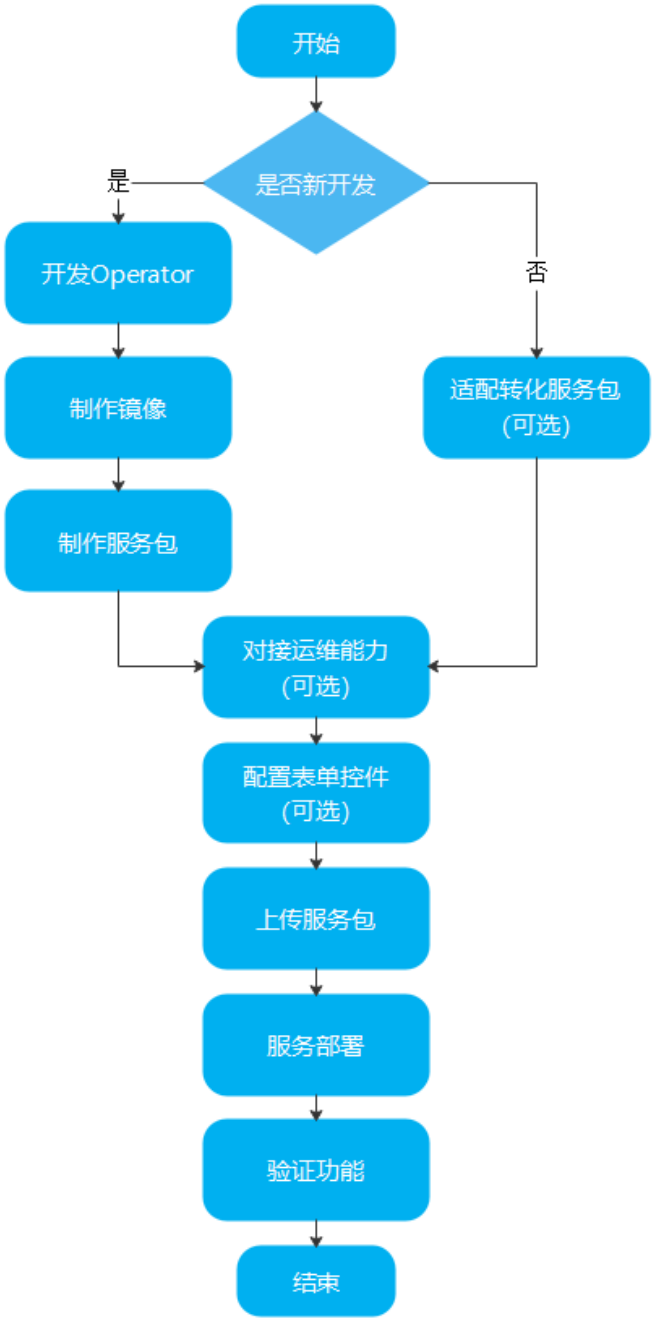
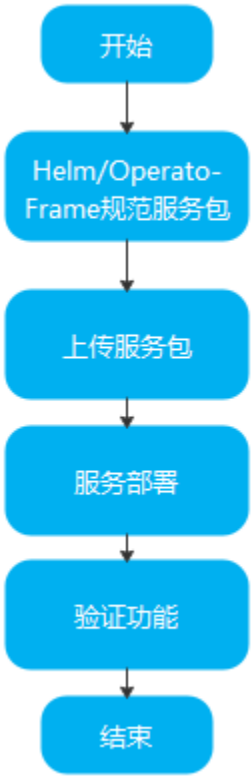


图 1-2 开源服务规范下开发流程图



2 OSC 服务规范

2.1 OSC 服务规范介绍

简介

云原生服务中心（Operator Service Center，OSC）服务规范旨在给出一种与云平台解耦的云原生服务的标准定义，可描述云原生服务在分布式云的部署和治理。

- 部署：云原生服务中心联合华为云分布式服务UCS，能够将应用部署到华为云集群、多云集群、本地集群、附着集群、伙伴集群五种场景，实现应用的跨云和全域部署。
- 治理：容器化只是服务云原生化的第一步，服务部署之后需要治理，不仅包括监控、日志、告警等基本运维能力，还包括弹性伸缩、数据备份恢复、故障迁移、故障恢复等高级运维能力，这些能力是服务高可靠、高可用、高SLA的必要条件。严重依赖云平台的能力，对开发者而言都是重复开发工作，高效快速地赋予服务这些能力是开发者面临的一个难题。

当前最常用的服务管理标准Helm和Operator都没有提供开箱即用的治理能力，为此华为云推出了云原生服务中心OSC，开发者基于服务规范和华为云OSC快速赋予服务全域部署能力和治理能力。

特性

云原生服务由业务功能和运维功能两部分组成，业务功能由开发者完成，运维能力可以由开发者自行提供，也可以直接接入OSC服务规范，托管于平台，让平台来提供。后者开发完成的服务，在OSC云平台可以确保业务功能正常并实现自动化运维。

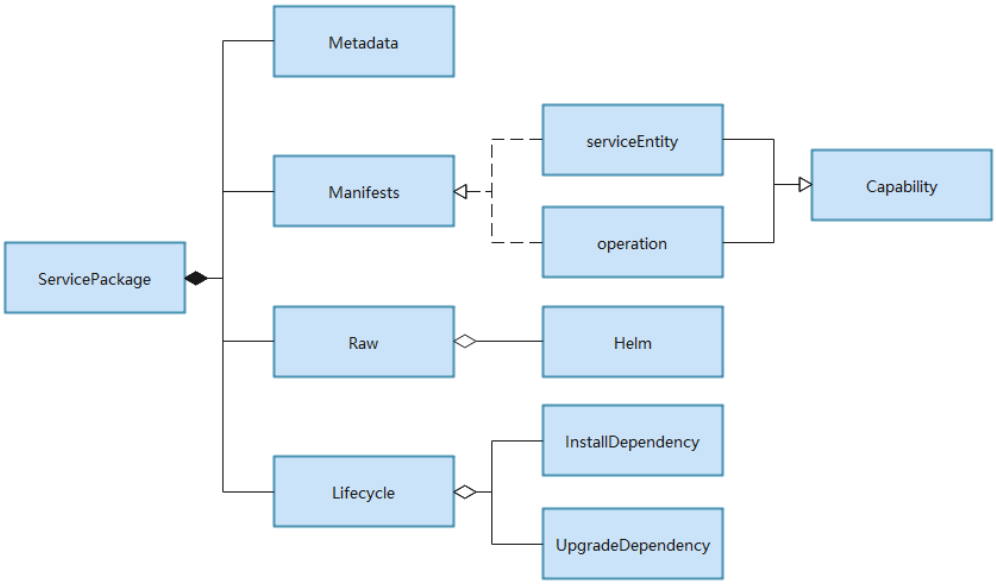
核心特性

- 指定部署平台：选择服务可部署的应用平台，支持云容器引擎、华为云UCS应用部署平台。
- 声明式对接平台运维能力：在服务包中添加运维能力对应的配置文件，无需修改业务代码，就能对接监控、日志、弹性伸缩等运维能力。
- 服务编排：可指定服务之间的拓扑依赖关系。
- 兼容社区规范：全面兼容Helm和Operator Framework服务管理规范。

包含服务

- 服务元数据Metadata：声明服务的核心数据，包括服务类型、支持架构、服务分类、使用场景等必要信息。
- 资源集合Manifests：Resources资源定义的集合，对资源增强的服务能力配置，以及平台公共能力配置信息。
- 第三方兼容Raw：兼容常用第三方服务管理标准，目前支持Helm规范。
- 生命周期Lifecycle：声明服务包的安装、升级等有关服务包的生命周期等相关信息。

图 2-1 领域模型



须知

Raw目录下内容有且仅支持Helm类型服务的管理标准，即服务包类型为Operator，也可能存在Raw目录，但是会在服务包上传部署过程中将Raw目录及目录下内容进行忽略。

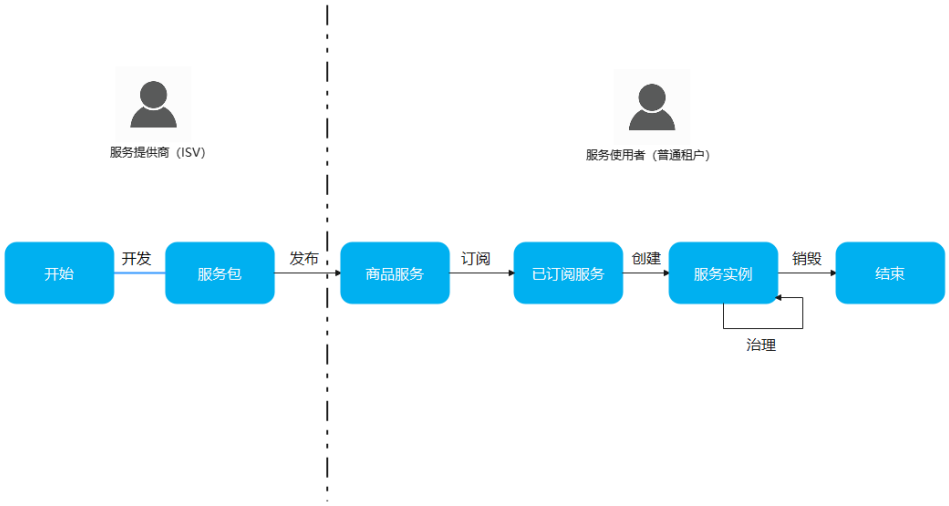
服务生命周期

服务包分为商品服务包和私有服务包。商品包只能由具备ISV资质的租户（服务提供商）进行发布，发布前需经过服务包格式校验、镜像扫描、镜像地址替换、服务自验证等流程，经审核后方可上架。上架后的服务可被其他租户（服务使用者）所使用。而私有包可以由任何一个租户发布，发布后仅由租户自己使用。

商品服务生命周期

1. 服务提供商将服务包上传到华为云云市场，发布成为云市场的**商品服务**。
2. 服务使用者在云市场购买商品服务，该商品服务变成该用户的**已订阅服务**。
3. 服务使用者通过已订阅服务创建**服务实例**。
4. 服务使用者可对服务实例进行编辑、升级、监控等治理操作。
5. 服务使用者可针对不需要的服务实例进行销毁。

图 2-2 商品服务生命周期



私有服务生命周期

- 华为云租户将服务包上传为私有服务。
- 华为云租户使用私有服务创建服务实例。
- 华为云租户可对服务实例进行编辑、升级、监控等治理操作。
- 华为云租户可针对不需要的服务实例进行销毁。

图 2-3 私有服务生命周期

术语

表 2-1 主要术语

名称	介绍
服务包	符合OSC服务包规范的一系列文件聚合，分为公有包和私有包，分别由服务提供商和一般租户开发。
服务目录	OSC平台提供，对公共服务包进行聚合呈现。
公共能力	由OSC平台提供的监控、运维等能力，所有服务包均可通过在vendor目录中声明配置文件，使用公共能力。
订阅	对已发布的商品服务进行订阅，以备进行服务部署。
服务实例	用户通过部署服务后创建的实例。
自定义资源	服务开发者根据自身需求定制的Kubernetes资源对象。

名称	介绍
自定义服务	OSC设计的通过声明式为CRD增强服务能力，包括能力引用，服务依赖，服务访问，交互体验等。

2.2 服务包规范

说明

- 本规范主要面向对象为初次开发的服务以及想要使用OSC平台提供的附加能力进行规范转换的服务。
- 如果开发者已有或仅想使用Helm/Operator-Framework规范的服务，可以直接参考章节**开源服务包规范**。

2.2.1 OSC 服务包介绍

目录结构

```
{OSC-Package}/
├─ metadata.yaml      #【必选】元数据文件
├─ manifests/         #【必选】资源集合
├─ raw/               #【可选】第三方目录
└─ lifecycle.yaml     #【必选】生命周期文件
```

表 2-2 目录结构

名称	目录/文件	描述	必选
元数据文件	metadata.yaml	存放服务包的基本属性和相关信息，包括名称、版本、描述等相关信息。	是
资源集合	manifests	存放Resources资源定义的集合，以及平台公共能力。	是
第三方目录	raw	存放兼容的第三方文件，如Helm/Operator-Framework。	否
生命周期文件	lifecycle.yaml	存放服务包的安装部署、升级等生命周期相关信息。	是

服务包约束

- 命名规则：只允许包含大小写字母、数字、下划线、中划线及点，且不能以下划线、中划线及点结尾。{service_pack_name}-{version}.zip，例如redis-operator-1.0.0.zip。
- 包格式：operator类型服务包支持zip格式，helm类型服务包支持tgz格式。
- 包大小：不超过4MB。

样例（以Kafka为示例）：

```
{OSC-Package}/
├─ metadata.yaml
├─ manifests/
```

```
│   ├── xxx_crd.yaml
│   ├── xxx_csd.yaml
│   └── vendor/
│       ├── monitor_config.yaml
│       ├── logs_config.yaml
├── raw/
│   ├── Chart.yaml
│   ├── values.yaml
│   ├── values.schema.json
│   ├── charts/
│   └── templates
│       └── xxx.yaml
└── lifecycle.yaml
```

2.2.2 元数据 Metadata

元数据文件：{OSC-Package}/metadata.yaml，配置的是服务包的基本信息，包括名称、版本、描述等相关信息。

metadata.yaml样例：

```
name: example-operator
version: "1.0.0"
appVersion: 2.1.2
alias: example-operator.v1.0.0
displayName: Example operator
briefDescription: example operator with an example instance and action
detail: |
  example operator detail description.
source: OpenSource
type: operator
containerImage: swr.cn-east-3.myhuaweicloud.com/osc-official/example-operator:1.0.0
repository: swr.cn-east-3.myhuaweicloud.com
architecture:
  - x86_64
  - aarch64
capabilities:
  - Basic Install
categories:
  - Database
devices:
  - CPU
  - GPU
  - NPU
industries:
  - education
  - media
logo:
  base64data: iVBORw.....
  mediatype: image/png
maintainers:
  - email: test@test.com
    name: test
provider:
  name: Example provider
  url: https://example.com/
scenes:
  - CCE
  - UCS
deployScenes:
  CCE:
    - self
  UCS:
    - self
    - partnercloud
    - onpremise
    - multicloud
    - attachedcluster
links:
```

```
- name: example link
  url: http://github.com/3scale/prome
```

表 2-3 Metadata 参数

参数名称	参数描述	参数示例	必选
name	名称，符合正则表达式 ^[a-z][a-z0-9\-]*[a-z0-9]$，如果服务名称为业界具有特殊含义的字符串，为了避免被转义，可使用引号引起来。	example-operator	是
version	版本。建议使用引号，否则YAML会将类似2.0的版本号识别为浮点数。	1.0.0	是
appVersion	应用版本信息，长度小于64	2.1.2	否
alias	别名	example operator	否
displayName	对外显示的名字	Example operator	否
briefDescription	简要描述，长度小于64	参考样例	否
detail	详细描述	参考样例	否
source	包来源，取值范围：{ISV, OpenSource, HuaweiProvided}，分别表示独立服务供应商、开源服务、华为自有服务，仅支持配置一个值。默认值为OpenSource	OpenSource	否
type	包类型，取值范围：{helm, operator}，仅支持配置一个值，默认值为operator	operator	否
containerImage	容器镜像	参见样例	否
repository	镜像仓库	swr.huawei.com	否
architecture	支持架构列表，取值范围：{x86_64, aarch64}，默认值为x86_64	- x86_64 - aarch64	否
capabilities	能力	Basic Install	否
categories	分类列表，范围不限，如"AI, Database"，注意，当前同一个服务不同版本须配置相同内容，一旦配置后当前不支持修改	Database	否
devices	实例运行需要使用的硬件设备列表，取值范围：{CPU, GPU, NPU}，默认值为CPU	- CPU - GPU	否
industries	行业列表，范围不限，如"education,media"	education	否

参数名称	参数描述	参数示例	必选
logo	图标，包含base64data和mediaType两个字段。如果不设置，则使用对应包类型的默认Logo	参考样例	否
maintainers	维护人员列表，包含name和email两个字段，注意，当前同一个服务不同版本仅支持配置相同内容，一旦配置后当前不支持修改	参考样例	否
provider	提供者，包含name和url两个字段，注意，当前同一个服务不同版本须配置相同内容，一旦配置后当前不支持修改	参考样例	否
scenes	部署平台列表，取值列表：{CCE，UCS}，分别表示CCE平台和UCS平台，大小写敏感，默认值为CCE。 该字段作为兼容性字段，依赖deployScenes属性： 1）当deployScenes不为空，该字段不起作用 2）当deployScenes为空时，该字段起作用	- CCE - UCS	否
deployScenes	部署场景deployScenes包含部署平台(CCE和UCS)和集群类型两部分： ● 当部署平台为CCE时：集群类型只支持华为云集群“self”，不设置集群类型表示不限制 ● 当部署平台为UCS时：集群类型支持以下5种，不设置集群类型表示不限制 – self – partnercloud – onpremise – multicloud – attachedcluster 当部署场景deployScenes未配置，默认只支持部署平台为CCE，且不限制集群类型。	方式一： CCE: - self UCS: - self - partnercloud - onpremise - multicloud - attachedcluster 方式二： CCE: [self] UCS: [self,partnercloud,onpremise, multicloud,attachedcluster]	否
links	附加说明链接列表，包含name和url两个字段，例如项目介绍链接	参考样例	否

2.2.3 资源集合 Manifests

2.2.3.1 资源集合

{OSC-Package}/manifests，用于存放自定义资源文件以及公共能力配置，不同资源用目录进行划分。

资源集合结构：

```
{OSC-Package}/
├── metadata.yaml
├── manifests/          #【必选】资源集合
│   ├── xxx_crd.yaml        #【必选】自定义资源文件
│   ├── xxx_csd.yaml        #【可选】自定义服务文件
│   └── vendor/         #【可选】公共能力目录
├── raw/
├── lifecycle.yaml
└── README.md
```

表 2-4 资源集合结构

目录或文件名称	描述	必选
xxx_crd.yaml	服务包的资源，例如kafka、redis，一个服务包可以包含一个或者多个具体资源目录。	是，至少一个
xxx_csd.yaml	自定义服务文件，每个csd文件对应于一个crd文件。	否
公共能力目录	公共能力目录，存放全部资源CR引用的能力配置文件。	否

2.2.3.2 自定义资源 CRD

自定义资源CRD是Kubernetes服务遵循，一个自定义资源对应一个目录，遵循Kubernetes CRD规范，详细介绍参考官方**CustomResourceDefinition**。

2.2.3.3 自定义服务 CSD

在不同业务环境下，服务对平台有着不同的诉求，Kubernetes提供自定义资源CRD的轻量化机制。而在实践过程中，发现对于服务能力如服务依赖、服务访问、服务部署、监控日志等公共能力欠缺。

因此围绕CRD自研CSD，通过声明式配置CRD的能力引用，增强服务治理能力。

CSD本身属于CustomServiceDefinition类型的资源，其符合k8s资源的通用定义方法。其apiVersion和kind是固定的，metadata.name可自行配置：

```
apiVersion: osc.io/v1beta1
kind: CustomServiceDefinition
metadata:
  name: kafka-csd
```

CSD的spec包含下面的配置：

表 2-5 CSD 的 spec 配置

配置项	配置描述	必填
CRDRef	通过apiVersion，kind以及matadata.name三个字段关联同一目录下的某个CRD资源	是
displayName	所属CRD名称	否
description	所属CRD描述	否
role	描述CRD角色，取值范围为{serviceEntity, operation}，大小写敏感，默认为serviceEntity。	否
capabilityRequirements	平台能力引用配置	否
serviceEntityDependencies	服务依赖配置	否
descriptors	服务安装时参数展示的UI控件定义	否
versionDefinition	服务实例版本定义	否
defaultConfiguration	CRD的CR示例	否

role

在Openshift Operator Framework规范中，开发者定义的各类CRD资源都是对等的，没有更进一步的划分，但从实际的功能属性上来说，很容易识别出，不同CRD所处的层级和使用场景并不相同，以etcd为例，可能定义有代表服务主体本身的etcd CRD，以及表示etcd备份的backup CRD，两者客观上存在从属和依赖关系。所以，平铺的定义方式并不能很好的体现这种资源的层级和保证用户体验，在这个维度进行了细化，把CRD的功能和应用场景，按其角色分为以下两类：

- serviceEntity: 服务实体类资源，即提供如etcd实例CRD定义。
- operation：运维功能类，服务实例安装后以独立页签的形式展示，用户可以浏览已有的operation实例以及主动下发新的operation实例，如etcd backup CRD。

其中operation属于服务实体资源的能力（Capability）范畴，与serviceEntity相比只是行为方式上有所不同。

特别地，认为Helm服务的场景是一个最基本的，只包含一个serviceEntity资源定义的场景。Operator则可以包含多个不同角色的CSD，服务包在转换为OSC服务包后，用户需要对不同角色的CRD对应的CSD文件进行定制化的配置。

capabilityRequirements

- 配置能力引用需要在csd文件中声明capabilityRequirements。
- 能力引用只支持引入operator类型的平台能力，即平台提供的包括监控、日志等的公共能力。
- 可通过defaultPath配置文件路径，统一配置到**vendor目录**下。

capabilityRequirements配置样例


```
# 能力引用配置
capabilityRequirements:
  - apiVersion: osc.io/v1          # 引用平台监控capability
    kind: MonitorConfig
    defaultPath: vendor/monitor_config.yaml
  - apiVersion: osc.io/v1          # 引用平台日志capability
    kind: LogConfig
    defaultPath: vendor/log_config.yaml
```

表 2-6 capabilityRequirements 配置

参数名称	参数描述，	参数示例	必填
apiVersion	所引用的能力的配置版本，固定值osc.io/v1	osc.io/v1	是
kind	所引用的能力的配置类型，范围列表MonitorConfig|LogConfig	MonitorConfig	是
defaultPath	默认配置文件路径，路径为vendor目录	vendor/monitor_config.yaml	是

serviceEntityDependencies

- 配置服务依赖需要在csd文件中配置serviceEntityDependencies。
- 仅支持对公有服务的依赖，不支持依赖私有服务。
- defaultPath为依赖实例的配置文件，会覆盖服务自带配置。

serviceEntityDependencies配置样例

```
# 服务依赖
serviceEntityDependencies:
  - service: rdsService              # 依赖operator类型的RDS实例
    apiVersion: rds.io/v1alpha1
    kind: rds
    defaultPath: vendor/rds_provider.yaml
  - service: redisService            # 依赖Helm类型的Redis实例，apiVersion、kind不配置
    defaultPath: vendor/redis_service_values.yaml  # redis的values.yaml文件
```

表 2-7 serviceEntityDependencies 配置

参数名称	参数描述	参数示例	必填
service	所依赖的服务的名称	rdsService	是
apiVersion	所依赖的服务的配置版本	rds.io/v1alpha1	否
kind	所依赖的服务的配置类型	rds	否
defaultPath	默认配置文件路径，统一配置在vendor目录下。	参见样例	是

服务依赖配置文件样例：

- *rds_provider.yaml*

```
apiVersion: osc.io/v1
kind: rds
spec:
  ...
```

- *redis_service_values.yaml*

```
apiVersion: osc.io/v1
kind: redis
spec:
  global:
    imageRegistry: ""
    imagePullSecrets: []
  clusterDomain: cluster.local
  resources:
    limits: {}
    requests: {}
```

descriptors

descriptor字段用于配置UI控件，提升部署服务的用户体验，对服务参数进行控件增强，包括服务规格spec和服务状态status。该控件为可选项。

descriptors样例：

```
# UI控件
descriptors:
  spec:
  - description: "kafka实例个数"
    displayName: 集群大小
    path: {kafka.replicas}
    restriction:
      scenes:
        - CCE
    x-descriptors:
      - "urn:alm:descriptor:com.tectonic.ui:podCount"
  status:
  - description: "kafka实例个数"
    displayName: 集群大小
    path: {kafka.replicas}
    restriction:
      scenes:
       - CCE
    x-descriptors:
      - "urn:alm:descriptor:com.tectonic.ui:podCount"
```

表 2-8 descriptors 配置

参数名称	参数描述	必填
displayName:	控件显示名称	是
description	控件显示描述	是
path	控件对应字段，描述的对象上字段的点分隔路径，例如globalConfig1.field1	是
restriction	如果不同的部署的场景需要不同的控件配置，可配置此字段，支持CCE、UCS两种配置，分别表示云容器引擎和华为云UCS场景	否
x-descriptors	控件类型，具体参考**配置表单控件**了解相关配置	是

versionDefinition

实例版本定义配置是在csd文件中配置versionDefinition。

- versionDefinition配置tag样例如下。

```
versionDefinition:
  mode: tag
  path: spec.kafka.image
  tags:
    - 1.1.1
    - 2.2.2
    - 3.3.3
    - 9.9.9
```

- versionDefinition配置url样例如下。

```
versionDefinition:
  mode: url
  path: spec.kafka.image
  tags:
    - 1.1.1
    - 2.2.2
    - 3.3.3
    - 9.9.9
```

```
defaultConfiguration: |-{
      ....
      ...
      "spec": {
        "kafka": {
          "config": {
          },
          "image": "example:1.1.1"
          "nodeIps": [],
          "nodeSelector": {},
          "podAntiAffinity": true,
          "ports": {
            "bootstrapServer": 9092,
            "prometheus": 9404
          },
          "replicas": 3,
          "storageSize": "8Gi"
        }
      }
    }
```

表 2-9 versionDefiniton 配置

参数名称	参数描述	参数示例	必填
description	单击升级时的提示信息，知会用户进行实例升级的注意事项	您可以直接升级到最新版本，升级期间会发生5s之内的连接闪断，请确认应用程序具备重连机制	否

参数名称	参数描述	参数示例	必填
mode	实例的镜像格式，当前仅支持url、tag这两种格式	取值范围固定为[url、tag] ● mode配置为url，则会将上述所示defaultConfiguration字段下的"image"字段的值替换为"example:2.2.2" ● mode配置为tag后，则会将上述所示defaultConfiguration字段下的"image"字段的值替换为"2.2.2"	必填, 当mode配置之后，path和tags就成了必填项
path	实例镜像替换路径，位于defaultConfiguration字段下的值	例如，path值配置为"spec.kafka.image"，则需要替换的值是"example:1.1.1"	必填
tags	实例镜像tag版本号列表。建议使用引号，否则YAML会将类似2.0的tag版本号识别为浮点数。 **注意** 配置在tags下的实例版本号默认都是兼容的实例版本，即这些版本之间能进行互相升级。不兼容的版本号配置在tags列表下会造成升级完成之后实例功能有问题。	以实例镜像文件为test-x86_64-2.0.1.20210929123638.tar为例, 实例镜像文件通过docker load -i test-x86_64-2.0.1.20210929123638.tar执行完成之后，会在控制台回显如下信息： Loaded image: test-x86_64:2.0.1.20210929123638 则tag版本号为2.0.1.20210929123638，如果有多个版本镜像，则都可以通过此方式获取tag版本号， 然后将这些tag版本号配置在tags下面	必填

⚠ 注意

对于lifecycle文件中配置了upgrade字段的服务包，需要确保versionDefinition字段与被升级的operator的所管理的实例相兼容，否则会导致上传服务包或升级服务operator失败，具体规则如下：

1. 在上传服务包阶段，upgrade.replaces字段指定的被升级operator的versionDefinition字段为空时，允许上传服务包；不为空时，新的versionDefinition.tags需要与其有交集才能发布。

2. 在服务operator升级阶段，被升级的operator当前管理的实例版本必须都在新的versionDefinition.tags列表中，否则不允许升级，需要确认新版本的operator是否能够管理这些实例，如果可以，则修改versionDefinition后重新上传服务包，否则需要删除不兼容的实例后再升级。

示例：csd.yaml完整示例，以kafka为例。

```
apiVersion: osc.io/v1beta1
kind: CustomServiceDefinition
metadata:
  name: kafka-csd

spec:
  # 通过CRDRef将此CSD跟同一文件夹下的某个CRD相关联
  CRDRef:
    apiVersion: apiextensions.k8s.io/v1beta1
    kind: CustomResourceDefinition
    metadata:
      name: kafkas.osc.huawei.com

  # CRD显示名字
  displayName: kafka

  # CRD 描述
  description: kafka Description

  # CRD的角色
  role: serviceEntity

  # 公共能力引用（只针对平台能力引用）
  capabilityRequirements:
    - apiVersion: osc.io/v1                    # 引用平台监控能力
      kind: MonitorConfig
      defaultPath: vendor/monitor_config.yaml
    - apiVersion: osc.io/v1                  # 引用平台日志能力
      kind: LogConfig
      defaultPath: vendor/log_config.yaml



  # 服务依赖
  serviceEntityDependencies:
    - service: rdsService
      apiVersion: rds.io/v1alpha1            # 依赖operator格式的RDS
      kind: rds
      defaultPath: vendor/rds_provider.yaml
    - service: redisService                  # 依赖Helm格式的redisService
      defaultPath: vendor/redis_service_values.yaml # redis的values.yaml文件

  # UI控件
  descriptors:
    spec:
      - description: ""
        displayName: ""
        path: {crd-name.replicas}
        x-descriptors:
```

```
      - "urn:alm:descriptor:com.tectonic.ui:podCount"
  status:
    - description: ""
      displayName: ""
      path: {crd-name.replicas}
      x-descriptors:
        - "urn:alm:descriptor:com.tectonic.ui:podCount"

# CRD对应的CR样例
defaultConfiguration: |-{
      "apiVersion": "osc.huawei.com/v1",
      "kind": "Kafka",
      "metadata": {
        "annotations": {
          "osc.io/occupied-port": "135,139"
        },
        "name": "kafka-sample",
        "namespace": "kfk",
        "finalizers": [
          "kafka.finalizers.osc.huawei.com"
        ]
      },
      "spec": {
        "kafka": {
          "config": {
          },
          "image": "xxx/xx/aa:1.0"
          "nodeIps": [],
          "nodeSelector": {},
          "podAntiAffinity": true,
          "ports": {
            "bootstrapServer": 9092,
            "prometheus": 9404
          },
          "replicas": 3,
          "storageSize": "8Gi"
        }
      }
    }

# 服务实例升级
versionDefinition:
  description: 您可以直接升级到最新版本，升级期间会发生5s之内的连接闪断，请确认应用程序具备重连机制
  mode: url
  path: spec.image
  tags:
    - "2.7.0"
    - "2.7.1"
    - "2.7.2"
```

2.2.3.4 vendor 目录

公共能力目录：{OSC-Package}/manifests/vendor，用于存放平台提供的能力配置文件，用于manifests目录中的CRD引用。

公共能力目录结构：

```
{OSC-Package}/
├── metadata.yaml
├── manifests/                  #【必选】新增资源目录
│   ├── xxx_crd.yaml
│   └── xxx_csd.yaml
│   └── vendor/                 #【可选】公共能力目录
│     ├── monitor_config.yaml    #【可选】监控对接的默认配置，参见如何配置监控
│     ├── log_config.yaml        #【可选】日志对接的默认配置，参见对接配置日志
│     └── xxx_config.yaml        #【可选】未来支持更多公共能力，如自定义指标、弹性伸缩、多云部署等
├── raw/
├── lifecycle.yaml
└── README.md
```

表 2-10 vendor 目录

名称	文件	描述	类别	必选
监控配置文件	monitor_config.yaml	对接平台的指标采集能力，可配置指标采集的维度，指标名称，指标聚合规则等，详情参考**如何配置监控**。	文件	否
日志配置文件	log_config.yaml	对接平台的日志采集能力，可配置日志采集路径，详情参考**对接配置日志**。	文件	否

2.2.4 第三方目录 Raw

第三方目录：{OSC-Package}/raw。raw目录下会包含原始Helm包的内容。其他类型服务包不存在该目录。

```
{OSC-Package}/
├── metadata.yaml          #【必选】元数据文件
├── manifests/             #【必选】资源目录
├── raw/                   #【可选】第三方目录，支持存放helm原始包
├── lifecycle.yaml         #【必选】生命周期文件
└── README.md              #【可选】使用说明文件
```

Helm包可通过oscctl工具方便地转成OSC服务包，也可通过手工转换。其中Helm规范参见**Helm规范**。

- Helm原始包结构

```
{Helm}/
├── Chart.yaml
├── values.yaml
├── values.schema.json
├── charts/
└── templates
    └── xxx.yaml
```

- 经转换后的OSC服务包结构

```
{OSC-Package}/
├── metadata.yaml                #【必选】新增元数据文件
├── manifests/                   #【必选】新增资源目录
│   ├── helmrelease_crd.yaml     #【必选】创建HelmRelease CRD文件
│   └── helmrelease_csd.yaml     #【可选】新增资源增强配置
├── raw/                         #【可选】第三方目录
│   ├── Chart.yaml
│   ├── values.yaml
│   ├── values.schema.json
│   ├── charts/
│   └── templates
│       └── xxx.yaml
├── lifecycle.yaml               #【必选】生命周期文件
└── README.md                    #【可选】使用说明文件
```

- 转换说明
  - 新增第三方目录raw，将Helm原始包内容全部放到该目录中。
  - 新增元数据文件metadata.yaml，该文件内容包含Chart.yaml，具体规范参见**元数据Metadata**。
  - 新增资源目录manifests，根据Chart.yaml的name创建对应的CRD和CSD。
    - 新增自定义资源文件helmrelease_crd.yaml，该文件以下划线命名，由工具生成，无需改动。参见**HelmRelease CRD示例**。

    - 新增自定义服务文件helmrelease_csd.yaml，可按需配置相关服务能力。
  - 特别说明
    - Helm类型包转成OSC服务包与Operator类型不相同。转换过程中会根据Helm Chart name自动创建对应的CRD，版本试用版v1。
    - CRD中group固定值helm.osc.huawei.com，kind对应chart name。
    - 如果metadata.yaml中修改了服务包名、版本等内容，需要同步修改raw内的helm包中Chart.yaml内容。

- HelmRelease CRD示例（crd.yaml）

```
apiVersion: apiextensions.k8s.io/v1  # 默认创建v1版本的CRD
kind: CustomResourceDefinition
metadata:
  name: rediss.helm.osc.huawei.com   # 命名规则:{sepc.names.plural}.{spec.group}
spec:
  group: helm.osc.huawei.com        # 固定值，helm.osc.huawei.com
  names:
    kind: RedisHelmRelease          # 命名规则：{chartname}+HelmRelease
    listKind: RedisHelmReleaseList  # 命名规则：{spec.names.kind}+List
    singular: redis                 # {chartname}的小写形式
    plural: rediss                  # singular的复数
  scope: Namespaced
  version: v1
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          properties:
            apiVersion:
              description: ''
              type: string
            kind:
              description: ''
              type: string
            metadata:
              properties:
                name:
                  type: string
              type: object
            spec:
              description: ''
              properties:
                values:
                  description: Values defines the running options of helm release
                  type: object
                  x-kubernetes-preserve-unknown-fields: true
              type: object
          type: object
```

2.2.5 生命周期 Lifecycle

生命周期文件：{OSC-Package}/lifecycle.yaml，描述当前OSC服务包的安装，升级方式信息。

- lifecycle.yaml文件样例

```
## 描述该OSC服务包的声明周期相关信息
install:
  strategy: deployment
  clusterPermissions:
    - rules:
```



```
      - apiGroups:
        - ""
        resources:
        - ""
        verbs:
        - ""
    serviceAccountName: sa
 deployments:
  - name: example-operator
    spec:
      template:
        metadata:
        spec:
upgrade:
 replaces: 0.0.1
 skips:
  - 0.0.2
  - 0.0.3


operations:
 logpath: /var/paas/sys/log/operator/xxx-operator.log
```

表 2-11 Lifecycle 配置

参数名称	参数描述	参数示例	必填
install.strategy	安装方式，目前仅支持deployment方式	deployment	否
install.clusterPermissions	安装时需要的集群级别的权限	Kubernetes RABC规范	否
install.deployments	安装时的内容	Kubernetes Deployment规范	是
upgrade.replaces	升级基线版本	0.0.1	否
upgrade.skips	可以跳过升级的版本列表 **说明** **举例说明：** 假设已经发布0.0.1, 0.0.2, 0.0.3, 0.0.4, 0.0.5五个版本，其中0.0.2配置replaces: 0.0.1，0.0.5配置replace: 0.0.1及skips: 0.0.2。 ● 0.0.2和0.0.5版本都配置基线版本为0.0.1，所以已经部署的0.0.1版本可以升级到0.0.2和0.0.5版本。 ● 0.0.5版本因为配置了跳过0.0.2版本并且基线版本为0.0.1，所以已经部署的0.0.2版本无法升级到0.0.5版本，但是可以先将0.0.2版本回退到0.0.1版本后再次升级到0.0.5版本	参见样例	否
operations	运维操作配置，以注解方式定义	参见样例	否
operations.logpath	日志目录	参见样例	否

⚠ 注意

对于lifecycle文件中配置了upgrade字段的服务包，需要确保该服务operator能够管理的实例版本（由CSD文件中的versionDefinition字段规定）与被升级的operator所能管理的实例版本相兼容，否则会导致上传服务包或升级服务operator失败。具体规则请参见**versionDefinition**。

2.2.6 README

使用说明文件：{OSC-Package}/README.md，描述该服务包的功能介绍、使用说明等信息。按照下面一级目录编写。

```
<!--Markdown格式，按照以下目录结构-->
# 介绍

# 功能

# 版本列表

# 服务架构

# 快速上手
<!-- 可提供链接 或者 文字-->

# 许可证
```

说明

使用说明文件中引用的文件或者文档链接尚未设计，作为后续规划。

2.3 开源服务包规范

说明

- OSC服务规范同时支持当前业界使用较为广泛的两种开发服务规范：Helm服务规范和Operator-Framework服务规范。
- OSC服务平台支持部署上述两种开源的服务包，但是如需同时使用OSC对外提供的运维和表单控件能力，则需要将部署包转换为符合OSC规范格式的服务包，工具的转换使用可以参考章节**osccl工具使用**。

2.3.1 Operator 规范

Operator包含三类核心文件：

- 软件包清单：package.yaml
- 集群服务版本：csv.yaml
- 自定义资源：crd.yaml

```
# Opeartor结构
{Operator-Package}
├─ xxx.package.yaml          #【必选】软件包清单
└─ {version}                 #【必选】version目录，例如21.7.1
   ├─ xxx_csv.yaml           #【必选】ClusterServiceVersion集群服务版本文件，可多个。
   └─ xxx_crd.yaml           #【必选】CustomResourceDefinition自定义资源定义文件，可多个。
```

更多介绍参见官网**Operator Framework**。

2.3.1.1 接入规范说明

开源OLM服务（operator-framework）服务支持自定义配置服务场景、支持架构等配置，其描述说明定义在clusterserviceversion.yaml中，具体样例如下：

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: example-operator.v0.0.3
  namespace: test
  annotations:
    scenes: UCS,CCE
    source: ISV
    architecture: x86_64,aarch64
    categories: Database
    devices: CPU,GPU,NPU
    industries: education,media
spec:
  apiservicedefinitions: {}
  skips:
  - example-operator.v0.0.2
  replaces: example-operator.v0.0.1
  customresourcedefinitions:
  ...
```

参数名称	参数描述	参数示例	必选
source	包来源，取值范围：{ISV, OpenSource, HuaweiProvided}，分别表示独立服务供应商、开源服务、华为自有服务，仅支持配置一个值。默认值为OpenSource	OpenSource	否
architecture	支持架构列表，取值范围：{x86_64, aarch64}，默认值为x86_64	x86_64,aarch64	否
categories	分类列表，范围不限，如"AI, Database"，注意，当前同一个服务不同版本须配置相同内容，一旦配置后当前不支持修改	Database	否
devices	实例运行需要使用的硬件设备列表，取值范围：{CPU, GPU, NPU}，默认值为CPU	CPU,GPU	否
industries	行业列表，范围不限，如"education,media"	education	否
scenes	部署平台列表，取值列表：{CCE，UCS}，分别表示CCE平台和UCS平台，大小写敏感，默认值为CCE。	CCE,UCS	否

2.3.2 Helm 规范

Helm的包格式被称为chart，描述Kubernetes相关资源的文件集合。chart包含：

- Chart.yaml
- values.yaml

- values.schema.json
- charts目录
- templates目录

```
{Helm}/
├─ Chart.yaml          #【必选】包含了chart信息文件，包括api版本、名称、描述、图标、类型、注释
├─ values.yaml         #【必选】参数值文档，为templates下的文档配置参数值
├─ values.schema.json  #【可选】一个使用JSON结构的values.yaml文件
├─ charts/             #【可选】包含chart依赖的其他chart
└─ templates           #【必选】模板目录,当和values 结合时，可生成有效的Kubernetes manifest文件
   └─ xxx.yaml         #【可选】Helm包的Kubernetes manifest若干文件
```

更多介绍参见官网**Helm**。

上传至OSC的helm包中，Chart.yaml文件的icon字段不生效，仅支持以图片形式配置logo，配置方式为：在与Chart.yaml同级的目录中增加logo.png（或jpg，svg后缀）命名的图片。

2.3.2.1 接入规范说明

开源helm服务支持自定义配置服务场景、支持架构等配置，其描述说明定义在Chart.yaml中，具体样例如下：

```
apiVersion: v1
description: "nginx \n"
maintainers:
- email: support@rancher.com
  name: Rancher Labs Support
name: helm-instance-update
version: 1.0.2
annotations:
  scenes: CCE,UCS
  source: ISV
  architecture: x86_64,aarch64
  categories: database
  devices: CPU,GPU,NPU
  industries: education,media
```

参数名称	参数描述	参数示例	必选
source	包来源，取值范围：{ISV, OpenSource, HuaweiProvided}，分别表示独立服务供应商、开源服务、华为自有服务，仅支持配置一个值。默认值为OpenSource	OpenSource	否
architecture	支持架构列表，取值范围：{x86_64, aarch64}，默认值为x86_64	x86_64,aarch64	否
categories	分类列表，范围不限，如"AI, Database"，注意，当前同一个服务不同版本须配置相同内容，一旦配置后当前不支持修改	Database	否
devices	实例运行需要使用的硬件设备列表，取值范围：{CPU, GPU, NPU}，默认值为CPU	CPU,GPU	否
industries	行业列表，范围不限，如"education,media"	education	否

参数名称	参数描述	参数示例	必选
scenes	部署平台列表，取值列表：{CCE，UCS}，分别表示CCE平台和UCS平台，大小写敏感，默认值为CCE。	CCE,UCS	否

3 服务接入

说明

- 本章节主要介绍如何将一个现有的服务包接入OSC平台，并可以使用OSC附加功能，有效增加自身服务的运维能力。如开发者需要从头开始编写程序制作应用，可以参考附录**如何从零开始开发Operator**。
- OSC平台支持OSC服务规范，开源Helm或Operator-Framework的规范服务，开发者可参考**开源服务包规范**确保服务包符合helm或者operator-framework规范，或参考**制作服务包**，生成符合OSC规范的服务包。
- 服务接入完成后，可以参考《 OSC 服务使用者指南 》了解如何登录OSC平台进行验证。

3.1 服务能力介绍

简介

云原生服务中心（ Operator Service Center，OSC ）能够提供大量开箱即用的云原生服务，支持服务的开发、发布、订阅、部署、升级、更新等，其面向对象包括服务开发者、服务使用者和服务提供商。本文旨在帮助开发者如何接入OSC平台并使用OSC能力。

优势

- **更高效的利用华为云基础设施**

  OSC作为开发者与华为云基础设施的桥梁，提供了统一的对接方式，开发者可以通过配置文件赋予服务云原生能力，如全域分发、弹性伸缩、智能调度、监控运维等。

- **一次开发，标准、快速交付**

  基于OSC开发规范开发的云服务，在公有云验证交付后，云服务提供商无需再针对云容器引擎、华为云UCS部署平台适配，可以随OSC拓展到云容器引擎、华为云UCS应用部署平台，使交付更标准，更快速，为生态伙伴的产品赋予了批量复制的能力。

- **兼容开源，快速发布成熟产品至云市场**

  OSC开发规范完全兼容Operator Framework以及Helm社区标准，其中Operator Framework支持版本v0.19.0+，Helm支持V3和V2的语法。满足Operator Framework以及Helm规范的服务无需修改，发布到云市场后可以通过OSC直接部署到用户的云容器引擎、华为云UCS。

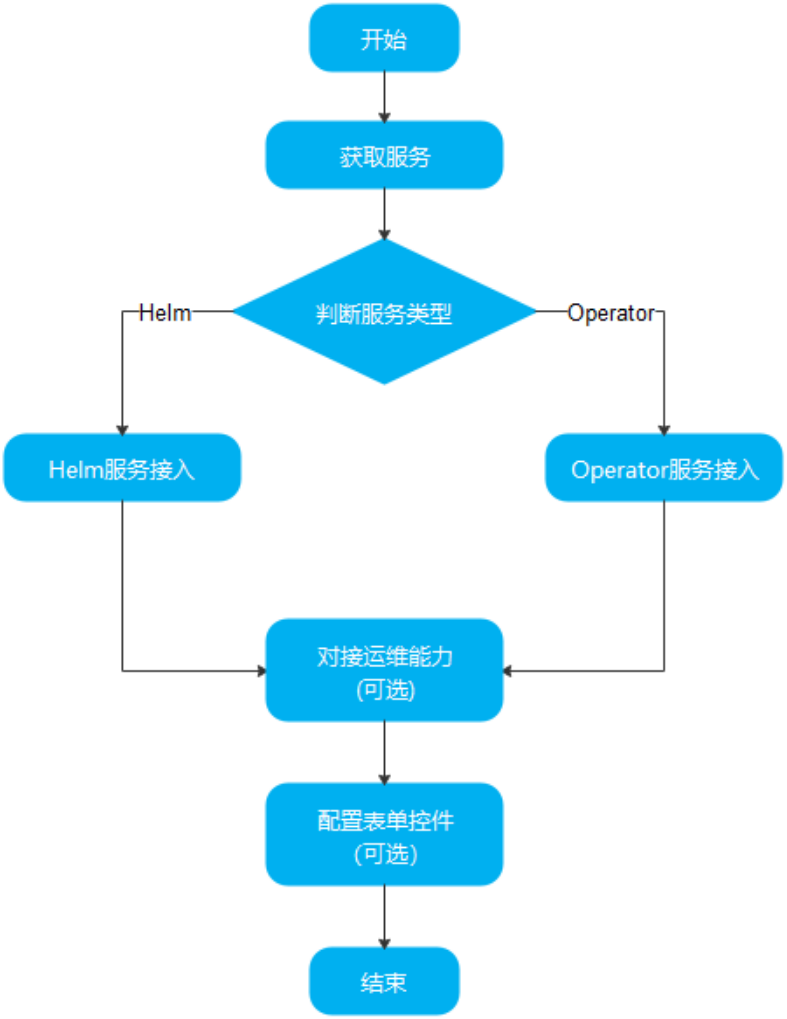
- **扩展开源，赋予服务平台高级能力**
  - 提供动态UI控件，支持表单方式创建实例。

    动态UI指根据服务包相关配置，动态渲染出表单，用户可以编辑表单创建服务实例。
  - 动态UI控件支持实例运行时状态显示。

    服务已配置动态UI控件，部署服务实例后，实例详情页面可以展示实例的运行详情，比如运行时参数，并支持配置修改。
  - 提供扩展目录，支持对接运维能力。

    OSC支持开发者在扩展目录中添加配置文件对接运维能力。基于该规范开发的服务在部署时，OSC会识别并解析配置文件中的内容，赋予服务对应的功能。

服务接入流程图



3.2 Helm 服务接入 OSC 平台

前置条件

阅读本章节前，用户需要拥有一个符合Helm规范的服务包，Helm服务规范可以参考**Helm规范**，下面以helm开源服务grafana为例进行详细说明。

接入流程

步骤1 用户参考附录**oscctl工具使用**中的链接下载oscctl工具oscctl-22.8.18.tar.gz，并放在linux执行机中。

步骤2 执行下面命令解压服务。

```
# 解压工具包
$ tar -xvzf oscctl-22.8.18.tar.gz
oscctl-22.8.18/
oscctl-22.8.18/linux-x86_64/
oscctl-22.8.18/linux-x86_64/oscctl
oscctl-22.8.18/win-amd64/
oscctl-22.8.18/win-amd64/oscctl.exe

$ chmod +x oscctl-22.8.18/linux-x86_64/oscctl

$ ./oscctl-22.8.18/linux-x86_64/oscctl convert -i grafana-5.5.7.tgz -o grafana-5.5.7.zip
Successfully converted original package to OSC package!
```

步骤3 查看转换后的服务包结构。

```
grafana-5.5.7
├── lifecycle.yaml              # operator生命周期文件，helm类型内容为空
├── manifests
│   ├── helmrelease_crd.yaml    # osc定义helm release的crd服务
│   └── helmrelease_csd.yaml    # osc附加能力定义文件，需和crd文件联动使用
├── metadata.yaml
└── raw                         # raw目录下存放的是原始的grafana helm包
    ├── Chart.yaml
    ├── templates
    │   ├── NOTES.txt
    │   ├── _helpers.tpl
    │   ├── deployment.yaml
    │   ├── hpa.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   ├── serviceaccount.yaml
    │   └── tests
    │       └── test-connection.yaml
    ├── README.md
    └── values.yaml
```

步骤4 生成的服务包如果需要配置OSC平台运维功能，请参考**对接运维能力(可选)**；如果不需要配置OSC平台运维功能，请直接查看《 OSC服务使用者指南 》了解如何上传服务包进行使用。

----结束

说明

oscctl工具同样支持直接在windows平台上运行，需要使用windows终端进入，执行oscctl-22.8.18/win-amd64/oscctl.exe命令，参考上面的步骤操作即可。

3.3 Operator 服务接入 OSC 平台

前置条件

阅读本章节前，用户需要拥有一个符合Operator规范的服务包，Operator服务规范可以参考**Operator规范**，下面以Operator开源服务datadog-operator为例进行详细说明。

接入流程

步骤1 用户参考附录**oscctl工具使用**中的链接下载oscctl工具oscctl-22.8.18.tar.gz，并放在linux执行机中。

步骤2 执行下面命令解压服务。

```
# 解压工具包
$ tar -xvzf oscctl-22.8.18.tar.gz
oscctl-22.8.18/
oscctl-22.8.18/linux-x86_64/
oscctl-22.8.18/linux-x86_64/oscctl
oscctl-22.8.18/win-amd64/
oscctl-22.8.18/win-amd64/oscctl.exe

$ chmod +x oscctl-22.8.18/linux-x86_64/oscctl

$ ./oscctl-22.8.18/linux-x86_64/oscctl convert -i datadog-operator.zip -o datadog-operator-0.3.2.zip
Successfully converted original package to OSC package!
```

步骤3 查看转换后的服务包结构。

```
datadog-operator-0.3.2
├── lifecycle.yaml              # operator生命周期文件，helm类型内容为空
├── manifests
│   ├── datadogmetric_crd.yaml    # datadogmetric crd文件
│   ├── datadogmetric_csd.yaml    # datadogmetric crd对应的csd文件
│   ├── datadogagent_crd.yaml     # datadogagent crd文件
│   └── datadogagent_csd.yaml     # datadogagent crd对应的csd文件
├── metadata.yaml
└── raw                         # raw目录下存放的是原始的operator包
    ├── 0.0.3
    │   ├── datadog-operator.v0.3.2.clusterserviceversion.yaml
    │   ├── datadoghq.com_datadogagents_crd.yaml
    │   └── datadoghq.com_datadogmetrics_crd.yaml
    └── datadog-operator.package.yaml
```

步骤4 生成的服务包如果需要配置OSC平台运维功能，请参考**对接运维能力(可选)**；如果不需要配置OSC平台运维功能，请直接查看《 OSC服务使用者指南 》了解如何上传服务包进行使用。

----结束

3.4 对接运维能力(可选)

3.4.1 服务包扩展目录

说明

运维能力是在OSC服务规范基础上产生的能力，因此如果需要对接运维能力，需要先确保服务包的格式转换为OSC自身的规范才行，默认的开源规范当前只支持安装部署等基本功能。

运维能力扩展

为了保证适用OSC的服务规范，将使用oscctl工具作为创建、转换OSC服务包，并且统一服务包格式。使用oscctl create命令会在manifests目录下生成一个vendor文件夹，该文件夹下存放日志、监控的扩展内容，扩展目录结构如下：

```
# 以hwfka-operator-package为示例
hwfka-operator-package
├─ metadata.yaml
├─ manifests/
│  ├─ hwfka_crd.yaml
│  └─ hwfka_csd.yaml
│  └─ vendor/
│     ├─ monitor_config.yaml
│     ├─ logs_config.yaml
└─ lifecycle.yaml

# 以helm-package为示例
helm-package
├── lifecycle.yaml
├── manifests
│   ├── helmrelease_crd.yaml
│   └── helmrelease_csd.yaml
│   └── vendor
│       ├── log_config.yaml
│       └── monitor_config.yaml
├── metadata.yaml
└── raw
    ├── Chart.yaml
    ├── templates
    │   ├── NOTES.txt
    │   ├── _helpers.tpl
    │   ├── deployment.yaml
    │   ├── hpa.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   ├── serviceaccount.yaml
    │   └── tests
    │       └── test-connection.yaml
    └── values.yaml
```

参数	描述	类别（路径/文件）	是否必选	OSC是否已经支持
hwfka-operator-package	通过oscctl创建出来的标准服务包。	路径	是	是
vendor	OSC扩展的目录，与平台高级能力对接的配置文件都放在这个路径下。	路径	否	是
monitor_config.yaml	对接平台的指标采集能力，可配置指标采集的维度，指标名称，指标聚合规则等，详情参考**如何配置监控**。	文件	否	是
log_config.yaml	对接平台的日志采集能力，可配置日志采集路径**对接配置日志**。	文件	否	是

参考如下步骤可以进行扩展目录文件配置：

在配置了扩展能力后，需要在csd.yaml文件中添加配置引用才能生效，新增配置如下：

```
# 公共能力引用（只针对平台能力引用）
# MonitorConfig：对接监控
# LogConfig：对接日志
capabilityRequirements:
  - apiVersion: osc.io/v1
    kind: MonitorConfig
    defaultPath: vendor/monitor_config.yaml
  - apiVersion: osc.io/v1
    kind: LogConfig
    defaultPath: vendor/log_config.yaml
```

3.4.2 如何配置监控

使用前提

对接监控，需要在csd.yaml文件中配置能力插件依赖功能声明，具体参考**服务包扩展目录** capabilityRequirements示例

对接原理

开源的**Prometheus**是目前业界广泛应用的监控指标采集系统，其服务端可作为时间序列数据库，采集客户端应用不同时间周期上报的监控指标数据并存储，还可用来查询数据和图形化展示数据趋势。其针对不同的应用，提供了多种客户端exporter实现，用来上报监控指标，如：**https://github.com/prometheus/jmx_exporter**。

华为云容器洞察引擎CIE服务兼容**Prometheus**提供应用指标采集和监控服务，开发者只需要配置映射规则，OSC平台使用内置的ops-operator插件来匹配工作负载（包括Deployment、StatefulSet或DaemonSet）资源的标签，自动发现需要上报监控指标数据的应用，并通过sidecar方式，为应用监控指标数据注入instance_id字段，用以区分不同实例的指标，并分层分级生成监控仪表盘，图形化展示监控数据。

对接流程

步骤1 获取OSC规范开发的服务包，如果只有原生的helm或OperatorFramework包，可以参考章节**服务包转换**进行转换。

步骤2 在目录manifests/vendor下新增配置文件monitor_config.yaml

配置具体样例如下：

```
apiVersion: osc.io/v1
kind: MonitorConfig
metadata:
  name: helm-release-1ughua-uxkrcb
spec:
  monitorConfigItems:
    - labelSelector:
        monit-test: monit-test
      prometheus:
        exporterPort: 9404
        displayOptions:
          levels:
            - name: "Pod"
              localeID: "ops.monitor.kafka.level.pod-slave"
              groupLabel: pod
```

```
selectedMetrics:
  - "kafka_server_ReplicaManager_Value"
metrics:
  - title: "Leader Count"
    localeID: "ops.monitor.kafka.memory.usage"
    expr: 'kafka_server_ReplicaManager_Value{name="LeaderCount",kubernetes_pod="$",}'
    unitType: ""
```

斜体字的部分则需要根据服务的实际情况进行适配更改。具体配置参数可见下面表格说明。

参数名	含义	类型	是否必填	备注
apiVersion	配置版本	string	是	固定为osc.io/v1
kind	配置种类	string	是	固定为MonitorConfig
metadata.name	配置名	string	否	监控配置默认名称 **须知** 目前对于monitor_config.yaml文件，一个服务包内有且仅有一个，目前该名字没有实际业务意义。在服务包下发时会根据服务包名称等，由OSC自动生成一个在Kubernetes集群该Namespace下面唯一的名字。
labelSelector	匹配实例的标签	string数组	是	同一operator CRD下也可能存在多个工作负载资源，需要通过label selector与应用实例label相关联，从而才可过滤出准确应用的数据并在界面展示。
exporterPort	普罗接口	integer	是	服务已实现的prometheus指标采集端口，此接口为容器运行时内部监听的端口。 **说明** 需要使用metrics形式进行相关指标的上报，即通过URL http://{Pod IP}:{exportPort}/metrics的形式对相关监控指标上报。具体指标书写请参考如下链接或步骤4中内容： https://prometheus.io/docs/concepts/metric_types/ https://prometheus.io/docs/concepts/data_model/ https://prometheus.io/docs/prometheus/latest/configuration/configuration/
levels.name	监控维度名	string	否	显示为不同层级的页签名称，如：Instance、Pod等。
localeID	指标国际化展示	string	否	暂未启用，后续支持国际化展示。

参数名	含义	类型	是否必填	备注
groupLabel	该层级的聚合参数	string	否	如果填写，则自动在expr中注入根据该labelkey+占位符的筛选条件。 **说明** 对于聚合条件为pod的，需要设置为pod。
selectedMetrics	指标数组	string数组	否	与groupLabel配套，用于确认groupLabel指定的key值可以通过哪些指标获取全部value。
title	监控名	string	否	监控图显示名称，长度不超过64字节
expr	监控语句	string	否	原生普罗监控语句promsql
unitType	监控数据单位	string	否	unitType可以是""（默认无单位，按原数据展示），byte（根据数量展示b，kb，mb，tb），time（us，ms，s，min，hour，day，month），条，个，percentage（%）等等，长度不超过10字节。

步骤3 在manifests/xxx_csd.yaml文件中增加引用监控能力的配置。

```
apiVersion: osc.io/v1beta1
kind: CustomServiceDefinition
metadata:
  name: helmrelease-csd
spec:
  CRDRef:
    apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
  capabilityRequirements:
    - apiVersion: osc.io/v1
      kind: MonitorConfig
      defaultPath: vendor/monitor_config.yaml
```

步骤4 下面以一个实际监控指标为样例进行展示

```
apiVersion: osc.io/v1
kind: MonitorConfig
metadata:
  name: hwfka-monitor
spec:
  monitorConfigItems:
    - labelSelector:
        # ops operator 根据 label 决定 Deployment、StatefulSet或DaemonSet 是否注入 sidecar
        osctest.huawei.com/service_provider: hwfka
        osctest.huawei.com/component: broker
      crd: hwfkas.osctest.huawei.com
      prometheus:
        exporterPort: 9404
        displayOptions:
          levels:
              - name: "实例"
                localeID: "ops.monitor.redis.level.instance"
                metrics:
                  - title: "Topic 总数"
                    # 实例中的 topic 总数
```



```
                localeID: "ops.monitor.redis.memory.usage"
                expr: 'sum(kafka_controller_KafkaController_Value{name="GlobalTopicCount"})'
                unitType: ""
              - title: "分区总数"
                # 实例中的分区总数（创建 topic 时设置的分区数，即仅 leader 数）
                localeID: "ops.monitor.redis.memory.usage"
                expr: 'sum(kafka_controller_KafkaController_Value{name="GlobalPartitionCount"})'
                unitType: ""
              - title: "副本总数"
                # 实例中的副本总数（含 leader）
                localeID: "ops.monitor.redis.memory.usage"
                expr: 'sum(kafka_server_ReplicaManager_Value{name="PartitionCount"})'
                unitType: ""
            - name: "Topic"
              localeID: "ops.monitor.redis.level.pod-slave"
              # console 通过查询语句 group(kafka_server_BrokerTopicMetrics_Count{topic!=""}) by (topic) 获取所有 Topic 列表
              groupLabel: topic
              selectedMetrics:
                - 'kafka_server_BrokerTopicMetrics_Count{topic!=""}'
              metrics:
                - title: "生产消息数量（条）"
                  # 每个 topic 的生产消息数量
                  localeID: "ops.monitor.redis.memory.usage"
                  expr: 'sum(kafka_server_BrokerTopicMetrics_Count{name="MessagesInPerSec"}) by (topic)'
                  unitType: ""
                - title: "生产消息速率（条/秒）"
                  # 每个 topic 的生产速率消息数量（每秒）
                  localeID: "ops.monitor.redis.memory.usage"
                  expr: 'sum(kafka_server_BrokerTopicMetrics_OneMinuteRate{name="MessagesInPerSec"}) by (topic)'
                  unitType: ""
```

----结束

说明

服务需要对接监控能力，必须确保服务对外提供/metrics接口，并遵循prometheus规范将指标数据通过此接口对外提供。

3.4.3 对接配置日志

使用前提

对接日志系统，需要在csd.yaml文件中配置能力插件依赖功能声明，具体参考**服务包扩展目录** capabilitiesrequirements示例。

对接原理

OSC内置了ops-operator插件，通过webhook的方式拦截标签匹配的工作负载（包括Deployment、StatefulSet或DaemonSet）资源，为其注入日志采集和上报功能，日志最终被采集上报至华为应用运维管理AOM。

如果需要对接日志系统，需要确保服务容器内部有输出日志文件，OSC服务会将日志文件挂载到外部供AOM采集上报，从而达到对接的效果。

对接流程

步骤1 获取OSC规范开发的服务包，如果只有原生的helm或OperatorFramework包，可以参考章节**服务包转换**进行转换。

步骤2 在目录manifests/vendor下新增配置文件log_config.yaml

配置规则如下所示：

```
apiVersion: osc.io/v1
kind: LogConfig
metadata:
  name: kafka-log
spec:
  logConfigItems:
    - labelSelector:
        app.kubernetes.io/component: kafka
        app.kubernetes.io/version: "20.9.3"
      logPaths:
        - /var/kafka/
        - /var/kafka/access/kafka-access.log
```

具体参数说明如下：

参数名	含义	类型	是否必填	备注
apiVersion	配置版本	string	是	固定为osc.io/v1
kind	配置种类	string	是	固定为LogConfig
metadata.name	配置名	string	否	日志配置默认名称 **须知** 目前对于log_config.yaml文件，一个服务包内有且仅有一个，目前该名字没有实际业务意义。在服务包下发时会根据服务包名称等，由OSC自动生成一个在Kubernetes集群该Namespace下面唯一的名字。
labelSelector	匹配实例的标签	string数组	是	同一operator CRD下也可能存在多个工作负载，需要通过label selector与应用实例label相关联。 此label要求和用户需要对接日志的工作负载中的label字段完全匹配，从而保证ops应用可以扫描到并进行绑定日志
logPaths	日志路径或文件	string数组	是	容器内日志路径，可填多个，但不允许存在包含关系。 日志内的路径支持目录和文件 目录：扫描此目录下的所有为log,out,trace结尾的日志并对接 文件：只对接指定的文件

步骤3 在manifests/xxx_csd.yaml文件中增加引用日志能力的配置。

```
apiVersion: osc.io/v1beta1
kind: CustomServiceDefinition
metadata:
  name: helmrelease-csd
spec:
  CRDRef:
    apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
  capabilityRequirements:
    - apiVersion: osc.io/v1
      kind: LogConfig
      defaultPath: vendor/log_config.yaml
```

步骤4 如下为一个具体的应用配置日志样例进行说明。

```
apiVersion: osc.io/v1
kind: LogConfig
metadata:
  name: hwfka-log
spec:
  logConfigItems:
    ## 同一operator CRD下也可能存在多个工作负载,
    ## 需要通过label selector将应用实例与相应的ops配置相关联
    - labelSelector:
        osc.huawei.com/service_provider: hwfka
        osc.huawei.com/component: broker
      crd: hwfkas.osctest.huawei.com
      logPaths:
        - /opt/kafka/logs
    - labelSelector:
        osc.huawei.com/service_provider: hwfka
        osc.huawei.com/component: zookeeper
      crd: hwfkas.osctest.huawei.com
      logPaths:
        - /opt/kafka/logs
    - labelSelector:
        osc.huawei.com/service_provider: hwfka
        osc.huawei.com/component: kafkamanager
      crd: hwfkas.osctest.huawei.com
      logPaths:
        - /opt/cmak-3.0.0.5/logs
```

----结束

3.4.4 对接实例升级能力

修改csd.yaml，声明实例版本定义信息。新增配置如下：

```
# 支持升级的镜像版本
versionDefinition:
  mode: url
  path: spec.image
  tags:
    - 0.0.1
    - 0.0.2
    - 0.0.3
```

具体配置参数说明可参考章节**versionDefinition**

3.5 配置表单控件

创建实例应用时，往往涉及大量的参数配置，为方便用户使用，OSC console提供了自动生成表单的能力。

在Kubernetes 1.8中，CRD定义基于OpenAPI v3的验证模式的能力。基于OpenAPI v3能力，可用于自动生成用于创建实例的表单。

3.5.1 控件类型

表单控件类型

1. 使用默认crd.yaml文件自动生成表单

由CRD中定义的openAPIV3Schema字段规定CR中可配置的参数以及这些参数的类型、范围，应用在创建CR的时候通过YAML文件指定这些参数。

2. 使用自定义csd.yaml文件生成表单

自动生成的创建表单已经很强大，但有时可能需要修改自定义表单的呈现方式。例如，可能需要对用户更友好的displayName、更简洁的描述，以及更改表单字段顺序，或排除一些非面向用户的字段。这就是“(x-descriptors)规范描述符”的作用。

3.5.2 表单控件

3.5.2.1 基础控件

OSC console依据crd.yaml文件可以自动生成Operator的创建表单。crd.yaml文件中的properties（openAPIV3Schema.properties.spec.properties）对象会自动映射到对应的UI组件。

目前提供的基础类型控件分别有string、number、boolean、object、array类型。

- **string控件**：满足一般的文本输入。
- **number控件**：满足一般的数字输入。
- **boolean控件**：开关控件，支持设置布尔值。
- **object控件**：支持结构体格式输入，具有多层嵌套能力。
- **array类型**：支持数组结构的输入，可以动态增删。

string 控件

一般场景的文本输入，支持校验规则的配置。

配置示例：

```
type: string
title: Extra RabbitMQ Configuration
description: Extra configuration to be appended to...
```

图 3-1 string 控件

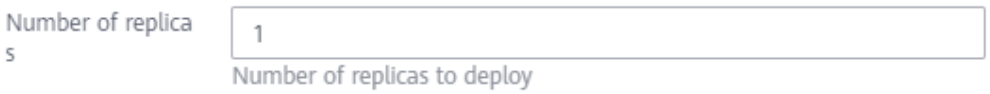
number 控件

一般场景的数字输入，支持设置数值的大小范围。

配置示例：

```
type: number
title: Number of replicas
description: Number of replicas to deploy
```

图 3-2 number 控件



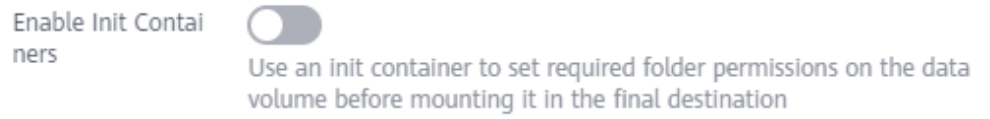
boolean 控件

boolean类型字段输入。

配置示例：

```
type: boolean
title: Enable Init Containers
description: Use an init container to set required...
```

图 3-3 boolean 控件



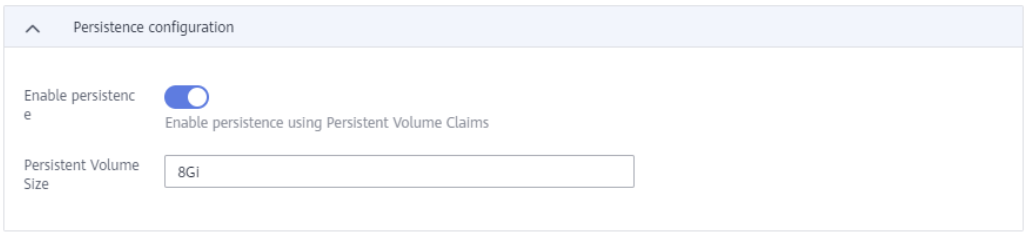
object 控件

结构体格式输入，支持子控件的嵌套。

配置示例：

```
type: object
title: Presistence configuration
properties:
  enablePersistence:
    type: boolean
    title: Enable Persistence
    description: Enable persistence using Persistent Volume Claims
  persistentVolumeSize:
    type: string
    title: Persistent Volume Size
    description:
```


图 3-4 object 控件



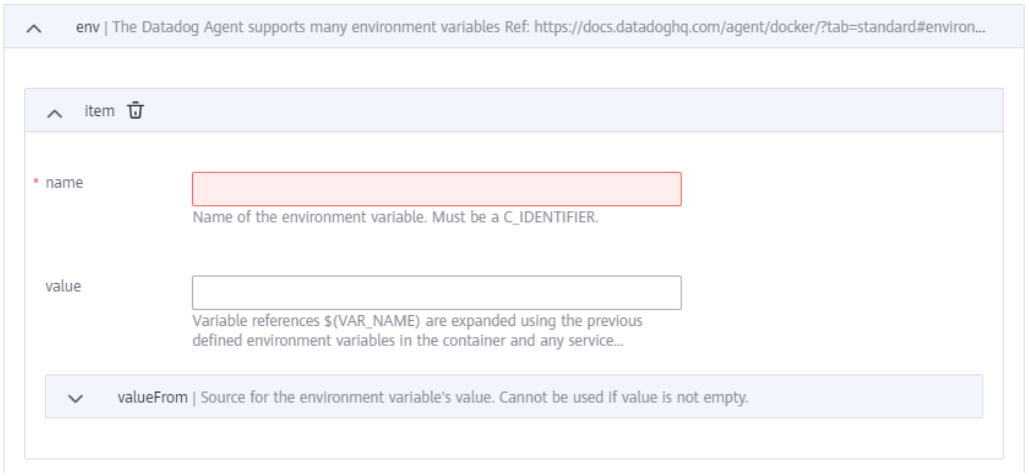
array 类型

满足数组类型的输入，支持设置最多项最少项等。

配置示例：

```
type: array
title: env
description: The DataDog ...
items:
  properties:
    name:
      type: string
      title: name
      description: Name of ...
    persistentVolumeSize:
      type: string
      title: Persistent Volume Size
      description:
    valueFrom:
      type: object
      properties:
        ...
```

图 3-5 array 类型



3.5.2.2 自定义控件

基于CRD自动生成表单的基础能力控件已经能很好的呈现表单输入，但是为了对用户更友好，更好的表达字段的含义以及隐藏一些非面向用户的字段，因此引入了x-descriptors描述符来定义自定义的控件。

x-descriptors描述符主要用于specDescriptors和statusDescriptors定义自定义控件。

specDescriptors:

- password(**密码输入框**)：支持密码类型字段输入。
- select(**下拉框**)：设置一些可选项，简化用户输入成本。
- imagePullPolicy(**镜像拉取策略**)：预置了K8S镜像拉取策略。
- k8sResourcePrefix(**K8S资源对象**)：支持设置资源类型并查询展示。
- podCount(**Pod数量**)：设置pod数量。
- fieldDependency(**显示联动**)：支持控件之间的依赖，控制显示隐藏。
- resourceRequirements(**资源配置**)：K8S原生资源规则配置，配置request和limit。
- hwDnatElbAccess(**公网访问**)：支持通过弹性公网IP访问实例。
- hidden(**字段隐藏**)：支持对基于CRD自动生成控件的字段表单隐藏。

statusDescriptors:

- text(展示文本): 实例的状态字段一般文本展示。
- link(访问链接): 公开实例的外部链接。

3.5.2.2.1 specDescriptors

specDescriptors用于描述自定义资源中规范字段的属性。每个字段应包含以下内容：

- displayName -字段的用户友好名称。
- description -字段表示内容的简单描述信息。
- path -对象中字段以点分隔的路径。
- restriction(可选)：指定控件使用场景。
- x-descriptors(可选) -字段能力的UI组件信息。

说明

配置Descriptor控件时，需要确保在csd.yaml中增加的path字段路径，以v1版本为例，需要和crd中的对应的openAPIV3Schema.properties.spec.properties内的path路径一致，否则会出现控件不生效的情况。

密码输入框

该控件支持先隐藏已输入的密码。

配置示例：

```
- path: password_test
  displayName: password_test
  description: password_test
  x-descriptors:
    - "urn:alm:descriptor:com.tectonic.ui:password"
```

图 3-6 密码输入框

图 3-7 单击显示密码后的效果

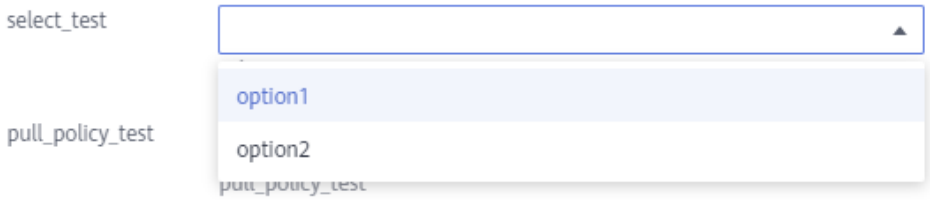
下拉框

支持设置枚举项，使用下拉框展示。

配置示例：

```
- path: select_test
  displayName: select_test
  description: "select_test"
  x-descriptors:
    - "urn:alm:descriptor:com.tectonic.ui:select:option1"
    - "urn:alm:descriptor:com.tectonic.ui:select:option2"
```

图 3-8 下拉框 UI



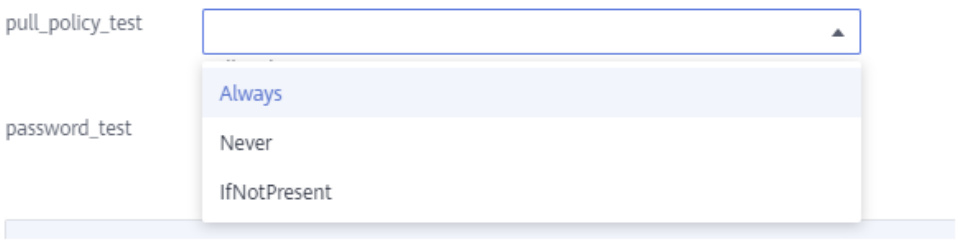
镜像拉取策略

预设的镜像拉取策略控件。

配置示例：

```
- path: pull_policy_test
  displayName: pull_policy_test
  description: "pull_policy_test"
  x-descriptors:
    - "urn:alm:descriptor:com.tectonic.ui:imagePullPolicy"
```

图 3-9 镜像拉取策略控件



K8S 资源对象

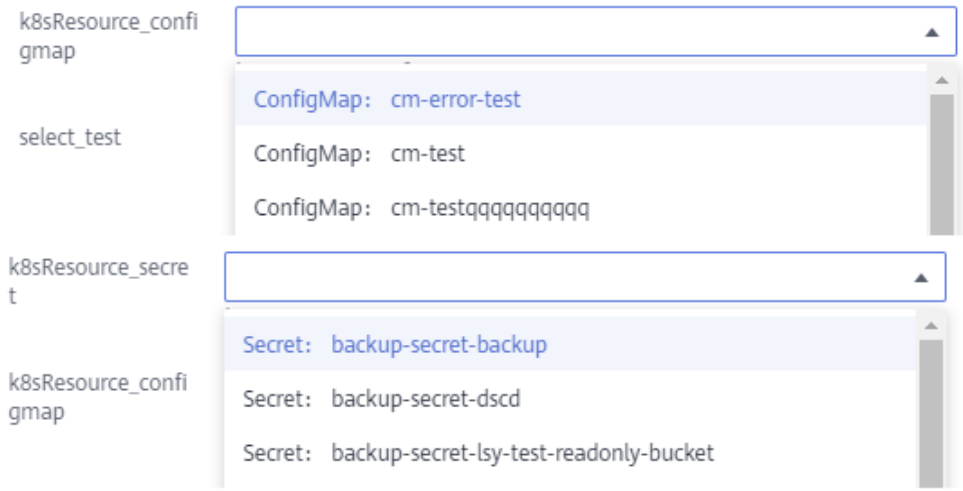
支持控件查询指定资源对象，目前支持configmap和secret。

配置示例：


```
- path: k8sResource_secret
  description: k8sResource_secret
  displayName: k8sResource_secret
  x-descriptors:
    - urn:alm:descriptor:io.kubernetes:Secret

- path: k8sResource_configmap
  description: k8sResource_configmap
  displayName: k8sResource_configmap
  x-descriptors:
    - urn:alm:descriptor:io.kubernetes:ConfigMap
```

图 3-10 K8S 资源对象



Pod 数量

预设的设置pod数量控件。

配置示例：

```
- path: advanceTest_podCount
  description: advanceTest_podCount
  displayName: advanceTest_podCount
  x-descriptors:
    - urn:alm:descriptor:com.tectonic.ui:podCount
```

图 3-11 Pod 数量控件

显示联动

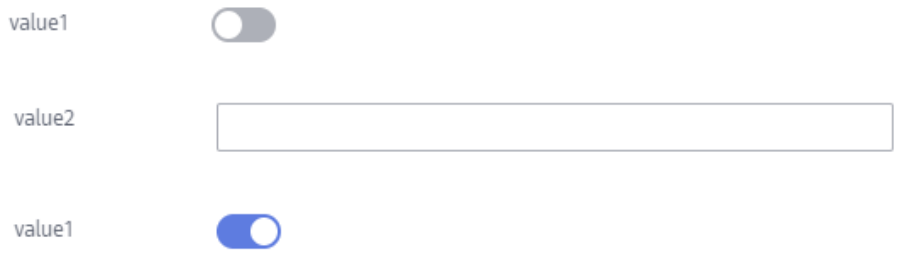
支持控件的展示联动。

配置示例：

```
- path: field_dependency_test.value2
  description: 依赖value1 开
```

```
displayName: value2
x-descriptors:
  - "urn:alm:descriptor:com.tectonic.ui:fieldDependency:field_dependency_test.value1:true"
```

图 3-12 控件显示联动



资源配置

和K8S配套的资源控件，可以设置limit和request。

配置示例：

```
- path: resources
  displayName: 单节点规格
  description: "单节点Redis的内存规格，因系统开销，实际使用内存是所填数值的1.25倍"
  x-descriptors:
    - "urn:alm:descriptor:com.tectonic.ui:resourceRequirements"
```

公网访问

支持通过弹性公网IP访问实例。

配置示例：

```
- description: 开启公网访问后，可通过弹性公网IP访问Redis，集群模式暂不支持。开启公网访问的同时建议开启密码认证，保证安全
  displayName: 公网访问
  path: externalAccess
  restriction:
    scenes:
      - CCE
  x-descriptors:
    - "urn:alm:descriptor:com.huawei.ui:hwDnatElbAccess"
```

其中restriction.scenes字段表示该控件只在特定的部署平台下展示。

hidden

此描述符用于隐藏并自动生成的表单的字段：如果基于CRD自动生成的表单字段有些不需要呈现给用户可以配置hidden隐藏。

配置示例

```
- description: ""
    displayName: ""
    path: image
    x-descriptors:
      - "urn:alm:descriptor:com.huawei.ui:hidden"
```

使用说明：

1.可以隐藏CRD描述的字段（如果是父目录的则叶子节点也会隐藏）。

2.该字段在UI表单上隐藏，但是yaml编辑器仍然是可以修改。

3.5.2.2.2 statusDescriptors

statusDescriptor用于描述自定义资源中状态字段的属性，其配置在csd.yaml文件的status字段。statusDescriptors的结构与specDescriptors相似，包括相同的字段。

1. text

x-descriptors: 文本形式公开实例的状态字段。

```
status:   [PATH_TO_THE_FIELD]: [FIELD_VALUE]
```

例子：

```
- description: 版本
  displayName: 版本
  path: version
  x-descriptors:
    - urn:alm:descriptor:text
```

2. link

此描述符用来公开实例的外部链接。它需要实例的状态块中的输出格式：

```
status:
   [PATH_TO_THE_FIELD]: [FIELD_VALUE]
```

例子：

```

- description: 开启公网访问后，可通过弹性公网IP访问，集群模式暂不支持
  displayName: CCE集群内访问地址
  path: publishedEndpoint
  restriction:
    scenes:
     - CCE
  x-descriptors:
    - urn:alm:descriptor:com.huawei.ui:link
```

实例status字段具体值信息如下:

```
status:
  publishedEndpoint: 'http://kafka-1wppvf-broker.default.svc.cluster'
```

3.5.3 表单控件配置组合示例

csd.yaml文件中配置添加如下配置：

```
# 配置表单控件
descriptors:
  spec:
    - displayName: Image
      description: The docker image name and version of Portworx Enterprise.
      path: image
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:text'
    - displayName: Size
      description: The desired number of member Pods for the etcd cluster.
      path: size
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:podCount'
```

以v1版本的crd.yaml为例：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
```



```
metadata:
  name: helmreleases.helm.osc.huawei.com
spec:
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        description: HelmRelease is the Schema for the helmreleases API
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            properties:
              image:
                type: string
              size:
                type: string
```

配置表单控件后，创建实例时，即可通过表单进行实例创建。

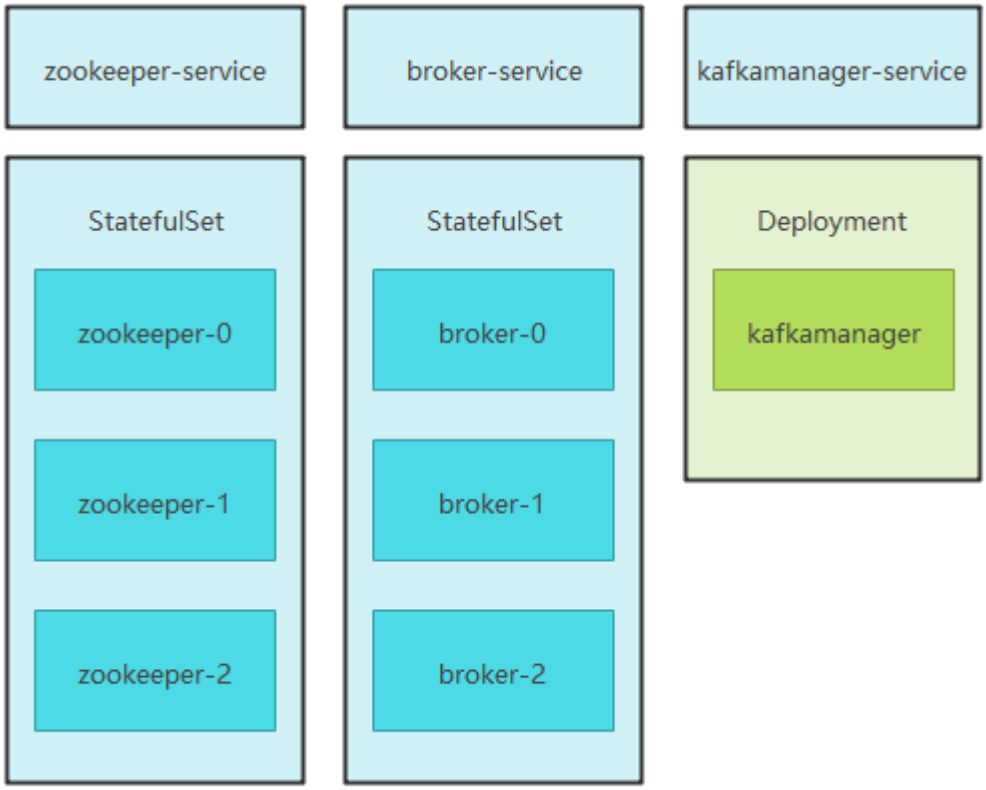
4 附录

4.1 如何从零开始开发 Operator

4.1.1 开发 Operator

开发原理

在本章节以Kafka为例，构建Kafka-Operator进行Kafka实例的管理，同时Kafka实例将以三节点集群的形式对外提供服务。对于Kafka实例/集群而言，需要依托于zookeeper进行构建，zookeeper用作Kafka集群的元数据管理，每个Kafka实例在各节点上会有一个zookeeper实例与其对应，存储broker内的消息、topic等数据。对于整体架构而言，先启动一个zookeeper三节点集群作为kafka启动的基础条件，然后启动Kafka三个节点实例（即broker实例）进行服务的提供，同时启动一个KafkaManager用作可视化管理。整体结构如下：



4.1.1.1 安装 Kubebuilder

环境准备

- 已安装go v1.16+。
- 已安装make。
- 已安装docker 17.03+。
- 已安装Kubernetes v1.15.0+ 。
- 已安装kustomize，安装方法请参见**安装 kustomize**。
- 已安装controller-gen，安装方法请参见**安装controller-gen**。

安装 Kubebuilder

Kubebuilder是基于custom resource definitions(CRDs)构建Kubernetes APIs的框架，有如下两种安装方法 。

- 安装方法一：根据kubebuilder官网提供的go语言版本和节点系统架构下载。

```
# go env GOOS -- 获取操作系统类型，例如：linux等
# go env GOARCH -- 获取系统架构，例如：arm或amd64等
$ curl -L -o kubebuilder https://go.kubebuilder.io/dl/latest/$(go env GOOS)/$(go env GOARCH)
$ chmod +x kubebuilder && mv kubebuilder /usr/local/bin/
```

- 安装方法二：避免因证书导致无法通过curl进行下载。

```
# 通过git clone及本地编译的方法安装kubebuilder可以避免因操作系统类型、系统架构输入错误导致
# kubebuilder二进制文件因操作系统和系统架构导致错误导致的无法使用。同时，通过git操作避免
# 在使用curl过程中需要对开发环境进行证书配置等操作
$ git clone https://github.com/kubernetes-sigs/kubebuilder.git
Cloning into 'kubebuilder'...
```

```
remote: Enumerating objects: 49241, done.
remote: Counting objects: 100% (2995/2995), done.
remote: Compressing objects: 100% (1087/1087), done.
remote: Total 49241 (delta 1897), reused 2679 (delta 1741), pack-reused 46246
Receiving objects: 100% (49241/49241), 63.87 MiB | 1.84 MiB/s, done.
Resolving deltas: 100% (25723/25723), done.
Updating files: 100% (1230/1230), done.

$ cd kubebuilder
$ git tag # 列出git目录下所有tag
release-0.1.6
tools-1.10.1
tools-1.11.0
v0.1.10
v0.1.11
v0.1.12
....
$ git checkout v3.2.0 # 目前最新tag为v3.2.0，可根据开发需求选择合适的版本
Note: switching to 'v3.2.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at b7a730c8 Merge pull request #2409 from camilamacedo86/release-3

# 检查是否切换到想要的版本
$ git branch
* (HEAD detached at v3.2.0)
  master

# 编译kubebuilder二进制文件
$ make build
go build -ldflags " -X main.kubeBuilderVersion=v3.2.0 -X main.goos=linux -X main.goarch=amd64 -X
main.gitCommit=b7a730c84495122a14a0faff95e9e9615fffbfc5 -X
main.buildDate=2021-12-14T06:30:08Z " -o bin/kubebuilder ./cmd
go: downloading github.com/gobuffalo/flect v0.2.3
go: downloading sigs.k8s.io/controller-tools v0.7.0
go: downloading sigs.k8s.io/kustomize/kyaml v0.10.21
go: downloading k8s.io/api v0.22.2
go: downloading k8s.io/apiextensions-apiserver v0.22.2

# 复制二进制文件到指定目录，方便后期使用
$ cp bin/kubebuilder /usr/local/bin/
```

4.1.1.2 构建 Operator

4.1.1.2.1 CRD 介绍

CRD(CustomResourceDefinition)是一种自定义的Kubernetes资源，在定义CRD后，可通过在集群内创建对应 CR(CustomResource)对应用进行统一管理。格式示例请参见**CRD典型格式**。

OpenAPI 校验

在定义CRD时，需同时定义基于OpenAPI的校验规则，其中包含创建CR时的字段及取值范围，以便校验用户创建的CR中字段值的合法性。使用Kubebuilder，可通过在API中定义Marker，自动生成spec.validation.openAPIV3Schema。

CRD 字段说明

- group，一般为组织名称，如：osctest。
- API版本，第一个版本一般为v1。
- scope，Namespaced级别，修改为Cluster级别，则集群内仅可以创建一个同名的CR。仅Cluster级别的operator可管理同级别CRD。
- API名称，假设应用名称为hwfka，则对应的多种名称格式如下：
  - API名称：kind: Hwfka，创建API的时候使用。
  - 列表形式：listKind: HwfkaList。
  - 复数形式：plural: hwfkas。
  - 单数形式：singular: hwfka。
  - 简称：shortNames: hfk。

构建 CR

通过CustomResource(CR)的方式创建新资源类型，在CR中为应用定义参数和状态属性。

参数

- size：应用实例包含的实例数量。
- image：应用的容器镜像地址。
- storage：应用数据存储相关配置。

状态

- phase：应用实例安装状态。
- server：应用访问地址。

以创建一个Kafka应用，指定其Pod数量为3为例，设计CR如下：

```
apiVersion: osctest.huawei.com/v1
kind: Hwfka
metadata:
  name: hwfka-sample
spec:
  size: 3
  image: swr.cn-north-7.myhuaweicloud.com/osc/kafka:v2.7.0
  storage:
    class: csi-disk
    accessModes: ReadWriteOnce
    size: 20Gi
    diskType: SSD
```

说明

- kind为CRD中定义的names.kind。
- metadata.name为CR名称，用户可自行更改。
- 该CR包含三个属性：size、image、storage，下文在定义API时需要为其定义Marker，以便自动生成OpenAPI schema。
- apiVersion由group、domain、version组成，group为osctest，domain为huawei.com，version为v1，与CRD中的定义对应，可根据需要修改。

4.1.1.2.2 创建 Operator 项目

背景

Operator是Kubernetes的扩展软件，通过定制资源管理应用和其他组件，实现一定自动运维能力。可以在不改动Kubernetes源码的情况下，通过一个或多个Operator来扩展集群能力，遵照自身业务需求、场景等灵活开发，节省运维成本。流程包括安装Kubernetes、构建Operator和实现Operator。

创建 Operator

本章节以应用名称为hwfka、取项目名称为hwfka-operator为例，说明如何创建Operator。

```
# 在 GOPATH 下新建工程目录
$ mkdir -p $GOPATH/src/hwfka-operator
$ cd $GOPATH/src/hwfka-operator
# 使用kubebuilder初始化脚手架框架，并设定域名为huawei.com
$ kubebuilder init --domain huawei.com
Writing kustomize manifests for you to edit...
Writing scaffold for you to edit...
Get controller runtime:
# 检查如果环境所需要的依赖包不满足kubebuilder要求，kubebuilder会下载相关依赖包，开发者可根据实际情况在项目的go.mod中更改依赖包
go get sigs.k8s.io/controller-runtime@v0.10.0
go: downloading sigs.k8s.io/controller-runtime v0.10.0
go: downloading k8s.io/client-go v0.22.1
go: downloading k8s.io/utils v0.0.0-20210802155522-efc7438f0176
go: downloading k8s.io/component-base v0.22.1
go: downloading k8s.io/apiextensions-apiserver v0.22.1
Update dependencies:
go mod tidy
go: downloading github.com/benbjohnson/clock v1.1.0
Next: define a resource with:
kubebuilder create api
```

创建Operator脚手架工程，目录结构如下：

```
hwfka-operator/
├── Dockerfile
├── Makefile
├── PROJECT
├── config
│   ├── default
│   │   └── ...
│   ├── manager
│   │   └── ...
│   ├── prometheus
│   │   └── monitor.yaml
│   └── rbac
│       └── ...
├── go.mod
├── go.sum
```

```
├── hack
│   └── boilerplate.go.txt
└── main.go
```

说明

- 参数domain一般为公司域名，如：huawei.com。
- main.go作为程序入口，初始化了Manager，由Manager来管理API和Controller。
- 初始化Manager时，可以指定参数，限制该Operator中所有Controller监听资源的namespace。

Operator 作用域

Operator的作用域分namespace级和cluster级，cluster级的operator可以监听和管理任意namespace的资源。使用**kubebuilder init**命令默认初始化cluster级的operator，仅支持cluster级的operator管理同级别的CRD。监听多个namespace，main.go修改代码如下：

```
// namespace 列表
namespaces := []string{"foo", "bar"}
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme:             scheme,
    MetricsBindAddress: metricsAddr,
    Port:               9443,
    LeaderElection:     enableLeaderElection,
    LeaderElectionID:   "f1c5ece8.huawei.com",
    NewCache:           cache.MultiNamespacedCacheBuilder(namespaces),  // 指定多个 namespace
})
```

说明

在修改了Operator作用域后，需同时修改RBAC（ config/rbac/... 路径下相关yaml文件 ），以授权Operator操作该namespace下的资源。

4.1.1.2.3 创建 API 和 Controller

根据以上设计的CR，在项目工程中创建对应的CRD类型：

```
$ kubebuilder create api --group osctest --version v1 --kind Hwfka
Create Resource [y/n]
y
Create Controller [y/n]
y
Writing kustomize manifests for you to edit...
Writing scaffold for you to edit...
api/v1/hwfka_types.go
controllers/hwfka_controller.go
Update dependencies:
go mod tidy
Running make:
make generate
go: creating new go.mod: module tmp
Downloading sigs.k8s.io/controller-tools/cmd/controller-gen@v0.7.0
go get: added sigs.k8s.io/controller-tools v0.7.0
/mnt/d/repository/Go/GOPATH/src/hwfka-operator/bin/controller-gen object:headerFile="hack/
boilerplate.go.txt" paths="./..."
Next: implement your new API and generate the manifests (e.g. CRDs,CRs) with:
make manifests
```

说明

通过kubebuilder create api命令，可以发现在重复下载controller-gen的二进制文件，包括后续使用Makefile去编译operator的二进制文件及打包镜像时，都可能会重复下载使用controller-gen二进制文件，建议提前安装controller-gen二进制，并修改Makefile中有关这个命令为使用系统命令，而非每次通过go get下载并编译后使用。

该命令主要执行以下动作：

1. 在PROJECT文件中增加API资源声明。

```
domain: huawei.com
layout:
- go.kubebuilder.io/v3
projectName: hwfka-operator
repo: hwfka-operator
resources:
- api:
    crdVersion: v1
    namespaced: true
  controller: true
  domain: huawei.com
  group: osctest
  kind: Hwfka
  path: hwfka-operator/api/v1
  version: v1
version: "3"
```

2. 新增CRD及CR的描述文件。

```
hwfka-operator
└── config
    ├── crd
    │   ├── kustomization.yaml
    │   ├── kustomizeconfig.yaml
    │   └── patches
    │       ├── cainjection_in_hwfkas.yaml
    │       └── webhook_in_hwfkas.yaml
    ├── rbac # 增加文件
    │   ├── hwfka_editor_role.yaml
    │   └── hwfka_viewer_role.yaml
    └── samples
        └── osctest_v1_hwfka.yaml
```

3. 新增api目录，包含CRD的类型定义。

```
hwfka-operator
└── api
    └── v1
        ├── groupversion_info.go
        ├── hwfka_types.go
        └── zz_generated.deepcopy.go
```

4. 新增controllers目录，包含控制器的业务逻辑。

```
hwfka-operator
└── controllers
    ├── hwfka_controller.go
    └── suite_test.go
```

5. 在main.go的Manager启动时，创建Controller。

```
if err = (&controllers.HwfkaReconciler{
    Client: mgr.GetClient(),
    Scheme: mgr.GetScheme(),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller", "Hwfka")
    os.Exit(1)
}
```

4.1.1.3 实现 Operator

4.1.1.3.1 定义 API

在设计的CR spec中，包含size、image、storage属性，因此需要修改api/v1/hwfka_types.go中HwfkaSpec和HwfkaStatus部分，为应用定义参数和状态属性。

```
package v1

import (
```

```
    corev1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/api/resource"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

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

    // +kubebuilder:validation:Minimum=1
    // +kubebuilder:validation:Maximum=3
    // +kubebuilder:validation:ExclusiveMaximum=false  // 包含最大值
    Size    int32        `json:"size"`    // Hwfka 应用包含的 broker 数量
    Image   string       `json:"image"`   // Hwfka 镜像地址
    Storage *StorageSpec `json:"storage"` // Hwfka 数据存储相关配置
}

type StorageSpec struct {
    Class       string                          `json:"class"`
    AccessModes corev1.PersistentVolumeAccessMode `json:"accessModes"`
    Size        resource.Quantity               `json:"size"`
    // 华为公有云 EVS 场景需额外指定 diskType, region, zone
    DiskType string `json:"diskType,omitempty"`  // omitempty 表示可以为空
    Region   string `json:"region,omitempty"`
    Zone     string `json:"zone,omitempty"`
}

// HwfkaStatus defines the observed state of Hwfka
type HwfkaStatus struct {
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    Phase  string `json:"phase,omitempty"`  // Hwfka 实例安装状态
    Server string `json:"server,omitempty"` // Hwfka 访问地址
}
```

使用Kubebuilder，可通过在API结构的属性上定义**Markers**，自动生成CRD中的spec.validation.openAPIV3Schema，即基于OpenAPI的校验规则，以便校验用户创建的CR中字段值的合法性。如上述设置size属性的最小值和最大值：

```
// +kubebuilder:validation:Minimum=1
// +kubebuilder:validation:Maximum=3
```

详细使用方法参见：**CRD Validation**。每次修改API定义后，需要执行命令自动重新生成代码和CRD：

```
$ make generate
$ make manifests
```

4.1.1.3.2 实现 Controller

Controller 实现背景

在创建Kafka实例时，Kafka-Operator需要创建的Kubernetes资源如下：

- 1个StatefulSet，包含3个Pod分别启动ZooKeeper；
- 1个Service，用来暴露ZooKeeper访问地址；
- 1个StatefulSet，包含3个Pod分别启动Hwfka broker；
- 1个Service，用来暴露Hwfka访问地址；
- 1个Deployment，包含1个Pod启动KafkaManager；
- 1个Service，用来暴露KafkaManager访问地址。

在创建API的时候，SDK已自动创建controllers/hwfka_controller.go，为SetupWithManager添加对Service、StatefulSet、Deployment的监听和管理。

```
import (
    "context"
    "fmt"

    "github.com/go-logr/logr"
    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/api/errors"
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/types"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/client"

    osctestv1 "hwfka-operator/api/v1"
)

func (r *HwfkaReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&osctestv1.Hwfka{}).
        Owns(&corev1.Service{}).
        Owns(&appsv1.StatefulSet{}).
        Owns(&appsv1.Deployment{}).
        Complete(r)
}
```

调度逻辑

对于Kubernetes集群内任何Hwfka或Service、StatefulSet、Deployment的变化，Controller都会监听到，并生成事件，触发Reconcile()方法。需要在Reconcile()方法中实现协调逻辑，创建Service、StatefulSet等资源，并更新应用实例状态，实现的过程中，可以参考：

1. **查询资源：**控制器使用controller-runtime库中的Client实现对Kubernetes资源的增查改删，示例代码参见：**example_test.go**。
2. **创建资源：**使用Go语言调用Kubernetes API创建资源，可参考**Kubernetes API Reference**。
3. **设置关联：**为创建的Kubernetes资源设置ownerReferences，以便其能在CR删除时被级联删除，可参考如下代码。

```
// 为指定命名空间和名称的 secret 设置 owner，返回 secret
func (k *K8sClient) SetSecretOwner(cr *oscv1.Kafka, secretName string) (*corev1.Secret, error) {
    ctx := context.Background()

    secret, err := k.GetSecretByNamespaceName(cr.Namespace, secretName)
    if err != nil {
        return nil, err
    }
    if secret.OwnerReferences == nil {
        patch := client.MergeFrom(secret.DeepCopy())
        secret.OwnerReferences = []metav1.OwnerReference{
            *metav1.NewControllerRef(cr, schema.GroupVersionKind{
                Group:   cr.GroupVersionKind().Group,
                Version: cr.GroupVersionKind().Version,
                Kind:    cr.Kind,
            }),
        }
        if err := k.Patch(ctx, secret, patch); err != nil {
            return nil, err
        }
    }
    return secret, nil
}
```

4. 如果一个动作的处理时间较长，为了避免Reconcile阻塞，需要使请求返回并重新排队，有四种方法：

```
// 请求成功，不再排队
return ctrl.Result{}, nil
// 请求失败，重新加入队列
return ctrl.Result{}, err
// 请求因某种原因需要重新加入队列
return ctrl.Result{Requeue: true}, nil
// 请求因某种原因，需要在指定时间后重新加入队列
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```

5. 当删除CR时，如果需要预先清理Kubernetes之外的资源，此时仅凭ownerReferences无法实现，可以利用finalizers特性。利用finalizers延时删除资源的方法请参见**Using Finalizers**。

   说明

   示例代码请参见**Controller实现**。

挂载存储

应用实例可以使用本地磁盘存储（HostPath），也可以使用网络存储或云存储（**推荐**），在Kubernetes中，通过PersistentVolume(PV)方式挂载存储，具体请参见**挂载存储** 。

云存储类型

云存储类型主要指标包括：

- IO读写时延：连续两次进行读写操作所需要的最小时间间隔。
- IOPS：每秒进行读写的操作次数。
- 吞吐量：每秒读取和写入的数据量。

说明

- 更多信息请参见**云硬盘EVS：磁盘类型及性能介绍**，云磁盘计费方式请参见**云硬盘计费说明**。
- 通过在集群中创建PVC的方式创建存储，创建方式请参见**创建存储**。

RBAC 权限管理

Operator使用基于角色的访问控制机制对Kubernetes中的资源进行访问，因此，要保证Operator正确的运行，需要使用管理员权限的账号对Operator进行授权，即预先创建对应的role和rolebinding。

在Reconcile()上添加markers，增加Operator对于Service、StatefulSet、Deployment、Pod的管理权限，SDK即可自动生成对应的role和rolebinding等资源描述文件。

```
//
+kubebuilder:rbac:groups=osctest.huawei.com,resources=hwfkas,verbs=get;list;watch;create;update;patch;del
ete
// +kubebuilder:rbac:groups=osctest.huawei.com,resources=hwfkas/status,verbs=get;update;patch
// +kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=apps,resources=statefulsets,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch
// +kubebuilder:rbac:groups=core,resources=persistentvolumeclaims,verbs=get;list;watch;update

func (r *HwfkaReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
```

```
    // ...
}
```

如果Operator为namespace级别的，则需要还添加namespace，如：

```
//
+kubebuilder:rbac:groups=osctest.huawei.com,namespace=example,resources=hwfkas,verbs=get;list;watch;cr
eate;update;patch;delete
// +kubebuilder:rbac:groups=osctest.huawei.com,namespace=example,resources=hwfkas/
status,verbs=get;update;patch
```

4.1.1.3.3 生成代码和资源描述文件

修改api/v1/memcached_types.go或controller中的markers之后，需要重新生成代码和资源描述文件。

```
# 生成 api/v1/zz_generated.deepcopy.go
make generate

# 生成 config/crd/bases 和 config/rbac/role.yaml
make manifests
```

说明

该步骤将根据API定义自动生成CRD和RBAC。

4.1.2 制作镜像

4.1.2.1 制作实例镜像

下载依赖包

1. 下载JRE：**https://download.java.net/openjdk/jdk11/ri/openjdk-11+28_linux-x64_bin.tar.gz**。
2. 下载Apache Kafka二进制包：**https://archive.apache.org/dist/kafka/2.7.0/kafka_2.13-2.7.0.tgz**。
3. 下载Kafka监控指标上报库：**https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.15.0/jmx_prometheus_javaagent-0.15.0.jar**。

创建配置文件

1. 创建初始化ZooKeeper的配置启动参数脚本zkGenConfig.sh。

```
#!/bin/bash

ZK_CONF_DIR=${ZK_CONF_DIR:-"/opt/kafka_2.13-2.7.0/config"}
ZK_CONFIG_FILE="$ZK_CONF_DIR/zookeeper.properties"
ZK_DATA_DIR=${ZK_DATA_DIR:-"/var/lib/zookeeper/zk/${NAMESPACE}/${INSTANCE}/${POD_NAME}/
data"}
ID_FILE="$ZK_DATA_DIR/myid"
ZK_REPLICAS=${ZK_REPLICAS:-"3"}
ZK_CLIENT_PORT=${ZK_CLIENT_PORT:-2181}
ZK_SERVER_PORT=${ZK_SERVER_PORT:-2888}
ZK_ELECTION_PORT=${ZK_ELECTION_PORT:-3888}

# 1. 生成 myid
echo "Generate myid"

HOST=$(hostname -s)
DOMAIN=$(hostname -d)
```

```
if [[ $HOST =~ (.*)-([0-9]+)$ ]]; then
  NAME=${BASH_REMATCH[1]}
  ORD=${BASH_REMATCH[2]}
else
  echo "Failed to extract ordinal from hostname $HOST"
  exit 1
fi

MY_ID=$((ORD + 1))
echo "MY_ID=$MY_ID"

mkdir -p $ZK_DATA_DIR
rm -f $ID_FILE
if [ ! -f $ID_FILE ]; then
  echo $MY_ID >>$ID_FILE
fi

# 2. 副本数量
if [ -z $ZK_REPLICAS ]; then
  echo "The environment variable ZK_REPLICAS is not exist."
  exit 1
fi

echo "ZK_REPLICAS=$ZK_REPLICAS"

# 3. 创建配置文件
rm -f $ZK_CONFIG_FILE
echo "tickTime=2000" >>$ZK_CONFIG_FILE
echo "initLimit=5" >>$ZK_CONFIG_FILE
echo "syncLimit=2" >>$ZK_CONFIG_FILE
echo "dataDir=$ZK_DATA_DIR" >>$ZK_CONFIG_FILE
echo "clientPort=$ZK_CLIENT_PORT" >>$ZK_CONFIG_FILE
for ((i = 1; i <= $ZK_REPLICAS; i++)); do
  echo "server.$i=$NAME-$((i - 1)).$DOMAIN:$ZK_SERVER_PORT:$ZK_ELECTION_PORT" >>
$ZK_CONFIG_FILE
done
```

2. 创建监控指标上报配置文件metrics-config.yaml。

```
startDelaySeconds: 0
lowercaseOutputName: false
lowercaseOutputLabelNames: false
rules:
  - pattern: kafka.controller<type=KafkaController, name=GlobalTopicCount><>Value
    name: kafka_controller_KafkaController_Value
    type: GAUGE
    labels:
      name: GlobalTopicCount
  - pattern: kafka.controller<type=KafkaController, name=GlobalPartitionCount><>Value
    name: kafka_controller_KafkaController_Value
    type: GAUGE
    labels:
      name: GlobalPartitionCount
  - pattern: kafka.server<type=ReplicaManager, name=LeaderCount><>Value
    name: kafka_server_ReplicaManager_Value
    type: GAUGE
    labels:
      name: LeaderCount
  - pattern: kafka.server<type=ReplicaManager, name=PartitionCount><>Value
    name: kafka_server_ReplicaManager_Value
    type: GAUGE
    labels:
      name: PartitionCount
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: BytesInPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesOutPerSec><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
```

```
    labels:
      name: BytesOutPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: MessagesInPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalProduceRequestsPerSec><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: TotalProduceRequestsPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalFetchRequestsPerSec><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: TotalFetchRequestsPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: BytesInPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesOutPerSec><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: BytesOutPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: MessagesInPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics,
name=TotalProduceRequestsPerSec><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: TotalProduceRequestsPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalFetchRequestsPerSec><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: TotalFetchRequestsPerSec
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec, topic=(.+)><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: BytesInPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesOutPerSec, topic=(.+)><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: BytesOutPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec, topic=(.+)><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: MessagesInPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalProduceRequestsPerSec,
topic=(.+)><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: TotalProduceRequestsPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalFetchRequestsPerSec,
```

```
topic=(.+)><>Count
    name: kafka_server_BrokerTopicMetrics_Count
    type: GAUGE
    labels:
      name: TotalFetchRequestsPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesInPerSec, topic=(.+)><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: BytesInPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=BytesOutPerSec,
topic=(.+)><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: BytesOutPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=MessagesInPerSec,
topic=(.+)><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: MessagesInPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalProduceRequestsPerSec,
topic=(.+)><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: TotalProduceRequestsPerSec
      topic: "$1"
  - pattern: kafka.server<type=BrokerTopicMetrics, name=TotalFetchRequestsPerSec,
topic=(.+)><>OneMinuteRate
    name: kafka_server_BrokerTopicMetrics_OneMinuteRate
    type: GAUGE
    labels:
      name: TotalFetchRequestsPerSec
      topic: "$1"
  - pattern: kafka.log<type=Log, name=(.+), topic=(.+), partition=(.+)><>Value
    name: kafka_log_Log_Value
    type: GAUGE
    labels:
      name: "$1"
      topic: "$2"
      partition: "$3"
  - pattern: java.lang<type=OperatingSystem><>SystemCpuLoad
    name: os_SystemCpuLoad
    type: GAUGE
    valueFactor: 100
  - pattern: java.lang<type=OperatingSystem><>ProcessCpuLoad
    name: os_ProcessCpuLoad
    type: GAUGE
    valueFactor: 100
  - pattern: java.lang<type=OperatingSystem><>TotalPhysicalMemorySize
    name: os_TotalPhysicalMemorySize
    type: GAUGE
  - pattern: java.lang<type=OperatingSystem><>FreePhysicalMemorySize
    name: os_FreePhysicalMemorySize
    type: GAUGE
```

3. 创建Dockerfile。

```
FROM centos:latest

ENV PATH="/opt/jre-11.0.11/bin:${PATH}" \
    KAFKA_HOME="/opt/kafka_2.13-2.7.0"

ADD openjdk-11+28_linux-x64_bin.tar.gz /opt/
ADD kafka_2.13-2.7.0.tgz /opt/
```



```
ADD zkGenConfig.sh ${KAFKA_HOME}/bin
ADD jmx_prometheus_javaagent-0.15.0.jar ${KAFKA_HOME}/libs
ADD metrics-config.yaml ${KAFKA_HOME}/config

RUN ln -s ${KAFKA_HOME} /opt/kafka \
    && chmod u+x ${KAFKA_HOME}/bin/zkGenConfig.sh \
    && sed -i "/kafka-run-class.sh/i\\export JMX_PORT=9999" ${KAFKA_HOME}/bin/kafka-server-start.sh \
    && sed -i "/kafka-run-class.sh/i\\export KAFKA_OPTS=\"\${KAFKA_OPTS} -javaagent:\$base_dir\/\.\.\/libs/jmx_prometheus_javaagent-0.15.0.jar=9404:\$base_dir\/\.\.\/config\/metrics-config.yaml\"" ${KAFKA_HOME}/bin/kafka-server-start.sh

WORKDIR $KAFKA_HOME
```

制作容器镜像

构建容器镜像命令如下：

```
$ docker build -t kafka:v2.7.0 .
```

4.1.2.2 制作 Operator 镜像

1. 修改hwfka-operator/Dockerfile内容如下：

```
FROM centos:latest
COPY bin/manager /
RUN chmod ug+x /manager
WORKDIR /
```

2. 构建容器镜像命令如下：

```
$ make && make docker-build IMG=hwfka-operator:v0.0.1
```

4.1.3 制作服务包

4.1.3.1 创建服务包模板

通过OSC提供的oscctl工具创建服务包模板，oscctl工具使用详情请参见**oscctl工具使用**。

1. 使用oscctl工具创建服务包模板。

```
# 使用oscctl工具，创建服务包
$ oscctl create -o ./hwfka-operator-package
Successfully created original package to OSC package!
```

2. 检查服务包目录结构。

```
# 服务包的结构
$ tree hwfka-operator-package

hwfka-operator-package
├── lifecycle.yaml #【必选】生命周期文件
├── manifests      #【必选】资源合计
│   ├── hwfka_crd.yaml
│   └── hwfka_csd.yaml
│   └── vendor
└── metadata.yaml #【必选】元数据文件
```

4.1.3.2 修改服务包内容

1. 修改lifecycle.yaml文件。该文件包括OSC服务包的生命周期的相关信息，例如：安装方式、安装时需要的集群/命名空间级别权限、日志目录等。

```
install:
  clusterPermissions: # 集群级别权限，请按照Kubernetes RABC相关规范填写
  - rules:
    - apiGroups:
        - "*" # * 表示所有
```

```
    resources:
      - "*" # 监控所有资源
    verbs:
      - "*" # 拥有所有权限
  serviceAccountName: hwfka-operator-sa # 附有该权限的SA名称，用户deployments中指定运行operator会获得该权限
 # permissions: null # 命名空间级别权限，与clusterPermissions结构相似，请按照Kubernetes RABC相关规范填写
 deployments: # 实际运行operator deployments的具体执行内容，请参照Kubernetes Deployment资源规范书写
 - name: hwfka-operator
   spec:
     replicas: 1
     selector:
       matchLabels:
         operator: hwfka-operator
     strategy: {}
     template:
       metadata:
         labels:
           operator: hwfka-operator
       spec:
         # initContainers 如果有需要可添加
         containers:
           - name: hwfka-operator
             image: hwfka-operator:0.0.1  # 该镜像为制作Operator镜像章节生成的镜像，请修改为实际的镜像地址
             imagePullPolicy: Always
         serviceAccountName: hwfka-operator-sa
 strategy: deployment # 现在operator部署仅适用deployment资源进行部署

# 由于是第一次部署该服务，不涉及升级
# upgrade:
#   replaces: 0.0.1
#   skips:
#     - 0.0.2
#   skipRange: '>=4.1.0 < 4.1.2'
# 该结构体用于运维操作配置，可以不进行配置
# operations:
#   logpath: ""
```

2. 修改metadata.yaml文件。

```
name: "hwfka-operator"
version: "0.0.1"
source: ISV
```

3. 修改manifests/hwfka_crd.yaml文件。该文件为描述operator使用到的自定义资源（CRD），可以直接复用hwfka-operator/config/crd/bases/osctest.huawei.com_hwfkas.yaml文件。因为该文件由Kubernetes社区提供的controller-gen及kustomize生成校验，所以可以直接作为标准CRD资源用户集群内CRD部署。

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
 annotations:
   controller-gen.kubebuilder.io/version: v0.7.0
 creationTimestamp: null
 name: hwfkas.osctest.huawei.com
spec:
 group: osctest.huawei.com
 names:
   kind: Hwfka
   listKind: HwfkaList
   plural: hwfkas
   singular: hwfka
 scope: Namespaced
 versions:
 - name: v1
   schema:
```



```
    openAPIV3Schema:
      description: Hwfka is the Schema for the hwfkas API
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this representation
            of an object. Servers should convert recognized schemas to the latest
            internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/
contributors/devel/sig-architecture/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/
community/contributors/devel/sig-architecture/api-conventions.md#types-kinds'
          type: string
        metadata:
          type: object
        spec:
          description: HwfkaSpec defines the desired state of Hwfka
          properties:
            image:
              type: string
            size:
              description: 包含最大值
              format: int32
              maximum: 3
              minimum: 1
              type: integer
            storage:
              properties:
                accessModes:
                  type: string
                class:
                  type: string
                diskType:
                  description: 华为公有云 EVS 场景需额外指定 diskType, region, zone
                  type: string
                region:
                  type: string
                size:
                  anyOf:
                  - type: integer
                  - type: string
                  pattern: ^(\+|-)?(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))(([KMGTPE]i)|[numkMGTPE]|([eE](\+|-)?
(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))))?$
                  x-kubernetes-int-or-string: true
                zone:
                  type: string
              required:
              - accessModes
              - class
              - size
              type: object
          required:
          - image
          - size
          - storage
          type: object
        status:
          description: HwfkaStatus defines the observed state of Hwfka
          properties:
            phase:
              type: string
            server:
              type: string
          type: object
      type: object
    served: true
    storage: true
```

```
    subresources:
      status: {}
status:
  acceptedNames:
    kind: ""
    plural: ""
  conditions: []
  storedVersions: []
```

4. 修改csd.yaml文件。

```
role: serviceEntity

defaultConfiguration: |-
  {
      "apiVersion": "hwfka.osc/v1",
      "kind": "Hwfka",
      "spec": {
        "image": "swr.cn-north-7.myhuaweicloud.com/osc-official/hwfka:0.0.1", # 该镜像为制作实例镜像章节生成的镜像，请修改为实际的镜像地址
        "size": 2
      }
  }
```

4.1.3.3 生成服务包

最终的服务目录结构如下：

```
$ tree hwfka-operator-package

hwfka-operator-package
├── lifecycle.yaml
├── manifests
│   ├── hwfka_crd.yaml
│   └── hwfka_csd.yaml
│   └── vendor
└── metadata.yaml
```

打包服务包命令如下：

```
$ tar -zcvf hwfka-operator-package.tgz hwfka-operator-package
```

4.2 OSC 扩展说明

4.2.1 oscctl 工具使用

oscctl是一款用于生成华为云OSC服务包脚手架或者将开源Helm或Operator服务包转换为OSC服务包的命令行工具。

工具安装

下载链接：**https://oscctl.obs.cn-east-3.myhuaweicloud.com/oscctl-22.6.23.tar.gz**

将下载后的oscctl工具安装包解压至本地后，将二进制文件oscctl复制到系统PATH路径下，使用下面的命令获取帮助信息：

```
$ oscctl --help
oscctl is a command-line tool that facilitates working with OSC packages, it can be used to
create a new OSC package scaffold from scratch or convert preexisting Helm or Operator package to OSC package.

Usage:
  oscctl [command]
```

```
Available Commands:
  convert     Converts a preexisting Helm or Operator package to an OSC package
  create      Creates a new OSC package scaffold from scratch
  help        Help about any command

Flags:
  -h, --help   help for oscctl

Use "oscctl [command] --help" for more information about a command.
```

创建服务包模板

在oscctl命令后添加子命令create用以创建华为云OSC服务包脚手架文件，使用--help可查看其支持的flag选项。

```
$ oscctl create --help
Creates a new OSC package scaffold from scratch, the scaffold will have a file structure and content
that satisfies OSC service package specification. Developers can modify the file content of the scaffold
to construct a package that's ready to use.

Usage:
  oscctl create [flags]

Flags:
  -h, --help            help for create
  -o, --output string   specify path to store the created or converted OSC package along with the package
name
```

示例：

```
$ oscctl create --output "./NewOSCPackage"
```

此命令将在工作目录下创建文件夹NewOSCPackage，其中包含OSC服务包必要的文件。

服务包转换

在oscctl命令后添加子命令convert用以将开源helm或operator服务包，使用--help可查看其支持的flag选项。

```
$ oscctl convert --help
Converts a preexisting Helm or Operator package to an OSC package, oscctl is able to distinguish helm and
operator package by detecting a Chart.yaml file in Helm package or a file with .package.yaml suffix in
Operator package

Usage:
  oscctl convert [flags]

Flags:
  -h, --help            help for convert
  -i, --input string    specify path of the original package.
  -o, --output string   specify path to store the created or converted OSC package along with the package
name
```

示例：

```
$ oscctl convert --input "./KafkaPackage.tgz" --output "./OSCPackage"
$ oscctl convert --input "./KafkaPackage" --output "./OSCPackage.zip"
$ oscctl convert --input "./KafkaPackage.tgz" --output "./OSCPackage.tgz"
```

上述命令将helm包KafkaPackage转换为OSC服务包，--input和--output均可为输入目录或者压缩包，oscctl会根据参数是否存在文件扩展名及扩展名类型自动判断并进行转换。现支持tgz和zip类型的压缩包输入和输出。

说明

oscctl转换工具，可以帮助已经拥有helm或者operator-framework类型的服务包，快速转换为OSC格式，从而对接运维和表单控件能力，对接能力参考**对接运维能力(可选)**和**配置表单控件**。如无此诉求，可直接将原生服务包上传验证即可。

4.3 Operator 代码示例

4.3.1 CRD 典型格式

以v1版本为例：

```
apiVersion: apiextensions.k8s.io/v1
# 创建 CRD
kind: CustomResourceDefinition
metadata:
  # 名称必须采用复数 <plural-name>.<group>.<domain> 格式
  name: memcacheds.osc.huawei.com
spec:
  group: osc.huawei.com
  names:
    # 可创建的对象类型
    kind: Memcached
    listKind: MemcachedList
    # 复数名称
    plural: memcacheds
    # CLI 别名, 显示的单数名称
    singular: memcached
    # CLI 中使用的简称
    shortNames:
    - mc
  # 可用范围: Namespaced or Cluster
  scope: Namespaced
  version: v1
  # 可定义多个版本
  versions:
  - name: v1alpha1
    served: false  # 需要卸载该版本时，先标记为 non-serving
    storage: false  # 再将 storage 设置到可用的版本
  - name: v1
    served: true
    storage: true
  # 基于 openAPIV3Schema 的校验规则
  validation:
    openAPIV3Schema:
      description: Memcached is the Schema for the Memcacheds API
      properties:
        # ...
      type: object
```

在Kubernetes中创建CRD时，将自动创建其对应的Kubernetes API，为RESTful endpoint形式，在namespace或cluster范围内可以访问其进行CRUD操作。

其格式如下：

```
/apis/<spec:group>/<spec:version>/<scope>/*/<names-plural>/...
```

示例：

```
/apis/osc.huawei.com/v1/namespaces/*/memcacheds/...
```

说明

*代表所有namespace，可以使用指定namespace名称查询特定namespace下的CR。

4.3.2 Controller 实现

修改controllers/hwfka_controller.go文件为如下内容：

```
func (r *HwfkaReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    log := r.Log.WithValues("hwfka", req.NamespacedName)

    // 获取 Hwfka CR
    instance := &osctestv1.Hwfka{}
    if err := r.Get(ctx, req.NamespacedName, instance); err != nil {
        log.Error(err, "Failed to get Kafka")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    for _, component := range components {
        if result, err, requeue := r.checkAndCreateResource(instance, component, ResourceService); requeue {
            return result, err
        }
    }

    // 检查并创建 ZooKeeper StatefulSet 资源
    if result, err, requeue := r.checkAndCreateResource(instance, ComponentZookeeper, ResourceStatefulSet);
requeue {
        return result, err
    }
    // 检查 ZooKeeper StatefulSet 是否 Ready
    zkSts := &appsv1.StatefulSet{}
    if err := r.Get(ctx, types.NamespacedName{Name: fmt.Sprintf("%v-%v", instance.Name,
ComponentZookeeper), Namespace: instance.Namespace}, zkSts); err != nil {
        log.Error(err, "Failed to get resource", "component", ComponentZookeeper, "resource",
ResourceStatefulSet)
        return ctrl.Result{}, err
    }
    if zkSts.Status.ReadyReplicas < zkSts.Status.Replicas/2+1 {
        return ctrl.Result{Requeue: true}, nil
    }

    // 检查并创建 Kafka StatefulSet 资源
    if result, err, requeue := r.checkAndCreateResource(instance, ComponentBroker, ResourceStatefulSet);
requeue {
        return result, err
    }

    // 更新实例 Status
    if len(instance.Status.Phase) == 0 || len(instance.Status.Server) == 0 {
        instance.Status.Phase = Available
        instance.Status.Server = fmt.Sprintf("%v-%v.%v.svc.cluster.local:%v", instance.Name,
ComponentBroker, instance.Namespace, BootstrapServerPort)
        err := r.Status().Update(ctx, instance)
        if err != nil {
            log.Error(err, "Failed to update instance status")
            return ctrl.Result{}, err
        }
    }
    return ctrl.Result{}, nil
}

func (r *HwfkaReconciler) checkAndCreateResource(instance *osctestv1.Hwfka, component string, resource
string) (ctrl.Result, error, bool) {
    ctx := context.Background()
    log := r.Log.WithValues("namespace", instance.Namespace, "instance name", instance.Name)

        var rs client.Object
    switch resource {
    case ResourceService:
        rs = &corev1.Service{}
    case ResourceStatefulSet:
        rs = &appsv1.StatefulSet{}
    case ResourceDeployment:
        rs = &appsv1.Deployment{}
```

```
    }
    // 检查并创建资源
    if err := r.Get(ctx, types.NamespacedName{Name: fmt.Sprintf("%v-%v", instance.Name, component),
Namespace: instance.Namespace}, rs); err != nil {
        if errors.IsNotFound(err) {
            // 创建资源
            var obj metav1.Object
            switch resource {
            case ResourceService:
                obj = NewServiceForComponent(instance, component)
            case ResourceStatefulSet:
                if component == ComponentZookeeper {
                    obj = NewStatefulSetForZooKeeper(instance)
                } else {
                    obj = NewStatefulSetForKafka(instance)
                }
            case ResourceDeployment:
                rs = &appsv1.Deployment{}
            }
            // 设置其 owner 为实例 CR
            if err = ctrl.SetControllerReference(instance, obj, r.Scheme); err != nil {
                log.Error(err, "Failed to set resource owner to Instance", "component", component, "resource",
resource)
                return ctrl.Result{}, err, true
            }
            log.Info("Creating a new resource", "component", component, "resource", resource)
                    if err = r.Create(ctx, obj.(client.Object)); err != nil {
                log.Error(err, "Failed to create new resource", "component", component, "resource", resource)
                return ctrl.Result{}, err, true
            }
            return ctrl.Result{Requeue: true}, nil, true
        }

        log.Error(err, "Failed to get resource", "component", component, "resource", resource)
        return ctrl.Result{}, err, true
    }
    return ctrl.Result{}, nil, false
}
```

创建controllers/constants.go文件，用来统一管理常量。

```
package controllers

const (
    // 配置 label
    LabelProvider  = "osctest.huawei.com/service_provider"
    LabelInstance  = "osctest.huawei.com/instance"
    LabelComponent = "osctest.huawei.com/component"

    ProviderKafka         = "hwfka"
    ComponentZookeeper    = "zookeeper"
    ComponentBroker       = "broker"
    ComponentKafkaManager = "hwfkamanager"

    ResourceService     = "service"
    ResourceStatefulSet = "statefulset"
    ResourceDeployment  = "deployment"

    // 配置 service 的端口
    ClientPortName          = "client"
    ZkClientPort            = 2181
    ServerPortName          = "server"
    ZkServerPort            = 2888
    LeaderElectionPortName  = "leader-election"
    ZkLeaderElectionPort    = 3888
    BootstrapServerPortName = "bootstrapserver"
    BootstrapServerPort     = 9092
    ExporterPortName        = "exporter"
    ExporterPort            = 9404
```



```
    // 环境变量
    EnvNamespace      = "NAMESPACE"
    EnvInstance       = "INSTANCE"
    EnvPodUid         = "POD_UID"
    EnvPodName        = "POD_NAME"
    EnvPodIp          = "POD_IP"
    EnvZkReplicas     = "ZK_REPLICAS"
    EnvZkClientPort   = "ZK_CLIENT_PORT"
    EnvZkServerPort   = "ZK_SERVER_PORT"
    EnvZkElectionPort = "ZK_ELECTION_PORT"

    PodUidRef  = "metadata.uid"
    PodNameRef = "metadata.name"
    PodIpRef   = "status.podIP"

    // 配置健康检查
    InitialDelaySeconds = 10

    // 存储
    DataVolumeName  = "datadir"
    ZkDataPath      = "/var/lib/zookeeper"
    KafkaDataPath   = "/opt/kafka/data"
    StorageClassEVS = "csi-disk"
    StorageClassSFS = "csi-nas"
    EvsAnnotation   = "everest.io/disk-volume-type"
    RegionLabel     = "failure-domain.beta.kubernetes.io/region"
    ZoneLabel       = "failure-domain.beta.kubernetes.io/zone"

    // 实例安装状态
    Installing    string = "Installing"
    InstallFailed string = "InstallFailed"
    Available     string = "Available"
    Abnormal      string = "Abnormal"
    Deleting      string = "Deleting"
)
```

在controllers下创建controllers/resources.go文件，用来生成所需要的资源。在创建StatefulSet的时候，使用volumeClaimTemplates自动创建PVC。

```
package controllers

import (
    "fmt"
    "strconv"

    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime/schema"

    oscv1 "hwfka-operator/api/v1"
)

var components = []string{
    ComponentZookeeper,
    ComponentBroker,
    ComponentKafkaManager,
}

// 以 CR 的 name-zookeeper/namespace 创建 HeadlessService
func NewServiceForComponent(cr *oscv1.Hwfka, component string) metav1.Object {
    labels := GenerateLabels(cr.Name, component)
    svc := &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:      fmt.Sprintf("%v-%v", cr.Name, component),
            Namespace: cr.Namespace,
            Labels:    labels,
        },
        Spec: corev1.ServiceSpec{
            ClusterIP: "None",
```



```
            Ports:     []corev1.ServicePort{},
            // HeadlessService match Pod 的 Label
            Selector: labels,
        },
    }

    switch component {
    case ComponentZookeeper:
        svc.Spec.Ports = []corev1.ServicePort{{
            Port: ZkClientPort,
            Name: ClientPortName,
        }, {
            Port: ZkServerPort,
            Name: ServerPortName,
        }, {
            Port: ZkLeaderElectionPort,
            Name: LeaderElectionPortName,
        }}
    case "linux":
        fmt.Println("Linux.")
    default:
        fmt.Printf("unsupported component: %s\n", component)
    }
    return svc
}

func NewStatefulSetForZooKeeper(cr *oscv1.Hwfka) metav1.Object {
    labels := GenerateLabels(cr.Name, ComponentZookeeper)

    sts := &appsv1.StatefulSet{
        ObjectMeta: metav1.ObjectMeta{
            Name:      fmt.Sprintf("%v-%v", cr.Name, ComponentZookeeper),
            Namespace: cr.Namespace,
            Labels:    labels,
        },
        Spec: appsv1.StatefulSetSpec{
            ServiceName:         fmt.Sprintf("%v-%v", cr.Name, ComponentZookeeper),
            Replicas:            &(cr.Spec.Size),
            PodManagementPolicy: appsv1.ParallelPodManagement,
            UpdateStrategy: appsv1.StatefulSetUpdateStrategy{
                Type: appsv1.RollingUpdateStatefulSetStrategyType,
            },
            Selector: &metav1.LabelSelector{
                MatchLabels: labels,
            },
            Template: corev1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: labels,
                },
                Spec: corev1.PodSpec{
                    Containers: getZooKeeperContainer(cr),
                },
            },
            VolumeClaimTemplates: []corev1.PersistentVolumeClaim{
                volumeClaimTemplates(cr),
            },
        },
    }
    return sts
}

func getZooKeeperContainer(cr *oscv1.Hwfka) []corev1.Container {
    //probe := []string{"sh", "-c", "netstat -lnt | grep -q 2181"}
    return []corev1.Container{{
        Name:  ComponentZookeeper,
        Image: cr.Spec.Image,
        //ImagePullPolicy: corev1.PullIfNotPresent,
        ImagePullPolicy: corev1.PullAlways,
        Env:             getZooKeeperEnv(cr),
```



```
        Command: []string{
            "sh", "-c", "bin/zkGenConfig.sh && bin/zookeeper-server-start.sh config/zookeeper.properties",
        },
        Ports: []corev1.ContainerPort{
            {
                ContainerPort: ZkClientPort,
                Name:          ClientPortName,
            },
            {
                ContainerPort: ZkServerPort,
                Name:          ServerPortName,
            },
            {
                ContainerPort: ZkLeaderElectionPort,
                Name:          LeaderElectionPortName,
            },
        },
        VolumeMounts: []corev1.VolumeMount{
            {
                Name:      DataVolumeName,
                MountPath: ZkDataPath,
            },
        },
        /*LivenessProbe: &corev1.Probe{
            Handler: corev1.Handler{
                Exec: &corev1.ExecAction{Command: probe},
            },
            InitialDelaySeconds: InitialDelaySeconds,
        },
        ReadinessProbe: &corev1.Probe{
            Handler: corev1.Handler{
                Exec: &corev1.ExecAction{Command: probe},
            },
            InitialDelaySeconds: InitialDelaySeconds,
        },*/
    }}
}

func getZooKeeperEnv(cr *oscv1.Hwfka) []corev1.EnvVar {
    envVar := []corev1.EnvVar{
        {
            Name:  EnvZkReplicas,
            Value: strconv.FormatInt(int64(cr.Spec.Size), 10),
        },
        {
            Name:  EnvZkClientPort,
            Value: strconv.FormatInt(int64(ZkClientPort), 10),
        },
        {
            Name:  EnvZkServerPort,
            Value: strconv.FormatInt(int64(ZkServerPort), 10),
        },
        {
            Name:  EnvZkElectionPort,
            Value: strconv.FormatInt(int64(ZkLeaderElectionPort), 10),
        },
        {
            Name:  EnvNamespace,
            Value: cr.Namespace,
        },
        {
            Name:  EnvInstance,
            Value: cr.Name,
        },
        {
            Name: EnvPodUid,
            ValueFrom: &corev1.EnvVarSource{
                FieldRef: &corev1.ObjectFieldSelector{
                    FieldPath: PodUidRef,
```

```
            },
        },
    },
    {
        Name: EnvPodName,
        ValueFrom: &corev1.EnvVarSource{
            FieldRef: &corev1.ObjectFieldSelector{
                FieldPath: PodNameRef,
            },
        },
    },
    {
        Name: EnvPodIp,
        ValueFrom: &corev1.EnvVarSource{
            FieldRef: &corev1.ObjectFieldSelector{
                FieldPath: PodIpRef,
            },
        },
    },
}
return envVar
}

func volumeClaimTemplates(cr *oscv1.Hwfka) corev1.PersistentVolumeClaim {
    pvc := corev1.PersistentVolumeClaim{
        ObjectMeta: metav1.ObjectMeta{
            Namespace:   cr.Namespace,
            Name:        DataVolumeName,
            Annotations: map[string]string{},
            Labels:      map[string]string{},
            OwnerReferences: []metav1.OwnerReference{
                *metav1.NewControllerRef(cr, schema.GroupVersionKind{
                    Group:   cr.GroupVersionKind().Group,
                    Version: cr.GroupVersionKind().Version,
                    Kind:    cr.Kind,
                }),
            },
        },
        Spec: corev1.PersistentVolumeClaimSpec{
            StorageClassName: &cr.Spec.Storage.Class,
            AccessModes: []corev1.PersistentVolumeAccessMode{
                cr.Spec.Storage.AccessModes,
            },
            Resources: corev1.ResourceRequirements{
                Requests: corev1.ResourceList{
                    corev1.ResourceStorage: cr.Spec.Storage.Size,
                },
            },
        },
    }
    if cr.Spec.Storage.Class == StorageClassEVS {
        pvc.ObjectMeta.Annotations[EvsAnnotation] = cr.Spec.Storage.DiskType
        if len(cr.Spec.Storage.Region) != 0 && len(cr.Spec.Storage.Zone) != 0 {
            pvc.ObjectMeta.Labels[RegionLabel] = cr.Spec.Storage.Region
            pvc.ObjectMeta.Labels[ZoneLabel] = cr.Spec.Storage.Zone
        }
    }
    return pvc
}

func NewStatefulSetForKafka(cr *oscv1.Hwfka) metav1.Object {
    labels := GenerateLabels(cr.Name, ComponentBroker)
    sts := &appsv1.StatefulSet{
        ObjectMeta: metav1.ObjectMeta{
            Name:      fmt.Sprintf("%v-%v", cr.Name, ComponentBroker),
            Namespace: cr.Namespace,
            Labels:    labels,
        },
        Spec: appsv1.StatefulSetSpec{
```



```
            ServiceName:         fmt.Sprintf("%v-%v", cr.Name, ComponentBroker),
            Replicas:            &(cr.Spec.Size),
            PodManagementPolicy: appsv1.ParallelPodManagement,
            UpdateStrategy: appsv1.StatefulSetUpdateStrategy{
                Type: appsv1.RollingUpdateStatefulSetStrategyType,
            },
            // StatefulSet match Pod 的 Label
            Selector: &metav1.LabelSelector{
                MatchLabels: labels,
            },
            Template: corev1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: labels,
                },
                Spec: corev1.PodSpec{
                    Containers: getKafkaContainer(cr),
                },
            },
            VolumeClaimTemplates: []corev1.PersistentVolumeClaim{
                volumeClaimTemplates(cr),
            },
        },
    }
    return sts
}

func getKafkaContainer(cr *oscv1.Hwfka) []corev1.Container {
    zookeeperConnect := fmt.Sprintf("%v-%v:%v", cr.Name, ComponentZookeeper, ZkClientPort)
    logDir := fmt.Sprintf("/opt/kafka/data/kafka/%v/%v/", cr.Namespace, cr.Name)
    return []corev1.Container{{
        Name:            ComponentBroker,
        Image:           cr.Spec.Image,
        ImagePullPolicy: corev1.PullAlways,
        Ports: []corev1.ContainerPort{
            {
                Name:          BootstrapServerPortName,
                ContainerPort: BootstrapServerPort,
            },
            {
                Name:          ExporterPortName,
                ContainerPort: ExporterPort,
            },
        },
        VolumeMounts: []corev1.VolumeMount{
            {
                Name:      DataVolumeName,
                MountPath: KafkaDataPath,
            },
        },
        Command: []string{
            "sh", "-c",
            fmt.Sprintf("bin/kafka-server-start.sh config/server.properties --override broker.id=$(echo
$HOSTNAME | awk -F '-' '{print $NF}') --override zookeeper.connect=%v --override log.dir=%v
$POD_NAME", zookeeperConnect, logDir),
        },
        Env: getKafkaEnv(cr),
    }}
}

func getKafkaEnv(cr *oscv1.Hwfka) []corev1.EnvVar {
    envVar := []corev1.EnvVar{
        {
            Name: EnvPodUid,
            ValueFrom: &corev1.EnvVarSource{
                FieldRef: &corev1.ObjectFieldSelector{
                    FieldPath: PodUidRef,
                },
            },
        },
```



```
        {
            Name: EnvPodName,
            ValueFrom: &corev1.EnvVarSource{
                FieldRef: &corev1.ObjectFieldSelector{
                    FieldPath: PodNameRef,
                },
            },
        },
        {
            Name: EnvPodIp,
            ValueFrom: &corev1.EnvVarSource{
                FieldRef: &corev1.ObjectFieldSelector{
                    FieldPath: PodIpRef,
                },
            },
        },
        {
            Name:  EnvNamespace,
            Value: cr.Namespace,
        },
        {
            Name:  EnvInstance,
            Value: cr.Name,
        },
    }
    return envVar
}

// 以 CR 的 Name, 组件名称 组装 Label
func GenerateLabels(name, component string) map[string]string {
    labels := map[string]string{
        LabelProvider: ProviderKafka,
        LabelInstance: name,
    }

    if len(component) != 0 {
        labels[LabelComponent] = component
    }
    return labels
}
```

4.3.3 挂载存储

在Kubernetes中，通过PersistentVolume(PV)方式挂载存储，典型的步骤如下：

1. 创建PV，为集群提供存储服务。

```
apiVersion: v1
kind: PersistentVolume
metadata:
    name: mypv1
    annotations:
        # 访问控制: 使用 group ID(GID) 配置的存储仅允许 Pod 使用相同的 GID 进行写入。
        pv.beta.kubernetes.io/gid: "1234"
spec:
    capacity:
        storage: 1Gi  # 指定PV容量
    accessModes:
        # 访问模式：
        # ReadWriteOnce 以读写模式 mount 到单个节点；
        # ReadOnlyMany 以只读模式 mount 到多个节点；
        # ReadWriteMany 以读写模式 mount 到多个节点。
        - ReadWriteOnce
    # 回收策略：
    # Retain管理员手工回收。PV的STATUS保持Released无法被其他PVC申请，需要删除并重新创建PV；
    # Recycle清除PV中的数据，相当于rm -rf /thevolume/*。自动启动一个Pod删除PV中的数据，删除后PV
的STATUS恢复为Available；
    # Delete删除存储资源。
    persistentVolumeReclaimPolicy: Recycle
    storageClassName: nfs  # PV分类
```

```
    nfs:
        path: /nfsdata/pv1
        server: 192.168.56.105
```

2. 声明PersistentVolumeClaim(PVC)，指定应用实例需要的存储资源。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: mypvc1
spec:
    accessModes:
        - ReadWriteOnce  # 指定访问模式
    resources:
        requests:
            storage: 1Gi  # 指定容量
    storageClassName: nfs  # 指定PV分类
```

3. 在Pod中挂载PVC。

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: mypvc1
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

4.3.4 创建存储

通过在集群中创建PVC的方式创建存储，典型的创建方式如下：

- EVS云硬盘

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-auto-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SATA
  labels:
    failure-domain.beta.kubernetes.io/region: cn-north-4
    failure-domain.beta.kubernetes.io/zone: cn-north-4a
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

  - storageClassName：存储卷动态供应关联的k8s storage class名称；v1.15集群使用的csi关联的storage class名称是csi-disk。
  - accessModes: 指定读写模式，v1.15集群版本只支持非共享卷，此字段设置为ReadWriteOnce。
  - storage：存储容量，单位为Gi。

  另外需要指定：

  - everest.io/disk-volume-type：云硬盘类型，全大写；当前支持普通I/O（SATA）、高I/O（SAS）和超高I/O（SSD）。
  - failure-domain.beta.kubernetes.io/region：集群所在的region。Region对应的值请参见**地区和终端节点**。
  - failure-domain.beta.kubernetes.io/zone：创建云硬盘所在的可用区，必须和工作负载规划的可用区保持一致。zone对应的值请参见**地区和终端节点**。

- SFS文件存储：适用于媒体处理、内容管理、大数据分析、分析工作负载等多读多写的场景。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  pvc-sfs-auto-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas
```

- SFS Turbo极速文件存储：按需申请，快速供给，弹性扩展，适用于DevOps、容器微服务、企业办公等场景。

  针对CCE敏捷版，提供local pv的方式创建本地目录存储。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: paas.hw/local-pv
    volume.kubernetes.io/selected-node: 192.168.8.101 # 指定节点
    volume.kubernetes.io/targetPath: /opt/hhh    #对应创建中间件实例时用户输入的目录，如果不存在，local-pv-immediate会自动创建目录，
  name: datadir-kafka-event-broker-0
  namespace: kafka-alarm
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
  storageClassName: local-pv-immediate
  volumeMode: Filesystem
```

  说明

  - 使用华为云存储，需要CCE集群中预先安装everest插件。
  - 上述资源可通过Go语言在Reconcile协调逻辑中实现，具体请参见**Kubernetes API Reference**。
  - 更多类型的存储可参见**云容器引擎CCE：存储管理**。

4.4 安装 Controller

安装 kustomize

基于模板生成YAML文件，下载kustomize二进制压缩包：**https://github.com/kubernetes-sigs/kustomize/releases/download/kustomize/v3.8.1/kustomize_v3.8.1_linux_amd64.tar.gz**

```
# 解压并安装
$ tar zxvf kustomize_v3.8.1_linux_amd64.tar.gz
```

```
$ chmod +x kustomize
$ mv kustomize /usr/local/bin/
$ kustomize version
{Version:kustomize/v3.8.1 GitCommit:0b359d0ef0272e6545eda0e99aacd63aef99c4d0
BuildDate:2020-07-16T00:58:46Z GoOs:linux GoArch:amd64}
```

安装 controller-gen

构建控制器所使用的Go语言controller-runtime库。

1. 下载源码**controller-tools-0.3.0**。
2. 构建二进制并安装。

```
$ unzip controller-tools-0.3.0.zip
$ go build -a -o controller-gen cmd/controller-gen/main.go
$ mv controller-gen /usr/local/bin/
$ controller-gen --version
Version: (devel)
```